CS201
Data Structures and Algorithms
Revision Session 7

binary heap

Heap

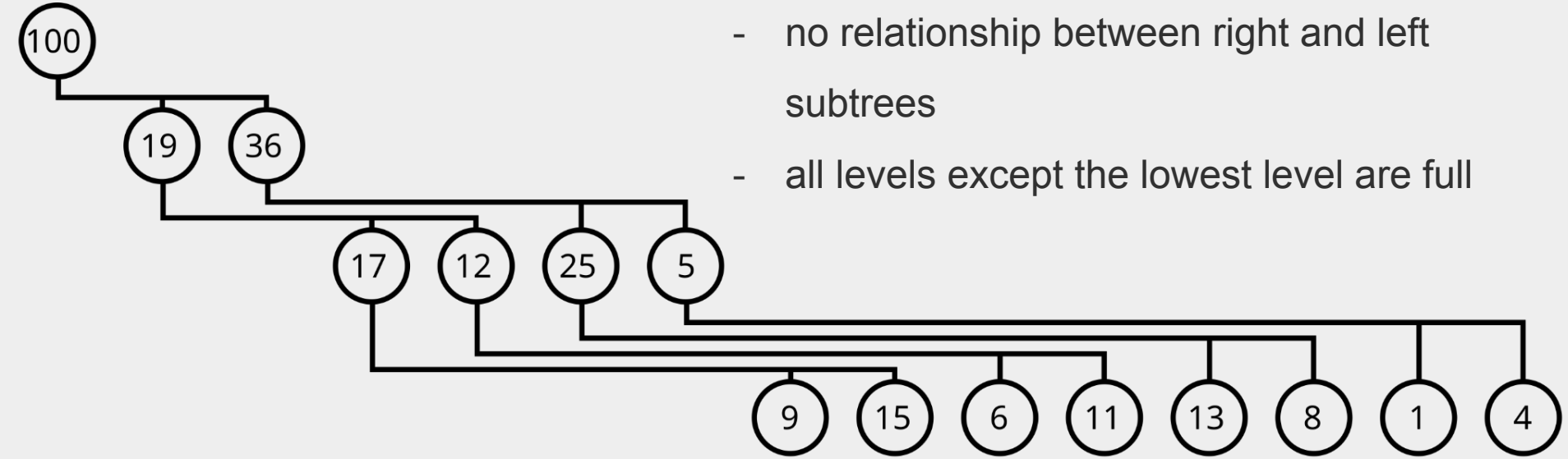
basics:
definition
implementation

binary heap:
deletion
insertion
search
update

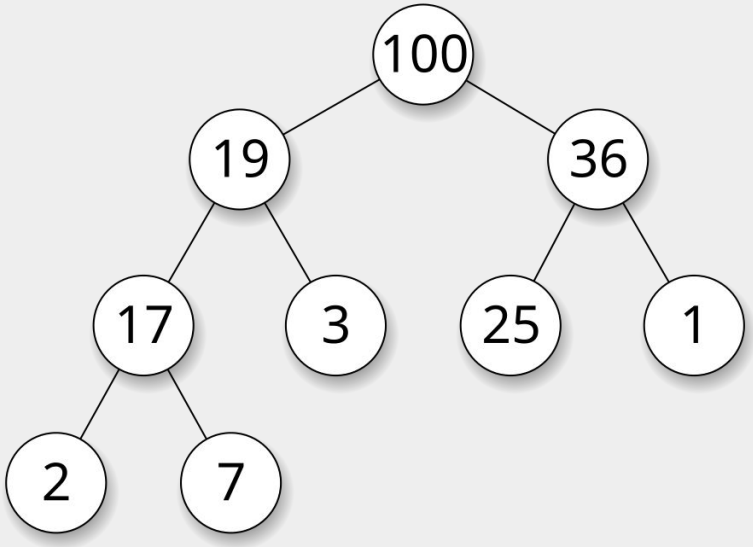
definition

- priority queue
- reaching the elements of maximum (or minimum) importance
- being able to remove that element when no longer needed

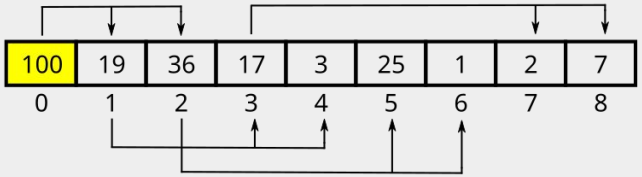
- not fully sorted like binary search trees
- no relationship between right and left subtrees
- all levels except the lowest level are full



Tree representation



Array representation

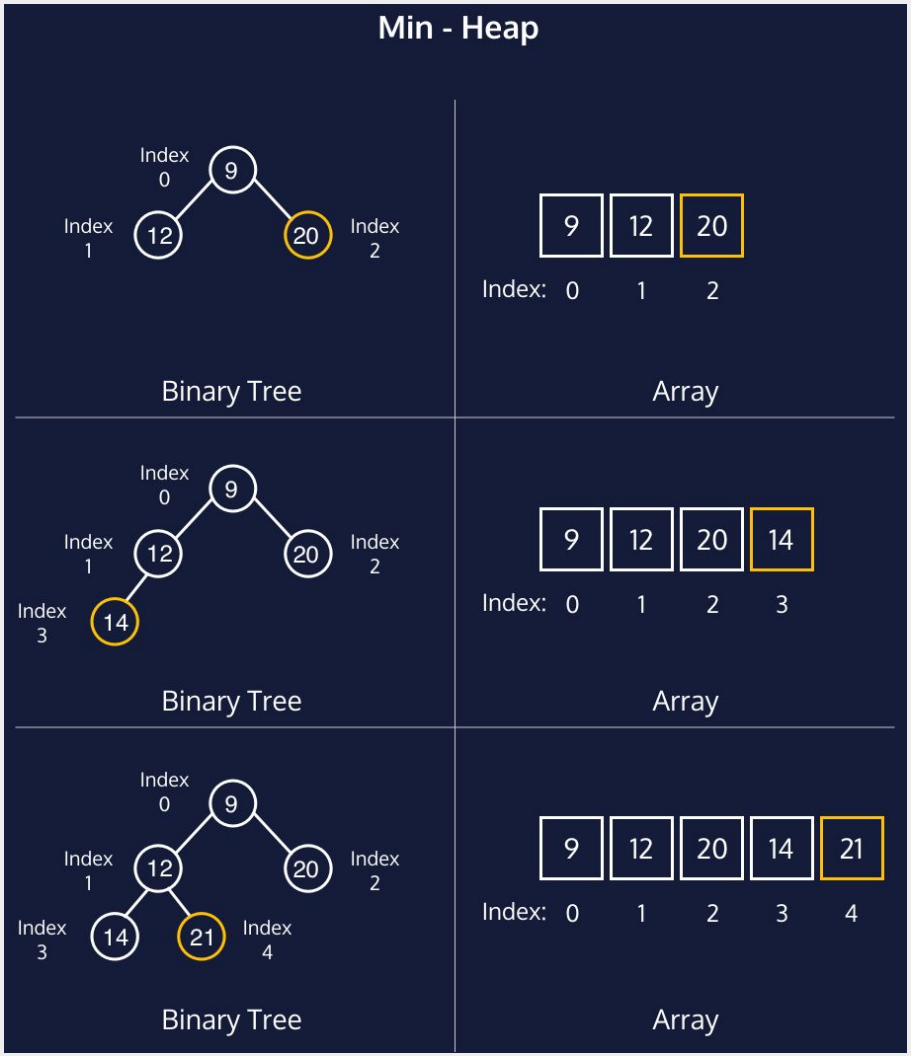


Max-heap

- **root has the highest value**
- **each parent node has higher value than its children**

0th index -> 1 (left) and 2 (right)
1st index -> 3 (left) and 4 (right)
2nd index -> 5 (left) and 6 (right)
3rd index -> 7 (left) and 8 (right)

right child -> 2x + 1
left child -> 2x + 2
parent -> (index-1) / 2



Min-heap

root has the lowest value

0th index -> 1 (left) and 2 (right)
1st index -> 3 (left) and 4 (right)
2nd index -> 5 (left) and 6 (right)
3rd index -> 7 (left) and 8 (right)

right child -> 2x + 1
left child -> 2x + 2
parent -> (index-1) / 2

implementation

```java
public class HeapNode {
    1 usage
    int data;
    1 usage
    int name;

    no usages   new *
    public HeapNode(int data, int name){
        this.data = data;
        this.name = name;
    }
}
```

```java
public class Heap {
    1 usage
    HeapNode[] array;
    2 usages
    int count;

    1 usage   new *
    public Heap(int N){
        array = new HeapNode[N];
        this.count = 0;
    }
```

```java
public boolean isEmpty(){
    return count == 0;
}
```

deletion of the highest priority element

- *deleteMax*
- *swapNode*
- *percolateDown*

1. **copy** the last element to the 0th index in the array

2. **bring the tree back to its correct structure** -> the last element should percolate down until all its children are smaller than its value

3. **decrement** the count by one

```java
public HeapNode deleteMax(){
    HeapNode tmp = array[0];

    array[0] = array[count-1];
    percolateDown( i: 0);

    count--;

    return tmp;
}
```

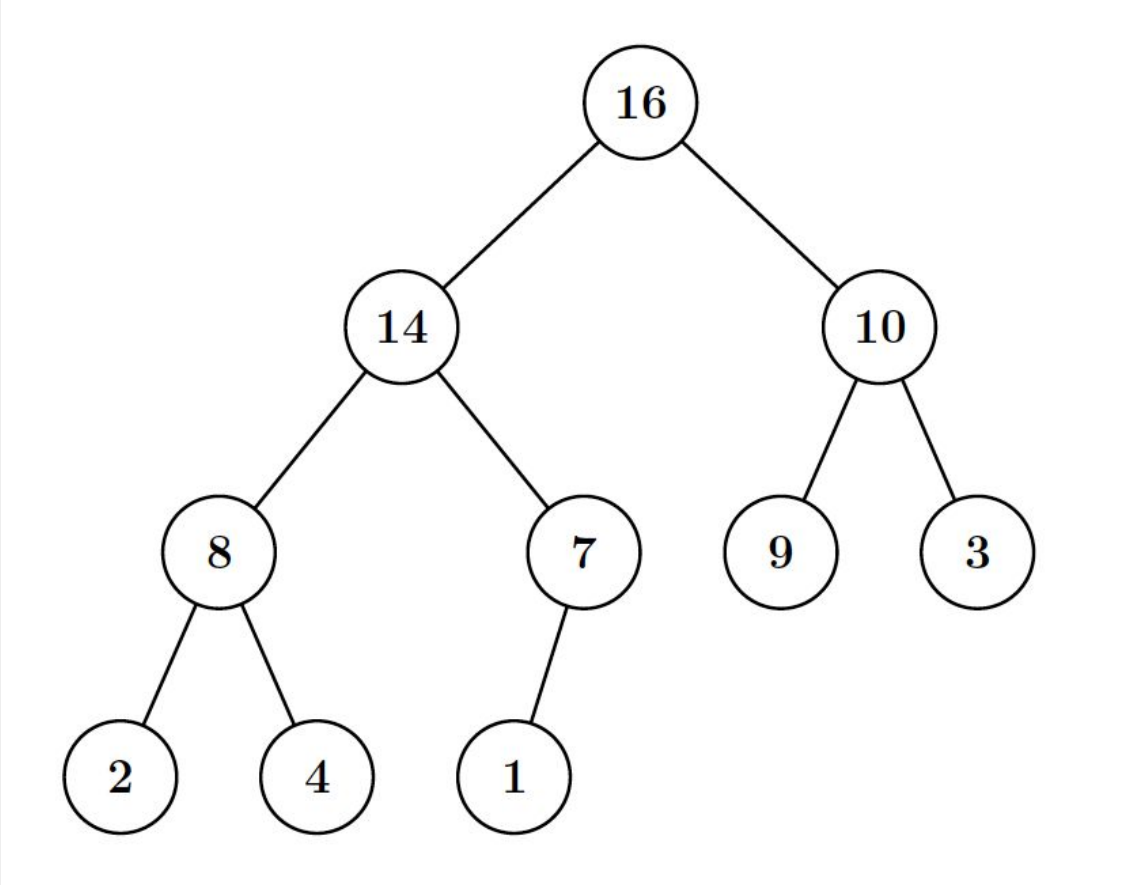
Keep the highest priority element in the tmp variable.

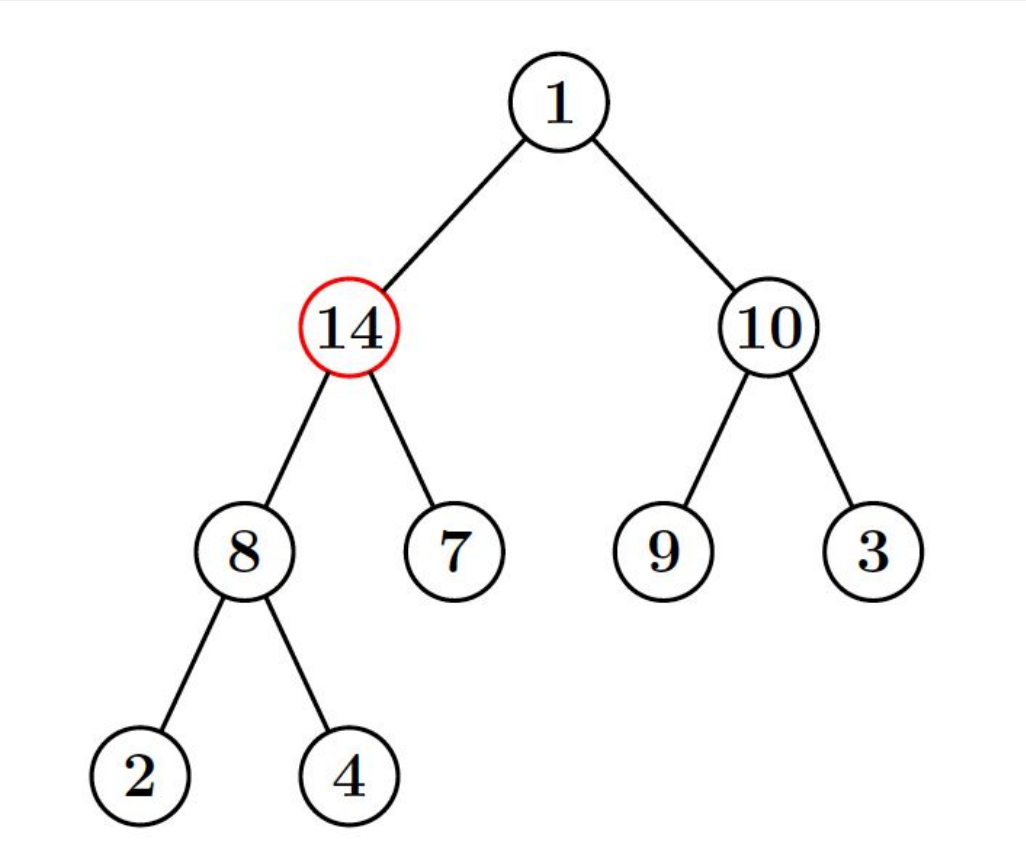
Take the final value in the array and assign it to the 0th index.

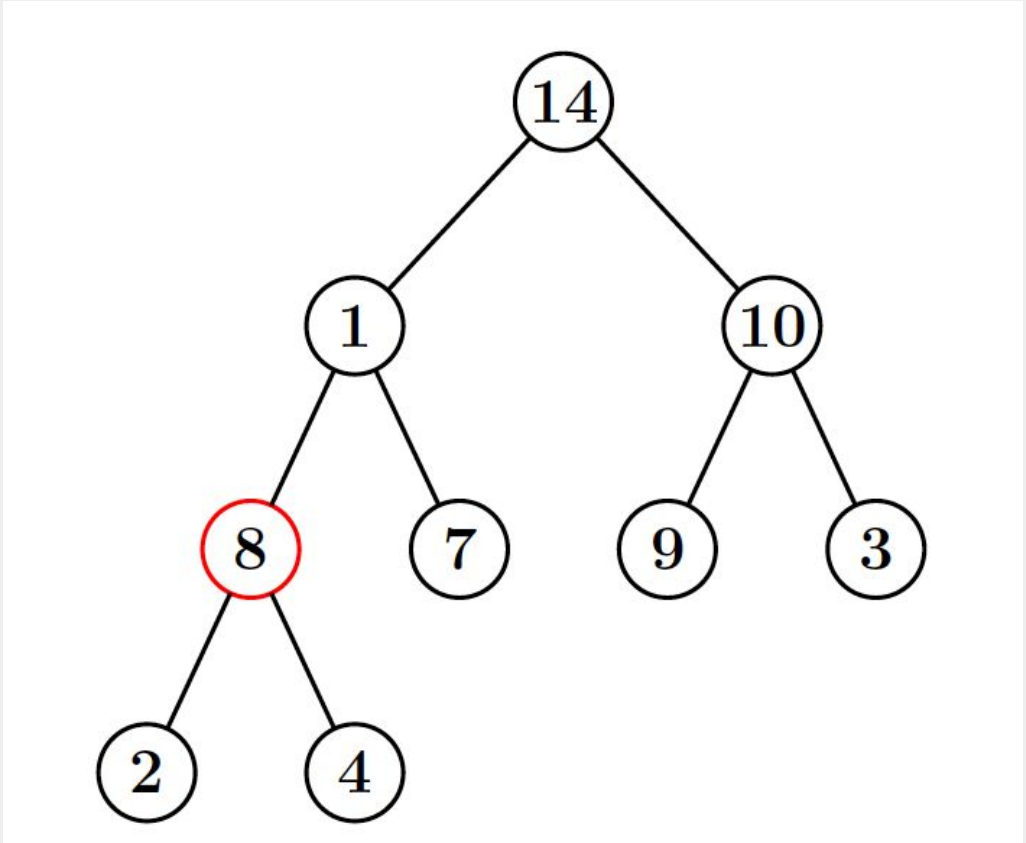
Send the assigned value down until the heap structure is satisfied.

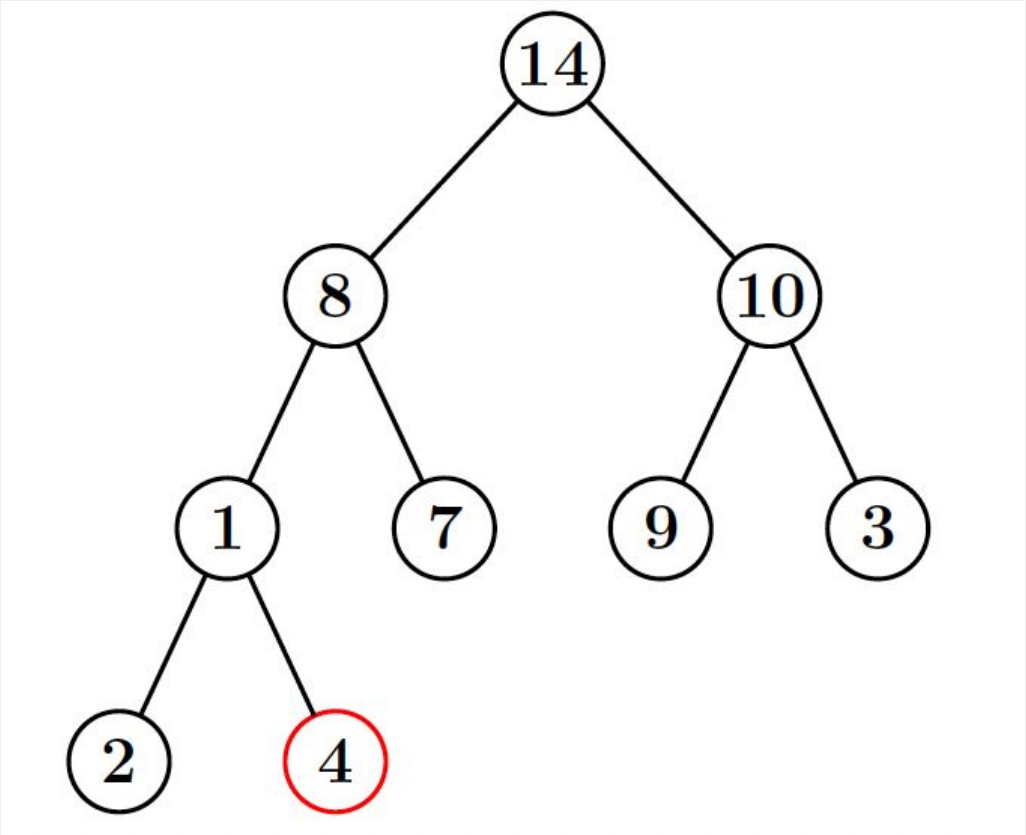
Decrement the count by one.

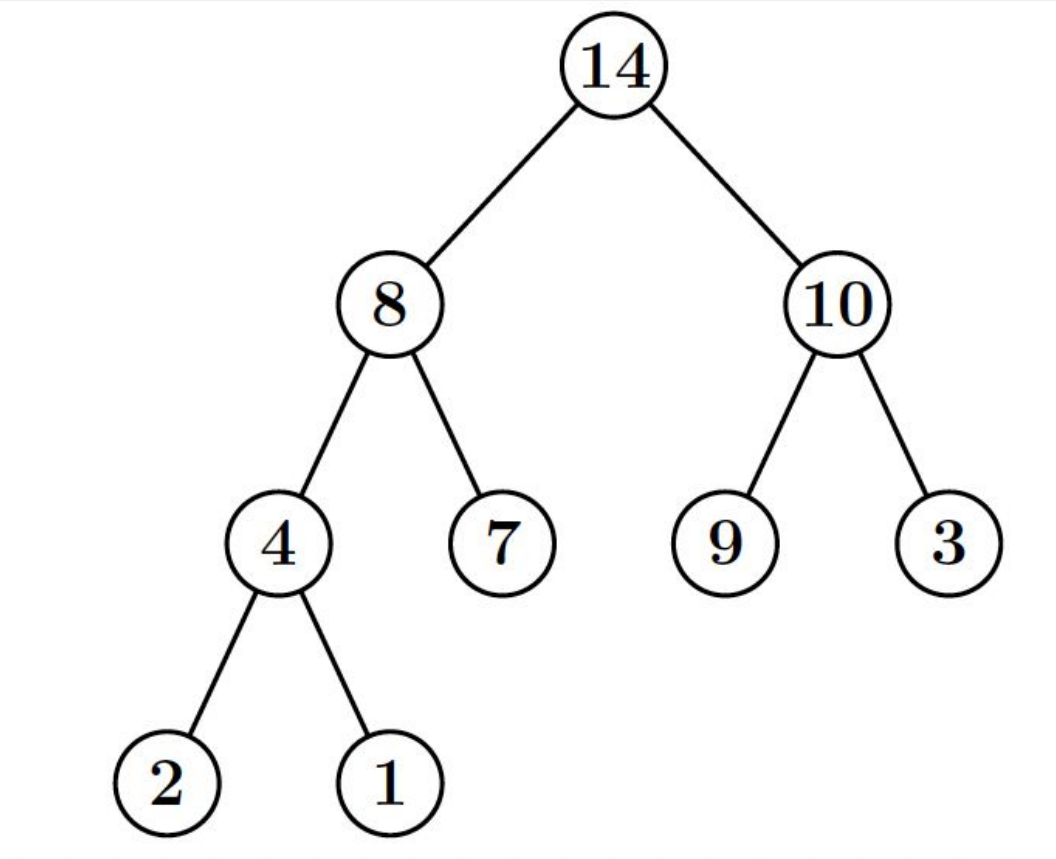
Return the highest priority value.


16
14
10
8
7
9
3
2
4
1

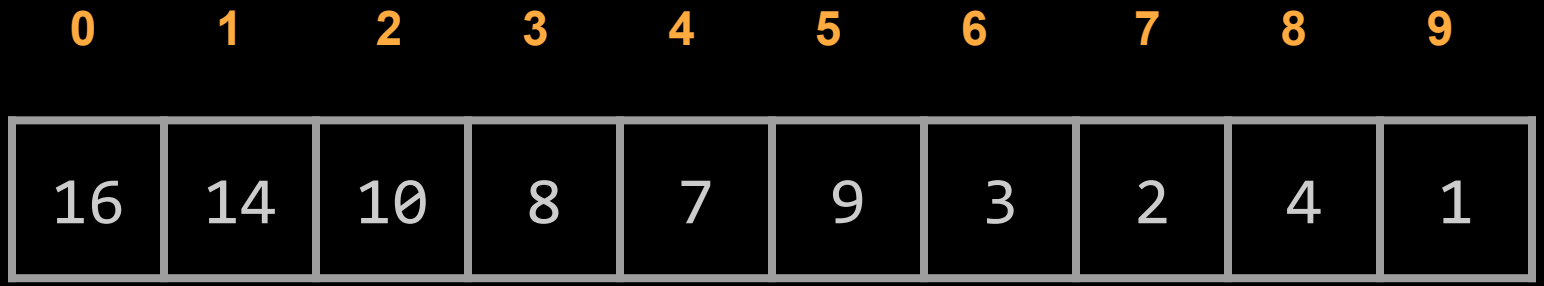

1
14
10
8
7
9
3
2
4


14
1
10
8
7
9
3
2
4


14
8
10
1
7
9
3
2
4


14
8
10
4
7
9
3
2
1

N = 10
count = 10



N = 10

count = 9

0	1	2	3	4	5	6	7	8	9
1	14	10	8	7	9	3	2	4	1

N = 10

count = 9

0	1	2	3	4	5	6	7	8	9
14	1	10	8	7	9	3	2	4	1

N = 10

count = 9

0	1	2	3	4	5	6	7	8	9
14	8	10	1	7	9	3	2	4	1

N = 10

count = 9

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	1

```java
public void swapHeapNode(int index1, int index2){
    HeapNode tmpNode = array[index1];
    array[index1] = array[index2];
    array[index2] = tmpNode;
}
```

```java
public void percolateDown(int index) {
    int leftChild, rightChild, largerChild;

    while (true) {
        leftChild = 2 * index + 1;
        rightChild = 2 * index + 2 ;

        // Check if left or right child exists and find the larger child
        if (leftChild < count && (rightChild >= count ||
                array[leftChild].data > array[rightChild].data)) {
            largerChild = leftChild;
        } else if (rightChild < count) {
            largerChild = rightChild;
        } else {
            break; // No children left to compare
        }

        // If the current node is smaller than the larger child, swap them
        if (array[index].data < array[largerChild].data) {
            swapHeapNode(index, largerChild);
            index = largerChild; // Move to the next index
        } else {
            break; // Heap property satisfied
        }
    }
}
```

Form an infinite loop and update left and right for each iteration.

Left child is larger if:
-the left child of current index is within bounds +
- if right child is out of bounds
OR data of left is larger than right
ELSE Right child is larger if:
- the right child is within bounds
ELSE
there are no children to compare.

If the data at the current index is smaller than the larger child, swap them and move to the next index. If the data at the current index is not smaller, the heap property is satisfied: break.

insertion

```java
public void insert(HeapNode node){
    count++;
    array[count-1] = node;
    percolateUp( addedIndex: count-1);
}
```

increment the number of nodes

place the given node at the final index

move it up until the heap structure is satisfied

N = 10

count = 9

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	

N = 10

count = 10

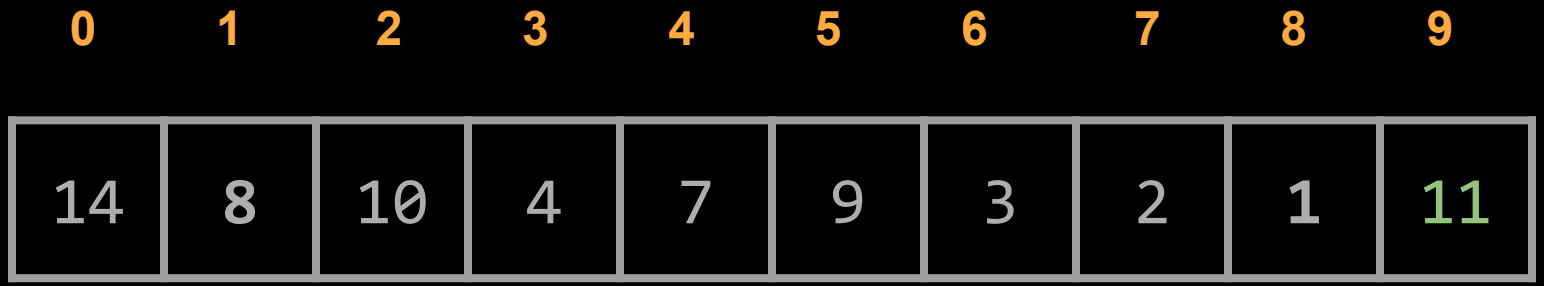
```
count++;
```

0	1	2	3	4	5	6	7	8	9
14	8	10	4	7	9	3	2	1	

N = 10

count = 10

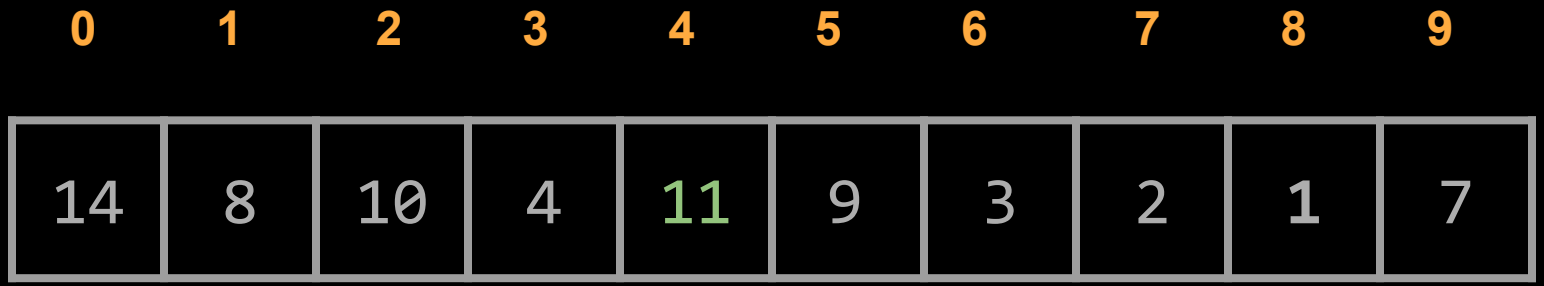
```
array[count-1] = newValue;
```



N = 10

count = 10

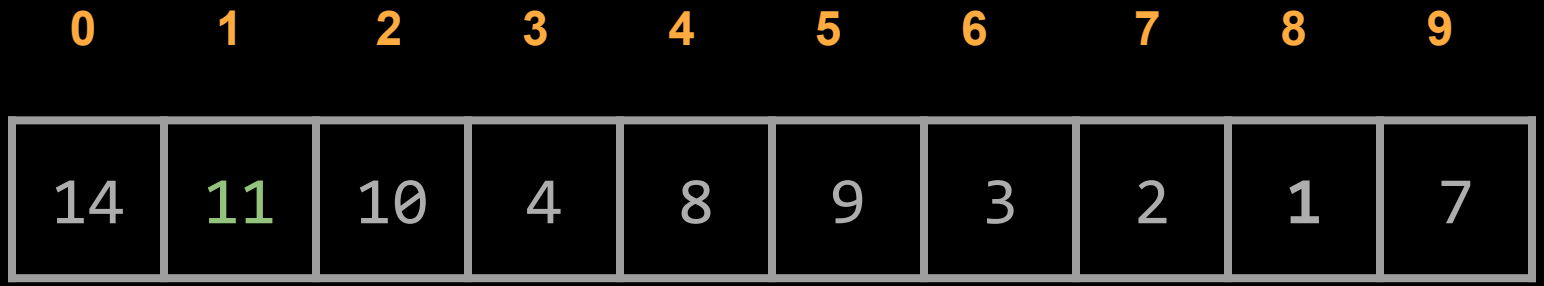
```
array[count-1] = newValue;
```



N = 10

count = 10

```
array[count-1] = newValue;
```



```java
private void percolateUp(int addedIndex) {
    int parent = (addedIndex - 1) / 2;

    while (parent > -1){
        if (array[parent].data < array[addedIndex].data){
            swapHeapNode(parent, addedIndex);
            addedIndex = parent;
            parent = (addedIndex - 1) / 2;
        } else {
            break;
        }
    }
}
```

Until parent’s index is negative:

If the data of parent is smaller than the data at the index, swap the nodes.

Update the current index to the parent index. Update the parent index.

If the data of the parent is not smaller, break out.

search

```
public int search(int name){
    int i;

    for (i = 0; i < count; i++){
        if (array[i].name == name){
            return i;
        }
    }
    return -1;
}
```

The heap data structure is not designed to search a specific element efficiently, but is designed to get the maximum priority element as fast as possible.

We should check each index either according to the value or the name. We return the index if we find it, we return -1 if we can’t find it.

N = 10

update

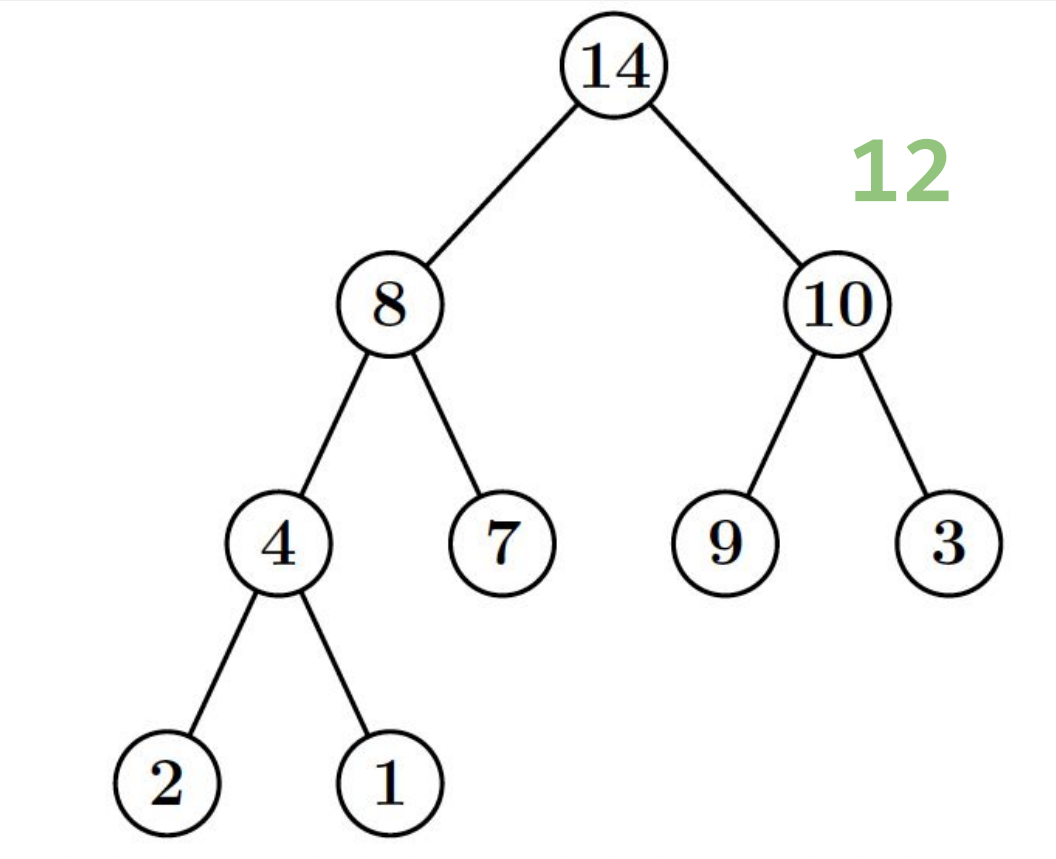
```java
public void update(int index, int newValue){
    int oldValue = array[index].data;
    array[index].data = newValue;

    if (oldValue > newValue){
        percolateDown(index);
    } else {
        percolateUp(index);
    }
}
```

Keep the old value at the given index in a separate variable.

Change the old value with the new value.

If the new value is **smaller** than the old one, try sending it **lower**. If the new value is **greater**, try sending it **higher** up the heap. If the new value does not mess with the heap structure, it stays where it is.


14
12
8
10
4
7
9
3
2
1