# CMPSCI 187: Programming With Data Structures

Lecture #30: Defining and Implementing Heaps
28 November 2011

## Defining and Implementing Heaps

- The Heap Conditions on a Binary Tree
- `HeapADT` and its Operations
- The `LinkedHeap` Class
- Adding and Removing in the Linked Heap
- Adding in the Array Heap
- Removing in the Array Heap

## The Heap Conditions on a Binary Tree

- A **heap** is a complete binary tree where each node holds an element from an ordered base type. In a **minheap** (defined by the code in L&C), the smallest element is at the root and each parent is less than or equal to its children. In a **maxheap** (as you will use in Project #7), the greatest element is at the root and each parent is greater than or equal to its children.

- Remember that a complete binary tree has all its leaves at two adjacent levels, and has its leaves at the lower level left-justified. If we place a tree structure on the numbers {0, 1,..., n-1} by making the two children of i the nodes 2i+1 and 2i+2, we get a complete binary tree with n nodes.

- All complete binary trees have the same shape, but there are in general many ways to place a given data set in a heap. As long as the heap condition is met at each node, nodes that are not ancestor and descendant may be in any relative order.

## `HeapADT` and its Operations

- If we add an element to a heap that has n nodes already, the heap has to expand into position n because it must remain a complete binary tree. But the new element may not belong there.

- The basic plan is to put the new element there and **promote** it as necessary until it fits into its new place. Promoting means exchanging with its parent.

- If we remove the minimum element from an n-node heap, it is position n-1 that must eventually be vacated. The basic plan is to take the element from position n, put it into the hole at the root, and let it fall down to a correct position. We move it down by exchanging it with the larger of its children.

```
public interface HeapADT<T> extends BinaryTreeADT<T> {
   public void addElement (T obj);
   public T removeMin( );
   public T findMin( );
```

## The `LinkedHeap` Class

- We make our heap from `HeapNode` objects, which are `BinaryTreeNode` objects augmented with a parent pointer. The `LinkedHeap` is then a `LinkedBinaryTree` with an additional pointer `lastNode` to the last leaf.

- The `addElement` operation uses two helper methods -- one to find the parent of the node to be inserted and one to restore the heap conditions after a new last node has been added.

```
public void addElement (T obj) {
   HeapNode<T> node = new HeapNode<T> (obj);
   if (root == null) root = node;
   else{HeapNode<T> nextParent = getNextParentAdd( );
      if (nextParent.left == null) nextParent.left = node;
      else nextParent.right = node;}
   lastNode = node;
   count++;
   if (count > 1) heapifyAdd( );}
```

## Helper Methods for Adding in a `LinkedHeap`

- Here is the method to locate the parent of the node that will be inserted (the parent of the node after `lastNode` in level order). Note the casts whenever we reference the `left` or `right` field of a `HeapNode` object -- since these are `BinaryTreeNode` fields their type is `BinaryTreeNode`.

```
private HeapNode<T> getNextParentAdd ( ) {
   HeapNode<T> result = lastNode;
   while ((result != root) && (result.parent.left != result))
      result = result.parent;
   if (result != root)
      if (result.parent.right == null) result = result.parent;
      else {result = (HeapNode<T>) result.parent.right;
         while (result.left != null)
            result = (HeapNode<T>) result.left;}
   else while (result.left != null)
      result = (HeapNode<T>) result.left;}
   return result;}
```

## Adding and Removing in the Linked Heap

- The `addElement` operation needs one more helper, below, which moves a too-small value from `lastNode` to its proper position.
- The `removeMin` operation also uses two helper methods. It returns the element at the root, finds the new last node, makes the old last node `null`, puts its value at the root, and reheapifies. The heapifyRemove method fills the hole at the root by successively promoting as many elements as needed. We won't do the code for `removeMin` here.

```
private void heapifyAdd( ) {
   T temp;
   HeapNode<T> next = lastNode;
   temp = next.element;
   while ((next != root) && (((Comparable) temp).compareTo
                             (next.parent.element) < 0)) {
      next.element = next.parent.element;
      next = next.parent;}
   next.element = temp;}
```

## Adding in the Array Heap

- We add the new element in the position after the last, then just heapify by promoting the new value as long as it is too small.  We compute parent nodes arithmetically as usual for this implementation.

```
public void addElement (T obj) {
   if (count == tree.length) expandCapacity( );
   tree [count] = obj;
   count++;
   if (count > 1) heapifyAdd( );}

private void heapifyAdd( ) {
   int next = count - 1;
   T temp = tree[next];
   while ((next != 0) && (((Comparable) temp).compareTo
                           (tree[(next-1)/2]) < 0)) {
      tree[next] = tree[(next-1)/2];
      next = (next - 1)/2;}
   tree[next] = temp;}
```

## Removing in the Array Heap

- Again the main effort is to reheapify once we create a problem by putting the element from the last node in the root position to replace the one we take out.

```
public T removeMin( ) throws EmptyCollectionException {
   if (isEmpty( )) throw new EmptyCollectionException (“Heap”);
   T minElement = tree[0];
   tree[0] = tree [count - 1];
   heapifyRemove( );
   count--; return minElement;}

private void heapifyRemove ( ) {
   T temp; int node = 0, left = 1, right = 2, next;
   if ((tree[left] == null) && tree[right] == null) next = count;
   else if (tree[left] == null) next = right;
   else if (tree[right] == null) next = left;
   else if (((Comparable) tree[left]).compareTo(tree[right]) < 0)
      next = left;
   else next = right;
   temp = tree[node];
   // move “next” down as long as its element is < temp’s
```

## The Rest of the `heapifyRemove` Method

- We fill the hole by promoting the smaller child into it and moving the hole downward. We get larger and larger elements in our hole until we either leave the tree entirely or get an element larger than or equal to “temp”. If we replace the *first* of these by “temp”, we preserve the heap property.

```
while ((next < count) &&
       (((Comparable) tree[next]).compareTo (temp) < 0)) {
   tree[node] = tree[next];
   node = next; left = 2*node + 1; right = 2*(node + 1);
   if ((tree[left] == null) && (tree[right] == null))
      next = count;
   else if (tree[left] == null) next = right;
   else if (tree[right] == null) next = left;
   else if (((Comparable)tree[left]).compareTo(tree[right]) < 0)
      next = left;
   else next = right;}
tree[node] = temp;}
```