Chapter 4: Static and Dynamic list

4.1. Introduction to list

A ***list*** to be a finite, ordered sequence of data items known as elements. This is close to the mathematical concept of a sequence. "Ordered" in this definition means that each element has a position in the list. So the term "ordered" in this context does not mean that the list elements are sorted by value.

Each list element must have some data type. In the simple list implementations discussed in this chapter, all elements of the list are usually assumed to have the same data type, although there is no conceptual objection to lists whose elements have differing data types if the application requires it. The operations defined as part of the list ADT do not depend on the elemental data type. For example, the list ADT can be used for lists of integers, lists of characters, lists of payroll records, even lists of lists.

A list is said to be ***empty*** when it contains no elements. The number of elements currently stored is called the ***length*** of the list. The beginning of the list is called the ***head***, the end of the list is called the ***tail***.

4.2. The list ADT

What basic operations do we want our lists to support? Our common intuition about lists tells us that a list should be able to grow and shrink in size as we insert and remove elements. We should be able to insert and remove elements from anywhere in the list. We should be able to gain access to any element's value, either to read it or to change it. We must be able to create and clear (or reinitialize) lists. It is also convenient to access the next or previous element from the "current" one.

Now we can define the ADT for a list object in terms of a set of operations on that object. We will use an interface to formally define the list ADT. `List` defines the member functions that any list implementation inheriting from it must support, along with their parameters and return types.

True to the notion of an ADT, an interface does not specify how operations are implemented. The two common implementations of List are Linked List and Array List.

The code below presents the list ADT. Any implementation for a ***container class*** such as a list should be able to support different data types for the elements. One way to do this in Java is to store data values of type `Object`. Languages that support generics (Java) or templates (C++) give more control over the element types.

```
class List
{ // List class ADT

  // Remove all contents from the list, so it is once again empty
public:
  virtual void clear() =0;

  // Inserts an item into the list at position index
  virtual bool insert(const ListItemType& newItem) =0;
```

```
  // Append "it" at the end of the list
  // The client must ensure that the list's capacity is not exceeded
  virtual bool append(const ListItemType& newItem) =0;

  // Deletes an item from the list at a given position
  virtual ListItemType remove() =0;

  // Set the current position to the start of the list
  virtual void moveToStart() =0;

  // Set the current position to the end of the list
  virtual void moveToEnd() =0;

   // Move the current position one step left, no change if already at
beginning
  virtual void prev() =0;

  // Move the current position one step right, no change if already at end
  virtual void next() =0;

  //Return the number of items stored in the list
  virtual int length() =0;

  // Return the position of the current element
  virtual int currPos() =0;

  // Set the current position to "pos"
  virtual bool moveToPos(int pos) =0;
 // Return true if current position is at end of the list
  virtual bool isAtEnd() =0;

  // Return the current element
  virtual ListItemType getValue() =0;

};
```

4.3. Implementation of lists

A list or sequence is an abstract data type that represents a countable number of ordered values, where the same value may occur more than once. An instance of a list is a computer representation of the mathematical concept of a tuple or finite sequence. The two common implementations of List are Linked List and Array List.

Array-based list implementation

```
class AList : public List {
  ListItemType listArray[MAX_SIZE];  //Array holding list elements
  int listSize;   //Current number of list items
  int curr;   //Position of current element
  public:
  //Constructor
  // Create a new list element with maximum size "MAX_SIZE"
  AList() : listSize(0) {
    //Initial the array
    for (int k = 0; k < MAX_SIZE; k++) listArray[k] = 0;
```

```
  } //end constructor

  bool isEmpty() const {
    return listSize == 0;
  }
  void clear() {             // Reinitialize the list
    listSize = curr = 0;     // Simply reinitialize values
  }

  // Insert "it" at current position
  bool insert(const ListItemType& it) {
    if (listSize >= MAX_SIZE) return false;
    for (int i = listSize; i > curr; i--) //Shift elements up
      listArray[i] = listArray[i-1];      //to make room
    listArray[curr] = it;
    listSize++;                           //Increment list size
    return true;
  }

  // Append "it" to list
  bool append(const ListItemType& it) {
    if ( listSize >= MAX_SIZE ) return false;
    listArray[listSize++] = it;
    return true;  }

  // Remove and return the current element
  ListItemType remove() {
    if( (curr < 0) || (curr >= listSize) )  // No current element
      return 0;
    ListItemType it = listArray[curr];      // Copy the element
    for (int i = curr; i < listSize; i++)   // Shift them down
      listArray[i] = listArray[i+1];
    listSize--;                             // Decrement size
    return it;
  }

  void moveToStart() { curr = 0; }          // Set to front
  void moveToEnd() { curr = listSize; }     // Set to end
  void prev() { if (curr != 0) curr--; }    // Move left
  void next() { if (curr < listSize) curr++; } // Move right
  int length() { return listSize; }         // Return list size
  int currPos() { return curr; }            // Return current position

  // Set current list position to "pos"
  bool moveToPos(int pos) {
    if ((pos < 0) || (pos > listSize)) return false;
    curr = pos;
    return true;
  }

  // Return true if current position is at end of the list
  bool isAtEnd() { return curr == listSize; }

  // Return the current element
  ListItemType getValue() {
    if ((curr < 0) || (curr >= listSize)) // No current element
      return 0;
    return listArray[curr];
  }
};
```

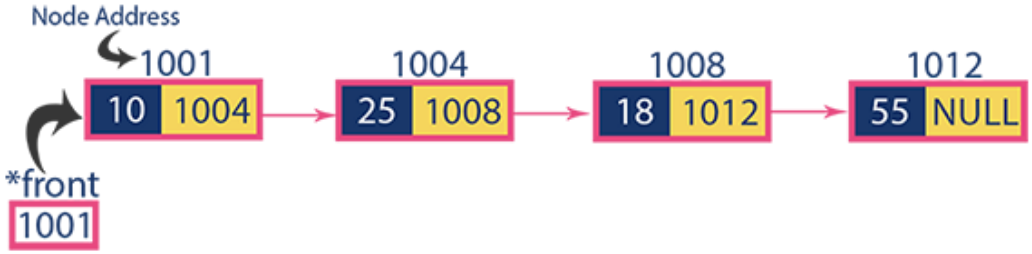
Linked list (Dynamic implementation of list)

Another list implementation is called a linked list. The linked list uses ***dynamic memory allocation***, that is, it allocates memory for new list elements as needed.

Introduction to linked list

The following diagram illustrates the linked list concept. The Figure 1 (a) shows a node with two fields. The Figure 1(b) shows a linked list with four nodes that are "linked" together. Each node has two fields-called data and link fields. The data field also called information field holds the actual data of the list. The link field holds an address of the next node in the list so that it point to the next node. If a node is the last node in the list, it doesn't point to any node. Therefore, the link field of last node contains NULL value. Sometimes, a null value in link field is also denoted by a diagonal slash.

(a)



(b)

Figure 1. (a) A node of a linked list (b) a linked list containing 4 nodes

A linked list is a linear data structure that contains a sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called "Node". The list built from such nodes is called a ***singly linked list***, or a ***one-way list***, because each list node has a single pointer to the next node on the list. In any ***single linked list,*** every "Node" contains two fields, data field (also called information field), and the link field (also called next field). The data field is used to store actual value of the node and next field is used to store the address of next node in the list.

In a singly linked list, the address of the first node is always stored in a reference node known as "front" (Sometimes it is also known as "head"). Always the link field (reference part) of the last node must be NULL.

Based on how a node is linked to other nodes in a list, they are of following types:

1. Singly linked list (*or simply called as linked list*)

2. Doubly linked list and
3. Circular linked list

The doubly linked list and circular linked list are discussed later in the separate section.

Operations on linked list

The following operations are performed on a Single Linked List

- Insertion
- Deletion
- Traversal (Display)

1. Insertion operations

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

a) Inserting node at beginning of the list
b) Inserting node at end of the list
c) Inserting node after a node in the list

a) Inserting node at beginning of the list

1. Create a **newNode**
2. Assign value to **newNode**
3. Check whether list is **Empty** (**head** == **NULL**)
4. If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
5. If it is **Not Empty** then, set **newNode→next** = **head** and **head** = **newNode**.

b) Inserting node at end of the list

1. Create a **newNode**
2. Assign value to **newNode** and set **newNode → next** as **NULL**.
3. Check whether list is **Empty** (**head** == **NULL**).
4. If it is **Empty** then, set **head** = **newNode**.
5. If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
6. Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
7. Set **temp → next** = **newNode.**

c) Inserting node after a node in the list

1. Create a **newNode** and assign given value.
2. Check whether list is **Empty** (**head** == **NULL**)
3. If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
4. If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

5. Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
6. Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list and Insertion is not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
7. Finally, Set '**newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

2. Deletion operations

In a single linked list, the deletion operation can be performed in three ways. They are as follows:

a) Deleting node at beginning of the list
b) Deleting node at end of the list
c) Deleting node after a node in the list

a) Deleting node at beginning of the list

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty. Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
4. Check whether list is having only one node (**temp → next** == **NULL**)
5. If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
6. If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

b) Deleting node at end of the list

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty. Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
4. Check whether list has only one Node (**temp1 → next** == **NULL**)
5. If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
6. If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until it reaches to the last node in the list. (until **temp1 → next** == **NULL**)
7. Finally, Set **temp2 → next** = **NULL** and delete **temp1**.

c) Deleting node after a node in the list

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
3. If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.

4. Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
5. If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
6. If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
7. If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
8. If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
9. If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
10. If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
11. If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
12. If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

3. Traversal operation (displaying elements)

We can use the following steps to display the elements of a single linked list:

1. Check whether list is **Empty** (**head** == **NULL**)
2. If it is **Empty** then, display **'List is Empty.'** and terminate the function.
3. If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
4. Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
5. Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

Linked list ADT

```
struct node
{
      int data;          // Data
      node *next;        // A reference to the next node
};

class Linked_List
{
      node *head;        // points to first node of list
      public:
      Linked_List()
      {
            front = NULL;
      }
      void add_front(int );
      void add_after(node* , int );
      void add_end(int );
      void delete_front(node*);
      void delete_after(node*);
```

```
            void delete_end(node*);
            void traverse();
      };
```

Linked implementation of list

Because a list node is a distinct object (as opposed to simply a cell in an array), it is good practice to make a separate list node class. The list built from such nodes is called a ***singly linked list***, or a ***one-way list***, because each list node has a single pointer to the next node on the list.

*(**Student practice:** implement the linked implementation of list using C++). Hint is given in C:*

```
void add_front(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(head == NULL)
   {
      newNode->next = NULL;
      head = newNode;
   }
   else
   {
      newNode->next = head;
      head = newNode;
   }
   printf("\nOne node inserted!!!\n");
}
void add_end(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode->next = NULL;
   if(head == NULL)
      head = newNode;
   else
   {
      struct Node *temp = head;
      while(temp->next != NULL)
      temp = temp->next;
      temp->next = newNode;
   }
   printf("\nOne node inserted!!!\n");
}

void delete_front()
{
   if(head == NULL)
      printf("\n\nList is Empty!!!");
   else
   {
      struct Node *temp = head;
      if(head->next == NULL)
```

```
      {
       head = NULL;
       free(temp);
      }
      else
      {
      head = temp->next;
      free(temp);
      printf("\nOne node deleted!!!\n\n");
      }
   }
}
void delete_end()
{
   if(head == NULL)
   {
      printf("\nList is Empty!!!\n");
   }
   else
   {
      struct Node *temp1 = head,*temp2;
      if(head->next == NULL)
      head = NULL;
      else
      {
       while(temp1->next != NULL)
       {
          temp2 = temp1;
          temp1 = temp1->next;
       }
       temp2->next = NULL;
      }
      free(temp1);
      printf("\nOne node deleted!!!\n\n");
   }
}
void traverse()
{
   if(head == NULL)
   {
      printf("\nList is Empty\n");
   }
   else
   {
      struct Node *temp = head;
      printf("\n\nList elements are - \n");
      while(temp->next != NULL)
      {
       printf("%d --->",temp->data);
       temp = temp->next;
      }
      printf("%d --->NULL",temp->data);
   }
}
```

Comparison of list implementations (Array-based vs Linked List)

Now that you have seen two substantially different implementations for lists, it is natural to ask which is better. In particular, if you must implement a list for some task, which implementation should you choose?

Space Comparison

Given a collection of elements to store, they take up some amount of space whether they are simple integers or large objects with many fields. Any container data structure like a list then requires some additional space to organize the elements being stored. This additional space is called ***overhead***.

Array-based lists have the disadvantage that their size must be predetermined before the array can be allocated. Array-based lists cannot grow beyond their predetermined size. Whenever the list contains only a few elements, a substantial amount of space might be tied up in a largely empty array. This empty space is the overhead required by the array-based list.

Linked lists have the advantage that they only need space for the objects actually on the list. There is no limit to the number of elements on a linked list, as long as there is ***free store*** memory available. The amount of space required by a linked list is $\Theta(n)$, while the space required by the array-based list implementation is $\Omega(n)$, but can be greater.

Array-based lists have the advantage that there is no wasted space for an individual element. ***Linked lists*** require that an extra pointer for the `next` field be added to every list node. So the linked list has these `next` pointers as overhead. If the element size is small, then the overhead for links can be a significant fraction of the total storage. When the array for the ***array-based*** list is completely filled, there is no wasted space, and so no overhead. The ***array-based*** list will then be more space efficient, by a constant factor, than the ***linked*** implementation.

A simple formula can be used to determine whether the array-based list or the linked list implementation will be more space efficient in a particular situation. Call n the number of elements currently in the list, P the size of a pointer in storage units (typically four bytes), E the size of a data element in storage units (this could be anything, from one bit for a Boolean variable on up to thousands of bytes or more for complex records), and D the maximum number of list elements that can be stored in the array. The amount of space required for the array-based list is DE, regardless of the number of elements actually stored in the list at any given time. The amount of space required for the linked list is n(P+E). The smaller of these expressions for a given value n determines the more space-efficient implementation for n elements. In general, the linked implementation requires less space than the array-based implementation when relatively few elements are in the list. Conversely, the array-based implementation becomes more space efficient when the array is close to full. Using the equation, we can solve for n to determine the ***break-even point*** beyond which the array-based implementation is more space efficient in any particular situation. This occurs when

n>DE/(P+E)

If P=E, then the break-even point is at D/2. This would happen if the element field is either a four-byte `int` value or a pointer, and the `next` field is a typical four-byte pointer. That is, the array-based

implementation would be more efficient (if the link field and the element field are the same size) whenever the array is more than half full.

As a rule of thumb, linked lists are more space efficient when implementing lists whose number of elements varies widely or is unknown. Array-based lists are generally more space efficient when the user knows in advance approximately how large the list will become, and can be confident that the list will never grow beyond a certain limit.

Time Comparison

Array-based lists are faster for access by position. Positions can easily be adjusted forwards or backwards by the next and prev methods. These operations always take Θ(1) time. In contrast, singly linked lists have no explicit access to the previous element, and access by position requires that we march down the list from the front (or the current position) to the specified position. Both of these operations require Θ(n) time in the average and worst cases, if we assume that each position on the list is equally likely to be accessed on any call to prev or moveToPos.

Given a pointer to a suitable location in the list, the insert and remove methods for linked lists require only Θ(1) time. Array-based lists must shift the remainder of the list up or down within the array. This requires Θ(n) time in the average and worst cases. For many applications, the time to insert and delete elements dominates all other operations. For this reason, linked lists are often preferred to array-based lists.

When implementing the array-based list, an implementor could allow the size of the array to grow and shrink depending on the number of elements that are actually stored. This data structure is known as a dynamic array. For example, both the Java and C++/STL Vector classes implement a dynamic array, and JavaScript arrays are always dynamic. Dynamic arrays allow the programmer to get around the limitation on the traditional array that its size cannot be changed once the array has been created. This also means that space need not be allocated to the dynamic array until it is to be used. The disadvantage of this approach is that it takes time to deal with space adjustments on the array. Each time the array grows in size, its contents must be copied. A good implementation of the dynamic array will grow and shrink the array in such a way as to keep the overall cost for a series of insert/delete operations relatively inexpensive, even though an occasional insert/delete operation might be expensive.

Array vs Linked List

Array	Linked List
Array is a collection of elements having same data type with common name.	Linked list is an ordered collection of elements which are connected by links.
Elements can be accessed randomly.	Elements cannot be accessed randomly. It can be accessed only sequentially.

Array elements can be stored in consecutive manner in memory.	Linked list elements can be stored at any available place as address of node is stored in previous node.
Insert and delete operation takes more time in array.	Insert and delete operation cannot take more time. It performs operation in fast and in easy way.
Memory is allocated at compile time.	Memory is allocated at run time.
It can be single dimensional, two dimensional or multidimensional.	It can be singly, doubly or circular linked list.
Each array element is independent and does not have a connection with previous element or with its location.	Location or address of element is stored in the link part of previous element or node.
Array elements cannot be added, deleted once it is declared.	The nodes in the linked list can be added and deleted from the list.
In array, elements can be modified easily by identifying the index value.	In linked list, modifying the node is a complex process.
Pointer cannot be used in array. So, it does not require extra space in memory for pointer.	Pointers are used in linked list. Elements are maintained using pointers or links. So, it requires extra memory space for pointers.

4.4. Queues as a list

A queue is an ordered list of elements. A list is also an ordered list of elements. An ordinary queue always works in first in first out (FIFO) fashion. All the elements get inserted at the *REAR* and removed from the *FRONT* of the queue. You can add an element anywhere in the list, change an element anywhere in the list, or remove an element from any position in the list.

However, both queues and list are the order list of elements. These both are linear data structure. These both data structures can be implemented using array and linked list. They are similar. They are only different in how operations are permitted on the elements they store.

The below table shows major differences between a Static Queue and a Singly Linked List.

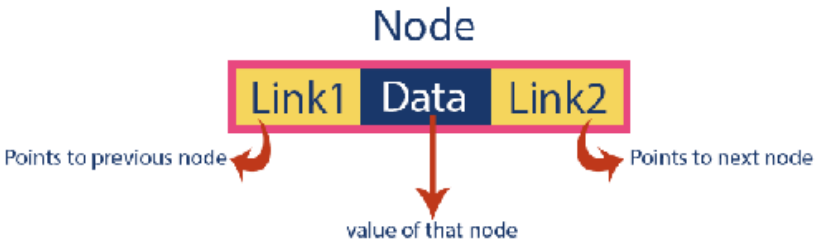
STATIC QUEUE	SINGLY LINKED LIST

Queue is a collection of one or more elements arranged in memory in a contiguous fashion.	A linked list is a collection of one or more elements arranged in memory in a dis-contiguous fashion.
Static Queue is always fixed size.	List size is never fixed.
In Queue, only one and single type of information is stored because static Queue implementation is through Array.	List also stored the address for the next node along with its content.
Static Queue is index based.	Singly linked list is reference based.
Insertion can always be performed on a single end called *REAR* and deletion on the other end called *FRONT*.	Insertion as well as deletion can performed anywhere within the list.
Queue is always based on FIFO.	List may be based on FIFI or LIFO etc.
Queue have two pointer FRONT and REAR.	While List has only one pointer basically called HEAD.

4.5. Doubly linked lists and its advantages

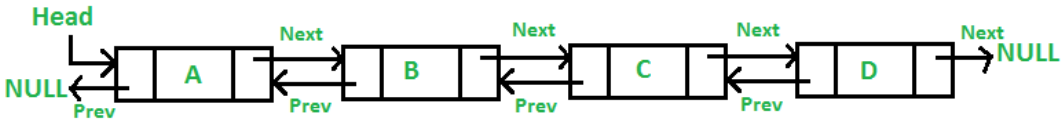
In a singly linked list, every node has a link to its next node in the sequence. So, we can traverse from one node to another node only in one direction and we cannot traverse back. We can solve this kind of problem by using a doubly linked list. A doubly linked list is a sequence of elements in which every element has links to its previous element and next element in the list.

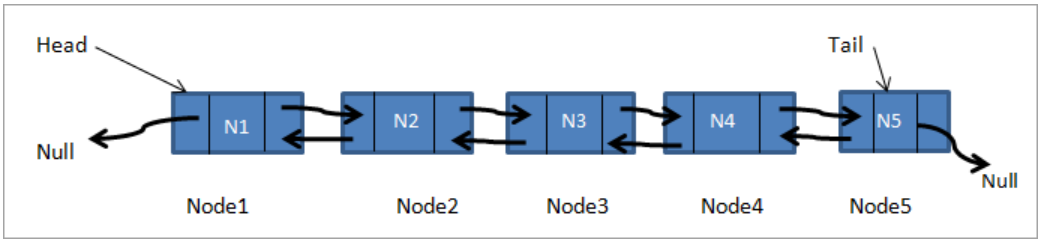
In a doubly linked list, every node has a link to its previous node and next node. So, we can traverse forward by using the next field and can traverse backward by using the previous field. A doubly linked list allows convenient access from a list node to the next node and also to the preceding node on the list. Every node in a doubly linked list contains three fields and they are shown in the following figure.



Here, **'link1'** field is used to store the address of the previous node in the sequence, **'link2'** field is used to store the address of the next node in the sequence and **'data'** field is used to store the actual value of that node.

Example





In doubly linked list, the first node must be always pointed by head. The head is a pointer and always contains the address of first node. Always the previous field of the first node must be NULL. Always the next field of the last node must be NULL.

Advantages:

- The doubly linked list can be traversed in forward as well as backward directions, unlike singly linked list which can be traversed in the forward direction only.
- Delete operation in a doubly-linked list is more efficient when compared to singly list when a given node is given. In a singly linked list, as we need a previous node to delete the given node, sometimes we need to traverse the list to find the previous node. This hits the performance.
- Insertion operation can be done easily in a doubly linked list when compared to the singly linked list.

Disadvantages:

- As the doubly linked list contains one more extra pointer i.e. previous, the memory space taken up by the doubly linked list is larger when compared to the singly linked list.
- Since two pointers are present i.e. previous and next, all the operations performed on the doubly linked list have to take care of these pointers and maintain them thereby resulting in a performance bottleneck.

Applications of Doubly Linked List

A doubly linked list can be applied in various real-life scenarios and applications as discussed below.

- A Deck of cards in a game is a classic example of a doubly linked list. Given that each card in a deck has the previous card and next card arranged sequentially, this deck of cards can be easily represented using a doubly linked list.
- Browser history/cache – The browser cache has a collection of URLs and can be navigated using the forward and back buttons is another good example that can be represented as a doubly linked list.
- Most recently used (MRU) also can be represented as a doubly linked list.
- Other data structures like Stacks, hash table, the binary tree can also be constructed or programmed using a doubly linked list.

4.6. Implementation of Doubly Linked List

Node of double linked list

A node in doubly linked list can be defined as:

```
struct node
{
      int data;          // Data
      node *prev;        // A reference to the previous node
      node *next;        // A reference to the next node
};
```

Operations on doubly linked list

Now we define our class Doubly Linked List. It has the following methods:

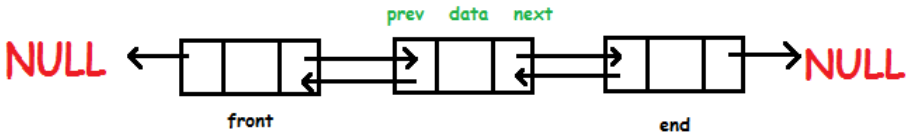
- **add_front:** Adds a new node in the beginning of list
- **add_after:** Adds a new node after another node
- **add_before:** Adds a new node before another node
- **add_end:** Adds a new node in the end of list
- **delete:** Removes the node
- **forward_traverse:** Traverse the list in forward direction
- **backward_traverse:** Traverse the list in backward direction

Doubly linked list ADT

```
class Doubly_Linked_List
{
      node *front;       // points to first node of list
      node *tail;        // points to last node of list
      public:
      Doubly_Linked_List()
      {
            front = NULL;
            tail = NULL;
      }
```

```
        void add_front(int );
        void add_after(node* , int );
        void add_before(node* , int );
        void add_end(int );
        void delete_node(node*);
        void forward_traverse();
        void backward_traverse();
};
```

Implementation of doubly linked list operations



Basic Operations

Following are some of the operations that we can perform on a doubly linked list.

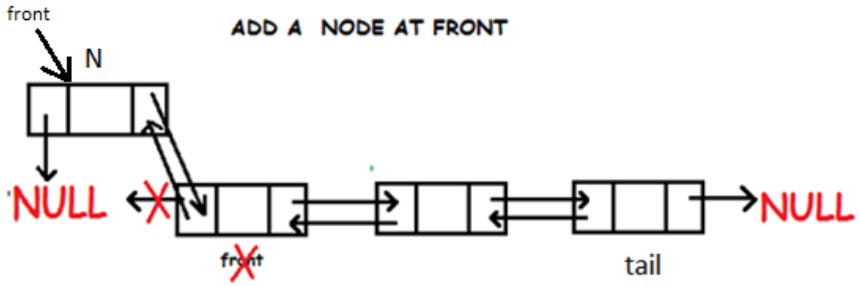
4. Insertion

Insertion operation of the doubly linked list inserts a new node in the linked list. Depending on the position where the new node is to be inserted, we can have the following insert operations.

a) **Insertion at front** – Inserts a new node as the first node.
b) **Insertion at the end** – Inserts a new node at the end as the last node.
c) **Insertion before a node** – Given a node, inserts a new node before this node.
d) **Insertion after a node** – Given a node, inserts a new node after this node.

d) Inserting a node at the front

1. Create a new node N.
2. Assign NULL to the **prev** pointer of node N.
3. If the node N is the first node of the list then we make **front** and **tail** point to this node N.
4. Else we only make **front** point to this node N.



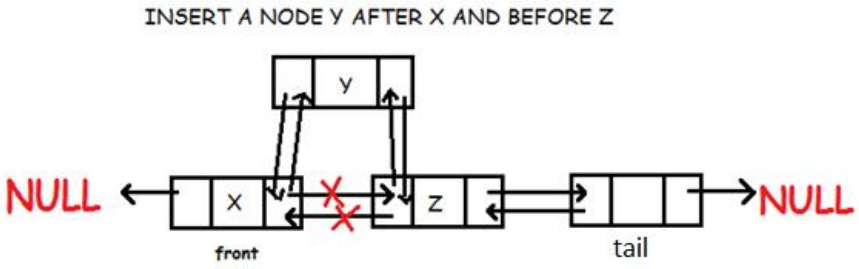
1. WE MAKE THE CURRENT FRONT NODE'S PREV POINT TO NEW NODE N
2. MAKE NEXT OF NEW NODE POINT TO CURRENT FRONT AND PREV OF NEW NODE POINT TO NULL.
3. WE CHANGE FRONT TO POINT TO THE NEW NODE.

The add_front() operation:

```
void Doubly_Linked_List :: add_front(int d)
{
	// Creating new node
	node *temp;
	temp = new node();
	temp->data = d;
	temp->prev = NULL;
	temp->next = front;

	// List is empty
	if(front == NULL)
		tail = temp;

	else
		front->prev = temp;

	front = temp;
}
```

e) Inserting a node before a node

Let's say we are inserting node X before Y. Then X's next pointer will point to Y and X's prev pointer will point the node Y's prev pointer is pointing. And Y's prev pointer will now point to X. We need to make sure that if Y is the first node of list then after adding X we make front point to X.

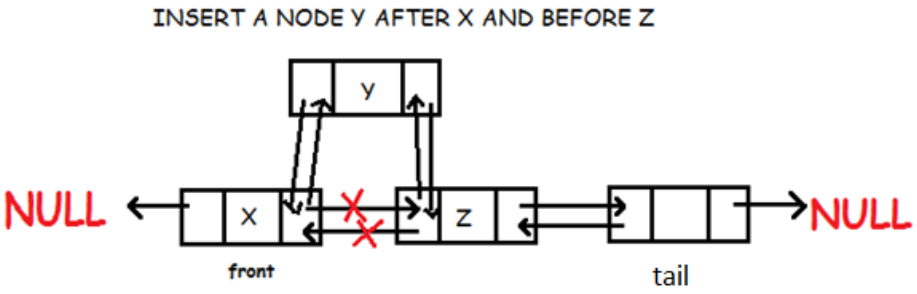


The add_before() operation:

```
void Doubly_Linked_List :: add_before(node *n, int d)
{
      node *temp;
      temp = new node();
      temp->data = d;
      temp->next = n;
      temp->prev = n->prev;
      n->prev = temp;

      //if node is to be inserted before first node
      if(n->prev == NULL)
            front = temp;
}
```

f) Insert a node after a Node

Let's say we are inserting node Y after X. Then Y's prev pointer will point to X and Y's next pointer will point the node X's next pointer is pointing. And X's next pointer will now point to Y. We need to make sure that if X is the last node of list then after adding Y we make end point to Y.



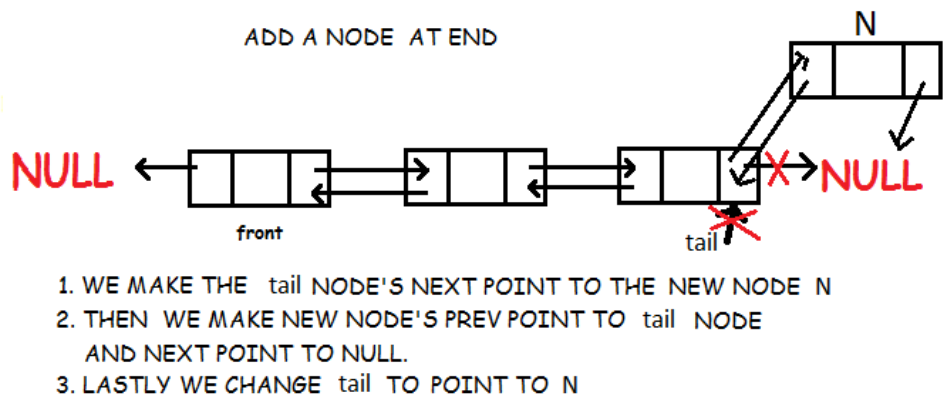
1. WE MAKE Y'S NEXT NODE POINT TO Z AND PREV NODE POINT TO X.
2. THEN MAKE X'S NEXT NODE POINT TO Y AND Z'S PREV NODE POINT TO Y.

The add_after() operation:

```
void Doubly_Linked_List :: add_after(node *n, int d)
{
      node *temp;
      temp = new node();
      temp->data = d;
      temp->prev = n;
      temp->next = n->next;
      n->next = temp;

      //if node is to be inserted after last node
      if(n->next == NULL)
            tail = temp;
}
```

g) Insert a node at the end

1. Create a new node named “N”.
2. Assign NULL value to the **next** pointer of node N.
3. Assign node tail to the **prev** of node N.
4. If the node N is the first node of the list then we make front and tail point to this node N.
5. Else we only make tail point to node N.



The add_end() operation:

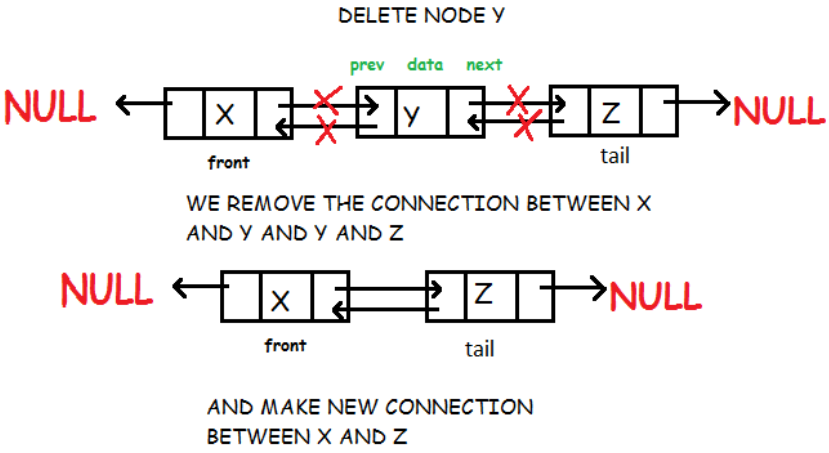
```
void Doubly_Linked_List :: add_end(int d)
{
	// create new node
	node *temp;
	temp = new node();
	temp->data = d;
	temp->prev = tail;
	temp->next = NULL;

	// if list is empty
	if(end == NULL)
		front = temp;
	else
		tail->next = temp;
	tail = temp;
}
```

5. Deletion

Deletion operation deletes a node from a given position in the doubly linked list.

1. **Deletion of the first node** – Deletes the first node in the list
2. **Deletion of the last node** – Deletes the last node in the list.
3. **Deletion of a node given the data** – Given the data, the operation matches the data with the node data in the linked list and deletes that node.

Removal of a node is quite easy in Doubly linked list but requires special handling if the node to be deleted is first or last element of the list. Unlike singly linked list where we require the previous node, here only the node to be deleted is needed. We simply make the next of the previous node point to next of current node (node to be deleted) and prev of next node point to prev of current node.



The delete_node() operation:

```
void Doubly_Linked_List :: delete_node(node *n)
{
	// if node to be deleted is first node of list
	if(n->prev == NULL)
	{
		front = n->next; //the next node will be front of list
		front->prev = NULL;
	}
	// if node to be deleted is last node of list
	else if(n->next == NULL)
	{
		tail = n->prev;   // the previous node will be last of list
		tail->next = NULL;
	}
	else
	{
		//previous node's next will point to current node's next
		n->prev->next = n->next;
		//next node's prev will point to current node's prev
		n->next->prev = n->prev;
	}
	//delete node
	delete(n);
}
```

6. Traversal

Traversal is a technique of visiting each node in the linked list. In a doubly linked list, we have two types of traversals as we have two pointers with different directions in the doubly linked list.

a) **Forward traversal** – Traversal is done using the next pointer which is in the forward direction.
b) **Backward traversal** – Traversal is done using the previous pointer which is the backward direction.

a) Forward Traversal

Start with the front node and visit all the nodes until the node becomes NULL.

```
void Doubly_Linked_List :: forward_traverse()
{
	node *trav;
	trav = front;
	while(trav != NULL)
	{
		cout<<trav->data<<endl;
		trav = trav->next;
	}
}
```

b) Backward Traversal

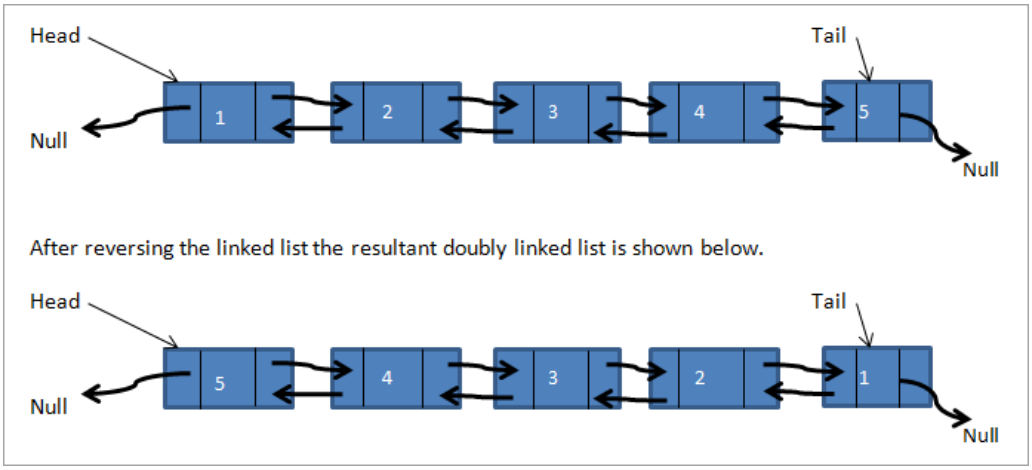
Start with the end node and visit all the nodes until the node becomes NULL.

```
void Doubly_Linked_List :: backward_traverse()
{
	node *trav;
	trav = tail;
	while(trav != NULL)
	{
		cout<<trav->data<<endl;
		trav = trav->prev;
	}
}
```

7. Reverse

This operation reverses the nodes in the doubly linked list so that the first node becomes the last node while the last node becomes the first node.

Reversing a doubly linked list is an important operation. In this, we simply swap the previous and next pointers of all the nodes and also swap the head and tail pointers.



(Practice for student: write a reverse () function to reverse the nodes in doubly linked list)

8. Search

Search operation in the doubly linked list is used to search for a particular node in the linked list. For this purpose, we need to traverse the list until a matching data is found.

(Practice for student: write a search () function to search an item in doubly linked list)

4.7. Circular linked list

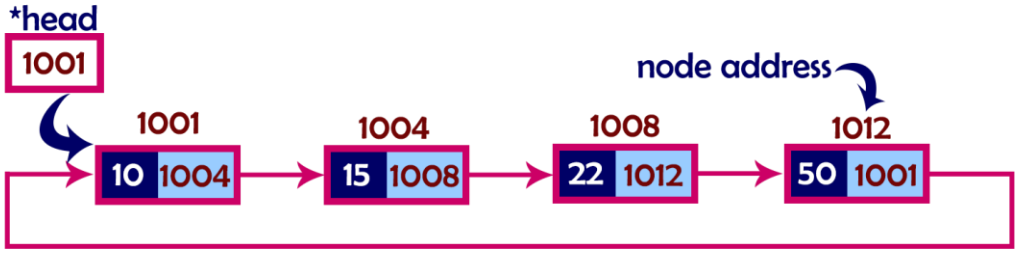
In single linked list, every node points to its next node in the list except the last node points NULL. But in circular linked list, every node points to its next node in the list but the last node points to the first node in the list. There are two types of circular linked list:

a) Circular singly linked list
b) Circular doubly linked list

Circular singly linked list

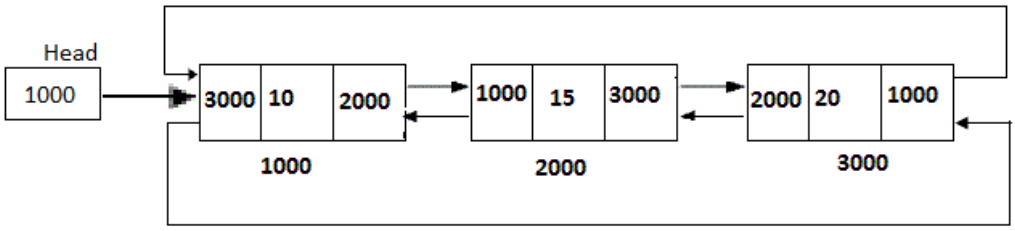
In a circular singly linked list, the last node of the list contains a pointer to the first node of the list. That means circular linked list is similar to the single linked list except that the last node points to the first node in the list.

We traverse a circular singly linked list until we reach the same node where we started. The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.



Circular doubly linked list

A circular doubly linked list or a circular two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence. The circular doubly linked list does not contain NULL in the previous field of the first node and the next field of the last node. Rather, the next field of the last node stores the address of the first node of the list, i.e., Head. Similarly, the previous field of the first field stores the address of the last node. A circular doubly linked list is one which has both the successor pointer and predecessor pointer in circular manner.



The advantage is that we can traverse back and forth using the next and pre pointers. As the last element is linked with the head so we can easily jump from the tail to the head of the Data structure. The disadvantage is that it needs two pointers for each node which makes it occupy extra space.

Application of Circular Linked List

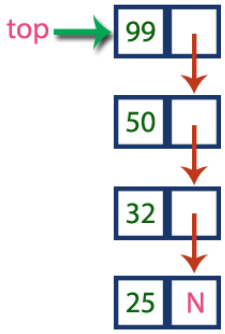
- The real life application where the circular linked list is used is our Personal Computers, where multiple applications are running. All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.
- Another example can be Multiplayer games. All the Players are kept in a Circular Linked List and the pointer keeps on moving forward ass a player's chance ends.
- Circular Linked List can also be used to create Circular Queue. In a Queue we have to keep two pointers, FRONT and REAR in memory all the time, where as in Circular Linked List, only one pointer is required.
- Browser surfing where a record of pages visited in the past by the user, is maintained in the form of circular linked lists and can be accessed again on clicking the previous button.

*(**Student practice:** implement the circular singly linked list and circular doubly linked list using C++)*

4.8. Linked Implementation of stacks

The major problem with the stack implemented using an array is, it works only for a fixed number of data values. That means the amount of data must be specified at the beginning of the implementation itself. Stack implemented using an array is not suitable, when we don't know the size of data which we are going to use. A stack data structure can be implemented by using a linked list data structure. The stack implemented using linked list can work for an unlimited number of values. That means, stack implemented using linked list works for the variable size of data. So, there is no need to fix the size at the beginning of the implementation. The Stack implemented using linked list can organize as many data values as we want.

In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'. Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list. The **next** field of the first element must be always **NULL**.



In the above example, the last inserted node is 99 and the first inserted node is 25. The order of elements inserted is 25, 32,50 and 99.

Push operation

We can use the following steps to insert a new node into the stack...

Step 1 - Create a **newNode** with given value.
Step 2 - Check whether stack is **Empty** (**top** == **NULL**)
Step 3 - If it is **Empty**, then set **newNode → next** = **NULL**.
Step 4 - If it is **Not Empty**, then set **newNode → next** = **top**.
Step 5 - Finally, set **top** = **newNode**.

Pop operation

We can use the following steps to delete a node from the stack...

Step 1 - Check whether **stack** is **Empty** (**top == NULL**).
Step 2 - If it is **Empty**, then display **"Stack is Empty"** and terminate the function
Step 3 - If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.
Step 4 - Then set '**top** = **top → next**'.
Step 5 - Finally, delete '**temp**'.

Linked list implementation of stack in C

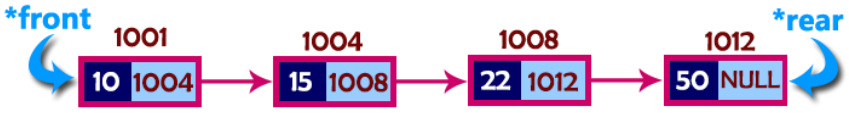
```
#include<stdio.h>
#include<conio.h>

struct Node
{
   int data;
   struct Node *next;
}*top = NULL;

void push(int);
void pop();
void display();

void main()
```

```
{
   int choice, value;
   clrscr();
   printf("\n:: Stack using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Push\n2. Pop\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d", &value);
                 push(value);
                 break;
         case 2: pop(); break;
         case 3: display(); break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void push(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   if(top == NULL)
      newNode->next = NULL;
   else
      newNode->next = top;
   top = newNode;
   printf("\nInsertion is Success!!!\n");
}
void pop()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      printf("\nDeleted element: %d", temp->data);
      top = temp->next;
      free(temp);
   }
}
void display()
{
   if(top == NULL)
      printf("\nStack is Empty!!!\n");
   else{
      struct Node *temp = top;
      while(temp->next != NULL){
         printf("%d--->",temp->data);
         temp = temp -> next;
      }
      printf("%d--->NULL",temp->data);
   }
}
```

4.9. Linked Implementation of Queues

The major problem with the queue implemented using an array is, it will work for an only fixed number of data values. That means, the amount of data must be specified at the beginning itself. Queue using an array is not suitable when we don't know the size of data which we are going to use. A queue data structure can be implemented using a linked list data structure. The queue which is implemented using a linked list can work for an unlimited number of values. That means, queue using linked list can work for the variable size of data (No need to fix the size at the beginning of the implementation). The Queue implemented using linked list can organize as many data values as we want.

In linked list implementation of a queue, the last inserted node is always pointed by '**rear**' and the first node is always pointed by '**front**'.



4.12.1.Enqueue operation

We can use the following steps to insert a new node into the queue...

Step 1 - Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
Step 2 - Check whether queue is **Empty** (**rear == NULL**)
Step 3 - If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.
Step 4 - If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

Dequeue operation

We can use the following steps to delete a node from the queue...

Step 1 - Check whether **queue** is **Empty** (**front == NULL**).
Step 2 - If it is **Empty**, then display "**Queue is Empty!!! Deletion is not possible!!!**" and terminate from the function
Step 3 - If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
Step 4 - Then set '**front = front → next**' and delete '**temp**' (**free(temp)**).

Linked list implementation of queue in C

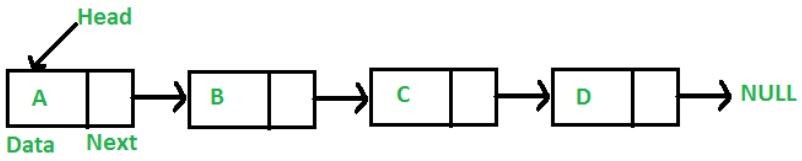
```
#include<stdio.h>
#include<conio.h>

struct Node
{
   int data;
   struct Node *next;
}*front = NULL,*rear = NULL;

void insert(int);
```

```
void delete();
void display();

void main()
{
   int choice, value;
   clrscr();
   printf("\n:: Queue Implementation using Linked List ::\n");
   while(1){
      printf("\n****** MENU ******\n");
      printf("1. Insert\n2. Delete\n3. Display\n4. Exit\n");
      printf("Enter your choice: ");
      scanf("%d",&choice);
      switch(choice){
         case 1: printf("Enter the value to be insert: ");
                 scanf("%d", &value);
                 insert(value);
                 break;
         case 2: delete(); break;
         case 3: display(); break;
         case 4: exit(0);
         default: printf("\nWrong selection!!! Please try again!!!\n");
      }
   }
}
void insert(int value)
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode->data = value;
   newNode -> next = NULL;
   if(front == NULL)
      front = rear = newNode;
   else{
      rear -> next = newNode;
      rear = newNode;
   }
   printf("\nInsertion is Success!!!\n");
}
void delete()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      front = front -> next;
      printf("\nDeleted element: %d\n", temp->data);
      free(temp);
   }
}
void display()
{
   if(front == NULL)
      printf("\nQueue is Empty!!!\n");
   else{
      struct Node *temp = front;
      while(temp->next != NULL){
```

```
            printf("%d--->",temp->data);
            temp = temp -> next;
        }
        printf("%d--->NULL\n",temp->data);
    }
}
```

4.10. Applications of linked list data structure

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



Applications of linked list in computer science

- Implementation of stacks and queues
- Implementation of graphs: Adjacency list representation of graphs is most popular which uses linked list to store adjacent vertices.
- Dynamic memory allocation: We use linked list of free blocks.
- Maintaining directory of names
- Performing arithmetic operations on long integers
- Manipulation of polynomials by storing constants in the node of linked list
- representing sparse matrices
- *Image viewer* – Previous and next images are linked, hence can be accessed by next and previous button.
- *Previous and next page in web browser* – We can access previous and next url searched in web browser by pressing back and next button since, they are linked as linked list.
- *Music Player* – Songs in music player are linked to previous and next song. you can play songs either from starting or ending of the list.
- 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.
- 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.
- Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

4.11. 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:

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

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).**

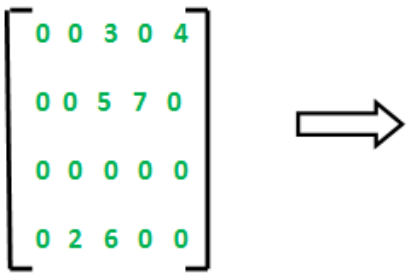
Sparse Matrix Representations can be done in many ways following are two common representations:

1. Array representation
2. Linked list representation

4.12.1. Sparse Matrix using Array Representation

Two dimensional array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



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

C++ program for Sparse Matrix Representation using array

```
#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++;

    // number of columns in compactMatrix (size) must be
    // equal to number of non - zero elements in
    // sparseMatrix
    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;
}
```

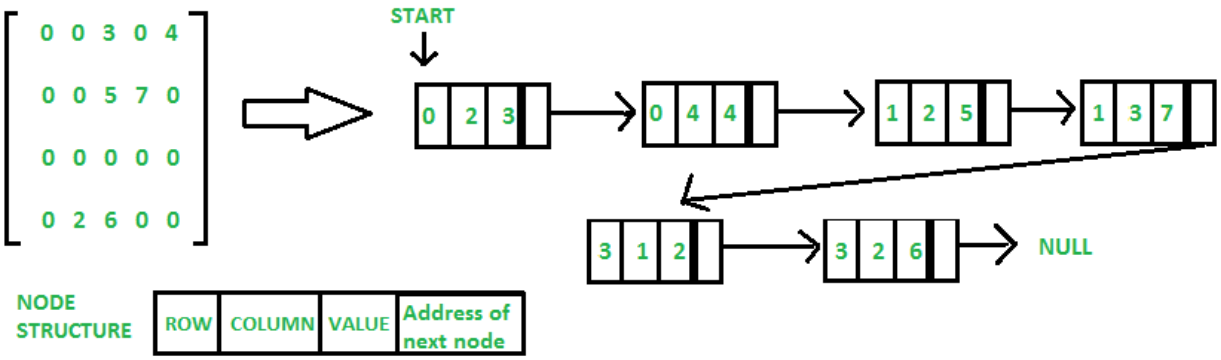
Output:

```
0 0 1 1 3 3

2 4 2 3 1 2

3 4 5 7 2 6
```

Sparse Matrix using Linked List Representation

In linked list, each node has four fields. These four fields are defined as:

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)
- **Next node:** Address of the next node



C program for Sparse Matrix Representation using Linked Lists

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

// Node to represent sparse matrix
struct Node
{
    int value;
    int row_position;
    int column_postion;
    struct Node *next;
};

// Function to create new node
void create_new_node(struct Node** start, int non_zero_element,
                     int row_index, int column_index )
{
    struct Node *temp, *r;
    temp = *start;
    if (temp == NULL)
    {
        // Create new node dynamically
        temp = (struct Node *) malloc (sizeof(struct Node));
        temp->value = non_zero_element;
        temp->row_position = row_index;
        temp->column_postion = column_index;
        temp->next = NULL;
        *start = temp;

    }
    else
    {
        while (temp->next != NULL)
            temp = temp->next;

        // Create new node dynamically
        r = (struct Node *) malloc (sizeof(struct Node));
        r->value = non_zero_element;
        r->row_position = row_index;
        r->column_postion = column_index;
        r->next = NULL;
        temp->next = r;
    }
}

// This function prints contents of linked list
// starting from start
void PrintList(struct Node* start)
{
    struct Node *temp, *r, *s;
    temp = r = s = start;

    printf("row_position: ");
    while(temp != NULL)
    {
```

```
        printf("%d ", temp->row_position);
        temp = temp->next;
    }
    printf("\n");

    printf("column_postion: ");
    while(r != NULL)
    {
        printf("%d ", r->column_postion);
        r = r->next;
    }
    printf("\n");
    printf("Value: ");
    while(s != NULL)
    {
        printf("%d ", s->value);
        s = s->next;
    }
    printf("\n");
}

int main()
{
   // Assume 4x5 sparse matrix
    int sparseMatric[4][5] =
    {
        {0 , 0 , 3 , 0 , 4 },
        {0 , 0 , 5 , 7 , 0 },
        {0 , 0 , 0 , 0 , 0 },
        {0 , 2 , 6 , 0 , 0 }
    };
    /* Start with the empty list */
    struct Node* start = NULL;

    for (int i = 0; i < 4; i++)
        for (int j = 0; j < 5; j++)

            // Pass only those values which are non - zero
            if (sparseMatric[i][j] != 0)
                create_new_node(&start, sparseMatric[i][j], i, j);
    PrintList(start);
    return 0;
}
```

Output:

```
row_position:    0 0 1 1 3 3

column_postion:  2 4 2 3 1 2

Value:           3 4 5 7 2 6
```

4.12. Polynomial expressions and linked list

Polynomials and Sparse Matrix are two important applications of arrays and linked lists. Polynomials can be represented using arrays and linked lists.

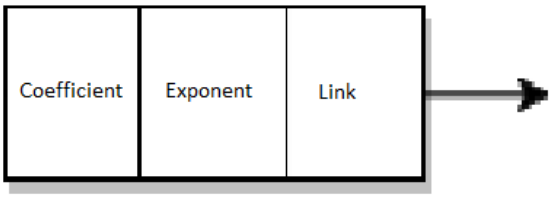
4.12.1. Linked list representation of polynomials

A polynomial is composed of different terms where each of them holds a coefficient and an exponent. 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.

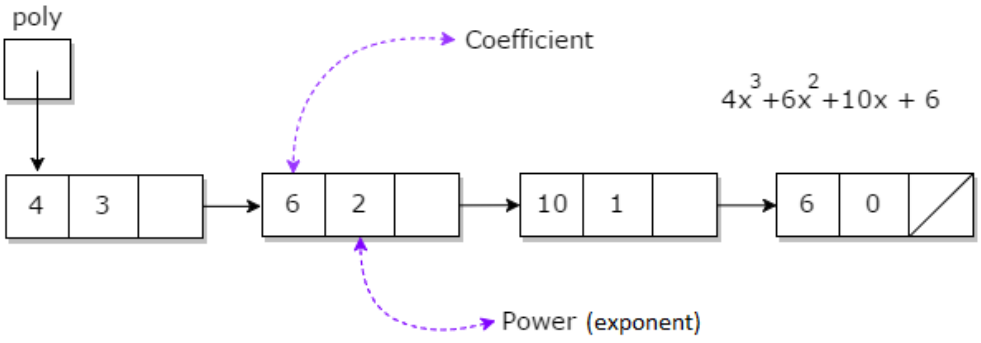
An essential characteristic of the polynomial is that each term in the polynomial expression consists of two parts:

- the coefficient
- the exponent

Therefore, to represent each term of polynomial requires two fields to store coefficient and exponent. To represent a whole polynomial using a linked list, each node requires third field which contains the address of the term's node which points to the next term of the polynomial. Therefore, a node needs 3 fields to represent a term of polynomial. Each node represent a term. The terms in polynomial are linked to form a linked list.



Example: Let us consider a polynomial- $4x^3 + 6x^2 + 10x + 6$, where 4, 6, 10 and 6 are coefficients and 3, 2, 1 and 0 are exponential value of their corresponding terms respectively. This polynomial can be represented using a linked list as shown in below figure.



Addition of two polynomials

Adding two polynomials that are represented by a linked list. We check values at the exponent value of the node. For the same values of exponent, we will add the coefficients.

Example:

```
Input:
p1= 13x^8 + 7x^5 + 32x^2 + 54
p2= 3x^12 + 17x^5 + 3x^3 + 98

Output: 3x^12 + 13x^8 + 24x^5 + 3x3 + 32x^2 + 152
```

Algorithm

Input – polynomial p1 and p2 represented as a linked list.

1. Loop around all values of linked list and follow step 2& 3.
2. If the value of a node's exponent is greater copy this node to result node and head towards the next node.
3. If the values of both node's exponent is same add the coefficients and then copy the added value with node to the result.
4. Print the resultant node.

Implementing of addition of two polynomials using C

```c
#include<bits/stdc++.h>
using namespace std;
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 *p1, struct Node *p2, struct Node *result){
   while(p1->next && p2->next){
      if(p1->pow > p2->pow){
         result->pow = p1->pow;
         result->coeff = p1->coeff;
         p1 = p1->next;
      }
      else if(p1->pow < p2->pow){
         result->pow = p2->pow;
         result->coeff = p2->coeff;
         p2 = p2->next;
      } else {
         result->pow = p1->pow;
         result->coeff = p1->coeff+p2->coeff;
         p1 = p1->next;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
      result->next = NULL;
   }
   while(p1->next || p2->next){
      if(p1->next){
         result->pow = p1->pow;
         result->coeff = p1->coeff;
         p1 = p1->next;
      }
      if(p2->next){
         result->pow = p2->pow;
         result->coeff = p2->coeff;
         p2 = p2->next;
      }
      result->next = (struct Node *)malloc(sizeof(struct Node));
      result = result->next;
      result->next = NULL;
   }
}
void printpoly(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 *p1 = NULL, *p2 = NULL, *result = NULL;
   create_node(41,7,&p1);
   create_node(12,5,&p1);
   create_node(65,0,&p1);
   create_node(21,5,&p2);
   create_node(15,2,&p2);
   printf("polynomial 1: ");
   printpoly(p1);
   printf("\npolynomial 2: ");
```

```
    printpoly(p2);
    result = (struct Node *)malloc(sizeof(struct Node));
    polyadd(p1, p2, result);
    printf("\npolynomial after adding p1 and p2 : ");
    printpoly(result);
    return 0;
}
```

Output:

```
polynomial 1: 41x^7 + 12x^5 + 65x^0
polynomial 2: 21x^5 + 15x^2
polynomial after adding p1 and p2 : 41x^7 + 33x^5 + 15x^2 + 65x^0
```