Binary Trees

From HPCM Wiki

Contents

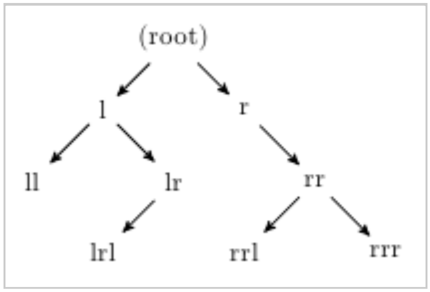
What is a Binary Tree?

A binary tree is a tree in which each node has at most two children. The children of a node are referred to as the left and right child of the node, and may be missing.

In other words, it is a set of nodes with two pointers to children stored in each node, where the points may be NULL to indicate a child is missing, and where all nodes can be reached from a single **root node** by a **unique** path that follows pointers from node to node. It is this last property, the uniqueness of the path from the root to each node, that makes the set of nodes into a **tree**.

Of course, given a binary tree, some terminology becomes obvious. If node N is the child of node M, then M is the **parent** of N. If node M is on the path from the root to node N, then M is an **ancestor** of N and N is a **descendant** of M.

We can give a node in a binary tree a **path label** that identifies the path from the root to the node as a sequence of `l's and `r's, an `l' meaning `go to the left child' and an `r' meaning `go to the right child'. The nodes in the Example Binary Tree have been given path labels in this manner. Thus the node labelled `ll' is the left child of the left child of the root, and the node labelled `rrl' is the left child of the right child of the right child of the root. In this scheme the root should be labelled with the empty sequence of characters, `', but in the example we have chickened out and labelled it `(root)' instead.

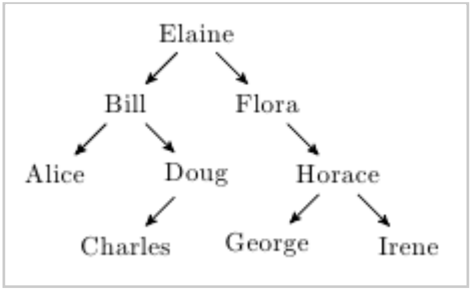


Example Binary Tree

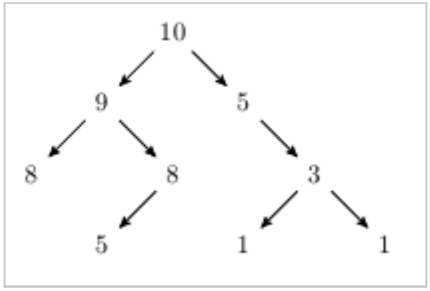
Notice that the nodes with path labels lrr and rl are missing in the example. Of course the nodes labeled lll and llr are also missing, as are many other nodes. A node with no children is called a **leaf**, so in our example the nodes labelled ll, lrl, rrl, and rrr are **leaves**.

Binary trees are commonly used to store information that is attached to the nodes of the tree. In our examples we will attach just one piece of information, which we will call the **node label** to distinguish it from the path label. Our node labels will be numbers or character strings. The set of all possible node labels will be **totally ordered**, which means that given any two unequal node labels one is smaller than the other. Character strings will be taken in lexical (dictionary) order.

Below are two examples of binary trees whose nodes have such labels. In one example, the tree is `sorted', and in the other, the tree is a `heap'. Our next task is to define these terms.



Sorted Binary Tree



Binary Tree Heap

Sorted Binary Trees

A sorted binary tree is a binary tree with labels on the nodes such that for each node N, if L is the label of that node, all the nodes in the subtree rooted at the left child of N have labels less than or equal to L, and all the nodes in the subtree rooted at the right child of N have labels greater than or equal to L.

But wait. What is a subtree?

A **subtree** of a containing tree is any node N in the containing tree, and all the descendents of N in the containing tree. N is the root of the subtree, and we say that the subtree is **rooted** at N.

Thus the subtree rooted at the node r in our Example Binary tree consists of the set {r, rr, rrl, rrr}, the subtree rooted at lr is {lr, lrl}, and the subtree rooted at lrl is just {lrl}. Since a **leaf** is just a node with no children, the subtree rooted at a leaf is just the leaf itself and nothing else.

Now the definition of sorted binary tree should make sense. In the Sorted Binary Tree example, the root is labelled Elaine, the subtree rooted at its left child is {Alice, Bill, Doug, Charles} all of whose labels are before Elaine in the dictionary, and the subtree rooted at the root's right child is {Flora, Horace, George, Irene}, all of whose labels are after Elaine in the dictionary. The rule of sorting also applies to nodes other than the root node. Thus the node labelled Bill has the subtree {Alice} rooted at its left child, and Alice is before Bill in the dictionary, whereas the subtree {Doug, Charles} is rooted at Bill's right child, and Doug and Charles are after Bill in the dictionary.

In fact, every subtree of a sorted binary tree is itself a sorted binary tree.

Now the value of a sorted binary tree is that one can quickly find a node with a given label in it. Thus to find Doug, we start at root Elaine. Since Doug comes before Elaine in the dictionary, we move to Elaine's left child Bill. Since Doug comes after Bill in the dictionary, we move to Bill's right child Doug. And we are done. The number of compares is 1 more than the number of ancestors of Doug in the tree, or in this case, 3 (we include the compare at the node Doug itself which is needed to conclude the search).

This leads us to additional definitions. The **depth** of a node N in a tree is the number of its ancestors. The **height** of a tree is the maximum depth of any node in the tree. This is of course the same as the maximum depth of any leaf in the tree.

Now the central idea of sorted binary trees is that if the height of the tree is H, the number of nodes in the tree can be as much as 2^{H+1}-1. Thus the time to find a node in the tree, which is proportional to H, can be much less than the number of nodes in the tree. For example, if H is 9, one can find any node with 10 compares, but there can be as many as 1023 nodes in the tree. Below we will show how we can build a sorted binary tree that has the nice property that the number of nodes in the tree is roughly 2^H, and even maintain this nice property while both adding data to the tree and removing data from the tree. It is this last fact, that we can maintain the nice property while both adding and removing data, that makes sorted binary trees valuable in the real world.

Finding Neighbors in a Sorted Binary Tree

If we want to find the node in a sorted binary tree with the smallest label, we need merely start at the root and follow left children till we arrive at a node with no left child. Similarly for the node with the largest label we follow right children till we arrive at a node with no right child.

Based on these ideas we can find the node M just before a node N in the label-sorted order of a sorted binary tree by using the following algorithm:

```
// N is the input, M the output
if N has a left child:
    M = left child of N
    // Find largest labelled node of subtree rooted at M:
       while M has a right child: M = right child of M
else:
    M = N
    while M is the left child of its parent: M = parent M
    // N is now the smallest labelled node in the subtree rooted at
        if M is the root:
            error: N is the smallest labelled node in the entire tr
        else:
            M = parent of M
            // N is the smallest labelled node in the right child c
```

The node M just after N can be found by the same algorithm but with `left' and `right' switched, and also with `largest' and `smallest' switched in the comments.

These algorithms are built into the C++ and JAVA implementations of binary sorted trees, but not the C implementation, which also has no way of finding children of nodes. But with a bit of trickery, the C implementation can find the parent of a node when it is inserted into the tree. So for this implementation one can maintain a doubly threaded list of nodes in label sorted order as the nodes are inserted into or removed from the tree, using the fact that the distance between the parent of a node and that node in the sorted list is the size of the subtree rooted at the left child of the node, if the node label is greater than its parent's label, or the subtree rooted at the right child of the node, if the node label is less than its parent's label. For a balanced tree the size of this subtree is 0 for 1/2 the nodes, 1 for 1/4 the nodes, 3 for 1/8 the nodes, 7 for 1/16 of the nodes, etc, so the expected size is a bit less than 1/2 the height of the tree, and the cost of finding a place in the doubly linked list to insert the node is on average proportional to the height of the tree.

Binary Tree Heaps

A heap is a tree such that the label of each node is greater than or equal to the labels of its children (or you can invert the sense of order and substitute `less than' for `greater than', but we will not do so in this discussion).

As a consequence, the root label is the greatest label in the tree. This makes a heap a natural structure for storing a set whose greatest element must be found quickly.

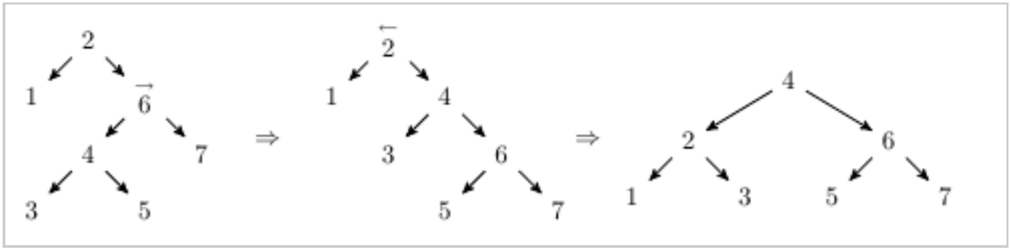
In a manner similar to sorted binary trees, the central idea of a binary heap is that if the height of the tree is H, the number of nodes in the heap can be as much as 2^{H+1}-1. Finding the node with the largest label always takes a small fixed amount of time, but inserting a new node can take time proportional to H, and similarly removing an arbitrary node can take time proportional to H. It is also easy to make heaps with the nice property that the number of nodes in the heap is roughly 2^H, so H is much much smaller than the number of nodes in the heap, and this fact makes heaps valuable in the real world.

Self-Balancing Sorted Binary Trees and Heaps

A **balanced** binary tree is one in which for every node the heights of the subtrees rooted at the children of the node do not differ by more than 1. It can be easily proved by induction on the height of the binary tree, that a balanced binary tree of depth H has between α^H and 2^{H+1}-1 nodes, for α=(sqrt(5)+1)/2 > 1.618 > 1. (The proof is by mathematical induction. The induction case is $\alpha^H \le 1+\alpha^{H-2}+\alpha^{H-1}$ and the base cases H=0 and H=1 must be checked separately. Taking α to be the positive solution of $\alpha^2=1+\alpha$ suffices.)

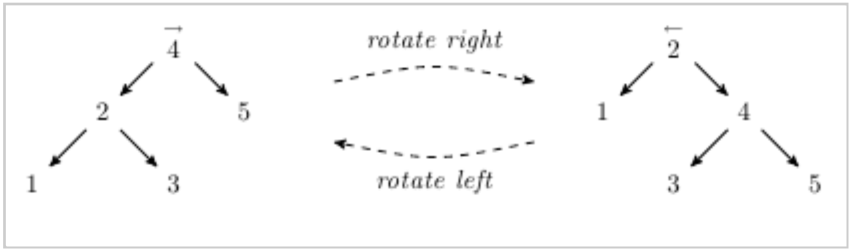
Thus operations on a binary tree that take time proportional to the height H of the tree will be very fast, relative to the number of nodes in the tree, if the tree is balanced. This fact makes it worthwhile to put reasonable effort into building balanced sorted binary trees and balanced binary tree heaps.

Given an unbalanced sorted binary tree, it is possible to balance it by rotation operations, as suggested by the example below.



Balancing a Binary Tree

Binary Tree Rotations



Binary Tree Rotations

Algorithms that balance binary trees use a basic operation called a **rotation**, as pictured in the Binary Tree Rotations figure. A rotation converts one subtree of a binary tree into another subtree by, speaking very roughly, exchanging two nodes. In the figure the nodes exchanged are labeled 2 and 4.

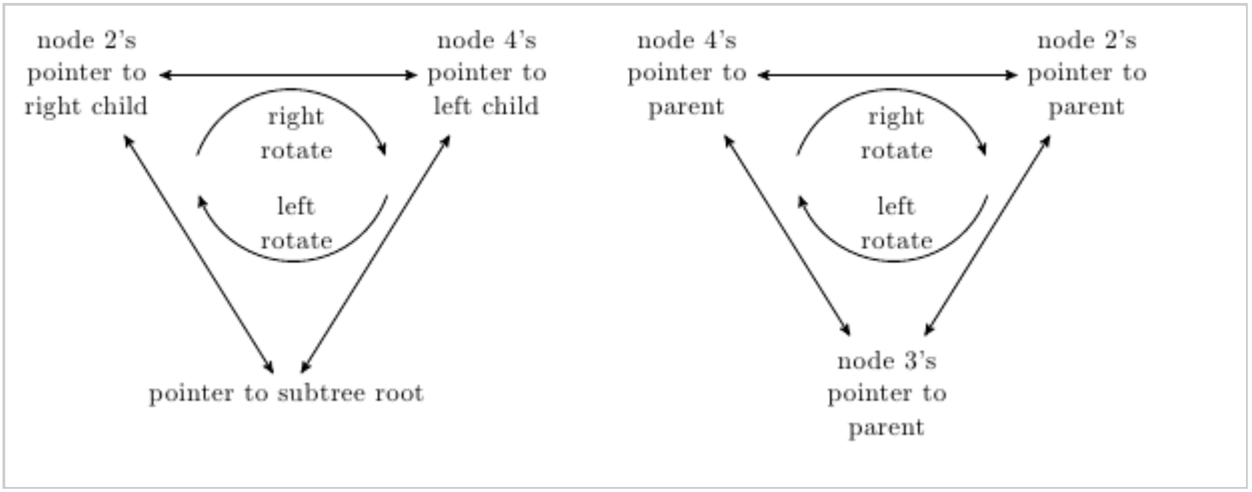
A **right rotation** makes the left child of the subtree root into the new root, and a **left rotation** makes the right child of a subtree root into the new root.

If we intend to perform a right rotation of the subtree rooted at a node N in a tree, we put a right arrow over the node, as in the figure. Similarly if we intend to perform a left rotation, we put a left arrow over the node.

Looking at the figure we see that:

- The pointers affected are node 2's pointer to right child, node 4's pointer to left child, and the pointer to the root of the subtree. In fact, these 3 pointers are `rotated' as indicated in the Binary Tree Pointer Rotation diagram.
- If in addition nodes contain pointers to their parents, the parent pointers in nodes 2, 3, and 4 are affected. In fact, these 3 pointers are also `rotated' as indicated in the Binary Tree Pointer Rotation diagram.
- Nodes 1, 3, and 5 need not be leaves; they can be entire subtrees.
- If the original subtree is sorted, so is the result of applying a rotation to the subtree.

  Here node 1 stands for a sorted subtree of nodes with labels less than 2, node 3 stands for a sorted subtree of nodes with labels between 2 and 4, and node 5 stands for a sorted subtree of nodes with labels above 4.


node 2's pointer to right child
node 4's pointer to left child
right rotate
left rotate
pointer to subtree root
node 4's pointer to parent
node 2's pointer to parent
right rotate
left rotate
node 3's pointer to parent

Binary Tree Pointer Rotation

It can be proved that any sorted tree can be converted by rotations to a balanced sorted tree. More importantly, if one is adding one node to an already balanced sorted tree, or removing one node, one can show that the number of rotations required to produce a balanced sorted tree is proportional to the height of the original tree. Similar results can be proved for heaps.

Practical Self-Balancing Binary Trees

Self-balancing sorted and heap binary trees are implemented by the standard libraries of some of the common languages. In this section we will briefly describe the interfaces of some of these implementations. Later sections will describe easy methods for building your own functions to maintain self-balancing binary trees.

In the following nodes contain both node labels, which are used for sorting or heapifying, and node values, which are what the user really wants when she uses a node label to find a node in a binary tree. For example, a symbol table may have name strings as node labels and, say, numbers as node values.

C Binary Trees

POSIX libc defines functions that implement binary trees. It is unclear whether these POSIX functions must be self-balancing, but the GNU implementation is self-balancing with the property that the maximum time to insert or remove an element is proportional to the height of a tree, or equivalently to the logarithm of the number of elements in the tree.

The documentation is a bit confusing. There are two kinds of structures, items allocated and freed by the library user, and tree nodes allocated and freed by the library. An item contains a key which labels a tree node; a tree node begins with a pointer to an item, which is the item of the tree node. Thus items may be compared by the `compare' function, and a pointer to a node may be cast to a pointer to a pointer to an item. Items may hold additional values besides a key.

The `const void * key' arguments, and the arguments to `compare', are pointers to items.

The `void *' return values are pointers to tree nodes (not items, and they should be declared const), as is the `const void * nodep' argument to `action'.

The root of the tree is a variable that holds a pointer to a tree node. The `void ** rootp' arguments are pointers to this root variable, while the `const void * root' argument to twalk is the value of the root variable.

An example implementing a simple symbol table that maps strings to double precision floating point numbers is:

```
#include <search.h>
#include <string.h>
/*
 * Define item data type.
 */
struct item {
    const char * name;
    double value;
};
/*
 * Define an empty table.
 */
void * table = NULL;
/*
 * Define a function to compare elements.
 */
int compare ( const void * xp, const void * yp )
{
    const struct item * item_xp = (const struct item *) x;
    const struct item * item_yp = (const struct item *) y;
    return strcmp ( item_xp->name, item_yp->name );
}
/*
 * Define a twalk function to compute the height of a tree.
 */
int height;  /* Initialize to 0. */
void action ( const void * nodep, VISIT which, int depth )
{
    if ( depth + 1 > height ) height = depth + 1;
}
/*
 * Define a twalk function to print the names and values of
 * each tree node item, in sorted order of item names,
 * with each indented by 2 * depth of node.
 */
void print ( const void * nodep, VISIT which, int depth )
{
    const struct item * itemp = * (const struct item **) nodep;
    if ( which != leaf && which != postorder ) return;
    for ( int i = 0; i < depth; ++ i ) printf ( "  " );
    printf ( "%s = %g\n", itemp->name, itemp->value );
}
. . . . .
/* Add an element with name "X" and value 5.5.  Replace any
 * previous element named "X".  Assume we do not run out of
 * memory (else itemp or nodep will be NULL).
 */
struct item * itemp = malloc ( sizeof ( struct item ) );
itemp->name = "X";
struct item ** nodep =
    (struct item **) tsearch ( itemp, & table, compare );
if ( * nodep != itemp ) free ( itemp );
(*nodep)->value = 5.5;
. . . . .
/* Return a pointer to a node pointing to an item named "Y"
 * if there is one, or NULL otherwise.
 */
struct item key;
key.name = "Y";
struct item ** nodep = (struct item **)
                       tfind ( & key, & table, compare );
/*
 * Remove this element if it exists.
 */
if ( nodep != NULL )
{
    struct item * itemp = * nodep;
    tdelete ( itemp, &table, compare );
    free ( itemp );
}
. . . . .
/* Print height of tree.
 */
height = 0;
twalk ( table, action );
```

```
  printf ( "height = %d\n", height );
  . . . . .
  /* Print the symbol names and values sorted by name with
   * each indented by the depth of the item in the tree.
   */
  twalk ( table, print );
```

If you need to find the node just before or just after a given node in the label-sorted sequence of a sorted binary tree, you can use the method at the end of Finding Neighbors in a Sorted Binary Tree. The following code can be used to find the parent of a node in the tree:

```
  /* Given a pointer to an item, itemp, that is known to be an
   * item of a node N in the tree, find a pointer parentp to the
   * item of the parent of N, if it exists, or set parentp to
   * NULL if N is the root node.
   */
  struct item * parentp = NULL;
  if ( itemp != * (struct item ** ) table )
  {
      parentp = * (struct item **) tdelete ( itemp, &table, compare
      tsearch ( itemp, &table, compare );
  }
```

C++ Binary Trees

The C++ `map` type implements a self-balancing sorted binary tree with the property that the maximum time to insert or remove an element is proportional to the height of the tree, or equivalently to the logarithm of the number of elements in the tree.

An example using `map` to implement a simple symbol table that maps strings to double precision floating point numbers is:

```
  #include <map>
  #include <string>
  using namespace std;
  map<string,double> table;
  . . . . .
  // Make an existing table empty.
  table.clear();
  // Add one element, or change an existing element.
```

```
table["X"] = 5.5;
// If an element exists, return it, otherwise return 0.
double v = table.count("Y") ? table["Y"] : 0;
// Remove an element if it exists
table.erase(table.find("Z"));
// Return the number of nodes in the map.
int count = table.size();
// Return the first (least) key in a non-empty map.
string first = table.begin()->first;
// Return the last (greatest) key in a non-empty map.
string last = (--table.end())->first;
. . . . .
typedef map<string,double>::iterator table_p;
table_p p = table.find ( "X" );
table_p q = p;
if ( p == table::end() )
    cout << "X not in table" << endl;
else
{
    if ( p == table::begin() )
        cout << "X at beginning of table" << endl;
    else
        cout << "X after " << (--q)->first << " in table" << endl;
    if ( ++ p == table::end() )
        cout << "X at end of table" << endl;
    else
        cout << "X before " << p->first << in table" << endl;
}
```

The type `map<S,T>` is a map in which the key values, the labels on the nodes which are used to sort the nodes, are of type S, and the other values attached to the nodes are of type T. The type S must have the < less than operator defined.

The type `map<S,T>::iterator` is a pointer to an element in the map. If `p` is such a pointer returned by the `find` function, `*p`, the value pointed at by `p`, has type `pair<S,T>`, with components `(*p)->first` of type S and `(*p)->second` of type T. The `begin()` member function returns a pointer to the first element of a table, and the `end()` member returns a pointer to just beyond the end of the table, with this being also returned by `find` if the element it is asked to find does not exist. Pointers may be incremented by `++` and decremented by `--` to move to the next or previous element in the table, which is sorted by element keys.

JAVA Binary Trees

The JAVA `TreeMap` type implements a self-balancing sorted binary tree with the property that the maximum time to insert or remove an element is proportional to the height of the tree, or equivalently to the logarithm of the number of elements in the tree.

An example using `TreeMap` to implement a simple symbol table that maps strings to double precision floating point numbers is:

```
import java.util.TreeMap;
import java.util.NoSuchElementException;
TreeMap<String,Double> table = new TreeMap<String,Double>();
. . . . .
// Make an existing table empty.
table.clear();
// Add one element, or change an existing element.
table.put("X", Double (5.5));
// If an element exists, return it, otherwise return null.
Double v = table.get("Y");
// Remove an element if it exists
table.remove("Z");
// Return the first (least) key in the map,
// or throw NoSuchElementException if there is none.
String first = table.firstKey();
// Return the last (greatest) key in the map,
// or throw NoSuchElementException if there is none.
String last = table.lastKey();
// Return the first (least) key higher than "Z" in the map,
// or null if there is none.
String next = table.higherKey ( "Z" );
// Return the last (greatest) key lower than "Z" in the map,
// or null if there is none.
String previous = table.lowerKey ( "Z" );
```

The type `TreeMap<S,T>` is a map in which the key values, the node labels used to sort the nodes, are of type S, and the other values attached to the nodes are of type T. The type S must implement the `Comparable<S>` interface. The type T must be a subtype of `Object`, so we need to use the type `Double` and not `double`.

COMMONLISP Binary Trees

We have not been able to find a standard implementation of sorted binary trees in COMMONLISP.

Building Your Own Sorted Binary Tree

Adding a node to a sorted binary tree is easy: you just go move down the tree going left if the current node label is greater than the label you are adding and right if it is less, until this tells you to move to an empty child, and then you insert the node at that child. But if the original tree was balanced, the result may not be balanced.

It turns out that if you build a sorted binary tree by randomly choosing the next node to add, then the result is effectively balanced: that is, the height is proportional to the logarithm of the total number of nodes with a probability so close to 1 that you will never see an exception.

An alternative to this random addition idea is the idea of a **treap**, which is both a tree sorted on node labels, and a heap on randomly chosen node weights. The height of a treap is proportional to the logarithm of the number of nodes in the treap, with a probability so close to 1 that you will never see an exception.

So each node contains a node label, a node weight, and node values. We will assume that the node weight is a randomly chosen integer and no two node's have the same weight. This could be done by using a pseudo-random number generator with a cycle length of close to 2^{64} to generate a 64-bit pseudo-random integer for each node when the node is created.

By definition, a treap is sorted on node labels, and a heap on node weights. It turns out that given a set of nodes with labels and weights, where no two nodes have the same label, and no two the same weight, then the treap containing these nodes is unique. This is easy to see: the node with the largest weight must be the root, nodes with smaller labels than the root must be in the root's left subtree, nodes with larger labels must be in the node's right subtree, and so mathematical induction on the number of nodes can be used to prove the treap is unique.

Inserting an Element

Create the new node, and insert it by moving down the tree and installing it when you get to an empty child. Now the tree is sorted but it will not generally be a heap. The new node will be a heap-defect in that its weight will be larger than its parent's weight (and possibly also the weights of some of its other ancestors).

So rotate at the parent away from the defective child so the defective child becomes the new root of the subtree the parent was the root of. Now this subtree is a valid treap, but its root may be a heap-defect in the containing treap. If it is, recursively repeat this process until the subtree root is not a defect, which must happen if the subtree is the entire tree.

Removing an Element

Find the element to remove. Rotate it away from which ever one of its children has the larger weight (an empty child is always considered to have the smaller weight) so that larger weight child becomes the new root of the subtree and the node to be removed becomes one of its children. Continue to remove the node which is now the root of a smaller subtree. Eventually this subtree will have no children and you can just remove the node.

Building Your Own Heap

It is not hard to build a heap that is *approximately* self-balancing. More specifically, the heap will be self-balancing if elements are only inserted into the heap. When elements are removed, the heap will get out of balance, but further insertions will act to re-balance the heap. The height of the heap will be proportional to the logarithm of the *maximum* number of elements that are ever in the heap.

Each node of the heap records its label and also the height of the subtree rooted at that node. Each node also contains a pointer to its parent.

Inserting an Element

If the element label is larger than the label at the root of the tree, exchange these labels. Then pick the subtree with smaller height, or pick arbitrarily if both have the same height, and recursively insert the element in the picked subtree.

When a leaf is inserted, the height of its parent must be recalculated. When the height of any node changes, the height of that node's parent must be recalculated.

Removing an Element

First you must find the element in the heap. If the element is the one with maximum label it will be the root. Otherwise you must have some other data structure for finding the element. For example, you may have a hash table that maps node labels to pointers to heap elements.

The element is actually removed only if it is a leaf. Otherwise the larger child label is moved to the element and that child is removed recursively.

When a leaf is removed, the height of its parent must be recalculated. When the height of any node changes, the height of that node's parent must be recalculated.