gradeup

Data Structure Part-5

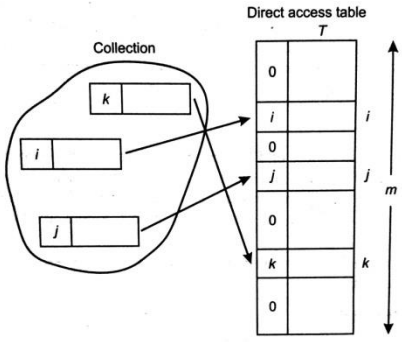
Data Structure Part - 5

Content:-

1. **Hash tables**
2. **Hashing**
3. **Linear Probing**
4. **Quadratic Probing**
5. **Double Hashing**
6. **Link List**
7. **Operation On Linked List**

HASH TABLES

It support one of the most efficient types of searching i.e. hashing. Fundamentally, hash table consists of an array in which the data is accessed via a special index known as **a key**. Primary idea behind a hash table is to establish a mapping between the set of all possible keys and positions in the array using hash function. A hash function accepts a key and returns its hash value or hash coding. Keys differ in type, but hash coding are integers. After all both calculating a hash value and indexing into an array can be performed in the constant time, the beauty of hashing is that one can use it to perform constant time searches. When a hash function can guarantee that no two keys will generate the same hash coding, then the resulting hash table is said to be directly addressed.



This is idea, but direct addressing is rarely possible in practice.

The number of entries in a hash table is little relative to the creation of possible keys. Consequently, most of the hash functions map some keys to same position in the table. When the two keys map to the same position, they collide. A good hash function minimizes collisions, but we must still be prepared to deal with them.

HASHING

It is a widely used class of data structures(DS) that support the operations of delete, insert and search on a dynamic set of keys S. the keys are drawn from a universe U(|U| >> |S|), which for our purposes will be a set of positive integers. These keys mapped into a hash table using a hash function; the size of hash table is usually within a constant factor of the number of elements in the current set S.

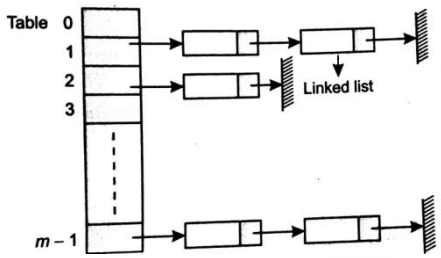
There are two main methods used to implement hashing : hashing with chaining and hashing with open addressing.

HASHING WITH CHAINING

In hashing with chaining, the elements in S are stored in a hash table T[0..m - 1] of size m, where m is somewhat largest than n, the size of S. Hash table is said to have m slots. Associated with hashing scheme is a hash function h which is a mapping from U to {0, .. m- 1}. Every key k ∈ S is stored in location T[h(k)], and we say that key k is hashed into slot h(k). If increased than one key in S hashes into the same slot, then we have a collision.

In such a case, all keys that hash into the same slot are placed in a linked list associated with the slot; this linked list is called the chain at that slot. The load factor of a hash table is defined to be α = n/m; it represents the average

number of keys per slot. We typically operate in the range m = θ(n), so α is usually a constant (usually α < 1). If a large number of insertions or deletions destroy this property, a more suitable value of m is chosen and the key are re-hashed into a new table with a new hash function.



In constant hashing we suppose that we have a 'good' hash function h such that any element in U is equally likely to hash into any of the m slots in T, free of the other keys in the table. We also suppose that h(k) can be computed in sustained time.

In hashing with chaining inserts and deletes can be accomplish in sustained time with a suitable linked list representation for the chains. In the defeat case, a search operation can take as long as n steps if all keys hash into the identical slot. However, if new assume simple uniform hashing then the expected time for a search operation is easily shown to be θ(α), which is a constant under the normal operation condition of m = θ (n). The main drawback with this method is the requirement that the scheme implements simple uniform hashing.

ANALYSIS OF HASHING WITH CHAINING

Specified a hash table T with m slots that stores n elements, we define the load factor α for T as n/m that is, the average number of elements stored in a chain. The worst-case time for searching is thus θ(n) plus the time to compute the hash function - no better than if we used one linked list for all the elements. Clearly, hash table are not used for their worst-case performance.

The mean performance of hashing turn on on how well the hash function h distributes the set of keys to be stored among the m slots, on the average.

Theorem

In a hash table in which collision are resolved by chaining, an unsuccessful search takes time $\theta(1 + \alpha)$ on the average, under the assumption of simple uniform hashing.

Example: Let us consider the insertion of elements 5, 28, 19, 15, 20, 33, 12, 17, 10 into a chained-hash table. Let us suppose the hash table has 9 slots and the hash function be h(k) = k mod 9.

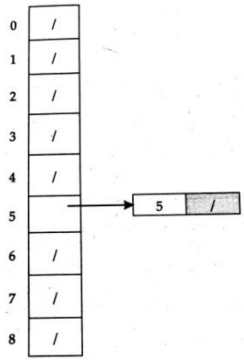
Solution: The initial state of the chained-hash table.

0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/

Insert 5:

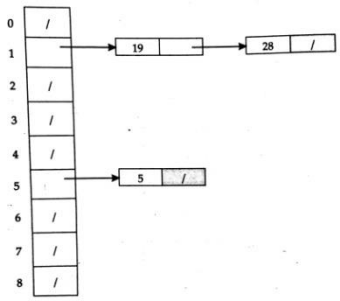
H(5) = 5 mod 9 = 5

Create a linked-list for T[5] and store value 5 in it.



Similarly insert 28. h(28) mod 9 = 1. Create a link-list for T[1] and store value 28 in it.

Now insert 19 h(19) = 19 mod 9 = 1. Insert value 19 in the slot T[1] in the beginning of the link-list.



Now insert 15. h(15) = 15 mod 9 = 6. Create a link list for T[6] and store value 15 in it.

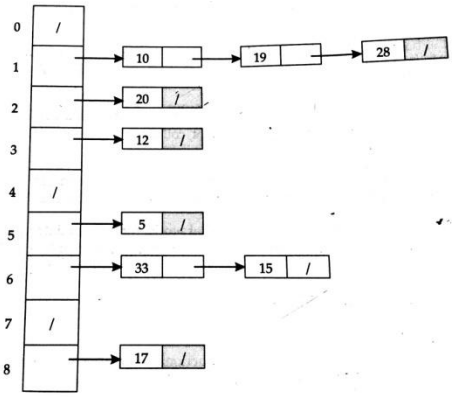
Similarly insert 20, h(20) = 20 mod 9 = 2 in T[2]. Insert 33, h(33) = 33 mod 9 = 6, in the beginning of the linked list for T[6]. Then,

Insert 12 i.e., h(12) = 12 mod 9 = 3 in T[3]

Insert 17 i.e., h(17) = 17 mod 9 = 8 in T[8]

Insert 10 i.e., h(10) = 10 mod 9 = 1 in T[1].

Thus the chained-hash-table after inserting key 10 is.



HASHING WITH OPEN ADDRESSING

All elements are stored in the hash table alone. i.e., every table entry carry either an element of the dynamic set or NIL. When finding for an element, we fully inspect table slots until the desired element is found or it is clear that the element is not in the table. Thus, the load factor α can never exceed 1.

The advantage of open addressing is that it avoid pointers altogether. Rather than following pointers, we calculate the sequence of slots to be examined. The extra memory release by not storing pointers provides the hash table with a increased number of slots for the same amount of memory, potentially yielding fewer collisions and faster retrieval.

The method of studying the locations in the hash table is called '**Probing'**

To execute insertion using open addressing, we continuously study, the hash table until we find an empty slot in which to put the key.

HASH-INSERT (T, k)

1. i ← 0
2. repeat j ← h(k, i)
3. if T[j] = NIL
4. then T[j] ← k
5. return j
6. else i ← i +1
7. until i = m
8. error "hash table overflow"

The algorithm for searching for key k probes the same sequence of slots that the insertion algorithm examined when key k was inserted. Therefore, the search can terminate (unsuccessfully) when it funds an empty slot, since k

would have been inserted there and not later in its probe sequence. (Note that this argument supports that keys are not deleted from the hash table). The method HASH-SEARCH takes as input a hash table T and a key k, returning j if slot j is found to contain key k, or NIL if key k is not present in table T.

HASH-SEARCH (T, K)

1. i ← 0
2. repeat j ← h (k, i)
3. if T[j] = j
4. then return j
5. i ← i + 1
6. until T[j] = NIL or i = m
7. return NIL

Deletion from an open-address hash table is difficult. When we delete a key from slot i, we cannot clearly mark that slot as empty by storing NIL in it. Doing so make it impossible to recover any key k during whose insertion we had probed slot i and found it occupied. One solution is to make the slot by storing in it the special value DELETED instead of NIL. We would then update the method HASH-SEARCH so that it keeps on looking when it sees the value DELELTED, while HASH-INSERT would treat such a slot as if it were empty so that a new key can be inserted. When we do this, though, the search times are no longer dependent on the load factor α, and for this reason chaining is more commonly selected as a collision resolution technique when keys must be deleted.

Three techniques are used to calculate the probe sequences required for open addressing: **linear probing, quadratic probing, and double hashing.**

LINEAR PROBING

Given an ordinary hash function h' : U {0, 1, …, m - 1}, the method of linear probing uses the hash function.

$$h(k, i) = (h'(k) + i) \bmod m$$

Where 'm' is the size of the hash table and h' (k) = k mod m (basic hash function). For i = 0, 1, … m - 1. Given key k, the first slot probed is T[h' (k)]. We next probe slot T[h' (k) + 1], and so on up to slot T[m - 1]. Then we wrap around to slots T[0], T[1],…, until we finally probe slot T[h' (k) - 1]. Since the initial probe position determines the entire probe sequence, only m distinct probe sequences are used with linear probing.

Example: Consider inserting the keys 26, 37, 59, 76, 65, 86 into a hash-table of size m = 11 using linear probing, consider the primary hash function is h' (k) = k mod m.

Solution: Initial state of the hash table.

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. **Insert 26.** We known h(k, i) = [h' (k) + i] mod m

   Now h(26, 0) = [26 mod 11 + 0] mod 11

   = (4 + 0) mod 11 = 4 mod 11 = 4

   Since T [4] is free, insert key 26 at this place.

2. **Insert 37.** Now h(370 ,) = [37 mod 11+ 0] mod 11

   = [4 + 0] mod 11 = 4

   Insert T[4] is not free, the next probe sequence is computed as

   h(370 ,) = [37 mod 11+ 1] mod 11

   = [4 + 1] mod 11 = 5 mod11 = 5

T[5] is free, insert key 37 at this place.

3. **Insert 59.** Now h(59, 0) = [59mod11 + 0] mod11
   = [4 + 0] mod 11 = 4

   T[4] is not free, so the next probe sequence is computed as
   h(59, 1) = [59mod11 + 1] mod11
   = 5

   T[5] is also not free, so the next probe sequence is computed.
   h(59, 2) = [59mod11 + 2] mod11
   = 6 mod 11 = 6

   T[6] is free. Insert key 59 at this place.

4. **Insert 76.** h(76, 0) = (65 mod 11 +0) mod 11
   = (10 + 0) mod 11 = 10

   T[10] is free, insert key at this place.

5. **Insert 65.** h(65, 0) = (65 mod 11 + 0) mod11
   = (10 + 0) mod11 = 10

   T[10] is occupied, the next probe sequence is calculated as
   h(65, 1) = (65 mod 11 + 1) mod11
   = (10 + 1) mod11 = 11 mod11 = 0

   T[0] is free, insert key 65 at this place.

6. **Insert 86.** h(86, 0) = (86 mod11 + 0) mod11
   = 9 mod 11 = 9

   T[9] is free, insert key 86 at this place.

   Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	65	/	/	/	26	37	59	/	/	86	76

Disadvantage of linear probing is that records tend to 'Cluster' i.e. appear closest, when the load factor is greater than 50%. Such clustering substantially

increases the average search time for a record. Two techniques to minimize clustering are as follows:

QUADRATIC PROBING

Suppose a record R with key k has the address H(k) = h then instead of searching the locations with addresses h, h + 1, h + 2, … We linearly search the locations with addresses.

$$h,\ h + 1,\ h + 4,\ h + 9,\ ….,\ h + i^2,\ …$$

It uses a hash function of the form

$$\mathbf{h(k,\ i) = (h'\ (k) + c_1 i + c_2 i^2)\ mod\ m}$$

Where (as in linear probing) h' is an auxiliary hash function, c_1 and $c_2 \neq 0$ are auxiliary constants, and i = 0, 1, …, m-1. The initial position probed is T[h' (k)]; later positions probed are offset by amounts that depend in a quadratic manner on the probe number i. This process works better than linear probing but to make full use of the hash table, the values of c_1, c_2 and m are constrained.

Example: Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size m = 11 using quadratic probing with c_1 = 1 and c_2 = 3. Examine that the primary hash function is h' (k) = k mod m.

Solution: For quadratic probing, we have

$$h(k,\ i) = [k\ mod\ m + c_1 i + c_2 i^2]\ mod\ m$$

0	1	2	3	4	5	6	7	8	9	10
/	/	/	/	/	/	/	/	/	/	/

This is the initial state of the hash table.

Here $c_1 = 1$ $c_2 = 3$

$$h(k, i) = [k \bmod m + I + 3i^2] \bmod m$$

1. **Insert 76.**

$$h(76, 0) = (76 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (10 + 0 + 0) \bmod 11 = 10$$

T[10] is free, insert the key 76 at this place.

2. **Insert 26.**

$$h(26, 0) = (26 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (4 + 0 + 0) \bmod 11 = 4$$

T[4] is free, insert the key 26 at this place.

3. **Insert 37.**

$$h(37, 0) = (37 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (4 + 0 + 0) \bmod 11 = 4$$

T[4] is not free, so next probe sequence is computed as

$$h(37, 1) = (37 \bmod 11 + 1 + 3 \times 1^2) \bmod 11$$
$$= (4 + 1 + 3) \bmod 11$$
$$= 8 \bmod 11 = 8$$

T[8] is free, insert the key 37 at this place.

4. **Insert 59.**

$$h(59, 0) = (59 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (4 + 0 + 0) \bmod 11 = 4 \bmod 11 = 4$$

T[4] is not free, so next probe sequence is computed as

$$h(59, 1) = (59 \bmod 11 + 1 + 3 \times 1^2) \bmod 11$$
$$= (4 + 1 + 3) \bmod 11 = 8 \bmod 11 = 8$$

T[8] is also not free, so next probe sequence is computed as

$$h(59, 2) = (59 \bmod 11 + 2 + 3 \times 2^2) \bmod 11$$
$$= (4 + 2 + 12) \bmod 11 = 18 \bmod 11 = 7$$

T[7] is free, insert the key 59 at this place.

5. **Insert 21.**

$$h(21, 0) = (21 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (10 + 0 + 0) \bmod 11 = 10 \bmod 11 = 10$$

T[10] is not free, so next probe sequence is computed as

$$h(21, 1) = (21 \bmod 11 + 1 + 3 \times 1^2) \bmod 11$$
$$= (10 + 1 + 3) \bmod 11 = 14 \bmod 11 = 3$$

T[3] is free, insert the key 21 at this place.

6. **Insert 65.**

$$h(65, 0) = (65 \bmod 11 + 0 + 3 \times 0) \bmod 11$$
$$= (10 + 0 + 0) \bmod 11 = 10 \bmod 11 = 10$$

Since, T[10] is not free, so next probe sequence is computed as

$$h(65, 1) = (65 \bmod 11 + 1 + 3 \times 1^2) \bmod 11$$
$$= (10 + 1 + 3) \bmod 11 = 14 \bmod 11 = 3$$

T[3] is also not free, so next probe sequence is computed as

$$h(65, 2) = (65 \bmod 11 + 2 + 3 \times 2^2) \bmod 11$$
$$= (10 + 2 + 12) \bmod 11 = 24 \bmod 11 = 2$$

T[2] is free, insert the key 65 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	65	21	26	/	/	59	37	/	76

DOUBLE HASHING

It is the process obtained for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

It uses a hash function of the form

$$\mathbf{h(k, i) = (h_1 (k) + ih_2 (k)) \bmod m}$$

Where h_1 and h_2 are auxiliary hash functions and m is the size of the hash table.

Double hashing represents an improvement over linear or quadratic probing in that $\theta(n^2)$ probe sequences are used, rather than $\theta(m)$, since each possible (h_1 (k), h_2 (k)) pair yield a distinct probe sequence, and as we vary the key, the initial probe position h_1 (k) and the offset h_2 (k) may vary independently. As a result, the performance of double hashing appear to be very close to the performance of the "idea" scheme of uniform hashing.

Example: Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size m = 11 using double hashing. Consider that the auxiliary hash functions are h_1 (k) = k mod 11 and h_2 (k) = k mod9.

Solution: Initial state of hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

$h_1(76) = 76 \bmod 11 = 10$

$h_2(76) = 76 \bmod 9 = 4$

$h(76, 0) = (10 + 0 \times 4) \bmod 11$

$= 10 \bmod 11 = 10$

T[10] is free, so insert key 76 at this place.

2. Insert 26.

$h_1(26) = 26 \bmod 11 = 4$

$h_2(26) = 26 \bmod 9 = 8$

$h(26, 0) = (4 + 0 \times 8) \bmod 11$

$= 4 \bmod 11 = 4$

T[4] is free, so insert key 26 at this place.

3. **Insert 37.**

$h_1(37) = 37 \bmod 11 = 4$

$h_2(37) = 37 \bmod 9 = 1$

$h(37, 0) = (4 + 0 \times 1) \bmod 11 = 4 \bmod 11 = 4$

T[4] is not free, the next probe sequence is computed as

$h(37, 1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$

T[5] is free, so insert key 37 at this place.

4. **Insert 59.**

$h_1(59) = 59 \bmod 11 = 4$

$h_2(59) = 59 \bmod 9 = 5$

$h(59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$

Since, T[4] is not free, the next probe sequence is computed as

$h(59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$

T[9] is free, so insert key 59 at this place.

5. **Insert 21.**

$h_1(21) = 21 \bmod 11 = 10$

$h_2(21) = 21 \bmod 9 = 3$

$h(21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 = 10$

T[10] is not free, the next probe sequence is computed as

$h(21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$

T[2] is free, so insert key 21 at this place.

6. **Insert 65.**

$h_1(65) = 65 \bmod 11 = 10$

$h_2(65) = 65 \bmod 9 = 2$

$h(65, 0) = (10 + 0 \times 2) \bmod 11 = 10 \bmod 11 = 10$

T[1] is not free, the next probe sequence is computed as

$$h(65, 1) = (10 + 1 \times 2) \bmod 11 = 12 \bmod 11 = 1$$

T[1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

0	1	2	3	4	5	6	7	8	9	10
/	65	21	/	26	37	/	/	/	59	76

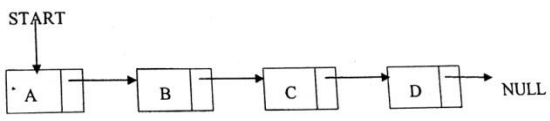
LIST - I

PRESENTATION OF CONTENTS

Linked List

We know that the sequential representation of the ordered list is expensive while inserting or deleting arbitrary elements stored at affixed distance in a fixed memory. The linked representation reduces the expense because the elements are not stored at fixed distance and they are represented randomly and operations such as insertion and deletion requires changes in link only rather than movement of data.

It is a linked representation of the ordered list. It is a limited collection of data elements termed as nodes whose linear order is given by means of link or pointer. Every node consist of two parts. The first part is called INFO, stored information of the data and second part is called LINK, contains the address of the next node in the list. A variable called START, always to the first node of the list and the link part of the last node always contains null value. A null value in the START variable indicate that the list is empty. It is shown in below figure.



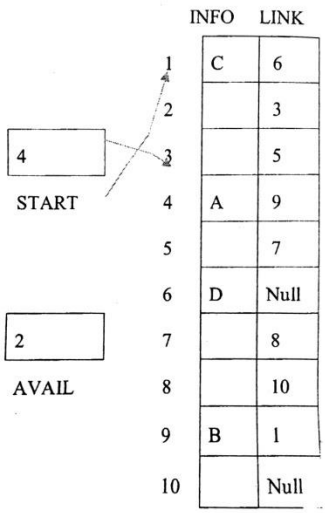
Along with the linked list in the memory, a special list is maintained which consists of list of unused memory cells or unused nodes. This list is called list of available space or availability list or list of free strong or free storage list or free pool. A variable AVAIL is used to store the starting address of the availability list.

Sometimes, during insertion, there may not be accessible space for inserting a data into a data structure, then the situation is called OVERFLOW. Programmers handle the condition by checking whether AVAIL is NULL or not. The situation where one wants to delete data from a data structure that is empty is called UNDERFLOW. The situation is encountered when START is NULL.

Representation of Linked List in Memory

When a linked list is maintained in computer memory, actually two lists, are maintained which are start list and list of available spaces. Let LIST be a linked list. Then LIST will be maintained in memory as follows. First of all, LIST required two linear arrays - we will can them here INFO and LINK which contains the information part and the next pointer field of a node of LIST respectively. START contains the location of the beginning of the list.

The following example of linked lists indicate that more than one list may be maintained in the same linear arrays i.e. INFO and LINK. Below figure pictures a linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters, or, in other words, the string, as follows:



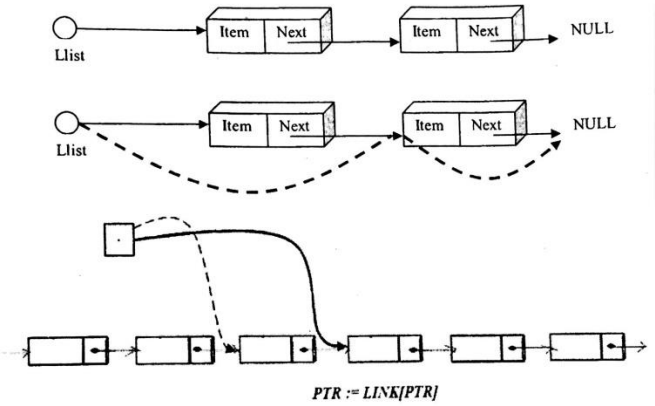
Operations on Linked List

We may perform the following operations on Linked List

a. Traversal: Processing each element in the Linked List
b. Search: Finding the location of the element with a given value
c. Adding a new element to the Linked List
d. Deletion: Removing an element from the Linked List.

Traversing Linked List

In many list - processing operations, we must process each node in the list in sequence; this is called traversing list. To traverse a list in order, we must start at the list head and follow the list pointers. The list can be traversed by using the assignment PTR:= LINK[PTR], which moves the pointer to the next node in the list. As shown in below figure, traversal is done by processing each node starting from first node, the traversal moves ahead via next(link) field and the process is continued until the next field is NULL.



Searching in Linked List

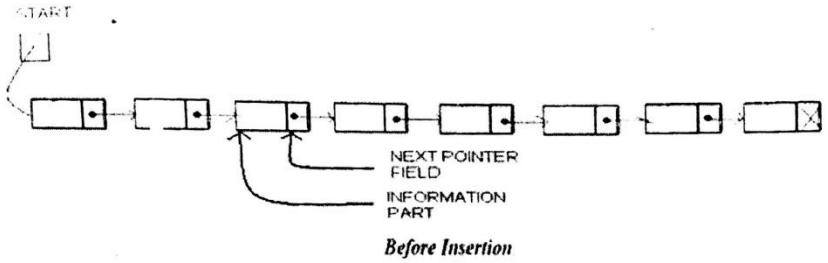
Suppose we have to search an ITEM in the given list. We will be discussing two different searching algorithms for finding the location (LOC) of the node where ITEM first appears in LIST. First algorithm does not assume that the data is in sorted order, while the second algorithm does assume that data is in sorted order. Both the algorithms will only find occurrence of the ITEM in the list.

LIST is Unsorted: If the data in the given list are not necessarily sorted, then we can search for ITEM in the given list by traversing through the list using a pointer variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one by updating the pointer PTR by PTR := LINK[PTR]. For continuing the loop we have to perform two tests. First we have to check whether we reached the end of the list; i.e., PTR = NULL. If not, then we check to see whether INFO[PTR] = ITEM. If the ITEM is found in the list, then list location is returned otherwise NULL is sorted in the location.

LIST is Sorted: If the data in the LIST are sorted, then we can search for ITEM in LIST by traversing the list using a point variable PTR and comparing ITEM with the contents INFO[PTR] of each node, one by one, of LIST. Here we can stop once either when ITEM matches with any node of the list or it exceeds INFO[PTR]

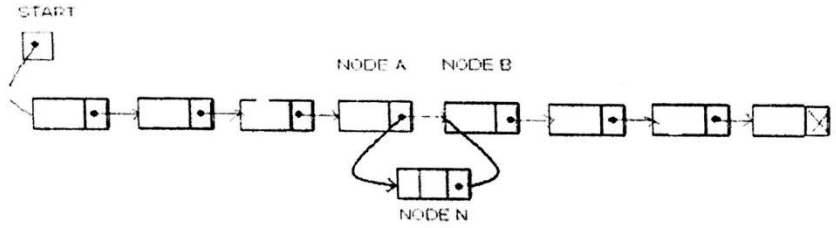
Insertion in Linked List

Let a linked list be represented as below figure. Linked list is having successive nodes A and B, as pictured in below figure. Suppose a node N is to be inserted into the list between nodes A and B. the schematic diagram of such an insertion appears in below figure.



Before Insertion

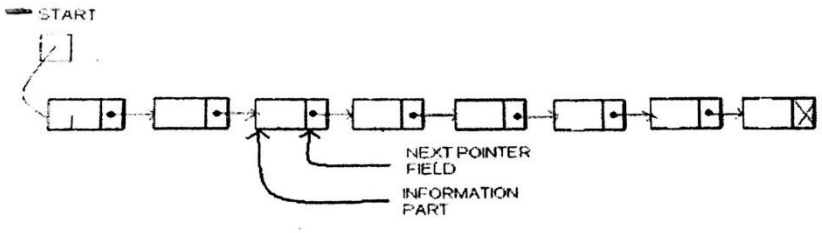
Figure 4.4(Linked-List Before Insertion)



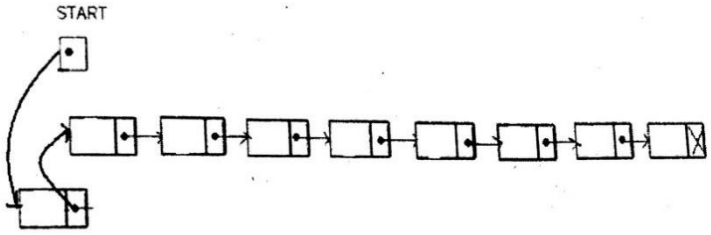
After Insertion

Insertion at the Beginning of a List: The linked list is assumed not to be sorted. The following algorithm inserts the node at the beginning of the list.

Below figures shows the insertions in a linked-list at the beginning of the list.

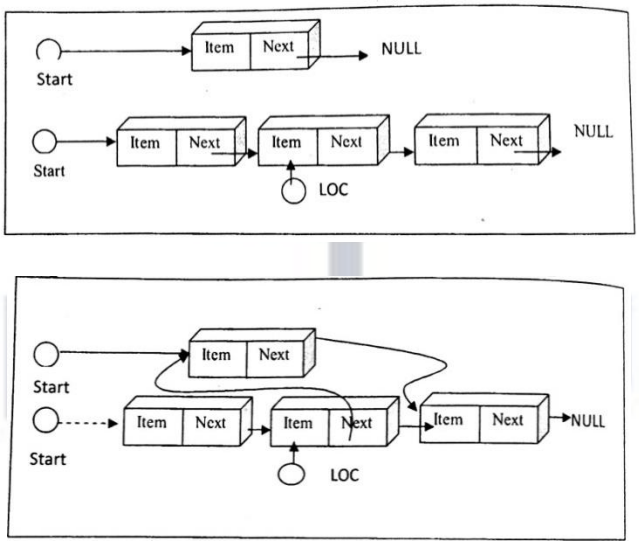


Before Insertion



After Insertion

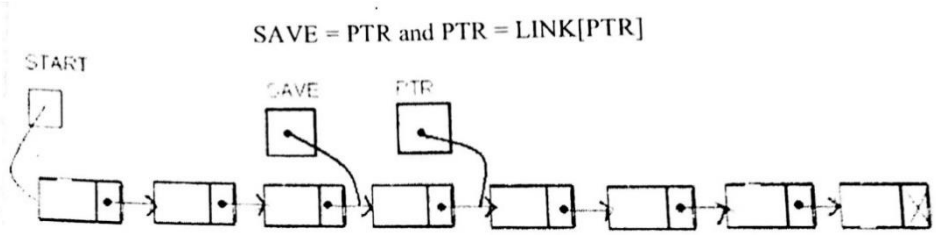
Insertion after a Given Node: Suppose we are given the value of LOC where LOC indicates that LOC is the location of the node after which new node is to be inserted. The following algorithm inserts ITEM into given list so that ITEM follows node for which LOC i.e. location is given, or ITEM is the first node when LOC = NULL.



Insertion into a Sorted Linked List: Suppose ITEM is to be inserted into a sorted linked list. This algorithm inserts the node into a sorted linked list. The ITEM must be inserted between nodes A and B so that INFO(A) < ITEM < INFO(B)

In this case first location LOC of the node after which new node is to be inserted is to be found. For this we will write a procedure that finds the location LOC of node A.

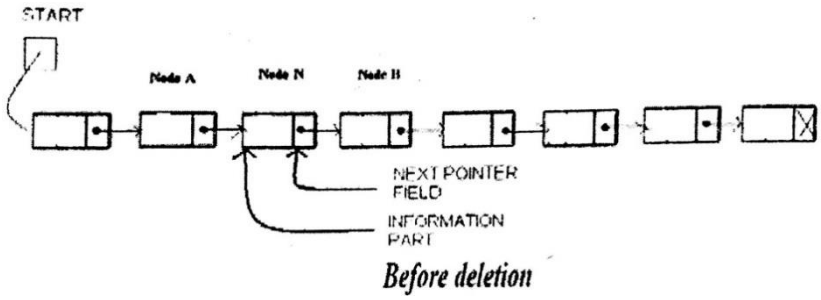
Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, we have to keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in below figure. Thus SAVE and PTR are updated by the assignments.



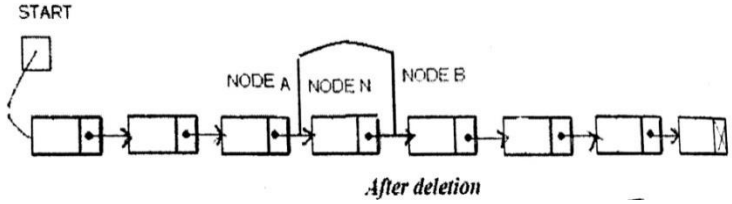
The traversing stops as soon as ITEM < INFO[PTR]. Then PTR points to node B. So SAVE will contain the location of the node A.

Deletion from Linked List

Let LIST be a linked list with a node N between nodes A and B, as pictured in below figure. Suppose node N is to be deleted from the linked list. The schematic diagram. Of such a deletion appears in below figure. The deletion occurs as soon as the next pointer fields of node A points to node B. therefore, when performing deletions, one must keep track of the address of the node which immediately precedes the node that is to be deleted.



Before deletion



After deletion

Deleting the Node Following a Given Node

Let LIST be a linked list in memory. Suppose we are given the location LOC of a node N in LIST. Furthermore, suppose we are given the location LOCP of the node preceding N or, when N is the first node, we are given LOCP = NUL. The following algorithm deletes N from the list.

Deleting the Node with a Given ITEM of Information

Let LIST be a linked list in memory. Assume that we are given an ITEM of information and we want to delete from the LIST the first node N which contains ITEM. First we give a producer which finds the location LOC of the node N containing ITEM and the location LOCP of the node preceding node N. If N is the first node, we set LOCP = NULL, and if ITEM does not appear in LIST, we set LOC = NULL. (This procedure is similar to Procedure in insertion in a sorted list)

Cross the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While crossing, keep track of the location of the earlier node by using a pointer variable SAVE. Thus SAVE and PTR are modified by the assignments.

SAVE := PTR and PTR := LINK[PTR]

The traversing stops ITEM = INFO[PTR]. Then PTR carry location LOC of node N and SAVE carry the location LOCP of the node preceding N.

The formal statement of our procedure follows. The cases where the list is empty or where INFO[START] = ITEM (i.e., where node N is the first node) are treated separately, since they do not involve the variable SAVE.

LINKED LIST - II

PRESENTATION OF CONTENTS

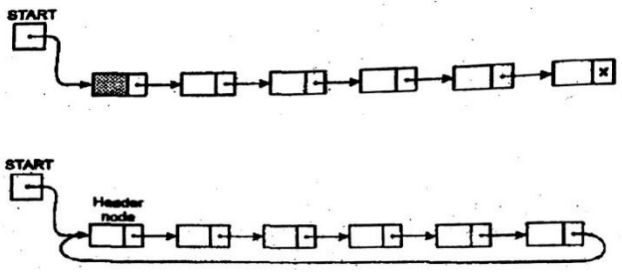
Header Linked List

It is a linked list always carry a special node, called the header node, at the beginning of the list. Two kinds of header list are:

1. A grounded header list is a header list where the last node carry the null pointer.
2. A circular header list is a header list where the last node end back to the header node.

Below figure carry schematic diagrams of these header lists. Unless stated or implied, our header lists will always be circular. In such cases, the header node also move as a sentinel indicating the end of the list.

It can maintain global properties of entire list and act as utility node. For example, in header node you can maintain count variable which gives number of nodes in list. You can modify header node count member whenever you add/delete any node. It will help in list count without negotiating in O(1) time.



Examine that the list pointer START always points to the header node. Hence, LINK[START] = NULL determines that a grounded header list is empty, and LINK[START] = START indicates that a circular header list is empty.

Circular Linked List

As discussed circular linked list is a list in which last node of the list points to the

First node instead of NULL. As in case of simple linked list, we can perform various operations on circular linked list as well.

Two Way(Doubly) Linked List

Let us now discuss a two-way list, which can be traversed in two direction i.e., either

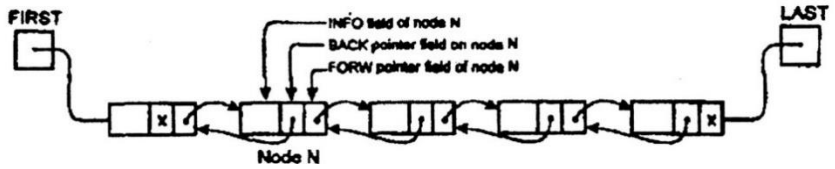
a. In the usual forward direction from the beginning of the list to the end.
b. In the backward direction from the end of the list to the beginning.

Moreover, given location LOC of a node N in the list one, now has immediate access to both the next node and the preceding node in the list. This means, in particular, we may be able to delete N directly from the list without traversing any part of the list.

It is a limited collection of data elements, called nodes, where each node N is divided into three parts.

1. An information filed INFO which carry data of N.
2. A pointer field FORW which carry location of the next node in the list
3. A pointer field BACK which carry location of the preceding node in the list.

The list also required two list pointer variables: FIRST, which points to the first node in the list, and LAST, which point to the last node in the list. Below figure contains a schematic diagram of such a list. Observe that the null pointer appears in the FORW field of the last node in the list and also in the BACK field of the first node in the list.



Examine that, using the variable FIRST and the pointer field FORW, we can traverse a two-way list in the forward direction. On the other hand, using the variable LAST and the pointer field BACK, we can also traverse the list in the backward direction.

Assume LOCA and LOCB are the locations, of nodes A and B in a two-way list respectively. Then the way that the pointers FORW and BACK are defined gives us the pointer property: FORW[LOCA] if and only if BACK[LOCB] = LOCA

The statement that node B follows node A is equivalent to the statement that node A precedes node B.

gradeup SUPER

Gradeup UGC NET
Super Superscription

Features:

1. 7+ Structured Courses for UGC NET Exam
2. 200+ Mock Tests for UGC NET & MHSET Exams
3. Separate Batches in Hindi & English
4. Mock Tests are available in Hindi & English
5. Available on Mobile & Desktop

Gradeup Super Subscription, Enroll Now

gradeup.co