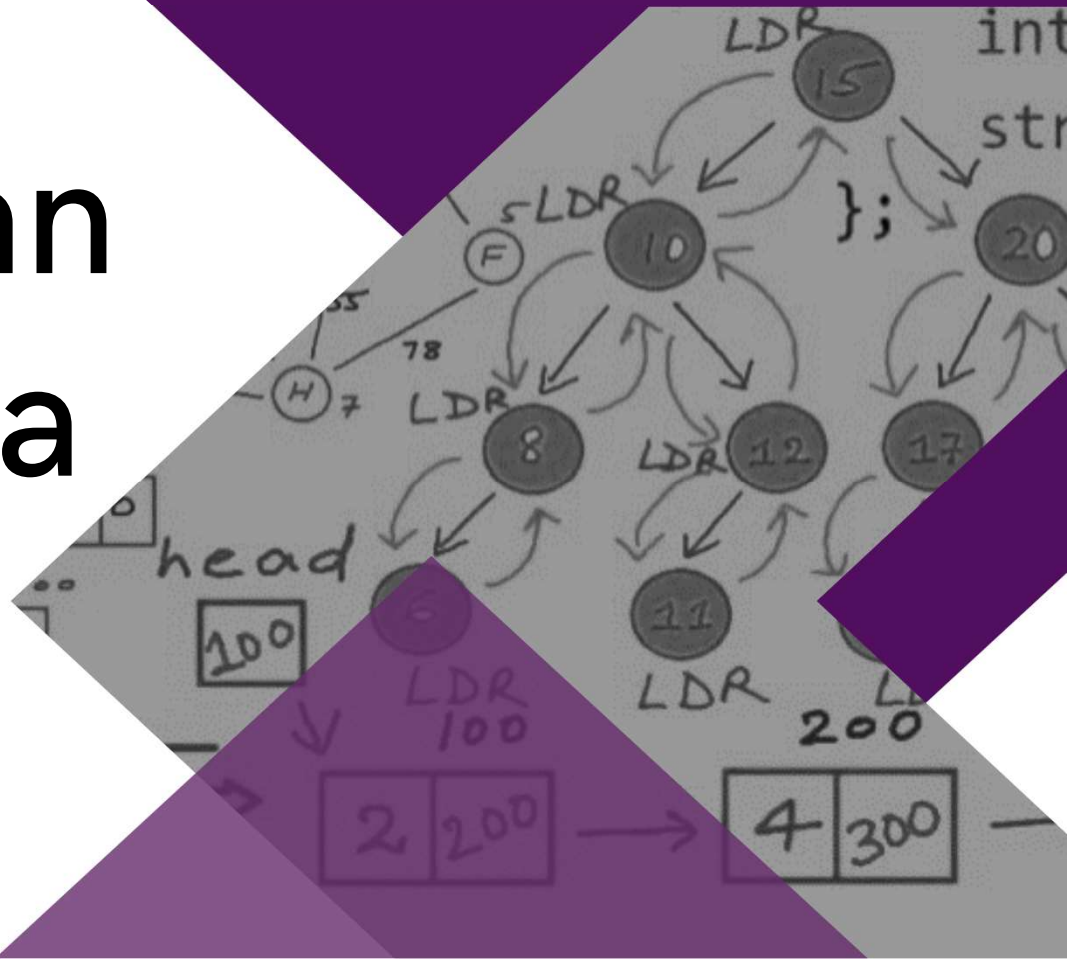
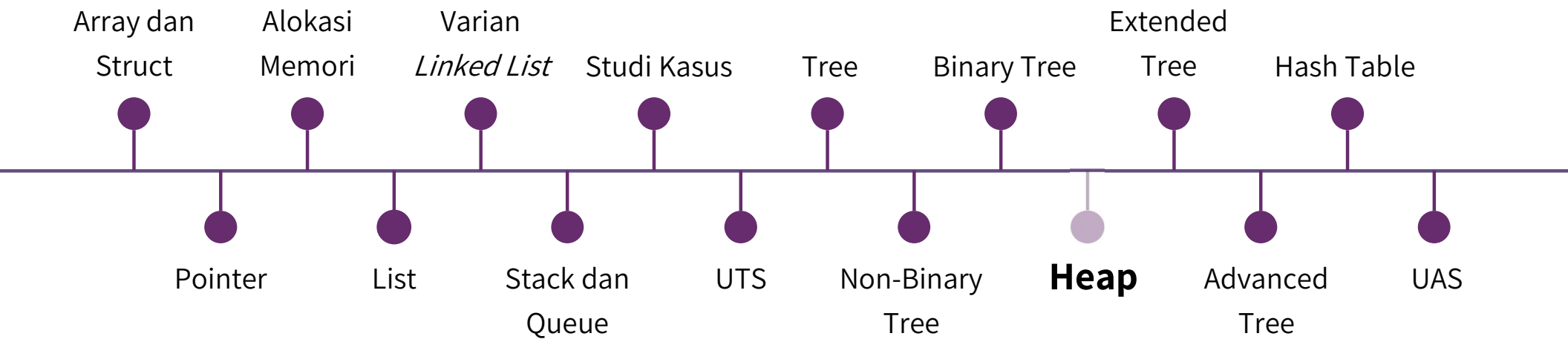
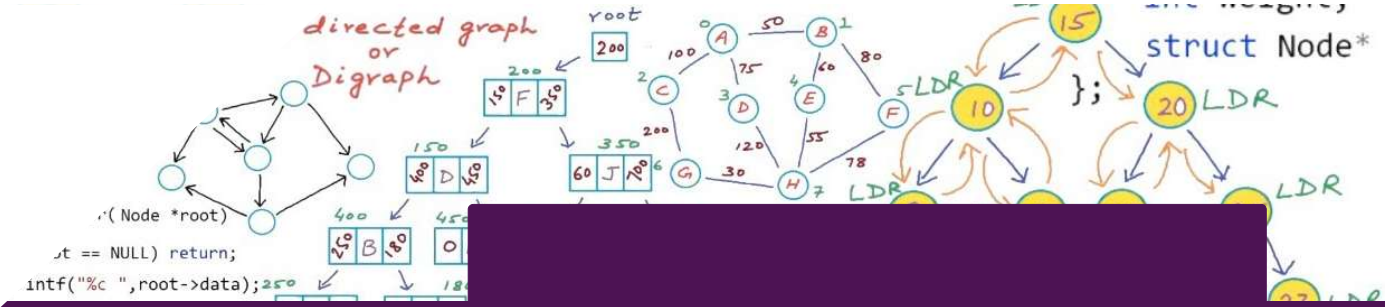
Algoritma dan Struktur Data



Pekan 12


Array dan Struct
Alokasi Memori
Varian *Linked List*
Studi Kasus
Tree
Binary Tree
Extended Tree
Hash Table
Pointer
List
Stack dan Queue
UTS
Non-Binary Tree
Heap
Advanced Tree
UAS

Tujuan



1	Mahasiswa memahami konsep priority tree
2	Mahasiswa mampu membedakan queue dan priority queue
3	Mahasiswa mampu mengimplementasikan ADT Priority queue

Priority Queue

Priority Queue

Implementation Level

Unsorted List	
Array-Based Sorted List	
Linked Sorted List	
Binary Search Tree	

Heap

Heap

A **complete** binary tree, each of whose elements contains a value that is greater than or equal to the value of each of its children.

Heap here is a data structure, **not free store** (the area of memory available for dynamically allocated data)

Heap Properties

- A **complete** binary tree
- The value stored in a heap are **partially ordered**

Heap Variants

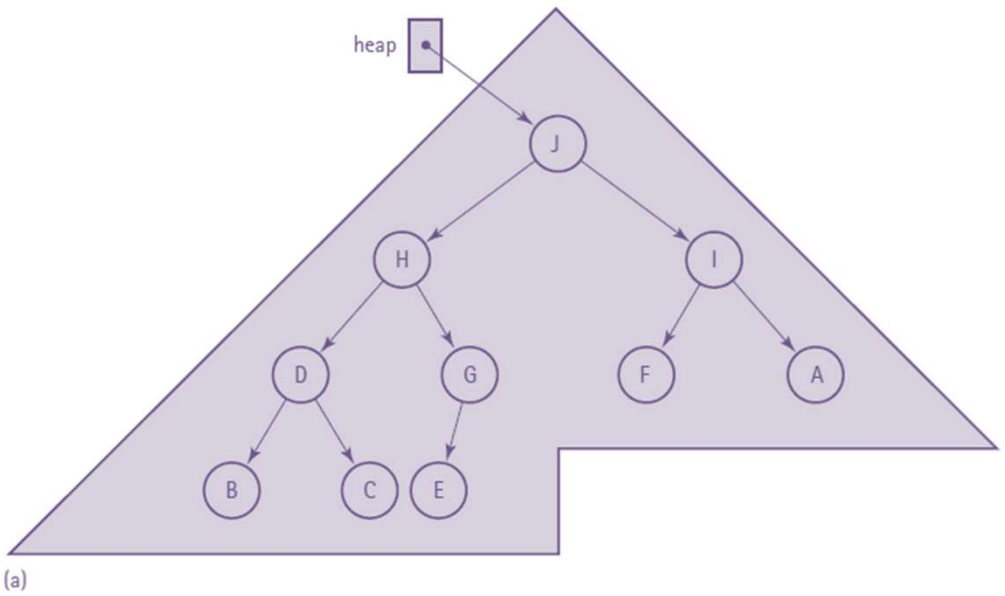
- **Max-heap**

  Every node stores a value that is greater than or equal to the value of either of its children.

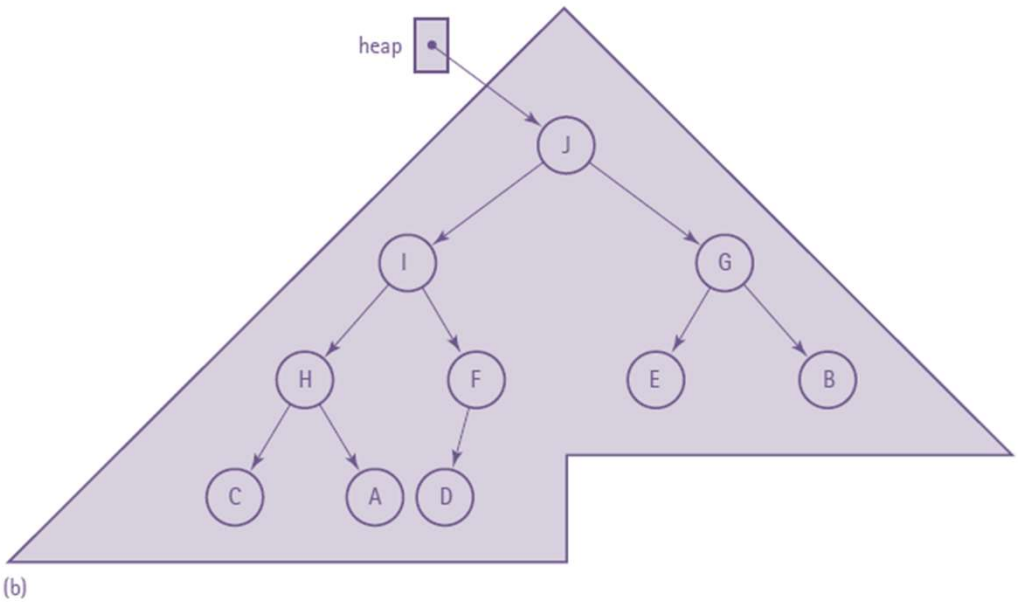
- **Min-heap**

  Every node stores a value that is less than or equal to that of its children.

Example



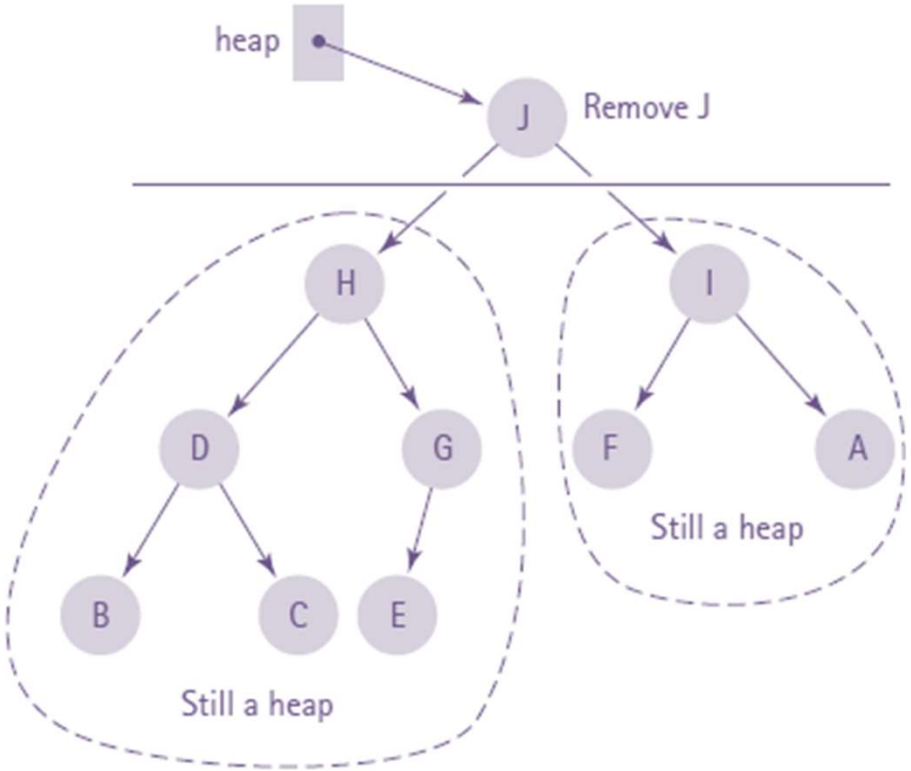
(a)



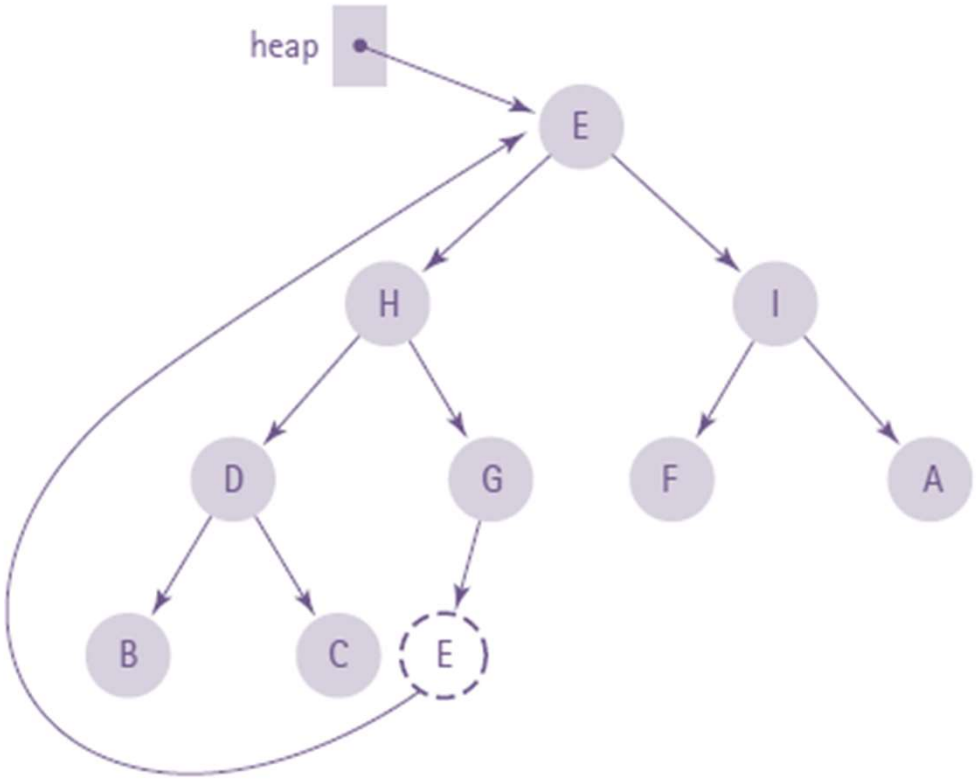
(b)

ReheapDown

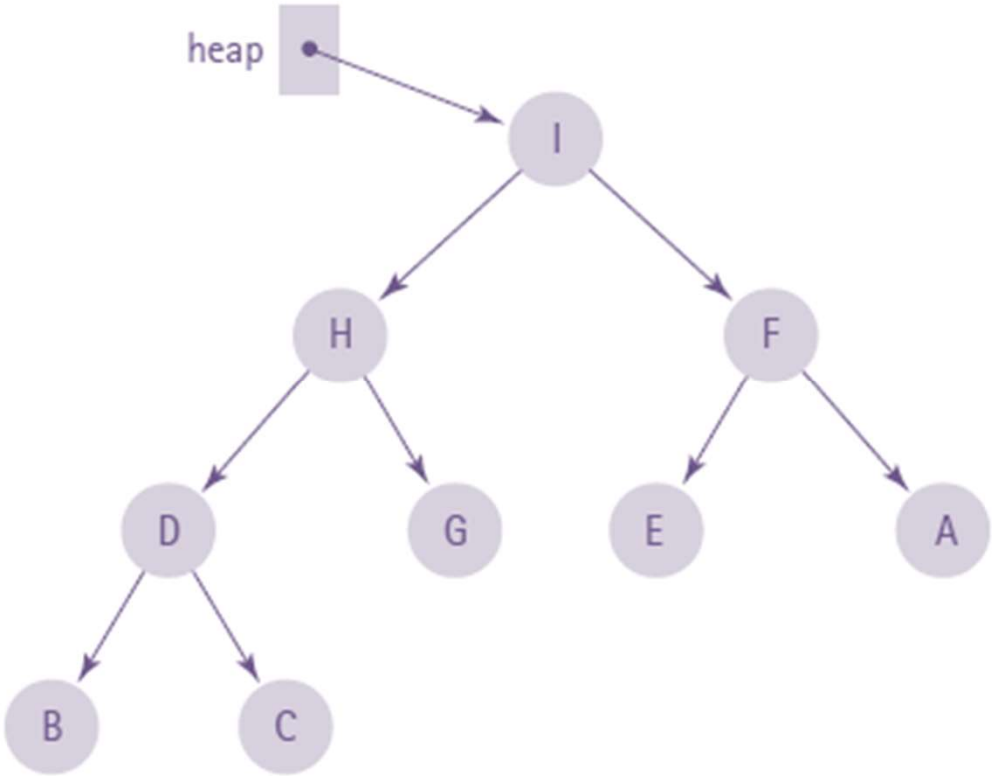
Example



Example

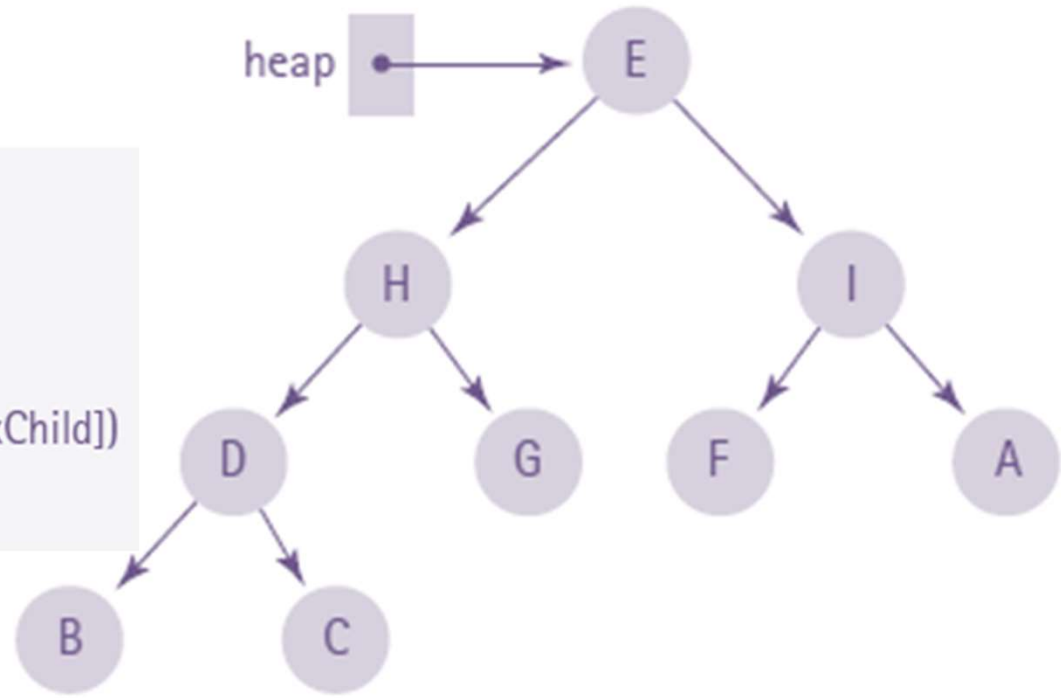


ReheapDown



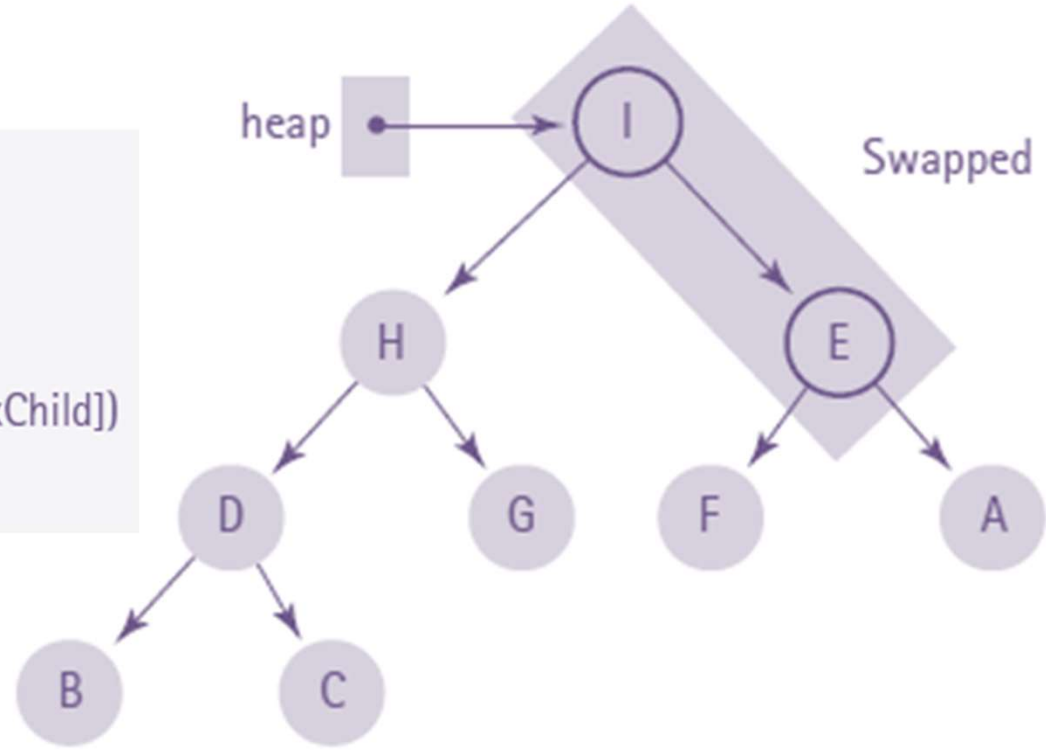
ReheapDown

```
ReheapDown(heap, root, bottom)
if heap.elements[root] is not a leaf
    Set maxChild to index of child with larger value
    if heap.elements[root] < heap.elements[maxChild]
        Swap(heap.elements[root], heap.elements[maxChild])
        ReheapDown(heap, maxChild, bottom)
```



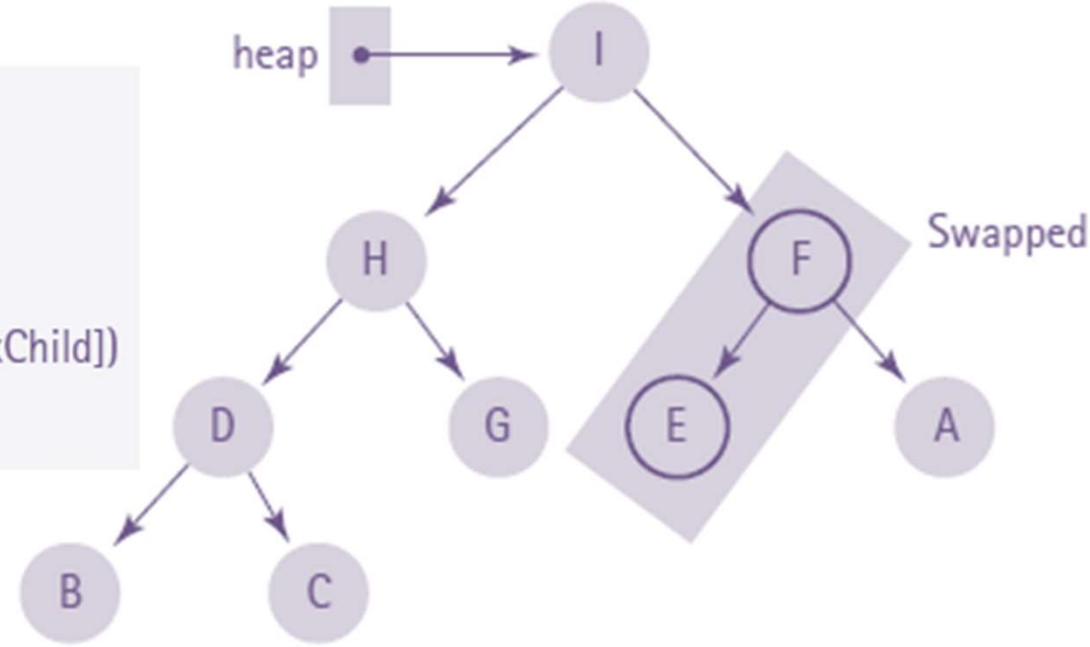
ReheapDown

```
ReheapDown(heap, root, bottom)
if heap.elements[root] is not a leaf
    Set maxChild to index of child with larger value
    if heap.elements[root] < heap.elements[maxChild]
        Swap(heap.elements[root], heap.elements[maxChild])
        ReheapDown(heap, maxChild, bottom)
```



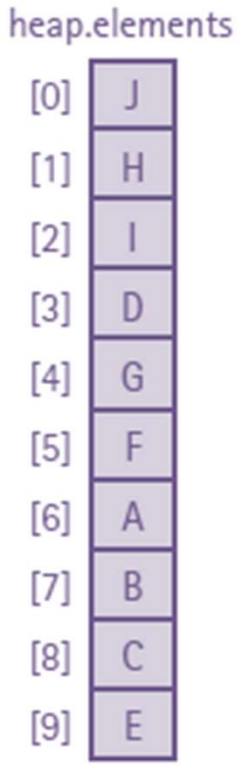
ReheapDown

```
ReheapDown(heap, root, bottom)
if heap.elements[root] is not a leaf
    Set maxChild to index of child with larger value
    if heap.elements[root] < heap.elements[maxChild]
        Swap(heap.elements[root], heap.elements[maxChild])
        ReheapDown(heap, maxChild, bottom)
```



ReheapDown

```
ReheapDown(heap, root, bottom)
if heap.elements[root] is not a leaf
    Set maxChild to index of child with larger value
    if heap.elements[root] < heap.elements[maxChild]
        Swap(heap.elements[root], heap.elements[maxChild])
        ReheapDown(heap, maxChild, bottom)
```



ReheapDown

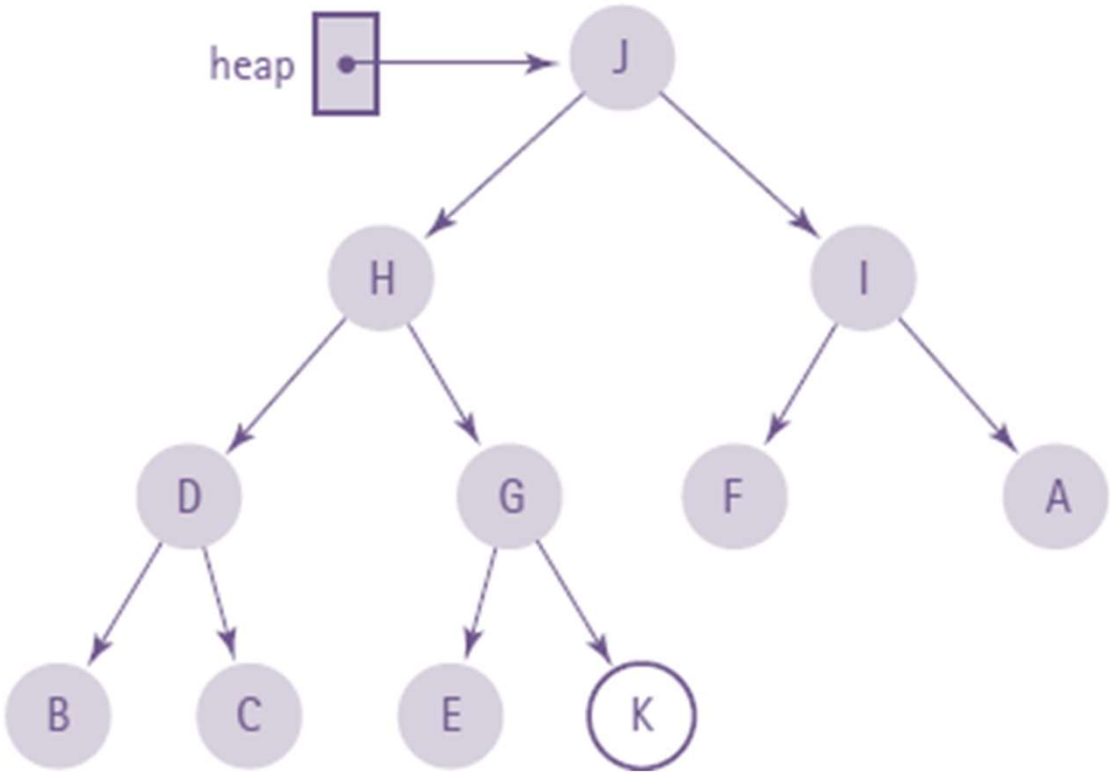
```
void ReheapDown(int root, int bottom)
{
  int maxChild;
  int rightChild;
  int leftChild;

  leftChild = root*2+1;
  rightChild = root*2+2;
  if (leftChild <= bottom)
  {
    if (leftChild == bottom)
      maxChild = leftChild;
    else
    {
      if (elements[leftChild] <= elements[rightChild])
        maxChild = rightChild;
      else
        maxChild = leftChild;
    }
    if (elements[root] < elements[maxChild])
    {
      Swap(elements[root], elements[maxChild]);
      ReheapDown(maxChild, bottom);
    }
  }
}
```

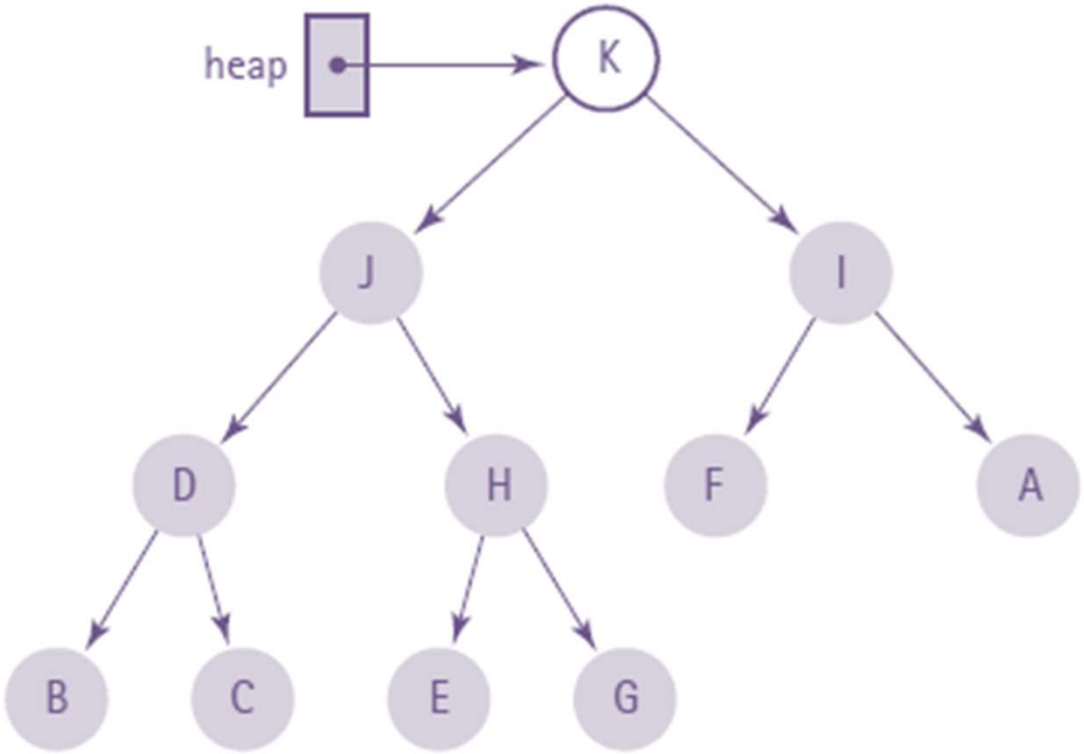
heap.elements

ReheapUp

Example

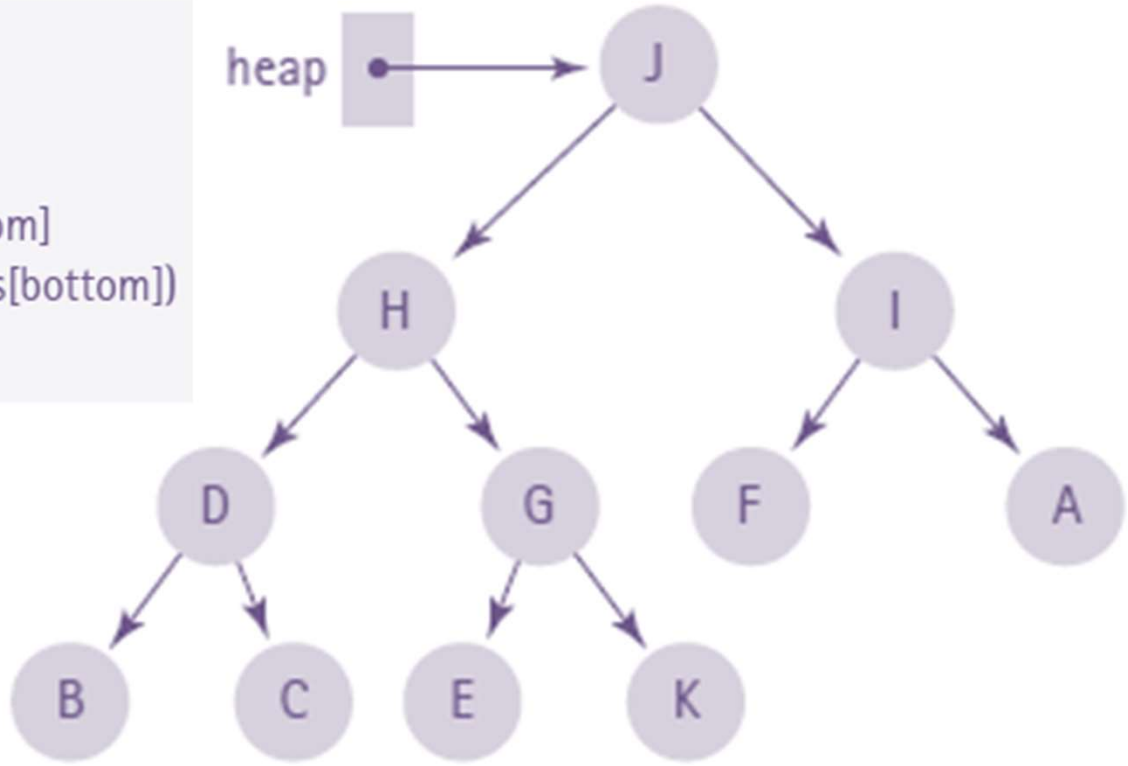


ReheapUp



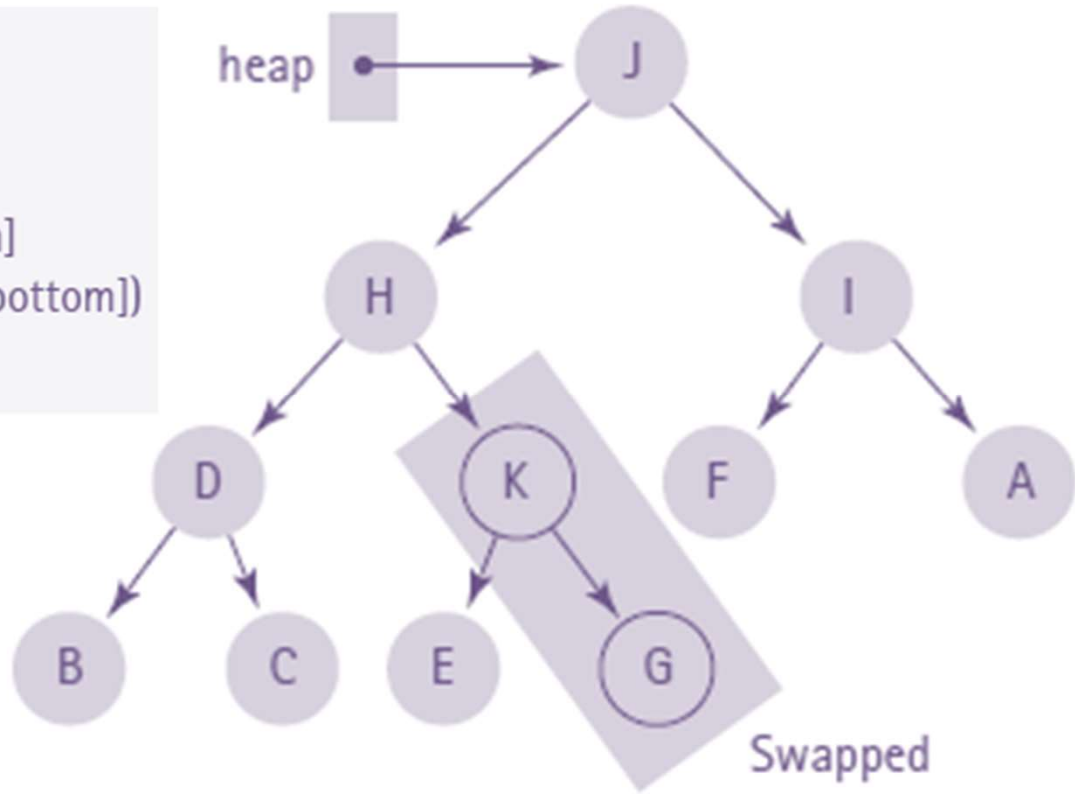
ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



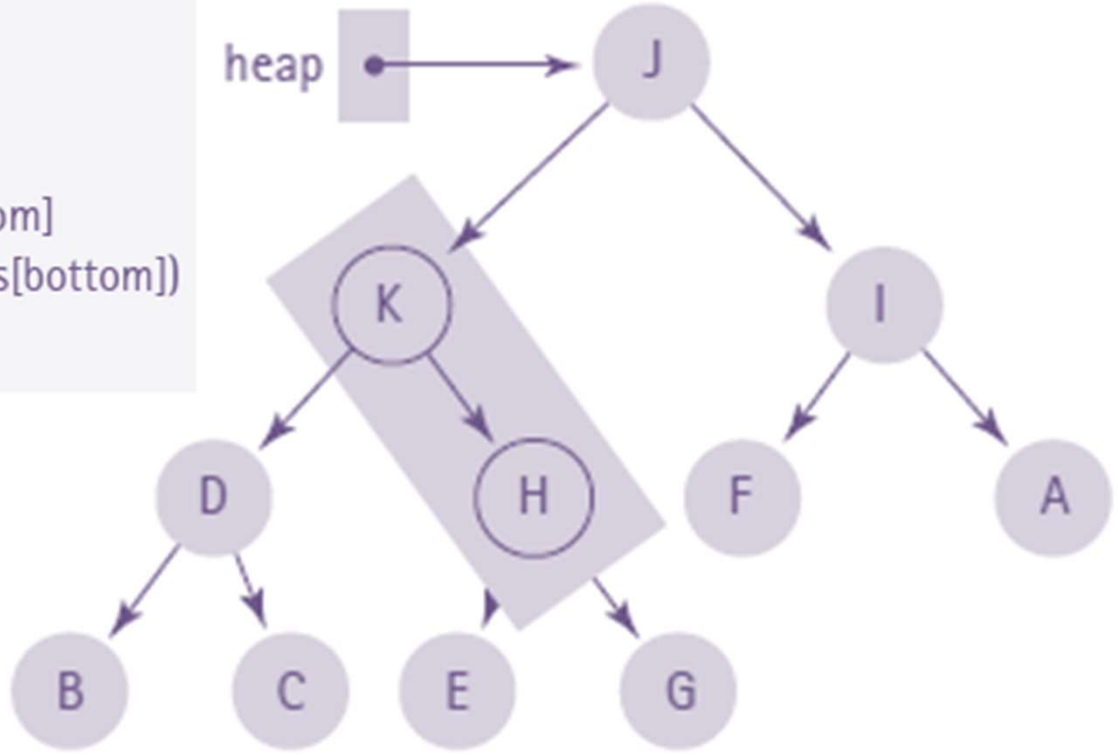
ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



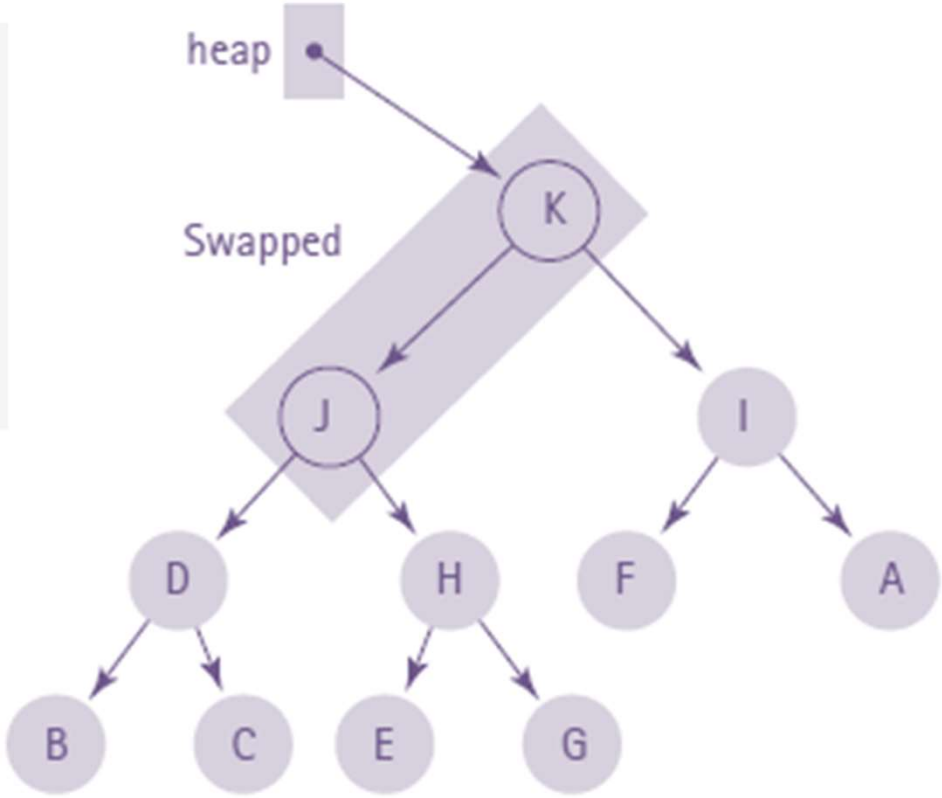
ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



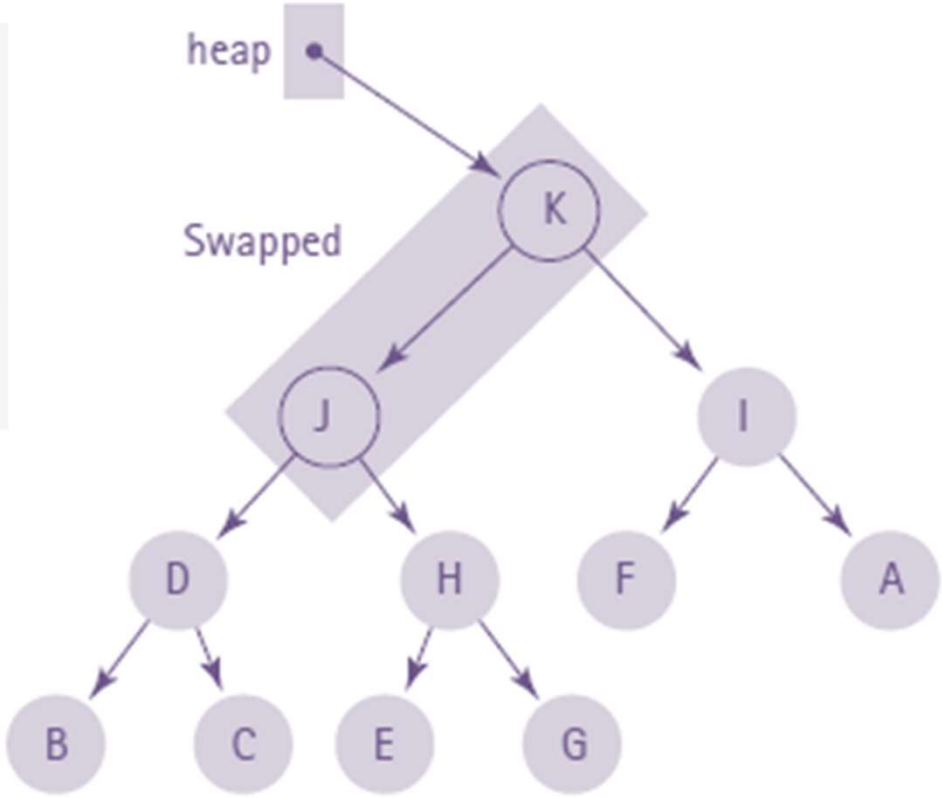
ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



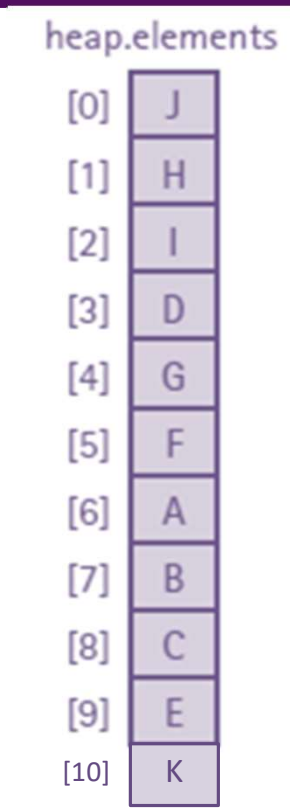
ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



ReheapUp

```
ReheapUp(heap, root, bottom)
if bottom > root
    Set parent to index of parent of bottom node
    if heap.elements[parent] < heap.elements[bottom]
        Swap(heap.elements[parent], heap.elements[bottom])
        ReheapUp(heap, root, parent)
```



ReheapUp

```
void ReheapUp(int root, int bottom)
{
  int parent;

  if (bottom > root)
  {
    parent = (bottom-1) / 2;
    if (elements[parent] < elements[bottom])
    {
      Swap(elements[parent], elements[bottom]);
      ReheapUp(root, parent);
    }
  }
}
```

heap.elements

[0]	J
[1]	H
[2]	I
[3]	D
[4]	G
[5]	F
[6]	A
[7]	B
[8]	C
[9]	E
[10]	K

Priority Queue

Priority Queue

isFull
isEmpty
makeEmpty

Enqueue
Dequeue

Dequeue

Dequeue

Set item to root element from queue → O(1)

Move last leaf element into root position → O(1)

Decrement length

items.ReheapDown(0, length - 1) → $O(\log_2 N)$

Dequeue

```
void dequeue(char heap[], int& size) {
    if (size > 0) {
        heap[0] = heap[--size];
        heapifyDown(heap, size, 0);
    } else {
        std::cerr << "Priority queue is empty.\n";
    }
}
```

Enqueue

Enqueue

Increment length → O(1)

Put newItem in next available position → O(1)

items.ReheapUp(0, length - 1) → $O(\log_2 N)$

Enqueue

```
void enqueue(char heap[], int& size, char value) {
    if (size < MAX_SIZE) {
        heap[size++] = value;
        heapifyUp(heap, size, size - 1);
    } else {
        std::cerr << "Priority queue is full.\n";
    }
}
```

Kompleksitas Implementasi Priority Queue

	Enqueue	Dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked list	$O(N)$	$O(1)$
Binary search tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$