Lecture III
BALANCED SEARCH TREES

It is said[1] that there is a dozen Eskimo words for snow. The tree data structure is the computer science equivalent of snow. The simplest tree 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 $\lfloor \lg n \rfloor$. We say that a family of binary trees is **balanced** if for each n, every tree in the family on n nodes has height $O(\log n)$. The implicit constant in the big-Oh notation here depends on the particular family of search trees. Such a family is useful if comes equipped with efficient algorithms to insert and delete 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 [**?**] gives a brief history of some balancing schemes.

STUDY GUIDE: all search tree algorithms are described in such a way that they could be internalized, and students are expected to carry out hand-simulations for concrete examples. We do not provide computer code, but once these algorithms are internalized, you should be able 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 some basic terminology for such search structures.

Search structures stores a set of objects which we call **items**, and the location of the items in the structures are called **nodes**. We may identify the notion of "nodes" with pointers [2] in the the programming 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, 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 pair (n, i) where n is a

[1] Some calls this claim a "myth", raising issues about how the count should be done, which Eskimo language is meant, etc. But these difficulties are generic, and just pointing out their existence alone does not discredit the claim. It is certainly possible to clarify in what sense the claim can be understood.

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

node, i is the next iterator. The node n points to an item, and i points to the next iterator. If $i = \mathsf{nil}$ then[3] there is no next item.

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 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.

[3] 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).

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, put into five groups:

(I) `make`($\cdot$)$\rightarrow$*Structure*
`kill`(*Structure*)
(II) `lookUp`(*Structure*,*Key*)$\rightarrow$*Node*,
`insert`(*Structure*,*Item*)$\rightarrow$*Node*,
`delete`(*Structure*,*Node*),
(III) `list`(*Structure*) $\rightarrow$*Node*,
`succ`(*Structure*,*Node*)$\rightarrow$*Node*,
`pred`(*Structure*,*Node*)$\rightarrow$*Node*,
`min`(*Structure*)$\rightarrow$*Node*,
`max`(*Structure*)$\rightarrow$*Node*,
`deleteMin`(*Structure*)$\rightarrow$*Item*,
(IV) `split`(*Structure*,*Key*)$\rightarrow$ *Structure*1,
`merge`(*Structure*1,*Structure*2),
(V) `lookUpAll`(*Structure*,*Key*)$\rightarrow$*Node*,
`deleteAll`(*Structure*,*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 `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 '`delete`$(S,$ `lookUp`$(S, K))$'.

(III) The operation `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 `min`(S) and `max`(S) operations are obvious. The successor `succ`(S, N) (resp., predecesssor `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 `list`(S) can be implemented using `min`(S) and `succ`(S, N) or `max`(S) and `pred`(S, N). Such a listing has the additional property of sorting the output.

The operation `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 `split`$(S, K) \to S'$ 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, 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.

- **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[4] 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

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

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 '`make`$(\cdot) \to S$; `insert`(S, I)'.

EXERCISES

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 Trees

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

Recall the basic properties of binary trees (Appendix of Lecture I). A binary tree T is called a **binary search tree** (BST) if 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.

In normal implementations, a node u will have two pointers u.`Left` and u.`Right`. Then u is a leaf iff both these pointers are null.

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.`Key` $= K$, we are done. Otherwise, either u.`Key` $< K$ or u.`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.`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.

More precisely, with the normal implementations, we insert key K as a right child of a leaf u in three steps: we create a new node v, make u.`Right` point to v, and set v.`Key` $= K$.

Rotation. This is not a listed operation in §2. It is an equivalence operation. By itself, rotation does not appear useful. Yet it will be the key to all our operations. 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).

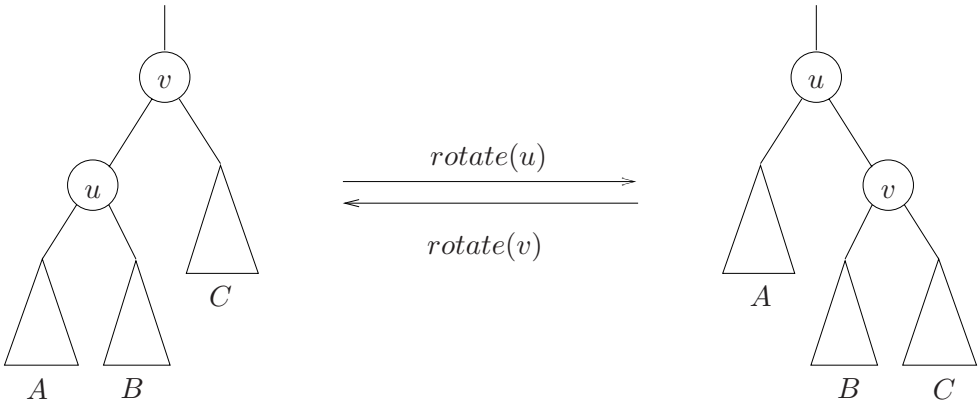

Figure 1: Rotation at u and its inverse.

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[5] 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.

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.

Extremal paths and Spines. Relative to a node u, we now introduce 5 paths. The first is the unique path from u to the root. This path goes upwards (we imagine the root at the top of the tree – this is the opposite of roots in biological trees). Next we introduce 4 downward paths from u. 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 path $(u, \text{rightpath}(u.\texttt{Left}))$. In case $u.\texttt{Left} = \texttt{nil}$,

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

the left spine is just the trivial path (u) of length 0. The **right-spine** is similarly defined. The last node in each of these downward paths are known as **tips** of the respective paths; they are significant in binary search trees. 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.

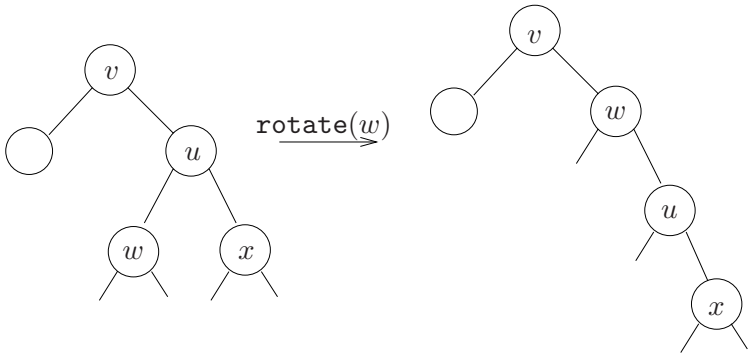

Figure 2: Reduction of the left-spine of u after `rotate`$(u$.`Left`$)$ = `rotate`(w).

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* `rotate`$(u$.`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 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. 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 applications where rotations is undesirable, this standard algorithm is preferred.

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 3. The numbers on the nodes are not keys, but serve as identifiers.

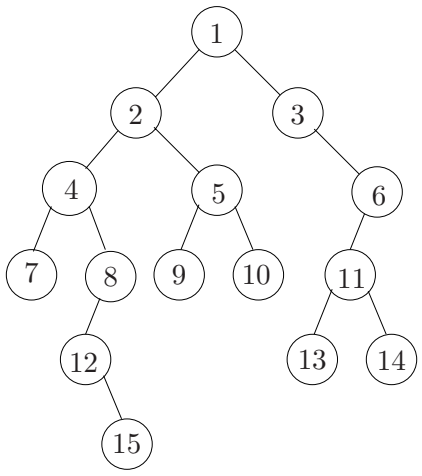

Figure 3: Binary tree.

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 $\texttt{succ}(u)$ and $\texttt{pred}(u)$ denotes the successor and predecessor of u. Of course, $\texttt{succ}(u)$ (resp., $\texttt{pred}(u)$) 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 $u.\texttt{Succ}$ and $u.\texttt{Pred}$. 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) $u.\texttt{Right} = \textsf{nil}$ 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 $u.\texttt{Right} = \textsf{nil}$ and u is the tip of the left-spine of v, then $\texttt{succ}(u) = v$.
(iii) If $u.\texttt{Right} \neq \textsf{nil}$ then $\texttt{succ}(u)$ is the tip of the right-spine of u.

It is easy to derive an algorithm for $\texttt{succ}(u)$ 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 the 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 $\texttt{succ}(u) = \textsf{nil}$ then u is the last node in the in-order traversal of the tree (so u has no successor). The algorithm for $\texttt{pred}(u)$ 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 $u.\texttt{Left} = \texttt{nil}$. Similarly, perform right rotations at u until $u.\texttt{Right} = \texttt{nil}$. 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.

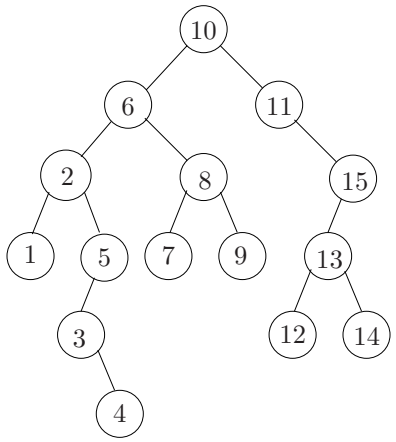

Figure 4: A binary search tree.

(a) Perform the operation $\texttt{split}(T, 5) \to T'$. 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: (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? ♢

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

Exercise 3.10: 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. 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 right subtree minus the height of the left 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. 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 5 for the smallest AVL trees of the first few values of h.

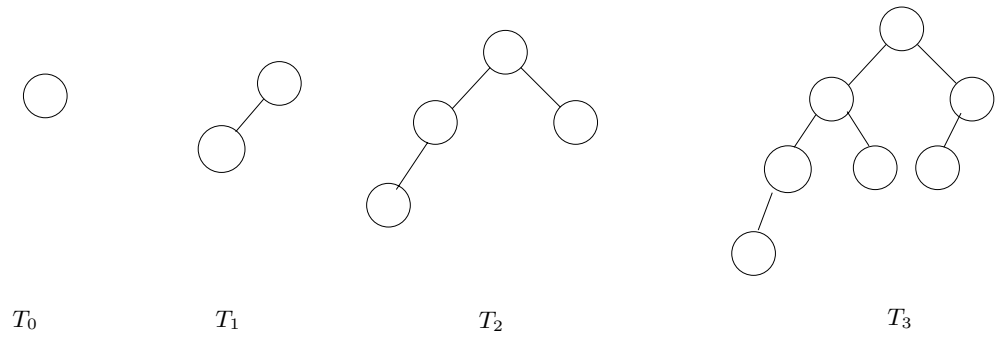

Figure 5: Smallest AVL trees of heights 0, 1, 2 and 3.

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

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

So any AVL tree with n nodes and height h must satisfy the inequality $\phi^h \leq n$ or $h \leq (\lg n)/(\lg \phi)$. 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.

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 6.

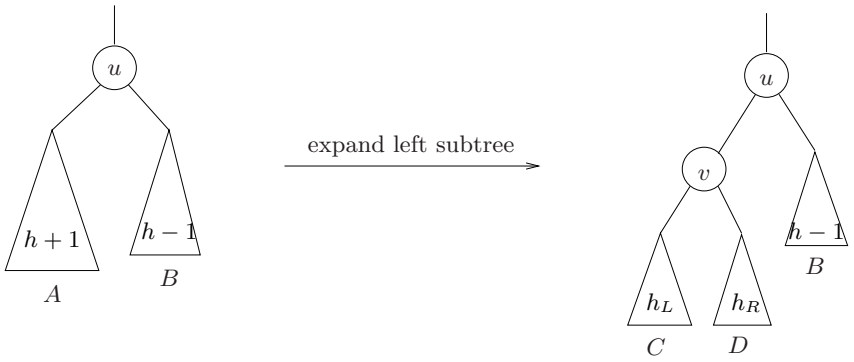

Figure 6: Node u is unbalanced after insertion or deletion.

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 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 $h+1$.

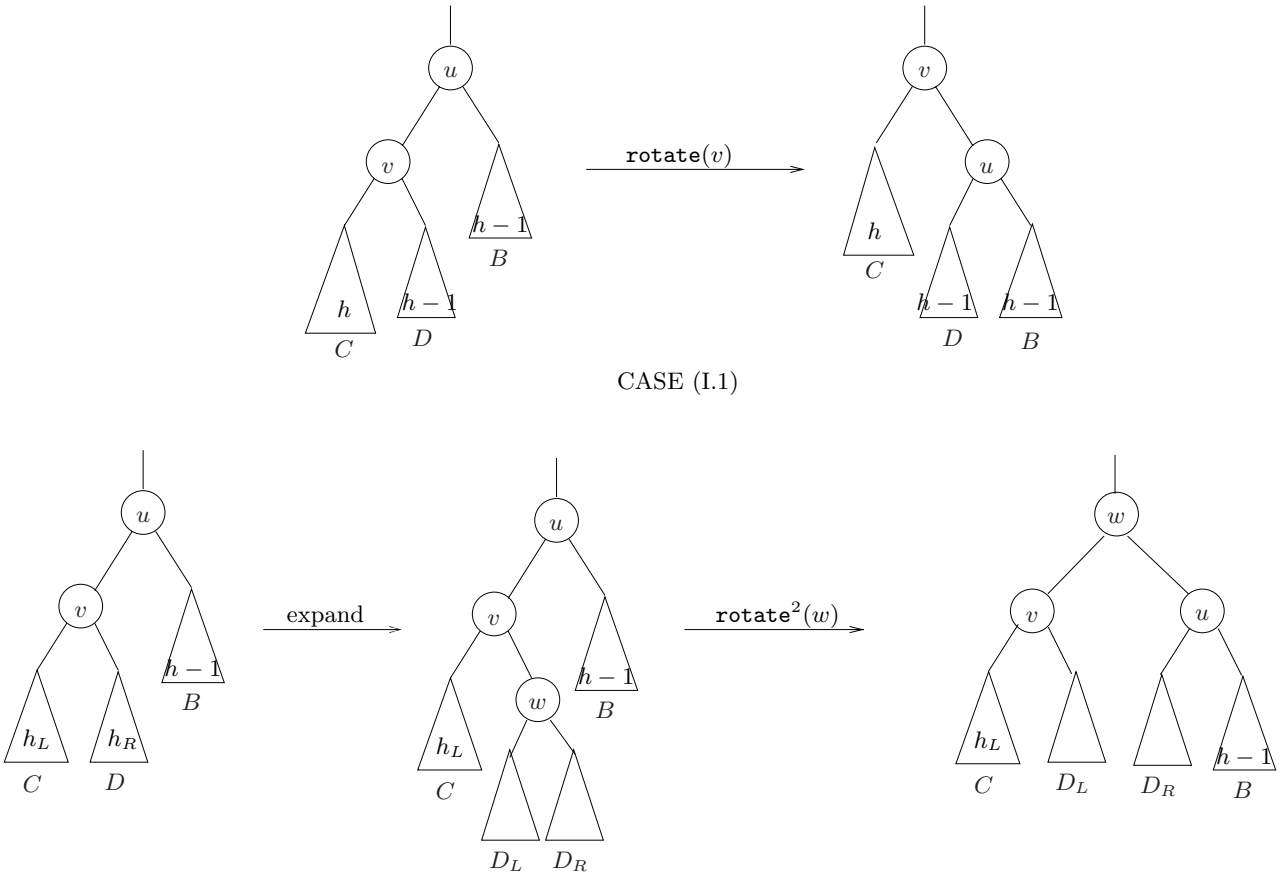

Figure 7: CASE (I.1): $\texttt{rotate}(u)$, CASE (I.2): $\texttt{rotate}^2(w)$.

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 6 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 assume x has height $h - 1$ and its sibling has height h. Let v be the sibling of x. Moreover, the height of v is one more than its sibling. Let w be the child of v that is an extreme grandchild of u. There are 2 cases: (A') the height of w is $h - 1$. In this case, we perform a rotation at v. See figure **??**(a). (B') the height of v is $h - 2$. In this case, we perform a double rotation at the sibling of w. See figure **??**(b). We verify that in either case, the sibling of x after the rotation has height at most $h - 1$. This means that a deletion below x will not violate the AVL property.

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

Exercises

Exercise 4.1: What is the minimum number of nodes in an AVL tree of height 10? ♢

Exercise 4.2: 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 4.3: Draw an AVL T of minimal number of nodes such that the following is true: there is a node x in T such that if you delete this node, the AVL rebalancing will require two "X-rotations". By "X-rotation" we mean either a "single rotation" or a "double rotation". Draw T and the node x. ♢

Exercise 4.4: 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 4.5: 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 4.6: 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 4.7: Improve the lower bound $\mu(h) \geq \phi^h$ by taking into consideration the effects of "+1" in the recurrence $\mu(h) = 1 + \mu(h-1) + \mu(h-2)$.
(a) Show that $\mu(h) \geq F(h-1) + \phi^h$ where $F(h)$ is the h-th Fibonacci number. Recall that $F(h) = h$ for $h = 0, 1$ and $F(h) = F(h-1) + F(h-2)$ for $h \geq 2$.
(b) Further improve (a). ♢

Exercise 4.8: Relationship between Fibonacci numbers and $\mu(h)$.
(a) (Jia-Suen Lin) Show that $\mu(h) = 3F(h) + 2F(h-1)$ for all $h \geq 1$. 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 4.9: 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 $b = -1, 0, 1, 2$. Note the bias towards allowing left subtrees to be relatively taller than right subtrees.
(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 4.10: Allocating one bit per AVL node is sufficient if we exploit the fact that leaf nodes are always balanced allow their bits to be used by the internal nodes. Work out the details for how to do this.
♢

Exercise 4.11: It is even possible to allocate no bits to the nodes of a binary search tree. The idea is to exploit the fact that in implementations of AVL trees, the space allocated to each node is constant. In particular, the leaves have two null pointers which are basically unused space. We can use this space to store balance information for the internal nodes. Figure out an AVL-like balance scheme that uses no extra storage bits. ♢

Exercise 4.12: Work out the details of the one-pass algorithms for AVL insertion and AVL deletion that are outlined in the text. ♢

End Exercises

§5. (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,

$$2 \leq a \leq b. \tag{3}$$

An (a, b)**-tree** is a tree in which

- DEPTH BOUND: All leaves are at the same depth.
- BRANCHING FACTOR BOUND: Each internal node has between a and b children. The exception is the root which has between 2 and b children.

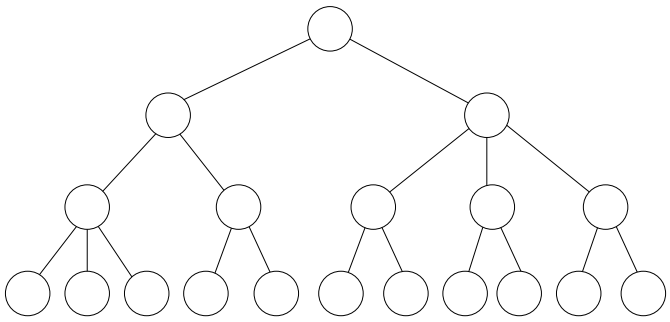

Figure 8: A $(2, 3)$-tree.

The structure of a $(2, 3)$-tree is illustrated in figure 8. The above definition of an (a, b)-tree gives purely structural requirements. We next give additional requirements needed when using (a, b)-trees for searching:

- Each leave stores one key.
- Each internal node stores an ordered list of keys: if node u has m children, then it has $m - 1$ keys. Let the keys be $k_1 \leq k_2 \leq \cdots \leq k_{m-1}$. For $i = 1, \ldots, m$, every key in the i-th subtree of u is at most k_i and at least k_{i-1} (assume that all actual keys are finite, but $k_0 = -\infty$ and $k_m = +\infty$). ith-subtree

An (a,b)-tree that stores keys in this manner will be called an (a,b)**-search tree**. In the special case where $b = 2a-1$, these are called **B-trees**. B-Trees were first introduced by McCreight and Bayer (REF). When $a = 2$, B-tree are called a 2-3 trees and these are first introduced by Hopcroft. 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". Thus, the concept of (a,b)-trees serve to unify a variety of search trees that have been studied. But the freedom to choose a and b more freely also leads to other benefits [**?**].

Let us discuss the organization of keys in nodes of an (a,b)-Search Tree. Normally, we assume that the keys and the pointers to children in a node u are stored in an array in the manner

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

where p_i is the pointer to the i-th child. We can then perform binary search in this array in the natural manner, taking time $O(\log m)$. However, inserting and deleting from this array would take time $O(m) = O(b)$. We shall see that this $O(m)$ complexity is not the critical one in applications. But if this complexity is important, we can use any efficient data structure for an ordered dictionary (§2) to store the keys and pointers in the node.

The Split and Merge Inequalities for (a,b)-tree parameters. The parameters a, b are usually required to satisfy an additional inequality in addition to (3). 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. During insertion, we may make a node that previously have b children to acquire a new child. The resulting node with $b+1$ children must be **split**. The obvious splitting creates 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

$$a \leq \left\lfloor \frac{b+1}{2} \right\rfloor. \tag{4}$$

During deletion, we may remove a child from a node that has a children. The resulting node with $a-1$ children may borrow a child from one of its **adjacent 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 $2a-1 \leq b$, *i.e.*,

$$a \leq \frac{b+1}{2}. \tag{5}$$

Clearly (4) implies (5). However, since a and b are integers, the reverse implication also holds! Thus we normally demand (4) or (5). The smallest choices of these parameters under the inequalities and also (3) is $a = 2, b = 3$, which mentioned above. The case of equality in these inequalities gives rise to $b = 2a-1$, which are precisely the B-trees.

The standard textbooks in the algorithms literature impose the inequality (5) on (a,b)-trees. Let us see show how to allow a/b to increase to any fraction less than 1. There are good reasons for desiring this. A node with m nodes is said to be (m/b)**-full**, and it is **full** when $m = b$. For instance, if a can be about $2b/3$ then this ensures that every node is always at least 2/3-full. This improves the space utilization of standard B-trees that may have only 1/2-full nodes. For example, when split 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 $\lceil (2b+1)/3 \rceil$ nodes. As a result, we require

$$a \leq \left\lfloor \frac{2b+1}{3} \right\rfloor. \tag{6}$$

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 that $a \geq 3$, 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

$$\left\lceil \frac{3a-1}{2} \right\rceil \leq b \tag{7}$$

Because of integrality constraints, the floor and ceiling symbols could be removed in both (6) and (7), without changing the relationship. And thus both inequality are seen to be equivalent to

$$a \leq \frac{2b+1}{3} \tag{8}$$

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

More generally, let $m \geq 2$ be any integer. We require the inequalities

$$m+1 \leq a \leq \frac{mb+1}{m+1}. \tag{9}$$

The lower bound on a ensures that when we merge or split a node, we can be assured of at least $a-1 \geq m$ 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 m siblings has a keys each. We can combine all these $a(m+1)-1$ keys and split them into m new nodes. This merging is valid because of the upper bound (9) on a. In case of splitting, the current node has $b+1$ keys. You may assume that $m-1$ siblings with b keys each. Then we combine all these $mb+1$ keys, and split them into $m+1$ new nodes. Again, the upper bound on a (9) is needed.

Space Utilization. The preceding discussion introduces a new parameter that is important to look at, namely a/b which we call the **space utilization ratio** of the family of (a,b)-search trees. It should be clarified that the model of space utilization assumes that we allocate the same amount of space to every node of a search tree. We emphasize that the ratio a/b is only approximate because of several factors: the space needed 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 show that in a random insertion model, the utilization is about $\lg 2 \sim 0.69\%$. (see [**?**]). This result was the beginning of a technique called "fringe analysis" which Yao [**?**] introduced in 1974. The survey of [**?**] notes that Nakamura and Mizoguchi [**?**] independently discovered the analysis, and Knuth had used the same ideas back in 1973. Note that our bounds (8) implies a space utilization that is close to $\lg 2$, but we guarantee this utilization in the worst case.

Exercises

Exercise 5.1: What is the the best ratio achievable under (5)? Under (8)? ♢

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

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

assuming the inequality (5). ♢

Exercise 5.3: Generalize our above family of (a, b)-trees which allows a utilization ratio of $2/3$ to any constant ratio $r < 1$. ♢

Exercise 5.4: We want to explore the size-balanced version of (a, b)-trees.
(a) Define such trees. Bound the heights of your size-balanced $(a, b$-trees.
(b) Describe an insertion algorithm for your definition.
(c) Describe a deletion algorithm. ♢

End Exercises