Linked Lists

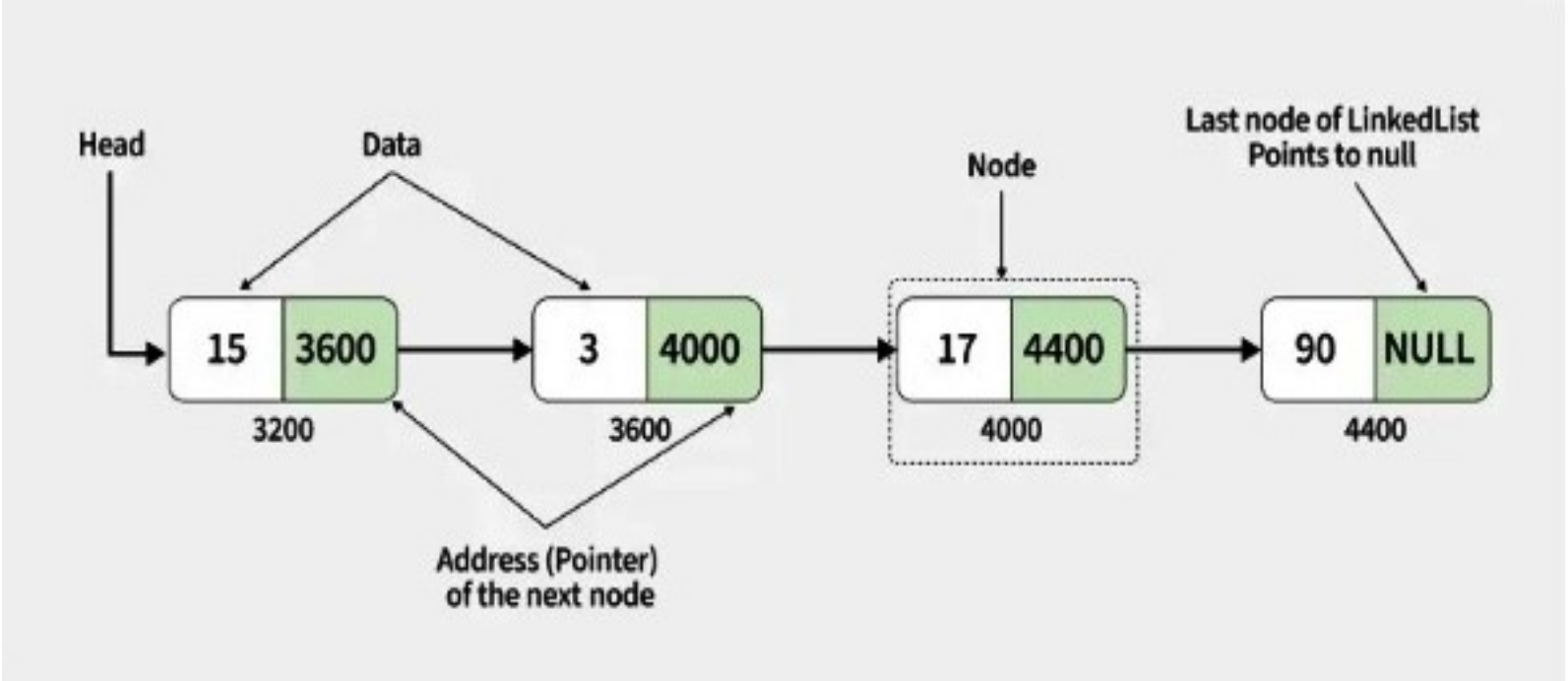
Dr. Anish Soni

Introduction-List

- A linked list is a linear data structure which can store a collection of "nodes" connected together via links i.e. pointers.
- Linked lists nodes are not stored at a contiguous location, rather they are linked using pointers to the different memory locations.
- Each node contains two parts: the actual **data** and a **pointer** (or reference) that links to the next node in the sequence.
- A linked list is a dynamic linear data structure whose memory size can be allocated or de-allocated at run time based on the operation insertion or deletion, this helps in using system memory efficiently.
- Linked lists can be used to implement various data structures like a stack, queue, graph, hash maps, etc.

Core Components of List:

- **Node**: The basic building block, typically a self-referential structure.
- **Head**: A pointer to the first node, serving as the entry point for the list.
- **Tail**: The last node, which usually points to NULL to signify the end of the list.



Why Linked List?

Arrays can be used to store linear data of similar types, but arrays have the following limitations.

1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage.

2) Inserting a new element in an array of elements is expensive because the room has to be created for the new elements and to create room existing elements have to be shifted.

For example

- in a system, if we maintain a sorted list of IDs in an array id[].

  id[] = [1000, 1010, 1050, 2000, 2040].

- And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).
  Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

Array vs. Linked List

Feature	Array	Linked List
Memory Allocation	Contiguous (elements are next to each other).	Non-contiguous (nodes are scattered, linked by pointers).
Size	Fixed size (declared beforehand).	Dynamic size (can grow/shrink at runtime).
Access Method	Direct access using index (random access).	Sequential access (must traverse from the beginning/head).
Access Time	**Fast (O(1))**.	Slow (O(n)).
Insertion/Deletion	Slow (O(n)) due to element shifting.	**Fast (O(1))** if the position/pointer is known.
Memory Efficiency	More memory-efficient (no pointer overhead).	Less memory-efficient (extra space for pointers in each node).
Cache Performance	Better (due to data locality in contiguous memory).	Poor (due to scattered memory locations).
Implementation	Simpler to implement.	More complex (requires careful pointer management).

Representation of Linked list in memory

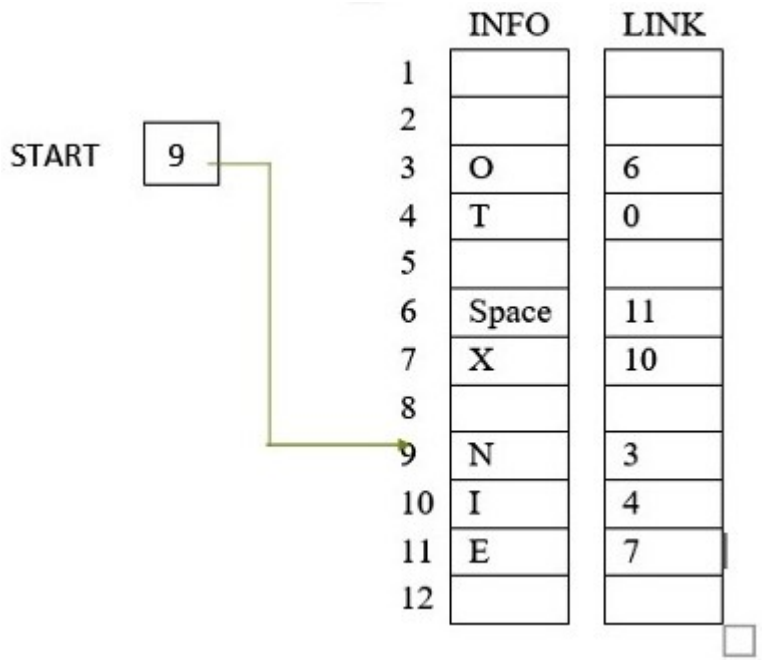
Let LIST be a linked list. Then LIST will be maintained in memory specified as follows.

- First of all, LIST requires two linear arrays
- we will call them here INFO and LINK-
- INFO[K] contain the information part.
- LINK[K] contain the nextpointer field of a node LIST.
- LIST also requires a variable name- such as START.
- START contains the location of the beginning of the list, and a nextpointer sentinel -denoted by NULL- which indicate the end of the list.
- Since the subscripts of the array INFO and LINK are usually positive, we will choose NULL=0.

Representation of Linked list in memory

The following example of linked list indicate that

- the nodes of a list need not occupy adjacent elements in the array INFO and LINK,
- more than one list may be maintained in the same linear array INFO and LINK. However, each list must have its own pointer variable giving the location of its first node.



Representation

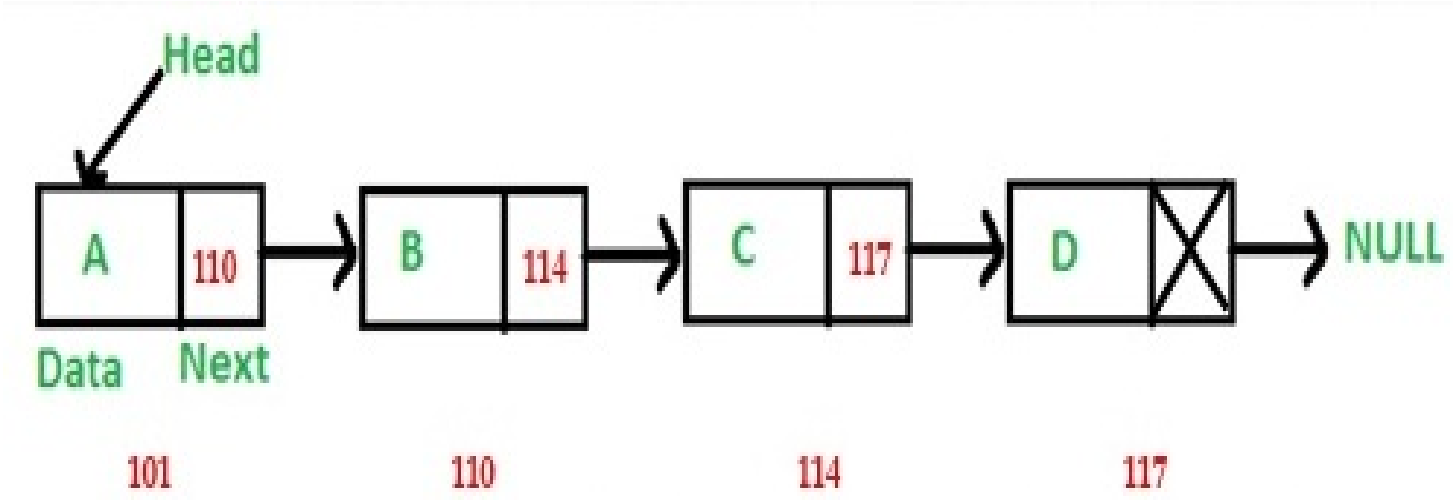
- Above picture is of linked list in memory where each node of the list contains a single character. We can obtain the actual list of characters as follows.
- **Algorithm**
- START= 9, so INFO[9]=N is the first character.
- LINK[9]=3, so INFO[3]=O is the second character.
- LINK[3]=6, so INFO[6]= (Blank) is the third character.
- LINK[6]=11, so INFO[11]=E is the fourth character.
- LINK[11]=7, so INFO[7]=X is the fifth character.
- LINK[7]=10, so INFO[10]=I is the sixth character.
- LINK[10]=4, so INFO[4]=T is the seventh character.
- LINK[4]=0, the NULL value, so the List has ended.
- **In other words, NO EXIT is the character string.**

Types of Linked List

- Following are the various types of linked list.
- **Simple Linked List** – Item navigation is forward only.
- **Doubly Linked List** – Items can be navigated forward and backward.
- **Circular Linked List** – Last item contains link of the first element as next and the first element has a link to the last element as previous.

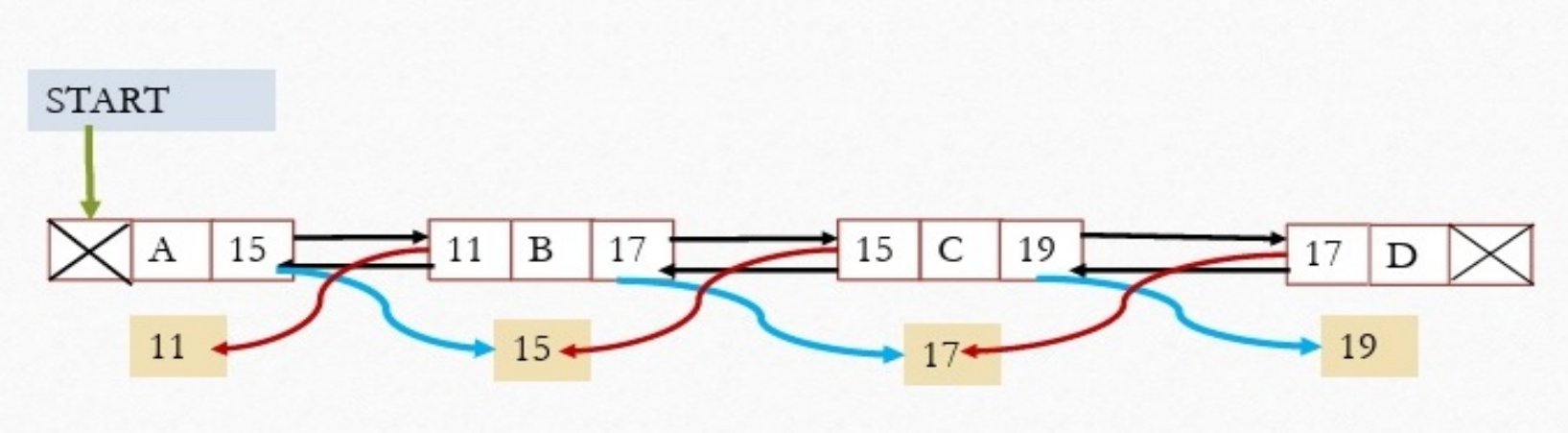
Singly Linked Lists

- Singly linked lists contain two "buckets" in one node; one bucket holds the data and the other bucket holds the address of the next node of the list. Traversals can be done in one direction only as there is only a single link between two nodes of the same list.



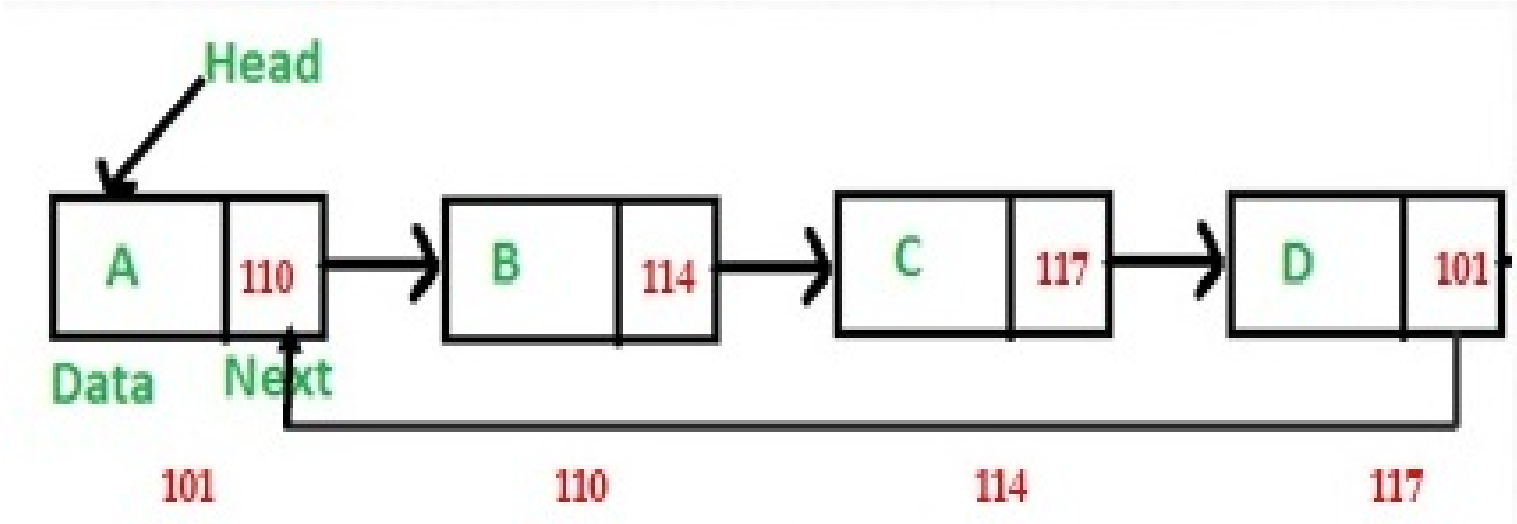
Doubly Linked Lists

- Doubly Linked Lists contain three "buckets" in one node; one bucket holds the data and the other buckets hold the addresses of the previous and next nodes in the list. The list is traversed twice as the nodes in the list are connected to each other from both sides.



Circular Linked Lists

- Circular linked lists can exist in both singly linked list and doubly linked list.
- Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.



Operations on Linked Lists

- The primary operations on a linked list include **insertion**, **deletion**, **traversal**, **searching**, and **updating**. These operations manipulate the nodes, each of which contains data and a pointer to the next node in the sequence.

Core Operations

- **Traversal**: This operation involves visiting each node in the list sequentially, starting from the head, and following the next pointers until the end of the list (indicated by a NULL pointer). Traversal is necessary for printing all elements, searching for a specific value, or counting nodes.
- **Insertion**: This adds a new node to the linked list. It can be performed at several positions:
  - **At the beginning** (head): This is efficient (O(1) time) as it only requires updating the new node's pointer to the current head and then making the new node the head.
  - **At the end** (tail): This requires traversing the entire list to find the last node and then updating its pointer to the new node (O(n) time).
  - **At a specific position** (after a given node): The list must be traversed to the node just before the desired position to adjust the pointers correctly (O(n) time).

Core Operations

- **Deletion**: This removes a node from the linked list and re-links the surrounding nodes.
  - **From the beginning**: Like insertion at the beginning, this is efficient (O(1) time) by simply moving the head pointer to the next node.
  - **From the end**: This requires traversing to the second-to-last node to update its pointer to NULL (O(n) time).
  - **From a specific position**: The list must be traversed to the node preceding the one to be deleted to update the pointers and skip the target node (O(n) time).
- **Searching**: This operation involves traversing the list from the head to find a node that contains a specific data value. The worst-case time complexity is O(n) because every node might need to be checked.
- **Updating**: This modifies the data part of an existing node. It requires first finding the node (O(n) time in the worst case) and then changing its data value

Advanced Operations

- **Reversing**: This operation changes the direction of all pointers in the list so that the last node becomes the new head and the list can be traversed in reverse (without modifying the node structure, which would make it a doubly linked list).
- **Sorting**: Various sorting algorithms can be applied to a linked list, with merge sort being one of the most performant for this data structure.

Traversing

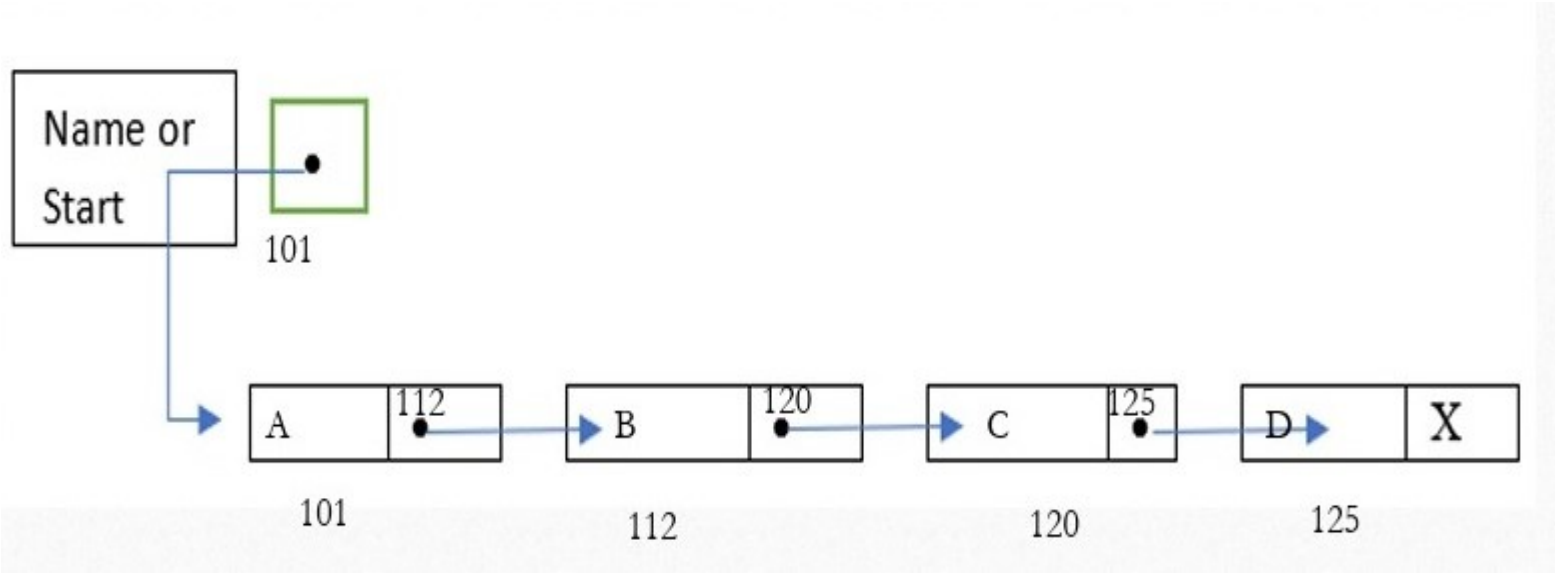
A linked list is a linear data structure that needs to be traversed starting from the head node until the end of the list. Unlike arrays, where random access is possible, linked list requires access to its nodes through sequential traversal. Traversing a linked list is important in many applications. For example, we may want to print a list or search for a specific node in the list. Or we may want to perform an advanced operation on the list as we traverse the list. The algorithm for traversing a list is fairly trivial.

a. Start with the head of the list. Access the content of the head node if it is not null.

b. Then go to the next node(if exists) and access the node information

c. Continue until no more nodes (that is, you have reached the last node)

Let LIST be a linked list in memory stored in linear array INFO and LINK with START pointing to the first element and NULL indicating the end of LIST. Suppose we want to traverse LIST in order to Process each node exactly once. This section presents an algorithm that does so and then uses the algorithm in some applications.


Name or
Start
101
A
112
B
120
C
125
D
X
101
112
120
125

Algorithm: 1. Set PTR:=START.[Initializes pointer PTR]

2. Repeat Step 3 and 4 while PTR≠ NULL.

3. Apply PROCESS to INFO[PTR].

4. Set PTR:= LINK[PTR]. [PTR now points to the next node.]

[End of Step 2 loop.]

5. Exit.

Algorithm details

Initialize PTR or START.

Then process INFO[PTR], the information at the first node.

Update PTR by the assignment PTR:=LINK[PTR], so that PTR points to the second node.

Then Process INFO[PTR], the information at the second node.

Again update PTR by the assignment operator PTR:=LINK[PTR], and then process INFO[PTR], the information at the third node.

And so on. Continue until PTR=NULL, which signals the end of the list.

The following procedure finds the number NUM of elements in a linked list.

Procedure: COUNT(INFO, LINK, START, NUM)

1. Set NUM : = 0. [Initializes counter.]
2. Set PTR : = START. [Initializes pointer.]
3. Repeat Steps 4 and 5 while PTR ≠ NULL.
4. Set NUM : = NUM + 1. [Increases NUM by 1.]
5. Set PTR : = LINK[PTR]. [Updates pointer.]

   [End of Step 3 loop.]
6. Return.

Searching Unsorted linked list

- Let LIST be a linked list in memory which is not sorted, then one searches for ITEM in 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, of LIST. Before we update the pointer PTR by PTR:=LINK[PTR]
- We require two tests. First we have to check to see whether we have reached the end of list i.e.
- PTR= NULL
- Then we check to see whether
- INFO[PTR]=ITEM

Algorithm:

SEARCH (INFO, LINK, START, ITEM, LOC)

LIST is a linked list in memory. This algorithm finds the location LOC of the node where ITEM first appear in LIST, or set LOC=NULL.

1 set PTR:=START.

2 Repeat step 3 while PTR ≠ NULL:

3 If ITEM =INFO[PTR], then:

Set LOC:=PTR, and Exit.

Else:

Set PTR:=LINK[PTR].[PTR now points to the next node.]

[End of IF structure]

[End of step 2 loop]

4 [Search is unsuccessful.] set LOC:=NULL.

5 Exit.

Searching Sorted List

Algorithm 5.3: SRCHSL(INFO, LINK, START, ITEM, LOC)
LIST is a sorted list in memory. This algorithm finds the location LOC of the node where ITEM first appears in LIST, or sets LOC = NULL.

1. Set PTR := START.
2. Repeat Step 3 while PTR ≠ NULL:
3. If ITEM < INFO[PTR], then:
   Set PTR := LINK[PTR]. [PTR now points to next node.]
   Else if ITEM = INFO[PTR], then:
   Set LOC := PTR, and Exit. [Search is successful.]
   Else:
   Set LOC := NULL, and Exit. [ITEM now exceeds INFO[PTR].]
   [End of If structure.]
   [End of Step 2 loop.]
4. Set LOC := NULL.
5. Exit.

Overflow

- Sometimes new data are to be inserted into a data structure but there is no available space, i.e. the free-storage list is empty. This situation is usually called overflow.

- The programmer may handle overflow by printing the message OVERFLOW.
- In such a case, the programmer may then modify the program by adding space to the underlying arrays.
- Observe that overflow will occur with our linked lists when AVAIL=NULL and there is an insertion.

UNDERFLOW

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

Insertion Algorithms

Algorithms which insert nodes into linked lists come up in various situations. We discuss three of them here. The first one inserts a node at the beginning of the list, the second one inserts a node after the node with a given location, and the third one inserts a node into a sorted list. All our algorithms assume that the linked list is in memory in the form LIST(INFO, LINK, START, AVAIL) and that the variable ITEM contains the new information to be added to the list.

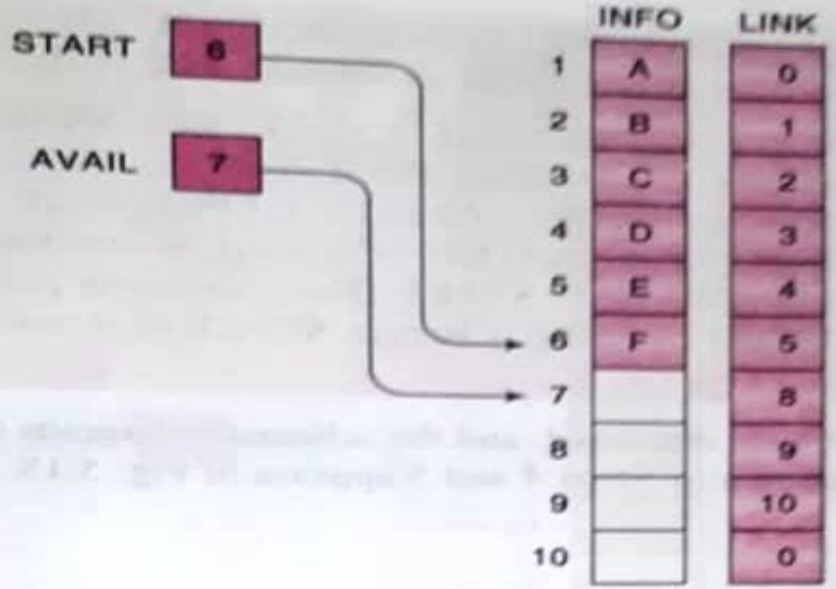


Fig. 5.16

Since our insertion algorithms will use a node in the AVAIL list, all of the algorithms will include the following steps:

(a) Checking to see if space is available in the AVAIL list. If not, that is, if AVAIL = NULL, then the algorithm will print the message OVERFLOW.

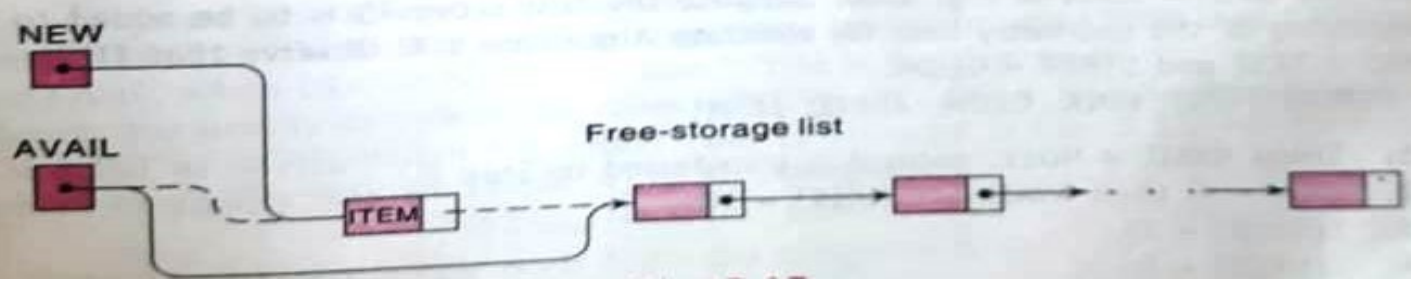
(b) Removing the first node from the AVAIL list. Using the variable NEW to keep track of the location of the new node, this step can be implemented by the pair of assignments (in this order)

NEW := AVAIL, AVAIL := LINK[AVAIL]

(c) Copying new information into the new node. In other words,

INFO[NEW] := ITEM

The schematic diagram of the latter two steps is pictured in Fig. 5.17.



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

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

Steps 1 to 3 have already been discussed, and the schematic diagram of Steps 2 and 3 appears in Fig. 5.17. The schematic diagram of Steps 4 and 5 appears in Fig. 5.18.

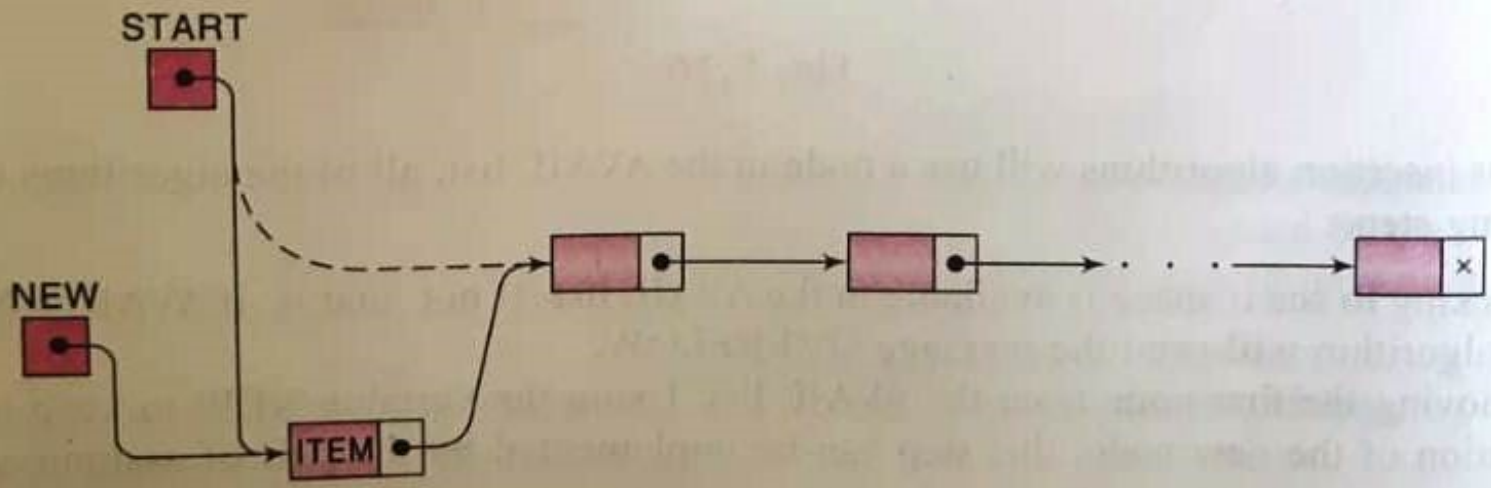


Fig. 5.18 *Insertion at the Beginning of a List*

Inserting after a Given Node

Suppose we are given the value of LOC where either LOC is the location of a node A in a linked LIST or LOC = NULL. The following is an algorithm which inserts ITEM into LIST so that ITEM follows node A or, when LOC = NULL, so that ITEM is the first node.

Let N denote the new node (whose location is NEW). If LOC = NULL, then N is inserted as the first node in LIST as in Algorithm 5.4. Otherwise, as pictured in Fig. 5.15, we let node N point to node B (which originally followed node A) by the assignment

LINK[NEW] := LINK[LOC]

and we let node A point to the new node N by the assignment

LINK[LOC] := NEW

A formal statement of the algorithm follows.

Algorithm 5.5: INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM)
This algorithm inserts ITEM so that ITEM follows the node with location LOC or inserts ITEM as the first node when LOC = NULL.

1. [OVERFLOW?] If AVAIL = NULL, then: Write: OVERFLOW, and Exit.
2. [Remove first node from AVAIL list.]
   Set NEW := AVAIL and AVAIL := LINK[AVAIL].
3. Set INFO[NEW] := ITEM. [Copies new data into new node.]
4. If LOC = NULL, then: [Insert as first node.]
   Set LINK[NEW] := START and START := NEW.
   Else: [Insert after node with location LOC.]
   Set LINK [NEW] := LINK[LOC] and LINK[LOC] := NEW.
   [End of If structure.]
5. Exit.

Inserting into a Sorted Linked List

Suppose ITEM is to be inserted into a sorted linked LIST. Then ITEM must be inserted between nodes A and B so that

$$\text{INFO(A)} < \text{ITEM} \leq \text{INFO(B)}$$

The following is a procedure which finds the location LOC of node A, that is, which finds the location LOC of the last node in LIST whose value is less than ITEM.

Traverse the list, using a pointer variable PTR and comparing ITEM with INFO[PTR] at each node. While traversing, keep track of the location of the preceding node by using a pointer variable SAVE, as pictured in Fig. 5.20. Thus SAVE and PTR are updated by the assignments

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

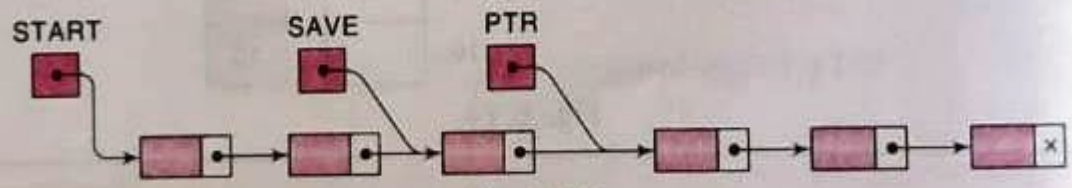


Fig. 5.20

The traversing continues as long as INFO[PTR] > ITEM, or in other words, 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.

The formal statement of our procedure follows. The cases where the list is empty or where ITEM < INFO[START], so LOC = NULL, are treated separately, since they do not involve the variable SAVE.

Procedure 5.6: FINDA(INFO, LINK, START, ITEM, LOC)
This procedure finds the location LOC of the last node in a sorted list such that INFO[LOC] < ITEM, or sets LOC = NULL.

1. [List empty?] If START = NULL, then: Set LOC := NULL, and Return.
2. [Special case?] If ITEM < INFO[START], then: Set LOC := NULL, and Return.
3. Set SAVE := START and PTR := LINK[START]. [Initializes pointers.]

4. Repeat Steps 5 and 6 while PTR ≠ NULL.
5. If ITEM < INFO[PTR], then:
   Set LOC := SAVE, and Return.
   [End of If structure.]
6. Set SAVE := PTR and PTR := LINK[PTR]. [Updates pointers.]
   [End of Step 4 loop.]
7. Set LOC := SAVE.
8. Return.

Now we have all the components to present an algorithm which inserts ITEM into a linked list. The simplicity of the algorithm comes from using the previous two procedures.

Algorithm 5.7: INSERT(INFO, LINK, START, AVAIL, ITEM)
This algorithm inserts ITEM into a sorted linked list.

1. [Use Procedure 5.6 to find the location of the node preceding ITEM.]
   Call FINDA(INFO, LINK, START, ITEM, LOC).
2. [Use Algorithm 5.5 to insert ITEM after the node with location LOC.]
   Call INSLOC(INFO, LINK, START, AVAIL, LOC, ITEM).
3. Exit.