Tutorial : Heaps

Exercise 1. Max Binary Heap

You are given an array of integers:

[10, 20, 5, 6, 1, 8, 9, 4]

1. Convert this array into a **Max Binary Heap**.
2. Perform **Insert(15)** into the Max Heap.
3. Perform **DeleteMax()** (remove the maximum element from the heap).

Solution.

Summary of Operations:

1. **Original Array → Max Heap:**
   - [20, 10, 9, 6, 1, 8, 5, 4]
2. **After Insert(15):**
   - [20, 15, 9, 10, 1, 8, 5, 4, 6]
3. **After DeleteMax():**
   - [15, 10, 9, 6, 1, 8, 5, 4]

Exercise 2: Min Binary Heap

You are given an array of integers:

[15, 10, 20, 17, 8, 25, 30]

1. Convert this array into a **Min Binary Heap**.
2. Perform **Insert(5)** into the Min Heap.
3. Perform **DeleteMin()** (remove the minimum element from the heap).

Solution.

Summary of Operations:

1. **Original Array → Min Heap:**
   - [8, 10, 20, 17, 15, 25, 30]
2. **After Insert(5):**
   - [5, 8, 20, 10, 15, 25, 30, 17]
3. **After DeleteMin():**
   - [8, 10, 20, 17, 15, 25, 30]

Exercise 3: Heap Sort Algorithm

You are given an array of integers:

[16, 14, 5, 8, 10, 20, 7, 12]

1. Sort the array using the **Heap Sort Algorithm**.
2. Show the step-by-step process of building the heap and performing the sorting.

Solution

Summary of Operations:

1. **Original Array → Max Heap:**
   - [20, 14, 16, 12, 10, 5, 7, 8]
2. **Heap Sort (Step-by-Step):**
   - [16, 14, 5, 8, 10, 20, 7, 12]
   - [5, 7, 8, 10, 12, 14, 16, 20]

Exercise 4: Max Priority Queue

You are given the following series of operations to perform on a **Max Priority Queue** implemented using a **Max Heap**:

1. **Insert(10)**
2. **Insert(20)**
3. **Insert(5)**
4. **Insert(7)**
5. **Insert(25)**
6. **ExtractMax()**
7. **Insert(30)**
8. **ExtractMax()**

Perform the operations step-by-step and maintain the heap structure after each operation.

Solution:

The **Max Priority Queue** uses a **Max Heap** to maintain the maximum element at the root. The key operations are:

1. **Insert(x):** Insert an element into the heap and "bubble it up" to maintain the heap property.
2. **ExtractMax():** Remove the maximum element (the root) from the heap, replace it with the last element, and "bubble it down" to maintain the heap property.

Summary of Operations:

1. **Insert(10):** [10]
2. **Insert(20):** [20, 10]
3. **Insert(5):** [20, 10, 5]
4. **Insert(7):** [20, 10, 5, 7]
5. **Insert(25):** [25, 20, 5, 7, 10]
6. **ExtractMax():** [20, 10, 5, 7]

7. **Insert(30):** [30, 20, 5, 7, 10]
8. **ExtractMax():** [20, 10, 5, 7]

Exercise 5: Heap Sort Algorithm

Write a function to sort the following array using the **Heap Sort** algorithm:

arr = [25, 12, 11, 16, 5, 30, 20, 10]

- Implement the heap sort algorithm step-by-step.
- Show how to build the max heap and perform sorting.
- Analyze the time complexity O(nlog n)O(n \log n)O(nlogn) using the recurrence relation.

Solution.

```
static void heapify(int arr[], int n, int i) {

    // Initialize largest as root
    int largest = i;

    // left index = 2*i + 1
    int l = 2 * i + 1;

    // right index = 2*i + 2
    int r = 2 * i + 2;

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest]) {
        largest = l;
    }

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest]) {
        largest = r;
    }

    // If largest is not root
    if (largest != i) {
        int temp = arr[i];
        arr[i] = arr[largest];
        arr[largest] = temp;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
```

```java
static void heapSort(int arr[]) {
    int n = arr.length;

    // Build heap (rearrange array)
    for (int i = n / 2 - 1; i >= 0; i--) {
        heapify(arr, n, i);
    }

    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {

        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;

        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}
```

Final Sorted Array:

$$[5, 10, 11, 12, 16, 20, 25, 30]$$

Time Complexity Analysis:

Time Complexity Analysis:

The **Heap Sort** algorithm has two main steps:

1. **Building the Max Heap:**
   - Heapifying takes $O(\log n)$ time for each element. However, building the heap for all elements takes $O(n)$ time in total. This is because only half of the nodes are non-leaf nodes, and the heapifying operation is less costly as we move up the tree. Hence, building the max heap is $O(n)$.
2. **Sorting the Array:**
   - We perform n extractions, and each extraction requires $O(\log n)$ heapify operations. Therefore, the sorting phase is $O(n \log n)$.

Overall time complexity: $O(n + n \log n) = O(n \log n)$

Exercise 6. Ternary Heaps

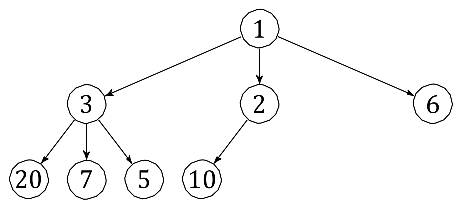
Consider the following sequence of numbers:

$$5, 20, 10, 6, 7, 3, 1, 2$$

(a) Insert these numbers into a min-heap where each node has up to *three* children, instead of two. (So, instead of inserting into a binary heap, we're inserting into a ternary heap.)

Draw out the tree representation of your completed ternary heap.

Solution:



(b) Draw out the array representation of the above tree. In your array representation, you should start at index 0 (not index 1).

Solution:

1, 3, 2, 6, 20, 7, 5, 10

(c) Given a node at index i, write a formula to find the index of the parent.

Solution:

$$\text{parent}(i) = \lfloor (i-1)/3 \rfloor$$

(d) Given a node at index i, write a formula to find the j-th child. Assume that $0 \leq j < 3$.

Solution:

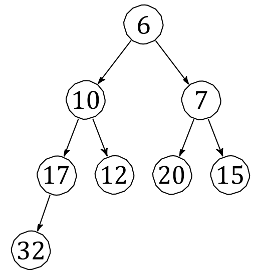
$$\text{child}(i, j) = 3i + j + 1$$

Exercise 7. Heaps – More Basics

a. Insert the following sequence of numbers into a binary *min heap*:

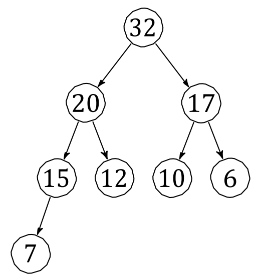
[10, 7, 15, 17, 12, 20, 6, 32]

Solution:



b. Now, insert the same values into a binary *max heap*.

Solution:



Solution:

[7, 8, 10, 9, 15, 13, 12]

Exercise 8. Food For Thought: More Heaps

3.1. Running Times

Let's think about the best and worst case for inserting into heaps.

You have elements of priority $1, 2, \ldots, n$. You're going to insert the elements into a min heap one at a time (by calling insertnot buildHeap) in an order that you can control.

(a) Give an insertion order where the total running time of all insertions is $\Theta(n)$. Briefly justify why the total time is $\Theta(n)$.

Solution:

Insert in increasing order (i.e. $1, 2, 3, \ldots, n$). For each insertion, it is the new largest element in the heap, so `percolateUp` only needs to do one comparison and no swaps. Since we only need to do those (constant) operations at each `insert`, we do $n \cdot \Theta(1) = \Theta(n)$ operations.

(b) Give an insertion order where the total running time of all insertions is $\Theta(n \log n)$.

Solution:

Insert in decreasing order. First let's show that this order requires at most $O(n \log n)$ operations – we have n insertions, each takes at most $O(\text{height})$ operations. The heap is always height at most $O(\log n)$, so the total is $O(n \log n)$.

Now let's show the number of operations is at least $\Omega(n \log n)$. For each insertion, the new element is the new smalliest thing in the heap, so `percolateUp` needs to swap it to the top. For the last $n/2$ elements, the heap is height $\Omega(\log n/2) = \Omega(\log n)$, so there are $\Omega(\log n)$ operations for each of the last $n/2$ insertions. That causes $\Omega(n \log n)$ operations.

Since the number of operations is both $O(n \log n)$ and $\Omega(n \log n)$ is $\Theta(n \log n)$ by definition.

Remark: it's tempting to say something like "there are n `insert`s and they each have $\Theta(logn)$ operations, but that's not true. The number of operations for the first few inserts is a constant, since the tree isn't that tall yet.

3.2. Sorting and Reversing (with Heaps)

(a) Suppose you have an array representation of a heap. Must the array be sorted?

Solution:

No, $[1, 2, 5, 4, 3]$ is a valid min-heap, but it isn't sorted.

(b) Suppose you have a sorted array (in increasing order). Must it be the array representation of a valid min-heap?

Solution:

Yes! Every node appears in the array before its children, so the heap property is satisfied.

(c) You have an array representation of a min-heap. If you reverse the array, does it become an array representation of a max-heap?

Solution:

No. For example, $[1, 2, 4, 3]$ is a valid min-heap, but if reversed it won't be a valid max-heap, because the maximum element won't appear first.

(d) Describe the most efficient algorithm you can think of to convert the array representation of a min-heap into a max-heap. What is its running time?

Solution:

You already know an algorithm – just use `buildHeap` (with percolate modified to work for a max-heap instead of a min-heap). The running time is $\mathcal{O}(n)$.

Exercise 9. HW5 Prep: Delete

You just finished implementing your heap of ints when your boss tells you to add a new method called delete.

```
public class DankHeap
        {
    // NOTE: Data starts at index 0!
    private int[] heapArray;
    private int heapSize;

    // Other heap methods here....

    /**
     * Removes the element k from the heap, and restores the heap
     * property. If no element equal to k exists in the heap, does
     * nothing.
     * @param int k, the element to remove.
     */
    public void delete(int k) {
        // TODO!
    }

    /**
     * You can assume this method correctly percolates the element at
     * the given index of heapArray up/down (if needed) until the heap
     * property is satisfied. You do *not* need to implement this!
     * @param int index, index of the heap member to percolate.
     */
    private void percolate(int index) {
        ...
    }
}
```

(a) How efficient do you think you can make this method? **Solution:**

> The best you can do in the worst case is $O(n)$ time. If you start at the top (unlike a binary search tree) the node of priority k could be in either subtree, so you might have to check both. In the worst case, this leads to checking every node.

(b) Write code for delete. Remember that heapArraystarts at index 0! **Hint:** You can use the percolatemethod defined in the DankHeapclass above. **Solution:**

```java
private void delete(int k) {
    // We need to loop through to find the element to delete.
    for (int i = 0; i < heapSize; i++)
    {
        int curElem = heapArray[i];
        // Bingo.
        if (curElem == k)
        {
            // We'll need to replace the element.
            // Note that if we're deleting the last valid
            // element of heapArray, this does nothing.
            heapArray[i] = heapArray[heapSize - 1];
            heapSize = heapSize - 1;

            // Only enters if we actually replaced something.
            if (i < heapSize)
            {
                // Replacement potentially broke our heap property, so we
                // use percolate to fix it!
                percolate(i);
            }

            // We're done, don't waste any more time.
            break;
        }
    }
}
```