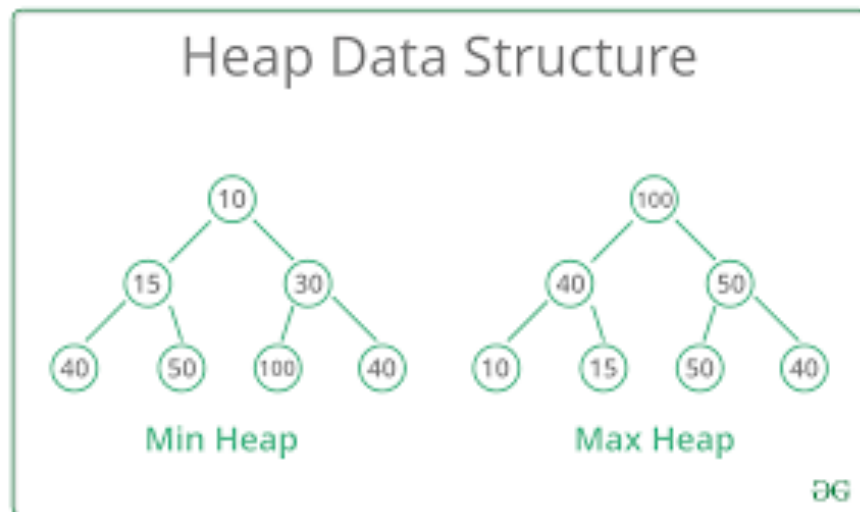


CSCI 3110 Heap, Priority Queue, and Heap Sort

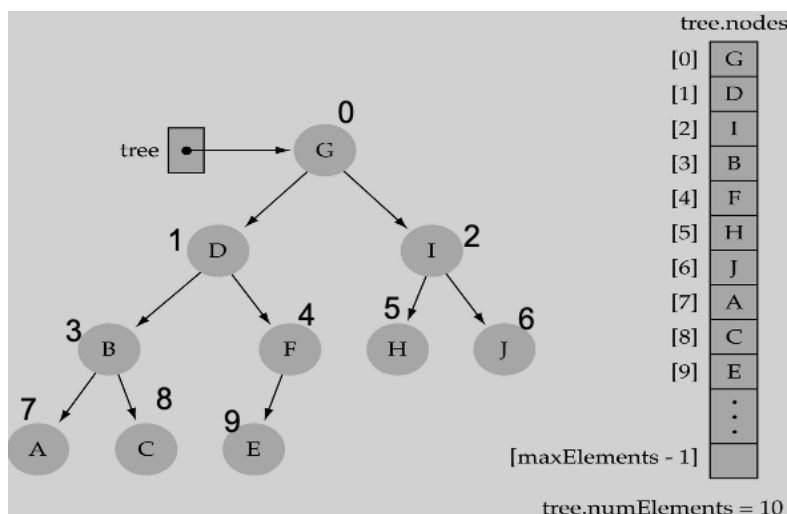
A Binary Heap: A heap is a complete binary tree stored in an array with the following *heap property*: *the parent of a node always has a value greater than or equal to the value at the node*



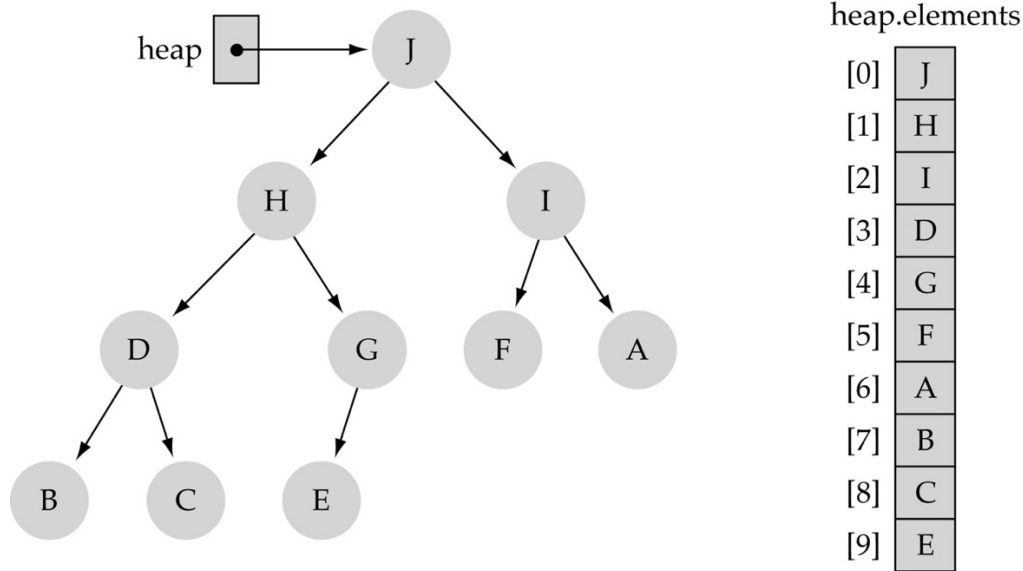
From the heap property, in Max Heap, the largest value of the heap is always stored at the root

The relationships between nodes in the tree are as follows:

- **Parent** of a node i is stored at array location: $\text{floor}((i-1)/2)$
- **Left child** of a node i is stored at array location: $2*i+1$
- **Right child** of node i is stored at array location: $2*i+2$
- Leaf nodes: $[\text{numElements}/2]$ to $[\text{numElements}-1]$



More examples of heap:



Practice exercise:

What is the tree representation of the heap: 20 10 8 5 2 4 ?

Priority Queue

- ☐ A Queue where each element being associated with a “priority”
- ☐ From the elements in the queue, the one with the highest priority is dequeued first
- ☐ Used in many applications, for example most graph search algorithms, AI informed search algorithms
- ☐ **Compare heaps with other priority queue representations**
 - Priority queue using sorted linked list
 - Remove the front queue item, i.e., the largest key → $O(1)$
 - Add a new queue item → $O(n)$
 - Priority queue using heaps
 - Remove the front queue item, i.e., the largest key → $O(\log n)$
 - Add a new queue item → $O(\log n)$
 - Priority queue using binary search tree – *discussed after heap*
 - Remove the front queue item, i.e., the largest key → $O(n)$ worst case
 - Add a new queue item → $O(n)$ worst case
 - Priority queue using AVL tree – *discussed after heap*
 - Remove the front queue item, i.e., the largest key → $O(\log n)$
 - Add a new queue item → $O(\log n)$

Two main heap operations:

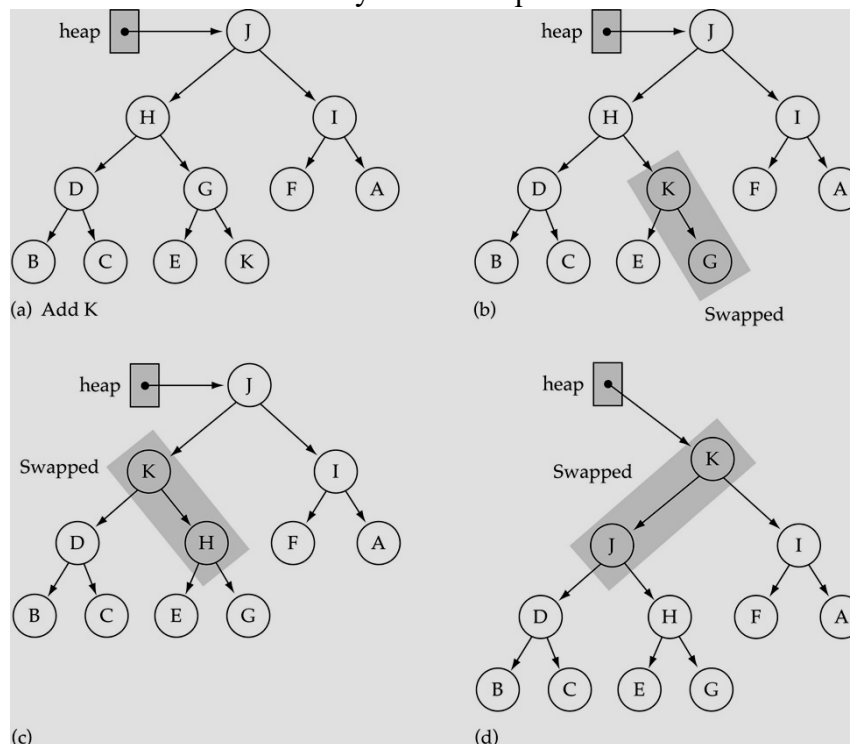
- ☐ Reheap up – add element to heap
- ☐ Reheap down – remove the root data from the heap

Reheaping Up to Add Elements

- When we enqueue an element in the priority queue, our queue/heap has one more value than before.
- We know exactly where the new element must go, i.e., in the next available array slot, so we can add it there, but this might destroy the heap property, e.g. this newly added value may be larger than its parent value.
- We will fix the property by applying the **reheap up** process:
 - We do this by comparing the element with its parent.
 - If the element is less than or equal to its parent then we're done.
 - Otherwise, we swap the element and its parent and repeat the reheaping up (recursive) process with the parent.
 - We continue this way until we have completely restored the heap property.
- This reheap up process may not stop until we reach the root. This means shifting up to $O(\log n)$ elements, therefore the reheap up process takes **$O(\log n)$ time in the worst case.**

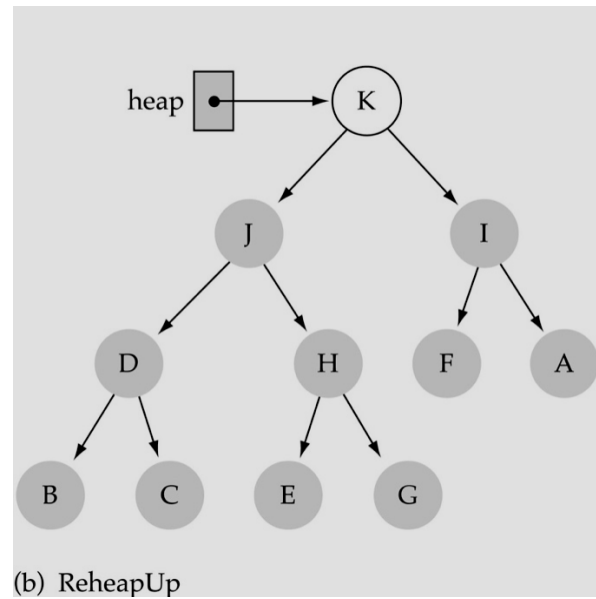
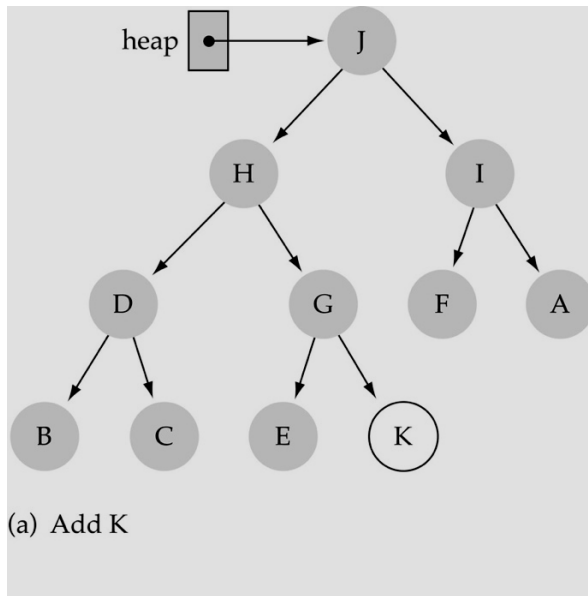
Reheap Up

In this case the heap property is violated at the rightmost node at the last level of the tree. This situation is encountered by insertion operation.



Practice exercise:

Suppose we add a new element 50 to this heap: 20 10 8 5 2 4 6, how many nodes will have a different value from before, after this operation?



Reheap Up code:

```
template<class ItemType>
struct HeapType {
    void ReheapDown(int, int);
    void ReheapUp(int, int);
    ItemType *elements; // allocate the array dynamically using the pointer
    int numElements; // heap elements
};

template<class ItemType>
void HeapType<ItemType>::ReheapUp(int root, int bottom) {
    int parent;

    if (bottom > root) { // heap is not empty
        parent = (bottom-1)/2;
        if (elements[parent] < elements[bottom]) {
            Swap(elements, parent, bottom);
            ReheapUp(root, parent);
        }
    }
}
```

Practice exercise:

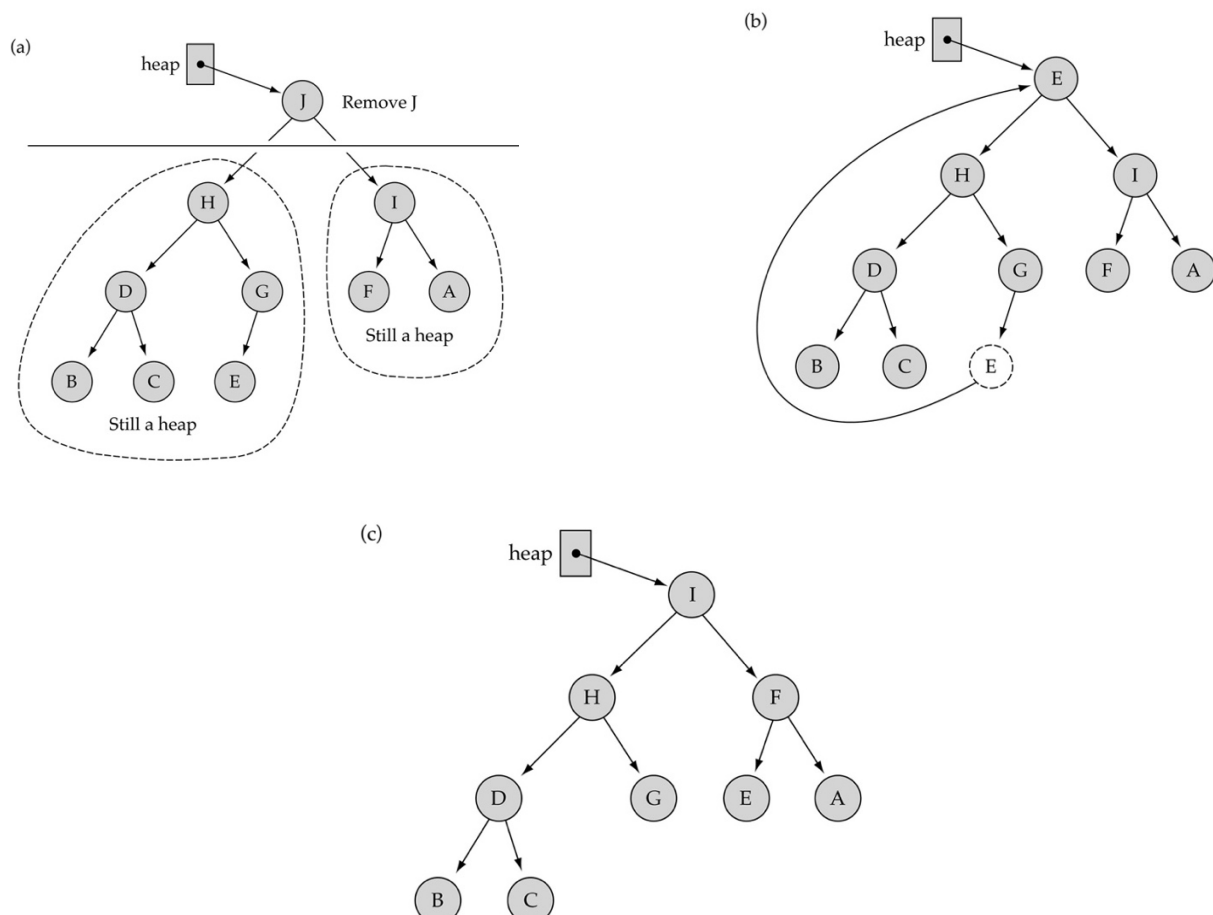
Suppose we add a new element 10 to this heap: 25 15 8 9 2 4, how many nodes will have a different value from before, after this operation?

Reheap Down Process (during *Dequeue*)

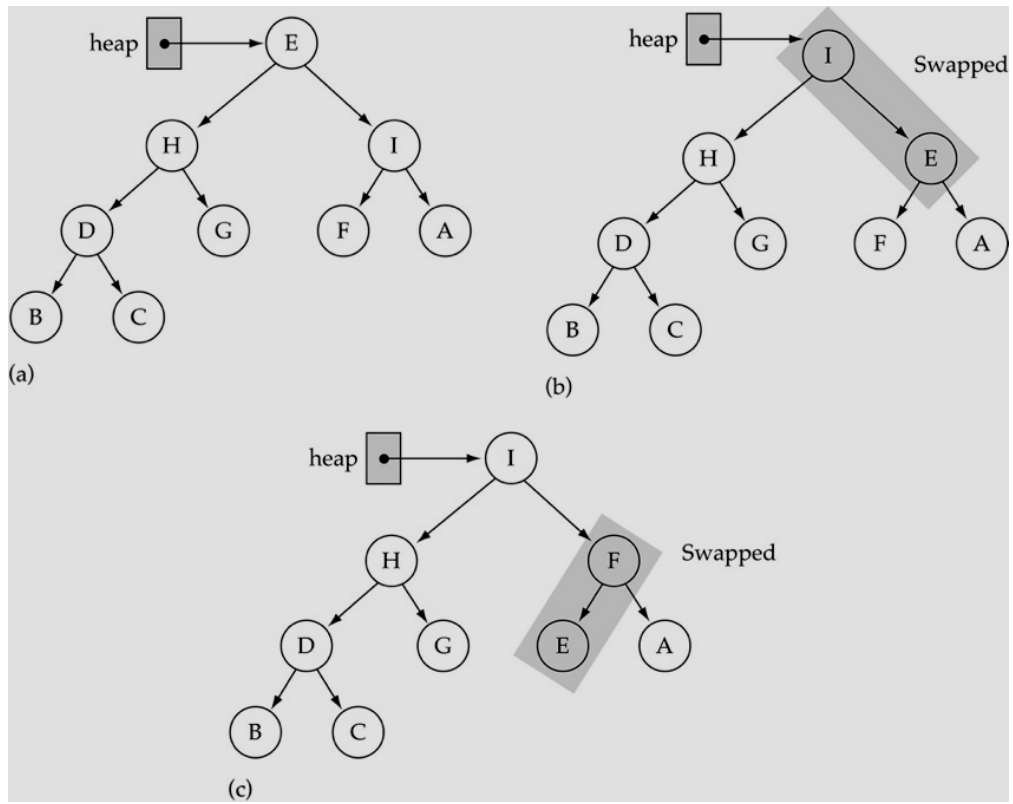
- After returning (and removing) the value in the root of the heap, we have to make adjustments to maintain the heap. We move the element from the former last location into the root.
- This may cause the root value to violate the heap property. We need to perform **reheap down** process to restore the heap. Repeatedly swapping elements between the parent and child nodes to move the element down until the heap property is restored.
- At each reheap down step, swap the parent with the larger of its two child nodes.
- The natural way to do this is recursively.

Illustration of moving the element from the former last location into the root, then recursively apply reheap down:

- Copy the bottom level rightmost data into the root
- Delete the bottom rightmost node
- Restore the heap property by calling **Reheap Down**



- **ReHeap Down process time complexity $O(\log n)$**



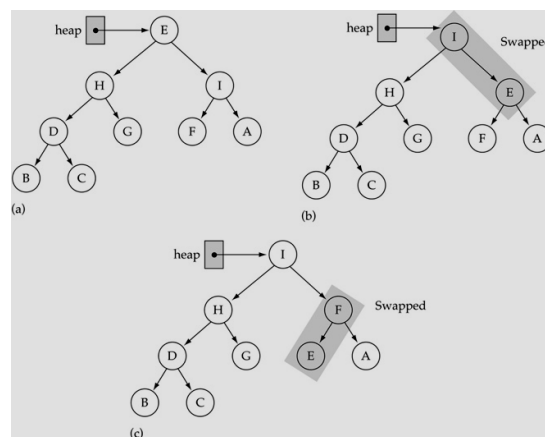
Code for Reheap Down

```
template<class ItemType>
struct HeapType {
    void ReheapDown(int, int);
    void ReheapUp(int, int);
    ItemType *elements; // allocate the array dynamically using the pointer
    int numElements; // heap elements
};

template<class ItemType>
void HeapType<ItemType>::ReheapDown(int root, int bottom) {
    int maxChild, rightChild, leftChild;

    leftChild = 2*root+1;
    rightChild = 2*root+2;

    if(leftChild <= bottom) { // left child is part of the heap
        if (leftChild == bottom) // only one child case
            maxChild = leftChild;
        else {
            if (elements[leftChild] <= elements[rightChild])
                maxChild = rightChild;
            else
                maxChild = leftChild;
        } // end if
        if (elements[root] < elements[maxChild]) {
            Swap(elements, root, maxChild);
            ReheapDown(maxChild, bottom);
        } // end if
    } // end if
}
```



Practice Exercise :

Remove 35 from the heap(35 25 15 8 9 2 4) → Apply Reheap Down process to restore heap property

Heap Sort

(Used in implementing partial_sort in <algorithm>)

Basic steps:

1. Build the heap from an array of data
2. Remove the current root of the heap and add it to the sorted part of the array being built
3. Restore heap property
4. Repeat steps 2-3 until the heap is empty

// Main function to do heap sort

```
void HeapSort(int arr[], int N){
```

```
    // Build max heap
```

```
    for (int i = N / 2 - 1; i >= 0; i--)
        heapify(arr, i, N);
```

```
    // Heap sort
```

```
    for (int i = N - 1; i >= 0; i--) {
        swap(arr[0], arr[i]);
        // Heapify root element to get highest element at root again
        heapify(arr, 0, i);
    }
}
```

// To heapify a subtree rooted with node I, which is an index in arr[], n is size of the heap

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

```
    // Find largest among root, left child and right child
```

```
    int largest = i; // Initialize largest as root
```

```
    int left = 2 * i + 1; // left = 2*i + 1
```

```
    int right = 2 * i + 2; // right = 2*i + 2
```

```
    // If left child is larger than root
```

```
    if (left < N && arr[left] > arr[largest])
        largest = left;
```

```
    // If right child is larger than largest so far
```

```
    if (right < N && arr[right] > arr[largest])
        largest = right;
```

```
    // Swap and continue heapifying if root is not largest
```

```
    if (largest != i) { // If largest is not root
```

```
        swap(arr[i], arr[largest]);
```

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

```
    }
```

```
}
```

What is the time complexity of Heap Sort? $O(n \log n)$

Example :

Sort the following data using Heap Sort. arr: 46, 81, 98, 28, 37, 32. ← N is 6

Function call: HeapSort(array, N)

Step 1: Build the max heap:

i=2, Heapify(arr, 2, 6) → no change

i=1, Heapify(arr, 1, 6) → no change

i=0, Heapify(arr, 0, 6) → swap(a[0], a[2]), arr: 98 81 46 28 37 32

Step 2: Heap Sort

i=5, swap(arr[0], arr[5]), arr: 32 81 46 28 37 || 98. (98 is in its sorted position)

Heapify(arr, 0, 5)

Swap(arr[0], arr[1]), swap(arr[1], arr[4])

arr: 81 37 46 28 32 || 98

i=4, swap(arr[0], arr[4]), arr: 32. 37. 46. 28. || 81 98 (81 and 98 in their sorted position)

Heapify(arr, 0, 4)

Swap(arr[0], arr[2])

arr: 46 37 32 28 || 81 98

i=3, swap(arr[0], arr[3]), arr: 28 37 32 || 46 81 98

Heapify(arr, 0, 3)

Swap(arr[0], arr[1])

arr: 37 28 32 || 46 81 98

i=2, swap(arr[0], arr[2]), arr: 32 28 || 37 46 81 98

_____Heapify(arr, 0, 2)
No change
arr: 32 28 || 37 46 81 98

i=1, swap(arr[0], arr[1]), arr: 28 || 32 37 46 81 98. ➔ complete!

Heap Sort analysis

Building max heap:

Repeat Heapify ($N/2-1$) times

Each Heapify is $O(\log N)$

Total $O(N \log N)$ time

Heap sort loop

Repeat $N-1$ times:

1. Swap out the first array element
2. Heapify

Total $f(N) = (N-1) * (1 + \log N)$

$O(N \log N)$

Therefore, the time complexity of Heap sort is $O(N \log N)$.

Practice Exercise:

Sort the following array of data into ascending order using Heap sort:

40 35 12 4 25 9 15