31/10/22

__/__/____

Trees (Non-linear DS).

Tree consists of:

1) Distinguished node r: root node.
2) and zero or more non-empty (sub)trees. $T_1 \ldots T_k$ each of whose roots are connected by a directed edge from r.

- Every node is root of some sub tree.
- A tree can also be empty.

- In a tree with n nodes; the number of edges are $(n-1)$. Each node other than the root node is associated with an edge.

- Normally root is drawn at the top.

Level of a Node:

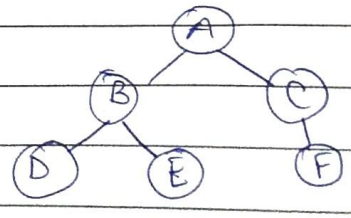
- Defined by letting the root be at level one.
If a node is at level L, its child nodes are at level $L+1$.

Degree of a node:

degree (A) = no. of sub trees of that node.

- Degree can also be 0.

degree (E) = 0
degree (A) = 2



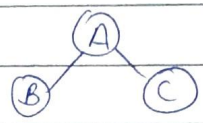
__/__/__

Degree of a tree:

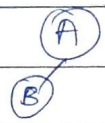
- max { degree(node) }.
- maximum degree of any node.

Tree is a Graph which is:
- ↳ connected
- ↳ acyclic (cycle not possible).

Parent: Node of a subtree.
A is parent of B and C.



Children: Root of the subtree.
B is a child of A.



Sibling: children of same parent.
B & C.

Leaf (Terminal node):
Degree = 0 (no children).

Deg > 0 => non-leaf (non-terminal) node.

Path from n_1 to n_k:
- Unique sequence of nodes: n_i is the parent of n_{i+1} for $1 \leq i < k$.

__/__/____

Length of path:
- Number of edges on the path.

Depth of ni:
- Length of unique path from the root to ni. (same as level).

Depth (root) = 1.

Height of ni:
- Length of longest path from ni to leaf.

Height (depth) of a tree:
height (root).
depth (deepest root).

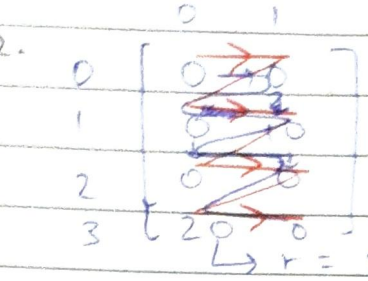
Ancestors of a node:
- all the nodes along the path from the node up to the root.

Decendants of a node:
- all nodes in the subtrees.

	rows (r)	columns (c)	v → no. of non-zero elements.
a[0]	4	2	3 (for zeroth value)
a[1]	1	1	10
a[2]	3	0	20

non-zero elements

This and * are always same value.



- we read from left to right in each row.

```
class sparce.
{
    int row, col, val;
    public:
        sparce() { row = col = val = 0; }
        sparce(int r, int c, int v)
        {
            row = r; col = c; val = v;
        }
        void slowTranspose(sparce *);
        void fastTranspose(sparce *);
        void display(sparce *);
                    ↳ (sparce a[]);
};
void main()
{
        ↳ array of objects.
    sparce a[10];
    cout << "Enter no. of non-zero elements",
                                rows & columns.
    cin >> n;
    cin >> r >> c;
    a[0] = sparce(r, c, n);
```

array of objects as parameter.

```
for (i=1; i<=n; i++)
{
    cout<<"Enter row, col, val";
    cin>>r>>c>>v;
    s[i] = sparce (r, c, v);
}
}
```

Sparce Matrix:

	0	1	2	3
0	0	2	0	0
1	1	0	0	0
2	0	0	0	4
3	0	3	0	0

No. of zeroes = 12. (3)

No. of non-zeroes (nz) = 4

	Row	Col	Val
a[0]	4	4	4
a[1]	0	1	2
a[2]	1	0	1
a[3]	2	2	4
a[4]	3	1	3

Slow Transpose:

	Row	Col	Val
b[0]	4	4	4
b[1]	0	1	1
b[2]	1	0	1
b[3]	1	3	3
b[4]	2	2	4

```
void SM::slowtranspose(SM a[])
{
    SM b[10]; int ctr=1; pass;
    for(pass=0; pass<a[0].col; pass++)
    {
        for(obj=1; obj<=a[0].val; obj++)
        {
            if(a[obj].col==pass)
            {
                b[ctr].col=a[obj].row;
                b[ctr].row=a[obj].col;
                b[str].val=a[obj].val;
                ctr=ctr+1;
            }
        }
    b[0].row=a[0].col;
    b[0].col=a[0].row;
    b[0].val=a[0].val;
}
void SM::display(SM a[])
{   int obj=0;
    cout<<"\t"<<rows<<"\t"<<col<<"\t"<<
                val;
    while(obj<=a[0].val)
    {
        cout<<"\t"<<a[obj].row;
        cout<<"\t"<<a[obj].col<<"\t"<<
                a[obj].val;
        obj=obj+1;
    }
}
```

Slow:

```
for pass = 0 to a[0].c → Inclusive.
    for obj = 1 to a[0].v → Inclusive.
        if (a[obj].col == pass)
            we're checking column.
            b[ctr].r = a[obj].c;
            b[ctr].c = a[obj].r;
            b[ctr].val = a[obj].val;
```

exchange row & column. ← `b[0].r = a[0].c ; b[0].c = a[0].r ; b[0].v = a[0].val`

- → Columns → rows.
- → Rows → column.
- → values remain same.
- → Rows of new matrix are in ascending order.
- → columns of each row are in ascending order.
- → Time - space & : c * e (much more).

Fast Transpose:

- we compare each object.
- input matrices, start position.
- Time - space : c + e (lesser)
  => better & faster.

16/9/22

	C_0	C_1	C_2	C_3
r_0	0	1	4	0
r_1	0	0	3	0
r_2	0	5	0	6

3×4.

no. of non-zero terms in that column.

	0	1	2	3
rowterm	0	2	2	1
start posn	1	1	3	5

↳ we start with these for rowterm

no. of rows.

	r	c	v
a[0]	3	4	5
a[1]	0	1	1
a[2]	0	2	4
a[3]	1	2	3
a[4]	2	1	5
a[5]	2	3	6

Transpose =

	r	c	v
b[0]	4	3	5
b[1]	1	0	1
b[2]	1	2	5
2nd → b[3]	2	0	4
b[4]	2	1	3
3rd → b[5]	3	2	6

$$\begin{bmatrix} 0 & 10 & 5 & 0 \\ 1 & 0 & 6 & 0 \\ 2 & 4 & 0 & 8 \end{bmatrix}$$

	r	c	v
a[0]	3	(4)	7
a[1]	0	(1)	10
a[2]	0	(2)	5
a[3]	1	(0)	1
a[4]	1	(2)	6
a[5]	2	(0)	2
a[6]	2	(1)	4
a[7]	2	(3)	8

by comparison.

	0	1	2	3
nwoterm	2	2	2	1
startpos	1	3	5	7

	r	c	v
b[0]	4	3	7
b[1]	0	1	1
b[2]	0	2	2
b[3]	1	0	10
b[4]	1	2	4
b[5]	2	0	5
b[6]	2	1	6
b[7]	3	2	8

19/9/22.

→ Code for fast transpose:

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

	↓	↓	↓	↓
	0	1	2	3
Rowterm	2	2	0	1
Start posn	1 →	3 →	5 →	5

start posn [0] = 1. start pos [i] = startpos [i-1] + rowterm [i-1].

	r	c	v
a[0]	3	4	5
a[1]	0	0	1
a[2]	1	0	2
a[3]	1	1	3
a[4]	1	3	5
a[5]	2	1	4

Code:

```
void SM:fastTranspose(SM a[])
{
    int rowterm[20], startpos[20];
    for (i=0; i<a[0].col; i++)
        rowterm[i]=0;
    for (i=1; i<=a[0].val; i++)
        rowterm[a[i].col]++;
    startpos[0]=1;
```

```
for (i=1, i<= a[0].col; i++)
    startpos[i] = startpos[i-1] +
                  rowterm[i-1];
b[0].col = a[0].row;
b[0].row = a[0].col;
b[0].val = a[0].val;
int b_i;
for (i=1; i<= a[0].val; i++)
{
    b_i = startpos[a[i].col];
    b[b_i].col = a[i].row;
    b[b_i].row = a[i].col;
    b[b_i].val = a[i].val;
    startpos[a[i].col]++;
}
cout << "Transpose array of objects:");
display(b);
}
```

	r	c	v
b[0]	4	3	5
b[1]	0	0	1
b[2]	0	1	2
b[3]	1	1	3
b[4]	1	2	4
b[5]	3	1	5

	0	1	2
0	0	0	0
1	1	0	5
2	2	3	0
3	0	4	0

4×3.

	r	c	v
a[0]	4	3	5
a[1]	1	0	1
a[2]	1	2	5
a[3]	2	0	2
a[4]	2	1	3
a[5]	3	1	4

→ we take 1st element of 0th of column.
- we check start posn of that number.
- we place that no. in the startposn no. of row & increment its start posn by 1.
- we swap values of r & c while writing to b.
- value of v remains same.

	0	1	2
& rowterm	2	2	1
startposn	1	3	5

+ This is the sum of the two.

This startposn is always 1.

	r	c	v
b[0]	3	4	5
b[1]	0	1	1
b[2]	0	2	2
b[3]	1	2	3
b[4]	1	3	4
b[5]	2	1	5

000 X

r_0, r_1, r_2

→ 000

__/__/__

Pointers:

→ memory location or a variable which stores the memory address of another variable or memory.

Uses:

- Accessing array elements.
- call by reference (arrays & strings to fns).
- dynamic allocation & deallocation of memory.
- linked lists.
- increase execution speed.
- reduces complexity.

↳ address of (operator).

```
p = &op;
    ↳ used to get address.
```

→ &(x+y): invalid since it does not have a memory location.

```
int *p;
    ↳ declares variable p as pointer.
    ↳ this points only to variable that is of
      integer type.
float *f; → only to variable of float type.
```

& : reference operator. (address of).
* : dereference operator. (value pointed by).

```
n = *p;
  ↳ it gives the value stored in address p.
      (value pointed by p).
```

```
int *p = &a, d;
```
↳ Invalid. Cannot declare anything after pointer.

Scale Factor:

- increment or decrement given to a pointer.
- changes the address of pointer to next memory address.
- when we increment a pointer, its value is increased by the length of the data type that it points to.

This length is called scale factor.

```
p1++;   pointing to initial value = 2800
  => p1 = p1 + 1
      => value of p1 = 2802.
```

7/10/22.

→ Array name is a pointer which represents base address (location of zeroth element).

```
x[10] = {11, 21 ....}
```
↳ value of x is memory location of 11.

$x = \&x[0]$.

→ When array is declared, compiler allocates base address & amound of memory required.

```
int x[5] = {1, 2, 3, 4, 5};
int *p;
p = x;  OR  p = &x[0];
```
p = &x : Invalid. (It won't point at base address; it will copy address of x).

Successive array elements can be accessed:

```
cout << *p;          or        cout << *(p+i);
p++;                           i++;
```

↓

increments by size of the pointer's datatype.
(for int : 2 bytes).
(memory is increment so 1 increment : 2 byte increment for integer.

```
p[0] ≡ *(p+0)
p[1] ≡ *(p+1)
p[2] ≡ *(p+2)
   and so on.
```

cout << *(p+i)
(in for loop, will also print the array).

- for float value; increment by 1 = increment by 4 bytes.

if p = 1000
p+1 = 1002
p+2 = 1004

content = *(&x[i])
↳ (p+i) : address.

```
for(j=0; j<5; j++)
    cout << *p++;
```

} prints elements of array p.

→ We can write diff. programs using this. (Exchange values, etc).

Pointers and Character Strings:

```
char name[] = "Example";
char *cptr = name;
while(*cptr != '\0')
    cptr++;

length = ctpr - name;
```

address of final char. — base address (does not change).

size of char = 1.

E	1000
X	1001
A	1002
M	1003
P	1004
L	1005
E	1006
\0	1007

$a[i][j] \equiv *(*(a+i)+j)$

i → row, j → column

Array of Pointers:

name[0] → "String 1"
name[1] → "String 2"
name[2] → "String 3"

Instead of 2D array like name[3][25], we can also use array of pointers.

=> `char *name[3] = {"---", "---", "---"};`

↳ we can have variable length of each where as in char[3][25], 25 is fixed.

- 25 memory locations are allocated whether we use or not (garbage value).
- 25 is the maximum length.
- In pointers we can have varying no. of length (no max^m or garbage value).

```
for (i=0, i<=2; i++)
    cout << name[i];
```

Pointers to Structures:

```
struct name
{
    data members;
};
```

→ There are no fns inside struct.

```
struct inventory
{
    char name[30];
    int number;
    float price;
};   → we can also initialize : {product[2], *ptr;
inventory product[2], *ptr;
```

ptr = product; (ptr = &product[0];).
ptr → name : product[0].name
ptr → number : product[0].number
ptr → price : product[0].price
ptr = &product[1];
ptr → name : same as product[1].name.

member
→ : selection operator. (arrow operator).

ptr → name OR (*ptr).name
ptr is holding address.
*(ptr).name : () is req. because . has higher precedence than *.

Returning Pointers:

```
int * larger (int * x, int *y)
{
    if (* x > * y)
        return (x);
    else
        return(y);
{
```

Dynamic Memory Allocation:

→ constructor is called.

new is used.

```
int *p; float *q;
p = new int;
q = new float;
```

OR

```
int *p = new int;
float *q = new float;
```

(we can write in one line).

```
int *p = new int(25); (memory of 1)
int *p = new int[25]; (array is allocated & base address is returned).
```

```
p = new int[3][4][5]; } Multidimensional
p = new int[m][3]; }    allocation.
```

Dynamic Memory De-allocation:

→ destructor is called.

delete is used.

```
delete p;
```

__/__/__

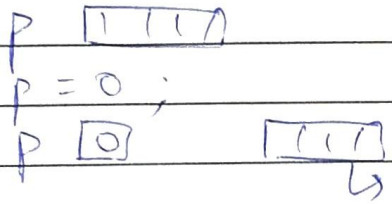
Dangling pointer problem:

```
delete p;
p = 0;      (p = NULL)
```

→ Pointer pointing to deallocated memory.

Memory Leak problem:

Memory which is no longer needed is not released.



↳ we do not know how to fetch this memory.

→ pointer got diff. value; previous memory is still there.

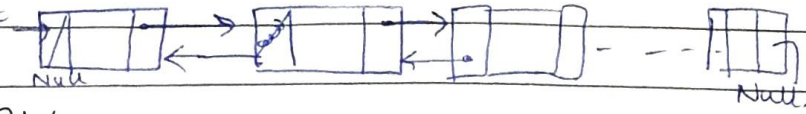
8/10/22

Linked List:

- Linear DS.

Types:

SLL: Singly.

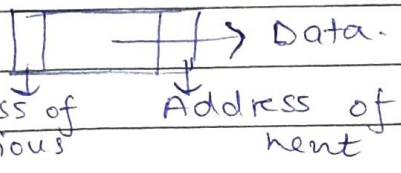


DLL: Doubly. →

CSLL: Circular SLL.

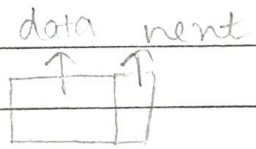
CDLL: Circular DLL.

DLL:



→ Dynamic in nature.

Code:




```
class SLL
{
    int data;
    SLL * ~~node~~ next; // pointer to class object.
public:
    // set of operations on SLL.
};
SLL *first = NULL;
```

global scope

↳ Address of 1st node.

or `};` SLL *first = NULL; (scope should be global).

```
void main()
{
    SLL obj;
```

Operations on SLL:

1) Inserting a new node (At the end of list).

ASCII value of NULL = 0.

__/__/____

2) Inserting new node in the beginning.
3) Count no. of nodes.
4) Display
{

insert → begin.
→ end.
→ before value.
→ after value.

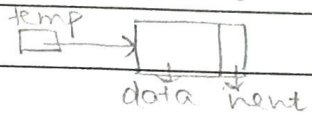
del → first
→ last
→ deleting a node with specific value.

Insert at end :

```
void SLL::ins_end()
{
    SLL *temp = new SLL;
```

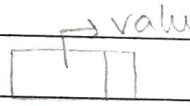
new object is created which has two parts → data → node (address).

- temp holds address of newly created node.




```
    cout << "Enter value";
    cin >> temp -> data;
```

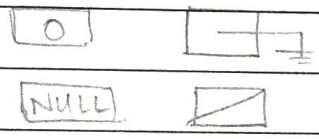
→ value will be saved here.



```
    temp -> next = NULL;
```

→ next is set to null.

```
    if(first == NULL) //Empty condition
```



this can also be written as if(!first).

```
        first = temp;

    else
    {
```

→ no body.

→ independent / stand alone for loop

```
    for(SLL *curr = first; curr -> next != NULL;
                curr = curr -> next);
```

curr holds address of last node

```
            curr -> next = temp;
        } // else.
    } // ins_end.
```

10/10/22.

Insert front:

```
void SLL :: ins_front()
{
    SLL * temp = new SLL;
    cout << "Enter a value";
    cin >> temp -> data;
    if (first == NULL)
        first = temp;
    else
    {
        temp -> next = first;
        first = temp;
    } // else.
} // ins_front.
```

Display:

```
void SLL:: display()
{
    if (first == NULL)
        cout << "List is empty";
    else
    {
        SLL * curr = first;
        while ( curr != NULL)
```

```
    {
        cout << curr -> data << "  ";
        curr = curr -> next;
    } // while.
  } // else.
} // display.
```

→ visiting each node until current becomes null and displaying it.

Count no. of elements :

```
int → returning no. of elements.
void SLL :: count()
{
    int c=0;
    if (first == NULL)
        cout << "Empty";
    else
    {
        SLL * curr = first
        while (curr != NULL)
        {
            c = c+1;
            curr = curr -> next;
        }
    }
    return(c);
}
```

→ if is not required.
We are returning no. of elements.

__/__/__

Delete:

Delete first:

```
void SLL::del_first()
{
    SLL * temp;
    if (first == NULL)
        cout << "List is empty";
    else
    {
        temp = first;        → We are deleting temp.
        first = first → next;
        delete (temp);
    }
}
```

Delete last:

```
void SLL::del_last()
{
    SLL * temp; SLL * prev;
    if (first == NULL)
        cout << "Empty, deletion not possible";
    else
    {
        for (SLL * curr = first ; curr → next != NULL ; curr = curr → next)
                                ① ④ ② ③
            prev = curr;
```

→ We take prev; it is equal to second last element. (We need to put that to NULL).
→ It is not an independent for loop.

```
        prev->next = NULL;
        delete (curr);
    }
}
```

→ we also need to check if only 1 node is present. we add an else if:

```
else if (first -> next == NULL)
{
        temp = first;
        first= first->next;
        delete (temp);
}
```

OR

```
delete (first)
first = NULL;
```

prev : previous to current.

Delete a node having specific value :

```
void SLL::del-specific()
{
    SLL *temp, *prev; int n; int flag=0;
    cout << "Enter node value to be deleted";
    cin >>n;
    if (first == NULL)
        cout<< "Empty";
    else if (first -> data == n)
    {
        temp = first;
        first= first->next;
        delete (temp);
    }   flag=1;
    else
```

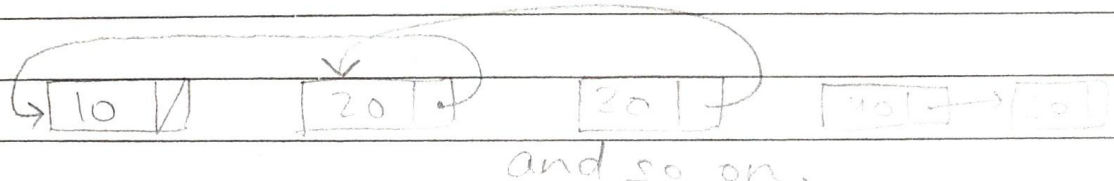
__/__/____

```
{
    SLL *curr = first;
    while (curr != NULL)
    {
        if (curr -> data == x)
        {
            prev -> next = curr -> next;
            delete (curr);
            flag = 1;
            break;
        }
        prev = curr;
        curr = curr -> next;
    }//while.
} //else.
if (flag)
    cout << x << "Found & deleted";
else
    cout << x << "Not found";
{
```

11/10/22.

Reversing SLL:

- We will put it in an array and make a new list.

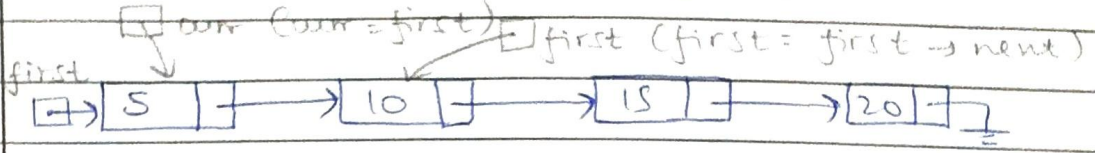


- We put each element at start one by one.

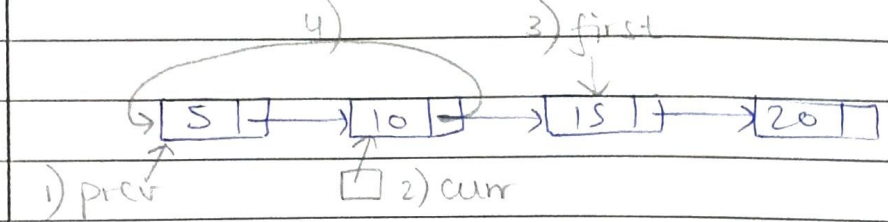
Pseudo code:

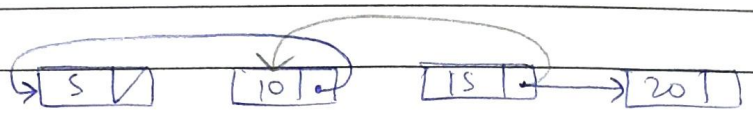
1) prev = curr
2) curr = first
3) first = first → next
4) curr → next = prev

} while (first != NULL)



Initial: prev = NULL.
curr = NULL.





At last : first = curr;

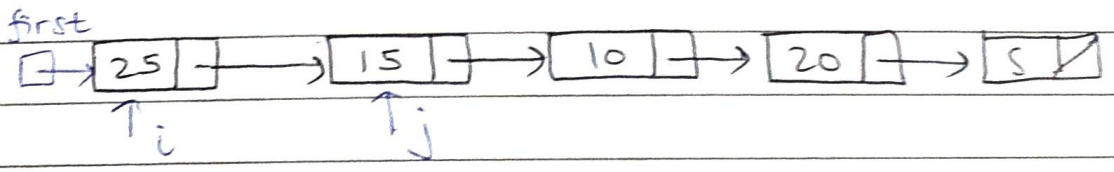
Code :

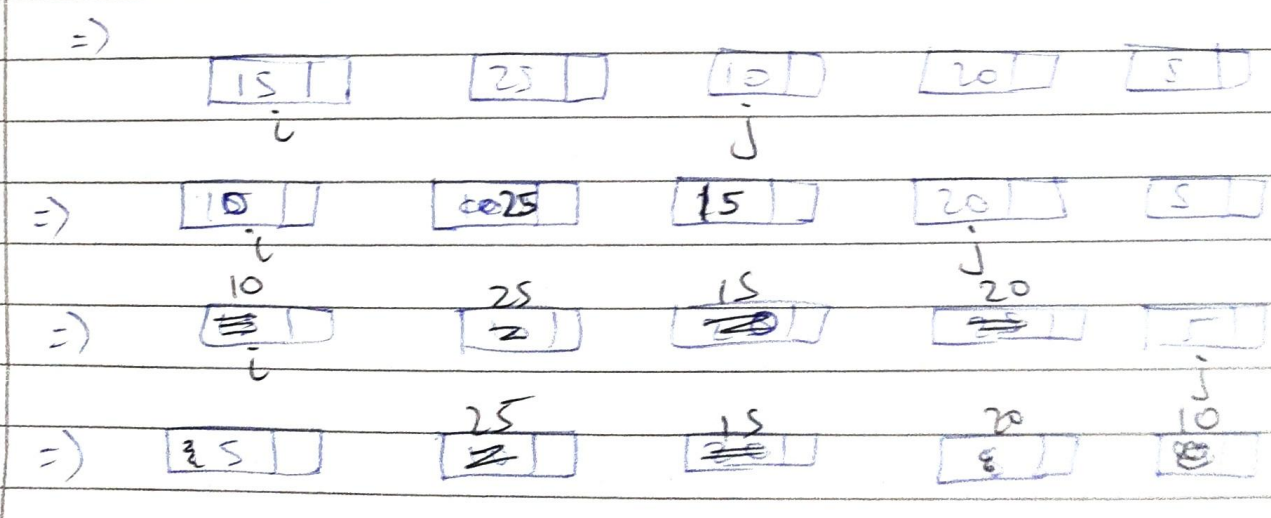
```
void SLL :: reverse()
{
    SLL *prev, *curr; prev = curr = NULL;
    while (first) // while (first != NULL)
    {
        prev = curr;
        curr = first;
        first = first → next;
        curr → next = prev;
    }
    first = curr;
    display();
}
```

__/__/__

Sorting SLL:

```
void SLL::sort() //Ascending order.
{
    SLL *i, *j;
    for (i= first; i->next != NULL;
                   i= i->next)
    {
        for (j= i->next; j != NULL;
                         j= j->next)
        {
            if (i->data > j->data)
            {
                int t= i->data;
                i->data = j->data;
                j->data = t;
            }
        }
    }
}
```





Iterations:

4	n-1
3	n-2
2	n-3
1	n-4

13/10/22.

Main function:

Menu: SLL obj:

1) Insert a node to first list
   obj. end_ins (f1);

2) Insert a node into second list.
   first2 = obj. ins-end (f2);

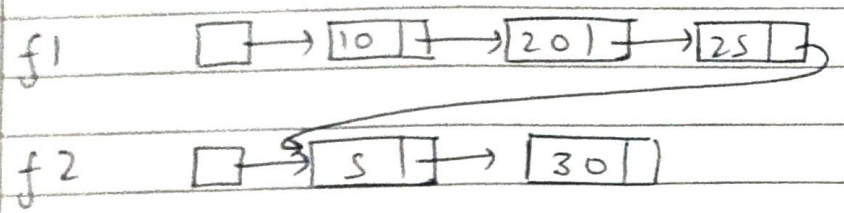
3) Display 1st list obj. display (f1);
4) Display 2nd list obj. display (f2);

5) Concatenate list 1 and 2.
6) Merge list 1 and 2.

Display:

```
void SLL :: display (SLL *first)
{
    SLL *curr = first;
    while (curr) // while (curr != NULL).
    {
        cout << curr -> data;
        curr = curr -> next;
    }
}
```

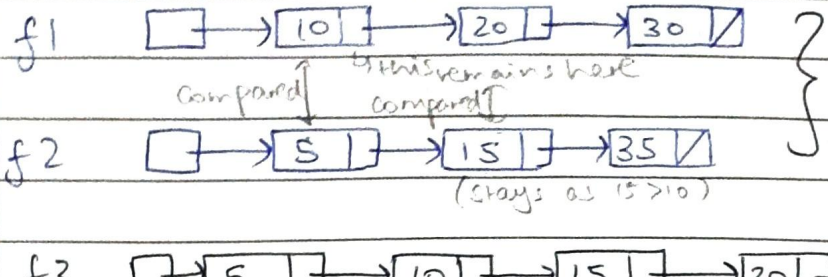
List Concatenation:

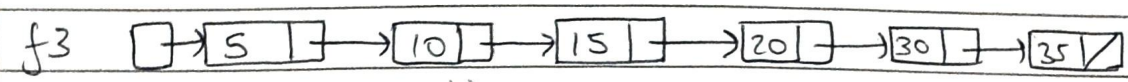



```
SLL * SLL :: concat ( SLL * first1, SLL *
                                    first2)
{
    SLL * first3;
f3 is not a copy. To make duplicate we
need to call insert again.
  if (first1 == NULL)
       first3 = first2;
  else if (first2 == NULL)
       first3 = first1;
  else
  {
      SLL * curr = first1;
      while (curr -> next != NULL) // holds
        the address of last node of first1.
            curr = curr -> next;
      curr -> next = first2;
      first3 = curr;
  }
  return (first3);
}
```

Comparisons:

List Merge: 10>5 15>10 20>15 35>20 35>30





It is in ascending order.

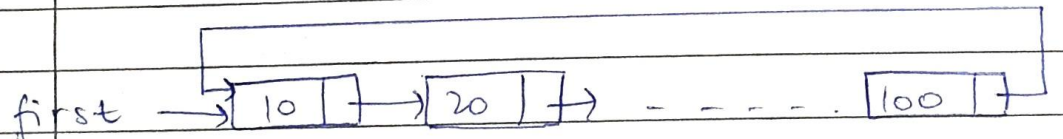
17/10/22.

Code:

```
SLL * merge (SLL *a, SLL *b)
{
    if (!a) // (a == NULL).
        return (b);
    if (!b)
        return (a);
    SLL * res = NULL;
    while (a && b) // while (a != NULL &&
    {                          b != NULL).
        if (a -> data < b -> data)
        {
            res = insert_end (res, a -> data)
            a = a -> next; (We write res as the
        }                   smaller one and insert
                            only that). (Then we check
        else if (a -> data > b -> data)   the next).
        {
            res = insert_end (res, b -> data);
            b = b -> next;
        }
        else // a -> data == b -> data.
```

```
        {
            res = insert_end (res, a->data);
            a = a->next;
            b = b->next;
        }
    }
    while (a)          If there are remaining nodes in a
    {                       if b = NULL then they are added.
        res = insert_end (res, a->data);
        a = a->next;
    } // remaining inden in a are added
      to res list.
    while (b)
    {
        res = insert_end (res, b->data);
        b = b->next;
    }
    display (res);
    return (res);
}
```

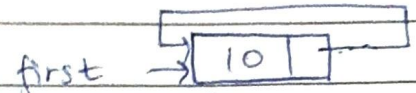
Circular SLL :



Empty : if (first == NULL).

Last node : curr = first;
while (curr->next != first)
curr = curr->next;

// current holds address of last node.



In case of only 1 node
=> Last node condition if applicable.
(since it points at itself).

Code:

```
class csll
{
   int data;
   csll * next;
public:
   csll * insert_end (csll *, int data);
   void display (csll * );
};
csll * first = NULL;
```

Insert end:

```
csll * csll :: insert_end (csll * f, int data)
{
   csll * temp = new csll;
   temp->data = data;
   if (first == NULL)  (Empty case).
   {
      f = temp;
      f->next = f;  (Make it point to itself).
   }
```

```
else
{
    csll * c~~urr~~ = f;
    while ( c->nent != f )
        c = c->nent;
    c->nent = temp;
    temp->nent = f;
} //else.
display (f);
return (f);
}
```

Display:

```
void csll::display(csll *f)
{
    csll *c = f;
    while (c->nent != f ~~NULL~~)
    {
        cout << c->data;
        c = c->nent;
    }
    cout << c->data; // last node value.
}
```

OR

```
do
{
    cout << c->data;
    c = c->nent;
} while (c != f);
```

Linked List Representation of Stack: (LIFO)

ins_end() : push() } end to first
del_end() : pop() } (display)

OR

ins_first() : push() } first to end
del_end() : pop() } (display)

Empty : first == NULL;

Represen^n of Queue : (FIFO)

ins_end() : insert } first to last.
del_first() : del }

OR

ins_first() : insert } last to first.
del_end() : del }

Empty : first == NULL;

→ Circular Q we can represent using circular SLL.

→ It is not required since in LL space is utilized properly and therefore we do not require circular Q.