Trees

Doan Nhat Quang

doan-nhat.quang@usth.edu.vn
University of Science and Technology of Hanoi
ICT department

Today Objectives

- Introduce the principles of data structure: Tree
- Study well-known problems then implement examples in C/C++.

Tree

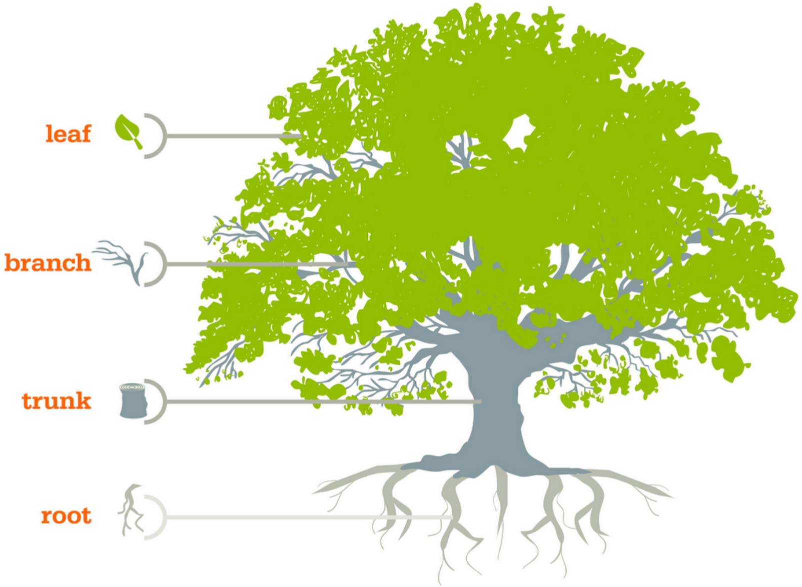
So far we have seen linear structures

- lists, arrays, stacks, queues

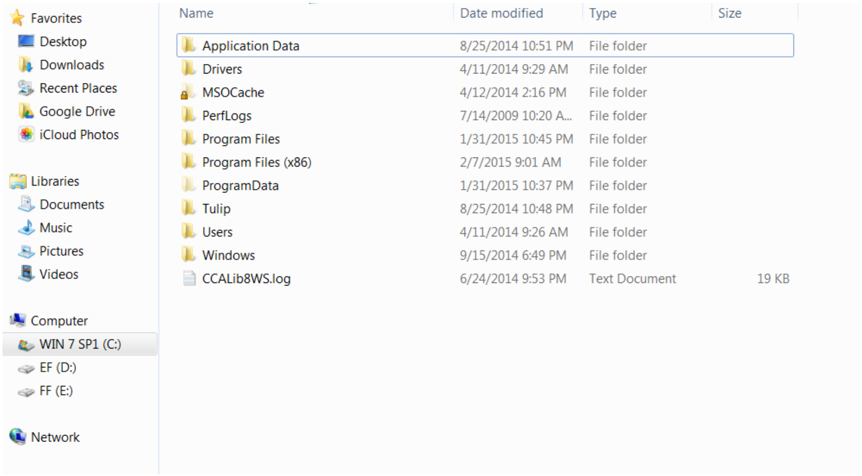
Today, we study non-linear structure: Trees

- probably the most fundamental structure in computing
- hierarchical structure

Tree



Tree



Name	Date modified	Type	Size
Application Data	8/25/2014 10:51 PM	File folder	
Drivers	4/11/2014 9:29 AM	File folder	
MSOCache	4/12/2014 2:16 PM	File folder	
PerfLogs	7/14/2009 10:20 A...	File folder	
Program Files	1/31/2015 10:45 PM	File folder	
Program Files (x86)	2/7/2015 9:01 AM	File folder	
ProgramData	1/31/2015 10:37 PM	File folder	
Tulip	8/25/2014 10:48 PM	File folder	
Users	4/11/2014 9:26 AM	File folder	
Windows	9/15/2014 6:49 PM	File folder	
CCALib8WS.log	6/24/2014 9:53 PM	Text Document	19 KB

Definition

Definition

A tree T is a set of nodes storing elements such that the nodes have a parent-child relationship:

- if T is not empty, T has a special tree node called the root r that has no parent.
- each node c of T different than the root has a unique parent node p; each node with parent p is a child of c

Recusive Definition

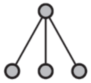
A tree T is:

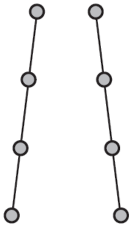
- T is empty
- T consists of a node r (the root) and a possibly empty set of trees whose roots are the children of r (or called subtrees)

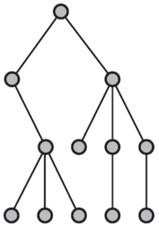
Tree

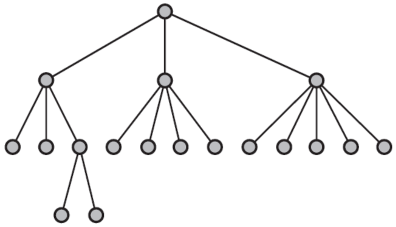
(a)
(a) is an empty tree

(b)

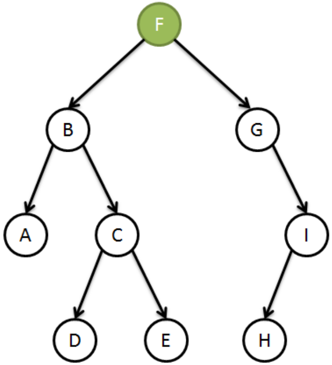

(c)


(d) (e)


(f)


(g)

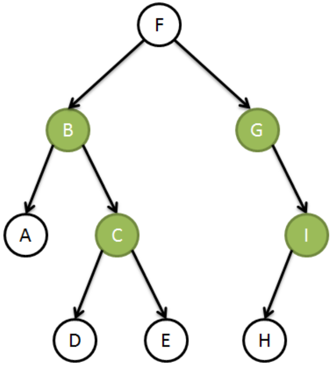
Terminology



Root

- The root of a tree is the starting node, the node that does not have a parent.
- A tree has one unique and only one root.
- The root is the node F.

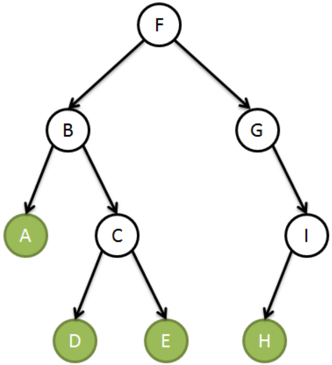
Terminology



Internal Node

- Internal nodes (parent nodes) are the nodes that have children.
- F, B, G, D, I are the internal nodes.
- F has two child nodes B and G or B and G have the parent node F

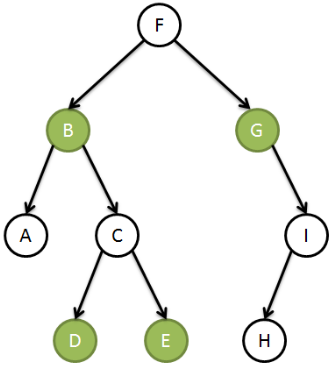
Terminology



Terminal Node

- Terminal nodes (called leaves) are the nodes that don't have any child node.
- A, D, E, H are the terminal nodes.

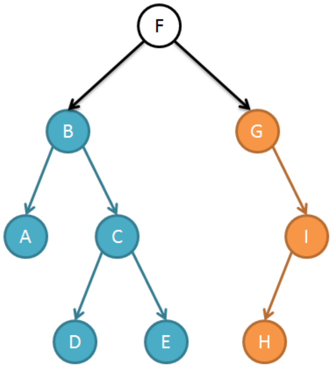
Terminology



Siblings

- Siblings nodes are the nodes that have the same parent node.
- B and G are siblings because their common parent is F; D and E are siblings. because their common parent is C.

Terminology



Subtree

- An internal node or a terminal node may be considered as the root of a subtree.
- B and G are the root of two subtree of the tree whose the root is F.

Application

Application

Tree ADT are widely used:

- Searching problems: Binary Search Tree.
- Presentation of complex data: tree data.
- Data Mining problems: declare variables for hierarchical methods Decision Tree, AntTree, etc.

Tree ADT

Tree ADT can be as following:

- init(): initialize an empty tree
- isRoot(): verify whether a node is the tree root
- isLeaf(): verify whether a node is terminal
- isEmpty(): verify whether a tree is empty
- getParent(): return the parent node of a node or null if this node is the root
- getChildren(): return the child nodes of a node or null if this node does not have any child

Tree ADT

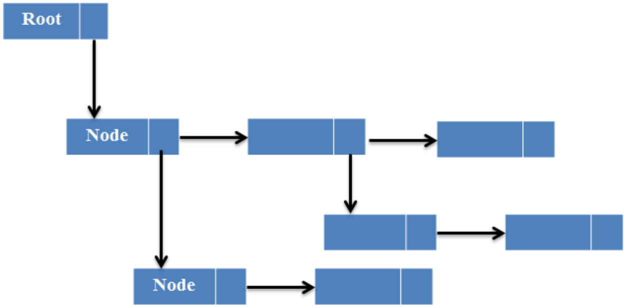
Application

Tree ADT can be as following:

- search(): find the node within the tree structure
- display(): show tree nodes' information
- insert(): add a node to the tree
- remove(): remove a node from the tree

Tree ADT

```c
typedef struct TreeNode{
      int val;
      struct TreeNode *firstChild;
      struct Treenode *nextSiblings;
} *Tree;
```



Binary Tree

Definition

A binary tree is a tree such that every node has at most 2 children denoted a left node or a right node

Recursive Definition

A binary tree T either is an empty tree or consists of:

- the root stores an element
- two binary trees: left subtree of T and right subtree of T

Binary Tree

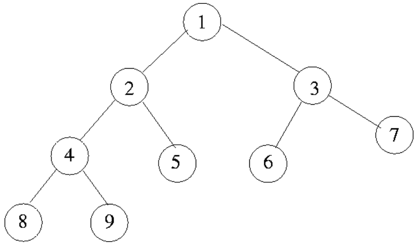
Definition

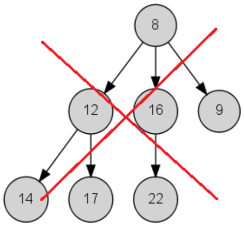
A binary tree is a tree such that every node has at most 2 children denoted a left node or a right node

Recursive Definition

A binary tree T either is an empty tree or consists of:

- the root stores an element
- two binary trees: left subtree of T and right subtree of T



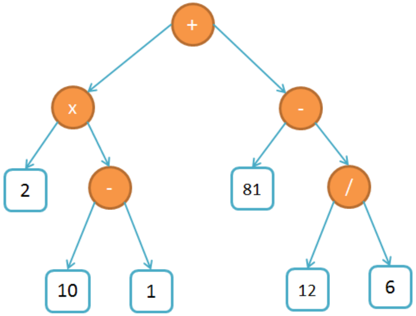


Binary Tree

Binary Tree can be used to represent arithmetic expression

- internal nodes: operators
- leaf nodes: operands

Example: $2 \times (10 - 1) + 81 - 12/6$

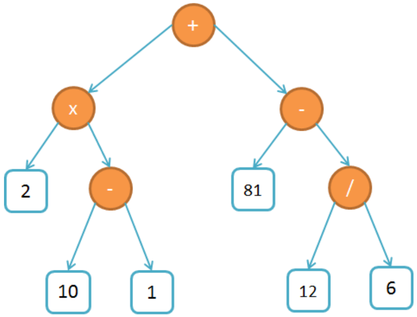


Binary Tree

Binary Tree can be used to represent arithmetic expression

- internal nodes: operators
- leaf nodes: operands

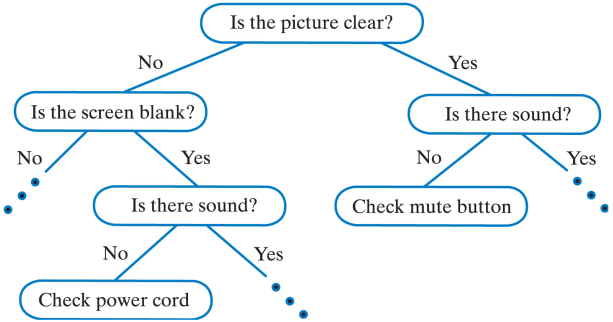
Example: $2 \times (10 - 1) + 81 - 12/6$



Binary Tree

Decision Tree is a kind of binary tree

- internal nodes: boolean test
- leaf nodes: decision



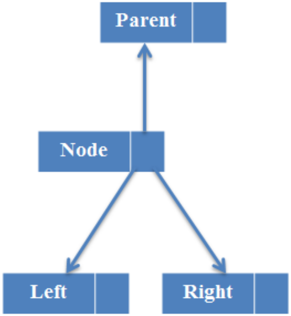
Binary Tree

Tree ADT can be as following:

- init(): initialize an empty tree
- isRoot(): verify whether a node is the tree root
- isLeaf(): verify whether a node is terminal
- isEmpty(): verify whether a tree is empty
- getParent(): return the parent node of a node or null if this node is the root
- getLeft(), getRight(): return the left or right child node of a node or null if otherwise

Binary Tree

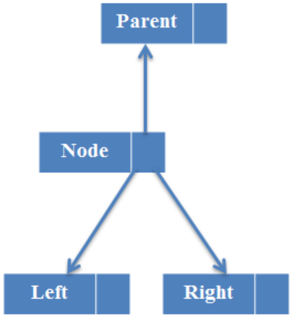
```
typedef struct TreeNode{
      int val;
      struct TreeNode *parent;
      struct TreeNode *tLeft;
      struct TreeNode *tRight;
} *Tree;
```



Write the function init() to initilize an empty tree

Binary Tree

```c
typedef struct TreeNode{
      int val;
struct TreeNode *parent;
      struct TreeNode *tLeft;
      struct TreeNode *tRight;
} *Tree;
```



Application

Tree ADT can be as following:

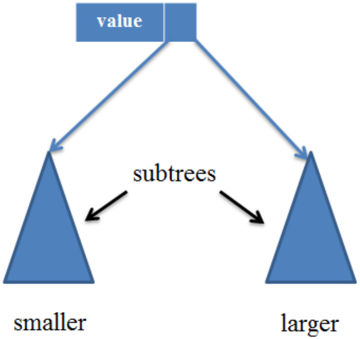
- insert() (insertLeft(), insertRight()): add a node to the tree
- remove(): remove a node from the tree if it does not have any child node
- display(): show tree nodes' information

Binary Search Tree

Definition

A binary search tree is a binary tree with a special property:

- every node value is larger than all values in its left subtree
- every node value is smaller than all values in its right subtree

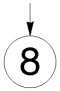


Binary Search Tree

Input array:

8	4	12	2	14	3	5	7	9	10

Binary Search Tree

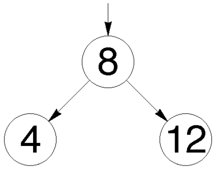


Binary Search Tree

Input array:

8	4	12	2	14	3	5	7	9	10

Binary Search Tree

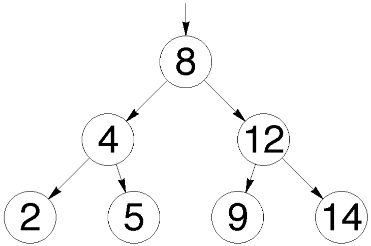


Binary Search Tree

Input array:

8	4	12	2	14	3	5	7	9	10

Binary Search Tree

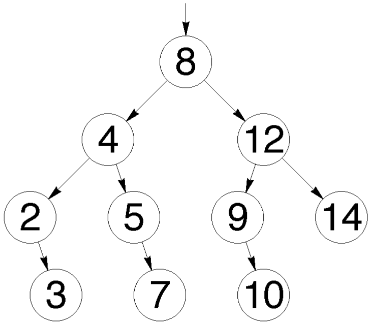


Binary Search Tree

Input array:

8	4	12	2	14	3	5	7	9	10

Binary Search Tree



Binary Search Tree

Searching

- Binary Tree is like unsorted array, this structure is not good enough for indexing or searching. However, due to the Binary Search Tree rules, BST allows to search quickly (Binary Search).
- To search a given key in Bianry Search Tree, we first compare it with root, if the key is present at root, we return root.
  - If key is greater than roots key, we recur for right subtree of root node.
  - Otherwise we recur for left subtree.

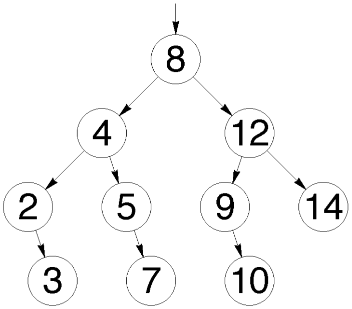
Binary Search Tree

Tree traversal refers to the process of visiting (checking and/or updating) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited.

We may do these things in any order and still have a legitimate traversal. If we do (L) before (R), we call it left-to-right traversal, otherwise we call it right-to-left traversal.

- Pre-order
- In-order
- Post-order

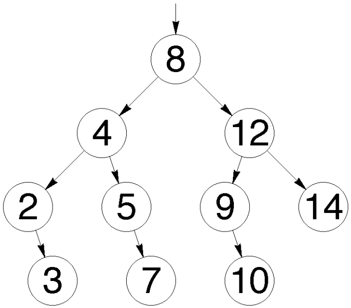
Binary Search Tree



Pre-order NLR (Node, Left, Right)

- Display the data part of the root (or current node).
- Traverse the left subtree by recursively.
- Traverse the right subtree by recursively.

Binary Search Tree

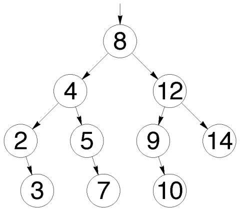


Pre-order NLR (Node, Left, Right)

- Display the data part of the root (or current node).
- Traverse the left subtree by recursively.
- Traverse the right subtree by recursively.
- Display: 8 4 2 3 5 7 12 9 10 14

Binary Search Tree

Pre-order traversal (NLR):



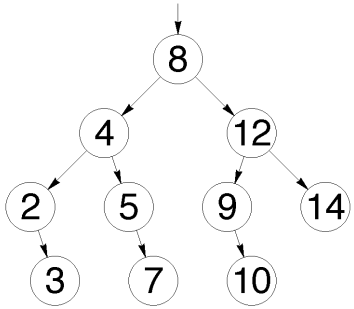
Step 1	8	L8	R8							
Step 2		4	L4	R4	R8					
Step 3			2	R2	R4	R8				
Step 4				3	R4	R8				
Step 5					5	R5	R8			
Step 6						7	R8			
Step 7							12	L12	R12	
Step 8								9	R9	R12
Step 9									10	R12
Step 10										14
Output	8	4	2	3	5	7	12	9	10	14

Binary Search Tree

NLR: Pre-order

```
void NLR (Tree tree) {
    if(tree != NULL) {
        cout << tree->val << ' ';
        NLR(tree->pLeft);
        NLR(tree->pRight);
    }
}
```

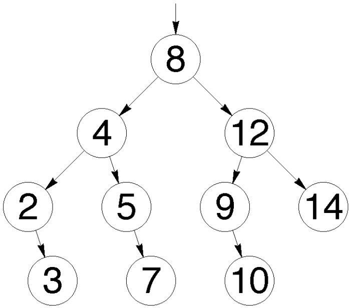
Binary Search Tree



In-order LNR (Left, Node, Right)

- Traverse the left subtree by recursively.
- Display the data part of the root (or current node).
- Traverse the right subtree by recursively.
- Display:

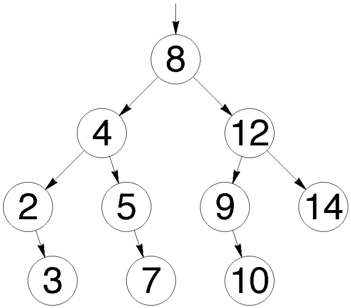
Binary Search Tree



Post-order LRN (Left, Right, Node)

- Traverse the left subtree by recursively.
- Traverse the right subtree by recursively.
- Display the data part of the root (or current node).
- Display:

Binary Search Tree

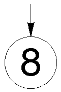


Post-order LRN (Left, Right, Node)

- Traverse the left subtree by recursively.
- Traverse the right subtree by recursively.
- Display the data part of the root (or current node).
- Display:

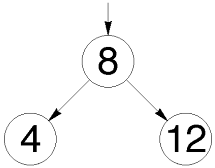
Binary Search Tree

To insert a new node, we start searching from root till we find a node which can be either a leaf node or a node having only one child. Once a leaf node is found, the new node is added as a child of this node.



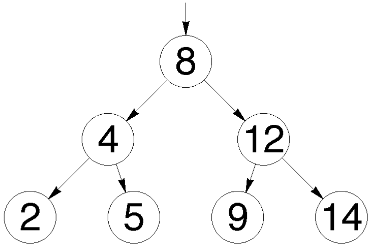
Binary Search Tree

To insert a new node, we start searching from root till we find a node which can be either a leaf node or a node having only one child. Once a leaf node is found, the new node is added as a child of this node.



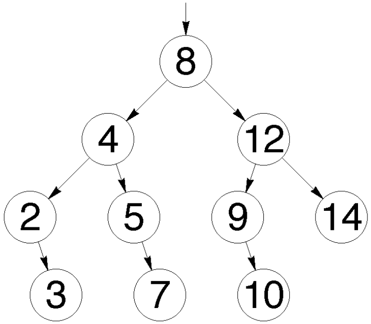
Binary Search Tree

To insert a new node, we start searching from root till we find a node which can be either a leaf node or a node having only one child. Once a leaf node is found, the new node is added as a child of this node.



Binary Search Tree

To insert a new node, we start searching from root till we find a node which can be either a leaf node or a node having only one child. Once a leaf node is found, the new node is added as a child of this node.



Binary Search Tree

```
int insert (Tree &tree, int x) {
    if (tree != NULL) {
         if (x == tree->val)
              return 0;
        else {
             if (x < tree->val)
                  insert(tree->tLeft, x);
             else
                  insert(tree->tRight, x);
        }
    } else{
        tree = new TreeNode;
        tree->val = x;
        tree->tLeft = tree->tRight = NULL;
        return 1;
    }
}
```

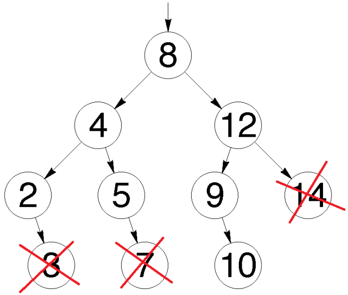
Binary Search Tree

There are three cases for removing a node from a tree:

1. removing a leaf node.
2. removing the root.
3. removing an internal node having a child.
4. removing an internal node having two children.

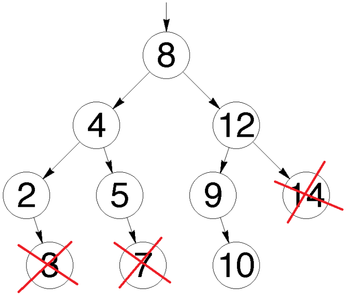
Binary Search Tree

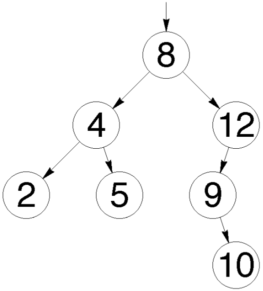
Case 1: removing a leaf node



Binary Search Tree

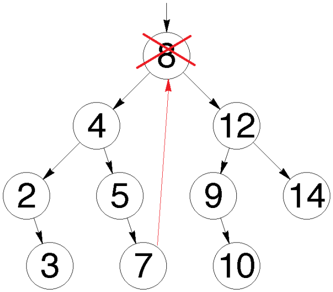
Case 1: removing a leaf node





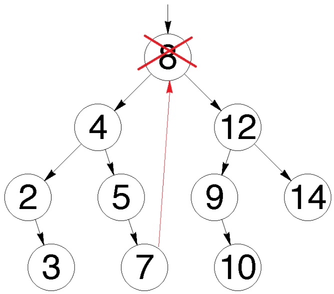
Binary Search Tree

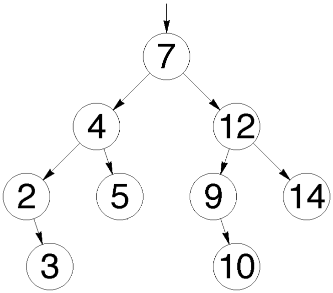
Case 2: removing the root



Binary Search Tree

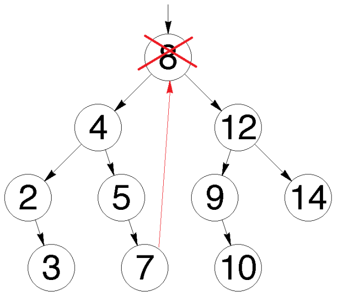
Case 2: removing the root

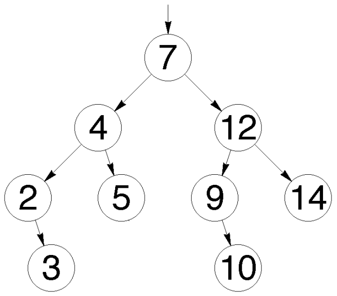




Binary Search Tree

Case 2: removing the root



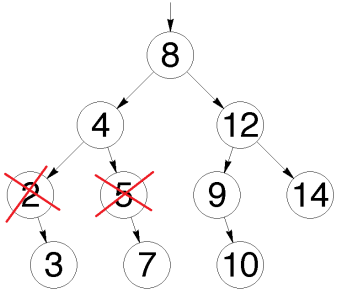


Replace the removed node by:

- the node having the minimum value of the right subtree.
- the node having the maximum value of the left subtree.

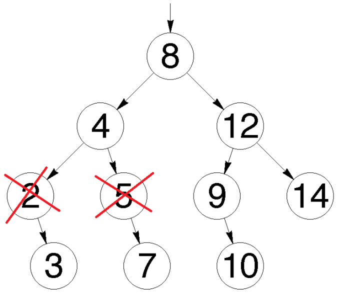
Binary Search Tree

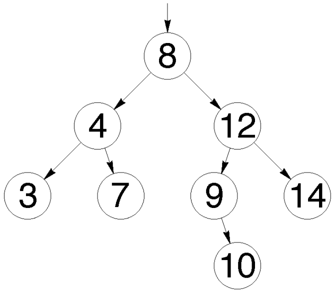
Case 3: removing an internal node having a child.



Binary Search Tree

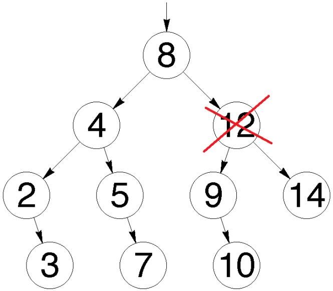
Case 3: removing an internal node having a child.





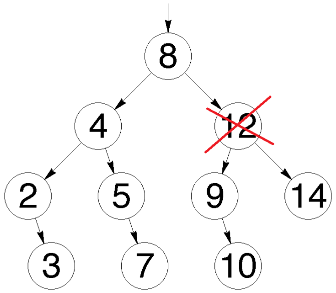
Binary Search Tree

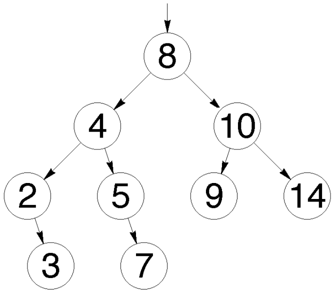
Case 4: removing an internal node having two children. This case can be considered as the second case where the node to be removed is the root of the subtree.



Binary Search Tree

Case 4: removing an internal node having two children. This case can be considered as the second case where the node to be removed is the root of the subtree.



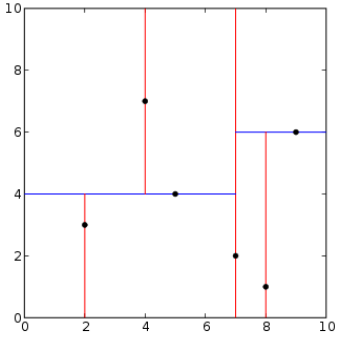


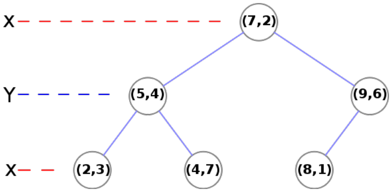
Binary Search Tree

```
int delNode(Tree &tree, int x){
    if (tree == NULL) return 0;
    else if (tree->val > x) return delNode(tree->tLeft, x);
    else if (tree->val < x) return delNode(tree->tRight, x);
    else{
        Tree P = tree;
        if (tree->tLeft == NULL) tree = tree->tRight;
        else if (tree->tRight == NULL) tree = tree->tLeft;
        else {
            Tree S = tree, Q = S->tLeft;
            while (Q->tRight != NULL){
                S = Q;
                Q = Q->tRight;
            }
            P->val = Q->val;
            S->tRight = Q->tLeft;
            delete Q;
        }
    }
    return 1;
}
```

k-d Tree

A k-d tree (k-dimensional tree) is a data structure for organizing points in a k-dimensional space. k-d trees can be considered as a special case of binary tree.



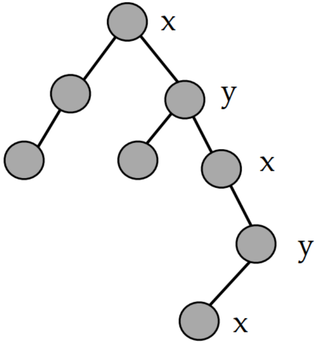


Example on a 2-D tree

k-d Tree

- Each level has a cutting dimension, a binary test for this dimension (if smaller on the left, larger on the right)
- Cycle through the dimensions as walking down the tree.
- Each node contains a point (x,y)

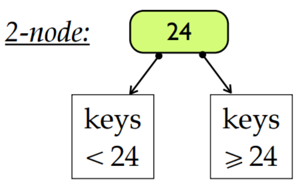
To find (x,y), only compare coordinate from the cutting dimension

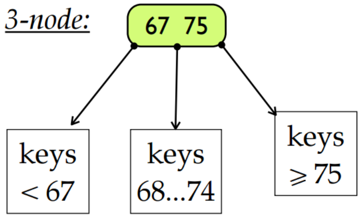


B-Tree

B-Tree: a generalization of a binary search tree

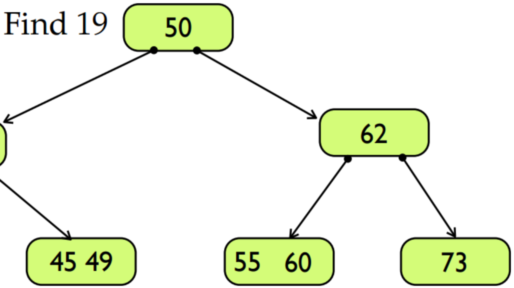
- all leaves are at the same level
- each internal node has either 2 or 3 children
- 2 children, it has 1 key (a test); 3 children, it has 2 key





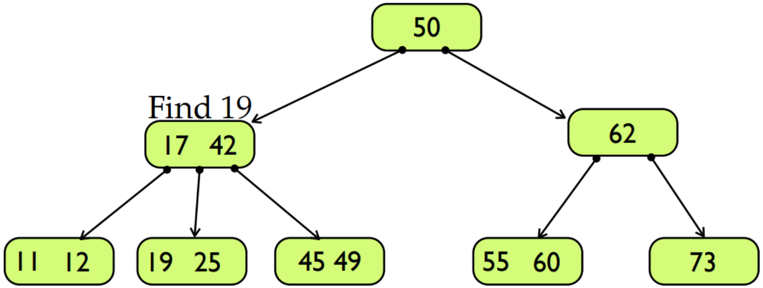
2,3-Tree

Searching with 2,3-Tree



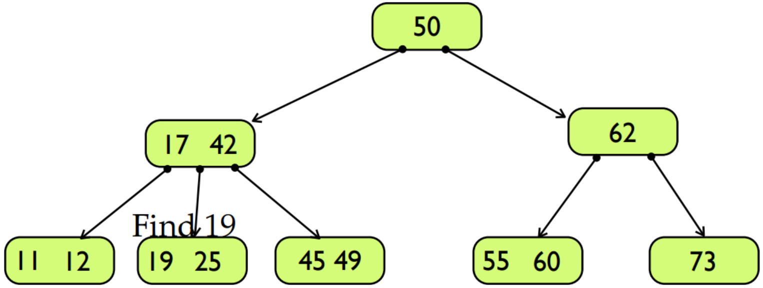
2,3-Tree

Searching with 2,3-Tree



2,3-Tree

Searching with 2,3-Tree


50
17 42
62
Find 19
11 12
19 25
45 49
55 60
73