7.5 \((a, b)\)-trees

**Definition 1**
For \(b \geq 2a - 1\) an \((a, b)\)-tree is a search tree with the following properties

1. all leaves have the same distance to the root
2. every internal non-root vertex \(v\) has at least \(a\) and at most \(b\) children
3. the root has degree at least 2 if the tree is non-empty
4. the internal vertices do not contain data, but only keys (external search tree)
5. there is a special dummy leaf node with key-value \(\infty\)
7.5 \((a, b)\)-trees

Each internal node \(v\) with \(d(v)\) children stores \(d - 1\) keys \(k_1, \ldots, k_{d-1}\). The \(i\)-th subtree of \(v\) fulfills

\[
k_{i-1} < \text{key in } i\text{-th sub-tree} \leq k_i,
\]

where we use \(k_0 = -\infty\) and \(k_d = \infty\).
7.5 \((a, b)\)-trees

Example 2

\[
\begin{array}{cccc}
1 & 3 & 5 & \\
\downarrow & \downarrow & \downarrow & \\
1 & 3 & 5 & 10 & 14 & 19 & 28 & \infty
\end{array}
\]
Variants

▶ The dummy leaf element may not exist; it only makes implementation more convenient.
▶ Variants in which \( b = 2a \) are commonly referred to as \( B \)-trees.
▶ A \( B \)-tree usually refers to the variant in which keys and data are stored at internal nodes.
▶ A \( B^+ \) tree stores the data only at leaf nodes as in our definition. Sometimes the leaf nodes are also connected in a linear list data structure to speed up the computation of successors and predecessors.
▶ A \( B^* \) tree requires that a node is at least \( 2/3 \)-full as opposed to \( 1/2 \)-full (the requirement of a \( B \)-tree).
Lemma 3
Let $T$ be an $(a, b)$-tree for $n > 0$ elements (i.e., $n + 1$ leaf nodes) and height $h$ (number of edges from root to a leaf vertex). Then

1. $2a^{h-1} \leq n + 1 \leq b^h$
2. $\log_b(n + 1) \leq h \leq 1 + \log_a\left(\frac{n+1}{2}\right)$

Proof.

- If $n > 0$ the root has degree at least 2 and all other nodes have degree at least $a$. This gives that the number of leaf nodes is at least $2a^{h-1}$.
- Analogously, the degree of any node is at most $b$ and, hence, the number of leaf nodes at most $b^h$. 

\[ \square \]
The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $\mathcal{O}(b \cdot h) = \mathcal{O}(b \cdot \log n)$, if the individual nodes are organized as linear lists.
The search is straightforward. It is only important that you need to go all the way to the leaf.

Time: $O(b \cdot h) = O(b \cdot \log n)$, if the individual nodes are organized as linear lists.
Insert

Insert element \( x \):

- Follow the path as if searching for \( \text{key}[x] \).
- If this search ends in leaf \( \ell \), insert \( x \) before this leaf.
- For this add \( \text{key}[x] \) to the key-list of the last internal node \( v \) on the path.
- If after the insert \( v \) contains \( b \) nodes, do \( \text{Rebalance}(v) \).
Insert

Rebalance($v$):

- Let $k_i$, $i = 1, \ldots, b$ denote the keys stored in $v$.
- Let $j := \lfloor \frac{b+1}{2} \rfloor$ be the middle element.
- Create two nodes $v_1$ and $v_2$. $v_1$ gets all keys $k_1, \ldots, k_{j-1}$ and $v_2$ gets keys $k_{j+1}, \ldots, k_b$.
- Both nodes get at least $\lfloor \frac{b-1}{2} \rfloor$ keys, and have therefore degree at least $\lfloor \frac{b-1}{2} \rfloor + 1 \geq a$ since $b \geq 2a - 1$.
- They get at most $\lceil \frac{b-1}{2} \rceil$ keys, and have therefore degree at most $\lceil \frac{b-1}{2} \rceil + 1 \leq b$ (since $b \geq 2$).
- The key $k_j$ is promoted to the parent of $v$. The current pointer to $v$ is altered to point to $v_1$, and a new pointer (to the right of $k_j$) in the parent is added to point to $v_2$.
- Then, re-balance the parent.
Insert

Insert(7)
Insert

Insert(7)

7.5 $(a, b)$-trees
Insert

Insert(7)

7.5 \((a, b)\)-trees
Insert

Insert(7)

7.5 ($a$, $b$)-trees

Ernst Mayr, Harald Räcke
Delete element $x$ (pointer to leaf vertex):

- Let $v$ denote the parent of $x$. If $\text{key}[x]$ is contained in $v$, remove the key from $v$, and delete the leaf vertex.

- Otherwise delete the key of the predecessor of $x$ from $v$; delete the leaf vertex; and replace the occurrence of $\text{key}[x]$ in internal nodes by the predecessor key. (Note that it appears in exactly one internal vertex).

- If now the number of keys in $v$ is below $a - 1$ perform $\text{Rebalance}'(v)$. 

7.5 $(a, b)$-trees
Rebalance’(v):

- If there is a neighbour of \( v \) that has at least \( a \) keys take over the largest (if right neighbour) or smallest (if left neighbour) and the corresponding sub-tree.
- If not: merge \( v \) with one of its neighbours.
- The merged node contains at most \( (a - 2) + (a - 1) + 1 \) keys, and has therefore at most \( 2a - 1 \leq b \) successors.
- Then rebalance the parent.
- During this process the root may become empty. In this case the root is deleted and the height of the tree decreases.
Animation for deleting in an $(a, b)$-tree is only available in the lecture version of the slides.
(2, 4)-trees and red black trees

There is a close relation between red-black trees and (2, 4)-trees:

First make it into an internal search tree by moving the satellite-data from the leaves to internal nodes. Add dummy leaves.
There is a close relation between red-black trees and \((2, 4)\)-trees:

Then, color one key in each internal node \(v\) black. If \(v\) contains 3 keys you need to select the middle key otherwise choose a black key arbitrarily. The other keys are colored red.
There is a close relation between red-black trees and (2,4)-trees:

Re-attach the pointers to individual keys. A pointer that is between two keys is attached as a child of the red key. The incoming pointer, points to the black key.
(2, 4)-trees and red black trees

There is a close relation between red-black trees and (2, 4)-trees:

Note that this correspondence is not unique. In particular, there are different red-black trees that correspond to the same (2, 4)-tree.
Augmenting Data Structures

Bibliography


A description of B-trees (a specific variant of $(a, b)$-trees) can be found in Chapter 18 of [CLRS90]. Chapter 7.2 of [MS08] discusses $(a, b)$-trees as discussed in the lecture.