Lecture Notes of CSCI5610 Advanced Data Structures

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong

July 17, 2020

Contents

Lecture 1: Course Overview and Computation Models

A *data structure*, in general, stores a set of elements, and supports certain operations on those elements. From your undergraduate courses, you should have learned two types of use of data structures:

- They alone can be employed directly for information retrieval (e.g., "find all the people whose ages are equal to 25", or "report the number of people aged between 20 and 40").
- They serve as building bricks in implementing algorithms efficiently (e.g., Dijkstra's algorithm for finding shortest paths would be slow unless it uses an appropriate structure such as the priority queue).

This (graduate) course aims to deepen our knowledge of data structures. Specifically:

- We will study a number of new data structures for solving several important problems in computer science with strong performance guarantees (heuristic solutions, which perform well only on some inputs, may also be useful in some practical scenarios, but will not be of interest to us in this course).
- We will discuss a series of *techniques* for designing and analyzing data structures with non-trivial performance guarantees. Those techniques are *generic* in the sense that they are useful in a great variety of scenarios, and may very likely enable you to discover innovative structures in your own research.

Hopefully, with the above, you would be able to better appreciate the beauty of computer science at the end of the course.

The random access machine (RAM) model. Computer science is a subject under mathematics. From your undergraduate study, you should have learned that, before you can even start to analyze the "running time" of an algorithm, you need to first define a computation model properly.

Unless otherwise stated, we will be using the standard RAM model. In this model, the *memory* is an infinite sequence of *cells*, where each cell is a sequence of w bits for some integer w, and is indexed by an integer *address*. Each cell is also called a *word*; and accordingly, the parameter w is often referred to as the *word length*. The *CPU*, on the other hand, has a (constant) number of cells, each of which is called a *register*. The CPU can perform only the following *atomic* operations:

- Set a register to some constant, or to the content of another register.
- Compare two numbers in registers.
- Perform $+, -, \cdot, /$ on two numbers in registers.
- Shift the word in a register to the left (or right) by a certain number of bits.

- Perform the AND, OR, XOR on two registers.
- When an address x has been stored in a register, read the content of the memory cell at address x into a register, or conversely, write the content of a register into the memory cell.

The *time* (or *cost*) of an algorithm is measured by the number of atomic operations it performs. Note that the time is an integer.

A remark is in order about the word length w: it needs to be long enough to encode all the memory addresses! For example, if your algorithm uses n^2 memory cells for some integer n, then the word length will need to have at least $2 \log_2 n$ bits.

Dealing with real numbers. In the model defined earlier, the (memory/register) cells can only store integers. Next, we will slightly modify the model in order to deal with real values.

Note that simply "allowing" each cell to store a real value does not give us a satisfactory model because it creates several nasty issues. For example, how many bits would you use for a real value? In fact, even if the number of bits *were* infinite, still we would not be able to represent all the real values even in a short interval like $[0, 1]$ — the set of real values in the interval is *uncountably* infinite! If we cannot even specify the word length for a "real-valued" cell, how to properly define the atomic operations for performing shifts and the logic operations AND, OR, and XOR?

We can alleviate this issue by introducing the concept of *black box*. We still allow a (memory/register) cell c to store a real value x, but in this case, the algorithm is forbidden to look *inside* c, that is, the algorithm has no control over the representation of x. In other words, c is now a black box, holding the value x *precisely* (by magic).

A black box remains as a black box after computation. For example, suppose that two registers are both storing $\sqrt{2}$. We can calculate their product 2, but the product must still be understood as a real value (even though it is an integer). This is similar to the requirement in C++ that the product of two float numbers remains as a float number.

Now we can formally extend the RAM model as follows:

- Each cell can store either an integer or a real value.
- For operations $+, -, *, /$, if one of the operand numbers is a real value, the result is a real value.
- Among the atomic operations mentioned earlier, shifting, AND, OR, and XOR cannot be performed on registers that store real values.

We should note that, although mathematically sound, the resulting model — often referred to as the *real RAM* model — is not necessarily a realistic model in practice because no one has proven that it is polynomial-time equivalent to Turing machines (it would be surprising if it was). We must be very careful *not to abuse the power of real value computation*. For example, in the standard RAM model (with only integers), it is still open whether a polynomial time algorithm exists for the following problem:

Input: integers $x_1, x_2, ..., x_n$ and k
Output: whether $\sum_{i=1}^{n} \sqrt{x_i} \geq k$.

It is rather common, however, to see people design algorithms by assuming that the square root operator can be carried out in polynomial time — in that case, the above problem can obviously be settled in polynomial time under the real-RAM model! We will exercise caution in the algorithms we design in this course, and will inject a discussion whenever issues like the above arise.

Randomness. All the atomic operations are *deterministic* so far. In other words, our models so far do not permit *randomization*, which is important to certain algorithmic techniques (such as hashing).

To fix the issue, we introduce one more atomic operation for both the RAM and real-RAM models. This operation, named *RAND*, takes two non-negative integer parameters x and y, and returns an integer chosen uniformly at random from $[x, y]$. In other words, every integer in $[x, y]$ can be returned with probability $1/(y - x + 1)$. The values of x, y should be in $[0, 2^w - 1]$ because they each need to be encoded in a word.

Math conventions. We will assume that you are familiar with the notations of $O(.), \Omega(.)$, $\Theta(.)$, $o(.)$, and $\omega(.)$. We also use $\tilde{O}(f(n_1, n_2, ..., n_x))$ to denote the class of functions that are $O(f(n_1, n_2, ..., n_x) \cdot \text{polylog}(n_1 + n_2 + ... + n_x))$, namely, $\tilde{O}(.)$ hides a polylogarithmic factor. $\mathbb{R}$ denotes the set of real values, while $\mathbb{N}$ denotes the set of integers.

Lecture 2: The Binary Search Tree and the 2-3 Tree

This lecture will review the binary search tree (BST) which you should have learned from your undergraduate study. We will also talk about the 2-3 tree, which is a replacement of the BST that admits simpler analysis in proving certain properties. Both structures store a set S of elements conforming to a total order; for simplicity, we will assume that $S \subseteq \mathbb{R}$. Set $n = |S|$.

2.1 The binary search tree

2.1.1 The basics

A BST on S is a binary tree $\mathcal{T}$ satisfying the following properties:

- Every node u in $\mathcal{T}$ stores an element in S, which is denoted as the *key* of u. Conversely, every element in S is the key of exactly one node in $\mathcal{T}$. This means $\mathcal{T}$ has precisely n nodes.
- For every non-root node u with parent p:
  - if u is the left child of p, the keys stored in the subtree rooted at u are smaller than the key of p;
  - if u is the right child of p, the keys stored in the subtree rooted at u are larger than the key of p.

The space consumption of $\mathcal{T}$ is clearly $O(n)$ (cells). We say that $\mathcal{T}$ is *balanced* if its height is $O(\log n)$. Henceforth, all BSTs are balanced unless otherwise stated.

The BST is a versatile structure that supports a large number of operations on S efficiently:

- **Insertion/deletion**: an element can be added to S or removed from S in $O(\log n)$ time.
- **Predecessor/successor search**: the *predecessor* (or *successor*, resp.) of $q \in \mathbb{R}$ is the largest (or smallest, resp.) element in S that is at most (or at least, resp.) q. Given any q, its predecessor/successor in S can be found in $O(\log n)$ time.
- **Range reporting**: Given an interval $I = [x, y]$ where $x, y \in \mathbb{R}$, all the elements in $I \cap S$ can be reported in $O(\log n + k)$ time where $k = |I \cap S|$.
- **Find-min/find-max**: Report the smallest/largest element of S in $O(\log n)$ time.

The following are two more sophisticated operations that may not have been covered by your undergraduate courses:

- **Split**: Given a real value $x \in S$, split S into two sets: (i) S_1 which includes all the elements in S less than x, and (ii) $S_2 = S \setminus S$. Assuming a BST on S, this operation also produces a BST on S_1 and a BST on S_2. All these can be done in $O(\log n)$ time.

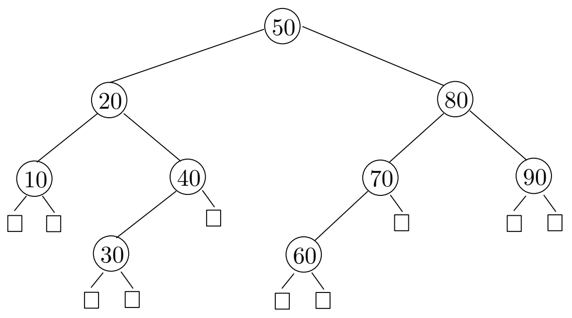


Figure 2.1: A BST (every square is a conceptual leaf)

- **Join**: Given two sets S_1 and S_2 of real values such that $x < y$ for any $x \in S_1, y \in S_2$, merge them into $S = S_1 \cup S_2$. Assuming a BST on each of S_1 and S_2, this operation also produces a BST on S. All these can be done in $O(\log n)$ time.

It is a bit complicated to implement the above two operations on the BST directly. This is the reason why we will talk about the 2-3 tree later (Section 2.2) which supports the two operations in an easier manner.

2.1.2 Slabs

Next we introduce the notion of *slab* which will appear very often in our discussion with BSTs.

Consider a BST $\mathcal{T}$ on S. Let u be a node in $\mathcal{T}$ for which either the left child or the right child does not exist (note: u is not necessarily a leaf node). In this case, we store a *nil pointer* for that missing child at u. It will be convenient to regard each nil pointer as a *conceptual leaf node*. You should not confuse this with a (genuine) leaf node z of $\mathcal{T}$ (every z has two conceptual leaf nodes as its "children"). The total number of conceptual leaf nodes is exactly $n + 1$. Henceforth, we will use the term *actual node* to refer to a "genuine" node in $\mathcal{T}$ that is not a conceptual leaf.

Given an actual/conceptual node u in $\mathcal{T}$, we now define its *slab*, denoted as $slab(u)$, as follows:

- If u is the root of $\mathcal{T}$, $slab(u) = (-\infty, \infty)$.
- Otherwise, let the parent of u be p, and x the key of p. Now, proceed with:
  - if u is the left child of p, then $slab(u) = slab(p) \cap (-\infty, x)$;
  - otherwise, $slab(u) = slab(p) \cap [x, \infty)$.

Note that $\mathcal{T}$ defines exactly $2n + 1$ slabs.

Example. Figure 2.1 shows a BST on the set $S = \{10, 20, ..., 90\}$. The slab of node 40 is $[20, 50)$, while that of its right conceptual leaf is $[40, 50)$. □

The following propositions are easy to verify:

Proposition 2.1. *For any two nodes u, v in $\mathcal{T}$ (which may be actual or conceptual):*

- *If u is an ancestor of v, then $slab(v)$ is covered by $slab(u)$;*
- *If neither of the two nodes is an ancestor of the other, then $slab(u)$ is disjoint with $slab(v)$.*

Proposition 2.2. *The slabs of the $n+1$ conceptual leaf nodes partition $\mathbb{R}$.*

Now we prove a very useful property:

Lemma 2.3. *Any interval $q = [x, y)$, where x and y take values from S, $-\infty$, or ∞, can be partitioned into $O(\log n)$ disjoint slabs.*

Proof. Let us first consider that q has the form $[x, \infty)$. We can collect a set Σ of disjoint slabs whose union equals q as follows:

1. Initially, $\Sigma = \emptyset$, and set u to the root of $\mathcal{T}$.
2. If the key of u equals x, then add the slab of the right child of u (the child may be conceptual) to Σ, and stop.
3. If the key of u is smaller than x, the set u to the right child of u, and repeat from 2.
4. Otherwise, add the slab of the right child of u (the child may be conceptual) to Σ. Then, set u to the left child of u, and repeat from 2.

Proving the lemma for general q is left to you as an exercise. □

Henceforth, we will refer to the slabs in the above lemma as the *canonical slabs* of q.

Example. In Figure 2.1, the interval $q = [30, 90)$ is partitioned by its canonical slabs $[30, 40)$, $[40, 50)$, $[50, 80)$, $[80, 90)$. □

2.1.3 Augmenting a BST

The power of the BST can be further enhanced by associating its nodes with additional information. For example, we can store at each node u of $\mathcal{T}$ a *count* which is the number of keys stored at the subtree rooted at u. The resulting structure will be referred to as a *count BST* henceforth.

The count BST supports all the operations in Section 2.1 with the same performance guarantees. In addition, it also supports:

- **Range counting**: Given an interval $q = [x, y]$ with $x, y \in \mathbb{R}$, report $|q \cap S|$, namely, the *number* of elements in S that are covered by q.

Corollary 2.4. *A count BST supports the range counting operation in $O(\log n)$ time.*

Proof. This is immediate from Lemma 2.3 (strictly speaking, the lemma requires the interval q to be open on the right; how would you deal with this subtlety?). □

2.2 The 2-3 tree

In a binary tree, every internal node has a *fanout* (i.e., number of child nodes) of either 1 or 2. We can relax this constraint by requiring only that each internal should have a constant fanout greater than 2. In this section, we will see a variant of the BST obtained following this idea. This variant, called the 2-3 tree, is a *replacement* of the BST in the sense that it can essentially attain all the performance guarantees of the BST, but interestingly, often admits simpler analysis. We will explain how to support the split and join operations in Section 2.1.1 on the 2-3 tree (these operations can also be supported by the BST, but in a more complicated manner).

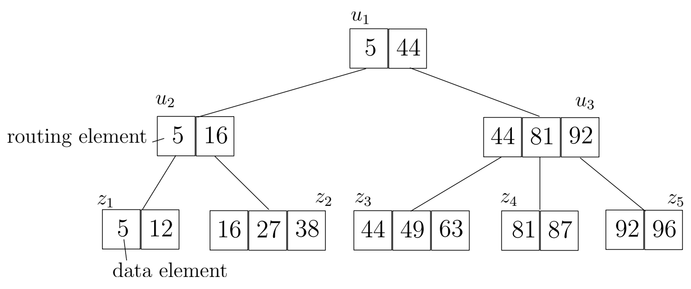


Figure 2.2: A 2-3 tree example

2.2.1 Description of the structure

A *2-3 tree* on a set S of n real values is a tree $\mathcal{T}$ satisfying the following conditions:

- All the leaf nodes are at the same level (recall that the *level* of a node is the number of edges on its path to the root of $\mathcal{T}$).
- Every internal node has 2 or 3 child nodes.
- Every leaf node u stores 2 or 3 elements in S. The only exception arises when $n = 1$, in which case $\mathcal{T}$ has a single leaf node that stores the only element in S.
- Every element in S is stored a single leaf node.
- If an internal node u has child nodes $v_1, ..., v_f$ where $f = 2$ or 3, it stores a *rounting element* e_i for every child v_i, which is the smallest element stored in the leaf nodes under v_i.
- If an internal node u has child nodes $v_1, ..., v_f$ ($f = 2$ or 3) with rounting elements $e_1, ..., e_f$, it must hold that all the elements stored at the leaf nodes under v_i are less than e_{i+1}, for each $i \in [1, f-1]$.

Note that an element in S may be stored multiple times in the tree (definitely once in some leaf, but perhaps also as a routing element in some internal nodes). The height of $\mathcal{T}$ is $O(\log n)$,

Example. Figure 2.2 shows a 2-3 tree on $S = \{5, 12, 16, 27, 38, 44, 49, 63, 81, 87, 92, 96\}$. Note that the leaf nodes of the tree present a sorted order of S. $\square$

As a remark, if you are familiar with the B-tree, you can understand the 2-3 tree as a special case with $B = 3$.

2.2.2 Handling overflows and underflows

Assume that $n \geq 2$ (i.e., ignoring the special case where $\mathcal{T}$ has only a single element). An internal or leaf node *overflows* if it contains 4 elements, or *underflows* if it contains only 1 element.

Treating overflows. We consider the case where the overflowing node u is not the root of $\mathcal{T}$ (the opposite case is left to you). Suppose that u contains elements $e_1, e_2, ..., e_4$ in ascending order, and that p is the parent of u. We create another node u', move e_3 and e_4 from u to u', and add a routing element e_3 to p for u'. See Figure 2.3. The steps so far take in constant time. Note that at this moment p may be overflowing, which is then treated in the same manner. Since the overflow may propagate all the way to the root, in the worst case we spend $O(\log n)$ time overall.

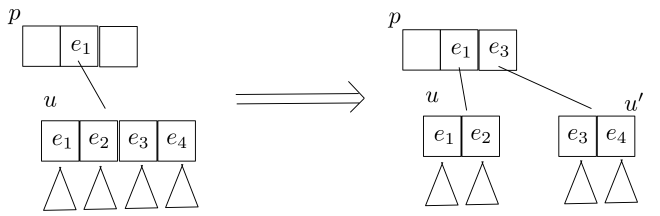
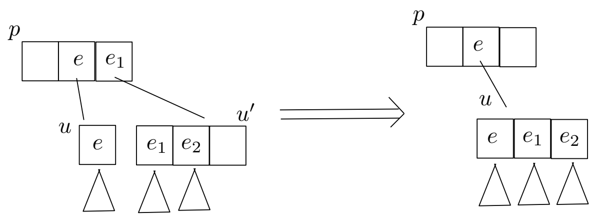
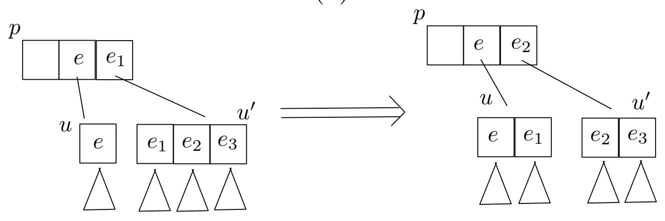


Figure 2.3: Treating an overflow



(a)



(b)

Figure 2.4: Treating an underflow

Treating underflows. We consider the case where the underflowing u is not the root of $\mathcal{T}$ (the opposite case is left to you). Suppose that the only element in u is e, and that p is the parent of u. Since p has at least two child nodes, u definitely has a *sibling* u'; due to symmetry, we will discuss only the case where u' is the right sibling of u. We proceed as follows:

- If u' has 2 elements, we move all the elements of u into u', delete u' from the tree, and remove the routing element in p for u'. See Figure 2.4(a). These steps require constant time. Note that p may be underflowing at this moment, which is treated in the same manner. Since the underflow may propagate all the way to the root, in the worst case we spend $O(\log n)$ time overall.

- If u' has 3 elements e_1, e_2, e_3, in constant time we move e_1 from u' into u, and modify the routing element in p for u'. See Figure 2.4(b). (Think: is there a chance the changes may propagate to the root?)

Remark. The underflow/overflow treating algorithms imply that an insertion or a deletion can be supported in $O(\log n)$ time (why?).

2.2.3 Splits and joins

Recall that our main purpose for discussing the 2-3 tree is to seek a (relatively) easy way to support the split and join operations, re-stated below:

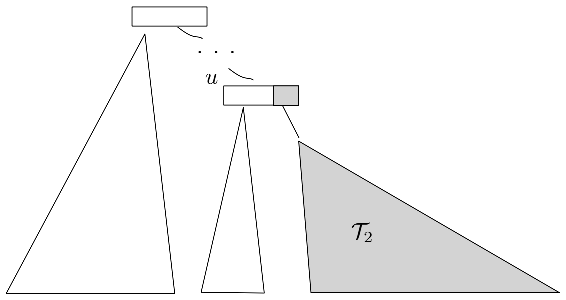


Figure 2.5: Join

- **Split**: Given a real value $x \in S$, split S into two sets: (i) S_1 which includes all the elements in S less than x, and (ii) $S_2 = S \setminus S$. Assuming a 2-3 tree on S, this operation should also produce a 2-3 tree on S_1 and a 2-3 tree on S_2. The time allowed is $O(\log n)$.
- **Join**: Given two sets S_1 and S_2 of real values such that $x < y$ for any $x \in S_1, y \in S_2$, merge them into $S = S_1 \cup S_2$. Assuming a 2-3 tree on each of S_1 and S_2, this operation should also produce a 2-3 tree on S. The time allowed is $O(\log n)$.

Join. Let us first deal with joins because the algorithm is simple, and will be leveraged to perform splits. Suppose that $\mathcal{T}_1$ and $\mathcal{T}_2$ are the 2-3 trees on S_1 and S_2, respectively. We can accomplish the join by adding one of the 2-3 trees as a subtree of the other. Specifically, denote by h_1 and h_2 the heights of $\mathcal{T}_1$ and $\mathcal{T}_2$, respectively. Due to symmetry, assume $h_1 \geq h_2$.

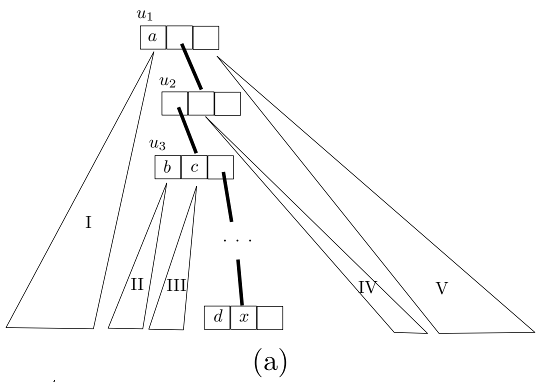
- If $h_1 = h_2$, just create a root u which has $\mathcal{T}_1$ as the left subtree and $\mathcal{T}_2$ as the right subtree.
- Otherwise, set $\ell = h_1 - h_2$. Let u be the level-$(\ell - 1)$ node on the rightmost path of $\mathcal{T}_1$. Add $\mathcal{T}_2$ as the *rightmost* subtree of u. See Figure 2.5. Note that this may trigger u to overflow, which is then treated in the way explained earlier.

Overall, a join can be performed in $O(1 + \ell)$ time, which is $O(\log n)$.

Split. Due to symmetry, we will explain only how to produce the 2-3 tree of S_1. Let $\mathcal{T}$ be the 2-3 tree on S. First, find the path Π in $\mathcal{T}$ from the root to the leaf containing the value x (used for splitting). It suffices to focus on the part of $\mathcal{T}$ that is "on the left" of Π. Interestingly, this part can be partitioned into a set Σ of $t = O(\log n)$ 2-3 trees. Before elaborating on this formally, let us first see an example.

Example. Consider Figure 2.6(a) where Π is indicated by the bold edges. We can ignore subtrees labeled as IV and V because they are "on the right" of Π. Now, let us focus on the part "on the left" of Π. At the root u_1 (level 0), Π descends from the 2nd routing element; the subtree labeled as I is added to Σ. At the level-1 node u_2, Π descends from the 1st routing element; no tree is added to Σ. At the level-2 node u_3, Π descends from the 3rd routing element; the 2-3 tree added to Σ has u_3 as the root, but only two subtrees labeled as II and III, respectively. The same idea applies to every level. At the leaf level, what is added to Σ is a 2-3 tree with only one node. Note how the 2-3 trees, shown in Figure 2.6(b), together cover all the elements of S_1. $\square$

Formally, we generate Σ by adding *at most* one 2-3 tree at each level ℓ. Let u be the level-ℓ node on Π. Denote by $e_1, ..., e_f$ the elements in u where $f = 2$ or 3.



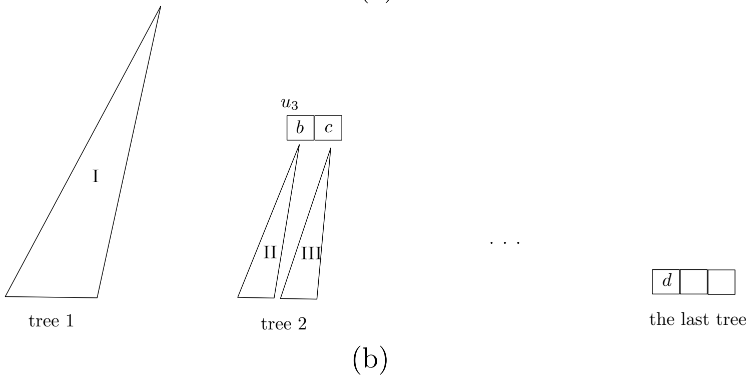


Figure 2.6: Split

- If Π descends from e_1, no tree is added to Σ.
- If Π descends from e_2, we add the subtree referenced by e_1 to Σ.
- If Π descends from e_3, we add the subtree rooted at u to Σ, after removing e_3 and its subtree.

Denote by $\mathcal{T}'_1, \mathcal{T}'_2, ..., \mathcal{T}'_t$ the 2-3 trees added by the above procedure in ascending order of level. Denote by h_i the height of $\mathcal{T}'_i$, $1 \le i \le t$. It must hold that:

$$h_1 \ge h_2 \ge ... \ge h_t.$$

We can now join all the trees together to obtain the 2-3 tree on S_1. To achieve $O(\log n)$ time, we must be careful with the order of joins. Specifically, we do the joins in *descending* order of i:

```
1. for $i = t$ to 2
2.     $\mathcal{T}'_{i-1} \leftarrow$ the join of $\mathcal{T}'_{i-1}$ and $\mathcal{T}'_i$
```

The final $\mathcal{T}'_1$ is the 2-3 tree on S_1. The cost of all the joins is:

$$\sum_{i=1}^{t} O(1 + h_{i-1} - h_i) = O(t + h_1) = O(\log n).$$

2.3 Remarks

The BST and the 2-3 tree are fundamental structures covered in most standard textbooks, a notable example of which is [14]. Their inventors are controversial, but the interested students may see whom Wikipedia attributes them to at `https://en.wikipedia.org/wiki/Segment_tree` and `https://en.wikipedia.org/wiki/Binary_search_tree`.

Exercises

Problem 1. Complete the proof of Lemma 2.3.

Problem 2 (range max). Consider n people for each of whom we have her/his age and salary. Design a data structure of $O(n)$ space to answer the following query in $O(\log n)$ time: find the maximum salary of all the people aged between x and y, where $x, y \in \mathbb{R}$.

Problem 3. Let S is a set of n real values. Given a count BST on S, explain how to answer following query in $O(\log n)$ time: find the k-th largest element in S, where k can be any integer from 1 to n.

Problem 4. Let $\mathcal{T}$ be a 2-3 tree on a set S of n real values. Given any $x \leq y$, describe an algorithm to obtain in $O(\log n)$ time a 2-3 tree on the set $S \setminus [x, y]$ (namely, the set of elements in S that are not covered by $[x, y]$).

Problem 5* (meldable heap). Design a data structure of $O(n)$ space to store a set S of n real values to satisfy the following requirements:

- An element can be inserted to S in $O(\log n)$ time.
- The smallest element in S can be deleted in $O(\log n)$ time.
- Let S_1, S_2 be two disjoint sets of real values. Given a data structure (that you have designed) on S_1 and another on S_2, you can obtain a data structure on $S_1 \cup S_2$ in $O(\log(|S_1| + |S_2|))$ time. Note that here we do not have the constraint that the values in S_2 should be larger than those in S_1.

Problem 6. Modify the 2-3 tree into a *count 2-3 tree* that supports range counting in $O(\log n)$ time. Also explain how to maintain the count 2-3 tree in $O(\log n)$ time under insertions, deletions, (tree) splits, and joins.

Lecture 3: Structures for Intervals

In this lecture, we will discuss the *interval tree* and the *segment tree*, which represent two different approaches to store a set S of intervals of the form $\sigma = [x, y]$ where x and y are real values. The ideas behind the two approaches are very useful in designing sophisticated structures on intervals, segments, rectangles, etc. (this will be clear later in the course). Set $n = |S|$. The interval tree uses linear space (i.e., $O(n)$), whereas the segment tree uses $O(n \log n)$ space.

We will also take the chance to introduce the *stabbing query.* Formally, given a search value $q \in \mathbb{R}$, a *stabbing query* returns all the intervals $\sigma \in S$ satisfying $q \in \sigma$. Both the interval tree and the segment tree answer a query in $O(\log n + k)$ time, where k is the number of intervals reported.

A remark is in order at this point. Since the interval tree has smaller space consumption, it may appear "better" than the segment tree. While there is some truth in this impression (e.g., the interval tree is indeed more superior for stabbing queries), we must bear in mind the real purpose of our discussion: to learn the approach that each structure takes to organize intervals. The segment tree approach may turn out to be more useful on certain problems, which we will see in the exercises.

3.1 The interval tree

3.1.1 Description of the structure

Given an interval $[x, y]$, we call x and y its *left* and *right endpoints*, respectively. Denote by P the set of endpoints of the intervals in S.

To obtain an interval tree on S, first create a BST $\mathcal{T}$ (Section 2.1) on P. For each node u in $\mathcal{T}$, define a set $stab(u)$ of intervals as follows:

> $stab(u)$ consists of every $\sigma \in S$ such that u is the *highest* node in $\mathcal{T}$ whose key is covered by σ.

We will refer to $stab(u)$ as the *stabbing set* of u. The intervals in $stab(u)$ are stored in two lists (i.e., two copies per interval): the first list sorts the intervals by left endpoint, while the second by right endpoint. Both lists are associated with u (i.e.,, we store in u a pointer to each list). This completes the construction of the interval tree.

Example. Consider $S = \{[1, 2], [3, 7], [4, 12], [5, 9], [6, 11], [8, 15], [10, 14], [13, 16]\}$. Figure 3.1 shows a BST on $P = \{1, 2, ..., 16\}$. The stabbing set of node 9 is $\{[6, 11], [4, 12], [5, 9], [8, 15]\}$; note that all the intervals in the stabbing set cover the key 9. The stabbing set of node 13, on the other hand, is $\{[10, 14], [13, 16]\}$. □

It is easy to verify that every interval in S belongs to the stabbing set of *exactly one* node. The space consumption of the interval tree is therefore $O(n)$.

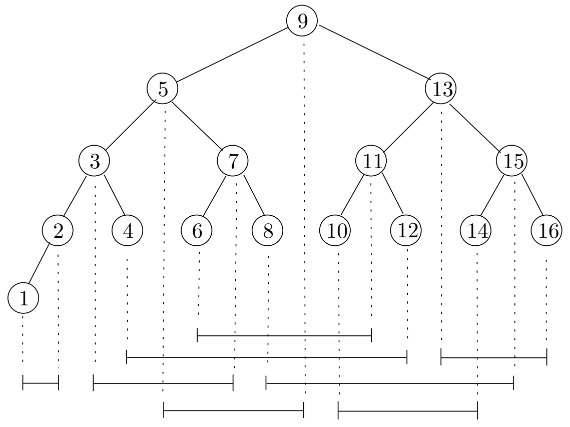


Figure 3.1: An interval tree

3.1.2 Stabbing query

Let us see how to use the interval tree on S constructed in Section 3.1.1 to answer a stabbing query search value q.

To get the main idea behind the algorithm, consider first the root u of the BST $\mathcal{T}$. Without loss of generality, let us assume that q is less than the key κ of u. We can forget about the intervals stored in the (stabbing sets of the nodes in the) right subtree of u, because all those intervals $[x, y]$ must satisfy $x \geq \kappa > q$, and hence, cannot cover q. We will, however, have to explore the left subtree of u, but that is something to be taken care of by recursion. At u, we must find a way to report the intervals in $stab(u)$ that cover q. Interestingly, this can be done in $O(1 + k_u)$ time, if k_u intervals are reported in $stab(u)$. For this purpose, we utilize the fact that all the intervals $[x, y] \in stab(u)$ *must* contain κ. Therefore, $[x, y]$ contains q if and only if $x \leq q$. We thus scan the intervals in $stab(u)$ in ascending order of *left* endpoint, and stop as soon as coming across an interval $[x, y]$ satisfying $x > q$.

This leads to the following algorithm for answering the stabbing query. First, descend a root-to-leaf path Π of $\mathcal{T}$ to reach the (only) conceptual leaf (Section 2.1.2) whose slab covers q. For every node u on Π with key κ:

- if $q < \kappa$, report the qualifying intervals in $stab(u)$ by scanning them in ascending order of left endpoint;
- if $q \geq \kappa$, report the qualifying intervals in $stab(u)$ by scanning them in descending order of right endpoint.

The query time is therefore

$$\sum_{u \in \Pi} O(1 + k_u) \quad = \quad O(\log n + k)$$

noticing that every interval is reported at exactly one node on Π.

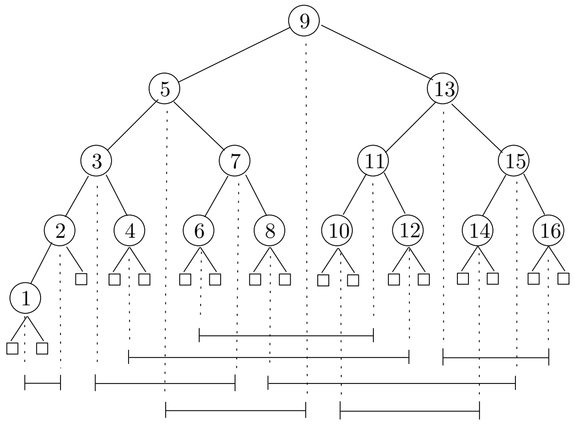


Figure 3.2: A segment tree (each box is a conceptual leaf)

3.2 The segment tree

3.2.1 Description of the structure

As before, let P be the set of end points in S. To obtain a segment tree on S, first create a BST $\mathcal{T}$ on P. Recall from Lemma 2.3 that every interval $\sigma \in S$ can be divided into $O(\log n)$ canonical intervals, each of which is the slab of an actual/conceptual node u in $\mathcal{T}$. We assign σ to every such u. Define S_u as the set of all intervals assigned to u. We store S_u in a linked list associated with u. This finishes the construction of the segment tree.

Example. Consider $S = \{[1,2], [3,7], [4,12], [5,9], [6,11], [8,15], [10,14], [13,16]\}$. Figure 3.2 shows a BST on $P = \{1, 2, ..., 16\}$ with the conceptual leaves indicated. Interval $[4,12]$, for example, is partitioned into canonical intervals $[4,5)$, $[5,9)$, $[9,11)$, $[11,12)$, and hence, is assigned to 4 nodes: the right conceptual leaf of node 4, node 7, node 10, and the left conceptual leaf of node 12. To illustrate S_u, let u be node 10 in which case S_u contains $[4,12]$ and $[6,11]$. $\square$

Since every interval in S has $O(\log n)$ copies, the total space of the segment tree is $O(n \log n)$.

3.2.2 Stabbing query

A stabbing query with search value q can be answered with a very simple algorithm:

```
1. identify the set $\Pi$ of actual/conceptual nodes whose slabs contain $q$
2. for every node $u \in \Pi$
3.     report $S_u$
```

Proposition 3.1. *No interval is reported twice.*

Proof. Follows from the fact that the canonical intervals of any interval are disjoint (Lemma 2.3). $\square$

Proposition 3.2. *If $\sigma \in S$ covers q, σ must have been reported.*

Proof. Follows from the fact that one of the canonical intervals of σ must cover q (because they *partition* σ). $\square$

It is thus clear that the query cost is $O(\log n + k)$, noticing that Π can be identified by following a single root-to-leaf path in $O(\log n)$ time.

3.3 Remarks

The interval tree was independently proposed by Edelsbrunner [17] and McCreight [32], while the segment tree was designed by Bentley [6].

Exercises

Problem 1. Describe how to construct an interval tree on n intervals in $O(n \log n)$ time.

Problem 2. Describe how to construct a segment tree on n intervals in $O(n \log n)$ time.

Problem 3. Let S be a set of n intervals in $\mathbb{R}$. Design a structure of $O(n)$ space to answer the following query efficiently: given an interval $q = [x, y]$ in $\mathbb{R}$, report all the intervals $\sigma \in S$ such that $\sigma \cap q \neq \emptyset$. Your query time needs to be $O(\log n + k)$, where k is the number of reported intervals.

Problem 4 (stabbing max). Suppose that we are managing a server. For every connection session to the server, we store its: (i) logon time, (ii) logoff time, and (iii) the network bandwidth consumed. Let n be the number of sessions. Design a data structure of $O(n)$ space to answer the following query in $O(\log n)$ time: given a timestamp t, return the session with the largest consumed bandwidth among all the sessions that were active on t.

(Hint: even though the space needs to be $O(n)$, still organize the intervals following the approach of the segment tree.)

Problem 5 (2D stabbing max). Let S be a set of n axis-parallel rectangles in $\mathbb{R}^2$ (i.e., each rectangle in S has the form $[x_1, x_2] \times [y_1, y_2]$). Each rectangle $r \in S$ is associated with a real-valued *weight*. Describe a structure of $O(n \log n)$ space that answers the following query in $O(\log^2 n)$ time: given a point $q \in \mathbb{R}^2$, report the maximum weight of the rectangles $r \in S$ satisfying $q \in r$.

Problem 6. Let S be a set of n horizontal segments of the form $[x_1, x_2] \times y$ in $\mathbb{R}^2$. Given a vertical segment $q = x \times [y_1, y_2]$, a query reports all the segments $\sigma \in S$ that intersect q. Design a data structure to store S in $O(n \log n)$ space such that every query can be answered in $O(\log^2 n + k)$ time, where k is the number of segments reported.

Lecture 4: Structures for Points

Each real value can be regarded as a 1D *point*; with this perspective, a BST can be regarded as a data structure managing 1D points. In this lecture, we will discuss several structures designed to manage *multidimensional* points in $\mathbb{R}^d$ where the dimensionality $d \geq 2$ is a constant. Our discussion will focus on $d = 2$, while in the exercises you will be asked to obtain structures of higher dimensionalities by extending the ideas we will learn.

Central to our discussion is *orthogonal range reporting*. Let S be a set of points in $\mathbb{R}^d$. Given an axis-parallel rectangle $q = [x_1, y_1] \times [x_2, y_2] \times ... \times [x_d, y_d]$, an (orthogonal) *range reporting* query returns $q \cap S$. This generalizes the 1D range reporting mentioned in Section 2.1.1 (which can be handled efficiently by a BST). Set $n = |S|$. The structures to be presented in this lecture will provide different tradeoffs between space and query time.

For simplicity, we will assume that the points of S are in *general position*: no two points in S have the same x-coordinate or y-coordinate. This assumption allows us to focus on the most important ideas, and can be easily removed with standard tie breaking techniques, as we will see in an exercise.

4.1 The kd-tree

This data structure stores S in $O(n)$ space, and answers a 2D range reporting query in $O(\sqrt{n} + k)$ time, where k is the number of points in $q \cap S$.

4.1.1 Structure

We describe the kd-tree in a recursive manner.

$\boldsymbol{n = 1}$**.** If S has only a single point p, the kd-tree has only a single node storing p.

$\boldsymbol{n \geq 2}$**.** Let ℓ be a *vertical* line that divides P as evenly as possible, that is, there are at most $\lceil n/2 \rceil$ points of P on each side of ℓ. Create a root node u of the kd-tree, and store ℓ (i.e., the x-coordinate of ℓ) at u. Let P_1 (or P_2, resp.) be the set of points in P that are on the left (or right, resp.) of ℓ.

Consider now P_1. If $|P_1| = 1$, create a left child v_1 of u storing the only point in P_1. Next, we assume $|P_1| \geq 2$. Let ℓ_1 be a *horizontal* line that divides P_1 as evenly as possible. Create a left child v_1 of u storing the line ℓ_1. Let P_{11} (or P_{12}, resp.) be the set of points in P_1 that are below (or above, resp.) of ℓ_1. Recursively, create a kd-tree $\mathcal{T}_{11}$ on P_{11} and a kd-tree $\mathcal{T}_{12}$ on P_{12}. Make $\mathcal{T}_{11}$ and $\mathcal{T}_{12}$ the left and right subtrees of v_1, respectively.

The processing of P_2 is similar. If $|P_2| = 1$, create a right child v_2 of u storing the only point in P_2. Otherwise, let ℓ_2 be a horizontal line that divides P_2 as evenly as possible. Create a right child v_2 of u storing the line ℓ_2. Let P_{21} (or P_{22}, resp.) be the set of points in P_2 that are below (or

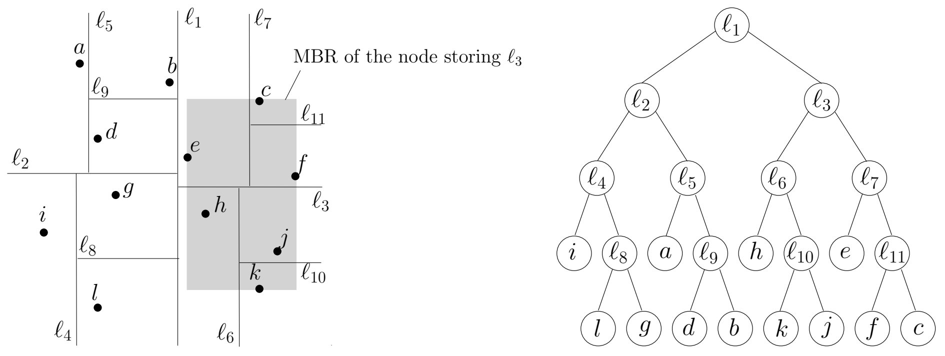

MBR of the node storing ℓ_3
ℓ_1 ℓ_2 ℓ_3 ℓ_4 ℓ_5 ℓ_6 ℓ_7 ℓ_8 ℓ_9 ℓ_{10} ℓ_{11}
a b c d e f g h i j k l

Figure 4.1: A kd-tree

above, resp.) of ℓ_2. Recursively, create a kd-tree $\mathcal{T}_{21}$ on P_{21} and a kd-tree $\mathcal{T}_{22}$ on P_{22}. Make $\mathcal{T}_{21}$ and $\mathcal{T}_{22}$ the left and right subtrees of v_2, respectively.

The kd-tree is a binary tree where every internal node has two children, and that the points of S are stored only at the leaf nodes. The total number of nodes is therefore $O(n)$.

For each node u in the kd-tree, we store its *minimum bounding rectangle* (MBR) which is the *smallest* axis-parallel rectangle covering all the points stored in the subtree of u. Note that the MBR of an internal node u can be obtained from those of its children in constant time.

Example. Figure 4.1 shows a kd-tree on a set S of 12 points. The shaded rectangle illustrates the MBR of the node storing the horizontal line ℓ_3 (i.e., the right child of the root). $\square$

4.1.2 Range reporting

Let $\mathcal{T}$ be a kd-tree on S. A range reporting query can be answered by simply visiting all the nodes in $\mathcal{T}$ whose MBRs intersect with the search rectangle q. Whenever a leaf node is encountered, we report the point p stored there if $p \in q$.

Next, we will prove that the query cost is $O(\sqrt{n} + k)$. For this purpose, we divide the nodes u accessed into two categories:

- Type 1: the MBR of u intersects with a boundary edge of q (note that q has 4 boundary edges).
- Type 2: the MBR of u is fully contained in q.

We will prove that there are $O(\sqrt{n})$ nodes of Type 1. In an exercise, you will be asked to prove that the number of nodes of Type 2 is bounded by $O(k)$. It will then follow that the query cost is $O(\sqrt{n} + k)$.

Lemma 4.1. *Any vertical line ℓ can intersect with the MBRs of $O(\sqrt{n})$ nodes.*

Proof. It suffices to prove the lemma only for the case where n is a power of 2 (think: why?). Fix any ℓ. We say that a node is *ℓ-intersecting* if its MBR intersects with ℓ. Let $f(n)$ be the maximum number of ℓ-intersecting nodes in any kd-tree storing n points. Clearly, $f(n) = O(1)$ for any constant n.

Now consider the kd-tree $\mathcal{T}$ we constructed on S. Let $\hat{u}$ be the root of $\mathcal{T}$; recall that $\hat{u}$ stores a vertical line ℓ_1. Due to symmetry, let us assume that ℓ is on the right of ℓ_1. Denote by u the right

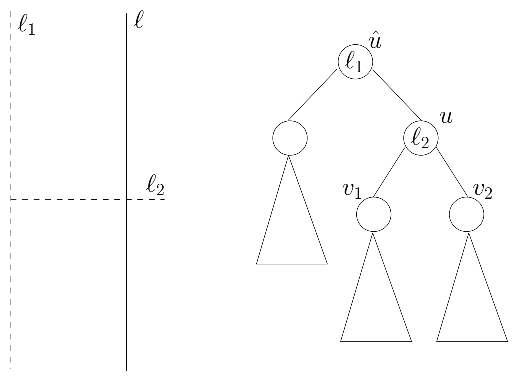


Figure 4.2: Proof of Lemma 4.1

child of p; note that the line ℓ_2 stored in u is horizontal. Let v_1 and v_2 be the left and right child nodes of u, respectively. See Figure 4.2 for an illustration.

What can be the ℓ-intersecting nodes in $\mathcal{T}$? Clearly, they can only be $\hat{u}$, u, and the ℓ-intersecting nodes in the subtrees of v_1 and v_2. Since the subtree of v_1 (or v_2) contains $n/4$ points, we thus have:

$$f(n) \le 2 + 2 \cdot f(n/4).$$

Solving the recurrence gives $f(n) = O(\sqrt{n})$. $\square$

An analogous argument shows that any *horizontal* line can intersect with the MBR of $O(\sqrt{n})$ nodes, too. Observe that the MBR of any Type-1 node must intersect with at least one of the following 4 lines: the two vertical lines passing the left and right edges of q, and the two vertical lines passing the lower and upper edges of q. It thus follows that there can be $O(\sqrt{n})$ Type-1 nodes.

4.2 A bootstrapping lemma

This section will present a technique to obtain a structure that uses $O(n)$ space, and answers any range reporting query in $O(n^\epsilon + k)$ time, where $\epsilon > 0$ can be any small constant (for the kd-tree, $\epsilon = 1/2$). The core of our technique is the following lemma:

Lemma 4.2. *Suppose that there is a structure Υ that can store n points in $\mathbb{R}^2$ in at most $F(n)$ space, and answers a range reporting query in at most $Q(n) + O(k)$ time. For any integer $\lambda \in [2, n/2]$, there exists a structure that uses at most $\lambda \cdot F(\lceil n/\lambda \rceil) + O(n)$ space and answers a range reporting query in at most $2 \cdot Q(\lceil n/\lambda \rceil) + \lambda \cdot O(\log(n/\lambda)) + O(k)$ time.*

Proof. Let S be the set of n points. Find $\lambda - 1$ vertical lines $\ell_1, ..., \ell_{\lambda-1}$ to satisfy the following requirements:

- No point of S falls on any line.
- If $x_1, ..., x_{\lambda-1}$ are the x-coordinates of $\ell_1, ..., \ell_{\lambda-1}$, respectively, let us define λ *slabs* as follows:
  - Slab 1 includes all the points of $\mathbb{R}^2$ with x-coordinate less than x_1;
  - Slab $i \in [2, \lambda - 1]$ includes all the points of $\mathbb{R}^2$ with x-coordinate in $[x_{i-1}, x_i)$;
  - Slab λ includes all the points of $\mathbb{R}^2$ with x-coordinate at least $x_{\lambda-1}$.

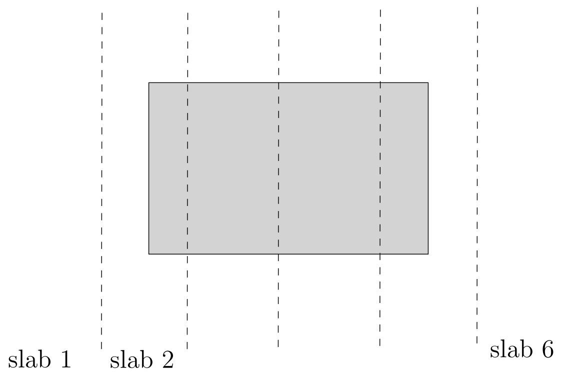


Figure 4.3: Proof of Lemma 4.2

We require that S should have at most $\lceil n/\lambda \rceil$ points in each slab.

For each $i \in [1, \lambda]$, define S_i to be the set of points in S that are covered by slab i. For each S_i, we create two structures:

- The data structure Υ (as stated in the lemma) on S_i; we will denote the data structure as $\mathcal{T}_i$.
- A BST B_i on the y-coordinates of the points in S_i.

The space consumption is clearly $\lambda \cdot F(\lceil n/\lambda \rceil) + O(n)$.

Let us now discuss how to answer a range reporting query with search rectangle q. If q falls entirely in some slab $i \in [1, \lambda]$, we answer the query using $\mathcal{T}_i$ directly in $Q(\lceil n/\lambda \rceil) + O(k)$ time.

Consider now the case where q intersects with at least two slabs. Denote by q_i the intersection of q with slab i, for every $i \in [1, \lambda]$. Each q_i is one of the following types:

- Type 1: empty — this happens when q is disjoint with slab i.
- Type 2: the x-range of q_i is precisely the x-range of slab i — this happens when the x-range of q spans the x-range of slab i.
- Type 3: the x-range of q_i is non-empty, but is shorter than that of slab i.

Figure 4.3 shows an example where q is the shaded rectangle, and $\lambda = 6$. Rectangles q_1 and q_6 are of Type 1, q_3 and q_4 are of Type 2, while q_2 and q_5 are of Type 3.

For Type 1, we do not need to do anything. For Type 3, we deploy $\mathcal{T}_i$ to find $q_i \cap S_i$ in $Q(\lceil n/\lambda \rceil) + O(k_i)$ time, where $k_i = |q_i \cap S_i|$. Note that there can be at most two rectangles of Type 3; so we spend at most $2 \cdot Q(\lceil n/\lambda \rceil) + O(k)$ time on them.

How about a rectangle q_i of Type 2? A crucial observation is that we can forget about the x-dimension. Specifically, a point $p \in S_i$ falls in q_i if and only if the y-coordinate of p is covered by the y-range of q_i. We can therefore find all the points of $q_i \cap S_i$ using B_i in $O(\log(n/\lambda) + k_i)$ time. Since there can be λ rectangles of Type 2, we end up spending at most $\lambda \cdot O(\log(n/\lambda)) + O(k)$ time on them. $\square$

The above lemma is *bootstrapping* because once we have obtained a data structure for range reporting, it may allow us to *improve* ourselves "automatically". For example, with the kd-tree, we

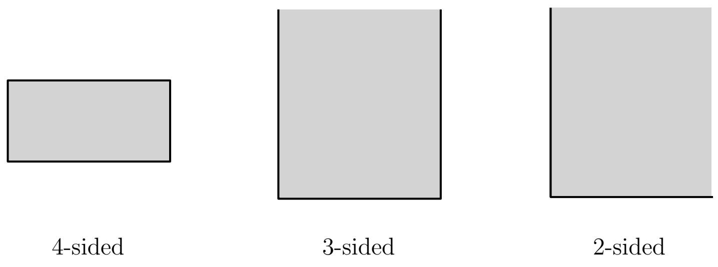


Figure 4.4: Different types of axis-parallel rectangles

have already achieved $F(n) = O(n)$ and $Q(n) = O(\sqrt{n})$. Thus, by Lemma 4.2, for any $\lambda \in [2, n/2]$ we immediately have a structure of $\lambda \cdot F(\lceil n/\lambda \rceil) = O(n)$ space whose query time is

$$O(\sqrt{n/\lambda}) + \lambda \cdot O(\log n)$$

plus the linear output time $O(k)$. Setting λ to $\Theta(n^{1/3})$ makes the query time $O(n^{1/3} \log n + k)$; note that this is a polynomial improvement over the kd-tree!

But we can do even better! Now that we have achieved $F(n) = O(n)$ and $Q(n) = O(n^{1/3} \log n)$, for any $\lambda \in [2, n/2]$ Lemma 4.2 immediately yields another structure of $O(n)$ space whose query time is

$$\tilde{O}((n/\lambda)^{1/3}) + \lambda \cdot O(\log n)$$

plus the linear output time $O(k)$. Setting λ to $\Theta(n^{1/4})$ makes the query time $\tilde{O}(n^{1/4}) + O(k)$, thus achieving another polynomial improvement!

Repeating this roughly $1/\epsilon$ times produces a structure of $O(n/\epsilon) = O(n)$ space and query time $O(n^{\epsilon} + k)$, where ϵ can be any positive constant.

4.3 The priority search tree

The 2D range reporting queries we have been considering so far are *4-sided* because the query rectangle q is "bounded" on all sides. More specifically, if we write q as $[x_1, x_2] \times [y_1, y_2]$, all the four values x_1, x_2, y_1, and y_2 are finite (they are neither ∞ nor $-\infty$). Such queries are difficult in the sense that no linear-size structures known today are able to guarantee a query time of $O(\log n + k)$.

If exactly one of the four values x_1, x_2, y_1, and y_2 takes an infinity value (i.e., $-\infty$ or ∞), q is said to be *3-sided.* More specially, if (i) two of the four values x_1, x_2, y_1, and y_2 take infinity values, and (ii) they are on different dimensions, q is said to be *2-sided.* See Figure 4.4 for an illustration.

Clearly, 3-sided queries are special 4-sided queries. Therefore, a structure on 4-sided queries also works on 3-sided queries, and but *not* the vice versa. In this section, we will introduce a 3-sided structure called the *priority search tree* which uses linear space, and answers a (3-sided) query in $O(\log n + k)$ time, where k is the number of points reported. Note that this is significantly better than using a kd-tree to answer 3-sided queries. The new structure also works on 2-sided queries because they are special 3-sided queries.

Due to symmetry, we consider search rectangles of the form $q = [x_1, x_2] \times [y, \infty)$ (as shown in the middle of Figure 4.4).

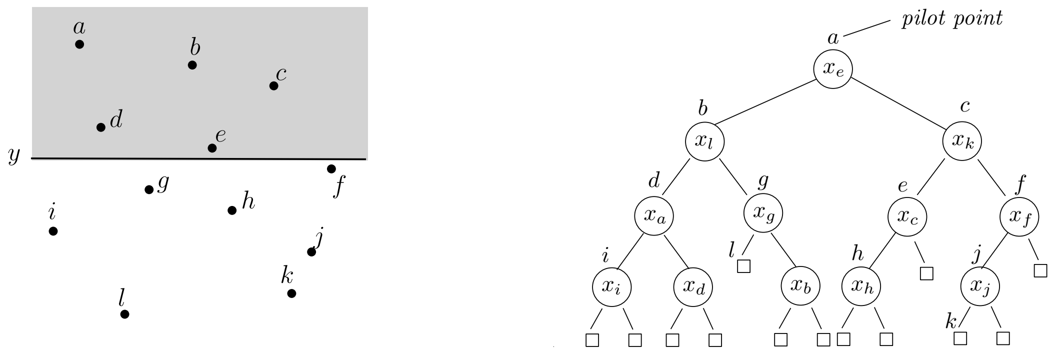


Figure 4.5: A priority search tree

4.3.1 Structure

To create a priority search tree on S, first create a BST $\mathcal{T}$ on the x-coordinates of the points in S. Each actual/conceptual node u in $\mathcal{T}$ may store a *pilot point*, defined recursively as follows:

- If u is the root of $\mathcal{T}$, its pilot point is the *highest* point in S.
- Otherwise, its pilot point is the highest among those points p satisfying
  - the x-coordinate of p is in $slab(u)$ (see Section 2.1.2 for the definition of slab), and
  - p is not the pilot point of any proper ancestor of u.

  If no such point exists, u has no pilot point associated.

This finishes the construction of the priority search tree. Note that every point in S is the pilot point of *exactly* one node (which is possibly conceptual). It is clear that the space is $O(n)$.

Example. Figure 4.5 shows a priority search tree on the point set $\{a, b, ..., l\}$. The x-coordinate of a point p is denoted as x_p in the tree. □

Remark. Observe that the priority search tree is simultaneously a *max heap* on the y-coordinates of the points in S. For this purpose, the priority search tree is also known by the name *treap*.

4.3.2 Answering a 3-sided query

Before talking about general 3-sided queries, let us first consider a (very) special version: the search rectangle q has the form $(-\infty, \infty) \times [y, \infty)$ (namely, q is "1-sided"). Equivalently, this is to ask how we can use the priority search tree to efficiently report all the points in S whose y-coordinate are at least y. Phrased yet in another way, this is to ask how we can we efficiently find all the keys at least y in a max heap. This can be done in $O(1 + k)$ time, where k is the number of elements returned.

Lemma 4.3. *Given a search rectangle $q = (-\infty, \infty) \times [y, \infty)$, we can find all the points in $S \cap q$ in $O(1 + k)$ time, where $k = |S \cap q|$.*

Proof. We answer the query using the following algorithm (setting u to the root of $\mathcal{T}$ initially):

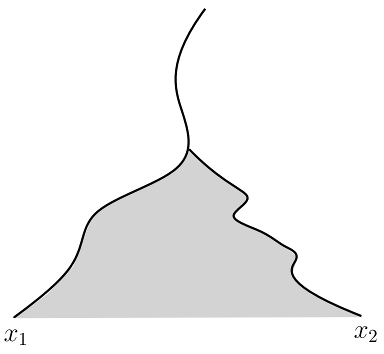


Figure 4.6: Search paths Π_1 and Π_2 and the portion in between

```
report-subtree(u, y)
/* u is an actual/conceptual node in T */
1. if u has no pilot point or its pilot point p has y-coordinate < y then return
2. report p
3. if u is a conceptual leaf then return
4. report-subtree(v_1, y) where v_1 is the left child of u (v_1 is possibly conceptual)
5. report-subtree(v_2, y) where v_2 is the right child of u (v_2 is possibly conceptual)
```

The correctness follows from the fact that the pilot point of u is the *highest* among all the pilot points stored in the subtree of u.

To bound the cost, notice that each node u we access can be divided into two types:

- Type 1: the pilot point of u is reported.
- Type 2: the pilot point is not reported.

Clearly, there are at most k nodes of Type 1. How many nodes of Type 2? A crucial observation is that the parent of a type-2 node must be of Type 1. Therefore, there can be at most $2k$ nodes of Type 2. The total cost is therefore $O(1 + k)$. □

Example. Suppose that q is the shaded region as shown in Figure 4.5 (note that q has the form $(-\infty, \infty) \times [y, \infty)$). The nodes accessed are: $x_e, x_l, x_a, x_i, x_d, x_g, x_k, x_c, x_h$, and x_f. □

We are now ready to explain how to answer a general 3-sided query with $q = [x_1, x_2] \times [y, \infty)$. Without loss of generality, we can assume that x_1 and x_2 are the x-coordinated of some points in S (think: why?). Let us first find

- the path Π_1 in $\mathcal{T}$ from the root to the node storing the x-coordinate x_1;
- the path Π_2 in $\mathcal{T}$ from the root to the node storing the x-coordinate x_2.

Figure 4.6 illustrates how Π_1 and Π_2 look like in general: they descend from the root and diverge at some node. We are interested in only the nodes u that

- are in $\Pi_1 \cup \Pi_2$, or
- satisfy $slab(u) \subseteq [x_1, x_2]$ — such are nodes are "in-between" Π_1 and Π_2 (the shaded portion in Figure 4.6).

For every other node v (violating both of the above), *slab*(v) must be disjoint with $[x_1, x_2]$; and therefore, the pilot point p of v cannot fall in q (recall that *slab*(v) must cover the x-coordinate of p).

This gives rise to the following the query algorithm:

```
1. find the paths $\Pi_1, \Pi_2$ as described above
2. **for** every node $u \in \Pi_1 \cup \Pi_2$
3.     report the pilot point $p$ of $u$ if $p \in q$
4. find the set $\Sigma$ of actual/conceptual nodes whose slabs are the canonical slabs of $[x_1, x_2]$
5. **for** every node $u \in \Sigma$
6.     **report-subtree**$(u, y)$
```

For every node $u \in \Sigma$, Line 5 finds *all* the qualifying pilot points (i.e., covered by q) that are stored in the subtree rooted at u, because (i) the subtree *itself* is a max heap, and (ii) we can forget about the x-range $[x_1, x_2]$ of q in exploring the subtree of u. By Lemma 4.3, the cost of **report-subtree**(u, y) is $O(1 + k_u)$ where k_u is the number of points reported from the subtree of u.

The total query cost is therefore bounded by

$$O\left(|\Pi_1| + |\Pi_2| + \sum_{u \in \Sigma}(1 + k_u)\right) = O(\log n + k).$$

The filtering technique. Usually when we look at a query time complexity such as $O(\log n + k)$, we would often interpret the $O(\log n)$ term as the "*search time we are prepared to waste without reporting anything*", and the $O(k)$ term as the "*reporting time we are justified to pay*". For example, in using a BST to answer a 1D range reporting query, we may waste $O(\log n)$ time because there can be $O(\log n)$ nodes that need to be visited but contribute nothing to the query result. As another example, in using a kd-tree to answer a 2D (4-sided) range reporting query, the number of such nodes is $O(\sqrt{n})$.

The above interpretation, however, misses a very interesting point: we can regard $O(\log n + k)$ more generally as $O(\log n + k + k)$, which says that we can actually "waste" as much as $O(\log n + k)$ time in "searching"! Indeed, this is true for using the priority search tree to answer a 3-sided query: notice that the algorithm may access $O(\log n + k)$ nodes whose pilot points are not reported! Subtly, we charge the time "wasted" this way on the output. *Only after* we have reported the pilot point p of a node u will we search the child nodes of u. The $O(1)$ cost of searching the child nodes hence is "paid" for by the reporting of p.

This idea (of charging the search time on the output) is known as *the filtering technique.*

4.4 The range tree

We now return to 4-sided queries, i.e., the search rectangle q is an arbitrary axis-parallel rectangle. We will introduce the *range tree* which consumes $O(n \log n)$ space, and answers a query in $O(\log^2 n + k)$ time.

4.4.1 Structure

First create a BST $\mathcal{T}$ on the x-coordinates of the points in S. For each actual/conceptual node u in $\mathcal{T}$, denote by S_u the set of points $p \in S$ satisfying $x_p \in$ *slab*(u) (recall that x_p is the x-coordinate of

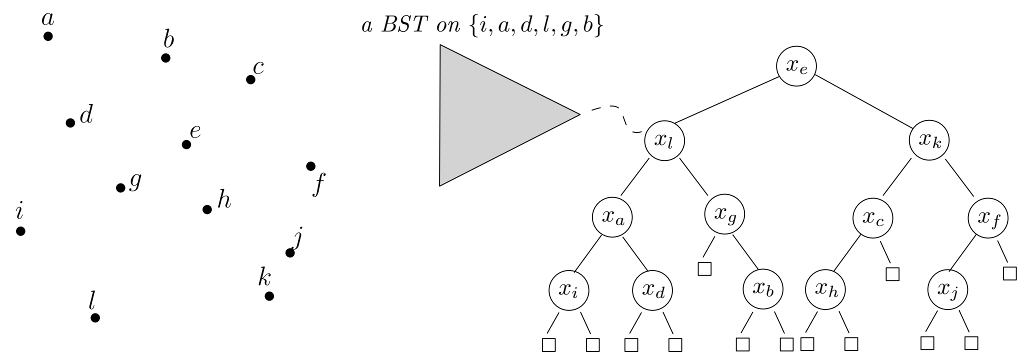


Figure 4.7: A range tree (the shaded triangle illustrates the secondary BST of node x_l)

p). For every node u, we associate it with a *secondary* BST $\mathcal{T}'_u$ on the y-coordinates of the points in S_u. Every point $p \in S_u$ is stored at the node in $\mathcal{T}'_u$ corresponding to the y-coordinate y_p of p.

Example. Figure 4.7 shows the BST $\mathcal{T}$ for the set of points shown on the left of the figure. If u is the node x_l, $S_u = \{i, a, d, l, g, b\}$. The secondary BST of u is created on the y-coordinates of those points. Point b is stored in the secondary BSTs of the right conceptual child of node x_b, node x_b itself, node x_g, node x_l, and node x_e. $\square$

Proposition 4.4. *For each point $p \in S$, x_p appears in the slabs of $O(\log n)$ nodes.*

Proof. By Proposition 2.1, if the slabs of two nodes u, v in $\mathcal{T}$ intersect, one of u, v must be an ancestor of the other. Thus, all the nodes whose slabs contain x_p must be on a single root-to-leaf path in $\mathcal{T}$. The proposition follows from the fact that the height of $\mathcal{T}$ is $O(\log n)$. $\square$

The space consumption is therefore $O(n \log n)$.

4.4.2 Range reporting

We answer a range reporting query with search rectangle $q = [x_1, x_2] \times [y_1, y_2]$ as follows (assuming x_1 and x_2 are the x-coordinates of some points in S, without loss of generality):

```
1. find the set $\Sigma$ of nodes in $\mathcal{T}$ whose slabs are the canonical slabs of $[x_1, x_2]$
2. for each node $u \in \Sigma$
3.     use $\mathcal{T}'_u$ to report $\{p \in S_u \mid y_p \in [y_1, y_2]\}$
```

Proposition 4.5. *Every point p in $q \cap S$ is reported exactly once.*

Proof. Clearly, $x_p \in [x_1, x_2]$. Therefore, x_p appears in *exactly* a canonical slab of $[x_1, x_2]$ (by Lemma 2.3, the canonical slabs form a partition of $[x_1, x_2]$). Let u be the node whose *slab*(u) is that canonical slab. Thus, $p \in S_u$ and will be reported *only* there. $\square$

The proof of the next proposition is left to you as an exercise:

Proposition 4.6. *The query time is $O(\log^2 n + k)$.*

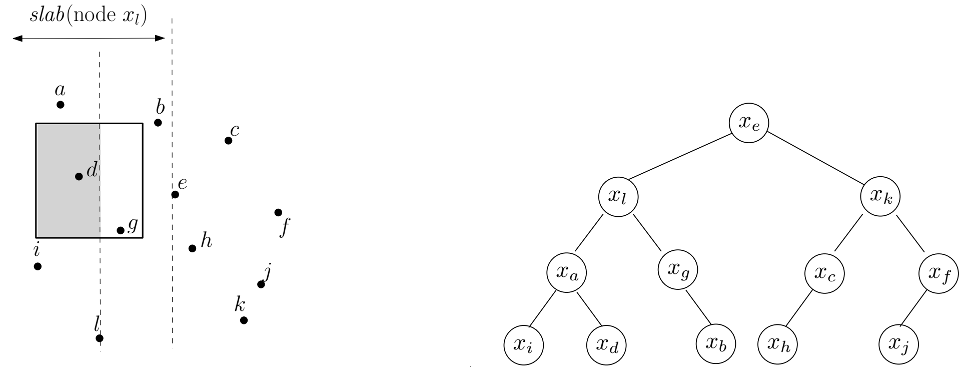


Figure 4.8: A modified range tree and how a query is cut into 3-sided queries

4.5 Another range tree with better query time

In this section, we will present a data structure that (finally) answers a 4-sided query in $O(\log n + k)$ time, while still retaining the $O(n \log n)$ space complexity. This is achieved by combining the range-tree idea with the priority search tree (Section 4.3), and converting a 4-sided query to two 3-sided queries.

4.5.1 Structure

First create a BST $\mathcal{T}$ on the x-coordinates of the points in S. For each actual/conceptual node u in $\mathcal{T}$, denote by S_u the set of points $p \in S$ satisfying $x_p \in slab(u)$. For every *actual* node u with key κ, define:

- $S_u^<$: the set of points $p \in S_u$ whose x-coordinate is less than κ;
- $S_u^\geq$: the set of points $p \in S_u$ whose x-coordinate is at least κ.

Note that $S_u^<$ and $S_u^\geq$ partition S_u. We associate u with two secondary structures:

- $\sqsubset_u$: a priority search tree on $S_u^<$ to answer "right-open" 3-sided queries, i.e., with search rectangles of the form $[x, \infty) \times [y_1, y_2]$;
- $\sqsupset_u$: a priority search tree on $S_u^\geq$ to answer "left-open" 3-sided queries, i.e., with search rectangles of the form $(-\infty, x] \times [y_1, y_2]$.

The space is $O(n \log n)$ by Proposition 4.4 (recall that each priority search tree uses space linear to the number of points stored).

Example. In Figure 4.8, as an example, let u be the node x_l. $\sqsubset_u$ is created on $S_u^< = \{a, d, i\}$, while $\sqsupset_u$ on $S_u^\geq = \{b, l, g\}$. □

4.5.2 Range reporting

Given a query with search rectangle $q = [x_1, x_2] \times [y_1, y_2]$ (assuming x_1 and x_2 are the x-coordinates of some points in S, without loss of generality), we answer it at the highest node u in $\mathcal{T}$ whose key κ is covered by q. Specifically, we construct

$$\begin{aligned} q_\sqsubset &= [x_1, \infty) \times [y_1, y_2] \\ q_\sqsupset &= (-\infty, x_2] \times [y_1, y_2] \end{aligned}$$

It is easy to see that

$$q \cap S \quad = \quad \left(q_{\sqsubset} \cap S_u^{<}\right) \cup \left(q_{\sqsupset} \cap S_u^{\geq}\right). \tag{4.1}$$

Example. Consider that search rectangle q shown in Figure 4.8. Node x_l is the node u designated earlier. Note how q is decomposed into two 3-sided rectangles: $q_{\sqsubset}$ is the part colored in gray, while $q_{\sqsubset}$ the part in white. The 3-sided query $q_{\sqsubset}$ on $S_u^{<}$ returns $\{d\}$, while the 3-sided query $q_{\sqsupset}$ on $S_u^{\geq}$ returns $\{g\}$. The union of $\{d\}$ and $\{g\}$ gives the result of the 4-sided query q. $\square$

Using the priority search trees $\sqsubset_u$ and $\sqsupset_u$, the point set on the right side of (4.1) can be retrieved in $O(\log n + k)$ time.

4.6 Pointer-machine structures

Have you noticed that, in all the structures we have discussed so far, the exploration of their content is always performed by following *pointers*? Indeed, they belong to a general class of structures known as the *pointer machine class.*

Formally, a *pointer machine structure* is a directed graph G satisfying the following conditions:

- There is a special node r in G that is called the *root*.
- Every node in G stores a constant number of words.
- Every node in G has a constant number of outgoing edges (but may have an arbitrary number of incoming edges).
- Any algorithm that accesses G must follow the rules below:
  - The first node visited must be the root r.
  - The algorithm is permitted to access a non-root node u in G only if it has already accessed an in-neighbor of u. This implies that the algorithm must have found a path from r to u in G.

You may convince yourself that all our structures so far, as well as their accompanying algorithms, satisfy the above conditions.

One simple structure that is *not* in the pointer machine class is the array. Recall that, given an array A of n, we can access directly $A[i]$ for any $i \in [1, n]$ in constant time, without following any path from some "root".

Pointer-machine structures bear unique importance in computer science because they are applicable in scenarios where it is not possible to perform any (meaningful) calculation on *addresses*. One such scenario arises from distributed computing where each "node" is a light weighted machine (e.g., your cell phone). A pointer to a node u is the IP address of machine u. No "arrays" can be implemented in such a scenario because, to enable constant time access to $A[i]$, you need to calculate the address of $A[i]$ by adding i to the starting address of A — something not possible in distributed computing (adding i to an IP address tells you essentially nothing).

4.7 Remarks

The kd-tree was first described by Bentley [5]. The priority search was invented by McCreight [33]. The range tree was independently developed by several works that appeared almost the same time, e.g., [7, 29, 30, 43].

Range reporting on pointer machines has been well understood. In 2D space, any pointer-machine structures achieving $O(\text{polylog}\, n + k)$ query time — let alone $O(\log n + k)$ — must consume $\Omega(n \frac{\log n}{\log \log n})$ space [13]. A structure matching this lower bound and attaining $O(\log n + k)$ query time has been found [11]. Note that the our structure in Section 4.5 is nearly optimal, except that its space is higher than the lower bound by an $O(\log \log n)$ factor. Similar results also hold for higher dimensionalities, except that both the space and query complexities increase by $O(\text{polylog}\, n)$ factors; see [1, 13].

By fully leveraging the power of the RAM model (address calculation and atomic operations that manipulate the bits within a word), it is possible to design structures with better complexities *outside* the pointer-machine class. For example, in 2D space, it is possible to achieve $O(\log n + k)$ time using $O(n \log^{\epsilon} n)$ space, where $\epsilon > 0$ can be any small constant [2, 12]. See also [10] for results of higher dimensionalities.

Exercises

Problem 1. Prove that there can be $O(k)$ nodes of Type 2 (as defined in Section 4.1.2).

Problem 2. Describe an algorithm to build the kd-tree on n points in $O(n \log n)$ time.

Problem 3. Explain how to remove the general position assumption for the kd-tree. That is, you still need to retain the same space and query complexities even if the assumption does not hold.

Problem 4. Let S be a set of points in $\mathbb{R}^d$ where $d \geq 2$ is a constant. Extend the kd-tree to obtain a structure of $O(n)$ space that answers any d-dimensional range reporting query in $O(n^{1-1/d} + k)$ time, where k is the number of points reported.

Problem 5. What is the counterpart of Lemma 4.2 in 3D space?

Problem 6*. Improve the query time in Lemma 4.2 to $2 \cdot Q(\lceil n/\lambda \rceil) + O(\log n + \lambda + k)$.

(Hint: one way to do so is to use the interval tree and stabbing queries.)

Problem 7. Consider the stabbing query discussed in Lecture 3 on a set S of n intervals in $\mathbb{R}$. Show that you can store S in a priority search tree such that any stabbing query can be answered in $O(\log n + k)$ time, where k is the number of intervals reported.

(Hint: turn the query into a 2-sided range reporting query on a set of n points converted from S.)

Problem 8. Prove Proposition 4.6.

Problem 9. Let S be a set of points in $\mathbb{R}^d$ where d is a constant. Design a data structure that stores S in $O(n \log^{d-1} n)$ space, and answers any orthogonal range reporting query on S in $O(\log^{d-1} n + k)$ time, where k is the number of reported points.

Problem 10 (range counting). Let S be a set of n points in $\mathbb{R}^2$. Given an axis-parallel rectangle q, a *range count* query reports $|q \cap S|$, i.e., the number of points in S that are covered by q. Design a structure that stores S in $O(n \log n)$ space, and answers a range count query in $O(\log^2 n)$ time.

Problem 11*. Let S be a set of n horizontal segments of the form $[x_1, x_2] \times y$ in $\mathbb{R}^2$. Given a vertical segment $q = x \times [y_1, y_2]$, a query reports all the segments $\sigma \in S$ that intersect q. Design a data structure to store S in $O(n)$ space such that every query can be answered in $O(\log^2 n + k)$ time, where k is the number of segments reported. (This improves an exercise in Lecture 3.)

(Hint: use the interval tree as the base tree, and the priority search tree as secondary structures.)

Problem 12. Prove: on a pointer-machine structure G with n nodes, the longest path from the root to a node in G has length $\Omega(\log n)$. (This implies that $O(\log n + k)$ is the best query bound one can hope for range reporting using pointer-machine structures.)

(Hint: suppose that each node has an outdegree of 2. Starting from the root, how many nodes can you reach within x hops?)

Lecture 5: Logarithmic Method and Global Rebuilding

We have seen some interesting data structures so far, but there is an issue: they are all *static* (except the BST and the 2-3 tree). It is not clear how they can be updated when the underlying set S of elements undergoes changes, i.e., insertions and deletions. This is something we will fix in the next few lectures.

In general, a structure is *semi-dynamic* if it allows elements to be inserted but not deleted; it is (fully) *dynamic* if both insertions and deletions are allowed. In this lecture, we will learn a powerful technique called the *logarithmic method* for turning a static structure into a semi-dynamic one. The technique is generic because it works (in exactly the same way) on a great variety of structures.

We will use the kd-tree (Section 4.1) to illustrate the technique. Indeed, the kd-tree serves as an excellent example because it seems exceedingly difficult to support any updates on that structure. Several constraints must be enforced for the structure to work. For example, the first cut ought to be a vertical line ℓ that divides the input set of points *as evenly as possible.* Unfortunately, a single point insertion would throw off the balance and thus destroy the whole tree. It may therefore be surprising that later we will make the kd-tree semi-dynamic *without* changing the structure at all!

There is another reason why we want to discuss the kd-tree: it can actually support deletions in a fairly easy way! In general, if a structure can support deletions but not insertions, the logarithmic method would turn it into a fully dynamic structure. Sometimes, for that to happen, we would also need to perform *global rebuilding*, which simply rebuilds everything from scratch! This is also something that can be illustrated very well by the kd-tree.

5.1 Amortized update cost

Recall that the BST supports an update (i.e., insertion/deletion) in $O(\log n)$ *worst-case* time. This means that an update definitely finishes after $O(\log n)$ atomic operations, where n is the number of nodes in the BST *currently* (assuming $n \geq 2$).

In this course, we will *not* aim to achieve worst-case update time (see, however, the remark at the end of this subsection). Instead, we will focus on obtaining small *amortized* update time. But what does it mean exactly to say that a structure has amortized time, for example, $O(\log n)$?

A structure with low amortized update cost should be able to support any number of updates with a small *total* cost, even though the time of an individual update may be large. Formally, suppose that the data structure processes n_{upd} updates. We can claim that the i-th update ($1 \leq i \leq n_{upd}$) takes $\bar{C}_i$ *amortized time*, only if the following is true

$$\text{total cost of the } n_{upd} \text{ updates} \quad \leq \quad \sum_{i=1}^{n_{upd}} \bar{C}_i. \tag{5.1}$$

Therefore:

- if a structure can process any sequence of n_{ins} insertions in $n_{ins} \cdot U_{ins}$ time, the insertion cost is U_{ins} amortized;
- if a structure can process any sequence of n_{del} deletions in $n_{ins} \cdot U_{del}$ time, the insertion cost is U_{del} amortized;
- if a structure can process any update sequence containing n_{ins} insertions and n_{del} deletions (in an arbitrary order) in $n_{ins} \cdot U_{ins} + n_{del} \cdot U_{del}$ time, the insertion cost is U_{ins} amortized, while the deletion cost is U_{del} amortized.

Remark. There are standard *de-amortization* techniques (see [35]) that convert a structure with small amortized update time into a structure with small *worst case* update time. Therefore, for many problems, it suffices to focus on amortized cost. The curious students may approach the instructor for a discussion.

5.2 Decomposable problems

We have discussed many types of *queries*, each of which retrieves certain information about the elements in the input set satisfying some conditions specified by the query. For example, for range reporting, the "information" is simply the elements themselves, whereas for range counting (Section 2.1.3), it is the number of those elements.

We say that a query is *decomposable* if the following is true for any *disjoint* sets of elements S_1 and S_2: given the query answer on S_1 and the answer on S_2, the answer on $S_1 \cup S_2$ can be obtained in constant time.

Consider, for example, orthogonal range reporting on 2D points. Given an axis-parallel rectangle q, the query answer on S_1 (or S_2) is the set Σ_1 (or Σ_2) of points therein covered by q. Clearly, $\Sigma_1 \cup \Sigma_2$ is the answer of the same query on $S_1 \cup S_2$. In other words, once Σ_1 and Σ_2 are available, we have already obtained the answer on $S_1 \cup S_2$ (nothing needs to be done). Hence, the query is decomposable.

As another example, consider range counting on a set of real values. Given an interval $q \subseteq \mathbb{R}$, the query answer on S_1 (or S_2) is the number c_1 (or c_2) of values therein covered by q. Clearly, $c_1 + c_2$ is the answer of the same query on $S_1 \cup S_2$. In other words, once c_1 and c_2 are available, we can obtain the answer on $S_1 \cup S_2$ in constant time. Hence, the query is decomposable.

Verify for yourself that all the queries we have seen so far are decomposable: predecessor/successor, find-min/max, range reporting, range counting/max, stabbing, etc.

5.3 The logarithmic method

This section serves as a proof of the following theorem:

Theorem 5.1. *Suppose that there is a static structure Υ that*

- *stores n elements in at most $F(n)$ space;*
- *can be constructed in at most $n \cdot U(n)$ time;*
- *answers a decomposable query in at most $Q(n)$ time (plus, if necessary, a cost linear to the number of reported elements).*

Set $h = \lceil \log_2 n \rceil$. There is a semi-dynamic structure Υ' that

- *stores n elements in at most $\sum_{i=0}^{h} F(2^i)$ space;*
- *supports an insertion in $O\left(\sum_{i=0}^{h} U(2^i)\right)$ amortized time;*
- *answers a decomposable query in $O(\log n) + \sum_{i=0}^{h} Q(2^i)$ time (plus, if necessary, a cost linear to the number of reported elements)*

Before proving the theorem, let us first see its application on the kd-tree. We know that the kd-tree consumes $O(n)$ space, can be constructed in $O(n \log n)$ time (this was an exercise of Lecture 4), and answers a range reporting query in $O(\sqrt{n} + k)$ time, where k is the number of reported elements. Therefore:

$$\begin{aligned} F(n) &= O(n) \\ U(n) &= O(\log n) \\ Q(n) &= O(\sqrt{n}). \end{aligned}$$

Theorem 5.1 immediately gives a semi-dynamic structure that uses

$$\sum_{i=0}^{\lceil \log_2 n \rceil} O(2^i) = O(n)$$

space, supports an insertion in

$$\sum_{i=0}^{\lceil \log_2 n \rceil} O\left(\log 2^i\right) = O(\log^2 n)$$

time, and answers a query in

$$\sum_{i=0}^{\lceil \log_2 n \rceil} O\left(\sqrt{2^i}\right) = O(\sqrt{n})$$

plus $O(k)$ time.

5.3.1 Structure

Let S be the input set of elements; set $n = |S|$ and $h = \lceil \log_2 n \rceil$. At all times, we divide S into disjoint subsets $S_0, S_1, ..., S_h$ (some of which may be empty) satisfying:

$$|S_i| \le 2^i. \tag{5.2}$$

Create a structure of Υ on each subset; denote by Υ_i the structure on S_i. Then, $\Upsilon_1, \Upsilon_2, ..., \Upsilon_h$ together constitute our "overall structure". The space usage is bounded by $\sum_{i=0}^{h} F(2^i)$.

Remark. At the beginning when we construct our "overall structure" from scratch, it suffices to set $S_h = S$, and $S_i = \emptyset$ for every $i \in [0, h-1]$.

5.3.2 Query

To answer a query q, we simply search all of $\Upsilon_1, ..., \Upsilon_h$. Since the query is decomposable, we can obtain the answer on S from the answers on $S_1, ..., S_h$ in $O(h)$ time. The overall query time is therefore

$$O(h) + \sum_{i=0}^{h} Q(2^i) = O(\log n) + \sum_{i=0}^{h} Q(2^i).$$

5.3.3 Insertion

To insert an element e_{new}, we first identify the smallest $i \in [0, h]$ satisfying:

$$1 + \sum_{j=0}^{i} |S_j| \quad \leq \quad 2^i. \tag{5.3}$$

We now proceed as follows:

- If i exists, we destroy $\Upsilon_0, \Upsilon_1, ..., \Upsilon_i$, and move *all* the elements in $S_0, S_1, ..., S_{i-1}$, together with e_{new}, *into* S_i (after this, $S_0, S_1, ..., S_{i-1}$ become empty). Build the structure Υ_i on the current S_i from scratch.
- If i does not exist, we destroy $\Upsilon_0, \Upsilon_1, ..., \Upsilon_h$, and move *all* the elements in $S_0, S_1, ..., S_h$, together with e_{new}, *into* S_{h+1} (after this, $S_0, S_1, ..., S_h$ become empty). Build the structure Υ_{h+1} on S_{h+1} from scratch. The value of h is then increased by 1.

Let us now analyze the amortized insertion cost with a charging argument. Each time Υ_i ($i \geq 0$) is rebuilt, we spend

$$O(|S_i|) \cdot U(|S_i|) \quad = \quad O(2^i) \cdot U(2^i) \tag{5.4}$$

cost (recall that the structure Υ on n elements can be built in $n \cdot U(n)$ time). The lemma below gives a crucial observation:

Lemma 5.2. *Every time when Υ_i is rebuilt, at least $1 + 2^{i-1}$ elements are added to S_i (i.e., every such element was in some S_j with $j < i$).*

Proof. Set $\lambda = i$. By the choice of i, we know that, before $S_0, ..., S_{\lambda-1}$ were emptied, (5.3) was violated when i was set to $\lambda - 1$. This means:

$$1 + \sum_{j=0}^{\lambda-1} |S_j| \quad \geq \quad 1 + 2^{\lambda-1}.$$

This proves the claim because all the elements in $S_1, ..., S_{\lambda-1}$, as well as e_{new}, are added to S_λ. $\square$

We can therefore charge the cost of rebuilding Υ_i — namely the cost shown in (5.4) — on the at least 2^{i-1} elements that are added to S_i, such that each of those elements bears only

$$\frac{O(2^i)}{2^{i-1}} \cdot U(2^i) \quad = \quad O(U(2^i))$$

cost.

In other words, every time an element e moves to new S_i, it bears a cost of $O(U(2^i))$. Note that an element never moves from S_i to an S_j with $j < i$! Therefore, e can be charged at most $h + 1$ times with a total cost of

$$O\left(\sum_{i=0}^{h} U(2^i)\right)$$

which is the amortized cost of the insertion of e. In other words, we have proved that any sequence of n insertions can be processed in

$$O\left(n \cdot \sum_{i=0}^{h} U(2^i)\right)$$

time.

5.4 Fully dynamic kd-trees with global rebuilding

Theorem 5.1 gives us a semi-dynamic version of the kd-tree. In this section, we will make the kd-tree fully dynamic, and take the chance to explain the *global rebuilding* technique.

5.4.1 The deletion algorithm

Recall that the kd-tree on a set S of n points is a binary tree $\mathcal{T}$ where every point $p \in S$ is stored at a leaf. The height of $\mathcal{T}$ is $O(\log n)$.

Suppose that we need to support only deletions, but not insertions. To delete a point $p \in S$, we carry out the following steps (assuming $n \geq 2$):

```
1. descend a root-to-leaf path $\Pi$ in $\mathcal{T}$ to find the leaf node $z$ storing $p$
2. remove $z$ from $\mathcal{T}$
3. $u \leftarrow$ the parent of $z$; $v \leftarrow$ the (only) child of $u$
4. if $u$ is the root of $\mathcal{T}$ then
5.     delete $u$, and make $v$ the root of $\mathcal{T}$
   else
6.     $\hat{u} \leftarrow$ the parent of $u$
7.     delete $u$, and make $v$ a child of $\hat{u}$
8. update the MBRs of the nodes on $\Pi$
```

See Figure 5.1 for an illustration. It is easy to verify that the deletion time is $O(\log N)$, where N is the number of points in S when the kd-tree was built (as deletions are carried out, n drops from N to 0). Note that $\mathcal{T}$ may appear "imbalanced" after a series of deletions.

We still answer queries using exactly the same algorithm explained in Section 4.1.2: namely, access all the nodes whose MBRs intersect with the search rectangle. Let us now discuss whether the above strategy is able to ensure $O(\sqrt{n} + k)$ query time. The lemma gives an affirmative answer, as long as n has not dropped too much:

Lemma 5.3. *The query time is $O(\sqrt{n} + k)$ as long as $n \geq N/2$.*

Proof. We will prove that the query time is $O(\sqrt{N} + k)$, which is $O(\sqrt{2n} + k) = O(\sqrt{n} + k)$. Let us recall the analysis we had in Section 4.1.2. We divided the nodes u accessed into two categories:

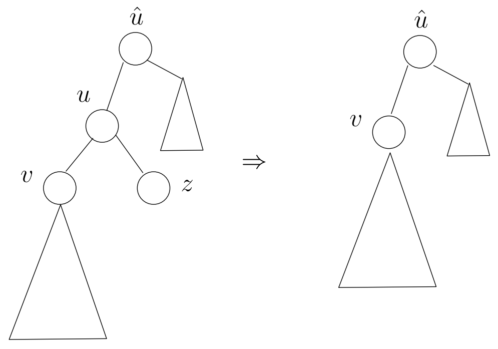


Figure 5.1: Deletion in a kd-tree (the deletion removes the point stored in the leaf node z)

- Type 1: the MBR of u intersects with a boundary edge of the search rectangle.
- Type 2: the MBR of u is fully contained in q.

The crux of our argument was to show that

- Claim 1: There are $O(\sqrt{N})$ nodes of Type 1.
- Claim 2: There are $O(k)$ nodes of Type 2.

Both claims are still correct! Specifically:

- For Claim 1, first note that the claim holds right after the kd-tree was built. Now it must still hold because nodes can only *disappear*, and MBRs can only *shrink*.
- For Claim 2 (which was left to you as an exercise) to hold, we only need to make sure that every internal node has two child nodes. This is guaranteed by our deletion algorithm.

□

When n is significantly less than N, two issues are created: (i) the query bound $O(\sqrt{n}+k)$ may no longer hold, and (ii) the height of $\mathcal{T}$ (which is $O(\log N)$) may cease to be bounded by $O(\log n)$. This is where global rebuilding comes in; this simple trick remedies both issues:

```
global-rebuilding
1. if $n = N/2$ then
2.     rebuild the kd-tree on the remaining $n$ points
3.     set $N = n$
```

What a drastic approach! But it works! Note that when we rebuild the kd-tree, $N/2$ deletions have taken place. Hence, the cost of rebuilding — which is $O((n/2)\log(n/2))$ — can be charged on those $N/2$ deletions, so that each deletion bears only

$$O\left(\frac{n\log n}{N}\right) = O(\log n)$$

cost. This increases the amortized cost of a deletion by only $O(\log n)$ because each deletion can be charged only once.

We thus have obtained a structure that consumes $O(n)$ space, answers a range reporting query in $O(\sqrt{n}+k)$ time, and supports a deletion in $O(\log n)$ amortized time.

5.4.2 Putting everything together

We can now slightly augment the logarithmic method (Section 5.3) to obtain a fully dynamic kd-tree.

As before, we divide the input set S of n points into disjoint subsets $S_0, S_1, ..., S_h$ (where $h = \Theta(\log n)$) satisfying $|S_i| \leq 2^i$ for every $i \in [0, h]$. Create a kd-tree $\mathcal{T}_i$ on each S_i ($i \in [0, h]$).

A query and an insertion are handled in the same way as in Sections 5.3.2 and 5.3.3, respectively. To delete a point p, we first locate the S_i (for some $i \in [0, h]$) containing p. Whether $p \in S_j$ ($j \in [0, h]$) can be decided in $O(\log n)$ time using $\mathcal{T}_j$ (we only need to descend a single root-to-leaf path in $\mathcal{T}_j$ for this purpose); and hence, S_i can be identified in $O(\log^2 n)$ time. After that, p can be deleted from $\mathcal{T}_i$ in $O(\log n)$ amortized time.

A tiny issue remains: if we have too many deletions, the value of h will cease to be bounded by $O(\log n)$. This can be taken care of again by global rebuilding (think: how?). We now have obtained a data structure of $O(n)$ space that answers a range reporting query in $O(\sqrt{n} + k)$ time, and supports an update (insertion/deletion) in $O(\log^2 n)$ amortized time.

5.5 Remarks

The logarithmic method and the global rebuilding approach were developed by Bentley and Saxe [8].

Exercises

Problem 1 (dynamic arrays). An array of size s is a sequence of s consecutive cells. In many operating systems, once the required space has been allocated to an array, accesses to the array are limited to that space (e.g., accessing the $(s+1)$-th cell will give a "segmentation fault" under Linux). Because of this, the size of an array is considered to be "fixed" by many people.

In this exercise, you are asked to partially remedy the above issue. Implement a data structure that stores a set S of n elements subject to the following requirements:

- The elements must be stored in n consecutive cells.
- The space of your structure must be $O(n)$.
- An insertion to S can be supported in $O(1)$ amortized time.

Problem 2. Tighten the loose end in Section 5.4.2, namely, what to do if h ceases to be bounded by $O(\log n)$?

Problem 3*. Improve the amortized deletion time of our fully dynamic kd-tree to $O(\log n)$.

(Hint: currently we spend $O(\log^2 n)$ amortized time on a deletion only because we don't know which tree contains the point to be deleted.)

Problem 4. Design a semi-dynamic data structure that stores a set of n intervals in $O(n)$ space, answers a stabbing query in $O(\log^2 n + k)$ time (where k is the number of intervals reported), and supports an insertion in $O(\log^2 n)$ amortized time.

Problem 5.** Design a data structure that stores a set of n intervals in $O(n)$ space, answers a stabbing query in $O(\log n + k)$ time (where k is the number of intervals reported), and supports a deletion in $O(\log n)$ amortized time. Your structure does not need to support insertions.

(Hint: the problem is extremely difficult if you try to delete nodes from the BST that defines the interval tree. But you don't have to! It suffices to update only the stabbing sets, but *not* the BST. Show that this is okay as long as you perform global rebuilding wisely.)

Problem 6.** Let S be a set of n points in $\mathbb{R}^2$ that have been *sorted* by x-coordinate. Design an algorithm to build the priority search tree on S in $O(n)$ time.

(Hint: in your undergraduate study, did you know that a max heap on n real values can actually be constructed in $O(n)$ time?)

Problem 7. Design a semi-dynamic data structure that stores a set of n 2D points in $O(n)$ space, answers a 3-sided range reporting query in $O(\log^2 n + k)$ time (where k is the number of points reported), and supports an insertion in $O(\log n)$ amortized time.

(Hint: obviously, use the result in Problem 6.)

Lecture 6: Weight Balancing

The logarithmic method in Lecture 5 has two inherent drawbacks. First, it applies only to insertions, but not deletions. Second, it needs to search $O(\log n)$ static structures in answering a query, and thus may cause a slow-down compared to the static structure. For example, applying the technique to the interval tree (Lecture 3) results in a semi-dynamic structure that answers a stabbing query in $O(\log^2 n + k)$ time, as opposed to $O(\log n + k)$ in the static case.

We will introduce a different technique called *weight balancing* that allows us to remedy the above drawbacks for many structures (including the interval tree). The technique in essence is an approach to maintain a small height for a BST under updates.

6.1 BB[α]-trees

Given a BST $\mathcal{T}$, we denote by $|\mathcal{T}|$ the number of nodes in $\mathcal{T}$. Given an arbitrary node u in $\mathcal{T}$, we represent the subtree rooted at u as $\mathcal{T}_u$. Define the *weight* of u as $|\mathcal{T}_u|$, and its *balance factor* as:

$$\rho(u) \quad = \quad \frac{\min\{|\mathcal{T}_1|, |\mathcal{T}_2|\}}{|\mathcal{T}_u|}$$

where $\mathcal{T}_1$ (or $\mathcal{T}_2$, resp.) is the left (or right, resp.) subtree of u.

Let α be a real-valued constant satisfying $0 < \alpha \le 1/5$. A node u in $\mathcal{T}$ is *α-balanced* if

- either $|\mathcal{T}_u| \le 4$
- or $\rho(u) \ge \alpha$.

In other words, either $\mathcal{T}_u$ has very few nodes (no more than 4) or each subtree of u has at least a constant fraction of the nodes in $\mathcal{T}_u$.

$\mathcal{T}$ is said to be a *BB*[α]*-tree* if every node is α-balanced (where BB stands for *bounded balanced*).

Lemma 6.1. *The height of a BB*[α]*-tree $\mathcal{T}$ is $O(\log n)$, where the big-O hides a constant factor dependent on α.*

Proof. Let $\mathcal{T}_1$ and $\mathcal{T}_2$ be the left and right subtrees of $\mathcal{T}$, respectively. By definition of BB[α], we know that $|\mathcal{T}_1| \le (1-\alpha)|\mathcal{T}|$ and $|\mathcal{T}_2| \le (1-\alpha)|\mathcal{T}|$. In other words, each time we descend into a child, the subtree size drops by a constant factor. □

Let S be the set of keys in $\mathcal{T}$; henceforth, we will consider S to be a set of real values. Set $n = |S|$.

Lemma 6.2. *If S has been sorted, a BB*[α]*-tree $\mathcal{T}$ can be constructed in $O(n)$ time.*

Proof. Take the median element $e \in S$ (i.e., the $\lceil n/2 \rceil$-smallest in S). Create a node u to store e as the key, and make u the root of $\mathcal{T}$. Each subtree of u has at least $n/2 - 1$ nodes. If $n \geq 4$, the balance factor $\rho(u) \geq \frac{n/2-1}{n} = 1/2 - 1/n \geq 1/4 > \alpha$. Therefore, u is α-balanced.

Now, construct the left subtree of u recursively on $\{e' < e \mid e' \in S\}$, and the right subtree of u recursively on $\{e' > e \mid e' \in S\}$. The above analysis implies that every node is α-balanced.

The construction time will be left as an exercise. □

Corollary 6.3. *After $\mathcal{T}$ has been constructed as in Lemma 6.2, each node with weight at least 4 has a balance factor at least* $1/4$.

Proof. Follows immediately from the proof of Lemma 6.2. □

For each node u, we store its weight along with u so that its balance factor can be calculated in constant time, once u has been identified. The space consumption of $\mathcal{T}$ remains $O(n)$.

6.2 Insertion

To insert a real value e_{new} in S, descend $\mathcal{T}$ to the leaf v whose slab (Section 2.1.2) covers e_{new}. Create a node z with e_{new} as the key, and make z a child of v. The cost so far is $O(\log n)$ by Lemma 6.1.

The insertion, however, may cause some nodes to stop being α-balanced. Such nodes can only appear on the path Π from the root to z (think: why?). Let u be the *highest* node that is no longer α-balanced. Node u, if exists, can be found in $O(\log n)$ time.

If u does not exist, the insertion finishes. Otherwise, use Lemma 6.2 to rebuild the entire $\mathcal{T}_u$. The set S_u of keys in $\mathcal{T}_u$ can be collected from $\mathcal{T}_u$ in sorted order using $O(|\mathcal{T}_u|)$ time (depth first traversal). Therefore, $\mathcal{T}_u$ can be rebuilt in $O(|\mathcal{T}_u|)$ time.

The insertion cost therefore is bounded by $O(\log n + |\mathcal{T}_u|)$, which can be terribly large. However, we will show later that rebuilding a subtree occurs infrequently such that each update is *amortized* only $O(\log n)$ time.

6.3 Deletion

To delete a real value e_{old} from S, first find the node whose key is e_{old}. For simplicity, we will consider only the case where e_{old} is the key of a leaf node z (the opposite case is left as an exercise). In that case, we simply delete z from $\mathcal{T}$. The cost so far is $O(\log n)$.

The deletion may cause some nodes to violate the α-balance requirement. Again, these nodes can only appear on the path Π from the root to z. Let u be the *highest* node that is no longer α-balanced. If u exists, rebuild $\mathcal{T}_u$ in the same way as in insertion.

The deletion cost is bounded by $O(\log n + |\mathcal{T}_u|)$. We will account for the term $|\mathcal{T}_u|$ with a charging argument in the next section.

6.4 Amortized analysis

Let us start with a crucial observation, which is the main reason for the usefulness of weight balancing:

Lemma 6.4. *Suppose that $\mathcal{T}_u$ has just been reconstructed. Let w_u be the weight of u at this moment (i.e., $w_u = |\mathcal{T}_u|$). Then, the next reconstruction of $\mathcal{T}_u$ can happen only after $w_u/24$ elements have been inserted or deleted in $\mathcal{T}_u$.*

Proof. If $w_u \le 24$, the lemma holds because trivially at least $1 \ge w_u/24 = \Omega(w_u)$ update is needed in $\mathcal{T}_u$ before the next reconstruction. Focus now on $w_u \ge 24$. By Corollary 6.3, $\rho(u) \ge 1/4$.

We argue that at least $w_u/24$ updates must have occurred in $\mathcal{T}_u$ before $\rho(u)$ drops below $\alpha \le 1/5$. Specifically, let n_1 be the number of nodes in the left subtree $\mathcal{T}_1$ of u. Hence, $n_1 \ge w_u/4$. Suppose that after x updates in $\mathcal{T}_u$, $|\mathcal{T}_1|/|\mathcal{T}_u| \le 1/5$. We will prove that $x \ge w_u/24$.

After x updates, $|\mathcal{T}_1| \ge n_1 - x$ while $|\mathcal{T}_u| \le w_u + x$. Therefore, $|\mathcal{T}_1|/|\mathcal{T}_u| \ge \frac{n_1 - x}{w_u + x}$. For the ratio to be at most $1/5$, we need:

$$\begin{aligned} \frac{n_1 - x}{w_u + x} &\le 1/5 \Rightarrow \\ 6x &\ge 5n_1 - w_u \ge w_u/4 \Rightarrow \\ x &\ge w_u/24. \end{aligned}$$

A symmetric argument shows that at least $w_u/24$ updates are needed for $|\mathcal{T}_2|/|\mathcal{T}_u| \le 1/5$ to happen, where $|\mathcal{T}_2|$ is the right subtree of u. This completes the proof.

As a remark, the above analysis paid little efforts to minimize constants. Indeed, a more careful analysis can reduce the constant 24 considerably. □

We now prove the main theorem of this lecture (review Section 5.1 for the definition of amortized cost):

Theorem 6.5. *The BB[α]-tree supports any sequence of n updates (mixture of insertions and deletions) in $O(n \log n)$ time, namely, $O(\log n)$ amortized time per update.*

Proof. It suffices to concentrate on the cost of subtree reconstruction. By Lemma 6.4, whenever a subtree $\mathcal{T}_u$ is rebuilt, we can charge the $O(|\mathcal{T}_u|)$ rebuilding cost on the $\Omega(|\mathcal{T}_u|)$ insertions/deletions that have taken place in $\mathcal{T}_u$ since the last reconstruction of $\mathcal{T}_u$ (we omitted some easy but subtle details here; can you spot them?). Each of those updates bears only $O(1)$ cost.

How many times can an update be charged this way? The answer is $O(\log n)$ because each insertion or deletion affects only $O(\log n)$ subtrees (each subtree is rooted at a node on the update path). □

6.5 Dynamization with weight balancing

The weight balancing technique can be used to dynamize all the structures in Lectures 3 and 4, except the kd-tree. Those structures have the common properties below:

- They use a BST $\mathcal{T}$ as the *primary* structure.
- Every node in $\mathcal{T}$ is associated with a *secondary* structure.

They are difficult to update because each secondary structure may be very large. Hence, when a node u of $\mathcal{T}$ changes its place in $\mathcal{T}$ (e.g., rotations in AVL-trees), we must pay a huge cost to rebuild its secondary structure, resulting in large update overhead. The weight-balancing technique remedies this issue effectively because subtree reconstructions occur very infrequently (Lemma 6.4).

To illustrate the core ideas, next we will describe an extension of the BST, which also has the two properties above, and would be exceedingly difficult to update before this lecture. Weight balancing, however, makes dynamization almost trivial. The same ideas apply to other structures as well.

6.5.1 Dynamic arrays

Let S be a set of n elements. A *dynamic array* $\mathcal{A}$ on S is an array satisfying the following:

- $\mathcal{A}$ has size $O(n)$.
- The elements of S are stored in the first n positions of $\mathcal{A}$ (ordering is not important).
- $\mathcal{A}$ supports an insertion on S in $O(1)$ amortized time.

The design of a dynamic array was an exercise of Lecture 5.

6.5.2 A BST augmented with dynamic arrays

Let $\mathcal{T}$ be a BST created on a set S of n real values. For each node u in $\mathcal{T}$, denote by $\mathcal{T}_u$ the subtree rooted at u, and by S_u the set of keys in $\mathcal{T}_u$. Create a dynamic array $\mathcal{A}_u$ on S_u, and make $\mathcal{A}_u$ the secondary structure of u.

Let us try to support insertions while maintaining the $O(\log n)$ height of $\mathcal{T}$. It is easy to update $\mathcal{T}$ itself, e.g., using rotations as in the AVL-tree. However, when rotation changes the position of a node u in $\mathcal{T}$, S_u can change significantly, and hence, so can $\mathcal{A}_u$. The size $|S_u|$ can be very large (in the worst case, $\Omega(n)$), because of which rebuilding $\mathcal{A}_u$ will incur terrible update cost!

6.5.3 Replacing the BST with a BB[α]-tree

Now, redefine $\mathcal{T}$ to be a BB[α]-tree on S. The meanings of $\mathcal{T}_u$, S_u, and $\mathcal{A}_u$ are the same as before. An insertion can now be supported easily in $O(\log^2 n)$ amortized time.

Given a real value e_{new}, we first create a new leaf z in $\mathcal{T}$ with e_{new} as the key. This takes $O(\log n)$ time by following a root-to-z path Π. For every node u on Π, e_{new} is inserted into $\mathcal{A}_u$ in $O(1)$ amortized time. The cost so far is $O(\log n)$ amortized.

The insertion procedure of the BB[α]-tree (Section 6.2) may need to reconstruct the subtree $\mathcal{T}_u$ of a node u on Π. When this happens, we simply reconstruct all the secondary arrays in $\mathcal{T}_u$ in $O(|S_u| \log |S_u|) = O(|S_u| \log n)$ time. By Lemma 6.4, $\Omega(|S_u|)$ updates must have taken place in $\mathcal{T}_u$ since the last reconstruction of $\mathcal{T}_u$. Each of those updates is therefore charged only $O(\log n)$ time for the reconstruction of $\mathcal{A}_u$.

An update can be charged only $O(\log n)$ times (e_{new} is in $O(\log n)$ subtrees), and hence, has amortized cost $O(\log^2 n)$.

6.6 Remarks

Our definition is one of the many ways to describe the BB[α] tree. See [34] for the original proposition.

The BB[α]-tree (our version in Section 6.1) can actually be updated in $O(\log n)$ *worst-case* time. Whenever a node u stops being α-balanced, it can be fixed in $O(1)$ time by either a single-rotation or a double-rotation, just like the AVL-tree. Even better, after a fix on a node u, the node u can violate α-balance only after $\Omega(w_u)$ updates have taken place in $\mathcal{T}_u$. The details can also be found in [9].

The structure in Section 6.5.2 is useful for *independent query sampling* [24].

Exercises

Problem 1. Prove the construction time in Lemma 6.2.

Problem 2. Complete the deletion algorithm for the case where e_{old} is the key of an internal node. (Hint: the "standard" strategy for BSTs suffices.)

Problem 3 (dynamic arrays with deletions). Let S be a set of n elements. Design a data structure with the properties below:

- the structure stores an array $\mathcal{A}$ of size $O(n)$.
- The elements of S are stored in the first n positions of $\mathcal{A}$ (ordering is not important).
- An insertion/deletion on S can be supported in $O(\log n)$ amortized time.

Problem 4. Explain how to support an insertion/deletion on the structure of Section 6.5.3 in $O(\log^2 n)$ amortized time.

Problem 5. Explain how to support an insertion/deletion on the interval tree (Section 3.1) in $O(\log^2 n)$ amortized time, where n is the number of intervals. Your structure must still be able to answer a stabbing query in $O(\log n + k)$ time, where k is the number of intervals reported.

Problem 6. Explain how to support an insertion/deletion on the priority search tree (Section 4.3) in $O(\log^2 n)$ amortized time, where n is the number of points. Your structure must still be able to answer a 3-sided range query in $O(\log n + k)$ time, where k is the number of points reported.

Problem 7*. Improve the update time in the previous problem to $O(\log n)$.

Problem 8. Explain how to support an insertion/deletion on the range tree (Section 4.4) in $O(\log^3 n)$ amortized time, where n is the number of points. Your structure must still be able to answer a 4-sided range query in $O(\log^2 n + k)$ time, where k is the number of points reported.

Lecture 7: Partial Persistence

A dynamic data structure is usually *ephemeral* because, once updated, its previous version is lost. For example, consider n insertions into an initially empty BST. At the end, we have a BST with n nodes (the final version). However, $n-1$ other versions had been created in history (one after each of the first $n-1$ insertions); all those versions have been lost.

Wouldn't it be nice if we could retain all versions? One naive approach to do so is to store a separate copy of each past version, which requires $O(n^2)$ space. Amazingly, we will learn a powerful technique called *partial persistence* that allows us to achieve the purpose in just $O(n)$ space (which is clearly optimal).

The technique in fact is applicable to *any* pointer-machine structure (Section 4.6), as long as each node in the structure has a constant in-degree (for the BST, the in-degree is 1). This includes most of the structures you already know: the linked list, the priority queue, all the structures in Lectures 3 and 4, and so on (but not dynamic arrays).

The implication of this technique goes beyond just retaining history. It can be used to solve difficult problems using surprisingly primitive structures. One example is the 3-sided range query that we dealt with using the priority search tree in Section 4.3. As we will see in an exercise, that problem can be settled by simply making the BST partially persistent.

7.1 The potential method

This is a generic method for *amortized* analysis we will apply later. As it may not have been covered at the undergraduate level, we include an introduction here.

Consider M operations on a data structure, the i-th ($1 \le i \le M$) of which has cost C_i. Suppose that we assign to the i-th operation a non-negative integer $\bar{C}_i$. Following the discussion in Section 5.1, we can claim that the i-th operation has amortized cost $\bar{C}_i$ if

$$\sum_{i=1}^{M} C_i \le \sum_{i=1}^{M} \bar{C}_i.$$

Define a function Φ which maps the current structure to a real value. Let T_0 be the structure before all operations, and T_i ($1 \le i \le M$) be the structure after operation i. Define for each $i \in [1, M]$:

$$\Delta_i = \Phi(T_i) - \Phi(T_{i-1}). \tag{7.1}$$

We can now claim:

Lemma 7.1. *If $\Phi(T_M) \ge \Phi(T_0)$, the amortized cost of operation i is at most $C_i + \Delta_i$.*

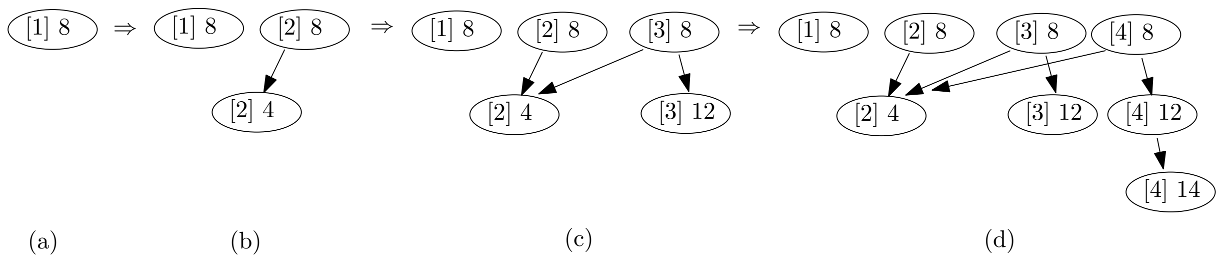


Figure 7.1: Illustration of naive copying on the insertion sequence of 8, 4, 12, 14.

Proof. It suffices to prove $\sum_{i=1}^{M} C_i \leq \sum_{i=1}^{n}(C_i + \Delta_i)$. This is obvious because

$$\sum_{i=1}^{M} \Delta_i \quad = \quad \Phi(T_M) - \Phi(T_0) \geq 0.$$

$\square$

Why do we want to claim that the amortized cost is $C_i + \Delta_i$, instead of C_i? This is because Δ_i can be negative! Indeed, a successful argument under the potential method must be able to assign a negative Δ_i to offset *every* large C_i.

It is worth mentioning that Φ is called a *potential function*.

7.2 Partially persistent BST

Starting with an empty BST $\mathcal{T}_0$, we will process a sequence of n updates (mixture of insertions and deletions). The i-th ($1 \leq i \leq n$) update is said to happen at *time* i. Denote by $\mathcal{T}_i$ the BST after the update, which is said to be of *version* i. Our goal is to retain the BSTs of all versions.

We will refer to the BST of the latest version as the *live* BST, and denote it as $\mathcal{T}$. In other words, after i updates, the live BST is $\mathcal{T} = \mathcal{T}_i$.

Denote by $\mathcal{A}$ the update algorithm of the BST, which can be any implementation of the BST, e.g., the AVL-tree, the red-black tree, the BB[α]-tree, etc.

7.2.1 The first attempt

Our first idea is to enforce the principle that *whenever $\mathcal{A}$ needs to change a node u, make a copy of u, and apply the changes on the new copy.*

Example. Consider the update sequence that inserts 8, 4, 12, and 14. Figure 7.1(a) shows the live BST $\mathcal{T} = \mathcal{T}_1$, which contains a single node. The node information has the format "[i] k", to indicate that the node is created at time i with key k.

To perform the second insertion, we create a node "[2] 4", and need to make it the left child of "[1] 8". Following the aforementioned principle, "[1] 8" is not altered; instead, we copy it to "[2] 8", and set "[2] 4" the left child of "[2] 8". As shown in Figure 7.1(b), both BSTs $\mathcal{T}_1$ and $\mathcal{T}_2$ are explicitly stored.

To insert 12, we create a node "[3] 12", which ought to be the right child of "[2] 8". Following the principle, we copy "[2] 8" to "[3] 8", and set "[3] 12" as the right child of "[3] 8". The structure now is shown in Figure 7.1(c). Note that the left child of "[3] 8" is still "[2] 4" ("[2] 4" is *not* copied because it is not modified by this update). Observe that Figure 7.1(c) implicitly stores 3 BSTs $\mathcal{T}_1$, $\mathcal{T}_2$, $\mathcal{T}_3$.

Figure 7.1(d) presents the final structure after inserting 14, which encodes BSTs $\mathcal{T}_1, ..., \mathcal{T}_4$. □

We will call the above method *naive copying*. Since each update on the live BST accesses $O(\log n)$ nodes, naive copying can create $O(\log n)$ nodes per update in the persistent structure. The overall space consumption is therefore $O(n \log n)$.

Any BST in the past can be found and searched efficiently. For any $i \in [1, n]$, the root of $\mathcal{T}_i$ can be identified in $O(\log n)$ time (by creating a separate BST on the root versions). Then, the search can proceed within $\mathcal{T}_i$ in same manner as a normal BST is searched.

The drawback of naive copying is that it sometimes copies a node that is not modified by $\mathcal{A}$. In Figure 7.1(d), for example, the only node that "really" needs to be modified is "[3] 12" but the method duplicates all its ancestors. This motivates our next improvement.

7.2.2 An improved method

The new idea is to introduce a *modification field* in each node u. When $\mathcal{A}$ needs to change the pointer of u, the change is recorded in the field. Only when the field has no more room will we resort to node copying. It turns out that a field of constant size suffices to reduce the space to $O(n)$.

Each node now takes the form $\{([i]\, k, ptr_1, ptr_2), mod\}$ where

- the first component $([i]\, k, ptr_1, ptr_2)$ indicates that the node is created at version i with key k and pointers ptr_1 and ptr_2 (which may be NULL);
- the second component mod is the modification field, which is empty when the node is created, and can log *exactly one* pointer change.

Example. We will first insert 8, 4, 12, 14, 2, and then delete 2, 14. Figure 7.2(a) shows the structure after the first insertion. Here, the ptr_1 and ptr_2 of node I are both NULL. The empty space on the right of the vertical bar indicates an empty mod.

To insert 4, we create node II, and make it the left child of node I. This means redirecting the left pointer of node I to node II at time 2. This pointer change is described in the mod of node I, as shown in Figure 7.2(b). Observe how the current structure encodes both $\mathcal{T}_1$ and $\mathcal{T}_2$.

The insertion of 12 creates node III, which should be the right child of node I. As the mod of node I is already *full*, we cannot log the pointer change inside node I. We thus resort to node copying. As shown in Figure 7.2(c), this spawns node IV which stores "[3] 8", and has ptr_1 and ptr_2 referencing nodes II and III, respectively. The current structures encodes $\mathcal{T}_1, \mathcal{T}_2$, and $\mathcal{T}_3$.

14 and 2 are then inserted in the same manner, as illustrated by Figures 7.2(d) and (e), respectively.

The next operation deletes 2. Accordingly, we should reset the pointer of node II to NULL (which removes node VI from the live tree). Since node II's mod is full, we copy it to node VII. This, in turn, requires changing the left pointer of node IV, as is recorded in its mod. The current structure in Figure 7.2(f) encodes $\mathcal{T}_1, ..., \mathcal{T}_6$.

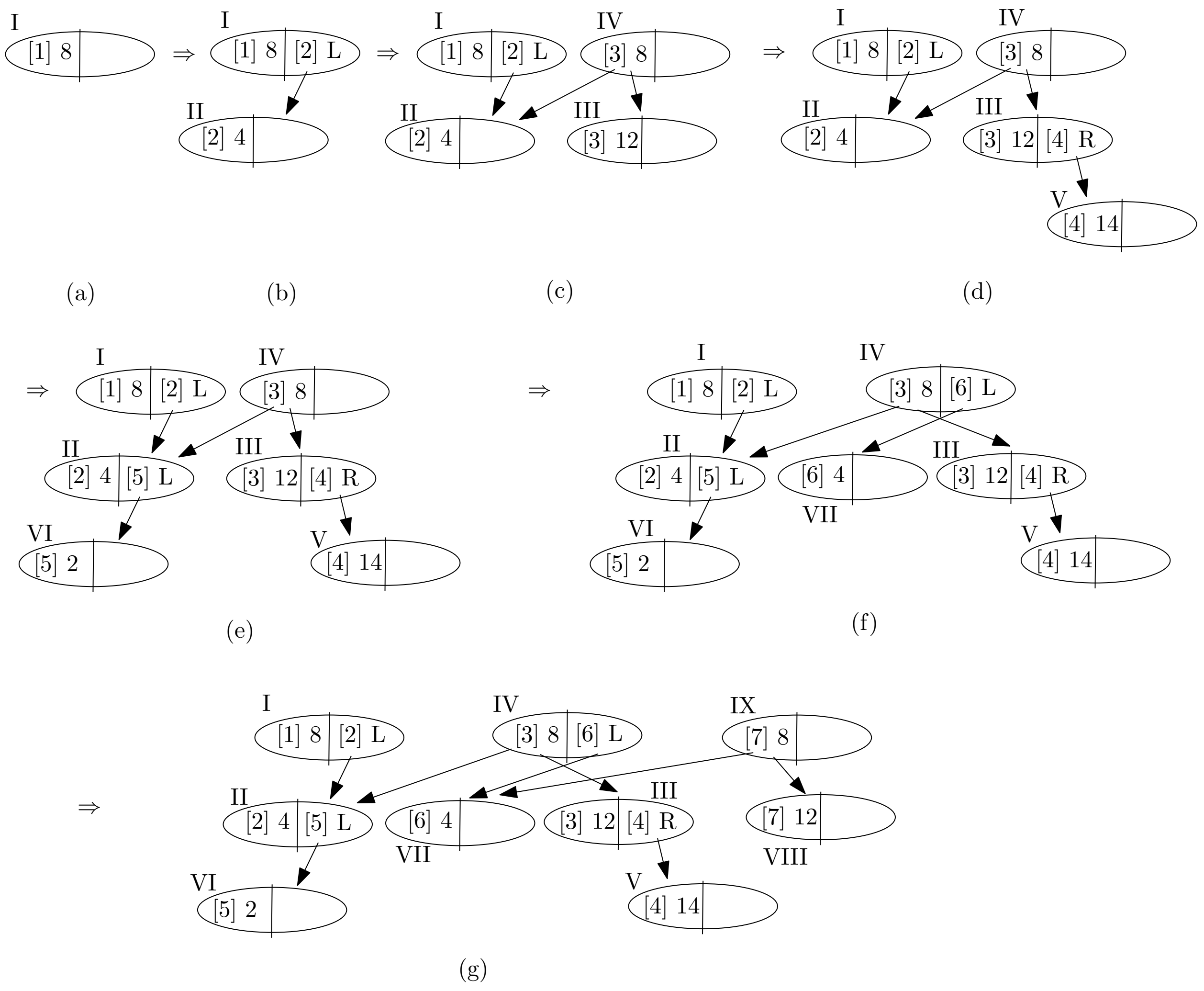


Figure 7.2: Illustration of the improved method on the update sequence of inserting 8, 4, 12, 14, 2 followed by deleting 2 and 14.

Finally, the deletion of 14 requires nullifying the right pointer of node III. As Node III's *mod* is full, it is copied to node VIII, which further triggers node IV to be copied to node IX. Figure 7.2(g) gives the final structure which encodes $\mathcal{T}_1, ..., \mathcal{T}_7$. □

In general, $\mathcal{A}$ can change the live BST with two operations:

- C-operation: creating a new node u. This happens only in insertion, and the node created stores the key being inserted. Accordingly, we also create a node in the persistent structure.
- P-operation: updating a pointer of a node u. We do so in the persistent structure as follows:

```
ptr-update($u$)
1. if the mod of $u$ is empty then
2.     record the pointer change in mod
3.     return
```

```
4. else /* mod full */
5.     copy $u$ to node $v$
6.     $\hat{u} \leftarrow$ the parent of $u$ in the live BST
7.     call ptr-update($\hat{u}$) to add a pointer from $\hat{u}$ to $v$
```

Note that Line 7 may recursively invoke **ptr-update** and thereby induce multiple node copies.

The time to build a persistent BST is clearly $O(n \log n)$ (the proof is left to you as an exercise). As in Section 7.2.1, we can identify the root of any $\mathcal{T}_i$ $(1 \leq i \leq n)$ in $O(\log n)$ time, after which $\mathcal{T}_i$ can then be navigated as a normal BST.

We will analyze the space consumption in the next subsection.

7.2.3 Space

Denote by m_i $(1 \leq i \leq n)$ the number of C/P-operations that $\mathcal{A}$ performs on the live tree in processing the i-th update. We will prove:

Lemma 7.2. *The algorithm in Section 7.2.2 creates $O(\sum_{i=1}^{n} m_i)$ nodes in the persistent tree.*

The lemma immediately implies:

Theorem 7.3. *Given a sequence of n updates on an initially empty BST, we can build a persistent BST of $O(n)$ space in $O(n \log n)$ time.*

Proof. The red-black tree performs at most one C-operation and $O(1)$ P-operations in each insertion/deletion. □

Proof of Lemma 7.2. Set

$$M = \sum_{i=1}^{n} m_i$$

namely, M is the total number of C/P-operations performed by $\mathcal{A}$. These operations happen in succession, and hence, can be ordered as operation 1, 2, ..., M.

Let C_j $(1 \leq j \leq M)$ be the number of nodes (in the persistent tree) created by the j-th operation. We will prove $\sum_{j=1}^{M} C_j = O(M)$, or equivalently, each operation creates $O(1)$ nodes amortized.

Denote by S_j $(1 \leq j \leq M)$ the set of nodes in the live tree after the j-th operation. Define specially S_0 the empty set. Define a potential function Φ that maps S_j to a real value; specifically, $\Phi(S_j)$ equals the number of nodes in S_j whose information fields are *non*-empty. Clearly, $\Phi(S_M) \geq \Phi(S_0) = 0$.

By Lemma 7.1, after amortization operation j creates at most

$$C_j + \Phi(S_j) - \Phi(S_{j-1}) \tag{7.2}$$

nodes. The remainder of the proof will show that the above is precisely 1 for every j, which will complete the proof of Lemma 7.2.

If operation j is a C-operation, it creates a node with an empty *mod* and finishes. Hence, $C_j = 1$, and $S_j = S_{j-1}$. Therefore, (7.2) equals 1.

Now, consider that operation j is a P-operation. Every new node is created by node copying (Line 5 of **ptr-update**). However, every time this happens, we lose a node with non-empty *mod*, and create a node with empty *mod*. At the end of the P-operation, we fill in the *mod* of one node, thus converting it from a node with empty *mod* to one with non-empty *mod*.[1] Therefore, $\Phi(S_j) - \Phi(S_{j-1})$ equals precisely $-C_j + 1$ such that (7.2) also equals 1.

7.3 General pointer-machine structures

The following result generalizes Theorem 7.3:

Theorem 7.4 ([16]). *Consider any pointer-machine structure defined in Section 4.6 where every node has a constant in-degree. Suppose that $\mathcal{A}$ is an algorithm used to process a sequence of n updates (mixture of insertions and deletions) with amortized update cost $U(n)$. Let m_i be the number of nodes created/modified by $\mathcal{A}$ in processing the i-th update ($1 \le i \le n$). Then, we can create a persistent structure that records all the historical versions in $O(n \cdot U(n))$ time. The structure consumes $O(\sum_{i=1}^{n} m_i)$ space. The root of every version can be identified in $O(\log n)$ time.*

For example, if the structure is the linked list, then $U(n) = O(1)$ and $m_i = O(1)$. Therefore, we can construct a persistent linked list of $O(n)$ space in $O(n)$ time. The head node of the linked list of every past version can be identified in $O(\log n)$ time.

The theorem can be established using the modification-logging approach in Section 7.2.2, except that the modification field should be made sufficiently large (but still have a constant size). The proof makes an interesting, but not compulsory, exercise.

7.4 Remarks

The methods in this lectures were developed by Driscoll, Sarnak, Dominic, and Tarjan in [16].

[1]At the end of a P-operation, we may also need to create a node, which can regarded as a C-operation.

Exercises

Problem 1. Prove the construction time in Theorem 7.3.

Problem 2. Let S be a set of n horizontal rays in $\mathbb{R}^2$, each having the form $[x, \infty) \times y$. Explain how to store S in a persistent BST of $O(n)$ space such that, given any vertical segment $q = x \times [y_1, y_2]$, we can report all the rays in S intersecting q using $O(\log n + k)$ time, where k is the number of rays reported.

Problem 3. Let P be a set of n points in $\mathbb{R}^2$. Explain how to store P in a persistent BST of $O(n)$ space such that any 3-sided range query of the form $(-\infty, x] \times [y_1, y_2]$ can be answered in $O(\log n + k)$ time, where k is the number of points reported. (Hint: Problem 2.)

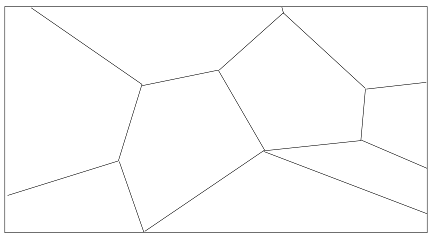
Problem 4. Let P be a set of n points in $\mathbb{R}^2$. Given an axis-parallel rectangle q, a *range count* query reports the number of points in P that are covered by q. Design a structure that stores P in $O(n \log n)$ space that can answer a range count query in $O(\log n)$ time.

Remark: this improves an exercise in Lecture 4. (Hint: persistent count BST.)

Problem 5. Prove Theorem 7.4 for the linked list.

Remark: the persistent linked list is one way to store all the past versions of a document that is being edited (regard a document as a sequence of characters.)

Problem 6* (point location). A polygonal subdivision of $\mathbb{R}^2$ is a set of non-overlapping convex polygons whose union is $\mathbb{R}^2$. The following shows an example (for clarity, the boundary of $\mathbb{R}^2$ is represented as a rectangle).



Given a point q in $\mathbb{R}^2$, a *point location* query reports the polygon that contains q (if q falls on the boundary of more than one polygon, any such polygon can be reported).

Let n be the number of segments in the subdivision. Design a structure of $O(n)$ space that can answer any point location query in $O(\log n)$ time. (Hint: persistent BST.)

Problem 7.** Prove Theorem 7.4.

Lecture 8: Dynamic Perfect Hashing

In the *dictionary search* problem, we want to store a set S of n integers in a data structure to answer the following queries efficiently: given an integer q, report whether $q \in S$ (the output is boolean: yes or no). At the undergraduate level, we have learned that the problem can be tackled with *hashing*. Specifically, we can store S in a hash table of $O(n)$ space which answers a query in $O(1)$ expected time.

In practice, we may not be satisfied with $O(1)$ *expected* query cost because it implies that the actual search time can still be large occasionally. Ideally, we would like to build a *perfect* hash table that guarantees $O(1)$ query cost in the *worst* case.

This lecture will introduce a technique called *cuckoo hashing* which can be used to maintain a perfect hash table of $O(n)$ size with $O(1)$ amortized expected time per update (what this means will be defined formally later). We will, however, establish only a weaker bound of $O(\log n)$; as a benefit in return, this illustrates nicely how data structures can arise from graph theory.

8.1 Two random graph results

Let U, V each be a set of $c \cdot n \geq 2$ vertices, for some integers $c > 0, n > 0$. We generate a random bipartite graph G by repeating the **gen-edge** operation n times:

gen-edge
1. pick a vertex $u \in U$ uniformly at random
2. pick a vertex $v \in V$ uniformly at random
3. connect u, v with an edge (if the edge does not exist)

Lemma 8.1. *When $c \geq 4e^2$, it holds with probability at least 7/8 that G contains no cycles.*

Proof. Consider any integer $\ell \in [4, 2cn]$. We will prove an upper bound on the probability that G has a cycle of length ℓ. Let us start with:

$$\mathbf{Pr}[\text{a cycle of length } \ell] \;\leq\; \sum_{\Sigma} \mathbf{Pr}[\Sigma \text{ induces a cycle in } G] \tag{8.1}$$

where the summation is over all $\Sigma \subseteq U \cup V$ with $|\Sigma| = \ell$.

Next, we will analyze $\mathbf{Pr}[\Sigma$ induces a cycle in $G]$ for an arbitrary $\Sigma = \{u_1, u_2, ..., u_\ell\}$.

$$\begin{aligned}&\mathbf{Pr}[\Sigma \text{ induces a cycle in } G] \\ &\quad\leq \sum_{I} \mathbf{Pr}[\text{all } \textbf{gen-edge} \text{ operations indexed by } I \text{ create an edge on } \Sigma]\end{aligned} \tag{8.2}$$

where the summation is over all $I \subseteq \{1, 2, ..., n\}$ with $|I| = \ell$; and the set of operations indexed by I is $\{\text{operation } i \mid i \in I\}$.

gen-edge creates an edge on Σ with probability at most $(\frac{\ell}{cn})^2$ because both u and v it chooses must fall in Σ. It follows that

$$\mathbf{Pr}[\text{all } \textbf{gen-edge} \text{ operations indexed by } I \text{ create an edge on } \Sigma] \leq \left(\frac{\ell}{cn}\right)^{2\ell}$$

and hence by (8.2)

$$\mathbf{Pr}[\Sigma \text{ induces a cycle in } G] \leq \binom{n}{\ell} \cdot \left(\frac{\ell}{cn}\right)^{2\ell}$$

with which (8.1) gives

$$\begin{aligned}\mathbf{Pr}[\text{a cycle of length } \ell] &\leq \binom{2cn}{\ell} \cdot \binom{n}{\ell} \cdot \left(\frac{\ell}{cn}\right)^{2\ell} \\ &\leq \left(\frac{e \cdot 2cn}{\ell}\right)^{\ell} \cdot \left(\frac{e \cdot n}{\ell}\right)^{\ell} \cdot \left(\frac{\ell}{cn}\right)^{2\ell} \\ &= \left(\frac{2e^2}{c}\right)^{\ell} \leq (1/2)^{\ell}.\end{aligned}$$

We now prove the lemma with

$$\begin{aligned}\mathbf{Pr}[G \text{ has a cycle}] &\leq \sum_{\ell=4}^{2cn} \mathbf{Pr}[\text{a cycle of length } \ell] \\ &\leq \sum_{\ell=4}^{2cn} (1/2)^{\ell} < 1/8.\end{aligned}$$

□

An almost identical argument establishes:

Lemma 8.2. *When $c \geq 4e^3$, it holds with probability at least $1 - \frac{c}{n^2}$ that G has no simple path longer than $4 \log_2 n$ edges (a path is simple if it passes no vertex twice).*

The proof is left as an exercise.

8.2 Cuckoo hashing

8.2.1 Amortized expected update cost

Suppose that a structure processes n_{op} updates. As mentioned in Section 5.1, we can claim that the i-th ($1 \leq i \leq n_{op}$) update has *amortized* cost $\bar{C}_i$ if

$$\sum_{i=1}^{n_{op}} C_i \leq \sum_{i=1}^{n_{op}} \bar{C}_i,$$

where C_i is the actual cost of the i-th update.

Now consider the structure to be randomized such that each C_i is a random variable. In this case, we can claim that the i-th ($1 \le i \le n_{op}$) update has *amortized expected* cost $\bar{C}_i$ if $\mathbf{E}[\sum_{i=1}^{n_{op}} C_i] \le \sum_{i=1}^{n_{op}} \bar{C}_i$, which means

$$\sum_{i=1}^{n_{op}} \mathbf{E}[C_i] \;\; \le \;\; \sum_{i=1}^{n_{op}} \bar{C}_i.$$

For example, if the structure has $O(1)$ amortized expected update time, it processes any n updates in $O(n)$ expected total time.

8.2.2 Hash functions

Denote by $\mathbb{D}$ the domain from which the elements of S are drawn. A *hash function* h maps $\mathbb{D}$ to a set of integers $\{1, 2, ..., N\}$ for some $N \ge 1$. The output $h(e)$ is the *hash value* of $e \in \mathbb{D}$. We will assume *uniform hashing*, which means:

- for any element $e \in \mathbb{D}$, $\mathbf{Pr}[h(e) = i] = 1/N$ for any $i \in [1, N]$;
- the above holds regardless of the hash values of the other elements in $\mathbb{D}$.

8.2.3 The hash table, query, and deletion

We maintain two arrays A, B each of size $N = O(n)$ where the concrete value of N will be chosen later. There are two hash functions g and h, both mapping $\mathbb{D}$ to $\{1, ..., N\}$. We enforce:

Invariant: Each element $e \in S$ is stored at either $A[g(e)]$ or $B[h(e)]$.

This makes querying and deletions very simple:

- **Query:** Given an element q, report yes if $A[g(e)]$ or $B[h(e)] = q$; otherwise, report no.
- **Deletion:** To delete an element $e \in S$, erase $A[g(e)]$ or $B[h(e)]$ whichever equals e.

Clearly, both operations finish in $O(1)$ worst-case time.

8.2.4 Insertion

To insert an element e_{new}, if $A[g(e_{new})]$ is empty, we store e_{new} at $A[g(e_{new})]$ and finish. Otherwise, if $B[h(e_{new})]$ is empty, we store e_{new} at $B[h(e_{new})]$ and finish.

If both $A[g(e_{new})]$ and $B[g(e_{new})]$ are occupied, we launch a *bumping process* which can be intuitively understood as follows. Remember every element $e \in S$ has two "nests": $A[g(e)]$ and $B[h(e)]$. If e is evicted from one nest, we are obliged to store it in the other. With this mentality, let us put e_{new} at $A[g(e_{new})]$ *anyway*, thus forcing the original element e there to be evicted. Thus, e must go into its other nest in B, thereby evicting another element there. The process then goes on until all the elements have been placed properly. There is a chance that this may not be possible, in which case we declare failure.

Formally, we begin the bumping process by calling **bump**(e_{new}):

```
**bump**($e$)
1.  turn = g; $cnt = 0$
2.  **while** $cnt \le 4\log_2 n$ **do**
3.      $cnt$++
4.      **if** turn = g **then**
5.          **if** $A[g(e)]$ empty **then**
6.              place $e$ at $A[g(e)]$; **return** success
            **else**
7.              swap $e$ and $A[g(e)]$; turn = h
        **else**
        /* turn = h */
8.          **if** $B[h(e)]$ empty **then**
9.              place $e$ at $B[h(e)]$; **return** success
            **else**
10.             swap $e$ and $B[h(e)]$; turn = g
11. **return** failure
```

Note that functions g and h are used in a round-robin fashion.

Example. Set $N = 4$. Currently, S contains a single element 10. Suppose that $g(10) = 2$, $h(10) = 3$, and that 10 is stored in $A[g(10)] = A[2]$.

Consider 16 as the next insertion, for which we assume $g(16) = 2$ and $h(16) = 4$. In other words, $A[g(16)] = A[2]$ is occupied, but $B[h(16)] = B[4]$ is empty. We thus store 16 at $B[4]$, and finish. Now $A = (-, 10, -, -)$ and $B = (-, -, -, 16)$.

The next insertion is 35, for which we assume $g(35) = 2$ and $h(35) = 4$. As both $A[g(35)]$ and $B[h(35)]$ are both occupied, a bumping process is launched. The process stores 35 at $A[g(35)] = A[2]$, and forces the original element 10 there to relinquish its place. For element 10, we find $B[h(10)] = B[3]$ empty, and thus store 10 there. The insertion finishes with $A = (-, 35, -, -)$ and $B = (-, -, 10, 16)$.

We receive one more insertion 29 with $g(29) = 2$ and $h(29) = 4$. As $A[2]$ and $B[4]$ are occupied, the bumping process removes 35 from $A[g(29)] = A[2]$, and stores 29 there instead. Currently, $A = (-, 29, -, -)$. As $h(35) = 4$, the process replaces 16 with 35 at $B[4]$, turning B to $(-, -, 10, 35)$. The process then puts 16 at $A[g(16)] = A[2]$, and remove element 29, after which $A = (-, 16, -, -)$. The process continues in this manner and eventually declares failure. $\square$

If the bumping process fails, we simply rebuild the whole structure:

```
**rebuild**
1.  choose another two hash functions $g$ and $h$
2.  insert the elements of $S$ one by one, and stop if failure declared
3.  **if** Line 2 fails **then**
4.      repeat from Line 1
```

Note that Line 2 can fail because it may invoke the bumping process.

8.2.5 Checkpoint rebuild

We ensure the following constraint on the array size N:

$$2e^3 \cdot n \le N \le 8e^3 \cdot n. \tag{8.3}$$

This can be achieved with global rebuilding (Section 5.4). Specifically, before processing the first update (which must be an insertion), we place a *checkpoint*, and set N to $4e^3$. In general, after $N/(8e^3)$ updates since the previous checkpoint, another checkpoint occurs, such that we reconstruct the structure by calling **rebuild** (Section 8.2.4) with array size $N = 4e^3 \cdot n$ (where n the current size of S).

Lemma 8.3. *Equation* (8.3) *holds at all times.*

Proof. Let n_{chk} be the size of S at the last checkpoint. There can be at most $n_{chk} + N/(8e^3) = 1.5n_{chk}$ distinct elements in S till the next checkpoint. Hence, at all times we have $2e^3 \cdot n \le 2e^3 \cdot 1.5n_{chk} < N$. On the other hand, there are at least $n_{chk} - N/(8e^3) = 0.5n_{chk}$ elements in S till the next check point. Hence, at all times we have $n \ge 0.5n_{chk} = N/(8e^3)$. □

8.3 Analysis

This section will prove:

Theorem 8.4. *Fix any sequence of n updates (mixture of insertions and deletions). The above algorithm maintains a perfect hash table under the updates in $O(n \log n)$ total expected time.*

The core of the proof is to establish:

Lemma 8.5. *Consider any checkpoint. Let N be the array size set at the checkpoint. The total cost of the following tasks is $O(N \log N)$ expected:*

- *rebuilding the structure at the checkpoint;*
- *performing the next $N/(8e^3)$ updates (i.e., until the next checkpoint).*

The lemma implies Theorem 8.4. To see this, first notice that The lemma implies $\Omega(N)$ updates between the previous checkpoint and the current one. Therefore, we can charge the $O(N \log N)$ cost on those updates such that each is amortized only $O(\log N)$ cost expected.

8.3.1 Proof of Lemma 8.5

We will prove only the first bullet, because essentially the same argument also proves the second bullet, which is left as an exercise.

We start by establishing a vital connection between cuckoo hashing and random graphs. Set $U = V = \{1, 2, ..., N\}$. For each an element $e \in S$, create an edge between vertex $g(e) \in U$ and vertex $h(e) \in V$. Let G be the bipartie graph obtained. As $g(e)$ (or $h(e)$, resp.) chooses each vertex in U (or V, resp.) with the same probability, G is precisely a random graph obtained in Section 8.1.

Corollary 8.6. *With probability at least* $1/2$, *G has both the properties below:*

- *G has no cycles.*
- *G has no simple path of longer than $4 \log_2 n$ edges.*

Proof. Consider first $n \le 16$. In this case, G obviously cannot have a simple path of length $4\log_2 n = 16$ because such a path needs 17 vertices. By Lemma 8.1, G has the first property with probability at least 7/8.

Consider now $n > 16$. Lemma 8.1 shows that the first property can be violated with probability at most 1/8. Since (8.3) always holds, Lemma 8.2 indicates that the second property can be violated with probability at most $c/n^2 = (4e^3)/n^2 \le (4e^3)/16^2 < 1/3$ (at the check point we choose the array size $N = 4e^3 \cdot n^2$; hence, $c = 4e^3$). Hence, the probability for at least one property to be violated is no more than $1/8 + 1/3 < 1/2$. □

Lemma 8.7. *Line 2 of* **rebuild** *(Section 8.2.4) takes $O(n \log n)$ time.*

Proof. Line 2 performs n insertions at Line 2. Each insertion takes $O(1)$ time if no bumping process is required. Otherwise, it takes $O(\log n)$ time before declaring success or failure. □

Lemma 8.8. *If G has both properties in Corollary 8.6, Line 2 of* **rebuild** *successfully builds the entire structure.*

Proof. We will prove that, with the two properties, the bumping process will *never* fail. This will establish the lemma.

When the bumping process evicts an element e from one nest to the other — say from $A[g(e)]$ to $B[h(e)]$ — we cross an edge in G from vertex $g(e) \in U$ to $h(e) \in V$. Therefore, if the process fails, we must have traveled on a path Π of more than $4\log_2 n$ edges.

Wait! Since G has no cycles, Π cannot pass two identical vertices. But then Π must be a *simple* path, which is not possible either. This creates a contradiction. □

We can now put together Corollary 8.6, Lemmas 8.7 and 8.8 to prove that **rebuild** finishes in $O(n \log n)$ expected time. Let X be the number of times that Line 2 is executed. By Corollary 8.6 and Lemma 8.8, every time Line 2 is executed, it fails with probability at most 1/2, which indicates that $\mathbf{Pr}[X = t] \le (1/2)^{t-1}$. Lemma 8.7 implies that the total cost of **rebuild** is $O(X \cdot n \log n)$. Therefore, the expected cost is

$$\sum_{t=1}^{\infty} O(t \cdot n \log n) \cdot \mathbf{Pr}[X = t] \;=\; \sum_{t=1}^{\infty} O(t \cdot n \log n) \cdot \left(\frac{1}{2}\right)^{t-1} = O(n \log n).$$

This completes the proof of the first bullet of Lemma 8.5.

8.4 Remarks

Our discussion of cuckoo hashing emphasized on its relationships to random graphs. The analysis presented, however, is loose in at least two ways. First, as mentioned, cuckoo hashing actually achieves $O(1)$ amortized expected time per update. Second, our hash table can have a size up to $8e^3 n$ (see (8.3)), which can be reduced considerably. We refer the interested students to the original paper [36] by Pagh and Rodler.

Our assumption of uniform hashing can also be relaxed, although this takes more efforts. As shown in [36], $O(\log n)$-wise independent hashing (i.e., intuitively this means that any $O(\log n)$ hash values are guaranteed to be independent; our assumption is essentially n-wise independence) is good enough, but the analysis would have to deviate significantly from the two lemmas in Section 8.1. It is worth noting that there exist $O(\log n)$-wise independent hash functions that can be evaluated in constant expected time; see [39].

Exercises

Problem 1. Prove Lemma 8.2.

Problem 2. Prove the second bullet of Lemma 8.5 assuming all those $N/(8e^3)$ updates are insertions.

(Hint: pretend all those insertions were given at the checkpoint and include them in the argument for proving the first bullet.)

Problem 3. Prove the second bullet of Lemma 8.5 in general (i.e., allowing deletions).

Problem 4 (a uniform hashing function requires lots of space to represent). Let $\mathbb{D}$ be the set of integers from 1 to D where $D \geq 1$ is an integer.

(a) How many different functions are there mapping $\mathbb{D}$ to $\{1, 2, ..., N\}$ where $N \geq 1$ is an integer?

(b) Prove: at least $D \log_2 N$ bits are required to represent all the above functions, regardless of how the functions are encoding in binary form.

(c)* Prove: if any uniform-hashing function from $\mathbb{D}$ to $\{1, 2, ..., N\}$ requires $D \log_2 N$ bits to represent. (Hint: such a hash function must be a random variable. What are the possible values of this random variable?)

Remark: this means that uniform hashing may not be a fair assumption for practical applications.

The next two exercises would help you gain intuition as to why cuckoo hashing guarantees $O(1)$ expected amortized update time.

Problem 5. Consider a checkpoint rebuild where $N = 4e^3 n$ and $n = |S|$. Recall that the **rebuild** algorithm (Section 8.2.4) inserts the elements of S one by one. Let $e \in S$ be the last element inserted. Prove: when e is inserted, $A[g(e)]$ is occupied with probability at most $\frac{1}{4e^3}$.

(Hint: for any $e' \neq e$, $\mathbf{Pr}[g(e) = g(e')] = 1/N$.)

Remark: this means the insertion of e requires no bumping process with probability at least $1 - \frac{1}{4e^3} > 98\%$.

Problem 6. Same settings as in Problem 5. Suppose that the insertion of e launches the bumping process. Recall that the process evicts a sequences of elements; let the sequence be $e_1, e_2, ..., e_\ell$.

(a) Prove: $\mathbf{Pr}[\ell > 1] \leq \frac{1}{4e^3}$.

(b) Assume that e, e_1, e_2 are distinct. Prove: $\mathbf{Pr}[\ell > 2] \leq \left(\frac{1}{4e^3}\right)^2$.

(c) Assume that $e, e_1, ..., e_t$ are distinct. Prove: $\mathbf{Pr}[\ell > t] \leq \left(\frac{1}{4e^3}\right)^t$.

(Hint: if you can solve (a), you can solve the rest. For (a), think of something similar to Problem 5.)

Lecture 9: Binomial and Fibonacci Heaps

In your undergraduate study, you must have learned that a *heap* (a.k.a. a *priority queue*) supports at least the following two operations on a set S of n elements drawn from an ordered domain:

- **Insertion:** add a new element in S;
- **Delmin:** find and remove the smallest element in S.

It is easy to design a data structure of $O(n)$ space that supports both operations in $O(\log n)$ worst-case time (e.g., the BST).

This lecture will introduce two new heap implementations. The first one, called the *binomial heap*, achieves $O(1)$ amortized insertion cost and $O(\log n)$ amortized delmin cost. Thus, any mixture of n_{ins} insertions and n_{dmin} delmins can be processed in $O(n_{ins} + n_{dmin} \cdot \log n_{ins})$ time, which is much better than the "undergraduate heap" if $n_{ins} \gg n_{dmin}$.

In the second part, we will modify the binomial heap to support an additional "decrease-key" operation in $O(1)$ amortized time (this operation's definition will be deferred to Section 9.2.5). The modification gives rise to the *Fibonacci heap*, which allows us to improve the running time of several fundamental graph algorithms (e.g., Dijkstra's and Prim's), compared to using "undergraduate heaps".

9.1 The binomial heap

9.1.1 The heap property

Let $\mathcal{T}$ be a tree where each node stores an integer *key*. We call $\mathcal{T}$ a *heap* if it has the property below:

> **Heap property:** For any node u in $\mathcal{T}$, its key is the smallest among all the keys stored in the subtree of u.

9.1.2 Binomial trees

Binomial trees are building bricks of the binomial heap (to be defined in the next subsection).

Definition 9.1. *A* **binomial tree of order 0** *is a single node. Inductively, a* **binomial tree of order k** *is a tree where the root has k subtrees the i-th ($1 \le i \le k$) of which is a binomial tree of order $i - 1$.*

See Figure 9.1 for an illustration.

Proposition 9.2. *A binomial tree of order k has 2^k nodes.*

Proof. Trivial by induction. □

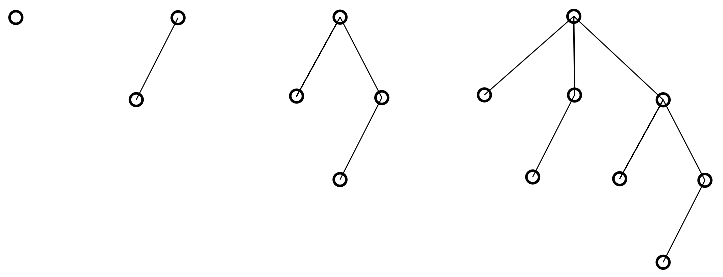


Figure 9.1: Binomial trees of orders 0, 1, 2, and 3

9.1.3 The structure of a binomial heap

Definition 9.3. *A* **binomial heap** *on a set S of n integers is a set Σ of binomial trees such that*

- *every node of every tree in Σ stores a distinct element of S as the* **key***;*
- *every integer in S is the key of a node of some tree in Σ;*
- *every tree in Σ is a heap (Section 9.1.1).*

The binomial heap is **clean** *if no two binomial trees in Σ have the same order; otherwise, it is* **dirty***.*

Proposition 9.4. *A binomial heap on S uses space $O(n)$, and that every binomial tree in Σ has order $O(\log n)$.*

Proof. The space bound follows directly from Definition 9.3. The claim on the order follows from Proposition 9.2. □

9.1.4 Insertion

To insert a new element e_{new}, we simply (i) make an order-0 binomial tree B where the only node stores e_{new} as the key, and (ii) add B into Σ. The cost is $O(1)$.

Note that an insertion may leave the binomial heap in a dirty state.

9.1.5 Delmin

Denote by m the size of Σ at the beginning of the delmin operation. To find the smallest element e_{min} in S, we spend $O(m)$ time to find the tree B in Σ whose root has the smallest key (which must be e_{min}). Next, we

- remove the root of B, which disconnects its subtrees $B_0, B_1, ..., B_s$ ($s \geq 0$), and
- add all of $B_0, ..., B_s$ into Σ (notice that each of $B_0, B_1, ..., B_s$ is a heap).

The cost is $O(m + s) = O(m + \log n)$ (Proposition 9.4).

Finally, we launch a *cleanup process* which converts the binomial heap to a clean state. First, create $O(\log n)$ linked lists L_i, $0 \leq i = O(\log n)$, where L_i contains a pointer to every binomial tree in Σ with order i. Then, we process i in ascending order as follows: as long as L_i has at least two trees B, B', merge them with the algorithm below:

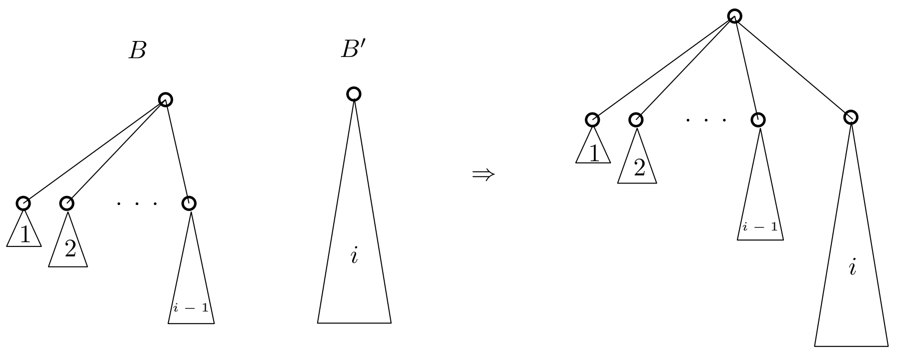


Figure 9.2: Illustration of merge

```
**merge**$(B, B')$
/* $B$ and $B'$ have the same order */
/* without loss of generality, assume that the root of $B$ has a smaller key */
1. $r \leftarrow$ root of $B$
2. make $B'$ the last subtree of $r$
```

See Figure 9.2 for an illustration. Note that B becomes an order-$(i+1)$ tree after the merge, and hence, is moved from L_i to L_{i+1}. B' is removed from L_i.

The number of trees in Σ decreases by one after every merge. Therefore, the cleanup process takes $O(m + \log n)$ time in total (Σ has at most $m + O(\log n)$ trees when the process starts).

9.1.6 Amortization

Next, we use the potential method (Section 7.1) to prove that each insertion is amortized $O(1)$ cost and each delmin is amortized $O(\log n)$ cost.

Define a potential function:

$$\Phi(\Sigma) = c \cdot |\Sigma|$$

where c is a sufficiently large constant to be decided later.

As explained in Section 9.1.4, each insertion takes constant time. The potential function increases by c afterwards. By Lemma 7.1, the insertion is amortized a cost of $O(1) + c = O(1)$.

Now, consider a delmin. Following Section 9.1.5, denote by m the size of Σ at the beginning of the operation. In other words, before the operation, the potential function was $c \cdot m$. At the end of the operation, the binomial heap is clean, meaning that $|\Sigma| = O(\log n)$. Therefore, the potential function has changed by $-c \cdot m + O(\log n)$. Given that each delmin is processed in $O(m + \log n)$ time, it is amortized

$$O(m + \log n) - c \cdot m + O(\log n)$$

cost, which is $O(\log n)$ when c is larger than the hidden constant in the first big-O.

9.2 The Fibonacci heap

The Fibonacci heap is similar to the Binomial heap, except that it adopts a relaxed version of binomial trees in order to support an extra decrease-key operation efficiently.

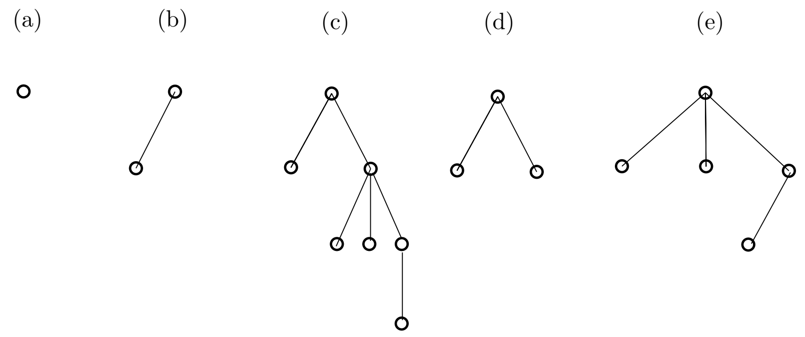


Figure 9.3: Relaxed binomial trees of order 0, 1, 2, 2, and 3, respectively

9.2.1 Relaxed binomial trees

Definition 9.5. *A* **relaxed binomial tree of order 0** *is a single node. Inductively, a* **relaxed binomial tree of order k** *is a tree whose root has k subtrees the i-th ($1 \le i \le k$) of which is a relaxed binomial tree of order at least* $\max\{0, i-2\}$.

See Figure 9.3 for an illustration.

Lemma 9.6. *A relaxed binomial tree of order k has at least $(\frac{1+\sqrt{5}}{2})^{k-2}$ nodes if $k \ge 2$.*

Proof. Define $f(k)$ to be the smallest number of nodes in any relaxed binomial tree of order k.

Proposition 9.7. *$f(k)$ monotonically increases with k.*

The proof is left as an exercise. An order-k relaxed binomial tree must have at least k subtrees such that the i-th subtree ($i \ge 2$) has order at least $i-2$. It follows from Proposition 9.7 that

$$f(k) \quad \ge \quad 1 + f(0) + \sum_{i=2}^{k} f(i-2) = 2 + \sum_{i=0}^{k-2} f(i). \tag{9.1}$$

Define *Fibonacci numbers* as

$$F_k \quad = \quad \begin{cases} 0 & \text{if } k = 0 \\ 1 & \text{if } k = 1 \\ F_{k-1} + F_{k-2} & \text{if } k \ge 2 \end{cases}$$

The following are two well-known properties of Fibonacci numbers (proof left as an exercise):

Proposition 9.8. *$F_{k+2} \ge (\frac{1+\sqrt{5}}{2})^k$, and $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.*

Next, we will prove

Claim: $f(k) \ge F_k$ for $k \ge 0$

which together with the above proposition will complete the proof of Lemma 9.6.

It is easy to verify that the claim is correct for $k = 0$ and $k = 1$. Assuming correctness for any $k \le t-1$ with $t \ge 2$, we prove the case of $k = t$ as follows:

$$\begin{aligned} f(t) &\ge 2 + \sum_{i=0}^{t-2} F_i \\ \text{(by Proposition 9.8)} &= 1 + F_{t-2} + F_{t-1} \\ &> F_t. \end{aligned}$$

□

9.2.2 The structure of a Fibonacci heap

Definition 9.9. *A* **Fibonacci heap** *on a set S of n integers is a set Σ of relaxed binomial trees such that*

- *every node of every tree in Σ stores a distinct element of S as the* **key***;*
- *every integer in S is the key of a node of some tree in Σ;*
- *every tree in Σ is a heap (Section 9.1.1)*

The Fibonacci heap is **clean** *if no two relaxed binomial trees in Σ have the same order; otherwise, it is* **dirty***.*

Proposition 9.10. *A Fibonacci heap on S uses space $O(n)$, and that every relaxed binomial tree in Σ has order $O(\log n)$.*

Proof. The space bound follows directly from Definition 9.9. The claim on the order follows from Proposition 9.6. □

We also introduce a *color* field in each node, which can be *white* or *black*. At all times, we enforce the following invariant for every *non-root* node u:

> **Invariant:** Suppose that u is the i-th child of its parent p. If u is white (or black, resp.), the subtree rooted at u must be a relaxed binomial tree of order at least $i-1$ (or $\max\{0, i-2\}$, resp.).

The color of a root node is unimportant.

9.2.3 Insertion

To insert a new element e_{new}, we simply (i) make a one-node tree R where where the only node stores e_{new} as the key, (ii) color the node white, and (iii) add R into Σ. The cost is $O(1)$.

Note that an insertion may leave the Fibonacci heap in a dirty state.

9.2.4 Delmin

Denote by m the size of Σ at the beginning of the delmin operation.

To find the smallest element e_{min} in S, we spend $O(m)$ time to find the tree $R \in \Sigma$ whose root has the smallest key (which must be e_{min}). Next, we remove the root r of R, color the child nodes of r white, and add all the subtrees of r to Σ. The cost is $O(m + \log n)$ (Proposition 9.6).

Launch a *cleanup process* to convert the Fibonacci heap to a clean state. First, create $O(\log n)$ linked lists L_i ($0 \le i = O(\log n)$) where L_i contains a pointer to every Fibonacci tree in Σ with order i. Then, we process i in ascending order as follows: as long as L_i has at least two trees R, R', merge them with the algorithm below:

```
merge(R, R')
/* R and R' have the same order */
/* without loss of generality, assume that the root of R has a smaller key */
1. color the roots of R and R' white
2. r ← root of R
3. make R' the last subtree of r
```

Proposition 9.11. *After the merge, R is an order-$(i+1)$ relaxed binomial tree.*

Proof. R' has order i, color white, and is the $(i+1)$-th child of r (where r is the root of R), which fulfills the requirement in Definition 9.5. □

After the merge, R is moved from L_i to L_{i+1}, while R' is removed from L_i.

The whole cleanup process takes $O(m + \log n)$ time in total.

9.2.5 Decrease-key

The real purpose of discussing the Fibonacci heap is to support the following operation:

> **Decrease-key(u, x_{new}):** Here, u is a node in some relaxed binomial tree in Σ such that the key of u is larger than x_{new}. The operation modifies the key of u to x_{new}.

Note that the node u is supplied to decrease-key as an input directly.

Let R be the tree containing u. If u is the root of R, we simply change the key of u to x_{new} and finish (think: why is this correct?).

Consider now the case where u is not the root of R, and thus, has a parent p. If modifying the key of u to x_{new} does not violate the heap property in Section 9.1.1 (namely, x_{new} is greater than the key of p), we carry out the modification and finish.

On the other hand, if the heap property is violated, we remove the subtree R_u of u from R, color u white, and add R_u to Σ.

Let us now focus on p, for which we have:

Proposition 9.12. *Every child v of p still satisfies the invariant in Section 9.2.2.*

Proof. Suppose that v was the i-th child of p before the removal of u, and the j-th afterwards. The claim follows from the fact that $j \le i$. □

Suppose that the color of p is white. In this case, we simply color p black and finish.

Proposition 9.13. *If p was white before the decrease-key, now it still satisfies the invariant in Section 9.2.2.*

Proof. If p is a root, the claim is trivially true. Otherwise, suppose that p is the i-th child of its parent. The color of p indicates p had $i-1$ child nodes before the decrease-key. Hence, now p must have at least $\max\{0, i-2\}$ child nodes, implying that the subtree of p is a relaxed binomial tree of order $\max\{0, i-2\}$. □

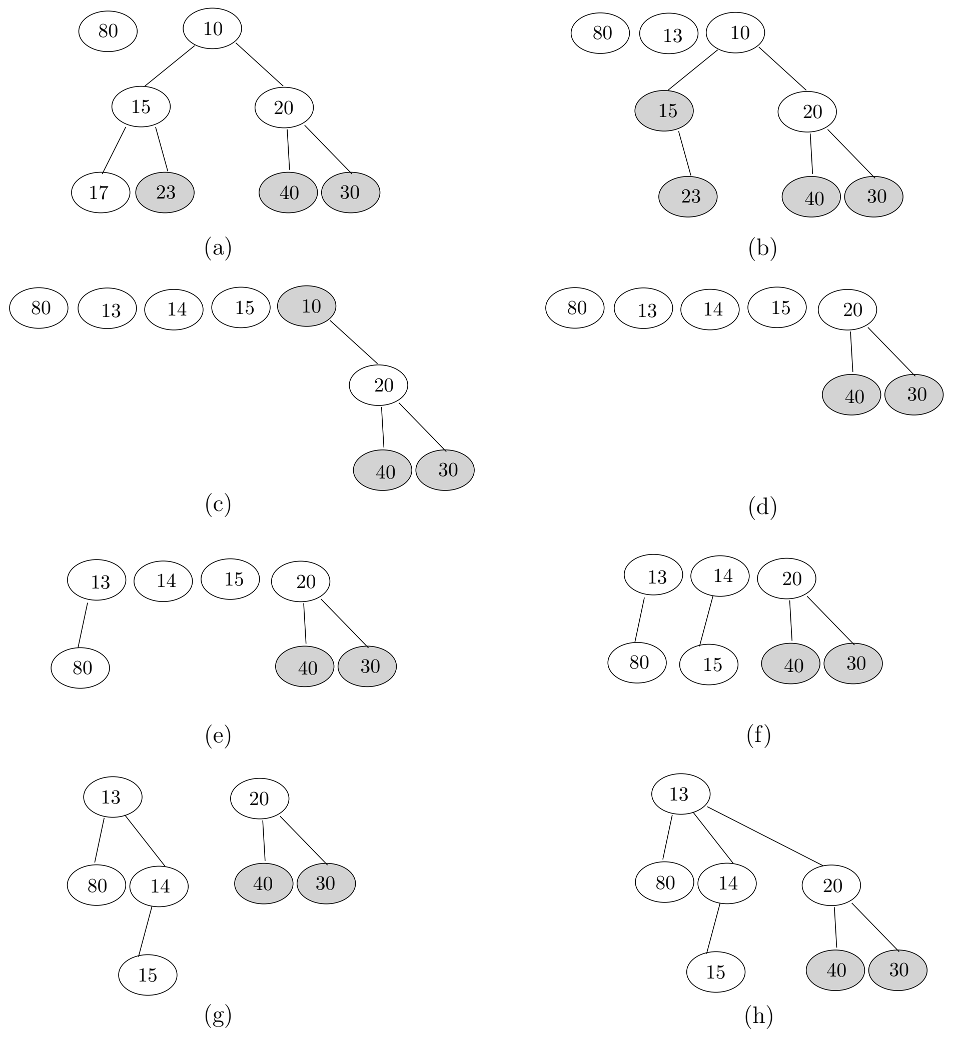


Figure 9.4: (a) shows a Fibonacci heap. (b) is the heap after decreasing the key 17 to 13, and (c) after decreasing the key 23 to 14. (d) shows the heap during a delmin operation; here the smallest element 10 has just been removed. Continuing the delmin, (e) merges 13 with 80, (f) merges 14 with 15, (g) merges 13 with 14, and (h) merges 13 with 20.

What if p was *already* black before the removal of R_u? In that case, we remove R_p from R, add R_p to Σ, and color p white; p is said to have been *repaired*.

But we are not done. If p has a parent p', the removal of R_p causes p' to lose a child. Thus, the aforementioned issue may happen to p' as well, which can be remedied in the same manner. The process may propagate all the way to the root.

In general, the decrease-key operation takes $O(1 + g)$ time, where g is the number of nodes repaired.

9.2.6 Example

Figure 9.4(a) shows a Fibonacci heap where each node is labeled with its key. At the moment, Σ has two relaxed binomial trees with orders 0 and 2, respectively.

Suppose that we perform a decrease-key to reduce the key 17 to 13. The operation removes the subtree of 13 (originally 17), and adds it to Σ (which now has 3 trees). Node 15 turns black. Figure 9.4(b) gives the current heap.

Lowering the key 23 to 14, the next decrease-key removes the subtree of 14 (originally 23), and adds it to Σ. The color of node 14 changes from black to white. As node 15 is black, we repair it by detaching its subtree from its parent (node 10), adding its subtree to Σ, and coloring it white. Node 10 then turns black[1]. The resulting heap is presented in Figure 9.4(c).

Let us then perform a delmin operation. We go through the roots of all the 5 trees in Σ to identify the smallest element 10. After 10 is deleted, its subtree is added to Σ, giving rise to Figure 9.4(d). A clean-up process then is launched to merge trees of the same order. Figure 9.4(e) merges 13 and 80 into a tree of order 1, and similarly, Figure 9.4(f) merges 14 with 15. The trees of 13 and 14 are then merged, yield a tree of order 2 as shown in Figure 9.4(g). Finally, the trees of 13 and 20 are merged, producing the final Fibonacci heap in Figure 9.4(h).

9.2.7 Amortization

We use the potential method to prove that an insertion and a decrease-key are amortized $O(1)$ cost, while a delmin is amortized $O(\log n)$ cost.

Define a potential function:

$$\Phi(\Sigma) \quad = \quad c_1 \cdot |\Sigma| + c_2 \cdot (\text{number of black nodes})$$

where c_1 and c_2 are constants to be decided later.

Each insertion takes constant time. The potential function increases by c_1 afterwards. By Lemma 7.1, the insertion is amortized a cost of $O(1) + c_1 = O(1)$.

Now, consider a delmin. Denote by m the size of Σ before the operation. Afterwards, the Fibonacci heap is clean, meaning that $|\Sigma| = O(\log n)$. The clean-up process can only decrease the number of black nodes. Therefore, after the delmin the potential function has decreased at least $c_1 \cdot m - O(\log n)$. Given that the delmin is processed in $O(m + \log n)$ time, it is amortized at most

$$O(m + \log n) - (c_1 \cdot m - O(\log n))$$

cost, which is $O(\log n)$ when c_1 is larger than the hidden constant in the first big-O.

Finally, consider a **decrease-key**(u, x_{new}). Denote by g the number of nodes repaired by the operation.

- Every such node must be black before the operation, but no longer so afterwards. On the other hand, decrease-key may turn one node from white to black (think: why?). Hence, the number of black nodes drops by $g - 1$.
- Since (i) the subtree of u is inserted into Σ, and (ii) so is the subtree of every repaired node, $|\Sigma|$ increases by $g + 1$.

Therefore, the potential function has changed by $c_1(g + 1) - c_2 \cdot (g - 1)$ after the operation. Given that the decrease-key is processed in $O(1 + g)$ time, it is amortized

$$O(1 + g) + c_1(g + 1) - c_2 \cdot (g - 1)$$

cost, which is $O(1)$ as long as c_2 is at least the sum of c_1 and the hidden constant of the first big-O.

[1]Since the color of a root node is unimportant, you may as well keep the color of node 10 white.

9.3 Remarks

The binomial heap was proposed by Vuillemin [42], while the Fibonacci heap by Fredman and Tarjan [20].

Exercises

Problem 1. Prove Proposition 9.7.

Problem 2. Prove Proposition 9.8.

Problem 3. Complete the decrease-key algorithm (Section 9.2.5) by explaining what to do when u is the root of R.

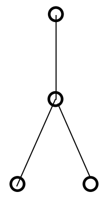
Problem 4. Suppose that we want to support an extra operator called **find-min** which reports the smallest key in S, but does not remove it. Explain how to adapt the binomial heap to support this operation in $O(1)$ worst-case time, without affecting the performance guarantees of insertion and delmin.

Problem 5*. Explain how to modify the binomial heap's algorithm to support delmin in $O(\log n)$ worst-case time, and (as before) an insertion in $O(1)$ amortized time.

(Hint: keep the binomial heap clean *at all times*).

Problem 6. Prove or disprove: a relaxed binomial tree of n nodes has height $O(\log n)$.

Problem 7. Give a sequence of insert, delmin, and decrease-key operations on an initially empty set such that the Fibonacci heap after all the operations has a single tree that looks like:



Problem 8 (meld). Let S_1 and S_2 be two disjoint sets. Given a Fibonacci heap Σ_1 on S_1 and a Fibonacci heap Σ_2 on S_2, explain how to obtain a Fibonacci heap on $S_1 \cup S_2$ in constant worst-case time.

Problem 9. Implement Dijkstra's algorithm on a graph of n nodes and m edges in $O(m + n \log n)$ time.

Lecture 10: Union-Find Structures

This lecture will discuss the *disjoint set problem*. Let V be a set of n integers. Define $\mathcal{S}$ to be a set of non-empty subsets of V such that:

- the sets in $\mathcal{S}$ are mutually disjoint;
- each element in V belongs to (exactly) one set in $\mathcal{S}$.

We want to store $\mathcal{S}$ in a data structure that supports the operations below:

- **makeset**(e): given an integer $e \notin V$, adds e to V, and a singleton set $\{e\}$ to $\mathcal{S}$;
- **find**(e): reports which set $S \in \mathcal{S}$ contains $e \in V$;
- **union**(e, e'): unions two different sets $S, S' \in \mathcal{S}$ that contain $e, e' \in V$, respectively.

The output of **find**(e) can be anything uniquely identifying the set of e. However, it must be ensured that the same identifier is always used for the same set (i.e., if e, e' belong to the same set, then the outputs of **find**(e) and **find**(e') must be identical). Also, for simplicity, we assume that V is empty at the beginning.

We will learn a surprisingly simple structure that optimally solves the problem. The analysis demonstrates in a unique manner how a structure's performance can be bounded by "non-conventional" functions (more specifically, "iterated logs" and "inverse of Ackermann function").

Data structures solving the disjoint set problem are often referred to as *union-find structures*. They have many applications in computer science. In the exercises, we will explore some applications in graph algorithms.

10.1 Structure and algorithms

Structure. We store each set $S \in \mathcal{S}$ in a tree T where

- T has as many nodes as $|S|$;
- every element $e \in S$ is stored at a distinct node u in T;
- each node u also stores a special integer referred to as its *rank* and denoted as *rank*(u).

Given a node u of some tree T, we will use *parent*(u) to denote the parent of u. If u is the root, then *parent*(u) is nil.

The structure has as many trees as the number of sets in $\mathcal{S}$. The space consumption is obviously $O(n)$.

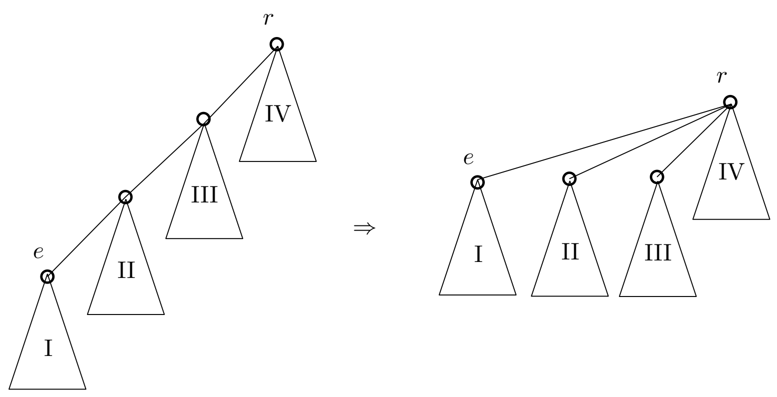


Figure 10.1: Illustration of **find**

Makeset(e)**.** Create a tree with a single node storing e, whose rank is 0. The time needed is $O(1)$.

Union(e, e')**.** Denote by T (or T', resp.) the tree that contains e (or e', resp.).

Assumption 1: The roots r and r' of the two trees are given.

The union operation is performed as follows:

```
union(r, r')
/* assume, without loss of generality, that $rank(r) \geq rank(r')$ */
1. make $r'$ a child of $r$
2. if $rank(r) = rank(r')$ then
3.     increase $rank(r)$ by 1
```

The cost is clearly constant.

Find(e)**.** For this operation, we need:

Assumption 2: The node where e is stored is given.

The operation proceeds as follows:

```
find(e)
/* let $T$ be the tree where $e$ is stored */
1. $\Pi \leftarrow$ the path from node $e$ to the root $r$ of $T$
2. for each node $u$ on $\Pi$ do
3.     set $parent(u) \leftarrow r$
4. return $r$
```

See Figure 10.1 for an illustration. Note that r is used as the identifier of the set stored in T. The running time is $O(|\Pi|)$ where $|\Pi|$ gives the number of nodes on Π. This may appear large, but as the rest of the lecture will discuss, the amortized cost of **find** is very low.

10.2 Analysis 1

We will prove that, under Assumptions 1 and 2, each operation is amortized $O(\log^* n)$ time, where $\log^* n$ is how many times we need to iteratively perform the $\log_2(.)$ function on n before getting a number less than 2. For example, $\log^* 16 = 3$ because $\log_2 \log_2 16 = 2$ while $\log_2 \log_2 \log_2 16 = 1$. It is worth mentioning that $\log^* n \le 5$ for all $n \le 2^{65536}$, which is already larger than the total number of atoms on earth.

The $O(\log^* n)$ bound will be subsumed by another result to be established in Section 10.3. However, the argument in this section is (much) simpler, and illustrates some properties that will also be useful in Section 10.3.

10.2.1 Basic properties

The analysis will revolve around node ranks (Section 10.1), about which this subsection will prove several basic facts.

Proposition 10.1. *Once a node u becomes a non-root node, rank(u) is fixed forever.*

Proof. The rank of a node u is modified only in **union**, and only when u is a root. Once a u becomes a non-root, it will never be a root again. □

Proposition 10.2. *Consider any nodes u, v such that $u = parent(v)$. Then, $rank(u) > rank(v)$.*

Proof. Assuming that this is true currently, it is easy to show that it remains so after a **union**/**find** operation. □

Proposition 10.3. *For any node u, every time* **find** *changes $parent(u)$, the new parent must have a rank larger than the old parent.*

Proof. Let $v = parent(u)$. If **find** modifies $parent(u)$, v cannot be the root r of the tree where u belongs. By Proposition 10.2, $rank(v) < rank(r)$. The claim follows from the fact that $parent(u) = r$ after the **find** operation. □

Proposition 10.4. *A root u with rank ρ has at least 2^ρ nodes in its subtree.*

Proof. This is obviously true for $\rho = 0$. Inductively, the correctness on $\rho = i \ge 1$ follows from the fact that, when $rank(u)$ increases from $i - 1$ to i in **union**, u takes a new child v with rank $i - 1$ whose subtree has size at least 2^{i-1}. □

Corollary 10.5. *The rank of a node is $O(\log n)$.*

Proof. Immediately from Proposition 10.4. □

Lemma 10.6. *At most $n/2^\rho$ nodes have rank ρ.*

Proof. Fix a particular a value of ρ. When the rank of a node u increases to ρ in a **union** operation, we *assign* all the nodes v in the subtree of u to u. We argue that every v is assigned at most once this way.

Suppose, on the contrary, that v is later assigned to another node u'. When this happens, u' must be a root of the tree T containing. Thus, u is also in T (if two nodes are in the same tree, they will remain so forever). Hence, u' is a proper ancestor of u. However, Propositions 10.1 and 10.2 suggests that $rank(u') > rank(u) \ge \rho$, giving a contradiction. □

Corollary 10.7. *At most $n/2^{\rho-1}$ nodes have rank at least ρ.*

Proof. By Lemma 10.6, the number of such nodes is at most

$$\sum_{i=\rho}^{\infty} \frac{n}{2^\rho} \quad < \quad \frac{n}{2^{\rho-1}}.$$

□

10.2.2 An $O(\log \log n)$ bound

In this subsection, we will prove an amortized bound of $O(\log \log n)$ per operation. The main purpose, however, is to introduce a charging argument which will be strengthened later.

We divide all the nodes with positive ranks into *groups*. Specifically, group $g \geq 0$ includes all the nodes u satisfying

$$rank(u) \in [2^g, 2^{g+1}). \tag{10.1}$$

Because of Corollary 10.5, the number of groups is $O(\log \log n)$.

Now consider a **find**(e) operation. Recall that it finishes in $O(|\Pi|)$ time, where Π is the path from the root r to the node e. We account for the cost by looking at each node $u \in \Pi$:

- Case 1: If u has rank 0, charge $O(1)$ cost on **find**.
- Case 2: If $u = r$ or $parent(u) = r$, charge $O(1)$ cost on **find**.
- Case 3: If u and $parent(u)$ are in different groups, charge $O(1)$ cost on **find**.
- Case 4: Otherwise, charge $O(1)$ cost on u.

Thus, all the $O(|\Pi|)$ time has been amortized on either **find** or individual nodes.

Proposition 10.8. *Cases 1-3 charge $O(\log \log n)$ time on each* **find.**

Proof. Cases 1 and 2 obviously charge only $O(1)$ time on **find.**

Consider Case 3. By Proposition 10.2, as we ascend Π, the node rank monotonically increases. Thus, if Case 3 applies x times, we can find x nodes on Π with increasingly larger group numbers. The claim follows from the fact that there are $O(\log \log n)$ groups. □

Lemma 10.9. *Case 4 can charge $O(n \log \log n)$ cost in total for all the* **find** *operations.*

Proof. Clearly, Case 4 charges only on a non-root node u. By Proposition 10.1, $rank(u)$ will no longer change; hence, neither will its group number g.

Every time this happens, $parent(u)$ changes because $parent(u) \neq r$ before **find** (otherwise, Case 2 should have applied), while $parent(u) = r$ afterwards. By Proposition 10.3, the new parent of u has a larger rank than the old one. By (10.1), there are 2^g distinct ranks in group g. Therefore, u can be charged at most 2^g times after which $parent(u)$ will forever be in a group numbered at least $g + 1$.

Corollary 10.7 implies that the number of nodes in group g is at most $n/2^{2^g}$. Therefore, the total cost charged on group-g nodes is

$$O\left(\frac{n}{2^{2^g}} \cdot 2^g\right) \quad = \quad O(n). \tag{10.2}$$

The lemma follows from the fact that there are $O(\log \log n)$ groups. □

We amortize the $O(n \log\log n)$ bound in Lemma 10.9 over the n **makeset** operations that created the n nodes in V. Each operation therefore bears $O(\log\log n)$ cost.

10.2.3 An $O(\log^* n)$ bound

Did you notice (10.2) is suspiciously loose? The culprit lies in the group definition in (10.1). Just for fun, let us change the definition of group g to

$$rank(u) \in [2^{2^g}, 2^{2^{g+1}}). \tag{10.3}$$

The number of groups drops to $O(\log\log\log n)$. Repeating the above analysis gives an amortized bound of $O(\log\log\log n)$, as is left as an exercise.

To push the power of the argument to the extreme, let us adopt the following definition:

$$rank(u) \in \Big[\underbrace{2^{2^{\cdots^2}}}_{g}, \underbrace{2^{2^{\cdots^2}}}_{g+1}\Big). \tag{10.4}$$

The number of groups is now $O(\log^* n)$. The same argument then yields an amortized bound of $O(\log^* n)$ (again, left as an exercise).

10.3 Analysis 2

In this section, we will prove that, under Assumptions 1 and 2, our algorithms in Section 10.1 actually achieve amortized $O(\alpha(n))$ time per operation, where $\alpha(n)$ is the inverse of the Ackermann's function, which is an extremely slow-growing function, e.g., $\alpha(n) \le 5$ for $n = \underbrace{2^{2^{\cdots^2}}}_{2^{2048}}$. We start with an introduction to this bizarre-looking function.

10.3.1 Ackermann's function and its inverse

There are many variants of Ackermann's function; and what we discuss below is one of them. Denote by $\mathbb{N}_{\ge 0}$ the set of positive integers. Given any function $f : \mathbb{N}_{\ge 0} \to \mathbb{N}$ (where $\mathbb{N}$ is the set of integers), we define for $k \ge 1$

$$f^{(k)}(n) = \underbrace{f(f(...f}_{k}(n)...)).$$

For example, $\log_2^{(2)} n = \log_2 \log_2 n$, and should not be confused with $\log_2^2 n = (\log_2 n)^2$.

Now, we introduce a family of functions from $\mathbb{N}_{\ge 0}$ to $\mathbb{N}$:

$$\begin{aligned} A_0(x) &= x + 1 \\ A_k(x) &= A_{k-1}^{(x+1)}(x) \text{ for } k \ge 1. \end{aligned} \tag{10.5}$$

To see how quickly these functions grow, consider some small values of k:

$$A_1(x) = A_0^{(x+1)}(x) = \underbrace{A_0(A_0(...A_0}_{x+1}(x)...)) = 2x + 1 \quad > \quad 2x$$

$$A_2(x) = A_1^{(x+1)}(x) = \underbrace{A_1(A_1(...A_1}_{x+1}(x)...)) > x2^x \quad \ge \quad 2^x$$

$$A_3(x) = A_2^{(x+1)}(x) = \underbrace{A_2(A_2(...A_2}_{x+1}(x)...)) \quad > \quad \underbrace{2^{2^{\cdots^2}}}_{x}.$$

If we define $2 \uparrow x = \underbrace{2^{2^{\cdots^2}}}_{x}$, then

$$A_4(x) = A_3^{(x+1)}(x) \quad > \quad \underbrace{2 \uparrow (2 \uparrow (...(2 \uparrow 2)...))}_{x\ \uparrow\text{'s}}$$

Calling $A_k(2)$ *Ackermann's function* (which is a function of k), we define the *inverse* of Ackermann's function as

$$\alpha(n) \quad = \quad \text{the smallest } k \text{ satisfying } A_k(1) \geq n. \tag{10.6}$$

We can prove $\alpha(n) \leq 5$ for $n \leq \underbrace{2^{2^{\cdots^2}}}_{2^{2048}}$.

10.3.2 An $O(\alpha(n))$ bound

For every non-root node u with $rank(u) \geq 1$, define $k(u)$ as the largest integer $k \geq 0$ satisfying

$$rank(parent(u)) \quad \geq \quad A_k(rank(u)) \tag{10.7}$$

where $A_k(.)$ is given in (10.5). Note that $k(u)$ is well-defined because, by Proposition 10.2, $rank(parent(u)) \geq rank(u) + 1 = A_0(rank(u))$.

To understand (10.7) more intuitively, first note that $rank(u)$ has forever been fixed (Proposition 10.1), while $rank(parent(u))$ monotonically increases over time (Proposition 10.3). As k grows, the value of $A_k(rank(u))$ increases very rapidly (Section 10.3.1), and eventually exceeds $rank(parent(u))$. The value of $k(u)$ captures the "last" k such that $A_k(rank(u))$ has not exceeded $rank(parent(u))$; clearly, $k(u)$ may increase over time along with $rank(parent(u))$.

We divide the non-root nodes into *groups*, but in a way different from Section 10.2.2. Specifically, group $g \geq 0$ includes all the non-root nodes u with $k(u) = g$.

Proposition 10.10. *$0 \leq k(u) \leq \alpha(n)$, namely, there are at most $1 + \alpha(n)$ groups.*

Proof. Lemma 10.6 implies that every node has rank at most $O(\log n)$. The claim follows from the definition in (10.6).[1] □

Consider a **find**(e) operation, which finishes in $O(|\Pi|)$ time where Π is the path from the root r to the node e. We account for the cost by looking at each node $u \in \Pi$:

- Case 1: If $rank(u) = 0$ or u is a root, charge $O(1)$ cost on **find**.
- Case 2: If u has a proper non-root ancestor v such that $k(v) = k(u)$, charge $O(1)$ cost on u.
- Case 3: Otherwise, charge the cost on **find**.

Thus, all the $O(|\Pi|)$ time has been amortized on either **find** or individual nodes.

Proposition 10.11. *Case 1 charges twice on each* **find.**

Proof. Only one rank-0 node and one root on Π. □

[1]You would probably ask why not $O(\alpha(\log n))$. In fact, it *is* $O(\alpha(\log n))$, except that this is not very helpful because we can prove $\alpha(n) = O(\alpha(\log n))$.

Proposition 10.12. *Case 3 charges $O(\alpha(n))$ time on each* **find.**

Proof. If Case 3 is applied x times on a **find**, we can find x nodes u on Π with *distinct* $k(u)$. The claim follows then from Proposition 10.10. □

The rest of the section serves as a proof for:

Lemma 10.13. *Case 2 can charge $O(n \cdot \alpha(n))$ cost in total for all the* **find** *operations.*

We amortize the above cost over the n **makeset** operations that created the n nodes in V. Each operation therefore bears $O(\alpha(n))$ cost. This gives the main theorem of this lecture:

Theorem 10.14. *Under Assumptions 1 and 2 (Section 10.1), any sequence of n operations (mixture of* **makeset**, **union**, *and* **find***) can be processed in $O(n \cdot \alpha(n))$ time.*

Proof of Lemma 10.13. We will prove later:

> **Claim 1:** A non-root node with rank ρ can be charged $O(\rho \cdot \alpha(n))$ cost in Case 2, summing over all the **find** operations.

Since there are $O(n/2^\rho)$ nodes with rank ρ (Lemma 10.6), it will then follow that the total time charged by Case 2 is bounded by

$$O\left(\sum_{\rho=1}^{\infty} \frac{n}{2^\rho} \cdot \rho \cdot \alpha(n)\right) \quad = \quad O(n \cdot \alpha(n)).$$

which will complete the proof of Lemma 10.13. Claim 1, on the other hand, is implied by:

> **Claim 2:** For each $g \in [0, \alpha(n)]$, when node u stays in group g, Case 2 can charge u at most $rank(u)$ times.

The rest of the discussion will focus on proving Claim 2.

When u belongs to group g, we have $g = k(u)$. Thus, by definition of $k(u)$ in (10.7):

$$rank(parent(u)) \quad \geq \quad A_g(rank(u)) = A_g^{(1)}(rank(u))$$

while

$$rank(parent(u)) \quad < \quad A_{g+1}(rank(u)) = A_g^{(rank(u)+1)}(rank(u)).$$

Consider one arbitrary **find** operation that charges u in Case 2. Let i be the largest integer in $[1, rank(u) + 1)$ satisfying

$$rank(parent(u)) \quad \geq \quad A_g^{(i)}(rank(u)) \tag{10.8}$$

before the operation.

Lemma 10.15. *After the* **find** *operation, it must hold that $rank(parent(u)) \geq A_g^{(i+1)}(rank(u))$.*

Proof. Let v be the proper non-root ancestor of u in Case 2. Thus, $k(v) = k(u) = g$.

Consider the root r of the tree where u belongs. Since v is a non-root node, it is a proper descendant of r. We have:

$$\begin{aligned} rank(r) &\geq rank(parent(v)) \quad \text{(by Proposition 10.2)} \\ &\geq A_g(rank(v)) \quad \text{(by def. of } k(v)\text{)} \\ &\geq A_g(rank(parent(x))) \quad \text{(by monotonicity of } A_g(.)\text{)} \\ &\geq A_g(A_g^{(i)}(rank(u))) \quad \text{(by (10.8))} \\ &= A_g^{(i+1)}(rank(u)). \end{aligned}$$

The lemma then follows from the fact that $parent(u) = r$ after the **find** operation. □

The lemma implies Claim 2, because after $rank(u)$ applications of the Lemma 10.15, it must hold that

$$rank(parent(u)) \quad \geq \quad A_g^{(rank(u)+1)}(rank(u)) = A_{g+1}(rank(u)). \tag{10.9}$$

This indicates that u will then move up to a group numbered at least $g + 1$.

10.4 Remarks

The union-find structure we described is due to Tarjan [40]. The amortized bound in Theorem 10.14 was proved to be tight by Fredman and Saks [19]. In other words, Tarjan's structure is already asymptotically optimal. Analysis 1 was adapted from the lecture notes at `http://people.seas.harvard.edu/~cs125/fall16/lec3.pdf` and those at `https://people.eecs.berkeley.edu/~daw/teaching/cs170-s03/Notes/lecture12.pdf`. Analysis 2 was adapted from the book [27] of Kozen.

Exercises

Problem 1. Prove an $O(\log\log\log n)$ amortized bound when the group is defined using (10.3).

Problem 2. Prove an $O(\log^* n)$ amortized bound when the group is defined using (10.4).

Problem 3. Show that Assumption 1 can be removed without affecting the amortized bound.

Problem 4*. Prove: each **find** operation finishes in $O(\log n)$ worst-case time.

(Hint: for each node u, prove its subtree has height at most $rank(u)$.)

Problem 5*. Describe a union-find structure that processes any sequence of n_1 **makeset** operations, n_2 **find** operations, and m **union** operations in $O(n_1 + n_2 + m \log n_1)$ time. Note that this is better than Theorem 10.14 for $m \leq n_1 / \log n_1$.

(Hint: store each set of $\mathcal{S}$ in a linked list.)

Problem 6 (dynamic connectivity). Consider an undirected graph $G = (V, E)$. Set $n = |V|$. Initially, E is empty (i.e., no edges). Design a structure to support the following operations:

- insert(u, v): add an edge between vertices $u, v \in V$ to E;
- query(u, v): given two vertices $u, v \in V$, report *whether* they belong to the same connected component in G.

Your structure must consume $O(n)$ space at all times (regardless of $|E|$), and support each operation in $O(\alpha(n))$ amortized time.

(Hint: be careful; there are no **makeset** operations here.)

Problem 7 (minimum spanning tree). Consider a weighted undirected graph $G = (V, E)$, where each edge in E is associated with a positive weight. Suppose that the edges in E have been sorted by weight. Describe an algorithm to obtaining a minimum spanning tree of G in $O(m \cdot \alpha(n))$ time, where $n = |V|, m = |E|$.

(Hint: implement Kruskal's algorithm with a union-find structure.)

Lecture 11: Dynamic Connectivity on Trees

Define $V = \{1, 2, ..., n\}$ where each element is called a *vertex*. F is a *forest* (i.e., a set of trees) such that

- each tree in F uses only vertices from V;
- every vertex in V belongs to exactly one tree in F.

We want to store F in a data structure that supports the following operations:

- **insert**(u, v) where vertices u and v belong to different trees in F: add an edge $\{u, v\}$, which effectively merges two trees (and hence, $|F|$ decreases by 1).
- **delete**(u, v) where u and v belong to the same tree $T \in F$: remove an edge $\{u, v\}$ from T, which effectively breaks T into two trees (and hence, $|F|$ increases by 1);
- **connected**(u, v): return whether $u, v \in V$ are in the same tree.

We will refer to the above as the *dynamic connectivity problem on trees.* This lecture will introduce the *Euler-tour structure* which consumes $O(n)$ space, and performs all operations in $O(\log n)$ time. Note that if no deletions are allowed, the problem can be settled with the union-find structure of Lecture 10.

In the second part of the lecture, we will extend the functionality of the Euler-tour structure beyond the above operations. Our final version of the structure will make a powerful tool for the next lecture where we study the dynamic connectivity problem on a graph.

Notations. Given a tree, $|T|$ represents the number of vertices in T.

11.1 Euler tour

Focusing on *one* tree T, this section will introduce a generic method for "linearizing" the vertices of T.

11.1.1 Rooting a tree

Recall that a *tree* T, in general, is defined as an undirected, connected, graph without cycles. It does not automatically have a "root", without which concepts such as "parents", "children", and "subtrees" are undefined.

Suppose that an arbitrary vertex r has been designated as the *root* of T. A vertex u *parents* another vertex v if (i) $\{u, v\}$ is a tree edge, and (ii) u is closer to r than v. Accordingly, v is a *child* of u. Removing the edge $\{u, v\}$ breaks T into two connected components (CCs):

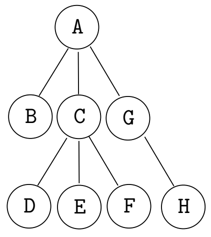


Figure 11.1: An Euler tour: ABACDCECFCAGHGA

- $T^u_{u,v}$: the CC containing u;
- $T^v_{u,v}$: the CC containing v.

We refer to $T^v_{u,v}$ as the *subtree* of v. Specially, the *subtree* of r is the entire T.

Sometimes we will emphasize on the existence of a root by calling T a *rooted tree.*

11.1.2 Euler tour on a rooted tree

Given a rooted tree T, we define an *Euler tour* as a sequence Σ of vertices output by:

```
euler-tour($T$)
1. $r \leftarrow$ root of $T$
2. append $r$ to the output sequence
3. for each child $u$ of $r$ do
4.     euler-tour(the subtree of $u$)
5.     append $r$ to the output
```

Example. Figure 11.1 shows a tree rooted at A. The figure's caption is an Euler tour, but so is ACECFCDCABAGHGA (there are many more). Note that both Euler tours have the same length. □

11.1.3 A cyclic view

Let $m = |T| - 1$ be the number of edges in T. Conceptually, replace each (undirected) edge $\{u, v\}$ in T with two directed edges (u, v) and (v, u). This creates $2m$ directed edges.

Did you notice that Σ always had length $|\Sigma| = 2m + 1$ in the earlier example? This is not a coincidence. Denote the vertex sequence in Σ as: $u_1, u_2, ..., u_{|\Sigma|}$. For each $i \in [1, |\Sigma| - 1]$, interpret the consecutive vertices u_i, u_{i+1} as enumerating a directed edge (u_i, u_{i+1}). By how **euler-tour** runs, each of the $2m$ directed edges is enumerated exactly once, implying that $|\Sigma| = 2m + 1$. Let Q be the sequence of directed edges $(u_1, u_2), (u_2, u_3), ..., (u_{2m}, u_{2m+1})$, which is a cycle because $u_1 = u_{2m+1}$.

Example. In Figure 11.1, the cycle Q is (A, B), (B, A), (A, C), (C, D), (D, C), (C, E), (E, C), (C, F), (F, C), (C, A), (A, G), (G, H), (H, G), (G, A). □

The reverse is also true:

Proposition 11.1. *Let Q be any permutation of the $2m$ directed edges $(u_1, v_1), (u_2, v_2), ..., (u_{2m}, v_{2m})$ satisfying*

- *$v_i = u_{i+1}$ for $i \in [1, 2m - 1]$;*

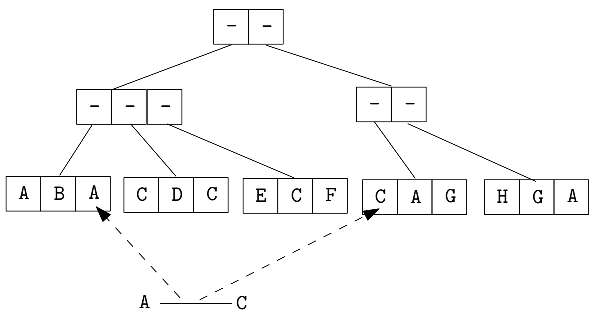


Figure 11.2: An Euler-tour structure for the tree in Figure 11.1 (for clarity, only the pointers of edge $\{\mathtt{A}, \mathtt{C}\}$ is shown)

- $v_{2m} = u_1$

defines an Euler tour $u_1 u_2 u_3 ... u_{2m} u_1$ *of* T *when* T *is rooted at* u_1.

The proof is left to you as an exercise.

11.2 The Euler-tour structure

Let T be a rooted tree with an Euler tour Σ. We store Σ in a 2-3 tree Υ. It follows from Section 11.1.3 that Υ has space $O(|T|)$.

For each edge $\{u, v\}$ in T, we store two pointers:

- one referencing the the occurrence of u that corresponds to the directed edge (u, v);
- the other referencing the the occurrence of v that corresponds to the directed edge (v, u);

The resulting structure is called an *Euler-tour structure* (ETS) of T. See Figure 11.2 for an illustration.

The following subsections will discuss several operations nicely supported by Υ.

11.2.1 Cut

The *cut*(u, v) operation removes an edge $\{u, v\}$ from a rooted T — assume, without loss of generality, that u parents v — which breaks T into two trees:

- T_1: the subtree rooted at v;
- T_2: the tree obtained by removing T_1 from T.

The operation produces an ETS for T_1 and T_2, respectively.

Let Σ be the Euler tour of T (that Υ is based on). Identify the subsequence Σ_1 of Σ that starts from the first occurrence of v, and ends at the last occurrence of v. These two occurrences of v can be identified using the pointers associated with the edge $\{u, v\}$. Denote by Σ_2 the sequence obtained by removing Σ_v from Σ. Observe that:

- Σ_1 is an Euler tour of T_1.
- Σ_2 has two consecutive occurrences of u; removing one of them gives an Euler tour of T_2.

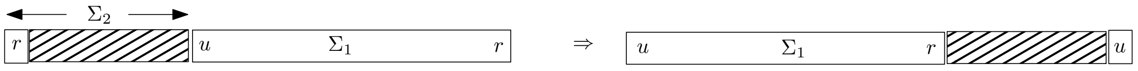


Figure 11.3: Changing the Euler tour in a re-root

Example. Consider deleting the edge {A, C} from Figure 11.1(a). Σ_1 = `CDCECFC` is the Euler tour of T_1 (the subtree of C). Σ_2 = `ABAAGHGA`. There are two consecutive occurrences of A in Σ_2. After removing one, Σ_2 = `ABAGHGA` becomes an Euler tour of T_2 (what remains in Figure 11.1 after trimming T_1). □

The ETS's of T_1 and T_2 can be obtained using the split and join operations of 2-3 trees (Section 2.2.3):

Lemma 11.2. *A cut operation can be performed in $O(\log |T|)$ time.*

The proof is easy and left as an exercise:

11.2.2 Link

Let T_1 and T_2 be two trees such that u is the root of T_1, while v is arbitrary node in T_2. The *link*(u, v) operation makes u a child of v by adding an edge $\{u, v\}$, which coalesces T_1 and T_2 into a single tree T. The operation produces an ETS for T.

Let Σ_1 (or Σ_2, resp.) be the Euler tour of T_1 (or T_2, resp.). An Euler tour of Σ of T can be derived as follows:

1. Identify an arbitrary occurrence of v in Σ_2.
2. Insert another occurrence of v right after the above one.
3. Put Σ_1 in between the above two occurrences.

Lemma 11.3. *A link operation can be supported in $O(\log(|T_1| + |T_2|))$ time.*

The proof should have become obvious, and is omitted.

11.2.3 Re-root

Remember that the ETS of T depends on the root r. Given any node $u \neq r$, the *re-root*(u) operation roots T at u, and produces an ETS consistent with the new root.

Let Σ be the current Euler tour of T (rooted at r). We can obtain a new Euler tour Σ_{new} (rooted at u) as follows:

1. Identify an arbitrary occurrence of u. Let Σ_1 be the subsequence of Σ from that occurrence to the end. Let Σ_2 be the subsequence obtained by trimming Σ_1 from Σ.
2. Delete the first vertex of Σ_2 (which must be r).
3. $\Sigma_{new} = \Sigma_1 : \Sigma_2$, where ":" denotes concatenation.
4. Append u to Σ_{new}.

See Figure 11.3 for an illustration. The correctness follows from the cyclic view explained in Section 11.1.3, and makes a good exercise for you.

Example. . Consider re-rooting the tree of Figure 11.1 at $u = \texttt{C}$. Before the operation, $\Sigma = \texttt{ABACDCECFCAGHGA}$. If $\Sigma_1 = \texttt{CFCAGHGA}$, then $\Sigma_2 = \texttt{ABACDCE}$. The procedure outputs $\Sigma_{new} = \texttt{CFCAGHGABACDCEC}$, which is indeed an Euler tour of T rooted at $\texttt{C}$. □

Lemma 11.4. *A re-root operation can be supported in* $O(\log |T|)$ *time.*

The proof is obvious and omitted.

11.3 Dynamic connectivity

We can now (easily) solve the dynamic connectivity problem on trees. Build an ETS on every tree of F, and support each operation as follows.

Insert(u, v)**.** First, identify the accommodating tree $T_1 \in F$ of u, and similarly T_2 for v. Let the ETS of T_1 (or T_2) be Υ_1 (or Υ_2, resp.). Re-root Υ_1 at u, and then perform a *link*(u, v) operation. The cost is $O(\log n)$ by Lemmas 11.3 and 11.4.

Deletion(u, v)**.** Let $T \in F$ be the tree containing the edge $\{u, v\}$. Simply perform *cut*(u, v) on the ETS of T. The cost is $O(\log n)$ by Lemma 11.2.

Connected(u, v)**.** Let $T_1 \in F$ be the tree containing u. Identify the leaf node in the ETS Υ_1 of T_1 which contains an arbitrary occurrence of u. Ascend from that leaf to the root r_1 of Υ_1. In the same manner, find the root r_2 of the ETS Υ_2 of the tree $T_2 \in F$ containing v. Declare "u connected to v" if and only if $r_1 = r_2$. The cost is $O(\log n)$ because every ETS has height $O(\log n)$.

11.4 Augmenting an ETS

Recall that we obtained the count BST (in Section 2.1.3) by augmenting the BST with aggregate information at internal nodes. In this section, we will apply the same type of augmentation to the ETS to (significantly) enhance its power.

11.4.1 Weighted vertices and trees

Commutative monoids. In discrete mathematics, a *commutative monoid* is a pair $(W, \oplus)$ where

- W is a set of elements called the *domain*;
- $\oplus$ is an operation closed on W (i.e., for any $w_1, w_2 \in W$, $w_1 \oplus w_2 \in W$);
- $\oplus$ is commutative (i.e., $w_1 \oplus w_2 = w_2 \oplus w_1$) and associative (i.e., $w_1 \oplus w_2 \oplus w_3 = w_1 \oplus (w_2 \oplus w_3)$);
- W has an *identity element* I satisfying $w \oplus I = w$ for any $w \in W$.

The following are some semi-groups commonly encountered in practice:

- $(\mathbb{R}, +)$: addition is closed on real numbers; $I = 0$.

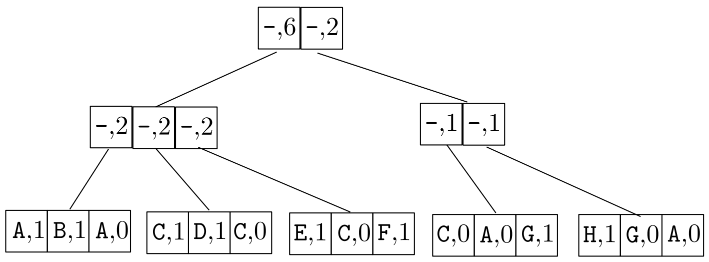


Figure 11.4: An augmented ETS (edge pointers omitted)

- $(\mathbb{R}, \min)$: minimization is closed on real numbers; $I = \infty$;
- $(\{0,1\}, \vee)$: OR is closed on $\{0,1\}$; $I = 0$.

For any subset $S \subseteq W$, we refer to

$$\bigoplus_{w \in S} w$$

as the *sum* of the elements in S.

Unless otherwise stated, for all the monoids in our discussion, we assume that

- each element in W can be stored in one cell;
- each evaluation of $\oplus$ takes constant time.

Vertex/tree weights. Fix a monoid $(W, \oplus)$. Suppose that T is *weighted* in the sense that every vertex u in the tree T is associated with a *weight* $w(u)$ drawn from W. The *weight* of T is defined as

$$\bigoplus_{u \text{ in } T} w(u).$$

By choosing $(W, \oplus)$ appropriately, we endow the weight of T with various semantics. For instance, if $(W, \oplus) = (\mathbb{R}, +)$ and every vertex is associated with weight 1, the weight of T is precisely the number of nodes in T. As another example, if $(\{0,1\}, \vee)$ and every vertex is associated with weight either 1 (black) or 0 (white), the weight of T indicates *whether* T has any black nodes.

11.4.2 Maintaining and querying weights

Let T be a weighted tree. Suppose that, in addition to the operations in Section 11.1, we want to support:

- **weight-update**(u, x) where u is a vertex in T and $x \in W$: change $w(u)$ to x.
- **tree-weight**: report the weight of T.

We can achieve the purpose by slightly augmenting the ETS Υ of T. Let Σ be the underlying Euler tour. For every vertex u in T, we

- store $w(u)$ at the leaf element in Υ corresponding to an arbitrary (e.g., the first) occurrence of u in Σ;

- store I (the identical element of the monoid; see Section 11.4.1) at the leaf elements corresponding to all the other occurrences of u;
- record (say, in a separate array) a pointer to the occurrence carrying $w(u)$.

Also, at every routing element e_{route} of Υ, we store the sum of the weights in all the leaf entries underneath e_{route}.

Example. Suppose that the monoid is $(\mathbb{R}, +)$, and that each vertex in the tree of Figure 11.1 is associated with weight 1. Figure 11.4 augments the structure in Figure 11.2. A leaf element is in the form "u, w" where u is a vertex and w a weight. A non-leaf element is in the form "-, w", where - is a routing element (which is unimportant and hence omitted), and w a weight. $\square$

Lemma 11.5. *All the statements below are true:*

- *After augmentation, the ETS still retains the performance in Lemmas 11.2-11.4.*
- *Each* **weight-update** *can be performed in* $O(\log |T|)$ *time.*
- *Each* **tree-weight** *can be performed in* $O(1)$ *time.*

The proof is left as an exercise.

11.5 Remarks

The Euler-tour structure we described is an adaptation of the structure developed by Henzinger and King in [22].

Exercises

Problem 1. Prove Proposition 11.1.

Problem 2. Prove Lemma 11.2.

Problem 3. Prove the correctness of the *re-root* algorithm in Section 11.2.3.

(Hint: Proposition 11.1.)

Problem 4. Prove Lemma 11.5.

(Hint: review an exercise in Lecture 2 about the "count 2-3 tree".)

Problem 5 (colored vertices). Same settings as in the dynamic connectivity problem. Suppose that each vertex is colored black or white. Design a data structure to satisfy all the requirements below:

- **insert**, **delete**, and **connected** still in $O(\log n)$ time.
- given a vertex $u \in V$, change its color in $O(\log n)$ time.
- given a vertex $u \in V$, find in $O(\log n)$ time the number of black vertices in the tree of F containing u.

Problem 6*. The same settings as in Problem 4, but one more requirement:

- given a vertex $u \in V$, find in $O(\log n)$ time an (arbitrary) black vertex in the tree of F containing u, or indicate that the tree has no black vertices.

(Hint: top-down search in a 2-3 tree.)

Problem 7*. Let T be a tree where each vertex is colored black or white. Describe how to store T in an augmented ETS to support the following operation in $O(\log |T|)$ time:

- given an edge $\{u, v\}$ in T, find the number of black vertices in $T^u_{u,v}$ (defined in Section 11.1.1).

(Hint: you can achieve the purpose using *cut*, *tree-weight*, and *link* as black boxes.)

Problem 8* (colored edges). Same settings as in the dynamic connectivity problem. Suppose that each edge is colored black or white. Design a data structure to satisfy all the requirements below:

- **insert**, **delete**, and **connected** still in $O(\log n)$ time.
- given an edge $\{u, v\}$ in the forest, change its color in $O(\log n)$ time.
- given a vertex $u \in V$, find in $O(\log n)$ time the number of black edges in the tree of F containing u.
- given a vertex $u \in V$, find in $O(\log n)$ time an (arbitrary) black edge in the tree of F containing u, or indicate that the tree has no black edges.

(Hint: convert the problem to one with colored vertices.)

Lecture 12: Dynamic Connectivity on a Graph

This lecture will tackle the *dynamic connectivity problem* in its general form. Specifically, we want to store an undirected graph $G = (V, E)$ in a data structure that supports the following operations:

- **insert**(u, v): add an edge $\{u, v\}$ into E;
- **delete**(u, v): remove an edge $\{u, v\}$ from E;
- **connected**(u, v): return whether vertex $u \in V$ is *connected* to vertex $v \in V$ (namely, whether a path exists between them).

We consider that G has no edges at the beginning.

If no deletions are allowed, the problem can be settled with the union-find structure of Lecture 10. Intuitively, insertions are easy because adding an edge $\{u, v\}$ always leaves u and v connected. Removing $\{u, v\}$, however, does *not* necessarily disconnect them (this is how the problem differs from the one in the previous lecture). Overcoming the obstacle requires new ideas.

Set $n = |V|$. Naively, each insertion/deletion can be supported in $O(|E|)$ time while ensuring constant time for **connected**. In this lecture, we will describe a structure developed in [23] of $\tilde{O}(n)$ space that performs all operations in $\tilde{O}(1)$ amortized time. Recall that $\tilde{O}(.)$ hides $\text{polylog}\, n$ factors. We will not be concerned with such factors when the major competitor (i.e., the naive solution) has update cost $\Omega(n)$.

Notations: For simplicity, we will assume that n is a power of 2. Set $h = \log_2 n$. For a tree T, $|T|$ represents the number of nodes in T. If u is a vertex, $u \in T$ indicates that u belongs to T.

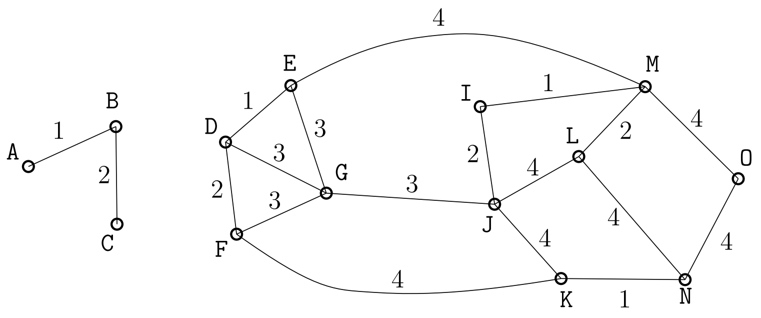
12.1 An edge leveling technique

12.1.1 Spanning trees, spanning forests, and Kruskal's algorithm

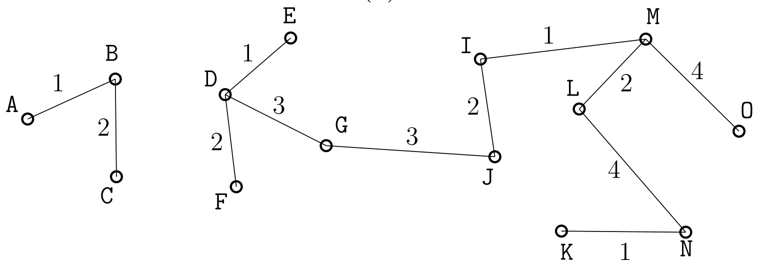
If G is connected, a *spanning tree* of G is a tree made of $|V| - 1$ edges in E (such a tree necessarily includes all the vertices in V). If G is not connected, then a *spanning forest* of G is a set F of trees, where each tree in F is a spanning tree of a different connected component (CC) of G.

We will preserve the connectivity of G by maintaining a spanning forest F. Two vertices $u, v \in V$ are connected if and only if they appear in the same tree in F. Remember that we have learned a powerful tool for managing trees, i.e., the Euler-tour structure (ETS). Indeed, we will store each tree of F in an ETS to process **connected**(u, v) efficiently.

The challenge is to update F along with edge insertions and deletions. For this purpose, we need to be careful in choosing the F to maintain. Our strategy will be closely related to Kruskal's algorithm for finding a *minimum spanning forest* (MSF). More specifically, we will give each edge



(a)



(b)

Figure 12.1: (a) shows a weighted graph, and (b) gives an MSF.

a *weight* which is a non-negative integer. The *weight* of F is the sum of weights of all the edges therein. F is an MSF if its weight is the minimum among all the spanning forests. Kruskal gave the following algorithm for finding an MSF:

```
Kruskal
1. $F \leftarrow$ the set of vertices, each regarded as a tree (of size 1)
2. while $\exists$ edge $\{u, v\}$ where $u, v$ are in different trees in $F$ do
   /* call $\{u, v\}$ a cross edge */
3.     $e \leftarrow$ a cross edge with the smallest weight
4.     merge two trees in $F$ with $e$
5. return $F$
```

We will maintain an F that can be thought of as having been picked by the above algorithm.

Example. Figure 12.1.1(a) shows a graph where the number next to each edge indicates its weight. Figure 12.1.1(b) is one possible MSF that can be output by **Kruskal**. □

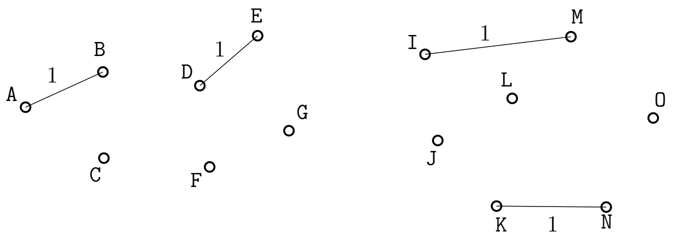
The following is a useful fact (from the undergraduate level) that will be useful:

MSF property: Let e be an edge not in F. Adding e to F creates a cycle. We call e a *short-cut edge* if the weight of e is strictly less than the weight of another edge in the cycle. The MSF property says that F is an MSF *if and only if* no short-cut edges exist.

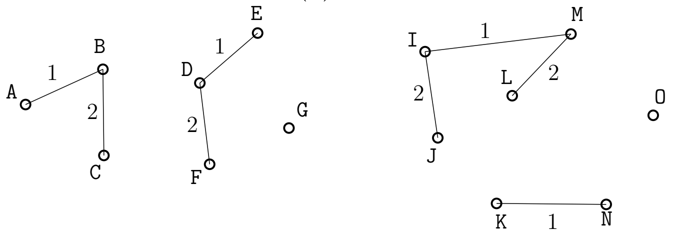
12.1.2 Edge leveling

We assign each edge $e \in E$ a *level* (a.k.a. its weight) — denoted as *level*(e) — which is an integer between 1 and h. Define for each $i \in [1, h]$:

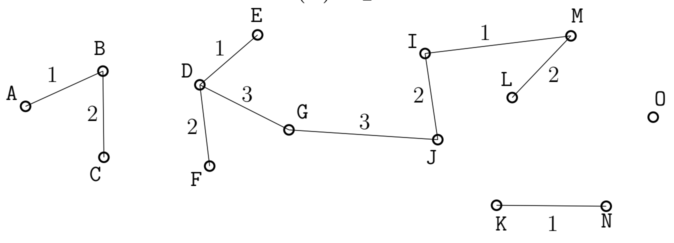
$$E_i = \text{the set of edges in } E \text{ with level } \textit{at most } i.$$



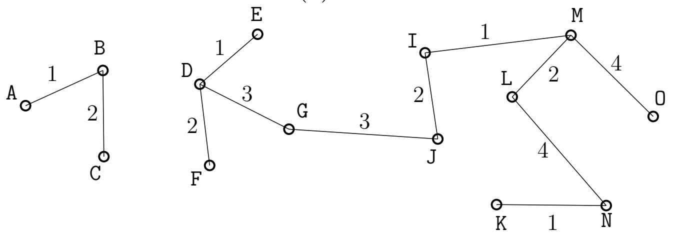
(a) F_1



(b) F_2



(c) F_3



(d) $F_4 = F$

Figure 12.2: Spanning forests for the graph in Figure 12.1.1(a)

Clearly:

$$E_1 \subseteq E_2 \subseteq ... \subseteq E_{\log_2 n} = E.$$

Accordingly, define:

$$G_i \quad = \quad \text{the graph } (V, E_i). \tag{12.1}$$

We demand:

Invariant 1: Each CC of G_i has at most 2^i vertices.

We maintain a spanning forest F_i of G_i for every $i \in [1, h]$, and make sure:

Invariant 2: For $i \in [1, h-1]$, all the edges in F_i must also be present in F_{i+1}.

Set $F = F_h$, which must be a spanning forest of G.

Example. Consider that G is the graph in Figure 12.1.1(a), where the level of each edge is indicated next to it. Here, $h = 4$. Figure 12.2 illustrates the spanning forests $F_1, F_2, ..., F_4$. $\square$

12.1.3 Connections between edge leveling and Kruskal's

Conceptually, we can imagine that the edges in F *had* been picked according to Kruskal's strategy:

1. First, keep picking level-1 cross edges until it is no longer possible to do so. This gives a spanning forest F_1 of G_1.
2. Iteratively, after F_{i-1} with $i \le h$ is ready, initialize $F_i = F_{i-1}$. Then, keep adding to F_i level-i cross edges until it is no longer possible to do so. Now, F_i is a spanning forest of G_i.
3. Repeat from 2 until $i = h$.

The following lemma formally confirms the above connection:

Lemma 12.1. *Consider any tree $T \in F_i$ (of any i) and any edge e in T. Remove e from T which disconnects T into trees T_1 and T_2. Then, any other edge connecting a node in T_1 with a node in T_2 must have level at least level(e).*

Proof. Assume the existence of an edge $e' = \{u, v\}$ of level $j < level(e)$ such that $u \in T_1$ and $v \in T_2$. This implies the existence of a path Π from u to v in F_j. By Invariant 2, all the edges in Π must be in F_i because $j < level(e) \le i$; this means that Π must be in T.

But since Π cannot contain e, we have found two different edges between T_1 and T_2 (i.e., e and some edge on Π), contradicting the fact that T is a tree. $\square$

Corollary 12.2. *F_i is an MSF of G_i for each $i \in [1, h]$.*

Proof. Immediate from the previous lemma and the MSF property (Section 12.1.1). $\square$

12.2 Dynamic connectivity

For each vertex u in G and each level $i \in [1, h]$, we store a linked list for:

$$L_i(u) \quad = \quad \{\text{the set of level-}i\text{ edges incident to }u\}. \tag{12.2}$$

Also, for each $i \in [1, h]$, build an ETS (Lecture 11) on each tree $T \in F_i$, denoted as $\Upsilon(T)$.

The subsequent discussion will concentrate on maintaining the graphs $G_1, ..., G_h$ and their spanning forests $F_1, ..., F_h$. Once this is clear, generating the necessary operations on the linked lists and ETS's becomes elementary exercises.

12.2.1 Connected

Handling a **connected**(u, v) operation amounts to finding out whether u and v belong to the same tree in F. We can do so in $\tilde{O}(1)$ time (Lecture 11) using the ETS's.

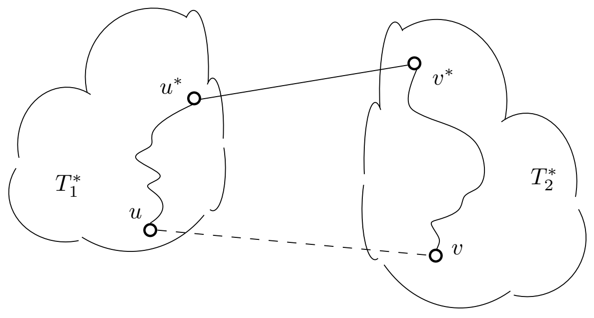


Figure 12.3: Proof of Lemma 12.4

12.2.2 Insertion

To perform an **insert**(u, v), we set the level of the new edge $\{u, v\}$ to h, and add it to G_h. If u and v are not connected, we link up with $\{u, v\}$ the trees in F containing u and v, respectively. This also takes $\tilde{O}(1)$ time (Section 12.2.1 and Lecture 11). It is obvious that Invariants 1 and 2 are still satisfied.

12.2.3 Deletion

Consider the deletion of an edge $e_{old} = \{u^*, v^*\}$. Set $\ell = level(e_{old})$. If e_{old} is not in F, no F_i of any i needs to be altered; and we finish by deleting e_{old} from $G_\ell, G_{\ell+1}, ..., G_h$. The subsequent discussion considers the opposite.

Replacement edges. Removing e_{old} from its tree $T^* \in F$ disconnects T^* into two trees T_1^*, T_2^*. Our goal is to find a *replacement edge* between T_1^* and T_2^* to connect them back into one tree in F. Of course, such a replacement edge may not exist, in which case T_1^*, T_2^* are now spanning trees of two different CCs.

Proposition 12.3. *A replacement edge must have level at least ℓ.*

Proof. Immediate from Lemma 12.1. □

Lemma 12.4. *If $\{u, v\}$ of level $i \geq \ell$ is a replacement edge, then u, v, u^*, v^* are all in the same CC of G_i.*

Proof. Since $e = \{u, v\}$ is not in T^*, adding it to T^* creates a cycle passing u, v, u^*, v^* (MSF property). See Figure 12.3. Furthermore, e must have the largest level (a.k.a. weight) in the cycle (MST property). Therefore, the four vertices are connected by a path consisting of edges with weight at most i. □

Algorithm. First remove e_{old} from all of $G_\ell, G_{\ell+1}, ..., G_h$ and $F_\ell, F_{\ell+1}, ..., F_h$. Next, we aim to find a replacement edge whose level is *as low as possible*, starting with $i = \ell$:

```
replacement($i$)
/* find a replacement edge of level $i$, if exists */
/* let $T$ be the tree in $F_i$ used to contain $e_{old}$; deleting $e_{old}$ disconnects $T$ into trees $T_1$ and $T_2$;
   w.o.l.g., assume $|T_1| \le |T_2|$ */
1. for each edge $e = \{u, v\}$ in $T_1$ of level $i$ do
2.     set $level(e)$ to $i - 1$, and add $e$ to $G_{i-1}$
       /* $i \ge 2$ by Invariant 1 (otherwise $T$ cannot have two edges $e_{old}$ and $e$) */
3.     connect two trees in $F_{i-1}$ with $e$
       /* Proposition 12.6 will prove that $u, v$ are in different CCs before the addition of $e$ to
       $G_{i-1}$ */
4. while $T_1$ has a vertex $u$ on which there is an edge $e = \{u, v\}$ of level $i$ do
5.     if $e$ is a replacement edge then
6.         return $e$
       else
7.         set $level(e)$ to $i - 1$, and add $e$ to $G_{i-1}$
8. return failure
```

If **replacement** returns failure, we increase i by 1 and try again, until i has exceeded h. If, on the other hand, a replacement edge e is found, we add e to $F_i, F_{i+1}, ..., F_h$.

Example. Suppose that we want to delete the edge $e_{old} = \{\mathtt{G}, \mathtt{J}\}$ in Figure 12.1.1(a), assuming $F_1, ..., F_4$ as in Figure 12.2. Thus, $\ell = 3$.

Consider the execution of **replacement**(3). Figure 12.4(a) shows the current G after deleting e_{old}, while Figure 12.4(b) illustrates T_1 (the left tree) and T_2 (the right); note that T_1 and T_2 are what remains after removing e_{old} from the largest spanning tree in Figure 12.2(c). The algorithm attempts to find a replacement edge of level 3 to reconnect T_1 and T_2. Lines 1-3 push all the level-3 edges in T_1 to level 2 (only one such edge {D, G}), yielding the situation in Figures 12.4(c) and (d). Lines 4-8 enumerate every level-3 edge incident to a vertex in T_1 (i.e., {E, G} and {F, G}). Since no such edges make a replacement edge, their levels are reduced to 2. The current situation is in Figures 12.4(e) and (f). The procedure **replacement**(3) returns failure.

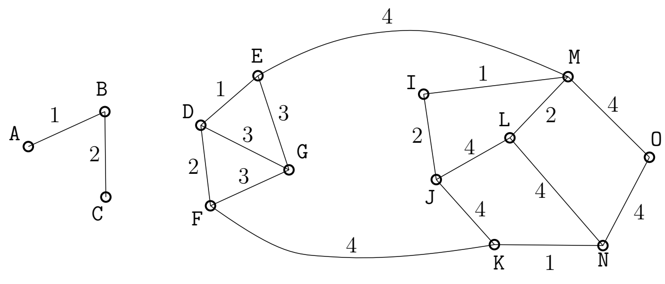
Next, we execute **replacement**(4), in an attempt to find a cross edge of weight 4. The current graph is as shown in Figure12.5(a) (same as Figure 12.4(e)). Figure 12.4(b) illustrate T_1 (the left tree) and T_2 (the right). Compare them to the right spanning tree in Figure 12.2(d) and understand what has caused the differences. The algorithm finds a replacement edge $\{\mathtt{G}, \mathtt{J}\}$ of level 4. No more changes are done to G (Figure 12.5(c)), but we use $\{\mathtt{G}, \mathtt{J}\}$ to link up T_1 and T_2 (Figure 12.5(d)), which yields a spanning tree in $F_4 = F$. □

Proposition 12.5. *Invariant 2 holds at all times.*

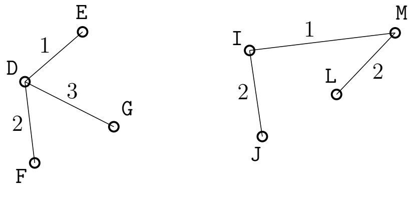
Proof. For each line in **replacement**, it is easy to prove that if Invariant 2 holds before the line, this is still true after the line. □

To prove the algorithm's correctness, we still need to show:

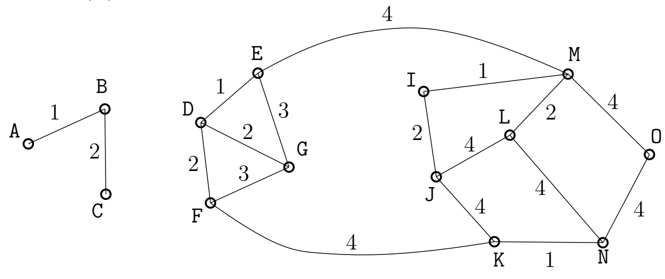
- **Claim 1:** If **replacement**(h) returns failure, no replacement edge exists.
- **Claim 2:** For each $i \in [1, h]$, F_i is still a spanning forest of G_i.



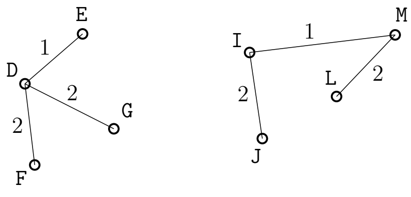
(a) G at the beginning of **replacement**



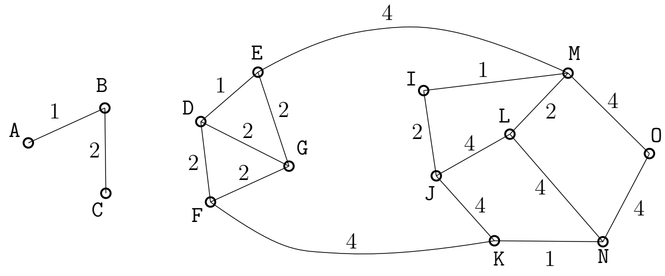
(b) T_1, T_2



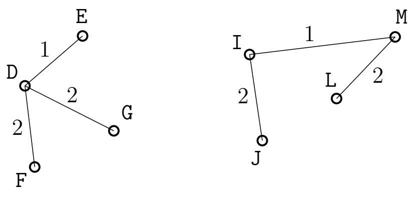
(c) G after Lines 1-3



(d) T_1, T_2



(e) G at the end



(f) T_1, T_2

Figure 12.4: Illustration of **replacement**(3)

- **Claim 3:** Invariant 1 still holds after the algorithm finishes.

Claim 1 is in fact a corollary of Lemma 12.4, and left as an exercise for you to prove. We will prove the other two claims in the following subsections.

12.2.4 Proof of Claim 2

We will establish the claim by proving a series of facts about **replacement**.

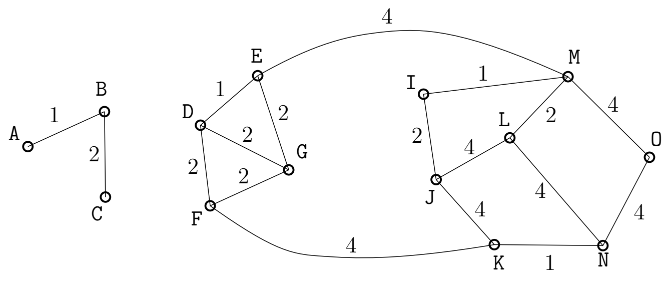
Proposition 12.6. *Consider one iteration of Lines 2-3. If F_{i-1} is a spanning forest of G_{i-1} before Line 2, it remains so after Line 3.*

Proof. It suffices to show that, for the edge $e = \{u, v\}$ identified by the iteration at Line 1, the vertices u and v must be in different CCs of G_{i-1}.

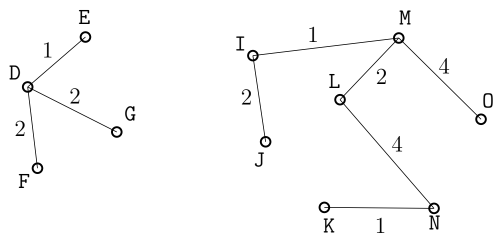
Suppose that this is not true. Consider the moment before *level*(e) is decreased at Line 2. There exists a path Π in F_{i-1} connecting u and v. All the edges in Π must belong to F_i (Proposition 12.5). But then Π and e make a cycle in F_i, giving a contradiction. $\square$

Proposition 12.7. *F_{i-1} remains as a spanning forest of G_{i-1} after each time Line 7 is executed.*

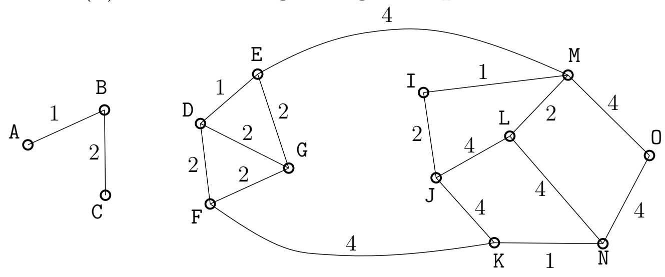
Proof. True because, right before the line, u and v must be connected in T_1 by a path in G_{i-1}. $\square$



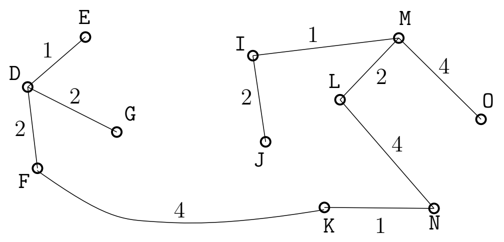
(a) G at the beginning of **replacement**



(b) T_1, T_2



(c) G at the end



(d) T_1, T_2 merged by replacement edge $\{\texttt{F}, \texttt{K}\}$

Figure 12.5: Illustration of **replacement**(4)

Proposition 12.8. *If* **replacement**(i) *returns failure, F_i is a spanning forest of G_i.*

Proof. Consider the connected component C of G_i represented by T before the removal of e_{old}. No new vertex can join C because edge levels can only decrease. Every vertex of C is in either T_1 or T_2. The edges in T_1 indicate that the vertices in T_1 are indeed connected by edges of level at most i. Same for T_2. That **replacement** returns failure indicates that no edges of level i exist between T_1 and T_2. Lemma 12.1 indicates that no edges of level less than i exist between T_1 and T_2, either. Therefore, T_1 and T_2 are now spanning trees of two CCs in G_i. □

Proposition 12.9. *After adding the replacement edge e to F_j where $j \geq i$, F_j is a spanning forest of G_j.*

Proof. Consider the connected component C of G_j represented by T before the removal of e_{old}. No new vertex can join C because edge levels can only decrease. Every vertex of C is in either T_1 or T_2. The edges in T_1 indicate that the vertices in T_1 are indeed connected by edges of level at most j. Same for T_2. The discovery of the edge e ascertains that every vertex in T_1 is connected to a vertex in T_2 by a path of edges with level at most j. The tree obtained by coalescing T_1 and T_2 with e is therefore a spanning tree of C. □

This completes the proof of Claim 2.

12.2.5 Proof of Claim 3

Fix an $i \in [1, h]$, and consider the execution of **replacement**(i). The following fact should have become easy to prove (left as an exercise):

Proposition 12.10. *After* **replacement**(i)*, the tree T_1 is the only new spanning tree in F_{i-1}, merging possibly several spanning trees originally in F_{i-1}.*

Hence, to prove Claim 3, it suffices to show that $|T_1| \leq 2^{i-1}$. For this purpose, note first that $|T| \leq 2^i$ due to Invariant 1 because T was a spanning tree in G_i before e_{old} disappeared. Thus, $|T_1| \leq 2^{i-1}$ follows from the fact that $|T_1| \leq |T_2|$ and $|T_1| + |T_2| = |T|$.

12.2.6 Implementation

Replacement can be efficiently implemented using ETS's:

- Obtain the size of a tree in F_i. See Section 11.4.
- At Line 1, the level-i edge e (of T_1) can be found in $\tilde{O}(1)$ time. This was an exercise in Lecture 11 (colored edges; hint: give a special color to each level-i edge).
- Line 2 is easy.
- Line 3 takes $\tilde{O}(1)$ time. See the same exercise as in the 1st bullet.
- At Line 4, an edge e can be found in $\tilde{O}(1)$ time. This was an exercise in Lecture 11 (colored vertices; hint: give a vertex a special color if it has level-i edges).
- The if-condition Line 5 can be checked in $\tilde{O}(1)$ time (a **connected** operation on trees).
- Line 5 takes $\tilde{O}(1)$ time. See the same exercise as in the 4th bullet.
- Line 7 is easy.

We also need to update the linked lists on all the $L_i(u)$'s (see (12.2)) whenever an edge moves from G_i to G_{i-1} for some $i \geq 2$. This can be trivially done in $O(1)$ time per move.

12.2.7 Amortization

Next, we will prove that the total cost of all the deletions is $\tilde{O}(m)$, where m is the number of edges that have ever existed in G. Since every edge must be added by an insertion, we can amortize the $\tilde{O}(m)$ cost over all the insertions such that each insertion bears only $\tilde{O}(1)$ cost.

By implementing our structure as in Section 12.2.6, we know that each deletion takes $O(1) + \tilde{O}(x)$ time, where x is the number of times we *demote* an edge, i.e., decreasing its level by 1. What is the largest possible number of demotions of all deletions? The answer is clearly $mh = \tilde{O}(m)$ because there are h levels, and edge levels never increase. We thus conclude that all deletions require $\tilde{O}(m)$ time.

12.3 Remarks

The dynamic connectivity algorithm discussed in this lecture is based on an approach developed by Holm, Lichtenberg, and Thorup in [23]. That paper also gives the precise polylog n factors we omitted.

Exercises

Problem 1. Prove the MSF property (Section 12.1.1).

Problem 2. Prove Claim 1.

Problem 3. Prove Proposition 12.10.

Problem 4. Verify all the bullets in Section 12.2.6.

Problem 5. Suppose that we want to support one more operation in the dynamic connectivity problem:

- **CC-size**(u): return the number of nodes in the CC that contains the given vertex $u \in V$.

Explain how to extend our structure to support the above operation in $\tilde{O}(1)$ amortized time, while retaining the same performance on **insert**, **delete**, and **connected**.

Problem 6. Same settings as in the dynamic connectivity problem, except that every vertex in G is colored black or white. Besides **insert**, **delete**, and **connected**, we also want to support:

- **blackest-CC**: return any node in a CC with the largest number of black vertices.

Describe a structure that supports all operations in $\tilde{O}(1)$ amortized time.

Lecture 13: Range Min Queries (Lowest Common Ancestor)

This lecture discusses the *range min query* (RMQ) problem, where we want to preprocess an array A of n real values to support:

> **Range min query:** Given integers x, y satisfying $1 \le x \le y \le n$, report $\min_{i=x}^{y} A[i]$.

The problem can be easily solved by an augmented BST (Section 2.1.3) which uses $O(n)$ space, and answers a query in $O(\log n)$ time. Today, we will learn an optimal structure that uses $O(n)$ space and answers a query in $O(1)$ time.

Closely related is the *lowest common ancestor* (LCA) problem where we want to preprocess a rooted tree T to support:

> **LCA query:** Given two nodes u, v in T, return their lowest common ancestor in T.

As you will explore in exercises, the RMQ and LCA problems turn out to be equivalent. We will focus on RMQ in the lecture.

We will consider that the elements in A are distinct (this assumption does not lose any generality; why?). For any x, y satisfying $1 \le x \le y \le n$, define

$$\mathit{minindex}_A(x, y) = \arg\min_{i=x}^{y} A[i].$$

In other words, if $k = \mathit{minindex}_A(x, y)$, then $A[k]$ is the smallest in $A[x], A[x+1], ..., A[y]$. The goal of an RMQ is to find k.

Notations. Given any $x, y \in [1, n]$, $A[x : y]$ is the subarray of A that starts from $A[x]$ and ends at $A[y]$. Specially, if $x > y$, $A[x : y]$ denotes the empty set.

13.1 How many different inputs really?

At first glance, there seems to be an infinite number of inputs because each element in A can be an arbitrary real number. This pessimistic view hardly touches the essence of the problem.

Let us define the *rank permutation* of A as a permutation R of $\{1, 2, ..., n\}$ such that, for each $i \in [1, n]$, $R[i]$ equals j if $A[i]$ is the j-th smallest element in A. What matters for RMQ are not the actual values in A, but instead, is its rank permutation. This is because, regardless of the content of A, we always have:

$$\mathit{minindex}_A(x, y) = \mathit{minindex}_R(x, y).$$

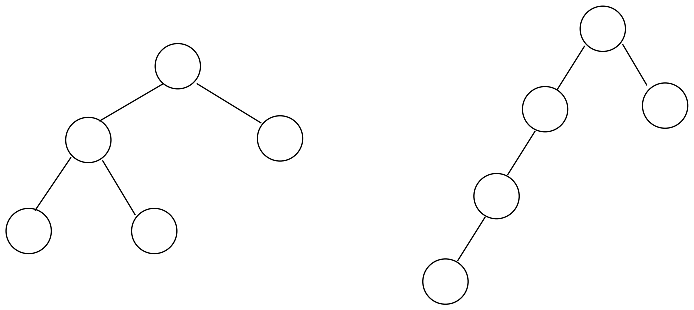


Figure 13.1: The left is the cartesian tree for $A = (4, 2, 5, 1, 3)$, and the right for $A = (4, 3, 2, 1, 5)$.

Example. Suppose that $n = 5$, $A_1 = (16, 7, 20, 2, 10)$ and $A_2 = (25, 11, 58, 3, 12)$. $R = (4, 2, 5, 1, 3)$ is the rank permutation of both A_1 and A_2. $\square$

It thus follows that there are at most $n!$ inputs that are "really" different. However, even this is a serious over-estimate! The following example allows you to see the reason intuitively:

Example. Suppose that $n = 5$, $R_1 = (4, 2, 5, 1, 3)$ and $R_2 = (3, 2, 4, 1, 5)$. For any x, y satisfying $1 \le x \le y \le n$, we always have $minindex_{R_1}(x, y) = minindex_{R_2}(x, y)$. $\square$

Formally, two arrays A_1 and A_2 of size n are said to be *identical* if $minindex_{A_1}(x, y) = minindex_{A_2}(x, y)$ holds for any legal x, y. Next, we will show that the number of distinct inputs is no more than 4^n (which is considerably smaller than $n!$).

Let us define the *cartesian tree* T on A recursively:

- If $n = 0$, then T is empty.
- If $n = 1$, then T has a single node.
- Otherwise, let $k = minindex_A(1, n)$. T is a binary tree where the root's left subtree is the cartesian tree on $A[1 : k-1]$, and the root's right subtree is the cartesian tree on $A[k+1 : n]$.

See Figure 13.1 for an illustration.

Lemma 13.1. *Arrays A_1 and A_2 are identical if and only if their cartesian trees are equivalent.*

The proof is left to you as an exercise.

It is well known that there are no more than 4^n different (rooted) binary trees with n nodes (you will prove a weaker result of 16^n in the exercises). Therefore, at most 4^n distinct inputs exist.

13.2 Tabulation for short queries

Fix any s satisfying $\Omega(\log n) = s \le \frac{1}{2}\log_4 n$. Assume, without loss of generality, that n is a multiple of s. We break A into *chunks* of size s, namely, the first chunk is $A[1 : s]$, the second $A[s + 1 : 2s]$, and so on. In this section, we consider only *short* queries where the indexes x and y fall into the *same* chunk. We will describe a structure of $O(n)$ space that answers all such queries in constant time.

Each chunk can be regarded as an array B of size s. How many different queries are there for chunk B? The answer is $s(s+1)/2$, which is the number distinct pairs (x, y) satisfying $1 \le x \le y \le s$.

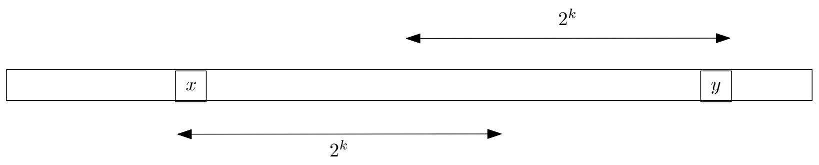


Figure 13.2: Using two pre-computed answers to cover a query

As a brute-force approach, we can store the answers for *all* possible queries. This takes $O(s^2)$ space per chunk, and hence, $O(\frac{n}{s}s^2) = O(n \log n)$ space overall. Unfortunately, this exceeds our linear space budget.

But wait! If two chunks have the same cartesian tree, they are identical (for RMQ), and hence, can share the same set of pre-computed answers! We only need at most 4^s pre-computed answer sets, because there are no more than 4^s different cartesian trees with s nodes (Section 13.1). The total amount of space required is bounded by

$$O(4^s \cdot s^2) = O\left(4^{\frac{1}{2}\log_4 n} \cdot s^2\right) = O(\sqrt{n} \cdot \log^2 n)$$

which is *significantly* less than our $O(n)$ budget!

Each pre-computed answer set is an array of size $O(s^2)$ length, referred to as an *answer array*. We store all (no more than) 4^s such arrays. For each chunk, we associate it with the starting address of its answer array. The total space is $O(n)$.

Given a query with interval $[x, y]$, we can identify the chunk covering $[x, y]$ in $O(1)$ time (think: how?), after which $minindex_A(x, y)$ can be easily acquired from the chunk's answer array in $O(1)$ time.

Remark. The method of pre-computing the answers of all queries in a small domain is known as the *tabulation technique.*

13.3 A structure of $O(n \log n)$ space

This section will describe a structure of $O(n \log n)$ space that answers an (arbitrary) RMQ in constant time.

Structure. For each $i \in [1, n]$, we store

- $minindex_A(i, j)$ for every $j = i + 1$, $i + 2^2 - 1$, ..., $i + 2^\lambda - 1$ where λ is the largest integer satisfying $i + 2^\lambda - 1 \le n$;
- $minindex_A(j, i)$ for every $j = i - 1$, $i - 2^2 + 1$, ..., $i - 2^\lambda + 1$ where λ is the largest integer satisfying $i - 2^\lambda + 1 \ge 1$.

In other words, for each $i \in [1, n]$, we pre-compute the answers of all queries whose ranges $[x, y]$ satisfy two requirements:

- $[x, y]$ covers a number of elements that is a power of 2;
- it starts or ends at i.

The number of such queries is $O(\log n)$. Therefore, the total space is $O(n \log n)$.

Query. Let $[x, y]$ be the search interval; note that it covers $y - x + 1$ elements. Set

$$\lambda = \lfloor \log_2(y - x + 1) \rfloor \tag{13.1}$$

If $y - x + 1$ is a power of 2, the query answer has explicitly been pre-computed, and can be retrieved in constant time.

Proposition 13.2. *If $y - x + 1$ is not a power of 2, $[x, y]$ is covered by the union of $[x, x + 2^\lambda - 1]$ and $[y - 2^\lambda + 1, y]$.*

The proof is obvious and hence omitted. See Figure 13.2 for an illustration. We can therefore obtain from the pre-computed answers $i = minindex_A(x, x+2^\lambda-1)$ and $j = minindex_A(y-2^\lambda+1, y)$, and then return the smaller between $A[i]$ and $A[j]$. The time required is $O(1)$.

To achieve $O(1)$ query time overall, however, we must be able to compute λ in (13.1) in constant time. This can be achieved with proper preprocessing, as you will explore in an exercise.

13.4 Remarks

Have we obtained the promised structure with $O(n)$ space and $O(1)$ query time? Well, not explicitly, but almost. All we need to do is to combine the solutions in Sections 13.2 and 13.3. This will be left as an exercise.

The elegant structure we discussed was designed by Bender and Farach-Colton [4]. It is worth pointing out that the first optimal LCA (and hence RMQ) structure is due to Harel and Tarjan [21].

Exercises

Problem 1. Prove Lemma 13.1.

Problem 2. Let T be a (rooted) binary tree of n nodes, where each internal node has two child nodes. Let $\Sigma = (u_1, u_2, ..., u_{2n-1})$ be the Euler tour obtained using the algorithm in Section 11.1.2. Σ decides a 0-1 sequence Σ' of length $2n-2$ as follows: for each $i \in [1, 2n-2]$, $\Sigma'[i] = 1$ if u_i is the parent of u_{i+1}, or 0 otherwise.

Prove: no two binary trees of n nodes can produce the same 0-1 sequence.

Problem 3. Prove: there are no more than 2^{4n} different binary trees of n nodes.

(Hint: Problem 2.)

Problem 4. Describe a structure of $O(n)$ space such that, given any integer $x \in [1, n]$, we can calculate $\lfloor \log_2 x \rfloor$ in constant time.

Problem 5. Design an optimal RMQ structure of $O(n)$ space and $O(1)$ query time.

Problem 6. Construct an optimal RMQ structure in $O(n \log\log n)$ time.

Problem 7.** Given an array A of size n, describe an algorithm to construct its cartesian tree in $O(n)$ time.

(Hint: scan A from left to right, and build the tree incrementally.)

Problem 8. Construct an optimal RMQ structure in $O(n)$ time.
(Hint: Problem 7.)

Problem 9* (RMQ implies LCA). For the LCA problem, describe a structure of $O(n)$ space and $O(1)$ query time, where n is the number of nodes in the input tree T.

(Hint: use an Euler tour of T.)

Problem 10 (LCA implies RMQ). Suppose that you know how to build an LCA structure of $O(n)$ space and $O(1)$ query time. Show that you can obtain an optimal structure for the RMQ problem.

Lecture 14: The van Emde Boas Structure (Y-Fast Trie)

.

This lecture revisits the *predecessor search* problem, where we want to store a set S of n elements drawn from an ordered domain to support:

- **Predecessor query:** given an element q (which may not be in S), return the *predecessor* of q, namely, the largest element in S that does not exceed q.

We already know that the binary search tree (BST) solves the problem with $O(n)$ space and $O(\log n)$ query time.

Our focus in the lecture will be the scenario where the domain of the elements has a finite size U. Without loss of generality, we will assume that all the elements in S are integers in $\{1, 2, ..., U\}$. We will learn the *van Emde Boas structure* (vEBS) which uses $O(n)$ space and answers a query in $O(\log\log U)$ time. Note that for practical scenarios where U is a polynomial of n, the query time is $O(\log\log U) = O(\log\log n)$, improving that of the BST.

The structure to be described also draws ideas from the *y-fast trie* (see Section 14.3 for more details).

For simplicity, we will assume that $\log_2 \log_2 U$ is an integer, namely, $U = 2^{2^x}$ for some integer $x \geq 1$ (think: why is this a fair assumption?). Also, we will assume, again without loss of generality, that S contains the integer 1 so that the predecessor of any $q \in \{1, ..., U\}$ always exists.

14.1 A structure of $O(n \log U)$ space

If $U = 4$ (which implies $n = O(1)$) or $n = O(1)$, we define the vEBS simply as a BST on S (which ensures constant space and query time). Next, we will consider $U \geq 16$.

14.1.1 Structure

We divide the domain $[1, U]$ into $\sqrt{U}$ disjoint *chunks* of size $\sqrt{U}$, namely, Chunk 1 is $[1, \sqrt{U}]$, Chunk 2 is $[\sqrt{U}+1, 2\sqrt{U}]$, and so on. Note that $\sqrt{U}$ is an integer. For each $i \in [1, \sqrt{U}]$, define

$$S_i \quad = \quad S \cap \text{ Chunk } i.$$

Chunk i is *empty* if $S_i = \emptyset$.

Collect the ids of all non-empty chunks into a *perfect* hash table H (Section 8). With H, for any $i \in [1, \sqrt{U}]$, we can check whether Chunk i is empty in constant time. We collect the ids of all non-empty chunks into a set P. Clearly, $|P| \leq \min\{n, \sqrt{U}\}$.

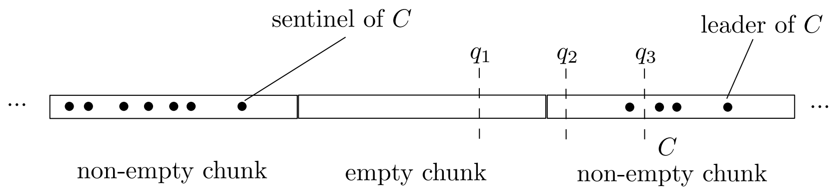


Figure 14.1: Each box shows a chunk, where points represent integers in ascending order from left to right.

Consider a non-empty chunk of id $i \in [1, \sqrt{U}]$. Recall that it corresponds to the range $[(i-1)\sqrt{U}+1, i\sqrt{U}]$. We define for the chunk:

- its *leader* to be the largest element in S_i;
- its *sentinel* to be the predecessor of $q = (i-1)\sqrt{U} - 1$, which is essentially the greatest leader from Chunks $1, 2, ..., i-1$.

See Figure 14.1.

We are now ready to define the vEBS on S recursively:

- hash table H;
- the leader and sentinel of every non-empty chunk;
- a vEBS Υ_P on P;
- a vEBS Υ_i on each non-empty S_i ($i \in [1, \sqrt{U}]$).

Note that Υ_P and each Υ_i are in a domain of size $\sqrt{U}$.

Let us analyze the space consumption. Denote by $f(n, U)$ the space of a vEBS on n integers in a domain of size U. We have:

$$f(n, U) \quad \leq \quad O(n) + f(n, \sqrt{U}) + \sum_{i=1}^{\sqrt{U}} f(|S_i|, \sqrt{U}).$$

Clearly, $f(0, U) = 0$, and $f(n, U) = O(1)$ when $1 \leq n = O(1)$ or $U \leq 4$. In an exercise, you will be asked to prove:

Lemma 14.1. $f(n, U) = O(n \log U)$.

14.1.2 Query

To find the predecessor of an integer $q \in [1, U]$, we first obtain the id $\lambda = \lfloor q/\sqrt{U} \rfloor + 1$ of the chunk that contains q. The following observations are obvious:

- If Chunk λ is empty, the predecessor of q is the leader of the first non-empty chunk to the left of Chunk λ.
- Otherwise, the predecessor of q is either the sentinel of Chunk λ or the predecessor of q in S_i.

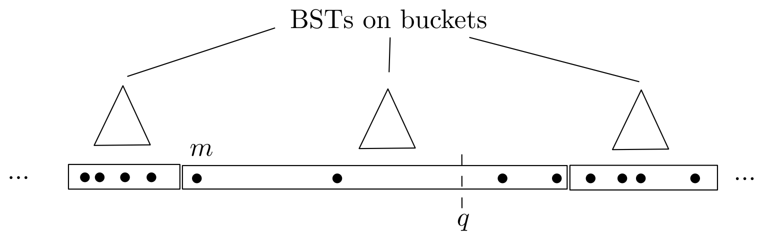


Figure 14.2: Each box shows a bucket, each of which has at most s points ($s = 4$ in this example).

Queries q_1, q_2, and q_3 in Figure 14.1 illustrates three different cases with queries q_1, q_2, and q_3.

The above naturally leads to the following algorithm. First, use H to decide in constant time whether Chunk λ is empty. If so, find the predecessor λ' of λ in P by searching Υ_P, and return the leader of Chunk λ'. Now, consider that Chunk λ is not empty. In this case, find the predecessor x of q in S_i by searching Υ_λ. If x exists, we return x as the final answer; otherwise, return the sentinel of Chunk λ.

Next, we prove that the query time is $O(\log\log U)$. Denote by $g(U)$ the query time of a vEBS when the domain has size U. No matter whether Chunk λ is empty or not, we always search a vEBS (i.e., Υ_P or Υ_i) created for a domain of size $\sqrt{U}$. Therefore:

$$g(U) \quad \leq \quad O(1) + g(\sqrt{U}).$$

Clearly, $g(4) = O(1)$. It thus follows that $g(U) = O(\log\log U)$.

14.2 Improving the space to $O(n)$

In this section, we will combine the structure of the previous section with a bucketing idea to reduce the space to linear while retaining the query time $O(\log\log U)$.

Structure. Set $s = \log_2 U$. Divide the input set S into *buckets*, each of which contains at most s elements of S. Specifically, sort S in ascending order, and then, group the first s elements into the first bucket, the next s elements into the second bucket, and so on. The total number of buckets is $O(n/s)$.

Collect the smallest element in each bucket into a set M. Build a vEBS on M, which consumes $O(|M|\log U) = O(\frac{n}{\log U}\log U) = O(n)$ space. Finally, for each bucket, create a BST (i.e., there are $O(n/s)$ BSTs). All the BSTs consume $O(n)$ space in total.

Query. Given a predecessor query q, we first find the predecessor m of q in M, which takes $O(\log\log U)$ time using the vEBS on M. The predecessor of q in the overall S must be the predecessor of q in the bucket of m, which can be found using the BST on that bucket in $O(\log s) = O(\log\log U)$ time. See Figure 14.2 for an illustration.

14.3 Remarks

The original ideas behind the vEBS are due to Boas [41], but the structure in [41] does not achieve $O(n)$ space. The version we described in this lecture is similar to what Willard [44] called the *y-fast trie*. Patrascu and Thorup [38] proved that no structure of $O(n \operatorname{polylog} n)$ space can *always* have query time strictly better than $O(\log\log U)$ (note: the BST improves the $O(\log\log U)$ query time *sometimes* — when $n \ll U$ — but not always; indeed, when $U = n^{O(1)}$, the vEBS is strictly faster than the BST). In other words, the vEBS is essentially optimal for the predecessor search.

Exercises

Problem 1. Prove Lemma 14.1.

Problem 2. Let S be a set of n integers in $\{1, ..., U\}$. Design a data structure of $O(n)$ space that answers a range reporting query (Section 2.2.1) on S in $O(\log\log U + k)$ time, where k is the number of integers reported.

Problem 3. Let S be a set of n integers in $\{1, ..., U\}$. Each integer in S is associated with a real-valued *weight*. Given an interval $q = [x, y]$ with $1 \leq x \leq y \leq U$, a *range min query* returns the smallest weight of the integers in $S \cap [x, y]$. Design a data structure of $O(n)$ space that answers a range min query in $O(\log\log U)$ time.

Problem 4. Describe how to support an insertion/deletion on the structure of Section 14.1.1 in $O(\log U)$ expected amortized time. You can assume that a perfect hash table can be updated in constant expected amortized time.

(Hint: think recursively. At the level of domain size U, you are making two insertions each into a domain size of $\sqrt{U}$.)

Problem 5.** Describe how to support an insertion/deletion on the structure of Section 14.2 in $O(\log\log U)$ expected amortized time.

(Hint: buckets can be split and merged periodically.)

Lecture 15: Leveraging the Word Length $w = \Omega(\log n)$ (2D Orthogonal Range Counting)

In all the structures discussed so far, we were never concerned about the length w of a word (a.k.a., a cell), i.e., the number of bits in a word. In the RAM model (Lecture 1), if the input set requires at least n cells to store, then $w \geq \log_2 n$ because this is the least number of bits needed to encode a memory address. Interestingly, this feature can often be used to improve data structures. We will see an example in this lecture.

We will discuss *orthogonal range counting* in 2D space. Let S be a set of n points in $\mathbb{R}^2$. Given an axis-parallel rectangle $q = [x_1, x_2] \times [y_1, y_2]$, a *range count query* reports $|S \cap q|$, namely, the number of points in S covered by q. At this stage of the course, you should know at least two ways to solve the problem. First, you can use the range tree (Section 4.4) to achieve $O(n \log n)$ space and $O(\log^2 n)$ query time (this was an exercise of Lecture 4). Second, by resorting to partial persistence, you can improve the query time to $O(\log n)$ although the space remains $O(n \log n)$ (an exercise of Lecture 7).

Today we will describe a structure with $O(n)$ space consumption and $O(\log n)$ query time. Our structure is essentially just the range tree, but incorporates *bit compression* to reduce the space by a factor of $\Theta(\log n)$.

It suffices to consider that every range count query is *2-sided*, namely, with search rectangle of the form $q = (-\infty, x] \times (-\infty, y]$ (this is known as *dominance counting*). Every general range count query can be reduced to four 2-sided queries (think: how?). We will assume that n is a power of 2; if not, simply add some dummy points to make it so. Finally, we will make the general position assumption that the points in S have distinct x- and y-coordinates (the assumption's removal was an exercise in Lecture 4).

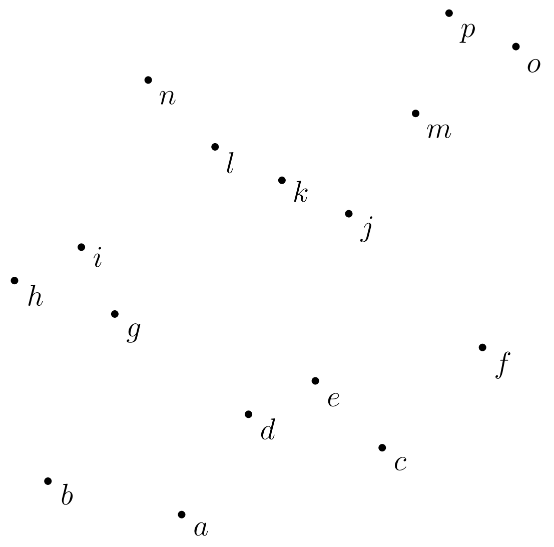
Notations. Given a point $p \in \mathbb{R}^2$, we denote by x_p and y_p its x- and y-coordinates, respectively. Given an array A of length ℓ, and any $i, j \in [1, \ell]$, we will denote by $A[i : j]$ the subarray that starts from $A[i]$ and ends at $A[j]$.

15.1 The first structure: $O(n \log n)$ space and $O(\log n)$ query time

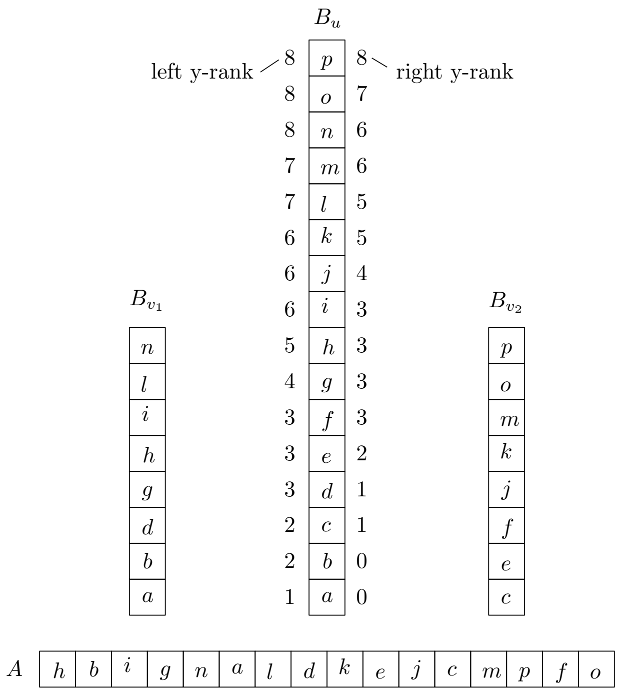
We will first explain how to achieve $O(n \log n)$ space and $O(\log n)$ query time. Our structure can be regarded as a fast implementation of the range tree.

A real number $\lambda \in \mathbb{R}$ is said to have

- *x-rank* r in S, if S has r points p satisfying $x_p \leq \lambda$;
- *y-rank* r in S, if S has r points p satisfying $y_p \leq \lambda$.



(a) The input set



(b) Array A, and the B-arrays of the root u and its child nodes v_1 and v_2

Figure 15.1: The first structure

Structure. Sort the points of S in ascending order of x-coordinate, and store the ordering in an array A of size n.

Construct a binary tree T as follows. First, create the root node which corresponds to $A[1 : n]$. In general, given a node u which corresponds to $A[a : b]$ for some integers $a < b$, create its left and right child nodes v_1, v_2 which correspond to $A[a : \frac{b-a+1}{2}]$ and $A[\frac{b-a+1}{2} + 1 : b]$, respectively. On the other hand, if $a = b$, u is a leaf of T. In any case, denote by S_u the set of points in $A[a : b]$.

Consider an arbitrary internal node u with left child v_1 and right child v_2. Note that $S_{v_1} \cup S_{v_2} = S_u$ and $S_{v_1} \cap S_{v_2} = \emptyset$. We associate u with an array B_u which sorts S_u in ascending of y-coordinate. Along with each $p \in S_u$, we store two integers:

- *left y-rank*: the y-rank of y_p in S_{v_1};
- *right y-rank*: the y-rank of y_p in S_{v_2}.

The B-arrays of all the nodes at the same level of T consume $O(n)$ space in total. Since T has $O(\log n)$ levels, the overall space of our structure is $O(n \log n)$.

Example. Figure 15.1(a) gives a set S of 16 points. The array A is shown at the bottom of Figure 15.1(b). Suppose that node u is the root of T, whose left and right child nodes are v_1 and v_2, respectively. Figure 15.1(b) also shows B_u, B_{v_1}, and B_{v_2}. The left and right y-ranks of each point in B_u are also indicated. □

Query. Let $q = (-\infty, x] \times (-\infty, y]$ be the search region. We assume that we are given the x-rank λ_1 of x in S, and the y-rank λ_2 of y in S (why is the assumption fair?).

Let us deal with a more general subproblem. Suppose that we are at a node u of T, and want to find out how many points in S_u are covered by q (if u is the root of T, $S_u = S$; and hence, the answer is precisely the final query result). We are told:

- λ_1: the x-rank of x in S_u;
- λ_2: the y-rank of y in S_u.

If u is a leaf node, S_u has only a single point; the answer can be found in constant time. Next, we consider that u has left child v_1 and right child v_2. We consider $\lambda_2 \geq 1$ because otherwise the answer is clearly 0.

Let p^* be the point at $B_u[\lambda_2]$ (p^* is the λ_2-th highest point in S_u). We distinguish two scenarios:

- Case 1: $\lambda_1 > |S_u|/2$. This means that all the points in S_{v_1} (note: $|S_{v_1}| = |S_u|/2$) have x-coordinates less than x. Thus, the number of points in S_{v_1} covered by q is exactly the left y-rank of p^*, which has already been pre-computed, and can be retrieved in constant time. However, we still need to find out how many points in S_{v_2} are covered by q. For this purpose, it suffices to solve the same subproblem recursively at v_2. But to do so, we need to prepare the x-rank of x in S_{v_2}, and the y-rank of y in S_{v_2}. Both can be easily obtained in constant time: the former equals $\lambda_1 - |S_{v_1}| = \lambda_1 - |S_u|/2$, while the latter is simply the right y-rank of p^*.
- Case 2: $\lambda_1 \leq |S_u|/2$. It suffices to find the number of points in S_{v_1} covered by q. We do so by recursively solving the subproblem at v_1. For this purpose, we need to prepare the x-rank of x in S_{v_1}, and the y-rank of y in S_{v_1}. Both can be obtained directly: the former is just λ_1, while the latter is the left y-rank of p^*.

In summary, we answer a range count query by descending a single root-to-leaf path in T, and spend $O(1)$ time at each node on the path. The query time is therefore $O(\log n)$.

15.2 Improving the space to $O(n)$

What is the culprit that makes the space complexity $O(n \log n)$? The B-arrays! For each node u in T, the array B_u has length $|S_u|$ and thus require $\Theta(|S_u|)$ words to store. Next, we will compress B_u into $O(1 + |S_u|/\log n)$ words. Accordingly, the overall space is reduced from $O(n \log n)$ to $O(n)$.

Henceforth, let s be an integer satisfying $\Omega(\log n) = s \leq \frac{1}{2}\log_2 n$. We will need:

Lemma 15.1. *With $o(n)$ pre-processing time, we can build a structure of $o(n)$ space to support the following operation in $O(1)$ time: given any bit vector of length s and any integer $t \in [1, s]$, return the number of 0's in the vector's first t bits.*

The proof is left to you as an exercise with hints.

Compressing B_u. We divide B_u into *chunks* of length s, except possibly for one chunk. Specifically, Chunk 1 includes the first s points of B_u, Chunk 2 the next s points, and so on. The last chunk may have less than s points if $|S_u|$ is not a multiple of s.

Let v_1 and v_2 be the left and right child nodes of u, respectively. For each chunk, we store two integers:

- left y-rank: the y-rank of y_p in S_{v_1}, where p is the highest point in the chunk;

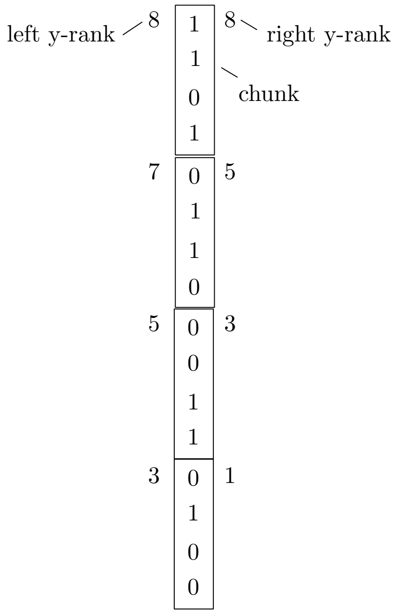


Figure 15.2: The compressed version of the array B_u in Figure 15.1

- right y-rank: the y-rank of y_p in S_{v_2}.

The total space to store the left/right y-ranks of all the chunks is $O(\lceil |S_u|/s \rceil) = O(1 + |S_u|/s)$ words, which is $O(w + \frac{w \cdot |S_u|}{s})$ bits.

For every chunk of σ points ($1 \le \sigma \le s$), we also store a *bit vector* of length σ. To explain, let the points in the chunk be $p_1, p_2, ..., p_\sigma$ in ascending order of y-coordinate. The i-th ($i \in [1, \sigma]$) bit in the bit-vector equals

- 0, if p_i comes from S_{v_1};
- 1, otherwise (i.e., p_i from S_{v_2}).

The bit vectors of all the chunks have precisely $|S_u|$ bits.

Example. Continuing the example of Figure 15.1, again let u be the root node. Figure 15.2 illustrates the compressed form of B_u. Here, $s = 4$, and B_u is cut into 4 chunks. The left and right y-ranks of each chunk are indicated outside the boxes. The bit vector of a chunk is given inside the boxes. For instance, the bit vector of the first (i.e., bottom-most) chunk is 0010, that of the second chunk is 1100, and so on. $\square$

Other than the chunks' left/right y-ranks and bit vectors, we store nothing else for u (in particular, B_u is no longer necessary). The space required for u is therefore:

$$O\left(w + \frac{w \cdot |S_u|}{s} + |S_u|\right) \text{ bits} \quad = \quad O\left(1 + \frac{|S_u|}{s} + \frac{|S_u|}{w}\right) \text{ words } = O\left(1 + \frac{|S_u|}{\log n}\right) \text{ words}$$

where the last equality used the fact $w \ge \log_2 n$.

Consider any point $p \in S_u$. Recall that, in the structure of Section 15.1, we stored the left and right y-ranks of p explicitly. This is no longer the case in our new structure. Nevertheless, the lemma below shows that this information is implicitly captured:

Lemma 15.2. *If we know that p is the r-th highest point in S_u (for some $r \in [1, |S_u|]$), we can obtain the left and right y-ranks of p in $O(1)$ time.*

Proof. Since the left and right y-ranks of p add up to r, it suffices to explain how to find the left y-rank in constant time.

Let $i = \lfloor r/s \rfloor + 1$ be the id of the chunk that contains p. If $i \geq 2$, denote by r_{prefix} the left y-rank of Chunk $i - 1$; otherwise, define $r_{\mathit{prefix}} = 0$. The value of r_{prefix} has been pre-computed and can be fetched in $O(1)$ time. Set $j = r - s(k-1)$; point p is the j-th highest point within Chunk k. Denote by $\boldsymbol{v}$ the bit-vector of the Chunk k. Use Lemma 15.1 to retrieve in $O(1)$ time the number r_{chunk} of 0's in the first j-th bits of $\boldsymbol{v}$. The left y-rank of p equals $r_{\mathit{prefix}} + r_{\mathit{chunk}}$. □

Example. Continuing the previous example, suppose that we want to find out the left y-rank of point j (see Figure 15.1) in B_u, knowing that j is the 10-th highest point in S_u. We first obtain the id 3 of the chunk containing j. Thus, $r_{\mathit{prefix}} = 5$, which is the left y-rank of Chunk 2. Within Chunk 3, point j is the second highest. In the bit-vector 0110 of the chunk, there is only $r_{\mathit{chunk}} = 1$ zero in the first 2 bits. Therefore, we conclude that the left y-rank of j must be $r_{\mathit{prefix}} + r_{\mathit{chunk}} = 6$. □

Space. We leave it as an exercise for you to prove that the overall space consumption of structure is now $O(n)$ words.

Query. Recall that the core in solving a range count query $q = (-\infty, x] \times (-\infty, y]$ is to tackle the following subproblem where, standing at an internal node u of T, we want to find out $|S_u \cap q|$, assuming that the following are known:

- λ_1: the x-rank of x in S_u;
- $\lambda_2 \geq 1$: the y-rank of y in S_u.

The algorithm in Section 15.1 spends $O(1)$ time at u before recursing into a child node of u. Lemma 15.2 allows us to obtain the left and right y-ranks of p^* in constant time, where p^* is the λ_2-th highest point in S_u. With this, the algorithm of Section 15.1 can still be implemented to run in $O(1)$ time at u.

The overall query time is therefore still $O(\log n)$.

15.3 Remarks

The structure we described is due to Chazelle [12]. When the x- and y-coordinates of all the points are integers, JaJa, Mortensen, and Shi [26] showed that the $w = \Omega(\log n)$ feature can even be used to improve the query time: they developed a structure of $O(n)$ space and $O(\log n / \log\log n)$ query time. Patrascu [37] showed that $O(\log n / \log\log n)$ query time is the best possible for any structure of $O(n \operatorname{polylog} n)$ space.

Exercises

Problem 1. Prove Lemma 15.1.

(Hint: tabulation; see Lecture 13.)

Problem 2. Prove that the structure of Section 15.2 uses $O(n)$ space.

Problem 3. Describe an algorithm to construct the structure of Section 15.2 in $O(n \log n)$ time.

Problem 4*. Make the structure of Section 15.2 fully dynamic to support each insertion and deletion in $O(\log^2 n)$ amortized time. The space consumption should still be $O(n)$. The structure must answer a range count query in $O(\log^2 n)$ time.

(Hint: logarithmic rebuilding + global rebuilding.)

Lecture 16: Approximate Nearest Neighbor Search 1: Doubling Dimension

We define a *metric space* as a pair $(U, dist)$ where

- U is a non-empty set (possibly infinite), and
- $dist$ is a function mapping $U \times U$ to $\mathbb{R}_{\geq 0}$ (where $\mathbb{R}_{\geq 0}$ is the set of non-negative real values) satisfying:
  - $dist(e, e) = 0$ for any $e \in U$;
  - $dist(e_1, e_2) \geq 1$ for any $e_1, e_2 \in U$ such that $e_1 \neq e_2$;
  - symmetry, i.e., $dist(e_1, e_2) = dist(e_2, e_1)$ for any $e_1, e_2 \in U$;
  - the triangle inequality, i.e., $dist(e_1, e_2) \leq dist(e_1, e_3) + dist(e_3, e_2)$ for any $e_1, e_2, e_3 \in U$.

We will refer to each element in U as an *object*, and to $dist$ as a *distance function.* For any $e_1, e_2 \in U$, $dist(e_1, e_2)$ is the *distance* between the two objects.

This lecture will discuss *nearest neighbor search.* The input is a set S of n objects in U. Given an object $q \in U \setminus S$, a *nearest neighbor query* reports an object $e^* \in S$ with the smallest distance to q, namely:

$$dist(q, e^*) \quad = \quad \min_{e \in S} dist(q, e).$$

The object e^* is a *nearest neighbor* of q.

Ideally, we would like to preprocess S into a data structure such that all nearest neighbor queries can be answered efficiently, no matter what the metric space is. Unfortunately, this is impossible: n distances must be calculated in the worst case, regardless of the preprocessing (we will discuss this in Section 16.4). In other words, the trivial algorithm which simply computes the distances from q to all the objects in S is already optimal. In fact, this problem is not easy even in the *specific* metric space where $U = \mathbb{N}^3$ and $dist$ is the Euclidean distance; see the remarks in Section 16.4

We therefore resort to approximation. Fix some constant $c > 1$. If $e^* \in S$ is a nearest neighbor of an object $q \in U \setminus S$, an object $e \in S$ is a *c-approximate nearest neighbor* of q if

$$dist(q, e) \quad \leq \quad c \cdot dist(q, e^*).$$

Accordingly, a *c-approximate nearest neighbor* (c-ANN) *query* returns an arbitrary c-approximate nearest neighbor of q (note: even nearest neighbors may not be unique, let alone c-ANNs). Unfortunately, this problem is still hopelessly difficult: calculating n distances is still necessary in the "hardest" metric space (Section 16.4).

Fortunately, the metric spaces encountered in practice may not be so hard, such that by pre-computing a structure of near-linear space we can answer c-ANN queries efficiently. For example, this is possible for $U = \mathbb{N}^d$ with a constant dimensionality d, and *dist* being the Euclidean distance. In this lecture, we will learn a structure for $c = 3$ that works for many metric spaces, and is *generic* because it treats objects and the function *dist* as *black boxes*. It does not matter whether the objects are multi-dimensional points or DNA sequences, or whether *dist* is the Euclidean distance (for points) or the edit distance (for DNA sequences); our structure works in exactly the same way.

Crucial to the structure is the concept of *doubling dimension* which allows us to measure how hard a metric space is. The performance of our structure is established with respect to the doubling dimension. Our structure is efficient when the doubling dimension is small (i.e., the metric space is easy), but is slow when the dimension is large (the metric space is hard). Even better, the concept is *data dependent*. More specifically, even if the metric space (U, dist) is hard, the input set S may still allow c-ANN queries to be answered efficiently, if the metric space (S, dist) has a small doubling dimension. This is useful in practice: even though c-ANN search under the edit distance may be difficult for arbitrary DNA sequences, it is possible to do much better on a *particular* set S of sequences.

We need to be clear how to measure the space and query time of a structure (remember: we will treat objects and the distance function as black boxes):

- The space of a structure is the number of memory cells occupied, plus the number of objects stored. For example, "$O(n)$ space" means not only the occupation of $O(n)$ memory, but also the storage of $O(n)$ objects.
- The query time will be measured as the sum of two terms: (i) the number of atomic operations of the RAM model, and (ii) the number of times that *dist* is invoked. For example, "$O(\log n)$ time" means that the algorithm performs $O(\log n)$ atomic operation *and* calculates $O(\log n)$ distances.

We define the *aspect ratio* of S as

$$\Delta(S) \;=\; \left(\sup_{e_1, e_2 \in S} \mathit{dist}(e_1, e_2)\right) / \left(\inf_{\text{distinct } e_1, e_2 \in S} \mathit{dist}(e_1, e_2)\right) \tag{16.1}$$

namely, the ratio between the maximum and minimum pair-wise distances in S.

Notations. We will reserve e, x, y, z for objects, and X, Y for sets of objects.

16.1 Doubling dimension

Consider an arbitrary metric space (U, dist). We will formalize its doubling dimension in three definitions:

Definition 16.1. *Let X be a non-empty subset of U. The* **diameter** *of X — denoted as $\mathit{diam}(X)$ — is the maximum distance of two objects in X, or formally:*

$$\sup_{e_1, e_2 \in X} \mathit{dist}(e_1, e_2).$$

Definition 16.2. *A non-empty $X \subseteq U$ can be* 2^λ**-partitioned** *(where $\lambda \geq 0$ is a real value) if X can be divided into (disjoint) subsets $X_1, X_2, ..., X_m$ such that*

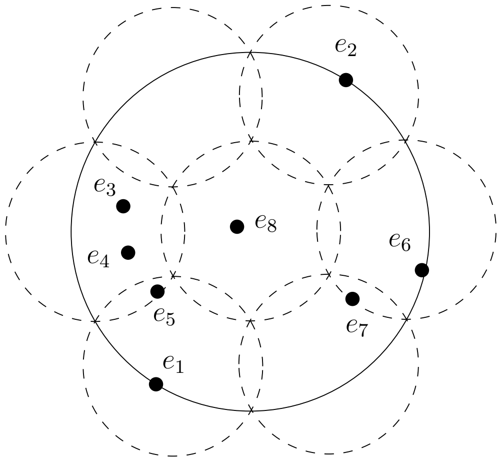


Figure 16.1: When $U = \mathbb{N}^2$ and *dist* is the Euclidean distance, any set X of points can be divided into 7 disjoint subsets whose diameters are at most $\frac{1}{2}diam(X)$.

- $m \le 2^\lambda$
- *every* X_i *(*$1 \le i \le m$*) has diameter at most* $\frac{1}{2}diam(X)$*.*

Definition 16.3. *The* **doubling dimension** *of the metric space* $(U, dist)$ *is the smallest real value* λ *such that every finite non-empty* $X \subseteq U$ *can be* 2^λ*-partitioned.*

Example. Let us look at a specific metric space where $U = \mathbb{N}^2$ and *dist* is the Euclidean distance. We will show that $(\mathbb{N}^2, \text{Euclidean})$ has a doubling dimension less than 3.

Suppose that we are given any set X of points in $\mathbb{N}^2$ with $|X| \ge 2$; Figure 16.1 shows an example where X is a finite set of 8 points. Denote by D the smallest disc covering X; in Figure 16.1, D is enclosed by the circle of the solid line. The diameter of D is at most $diam(X)$ (think: why?). We can always find 7 discs $D_1, ..., D_7$ of diameter $\frac{1}{2}diam(D)$ such that they together cover D (the proof requires only high-school geometry, and is left as an exercise); in the figure, those discs are indicated in dashed lines. Now, assign each point $e \in X$ to a disc that covers it; if e is covered by more than one disc, assign it to an arbitrary disc (but only one disc). For each $i \in [1, 7]$, define X_i as the set of points assigned to disc D_i. Thus, $X_1, ..., X_7$ partition X; and each X_i has diameter at most $\frac{1}{2}diam(D) \le \frac{1}{2}diam(X)$.

It thus follows that X can be $2^{\log_2 7}$-partitioned. Therefore, the metric space has a doubling dimension of $\log_2 7 < 3$. □

The fact below follows immediately from Definition 16.3:

Proposition 16.4. *For any non-empty subset* $X \subseteq U$*, the doubling dimension of* $(X, dist)$ *is no more than that of* $(U, dist)$*.*

16.2 Two properties in the metric space

Balls. Recall that, in $\mathbb{R}^d$, a "ball" is the set of points inside a d-dimensional sphere (a 2D ball is a disc). Next, we generalize the concept to metric spaces:

Definition 16.5. *For any object $e \in U$ and real value $r \geq 0$, the* **ball** *$B(e, r)$ includes all the objects $e' \in U$ such that $dist(e, e') \leq r$. The object e is the* **center** *of the ball, while the value r is the* **radius**.

In $\mathbb{R}^d$, a d-dimensional ball of radius r can be covered by $2^{O(d)}$ balls of radius $\Omega(r)$. A similar result holds for metric spaces too.

Lemma 16.6. *Let λ be the doubling dimension of the metric space $(U, dist)$, and $c \geq 1$ be a constant. Then, any ball $B(e, r)$ can be covered by at most $2^{O(\lambda)}$ balls of radius r/c, namely, there exist objects $e_1, ..., e_m \in U$ such that*

- $m \leq 2^{O(\lambda)}$;
- $B(e, r) \subseteq \bigcup_{i=1}^{m} B(e_i, r/c)$.

Proof. Set $X = B(e, r)$. The triangle inequality implies that $diam(X) \leq 2r$ (think: why?).

Let us first prove the theorem for $c = 2$. By definition of λ, we can divide X into subsets $X_1, ..., X_{m'}$ ($m' \leq 2^\lambda$) all of which have diameter at most r. In turn, each X_i ($1 \leq i \leq m'$) can be divided into at most 2^λ subsets, each of which has diameter at most $r/2$, and hence, can be covered by a ball with radius $r/2$ (think: why?). It thus follows that X can be covered by at most $2^\lambda \cdot 2^\lambda = 2^{2\lambda}$ balls of radius $r/2$.

The proof for the case $c \neq 2$ is left to you as an exercise. □

Constant aspect-ratio object sets. In $\mathbb{R}^d$, you can place at most $2^{O(d)}$ points in a sphere of radius 1 while ensuring the distance of any two points to be at least 1/2. The next lemma generalizes this to any metric space:

Lemma 16.7. *Suppose that the metric space $(X, dist)$ has doubling dimension λ, and that the aspect ratio of X is bounded by a constant. Then, X can have no more than $2^{O(\lambda)}$ objects.*

Proof. If $|X| = 1$, the lemma is vacuously true. Next, we consider $|X| \geq 2$. Define:

$$dist_{min} = \inf_{\text{distinct } x_1, x_2 \in X} dist(x_1, x_2)$$

Thus, $diam(X)/dist_{min} = \Delta(X) = O(1)$. This means $diam(X) \leq O(1) \cdot dist_{min}$.

Set

$$c = 4 \cdot \frac{diam(X)}{dist_{min}} = 4 \cdot \Delta(X) = O(1).$$

Take any object $x \in X$. Clearly, the entire $X \subseteq B(x, diam(X))$. By Lemma 16.6, $B(x, diam(X))$ is covered by $m \leq 2^{O(\lambda)}$ balls of radius $diam(X)/c = \frac{1}{4} dist_{min}$. Denote those balls as $B_1, ..., B_m$.

Each B_i ($1 \leq i \leq m$) can cover exactly one object in X. To see this, assume that e is the center of B_i. If B_i contains two objects $x, y \in X$, it must hold that $dist(x, y) \leq dist(x, e) + dist(e, y) \leq \frac{1}{2} dist_{min}$, which contradicts the definition of $dist_{min}$. □

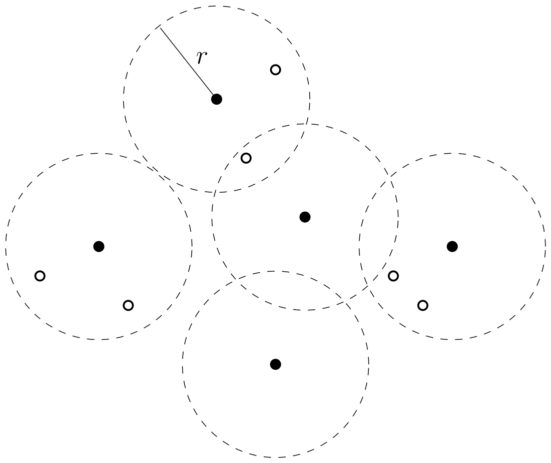


Figure 16.2: A sample net example: X is the set of all points shown, and Y the set of black points.

16.3 A 3-approximate nearest neighbor structure

We are now ready to introduce the promised 3-ANN structure. As before, denote by $(U, dist)$ the underlying metric space, and by $S \subseteq U$ the input set of $n \geq 2$ objects. Set

$$h = \lceil \log_2 diam(S) \rceil \tag{16.2}$$

where $diam(S)$ is the diameter of S (Definition 16.1). Denote by λ the doubling dimension of $(S, dist)$; note that λ can be smaller than the doubling dimension of $(U, dist)$ (Proposition 16.4).

We aim to establish:

Theorem 16.8. *There is a structure of $2^{O(\lambda)} \cdot n \cdot h$ space that answers a 3-ANN query in $2^{O(\lambda)} \cdot h$ time.*

When $\lambda = O(1)$, the space is $O(nh)$ and the query time is $O(h)$. In Section 16.4, we will discuss a number of scenarios where this is true.

16.3.1 Sample nets

Definition 16.9. *Consider any $X \subseteq S$ and any real value $r > 0$. A non-empty $Y \subseteq X$ is an* **r-sample net** *of X if the following two conditions hold:*

- *for any distinct objects $y_1, y_2 \in Y$, $dist(y_1, y_2) > r$;*
- *$X \subseteq \bigcup_{y \in Y} B(y, r)$.*

Note that the second bullet indicates that, for any object $x \in X$, Y has an object y such that $dist(x, y) \leq r$. See Figure 16.2 for an example for the metric space ($\mathbb{N}^2$, Euclidean).

16.3.2 Structure

Our strategy is to gradually "sparsify" the input set S. Define for each $i \in [0, h]$:

$$Y_i = 2^i\text{-sample net of } S.$$

The following facts are obvious:

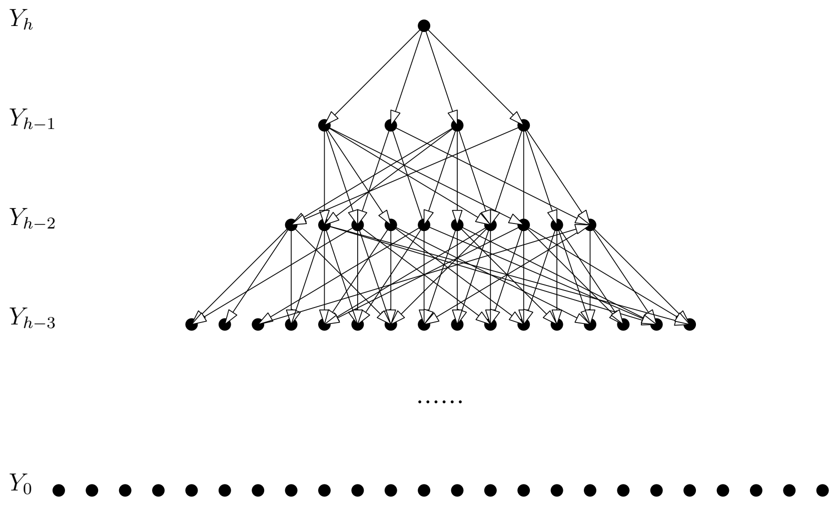


Figure 16.3: Illustration of G

- Y_h has a single object, noticing that $2^h \geq \mathit{diam}(S)$ (see (16.2));
- $|Y_i| \leq n$ for all i.

It thus follows that the total size of $Y_0, Y_1, ..., Y_h$ is $O(n \cdot h)$.

We will build a directed graph G as follows. The vertices of G form $h+1$ layers 0, 1, ..., h, where the i-th layer ($1 \leq i \leq h$) contains a vertex for each object in Y_i. Edges of G exist only between two *consecutive* layers. Specifically, an object y (a.k.a. vertex) in Y_i ($i \geq 1$) has an out-going edge to an object z (a.k.a. vertex) in Y_{i-1} if and only if

$$\mathit{dist}(y, z) \quad \leq \quad 7 \cdot 2^i. \tag{16.3}$$

See Figure 16.3 for an illustration.

For each object $y \in Y_i$, we denote by $N_i^+(y)$ the set of out-neighbors of y. Each node in $N_i^+(y)$ will be referred to as a *child* of y.

Lemma 16.10. $|N_i^+(y)| = 2^{O(\lambda)}$.

Proof. Due to Lemma 16.7, it suffices to show that $N_i^+(y)$ has a constant aspect ratio. Clearly, $N_i^+(y) \subseteq Y_{i-1}$; hence, any two distinct objects $z_1, z_2 \in N_i^+(y)$ must have distance at least 2^{i-1}. On the other hand, $\mathit{dist}(z_1, z_2) \leq \mathit{dist}(z_1, y) + \mathit{dist}(y, z_2) \leq 7 \cdot 2^i + 7 \cdot 2^i = 14 \cdot 2^i$. Therefore, $N_i^+(y)$ has an aspect ratio at most 28. □

The G constitutes our data structure. It is clear that the space consumption is $2^{O(\lambda)} \cdot n \cdot h$.

16.3.3 Query

Given a query object $q \in U \setminus S$, we descend a single path π in G as follows:

- The first node visited is the root of G, namely, the sole vertex in Y_h.
- Suppose that π contains an object (i.e., vertex) $y \in Y_i$ for some $i \geq 1$. Then, we add to π the child z of y with the smallest $\mathit{dist}(q, z)$, breaking ties arbitrarily.

Then, we return the object in π closest to q as our final answer.

The query time is clearly $2^{O(\lambda)} \cdot h$ (Lemma 16.10). In the rest of the section, we will prove that our answer is a 3-ANN of q.

Denote by e^* the (exact) nearest neighbor of q. Let $y_h, y_{h-1}, ..., y_0$ be the objects in π, where y_i belongs to Y_i for each $i \in [0, h]$. It suffices to prove that at least one of $y_h, y_{h-1}, ..., y_0$ has distance to q at most $3 \cdot dist(q, e^*)$.

Let j be the largest integer satisfying

$$dist(q, y_j) \quad > \quad 3 \cdot 2^j. \tag{16.4}$$

Note that j may not exist. Indeed, our argument proceeds differently depending on whether it does.

Case 1: j does not exist. This means $d(q, y_0) \le 3 \cdot 2^0 = 3 \le 3 \cdot dist(q, e^*)$, where the last inequality used the fact that $dist(q, e^*) \ge 1$ (recall that $q \notin S$).

Case 2: $j = h$. In other words, $dist(q, y_h) > 3 \cdot 2^h \ge 3 \cdot diam(S)$. We have:

$$\begin{aligned} dist(q, e^*) &\ge dist(q, y_h) - dist(e^*, y_h) \\ &\ge 3 \cdot diam(S) - diam(S) = 2 \cdot diam(S) \end{aligned} \tag{16.5}$$

which intuitively means that q is far away from the entire S. We can further derive:

$$\begin{aligned} dist(q, y_h) &\le dist(q, e^*) + dist(e^*, y_h) \\ &\le dist(q, e^*) + diam(S) \\ &\le 1.5 \cdot dist(q, e^*) \end{aligned}$$

where the last inequality used (16.5).

Case 3: $j < h$. It thus follows that $dist(q, y_{j+1}) \le 3 \cdot 2^{j+1}$. Next, we will argue that y_{j+1} is a 3-ANN of q.

Recall that Y_j is a 2^j-sample net of S. Hence, there must exist an object $z \in Y_j$ such that $dist(e^*, z) \le 2^j$.

Lemma 16.11. *z is a child of y_{j+1}.*

Proof.

$$\begin{aligned} dist(z, y_{j+1}) &\le dist(z, e^*) + dist(e^*, y_{j+1}) \\ &\le dist(z, e^*) + dist(q, e^*) + dist(q, y_{j+1}) \\ (e^* \text{ is the nearest neighbor}) \quad &\le dist(z, e^*) + 2 dist(q, y_{j+1}) \\ &\le 2^j + 2 \cdot 3 \cdot 2^{j+1} < 7 \cdot 2^{j+1}. \end{aligned}$$

□

Also recall that y_j is the child of y_{j+1} *closest* to q, which means:

$$dist(q, z) \ge dist(q, y_j) \ge 3 \cdot 2^j.$$

We now have:

$$\begin{aligned}
dist(q, e^*) &\geq dist(q, z) - dist(e^*, z) \\
&\geq 3 \cdot 2^j - 2^j = 2^{j+1}.
\end{aligned}$$

Therefore, $dist(q, y_{j+1}) \leq 3 \cdot dist(q, e^*)$.

We now complete the whole proof of Theorem 16.8.

16.4 Remarks

The above structure, which is due to Krauthgamer and Lee [28], is efficient when the underlying metric space $(U, dist)$ has a small doubling dimension λ. This is true when $U = \mathbb{N}^d$ for a constant dimensionality d, and *dist* is the Euclidean distance. It can be proved [3] that ($\mathbb{N}^d$, Euclidean) has a doubling dimension of $O(d) = O(1)$ (you will be asked to prove a somewhat weaker statement in an exercise). It immediately follows from Theorem 16.8 (with an improvement you will see in the exercises) that, we can store a set S of n points in $\mathbb{N}^d$ in a structure of $O(n \cdot \log \Delta(S))$ space such that a 3-ANN of any query point can be found in $O(\log \Delta(S))$ time. In comparison, for *exact* nearest neighbor search in $\mathbb{N}^3$ (Euclidean distance), no known structure can achieve $n \operatorname{polylog} n$ space and $\operatorname{polylog} n$ query time simultaneously, even if $\Delta(S)$ is a polynomial of n.

Given a point $p \in \mathbb{N}^d$, let us use $p[i]$ to denote the coordinate of p on dimension i. For any real value $t > 0$, the so-called L_t*-norm* between two points p and q is:

$$\left(\sum_{i=1}^{d} \Big| p[i] - q[i] \Big|^t \right)^{1/t}.$$

The Euclidean distance is simply the L_2 norm. It is known that the metric space ($\mathbb{N}^d$, L_t-norm) also has doubling dimension $O(d)$, regardless of t. When d is a constant, we can once again obtain an efficient 3-ANN structure using Theorem 16.8.

What if λ is large? In this case, the metric space is "hard"; and Theorem 16.8 does not work for all inputs S. In the next lecture, we will introduce another technique that permits us to deal with some hard metric spaces (but not all). On the other hand, note that the λ in Theorem 16.8 pertains *only* to the metric space $(S, dist)$, as opposed to $(U, dist)$. Hence, if the input set S is "easy", the theorem still yields a good structure, even though the underlying metric space is hard.

Let us also briefly discuss lower bounds. Remember that our goal is to design a *generic* data structure that treats objects and distance functions as black boxes. In that case, a simple adversary argument suffices to show that no structure can avoid calculating n distances in answering a query if the *exact* nearest neighbor is desired. For this purpose, simply define a set S of n objects where the distance between any two distinct objects is 4. Now, issue a query with an object $q \notin S$. Design the distances in such a way that $dist(q, e^*) = 1$ for exactly one object $e^* \in q$ while $dist(q, e) = 4$ for all other $e \in S \setminus \{e^*\}$. The design clearly satisfies the requirements of a metric space. The trick, however, is that the adversary decides which object in S is e^* by observing how the query algorithm $\mathcal{A}$ runs. Specifically, whenever $\mathcal{A}$ asks for the distance $dist(q, x)$ for some $x \in S$, the adversary answers 4. The only exception happens when x is the last object in S whose distance to q has not been calculated; in this case, the adversary answers $dist(q, x) = 1$, i.e., setting $e^* = x$. Therefore, $\mathcal{A}$ cannot terminate before all the n distances have been calculated (think: what could go wrong if $\mathcal{A}$ terminates, say, after computing $n - 1$ distances?).

The same argument also shows that n distances must be calculated even if our goal is to return a 3-ANN (think: why?). Krauthgamer and Lee [28] presented a stronger lower bound argument. They showed that if λ is the doubling dimension of the metric space $(S, dist)$, $2^{\Omega(\lambda)} \log |S|$ distances must be calculated to answer c-ANN queries with constant c.

Finally, it is worth mentioning that Krauthgamer and Lee [28] developed a more sophisticated structure that uses $O(n)$ space and answers any $(1 + \epsilon)$-ANN query in $2^{O(\lambda)} \log \Delta(S) + (1/\epsilon)^{O(\lambda)}$ time, where λ is the doubling dimension of $(S, dist)$ and $\epsilon > 0$ is an arbitrary real value.

Exercises

Problem 1*. Prove: in $\mathbb{R}^2$, any disc of radius 1 can be covered by 7 discs of radius 1/2.

(Hint: observe the intersection points made by the 7 + 1 = 8 circles in Figure 16.1.)

Problem 2. Finish the proof of Lemma 16.6.

(Hint: for $c < 2$, manually increase c to 2. To prove $c = 4$, apply the argument in the proof of Lemma 16.6 twice.)

Problem 3. Given an algorithm to find an r-sample net of S in $O(n^2)$ time where $n = |S|$.

Problem 4. Consider the metric space $(U, dist)$ where $dist(e, e') = 1$ for any distinct $e, e' \in U$. If U has a finite size, what is the doubling dimension of $(U, dist)$?

Problem 5*. Prove: the metric space $(\mathbb{N}^d, \text{Euclidean})$ has doubling dimension $O(d \log d)$.

(Hint: in 2D space, a disc of radius 1 is covered by a square of side length 2, and covers a square of side length $\sqrt{2}$. Extend this observation to $\mathbb{N}^d$.)

Problem 6. Let w be the word length. Let $\mathbb{N}_w$ be the set of integers from 0 to $2^w - 1$. Let P be a set of n points in $\mathbb{N}_w^d$ where d is a fixed constant. The value of n satisfies $w = \Theta(\log n)$. Describe a structure of $O(n \log n)$ space such that, given any point $q \in \mathbb{N}_w^d$, we are able to find a 3-ANN of q in P using $O(\log n)$ time. The distance metric is the Euclidean distance.

Problem 7*. Improve the structure of Theorem 16.8 to achieve $2^{O(\lambda)} \cdot O(n \cdot \log \Delta(S))$ space and $2^{O(\lambda)} \cdot O(\log \Delta(S))$ query time.

(Hint: $Y_i = S$ until i becomes sufficiently large).

Lecture 17: Approximate Nearest Neighbor Search 2: Locality Sensitive Hashing

This lecture continues our discussion on the c-approximate nearest neighbor (c-ANN) search problem. We will learn a technique called *locality sensitive hashing* (LSH). If $(U, dist)$ is a metric space with a constant doubling dimension λ, LSH usually performs worse than the structure of Theorem 16.8. However, the power of LSH is reflected in its ability to deal with "hard" metric spaces with large λ.

For example, consider the metric space $(U, dist) = (\mathbb{N}^d, \text{Euclidean})$, where the dimensionality d should *not* be regarded as a constant. The metric space has doubling dimension $\Theta(d)$. Theorem 16.8 yields a structure that calculates $2^{\min\{\log_2 n,\, \Omega(d)\}}$ distances, which is already n even for $d = \Omega(\log n)$! In fact, for a difficult problem like this, it is challenging even just to beat the naive query algorithm (which computes n distances) by a polynomial factor, while consuming a polynomial amount of space; e.g., $O((dn)^2)$ space and $O(dn^{0.99})$ query time would make a great structure. LSH allows us to achieve the purpose.

When the objects in U and $dist$ are treated as black boxes, we will measure the space and query time of a structure in a more careful manner compared to the last lecture:

- The space of a structure is expressed with two terms: (i) the number of memory cells occupied, and (ii) the number of objects stored.
- The query time is also expressed with two terms: (i) the number of atomic operations performed, and (ii) the number of distances calculated.

Notations and math preliminaries. We will reserve e, x for objects, and Z for sets of objects. Given a set $Z \subseteq U$, we denote by $diam(Z)$ the *diameter* of Z, defined in the same way as in Definition 16.1.

If Z_1, Z_2 are two sets of objects, their *multi-set* union is the collection of all the objects in Z_1 and Z_2, with duplicates retained.

If x is a point in $\mathbb{N}^d$, $x[i]$ denotes its coordinate on the i-th dimension ($i \in [1, d]$).

We will reserve X for random variables. If $X \geq 0$ is a real-valued random variable, we must have for any $t \geq 1$

$$\mathbf{Pr}\Big[X \geq t \cdot \mathbf{E}[X]\Big] \leq \frac{1}{t} \tag{17.1}$$

which is known as Markov's inequality.

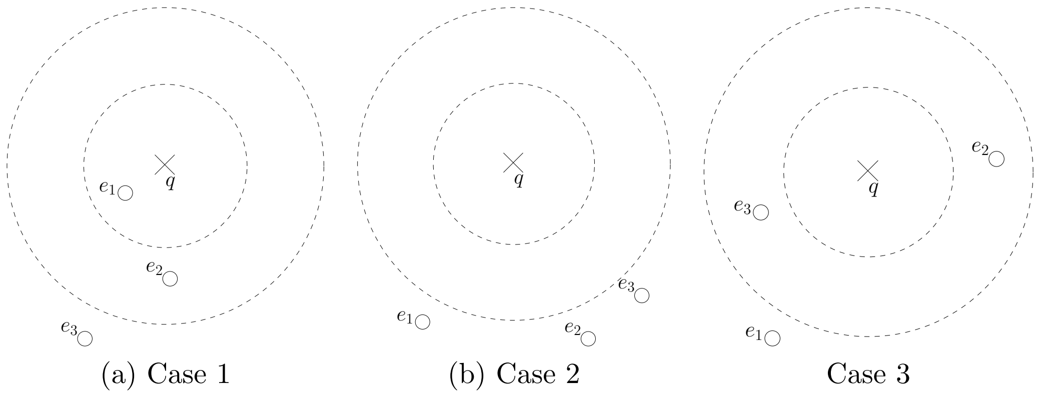


Figure 17.1: Illustrate of $(r, 2)$-near neighbor queries

17.1 (r, c)-near neighbor search

We will define a problem called (r, c)-*near neighbor search*, where $r \geq 1$ and $c > 1$ are real values. Let S be a set of n objects in U. Given an object $q \in U$, an (r, c)-*near neighbor query* — abbreviated as (r, c)-NN query — returns:

- **Case 1:** an object with distance at most cr to q, if S has an object with distance at most r to q;
- **Case 2:** nothing, if S has no object with distance at most cr to q;
- **Case 3:** either nothing or an object with distance at most cr to q, otherwise.

Example. Suppose that $U = \mathbb{N}^2$ and *dist* is the Euclidean distance. Figure 17.1(a) illustrates Case 1, where the inner and outer circles have radii r and $2r$, respectively. $S = \{e_1, e_2, e_3\}$. The cross point q indicates an $(r, 2)$-NN query. Since $dist(q, e_1) \leq r$, the query must return an object, but the object can be either e_1 or e_2. In Figure 17.1(b), however, the query *must not* return anything because all the objects in S have distances to q greater than $2r$ (Case 2). Figure 17.1(c) demonstrates Case 3, where the query may or may not return something; however, if it does, the object returned must be either e_2 or e_3. □

Lemma 17.1. *Suppose that, for any $r \geq 1$ and constant $c > 1$, we know how to build a structure on S that answers (r, c)-NN queries. By building $O(\log diam(S))$ such structures, we can answer any c^2-ANN query on S by issuing $O(\log diam(S))$ (r, c)-NN queries with the same c but different r.*

The proof is left to you as an exercise. In the rest of the lecture, we will focus on (r, c)-NN search.

17.2 Locality sensitive hashing

A *random function h* as a function that is drawn from a family H of functions according to a certain distribution.

Definition 17.2. *Consider a metric space $(U, dist)$. Let r, c, p_1, and p_2 be real values satisfying:*

- *$r \geq 1$, $c > 1$;*

- $0 < p_2 < p_1 \leq 1$.

*A random function $h : U \to \mathbb{N}$ is an (r, cr, p_1, p_2)-***locality sensitive hash function** *if:*

- *for any objects $x, y \in U$ satisfying $dist(x, y) \leq r$, it holds that $\mathbf{Pr}[h(x) = h(y)] \geq p_1$;*
- *for any objects $x, y \in U$ satisfying $dist(x, y) > cr$, it holds that $\mathbf{Pr}[h(x) = h(y)] \leq p_2$.*

We will abbreviate 'locality sensitive hash function" as "LSH function". Given a (r, cr, p_1, p_2)-LSH function h, we define

$$\rho = \frac{\ln(1/p_1)}{\ln(1/p_2)} \tag{17.2}$$

as the *log-ratio* of h. Note that $\rho < 1$.

Lemma 17.3. (The amplification lemma) *Suppose that we know how to obtain an (r, cr, p_1, p_2)-LSH function h. Then, for any integer $\ell \geq 1$, we can build an $(r, cr, p_1^\ell, p_2^\ell)$-LSH function g such that for any object x:*

- *$g(x)$ can be computed in cost $O(\ell)$ times higher than $h(x)$;*
- *$g(x)$ can be stored in $O(\ell)$ space.*

Proof. Take ℓ independent (r, cr, p_1, p_2)-LSH functions $h_1, h_2, ..., h_\ell$. Design $g(x)$ to be the string that concatenates $h_1(x), h_2(x), ..., h_\ell(x)$. For any objects x and y, $g(x) = g(y)$ if and only if $h_i(x) = h_i(y)$ for all $i \in [1, \ell]$. □

Example. We will describe how to obtain an (r, cr, p_1, p_2)-LSH function for $(\mathbb{N}^d, \text{Euclidean})$. First, generate d independent random variables $\alpha_1, \alpha_2, ..., \alpha_d$ each of which follows the normal distribution (i.e., mean 0 and variance 1). Let $\beta > 0$ be a real value that depends on c, and γ a real value generated uniformly at random in $[0, \beta]$. For any point $x \in \mathbb{N}^d$, define:

$$h(x) = \left\lfloor \frac{\gamma + \sum_{i=1}^{d}(\alpha_i \cdot x[i]/r)}{\beta} \right\rfloor. \tag{17.3}$$

Lemma 17.4 ([15]). *For any $r \geq 1$ and any constant $c > 0$, the function in* (17.3) *is an (r, cr, p_1, p_2)-LSH function satisfying:*

- *p_2 is a constant;*
- *the log-ratio ρ of the function is at most $1/c$.*

The proof is non-trivial and not required in this course. □

17.3 A structure for (r, c)-NN search

We will now describe a structure for answering (r, c)-NN queries on a set S of n objects in U, assuming the ability to build (r, cr, p_1, p_2)-LSH functions with a log-ratio ρ (see (17.2)). Denote by t_{lsh} the time needed to evaluate the value of an (r, cr, p_1, p_2)-LSH function (e.g., $t_{lsh} = O(d)$ for the function in (17.3)).

Our goal is to prove:

Theorem 17.5. *There is a structure using $O(n^{1+\rho} \cdot \log_{1/p_2} n)$ memory cells and storing $O(n^{1+\rho})$ objects that can answer one single (r, c)-NN query correctly with probability at least $1/10$. The query time is $O(n^{\rho} \cdot \log_{1/p_2} n \cdot t_{lsh})$, plus the cost of calculating $O(n^{\rho})$ distances.*

You may be disappointed: the structure can answer only *one* query with a low success probability. Don't be! Using standard techniques, we can improve the structure to support an arbitrary number of queries with high probability (e.g., $1 - 1/n^{100}$), by increasing the space and query time only by a logarithmic factor; you will explore this in an exercise.

17.3.1 Structure

Let $\ell \geq 1$ and $L \geq 1$ be integers to be determined later. Use Lemma 17.3 to obtain L independent $(r, cr, p_1^{\ell}, p_2^{\ell})$-LSH function $g_1, g_2, ..., g_L$. For each $i \in [1, L]$, define a *bucket* as a maximal set of objects $x \in S$ with the same $g_i(x)$. A *hash table* T_i collects all the non-empty buckets.

The hash tables $T_1, ..., T_L$ constitute our structure. The space consumption is $O(n \cdot L \cdot \ell)$ memory cells plus $O(n \cdot L)$ objects.

17.3.2 Query

Consider an (r, c)-NN query with search object q. For each $i \in [1, L]$, let b_i be the bucket of $g_i(q)$.

We take a collection Z of $2L + 1$ *arbitrary* objects from the *multi-set* union of $b_1, ..., b_L$. In the special case where $\sum_{i=1}^{L} |b_i| \leq 4L + 1$, Z collects all the objects in those buckets. We find the object e in Z closest to q, breaking ties arbitrarily. Return e if $dist(q, e) \leq cr$, or nothing, otherwise.

The query time is $O(t_{lsh} \cdot \ell \cdot L)$ atomic operations, plus the cost of computing $O(L)$ distances.

17.3.3 Analysis

We now choose the values of ℓ and L:

$$\ell = \log_{\frac{1}{p_2}} n \tag{17.4}$$
$$L = n^{\rho}. \tag{17.5}$$

Clearly, the space and query time of our structure match the claims in Theorem 17.5. We still need to prove that the query algorithm succeeds with probability at least 1/10. It suffices to consider that S contains an object e^* with $dist(q, e^*) \leq r$; otherwise, the algorithm is obviously correct (think: why?).

An object $x \in S$ is *good* if $dist(q, x) \leq cr$, or *bad* otherwise. Note that we succeed only if a good object is returned.

Lemma 17.6. *The query is answered correctly if the following two conditions hold:*

- **C1:** *e^* appears in at least one of $b_1, ..., b_L$;*
- **C2:** *there are at most $2L$ bad objects in the multi-set union of $b_1, ..., b_L$.*

Proof. If the multi-set union of $b_1, ..., b_L$ has a size at most $2L$, then **C1** ensures $e^* \in Z$. Otherwise, by **C2**, Z must contain at least a good object. □

Lemma 17.7. **C1** *fails with probability at most 1/e.*

Proof.

$$\begin{aligned}
\mathbf{Pr}\left[e^* \notin \bigcup_{i=1}^{L} b_i\right] &= \prod_{i=1}^{L} \mathbf{Pr}[e^* \notin b_i] \\
&= \prod_{i=1}^{L} \Big(1 - \mathbf{Pr}[g_i(e^*) = g_i(q)]\Big) \\
(g_i \text{ is an } (r, cr, p_1^\ell, p_2^\ell)\text{-LSH function}) \quad &\le \prod_{i=1}^{L} \Big(1 - p_1^\ell\Big) \\
&= \Big(1 - p_1^\ell\Big)^L. \qquad (17.6)
\end{aligned}$$

By (17.4), we know

$$\begin{aligned}
p_1^\ell &= p_1^{\log_{1/p_2} n} \\
&= \Big((1/p_2)^{\log_{1/p_2} p_1}\Big)^{\log_{1/p_2} n} \\
&= n^{\log_{1/p_2} p_1} \\
&= (1/n)^{\log_{1/p_2}(1/p_1)} \\
&= n^{-\rho}.
\end{aligned}$$

Therefore:

$$(17.6) = \Big(1 - n^{-\rho}\Big)^L \le \exp\big(-n^{-\rho} \cdot L\big) = 1/e$$

where the "$\le$" used the fact $(1+z) \le e^z$ for all $z \ge 0$, and the last equality used (17.5). □

Lemma 17.8. C2 *fails with probability at most 1/2.*

Proof. Let X be the total number of bad objects in the multi-set union of $b_1, ..., b_L$. Fix an arbitrary $i \in [1, L]$. Since g_i is an $(r, cr, p_1^\ell, p_2^\ell)$-LSH function, a bad object has probability at most $p_2^\ell = 1/n$ to fall in the same bucket as q. Hence, in expectation, there is at most 1 bad object in b_i. This means $\mathbf{E}[X] \le L$. By Markov's inequality (17.1), $Pr[X \ge 2L] \le 1/2$. □

Therefore, **C1** and **C2** hold simultaneously with probability at least $1 - (1/e + 1/2) > 0.1$. This completes the proof of Theorem 17.5.

17.4 Remarks

The LSH technique was proposed by Indyk and Motwani [25]. Today, effective LSH functions have been found for a large variety of spaces (U, *dist*), making the technique applicable to many distance functions. The function (17.3) is due to Datar, Immorlica, Indyk, and Mirrokni [15]. The function requires generating only $d+1$ real values: $\alpha_1, ..., \alpha_d$, and γ. This is not a problem in practice, but we must exercise care in theory. Consider, for example, γ, which is a real value in $[0, \beta]$. In the RAM model, we simply cannot generate γ because the only random atomic operation — RAND (Lecture 1) — has only finite precision. The same issue exists for $\alpha_1, ..., \alpha_d$ whose distributions are even more complex. To remedy the issue, we must carefully analyze the amount of precision required to attain a sufficiently accurate version of Lemma 17.4, which is rather difficult (and tedious). We will not delve into that in this course.

Exercises

Problem 1. Prove Lemma 17.1.

Problem 2. Prove the following stronger version of Theorem 17.5: there is a structure using $O(n^{1+\rho} \cdot \log_{1/p_2} n \cdot \log n)$ memory cells and storing $O(n^{1+\rho} \cdot \log n)$ objects that can answer one single (r, c)-NN query correctly with probability at least $1 - 1/n^{100}$. The query time is $O(n^{\rho} \cdot \log_{1/p_2} n \cdot t_{lsh} \cdot \log n)$, plus the cost of calculating $O(n^{\rho} \cdot \log n)$ distances.

(Hint: build $O(\log n)$ independent structures of Theorem 17.5; a query succeeds if it succeeds in any of those structures.)

Problem 3. Prove an even stronger statement: there is a structure using $O(n^{1+\rho} \cdot \log_{1/p_2} n \cdot \log n)$ memory cells and storing $O(n^{1+\rho} \cdot \log n)$ objects that, with probability at least $1 - 1/n^2$, can answer n^{98} (r, c)-NN queries correctly. The query time is $O(n^{\rho} \cdot \log_{1/p_2} n \cdot t_{lsh} \cdot \log n)$, plus the cost of calculating $O(n^{\rho} \cdot \log n)$ distances.

(Hint: if each query fails with probability at most $1/n^{100}$, the probability of answering all n^{98} queries correctly is at least $1 - 1/n^2$.)

Problem 4. Let w be the word length. Let $\mathbb{N}_w$ be the set of integers from 0 to $2^w - 1$. Let P be a set of n points in $\mathbb{N}_w^d$ where $d \geq 1$ should not be regarded as a constant. The value of n satisfies $w = \Theta(\log n)$. Given a point $q \in \mathbb{N}_w^d$, a *query* returns a 4-ANN of q in P. Describe a structure of $\tilde{O}(dn^{1.5})$ space that can answer one query in $\tilde{O}(d\sqrt{n})$ time with probability at least $1 - 1/n^{100}$. The distance metric is the Euclidean distance.

Problem 5 (LSH for the hamming distance). Consider $U = \{0, 1\}^d$ where $d \geq 1$ is an integer. Call each element in U a *string* (i.e., a bit sequence of length d). Given a string e, use $e[i]$ to denote its i-th bit, for $i \in [1, d]$. Given two strings e_1, e_2, $dist(e_1, e_2)$ equals the number of indexes at which e_1 and e_2 differ, or formally $|\{i \in [1, d] \mid e_1[i] \neq e_2[i]\}|$.

Design a function family H where each function maps a string $x \in \{0, 1\}^d$ to $\{0, 1\}$. Specifically, H has exactly d functions $h_1, ..., h_d$ where

$$h_i(x) \quad = \quad x[i].$$

A random function h is drawn uniformly at random from H. For any integers $r \geq 1$ and $c \geq 2$, prove: h is a $(r, cr, \frac{d-r}{d}, \frac{d-cr}{d})$-LSH function.

Lecture 18: Pattern Matching on Strings

In this lecture, we will discuss data structures on *strings*. Denote by Σ an *alphabet* which can be an arbitrarily large set (possibly infinite) where each element is called a *character*. A *string* σ is defined as a finite sequence of characters; denote by $|\sigma|$ the *length* of σ. Specially, define an empty sequence — denoted as $\emptyset$ — as a string of length 0. We will use $\sigma[i]$ $(1 \le i \le |\sigma|)$ to represent the i-th character of σ, and $\sigma[i:j]$ $(1 \le i \le j \le |\sigma|)$ to represent the *substring* of σ which concatenates $\sigma[i]$, $\sigma[i+1]$, ..., $\sigma[j]$. We will assume that each character in Σ can be stored in a cell.

Suppose that we are given a (long) string σ^* of length n. Given a string q, we say that a substring $\sigma^*[i:j]$ is an *occurrence* of q if

- $j - i + 1 = |q|$, and
- $q[x] = q[i + x - 1]$ for every $x \in [1, |q|]$.

A *pattern matching query* reports the starting positions of all the occurrences of q, namely, all $i \in [1, |\sigma|]$ such that $\sigma^*[i : i + |q| - 1]$ is an occurrence of q.

Example. Suppose that $\sigma^* =$ `aabcaabcabc`. Given $q =$ `abc`, the query should return 2, 6, and 9, whereas given $q =$ `aabca`, the query should return 1 and 5. $\square$

We want to store Σ in a data structure such that all pattern matching queries can be answered efficiently. We will refer to this as the *pattern matching problem*. Our goal is to prove:

Theorem 18.1. *There is a data structure that consumes $O(n)$ space, and answers any pattern matching query with a non-empty search string q in $O(|q| + occ)$ time, where occ is the number of occurrences of q.*

Both the space usage and the query time are optimal.

18.1 Prefix matching

Consider two strings q and σ with $|q| \le |\sigma|$. We say that q is a *prefix* of σ if $q = \sigma[1 : |q|]$. For example, `aabc` is a prefix of `aabcaab`. The empty string $\emptyset$ is a prefix of any string.

Our discussion will mainly concentrate on a different problem called *prefix matching*. Let S be a set of n distinct non-empty strings $\sigma_1, \sigma_2, .., \sigma_n$. The subscript $i \in [1, n]$ will be referred to as the *id* of σ_i. We are not responsible for *storing* S; to make this formal, we assume that there is an *oracle* which, given any $i \in [1, n]$ and any $j \in [1, |\sigma_i|]$, tells us the character $\sigma_i[j]$ in constant time. Given a query string q, a *prefix matching query* reports all the ids $i \in [1, n]$ such that q is a prefix of σ_i. We want to design a data structure such that any such query can be answered efficiently.

Example. Suppose that S consists of 11 strings as shown in Figure 18.1. Given $q =$ `abc`, the query should return 3, 6, and 10, whereas given $q =$ `aabca`, the query should return 7 and 11. $\square$

σ_1	`c`
σ_2	`bc`
σ_3	`abc`
σ_4	`cabc`
σ_5	`bcabc`
σ_6	`abcabc`
σ_7	`aabcabc`
σ_8	`caabcabc`
σ_9	`bcaabcabc`
σ_{10}	`abcaabcabc`
σ_{11}	`aabcaabcabc`

Figure 18.1: An input set S of strings for the prefix matching problem

We will prove:

Theorem 18.2. *There is a data structure that consumes $O(n)$ space, and answers any prefix matching query with a non-empty search string q in $O(|q| + k)$ time, where k is the number of ids reported.*

Sections 18.2 and 18.3 together serve as a proof of the above lemma.

18.2 Tries

Let us append a special character $\perp$ to each string in S; e.g., σ_2 in Figure 18.1 now becomes `bc`$\perp$. The distinctness of the (original) strings in S ensures that, with $\perp$ appended, now no string in S is a prefix of another.

In this section, we will introduce a simple structure — which is called the *trie* — that is able to achieve the query time in Theorem 18.2, but consumes more space than desired.

We define a *trie* on S as a tree T satisfying all the properties below:

- Every edge of T is labeled with a character in Σ.
- Concatenating the characters on any root-to-leaf path in Σ gives a string in S.
- There do not exist distinct nodes u, v in T such that, concatenating the characters on the root-to-u path gives the same string as concatenating the characters on the root-to-v path.

The second bullet implies that the number of leaf nodes of T is precisely n (i.e., the number of strings in Σ).

Example. Figure 18.2 shows a trie T on the set S of strings in Figure 18.1. The right most path of T, for example, corresponds to the string `cabc`$\perp$. $\square$

We answer a prefix-matching query q as follows. At the beginning, set $i = 0$ and u to the root of T. Iteratively, assuming $i < |q|$, we carry out the steps below:

1. Check whether u has a child v such that the edge (u, v) is labeled with $q[i]$.
2. If not, terminate the algorithm by returning nothing.
3. Otherwise, set u to v, and increment i by 1.
4. If $i < |q|$, repeat from Step 1.

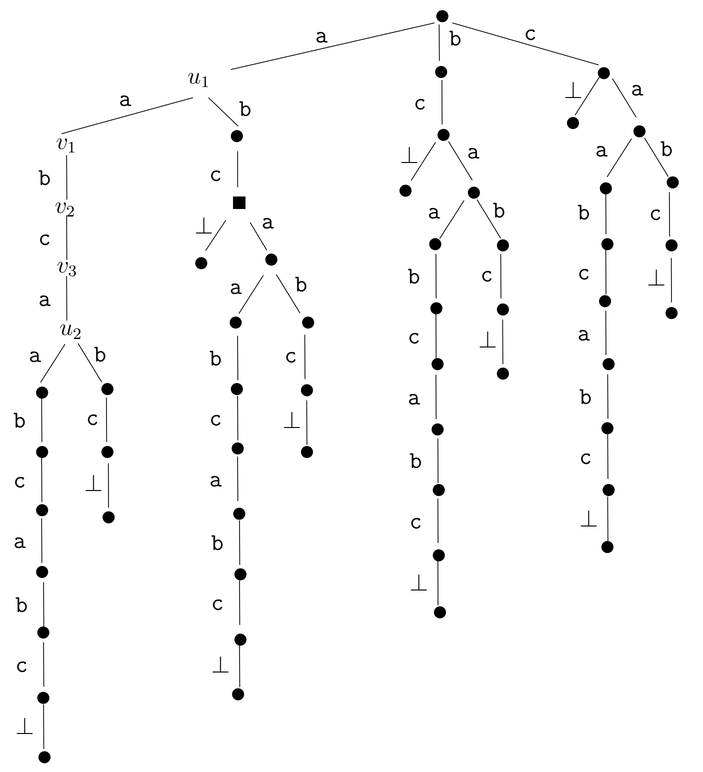


Figure 18.2: A trie

5. Otherwise, report the ids of strings corresponding to the leaves underneath u.

Example. Consider answering a query with $a =$ `abc` on the trie of Figure 18.2. The query algorithm descends to the node marked as a black square. There are three leaves under that node, corresponding to strings σ_3, σ_6, and σ_{10}, respectively. $\square$

The correctness of the algorithm is obvious. To implement the algorithm in $O(|q| + k)$ time, we need to (i) find v at Step 1 or declare its absence in $O(1)$ time, and (ii) report all the k leaves underneath the final u in $O(k)$ time. Achieving these purposes is easy and left to you as an exercise.

T, however, can have $\Omega(\sum_{i=1}^{n} |\sigma_i|)$ nodes, and thus, may consume more than $O(n)$ space. However, we have not utilized a crucial property stated in the prefix-matching problem: we are *not* responsible for storing S! In the next section, we will show how to leverage the property to reduce the space to $O(n)$ without affecting the query time.

18.3 Patricia Tries

A trie may have many nodes that have only one child (see Figure 18.2). Intuitively, such nodes waste space because they do not help to distinguish the strings in S. Our improved structure — called the *Patricia trie* — saves space by compressing such nodes.

Let the first define the *longest common prefix* (LCP) of S as the longest string that is a prefix of all the strings in S.

Example. For example, if S is the set of strings in Figure 18.1, then the LCP is $\emptyset$. On the other

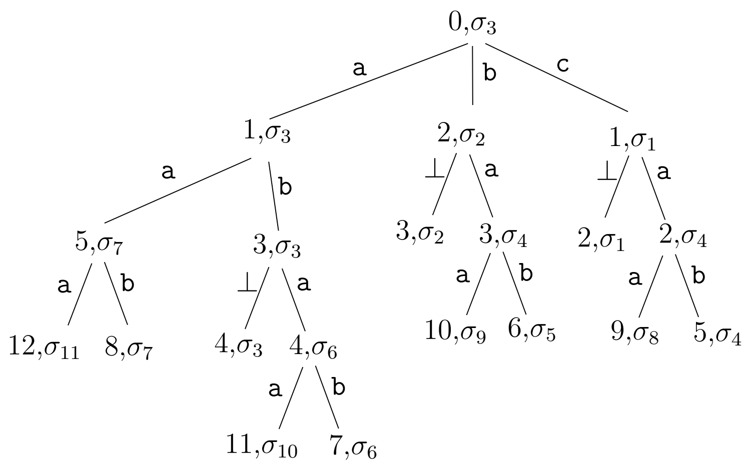


Figure 18.3: A Patricia trie

hand, if S consists of only σ_3, σ_6, and σ_{10}, then the LCP is `abc`. If we add also σ_{11} to S, then the LCP becomes `a`. $\square$

Given a string π, we define $S_\pi = \{\sigma \in S \mid \pi \text{ is a prefix of } \sigma\}$.

Example. Let S be the set of strings in Figure 18.1. $S_{\texttt{a}} = \{\sigma_3, \sigma_6, \sigma_7, \sigma_{10}, \sigma_{11}\}$ and $S_{\texttt{aabca}} = \{\sigma_7, \sigma_{11}\}$. $\square$

Now consider π to be the the LCP of S. Given a character $x \in \Sigma$, we denote by $\pi \circ x$ the string obtained by appending x to π. We call x an *extension character* of S if $|S_{\pi \circ x}| \geq 1$. Note that π being an LCP implies $S_{\pi \circ x}$ is a *proper* subset of S.

Proposition 18.3. *If $|S| \geq 2$, then S has at least two extension characters.*

The proof is easy and left to you.

Example. Let S be the set of strings in Figure 18.1. Its LCP is $\emptyset$. Characters `a`, `b`, and `c` are all extension characters. For $S_{\texttt{aa}} = \{\sigma_7, \sigma_{11}\}$, the LCP is `aabca`. Character `c` is not an extension character of $S_{\texttt{aa}}$ because $S_{\pi \circ \texttt{c}}$ is empty. The extension characters of $S_{\texttt{aa}}$ are `a` and `b`. $\square$

We are ready to define the Patricia trie T on a non-empty S recursively:

- If S has only a single string σ, T is a tree with only one node, labeled as $(|\sigma|, id(\sigma))$ where $id(\sigma)$ is the id of σ.
- Consider now $|S| \geq 2$. Let π be the LCP of S, and X be the set of extension characters of S. T is a tree where
  - the root is labeled as $(|\pi|, id(\sigma))$, where σ is an arbitrary string in S;
  - for every extension character $x \in X$, the root has a subtree which is the Patricia trie on $S_{\pi \circ x}$.

Example. Figure 18.3 shows the Patricia trie on the set S of strings in Figure 18.1. Recall that each string in Figure 18.1 has been appended with the special character $\perp$. $\square$

We leave the proof of the following lemma to you as an exercise:

Lemma 18.4. *The patricia trie on S has n leaves and at most $n-1$ internal nodes.*

As mentioned earlier, the Patricia trie is merely a compressed version of the trie. We illustrate this using an example:

Example. Compare the Patricia trie in Figure 18.3 to the trie in Figure 18.2. It is easy to see that nodes $(1, \sigma_3)$ and $(5, \sigma_7)$ in Figure 18.3 correspond to nodes u_1 and u_2 in Figure 18.2, respectively. As explained next, whenever needed, the entire path $u_1 \xrightarrow{\mathsf{a}} v_1 \xrightarrow{\mathsf{b}} v_2 \xrightarrow{\mathsf{c}} v_3 \xrightarrow{\mathsf{a}} u_2$ in Figure 18.2 can be reconstructed based on the integer 5 and string σ_7.

Denote by S' the set of strings corresponding to the leaves in the left subtree of node u_2 in Figure 18.2 ($S' = \{\sigma_7, \sigma_{11}\}$ but we do not need this in the following discussion). By how the Patricia trie was constructed, from $(5, \sigma_7)$ we know that S' must have an LCP π of length 5. As can be inferred from $(1, \sigma_3)$, for constructing the path $u_1 \xrightarrow{\mathsf{a}} v_1 \xrightarrow{\mathsf{b}} v_2 \xrightarrow{\mathsf{c}} v_3 \xrightarrow{\mathsf{a}} u_2$, it suffices to derive the last $5 - 1 = 4$ characters of π, i.e., $\pi[2], \pi[3], \pi[4]$, and $\pi[5]$. This is easy: $\pi[i]$ is simply $\sigma_7[i]$ for each $2 \leq i \leq 5$, and thus, can be obtained from the oracle in constant time. □

The proof for the next lemma is left as an exercise.

Lemma 18.5. *The patricia trie on S can be used to answer any prefix matching query with a non-empty search string q in $O(|q| + k)$ time, where k is the number of ids reported. .*

Theorem 18.2 thus follows from Lemmas 18.4 and 18.5.

18.4 The suffix tree

We now return to the pattern matching problem. Recall that the input is a string σ^* of length n. For each $i \in [1, n]$, define

$$\sigma_i = \sigma^*[i : n].$$

Note that σ_i is a suffix of σ^*. The next fact is immediate:

Proposition 18.6. *For any non-empty string q and any $i \in [1, n]$, $\sigma^*[i : i + |q| - 1]$ is an occurrence of q if and only if q is a prefix of σ_i.*

Create a structure of Theorem 18.2 on $S = \{\sigma_i \mid i \in [1, n]\}$. The structure — called the *suffix tree* on S — achieves the performance guarantees in Theorem 18.1. The proof is left to you as an exercise (think: what is the oracle?).

18.5 Remarks

The suffix tree is due to McCreight [31]. Farach [18] developed a (rather sophisticated) algorithm for constructing the tree in $O(n)$ time.

Exercises

Problem 1. Complete the query algorithm in Section 18.2 to achieve the time complexity of $O(|q| + k)$.

Problem 2. Complete the proof of Lemma 18.4.

(Hint: Proposition 18.3 implies that every internal node has at least two children.)

Problem 3. Complete the proof of Lemma 18.5.

Problem 4. Complete the proof of Theorem 18.1 in Section 18.4.

Problem 5. Let σ^* be a string of length n. Design a data structure of $O(n)$ space such that, given any non-empty string q, we can report the number of occurrences of q in σ^* in $O(|q|)$ time.

Problem 6*. Let S be a set of n strings $\sigma_1, \sigma_2, ..., \sigma_n$. Define $m = \sum_{i=1}^{n} |\sigma_i|$. Given a non-empty string q, an *occurrence* of q is defined by a pair (i, j) such that $\sigma_i[j : j + |q| - 1] = q$. A *general pattern matching query* reports all such pairs. Design a data structure of $O(m)$ space that can answer any query in $O(|q| + occ)$ time, where occ is the number of occurrences of q.

Bibliography

[1] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

[2] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

[3] P. Assouad. Plongements lipschitziens dans r^n. *Bull. Soc. Math. France*, 111(4):429–448, 1983.

[4] M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics (LATIN)*, volume 1776, pages 88–94.

[5] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM (CACM)*, 18(9):509–517, 1975.

[6] J. L. Bentley. Solutions to klee's rectangle problems. Technical report, Carnegie Mellon University, 1977.

[7] J. L. Bentley. Decomposable searching problems. *Information Processing Letters (IPL)*, 8(5):244–251, 1979.

[8] J. L. Bentley and J. B. Saxe. Decomposable searching problems I: Static-to-dynamic transformation. *Journal of Algorithms*, 1(4):301–358, 1980.

[9] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[10] T. M. Chan, K. G. Larsen, and M. Patrascu. Orthogonal range searching on the ram, revisited. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 1–10, 2011.

[11] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM Journal of Computing*, 15(3):703–724, 1986.

[12] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, 1988.

[13] B. Chazelle. Lower bounds for orthogonal range searching: I. the reporting case. *Journal of the ACM (JACM)*, 37(2):200–212, 1990.

[14] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, 2001.

[15] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of Symposium on Computational Geometry (SoCG)*, pages 253–262, 2004.

[16] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences (JCSS)*, 38(1):86–124, 1989.

[17] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. *Report F59, Inst. Informationsverarb., Tech. Univ. Graz*, 1980.

[18] M. Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 137–143, 1997.

[19] M. L. Fredman and M. E. Saks. The cell probe complexity of dynamic data structures. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

[20] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[21] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal of Computing*, 13(2):338–355, 1984.

[22] M. R. Henzinger and V. King. Randomized fully dynamic graph algorithms with polylogarithmic time per operation. *Journal of the ACM (JACM)*, 46(4):502–516, 1999.

[23] J. Holm, K. de Lichtenberg, and M. Thorup. Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *Journal of the ACM (JACM)*, 48(4):723–760, 2001.

[24] X. Hu, M. Qiao, and Y. Tao. Independent range sampling. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, pages 246–255, 2014.

[25] P. Indyk and R. Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 604–613, 1998.

[26] J. JaJa, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. pages 558–568, 2004.

[27] D. C. Kozen. *The Design and Analysis of Algorithms*. Springer New York, 1992.

[28] R. Krauthgamer and J. R. Lee. Navigating nets: simple algorithms for proximity search. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 798–807, 2004.

[29] D. T. Lee and C. K. Wong. Quintary trees: A file structure for multidimensional database systems. *ACM Transactions on Database Systems (TODS)*, 5(3):339–353, 1980.

[30] G. S. Lueker. A data structure for orthogonal range queries. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 28–34, 1978.

[31] E. M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM (JACM)*, 23(2):262–272, 1976.

[32] E. M. McCreight. Efficient algorithms for enumerating intersecting intervals and rectangles. *Report CSL-80-9, Xerox Palo Alto Res. Center*, 1980.

[33] E. M. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[34] J. Nievergelt and E. M. Reingold. Binary search trees of bounded balance. *SIAM Journal of Computing*, 2(1):33–43, 1973.

[35] M. H. Overmars. *The Design of Dynamic Data Structures.* Springer-Verlag, 1987.

[36] R. Pagh and F. F. Rodler. Cuckoo hashing. *J. Algorithms*, 51(2):122–144, 2004.

[37] M. Patrascu. Lower bounds for 2-dimensional range counting. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 40–46, 2007.

[38] M. Patrascu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 232–240, 2006.

[39] J. P. Schmidt and A. Siegel. The spatial complexity of oblivious k-probe hash functions. *SIAM Journal of Computing*, 19(5):775–786, 1990.

[40] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM (JACM)*, 22(2):215–225, 1975.

[41] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Information Processing Letters (IPL)*, 6(3):80–82, 1977.

[42] J. Vuillemin. A data structure for manipulating priority queues. *Communications of the ACM (CACM)*, 21(4):309–315, 1978.

[43] D. E. Willard. The super-b-tree algorithm. Technical report, Harvard University, 1979.

[44] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Processing Letters (IPL)*, 17(2):81–84, 1983.