CMSC 123: Data Structures

1st Semester AY 2019-2020

Prepared by: CCS Templado & KBP Pelaez

Priority Queue ADT

A priority queue is an improved version of a queue wherein the value with highest priority is found at one end of the queue. This is to ensure that whenever a dequeue is performed, the value with the highest priority will be returned.

Priority queues can be seen in fast food chains with only one counter and whenever an elderly or disabled person arrives, they are served first even though they arrived later than the others who are already in queue. In computer science, one application of priority queues is the job schedulers. Some jobs or processes must be executed first before any other job. Priority queues can be implemented using lists or heaps.

Heaps

A heap can be derived from a complete tree. We have discussed binary (search) trees and we will be using it as a blueprint for our binary heap. Heaps have two properties namely the structure property and the heap-order property.

Structure Property

As stated earlier, a heap can be derived from a complete tree. A tree is said to be complete if all levels, except the last one, are filled with nodes. In our case, a binary heap is a complete binary tree.

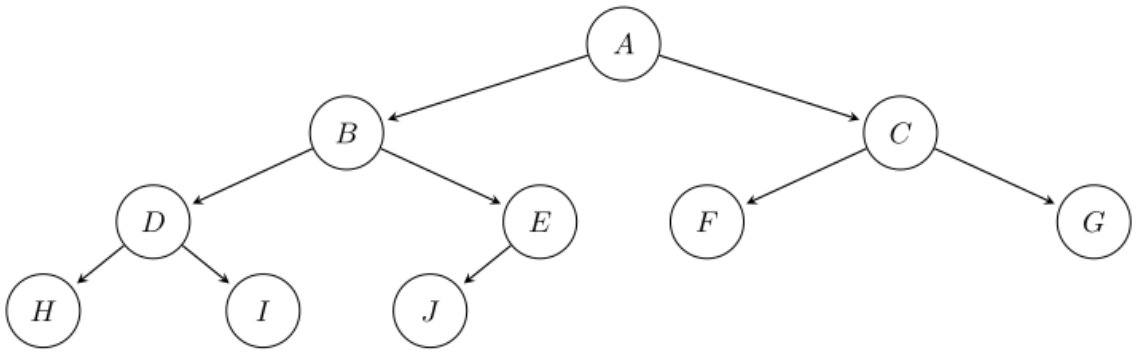

Figure 1: Complete Binary Tree

We have previously implemented BST as a linked list but it can also be implemented using an array. An array implementation is mostly used in heaps since we can easily infer the relationship among the nodes without using pointers. Moreover, this array starts at index 1 in order to make use of the following child-parent representations. Given any element at index `i`, its left child is at index `2i` and its right child is at index `2i + 1`, while its parent is at position `i/2`. The array implementation is only space-efficient if the binary tree is complete; think of a right-leaning tree and how much free space is wasted.

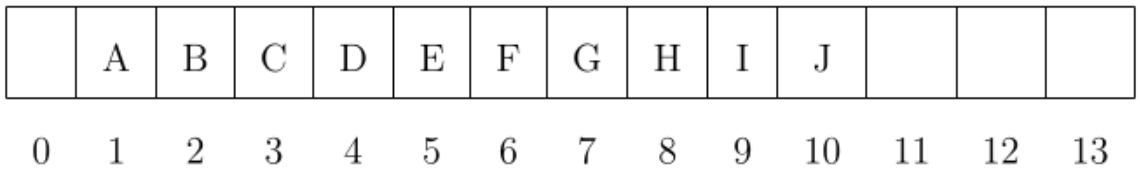

Figure 2: Array Implementation of Heaps

Heap-Order Property

This property determines the arrangement or relationship between children and parents. The heap-order property also determines what type of heap you have. The two (2) basic type of heaps are as follows:

1. **Min heap** - The heap-order property of a min heap states that for every node `x` that is not the root, the key of the `parent of x` should be less than the key of `x`.

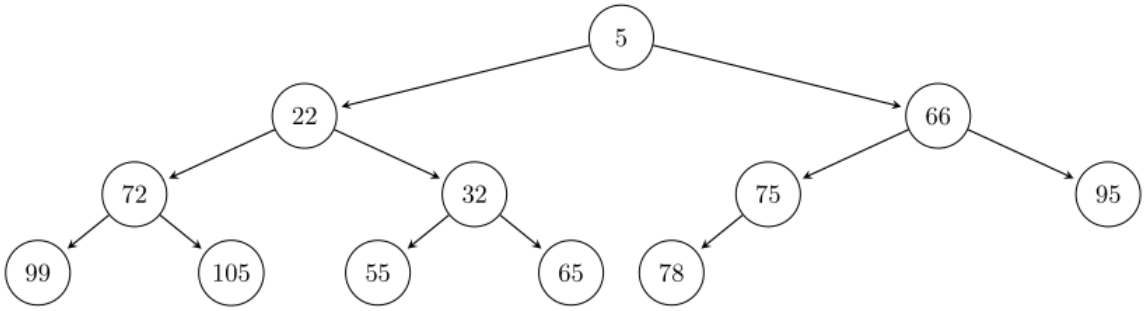

Figure 3: Min heap example

2. **Max heap** - In a max heap the `parent of x` is always larger than `x`.

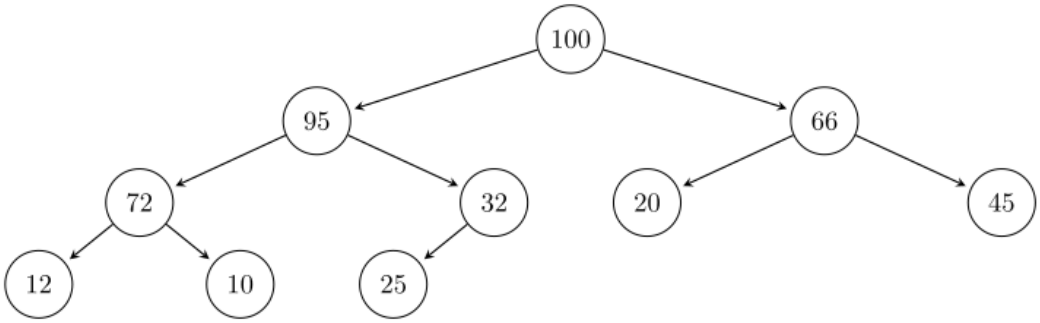

Figure 4: Max heap example

With this, the minimum or maximum element can always be found at the root (in our case at index 1) if the heap is a min heap or max heap, respectively.

Main Operations

Insertion

Inserting a value `v` to the heap involves a strategy called *percolate up* which swaps `v` with the other values in the heap to maintain the heap property. This strategy involves the following steps, considering a min heap:

1. Place `v` in the next available location (which is equivalent to the number of elements in the array +1)
2. If `v` is *less than* the key of the `parent of v`, swap their positions. Repeat this step until `v` is no longer less than its parent or it becomes the new root.

The newly inserted value goes up in the tree, hence calling it percolate up. Take note that the condition in step 2 should be changed if max heap will be implemented.

Consider the example below that inserts the value 15 to the min heap from the previous section.

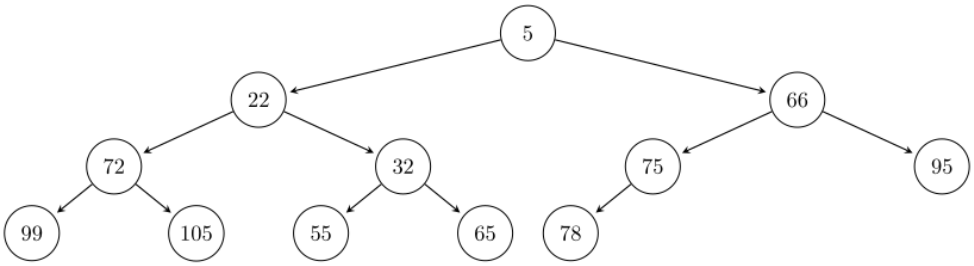

Figure 5: Initial heap

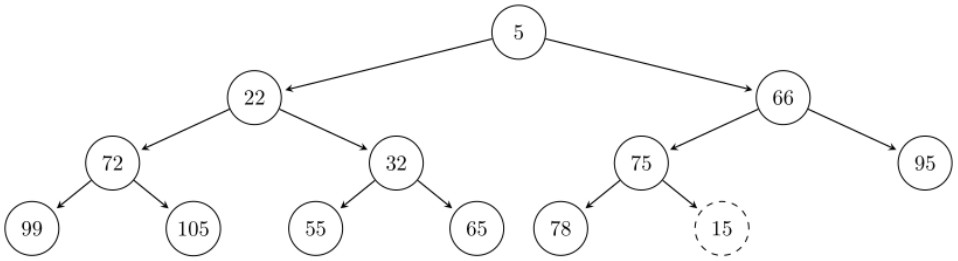

Figure 6: Inserting 15 to the heap

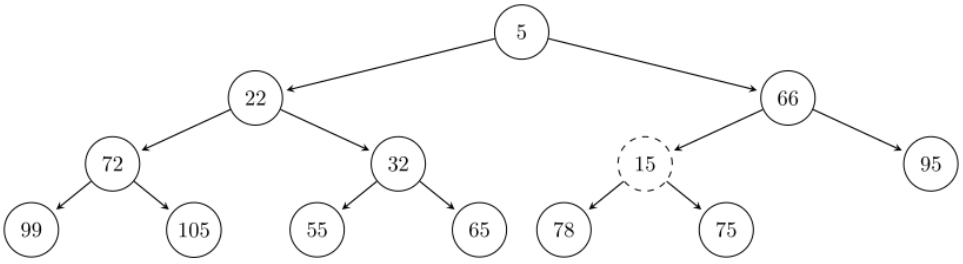

Figure 7: The position of 15 and 75 are swapped

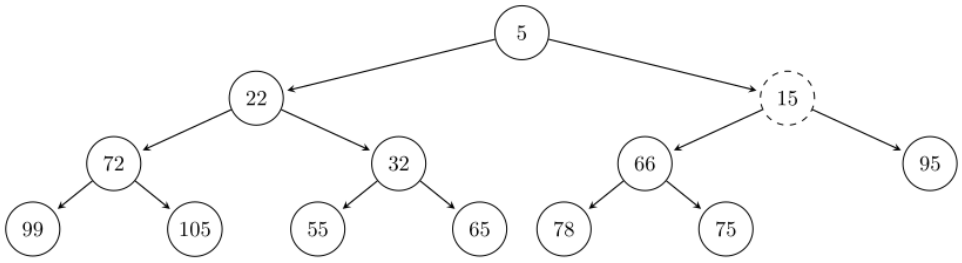

Figure 8: The position of 15 and 66 are swapped

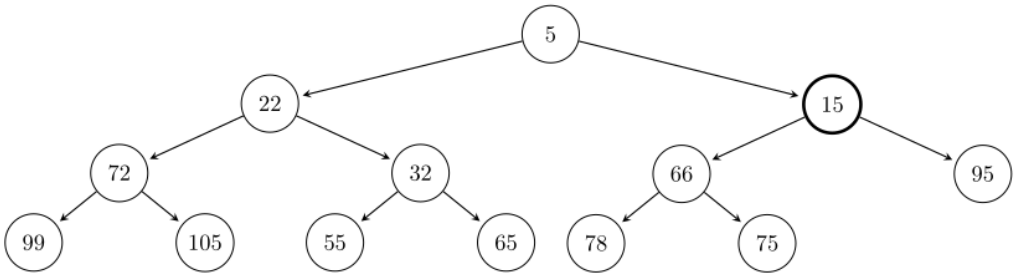

Figure 9: Final snapshot of the heap

Deletion

As previously mentioned, the dequeue in a priority heap always returns the value with the highest priority. In a heap, the value with the highest priority is the root value. Thus, deletion in a heap *always* involves the root value.

To delete the value at the root, a strategy called *percolate down* must be done, to maintain the heap-order property. In a max heap, the following steps must be done:

1. Swap the root value `r` with the last value `v` in the heap.
2. Decrease the size of the heap.
3. If `v` is less than the value of at least one of its children, swap it with the child with the maximum value. Repeat this step until `v` is no longer less than than its children or until `v` becomes a leaf node.

Consider the example shown in Figures 10-16 that deletes the root of the max heap from the previous section.

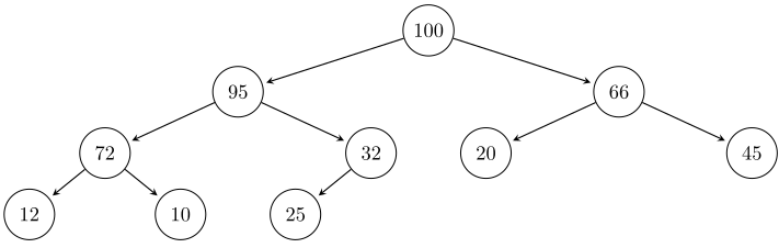

Figure 10: Initial heap

Notice that we are not really deleting (i.e. freeing) the deleted values. We are only putting the deleted value at the end of the current heap, and then decreasing the size of the heap, thus placing the swapped value in a

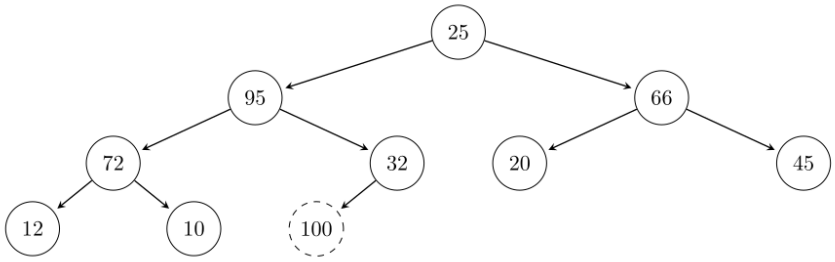

Figure 11: Swapping the root (100) with the last value (25)

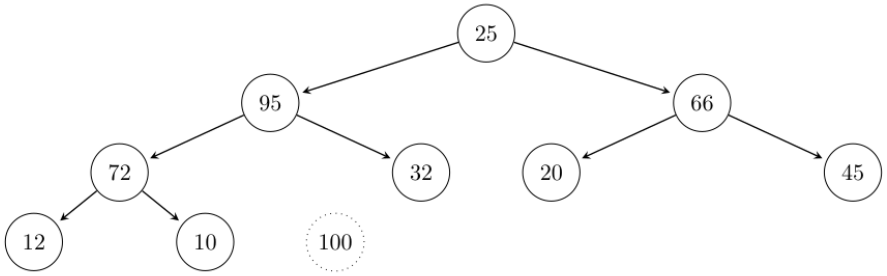

Figure 12: Decreasing the size of the heap will detach the last node from the tree

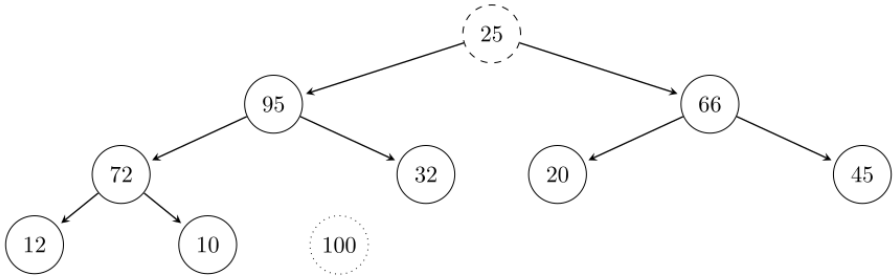

Figure 13: Percolate down 25 by swapping with the largest child that is greater than 25

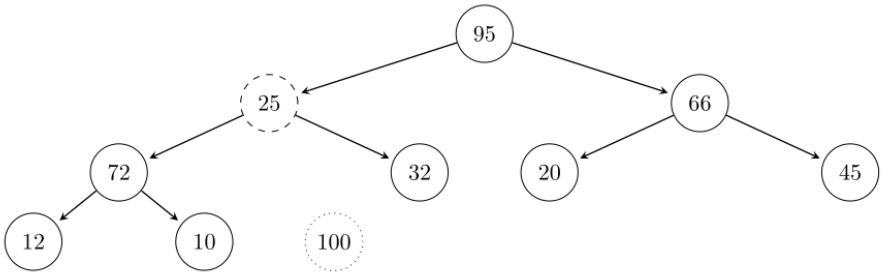

Figure 14: Swapping 25 with 95

location that will not be accessed anymore.

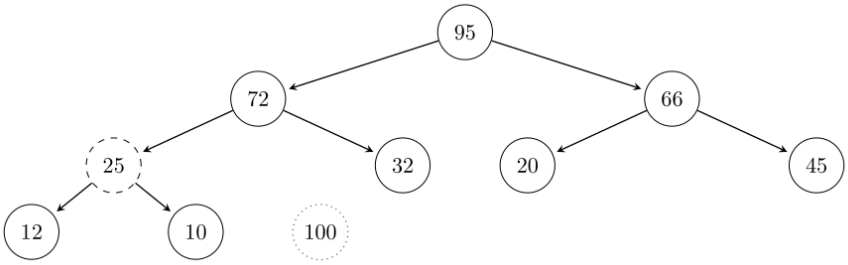

Figure 15: Swapping 25 with 72

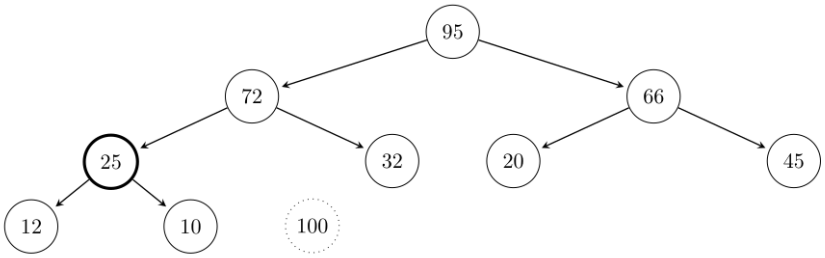

Figure 16: Final snapshot of the heap

Heap Sort

Given a heap, values in a heap can be easily sorted by repeatedly deleting the root.

In a min heap, since the root is always the smallest value, deleting the root will give you the smallest value. Deleting again will give you the second smallest value, then the third, and so on. After deleting all the values in the heap, the array should contain the values of the heap in descending order (last value to be dequeued is the largest value).

Thus, in a max heap, deleting all the values in the heap will result to an array in ascending order.

Other operations

1. `decreaseKey(p, delta)` - decreases the value at position `p` by a positive amount `delta`; the heap-order might be violated when this is performed
2. `increaseKey(p, delta)` - increases the value at position `p` by a positive amount `delta`; the heap-order might be violated when this is performed
3. `remove(p)` - removes the node at position `p`
4. `buildHeap(arr)` - creates a heap with values from the array `arr`

References

Cormen, Thomas H., Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.

Weiss, Mark Allen. *Data Structures and Algorithms*. Benjamin/Cummings.