Unit 2

Linked Lists

	Syllabus	Guidelines	Suggested number of lectures
1	Singly- linked, doubly-linked and circular lists, analysis of insert, delete and search operations in all the three types	Chapter 3, Section 3.2, 3.3, 3.4, **Ref 2**	12
2	Implementing sparse matrices.	Chapter 7 (Section 7.4.upto pg. no. 253) **Additional Resource 3**	
3	Introduction to Sequences.	Chapter 6, Section 6.3, **Ref 2**	

References

1. Ref 1: . Drozdek, A., (2012), *Data Structures and algorithm in C++*. 3rd edition. Cengage Learning. Note: 4th edition is available. Ebook is 4th edition

2. Ref 2.: Goodrich, M., Tamassia, R., & Mount, D., (2011). *Data Structures and Algorithms Analysis in C+ +*. 2nd edition. Wiley.

3. Additional Resource 3: Sahni, S. (2011). Data Structures, Algorithms and applications in C++. 2ndEdition, Universities Press

4. Additional Resource 4: Tenenbaum, A. M., Augenstein, M. J., & Langsam Y., (2009), *Data Structures Using C and C++*. 2nd edition. PHI.

Note: Ref1, Additional resource etc. as per the LOCF syllabus for the paper.

ch-3. Linked Lists (A. Drozdek).

Limitation of Arrays

1). Its size has to be known at compilation time.
2). The data in the array are separated in computer memory by the same distance.

These limitations can be overcomed by using linked list structures.

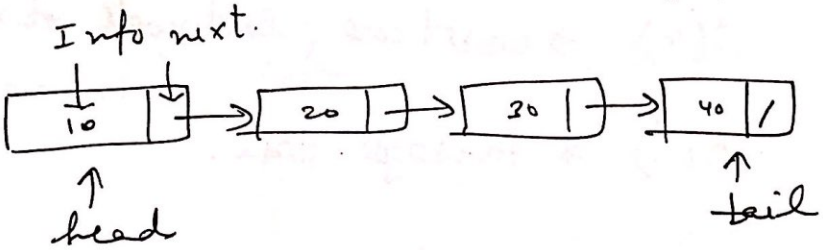
Linked structure :- is a collection of nodes storing data & links to other nodes.

Types of Linked List

1) Singly linked list.
2) Doubly linked list.
3) circular singly l.list.
4) circular doubly l.list.

(I) Singly linked list

Sequence of nodes, in which each node is holding some information & pointer to another node (successor) in the list, is called singly linked list.



Self-Referential objects :- 'node' is defined in terms of itself because one data member 'next' is pointer to a node of the same type. objects that include such datamembers are called self-referential objects.

```
class node
{
Public:
        int info;
        node* next;
};
```

```
Reverse()
{   x1 = last;
    do
    {
       x = first;
       while (x -> ptr != x1)
                    x = x -> ptr;
       x1 -> ptr = x;
       x1 = x;
    } while (x1 != first);
   first -> ptr = NULL;
   temp = last;
   last = first;
   first = temp;
}
```

Analysis :-

1) addtohead() :- O(1) :-
2) addtotail() :- O(1) :- } no. of operation performed by these two f^n does not exceed some constant no. 'c'.

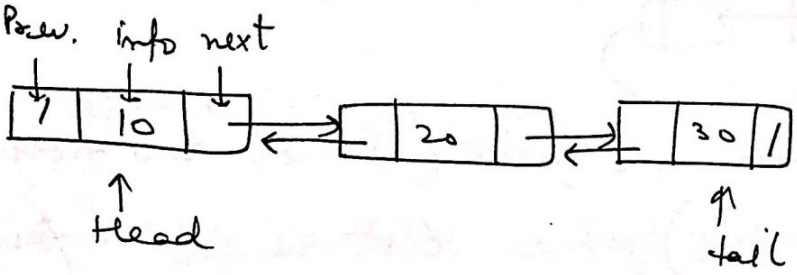
3) Deletfromhead() :- O(1) [Best case]

4) Deletefromtail() :- since for loop performs 'n-1' iterations in a list of 'n' nodes to reach second last node, it takes O(n) time to delete last node. [worst case]

5) searchnode() :- O(1) → Best case (first node)
O(n) → worst case (last node or no node)
O(n) → Average case.

3.2. Doubly linked list

Each node in the list has two pointers :-
one to the sucessor & one to the predecessor.




```
class node
{ public:
        int info;
        node* prev;
        node* next;
};
```

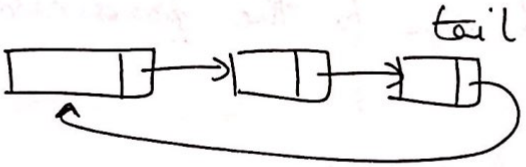
Analysis :-

1) Addto head() = O(1)
2) Addto tail = O(1)
3) Delete from head() = O(1)
4) Delete from tail() = O(1)
5) search node (-) = O(n).

3.3 circular Lists

@ C. singly.

All the nodes form a ring structure.

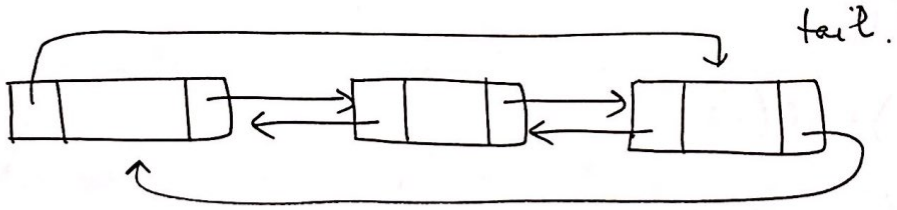


single pointer 'tail' is there (& not two pointers - head/tail as in SLL) since list is in the form of ring.

Analysis :-

1) Insertatbegin() = O(1)
2) Insertatend() = O(1)
3) Deletefromtail() = O(n)
4) Deletefromhead() = O(1)

(b) circular Doubly Linked list.



The list forms two rings :-
one going forward through 'next' members and one going backward through 'Prev' members.

Analysis

(1) Insertatend() = O(1)
(2) Insertatbegin() = O(1)
(3) Deletefromtail() = O(1)
(4) Deletefromhead() = O(1).

Destructors

(1) for singly link list:-

```
~ SLL();  // declaration.

SLL<t> :: ~ SLL()
{
    for ( node<t> * temp ; ! isempty(); )
    {
        temp = first -> next;
        delete first;
        first = temp;
    }
}

class SLL
{
    ==

    t isempty()
    {
        return (first == NULL);
    }
}
```

Changes in Linked List when Templates are used

①
```
templete < class X >
class node
{

}

templete <class X>
class List
 {
→   node <X> * head, *tail;

}
```

② `List <X>* operator + (List <X>&);`

③ Def^n of f^n s.
```
templete <class X>
Void List<X> :: reverse()
 {

 }
```

④
```
templete <class X>
list <X> *  List <X> :: operator + (List<X> &p)
{

}
```

⑤ Destructor
```
~SLL ()          // using for
{  for (node* p ; head != 0 ; )
        {  p = head → next;
           delete head;
        } head = p;
}
```

```
// using while
~SLL ()
{  node * p;
   while(head != 0)
     {  p = head → next;
        delete head;
     } head = p;
}
```

⑥
```
Void main()
{
cout<<"welcome to
   interger format";

SLL < int > ob, ob1 ;
      :
      :
      .

cout<<" character format";
 SLL <char> ob3, ob1;
      :
      :
      .

}
```

3.4 skip lists

Drawback of Linked List:- To search a particular element, it required sequential scanning from first node to last until the element is found or list has terminated giving no result.

solution:-

1) ordering elements (ascending / desc) on the list can speed up searching, but sequential search is still required.
2) There can be a list that allow for skipping certain nodes to avoid sequential processing.
3) A skip list is one of the sol^n of ordered linked list that makes such a non-sequential search possible.

In a skip list of 'n' nodes, for each 'k' & 'i', such that

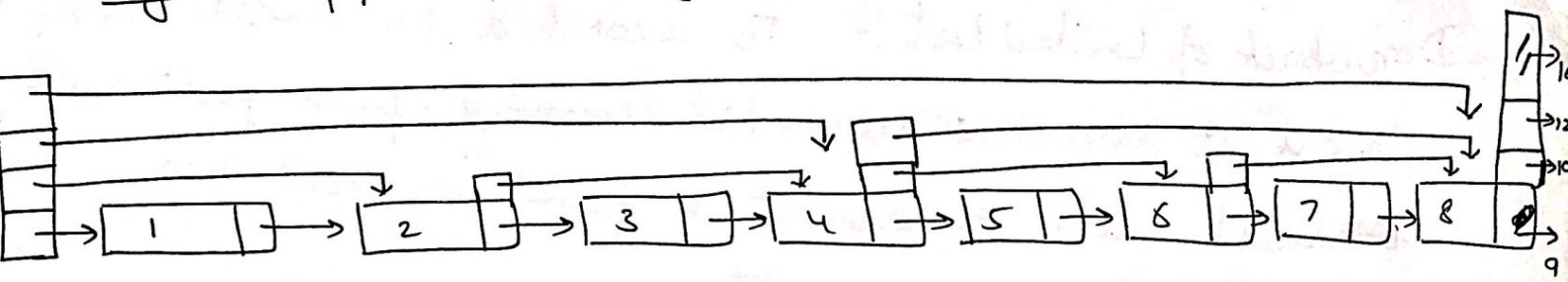
$1 \leq k \leq \lfloor \log_2 n \rfloor$ and

$1 \leq i \leq \lfloor n / 2^{k-1} \rfloor - 1$

The node in position

$a^{k-1}(i)$ ——Points to——→ the node in pos. $a^{(k-1)}(i+1)$

Eg. Suppose $n = 8$ (8 nodes in list)



Now, $(2^{k-1})(i) \xrightarrow{\text{pts. to}} (2^{k-1})(i+1)$

Also, $1 \le K \le \lfloor \log_2 n \rfloor$ ie $1 \le K \le \lfloor \log_2 8 \rfloor$ or

$\boxed{1 \le K \le 3.}$

and. $1 \le i \le \lfloor n / 2^{K-1} \rfloor - 1$ ie if $K = 1$, then

$1 \le i \le \lfloor 8/1 \rfloor - 1$ ie $\boxed{1 \le i \le 7.}$

$K = 1$, $i = 1 \ldots 7$.

$K = 1$, $i = 1$, $2^{1-1}(1) \xrightarrow{\text{pts. to}} 2^{1-1}(2)$ ie $1 \longrightarrow 2$

$i = 2$, $2 \longrightarrow 3$

$i = 3$, $3 \longrightarrow 4$

$\vdots$

$i = 7$, $7 \longrightarrow 8$.

$K = 2$, i is $1 \le i \le \lfloor 8/2^1 \rfloor - 1$ ie $1 \le i \le 3$.

$K = 2$, $i = 1$, $2^{2-1}(1) \xrightarrow{\text{pts. to}} 2^{2-1}(1+1)$ ie $2 \xrightarrow{\text{pts. to}} 4$

$i = 2$, $4 \longrightarrow 6$

$i = 3$, $6 \longrightarrow 8$.

$K = 3$, i is $1 \le i \le \lfloor 8/2^2 \rfloor - 1$ ie $1 \le i \le 1$

$K = 3$, $i = 1$, $2^{3-1}(1) \xrightarrow{\text{pts. to}} 2^{3-1}(2)$ ie $4 \longrightarrow 8$

Half of the nodes have just one pointer (4) ie nodes 1, 3, 5, 7.
one-fourth of the nodes have two ~~too~~ pointers (2) ie nodes 2, 6.
one-eighth of the nodes have three pointers (1) ie node 4.

The no. of pointers indicates the level of each node.
ie node 1 $\xrightarrow{\text{has}}$ level 1
node 2 has level 2 & so on.

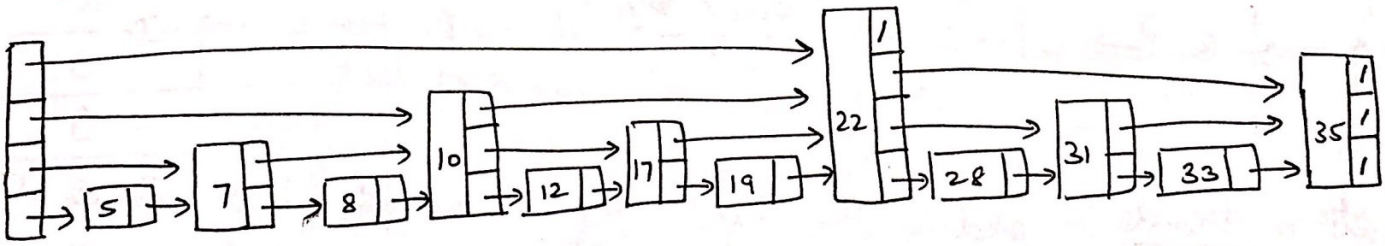
maximum levels :- $\lfloor \log_2 n \rfloor + 1$ ie $\log_2 8 = 3$. in eg. ie 3+1 = 4 level
last node has 4 levels.
(8)

~~$\lfloor \log_2 n \rfloor + 1$~~

Disadvantages of skip list

- Design of skip list can lead to very inefficient insertion & deletion procedures.

eg. $n = 12$



3.5 self-organizing lists

We can improve the efficiency of search by dynamically organizing the list in a certain manner. There are many different ways to organize the lists :-

1) **move-to front method** :- After the desired element is located, put it at the beginning of the list.

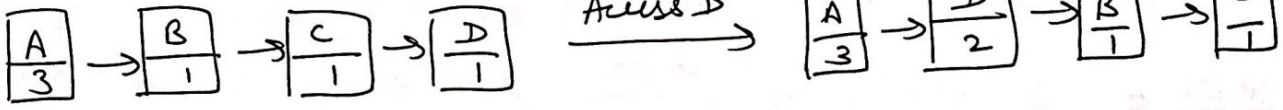
$$A \to B \to C \to D \xrightarrow{\text{Access D}} D \to A \to B \to C.$$

2) **Transpose method** :- After the desired element is located, swap it with its predecessor, unless it is at the head of the list.

eg 1. $A \to B \to C \to D \xrightarrow{\text{Access D}} A \to B \to D \to C$

eg 2. $A \to B \to C \to D \xrightarrow{\text{Access A}} A \to B \to C \to D.$
(A is at the head of list)

3) **count method** :- order the list by the no. of times elements are being accessed



4) **ordering method** :- order the list using certain criteria natural for the information under scrutiny.

$$A \to B \to D \xrightarrow{\text{Access C}} A \to B \to C \to D$$

In first three methods, new inf" is stored in a node added to the end of list.

for eg:-

$A \rightarrow B \rightarrow D$. Access C (move-to front) $A \rightarrow B \rightarrow D \rightarrow C$

Access C (transpose) $A \rightarrow B \rightarrow D \rightarrow C$

Access C (count) $A \rightarrow B \rightarrow D \rightarrow C$

But in fourth method, new inf" is stored in a node inserted somewhere in the list to maintain the order of the list.

ie

$A \rightarrow B \rightarrow D$ Access C (ordering) $A \rightarrow B \rightarrow C \rightarrow D$

eg: element searched for	plain	move to front	transpose	count	ordering.
A	A	A	A	A_1	A
C	AC	AC	AC	$A_1 C_1$	AC
B	ACB	ACB	ACB	$A_1 C_1 B_1$	ABC
C	ACB	CAB	CAB	$C_2 A_1 B_1$	ABC
D	ACBD	CABD	CABD	$C_2 A_1 B_1 D_1$	ABCD
A	ACBD	ACBD	ACBD	$C_2 A_2 B_1 D_1^{\times}$	ABCD
D	ACBD	DACB	ACDB	$C_2 A_2 D_1 B_1^{\times}$ (DC AB) → Break	ABCD
A	ACBD	ADCB	ACDB	$A_3 D_2 C_2 B_1$	ABCD
C	ACBD	CADB	CADB	$C_3 A_3 D_2 B_1$	ABCD
A	ACBD	ACDB	ACDB	$A_4 C_3 D_2 B_1$	ABCD
C	ACBD	CADB	CADB	$(A_4 C_4 D_2 B_1)$ ☆	ABCD
C	ACBD	CADB	CADB	$^{5}C_4 A_3 D_2 B_1$	ABCD
E	ACBDE	CADBE	CADBE	$E_5 A_4 D_3 B_2 E_1$	ABCDE
E	ACBDE	ECADB	CADEB	$^{5}C_4 A_3 \cancel{D_2} D_2 B_1$ DE	ABCDE

special matrices

A square matrix has same no. of rows & columns.
special form of square matrices are:-

1) Diagonal:- M is diagonal iff $m(i,j) = 0$ for $i \neq j$.

$$\begin{matrix} & j \\ i & \end{matrix}\begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 \end{bmatrix}$$

2) Tridiagonal :- M is tridigonal iff $m(i,j)=0$ for $|i-j| > 1$

$$\begin{matrix} & j \\ i & \end{matrix}\begin{bmatrix} 2 & 1 & 0 & 0 \\ 3 & 1 & 3 & 0 \\ 0 & 5 & 2 & 7 \\ 0 & 0 & 9 & 0 \end{bmatrix}$$

3) Lower Triangular :- M is lower-Triangular iff $m(i,j)=0$ for $i < j$.

$$\begin{bmatrix} 2 & 0 & 0 & 0 \\ 5 & 1 & 0 & 0 \\ 0 & 3 & 1 & 0 \\ 4 & 2 & 7 & 0 \end{bmatrix}$$

4) upper Triangular :- M is upper Triangular iff $m(i,j) = 0$ for $i > j$.

$$\begin{bmatrix} 2 & 1 & 3 & 0 \\ 0 & 1 & 3 & 8 \\ 0 & 0 & 1 & 6 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

5) <u>symmetric</u> :- Matrix M is symmetric iff ~~M(i,j) = M(j,i)~~

$M(i,j) = M(j,i)$ for all i & j.

$$\begin{bmatrix} 2 & 4 & 6 & 0 \\ 4 & 1 & 9 & 5 \\ 6 & 9 & 4 & 7 \\ 0 & 5 & 7 & 0 \end{bmatrix}$$

<u>Sparse Matrices</u>.

An $m \times n$ matrix is said to be sparse if "many" of its elements are zero.

A matrix that is not sparse is dense.

eg:- Diagonal & Tridigonal matrices are sparse.
Each has $O(n)$ nonzero terms & $O(n^2)$ zero terms.

Triangular matrix (lower & upper) has —

atleast $\frac{n(n-1)}{2}$ zero terms &

atmost $\frac{n(n+1)}{2}$ non-zero terms.

The no. of non-zero terms in SPARSE matrices need to be less than $n^2/3$ & in some cases $n^2/5$.

In this context, triangular matrices are dense.

eg:- 4 × 8 sparse matrix

$$\begin{bmatrix} 0 & 0 & 0 & 2 & 0 & 0 & 1 & 0 \\ 0 & 6 & 0 & 0 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 9 & 0 & 8 & 0 & 0 \\ 0 & 4 & 5 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

its representation

a[]	0	1	2	3	4	5	6	7	8
row	1	1	2	2	2	3	3	4	4
col	4	7	2	5	8	4	6	2	3
value	2	1	6	7	3	9	8	4	5

Linked Representation