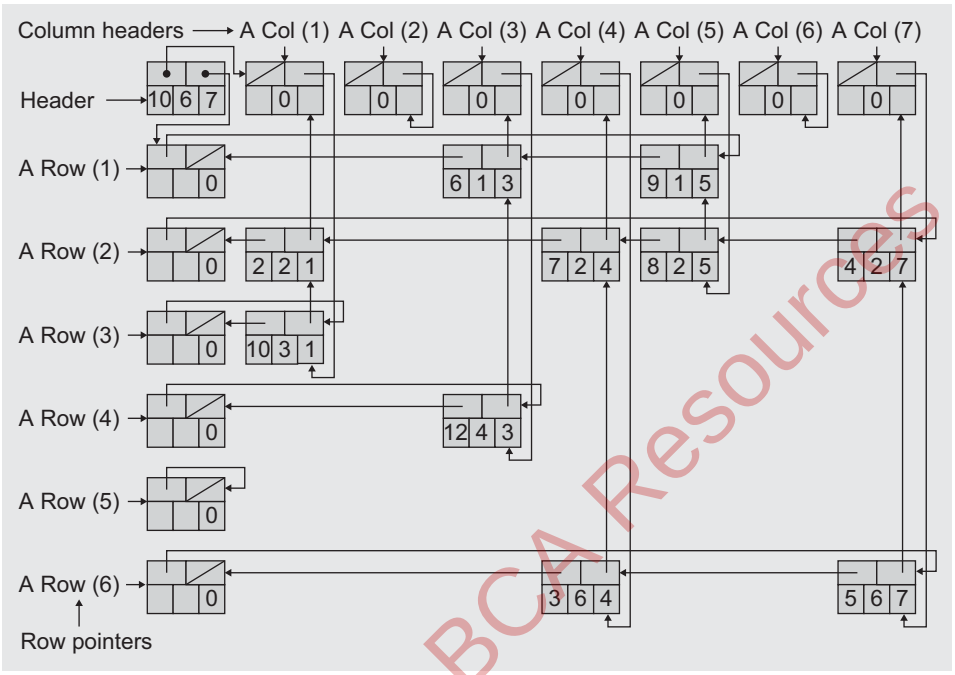

Fig. 6.48 Multilinked sparse matrix

6.11 LINKED STACK

In Chapter 3, we have implemented stacks using arrays. However, an array implementation has certain limitations. One of the limitations is that such a stack cannot grow or shrink dynamically. This drawback can be overcome by using linked implementation. We have studied linked list implementation of a linear list. Let us study the same linked list with restriction on addition and deletion of a node to use it as a stack. A stack implemented using a linked list is also called *linked stack*.

Each element of the stack will be represented as a node of the list. The addition and deletion of a node will be only at one end. The first node is considered to be at the top of the stack, and it will be pointed to by a pointer called `top`. The last node is the bottom of the stack, and its link field is set to `Null`. An empty stack will have `Top = Null`. A linked stack with elements (*X*, *Y*, Z) in order (*X* on top) may be represented as in Fig. 6.49.

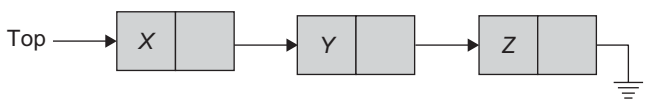

Fig. 6.49 Linked stack of elements (*X*, *Y*, Z)

Figure 6.49 shows a pictorial representation of the stack *S* containing three elements (*X*, *Y*, Z). Here, `top` is a pointer pointing to the top element of the stack. *X* is at the top of the stack and *Z* is at the bottom of the stack. SLL is suitable to implement stack using linked organization as we operate at one end of the list only.

6.11.1 Class for Linked Stack

The node of the list structure is defined in Program Code 6.17.

PROGRAM CODE 6.17

```
class Stack_Node
{
   public:
      int data;
      Stack_Node *link;
};

class Stack
{
   private:
      Stack_Node *Top;
      int Size;
      int IsEmpty();
   public:
      Stack()
      {
         Top = Null;
         Size = 0;
      }
      int GetTop();
      int Pop();
      void Push( int Element);
      int CurrSize();
};
```


Here, the stack can have any data type such as `int`, `char`, `float`, `struct`, and so on for the data field. The link field is a pointer pointing to the node below (next to) it. The `Top` serves the purpose of the variable associated with the data structure `stack` here. Similar to array implementation, an empty stack can be created by initializing the `Top`. This is going to hold the address of a node. It is a pointer rather than an integer as in contiguous

stack. Hence to represent an empty stack, `Top` is initialized to `Null`. Every `insert` and `delete` of a node will be only at the end pointed by the pointer variable `Top`. Figure 6.50 represents the insertion of data in a linked stack considering the following sequence of instruction:

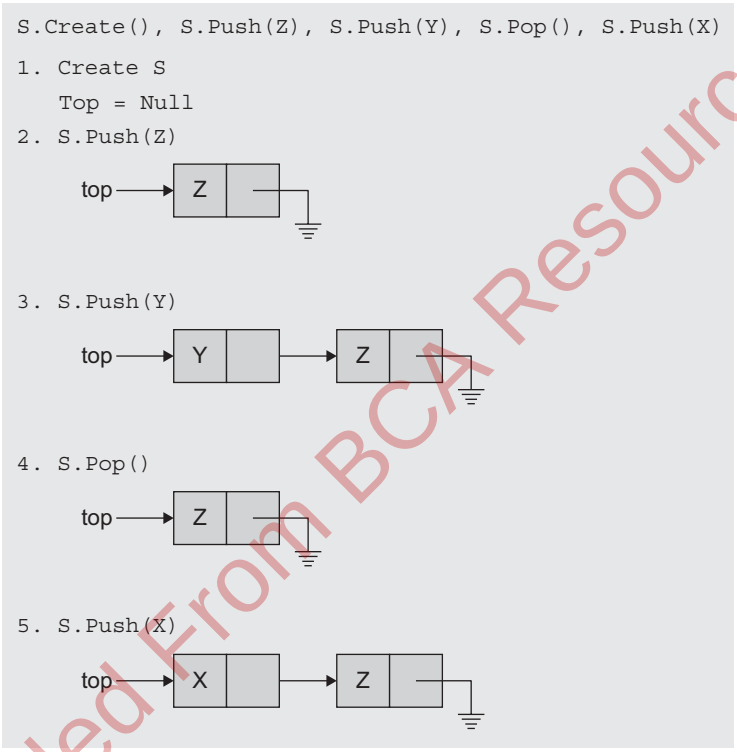

Fig. 6.50 Insertion of data in linked stack

Here, the stack grows and also shrinks at `Top`. Let us see the functions required to implement a stack using a linked list.

6.11.2 Operations on Linked Stack

The memory for each node is dynamically allocated on the heap. So when an item is pushed, a node for it is created, and when an item is popped, its node is freed (using `delete`). The only difference is that the capacity of a linked stack is generally greater than that of a contiguous stack since a linked stack will not become full until the dynamic memory is exhausted Program Code 6.18 shows operations on a linked stack. Figure 6.51 shows a logical view of the linked stack.

PROGRAM CODE 6.18

```
class Stack_Node
{
   public:
      int data;
      Stack_Node *link;
};

class Stack
{
   private:
      Stack_Node *Top;
      int Size;
      int IsEmpty();
   public:
      Stack()
      {
         Top = Null;
         Size = 0;
      }
      int GetTop();
      int Pop();
      void Push(int Element);
      int CurrSize();
};

int Stack :: IsEmpty()
{
   if(Top == Null)
      return 1;
   else
      return 0;
}

int Stack :: GetTop()
{
   if(!IsEmpty())
      return(Top->data);
}
```

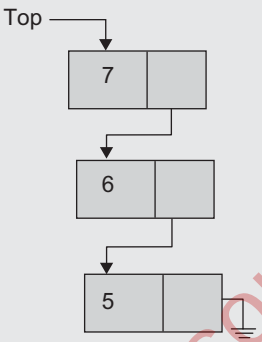

Fig. 6.51 Logical view of a linked stack

```
void Stack :: Push(int value)
{
   Stack_Node* NewNode;
   NewNode = new Stack_Node;
   NewNode->data = value;
   NewNode->link = Null;
   NewNode->link = Top;
   Top = NewNode;
}

int Stack :: Pop()
{
   Stack_Node* tmp = Top;
   int data = Top->data;
   if(!IsEmpty())
   {
      Top = Top->link;
      delete tmp;
      return(data);
   }
}
```

We have designed the functions for operations on stack, where the stack is implemented using linked organization. The `Top` is initialized to `Null` to indicate empty stack. The `Push()` function dynamically creates a new node. After creating a new node, the pointer variable `Top` should point to the newly added node in the stack.

```
void main()
{
   Stack S;
   S.Push(5);
   S.Push(6);
   cout << S.GetTop()<<endl;
   cout << S.Pop()<<endl;
   S.Push(7);
   cout << S.Pop()<<endl;
   cout << S.Pop()<<endl;
}

Output
   6
   6
   7
   5
```

6.12 LINKED QUEUE

We studied about how to represent queues using sequential organization in Chapter 5. Such a representation is efficient if we have a circular queue of fixed size. However, there are many drawbacks of implementing queues using arrays. The fixed sizes do not give flexibility to the user to dynamically exceed the maximum size. The declaration of arbitrarily maximum size leads to poor utilization of memory. In addition, the major drawback is the updating of `front` and `rear`. For correctness of the said implementation, the shifting of the queue to the left is necessary and to be done frequently. Here is a good solution to this problem which uses linked list. We need two pointers, `front` and `rear`. Figure 6.52 shows a linked queue which is easy to handle.

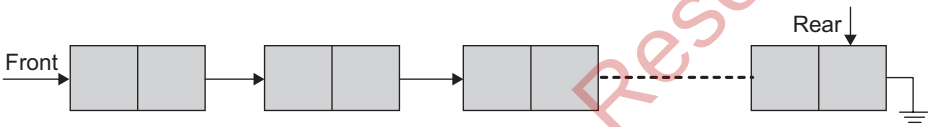

Fig. 6.52 The linked queue

Notice that the direction link for nodes is to facilitate easy insertion and deletion of nodes. One can easily add a node at the `rear` and delete a node from the `front`.

One of the node structures could be as in Program Code 6.19.

PROGRAM CODE 6.19

```
class Student
{
   public:
      int Roll_No;
      char Name[30];
      int Year;
      char Branch[8];
      Student *link;
};
class Queue
{
   Student *front, *rear;
   public:
      Queue()
      {
         front = rear = Null;
      }
};
```

Let us consider the following node structure for studying the linked queue and operations:

```
class QNode
{
   public:
      int data;
      QNode *link;
};

class Queue
{
   QNode *Front, *Rear;
   int IsEmpty();
   public:
      Queue()
      {
         Front = Rear = Null;
      }
      void Add( int Element);
      int Delete();
      int FrontElement();
      ~Queue();
};

int Queue :: IsEmpty()
{
   if(Front == Null)
      return 1;
   else
      return 0;
}
```

The queue element is declared using the class `QNode`. Each node contains the data declaration and the link pointer to the next element in the queue. This declaration creates an empty queue and initializes the pointers `front` and `rear` to `Null`. Here, `front` always points to the first node of queue and `rear` points to the last node of queue.

`Queue empty` condition is simply checked by comparing the `front` with `Null`. The function `IsEmptyQ` returns 1 (i.e., true) if the queue is empty and returns 0 (i.e., false), otherwise.

```
int Queue :: IsEmpty()
{
   if(Front == Null)
      return 1;
   else
      return 0;
}
```

`FrontElement()` returns the data element at the front of the queue. Here, the `front` is not updated. `FrontElement()` just reads what is at front.

```
int Queue :: GetFront()
{
   if(!IsEmpty())
      return(Front->data);
}
```

Note that if the `NewNode` is a node getting added in an empty queue, then along with the `rear`, the `front` should also be set to point to the newly added node, which is at the front of the queue. Hence, as both the `front` and the `rear` may get updated. Program Code 6.20 shows the addition of an element to a linked queue.

PROGRAM CODE 6.20

```
void Queue :: Add(int x)
{
   QNode *NewNode;
   NewNode = new QNode;
   NewNode->data = x;
   NewNode->link = Null;
   // if the new is a node getting added in empty queue
   //then front should be set so as to point to new
   if(Rear == Null)
   {
      Front = NewNode;
      Rear = NewNode;
   }
   else
   {
      Rear->link = NewNode;
      Rear = NewNode;
   }
}
```

`Delete()` function first verifies if there is any data element in the queue. If there is an element, `Delete()` gets and returns the data at the front of the queue to the caller function. Then, the `front` is set to point to the new queue front node, which is next to the node being deleted. If the last node is being deleted, then the deleted node's next pointer is guaranteed to be `Null`. Note that if the current deletion of a node results in `queue empty` state, then along with the `front`, the `rear` should also be set to `Null`.

```
int Queue :: Delete()
{
```

```
    int temp;
    QNode *current = Null;
    if(!IsEmpty())
    {
        temp = Front->data;
        current = Front;
        Front = Front->link;
        delete current;
        if(Front == Null)
            Rear = Null;
        return(temp);
    }
}

int Queue :: FrontElement()
{
    if(!IsEmpty())
        return(Front->data);
}

void main()
{
    Queue Q;
    Q.Add(11);
    Q.Add(12);
    Q.Add(13);
    cout << Q.Delete() << endl;
    Q.Add(14);
    cout << Q.Delete() << endl;
    cout << Q.Delete() << endl;
    cout << Q.Delete() << endl;
    Q.Add(15);
    Q.Add(16);
    cout << Q.Delete() << endl;
    cout << Q.Delete() << endl;
}
Output
11
12      // due to FrontElement
12
13
14
15
16
```


6.12.1 Erasing a Linked Queue

The following function in Program Code 6.21 traverses through the whole queue and also releases the memory allocated for each node. This task is handled by a destructor.

PROGRAM CODE 6.21

```
void Queue :: ~Queue()
{
   QNode *temp;
   while(Front! = Null)
   {
      temp = Front;
      Front = Front->link;
      delete temp;
   }
   Front = Rear = Null;
}
```

The linked queue may have the first node on a queue as a header node where the data field may hold some relevant information. In such a list, the first node, that is, the header node, is ignored (i.e. skipped) during `Delete()` operation. Similarly, the `Add()` function and `queue empty` condition will be changed accordingly.

6.13 GENERALIZED LINKED LIST

We have defined and represented linear list, which contains series of data elements, all of which had the same data type. In this topic, we shall extend the notion of list even further. We shall study generalized lists, which may be a list of lists.

Generalized lists are defined recursively as lists whose members may be single data elements or other generalized lists. Generalized lists are the most flexible and useful structures. We can use such lists to represent virtually all of the data structures. In addition, generalized lists provide the key data structure for several programming languages, such as LISP. Other languages, such as T and Miranda, include generalized lists and their operations as built-in capabilities. This widespread inclusion of generalized lists in many languages and environments attests the value of such lists in many applications.

6.13.1 Definition

A generalized list is a linear list (non-indexed) of zero or more data elements or generalized lists. In other words, a generalized list is a finite sequence of $n \geq 0$ elements, α_1, α_2, ... α_n, which we write as list $A = (\alpha_1, \alpha_2, ..., \alpha_n)$, where a_i is either an atom or the list. The elements of α_i, where $1 \leq i \leq n$, which are not atoms are said to be the sub-lists of the list. Here A is the name of generalized list and n is its length.

Thus, a generalized list may be made up of a number of components, some of which are data elements (atoms) and others are generalized lists.