1.3 DATA STRUCTURES

Data may be organized in many different ways; the logical or mathematical model of a particular organization of data is called a *data structure*. The choice of a particular data model depends on two considerations. First, it must be rich enough in structure to mirror the actual relationships of the data in the real world. On the other hand, the structure should be simple enough that one can effectively process the data when necessary. This section will introduce us to some of the data structures which will be discussed in detail later in the text.

Classification of Data Structures

Data structures are generally classified into primitive and non-primitive data structures. Basic data types such as integer, real, character and boolean are known as primitive data structures. These data types consist of characters that cannot be divided, and hence they are also called simple data types.

The simplest example of non-primitive data structure is the processing of complex numbers. Very few computers are capable of doing arithmetic on complex numbers. Linked-lists, stacks, queues, trees and graphs are examples of non-primitive data structures. Figure 1.1 shows the classification of data structures.

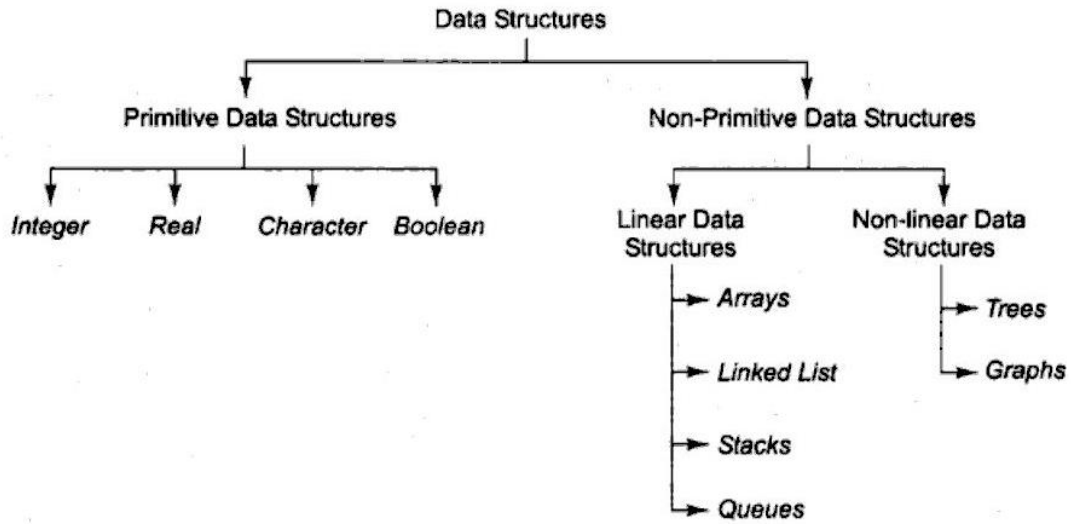


Fig. 1.1 Classification of Data Structures

Based on the structure and arrangement of data, non-primitive data structures are further classified into linear and non-linear.

A data structure is said to be *linear* if its elements form a sequence or a linear list. In linear data structures, the data is arranged in a linear fashion although the way they are stored in memory need not be sequential. Arrays, linked lists, stacks and queues are examples of linear data structures.

Conversely, a data structure is said to be *non-linear* if the data is not arranged in sequence. The insertion and deletion of data is therefore not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

Arrays

The simplest type of data structure is a *linear* (or *one-dimensional*) *array*. By a *linear array*, we mean a list of a finite number n of similar data elements referenced respectively by a set of n consecutive numbers, usually 1, 2, 3, ..., n. If we choose the name A for the array, then the elements of A are denoted by subscript notation

$$a_1, a_2, a_3, \ldots, a_n$$

or by the parenthesis notation

$$A(1), A(2), A(3), \ldots, A(N)$$

or by the bracket notation

$$A[1], A[2], A[3], \ldots, A[N]$$

Regardless of the notation, the number K in A[K] is called a *subscript* and A[K] is called a *subscripted variable.*

Remark: The parentheses notation and the bracket notation are frequently used when the array name consists of more than one letter or when the array name appears in an algorithm. When using this notation we will use ordinary uppercase letters for the name and subscripts as indicated above by the A and N. Otherwise, we may use the usual subscript notation of italics for the name and subscripts and lowercase letters for the subscripts as indicated above by the a and n. The former notation follows the practice of computer-oriented texts whereas the latter notation follows the practice of mathematics in print.

1.4 DATA STRUCTURE OPERATIONS

The data appearing in our data structures are processed by means of certain operations. In fact, the particular data structure that one chooses for a given situation depends largely on the frequency with which specific operations are performed. This section introduces the reader to some of the most frequently used of these operations.

There are basically six operations:

1. *Traversing:* Accessing each record exactly once so that certain items in the record may be processed. (This accessing and processing is sometimes called "visiting" the record.)
2. *Searching:* Finding the location of the record with a given key value, or finding the locations of all records which satisfy one or more conditions.
3. *Inserting:* Adding a new record to the structure.
4. *Deleting:* Removing a record from the structure.

5. Sorting: Arranging the elements of list in an order (either ascending or descending).

6. Merging: combining the two list into one list.

Algorithm:

An algorithm is a well-defined list of steps for solving a particular problem.

The time and space are the two measure for efficiency The complexity of an algorithm is the function which gives the running time and/or space in terms of the input size.

Complexity of Algorithm:

The analysis of algorithms is a major task in computer science. In order to compare algorithms, we must **have some criteria to measure the efficiency of our algorithms.**

Suppose M is an algorithm, and suppose n is the size of the input data. The time and space used by the algorithm M are the two main measures for the efficiency of M. The time is measured by counting the number of key operations—in sorting and searching algorithms, for example, the number of comparisons. That is because key operations are so defined that the time for the other operations is much less than or at most proportional to the time for the key operations. The space is measured by counting the maximum of memory needed by the algorithm.

The *complexity* of an algorithm M is the *function* $f(n)$ which gives the running time and/or storage space requirement of the algorithm in terms of the size n of the input data. Frequently, the storage space required by an algorithm is simply a multiple of the data size n. Accordingly, unless otherwise stated or implied, the term "complexity" shall refer to the running time of the algorithm.

Time complexity is of three types:

1. *Worst case:* the maximum value of $f(n)$ for any possible input
2. *Average case:* the expected value of $f(n)$

Sometimes we also consider the minimum possible value of $f(n)$, called the *best case.*

Asymptotic Notation:

1. Big Oh Notation:

Suppose M is an algorithm, and suppose n is the size of the input data. Clearly the complexity $f(n)$ of M increases as n increases. It is usually the rate of increase of $f(n)$ that we want to examine. This is usually done by comparing $f(n)$ with some standard function, such as

$$\log_2 n, \qquad n \log_2 n, \qquad n^2, \qquad n^3, \qquad 2^n$$

Definition

Suppose $f(n)$ and $g(n)$ are functions defined on the positive integers with the property that $f(n)$ is bounded by some multiple of $g(n)$ for almost all n. That is, suppose there exist a positive integer n_0 and a positive number M such that, for all $n > n_0$, we have

$$|f(n)| \leq M|g(n)|$$

Then we may write

$$f(n) = O(g(n))$$

which is read "$f(n)$ is of order $g(n)$." For any polynomial $P(n)$ of degree m, we show in Solved

For example:

$$8n^3 - 576n^2 + 832n - 248 = O(n^3)$$

Omega Notation (Ω)

The omega notation is used when the function $g(n)$ defines a lower bound for the function $f(n)$.

Definition

$f(n) = \Omega(g(n))$ (read as f of n is omega of g of n), iff there exists a positive integer n_0 and a positive number M such that $|f(n)| \geq M|g(n)|$, for all $n \geq n_0$.

For $f(n) = 18n + 9$, $f(n) > 18n$ for all n, hence $f(n) = \Omega(n)$. Also, for $f(n) = 90n^2 + 18n + 6$, $f(n) > 90n^2$ for $n \geq 0$ and therefore $f(n) = \Omega(n^2)$.

For $f(n) = \Omega(g(n))$, $g(n)$ is a lower bound function and there may be several such functions, but it is appropriate that the function which is almost as large a function of n as possible such that the definition of Ω is satisfied, is chosen as $g(n)$. Thus for example, $f(n) = 5n + 1$ leads to both $f(n) = \Omega(n)$ and $f(n) = \Omega(1)$. However, we never consider the latter to be correct, since $f(n) = \Omega(n)$ represents the largest possible function of n satisfying the definition of Ω and hence is more informative.

Theta Notation (Θ)

The theta notation is used when the function $f(n)$ is bounded both from above and below by the function $g(n)$.

Definition

$f(n) = \Theta(g(n))$ (read as f on n is theta of g of n) iff there exist two positive constants c_1 and c_2, and a positive integer n_0 such that $c_1|g(n)| \leq |f(n)| \leq c_2|g(n)|$ for all $n \geq n_0$.

From the definition it implies that the function $g(n)$ is both an upper bound and a lower bound for the function $f(n)$ for all values of n, $n \geq n_0$. In other words, $f(n)$ is such that, $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

For $f(n) = 18n + 9$, since $f(n) > 18n$ and $f(n) \leq 27n$ for $n \geq 1$, we have $f(n) = \Omega(n)$ and $f(n) = O(n)$ respectively, for $n \geq 1$. Hence $f(n) = \Theta(n)$. Again, $16n^2 + 30n - 90 = \Theta(n^2)$ and $7.2^n + 30n = \Theta(2^n)$.

Array:

Data structures are classified as either linear or nonlinear. A data structure is said to be linear if its elements form a sequence, or, in other words, a linear list. There are two basic ways of representing such linear structures in memory. One way is to have the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called *arrays* and form the main subject matter of this chapter. The other way is to have the linear relationship between the elements represented by means of pointers or links. These linear structures are called linked list.

The operations one normally performs on any linear structure, whether it be an array or a linked list, include the following:

(a) *Traversal*. Processing each element in the list.
(b) *Search*. Finding the location of the element with a given value or the record with a given key.
(c) *Insertion*. Adding a new element to the list.
(d) *Deletion*. Removing an element from the list.
(e) *Sorting*. Arranging the elements in some type of order.
(f) *Merging*. Combining two lists into a single list.

Length or size of array can be given as

$$\text{Length} = UB - LB + 1$$

Where LB is the lower bound(first index) and UB is the upper bound(last index) of the array.

Representation of Linear Array in memory:

Let LA be a linear array in the memory of the computer. Recall that the memory of the computer is simply a sequence of addressed locations as pictured in Fig. 4.3. Let us use the notation

LOC(LA[K]) = address of the element LA[K] of the array LA

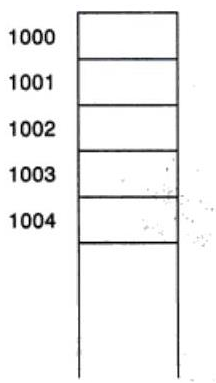


Fig: Computer Memory

As previously noted, the elements of LA are stored in successive memory cells. Accordingly, the computer does not need to keep track of the address of every element of LA, but needs to keep track only of the address of the first element of LA, denoted by

Base(LA)

and called the *base address* of LA. Using this address *Base*(LA), the computer calculates the address of any element of LA by the following formula:

$$\text{LOC (LA[K])} = \textit{Base}(\text{LA}) + w(\text{K} - \text{lower bound})$$

where w is the number of words per memory cell for the array LA. Observe that the time to calculate LOC(LA[K]) is essentially the same for any value of K. Furthermore, given any subscript K, one can locate and access the content of LA[K] without scanning any other element of LA.

Example:

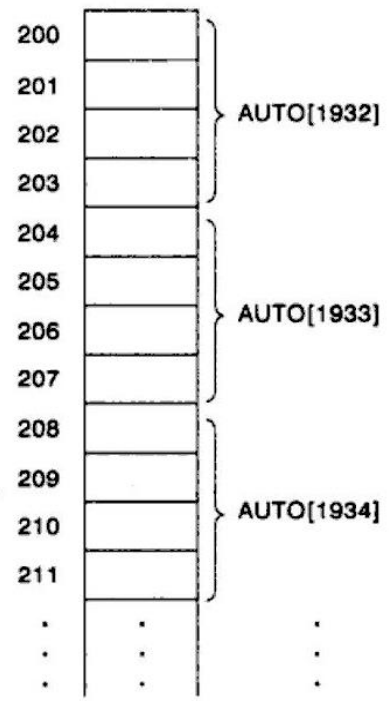
Consider the array AUTO in Example 4.1(b), which records the number of automobiles sold each year from 1932 through 1984. Suppose AUTO appears in memory as pictured in Fig. 4.4. That is, *Base*(AUTO) = 200, and w = 4 words per memory cell for AUTO. Then

LOC(AUTO[1932]) = 200, LOC(AUTO[1933]) = 204, LOC(AUTO[1934]) = 208, ...

The address of the array element for the year K = 1965 can be obtained by using Eq. (4.2):

$$\begin{aligned} \text{LOC(AUTO[1965])} &= \textit{Base}(\text{AUTO}) + w(1965 - \text{lower bound}) \\ &= 200 + 4(1965 - 1932) = 332 \end{aligned}$$

Again we emphasize that the contents of this element can be obtained without scanning any other element in array AUTO.



Traversing the Linear Array:

Let A be a collection of data elements stored in the memory of the computer. Suppose we want to print the contents of each element of A or suppose we want to count the number of elements of A with a given property. This can be accomplished by *traversing* A, that is, by accessing and processing (frequently called *visiting)* each element of A exactly once.

The following algorithm traverses a linear array LA. The simplicity of the algorithm comes from the fact that LA is a linear structure. Other linear structures, such as linked lists, can also be easily traversed. On the other hand, the traversal of nonlinear structures, such as trees and graphs, is considerably more complicated.

Insertion and Deletion

Let A be a collection of data elements in the memory of the computer. "Inserting" refers to the operation of adding another element to the collection A, and "deleting" refers to the operation of removing one element from array.

Inserting an element at the "end" of a linear array can be easily done provided the memory space allocated for the array is large enough to accommodate the additional element. On the other hand, suppose we need to insert an element in the middle of the array. Then, on the average, half of the elements must be moved downward to new locations to accommodate the new element and keep the order of the other elements.

Similarly, deleting an element at the "end" of an array presents no difficulties, but deleting an element somewhere in the middle of the array would require that each subsequent element be moved one location upward in order to "fill up" the array.
"downward" refers to locations with larger subscripts, and the term "upward" refers to locations with smaller subscripts.

(Inserting into a Linear Array) INSERT (LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm inserts an element ITEM into the Kth position in LA.

1. [Initialize counter.] Set J : = N.
2. Repeat Steps 3 and 4 while $J \geq K$.
3. [Move Jth element downward.] Set LA[J + 1] := LA[J].
4. [Decrease counter.] Set J := J – 1.
   [End of Step 2 loop.]
5. [Insert element.] Set LA[K] := ITEM.
6. [Reset N.] Set N := N + 1.
7. Exit.

The following algorithm deletes the Kth element from a linear array LA and assigns it to a variable ITEM.

(Deleting from a Linear Array) DELETE(LA, N, K, ITEM)
Here LA is a linear array with N elements and K is a positive integer such that $K \leq N$. This algorithm deletes the Kth element from LA.

1. Set ITEM := LA[K].
2. Repeat for J = K to N – 1:
   [Move J + 1st element upward.] Set LA[J] := LA[J + 1].
   [End of loop.]
3. [Reset the number N of elements in LA.] Set N: = N – 1.
4. Exit.

Searching:

Searching is a process of checking and finding an element from a list of elements. Let A be a collection of data elements, *i.e.,* A is a linear array of say *n* elements. If we want to find the presence of an element "*data*" in A, then we have to search for it. The search is successful if *data* does appear in A and unsuccessful if otherwise. There are several types of searching techniques; one has some advantage(s) over other. Following are the important searching techniques:

1. Linear or Sequential Searching
2. Binary Searching

Let DATA be a collection of data elements in memory, and suppose a specific ITEM of information is given. *Searching* refers to the operation of finding the location LOC of ITEM in DATA, or printing some message that ITEM does not appear there. The search is said to be *successful* if ITEM does appear in DATA and *unsuccessful* otherwise.

Linear Search

Suppose DATA is a linear array with *n* elements. Given no other information about DATA, the most intuitive way to search for a given ITEM in DATA is to compare ITEM with each element of DATA one by one. That is, first we test whether DATA[1] = ITEM, and then we test whether DATA[2] = ITEM, and so on. This method, which traverses DATA sequentially to locate ITEM, is called *linear search* or *sequential search.*

To simplify the matter, we first assign ITEM to DATA[N + 1], the position following the last element of DATA. Then the outcome

$$LOC = N + 1$$

where LOC denotes the location where ITEM first occurs in DATA, signifies the search is unsuccessful. The purpose of this initial assignment is to avoid repeatedly testing whether or not we have reached the end of the array DATA. This way, the search must eventually "succeed."

ALGORITHM FOR LINEAR SEARCH

Let A be an array of *n* elements, A[1],A[2],A[3], ... A[*n*]. "*data*" is the element to be searched. Then this algorithm will find the location "*loc*" of data in A. Set *loc* = – 1,if the search is unsuccessful.

1. Input an array A of n elements and "data" to be searched and initialise *loc* = – 1.
2. Initialise *i* = 0; and repeat through step 3 if (*i* < *n*) by incrementing *i* by one .
3. If (data = A[*i*])
(*a*) loc = *i*
(*b*) GOTO step 4
4. If (loc > 0)
(*a*) Display "data is found and searching is successful"
5. Else
(*a*) Display "data is not found and searching is unsuccessful"
6. Exit

BINARY SEARCH

Binary search is an extremely efficient algorithm when it is compared to linear search. Binary search technique searches "data" in minimum possible comparisons. Suppose the given array is a sorted one, otherwise first we have to sort the array elements. Then apply the following conditions to search a "data".

1. Find the middle element of the array (*i.e.*, *n*/2 is the middle element if the array or the sub-array contains *n* elements).
2. Compare the middle element with the data to be searched, then there are following three cases.
(*a*) If it is a desired element, then search is successful.
(*b*) If it is less than desired data, then search only the first half of the array, *i.e.*, the elements which come to the left side of the middle element.
(*c*) If it is greater than the desired data, then search only the second half of the array, *i.e.*, the elements which come to the right side of the middle element.

Repeat the same steps until an element is found or exhaust the search area.

ALGORITHM FOR BINARY SEARCH

Let A be an array of n elements A[1],A[2],A[3],..... A[*n*]. "Data" is an element to be searched. "mid" denotes the middle location of a segment (or array or sub-array) of the element of A. LB and UB is the lower and upper bound of the array which is under consideration.

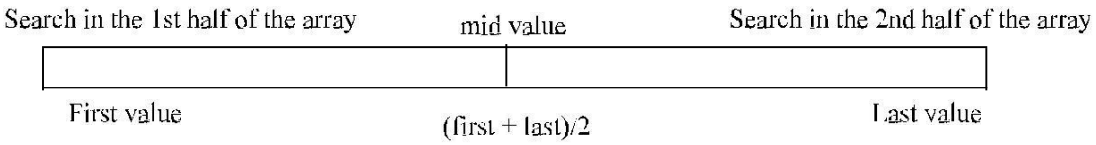


Fig. 7.1

1. Input an array A of n elements and "data" to be sorted
2. LB = 0, UB = *n*; mid = int ((LB+UB)/2)
3. Repeat step 4 and 5 while (LB <= UB) and (A[mid] ! = data)
4. If (data < A[mid])
   (*a*) UB = mid–1
5. Else
   (*a*) LB = mid + 1
6. Mid = int ((LB + UB)/2)
7. If (A[mid]== data)
   (*a*) Display "the data found" 8.

Else

   (*a*) Display "the data is not found" 9.

Exit

Suppose we have an array of 7 elements

9	10	25	30	40	45	70
0	1	2	3	**4**	5	**6**

Following steps are generated if we binary search a data = 45 from the above array.

Step 1:

LB						UB
9	10	25	30	40	45	70
0	1	2	3	4	5	6

LB = 0; UB = 6 mid =

(0 + 6)/2 = 3 A[mid] =

A[3] = 30

Step 2:

Since (A[3] < data) - *i.e.,* 30 < 45 - reinitialise the variable LB, UB and mid

			LB			UB
9	10	25	30	40	45	70
0	1	2	3	4	5	6

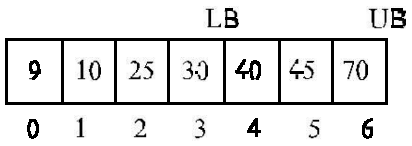
LB = 3 UB = 6 mid =

(3 + 6)/2 = 4 A[mid] =

A[4] = 40

Step 3:

Since (A[4] < data) - *i.e.,* 40 < 45 - reinitialise the variable LB, UB and mid



LB = 4 UB = 6 mid =

(4 + 6)/2 = 5 A[mid] =

A[5] = 45

Step 4:

Since (A[5] == data) - *i.e.,* 45 == 45 - searching is successful.

Linked List

If the memory is allocated for the variable during the compilation (*i.e.; before execution)* of a program, then it is fixed and cannot be changed. For example, an array A[100] is declared with 100 elements, then the allocated memory is fixed and cannot decrease or increase the SIZE of the array if required. So we have to adopt an alternative strategy to allocate memory only when it is required. There is a special data structure called linked list that provides a more flexible storage system and it does not require the use of arrays.

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.

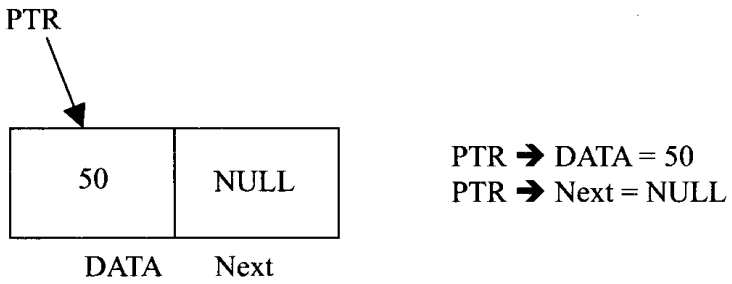


Fig. 5.1. Nodes.

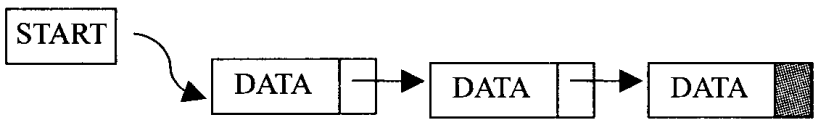


Fig. 5.2. Linked List.

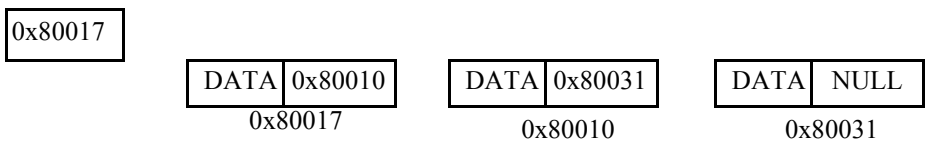


Fig. 5.3. Linked List representation in memory.

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).

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).

Suppose we want to store a list of integer numbers using linked list. Then it can be schematically represented as

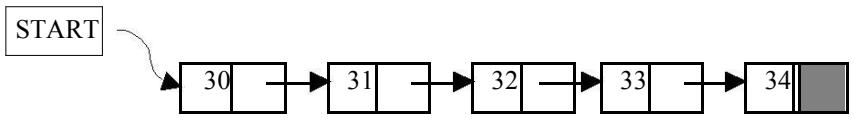


Fig. 5.4. Linked list representation of integers

The linear linked list can be represented in memory with the following declaration.

```
struct Node
{
int DATA; //Instead of 'DATA' we also use 'Info'
struct Node *Next; //Instead of 'Next' we also use 'Link'
};
typedef struct Node *NODE;
```

ADVANTAGES AND DISADVANTAGES

Linked list have many advantages and some of them are:

1. Linked list are dynamic data structure. That is, they can grow or shrink during the execution of a program.
2. Efficient memory utilization: In linked list (or dynamic) representation, memory is not pre-allocated. Memory is allocated whenever it is required. And it is deallocated (or removed) when it is not needed.
3. Insertion and deletion are easier and efficient. Linked list provides flexibility in inserting a data item at a specified position and deletion of a data item from the given position.
4. Many complex applications can be easily carried out with linked list.

Linked list has following disadvantages

1. More memory: to store an integer number, a node with integer data and address field is allocated. That is more memory space is needed.
2. Access to an arbitrary data item is little bit cumbersome and also time consuming.

OPERATION ON LINKED LIST

The primitive operations performed on the linked list are as follows

1. Creation
2. Insertion
3. Deletion
4. Traversing
5. Searching
6. Merging

Creation operation is used to create a linked list. Once a linked list is created with one node, insertion operation can be used to add more elements in a node.

Insertion operation is used to insert a new node at any specified location in the linked list. A new node may be inserted.

(*a*) At the beginning of the linked list

(*b*) At the end of the linked list
(*c*) At any specified position in between in a linked list

Deletion operation is used to delete an item (or node) from the linked list. A node may be deleted from the
(*a*) Beginning of a linked list
(*b*) End of a linked list
(*c*) Specified location of the linked list

Traversing is the process of going through all the nodes from one end to another end of a linked list. In a singly linked list we can visit from left to right, forward traversing, nodes only. But in doubly linked list forward and backward traversing is possible.

Merging is the process of appending the second list to the end of the first list. Consider a list A having n nodes and B with m nodes. Then the operation concatenation will place the 1st node of B in the (n+1)th node in A. After concatenation A will contain (n+m) nodes.

TYPES OF LINKED LIST

Basically we can divide the linked list into the following four types in the order in which they (or node) are arranged.

1. Singly linked list
2. Doubly linked list
3. Circular linked list
4. Circular doubly linked List

The maintenance of linked lists in memory assumes the possibility of inserting new nodes into the lists and hence requires some mechanism which provides unused memory space for the new nodes. Analogously, some mechanism is required whereby the memory space of deleted nodes becomes available for future use.

Together with the linked lists in memory, a special list is maintained which consists of unused memory cells. This list, which has its own pointer, is called the *list of available space* or the *free-storage list* or the *free pool.*

Overflow and Underflow

Sometimes new data are to be inserted into a data structure but there is no available space, i.e., the free-storage list is empty. This situation is usually called *overflow.* The programmer may handle overflow by printing the message OVERFLOW. In such a case, the programmer may then modify the program by adding space to the underlying arrays. Observe that overflow will occur with our linked lists when AVAIL = NULL and there is an insertion.

Analogously, the term *underflow* refers to the situation where one wants to delete data from a data structure that is empty. The programmer may handle underflow by printing the message UNDERFLOW. Observe that underflow will occur with our linked lists when START = NULL and there is a deletion.

Singly Linked List

All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure. Following figure explains the different operations on a singly linked list.

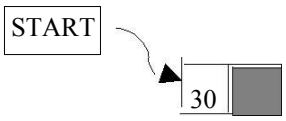


Fig. 5.5. Create a node with DATA(30)

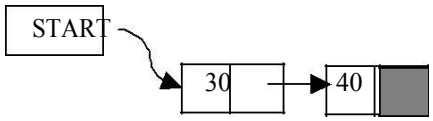


Fig. 5.6. Insert a node with DATA(40) at the end

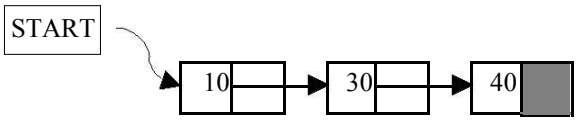


Fig. 5.7. Insert a node with DATA(10) at the beginning

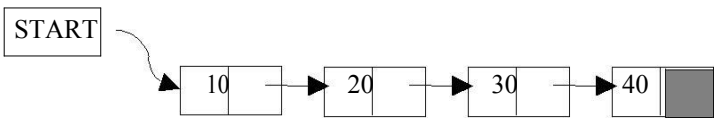


Fig. 5.8. Insert a node with DATA(20) at the 2nd position

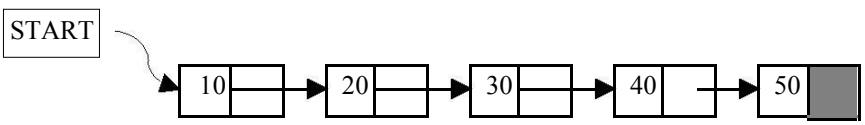


Fig. 5.9. Insert a node with DATA(50) at the end

Output → 10, 20, 30, 40, 50

Fig. 5.10. Traversing the nodes from left to right

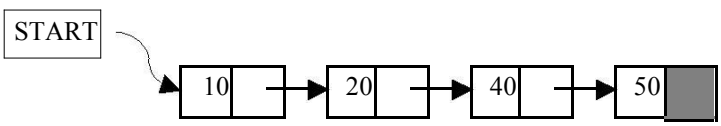


Fig. 5.11. Delete the 3rd node from the list

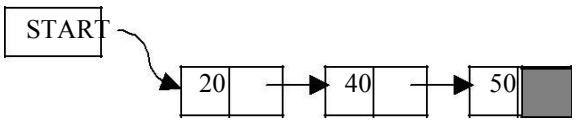


Fig. 5.12. Delete the 1st node

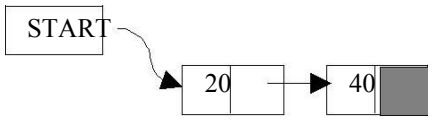


Fig. 5.13. Delete the last node

ALGORITHM FOR INSERTING A NODE

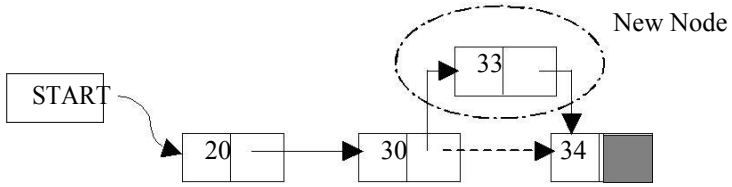


Fig. 5.14. Insertion of New Node

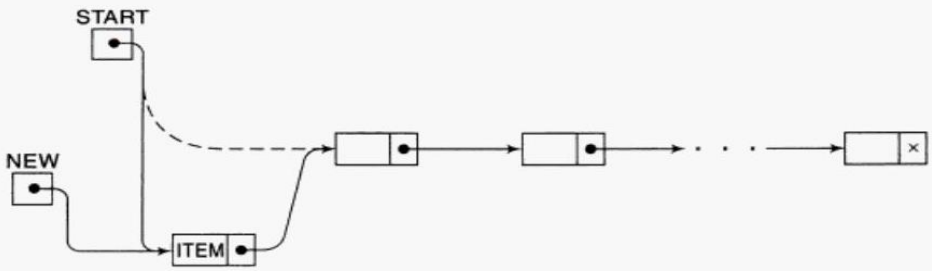
Inserting at the Beginning of a List

Suppose our linked list is not necessarily sorted and there is no reason to insert a new node in any special place in the list. Then the easiest place to insert the node is at the beginning of the list. An algorithm that does so follows.

Algorithm 5.4: INSFIRST(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM as the first node in the list.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node]
4. Set LINK[NEW] := START. [New node now points to original first node.]
5. Set START := NEW. [Changes START so it points to the new node.]
6. Exit.

Steps 1 to 3 have already been discussed, and the schematic diagram of Steps 2 and 3 appears in Fig. 5.18. The schematic diagram of Steps 4 and 5 appears in Fig. 5.19.



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 end

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
8. If (SATRT equal to NULL)
   (*a*) START = NewNode
9. Else
   (*a*) TEMP = START
   (*b*) While (TEMP → Next not equal to NULL) (*i*)
   TEMP = TEMP → Next
10. TEMP → Next = NewNode
11. Exit

Insert a Node at any specified position

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

ALGORITHM FOR DELETING A NODE

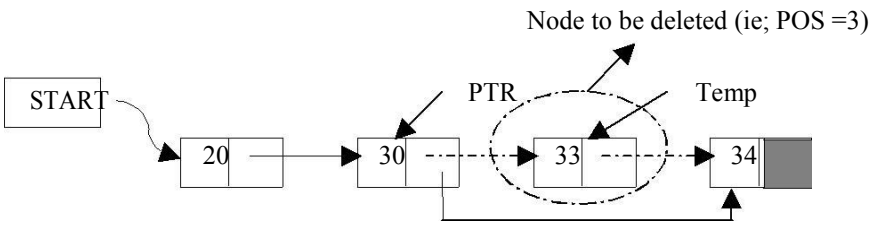


Fig. 5.15. Deletion of a Node.

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 algo-rithm 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

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 next node) that is RPoint will hold the address of the next node. DATA will store the information of the node.

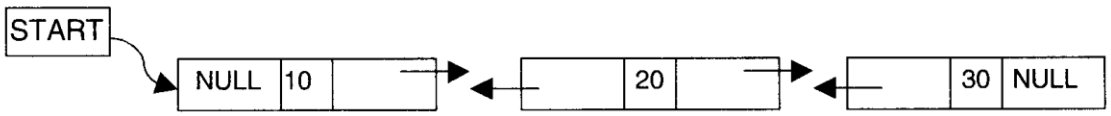


Fig. 5.25. Doubly Linked List

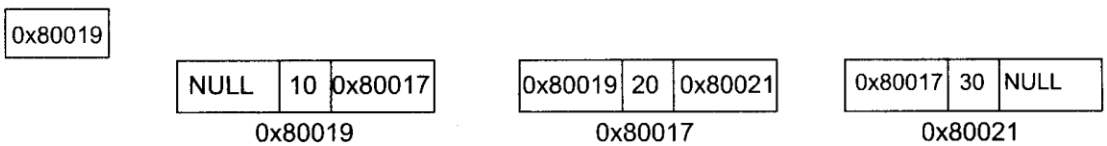


Fig. 5.26. Memory Representation of Doubly Linked List

5.11.1. 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;
        struct Node *RChild;
        struct Node *LChild;
};
        typedef struct Node *NODE;
```

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)

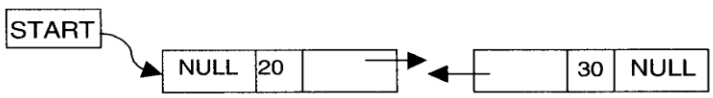


Fig 5.28. Insert (30) at the end

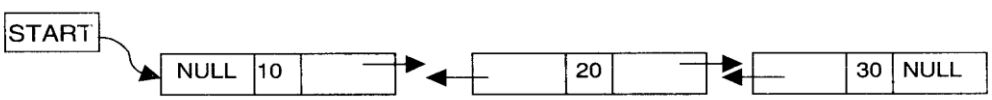


Fig 5.29. Insert (10) at the beginning

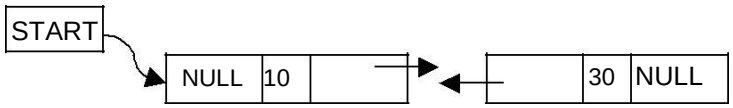


Fig 5.30. Delete a node at the 2nd position

Algorithm for Creation:

```
tmp=create a new node
tmp->info=num //assigning the data to the new node
tmp->next=NULL;
if(start==NULL)

      tmp->prev=NULL;
      start->prev=tmp;
      start=tmp;

else
       q=start;
       while(q->next!=NULL)
       q=q->next;
      q->next=tmp;
      tmp->prev=q;
```

Algorithm for insertion at Begining

```
//a new node is created for inserting the data
tmp=create a new node
tmp->prev=NULL;
tmp->info=num;
tmp->next=start;
start->prev=tmp;
start=tmp;
```

ALGORITHM FOR INSERTING AT ANY POSITION NODE

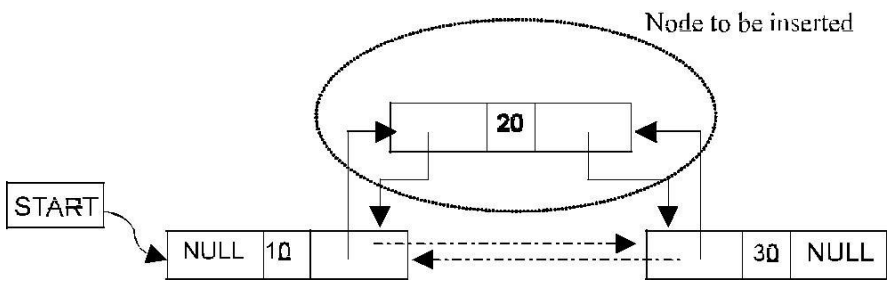


Fig. 5.31. Insert a node at the 2nd position

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 → RPoint; 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 → RPoint = TEMP → RPoint
   (*d*) NewNode → LPoint = TEMP
   (*e*) (TEMP → RPoint) → LPoint = NewNode
   (*f*) TEMP → RPoint = New Node
6. Else
   (*a*) Display "Position NOT found"
7. Exit

ALGORITHM FOR DELETING A NODE

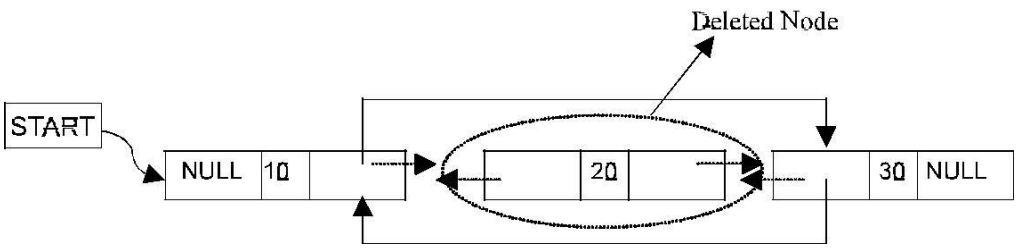


Fig. 5.32. Delete a node at the 2nd position

Suppose START is the address of the first node in the linked list. Let POS is the position of the node to be deleted. TEMP is the temporary pointer to hold the address of the node. After deletion, DATA will contain the information on the deleted node.

1. Input the 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 → RPoint; 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 → RPoint = TEMP → RPoint
   (*d*) NewNode → LPoint = TEMP
   (*e*) (TEMP → RPoint) → LPoint = NewNode
   (*f*) TEMP → RPoint = New Node
6. Else
   (*a*) Display "Position NOT found"
7. Exit

CIRCULAR LINKED LIST

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. A circular linked list is shown in Fig. 5.33.

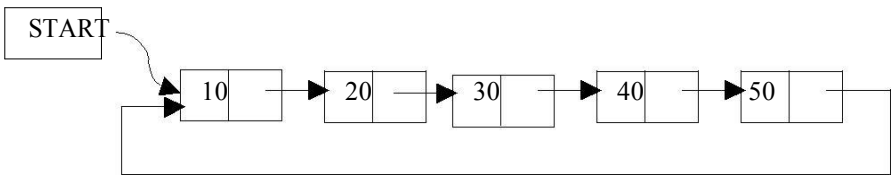


Fig. 5.33. Circular Linked list

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner as shown in the Fig. 5.34.

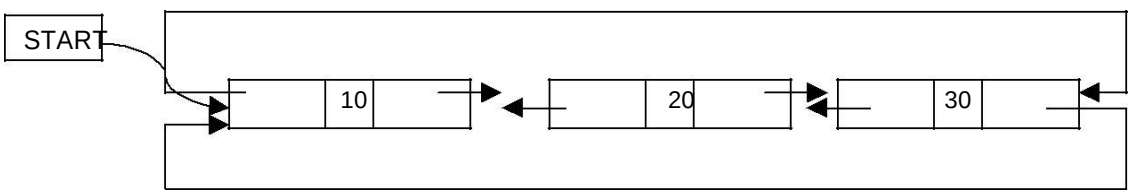


Fig. 5.34. Circular Doubly Linked list

STACK:

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.

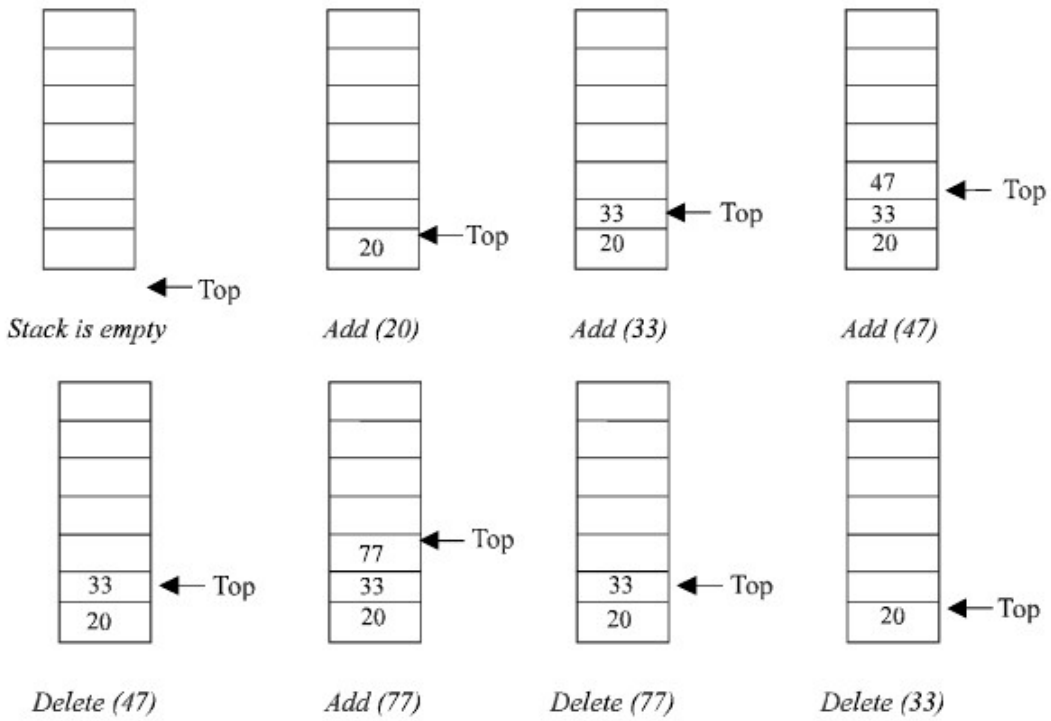


Fig. 3.1. Stack operation.

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 (using pointers)

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

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 < 0, then
(*a*) Display "The Stack is empty"
(*b*) Exit
2. Else remove the Top most element
3. DATA = STACK[TOP]
4. TOP = TOP – 1
5.Exit.