Lecture III
BALANCED SEARCH TREES

It is said that there are a dozen Eskimo words for snow. The tree data structure is the computer science equivalent of snow, and we have as many ways to classify them. The simplest is the binary search tree which 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 $\lg n$. We say that a family of binary trees is **balanced** if for each n, every tree in the family on n nodes has height $O(\log n)$. Naturally, we also insist that there is least one tree in the family with n nodes. The implicit constant in the big-Oh notation here depends on the particular family of search trees. What makes any particular family interesting is that they come equipped with specialized algorithms to insert and delete items from trees in the family, while preserving membership in the family.

Many balanced families have been invented in computer science. These 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 it is too inflexible for certain 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**. Tarjan [3] gives a brief history of some balancing schemes.

Teaching Note: all our algorithms for search trees are described in such a way that they can be easily internalized. We expect students to be able to do hand-simulations for concrete examples. We do not provide any computer code, but once these algorithms are clearly understood, there should be no difficulty in 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 some basic terminology for such search structures.

Search structures stores a set of objects which we call **items**, and the location[1] of the items in the structures are called **nodes**. We may loosely identify the notion of "nodes" with pointers or references in conventional programming languages. 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, we write $u.\mathtt{Key}$ and $u.\mathtt{Data}$ 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 pair (n, i) where n is a node, i is the next iterator. The node n points to an, and i points to the next iterator. If $i = \mathsf{nil}$ then[2] there is no next item.

[1] A related programming notion called **locatives** is introduced by H. R. Lewis and L. Denenberg in **Data Structures and Their Algorithms**, Harper Collins Publishers, 1991. The proper treatment of pointers, references, locatives would take us into programming semantics.

[2] We are implicitly thinking of an iterator i as a node/pointer in this notation. However, we have automatically performed a dereferencing when we use i as an iterator (cf. locatives).

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.

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). It is also natural to refer such structures as **keyed search structures**.

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. In this case, our terminology of nodes is an unfortunate one.

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 N in S such that the item in N has key K. If no such node exists, it returns $N = \mathsf{nil}$. 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

Search structures such as binary trees can support any subset of the following operations, put into four groups:

(I)	$\texttt{make}(\cdot) \rightarrow \mathit{Structure}$
	$\texttt{kill}(\mathit{Structure})$
(II)	$\texttt{lookUp}(\mathit{Structure}, \mathit{Key}) \rightarrow \mathit{Node}$,
	$\texttt{insert}(\mathit{Structure}, \mathit{Item}) \rightarrow \mathit{Node}$,
	$\texttt{delete}(\mathit{Structure}, \mathit{Node})$,
(III)	$\texttt{list}(\mathit{Structure}) \rightarrow \mathit{Node}$,
	$\texttt{succ}(\mathit{Structure}, \mathit{Node}) \rightarrow \mathit{Node}$,
	$\texttt{pred}(\mathit{Structure}, \mathit{Node}) \rightarrow \mathit{Node}$,
	$\texttt{min}(\mathit{Structure}) \rightarrow \mathit{Node}$,
	$\texttt{max}(\mathit{Structure}) \rightarrow \mathit{Node}$,
	$\texttt{deleteMin}(\mathit{Structure}) \rightarrow \mathit{Item}$,
(IV)	$\texttt{split}(\mathit{Structure}, \mathit{Key}) \rightarrow \mathit{Structure}1$,
	$\texttt{merge}(\mathit{Structure}1, \mathit{Structure}2)$,
(V)	$\texttt{lookUpAll}(\mathit{Structure}, \mathit{Key}) \rightarrow \mathit{Node}$,
	$\texttt{deleteAll}(\mathit{Structure}, \mathit{Node})$.

The meaning of these operations are quite intuitive. We briefly explain them. Let S, S' be search structures, K be a key and N a node.

(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 next three operations constitute the "dictionary operations". The node N returned by $\texttt{lookUp}(S, K)$ 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 as needed. 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 '$\texttt{delete}(S, \texttt{lookUp}(S, K))$'.

(III) The operation $\texttt{list}(S)$ returns a node that can be regarded as the beginning of a list that contains all the items in S in *some arbitrary* order. So the ordering of keys is not used. In the programming literature, the node returned by `list` is sometimes called an **iterator** because we can use it to iterate through all the desired elements. The remaining operations in this group depend on the ordering properties of keys. The $\texttt{min}(S)$ and $\texttt{max}(S)$ operations are obvious. The successor $\texttt{succ}(S, N)$ (resp., predecesssor $\texttt{pred}(S, N)$) of a node N refers to the node in S whose key has the next larger (resp., smaller) value. This is undefined if N has the largest (resp., smallest) value in S.

Note that $\texttt{list}(S)$ can be implemented using $\texttt{min}(S)$ and $\texttt{succ}(S, N)$ or $\texttt{max}(S)$ and $\texttt{pred}(S, N)$. Such a listing has the additional property of sorting the output.

The operation $\texttt{deleteMin}(S)$ 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.

(IV) If $\texttt{split}(S, K) \rightarrow S'$ then all the items in S with keys greater than K are moved into a new structure S'; the remaining items are retained in S. In the operation $\texttt{merge}(S, S')$, 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.

(V) We introduce two variants of `lookUp` and `delete`, when keys are not unique. In this case, we may want to lookup or delete all items with a given key. The `lookUpAll` variant returns a node, that is the start of a linked list of all the items with that key. Thus the `lookUpAll` function can be regarded as returning an iterator. The operation `deleteAll` deletes all items with the specified key.

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. The operations `make` and `kill`(from group (I)) are assumed to be present in each ADT.

- **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.

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. Minor variant interpretations of all these operations need not concern us. 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 '`make`$(\cdot) \to S$; `insert`(S, I)'.

Exercises

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

Exercise 2.1: Describe algorithms to implement all of the above operations where the concrete data structure are (a) linked lists, (b) arrays. ♢

§3. Binary Search Tree Operations

We focus on the concrete data type of binary search trees. We now show that binary search trees can support all the above operations. Our approach to deletion is different from the conventional binary search tree deletion algorithms (but we will point out the difference).

Recall the basic properties of **binary trees** (see Appendix of Lecture I). A **binary search tree** T is a binary tree in which we store a key u.`Key` at each node u, subject to the **binary search tree property**: the key u.`Key` at node u is no smaller than any key in the left subtree below u, and no larger than any key in the right subtree below u.

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 $u.\texttt{Key} = K$, we are done. Otherwise, either $u.\texttt{Key} < K$ or $u.\texttt{Key} > K$. In the former case, we recursively search the left subtree of u; otherwise, we recurse in the right subtree of u.

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. Otherwise, we reach a leaf node u. Then the item can be inserted as the left child of u if $u.\texttt{Key} > K$, and otherwise it can be inserted as the right child of u. In any case, the inserted item is a new leaf of the tree.

Rotation. This is not a listed operation in §2. It is an equivalence operation. By itself, rotation does not appear useful. However, we see that most operations on binary search trees are easily reduced to repeated rotations.

Assume u is a non-root node in a binary search tree T. The operation `rotate`(u) amounts to the following transformation of T (see figure 1).

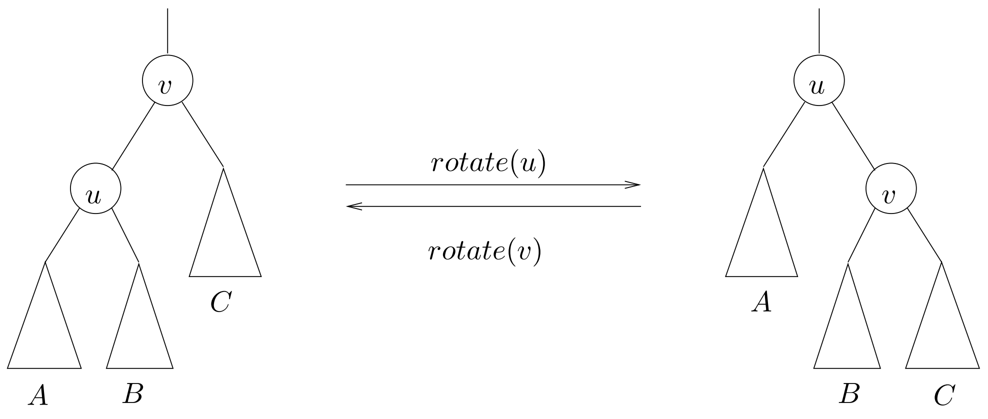


Figure 1: Rotation at u and its inverse.

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

$$A < B < C$$

in the sense that each key in A is less than any key in B, etc. Actually, only parent of the root of B has switched from u to v. Notice that after `rotate`(u), the former parent of v (not shown) will now have u instead of v as a child. Clearly the inverse of `rotate`(u) is `rotate`(v). The explicit pointer manipulations for a rotation are left as an exercise. After a rotation at u, the depth of u is decreased by 1. Note that `rotate`(u) followed by `rotate`(v) is the identity[4] operation, as illustrated in figure 1.

The above description assumes that u is a non-root. In case u is a root, we shall define `rotate`(u) to be a null operation (or identity transformation). In any case, a rotation is an equivalence transformation because the result of a rotation is equivalent to the original structure.

Implementation of rotation: by looking at figure 1, it is easily seen that there are links to be modified when we perform a rotation at u. Then the links to be modified are: (1) u.`Left`, (2) u.`Parent`, (3) v.`Right`, (4) v.`Parent`, (5) w.`Left` or w.`Right` where w is the current parent of v. Thus this operation takes $O(1)$ time.

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

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

It is not hard to modify all our rotation-based algorithms to use the left- and right-rotation formulation if we do not have parent pointers. However, this would be less elegant pedagogically.

Variation on LookUp. Let us first demonstrate the use of rotations on a problem we already know how to solve: the lookup algorithm. This time, we operate as follows: starting at the root r, we first check if the key k we want is at the root. If not, and if $k > r.Key$. then we perform `rotate`$(u.$`Right`$)$ and otherwise, we perform `rotate`$(u.$`Left`$)$. Now we repeat with r set to the new root. We stop with success when the desired key appear at the root, or with failure when we attempt to perform a rotate on a non-existent child. The nice thing about this algorithm is that we are constantly changing the shape of the tree, and under certain randomness assumptions, we can expect the tree to have expected logarithmic height. The keys we are interested in tend to be near the root (as this is a variation of the "move to front" rule that we will investigate later).

Extremal paths and Spines. The **left-path** and **right-path** of a node u is simply the path that starts from u and keeps moving towards the left or right child until we cannot proceed further. The last element of this path is therefore the mininum or maximum item in the subtree at u. 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 trivial path $(u, \mathrm{rightpath}(u.\texttt{Left}))$. In case $u.\texttt{Left} = \mathsf{nil}$, the left spine is just the trivial path (u) of length 0. The **right-spine** is similarly defined. We have defined 4 paths relative to u. The last elements of each of these paths are known as **tips** of the respective paths, and they have significance in binary search trees: thus, the

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

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, corresponds to the successor and predecessor of u. Clearly, u is a leaf iff all these four tips are identical and equal to u.

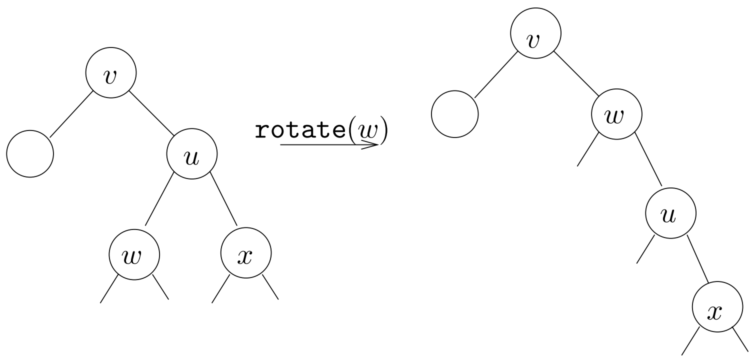


Figure 2: Reduction of the left-spine of u after $\texttt{rotate}(u.\texttt{Left}) = \texttt{rotate}(w)$.

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 2. 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 path from the root to $u = v_m$. After performing* $\texttt{rotate}(u.\texttt{Left})$*,*
(i) the left-spine of u becomes $(u_0, u_2, \ldots, u_k)$ of length $k-1$, and
(ii) the path from the root to 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 path from the root to 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. It is easy to describe an algorithm for deleting a node u from a binary search tree:

```
DELETE($T, u$):
Input:    $u$ is node to be deleted from $T$.
Output:   $T$, the tree with $u$ deleted.
    while $u$.Left $\neq$ nil do
        rotate($u$.Left).
    while $u$.Right $\neq$ nil do
        rotate($u$.Right).
    Delete node $u$ (which is now a leaf).
```

A more efficient alternative is to avoid one of the two while loops: after the first loop, u has at most one child. We can now delete u, and if u has any child v, we make v the new child of the parent of u. Now we are done. 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 second loop instead of the first loop.

Let us contrast this with the following **standard deletion algorithm**. To delete u, one considers three cases:
(i) If u has only one child v and has a parent p we make p point to v instead of u (so effectively deleting u).
(ii) If u is a leaf or u has only one child but no parent then we just delete u.
(iii) 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 the node u may not be physically removed, only the item represented by u is removed. Furthermore, *the node that is physically removed has at most one child*. In applications where rotations may be expensive (because they force subsidiary book-keeping actions), this standard algorithm may be preferred. The main attraction of our rotation-based deletion is its simplicity.

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 $\neq$ nil then In-order($u$.Left).
    2.   Visit($u$).
    3.   If $u$.Right $\neq$ 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 3. The numbers on the nodes are not keys, but serve as identifiers.

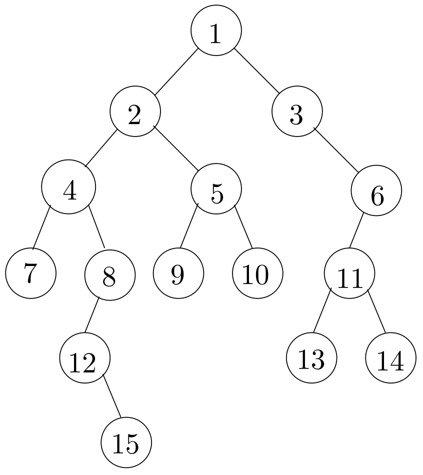


Figure 3: Binary tree.

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

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

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 3, 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

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

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. Suppose T is a binary search tree and K is any key. Then the **successor** of K in T is defined to be the least key in T that is larger than K. Again, K may have no successor. We similarly define the **predecessor** of K in T.

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 assumption for binary trees is not to assume 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 $\neq$ nil {return the tip of the right-spine of $u$}
    1.1       $v \leftarrow u$.Right;
    1.2       while $v$.Left $\neq$ nil, $v \leftarrow v$.Left;
    1.3       return($v$).
    2.    else {return the $v$ where $u$ is the tip of the left-spine of $v$}
    2.1       $v \leftarrow u$.Parent;
    2.2       while $v \neq$ nil and $u = v$.Left,
    2.3           $(u, v) \leftarrow (v, v$.Parent$)$.
    2.4       return($v$).
```

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

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

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

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

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

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 4.

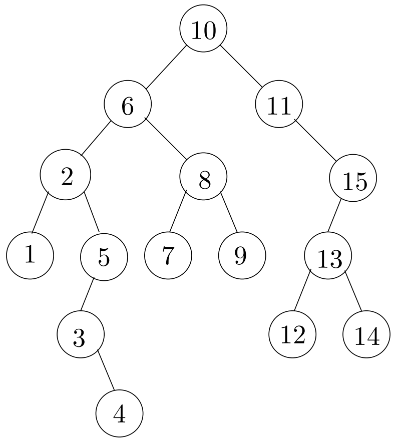


Figure 4: A binary search tree.

(a) Perform the operation `split`$(T, 5) \to T'$ Display T and T' after the split.
(b) Now perform `insert`$(T, 9.5)$ where T is the tree after the operation in (a). Display T after the insertion.
(c) Finally, perform `merge`(T, T') where T is the tree after the insert in (b) and T' is the tree in (a). ♢

Exercise 3.3: (a) 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. HINT: use induction.

NOTE: this shows that rotation is a "universal" equivalence transformation.
(b) If the trees in part(a) has n nodes each, what is an upper bound on the number of rotations that are necessary for the transformation $T_0 \to T_1$? ♢

Exercise 3.4: 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 3.5: 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 3.6: 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 3.7: 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 3.8: 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? ♢

End Exercises

§4. Variations on a Theme trees

We introduce an alternative view of binary trees, called **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[5] **nil nodes** while the other nodes are called **non-nil nodes**. See figure 5(a) for a standard binary tree and figure 5(b) for the corresponding extended version. In this figure, we see a common convention of representing nil nodes by black squares.

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

[5]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".

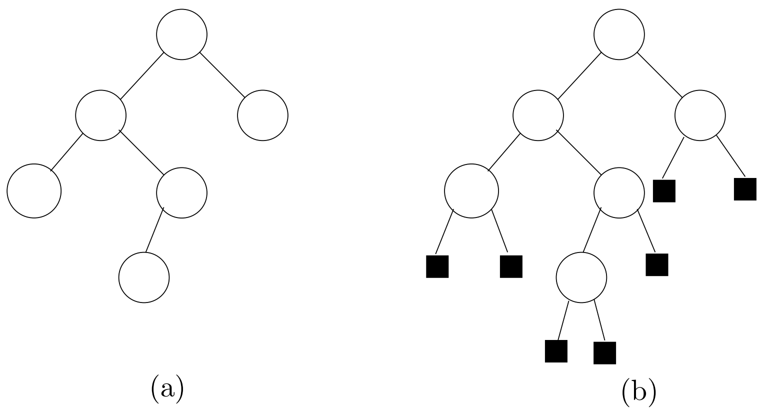


Figure 5: Binary Trees: (a) standard, (b) extended.

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 when we can 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 this term so as to be consistent with the corresponding 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 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 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[6] Tarjan [3], 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

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

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 yet 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 these information are independent, while others are 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 a derived information is **strongly local** if 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. ♢

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.

$\diamondsuit$

End Exercises

§5. AVL Trees

AVL trees is the first known example of balanced trees, and is one of the simplest to understand. 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 right subtree minus the height of the left subtree. The node is **perfectly balanced** if the balance is 0. It is **AVL-balanced** if the balance is either $-1, 0$ 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, an AVL tree will reserve 2 bits per node for the balance information (but see exercise). Not surprisingly, the algorithms for AVL trees are even easier than the ones for red-black trees.

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. Clearly, $\mu(1) = 2$. But is $\mu(0)$ 0 or 1? To resolve this, we make a convention to answer this question, and to make our induction as simple as possible: we shall define the **height** of an empty tree to be -1. Therefore $\mu(0) = 1$ and $\mu(-1) = 0$. In general, $\mu(h)$ clearly satisfy the following recurrence:

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

This corresponds to the minimum size tree of height h having left and right subtrees which are minimum size trees of heights $h-1$ and $h-2$. For instance, $\mu(2) = 1 + \mu(1) + \mu(0) = 1 + 2 + 1 = 4$. See figure 6 for the smallest AVL trees of the first few values of h.

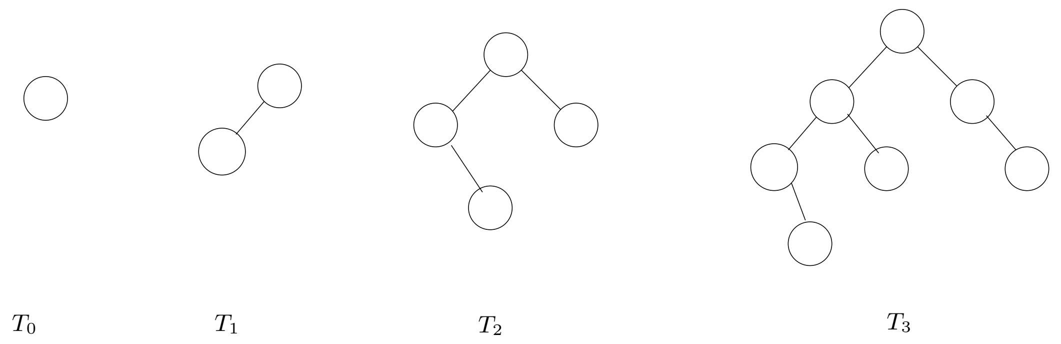


Figure 6: Smallest AVL trees of heights 1, 2 and 3.

The claim that AVL trees are balanced is a consequence of

$$\mu(h) \geq C^h, \qquad (h \geq 1) \tag{2}$$

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

$$n \geq \mu(h)$$

which, by (2), implies $n \geq C^h$. Taking logs, we obtain $\log_C(n) \geq h$ or $h = O(\log n)$. It is easy to establish (2) with $C = 2^{1/2} = \sqrt{2}$: from (1), we have $\mu(h) \geq 2\mu(h-2)$ for $h \geq 1$. Then it is easy to see by inductin that $\mu(h) \geq 2^{h/2}$ for all $h \geq 1$. 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:

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

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

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

We conclude that any AVL tree with n nodes has height at most $\log_\phi(n) = \log_\phi 2 \ln n$ 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.

REBALANCE PHASE: Let x be the node that was just inserted or just deleted. We now retrace the path from x towards the root, rebalancing nodes along this path as necessary.

In fact, this scheme works for most of the balanced tree schemes in the literature. Let us discuss the rebalance phase. It is clear that any node u that is unbalanced along this path has balance of ± 2. We will show that our rebalancing preserves this invariant. Therefore, by symmetry, we may suppose that the current unbalanced node u has balance 2. Suppose its left child is node v and has height $h+1$. Then its right child v' has height $h-1$. See figure 7.

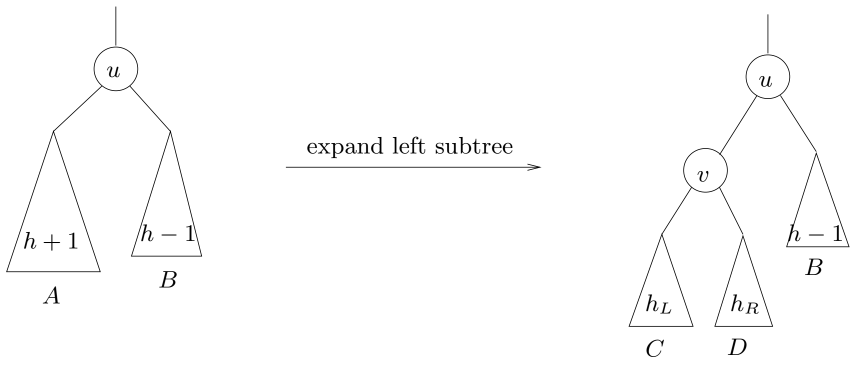


Figure 7: Node u is unbalanced after insertion or deletion.

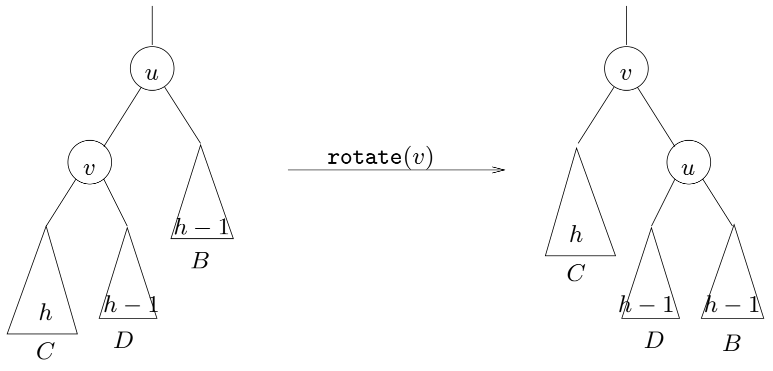
By induction, all the proper descendents of u are balanced. The current height of u is $h+2$. In any case, let the current heights of the children of v be h_L and h_R, respectively.

Insertion Rebalancing. Suppose that this unbalance 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)

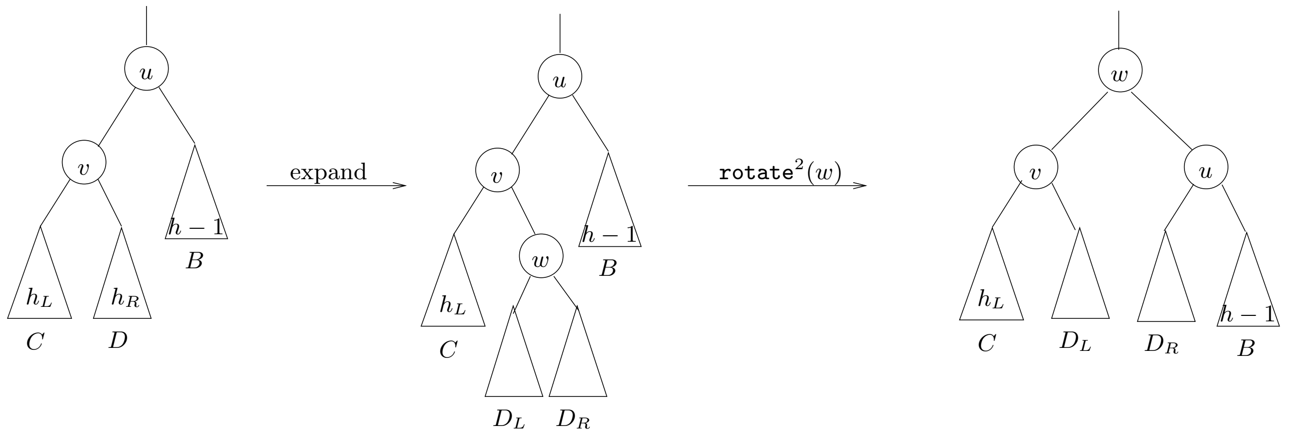
$$h+1, h, h-1.$$

The inserted node x must be in the subtree rooted at v. Clearly, the heights of the children of v satisfies $\max(h_L, h_R) = h$. 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 v. Therefore, we only 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)



CASE (II)

Figure 8: CASE (I): $\texttt{rotate}(u)$, CASE (II): $\texttt{rotate}^2(w)$.

CASE (I.2): $h_L = h-1$ and $h_R = h$. This means the inserted node is in the right subtree of v. In this case let us expand the subtree D and let w be its root. The two children of w will have heights of $h-1$ and $h-1-\delta$ ($\delta = 0, 1$). It turns out that it does not matter which of these is the left child (despite the apparent assymetry of the situation). If we double rotate w (*i.e.*, $\texttt{rotate}(w), \texttt{rotate}(w)$), 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 unbalance in figure 7 comes from a deletion. The previous heights of u, v, v' must have been

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

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

CASE (D.1): $h_L = h, h_R = h-1$. 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 $h+1$. Since the original height is $h+2$, we have to continue checking for balance further up the path to the root.

CASE (D.2): $h_L = h-1, h_R = h$. This is like case (I.2) and treated the same way, by performing a double rotation at w. Again, this is not a terminal case.

CASE (D.3): $h_L = h_R = h$. This case is new. But we simply rotate at v. We also check that 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.

One-Pass Algorithms. The above algorithms requires two passes along the path from the root to x. We can provide a one-pass version of these algorithms, possibly at the cost of extra rotations. The idea is to perform "pre-emptive rotations". Recall that in insertion or deletion of a key K, we first perform a `lookUp`(K). We perform a variation of `lookUp` which will perform rotations.

The variation on `lookUp` depends on whether we are doing insertion or deletion. First consider the case of insertion. Suppose we are currently at some node u. If we found K, then we are done with `lookUp`(K). Otherwise, we have to go down to the left or right child of u. Let v be this child. If v is non-existent (i.e., nil) then we are also done. Otherwise, *if v has height greater than its sibling, then we may have to perform a preemptive rotations.* In the case of deletion, the condition is just the reverse: *if v has height less than its sibling, then we may have to perform preemptive rotations.*

Preemptive Insertion Rotation. Suppose that we are at a node u during an insertion, and we are about to proceed to a child node v. If the height h of v is less than or equal to that of its sibling, there is no need for pre-emptive rotation. So assume that the height of the sibling of v is $h-1$. We examine further to see that our search will next proceed to a child w of v. There are two possibilities. (A) First, suppose w is an outer grandchild of u. See figure **??**(a). In this case, we perform a rotation at v. (B) If w is an inner grandchild of u, then we perform a double rotation at w. See figure **??**(b). Define the node x as follows: in case (A), let $x = w$ and in case (B) let x be the child of w that we next visit. In either case, we can check that the height of x is at most that of its sibling. Thus we have prevented any need for rotation when the height of x is subsequently increased by 1.

Preemptive Deletion Rotation. Suppose that we are at a node u during an deletion, and we are about to proceed to a child node x. If x has height at least that of its sibling, we need no pre-emptive rotation. So