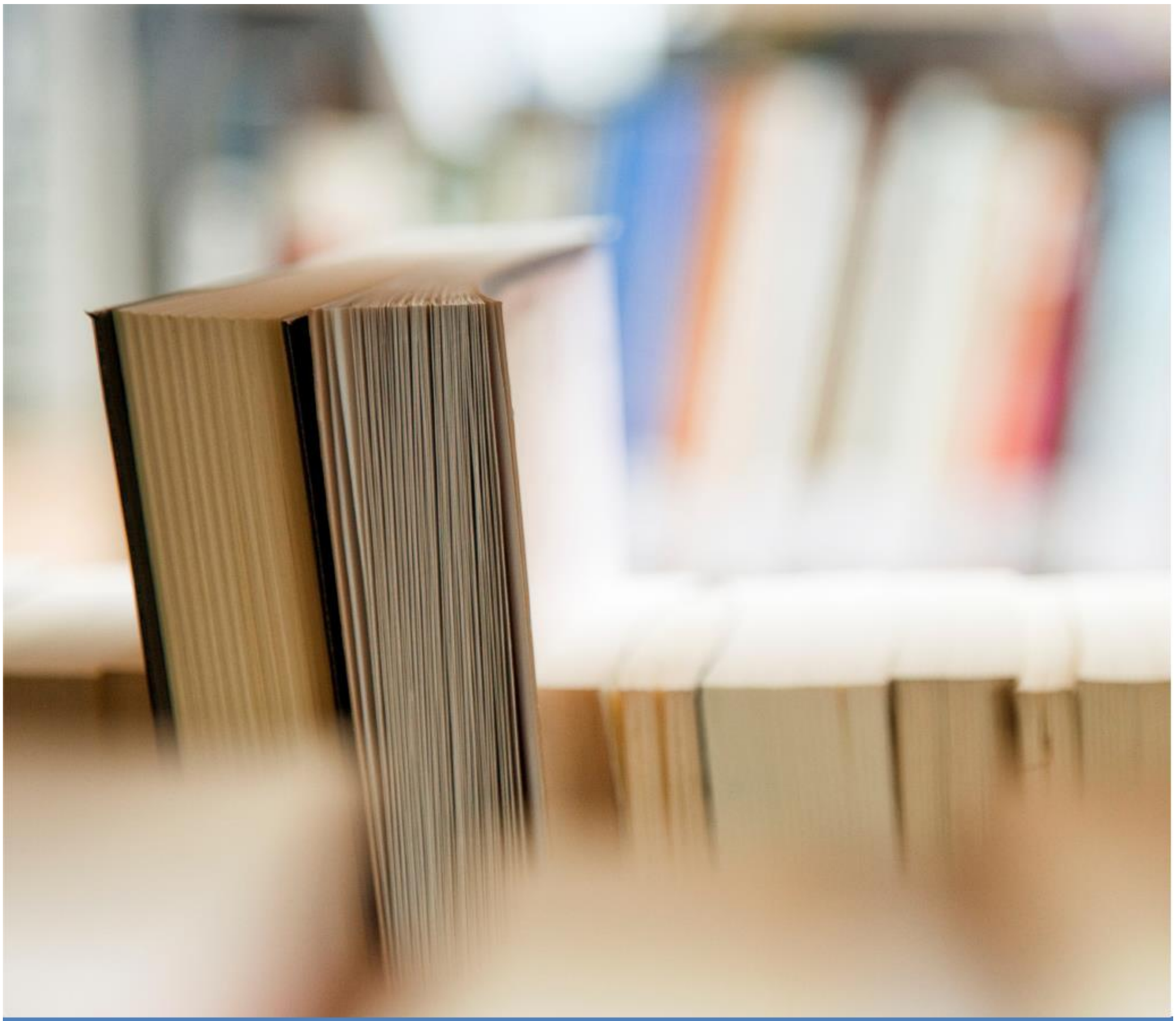


DATA STRUCTURES

BY : AMAR PANCHAL

INDEX

CHAPTER 3

QUEUES

INTRODUCTION

A queue is one of the simplest data structures and can be implemented with different methods on a computer. Queue of tasks to be executed in a computer is analogous to the queue that we see in our daily life at various places. The theory of a queue is common to all. This data structure is very useful in solving various real life problems.

A queue is a non-primitive, linear data structure, and a sub-class of list data structure. It is an ordered, homogenous collection of elements in which elements are appended at one end called rear end and elements are deleted at other end called front end. The meaning of front is face side and rear means back side. The first entry in a queue to which the service is offered is to the element that is on front. After servicing, it is removed from the queue. The information is manipulated in the same sequence as it was collected. Queue follows the rule first-in-first-out (FIFO). Fig. 3.1 illustrates a queue.

Figure 3.1. Queue

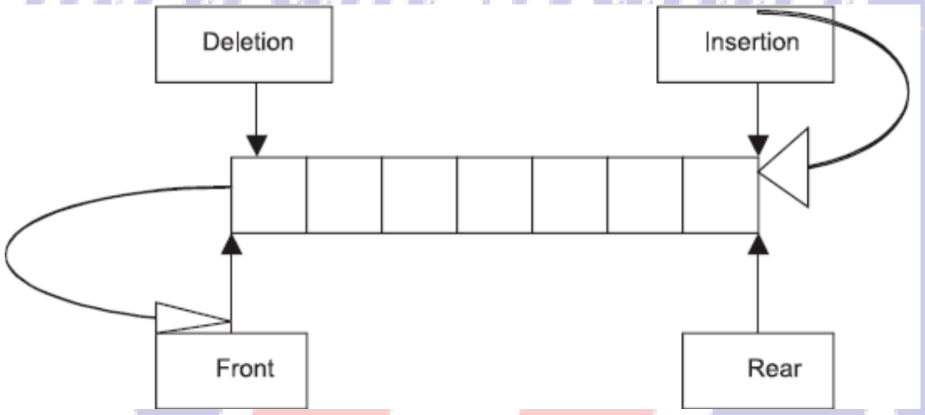


Figure 3.1 shows that insertion of elements is done at the rear end and deletion at the front end.

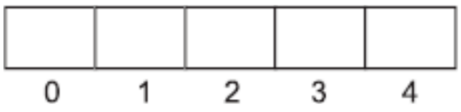
In other words, the service is provided to the element, which is at front and this element is to be removed first. We call this entry front of the queue. The entry of element that is recently added is done from rear of the queue or it can be called as tail end of the queue.

Two operations are frequently carried on queue. They are insertion and deletion of elements. These two operations depend upon the status of the queue. The insertion operation is possible, only when the queue contains elements less than the capacity it can hold. The deletion operation is only possible when the queue contains at least one element.

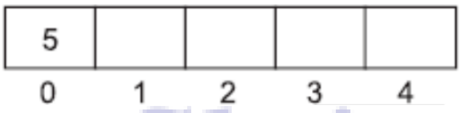
DISADVANTAGES OF SIMPLE QUEUES

In the previous sections we have studied implementation of queues using arrays and pointers. There are some disadvantages in simple queue implementation using arrays. In the following example, it is considered that the elements after deletion are not shifted to beginning of an array. Consider the following example:

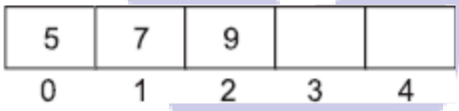
Queue [5] is a simple queue declared. The queue is initially empty. In the following figures insertion and deletion operations will be performed.



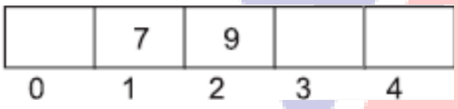
The above figure is an empty queue. Initially the queue is always empty. Here, rear = -1 and front = -1. The user can also assign 0 instead of -1. However, the -1 is suitable because we are implementing this problem with arrays and an array element counting begins always with zero. Thus, it is suitable for problem logic.



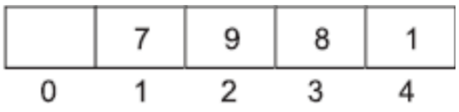
In the above figure, one element is inserted. So, the new values of front and rear are increased and reaches to 0 (zero). This is the only stage where front and rear have same values.



In the above figure, two elements are appended in the queue. At this moment, the value of rear = 2 and front = 0.-



In the above figure one element is deleted and the value of rear = 2 and f = 1. The value of front is increased due deletion of one element.



In the above figure, two more elements are appended to queue. The value of rear = 4 and front = 1.

If more elements are deleted, the value of front will be increased and as a result, the beginning portion of queue will be empty. In such a situation if we try to insert new elements to queue, no elements can be inserted in the queue. This is because, new elements are always inserted from the rear and if the last location of queue (rear) holds an element, even though the space is available at the beginning of queue, no elements will be inserted. The queue will be treated as overflow.

To overcome this problem, we have to update the queue. The solution to this problem is circular queue.

Operations :

Three operations can be performed on a queue.

1. insert(x)- Inserts item x at the rear of the queue q.

2. x=remove(); Deletes the front element from the queue and sets x to its contents.
3. empty()- Returns false or true depending on whether or not the queue contains any elements.
4. Full()-Returns false or true depending on whether or not the queue contains maximum elements.
5. Element_at_rear()-print element at rear.
6. Element_at_front()-print element at front.

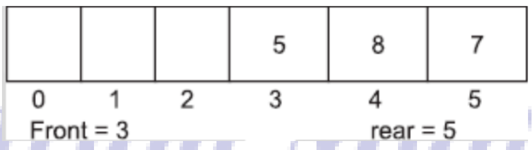
TYPES OF QUEUES

In the last few topics, we have studied simple queue and already seen the disadvantages. When rear pointer reaches to the end of the queue (array), no more elements can be added in the queue, even if beginning memory locations of array are empty. To overcome the above disadvantage, different types of queues can be applied. Different types of queues are:

3.8.1 Circular Queue

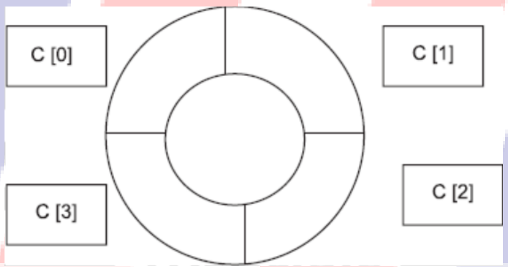
The simple or in a straight line queue there are several limitations we know, even if memory locations at the beginning of queue are available, elements cannot be inserted as shown in the Fig. 3.9. We can efficiently utilize the space available of straight-line queue by using circular queue.

Figure 3.9. Simple queues with empty spaces



In the above discussion, it is supposed that front is not shifted to the beginning of the queue. As shown in the Fig. 3.10 a circular queue is like a wheel or a perfect ring. Suppose, c [4] is an array declared for implementation of circular queue. However, logically elements appear in circular fashion but physically in computer memory, they are stored in successive memory locations.

Figure 3.10. Circular queue



In the circular queue, the first element is stored immediately after last element. If last location of the queue is occupied, the new element is inserted at the first memory location of queue implementing an array (queue). For example, C[n], C is queue of n elements. After inserting, storing an element in the last memory location [(n-1th) element], the next element will be stored at the first location, if space is available. It is like a chain of where starting and ending points are joined and a circle is formed. When an element is stored in the last location wrapping takes place and element inserting routine of the program pointed to beginning of the queue.

Recall that a pointer (stack and queue pointer) plays an important role to know the position of the elements in the stack and queue. Here, as soon as last element is filled, the pointer is wrapped to the beginning of the queue by shifting the pointer value to one.

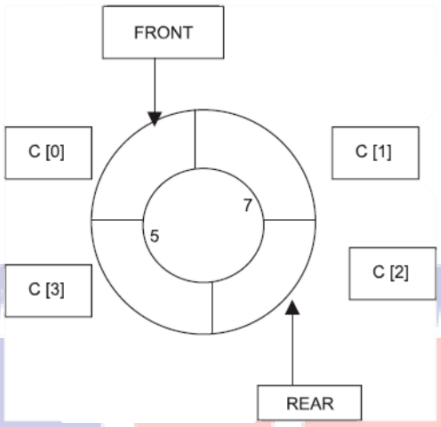
A circular queue overcomes the limitation of the simple queue by utilizing the entire space of the queue.

Like a simple queue, the circular queue also have front and rear ends. The rear and front ends help us to know the status of the queue after the insertion and deletion operations. In the circular queue implementation, the element shifting operation of queue that we apply in the simple queue is not required. The pointer itself takes care to move at vacant location of the queue and element is placed at that location.
In order to simulate the circular queue a programmer should follow the following points:

1. The front end of the queue always points to the first element of the queue.
2. If the values of front and queue are equal, the queue is empty. The values of front and rear pointers are only increased / decreased after insertion/deletion operation.
3. Like simple queue, rear pointer is incremented by one when a new element is inserted and front pointer is incremented by one when an element is deleted.

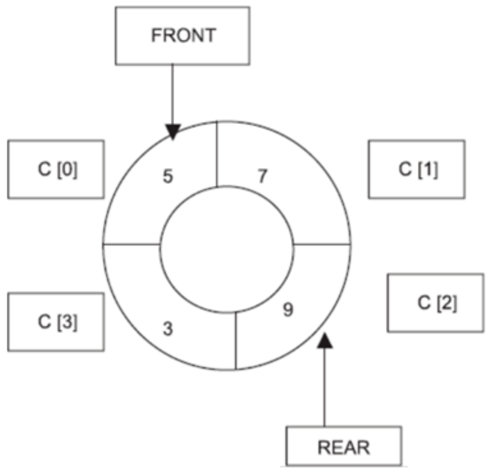
Insertion

Figure 3.11(a). Insertion in a queue



As per Fig. 3.11(a), the insertion of an element in a queue will be same as in a linear queue. The programmer must have to trace the values of front and rear variables.
In Fig. 3.11 (a), only two elements are there in the queue. If we continue to add elements, the queue would be as shown in Fig. 3.11 (b). The new element is always inserted from the rear end. The position where the next element is to be placed can be calculated by the following formula.
REAR = (1+*REAR*) % *MAXSIZE*
C[REAR] = *NUMBER*

Figure 3.11(b). Full queue

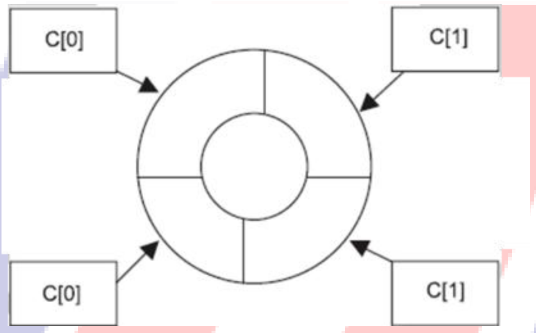


From the Fig. 3.11(b), the value of rear is three and the capacity to store maximum element of queue is four. Therefore,

REAR = (1+*REAR*) % *MAXSIZE*
REAR = (1+3)%4
REAR = 4%4 = 0.

The operator % is modular divisor operator and returns remainder. Thus, in the above equation the remainder obtained is zero. The value of front as well as rear is zero, it means stack is full. Any attempt to insert new element will display the "queue full" message. Consider, the following Fig. 3.11 (c),

Figure 3.11(c). Circular queue

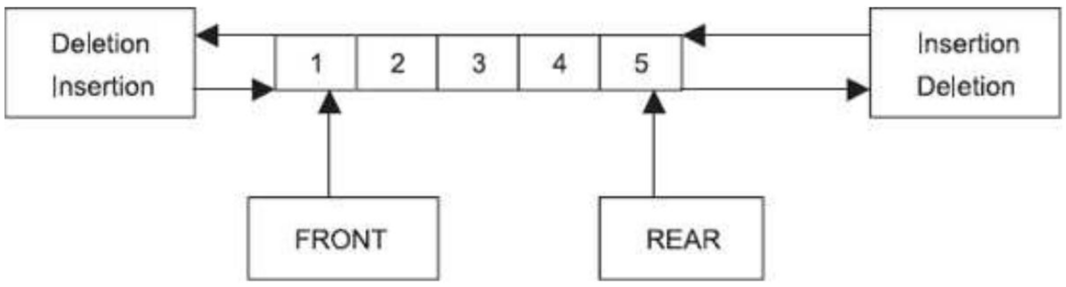


Double Ended Queues

Till now, we have studied stack and queue. The stack has only one end and can be used alternately to insert and delete elements. On the other hand, the double-ended queue has two ends, front and rear. The front end is used to remove element and rear end is used to insert elements. Here, in the double-ended queues, insertion and deletion can be performed from both the ends and therefore, it is called as double-ended queue and in short deque. It is a homogenous list of elements. The deque is a general representation of both stack and queue and can be used as stack and queue. There are various methods to implement deque. Linked list and array can be used to perform the deque. The array implementation is easy and straightforward.

Consider the following Fig. 3.14:

Figure 3.14. Deque



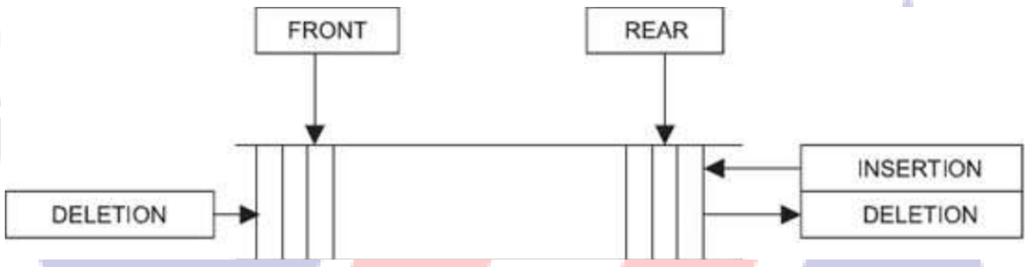
In deque, it is necessary to fix the type of operation to be performed on front and rear ends. The deque can be classified into two types:

Input Restricted Deque

In the input restricted deque, insertion and deletion operation are performed at rear end whereas only deletion operation is performed at front end. The Fig. 3.15 represents the input restricted deque. The following operations are possible in the input restricted deque:

1. Insertion of an element at the rear end.
2. Deletion of element at both front and rear ends.

Figure 3.15. Input restricted deque

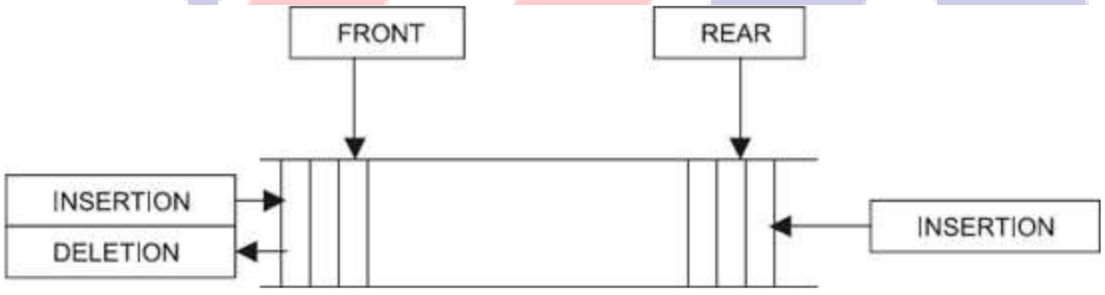


Thus, input restricted deque allows insertion at one and deletion at both ends.

Output Restricted Deque

In the output restricted deque insertion of an element can be done at both front and rear end and deletion operation can be done only at front end. The output restricted deque is shown in Fig. 3.16.

Figure 3.16. Output restricted deque



The following operations are possible in the output restricted deque:

1. Insertion at both front and rear end.
2. Deletion at only front end.

Priority Queues

We know that queue is based on the technique first come first out (FIFO). The element inserted first will be deleted first. A priority queue is another type of queue. Here, the priority is determined according to the position of the queue, which is to be entered. Various real life problems are based on priority queues. For example, in the buses few seats are reserved for ladies and handicapped; if a company is providing any scheme, the employees of the same company are given second priority.

Priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

There are two types of priority queues:

Ascending Priority Queue

Descending Priority Queue

In this queue also, elements can be inserted randomly but the largest element is deleted first.

In both the above types, if elements with equal priority are present, the FIFO technique is applied. In case the queue elements are sorted, the deletion of an element will be quick and easy because elements will appear either in ascending or descending order. However, the insertion operation will be easier said than done because empty locations will have to be searched and only then elements can be placed at that location.

In case the queue elements are not in order, i.e. not sorted, insertion operation will be quick and easy.

However, the deletion operation will take place after the priority value set. Therefore, we can say that in both the above types insertion operation can be carried out easily. However, the deletion operation, which is, based on certain priority, i.e. the smallest or largest is first searched out and later it is removed from the queue. The locations of that element do not matter. In
stack and queues deletion operation is performed at the ends. Conceptually, there is no provision for deleting an element, which is neither first nor last element of the list.

In the above program, we have not arranged the elements in an ascending or descending order. Just smallest or largest element is searched and erased.

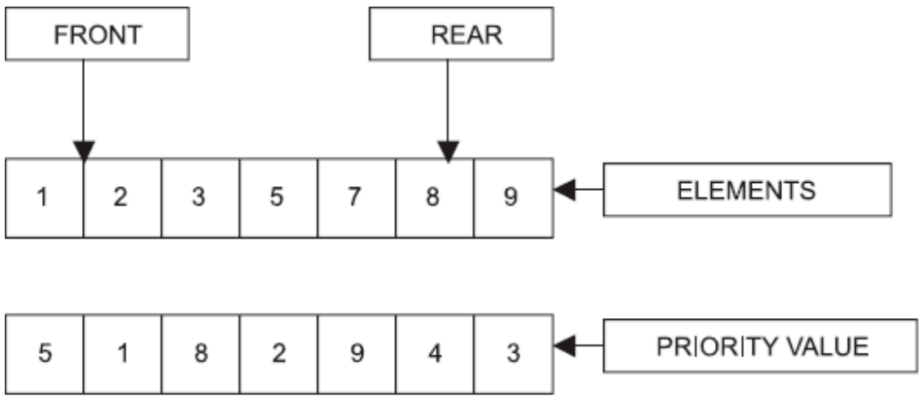
In deletion operation the element may not be physically removed from the queue and it can be kept inaccessible in the program. Alternatively, special character called empty indicator such as #, $ can be placed at that place. Both the insertion and deletion operation perform scanning of the queue elements. While performing deletion operation the element must be deleted physically. It is possible to make its access denied. However, for other computation operation, the element logically deleted but physically existing will be taken into account and will change the result. So, for sure result, the element must be removed.

The elements of the queue can be number, character or any complex object which is a composition of one or more basic data types.

In priority queue every element has been assigned with a priority value called priority. The elements can be inserted or deleted randomly anywhere in the queue. Consider the following points of queue:

1. An element of upper priority is processed prior to an element of lower priority.
2. If two elements have the same priority, they are processed depending on the order they are inserted in the queue, i.e. FIFO.

Figure 3.17. Priority queue



As shown in Fig. 3.17, the element 7 is deleted first. However, the element 1 is nearer to the front as compared to 7, but 7's priority value is nine, which is higher, and hence it is deleted first. Though the queue is based on FIFO technique, the priority queue is not based on FIFO operation firmly. A program on priority queue is provided below for detailed understanding.

APPLICATIONS OF QUEUES

There are various applications of computer science, which are performed using data structure queue. This data structure is usually used in simulation, various features of operating system, multiprogramming platform systems and different type of scheduling algorithm are implemented using queues. Round robin technique is implemented using queues. Printer server routines, various applications software are also based on queue data structure.

3.9.1 Round Robin Algorithm

Round Robin (RR) algorithm is an important scheduling algorithm. It is used especially for the time-sharing system. The circular queue is used to implement such algorithms.

For example, there are N procedures or tasks such as P1, P2, P3 ... P_N. All these tasks are to be executed by the central processing unit (CPU) of the computer system. The execution times of the tasks or processes are different. The tasks are executed in sequence P1, P_2, P_3 and P_N.

In time, sharing mode the tasks are executed one by one. The algorithm forms a small unit of time, say, from 10 to 100 milliseconds for each task. This time is called time slice or time quantum of a task. The CPU executes tasks from P_1 to P_N allocating fixed amount of time for each process. On allocating time to all tasks, CPU resumes the first task. That is, when all tasks are completed, it returns to P_1. In the time-sharing system, if any task is completed before the estimated time, the next task is taken up for execution immediately. Consider the following table:

Tasks	Expiry Time (Units)
P1	10
P2	19
P3	8
P4	5

There are total four tasks and the total time to complete all the tasks would be (10+19+8+5) 42 units. Suppose, the time slice is of 7 minutes. The RR scheduling for the above case would be
In the first pass each task takes seven units of time-task P4 is executed in the first pass whereas task

P1, P2, P3 requires more than 7 units of time. Hence, in the second round task P1 and P3 are executed. At last, P2 is executed.

3.9.2 Simulation

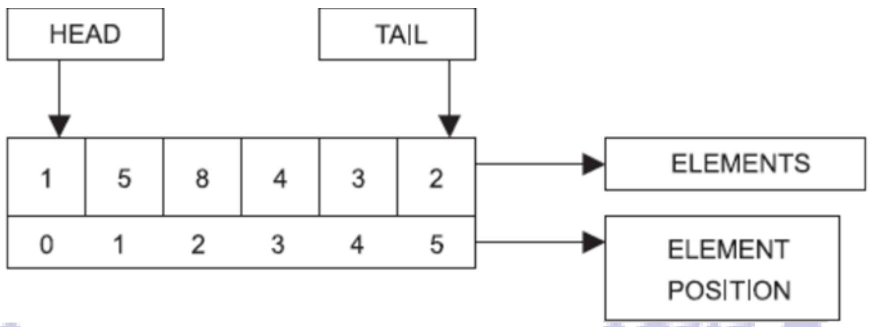
It is an extremely powerful tool used for experimentation purpose. Without performing real experiments simulation permits to take the results and if necessary modification can be done as per expected results. One of the standard applications of queue can be implemented with simulation. Simulation is a method of managing a theoretical illustration of a real life problem with the purpose of understanding the effect of modification, concern factors and implementing some approaches to get reliable solution. The main goal of simulation is to help the user or to guess for obtaining the output of the program after implementing some approaches. It permits the user to make several trials for getting the actual results and planned situations without disturbing the real situation.

There are few disadvantages in the simulation. Lengthy simulation creates execution overhead on computer. For solving a real life problem, various assumptions are made to find out the key solution. If the assumption is straightforward, the outcome will be less reliable. In contrast, if the assumption is having more particulars, the outcome will be reliable and better. If the primary assumptions are wrong, even though, the program source code is given in detail, the guess obtained will be incorrect. In this situation, the program developed becomes futile.

Chapter 4
Linked list

A list is a series of linearly arranged finite elements (numbers) of same type. The data elements are called nodes. The list can be of two types, i.e. basic data type or custom data type. The elements are positioned one after the other and their position numbers appear in sequence. The first element of the list is known as head/root and the last element is known as tail.

Figure 4.1. Static list



As shown in the above-Fig. 4.1-, the element 1 is at head position (0th) and element 2 is at tail position (5th). The element 5 is predecessor of element 8 and 4 is successor. Every element can act as predecessor excluding the first element because it does not have predecessor in the list. The list has following properties:

a. The list can be enlarged or reduced from both the ends.
b. The tail (ending) position of the list depends on how long the list is extended by the user.
c. Various operations such as transverse, insertion and deletion can be performed on the list.
d. The list can be implemented by applying static (array) or dynamic (pointer) implementation.

IMPLEMENTATION OF LIST

There are two methods of implementation of the list: they are static and dynamic.

4.2.1 Static Implementation

Static implementation can be implemented using arrays. It is a very simple method but it has few limitations. Once a size is declared, it cannot be changed during the program. It is also not efficient for memory. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupies the memory space. In both the cases, if we store less arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact number of elements are to be stored.

4.2.2 Dynamic Implementation

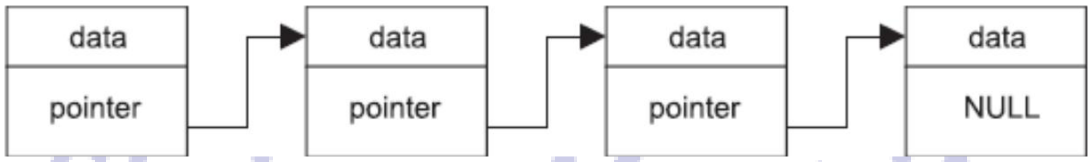
The linked list is a major application of the dynamic implementation and the pointers are used for the implementation. The limitations noticed in static implementation can be removed by using dynamic implementation. The memory is utilized efficiently in this method. Hence, it is superior than the static implementation. With pointers, link does not have any restriction on number of elements at run time.
The list can be stretched like elastic. Memory is allocated only after nodeis added or element is

accepted in the list. Memory is de-allocated whenever node or element is removed. Dynamic memory management policy is used in the implementation of linked list and because of this memory is used resourcefully. In addition to that insertion and deletion of a node can be done easily.

LINKED LIST

A linked list is a dynamic data structure. It is an ideal technique to store data when the user is not aware of the number of elements to be stored. The dynamic implementation of list using pointers is also known as linked list. Each element of the list is called as node. Each element points to the next element. In the linked list a node can be inserted or deleted at any position. Each node of linked list has two components. The first component contains the information or any data field and second part the address of the next node. In other words, the second part holds address of the next element. This pointer points to the next data item. The pointer variable member of the last record of the list is generally assigned a NULL value to indicate the end of the list. Fig. 4.5 indicates the linked list.

Figure 4.5. Linked list



The basic data type in the linked list can be int, float and user defined data types can be created by struct call. . The link structure as well as a pointer of structure type to the next object in the link is observed in it.

IMPORTANT TERMS

We have already discussed in previous sections that a linked list is a non-sequential collection of elements called nodes. These nodes are nothing but objects. These nodes are declared using structure or classes. Every node has two fields and they are:

1. Data field: In this field, the data or values are stored and processed.
2. Link field: This field holds address of the next data element of the list. This address is used to access the successive elements of the list.

In the linked list the ordering of elements is not done by their physical location in the memory but by logical links, which are stored in the link field.

Node

The components or objects which form the list are called nodes.

Null Pointer

The link field of the last record is assigned a NULL value instead of any address. It means NULL pointer not pointing to any element.

External Pointer

It is a pointer to the starting node. The external pointer contains the base, i.e. address of first node. Once a base address is available, its next successive nodes can be accessed.

Em pty List

When there is no node in the list, it is called as empty list. If the external pointer were assigned a value NULL, the list would be empty.

Types of Linked List

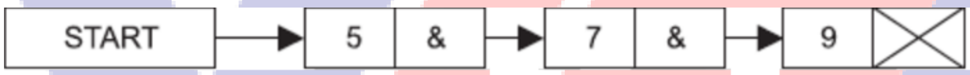
The linked lists are classified in the following types:

SINGLY LINKED LIST

Recall that linked list is a dynamic data structure with ability to expand and shrink as per the program requirement. The singly linked list is easy and straightforward data structure as compared to other structures. By changing the link position other type of linked list such as circular, doubly linked list can be formed. For creating linked list the structure is defined as follows,

```
struct node
{
 int number;
 struct node *p;
};
```

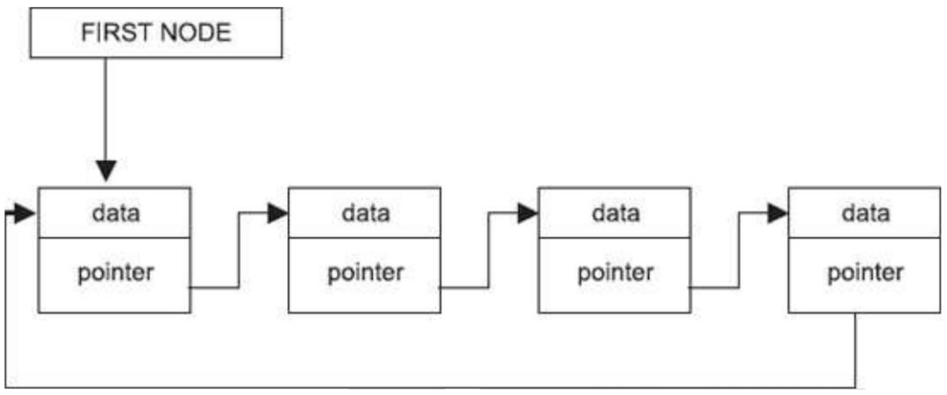
The above structure is used to implement the linked list. In the number, variable entered numbers are stored. The second member is pointer to the same structure. The pointer *p points to the same structure. Here, though the declaration of struct node has not been completed, the pointer declaration of the same structure type is permitted by the compiler. However, the variable declaration is not allowed. This is because, the pointers are dynamic in nature whereas variables are formed by early binding. The declaration of objects inside the struct leads to preparation of very complex data structure.
This concept is called object composition and its detailed discussion is out of the scope of this book. We are familiar with the array and we know the importance of base address. Once a base address is obtained, successive elements can also be accessed. In the linked list, list can be created with or without header node. The head holds the starting address.
Figure 4.8. Singly linked list



CIRCULAR LINKED LIST

In circular linked list the last node points to the header node. The linear link list can be converted to circular linked list by linking the last node to the first node. The last node of the linear linked list holds NULL pointer, which indicates the end of linked list but performance of linked list can be advanced with minor adjustment in the linear linked list. Instead of placing the NULL pointer, the address of the first node can be given to the last node, such a list is called circular linked list (as shown in Fig. 4.22).

Figure 4.22. Circular linked list



The circular linked list is more helpful as compared to singly linked list. In the circular linked list, all the nodes of the list are accessible from the given node. Once a node is accessed, by traversing all the nodes can be accessed in succession.

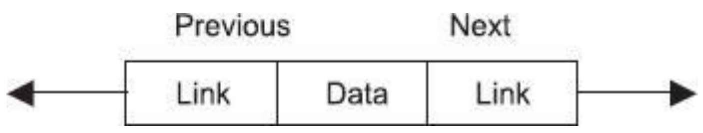
In this type of list, the deletion operation is very easy. To delete an element from the singly linked list, it is essential to obtain the address of the first node of the list. For example, we want to delete the element, say 5, which exists in the middle of the list. To remove the element five, we need to find predecessor of five. Obviously, a particular element can be searched using searching process in which all elements are visited and compared. In the circular linked list, no such process is needed. The address of predecessor can be found from the given element itself. In addition, the operation splitting and concatenation of the list (discussed later) are easier. Though the circular linked list has advantages over linear linked list, it also has some limitations.

This list does not have first and last node. While traversing the linked list, due to lack of NULL pointer, there may be a possibility to get into an infinite loop. Thus, in the circular linked list it is necessary to identify the end of the list. This can be done by setting up the first and last node by convention. We detect the first node by creating the list head, which holds the address of the first node. The list head is also called as external pointer. We can also keep a counter, which is incremented when nodes are created and end of the node can be detected. The Fig. 4.23 gives an example of circular linked list with header.

DOUBLY LINKED LIST

The singly linked list and circular linked list contain only one pointer field. Every node holds an address of next node. Thus, the singly linked list can traverse only in one direction, i.e. forward. This limitation can be overcome by doubly linked list. Each node of the doubly linked list has two pointer fields and holds the address of predecessor and successor elements. These pointers enable bi-directional traversing, i.e. traversing the list in backward and forward direction. In several applications, it is very essential to traverse the list in backward direction. The pointer pointing to the predecessor node is called left link and pointer pointing to successor is called right link. A list having such type of node is called doubly linked list. The pointer field of the first and last node holds NULL value, i.e. the beginning and end of the list can be identified by NULL value. The structure of the node is as shown in Fig. 4.30.

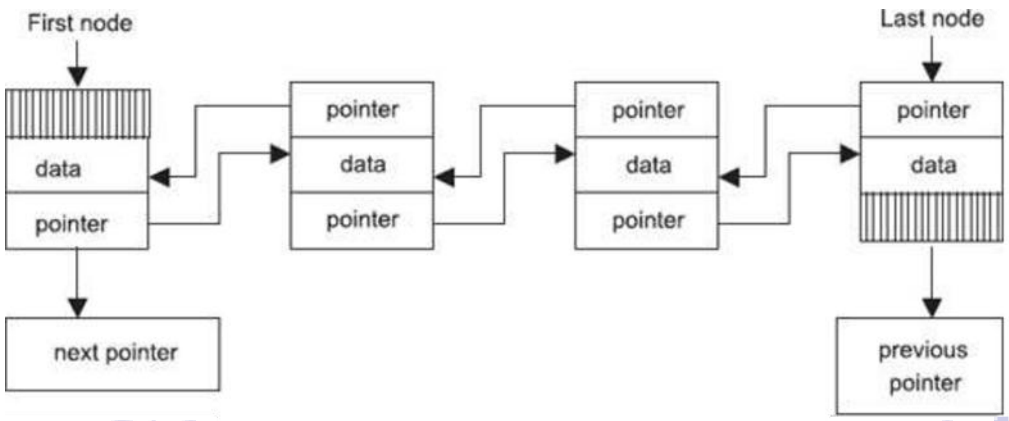
Figure 4.30. Structure of node



The structure of node would be as follows:

```
struct node
{
  int number;
  struct node *llink
  struct node *rlink;
}
```

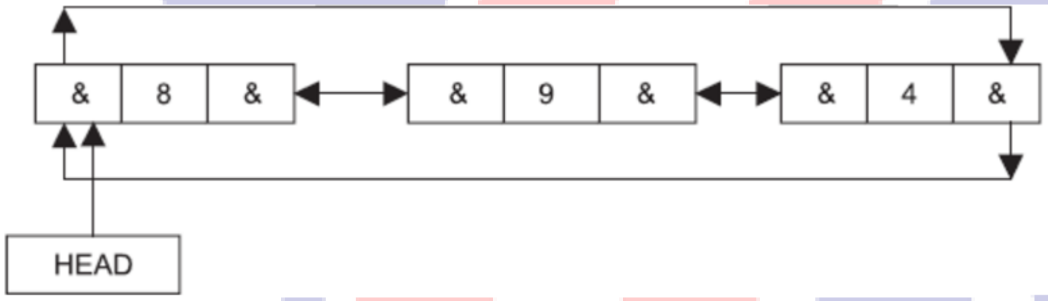
the above structure can be represented by using the Fig. 4.31.
Figure 4.31. Doubly linked list



CIRCULAR DOUBLY LINKED LIST

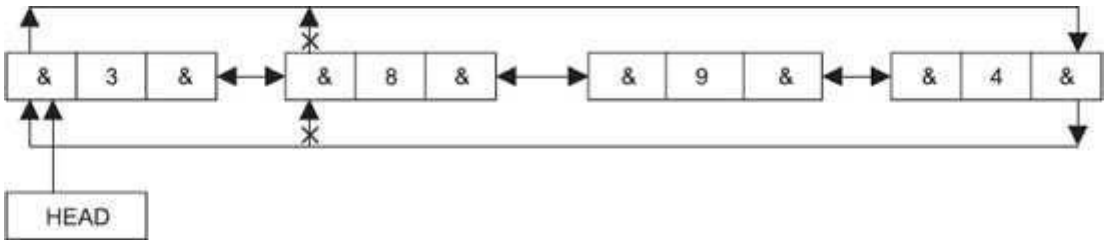
A circular doubly linked list has both successor and predecessor pointers. Using the circular fashioned doubly linked list the insertion and deletion operation, which are little complicated in the previous types of linked list are easily performed. Consider the following Fig. 4.36:

Figure 4.36. Circular doubly linked list



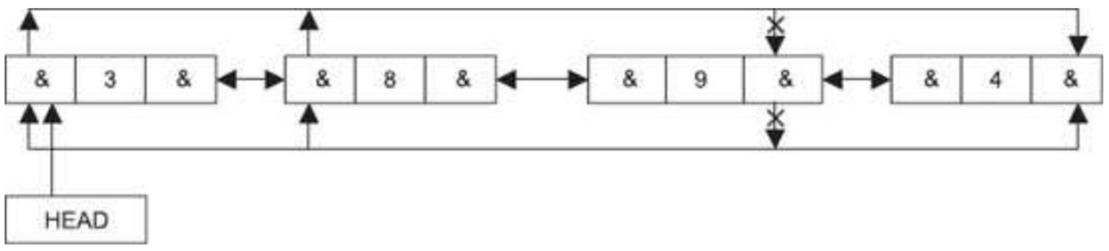
4.27.1 Insertion and Deletion Operation

The insertion operation is similar to what we have already learnt in previous types. The Only difference is the way we link the pointer fields. Consider Fig. 637.
Figure 4.37. Inserting a node at the beginning



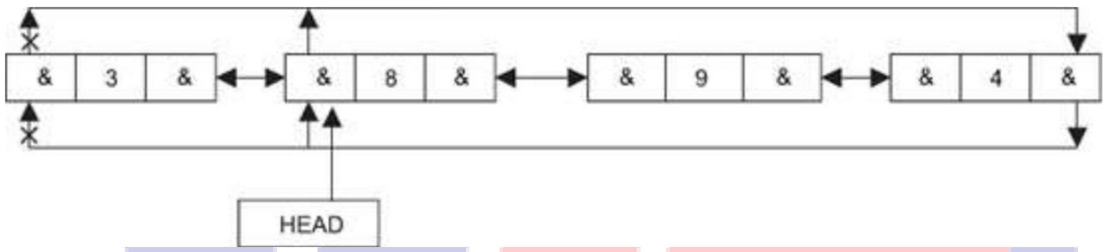
The × indicates that the previous links are destroyed. The pointer links from ex-first node are removed and linked to new inserted node at the beginning. The element 8 was previous first node and 3 is the new node inserted and becomes first node now after inserting it at the beginning.

Figure 4.38. Inserting the node at the end



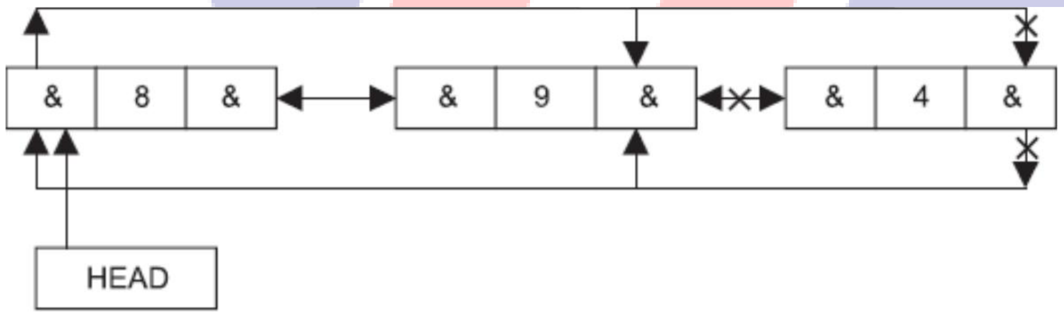
The × indicates, that the previous links are destroyed. The pointer links from ex-last node are removed and linked to new inserted node at the end. The address of last node is given to first node to form circular list. The node 9 was previously the last node but after insertion of the node at the end, newly inserted node is the last node. Fig. 4.39 shows the deletion of the node at beginning.

Figure 4.39. Deleting a node from the beginning



The × indicates that the previous links are destroyed. After deletion of the first node, the second node becomes first node. The pointer head also points to the newly appeared first node. Thus, when a node from the beginning is removed, the node followed by it will become the head node (first node). Accordingly, pointer adjustment is performed in the real application.

Figure 4.40. Deletion of the node at the end



The × as usual is the symbol destroying previous links. When the last node is removed, one node will be the last node and its links are established with the first node. The removal operation is shown in Fig. 4.40.

MEMORY ALLOCATION AND DE-ALLOCATION

Malloc (): This library function is used to allocate memory space in bytes to the variable. The function reserves bytes of requested size and returns the base address to pointer variable. The prototype of this function is declared in the header file alloc.h and stdlib.h. One of the header files must be included in the header. The syntax of the malloc() function is as follows:

Syntax: pnt=(data type*) malloc(size);

Here, pnt is a pointer.

Example: pnt =(int *) malloc(20);

In the above statement, 20 bytes are allocated to integer pointer pnt. In addition, there are other various memory allocation functions and its complete description is out of the scope of this book. free(): This function is used to release the memory allocated by the malloc () function.

Syntax: free(pointer variable)

Example: free(pnt)

The above statement releases the memory allocated to pointer pnt.

OPERATIONS ON LINKED LISTS

The following primitive operations can be performed with linked list:

1. Creation
2. Display
   a. Ascending
   b. Descending
3. Traversing
4. Insertion
   a. At beginning
   b. Before or after specified position
5. Searching
6. Concatenation
7. Merging

Creation

The linked list creation operation involves allocation of structure size memory to pointer of the same structure. The structure must have a member which points recursively to the same structure. In this operation, constituent node is created and it is linked to the link field of preceding node.

Traversing

It is the procedure of passing through (visiting) all the nodes of the linked list from starting to end. When any given record is to be searched in the linked list, traversing is applied. When the given element is found, the operation can be terminated or continued for next search. The traversing is a common procedure and it is necessary because operations like insertion, deletion, listing cannot be carried out without traversing linked list.

Display

The operation in which data field of every node is accessed and displayed on the screen. In the display operation from beginning to end, link field of every node is accessed which contains address of next node. The data of that field is displayed. When NULL is detected the operation ends. Each node points to the next node and this recursion fashion enables the pointer to reach the successive elements.

The above three operations are common and every program involves these operations. Hence, separate program is not given.

Searching

The searching is a process, in which a given element is compared with all the linked list elements. The if statement is placed and it checks entire list elements with the given element. When an element is found it is displayed.

Besides the above operations additional operations such as concatenation and merging of list can be done.

APPLICATIONS OF LINKED LIST-

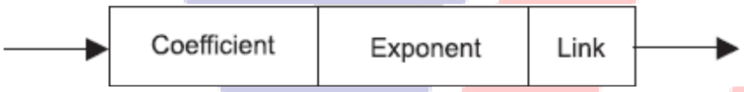
The most useful linear data structure is linked list. This section introduces you to a few applications of-linked lists which are useful in computer science.-

1 Polynomial Manipulation-

A polynomial can be represented and manipulated using linear link list. Various applications on-polynomials can be implemented with linked lists. We perform various operations such as addition, multiplication with polynomials. To get better proficiency in processing, each polynomial is stored in decreasing order. These arrangements of polynomial in series allow easy operation on them. Actually, two polynomials can be added by checking each term. The prior comparison can be easily done to add corresponding terms of two polynomials.-

A polynomial is represented with various terms containing coefficients and exponents. In other words, a polynomial can be expressed with different terms, each of which comprises of coefficients and exponents. The structure of a node of linked list for polynomial implementation will be as follows. Its pictorial representation is shown in Fig. 4.41.-

Figure 4.41. A term of a polynomial-



The coefficient field contains the value of coefficient of the term. Similarly, the exponent field contains the value of exponent. As usual, the link field points to the term (next node). The structure for the above node would be as follows:

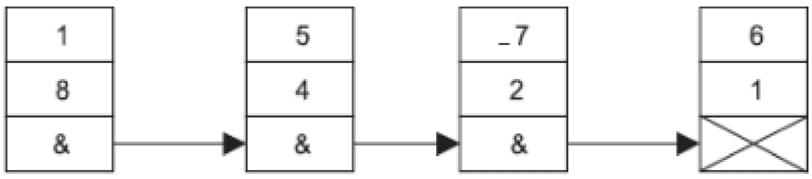
```
struct poly
{
  double coeff;
  int exp;
  struct poly *next;
};
```

Consider a polynomial

$P = P^8 + 5P^4 - 7P^2 + 6P$

In this equation 1,5,7 and 6 are coefficients and exponents are 8,4,2 and 1. The number of nodes required would be the same as the number of terms in the polynomial. There are four terms in this polynomial hence it is represented with four nodes.

Figure 4.42. Polynomial representation



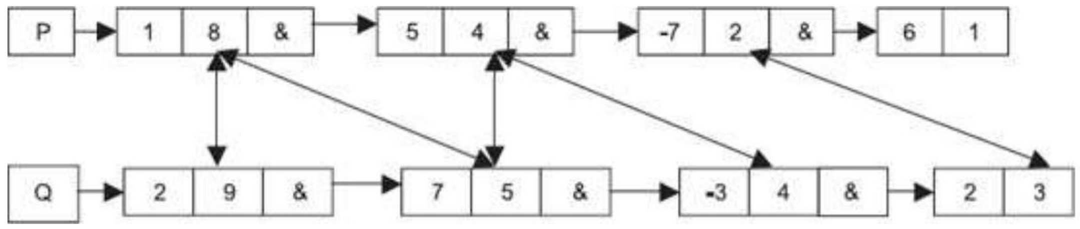
The top of every node represents coefficients of the polynomial, exponents are at the centre and the pointers are (next) at the bottom. The terms are stored in order of descending exponent in the linked list. It is assumed that no two terms have the similar exponents. Fig. 4.42 shows the polynomial. Consider the following equations,

$P = P_P{}^8P + 5P_P{}^4P - 7P_P{}^2P + 6P$

$Q = 2P_P{}^9P + 7P_P{}^5P - 3P_P{}^4 + 2P^3$

The representation of the above two polynomials can be shown in Fig. 4.43.

Figure 4.43. Addition of two polynomials



If the term is shown without coefficient then the coefficient is assumed to be 1. In case the term is without variable, the coefficient is zero. Such terms are not required to be stored in the memory. The arrow in the figure indicates that the two exponents are compared. Where the link of arrow is disconnected, exponent of both the terms is same. The term, which has no touch of any arrow, means it is the last node and inserted in the resulting list at the end. The arrow indicates two terms, which are compared systematically from left to right.

Table 4.1. Exponent

Exponent comparison from list p and Q	List R (Inserted exponent)	Smaller exponent
8<9	9	8 is carried forward
8>5	8	5 is carried forward
5>4	5	4 is carried forward
4= =4	4	No term is carried
	(Sum of coefficient is taken)	
2<3	3	2 is carried forward
1 is taken (last node of P)	1	End of linked list

Traverse the list p and Q. Compare the corresponding terms of list p and Q. In case one node has larger exponent value than the other then insert the larger exponent node in the third list and forward the pointer to next node of the list whose current term is inserted in the third list. The pointer of the list whose exponent is smaller will not be forwarded. The pointer of the lists forwarded only when the current nodes from the lists are inserted into the third list. Table 4.1 shows these operations.

If exponents are equal, add the coefficients and insert their addition in the third list. In this step, exponents from both the expressions are same, move the pointer to next node in both the list p and Q. Repeat the same process until one list is scanned completely.

expo(P) = expo(Q).

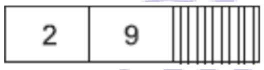
The possible conditions can be stated as follows:

1. If exponent of list p is greater than corresponding exponent of list Q, insert the term of p into the list R and forward the pointer in list p to access the next term. In this case, the pointer in the list Q will point to the same term (will not be forwarded).
2. If exponent of list Q is greater than exponent P, the term from Q is inserted in the list R. The pointer of list Q will be forwarded to point to next node. The pointer in the list p will remain at the same node.
3. If exponents of both nodes are equal, addition of coefficients is taken and inserted in the list R. In this case, pointers in both the lists are forwarded to access the next term.

The steps involed,

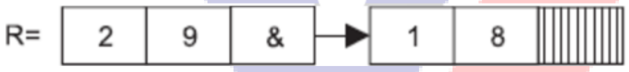
1. Traverse the two lists (P) and (Q) and inspect all the elements.
2. Compare corresponding exponents p and Q of two polynomials. The first terms of the two polynomials contain exponents 8 and 9, respectively. Exponent of first term of first polynomial is smaller than the second one. Hence 8<9. Here, the first exponent of list Q is greater than the first exponent of list P. Hence, the term having larger exponent will be inserted in the list R. The list R initially looks like as shown in Fig. 4.44(a).

Figure 4.44(a).



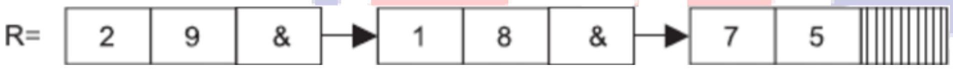
Next, check the next term of the list Q. Compare it with the present term (exponent) of list P. The next node from p will be taken when current node is inserted in the list R because 8>5. Here, the exponent of current node of list p is greater that list Q. Hence, current term (node) of list p will be inserted in the list R and the list R becomes as Fig. 4.44 (b).

Figure 4.44(b).



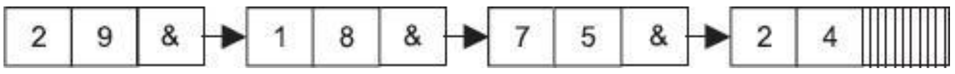
After moving to next node in list P, the exponent is 4, and exponent of Q is 5. Compare 4 with 5. Of course 4<5. Here, the term of list Q is greater than term of P. Therefore, the term of list, Q (7,5) will be inserted to list R. The list R becomes Fig. 4.44 (c).

Figure 4.44(c).



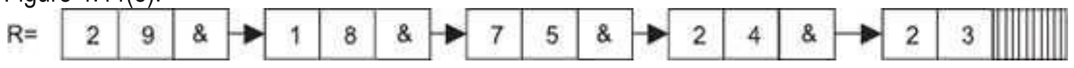
In this step a node from list Q is inserted and therefore, the pointer in the list Q will be forwarded and point to the term (–3,4) and from list p we have the current node (5,4). Compare exponent of these two terms. The condition observed here is 4==4. Here, exponents of both the terms are equal. Therefore, addition of coefficients is taken and result is inserted in the list R. The addition is 2 (5–3). The list R becomes as the Fig. 4.44 (d).

Figure 4.44(d).



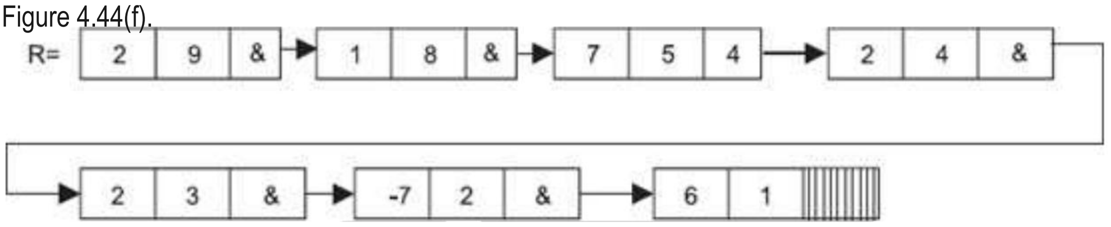
Move forward the pointers to next nodes in both the lists, since, the previous terms were having same exponents. The next comparison is 2<3. Here, the exponent of current node of list Q is greater than of P. The node from Q will be inserted to list R. The list R will be shown as Fig. 4.44(e).

Figure 4.44(e).



The list Q is completely scanned and reached to end. The remaining node from the list p will be inserted to list R. The list R is as Fig. 4.44(f).

Figure 4.44(f).


R=
2
9
&
1
8
&
7
5
4
2
4
&
2
3
&
-7
2
&
6
1