(2,3)-Tree

Yufei Tao

ITEE
University of Queensland

We have learned that the binary search tree (BST) solves the dynamic predecessor search problem with good performance guarantees. In this class, we will learn another structure—called the (2,3)-tree—that settles the problem with the same asymptotic guarantees.

The (2,3)-tree, however, serves as a representative structure where all the data items are kept at the leaf nodes (recall that this is not true for the BST). Its update algorithm does not rely on rotations, but is instead based on splits and merges. This is an important technique for designing dynamic data structures.

Recall:

Dynamic Predecessor Search

Let S be a set of integers. We want to store S in a data structure to support the following operations:

- A predecessor query: give an integer q, find its predecessor in S, which is the largest integer in S that does not exceed q;
- Insertion: adds a new integer to S;
- Deletion: removes an integer from S.

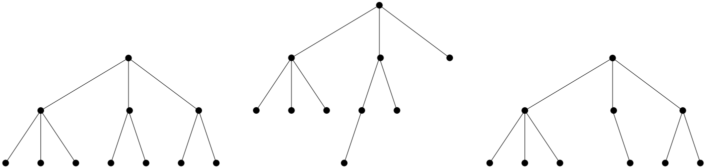
3-Ary Tree

Recall that a 3-ary tree T is a rooted tree where each internal node has at most 3 child nodes.

In this course, we say that T is a good 3-ary tree if all the following are true:

- All the leaves of T are at the same level.
- Every internal node has at least 2 child nodes.

Example



Only the first tree is a good 3-ary tree.

A Good 3-Ary Tree is Balanced

Theorem: If a good 3-ary tree has n leaf nodes, the height of the tree is $O(\log n)$.

Proof: Suppose that the height of the tree is h. Thus, all the leaf nodes are at level $h-1$. Since every internal node has at least 2 child nodes, the number of nodes at level $h-2$ is at most $n/2$. Similarly, the number of nodes at level $h-3$ is at most $n/2^2$. By the same reasoning, the number of nodes at level 0 is at most $n/2^{h-1}$. Therefore:

$$1 \le \frac{n}{2^{h-1}}$$

which solves to $h \le 1 + \log_2 n$. $\square$

(2,3)-Tree

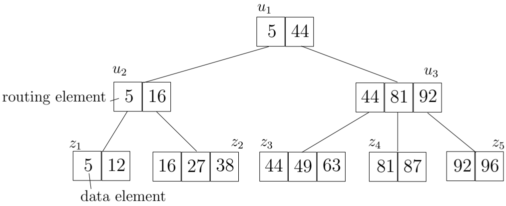
A (2,3)-tree on a set S of n integers is a good 3-ary tree T satisfying all the following conditions:

1. Every leaf node—if not the root—stores either 2 or 3 data elements, each of which is an integer in S.
2. Every integer in S is stored as a data element exactly once.
3. For every internal node u, if its child nodes are $v_1, ..., v_f$ ($f = 2$ or 3), then
   1. For any $i, j \in [1, f]$ such that $i < j$, all the data elements in the subtree of v_i are smaller than those in the subtree of v_j.
   2. For each $i \in [1, f]$, u stores the a routing element, which is an integer that equals the smallest data element in the subtree of v_i.

The space consumption is clearly $O(n)$.

Example

The following is a (2,3)-tree on $S = \{5, 12, 16, 27, 38, 44, 49, 63, 81, 87, 92, 96\}$.



The tree has 5 leaf nodes $z_1, z_2, ..., z_5$, and 3 internal ndoes u_1, u_2, u_3.

> Let v be a child of u. The routing element e corresponding to v can be obtained from v in $O(1)$ time (think: how?).

Predecessor Search

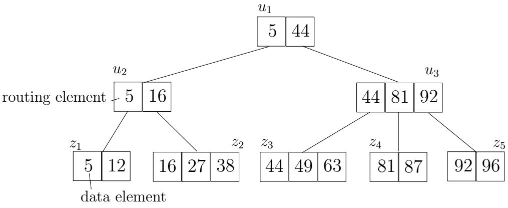
Consider a predecessor query with search value q. Without loss of generality, we assume that q has a predecessor in S—this can be easily ensured by manually inserting $-\infty$ into S.

We answer the query using using a (2,3)-tree T on S as follows:

1. Set $u \leftarrow$ the root of T
2. If u is a leaf, return the predecessor of q among the data elements in u.
3. Otherwise, let e be the predecessor of q among the routing elements in u.
4. Set u to the child node corresponding e.
5. Repeat from Line 2.

Example

Suppose that we want to find the predecessor of $q = 85$.



- At the root u_1, the predecessor of q (among the routing elements there) is 44. So we descend to u_3.
- At u_3, the predecessor of q is 81. So we descend to z_4.
- At z_4, report the predecessor of q among all the data elements there, namely, 81.

Time of a Predecessor Query

The (2,3)-tree has height $O(\log n)$, as proved earlier.

The query algorithm spends $O(1)$ time at each level of the tree. Therefore, the total query time is $O(\log n)$.

Next we will discuss how to support insertions and deletions in $O(\log n)$ time per update. We will first clarify two fundamental operations: split and merge. The update algorithms are based on these operations.

Split

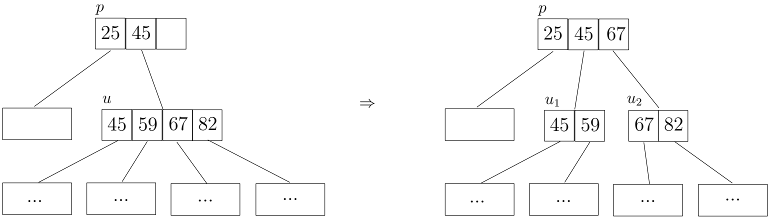
We say that an internal/leaf node u overflows if it contains 4 routing/data elements.

A split operation takes an overflowing node u, and does the following:

1. Create two nodes u_1, u_2 such that
   1.1 u_1 contains the two smaller routing/data elements of u.
   - Note: if a routing element e corresponds to a child v of u, assigning e to u_1 implies also making v a child of u_1, still with e being the routing element for v.

   1.2 u_2 contains the two larger routing/data elements of u.
2. Remove u from T.
3. If u had a parent p, then
   3.1 Make u_1 a child of p, in replacement of u.
   3.2 Make u_2 a child of p.
4. Otherwise, create a new root with u_1, u_2 as the child nodes.

Example



Node u overflows, and is split into u_1 and u_2.

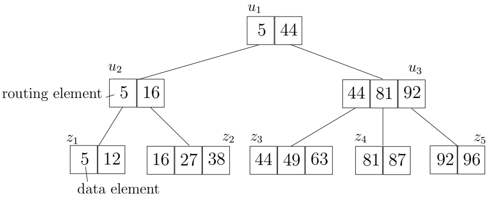
Each split takes $O(1)$ time.

Sibling

Suppose that an internal node u has child nodes $v_1, ..., v_f$ ($f = 2$ or 3), with routing elements $e_1, ..., e_f$ satisfying $e_1 < ... < e_f$. Then:

- v_i is the left sibling of v_{i+1} for every $i \in [1, f-1]$.
- v_{i+1} is the right sibling of v_i for every $i \in [1, f-1]$.

Example



Node z_4 is the right sibling of z_3, and the left sibling of z_5.

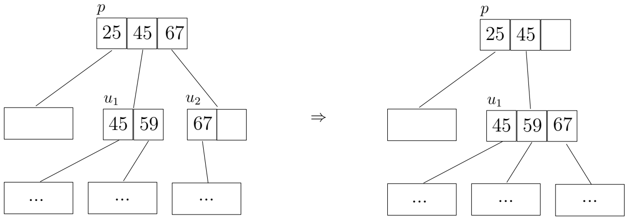
Merge

We say that a non-root internal/leaf node u underflows if it contains only 1 routing/data element.

A merge operation takes two nodes u_1, u_2 where (i) u_1 is the left sibling of u_2, and (ii) exactly one of them is underflowing. This operation does the following

1. Move all the routing/data elements of u_2 into u_1.
   - Note: if a routing element corresponds to a child of u_2, the child now becomes a child of u_1.
2. Remove u_2 from T.
3. Remove the routing element for u_2 in its parent p.
4. If u_1 overflows, split u_1.

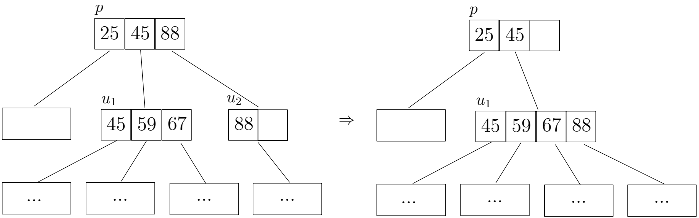
Merge Example 1



Node u_2 underflows, and is merged with its left sibling u_1.

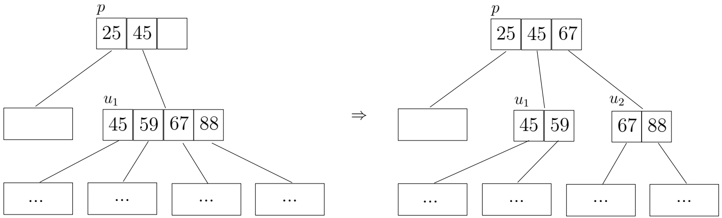
Each merge takes $O(1)$ time.

Merge Example 2



Node u_2 underflows, and is merged with its left sibling u_1. However, now u_1 overflows, and needs to be split. See the next slide.

Merge Example 2



The final situation after the split.

In general, a merge may trigger a split. Since we have shown that a split takes $O(1)$ time, the cost of treating an underflowing node is $O(1)$ overall in any case.

We are now ready to discuss the update algorithms, starting with insertions before attending to deletions. As we will see, these algorithms do not involve rotations, and may look simpler than those of the AVL-tree.

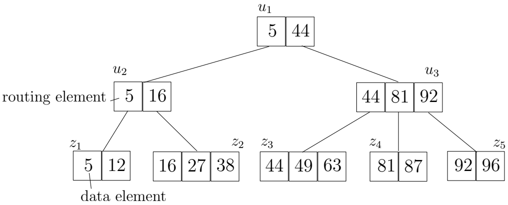
Insertion

To insert a new integer e into a (2,3)-tree T, we carry out the following steps:

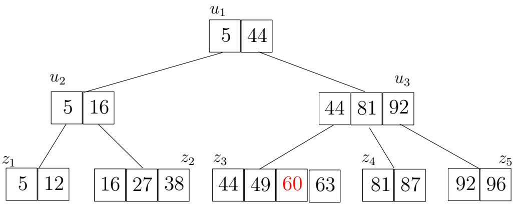
1. Perform a predecessor search with value e. Let z be the leaf node that the search ends up with. This is the leaf where e will be stored.
2. Add e as a new data element into z. Set $u \leftarrow z$.
3. If u does not overflow, return (the insertion is done).
4. Otherwise:
    - Split u.
    - Set u to its parent p, and repeat from Line 3.

Example

Suppose that we want to insert 60 into the following tree. It should go into Leaf z_3 (found be predecessor search).

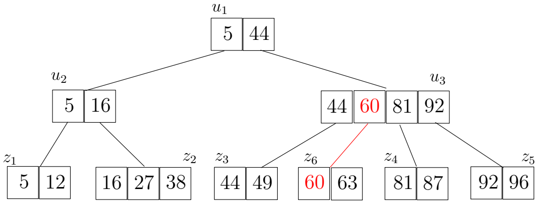


Now z_3 overflows, and needs to be split.

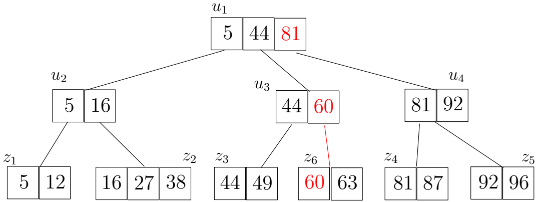


Example

Splitting z_6 makes its parent u_3 overflow, which also needs to be split.



Now the insertion completes.



Time of Insertion

The predecessor search obviously takes $O(\log n)$ time.

Then the insertion may trigger an overflow at each level. Since fixing an overflow with a split takes only $O(1)$ time, overall the insertion finishes in $O(\log n)$ time.

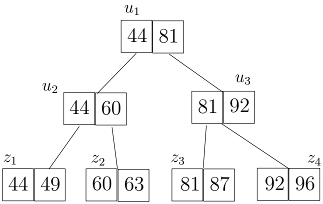
Deletion

To delete an integer e from a (2,3)-tree T, we carry out the following steps:

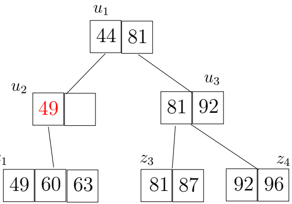
1. Find the leaf z containing e (with predecessor search).
2. Remove e from z. Set $u \leftarrow z$.
3. If u does not underflow, return (the deletion is done).
4. If u underflows and is the root of T, delete u from T (the height of T decreases by 1).
5. Otherwise:
   - Take either the left or right sibling u' of u.
   - Merge u with u'.
   - Set u to its parent p, and repeat from Line 3.

Example

Suppose that we want to delete 44 from the following tree. Remove it from Leaf z_1, which then underflows.

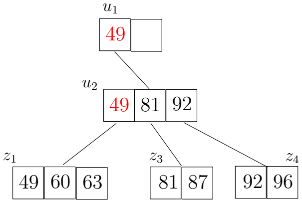


Merging z_1 with its right sibling z_2 causes their parent u_2 to underflow.

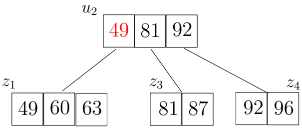


Example

Merging u_2 with its sibling u_3 causes u_1 to underflow.



But since u_1 is the root, we simply remove it from the tree (which now has only 2 levels). This is the end of the deletion.



Time of Deletion

The predecessor search takes $O(\log n)$ time.

Then the deletion may trigger an underflow at each level. Since fixing an underflow with a merge (possibly followed by a split) takes only $O(1)$ time, overall the deletion finishes in $O(\log n)$ time.

Summary

Now we know that a (2,3)-tree on a set of n integers has the following guarantees:

- Space consumption $O(n)$
- Predecessor query $O(\log n)$ (how to support in successor query also in $O(\log n)$ time?)
- Insertion $O(\log n)$ time
- Deletion $O(\log n)$ time.

So, we have learned two structures—the AVL-tree and the (2,3)-tree—for solving the dynamic predecessor search problem. Which one do you like better?

Regardless of your choice, pay attention to the differences in the methodology behind the two structures. Observe that both of them need to guarantee that the tree height is $O(\log n)$, but they have done so in different ways.