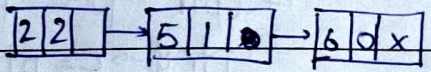
Merging Linked Lists (sorted)

→ (LL_1, LL_2) take one node from LL_2 and search where to put the node in LL_1

Code:

```
temp = head2
while(temp1 != NULL)
    curr = head1
    while(curr != NULL or curr->val > temp->val)
        curr = curr->next
    if (curr = NULL)
        insertatend()
    else
        insert at mid()
```

Polynomials

$2x^2+5x+6$ ⇒ 

→ uses double linked lists

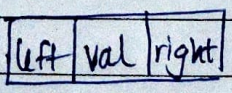
Addition (code)

```
head = tail = null
p1 = head1, p2 = head2
while(p1 != NULL and p2 != NULL)
    if (p1.power > p2.power)
        tail.append(tail, p1.p, p1.coeff)
        p1 = p1->next
    if(p2.p > p1.p)
        //same as earlier
    else:
        coeff = p1.coeff + p2.coeff
        if coeff != 0
            tail.append(tail, p1.p, coeff)
        p1 = p1->next
    p2 = p2->next
if head = NULL
    head = tail.
//reverse both LL
return head;
```

Multiplication (code)

```
ptr1 = poly1, ptr2 = poly2
while(ptr1 != NULL){
    while(ptr2 != NULL){
        int c,p;
        c = p1->coeff + p2->coeff
        p = p1->p + p2->p
        poly3 = insertatend(poly3, c, p)
        p2 = p2->next }
    ptr2 = poly2;
    p1 = p1->next; }
ptr = poly3
while(ptr != NULL and ptr1->next != NULL){
    ptrr = ptr
    while(ptrr->next != NULL){
        if (ptr->p == ptrr->next->p) {
            ptr->c = ptr->c + ptrr->next->c
            dup = ptrr->next
            ptr->next = ptr->next->next}
        else {ptrr = ptrr->next}}
    ptr = ptr->next;}
```

Double Linked Lists



→ we can move forward and backwards as well. (dual movement)

→ more efficient for insertion, deletion and traversal.

```
struct node{
    int val;
    node* left;
    node* right; }
```

__/__/__

```
insert_beg()
if(head != NULL)
    head->prev = nn
    head = nn
```

```
insertend()
if(head = NULL){
    head = nn
    nn->prev = NULL}
temp = head
while(temp->next != NULL)
    temp = temp->next
temp->next = nn
nn->prev = temp
```

```
delete_beg()
head = head->next
head->prev = NULL
```

```
delete_end()
temp = head
while(temp->next != NULL)
    temp = temp->next
t2 = temp->prev
t2->next = NULL
```

```
insertMid(N)
temp = head
while(N--)
    temp = temp->next
temp2 = temp->next
nn->next = temp->next
nn->prev = temp
temp->next = nn
temp2->prev = nn
```

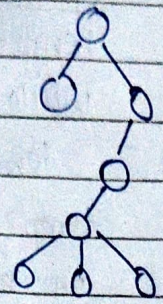
Note
for deletion, please add head = NULL case!

Also add only one node case

```
deletemid(N)
temp = head
while(N--)
    temp = temp->next
p = temp->prev
n = temp->next
p->next = t->next
n->prev = t->prev
```

→ in SLL only update next, here we update 2 ptrs.
→ deleting node here is more efficient than SLL (here no need to traverse to find prev).

Trees



Defn

Trees are finite set of elements called nodes such that

① There is a special designated node called root.

② The remaining nodes are partitioned into $n \geq 0$ number of disjoint subsets. T_i, which is also a tree.

Binary Tree

Defn: is a finite set of elements called nodes such that

① T is empty, or

② special root, which has rest of the trees having attachment to only 2 other trees.

Max nodes in a level $i = 2^i$ if $i \geq 0$

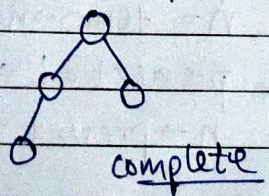
$= 2^{i-1}$ if $i \geq 1$

Full binary tree

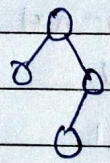
↳ if all the levels contain max possible nodes

Complete binary tree

↳ if all the levels except the last contain max possible nodes and in the last level, the nodes are as left as possible.



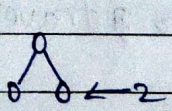
complete



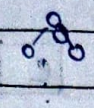
not complete

Prove max no of nodes at level i is 2^i.

Ans $i=0$ $2^0 = 1$ ⊙ ← 1

$i=1$ $2^1 = 2$  ← 2

$i=k \Rightarrow i=k+1$



$k \Rightarrow 2^k$ nodes

k^{th} level has 2^k nodes

max each node in k can have 2 children

So, nodes in level $(k+1)$

$= 2 \times 2^k = 2^{k+1}$

Hence proved

Total nodes upto depth i is $2^{i+1}-1$.

Ans At level i, we have 2^i nodes.

hence till level i, total nodes -

$= 1+2^1+2^2+\cdots+2^i$

$= \frac{1(2^{i+1}-1)}{2-1} = 2^{i+1}-1$

Hence proved

Height of binary Tree $= \lceil \log_2 n+1 \rceil - 1$

Ans $n \leq 2^{i+1}-1 \Rightarrow n+1 \leq 2^{i+1} \Rightarrow i \geq \lceil \log_2 n+1 \rceil - 1$

Total nodes = no of edges + 1

$n_0 = n_2 + 1$ [$n_i \rightarrow$ no of nodes with degree i]

Ans Let total nodes be n

So, $n = n_0 + n_1 + n_2$

$e = 0 \times n_0 + 1 \times n_1 + 2 \times n_2$

$= n_1 + 2n_2$

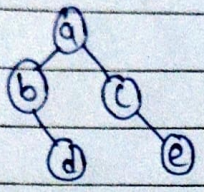
Now, $n = e+1$

$n_1 + 2n_2 + 1 = n_0 + n_1 + n_2$

$n_0 = n_2 + 1$

hence proved

Representing Binary Tree



a	b	c		d		e		
1	2	3	4	5	6	7	8	

Parent of node i $\Rightarrow \lfloor i/2 \rfloor$

Left Child of node i $\Rightarrow 2i$

Right " " " " $\Rightarrow 2i+1$

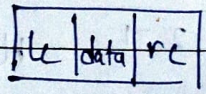
Problems

① Wastage of a lot of space / additional gaps

② static memory so can't really modify size.

double linked list approach

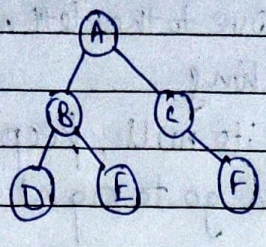
```
struct Node{
    int data;
    Node* lc;
    Node* rc;};
```



Level order insertion

```
Node* insert (root, item){
    nn = createNode(item)
    if root =NULL
        root = nn
    else if q→fr→lc = NULL
        q→fr→lc = nn
    elseif q→fr→rc = NULL
        q→fr→rc = nn
        pop(q)
    push(nn, q)
    return root }
```

Tree Traversal



Preorder : Root/left/right
Inorder : left/root/right
Postorder : left/right/root

↳ Preorder : A → B → D → E → C → F

Inorder : D → B → E → A → C → F

Postorder : D → E → B → F → C → A

Using Recursion

left : func(root → lc)

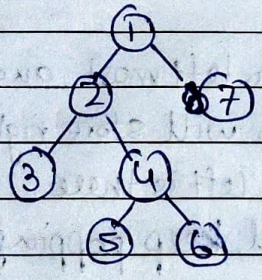
right : func(root → rc)

root : print

write in the order above, to achieve recursive solution.

Iterative methods

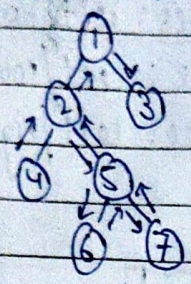
① Preorder (Ro/L/R)



→ push root to stack
→ pop and print
→ push right & left (in order)
→ pop left and push left and right of left
→ go till empty stack

```
preorder(root){
    if root = NULL return;
    stk.push(root);
    while(!isempty(stk)){
        root = top(stk);
        pop(stk);
        print(root→val);
        if(root→rc != NULL)
            stk.push(root→rc)
        if(root→lc != NULL)
            stk.push(root→lc)
    }
```

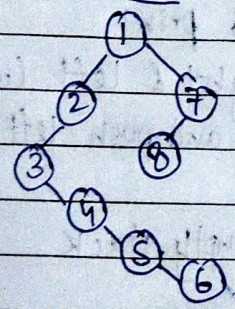
② Inorder (L/Ro/R)



→ move to the left most, and keep on pushing.
→ till its null, keep popping
→ then go to right and keep doing it.

```
inorder(root) {
  while(1){
    if (root != NULL) {
      stk.push(root)
      root = root→lc }
    else{
      if (isempty(stk)) break;
      root = st.top()
      pop(stk)
      print(root→val)
      root = root→rc; }}
```

③ Postorder(root) [L/R/Ro]



→ go to left most., and keep pushing
→ temp will store right & into stk if no left is there.
→ if null, keep popping

```
Code (curr = root)
while (curr != NULL || !stk.empty()){
  if (curr != NULL)
    stk.push(curr)
    curr = curr→left
  else temp = st.top()→rc
    if (temp != NULL)
      temp = stk.top()
      st.pop()
      print
      while(!st.empty() && temp ==
       stk.top()→rc) {
        temp = st.top()
        print(temp→val)
      else
        curr = temp.
```

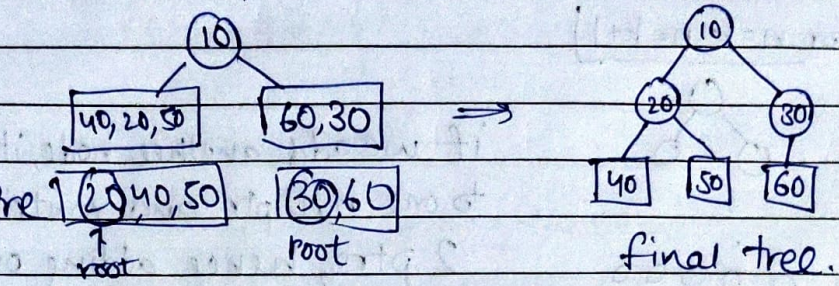
Construct Binary Tree from inorder/preorder

Ex Inorder: 40 → 20 → 50 → 10 → 60 → 30

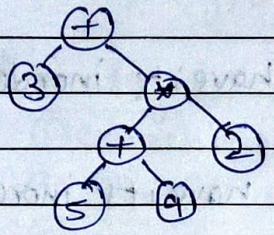
Preorder: 10 → 20 → 40 → 50 → 30 → 60

Ans from preorder, 10 is root [as its Ro/L/R]

from inorder,



Expression Tree



⇒ 3+((9+5)*2)

Preorder ⇒ Prefix

Inorder ⇒ Infix

Postorder ⇒ Postfix

How to construct?

↳ if operand, push to stack

↳ if operator, pop 2 and make them child and push curr node.

Code

```
if(s.isoperator()){
    z=new(s);
    x=pop();
    y=pop();
    z->lc=y; z->rc=x;
    push(z);}
else{z=new(s)
    push(z);}
```

Threaded Binary Tree

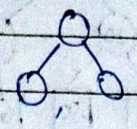
For a btree with n nodes, there are (n+1) NULL pointers.

Proof for n=1 | x | v | x | ⇒ 2 NULL ptr

for n=2 | x | v | → 3 NULL ptr

| x | v | x |

for n=k, n=k+1



if we add another node, it'll connect to one nullptr and contribute to its 2 ptrs, hence adding one nullptr

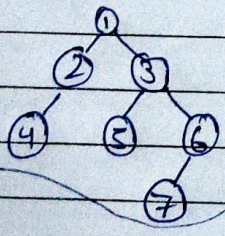
k nodes ⇒ k+1 NULL ptrs) k+1 nodes ⇒ k+2 NULL ptrs

- If node→rc = NULL, it will have its inorder succ-essor
- If node→lc = NULL, it will have its inorder pred-essor

| lt | lc | val | rc | rt |

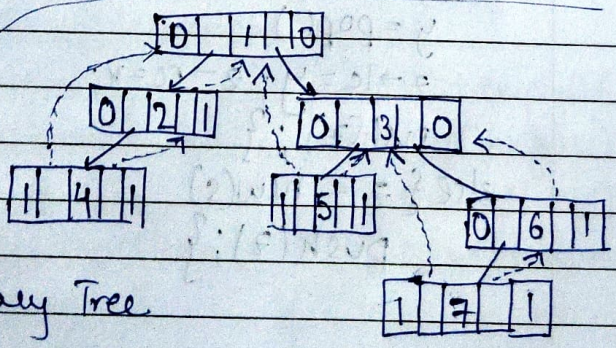
if lt = 0, lc points to child
" lt = 1, " " predecessor
" rL = 0, rc " " child
" rt = 0, rc " " psuccesor.

Eg



Inorder: 4215736
make threaded binary.

Ans



Threaded Binary Tree.

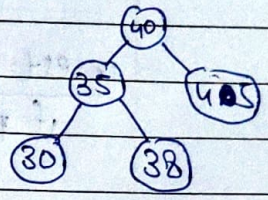
How to find inorder successor?

```
succ(ptr) {
    suc = ptr->rc;
    if(ptr->rbit = 0) then
        while(suc->lbit = 0)
            suc = suc->lc
    return suc; }
```

Traversed by Morris Traversal basically inorder but if thread exists, uses the rc to move forward.

Binary Search Tree

↳ ~~for any node having value a~~
- ~~left subtree value < a~~
- ~~right " " > a~~



Advantages:

1. Searching Time is less
2. Time complexity to search is O(log n)
3. When worst case, it is a straight line called a skew BST ⟹ O(n) Time complexity.

Searching (recursive)

```
search(root, item){
    if root = NULL
        print(NOT FOUND)
    else
        if(root->data = item)
            print(FOUND)
        else
            if(item < root->data)
                search(root->lc, item)
            else
                search(root->rc, item)
```

insert (root, item) [recursive]

```
if (root == NULL)
    root = nn
else{
    if (item < root->data)
root->lc = insert (tree->lc, item)
    else if (item > root->data)
    root->rc = insert (tree->rc, item)
    else
        print (item is alr in tree)
```

insert (root, item) [iterative]

```
if root = NULL
    root = nn
parent = NULL, curr = root
while (curr != NULL)
    parent = curr
    if (curr->val > x)
        curr = curr->lc
    else if (curr->val < x)
        curr = curr->rc
    else
        print (alr there)
if (parent->val > x)
    parent->lc = nn
else
    parent->rc = nn
```

Note
Duplicate Values are not allowed.

Note
Searching iteratively is similar to this. just instead of ~~see~~ inserting, print "Found"

Insertion in BST is always at a leaf node, then other cases are seen

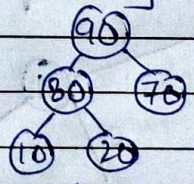
Deletion

```
if root = NULL
    ↳ print(naw)
if (root → key > x)
    root → lc = del(root → lc, x)
else if (root → key < x)
    root → lc = del(root → rc, x)
else { if (root → Lc = NULL) }
        temp = root → rc
        free(root)
        return temp }
    if (root → rc = NULL) {
        temp = root → lc
        free(root)
        return temp }
```

```
succ = inSucc(root)
root → key = succ → key
root → rc = delNode(root →
                rc, succ → key)
return root;
```

Heaps

↳ root will be greater than both children (maxHeap)
↳ additionally it will be a complete binary tree.



as an array

↳ parent = i/2
↳ lc index = 2i
↳ rc index = 2i+1

90	80	70	10	20

Insertion

~~n = n+1~~
~~a[n-1] = key~~
~~heapify(a, n, n-1)~~

Insertion

```
n = n+1
for i = n; i > 1; i = i/2
    if item < heap[i/2] break
    else heap[i] = heap[i/2]
    exit if heap[i] = item
```

for delete, only delete root node, else not allowed

① HeapSort
- in descending order, elements are deleted (max Heap)
- " ascending ", elements " " (min Heap)

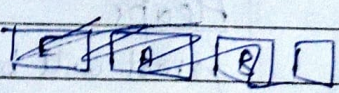
② PriQueues can also be written using Heaps

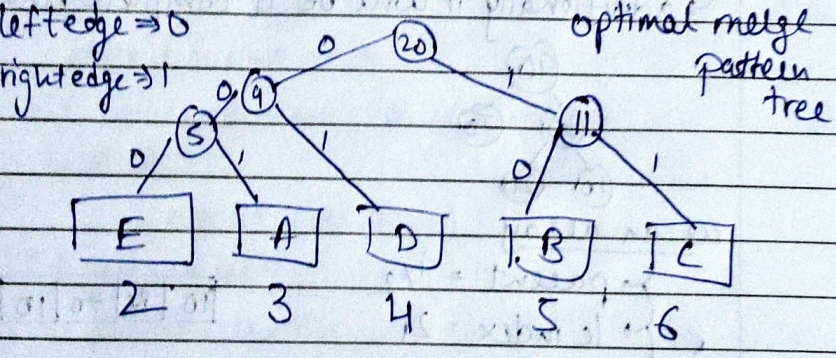
Huffman's Algorithm/Coding
↳ compression technique for message.

Message: BCCABBDDAECCBBAEDDCC
↳ length = 20

char	freq	code
A	3	000
B	5	001
C	6	010
D	4	011
E	2	100

55 bits





Keep combining least two nodes.

Code ⇒ path from root to that var.

char	code
A	001
B	10
C	11
D	01
E	000

45 bits

& shortened msg

AVL Tree

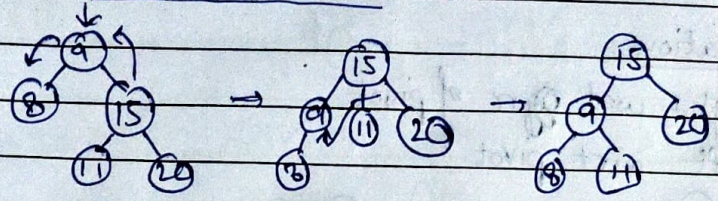
↳ self balancing binary tree (BST), and diff b/w heights of left & right subtree can't be more than one

↳ Balance Factor = height of left subtree − height of right subtree

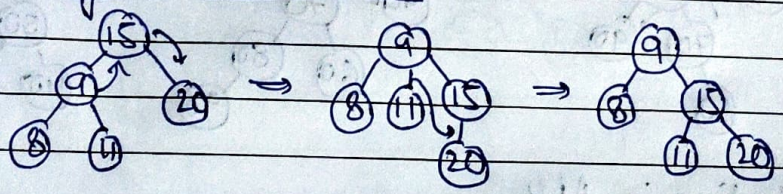
↳ |Balance factor| ≤ 1

Rotates

(1) Left rotate wrt node

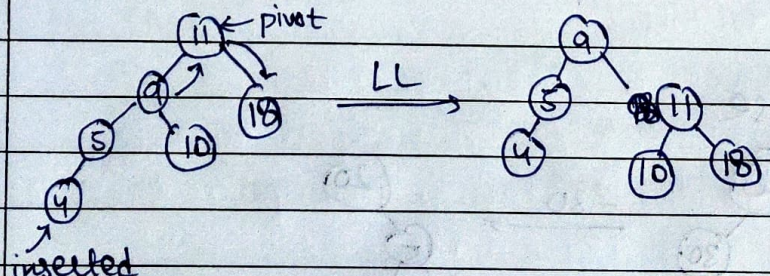


(2) Right rotate wrt node

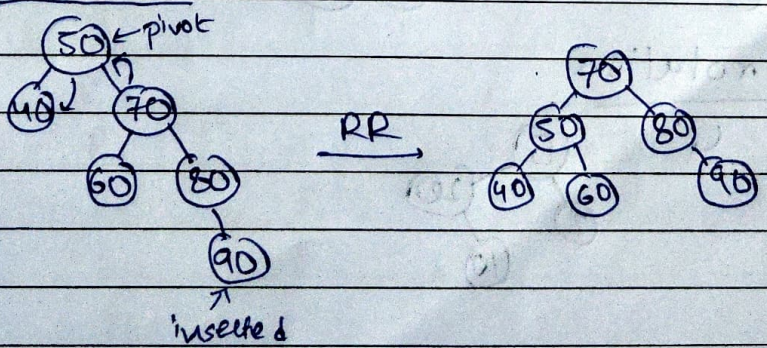


(3) LL rotation

Pivot: inserted node, nearest ancestor, which is unbalanced

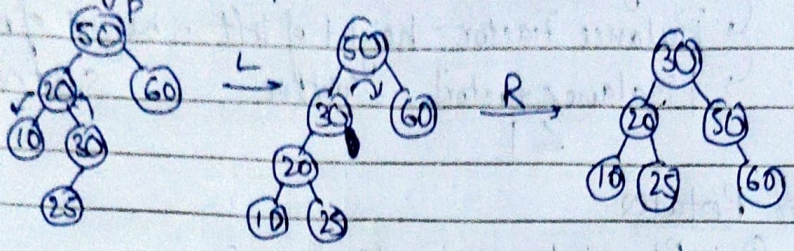


(4) RR rotation



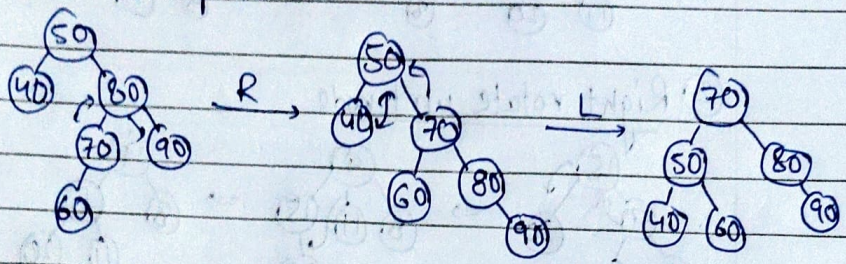
⑤ LR rotation

1) Left w.r.t LC of pivot
2) right w.r.t pivot



⑥ RL rotation

1) right wrt rc of pivot
2) left wrt pivot

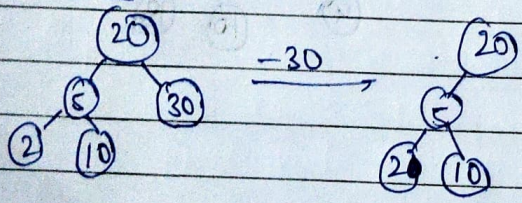


Deletion in AVL

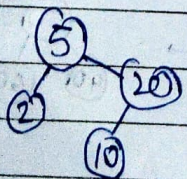
① delete as per BST
② check balanced
③ if not balanced, see case and rotate.

Cases

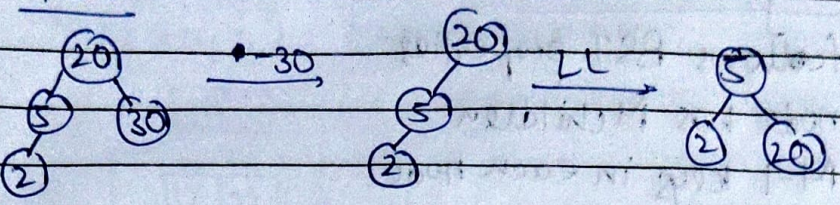
#1) R_0 case



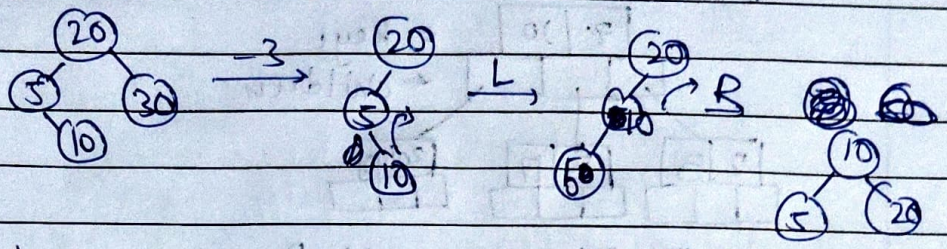
LL rotation



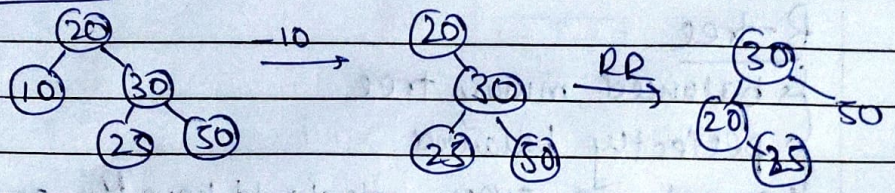
#2 R_1 case



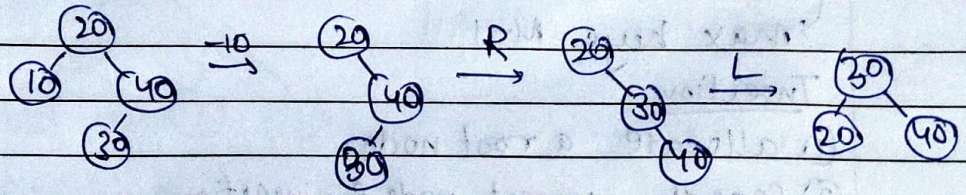
#3 R_{-1} case



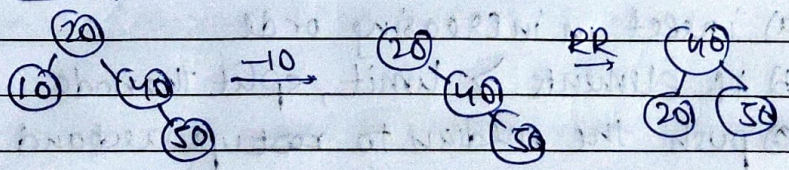
#4 L_0 case



#5 L_{+1} case



#6 L_{-1} case

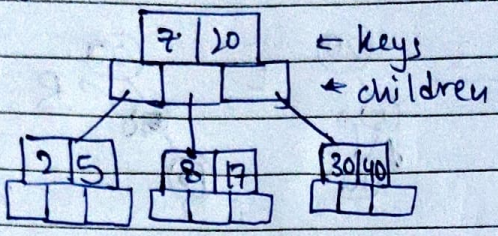


Case	Rotation
R_0	LL
R_{-1}	LR
R_1	LL
L_0	RR
L_{-1}	RL
R_1	RR

Do these

Mway Search Tree

⇒ follows BST properties
⇒ node has M children
⇒ M-1 keys in each node
⇒ keys in ascending order.



vals < 7 7 < vals < 20 vals > 20

B-tree

- balanced mway tree
- perfectly balanced
- except root, every node should have $M/2$ or $M/2 + k$ children
- max children = M
- max keys = M-1

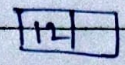
Insertion

1. allocate a root node.
2. search correct node for insertion
3. if node full:
   a) insert in increasing order
   b) if elements > limit, split the node
   c) push the median to ~~root~~ upwards and left is lc and right is rc.
   d) if upper node not full, insert the node
   e) " " " is full, repeat step 3.

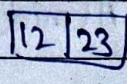
Q Create BTree (M=3) vals- 12, 23, 6, 8, 15, 19, 45, 1

Ans Max key = 2 Max children = 2

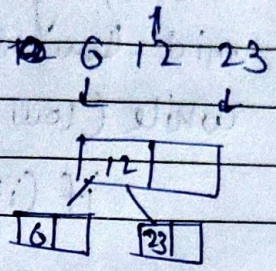
S1



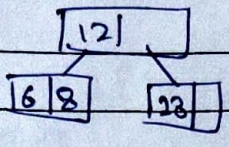
S2



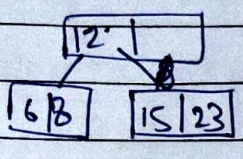
S3



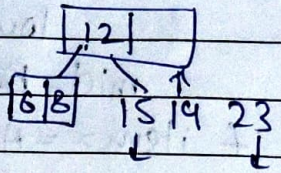
S4



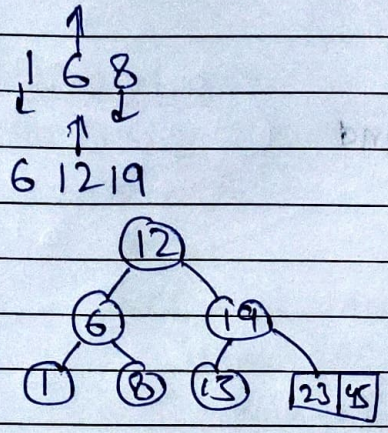
S5

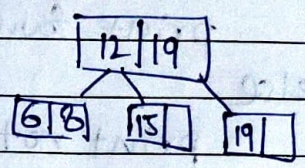


S6



S7





Searching

(1) Linear Search (A, item)

```
i = 1
while (i ≤ n and A[i] != item)
        i = i+1
if (i ≤ n)
    print found
else not found
```

Best case = O(1) — first element

Worst case = O(n) — not in array

Avg case = $\sum P_k k = \frac{1}{n} \sum k = \frac{n+1}{2}$

② Binary search(A, l, h, item) — [Use sorted array]

```
low=l, high=h
mid = (low+high)/2
while (low ≤ end and item = A[mid])
        if (item < A[mid])
            high = mid-1
        else
            low = mid+1
        mid = (low+high)/2
    if item == A[mid]
        print found
    else
        print not found
```

Best case: O(1)
Worst case: O(log n)

Sorting

① Bubble Sort

```
for(i=1; i<n; i++)
    for(j=1; j ≤ n-i; j++)
        if(A[j] > A[j+1])
            swap(A[j], A[j+1])
```

Time complexity
$= (n-1) + (n-2) + \dots + 2 + 1 = \frac{n(n-1)}{2}$ $O(n^2)$

You can modify it by doing this, if sorted in pass, continue (modified form)

② Insertion Sort

```
for(j=2 ; j≤n; j++)
    key=A[j]
    i=j-1
    while (i>0 and A[i]>key)
        A[i+1]=A[i]
        i=i-1
    A[i+1]=key
```

$T(n) = 1+2+3+\dots+n-1 = \frac{n(n-1)}{2} \Rightarrow O(n^2)$

Best case (alr sorted).

$2, 3, \dots, n \Rightarrow O(n)$

③ Selection Sort

```
for (i=1; i≤n-1; i++)
    min=A[i]
    loc=i
    for (j=i+1; j≤n; j++)
        if(min>A[j])
            min=A[j]
            loc=j
    if (loc!=i)
        A[loc]=A[i]
        A[i]=min
```

$T(n) = (n-1)+(n-2)+\dots+2+1 = \frac{n(n-1)}{2} \Rightarrow O(n^2)$

④ Merge Sort

↳ uses divide and conquer method

↳ use recursion.

```
MergeSort(l,h)
  if(l<h){
    mid=(l+h)/2
    mergeSort(l,mid)
    mergeSort(mid+1,high)
     Merge(l,mid,h);  }
Merge(l,m,h)
  k=low i=low j=mid+1
while(k≤mid & j≤high)
   if A(k) < A(j){
        B[i]=A[k]
        k=k+1}
    else
        B[i]=A[j]
        j=j+1
    i=i+1
if (k>mid)
   for(k=j to high)
       B[i]=A[j];j++;i++;
else.
    for(k=k to mid)
      B[i]=A[k];k++;i++;
for(p=low to high)
    A[p]=B[p]
```

← divide

$T(n) = 2T(\frac{n}{2}) + n$ ← merge func

$= 2\left[2T(\frac{n}{4}) + n\right] + n$

$= 2^2 T(\frac{n}{2^2}) + 2n = 2^k T(\frac{n}{2^k}) + kn$

$= cn + n\log_2 n$ $\quad O(n\log_2 n)$

⑤ Quick Sort

```
Partition(A,q,r)
  x = A[q]
  i = q-1
  for(j=q+1 to r)
    if A[j] <= x
      i+=1
      swap(A[i],A[j])
```

```
QSort(A,p,r)
  if (p<r)
    q = Partition(A,p,r)
    QSort(A,p,q-1)
    QSort(A,q+1,r)
```

$T(n) = O(n^2)$

Best case

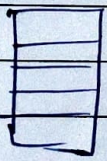
$T(n) = 2T\left(\frac{n}{2}\right) + cn$

$\Rightarrow n \log_2 n$

Hashing

↳ searching Technique

↳ searching → O(1) and not dependent on n



hash table → has address of the datas

hash function ⇒ returns the address

$H : K \rightarrow L$

K: key, L: address

Properties

① hash function computation → easy

② returned address should be uniformly distributed

③ collision should be less

Division Method

$HF \equiv K \pmod m$

↑ prime no.

$K = 1234 \Rightarrow 1234 \equiv 70 \pmod{97}$

↑ address

70	1234

Mid square method

$HF = k^2$

$k = 1234 \Rightarrow HF = 1522756$ select from left

22	1234

Folding method

↳ Sum of digits (halved)

$k = 1234$

↳ $12 + 34 = 46$

46	1234

Collision Avoidance

① Linear probing

mod 7 16, 40, 27, 9, 75

2 5 6 2 5.

0	75
1	
2	16
3	9
4	
5	40
6	75

} fill next if empty

for quadratic probing instead of $k+1$ check k^2+1

② Double Hashing

↳ use two hashing funcs.

③ Chaining

mod 6 24, 75, 81, 42, 63, 65
0 3 3 0 3 5.

0 24 → 42
1
2
3 75 → 81 → 63
4
5 65

} store as LL at the HF fine values.