City University, Bangladesh

Final Lab Report (Summer 2018)

Course Title: Data Structure
Course Code: CSE213

Submitted By

Rashed Talukder

Id: 171442521

Program: CSE (Day)

Batch: 44th

Submitted To

Richard Philip

Lecturer

Dept. of CSE

City University

Submission Date:31-10-2018

Table of Contents

Data & Structure (Lecture 01 - Introduction to Data Structure, 2017)

What is *Data*?

Data means raw facts or information that can be processed to get results.

What is *Structure*?

Some elementary items constitute a unit and that unit may be considered as a structure.

A structure may be treated as a frame where we organize some elementary items in different ways.

Data Structure

So, what is *Data Structure*?

Data structure is a structure where we organize elementary data items in different ways and there exits structural relationship among the items so that it can be used efficiently.

In other words, a data structure is means of structural relationships of elementary data items for storing and retrieving data in computer's memory.

Data Structure…

Usually elementary data items are the ***elements*** of a data structure.

However, a ***data structure may be an element of another data structure***. That means a data structure may contain another data structure.

Time Complexity

This complexity is related *to execution time* of the algorithm.

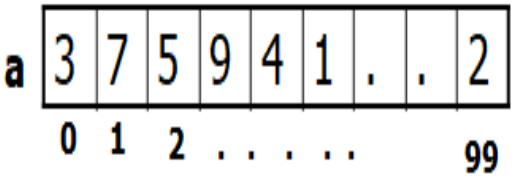
It depends on the number of element*comparisons* and number of element *movement* (movement of data from one place to another).

The first Data Structure - An Array!

An array in C / C++ is a collection of related data elements of the same type that are referenced by a common name and stored in a contiguous memory location.

The simplest form of an Array is a *one dimensional array* that may be defined as a finite ordered set of homogenous elements.

For Example



int a[100];

One Dimensional Array: Declaration

- A single dimensional array declaration has the following form:

data_type array_name[array_size];

Array Initialization

●In line initialization of an array may takes the following form:type array_name[size] = {element_list};

Two-Dimensional Array

●Some data fit better in a table with several rows and columns.

● This can be constructed by using two-dimensional arrays using two subscripts/indices. The first refers to the row, and the second, to the column.

datatype array_name[1st dimn size][2nd dimn size];

Example:

int rows = 4, cols = 3;

int exams [rows] [cols];

Structure

●The array takes simple data types like int, char or double and organizes them into a linear array of elements all of the same type.

●Now, consider a record card which records *name*, *age* and *salary*. The name would have to be stored as a *string*, the age could be int and salary could be float. As this record is about one person, it would be best if they are all stored under one variable.

●At the moment the only way we can work with this collection of data is as separate variables. This isn't as convenient as a single data structure using a single name and so the C language provides *structure*.

●A *structure* is an aggregate data type built using elements of other types.

Defining Structure in C

In general "structure" in C/C++ is defined as follows:

```
struct name{

list of component variables

};
```

For example

```
struct EmployeeRecord{

char name[5];

int age; float salary; } ;
```

Stack

●A stack or LIFO (last in, first out) is an abstract data type that serves as a collection of elements, with two principal operations:

push adds an element to the collection;

pop removes the last (top of the stack) element that was added. (Lecture - 06 - Stack)

Bounded capacity

If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state.

A pop either reveals previously concealed items or results in an empty stack – which means no items are present in stack to be removed.

Non-Bounded capacity

Dynamically allocate memory for stack. No overflow.

Stack (Lecture - 06 - Stack)

int Stack[100], Top=0, MaxSize=100; // Stack holds the elements;

// Top is the index of Stack always pointing to the first/top element of the stack.

bool IsEmpty(); // returns True if stack has no element

bool IsFull(); // returns True if stack full

void Push(int Element); // inserts Element at the top of the stack

int Pop(); // deletes top element from stack into Element

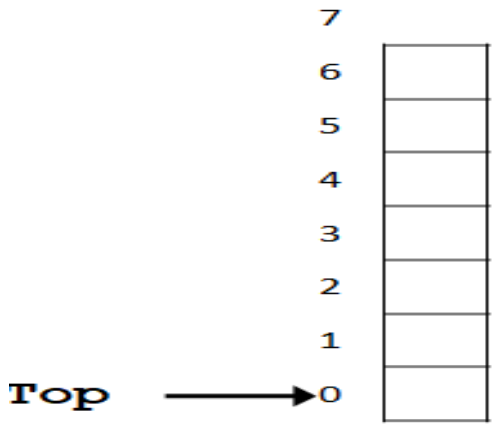
int TopElement(); // gives the top element in Element

void Display(); // prints the whole stack

●**bool** IsEmpty(){

// returns True if stack has no element

return (Top == 0);}Considering **MaxSize** = 7

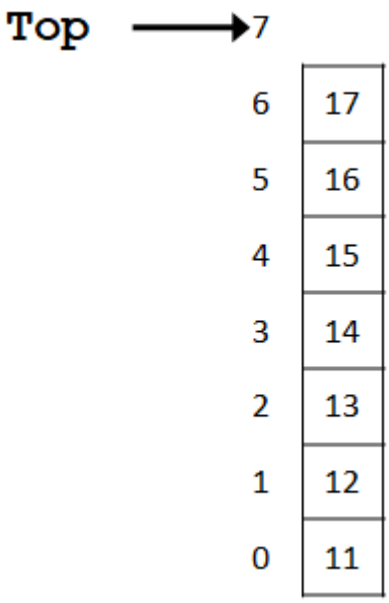


●**bool** IsFull(){

// returns True if stack full

return (Top == MaxSize);

}Considering **MaxSize** = 7



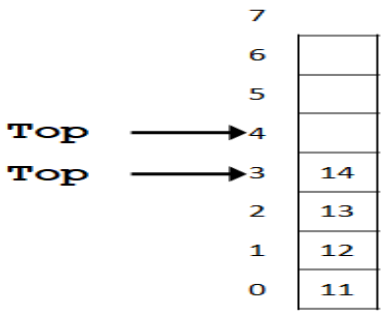
●void push(int data){

head++;

stack[head] = data;

}Considering **MaxSize** = 7

There are 3 elements inside Stack



So next element will be pushed at index 3

●int pop(){

int data = stack[head];

head--;

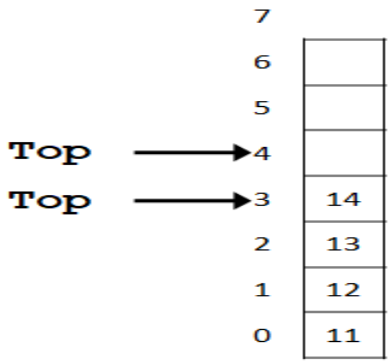
return data;

}

Considering **MaxSize** = 7

There are 4 elements inside Stack

So element will be popped from index 3



●**int** TopElement(){

// gives the top element in Element

return Stack[Top - 1];

}Considering **MaxSize** = 7

There are 4 elements inside Stack

So top element will be at index 3

●void display(){

printf("Data in your stack\n");

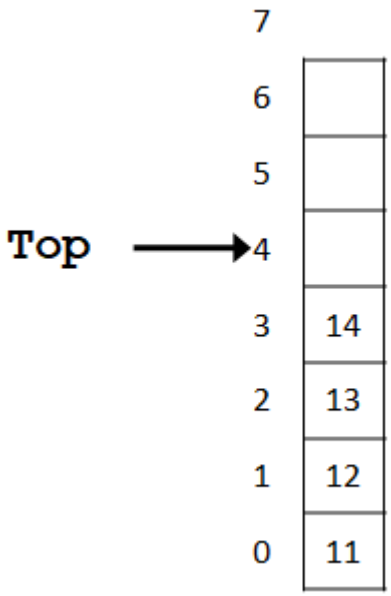
for(int i=0;i<=head;i++){

printf("%d",stack[i]); }}

Considering **MaxSize** = 7

There are 4 elements inside Stack

So element will be shown from index 3 down to index 0.



Queue (Lecture 07 - Queue)

●A queue is a waiting line – seen in daily life

●A line of people waiting for a bank teller

- A line of cars at a toll both
- Queue data structure is like a container with both end opening.

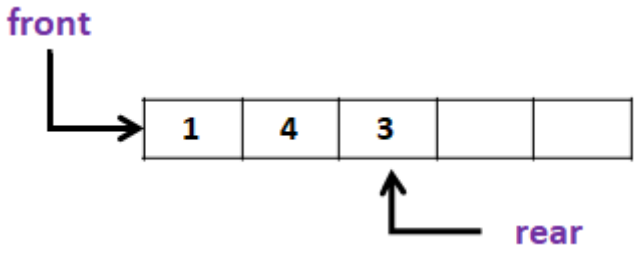
  An end called **rear**

  Another end is **front**
- This mechanism is called First-In-First-Out (FIFO).
- Some of the applications are :

  Device queue, printer queue, keystroke queue, etc.

Queue in Computer Language

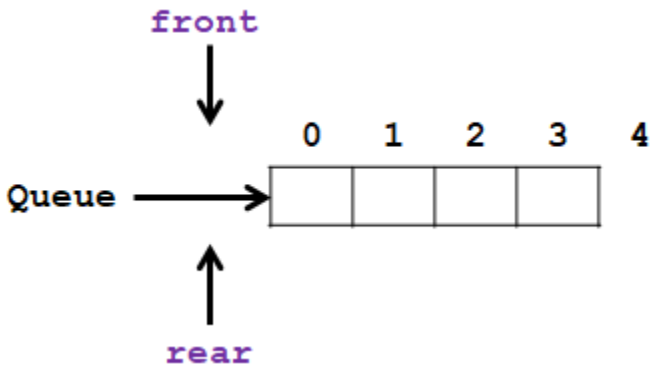
- A queue is a sequence of data elements
- In the sequence
  - Items can be added only at the **rear**
  - Items can be removed only at the other end, **front**
- Basic operations
  - Check **IsEmpty**
  - Check **IsFull**
  - **EnQueue** (add element to back i.e. at the rear)
  - **DeQueue** (remove element from the front)
  - **FrontValue** (retrieve value of element from front)
  - **ShowQueue** (print all the values of queue from front to rear)



Queue – Operation (Lecture 07 - Queue)

Queue[4],MaxSize=4;

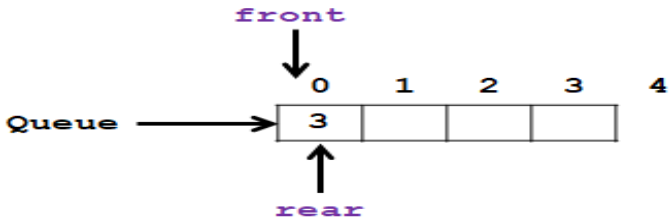
Initialize( )→front=rear=-1



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0



Queue[4],MaxSize=4;

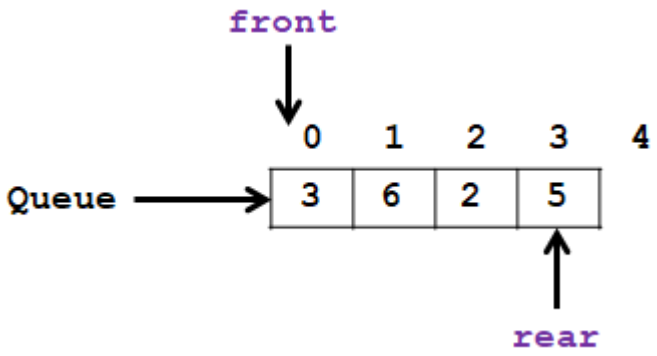
Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0

EnQueue(6)

EnQueue(2)

EnQueue(5)



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

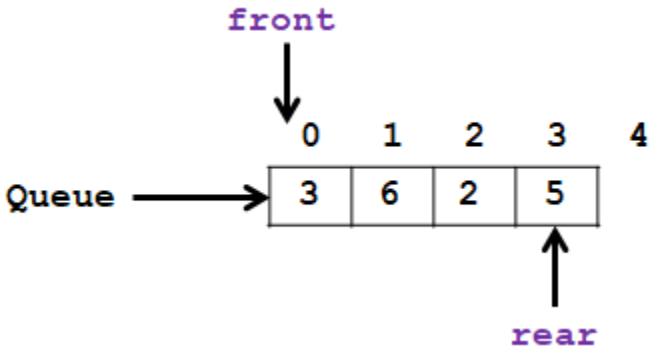
EnQueue(3) → front=rear=0

EnQueue(6)

EnQueue(2)

EnQueue(5)

EnQueue(9) → Queue Full, (rear==(MaxSize-1))



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0

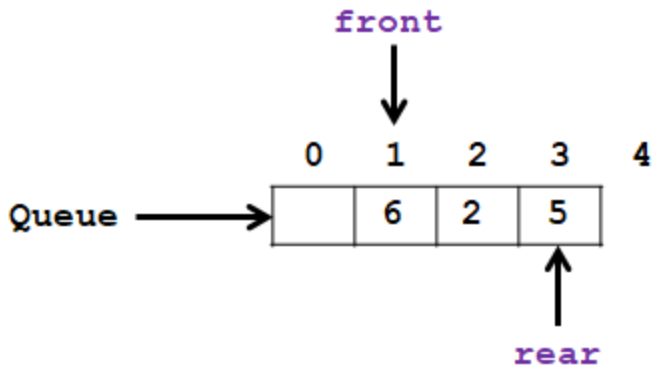
EnQueue(6)

EnQueue(2)

EnQueue(5)

EnQueue(9) → Queue Full, (rear==(MaxSize-1))

DeQueue() → 3



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0

EnQueue(6)

EnQueue(2)

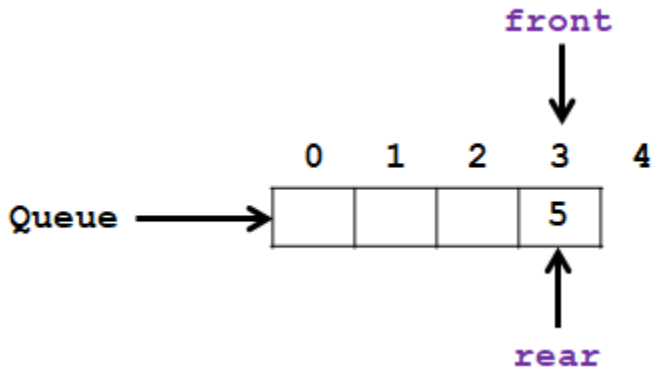
EnQueue(5)

EnQueue(9) → Queue Full, (rear==(MaxSize-1))

DeQueue() → 3

DeQueue() → 6

DeQueue() → 2



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0

EnQueue(6)

EnQueue(2)

EnQueue(5)

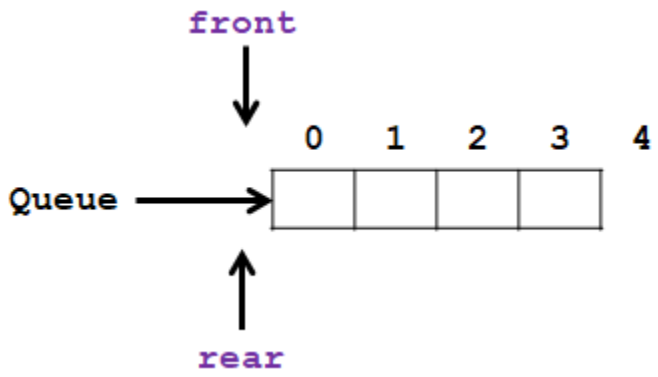
EnQueue(9) → Queue Full, (rear==(MaxSize-1))

DeQueue() → 3

DeQueue() → 6

DeQueue() → 2

DeQueue() → 5, front=rear=-1



Queue[4],MaxSize=4;

Initialize( ) → front=rear=-1

EnQueue(3) → front=rear=0

EnQueue(6)

EnQueue(2)

EnQueue(5)

EnQueue(9) → Queue Full, (rear==(MaxSize-1))

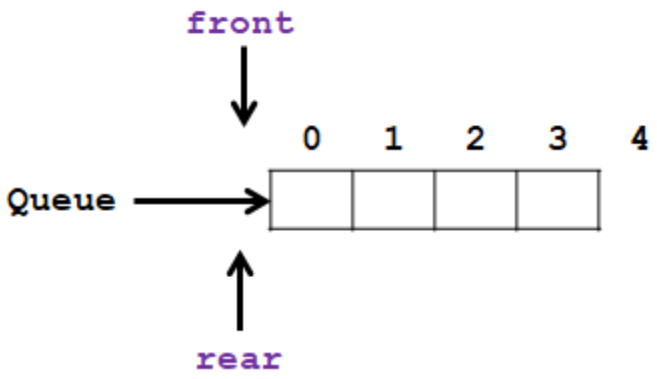
DeQueue() → 3

DeQueue() → 6

DeQueue() → 2

DeQueue() → 5, front=rear=-1

DeQueue() → Queue Empty, (front==-1) && (rear==-1)



Linked List (Lecture 09 - LinkedList)

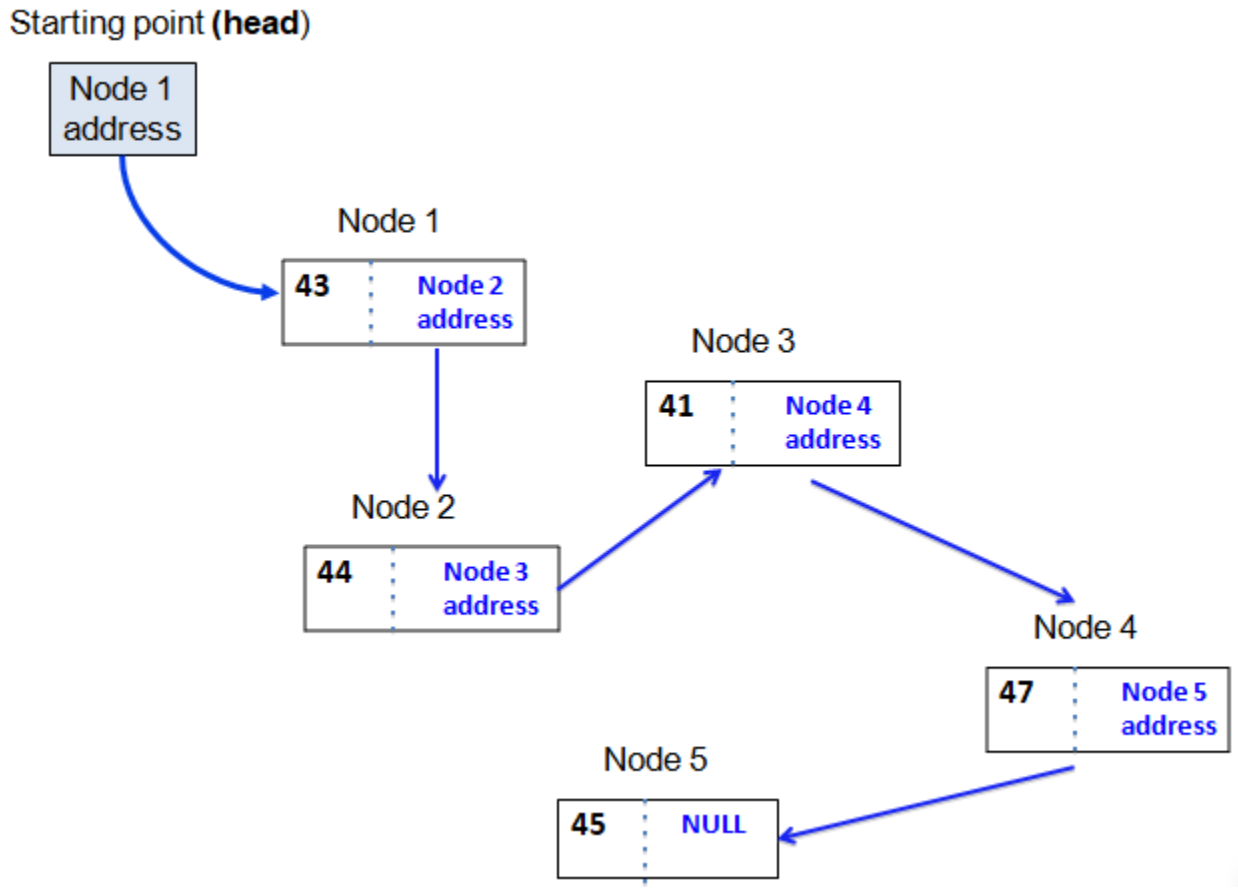
♦Linked list is a data structure consisting of a group of memory space which together represent a list i.e. a sequence of data.

♦A sequence of data can also be represented as an array. But in an array, data are stored consecutively in the memory

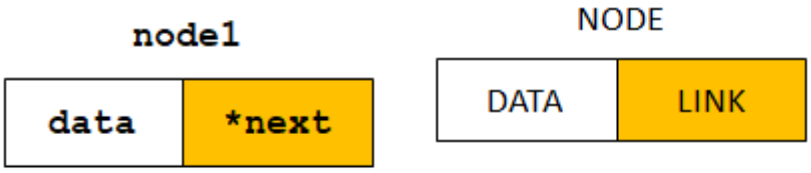
♦A linked list is also a sequence of data. But in a linked list the data are not stored consecutively in the memory.

♦Each data is stored in a separate memory space/block (called cell/node).

♦Each memory block contains the data along with link/location/address to the memory location for the next data in the list.



Representation – node



Representation of a NODE in C/C++

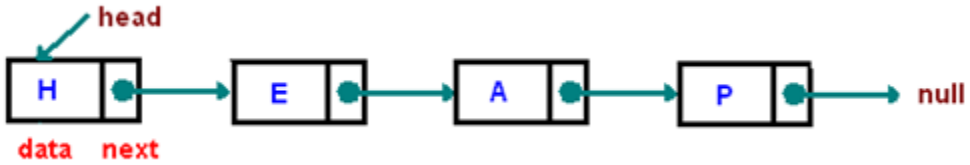
- **struct ListNode{ //Each NODE of the list contains- int data; The Data item**

ListNode *next; **A LINK/POINTER to store the next**

}; **nodes address.**

ListNode node1;

Basics of Linked Lists



- A **group of nodes**.
- **Each node** represents by a block of memory and it has two fields/parts:
    1. **data**/value and
    2. a next **pointer** to point the **next** node.

Type of Linked list

1. Singly linked list (that has single references)
2. Doubly linked list (that has two references)
3. Multiply linked list (that has more than two references)
4. Circular list (tail element's next pointer points to the head element)

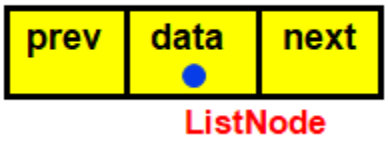
Two-way linked lists

- One that can be traversed in both directions – forward and backward
- Every node has two pointers:
    - One pointing to the next node (except the last node)
    - One pointing to the previous node (except the first node)
- Two external pointers - head and tail

Two way or Doubly Linked Lists (Lecture 10 - Doubly Linked List)

Each node of the list contains

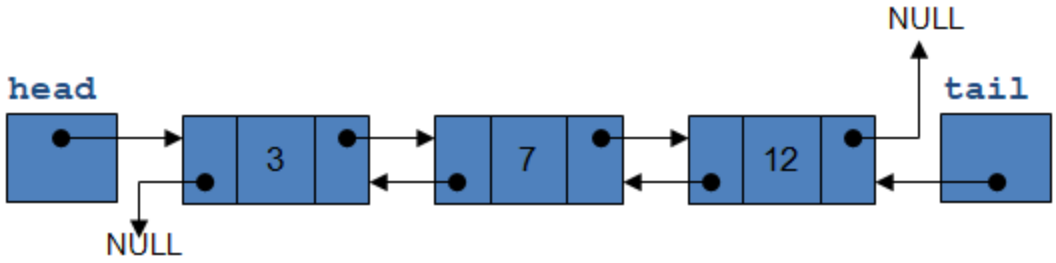
- the data item (an object)
- a pointer to the next node
- a pointer to the previous node




```
typedef struct ListNode {

        int data;
        struct ListNode* next;
        struct ListNode* prev;
}*nodeptr;
```

Two-way linked list of integers



Data Structure – Tree (Data Structure_Tree Mushfiqur Rahman)

- Many applications are hierarchical in nature.
- Linear data structures are not appropriate for these type of applications.

Common Use of Tree as a Data Structure

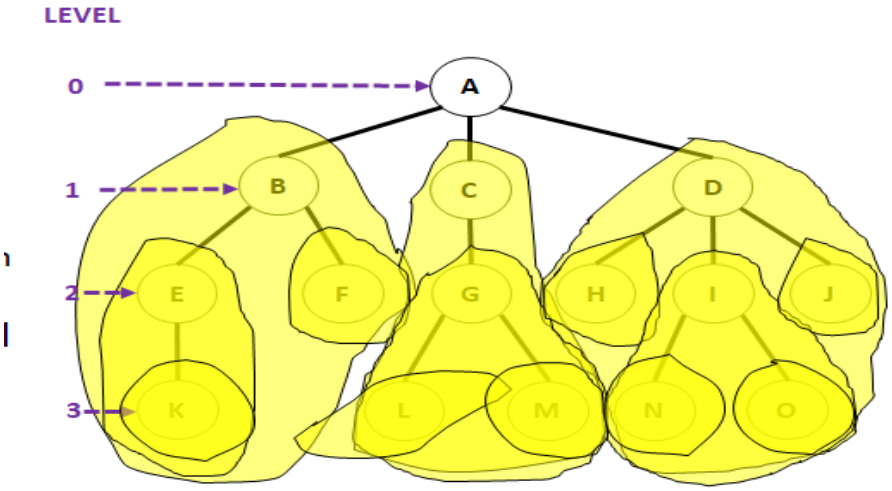
- Representing **hierarchical data**
- Storing data in a way that makes it **easily searchable**
- Representing **sorted lists** of data
- As a workflow for **compositing digital images** for visual effects
- **Routing** algorithms

Tree - Definition

- As a data structure, a tree consists of **one or more nodes**, where each node has a value and a list of references to other (*its children*) nodes.
- A tree must have a node designated as **root**. If a tree has only one node, that node is the root node. Root is never referenced by any other node.
- Having more than one node indicates that the root have some (at least one) references to its children nodes and the children nodes (might) have references to their children nodes and so on.
- Generally it is considered that in a tree, there is only one path going from one node to another node.
- There cannot be any cycle or loop.
- The link from a node to other node is called an edge.
- **Nodes**
- **Parent Nodes**&**Child Nodes**
- **Leaf Nodes**: nodes with no child
- **Root Node**: node with no parent
- **Sub Tree**: the tree rooted by a child
- **Level** of a tree:

- Root at level 0;
- Each children have the level one more than its parent.

- **Height/depth** of the tree: Total number of Levels
- **Height of a node**: Total number of levels from bottom [Tree height – node level].



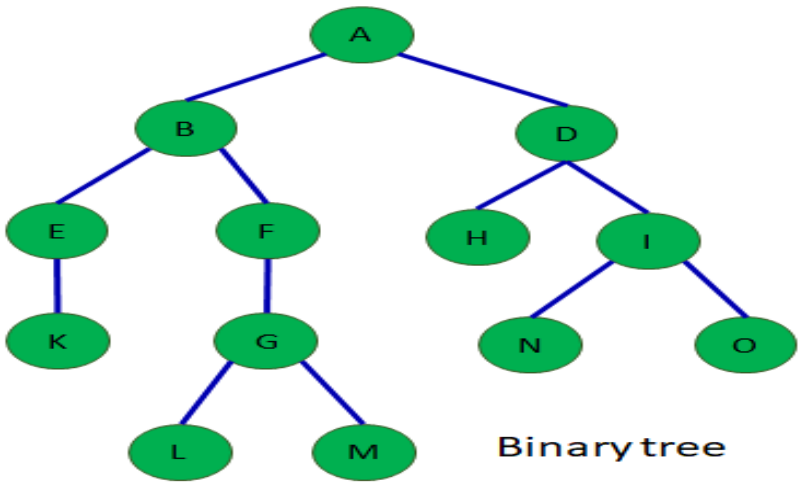
Height of root A is 4;Height of nodes B, C, D is 3;Height of E, F, G, H, I, J is 2;

Height of nodes K, L , M, N, O is 1.

Height of this tree is 4, as there are four levels (0…3).

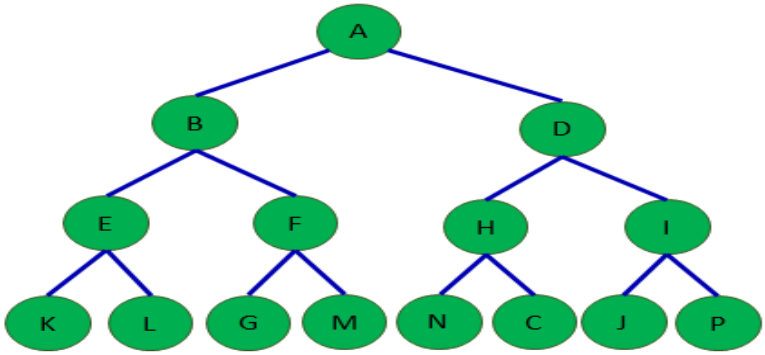
Binary Tree (BT)

- Each node of a binary Tree has at most 2 children.



Complete and Full Binary Tree

- A complete/full binary tree is a binary tree, which is completely filled with nodes from top to bottom and left to right.
- The tree starts from the root (top), goes to the next level with first the left child and then the right child (left to right). The process repeats for each next level with each node till the last (bottom) level.
- In **complete binary tree** some of the nodes only for the bottom level might be absent (*here nodes after **N***).
- In **Full binary tree** the last/bottom level must also be filled up.



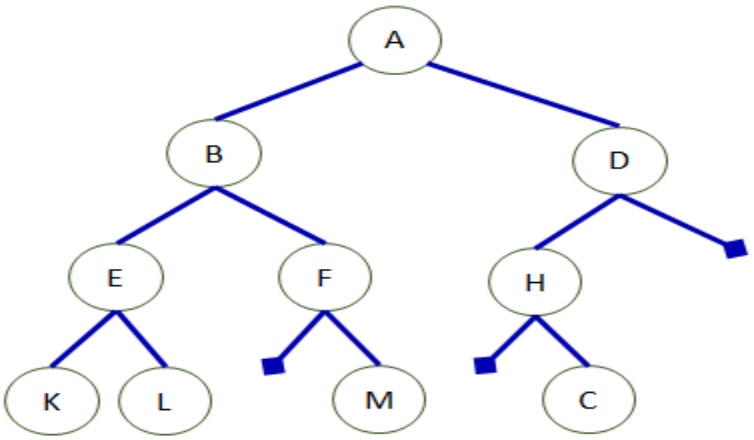
Traversal (10_BST)

- Systematic way of visiting all the nodes.

- Methods:
  - In-order
  - Post-order
  - Preorder
- They all traverse the left subtree before the right subtree.

In-order Traversal – Left-Root-Right

- Traverse the left subtree
- Visit the parent node
- Traverse the right subtree

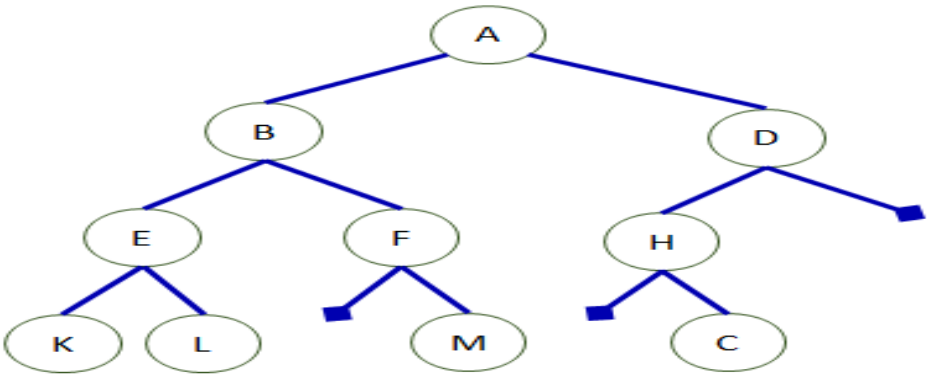


K E L B F M A H C D

Post-order Traversal – Left-Right-Root

- Traverse the left subtree
- Traverse the right subtree

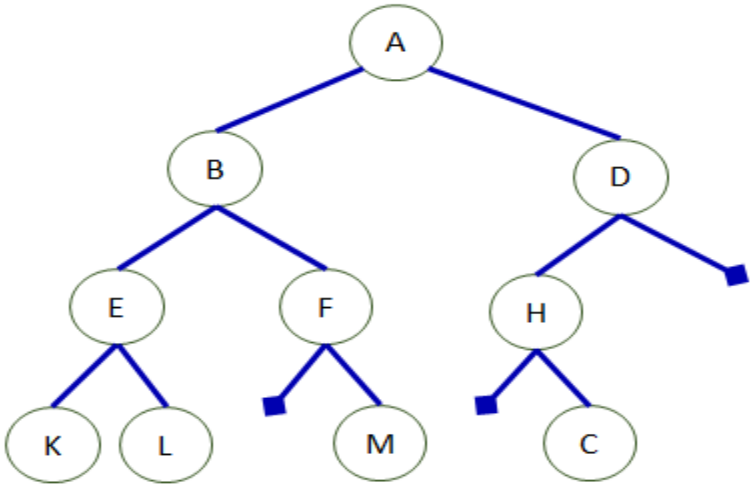
- Visit the parent node



K L E M F B C H D A

Preorder Traversal – Root-Left-Right

- Visit the parent node
- Traverse the left subtree
- Traverse the right subtree



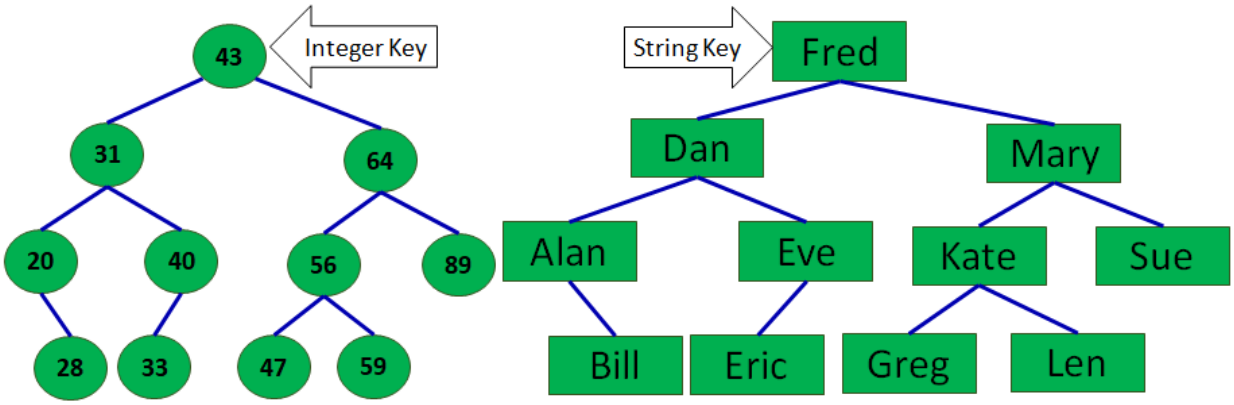
A B E K L F M D H C

Binary Search Tree (BST) (10_BST)

Is a Binary Tree such that:

Every node entry has a unique key (i.e. no duplication item).

- All the keys in the left subtree of a node are less than the key of the node.
- All the keys in the right subtree of a node are greater than the key of the node.



Max Heap

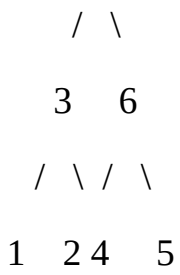
Given a Binary Search Tree which is also a Complete Binary Tree. The problem is to convert a given BST into a Max Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Max Heap.

Examples:

Input: 4

```
      /  \
     2    6
    / \  / \
   1  3 5   7
```

Output: 7



The given **BST** has been transformed into a **Max Heap.**

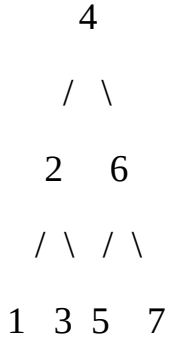
All the nodes in the Max Heap satisfies the given condition, that is, values in the left sub tree of a node should be less than the values in the right sub tree of the node.

Min Heap

Given a binary search tree which is also a complete binary tree. The problem is to convert the given BST into a Min Heap with the condition that all the values in the left subtree of a node should be less than all the values in the right subtree of the node. This condition is applied on all the nodes in the so converted Min Heap.

Examples:

Input:



Output:

2 5

/ \ / \

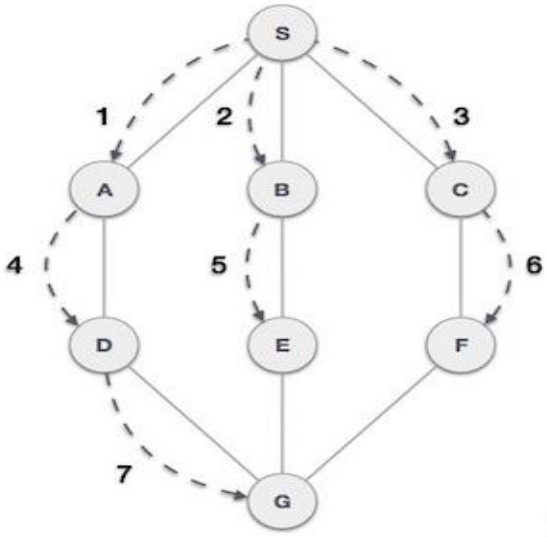
3 4 6 7

The given **BST** has been transformed into aMin Heap.

All the nodes in the Min Heap satisfies the givencondition, that is, values in the left subtree ofa node should be less than the values in the rightsubtree of the node.

Breadth First Search (BFS)

Breadth First Search (BFS) algorithm traverses a graph in a Breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.



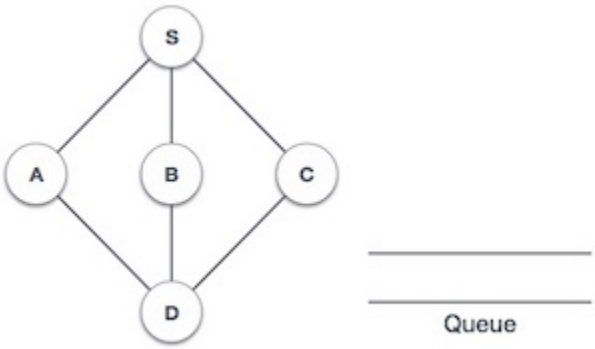
As in the example given above, BFS algorithm traverses from A to B to E to F first then to C and G lastly to D. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

- **Rule 2** − If no adjacent vertex is found, remove the first vertex from the queue.
- **Rule 3** − Repeat Rule 1 and Rule 2 until the queue is empty.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

The implementation of this algorithm in C programming language can be seen below,We shall not see the implementation of Breadth First Traversal (or Breadth First Search) in C programming language. For our reference purpose, we shall follow our example and take this as our graph model –



Implementation in C

```
#include <stdio.h>

#include <stdlib.h>

#include <stdbool.h>


#define MAX 5


struct Vertex {

char label;
```

```
bool visited;
};

//queue variables

int queue[MAX];
int rear = -1;
int front = 0;
intqueueItemCount = 0;

//graph variables

//array of vertices
struct Vertex* lstVertices[MAX];

//adjacency matrix
intadjMatrix[MAX][MAX];

//vertex count
intvertexCount = 0;

//queue functions
```

```
void insert(int data) {
queue[++rear] = data;
queueItemCount++;
}

intremoveData() {
queueItemCount--;
return queue[front++];
}

boolisQueueEmpty() {
returnqueueItemCount == 0;
}

//graph functions

//add vertex to the vertex list
voidaddVertex(char label) {
struct Vertex* vertex = (struct Vertex*) malloc(sizeof(struct Vertex));
vertex->label = label;
vertex->visited = false;
```

```
lstVertices[vertexCount++] = vertex;
}

//add edge to edge array
voidaddEdge(intstart,int end) {
adjMatrix[start][end] = 1;
adjMatrix[end][start] = 1;
}

//display the vertex
voiddisplayVertex(intvertexIndex) {
printf("%c ",lstVertices[vertexIndex]->label);
}

//get the adjacent unvisited vertex
intgetAdjUnvisitedVertex(intvertexIndex) {
inti;

for(i = 0; i<vertexCount; i++) {
if(adjMatrix[vertexIndex][i] == 1 &&lstVertices[i]->visited == false)
returni;
}
```

```
return -1;
}

voidbreadthFirstSearch() {
inti;

//mark first node as visited
lstVertices[0]->visited = true;

//display the vertex
displayVertex(0);

//insert vertex index in queue
insert(0);
intunvisitedVertex;

while(!isQueueEmpty()) {
//get the unvisited vertex of vertex which is at front of the queue
inttempVertex = removeData();

//no adjacent vertex found
```

```
while((unvisitedVertex = getAdjUnvisitedVertex(tempVertex)) != -1) {
lstVertices[unvisitedVertex]->visited = true;
displayVertex(unvisitedVertex);
insert(unvisitedVertex);
}

}

//queue is empty, search is complete, reset the visited flag
for(i = 0;i<vertexCount;i++) {
lstVertices[i]->visited = false;
}
}

int main() {
inti, j;

for(i = 0; i<MAX; i++) // set adjacency {
for(j = 0; j<MAX; j++) // matrix to 0
adjMatrix[i][j] = 0;
}
```

```
addVertex('S');   // 0
addVertex('A');   // 1
addVertex('B');   // 2
addVertex('C');   // 3
addVertex('D');   // 4

addEdge(0, 1);    // S - A
addEdge(0, 2);    // S - B
addEdge(0, 3);    // S - C
addEdge(1, 4);    // A - D
addEdge(2, 4);    // B - D
addEdge(3, 4);    // C - D

printf("\nBreadth First Search: ");

breadthFirstSearch();

return 0;
}
```

If we compile and run the above program, it will produce the following result −

Output:

Breadth First Search: S A B C D

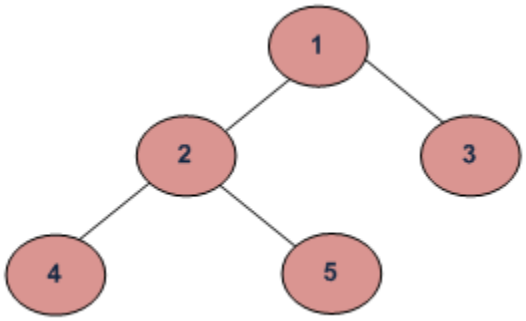
Depth First Search (DFS)

Depth First Traversals

In-order Traversal (Left-Root-Right)

Pre-order Traversal (Root-Left-Right)

Post-order Traversal (Left-Right-Root)



DFS of given Tree

Depth First Traversals:

Pre-order Traversal: 1 2 4 5 3

In-order Traversal: 4 2 5 1 3

Post-order Traversal: 4 5 2 3 1

Implementation in C:

```
#include <stdio.h>

#include <stdlib.h>


struct node {
```

```
int data;
struct node* left;
struct node* right;
};

voidinorder(struct node* root){
if(root == NULL) return;
inorder(root->left);
printf("%d ->", root->data);
inorder(root->right);
}

void preorder(struct node* root){
if(root == NULL) return;
printf("%d ->", root->data);
preorder(root->left);
preorder(root->right);
}

voidpostorder(struct node* root) {
if(root == NULL) return;
postorder(root->left);
```

```
postorder(root->right);
printf("%d ->", root->data);
}


struct node* createNode(value){
struct node* newNode = malloc(sizeof(struct node));
newNode->data = value;
newNode->left = NULL;
newNode->right = NULL;

returnnewNode;
}

struct node* insertLeft(struct node *root, int value) {
root->left = createNode(value);
return root->left;
}


struct node* insertRight(struct node *root, int value){
root->right = createNode(value);
```

```
return root->right;
}
int main(){
struct node* root = createNode(1);
insertLeft(root, 12);
insertRight(root, 9);

insertLeft(root->left, 5);
insertRight(root->left, 6);

insertLeft(root->right, 10);
insertRight(root->right, 12);

insertRight(root->left->left, 20);
insertRight(root->left->right, 30);


printf("In-order traversal \n");
inorder(root);

printf("\nPre-order traversal \n");
preorder(root);
```

```
printf("\nPost-order traversal \n");
postorder(root);
}
```

Output of this program will be:

In-order traversal

5 ➔20 ➔12 ➔6 ➔30 ➔1 ➔10 ➔9 ➔12 ➔

Pre-order traversal

1 ➔12 ➔5 ➔20 ➔6 ➔30 ➔9 ➔10 ➔12 ➔

Post-order traversal

20 ➔5 ➔30 ➔6 ➔12 ➔10 ➔12 ➔9 ➔1 ➔

References

10_BST. Mushfiqur Rahman mushfiqur@aiub.edu.

Data Structure__Tree Mushfiqur Rahman. mushfiqur@aiub.edu.

GeeksforGeeks. (n.d.). Retrieved from https://www.geeksforgeeks.org/convert-bst-to-max-heap/

GeeksforGeeks. (n.d.). Retrieved from https://www.geeksforgeeks.org/convert-bst-min-heap/

Lecture - 06 - Stack. Mushfiqur Rahman___mushfiqur@aiub.edu.

(2017). *Lecture 01 - Introduction to Data Structure.* Mushfiqur Rahman mushfiqur@aiub.edu.

Lecture 07 - Queue . Mushfiqur Rahman .

Lecture 09 - LinkedList. Mushfiqur Rahman.

Lecture 10 - Doubly Linked List. Mushfiqur Rahman mushfiqur@aiub.edu.