Example

$gcd(24, 30) = gcd(30, 24) = 6$
$gcd(5, 7) = 1$
$gcd(0, 9) = 9$

Some of the important properties of *gcd* are:

(1) $gcd(0, 0) = 0$
(2). $gcd(a, b) =$ an integer between 1 and $min(|a|, |b|)$, for $a > 0$
(3) $gcd(a, b) = gcd\ (b, a)$
(4) $gcd(a, b) = gcd\ (-a, b)$
(5) $gcd(a, b) = gcd(|a|, |b|)$
(6) $gcd(a, 0) = |a|$
(7) $gcd(a\ n, b\ n) = n\ gcd(a, b)$

4.5.1 Euclid's GCD Algorithm: *gcd*(a, b)

In this section, we use Euclid's algorithm to compute the *gcd* of two integers efficiently. We shall assume that only positive integers for *a* and *b* are to be given. Euclid's algorithm is primarily based on the following recurrence relation:

$$gcd\ (a, b) = gcd\ (b, a \textbf{ mod } b) \qquad ....(4.3)$$

That is,
$$gcd\ (a, b) = \begin{cases} a, & \text{if } b = 0 \\ gcd\ (b, a \textbf{ mod } b), & \text{otherwise} \end{cases} \qquad ....(4.4)$$

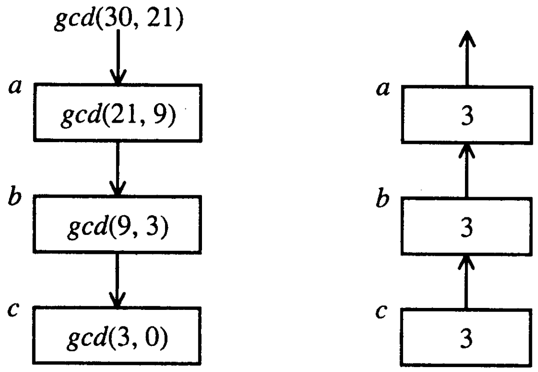
C Implementation

In C language, fortunately, we have the **mod** operator (%) that can be directly used to compute the *gcd* recursively. It is shown in Program 4.3.

Program 4.3
gcd of two numbers

```
int gcd (int a, int b)
{
    if (b == 0) return a;
    else  return gcd(b, a % b);
}
```

The terminating condition for the `gcd()` function is when `b` becomes 0, the value of `a` is to be returned to the calling program. We shall restrict to show the execution of `gcd()` by the call tree structure as shown in Figure 4.3 for `a` = 30 and `b` = 21.



(a) Call sequence **(b) Return path**

Fig. 4.3 gcd(30, 21) – Call tree

With a = 30 and b = 21, the first recursive call is involved (box-*a*) with b = 9. Since, b ≠ 0, a call to gcd() with a = 9 and b = 3 (box-*b*) is invoked. Now, b is not reached its terminating condition, so once again gcd() is called with b = 3, (box-*c*). Since, 9 % 3 = 0, gcd(3, 0) returns a value of 3 to box-*c*. We do not have any other work to do after getting the final value, unlike factorial function and therefore this value is carried to box-*b* and then to box-*a*. Finally, the result of gcd(30, 21) = 3 is returned of the calling routine.

4.6 TOWERS OF HANOI

In a monastery in *Benares* India there are three diamond towers holding 64 disks made of gold. The disks are each of a different size and have holes in the middle so that they slide over the towers and sit in a stack. When they started, 1500 years ago, all 64 disks were stacked on the first tower arranged with the largest on the bottom and the smallest on the top. The monk's job is to move all of the disks to the third tower following these three rules.

1. Each disk sits over a tower except when it is being moved.
2. No disk may ever rest on a smaller disk.
3. Only one disk at a time may be moved.

The monks are very good at this job after all this time and can move about 1 disk per second (because disks are very heavy). When they complete their task, THE WORLD WILL END! The question is, how much time do we have left? This is called as **Towers of Hanoi Problem** and Figure 4.4 shows a sample of it. The three towers (also called as **Pegs**) are named as A, B and C.

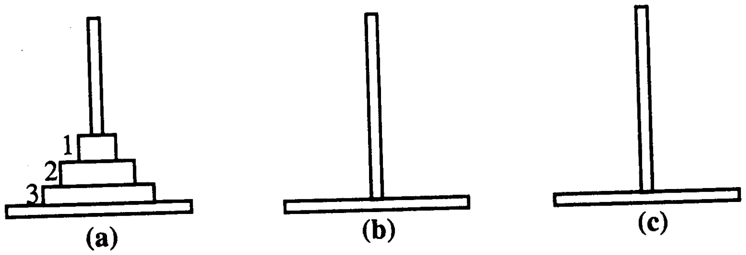


Fig 4.4 Towers of Hanoi problem

The Objective

The previous paragraph gave a bit of history behind the Towers of Hanoi Problem. Let us now formally state what we are supposed to do?

In this problem, we are given *n* disks and three pegs. The disks are initially stacked on peg A in decreasing order of size from bottom to top. We are to move all *n* disks from peg A to peg B, satisfying the three constraints stated already. You may use peg C as a spare or auxiliary peg.

The Method

A very elegant solution results from the use of recursion. However, we start with mathematical induction applied to this problem and derive a recurrence relation.

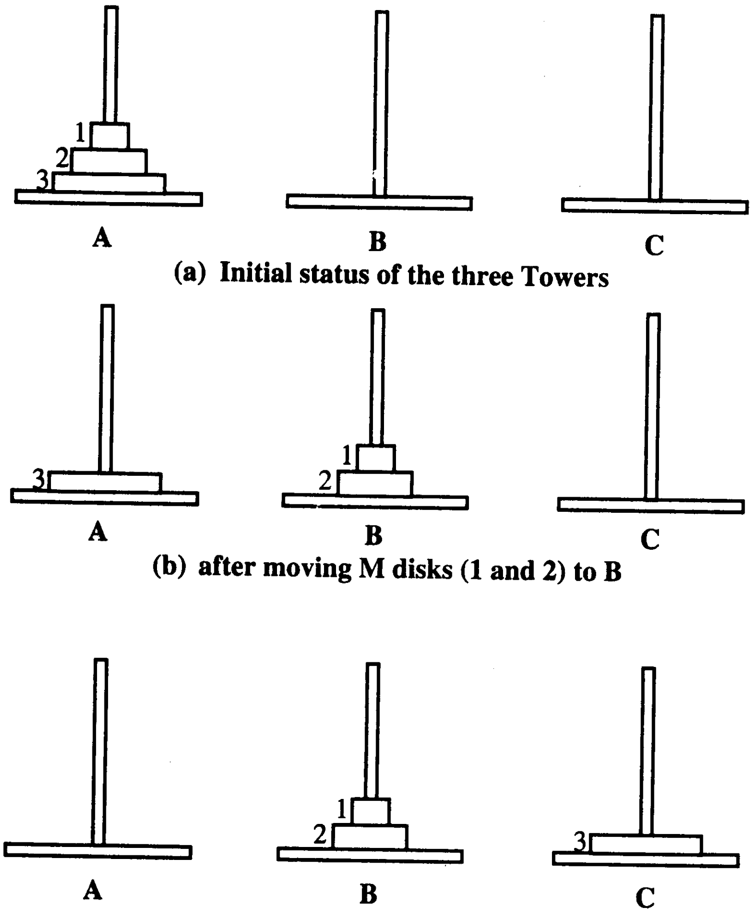
The induction method applied to our problem says that if you can solve it for one disk and if, assuming you can solve the problem for M disks, you can tell how to solve it for M + 1 disks. Then, you can solve it for any number of disks.

Solving this problem for one disk is very simple- just move it from A to B. Given a stack of M + 1 disks, and knowing how to move M disks, can we figure out how to solve the problem? First notice that the top M disks of this pile would not know about the bottom disk at all – since it is the largest disk, if it were alone on the tower, any other disk could move on top of it just like it wasn't there. Let's assume that we have M + 1 disks on peg A. By induction, we assume that the top M disks can be moved from A to B. We are almost through with our task. Because, now we have peg A left with the largest disk, peg B with the other M disks and peg C empty. Now, move the largest disk from A to C (see Figure 4.5).

Next, since we have a method that moves M pegs from A to B, we can just re-label the pegs appropriately (see Figure 4.6). That is,

- A has the M pegs,
- B is the one with the large disk on it that we want to move the other M to, and
- C is the spare peg.

Now apply our method again for moving M pegs from A to B once again and we are done!

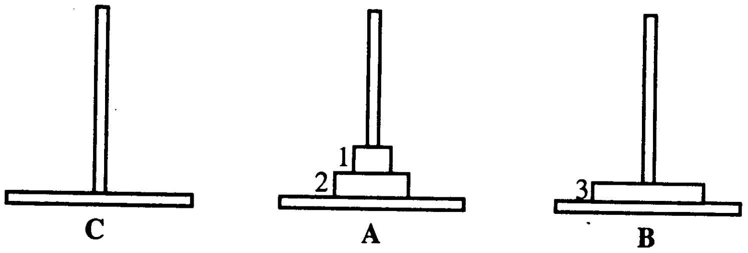


(a) Initial status of the three Towers

(b) after moving M disks (1 and 2) to B

(c) after moving the largest disk to peg C

Fig. 4.5



(a) A is re-labeled as C, B is re-labeled as A, C is re-labeled as B

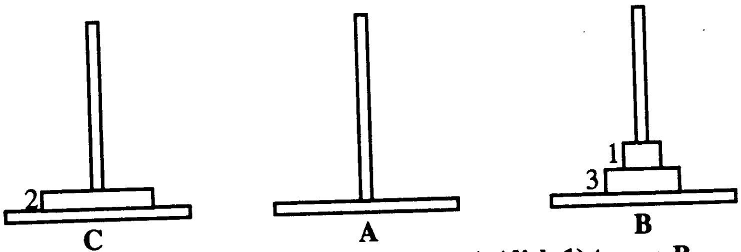


Fig. 4.6 (b) After M disks on peg A (disk 1) to peg B and moving the current largest disk 2 to C

Using the idea of recursion, the program for Towers of Hanoi can readily be written and is shown in Program 4.4.

Program 4.4
Towers of Hanoi

```
void TowersOfHanoi(int n,char src,char dst,
                          char aux,int *moves)
{
     if (n > 0)
     {
          TowersOfHanoi(n-1, src, aux, dst, moves);
          printf("Move disk %d from peg %c
                              to peg %c\n", n, src, dst);
          (*moves)++;
          TowersOfHanoi(n-1, aux, dst, src, moves);
     }
}
```

Note that the function `TowersOfHanoi()` should be invoked as,

```
     TowersofHanoi(n, 'A', 'B', 'C');
```

When, n = 3, the sample run of the program generates the following result:

Example Run

```
Enter the number of disks: 3
Move disk 1 from peg A to peg B
Move disk 2 from peg A to peg C
Move disk 1 from peg B to peg C
Move disk 3 from peg A to peg B
Move disk 1 from peg C to peg A
Move disk 2 from peg C to peg B
Move disk 1 from peg A to peg B
No. of moves = 7
```

The call tree structure for Towers of Hanoi execution is very difficult compared to other recursively solved problems. However, the call tree is written (see Figure 4.7) in such a way that the details of the pegs and value of *n* are shown in each box. Boxes represent the recursive invocation of `TowersofHanoi()` function.

The nodes which are numbered – non-zero-leaf nodes (nodes at the bottom) correspond to `printf()` and it actually prints the disk number plus `src` and `dst` pegs. The order corresponds to the same way as that of as the execution sequence as shown above.

Assume,

- n = 3 (disks are numbered from small to big)
- A, B and C are the towers in which the source peg is A, destination peg is B and spare peg is C.
- The dotted arrows indicate the recursive calls and returns.
- The syntax used in the boxes are (*n*, *src*, *dst*, *aux*)

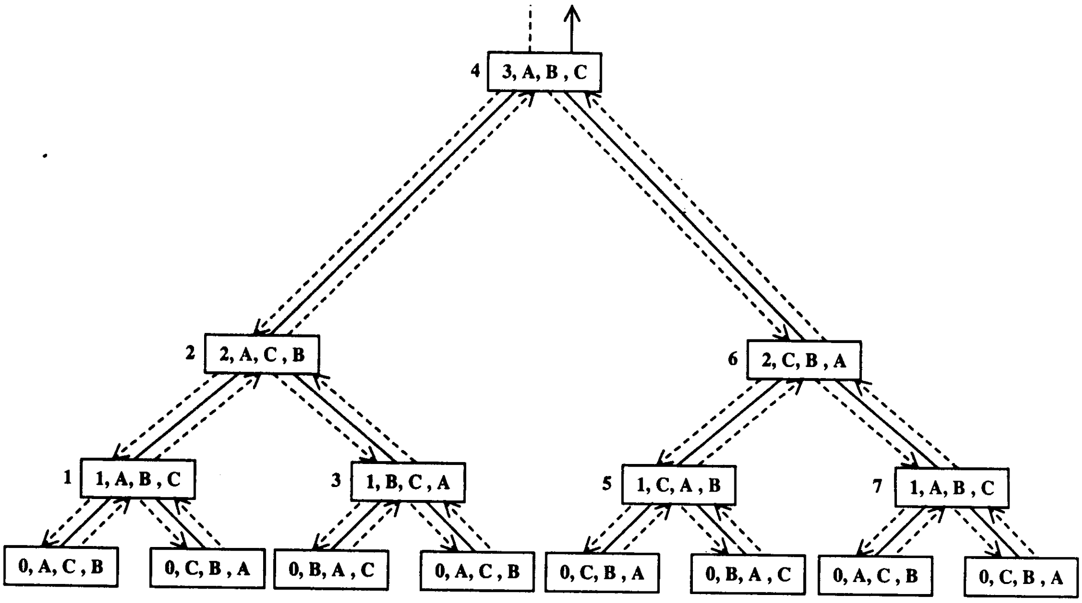


Figure 4.7 Call tree for Towers-Of-Hanoi with n = 3

As mentioned already, `printf()` will be executed when n ≠ 0 in Figure 4.7 to find the number of moves, we must calculate the number of internal nodes (non-leaf nodes).

$$\left.\begin{array}{l}\text{The maximum number of internal}\\ \text{nodes in a general tree is}\end{array}\right\} = \frac{N^{h+1}-1}{N-1} \qquad ....(4.4)$$

Where, N – number of branches and h – height of the tree

The Towers of Hanoi tree is a binary tree and hence N = 2. Then Equation 4.4 reduces to,

$$= \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1 \qquad .....(4.5)$$

The height of the tree is same as the number of disks,

Therefore, the number of moves = $2^n - 1$

For, $n = 3$, We have, $= 2^3 - 1 = 7$ moves

Similarly, for 64 disks, the number of moves $= 2^{64} - 1$.

4.7 BINOMIAL CO-EFFICIENT (BC)

A K-combination of an n-set S is simply a k-subset of S. There are six 2-combinations of the 4-set $\{a, b, c, d\}$.

ab, ac, ad, bc, bd, cd.

We can construct a k-combination of an n-set by choosing k distinct (different) elements from the n-set.

The number of k- permutations of an n-set is

$$n(n - 1)(n - 2) \ldots (n - k + 1) = \frac{n!}{(n-k)!} \qquad ...(4.6)$$

The number of k-combinations of an n-set is the number of k-permutations divided by $k!$. Thus from Equation (4.6), we have,

$$= \frac{n!}{k!(n-k)!}$$

We use the notation $\binom{n}{k}$ (read ***n* choose *k***) to denote the number of k-combination of an n-set from Equation (4.6), we have

$$\binom{n}{k} = \frac{n!}{k!(n-k)!} \qquad ....(4.7)$$

The number generated for different values of n and k are called **as binomial coefficients**. The recurrence relation for Binomial Coefficient (BC) is given as, is divided into smaller and smaller problems so that it can be handled more easily and naturally. Iteration also can be used for divide-and-conquer, but it may not be as easier than recursive technique.

4.9 ADDITIONAL EXAMPLES

Though many examples were shown in this chapter on recursive program design, some new examples related to data structures are dealt in this section.

Example-1 To find the product of n *natural numbers.*

Finding the product of *n* numbers (assumed as integers) that are stored in an array *a* is the first example we consider in this section. For simplicity, assume the array to contain the elements from 1st location instead of 0th location. That is,

	1	2	3	4
a[] =	7	2	1	5

Result = a[1] * a[2] * a[3] * a[4]
= 70

The function `Prod()` is shown in Program 4.6. The function is invoked as `Prod(a, n), n > 0`.

Program 4.6
Product of n natural numbers

```
long Prod (int a[], int n)
{
    if (n <= 0) return 0; /* illegal values for n */
    if (n == 1)
          return a[1];
    else
             return a[n] * Prod(a, n-1);
}
```

In this program a is the array and n is the size of the array. We should think recursively so that when n = 1, we simply return the contents of a [1].

That is, `Prod(a, 1) =return(1 * a[1]);`

When, n > 1, the same task is done again.

```
Prod(a, 2) = return(a[2] * Prod(a, 1));
```

Since, `Prod(a, 1)` is already known, it would return `a[2] * a[1] * 1` which is what we expect. Continuing in the same manner, for n numbers, we have the Program 4.6.

Example

a[ ] = [7, 2, 1, 5], $n = 4$
Prod(a, 4)= $a[4]$ * *Prod*(a, 3)
= $a[4]$ * $a[3]$ * *Prod*(a, 2)
= $a[4]$ * $a[3]$ * $a[2]$ * *Prod*(a, 1)
= $a[4]$ * $a[3]$ * $a[2]$ * $a[1]$ * 1
= 70.

Example-2 To find Maximum element

The designing of `Max()` function is same as `Prod()` with a difference that in each recursive call the current array element and the current maximum (found so far) should be compared. Next, the bigger one is carried to the previous state and so on until the main program calling is reached. The array elements of `a` are assumed to be integers with the starting position as 1 and the size is `n`. The C code for finding the maximum element in an given array a is shown in Program 4.7.

Program 4.7
Maximum in a[1:n]

```
int Max (int a[], int n)
{
        int y;
    if (n <= 0) return 0; /* illegal values for n */
    if (n == 1)
          return a[1];
    else
    {
          y = Max(a, n-1); /* save current max */
          if (a[n] >= y)
               return a[n];
          else
               return y;
    }
}
```

Perhaps the validity of the code is very simple by writing call tree and is shown in Figure 4.9.
a[] = [15, 16, 11, 17], $n = 4$

The function `int Check_Ascending (const int *a, int n);`
// a is an array and n is the number of elements (n > 0).

Check_Ascending (*a*, *n*) = 1, if *n* == 1
Check_Ascending (*a*, *n*) = 1, if *n* > 1 and *a*[0] < *a*[1] < ... < *a*[*n*-1]
= 0, otherwise.

Our solution must rely on recursion and there may not be any loops in the function definition. Here is one way to solve the problem by first solving a simpler problem of the same kind:
If there is only one element, the result is 1. That's the escape condition. Otherwise, the result is 1 if and only if two things are true:

(1) `a[0] < a[1]`; and

(2) Elements `a[1]` to `a[n-1]` are in ascending order.

Notice that checking (2) is a simpler problem of the same kind. This can be coded in C as in Program 4.10.

Program 4.10
To find for ascending order

```
int Check_Ascending(const int *a, int n)
{
    if (n < 0)  return 0;
    if (n == 1) return 1;
    else
         return (a[0] < a[1]
                && Check_Ascending(a+1, n-1));
}
```

When n value is 1, the array is already in ascending order and there is no need to invoke recursive calls. However, when n is greater than 1, a pair of elements (first two elements of sub-array) is checked for ascending order. As explained earlier, the reader can check the working of this program easily.

4.10 SUMMARY

- **Recursion** is a process by which we define some thing in terms of itself. In Latin, **re** - means **back** and **currere** means **to run**. A procedure or function that is run over and over for a definite number of times is recursion.
- Problems like Factorial, GCD, Fibonacci, etc. - use recursive technique to solve.
- The recurrence relation for **factorial** of a positive of number n is defined as,

$$n! = \begin{cases} 1 & n = 0 \\ n * (n-1) & n > 0 \end{cases}$$

- The recurrence relation for the nth Fibonacci number can be written as,

$$fib(n) = \begin{cases} 1, \text{if } n = 0 \\ 1, \text{if } n = 1 \\ fib(n-1) + fib(n-2), if\ n > 1 \end{cases}$$

- The recurrence relation for GCD of two numbers is,

$$gcd\,(a, b) = \begin{array}{ll} a, & \text{if } b = 0 \\ gcd\,(b, a \textbf{ mod } b), & \text{otherwise} \end{array}$$

The recurrence relation for Binomial Coefficient is,

$$\text{BC}\,(n, k) = \begin{cases} \text{BC}(n, 0) = 1, & \text{if } k = 0 \\ \text{BC}(n, n) = 1, & \text{if } k = n \\ \text{BC}(n - 1, k) + \text{BC}(n - 1, k - 1), & \text{if } n > k > 0 \end{cases}$$

- The iterative method is efficient than recursive approach. However, certain problems like Towers of Hanoi may be solved very easily based on recursive method.

```
    return f(x, y/2);
}
```

Find f(4,3).

4.9 Trace the execution of the following:
(a) Fact(6);

(b) Fib(9);

(c) GCD(7, 19);

(d) ToersOfHanoi(6);

(e) BC(6, 10);

(f) Power(5, 3);

(g) Reverse("Bangalore");

4.10 Write a C program to find the odd numbers in a given array. Employ recursive approach to solve this problem.

4.11 Repeat the problem 4.10, but now find the even numbers only.

Chapte 5

Queues

5.1 INTRODUCTION

The study of queues helps in solving many problems in data structures and other areas. **Queue** is also a linear list in which elements are inserted and deleted from both the ends (unlike stack where deletion and insertion takes place at the same end).

An elementary discussion on definitions of queues through ADT, special types of queues, applications of queues appear in this chapter. More precisely, we develop algorithms and C code for the following topics:

- Ordinary queues
- Circular queues
- Priority queues (**apq** and **dpq**)
- Double ended queues (**deque**)

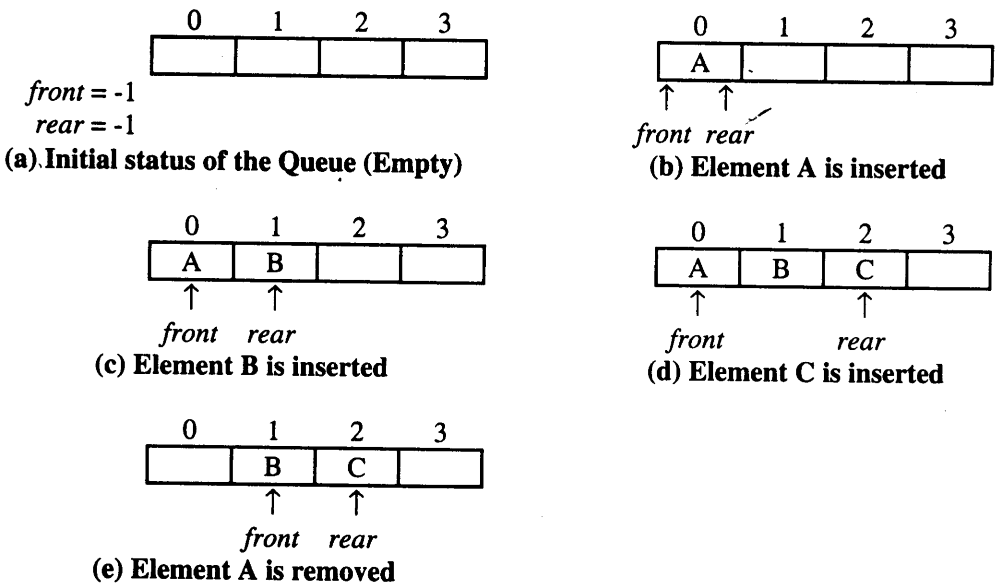
All the above mentioned queues will be implemented using arrays.

5.2 DEFINITION (ADT SPECIFICATION)

A **queue** is a linear list in which additions and deletions takes place at different ends. The end at which insertions take place is called the **rear**, and at which deletions happen is called the **front**.

In our examples, we always assume that the right side is the *rear* end and the left is the *front* end. This assumption is only for convenience and not mandatory.

The moment we say a queue, it is easy to understand that it is same as queues that we see in our day to day life; people standing in milk booth, cinema theatre, banks, ATMs, passport office, consulate offices, etc. The queue policy is that the service is provided to people who have arrived first. In other words, queue operates on a **First-In-First-Out (FIFO)** manner. The operations insertion and deletion do not happen at the same end. For instance, insertion occurs in *rear* and deletion occurs in the *front*. Let us understand the working of a queue with the help of a diagram shown in Figure 5.1.



(a) Initial status of the Queue (Empty)

(b) Element A is inserted

(c) Element B is inserted

(d) Element C is inserted

(e) Element A is removed

Fig. 5.1 Queue addition and deletion

When no elements are in the queue, it is called as *empty* state (shown in figure 5.1(a)). Notice that, two pointers *front* and *rear* are maintained for queue operations (compare this with *Top* of stack).

The elements are added into queue based upon the *rear* pointer and deletion takes place based on the *front* pointer. The *front* pointer remembers who arrived first. When no elements are in the queue, *front* and *rear* will be set to −1. When an element A is inserted (see Figure 5.1(b)) both the pointers point to this element. However, when the next element B is added (Figure 5.1(c)) *rear* points to B where as *front* still points to A, because A has arrived first. Figure 5.1(e) shows a situation, where queue deletion occurs. Since, A is removed from the queue, *front* pointer now points to the second element arrived, i.e., B.

ADT for the Queue

Similar to stack, the ADT for an ordinary queue can also be given (see Figure 5.2).

```
ADT Queue
{
    specification:
          ordered list of elements;
          front and rear pointers for either end;
    operations:
          Qinsert(x)  : Add element x to the queue;
          QDelete()   : Return the front element:
          QFull()     : Return true, if the queue is full;
          QEmpty()    : Return true, if the queue is empty;
}
```

Figure 5.2 ADT for a Queue.

In addition to the four operations given in Figure 5.2 you may add two more:

```
Last() : Return the last element of queue;
First() : Return the first element of queue;
```

The `Last()` operation returns the contents of the queue pointed by *rear* pointer but it does not alter its value. Similarly, `First()` returns the contents of the queue pointed by *front*, and it does not alter its value. In fact, for our discussion of the rest of the topics, we do not need these two operations and hence it is left to the reader to use them appropriately.

5.3 REPRESENTING QUEUES IN C

An important item for a queue to operate is a container to hold the queue elements. In C, we have arrays, which takes care of queue elements and *front* and *rear* are simple integer variables. We will consider the elements of the queue are of integer type and hence we declare an integer array. The queue template can be written as,

```
#define MAX 10
struct  que
{
     int items [MAX];
     int front;
     int rear;
};
typedef struct que * Queue;
```

The queue members are accessed using the C syntax as given below:

```
Queue q;          /* queue variable */
q -> items[i];    /* queue items    */
q -> front;
q -> rear;
```

5.3.1 Queue Initialization

Before any queue operation is done, initializing its members is very important. You have seen in Figure 5.1 that the initial status of the queue is that the `front` and `rear` pointers should have,

```
q->front = -1;
q->rear = -1;
```

If we wish to add an element x to the queue it is to be done based on `rear`. Hence, the following syntax is used,

```
q->items[q->rear] = x;
```

5.3.2 Implementing Queue insertion – Qinsert() function

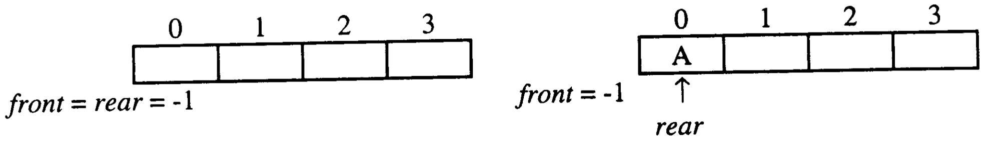
The objective of this section is to show how an element can be inserted into the queue. Elements can be added to the queue only when there is some space in the array. Therefore, before you attempt to add an element check for the availability of space in the queue.

Secondly, once we are sure that some vacant space is available, then increment the `rear` pointer and add the element. This is because, `rear` pointer always points to a filled element in the queue (like *Top* in stack). The function `Qinsert()` is shown in Program 5.1. Note that the parameter `q` is a reference parameter as the queue status changes after the execution of this function.

Program 5.1
Queue Insertion

```
void Qinsert(Queue q , int x)
{
    if (QFull(q))   /* check for overflow */
    {
        printf("Error-overflow\n");
        return;
    }
    q->items[++q->rear] = x;
    if (q->front == -1)
        q->front = 0;
}
```

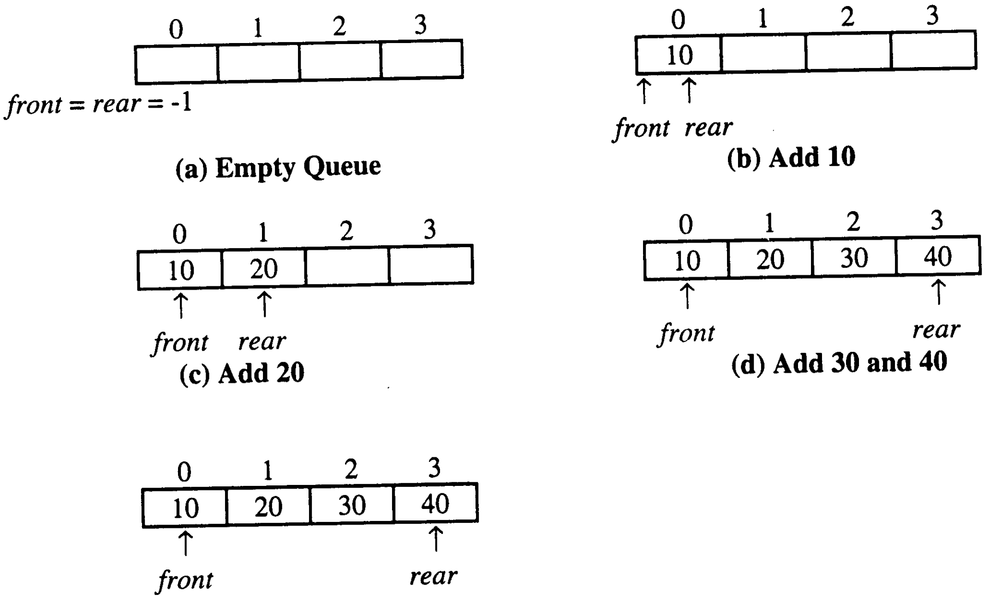
One more work has to be done before we complete our design. Assume that a new element is being added to the queue from the empty state. We mentioned earlier that, increment `rear` pointer and add the element. Now, `rear` will be pointing to the 0th location. However, `front` will still be equal to 0 (see below).



When we try to delete from the queue, the operation is invalid as `front` is not pointing to A. Hence, whenever the first element is inserted in the queue, make `front` to point to 0.This is shown in Program 5.1 as,

```
if (q->front == -1) q->front = 0;
```

Thirdly, the elements cannot be added into queue indefinitely because there is a limitation in array size (`#define MAX 10`). Therefore, we must flag an error when `rear` reaches `MAX - 1`, called as **queue overflow**. The working of `Qinsert()` function is shown in Figure 5.3, assuming MAX = 4.



(a) Empty Queue

(b) Add 10

(c) Add 20

(d) Add 30 and 40

(e) Add 50 (Overflow!)

Fig. 5.3 Queue insertion

When elements 10, 20, 30 and 40 are inserted in the queue, `rear` pointer reaches MAX - 1 (that is, 3). Hence, no more elements can be added. This is an example of queue overflow that will be handled by `QFull()` function.

5.3.3 Implementing Queue Deletion – QDelete()

The objective of this section is to design a function to delete an element from the queue with the following constraints:

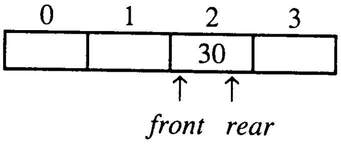
(i) We must obtain the first element in the queue.
(ii) The queue should not be empty.

To satisfy the (i) constraint, the element pointed by `front` should be considered for deletion. The second condition is to check whether the queue is empty or not and for this function `QEmpty()` will be used. Program 5.2 shows the `QDelete()` function with only one parameter `q` of type Queue and it is a reference parameter as well. Whenever we delete an element from the queue, its status will change.

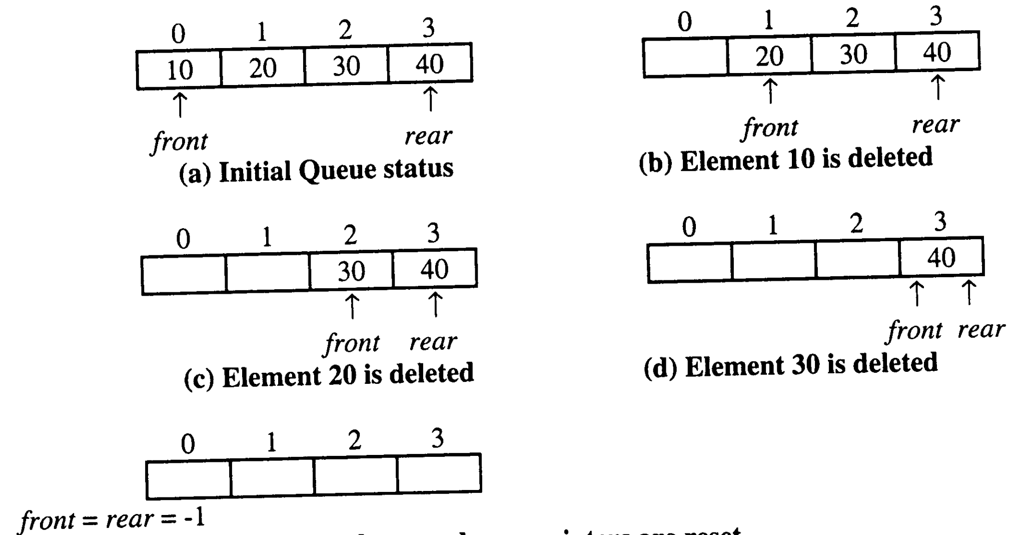
Program 5.2
Queue deletion

```
int Qdelete (Queue q)
{
    int temp;
    if (QEmpty(q))
         return(-1);                 /* underflow      */
    temp = q->items[q->front];
    if (q->front == q->rear)         /* queue is empty */
         q->front = q->rear = -1;
    else
         q->front++;
    return(temp);
}
```

After deletion, `front` pointer is incremented to point to the next element in the queue. Wait!, it can not be incremented blindly this way. Look at the below figure that shows a situation where only one element is present.



When `QDelete()` is called, element 30 is saved in the local variable `temp`. If we simply advance `front`, it would point to a junk value. Whenever, `front == rear` the queue becomes empty especially during a deletion operation. So, `front = rear = -1` is done to make sure that the queue behaves like a fresh one. The queue empty status is detected by calling `QEmpty()` function and is explained later in this topic. Queue deletion is explained starting from the Figure 5.3(f) status and is shown in Figure 5.4.



(a) Initial Queue status

(b) Element 10 is deleted

(c) Element 20 is deleted

(d) Element 30 is deleted

(e) Element 40 is deleted and front and rear pointers are reset.

Fig. 5.4 Execution of Qdelete()

When `front` catches `rear`, both the pointers are reset to –1 as queue becomes empty.

5.3.4 QFull( ) & QEmpty( ) functions

Both the functions receive `ps` as parameter of type `Queue`. The parameter need not be of reference type. The queue full occurs when `rear` pointer is `MAX-1` and queue empty occurs when `front` pointer -1. Both the functions are shown in Program 5.3 and 5.4.

Program 5.3
Queue Full

```
int QFull (Queue q)
{
    if (q->rear == MAX-1)
        return 1;
    else return 0;
}
```

Program 5.4
Queue Empty

```
int QEmpty (Queue q)
{
     if (q->front == -1) return 1;
     else return 0;
}
```

5.4 CIRCULAR QUEUE

One of the major draw backs of an ordinary queue is unnecessary wastage of memory as already discussed is Section 5.3. Consider the Figure 5.4(c), with *front* = 2 and *rear* = 3. Since the maximum size of the queue is 4 any further addition in the queue will result in an overflow error. But, the queue is actually not full! You have locations 0 and 1 empty.

We can solve this problem considering the queue as a **circular** one, rather than as a flat queue. This means that, when *rear* pointer reaches MAX - 1, it is set to 0 so that new elements can be added. This suggests that we must know that there is still space available in the queue. Also, under these circumstances *rear* will be behind *front* pointer (see Figure 5.5).

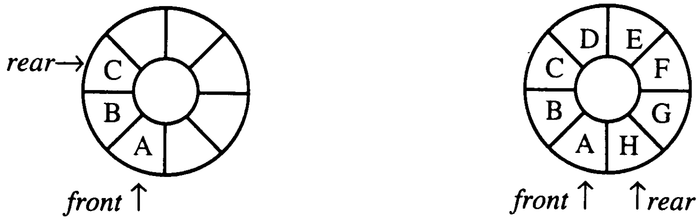


Fig. 5.5 Circular Queues

5.4.1 The template

We prefer to go for a new structure definition, instead of using the definition of an ordinary queue. This is to make the coding easier as you will understand soon. An additional member – a **counter** – is added to the definition of the ordinary queue as show below:

```
struct cir_queue
{
     int count;
     int front;
     int rear;
     int items[MAX];
};
```

```
typedef struct cir_queue * CQueue;
```

The role of count is to maintain the number of elements present in the queue at any point of time. We will also assume that the circular queue members are initialized as,

```
count = 0;
front = 0;
rear = -1;
```

5.4.2 Implementation of CQinsert() function

This function takes on two parameters cq and x. As usual cq is of type CQueue and reference parameter. We wish to insert element x in the queue and is of value parameter, as we don not change its value in the function.
The major task lies in the design of the formula for rear. The requirements are,

(1) During the addition of elements from 0 to MAX - 1 and rear pointer should simply be addressed to the corresponding locations.

(2) When rear ruches MAX - 1 and assuming that there are few vacant positions, it should be made to point to 0.

This can be accomplished using a **mod** operation (%) which is available is C as an operator.

```
cq->rear = (cq->rear + 1)  % MAX;                    ...(5.1)
```

The +1 is because C arrays start from 0. With the initial value of rear being –1, the rear pointer will move as (assuming MAX = 4).
When,

rear = -1;	*rear* = (-1 + 1) % 4 = 0
rear = 0;	*rear* = (0 + 1) % 4 = 1
rear = 1;	*rear* = (1 + 1) % 4 = 2
rear = 2;	*rear* = (2 + 1) % 4 = 3
rear = 3;	*rear* = (3 + 1) % 4 = 0

Therefore, with rear reaching the maximum size of the queue, it is set back to 0. The Program 5.5 shows the CQinsert() function.

Program 5.5
Circular Queue Insertion

```
void CQinsert (CQueue cq , int x)
{
    if (CQFull(cq))    /* check for overflow */
    {
        printf("Error-Queue Full\n");
        return;
    }
```

```
        cq->count++;        /* update count */
        cq->rear = (cq->rear + 1) % MAX;
        cq->items[cq->rear] = x;
    }
```

5.4.3 Implementation of CQdelete() function

The design of `CQdelete()` is same as the insert function, as the formula for `front` pointer is to be obtained. This function returns the deleted element to the calling program and therefore its return type is declared as `int`.

Program 5.6 give the code for deleting as element from the circular queue.

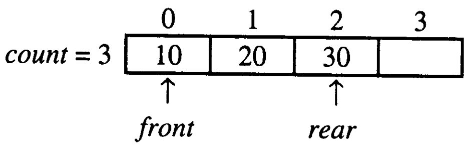
Program 5.6
Circular Queue Deletion

```
int CQdelete (CQueue cq )
{
      int temp;
      if (CQEmpty(cq))
            return(-1);    /* underflow    */
      cq->count--;           /* update count */
      temp = cq->items[cq->front];
      cq->front = (cq->front + 1) % MAX;
      return(temp);
}
```

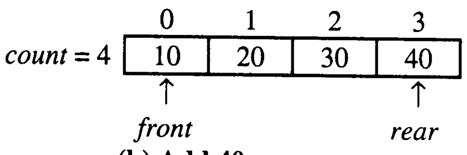
Since the `front` pointer should also work circularly, that is when if reaches MAX - 1 it should be set to 0, we use the mod operator %. The formula is as follows.

```
cq->front = (cq->front + 1) % MAX                    ...(5.2)
```

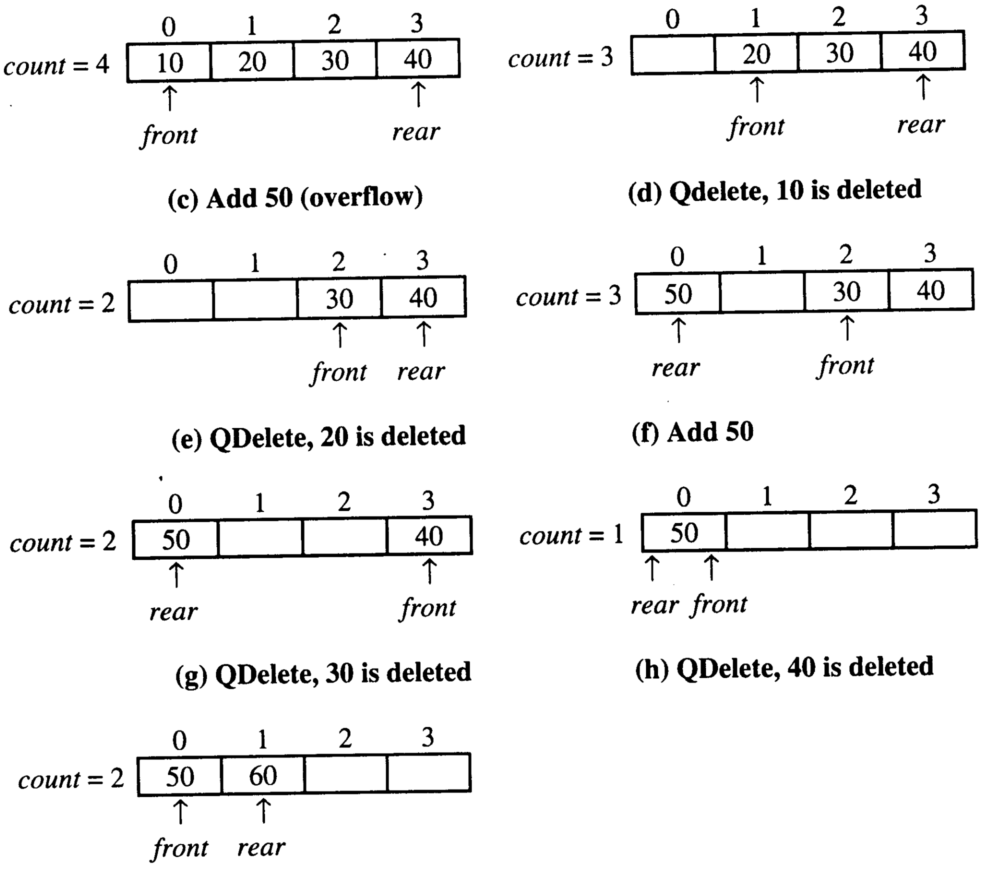
The circular queue is checked for its empty status using the variable `count`. When `count` is zero, obviously queue does not have any elements and retrieving leads to underflow. We will demonstrate the working of both insert and delete functions using the Figure 5.6 Assume MAX = 4.



(a)Initial status



(b) Add 40



(c) Add 50 (overflow)

(d) Qdelete, 10 is deleted

(e) QDelete, 20 is deleted

(f) Add 50

(g) QDelete, 30 is deleted

(h) QDelete, 40 is deleted

(i) Add 60

Fig. 5.6 Working of a Circular Queue

Figures (a) to (d) is self explanatory. Figure (e) shows the circular queue with two elements. When 50 is tried for insertion, this element is added at the position pointed by `rear`, i.e. 0. (see Figure 5.6(f)) and `count` is incremented.

```
    rear = (3 + 1) % 4 = 0
```

Similarly, Figure (h) shows how `front` is brought to 0th position.

CQFull() and CQEmpty() functions

The circular queue is full when `count` reaches the queue size (MAX) as we cannot detect based on `front` and `rear` pointer rules. In the same way, when `count` = 0, no elements can be there in the circular queue.

5.5 DOUBLE ENDED QUEUE (DEQUE)

Definition

A double ended queue, also called as **deque**, is a linear list in which insertion and deletion are done at both ends. It is a special kind of queue where the queue policy may not appear to be strictly followed. Figure 5.7 shows a typical structure of a *deque*.

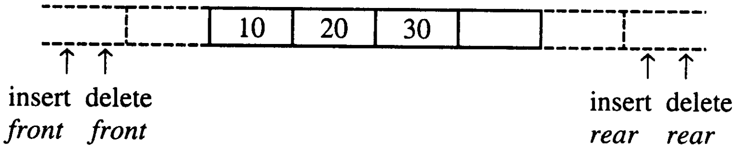


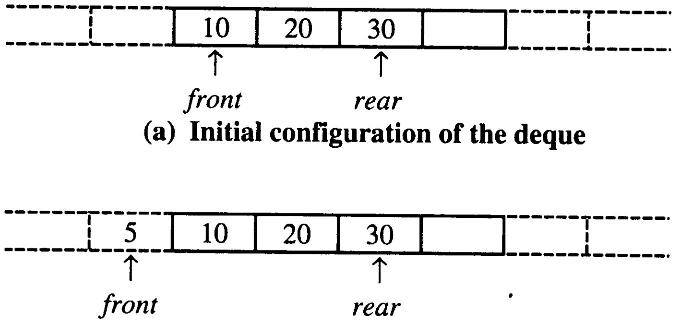
Fig. 5.7 A typical deque

Generally, elements are added at the rear end as per the definition of the queue. But, in a deque we allow insertion at the front end also. Similarly, in addition to front deletion, we allow rear deletion as well. Hence, there are four operations permitted for a *deque*.

1) Front Insertion.
2) Front Deletion.
3) Rear Insertion.
4) Rear Deletion.

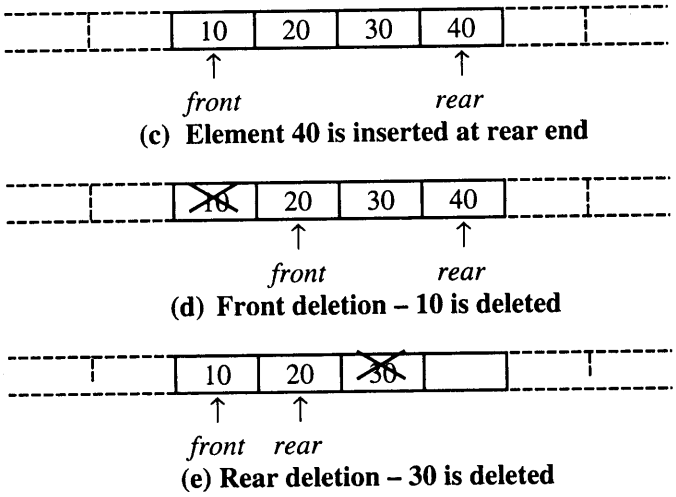
5.5.1 Working of a *deque*

You may be wondering when we allow insert and delete operations on either end, how does the queue behave? When we invoke any of the four operations as seen in Figure 5.7, it should be performed and the appropriate element is to be added (or deleted). We will select the same structure definition like a circular queue with a static array to hold the deque elements (see Figure 5.8).



(a) Initial configuration of the deque

(b) Element 5 is inserted at front end



(c) **Element 40 is inserted at rear end**

(d) **Front deletion – 10 is deleted**

(e) **Rear deletion – 30 is deleted**

Fig. 5.8 The deque operation

Figure (c) and (d) are same as ordinary and circular queue-nothing new. But, Figure (b) and (d) are new operations in which you see that element 5 is added to front of element 10 and the *front* pointer is pointing to element 5. This of course is violating the queue discipline, as somebody who arrives newly stands ahead of 10 rather than standing after 30. When you delete at front with this *deque* contents, you get 5 deleted.

Similarly Figure (d) shows what happens when rear deletion is carried out. The element 30 is deleted and *rear* pointer is decremented by one and points to 20.

5.5.2 Implementation of InsFront( ) and DelRear( ) functions

Designing these two functions efficiently is not an easy task. For example, when we want to add an element to the front, it should be decremented by one to accommodate the new element. But if *front* is already pointing to 0 then *front* = *front* – 1, leads negative value for *front* pointer. This is not acceptable. To avoid this, we can work in a circular fashion. That is, when *front* = 0, simply make this to point to MAX - 1 (last position in the *deque*). The C code given in Program 5.9 shows this operation and also assuming the following initialization:

```
count = 0;
front = 0;
rear = MAX - 1;
```

Program 5.9
Insertion into front of a deque

```
void InsFront (DQueue dq, int x)
{
    if (dq->count == MAX)
```

```
            { printf("Full\n"); return; }
        if (dq->front-- == 0) dq->front = MAX - 1;
        dq->items[dq->front] = x;
        dq->count++;
}
```

Program 5.10
Deletion from rear in a deque

```
int DelRear(DQueue dq)
{
        int t;
        if (dq->count == 0) return -1;
        t = dq->items[dq->rear];
        if (dq->rear-- == 0) dq->rear = MAX - 1;
        dq->count--; return t;
}
```

The design of rear deletion is same as front insertion except that *rear* pointer takes the place of *front* pointer (see Figure 5.10). In the normal case when an element is deleted at the rear end, the *rear* pointer need to be decremented. However, when it is in 0th position it cannot be decremented but made to point to MAX - 1.

Displaying the contents of a *deque* can be done using the accessing formula of a circular queue. The code is shown in Program 5.11.

Program 5.11
Displaying contents of a deque

```
void Display (DQueue dq)
{
        int i, f;
        if (!DQEmpty(dq))
        {
                f = dq->front;
                for (i = 1; i <= dq->count; i++)
                {
                        printf("%d ", dq->items[f]);
                        f = (f + 1) % MAX;
                }
                printf("\n");
        }
        else printf("DQueue Empty!\n");
}
```

5.6 PRIORITY QUEUE

An ordinary queue works on FIFO policy, but the order of deletion in a **priority queue** depends upon the **element priority**. Elements are deleted either in increasing or decreasing order of priority rather than in the order in which they arrived in the queue.

Definition

A **priority queue** is a collection of elements, each one having an assigned priority. Insertion and deletion are two basic operations to be done even for this queue. However, deletion operation is somewhat different.
We classify two types of priority queues based upon the way in which the elements are deleted.

(1) Ascending priority queue (**Min Priority queue)**
  - Element with minimum priority or value is to be deleted.

(2) Descending priority queue (**Max priority queue)**
  - Element with maximum priority or value is to be deleted.

The elements in a priority queue need not have distinct priorities; this means that two or more elements can have the same priority. The abstract data type (ADT) specification for an Ascending Priority Queue (*apq*) is shown in Figure 5.9.

```
ADT apq
{
    specifications:
         finite collection of elements in a list each
         with an associated priority.
    operations:
         ApqInsert(x)    : Insert an element x in a apq.
         ApqMinDelete()  : Return the element with minimum
                           priority.
}
```

Figure 5.9 ADT for a Min Priority Queue

Example

Assume that in an Internet browsing center, the users are to be allotted the machines on a priority basis. Each user pays a fixed amount per use. However, the time needed by the users may vary. The objective of the owner would be not to keep any machine idle and maximize his returns.

To accomplish this task ascending priority queues may be used to maintain the users who are waiting for the machine time. Who ever has asked lesser time will be put in a higher priority.

5.6.1 Implementation techniques

An efficient implementation for the priority queue is to use a **heap** (discussed in Section 7.10). In this section we develop array implementation for *apq* (or *dpq*). To differentiate between the priority of an element and its actual value, we shall modify the queue structure as,

```
struct  pr_que
{
     int pr;
     int items;
}
struct pr_que pq[MAX];
```

Figure 5.10 Template of a priority queue

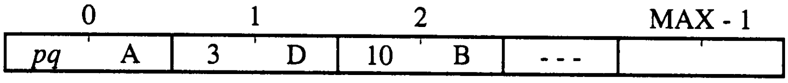
The *front* and *rear* pointer may be declared separately to complete the definition of the priority queue. Alternatively, the `pr-que` structure and the pointers (`front` and `rear`) can be grouped under another structure definition as,

```
struct pq
{
     struct pr_que[MAX];
     int front, rear;
};
struct pq p;
```

To access the priority of an element pointed by `front`, use

```
p.pq[p.front].pr
```

Using the definition shown in Figure 5.10, the priority queue elements would appear as in Figure 5.11.



The array `pq[0:MAX-1]` has two data elements in each location – the priority and the actual value (considered as *char* type).

5.7 SUMMARY

- A **queue** is a linear list in which additions and deletions takes place at different ends. The end at which insertions take place is called the **rear**, and at which deletions happen is called the **front**.
- The queue policy is that the service is provided to people who have arrived first. In other words, queue operates on a **First-In-First-Out (FIFO)** manner.
- The C functions for queue insertion, queue delete, queue *full* and queue *empty*, etc. were developed.
- Circular queues are memory efficient than ordinary queues. Instead of considering a flat queue we can imagine a queue to be of circular type. This means that, when *rear* pointer reaches MAX - 1, it is set to 0 so that new elements can be added, provided there is some space in the queue.
- A double ended queue, also called as **deque**, is a linear list in which insertion and deletion are done at both ends.
- A **priority queue** is a collection of elements, each one having an assigned priority. Insertion and deletion are two basic operations to be done for this queue as well based upon the assigned priority.
- There are two types of queues called as ascending priority queue or *apq* and descending priority queue or *dpq*.
  - Element with minimum priority or value is to be deleted in an *apq*.
  - Element with maximum priority or value is to be deleted in a *dpq*.

5.8 EXERCISES

5.1 What is a queue? Mention some of the applications of queues in general problem solving.

5.2 Implement an ordinary queue using static structures (that is, using arrays).

5.3 Under what conditions a queue may become (1) Full and (2) Empty? Give relevant figures.

5.4 What are the drawbacks of an ordinary queue and how will you overcome it?

5.5 Write the queue contents after every operation as given below:
(Assume, MAX = 4)

(1) Qinsert(11);
(2) Qdelete();
(3) Qdelete();
(4) Qinsert(11);
(5) Qinsert(22);
(6) Qinsert(33);
(7) Qinsert(44);
(8) Qinsert(55);
(9) Qdelete( );
(10) Qdelete( );
(11) Qinsert(66);
(12) Qdelete(77);

5.6 Repeat problem 5.6, but use a circular queue.

5.7 Explain Ascending and Descending priority queues and bring out all possible methods of their array implementation issues in C.

5.8 Simulate the following problem:
Assume that a super market has two queues *Q1* and *Q2*. Each customer has an *id* and the number of items purchased, *y*. The queue *Q2* is populated when the number of items exceeds 10 for each customer, and if it is less than or equal to 10, then they will be put in *Q1*. Write C functions

(a) To add a customer in the appropriate queue.

(b) To delete the customer from queue *Q1*, if it is not empty. In case, if it is empty then delete from *Q2*. If both are empty flag an error.

While designing your C functions, use static data structures only. Do not consider the queue as a circular one.

5.9 Develop a complete C program to simulate the working of a priority queue using arrays. You must have functions that would insert, delete, modify an element (integer) in the queue. Trace your program using an appropriate data set of your choice (consider $n = 5$).

5.10 In this problem, you must concatenate two queues retaining the same order of arrival in both the queues.

5.11 Assume that suddenly we decide that a queue has to become a stack and a stack has to become a queue, for some reason. Develop two separate functions to do this.

5.12 The City Traffic department wants to simulate the following:
There are four signal lights kept in a road junction. We will assume that they are numbered as R1, R2, R3 and R4. The vehicles arriving at all these four roads have to be kept in separate queues. The deletion (movement of the traffic) process should happen in a circular fashion starting from R1, R2, R3 and R4.

5.13 Design an algorithm to operate two queues using a single array just to save memory space. You must write Qinsert() Qdelete() functions for this type of queue.

Chapte 6

Linked Lists

6.1 INTRODUCTION

So far, the data objects that we have discussed were all stored in an array - for example, stack, queue, etc. The data objects were accessed using a formula, because the elements were stored in contiguous locations. In this chapter, we discuss a **linked** representation for the instance of each data object. No formula is used to locate individual elements. Linked representation with dynamic memory allocation offers a new dimension to data storage and retrieval.
Following topics will be explained in depth:

1. Singly linked lists
2. Circular linked lists
3. Doubly linked lists

Under each topic, we will show how to implement the data structure like stack, queue, etc. using **linked lists.** There are many problems that uses linked lists to represent the data. For instance, to represent a polynomial, linked representation is an ideal choice.

6.2 SINGLY LINKED LIST - DEFINITION

In the linked representation, each data object is represented as a **node**. Each node consists of two compartments - the first one is the **value** of the data object and the second is the information regarding the location of the other nodes. This explicit information about the location of the adjacent nodes is called a **link** or **pointer** (see Figure 6.1).

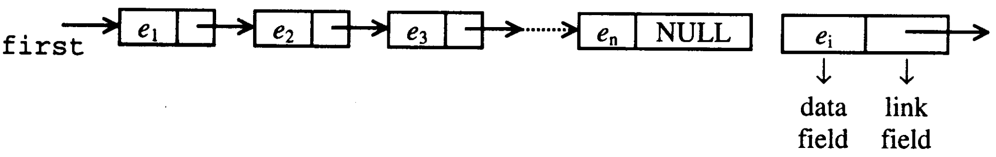


Fig. 6.1 Linked representation of a list of elements

In Figure 6.1, e_1 is the data element and the arrow indicates the **link** field that points to the next node. The variable `first` is a pointer pointing to the first node. Since each node has exactly one link, this structure is called as a **singly linked list**. The node with data element e_n has no link to other nodes as this is the last node. Notice that its link field has a **NULL** or **0**.

6.3 LINKED LIST – A closer look

You may notice that a linked list does not require maximum number of node specification. It can grow or shrink dynamically making it more memory efficient. Compare this with an array representation where the number of elements needed to be specified in advance (specifically).

Since, no formula is used to locate an element in a linked list, the address of every node is stored in its previous node. That is, e_i, links to that for $e_i + 1$, $1 \leq i \leq n$ this suggests that the nodes may be scattered in memory. Hence, unless these nodes are linked through pointers, it is not possible to locate each element. The nodes can be added or deleted in the linked list at run time. This is done by allocating the memory is a heap and this is the technique followed by most of the compilers.

6.4 ALGORITHMIC NOTATIONS

To enable us to write algorithms for linked list based problems, we must define algorithmic notations. These notations are purely for the purpose of writing the algorithms and can not be used in C programs. The variable names and their meanings given below will also be applicable through out this chapter.

(1) *first* – a pointer to the first node of the list.

(2) *link(first)* – address of the link field of first.

(3) *info(first)* – data element field of first.

(4) *last* – address of the end node.

(5) NULL – a special address with value 0.

Example

Consider a typical linked with 4 nodes, assuming that the information field (or data field) is of *char* type shown in Figure 6.2.

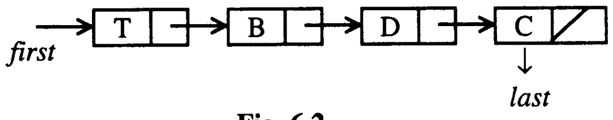


Fig. 6.2

Applying the notations shown in Section 6.4, on the linked list shown in Figure 6.2, we have

(1) *info(first)* yields T
(2) *info(last)* yields C
(3) *first = link(first)*; *first* to point to the second node.
(4) *info(link(first))* yields B
(5) *current = link(first)* *current* to point to the second node .
info(current) yields B
(6) Accessing the fields of pointer variable when it is pointing to NULL is illegal.
(7) *first = getnode*(); allocate dynamic memory to *first.*
(8) *free(first)*; deallocate memory of *first.*

Notice that the statement shown in serial number 3 is useful in traversing the list, if it is executed repeatedly. In a linked list you can move through the list one node at a time using this kind of statement. Traversing will end when the last node's NULL field is reached.

6.5 ADT FOR LINKED LISTS

The abstract data type (ADT) specification for a linked arrangement of a list is shown in Figure 6.3.

```
ADT LinearLinkedList
{
     specification:
          finite collection of zero or more nodes. Each
          node has an information field and a link field.
     operations:
          Create(x):Create an empty linked list and add x
                   at front or rear.
          Destroy():Delete all the elements (nodes) in the
                   list.
```

```
              Length():Return the length (number of nodes) in
                       the list.
              Find(k,x):return the kth element of the list in
                        x, return 0 if there is no kth element,
                        return otherwise.
              LDelete(x):delete the element x in the list and
                         return the same. Return the modified
                         list.
              Insert(k,x):Insert element x just after kth
                          element return the modified list.
              Display():Display the contents of the list on
                        screen.
}
```

Fig. 6.3 ADT for a linked list

The operations specified for a linked list in Figure 6.3 need not necessarily be the only set, but additional operations may be added. This generally depends upon the particular problem, which we are trying to solve.

In general, `Create()` and `Display()` are two basic operations required invariably for all linked list based problems. Also, all representations of the ADT must satisfy the specification and operations.

6.6 IMPLEMENTATION USING C

A linked list is a collection of nodes. Each node has two different types of information. The first field is information filed, may be of any primitive or structured data types like *int*, *float*, *char*, etc., and the second one is a pointer field. Since, a node specification combines two different data types, we prefer a structure definition as shown below (assuming an integer `info` field).

```
struct List
{
     int info;
     struct List *link;
}
typedef struct List * NODE;
```

The second member is a pointer type and its data type is `struct List`, because the link field of every *node* points to a *node* again. This type of structure definition is knows as **self-referential structure**. Notice that before completion of definition of the list structure, it uses a member *link. This is why it is called as self-referential.

6.6.1 Dynamic Memory Allocation to a Node

The nodes in a list grow (or shrink) dynamically. Initially, there may be, say, four nodes created and if there is a requirement for few more nodes then, nodes can be added to the list at run time. Therefore, every node should get the memory dynamically and is done using `malloc()` function in C. The syntax of `malloc()` function is as follows,

```
current = (void *) malloc (size in bytes);
```

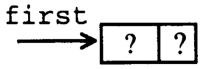
This is an incomplete or incorrect definition, but we will correct this soon. The `malloc()` function expects the size (in bytes) of the memory to be allocated in heap dynamically. Since this memory could be used for storing any type of data, it returns a *void* pointer. To balance the data types on either side, we write

```
current = (NODE) malloc (sizeof(struct List));
```

assuming, `current` is of type NODE. Invariably, all linked list programs will use this method to obtain dynamic memory.

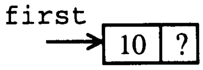
Example

Any number of pointer variables can be made to point to a linked list or an individual node provided it is of same type as the list. We will demonstrate the operations with linked lists by showing the following series of statements. Be sure that the dynamic memory is allocated for the pointer variable before you put any data element or retrieve from it (see Figure 6.4) Assume, `first`, `last` are of type NODE.

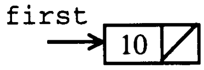


Step 1: `first = (NODE) malloc (sizeof(struct List));`

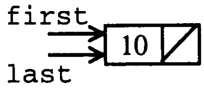
Step 2: `first->info = 10;`



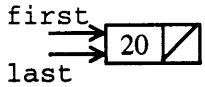
Step 3: `first->link = NULL;`

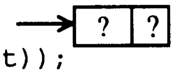


Step 4: `last = first;`



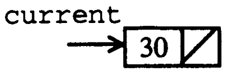
Step 5: `last->info = 20;`





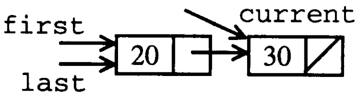
Step 6: `current = (NODE)malloc(sizeof(struct List));`

Step 7: `current->info = 30;`



```
        current->link = NULL;
```

Step 8: `first->link = current;`



Step 9: `last = current;`

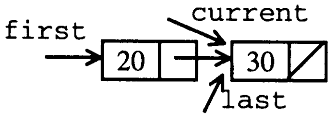


Fig. 6.4 Simple linked list operations

In Step 1, `first` points to an uninitialized node - this means that the contents of `info` field and `link` filed are junk values. In Step 2 and Step 3 element 10 is stored in the `info` field pointed by `first`, and NULL (or 0) in its `link` field
Note the syntax,

- `first` points to a node.
- `first->info` and `first->link` to access the structure members.

The pointer variable last also points to the same node pointed by first (Step 4). Notice that the `info` field can be altered by using last (Step 5). Step 6 creates one more node pointed by `current` and is initialized to 30 (Step 7). To join two nodes pointed by `first` and `current`, simply put the address of `current` into the `link` field of `first` so that both the nodes form a chain (see Step 8).

If you loose the address of the first node of the linked list (by some means), then you have no way to get back it. Therefore, you must be extremely careful when you operate with linked lists. Consider the following piece of C code,

```
free(first);
```

Assuming that `first` points to a linked list of say 4 nodes, executing this statement leads to deallocating the memory of the first node (not all nodes - you can only deallocate one node at a time) and returning this memory to the heap. Once the memory is deallocated, you can't use first to access the list.

6.7 LINKED LIST OPERATIONS

This section will address the operations pertaining to linked lists based upon the ADT specification. Every operation is discussed with the objective, the design, relevant figures, and C code.