# Binary Heap

## What is Binary Heap?

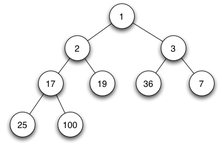
Properties similar to the Binary tree

Follows Heap property:

1. Value at any node is lesser than its children(Min-heap)
2. Value at any node is greater than its children(Max-heap)

Follow the property of complete tree:

All the nodes should be filled except last level and the value will be filled from its left



## Why Binary heap?

Finding Minimum/Maximum number will take Log n time

Insertion function takes only Log n time

**Other possible solution:**

**Array:**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 5 | 10 | 15 | 20 |  |  |

Search takes – O (1) Insertion: O (n)

**Linked List**: Insertion: O (n) Search O (1)

**Application:**

Prim’s Algorithm

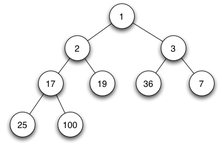
Heap sort

Priority Queue

# Type of heap?

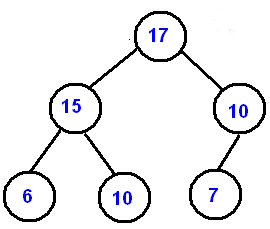
**Min- Heap:**

Value at any node is lesser than its children (Min-heap)



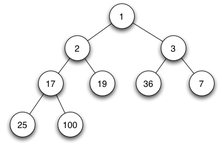
**Max Heap:**

Value at any node is greater than its children.



# Creation of heap

Array:



Logic:

Left = 2X

Right = 2X+1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

**Algorithm:**

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
|  |  |  |  |  |  |  |  |  |  |  |

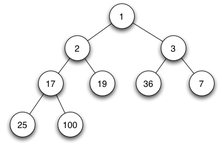
CreateHeap(Size)

Create a blank array with Size

Intiatize Size=0

**Time Complexity: O (1) ; Space Complexity: O (1)**

# Peek of heap



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

Peek()

If tree exist()

Return Error

Else

Return Arr[-1]

**Time Complexity: O (1) ; Space Complexity: O (1)**

## Size of heap

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

Size()

If tree exist()

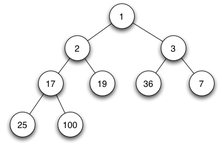
Return Error

Else

Return SizeofHeap

**Time Complexity: O (1) ; Space Complexity: O (1)**

# Insert heap



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

Insert(Value)

Insert value in first array

sizeHeap++

HeapBottomToTop(sizeHeap)

SizeHeap:

Compare if its parent is lesser than the node

If parent < child

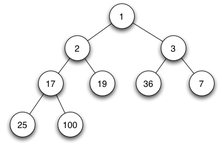
Return null

Else

Swap parent and child and repeat the comparison

**Time Complexity: O (logn) ; Space Complexity: O (logn)**

# Delete element from heap:



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

Delete()

Extract first cell

Promote the last cell to root

sizeofHeap—

heapifyTopBottom()

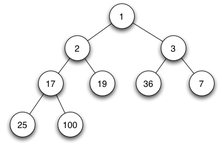
heapifyTopBottom()

Compare which node is smaller

Swap child to parent node

**Time Complexity: O (logn) ; Space Complexity: O (logn)**

# Deletion of Heap



|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |  |
| **X** | 1 | 2 | 3 | 17 | 19 | 36 | 7 | 25 | 100 |  |

DeleteHeap()

Set array to null