CS 32 Week 8
Discussion 2C

UCLA CS

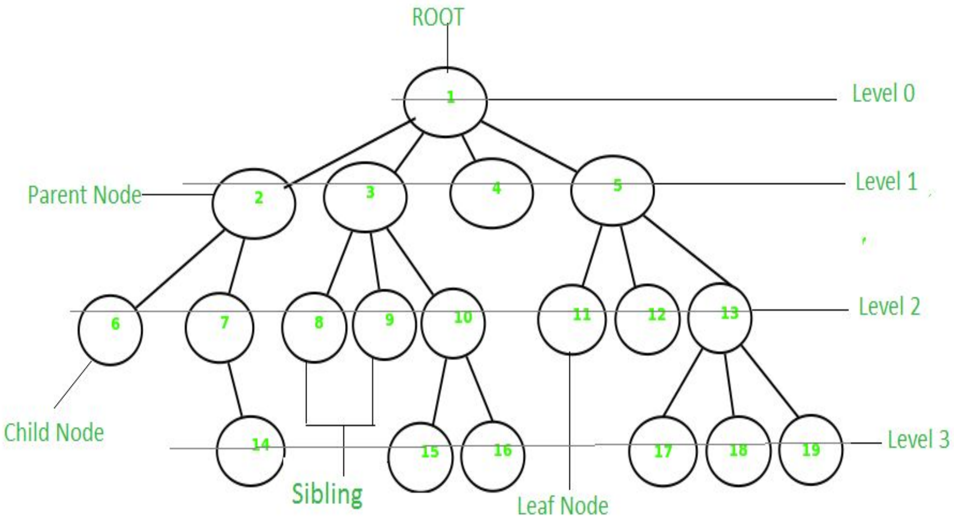
Yiyou Chen / Katie Chang

Topics

Tree

- Tree and Binary Tree
- General Tree:
    - Build a tree (insertion)
    - Tree traversal (preorder, postorder)
    - Deletion of nodes (by value)
    - Find the height of a tree
- Binary Search Tree
    - Insertion
    - Search by value
    - Traversal (preorder, inorder, postorder)
    - Deletion
    - Find kth smallest element
- Self-balancing Binary Search Trees and C++ STL: set, multiset, map

General Tree



Height: 3
Depth of node 11: 2
Leaf nodes: nodes without children.

```
struct Node {
    string val;
    vector <Node*> child;
};
Node* root = nullptr;
```

Image: www.geeksforgeeks.org

Binary Tree

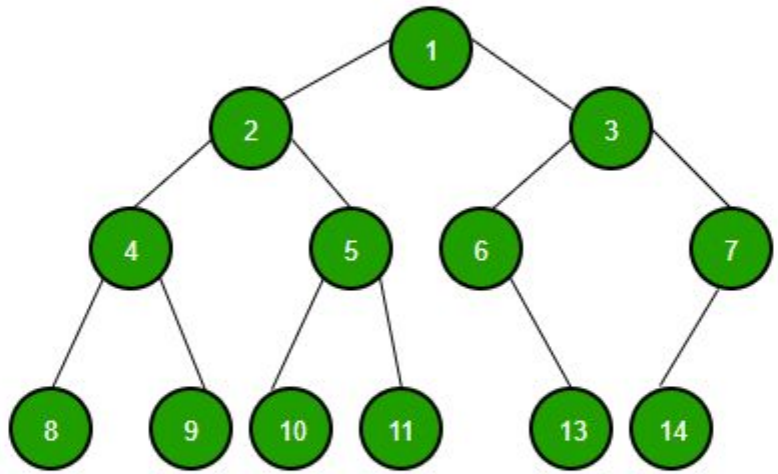


Image: www.geeksforgeeks.org

Fact: all general trees can be converted to binary trees (up to constant factor in depth). So we may use binary trees as the data structure. Parent, Left child, right child.

```
struct BNode {
  string val;
  BNode *left, *right;
};

BNode* Broot = nullptr;
```

General Tree: insert to Node* p

```
void Insert(Node*& p, string pval) {

}
```

General Tree: insert to Node* p

```
void Insert(Node*& p, string pval) {
  if (p == nullptr) { //empty tree
    p = new Node;
    p -> val = pval;
    return;
  }
  Node* newnode = new Node;
  newnode->val = pval;
  (p->child).push_back(newnode);
  return;
}
```

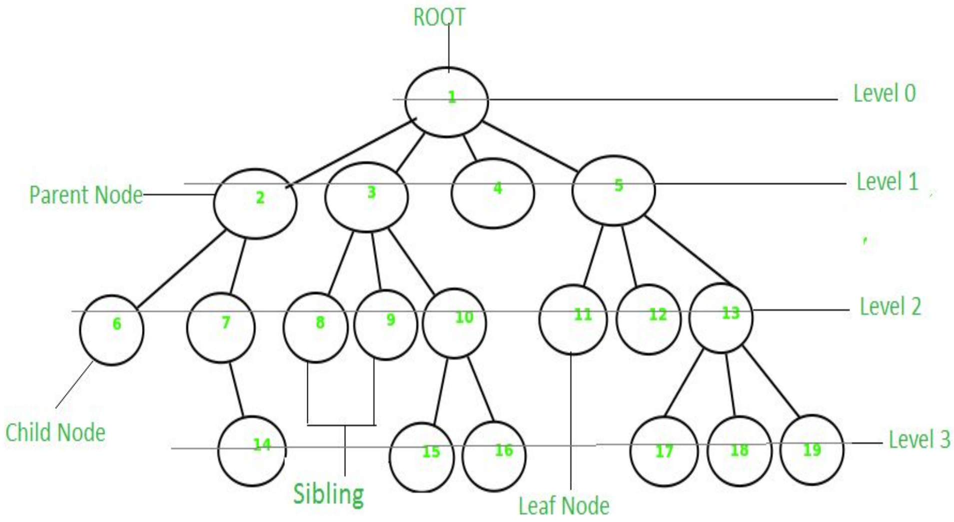
General Tree: insert to Node* p

```
void Insert(Node*& p, string pval) {
  if (p == nullptr) { //empty tree
    p = new Node;
    p -> val = pval;
    return;
  }
  Node* newnode = new Node;
  newnode->val = pval;
  (p->child).push_back(newnode);
  return;
}
```

Why by reference?

```
Node* root = nullptr;
Insert(root, "hi");
```

General Tree: traversal (preorder, postorder)



Preorder:

Parent node first, then children.

Postorder:

Children first, then parent node.

General Tree: traversal (preorder)

```
void Preorder(Node* p) {

}
```

General Tree: traversal (preorder)

```
void Preorder(Node* p) {
  if (p == nullptr) {
    return ;
  }
  cout << p->val << endl;

  for (auto it: p->child) {
    Preorder(it);
  }
}
```

General Tree: traversal (postorder)

```
void Postorder(Node* p) {

}
```

General Tree: traversal (postorder)

```
void Postorder(Node* p) {
  if (p == nullptr) {
    return ;
  }
  for (auto it: p->child) {
    Postorder(it);
  }
  cout << p->val << endl;
}
```

General Tree: deletion

```
Node* DeleteVal(Node* p, const string& pval){

}
```

```
root = DeleteVal(root, "hi");
```

General Tree: deletion

```
Node* DeleteVal(Node* p, const string& pval) {
  if (p == nullptr) return p; //if empty tree
  if (p->val == pval) {
    if ((p->child).empty()) { //leaf
      delete p;
      return nullptr;
    }
    Node* newnode = (p->child)[0];
    for (int i = 1; i < (p->child).size(); ++i) {
      (newnode->child).push_back((p->child)[i]);
    }
    delete p;
    return newnode;
  }
  for (auto it: p->child) {
    it = DeleteVal(it, pval);
  }
  vector<Node*>::iterator it = (p->child).begin();
  while(it != (p->child).end()) {
    if( (*it) == nullptr)
      it = (p->child).erase(it);
    else ++it;
  }
  return p;
}
```

General Tree: find height

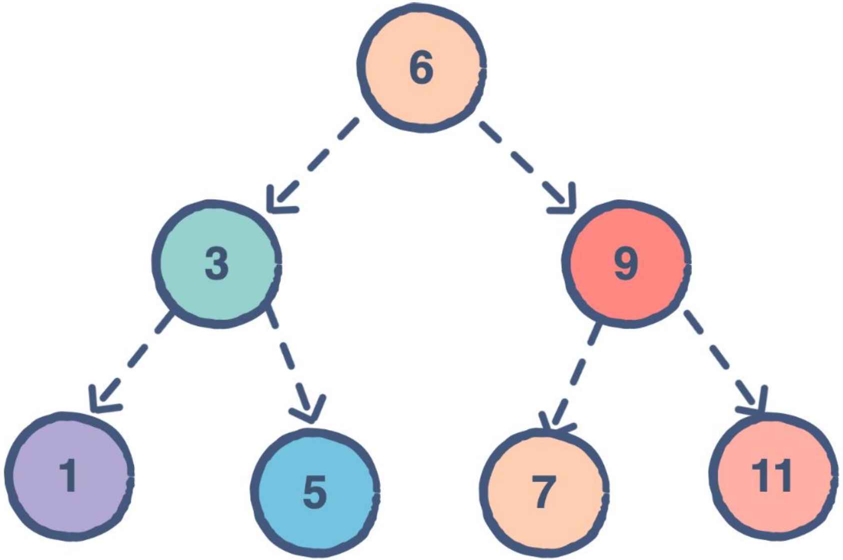
```
int Find_Height(Node* p) {

}
```

General Tree: find height

```
int Find_Height(Node* p) {
  if (p == nullptr) return -1;
  if ((p->child).empty())
    return 0;
  int max_h = 0;
  for (auto it: p->child) {
    max_h = max(max_h, Find_Height(it));
  }
  return max_h + 1;
}
```

Binary Search Tree (BST)



An example of a binary search tree

Binary tree

val(left_child)<val(parent)<val(right_child)

Image: https://medium.com

Binary Search Tree: insertion

```
void BInsert(BNode*& p, const string& pval) {

}
```

Binary Search Tree: insertion

```
void BInsert(BNode*& p, const string& pval) {
  if (p == nullptr) { //empty node
    p = new BNode;
    p->val = pval;
    p->left = p->right = nullptr;
    return;
  }
  if (pval < p->val) { //insert left
    BInsert(p->left, pval);
  }
  else if(pval == p->val) //already exists
    return;
  else //insert right
    BInsert(p->right, pval);
}
```

Average: O(logn)

Binary Search Tree: look up value

```
bool BSearch(BNode* p, const string pval) {

}
```

Binary Search Tree: look up value

```
bool BSearch(BNode* p, const string pval) {
  if (p == nullptr) return false;
  if (pval == p->val)
    return true;
  if (pval < p->val)
    return BSearch(p->left, pval);
  return BSearch(p->right, pval);
}
```

Average: O(logn)

Binary Search Tree: traversal

Preorder, inorder, postorder traversal. Pre, in, post indicate the place we put the parent.

Preorder: parent, left, right
Inorder: left, parent, right
Postorder: left, right, parent

What does inorder traversal do?
Tree sort

We'll implement postorder traversal for example.

Binary Search Tree: Postorder Traversal

```
void BPostorder(BNode* p) {

}
```

Binary Search Tree: Postorder Traversal

```
void BPostorder(BNode* p) {
  if (p == nullptr) return ;
  BPostorder(p->left);
  BPostorder(p->right);
  cout << p->val << endl;
}
```

Binary Search Tree: Deletion

```
BNode* BDelete(BNode* p, const string& pval) {

}
```

```
Broot = BDelete(Broot, "hi");
```

Binary Search Tree: Deletion

```
BNode* largest(BNode* p) {
  return (p->right == nullptr) ? p : largest(p->right);
}
//Assume each value occurs once
BNode* BDelete(BNode* p, const string& pval) {
  if (p == nullptr)
    return p;
  if (pval < p->val)
    p->left = BDelete(p->left, pval);
  else if (pval > p->val)
    p->right = BDelete(p->right, pval);
  else {
    //leaf
    if(p->left == nullptr && p->right == nullptr) {
      delete p;
      return nullptr;
    }
    //left or right empty
    else if (p->left == nullptr || p->right == nullptr) {
      BNode* temp = (p->left == nullptr) ? p->right : p->left;
      delete p;
      return temp;
    }
    else {
      //find largest in left subtree
      BNode* temp = largest(p->left);
      p->val = temp->val;
      p->left = BDelete(p->left, temp->val);
    }
  }
  return p;
}
```

Average: O(logn)

Binary Search Tree: find kth smallest element

Allow duplicates

```
string Find_k_th(BNode* p, const int k) {

}
```

Binary Search Tree: find kth smallest element

```
string Find_k_th(BNode* p, const int k) {

}
```

```
struct BNode {
  string val;
  BNode *left, *right;
  int m_size; //size of the subtree
};
```

```
void BInsert(BNode*& p, const string& pval) {
  if (p == nullptr) { //empty node
    p = new BNode;
    p->m_size = 1;
    p->val = pval;
    p->left = p->right = nullptr;
    return;
  }
  p->m_size++;
  if (pval < p->val) { //insert left
    BInsert(p->left, pval);
  }
  else //insert right
    BInsert(p->right, pval);
}
```

Binary Search Tree: find kth smallest element

```
string Find_k_th(BNode* p, const int k) {
  int l_size = 0;
  if (p->left != nullptr)
    l_size = p->left->m_size;
  if (l_size == k - 1) {
    return p->val;
  }
  else if (l_size < k - 1) {
    return Find_k_th(p->right, k - 1 - l_size);
  }
  else {
    return Find_k_th(p->left, k);
  }
}
```

Average: O(logn)

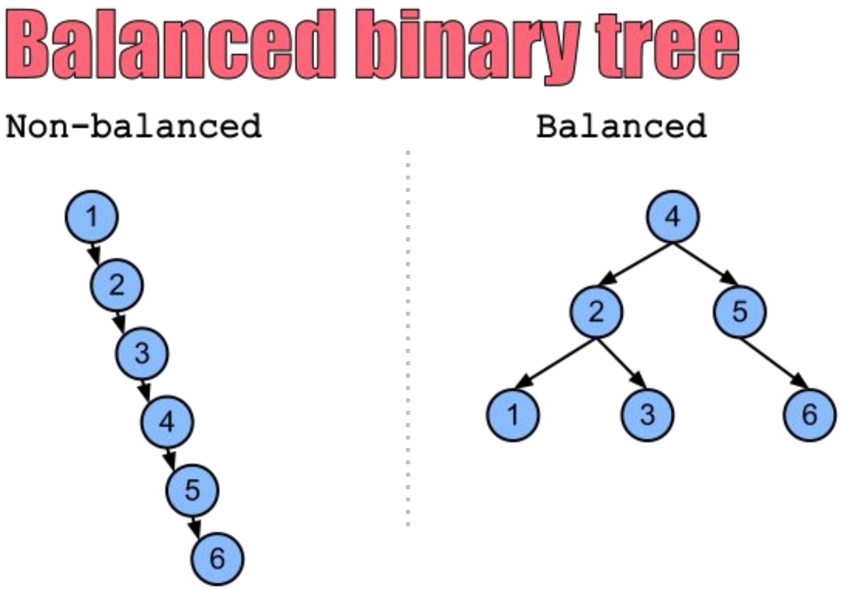
Binary Search Tree: Practice

Q1: Given n elements, what’s the time complexity of converting them to a BST structure?

Why might we use BST in this case (compared to saving them in an array which takes O(n) time)?

Q2: How would you keep track of the median of a data stream?

Self-balancing BST



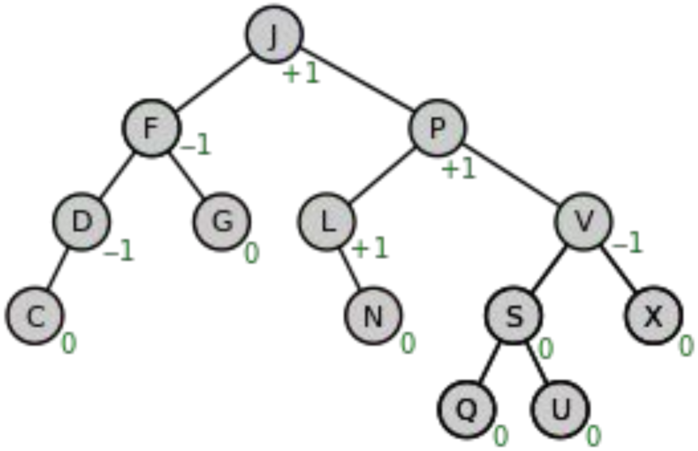
BST operations:
worst case O(n)

Self-balancing BST operations:
worst case O(logn)

Self-balancing BST: AVL Tree

Idea: keep the heights of left and right subtrees balanced.

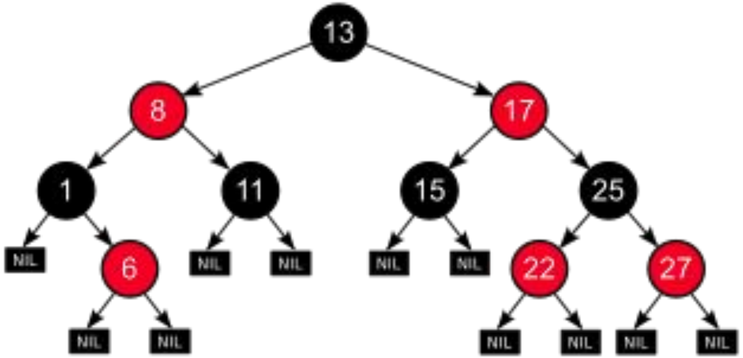
Balancing takes O(logn).



Self-balancing BST: Red-Black Tree

Idea: color the tree nodes to know where to insert.

Balancing takes O(logn).



Self-balancing BST: other popular ones

- 2-3 tree
- Splay tree
- Treap
- Scapegoat tree

As long as balancing takes O(logn) it has worst case complexity O(logn) for basic operations.

STL: Set, Multiset, and Map

All use self-balancing BST.

#include <set> for set and multiset.
set<type> s;
set<type> ms;
Set and multiset save data in sorted manner. Set auto removes duplicates. Multiset keeps duplicates.

#include <map> for map.
map<keytype, valuetype> m; (as in homework and projects)
Map saves paired data, which maps keys to corresponding values.
Map is auto sorted by keys.
Map removes duplicates and keeps the pair of the first occurrence of the key.

Please visit www.cplusplus.com for detailed usage.

STL: Set, Multiset, and Map

```
string s[7] = {"hi", "this", "is", "cs", "is", "at", "ucla"};
set<string> ss;
multiset<string> ms;
map<string, int> m;
for (int i = 0; i < 7; ++i) {
  ss.insert(s[i]); //no duplicates
  ms.insert(s[i]); //keeps duplicates
  m.insert(pair<string, int> (s[i], i));
}
cout << ms.count("is") << endl; //2
cout << m["is"] << endl; //2
for (auto it: ss) //at cs hi is this ucla
  cout << it << ' ';
for (auto it: ms) //at cs hi is is this ucla
  cout << it << ' ';
for (auto it: m) //(at,5) (cs,3) (hi,0) (is,2) (this,1) (ucla,6)
  cout << '(' << it.first << ',' << it.second<<')' << " ";
```

STL: Set, Multiset, and Map

What are the worst time complexity for insertion, deletion, and query for Set, Multiset, and Map?

How would you implement Multiset using (self-balancing)BST?