Walmart Internship Task_1

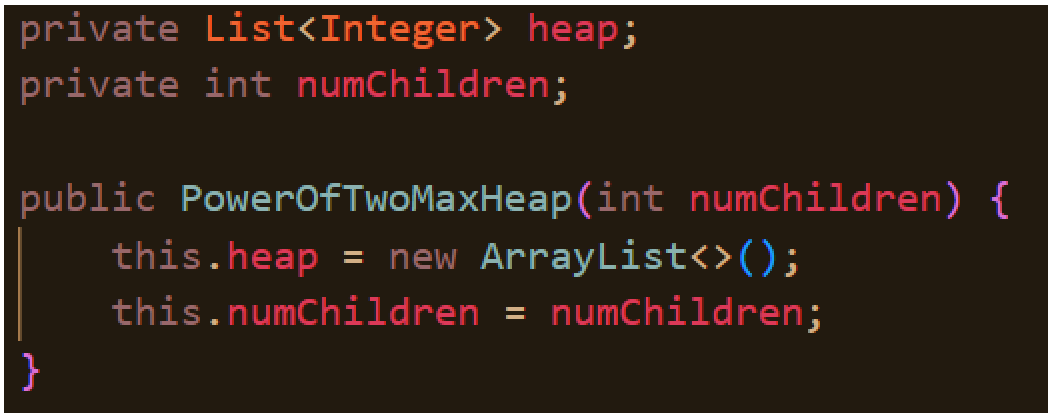
Amit jakhar
amitjakhar@iitbhilai.ac.in
GitHub link for task

Our task is to implement a novel data structure according to the given information
The requirements of the data structure are as follows:

1. The heap must satisfy the heap property.
2. Every parent node in the heap must have 2^x children.
3. The value of x must be a parameter of the heap's constructor.
4. The heap must implement an insert method.
5. The heap must implement a pop max method.
6. The heap must be implemented in Java.
7. The heap must be performant.

Let's take no. of children per node (x) as **numChildren** as we have to take the more descriptive name for x.

You can find the code file in the GitHub repository. But I'm also attaching here images for an explanation.
First, I defined the class name PowerOfTwoMaxHeap as a parent class

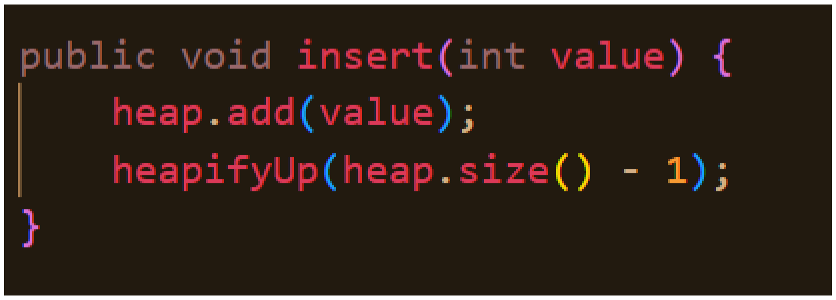


```java
private List<Integer> heap;
private int numChildren;

public PowerOfTwoMaxHeap(int numChildren) {
    this.heap = new ArrayList<>();
    this.numChildren = numChildren;
}
```

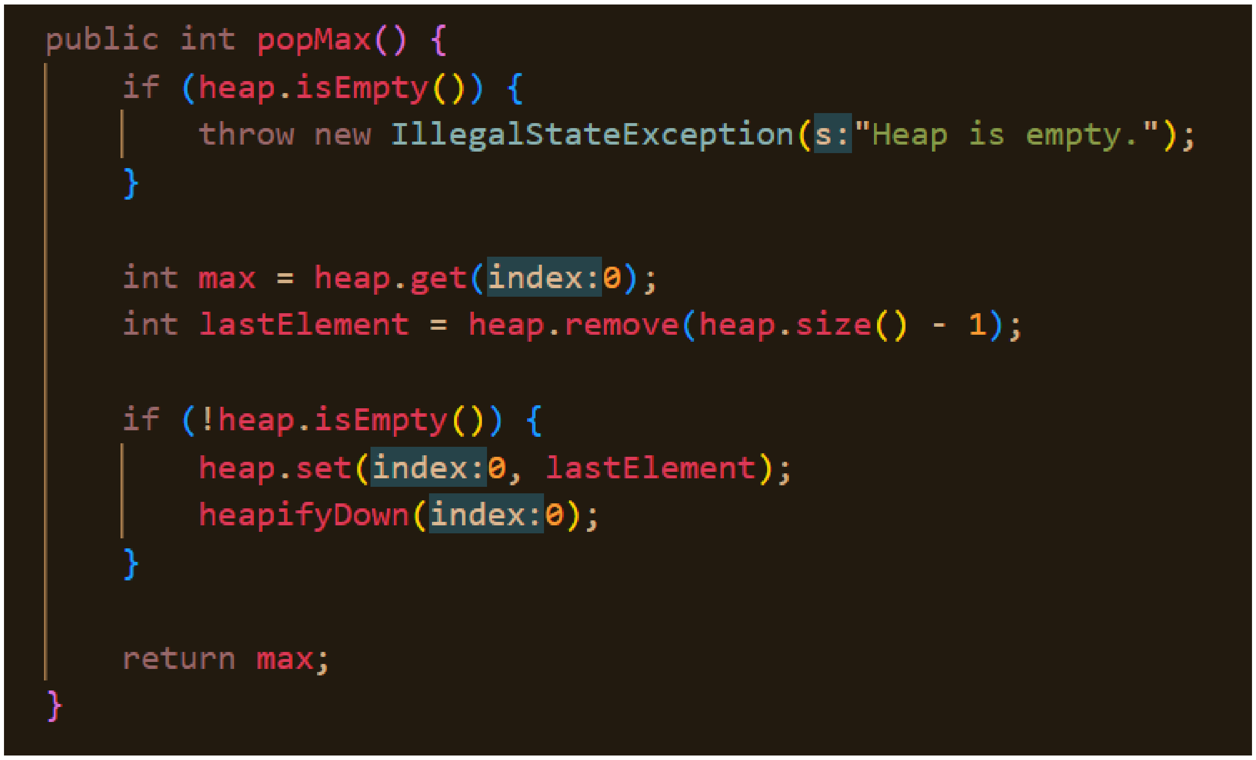
'private List<Integer> heap;' this line declares a private instance variable 'heap', which is of type **'List<Integer>'.** It will be used to store the elements of the power of two max heap.
'Private int numChildren;' this line declares another private instance variable 'numChildren', which represents the number of children each parent node should have in the power of two max heap.

‘Public PowerOfTwoMaxHeap(int numChildren)’ => this is the constructor for the **‘PowerOfTwoMaxHeap’** class. It takes an ‘int’ parameter ‘numChildren’, which specifies the desired number of children per parent node. It initialized the ‘heap’ variable as a new ‘ArrayList<Integer>()’ to store the elements and assigns the ‘numChildren’ value to the ‘numChildren’ variable.



```java
public void insert(int value) {
    heap.add(value);
    heapifyUp(heap.size() - 1);
}
```

The **‘insert’** method adds the new element to the end of the list and then performs the necessary adjustments using ‘heapifyUp’ to maintain the heap property. By moving the element up the heap as needed, it ensures that the maximum element is at the root of the heap.



```java
public int popMax() {
    if (heap.isEmpty()) {
        throw new IllegalStateException(s:"Heap is empty.");
    }

    int max = heap.get(index:0);
    int lastElement = heap.remove(heap.size() - 1);

    if (!heap.isEmpty()) {
        heap.set(index:0, lastElement);
        heapifyDown(index:0);
    }

    return max;
}
```

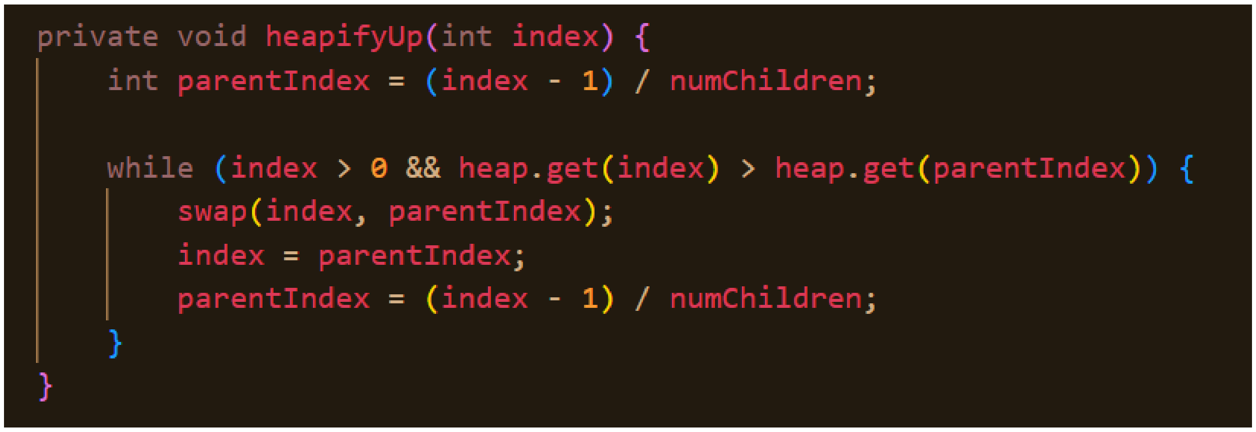
The **‘popMax’** method allows to removal and retrieval the maximum element from the power of two max heap while maintaining the heap property by rearranging the remaining elements using the ‘heapifyDown’ method if necessary.
‘public int popMax()’; this is the **‘popMax’** method, which removes and returns the maximum element from the power of two max heap. It has an ‘int’ return type, indicating that it returns the maximum element.
And then we check if the heap is empty

‘Int max = heap. get(0);’ this line retrieves the maximum element from the heap. In the power of two max heap, the maximum is always stored at the root, which is the first element in the ‘heap’ list.

‘Int lastElement = heap. remove(heap. size()-1);’ this line removes the last element from the ‘heap’ list and assigns it to the ‘last element’ variable. This step is necessary to maintain the size and structure of the heap after removing the maximum element.

And next, the statement checks if the heap is not empty, and if it is not then it replaces the element at the root (index 0) with the ‘last element’ that was removed. Then, it calls the ‘heapifyDown’ method with index 0 to restore the heap property by comparing the new root with its children and swapping if necessary.

And then we **return max** that was initially retrieved from the heap.

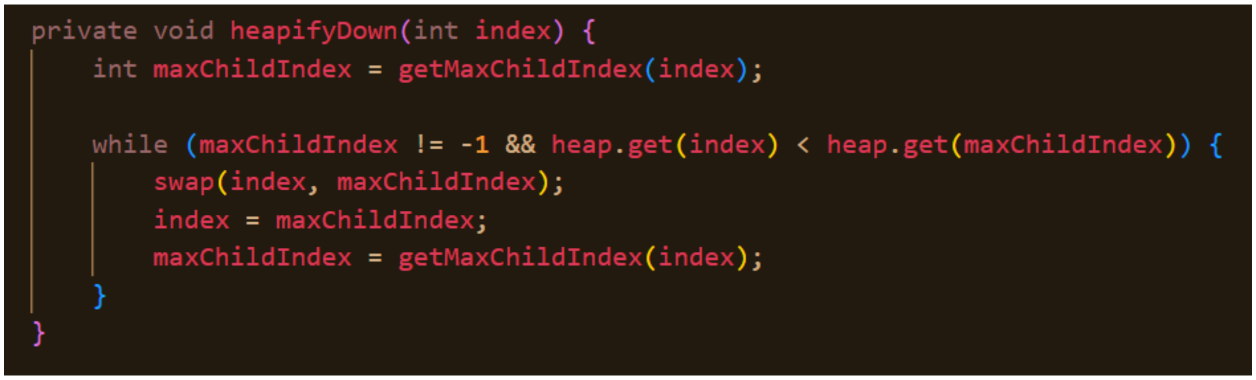

```java
private void heapifyUp(int index) {
    int parentIndex = (index - 1) / numChildren;

    while (index > 0 && heap.get(index) > heap.get(parentIndex)) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / numChildren;
    }
}
```

The **‘heapifyUp’** method is responsible for restoring the heap property by comparing the element at the given ‘index’ with its parent and moving it up the heap if necessary.

‘Int parentIndex = (index-1)/numChildren;’ this line calculates the parent index of the current element by using the formula ‘(index -1)/numChildren’. It determines the position of the parent node in the heap based on the number of children per parent.

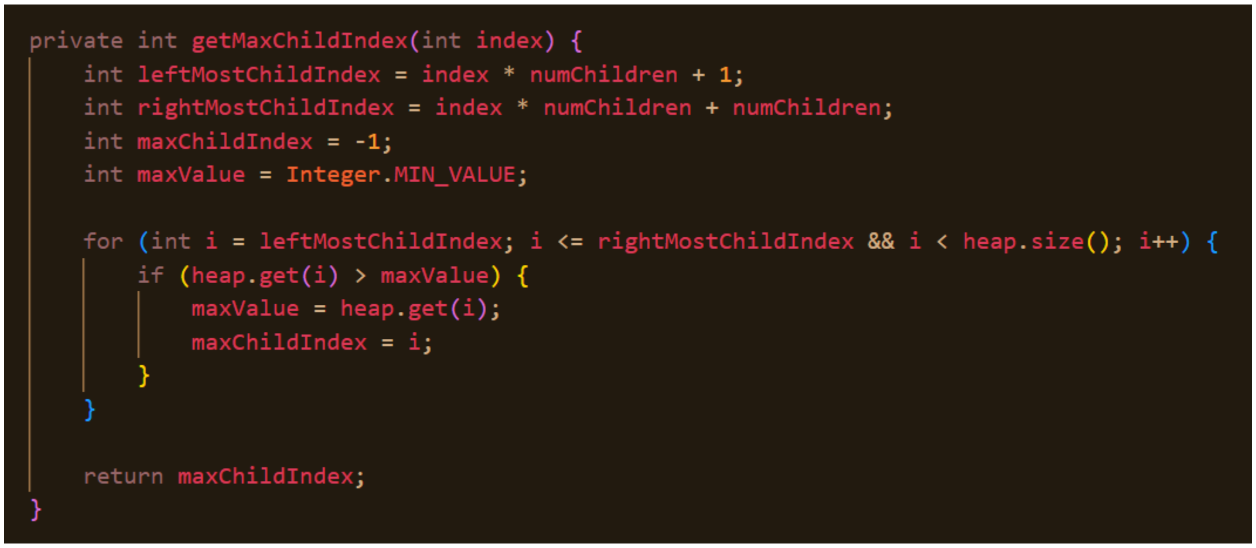
‘While (index > 0 && heap.get(index) > heap.get(pareantIndex)): here a loop that as long as the current element is not at the root and the current element is greater than its parent. This loop ensures that the current element is moved up the heap until it reaches a position where the heap property is satisfied.
‘swap(index,parentIndex);’ If the current element is greater than its parent, this line swaps the current element with its parent. It uses the ‘swap’ helper method to perform the swap operation.
‘parentIndex = (index-1)/numChildren;’: The ‘parentIndex’ is recalculated based on the updated ‘index’. This ensures that the loop continues comparing the element with its new parent until the heap property is satisfied.
By repeatedly swapping the current element with its parent and updating the indices, the **‘heapifyUp’** method moves the element up the heap until if finds a proper position where the heap property is maintained.



```java
private void heapifyDown(int index) {
    int maxChildIndex = getMaxChildIndex(index);

    while (maxChildIndex != -1 && heap.get(index) < heap.get(maxChildIndex)) {
        swap(index, maxChildIndex);
        index = maxChildIndex;
        maxChildIndex = getMaxChildIndex(index);
    }
}
```

The **‘heapifyDown’** method maintains the heap property when removing the maximum element from the power of two max heap. It ensures that the element is placed in the correct position by comparing it with its children and moving it down the heap if necessary.



```
private int getMaxChildIndex(int index) {
    int leftMostChildIndex = index * numChildren + 1;
    int rightMostChildIndex = index * numChildren + numChildren;
    int maxChildIndex = -1;
    int maxValue = Integer.MIN_VALUE;

    for (int i = leftMostChildIndex; i <= rightMostChildIndex && i < heap.size(); i++) {
        if (heap.get(i) > maxValue) {
            maxValue = heap.get(i);
            maxChildIndex = i;
        }
    }

    return maxChildIndex;
}
```

The **‘getMaxChildIndex’** method is a helper method used by the ‘heapifyDown’ method. It determines the index of the child node with the maximum value among all the children of the current element.
Here is how it works.

‘Int leftMostChildIndex = index * numChildren + 1;’: here we calculate the index of the leftmost child node of the current element. In the power of two max heap, the children of a parent node are located at positions **‘(index * maxChildren) +1’ to (index * numChildren) + numChildren’**

‘Int rightMostChildIndex = index * numChildren + numChildren;’
Here we calculate the index of the rightmost child node of the current element.

‘Int maxChildIndex = -1;’ This initializes the ‘maxChildIndex’ variable to -1, indicating that there is currently no maximum child index.

The for loop checks if each child index is within the valid range**(‘i<heap.size()’);**

The **‘getMaxChildIndex’** method helps identify the index of the child node with the highest value among the children of a parent node. This information is crucial in the **‘heapifDown’** method to maintaining the heap property and determining the correct position for the element being moved down the heap.

```
    private void swap(int i, int j) {
        int temp = heap.get(i);
        heap.set(i, heap.get(j));
        heap.set(j, temp);
    }

    public boolean isEmpty() {
        return heap.isEmpty();
    }
```

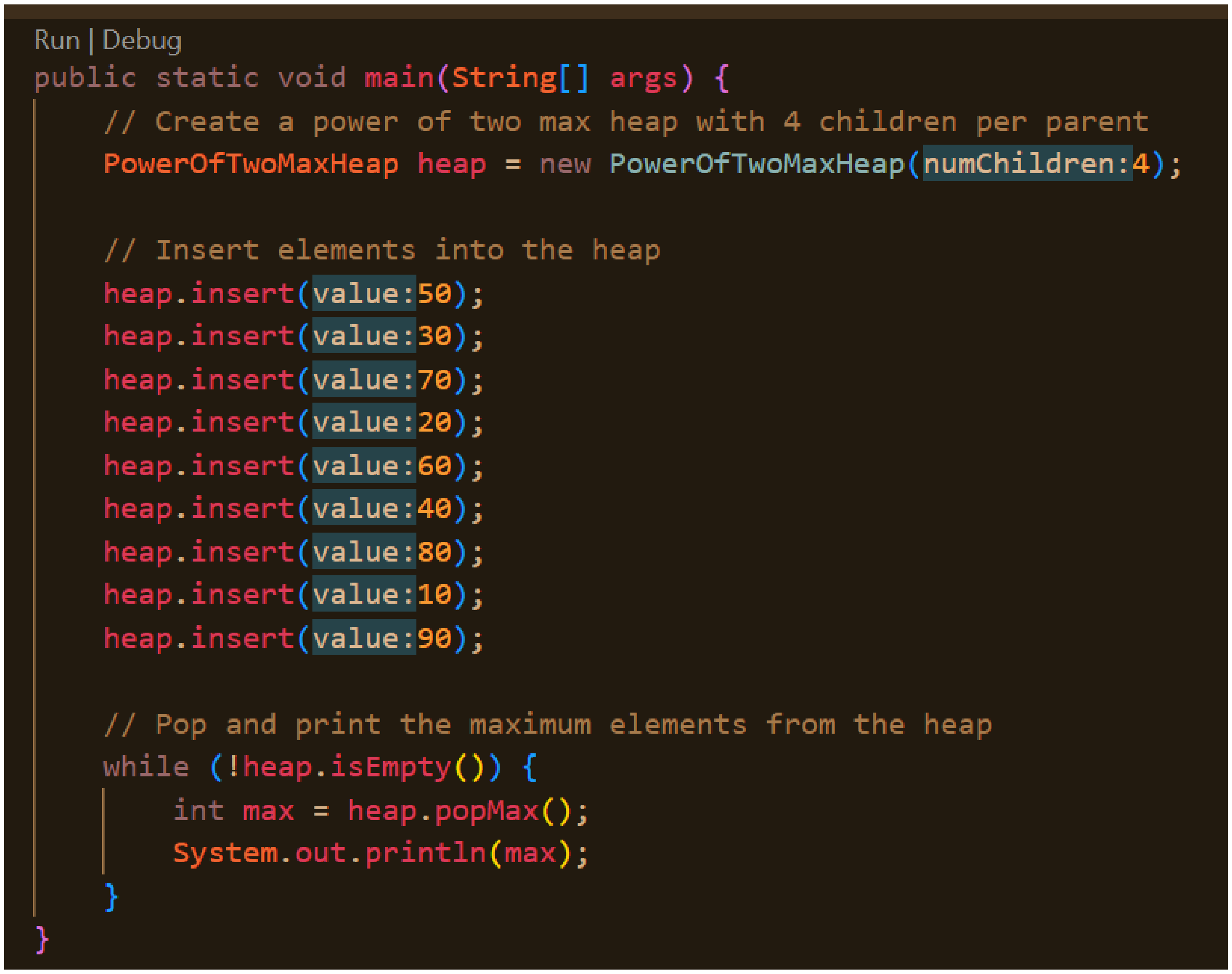
These are two simple methods for swap and isEmtpy()
Int temp stores the value at index ‘i’

‘heap.set(i,heap.get(j));’ this line sets the value at index ‘i’ to the value at index ‘j’. It replaces the value at ‘i’ with the value of ‘j’.

‘heap.set(j,temp);’. This line sets the value at index ‘j’ to the temporary variable ‘temp’. It assigns the original value of ‘i’ (stored in ‘temp’) to ‘j’.

After the swap, the elements at indices ‘i’ and ‘j’ have been exchanged in the heap.
The ‘swap’ method is used in both the ‘heapifyUp’ and ‘heapifyDown’ methods to swap elements and maintain the correct order in the heap.

Then our main method comes up to check if our code works properly

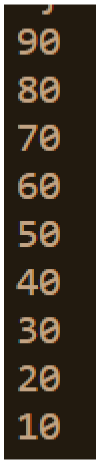


```
Run | Debug
public static void main(String[] args) {
    // Create a power of two max heap with 4 children per parent
    PowerOfTwoMaxHeap heap = new PowerOfTwoMaxHeap(numChildren:4);

    // Insert elements into the heap
    heap.insert(value:50);
    heap.insert(value:30);
    heap.insert(value:70);
    heap.insert(value:20);
    heap.insert(value:60);
    heap.insert(value:40);
    heap.insert(value:80);
    heap.insert(value:10);
    heap.insert(value:90);

    // Pop and print the maximum elements from the heap
    while (!heap.isEmpty()) {
        int max = heap.popMax();
        System.out.println(max);
    }
}
```

Here first declare **numChildren(x) as 4.** And then we insert numbers in random order and then popMax() method called until **heap.isEmpty()** condition becomes **true**. And in the terminal, we got our desired output.



```
90
80
70
60
50
40
30
20
10
```

This is output as expected.

For any more understanding please reach out to me
amitjakhar@iitbhilai.ac.in