SPCAM

BCA SEM 3

US03MABCA01 – FUNDAMENTALS OF DATA STRUCTURES

UNIT – IV - LINKED LISTS, SORTING AND SEARCHING TECHNIQUES

Unit-IV : Topics
• **Introduction to linked list**
• **Types of linked lists: Singly Linked list** **Doubly Linked list** **Circular Linked list** **Circular Linked list**
• **Operations on Singly Linked lists: Insertion : at Front,** **Deletion : from beginning**
• **Introduction to Sorting and Searching**
• **Sorting techniques – Bubble Sort and Merge Short**
• **Searching techniques – Sequential Search and Binary Search**

INTRODUCTION

An array is a data structure where elements are stored in consecutive memory locations. In order to occupy the adjacent space, a block of memory that is required for the array should be allocated beforehand. Once memory is allocated it cannot be extended any more. This is why array is known as a static data structure.

Linked list is called dynamic data structure where the amount of memory required can be varied during its use. In linked list, the adjacency between the elements is maintained by means of links or pointers.

A link or pointer actually is the address (memory location) of the subsequent element. Thus, in a linked list, data (actual content) and link (to point to the next data) both are required to be maintained.

An element in a linked list is specially termed as node, which can be viewed as shown in Figure 3.1. A node consists of two fields:

- DATA (to store the actual information) and
- LINK (to point to the next node).

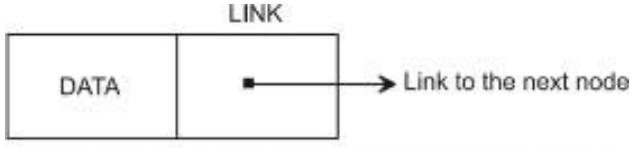


Figure 3.1 Node: an element in a linked list.

DEFINITION

A linked list is an ordered collection of finite, homogeneous data elements called nodes where the linear order is maintained by means of links or pointers.

Depending on the requirements the pointers are maintained, and accordingly linked list can be classified into three major groups:

- Singly Linked List
- Doubly Linked List
- Circular Linked List
- Circular Doubly Linked List

SINGLY LINKED LIST

In a single linked list, each node contains only one link which points the subsequent node in the list. Figure 3.2 shows a linked list with six nodes. Here, N1, N2.....N6 are the constituent nodes in the list. HEADER is an empty node (having data element NULL) and only used to store a pointer to the first node N1.

Thus, if one knows the address of the HEADER node from the link field of this node, the next node can be traced, and so on.
This means that starting from the first node one can reach to the last node whose link field does not contain any address but has a null value.

Note that in a single linked list one can move from left to right only; this is why a single linked list is also called **One way list.**

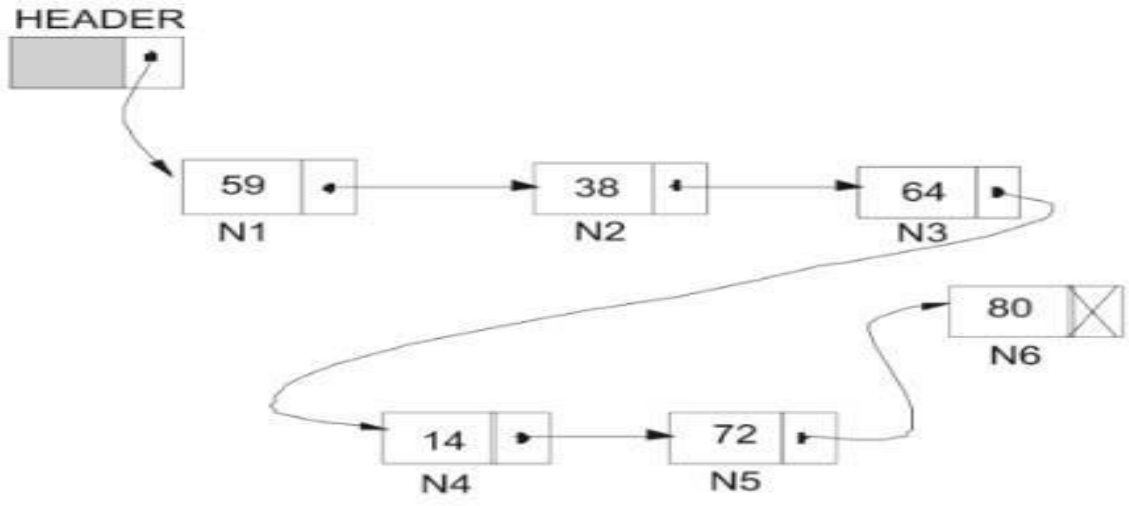


Figure 3.2 A single linked list with six nodes.

CIRCULAR LINKED LIST

In our previous discussion, we have noticed that in single linked list, the link field of the last node is null (hereafter a single linked list may be read as ordinary linked list), but a number of advantages can be gained if we utilize this link field to store the pointer of the header node.

A linked list where the last node points the header node is called circular linked list. Figure 3.3 shows a pictorial representation of a circular linked list.

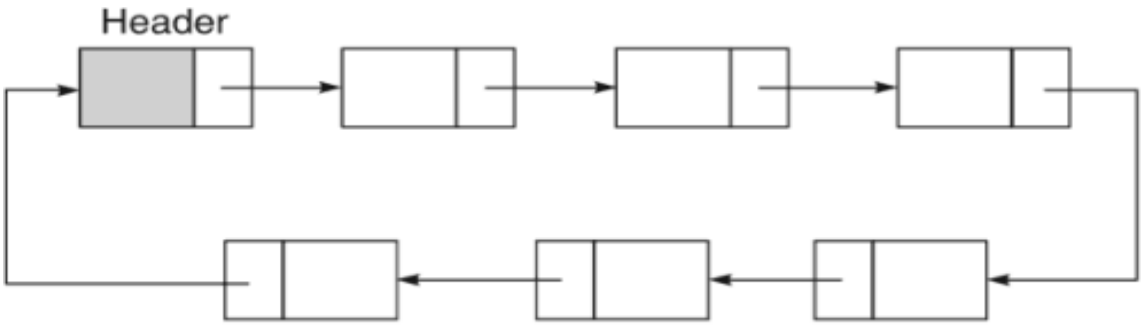


Figure 3.8 A circular linked list.

Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again. Useful for implementation of queue. Unlike the implementation, we don't need to maintain two pointers for front and rear if we use circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next of last. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list so that when it reaches the end of the list it can cycle around to the front of the list.

DOUBLY LINKED LISTS

In a single linked list one can move beginning from the header node to any node in one direction only (from left to right). This is why a single linked list is also termed a one-way list.

On the other hand, a double linked list is a **two-way list** because one can move in either direction, either from left to right or from right to left.

This is accomplished by maintaining two link fields instead of one as in a single linked

list. A structure of a node for a double linked list is represented as in Figure 3.10.

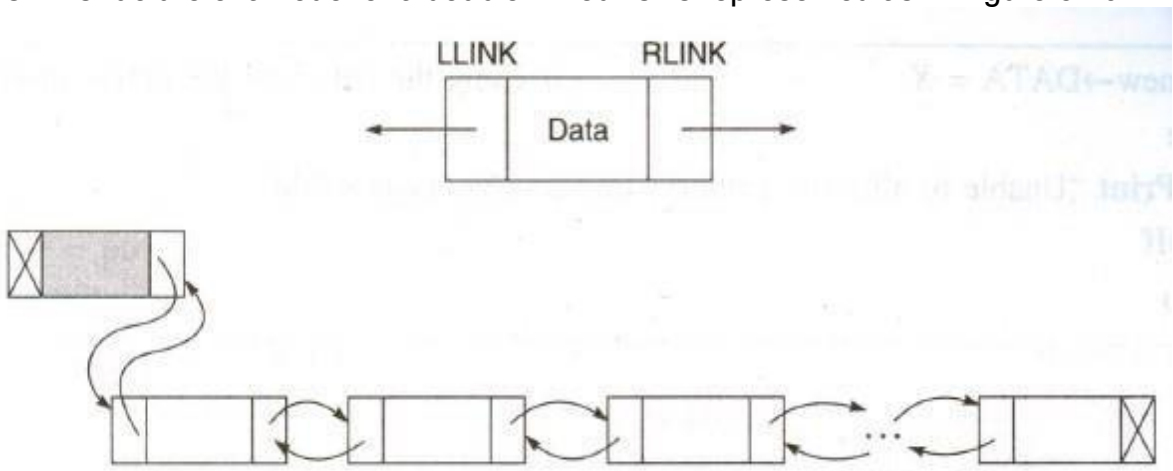


Figure 3.10 Structure of a node and a double linked list.

The figure shows two fields, namely RLINK and LLINK, point to the nodes on the right side and left side of the node, respectively. Thus, every node, except the header node and the last node, points to its immediate predecessor and immediate successor.

CIRCULAR DOUBLY LINKED LISTS

The advantage of both double linked list and circular linked list are incorporated into another type of list structure called circular double linked list and it is known to be the best of its kind. Figure 3.13 shows a schematic presentation of a circular double linked list.

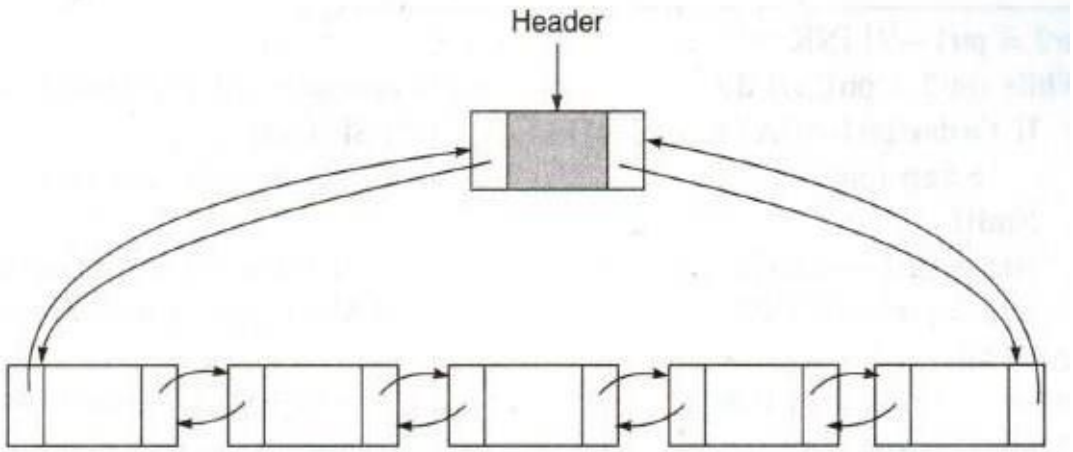


Figure 3.13 A circular double linked list.

Here note that the

- RLINK (right link) of the rightmost node and LLINK (left link) of the leftmost node contain the address of the header node;
- again the RLINK and LLINK of the header node contain the address of the rightmost node and the leftmost node, respectively.

An empty circular double linked list is represented as shown in figure 3.14. In case of an empty list, both LLINK and RLINK of the header node point to itself.

Figure 3.14 An empty circular double linked list.

INSERTING A NODE INTO A SINGLY LINKED LIST

There are various positions where a node can be inserted:

1. Insert at front (as a first element)
2. Insert at end (as a last element)
3. Insert at any other position

Inserting of a node at the front of a singly linked list

The algorithm InsertFront_SL is used to insert a node at the front of a single linked list.

Algorithm InsertFront_SL

Input:	HEADER is the pointer to the header node and X is the data of the node to be inserted.
Output:	A single linked list with newly inserted node at the front of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

```
1.  new = Get Node (NODE)                    // Get a memory block of type NODE and
                                                store its pointer in new
2.  If (new = NULL) then                     // Memory manager returns NULL on searching
                                                the memory bank
3.      Print "Memory underflow: No insertion"
4.      Exit
5.  Else                                     // Memory is available and get a node from memory bank
6.      new → LINK = HEADER → LINK           // Change of pointer 1 as shown
                                                in Fig.3.5(a)
7.      new → DATA = X                       // Copy the data X to newly availed node
8.      HEADER → LINK = new                  // Change of pointer 2 as shown in Fig.3.5(a)
9.  EndIf
10. Stop
```

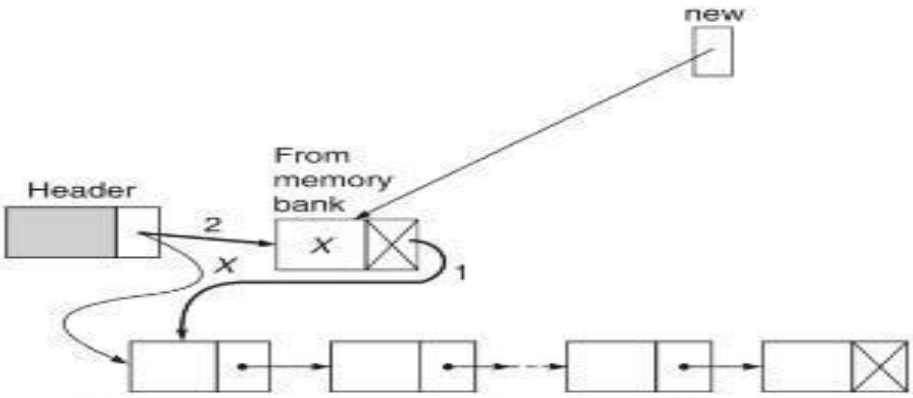


Figure 3.5(a) Inserting a node in the front of a single linked list.

DELETION OF A NODE FROM A SINGLY LINKED LIST

Like insertions, there are also three cases of deletion:

1. Deleting from the begining of the list
2. Deleting from the end of the list
3. Deleting from any position in the list.

Deleting the node from Begining of a singly linked list

The algorithm DeleteFront_SL is used to delete the node at the front of a single linked list. Such a delete operation is explained in figure 3.6(a).

Algorithm DeleteFront_SL

Input:	HEADER is the pointer to the header node.
Output:	A single linked list after eliminating the node at the front of the list.
Data structures:	A single linked list whose address of the starting node is known from the HEADER.

Steps:

1. ptr = HEADER → LINK // Pointer to the first node
2. **If** (ptr = NULL) **then** // If the list is empty
3. **Print** "The list is empty: No deletion"
4. Exit // Quit the program
5. **Else** // The list is not empty
6. ptr1 = ptr → LINK // ptr1 is the pointer to the second node, if any
7. HEADER → LINK = ptr1 // Next node becomes the first node as in Fig. 3.6(a)
8. **Return Node** (ptr) // Deleted node is freed to the memory bank for future use
9. **EndIf**
10. **Stop**

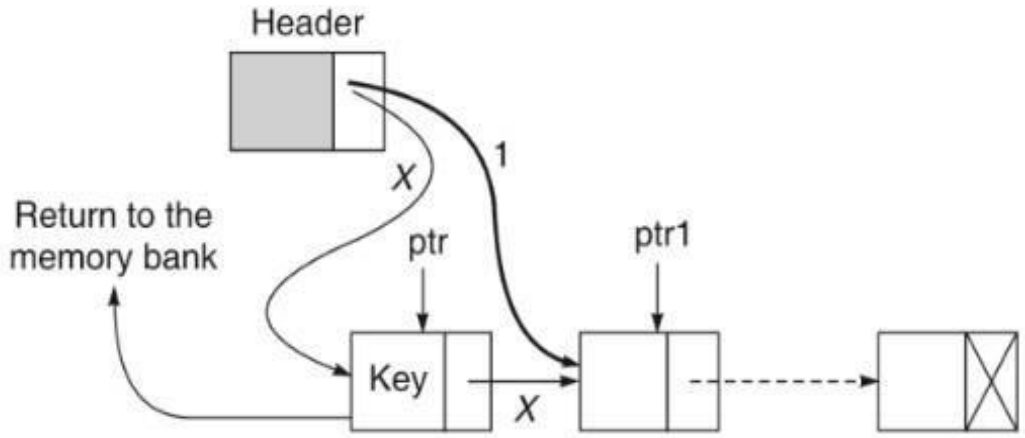


Figure 3.6(a) Deleting the node at the front of a single linked list.

SORTING AND SEARCHING TECHNIQUES

Sorting: It is the process to arrange the data in ascending or descending order.

Categories of Sorting:

Sorting can be classified into two broad categories:

1. Internal sorting and
2. External sorting

Internal sort: When a set of data is small enough such that entire sorting can be performed in a computer's internal storage (primary memory) then the sorting is called internal sort.

External sort: Sorting of a large set of data, which is stored in low speed computer's external memory (such as hard disk, magnetic tape, etc) is called external sort. It involves large amount of data transfer between external memory (low speed) and main memory (high speed).

Sorting Technique: A technique that arranges the data in ascending or descending order.

Types of Sorting Techniques:

The sorting techniques are:

1. Selection sort
2. Bubble sort
3. Insertion sort
4. Shell sort
5. Quick sort
6. Merge Sort

Bubble Sort

An algorithm to sort the data into ascending order using Bubble sort.

A	:	Vector (Array)
N	:	Number of elements in a vector
PASS	:	Pass counter
LAST	:	Position of last unsorted element
I	:	Index (subscript) used for vector elements
EXCHS	:	Used to count number of exchanges done in any pass

Step -1 : [Initialize **]**

LAST ⟵ N

Step -2 : [Loop on pass index **]**

Repeat through step – 5 for PASS = 1, 2, 3, ... N - 1

Step -3 : [Initialize exchange variable EXCHS **]**

EXCHS ⟵ 0

Step -4 : [Perform pair wise comparison **]**

Repeat for I = 1, 2, ... LAST – 1

IF (A[I] > A[I+1])

THEN

A[I] ⟷ A[I+1]

EXCHS ⟵ EXCHS + 1

Step -5 : [Exchange made or not **]**

IF (EXCHS = 0)

THEN

Return (Mission accomplish, return early)

ELSE

LAST ⟵ LAST – 1 (Reduce size of unsorted array)

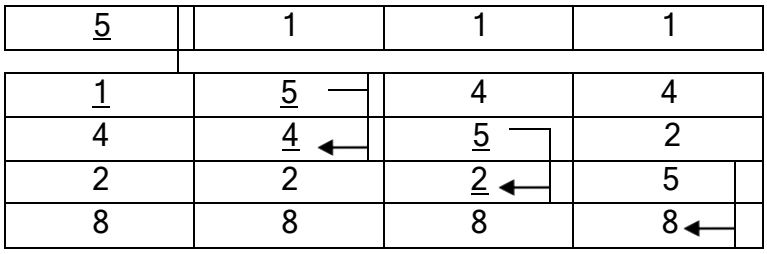
Step -6 : [Finished **]**

Return

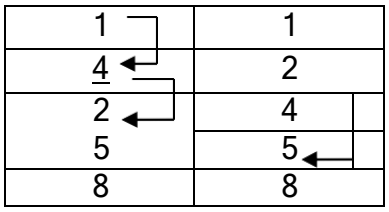
Bubble sort dry run

Given array: 5, 1, 4, 2, 8

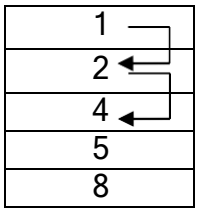
Pass-1



Pass-2



Pass-3



Advantages of Bubble sort

1. Easy to understand.
2. Easy to implement.
3. Better algorithm for almost sorted data.

Disadvantages of Bubble sort

Large amount of data movement required if data is in random order or reverse sorted order.

Merge Sort

Algorithm: SIMPLE_MERGE (K, FIRST, SECOND, THIRD)

This algorithm sorts elements into ascending order.

K	Vector (Array) contains two ordered arrays
TEMP	Temporary vector
FIRST	Position of first element of First vector in K vector
SECOND	Position of first element of Second vector in K vector
THIRD	Position of last element of Second vector in K vector
I	Index (subscript) used for first vector elements
J	Index (subscript) used for second vector elements
L	Index (subscript) used for TEMP vector elements

1.	[Initialize] I←FIRST J ←SECOND L ←0

2	[Compare corresponding elements and output the smallest]
	Repeat while I < SECOND and J ≤ THIRD
	If K[I] ≤ K [J] then
	L ← L + 1
	TEMP [L] ← K[I]
	I←I + 1
	Else
	L ← L + 1
	TEMP [L] ← K[J]
	J ← J + 1

3.	[Copy the remaining unprocessed elements in output area]
	If I≥ SECOND then
	Repeat while J ≤ THIRD
	L ← L + 1
	TEMP [L] ← K[J]
	J ← J + 1
	Else
	Repeat while I< SECOND
	L ← L + 1
	TEMP [L] ← K[I]
	I←I + 1

4.	[Copy elements of temporary vector into original area] Repeat for I = 1, 2, ... , L K [FIRST – 1 + I] ←TEMP [I]
5.	[Finished] Return

Merge sort dry run

Given array: 11, 23, 42, 9, 25

Array-1	Array-2	Temp array
11 →	~~9~~	9
23	25	
42		

~~11~~ →	25	9
23		11
42		
~~23~~ →	25	9
42		11
		23

42 →	~~25~~	9
		11
		23
		25
		42

Advantages of Merge sort

Easy to merge already sorted lists into a new sorted list with merge sort.

Disadvantages of Merge sort

Merge sort requires extra storage space for temporary vector.

Searching: It is the process to find the particular element from the array.

Searching Technique: A technique that find the particular element from the array.

The searching techniques are:

1. Linear Search or Sequential search
2. Binary Search

LINEAR SEARCH

This algorithm searches an element from unordered / ordered array.

A : Array consist N elements
N : Number of elements in a array
X : Element to be searched

Step -1 : [Initialize **]**

I = 1
Loc = 0

Step -2 : Repeat step 3 and 4 while i<=n and loc =0

Step -3: If x = a[i]

Set loc = i

Step -4: I = i+1

Step -5: if loc = 0

Write ("unsuccessful search")

Else

Write ("successful search: element found at loc position")

Advantages of Linear search

1. Linear searching is the basic and simple method of searching.
2. Easy to implement.
3. Useful for searching an element in an unordered or ordered list.

Disadvantages of Linear search

Linear search is time consuming.

BINARY SEARCH

BINARY_SEARCH(A, N, X) : This algorithm is used to find a particular elements X from the array A which is consisting of N elements using binary search. It returns the index of the array if the element is found, otherwise returns 0.

Step -1 : [Initialize **]**

LOW ⟵ 1

HIGH ⟵ N

Step -2 : [Perform Search **]**

Repeat through step – 4 while LOW <= HIGH

Step -3 : [Calculate MIDDLE **]**

MIDDLE ⟵ (LOW + HIGH) / 2

Step -4 : [Compare **]**

IF (X < A [MIDDLE])

```
THEN
        HIGH ◄——— MIDDLE - 1
ELSE
        IF ( X > A [ MIDDLE ] )
        THEN
                LOW  ◄———  MIDDLE + 1
        ELSE
                WRITE ('SUCCESSFUL SEARCH')
                Return ( MIDDLE )
```

Step -5 : [Unsuccessful search **]**

WRITE ('UNSUCCESSFUL SEARCH')

Return (0)

Advantages of Binary search

1. Binary search is very efficient algorithm.
2. Require fewer number of comparisons as compared to Linear search.

Disadvantages of Binary search

1. Binary search is not useful when the array elements are frequently changed.
2. Array must be sorted to perform binary search.

Sorting V/s Searching:

	Sorting		Searching
1.	It is the process to arrange the data in ascending or descending order.	1.	It is the process to find the particular element from the array.
2.	The most common sorting algorithms are: • Bubble Sort • Insertion Sort • Selection Sort • Quick Sort • Merge Sort • Shell Sort	2.	The most common searching algorithms are: • Linear search • Binary search • Interpolation search
3.	This is the operation of arranging the elements of a table into some sequential order	3.	This is the process by which one searches the group of elements for the desired element.
4.	Sorting returns an array with the elements arranged in ascending or descending order.	4.	Searching returns the position of the searched array
5.	Output of sorting algorithms is sorted elements.	5.	Output of searching algorithm is successful or unsuccessful search.
6.	After performing sorting, searching becomes easy.	6.	Without performing sorting, searching becomes difficult.
7.	After performing sorting techniques, the position of data elements or data records are changed.	7.	After performing searching techniques, the position of data elements or data records are not changed.