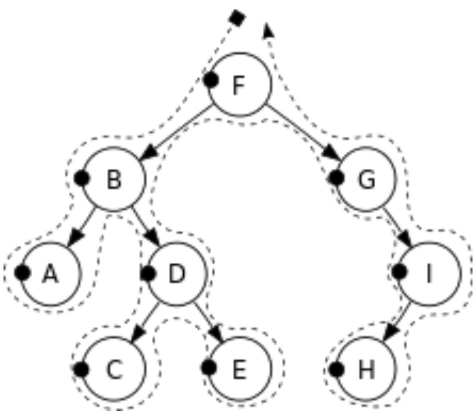
Trees

BST,c++,lecture11,lecture12,lecture13,traversal,trees

Tree Traversal

Pre-Order Traversal (touch the left side)



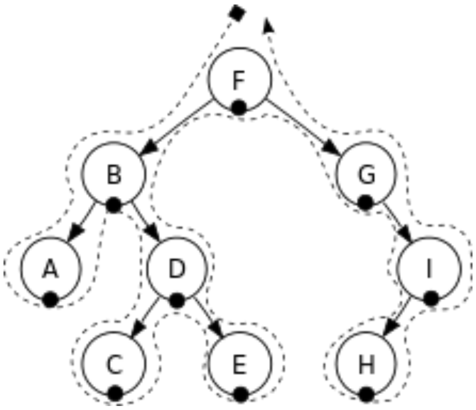
- process current node
- process left sub-tree
- process right sub-tree

```
void PreOrder(Node *cur)
{
    if (cur == NULL)      // if empty, return…
        return;

    cout << cur->value;        // Process the current node.

    PreOrder(cur->left);     // Process nodes in left sub-tree.
    PreOrder(cur-> right);  // Process nodes in left sub-tree.
}
```

In-Order Traversal (touch the bottom side)



- process nodes in left sub-tree
- process current node
- process nodes in right sub-tree

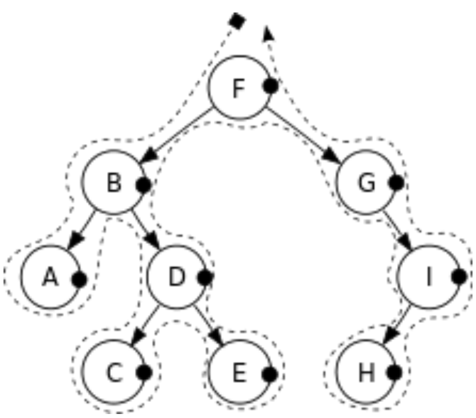
```
void InOrder(Node *cur)
{
    if (cur == NULL)      // if empty, return…
        return;

    InOrder(cur->left);     // Process nodes in left sub-tree.

    cout << cur->value;      // Process the current node.

    InOrder(cur-> right);  // Process nodes in left sub-tree.
}
```

Post-Order Traversal (touch the right side)



- process nodes in left sub-tree
- process nodes in right sub-tree
- process current node

```
void PostOrder(Node *cur)
{
    if (cur == NULL)      // if empty, return…
        return;

```

```
    PostOrder(cur->left);     // Process nodes in left sub-tree.

    PostOrder(cur-> right);  // Process nodes in right sub-tree.

    cout << cur->value;      // Process the current node.
}
```

Level Order Traversal

- visit each level's nodes, from left to right
- then visit next level

Algorithm

```
Use a temp pointer variable and a queue of node pointers.
Insert the root node pointer into the queue.

While the queue is not empty:
    Dequeue the top node pointer and put it in temp.
    Process the node.
    Add the node's children to queue if they are not NULL.
```

Binary Search Trees

- all nodes to the **left** of a node must be **less than** that node
- all nodes to the **right** must be **greater than** the node

Searching a Binary Tree

```
Start at the root of the tree
Keep going until we hit the NULL pointer
   If V is equal to current node's value, then found!
   If V is less than current node's value, go left
   If V is greater than current node's value, go right
If we hit a NULL pointer, not found.
```

- the Big-O of searching a BST is **$\log_2 n$**

Inserting a New Value into a BST

- we must place the new node so that the resulting tree is still **valid**
- big-O is $\log_2 n$

```
If the tree is empty
   Allocate a new node and put V into it
   Point the root pointer to our new node. DONE!

```

```
Start at the root of the tree

While we're not done…
p;   If V is equal to current node's value, DONE! (nothing to do…)

    If V is less than current node's value
        If there is a left child, then go left
       ELSE allocate a new node and put V into it, and
              set current node's left pointer to new node. DONE!

    If V is greater than current node's value
       If there is a right child, then go right
       ELSE allocate a new node and put V into it,
              set current node's right pointer to new node. DONE!
```

Finding min or max of a BST

- **min:** located at **left-most** node
- **max:** located at **right-most** node

Printing a Tree in Order

- just use an **in-order traversal** where the "process the current node" line is:
  - `cout << p->val << endl;`

Freeing a Whole Tree

- use a **post-order traversal** to delete children first, then delete current node

Deleting a Node from a Binary Search Tree

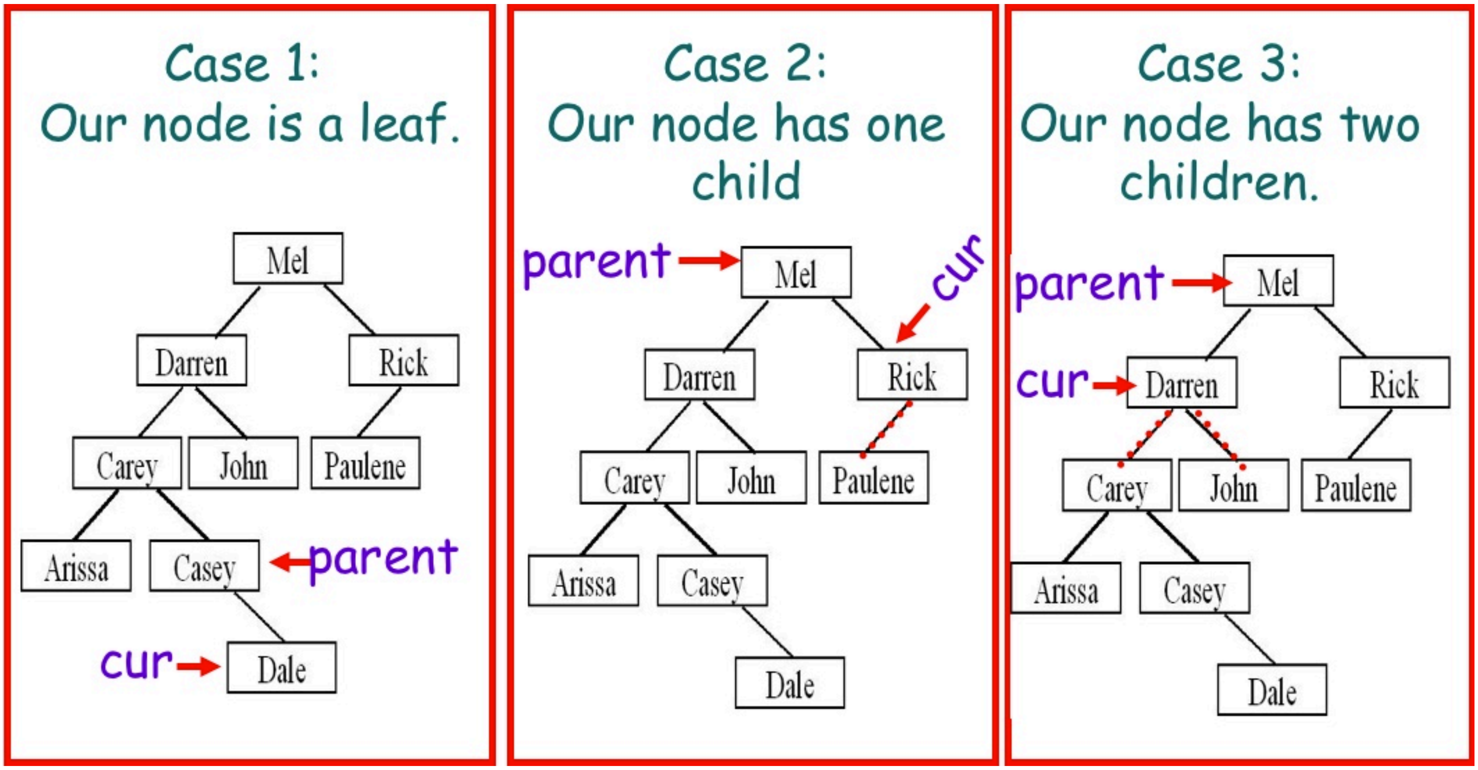
Searching for a value V to delete

- keep a parent pointer and current pointer (which you will delete)

```
parent = NULL
cur = root
While (cur != NULL)
If (V == cur->value) then we're done.
If (V < cur->value)
        parent = cur;
        cur = cur->left;
    Else if (V > cur->value)
```

```
        parent = cur;
        cur = cur->right;

// parent points at node above deletion node
// cur points at node to be deleted
```



Case 1 (no child, e.g. leaf)

Subcase 1 (target node is NOT root node)

1. unlink parent node by setting parent's appropriate link to NULL
2. delete (cur) node

Subcase 2 (target node IS root node)

1. set the root pointer to NULL
2. delete (cur) node

Above we have subcase 1

- **you would set Casey's right pointer to NULL and delete Dale.**

Case 2 (one child)

Subcase 1 (target node is NOT root node)

1. Relink **parent node** to the target (cur) node's only child
2. delete the target (cur) node

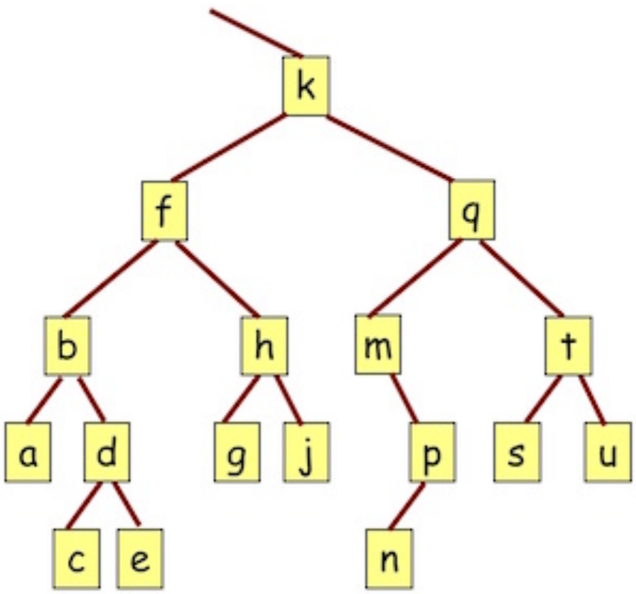
Subcase 2 (target node IS the roote node)

1. relink the **root pointer** to the target (cur) node's only child
2. delete the target (cur) node

Above we have subcase 1 (trying to delete Rick)

- **you would set Mel's right to point to Paulene then delete Rick**

Case 3 (two children)



- If we want to remove **k**, replace it with either:
  - the largest node from the left sub-tree (**j**)
    - go left then right until you can't go right anymore
  - the smallest node from the right sub-tree (**m**)
    - go right then left until you can't go left anymore
- the node we choose to replace **k** is guaranteed to have **zero** or **one** node (case 1 or 2)
  - if we replace with **j**, case 1
  - if we replace with **m**, case 2

BST's in STL

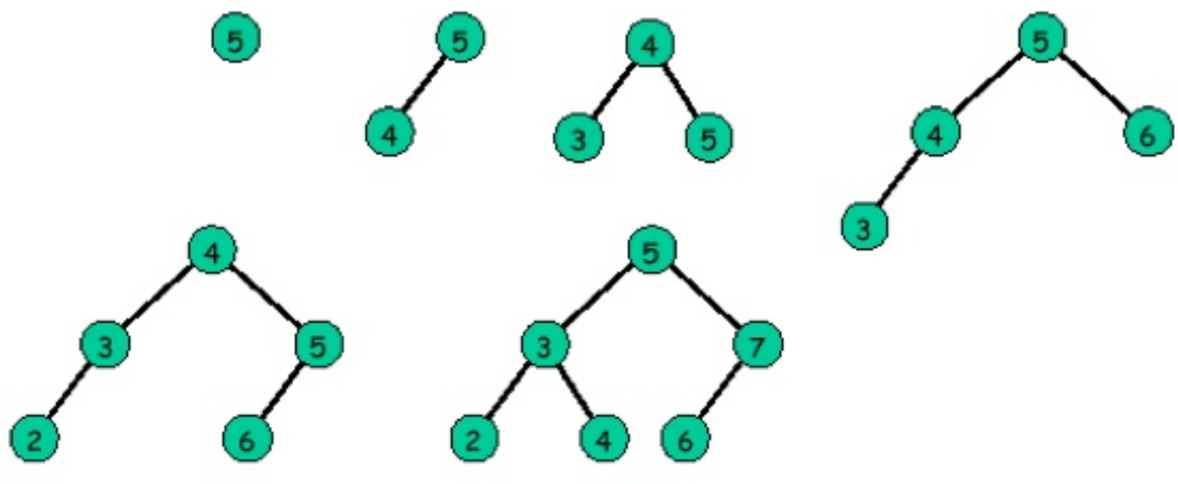
- maps and sets both use BST's to structure their data
- therefore, sets and maps always have big-O of $\mathbf{log_2 n}$ for searching, inserting, etc.

Balanced Search Trees

- If trees are inserted into in some sort of order, they will be extremely inefficient
  - All values will go on the right or left child of the bottom node, making efficiency $\mathbf{log_2 n}$

Perfectly Balanced Search Tree

- for each node, the number of nodes in its left and right subtrees differ by at most 1



- have maximum height of **$\log_2 n$** but are difficult to maintain during insertion/deletion
- there are 3 popular approaches to building a balanced binary search tree
  - AVL trees
  - 2-3 trees
  - red-black trees