Data structure

Data structure is the structural representation of logical relationships between elements of data. In other words a data structure is a way of organizing data items by considering its relationship to each other.

Data structure mainly specifies the structured organization of data, by providing accessing methods with correct degree of associativity. Data structure affects the design of both the structural and functional aspects of a program.

Program = Algorithm + Data Structure

Data structures are the building blocks of a program; here the selection of a particular data structure will help the programmer to design more efficient programs as the complexity and volume of the problems solved by the computer is steadily increasing day by day. The programmers have to strive hard to solve these problems. If the problem is analyzed and divided into sub problems, the task will be much easier *i.e.*, divide, conquer and combine.

A complex problem usually cannot be divided and programmed by set of modules unless its solution is structured or organized. This is because when we divide the big problems into sub problems, these sub problems will be programmed by different programmers or group of programmers. But all the programmers should follow a standard structural method so as to make easy and efficient integration of these modules. Such type of hierarchical structuring of program modules and sub modules should not only reduce the complexity and control the flow of program statements but also promote the proper structuring of information. By choosing a particular structure (or data structure) for the data items, certain data items become friends while others loses its relations.

Algorithm

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program.

Representation of algorithm can written By:-

In natural language (English) / pseudo-code / diagrams (Flow chart) / etc.

Pseudo- code:-

A mixture of natural language and high – level programming concepts that describes the main ideas behind a generic implementation of a data structure or algorithm**. Pseudo- code** is more structured than usual language but less formal than a programming language.

E.g.:- Algorithm arrayMax (A, n)

input: An array A sorting n integers

output: The Maximum element in A

```
currentMax ← A[0]
for i ← 1 to n-1 do
if currentMax < A[i] then currentMax ← A[i]
return currentMax
```

Ex: An algorithm to find sum n numbers between any given range numbers

```
1- Start
2- Read N
3- Sum =0
4- For I= 1 to N
      4.1 sum = sum + I
      4.1 next I
5- print Sum
6- End
```

What Makes a Good Algorithm?

Suppose you have two possible algorithms or data structures that basically do the same thing; which is better?

- Faster
- Less space
- Easier to code
- Easier to maintain

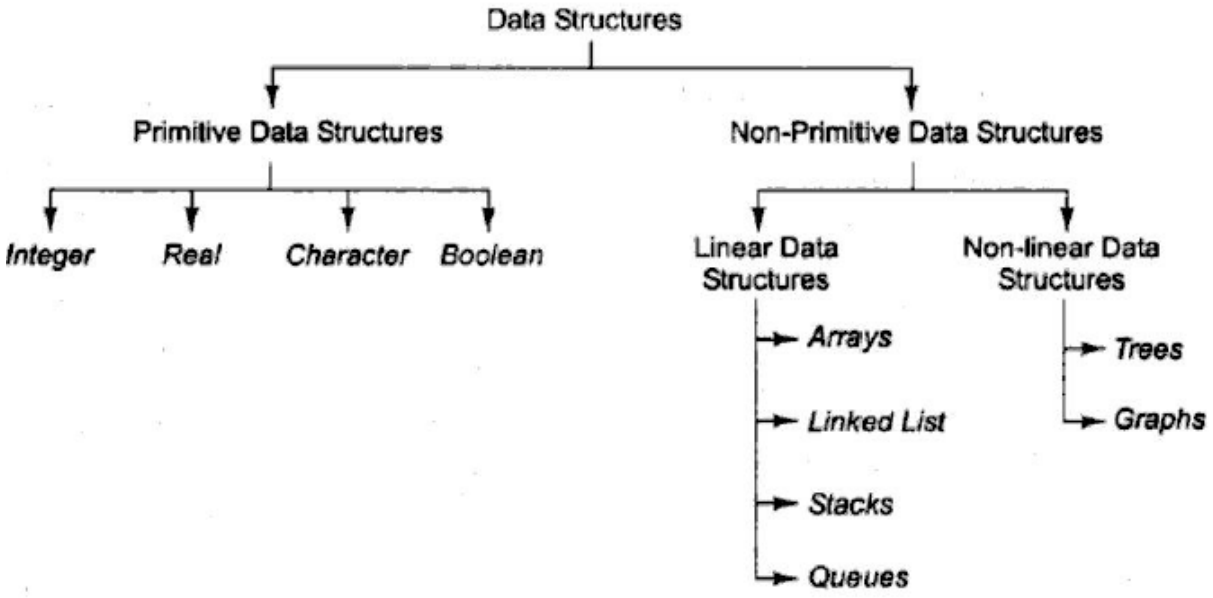


Fig.1 Classifications of data structures

Classification of data structure

Data structures are broadly divided into two:

1. Primitive data structures: These are the basic data structures and are directly operated upon by the machine instructions, which is in a primitive level. They are integers, floating point numbers, characters, string constants, pointers etc. These primitive data structures are the basis for the discussion of more sophisticated (non-primitive) data structures.

2. Non-primitive data structures: It is a more sophisticated data structure emphasizing on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Array, list, files, linked list, trees and graphs fall in this category.

How to choose the suitable data structure:-

For each set of data, there are different methods to organize these data in a particular data structure. To choose the suitable data structure, we must use the following criteria:-

1- Data size and the required memory.
2- The dynamic nature of the data.
3- The required time to obtain any data element from the data structure.
4- The programming approach and the algorithm that will be used to manipulate these data.

Assignment -1-

- **Write an algorithm for the following**
  a- count even& odd numbers in given range

LINKED LIST DATA STRUCTURE

A linked list is a linear collection of specially designed data elements, called nodes, linked to one another by means of pointers. Each node is divided into two parts: the first part contains the information of the element, and the second part contains the address of the next node in the linked list. Address part of the node is also called linked or next field. Following Fig 5:1 shows a typical example of node.

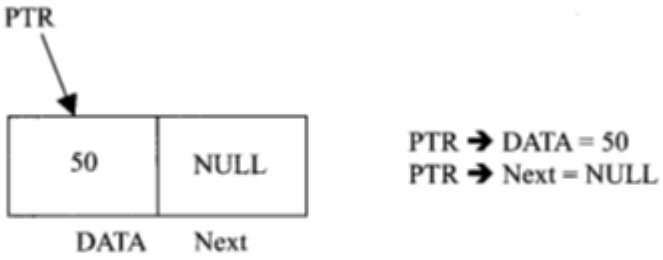


Fig. 5.1. Nodes.

Fig. 5.2. Linked List.

Fig.5.2.shows a schematic diagram of a linked list with 3 nodes. Each node is pictured with two parts. The left part of each node contains the data items and the right part represents the address of the next node; there is an arrow drawn from it to the next node. The next pointer of the last node contains a special value, called the NULL pointer, which does not point to any address of the node. That is NULL pointer indicates the end of the linked list. START pointer will hold the address of the 1st node in the list START = NULL if there is no list (*i.e.*; NULL list or empty list).

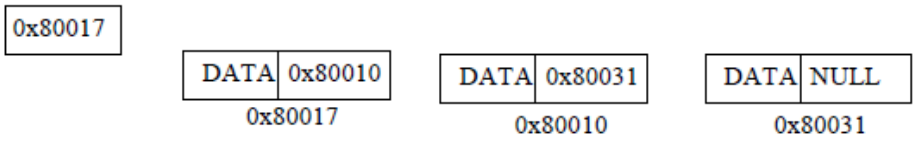


Fig. Linked List representation in memory.

Explanation:

Because each node of a linked list has two components, we need to declare each node as a class or struct. The data type of each node depends on the specific application—that is, what kind of data is being processed. However, the link component of each node is a pointer. The data type of this pointer variable is the node type itself. For the previous linked list, the definition of the node is as follows. (Suppose that the data type is int.)

```
struct nodename
      {
        int info;
        nodename *link;
       };
```

The variable declaration is as follows:
nodename *head;

Linked List: Some Properties

To better understand the concept of a linked list and a node, some important properties of linked lists are described next.

Consider the linked list in Figure 5-4.

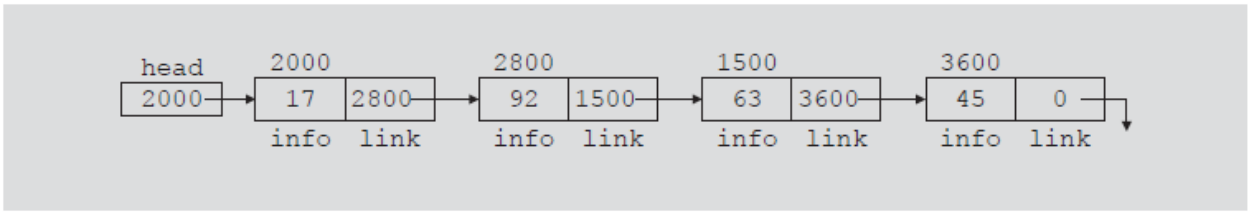


FIGURE 5-4 Linked list with four nodes

This linked list has four nodes. The address of the first node is stored in the pointer head.

Each node has two components: info, to store the info, and link, to store the address of the next node. For simplicity, we assume that info is of type int.

Suppose that the first node is at location 2000, the second node is at location 2800, the third node is at location 1500, and the fourth node is at location 3600. Table Table 5-1 shows the values of head and some other nodes in the list shown in Figure 5-4.

TABLE 5-1 Values of head and some of the nodes of the linked list in Figure 5-4

	Value	Explanation
head	2000	
head->info	17	Because head is 2000 and the info of the node at location 2000 is 17
head->link	2800	
head->link->info	92	Because head->link is 2800 and the info of the node at location 2800 is 92

Suppose that current is a pointer of the same type as the pointer head. Then the statement

current = head;

copies the value of head into current. Now consider the following statement:

current = current->link;

This statement copies the value of current->link, which is 2800, into current. Therefore, after this statement executes, current points to the second node in the list. (When working with linked lists, we typically use these types of statements to advance a pointer to the next node in the list.) See Figure 5-5.

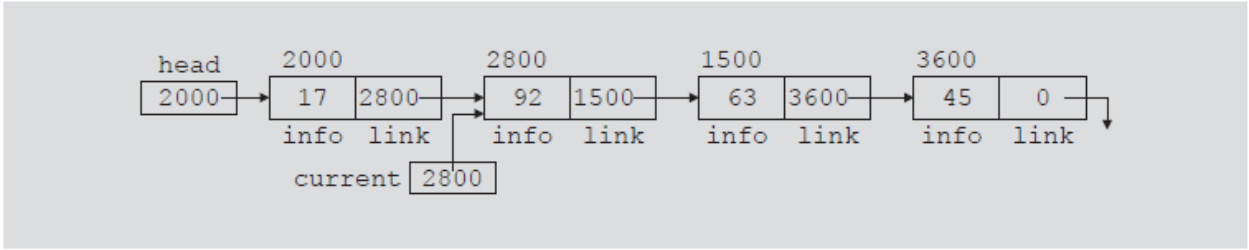


FIGURE 5-5 List after the statement current = current->link; executes

Table 5-2 shows the values of current, head, and some other nodes in Figure 5-5

TABLE 5-2 Values of current, head, and some of the nodes of the linked list in Figure 5-5

	Value
current	2800
current->info	92
current->link	1500
current->link->info	63
head->link->link	1500
head->link->link->info	63
head->link->link->link	3600
current->link->link->link	0 (that is, NULL)
current->link->link->link->info	Does not exist (run-time error)

TRAVERSING A LINKED LIST

The basic operations of a linked list are as follows: Search the list to determine whether a particular item is in the list, insert an item in the list, display the elements of the list, and delete an item from the list.

These operations require the list to be traversed. That is, given a pointer to the first node of the list, we must step through the nodes of the list.

Suppose that the pointer **head** points to the first node in the list, and the link of the last node is **NULL**. We cannot use the pointer **head** to traverse the list because if we use the **head** to traverse the list, we would lose the nodes of the list. This problem occurs because the links are in only one direction. The pointer **head** contains the address of the first node, the first node contains the address of the second node, the second node contains the address of the third node, and so on. If we move **head** to the second node, the first node is lost (unless we save a pointer to this node). If we keep advancing **head** to the next node, we will lose all the nodes of the list (unless we save a pointer to each node before advancing **head**, which is impractical because it would require additional computer time and memory space to maintain the list). Therefore, we always want **head** to point to the first node. It now follows that we must traverse the list using another pointer of the same type. Suppose that **current** is a pointer of the same type as **head**. The following code traverses the list:

```
current = head;
while (current != NULL)
{
        //Process current
        current = current->link;
}
```

For example, suppose that **head** points to a linked list of numbers. The following code outputs the data stored in each node:

```
current = head;
while (current != NULL)
{
         cout << current->info << " ";
         current = current->link;
}
```

LINKED LIST ALGORITHMS

This section discusses the algorithms of linked list data structures. Consider the following definition of a node. (For simplicity, we assume that the info type is **int**.)

```
struct nodename
{
    int info;
    nodename *link;
};
```

We will use the following variable declaration:

```
nodename *head, *p, *q, *newNode;
```

ALGORITHM FOR INSERTING A NODE

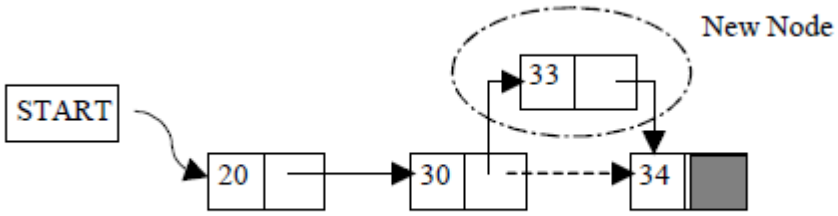


Fig. 5.14. Insertion of New Node

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the new node is to be inserted. TEMP is a temporary pointer to hold the node address.

Insert a Node at the beginning

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. If (SATRT equal to NULL)
   (a) NewNode -> Link = NULL
5. Else
   (a) NewNode -> Link = START
6. START = NewNode
7. Exit

Insert a Node at the end

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode -> DATA = DATA
4. NewNode -> Next = NULL
5. If (SATRT equal to NULL)
   (a) START = NewNode
6. Else
   (a) TEMP = START
   (b) While (TEMP -> Next not equal to NULL)
      (i) TEMP = TEMP -> Next
7. TEMP -> Next = NewNode
8. Exit

Insert a Node at any specified position

1. Input DATA and POS to be inserted
2. initialize TEMP = START; and k = 0
3. Repeat the step 3 while(k is less than POS)

(a) TEMP = TEMP -> Next
(b) If (TEMP is equal to NULL)
(i) Display "Node in the list less than the position"
(ii) Exit
(c) k = k + 1
4. Create a New Node
5. NewNode -> DATA = DATA
6. NewNode -> Next = TEMP -> Next
7. TEMP -> Next = NewNode
8. Exit

Consider the linked list shown in Figure 5-6.

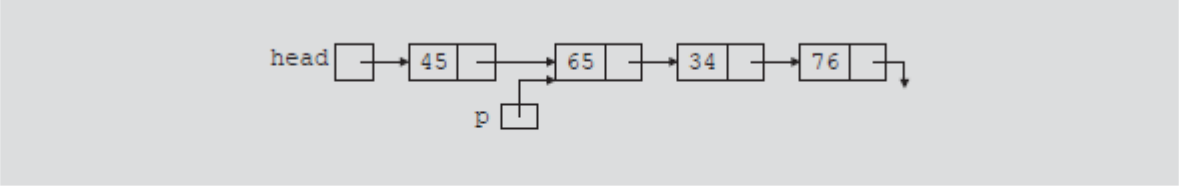


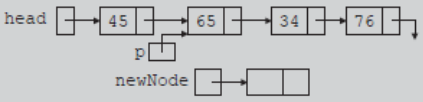
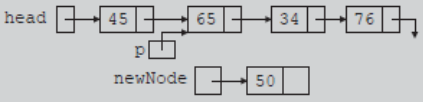
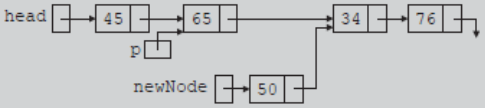
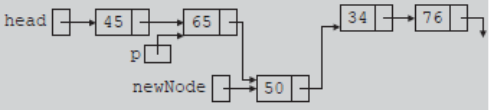
FIGURE 5-6 Linked list before item insertion

Suppose that **p** points to the node with **info 65**, and a new node with **info 50** is to be created and inserted after **p**. Consider the following statements:

```
newNode = new nodename;          //create newNode
newNode->info = 50;              //store 50 in the new node
newNode->link = p->link;
p->link = newNode;
```

Table 5-3 shows the effect of these statements.

TABLE 5-3 Inserting a node in a linked list

Statement	Effect
`newNode = new nodeType;`	
`newNode->info = 50;`	
`newNode->link = p->link;`	
`p->link = newNode;`	

Note that the sequence of statements to insert the node, that is,

```
newNode->link = p->link;
p->link = newNode;
```

is very important because to insert **newNode** in the list we use only one pointer, **p**, to adjust the links of the nodes of the linked list. Suppose that we reverse the sequence of the statements and execute the statements in the following order:

```
p->link = newNode;
newNode->link = p->link;
```

Figure 5-7 shows the resulting list after these statements execute.

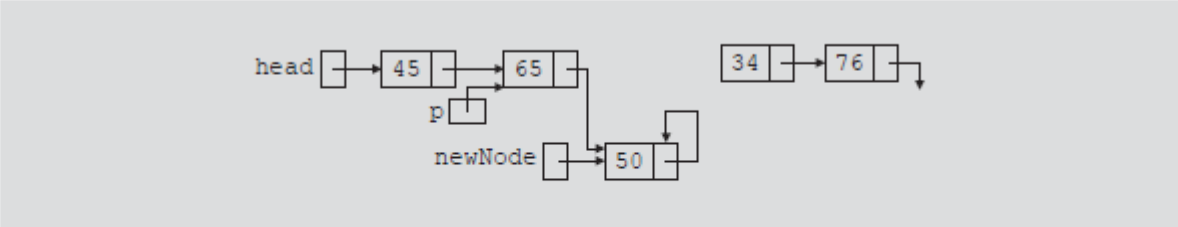


FIGURE 5-7 List after the execution of the statement `p->link = newNode;` followed by the execution of the statement `newNode->link = p->link;`

From Figure 5-7, it is clear that **newNode** points back to itself and the remainder of the list is lost. Using two pointers, we can simplify the insertion code somewhat. Suppose **q** points to the node with **info 34**. (See Figure 5-8.)

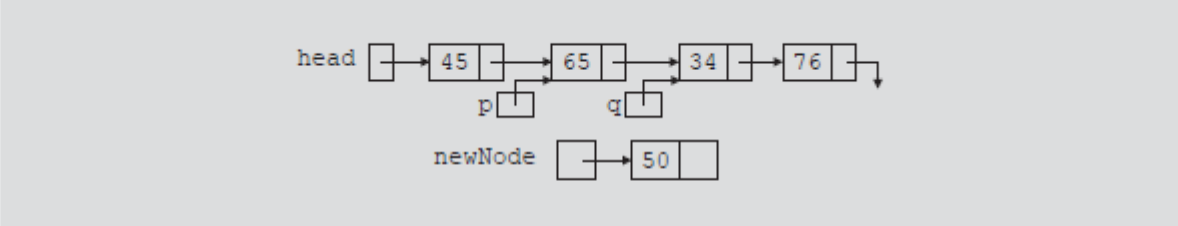


FIGURE 5-8 List with pointers `p` and `q`

The following statements insert **newNode** between **p** and **q**:

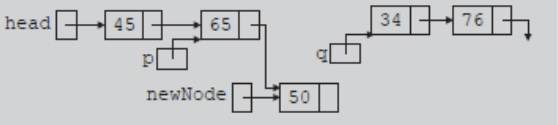
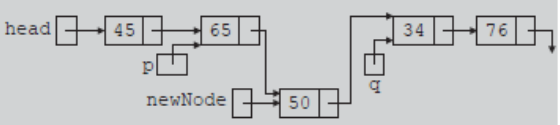
```
newNode->link = q;
p->link = newNode;
```

The order in which these statements execute does not matter. To illustrate this, suppose that we execute the statements in the following order:

```
p->link = newNode;
newNode->link = q;
```

Table 5-4 shows the effect of these statements.

TABLE 5-4 Inserting a node in a linked list using two pointers

Statement	Effect
`p->link = newNode;`	
`newNode->link = q;`	

ALGORITHM FOR DELETING A NODE

Consider the linked list shown in Figure 5-9.

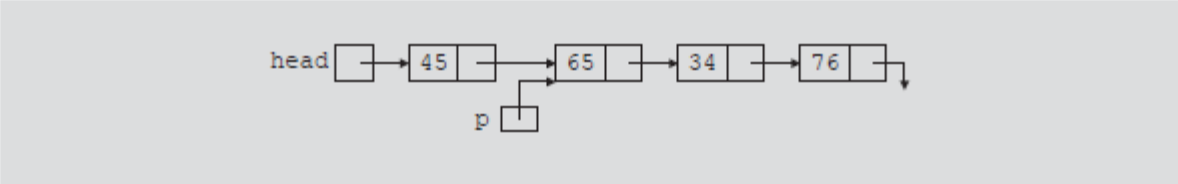


FIGURE 5-9 Node to be deleted is with `info 34`

Suppose that the node with **info 34** is to be deleted from the list. The following statement removes the node from the list:

```
p->link = p->link->link;
```

Figure 5-10 shows the resulting list after the preceding statement executes.

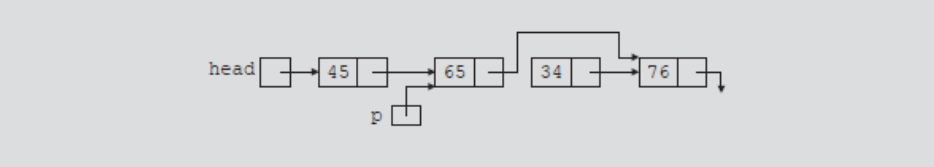


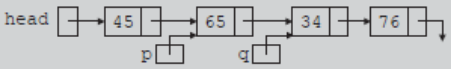
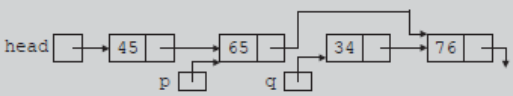
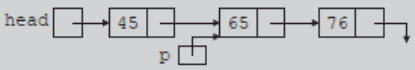
FIGURE 5-10 List after the statement `p->link = p->link->link;` executes

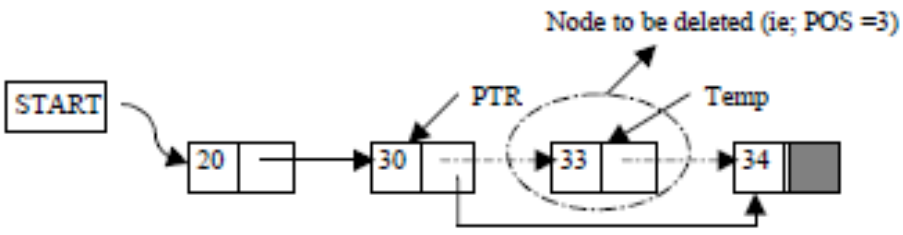
From Figure 5-10, it is clear that the node with **info 34** is removed from the list.
However, the memory is still occupied by this node and this memory is inaccessible; that is, this node is dangling. To deallocate the memory, we need a pointer to this node. The following statements delete the node from the list and deallocate the memory occupied by this node:

```
q = p->link;
p->link = q->link;
delete q;
```

Table 5-5 shows the effect of these statements.

TABLE 5-5 Deleting a node from a linked list

Statement	Effect
`q = p->link;`	
`p->link = q->link;`	
`delete q;`	



Deletion of a Node

Suppose START is the first position in linked list. Let DATA be the element to be deleted. TEMP, HOLD is a temporary pointer to hold the node address.

1. Input the DATA to be deleted
2. if ((START -> DATA) is equal to DATA)
    (a) TEMP = START
    (b) START = START -> Next
    (c) Set free the node TEMP, which is deleted
    (d) Exit
3. HOLD = START
4. while ((HOLD -> Next -> Next) not equal to NULL))
    (a) if ((HOLD -> NEXT -> DATA) equal to DATA)
        (i) TEMP = HOLD -> Next
        (ii) HOLD -> Next = TEMP -> Next
        (iii) Set free the node TEMP, which is deleted
        (iv) Exit
    (b) HOLD = HOLD -> Next
5. if ((HOLD -> next -> DATA) == DATA)
    (a) TEMP = HOLD -> Next
    (b) Set free the node TEMP, which is deleted
    (c) HOLD -> Next = NULL
    (d) Exit
6. Disply "DATA not found"
7. Exit

ALGORITHM FOR SEARCHING A NODE

Suppose START is the address of the first node in the linked list and DATA is the information to be searched. After searching, if the DATA is found, POS will contain the corresponding position in the list.

1. Input the DATA to be searched
2. Initialize TEMP = START; POS =1;
3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL)
4. If (TEMP → DATA is equal to DATA)
    (a) Display "The data is found at POS"
    (b) Exit
5. TEMP = TEMP → Next
6. POS = POS+1
7. If (TEMP is equal to NULL)
    (a) Display "The data is not found in the list"
8. Exit

ALGORITHM FOR DISPLAY ALL NODES

Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end.

1. If (START is equal to NULL)
   (a) Display "The list is Empty"
   (b) Exit
2. Initialize TEMP = START
3. Repeat the step 4 and 5 until (TEMP == NULL)
4. Display "TEMP → DATA"
5. TEMP = TEMP → Next
6. Exit

BUILDING A LINKED LIST

Now that we know how to insert a node in a linked list, let us see how to build a linked list. First, we consider a linked list in general. If the data we read is unsorted, the linked list will be unsorted. Such a list can be built in two ways: forward and backward. In the forward manner, a new node is always inserted at the end of the linked list. In the backward manner, a new node is always inserted at the beginning of the list. We will consider both cases.

BUILDING A LINKED LIST FORWARD

Suppose that the nodes are in the usual **info-link** form and **info** is of type **int**. Let us assume that we process the following data: 2 15 8 24 34

We need three pointers to build the list: one to point to the first node in the list, which cannot be moved, one to point to the last node in the list, and one to create the newnode. Consider the following variable declaration:

```
nodename *first, *last, *newNode;
int num;
```

Suppose that first points to the first node in the list. Initially, the list is empty, so both first and last are NULL. Thus, we must have the statements

```
first = NULL;
last = NULL;
```

to initialize first and last to NULL. Next, consider the following statements:

```
1  cin >> num;                    //read and store a number in num
2  newNode = new nodename;        //allocate memory of type nodename and store the address of
       the
                                  //allocated memory in newNode
3  newNode->info = num;           //copy the value of num into the info field of newNode
4  newNode->link = NULL;          //initialize the link field of newNode to NULL
5  if (first == NULL)             //if first is NULL, the list is empty;
                                  //make first and last point to newNode
       {
5a             first = newNode;
5b             last = newNode;
       }
```

```
6  else                                    //list is not empty
       {
6a          last->link = newNode;        //insert newNode at the end of the list
6b          last = newNode;              //set last so that it points to the
                                           //actual last node in the list
       }
```

Let us now execute these statements. Initially, both **first** and **last** are **NULL**. Therefore, we have the list as shown in Figure 5-11.

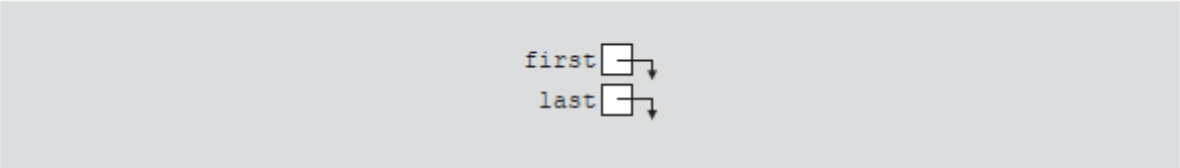


FIGURE 5-11 Empty list

After statement 1 executes, **num** is 2. Statement 2 creates a node and stores the address of that node in **newNode**. Statement 3 stores 2 in the info field of **newNode**, and statement 4 stores **NULL** in the link field of **newNode**. (See Figure 5-12.)

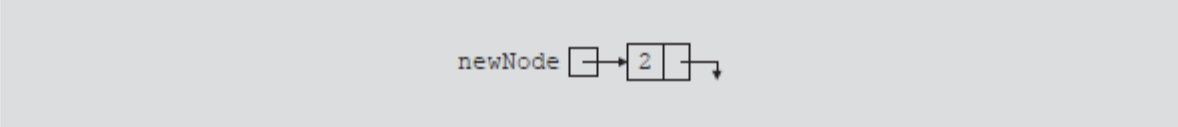


FIGURE 5-12 newNode with info 2

Because first is NULL, we execute statements 5a and 5b. Figure 5-13 shows the resulting list.

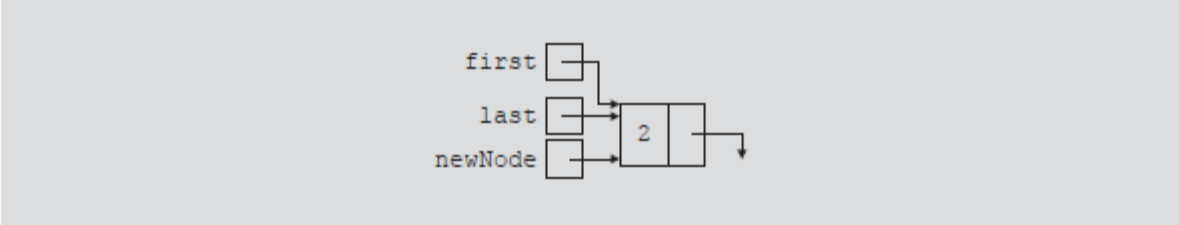


FIGURE 5-13 List after inserting newNode in it

We now repeat statements 1 through 6b. After statement 1 executes, **num** is 15. Statement 2 creates a node and stores the address of this node in **newNode**. Statement 3 stores 15 in the info field of **newNode**, and statement 4 stores **NULL** in the link field of **newNode**. (See Figure 5-14.)

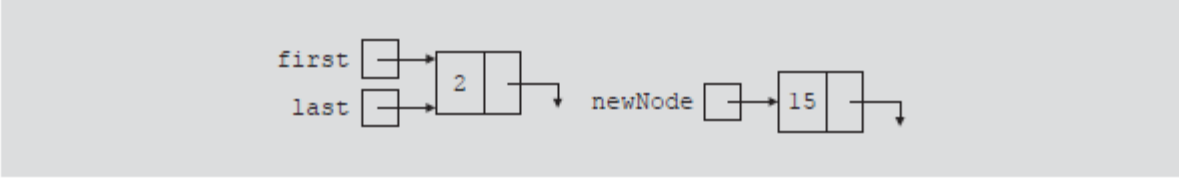


FIGURE 5-14 List and newNode with info 15

Because **first** is not **NULL**, we execute statements 6a and 6b. Figure 5-15 shows the resulting list.

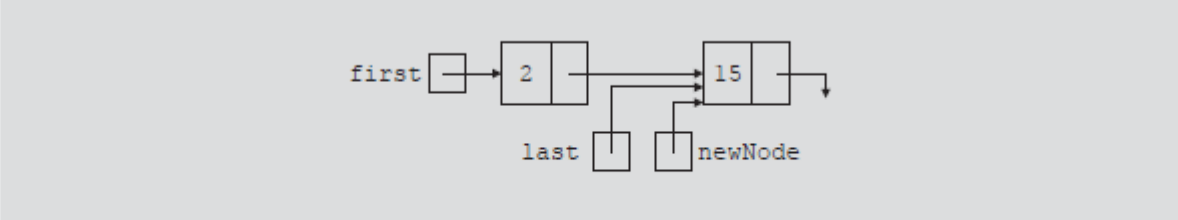


FIGURE 5-15 List after inserting newNode at the end

We now repeat statements 1 through 6b three more times. Figure 5-16 shows the resulting list.

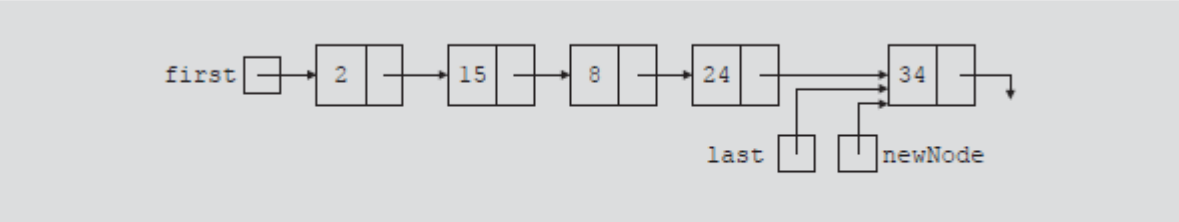


FIGURE 5-16 List after inserting 8, 24, and 34

We can put the previous statements in a loop, and execute the loop until certain conditions are met, to build the linked list. We can, in fact, write a C++ function to build a linked list.

Suppose that we read a list of integers ending with -999. The following function, buildListForward, builds a linked list (in a forward manner) and returns the pointer of the built list:

```
nodename* buildListForward()
{
    nodename *first, *newNode, *last;
    int num;

    cout << "Enter a list of integers ending with -999." << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodename;
        newNode->info = num;
        newNode->link = NULL;

        if (first == NULL)
        {
            first = newNode;
            last = newNode;
        }
        else
        {
            last->link = newNode;
```

```
            last = newNode;
        }
        cin >> num;
    }   //end while

    return first;

}       //end buildListForward
```

BUILDING A LINKED LIST BACKWARD

Now we consider the case of building a linked list backward. For the previously given data—2, 15, 8, 24, and 34—the linked list is as shown in Figure 5-17.

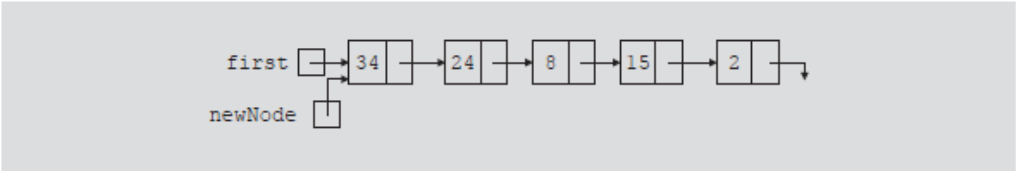


FIGURE 5-17 List after building it backward

Because the new node is always inserted at the beginning of the list, we do not need to know the end of the list, so the pointer **last** is not needed. Also, after inserting the new node at the beginning, the new node becomes the first node in the list. Thus, we need to update the value of the pointer **first** to correctly point to the first node in the list. We see, then, that we need only two pointers to build the linked list: one to point to the list and one to create the new node. Because initially the list is empty, the pointer first must be initialized to **NULL**. The following C++ function builds the linked list backward and returns the pointer of the built list:

```
nodename* buildListBackward()
{
    nodename *first, *newNode;
    int num;

    cout << "Enter a list of integers ending with -999." << endl;
    cin >> num;
    first = NULL;

    while (num != -999)
    {
        newNode = new nodename;      //create a node
        newNode->info = num;         //store the data in newNode
        newNode->link = first;       //put newNode at the beginning
                                     //of the list
        first = newNode;             //update the head pointer of
                                     //the list, that is, first
        cin >> num;                  //read the next number
    }

    return first;
}                                    //end buildListBackward
```

DOUBLY LINKED LIST

A doubly linked list is one in which all nodes are linked together by multiple links which help in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every nodes in the doubly linked list has three fields: LeftPointer, RightPointer and DATA. Fig. 5.22 shows a typical doubly linked list.

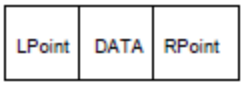


Fig. 5.24. A typical doubly linked list node

LPoint will point to the node in the left side (or previous node) that is LPoint will hold the address of the previous node. RPoint will point to the node in the right side (or nex node) that is RPoint will hold the address of the next node. DATA will store the information of the node.

Fig. 5.25. Doubly Linked List

Fig. 5.26. Memory Representation of Doubly Linked List

Representation of doubly linked list

A node in the doubly linked list can be represented in memory with the following declarations.

```
Struct Node
{
    Int DATA;
    Node *next;
    Node *prev;
};
```

All the operations performed on singly linked list can also be performed on doubly linked list. Following figure will illustrate the insertion and deletion of nodes.

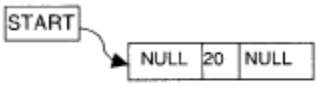


Fig. 5.27. Add(20)

Fig 5.28. Insert (30) at the end

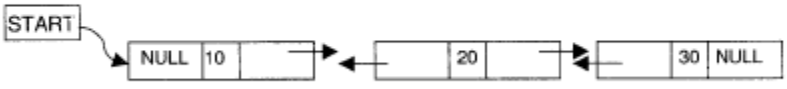


Fig 5.29. Insert (10) at the beginning

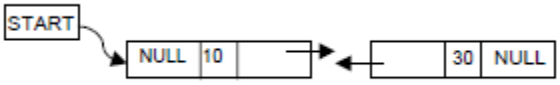


Fig 5.30. Delete a node at the 2nd position

Algorithm for creating a doubly linked list (inserting at the end)

1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->info=DATA
4. TEMP->next=NULL
5. if (START is equal to NULL)
    5.a. TEMP->prev=NULL
    5.b. START=TEMP
    5.c. exit
6. else
    6.a. HOLD =START
    6.b. while(HOLD->next not equal to NULL)
        6.b.1 HOLD=HOLD->next
    6.c. HOLD->next=TEMP
    6.d. TEMP->prev=HOLD
7. exit

Algorithm for inserting a node at the beginning

1. Input the DATA to be inserted
2. TEMP to create a new node
3. TEMP->prev=NULL
4. TEMP->info=DATA
5. TEMP->next=START
6. if (START is equal to NUUl)
    6.a. START=TEMP
    6.b. exit
7. START->prev=TEMP
8. START=TEMP
9. exit

Algorithm for inserting a node at a specific position

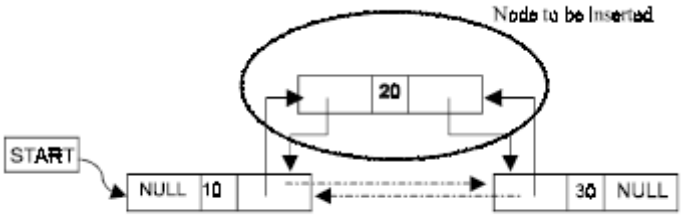


Fig. 5.31. Insert a node at the 2nd position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the NewNode is to be inserted. TEMP is a temporary pointer to hold the node address.

1. Input the DATA and POS
2. Initialize TEMP = START; i = 0

3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP ->next; i = i +1
5. If (TEMP not equal to NULL) and (i equal to POS)
    (a) Create a New Node
    (b) NewNode -> DATA = DATA
    (c) NewNode -> next = TEMP -> next
    (d) NewNode -> prev = TEMP
    (e) (TEMP -> next) -> prev = NewNode
    (f) TEMP -> next = New Node
6. Else
    (a) Display "Position NOT found"
7. Exit

Algorithm for deleting a node

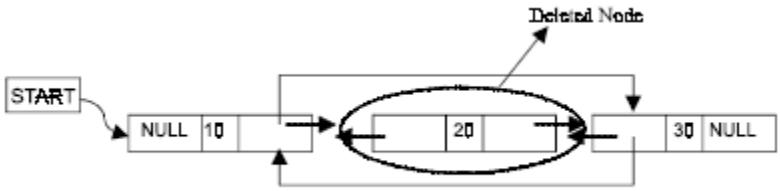


Fig. 5.32. Delete a node at the 2nd position

Suppose START is the address of the first node in the linked list. Let DATA be the Element to be deleted. TEMP, HOLD is the temporary pointer to hold the address of the node.

1. Input the DATA to be deleted
2. if ((START->DATA)is equal to DATA)
    (a) TEMP = START
    (b) START = STAR->next
    (c) START->prev=NULL
    (d) set free the node TEMP , which is deleted
    (e) Exit
3. HOLD = START
4. while((HOLD->next->next) not equal to NULL)
    (a)if (HOLD->next->DATA) equal to DATA)
        (a1) TEMP = HOLD->next
        (a2)HOLD->next=TEMP->next
        (a3) TEMP->next->prev=HOLD
        (a4) set free the node TEMP, which is deleted
        (a5) Exit
    (b) HOLD=HOLD->next
5. if ((HOLD->next->DATA)==DATA)
    (a) TEMP=HOLD->next
    (b) set free the node TEMP, which is deleted
    (c) HOLD->next=NULL
    (d) exit
6. Display "DATA not found"

7.Exit

Algorithm for displaying the doubly linked list

1. if (START is equal to NULL)
    1.a. display “The list is empty”
    1.b. exit
2. TEMP=START
3. while TEMP is not equal to NULL
    3.a. display TEMP->info
    3.b. TEMP=TEMP->next
4. exit

Assignment

1. write an algorithm and a function to display a doubly linked list in reverse order.
2. write an algorithm and a function to count the number of elements in the doubly linked list.
3. write an algorithm and function for searching a number in doubly linked list.

The Stack data structure

A stack is one of the most important and useful non-primitive linear data structure in computer science. It is an ordered collection of items into which new data items may be added/inserted and from which items may be deleted at only one end, called the top of the stack. As all the addition and deletion in a stack is done from the top of the stack, the last added element will be first removed from the stack. That is why the stack is also called Last-in-First-out (LIFO). Note that the most frequently accessible element in the stack is the top most elements, whereas the least accessible element is the bottom of the stack. The operation of the stack can be illustrated as in Fig. 3.1.

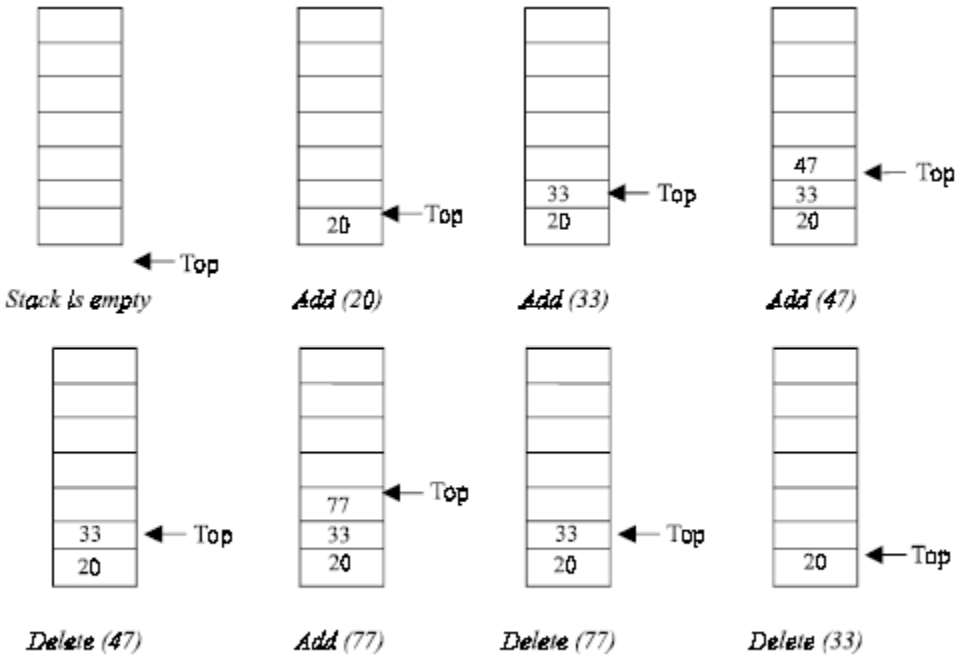


Fig. 3.1. Stack operation.

The insertion (or addition) operation is referred to as push, and the deletion (or remove) operation as pop. A stack is said to be empty or underflow, if the stack contains no elements. At this point the top of the stack is present at the bottom of the stack. And it is overflow when the stack becomes full, i.e., no other elements can be pushed onto the stack. At this point the top pointer is at the highest location of the stack.

OPERATIONS PERFORMED ON STACK

The primitive operations performed on the stack are as follows:

PUSH: The process of adding (or inserting) a new element to the top of the stack is called PUSH operation. Pushing an element to a stack will add the new element at the top. After every push operation the top is incremented by one. If the array is full and no new element can be accommodated, then the stack overflow condition occurs.

POP: The process of deleting (or removing) an element from the top of stack is called POP operation. After every pop operation the stack is decremented by one. If there is no element in the stack and the pop operation is performed then the stack underflow condition occurs.

STACK IMPLEMENTATION

Stack can be implemented in two ways:

1. Static implementation (using arrays)

2. Dynamic implementation (linked list)

Static implementation uses arrays to create stack. Static implementation using arrays is a very simple technique but is not a flexible way, as the size of the stack has to be declared during the program design, because after that, the size cannot be varied (i.e., increased or decreased). Moreover static implementation is not an efficient method when resource optimization is concerned (i.e., memory utilization). For example a stack is implemented with array size 50. That is before the stack operation begins, memory is allocated for the array of size 50. Now if there are only few elements (say 30) to be stored in the stack, then rest of the statically allocated memory (in this case 20) will be wasted, on the other hand if there are more number of elements to be stored in the stack (say 60) then we cannot change the size array to increase its capacity. The above said limitations can be overcome by dynamically implementing (is also called linked list representation) the stack using pointers.

STACK USING ARRAYS

Implementation of stack using arrays is a very simple technique. Algorithm for pushing (or add or insert) a new element at the top of the stack and popping (or delete) an element from the stack is given below.

Algorithm for push

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. Let DATA is the data item to be pushed.

1. If TOP = SIZE – 1, then:
   (*a*) Display "The stack is in overflow condition"
   (*b*) Exit
2. TOP = TOP + 1
3. STACK [TOP] = ITEM
4. Exit

```
void push(void)
{
        int x;
        if(top==max-1)     // Condition for checking If Stack is Full
        {
                cout<<"\nstack overflow\n";
                return;
        }
        cout<<"enter a no:  ";
        cin>>x;
        a[++top]=x;    //increment the top and inserting element
        cout<< "\nsucc. pushed:  " << x << endl << endl;
             return;
}
```

Algorithm for pop

Suppose STACK[SIZE] is a one dimensional array for implementing the stack, which will hold the data items. TOP is the pointer that points to the top most element of the stack. DATA is the popped (or deleted) data item from the top of the stack.

1. If TOP is equal to -1, then
   (a) Display "The Stack is empty"
   (b) Exit
2. DATA = STACK[TOP]
3. TOP = TOP – 1
4. Exit

```
void pop(void)
{
        int y;
        if(top==-1)            // Condition for checking If Stack is Empty
        {
                cout <<"stack underflow\n";
                return;
        }
        y=a[top];
        a[top--]=0;     //insert 0 at place of removing element and decrement the top
        cout <<"\n succ.poped\n\n";
        return;
}
```

Algorithm for display

1. If TOP is equal to -1, then
   (a) Display "the stack is empty"
   (b) exit
2. i ←TOP to 0
   (a) display Array[i]
3. Exit

```
void display(void)
{
        if(top==-1)
        {
               cout <<"stack is empty\n";
               return;
        }
        cout<<"\nelements of Stack are : ";
        for(int i=0;i<=top;i++)
        {
                cout << a[i] << "  ";
         }
             cout << endl << endl;
         return;
}
```

STACK USING LINKED LIST

we have discussed the implementation of stack using array, i.e., static memory allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in following figures.

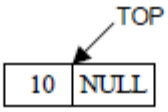


Fig. 5.11. push (10)

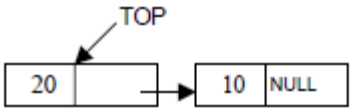


Fig. 5.12. push (20)

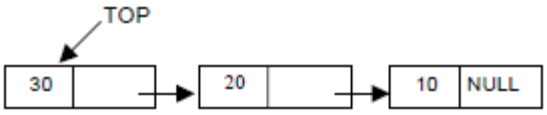


Fig. 5.13. push (30)

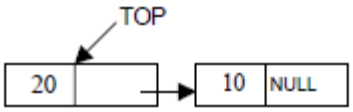


Fig. 5.14. X = pop() (ie; X = 30)

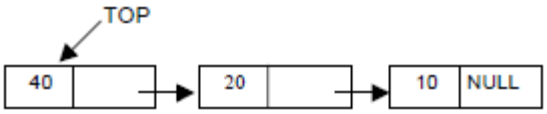


Fig. 5.15. push (40)

Algorithm for push operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Creat a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit

```
void push()
{
     int item;
     Node *NewNode;
     NewNode = new Node;
     cout<<"\ninput the new value to be pushed on the stack: ";
     cin>>item;
     NewNode->info=item;
     NewNode->link=top;
     top=NewNode;
}
```

Algorithm for pop operation

Suppose TOP is a pointer, which is pointing towards the topmost element of the stack. TOP is NULL when the stack is empty. TEMP is pointer variable to hold any nodes address. DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
   (a) Display "The stack is empty"
2. Else
   (a) TEMP = TOP
   (b) Display "The popped element TOP → DATA"
   (c) TOP = TOP → Next
   (d) TEMP → Next = NULL
   (e) Free the TEMP node
3. Exit

```
void pop(){
if (top==NULL)
    cout<<"the stack is empty\n";
else
    {
       Node *temp=top;
       cout<<"the poped element is: "<<top->info;
       top=top->link;
       temp->link=NULL;
       delete temp;
    }
}
```

Algorithm for display operation

1. if (TOP is equal to NULL)
   (a) display "the stack is empty"
   (b) exit
2. else
   (a) temp = top
   (b) while temp is not equal to null
      (b.1) display temp->info
      (b.2) temp = temp->link
3. exit

```
void display(){
if(top==NULL)
    cout<<"the stack is empty"<<endl;
else
{
    cout<<"\nthe stack elements are: "<<endl;
    Node *temp=top;
    while(temp!=NULL)
    {
       cout<<temp->info;
       temp=temp->link;
    }
}
}
```

Stack applications

Expression

An application of stack is calculation of postfix expression. There are basically three types of notation for an expression (mathematical expression; An expression is defined as the number of operands or data items combined with several operators.)

1. Infix notation: A+B

2. Prefix notation: +AB

3. Postfix notation: AB+

The infix notation is what we come across in our general mathematics, where the operator is written in-between the operands. For example: The expression to add two numbers A and B is written in infix notation as: A + B
Note that the operator '+' is written in between the operands A and B.

The prefix notation is a notation in which the operator(s) is written before the operands, it is also called **polish notation**. The same expression when written in prefix notation looks like: + A B
As the operator '+' is written before the operands A and B, this notation is called prefix (pre means before).

In the postfix notation the operator(s) are written after the operands, so it is called the postfix notation (post means after), it is also known as **suffix notation** or **reverse polish notation**.
The above expression if written in postfix expression looks like:
A B +

The prefix and postfix notations are not really as awkward to use as they might look. For example, a C function to return the sum of two variables A and B (passed as argument) is called or invoked by the instruction: add(A, B)
Note that the operator add (name of the function) precedes the operands A and B.

Because the postfix notation is most suitable for a computer to calculate any expression (due to its reverse characteristic), and is the universally accepted notation for designing Arithmetic and Logical Unit (ALU) of the CPU (processor). Therefore it is necessary to study the postfix notation. Moreover the postfix notation is the way computer looks towards arithmetic expression, any expression entered into the computer is first converted into postfix notation, stored in stack and then calculated. In the preceding sections we will study the conversion of the expression from one notation to other.

Advantages of using postfix notation

Human beings are quite used to work with mathematical expressions in infix notation, which is rather complex. One has to remember a set of nontrivial rules while using this notation and it must be applied to expressions in order to determine the final value. These rules include precedence, BODMAS (Order of Operations), and associativity.
Using infix notation, one cannot tell the order in which operators should be applied.
Whenever an infix expression consists of more than one operator, the precedence rules (BODMAS) should be applied to decide which operator (and operand associated with that operator) is evaluated

first. But in a postfix expression operands appear before the operator, so there is no need for operator precedence and other rules. As soon as an operator appears in the postfix expression during scanning of postfix expression the topmost operands are popped off and are calculated by applying the encountered operator. Place the result back onto the stack; likewise at the end of the whole operation the final result will be there in the stack.

Notation Conversions

Let A + B * C be the given expression, which is an infix notation. To calculate this expression for values 4, 3, 7 for A, B, C respectively we must follow certain rule (called BODMAS in general mathematics) in order to have the right result. For example:

A + B * C = 4 + 3 * 7 = 7 * 7 = 49

The answer is not correct; multiplication is to be done before the addition, because multiplication has higher precedence over addition. This means that an expression is calculated according to the operator's precedence not the order as they look like. The error in the above calculation occurred, since there were no braces to define the precedence of the operators. Thus expression A + B * C can be interpreted as A + (B * C). Using this alternative method we can convey to the computer that multiplication has higher precedence over addition.

Operator precedence

Exponential operator	^	Highest precedence
Multiplication/Division	*, /	Next precedence
Addition/Subtraction	+, -	Least precedence

Converting infix to postfix expression

The method of converting infix expression A + B * C to postfix form is:

A + B * C Infix Form
A + (B * C) Parenthesized expression
A + (B C *) Convert the multiplication
A (B C *) + Convert the addition
A B C * + Postfix form

The rules to be remembered during infix to postfix conversion are:

1. Parenthesize the expression starting from left to light.
2. During parenthesizing the expression, the operands associated with operator having higher precedence are first parenthesized. For example in the above expression B * C is parenthesized first before A + B.
3. The sub-expression (part of expression), which has been converted into postfix, is to be treated as single operand.
4. Once the expression is converted to postfix form, remove the parenthesis.

Problem 1. Give postfix form for A + [(B + C) + (D + E) * F] / G

Solution. Evaluation order is

A + { [(BC +) + (DE +) * F] / G}
A + { [(BC +) + (DE +) F *] / G}
A + { [(BC +) (DE + F * +] / G} .

A + [BC + DE + F *+ G /]
ABC + DE + F * + G / + Postfix Form

Problem 2. Give postfix form for (A + B) * C / D + E ^ A / B
Solution. Evaluation order is
[(AB +) * C / D] + [(EA ^) / B]
[(AB +) * C / D] + [(EA ^) B /]
[(AB +) C * D /] + [(EA ^) B /]
(AB +) C * D / (EA ^) B / +
AB + C * D / EA ^ B / + Postfix Form

Algorithm
Suppose P is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression Q. Besides operands and operators, P (infix notation) may also contain left and right parentheses. We assume that the operators in P consists of only exponential (^), multiplication (*), division (/), addition (+) and subtraction (-). The algorithm uses a stack to temporarily hold the operators and left parentheses. The postfix expression Q will be constructed from left to right using the operands from P and operators, which are removed from stack. We begin by pushing a left parenthesis onto stack and adding a right parenthesis at the end of P. the algorithm is completed when the stack is empty.

1. Scan P from left to right and repeat Steps 3 to 6 for each element of P until the stack is empty.
2. If an operand is encountered, add it to Q.
3. If a left parenthesis is encountered, push it onto stack.
4. If an operator $\otimes$ is encountered, then:
   (a) Repeatedly pop from stack and add to Q each operator (on the top of stack), which has the same precedence as, or higher precedence than $\otimes$.
   (b) Add $\otimes$ to stack.
5. If a right parenthesis is encountered, then:
   (a) Repeatedly pop from stack and add to Q (on the top of stack until a left parenthesis is encountered.
   (b) Remove the left parenthesis. [Do not add the left parenthesis to stack.]
6. Exit.

Note. Special character $\otimes$ is used to symbolize any operator in P.
Consider the following arithmetic infix expression P

P = A + (B / C - (D * E ^ F) + G) * H)

Fig. 1 shows the character (operator, operand or parenthesis) scanned, status of the stack and postfix expression Q of the infix expression P.

Character scanned	Stack	Postfix Expression (Q)
A	(	A
+	(+	A
(	(+ (	A
B	(+ (	A B
/	(+ (/	A B
C	(+ (/	A B C
–	(+ (-	A B C /
(	(+ (- (	A B C /
D	(+ (- (	A B C / D
*	(+ (- (*	A B C / D
E	(+ (- (*	A B C / D E
^	(+ (- (* ^	A B C / D E
F	(+ (- (* ^	A B C / D E F
)	(+ (-	A B C / D E F ^ *
+	(+ (+	A B C / D E F ^ * -
G	(+ (+	A B C / D E F ^ * - G
)	(+	A B C / D E F ^ * - G +
*	(+ *	A B C / D E F ^ * - G +
H	(+ *	A B C / D E F ^ * - G + H
)		A B C / D E F ^ * - G + H * +

Fig. 1

H.W. Write a C++ program to implement the algorithm above (converting infix to postfix algorithm).

Evaluating postfix expression

Following algorithm finds the RESULT of an arithmetic expression P written in postfix notation. The following algorithm, which uses a STACK to hold operands, evaluates P.

Algorithm

1. Scan P from left to right and repeat Steps 3 and 4 for each element of P.
2. If an operand is encountered, put it on STACK.
3. If an operator $\otimes$ is encountered, then:
   (a) Remove the two top elements of STACK, where A is the top element and B is the next-to-top element.
   (b) Evaluate B $\otimes$ A.
   (c) Place the result on to the STACK.
4. Result equal to the top element on STACK.
5. Exit.

H.W. Write a C++ program to implement the algorithm above (Evaluating postfix expression).

The Queues

A queue is logically a *first in first out (FIFO or first come first serve)* linear data structure. The concept of queue can be understood by our real life problems. For example a customer come and join in a queue to take the train ticket at the end (rear) and the ticket is issued from the front end of queue. That is, the customer who arrived first will receive the ticket first. It means the customers are serviced in the order in which they arrive at the service center.

It is a homogeneous collection of elements in which new elements are added at one end called *rear*, and the existing elements are deleted from other end called *front*.

The basic operations that can be performed on queue are

1. Insert (or add) an element to the queue (push)
2. Delete (or remove) an element from a queue (pop)

Push operation will insert (or add) an element to queue, at the rear end, by incrementing the array index. Pop operation will delete (or remove) from the front end by decrementing the array index and will assign the deleted value to a variable. Total number of elements present in the queue is front-rear+1, when implemented using arrays. Following figure will illustrate the basic operations on queue.

Fig. 4.1. Queue is empty.

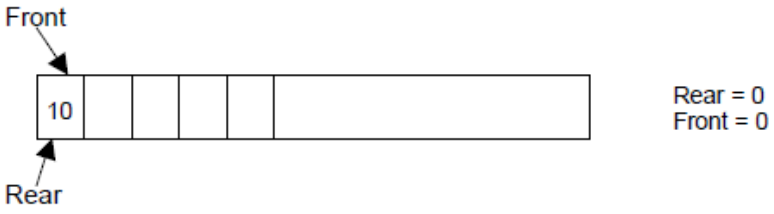


Fig. 4.2. push(10)

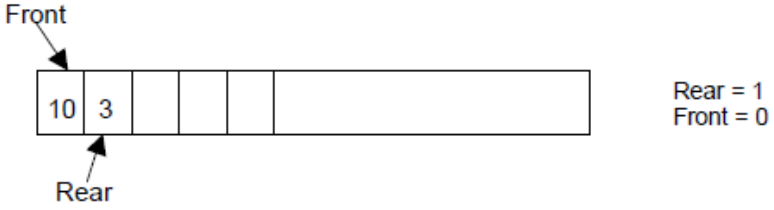


Fig. 4.3. push(3)

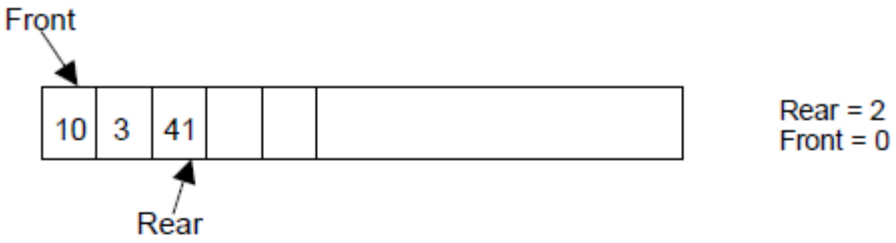


Fig. 4.4. push(41)

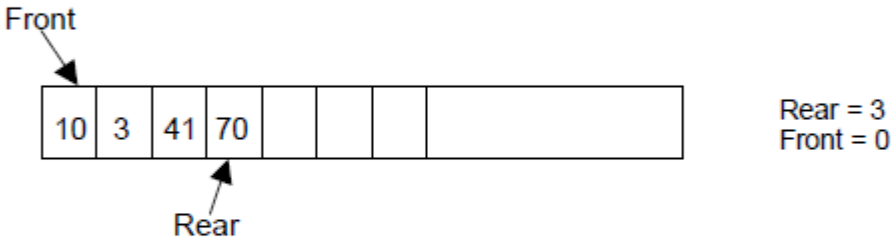


Fig. 4.5. push(70)

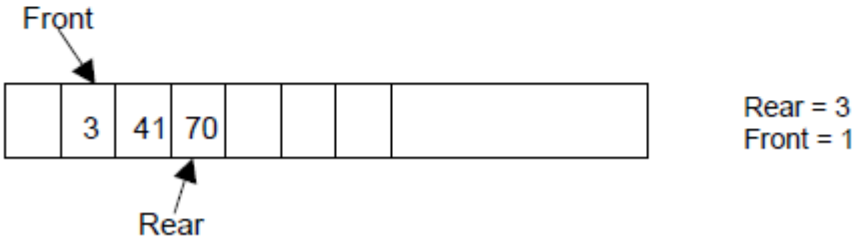


Fig. 4.6. x = pop() (*i.e.*; x = 10)

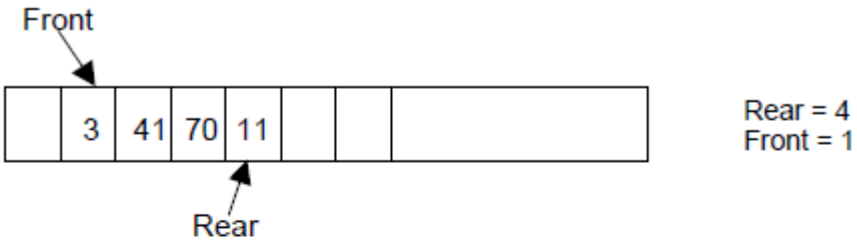


Fig. 4.7. push(11)

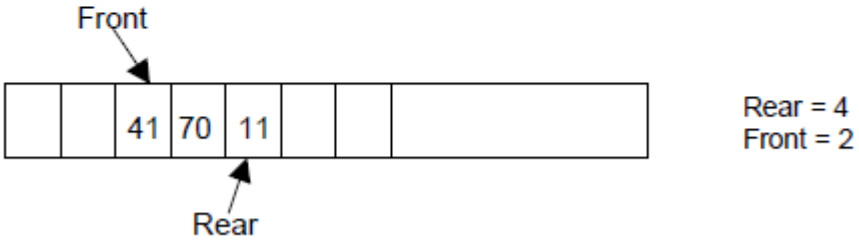


Fig. 4.8. x = pop() (*i.e.*; x = 3)

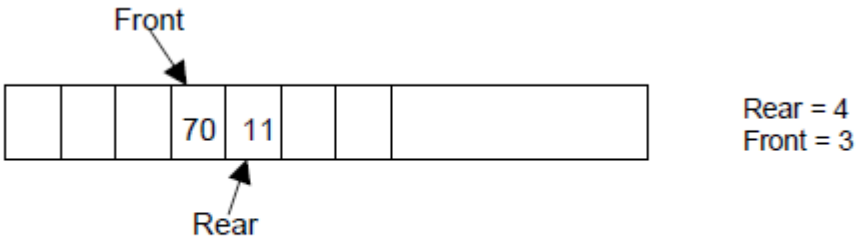


Fig. 4.9. x = pop() (*i.e.*, x = 41)

Queue can be implemented in two ways:

1. Using arrays (static)
2. Using pointers (Linked List) (dynamic)

Implementation of queue using pointers will be discussed later. Let us discuss underflow and overflow conditions when a queue is implemented using arrays.

If we try to pop (or delete or remove) an element from queue when it is empty, underflow occurs. It is not possible to delete (or take out) any element when there is no element in the queue.

Suppose maximum size of the queue (when it is implemented using arrays) is 50. If we try to push (or insert or add) an element to queue, overflow occurs. When queue is full it is naturally not possible to insert any more elements.

ALGORITHM FOR QUEUE OPERATIONS

Let Q be the array of some specified size say SIZE

1- INSERTING AN ELEMENT INTO THE QUEUE

1. Initialize front= –1, rear = –1
2. Input the value to be inserted and assign to variable "data"
3. If (rear = = SIZE-1)
   (*a*) Display "Queue overflow"
   (*b*) Exit
4. Else
   (*a*) Rear = rear +1
5. Q[rear] = data
6. Exit

2- DELETING AN ELEMENT FROM QUEUE

1. If (rear< front) or (front and rear is equal to -1)
    (a) Front = –1, rear = –1
    (b) Display “The queue is empty”
    (c) Exit
2. Else
    (a) Data = Q[front]
3. Front = front +1
4. Exit

3- DISPLAY THE ELEMENTS OF QUEUE

1. If (rear< front) or (front and rear is equal to -1)
    (a) Display “The queue is empty”
    (b) Exit
2. Else
    (a) i=front to rear
        (1) display Queue[i]
    (b) Exit
3. Exit

QUEUE USING LINKED LIST

Queue is a First In First Out [FIFO] data structure. In chapter 4, we have discussed about stacks and its different operations. And we have also discussed the implementation of stack using array, ie; static memory allocation. Implementation issues of the stack (Last In First Out - LIFO) using linked list is illustrated in the following figures.

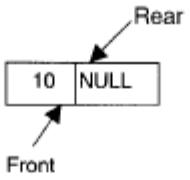


Fig. 5.16. push (10)

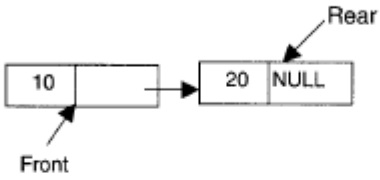


Fig. 5.17. push (20)

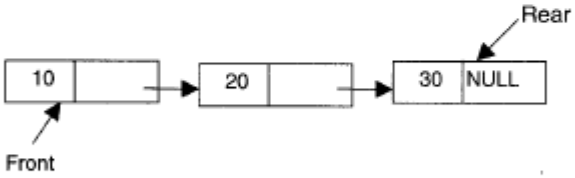


Fig. 5.18. push (30)

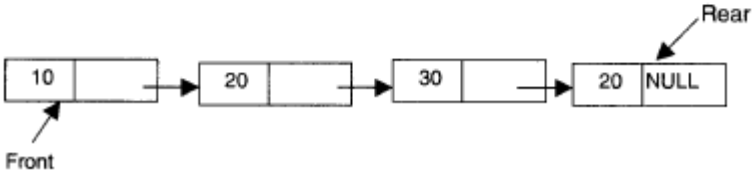


Fig. 5.19. push (40)

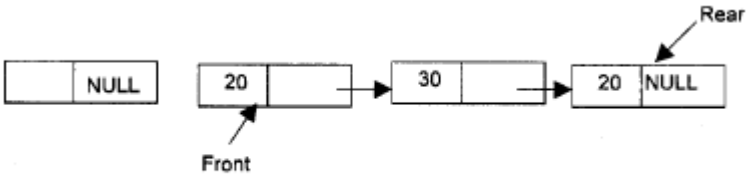


Fig. 5.20. X = pop() (*i.e.*; X = 10)

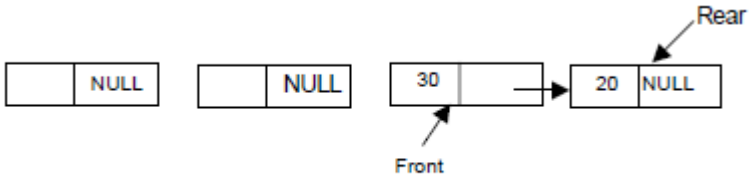


Fig. 5.21. X = pop() (*i.e.*; X = 20)

ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If(front is equal to NULL and rear is equal to NULL)
   (*a*) front = rear = NewNode
   (b) exit
6. rear → next = NewNode
7. rear = NewNode
7. Exit

ALGORITHM FOR POPPING AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. declare temp = FRONT
2. If (FRONT is equal to NULL)
   (a) Display "The Queue is empty"
3. Else if (FRONT is equal to REAR)
   (a) FRONT = REAR = NULL
4. Else

(a) FRONT = FRONT → next

5. delete temp

6. Exit