Title: Heaps

Author: Pınar Yücel

ID: 21802188

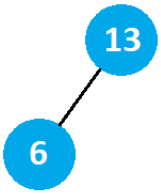
Assignment: 3

Question Part

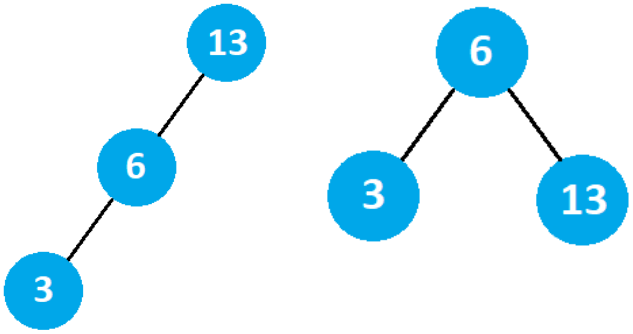
a)

After Insertion (13)

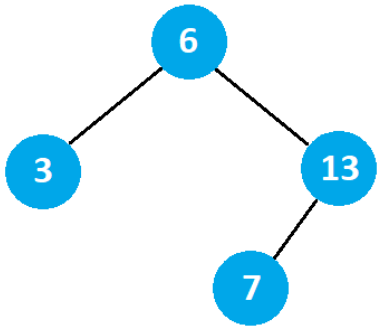
After Insertion (6)



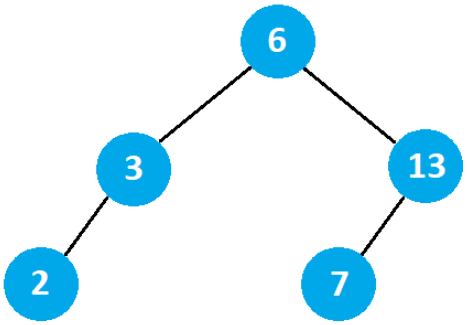
After Insertion (3)



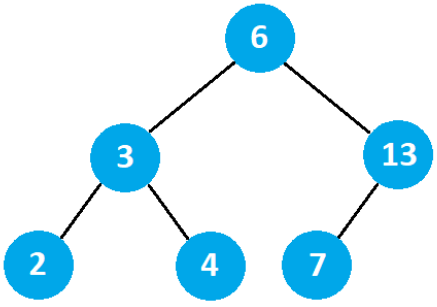
After Insertion (7)



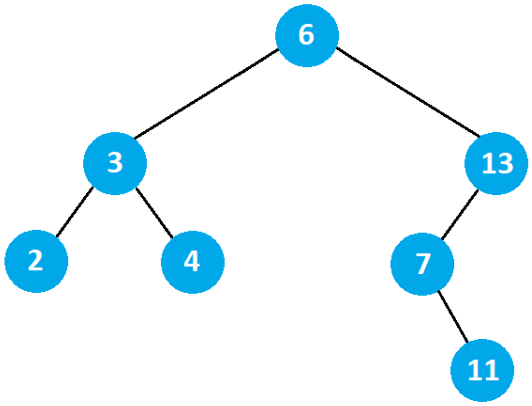
After Insertion (2)

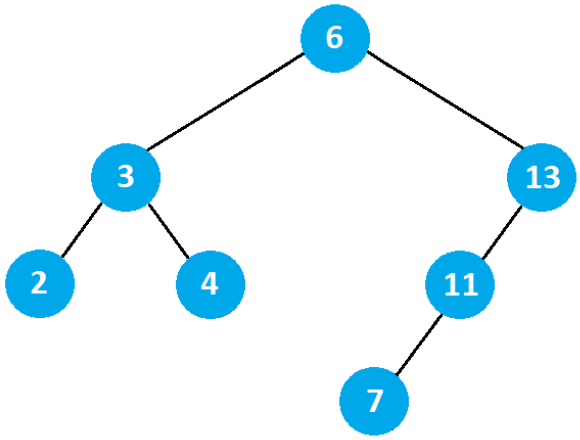


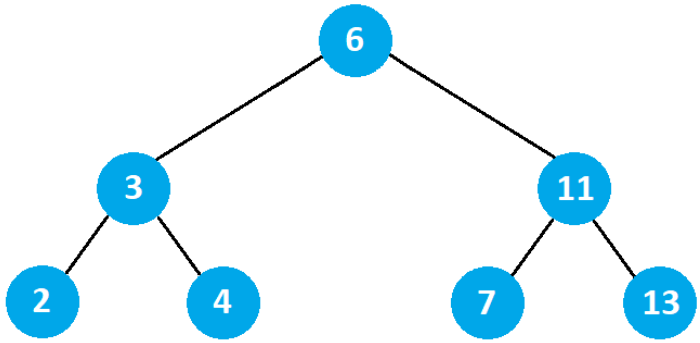
After Insertion (4)



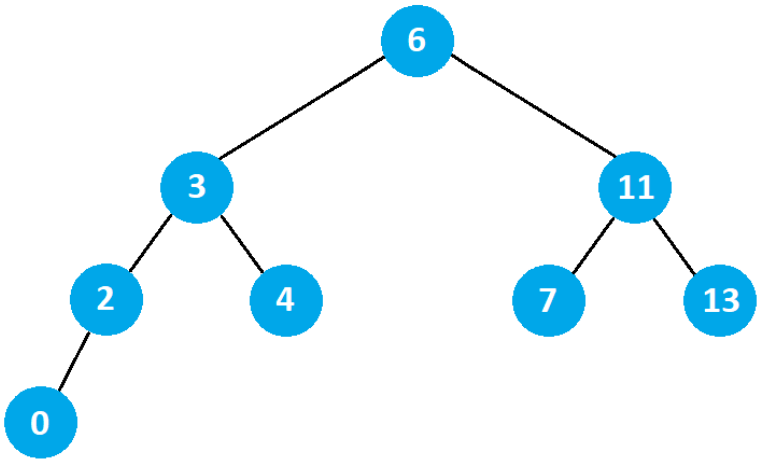
After Insertion (11)



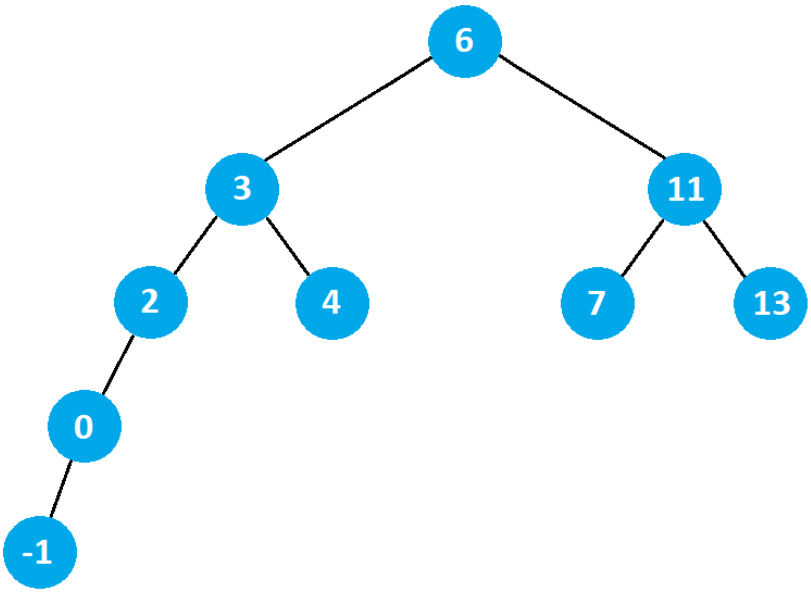


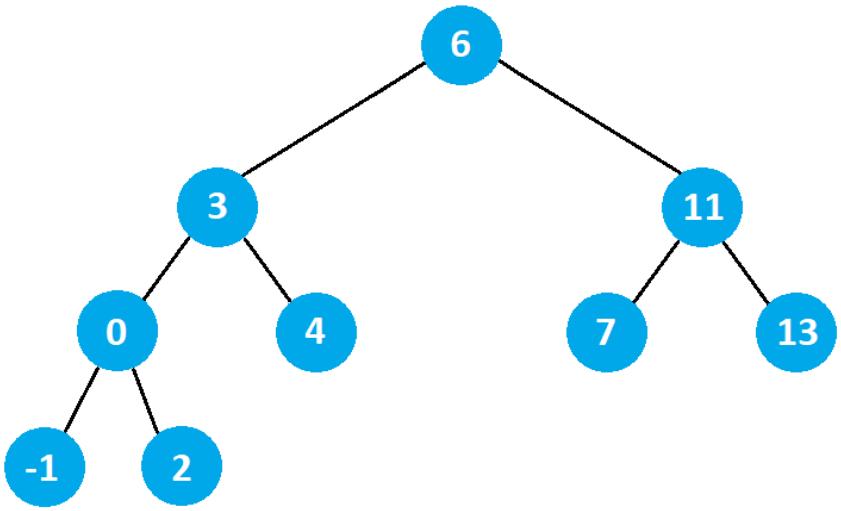


After Insertion (0)

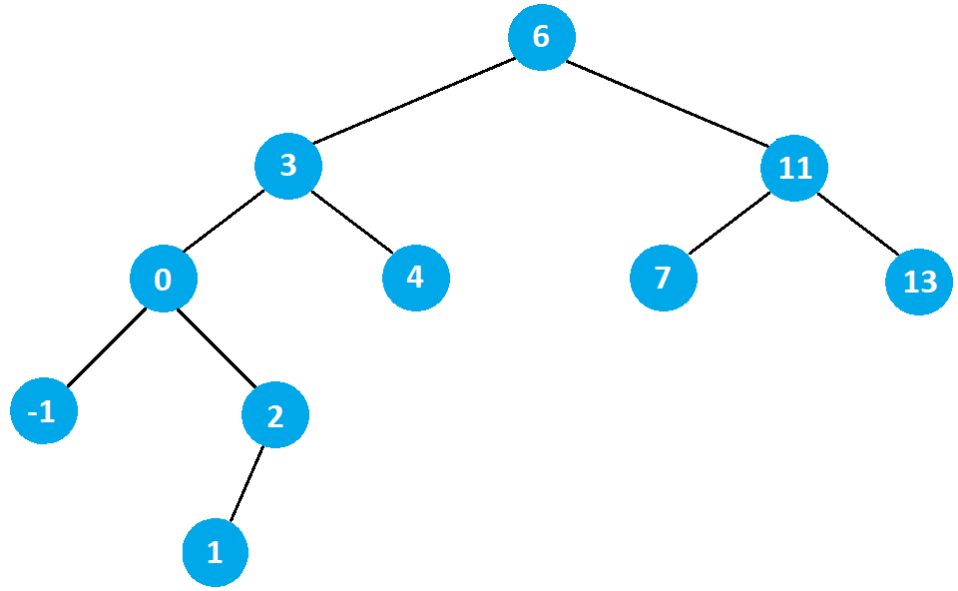


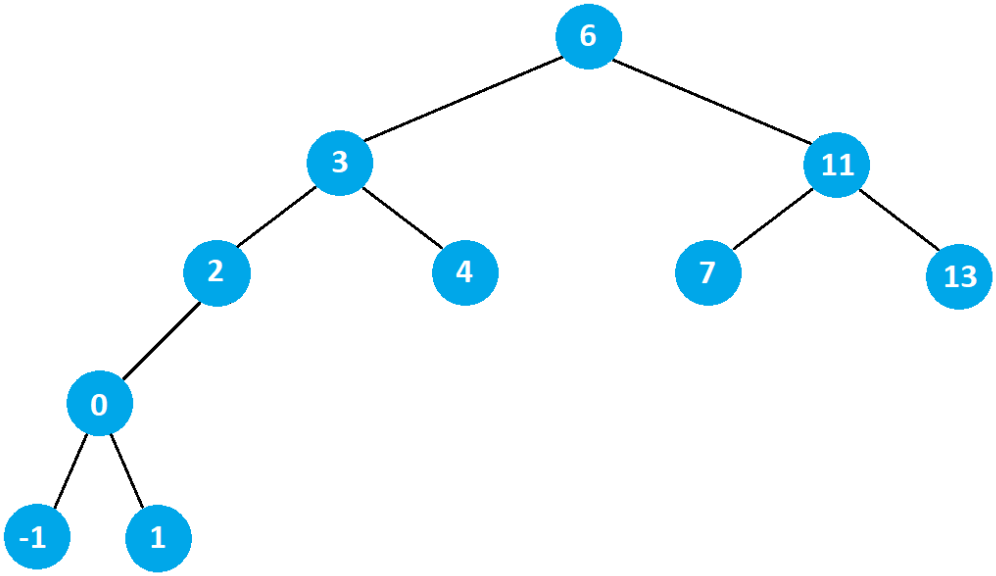
After Insertion (-1)

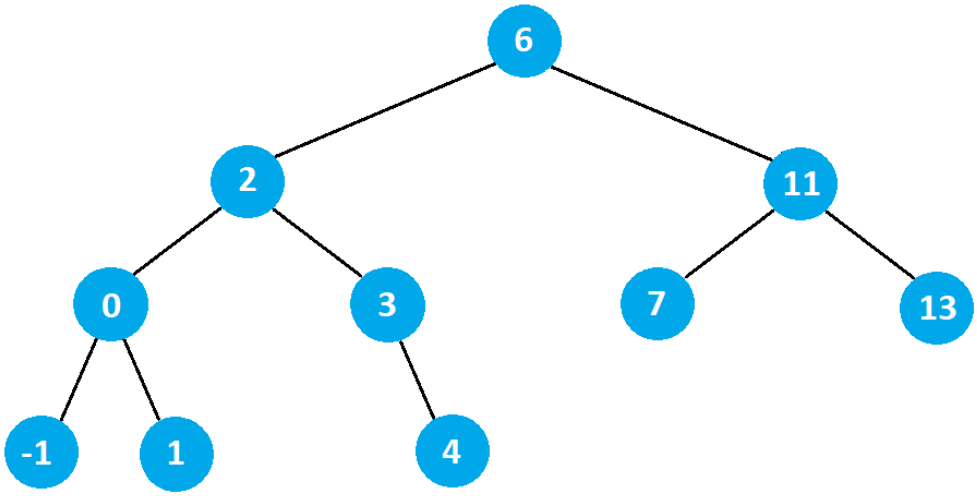




After Insertion (1)



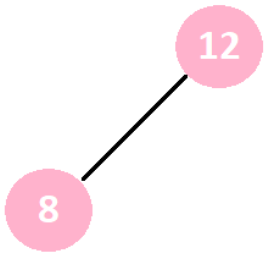


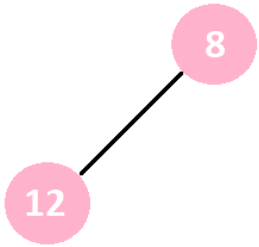


b)

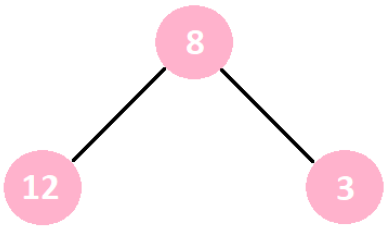
After Insertion (12)

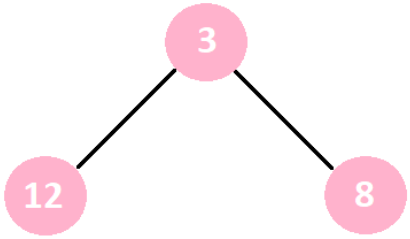
After Insertion (8)



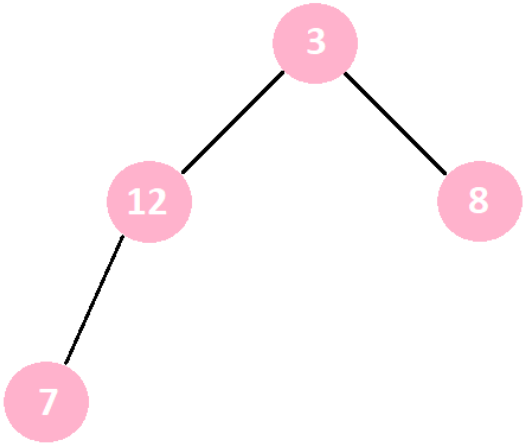


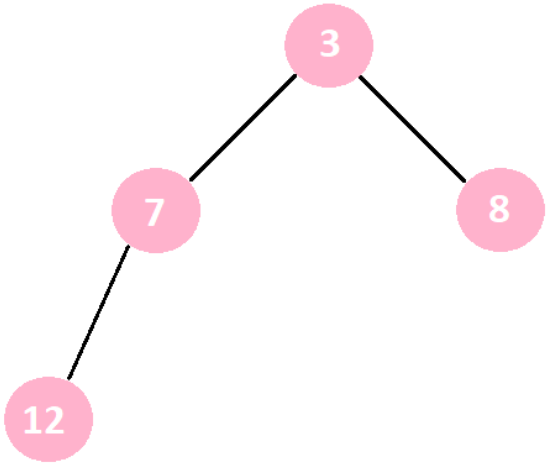
After Insertion (3)



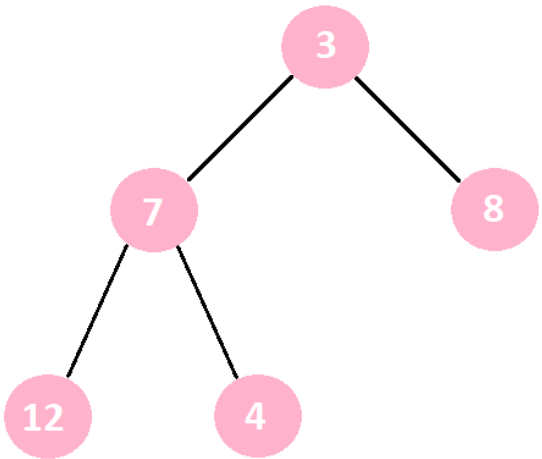


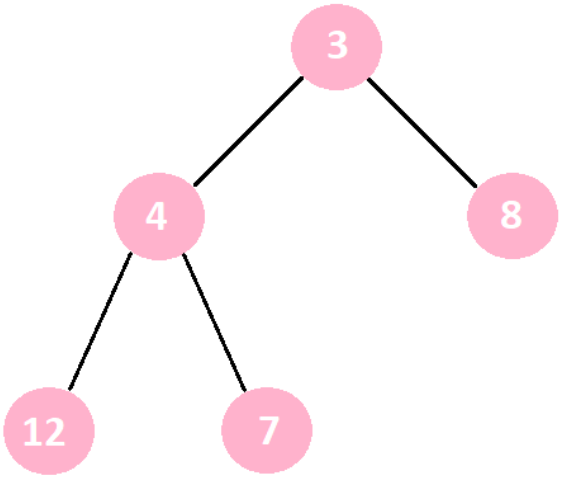
After Insertion (7)



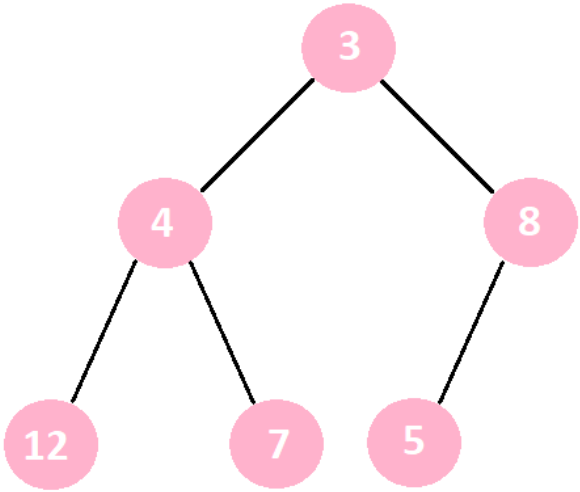


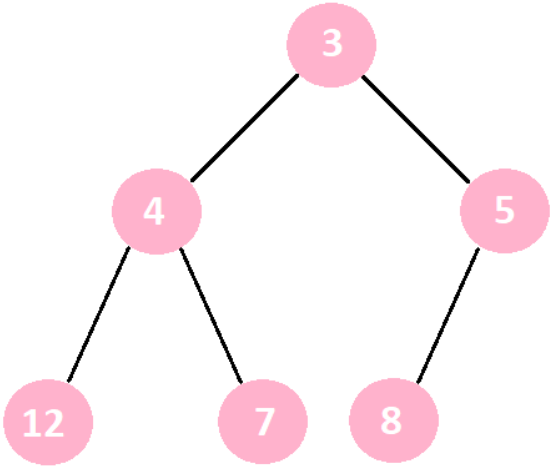
After Insertion (4)



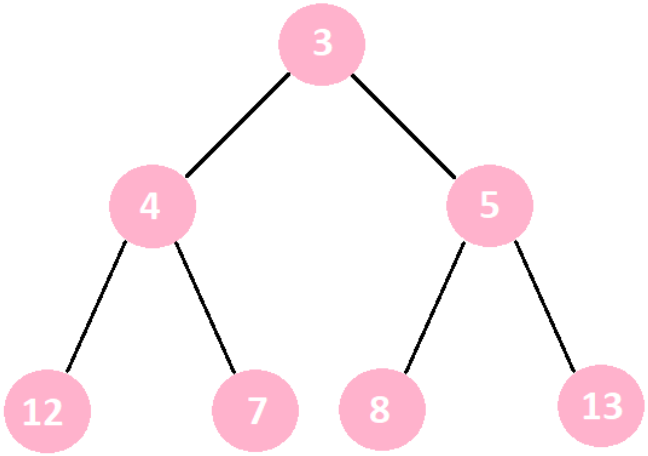


After Insertion (5)

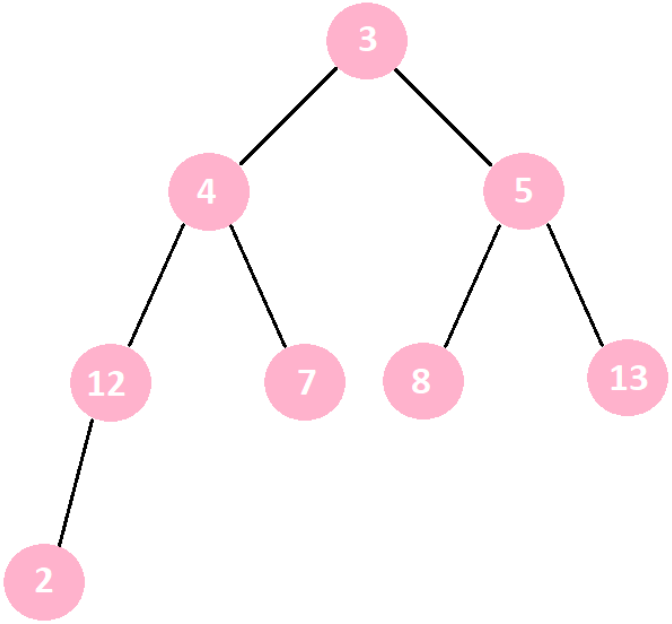


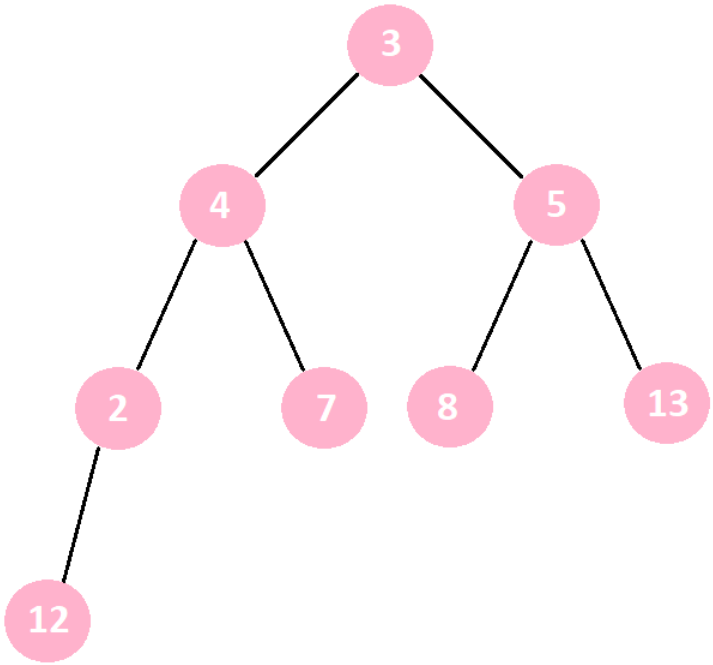


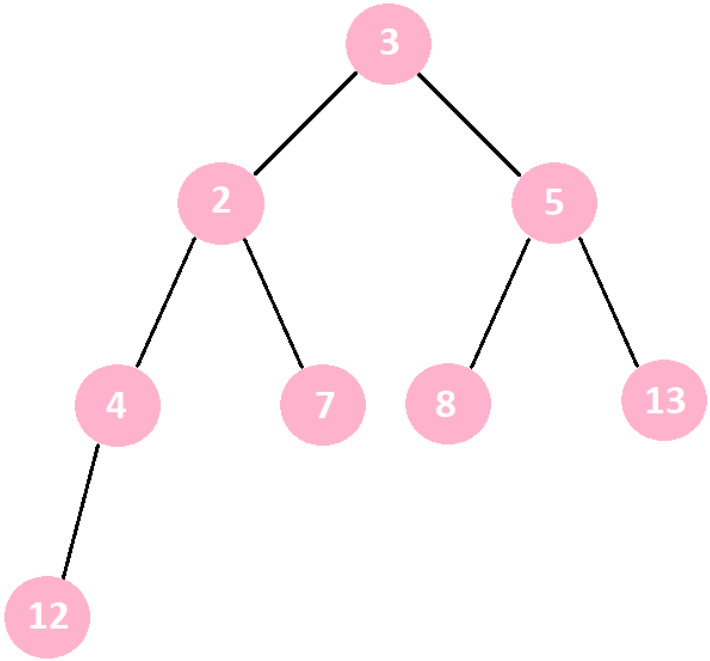
After Insertion (13)

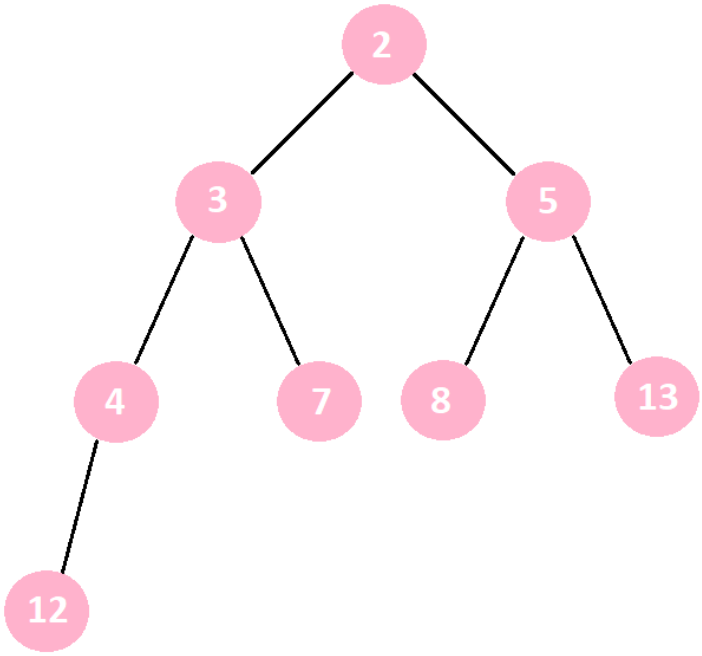


After Insertion (2)

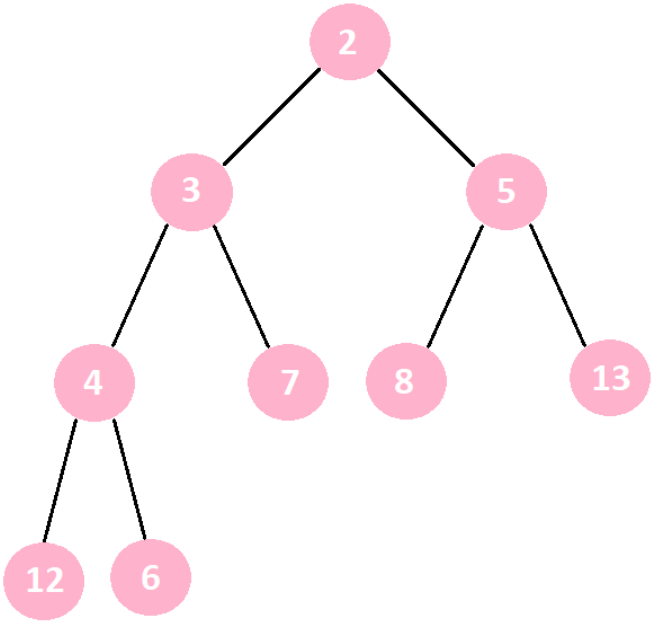



3
4
5
2
7
8
13
12

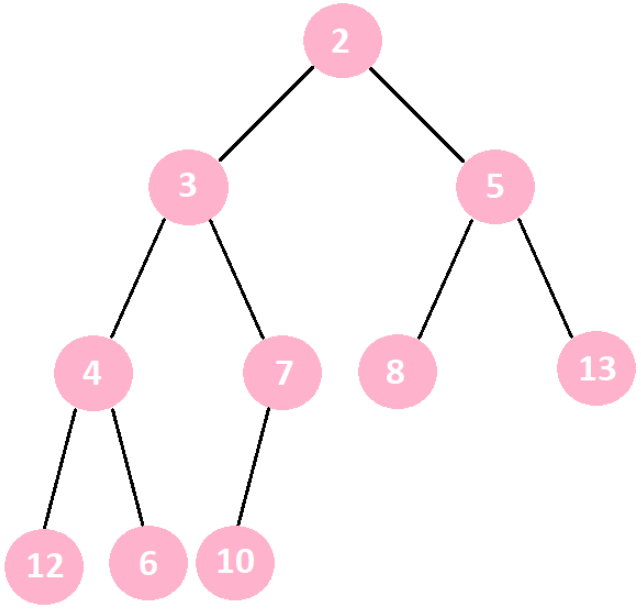

3
2
5
4
7
8
13
12



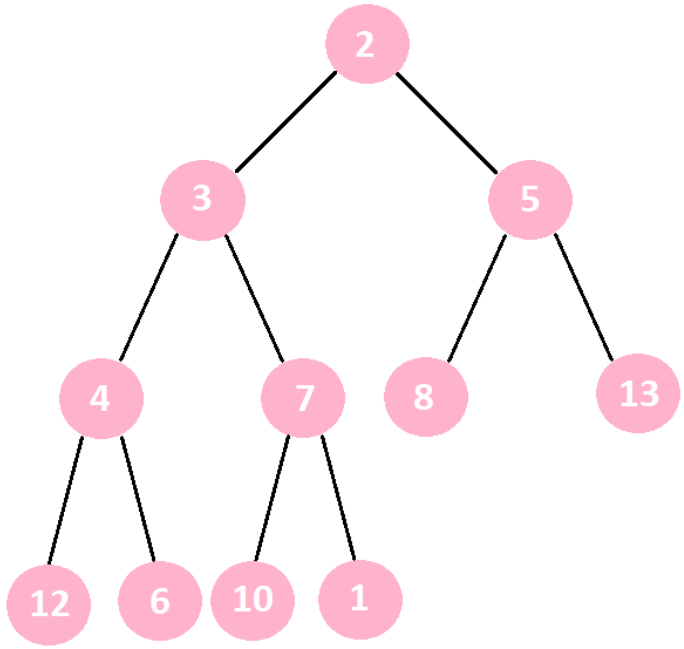
After Insertion (6)

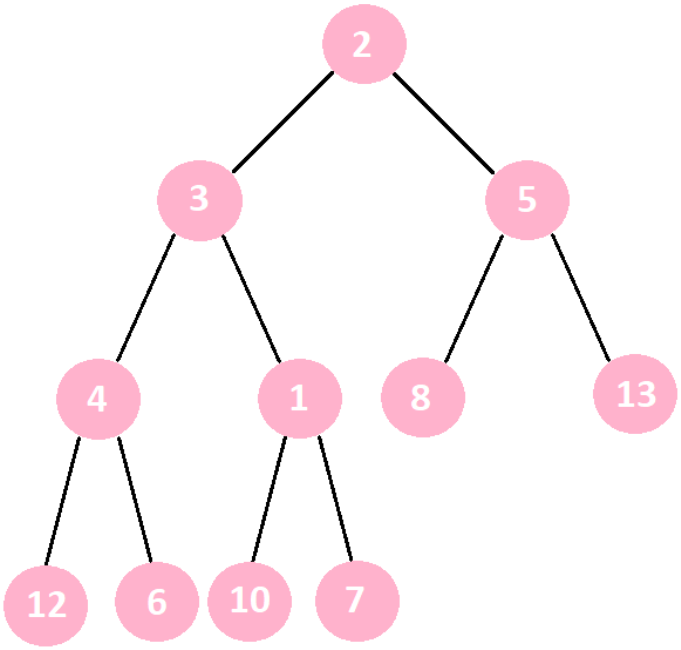


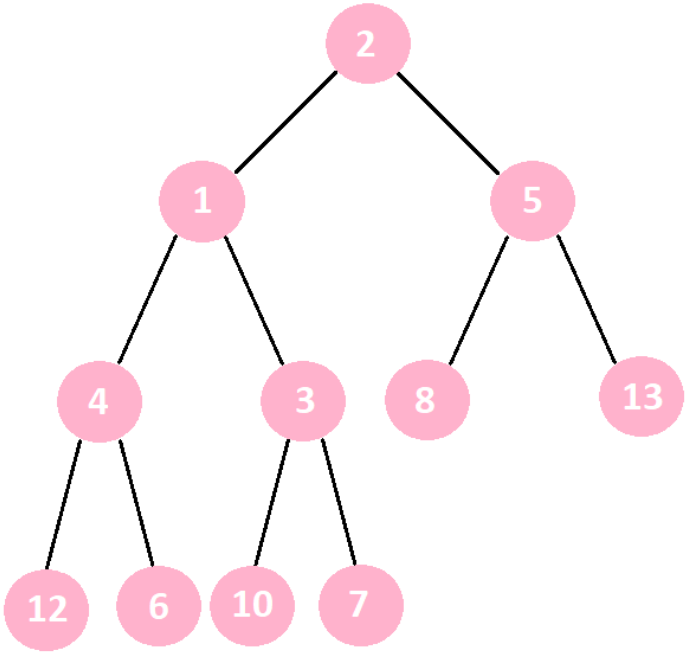
After Insertion (10)

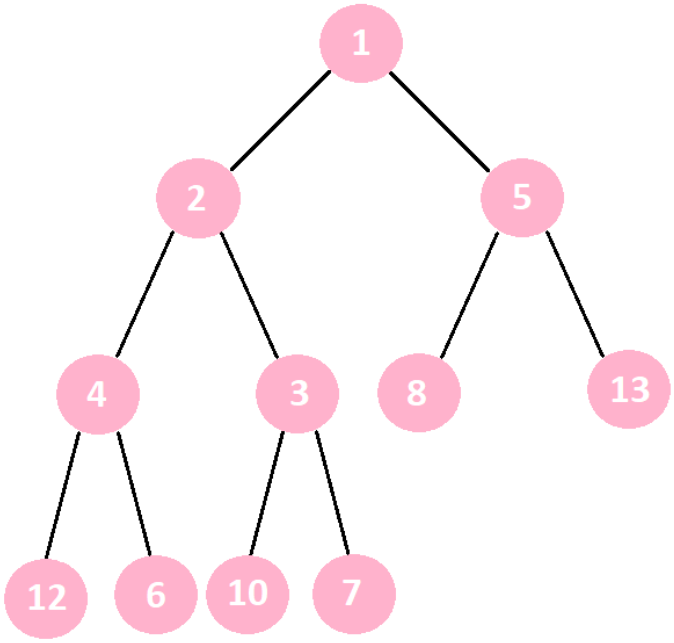


After Insertion (1)

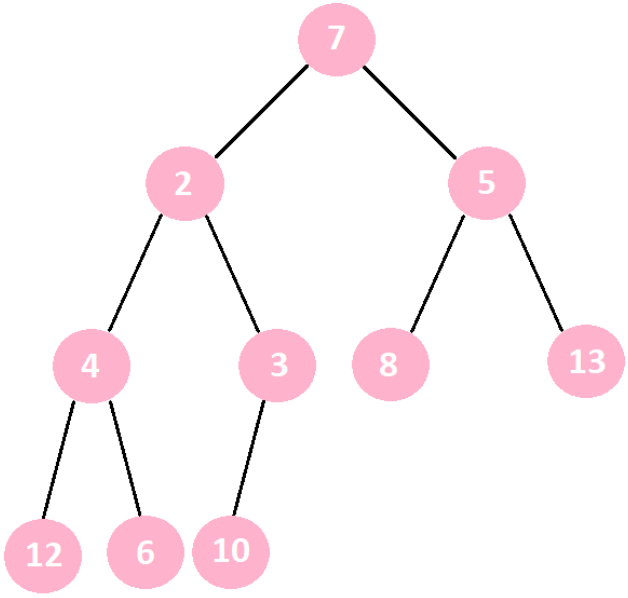


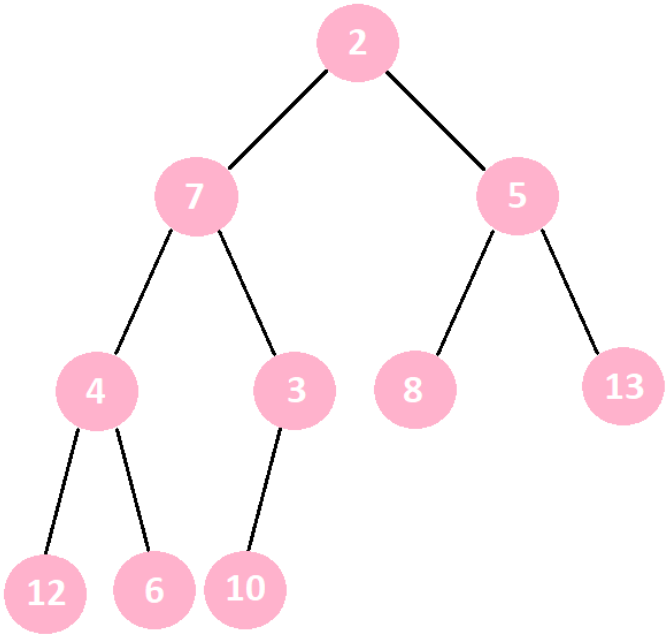

2
3
5
4
1
8
13
12
6
10
7

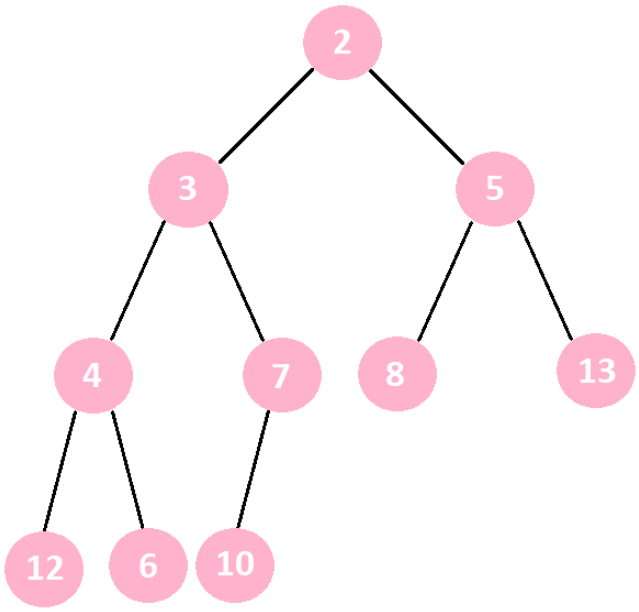

2
1
5
4
3
8
13
12
6
10
7



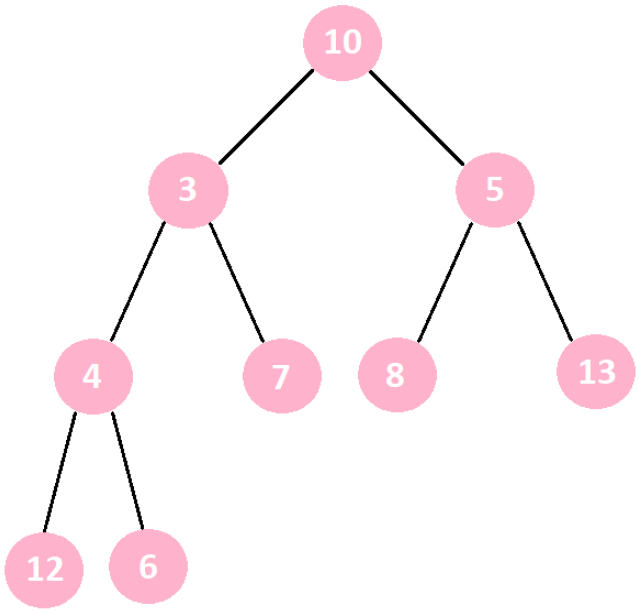
After Deletion (1)

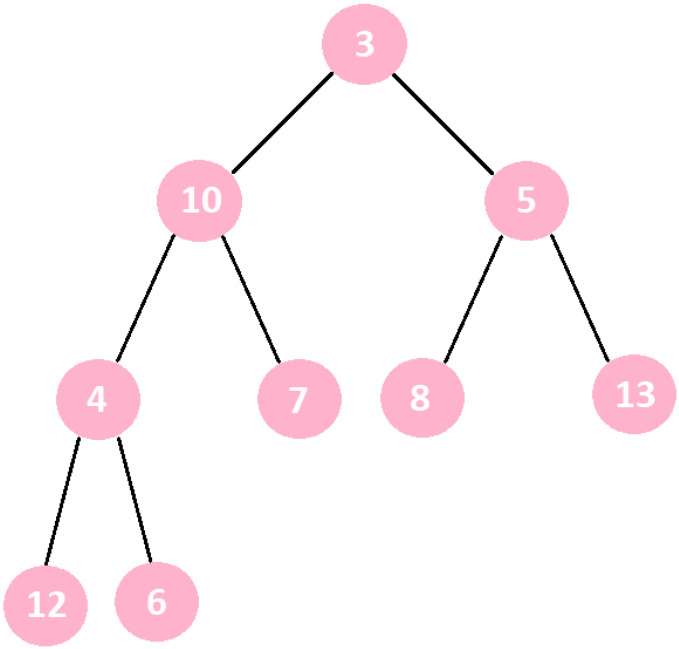


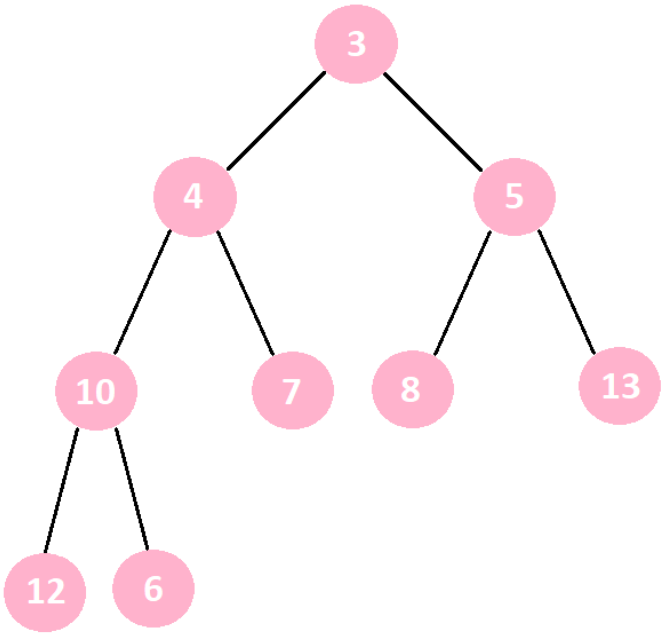

2
7
5
4
3
8
13
12
6
10

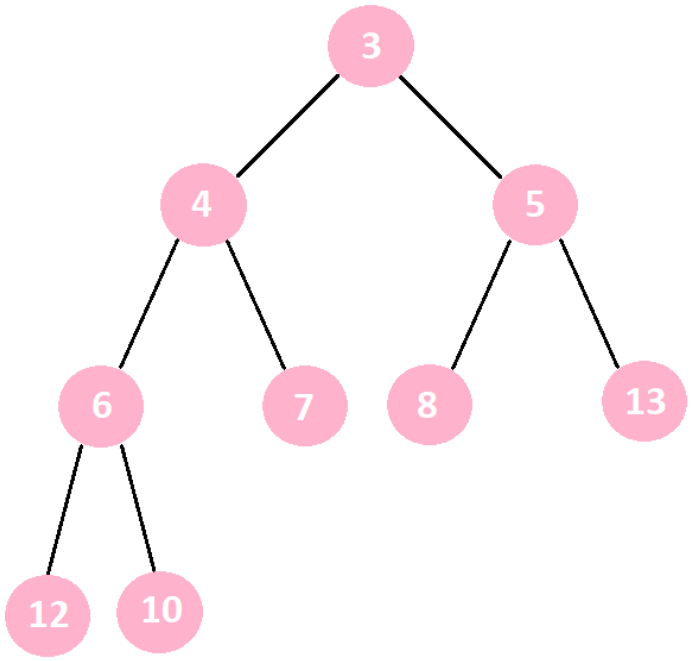

2
3
5
4
7
8
13
12
6
10

After Deletion (2)





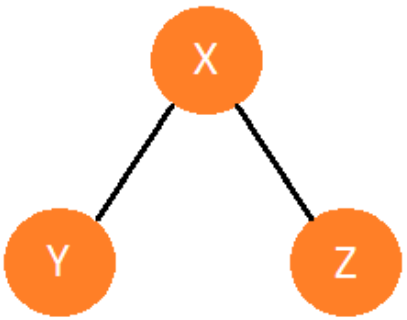

3
4
5
10
7
8
13
12
6


3
4
5
6
7
8
13
12
10

c)

Answer: It is never guaranteed that the output will be sorted in any case. Proof is below.

For Max Heap



Suppose that X, Y and Z represent numbers. From the properties of heaps we know that X is bigger than both Y and Z. However, we can not know whether Y is smaller than or larger than Z.

Preorder traversal: X Y Z

Since we know that X is bigger than Y and Z, the output can not be sorted in ascending order because the biggest element comes first.

For the output to be sorted in descending order Y should be bigger than Z but this is not guaranteed.

Inorder traversal: Y X Z

Since we know that X is bigger than Y and Z, the output can not be sorted in ascending order because the biggest element is at the middle.

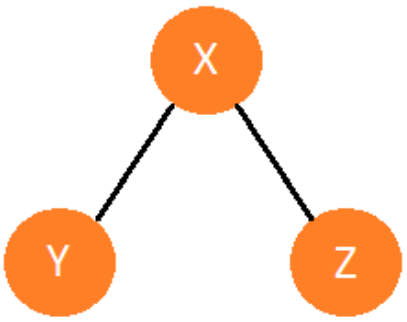
Since we know that X is bigger than Y and Z, the output can not be sorted in descending order because the biggest element is at the middle.

Postorder traversal: Y Z X

For the output to be sorted in ascending order Y should be smaller than Z but this is not guaranteed.

Since we know that X is bigger than Y and Z, the output can not be sorted in descending order since the biggest element comes last.

For Min Heap



Suppose that X, Y and Z represent numbers. From the properties of heaps we know that X is smaller than both Y and Z. However, we can not know whether Y is smaller than or larger than Z.

Preorder traversal: X Y Z

For the output to be sorted in ascending order Y should be smaller than Z but this is not guaranteed.

Since we know that X is smaller than Y and Z, the output can not be sorted in descending order because the smallest element comes first.

Inorder traversal: Y X Z

Since we know that X is smaller than Y and Z, the output can not be sorted in ascending order because the smallest element is at the middle.

Since we know that X is smaller than Y and Z, the output can not be sorted in descending order since the smallest element is at the middle.

Postorder traversal: Y Z X

Since we know that X is smaller than Y and Z, the output can not be sorted in ascending order because the smallest element comes last.

For the output to be sorted in descending order Y should be bigger than Z but this is not guarateed.

d)

For an AVL tree of height h to have minimum number of nodes there should be just one node at level h. At level k there are 2^{k-1} nodes. The tree should be full up to and including level h-1. Number of nodes up to and including level h-1 is

$2^0 + 2^1 + 2^2 + ... + 2^{h-2} = 2^{h-1} - 1$

When we add $2^{h-1} - 1$ with 1 (node at level h) we get 2^{h-1}.

Answer: 2^{h-1}

When we insert 15 to this formula we get 2^{14}.

Answer: 2^{14}

e)

```
int countNodes(root)
  if the root is null
    return 0
  else
    return countNodes(left child of the root) + countNodes(right child of the root) + 1

bool isComplete(root, index, countOfNodes)
  if the root is null
    return true
  else if index is greater than or equal to countOfNodes
    return false
  else
    return isComplete(left child of the root, 2*index + 1, countOfNodes) and
    isComplete(right child of the root, 2*index + 2, countOfNodes)

bool checkMinHeapHelper(root)
  if the tree is empty or the root has no children
    return true
  else if the root has no right child
```

if data of the root is less than the data of the left child of the root

return true

else

return false

else if the data of the root is greater than the data of its right or left child

return false

else

return checkMinHeapHelper(left child of the root) and checkMinHeapHelper (right child of the root)

bool checkMinHeap(root)

return isComplete(root, 0, countNodes(root)) and checkMinHeapHelper(root)

Report

Heap Data Structure

Heap()

This function initializes the private members of the Heap class.

~Heap()

This function deallocates the memory which is used by the heap.

void insert(const int a)

This function first expands the heap then inserts the new element on the last available position then trickles it up to a proper place. I implemented a max heap.

int maximum()

Returns the value of the maximum element present in the heap.

int popMaximum()

Replaces the value of the element at index zero with the value of the element at the last index. Shrinks the heap. Calls heapRebuild function with parameter 0. Returns the value of the popped element.

int insertAndCountComparisons(const int a, int& countOfComparisons)

Same as void insert(const int a). The only difference is that this functions counts the comparisons.

int popMaximumAndCountComparisons(int& countOfComparisons)

Same as int popMaximum(). The only difference is that this functions counts the comparisons.

void printHeap()

Prints the heap in array form.

int* items

Points to the elements of the heap.

int size

Holds the count of the elements in the heap.

void expandHeap()

Expands the heap by one.

void shrinkHeap()

Shrinks the heap by one.

void heapRebuild(int root)

Swaps the root with its children if necessary. Calls itself with the children after swaps.

void heapRebuildAndCountComparisons(int root, int& countOfComparisons)

Same as void heapRebuild(int root). Only difference is that this functions counts the comparisons.

Heapsort Functions and Utility Functions

int heapsort(int* arr, int size)

Inserts the items that are pointed by arr into a heap. Pops each element one by one and puts the value of the popped elements to the arr index size - 1 to 0. At the end, elements pointed by arr are sorted.

double stringToDouble(string str)

Takes a string and converts it to double.

int* fileToArray(string fileName, int& size)

Reads the contents of the file named fileName and counts how many elements it contains into size. Allocates space for the array that will be returned. Reads the file again but this time fills the array. Returns a pointer to the created array.

void arrayToFile(string fileName, int* arr, int size)

Opens a file called fileName. Fills the file line by line with the contents of arr.

Calculation of Number of Comparisons in Heapsort

1. Insert all items into the heap

Maximum number of comparisons required by the k^{th} added item is $\lfloor \log k \rfloor$. Maximum number of comparisons required by heapsort to insert n items into a heap is

$\lfloor \log 1 \rfloor + \lfloor \log 2 \rfloor + ... + \lfloor \log n \rfloor < \lfloor \log n \rfloor + \lfloor \log n \rfloor + ... + \lfloor \log n \rfloor < \log n + \log n + ... + \log n = n \log n$

So heapsort makes O(nlogn) comparisons to insert n items.

2. Pop all items

Maximum number of comparisons required by the k^{th} popped item is $2\lfloor \log k \rfloor$. Maximum number of comparisons required by heapsort to pop n items is

$2\lfloor \log 1 \rfloor + 2\lfloor \log 2 \rfloor + ... + 2\lfloor \log n \rfloor < 2\lfloor \log n \rfloor + 2\lfloor \log n \rfloor + ... + 2\lfloor \log n \rfloor < 2 \log n + 2\log n + ... + 2\log n = 2n\log n$

So heapsort makes O(2nlogn) comparisons to pop n items.

As a result heapsort makes O(3nlogn) comparisons in total.

Theoretical Number of Comparisons vs Actual Number of Comparisons

Data No	Numberof Data	Theoretical Number of Comparisons	Actual Number of Comparisons
1	**1000**	**29897**	**17242**
2	**2000**	**65794**	**38543**
3	**3000**	**103956**	**61301**
4	**4000**	**143589**	**85068**
5	**5000**	**184315**	**109463**

More Precise

Maximum number of comparisons made by heapsort is actually equal to

$\lfloor \log 1 \rfloor + \lfloor \log 2 \rfloor + ... + \lfloor \log n \rfloor + 2\lfloor \log 1 \rfloor + 2\lfloor \log 2 \rfloor + ... + 2\lfloor \log n \rfloor$

$= 3\lfloor \log 1 \rfloor + 3\lfloor \log 2 \rfloor + ... + 3\lfloor \log n \rfloor$

There is no formula for this sum. However, a simple program can be used.

With the help of the simple program belove (next page), a more precise table can be obtained.

Theoretical Number of Comparisons vs Actual Number of Comparisons

Data No	Numberof Data	Theoretical Number of Comparisons	Actual Number of Comparisons
1	**1000**	**23961**	**17242**
2	**2000**	**53892**	**38543**
3	**3000**	**86751**	**61301**
4	**4000**	**119751**	**85068**
5	**5000**	**155466**	**109463**

```cpp
#include <iostream>
using namespace std;
#include <math.h>

double sum(int n)
{
    double sum = 0;

    for(int i = 1; i <= n; i++)
    {
        sum = sum + floor(log2(i));
    }

    return 3 * sum;

}

int main()
{
    cout << sum(1000) << endl;
    cout << sum(2000) << endl;
    cout << sum(3000) << endl;
    cout << sum(4000) << endl;
    cout << sum(5000) << endl;
}
```