<https://github.com/TT00FE39-3001/lecture6>

<https://github.com/seppotk/Datastructures_and_algorithms.git>

<https://binary-tree-visualizer.netlify.app/custom-tree>

# Outline

## Topics

* [Review](https://github.com/TT00FE39-3001/lecture6/blob/main/review.md)
* Binary Trees
* Binary Search Trees
* Heaps

## This Week in Points

* Group Activities (Max 9 points)
* Homework (Max 9 points)
* Peer reviews (Max 7 points)

## Part 1:

* Linear vs Non Linear data structures
* Binary Trees
  + Terminologies
  + Types of Binary Trees
  + Properties
  + Representation of Binary Trees
* [Activity 1](https://github.com/TT00FE39-3001/lecture6/blob/main/activity1)

## Part 2:

* Binary Search Trees
  + Visualization
  + Operations on Binary Search Trees
* [Activity 2](https://github.com/TT00FE39-3001/lecture6/blob/main/activity2)

## Part 3:

* Heaps
  + Visualization
  + Max vs Min Heaps
  + Array implementation of Heaps
  + Applications of Heap Data Structure
* [Activity 3](https://github.com/TT00FE39-3001/lecture6/blob/main/activity3)

**# Outline**

**## Topics**

- [Review](./review.md)

- Binary Trees

- Binary Search Trees

- Heaps

**## This Week in Points**

- Group Activities (Max 9 points)

- Homework (Max 9 points)

- Peer reviews (Max 7 points)

**## Part 1:**

- Linear vs Non Linear data structures

- Binary Trees

  - Terminologies

  - Types of Binary Trees

  - Properties

  - Representation of Binary Trees

- [Activity 1](./activity1)

**## Part 2:**

- Binary Search Trees

  - Visualization

  - Operations on Binary Search Trees

- [Activity 2](./activity2)

**## Part 3:**

- Heaps

  - Visualization

  - Max vs Min Heaps

  - Array implementation of Heaps

  - Applications of Heap Data Structure

- [Activity 3](./activity3)

REVIEW

**# Review**

**### Algorithms: Techniques/Classification**

- Brute Force

  - Linear search

  - Bubble sort

  - Selection sort

- Decrease and conquer

  - Binary Search

  - Insertion sort

- Divide-and-Conquer

  - Quick Sort

  - Merge Sort

- Dynamic Programming

  - Bottom Up: Tabulation

  - Top Down: Memoization

    - 0/1 Knapsack & Staircase Problems

**### Data Structures & Abstract data types (ADT)**

- Data Structures (Physical)

  - Arrays

  - Linked Lists

- ADT (Logical)

  - Linear

    - Queues

    - Stacks

    - Hash Tables

  - Non Linear

    - **\*\*Binary Tree\*\***

    - **\*\*Binary Search Tree\*\***

    - **\*\*Heaps\*\***

**### Analysis of Algorithm Efficiency**

- Big O Complexity

- Average case vs worst case

- Space vs Time

**### Misc**

- FIFO vs LIFO

- Recursion vs Iteration

  - The Top-Down Thought Process

- Logarithms vs Exponential

LINKS

**# Links**

- <https://cpp.sh/>

- <https://www.softwaretestinghelp.com/cpp-tutorials>

- <https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/index.html>

- <https://www.cs.usfca.edu/~galles/visualization/Algorithms.html>

- <https://www.enjoyalgorithms.com/data-structures-and-algorithms-course/>

HOMEWORK

**# Homework**

**## Task 1/3:Videos**

- [Heap - Heap Sort - Heapify - Priority Queues](<https://youtu.be/HqPJF2L5h9U>)

- [Binary Search Tree](<https://youtu.be/vLS-zRCHo-Y>)

**## Task 2/3: Reading**

- [Binary Search Tree C++](<https://www.softwaretestinghelp.com/binary-search-tree-in-cpp/>)

- [Heap Data Structure In C++](<https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/>)

**## Task 3/3: Pre-Lecture**

- [Greedy Method](<https://youtu.be/ARvQcqJ_-NY>)

- [MultiStage Graph](<https://youtu.be/9iE9Mj4m8jk>)

ANSWERS:

- [Heap - Heap Sort - Heapify - Priority Queues](<https://youtu.be/HqPJF2L5h9U>)

51:07

# Heap - Heap Sort - Heapify - Priority Queues

ABDUL BARI

[2:34](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=154s) - Representation of a Binary Tree using Array

[14:15](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=855s) - Heap ( Max & Min )

[16:21](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=981s) - Insert in Heap (Max)

[22:18](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=1338s) - Delete in a Heap (Max)

[30:12](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=1812s) - Heapsort

[41:37](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=2497s) - Heapify

[47:00](https://www.youtube.com/watch?v=HqPJF2L5h9U&t=2820s) - Priority Queues

Very good presentation, small mistakes were made.

- [Binary Search Tree](<https://youtu.be/vLS-zRCHo-Y>)

30:18

# 4.6 Optimal Binary Search Tree (Successful Search Only) - Dynamic Programming

ABDUL BARI

keys and frequensies

this was great!!

**## Task 2/3: Reading**

- [Binary Search Tree C++](<https://www.softwaretestinghelp.com/binary-search-tree-in-cpp/>)

# Binary Search Tree C++: BST Implementation And Operations With Examples

**Detailed Tutorial on Binary Search Tree (BST) In C++ Including Operations, C++ Implementation, Advantages, and Example Programs:**

A Binary Search Tree or BST as it is popularly called is a binary tree that fulfills the following conditions:

1. The nodes that are lesser than the root node which is placed as left children of the BST.
2. The nodes that are greater than the root node that is placed as the right children of the BST.
3. The left and right subtrees are in turn the binary search trees.

This arrangement of ordering the keys in a particular sequence facilitates the programmer to carry out operations like searching, inserting, deleting, etc. more efficiently. If the nodes are not ordered, then we might have to compare each and every node before we can get the operation result.

- [Heap Data Structure In C++](<https://www.softwaretestinghelp.com/avl-trees-and-heap-data-structure-in-cpp/>)

**AVL Tree And Heap Data Structure In C++**

**This Tutorial Provides a Detailed Explanation of AVL Trees and Heap Data Structure In C++ Along with AVL Tree Examples for Better Understanding:**

AVL Tree is a height-balanced binary tree. Each node is associated with a balanced factor which is calculated as the difference between the height of its left subtree and the right subtree.

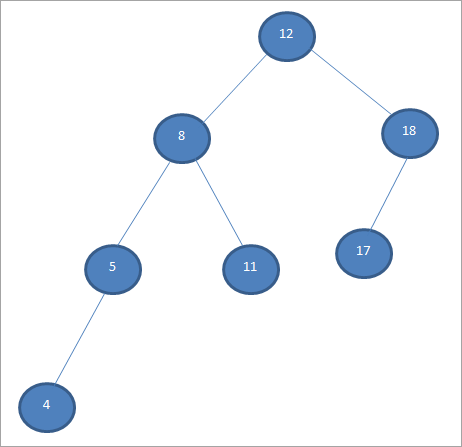
The AVL tree is named after its two inventors i.e. G.M. Abelson-Velvety and E.M. Landis, and was published in 1962 in their paper “An algorithm for the organization of information”.

ABELSON VALVETY LANDIS

## AVL Tree In C++

For the tree to be balanced, the balanced factor for each node should be between -1 and 1. If not the tree will become unbalanced.

**An Example AVL Tree is shown below.**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/An-example-AVL-tree.png)

In the above tree, we can notice that the difference in heights of the left and right subtrees is 1. This means that it’s a balanced BST. As the balancing factor is 1, this means that the left subtree is one level higher than the right subtree.

If the balancing factor is 0, then it means that the left and right subtrees are at the same level i.e. they contain equal height. If the balancing factor is -1, then the left subtree is one level lower than the right subtree.

AVL tree controls the height of a binary search tree and it prevents it from becoming skewed. Because when a binary tree becomes skewed, it is the worst case (O (n)) for all the operations. By using the balance factor, AVL tree imposes a limit on the binary tree and thus keeps all the operations at O (log n).

### AVL Tree Operations

**The following are the operations supported by AVL trees.**

#### **#1) AVL Tree Insertion**

Insert operation in the C++ AVL tree is the same as that of the binary search tree. The only difference is that in order to maintain the balance factor, we need to rotate the tree to left or right so that it doesn’t become unbalanced.

#### **#2) AVL Tree Deletion**

Deletion operation also is performed in the same way as the delete operation in a Binary search tree. Again we need to rebalance the tree by performing some AVL Tree rotations.

### AVL Tree Implementation

**Following is the C++ program to demonstrate the AVL tree and its operations.**

// C++ program for AVL Tree

#include<iostream>

using namespace std;

// An AVL tree node

class AVLNode

{

    public:

    int key;

    AVLNode \*left;

    AVLNode \*right;

    int depth;

};

//get max of two integers

int max(int a, int b){

    return (a > b)? a : b;

}

//function to get height of the tree

int depth(AVLNode \*n)

{

    if (n == NULL)

        return 0;

    return n->depth;

}

// allocate a new node with key passed

AVLNode\* newNode(int key)

{

    AVLNode\* node = new AVLNode();

    node->key = key;

    node->left = NULL;

    node->right = NULL;

    node->depth = 1; // new node added as leaf

    return(node);

}

// right rotate the sub tree rooted with y

AVLNode \*rightRotate(AVLNode \*y)

{

    AVLNode \*x = y->left;

    AVLNode \*T2 = x->right;

    // Perform rotation

    x->right = y;

    y->left = T2;

    // Update heights

    y->depth = max(depth(y->left),

                    depth(y->right)) + 1;

    x->depth = max(depth(x->left),

                    depth(x->right)) + 1;

    // Return new root

    return x;

}

// left rotate the sub tree rooted with x

AVLNode \*leftRotate(AVLNode \*x)

{

    AVLNode \*y = x->right;

    AVLNode \*T2 = y->left;

    // Perform rotation

    y->left = x;

    x->right = T2;

    // Update heights

    x->depth = max(depth(x->left),

                    depth(x->right)) + 1;

    y->depth = max(depth(y->left),

                    depth(y->right)) + 1;

    // Return new root

    return y;

}

// Get Balance factor of node N

int getBalance(AVLNode \*N)

{

    if (N == NULL)

        return 0;

    return depth(N->left) -

           depth(N->right);

}

//insertion operation for node in AVL tree

AVLNode\* insert(AVLNode\* node, int key)  {

    //normal BST rotation

    if (node == NULL)

        return(newNode(key));

    if (key < node->key)

        node->left = insert(node->left, key);

    else if (key > node->key)

        node->right = insert(node->right, key);

    else // Equal keys not allowed

        return node;

    //update height of ancestor node

    node->depth = 1 + max(depth(node->left),  depth(node->right));

    int balance = getBalance(node);        //get balance factor

    // rotate if unbalanced

    // Left Left Case

    if (balance > 1 && key < node->left->key)

        return rightRotate(node);

    // Right Right Case

    if (balance < -1 && key > node->right->key)

        return leftRotate(node);

  // Left Right Case

    if (balance > 1 && key > node->left->key)

    {

        node->left = leftRotate(node->left);

        return rightRotate(node);

    }

    // Right Left Case

    if (balance < -1 && key < node->right->key)

    {

        node->right = rightRotate(node->right);

        return leftRotate(node);

    }

    return node;

}

// find the node with minimum value

AVLNode \* minValueNode(AVLNode\* node)

{

    AVLNode\* current = node;

    // find the leftmost leaf \*/

    while (current->left != NULL)

        current = current->left;

    return current;

}

// delete a node from AVL tree with the given key

AVLNode\* deleteNode(AVLNode\* root, int key)

{

    if (root == NULL)

        return root;

    //perform BST delete

    if ( key < root->key )

        root->left = deleteNode(root->left, key);

    else if( key > root->key )

        root->right = deleteNode(root->right, key);

else

    {

        // node with only one child or no child

        if( (root->left == NULL) ||

            (root->right == NULL) )

        {

            AVLNode \*temp = root->left ?

                         root->left :

                         root->right;

            if (temp == NULL)

            {

                temp = root;

                root = NULL;

            }

            else // One child case

            \*root = \*temp;

            free(temp);

        }

   else

        {

            AVLNode\* temp = minValueNode(root->right);

            root->key = temp->key;

            // Delete the inorder successor

            root->right = deleteNode(root->right,

                                     temp->key);

        }

    }

    if (root == NULL)

    return root;

    // update depth

    root->depth = 1 + max(depth(root->left),

                           depth(root->right));

    // get balance factor

    int balance = getBalance(root);

    //rotate the tree if unbalanced

    // Left Left Case

    if (balance > 1 &&

        getBalance(root->left) >= 0)

        return rightRotate(root);

    // Left Right Case

    if (balance > 1 &&  getBalance(root->left) < 0)  {

        root->left = leftRotate(root->left);

        return rightRotate(root);

    }

    // Right Right Case

    if (balance < -1 &&  getBalance(root->right) <= 0)

        return leftRotate(root);

    // Right Left Case

    if (balance < -1 && getBalance(root->right) > 0)   {

        root->right = rightRotate(root->right);

        return leftRotate(root);

    }

    return root;

}

// prints inOrder traversal of the AVL tree

void inOrder(AVLNode \*root)

{

    if(root != NULL)

    {

        inOrder(root->left);

        cout << root->key << " ";

        inOrder(root->right);

    }

}

  // main code

int main()

{

    AVLNode \*root = NULL;

    // constructing an AVL tree

    root = insert(root, 12);

    root = insert(root, 8);

    root = insert(root, 18);

    root = insert(root, 5);

    root = insert(root, 11);

    root = insert(root, 17);

    root = insert(root, 4);

    //Inorder traversal for above tree : 4 5 8 11 12 17 18

    cout << "Inorder traversal for the AVL tree is: \n";

    inOrder(root);

    root = deleteNode(root, 5);

    cout << "\nInorder traversal after deletion of node 5: \n";

    inOrder(root);

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\AVL\_tree

Inorder traversal for the AVL tree is:

4 5 8 11 12 17 18

Inorder traversal after deletion of node 5:

4 8 11 12 17 18

### Applications Of AVL Trees

1. AVL trees are mostly used for in-memory sorts of sets and dictionaries.
2. AVL trees are also used extensively in database applications in which insertions and deletions are fewer but there are frequent lookups for data required.
3. It is used in applications that require improved searching apart from the database applications**.**

### Binary Heap C++

A binary heap is the common implementation of a heap data structure.

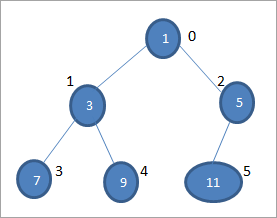
**A binary heap has the following properties:**

* It is a complete binary tree when all the levels are completely filled except possibly the last level and the last level has its keys as much left as possible.
* A binary heap can be a min-heap or max-heap.

A binary heap is a complete binary tree and thus it can best be represented as an array.

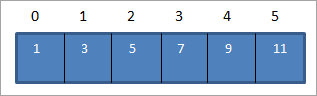
**Let’s look into the array representation of binary heap.**

Consider the following binary heap.

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/binary-heap.png)

In the above diagram, traversal for the binary heap is called level order.

**Thus the array for the above binary heap is shown below as HeapArr:**

[](https://www.softwaretestinghelp.com/wp-content/qa/uploads/2019/08/HeapArr.png)

As shown above, HeapArr[0] is the root of the binary heap. We can represent the other elements in general terms as follows:

**If HeapArr[i] is an ith node in a binary heap, then the indexes of the other nodes from the ith node are:**

* HeapArr [(i-1)/2] => Returns the parent node.
* HeapArr [(2\*i)+1] => Returns the left child node.
* HeapArr [(2\*i)+2] => Returns the right child node.

**The binary heap satisfies “ordering property” which is of two types as stated below:**

* **Min Heap property:**The minimum value is at the root and the value of each node is greater than or equal to its parent.
* **Max Heap property:**The maximum value is at the root and the value of each node is less than or equal to its parent.

### Operations On Binary Heap

The following are the basic operations that are carried out on minimum heap. In the case of the maximum heap, the operations reverse accordingly.

**#1) Insert()** – Inserts a new key at the end of the tree. Depending on the value of the key inserted, we may have to adjust the heap, without violating the heap property.

**#2) Delete()** – Deletes a key. **Note** that the time complexity of both the insert and delete operations of the heap is O (log n).

**#3) decreaseKey()** – Decreases the value of the key. We might need to maintain the heap property when this operation takes place. The time complexity of the decreaseKey operation of the heap is also O (log n).

**#4) extractMin()** – Removes the minimum element from the min-heap. It needs to maintain the heap property after removing the minimum element. Thus its time complexity is O (log n).

**#5) getMin()** – Returns the root element of the min-heap. This is the simplest operation and the time complexity for this operation is O(1).

### Heap Data Structure Implementation

**Given below is the C++ implementation to demonstrate the basic functionality of min-heap.**

#include<iostream>

#include<climits>

using namespace std;

// swap two integers

void swap(int \*x, int \*y)

{

    int temp = \*x;

    \*x = \*y;

    \*y = temp;

}

// Min-heap class

class Min\_Heap

{

    int \*heaparr; // pointer to array of elements in heap

    int capacity; // maximum capacity of min heap

    int heap\_size; // current heap size

public:

    Min\_Heap(int cap){

        heap\_size = 0;

        capacity = cap;

        heaparr = new int[capacity];

    }

    // to heapify a subtree with the root at given index

    void MinHeapify(int );

    int parent(int i) { return (i-1)/2; }

    // left child of node i

    int left(int i) { return (2\*i + 1); }

    // right child of node i

    int right(int i) { return (2\*i + 2); }

    // extract minimum element in the heap(root of the heap)

    int extractMin();

    // decrease key value to newKey at i

    void decreaseKey(int i, int newKey);

    // returns root of the min heap

    int getMin() { return heaparr[0]; }

    // Deletes a key at i

    void deleteKey(int i);

    // Inserts a new key 'key'

    void insertKey(int key);

    void displayHeap(){

        for(int i = 0;i<heap\_size;i++)

            cout<<heaparr[i]<<" ";

        cout<<endl;

    }

};

// Inserts a new key 'key'

void Min\_Heap::insertKey(int key)

{

    if (heap\_size == capacity)  {

        cout << "\nOverflow: Could not insertKey\n";

        return;

    }

    // First insert the new key at the end

    heap\_size++;

    int i = heap\_size - 1;

    heaparr[i] = key;

    // Fix the min heap property if it is violated

    while (i != 0 && heaparr[parent(i)] > heaparr[i])

    {

       swap(&heaparr[i], &heaparr[parent(i)]);

       i = parent(i);

    }

}

void Min\_Heap::decreaseKey(int i, int newKey) {

    heaparr[i] = newKey;

    while (i != 0 && heaparr[parent(i)] > heaparr[i])  {

       swap(&heaparr[i], &heaparr[parent(i)]);

       i = parent(i);

    }

}

  int Min\_Heap::extractMin()

{

    if (heap\_size <= 0)

        return INT\_MAX;

    if (heap\_size == 1)   {

        heap\_size--;

        return heaparr[0];

    }

    // Store the minimum value,delete it from heap

    int root = heaparr[0];

    heaparr[0] = heaparr[heap\_size-1];

    heap\_size--;

    MinHeapify(0);

    return root;

}

void Min\_Heap::deleteKey(int i)

{

    decreaseKey(i, INT\_MIN);

    extractMin();

}

void Min\_Heap::MinHeapify(int i)

{

    int l = left(i);

    int r = right(i);

    int min = i;

    if (l < heap\_size && heaparr[l] < heaparr[i])

        min = l;

    if (r < heap\_size && heaparr[r] < heaparr[min])

        min = r;

    if (min != i)

    {

        swap(&heaparr[i], &heaparr[min]);

        MinHeapify(min);

    }

}

// main program

int main()

{

    Min\_Heap h(11);

    h.insertKey(2);

    h.insertKey(4);

    h.insertKey(6);

    h.insertKey(8);

    h.insertKey(10);

    h.insertKey(12);

    cout<<"Heap after insertion:";

    h.displayHeap();

    cout<<"root of the heap: "<<h.getMin()<<endl;

    h.deleteKey(2);

    cout<<"Heap after deletekey(2):";

    h.displayHeap();

    cout <<"minimum element in the heap: "<< h.extractMin() <<endl;

    h.decreaseKey(1, 1);

    cout <<"new root of the heap after decreaseKey: "<< h.getMin()<<endl;

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\Programs> .\min\_heap

Heap after insertion:2 4 6 8 10 12

root of the heap: 2

Heap after deletekey(2):2 4 12 8 10

minimum element in the heap: 2

new root of the heap after decreaseKey: 1

**## Task 3/3: Pre-Lecture**

- [Greedy Method](<https://youtu.be/ARvQcqJ_-NY>)

# 3. Greedy Method - Introduction

12:01

ABDUL BARI

Introduction to Greedy Method What are Feasible and Optimal Solutions General Method of Greedy Examples to Explain Greedy Method PATREON : [https://www.patreon.com/bePatron?u=20...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbnJSYzNyVTNla01MZVVUNHpUSUdrY3NwYVBMUXxBQ3Jtc0tsYkNmZ25RaGNDNDh5RHlVWWEyUEpMOE9JeUZsVWU5VE4yX0VEOTJZblBncnVHdDBfdDZnT1k5OW0xQUd6a28wWWk1UlJuOHlZTGV6RVNtSmhHQ3pEbFdRVE0xU3FTMTdVZDRMZ0NNWkYxMVU3djVGMA&q=https%3A%2F%2Fwww.patreon.com%2FbePatron%3Fu%3D20475192&v=ARvQcqJ_-NY) Courses on Udemy ================ Java Programming [https://www.udemy.com/course/java-se-...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbUFTanlydml5WWUzYjFzMXMzZ3RtZWhsMHk5UXxBQ3Jtc0tsLWpqUktCazFLSTFvSmRKYVk1VDdZT1NNTDJCSFJUaHdQLUVIR2hrTHZyQ3FJRlpaeDBqbS02aWNsWHQ2ZUhzc3JSall5TV9PR2kwelNjaktfRmc4Q3Fzdl9jUUc0WWNjeDVUb3l5MV9DM0dRNUlZMA&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-se-programming%2F%3FreferralCode%3DC71BADEAA4E7332D62B6&v=ARvQcqJ_-NY) Data Structures using C and C++ [https://www.udemy.com/course/datastru...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa3hOOVRlRXJ4b0xLS3VGVksxLUkzYkJQckk5UXxBQ3Jtc0ttMTZKVXRRTXU5OFd1aWpDOEF5Mk5RcTM4NzlQRjRkZE9SdHhlR0pVeFpQdjktSG9RVVd4SkRFVlU2RFJJb2R3aWx3UVBuNWdkN1g4TjNMc3ZQOHNnRXhUTnlRU1ZsX1JLUktaSjU4aWJIVVNXVmtyOA&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fdatastructurescncpp%2F%3FreferralCode%3DBD2EF8E61A98AB5E011D&v=ARvQcqJ_-NY) C++ Programming [https://www.udemy.com/course/cpp-deep...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqa2FOLWRMVmtSVV9TVXREalBoYzF6eklYMEQ5d3xBQ3Jtc0tuUHY5dEgzX2NIcVJzaXY1MU5CWG9nNElvU1h0X1JXcVQzaUowUDBUX3pOcFN5VDRJaEZoRXNpSTRrYUthRkxtYTFqWGstOFJmMlRLZnhNbmRtcTB2eGk1d1VucjlnUlRidHduVUtBVlR0emdmcXVGbw&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fcpp-deep-dive%2F%3FreferralCode%3DE4246A516919D7E84225&v=ARvQcqJ_-NY)

You find minimum or maximum solution.

conditions 🡪 feasible solution.-- > optimal solution.

- [MultiStage Graph](<https://youtu.be/9iE9Mj4m8jk>)

# 4.1 MultiStage Graph - Dynamic Programming

21:06

Multistage Graph Problem Solved using Dynamic Programming Forward Method PATREON : [https://www.patreon.com/bePatron?u=20...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbVJJR0xURm1TSTRiYjBFXzc1aDV5RUhCT1g4Z3xBQ3Jtc0tuVGg0blVVRlQyOHljV3phMFRUTGlKeEEzcUZvN0c4Wl9OUVZ1d2xLQWhtdnNXNUs0NTJJYnhyMlBVajQzM25DcGl3eXRlQjFOY3c2MmhzLU1mMlVpOHdXM3hIb2hha1NjMXFXb09Xbk5uZ0VhdVpicw&q=https%3A%2F%2Fwww.patreon.com%2FbePatron%3Fu%3D20475192&v=9iE9Mj4m8jk) Courses on Udemy ================ Java Programming [https://www.udemy.com/course/java-se-...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbVpWVHkwdVZQSkgxNUYxZExUTTBTdVB1Wk1DQXxBQ3Jtc0ttdHo0bnRHWUNsclJoMWNDcS1OZ2VGbE16TTVmcG9yWjlxYTlmeXlYTXNyckJJcnhZaElsSHVSXzR4dDJPbDFCSFRGbTlqdC0yRVRZUzFZeS1VdFVjNGF1UnE2cUl3YU1MWU9WMUVqaS15SXYycGpWRQ&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fjava-se-programming%2F%3FreferralCode%3DC71BADEAA4E7332D62B6&v=9iE9Mj4m8jk) Data Structures using C and C++ [https://www.udemy.com/course/datastru...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbHktdUItZ2pjaGxIM2pCU2hLNHFweGFNRmh1d3xBQ3Jtc0trVWVGNHU4Z2ViUHFScG1xeC1LSmxzMzQ3TWhaQW1KZmtuSnFaZ0pnX3NOLVpBRHFXNlJiWmZjd0JjOU9kcmVlMjdiUjZkZjFNQnp3VlRrUUlaOTZ1MDBqRWNPa3M1ZFM5TEZfcHBMbW1qbDl2SVNsbw&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fdatastructurescncpp%2F%3FreferralCode%3DBD2EF8E61A98AB5E011D&v=9iE9Mj4m8jk) C++ Programming [https://www.udemy.com/course/cpp-deep...](https://www.youtube.com/redirect?event=video_description&redir_token=QUFFLUhqbjljWmpPdl9zUkJDZmp6cUIxbW5vN0ZCZTZZZ3xBQ3Jtc0trd1lVdFhRcVdsZVNvNWtHVXd5UnU3SU81R3o5aXVwQ2wzZWZWX2RKQ1BRZkU3MXN4ZVI0MFpiR0ZRVXBmTXMwcHBIUVdGcTBsTi11VVppazZLRU9rVF9TdGJUYmZJOG5wUHJDQmVNV2xGSF9PX3VMQQ&q=https%3A%2F%2Fwww.udemy.com%2Fcourse%2Fcpp-deep-dive%2F%3FreferralCode%3DE4246A516919D7E84225&v=9iE9Mj4m8jk)

ACTIVITY 1

**# Activities**

**## Task 1: Tree Definition**

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/DefSumm.html>

**## Task 2**

- Explain how the code in ./src/bt.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/binary-tree-array-implementation/>

**## Task 3: Tree Traversal**

Refer to te following links. Discuss the difference between the different ways binary trees can be traversed.

- [Pre-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPreorderPRO.html>)

- [Post-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPostorderPRO.html>)

- [In-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravInorderPRO.html>)

**## Task 4: Individual (at home)**

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/TravSumm.html>

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/Treeprobs.html>

**## Links**

- https://cpp.sh/

- <https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BinaryTree.html>

- <https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BinaryTreeTraversal.html>

ANSWERS:

**## Task 1: Tree Definition**

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/DefSumm.html>

**What is the minimum number of nodes in a full binary tree with height 3?**

Diagram

Description automatically generated

ANSWER 7

**Which statement is false?**

* Every binary tree has at least one node
* Every non-root node in a binary tree has exactly one parent
* Every node in a binary tree has exactly two children
* Every non-empty binary tree has exactly one root node
* None of the above

FALSE:

Every binary tree has at least one node

**Select the one true statement.**

* Every binary tree is either complete or full
* Every complete binary tree is also a full binary tree
* Every full binary tree is also a complete binary tree
* No binary tree is both complete and full
* None of the above

TRUE:

None of the above

**What is the minimum number of nodes in a complete binary tree with height 3?**



ANSWER 8 ????

Start filling in the tree level by level.

**Stop when you draw the first node at level 3**

**What is the minimum number of internal nodes in a binary tree with 8 nodes?**



ANSWER 4

**## Task 2**

- Explain how the code in ./src/bt.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/binary-tree-array-implementation/>

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture6-main\activity1\src> .\bt

ABCDE-F---

Binary tee formation, the – marks are empty places.

{

    root('A');

    set\_left('B', 0);

    set\_right('C', 0);

    set\_left('D', 1);

    set\_right('E', 1);

    set\_right('F', 2);

    print\_tree();

    return 0;

}

**Illustration:**

A(0)

/ \

B(1) C(2)

/ \ \

D(3) E(4) F(6)

OR,

A(1)

/ \

B(2) C(3)

/ \ \

D(4) E(5) F(7)

// C++ implementation of tree using array

// numbering starting from 0 to n-1.

#include <iostream>

using namespace std;

char tree[10];

int root(char key)

{

    if (tree[0] != '\0')

        cout << "Tree already had root";

    else

        tree[0] = key;

    return 0;

}

int set\_left(char key, int parent)

{

    if (tree[parent] == '\0')

        cout << "\nCan't set child at "

             << (parent \* 2) + 1

             << " , no parent found";

    else

        tree[(parent \* 2) + 1] = key;

    return 0;

}

int set\_right(char key, int parent)

{

    if (tree[parent] == '\0')

        cout << "\nCan't set child at "

             << (parent \* 2) + 2

             << " , no parent found";

    else

        tree[(parent \* 2) + 2] = key;

    return 0;

}

int print\_tree()

{

    cout << "\n";

    for (int i = 0; i < 10; i++)

    {

        if (tree[i] != '\0')

            cout << tree[i];

        else

            cout << "-";

    }

    return 0;

}

// Driver Code

int main()

{

    root('A');

    set\_left('B', 0);

    set\_right('C', 0);

    set\_left('D', 1);

    set\_right('E', 1);

    set\_right('F', 2);

    print\_tree();

    return 0;

}

**## Task 3: Tree Traversal**

Refer to te following links. Discuss the difference between the different ways binary trees can be traversed.

- [Pre-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPreorderPRO.html>)

- [Post-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPostorderPRO.html>)

- [In-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravInorderPRO.html>)

ANSWERS:

- [Pre-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPreorderPRO.html>)

Text

Description automatically generated

- [Post-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravPostorderPRO.html>)

Text

Description automatically generated

- [In-order Traversal](<https://opendsa-server.cs.vt.edu/OpenDSA/AV/Binary/btTravInorderPRO.html>)

Text

Description automatically generated

**## Task 4: Individual (at home)**

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/TravSumm.html>

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/Treeprobs.html>

ANSWERS;

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer. <https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/TravSumm.html>

Graphical user interface, text, application, email

Description automatically generated

**static** void preorder(BinNode rt) {

**if** (rt == **null**) **return**; *// Empty subtree - do nothing*

visit(rt); *// Process root node*

preorder(rt.left()); *// Process all nodes in left*

preorder(rt.right()); *// Process all nodes in right*

}

*// This is a bad idea*

**static** void preorder2(BinNode rt) {

visit(rt);

**if** (rt.left() != **null**) preorder2(rt.left());

**if** (rt.right() != **null**) preorder2(rt.right());

}

At first it might appear that preorder2 is more efficient than preorder, because it makes only half as many recursive calls (since it won’t try to call on a null pointer). On the other hand, preorder2 must access the left and right child pointers twice as often. The net result is that there is no performance improvement

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text, application, email

Description automatically generated

- Answer at least 5 questions from the following link. Make sure you write the question as well as the answer.

<https://opendsa-server.cs.vt.edu/OpenDSA/Exercises/Binary/Treeprobs.html>

Text

Description automatically generated with medium confidence

**Anything that is not a leaf node is an internal node. Internal nodes might have 1 or 2 children**

Graphical user interface

Description automatically generated with medium confidence

**Nodes have depth.**

Diagram

Description automatically generated

Diagram

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

**Siblings must have the same parent. If a node has two children, then each such child has a sibling.**

A picture containing text

Description automatically generated

ACTIVITY 2

**# Activities**

**## Task 1**

- Refer to the following link. Discuss how Binary Search Trees work:

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

**## Task 2**

- Refer to the following link.

<https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/>

Discuss the Time and Space complexity of Binary Search Tree (BST) operations (Searching, Insertion, Deletion) in the Best case, average case and worst case.

**## Task 3**

- Explain how the code in ./src/bst.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/introduction-to-binary-search-tree-data-structure-and-algorithm-tutorials/>

**## Task 4: Individual (at home)**

- Refer to te following link. Explain how In-order Traversal is used to print a binary search tree.

<https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BinaryTreeTraversal.html#inorder-traversal>

**## Link(s)**

- <https://cpp.sh/>

- <https://www.geeksforgeeks.org/binary-search-tree-data-structure/>

ANSWERS:

**## Task 1**

- Refer to the following link. Discuss how Binary Search Trees work:

<https://www.cs.usfca.edu/~galles/visualization/BST.html>

Graphical user interface, application

Description automatically generated

**## Task 2**

- Refer to the following link.

<https://iq.opengenus.org/time-and-space-complexity-of-binary-search-tree/>

Discuss the Time and Space complexity of Binary Search Tree (BST) operations (Searching, Insertion, Deletion) in the Best case, average case and worst case.

**Complexities of all operations**

The summary is as follows:

| **OPERATION** | **WORST CASE** | **AVERAGE CASE** | **BEST CASE** | **SPACE** |
| --- | --- | --- | --- | --- |
| Search | O(N) | O(logN) | O(1) | O(N) |
| Insert | O(N) | O(logN) | O(1) | O(N) |
| Delete | O(N) | O(logN) | O(N) | O(N) |

**## Task 3**

- Explain how the code in ./src/bst.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/introduction-to-binary-search-tree-data-structure-and-algorithm-tutorials/>

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture6-main\activity2\src> .\bst

20 30 40 50 60 70 80

# Introduction to Binary Search Tree – Data Structure and Algorithm Tutorials

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node’s value and the right child has a value greater than the node’s value. This property is called the BST property and it makes it possible to efficiently search, insert, and delete elements in the tree.

The root of a BST is the node that has the largest value in the left subtree and the smallest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.

[Binary Search Tree](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/) is a node-based binary tree data structure that has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performing, a binary search is very easy.
* The left and right subtree each must also be a binary search tree.    
  There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

Shape

Description automatically generated

Handling approach for Duplicate values in the Binary Search tree:

* You can not allow the duplicated values at all.
* We must follow a consistent process throughout i.e either store duplicate value at the left or store the duplicate value at the right of the root, but be consistent with your approach.
* We can keep the counter with the node and if we found the duplicate value, then we can increment the counter

Below are the various operations that can be performed on a BST:

* [**Insert a node into a BST**](https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/)**:** A new key is always inserted at the leaf. Start searching a key from the root till a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.

// C++ program to insert a node

// in a BST

#include <iostream>

using namespace std;

// Given Node

struct node

{

    int key;

    struct node \*left, \*right;

};

// Function to create a new BST node

struct node \*newNode(int item)

{

    struct node \*temp = (struct node \*)malloc(

        sizeof(struct node));

    temp->key = item;

    temp->left = temp->right = NULL;

    return temp;

}

// Function to insert a new node with

// given key in BST

struct node \*insert(struct node \*node, int key)

{

    // If the tree is empty, return a new node

    if (node == NULL)

        return newNode(key);

    // Otherwise, recur down the tree

    if (key < node->key)

    {

        node->left = insert(node->left, key);

    }

    else if (key > node->key)

    {

        node->right = insert(node->right, key);

    }

    // Return the node pointer

    return node;

}

// Function to do inorder traversal of BST

void inorder(struct node \*root)

{

    if (root != NULL)

    {

        inorder(root->left);

        cout << root->key << " ";

        inorder(root->right);

    }

}

// Driver Code

int main()

{

    /\* Let us create following BST

            50

        /    \

        30   70

        / \ / \

    20 40 60 80

\*/

    struct node \*root = NULL;

    // Inserting value 50

    root = insert(root, 50);

    // Inserting value 30

    insert(root, 30);

    // Inserting value 20

    insert(root, 20);

    // Inserting value 40

    insert(root, 40);

    // Inserting value 70

    insert(root, 70);

    // Inserting value 60

    insert(root, 60);

    // Inserting value 80

    insert(root, 80);

    // Print the BST

    inorder(root);

    return 0;

}

// This code is contributed by shubhamsingh10

**Time Complexity:** O(N), where N is the number of nodes of the BST   
**Auxiliary Space:** O(1)

**## Task 4: Individual (at home)**

- Refer to te following link. Explain how In-order Traversal is used to print a binary search tree.

<https://opendsa-server.cs.vt.edu/OpenDSA/Books/Everything/html/BinaryTreeTraversal.html#inorder-traversal>

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

The letter is select when you visit the letter. Now first letter B is selected.

order:

**B D A G E C H F I**.

ACTIVITY 3

**# Activities**

**## Task 1**

- Refer to the following link. Discuss how Min Heap data structure works:

<https://www.cs.usfca.edu/~galles/visualization/Heap.html>

**## Task 2**

- Refer to the following link.

<https://iq.opengenus.org/time-and-space-complexity-of-heap/>

Discuss the Time and Space Complexity of Heap data structure operations, in the Best case, average case and worst case.

**## Task 3**

- Explain how the code in ./src/priority-queue.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/priority-queue-using-binary-heap/>

**## Task 4: Individual (at home)**

- Explain how the code in ./src/heap.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/binary-heap/>

**## Link(s)**

- <https://cpp.sh/>

- <https://www.geeksforgeeks.org/array-representation-of-binary-heap/>

**## Task 1**

- Refer to the following link. Discuss how Min Heap data structure works:

<https://www.cs.usfca.edu/~galles/visualization/Heap.html>

Diagram

Description automatically generated

**## Task 2**

- Refer to the following link.

<https://iq.opengenus.org/time-and-space-complexity-of-heap/>

Discuss the Time and Space Complexity of Heap data structure operations, in the Best case, average case and worst case.

Complexity of Heap data structure operations including different cases like Worst, Average and Best case. At the end, we have added a table summarizes the complexities.

###### List of Operations

* [Insertion](https://iq.opengenus.org/time-and-space-complexity-of-heap/#insertion)
  + [Best Case: O(1)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#insertionbestcaseo1)
  + [Worst Case: O(logN)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#insertionworstcaseologn)
  + [Average Case: O(logN)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#insertionaveragecaseologn)
* [Deletion](https://iq.opengenus.org/time-and-space-complexity-of-heap/#deletion)
  + [Best Case: O(1)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#deletionbestcaseo1)
  + [Worst Case: O(logN)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#deletionworstcaseologn)
  + [Average Case: O(logN)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#deletionaveragecaseologn)
* [Searching](https://iq.opengenus.org/time-and-space-complexity-of-heap/#searching)
  + [Best Case: O(1)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#searchingbestcaseo1)
  + [Worst Case: O(N)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#searchingworstcaseon)
  + [Average Case: O(N)](https://iq.opengenus.org/time-and-space-complexity-of-heap/#searchingaveragecaseon)

Here's the list of all Time Complexity and Space Complexity we looked at,

|  |  |  |  |
| --- | --- | --- | --- |
| **OPERATION** | **TIME COMPLEXITY** | | **SPACE COMPLEXITY** |
| Insertion | Best Case: | O(1) | O(1) |
| Worst Case: | O(logN) |
| Average Case: | O(logN) |
| Deletion | Best Case: | O(1) | O(1) |
| Worst Case: | O(logN) |
| Average Case: | O(logN) |
| Searching | Best Case: | O(1) | O(1) |
| Worst Case: | O(N) |
| Average Case: | O(N) |
| Max Value | In MaxHeap: | O(1) | O(1) |
| In MinHeap: | O(N) |
| Min Value | In MinHeap: | O(1) | O(1) |
| In MaxHeap: | O(N) |
| Sorting | All Cases: | O(NlogN) | O(1) |
| Creating a Heap | By Inserting all elements: | O(NlogN) | O(N) |
| Using Heapify | O(N) | O(1) |

**## Task 3**

- Explain how the code in ./src/priority-queue.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/priority-queue-using-binary-heap/>

# Priority Queue using Binary Heap

[Priority Queue](https://www.geeksforgeeks.org/priority-queue-set-1-introduction/) is an extension of the [queue](https://www.geeksforgeeks.org/queue-data-structure/) with the following properties:

1. Every item has a priority associated with it.
2. An element with high priority is dequeued before an element with low priority.
3. If two elements have the same priority, they are served according to their order in the queue.

A [Binary Heap](https://www.geeksforgeeks.org/binary-heap/) is a Binary Tree with the following properties:

1. It is a [Complete Tree](https://www.geeksforgeeks.org/check-whether-binary-tree-complete-not-set-2-recursive-solution/). This property of Binary Heap makes them suitable to be stored in an [array](https://www.geeksforgeeks.org/introduction-to-arrays/).
2. A Binary Heap is either **Min Heap** or **Max Heap**.
3. In a **Min Binary Heap**, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in [Binary Tree](https://www.geeksforgeeks.org/binary-tree-data-structure/).
4. Similarly, in a **Max Binary Heap**, the key at the root must be maximum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree.

### Operation on Binary Heap

* **insert(p):** Inserts a new element with priority **p**.
* **extractMax():** Extracts an element with maximum priority.
* **remove(i):** Removes an element pointed by an iterator i.
* **getMax():** Returns an element with maximum priority.
* **changePriority(i, p):** Changes the priority of an element pointed by **i** to **p**.

**Time Complexity:** The time complexity of all the operation is **O(log N)** except for GetMax() which has time complexity of O(1).   
***Auxiliary Space:****O(N)*

// C++ code to implement priority-queue

// using array implementation of

// binary heap

#include <iostream>

using std::cin;

using std::cout;

using std::swap;

int H[50];

int size = -1;

// Function to return the index of the

// parent node of a given node

int parent(int i)

{

    return (i - 1) / 2;

}

// Function to return the index of the

// left child of the given node

int leftChild(int i)

{

    return ((2 \* i) + 1);

}

// Function to return the index of the

// right child of the given node

int rightChild(int i)

{

    return ((2 \* i) + 2);

}

// Function to shift up the node in order

// to maintain the heap property

void shiftUp(int i)

{

    while (i > 0 && H[parent(i)] < H[i])

    {

        // Swap parent and current node

        swap(H[parent(i)], H[i]);

        // Update i to parent of i

        i = parent(i);

    }

}

// Function to shift down the node in

// order to maintain the heap property

void shiftDown(int i)

{

    int maxIndex = i;

    // Left Child

    int l = leftChild(i);

    if (l <= size && H[l] > H[maxIndex])

    {

        maxIndex = l;

    }

    // Right Child

    int r = rightChild(i);

    if (r <= size && H[r] > H[maxIndex])

    {

        maxIndex = r;

    }

    // If i not same as maxIndex

    if (i != maxIndex)

    {

        swap(H[i], H[maxIndex]);

        shiftDown(maxIndex);

    }

}

// Function to insert a new element

// in the Binary Heap

void insert(int p)

{

    size = size + 1;

    H[size] = p;

    // Shift Up to maintain heap property

    shiftUp(size);

}

// Function to extract the element with

// maximum priority

int extractMax()

{

    int result = H[0];

    // Replace the value at the root

    // with the last leaf

    H[0] = H[size];

    size = size - 1;

    // Shift down the replaced element

    // to maintain the heap property

    shiftDown(0);

    return result;

}

// Function to change the priority

// of an element

void changePriority(int i, int p)

{

    int oldp = H[i];

    H[i] = p;

    if (p > oldp)

    {

        shiftUp(i);

    }

    else

    {

        shiftDown(i);

    }

}

// Function to get value of the current

// maximum element

int getMax()

{

    return H[0];

}

// Function to remove the element

// located at given index

void remove(int i)

{

    H[i] = getMax() + 1;

    // Shift the node to the root

    // of the heap

    shiftUp(i);

    // Extract the node

    extractMax();

}

// Driver Code

int main()

{

    /\*       45

            /    \

        31   14

        / \ / \

        13 20 7 11

        / \

    12 7

    Create a priority queue shown in

    example in a binary max heap form.

    Queue will be represented in the

    form of array as:

    45 31 14 13 20 7 11 12 7 \*/

    // Insert the element to the

    // priority queue

    insert(45);

    insert(20);

    insert(14);

    insert(12);

    insert(31);

    insert(7);

    insert(11);

    insert(13);

    insert(7);

    int i = 0;

    // Priority queue before extracting max

    cout << "Priority Queue : ";

    while (i <= size)

    {

        cout << H[i] << " ";

        i++;

    }

    cout << "\n";

    // Node with maximum priority

    cout << "Node with maximum priority : "

         << extractMax() << "\n";

    // Priority queue after extracting max

    cout << "Priority queue after "

         << "extracting maximum : ";

    int j = 0;

    while (j <= size)

    {

        cout << H[j] << " ";

        j++;

    }

    cout << "\n";

    // Change the priority of element

    // present at index 2 to 49

    changePriority(2, 49);

    cout << "Priority queue after "

         << "priority change : ";

    int k = 0;

    while (k <= size)

    {

        cout << H[k] << " ";

        k++;

    }

    cout << "\n";

    // Remove element at index 3

    remove(3);

    cout << "Priority queue after "

         << "removing the element : ";

    int l = 0;

    while (l <= size)

    {

        cout << H[l] << " ";

        l++;

    }

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture6-main\activity3\src> .\priority-queue

Priority Queue : 45 31 14 13 20 7 11 12 7

Node with maximum priority : 45

Priority queue after extracting maximum : 31 20 14 13 7 7 11 12

Priority queue after priority change : 49 20 31 13 7 7 11 12

Priority queue after removing the element : 49 20 31 12 7 7 11

**## Task 4: Individual (at home)**

- Explain how the code in ./src/heap.cpp works. Refer to the following link:

<https://www.geeksforgeeks.org/binary-heap/>

// A C++ program to demonstrate common Binary Heap Operations

#include <iostream>

using namespace std;

// Prototype of a utility function to swap two integers

void swap(int \*x, int \*y);

// A class for Min Heap

class MinHeap

{

    int \*harr;     // pointer to array of elements in heap

    int capacity;  // maximum possible size of min heap

    int heap\_size; // Current number of elements in min heap

public:

    // Constructor

    MinHeap(int capacity);

    // to heapify a subtree with the root at given index

    void MinHeapify(int);

    int parent(int i) { return (i - 1) / 2; }

    // to get index of left child of node at index i

    int left(int i) { return (2 \* i + 1); }

    // to get index of right child of node at index i

    int right(int i) { return (2 \* i + 2); }

    // to extract the root which is the minimum element

    int extractMin();

    // Decreases key value of key at index i to new\_val

    void decreaseKey(int i, int new\_val);

    // Returns the minimum key (key at root) from min heap

    int getMin() { return harr[0]; }

    // Deletes a key stored at index i

    void deleteKey(int i);

    // Inserts a new key 'k'

    void insertKey(int k);

};

// Constructor: Builds a heap from a given array a[] of given size

MinHeap::MinHeap(int cap)

{

    heap\_size = 0;

    capacity = cap;

    harr = new int[cap];

}

// Inserts a new key 'k'

void MinHeap::insertKey(int k)

{

    if (heap\_size == capacity)

    {

        cout << "\nOverflow: Could not insertKey\n";

        return;

    }

    // First insert the new key at the end

    heap\_size++;

    int i = heap\_size - 1;

    harr[i] = k;

    // Fix the min heap property if it is violated

    while (i != 0 && harr[parent(i)] > harr[i])

    {

        swap(&harr[i], &harr[parent(i)]);

        i = parent(i);

    }

}

// Decreases value of key at index 'i' to new\_val. It is assumed that

// new\_val is smaller than harr[i].

void MinHeap::decreaseKey(int i, int new\_val)

{

    harr[i] = new\_val;

    while (i != 0 && harr[parent(i)] > harr[i])

    {

        swap(&harr[i], &harr[parent(i)]);

        i = parent(i);

    }

}

// Method to remove minimum element (or root) from min heap

int MinHeap::extractMin()

{

    if (heap\_size <= 0)

        return INT\_MAX;

    if (heap\_size == 1)

    {

        heap\_size--;

        return harr[0];

    }

    // Store the minimum value, and remove it from heap

    int root = harr[0];

    harr[0] = harr[heap\_size - 1];

    heap\_size--;

    MinHeapify(0);

    return root;

}

// This function deletes key at index i. It first reduced value to minus

// infinite, then calls extractMin()

void MinHeap::deleteKey(int i)

{

    decreaseKey(i, INT\_MIN);

    extractMin();

}

// A recursive method to heapify a subtree with the root at given index

// This method assumes that the subtrees are already heapified

void MinHeap::MinHeapify(int i)

{

    int l = left(i);

    int r = right(i);

    int smallest = i;

    if (l < heap\_size && harr[l] < harr[i])

        smallest = l;

    if (r < heap\_size && harr[r] < harr[smallest])

        smallest = r;

    if (smallest != i)

    {

        swap(&harr[i], &harr[smallest]);

        MinHeapify(smallest);

    }

}

// A utility function to swap two elements

void swap(int \*x, int \*y)

{

    int temp = \*x;

    \*x = \*y;

    \*y = temp;

}

// Driver program to test above functions

int main()

{

    MinHeap h(11);

    h.insertKey(3);

    h.insertKey(2);

    h.deleteKey(1);

    h.insertKey(15);

    h.insertKey(5);

    h.insertKey(4);

    h.insertKey(45);

    cout << h.extractMin() << " ";

    cout << h.getMin() << " ";

    h.decreaseKey(2, 1);

    cout << h.getMin();

    return 0;

}

PS C:\Users\Seppo\Downloads\Metropolia\2023\Datastructures\_and\_algorithms\lecture6-main\activity3\src> .\heap

2 4 1

# Binary Heap

* **Difficulty Level :** [Medium](https://www.geeksforgeeks.org/medium/)
* **Last Updated :** 10 Mar, 2023

 Read

 Discuss(70)

 Courses

 Practice

 Video

*A****Binary Heap****is a*[***complete Binary Tree***](https://www.geeksforgeeks.org/complete-binary-tree/)*which is used to store data efficiently to get the max or min element based on its structure.*

A Binary Heap is either Min Heap or Max Heap. In a Min Binary Heap, the key at the root must be minimum among all keys present in Binary Heap. The same property must be recursively true for all nodes in Binary Tree. Max Binary Heap is similar to MinHeap.

### Examples of Min Heap:

*10                       10  
         /      \                 /         \    
     20     100        15           30    
   /                        /    \         /    \  
30                     40   50   100   40*

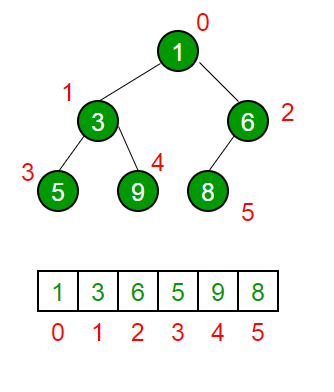
## How is Binary Heap represented?

A Binary Heap is a **Complete Binary Tree**. A binary heap is typically represented as an array.

* The root element will be at Arr[0].
* The below table shows indices of other nodes for the ith node, i.e., Arr[i]:

|  |  |
| --- | --- |
| Arr[(i-1)/2] | Returns the parent node |
| Arr[(2\*i)+1] | Returns the left child node |
| Arr[(2\*i)+2] | Returns the right child node |

The traversal method use to achieve Array representation is[**Level Order Traversal**](https://www.geeksforgeeks.org/level-order-tree-traversal/). Please refer to [Array Representation Of Binary Heap](https://www.geeksforgeeks.org/array-representation-of-binary-heap/) for details.



## Operations on Heap:

Below are some standard operations on min heap:

* **getMin():** It returns the root element of Min Heap. The time Complexity of this operation is **O(1)**. In case of a maxheap it whould be **getMax()**.
* **extractMin():** Removes the minimum element from MinHeap. The time Complexity of this Operation is **O(log N)** as this operation needs to maintain the heap property (by calling **heapify()**) after removing the root.
* **decreaseKey():** Decreases the value of the key. The time complexity of this operation is **O(log N)**. If the decreased key value of a node is greater than the parent of the node, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
* **insert():** Inserting a new key takes **O(log N)** time. We add a new key at the end of the tree. If the new key is greater than its parent, then we don’t need to do anything. Otherwise, we need to traverse up to fix the violated heap property.
* **delete():** Deleting a key also takes **O(log N)** time. We replace the key to be deleted with the minimum infinite by calling **decreaseKey()**. After decreaseKey(), the minus infinite value must reach root, so we call **extractMin()** to remove the key.