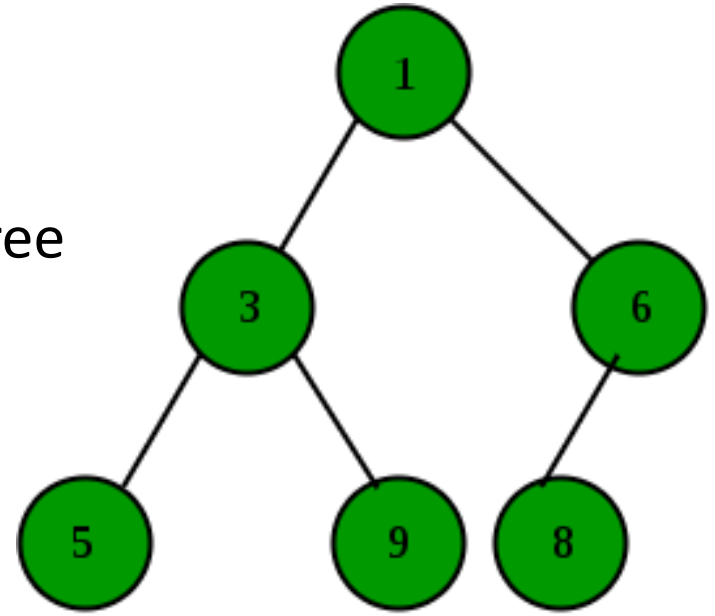
Heap

(Binary) Heap

A heap is a balanced binary tree

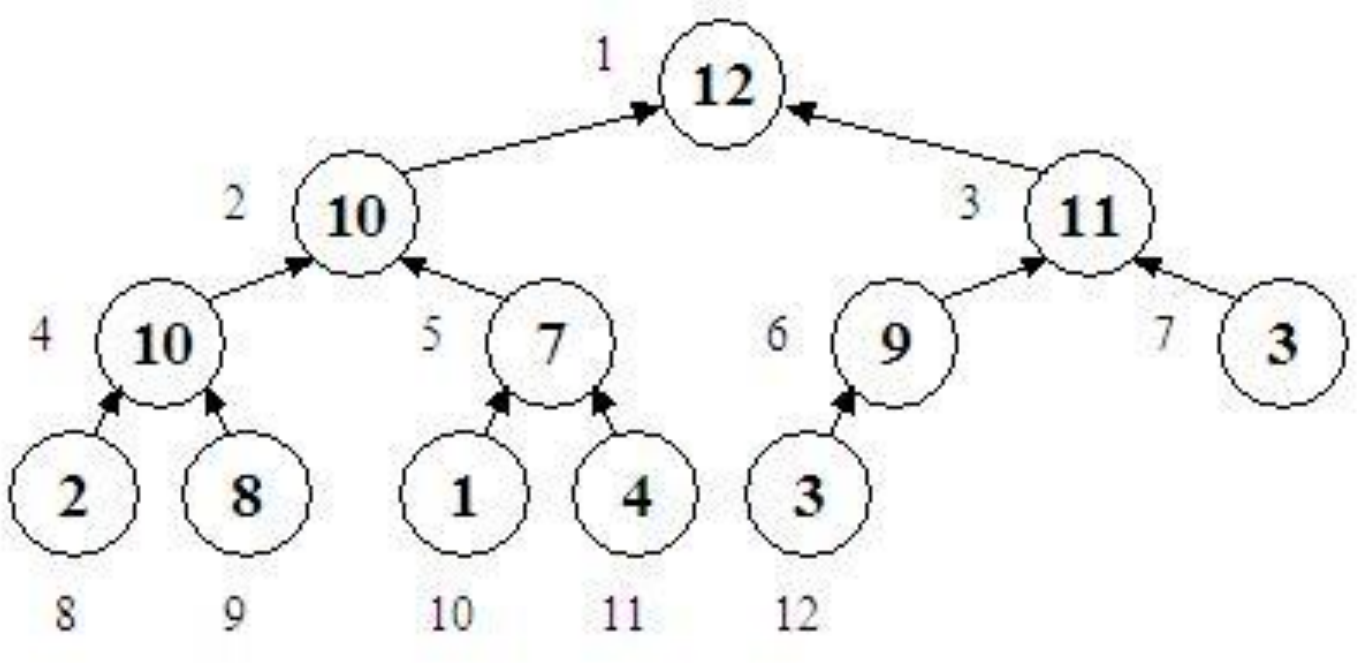


A heap can be represented by an array

- array[(i-1)/2] returns the parent node
- array[(2*i)+1] returns the left child node
- array[(2*i)+2] returns the right child node

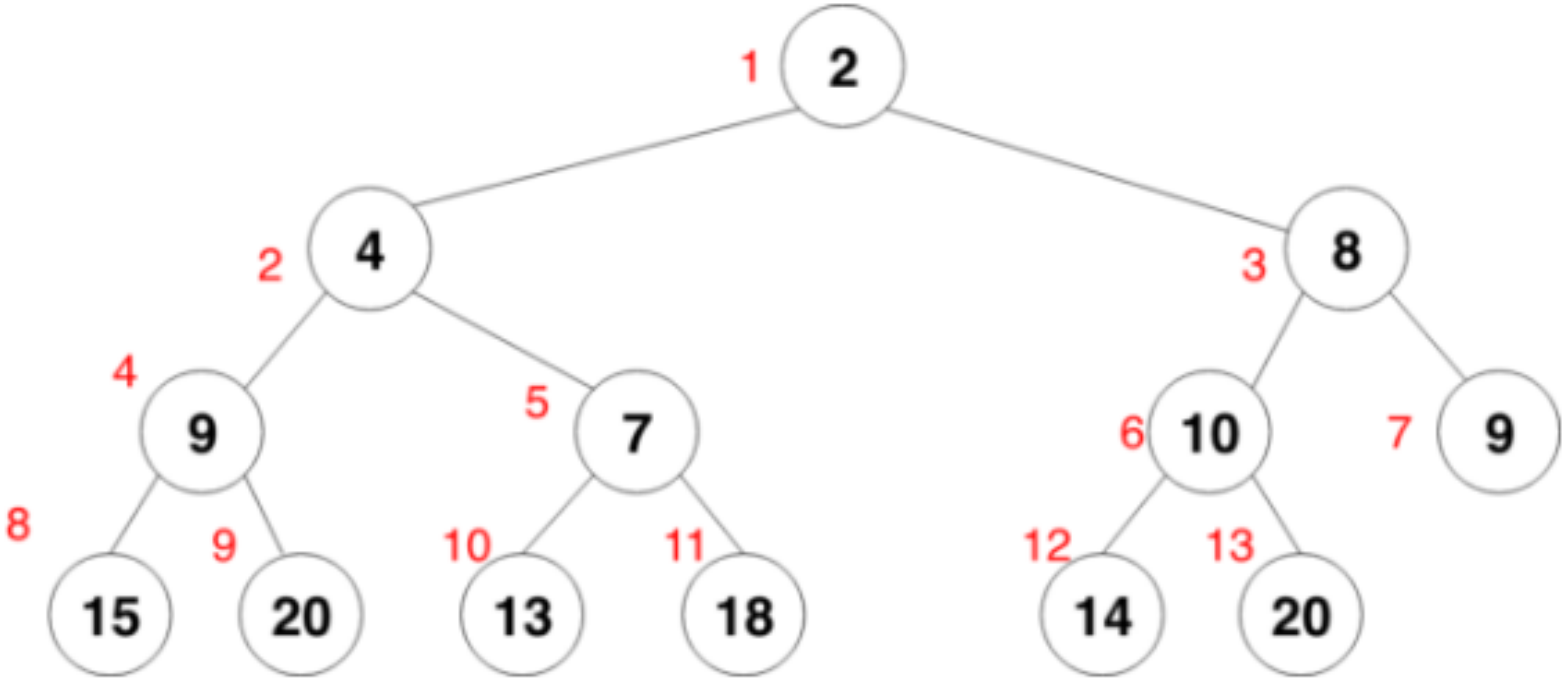
Max-Heap

- valoarea oricarui nod este mai mare sau egala cu valoarea oricarui fiu al sau
- toate nivelurile sunt complete, cu exceptia ultimului, care se completeaza de la stanga spre dreapta



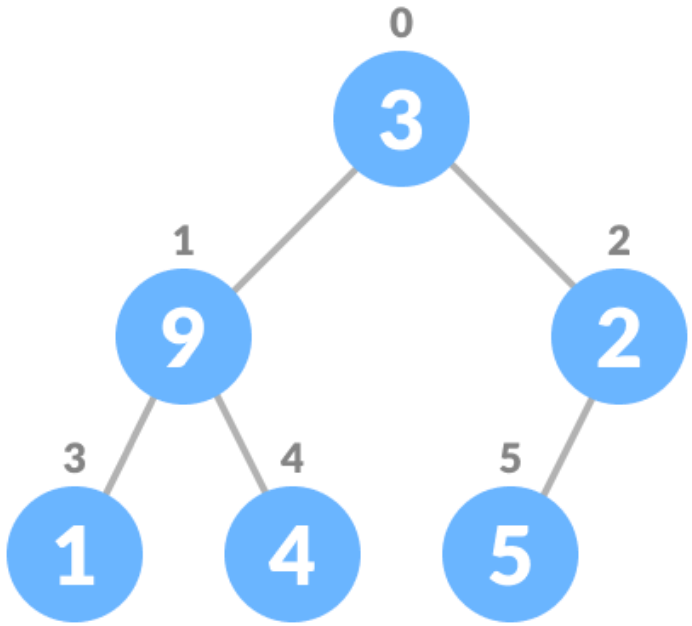
Min-Heap

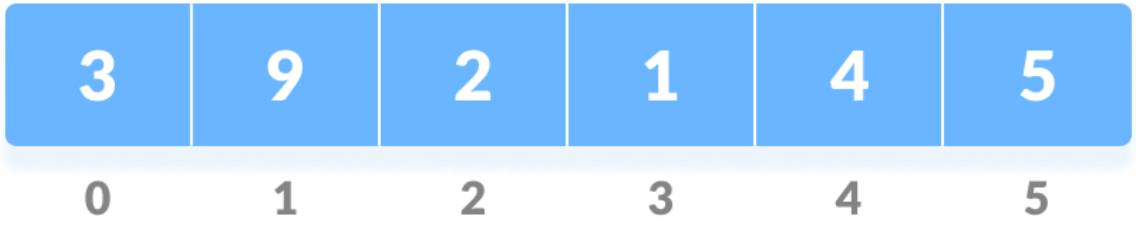
- elementele sunt mai mici decat copii acestora
- radacina heap-ului va fi intotdeauna elementul cel mai mic



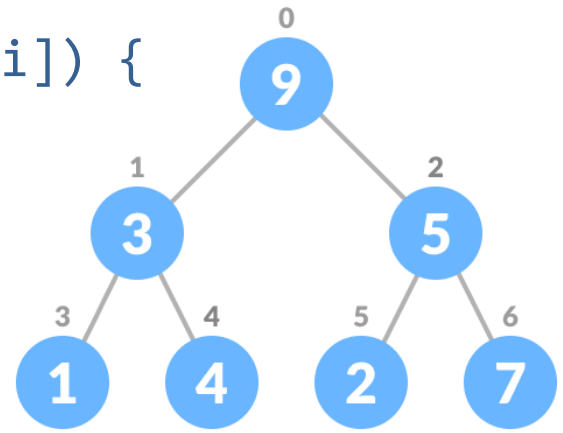
INSERT

```
// returns the index of the parent node
int parent(int i) {
    return (i - 1) / 2;
}

// return the index of the left child
int left_child(int i) {
    return 2*i + 1;
}

// return the index of the right child
int right_child(int i) {
    return 2*i + 2;
}
```



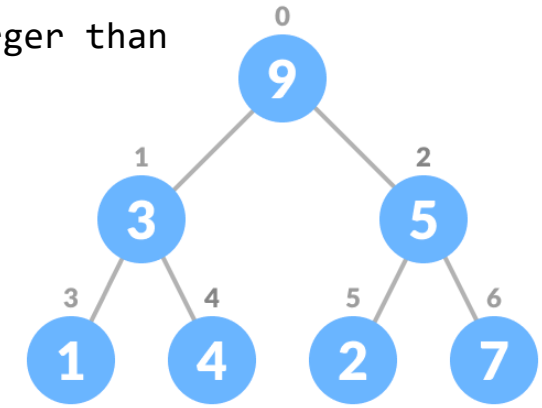


```
// insert the item at the appropriate position
void insert(int a[], int data, int *n) {
    if (*n >= MAX_SIZE) {
        printf("%s\n", "The heap is full. Cannot insert");
        return;
    }
    // first insert the time at the last position of the array
    // and move it up
    a[*n] = data;
    *n = *n + 1;

    // move up until the heap property satisfies
    int i = *n - 1;
    while (i != 0 && a[parent(i)] < a[i]) {
        swap(&a[parent(i)], &a[i]);
        i = parent(i);
    }
}
```



Heapify

```
// moves the item at position i of array a[] into its appropriate position
void max_heapify(int a[], int i, int n){
    // find left child node
    int left = left_child(i);

    // find right child node
    int right = right_child(i);

    // find the largest among 3 nodes
    int largest = i;

    // check if the left node is larger than the current node
    if (left <= n && a[left] > a[largest]) {
        largest = left;
    }

    // check if the right node is larger than the current node
    if (right <= n && a[right] > a[largest]) {
        largest = right;
    }

    // swap the largest node with the current node
    // and repeat this process until the current node is larger than
    // the right and the left node
    if (largest != i) {
        int temp = a[i];
        a[i] = a[largest];
        a[largest] = temp;
        max_heapify(a, largest, n);
    }
}
```



building max heap

```
// converts an array into a heap
void build_max_heap(int a[], int n) {
    int i;
    for (i = n/2; i >= 0; i--) {
        max_heapify(a, i, n);
    }
}
```

extract max

```
// deletes the max item and return
int extract_max(int a[], int *n) {
    int max_item = a[0];

    // replace the first item with the last item
    a[0] = a[*n - 1];
    *n = *n - 1;

    // maintain the heap property by heapifying the
    // first item
    max_heapify(a, 0, *n);
    return max_item;
}
```

```
// build Max Heap (varianta)

// value of child is smaller than value of their parent
void buildMaxHeap(int arr[], int n)
{
    for (int i = 1; i < n; i++)
    {
        // if child is bigger than parent
        if (arr[i] > arr[(i - 1) / 2])
        {
            int j = i;

            // swap child and parent until
            // parent is smaller
            while (arr[j] > arr[(j - 1) / 2])
            {
                swap(arr[j], arr[(j - 1) / 2]);
                j = (j - 1) / 2;
            }
        }
    }
}
```

```
void heapSort(int arr[], int n) {
    buildMaxHeap(arr, n);

    for (int i = n - 1; i > 0; i--) {
        // swap value of first indexed with last indexed
        swap(arr[0], arr[i]);

        // maintaining heap property after each swapping
        int j = 0, index;

        do {
            index = (2 * j + 1);

            // if left child is smaller than
            // right child point index variable
            // to right child
            if (arr[index] < arr[index + 1] && index < (i - 1))
                index++;

            // if parent is smaller than child
            // then swapping parent with child
            // having higher value
            if (arr[j] < arr[index] && index < i)
                swap(arr[j], arr[index]);

            j = index;

        } while (index < i);
    }
}
```