DHABALESWAR INSTITUTE OF POLYTECHNIC,ATHGARH,CUTTACK

DATA STRUCTURE USING C LECTURE NOTES

3RD SEMESTER(WINTER)

DEPT. OF COMPUTER SCIENCE & ENGG.

NAME OF THE TEACHER : RASMAN KUMAR SANTI

Module-1

Lecture-01

Introduction to Data structures

In computer terms, a data structure is a Specific way to store and organize data in a computer's memory so that these data can be used efficiently later. Data may be arranged in many different ways such as the logical or mathematical model for a particular organization of data is termed as a data structure. The variety of a particular data model depends on the two factors -

- Firstly, it must be loaded enough in structure to reflect the actual relationships of the data with the real world object.
- Secondly, the formation should be simple enough so that anyone can efficiently process the data each time it is necessary.

Categories of Data Structure:

The data structure can be sub divided into major types:

- Linear Data Structure
- Non-linear Data Structure

Linear Data Structure:

A data structure is said to be linear if its elements combine to form any specific order. There are basically two techniques of representing such linear structure within memory.

- First way is to provide the linear relationships among all the elements represented by means of linear memory location. These linear structures are termed as arrays.
- The second technique is to provide the linear relationship among all the elements represented by using the concept of pointers or links. These linear structures are termed as linked lists.

The common examples of linear data structure are:

- Arrays
- Queues
- Stacks
- Linked lists

Non linear Data Structure:

This structure is mostly used for representing data that contains a hierarchical relationship among various elements.

Examples of Non Linear Data Structures are listed below:

- Graphs
- family of trees and
- table of contents

Tree: In this case, data often contain a hierarchical relationship among various elements. The data structure that reflects this relationship is termed as rooted tree graph or a tree.

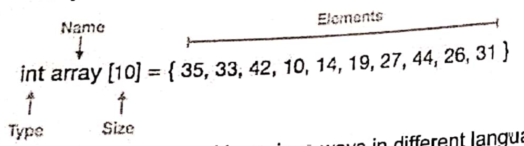
Graph: In this case, data sometimes hold a relationship between the pairs of elements which is not necessarily following the hierarchical structure. Such data structure is termed as a Graph.

Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

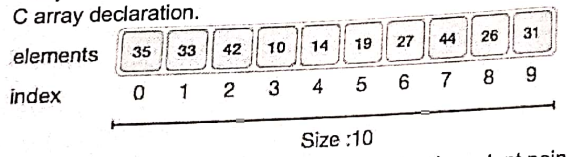
- **Element** – Each item stored in an array is called an element.
- **Index** – Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation:(Storage structure)

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

- Index starts with 0.
- Array length is 10 which means it can store 10 elements.
- Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

- **Traverse** – print all the array elements one by one.
- **Insertion** – Adds an element at the given index.
- **Deletion** – Deletes an element at the given index.
- **Search** – Searches an element using the given index or by the value.
- **Update** – Updates an element at the given index.

In C, when an array is initialized with size, then it assigns defaults values to it elements in following order.

Data Type	Default Value
bool	false
char	0
int	0
float	0.0
double	0.0f
void	
wchar_t	0

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array –

Algorithm

Let **LA** be a Linear Array (unordered) with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm where ITEM is inserted into the K^th^ position of LA –

```
1. Start
2. Set J = N
3. Set N = N+1
4. Repeat steps 5 and 6 while J >= K
5. Set LA[J+1] = LA[J]
6. Set J = J-1
7. Set LA[K] = ITEM
8. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>

main() {
  int LA[] = {1,3,5,7,8};
  int item = 10, k = 3, n = 5;
  int i = 0, j = n;
  printf("The original array elements are :\n");
  for(i = 0; i<n; i++) {
     printf("LA[%d] = %d \n", i, LA[i]);
  }
  n = n + 1;
  while( j >= k) {
     LA[j+1] = LA[j];
     j = j - 1;
  }
  LA[k] = item;
  printf("The array elements after insertion :\n");
  for(i = 0; i<n; i++) {
     printf("LA[%d] = %d \n", i, LA[i]);
  }
}
```

When we compile and execute the above program, it produces the following result –

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after insertion :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 10
LA[4] = 7
LA[5] = 8
```

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

```
1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set LA[J] = LA[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop
```

Example

Following is the implementation of the above algorithm –

```
#include <stdio.h>

void main() {
   int LA[] = {1,3,5,7,8};
   int k = 3, n = 5;
   int i, j;
     printf("The original array elements are :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }

   j = k;
   while( j < n) {
      LA[j-1] = LA[j];
      j = j + 1;
   }
   n = n -1;
     printf("The array elements after deletion :\n");
   for(i = 0; i<n; i++) {
      printf("LA[%d] = %d \n", i, LA[i]);
   }
}
```

When we compile and execute the above program, it produces the following result –

Output

```
The original array elements are :
LA[0] = 1
LA[1] = 3
LA[2] = 5
LA[3] = 7
LA[4] = 8
The array elements after deletion :
LA[0] = 1
LA[1] = 3
LA[2] = 7
LA[3] = 8
```

Lecture-03

Sparse Matrix and its representations

A matrix is a two-dimensional data object made of m rows and n columns, therefore having total m x n values. If most of the elements of the matrix have **0 value**, then it is called a sparse matrix.

Why to use Sparse Matrix instead of simple matrix ?

- **Storage:** There are lesser non-zero elements than zeros and thus lesser memory can be used to store only those elements.
- **Computing time:** Computing time can be saved by logically designing a data structure traversing only non-zero elements..

Example:

```
0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
```

Representing a sparse matrix by a 2D array leads to wastage of lots of memory as zeroes in the matrix are of no use in most of the cases. So, instead of storing zeroes with non-zero elements, we only store non-zero elements. This means storing non-zero elements with **triples- (Row, Column, value).**

Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

Method 1: Using Arrays

```
#include<stdio.h>
 int main()
{
   // Assume 4x5 sparse matrix
   int sparseMatrix[4][5] =
   {
      {0 , 0 , 3 , 0 , 4 },
      {0 , 0 , 5 , 7 , 0 },
      {0 , 0 , 0 , 0 , 0 },
      {0 , 2 , 6 , 0 , 0 }
   };

   int size = 0;
   for (int i = 0; i < 4; i++)
      for (int j = 0; j < 5; j++)
         if (sparseMatrix[i][j] != 0)
            size++;
   int compactMatrix[3][size];
   // Making of new matrix
```

```
        int k = 0;
        for (int i = 0; i < 4; i++)
            for (int j = 0; j < 5; j++)
                if (sparseMatrix[i][j] != 0)
                {
                    compactMatrix[0][k] = i;
                    compactMatrix[1][k] = j;
                    compactMatrix[2][k] = sparseMatrix[i][j];
                    k++;
                }
      for (int i=0; i<3; i++)
      {
         for (int j=0; j<size; j++)
            printf("%d ", compactMatrix[i][j]);
         printf("\n");
      }
      return 0;
}
```

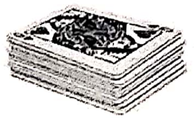
$$\begin{bmatrix} 0 & 0 & 3 & 0 & 4 \\ 0 & 0 & 5 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 6 & 0 & 0 \end{bmatrix} \Rightarrow$$

Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Lecture-04

STACK

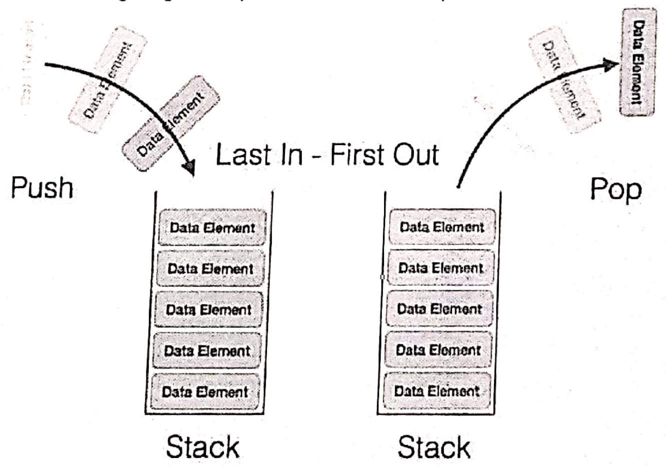
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.



A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.
This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

Stack Representation

The following diagram depicts a stack and its operations –



A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

Basic Operations

Stack operations may involve initializing the stack, using it and then de-initializing it. Apart from these basic stuffs, a stack is used for the following two primary operations –

- **push()** – Pushing (storing) an element on the stack.

- **pop()** – Removing (accessing) an element from the stack.

When data is PUSHed onto stack.

To use a stack efficiently, we need to check the status of stack as well. For the same purp
the following functionality is added to stacks –

- **peek()** – get the top data element of the stack, without removing it.
- **isFull()** – check if stack is full.
- **isEmpty()** – check if stack is empty.

At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer al
represents the top of the stack, hence named **top**. The **top** pointer provides top value o
stack without actually removing it.

First we should learn about procedures to support stack functions –

peek()

Algorithm of peek() function –

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function –

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code –

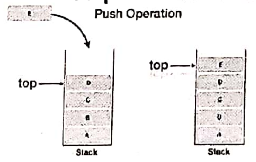
Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps –

- **Step 1** – Checks if the stack is full.
- **Step 2** – If the stack is full, produces an error and exit.
- **Step 3** – If the stack is not full, increments **top** to point next empty space.
- **Step 4** – Adds data element to the stack location, where top is pointing.
- **Step 5** – Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows –

```
begin procedure push: stack, data

   if stack is full
```

```
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

Implementation of this algorithm in C, is very easy. See the following code –

Example

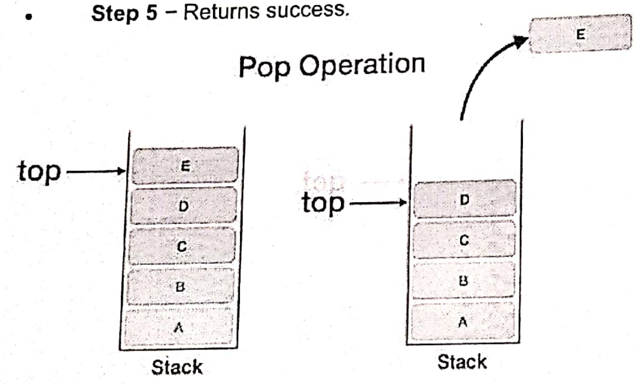
```
void push(int data) {
   if(!isFull()) {
      top = top + 1;
      stack[top] = data;
   } else {
      printf("Could not insert data, Stack is full.\n");
   }
}
```

Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. array implementation of pop() operation, the data element is not actually rem instead **top** is decremented to a lower position in the stack to point to the next value. linked-list implementation, pop() actually removes data element and deallocates memory s

A Pop operation may involve the following steps –

- **Step 1** – Checks if the stack is empty.
- **Step 2** – If the stack is empty, produces an error and exit.
- **Step 3** – If the stack is not empty, accesses the data element at which **top** is pointir
- **Step 4** – Decreases the value of top by 1.
- **Step 5** – Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows –

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

Implementation of this algorithm in C, is as follows –

Example

```
int pop(int data) {

   if(!isempty()) {
      data = stack[top];
      top = top - 1;
      return data;
   } else {
      printf("Could not retrieve data, Stack is empty.\n");
   }
}
```

Lecture-05

Stack Applications

Three applications of stacks are presented here. These examples are central to many acti that a computer must do and deserve time spent with them.

1. Expression evaluation
2. Backtracking (game playing, finding paths, exhaustive searching)
3. Memory management, run-time environment for nested language features.

Expression evaluation

In particular we will consider arithmetic expressions. Understand that there are boolea logical expressions that can be evaluated in the same way. Control structures can al treated similarly in a compiler.

This study of arithmetic expression evaluation is an example of problem solving where you a simpler problem and then transform the actual problem to the simpler one.

Aside: *The NP-Complete problem*. There are a set of apparently intractable problems: fi the shortest route in a graph (Traveling Salesman Problem), bin packing, linear program etc. that are similar enough that if a polynomial solution is ever found (exponential sol abound) for one of these problems, then the solution can be applied to all problems.

Infix, Prefix and Postfix Notation

We are accustomed to write arithmetic expressions with the operation between the two operands: a+b or c/d. If we write a+b*c, however, we have to apply precedence rules to av the ambiguous evaluation (add first or multiply first?).

There's no real reason to put the operation between the variables or values. They can just well precede or follow the operands. You should note the advantage of prefix and postfix: t need for precedence rules and parentheses are eliminated.

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c		
40 - 3 * 5 + 1		

Postfix expressions are easily evaluated with the aid of a stack.

Infix, Prefix and Postfix Notation KEY

Infix	Prefix	Postfix
a + b	+ a b	a b +
a + b * c	+ a * b c	a b c * +
(a + b) * (c - d)	* + a b - c d	a b + c d - *
b * b - 4 * a * c	- * b b * * 4 a c	b b * 4 a * c * -
40 - 3 * 5 + 1 = 26	+ - 40 * 3 5 1	40 3 5 * - 1 +

Postfix Evaluation Algorithm

Assume we have a string of operands and operators, an informal, by hand process is

1. Scan the expression left to right
2. Skip values or variables (operands)
3. When an operator is found, apply the operation to the preceding two operands
4. Replace the two operands and operator with the calculated value (three symbols are replaced with one operand)
5. Continue scanning until only a value remains--the result of the expression

The time complexity is O(n) because each operand is scanned once, and each operation is performed once.

A more formal algorithm:

```
create a new stack
while(input stream is not empty){
  token = getNextToken();
  if(token instanceof operand){
     push(token);
  } else if (token instance of operator)
     op2 = pop();
     op1 = pop();
     result = calc(token, op1, op2);
     push(result);
  }
}
return pop();
```

Demonstration with 2 3 4 + * 5 -

Infix transformation to Postfix

This process uses a stack as well. We have to hold information that's expressed inside parentheses while scanning to find the closing ')'. We also have to hold information on operations that are of lower precedence on the stack. The algorithm is:

1. Create an empty stack and an empty postfix output string/stream
2. Scan the infix input string/stream left to right
3. If the current input token is an operand, simply append it to the output string (note the examples above that the operands remain in the same order)
4. If the current input token is an operator, pop off all operators that have equal or higher

precedence and append them to the output string; push the operator onto the stack. order of popping is the order in the output.

5. If the current input token is '(', push it onto the stack
6. If the current input token is ')', pop off all operators and append them to the output st until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the outp string.

This algorithm doesn't handle errors in the input, although careful analysis of parenthesis c of parenthesis could point to such error determination.

Apply the algorithm to the above expressions.

Backtracking

Backtracking is used in algorithms in which there are steps along some path (state) from starting point to some goal.

- Find your way through a maze.
- Find a path from one point in a graph (roadmap) to another point.
- Play a game in which there are moves to be made (checkers, chess).

In all of these cases, there are choices to be made among a number of options. We need way to remember these decision points in case we want/need to come back and t alternative

Consider the maze. At a point where a choice is made, we may discover that the choice to a dead-end. We want to retrace back to that decision point and then try the other alternative.

Again, stacks can be used as part of the solution. Recursion is another, typically more fa solution, which is actually implemented by a stack.

Memory Management

Any modern computer environment uses a stack as the primary memory management moc a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the cer the run-time environment for Java, C++, Ada, FORTRAN, etc.

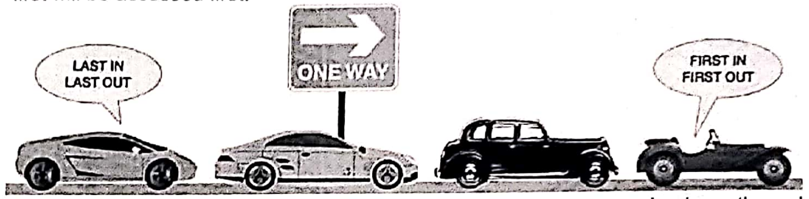
The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix ru environments.

Each program that is running in a computer system has its own memory allocation cont the typical layout as shown below.

<u>Lecture-06</u>

QUEUE

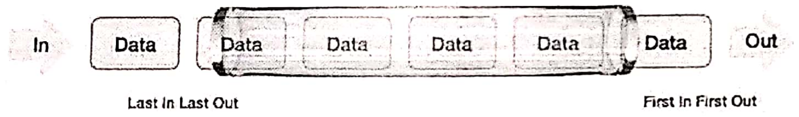
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

Queue Representation

As we now understand that in queue, we access both ends for different reasons. The following diagram given below tries to explain queue representation as data structure –



Queue

As in stacks, a queue can also be implemented using Arrays, Linked-lists, Pointers and Structures. For the sake of simplicity, we shall implement queues using one-dimensional array.

Basic Operations

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues –

- **enqueue()** – add (store) an item to the queue.
- **dequeue()** – remove (access) an item from the queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- **peek()** – Gets the element at the front of the queue without removing it.
- **isfull()** – Checks if the queue is full.
- **isempty()** – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by **front** pointer and while enqueing (or storing) data in the queue we take help of **rear** pointer.

Let's first learn about supportive functions of a queue –

peek()

This function helps to see the data at the **front** of the queue. The algorithm of peek() func
as follows –

Algorithm

```
begin procedure peek
   return queue[front]
end procedure
```

Implementation of peek() function in C programming language –

Example

```
int peek() {
   return queue[front];
}
```

isfull()

As we are using single dimension array to implement queue, we just check for the rear p
to reach at MAXSIZE to determine that the queue is full. In case we maintain the queu
circular linked-list, the algorithm will differ. Algorithm of isfull() function –

Algorithm

```
begin procedure isfull

   if rear equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language –

Example

```
bool isfull() {
   if(rear == MAXSIZE - 1)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function –

Algorithm

```
begin procedure isempty

   if front is less than MIN  OR front is greater than rear
      return true
   else
      return false
   endif

end procedure
```

If the value of **front** is less than MIN or 0, it tells that the queue is not yet initialized, hence empty.

Here's the C programming code –

Example

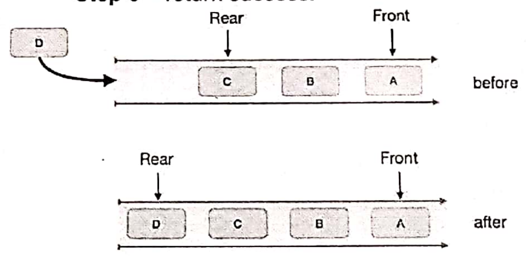
```
bool isempty() {
   if(front < 0 || front > rear)
      return true;
   else
      return false;
}
```

Enqueue Operation

Queues maintain two data pointers, **front** and **rear**. Therefore, its operations are comparatively difficult to implement than that of stacks.

The following steps should be taken to enqueue (insert) data into a queue –

- **Step 1** – Check if the queue is full.
- **Step 2** – If the queue is full, produce overflow error and exit.
- **Step 3** – If the queue is not full, increment **rear** pointer to point the next empty space.
- **Step 4** – Add data element to the queue location, where the rear is pointing.
- **Step 5** – return success.



Queue Enqueue

Sometimes, we also check to see if a queue is initialized or not, to handle any unforesee situations.

Algorithm for enqueue operation

```
procedure enqueue(data)

   if queue is full
      return overflow
```

```
  endif

  rear ← rear + 1
  queue[rear] ← data
  return true

end procedure
```

Implementation of enqueue() in C programming language −

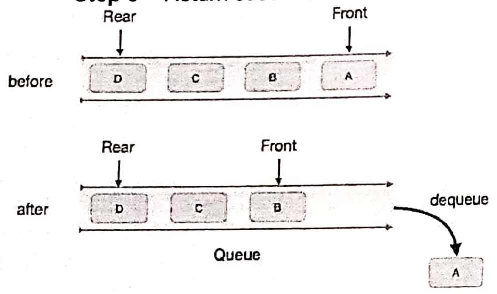
Example

```
int enqueue(int data)
  if(isfull())
    return 0;

  rear = rear + 1;
  queue[rear] = data;

  return 1;
end procedure
```

Dequeue Operation

Accessing data from the queue is a process of two tasks − access the data where t pointing and remove the data after access. The following steps are tak perform **dequeue** operation −

- **Step 1** − Check if the queue is empty.
- **Step 2** − If the queue is empty, produce underflow error and exit.
- **Step 3** − If the queue is not empty, access the data where **front** is pointing.
- **Step 4** − Increment **front** pointer to point to the next available data element.
- **Step 5** − Return success.



Queue Dequeue

Algorithm for dequeue operation

```
procedure dequeue

  if queue is empty
    return underflow
  end if

  data = queue[front]
  front ← front + 1
  return true

end procedure
```

Implementation of dequeue() in C programming language −

Example

```
int dequeue() {
  if(isempty())
    return 0;

  int data = queue[front];
  front = front + 1;

  return data;
}
```

Lecture-07

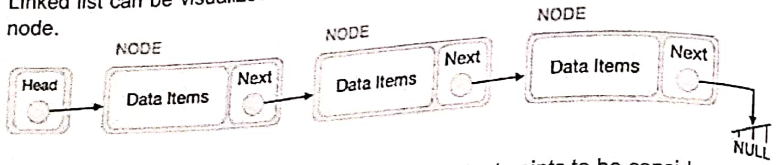
LINKED LIST

A linked list is a sequence of data structures, which are connected together via links. Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array. Following are the important terms to understand the concept of Linked List.

- **Link** – Each link of a linked list can store a data called an element.
- **Next** – Each link of a linked list contains a link to the next link called Next.
- **LinkedList** – A Linked List contains the connection link to the first link called First.

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.



As per the above illustration, following are the important points to be considered.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

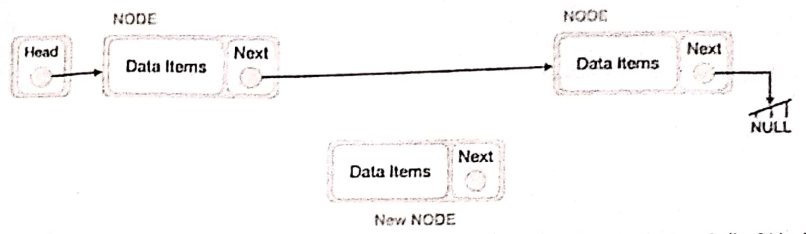
Basic Operations

Following are the basic operations supported by a list.

- **Insertion** – Adds an element at the beginning of the list.
- **Deletion** – Deletes an element at the beginning of the list.
- **Display** – Displays the complete list.
- **Search** – Searches an element using the given key.
- **Delete** – Deletes an element using the given key.

Insertion Operation

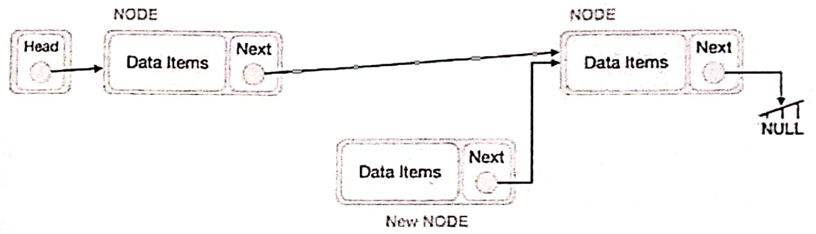
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



Imagine that we are inserting a node **B** (NewNode), between **A** (LeftNode) and **C** (RightNode). Then point B.next to C –

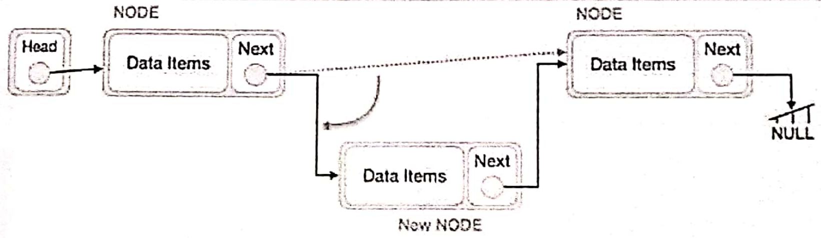
```
NewNode.next -> RightNode;
```

It should look like this –

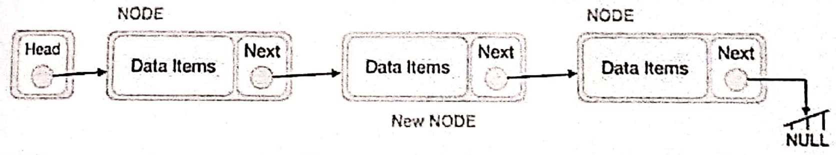


Now, the next node at the left should point to the new node.

```
LeftNode.next -> NewNode;
```



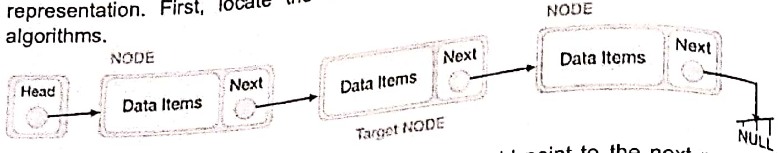
This will put the new node in the middle of the two. The new list should look like this –



Similar steps should be taken if the node is being inserted at the beginning of the list. While inserting it at the end, the second last node of the list should point to the new node and the new node will point to NULL.

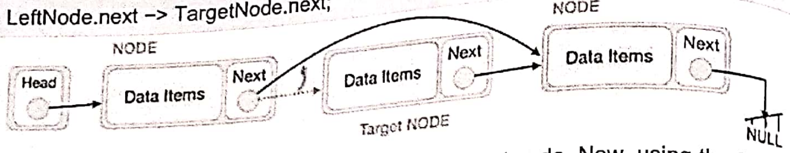
Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.



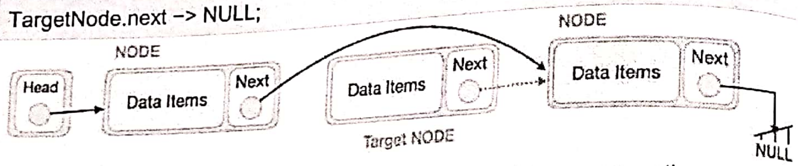
The left (previous) node of the target node now should point to the next node of the target node –

```
LeftNode.next –> TargetNode.next;
```

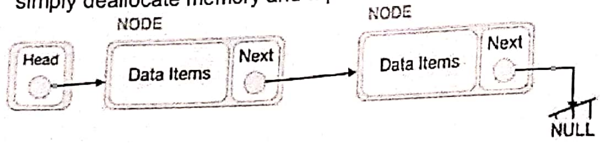


This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

```
TargetNode.next –> NULL;
```

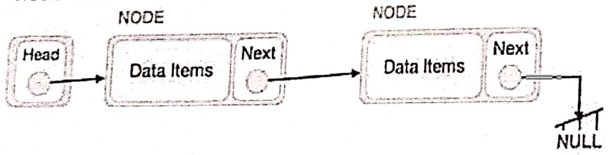


We need to use the deleted node. We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

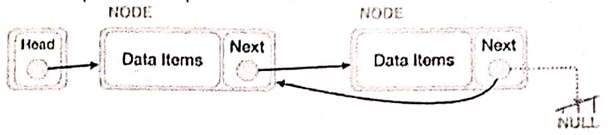


Reverse Operation

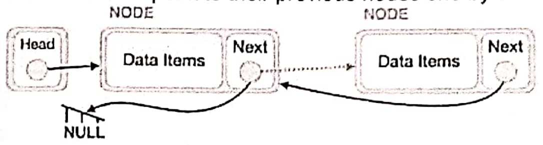
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



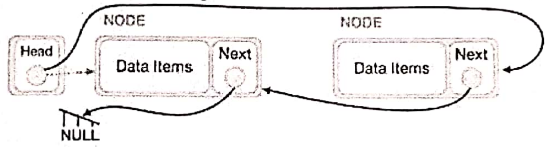
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node –



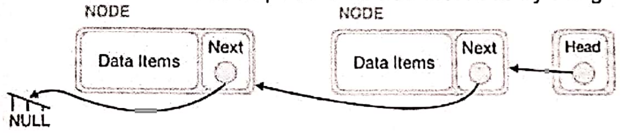
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



Except the node (first node) pointed by the head node, all nodes should point to their predecessor, making them their new successor. The first node will point to NULL.



We'll make the head node point to the new first node by using the temp node.



The linked list is now reversed.

Program:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

struct node {
  int data;
  int key;
  struct node *next;
};
```

```
struct node *head = NULL;
struct node *current = NULL;

//display the list
void printList() {
   struct node *ptr = head;
   printf("\n[ ");

   //start from the beginning
   while(ptr != NULL) {
      printf("(%d,%d) ",ptr->key,ptr->data);
      ptr = ptr->next;
   }

   printf(" ]");
}

//insert link at the first location
void insertFirst(int key, int data) {
   //create a link
   struct node *link = (struct node*) malloc(sizeof(struct node));

   link->key = key;
   link->data = data;

   //point it to old first node
   link->next = head;

   //point first to new first node
   head = link;
}

//delete first item
struct node* deleteFirst() {

   //save reference to first link
   struct node *tempLink = head;

   //mark next to first link as first
   head = head->next;

   //return the deleted link
   return tempLink;
}

//is list empty
bool isEmpty() {
   return head == NULL;
}

int length() {
   int length = 0;
   struct node *current;

   for(current = head; current != NULL; current = current->next) {
      length++;
   }

   return length;
}

//find a link with given key
struct node* find(int key) {

   //start from the first link
   struct node* current = head;

   //if list is empty
   if(head == NULL) {
      return NULL;
   }

   //navigate through list
   while(current->key != key) {

      //if it is last node
      if(current->next == NULL) {
         return NULL;
      } else {
         //go to next link
         current = current->next;
      }
   }
```

```
      //if data found, return the current Link
      return current;
   }

   //delete a link with given key
   struct node* delete(int key) {

      //start from the first link
      struct node* current = head;
      struct node* previous = NULL;

      //if list is empty
      if(head == NULL) {
         return NULL;
      }

      //navigate through list
      while(current->key != key) {

         //if it is last node
         if(current->next == NULL) {
            return NULL;
         } else {
            //store reference to current link
            previous = current;
            //move to next link
            current = current->next;
         }
      }

      //found a match, update the link
      if(current == head) {
         //change first to point to next link
         head = head->next;
      } else {
         //bypass the current link
         previous->next = current->next;
      }

      return current;
   }
```

```
void sort() {

   int i, j, k, tempKey, tempData;
   struct node *current;
   struct node *next;

   int size = length();
   k = size ;

   for ( i = 0 ; i < size - 1 ; i++, k-- ) {
      current = head;
      next = head->next;

      for ( j = 1 ; j < k ; j++ ) {

         if ( current->data > next->data ) {
            tempData = current->data;
            current->data = next->data;
            next->data = tempData;

            tempKey = current->key;
            current->key = next->key;
            next->key = tempKey;
         }

         current = current->next;
         next = next->next;
      }
   }
}

void reverse(struct node** head_ref) {
   struct node* prev   = NULL;
   struct node* current = *head_ref;
   struct node* next;

   while (current != NULL) {
      next  = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
```

```
    *head_ref = prev;
}

void main() {
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("Original List: ");

   //print list
   printList();

   while(!isEmpty()) {
      struct node *temp = deleteFirst();
      printf("\nDeleted value:");
      printf("(%d,%d) ",temp->key,temp->data);
   }

   printf("\nList after deleting all items: ");
   printList();
   insertFirst(1,10);
   insertFirst(2,20);
   insertFirst(3,30);
   insertFirst(4,1);
   insertFirst(5,40);
   insertFirst(6,56);

   printf("\nRestored List: ");
   printList();
   printf("\n");

   struct node *foundLink = find(4);

   if(foundLink != NULL) {
      printf("Element found: ");
      printf("(%d,%d) ",foundLink->key,foundLink->data);
      printf("\n");
   } else {
      printf("Element not found.");
   }

   delete(4);
   printf("List after deleting an item: ");
   printList();
   printf("\n");
   foundLink = find(4);

   if(foundLink != NULL) {
      printf("Element found: ");
      printf("(%d,%d) ",foundLink->key,foundLink->data);
      printf("\n");
   } else {
      printf("Element not found.");
   }

   printf("\n");
   sort();

   printf("List after sorting the data: ");
   printList();

   reverse(&head);
   printf("\nList after reversing the data: ");
   printList();
}
```

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

Output

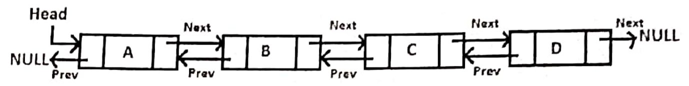
```
Original List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
Deleted value:(6,56)
Deleted value:(5,40)
Deleted value:(4,1)
Deleted value:(3,30)
Deleted value:(2,20)
Deleted value:(1,10)
List after deleting all items:
[ ]
Restored List:
[ (6,56) (5,40) (4,1) (3,30) (2,20) (1,10) ]
```

```
Element found: (4,1)
List after deleting an item:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
Element not found.
List after sorting the data:
[ (1,10) (2,20) (3,30) (5,40) (6,56) ]
List after reversing the data:
[ (6,56) (5,40) (3,30) (2,20) (1,10) ]
```

Lecture-09

Doubly Linked List

A Doubly Linked List (DLL) contains an extra pointer, typically called *previous pointer*, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node in C language.

```
/* Node of a doubly linked list */
struct Node {
    int data;
    struct Node* next; // Pointer to next node in DLL
    struct Node* prev; // Pointer to previous node in DLL
};
```

Following are advantages/disadvantages of doubly linked list over singly linked list.

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
3) We can quickly insert a new node before a given node.
In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.

Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer. It is possible to implement DLL with single pointer though
2) All operations require an extra pointer previous to be maintained. For example, in insertion, we need to modify previous pointers together with next pointers. For example in following functions for insertions at different positions, we need 1 or 2 extra steps to set previous pointer.

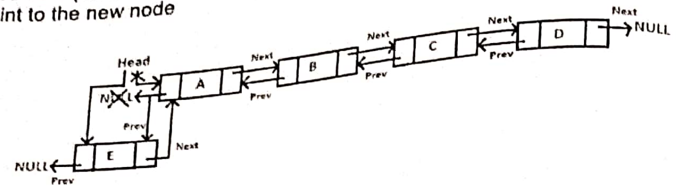
Insertion

A node can be added in four ways
1) At the front of the DLL
2) After a given node.
3) At the end of the DLL
4) Before a given node.

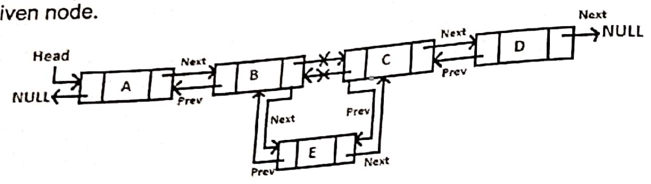
1) Add a node at the front: (A 5 steps process)

The new node is always added before the head of the given Linked List. And newly added node becomes the new head of DLL. For example if the given Linked List is

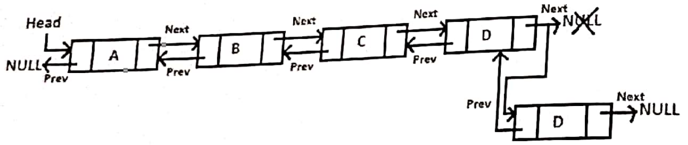
10152025 and we add an item 5 at the front, then the Linked List becomes 5101520
Let us call the function that adds at the front of the list is push(). The push() m
receive a pointer to the head pointer, because push must change the head pointe
point to the new node



2) Add a node after a given node.: (A 7 steps process)
We are given pointer to a node as prev_node, and the new node is inserted after
given node.

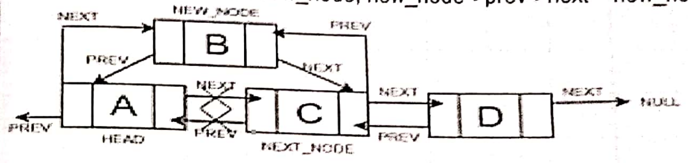


3) Add a node at the end: (7 steps process)
*The new node is always added after the last node of the given Linked List. For exam
if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becom
51015202530. Since a Linked List is typically represented by the head of it, we have
traverse the list till end and then change the next of last node to new node.*



4) Add a node before a given node:
Steps
*Let the pointer to this given node be next_node and the data of the new node to
added as new_data.*

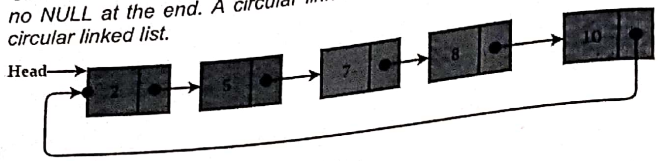
1. Check if the next_node is NULL or not. If it's NULL, return from the function because any new node can not be added before a NULL
2. Allocate memory for the new node, let it be called new_node
3. Set new_node->data = new_data
4. Set the previous pointer of this new_node as the previous node of the next_node, new_node->prev = next_node->prev
5. Set the previous pointer of the next_node as the new_node, next_node->prev = new_node
6. Set the next pointer of this new_node as the next_node, new_node->next = next_node;
7. If the previous node of the new_node is not NULL, then set the next pointer of this previous node as new_node, new_node->prev->next = new_node



Lecture-10

Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle. There no NULL at the end. A circular linked list can be a singly circular linked list or dou circular linked list.



Advantages of Circular Linked Lists:

1) Any node can be a starting point. We can traverse the whole list by starting from a point. We just need to stop when the first visited node is visited again.

2) Useful for implementation of queue. Unlike this implementation, we don't need maintain two pointers for front and rear if we use circular linked list. We can maintai pointer to the last inserted node and front can always be obtained as next of last.

3) Circular lists are useful in applications to repeatedly go around the list. For examp when multiple applications are running on a PC, it is common for the operating syste to put the running applications on a list and then to cycle through them, giving each them a slice of time to execute, and then making them wait while the CPU is given another application. It is convenient for the operating system to use a circular list so t when it reaches the end of the list it can cycle around to the front of the list.

4) Circular Doubly Linked Lists are used for implementation of advanced data structu like Fibonacci Heap.

Insertion in an empty List

Initially when the list is empty, *last* pointer will be NULL.

After inserting a node T,

After insertion, T is the last node so pointer *last* points to node T. And Node T is first and last node, so T is pointing to itself.
Function to insert node in an empty List,

```
struct Node *addToEmpty(struct Node *last, int data)
{
  // This function is only for empty list
  if (last != NULL)
    return last;

  // Creating a node dynamically.
  struct Node *last =
       (struct Node*)malloc(sizeof(struct Node));

  // Assigning the data.
  last -> data = data;

  // Note : list was empty. We link single node
  // to itself.
  last -> next = last;

  return last;
}
```

Run on IDE

Insertion at the beginning of the list

To Insert a node at the beginning of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next.
3. last -> next = T.

After insertion,

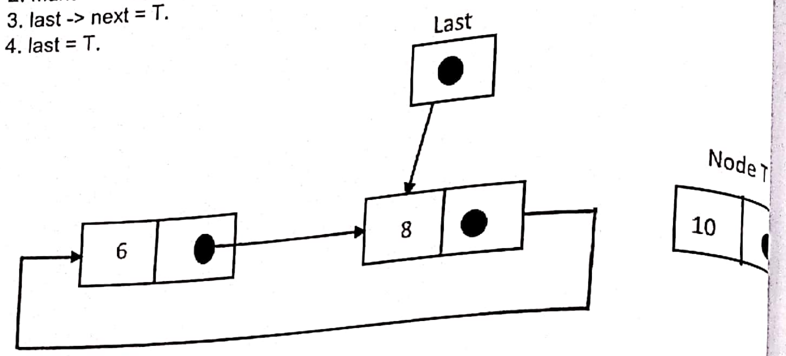
Function to insert node in the beginning of the List,

```
struct Node *addBegin(struct Node *last, int data)
{
  if (last == NULL)
     return addToEmpty(last, data);

  // Creating a node dynamically.
  struct Node *temp
        = (struct Node *)malloc(sizeof(struct Node));

  // Assigning the data.
  temp -> data = data;

  // Adjusting the links.
  temp -> next = last -> next;
  last -> next = temp;

  return last;
```

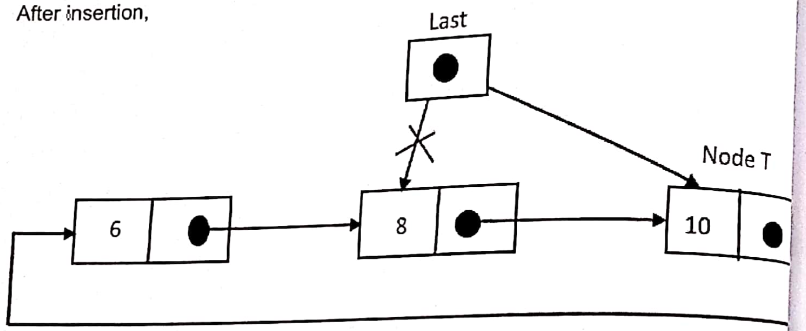
}

Insertion at the end of the list

To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Make T -> next = last -> next;
3. last -> next = T.
4. last = T.



After insertion,



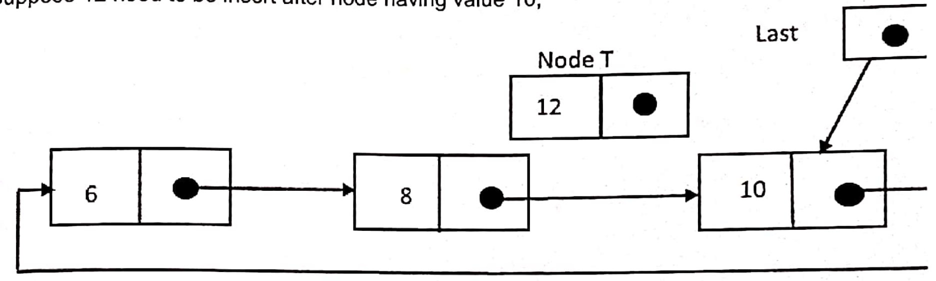
Function to insert node in the end of the List,

```
struct Node *addEnd(struct Node *last, int data)
{
  if (last == NULL)
    return addToEmpty(last, data);

// Creating a node dynamically.
struct Node *temp =
      (struct Node *)malloc(sizeof(struct Node));

// Assigning the data.
temp -> data = data;

// Adjusting the links.
temp -> next = last -> next;
last -> next = temp;
last = temp;

return last;
}
```

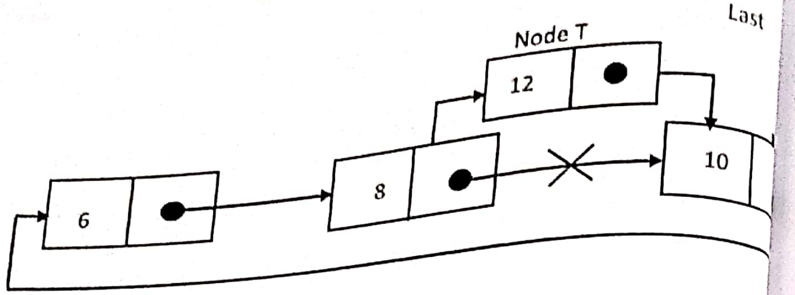
Insertion in between the nodes

To Insert a node at the end of the list, follow these step:
1. Create a node, say T.
2. Search the node after which T need to be insert, say that node be P.
3. Make T -> next = P -> next;
4. P -> next = T.

Suppose 12 need to be insert after node having value 10,



After searching and insertion,



Function to insert node in the end of the List.

```
struct Node *addAfter(struct Node *last, int data, int item)
{
  if (last == NULL)
    return NULL;
  struct Node *temp, *p;
 p = last -> next;
 // Searching the item.
do
{
  if (p ->data == item)
  {
     temp = (struct Node *)malloc(sizeof(struct Node));
     // Assigning the data.
    temp -> data = data;
    // Adjusting the links.
   temp -> next = p -> next;
   // Adding newly allocated node after p.
  p -> next = temp;
  // Checking for the last node.
 if (p == last)
    last = temp;
 return last;
}
 p = p -> next;
} while (p != last -> next);

cout << item << " not present in the list." << endl;
return last;
}
```

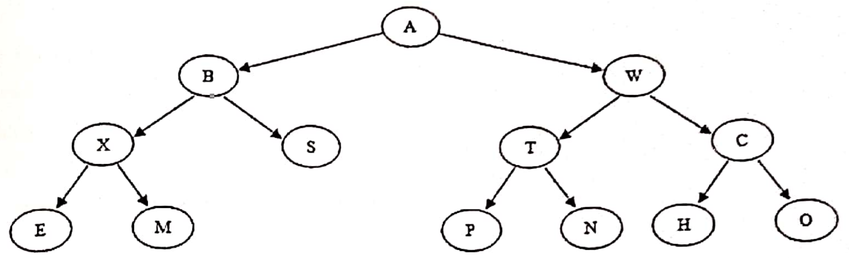
Lecture-13

Binary Tree

A *binary tree* consists of a finite set of nodes that is either empty, or consists of one specially designated node called the *root* of the binary tree, and the elements of two disjoint binary trees called the *left subtree* and *right subtree* of the root.

Note that the definition above is recursive: we have defined a binary tree in terms of binary trees. This is appropriate since recursion is an innate characteristic of tree structures.

Diagram 1: A binary tree



Binary Tree Terminology

Tree terminology is generally derived from the terminology of family trees (specifically, the type of family tree called a *lineal chart*).

- Each root is said to be the *parent* of the roots of its subtrees.
- Two nodes with the same parent are said to be *siblings*; they are the *children* of their parent.
- The root node has no parent.
- A great deal of tree processing takes advantage of the relationship between a parent and its children, and we commonly say a *directed edge* (or simply an *edge*) extends from a parent to its children. Thus edges connect a root with the roots of each subtree. An *undirected edge* extends in both directions between a parent and a child.

- *Grandparent and grandchild relations* can be defined in a similar manner; w could also extend this terminology further if we wished (designating nodes a cousins, as an uncle or aunt, etc.).

Other Tree Terms

- The number of subtrees of a node is called the *degree* of the node. In a binar tree, all nodes have degree 0, 1, or 2.
- A node of degree zero is called a *terminal node* or *leaf node*.
- A non-leaf node is often called a *branch node*.
- The *degree of a tree* is the maximum degree of a node in the tree. A binary tre is degree 2.
- A *directed path* from node n_1 to n_k is defined as a sequence of nodes n_1, n_2 ..., n_k such that n_i is the parent of n_i+1 for $1 <= i < k$. An *undirected path* is a similar sequence of undirected edges. The length of this path is the number o edges on the path, namely $k – 1$ (i.e., the number of nodes – 1). There is a pat of length zero from every node to itself. Notice that in a binary tree there i exactly one path from the root to each node.
- The *level* or *depth* of a node with respect to a tree is defined recursively: the leve of the root is zero; and the level of any other node is one higher than that of it parent. Or to put it another way, the level or depth of a node n_i is the length of th unique path from the root to n_i.
- The *height* of n_i is the length of the longest path from n_i to a leaf. Thus all leaves in the tree are at height 0.
- The *height of a tree* is equal to the height of the root. The *depth of a tree* is equa to the level or depth of the deepest leaf; this is always equal to the height of the tree.
- If there is a directed path from n_1 to n_2, then n_1 is an ancestor of n_2 and n_2 is a descendant of n_1.

Lecture-15

Tree Traversal:

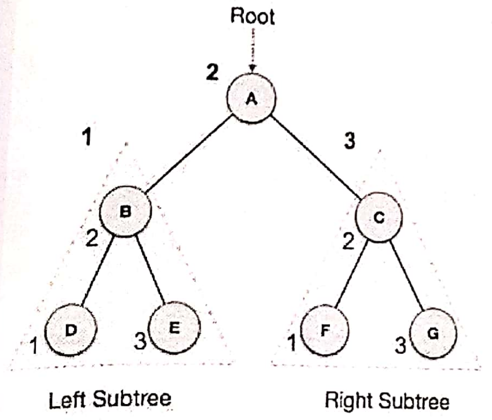
Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree. There are three ways which we use to traverse a tree –

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

Generally, we traverse a tree to search or locate a given item or key in the tree or to print all the values it contains.

In-order Traversal

In this traversal method, the left subtree is visited first, then the root and later the right sub-tree. We should always remember that every node may represent a subtree itself. If a binary tree is traversed **in-order**, the output will produce sorted key values in an ascending order.



We start from **A**, and following in-order traversal, we move to its left subtree **B**. **B** is also traversed in-order. The process goes on until all the nodes are visited. The output of inorder traversal of this tree will be –

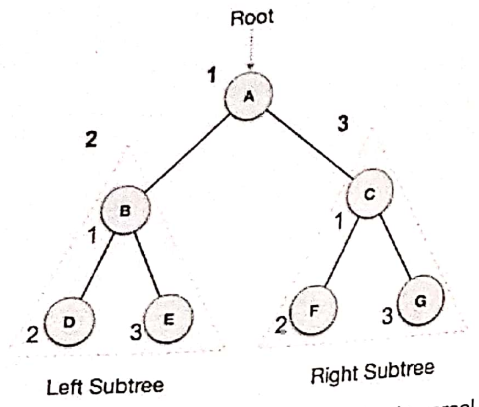
D → B → E → A → F → C → G

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Visit root node.
Step 3 − Recursively traverse right subtree.
```

Pre-order Traversal

In this traversal method, the root node is visited first, then the left subtree and finally the right subtree.



We start from **A**, and following pre-order traversal, we first visit **A** itself and then move to its left subtree **B**. **B** is also traversed pre-order. The process goes on until all the nodes are visited. The output of pre-order traversal of this tree will be −

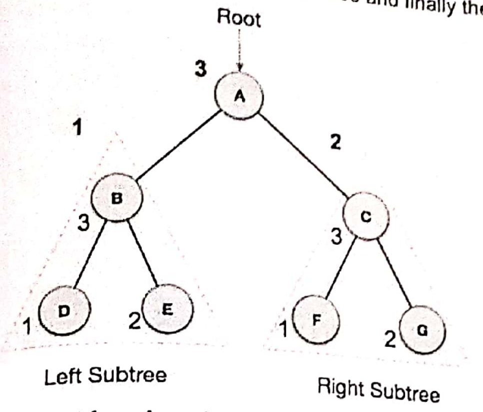
A → B → D → E → C → F → G

Algorithm

```
Until all nodes are traversed −
Step 1 − Visit root node.
Step 2 − Recursively traverse left subtree.
Step 3 − Recursively traverse right subtree.
```

Post-order Traversal

In this traversal method, the root node is visited last, hence the name. First we traverse the left subtree, then the right subtree and finally the root node.



We start from **A**, and following Post-order traversal, we first visit the left subtree **B**. **B** is also traversed post-order. The process goes on until all the nodes are visited. The output of post-order traversal of this tree will be −

D → E → B → F → G → C → A

Algorithm

```
Until all nodes are traversed −
Step 1 − Recursively traverse left subtree.
Step 2 − Recursively traverse right subtree.
Step 3 − Visit root node.
```