7. 堆 (Heap) 和 二叉堆 (Binary Heap) 的实现和特性

堆 Heap

[Heap 维基百科](https://en.wikipedia.org/wiki/Heap_(daata_structure))

- 定义：Heap 是可以迅速找到一堆数中的 最大或者最小值 的数据结构.

- 分类：
  - 大顶堆/大根堆：根节点最大的堆；
  - 小顶堆/小根堆：根节点最小的堆.

- 常见的堆：
  - 二叉堆；
  - 斐波那契堆.

- 常见操作 API

```
// 大顶堆
find-max : O(1)
delete-max : O(logN)
insert(create) : O(logN) or O(1)
```

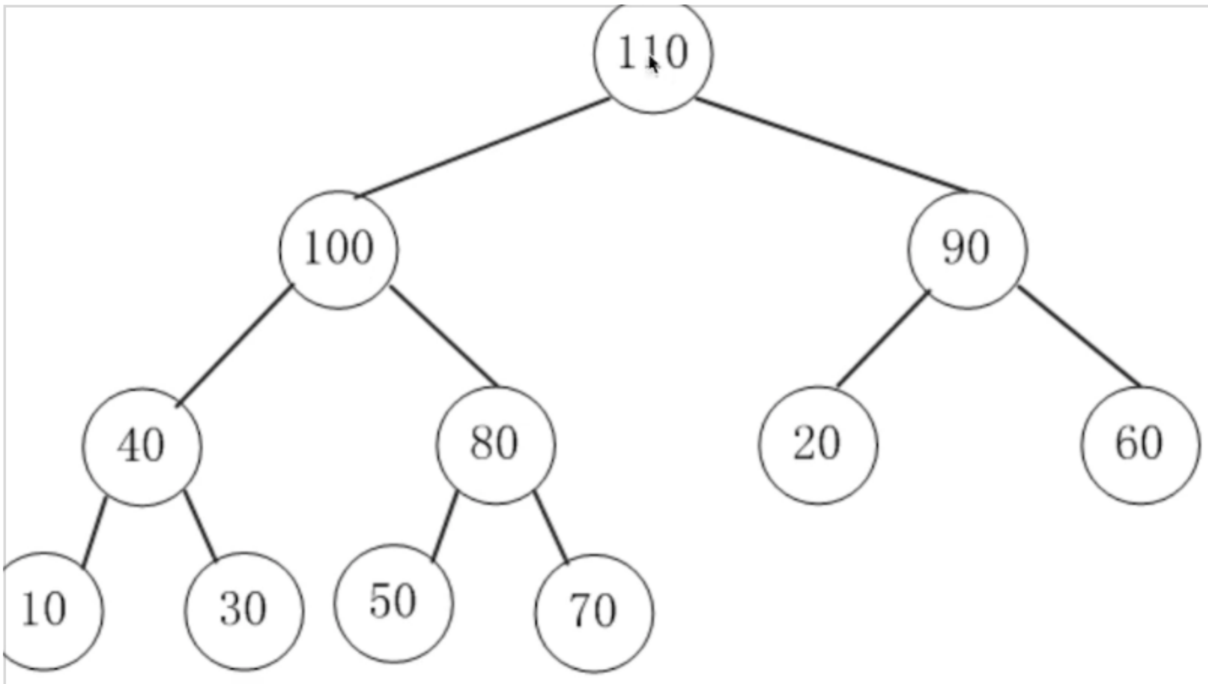
- 不同方法来实现 Heap 的 时间复杂度

Operation	find-max	delete-max	insert	increase-key	meld
Binary[8]	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Leftist	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)$	$\Theta(\log n)$
Binomial[8][9]	$\Theta(1)$	$\Theta(\log n)$	$\Theta(1)$[b]	$\Theta(\log n)$	$O(\log n)$[c]
Fibonacci[8][10]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Pairing[11]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$o(\log n)$[b][d]	$\Theta(1)$
Brodal[14][e]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Rank-pairing[16]	$\Theta(1)$	$O(\log n)$[b]	$\Theta(1)$	$\Theta(1)$[b]	$\Theta(1)$
Strict Fibonacci[17]	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
2–3 heap[18]	$O(\log n)$	$O(\log n)$[b]	$O(\log n)$[b]	$\Theta(1)$	?

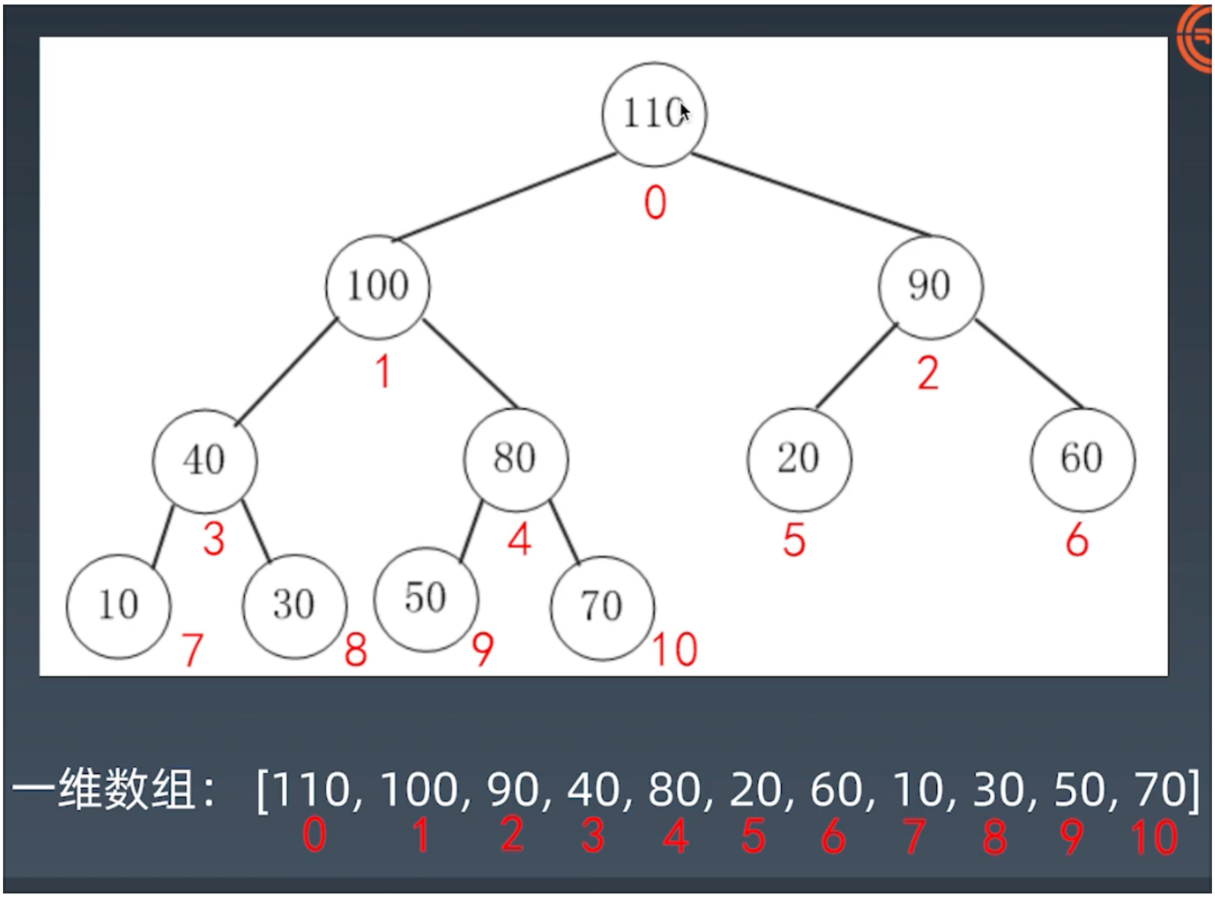
二叉堆 Binary Heap

- 定义：通过 二叉树 来实现的堆
  - （**注意：不是二叉搜索树**，因为找最小值,即左子树最左边的值，其时间复杂度是 O(long(n)）不是 O(1))
  - 注意：二叉堆 是 堆 的一种常见且简单的实现，但并不是最优的实现.所以，在工程中直接使用优先队列 priority_queue 即可.

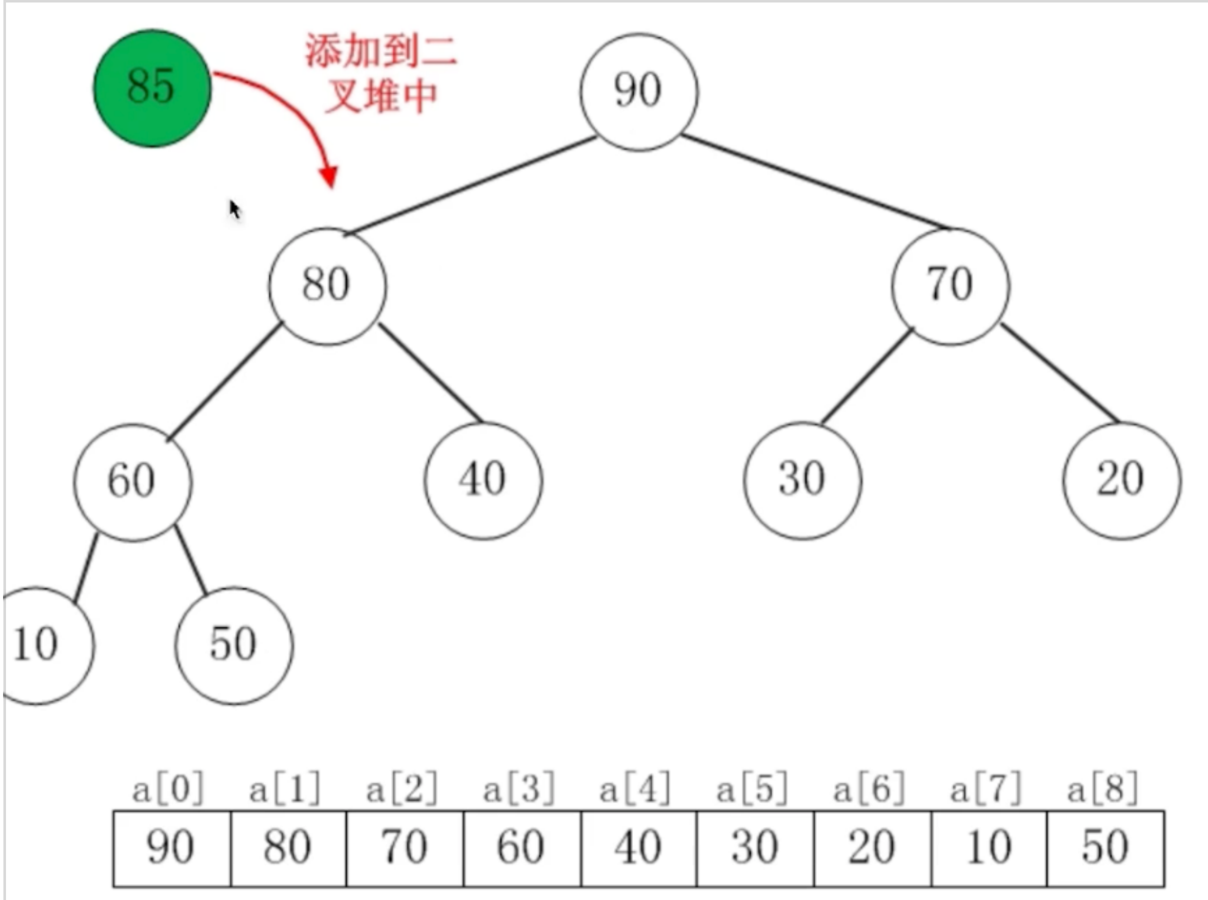
- 二叉堆（大顶）性质：
  1. 是一棵完全树;
  2. 树中任意节点的值总是 >= 其子节点值， 保证了每个子树的根节点是子树中最大的值.

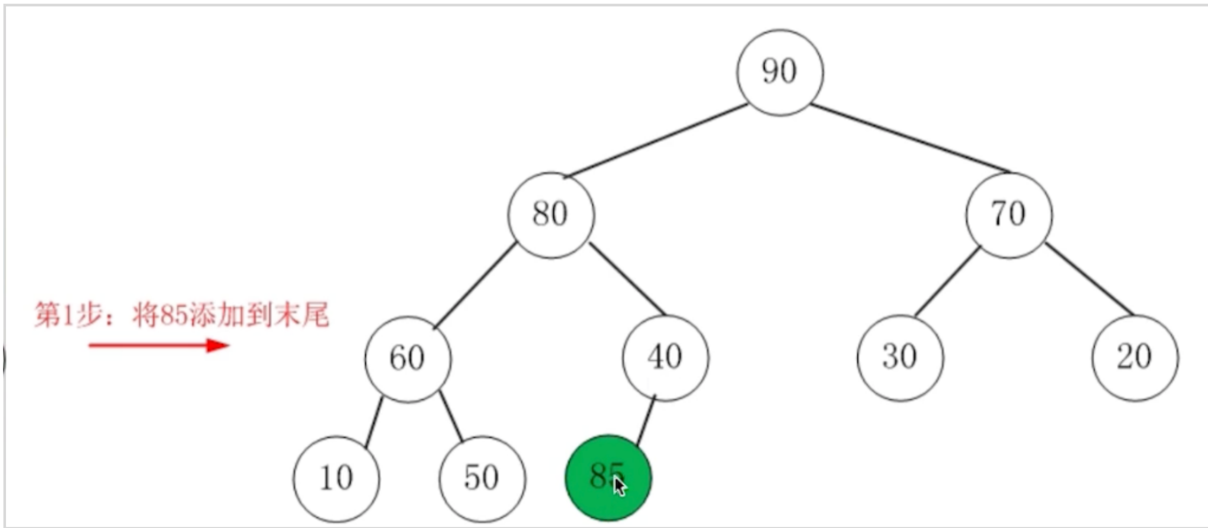


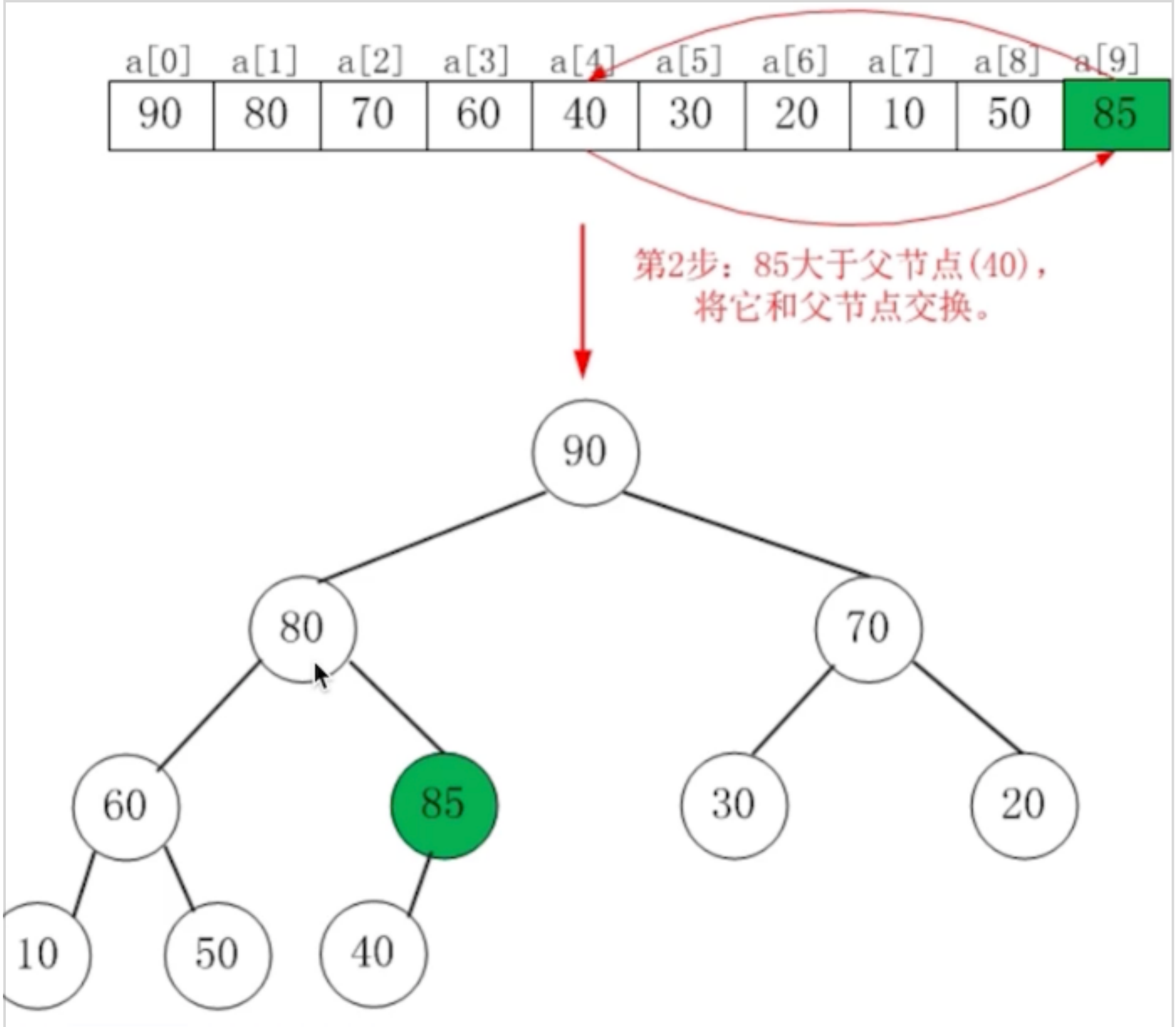
- Binary Heap 实现细节
    1. 二叉堆一般都通过 “数组” 来实现；
    2. 假设 “第一个元素”在数组中的索引为 0 的话，则父节点和子节点的位置关系如下：
        a. 索引为 i 的 左子节点 的索引是 (2 * i + 1);
        b. 索引为 i 的 右子节点 的索引是 (2 * i + 2);
        c. 索引为 i 的 父节点 的索引是 floor((i-1)/2).

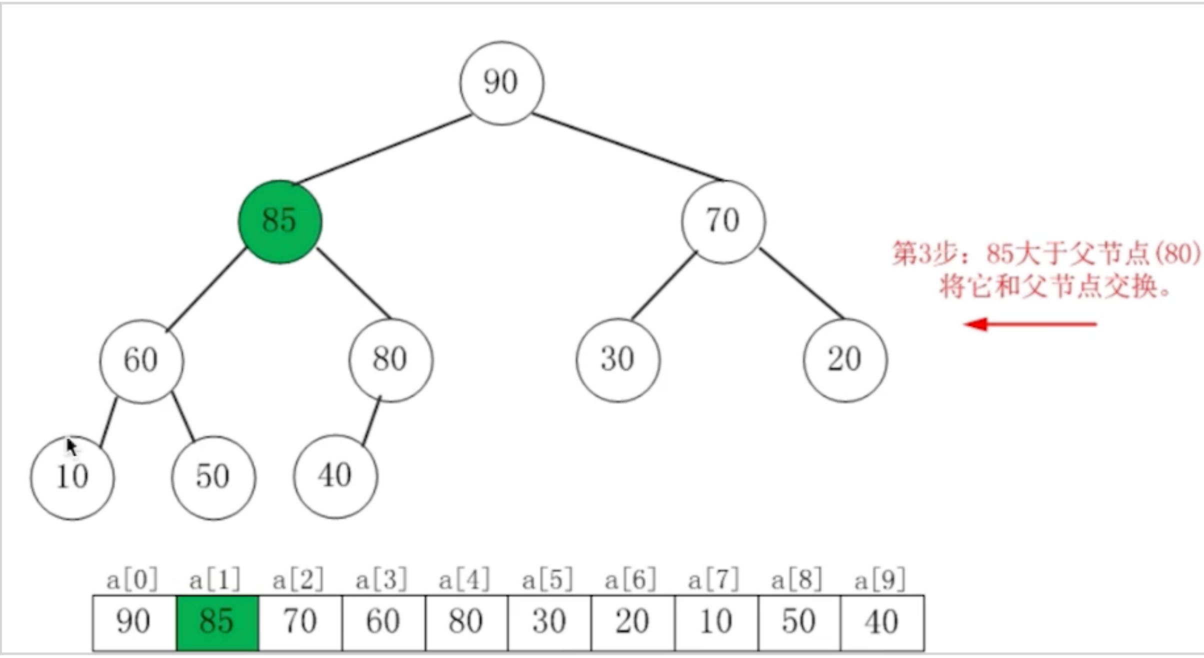


- Insert 插入操作 ： O(longN)
    1. 新元素一律先插入到堆的尾部；
    2. Heapify Up 依次向上调整整个堆的结构（一直到根即可）；


85
添加到二
叉堆中
90
80
70
60
40
30
20
10
50
a[0] a[1] a[2] a[3] a[4] a[5] a[6] a[7] a[8]
90 80 70 60 40 30 20 10 50

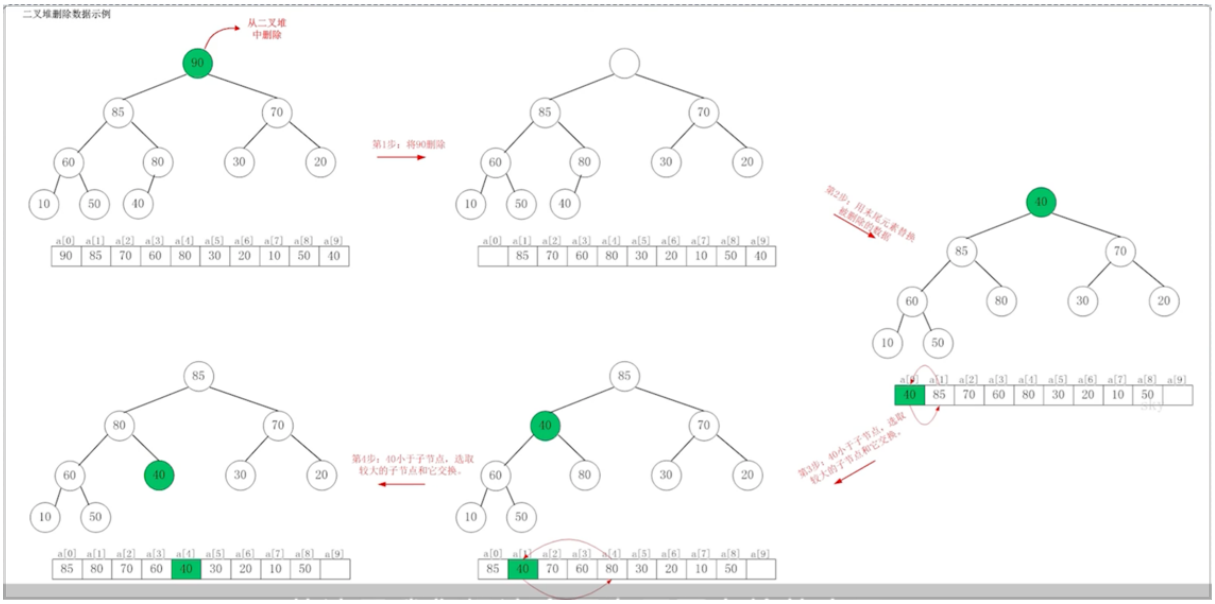

90
80
70
第1步：将85添加到末尾
60
40
30
20
10
50
85





- Delete Max 删除栈顶操作
  1. 将堆尾元素替换到顶部（即堆顶被堆尾替代删除掉）；
  2. Heapify Down 依次从根部向下调整整个堆的结构（一直到堆尾即可）.

     3



堆的实现代码

Java

```
import java.util.Arrays;
import java.util.NoSuchElementException;


public class BinaryHeap {


    private static final int d = 2;
    private int[] heap;
    private int heapSize;


    /**
     * This will initialize our heap with default size.
     */
    public BinaryHeap(int capacity) {
        heapSize = 0;
        heap = new int[capacity + 1];
        Arrays.fill(heap, -1);
```

```java
    }



    public boolean isEmpty() {
        return heapSize == 0;
    }



    public boolean isFull() {
        return heapSize == heap.length;
    }






    private int parent(int i) {
        return (i - 1) / d;
    }



    private int kthChild(int i, int k) {
        return d * i + k;
    }



    /**
     * Inserts new element in to heap
     * Complexity: O(log N)
     * As worst case scenario, we need to traverse till the root
     */
    public void insert(int x) {
        if (isFull()) {
            throw new NoSuchElementException("Heap is full, No space to insert
new element");
        }
        heap[heapSize] = x;
        heapSize ++;
        heapifyUp(heapSize - 1);
    }
```

```java
    /**
     * Deletes element at index x
     * Complexity: O(log N)
     */
    public int delete(int x) {
        if (isEmpty()) {
            throw new NoSuchElementException("Heap is empty, No element to
delete");
        }
        int maxElement = heap[x];
        heap[x] = heap[heapSize - 1];
        heapSize--;
        heapifyDown(x);
        return maxElement;
    }


    /**
     * Maintains the heap property while inserting an element.
     */
    private void heapifyUp(int i) {
        int insertValue = heap[i];
        while (i > 0 && insertValue > heap[parent(i)]) {
            heap[i] = heap[parent(i)];
            i = parent(i);
        }
        heap[i] = insertValue;
    }


    /**
     * Maintains the heap property while deleting an element.
     */
    private void heapifyDown(int i) {
        int child;
        int temp = heap[i];
        while (kthChild(i, 1) < heapSize) {
```

```java
            child = maxChild(i);
            if (temp >= heap[child]) {
                break;
            }
            heap[i] = heap[child];
            i = child;
        }
        heap[i] = temp;
    }


    private int maxChild(int i) {
        int leftChild = kthChild(i, 1);
        int rightChild = kthChild(i, 2);
        return heap[leftChild] > heap[rightChild] ? leftChild : rightChild;
    }


    /**
     * Prints all elements of the heap
     */
    public void printHeap() {
        System.out.print("nHeap = ");
        for (int i = 0; i < heapSize; i++)
            System.out.print(heap[i] + " ");
        System.out.println();
    }


    /**
     * This method returns the max element of the heap.
     * complexity: O(1)
     */
    public int findMax() {
        if (isEmpty())
            throw new NoSuchElementException("Heap is empty.");
        return heap[0];
    }
```

```
    public static void main(String[] args) {
        BinaryHeap maxHeap = new BinaryHeap(10);
        maxHeap.insert(10);
        maxHeap.insert(4);
        maxHeap.insert(9);
        maxHeap.insert(1);
        maxHeap.insert(7);
        maxHeap.insert(5);
        maxHeap.insert(3);



        maxHeap.printHeap();
        maxHeap.delete(5);
        maxHeap.printHeap();
        maxHeap.delete(2);
        maxHeap.printHeap();
    }
}
```

C/C++

```
#include <iostream>
using namespace std;

class BinaryHeap {
public:
    BinaryHeap(int capacity);
    void insert(int x);
    int erase(int x);
    int findMax();
    void printHeap();

    bool isEmpty() { return heapSize == 0; }
    bool isFull() { return heapSize == capacity; }
    ~BinaryHeap() { delete[] heap; }
```

```
private:
    void heapifyUp(int i);
    void heapifyDown(int i);
    int maxChild(int i);

    int parent(int i) { return (i - 1) / 2; }
    int kthChild(int i, int k) { return 2 * i + k; }

private:
    int *heap;
    int heapSize;
    int capacity;
};

/**
 * This will initialize our heap with default size.
*/
BinaryHeap::BinaryHeap(int capacity) {
    this->heapSize = 0;
    this->capacity = capacity;
    this->heap = new int[capacity + 5];
}

/**
 * Inserts new element in to heap
 * Complexity: O(log N)
 * As worst case scenario, we need to traverse till the root
 */
void BinaryHeap::insert(int x) {
    try {
        if (isFull())
            throw -1;

        heap[heapSize] = x;
        heapSize ++;
        heapifyUp(heapSize - 1);
        return ;
    } catch (int e) {
```

```
        cout << "Heap is full, No space to insert new element" << endl;
        exit(-1);
    }
}

/**
 * Deletes element at index x
 * Complexity: O(log N)
 */
int BinaryHeap::erase(int x) {
    try {
        if (isEmpty())
            throw -1;

        int maxElement = heap[x];
        heap[x] = heap[heapSize - 1];
        heapSize--;
        heapifyDown(x);
        return maxElement;
    } catch (int e) {
        cout << "Heap is empty, No element to delete" << endl;
        exit(-1);
    }
}

/**
 * Maintains the heap property while inserting an element.
 */
void BinaryHeap::heapifyUp(int i) {
    int insertValue = heap[i];
    while (i > 0 && insertValue > heap[parent(i)]) {
        heap[i] = heap[parent(i)];
        i = parent(i);
    }
    heap[i] = insertValue;
}

/**
 * Maintains the heap property while deleting an element.
```

```
 */
void BinaryHeap::heapifyDown(int i) {
    int child;
    int temp = heap[i];
    while (kthChild(i, 1) < heapSize) {
        child = maxChild(i);
        if (temp >= heap[child]) {
            break;
        }
        heap[i] = heap[child];
        i = child;
    }
    heap[i] = temp;
}

int BinaryHeap::maxChild(int i) {
    int leftChild = kthChild(i, 1);
    int rightChild = kthChild(i, 2);
    return heap[leftChild] > heap[rightChild] ? leftChild : rightChild;
}

/**
 * This method returns the max element of the heap.
 * complexity: O(1)
 */
int BinaryHeap::findMax() {
    try {
        if (isEmpty())
            throw -1;

        return heap[0];
    } catch (int e) {
        cout << "Heap is empty." << endl;
        exit(-1);
    }
}

/**
 * Prints all elements of the heap
```

```
 */
void BinaryHeap::printHeap() {
    cout << "nHeap = ";
    for (int i = 0; i < heapSize; i++)
        cout << heap[i] << " ";
    cout << endl;
    return ;
}



int main() {
    BinaryHeap maxHeap(10);

    maxHeap.insert(10);
    maxHeap.insert(4);
    maxHeap.insert(9);
    maxHeap.insert(1);
    maxHeap.insert(7);
    maxHeap.insert(5);
    maxHeap.insert(3);

    maxHeap.printHeap();
    maxHeap.erase(5);
    maxHeap.printHeap();
    maxHeap.erase(2);
    maxHeap.printHeap();

    return 0;
}
```

Python

```


import sys


class BinaryHeap:
```

```
    def __init__(self, capacity):
        self.capacity = capacity
        self.size = 0
        self.Heap = [0]*(self.capacity + 1)
        self.Heap[0] = -1 * sys.maxsize
        self.FRONT = 1

    def parent(self, pos):
        return pos//2

    def leftChild(self, pos):
        return 2 * pos

    def rightChild(self, pos):
        return (2 * pos) + 1

    def isLeaf(self, pos):
        if pos >= (self.size//2) and pos <= self.size:
            return True
        return False

    def swap(self, fpos, spos):
        self.Heap[fpos], self.Heap[spos] = self.Heap[spos], self.Heap[fpos]

    def heapifyDown(self, pos):

        if not self.isLeaf(pos):
            if (self.Heap[pos] > self.Heap[self.leftChild(pos)] or
               self.Heap[pos] > self.Heap[self.rightChild(pos)]):

                if self.Heap[self.leftChild(pos)] <
self.Heap[self.rightChild(pos)]:
                    self.swap(pos, self.leftChild(pos))
                    self.heapifyDown(self.leftChild(pos))

                else:
                    self.swap(pos, self.rightChild(pos))
                    self.heapifyDown(self.rightChild(pos))
```

```python
def insert(self, element):
    if self.size >= self.capacity :
        return
    self.size+= 1
    self.Heap[self.size] = element

    current = self.size

    while self.Heap[current] < self.Heap[self.parent(current)]:
        self.swap(current, self.parent(current))
        current = self.parent(current)

def Print(self):
    for i in range(1, (self.size//2)+1):
        print(" PARENT : "+ str(self.Heap[i])+" LEFT CHILD : "+
                            str(self.Heap[2 * i])+" RIGHT CHILD : "+
                            str(self.Heap[2 * i + 1]))

def minHeap(self):

    for pos in range(self.size//2, 0, -1):
        self.heapifyDown(pos)

def delete(self):

    popped = self.Heap[self.FRONT]
    self.Heap[self.FRONT] = self.Heap[self.size]
    self.size-= 1
    self.heapifyDown(self.FRONT)
    return popped


def isEmpty(self):
    return self.size == 0

def isFull(self):
    return self.size == self.capacity
```

```python
if __name__ == "__main__":

    print('The minHeap is ')
    minHeap = BinaryHeap(5)
    minHeap.insert(5)
    minHeap.insert(3)
    minHeap.insert(17)
    minHeap.insert(10)
    minHeap.insert(84)
    minHeap.insert(19)
    minHeap.insert(6)
    minHeap.insert(22)
    minHeap.insert(9)
    minHeap.minHeap()

    minHeap.Print()
    print("The Min val is " + str(minHeap.delete()))
```

#Algorithm/Part II : Theory/Data Structure#