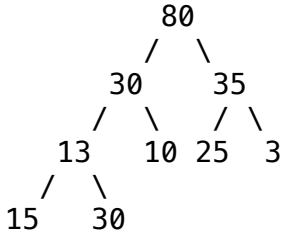
1.1 Draw a heap

Given array => 80, 30, 35, 13, 10, 25, 3, 15

Given Tree =>



1.2 Heapify

```
Given Array = [5, 10, 3, 2, 5, 80, 15, 72]
Array after swap: 5 10 3 72 5 80 15 2
Array after maxHeapify: 5 10 3 72 5 80 15 2
Array after swap: 5 10 80 72 5 3 15 2
Array after maxHeapify: 5 10 80 72 5 3 15 2
Array after swap: 5 72 80 10 5 3 15 2
Array after maxHeapify: 5 72 80 10 5 3 15 2
Array after swap: 80 72 5 10 5 3 15 2
Array after swap: 80 72 15 10 5 3 5 2
Array after maxHeapify: 80 72 15 10 5 3 5 2
Final Array: 80 72 15 10 5 3 5 2
```

1.3 The why of Heaps

One situation where a heap would be preferable to a red-black tree is when we need to maintain a priority queue, and the priority of elements changes frequently. In a priority queue, we want to quickly access the element with the highest or lowest priority, and heaps provide efficient O(1) access to the highest or lowest element. Heap operations such as insertion and deletion are also faster than red-black tree operations since they only involve swapping elements at specific indices and then restoring the heap property, which has a time complexity of O(log n) for a heap. Red-black trees, on the other hand, require balancing operations to maintain the balanced tree property after insertion or deletion, which can take longer than heap operations and have a time complexity of O(log n).

One situation where a red-black tree would be preferable to a heap is when we need to perform frequent search operations on the data structure. Red-black trees provide efficient search operations since they are balanced and guarantee a maximum height of O(log n), resulting in a time complexity of O(log n) for search operations. In contrast, heaps do not guarantee any specific ordering of elements beyond the heap property, making search operations inefficient, with a time complexity of O(n) in the worst case. Therefore, if we need to

perform many search operations, a red-black tree would be a better choice than a heap. Additionally, red-black trees support more complex operations such as range queries and iteration in sorted order, which can be useful in many applications.

2 Sorting

2.1 MergeSort

```
Original array: 4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55

Dividing array from index 0 to 15:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 0 to 7:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 0 to 3:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 0 to 1:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 0 to 1:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 2 to 3:
4 22 98 1 18 15 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 2 to 3:
4 22 1 98 18 15 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 0 to 3:
1 4 22 98 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 4 to 7:
1 4 22 98 18 15 3 91 72 5 9 34 2 17 46 55
Dividing array from index 4 to 5:
1 4 22 98 18 15 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 4 to 5:
1 4 22 98 15 18 3 91 72 5 9 34 2 17 46 55
Dividing array from index 6 to 7:
1 4 22 98 15 18 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 6 to 7:
1 4 22 98 15 18 3 91 72 5 9 34 2 17 46 55
Merging arrays from index 4 to 7:
1 4 22 98 3 15 18 91 72 5 9 34 2 17 46 55
Merging arrays from index 0 to 7:
1 3 4 15 18 22 91 98 72 5 9 34 2 17 46 55
Dividing array from index 8 to 15:
1 3 4 15 18 22 91 98 72 5 9 34 2 17 46 55
Dividing array from index 8 to 11:
1 3 4 15 18 22 91 98 72 5 9 34 2 17 46 55
Dividing array from index 8 to 9:
1 3 4 15 18 22 91 98 72 5 9 34 2 17 46 55
Merging arrays from index 8 to 9:
1 3 4 15 18 22 91 98 5 72 9 34 2 17 46 55
Dividing array from index 10 to 11:
1 3 4 15 18 22 91 98 5 72 9 34 2 17 46 55
```

```
Merging arrays from index 10 to 11:
1 3 4 15 18 22 91 98 5 72 9 34 2 17 46 55
Merging arrays from index 8 to 11:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Dividing array from index 12 to 15:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Dividing array from index 12 to 13:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Merging arrays from index 12 to 13:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Dividing array from index 14 to 15:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Merging arrays from index 14 to 15:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Merging arrays from index 12 to 15:
1 3 4 15 18 22 91 98 5 9 34 72 2 17 46 55
Merging arrays from index 8 to 15:
1 3 4 15 18 22 91 98 2 5 9 17 34 46 55 72
Merging arrays from index 0 to 15:
1 2 3 4 5 9 15 17 18 22 34 46 55 72 91 98

Sorted array: 1 2 3 4 5 9 15 17 18 22 34 46 55 72 91 98
```

2.2 Quicksort
Explain a situation that would result in quicksort taking O(n2) time?

The worst-case scenario for quicksort occurs when the pivot element is always the largest or smallest element in the current sub-array. This can happen, for example, when the input array is already sorted in ascending or descending order. In this case, the algorithm would make n-1 recursive calls (where n is the number of elements in the array) and each call would only reduce the size of the sub-array by one element. This results in a total of n*(n-1)/2 comparisons, which is O(n^2).

2.3 Sorting Discussion
Describe the benefits and drawbacks of the following algorithms (relative to one another):

- Insertion Sort
Benefits: simple implementation, efficient for small arrays, stable (i.e., preserves the relative order of equal elements)

Drawbacks: time complexity is O(n^2), which makes it inefficient for large arrays, not suitable for online sorting (i.e., when elements are added to the array as the sorting is being performed), not as efficient as other sorting algorithms in most cases.

- Mergesort

Benefits: has a time complexity of O(n log n), which makes it efficient for large arrays, is stable, suitable for online sorting, does not depend on the initial order of the input array.

Drawbacks: requires extra memory to store the sorted sub-arrays during the merging process, may not be as efficient as other algorithms when working with small arrays.

- Heapsort
Benefits: has a time complexity of O(n log n), which makes it efficient for large arrays, does not require additional memory to store the sorted array, is often faster than mergesort and quicksort in practice.

Drawbacks: not stable, requires extra space for the heap data structure, may have higher overhead than other algorithms.

- Quicksort
Benefits: has an average time complexity of O(n log n), which makes it efficient for large arrays, is often faster than other algorithms in practice, can be implemented in-place (i.e., does not require additional memory to store the sorted array).

Drawbacks: not stable, worst-case time complexity of O(n^2) in some cases
(e.g., when the input array is already sorted), may have higher overhead than other algorithms.