UCLA Samueli Computer Science

CS32: Introduction to Computer Science II
Discussion Week 9

Yichao (Joey)

March 6, 2020

Credit: Carey Nachenberg, Junheng Hao

Outline Today

- Tree
- Binary Search Tree

Announcements

- Project 4 is due 11:00 PM Tuesday, March 12.

Tree

Definition

- Terms: Node/edge, root node, leaf node, parent and child node, subtree, levels (height/depth).
- Features: No loop, no shared children
- Question: How many edges should there be in a tree with ***n*** nodes?
- Binary tree: no node has more than two children.
- Question: How many nodes can a binary tree of height ***h*** have? → Full binary tree

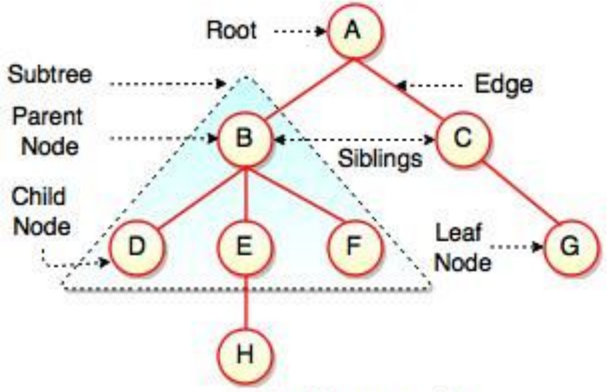
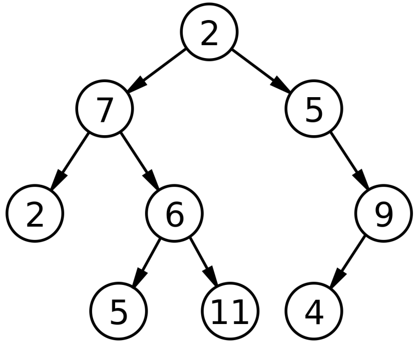


Fig. Structure of Tree



Tree

As a data structure

- Tree is a useful data structure!
- Basic functions: insert, remove, search, **traverse**
- How to traverse a tree?

```
struct Node{
  ItemType val;
  Node* leftChild;
  Node* rightChild;
} // a simple node
```

```
Class Tree{
public:
  // ???
private:
  // ???
}
```

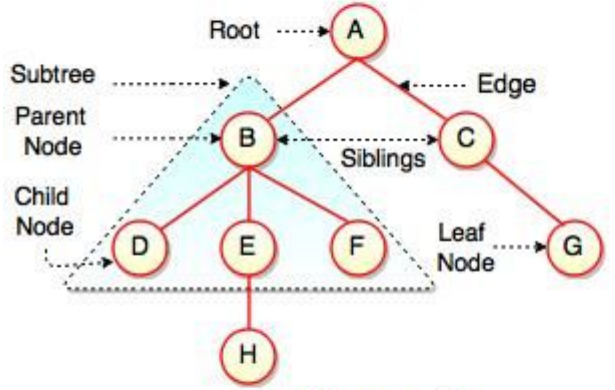
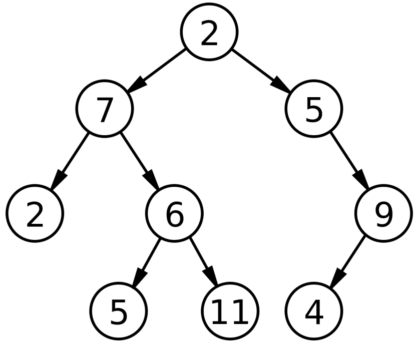


Fig. Structure of Tree



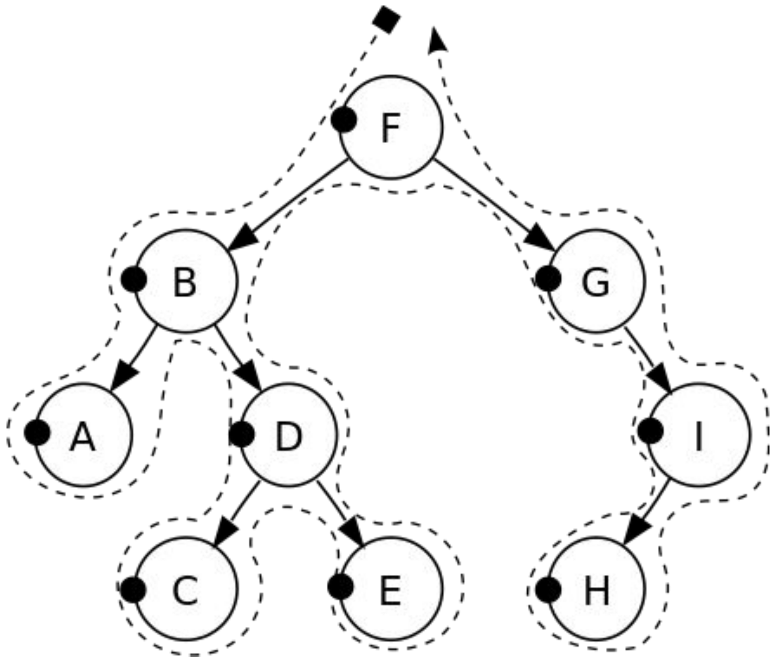
Tree Traversal: Pre-order

Three methods of tree traversal

```
void preorder(const Node* node)
{
  if (node == nullptr) return;
  cout << node->val << “,”;
  preorder(node->left);
  preorder(node->right);
}
```

Pre-order output:

```
F, B, A, D, C, E, G, I, H
```



Tree Traversal: In-order

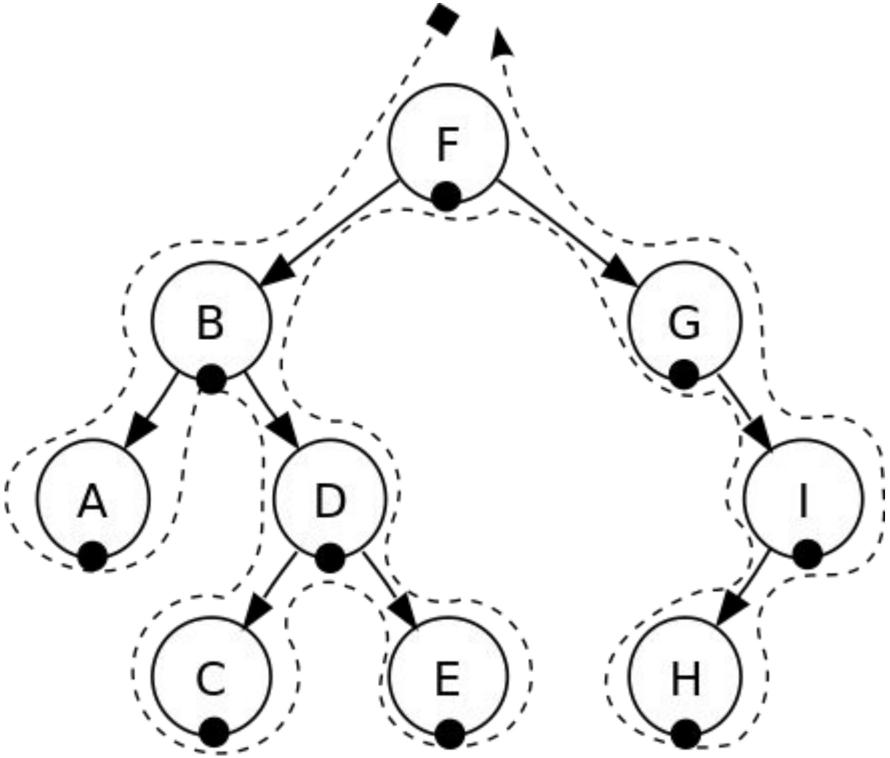
Three methods of tree traversal

UCLA Samueli Computer Science

```
void inorder(const Node* node)
{
  if (node == nullptr) return;
  inorder(node->left);
  cout << node->val << “,”;
  inorder(node->right);
}
```

In-order output:

```
A, B, C, D, E, F, G, H, I
```



Tree Traversal: Post-order

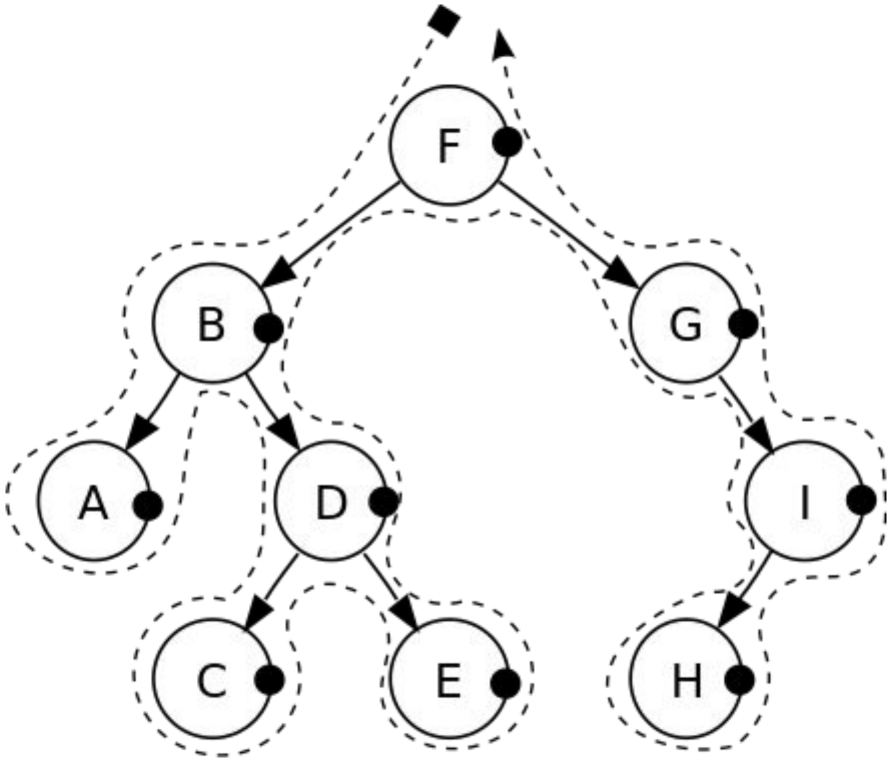
Three methods of tree traversal

UCLA Samueli Computer Science

```
void postorder(const Node* node)
{
  if (node == nullptr) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->val << “,”;
}
```

Post-order output:

```
A, C, E, D, B, H, I, G, F
```



Tree Traversal: Compare

UCLA Samueli Computer Science

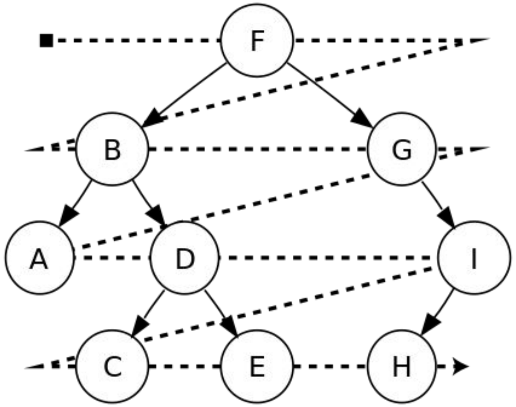
```
void preorder(const Node* node)
{
  if (node == nullptr) return;
  cout << node->val << “,”;
  preorder(node->left);
  preorder(node->right);
}
```

```
void inorder(const Node* node)
{
  if (node == nullptr) return;
  inorder(node->left);
  cout << node->val << “,”;
  inorder(node->right);
}
```

```
void postorder(const Node* node)
{
  if (node == nullptr) return;
  postorder(node->left);
  postorder(node->right);
  cout << node->val << “,”;
}
```

```
//Other ways?
```

Level-order or say breadth-first search!

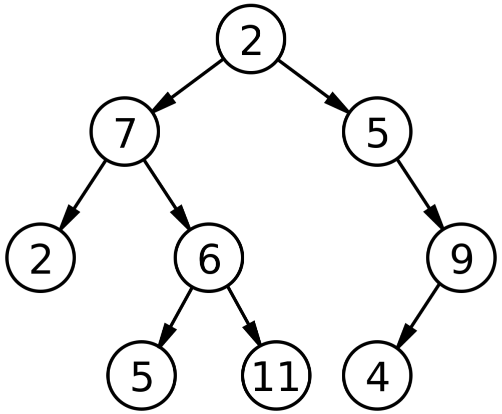


Tree

Recursion in trees

- It is easy and natural to apply recursion on trees!
- Pre-order / in-order / post-order are all recursive methods to traverse a tree.
- Question: How to calculate a height of a given tree?

```
int getTreeHeight(const Node* node)
{
  if (node == nullptr) return 0;
  int leftHeight = getTreeHeight(node->left);
  int rightHeight = getTreeHeight(node->right);
  if (leftHeight > rightHeight)
    return leftHeight + 1;
  else
    return rightHeight + 1;
}
```



Tree

Conclusions and applications

UCLA Samueli Computer Science

- Non-linear data structures unlike arrays or lists.
- Present hierarchy
- Optimized (like balanced) tree and variants can improve efficiency of search, sort and other typical problems.
- Other tree variants:
    - Binary search Tree
    - B tree / B+ tree
    - Red-black tree
    - K-D Tree
    - Suffix Tree

```
file system
-----------
     /    <-- root
  /     \
...    home
      /    \
   ugrad    course
    /      /   |   \
  ...   cs31 cs32 cs35L
```

Tree
Question 1

1. Write a function that takes a pointer to a tree and counts the number of leaves in the tree. In other words, the function should return the number of nodes that do not have any children. Note that this is not a binary tree. Also, it would probably help to use recursion. Example:

   Use the following Node definition and header function to get started.

```
Node {
   int val;
   vector<Node*> children;
}

int countLeaves(Node* root);
```

Tree

Question 1: Solution

```
int countLeaves(Node* root) {
     if (root == nullptr) // no leaves from this node
          return 0;
     if (root->children.size() == 0) // this node is a leaf
          return 1;

     int count = 0;
     for (int i = 0; i < root->children.size(); i++)
     {
          count += countLeaves(root->children[i]);
     }
     return count;
}
```

Tree

Question 2

1. Write a function that does a level-order traversal of a binary tree and prints out the nodes at each level with a new line between each level.

```
          1   <- root
         / \
        2   3
       / \   \
      4   5   6
     / \  /
    7  8 9
```

```
1
2 3
4 5 6
7 8 9
```

Question 2: Solution

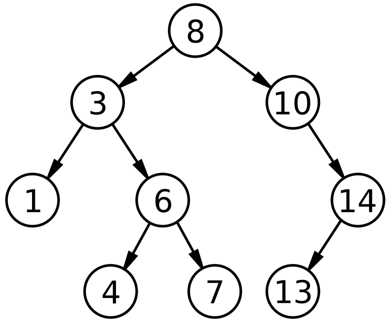
```
void levelOrder(Node* root){
     queue<Node*> cur;

     if(root != NULL)
          q.push(root);

     while(!cur.empty()){
          queue<Node*> next;
          while(!cur.empty()){
               Node* temp = q.front();
               q.pop();
               cout << temp->val << “ ”;
               if(temp->left)
                    next.push(temp->left);
               if(temp->right)
                    next.push(temp->right);
          }
          cout << “\n”;
          cur = next;

     }

}
```

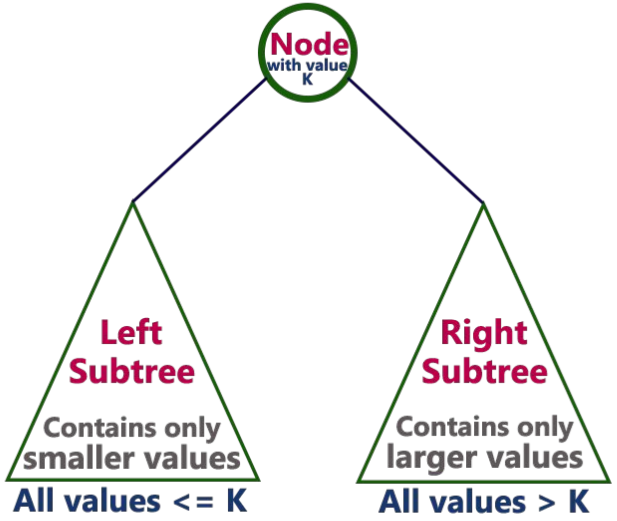
Binary Search Tree

Definition, Properties

UCLA Samueli Computer Science

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



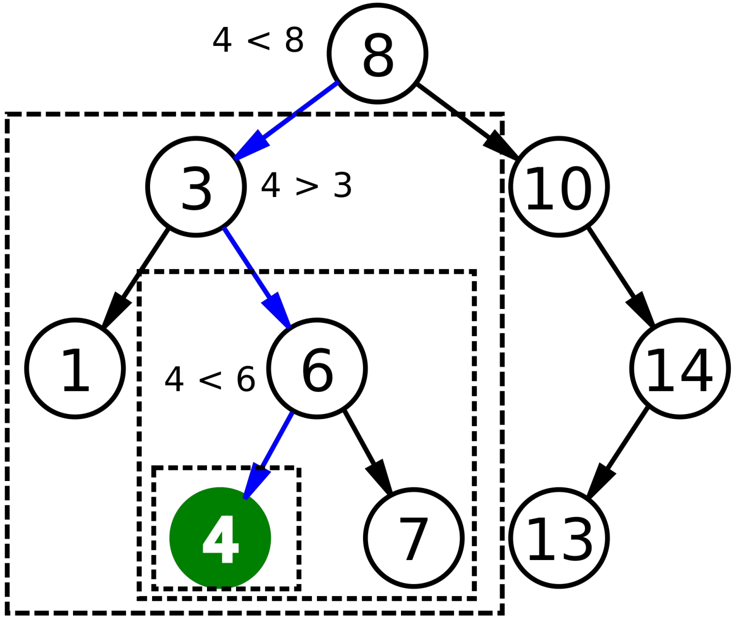


Binary Search Tree

Search

UCLA Samueli Computer Science

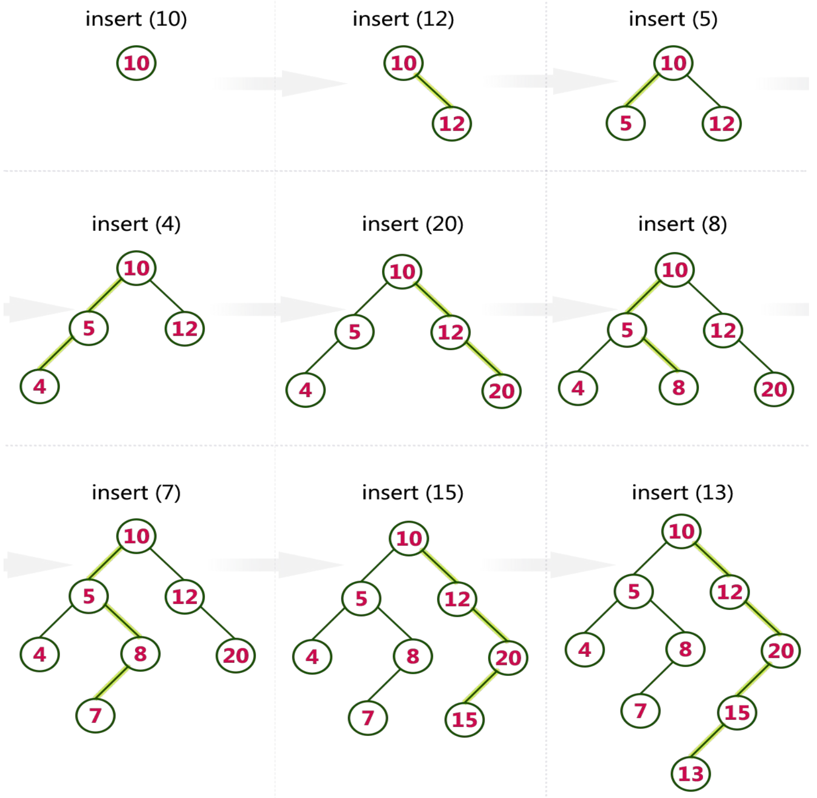
```
node* search(node* node, ItemType key)
{
    /* If the tree is empty, return null pointer */
    if (node == nullptr) return nullptr;

    /* compare with current node and decide*/
    if (key == node->key)
        return node;
    else if (key < node->key)
        return search(node->left, key);
    else
        return (node->right, key);
}
```



Binary Search Tree

Insertion

```
node* insert(node* node, ItemType key)
{
    /* If the tree is empty, return a new node */
    if (node == NULL) return newNode(key);

    /* Otherwise, recur down the tree */
    if (key < node->key)
        node->left  = insert(node->left, key);
    else if (key > node->key)
        node->right = insert(node->right, key);

    /* return the (unchanged) node pointer */
    return node;
}
```



Binary Search Tree

Deletion

UCLA Samueli Computer Science

- Deletion is the most tricky one.
- 3 cases:
  - The node is a leaf node.
  - The node has one child.
  - The node have two children.

Super easy! Just delete that node!

Not difficult! Copy the child to the node and delete the child.

Find inorder successor of the node. Copy contents of the inorder successor to the node and delete the inorder successor.
Note that inorder predecessor can also be used.

```
          50                                50
        /      \          delete(20)      /     \
      30        70        --------->    30       70
     /  \      /  \                       \     /  \
   20    40  60    80                      40  60    80
```

```
        50                                  50
      /      \            delete(30)      /    \
    30        70          --------->    40      70
      \      /  \                              /  \
       40  60    80                          60    80
```

```
       50                                 60
     /      \            delete(50)     /    \
   40        70          --------->   40      70
            /  \                                \
          60    80                               80
```

Binary Search Tree

Deletion

UCLA Samueli Computer Science

```
node* delete(node* node, ItemType key)
{
  if (node == nullptr) return nullptr;
  if (key < node->key) { node->left = delete(node->left, key); }
  else if (key > node->key) { node->right = delete(node->right, key); }
  else{
    /* case 1 & case 2*/
    if (node->left == nullptr) { node *temp = node->right; delete(node); return temp; }
    else if (node->right == nullptr) {node *temp = node->left; delete(node); return temp;}
    /* case 3 */
    node* temp = minValueNode(node->right);
    node->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
  }
}
```

Binary Search Tree

Analysis of BST

- Insertion
  - The (worst) case time complexity of search and insert operations is O(h) where h is height of Binary Search Tree. In worst case, we may have to travel from root to the deepest leaf node.
  - The height of a skewed tree may become n and the time complexity of search and insert operation may become O(n).
  - Average time complexity: O(log *n*)
- Deletion
  - Similar to insertion for complexity analysis

Binary Search Tree

FindMin and FindMax by using recursion

UCLA Samueli Computer Science

Question: How to test a tree is a valid BST?

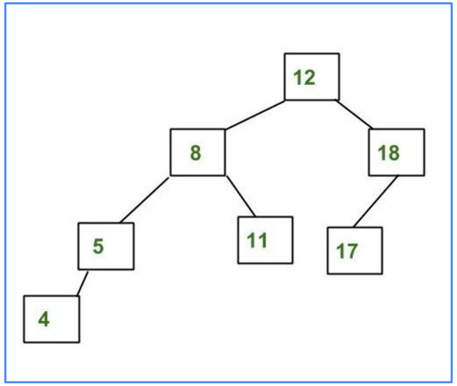
One possible recursion solution by using **`findMinKey`** and **`findMaxKey`**.

```
bool isValidBST(const node* node)
{
  if (node == nullptr) return true;
  /* check left subtree and right subtree condition*/
  if (node->left != nullptr && findMaxKey(node->left) > node->key)
    return false;
  if (node->right != nullptr && findMinKey(node->right) < node->key)
    return false;
  /* further check subtree with left child and right child */
  return isValidBST(node->left) && isValidBST(node->right)
}
```

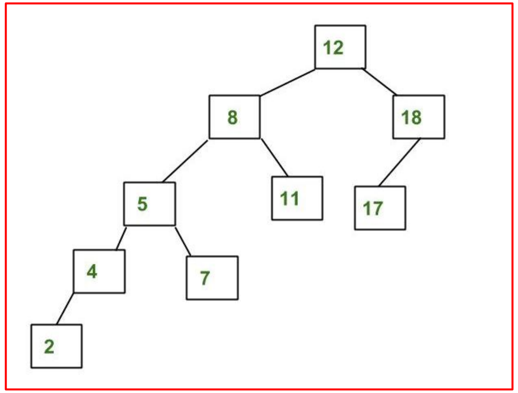
Beyond Binary Search Tree

AVL Tree

- What is the drawback of naive BST? → It can be skewed! Not good!
- AVL (Adelson-Velsky and Landis) Tree is a self-balancing BST.



The above tree is AVL because differences between heights of left and right subtrees for every node is less than or equal to 1.



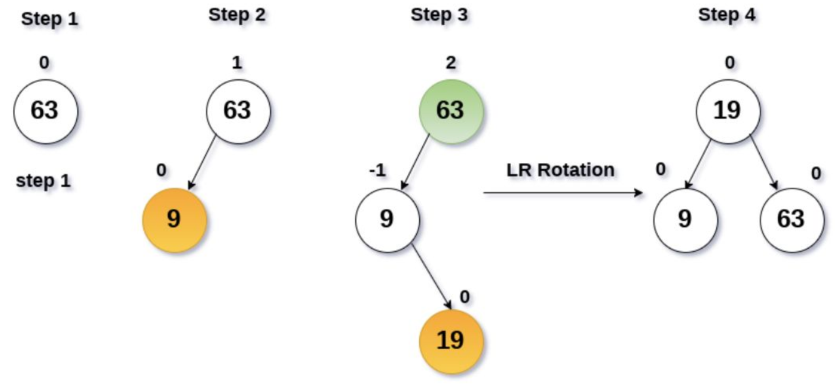
The above tree is not AVL because differences between heights of left and right subtrees for 8 and 18 is greater than 1.

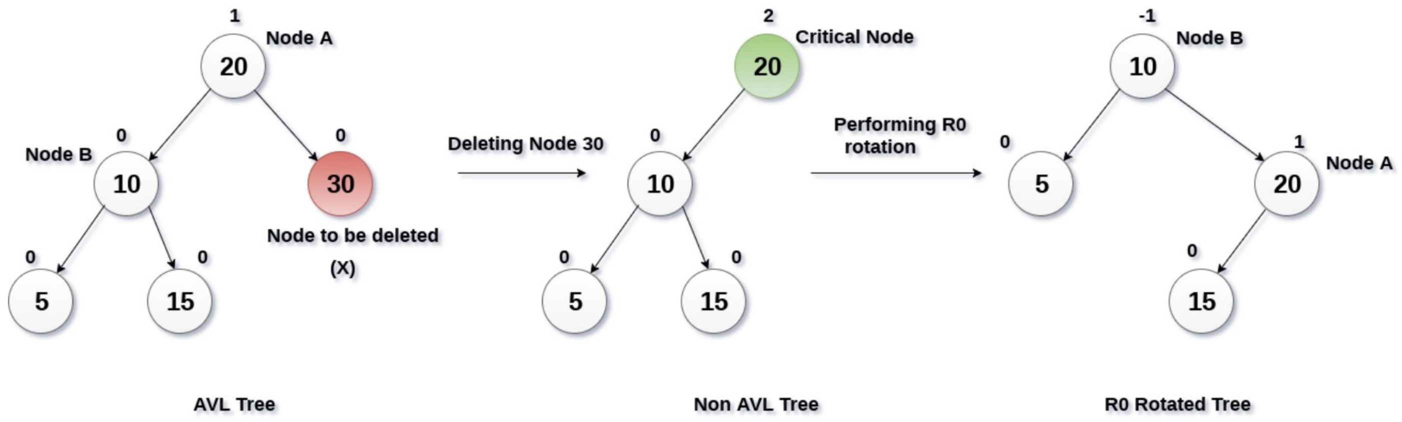
Beyond Binary Search Tree

AVL Tree: Operations

- Insertion
- Deletion

Please check this interesting demo!





Beyond Binary Search Tree

The tree family (1)

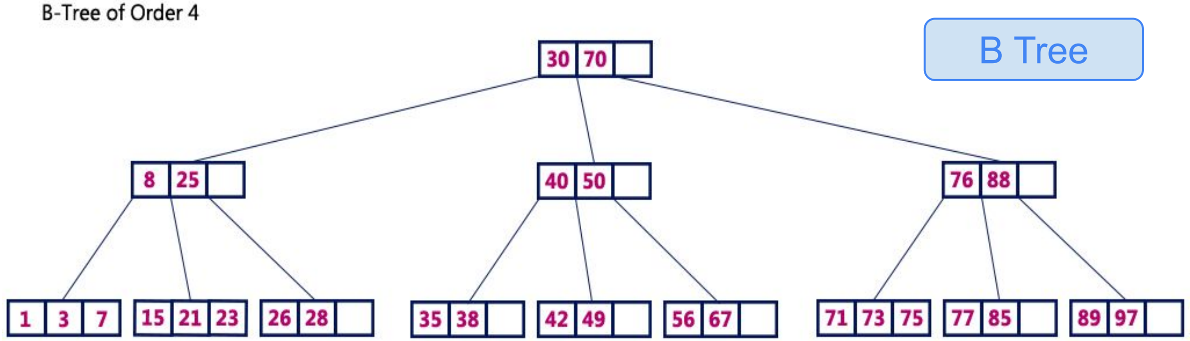
There are many interesting tree structures such as:

- B tree and B+ tree
- 2-3-4 tree
- R tree (spatial index tree)
- Red-black tree
- K-D tree

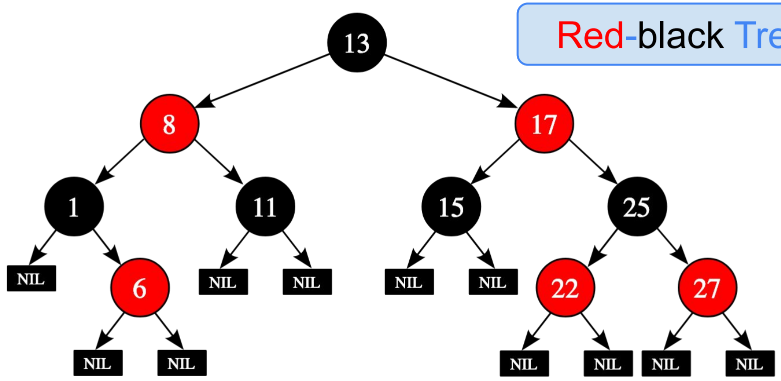
Beyond Binary Search Tree

The tree family (2)

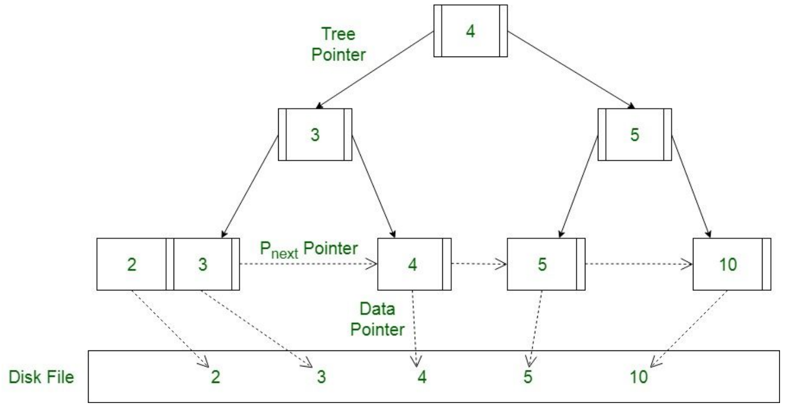
B Tree



Red-black Tree



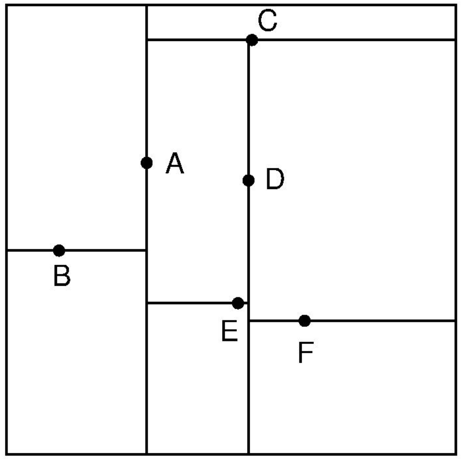
B+ Tree



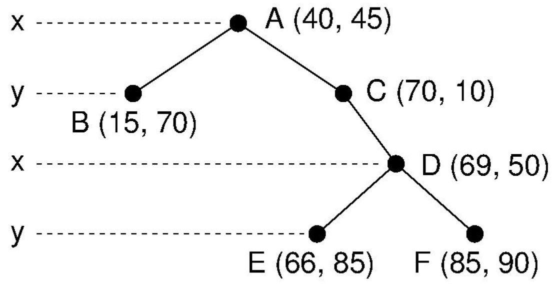
Beyond Binary Search Tree

The tree family (3)

KD Tree

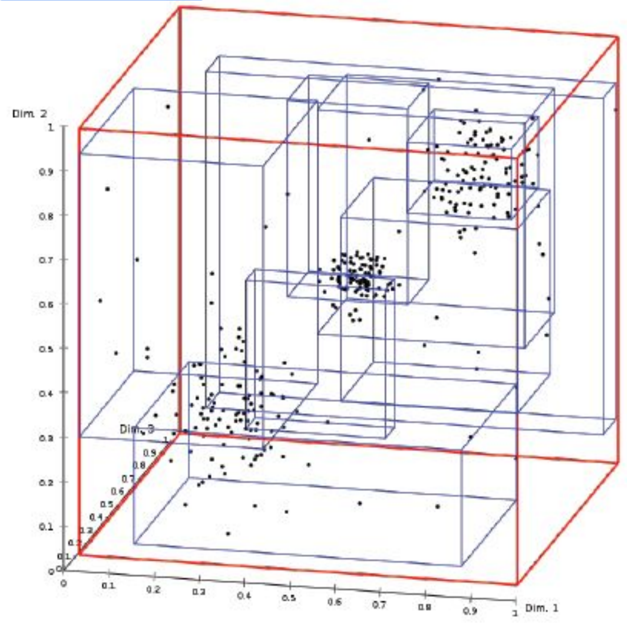


(a)



(b)

R Tree



Visualization of an R*-tree for 3D cubes using ELKI

Question 1

UCLA Samueli Computer Science

Note: The following questions deal with binary trees. They may or may not be binary search trees. Always read the question carefully!

Write a function that returns whether or not an integer value n is contained in a binary tree (that might or might not be a binary search tree). That is, it should traverse the entire tree and return true if a Node with the value n is found, and false if no such Node is found. (Hint: recursion is the easiest way to do this.)

```
class Node {
  int val;
  Node* left;
  Node* right;
};

bool treeContains(const Node* head, int n);
```

Question 1: Solution

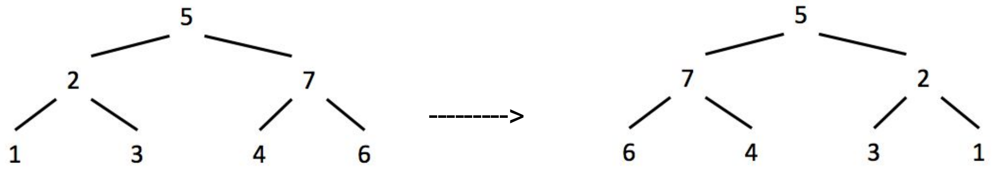
UCLA Samueli Computer Science

```
bool treeContains(const Node* head, int n) {
  // Base case
  if (head == nullptr) {
    return false;
  }
  else if (head->val == n) {
    return true;
  }
  else {
    // Check all children
    return treeContains(head->left, n) || treeContains(head->right, n);
  }
}
```

Question 2

Write a function that takes a pointer to the root of a binary tree and recursively reverses the tree.

Example:



Use the following Node definition and header function to get started.

```
Node {
   int val;
   Node* left;
   Node* right;
};

void reverse(Node* root)
```

Question 2: Solution

UCLA Samueli Computer Science

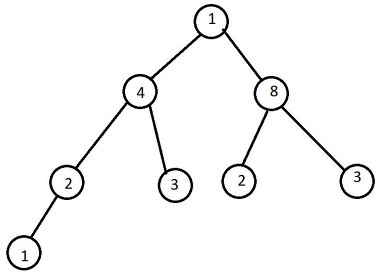
```
void reverse(Node* root) {
    if (root != nullptr) {
        Node* temp = root->left;
        root->left = root->right;
        root->right = temp;

        reverse(root->left);
        reverse(root->right);
    }
}
```

Question 3

Write a function that finds the maximum depth of a binary tree. A tree with only one node has a depth of 0; let's decree that an empty tree has a depth of -1.

```
struct Node {
	int val;
	Node* left, right;
};

int maxDepth(Node *root);
```

Example:



This tree has a maximum depth of 3.

Question 3: Solution

UCLA Samueli Computer Science

```
int maxDepth(Node* root) {
    if (root == nullptr)
        return -1;
    // computer depth of each subtree
    int lDepth = maxDepth(root->left);
    int rDepth = maxDepth(root->right);

    // return the max of the two
    if (lDepth > rDepth)
        return lDepth+1;
    else
        return rDepth+1;
}
```

Leetcode Question

Given a pointer to the root of a binary tree, determine whether or not the tree is a binary search tree.

```
class Node {
  int val;
  Node* left;
  Node* right;
};

bool isBinaryTree(Node* root);
```

Leetcode Question Solution

```
void inOrder(Node* root, vector<int>& v){
      if(root == NULL)
            return;
      traverse(root->left);
      v.push(root->val);
      traverse(root->right);

}

bool isBinaryTree(Node* root){
      vector<int> v;
      inOrder(root, v);
      return is_sorted(v.begin(), v.end());
}
```