HEAPS

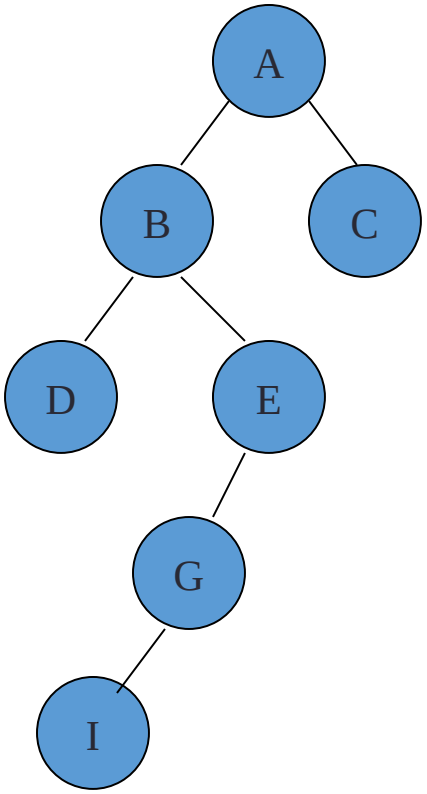
CSC212: Data Structures

Sequential Representation of binary trees

- There are three methods of representing a binary tree using array representation.
  1. Using index values to represent edges:

```
class Node<T> {
    T       data;
    int     left;
    int     right;
}
Node<T>[] BinaryTree=new Node[TreeSize];
```

Method 1: Example



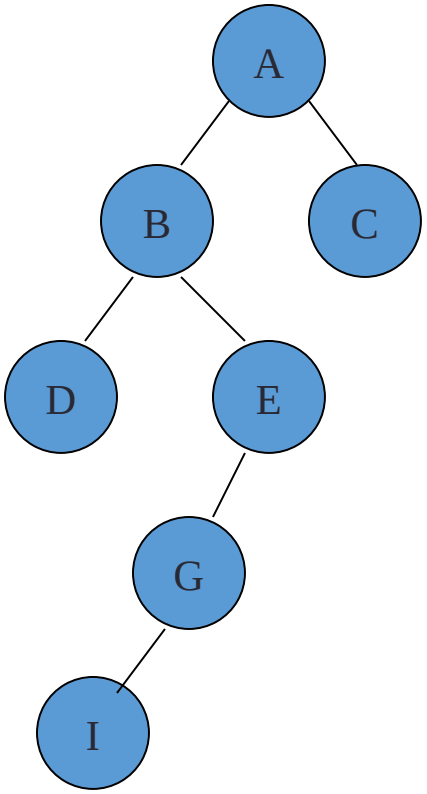
Index	Element	Left	Right
1	A	2	3
2	B	4	6
3	C	0	0
4	D	0	0
5	I	0	0
6	E	7	0
7	G	5	0

Method 2

2. Store the nodes in one of the natural traversals:

```
class Node<T> {
    T       data;
    boolean  left;
    boolean  right;
};
Node<T>[] BinaryTree=new
Node<T>[TreeSize];
```

Method 2: Example



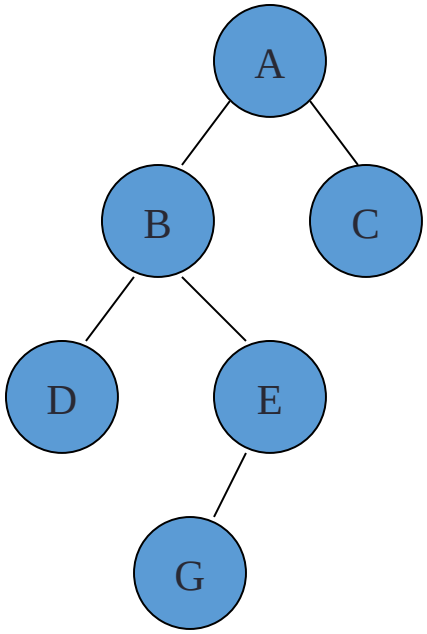
Index	Element	Left	Right
1	A	T	T
2	B	T	T
3	D	F	F
4	E	T	F
5	G	T	F
6	I	F	F
7	C	F	F

Elements stored in Pre-Order traversal

Method 3

3. Store the nodes in fixed positions: (i) root goes into first index, (ii) in general left child of tree[i] is stored in tree[2i] and right child in tree[2i+1].

Method 3: Example

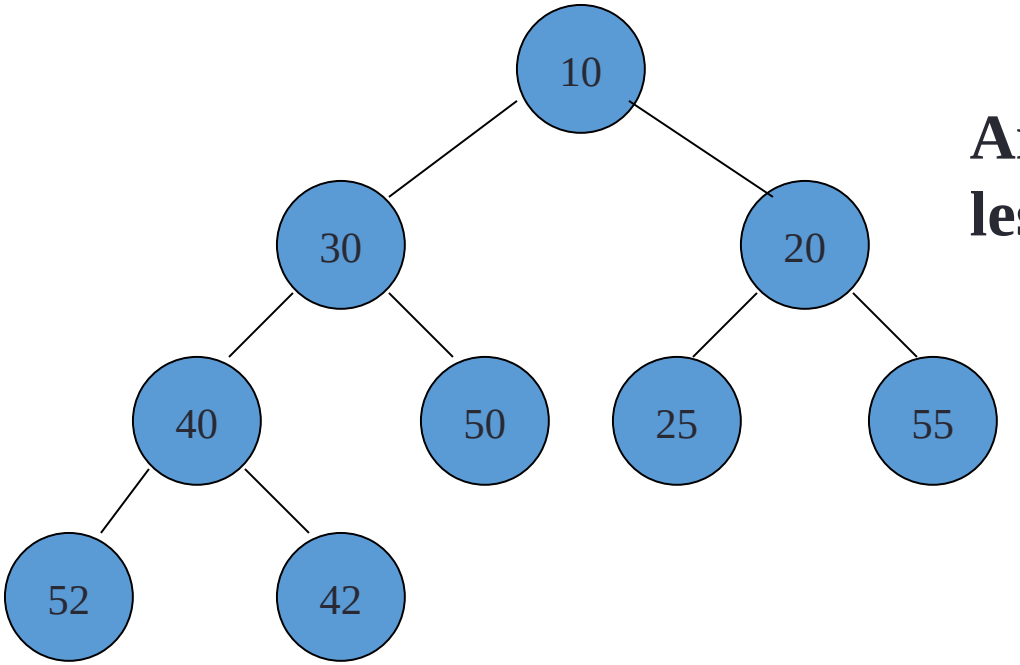


A	B	C	D	E	-	-	-	-	G	-	
1	2	3	4	5	6	7	8	9	10	11	12

Heaps

- A heap is a complete binary tree.
- A heap is best implemented in sequential representation (using an array).
- Two important uses of heaps are:
  - (i) efficient implementation of priority queues
  - (ii) sorting -- Heapsort.

A Heap



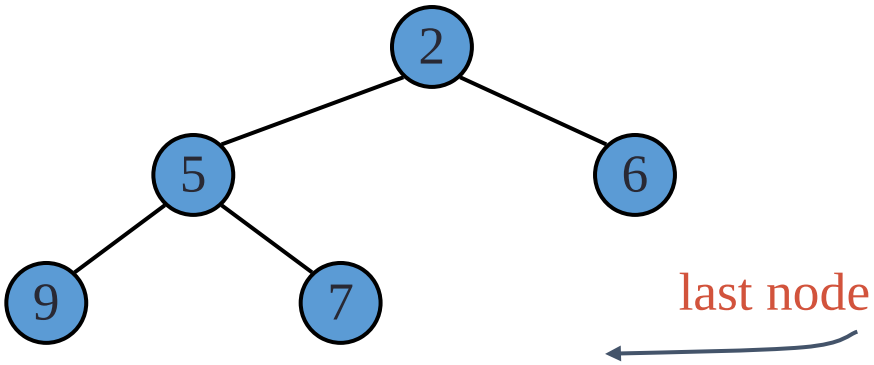
Any node's key value is less than its children's.

Heaps

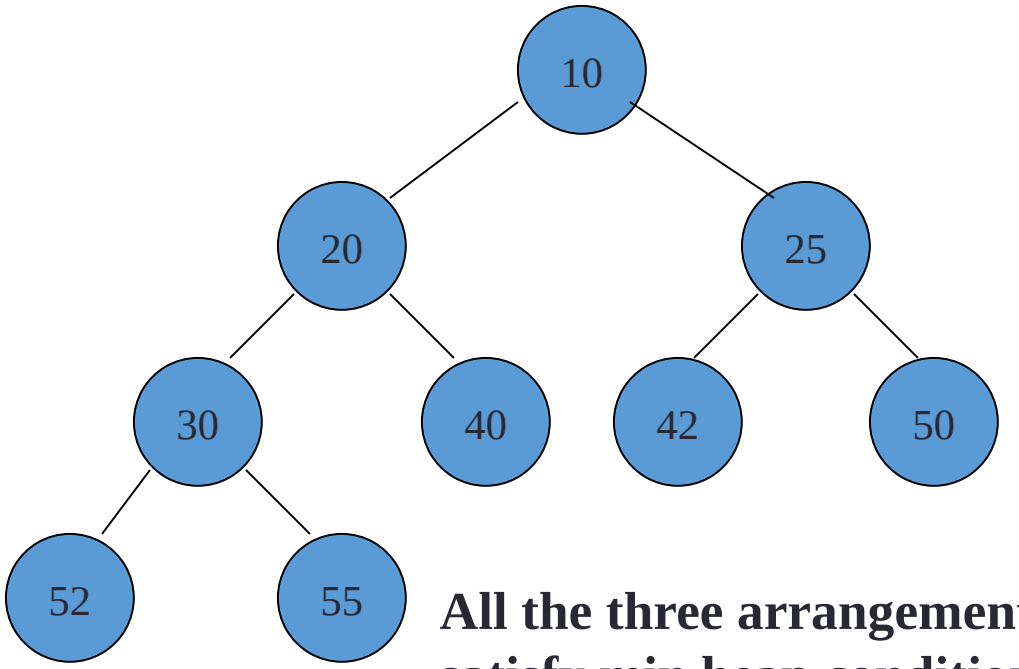
- Heaps are represented sequentially using the third method.
- Heap is a complete binary tree: shortest-path length tree with nodes on the lowest level in their leftmost positions.
- Complete Binary Tree: let ***h*** be the height of the heap
  - for ***i*** = 0, … , ***h*** - 1, there are 2^i nodes of depth ***i***
  - at depth ***h*** - 1, all nodes in the last level are as far left as possible

Heaps (Cont.)

- Max-Heap has max element as root. Min-Heap has min element as root.
- The elements in a heap satisfy heap conditions: for Min-Heap: key[parent] <= key[left-child] and key[right-child].
- The last node of a heap is the rightmost node of maximum depth



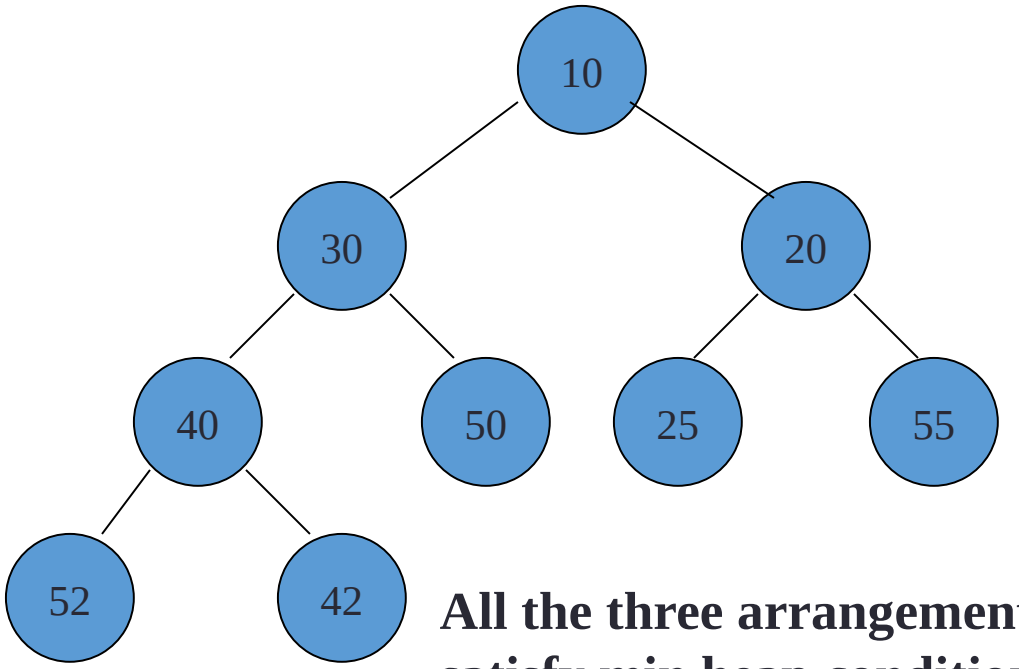
Heap: An example



All the three arrangements satisfy min heap conditions

[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

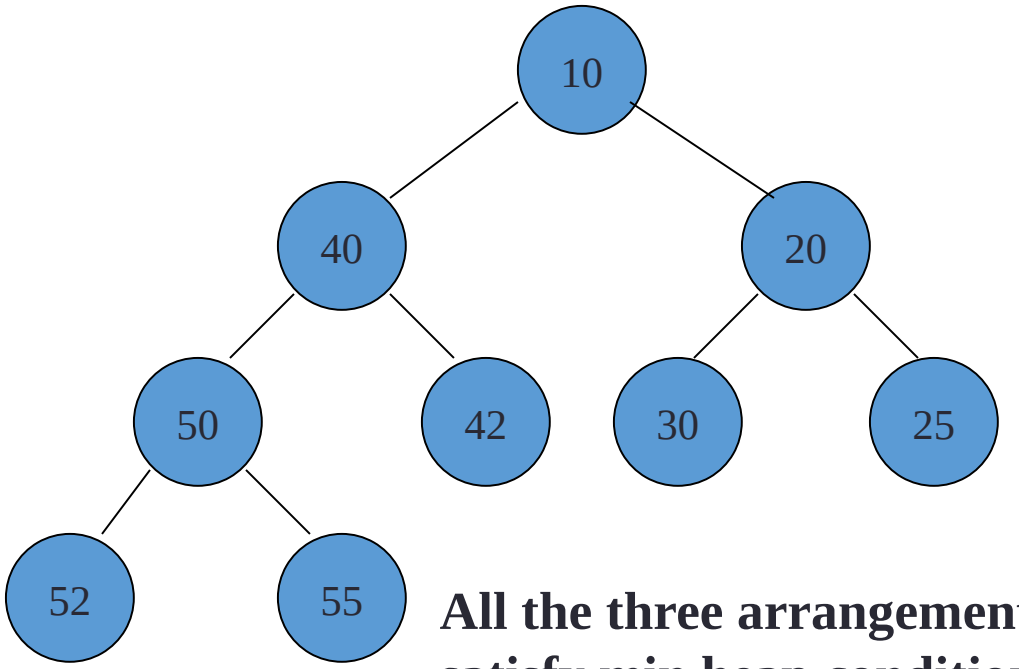
Heap: An example



All the three arrangements satisfy min heap conditions

[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

Heap: An example



All the three arrangements satisfy min heap conditions

[1]	10	10	10
[2]	20	30	40
[3]	25	20	20
[4]	30	40	50
[5]	40	50	42
[6]	42	25	30
[7]	50	55	25
[8]	52	52	52
[9]	55	42	55

ADT Heap

Elements: The elements are called HeapElements.

Structure: The elements of the heap satisfy the heap conditions.

Domain: Bounded. Type name: Heap.

ADT Heap

Operations:

- **Method** SiftUp ()
  Input: none. **requires**: Elements H[1],H[2],…,H[n-1] satisfy heap conditions.
  results: Elements H[1],H[2],…,H[n] satisfy heap conditions. **Output**: none.
- **Method** SiftDown (int i)
  Input: i. **requires**: Elements H[i+1],H[i+2],…,H[n] satisfy the heap conditions.
  results: Elements H[i],H[i+1],…,H[n] satisfy the heap conditions.
  Output: none.
- **Method** Insert(int key, T data)
  input: key, data. **requires**: Elements H[1],H[2],…,H[n] satisfy heap conditions.
  results: The key and data are inserted in H[n+1]. Elements H[1],H[2], ….H[n+1] must satisfy the heap conditions. **Output**: none

ADT Heap

Operations:

- **Method** RemoveRoot(HeapElement<T> result)
  input: none. **requires**: Elements H[1],H[2],…,H[n] satisfy heap condition.
  results: The HeapElement in H[1] is removed, and it is value is assigned to result. Elements H[1],H[2],….H[n-1] must satisfy the heap conditions. **output**: none.
- **Method** Full(boolean result)
- **Method** Size(int result)

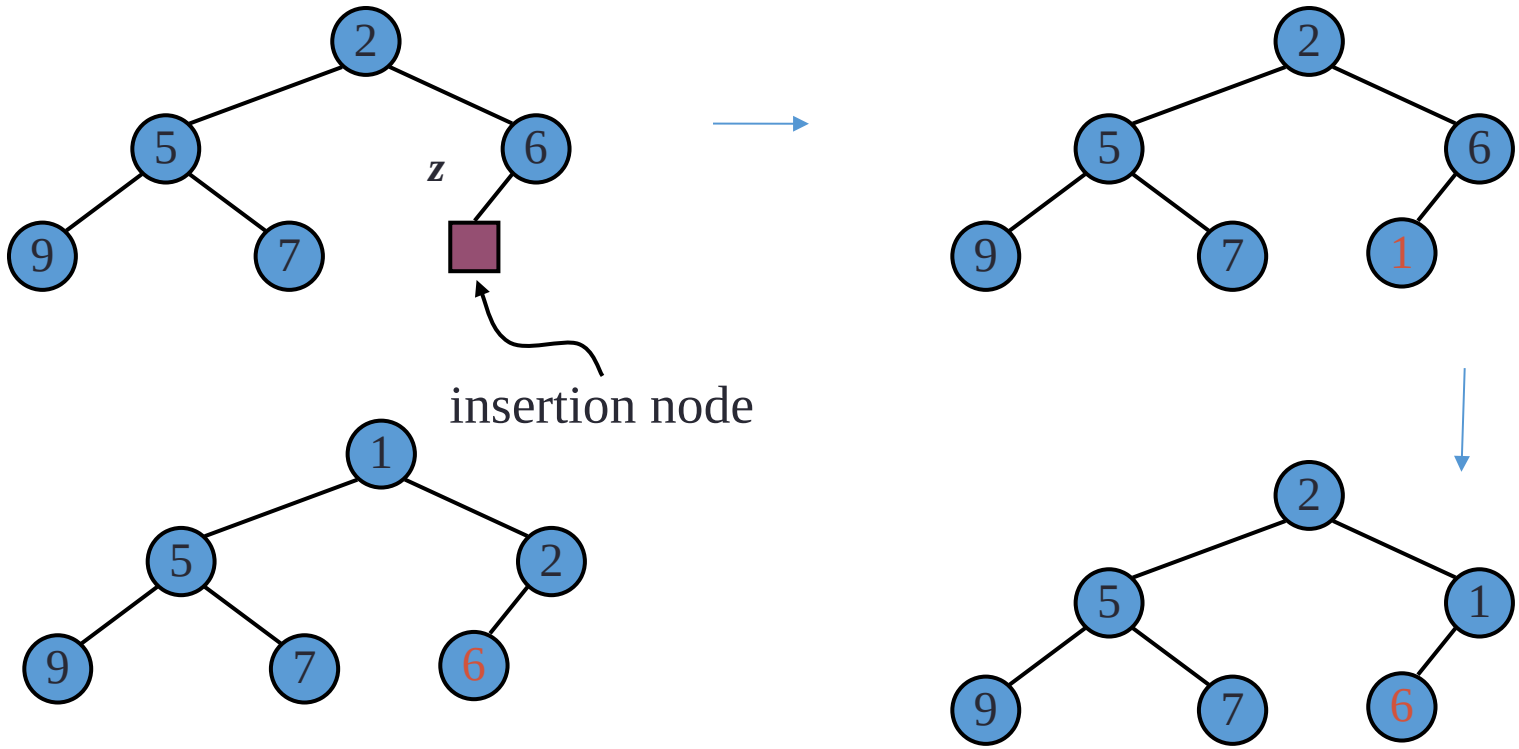
Insertion into a Heap

- The insertion algorithm consists of three steps
  - Find the insertion node z (the new last node)
  - Store k at z
  - Restore the heap-order property (discussed next)

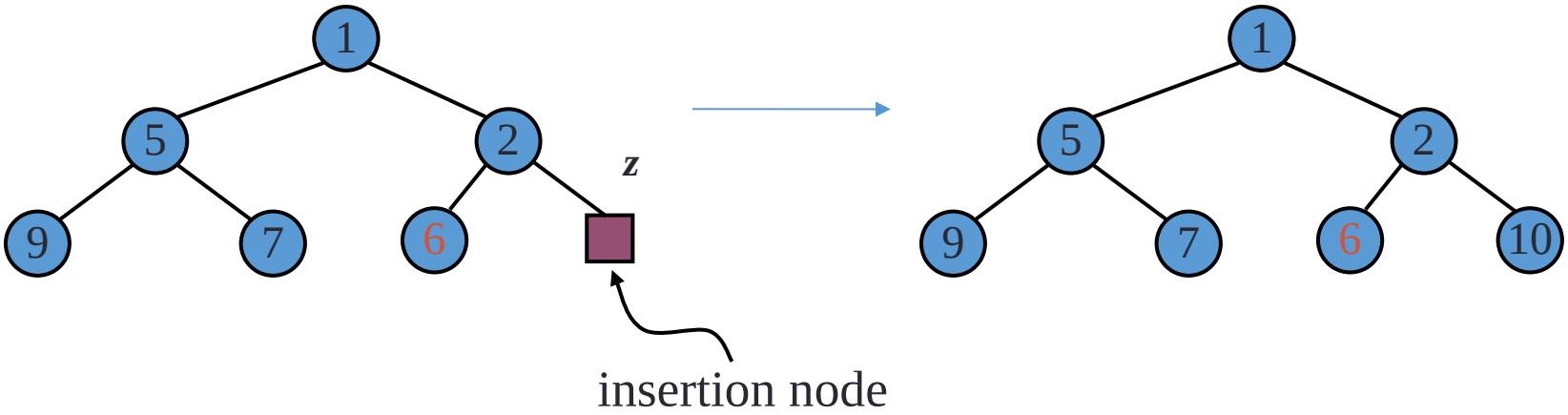
Upheap

- After the insertion of a new key k, the heap-order property may be violated
- Algorithm upheap (siftUp) restores the heap-order property by swapping k along an upward path from the insertion node
- Upheap terminates when the key k reaches the root or a node whose parent has a key smaller than or equal to k
- Since a heap has height $O(\log n)$, upheap runs in $O(\log n)$ time

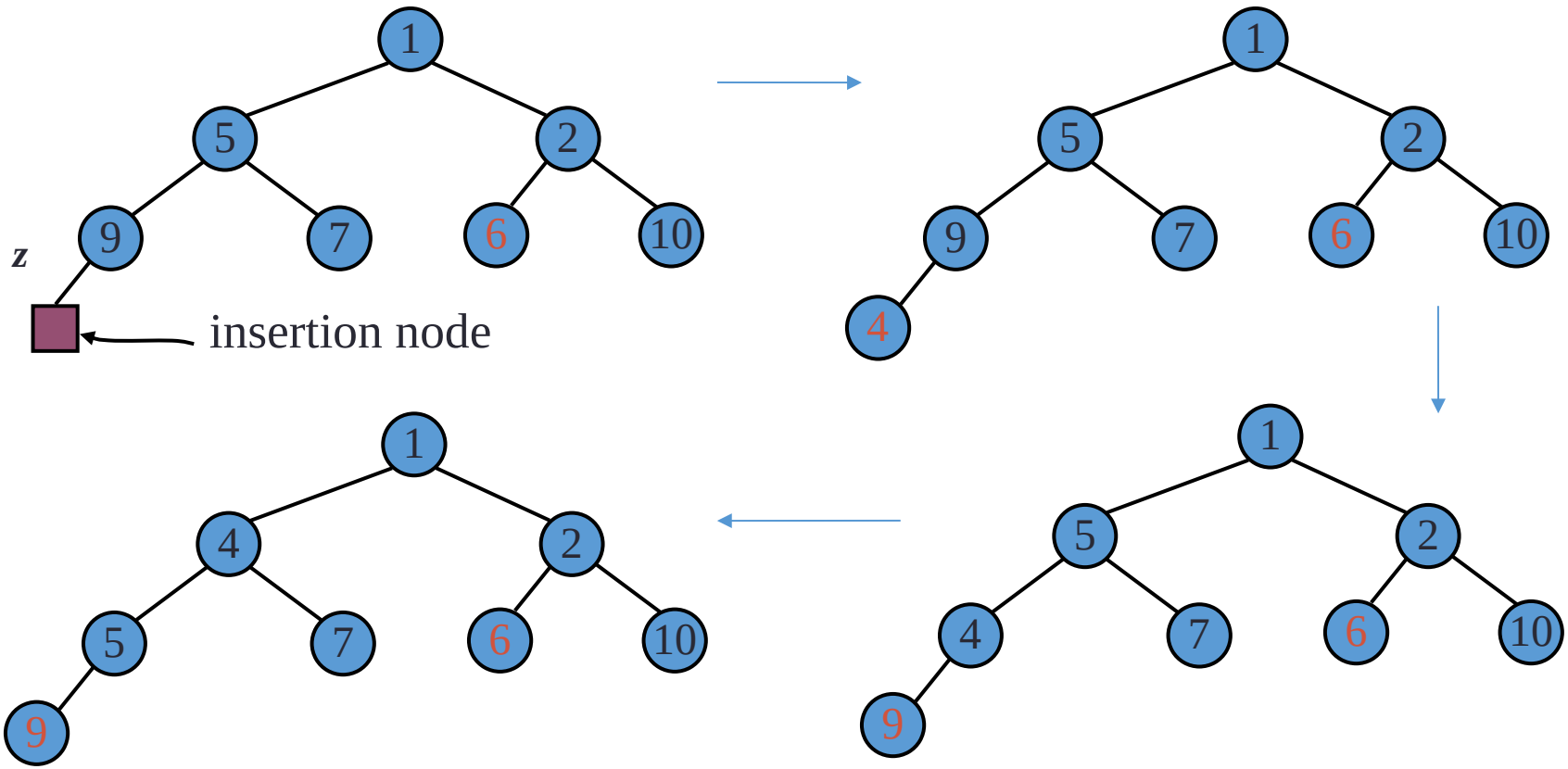
Example 1



Example 2

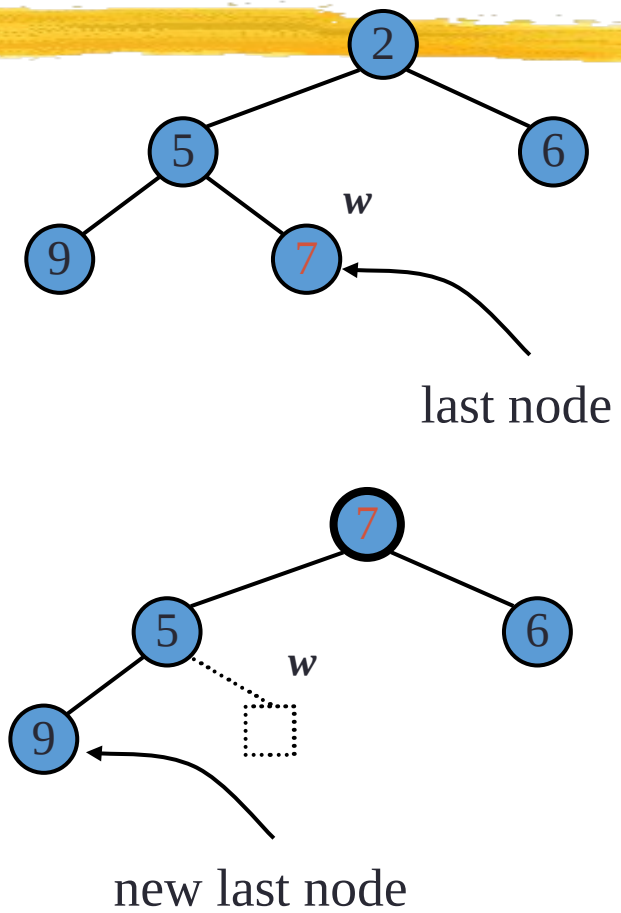


Example 3



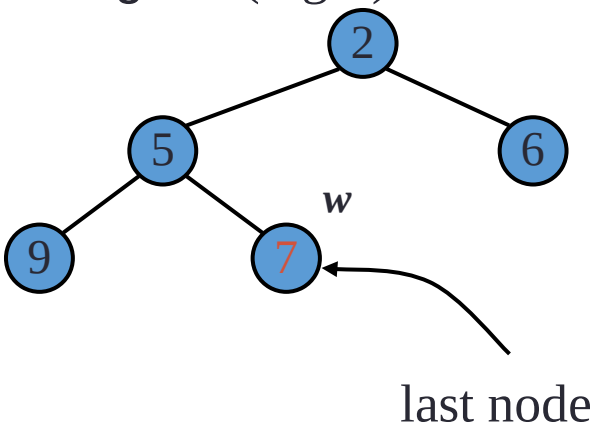
Removal from a Heap

- The removal algorithm consists of three steps
  - Replace the root key with the key of the last node ***w***
  - Remove ***w***
  - Restore the heap-order property (discussed next)

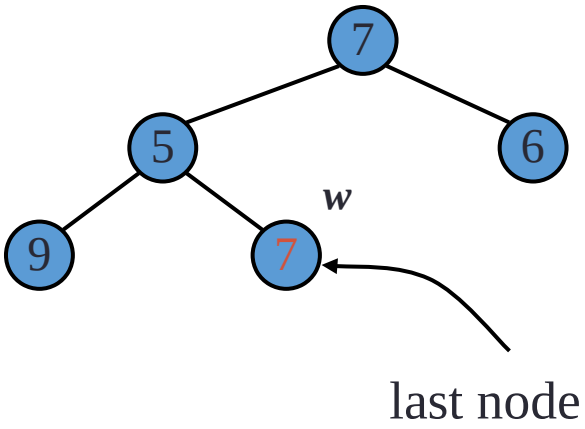


Downheap

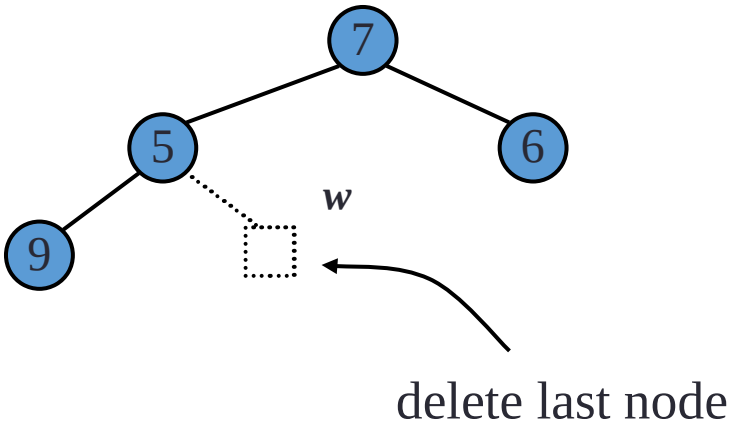
- After replacing the root key with the key k of the last node, the heap-order property may be violated
- Algorithm downheap (siftDown) restores the heap-order property by swapping key k along a downward path from the root
- Downheap terminates when key k reaches a leaf or a node whose children have keys greater than or equal to k
- Since a heap has height $O(\log n)$, downheap runs in $O(\log n)$ time



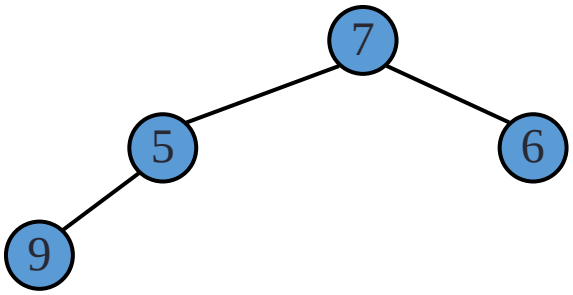
Downheap



Downheap

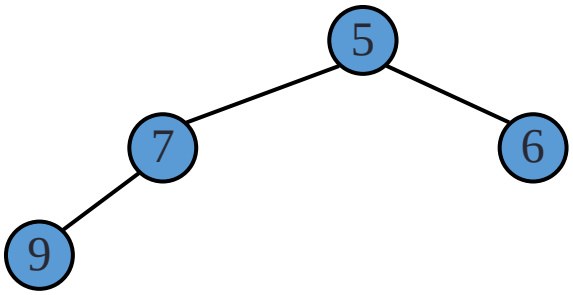


Downheap

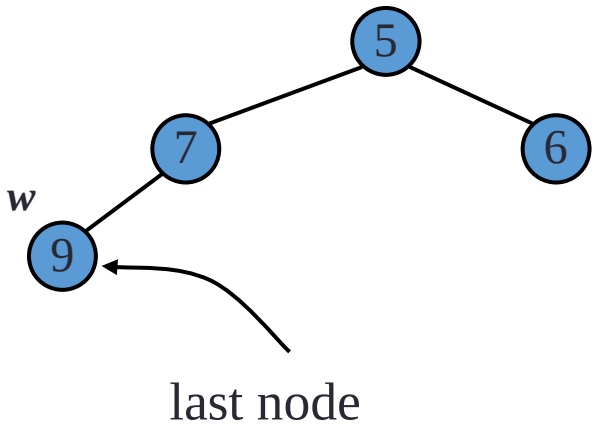


DownHeap/SiftDown

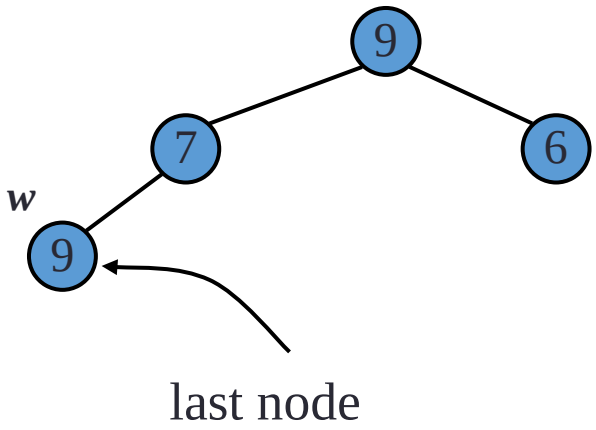
Downheap



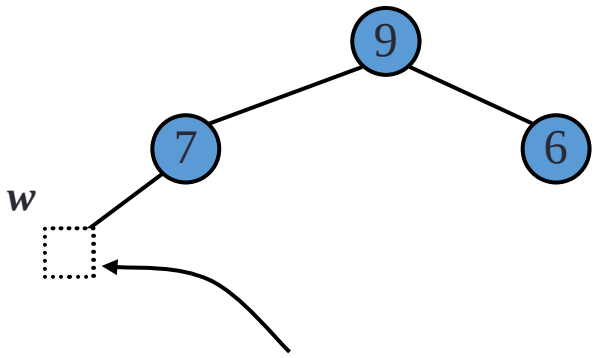
Downheap



Downheap

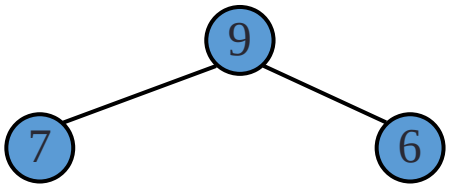


Downheap



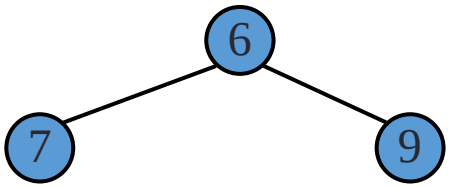
delete last node

Downheap

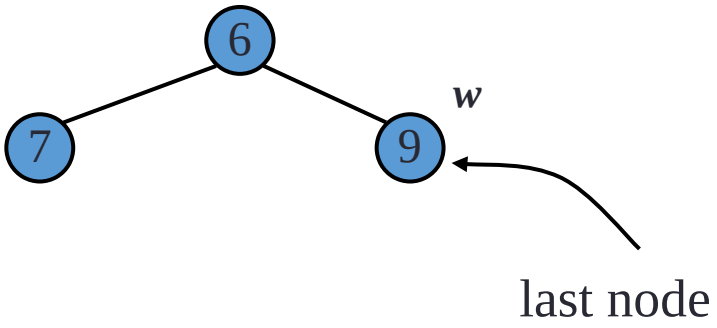


DownHeap/SiftDown

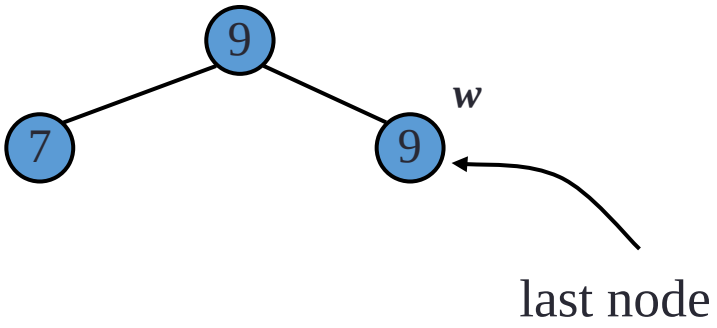
Downheap



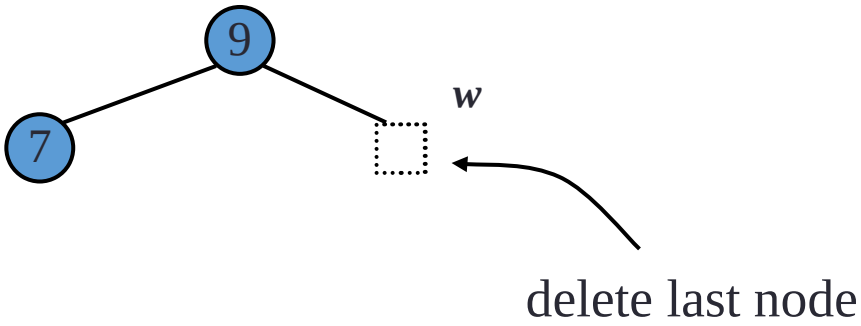
Downheap



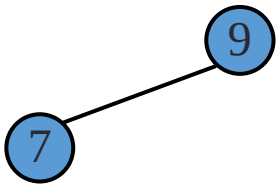
Downheap



Downheap

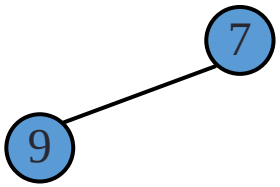


Downheap

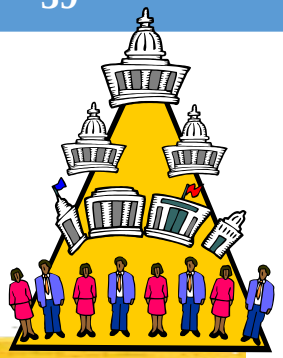


DownHeap/SiftDown

Downheap



Heap applications



Priority queue

- Consider a priority queue with n items implemented by means of a heap
  - the space used is $O(n)$
  - methods enqueue and serve take $O(\log n)$ time
  - methods length, full take time $O(1)$ time

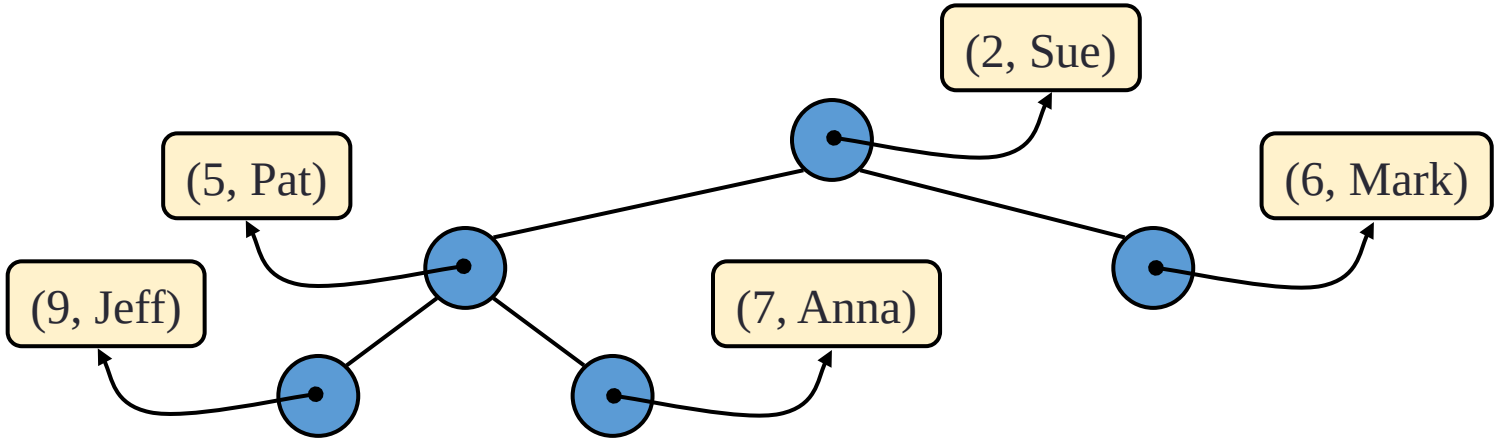
Heap sort

- Using a heap-based priority queue, we can sort a sequence of n elements in $O(n \log n)$ time
- The resulting algorithm is called heap-sort
- Heap-sort is much faster than quadratic sorting algorithms, such as bubble sort and selection-sort

Priority Queue

Heaps and Priority Queues

- We can use a heap to implement a priority queue
- We store a (key, element) item at each internal node
- We keep track of the position of the last node



ADT Heap: Element

```
public class HeapElem <T>{
    public int key;
    public T data;

    public HeapElem(int _key, T _data){
        key= _key;
        data= _data;
    }
}
```

Priority Queue as Heap

Representation as a Heap

```
public class HeapPQ<T> {

    private Heap<T> heap;

    public HeapPQ(int _maxSize){
        heap= new Heap<T>(_maxSize);
    }
```

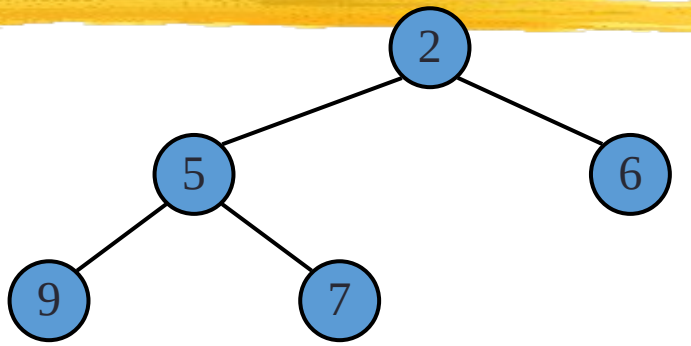
Priority Queue as Heap

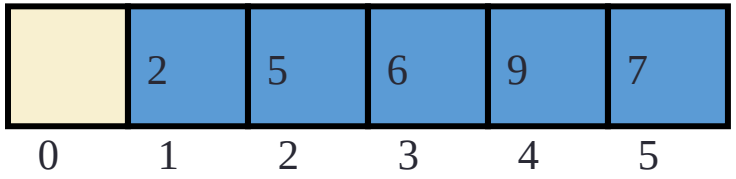
```
public int length(){
        return heap.size();
    }

    public boolean full(){
        return heap.full();
    }

    public void enqueue(int pr, T val){
        heap.insert(pr, val);
    }

    public HeapElem<T> serve(){
        return heap.removeRoot();
    }
```

Heap sort

Vector-based Heap Implementation

- We can represent a heap with n keys by means of a vector of length $n + 1$
- For the node at rank i
  - the left child is at rank $2i$
  - the right child is at rank $2i + 1$
- Links between nodes are not explicitly stored
- The cell at rank 0 is not used
- Operation insert corresponds to inserting at position $n + 1$
- Operation serve corresponds to removing at position n
- Yields in-place heap-sort



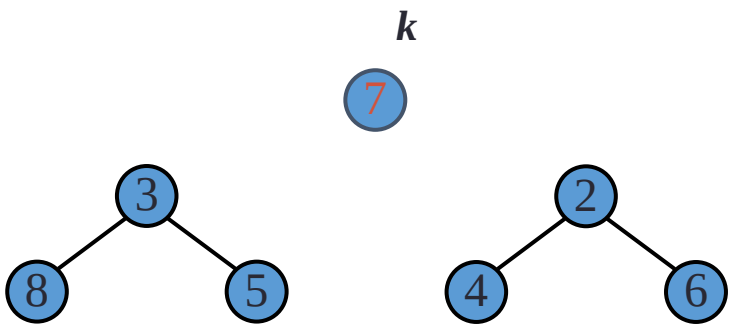


Merging Two Heaps

- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property

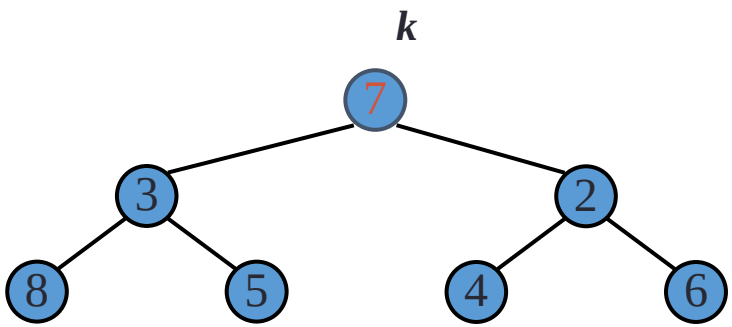
Merging Two Heaps

- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Merging Two Heaps

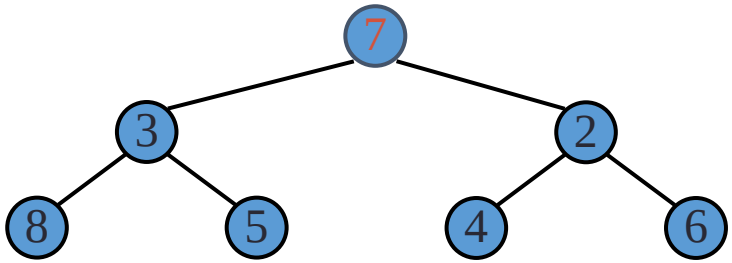
- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Merge

Merging Two Heaps

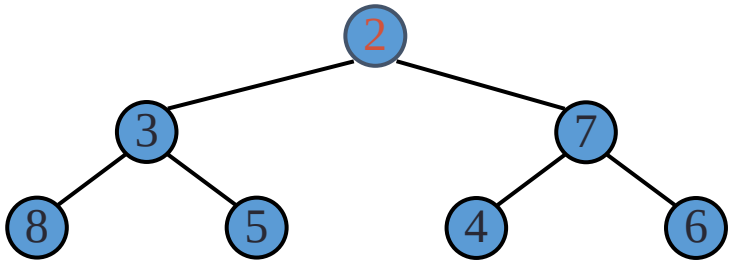
- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Downheap/SiftDown

Merging Two Heaps

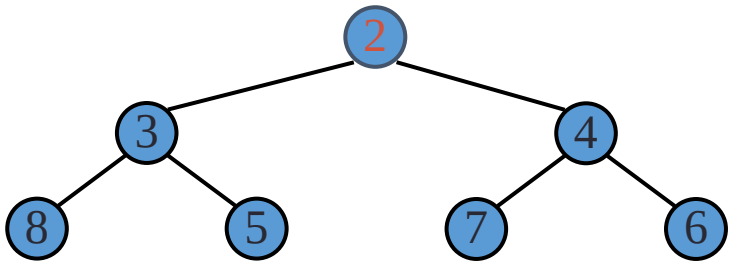
- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Downheap/SiftDown

Merging Two Heaps

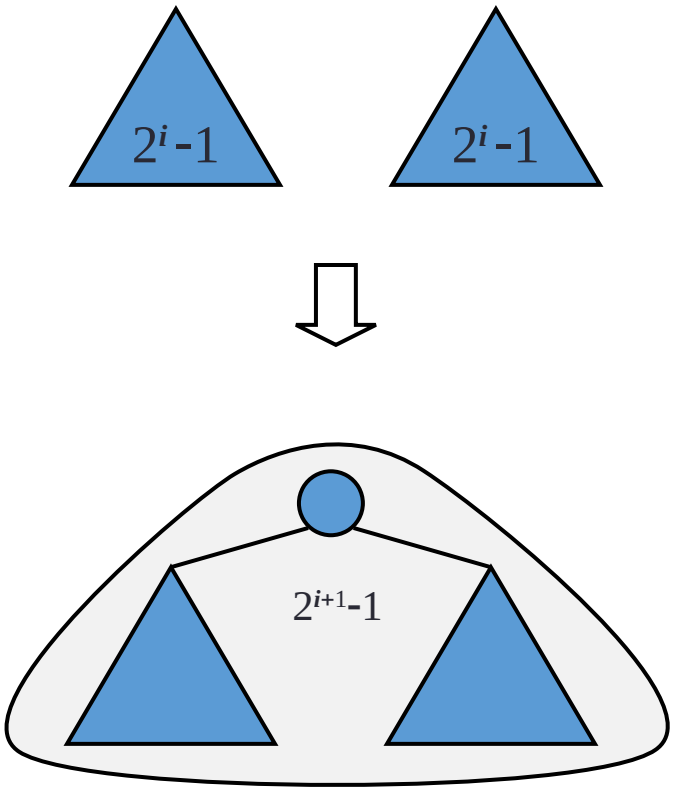
- We are given two two heaps and a key ***k***
- We create a new heap with the root node storing ***k*** and with the two heaps as subtrees
- We perform downheap to restore the heap-order property



Downheap/SiftDown

Bottom-up Heap Construction

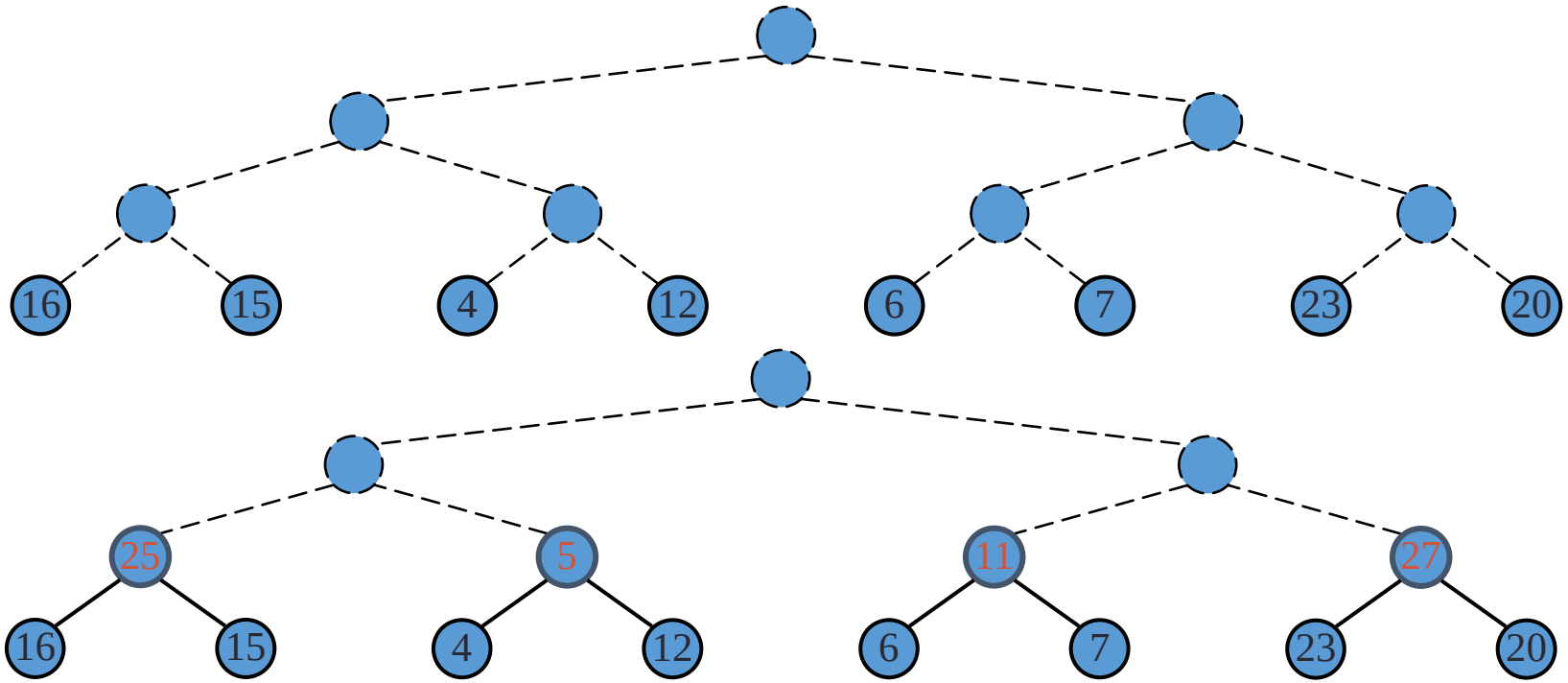
- We can construct a heap storing n given keys in using a bottom-up construction with $\log n$ phases
- In phase i, pairs of heaps with 2^i-1 keys are merged into heaps with $2^{i+1}-1$ keys



Example of bottom-up heap construction

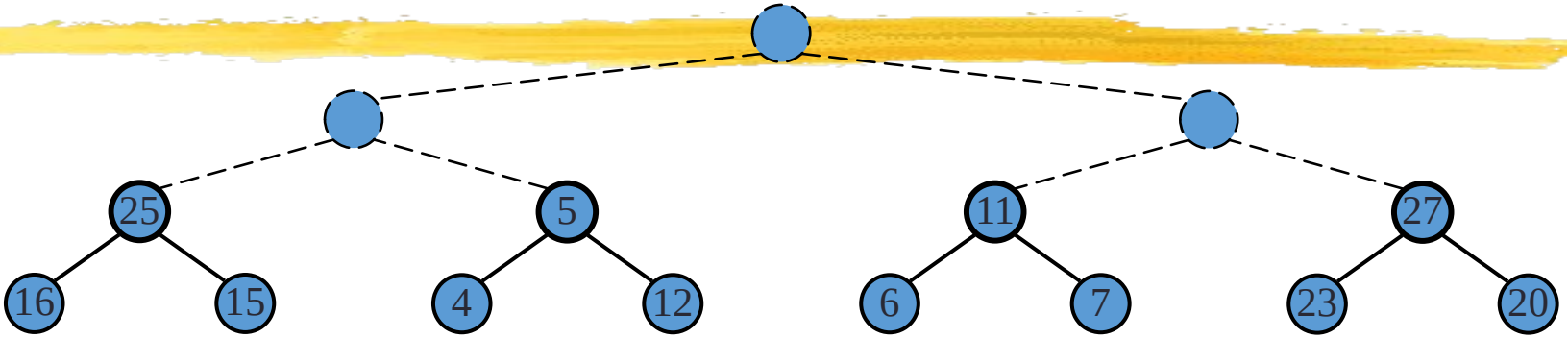
Initial array

10	7	8	25	5	11	27	16	15	4	12	6	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

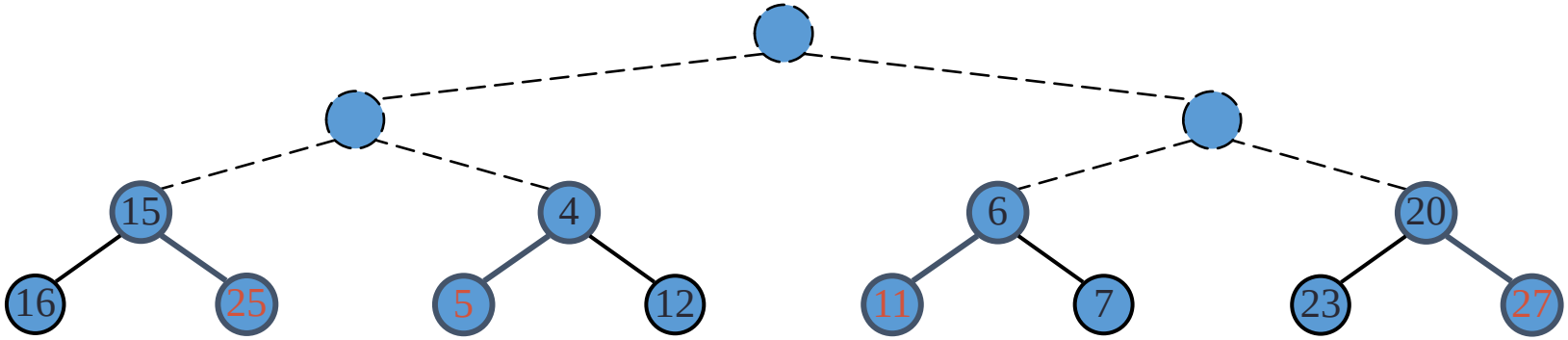


If there are n elements in the array, all elements after the index floor(n/2) become leaf nodes.

Example (contd.)



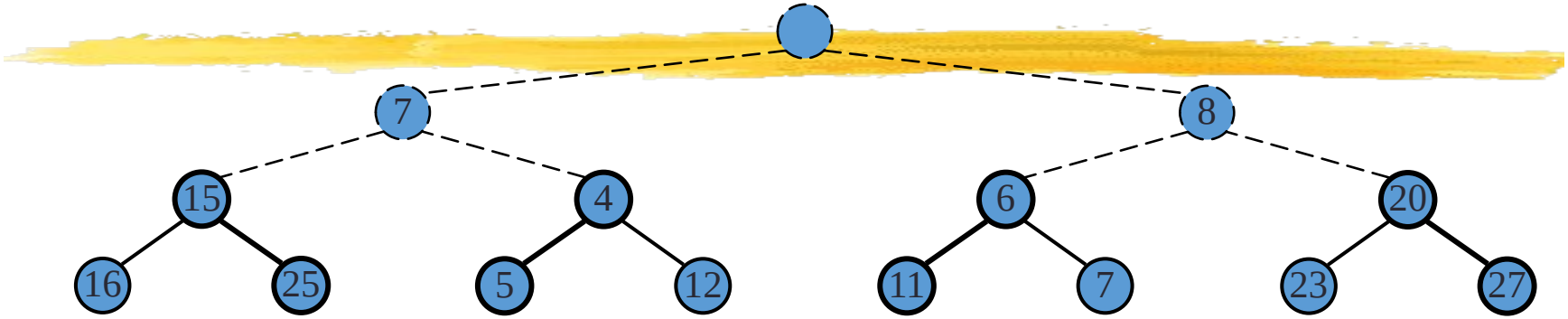
10	7	8	25	5	11	27	16	15	4	12	6	7	23	20
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



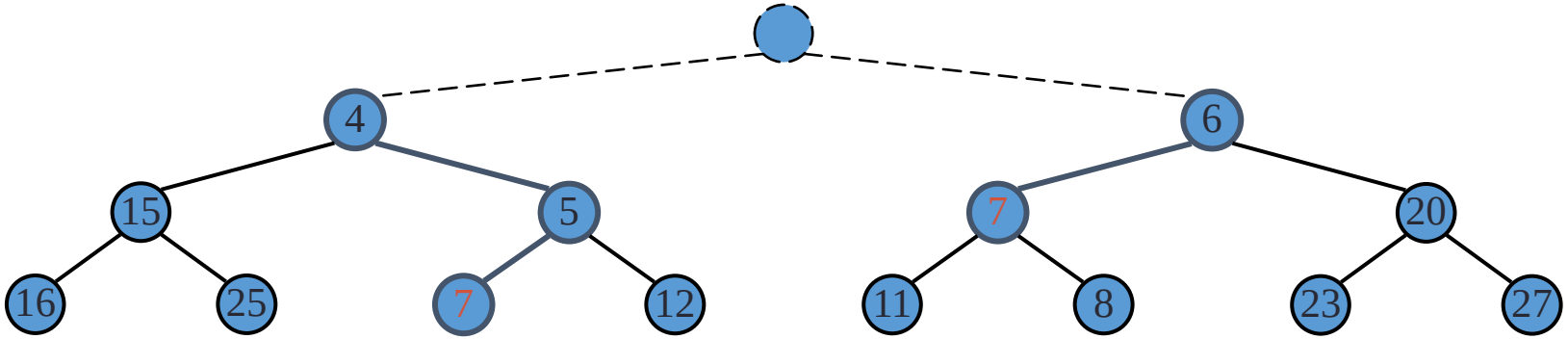
Sift down

10	7	8	15	4	6	20	16	25	5	12	11	7	23	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example (contd.)



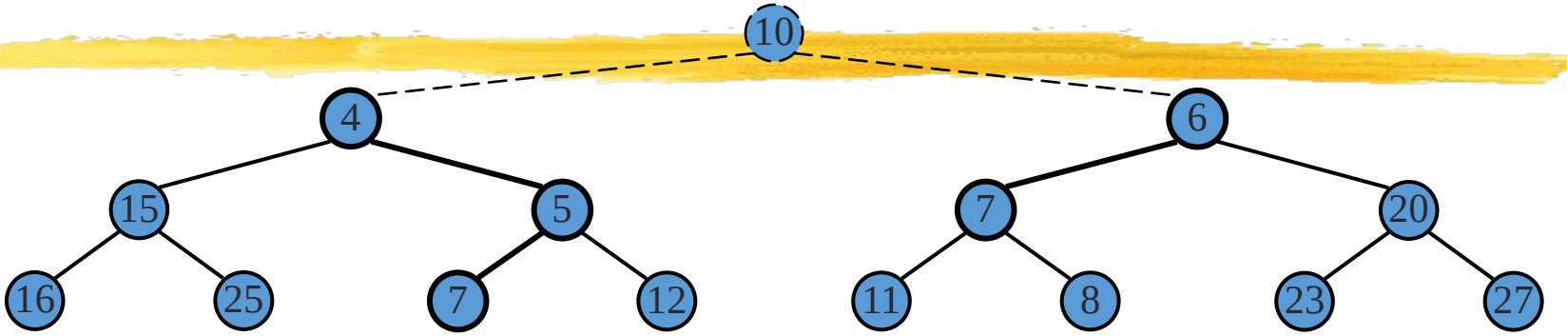
10	7	8	15	4	6	20	16	25	5	12	11	7	23	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



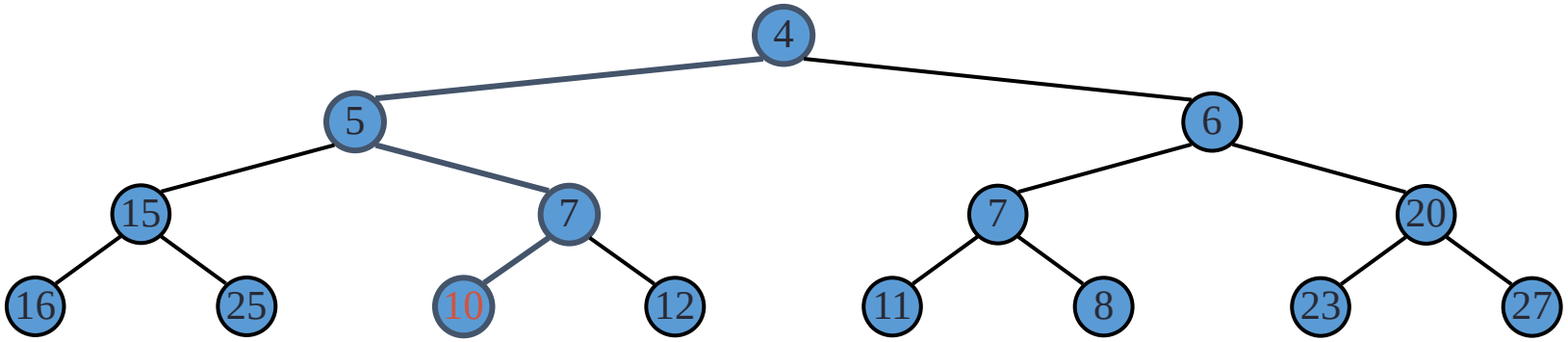
Sift down

10	4	6	15	5	7	20	16	25	7	12	11	8	23	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Example (end)



10	4	6	15	5	7	20	16	25	7	12	11	8	23	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



Sift down

4	5	6	15	7	7	20	16	25	10	12	11	8	23	27
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

HeapSort

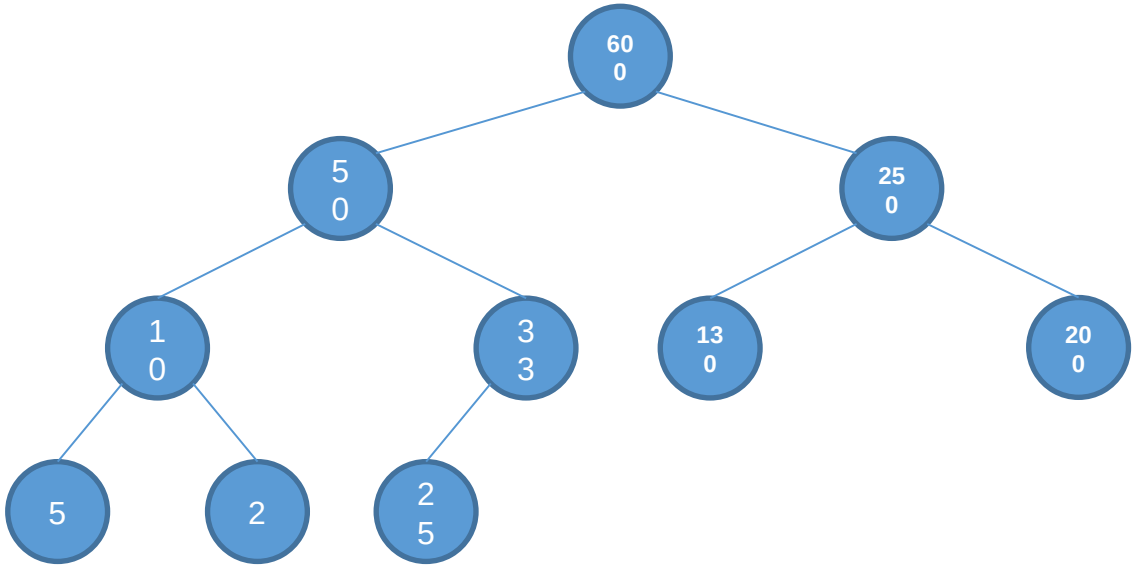
- Heap can be used for sorting.
- HeapSort based on the idea that heap always has the smallest or largest element at the root.
- Two step process:
  - Given arbitrary array (i.e. not a heap):
  - Max-heapify the array (build a max heap from the array) in order to sort the elements from smallest to largest number .
  - Delete max items one by one (thus moving max to end of array).
  - Items take a round trip.

ADT Heap: Implementation

```
public void sort(){

        int n= size;
        for(int i= 1; i<=n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Example of Heap-sorting

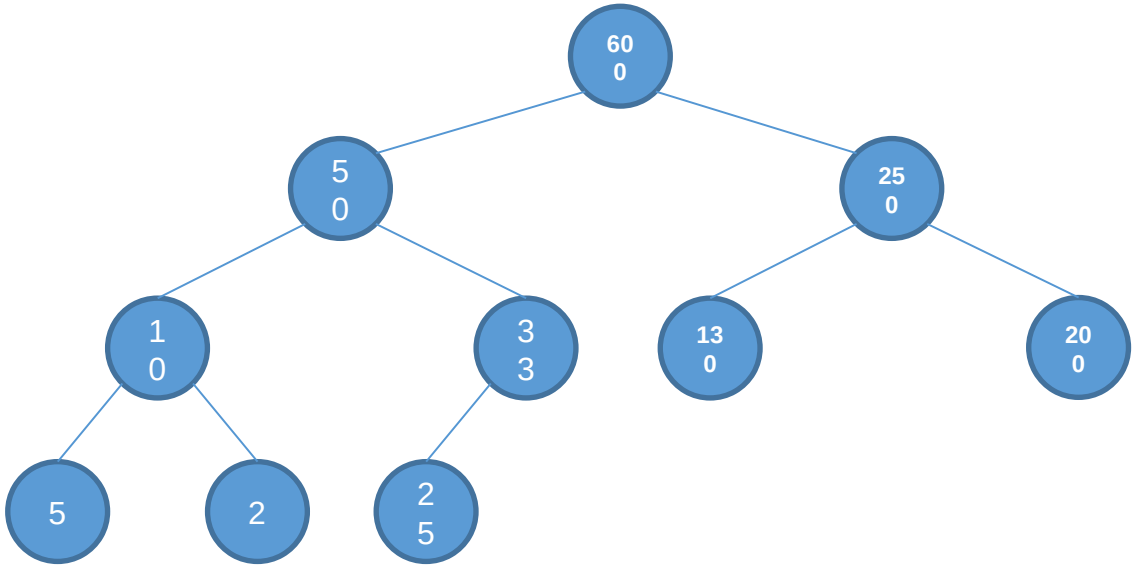
X	600	50	250	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 10

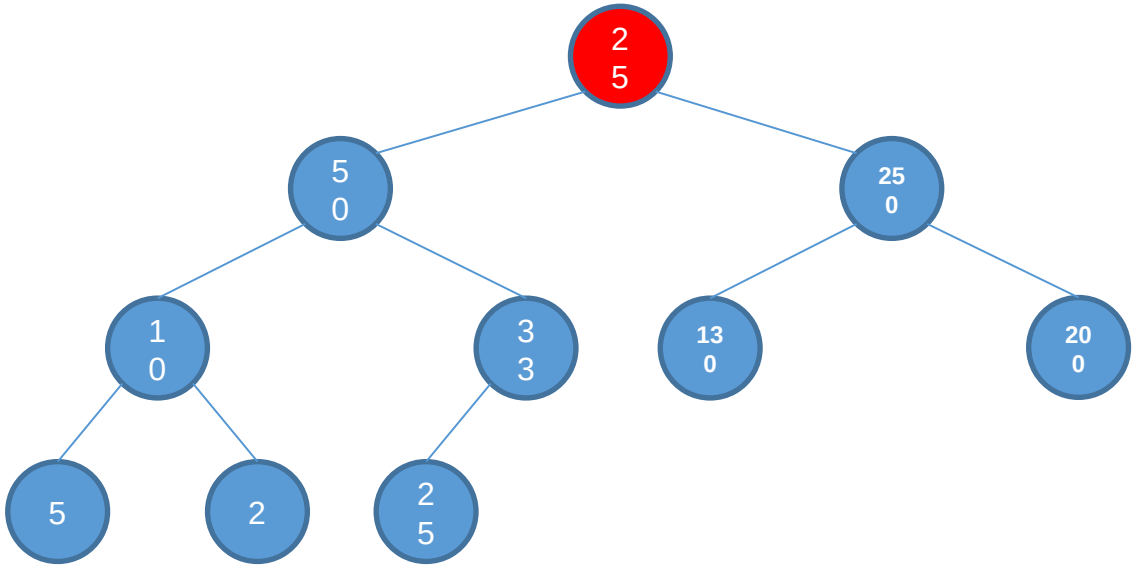
X	600	50	250	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 10
tempKey = 600

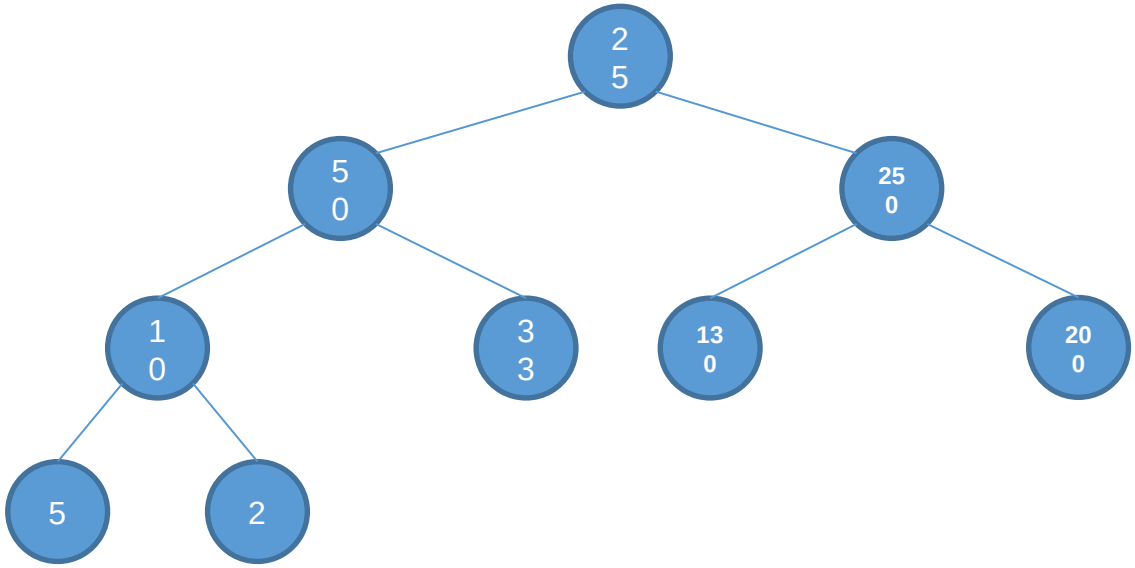
X	25	50	250	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 10
tempKey = 600

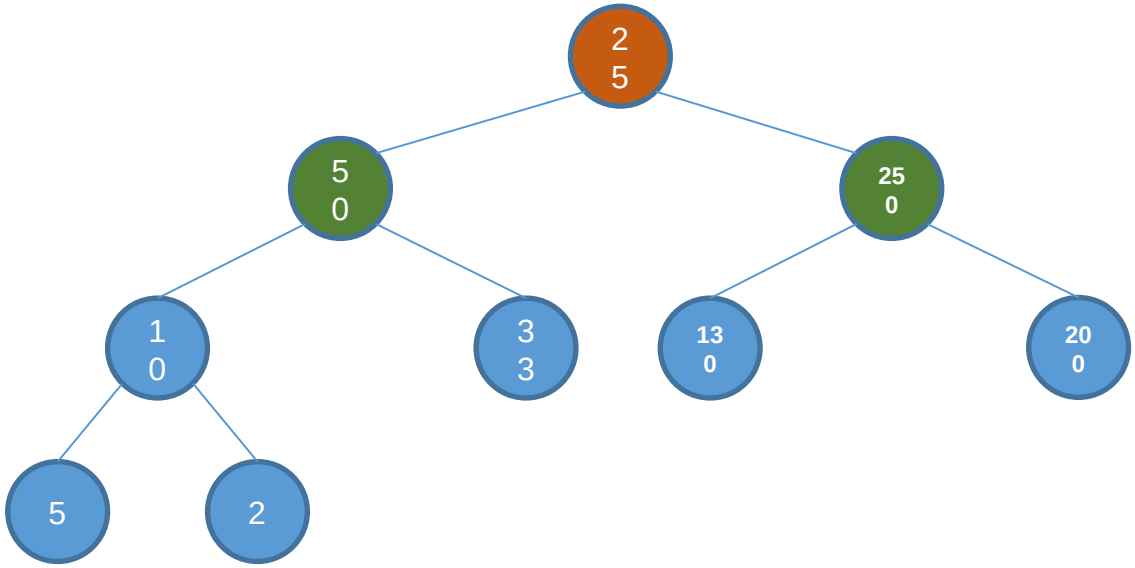
X	25	50	250	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

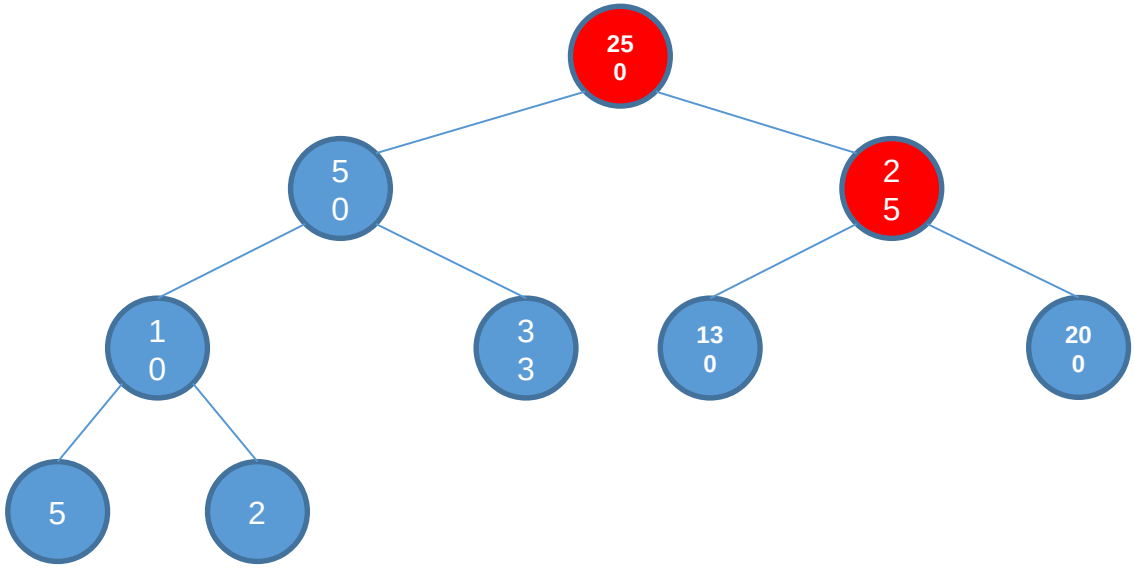
X	25	50	250	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

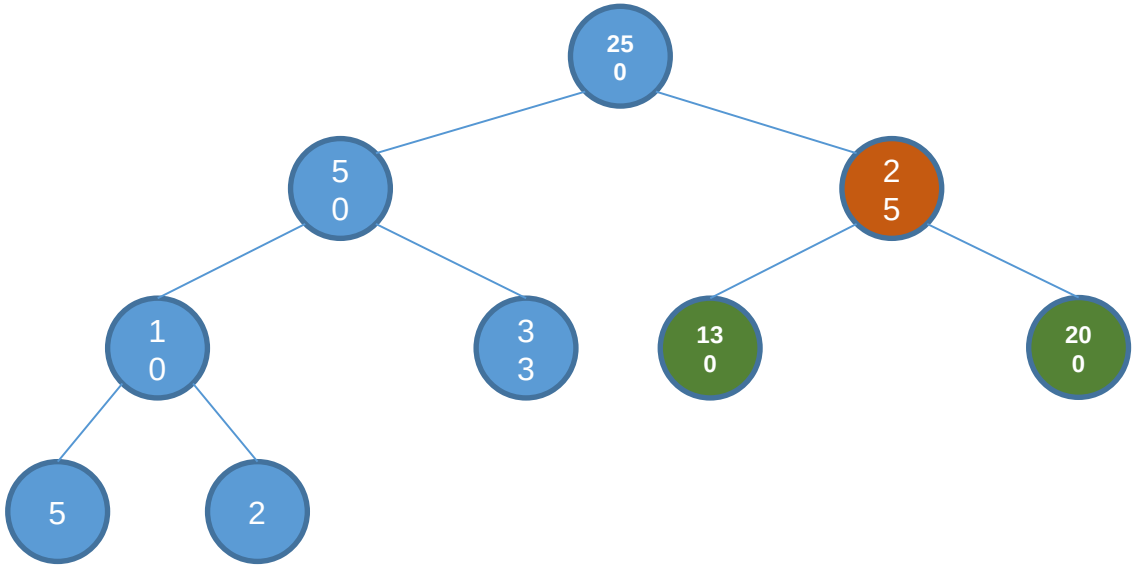
X	250	50	25	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

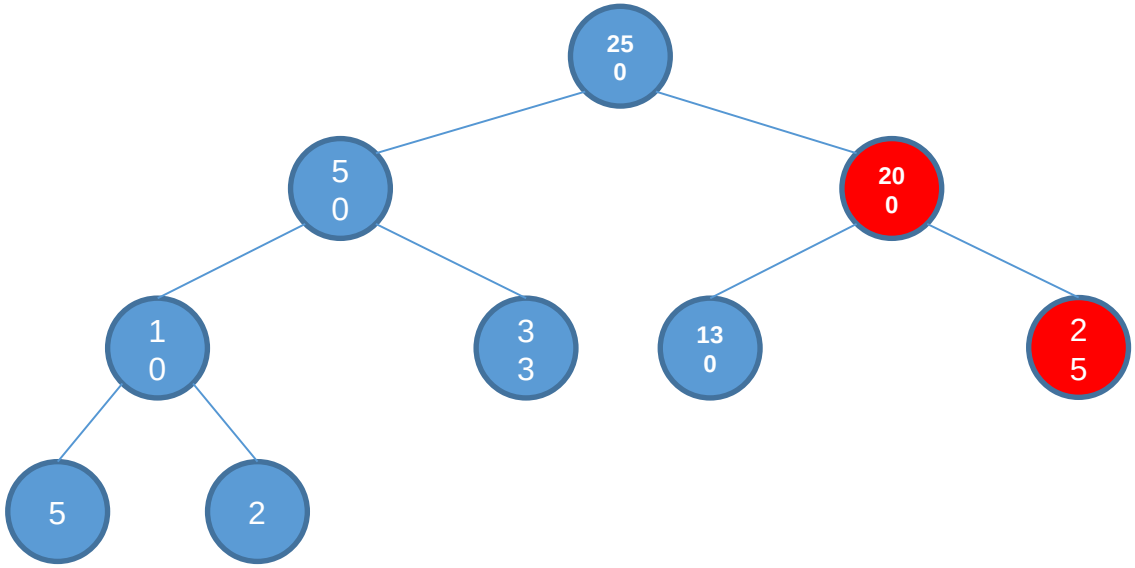
X	250	50	25	10	33	130	200	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

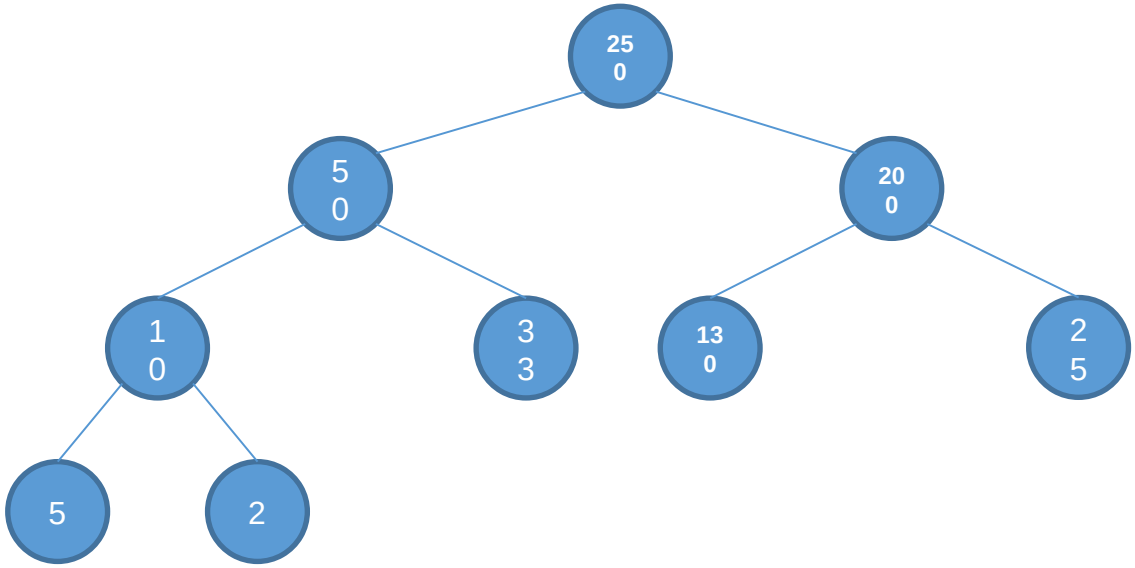
X	250	50	200	10	33	130	25	5	2	25		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

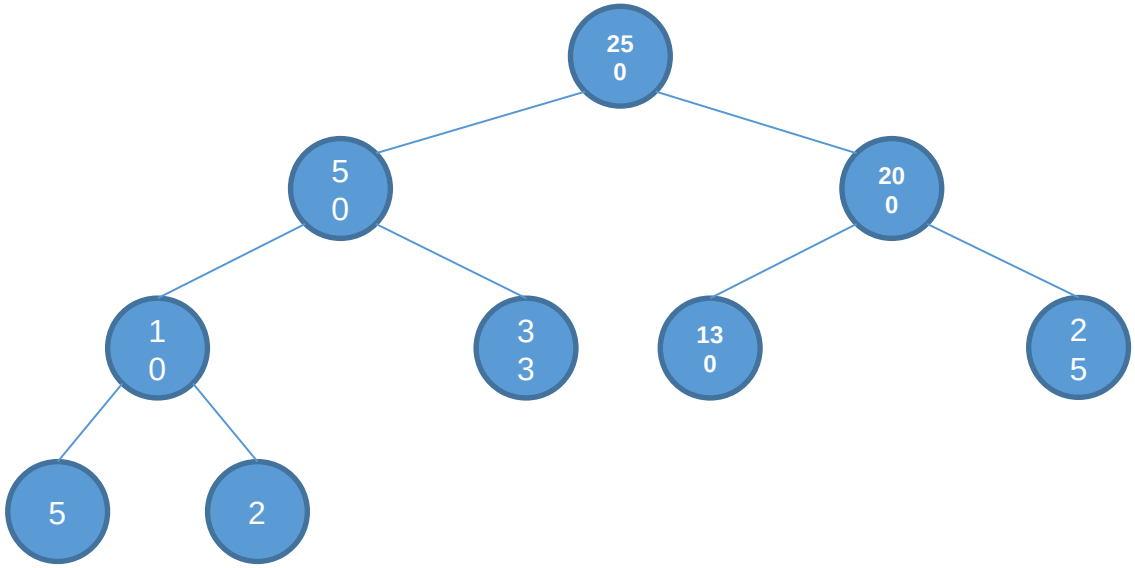
X	250	50	200	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 600

X	250	50	200	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

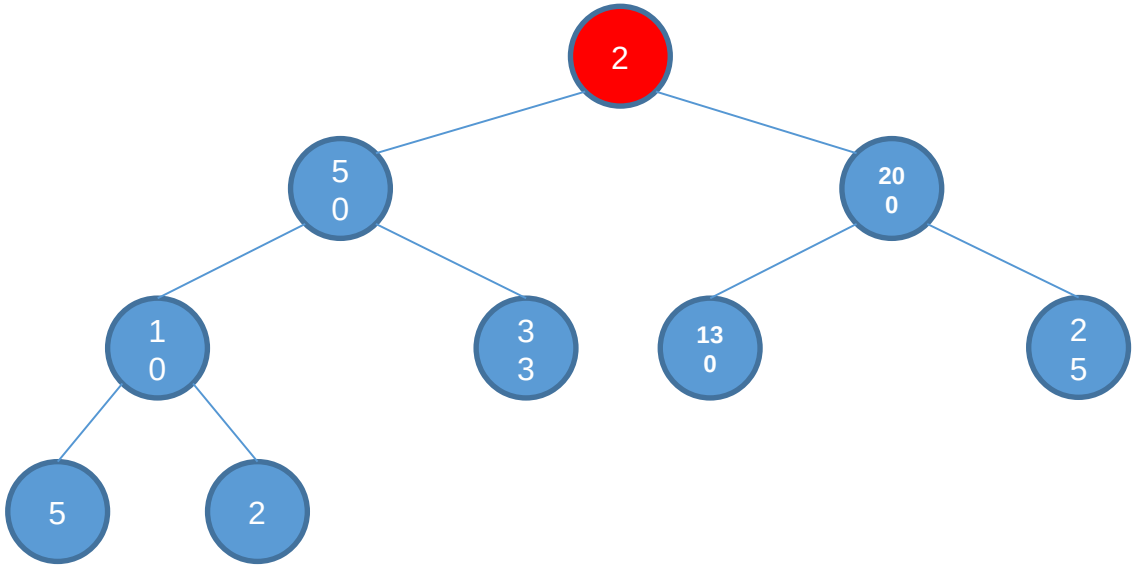



```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9

tempKey = 250

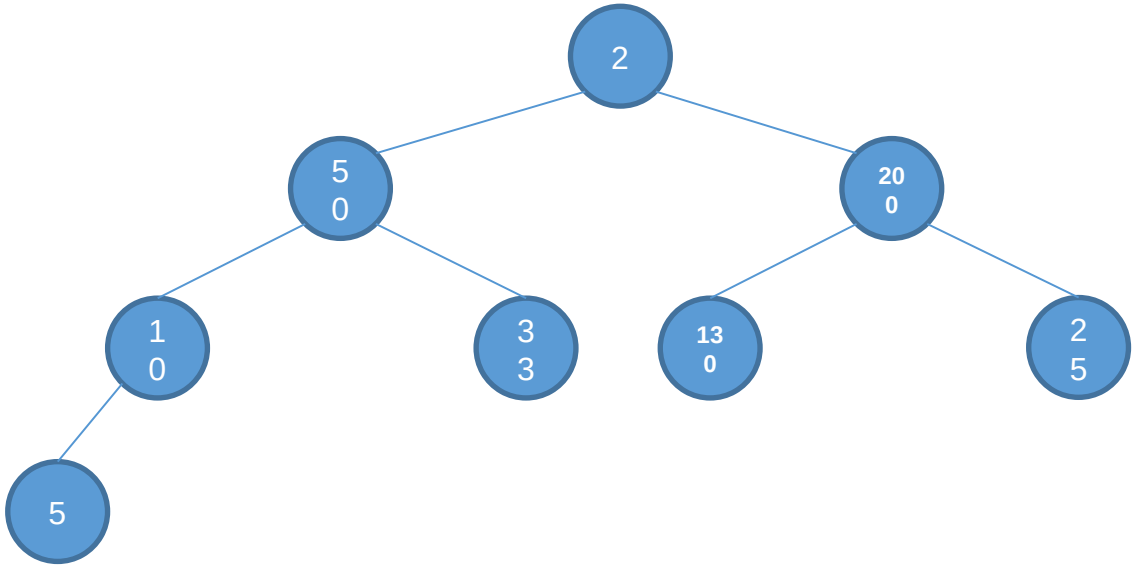
X	2	50	200	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 9
tempKey = 250

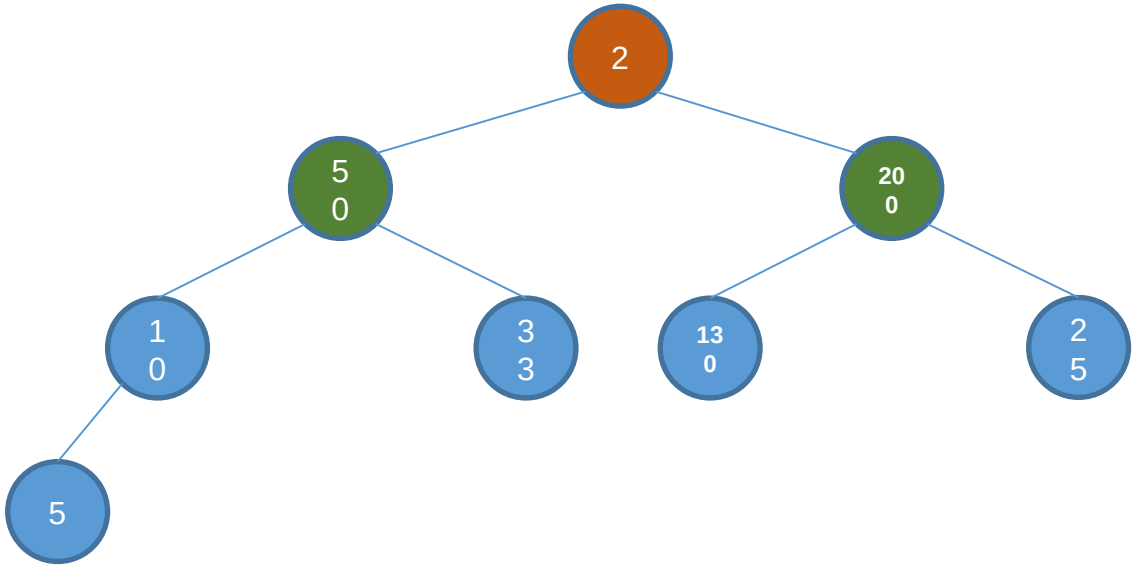
X	2	50	200	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

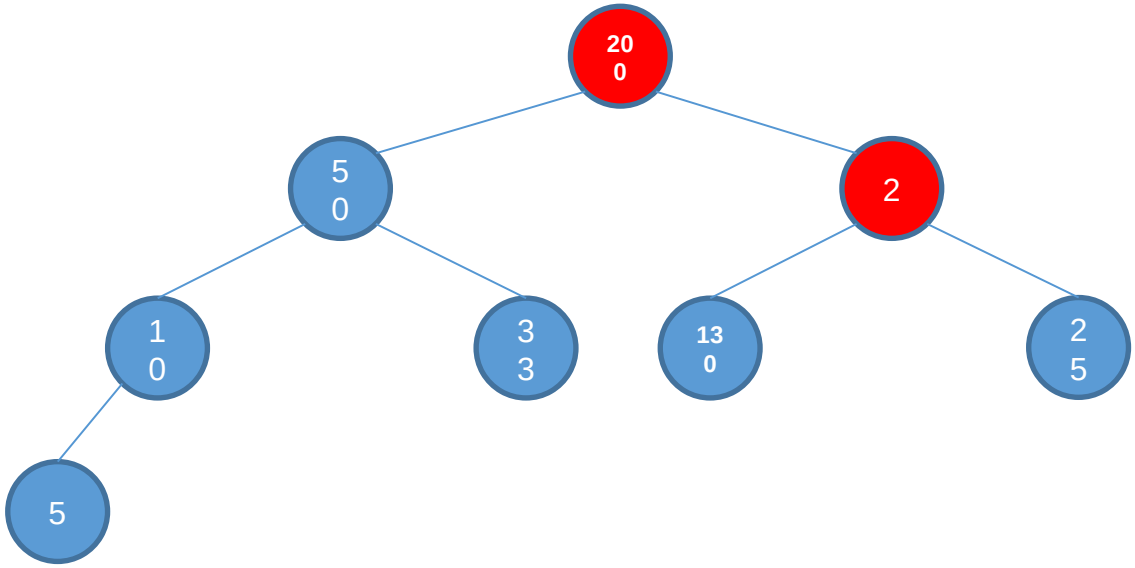
X	2	50	200	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

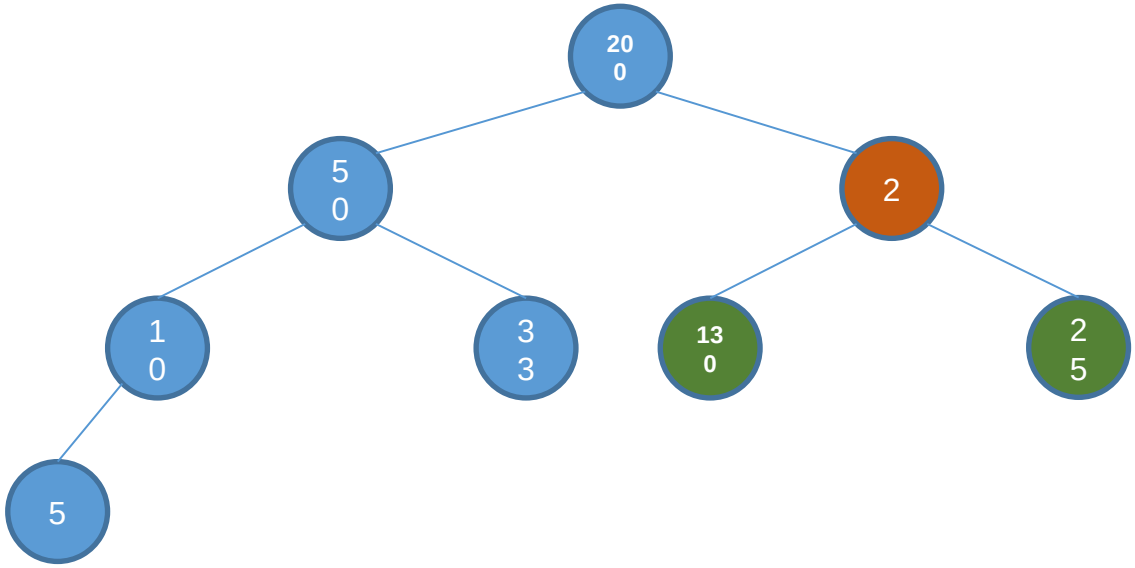
X	200	50	2	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

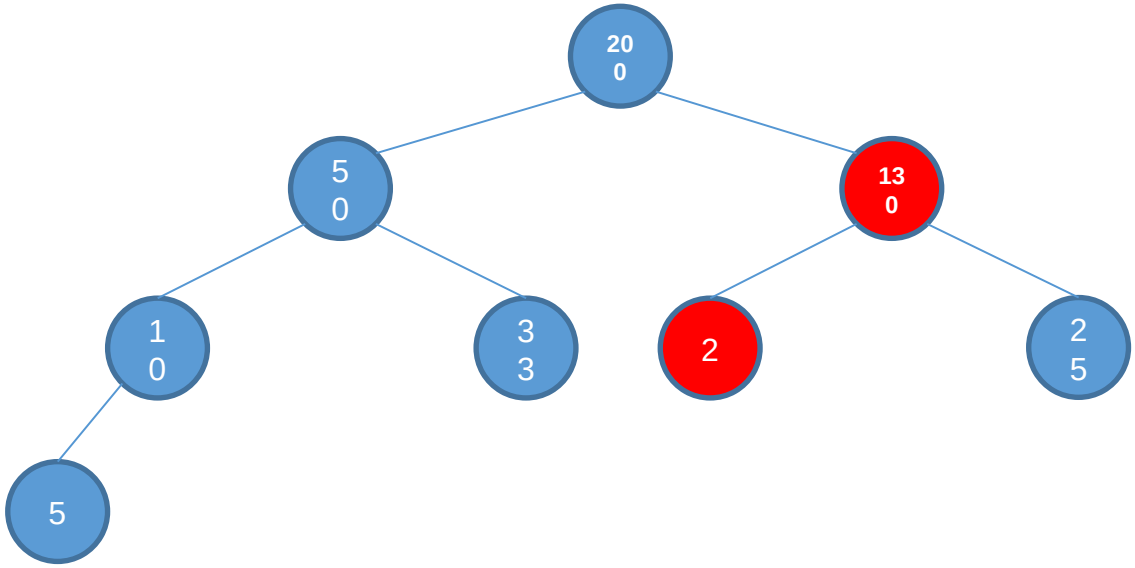
X	200	50	2	10	33	130	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

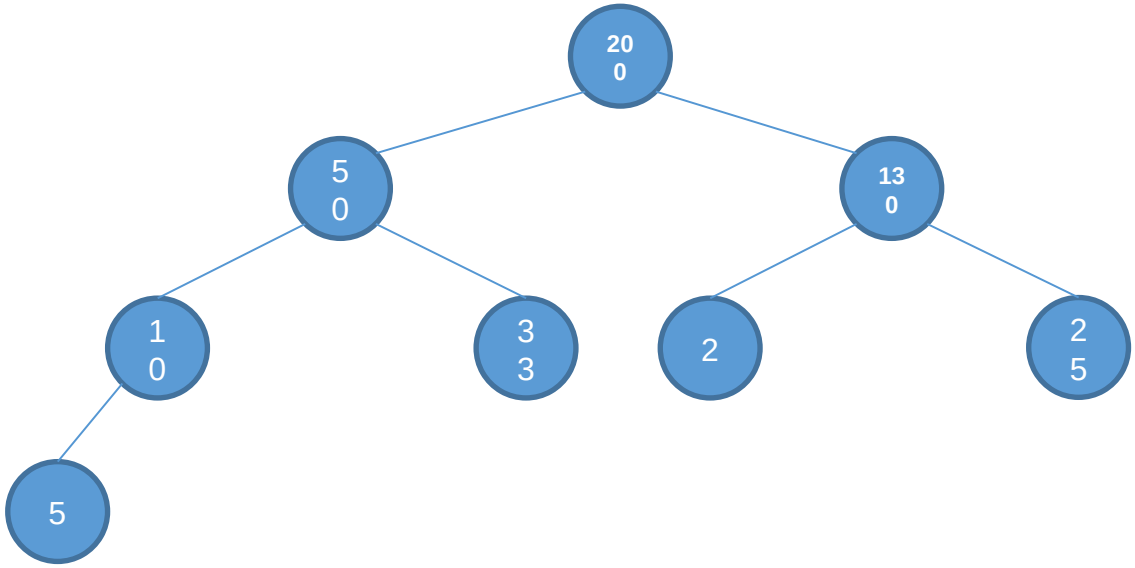
X	200	50	130	10	33	2	25	5	2	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

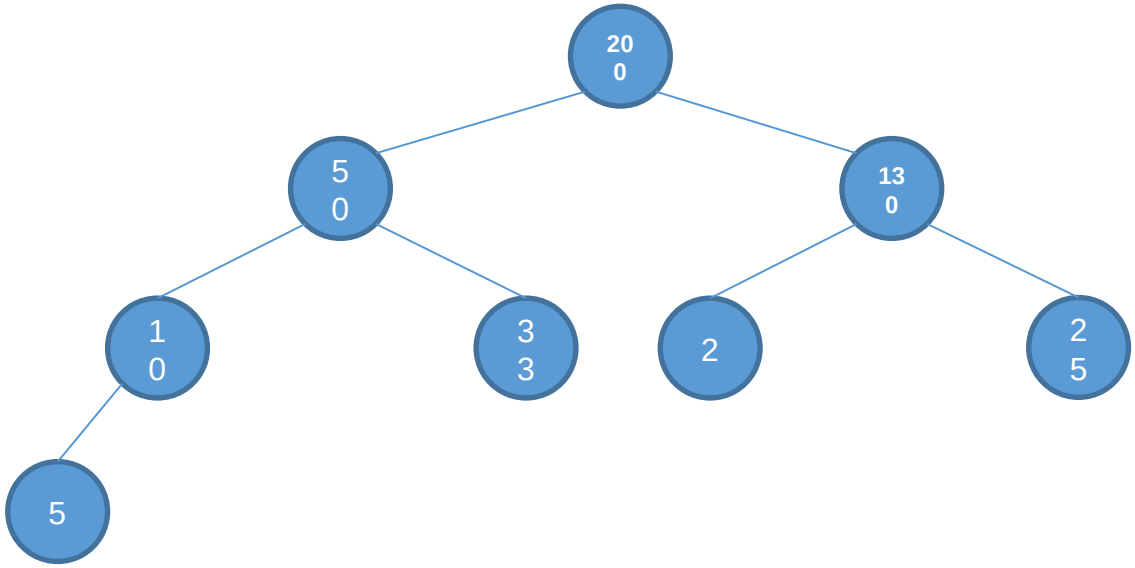
X	200	50	130	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 250

X	200	50	130	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

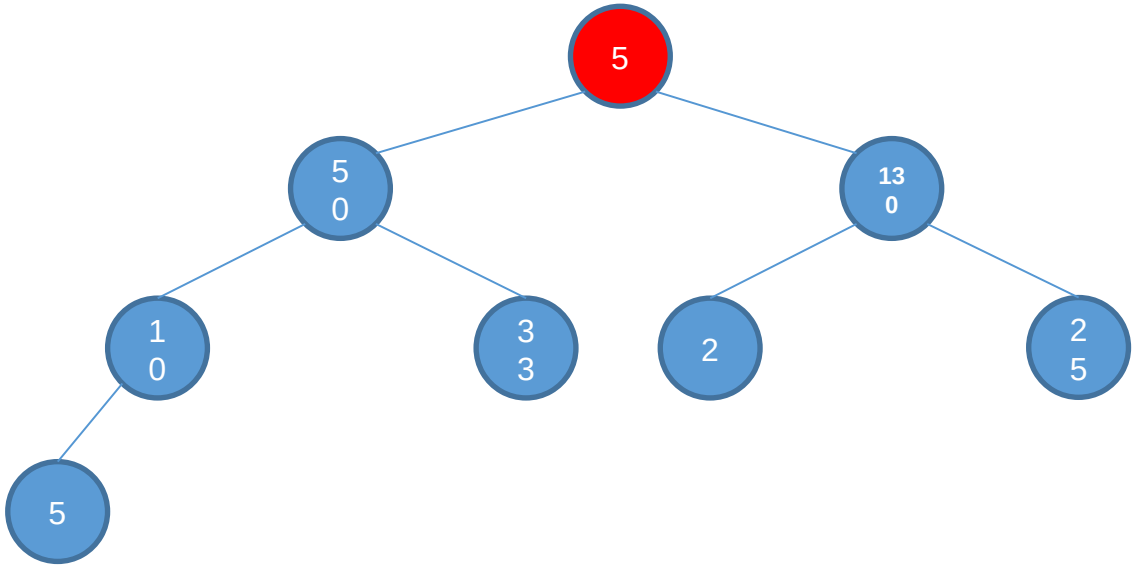



```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8

tempKey = 200

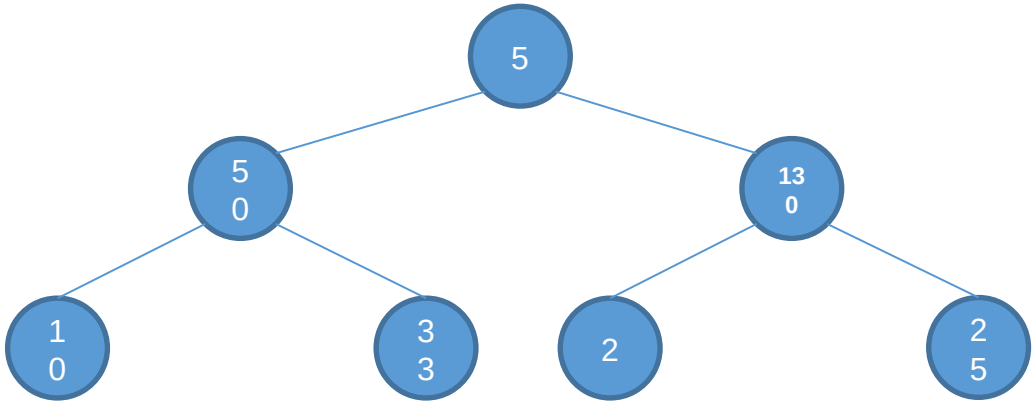
X	5	50	130	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 8
tempKey = 200

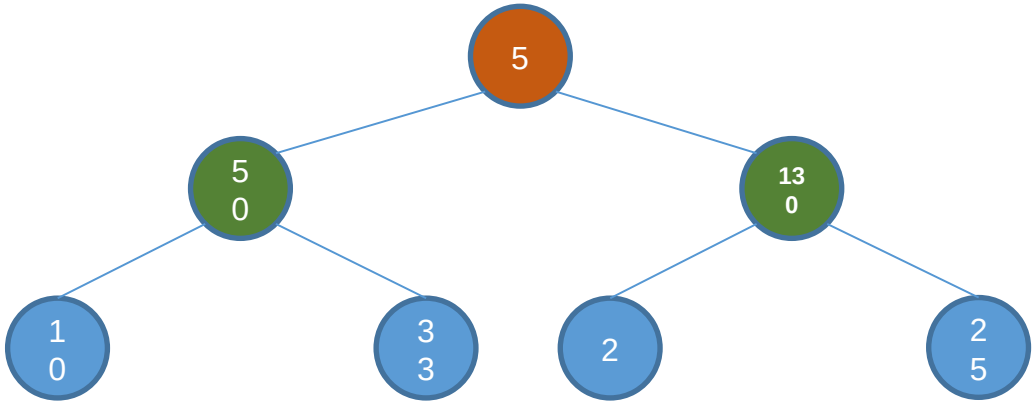
X	5	50	130	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

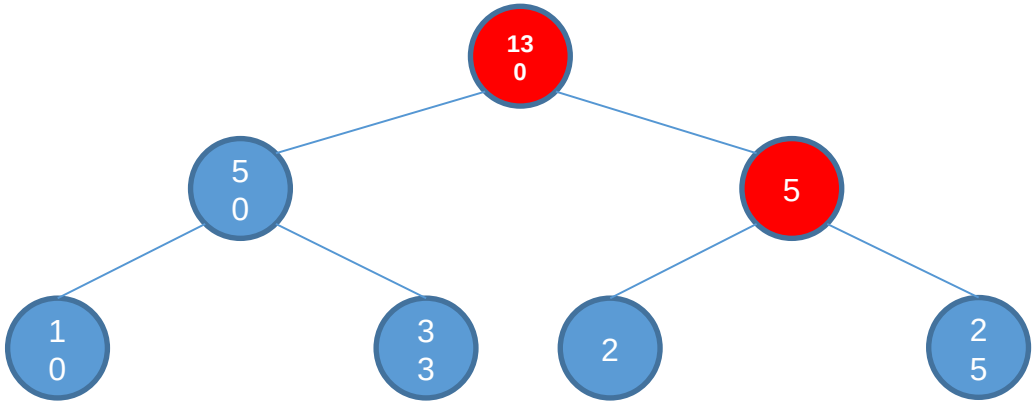
X	5	50	130	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

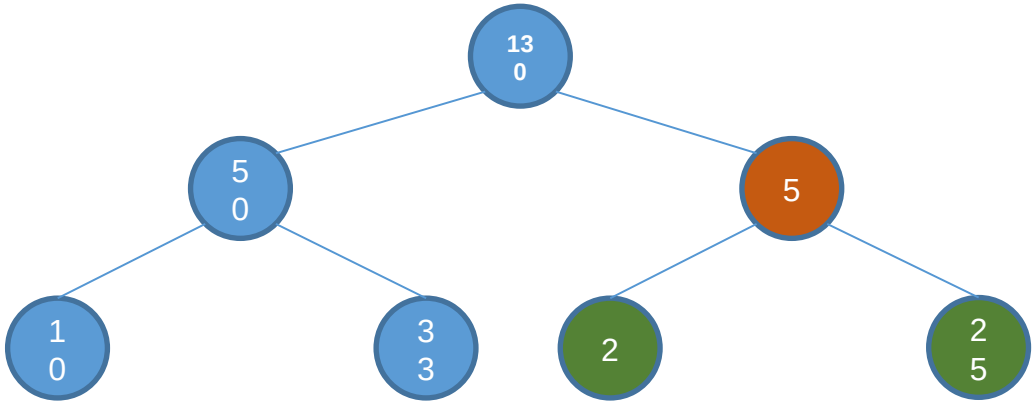
X	130	50	5	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

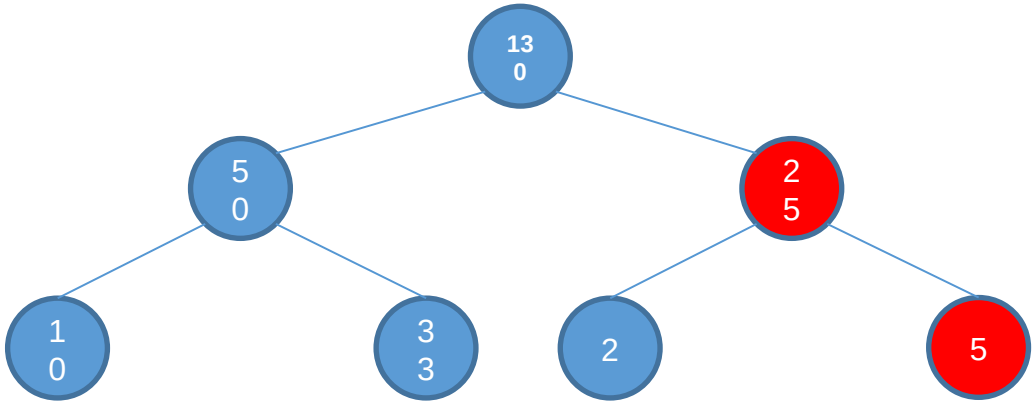
X	130	50	5	10	33	2	25	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

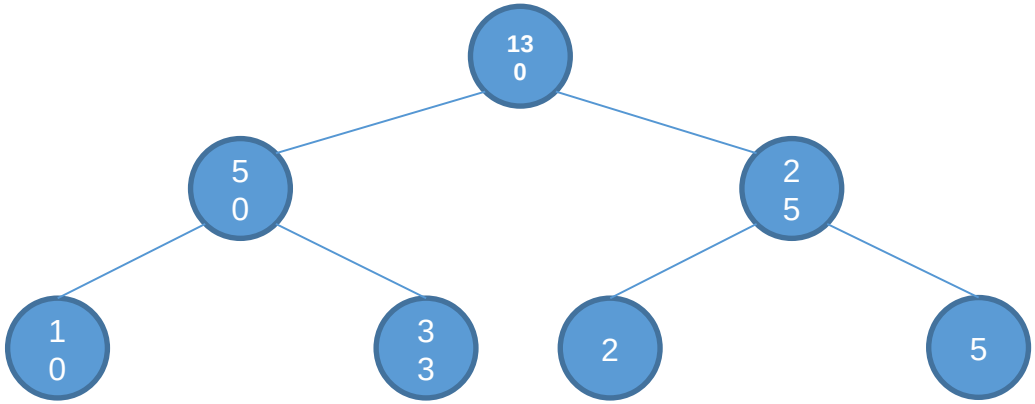
X	130	50	25	10	33	2	5	5	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

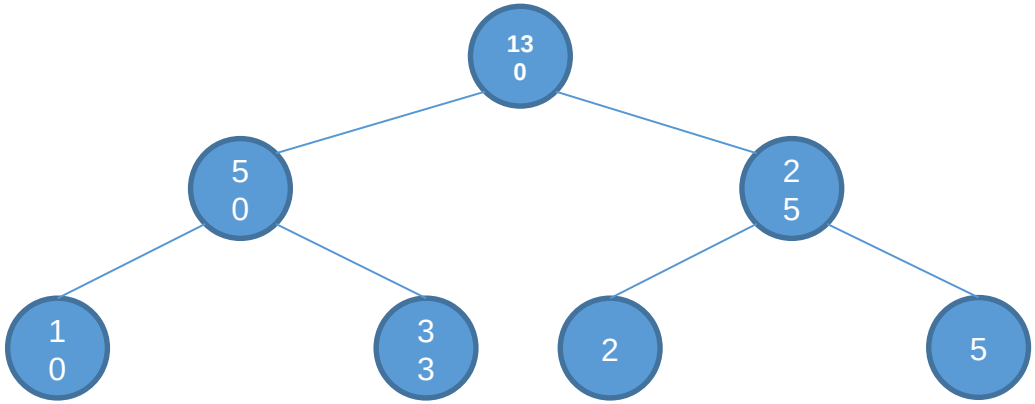
X	130	50	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 200

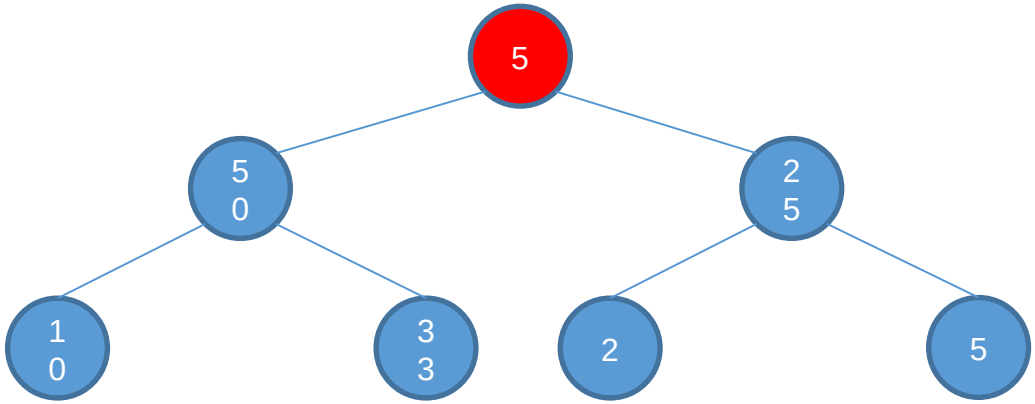
X	130	50	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 130

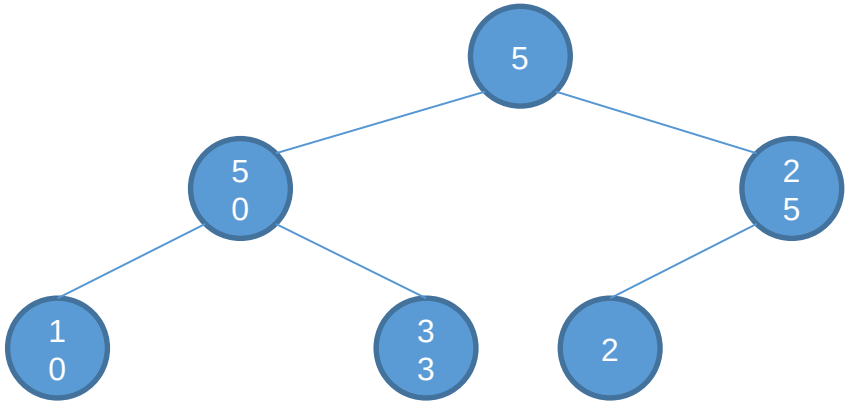
X	5	50	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 7
tempKey = 130

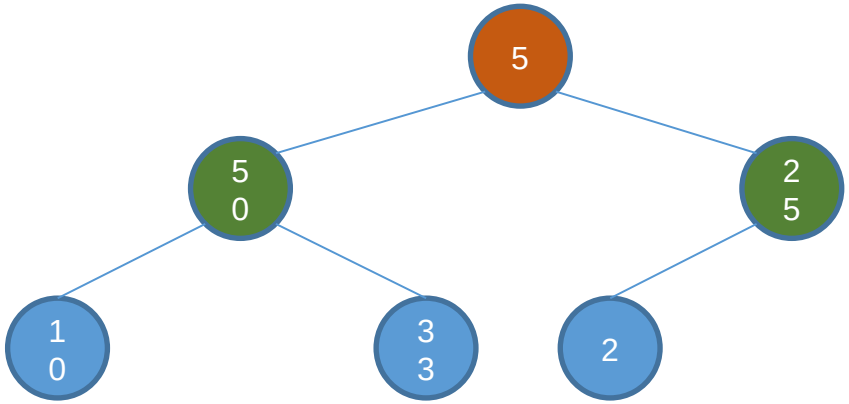
X	5	50	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

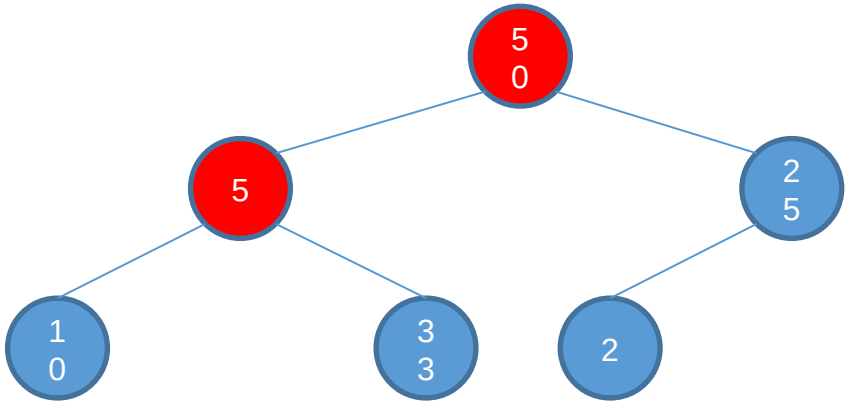
X	5	50	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

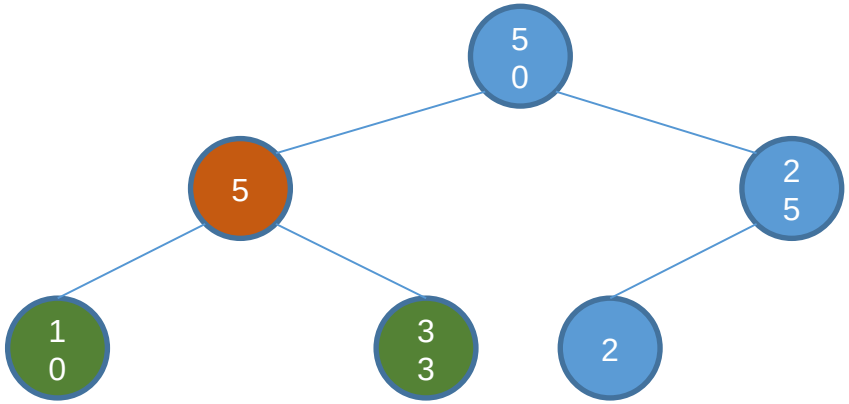
X	50	5	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

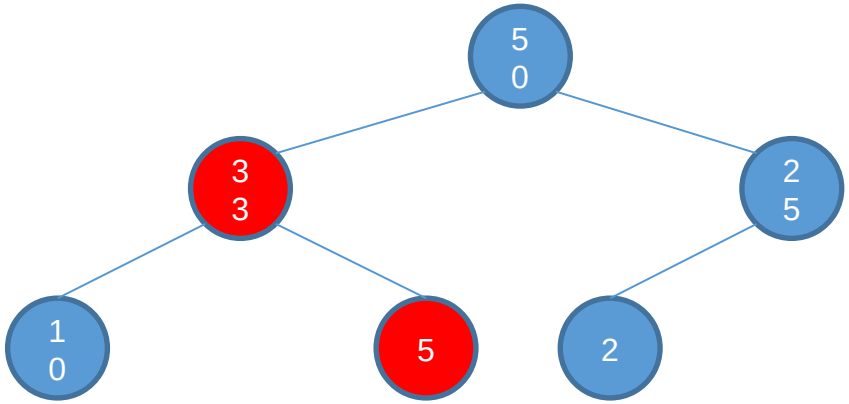
X	50	5	25	10	33	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

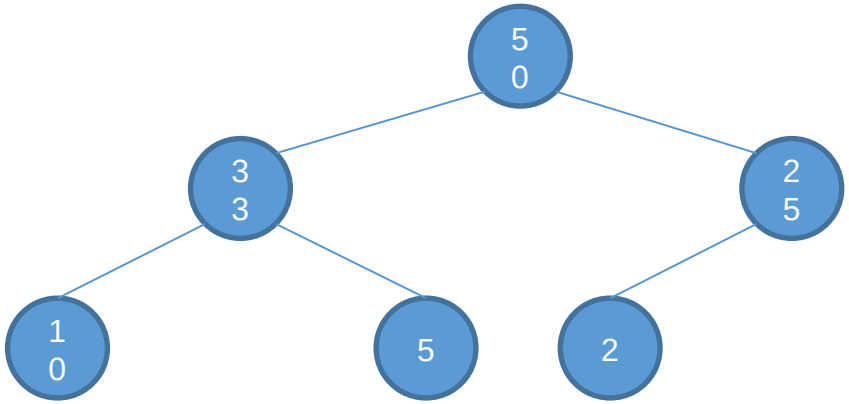
X	50	33	25	10	5	2	5	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

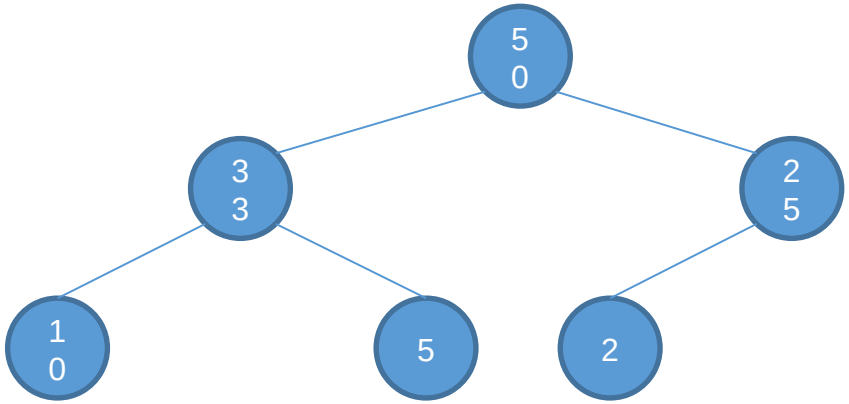
X	50	33	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 130

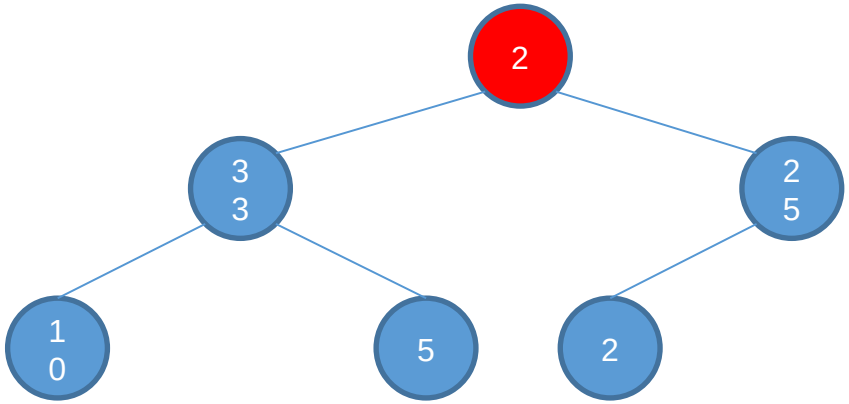
X	50	33	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 50

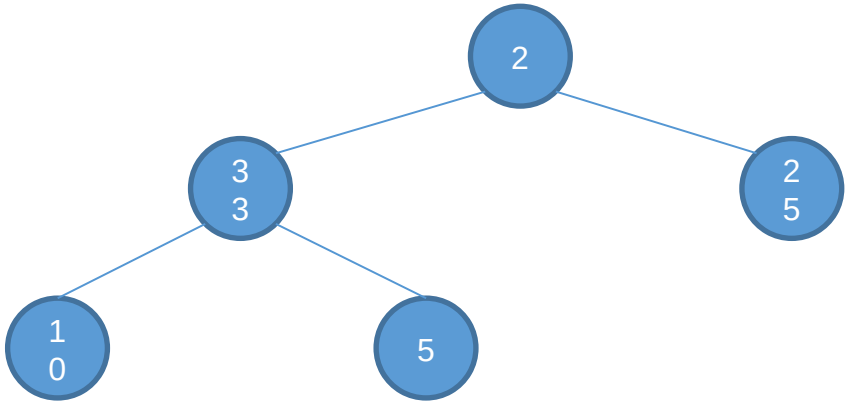
X	2	33	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 6
tempKey = 50

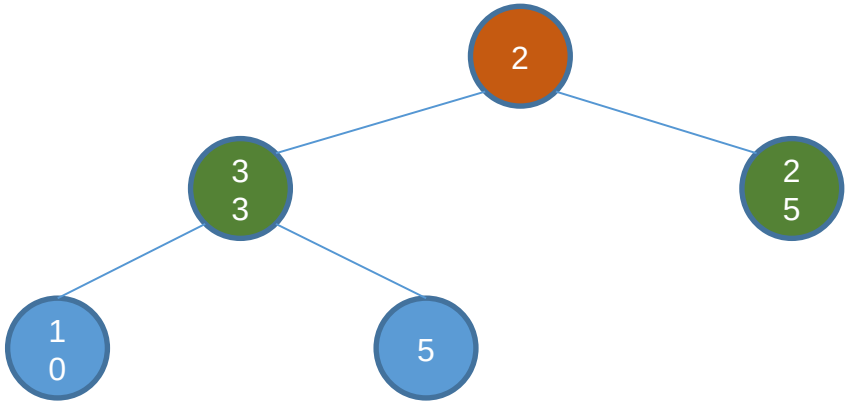
X	2	33	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

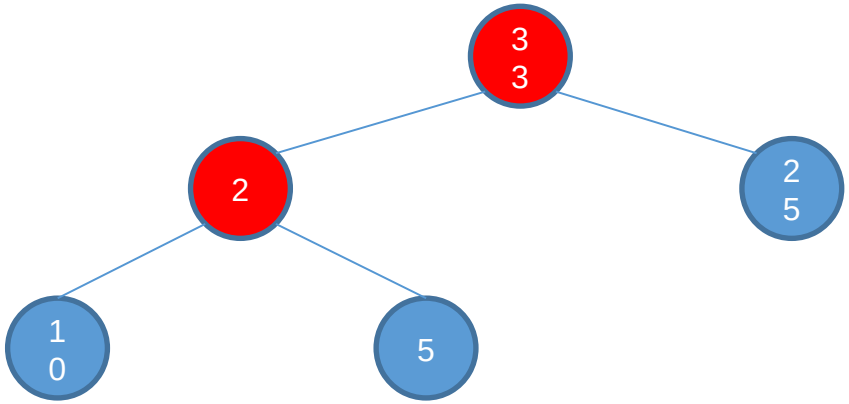
X	2	33	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

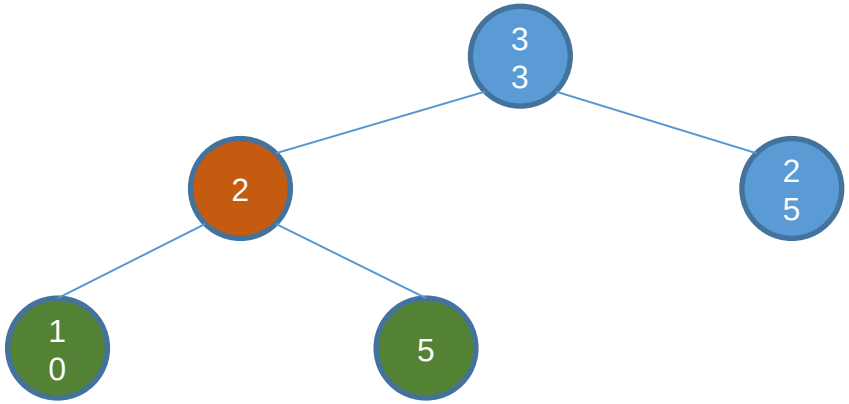
X	33	2	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

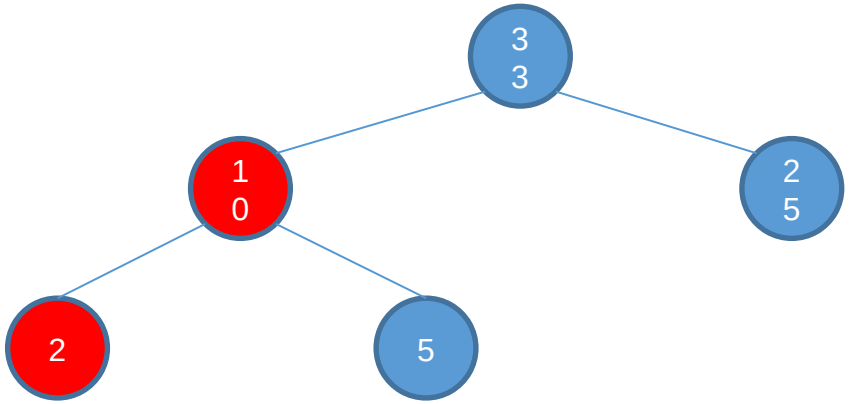
X	33	2	25	10	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

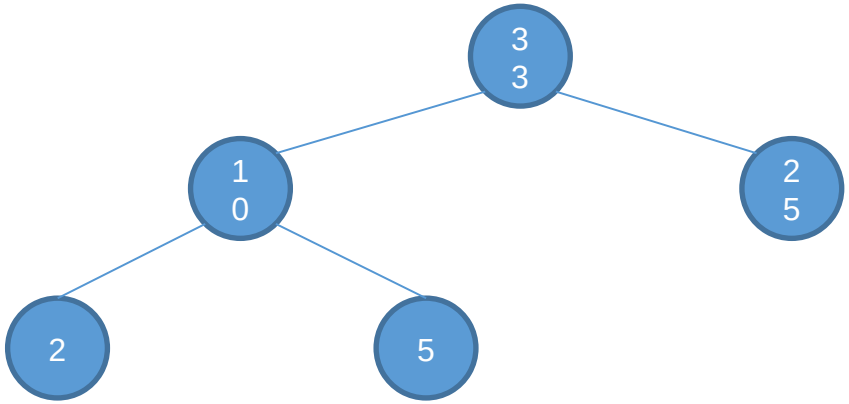
X	33	10	25	2	5	2	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

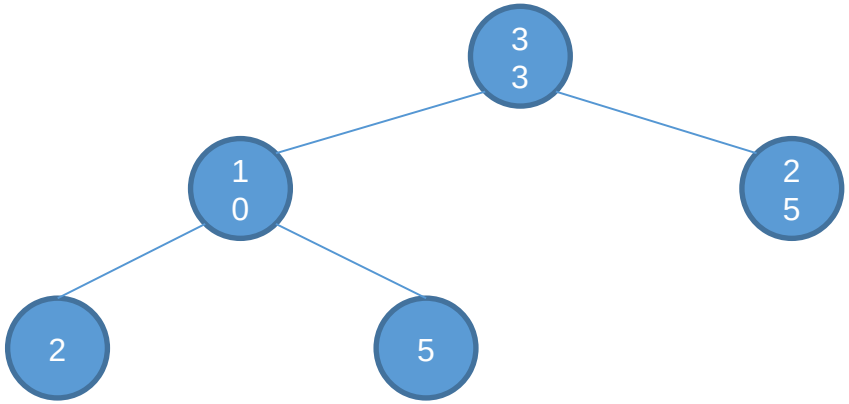
X	33	10	25	2	5	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 50

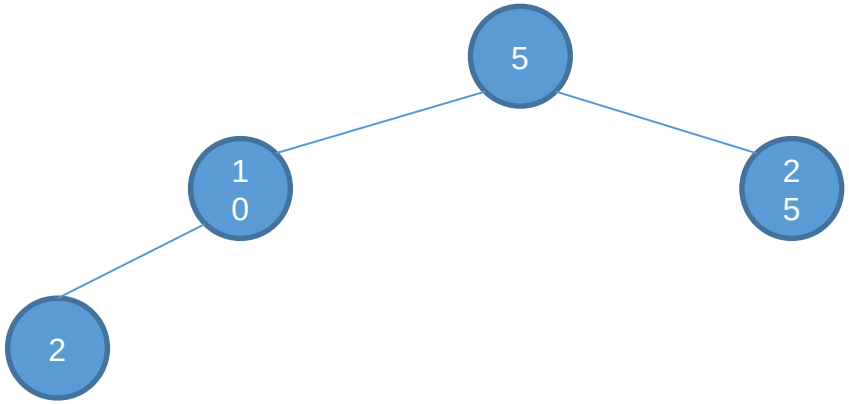
X	33	10	25	2	5	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 5
tempKey = 33

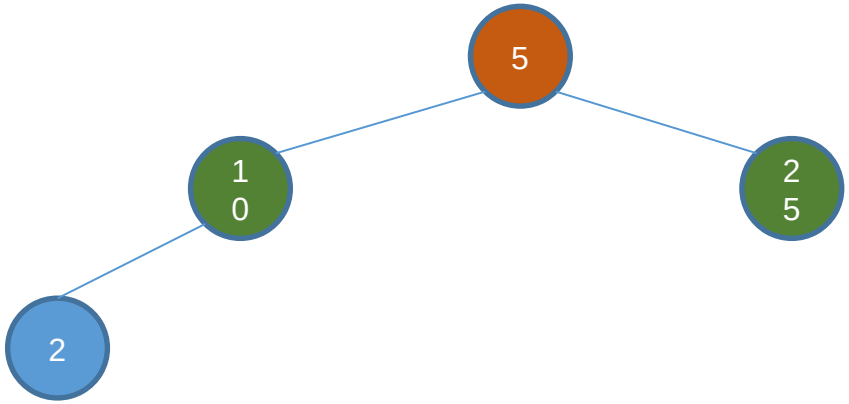
X	5	10	25	2	5	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 33

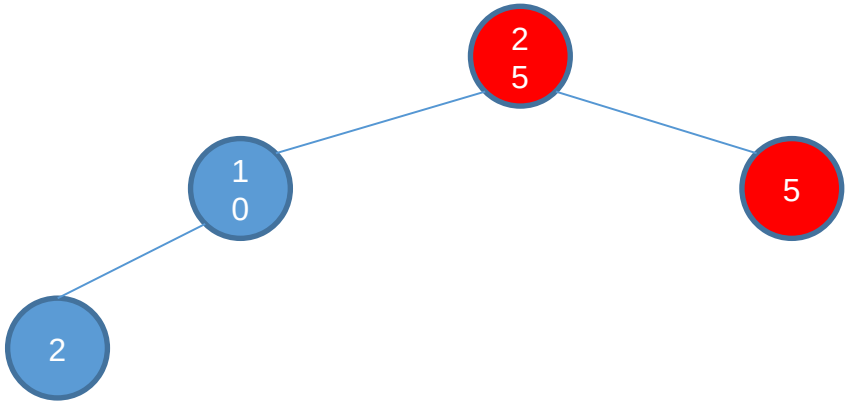
X	5	10	25	2	5	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 33

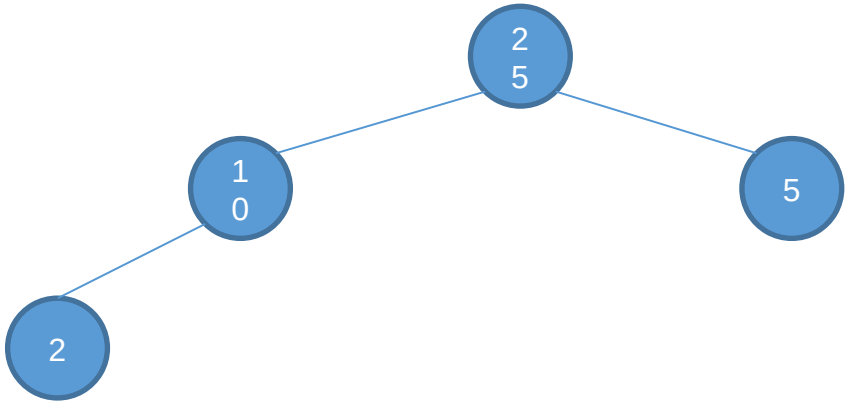
X	25	10	5	2	5	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 33

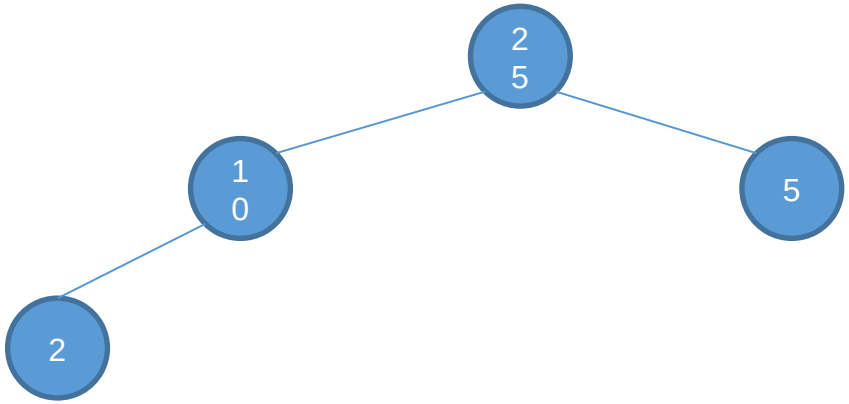
X	25	10	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 33

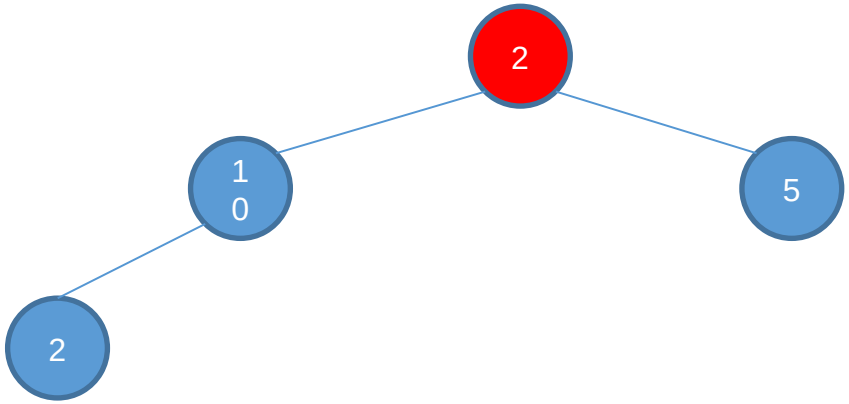
X	25	10	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 25

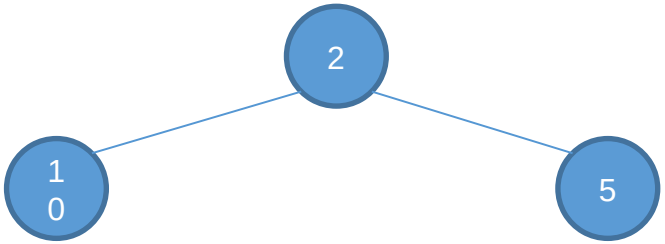
X	2	10	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 4
tempKey = 25

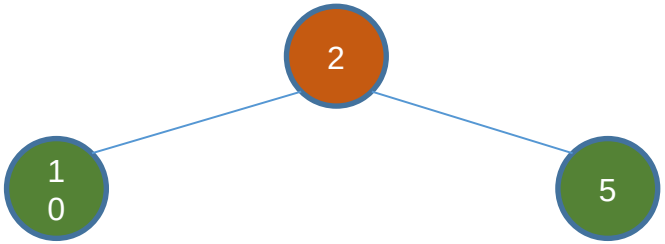
X	2	10	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 25

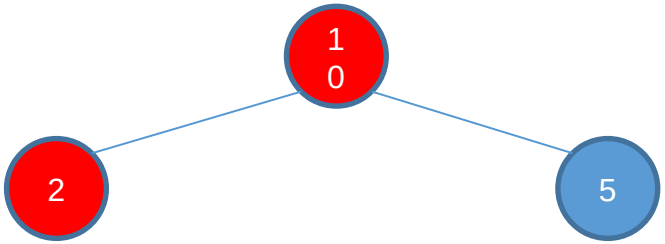
X	2	10	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 25

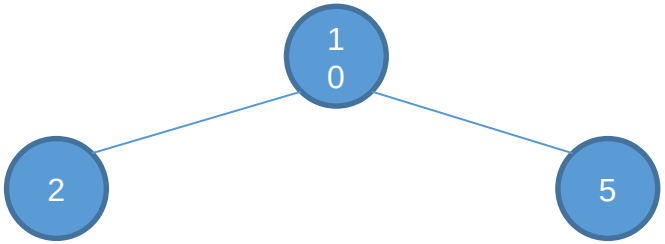
X	10	2	5	2	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 25

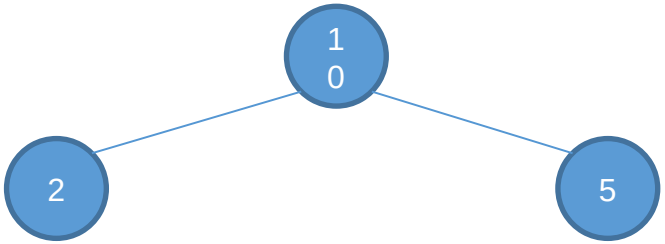
X	10	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 25

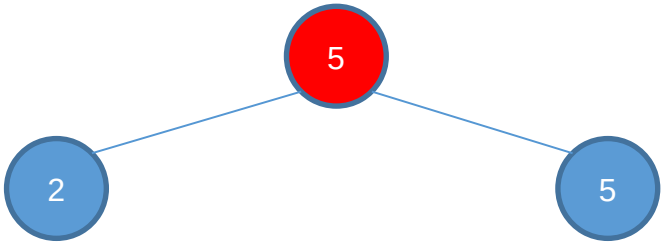
X	10	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 10

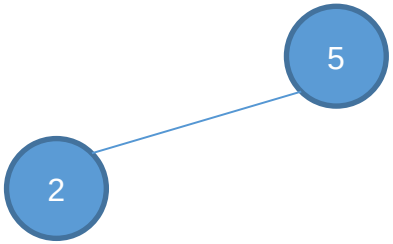
X	5	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 3
tempKey = 10

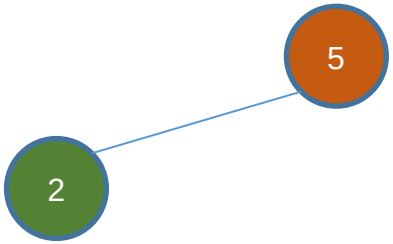
X	5	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 10

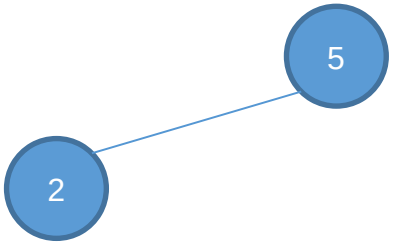
X	5	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 10

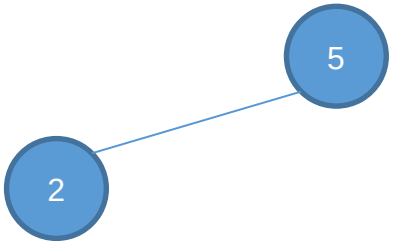
X	5	2	5	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 10

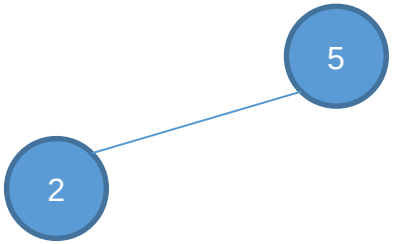
X	5	2	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 10

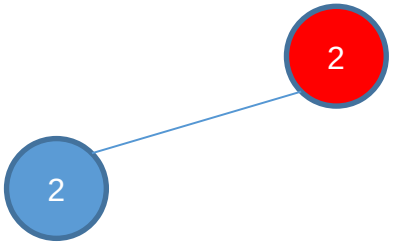
X	5	2	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 5

X	2	2	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12




```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 2
tempKey = 5

X	2	2	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

2

```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 1
tempKey = 5

X	2	2	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

2

```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 1
tempKey = 5

X	2	5	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

2

```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 1
tempKey = 5

X	2	5	10	25	33	50	130	200	250	600		
0	1	2	3	4	5	6	7	8	9	10	11	12

2

```
public void sort(){
        int n= size;
        for(int i= 1; i < n; i++){
            int tmpKey= keys[1];
            T tmpData= data[1];
            keys[1]= keys[size];
            data[1]= data[size];
            size--;
            siftDown(1);
            keys[size+1]= tmpKey;
            data[size+1]= tmpData;
        }
    }
}
```

Size = 1
tempKey = 5

Time complexity

	Running time
siftUp (upHeap)	O(log n)
siftDown (downHeap)	O(log n)
enqueue in heap priority queue	O(log n)
serve() in heap priority queue	O(log n)
Bottom-up construction of a heap	O(n)
Heap sort	O(n log n)