Lecture Notes: Weight-Balanced B-tree

Yufei Tao
Department of Computer Science and Engineering
Chinese University of Hong Kong
taoyf@cse.cuhk.edu.hk

In this lecture, we will study a technique called *weight-balancing*, which is very important in designing data structures, as we will see in later lectures. We will introduce the technique on the *B-tree*, which can be regarded as the EM equivalent of the binary search tree in RAM.

1 B-tree

Structure. Let S be a set of N elements in $\mathbb{R}$. A B-tree T on S is parameterized by two integer values: a *leaf parameter* $b \geq B$ and a *branching parameter* $p \geq 16$. We assume that both b and p are multiples of 16. Given a node u of T, we denote by $sub(u)$ the subtree of u. All the leaves of T are at the same level, namely, the length of each root-to-leaf path is the same. Each leaf node, if it is not the root, contains between $b/4$ and b elements in S—referred to as *leaf elements*. Each element of S is stored in one, and exactly one, leaf.

Consider now an internal node v with child nodes $u_1, u_2, ..., u_f$. We refer to the value of f as the *fanout* of v. If v is not the root, the value of f must satisfy $p/4 \leq f \leq p$; otherwise, it must hold that $f \geq 2$. For each u_i ($1 \leq i \leq f$), v stores a *routing element* e_i, which equals the smallest leaf element in $sub(u_i)$. Without loss of generality, suppose that $e_1, e_2, ..., e_f$ are in ascending order. For each $i \in [1, f-1]$, it must hold that *all* the leaf elements in $sub(u_i)$ be smaller than e_{i+1}.

T has $O(N/b)$ nodes in total, and therefore, occupies $O(N/b)$ space. We say that the leaves of T are at *level* 0, and inductively, the parent of a level-i node in T is at *level* $i+1$ ($i \geq 0$). The total number of levels is $O(\log_p(N/b))$.

We usually set $b = B$ and $p = B^c$ for some constant $c \in (0, 1]$. This ensures that the B-tree consumes $O(N/B)$ space, and has $O(\log_B N)$ levels. Such a B-tree can be harnessed to answer a large variety of queries efficiently. The following are two examples:

- *Predecessor search.* Given a value $q \in R$, a *predecessor query* returns the *predecessor* of q in S, namely, the largest element in S that is at most q. The predecessor can be found in $O(\log_B N)$ I/Os.
- *Range reporting.* Given an interval $I = [x, y]$, a *range query* reports all the elements in $S \cap I$. We can answer such a query in $O(\log_B N + K/B)$ I/Os, where $K = |S \cap I|$.

We leave to you to figure out the query algorithms.

Re-balancing Operations. The B-tree supports both insertions and deletions. Before we clarify the update algorithms, let us first elaborate on two re-balancing operations: split and merge.

Given a leaf/internal node u, we denote by $|u|$ the number of leaf/routing elements in u. We say that a leaf (or internal) u *overflows* if $|u| > b$ (or $|u| > p$, resp.). We will adhere to the constraint that an overflowing leaf (or internal) node u should always satisfy $|u| \leq 5b/4$ (or $|u| \leq 5p/4$, resp.). Denote by *parent*(u) the parent of u. A *split* of u is performed as follows:

- Create a new node u'. Move the $\lceil |u|/2 \rceil$ largest elements in u over to u' (note that if a routing element e is moved to u', then the child node of u that e corresponds to now becomes a child node of u'). Make u' a new child at *parent*(u) (this means that a routing element is added to *parent*(u) for u'). If *parent*(u) does not exist, create a new root with child nodes u and u'.

Let u be a non-root node. If u is a leaf (or internal) node, we say that u *underflows* if $|u| = b/4-1$ (or $|u| = p/4-1$, resp.). Let u' be a *neighboring sibling* of u, namely, no routing element in *parent*(u) is in between the two routing elements (in *parent*(u)) corresponding to u and u', respectively. Assuming that u' neither overflows nor underflows, a *merge* of u, u' is performed as follows:

- Move all the elements in u' into u (if u' is an internal node, this means that all the child nodes of u' are now child nodes of u). Remove u' from the tree, which reduces the fanout of *parent*(u) by 1. If *parent*(u) is the root and has only one child left (which must be u), make u the new root. If u is a leaf node and $|u| \geq 3b/4$, split u; similarly, if u is an internal node and $|u| \geq 3p/4$, split u.

We refer to splits and merges collectively as *rebalancing operations*. Each such operation can be carried out in $O((b+p)/B)$ I/Os at the leaf level, or $O(\lceil p/B \rceil)$ I/Os at the internal level.

Update. To insert an element e, descend a root-to-leaf path to the leaf node z that should accommodate e, and add e to z. The insertion finishes if z does not overflow. Otherwise, split z. The split may leave *parent*(z) overflowing; in this case, split *parent*(z), and handle the potential overflow in the parent of *parent*(z) in the same way.

To delete an element e, first descend a root-to-leaf path to the leaf node z where e resides, and then remove e from z. The deletion finishes if either z is the root, or z does not underflow. Otherwise, merge z with a neighboring sibling (if z has two neighboring siblings, the choice is arbitrary). We are done if either *parent*(z) is the root, or *parent*(z) does not underflow. Otherwise, merge *parent*(z) with a neighboring sibling, and handles the parent of *parent*(z) in the same way.

It is clear from the above discussion that, each insertion/deletion takes at most $O((b/B) + \lceil p/B \rceil \cdot \log_p(N/b))$ I/Os.

Remarks. Here are two interesting questions for you to think about:

- If we perform any mixture of N insertions and deletions, how many rebalancing operations can be triggered? The answer is $O(N/b)$, why?
- Consider a B-tree with $b = f = B$. Suppose that u and u' are two nodes at level-ℓ. What is the largest ratio between the numbers of leaf elements in their subtrees? For example, if $\ell = 1$, the answer is 4.

2 Weight-Balanced B-tree

Structure. Once again, let S be a set of N elements in $\mathbb{R}$. A *weight-balanced B-tree* [1] T on S is also parameterized by a leaf parameter $b \geq B$ and a branching parameter $p \geq 16$. We assume that b and p are multiples of 16. All the leaves of T are at the same level. Each leaf node, if not the root, contains between $b/4$ and b elements in S—referred to as *leaf elements*. Each element of S is stored in one, and exactly one, leaf.

Define the *weight* of u—denoted as $w(u)$—to be the number of leaf elements stored in *sub*(u) (i.e., the subtree of u). We say that the leaves of T are at *level* 0, and inductively, the parent of a level-i node in T is at *level* $i+1$ ($i \geq 0$). Let v be an internal node with child nodes $u_1, ..., u_f$. For each child node u_i ($1 \leq i \leq f$), v stores (i) a *routing element* e_i, which equals the smallest leaf element in *sub*(u_i), and (ii) the value of $w(u_i)$. Without loss of generality, suppose that $e_1, e_2, ..., e_f$

are in ascending order. For each $i \in [1, f-1]$, it must hold that *all* the leaf elements in $sub(u_i)$ be smaller than e_{i+1}.

The following *weight-balancing constraint* must hold for *every* non-root node u in T:

If u is at level ℓ, then its weight is between $p^\ell b/4$ and $p^\ell b$.

We complete the definition of T by requiring the root to have at least 2 child nodes.

You may be wondering: why haven't we imposed any constraints on the fanout of an internal node? In fact, we have done so implicitly via the weight-balancing constraint:

Lemma 1. *Each internal node has fanout between* $p/4$ *and* $4p$.

Proof. Consider an internal node v at level ℓ with child nodes $u_1, ..., u_f$. Clearly, $w(v) = \sum_{i=1}^{f} w(u_i)$. The lemma follows from the fact that $w(v) \in [p^\ell b/4, p^\ell b]$ whereas $w(u_i) \in [p^{\ell-1} b/4, p^{\ell-1} b]$ for each $i \in [1, f]$. □

As a result, T has consumes $O(N/b)$ space, and has height $O(\log_p(N/b))$. By setting $b = B$ and $p = B^c$ for some constant $c \in (0, 1]$, T answers predecessor and range queries with the same cost as a B-tree with the same b and p.

Remark. Let u, u' be two level-ℓ nodes of T. The weight-balancing constraint says that $w(u)$ and $w(u')$ differ by a factor of at most 4. In other words, the subtrees of u, u' contain roughly the same number of leaf elements. This is why T is said to be "weight-balanced".

Rebalancing Operations. We now re-design the split and merge operations for the weight-balanced B-tree. Given a non-root node u at level ℓ, we say that u *overflows* if $w(u) > p^\ell b$, or *underflows* if $w(u) = \frac{1}{4} p^\ell b - 1$. Given a level-$\ell$ overflowing node u with $w(u) \in [\frac{7}{8} p^\ell b, \frac{5}{4} p^\ell b]$, a *split* operation is performed as follows:

- *Case 1: u is a leaf node.* Create a new node u', and move half of the elements in u to u'. Update $parent(u)$ accordingly if u is not the root; otherwise, create a new root with child nodes u, u'. Note that the weights of u and u' are both in $[\frac{7}{16} b, \frac{5}{8} b]$.
- *Case 2: u is an internal node.* Suppose that u has child nodes $u_1, ..., u_f$. We find the maximum s satisfying

$$\sum_{i=1}^{s} w(u_i) \le \sum_{i=s+1}^{f} w(u_i). \tag{1}$$

Create a nodes u' and u''. Detach u from $parent(u)$, and $u_1, ..., u_f$ from u. Make $u_1, ..., u_s$ child nodes of u', and $u_{s+1}, ..., u_f$ child nodes of u''. Make u', u'' child nodes of $parent(u)$ if $parent(u)$ exists; otherwise, create a new node with u', u'' as the child nodes.

Next we analyze $w(u')$ and $w(u'')$. Clearly, $w(u') = \sum_{i=1}^{s} w(u_i)$ and $w(u'') = \sum_{i=s+1}^{f} w(u_i)$. Note that $w(u')$ and $w(u'')$ can differ by at most $2p^{\ell-1} b$ (otherwise, s could have increased by 1 without violating (1)). Therefore:

$$\begin{aligned} w(u') &\in \left[\frac{w(u)}{2} - p^{\ell-1} b,\ \frac{w(u)}{2}\right] \\ w(u'') &\in \left[\frac{w(u)}{2},\ \frac{w(u)}{2} + p^{\ell-1} b\right] \end{aligned}$$

With the fact that $w(u) \in [\frac{7}{8}p^i b, \frac{5}{4}p^i b]$ and that $p \geq 16$, it is easy to obtain:

$$w(u') \in \left[\frac{6}{16}p^\ell b,\ \frac{5}{8}p^\ell b\right]$$
$$w(u'') \in \left[\frac{7}{16}p^\ell b,\ \frac{11}{16}p^\ell b\right]$$

Next, we clarify *merge*. Given a level-ℓ underflowing node u, and an immediate sibling u' of u such that $w(u') \in [\frac{1}{4}p^\ell b, p^\ell b]$, this operation is performed as follows:

- *Merge.* Create a node $\bar{u}$. Detach all the child nodes of u, u' from their parents, and make all of them child nodes of $\bar{u}$. Detach u, u' from *parent*(u), and make $\bar{u}$ a child of *parent*(u). If *parent*(u) is the root and has only one child left, make $\bar{u}$ the new root. Note that at this moment $w(\bar{u})$ can be as large as $\frac{5}{4}p^\ell b - 1$. The merge finishes if $w(\bar{u}) \leq \frac{7}{8}p^\ell b$; otherwise, split $\bar{u}$.

Update. The description of the update algorithms in Section 1 applies verbatim here. Each update takes $O((b/B) + \lceil p/B \rceil \cdot \log_p(N/b))$ I/Os.

A Crucial Property of Weight Balancing. As we have seen, the WBB-tree has exactly the same space, update, and even query complexities (for predecessor and range queries) as the B-tree. So what have we gained? The answer is the following important lemma:

Lemma 2. *Let u be a node of a WBB-tree that is created by a split or a merge. Node u will not underflow or overflow unless $\Omega(w(u))$ leaf elements have been inserted or deleted in $sub(u)$.*

Proof. It follows from the above discussion that $w(u) \in [\frac{6}{16}p^\ell b, \frac{11}{16}p^\ell b]$. Hence, at least $\frac{2}{16}p^\ell b$ leaf elements must be deleted in $sub(u)$ for u to underflow, and at least $\frac{5}{16}p^\ell b$ leaf elements must be inserted in $sub(u)$ for u to overflow. □

The above property plays a crucial role in designing many data structures; we will see some examples in later lectures.

Remark. You may be wondering whether the B-tree in Section 1 also guarantees such a property. The answer is, as you could have guessed, no. Consider, for example, a B-tree of $b = p = B$. Suppose that a level-ℓ node u in the tree has just been produced by a split. Then, in the worst case, u will be split again after around $(B/2)^{\ell+1}$ insertions. On the other hand, a WBB-tree with $b = p = B$ can control this number to be $\Theta(B^{\ell+1})$.

References

[1] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *FOCS*, pages 560–569, 1996.