Queues(Unit 2.2)

1.Introduction

Queues are useful to solve various system programs. Some simple applications of queues in our everyday life as well as in computer science.

Queuing in front of a counter

Suppose there are a number of customers in front of a counter to get service (say, to collect tickets or to withdraw/deposit money in a teller of a bank). The customers are forming a queue and they will be served in the order they arrived, that is, a customer who comes first will be served first.

Traffic control at a turning point

Suppose there is a turning point in a highway where the traffic has to turn. All the traffic will have wait in a line till it gets the signal for moving. On getting the 'Go' signal the vehicles will turn on a first come, first turn basis.

Process synchronization in multi-user environment

In a multi-user environment, more than one process is handled by the monitor (operating system). The three different states that a process may have are the following: READY, RUNNING, and AWAITED. A process is in the READY state when it is submitted to the system for execution. A process is in the RUNNING state if it is currently under execution. Similarly, a process will be transferred to the AWAITED state when it requires resource(s) which is/are busy: In order to synchronize the execution of processes, the monitor has to maintain two queues, namely Q 1 and Q2, for READY and A WAITED states respectively where a process which entered a queue first will be exited first.

Resource sharing in a computer centre

In a computer centre, where resources are limited compared to the demand, users must sign a waiting register. The user who has been waiting for a terminal for the longest period of time gets hold of the resource first, then the second candidate, and so on. Here the waiting list maintains a queue and the first signed will be the first allowed.

2 DEFINITION

Like a stack, a queue is an ordered collection of homogeneous data elements; in contrast with the stack, here, insertion and deletion operations take place at two extreme ends. A queue is also a linear data structure like an array, a stack and a linked list where the ordering of elements is in a-linear fashion. The only difference between a stack and a queue is that in the case of stack insertion and deletion (PUSH and POP) operations are at one end (TOP) only, but in a queue insertion (called ENQUEUE) and deletion (called DEQUEUE) operations take place at two ends called the REAR and FRONT of the queue, respectively. Figure represents a model of a queue structure. Queue is also termed first-in first-out (FIFO)

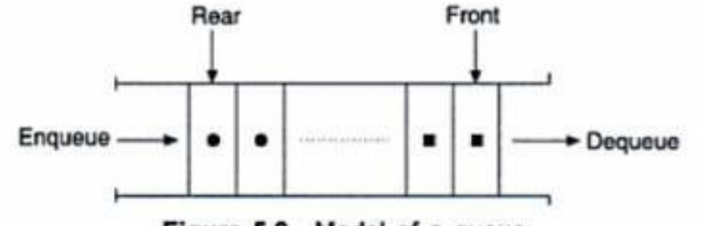


Figure 5.2 Model of a queue.

3 REPRESENTATION OF QUEUES

There are two ways to represent a queue in memory: Using an array & Using a linked list
The first kind of representation uses a one-dimensional array and it is a better choice where a queue of fixed size is required. The other representation uses a double linked list and provides a queue whose size can vary during processing.

3.1 Representation of a Queue using an Array

A one-dimensional array, say Q[l ... N], can be used to represent a queue. Figure shows an instance of such a queue. With this representation, two pointers, namely FRONT and REAR, are used to indicate the two ends of the queue. For the insertion of the next element, the pointer REAR will be the consultant and for deletion the pointer FRONT will be the consultant.

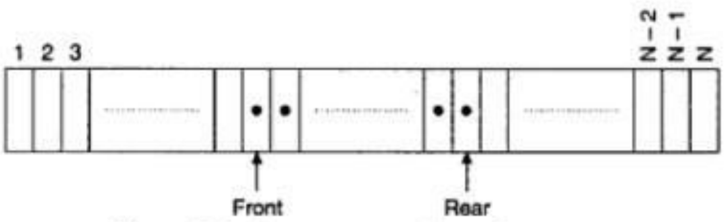


Figure 5.3 Array representation of a queue.

Three states of a queue with this representation are given below:

Queue is empty

FRONT = 0
REAR = 0 (and/or)

Queue is full

REAR = N
FRONT = 1 (when full by compact)

Queue contains elements ≥ 1

FRONT ≤ REAR
Number of elements = REAR − FRONT + 1

Algorithm Enqueue

Input: An element ITEM that has to be inserted.
Output: The ITEM is at the REAR of the queue.
Data structure: Q is the array representation of a queue structure; two pointers FRONT and REAR of the queue Q are known.

```
Steps:
1.  If (REAR = N) then                        // Queue is full
2.      Print "Queue is full"
3.      Exit
4.  Else
5.      If (REAR = 0) and (FRONT = 0) then     // Queue is empty
6.          FRONT = 1
7.      EndIf
8.      REAR = REAR + 1            // Insert the item into the queue at REAR
9.      Q[REAR] = ITEM
10. EndIf
11. Stop
```

Algorithm Dequeue

Input: A queue with elements. FRONT and REAR are the two pointers of the queue Q.
Output: The deleted element is stored in ITEM.
Data structures: Q is the array representation of a queue structure.

```
Steps:
1.  If (FRONT = 0) then
2.      Print "Queue is empty"
3.      Exit
4.  Else
5.      ITEM = Q[FRONT]                              // Get the element
6.      If (FRONT = REAR)        // When the queue contains a single element
7.          REAR = 0                             // The queue becomes empty
8.          FRONT = 0
9.      Else
10.         FRONT = FRONT + 1
11.     EndIf
12. EndIf
13. Stop
```

Let us trace the above two algorithms with a queue of size = 10. Suppose the current state of the queue is FRONT = 8, REAR = 9. Ten operations are requested as under:

1. DEQUEUE
2. ENQUEUE
3. ENQUEUE
4. DEQUEUE
5. DEQUEUE
6. DEQUEUE
7. ENQUEUE
8. ENQUEUE
9. DEQUEUE
10. DEQUEUE

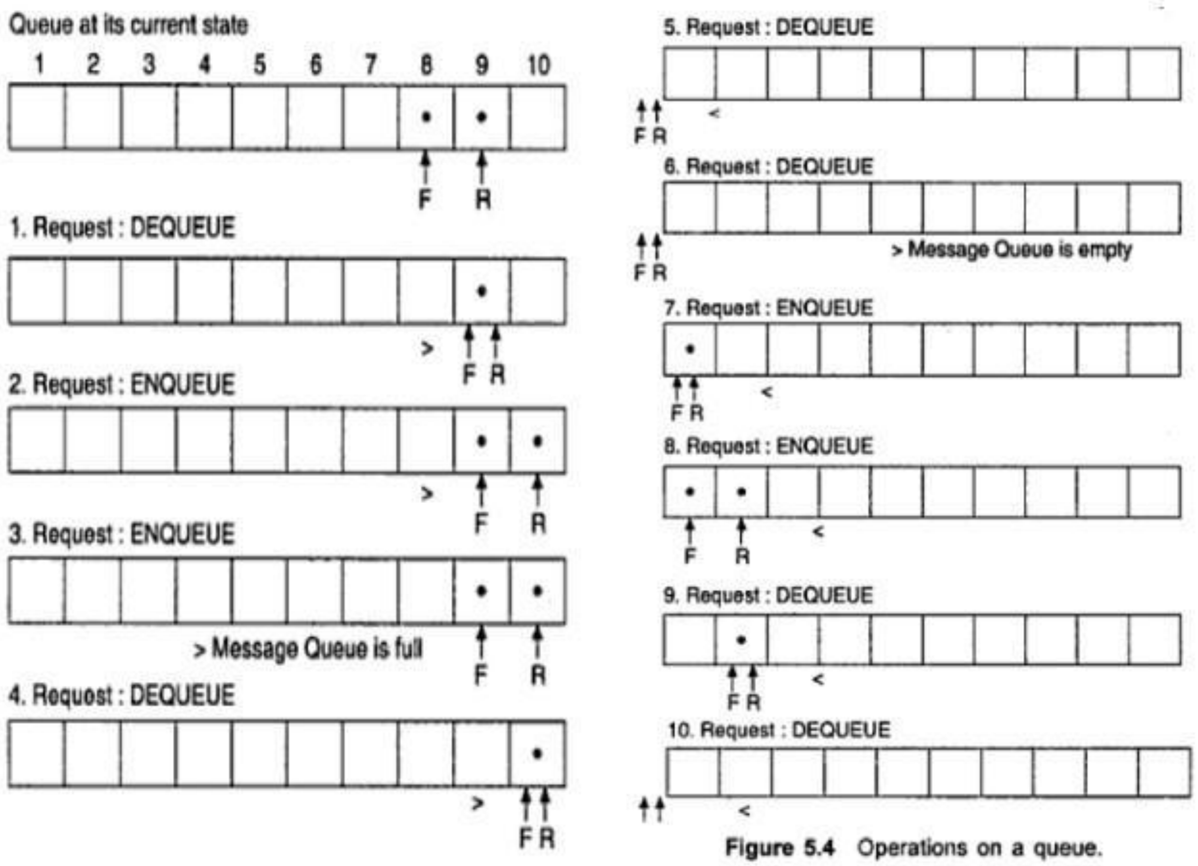


Figure 5.4 Operations on a queue.

There is one potential problem with this representation. From Figure, we can see that with this representation, a queue may not be full, still a request for insertion operation may be denied. For example, on request (3) (in Figure) 8 rooms are available but insertion is not possible as the insertion pointer reaches the end of the queue. This is simply wastage of the storage. This type of representation can be recommended for an application where the queue is emptied at certain intervals.

3.2 Representation of a Queue using a Linked List

One more limitation of a queue, other than the inadequate service of insertion represented with an array, is the rigidness of its length. In several applications, the length of the queue cannot be predicated before and it varies abruptly. To overcome this problem, another preferable representation of a queue is with a linked list. Here, we select a double linked list which allows us to move both ways. Figure shows the double, linked list representation of a queue. The pointers FRONT and REAR point the first node and the last node in the list.

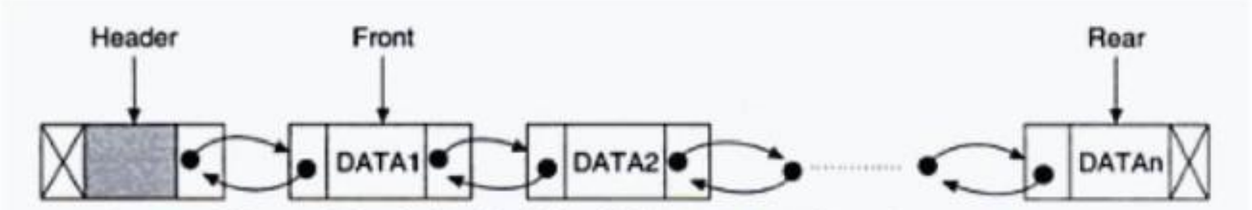


Figure 5.6 A double linked list representation of a queue.

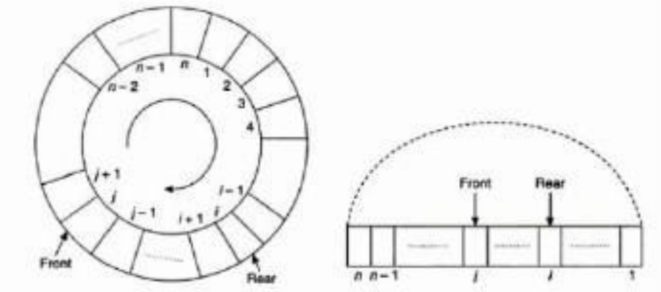
Two states of the queue, either empty or containing some elements, can be judged by the following tests:

4 VARIOUS QUEUE STRUCTURES

Two different queue structures, that is, either using an array or using a linked list. Other than these, there are some more known queue structures.

4.1 Circular Queue

For a queue represented using an array when the REAR pointer reaches the end, insertion will be denied even if room is available at the front. One way to avoid this is to use a circular array. Physically, a circular array is the same as an ordinary array, say A[1 ... N], but logically it implies that A[1] comes after A[N] or after A[N], A[1] appears. Figure shows logical and physical views of a circular array.



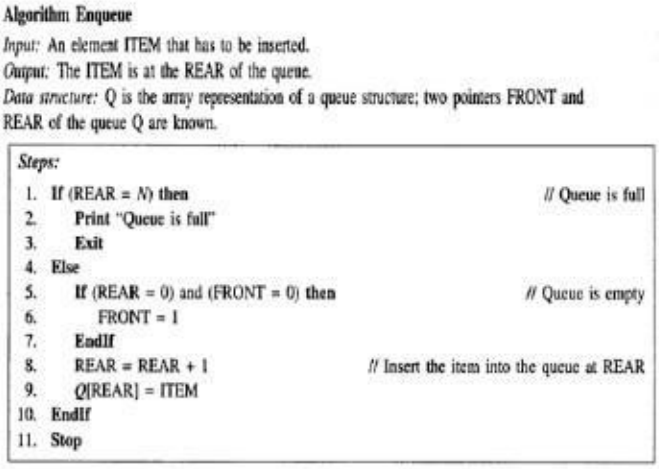
(a) Circular queue (logical) (b) Circular array (physical)

Figure 5.7 Logical and physical views of a circular queue.

The principle underlying the representation of a circular array is as stated below:
Both pointers will move in a clockwise direction. This is controlled by the MOD operation; for example, if the current pointer is at i then shift to the next location will be i MOD LENGTH + 1, 1 <=i <= LENGTH (where LENGTH is the queue length). Thus, if i = LENGTH (that is at the end), then the next position for the pointer is 1. With this principle the two states of the queue regarding, i.e. empty or full, will be decided as follows:

Circular queue is empty	Circular queue is full
REAR = 0	FRONT = (REAR MOD LENGTH) + 1FRONT = 0

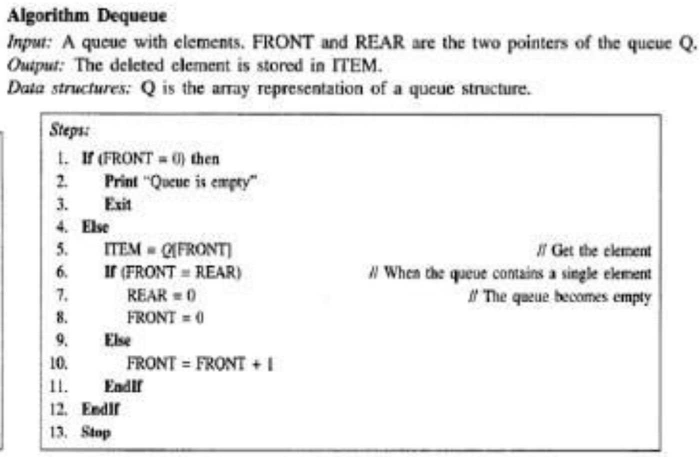
The following two algorithms describe the insertion and deletion operations on a circular queue.



Algorithm Enqueue

Input: An element ITEM that has to be inserted.
Output: The ITEM is at the REAR of the queue.
Data structure: Q is the array representation of a queue structure; two pointers FRONT and REAR of the queue Q are known.

```
Steps:
If (REAR = N) then                          // Queue is full
    Print "Queue is full"
    Exit
Else
    If (REAR = 0) and (FRONT = 0) then      // Queue is empty
        FRONT = 1
    EndIf
    REAR = REAR + 1                         // Insert the item into the queue at REAR
    Q[REAR] = ITEM
EndIf
Stop
```



Algorithm Dequeue

Input: A queue with elements. FRONT and REAR are the two pointers of the queue Q.
Output: The deleted element is stored in ITEM.
Data structures: Q is the array representation of a queue structure.

```
Steps:
If (FRONT = 0) then
    Print "Queue is empty"
    Exit
Else
    ITEM = Q[FRONT]                          // Get the element
    If (FRONT = REAR)                        // When the queue contains a single element
        REAR = 0                             // The queue becomes empty
        FRONT = 0
    Else
        FRONT = FRONT + 1
    EndIf
EndIf
Stop
```

Assume that initially the queue is empty, that is, FRONT = REAR = 0.

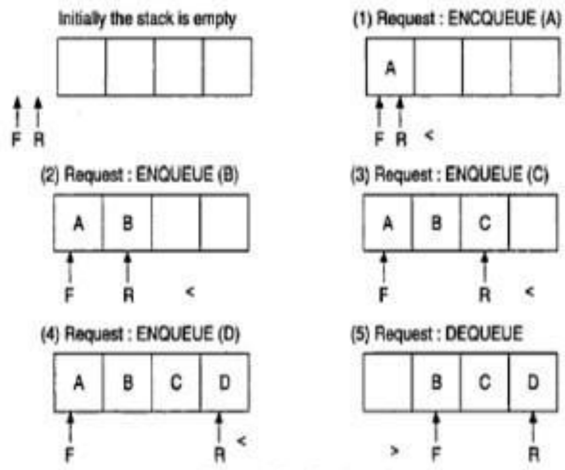

Initially the stack is empty
F R
(1) Request : ENCQUEUE (A)
A
F R <
(2) Request : ENQUEUE (B)
A B
F R <
(3) Request : ENQUEUE (C)
A B C
F R <
(4) Request : ENQUEUE (D)
A B C D
F R <
(5) Request : DEQUEUE
B C D
> F R

Figure 5.8 Continued.

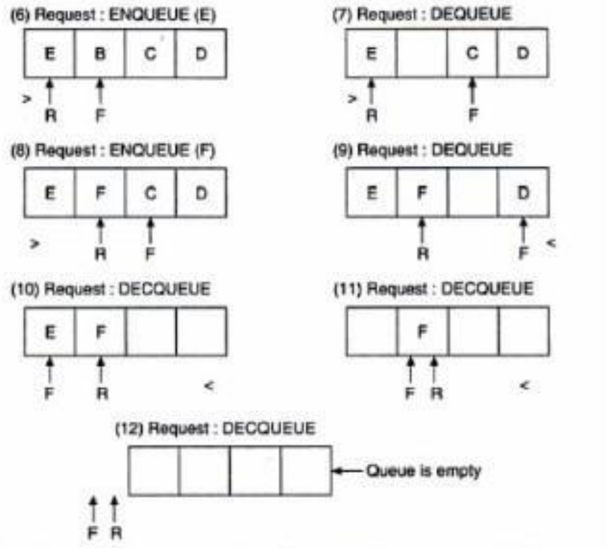


Figure 5.8 Tracing insertion and deletion operations on a circular queue.

4.2 Deque

Another variation of the queue is known as deque (may be pronounced 'deck'). Unlike a queue, in deque, both insertion and deletion operations can be made at either end of the structure. Actually, the term deque has originated from double ended queue. Such a structure is shown in Figure.

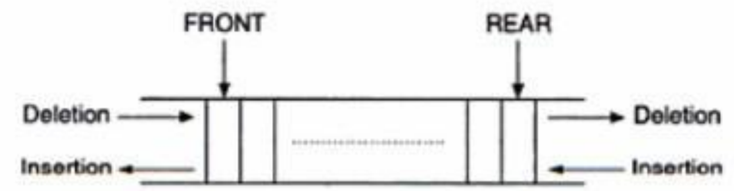


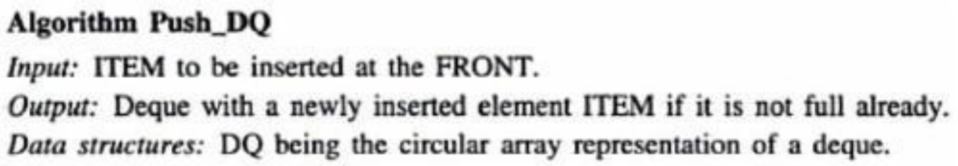
Figure 5.9 A deque structure.

It is clear from the deque structure that it is a general representation of both stack and queue. In other words, a deque can be used as a stack as well as a queue. There are various ways of representing a deque on the computer. One simpler way to represent it is by using a double linked list. Another popular representation is using a circular array (as used in a circular queue).

The following four operations are possible on a deque which consists of a list of items:

1.Push_DQ(ITEM): To insert ITEM at the FRONT end of a deque.
2.Pop_DQ( ): To remove the FRONT item from a deque.
3.Inject(ITEM): To insert ITEM at the REAR end of a deque.
4.Eject( ): To remove the REAR ITEM from a deque.

These operations are described for a deque based on a circular array of length LENGTH.
Let the array be DQ[1 ... LENGTH].

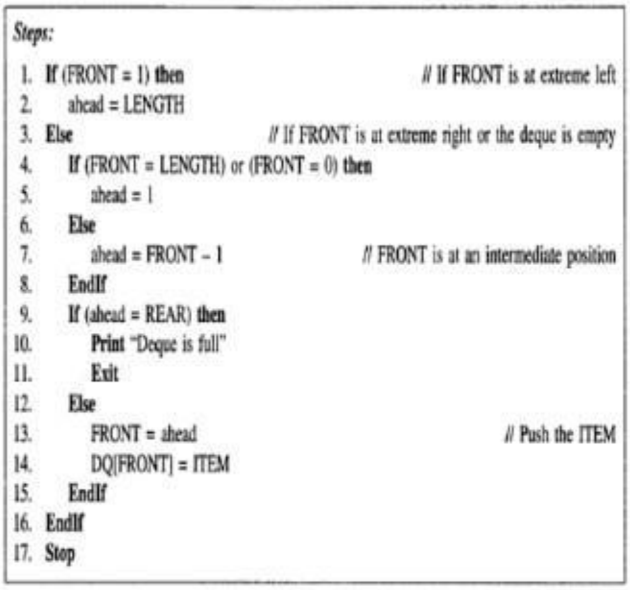


Algorithm Push_DQ

Input: ITEM to be inserted at the FRONT.

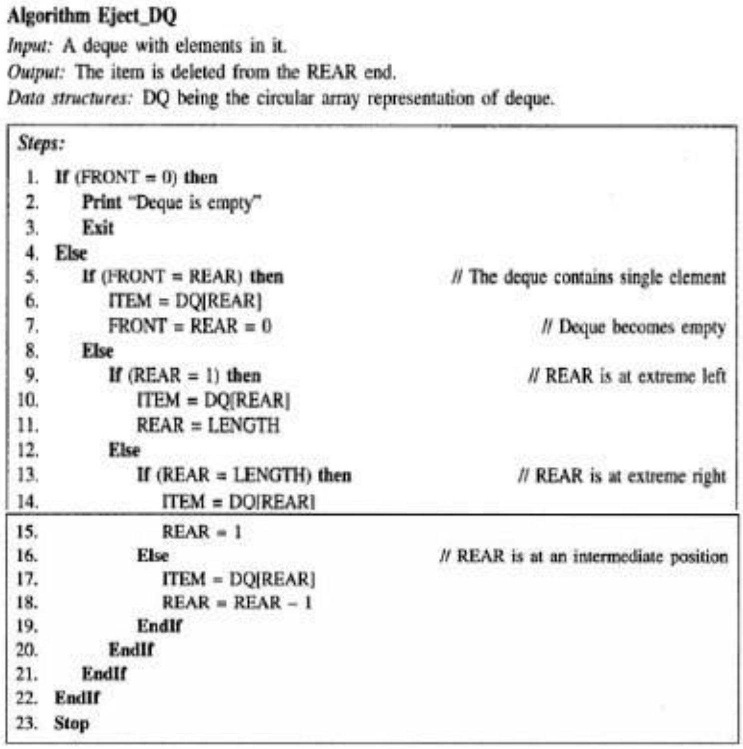
Output: Deque with a newly inserted element ITEM if it is not full already.

Data structures: DQ being the circular array representation of a deque.



Steps:

```
If (FRONT = 1) then                          // If FRONT is at extreme left
    ahead = LENGTH
Else                          // If FRONT is at extreme right or the deque is empty
    If (FRONT = LENGTH) or (FRONT = 0) then
        ahead = 1
    Else
        ahead = FRONT - 1                    // FRONT is at an intermediate position
    EndIf
    If (ahead = REAR) then
        Print "Deque is full"
        Exit
    Else
        FRONT = ahead                        // Push the ITEM
        DQ[FRONT] = ITEM
    EndIf
EndIf
Stop
```



Algorithm Eject_DQ

Input: A deque with elements in it.

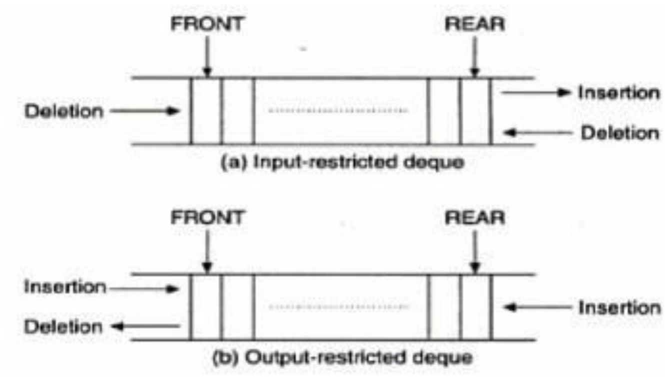
Output: The item is deleted from the REAR end.

Data structures: DQ being the circular array representation of deque.

Steps:

```
If (FRONT = 0) then
    Print "Deque is empty"
    Exit
Else
    If (FRONT = REAR) then                   // The deque contains single element
        ITEM = DQ[REAR]
        FRONT = REAR = 0                     // Deque becomes empty
    Else
        If (REAR = 1) then                   // REAR is at extreme left
            ITEM = DQ[REAR]
            REAR = LENGTH
        Else
            If (REAR = LENGTH) then          // REAR is at extreme right
                ITEM = DQ[REAR]
                REAR = 1
            Else                             // REAR is at an intermediate position
                ITEM = DQ[REAR]
                REAR = REAR - 1
            EndIf
        EndIf
    EndIf
EndIf
Stop
```

There are, however, two known variations of deque: Input-restricted deque and Output-restricted deque. These two types of variations are actually intermediate between a queue and a deque. Specifically, an input-restricted deque is a deque which allows insertions at one end (say REAR end) only, but allows deletions at both ends. Similarly, an output-restricted deque is a deque where deletions take place at one end only (say FRONT end), but allows insertions at both ends. Figure represents two such variations of deque.



(a) Input-restricted deque

(b) Output-restricted deque

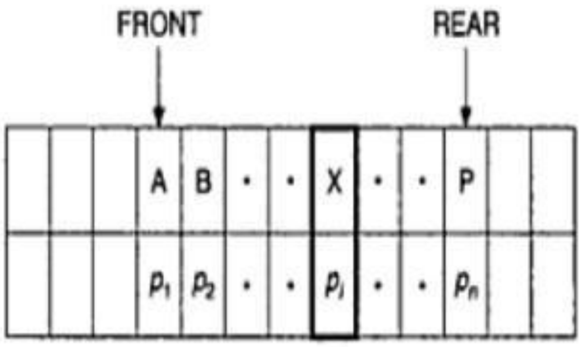


Figure 5.11 View of a priority queue.

4.3 Priority Queue

A priority queue is another variation of queue structure. Here, each element has been assigned a value, called the priority of the element, and an element can be inserted or deleted not only at the ends but at any position on the queue. Figure shows a priority Queue. With this structure, an element X of priority Pi may be deleted before an element which is at FRONT. Similarly, insertion of an element is based on its priority, that is, instead of adding it after the REAR it may be inserted at an intermediate position dictated by its priority value. There are various models of priority queue known in different applications. Let us consider a particular model of priority queue.

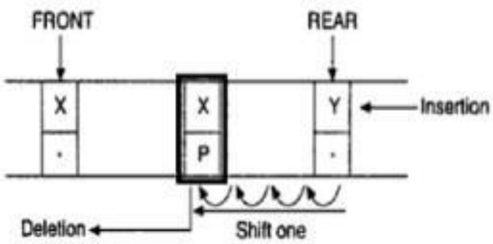
1. An element of higher priority is processed before any element of lower priority.
2. Two elements with the same priority are processed according to the order in which they were added to the queue.

Here, process means two basic operations namely insertion or deletion. There are various ways of implementing the structure of a priority queue. These are:
“Using a simple/circular array, Multi-queue implementation, Using a double linked list, Using heap tree.”

Priority queue using an array

With this representation, an array can be maintained to hold the item and its priority value. The element will be inserted at the REAR end as usual. The deletion operation will then be performed in either of the two following ways:

(a) Starting from the FRONT pointer, traverse the array for an element of the highest priority. Delete this element from the queue. If this is not the front-most element, shift all its trailing elements after the deleted element one stroke each to fill up the vacant position(see figure) This implementation, however, is very inefficient as it involves searching the queue for the highest priority element and shifting the trailing elements after the deletion. A better implementation is as follows:



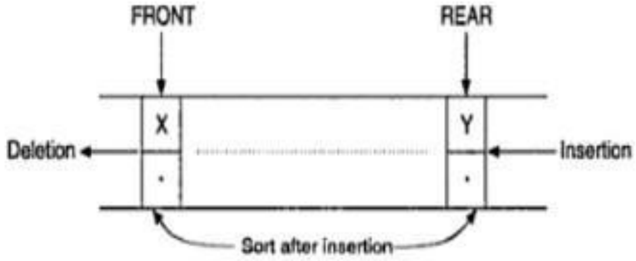


Figure 5.12 Deletion operation in an array representation of a priority queue. **Figure 5.13** Another array implementation of a priority queue.

(b) Add the elements at the REAR end as earlier. Using a stable sorting algorithm", sort the elements of the queue so that the highest priority element is at the FRONT end. When a deletion is required, delete it from the FRONT end only (see Figure).

The second implementation is comparatively better than the first one; here the only burden is to sort the elements.

Multi-queue implementation

This implementation assumes N different priority values. For each priority Pi there are two pointers Fi and Ri corresponding to the FRONT and REAR pointers respectively. The elements between Fi and Ri are all of equal priority value Pi. Figure represents a view of such a structure.

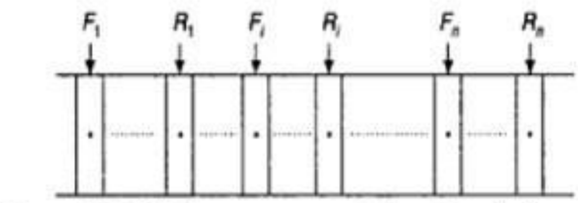
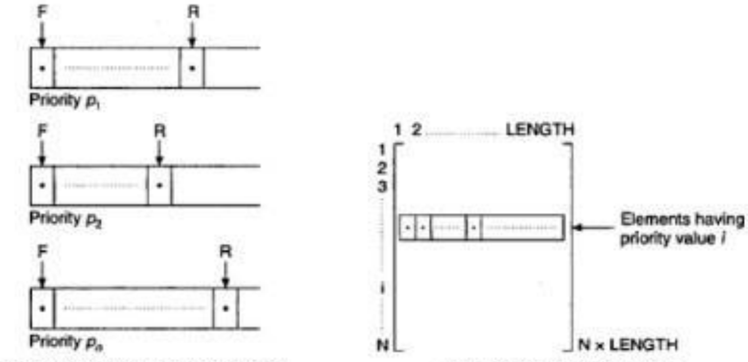


Figure 5.14 Multi-queue representation of a priority queue.



(a) Multiple queue with simple queues

(b) Multiple queue with a matrix

Figure 5.15 Multi-queue implementation with multiple simple queues and matrix.

With this representation, an element with priority value Pi will consult Fi for deletion and Ri for insertion.
But this implementation is associated with a number of difficulties:

(i) It may lead to a huge shifting in order to make room for an item to be inserted.

(ii) A large number of pointers are involved when the range of priority values is large.

In addition to the above, there are two other techniques to represent a multi-queue, which are shown in Figures (a) and (b). It is clear from Figure (a) that for each priority value a simple queue is to be maintained. An element will be added into a particular queue depending on its priority value. The priority queue as shown in Figure (b) is in some way better than the multi-queue with multiple queues. Here one can get rid of maintaining several pointers for FRONT and REAR in several queues. A multi-queue with multiple queues has one advantage that one can have different queues of arbitrary length. In some applications, it is seen that the number of occurrences of elements with some priority value is much larger than the other value, thus demanding a queue of larger size.

Linked list representation of a priority queue

This representation assumes the node structure as shown in Figure. LLINK and RLINK are two usual link fields, DATA is to store the actual content and PRIORITY is to store the priority value of the item. We will consider FRONT and REAR as two pointers pointing the first and last nodes in the queue, respectively. Here all the nodes are in sorted order according to the priority values of the items in the nodes. The following is an instance of a priority queue.

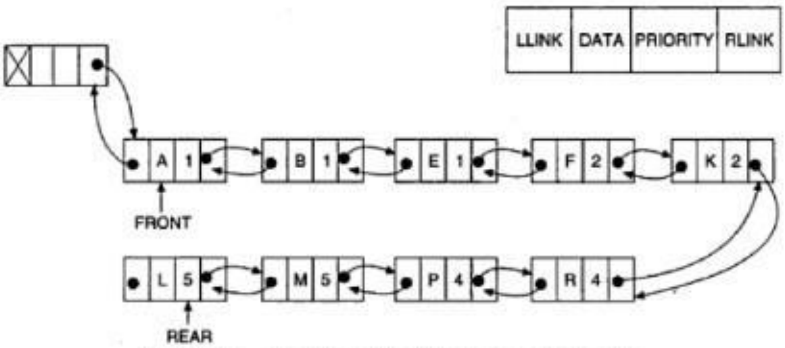


Figure 5.16 Linked list representation of a priority queue.

With this structure, to delete an item having priority p, the list will be searched starting from the node under pointer REAR and the first occurring node with PRIORITY = P will be deleted. Similarly, to insert a node containing an item with priority p, the search will begin from the node under the pointer FRONT and the node will be inserted before a node found first with priority value p, or if not found then before the node with the next priority value. The following two algorithms Insert PQ and Delete_PQ are used to implement the insertion and deletion operations on a priority queue.

5. APPLICATIONS OF QUEUES

Numerous applications of queue structures are known in computer science. One major application of queues is in simulation. Another important application of queues is observed in the implementation of various aspects of an operating system. A multiprogramming environment uses several queues to control various programs. Various scheduling algorithms are known to use varieties of queue structures.

5.2 CPU Scheduling in a Multiprogramming Environment

In a multiprogramming environment, a single CPU has to serve more than one program simultaneously. This section gives a brief idea about how queues are important to manage various programs in such an environment. Let us consider a multiprogramming environment where the possible jobs for the CPU are categoriz ed into three groups:

1. Interrupts to be serviced. A variety of devices and terminals are connected to the CPU and they may interrupt the CPU at any moment to get a particular service from it.
2. Interactive users to be serviced. These are mainly user's programs under execution at various terminals.
3. Batch jobs to be serviced.

Here the problem is to schedule all sorts of jobs so that the required level of performance of the environment will be attained. One way to implement complex scheduling is to classify the workload acc ording to its characteristics and to maintain separate process queues. So far as the environment is concerned, we can maintain three queues, as depicted in Figure. This approach is often called multi-level queues scheduling. Processes will be assigned to their respective queues. The CPU will then service the processes as per the priority of the queues. In the case of a simple strategy, absolute priority, the process from the highest priority queue (for example, system processes) are serviced until the queue becomes empty. Then the CPU switches to the queue of interactive processes which has medium priority, and so on. A lower priority process may, of course, be pre-empted by a higher-priority arrival in one of the upper level queues.

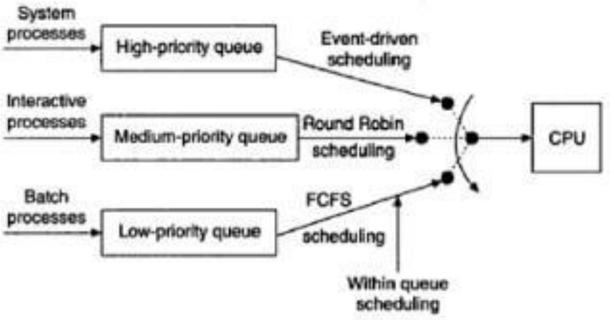


Figure 5.20 Process scheduling with multi-level queues.

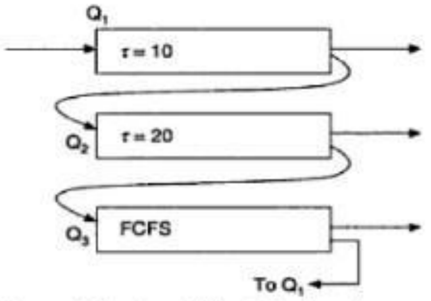


Figure 5.21 A multi-level feedback queue.

Multi-level queues strategy is a general discipline but has some drawbacks. The main drawback is that when processes arriving in higher-priority queues are very high, the processes in a lower-priority queue may starve for a long time. One way out to solve this problem is to time slice between the queues. Each queue gets a certain portion of the CPU time. Another possibility is known as multi-level feedback queue strategy. Normally in multi-level queue strategy, as we have seen, processes are permanently assigned to a queue upon entry to the system and processes do not move between queues. The multi-level feedback queue strategy, on the contrary, allows a process to move between queues. The idea is to separate out the processes with different CPU burst characteristics. If a process uses too much of CPU time (that is, long run process), it will be 'moved to a lower-priority queue. Similarly, a process which is waiting for too long a time in a lower-priority queue, may be moved to a higher-priority queue. For example, consider a multi-level feedback queue strategy with three queues Q1, Q2 and Q3 (Figure 5.21).

A process entering the system is put in queue Ql. A process in Ql is given a time quantum λ of 10 ms, say. If it does not finish within this time, it is moved to the tail of queue Q2. If Ql is empty, the process at the front of queue Q2 is given a time quantum ґ of 20 ms, say. If it does not complete within this time quantum, it is pre-empted and put into queue Q3. Processes in queue Q3 are serviced only when queues Ql and Q2 are empty. Thus, with this strategy, the CPU first executes all processes in queue Q1. Only when Q1 is empty it will execute all processes in queue Q2. Similarly, processes in queue Q3 will only be executed if only queues Ql and Q2 are empty. A process which arrives in queue Q1 will prempt a process in queue Q2 or Q3.

It can be observed that this strategy gives the highest priority to any process with a CPU burst of 10 ms or less. Processes which need more than 10 ms, but less than or equal to 20 ms are also served quickly, that is, they get the next highest priority over the shorter processes. Longer processes automatically sink to queue Q3; from Q3, processes will be served on a first- come first-serve (FCFS) basis and in the case of a process waiting for too long a time (as decided by the scheduler) it may be put into the tail of queue Q1.

5.3 Round Robin Algorithm

The round robin (RR) algorithm is a well-known scheduling algorithm and is designed especially for time sharing systems. Here, we will see how a circular queue can be used to implement such an algorithm. Before going to implement the RR algorithm, we should first describe the algorithm with illustration. Suppose, there are n processes P1, P2, ... , P n required to be served by the CPU. Different processes require different execution times. Suppose, the sequence of processes' arrivals according to their subscripts, that is, PI comes before P_2 and, in general, Pi comes after P_{i-1} for 1 <i<=n. The RR algorithm first decides a small unit of time, called a time quantum or time slice, λ. A time quantum is generally from 10 to 100 milliseconds. The CPU starts service with P1. P1 gets the CPU for time λ; afterwards the CPU switches to P2, and so on. When the CPU reaches the end of time quantum of Pn it returns to P1 and the same process will be repeated. Now, during time sharing, if a process finishes its execution before the finishing of its time quantum, the process then simply releases the CPU and the next process in waiting will get the CPU immediately. The total CPU time required is 30 unit. Let us assume a time quantum of 4 unit. The RR scheduling for this will be as shown in Figure.

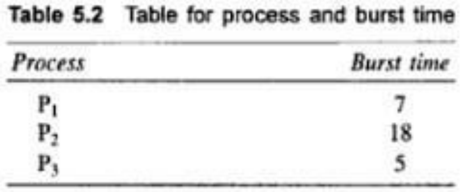


Table 5.2 Table for process and burst time

Process	*Burst time*
P_1	7
P_2	18
P_3	5

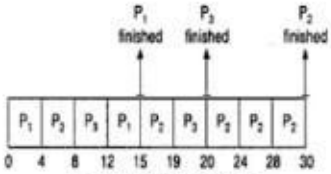
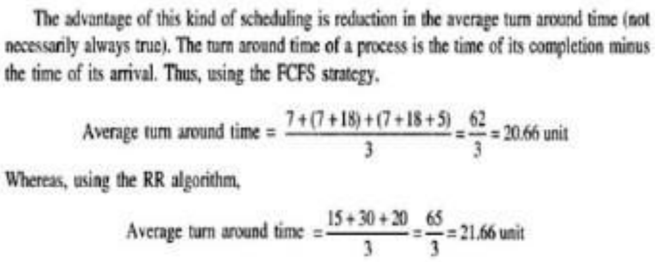


Figure 5.22 RR scheduling.



The advantage of this kind of scheduling is reduction in the average turn around time (not necessarily always true). The turn around time of a process is the time of its completion minus the time of its arrival. Thus, using the FCFS strategy,

$$\text{Average turn around time} = \frac{7+(7+18)+(7+18+5)}{3} = \frac{62}{3} = 20.66 \text{ unit}$$

Whereas, using the RR algorithm,

$$\text{Average turn around time} = \frac{15+30+20}{3} = \frac{65}{3} = 21.66 \text{ unit}$$

See the result by repeating the calculations but using the sequence of processes as P_2, P1 and P_3. In time sharing systems any process may arrive at any instant of time. Generally, all the processes currently under execution are maintained in a queue. When a process finishes its execution it is deleted from the queue and whenever a new process arrives it is inserted at the tail of the queue and waits for its turn. To illustrate this, let us consider Table 5.3.

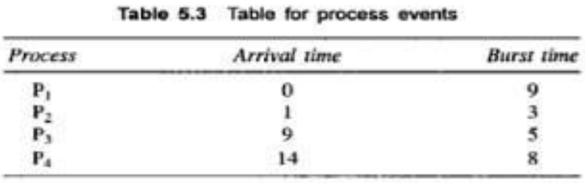


Table 5.3 Table for process events

Process	*Arrival time*	*Burst time*
P_1	0	9
P_2	1	3
P_3	9	5
P_4	14	8

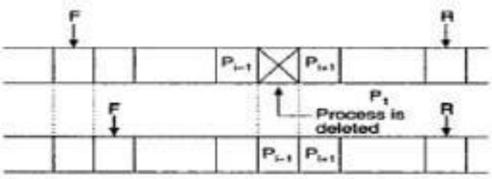


Figure 5.24 Deletion of a process from a circular queue.

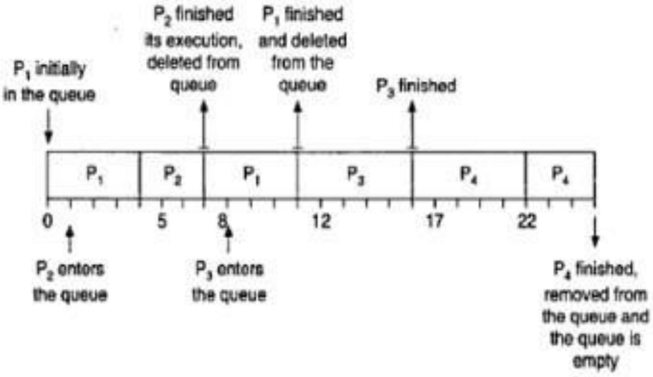

P_1 initially in the queue
P_2 finished its execution, deleted from queue
P_1 finished and deleted from the queue
P_3 finished
P_1
P_2
P_1
P_3
P_4
P_4
0
5
8
12
17
22
P_2 enters the queue
P_3 enters the queue
P_4 finished, removed from the queue and the queue is empty

Figure 5.23 In and out in a queue during RR scheduling.

The total CPU time required is 25 units. Let the time quantum be t = 5 unit. Figure 5.23 illustrates the snapshot at various instants with RR scheduling. Now let us discuss the implementation of the RR scheduling algorithm. A circular queue is the best choice for it. It may be noted that it is not strictly a circular queue, because here a process upon completion is deleted from the queue and it is not necessarily from the front of the queue rather it can be from any position of the queue. Except this, RR scheduling follows all the properties of a queue, that is, the process which comes first gets its turn first. The implementation of the RR algorithm using a circular queue is straightforward. Here, we use a variable sized circular queue; the size of the queue at any instant is decided by the number of processes in execution at that instant. Another mechanism is necessary; whenever a process is deleted, to fill the space of the deleted process, it is required to squeeze all the processes preceding to it, starting from the front pointer(Figure5.24).

6. HASH TABLE

There are other types of tables which help us to retrieve information very efficiently. The ideal hash table is merely an array of some constant size; the size depends on the application where it will be used. The hash table contains key values with pointers to the corresponding records. The basic idea of a hash table is that we have to place a key value into a location in the hash table; the location will be calculated from the key value itself. This one-to-one correspondence between a key value and an index in the hash table is known as address calculation indexing or more commonly hashing. In the present section, we will discuss hashing techniques and their related issues.

6.1 Hashing Techniques

The main idea behind any hashing technique is to find a one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in Figure, where K denotes a set of key values, I denotes a range of indices and H denotes the mapping function from K to I.

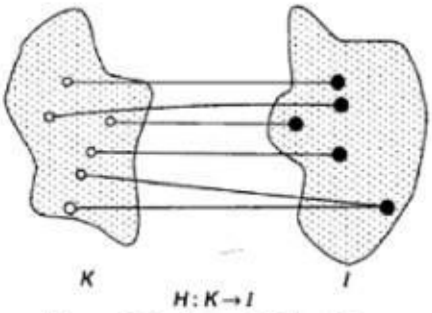


Figure 6.6 Concept of hashing.

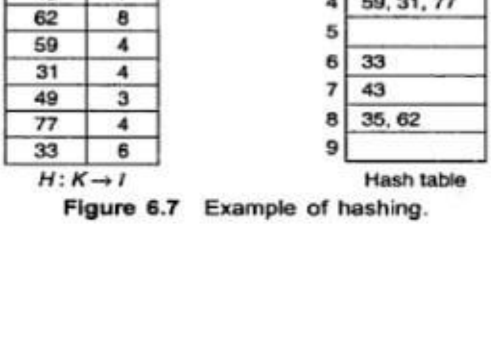
It may be noted that the mapping is subjective, that is all key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the hash function. A particular hashing technique uses a particular hash function. The hash function plays a dominant role in hashing techniques. There are two principal criteria in deciding a hash function H:K ~ I as follows:

1. The function H should be very easy and quick to compute.
2. The function H should as far as possible give two different indices for two different key values.

As an example, let us consider a hash table of size 10 whose indices are 0, 1, 2, ... , 8, 9. Suppose a set of key values are: 10, 19, 35, 43, 62, 59, 31, 49, 77, 33. Let us assume the hash function H is as stated below:

∑ Add the two digits in the key.
∑ Take the digit at the unit place of the result as the index; ignore the digit at the tenth place, if any.

Using this hash function, the mappings from key values to indices and to hash table are shown in Figure. In this example, for the given set of key values, the hash function does not distribute them uniformly over the hash table; some entries are there which are empty, and in some entries more than one key value needs to be stored. Allotment of more than one key value in one location in the hash table is called collision. We have found three collisions for 62, 31 and 77 in the above-mentioned example. It can be noted that |K| = |I|, that is, the number of key values is the same as the size of the hash table, but this is not the case always. In general, |K| > | I |. The following are some hash functions which are very common and popularly applied in various applications.



K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

H: K→I

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

Figure 6.7 Example of hashing.

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h . larger than the number N of keys in K. The hash function H is then defined by

H(k) = k(MOD h) if indices start from 0
H(k) = k(MOD h) + 1 if indices start from 1

where k € K, a key value. The operator MOD defines the modulo arithmetic operation, which is equal to the remainder of dividing k by h. For example, if k = 31 and h = 13 then

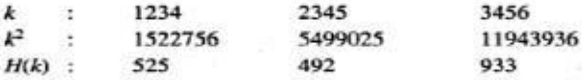
H(31) = 31(MOD 13) = 5
or
H(31) = 31(MOD 13) + 1 = 6

The number h is usually chosen to be a prime number or a number without small divisors, since this usually minimizes the number of collisions. Generally, h is a prime number and equal to the size of the hash table.

Midsquare method

Another hash function which has been widely used in many applications is the midsquare method. The method is defined as follows:

The hash function H is defined by H(k) = x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. This selection usually depends on the size of the hash table. It needs to be emphasized that the same criteria should be used for selecting the bits or digits for all of the keys. As an example, suppose the key values are of the integer type, and we require 3-digit addresses. Our selection criteria are to select 3 digits at even positions starting from the right- most digit in the square. Let us see the address calculations, for 3 distinct keys and with the hash function, as defined above:



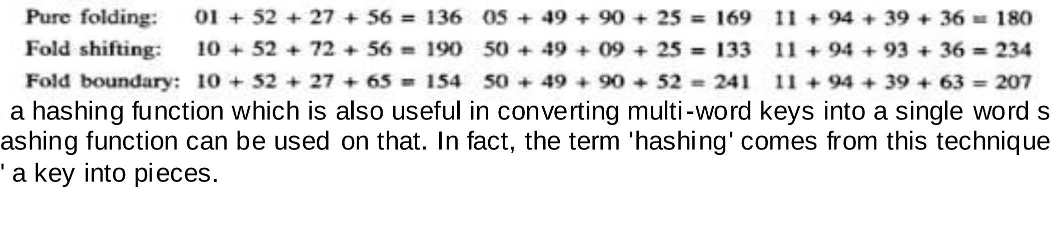
k	:	1234	2345	3456
k^2	:	1522756	5499025	11943936
$H(k)$	:	525	492	933

Here, we observe that the second, the fourth, and the sixth digits, counting from the right, are chosen for the hash addresses. The midsquare method has been criticized because of time-consuming computation (multiplication operation), but it usually gives good results so far as the uniform distribution of the keys over the hash table is concerned.

Folding method

Another fair method for a hash function is the folding method. The method can be defined as follows: Partition the key k into a number of parts kb k2' ... , kno where each part, except possibly the last, has the same number of bits or digits as the required address width. Then the parts are added together, ignoring the last carry, if any. Alternatively, H(k) = k1 + k2 + ... + k_n

where the last carry, if any, is ignored. If the keys are in binary form, the exclusive-OR operation may be substituted for addition. There are many variations known in this method. One is called the fold shifting method, where the even number parts, k2, k4, ... are each reversed before the addition. Another variation is called the fold boundary method. Here, two boundary parts, namely, k1 and kn, each are reversed and then added to all other parts. As an example, let us take the size of each part to be 2; the following calculations are performed on the given key values (integers) as shown below.

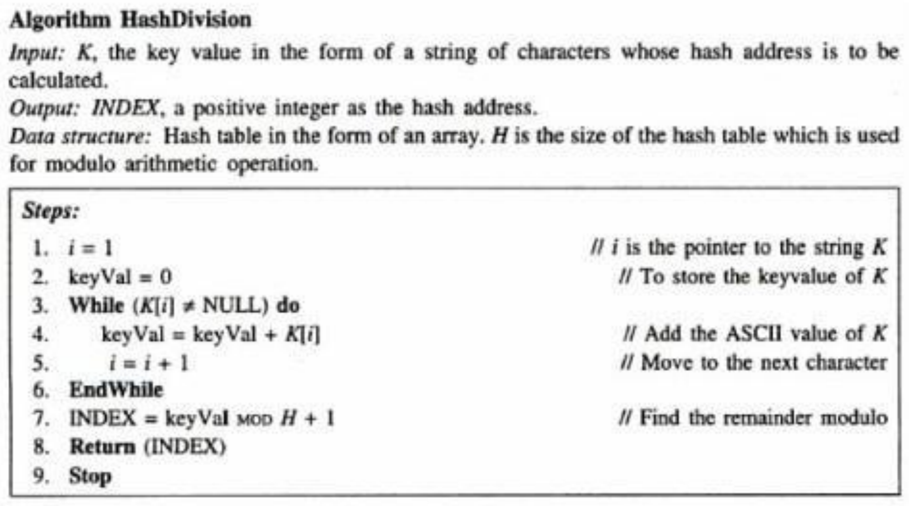


k:	1522756	5499025	11943936
Chopping:	01 52 27 56	05 49 90 25	11 94 39 36
Pure folding:	01 + 52 + 27 + 56 = 136	05 + 49 + 90 + 25 = 169	11 + 94 + 39 + 36 = 180
Fold shifting:	10 + 52 + 72 + 56 = 190	50 + 49 + 09 + 25 = 133	11 + 94 + 93 + 36 = 234
Fold boundary:	10 + 52 + 27 + 65 = 154	50 + 49 + 90 + 52 = 241	11 + 94 + 39 + 63 = 207

Folding is a hashing function which is also useful in converting multi-word keys into a single word so that another hashing function can be used on that. In fact, the term 'hashing' comes from this technique of 'chopping' a key into pieces.

Digit analysis method

The basic idea of this hashing function is to form hash addresses by extracting and/or shifting the extracted digits or bits of the original key. As an example, given a key value, say 6732541, it can be transformed to the hash address 427 by extracting the digits in even positions and then reversing this combination. For a given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and then rearrangement is based on some analysis. To do this, an analysis is performed to determine which key positions should be used in forming hash addresses. For each criterion, hash addresses are calculated and then a graph is plotted, then that criterion is selected which produces the most uniform distribution, that is with the smallest peaks and valleys. This method is particularly useful in the case of static files where the key values of all the records are known in advance. We have assumed the key values as integers in our previous discussions, but it need not be so always. In fact, any key value can be represented by a string of characters and then ASCII values of its constituent characters can be taken to convert it into a numeric value. Thus, assuming that a key value k = k1k2k3 ... $k_m$where each ki is the constituent character in k. The hash function using the division method is stated as below in algorithm HashDivision.



Algorithm HashDivision

Input: *K*, the key value in the form of a string of characters whose hash address is to be calculated.

Output: *INDEX*, a positive integer as the hash address.

Data structure: Hash table in the form of an array. *H* is the size of the hash table which is used for modulo arithmetic operation.

Steps:

```
i = 1                                // i is the pointer to the string K
keyVal = 0                           // To store the keyvalue of K
While (K[i] ≠ NULL) do
    keyVal = keyVal + K[i]           // Add the ASCII value of K
    i = i + 1                        // Move to the next character
EndWhile
INDEX = keyVal MOD H + 1             // Find the remainder modulo
Return (INDEX)
Stop
```

6.2 Collision Resolution Techniques

Whatever the hash function used in hashing, the complete removal of collisions is almost impossible. This can be emphasized with an example called birth day surprise. Suppose there is a class of 24 students and they. are having the same year of birth. We want to know the probability that two students have the same date of birth. The probability can be calculated as follows:

Open the calendar of the year of their birth. Assume that there are 365 days. Start with any student, and put a tick on his birthday date on the calendar. Now, the probability that the second student has a different birthday is 364/365. Tick this date off. The probability that a third student has a different birthday is now 363/365. Continuing this way, we see that if the first (n - 1) students have different birthdays, then the probability that the nth student has a different birthday is

$$\frac{365-(n-1)}{365} \quad \text{or} \quad \frac{365-n+1}{365}$$

Since the birthdays of different people are independent, we obtain the probability that n students all have a different birthday is

$$\frac{364}{365}\times\frac{363}{365}\times\frac{362}{365}\times\cdots\times\frac{365-n\times 1}{365}$$

This probability can be calculated as less than 0.5 whenever n>= 24.

In other words, suppose there is a hash table of size 365 and we want to store the records of all the 24 students based on birthdays as their key values. It is therefore a fifty-fifty chance that two of the students have the same birthday and hence a collision. So, collision in hashing cannot be ignored, whatever be the size of the hash table. The next question arises therefore is what to do if there is a collision? There are several techniques to resolve the collisions. Two important methods are listed below:

(a) Closed hashing (also called linear probing)
(b) Open hashing (also called chaining).

6.3 Closed Hashing

The simplest method to resolve a collision is closed hashing. Suppose there is a hash table of size h and the key value of interest is mapped to an address location i, with a hash function. The closed hashing then can be stated as follows:

Start with the hash address where the collision has occurred, let it be i. Then follow the following sequence of locations in the hash table and do the sequential search. i, i + 1, i + 2, ... , h, 1, 2, ... , i-1 The search will continue until anyone of the following cases occurs:

∑ The key value is found.
∑ An unoccupied (or empty) location is encountered.
∑ The searches reaches the. location where the search had started.

The first case corresponds to the successful search and the last two cases correspond to unsuccessful search. Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is also alternatively termed linear probing; probe means key comparison. Let us illustrate the method with an example. Assume that there is a hash table of size 10 and the hash function uses the division method with remainder modulo 7, namely, H(k) = k MOD (7 + 1). Let us consider the build up of the hash table (initially, the table is empty) with the following set of key values: 15 11 25 16 9 8 12 8
The loading of the hash table will take place successively by performing a search for a key and inserting it into the table in an empty room if the key is not in the table and leaving if it is overflow, that is, no free room to accommodate any further key value. This is illustrated in Figure.

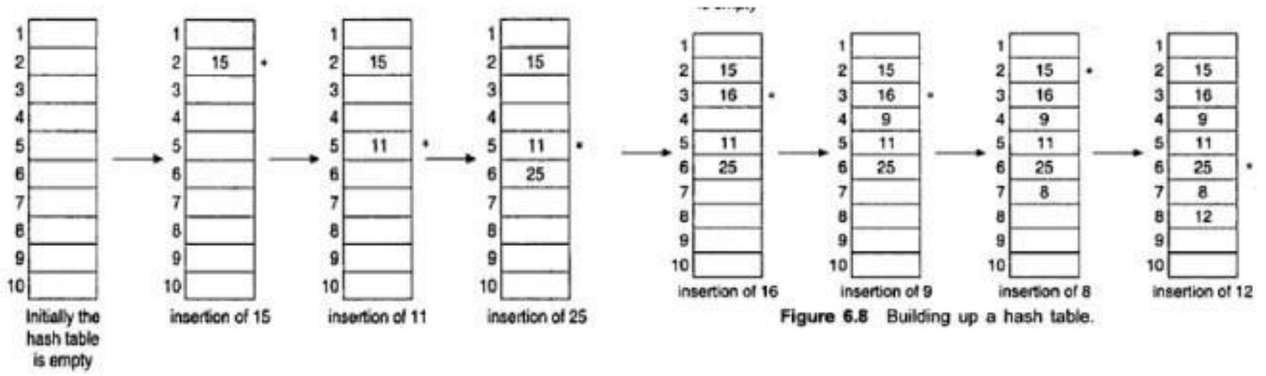
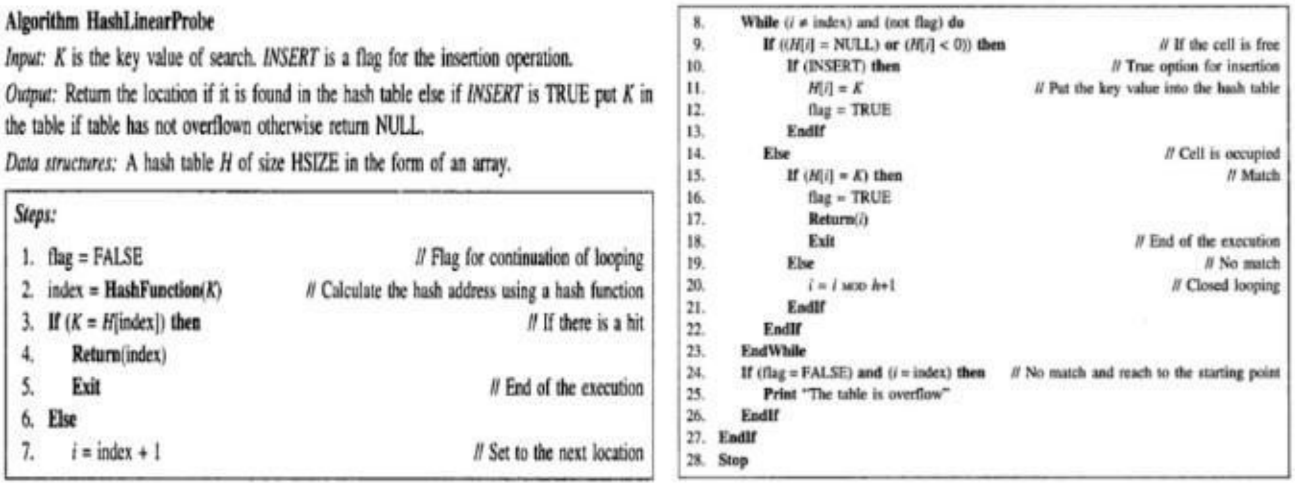


Figure 6.8 Building up a hash table.

Next, let us define the operation for searching a key-value and inserting a key-value. The algorithm HashLinearProbe for searching a key value K in a hash table of size HSIZE is given below:



Algorithm HashLinearProbe

Input: *K* is the key value of search. *INSERT* is a flag for the insertion operation.

Output: Return the location if it is found in the hash table else if *INSERT* is TRUE put *K* in the table if table has not overflown otherwise return NULL.

Data structures: A hash table *H* of size HSIZE in the form of an array.

```
Steps:
1.  flag = FALSE                              // Flag for continuation of looping
2.  index = HashFunction(K)                   // Calculate the hash address using a hash function
3.  If (K = H[index]) then                    // If there is a hit
4.      Return(index)
5.      Exit                                  // End of the execution
6.  Else
7.      i = index + 1                         // Set to the next location
8.      While (i ≠ index) and (not flag) do
9.          If ((H[i] = NULL) or (H[i] < 0)) then      // If the cell is free
10.             If (INSERT) then                       // True option for insertion
11.                 H[i] = K                           // Put the key value into the hash table
12.                 flag = TRUE
13.             EndIf
14.         Else                                       // Cell is occupied
15.             If (H[i] = K) then                     // Match
16.                 flag = TRUE
17.                 Return(i)
18.                 Exit                               // End of the execution
19.             Else                                   // No match
20.                 i = i MOD h+1                      // Closed looping
21.             EndIf
22.         EndIf
23.     EndWhile
24.     If (flag = FALSE) and (i = index) then   // No match and reach to the starting point
25.         Print "The table is overflow"
26.     EndIf
27. EndIf
28. Stop
```

Note Step 9 in the above algorithm. Here, we assume that whenever a key value is deleted from the hash table its corresponding entries are made negative instead of NULL. Writing an algorithm for deleting a key value is straightforward and is left as an exercise.

Drawback of closed hashing and its remedies
The major drawback of closed hashing is that, as half of the hash table is filled, there is a tendency towards clustering; that is key values are clustered in large groups and as a result a sequential search becomes slower and slower. This kind of clustering is typically known as primary clustering. The following are some solutions known to avoid this situation:

(a) Random probing (b) Double hashing or rehashing (c) Quadratic probing.

Random probing: This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in the linear probing method. The random sequence generated by the pseudo random number generator contains all the positions between 1 and h, the

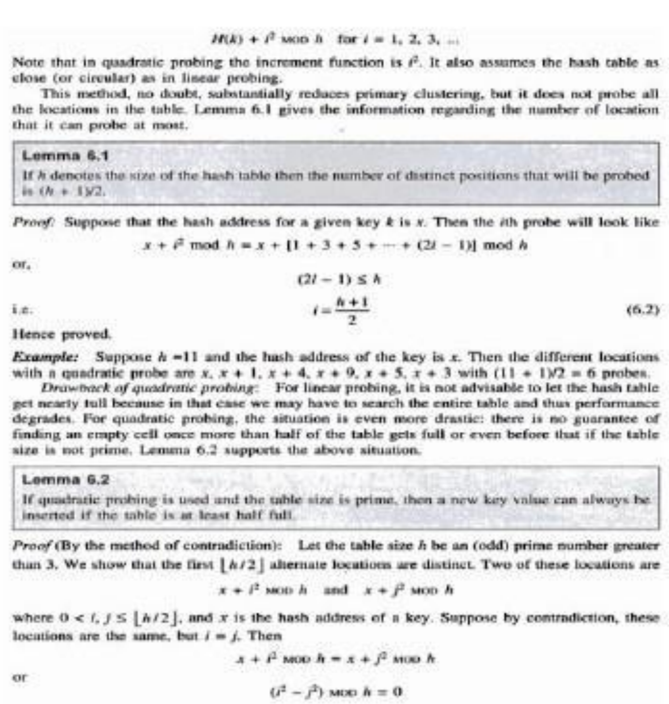
highest location of the hash table. An example of a pseudo random number generator that produces such a random sequence of locations is given below: i = (i + m) MOD h + 1
where i is a number in the sequence, and m and h are integers that are relatively prime to each other (that is, their greatest common divisor is 1). For example, suppose m = 5 and h = 11 and itially i = 2, then the above-mentioned pseudo random number generator generates the sequence as: 8, 3, 9, 4, 10, 5, 11, 6, 1, 7, 2
We stop producing the numbers when the first location is duplicated. Observe that here all the numbers between 1 and 11 are generated but randomly. We can avoid primary clustering if the probe follows the said random sequence.

Double hashing: Random hashing however is not free form clustering. Another type of clustering, called secondary clustering, is involved here. In particular, clustering occurs when two keys are hashed into the same location. In such an instance, if the same sequence of locations is generated for two different keys by the random probing method then clustering takes place. An alternative approach to avoid the secondary clustering problem is to use a second hash function in addition to the first one. This second hash function results in the value of m for the pseudo random number generator as employed in the random probing method. This second function should be selected in such a way that the hash addresses generated by the two hash functions are distinct and the second function generates a value m for the key k so that m and h are relatively prime. Let us consider the following example.
Suppose $H_1(k)$ is the initially used hash function and $H_2(k)$ is the second one. These two functions are defined as

$$H_1(k) = (k \text{ MOD } h) + 1$$
$$H_2(k) :::: (k \text{ MOD } (h - 4)) + 1$$

Let h = 11 and k = 50 for an instance. Then, $H_1(50) = 7$ and $H_2(50) = 2$. Therefore, $H_1(50)!=H_2(50)$, that is, H_1 and H_2 are independent and m = 2, h = 11 are relatively prime. Hence, using i = [(i + 2) MOD 11] + 1, and initially i = 7, we have the random sequence as 10, 2, 5, 8, 11, 3, 6, 9, 1, 4, 7
Now, let us choose another key value which has the same hash address as that of 50 (that is, 7) with the first hash function H1. Let it be 28 (since $H_1(28)$ = 28 MOD 11 + 1 = 7). Then $H_2(28)$ = 28 MOD 7 + 1 = 5
So using i = [(i + m) MOD 11] with i = 7 and m = 5, we get the sequence: 2, 8, 3, 9, 4, 10, 5, 11, 6, 1, 7
Thus, for the two key values where the hash address is the same and using rehashing, two different random sequences are generated, thereby alleviating the secondary clustering.

Quadratic probing: Quadratic probing is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations i + 1, i + 2, i + 3, etc. are probed; but in quadratic probing, the next locations to be probed are i + 1_2, i + 2_2, i + 3_2, etc. Mathematically, if h is the size of the hash table and H(k) is the hash function then the quadratic probing searches the locations:



$$H(k) + i^2 \text{ MOD } h \quad \text{for } i = 1, 2, 3, \ldots$$

Note that in quadratic probing the increment function is i^2. It also assumes the hash table as close (or circular) as in linear probing.

This method, no doubt, substantially reduces primary clustering, but it does not probe all the locations in the table. Lemma 6.1 gives the information regarding the number of location that it can probe at most.

Lemma 6.1

If h denotes the size of the hash table then the number of distinct positions that will be probed is $(h + 1)/2$.

Proof: Suppose that the hash address for a given key k is x. Then the ith probe will look like

$$x + i^2 \bmod h = x + [1 + 3 + 5 + \cdots + (2i - 1)] \bmod h$$

or,

$$(2i - 1) \leq h$$

i.e.

$$i = \frac{h+1}{2} \qquad (6.2)$$

Hence proved.

Example: Suppose $h = 11$ and the hash address of the key is x. Then the different locations with a quadratic probe are x, $x + 1$, $x + 4$, $x + 9$, $x + 5$, $x + 3$ with $(11 + 1)/2 = 6$ probes.

Drawback of quadratic probing: For linear probing, it is not advisable to let the hash table get nearly full because in that case we may have to search the entire table and thus performance degrades. For quadratic probing, the situation is even more drastic: there is no guarantee of finding an empty cell once more than half of the table gets full or even before that if the table size is not prime. Lemma 6.2 supports the above situation.

Lemma 6.2

If quadratic probing is used and the table size is prime, then a new key value can always be inserted if the table is at least half full.

Proof (By the method of contradiction): Let the table size h be an (odd) prime number greater than 3. We show that the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Two of these locations are

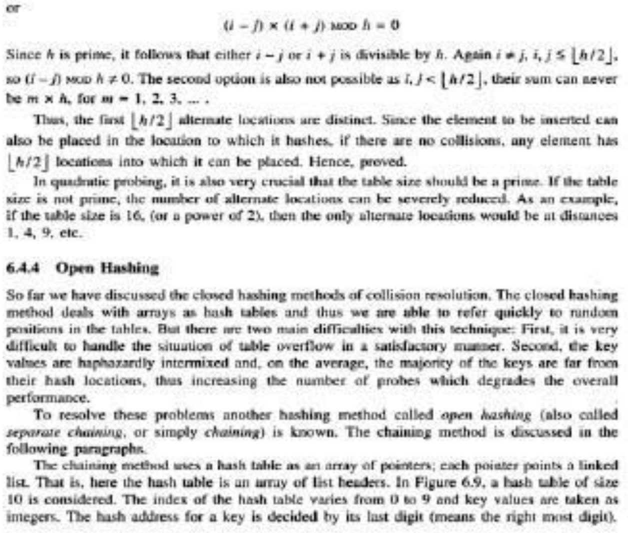
$$x + i^2 \text{ MOD } h \quad \text{and} \quad x + j^2 \text{ MOD } h$$

where $0 < i, j \leq \lfloor h/2 \rfloor$, and x is the hash address of a key. Suppose by contradiction, these locations are the same, but $i = j$. Then

$$x + i^2 \text{ MOD } h = x + j^2 \text{ MOD } h$$

or

$$(i^2 - j^2) \text{ MOD } h = 0$$



or

$$(i - j) \times (i + j) \text{ MOD } h = 0$$

Since h is prime, it follows that either $i - j$ or $i + j$ is divisible by h. Again $i \neq j$, $i, j \leq \lfloor h/2 \rfloor$, so $(i - j) \text{ MOD } h \neq 0$. The second option is also not possible as $i, j < \lfloor h/2 \rfloor$, their sum can never be $m \times h$, for $m = 1, 2, 3, \ldots$.

Thus, the first $\lfloor h/2 \rfloor$ alternate locations are distinct. Since the element to be inserted can also be placed in the location to which it hashes, if there are no collisions, any element has $\lfloor h/2 \rfloor$ locations into which it can be placed. Hence, proved.

In quadratic probing, it is also very crucial that the table size should be a prime. If the table size is not prime, the number of alternate locations can be severely reduced. As an example, if the table size is 16, (or a power of 2), then the only alternate locations would be at distances 1, 4, 9, etc.

6.4.4 Open Hashing

So far we have discussed the closed hashing methods of collision resolution. The closed hashing method deals with arrays as hash tables and thus we are able to refer quickly to random positions in the tables. But there are two main difficulties with this technique: First, it is very difficult to handle the situation of table overflow in a satisfactory manner. Second, the key values are haphazardly intermixed and, on the average, the majority of the keys are far from their hash locations, thus increasing the number of probes which degrades the overall performance.

To resolve these problems another hashing method called *open hashing* (also called *separate chaining*, or simply *chaining*) is known. The chaining method is discussed in the following paragraphs.

The chaining method uses a hash table as an array of pointers; each pointer points a linked list. That is, here the hash table is an array of list headers. In Figure 6.9, a hash table of size 10 is considered. The index of the hash table varies from 0 to 9 and key values are taken as integers. The hash address for a key is decided by its last digit (means the right most digit).

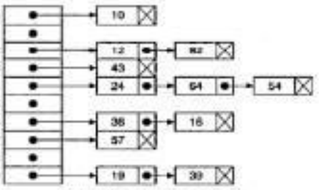


Figure 6.9 An open hashing.

For a given key value, the hash address is calculated. It then searches the linked list pointed by the pointers at that location. If the element is found it returns the pointer to the node containing that key value else inserts the element at the end of that list. The implementation of open hashing is stated in the algorithm *HashChaining* as follows:

Algorithm HashChaining

Input: K is the item of interest. *INSERT* is a flag for the option of insertion.
Output: If K is found in the hash table then return the pointer of the node which contains the key value K else insert K into the linked list when the *INSERT* flag is TRUE.
Data structure: Hash table H having size *HSIZE* storing pointer to the single linked list structure.

```
Steps:
index = HashFunction(K)                          // Calculate the hash address of K
ptr = H[index]                                   // ptr is a pointer to any node in the list
flag = FALSE                                     // flag for controlling the search
While (ptr ≠ NULL) and (flag = FALSE) do
    If (ptr→DATA = K) then                       // End of search
        flag = TRUE
        Return(ptr)
        Exit                                     // End of execution
    Else
        ptr = ptr.LINK                           // Move to the next node
    EndIf
EndWhile
If (flag = FALSE) then
    Print "Key value does not exist"
    If (INSERT) then
        InsertEnd_SL(H[index])                   // Insert it into the table
    EndIf
EndIf
Stop
```

A key value if it exist can be deleted from a hash table for which a procedure *HashKeyDelete(...)* can be written. This is left as an exercise for the reader.

Advantages and disadvantages of chaining

There are several advantages of the chaining method. The most important advantages are stated below:

1. An overflow situation never arises. The hash table maintains lists which can contain any number of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys, so that keys can be searched quickly.
3. Insertion and deletion become a quick and an easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in a single linked list.
4. Finally, open hashing is best suitable in applications where the number of key values varies drastically as open hashing uses dynamic storage management policy.

The only disadvantage of the chaining method is that of maintaining linked lists and extra storage space for link fields.

6.4.5 Comparison of Collision Resolution Techniques

We will conclude the discussion of hash tables by giving an analytical comparison of various collision resolution techniques discussed. Let us define the *load factor*, λ, of a hash table as

$$\lambda = \frac{\text{Total number of key values}}{\text{Size of the hash table}} \tag{6.3}$$

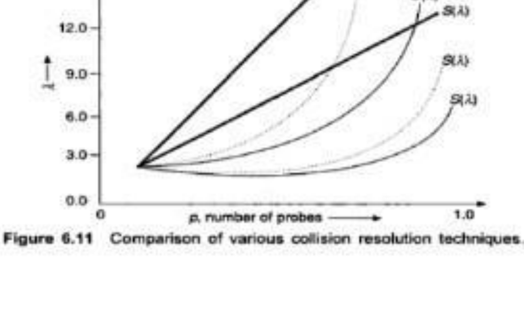


Figure 6.11 Comparison of various collision resolution techniques.

So $\lambda = 1.0$ means that the number of key values is the same as the total capacity of the hash table. We also define $S(\lambda)$ and $U(\lambda)$ as

$S(\lambda)$ = average number of probes for a successful search.
$U(\lambda)$ = average number of probes for an unsuccessful search.

These two quantities will measure the performance of collision resolution methods.

Analysis of closed hashing

To analyze the performance of closed hashing, let us assume the case of random probing and ignore the problem of clustering for the sake of simplicity.

Let us first consider the case of unsuccessful search. It is evident that the probability that the first probe hits an occupied cell is λ, the load factor. The probability that a probe hit an empty cell is $1 - \lambda$. The probability that the unsuccessful search terminates in exactly two probes is therefore $\lambda(1 - \lambda)$. Arguing similarly this way, the probability that exactly k probes are made in an unsuccessful search is $\lambda^{k-1}(1 - \lambda)$. The average number of probes for an unsuccessful search is therefore

$$U(\lambda) = \sum_{k=1}^{\infty} k\lambda^k(1-\lambda) = (1-\lambda)\sum_{k=1}^{\infty} k\lambda^k \tag{6.4a}$$

Since $\lambda \le 1$ and $\sum_{k=1}^{\infty} k\lambda^k = \frac{1}{(1-\lambda)^2}$, we have

$$U(\lambda) = (1-\lambda)\frac{1}{(1-\lambda)^2} = \frac{1}{1-\lambda} \tag{6.4b}$$

Next, let us consider the case of a successful search. We can think of this problem through insertion of key values. Then the number of probes required will be exactly one more than the number of probes made in the unsuccessful search before inserting the item. Let us consider the case when the table is initially empty. In that state, key values are inserted one at a time. Now as the items are inserted, the load factor grows slowly from 0 to λ. Thus, we can express the average number of probes in a successful search as

$$\begin{aligned} S(\lambda) &= \frac{1}{\lambda}\int_0^{\lambda} U(x)\,dx \\ &= \frac{1}{\lambda}\int_0^{\lambda}\frac{1}{1-x}\,dx \\ &= \frac{1}{\lambda}\ln\frac{1}{1-\lambda} \end{aligned} \tag{6.5}$$

A similar calculation can be performed for closed hashing with linear probing. This is left as an assignment for the student.

Analysis of open hashing

Let us recall the case of chaining. In chaining, we move to the linked list before doing any probes. Suppose that a list contains n key values. Assuming that the key values are equally probable in any list, the expected number of key values on any list is n/h, h being the size of the hash table. This is nothing but l, the load factor. Now, if the list contains n items, the number of key comparisons for an unsuccessful search is n. Thus, the average number of probes for an unsuccessful search is

$$U(\lambda) = \lambda \tag{6.7}$$

Now, suppose the search is successful. From the analysis of sequential search over a list of n items, we can write

$$\text{Number of comparisons} = \frac{1}{n}\sum_{i}^{n} i = \frac{n+1}{2} \tag{6.8}$$

Assume that an item is equally probable in any place. Since the average number of key values in any list is λ, the average number of probes in a successful search is

$$S(\lambda) = \frac{\lambda+1}{2} \tag{6.9}$$

We can draw several conclusions from the results thus obtained. Let us draw a graph (Figure 6.11) for these results. From this graph, the following points are evident:

1. Open hashing always requires fewer probes than closed hashing.
2. Chaining is especially advantageous when the load factor is significantly low.
3. With closed hashing and successful search, linear probing is not significantly slower if λ is high. For unsuccessful searches, however, clustering will occur which quickly degenerates into a long sequential search.

We might therefore conclude that if searches are quite likely to be successful and the load factor is moderate, closed hashing is quite satisfactory, but in other circumstances open hashing is promising.