Introduction to trees

Introduction to trees

- Trees and terminology
- Tree traversals
- Binary trees
- Complete binary trees / Almost complete binary tree (ACBT)
- Array implementation of ACBT
- Binary search trees
- AVL tree
- Multi-way tree
- Brief introduction and uses cases of B-Tree /B+Tree.

Non Linear Data Structures

- un order --- no fixed sequence.
- have one-to-many or many-to-many relationship
- e.g. Tree, graph

Tree

finite set of nodes with one specially designated node called the root.

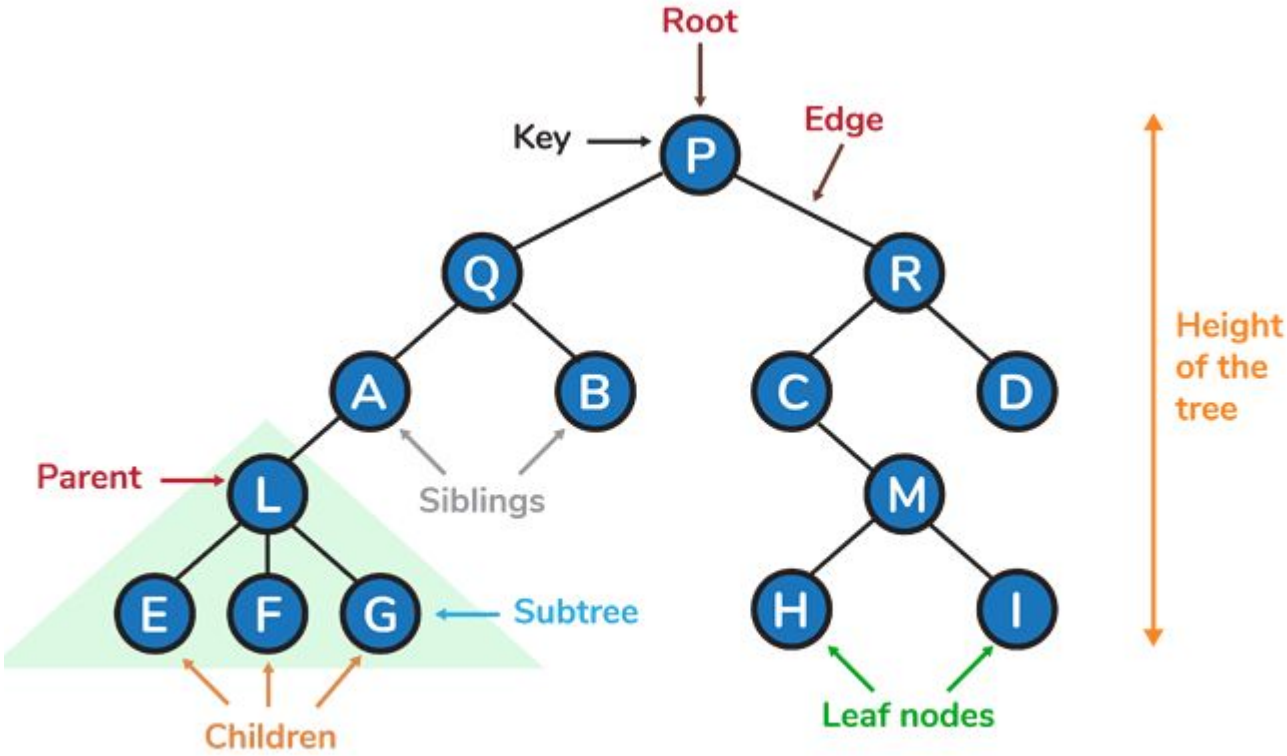
root - Starting point of Tree - does not have parent

remaining nodes are partitioned into n>=0 disjoint sets T1, T2, ...,Tn called subtrees of the root.

Tree Terminology

1. Tree
2. Null tree
3. Node
4. Parent node
5. Leaf node
6. Siblings
7. Degree of a node
8. Degree of a tree

9. Descendents
10. Ancestors
11. Level of a node
12. Height or depth of a tree
13. Forest


Root
Key
P
Edge
Q
R
A
B
C
D
Parent
L
Siblings
M
E
F
G
Subtree
H
I
Height of the tree
Children
Leaf nodes

Types of Tree

1. Binary Tree
2. M way Tree
3. B Tree
4. B+ Tree

Types of Binary Tree

Strictly binary tree

- Either 0 or 2 child

Full binary tree -

- All levels must be full total elt = 2^level-1

Complete binary tree -

- If n levels then n-1 levels must be full and last level partially full from left to right

Skewed binary tree -

- A binary tree in which every node has only one child,

Heap tree

Complete binary tree parent node contain either greater value than its children or smaller value than its children.

Two types of heap trees -

Max heap tree : every parent will have greater value than its children.

Therefore

root node of max heap tree is the largest value among all the values.

Min heap tree : every parent will have smaller value than its children.

Therefore

root node of min heap tree is the smallest value among all the values.

Binary search tree

- binary tree
- Child nodes are placed based on data comparison
- Left child has less value than its parent and right child will have greater or equal to value than parent
- If we insert the data in the BST in ascending order then BST becomes right skewed
- if we insert the data in descending order then BST becomes left skewed.

Expression Tree

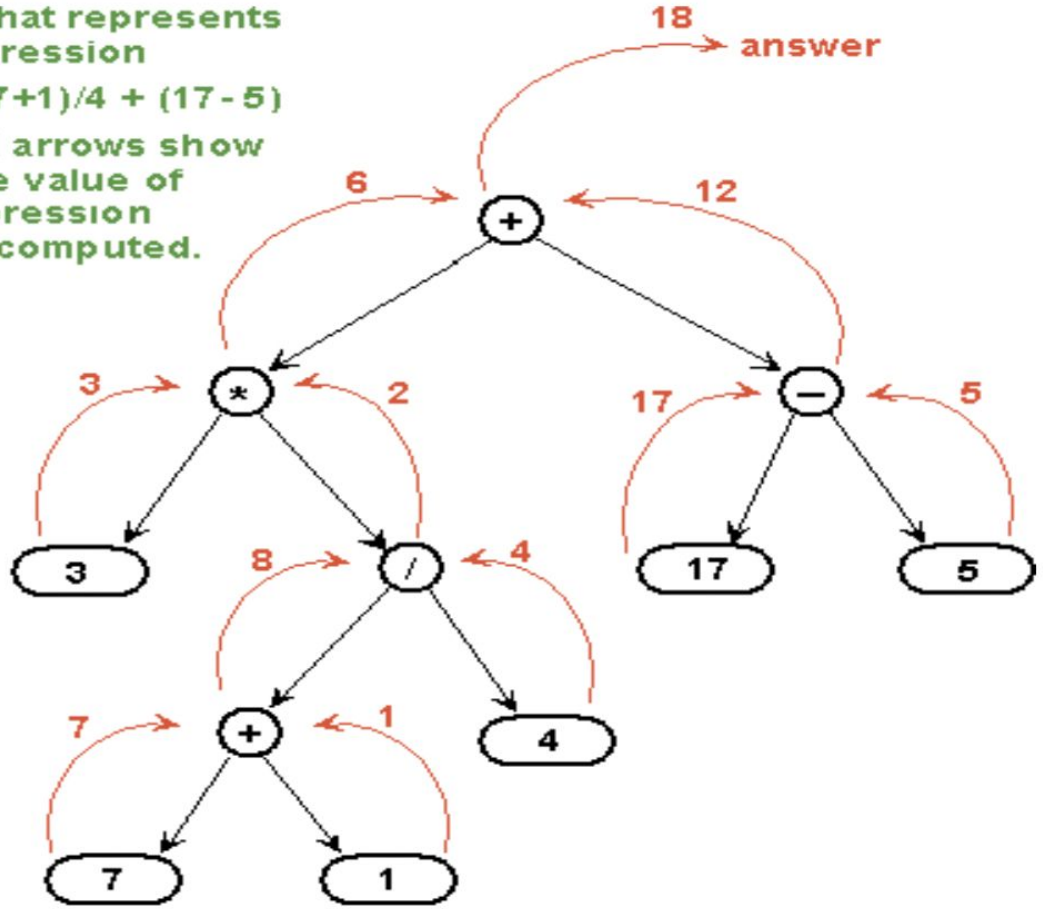
- When an expression is represented through a tree, it is known as expression tree.
- The leaves of an expression tree are operands, such as constant variable names and all internal nodes contain operators.
- Preorder traversal of an expression tree gives prefix equivalent of the expression.
- Postorder traversal of an expression gives the postfix equivalent of the expression.

Expression Tree

A tree that represents the expression

3 * (7+1)/4 + (17 - 5)

The red arrows show how the value of the expression can be computed.



Expression tree -

- Binary tree
- Used to represent mathematical expression of operator
- All leaf node are operands and all internal nodes are operator

AVL Tree -

- AVL tree is a self-balancing binary search tree
- Height balanced Binary search tree.
- balance factor is maintained for every node

- Balance factor has values either -1, 0 or +1.
- Balance Factor = (Height of Left Subtree - Height of Right Subtree) or (Height of Right Subtree - Height of Left Subtree)
- Search efficient tree less comparisons will be needed
- While constructing AVL tree need to maintain balance factor for every node
- If bf goes out or has other than the fixed values from range need to perform some operations to get balance factor from the range

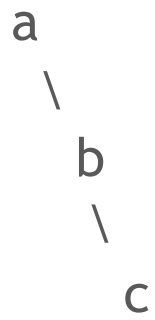
2 types of Operations

1. Single Rotation
    - LL Rotation
    - RR Rotation
2. Double Rotation
    - LR Rotation
    - RL rotation

1. LL/RR Rotation
    a. By adding node at right/left in tree node in tree gets unbalanced so need to apply LL rotation which makes tree balance
2. LR/RL Rotation
    a. In this type of rotation we need to apply 2 operations at one place
    b. LL then RR rotation // RR then LL rotation

The AVL Tree Rotations **Left Rotation (LL)**

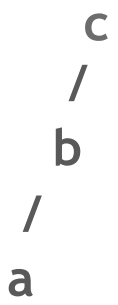
Imagine we have this situation:



- To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:
  - b becomes the new root.
  - a takes ownership of b's left child as its right child, or in this case, null.
  - b takes ownership of a as its left child.
- The tree now looks like this:

The AVL Tree Rotations **Right Rotation (RR)**

- A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:



- To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:
  - b becomes the new root.
  - c takes ownership of b's right child, as its left child. In this case, that value is null.
  - b takes ownership of c, as it's right child.

- The resulting tree:

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

Sometimes a single left rotation is not sufficient to balance an unbalanced tree. Take this situation:

```
  a
   \
    c
```

- It's balanced. Let's insert 'b'.

```
  a
   \
    c
   /
  b
```

```
    c
   /
  a
   \
    b
```

- Our initial reaction here is to do a single left rotation. Let's try that

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- This is a result of the right subtree having a negative balance.
- Because the right subtree was left heavy, our rotation was not sufficient.
- The answer is to perform a right rotation on the right subtree. We are not rotating on our current root. We are rotating on our right child.

- Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

The AVL Tree Rotations **Left-Right Rotation (LR) or "Double left“**

- After performing a rotation on our right subtree, we have prepared our root to be rotated left. Here is our tree now:

```
    a
     \
      b
       \
        c
```

left rotation. b

```
      / \
        a   c
```

Right-Left Rotiation (RL) or "Double right"

- A double right rotation, or right-left rotation, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy.

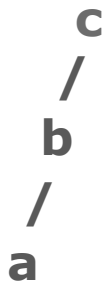
- This is a mirror operation of what was illustrated in the section on Left-Right Rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.

- The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem.

Right-Left Rotiation (RL) or "Double right"

- The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem .

- The answer is to make our left subtree left-heavy. We do this by performing a left rotation our left subtree. Doing so leaves us with this situation:



- This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C. The result:

Types Representation

1. By Array
2. By Linked List

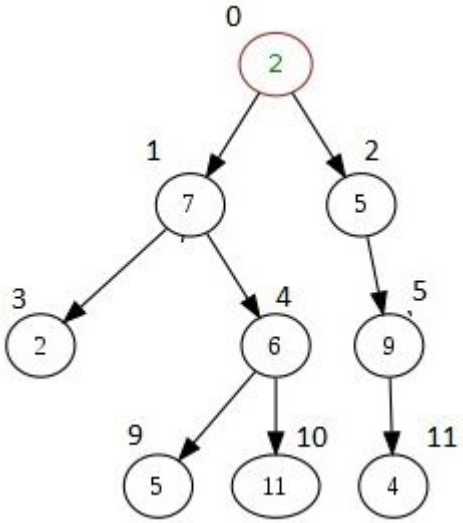
Representation of Complete binary trees / Almost complete binary tree (ACBT) in array

No. of elements in tree - depends on level of tree

Total no. of elts in tree = 2^(level+1) - 1

If level = 2 no. of elt = 2^3-1 ⇒ 7

Level = 3 n= 2^4-1 ⇒ 15



2	7	5	2	6	9				5	11	4			

level = 3 n= 2^4-1 ==15

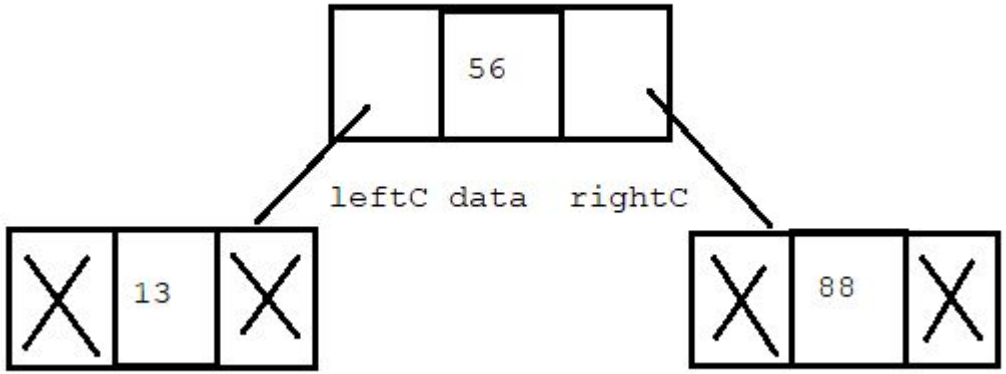
1. By Linked List

Every node will be storing value , links to it’s childs

Structure of node -

Operations on tree

1. Insert
2. Delete
3. Traversal
4. Search
5. …..



Tree Traversals

2 types of traversals

1. Breadth first Traversal
2. Depth first Traversal
    a. Inorder Traversal
    b. Preorder Traversal
    c. Postorder Traversal

Binary search tree

It is a binary tree that is either empty or in which each node contains a key that satisfies the conditions:

1. All keys (if any) in the left sub tree of the root precede the key in the root.

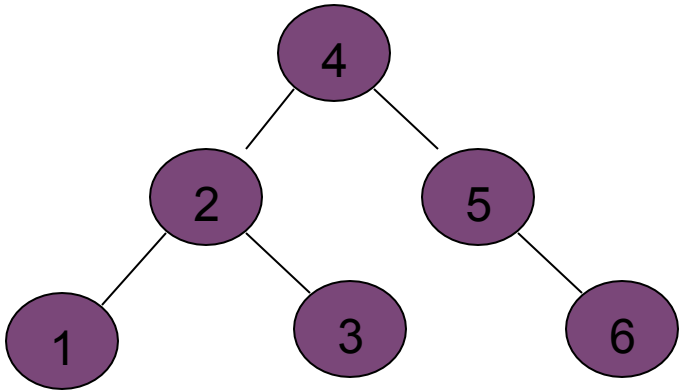
2. The key in the root precedes all keys (if any) in the right sub tree.

3. The left and right sub trees of the root are again search trees.

Example Binary search Tree

Put following numbers in a linked list into a binary search tree

4,2,5,1,3,6



Exercise

Create a binary search tree for the following sequence of numbers

10, 30, 50, 5, 40, 7, 3, 45, 20

Traverse the created in

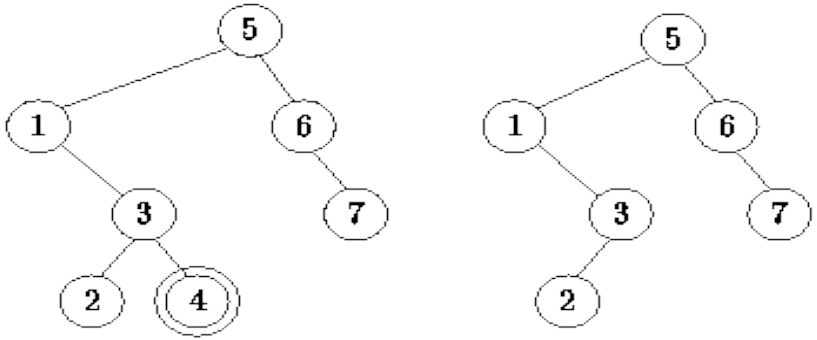
1. Preorder
2. Inorder
3. Postorder

Removing Items from Binary Search Tree

- When removing an item from a search tree, it is imperative that the tree satisfies the data ordering criterion.
- If the item to be removed is in a leaf node, then it is fairly easy to remove that item from the tree since doing so does not disturb the relative order of any of the other items in the tree.

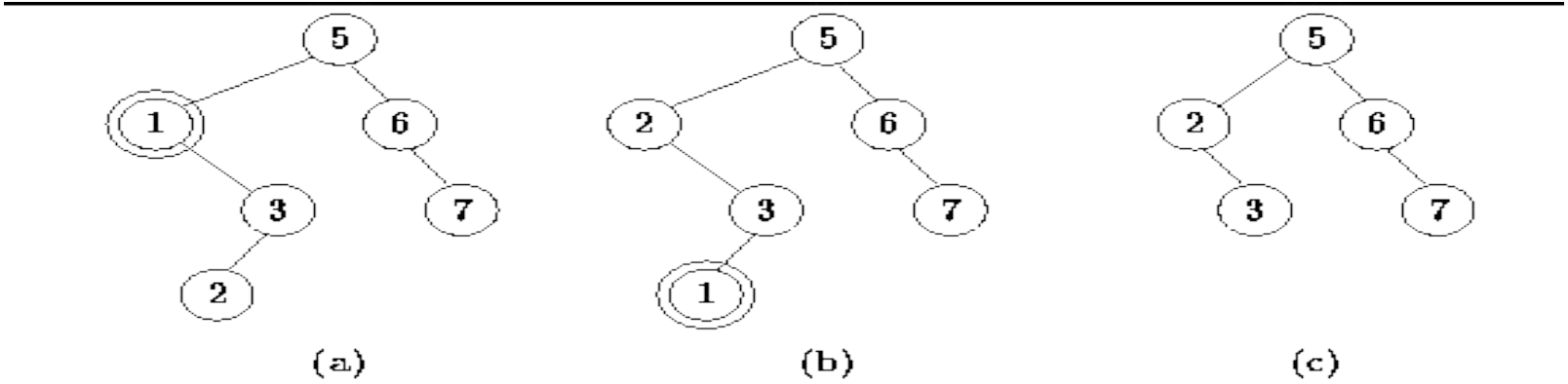
Example

- For example, consider the binary search tree shown in Figure (a). Suppose we wish to remove the node labeled 4. Since node 4 is a leaf, its subtrees are empty. When we remove it from the tree, the tree remains a valid search tree as shown in Figure (b).



Removing Non Leaf Node from a Binary Search Tree

- To remove a non-leaf node, we move it down in the tree until it becomes a leaf node since a leaf node is easily deleted.
- To move a node down we swap it with another node which is further down in the tree.
- For example, consider the search tree shown in Figure (a). Node 1 is not a leaf since it has an empty left sub tree but a non-empty right subtree.
- To remove node 1, we swap it with the smallest key in its right subtree, which in this case is node 2, Figure (b).
- Since node 1 is now a leaf, it is easily deleted. Notice that the resulting tree remains a valid search tree, as shown in Figure (c).



- To move a non-leaf node down in the tree, we either swap it with the smallest key in the right subtree or with the largest one in the left subtree.
- At least one such swap is always possible, since the node is a non-leaf and therefore at least one of its subtrees is non- empty.
- If after the swap, the node to be deleted is not a leaf, then we push it further down the tree with yet another swap. Eventually, the node must reach the bottom of the tree where it can be deleted

Creation Binary Tree from Traversal Sequence

- Creation of Binary Tree from Preorder & Inorder Traversals
- E.g. Inorder - E A C K F H D B G
- Preorder - F A E K C D H G B

- Creation of Binary Tree from Postorder & Inorder Traversals.
- Inorder: B I D A C G E H F
- Postorder: I D B G C H F E A

M way Tree

multiway search tree of order m (or an m-way search tree).

Definition -

m-way search tree should satisfy -

Each node has 0 .. m subtrees

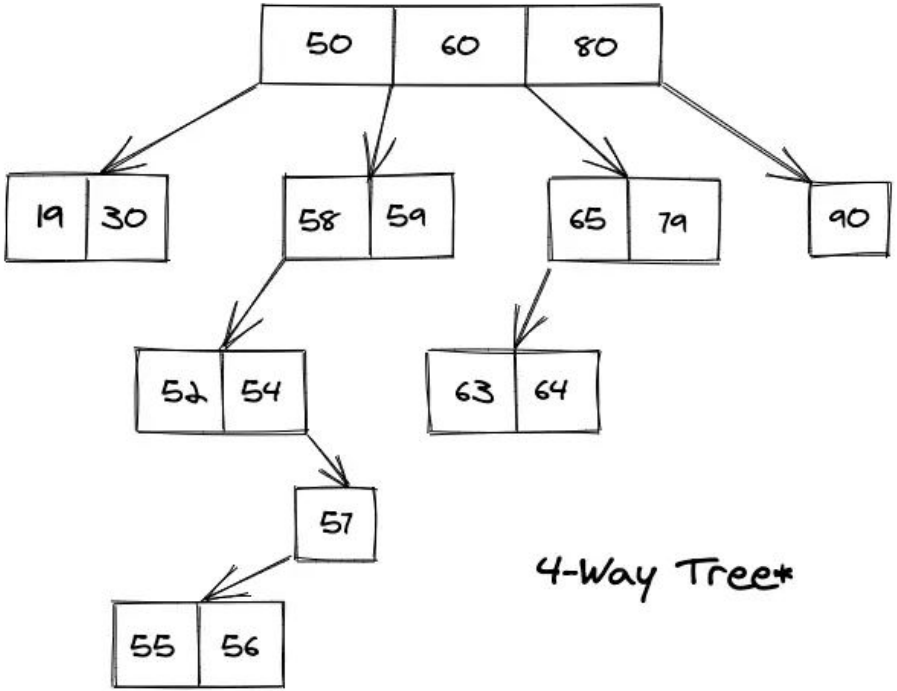
A node with k < m subtrees, contains k-1 keys.

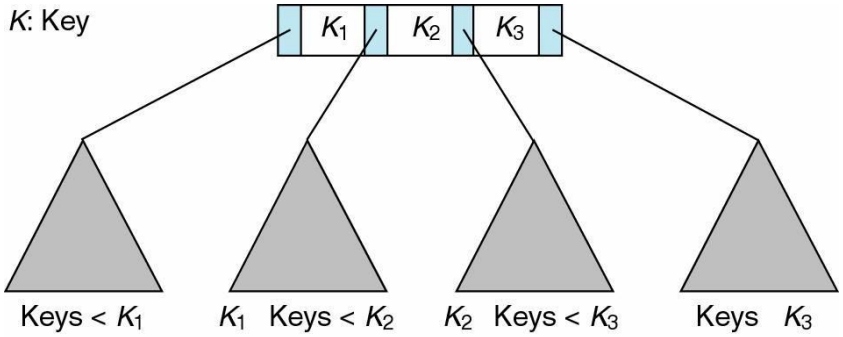
The key values of the first subtree are all less than the key value.

The data entries are ordered.

All subtrees are m-way trees.

Example


50
60
80
19
30
58
59
65
79
90
52
54
63
64
57
55
56
4-Way Tree*


K: Key
K1
K2
K3
Keys < K1
K1 Keys < K2
K2 Keys < K3
Keys K3

B Tree

The m-way tree is not balanced.

In 1970 Bayer and McCreight created the B-tree data structure.

In B-Tree a node can store more than one value (key) and it can have more than two children.

B-Tree is a self-balanced search tree with multiple keys in every node and more than two children for every node.

B-tree is an extension of m-way tree with the following additional properties:

Property #1 - All leaf nodes must be at same level.

Property #2 - All nodes except root must have at least [m/2]-1 keys and maximum of m-1 keys.

Property #3 - All non leaf nodes except root (i.e. all internal nodes) must have at least m/2 children.

Property #4 - If the root node is a non leaf node, then it must have at least 2 children.

Property #5 - A non leaf node with n-1 keys must have n number of children.

Property #6 - All the key values in a node must be in Ascending Order.

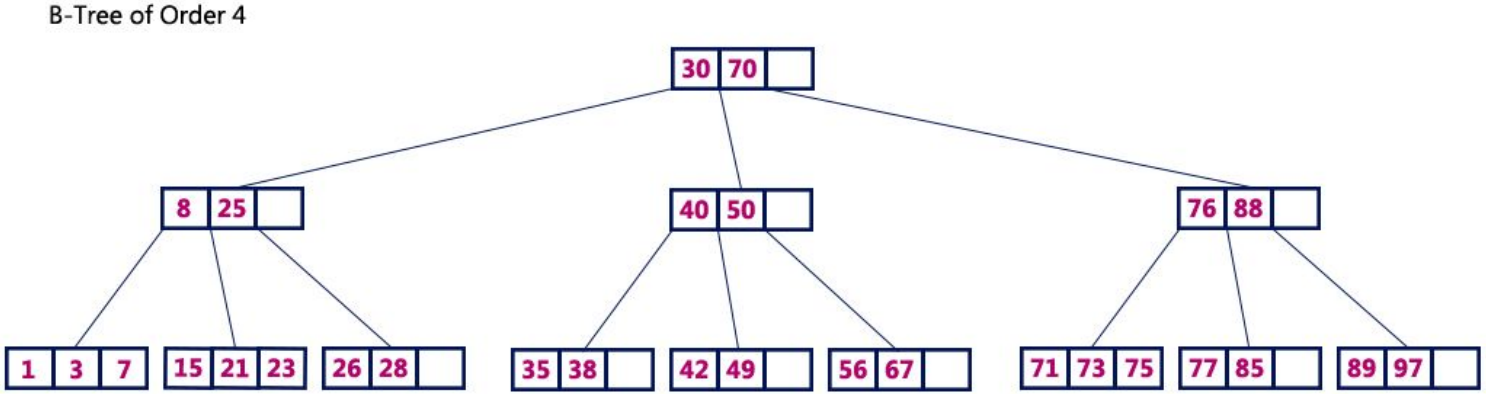
Operations on a B-Tree

The following operations are performed on a B-Tree...

Search

insertion

Deletion



B+ Tree

B+ tree is a variation of B-tree data structure.

In a B+ tree, data pointers are stored only at the leaf nodes of the tree.

Structure of a leaf node differ from the structure of internal nodes.

The leaf nodes have value of the search field, along with a data pointer to the record (or to the block that contains this record).

The leaf nodes of the B+ tree are linked together to provide ordered access on the search field to the records.

The drawback of B-tree used for indexing stores the data pointer (a pointer to the disk file block containing the key value), corresponding to a particular key value, along with that key value in the node of a B-tree.

reduces the number of entries that can be packed into a node of a B-tree, increase in the number of levels in the B-tree,

increasing the search time of a record.

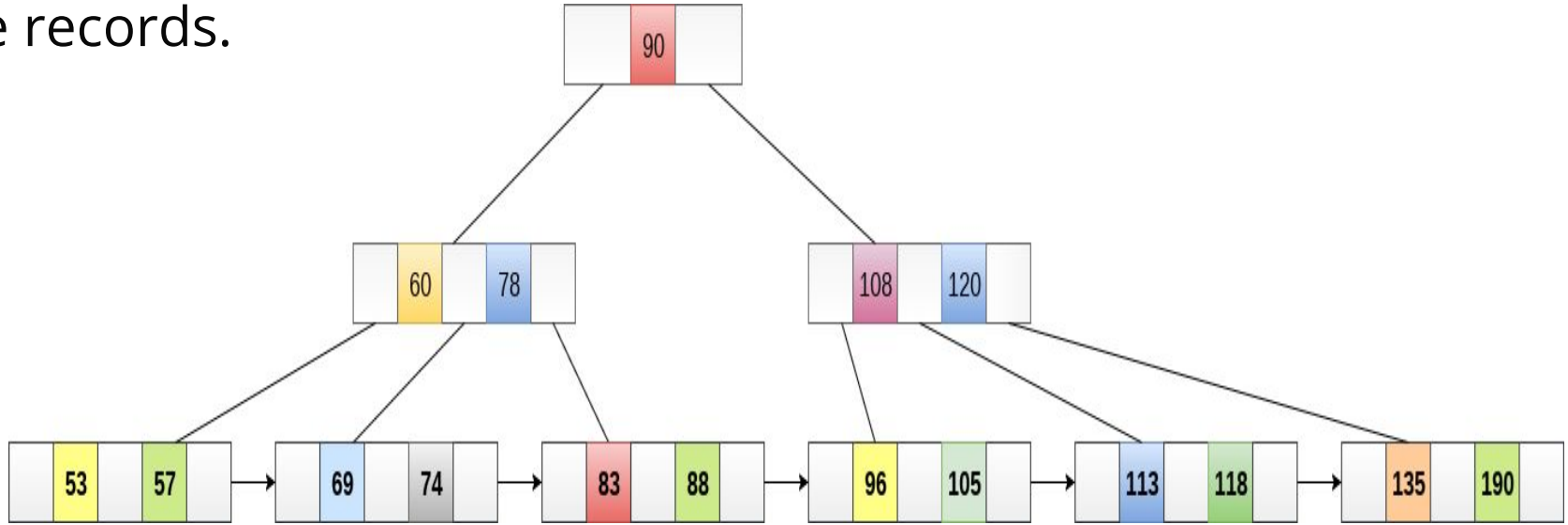
Why B+ tree

storing data pointers only at the leaf nodes of the tree.

structure of leaf nodes of a B+ tree is quite different

leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, in order to access them.

Moreover, the leaf nodes are linked to provide ordered access to the records.



Advantages of B+ Tree

Records can be fetched in equal number of disk accesses.

Height of the tree remains balanced and less as compare to B tree.

We can access the data stored in a B+ tree sequentially as well as directly.

Keys are used for indexing.

Faster search queries as the data is stored only on the leaf nodes