3 (a,b)-trees

In this chapter, we will study an extension of binary search trees. It will store multiple keys per node, so the nodes will have more than two children. The structure of such trees will be more complex, but we will gain much more straightforward balancing operations.

3.1 Definition and operations

Definition: A *multi-way search tree* is a rooted tree with specified order of children in every node. Nodes are divided to internal and external.

Each *internal node* contains one or more distinct keys, stored in increasing order. A node with keys $x_1 < \ldots < x_k$ has $k+1$ children $s_0, \ldots, s_k$. The keys in the node separate keys in the corresponding subtrees. More formally, if we extend the keys by sentinels $x_0 = -\infty$ and $x_{k+1} = +\infty$, every key y in the subtree $T(s_i)$ satisfies $x_i < y < x_{i+1}$.

External nodes carry no data and they have no children. These are the leaves of the tree. In a program, we can represent them by null pointers.

Observation: Searching in multi-way trees is similar to binary trees. We start at the root. In each node, we compare the desired key with all keys of the node. Then we either finish or continue in a uniquely determined subtree. When we reach an external node, we conclude that the requested key is not present.

The universe is split to open intervals by the keys of the tree. There is a 1-to-1 correspondence between these intervals and external nodes of the tree. This means that searches for all keys from the same interval end in the same extrnal node.

As in binary search trees, multiway-trees can become degenerate. We therefore need to add further invariants to keep the trees balanced.

Definition: An *(a,b)-tree* for parameters $a \geq 2$ and $b \geq 2a - 1$ is a multi-way search tree, which satisfies:

1. The root has between 2 and b children (unless it is a leaf, which happens when the set of keys is empty). Every other internal node has between a and b children.
2. All external nodes have the same depth.

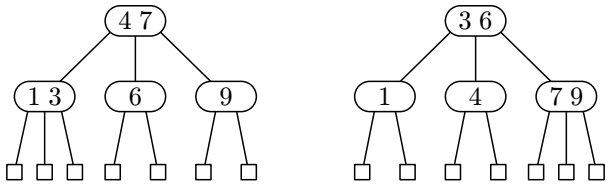

Figure 3.1: Two $(2,3)$-trees for the same set of keys

The requirements on a and b may seem mysterious, but they will become clear when we define operations. The minimum working type of (a,b)-trees are $(2,3)$-trees — see figure 3.1 for examples. Internal nodes are drawn round, external nodes are square.

Unlike in other texts, we will *not* assume that a and b are constants, which can be hidden in the $\mathcal{O}$. This will help us to examine how the performance of the structure depends on these parameters.

We will start with bounding the height of (a,b)-trees. Height will be measured in edges, a tree of height h has the root at depth 0 and external nodes at depth h. We will call the set of nodes at depth i the i-th *level.*

Lemma: The height of an (a,b)-tree with n keys lies between $\log_b(n+1)$ and $1+\log_a((n+1)/2)$.

Proof: Let us start with the upper bound. We will calculate the minimum number of keys in a tree of height h. The minimum will be attained by a tree where each node contains the minimum possible number of keys, so it has the minimum possible number of children. Level 0 contains only the root with 1 key. Level h contains only external nodes. The i-th level inbetween contains $2a^{i-1}$ nodes with $a-1$ keys each. Summing over levels, we get:

$$1+\sum_{i=1}^{h-1} 2a^{i-1}(a-1) = 1+2(a-1)\sum_{i=0}^{h-2} a^i = 1+2(a-1)\frac{(a^{h-1}-1)}{(a-1)} = 2a^{h-1}-1.$$

So $n \geq 2a^{h-1}-1$ in any tree of height h. Solving this for h yields $h \leq 1+\log_a((n+1)/2)$.

For the lower bound, we consider the maximum number of keys for height h. All nodes will contain the highest possible number of keys. Thus we will have b^i nodes at level i,

each with $b-1$ keys. In total, the number of keys will reach

$$\sum_{i=0}^{h-1} b^i(b-1) = (b-1)\sum_{i=0}^{h-1} b^i = (b-1)\cdot\frac{b^h-1}{b-1} = b^h - 1.$$

Therefore in each tree, we have $n \leq b^h - 1$, so $h \geq \log_b(n+1)$. □

Corollary: The height is $\Omega(\log_b n)$ and $\mathcal{O}(\log_a n)$.

Searching for a key

FIND(x) follows the general algorithm for multi-way trees. It visits $\mathcal{O}(\log_a n)$ nodes, which is $\mathcal{O}(\log n/\log a)$. In each node, it compares x with all keys of the node, which can be performed in time $\mathcal{O}(\log b)$ by binary search. In total, we spend time $\Theta(\log n \cdot \log b/\log a)$.

If b is polynomial in a, the ratio of logarithms is $\Theta(1)$, so the complexity of FIND is $\Theta(\log n)$. This is optimum since each non-final comparison brings at most 1 bit of information and we need to gather $\log n$ bits to determine the result.

Insertion

If we want to insert a key, we try to find it first. If the key is not present yet, the search ends in a leaf (external node). However, we cannot simply turn this leaf into an internal node with two external children — this would break the axiom that all leaves lie on the same level.

Instead, we insert the key to the parent of the external node — that is, to a node on the lowest internal level. Adding a key requires adding a child, so we add a leaf. This is correct since all other children of that node are also leaves.

If the node still has at most $b-1$ keys, we are done. Otherwise, we split the overfull node to two and distribute the keys approximately equally. In the parent of the split node, we need to replace one child pointer by two, so we have to add a key to the parent. We solve this by moving the middle key of the overfull node to the parent. Therefore we are splitting the overfull node to three parts: the middle key is moved to the parent, all smaller keys form one new node, and all larger keys form the other one. Children will be distributed among the new nodes in the only possible way.

This way, we have reduced insertion of a key to the current node to insertion of a key to its parent. Again, this can lead to the parent overflowing, so the splitting can continue, possibly up to the root. If it happens that we split the root, we create a new root with a single key and two children (this is correct, since we allowed less than a children in the root). This increases the height of the tree by 1.

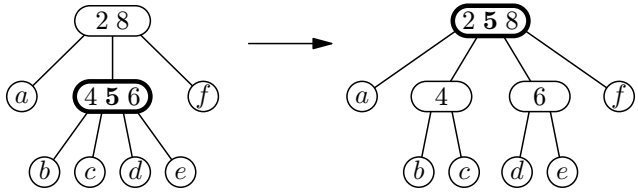

Figure 3.2: Splitting an overfull node on insertion to a $(2, 3)$-tree

Let us calculate time complexity of INSERT. In the worst case, we visit $\Theta(1)$ nodes on each level and we spend $\Theta(b)$ time on each node. This makes $\Theta(b \cdot \log n / \log a)$ time total.

It remains to show that nodes created by splitting are not undersized, meaning they have at least a children. We split a node v when it reached $b + 1$ children, so it had b keys. We send one key to the parent, so the new nodes v_1 and v_2 will take $\lfloor (b-1)/2 \rfloor$ and $\lceil (b-1)/2 \rceil$ keys. If any of them were undersized, we would have $(b-1)/2 < a-1$ and thus $b < 2a-1$. Voilà, this explains why we put the condition $b \geq 2a-1$ in the definition.

Deletion

If we want to delete a key, we find it first. If it is located on the last internal level, we can delete it directly, together with one of the leaves under it. We still have to check for underflow, though.

Keys located on the higher levels cannot be removed directly — the internal node would lose one pointer and we would have a subtree left in our hands with no place to connect it to. This situation is similar to deletion of a node with two children in a binary search tree, so we can solve it similarly: We replace the deleted key by its successor (which is the leftmost key in the deleted key's right subtree). The successor lies on the last internal level, so it can be deleted directly.

The only remaining problem is fixing an undersized node. For a moment we will assume that the node is not the root, so it has $a-2$ keys. It is tempting to solve the underflow by merging the node with one of its siblings. However, this can be done only if the sibling contains few keys; otherwise, the merged node could be overfull. But if the sibling is large, we can fix our problem by borrowing a key from it.

Let us be exact. Suppose that we have an undersized node v with $a-2$ keys and this node has a left sibling ℓ separated by a key p in their common parent. If there is no left sibling, we use the right sibling and follow a mirror image of the procedure.

If the sibling has only a children, we merge nodes v and ℓ to a single node and we also move the key p from the parent there. This creates a node with $(a-2)+(a-1)+1 = 2a-2$

keys, which cannot exceed $b-1$. Therefore we reduced deletion from v to deletion from its parent.

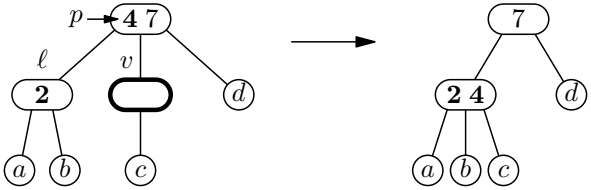

Figure 3.3: Merging nodes on deletion from a $(2,3)$-tree

On the contrary, if the sibling has more than a children, we disconnect its rightmost child c and its largest key m. Then we move the key m to the parent and the key p from the parent to v. There, p becomes the smallest key, before which we connect the child c. After this ''rotation of keys'', all nodes will have numbers of children in the allowed range, so we can stop.

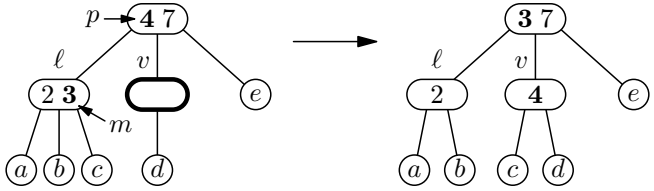

Figure 3.4: Borrowing a key from a sibling on deletion from a $(2,3)$-tree

In each step of the algorithm, we either produce a node which is not undersized, or we fix the underflow by borrowing from a sibling, or we merge two nodes. In the first two cases, we stop. In the third case, we continue by deleting a key on the next higher level, continuing possibly to the root. If the root becomes undersized, it has no keys left, in which case we make its only child the new root.

In the worst case, DELETE visits $\Theta(1)$ nodes on each level and it spends $\Theta(b)$ time per node. Its time complexity is therefore $\Theta(b \cdot \log n / \log a)$.

The choice of parameters

For a fixed a, the complexity of all three operations increases with b, so we should set b small. The usual choices are the minimum value $2a-1$ allowed by the definition or $2a$. As we will see in the following sections, the latter value has several advantages.

If $b \in \Theta(a)$, the complexity of FIND becomes $\Theta(\log n)$. Both INSERT and DELETE will run in time $\Theta(\log n \cdot (a/\log a))$. We can therefore conclude that we want to set a to 2 or possibly to another small constant. The best choices are the $(2,3)$-tree and the $(2,4)$-tree.

This is true on the RAM, or on any machine with ideal random-access memory. If we store the tree on a disk, the situation changes. A disk is divided to blocks (the typical size of a block is on the order of kilobytes) and I/O is performed on whole blocks. Reading a single byte is therefore as expensive as reading the full block. In this situation, we will set a to make the size of a node match the size of the block. Consider an example with 4 KB blocks, 32-bit keys, and 32-bit pointers. If we use the $(256, 511)$-tree, one node will fit in a block and the tree will be very shallow: four levels suffice for storing more than 33 million keys. Furthermore, the last level contains only external nodes and we can keep the root cached in memory, so each search will read only 2 blocks.

The same principle actually applies to main memory of contemporary computers, too. They usually employ a fast *cache* between the processor and the (much slower) main memory. The communication between the cache and the memory involves transfer of *cache lines* — blocks of typical size around 64 B. It therefore helps if we match the size of nodes with the size of cache lines and we align the start of nodes to a multiple of cache line size. For 32-bit keys and 32-bit pointers, we can use a $(4,7)$-tree.

Other versions

Our definition of (a,b)-trees is not the only one used. Some authors prefer to store useful keys only on the last level and let the other internal nodes contain copies of these keys (typically minima of subtrees) for easy navigation. This requires minor modifications to our procedures, but the asymptotics stay the same. This version can be useful in databases, which usually associate potentially large data with each key. They have two different formats of nodes, possibly with different choices of a and b: leaves, which contain keys with associated data, and internal nodes with keys and pointers.

Database theorists often prefer the name *B-trees.* There are many definitions of B-trees in the wild, but they are usually equivalent to $(a, 2a-1)$-trees or $(a, 2a)$-trees, possibly modified as in the previous paragraph.

3.2 Amortized analysis

In an amortized sense, (a,b)-trees cannot perform better than in $\Omega(\log n)$, because all operations involve search, which can take $\mathcal{O}(\log n)$ repeatedly. However, if we already know the location of the key in the tree, the rest of the operation can amortize well. In particular, the amortized number of modified nodes is constant for some choices of a and b.

Theorem: A sequence of m INSERTs on an initially empty (a, b)-tree performs $\mathcal{O}(m)$ node modifications.

Proof: Each INSERT performs a (possibly empty) sequence of node splits and then it modifies one node. A split modifies two nodes: one existing and one newly created. Since each split creates a new node and the number of nodes never decreases, the total number of splits is bounded by the number of nodes at the end of the sequence, which cannot exceed m.

So we have $\mathcal{O}(m)$ splits, each modifying $\mathcal{O}(1)$ nodes, and $\mathcal{O}(m)$ modifications outside splits. □

When we start mixing insertions with deletions, the situation becomes more complicated. In a $(a, 2a-1)$-tree, it might happen that INSERT(x) forces splitting up to the root and the immediately following DELETE(x) undoes all the work and reverts the tree back to the original state. So we can construct an arbitrarily long sequence of operations which have $\Omega(\log n)$ cost each.

We already encountered this problem with the flexible arrays of section **??** — if the thresholds for stretching and shrinking are too close, the structure tends to oscillate expensively between two states. We cured this by allowing the structure extra ''breathing space''.

The same applies to (a, b)-trees. When we increase b to at least $2a$, it is sufficient to avoid oscillations and keep the amortized number of changes constant. For simplicity, we will prove this for the specific case $b = 2a$.

Theorem: A sequence of m INSERTs and DELETEs on an initially empty $(a, 2a)$-tree performs $\mathcal{O}(m)$ node modifications.

Proof: We define the cost of an operation as the number of nodes it modifies. We will show that there exists a potential Φ such that the amortized cost of splitting and merging with respect to Φ is zero or negative and the amortized cost of the rest of INSERT and DELETE is constant.

The potential will be a sum of node contributions. Every node with k keys contributes $f(k)$ to the potential, where f will be a certain non-negative function. We allow k from $a-2$ to $b = 2a$ to handle nodes, which are temporarily overflowed or underflowed. By definition, Φ is initially zero and always non-negative.

The function f should satisfy the following requirements:

1. $|f(i) - f(i+1)| \leq c$ for some constant c.

This means that if the number of keys in the node changes by 1, the node's contribution changes at most by a constant.

2. $f(2a) \geq f(a) + f(a-1) + c + 1$.

   This guarantees free splits: If we split a node with $2a$ keys, we release $f(2a)$ units of potential. We have to spend $f(a)$ and $f(a-1)$ units on the two new nodes and at most c on inserting a key to the parent node. Still, we have one unit to pay for the real cost. (Strictly speaking, the real cost is 3, but we can make it 1 by scaling the costs and the potential.)

3. $f(a-2) + f(a-1) \geq f(2a-2) + c + 1$.

   This guarantees free merges: We are always merging an underflowing node ($a-2$ keys) with a minimum allowed node ($a-1$ keys). The potential released by removing these nodes is spent on creation of the merged node with $2a-2$ keys, removing a key from the parent node and paying for the real cost of the operation.

By little experimentation, we can construct a function f meeting these criteria with $c = 2$:

k	$a-2$	$a-1$	a	$\ldots$	$2a-2$	$2a-1$	$2a$
$f(k)$	2	1	0	$\ldots$	0	2	4

The function is zero between a and $2a-2$.

An Insert begins by adding the new key, which has $\mathcal{O}(1)$ real cost and it changes the potential by $\mathcal{O}(1)$. Then it performs a sequence of splits, each with zero amortized cost.

A Delete removes the key, which has $\mathcal{O}(1)$ cost both real and amortized. If the node was undersized, it can perform a sequence of merges with zero amortized cost. Finally, it can borrow a key from a neighbor, which has $\mathcal{O}(1)$ real and amortized cost, but this happens at most once per Delete.

We conclude that the amortized cost of both operations is constant. As the potential is non-negative and initially zero, this guarantees that the total real cost of all operations is $\mathcal{O}(m)$. □

3.3 Top-down (a,b)-trees

Our implementation of Insert and Delete first traverses the tree from the root downwards, looking for the key. Then it performs the operation locally and traverses the

same path upwards, fixing violations of invariants. We will show that if $b \geq 2a$, a single top-down traversal suffices.

INSERT will employ *pre-emptive splitting* on the way down. It will maintain the invariant that the current node contains less than $b - 1$ keys (that is, the node is not completely full). Whenever it encounters a node with $b - 1$ keys, it splits the node. This requires adding a key to the parent, but we can be sure there is enough space for the key there. When we reach the bottommost level, we simply insert the new key there.

DELETE is similar — we make sure that the current node has more keys than the minimum $a - 1$. We accompish this by either borrowing from a sibling or by merging with a sibling. The latter requires pulling one key from the parent, which is always possible.

In chapter **??**, we will use this approach to build a search tree with parallel access.