Unit 13 – The Heap

Unit 13.1 – Heap Basics

The heap is built using a complete binary tree. It is also known as a “priority queue”. Recall that the queue we studied before was the FIFO queue. The heap returns the smallest (or largest) element in the structure next.

The heap has the following properties:

- **shape property**: a complete binary tree
- **order property**: each node is smaller than both of its children.

Recall that a complete binary tree is a perfect tree with nodes added at the lowest level starting on the left. We will see later why this is important. Here are some complete binary trees as a refresher:

![Complete Binary Trees](image)

Sometimes you want the order property of the heap to be reversed (each node is larger than both its children). It is easy to change between these two cases by swapping the “>” and “<” operators in the code.

First let’s build a heap. The method is called “enqueue”. A new node always goes at the lowest level from left to right. Adding a new node might break the order property, in which case we do a “reheap up” operation to restore the order property.

Here’s pseudocode for reheap up:

1. If a node is smaller than its parent, swap with parent, move to parent.
2. Repeat to root if necessary.

Let’s build a heap with these values: 25, 16, 9, 12, 10, 6.

After adding 25 we have (trivially):
Next we add 16. Initially it goes in as the left child of 25.

However, we’ve broken the order property. We need to do the reheap up operation. We start at 16. Compare 16 with its parent 25 and we see that we need to swap since $16 < 25$. After the swap the order property is restored and it is a valid heap again.

Next we add 9. It goes to the right of 16.

The order property is broken again. We reheap up starting at 9. We swap with 16 since $9 < 16$.

It is a valid heap again. Next we add 12. The next available location (since the tree is perfect) is to the left of 25. The order property is broken again.
We reheap up starting at 12.  We swap with 25 since 12<25.  After one swap the order property is restored (since 12>9).

Next we add 10. It becomes the right child of 12.

The order property is broken again. We reheap up starting at 10. We swap with 12 because 10<12. The order property is now restored.

It is interesting to note that 12 is at a lower level than 16. The order property of the heap has nothing to say about the sizes of keys that are side-by side. Only that each node is larger than both its children (and thus all the keys in its left and right subtrees).
Finally, we add 6. I chose to add a small value last so that we could see the recursion in the reheap up method. Initially 6 becomes the left child of 16.

![Tree diagram with nodes 9, 10, 16, 25, 12, 6]

The order property is broken. We start the reheap up at 6 and swap with 16 since 6<16.

![Tree diagram after reheap up]

The order property is still broken because 6<9. We swap again.

![Tree diagram after reheap up]

Now the order property is restored and we have a valid heap again.

To dequeue from a heap we remove the item from the root node and store it in a temporary variable. We then copy the item from the last node (bottom level all the way on right) into the root node. We then remove the last node. To restore the order property we do a “reheap down” operation. It goes as follows:

**Start at a node. If the key at root is larger than either child, swap with smallest child. Move to that child. Repeat to leaf node if necessary.**

Consider the following heap:

![Tree diagram with nodes 9, 10, 16, 25, 12, 6]
The first dequeue operation will return B. Then H is copied into the root node and the node that had H is removed.

The order property is broken, so we reheap down to restore it. The reheap down will start by swapping H with C (since C<D, swap with smallest child).

The order property is still broken since H>G. We continue the reheap down by swapping G and H.

Now it is a heap again.

The next dequeue will return C. H is copied to root and the node that had H in it is removed.
This time the reheap down will swap H and D (since D<G, swap with smallest node).

The order property is still broken since H>E and H>F. We continue the reheap down by swapping H with E (since E<F). The order property is now restored.

The next dequeue returns D, copies F into root, and removes the last node.

The reheap down will swap F and E, at which point the order property of the heap is restored.
Unit 13.2 – Efficiency of Heap Methods

Reheap up and reheap down are both $O(\log_2 N)$ since they follow a single path from root to leaf or vice versa. Since a complete tree is always perfect up to the last level, we don’t have issues with balance of nodes in the tree like we did with the BST. We never do a traversal (unless you are debugging).

Thus all heap operations are $O(\log_2 N)$.

Unit 13.3 – Array Representation of a Heap

Heaps can be stored as binary trees as was done in the BST, but this is usually not done. Because the heap is a complete binary tree it is usually stored as an array. To see this, consider the following heap and its equivalent array:

We populate the array as follows: the first level takes one element. The second takes two. The third, four, etc. The last level is only partially full and can take any number from 0 to $2^{n-1}$ where $n$ is the level. Note that since the heap is a complete binary tree there won’t be any gaps in the array.

There are simple formulas for finding a parent, left child or right child from the array indices. Here they are ($n$ is the array index):
Unit 13.4 – The Heap ADT

Here's the basic structure of the heap ADT. We will always use an array to store the heap.

```java
public class CM307Heap<E extends Comparable<E>> {
    private int size=0;
    private E[] heap;

    public CM307Heap()
    {
        heap=(E[])new Comparable[10];
    }
}
```

There should be a reallocate method in order to increase the array size when it is full. It simply creates an array twice as big and moves the items over:

```java
private void reallocate()
{
    E[] newHeap=(E[])new Comparable[heap.length*2];
    for(int i=0;i<size;i++)
    {
        newHeap[i]=heap[i];
    }
    heap=newHeap;
}
```

Let's start with the enqueue method. Recall the theory. When we enqueue into a heap we add the new element into the next position in the tree. That will be at the lowest level as far to the left as possible. In the array representation we just add after the last element (at size) and increase the size by 1.

We then need to reheap up to restore the order property of the heap. We will visualize this in a tree, but it is really happening in the array.

Here's the code:
public void enqueue(E item)
{
    if(size==heap.length) reallocate();
    heap[size]=item;
    size++;
    reheapUp(size-1);
}

The reheapUp method is recursive. It takes an array index as its argument. It calculates the location of the parent from the formula p=(n-1)/2. Recall that it will swap with the parent if the current key is smaller than the key of the parent. If so, it continues to reheapUp all the way to root (the base case).

Here’s the code for reheapUp( ):

private void reheapUp(int n)
{
    if(n==0) return;
    int p=(n-1)/2;
    if(heap[n].compareTo(heap[p])<0)
    {
        E temp=heap[n];
        heap[n]=heap[p];
        heap[p]=temp;
        reheapUp(p);
    }
}

Let’s do a walkthrough starting with this heap:

Let’s enqueue the key “6” into the heap. The first step is to add 6 in the next location (at the end of the array).
The first call to reheapUp( ) call starts with n=5 and p=2. Since 6<15 a swap is done. Since a swap was done, we continue deeper into recursion.

The second call has n=2 and p=0. Since 6<7 we swap again and go deeper into recursion.

In the third call we hit the base case and the recursion ends.

The dequeue method is shown below. We store the item to be dequeued, reduce the size of the heap, copy the last element to root, reheap down and return the stored element.
public E dequeue()
{
    E item=heap[0];
    size--;
    heap[0]=heap[size];
    reheapDown(0);
    return item;
}

The reheapDown method is a little tricky. Recall that reheap down starts at root and compares the key at root to the keys of both children. It is possible that a node has a left child but not a right child. If you try to access a right child that doesn’t exist you can get an ArrayIndexOutOfBoundsException but only if the array is just the right (or wrong) size.

private void reheapDown(int n)
{
    int left=2*n+1;
    int right=2*n+2;
    if(right<size) // has 2 children
    {
        if(heap[left].compareTo(heap[n])<0||heap[right].compareTo(heap[n])<0)
        {
            if(heap[left].compareTo(heap[right])<0)
            {
                E temp=heap[n];
                heap[n]=heap[left];
                heap[left]=temp;
                reheapDown(left);
            }
            else
            {
                E temp=heap[n];
                heap[n]=heap[right];
                heap[right]=temp;
                reheapDown(right);
            }
        }
        else
        {
            E temp=heap[n];
            heap[n]=heap[left];
            heap[left]=temp;
            reheapDown(left);
        }
    }
    else if(left<size) // has only left child
    {
        if(heap[left].compareTo(heap[n])<0)
        {
            E temp=heap[n];
            heap[n]=heap[left];
            heap[left]=temp;
        }
    }
}

Let’s do a walkthrough on the following heap.
The 5 is stored to be returned later. The last element, 16 is copied to the root node. Size is decremented and reheapDown is called with n=0 (the root node).

In the first call n=0, left=1 and right=2. There are two children so we do the if(right<size) and not the else. Next we check to see if 16 is less than 7 or less than 11, which is true. Next we check if 7<11, which it is. We therefore swap the 16 with the 7 and go deeper in recursion. In the second call n=1, left=3 and right=4.

There are two children so we check to see if 16<12 or 16<15 which is true. We then compare the left child with the right child 12<15, which is true, so we swap the 12 and the 16 and go deeper into recursion.
In the third call n=3, left=7 and right=8. We finally hit the base case (which is subtle, notice there is no else to match the if(right<size)/else if(left<size)). The base case happens when neither of these is true, in which case we’re done.