STACK

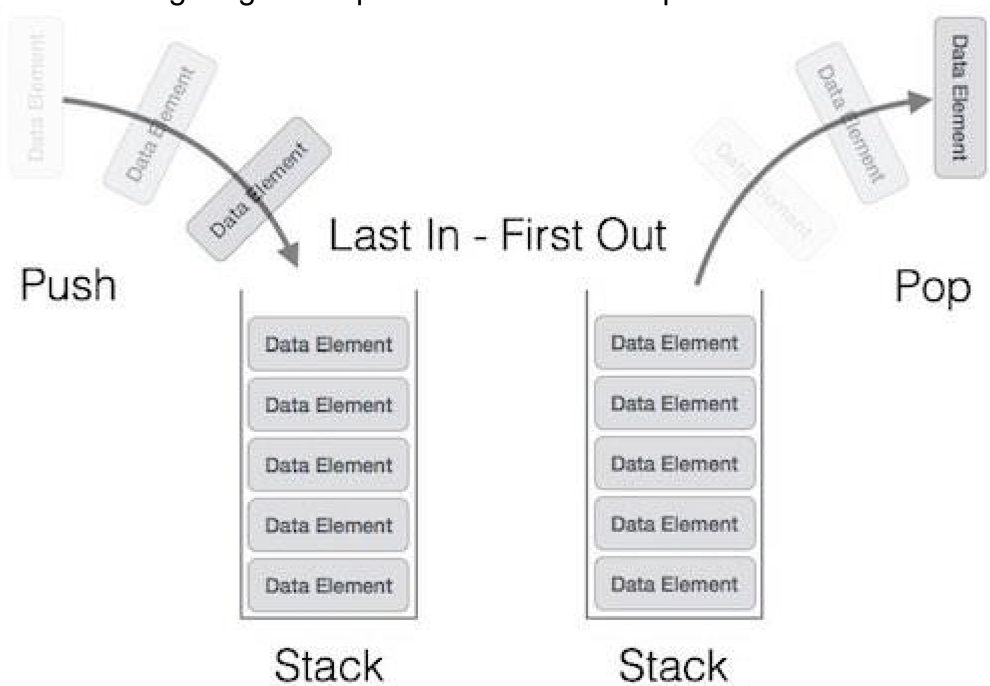
A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

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 purpose, 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 always represents the top of the stack, hence named **top**. The **top** pointer provides top value of the 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 −

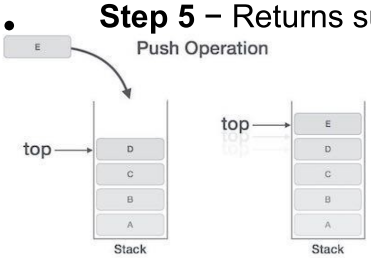
Example

```
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

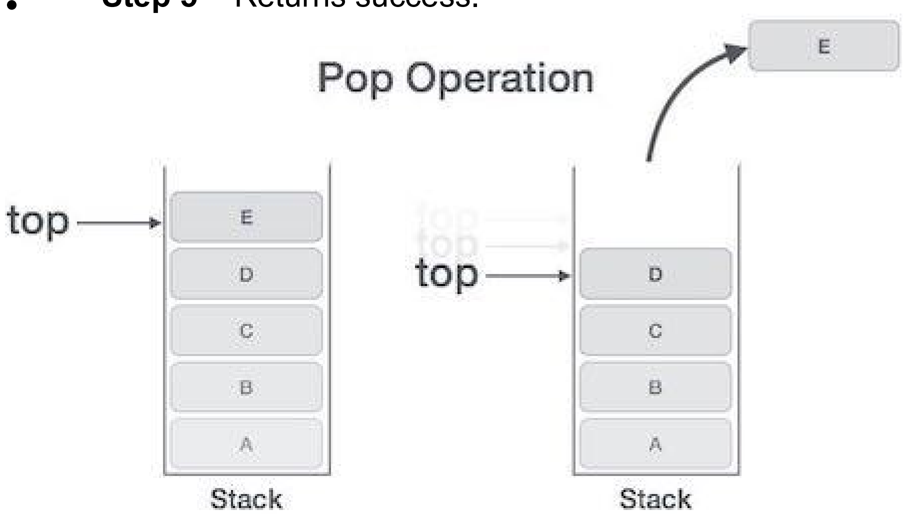
```
void push(int data) {
  if(!isFull()) {
    top = top + 1;
    stack[top] = data;
  } else {
    printf("Could not insert data, Stack is full.\n");
  }
}
```

Pop Operation

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

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 pointing.
- **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");
   }
}
```

Stack Applications

Three applications of stacks are presented here. These examples are central to many activities 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 boolean and logical expressions that can be evaluated in the same way. Control structures can also be treated similarly in a compiler.

This study of arithmetic expression evaluation is an example of problem solving where you solve 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: finding the shortest route in a graph (Traveling Salesman Problem), bin packing, linear programming, etc. that are similar enough that if a polynomial solution is ever found (exponential solutions 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 avoid 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 as well precede or follow the operands. You should note the advantage of prefix and postfix: the 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. The 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 string until a '(' is popped; discard the '('.
7. If the end of the input string is found, pop all operators and append them to the output string.

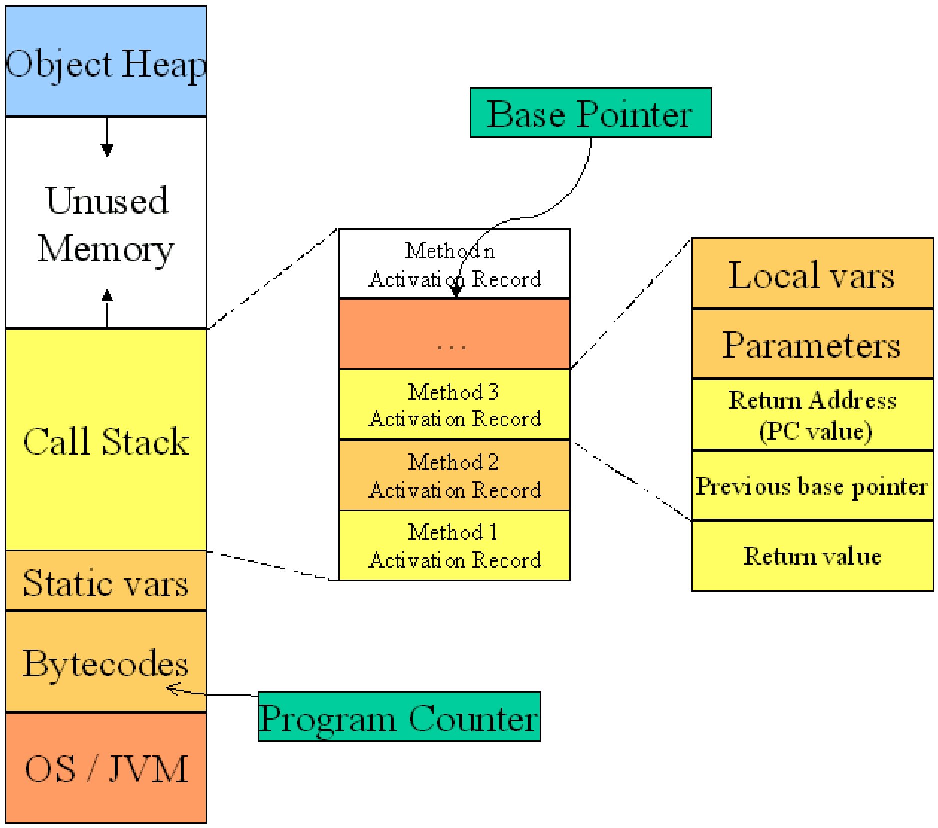
This algorithm doesn't handle errors in the input, although careful analysis of parenthesis or lack of parenthesis could point to such error determination.
Apply the algorithm to the above expressions.

Backtracking
Backtracking is used in algorithms in which there are steps along some path (state) from some 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 some way to remember these decision points in case we want/need to come back and try the alternative
Consider the maze. At a point where a choice is made, we may discover that the choice leads to a dead-end. We want to retrace back to that decision point and then try the other (next) alternative.
Again, stacks can be used as part of the solution. Recursion is another, typically more favored, solution, which is actually implemented by a stack.

Memory Management
Any modern computer environment uses a stack as the primary memory management model for a running program. Whether it's native code (x86, Sun, VAX) or JVM, a stack is at the center of the run-time environment for Java, C++, Ada, FORTRAN, etc.
The discussion of JVM in the text is consistent with NT, Solaris, VMS, Unix runtime environments.
Each program that is running in a computer system has its own memory allocation containing the typical layout as shown below.



Call and return process

When a method/function is called

1. An activation record is created; its size depends on the number and size of the local variables and parameters.
2. The Base Pointer value is saved in the special location reserved for it
3. The Program Counter value is saved in the Return Address location
4. The Base Pointer is now reset to the new base (top of the call stack prior to the creation of the AR)
5. The Program Counter is set to the location of the first bytecode of the method being called
6. Copies the calling parameters into the Parameter region
7. Initializes local variables in the local variable region

While the method executes, the local variables and parameters are simply found by adding a constant associated with each variable/parameter to the Base Pointer.

When a method returns

1. Get the program counter from the activation record and replace what's in the PC
2. Get the base pointer value from the AR and replace what's in the BP
3. Pop the AR entirely from the stack.

Lecture-06

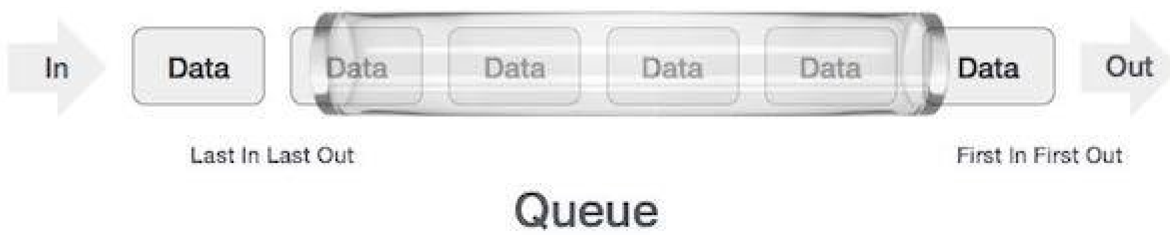
QUEUE

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 −



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() function is 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 pointer to reach at MAXSIZE to determine that the queue is full. In case we maintain the queue in a 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

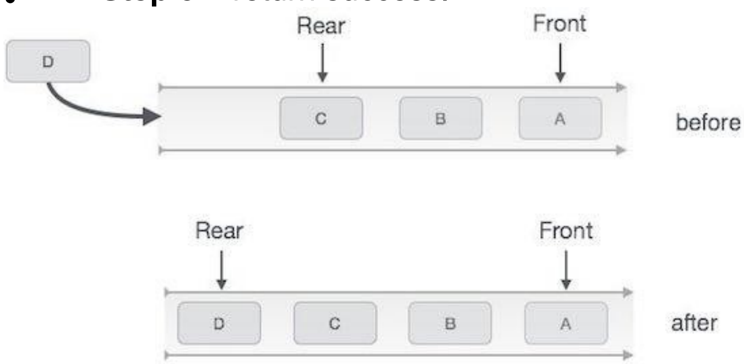
```
bool isempty() {
  if(front < 0 || front > rear)
    return true;
  else
    return false;
}
```

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 unforeseen 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 −

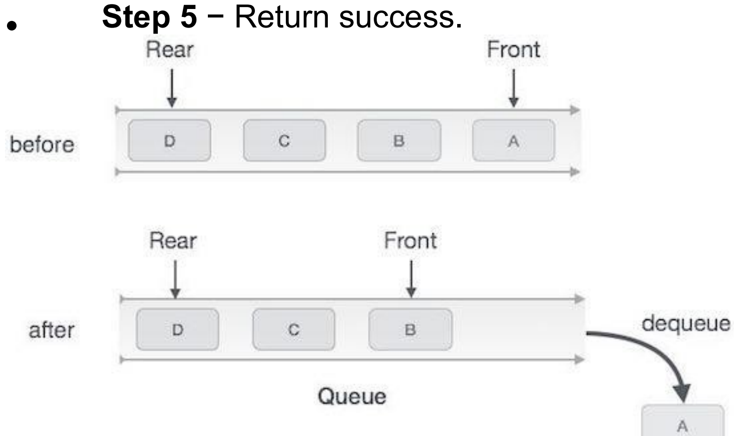
Example

```
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 **front** is pointing and remove the data after access. The following steps are taken to 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;
}
```

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.

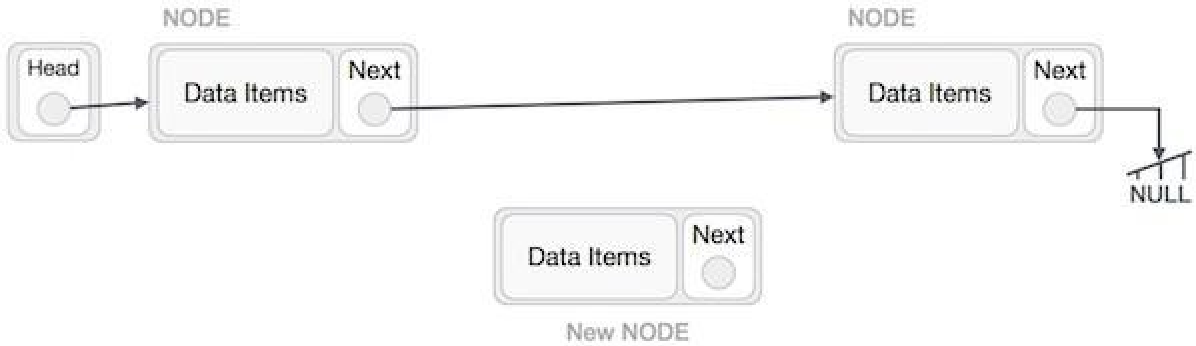
Basic Operations

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

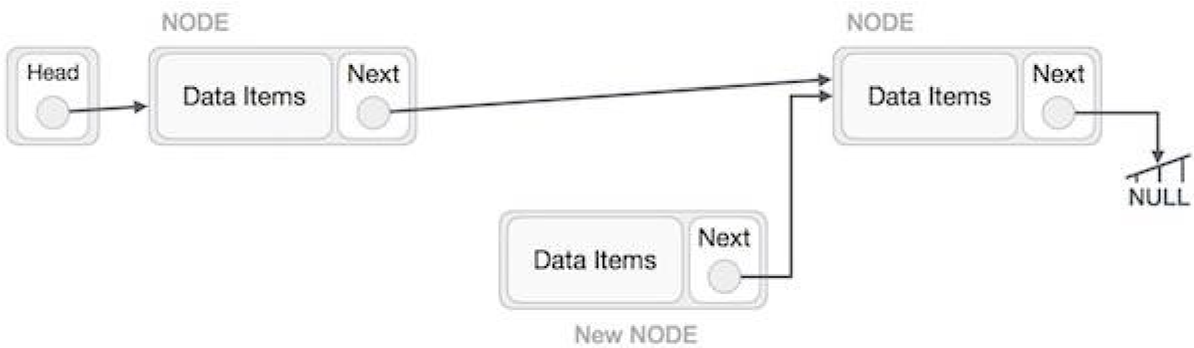
Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.



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

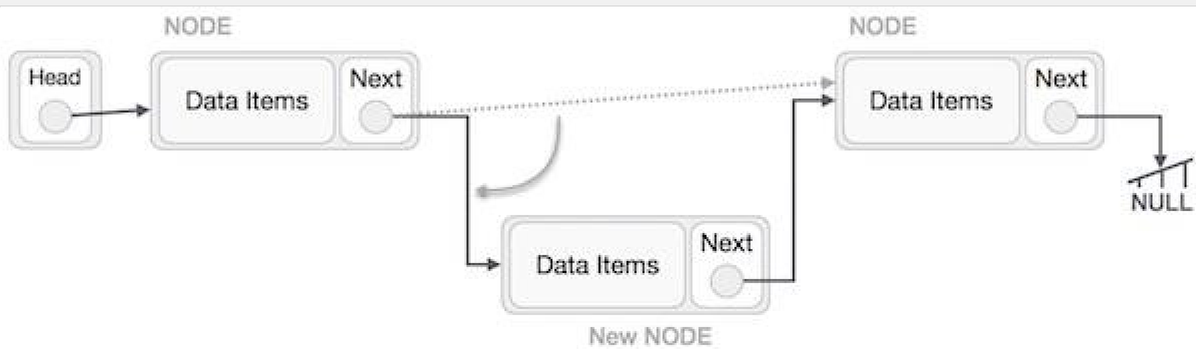
```
NewNode.next −> RightNode;
```

It should look like this −

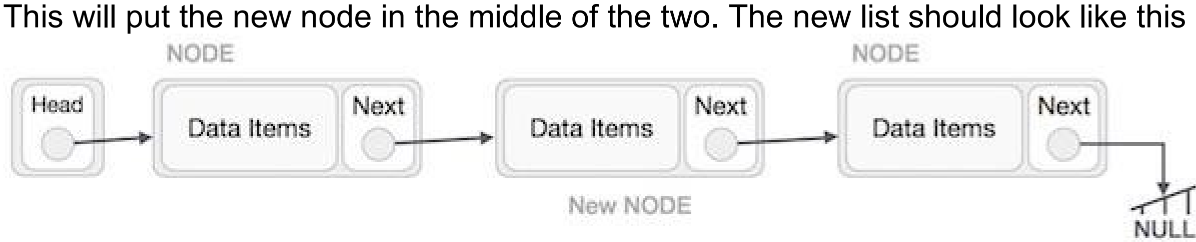


Now, the next node at the left should point to the new node.

```
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.

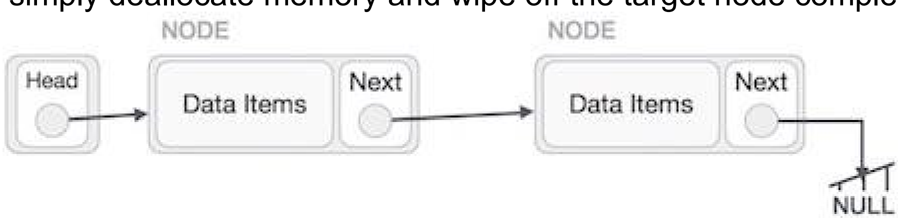
The left (previous) node of the target node now should point to the next node of the target node −

```
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.

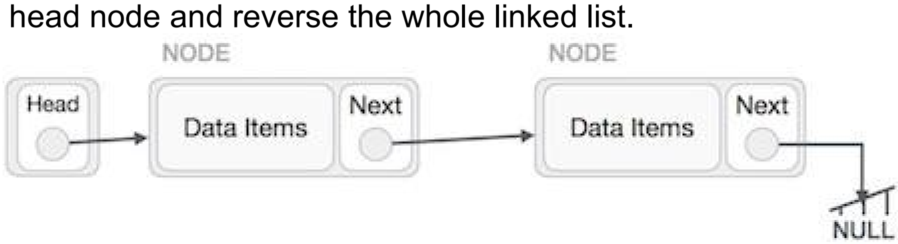
```
TargetNode.next −> NULL;
```

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.

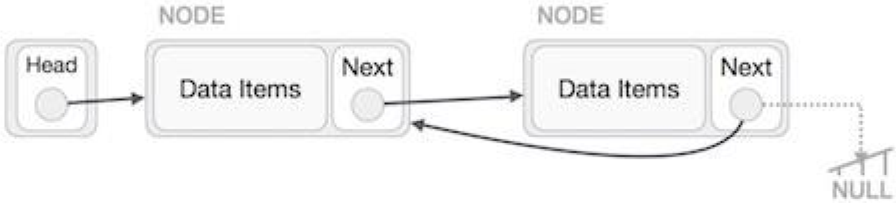


Reverse Operation

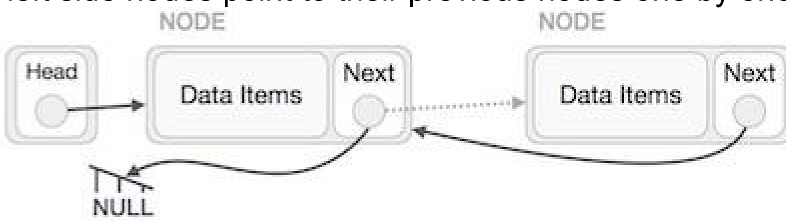
This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.



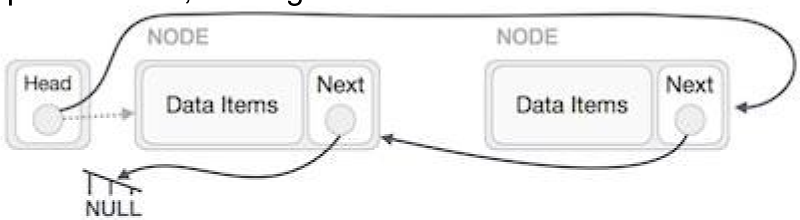
First, we traverse to the end of the list. It should be pointing to NULL. Now, we shall make it point to its previous node −



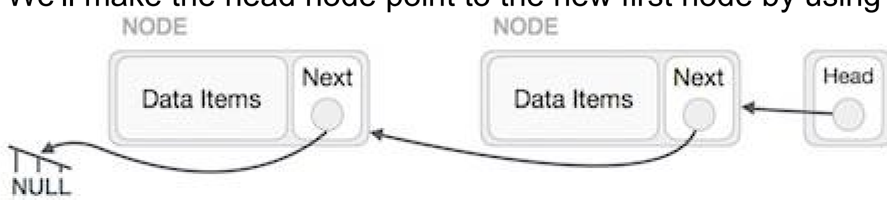
We have to make sure that the last node is not the lost node. So we'll have some temp node, which looks like the head node pointing to the last node. Now, we shall make all left side nodes point to their previous nodes one by one.



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) ]
```

Polynomial List

A polynomial p(x) is the expression in variable x which is in the form ($ax^n + bx^{n-1} + ... + jx + k$), where a, b, c …., k fall in the category of real numbers and 'n' is non negative integer, which is called the degree of polynomial.

An important characteristics of polynomial is that each term in the polynomial expression consists of two parts:

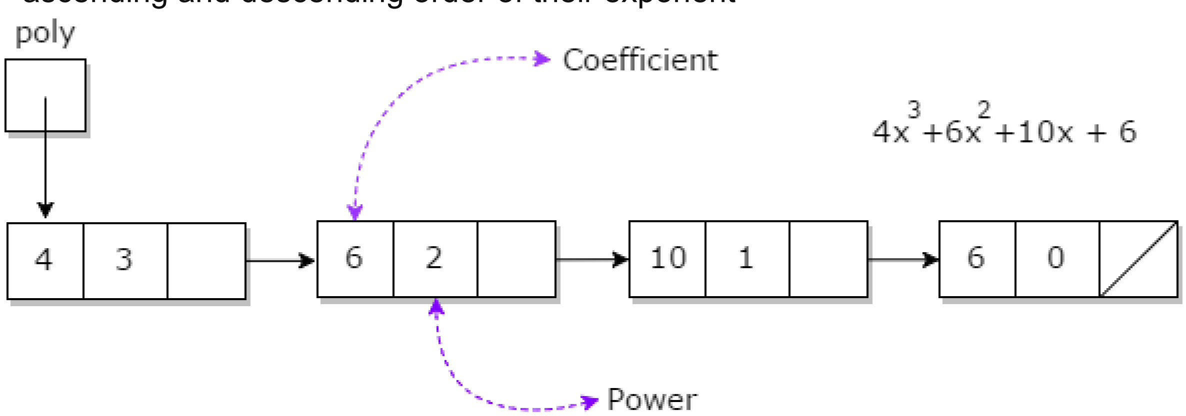
- one is the coefficient
- other is the exponent

Example:

$10x^2 + 26x$, here 10 and 26 are coefficients and 2, 1 are its exponential value.

Points to keep in Mind while working with Polynomials:

- The sign of each coefficient and exponent is stored within the coefficient and the exponent itself
- Additional terms having equal exponent is possible one
- The storage allocation for each term in the polynomial must be done in ascending and descending order of their exponent



Representation of Polynomial

Polynomial can be represented in the various ways. These are:

- By the use of arrays
- By the use of Linked List

Representation of Polynomials using Arrays

There may arise some situation where you need to evaluate many polynomial expressions and perform basic arithmetic operations like: addition and subtraction with those numbers. For this you will have to get a way to represent those polynomials. The simple way is to represent a polynomial with degree 'n' and store the coefficient of n+1 terms of the polynomial in array. So every array element will consists of two values:

- Coefficient and
- Exponent

Representation of Polynomial Using Linked Lists

A polynomial can be thought of as an ordered list of non zero terms. Each non zero term is a two tuple which holds two pieces of information:

- The exponent part
- The coefficient part

Adding two polynomials using Linked List

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

1st number = 5x^2 + 4x^1 + 2x^0

2nd number = 5x^1 + 5x^0

Output:

5x^2 + 9x^1 + 7x^0

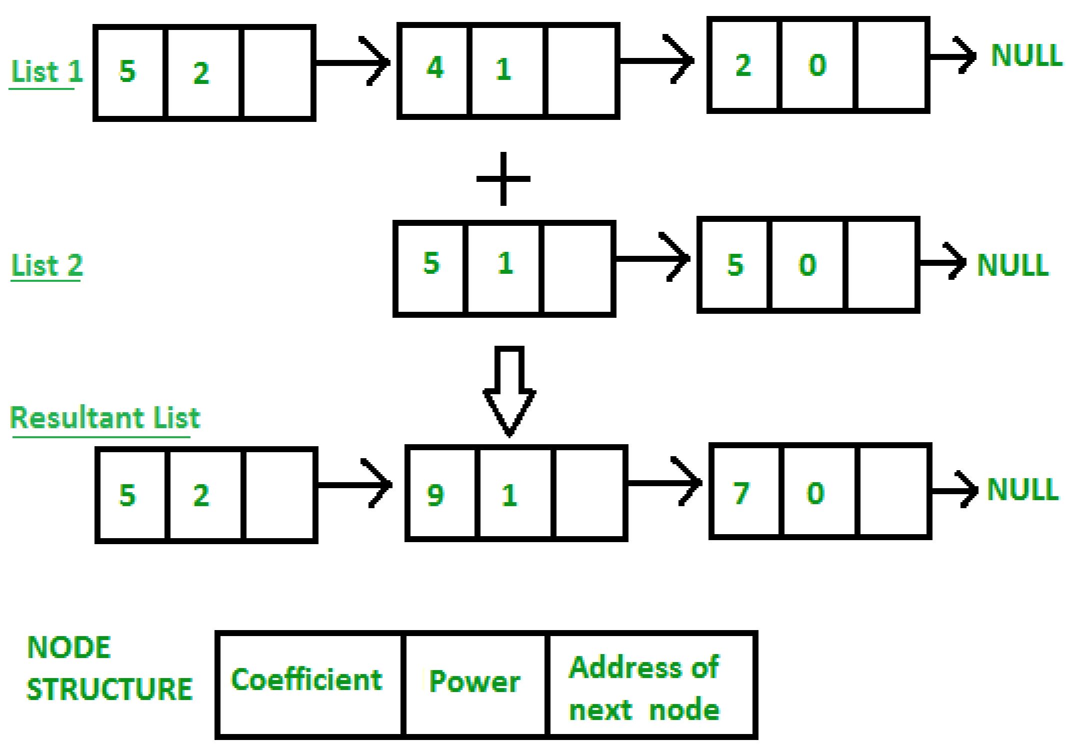
Input:

1st number = 5x^3 + 4x^2 + 2x^0

2nd number = 5x^1 + 5x^0

Output:

5x^3 + 4x^2 + 5x^1 + 7x^0



```
struct Node
{
    int coeff;
    int pow;
    struct Node *next;
};
void create_node(int x, int y, struct Node **temp)
{
    struct Node *r, *z;
    z = *temp;
    if(z == NULL)
    {
        r =(struct Node*)malloc(sizeof(struct Node));
        r->coeff = x;
        r->pow = y;
        *temp = r;
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
    else
    {
        r->coeff = x;
        r->pow = y;
```

```
        r->next = (struct Node*)malloc(sizeof(struct Node));
        r = r->next;
        r->next = NULL;
    }
}
void polyadd(struct Node *poly1, struct Node *poly2, struct Node *poly)
{
while(poly1->next && poly2->next)
    {
        if(poly1->pow > poly2->pow)
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        else if(poly1->pow < poly2->pow)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        else
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff+poly2->coeff;
```

```
        }

            poly1 = poly1->next;
            poly2 = poly2->next;
        }
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
while(poly1->next || poly2->next)
    {
        if(poly1->next)
        {
            poly->pow = poly1->pow;
            poly->coeff = poly1->coeff;
            poly1 = poly1->next;
        }
        if(poly2->next)
        {
            poly->pow = poly2->pow;
            poly->coeff = poly2->coeff;
            poly2 = poly2->next;
        }
        poly->next = (struct Node *)malloc(sizeof(struct Node));
        poly = poly->next;
        poly->next = NULL;
    }
```

```
}
void show(struct Node *node)
{
while(node->next != NULL)
   {
   printf("%dx^%d", node->coeff, node->pow);
   node = node->next;
   if(node->next != NULL)
      printf(" + ");
   }
}
 int main()
{
   struct Node *poly1 = NULL, *poly2 = NULL, *poly = NULL;
   // Create first list of 5x^2 + 4x^1 + 2x^0
   create_node(5,2,&poly1);
   create_node(4,1,&poly1);
   create_node(2,0,&poly1);
    // Create second list of 5x^1 + 5x^0
   create_node(5,1,&poly2);
   create_node(5,0,&poly2);
   printf("1st Number: ");
   show(poly1);
    printf("\n2nd Number: ");
   show(poly2);
```

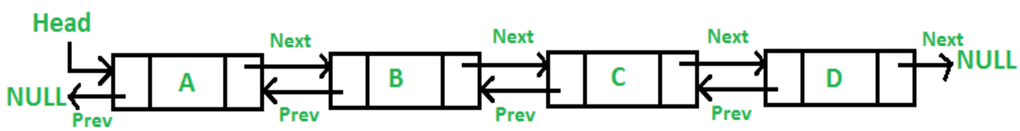
```
    poly = (struct Node *)malloc(sizeof(struct Node));
     // Function add two polynomial numbers
    polyadd(poly1, poly2, poly);
    // Display resultant List
    printf("\nAdded polynomial: ");
    show(poly);
 return 0;
}
```

Output:

```
1st Number: 5x^2 + 4x^1 + 2x^0
2nd Number: 5x^1 + 5x^0
Added polynomial: 5x^2 + 9x^1 + 7x^0
```

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.

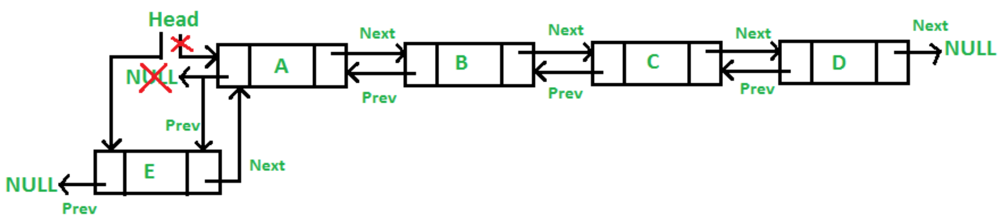
Insertion

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.

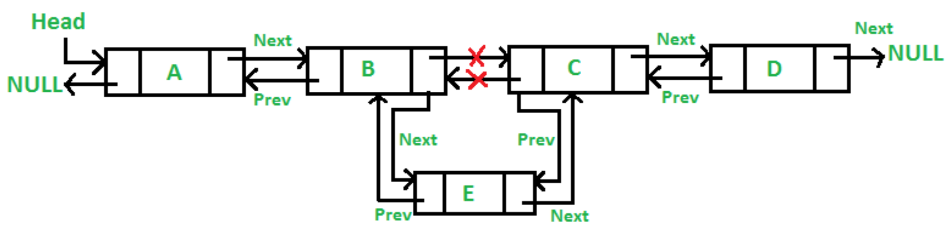
1) Add a node at the front: (A 5 steps process)

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

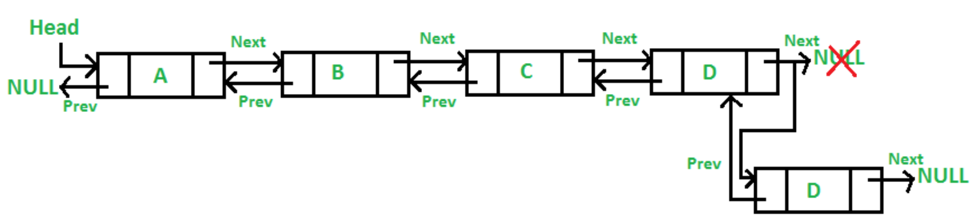
10152025 and we add an item 5 at the front, then the Linked List becomes 510152025. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



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 the given node.

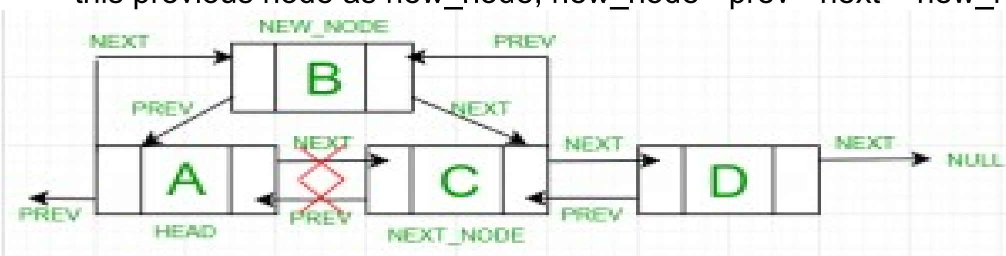


3) Add a node at the end: (7 steps process)
The new node is always added after the last node of the given Linked List. For example if the given DLL is 510152025 and we add an item 30 at the end, then the DLL becomes 51015202530. Since a Linked List is typically represented by the head of it, we have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



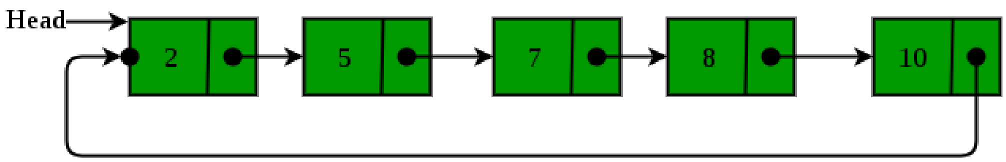
4) Add a node before a given node:
Steps
Let the pointer to this given node be next_node and the data of the new node to be added as new_data.

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



Circular Linked List

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly 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 any 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 to maintain two pointers for front and rear if we use circular linked list. We can maintain a 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 example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that 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 structures 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,

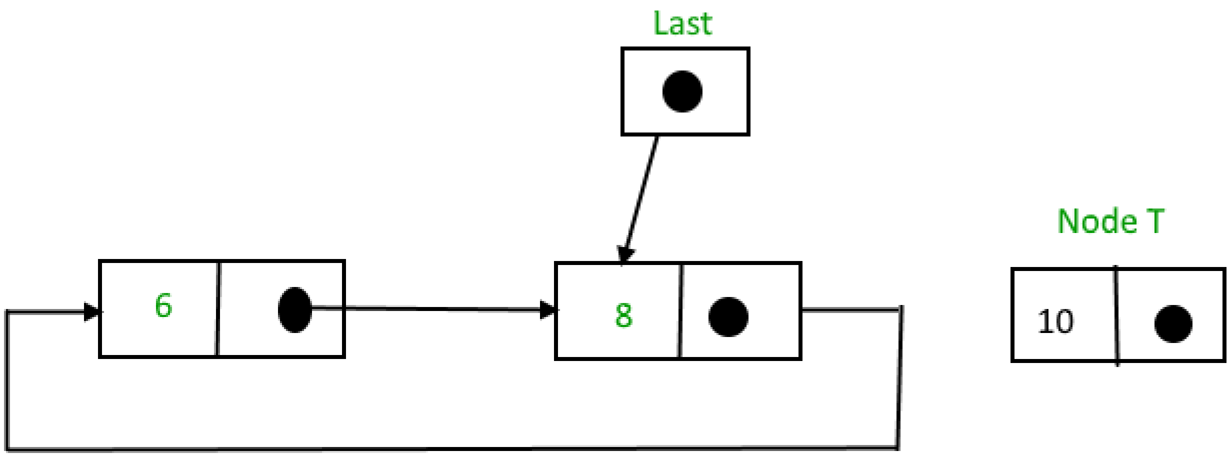
Function to insert node in the beginning of the List,

```
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;
```

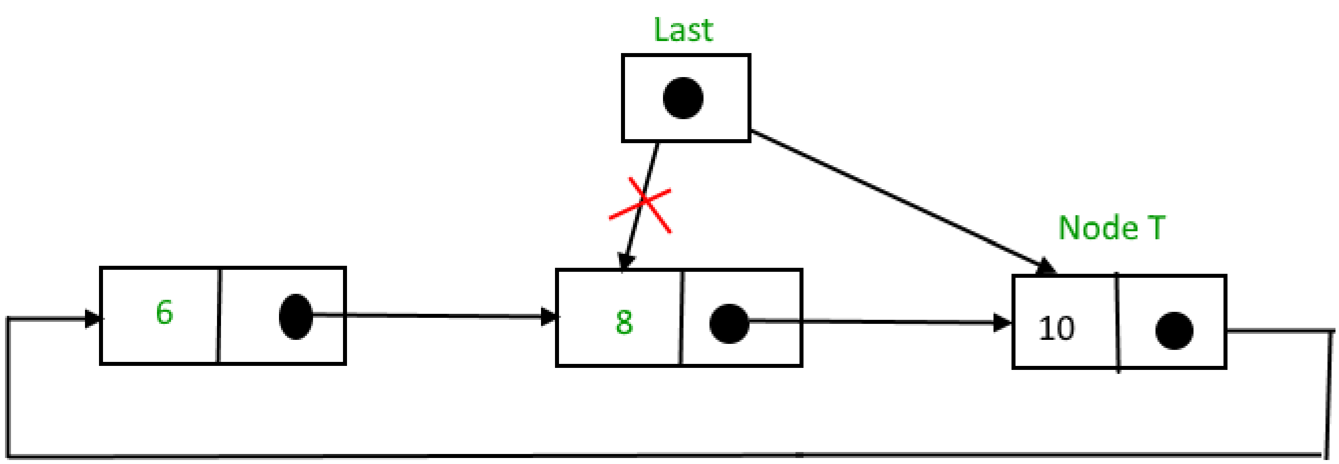
}

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,



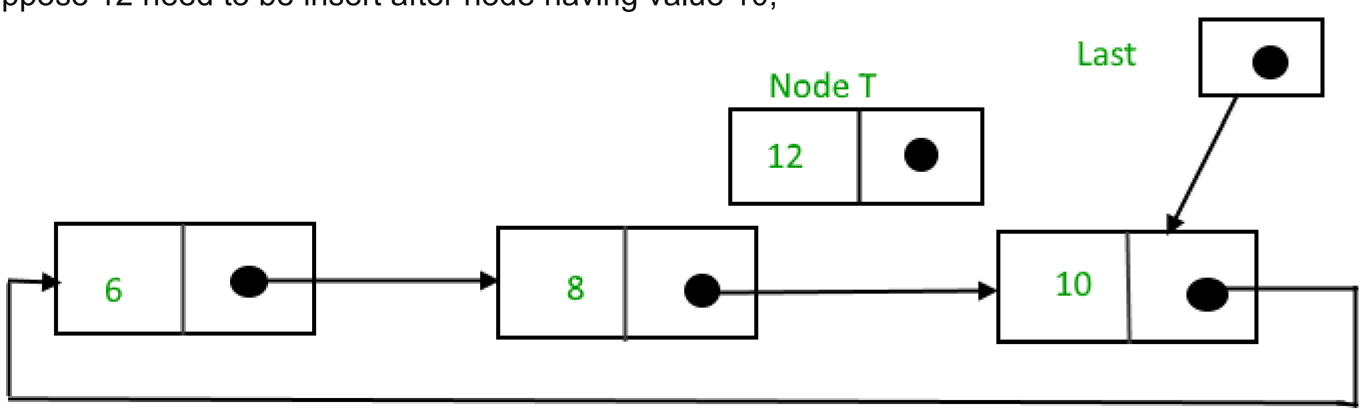
Function to insert node in the end of the List,
struct Node *addEnd(struct Node *last, int data)
{
  if (last == NULL)
    return addToEmpty(last, data);

  // Creating a node dynamically.

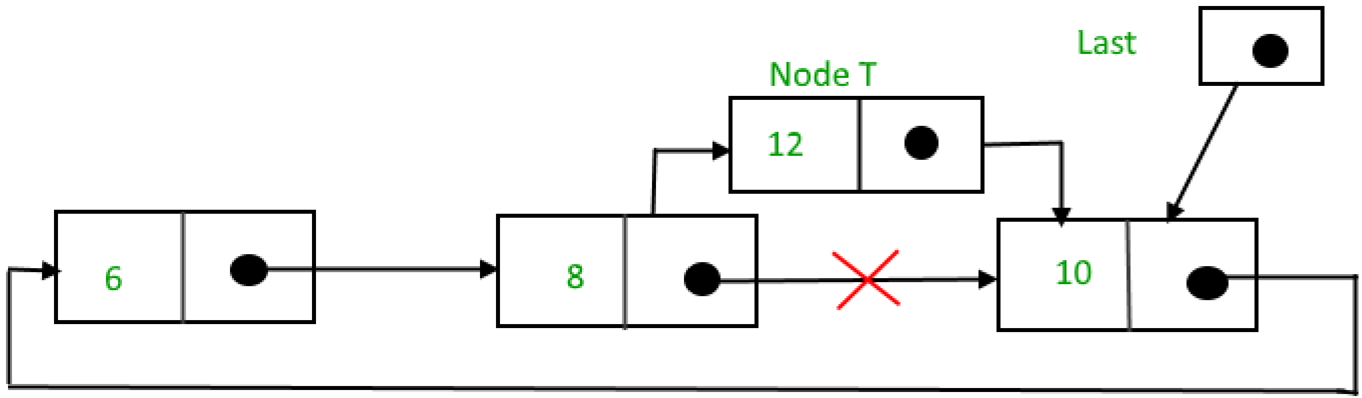
```
  struct Node *temp =
        (struct Node *)malloc(sizeof(struct Node));

  // 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;
}
```

Memory Allocation-

Whenever a new node is created, memory is allocated by the system. This memory is taken from list of those memory locations which are free i.e. not allocated. This list is called AVAIL List. Similarly, whenever a node is deleted, the deleted space becomes reusable and is added to the list of unused space i.e. to AVAIL List. This unused space can be used in future for memory allocation.

Memory allocation is of two types-

1. Static Memory Allocation
2. Dynamic Memory Allocation

1. **Static Memory Allocation:**

   When memory is allocated during compilation time, it is called 'Static Memory Allocation'. This memory is fixed and cannot be increased or decreased after allocation. If more memory is allocated than requirement, then memory is wasted. If less memory is allocated than requirement, then program will not run successfully. So exact memory requirements must be known in advance.

2. **Dynamic Memory Allocation:**

   When memory is allocated during run/execution time, it is called 'Dynamic Memory Allocation'. This memory is not fixed and is allocated according to our requirements. Thus in it there is no wastage of memory. So there is no need to know exact memory requirements in advance.

Garbage Collection-

Whenever a node is deleted, some memory space becomes reusable. This memory space should be available for future use. One way to do this is to immediately insert the free space into availability list. But this method may be time consuming for the operating system. So another method is used which is called 'Garbage Collection'. This method is described below: In this method the OS collects the deleted space time to time onto the availability list. This process happens in two steps. In first step, the OS goes through all the lists and tags all those cells which are currently being used. In the second step, the

OS goes through all the lists again and collects untagged space and adds this collected space to availability list. The garbage collection may occur when small amount of free space is left in the system or no free space is left in the system or when CPU is idle and has time to do the garbage collection.

Compaction

One preferable solution to garbage collection is compaction. The process of moving all marked nodes to one end of memory and all available memory to other end is called compaction. Algorithm which performs compaction is called compacting algorithm.

Infix to Postfix Conversion

```
#include<stdio.h>
char stack[20];
int top = -1;
void push(char x)
{
    stack[++top] = x;
}

char pop()
{
    if(top == -1)
        return -1;
    else
        return stack[top--];
}

int priority(char x)
{
    if(x == '(')
        return 0;
    if(x == '+' || x == '-')
        return 1;
    if(x == '*' || x == '/')
        return 2;
}

main()
{
    char exp[20];
    char *e, x;
    printf("Enter the expression :: ");
    scanf("%s",exp);
    e = exp;
    while(*e != '\0')
    {
        if(isalnum(*e))
            printf("%c",*e);
        else if(*e == '(')
            push(*e);
        else if(*e == ')')
        {
```

```
            while((x = pop()) != '(')
                printf("%c", x);
        }
        else
        {
            while(priority(stack[top]) >= priority(*e))
                printf("%c",pop());
            push(*e);
        }
        e++;
    }
    while(top != -1)
    {
        printf("%c",pop());
    }
}
```

OUTPUT:

```
Enter the expression :: a+b*c
abc*+

Enter the expression :: (a+b)*c+(d-a)
ab+c*da-+
```

Evaluate POSTFIX Expression Using Stack

```
#include<stdio.h>
int stack[20];
int top = -1;
 void push(int x)
{
        stack[++top] = x;
}

int pop()
{
        return stack[top--];
}

int main()
{
        char exp[20];
        char *e;
        int n1,n2,n3,num;
        printf("Enter the expression :: ");
        scanf("%s",exp);
        e = exp;
        while(*e != '\0')
        {
                if(isdigit(*e))
```

```
        {
            num = *e - 48;
            push(num);
        }
        else
        {
            n1 = pop();
            n2 = pop();
            switch(*e)
            {
                case '+':
                {
                    n3 = n1 + n2;
          break;
                }
                case '-':
                {
                    n3 = n2 - n1;
                    break;
                }
                case '*':
                {
                    n3 = n1 * n2;
                    break;
                }
```

```
                    case '/':
                    {
                        n3 = n2 / n1;
                        break;
                    }
                }
                push(n3);
            }
            e++;
        }
        printf("\nThe result of expression %s  =  %d\n\n",exp,pop());
        return 0;

    }

```

Output:

```
Enter the expression :: 245+*

The result of expression 245+*  =  18
```