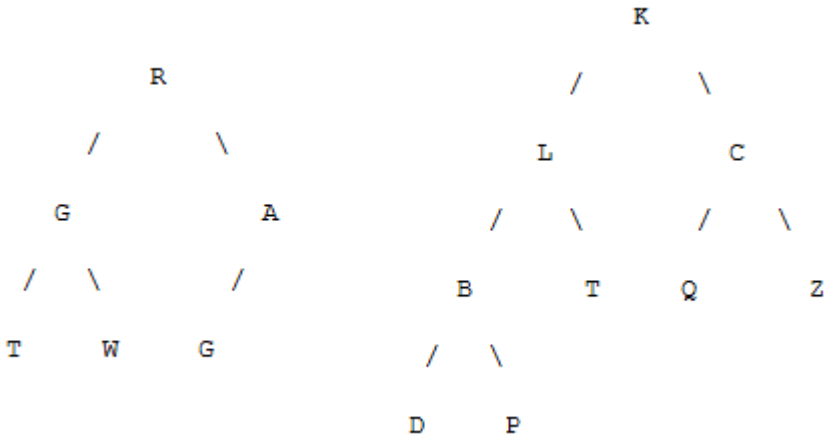
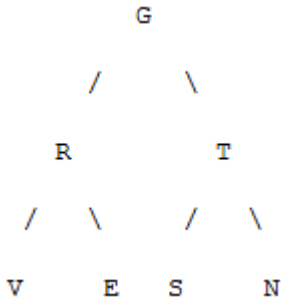
Heaps and Heapsort - Lab Handouts

An **almost complete** binary tree is a binary tree:

- all the leaves are at the bottom level or the bottom 2 levels
- all the leaves are in the leftmost possible positions
- all levels are completely filled with nodes (except for the bottom level)



A **complete binary tree:**

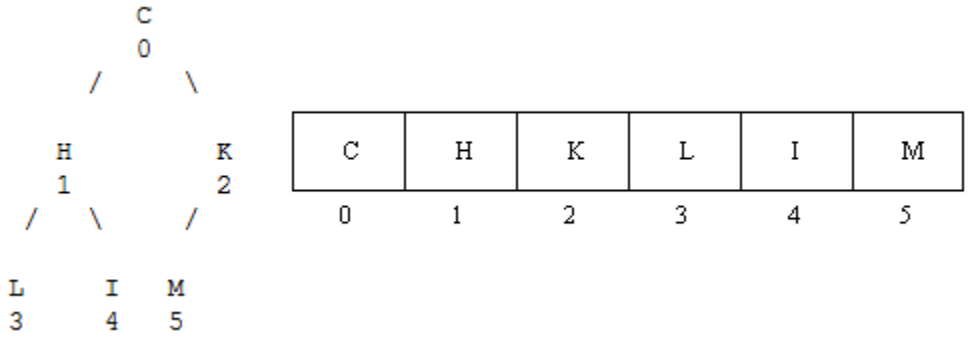


A **minimal heap** (descending heap) is an almost complete binary tree in which the value at each parent node is less than or equal to the values in its child nodes.

The minimum value is in the root node. Any path from a leaf to the root passes through the data in descending order.

Storage of Heap Data

Begin by numbering the nodes level by level from the top down, left to right. Then store the data in an array.



The advantage of this method over using the usual pointers and nodes is that there is no wasting of space due to storing two pointer fields in each node.

Considering CI to be the current index, we can easily calculate the index of its parent and its child nodes:

Parent(CI) = (CI - 1) / 2

RightChild(CI) = 2 * (CI + 1)

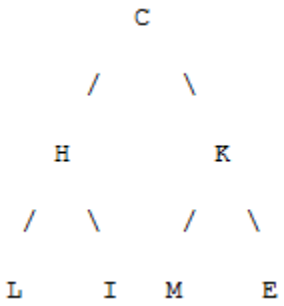
LeftChild(CI) = 2 * CI + 1

Inserting into a Heap

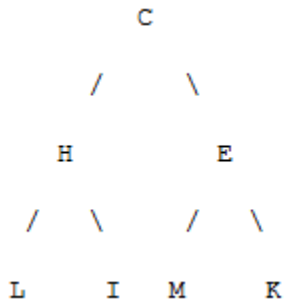
This is done by temporarily placing the new item at the end of the heap (array) and then calling a FilterUp routine to make any needed adjustments on the path from this leaf to the root.

Insert E in the above heap:

(1) Place E in the next available position:



(2) The FilterUp routine now checks the parent, K, and sees that things would be out of order as they are. So K is moved down to where E was. Then the parent above that, C, is checked. It is in order relative to the target item E, so the C is not moved down. The hole left behind is filled with E, then, as this is the correct position for it.

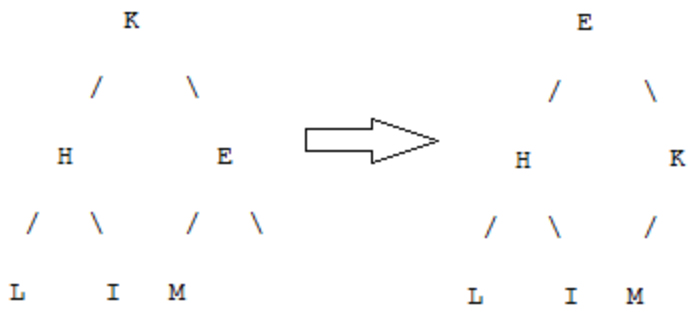


Removing from a Heap

- remove the item from the root
- adjust the binary tree so that we again have a heap (with one less item)

The algorithm works like this: first, remove the root item and replace it temporarily with the item in the last position. Call this replacement the target. A FilterDown routine is then used to check the path from the root to a leaf for the correct position for this target.

Remove C from the above heap:



Heapsort

- convert the array you want to sort into a heap
- remove the root item (the smallest), readjust the remaining items into a heap, and place the removed item at the end of the heap (array)
- remove the new item in the root (the second smallest), readjust the heap, and place the removed item in the next to the last position and so on

Exercises:

1. Insert into an empty min-heap the following numbers:
25, 17, 36, 2, 3, 100, 1, 19, 17
Delete all the elements of your heap.
2. Implement an array-based heap data structure. Implement the functions for returning the parent and the childs of a given node, using the following signatures:

```
template <typename T>
int Heap<T>::parent(int poz)
{
    // TODO
}

template <typename T>
int Heap<T>::leftSubtree(int poz)
{
    // TODO
}

template <typename T>
int Heap<T>::rightSubtree(int poz)
{
    // TODO
}
```

If the parent/childs don't exist, the functions will return -1.

3. Consider the following array:
25, 17, 36, 2, 3, 100, 1, 19, 17
Transform it into a min-heap.
Sort the array into descending order using heap-sort:

- remove the root item (the smallest), readjust the remaining items into a heap, and place the removed item at the end of the heap (array)

- remove the new item in the root (the second smallest), readjust the heap, and place the removed item in the next to the last position and so on

Homework:

Implement the heapsort and test it.

References:

http://cis.stvincent.edu/html/tutorials/swd/heaps/heaps.html

Extra:

Code to implement lots with tables (you must copy the code in a C + + compiler)
Warning: root can have index 1, not 0.
So, in this case:
If CI is the current index:
Parent (CI) = CI / 2
RightChild (CI) = 2 * CI
LeftChild (CI) = 2 * 1 + CI

```
#include <stdio.h>
template<typename T> class Heap {
   public:
      T *H;
      int currentDim, maxDim;
      Heap(int maxDim) {
         this->maxDim = maxDim;
         H = new T[this->maxDim + 1];
         currentDim = 0;
      }
      void insertElement(T x) {
         if (currentDim == maxDim) {
            fprintf(stderr, "Error!\n");
            return;
         }
         currentDim++;
         H[currentDim] = x;
         filterUp(currentDim);
}
T peek() {
         if (currentDim == 0) {
            fprintf(stderr, "Error!\n");
            T x;
            return x;
         }
         return H[1];
      }
      T extractMin() {
         if (currentDim == 0) {
```

```
            fprintf(stderr, "Error!\n");
            T x;
            return x;
        }
        T minValue = H[1];
        H[1] = H[currentDim];
        currentDim--;
                        if (currentDim > 0)
            filterDown();
        return minValue;
    }
    void filterUp(int l) {
        int parent;
        T vaux;
        parent = l / 2;
        while (l > 1 && H[parent] > H[l]) {
            vaux = H[parent];
            H[parent] = H[l];
            H[l] = vaux;
            l = parent;
            parent = l / 2;
        }
    }
    void filterDown() {
        int l = 1;
        T vaux;
        while (1) {
            if (2 * l + 1 > currentDim) {
                if (2 * l > currentDim)
                    break;
                else if (H[2 * l] < H[l]) {
                    vaux = H[2 * l];
                    H[2 * l] = H[l];
                    H[l] = vaux;
                    l = 2 * l;
                }
                else
                    break;
            }
            else {
                if (H[2 * l] <= H[2 * l + 1] && H[2 * l] < H[l]) {
                    vaux = H[2 * l];
                    H[2 * l] = H[l];
H[l] = vaux;
                    l = 2 * l;
                }
```

```
                else
                if (H[2 * l + 1] <= H[2 * l] && H[2 * l + 1] < H[l]) {
                    vaux = H[2 * l + 1];
                    H[2 * l + 1] = H[l];
                    H[l] = vaux;
                    l = 2 * l + 1;
                }
                else
                    break;
            }
        }
    }
};
int main() {
      Heap<int> heap(1000);
   heap.insertElement(10); heap.insertElement(100);
   heap.insertElement(21); heap.insertElement(4);
         heap.insertElement(17); heap.insertElement(6);
   heap.insertElement(9); heap.insertElement(7);
   heap.insertElement(13);
   printf("%d\n", heap.peek());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.peek());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   printf("%d\n", heap.extractMin());
   return 0;
}
```