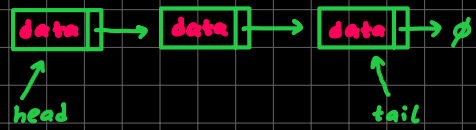
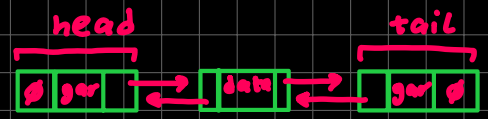
Singly linked list




```
template <class T>
class List {
  private:

    class Node {
      public:

        T data;
        Node* next;

        Node () {

          next= NULL;
        }

        Node (T _d, Node* n= NULL) {

          data = _d;
          next = n;
        }
    };
    Node* head;
    Node* tail;

public:
    List () {
      head = NULL;
      tail = NULL;
    }

    void InsertAthead (T d) {
      Node* temp= new Node (d);

      if (head == NULL)
        tail = temp;
      else
        temp->next= head;
      head = temp;
    }

    void InsertAtEnd (T d) {

      if (head == NULL)
        InsertAthead (d);
      else { // use tail if made , else

        Node * temp = new Node (d);
        tail -> next = temp
        temp -> next = NULL;
      }
    }

    void Print () {
      if (head != NULL) {

        Node * temp= head;
        while (temp != NULL) {
            cout << temp -> data << " ";
            temp= temp->next;
        }
      }
    }

    ~ List () {
      if (head != NULL) {
        Node * temp= head -> next
        Node* c= head;
        while (temp != NULL) {
          delete c;
          c = temp;
          temp= temp -> next;
        }
        delete c;
      }
    }
};
```

while (temp != NULL)
temp= temp->
next;

Doubly list w iterators




```
template <class T>
class list{
    class Node{
      public:
          T data;
          Node* next;
          Node* prev;
      // constructors
    };

    Node * head;
    Node * tail;
    int size;

public:
     list(){
        head = new node();
        tail= new node();
        head->next= tail;
        tail-> prev= head;
     }

     class Iterator{
     friend class list;
       node* curr;

       Iterator(node *p){
             curr =p;
       }
     public:
       Iterator(){
          curr= NULL;
       }

       T & operator *(){
          return curr->data;
       }

       Iterator & operator ++(){
             curr= curr->next;
             return *this;
       }

       Iterator & operator --(){
             curr= curr->prev;
             return *this;
       }
          // post increment , same for post decrement
       Iterator operator ++(int){
             Iterator old= *this;
             curr= curr->next;
             return old;
       }

       bool operator != (Iterator rhs){
            return curr != rhs.curr;
       }

       bool operator == (Iterator rhs){
            return cur== rhs.curr;
       }

       Iterator begin(){
          Iterator it(head->next);
          return it;
       }

       Iterator End(){
          Iterator it(tail);
          return it;
       }
};
```

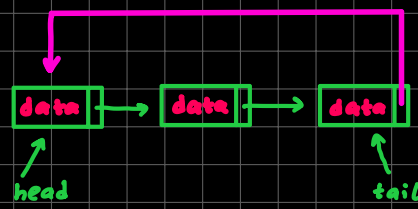
```
void print(){

  Iterator it;

  for(it=begin(); it!=
                  end();
                  it++)
    cout<< it<<endl;
```

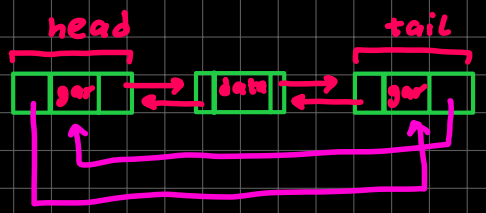
```
Iterator insert(iterator i, T d) {
        node * p= curr;
        node * t= new node(d, p->prev, p);

        p->prev->next = t;
        p->prev= t;

        Iterator it (t);
        return it;
}

Iterator Remove (Iterator i) {
     node * p = curr;
     p->prev->next= p-> next;
     p->next->prev = p->prev;

     Iterator t (p->next);
     delete p;
     return t;
}

void insertAtHead (T d) {
   Iterator it (head-> next); // it= head->next;
   insert(it, d);
}

void insertAtEnd (T d) {
   Iterator it (tail);
   insert(it, d);
}

void RemoveAtHead ()
   Iterator it ( head -> next);
   Remove (it);
}

void RemoveAtEnd () {
   Iterator it (tail);
   Remove (--it);
}

};
```

main

```
list <int>:: Iterator it;
```

· circular list :





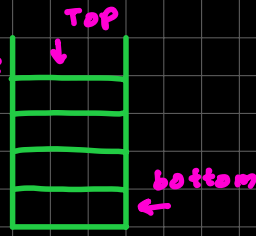
Stacks

- class Stack { List implementation

```
list<T> s;

public:
    void push(T d){
        insertAtHead(d);
    }

    bool pop(T &d){
        if(!s.Empty()){
            d = s.getHead();
            s.removeHead();
            return true;
        }
        else
            return false;
    }
```

- Queue: Check DSA Lab 5
  → Array, List Implementations

LIFO Structure



- template <class T> Array implemention

```
class Stack{

    T * stk;
    int maxSize;
    int stkptr;

public:
    Stack(int s=10){
        stk= new T[10];
        maxSize= s;
        stkptr=0;
    }

    bool push(T d){
        if(!IsFull()){
            stk[stkptr++] = d;
            return true;
        }
        else
            return false;
    }

    bool pop(T &d){
        if(!IsEmpty()){
            d= stk[stkptr--];
            return true;
        }
        else
            return false;
    }
```

```
bool Top(T &d){
    if(!IsEmpty()){
        d= stk[stkptr];
        return true;
    }
    else return false;
}
```

Recursion & BSTs

- Calculating max sum in an array
- Sorting → $n/2$ approach + simple approach
- Printing all possible subsets

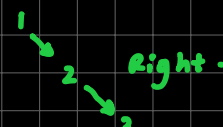
- **Binary Tree:** Simple implementation → no sorted order maintained
- **Binary Search Tree:** Implementation + sorted order maintained

→ Insertion, search, print, delete funcs using recursion / pre-order, post order, in order

- See void remove func → find min on right-subtree
  = find max on left-subtree

- **Binary Tree:** A node having at most 2 children
- **Binary Search Tree:** A binary tree with order maintained → height = $O(\log(n))$
  insertion = //
  searching = //

- worst-case →



Right-skewed

Left-skewed

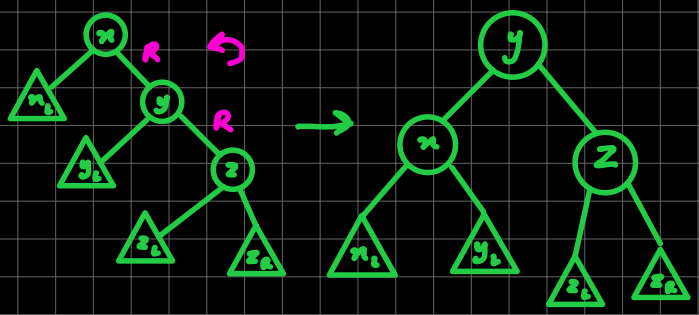
A BST operates like a linked list
∴ every operation needs $O(n)$ time
defies purpose of BST, so we resort to AVL.

Preorder: Root Left Right
Inorder: Left Root Right
Postorder: Left right root

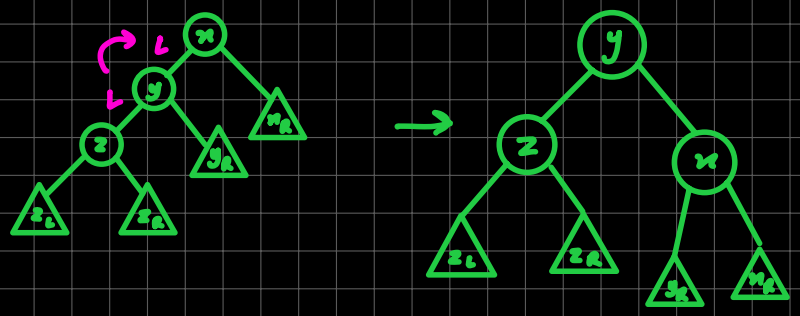
- **AVL / Balanced Search Tree:** Balanced Factor involvement

- BF can be -1, 0, 1
- BF = Height of LST − Height of RST

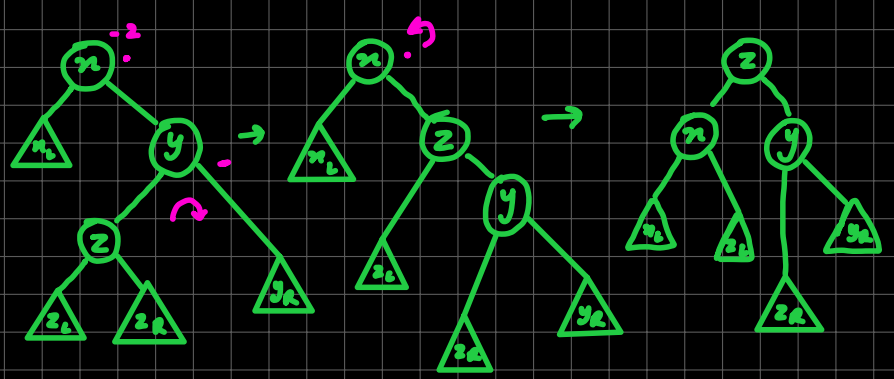
RR Case



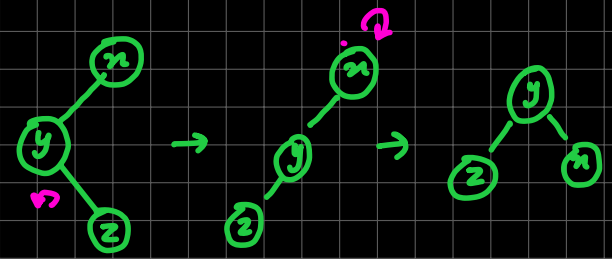
LL Case



RL Case



LR Case

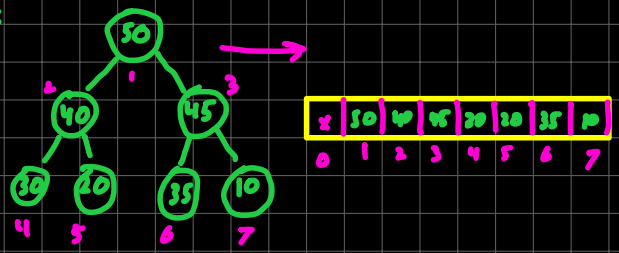


Heaps

- **Heap:** A **complete binary tree** that satisfies **heap property**
- **min-heap:** Root is smaller than its children
- **max-heap:** Root is bigger than its children

- Every node must be filled except for last level, gaps on right

E.g:

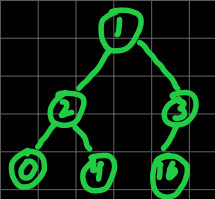


Node = i

parent = $\lfloor \frac{i}{2} \rfloor$

left_child = 2 * i

right child = 2 * i + 1



- Insert bottom left jahan gap ho. Phir according to max, and min heap parent kay saath compare kartay jaana hai, and swap them until root is reached.

- Last node ko deleted elem par lay aao

- Heapify & Heapsort

Graphs

- GPU compatibility checker app.

- **BFS:** Go to immediate neighbours (indirect level order)
→ visited arr = Number of vertices
→ queue → add neighbour elems

```
If (visited [curr] == F)
   print curr
   visited [curr] = T;
   curr node neighbours pushed on queue.
```

- for not connected graph take starting point visited [curr] == F.

- **DFS:** keep going to the 1st neighbour (recursively) Pre-order, Post, In → Depth traversals