Lecture III
BALANCED SEARCH TREES

It is said[1] that there are more than a dozen Eskimo words for snow. The tree word is the computer science equivalent of snow: (a, b)*-tree, alpha-beta tree, AVL tree, B-tree, BSP tree, conjugation tree, dynamic weighted tree, finger tree, half-balanced tree, heaps, interval tree, kd-tree, quadtree, octtree, optimal binary search tree, randomized search tree, range tree, red-black tree, segment tree, splay tree, suffix tree, treaps, priority search tree, tries, weight-balanced tree, etc.* We do not attempt a taxonomy of these terms. This list can be considerably enlarged (e.g., there are subspecies of B-trees called B^+- and B^*-trees, etc). Our list of tree words here is restricted to those used as search data structures. Such trees have some structural constraints and the presence of "search keys" in nodes. But trees also arise in specific applications (e.g., Huffman tree, DFS tree, alpha-beta tree) and these are even more diverse.

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 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 and we will discuss the family called (a, b)**-trees**. 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. Keyed Search Structures

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 an object which we will call an **item**. We will be informal about how we manipulate nodes – they will variously look like pointers[2] as in the the programming

[1] Some call this a myth, raising issues about how to count words, which Eskimo dialect is meant, suggesting that perhaps English may have more snow words, etc. No one would deny such difficulties. Similar issues also arise in the case "tree words" in computer science.

[2] Lewis and Denenberg [5] introduce a programming notion called **locatives** which provide suitable referencing and dereferencing semantics for location variables.

language `C/C++`, or references in `Java`. 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 an item or node, we write u.`Key` and u.`Data` for the key and data associated with u.

Another important concept is that of **iterators**. In search queries, we sometimes need to return a set of items. We basically have to return a linked list of nodes containing all the items in the set (in some order). The concept of an iterator captures this in an abstract way: We view an iterator as a node u which has an associated field denoted u.`next`. This field refers to another iterator or to 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 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. 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.

Search structures such as binary trees can support any subset of the following operations, organized into four groups (I)-(IV) below.

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

The meaning of these operations are fairly intuitive. We will briefly explain them. Let S, S' be search structures, K be a key and u a node. Each of the operations are invoked from some structure variable S: thus, S.`make`() will initialize the structure S, and S.`max`() returns the maximum value in S. But we usually 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 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., predecessor `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) $\to S'$ 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`.
- **Fully mergeable dictionary**: `lookUp`, `insert`, `delete`, `merge`, `split`.

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[3] 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)'.

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. ◊

End Exercises

§3. Binary Search Trees

We show that binary search trees can support all the above operations. Our approach is somewhat unconventional, because we want to reduce all these operations to the single operation of "rotation".

The basic properties of binary trees is described in the Appendix of Lecture I. A binary tree T is called **binary search tree** (BST) if we store a key u.`Key` at each node u, subject to the **binary search tree property**:

$$u_L.Key \leq u.Key \leq u_R.Key. \tag{1}$$

where u_L and u_R is a **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.`left` and u.`right`.

Before proceeding, it is important for the student to remember a meta-principle about proving properties of binary trees: just as binary trees are best defined by structural induction, *most basic properties about binary trees are best proved by induction on the structure of the tree.*

[3] Strictly speaking, the distinction between ADT and concrete data types is a matter of degree.

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. The algorithms in the former case is invariably simpler than the latter. *In the following, we will assume distinct keys unless otherwise noted.* A possible simplication of binary search trees might arise by replacing (1) with

$$u_L.Key < u.Key \leq u_R.Key. \tag{2}$$

We call this the **strong duplicate key condition**.

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.

What if we allow duplicate keys? In this case, on finding a node u whose key is equal to K, we continue recursive insertion into the subtree rooted at $u.\mathtt{right}$. Note that this will preserve the strong duplicate key condition.

Rotation. This is not a listed operation in §2. It is an equivalence operation. By itself, rotation does not appear to do anything useful. But it is actually the basis for many of our operations.

The operation $\mathtt{rotate}(u)$ 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 $\mathtt{rotate}(u)$ amounts to the following transformation of T (see figure 1).

In $\mathtt{rotate}(u)$, 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 $\mathtt{rotate}(u)$, the former parent of v (not shown) will now have u instead of v as a child. Clearly the inverse of $\mathtt{rotate}(u)$ is $\mathtt{rotate}(v)$. 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 $\mathtt{rotate}(u)$ followed by $\mathtt{rotate}(v)$ is the identity[4] operation, as illustrated in figure 1.

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

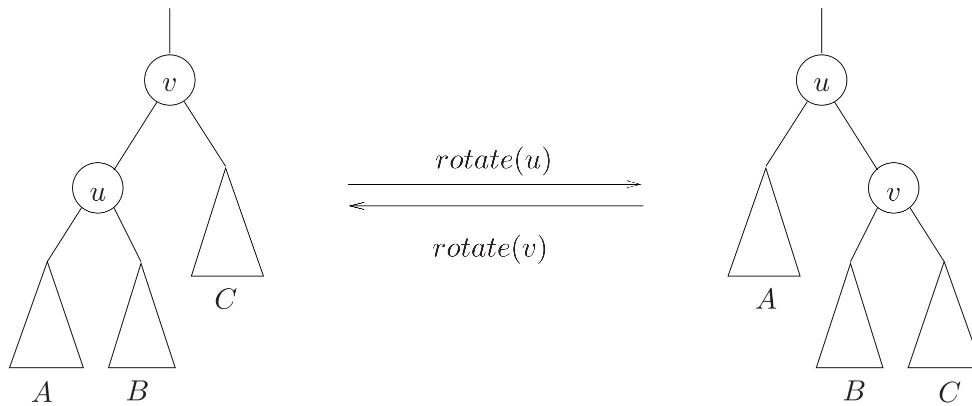


Figure 1: Rotation at u and its inverse.

Recall that two search structures are equivalent if they contain the same set of items. Clearly, rotation is an equivalence transformation. Does rotation preserve the strong duplicate key property? It almost does: let u be a child of v. If $u.Key \neq v.Key$ or u is a left child of v, then `rotate`(u) *preserves the strong duplicate key condition.* In proof, we must make sure that no node y has any left descendent z such that $z.Key = y.Key$. Looking at Figure 1, we see that no new (y, z) pairs that are created are created by `rotate`(u). Similarly, no new pairs are created by the inverse operation `rotate`(v).

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.

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.`parent` = Nil, and u is type left iff u.`parent.left` $= u$. Clearly, `rotate`(u) is sensitive to the type of u. In particular, if u is a root then `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 $\overline{\texttt{right}} = \texttt{left}$.

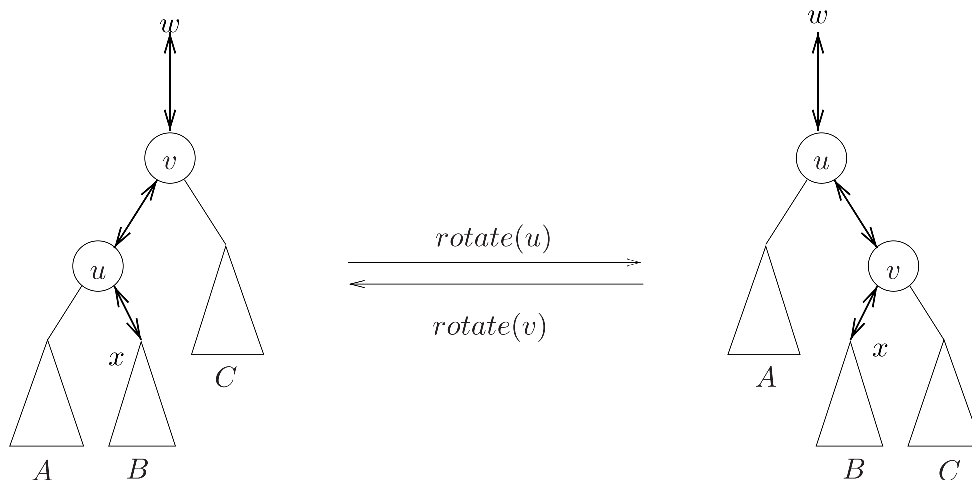


Figure 2: Links that must be fixed in `rotate`(u).

We are ready to discuss the function `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 Nil. Thus we have potentially three child-parent pairs:

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

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

$$(x, v), (v, u), (u, w). \tag{4}$$

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.

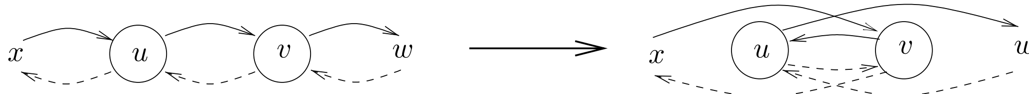


Figure 3: Simplified view of $\texttt{rotate}(u)$ as fixing a doubly-linked list (x, u, v, w).

Such re-assignments must be done in the correct order. It is best to see what is needed by thinking of (3) as a doubly-linked list (x, u, v, w) which must be converted into the doubly-linked list (x, v, u, w) in (4). 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[5] the code:

```
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 $u.\texttt{next}$ pointer may be identified with $u.\texttt{parent}$ pointer. However, $u.\texttt{prev}$ would be $u.T$ where $T \in \{\texttt{left}, \texttt{right}\}$ is the type of x. Moreover, $v.\texttt{prev}$ is $v.\overline{T}$. Also $w.\texttt{prev}$ 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.

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 $u.\texttt{parent}$. *This is our default assumption for binary*

[5] 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.

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.

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.]

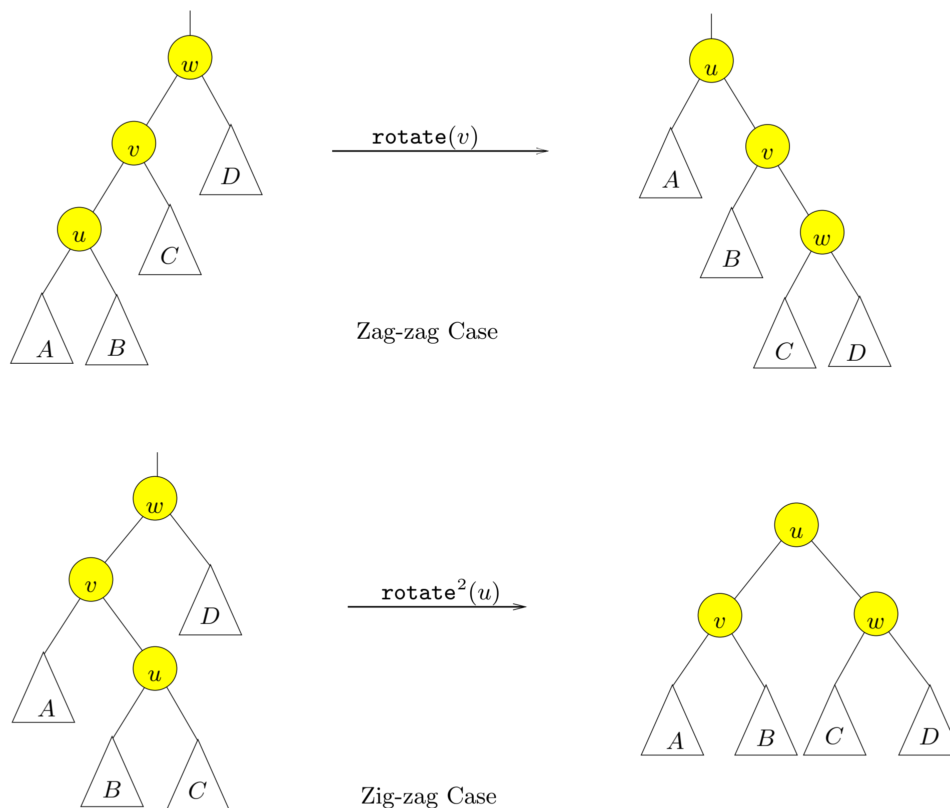


Figure 4: Two outcomes of $\texttt{rotate}^2(u)$

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.

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 $(u, \mathrm{rightpath}(u.\mathtt{left}))$. In case $u.\mathtt{left} = \mathsf{Nil}$, 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.

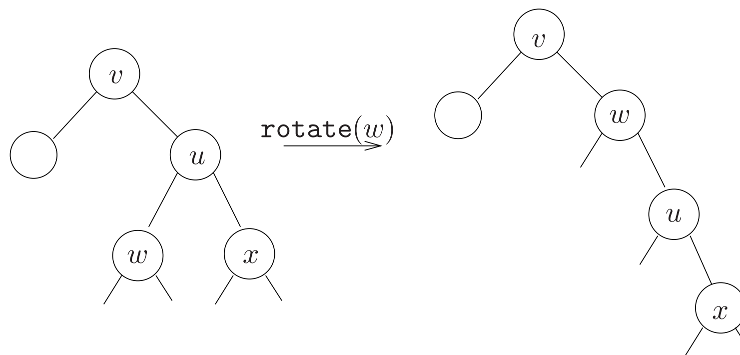


Figure 5: Reduction of the left-spine of u after $\mathtt{rotate}(u.\mathtt{left}) = \mathtt{rotate}(w)$.

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* $\mathtt{rotate}(u.\mathtt{left})$,
(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 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 ≠ 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**:
(i) If u has at most one child, apply Cut(u).
(ii) If u has two children, let v be the tip of the right (or left) spine of u. Note that v has at most one child. We move the item in v into u and delete v, as in case (i) or (ii).

Note that in case (ii), the node u is not physically removed: only the item represented by u is removed. Furthermore, *the node that is physically removed has at most one child.*

In practice, our rotation-based algorithm is undesirable and slower than this standard algorithm. The reason for introducing rotation-based approaches is its conceptual simplicity. They will also be useful for amortized algorithms later.

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.
     1.   If u.left ≠ Nil then In-order(u.left).
     2.   Visit(u).
     3.   If u.right ≠ Nil then In-order(u.right).
```

The Visit(u) subroutine is application dependent, and may be as simple as "print u.Key". For example, consider the tree in figure 6. The numbers on the nodes are not keys, but serve as identifiers.

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

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

Changing the order of these three steps in the above procedure (but always visiting the left subtree before the right subtree), we obtain two other methods of tree traversal. If we perform step 2 before steps 1 and 3, the result is called the **pre-order traversal** of the tree. 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. Using the same example, we obtain

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

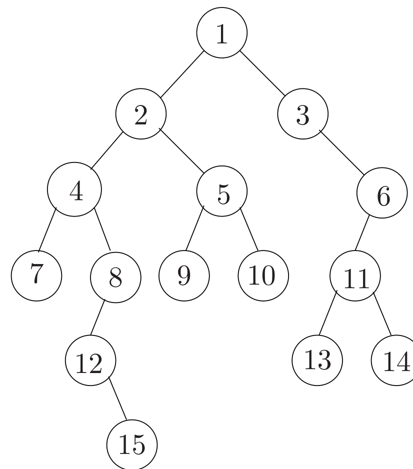


Figure 6: Binary tree.

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 $\mathtt{succ}(u)$ and $\mathtt{pred}(u)$ denotes the successor and predecessor of u. Of course, $\mathtt{succ}(u)$ (resp., $\mathtt{pred}(u)$) 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 $u.\mathtt{succ}$ and $u.\mathtt{pred}$. 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) $u.\mathtt{right} = \mathsf{Nil}$ *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 $u.\mathtt{right} = \mathsf{Nil}$ *and u is the tip of the left-spine of v, then* $\mathtt{succ}(u) = v$.
(iii) If $u.\mathtt{right} \neq \mathsf{Nil}$ *then* $\mathtt{succ}(u)$ *is the tip of the right-spine of u.*

It is easy to derive an algorithm for $\mathtt{succ}(u)$ 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`$(u) = \mathsf{Nil}$ 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 $u.\mathtt{left} = \mathsf{Nil}$. Similarly, perform right rotations at u until $u.\mathtt{right} = \mathsf{Nil}$. 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: (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 3.2: Let T be the binary search tree in figure 7.

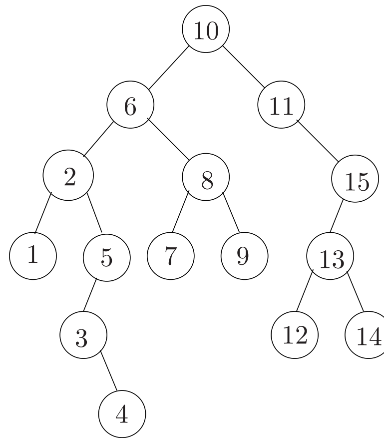


Figure 7: A binary search tree.

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

Exercise 3.3: 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 $u.\texttt{next.prev.prev} \leftarrow u.\texttt{prev}$ 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. ♢

Exercise 3.4: 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. ♢

Exercise 3.5: Open-ended: The problem of implementing $\texttt{rotate}(u)$ 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., $\texttt{next}, \texttt{prev}, \texttt{left}, \texttt{right}$) 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 $\texttt{rotate}(u)$)? Under what conditions is the transformation achievable at all (using more intermediate variables? We also want to achieve minimum time. ♢

Exercise 3.6: 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 $R_2(n)$. ♢

Exercise 3.7: 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. ♢

Exercise 3.8: 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. ♢

Exercise 3.9: 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 3.10: 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 3.11: 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 3.12: Give the rotation-based version of the successor algorithm. ♢

Exercise 3.13: 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). ♢

End Exercises

§4. Variations on Binary Trees

There is an alternative view of binary trees; let us call them **extended binary trees**. For emphasis, the original version will be called **standard binary trees**. In the extended trees, every node has 0 or 2 children; nodes with no children are called[6] **nil nodes** while the other nodes are called **non-nil nodes**. See figure 8(a) for a standard binary tree and figure 8(b) for the corresponding extended version. In this figure, we see a common convention of representing nil nodes by black squares.

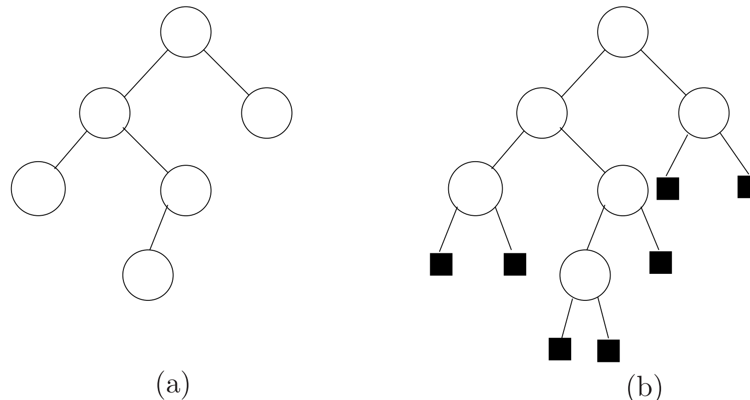


Figure 8: Binary Trees: (a) standard, (b) extended.

The bijection between extended and standard binary trees is given as follows:

1. For any extended binary tree, if we delete all its nil nodes, we obtain a standard binary tree.
2. Conversely, for any standard binary tree, if we give every leaf two nil nodes as children and for every internal node with one child, we give it one nil node as child, then we obtain a corresponding extended binary tree.

In view of this correspondence, we switch between the two viewpoints depending on which is more convenient. Generally, we avoid drawing the nil nodes since they just double the number of nodes without conveying any new information!

The terminology of "nil nodes" is easy to appreciate when we realize that in conventional realization of binary trees, we allocate two pointers to every node, regardless of whether the node has two children or not. The lack of a child is indicated by making the corresponding pointer take the Nil value. We may use the term "extended nodes" to refer to nodes in an extended tree.

We can easily extend the notion of extended binary tree to **extended binary search tree**. Here, the non-nil nodes store keys in the usual nodes but the nil nodes do not hold keys (obviously).

The concept of a "leaf" of an extended binary tree is apt to cause some confusion. We shall use use the "leaf" terminology so as to be consistent with standard binary trees. A node of an extended binary tree is called a **leaf** if it is the leaf of the corresponding standard binary tree. Alternatively, a leaf in an extended binary tree is a node with two nil nodes as children. *So a nil node is never a leaf.*

Exogenous versus Endogenous Search Structures Let us return to the use of binary trees for searching, *i.e.*, as binary search trees. There are an implicit assumptions in the use of binary search trees that

[6]A binary tree in which every node has 2 or 0 children is said to be "full". Knuth calls the nil nodes "external nodes". A path that ends in an external node is called an "external path".

is worth examining. We normally think of the data associated with a key to be stored with the key itself. This means that each node of our binary search tree actually store items (recall items is just a key-data pair). There is an alternative organization that applies to search structures in general: we assume the set of items to be searched is stored independently of the search structure (the binary search tree in this case). In the case of binary search trees, it means that we associate with each key a pointer to the associated data. Following[7] Tarjan [8], we call this an **exogenous search structure** In contrast, if the data is directly stored with the key, we call it an **endogenous search structure**. What is the relative advantage of either form? In the exogenous case, we have actually added an extra level of indirection (the pointer) which uses extra space). But on the other hand, it means that the actual data can be freely re-organized more easily and independently of the search structure. In databases, this is important because the exogenous search structure are called "indexes". Users can freely create and destroy such indexes into the stored set of items.

Internal verses External Search Structures. There is an implicit assumption that each key in our binary search tree corresponds to an item. An alternative is to associate items only to keys at the leaves of the tree. The internal keys are just used for guiding the search. For the lack of a better name, we shall call this version of search structures an **external search structures**. So the common concept of binary search trees illustrates the notion of **internal search structures**. Like exogenous search structures, internal search structures apparently uses extra space: e.g., in binary search trees, the keys in the internal nodes are possibly duplicates of the actual keys stored with the items. On the other hand, this also give us added flexibility – we can introduce new keys in the search structure which are not in the items. For instance, we may want to use use more compact keys than the actual keys in items, which may be very long. Our usual search tree algorithms are easily modified to handle external search structures.

Auxiliary Information. In some applications, additional information must be maintained at each node of the binary search tree. We already mentioned the predecessor and successor links. Another information is the the size of the subtree at a node. Some of this information is independent, while other is dependent or **derived**. Maintaining the derived information under the various operations is usually straightforward. In all our examples, the derived information is **local** in the following sense that *the derived information at a node u can only depend on the information stored in the subtree at u.* We will say that derived information is **strongly local** if it depends only on the independent information at node u, together with all the information at its children (whether derived or independent).

Implicit Keys and Parametrized Binary Search Trees. Perhaps the most interesting variation of binary search trees is when the keys used for comparisons are only implicit. The information stored at nodes allows us to make a "comparison" and decide to go left or to go right at a node but this comparison may depend on some external data beyond any explicitly stored information. We illustrate this concept in the lecture on convex hulls in Lecture V.

EXERCISES

Exercise 4.1: Describe what changes is needed in our binary search tree algorithms for the exogeneous case. ♢

Exercise 4.2: Suppose we insist that for exogenous binary search trees, each of the keys in the internal nodes really correspond to keys in stored items. Describe the necessary changes to the deletion algorithm that will ensure this property. ♢

[7] He used this classification only in the case of the linked lists data structure, but the extension is obvious.

Exercise 4.3: Consider the usual binary search trees in which we no longer assume that keys in the items are unique. State suitable conventions for what the various operations mean in this setting. E.g., `lookUp`(K) means find any item whose key is K or find all items whose keys are equal to K. Describe the corresponding algorithms. ♢

Exercise 4.4: Describe the various algorithms on binary search trees that store the size of subtree at each node. ♢

Exercise 4.5: Normally, each node u in a binary search tree maintains two fields, a key value and perhaps some balance information, denoted u.KEY and u.BALANCE, respectively. Suppose we now wish to "augment" our tree T by maintaining two additional fields called u.PRIORITY and u.MAX. Here, u.PRIORITY is an integer which the user arbitrarily associates with this node, but u.MAX is a pointer to a node v in the subtree at u such that v.PRIORITY is maximum among all the priorities in the subtree at u. (Note: it is possible that $u = v$.) Show that rotation in such augmented trees can still be performed in constant time.

♢

End Exercises

§5. AVL Trees

AVL trees is the first known example of balanced trees, and have relatively simple algorithms. 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.

In general, 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:

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

The node is **perfectly balanced** if the balance is 0. It is **AVL-balanced** if the balance is either 0 or ± 1. Thus, at each AVL node we only need to store one of three possible values. This means the space requirement for balancing information is only $\lg 3 < 1.585$ 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 useful to introduce the function $\mu(h)$ defined to be the minimum number of nodes in any AVL tree with height h. It is not hard to see by case analysis that

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

But is $\mu(0) = 0$ or 1? To resolve this, we first look at the generic situation: in general, $\mu(h)$ clearly satisfies the following recurrence:

$$\mu(h) = 1 + \mu(h-1) + \mu(h-2), \qquad (h \geq 1). \tag{5}$$

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, $\mu(2) = 1 + \mu(1) + \mu(0)$. It is also easy to see that $\mu(2) = 4$. Thus, we are forced to define $\mu(0) = 1$ so that $\mu(2) = 1 + 2 + 1 = 4$. Next, applying (5) to $h = 1$, we get $\mu(1) = 1 + \mu(0) + \mu(-1)$. Since we know that $\mu(1) = 2$, we conclude that $\mu(-1)$ must be defined to be 0.

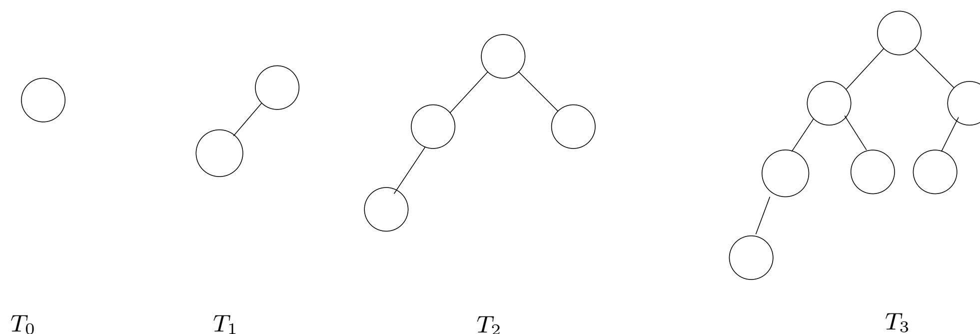


Figure 9: Smallest AVL trees of heights 0, 1, 2 and 3.

Thus we define the **height** of an empty tree as -1. See figure 9 for the smallest AVL trees of the first few values of h.

The claim that AVL trees are balanced is a consequence of

$$\mu(h) \ge C^h, \qquad (h \ge 1) \tag{6}$$

for some constant $C > 1$. This is because if an AVL tree has n nodes and height h then we see that

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

Combined with (6), we conclude that $n \ge C^h$. Taking logs, we obtain $\log_C(n) \ge h$ or $h = O(\log n)$.

Next, to establish (6) with $C = 2^{1/2} = \sqrt{2}$: from (5), we have $\mu(h) \ge 2\mu(h-2)$ for $h \ge 1$. Then it is easy to see by induction that $\mu(h) \ge 2^{h/2}$ for all $h \ge 1$. 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:

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

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

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

So any AVL tree with n nodes and height h must satisfy the inequality $\phi^h \le n$ or $h \le (\lg n)/(\lg \phi)$. So 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 we might sharpen this estimate.

The basic insertion and deletion 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 10.

Figure 10: 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.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.*, $\texttt{rotate}(w), \texttt{rotate}(w)$), 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.

Deletion Rebalancing. Suppose the imbalance in figure 10 comes from a deletion. The previous heights of u, v, v' must have been

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

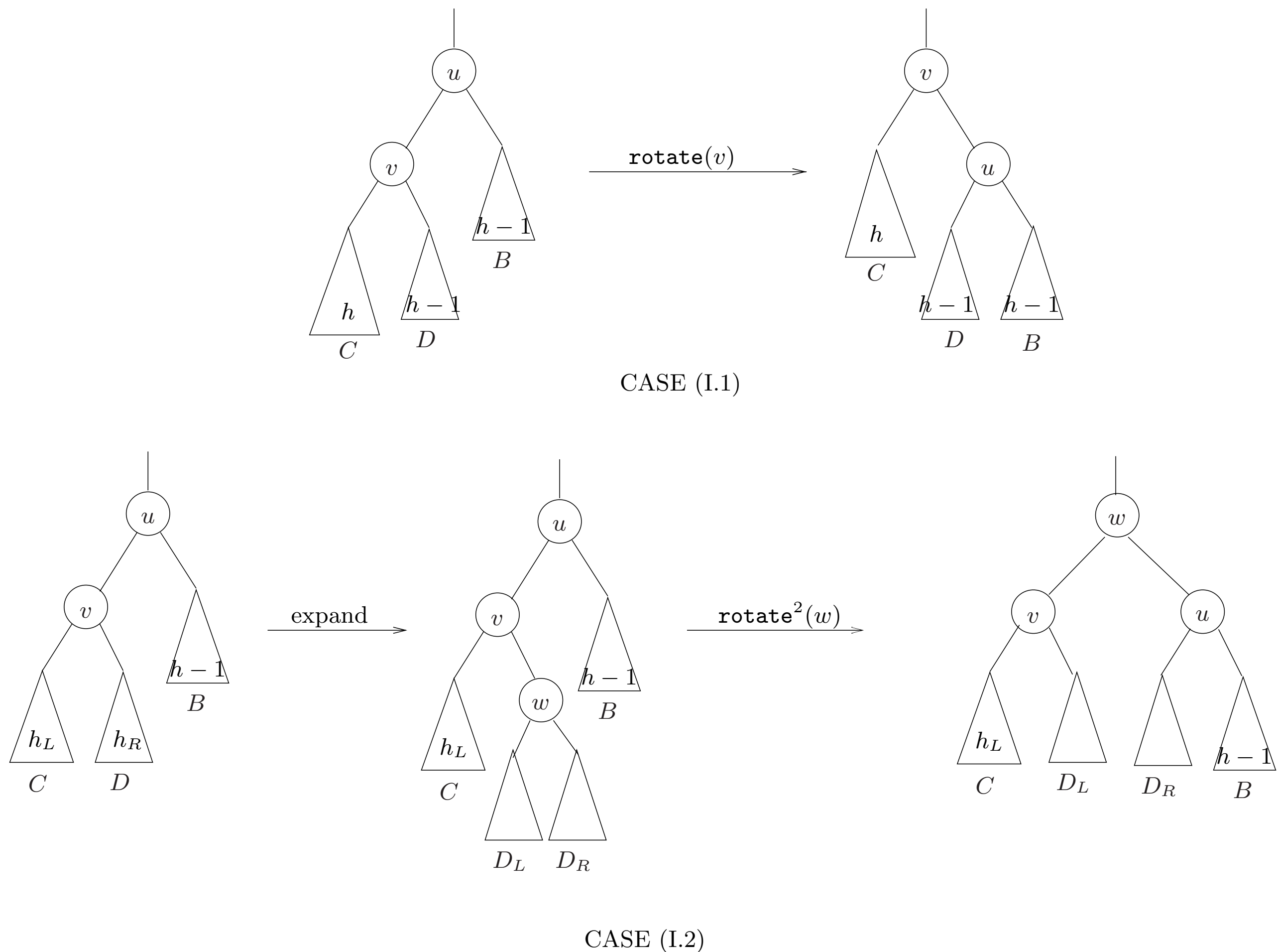


Figure 11: CASE (I.1): `rotate`(v), CASE (I.2): `rotate`$^2(w)$.

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 12. 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. Both algorithms takes $O(\log n)$ time. In the deletion case, we may have to do $O(\log n)$ rotations but in the insertion case, $O(1)$ rotations suffices.

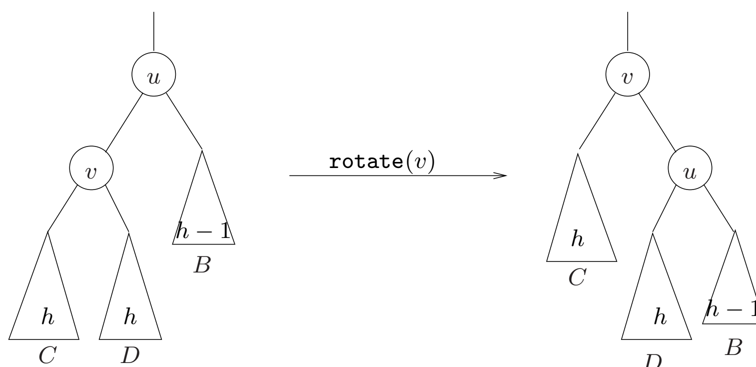


Figure 12: CASE (D.3): $\texttt{rotate}(v)$

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

Exercise 5.2: 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 5.3: 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 5.4: 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 5.5: 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 $19, 1817, 16, 15, 14, 13, 12, 11$. ♢

Exercise 5.6: 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 5.7: 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 the rotation. ♢

Exercise 5.8: Improve the lower bound $\mu(h) \ge \phi^h$ by taking into consideration the effects of "+1" in the recurrence $\mu(h) = 1 + \mu(h-1) + \mu(h-2)$.
(a) Show that $\mu(h) \ge F(h-1) + \phi^h$ 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 $h \ge 2$.
(b) Further improve (a). ♢

Exercise 5.9: Relationship between Fibonacci numbers and $\mu(h)$.
(a) (Jia-Suen Lin) Show that $\mu(h) = 3F(h) + 2F(h-1)$ for all $h \ge 1$. Here $F(h)$ is the standard Fibonacci sequence where $F(i) = i$ for $i = 0, 1$.
(b) 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 5.10: In a typical AVL tree implementation, we may 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) Describe and analyze an insertion algorithm for such trees.
(c) Describe and analyze a deletion algorithm.
(d) What are the relative advantages and disadvantages of biased-AVL trees? ♢

Exercise 5.11: 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 5.12: 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 5.13: Relaxed AVL Trees
Let us define **AVL(2) balance condition** to mean that at each node u in the binary tree, $|balance(u)| \le 2$.
(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. ♢

Exercise 5.14: 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 difficulity 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? $\diamondsuit$

End Exercises

§6. (a, b)-Search Trees

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

$$2 \leq a < b. \tag{7}$$

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

- DEPTH BOUND: All leaves are at the same depth.
- BRANCHING BOUND: Let m be the number of children of an internal node u. If the depth of u is 2 or more then
$$a \leq m \leq b. \tag{8}$$
If the depth is zero (i.e., u is the root) then $2 \leq m \leq b$, and if the depth is 1, then $\min\{a, \lfloor (b+1)/2 \rfloor\} \leq m \leq b$.

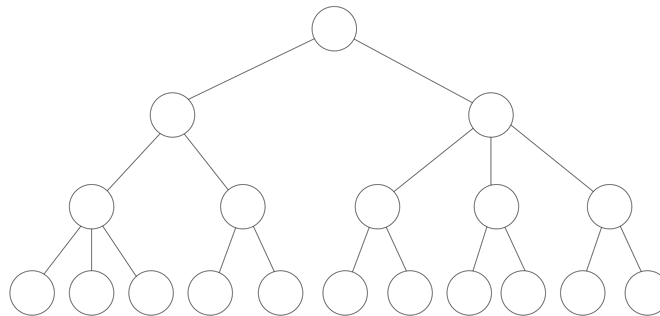


Figure 13: A $(2, 3)$-tree.

The definition of (a, b)-trees has purely structural requirements. Figure 13 illustrates an (a, b)-tree for $(a, b) = (2, 3)$. Next, to use (a, b)-trees as a search structure, we need to give additional requirements.

Recall that an item is a $(\texttt{Key}, \texttt{Data})$ pair. An (a, b)**-search tree** is an (a, b)-tree whose nodes are organized as follows:

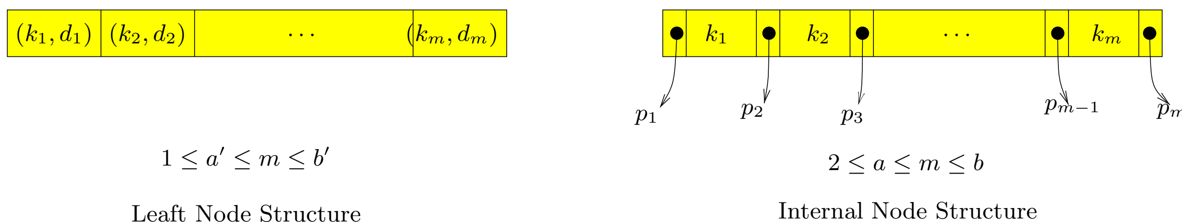


Figure 14: Structure of Nodes in (a,b)-search trees

- LEAF: Each leaf stores a list of items, sorted by their keys. Hence we represent a leaf u with m items as the list,
$$u = (k_1, d_1, k_2, d_2, \ldots, k_m, d_m) \tag{9}$$
where (k_i, d_i) is the ith smallest item. See Figure 14. If leaf u is at depth 2 or more, then $a' \le m \le b'$ for some $1 \le a' \le b'$. There are two canonical choices for the parameters a', b'. The simplest is $a' = b' = 1$. This means each leaf stores exactly one item. Our examples use this version. A more realistic choice is
$$a' = a, b' = b. \tag{10}$$
If u is the root, then $1 \le m \le b'$, and if u is a child of the root then $\min\{a', \lfloor (b'+1)/2 \rfloor\} \le m \le b'$.

- INTERNAL NODE: Each internal node stores an alternating list of keys and pointers (node references), in the form:
$$u = (p_1, k_1, p_2, k_2, p_3, \ldots, p_{m-1}, k_{m-1}, p_m) \tag{11}$$
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 list is one less than the number m of children. See Figure 14. Thus m must satisfy the bounds given by the structural requirement, i.e., $a \le m \le b$ for depth 2 or more. 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
$$k_{i-1} \le k < k_i, \tag{12}$$
with the convention that $k_0 = -\infty < k_i < k_m = +\infty$.

Thus, an (a,b)-search tree is just a (a,b)-tree that has been organized into a search tree. For simplicity, assume that the set of items in an (a,b)-search tree have unique keys. But as we shall see below, this does not mean that the keys in internal nodes must be distinct from keys in the leaves.

An example of a $(2,3)$-search tree is given in Figure 15. Following the usual convention, we do not display the data that is associated with the keys in the leaves. Here we are assuming the canonical choice $a' = b' = 1$.

The simplest (a,b)-search trees is when $(a,b) = (2,3)$. These are called **2-3 trees** and were introduced by Hopcroft (1970). By choosing
$$b = 2a - 1 \tag{13}$$
(for any $a \ge 2$), we obtain a 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** (or its variant known as **red-black trees**). Thus, the concept of (a,b)-trees serve to unify a variety of search trees.

Although B-trees achieves the relationship (13) between the a and b parameters that is optimal in a certain sense, there are benefits when we allow more general relationships between a and b. E.g., relaxation

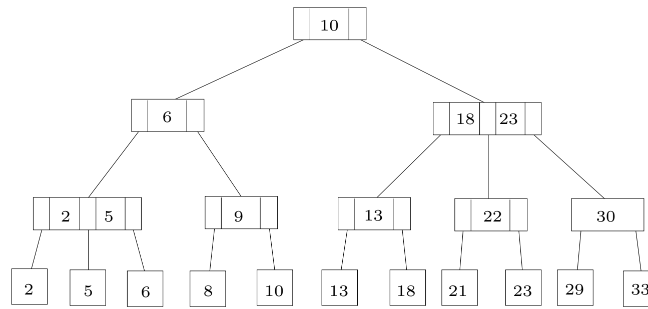


Figure 15: A $(2, 3)$-search tree.

can benefit the amortized behaviour of (a, b)-search trees [3]). The price paid by our generalization is the specialized rule for the degree of nodes at level 1.

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

- Suppose u is an internal node given by (11). Then we find the p_i such that $k_{i-1} \leq k < k_i$ (with $k_0 = -\infty, k_m = \infty$). Set p_i as the new current node and continue.
- Suppose u is a leaf node given by (9). 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.

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

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

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 (11) for internal nodes, and (9) 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 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. There is a major difference between (a, b)-search trees and, say, 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.

Following Tarjan [8, p. 9], we call search structures that store data in the leaves only to be **exogenous**, and otherwise **endogenous**.

Thus, the keys in the internal nodes of (a, b)-search trees are used purely for searching: they are not associated with any data. This distinction is brought out in the above organization of nodes in (a, b)-search tree: the leaves are basically storing items. 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 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 are exogenous search structures (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 actually stored in the leaves. This is seen in Figure 15 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. The reason 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 (9) 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.

There is yet another reason for preferring exogenous structures: In databases, the number of items is very large and these tend to be stored in disk memory. The I/O speed for transferring data between main memory and disk memory is the main bottleneck. 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). That is the space budget we have for each node. We must minimize the number of disk accesses by packing as many keys into each block as possible. Thus the parameter b in our (a, b)-trees is chosen to match the *b*lock size. Below, we discuss some constraints on how the parameter a is chosen.

The Split and Merge Inequalities for (a, b)-trees. The parameters a, b are usually required to satisfy an additional inequality in addition to (7). This inequality, which we now derive, comes from two low-level operations in (a, b)-tree algorithms, called **split** and **merge**. These operations arise in insertion and deletion into (a, b)-trees, respectively. There is actually a family of such inequalities, but we will first derive the simplest one.

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 these satisfy the (a, b)-tree requirement, we have

$$a \leq \left\lfloor \frac{b+1}{2} \right\rfloor . \tag{14}$$

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 $2a-1 \le b$, *i.e.*,

$$a \le \frac{b+1}{2}. \tag{15}$$

Clearly (14) implies (15). However, since a and b are integers, the reverse implication also holds! Thus we normally demand (14) or (15). The smallest choices of these parameters under the inequalities and also (7) is $(a, b) = (2, 3)$, which has been mentioned above. The case of equality in (14) and (15) 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]).

This is also a good place to understand the exceptional treatment of the root in the definition of (a, b)-trees: whenever we split (resp., merge) a node, we increase (resp., decrease) the degree of the parent node by 1. The exceptions to this rule are (i) when we split the root itself. In this case, we need to create a new root with two children. Hence the new root has degree only 2, which is independent of our choice of a or b. (ii) when we merge two nodes of the root, the degree of the root can decrease below a, and there is nothing we can do about it. However, if the degree of the root is 1 after a merge, we can simply delete this root. Note that (i) and (ii) are the *only* means for increasing and decreasing the height of the (a, b)-tree.

Achieving 2/3-factor Space Utility. The factor a/b is called the **space utilization ratio**. We would like this factor to be arbitrarily close to 1. A node with m children is said to be (m/b)**-full**, and it is **full** when $m = b$. The standard treatment in the literature imposes inequality (15) on (a, b)-trees. This implies that the space utilization on such trees (essentially B-trees) can never[8] be better than 1/2. For instance, if a can be about $2b/3$ then this ensures that every node is always at least 2/3-full.

Here is how to do this: when splitting a node with $b+1$ child, we may assume that its sibling is already full. In this case, we can take the $2b+1$ nodes in these siblings and divide them 3 ways as evenly as possible. So the nodes have between $\lfloor (2b+1)/3 \rfloor$ and $\lceil (2b+1)/3 \rceil$ keys. More precisely, the number of keys in the three nodes are exactly

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

where $\lfloor (2b+1)/3 \rceil$ denotes **rounding** to the nearest integer. As a result, we require

$$a \le \left\lfloor \frac{2b+1}{3} \right\rceil. \tag{16}$$

Similarly, when we merge a node with $a-1$ nodes, we may look at the adjacent sibling of its adjacent sibling to try to borrow a node. Assuming

$$a \ge 3, \tag{17}$$

there are always such siblings to consider. When this proves impossible, we will need to merge three nodes with a total of $3a-1$ children and redistribute them into two nodes with $\lfloor (3a-1)/2 \rfloor$ and $\lceil (3a-1)/2 \rceil$ children, respectively. This means

$$\left\lceil \frac{3a-1}{2} \right\rceil \le b \tag{18}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (16) and (18), without changing the relationship. And thus both inequality are seen to be equivalent to

$$a \le \frac{2b+1}{3} \tag{19}$$

[8] The ratio a/b is only an approximate measure of space utility for various reasons. For one thing, the true ratio is $\lfloor (b+1)/2 \rfloor /b$, which is strictly greater than 1/2. E.g., for $b = 3$, we get $\lfloor (b+1)/2 \rfloor /b = 2/3$. So 1/2 is only the asymptotic limit as b grows. Furthermore, the relative sizes for keys and pointers also affect the space utilization. We have assumed that keys and pointers have the same size.

As in the standard (a,b)-trees, we need to allow exceptions for the root. Here, the number m of children of the root satisfies the bound $2 \le m \le b$. But we now need a further exception: if u is a child of the root with m children, then we only require:

$$\min\{a, \frac{b+1}{2}\} \le m \le b. \tag{20}$$

That is because we may not find a sibling while trying to split the root, nor find three siblings for merging into two nodes. Note that, since $(b+1)/2 \le (2b+1)/3$, the constraint (20) is a relaxation of the usual constraint (8).

The smallest example of such a (2/3)-full tree is $(a,b) = (3,4)$. Note that is this case, we actually achieve a 3/4-full tree.

Insertion Algorithm Let us now describe the insertion algorithm under the constraint (19). There are several useful subroutines to develop:

- Suppose u is a leaf. Let $INS(k,d,u)$ amount to inserting the item (k,d) into u and splitting u if necessary. It returns a new node u' which is non-null if a split actually occurs. This u' needs to be inserted into the parent.
- Suppose u has a sibling v that has less than the maximum number of keys. We need to move a key from u to v. Let w be the parent of u and v. Then there is a key k_0 in w that separates the keys in u and v. Suppose that the keys in u is less than that in v, so we need to move the largest key k_1 out of u, and also the last pointer p_1 out of u. This is done by making (p_1, k_0) be the first two components of v; also, in w, we replace k_0 by k_1 in w.

  [FIGURE HERE]

  In general, v could be a cousin rather than a sibling. In that case w should be the least common ancestor (LCA) of u and v.
- Suppose u has a sibling v that has the maximum number of keys. Again, let w be their parent and k_0 be the key that separates the keys of u and v. We will form a new (pointer,key)-list from the list in u, followed by k_0, followed by the list in v. We then split this into three new lists and store them in u, x and v (in this order), where x is a new node. But u and v are the original nodes, with new lists. This splitting will generate a triple (k, p_x, k') where p_x points to x, and k separates the keys of u and x while k' separates the keys of x and v. We insert this triple into the list of w, in place of k_0.

  [FIGURE HERE]
- Suppose u is a root.

Generalized Split-Merge Method for (a,b,c)-trees. The 2/3-space utility method is only one member of a parametrized family of split-merge algorithms. This parameter is denoted c and can be any integer ≥ 1. When $c = 1$, we have the B-trees and when $c = 2$, we achieve the 2/3-space utilization factor above. We call these (a,b,c)-**trees** and they achieve a space utilization factor of $c/(c+1)$. We now require the inequalities

$$c + 1 \le a \le \frac{cb+1}{c+1}. \tag{21}$$

The lower bound on a ensures that when we merge or split a node, we can be assured of at least $a - 1 \ge c$ siblings in which to borrow (if you are short) or share (if you are long) your 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

(21) on a. In case of splitting, the current node has $b+1$ keys. You may assume $c-1$ siblings with b keys each. In order for there to be $c-1$ siblings, we require

$$c \le b. \tag{22}$$

Then we combine all these $cb+1$ keys, and split them into $c+1$ new nodes. Again, the upper bound on a (21) is needed.

In case $a = \frac{cb+1}{c+1}$, we call such an (a,b)-tree a **generalized B-tree**. Thus standard B-trees correspond to the case $c = 1$. Refer to the extra c parameter, we may speak of (a,b,c)**-trees**.

Insertion and Deletion Algorithms for (a,b,c)**-trees.** We first describe the insertion algorithm for $(a,b,2)$-trees. The generalization to $c > 2$ should be immediate. Suppose we want to insert a given data item (k,d) where k is a key and d is the associated data. The following algorithm uses an explicit stack (we could also avoid this by using a recursive algorithm).

We use the following terminology in the algorithm below: a node is **full** if it has the maximum number of keys (recall that this maximum number is $b-1$ for an internal node at depth ≥ 2, and another number b' for leaves). If a node is not full, we say it is **overfull** or **underfull** depending on whether the number of keys is more or less (resp.) than this maximum. Suppose $u_1, \ldots, u_k$ are all the children of a node, and the keys in these nodes are ordered $u_1 < u_2 < \cdots < u_k$, then we say u_i and u_{i+1} are immediate siblings. Thus every node has at most two immediate siblings.

Insertion Algorithm.
1. Initialization Let u be initialized to the root of the (a,b,c)-tree. We go down the tree, searching for the key k in the obvious way, as given in the next step.
2. Searching Repeat while u is not a leaf: let v be the child of u that we must next visit. Push u to the stack and update $u \leftarrow v$.
3. Insertion At this point, u is a leaf. If the key k is already in the u, return an error (say). Otherwise, we insert (k,d) into the u.
4. Leaf Split If u is not overfull, return successfully. Otherwise pop w from the stack. So w is the parent of u. If u has an immediate sibling s that is underfull, move an item from u to s and after adjusting w accordingly, we return with success. Otherwise let u'' be an immediate sibling of u that is full. Assume that the keys in u are less than the keys in u'' (the other case is symmetric). We create a new node u' and redistribute the items in (u,u'') into (u,u',u'') (the keys are ordered so $u < u' < u''$). We insert a pointer to u' into w, and adjust the keys in w. Let $u \leftarrow w$.
5. Recursive Split Repeat while u is overfull: Pop w from the stack (so u is a child of w). If u has an immediate sibling s that is underfull, then we move a pointer from u to s and after adjusting the keys in w appropriately, we return success. Otherwise let u'' be an immediate sibling of u that is full. Assume that the keys in u are greater than the keys in u'' (the other case is symmetric). We create a new node u' and redistribute the keys in (u'',u) into (u'',u',u) (the keys are ordered so $u'' < u' < u$). We insert a pointer to u' into w, and adjust the keys in w. If w is not overfull, we return success. Otherwise $u \leftarrow w$ (and repeat).

The above insertion and deletion algorithms use the strategy of "share a key if you can" in order to avoid splitting or merging. The above assume $c = 2$. In case $c > 2$, we must look further than the immediate siblings. For instance, with $c = 3$, the insertion algorithm will look for the immediate sibling of an immediate sibling for an underfull node for sharing. When this is impossible, it will take the current overfull node u with two other full immediate siblings (u', u'') and redistribute their keys and pointers into (u, u', u'', u''') where u''' is a new node.

An analogous strategy will work for deletion: "borrow a key if you can" instead of merging. We leave the details to the reader.

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 version. This is discussed in the exercises. The basic idea is to pre-emptively split (in case of insertion) or pre-emptively merge (incase of deletion).

Space Utilization. Let us look at the space utilization ratio more carefully. Our model of space utilization assumes that we allocate the same amount of space to every node of a search tree. Furthermore, the ratio a/b is only an approximate guage because of several issues. In particular, we need to know the space requirement for each of the following objects: a key value, a pointer to a node, either a pointer to an item (in the exogenous case) or the data itself (in the endogenous case).

With the standard B-tree, we 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 result was the beginning of a technique called "fringe analysis" which Yao [9] introduced in 1974. The survey of [1] notes that Nakamura and Mizoguchi [7] independently discovered the analysis, and Knuth had used the same ideas back in 1973. Note that our bounds (19) implies a space utilization that is close to $\lg 2$, but we guarantee this utilization in the worst case.

EXERCISES

Exercise 6.1: What is the the best ratio achievable under (15)? Under (19)? ♢

Exercise 6.2: 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 (15). ♢

Exercise 6.3: 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 6.4: Describe in detail the insertion and deletion algorithms in the following cases: $(2, 3)$-trees and $(2, 4)$-trees. ♢

Exercise 6.5: The special rule for the root of an (a, b)-tree can be replaced 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 6.6: 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

§7. Red-Black Trees

Red-black trees form a balanced family that, in some sense, is the most economical among height-balanced binary trees. Each node in such a tree is colored[9] red or black, where red nodes are considered deficient. Such trees have some nice properties (see Exercise on treaps below). The drawback is the relative complexity of their insertion and deletion algorithms. There is an alternative view of red-black Trees that connects then to $(2,4)$-trees. Since $(2,4)$-trees (as a special case of (a,b)-trees) have simpler algorithms, we can think of the red-black tree algorithms as the unravelled $(2,4)$-tree algorihms.

It is convenient to introduce the "extended" version of red-black trees:

An **extended red-black tree** is an extended binary search tree in which each node is colored either red or black. The color scheme satisfies properties $B(x)$, $H(x)$ and $P(x)$ at each node x:

Basis property $B(x)$:	If x is a nil node, then it is black.
Height property $H(x)$:	The number of black nodes in a path from node x to any nil node is invariant. This invariant number, minus one, is called the **black height** of x. So a nil node has black height of 0.
Parent property $P(x)$:	If x is red then its *p*arent (if any) is black.

A **(standard) red-black tree tree** is a standard binary search tree T whose nodes are colored either red or black such that its corresponding extended version T' (with nil nodes colored black) is an extended red-black tree. The two versions of red-black tree are interchangeable. Sometimes it is easier to work with the extended version but most of the time we assume the standard version. Figure 16 illustrates a red-black tree in its two forms.

The theoretical advantage of red-black trees over AVL trees is that we use only 1 bit of balance information per node, as opposed to $\lg 3$ bits in AVL trees.

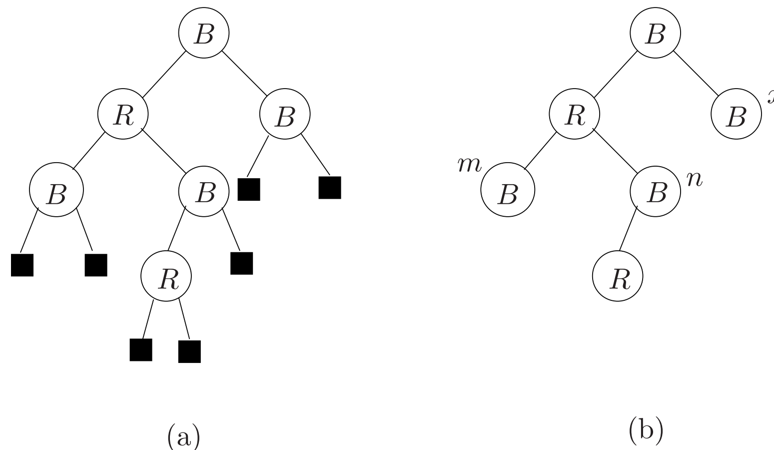


Figure 16: Two equivalent red-black trees: (a) extended version, (b) standard version.

[9]This color terminology is borrowed from accounting practice, as in balance sheets being in the red or in the black.

There are some easy consequences of the basis, height and parent properties:
(i) Each node in a red-black tree of height two or more must have two children. (Recall that a node has height 0 iff it is a leaf.)
(ii) A node with exactly one child must be black, and its unique child must be a leaf with color red.

With these observations, we can now give a direct definition of red-black trees (without going through the extended binary search trees). At each node x, the following three properties hold:

Basis property $B'(x)$:	If x has only one child, then that child is a leaf with color red.
Height property $H'(x)$:	The number of black nodes in a path from node x to any leaf is invariant. This invariant number is the **black height** of x.
Parent property $P'(x)$:	If x is red then its *p*arent (if any) is black.

Note that the new parent property $P'(x)$ is identical to $P(x)$. In the following descriptions, we shall continue to use $B(x)$ and $H(x)$ instead of $B'(x)$ and $H'(x)$.

Let T be an ordinary binary tree whose nodes are colored red or black and x is a node of T. The meaning should be clear when we say that T **violates the parent property** $P(x)$, or equivalently, there is a P**-violation at** x. Similarly for the height property, we speak of violating $H(x)$ or a H-violation at x. It may be helpful[10] for the reader to think of the red nodes as deficient in a certain sense. The parent property ensures that there are no two consecutive deficient nodes in a path.

Note that if the root of a red-black tree is red, we can just color it black, and the result is still a red-black tree. For this reason, some literature assumes that the root of a red-black tree is always black.

If x is a node in a red-black tree T, let $\mathtt{bht}[x] = \mathtt{bht}_T[x]$ denote the black height of x in T and $\mathtt{bht}[T]$ denote the black height of the root of T. For instance, if x is a black leaf, $\mathtt{bht}[x] = 1$. For any path p, we also let $\mathtt{bht}[p]$ denote the number of black nodes in p. Red-black trees are automatically balanced:

Lemma 3 *Suppose a red-black tree T has n nodes and black height h. Then $n \geq 2^h - 1$ and hence $h \leq \lg(n+1)$. Hence T has height at most $2\lfloor \lg(n+1) \rfloor$.*

Proof. We prove that $n \geq 2^h - 1$ by induction on $h \geq 0$. This is true when $h = 0$, for T has 0 or 1 node. If $h \geq 1$, then the left and right subtree at T has black height at least $h - 1$ (they could have height h). By induction, they each have at least $2^{h-1} - 1$ nodes. Thus T has at least $2(2^{h-1} - 1) + 1 = 2^h - 1$ nodes, as claimed. The rest of the lemma is immediate: from $n \geq 2^h - 1$ we get $h \leq \lfloor \lg(n+1) \rfloor$. By the parent property, the height of the tree is at most $2h$. **Q.E.D.**

Operations on red-black trees. We next consider the basic operations of `lookUp`(*Key*), `insert`(*Item*) and `delete`(*Node*) on red-black trees. Since red-black trees are binary search trees, the lookup operation can be done as for binary search trees. The following terminology will be useful in our descriptions. The usual terminology for binary trees views them as a kind of family tree where each node has at most 2 children and reproduction is asexual. Extending this analogy, the children of a sibling are called **nephews**. But we may distinguish one of these nephews as a **near-nephew** and the other as a **far-nephew**. For instance, in figure 16(b), the nodes n, m are respectively the near-nephew and far-nephews of x. Of course, the reciprocal

[10] The colors in our trees comes from "red ink" and "black ink". That is, they are accounting analogies, with no racial connotations!

relation is an **uncle**: x is the unique uncle of m and of n. Note that a node is either a near-nephew or a far-nephew of its uncle, but not both.

Exercises

Exercise 7.1: For each of the following binary trees, say whether it could be the shape of a red-black tree or not. If yes, show a coloring scheme; if not, argue why not. i ♢

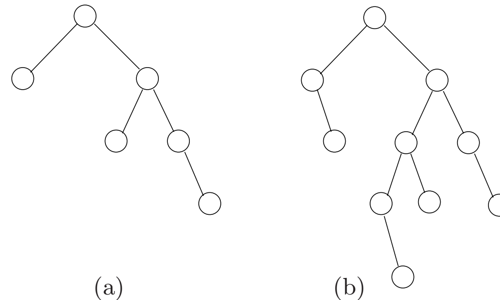


Figure 17: Some binary trees.

Exercise 7.2:
(a) What is the maximum possible height (not black height) of a red-black tree with 10 keys? Argue rigorously why your bound is correct.
(b) Show a lower bound (draw a read-black tree with this bound). ♢

Exercise 7.3: Give a direct definition of red-black trees for ordinary binary search trees. (In our text, we defined it indirectly, via extended binary search trees.) ♢

Exercise 7.4: A "2-4 tree" is a search tree in which each interior node has degree 2, 3 or 4 children, and every path from the root to a leaf has the same length. It is a search tree in the sense that if a node has d ($d = 2, 3, 4$) children, then the node stores $d-1$ items. The keys of these items $K_1 < K_2 < \cdots < K_{d-1}$ are ordered and we have a generalization of the binary search tree property. This facilates searching for any key.
(a) Show that every red-black tree can be interpreted as a 2-4 tree. HINT: assume that the root is black.
(b) Do a detail comparison of the insertion algorithms in $(2, 4)$-trees and red-black trees.
(c) Repeat part (b) for deletion. ♢

Exercise 7.5: (Cormen-Leiserson-Rivest) Alternative definition of black height: define the **black height** of node u to be the number of black nodes along any path from u to a nil node, but *not* counting u. That is, if u is red, then our black height is 1 less than the alternative definition. In particular, if a red-black tree has a red node, we can change it to black and its height is not changed according to this definition. One disadvantage of this definition is the need to refer to nil nodes in its definition. But does this alternative definition make any difference to our algorithms? ♢

Exercise 7.6: An alternative definition of an extended red-black trees is in terms of a rank function $r(x)$ for each node x with these properties: (i) If the parent $p(x)$ of x exists, then $r(x) \le r(p(x)) \le r(x)+1$.

(ii) If the grandparent $p(p(x))$ exists, then $r(x) < r(p(p(x)))$. (iii) If x is a leaf, $r(x) = 1$ and if $x = \mathsf{Nil}$ then $r(x) = 0$. Show the "equivalence" of such trees and red-black trees; part of the problem is to define "equivalence" precisely. HINT: use the definition of black height in the previous exercise. ♢

Exercise 7.7: (Olivié) A binary tree is **half-balanced** if for every node x, the length of the longest path from x to a leaf is at most twice as long as the length of the shortest path from x to a leaf. As in the previous exercise, show the "equivalence" of such trees and red-black trees. ♢

Exercise 7.8: Let $M(h)$ denote the maximum number of key-bearing nodes in a red-black tree with black height of h. Write the recurrence equation for $M(h)$. Solve the recurrence. NOTE: what if we do the same for $m(h)$, denoting the minimum number of key-bearing nodes in a red-black tree with black height h? ♢

End Exercises

§8. Red-Black Insertion

The following description is designed so that the student can easily commit to memory the insertion algorithm. Consequently, the student should be able to perform hand-simulation of the algorithm on actual trees. For insertion, we prefer to treat the tree as a standard binary tree.

Suppose we want to insert a node x into a tree T. There are three steps in insertion:

1. Insert x into T as in a binary search tree.
2. Make x a red node.
3. Rebalance at x.

We must explain the third step of rebalancing. After steps 1 and 2, x is a red leaf. We verify that the tree automatically satisfies the height property. Next, property $P(u)$ holds at each node u except possibly when $u = x$. In fact, the only possible violation of red-black tree properties is when the parent of x is red. We are done if the parent of x is black. Hence assume otherwise.

This single violation is key to understanding the insertion operation, and deserves a definition: let x be any node of binary search tree T whose nodes are colored red or black. We call T an **almost red-black tree at node** x if T satisfies the basis and height properties. Moreover, it satisfies property $P(u)$ for all nodes $u \neq x$ but does not satisfy $P(x)$. In this case, we also say T has a P**-violation** at x or an **insertion violation** at x.

Rebalancing an almost red-black tree at x. Rebalancing an almost red-black tree means to convert the tree into a red-black tree. This is reduced to a sequence of (repeated) "rebalancing steps". Each rebalancing step either (I) transfers the violation at x to the grandparent of x, or (II) remove the violation completely (and we terminate). The color of the **uncle** of x decides which of these two cases applies. We consider the following scenario (see figure 18):

Insertion scenario.
We are trying to rebalance an almost red-black tree T at x. Both x and its parent y are red. Let z be the parent of y and u the sibling of y (so u is the uncle of x). Then z must be black but the color of u is unknown and is the critical determinant of the action we take.

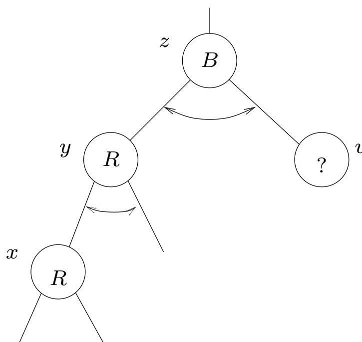


Figure 18: Insertion scenario: violation at x.

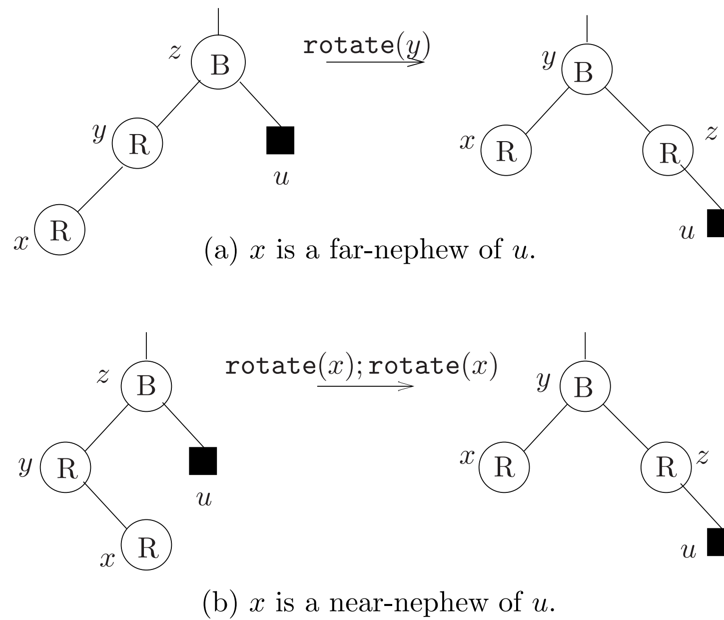
Graphical convention: Figure 18 illustrates a convention for displaying binary trees: a circular arc connecting the two edges emanating from node z to its children u and y indicates that the left-right ordering of u and y is not specified by our figure. Of course, a figure must illustrate one of the two possibilities, but this choice is arbitrary. Thus figure 18 is really a summary of 4 possibilities.

Justification for Insertion Scenario. Why does this scenario hold? Note that if y did not exist, we would not have a violation at x. If the grandparent z of x did not exist, then we can simply color y black and the result would be a red-black tree. *This recoloring is the only operation in the insertion algorithm that increases the black height of a tree.* What about u? In general, its existence is guaranteed by the basis property $B(z)$. However, if x has *just* been inserted, u may not exist (that is, u may be a nil node). In this case, it is easy to see that x has no sibling or children. This situation can easily be fixed as illustrated in figure 19:

FixSuperNode(x):
(a) If x is a far-nephew of u, we rotate at y,
blacken y and redden its children x, z.
(b) If x is a near-nephew of u, we rotate twice at x.
We blacken x and redden its children (y and z).

In either case, the reader verifies that the result is a red-black tree.

This transformation of the triple x, y, z will be encountered again. It is helpful to think of x, y, z as a *supernode* (hence the name of this little routine). Hence (a) and (b) are just two cases in the rebalancing of a supernode. The operation in (b) is quite common and is called a **double rotation**. In general, a double rotation at a node x is defined if x is the near-nephew of its uncle and the operation amounts to two consecutive rotations of x.



(a) x is a far-nephew of u.

(b) x is a near-nephew of u.

Figure 19: When u does not exist: two cases.

To conclude, this "justification" of the insertion scenario amounts to a procedure (let us call it Convert) to transform an insertion violation into the insertion scenario, or else to remove the insertion violation.

Rebalance Step. We assume the insertion scenario and consider two cases.

CASE I: The uncle u is red. This case is easy: we simply redden z and blacken both y and u (see figure 20).

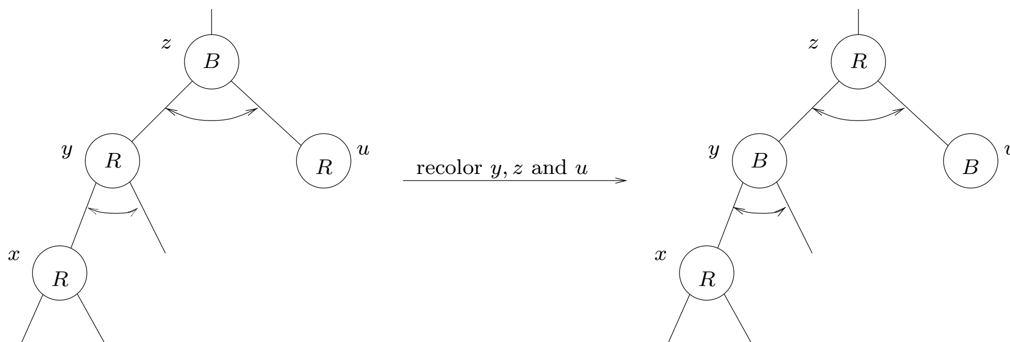


Figure 20: Red uncle: possible new violation at z.

It is not hard to check that the result is either a red-black tree or an almost red-black tree at z. In the latter case, we recursively do a rebalance at z. In short, the rebalance step has either removed the insertion violation or moved it closer to the root. Eventually the recursive process must stop.

CASE II: The uncle u is black. In this case, we transform the tree as indicated in figure 21: This

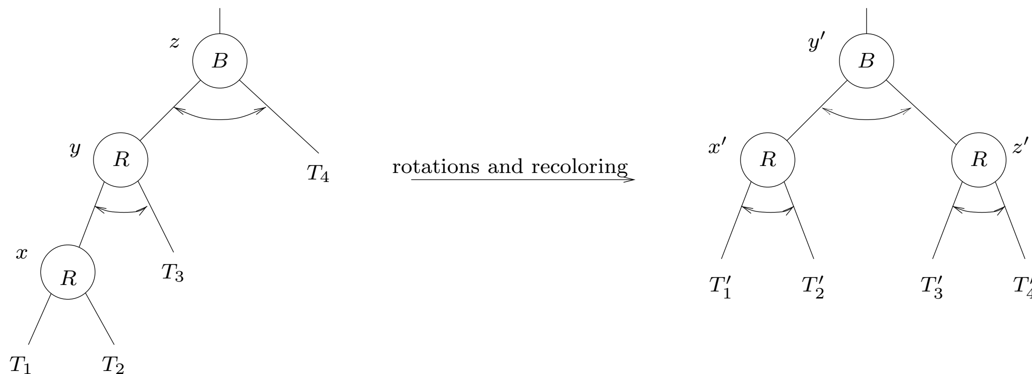


Figure 21: Black uncle: root of T_4, the uncle of x, is black.

amounts to rebalancing the supernode x, y, z. That is, we just call the routine FixSuperNode(x). Since the result is a red-black tree, the black uncle case is a terminal one.

We expand on figure 21 a little bit. The left-hand side is a combination of 4 cases, depending on whether x is a left or right child of y and whether y is a left or right child of z. Note that the roots of the four subtrees T_1, T_2, T_3 and T_4 are all black (u, the root of T_4, is black by assumption and the others are black by the parent property). This transformation is completely specified by the following requirements:

- x', y', z' is simply x, y, z in non-decreasing order of their keys.
- T_1', T_2', T_3' and T_4' are the the subtrees T_1, T_2, T_3 and T_4 in non-decreasing order.

The reason this is enough to determine the tree on the right-hand side of figure 21 is because the result must be a binary search tree. Take the possibility where x is the right child of y, and y the left child of z (see figure 22). Here, if we perform a double rotations at x, redden z and blacken x, we obtain the desired

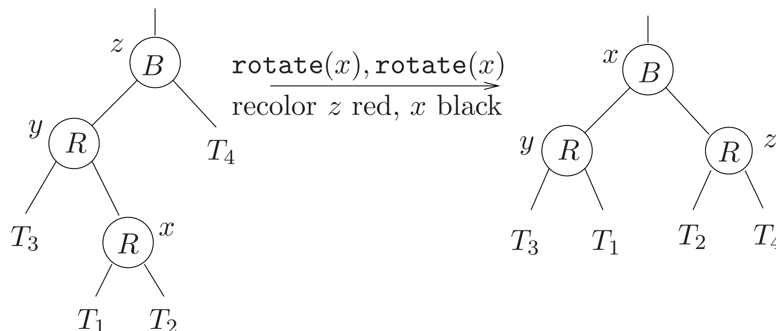


Figure 22: Black Uncle: x is right child of y, y is left child of z.

red-black tree. We can similarly work out the other three possibilities.

This completes our discussion of the black uncle case and hence of the rebalance step. To summarize the insertion algorithm:

```
INSERT( 1 I):
    1.    Insert I as in a binary search tree; color the inserted node red.
    2.    while there is a violation, do
    2.1       Convert to the insertion scenario.
    2.2       Perform a rebalance step.
```

Exercises

Exercise 8.1: Insert the following sequence of keys (in succession) into an initially empty tree: $10, 8, 3, 1, 4, 2, 5, 9, 6, 7$. ♢

Exercise 8.2: Verify carefully that after the rotations and recolorings in the rebalance steps always result in a red-black tree or an almost red-black tree. ♢

Exercise 8.3: Draw a red-black tree T_1 with black height 2 and specify a key k such that inserting k into T_1 will increase the black height of T_1. Draw the red-black tree after inserting k. **HINT:** when does the black height increase in the insertion procedure? ♢

Exercise 8.4: Is the recoloring in our rebalancing steps ad hoc? Try to give a more systematic account of how to assign colors. (This is certainly another useful aid to memory.) ♢

Exercise 8.5: (One pass version) Our insertion algorithm requires two pases, one pass down and the other pass up the tree. Design an alternative insertion algorithm which has only one pass. HINT: the idea is "preemptive rebalance". Try to make sure that a node is black before you visit it. ♢

End Exercises

§9. Red-Black Deletion

Deletion is slightly more involved than insertion. Again, we design the algorithm to be remembered, so that the reader to perform hand-simulation of the algorithm. For this description, we prefer to use the terminology of extended binary trees.

Suppose we want to delete the item in a node u in a red-black tree T. We delete item in u as we would in an ordinary binary tree, and then we rebalance to ensure that the result is still a red-black tree. It is important that we use the "standard deletion algorithm" (see §3) which does not use rotations. Note that the standard deletion algorithm may not actually remove the node u. Instead, some node u' with only one child will be removed.

Deletion violation. Again, the deletion of node u' causes a "violation", which we now explain. Imagine the nodes of our search tree to carry "tokens" – a red node has no token and a black node has one token. The nil nodes (recall we view T as an extended binary tree now) are automatically black and so carries a token. Thus, the black height of a node u is one less than the number of tokens along any path from u to a nil node. The height property simply says that the number of black tokens along a path from any node to a nil node is invariant. Thus if the deleted node u' is red, we do not change the number of tokens along any path, so the height property is preserved. So assume that the deleted node u' is black. Recall that u' has at most one child. If u' has no parent, then the result is again a red-black tree. Hence assume that u' has a parent y. After deleting u', one of the children of y becomes a node x in place of u'. Note that x is a nil node if u' was a leaf, but otherwise x was the only child of u'. *Let us give the token of* u' *to* x *after deletion.* Now the height property is restored in the sense that every path to a nil node still contains the same number of tokens!

What can still go wrong? Well, if x already has a token, then giving x another token means that x now has two tokens. We say a node is **doubly-black** if it has 2 tokens. First observe that if u' was not a leaf, then we know that x must be a red leaf. This means that x is black, not doubly-black. But if u' is a leaf, then x is a nil node and hence it is now doubly-black. In figures, we use R, B, D to denote red, black and doubly-black nodes, respectively.

An extended binary search tree T in which an extended node x is colored doubly-black (D), with the remaining nodes are colored black (B) or red (R), is said to be an **almost red-black tree at** x if T satisfies the basis and parent properties, and also the modified height property, interpreted in terms of counting tokens as above. We also say[11] that T has a **deletion violation at** x.

We summarize the deletion algorithm:

1. Use the standard deletion algorithm to delete (the item in) u.
   Let u' be the node that is physically removed in this deletion.
2. If u' is red or has no parent, we terminate with a red-black tree.
   `{Henceforth assume` u' `is black and has parent` y`.}`
3. Let x be the extended node that is the child of y in place of u'. If x is non-nil, we again terminate.
4. Otherwise, we doubly blacken the nil node x.
5. Call the rebalancing procedure at node x.

In the remainder of this section, we describe the rebalancing procedure for an almost red-black tree at a node x. The rebalancing procedure is recursive and consists of repeated application of a "rebalancing step". Each step either removes the violation at x, or move it to some node nearer the root. Each rebalancing step assumes the following basic scenario (see figure 23):

> **Deletion scenario.**
> We have an almost red-black tree which is doubly-black at x. The node x may be nil. The parent and sibling of x is p and y, respectively. The children of y are n and m, which are respectively the near-nephew and far-nephew of x.

Conversion to the Deletion Scenario. The illustration in figure 23 is completely general, up to a mirror symmetry (that is, x may be the right child of p, etc). How can we "justify" this scenario? Suppose we have

[11] We used the same terminology "almost red-black tree" when the tree has an insertion violation; this ambiguity is not a problem because we normally are in either insertion or deletion mode, but not both simultaneously. The context will make the type of implied violation clear.

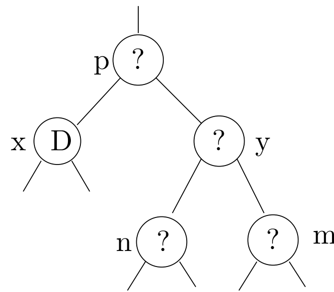


Figure 23: Deletion scenario: violation at x.

a deletion violation at x. If p did not exist, we can simply color x black and the result is a red-black tree. If p exists then the height property implies the existence of y. Now the black-height of y is equal to the black height of x, which is at least 2. Hence y is not nil. Note that if x is non-nil, then the two children m, n of y must also be non-nil. This justification amounts to a tiny procedure Convert for bringing an deletion violation into the deletion scenario (or, failing that, to terminate in a red-black tree).

Rebalancing Step. We now describe the rebalancing step under the hypothesis of the deletion scenario. There are 3 cases to consider, depending on the colors of y, m, n. The simplest is the following.

I. All-black case. The sibling y and the two nephews m, n are all black (see figure 24). Then by coloring y red and by giving a black token to the parent p, we get either a red-black tree (if p was originally red) or an almost red-black tree at p (if p was originally black). Thus the deletion violation is either removed or moved closer to the root.

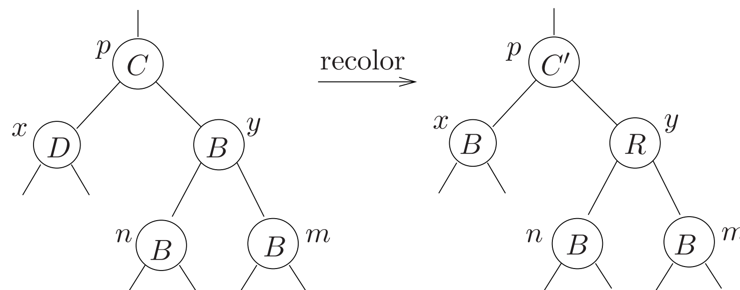


Figure 24: x is doubly black, with black sibling and nephews.

II. Red-nephew case. Suppose some nephew of x is red. So y is black. There are two possibilities:
(a) The far-nephew m is red: See figure 25. We can rotate at y, give the color of p to y, and recolor m, p and x to be black. The reader can verify that the result is a red-black tree. So this is a terminal case.

(b) The near-nephew n is red: See figure 26. We may further assume that the far-nephew m is black. By rotating at n, blackening n and reddening y, we have reduced this to case (a) where the far-nephew is red. (But note that case (a) will immediately cause a rotation at n, so in effect, we have a double rotation at n and this case may be regarded as terminal also.)

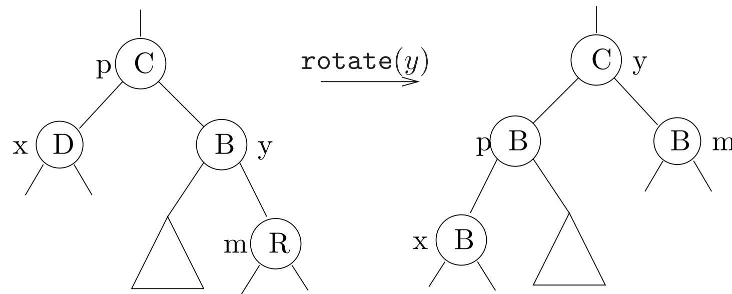


Figure 25: Far-nephew m is red.

HINT: We can combine both cases and view this as the rebalancing of a supernode (cf. the FixSuperNode routine above). More precisely, in case (a), we rebalance the supernode m, y, p. Then y is given the old color of p, and m, p are blackened. Case (b) is similar: we rebalance the supernode m, y, p. (That is, the role of m taken over by n.) Then y is given the old color of p; and n, p are blackened. In both cases, we make x black and terminate.

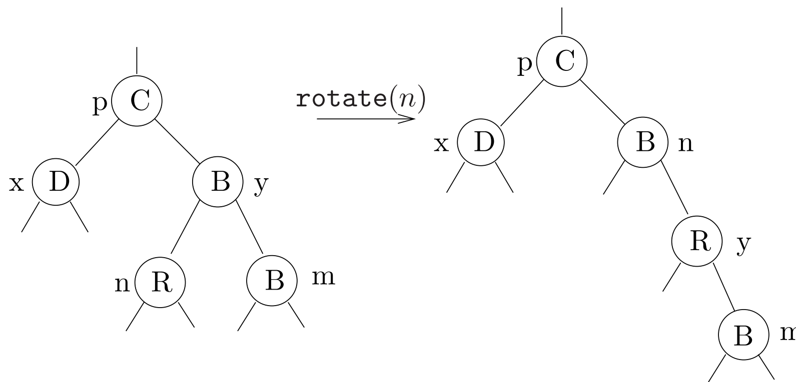


Figure 26: Near-nephew n is red.

III. Red sibling case. The sibling y of x is red (figure 27). So the common parent p of x and y is black. In this case, we rotate at y, redden p and blacken y. The result is still an almost red-black tree at x, except that the sibling of x is black. This means we have reduced our situation to cases I or II.

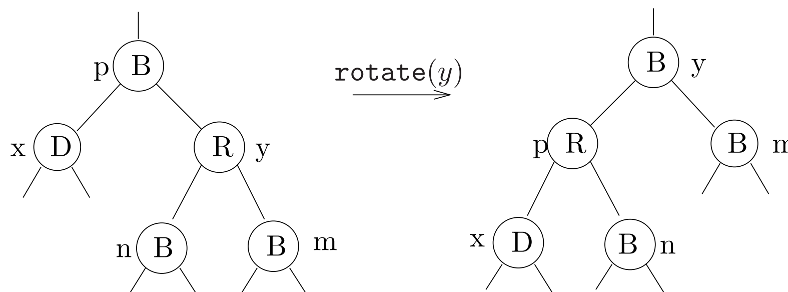


Figure 27: Red sibling case.

But there is a subtle point here – the depth of x is increased. To be sure that we are not in a non-

terminating loop, we must analyze deeper: since case II is terminal, we only have to worry about a reduction to case I (all-black case) which may or may not terminate. But if we were reduced to the all-black case, it is easy to check that we would terminate after the necessary recoloring. Remark: this check is just for our analysis – the algorithm need not do anything special.

HINT: we can view case III as rebalancing the supernode m, y, p, somewhat like the red far-nephew case.

The above analysis shows that the only recursive case is the all-black case. This recursion can repeat at most $2 \lg n$ times before we reach the root. But, regardless of how rebalancing steps are performed, only a constant number of rotations are performed.

Exercises

Exercise 9.1:
(a) Execute the following the red-black tree insert and delete operations (the meaning is self-explanatory): $Ins(5, 7, 3, 9, 10, 8), Del(10)$.
(b) Instead of $Del(10)$, do $Del(3)$. ♢

Exercise 9.2: Verify that the deletion operation makes only a constant number of rotations, not $\Theta(\lg n)$ rotations. (You should flesh out some of the claims in the text about termination.) ♢

Exercise 9.3: Write the deletion algorithm in a reasonable pseudo-code, making explicit any assumptions about the datastructure and basic operations. Notes: there should be a procedure called Convert. Presumably, the argument to Convert is a node x where we have a deletion violation. The technical problem is that x may be nil. So a solution is that we call convert with the pair of nodes, Convert(x, y) where y is the parent of x. Here, either x or y may be Nil. ♢

Exercise 9.4: (a) When does the black height of a tree decrease in a deletion?
(b) Is it possible to have a red-black tree so that its black height increases when you insert a certain key, and its black height decreases when you delete a certain node? ♢

Exercise 9.5: (One pass version) Our deletion algorithm requires two pases, one pass down and the other pass up the tree. Design an alternative deletion algorithm which has only one pass. HINT: do "pre-emptive rebalance" by making sure that a node is red before you visit it. ♢

Exercise 9.6: Give an alternative description (not code) for the deletion algorithm without reference to nil nodes. That is, view them as standard binary search trees. ♢

Exercise 9.7:
(a) Show that we can modify the colors of a red-black tree so that each node of height 1 is black. Note that a leaf has height 0.
(b) Modify the insertion and deletion algorithms for such red-black trees. ♢

Exercise 9.8: Let S be a set of n points in the plane. The **treap** data structure of E. McCreight stores a set S of points using their x-coordinate as key. I also stores at each node u the largest y-coordinate

among all the points in the subtree at u. The underlying data structure is a binary search tree.
(a) Assume that binary search tree is a red-black tree. Show how to insert and delete points from treaps.
(b) Analyze the complexity of the algorithms in (a).
(c) Can you achieve the same complexity if you use AVL trees?
(d) Can you achieve the same complexity if you use 2-3 trees? ♢

Exercise 9.9: (open-ended) Suppose we have *tricolored binary search trees* in which each node is colored red, black or doubly-black, satisfying some suitable modified Basis, Height and Parent properties. Work out the insert and delete algorithms for such search trees. Discuss advantages or disadvantages of these trees. ♢

END EXERCISES

§10. Merge and Split

Suppose we want to implement the additional operations of merging and splitting, *i.e.*, the operations of a fully mergeable dictionary (§2). We shall write

$$T_1 < T_2$$

to indicate that all the keys in T_1 are less than the keys in T_2.

Merge. Consider how to merge T_1 and T_2 under the assumption $T_1 < T_2$. First we delete the maximum item u in T_1, and still call the resulting tree T_1. This takes $O(\log n)$ time. So we have to solve the related problem of merging the following three trees,

$$T_1 < u < T_2.$$

If $\mathtt{bht}[T_1] = \mathtt{bht}[T_2]$, merging is trivial: we just make u the root with T_1, T_2 as the left and right subtrees. So now assume $\mathtt{bht}[T_1] > \mathtt{bht}[T_2]$ (the other case is similarly treated).

Let us now walk down the right subpath $(x_0, x_1, \ldots)$ of T_1, terminating at a node x_t whose right subtree $S'_t = S'$ has the same blackheight as T_2. In general, the left and right subtrees of x_i are denoted S_i and S'_i, respectively (see figure 28). Note that we may assume that S'_t has a black root (by choosing x_t appropriately). We install u as the right child of x_t, make S' and T_2 the left and right subtrees of u. We also color u red. The result is an almost red-black tree with possibly an insertion violation at u (if x_t is red). As in the insertion algorithm, we can now perform the rebalancing algorithm to convert an almost red-black tree into a red-black tree. The time to carry out the rebalancing is

$$O(1 + \mathtt{bht}[T_1] - \mathtt{bht}[T_2]) \tag{23}$$

which is $O(\log n)$. This concludes our description of merging.

Splitting. Next consider the problem of splitting a red-black tree T_1 at a key k. This is slightly more complicated, and will use the merging algorithm just described as a subroutine.

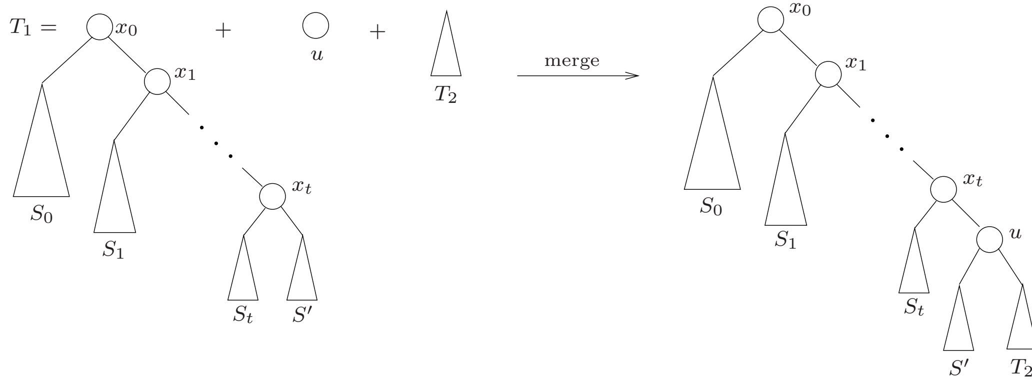


Figure 28: Decomposition of T_1 and merging of T_1, u, T_2.

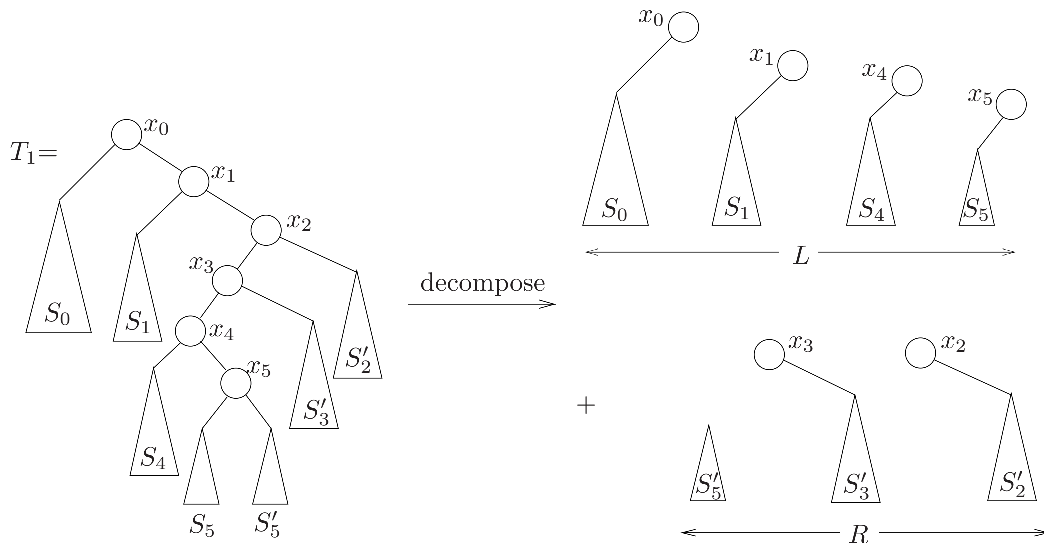


Figure 29: Split: decomposition of T_1.

We first perform a `lookUp` on k, using $O(\log n)$ time. This leads us down a path $(x_0, x_1, \ldots, x_t)$ to a node x_t with key k' that is equal to k if k is in the tree; otherwise it is equal to the predecessor or successor of k in the tree. See figure 29 for a particular case where $t = 5$.

Again, let S_i, S'_i denote the left and right subtrees of x_i. Let us form two collections L and R of RBT's: for each i, if the key in x_i is greater than k, we put x_i (viewed as a RBT with one key) and S'_i into R. Otherwise, we put x_i and S'_i into L. This takes care of every key in T_1, with the possible exception of a subtree of x_t: if x_t is put into R, then we put S_t into L. Otherwise, x_t is put into L and we put S'_t into R. In figure 29, if x_i and S_i are put in L, we display them together as one tree with x_i as root; a similar remark holds if x_i, S'_i are put in R.

Clearly, our task is completed if we now combine the trees in L into a new T_1, and similarly combine the trees in R into a tree which is returned as the value of this procedure call.

Let us focus on the set of trees in L (R is similarly treated). It is not hard to do see that there are $O(\log n)$ trees in L and so we can easily merge them into one tree in $O(\log^2 n)$ time. But in fact, let us now

show that $O(\log n)$ suffices. Let us note that the trees in L can in fact be relabeled and ordered as follows:

$$L_1 > y_1 > L_2 > y_2 > \cdots > L_\ell > y_\ell > L_{\ell+1} \tag{24}$$

where the y_j's are singleton trees (coming from the x_i's), and

$$\mathtt{bht}[L_1] \geq \mathtt{bht}[L_2] \geq \cdots \geq \mathtt{bht}[L_{\ell+1}] \geq 0. \tag{25}$$

Note that $L_{\ell+1}$ could be empty. For instance, the set L in figure 29 is relabeled as in figure 30 with L_5 as an empty tree. The basic idea is to merge the trees from right to left. More precisely: initially, we merge

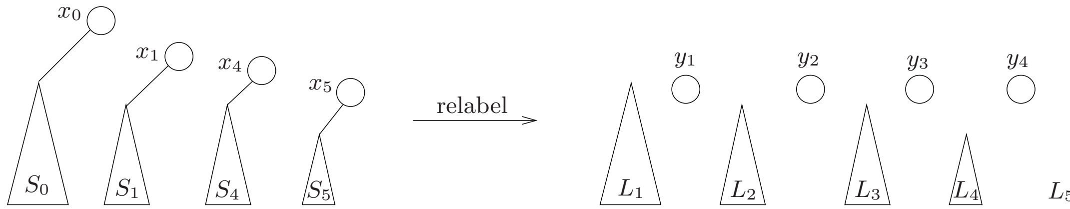


Figure 30: Relabelling the trees in the set L.

$L_\ell, y_\ell, L_{\ell+1}$ and let the result be denoted $L'_{\ell-1}$. Inductively, assume that

$$L_{i+1}, y_{i+1}, \ldots, y_\ell, L_{\ell+1}$$

have been merged into a tree denoted by L'_i. If $i > 0$, we continue inductively by merge

$$L_i, y_i, L'_i \tag{26}$$

to form L'_{i-1}. Otherwise, L'_0 is the final result and we stop. This completes the merge algorithm.

The main result is the following:

Lemma 4 *The time to merge all the trees in L is $O(\mathtt{bht}[L_1] + \ell) = O(\log n)$.*

Verifying this requires some careful analysis but the idea is as follows. The inductive merge step (26) takes time

$$O(1 + |\mathtt{bht}[L_i] - \mathtt{bht}[L'_i]|). \tag{27}$$

Now, if we could assume that $\mathtt{bht}[L'_i] \leq 1 + \mathtt{bht}[L_{i+1}]$ then we could replace (27) by

$$O(2 + \mathtt{bht}[L_i] - \mathtt{bht}[L_{i+1}]).$$

and the overall cost (after telescoping) would be $O(\mathtt{bht}[L_1] + \ell)$,

An $O(\log n)$ bound for splitting. We refer to the collection L (see equations (24) and (25)). Let us call L_i **red** or **black** according as its root is red or black. Notice that if L_i is red, then $\mathtt{bht}[L_{i+1}] > \mathtt{bht}[L_i]$. We claim: for $i = 2, \ldots, \ell$,

$$\mathtt{bht}[L_{i-1}] = \mathtt{bht}[L_i] \quad \Longrightarrow \quad \mathtt{bht}[L_i] > \mathtt{bht}[L_{i+1}].$$

For, if $\mathtt{bht}[L_{i-1}] = \mathtt{bht}[L_i]$ then the root of L_i is a near nephew of the root L_{i-1} and the parent $p = x_i$ of L_i is red. Similarly, if $\mathtt{bht}[L_i] = \mathtt{bht}[L_{i+1}]$, we conclude that the parent $q = x_{i+1}$ of L_{i+1} is red. But p is the parent of q, by the height property. This is a contradiction because now the parent property is violated, proving our claim.

Suppose inductively, for some $h = 1, \ldots, \ell$, we have already merged

$$L_{h+1}, y_{h+1}, L_{h+2}, \ldots, y_\ell, L_{\ell+1}$$

into a RBT denoted L'_h. In case

$$\mathtt{bht}[L'_h] > \mathtt{bht}[L_h]$$

we will say that an **inversion** has occurred at L'_h.

Lemma 5 (Inversion Lemma) *In case of an inversion at L'_h, the following holds.*

(i) $\mathtt{bht}[L'_h] = 1 + \mathtt{bht}[L_h]$.

(ii) *Either L'_h is black or* $\mathtt{bht}[L_{h-1}] > \mathtt{bht}[L_h]$.

Proof. To see this lemma, consider the previous step which combined $L_{h+1}, y_{h+1}, L'_{h+1}$ into L'_h. We use three easy remarks. First, it is clear from the merge algorithm that

$$\mathtt{bht}[L'_h] \le 1 + \max\{\mathtt{bht}[L_{h+1}], \mathtt{bht}[L'_{h+1}]\}. \tag{28}$$

Second, if equality is achieved in (28) then L'_h is black (using a basic property of our rebalancing procedure for removing insertion violations). Third, if $\mathtt{bht}[L'_{h+1}] > \mathtt{bht}[L_{h+1}]$ and L'_{h+1} is black, then (28) is a strict inequality. (Similarly if $\mathtt{bht}[L'_{h+1}] < \mathtt{bht}[L_{h+1}]$ and L_{h+1} is black then we also have a strict inequality.)

There are two cases.
CASE I: there is no inversion at L'_{h+1}, *i.e.*, $\mathtt{bht}[L_{h+1}] \ge \mathtt{bht}[L'_{h+1}]$. Then clearly property (i) holds and L'_h is black (thus satisfying (ii)).
CASE II: there is an inversion at L'_{h+1}. We assume that this inversion satisfies the hypothesis in our lemma. So, either (a) L'_{h+1} is black or (b) $\mathtt{bht}[L_h] > \mathtt{bht}[L_{h+1}]$. In subcase (a), in order to have an inversion at L'_h, our first remark implies that $\mathtt{bht}[L_h] = \mathtt{bht}[L_{h+1}]$. But this means $\mathtt{bht}[L_{h-1}] > \mathtt{bht}[L_h]$, satisfying property (ii). To see property (i), note that remark 3 implies $\mathtt{bht}[L'_h] \le \mathtt{bht}[L'_{h+1}]$ and hence $\mathtt{bht}[L'_h] = \mathtt{bht}[L'_{h+1}]$. By induction, property (i) says that $\mathtt{bht}[L'_{h+1}] = 1 + \mathtt{bht}[L_{h+1}] = 1 + \mathtt{bht}[L_h]$.

In subcase (b), property (i) follows because

$$\begin{aligned}
\mathtt{bht}[L'_h] &\le 1 + \mathtt{bht}[L'_{h+1}] \\
&\le 2 + \mathtt{bht}[L_{h+1}] \quad \text{(by induction)} \\
&\le 1 + \mathtt{bht}[L_h] \quad \text{(subcase b)} \\
&\le \mathtt{bht}[L'_h] \quad \text{(inversion assumption).}
\end{aligned}$$

Property (ii) holds since L'_h is black by the third remark. **Q.E.D.**

Since $1 + \mathtt{bht}[L_h] \ge \mathtt{bht}[L'_h] \ge \mathtt{bht}[L_{h+1}]$, the cost of combining L_h, y_h, L'_h into L'_{h-1} is

$$O(1 + \mathtt{bht}[L_h] - \mathtt{bht}[L_{h+1}]).$$

Summing up for $h = 1, \ldots, \ell$, we obtain the bound $O(\ell + \mathtt{bht}[L_1] - \mathtt{bht}[L_{\ell+1}]) = O(\log n)$. This concludes our proof.

Exercises

Exercise 10.1: Let T_1 be the tree obtained in Exercise 6.1.
(a) Merge this with the tree T_2 with two keys: 12 and 11. The root of T_2 is 12, assumed to be black.
(b) Now split the tree obtained in (a) at the key 6. ♢

Exercise 10.2: Our $O(\log n)$ bound for merging the $O(\log n)$ red-black trees in the split algorithm has fairly tight "constants" because of the inversion lemma. Give a simpler proof of an $O(\log n)$ bound by using only the assumptions of equations (24) and (25). That is, do not assume that the trees came from any particular process so that they have certain properties. ♢

End Exercises

§11. Conclusion

We conclude by mentioning some additional themes which we do not have time to explore.

§12. Weight-Balanced Trees

We specify that the ratio $s_L : s_R$ of the sizes of the two subtrees at any node should lie between $1/\beta$ and β where $\beta > 1$ is fixed for the family. It is immediate to see that this defines a balanced family of trees. This is more flexible than height balanced trees, and this is important for some applications. The price we pay is that we need up to $\lg n$ bits of balance information at each node.

Exercises

Exercise 12.1: Since in practice we need to reserve 2 bits of information per node in an AVL tree, let us try to take full advantage of this. Consider AVL trees in which the balance information at each node is $b = -1, 0, 1, 2$. Here b is the height of the left subtree minus the height of the right subtree. What are the advantages of this new flexibility? ♢

Exercise 12.2: Work out the details about how to use only one balance bit per AVL node. ♢

Exercise 12.3: Design a one-pass algorithm for AVL insertion and AVL deletion. ♢

Exercise 12.4:
(a) Show how to maintain the min-heap property in a binary tree under `insert` and `deleteMin`.
(b) Modify your solution in part (a) to ensure that the depth of the binary tree is always $O(\log n)$ if there are n items in the tree. ♢

Exercise 12.5: Suppose T is a binary tree storing items at each node with the property that at each internal node u,

$$u_L.\texttt{Key} < u.\texttt{Key} < u_R.\texttt{Key},$$

where u_L and u_R are the left and right children of u (if they exist). So this is weaker than a binary search tree. For simplicity, let us assume that T has exactly $2^h - 1$ nodes and height $h-1$, so it is a complete binary tree. Now, among all the nodes at a given depth, we order them from left to right in the natural way. Then, except for the leftmost and rightmost node in a level, every node has a successor and predecessor node in its level. One of them is a sibling. The other is defined to be its **partner**. For completeness, let us define the leftmost and rightmost nodes to be each other's partner. See figure 31. Now define the following parallel operations. The first operation is *sort*1: at each internal node u at

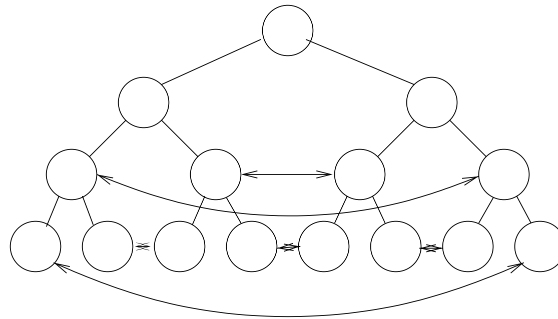


Figure 31: Partners

an odd level, we order the keys at u and its children u_L, u_R so that $u_L.\mathtt{Key} < u.\mathtt{Key} < u_R.\mathtt{Key}$. The second is *sort*2, and it is analogous to *sort*1 except that it applies to all internal nodes at an even level. The third operation is *swap* which order the keys of each pair of partners (by exchanging their keys if necessary). Suppose we repeatedly perform the sequence of operations $(sort1, sort2, swap)$. Will this eventually stabilize? If we start out with a binary search tree then clearly this is a stable state. Will we always end up in a binary search tree? ♢

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.