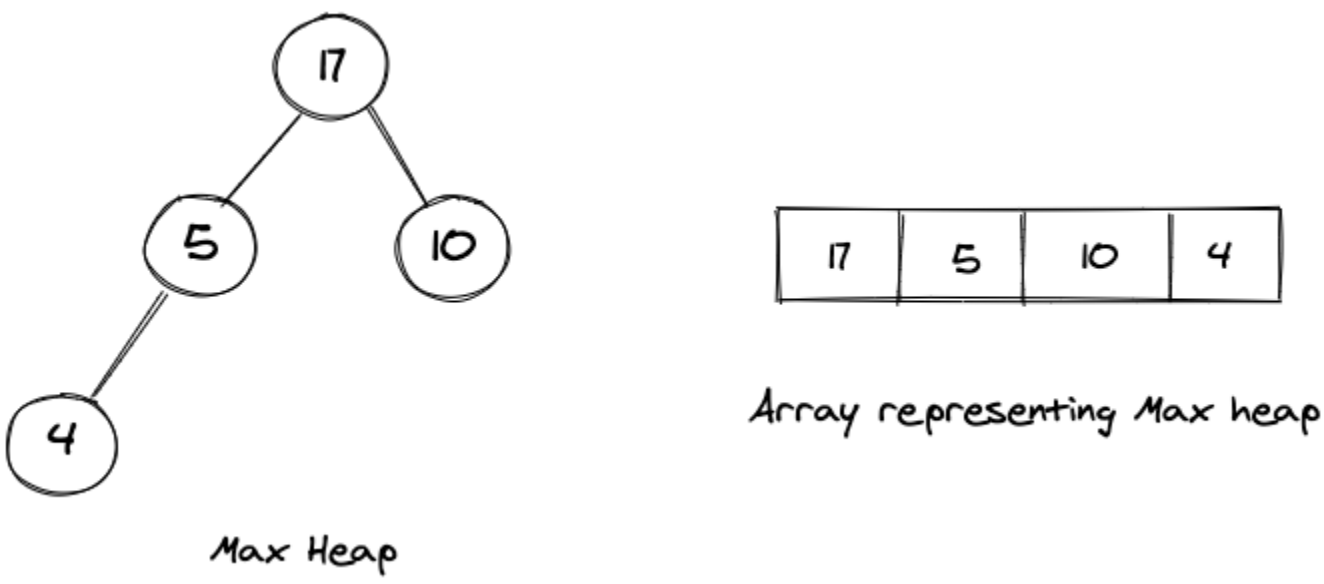
Implementing Heaps

In the previous article we learned all about heaps. It is time to implement them. Arrays are a preferred choice for implementing heaps because of the heap's complete binary tree property, we can be assured that no wastage of array locations will be there.

Let us consider a binary heap shown below, this heap can be represented by an array also shown in the pictorial representation below.



Implementing Heap

Let's see how we can implement a Heap data structure.

1. Structure of Heap

We need an array to store the elements of the heap, besides that we will also have a variable that represents the size of the heap. Consider the code snippet below:

```
private int[] items = new int[10];

private int size;
```

Copy

The initial capacity can be dynamic also, but here I have chosen a fixed length of 10 to represent the same.

2. Inserting into a Heap

When we want to insert an item into a heap, we will simply insert it into the array and then do a `bubbleUp()` operation if it doesn't satisfy any of the heap properties. The code will look something like this:

```
private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();
```

```
}
```

Copy

A case might arrive where the heap is full, and then if we try to insert any more items into it, it should return an Exception. The `isFull()` method looks like this:

```
private boolean isFull(){ return size == items.length; }
```

Copy

After this `isFull()` call returns false, we move ahead and insert the item into our heap, and increase the size of the array as well. Now, the most important part is to correct the position of this element that we have inserted in the heap. We do this using `bubbleUp()` method. The code of `bubbleUp()` method is shown below:

```
private void bubbleUp(){

        int index = size - 1;

        while(index > 0 && items[index] > items[parent(index)]){

            swap(index,parent(index));

            index = parent(index);

        }

}
```

Copy

In the above method we are taking the index of the element we just inserted and then swapping it with the parent element in case this element is bigger than the parent element, also we are updating the index so that we don't pick the wrong element. The `swap()` method looks like this:

```
private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

}
```

Copy

It should also be noted that we are calling a parent method in our `bubbleUp()` method, and that parent method returns us the parent of a current index. The code of this method looks like this:

```
private int parent(int index){ return (index - 1)/2; }
```

Copy

The above methods are all we need to insert an element into a binary heap.

3. Removing from a Heap

Removing an element from a heap is a bit tricky process as when compared to inserting into a heap, as in this we need to `bubbleDown()` and this process of bubbling down involves checking the

parent element about its validity, also methods to insert into the tree as left as possible to maintain the complete binary tree property.

The `remove()` method looks like this:

```
private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

}
```

Copy

The first check is to make sure that we are not trying to remove anything from an empty heap. Then we insert the element at the start, and then `bubbleDown()` the element to it's correct position. The `bubbleDown()` method looks like this:

```
private void bubbleDown(){

        int index = 0;

        while(index <= size && !isValidParent(index)){

            int largerChildIndex = largerChildIndex(index);

            swap(index,largerChildIndex);

            index = largerChildIndex;

        }

}
```

Copy

We are checking that the index we have should be less than the size and also making sure that the parent node should not be a valid (should be less than the child values and other checks). The `largerChildIndex()` method return us the children item with which we will replace this current item while bubbling downwards. The code for `largerChildIndex()` looks like this:

```
private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

}
```

Copy

The code for isValidParent() method is shown below:

```
private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

}
```

Copy

In the above code snippet we are just making sure that the parent is valid, and also if the left child is null then we return true. The leftChild(), rightChild(), hasLeftChild(), hasRightChild() methods looks like this:

```
private int leftChild(int index){

        return items[leftChildIndex(index)];

}



private int rightChild(int index){

        return items[rightChildIndex(index)];

}



private boolean hasLeftChild(int index){

        return leftChildIndex(index) <= size;

}



private boolean hasRightChild(int index){

        return rightChildIndex(index) <= size;

}
```

Copy

The above method calls the methods to get the left and right children of a binary heap element, these are:

```
private int leftChildIndex(int index){

        return 2*index + 1;

}



private int rightChildIndex(int index){

        return 2*index + 2;

}
```

Copy

Now we are done with all the methods we need to insert or delete an element from the heap. The complete code is given below:

```
public class Heap {


    private int[] items = new int[10];

    private int size;


    private void insert(int item){

        if(isFull()){

            throw new IllegalStateException();

        }

        items[size++] = item;

        bubbleUp();

    }


    private void remove(){

        if(isEmpty()){

            throw new IllegalStateException();

        }

        items[0] = items[--size];

        bubbleDown();

    }
```

```
    private int largerChildIndex(int index){

        if(!hasLeftChild(index)) return index;

        if(!hasRightChild(index)) return leftChildIndex(index);

        return (leftChild(index) > rightChild(index)) ?
leftChildIndex(index) : rightChildIndex(index);

    }



    private boolean isValidParent(int index){

        if(!hasLeftChild(index)) return true;

        var isValid = items[index] >= leftChild(index);

        if(hasRightChild(index)){

            isValid &= items[index] >= rightChild(index);

        }

        return isValid;

    }


    private int leftChild(int index){

        return items[leftChildIndex(index)];

    }


    private int rightChild(int index){

        return items[rightChildIndex(index)];

    }


    private int leftChildIndex(int index){

        return 2*index + 1;

    }


    private int rightChildIndex(int index){

        return 2*index + 2;

    }
```

```
    private boolean hasLeftChild(int index){
        return leftChildIndex(index) <= size;
    }

    private boolean hasRightChild(int index){
        return rightChildIndex(index) <= size;
    }

    private boolean isFull(){
        return size == items.length;
    }

    private boolean isEmpty(){
        return size == 0;
    }

    private void bubbleUp(){
        int index = size - 1;
        while(index > 0 && items[index] > items[parent(index)]){
            swap(index,parent(index));
            index = parent(index);
        }
    }

    private void bubbleDown(){
        int index = 0;
        while(index <= size && !isValidParent(index)){
            int largerChildIndex = largerChildIndex(index);
            swap(index,largerChildIndex);
```

```
            index = largerChildIndex;

        }

    }


    private int parent(int index){

        return (index - 1)/2;

    }


    private void swap(int left,int right){

        int temp = items[left];

        items[left] = items[right];

        items[right] = temp;

    }


    private int getMax(){

        return items[0];

    }


    public static void main(String[] args) {

        Heap heap = new Heap();

        heap.insert(10);

         heap.insert(5);

         heap.insert(17);

         heap.insert(4);

         heap.insert(22);

        /// heap.remove();

         System.out.println("Done!");

    }

}
```

Copy

Conclusion

- We learned how to implement heaps, mainly the **bubbleUp()** and **bubbleDown()** methods.