UNIT III
LINEAR DATA STRUCTURES

DATA STRUCTURE

An organization of information, usually in memory, for better algorithm efficiency, such as queue, stack, linked list, heap, dictionary, and tree.

Why do we need data structures?

Data structures allow us to achieve an important goal: component reuse. Once each data structure has been implemented once, it can be used over and overagain in various applications.

Common data structures are

• Stacks • Queues • Lists

• Trees • Graphs • Tables

.Operations performed on any linear structure:

1. Traversal – Processing each element in the list

2. Search – Finding the location of the element with a given value.

3. Insertion – Adding a new element to the list.

4. Deletion – Removing an element from the list.

5. Sorting – Arranging the elements in some type of order.

6. Merging – Combining two lists into a single list.

Types:-

A data structure can be broadly classified into

(i) Primitive data structure
(ii) Non-primitive data structure

(i) Primitive data structure

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double incase of ‘c’ are known as primitive data structures.

Eg : int, char and double.

(ii) Non-primitive data structure

The data structures, which are not primitive are called non-primitive data structures.

There are two types of-primitive data structures.

1. **Linear Data Structures:-**

A list, which shows the relationship of adjacency between elements, is said to be linear data structure. The most, simplest linear data structure is a 1-D array, but because of its deficiency, list is frequently used for different kinds of data.

Eg: Arrays, Lists, Stacks and Queues.

2. **Non-linear data structure:-**

A list, which doesn't show the relationship of adjacency between elements, is said to be non-linear data structure.

Eg : Trees, Graphs.

PRIMITIVE AND NON-PRIMITIVE DATA TYPES

A data type is a classification of data, which can store a specific type of information.

Primitive data types are predefined types of data, which are supported by the programming language. For example, integer, character, and string are all primitive data types. Programmers can use these data types when creating variables in their programs. For example, a programmer may create a variable called "lastname" and define it as a string data type. The variable will then store data as a string of characters.

Non-primitive data types are not defined by the programming language, but are instead created by the programmer. They are sometimes called "reference variables," or "object references," since they reference a memory location, which stores the data.

REPRESENATION OF DATA STRUCTURES:-

Any data structure can be represented in two ways. They are: -

(i) Sequential representation
(ii) Linked representation

ABSTRACT DATA TYPE

An abstract data type is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. A data type can be considered **abstract** when it is defined in terms of operations on it, and its implementation is hidden (so that we can always replace one implementation with

another for, e.g., efficiency reasons, and this will not interfere with anything in the program).

A set of data values and associated operations that are precisely specified independent of any particular implementation.

Example: dictionary, stack, queue, priority queue, set, bag, tree, associative array.

One of the simplest abstract data types is the stack. The operations new(), push(v, S), top(S), and popOff (S) may be defined with axiomatic semantics as following.

1. new() returns a stack
2. popOff(push(v, S)) = S
3. top(push(v, S)) = v

where S is a stack and v is a value.

- Integers
- Real numbers.
- Complex numbers
- Polynomials
- Matrices
- Sets
- Multiset
- Book

Basic Properties of ADT are: -

- Encapsulation and Generalization

Example:

```
Struct student
{
        int rno;
        char name[21],branch[11]
        int marks;.
};
```

The above structure can be used to collect or retrieve the information of a student. The structure can be called as ADT if all the operations on student can be performed using the structure.

Uses of ADT: -

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmer has to keep in mind at any time

4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and Debugging.

LIST ADT: LINEAR DATA STRUCTURE

A list is an ordered list, which consists of different data items connected by means of a link or pointer. A **list** or **sequence** is an abstract data type that implements a finite ordered collection of values, where the same value may occur more than once. The list operations are:

- printList
- makeEmpty
- find
- findKth
- insert
- remove
- isEmpty
- zeroth
- first
- findPrevious

Implementation of List ADT

1. Array Implementation
2. Linked List Implementation
3. Cursor Implementation.

ARRAY IMPLEMENTATION OF LIST

- Array is a collection of specific number of data stored in consecutive memory locations.
- Insertion and Deletion operation are expensive as it requires more data movement
- Find and Print list operations takes constant time.
- Even if the array is dynamically allocated, an estimate of the maximum size of the list is required which considerably wastes the memory space.

The very common linear structure is array. Since arrays are usually easy to traverse, search and sort, they are frequently used to store relatively permanent collections of data.

An array is a list of a finite number n of homogeneous data elements (i.e., data elements of the same type) such that:
a) The elements of the array are referenced respectively by an index consisting of n consecutive numbers.
b) The elements of the array are stored respectively in successive memory locations.

Operations of Array:

Two basic operations in an array are storing and retrieving (extraction)

Storing: A value is stored in an element of the array with the statement of the form,

Data[i] = X ; Where I is the valid index in the array And X is the element

Extraction : Refers to getting the value of an element stored in an array.

X = Data [i], Where I is the valid index of the array and X is the element.

Advantages:

• Reduces memory access time, because all the elements are stored sequentially. By incrementing the index, it is possible to access all the elements in an array.

• Reduces no. of variables in a program.

• Easy to use for the programmers.

Disadvantages:

• Wastage of memory space is possible. For example: Storing only 10 elements in a 100 size array. Here, remaining 90 elements space is waste because these spaces can't be used by other programs till this program completes its execution.

• Storing heterogeneous elements are not possible.

• Array bound checking is not available in 'C'. So, manually we have to do that.

LINKED LIST IMPLEMENTATION

- Linked list consists of series of nodes. Each node contains the element and a pointer to its
- successor node. The pointer of the last node points to NULL.
- Insertion and deletion operations are easily performed using linked list.

A collection of items accessible one after another beginning at the head and ending at the tail.

Head: The first item in a list

Tail: The last item in a list

Some demerits of array, leads us to use linked list to store the list of items. They are,

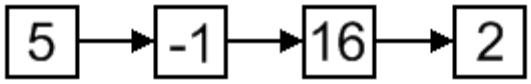
1. It is relatively expensive to insert and delete elements in an array.
2. Array usually occupies a block of memory space, one cannot simply double or triple the size of an array when additional space is required. (For this reason, arrays are called **"dense lists"** and are said to be **"static"** data structures.)

Singly-linked list

Linked list is a very important dynamic data structure. Basically, there are two types of linked list, singly-linked list and doubly-linked list. In a singly-linked list every element contains some data and a link to the next element, which allows to keep the structure. On the other hand, every node in a doubly-linked list also contains a link to the previous node. Linked list can be an underlying data structure to implement stack, queue or sorted list.

Example

Sketchy, singly-linked list can be shown like this:



Each cell is called a **node** of a singly-linked list. First node is called **head** and it's a dedicated node. By knowing it, we can access every other node in the list. Sometimes, last node, called **tail**, is also stored in order to speed up add operation.

Operations on a singly-linked list

Concrete implementation of operations on the singly-linked list depends on the purpose, it is used for. Following the links below, you can find descriptions of the common concepts, proper for every implementation.

- Singly-linked list traversal
- Adding a node
- Removing a node

Singly-linked list. Addition (insertion) operation.

Insertion into a singly-linked list has two special cases. It's insertion a new node before the head (to the very beginning of the list) and after the tail (to the very end of the list). In any other case, new node is inserted in the middle of the list and so, has a predecessor and successor in the list. There is a description of all these cases below.

Empty list case

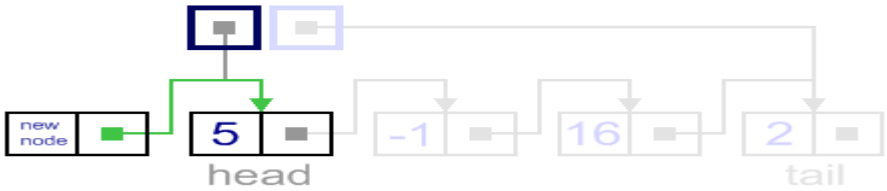
When list is empty, which is indicated by (head == NULL)condition, the insertion is quite simple. Algorithm sets both head and tail to point to the new node.

Add first

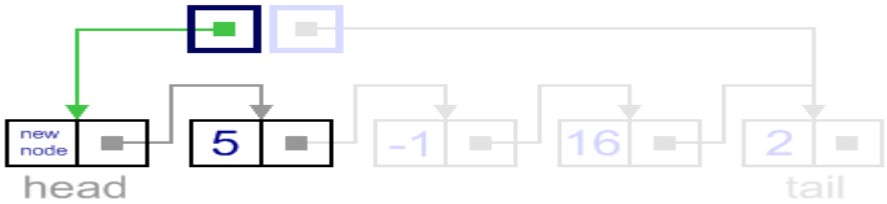
In this case, new node is inserted right before the current head node.

It can be done in two steps:

1. Update the next link of a new node, to point to the current head node.

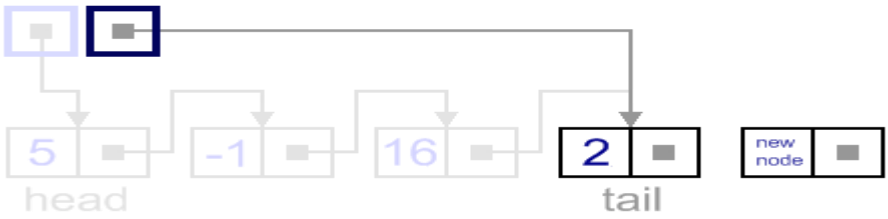


2. Update head link to point to the new node.



Add last

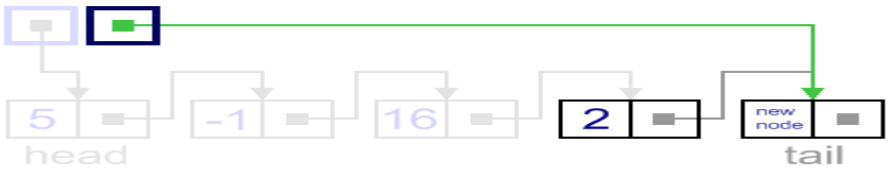
In this case, new node is inserted right after the current tail node.



It can be done in two steps:

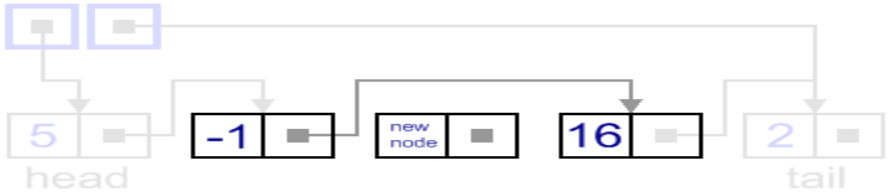
1. Update the next link of the current tail node, to point to the new node.

2. Update tail link to point to the new node.



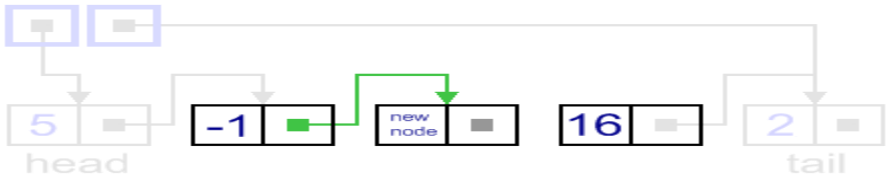
General case

In general case, new node is **always inserted between** two nodes, which are already in the list. Head and tail links are not updated in this case.

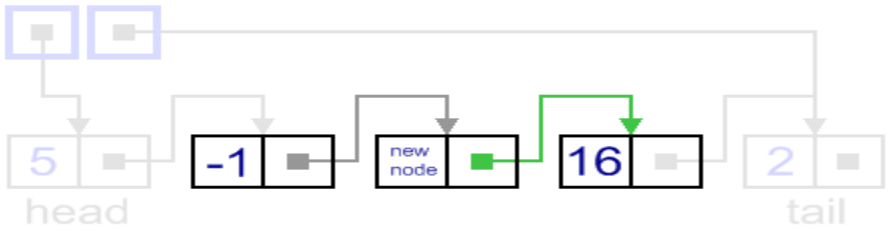


Such an insert can be done in two steps:

1. Update link of the "previous" node, to point to the new node.



2. Update link of the new node, to point to the "next" node.



DECLARATION FOR LINKED LIST

```
Struct node ;
typedef struct Node *List ;
typedef struct Node *Position ;
int IsLast (List L) ;
int IsEmpty (List L) ;
position Find(int X, List L) ;
void Delete(int X, List L) ;
position FindPrevious(int X, List L) ;
position FindNext(int X, List L) ;
void Insert(int X, List L, Position P) ;
void DeleteList(List L) ;
Struct Node
{
int element ;
position Next ;
};
```

ROUTINE TO INSERT AN ELEMENT IN THE LIST

```
void Insert (int X, List L, Position P)
/* Insert after the position P*/
{
position Newnode;
Newnode = malloc (size of (Struct Node));
If (Newnode! = NULL)
{
Newnode ->Element = X;
Newnode ->Next = P-> Next;
P-> Next = Newnode;
```

Data / Info field Link /Null field
} }
INSERT (25, P, L)
ROUTINE TO CHECK WHETHER THE LIST IS EMPTY

```
int IsEmpty (List L) /*Returns 1 if L is empty */
{ if (L -> Next = = NULL)
return (1);
}
```

ROUTINE TO CHECK WHETHER THE CURRENT POSITION IS LAST

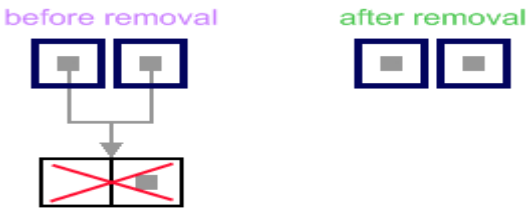
```
int IsLast (position P, List L) /* Returns 1 is P is the last position in L */
{ if (P->Next = = NULL)
Return
}
```

SINGLY-LINKED LIST. REMOVAL (DELETION) OPERATION.

There are four cases, which can occur while removing the node. These cases are similar to the cases in add operation. We have the same four situations, but the order of algorithm actions is opposite. Notice, that removal algorithm includes the disposal of the deleted node, which may be unnecessary in languages with automatic garbage collection (i.e., Java).

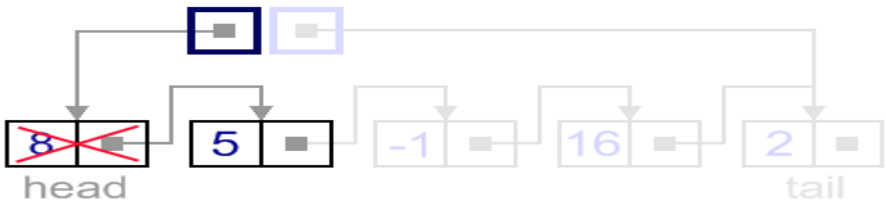
List has only one node

When list has only one node, which is indicated by the condition, that the head points to the same node as the tail, the removal is quite simple. Algorithm disposes the node, pointed by head (or tail) and sets both head and tail to *NULL*.



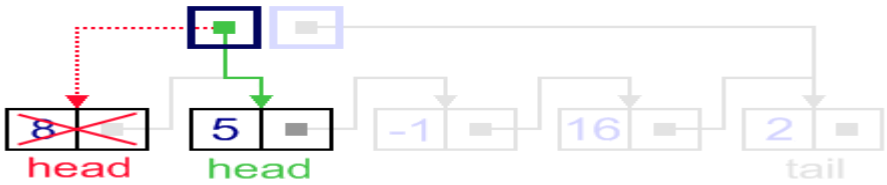
Remove first

In this case, first node (current head node) is removed from the list.

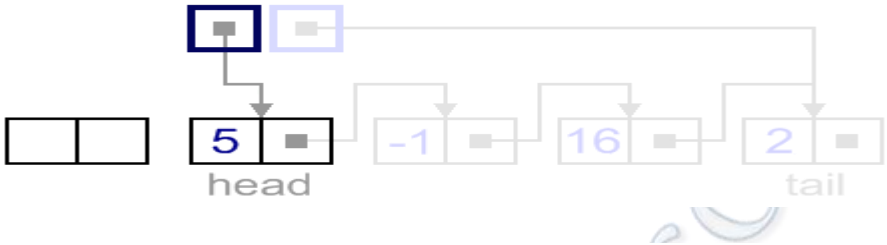


It can be done in two steps:

1. Update head link to point to the node, next to the head.

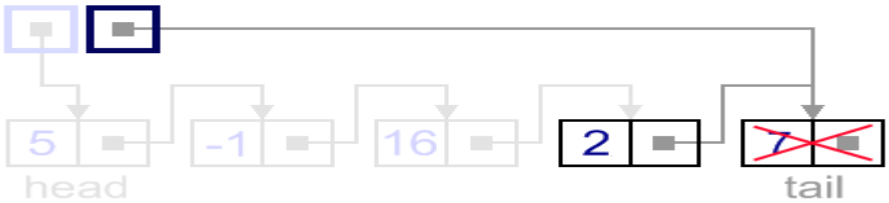


2. Dispose removed node.



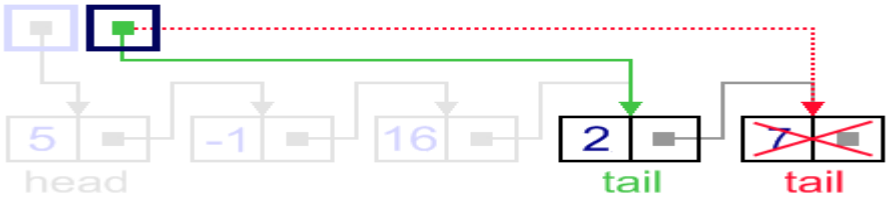
Remove last

In this case, last node (current tail node) is removed from the list. This operation is a bit more tricky, than removing the first node, because algorithm should find a node, which is previous to the tail first.

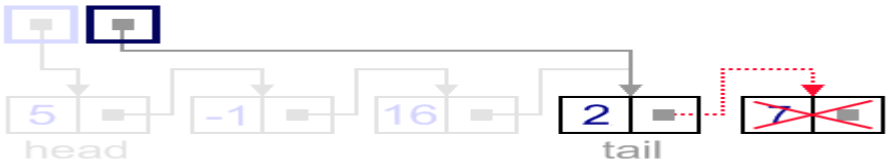


It can be done in three steps:

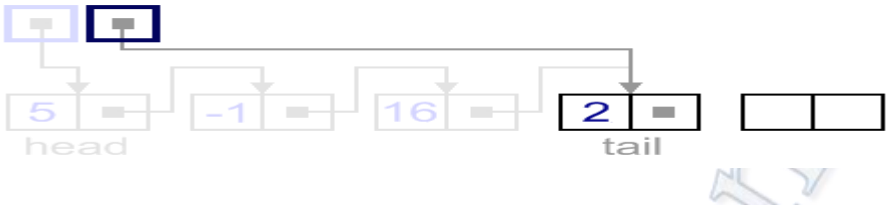
1. Update tail link to point to the node, before the tail. In order to find it, list should be traversed first, beginning from the head.



2. Set next link of the new tail to NULL.

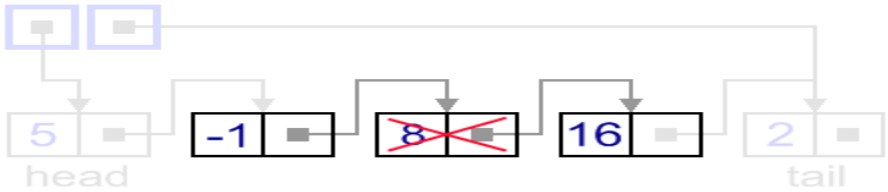


3. Dispose removed node.



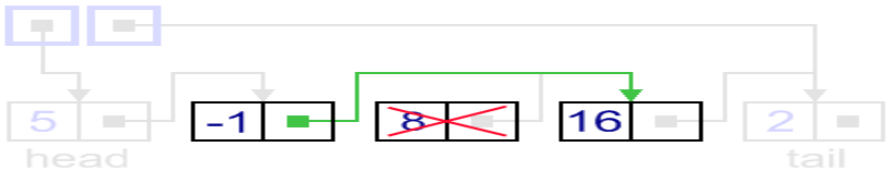
General case

In general case, node to be removed is **always located between** two list nodes. Head and tail links are not updated in this case.

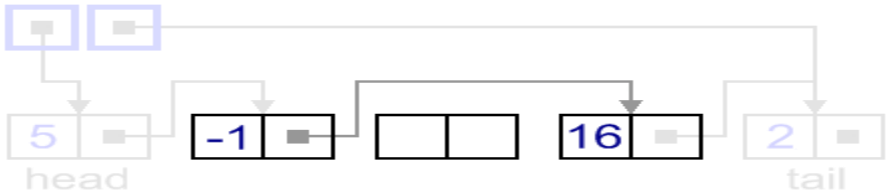


Such a removal can be done in two steps:

1. Update next link of the previous node, to point to the next node, relative to the removed node.



2. Dispose removed node.



ROUTINE TO DELETE AN ELEMENT FROM THE LIST

```
void Delete(int X, List L)
{
/* Delete the first occurence of X from the List */
```

```
position P, Temp;
P = Findprevious (X,L);
If (!IsLast(P,L))
{
Temp = P→Next;
P →Next = Temp→Next;
Free (Temp);
}
}
```

ROUTINE TO DELETE THE LIST

```
void DeleteList (List L)
{
position P, Temp;
P = L →Next;
L→Next = NULL;
while (P! = NULL)
{
Temp = P→Next
free (P);
P = Temp;
}
}
```

Search Operations in Singly Linked List

FIND ROUTINE

```
position Find (int X, List L)
{
/*Returns the position of X in L; NULL if X is not found */
position P;
P = L-> Next;
while (P! = NULL && P Element ! = X)
P = P->Next;
return P;
} }
```

FIND PREVIOUS ROUTINE

```
position FindPrevious (int X, List L)
{
/* Returns the position of the predecessor */
position P;
P = L;
while (P -> Next ! = Null && P ->Next Element ! = X)
P = P ->Next;
return P;
}
```

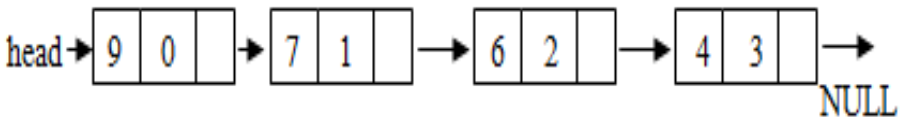
FINDNEXT ROUTINE

```
position FindNext (int X, List L)
{
/*Returns the position of its successor */
```

```
P = L ->Next;
while (P Next! = NULL && P Element ! = X)
P = P→Next;
return P→Next;
}
```

DOUBLE LINKED LIST

A Doubly linked list is a linked list in which each node has three fields namely data field,forward link (FLINK) and Backward Link (BLINK). FLINK points to the successor node in the list whereas BLINK points to the predecessor node. A double linked list is used to traverse among the nodes in both the directions.



STRUCTURE DECLARATION : -

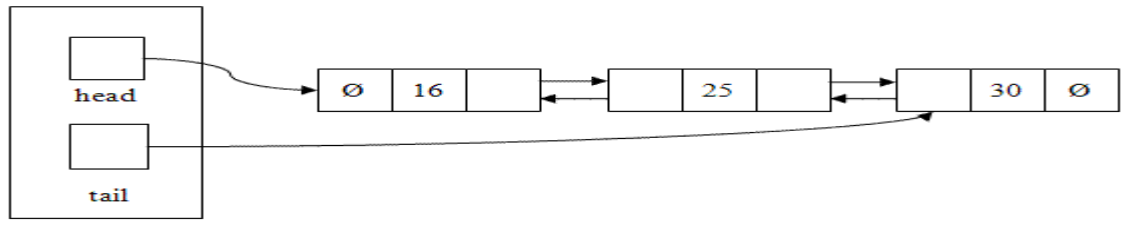
```
Struct Node
{
        int Element;
        Struct Node *FLINK;
        Struct Node *BLINK
};
```

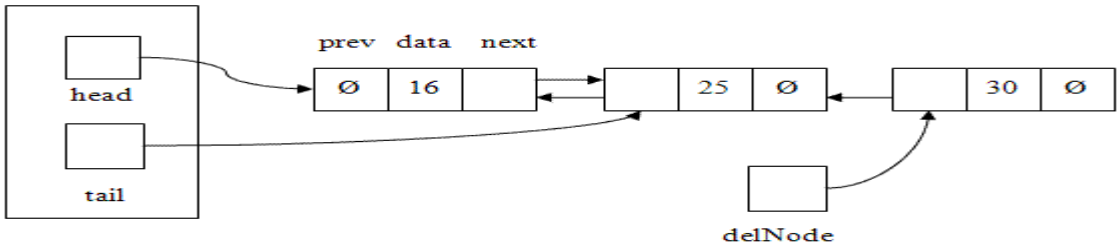
Inserting an Element

To insert an element in the list, the first task is to allocate memory for a new node, assign the element to be inserted to the info field of the node, and then the new node is placed at the appropriate position by adjusting appropriate pointers. Insertion in the list can take place at the following positions:

- At the beginning of the list
- At the end of the list
- After a given element
- Before a given element

Insertion at the Beginning of the List





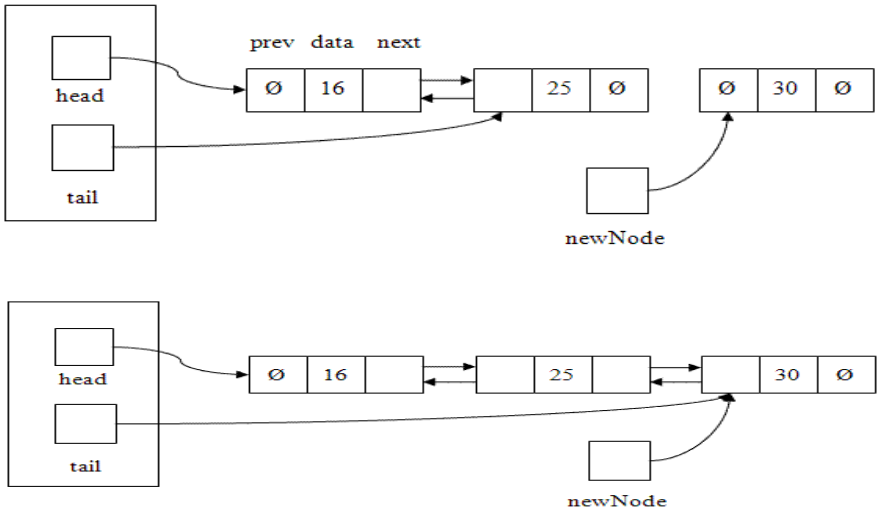
First, test whether the linked list is empty, if yes, then the element is inserted as the first and only one element by performing the following steps:

- Assign NULL to the next pointer and prev pointer fields of the new node
- Assign the address of the new node to head and tail pointer variables.

If the list is not empty, then the element is inserted as the first element of the list by performing the following steps:

- Assign NULL to the prev pointer field of the new node.
- Assign the value of the head variable (the address of the first element of the existing list) to the next pointer field of the new node.
- Assign the address of the new node to prev pointer field of the node currently pointed by head variable, i.e. first element of the existing list.
- Finally assign the address of the new node to the head variable

Inserting at the End of the List



First test whether the linked list is initially empty, if yes, then the element is inserted as the first and only one element by performing the following steps:

- Assign NULL value to the next pointer and prev pointer field of the new node
- Assign address of new node to head and tail pointer variable.

If the list is not empty, then element is inserted as the last element of the list by performing the following steps:

- Assign NULL value to the next pointer field of the new node.
- Assign value of the tail variable (the address of the last element of the existing list) to the prev pointer field of the new node.
- Assign address of the new node to the next pointer field of the node currently pointed by tail variable i.e. last element of the existing list.
- Finally assign the address of the new node to tail variable.

```
void insertatend (node **head, node **tail, int item)
{
   node *ptr;
   ptr = malloc(sizeof(node));
```

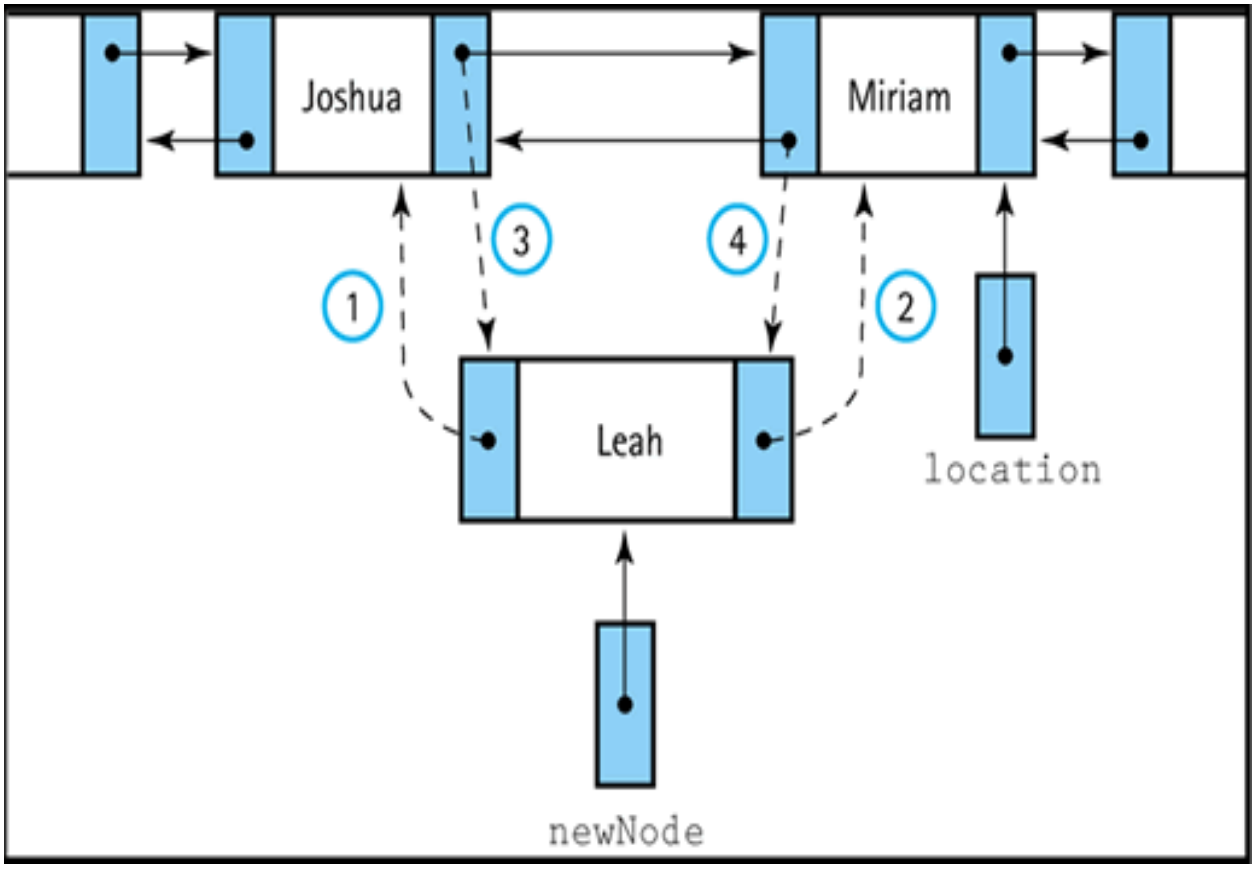
```
    ptr->info = item;
    if (*head == NULL)
    {
        ptr->next = ptr->prev=NULL;
        *head = *tail = ptr;
    }
    else
    {
        ptr->next=NULL;
        ptr->prev=*tail;
        (*tail)->next=ptr;
        *tail=ptr;
    }
}
```

Inserting before a Given Element

Find the location of after which you want to insert and store it in loc variable. If the location is NULL terminate process, else insert node using following steps

- Create a new node to insert and assign data(element which you want to insert) to new node.
- If loc-> prev =NULL means loc points to first node of linked list. So have to insert new node at the begining. Assign the following values
    - newnode->next=loc
    - newnode->prev=NULL
    - header=newnode (newnode becomes header node)
- else have to insert newnode between two nodes loc and loc->prev
    1. newnode->prev=loc->prev
    2. newnode->next=loc
    3. loc->prev->next=newnode
    4. loc->prev=newnode

Note: First create two forward links and then two backward links.In the following code newnode refers to ptr loc references to location and header refers to head .




```
void insertbeforeelement (node **head, int item, int before)
{
    node *ptr, *loc;
    ptr=*head;
    loc=search(ptr,before);
    if(loc==NULL)
        return;
    ptr=malloc(sizeof(node));
    ptr->info=item;
    if(loc->prev==NULL)
    {
```

```
        ptr->prev=NULL;
        loc->prev=ptr;
        ptr->next=*head;
        *head=ptr;
    }
    else
    {
        ptr->prev=loc->prev;
        ptr->next=loc;
        (loc->prev)->next=ptr;
        loc->prev=ptr;
    }
}
```

Inserting after a Given Element

Find the location of after which you want to insert and store it in loc variable. If the location is NULL terminate process, else insert node using following steps

- Create a new node to insert and assign data(element which you want to insert) to new node.
- If loc-> next =NULL means loc points to last node of linked list. So have to insert new node at the end. Assign the following values
    - loc-> next=newnode
    - newnode-> previous=loc
    - newnode->next=NULL
- else have to insert newnode between two nodes loc and loc->next
    - loc->next->prev=newnode
    - newnode->next=loc->next
    - newnode->prev=loc
    - loc->next=newnode

Note: First create two backward links and then two forward links. In the following code newnode refers to ptr and header refers to head.

```
void insertafterelement (node **head, node **tail, int item, int after)
{
   node *ptr, *loc;
   ptr = *head;
   loc = search(ptr,after);
   if(loc == NULL)
      return;
   ptr=malloc(sizeof(node));
   ptr->info = item;
   if(loc->next == NULL)
   {
      ptr->next = NULL;
      loc->next = ptr;
      ptr->prev = *tail;
      *tail = ptr;
   }
   else
   {
      ptr->prev = loc;
      ptr->next = loc->next;
      (loc->next)->prev = ptr;
      loc->next = ptr;
   }
}
```

REPRESENTING A POLYNOMIAL USING A LINKED LIST

A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term. However, for any polynomial operation , such as addition or multiplication of polynomials , you will find that the linked list representation is more easier to deal with. First of all note that in a polynomial all the terms may not be present, especially if it is going to be a very high order polynomial.

Consider

$5 x^{12} + 2 x^{9} + 4x^{7} + 6x^{5} + x^{2} + 12 x$

Now this 12th order polynomial does not have all the 13 terms (including the constant term).

It would be very easy to represent the polynomial using a linked list structure, where each node can hold information pertaining to a single term of the polynomial. Each node will need to store the variable x, the exponent and the coefficient for each term. It often does not matter whether the polynomial is in x or y. This information may not be very crucial for the intended operations on the polynomial. Thus we need to define a node structure to hold two integers , viz. exp and coff

Compare this representation with storing the same polynomial using an array structure. In the array we have to have keep a slot for each exponent of x, thus if we have a polynomial of order 50 but containing just 6 terms, then a large number of entries will be zero in the array. You will also see that it would be also easy to manipulate a pair of polynomials if they are

represented using linked lists.

Addition of two polynomials

Consider addition of the following polynomials

5 x12 + 2 x9 + 4x7 + 6x6 + x3

7 x8 + 2 x7 + 8x6 + 6x4 + 2x2 + 3 x + 40

The resulting polynomial is going to be

5 x12 + 2 x9 + 7 x8 + 6 x7 + 14x6 + 6x4 +x3

2x2 + 3 x + 40

Now notice how the addition was carried out. Let us say the result of addition is going to be stored in a third list. We started with the highest power in any polynomial. If there was no item having same exponent , we simply appended the term to the new list, and continued with the process. Wherever we found that the exponents were matching, we simply added the coefficients and then stored the term in the new list. If one list gets exhausted earlier and the other list still contains some lower order terms, then simply append the remaining terms to the new list.

Now we are in a position to write our algorithm for adding two polynomials.

Let phead1 , phead2 and phead3 represent the pointers of the three lists under consideration.

Let each node contain two integers exp and coff .

Let us assume that the two linked lists already contain relevant data about the two polynomials.

Also assume that we have got a function **append** to insert a new node at the end of the given list.

```
p1 = phead1;
p2 = phead2;
```

Let us call malloc to create a new node p3 to build the third list

```
p3 = phead3;
/* now traverse the lists till one list gets exhausted */
while ((p1 != NULL) || (p2 != NULL))
{
/ * if the exponent of p1 is higher than that of p2 then the next term in final list is going to be the node of p1* /
while (p1 ->exp > p2 -> exp )
{
p3 -> exp = p1 -> exp;
p3 -> coff = p1 -> coff ;
append (p3, phead3);
/* now move to the next term in list 1*/
p1 = p1 -> next;
}
/ * if p2 exponent turns out to be higher then make p3 same as p2 and append to final list * /
while (p1 ->exp < p2 -> exp )
{
p3 -> exp = p2 -> exp;
p3 -> coff = p2 -> coff ;
append (p3, phead3);
p2 = p2 -> next;
}
/* now consider the possibility that both exponents are same , then we must add the coefficients
```

to get the term for the final list */
while (p1 ->exp = p2 -> exp)
{
p3-> exp = p1-> exp;
p3->coff = p1->coff + p2-> coff ;
append (p3, phead3) ;
p1 = p1->next ;
p2 = p2->next ;
}
}
/* now consider the possibility that list2 gets exhausted , and there are terms remaining only in list1. So all those terms have to be appended to end of list3. However, you do not have to do it term by term, as p1 is already pointing to remaining terms, so simply append the pointer p1 to phead3
*/
if (p1 != NULL)
append (p1, phead3) ;
else
append (p2, phead3);
Now, you can implement the algorithm in C, and maybe make it more efficient.

Declaration For Linked List Implementation Of Polynomial ADT
Struct poly
{
int coeff ;
int power;
Struct poly *Next;
}*list 1, *list 2, *list 3;
CREATION OF THE POLYNOMIAL
poly create (poly *head1, poly *newnode1)
{
poly *ptr;
if (head1= =NULL)
{
head1 = newnode1;
return (head1);
}
Else
{
ptr = head1;
while (ptr next ! = NULL)
ptr = ptr→next;
ptr→next = newnode1;
} return (head1);
}
ADDITION OF TWO POLYNOMIALS

```
void add ( )
{
poly *ptr1, *ptr2, *newnode;
ptr1 = list1; ptr2 = list2;
while (ptr1! = NULL && ptr2! = NULL)
{
newnode = malloc (sizeof (Struct poly));
if (ptr1→power = = ptr2 power)
{
newnode→coeff = ptr1→coeff + ptr2→coeff;
newnode→power = ptr1→power;
newnode→next = NULL;
list 3 = create (list3, newnode);
ptr1 = ptr1→next;
ptr2 = ptr2→next;
}
else
{
if (ptr1→power > ptr2→ power)
{
newnode→coeff = ptr1→coeff;
newnode→power = ptr1→power;
newnode→next = NULL;
list3 = create (list3, newnode);
ptr1 = ptr1→next;
}
Else
{
newnode→coeff = ptr2→coeff;
newnode→power = ptr2→power;
newnode→next = NULL;
list3 = create (list3, newnode);
ptr2 = ptr2→ next;
}
}
}
```

SUBTRACTION OF TWO POLYNOMIAL

```
void sub ( )
{
poly *ptr1, *ptr2, *newnode;
ptr1 = list1 ;
ptr2 = list 2;
while (ptr1! = NULL && ptr2! = NULL)
{
newnode = malloc (sizeof (Struct poly));
if (ptr1 power = = ptr2 power)
```

```
{
newnode→coeff = (ptr1 coeff) - (ptr2 coeff);
newnode→power = ptr1 power;
newnode→next = NULL;
list3 = create (list 3, newnode);
ptr1 = ptr1→next;
ptr2 = ptr2→next;
}
Else
{
if (ptr1→power > ptr2→power)
{ newnode→coeff = ptr1→coeff;
newnode→power = ptr1→power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr1 = ptr1→next;
}
Else
{ newnode→coeff = - (ptr2 coeff);
newnode→power = ptr2→ power;
newnode→next = NULL;
list 3 = create (list 3, newnode);
ptr2 = ptr2 next;
}
}
}
}
```

POLYNOMIAL DIFFERENTIATION

```
void diff ( )
{
poly *ptr1, *newnode;
ptr1 = list 1;
while (ptr1 ! = NULL)
{
newnode = malloc (sizeof (Struct poly));
newnode coeff = ptr1 coeff *ptr1 power;
newnode power = ptr1 power - 1;
newnode next = NULL;
list 3 = create (list 3, newnode);
ptr1 = ptr1→next;
}
}
```

STACKS

In computer science, a stack is a particular kind of abstract data type or collection in which the principal (or only) operations on the collection are the addition of an entity to the collection, known as push and removal of an entity, known as pop.[1] The relation between the

push and pop operations is such that the stack is a Last-In-First-Out (LIFO) data structure. In a LIFO data structure, the last element added to the structure must be the first one to be removed. This is equivalent to the requirement that, considered as a linear data structure, or more abstractly a sequential collection, the push and pop operations occur only at one end of the structure, referred to as the top of the stack. Often a peek or top operation is also implemented, returning the value of the top element without removing it.

A stack may be implemented to have a bounded capacity. If the stack is full and does not contain enough space to accept an entity to be pushed, the stack is then considered to be in an overflow state. The pop operation removes an item from the top of the stack. A pop either reveals previously concealed items or results in an empty stack, but, if the stack is empty, it goes into underflow state, which means no items are present in stack to be removed.

Implementation

In most high level languages, a stack can be easily implemented either through an array or a linked list. What identifies the data structure as a stack in either case is not the implementation but the interface: the user is only allowed to pop or push items onto the array or linked list, with few other helper operations. The following will demonstrate both implementations, using C.

Array Implementation

The **array implementation** aims to create an array where the first element (usually at the zero-offset) is the bottom. That is, array[0] is the first element pushed onto the stack and the last element popped off. The program must keep track of the size, or the length of the stack. The stack itself can therefore be effectively implemented as a two-element structure in C:

```
typedefstruct {
size_t size;
int items[STACKSIZE];
} STACK;
```

The push() operation is used both to initialize the stack, and to store values to it. It is responsible for inserting (copying) the value into the ps->items[] array and for incrementing the element counter (ps->size). In a responsible C implementation, it is also necessary to check whether the array is already full to prevent an overrun.

```
void push(STACK *ps, int x)
{
if (ps->size == STACKSIZE) {
fputs("Error: stack overflow\n", stderr);
abort();
   } else
ps->items[ps->size++] = x;
}
```

The pop() operation is responsible for removing a value from the stack, and decrementing the value of ps->size. A responsible C implementation will also need to check that the array is not already empty.

```
int pop(STACK *ps)
{
if (ps->size == 0){
fputs("Error: stack underflow\n", stderr);
abort();
   } else
returnps->items[--ps->size];
}
```

If we use a dynamic array, then we can implement a stack that can grow or shrink as much as needed. The size of the stack is simply the size of the dynamic array. A dynamic array is a very efficient implementation of a stack, since adding items to or removing items from the end of a dynamic array is amortized O(1) time.

Applications:

Stacks have numerous applications. We see stacks in everyday life, from the books in our library, to the blank sheets of paper in our printer tray. All of them follow the *Last In First Out* (LIFO) logic, that is when we add a book to a pile of books, we add it to the top of the pile, whereas when we remove a book from the pile, we generally remove it from the top of the pile.

Given below are a few applications of stacks in the world of computers:

1. Converting a decimal number into a binary number
2. Towers of Hanoi
3. Evaluation of an infix expression
4. Evaluation of prefix expression
5. Conversion of an Infix expression into a Postfix expression

1. Evaluating Arithmetic Expression

To evaluate an arithmetic expressions, first convert the given infix expression to postfix expression and then evaluate the postfix expression using stack.

Infix to Postfix Conversion

Read the infix expression one character at a time until it encounters the delimiter. "#"
Step 1 : If the character is an operand, place it on to the output.
Step 2 : If the character is an operator, push it onto the stack. If the stack operator has a higher
or equal priority than input operator then pop that operator from the stack and place it onto the output.
Step 3 : If the character is a left paraenthesis, push it onto the stack.

Step 4 : If the character is a right paraenthesis, pop all the operators from the stack till it encounters left parenthesis, discard both the parenthesis in the output.

Evaluating Postfix Expression

Read the postfix expression one character at a time until it encounters the delimiter `#'. Step 1 : - If the character is an operand, push its associated value onto the stack. Step 2 : - If the character is an operator, POP two values from the stack, apply the operator to them and push the result onto the stack.

4.2 THE QUEUE ADT

Queue Model

A Queue is a linear data structure which follows First In First Out (FIFO) principle, in which insertion is performed at rear end and deletion is performed at front end.
Example : Waiting Line in Reservation Counter.

Operations on Queue

The fundamental operations performed on queue are

1. Enqueue
2. Dequeue

Enqueue : The process of inserting an element in the queue.

Dequeue : The process of deleting an element from the queue.

Exception Conditions

Overflow : Attempt to insert an element, when the queue is full is said to be overflow condition.
Underflow : Attempt to delete an element from the queue, when the queue is empty is said to be underflow.

Implementation of Queue

Queue can be implemented using arrays and pointers.

Array Implementation

In this implementation queue Q is associated with two pointers namely rear pointer and front pointer.

To insert an element X onto the Queue Q, the rear pointer is incremented by 1 and then set Queue [Rear] = X

To delete an element, the Queue [Front] is returned and the Front Pointer is incremented by 1.

ROUTINE TO ENQUEUE

```
void Enqueue (int X)
 {
        if (rear > = max _ Arraysize)
        print (" Queue overflow");
        else
        {       Rear = Rear + 1;
                Queue [Rear] =
                X;
        }
 }
```

ROUTINE FOR DEQUEUE

```
void dequeue ( )
{
        if (Front < 0)
        print (" Queue Underflow");
        else
        {
        X = Queue [Front];
        if (Front = = Rear)
        {
        Front = 0;
        Rear = -1;
        }
        else
        Front = Front + 1 ;
}        }
```

In **Dequeue** operation, if Front = Rear, then reset both the pointers to their initial values. (i.e. F = 0, R = -1)

APPLICATIONS OF QUEUE

- Batch processing in an operating system
- To implement Priority Queues.
- Priority Queues can be used to sort the elements using Heap Sort.
- Simulation and Mathematics user Queueing theory.
- Computer networks where the server takes the jobs of the client as per the queue strategy.
- **TEXT BOOKS:**
- 1. Deitel and Deitel, “C++, How To Program”, Fifth Edition, Pearson Education, 2005.
- 2. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C++”, Third Edition, Addison-Wesley, 2007.
-

- **REFERENCES:**

- 1. Bhushan Trivedi, “Programming with ANSI C++, A Step-By-Step approach”, Oxford University Press, 2010.
- 2. Goodrich, Michael T., Roberto Tamassia, David Mount, “Data Structures and Algorithms in C++”, 7th Edition, Wiley. 2004.
- 3. Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, "Introduction to Algorithms", Second Edition, Mc Graw Hill, 2002.
- 4. Bjarne Stroustrup, “The C++ Programming Language”, 3rd Edition, Pearson Education, 2007.
- 5. Ellis Horowitz, Sartaj Sahni and Dinesh Mehta, “Fundamentals of Data Structures in C++”, Galgotia Publications, 2007.