Heap Data Structures, Reading

It's not like we hit upon some new and clever way to use an array to represent hierarchical structures. This data structure is actually called a "heap".

A heap is a hierarchical structure, where (1) there is one value at the top, (2) each value has at most two children and is greater-than or equal-to those children, and (3) there are no holes in the array -- all values in the index range 0 to siz-1 are in use, all values at index siz and above are not in use.

Heap Push

Here's the process for adding a new value:

1. if the array is full, double its capacity
2. copy the new value to **V* values;** array at index siz
3. create an **int index** and initialize it to siz -- *for traversing up the hierarchy*
4. start loop -- *to promotion the newly added value to it's right place in the hierarchy*
    - if index == zero, it's reached the top -- exit the loop -- *newly added value got promoted to the top*
    - compute the parent index of index
    - if the *value* at index **<** the *value* at parent index, exit the loop *(no more promotions)*
    - swap values at the parent index and index -- *promote and demote*
    - set index equal to the parent index -- *traversing up the hierarchy*
5. increment siz

This organizes the hierarchy so that the largest value is at the top. That's a "max-heap", and is the default for heaps. To make a "min-heap" with the smallest value on top (like the time for the next occurrence in an event queue), reverse any value comparisons -- but do *not* reverse the less-than to be a greater than, because our convention for requirements on **typename V** will always be equals-equals and less-than for compares, and never any of the other compares (like greater-than). That's to be consistent with the C++ STL, and to make things easier for programmers using our templates.

Heap Pop v.1: Filling The Hole

Removing seems like it would be very straightforward -- just remove the value at index zero, and promote children in a loop until the bottom of the hierarchy is reached. But if we promote from the bottom level and **leave a hole in the array** as a result, we no longer have a "heap".

There are two common ways to deal with this problem: (1) fill the hole and (2) avoid the hole.

Filling the hole is perhaps the more intuitive solution. We simply copy the value at the values array index siz (*after* decrementing it) into the hole, thereby filling it. It's old value does not need to be "zeroed out" in any way, because by decrementing siz, we're ignoring that place in the array. But having inserted the copied value, we have to check to see that it is less than its parent, because if not, we need to start a traversal up the hierarchy until the parent-child relationships are resolved.

So we end up with *two* loops, one working from the top down, followed by one working from the bottom up.

1. create an **int index**, initialized to zero -- *for traversing through the hierarchy*
2. start loop -- *to refill the position at the top of the hierarchy -- the one that's getting popped*
    - compute the left and right child indexes of index
    - if there is no left child, exit the loop -- *we reached the bottom of the heap*
    - if there's no right child *OR* the *value* at the right child index < the value at the left child index, copy the value at the left child index to the value at index and set index to the left child index
    - otherwise copy the value at the *right child* index to the value at index and set index to the *right child* index
3. decrement siz -- *now if index just happens to equal siz, we're done, but no harm in continuing anyway*
4. copy the value at siz to the value at index -- *it's the right-most value in the bottom-most row*
5. start loop -- *to promote the newly-repositioned value at index*
    - if index == 0, exit the loop -- *the value got promoted all the way to the top*
    - compute the parent index of index
    - if the value at index < the value at the parent index, exit the loop -- *no more promotions*
    - swap values at the parent index and index -- *promote and demote*
    - set index equal to the parent index -- *traversing up the hierarchy*

The reason that step 3 "continues anyway" if index==siz is that the next loop exits during its first cycle, doing nothing -- it avoids having an extra if-

The reason that step 3 continues anyway if index == siz is that the next loop exits during its first cycle, doing nothing -- it avoids having an extra if statement to skip over steps 4 and 5.

Heap Pop v.2: Avoiding The Hole

Or we can avoid the hole altogether, but the solution is not intuitive. It involves 3-way "competitions" for promotion opportunities, and only one loop, working from the top down. One loop may seem better than two, but those two involved 2-way competitions for promotion, so "six of one, half a dozen of the other"...

The third competitor comes from *removing the value at siz* (after decrementing it). It competes with the top value's children for the top. What's not intuitive is why we would ever take a value from the very bottom level and have that value compete with values at higher levels. In any case, it works like this:

1. decrement siz -- *so that the value at index siz can participate in promotion "competitions"*
2. create an **int index**, initialized to zero -- *for traversing through the hierarchy*
3. start the loop -- *to promote among three competitors: the left and right children, and the value at siz*
    - compute the left and right child indexes of index -- *find siz's competitors*
    - if there's no left child, exit the loop -- *the value at siz wins by default*
    - if there's no right child -- *it's a 2-way competition between left child and siz*
        1. if the value at the left child index < value at siz, exit the loop -- *the value at siz wins*
        2. otherwise copy the left child's value to the value at index, and set index to the left child's index -- *the left child wins*
    - otherwise if the value at the left child index < value at siz *and* the value at the right child index < value at siz, exit the loop -- *the value at siz wins*
    - otherwise if the the value at the left child index < value at the right child index, copy the right child's value to the value at index, and set index to the right child's index -- *the right child wins*
    - otherwise copy the left child's value to the value at index, and set index to the left child's index -- *the left child wins*
4. copy the value at siz to the value at index

Checking For Children

In each of these algorithms there are references to children existing or no. To test whether or not there is a left or right child, compare its index to **siz**. If an index is greater than or equal to **siz**, then it's not part of the heap, and there is no value there.