Computer Science I — CSCI 141
Heaps — Lecture

1. Problems of Ordering

There are many kinds of ordering problems.

1.1 Traffic Simulation

A traffic simulation system simulates the passage of time and gives users information about congestion, general traffic speeds, etc. One piece of the system may measure vehicles passing through a toll booth. When a vehicle arrives at the toll booth, the system stores a timestamp in a queue of *events*. For each step of the simulation, the system looks in the queue of events for the earliest timestamped event to process.

1.2 Studio 54

Studio 54 was the hottest nightclub in New York City in the late 1970s, and many celebrities went there in addition to more *normal people*. There were always long lines waiting to get in. If you think that Andy Warhol waited in line to get in, think again. http://www.youtube.com/watch?v=_dl726_FKhc For a celebrity like Andy, the gatekeeper let him jump right to the front and go in the club. In fact the gatekeepers used to walk the line and choose the coolest of the normal people to let in.

1.3 Airplane Boarding

When an airline boards people, it usually calls them in order of first class, then the front rows, and then progresses to the end of the plane. The airline does not board them by seat number, but there are a finite number of people to board any given plane.

1.4 Sorting or What?

While we can handle all these problems with sorting, we know that can be expensive. In the cases of the traffic simulation and the Studio 54 waiting line, there is a potentially unbounded number of elements; events and people keep coming and coming, and we would have to continually sort after each arrival.

We would like a better way than having to continually sort changing collections.

2. Solution Options

2.1 A List

We could use a list. Algorithms such as *insertionsort* and *selectionsort* can sort the data in $O(N^2)$ time. (We soon will study improved sorting methods that sort in $O(N \log N)$ time.) If the list was already sorted, the first element (largest or smallest) can be removed from the beginning of the list in $O(1)$ time.

Since elements are dynamically added and removed, however, we would have to re-sort the data each time a new element is added.

2.2 A Priority Queue

We want a data structure that maintains a collection of elements and provides the following two operations efficiently:

- Add: Insert a new element into the collection.
- Remove_Min*(or Max)*: Remove the smallest (largest) element from the collection.
- Peek_Min*(or Max)*: Access the smallest (largest) element in the collection without removing it.

This data structure is known as a *priority queue*.

Order by Removal or Insertion?

Until now, we have learned two approaches for implementing priority queues. We could add to the list in *insertion order* and search for the minimum or maximum when we remove. Or the priority queue could add to the list in *sorted order* and remove the first one to produce the minimum/maximum element.

A complete re-sort is not needed for either approach, and either can use an array or a linked list. The bad news is that one or two of the three operations will take ***O(N)*** time.

Here is how these approaches perform the two essential operations:

1. Priority Queue implemented by an Insertion Order List
    - add = "add to front of list", ***O(1)***
    - remove_min = "search for smallest item in list and remove and return it", ***O(N)***
    - peek_min = “search for smallest item in list and return it", ***O(N)***
2. Priority Queue implemented by a Sorted Order List
    - add = "search for proper spot in sorted list and insert there", ***O(N)***
    - remove_min = "remove and return the item from the front of the list", ***O(1)***
    - peek_min = "return the item from the front of the list", ***O(1)***

There is a third way, however; it is called a Heap.

3. The Heap

When we try to improve an algorithm that runs in **O(N)** linear time, our first thought should be, "Can we reduce that to ***O(log N)*** time?"

Two possibilities should come to mind:

Divide and Conquer: Split the problem into smaller problems, solve them, and put the answers together to get the final answer. The divide and conquer approach used for the faster sorting techniques bears no fruit for this problem. Finding the location to insert into a sorted list can be done with binary search, but that requires indexing an array; actually adding the element in the right spot requires O(N) time due to element shifting.

A Tree Data Structure: If the data can be arranged in a reasonably balanced tree so that search distances are no worse than the distance from a leaf to the root or vice-versa, then ***O(log N)*** time can be achieved. The tree approach can help. We will create a special binary tree called a *heap*.

Heap: a binary tree where each node that has children contains an element whose (priority) value is ordered before all of its children's element values. This ordering property is true recursively through the tree. The ordering can be less than or greater than, which means we can create *min-heap* or a *max-heap* respectively.

The tree in Figure 1 is a *min-heap* because the element value in each node is less than or equal to the element value in either of its child nodes, if they exist.

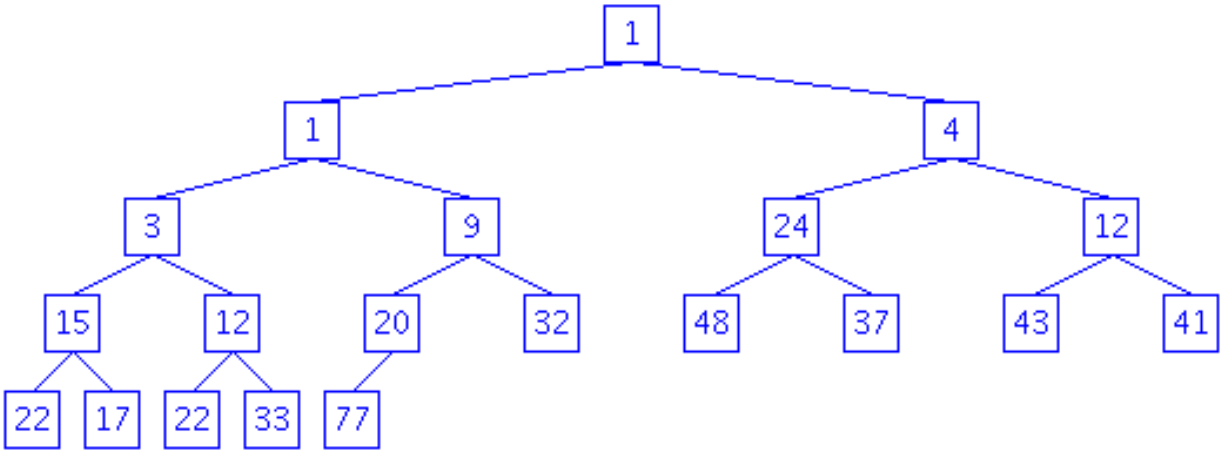


Figure 1. A Heap of 20 numbers

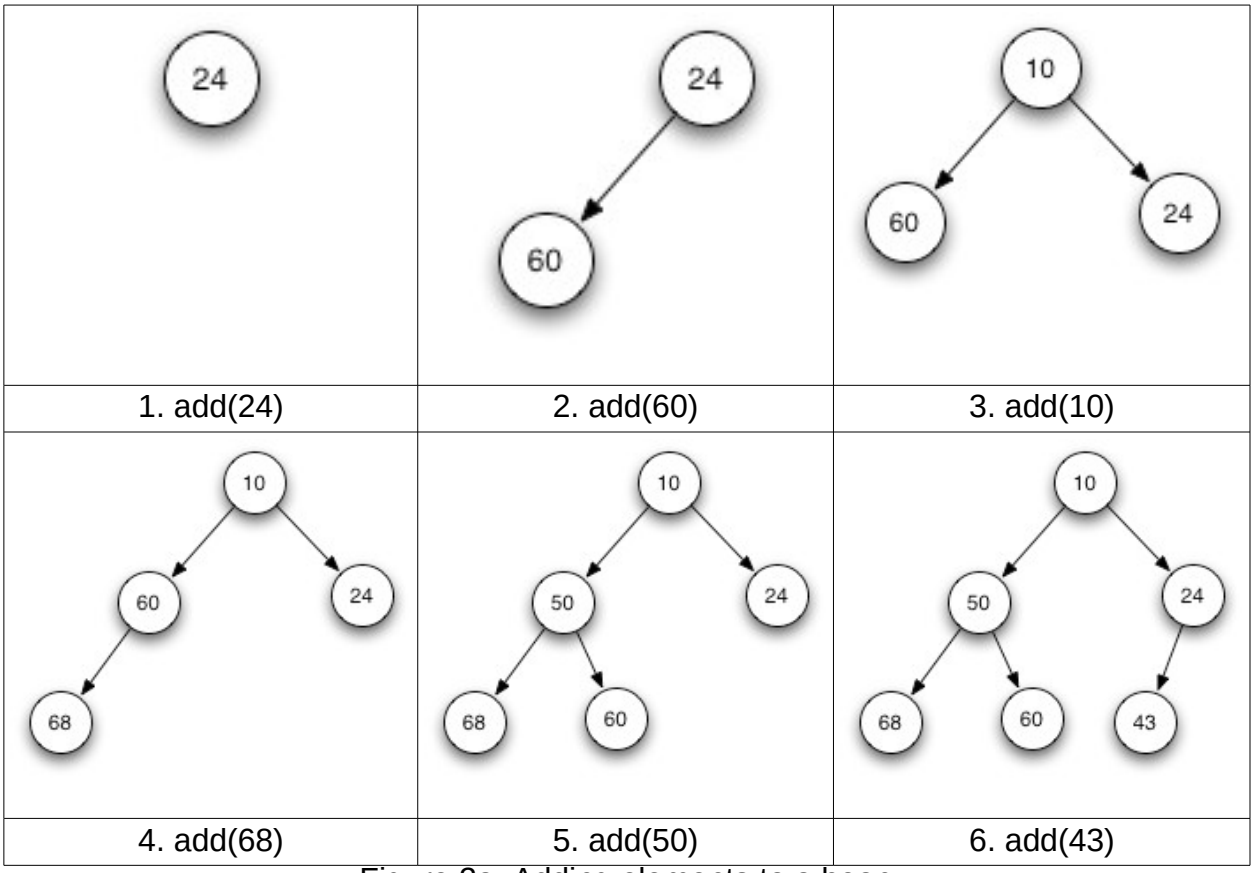
If you have the latest Java installed, you can experiment with an animation at http://window.terratron.com/~sosman/computers/java/heap/ to observe heap operations dynamically.

A heap is a weaker arrangement than a sorted binary search tree. There is no "horizontal" relationship. There is no tree traversal order that can guarantee an *in-order* traversal of the elements.

3.1 Adding to a Heap

We build a heap by adding a new element at the first available leaf location and perform swap operations to move the element up to its correct, heap-order place in the tree. Since we follow only one path from a leaf up to the right location, we achieve the desired complexity of a ***O(log N)*** insertion. Study the sequence of heap operations in Figure 2a. Analyze how each added element moved around so that it arrived at its location.

Notice that this 'heap tree' always remains ***complete***. A heap must not only maintain an ordering property; it must be complete: every level of the tree is completely full with the possible exception of the bottom level, and the bottom level's nodes must all be *flush-left* with no gaps in their arrangement.



1. add(24)	2. add(60)	3. add(10)
4. add(68)	5. add(50)	6. add(43)

Figure 2a. Adding elements to a heap

Here is the basic algorithm for adding a new value to a min heap.

```
add( v ):
        Place a new node containing v in the leftmost open position in the bottom level.
        (If that level is full, place the node at the leftmost position in a new bottom  level.)
        While the node is not the root and its parent's value > v:
                Swap this node's value with its parent's value.
                Move up to the parent and make it the node being checked.
```

The while loop performs a process commonly called *sifting up*. It is very similar to the insertion operation in the insertion-sort algorithm. Since the node starts at a leaf position and then (in the worst case) is sifted all the way to the root of the tree, the sift-up operation has a time complexity of ***O(log N)***. Finding the leaf position to place the new node has a time complexity of ***O(log N)*** or, in some efficient implementations of heaps, a time complexity of ***O(1)***. Therefore, the add operation has time complexity ***O(log N)***.

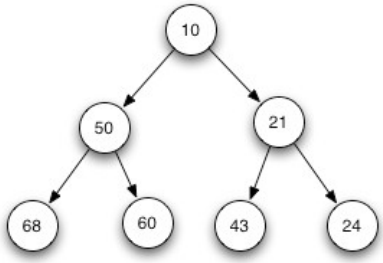
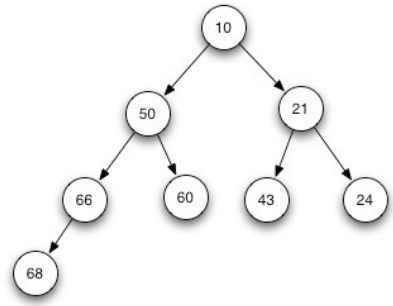
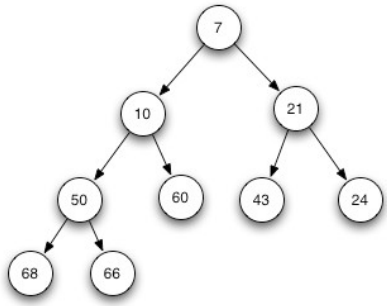
<table>
<tr><td> 
</td><td> 
</td><td> 
</td></tr>
<tr><td>7. add(21)</td><td>8. add(66)</td><td>9. add(7)</td></tr>
</table>

Figure 2b shows result of the additions of 21, 66, and 7 to the heap.

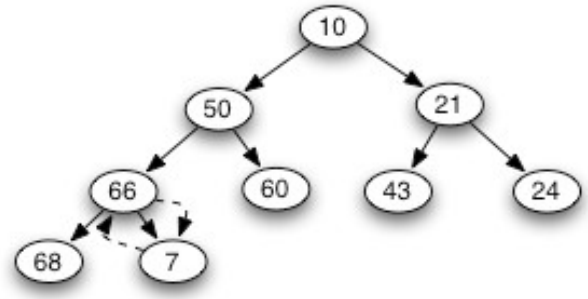
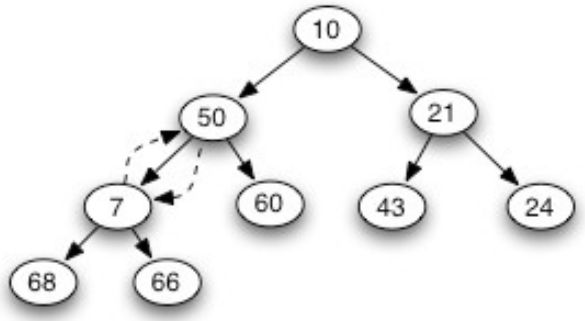
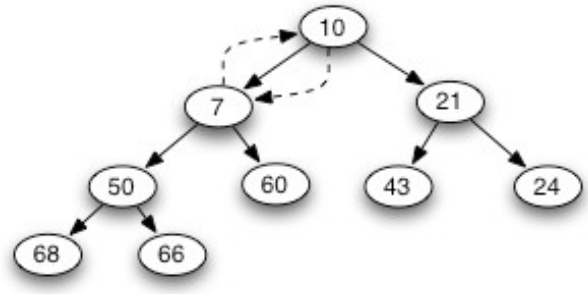
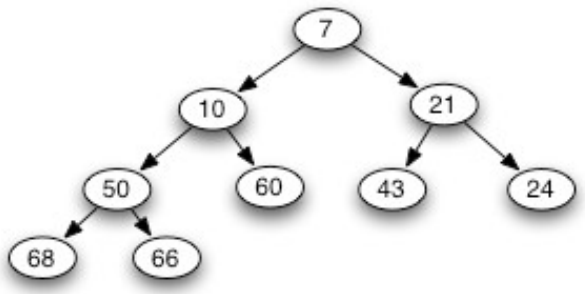
<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>1. 7 is placed at the next spot in the tree. Since 7 < 66, 7 must move up.</td><td>2. Since 7 < 50, swap.</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>3. Since 7 < 10, swap.</td><td>4. Heap is now valid again.</td></tr>
</table>

Figure 2c. shows details of the last addition, 7, to the heap.

3.2 Removing From a Heap

Removing elements works in an opposite manner. The element we want to remove is at the root. Figure 3a shows results of a sequence of `remove_min` operations.

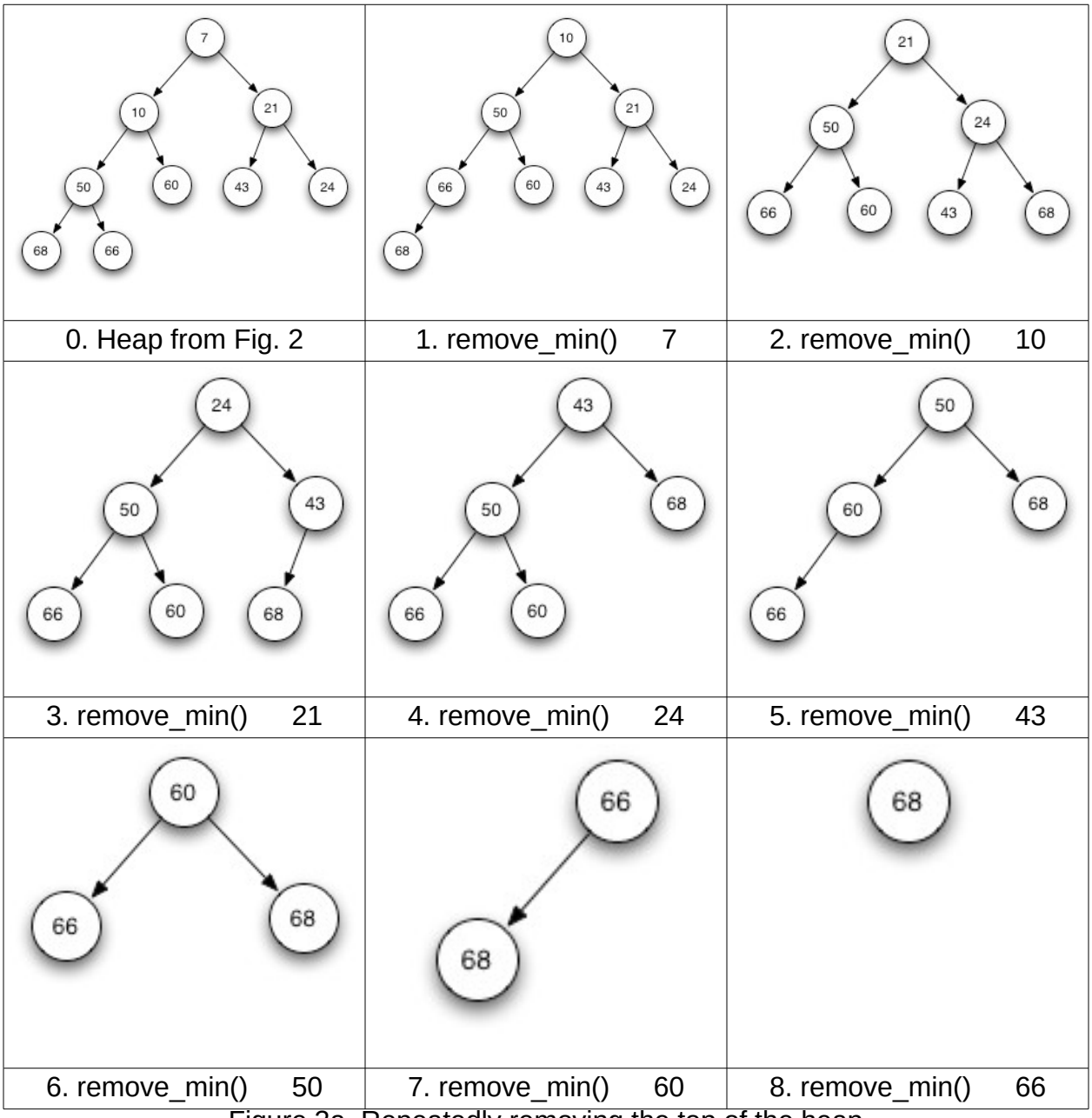


Figure 3a. Repeatedly removing the top of the heap

The removal algorithm is a bit more complex. To keep the tree complete, we replace the element at the root node with the element at the node in the last position in the tree, and the tree shrinks in size. That element must then be moved to its proper position to re-establish the heap ordering property. Moving an element downward involves a *three-way comparison* to choose one of three node locations for the repositioned element.

```
remove_min()
       Save the value found in the root node so that it may be returned by this function.
       Remove the node at the last position in the bottom level of the heap; put its value in v.
       Set the root's value to v.

       Let n be the root node.
       While n has children and n's value is not less than both of its children's values:
              Swap n's value with that of the child node of n that has the smaller value.
              Let n be the child node whose value was changed.

       Return the old root value that was saved.
```

The remove operation's while loop performs a process known as *sifting down*. Since the element starts at the root node position and then is sifted all the way to a leaf node of the tree (in the worst case), the sift-down operation has a time complexity of ***O(log N)***. Finding the leaf element to replace the root element has a time complexity of between ***O(1)*** and ***O(log N)*** depending on the implementation. If accessing the last node's element is ***O(1)***, the remove-min operation has time complexity ***O(log N)***.

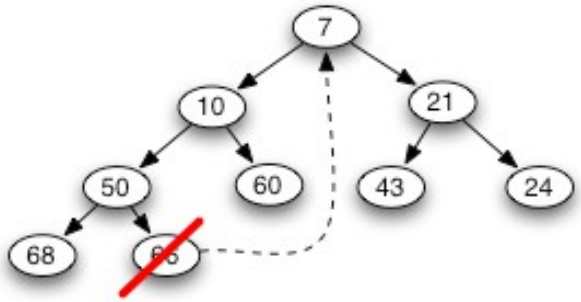
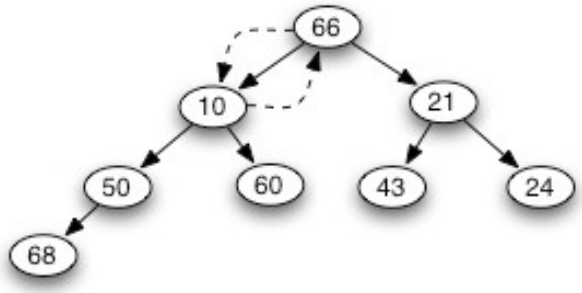
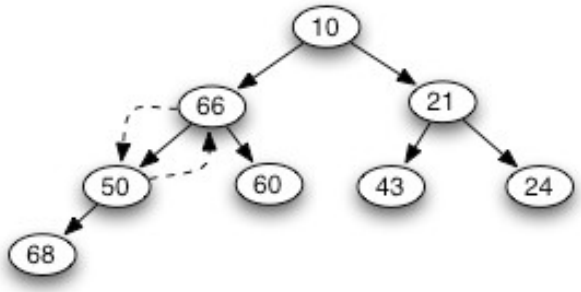
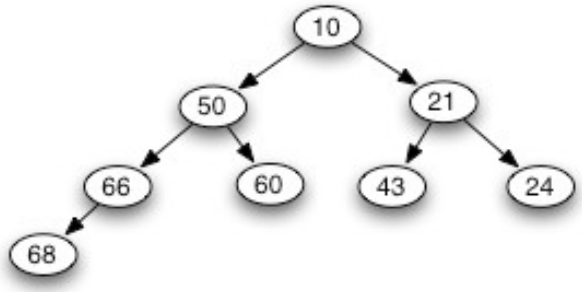
<table>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>1. Beginning state of the tree. 7 must be removed, and the last node's value 66 will replace it at the root.</td><td>2. Since 10 is smallest of 66, 10, and 21, swap its value with the root's value (66).</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>3. To fix the left subtree, 50 must move up. The value 66 must go down to the left.</td><td>4. Since 66 < 68 and there is no right child, 66 must stay put. The heap is restored.</td></tr>
</table>

Figure 3b. Details of removing the minimum value from a heap.

3.3 Linked Node Heap Design

In an application such as the priority queue for the traffic simulator, the maximum size is unknown. A linked implementation of the tree may be best if a large number of insertions and removals are expected. A TreeNode class would need these attributes:

- The element, which must contain some ordering value;
- A left child TreeNode;
- A right child TreeNode;
- A parent TreeNode;

The parent link is important to enable sifting up.

Figure 4 illustrates such a construction.

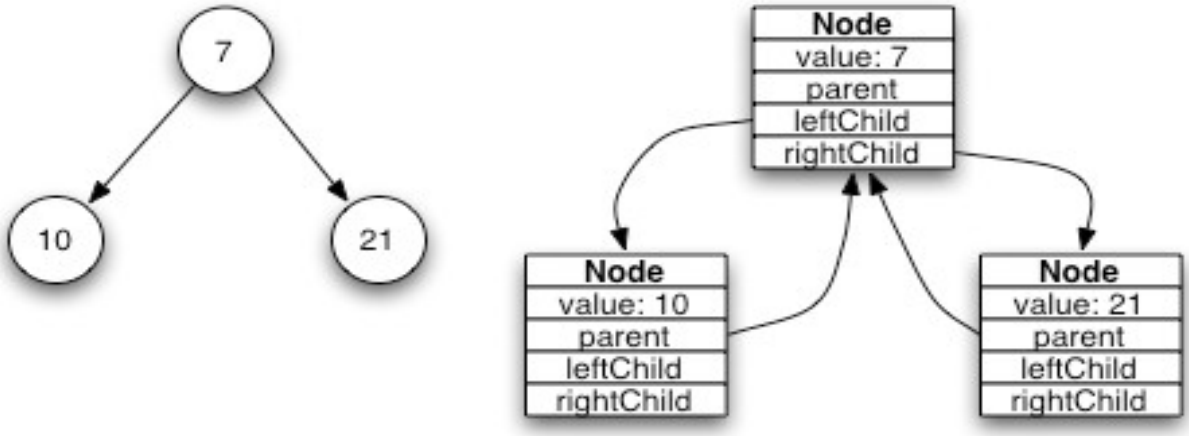


Figure 4. A Linked Node data structure for a binary tree.

4. Sorting with Heaps

We can also use a heap to sort a collection of elements. The first heapsort algorithm is written in a simple way that performs a lot of copying, using a lot of extra storage.

```
heapsort( list ):

       Create an empty heap.
       While the list is not empty:
              Remove the next value from the list; call it v.
              Add v to the heap.

       While the heap is not empty:
              Remove the next value from the heap; call it w.
              Append w to the list.
```

Since the element removed from a min heap is always the smallest, the result will be a sorted list.

4.1 Heapsort Time Complexity Analysis

Insertions take ***O(log N)*** time, and *N* insertions are done. Therefore, the first while-loop has time complexity ***O(N log N)***.

Removals take ***O(log N)*** time, and *N* removals are done; another ***O(N log N)***.

Since this is the whole algorithm, heapsort has a time complexity of ***O(N log N)***.

4.2 Sorting with Heaps Inside Arrays

Since in sorting we know the size of the list to be sorted, we can overlay the tree structure onto the original list itself. Of crucial importance is the indexability of the list. In other words, we want an array.

The overlay is organized so that visiting the elements in the array from left to right is the same as doing a breadth-first traversal of the tree. For now, let's pretend that the indices of the array range from 1 to *n*. It makes the formulae that follow easier to see.

The root, being first, is at index 1. For the node at position *k* in the array, its parent will be at position ***floor(k/2)***, its left child will be at position ***2k*** and its right child will be at position ***2k+1***.

Adjusting the formulae for the standard array layout where the first index is 0, not 1:

- The parent of the node at index *k* is ***floor(k-1)/2)***.
- The left child of the node at index *k* is ***2k+1***.
- The right child of the node at index *k* is is ***2k+2***.

For each of these, we must also check that the result is within the bounds of the array (greater than or equal to 0 and less than the length of the array). Note that the formula for the parent of the node at index 0 will produce -1, which is not within the bounds of the array; this is correct as the node at index 0 is the root node, which has no parent.

Why is it useful to implement the heap tree using an array?

Because it allows us to do a sort of the array *in situ*. The heap sort algorithm still has two parts. The first part, often called *heapify*, transforms the unordered array into a heap, and the second part consumes the heap and replaces it with the content of the sorted list.

In the array implementation we can establish an *invariant* in the first loop:

> *The array is divided so that the first section is a heap and the second section is unordered.*

The first section grows steadily as the second section shrinks. The heapify loop ends when the second section is empty.

We can establish a similar invariant in the second loop of the heap sort algorithm:

> *The array is divided so that the first section is a heap and the second is sorted.*

For the second part of the algorithm, the first section steadily shrinks as the second section grows. This part ends when the second section occupies the entire array.

We can now add details to the heapsort algorithm. The comments describing the invariants and other assertions are in **bold**.

```
sort( list ):

        Set h to 1.
        At this point, there is a heap of size 1 at the start of the array
        and n-1 unordered elements at the end.
        While h < n:
                Increment h.
                The heap is now invalid due to the value added at position h.
                Fix it by sifting the value added at position h up to its proper location.
                At this point, there is a heap of size h at the start of the array
                and n-h unordered elements at the end.

        The heap now occupies the entire array.

        Set h to n-1.
        At this point, there is a heap of size h at the start of the array
        and n-h sorted elements at the end.
        While h > 0:
                The minimum value in the heap is at position 0, the root.
                Swap the value at position 0 with the value at position h.
                Decrement h.
                The heap is now invalid due to the new value placed at the root.
                Fix it by sifting the value added at position 0 down to its proper location.
                At this point, there is a heap of size h at the start of the array
                and n-h sorted elements at the end.

        The sorted section now occupies the entire array.
```

The time complexity of the sort has not changed; the first loop executes ***O(N)*** times, and within each cycle, the sifting up takes ***O(log N)*** time. The second loop is similar: ***O(N)*** executions of an ***O(log N)*** sifting down operation. We have another ***O(N log N)*** sorting algorithm!

There is one little problem with this approach. Because we have been using a min heap, the sorted list comes out ... *backwards*! Consequently, we might want to use a *max heap* for sorting. Nothing substantial changes; we simply replace "less than" with "greater than" and vice-versa in the algorithms.

For a simple animation of heap-sort, please see
http://www.ee.ryerson.ca/~courses/coe428/sorting/heapsort.html

If you have the latest Java installed, it is possible to experiment interactively with an on-line max heap:

http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/heap/heapen.htm

On the next page, Figure 5 shows the complete sorting of six elements; this includes the heapifying stage (A1-A6) and the transformation of the heap into a sorted list (B1-B6).

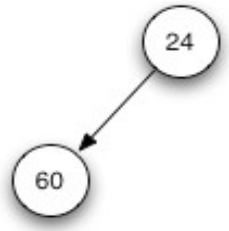
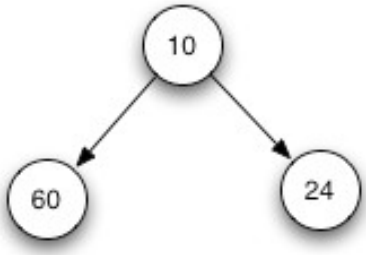
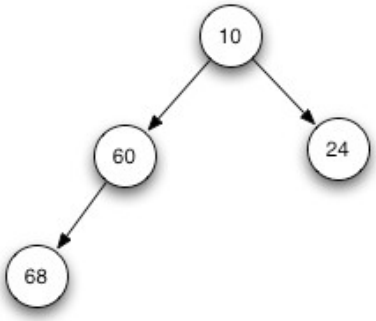
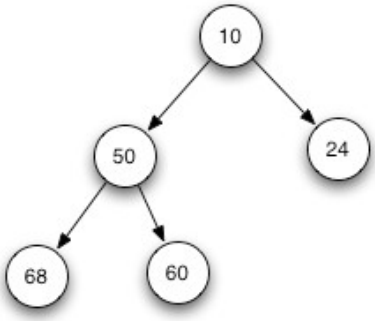
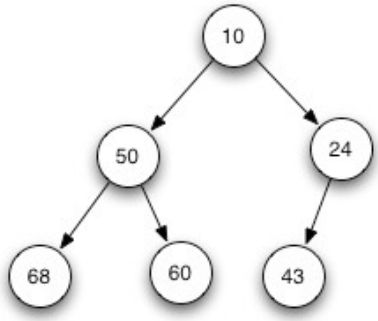
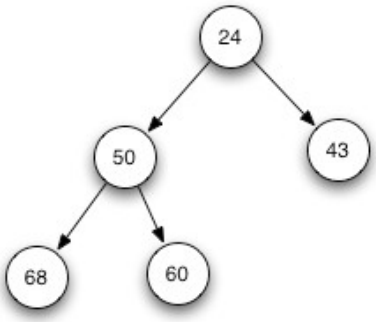
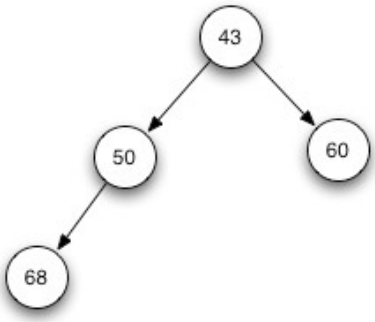
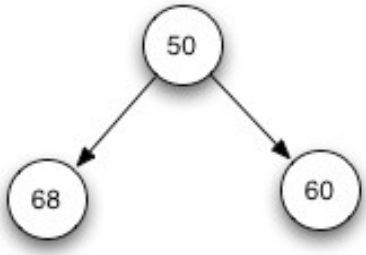
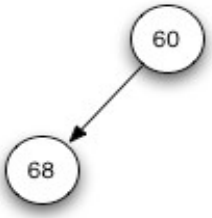
		
A1. [24, 60, 10, 68, 50, 43]	A2. [24, 60, 10, 68, 50, 43]	A3. [10, 60, 24, 68, 50, 43]
		
A4. [10, 60, 24, 68, 50, 43]	A5. [10, 50, 24, 68, 60, 43]	A6. [10, 50, 24, 68, 60, 43]
		
B1. [24, 50, 43, 68, 60, 10]	B2. [43, 50, 60, 68, 24, 10]	B3. [50, 68, 60, 43, 24, 10]
		
B4. [60, 68, 50, 43, 24, 10]	B4. [68, 60, 50, 43, 24, 10]	B4. [68, 60, 50, 43, 24, 10]

Figure 5. Heap Sort in an Array