Lecture III
BALANCED SEARCH TREES

Anthropologists inform that[1] there is an unusually large number of Eskimo words for snow. The Computer Science equivalent of snow is the tree word: we have *(a,b)-tree, AVL tree, B-tree, binary search tree, BSP tree, conjugation tree, dynamic weighted tree, finger tree, half-balanced tree, heaps, interval tree, kd-tree, quadtree, octtree, optimal binary search tree, priority search tree, R-trees, randomized search tree, range tree, red-black tree, segment tree, splay tree, suffix tree, treaps, tries, weight-balanced tree, etc.* The above list is restricted to trees used as search data structures. If we include trees arising in specific applications (e.g., Huffman tree, DFS/BFS tree, alpha-beta tree), we obtain an even more diverse list. The list can be enlarged to include variants of these trees: thus there are subspecies of B-trees called B^+- and B^*-trees, etc.

The simplest search tree is the binary search tree. It is usually the first non-trivial data structure that students encounter, after linear structures such as arrays, lists, stacks and queues. Trees are useful for implementing a variety of **abstract data types**. We shall see that all the common operations for search structures are easily implemented using binary search trees. Algorithms on binary search trees have a worst-case behaviour that is proportional to the height of the tree. The height of a binary tree on n nodes is at least $\lfloor \lg n \rfloor$. We say that a family of binary trees is **balanced** if every tree in the family on n nodes has height $O(\log n)$. The implicit constant in the big-Oh notation here depends on the particular family of search trees. Such a family usually comes equipped with algorithms for inserting and deleting items from trees, while preserving membership in the family.

Many balanced families have been invented in computer science. They come in two basic forms: **height-balanced** and **weight-balanced schemes**. In the former, we ensure that the height of siblings are "approximately the same". In the latter, we ensure that the number of descendents of sibling nodes are "approximately the same". Height-balanced schemes require us to maintain less information than the weight-balanced schemes, but the latter has some extra flexibility that are needed for some applications. The first balanced family of trees was invented by the Russians Adel'son-Vel'skii and Landis in 1962, and are called **AVL trees**. We will describe several balanced families, including AVL trees and red-black trees. The notion of balance can be applied to non-binary trees; we will study the family of (a,b)**-trees** and generalizations. Tarjan [8] gives a brief history of some balancing schemes.

STUDY GUIDE: all algorithms for search trees are described in such a way that they can be internalized, and we expect students to carry out hand-simulations on concrete examples. We do not provide any computer code, but once these algorithms are understood, it should be possible to implementing them in your favorite programming language.

§1. Search Structures with Keys

Search structures store a set of objects subject to searching and modification of these objects. Here we will standardize basic terminology for such search structures.

Most search structures can be viewed as a collection of **nodes** that are interconnected by pointers. Abstractly, they are just directed graphs. Each node stores or represents an object which we call an **item**. We will be informal about how we manipulate nodes – they will variously look like ordinary variables and pointers[2] as in the programming language `C/C++`, or like references in `Java`.

[1] Myth debunkers notwithstanding.

[2] The concept of **locatives** introduced by Lewis and Denenberg [5] may also be used: a locative u is like a pointer variable in programming languages, but it has properties like an ordinary variable. Informally, u will act like an ordinary variable in situations where this is appropriate, and it will act like a pointer variable if the situation demands it. This is achieved by suitable automatic referencing and dereferencing semantics for such variables.

It is convenient to assume a special kind of node called the `nil` node. Each item is associated with a **key**. The rest of the information in an item is simply called **data** so that we may view an item as the pair $(Key, Data)$. If u is a node, we write u.`Key` and u.`Data` for the key and data associated with the item represented by u.

Another concept is that of **iterators**. In search queries, we sometimes need to return a set of items. We basically have to return a list of nodes that represent these items in some order. The concept of an iterator captures this in an abstract way: We can view an iterator as a node u which has an associated field denoted u.`next`. This field is another iterator, or it is the special node `nil`. Thus, an iterator naturally represents a list of items.

Examples of search structures are:

(i) An *employee database* where each item is an employee record. The key of an employee record is the social security number, with associated data such as address, name, salary history, etc.

(ii) A *dictionary* where each item is a word entry. The key is the word itself, associated with data such as the pronounciation, part-of-speech, meaning, etc.

(iii) A *scheduling queue* in a computer operating systems where each item in the queue is a job that is waiting to be executed. The key is the priority of the job, which is an integer.

It is also natural to refer such structures as **keyed search structures**. From an algorithmic point of view, the properties of the search structure are solely determined by the keys in items, the associated data playing no role. This is somewhat paradoxical since, for the users of the search structure, it is the data that is more important. In any case, we may often ignore the data part of an item in our illustrations, thus identifying the item with the key (if the keys are unique).

Binary search trees is an example of a keyed search structure. Usually, each node of the binary search trees stores an item. In this case, our terminology of “nodes” for the location of items happily coincides with the concept of “tree nodes”. However, there are versions of binary search trees whose items resides only in the leaves – the internal nodes only store keys for the purpose of searching.

Key values usually come from a totally ordered set. Typically, we use the set of integers for our ordered set. Another common choice for key values are character strings ordered by lexicographic ordering. For simplicity in these notes, the default assumption is that items have unique keys. When we speak of the “largest item”, or “comparison of two items” we are referring to the item with the largest key, or comparison of the keys in two items, etc. Keys are called by different names to suggest their function in the structure. For example, a key may called a

- **priority**, if there is an operation to select the “largest item” in the search structure (see example (iii) above);
- **identifier**, if the keys are unique (distinct items have different keys) and our operations use only equality tests on the keys, but not its ordering properties (see examples (i) and (ii));
- **cost** or **gain**, depending on whether we have an operation to find the minimum or maximum value;
- **weight**, if key values are non-negative.

More precisely, a **search structure** S is a representation of a set of items that supports the `lookUp` query. The lookup query, on a given key K and S, returns a node u in S such that the item in u has key K.

If no such node exists, it returns $u = \mathsf{nil}$. Since S represents a set of items, two other basic operations we might want to do are inserting an item and deleting an item. If S is subject to both insertions and deletions, we call S a **dynamic set** since its members are evolving over time. In case insertions, but not deletions, are supported, we call S a **semi-dynamic set**. In case both insertion and deletion are not allowed, we call S a **static set**. The dictionary example (ii) above is a static set from the viewpoint of users, but it is a dynamic set from the viewpoint of the lexicographer.

Two search structures that store exactly the same set of items are said to be **equivalent**. An operation that preserves the equivalence class of a search structure is called an **equivalence transformation**.

§2. Abstract Data Types

This section contains a general discussion on abstract data types (ADT's). It may be used as a reference on ADT's.

It seems best the use the terminology of "classes" as found in programming languages such as `C/C++` or `Java`.

Search structures such as binary trees can be viewed as instances of a suitable class (e.g., a "binary tree class"). The functions (or "methods") of such class support some subset of the following operations, organized into four groups (I)-(IV) below.

(I) Initializer and Destroyers	`make`()
	`kill`()
(II) Enumeration and Order	`list`() $\rightarrow$*Node*,
	`succ`(*Node*)$\rightarrow$*Node*,
	`pred`(*Node*)$\rightarrow$*Node*,
	`min`()$\rightarrow$*Node*,
	`max`()$\rightarrow$*Node*,
	`deleteMin`()$\rightarrow$*Item*,
(III) Dictionary Operations	`lookUp`(*Key*)$\rightarrow$*Node*,
	`insert`(*Item*)$\rightarrow$*Node*,
	`delete`(*Node*),
(IV) Set Operations	`split`(*Key*)$\rightarrow$ *Structure*1,
	`merge`(*Structure*).

The meaning of these operations are fairly intuitive. We will briefly explain them. Let S, S' be search structures, viewed as instances of a suitable class. Let K be a key and u a node. Each of the above operations are invoked from some S: thus, S.`make`() will initialize the structure S, and S.`max`() returns the maximum value in S. But when there is only one structure S, we may suppress the reference to S, e.g., we write "`merge`(S')" instead of "S.`merge`(S')".

(I) We need to initialize and dispose of search structures. Thus `make` (with no arguments) returns a brand new empty instance of the structure. The inverse of `make` is `kill`, to remove a structure.

(II) The operation `list`() returns a node that is an iterator. This iterator is the beginning of a list that contains all the items in S in *some arbitrary* order. The ordering of keys is not used by the iterators.

The remaining operations in this group depend on the ordering properties of keys. The `min`() and `max`() operations are obvious. The successor `succ`(u) (resp., predecesssor `pred`(u)) of a node u refers to the node in S whose key has the next larger (resp., smaller) value. This is undefined if u has the largest (resp., smallest) value in S.

Note that `list`() can be implemented using `min`() and `succ`(u) or `max`() and `pred`(u). Such a listing has the additional property of sorting the output by key value.

The operation `deleteMin`() operation deletes the minimum item in S. In most data structures, we can replace `deleteMin` by `deleteMax` without trouble. However, this is not the same as being able to support both `deleteMin` and `deleteMax` simultaneously.

(III) The next three operations constitute the "dictionary operations". The node u returned by `lookUp`(K) should contain an item whose associated key is K. In conventional programming languages such as `C`, nodes are usually represented by pointers. In this case, the `nil` pointer can be returned by the `lookUp` function in case there is no item in S with key K. The structure S itself may be modified to another structure S' but S and S' must be equivalent.

In case no such item exists, or it is not unique, some convention should be established. At this level, we purposely leave this under-specified. Each application should further clarify this point. For instance, in case the keys are not unique, we may require that `lookUp`(K) returns an iterator that represents the entire set of items with key equal to K.

Both `insert` and `delete` have the obvious basic meaning. In some applications, we may prefer to have deletions that are based on key values. But such a deletion operation can be implemented as '`delete`(`lookUp`(K))'. In case `lookUp`(K) returns an iterator, we would expect the deletion to be performed over the iterator.

(IV) If `split`(K) $\rightarrow S'$ then all the items in S with keys greater than K are moved into a new structure S'; the remaining items are retained in S. In the operation `merge`(S'), all the items in S' are moved into S and S' itself becomes empty. This operation assumes that all the keys in S are less than all the items in S'. In a sense, `split` and `merge` are inverses of each other.

Some Abstract Data Types. The above operations are defined on typed domains (keys, structures, items) with associated semantics. An **abstract data type** (acronym "ADT") is specified by

- one or more "typed" domains of objects (such as integers, multisets, graphs);
- a set of operations on these objects (such as lookup an item, insert an item);
- properties (axioms) satisfied by these operations.

These data types are "abstract" because we make no assumption about the actual implementation. The following are some examples of abstract data types.

- **Dictionary**: `lookUp`, [`insert`, [`delete`]].
- **Ordered Dictionary**: `lookUp`, `insert`, `delete`, `succ`, `pred`.
- **Priority queue**: `deleteMin`, `insert`, [`delete`, [`update`]].

- **Fully mergeable dictionary**: `lookUp`, `insert`, `delete`, `merge`, `split`.

Sometimes, those operations within $[\cdots]$ can be omitted. If the deletion in dictionaries are based on keys (see comment above) then we may think of a dictionary as a kind of **associative memory**. If we omit the `split` operation in fully mergeable dictionary, then we obtain the **mergeable dictionary** ADT. The operations `make` and `kill` (from group (I)) are assumed to be present in every ADT.

In contrast to ADTs, data structures such as linked list, arrays or binary search trees are called **concrete data types**. We think of the ADTs as being **implemented** by concrete data types. For instance, a priority queue could be implemented using a linked list. But a more natural implementation is to represent D by a binary tree with the **min-heap property**: a tree has this property if the key at any node u is no larger than the key at any child of u. Thus the root of such a tree has a minimum key. Similarly, we may speak of a **max-heap property**. It is easy to design algorithms (Exercise) that maintains this property under the priority queue operations.

REMARKS:
1. Variant interpretations of all these operations are possible. For instance, some version of `insert` may wish to return a boolean (to indicate success or failure) or not to return any result (in case the application will never have an insertion failure).
2. Other useful functions can be derived from the above. E.g., it is useful to be able to create a structure S containing just a single item I. This can be reduced to 'S.`make`(); S.`insert`(I)'.
3. The concept of ADT was a major research topic in the 1980's, and many of these ideas found their way into structured programming languages such as Pascal and their modern successors. Our discussion of ADT is somewhat informal, but one way to study them formally is to describe axioms that these operations satisfy. For instance, if S is a stack, then pushing an item x on S followed by popping S should return the item x. Of course, we have relied on informal understanding of these ADT's to avoid such axiomatic treatment. For instance, an interface in Java is a kind of ADT where we capture only the types of operation.

Exercises

Exercise 2.1:
(a) Describe algorithms to implement all of the above operations where the concrete data structure are linked lists.
(b) Analyze the complexity of your algorithms in each case. ♢

Exercise 2.2: Repeat the previous question, but using arrays instead of linked lists in your implementation. ♢

Exercise 2.3: Suppose D is a dictionary with the dictionary operations of lookup, insert and delete. List a complete set of axioms for these operations. ♢

End Exercises

§3. Binary Search Trees

We introduce binary search trees and show that such trees can support all the operations described in the previous section on ADT. Our approach will be somewhat unconventional, because we want to reduce all these operations to the single operation of "rotation".

Recall the definition and basic properties of binary trees in the Appendix of Chapter I. Briefly, a binary tree T is a set N of nodes that is either the empty set, or N has a node u called the root. The remaining nodes $N \setminus \{u\}$ are partitioned into two sets of nodes that recursively form binary trees, T_L and T_R. If T_L (resp., T_R) is non-empty, then its root is called the left (resp., right) child of u. This definition of binary trees is called **structural induction**. The **size** of T is $|N|$, and is denoted $|T|$; also T_L, T_R are the **left** and **right subtrees** of T.

To turn binary trees into search structures, we must store a key at each node. A binary tree T is called **binary search tree** (BST) if we associate a key $u.\mathtt{Key}$ at each node u, subject to the **binary search tree property**:

$$u_L.Key < u.Key \le u_R.Key. \tag{1}$$

where u_L and u_R are (resp.) any **left descendent** and **right descendent** of u. By definition, a left (right) descendent of u is a node in the subtree rooted at the left (right) child of u. The left and right children of u are denoted by $u.\mathtt{left}$ and $u.\mathtt{right}$.

Before proceeding, the student will do well to remember a general rule about binary trees: just as binary trees are best defined by structural induction, *most properties about binary trees are best proved by induction on the structure of the tree.*

Distinct versus duplicate keys. Binary search trees can be used to store a set of items whose keys are distinct, or items that may have duplicate keys. Since we allow the right child to have an equal key with its parent, we see that under pure insertions, all the elements with the same key forms a "right path". We could modify all our algorithms to preserve this property. Alternatively, we could just place all the equal-key items in a linked list as an auxilliary structure attached to a node. In any case, this would complicate our algorithms. Hence, in the following, *we will assume distinct keys unless otherwise noted.*

Lookup. The algorithm for key lookup in a binary search tree is almost immediate from the binary search tree property: to look for a key K, we begin at the root. In general, suppose we are looking for K in some subtree rooted at node u. If $u.\mathtt{Key} = K$, we are done. Otherwise, either $K < u.\mathtt{Key}$ or $K > u.\mathtt{Key}$. In the former case, we recursively search the left subtree of u; otherwise, we recurse in the right subtree of u. In the presense of duplicate keys, what does lookup return? There are two interpretations: (1) We can return the first node u we find that has the given key K. (2) We may insist that we continue to explicitly locate all the other keys.

In any case, requirement (2) can be regarded as an extension of (1), namely, given a node u, find all the other nodes below u with same same key as $u.Key$. This can be solved separately. Hence we may assume interpreation (1) in the following.

Insertion. To insert an item with key K, we proceed as in the Lookup algorithm. If we find K in the tree, then the insertion fails (assuming distinct keys). Otherwise, we reach a leaf node u. Then the item can be inserted as the left child of u if $u.\mathtt{Key} > K$, and otherwise it can be inserted as the right child of u. In any case, the inserted item is a new leaf of the tree.

Rotation. This is not a listed operation in §2. It is an equivalence operation, i.e., it transforms a binary search tree into another one with exactly the same set of keys. By itself, rotation does not appear to do anything useful. But it can be the basis for almost all of our operations.

The operation `rotate`(u) is a null operation ("no-op" or identity transformation) when u is a root. So assume u is a non-root node in a binary search tree T. Then `rotate`(u) amounts to the following transformation of T (see figure 1).

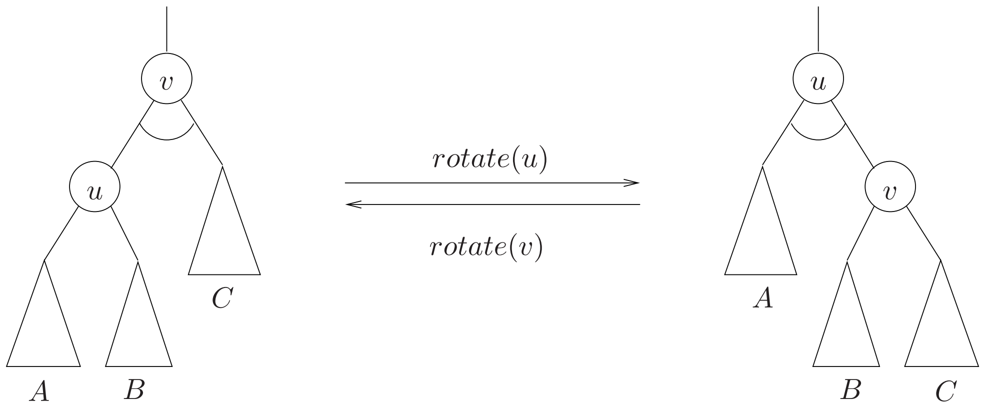


Figure 1: Rotation at u and its inverse.

In `rotate`(u), we basically want to invert the parent-child relation between u and its parent v. The other transformations are more or less automatic, given that the result is to remain a binary search tree. If the subtrees A, B, C (any of these can be empty) are as shown in figure 1, then they must re-attach as shown. This is the only way to reattach as children of u and v, since we know that

$$A < B < C$$

in the sense that each key in A is less than any key in B, etc. Actually, only the parent of the root of B has switched from u to v. Notice that after `rotate`(u), the former parent of v (not shown) will now have u instead of v as a child. Clearly the inverse of `rotate`(u) is `rotate`(v). The explicit pointer manipulations for a rotation are left as an exercise. After a rotation at u, the depth of u is decreased by 1. Note that `rotate`(u) followed by `rotate`(v) is the identity[3] operation, as illustrated in figure 1.

Recall that two search structures are equivalent if they contain the same set of items. Clearly, rotation is an equivalence transformation.

Graphical convention: Figure 1 encodes two conventions: consider the figure on the left side of the arrow (the same convention hold for the figure on the right side). First, the edge connecting v to its parent is directed vertically upwards. This indicates that v can be the left- or right-child of its parent. Second, the two edges from v to its children are connected by a circular arc. This is to indicate that u and its sibling could exchange places (i.e., u could be the right-child of v even though we choose to show u as the left-child). Thus Figure 1 is a compact way to represent four distinct situations.

Implementation of rotation. Let us discuss how to implement rotation. Until now, when we draw binary trees, we only display child pointers. But we must now explicitly discuss parent pointers.

[3] Also known as null operation or no-op

Let us classify a node u into one of three **types**: *left*, *right* or *root*. This is defined in the obvious way. E.g., u is a left type iff it is not a root and is a left child. The type of u is easily tested: u is type root iff $u.\texttt{parent} = \textsf{nil}$, and u is type left iff $u.\texttt{parent.left} = u$. Clearly, $\texttt{rotate}(u)$ is sensitive to the type of u. In particular, if u is a root then $\texttt{rotate}(u)$ is the null operation. If $T \in \{\texttt{left}, \texttt{right}\}$ denote left or right type, its **complementary type** is denoted $\overline{T}$, where $\overline{\texttt{left}} = \texttt{right}$ and $\overline{\texttt{right}} = \texttt{left}$.

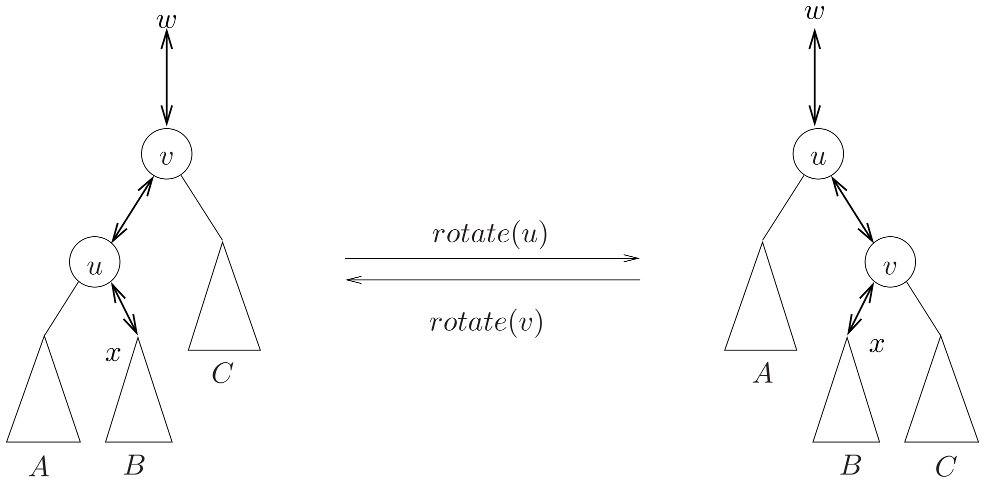


Figure 2: Links that must be fixed in $\texttt{rotate}(u)$.

We are ready to discuss the function $\texttt{rotate}(u)$, which we assume will return the node u. Assume u is not the root, and its type is $T \in \{\texttt{left}, \texttt{right}\}$. Let $v = u.\texttt{parent}$, $w = v.\texttt{parent}$ and $x = u.\overline{T}$. Note that w and x might be $\textsf{nil}$. Thus we have potentially three child-parent pairs:

$$(x, u), (u, v), (v, w). \tag{2}$$

But after rotation, we will have the transformed child-parent pairs:

$$(x, v), (v, u), (u, w). \tag{3}$$

These pairs are illustrated in Figure 2 where we have explicitly indicated the parent pointers as well as child pointers. Thus, to implement rotation, we need to reassign 6 pointers (3 parent pointers and 3 child pointers). We show that it is possible to achieve this re-assignment using exactly 6 assignments.

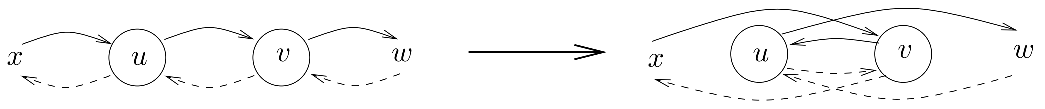


Figure 3: Simplified view of $\texttt{rotate}(u)$ as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (2) as a doubly-linked list (x, u, v, w) which must be converted into the doubly-linked list (x, v, u, w) in (3). This is illustrated in Figure 3. For simplicity, we use the terminology of doubly-linked list so that $u.\texttt{next}$ and $u.\texttt{prev}$ are the forward and backward pointers of a doubly-linked list. Here is[4] the code:

[4] In Lines 3 and 5, we used the node u as a pointer on the right hand side of an assignment statement. Strictly speaking, we ought to take the address of u before assignment. Alternatively, think of u as a "locator variable" which is basically a pointer variable with automatic ability to dereference into a node when necessary.

```
Rotate($u$):
    ▷ Fix the forward pointers
        1.  $u$.prev.next ← $u$.next
                ◁ $x$.next = $v$
        2.  $u$.next ← $u$.next.next
                ◁ $u$.next = $w$
        3.  $u$.prev.next.next ← $u$
                ◁ $v$.next = $u$
    ▷ Fix the backward pointers
        4.  $u$.next.prev.prev ← $u$.prev
                ◁ $v$.prev = $x$
        5.  $u$.next.prev ← $u$
                ◁ $w$.prev = $u$
        6.  $u$.prev ← $u$.prev.next
                ◁ $u$.prev = $v$
```

We can now translate this sequence of 6 assignments into the corresponding assignments for binary trees: the u.`next` pointer may be identified with u.`parent` pointer. However, u.`prev` would be $u.T$ where $T \in \{\texttt{left}, \texttt{right}\}$ is the type of x. Moreover, v.`prev` is $v.\overline{T}$. Also w.`prev` is $w.T'$ for another type T'. A further complication is that x or/and w may not exist; so these conditions must be tested for, and appropriate modifications taken.

If we use temporary variables in doing rotation, the code can be simplified (Exercise).

Variations on Rotation. The above rotation algorithm assumes that for any node u, we can access its parent. This is true if each node has a parent pointer u.`parent`. *This is our default assumption for binary trees.* In case this assumption fails, we can replace rotation with a pair of variants: called **left-rotation** and **right-rotation**. These can be defined as follows:

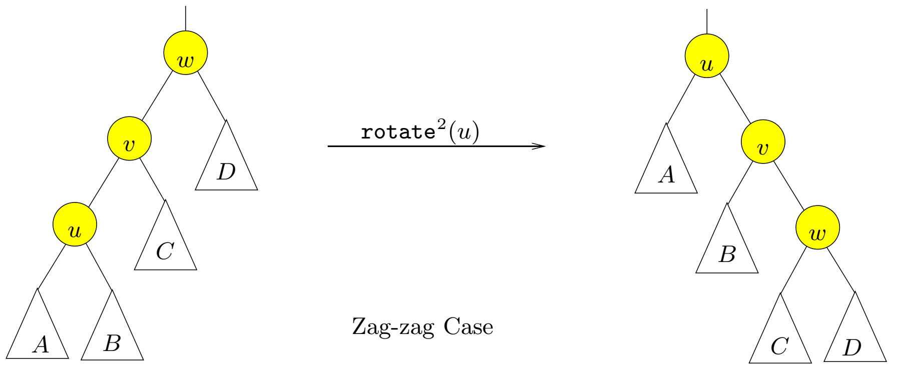
$$\texttt{left-rotate}(u) \equiv \texttt{rotate}(u.\texttt{left}), \qquad \texttt{right-rotate}(u) \equiv \texttt{rotate}(u.\texttt{right}).$$

It is not hard to modify all our rotation-based algorithms to use the left- and right-rotation formulation if we do not have parent pointers. Of course, the corresponding code would be twice as fast since we have halved the amount of work.

Double Rotation. Suppose u has a parent v and a grandparent w. Then two successive rotations on u will ensure that v and w are descendents of u. We may denote this operation by $\texttt{rotate}^2(u)$. Up to left-right symmetry, there are two distinct outcomes in $\texttt{rotate}^2(u)$: (i) either v, w are becomes children of u, or (ii) only w becomes a child of u and v a grandchild of u. These depend on whether u is the **outer** or **inner** grandchildren of w. These two cases are illustrated in Figure 4. [As an exercise, we ask the reader to draw the intermediate tree after the first application of $\texttt{rotate}(u)$ in this figure.]

It turns out that case (ii) is the more important case. For many purposes, we would like to view the two rotations in this case as one indivisible operation: hence we introduce the term **double rotation** to refer to case (ii) only. For emphasis, we might call the original rotation a **single rotation**.

These two cases are also known as the zig-zig (or zag-zag) and zig-zag (or zag-zig) cases, respectively. This terminology comes from viewing a left turn as zig, and a right turn as zag, as we move from up a root path. The Exercise considers how we might implement a double rotation more efficiently than by simply doing two single rotations.



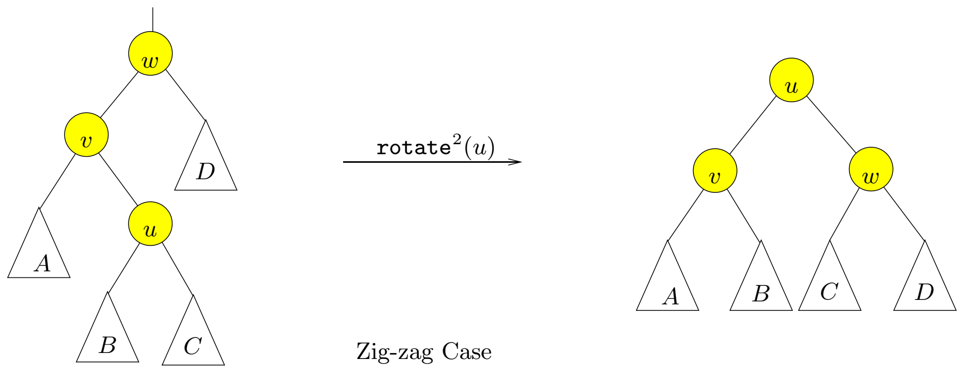


Figure 4: Two outcomes of $\texttt{rotate}^2(u)$

Root path, Extremal paths and Spines. A path is a sequence of nodes $(u_0, u_1, \ldots, u_n)$ where u_{i+1} is a child of u_i. The length of this path is n, and u_n is also called the **tip** of the path. Relative to a node u, we now introduce 5 paths that originates from u. The first is the path from u to the root, called the **root path** of u. In figures, the root path is displayed as an upward path from the node u. Next we introduce 4 downward paths from u. The **left-path** of u is simply the path that starts from u and keeps moving towards the left or right child until we cannot proceed further. The **right-path** of u is similarly defined. Collectively, we refer to the left- and right-paths as **extremal paths**. Next, we define the **left-spine** of a node u is defined to be the path $(u, \mathrm{rightpath}(u.\texttt{left}))$. In case $u.\texttt{left} = \mathsf{nil}$, the left spine is just the trivial path (u) of length 0. The **right-spine** is similarly defined. The tips of the left- and right-paths at u correspond to the minimum and maximum keys in the subtree at u. The tips of the left- and right-spines, provided they are different from u itself, correspond to the predecessor and successor of u. Clearly, u is a leaf iff all these four tips are identical and equal to u.

After performing a left-rotation at u, we reduce the left-spine length of u by one (but the right-spine of u is unchanged). See Figure 5. More generally:

Lemma 1. *Let $(u_0, u_1, \ldots, u_k)$ be the left-spine of u and $k \geq 1$. Also let $(v_0, \ldots, v_m)$ be the root path of u, where v_0 is the root of the tree and $u = v_m$. After performing* $\texttt{rotate}(u.\texttt{left})$*,*
(i) the left-spine of u becomes $(u_0, u_2, \ldots, u_k)$ of length $k-1$,
(ii) the right-spine of u is unchanged, and
(iii) the root path of u becomes $(v_0, \ldots, v_m, u_1)$ of length $m+1$.

In other words, after a left-rotation at u, the left child of u transfers from the left-spine of u to the root path of u. Similar remarks apply to right-rotations. If we repeatedly do left-rotations at u, we will reduce

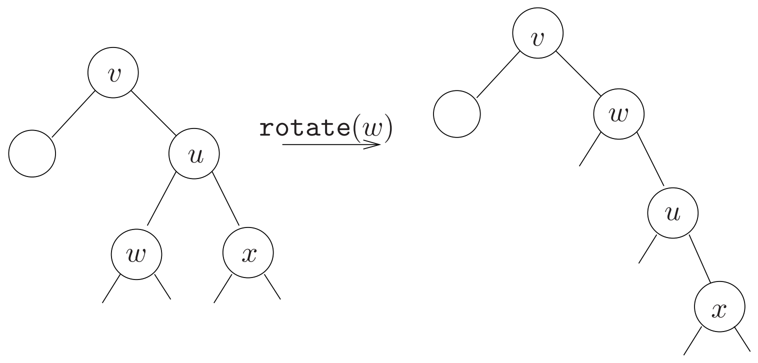


Figure 5: Reduction of the left-spine of u after $\texttt{rotate}(u.\texttt{left}) = \texttt{rotate}(w)$.

the left-spine of u to length 0. We may also alternately perform left-rotates and right-rotates at u until one of its 2 spines have length 0.

Deletion. Suppose we want to delete a node u. In case u has at most one child, this is easy to do – simply redirect the parent's pointer to u into the unique child of u (or nil if u is a leaf). Call this procedure $Cut(u)$. It is now easy to describe a general algorithm for deleting a node u:

```
DELETE($T, u$):
Input:    $u$ is node to be deleted from $T$.
Output:   $T$, the tree with $u$ deleted.
    while $u$.left $\neq$ nil do
        rotate($u$.left).
    Cut($u$)
```

If we maintain information about the left and right spine heights of nodes (Exercise), and the right spine of u is shorter than the left spine, we can also perform the while-loop by going down the right spine instead.

Contrast this with the following **standard deletion algorithm**:

```
DELETE($T, u$):
Input:    $u$ is node to be deleted from $T$.
Output:   $T$, the tree with item in $u$ deleted.
    if $u$ has at most one child, apply Cut($u$) and return.
    else let $v$ be the tip of the right spine of $u$.
        Move the item in $v$ into $u$ (effectively removing the item in $u$)
        Cut($v$).
```

Note that in the else-case, the node u is not physically removed: only the item represented by u is removed. Again, the node v that is physically removed has at most one child.

The rotation-based deletion is conceptually simpler, and will also be useful for amortized algorithms later. But all else being equal, the rotation-based algorithm seems to be less desirable and slower.

Tree Traversals. There are three systematic ways to list all the nodes in a binary tree: the most important is the **in-order** or **symmetric traversal**. Here is the recursive procedure to perform an in-order traversal of a tree rooted at u:

```
IN-ORDER(u):
Input:    u is root of binary tree T to be traversed.
Output:   The in-order listing of the nodes in T.
     0.   BASE(u).
     1.   In-order(u.left).
     2.   VISIT(u).
     3.   In-order(u.right).
```

This recursive program calls two subroutines. The BASE subroutine is just the line:

if ($u =$ nil) return.

The VISIT(u) subroutine is simply:

Print u.Key.

For example, consider the tree in figure 6. The numbers on the nodes are not keys, but serve as identifiers.

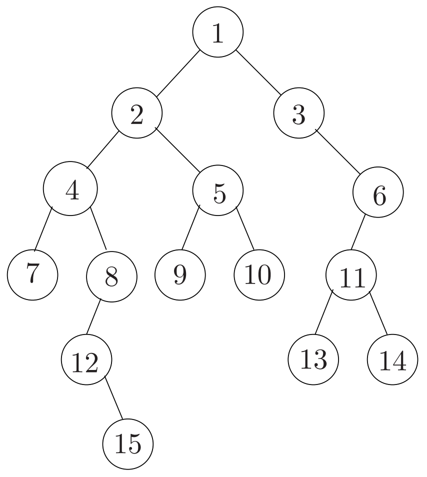


Figure 6: Binary tree.

An in-order traversal of the tree will produce the listing of nodes

$$7, 4, 12, 15, 8, 2, 9, 5, 10, 1, 3, 13, 11, 14, 6.$$

We can write the above problem succinctly as:

$$IN(u) \equiv [BASE(u); IN(u.\mathtt{left}); VISIT(u); IN(u.\mathtt{right})]$$

Changing the order of Steps 1, 2 and 3 in the In-Order procedure (but always doing Step 1 before Step 3), we obtain two other methods of tree traversal. Thus, if we perform Step 2 before Steps 1 and 3, the result is called the **pre-order traversal** of the tree:

$$PRE(u) \equiv [BASE(u); VISIT(u); PRE(u.\mathtt{left}); PRE(u.\mathtt{right})]$$

Applied to the tree in figure 6, we obtain

$$1, 2, 4, 7, 8, 12, 15, 5, 9, 10, 3, 6, 11, 13, 14.$$

If we perform Step 2 after Steps 1 and 3, the result is called the **post-order traversal** of the tree:

$$POST(u) \equiv [BASE(u); POST(u.\texttt{left}); POST(u.\texttt{right}); VISIT(u)]$$

Using the same example, we obtain

$$7, 15, 12, 8, 4, 9, 10, 5, 2, 13, 14, 11, 6, 3, 1.$$

Tree traversals may not appear interesting on their own right. However, they serves as "shells" for solving other more interesting problems. All we have to do is to choose a different subroutines for BASE(u) and VISIT(u). Let us illustrate this: suppose we want to compute the height of each node of a BST. Assume that each node u has a variable $u.H$ that is to store the height of node u. We can use the "post-order shell" to accomplish this task:

```
POST($u$)
    $BASE(u)$.
    $POST(u.\texttt{left})$.
    $POST(u.\texttt{right})$.
    $VISIT(u)$.
```

We can keep the previous BASE subroutine, but modify $VISIT(u)$ to the following task:

```
VISIT($u$)
    if ($u.\texttt{left}$ = nil) then $L \leftarrow -1$.
        else $L \leftarrow u.\texttt{left}.H$.
    if ($u.\texttt{right}$ = nil) then $R \leftarrow -1$.
        else $L \leftarrow u.\texttt{right}.H$.
    $u.H \leftarrow 1 + \max\{L, R\}$.
```

The correctness of this procedure is obvious.

Successor and Predecessor. If u is a node of a binary tree T, the **successor** of u refers to the node v that is listed **after** u in the in-order traversal of the nodes of T. By definition, u is the **predecessor** of v iff v is the successor of u. Let `succ`(u) and `pred`(u) denotes the successor and predecessor of u. Of course, `succ`(u) (resp., `pred`(u)) is undefined if u is the last (resp., first) node in the in-order traversal of the tree.

We will define a closely related concept, but applied to any key K. Let K be a key, not necessarily occurring in T. Define the **successor** of K in T to be the least key K' in T such that $K < K'$. We similarly define the **predecessor** of K in T to be the greatest K' in T such that $K' > K$.

In some applications of binary trees, we want to maintain pointers to the successor and predecessor of each node. In this case, these pointers may be denoted u.`succ` and u.`pred`. Note that the successor/predecessor pointers of nodes is unaffected by rotations. *Our default version of binary trees do not include such pointers.*

Let us make some simple observations:

Lemma 2. *Let u be a node in a binary tree, but u is not the last node in the in-order traversal of the tree.*
(i) u.right = nil iff u is the tip of the left-spine of some node v. Moreover, such a node v is uniquely determined by u.
(ii) If u.right = nil and u is the tip of the left-spine of v, then succ$(u) = v$.
(iii) If u.right $\neq$ nil then succ(u) is the tip of the right-spine of u.

It is easy to derive an algorithm for succ(u) using the above observation.

```
Succ(u):
    1.    if u.right ≠ nil ◁ return the tip of the right-spine of u
    1.1        v ← u.right;
    1.2        while v.left ≠ nil, v ← v.left;
    1.3        return(v).
    2.    else ◁ return v where u is the tip of the left-spine of v
    2.1        v ← u.parent;
    2.2        while v ≠ nil and u = v.right,
    2.3            (u, v) ← (v, v.parent).
    2.4        return(v).
```

Note that if succ(u) = nil then u is the last node in the in-order traversal of the tree (so u has no successor). The algorithm for pred(u) is similar.

Min, Max, DeleteMin. This is trivial once we notice that the minimum (maximum) item is in the last node of the left (right) subpath of the root.

Merge. To merge two trees T, T' where all the keys in T are less than all the keys in T', we proceed as follows. Introduce a new node u and form the tree rooted at u, with left subtree T and right subtree T'. Then we repeatedly perform left rotations at u until u.left = nil. Similarly, perform right rotations at u until u.right = nil. Now u is a leaf and can be deleted. The result is the merge of T and T'.

Split. Suppose we want to split a tree T at a key K. First we do a lookUp of K in T. This leads us to a node u that either contains K or else u is the successor or predecessor of K in T. Now we can repeatedly rotate at u until u becomes the root of T. At this point, we can split off either the left-subtree or right-subtree of T. This pair of trees is the desired result.

Complexity. Let us now discuss the worst case complexity of each of the above operations. They are all $O(h)$ where h is the height of the tree. It is therefore desirable to be able to maintain $O(\log n)$ bounds on the height of binary search trees.

REMARK: It is important to stress that our rotation-based algorithms for insertion and deletion is going to be slower than the "standard" algorithms which perform only a constant number of pointer re-assignments. Therefore, it seems that rotation-based algorithms may be impractical unless we get other benefits. One possible benefit of rotation will be explored in Chapter 6 on amortization and splay trees.

Exercises

Exercise 3.1: The function Verify(u) is supposed return true iff the binary tree rooted at u is a binary search tree with distinct keys:

```
Verify(Node $u$)
    if ($u = $nil) return(true)
    if (($u$.left $\neq$ nil) and ($u.Key < u$.left.$Key$)) return(false)
    if (($u$.right $\neq$ nil) and ($u.Key > u$.right.$Key$)) return(false)
    return(Verify($u$.left)$\wedge$Verify($u$.right))
```

Either argue for it's correctness, or give a counter-example showing it is wrong. ♢

Exercise 3.2: TRUE or FALSE: Recall that a rotation can be implemented with 6 pointer assignments. Suppose a binary search tree maintains successor and predecessor links (denoted u.succ and u.pred in the text). Now rotation requires 12 pointer assignments. ♢

Exercise 3.3: (a) Implement the above binary search tree algorithms (rotation, lookup, insert, deletion, etc) in your favorite high level language. Assume the binary trees have parent pointers.
(b) Describe the necessary modifications to your algorithms in (a) in case the binary trees do not have parent pointers. ♢

Exercise 3.4: Let T be the binary search tree in figure 7.

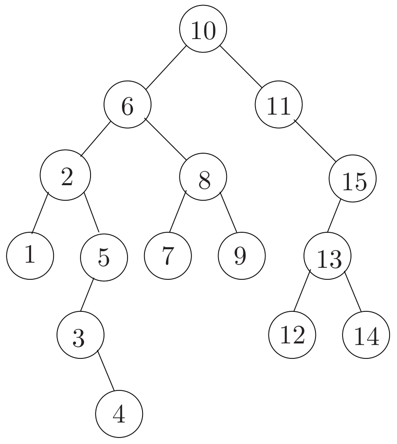


Figure 7: A binary search tree.

(a) Perform the operation split$(T, 5) \to T'$. Display T and T' after the split.
(b) Now perform insert$(T, 3.5)$ where T is the tree after the operation in (a). Display the tree after insertion.
(c) Finally, perform merge(T, T') where T is the tree after the insert in (b) and T' is the tree after the split in (a). ♢

Exercise 3.5: Give the code for rotation which uses temporary variables. ♢

Exercise 3.6: Instead of minimizing the number of assignments, let us try to minimize the time. To count time, we count each reference to a pointer as taking unit time. For instance, the assignment

u.`next.prev.prev` $\leftarrow u$.`prev` costs 5 time units because in addition to the assignment, we have to make access 4 pointers.
(a) What is the rotation time in our 6 assignment solution in the text?
(b) Give a faster rotation algorithm, by using temporary variables. $\diamondsuit$

Exercise 3.7: We could implement a double rotation as two successive rotations, and this would take 12 assignment steps.
(a) Give a simple proof that 10 assignments are necessary.
(b) Show that you could do this with 10 assignment steps. $\diamondsuit$

Exercise 3.8: Open-ended: The problem of implementing `rotate`(u) without using extra storage or in minimum time (previous Exercise) can be generalized. Let G be a directed graph where each edge ("pointer") has a name (e.g., `next`, `prev`, `left`, `right`) taken from a fixed set. Moreover, there is at most one edge with a given name coming out of each node. Suppose we want to transform G to another graph G', just by reassignment of these pointers. Under what conditions can this transformation be achieved with only one variable u (as in `rotate`(u))? Under what conditions is the transformation achievable at all (using more intermediate variables? We also want to achieve minimum time. $\diamondsuit$

Exercise 3.9: The goal of this exercise is to show that if T_0 and T_1 are two equivalent binary search trees, then there exists a sequence of rotations that transforms T_0 into T_1. Assume the keys in each tree are distinct. This shows that rotation is a "universal" equivalence transformation. We explore two strategies.
(a) One strategy is to first make sure that the roots of T_0 and T_1 have the same key. Then by induction, we can transform the left- and right-subtrees of T_0 so that they are identical to those of T_1. Let $R_1(n)$ be the worst case number of rotations using this strategy on trees with n keys. Give a tight analysis of $R_1(n)$.
(b) Another strategy is to show that any tree can be reduced to a canonical form. Let us choose the canonical form where our binary search tree is a **left-list** or a **right-list**. A left-list (resp., right-list) is a binary trees in which every node has no right-child (resp., left-child). Let $R_2(n)$ be defined for this strategy in analogy to $R_1(n)$. Give a tight analysis of $R_2(n)$. $\diamondsuit$

Exercise 3.10: Design an algorithm to find both the successor and predecessor of a given key K in a binary search tree. It should be more efficient than just finding the successor and finding the predecessor independently. $\diamondsuit$

Exercise 3.11: Give the in-order, pre-order and post-order listing of the tree in Figure 11. $\diamondsuit$

Exercise 3.12: Tree traversals. Assume the following binary trees have distinct (names of) nodes.
(a) Let the in-order and pre-order traversal of a binary tree T with 10 nodes be $(a, b, c, d, e, f, g, h, i, j)$ and $(f, d, b, a, c, e, h, g, j, i)$, respectively. Draw the tree T.
(b) Prove that if we have the pre-order and in-order listing of the nodes in a binary tree, we can reconstruct the tree.
(c) Consider the other two possibilities: (c.1) pre-order and post-order, and (c.2) in-order and post-order. State in each case whether or not they have the same reconstruction property as in (b). If so, prove it. If not, show a counter example.
(d) Redo part(b) for full binary trees. $\diamondsuit$

Exercise 3.13:

(a) Here is the set of keys from post-order traversal of a binary search tree:

$$2, 1, 4, 3, 6, 7, 9, 11, 10, 8, 5, 13, 16, 15, 14, 12$$

Draw this binary search tree.
(b) Describe the general algorithm to reconstruct a BST from its post-order traversal. ♢

Exercise 3.14: Show that if a binary search tree has height h and u is any node, then a sequence of $k \geq 1$ repeated executions of the assignment $u \leftarrow \mathit{successor}(u)$ takes time $O(h + k)$. ♢

Exercise 3.15: Show how to efficiently maintain the heights of the left and right spines of each node. (Use this in the rotation-based deletion algorithm.) ♢

Exercise 3.16: We refine the successor/predecessor relation. Suppose that T^u is obtained from T by pruning all the proper descendants of u (so u is a leaf in T^u). Then the successor and predecessor of u in T^u are called (respectively) the **external successor** and **predecessor** of u in T Next, if T_u is the subtree at u, then the successor and predecessor of u in T_u are called (respectively) the **internal successor** and **predecessor** of u in T
(a) Explain the concepts of internal and external successors and predecessors in terms of spines.
(b) What is the connection between successors and predecessors to the internal or external versions of these concepts? ♢

Exercise 3.17: Give the rotation-based version of the successor algorithm. ♢

Exercise 3.18: Suppose that we begin with u at minimum node of a binary tree, and continue to apply the rotation-based successor (see previous question) until u is at the maximum node. Bound the number of rotations made as a function of n (the size of the binary tree). ♢

Exercise 3.19: Suppose we allow allow duplicate keys (but when we want to distinguish among these keys in Lookup, we can somehow do this). Under (1), all the keys with the same value must lie in a "right-path chain". Discuss how this property can be preserved in the algorithms for rotation, insertion, deletion. Also, discuss the effect of this on complexity. ♢

End Exercises

§4. AVL Trees

AVL trees is the first known family of balanced trees. By definition, an AVL tree is a binary search tree in which the left subtree and right subtree at each node differ by at most 1 in height. They also have relatively simple insertion/deletion algorithms.

More generally, define the **balance** of any node u of a binary tree to be the height of the left subtree minus the height of the right subtree:

$$\mathit{balance}(u) = \mathit{ht}(u.\texttt{left}) - \mathit{ht}(u.\texttt{right}).$$

The node is **perfectly balanced** if the balance is 0. It is **AVL-balanced** if the balance is either 0 or ± 1. Our insertion and deletion algorithms will need to know this balance information at each node. Thus we need to store at each AVL node a 3-valued variable. Theoretically, this space requirement amounts to $\lg 3 < 1.585$ bits per node. Of course, in practice, AVL trees will reserve 2 bits per node for the balance information (but see Exercise).

Let us first prove that the family of AVL trees is a balanced family. It is best to introduce the function $\mu(h)$, defined as the minimum number of nodes in any AVL tree with height h. The first few values are

$$\mu(-1) = 0, \qquad \mu(0) = 1, \qquad \mu(1) = 2, \qquad \mu(2) = 4.$$

These initial values are not entirely obvious: it seems clear that $\mu(0) = 1$ since there is a unique tree with height 0. To see[5] that $\mu(1) = 2$, *we must define the height of the empty tree to be* -1. This explains why $\mu(-1) = 0$. We can verify $\mu(2) = 4$ by case analysis.

In general, $\mu(h)$ is seen to satisfy the recurrence

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geq 1). \tag{4}$$

This corresponds to the minimum size tree of height h having left and right subtrees which are minimum size trees of heights $h-1$ and $h-2$. For instance, $\mu(2) = 1 + \mu(1) + \mu(0) = 1 + 2 + 1 = 4$, as we found by case analysis above. We similarly check that the recurrence (4) holds for $h = 1$. See figure 8 for the smallest AVL trees of the first few values of h. We have drawn these AVL trees where all internal nodes has balance of $+1$.

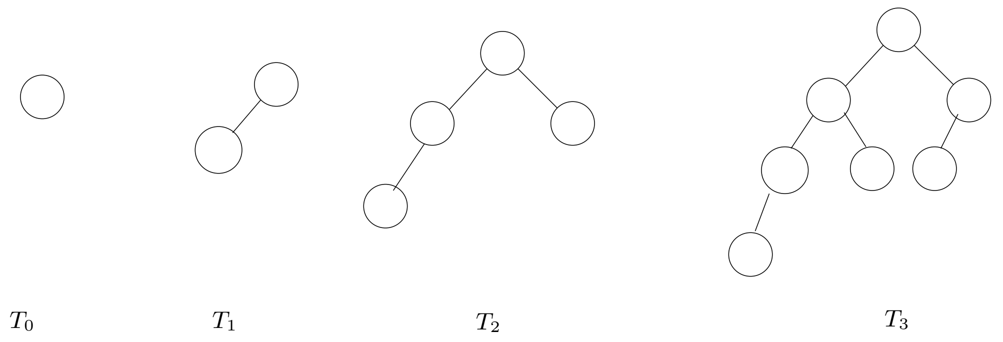


Figure 8: Smallest AVL trees of heights 0, 1, 2 and 3.

Lemma 3.

$$\mu(h) \geq C^h, \qquad (h \geq 1) \tag{5}$$

where $C = \sqrt{2} = 1.4142\ldots$.

Proof. From (4), we have $\mu(h) \geq 2\mu(h-2)$ for $h \geq 1$. It is easy to see by induction that $\mu(h) \geq 2^{h/2}$ for all $h \geq 1$. **Q.E.D.**

We can easily sharpen the constant C in this lemma. Let $\phi = \frac{1+\sqrt{5}}{2} > 1.6180$. This is the golden ratio and it is the positive root of the quadratic equation $x^2 - x - 1 = 0$. Hence, $\phi^2 = \phi + 1$. We claim:

$$\mu(h) \geq \phi^h, \quad h \geq 0.$$

The cases $h = 0$ and $h = 1$ are immediate. For $h \geq 2$, we have

$$\mu(h) > \mu(h-1) + \mu(h-2) \geq \phi^{h-1} + \phi^{h-2} = (\phi + 1)\phi^{h-2} = \phi^h.$$

[5]See Exercise for a different version.

So any AVL tree with n nodes and height h must satisfy the inequality $\phi^h \le n$ or $h \le (\lg n)/(\lg \phi)$. Thus the height of an AVL Tree on n nodes is at most $(\log_\phi 2)\lg n$ where $\log_\phi 2 = 1.4404....$ An exercise below shows how to further sharpen this estimate.

If an AVL tree has n nodes and height h then

$$n \ge \mu(h)$$

follows by definition of $\mu(h)$. Combined with (5), we conclude that $n \ge C^h$. Taking logs, we obtain $\log_C(n) \ge h$ or $h = O(\log n)$. This proves:

Corollary 4. *The family of AVL trees is balanced.*

Insertion and Deletion Algorithms. These algorithms for AVL trees are relatively simple, as far as balanced trees go. In either case there are two phases:

UPDATE PHASE: Insert or delete as we would in a binary search tree. REMARK: We assume here the *standard* deletion algorithm, not its rotational variant. Furthermore, the node containing the deleted key and the node we *physically* removed may be different.

REBALANCE PHASE: Let x be the parent of node that was just inserted, or just *physically* deleted, in the UPDATE PHASE. We now retrace the path from x towards the root, rebalancing nodes along this path as necessary. For reference, call this the **rebalance path**.

It remains to give details for the REBALANCE PHASE. If every node along the rebalance path is balanced, then there is nothing to do in the REBALANCE PHASE. Otherwise, let u be the first unbalanced node we encounter as we move upwards from x to the root. It is clear that u has a balance of ± 2. In general, we fix the balance at the "current" unbalanced node and continue searching upwards along the rebalance path for the next unbalanced node. Let u be the current unbalanced node. By symmetry, we may suppose that u has balance 2. Suppose its left child is node v and has height $h+1$. Then its right child v' has height $h-1$. This situation is illustrated in Figure 9.

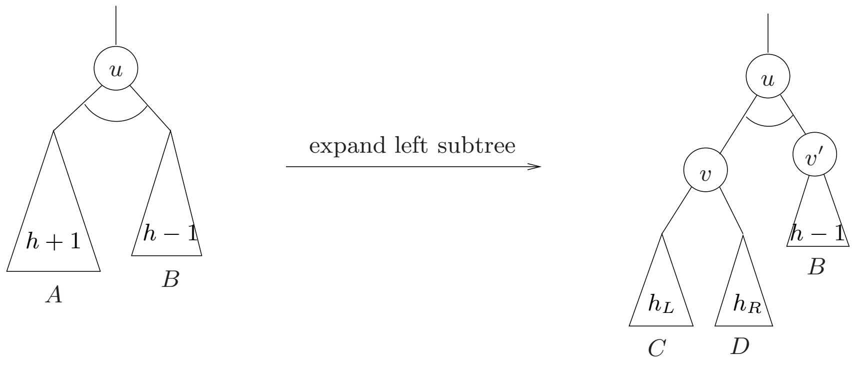


Figure 9: Node u is unbalanced after insertion or deletion.

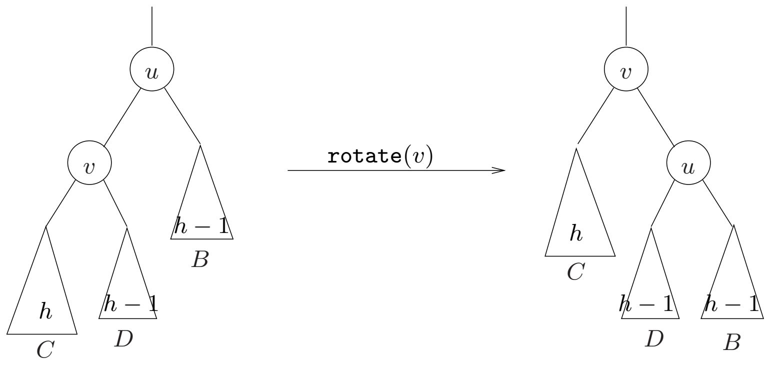
By definition, all the proper descendents of u are balanced. The current height of u is $h+2$. In any case, let the current heights of the children of v be h_L and h_R, respectively.

Insertion Rebalancing. Suppose that this imbalance came about because of an insertion. What was the heights of u, v and v' before the insertion? It is easy to see that the previous heights are (respectively)

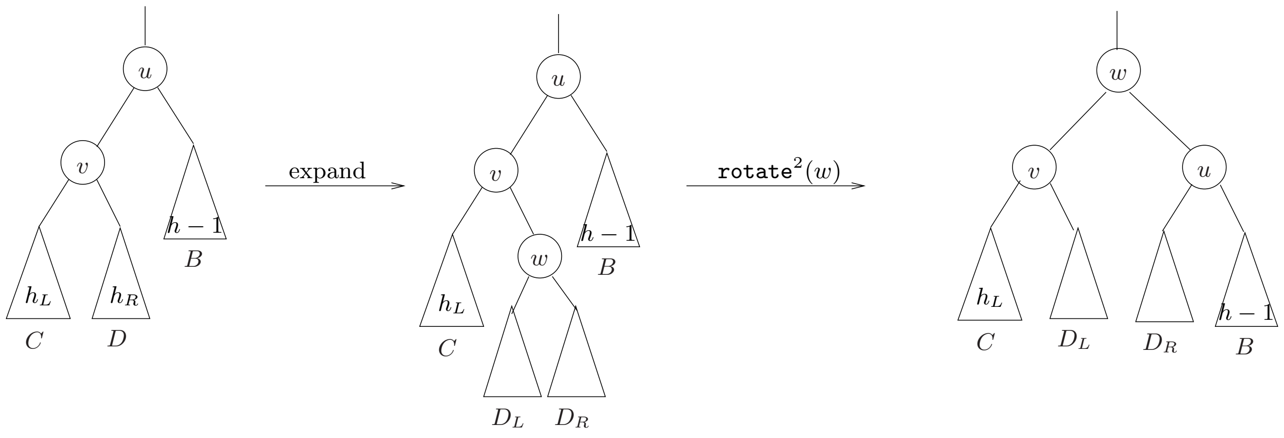
$$h+1, h, h-1.$$

The inserted node x must be in the subtree rooted at v. Clearly, the heights h_L, h_R of the children of v satisfy $\max(h_L, h_R) = h$. Since v is currently balanced, we know that $\min(h_L, h_R) = h$ or $h-1$. But in fact, we claim that $\min(h_L, h_R) = h-1$. To see this, note that if $\min(h_L, h_R) = h$ then the height of v *before* the insertion was also $h+1$ and this contradicts the initial AVL property at u. Therefore, we have to address the following two cases.

CASE (I.1): $h_L = h$ and $h_R = h-1$. This means that the inserted node is in the left subtree of v. In this case, if we rotate v, the result would be balanced. Moreover, the height of u is $h+1$.



CASE (I.1)



CASE (I.2)

Figure 10: CASE (I.1): `rotate`(v), CASE (I.2): `rotate`$^2(w)$.

CASE (I.2): $h_L = h-1$ and $h_R = h$. This means the inserted node is in the right subtree of v. In this case let us expand the subtree D and let w be its root. The two children of w will have heights of $h-1$ and $h-1-\delta$ ($\delta = 0, 1$). It turns out that it does not matter which of these is the left child (despite the apparent assymetry of the situation). If we double rotate w (*i.e.*, `rotate`(w), `rotate`(w)), the result is a balanced tree rooted at w of height $h+1$.

In both cases (I.1) and (I.2), the resulting subtree has height $h+1$. Since this was height before the insertion, there are no unbalanced nodes further up the path to the root. Thus the insertion algorithm terminates with at most two rotations.

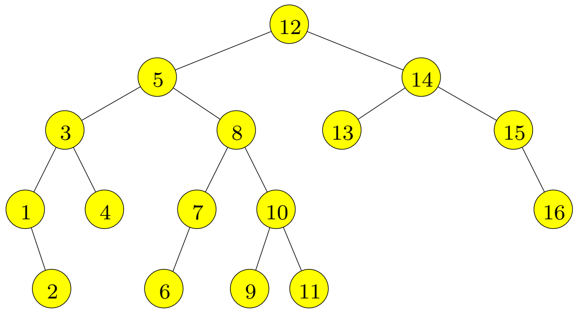


Figure 11: An AVL tree

For example, suppose we begin with the AVL tree in Figure 11, and we insert the key 9.5. The resulting transformations is shown in Figure 12.

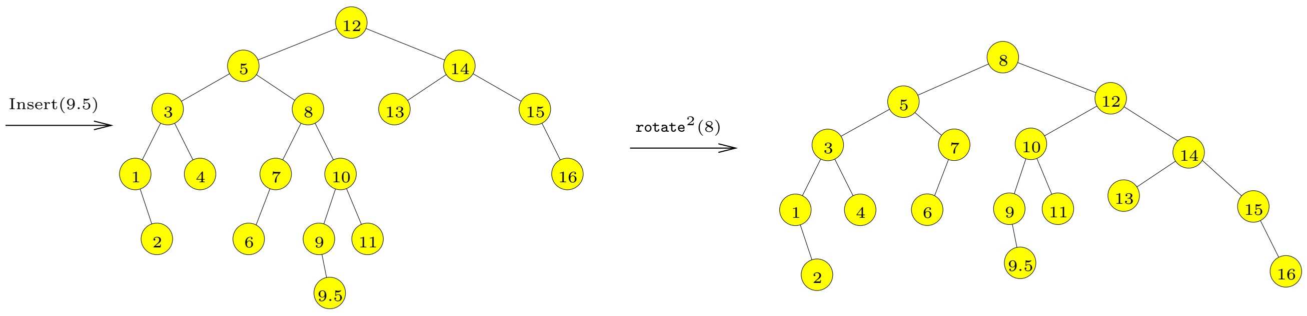


Figure 12: Inserting 9.5 into an AVL tree

Deletion Rebalancing. Suppose the imbalance in figure 9 comes from a deletion. The previous heights of u, v, v' must have been

$$h+2, h+1, h$$

and the deleted node x must be in the subtree rooted at v'. We now have three cases to consider:

CASE (D.1): $h_L = h$ and $h_R = h-1$. This is like case (I.1) and treated in the same way, namely by performing a single rotation at v. Now u is replaced by v after this rotation, and the new height of v is $h+1$. Now u is AVL balanced. However, since the original height is $h+2$, there may be unbalanced node further up the root path. Thus, this is a non-terminal case (i.e., we have to continue checking for balance further up the root path).

CASE (D.2): $h_L = h-1$ and $h_R = h$. This is like case (I.2) and treated the same way, by performing a double rotation at w. Again, this is a non-terminal case.

CASE (D.3): $h_L = h_R = h$. This case is new, and appears in Figure 13. We simply rotate at v. We check that v is balanced and has height $h+2$. Since v is in the place of u which has height $h+2$ originally, we can safely terminate the rebalancing process.

This completes the description the insertion and deletion algorithms for AVL trees. In illustration, suppose we delete key 13 from Figure 11. After deleting 13, the node 14 is unbalanced. This is restored by

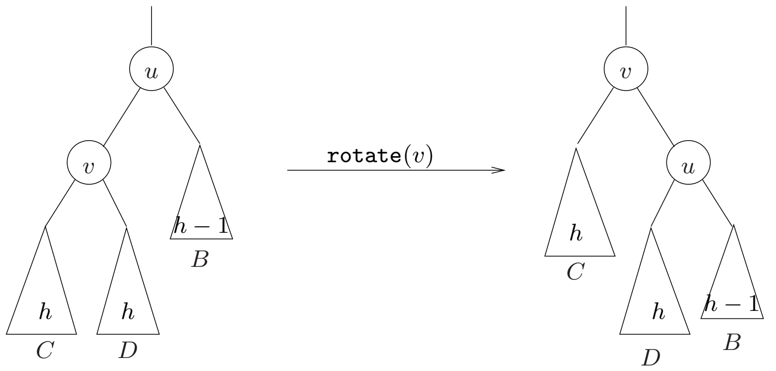


Figure 13: CASE (D.3): `rotate`(v)

a single rotation at 15. Now, the root containing 12 is unbalanced. Another single rotation at 5 will restore balance. The result is shown in Figure 14.

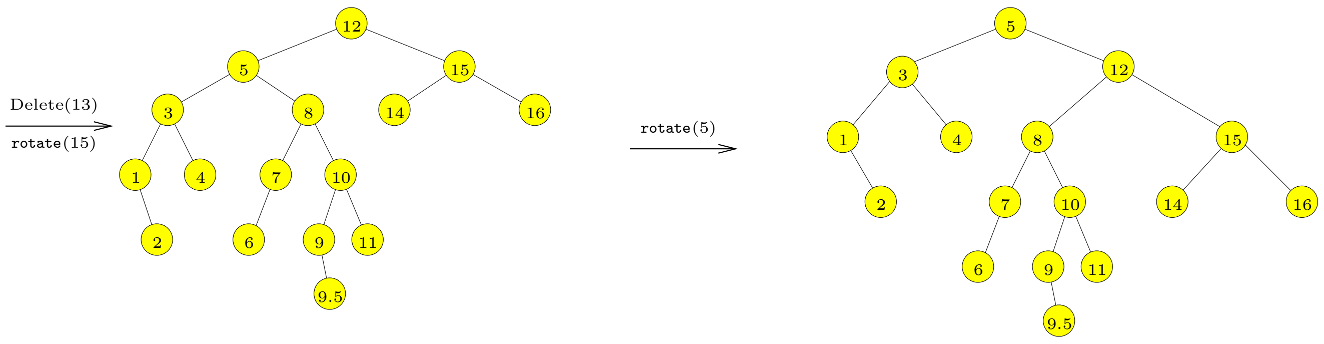


Figure 14: Inserting 13 from an AVL tree

Both insertion and deletion take $O(\log n)$ time. In case of deletion, we may have to do $O(\log n)$ rotations but a single or double rotation suffices for insertion.

Relaxed Balancing. Larsen [4] shows that we can decouple the rebalancing of AVL trees from the updating of the maintained set. In the semidynamic case, the number of rebalancing operations is constant in an amortized sense (amortization is treated in Chapter 5).

Exercises

Exercise 4.1: Give an algorithm to check if a binary search tree T is really an AVL tree. Your algorithm should take time $O(|T|)$. HINT: Use one of the tree-traversal methods as a shell for this algorithm. ♢

Exercise 4.2: What is the minimum number of nodes in an AVL tree of height 10? ♢

Exercise 4.3: My pocket calculator tells me that $\log_\phi 100 = 9.5699\cdots$. What does this tell you about the height of an AVL tree with 100 nodes? ♢

Exercise 4.4: Draw an AVL T with minimum number of nodes such that the following is true: there is a node x in T such that if you delete this node, the AVL rebalancing will require two "X-rotations". By "X-rotation" we mean either a "single rotation" or a "double rotation". Draw T and the node x. ♢

Exercise 4.5: Consider the height range for AVL trees with n nodes.
(a) What is the range for $n = 15$? $n = 20$ nodes?
(b) Is it true that there are arbitrarily large n such that AVL trees with n nodes has a unique height? ♢

Exercise 4.6: Draw the AVL trees after you insert each of the following keys into an initially empty tree: $1, 2, 3, 4, 5, 6, 7, 8, 9$ and then $19, 18, 17, 16, 15, 14, 13, 12, 11$. ♢

Exercise 4.7: Insert into an initially empty AVL tree the following sequence of keys: $1, 2, 3, \ldots, 14, 15$.
(a) Draw the trees at the end of each insertion as well as after each rotation or double-rotation. [View double-rotation as an indivisible operation].
(b) Prove the following: if we continue in this manner, we will have a complete binary tree at the end of inserting key $2^n - 1$ for all $n \geq 1$. ♢

Exercise 4.8: Starting with an empty tree, insert the following keys in the given order: $13, 18, 19, 12, 17, 14, 15, 16$. Now delete 18. Show the tree after each insertion and deletion. If there are rotations, show the tree just after the rotation. ♢

Exercise 4.9: Improve the lower bound $\mu(h) \geq \phi^h$ by taking into consideration the effects of "+1" in the recurrence $\mu(h) = 1 + \mu(h-1) + \mu(h-2)$.
(a) Show that $\mu(h) \geq F(h-1) + \phi^h$ where $F(h)$ is the h-th Fibonacci number. Recall that $F(h) = h$ for $h = 0, 1$ and $F(h) = F(h-1) + F(h-2)$ for $h \geq 2$.
(b) Further improve (a). ♢

Exercise 4.10: Relationship between Fibonacci numbers and $\mu(h)$.
(a) Recall the well-known exact formulas for Fibonacci numbers in terms of $\phi = (1 + \sqrt{5})/2$ and $\widetilde{\phi} = (\sqrt{5} - 1)/2$. Give an exact solution for $\mu(h)$ in these terms.
(c) Using your formula in (b), bound the error when we use the approximation $\mu(h) \simeq \phi^h$. ♢

Exercise 4.11: Allocating one bit per AVL node is sufficient if we exploit the fact that leaf nodes are always balanced allow their bits to be used by the internal nodes. Work out the details for how to do this. ♢

Exercise 4.12: It is even possible to allocate no bits to the nodes of a binary search tree. The idea is to exploit the fact that in implementations of AVL trees, the space allocated to each node is constant. In particular, the leaves have two null pointers which are basically unused space. We can use this space to store balance information for the internal nodes. Figure out an AVL-like balance scheme that uses no extra storage bits. ♢

Exercise 4.13: TRUE or FALSE: In CASE (D.3) of AVL deletion, we performed a single rotation at node v. This is analogous to CASE (D.1). Could we have also have performed a double rotation at w, in analogy to CASE (D.2)? ♢

Exercise 4.14: Relaxed AVL Trees
Let us define **AVL(2) balance condition** to mean that at each node u in the binary tree, $|balance(u)| \leq 2$.
(a) Derive an upper bound on the height of a AVL(2) tree on n nodes.
(b) Give an insertion algorithm that preserves AVL(2) trees. Try to follow the original AVL insertion as much as possible; but point out diferences from the original insertion.
(c) Give the deletion algorithm for AVL(2) trees. ♢

Exercise 4.15: To implement we reserve 2 bits of storage per node to represent the balance information. This is a slight waste because we only use 3 of the four possible values that the 2 bits can represent. Consider the family of "biased-AVL trees" in which the balance of each node is one of the values $b = -1, 0, 1, 2$.
(a) In analogy to AVL trees, define $\mu(h)$ for biased-AVL trees. Give the general recurrence formula and conclude that such trees form a balanced family.
(b) Is it possible to give an $O(\log n)$ time insertion algorithm for biased-AVL trees? What can be achieved? ♢

Exercise 4.16: The AVL insertion algorithm makes two passes over its search path: the first pass is from the root down to a leaf, the second pass goes in the reverse direction. Consider the following idea for a "one-pass algorithm" for AVL insertion: during the first pass, before we visit a node u, we would like to ensure that (1) its height is less than or equal to the height of its sibling. Moreover, (2) if the height of u is equal to the height of its sibling, then we want to make sure that if the height of u is increased by 1, the tree remains AVL.

The following example illustrates the difficulty of designing such an algorithm:

Imagine an AVL tree with a path $(u_0, u_1, \ldots, u_k)$ where u_0 is the root and u_i is a child of u_{i-1}. We have 3 conditions:
(a) Let $i \geq 1$. Then u_i is a left child iff i is odd, and otherwise u_i is a right child. Thus, the path is a pure zigzag path.
(b) The height of u_i is $k - i$ (for $i = 0, \ldots, k$). Thus u_k is a leaf.
(c) Finally, the height of the sibling of u_i is $h - i - 1$.

Suppose we are trying to insert a key whose search path in the AVL tree is precisely $(u_0, \ldots, u_k)$. Can we preemptively balance the AVL tree in this case? ♢

End Exercises

§5. (a, b)-Search Trees

We consider another class of trees that is important in practice, especially in database applications. These are no longer binary trees, but are parametrized by a choice of two integers,

$$2 \leq a < b. \quad (6)$$

An (a, b)**-tree** is a rooted, ordered tree with the following requirements:

- DEPTH BOUND: All leaves are at the same depth.

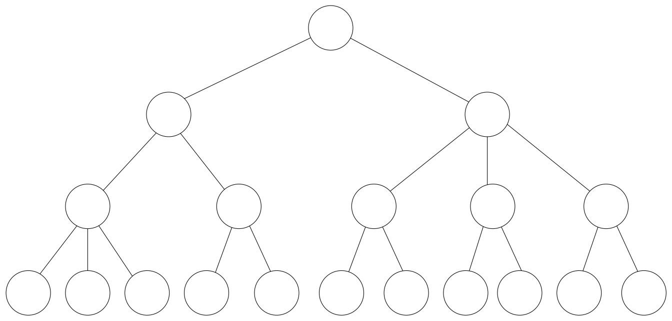


Figure 15: A $(2,3)$-tree.

- BRANCHING BOUND: Let m be the number of children of an internal node u. In general, we have the bounds
$$a \le m \le b. \tag{7}$$
The root is an exception, with the bound $2 \le m \le b$.

To see the intuition behind these conditions, compare with binary trees. In binary trees, the leaves do not have to be at the same depth. To re-introduce some flexibility into trees where leaves have the same depth, we allow the number of children of an internal node to vary over a larger range $[a,b]$. Moreover, in order to ensure logarithmic height, we require $a \ge 2$. This means that if there n leaves, the height is at most $\log_a(n) + \mathcal{O}(1)$. Therefore, (a,b)-trees forms a balanced family of trees.

The definition of (a,b)-trees imposes purely structural requirements. Figure 15 illustrates an (a,b)-tree for $(a,b) = (2,3)$. To use (a,b)-trees as a search structure, we need to give additional requirements, in particular how keys are stored in these trees. Before giving the definition, we can build some intuitions by studying an example of such a search tree in Figure 16. The 14 items stored in this tree are all at the leaves, with the keys $2,4,5,\ldots,23,25,27$. As usual, we do not display the associated data in items. The keys in the internal nodes do not correspond to items.

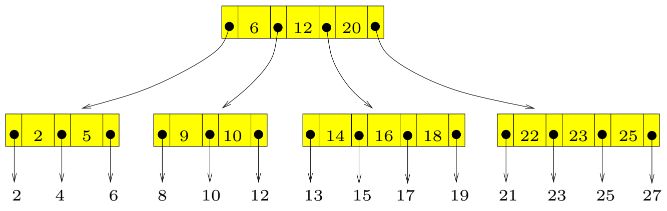


Figure 16: A (3,4)-search tree on 14 items

.

Recall that an item is a (`Key`, `Data`) pair. We define an (a,b)**-search tree** to be an (a,b)-tree whose nodes are organized as follows:

- LEAF: Each leaf stores a sequence of items, sorted by their keys. Hence we represent a leaf u with m items as the sequence,
$$u = (k_1, d_1, k_2, d_2, \ldots, k_m, d_m) \tag{8}$$

where (k_i, d_i) is the ith smallest item. See Figure 17(i). In practice, d_i might only be a pointer to the actual location of the data. If leaf u is not the root, then $a' \le m \le b'$ for some $1 \le a' \le b'$. Thus (a', b') is an additional pair of parameters. There are two canonical choices for a', b'. The simplest is $a' = b' = 1$. This means each leaf stores exactly one item. Our examples (e.g., Figure 16) use this assumption because of its simplicity, and because this choice does not affect our main algorithms. Another canonical and perhaps more realistic choice is

$$a' = a, \quad b' = b. \tag{9}$$

In any case, we must allow an exception when u is the root. We allow the root to have between 0 and $2b' - 1$ items.

- INTERNAL NODE: Each internal node with m children stores an alternating sequence of keys and pointers (node references), in the form:

  $$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{10}$$

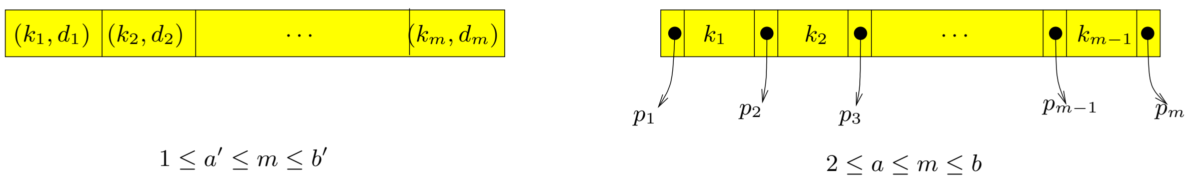
  where p_i is a pointer (or reference) to the i-th child of the current node. Note that the number of keys in this sequence is one less than the number m of children. See Figure 17(ii). The keys are sorted so that

  $$k_1 < k_1 < \cdots < k_{m-1}.$$

  For $i = 1, \ldots, m$, each key k in the i-th subtree of u satisfies

  $$k_{i-1} \le k < k_i, \tag{11}$$

  with the convention that $k_0 = -\infty < k_i < k_m = +\infty$. Note that this is just a generalization of the binary search tree property in (1).



(i) Leaf Node Organization

(ii) Internal Node Organization

Figure 17: Organization of nodes in (a, b)-search trees

Thus, an (a, b)-search tree is just a (a, b)-tree that has been organized into a search tree. As usual, we assume that the set of items in an (a, b)-search tree has unique keys. But as seen in Figure 16, the keys in internal nodes may be the same as keys in the leaves.

Another (a, b)-search tree is shown in Figure 18, for the case $(a, b) = (2, 3)$. In contrast to Figure 16, here we use a slightly more standard convention of representing the pointers as tree edges.

Special Cases of (a, b)-Search Trees. The earliest and simplest (a, b)-search trees correspond to the case $(a, b) = (2, 3)$. These are called **2-3 trees** and were introduced by Hopcroft (1970). By choosing

$$b = 2a - 1 \tag{12}$$

(for any $a \ge 2$), we obtain the generalization of $(2, 3)$-trees called **B-trees**. These were introduced by McCreight and Bayer [2]. When $(a, b) = (2, 4)$, the trees have been studied by Bayer (1972) as **symmetric binary B-trees** and by Guibas and Sedgewick as **2-3-4 trees**. Another variant of 2-3-4 trees is **red-black**

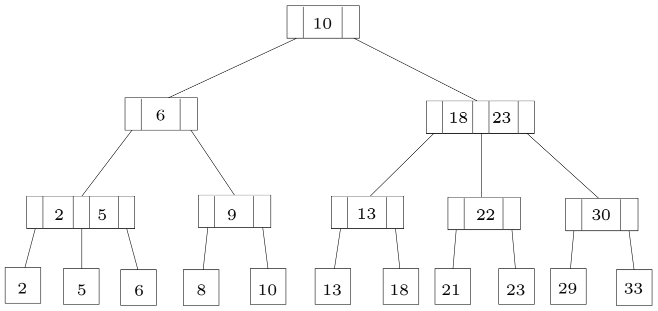


Figure 18: A $(2, 3)$-search tree.

trees. The latter can be viewed as an efficient way to implement 2-3-4 trees, by embedding them in binary search trees. But the price of this efficiency is complicated algorithms for insertion and deletion. Thus it is clear that the concept of (a, b)-search trees serves to unify a variety of search trees. The terminology of (a, b)-trees was used by Mehlhorn [6].

The B-tree relationship (12) is optimal in a certain[6] sense. Nevertheless, there are other benefits in allowing more general relationships between a and b. E.g., the amortized complexity of (a, b)-search trees algorithms can improve [3].

Searching. The organization of an (a, b)-search tree supports an obvious lookup algorithm that is a generalization of binary search. Namely, to do `lookUp(Key` k`)`, we begin with the root as the current node. In general, let u be the current node.

- Base Case: suppose u is a leaf node given by (8). If k occurs in u as k_i (for some $i = 1, \ldots, m$), then we return the associated data d_i. Otherwise, we return the null value, signifying search failure.
- Inductive Case: suppose u is an internal node given by (10). Then we find the p_i such that $k_{i-1} \le k < k_i$ (with $k_0 = -\infty, k_m = \infty$). Set p_i as the new current node and continue.

We briefly discuss alternative organizations within the nodes. The running time of the `lookUp` algorithm is $O(hb)$ where h is the height of the (a, b)-tree, and we spend $O(b)$ time at each node. If an (a, b)-tree has n leaves we conclude that

$$\lceil \log_b n \rceil \le h - 1 \le \lceil \log_a n \rceil .$$

Note that b determine the lower bound and a determine the upper bound on h. Our design goal is to maximize a for speed, and to minimize b/a for space efficiency (see below). Typically b/a is bounded by a small constant close to 2, as in B-trees.

Since the size of nodes in a (a, b)-tree is not a small constant, the organization of the data (10) for internal nodes, and (8) for leaves, can be an issue. These lists can be stored as an array, a singly- or doubly-linked list, or as a balanced search tree. These have their usual trade-offs. With an array or balanced search tree at each node, the time spent at a node improves from $O(b)$ to $O(\log b)$. In practice, b is a medium size constant (say, $b < 1000$) and setting a balanced search tree takes up extra space and further reduce the value of b. So, to maximize the value of b, a practical compromise is to simply store the list as an array in each node. This achieves $O(\lg b)$ search time but each insertion and deletion in that node requires $O(b)$ time. Indeed, when we take into account the effects of secondary memory, the time for searching within a node is negligible

[6]I.e., assuming a certain type of split-merge inequality, which we will discuss below.

compared to the time accessing each node. This argues that the overriding goal should be to maximize b and a.

Exogenous and Endogenous Search Structures. Search trees store items. But where these items are stored constitute a major difference between (a,b)-search trees and the binary search trees which we have presented. Items in (a,b)-search trees are stored in the leaves only, while in binary search trees, items are stored in internal nodes as well. Tarjan [8, p. 9] calls a search structure **exogenous** if it stores items in leaves only; otherwise it is **endogenous**.

The keys in the internal nodes of (a,b)-search trees are used purely for searching: they are not associated with any data. In our description of binary search trees (or their balanced versions such as AVL trees), we never explicitly discuss the data that are associated with keys. So how do we know that these data structures are endogenous? We deduce it from the observation that, in looking up a key k in a binary search tree, if k is found in an internal node u, we stop the search and return u. Implicitly, it means we have found the item with key k (effectively, the item is stored in u). For (a,b)-search tree, we cannot stop at any internal node, but must proceed until we reach a leaf before we can conclude that an item with key k is, or is not, stored in the search tree. It is possible to modify binary search trees so that they become exogenous (Exercise).

There is another important consequence of this dual role of keys in (a,b)-search trees. The keys in the internal nodes need not be the keys of items that are stored in the leaves. This is seen in Figure 18 where the key 9 in an internal node does not correspond to any actual item in the tree. On the other hand, the key 13 appears in the leaves (as an item) as well as in an internal node.

Database Application. One reason for treating (a,b)-trees as exogenous search structures comes from its applications in databases. In database terminology, (a,b)-search tree constitute an **index** over the set of items in its leaves. A given set of items can have more than one index built over it. If that is the case, at most one of the index can actually store the original data in the leaves. All the other indices must be contented to point to the original data, i.e., the d_i in (8) associated with key k_i is not the data itself, but a reference/pointer to the data stored elsewhere. Imagine a employee database where items are employee records. We may wish to create one index based on social security numbers, and another index based on last names, and yet another based on address. We chose these values (social security number, last name, address) for indexing because most searches in such a data base is presumably based on these values. It seems to make less sense to build an index based on age or salary, although we could.

Disk I/O Considerations: How to choose the parameter b. There is yet another reason for preferring exogenous structures: In databases, the number of items is very large and these are stored in disk memory. If there are n items, then we need at least n/b' internal nodes. This many internal nodes implies that the nodes of the (a,b)-trees is also stored in disk memory. Therefore, while searching through the (a,b)-tree, each node we visit must be brought into the main memory from disk. The I/O speed for transferring data between main memory and disk is relatively slow, compared to CPU speeds. Moreover, disk transfer at the lowest level of a computer organization takes place in fixed size blocks. E.g., in UNIX, block sizes are traditionally 512 bytes. To minimize the number of disk accesses, we want to pack as many keys into each node as possible. So the size for a node must match the size of computer blocks. Thus the parameter b of (a,b)-trees is chosen to be the largest value so that a node has this $b$lock size. Below, we discuss constraints on how the parameter a is chosen.

The Standard Split and Merge Inequalities for (a,b)-trees. To support efficient insertion and deletion algorithms, the parameters a,b must satisfy an additional inequality in addition to (6). This inequality, which we now derive, comes from two low-level operations on (a, b)-search tree. These **split** and **merge** operations are called as subroutines by the insertion and deletion algorithms (respectively). There is actually a family of such inequalities, but we first derive the simplest one ("the standard inequality").

During insertion, a node that previously had b children may acquire a new child. Such a node violates the requirements of an (a, b)-tree, so an obvious response is to **split** it into two nodes with $\lfloor (b+1)/2 \rfloor$ and $\lceil (b+1)/2 \rceil$ children, respectively. In order that the result is an (a, b)-tree, we require the following split inequality:

$$a \le \left\lfloor \frac{b+1}{2} \right\rfloor. \tag{13}$$

During deletion, we may remove a child from a node that has a children. The resulting node with $a-1$ children violates the requirements of an (a, b)-tree, so we may borrow a child from one of its **siblings** (there may be one or two such siblings), provided the sibling has more than a children. If this proves impossible, we are forced to **merge** a node with $a-1$ children with a node with a children. The resulting node has $2a-1$ children, and to satisfy the branching factor bound of (a, b)-trees, we have $2a-1 \le b$. Thus we require the following merge inequality:

$$a \le \frac{b+1}{2}. \tag{14}$$

Clearly (13) implies (14). However, since a and b are integers, the reverse implication also holds! Thus we normally demand (13) or (14). The smallest choices of these parameters under the inequalities and also (6) is $(a, b) = (2, 3)$, which has been mentioned above. The case of equality in (13) and (14) gives us $b = 2a - 1$, which leads to precisely the B-trees. Sometimes, the condition $b = 2a$ is used to define B-trees; this behaves better in an amortized sense (see [6, Chap. III.5.3.1]).

Treatment of Root. This is a good place to understand the exceptional treatment of roots in the definition of (a, b)-trees.

(i) Normally, when we split a node u, its parent gets one extra child. But when u is the root, we create a new root with two children. Hence the standard requirement that the roots have between 2 and b children.

(ii) Normally, when we merge two siblings u and v, the parent loses a child. But when the parent is the root, the root may now have only one child. In this case, we delete the root and its sole child is now the root.

Note that (i) and (ii) are the *only* means for increasing and decreasing the height of the (a, b)-tree.

Another issue arise when the root is also a leaf. We cannot treat it like an ordinary leaf having between a' to b' items. Let the root have between a'_0, b'_0 items when it is a leaf. Initially, there may be no items in the root, so we must let $a'_0 = 0$. Also, when it exceed b'_0 items, we must split into two or more children with at least a' items. The standard literature allows the root to have 2 children and this requires $2a' \le b'_0 + 1$ (like the standard split-merge inequality). Hence we choose $b'_0 = 2a' - 1$.

Achieving 2/3 Space Utility Ratio. A node with m children is said to be **full** when $m = b$, and more generally, **(m/b)-full**. Hence, our nodes can be as small as (a/b)-full. Call the ratio $a : b$ the **space utilization ratio**. The standard inequality (14) on (a, b)-trees implies that the space utilization in such trees can never[7] be better than $\lfloor (b+1)/2 \rfloor / b$, and this can be achieved by B-trees. This ratio can be slightly larger than 1 : 2, but at most 2 : 3. Of course, in practice b is fairly large and this ratio is essentially $a : b$.

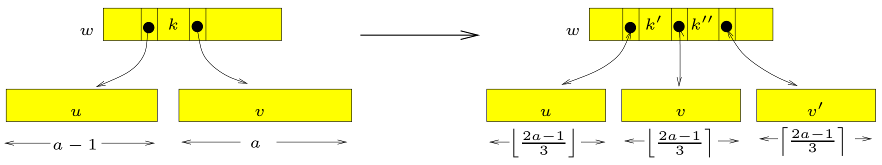
[7]The ratio $a : b$ is only an approximate measure of space utility for various reasons. First of all, it is an asymptotic limit as b grows. Furthermore, the relative sizes for keys and pointers also affect the space utilization. The ratio $a : b$ is a reasonable estimate only in case the keys and pointers have about the same size.

We now address the issue of achieving ratios that are arbitrarily close to 1. The following shows how to achieve 2/3 asymptotically.

Consider the following modified insertion: to remove an **overfull** node u with $b+1$ children, we first look at a sibling v to see if we can **donate** a child to the sibling. If v is not full, we may donate to v. Otherwise, v is full and we can take the $2b+1$ children in u and v, and divide them into 3 groups as evenly as possible. So each group has between $\lfloor (2b+1)/3 \rfloor$ and $\lceil (2b+1)/3 \rceil$ keys. More precisely, the size of the three groups are

$$\lfloor (2b+1)/3 \rfloor, \quad \lfloor (2b+1)/3 \rceil, \quad \lceil (2b+1)/3 \rceil$$

where $\lfloor (2b+1)/3 \rceil$ denotes **rounding** to the nearest integer. Nodes u and v will (respectively) have one of these groups as their children, but the third group will be children of a new node. See Figure 19.



w
k
u
v
$a-1$
a
w
k'
k''
u
v
v'
$\lfloor \frac{2a-1}{3} \rfloor$
$\lfloor \frac{2a-1}{3} \rceil$
$\lceil \frac{2a-1}{3} \rceil$

Figure 19: Generalized (2-to-3) split

We want these groups to have between a and b children. The largest of these groups has at most b children (assuming $b \geq 2$). However, for the smallest of these groups to have at least a children, we require

$$a \leq \left\lfloor \frac{2b+1}{3} \right\rfloor. \tag{15}$$

This process of merging two nodes and splitting into three nodes is called **generalized split** because it involves merging as well as splitting. Let w be the parent of u and v. Thus, w will have an extra child v' after the generalized split. If w is now overfull, we have to repeat this process at w.

Next consider a modified deletion: to remove an **underfull** node u with $a-1$ nodes, we again look at an adjacent sibling v to **borrow** a child. If v has a children, then we look at another sibling v' to borrow. If both attempts at borrowing fails, we merge the $3a-1$ children[8] the nodes u, v, v' and then split the result into two groups, as evenly as possible. Again, this is a **generalized merge** that involves a split as well. The sizes of the two groups are $\lfloor (3a-1)/2 \rfloor$ and $\lceil (3a-1)/2 \rceil$ children, respectively. Assuming

$$a \geq 3, \tag{16}$$

v and v' exists (unless u is a child of the root). This means

$$\left\lceil \frac{3a-1}{2} \right\rceil \leq b \tag{17}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (15) and (17), without changing the relationship. And thus both inequality are seen to be equivalent to

$$a \leq \frac{2b+1}{3} \tag{18}$$

As in the standard (a,b)-trees, we need to make exceptions for the root. Here, the number m of children of the root satisfies the bound $2 \leq m \leq b$. So during deletion, the second sibling v' may not exist if u is

[8]Normally, we expect v, v' to be immediate siblings of u (to the left and right of u). But if u is the eldest or youngest sibling, then we may have to look slightly farther for the second sibling.

a child of the root. In this case, we can simply merge the level 1 nodes, u and v. This merger is now the root, and it has $2a-1$ children. This suggests that we allow the root to have between a and $\max\{2a-1,b\}$ children.

If we view b as a hard constraint on the maximum number of children, then the only way to allow the root to have $\max\{2a-1,b\}$ children is to insist that $2a-1 \le b$. Of course, this constraint is just the standard split-merge inequality (14); so we are back to square one. This says we must treat the root as an exception to the upper bound of b. Indeed, one can make a strong case for treating the root differently:
(1) It is desirable to keep the root resident in memory at all times, unlike the other nodes.
(2) Allow the root to be larger than b can speed up the general search.

The smallest example of a (2/3)-full tree is where $(a,b)=(3,4)$. We have already seen a $(3,4)$-tree in Figure 16. The nodes of such trees are actually 3/4-full, not 2/3-full. But for large b, the "2/3" estimate is more reasonable.

Mechanics of Insertion and Deletion. Both insertion and deletion can be described as a repeated application of the following while-loop:

```
▷ INITIALIZATION
To insert an item $(k,d)$ or delete a key $k$, we first do a lookup on $k$.
Let $u$ be the leaf where $k$ is found.
Bring $u$ into main memory and perform the indicated operation.
Call $u$ the current node.
▷ MAIN LOOP
while $u$ is not overfull or underfull, do:
    If $u$ is root, handle as a special case and terminate.
    Bring the parent $v$ of $u$ into main memory.
    Depending on the case, some siblings $u_1, u_2$, etc, of $u$ may be brought into main memory as well.
    Make the necessary transformations of $u$ and its siblings and $v$ in the main memory.
    While they are in main memory, the number of children are allowed to exceed the bound $b$
    We may have created a new node $u'$ in the course of this computation.
    When all the children of $v$ are done, we write these children back into secondary memory.
    Make $v$ our new current node (denoted $u$) and repeat this loop.
Write the current node $u$ to secondary memory and terminate.
```

Let us illustrate this process. Consider how we might insert the key 24 into the $(3,4)$-search tree in Figure 16. First insert 24 into a leaf. Then we make this leaf a new child its parent node u. Now u has 5 children and is overfull. These 5 children are separated by the keys

$$u : [22, 23, \underline{24}, 25],$$

where key 24 is newly generated in u. Being overfull, u tries to donate a key to an adjacent sibling. But this sibling $v : [14, 16, 18]$ is already full. Hence we must merge nodes u and v, and then split the result into three new nodes. The merger of u and v yields

$$[14, 16, 18, \underline{20}, 22, 23, 24, 25]. \tag{19}$$

where the key 20 from the parent w of u. Thus w becomes $[6, 12]$. Then we split (19) into three nodes,

$$u : [14, 16], \quad v : [20, 22], \quad x : [24, 25]$$

where u, v are reused but x is a new node. The keys 18 and 23 are sent to the parent w, which becomes

$$w : [6, 12, 18, 23].$$

So w has 5 children. Normally, this would make w overfull. But since w is the root, and our rules allow it to have $\max\{2a-1, b\} = 5$ children. Hence the algorithm can halt.

Generalized Split-Merge Method for (a,b)-trees. Thus insertion and deletion algorithms uses the strategy of "share a key if you can" in order to avoid splitting or merging. Here, "sharing" encompasses donating or borrowing. The 2/3-space utility method is now generalized by introducing a new parameter $c \geq 1$.

When $c = 1$, we have the B-trees; when $c = 2$, we achieve the 2/3-space utilization ratio above. We could call these (a,b,c)**-trees**. They achieve a space utilization ratio of $c : c+1$ which approaches 1 as $c \to \infty$. We introduce the **generalized split-merge inequality**,

$$c + 1 \leq a \leq \frac{cb+1}{c+1}. \tag{20}$$

The lower bound on a ensures that when we merge or split a node, we can be assured of at least $a - 1 \geq c$ siblings from which to borrow (if you are short) or to donate (if you are long) keys. In case of merging, the current node has $a-1$ keys. We may assume that the other c siblings have a keys each. We can combine all these $a(c+1)-1$ keys and split them into c new nodes. This merging is valid because of the upper bound (20) on a. In case of splitting, the current node has $b+1$ keys. If you fail to donate a key, you may assume to have found $c-1$ siblings with b keys each. We combine all these $cb+1$ keys, and split them into $c+1$ new nodes. Again, the upper bound on a (20) guarantees success.

In summary, the generalized split of (a,b,c)-trees transforms c nodes into $c+1$ nodes; the generalized merges is the reverse transformation of $c+1$ nodes into c nodes. When

$$a = \frac{cb+1}{c+1}, \tag{21}$$

we may call the (a,b,c)-tree a **generalized B-tree**.

REMARK: An (a,b,c)-trees is structurally indistinguishable from an (a,b)-tree. For any (a,b) parameter, we can compute the smallest c such that this could be a (a,b,c)-tree.

Pre-emptive or 1-Pass Algorithms. The above algorithm uses 2-passes through nodes from the root to the leaf: one pass to go down the tree and another pass to go up the tree. There is a 1-pass versions of these algorithms. Such algorithms could potentially be twice as fast as the corresponding 2-pass algorithms since they could reduce the bottleneck disk I/O. The basic idea is to pre-emptively split (in case of insertion) or pre-emptively merge (in case of deletion).

More precisely, during insertion, if we find that the current node is already full (i.e., has b children) then it might be advisable to split u at this point. Splitting u will introduce a new child to its parent, v. We may assume that v is already in core, and by induction, v is not full. So v can accept a new child without splitting. In case of standard B-trees ($c = 1$), this involves no extra disk I/O. Such pre-emptive splits might turn out to be unnecessary, but this extra cost is negligible when the work is done in-core. Unfortunately, this is not true of generalized B-trees since a split requires looking at siblings which must be brought in from the disk. Further studies would be needed.

For deletion, we can again do a pre-emptive merge when the current node u has a children. Unfortunately, even for standard B-trees, this may involve extra disk I/O because we need to try to donate to a sibling first.

In modern computers, main memory is large and storing the entire path of nodes in the 2-pass algorithm seems to impose no burden. In this situation, the 1-pass algorithms may actually be slower than 2-pass algorithms.

Background on Space Utilization. Using the $a : b$ measure, we see that standard B-trees have about 50% space utilization. Yao showed that in a random insertion model, the utilization is about $\lg 2 \sim 0.69\%$. (see [6]). This was the beginning of a technique called "fringe analysis" which Yao [9] introduced in 1974. Nakamura and Mizoguchi [7] independently discovered the analysis, and Knuth used similar ideas in 1973 (see the survey of [1]).

Now consider the space utilization ratio of generalized B-trees. Under (21), we see that the ratio $a : b$ is $\frac{cb+1}{(c+1)} : b$, and is greater than $c : c + 1$. In case $c = 2$, our space utilization that is close to $\lg 2$. Unlike fringe analysis, we guarantee this utilization in the worst case. It seems that most of the benefits of (a, b, c)-trees are achieved with $c = 2$ or $c = 3$.

Exercises

Exercise 5.1: What is the the best ratio achievable under (14)? Under (18)? ♢

Exercise 5.2: Give a more detailed analysis of space utilization based on parameters for (A) a key value, (B) a pointer to a node, (C) either a pointer to an item (in the exogenous case) or the data itself (in the endogenous case). Suppose we need k bytes to store a key value, p bytes for a pointer to a node, and d bytes for a pointer to an item or for the data itself. Express the space utilization ratio in terms of the parameters

$$a, b, k, p, d$$

assuming the inequality (14). ♢

Exercise 5.3: Describe the exogenous version of binary search trees. Give the insertion and deletion algorithms. NOTE: the keys in the leaves are now viewed as surrogates for the items. Moreover, we allow the keys in the internal nodes to duplicate keys in the leaves, and it is also possible that some keys in the internal nodes correspond to no stored item. ♢

Exercise 5.4: Describe in detail the insertion and deletion algorithms in the following cases: $(2, 3)$-trees and $(2, 4)$-trees. ♢

Exercise 5.5: Suppose we want the root, if non-leaf, to have between a and b children, just like other nodes. Here is the idea: we allow the root, when it is a leaf, to have up to $a'a - 1$ items. Here, (a', b') is the usual bound on the number of items in non-root leaves. When this limit is exceeded, the root has $a'a$ items and we can split it into a leaves, each with a' items. The new root will have these a leaves as children. The beauty is that there is no exception with respect to the (a, b) bound. Discuss the issues that might arise with this design.

♢

Exercise 5.6: Let us replace the special rule for the root of an (a, b)-tree by the following: instead of requiring the root to have between 2 and b children, we can require the root to have between a and $a^2 - 1$ children. Discuss the pros and cons of this design. ♢

Exercise 5.7: We want to explore the weight balanced version of (a, b)-trees.
(a) Define such trees. Bound the heights of your weight-balanced $(a, b$-trees.
(b) Describe an insertion algorithm for your definition.
(c) Describe a deletion algorithm. ♢

End Exercises

References

[1] R. A. Baeza-Yates. Fringe analysis revisited. *ACM Computing Surveys*, 27(1):109–119, 1995.

[2] R. Bayer and McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

[3] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Inform.*, 17:157–184, 1982.

[4] K. S. Larsen. AVL Trees with relaxed balance. *J. of Computer and System Sciences*, 61:508–522, 2000.

[5] H. R. Lewis and L. Denenberg. *Data Structures and their Algorithms.* Harper Collins Publishers, New York, 1991.

[6] K. Mehlhorn. *Datastructures and Algorithms 1: Sorting and Sorting.* Springer-Verlag, Berlin, 1984.

[7] T. Nakamura and T. Mizoguchi. An analysis of storage utilization factor in block split data structuring scheme. *VLDB*, 4:489–195, 1978. Berlin, September.

[8] R. E. Tarjan. *Data Structures and Network Algorithms.* SIAM, Philadelphia, PA, 1974.

[9] A. C.-C. Yao. On random 2-3 trees. *Acta Inf.*, 9(2):159–170, 1978.