Lecture 14
Binary Search Trees

EECS 281: Data Structures & Algorithms

Search

- Retrieval of a particular piece of information from large volumes of previously stored data
- Purpose is typically to access information within the item (not just the key)
- Recall that arrays, linked lists are worst-case $O(n)$ for either searching or inserting

Need a data structure with optimal efficiency for searching and inserting

Symbol Table

Definition: abstract data structure of items with keys that supports two basic operations:

- *Insert* a new item
- *Search* for an item with a given key

Symbol Table: ADT

- **`insert`** a new item
- **`search`** for an item (items) with a given key
- **`remove`** an item with a specified key
- **`sort`** the symbol table
- **`select`** the item with the k^{th} largest key
- **`join`** two symbol tables

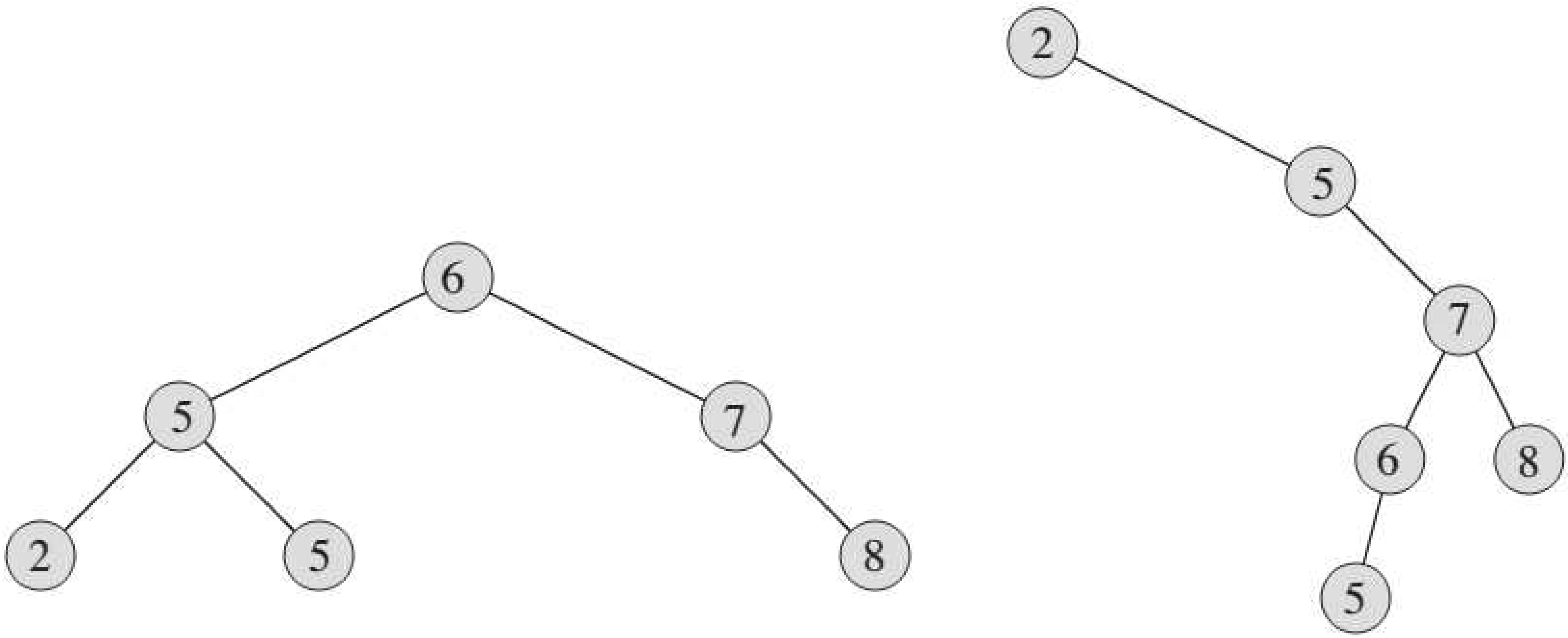
Also may want construct, test if empty, destroy, copy...

Binary Search Tree

- A binary search tree is organized as a binary tree
- The keys in a binary search tree satisfy the binary-search-tree property
  - The key of any node is:
    ≥ keys of all nodes in its left subtree and
    ≤ keys of all nodes in its right subtree
- Essential property of BST is that insert is as easy to implement as search

Binary Search Tree Property

- The key of any node is:
  ≥ keys of all nodes in its left subtree and
  ≤ keys of all nodes in its right subtree



Concrete Implementation

- Node in a Binary Tree

```
template <typename KEY>
struct Node {
    KEY key;
    Node *left, *right;
};
```

- What if we need to remove a node or add a node?
- Do we need to move up the tree?
- How would you change this implementation?

Concrete Implementation

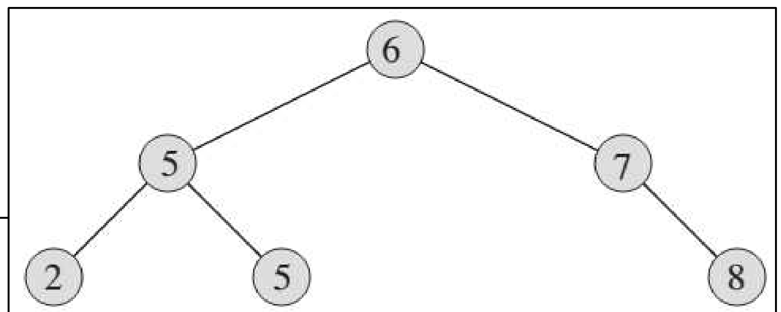
- Node in a Binary Tree (with parent)

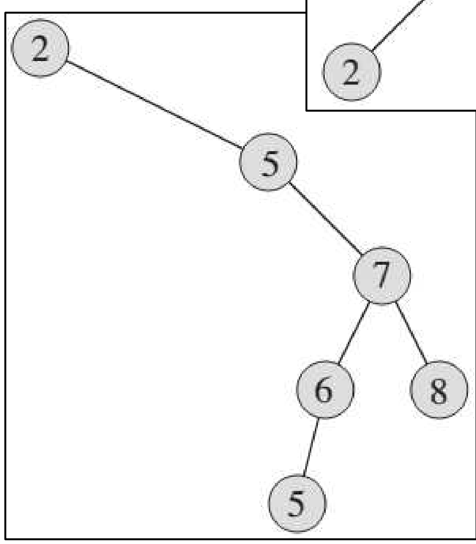
```
template <typename KEY>
struct Node {
    KEY key;
    Node *left, *right, *parent;
};
```

- Parent pointer is helpful for operations on internal nodes, for example, delete

Exercise

- Write the output for inorder, preorder and post-order traversals of these BSTs






```
void inorder(Node *x) {
  if (!x) return;
  inorder(x->left);
  print(x->key);
  inorder(x->right);
} // inorder()
void preorder(Node *x) {
  if (!x) return;
  print(x->key);
  preorder(x->left);
  preorder(x->right);
} // preorder()
void postorder(Node *x) {
  if (!x) return;
  postorder(x->left);
  postorder(x->right);
  print(x->key);
} // postorder()
```

Sort: Binary Search Tree

Can you think of an easy method of sorting using a binary search tree?

Search

- How can we find a key in a binary search tree?

```
//return a pointer to a node with key k if
//one exists; otherwise, return nullptr
Node *tree_search(Node *x, Key k);
```

- Hint: remember the key of any node is:
  ≥ keys of all nodes in its left subtree and
  ≤ keys of all nodes in its right subtree
- Bonus: what are the average and worst-case complexities?

Search

```
//return a pointer to a node with key k if
//one exists; otherwise, return nullptr
Node *tree_search(Node *x, Key k) {
    while (x != nullptr && k != x->key) {
        if (k < x->key)
            x = x->left;
        else
            x = x->right;
    } // while
    return x;
} // tree_search()
```

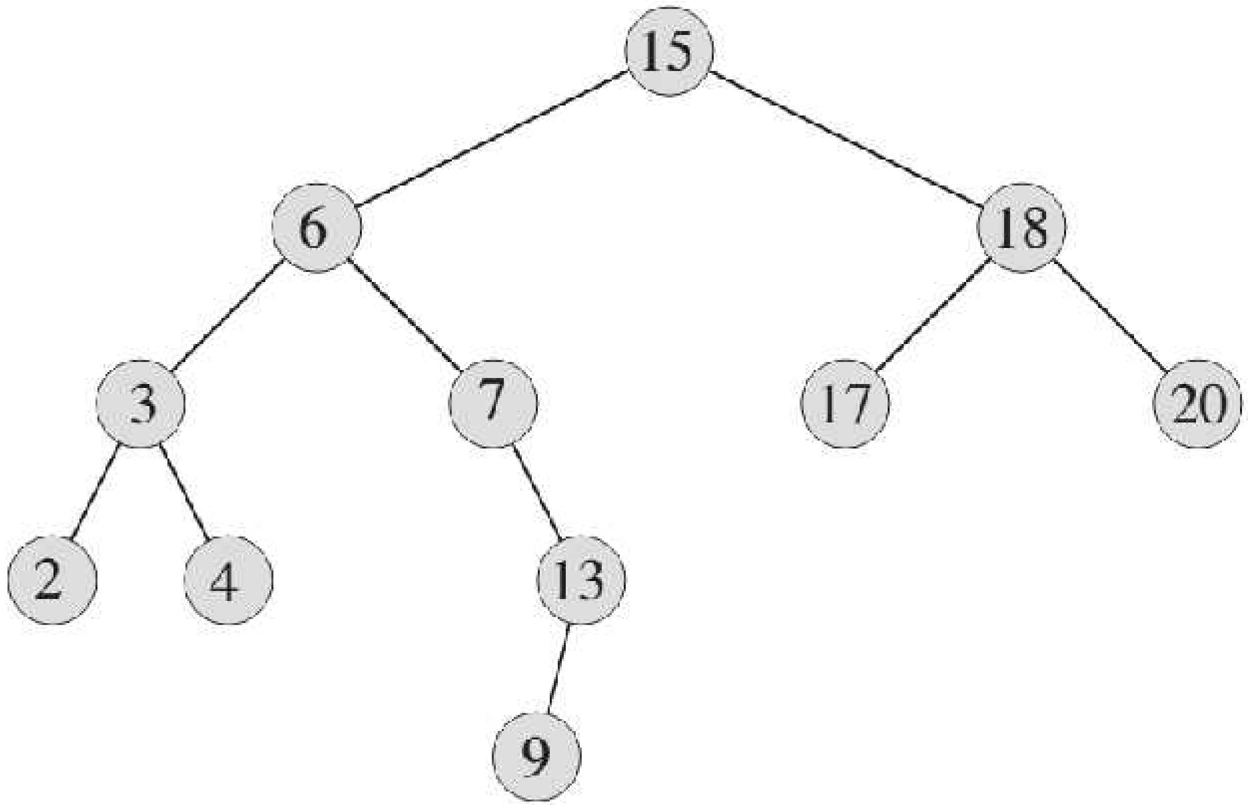
Search

```
//return a pointer to a node with key k if
//one exists; otherwise, return nullptr
Node *tree_search(Node *x, Key k) {
    if (x == nullptr || x->key == k) return x;
    if (k < x->key)
        return tree_search(x->left, k);
    return tree_search(x->right, k);
} // tree_search()
```

- Same as BST
- To search for a key k, we trace a downward path starting at the root
- The next node visited depends on the outcome of the comparison of k with the key of the current node

Search Example

```
tree_search(<ptr to 15>, 9);
```



Search

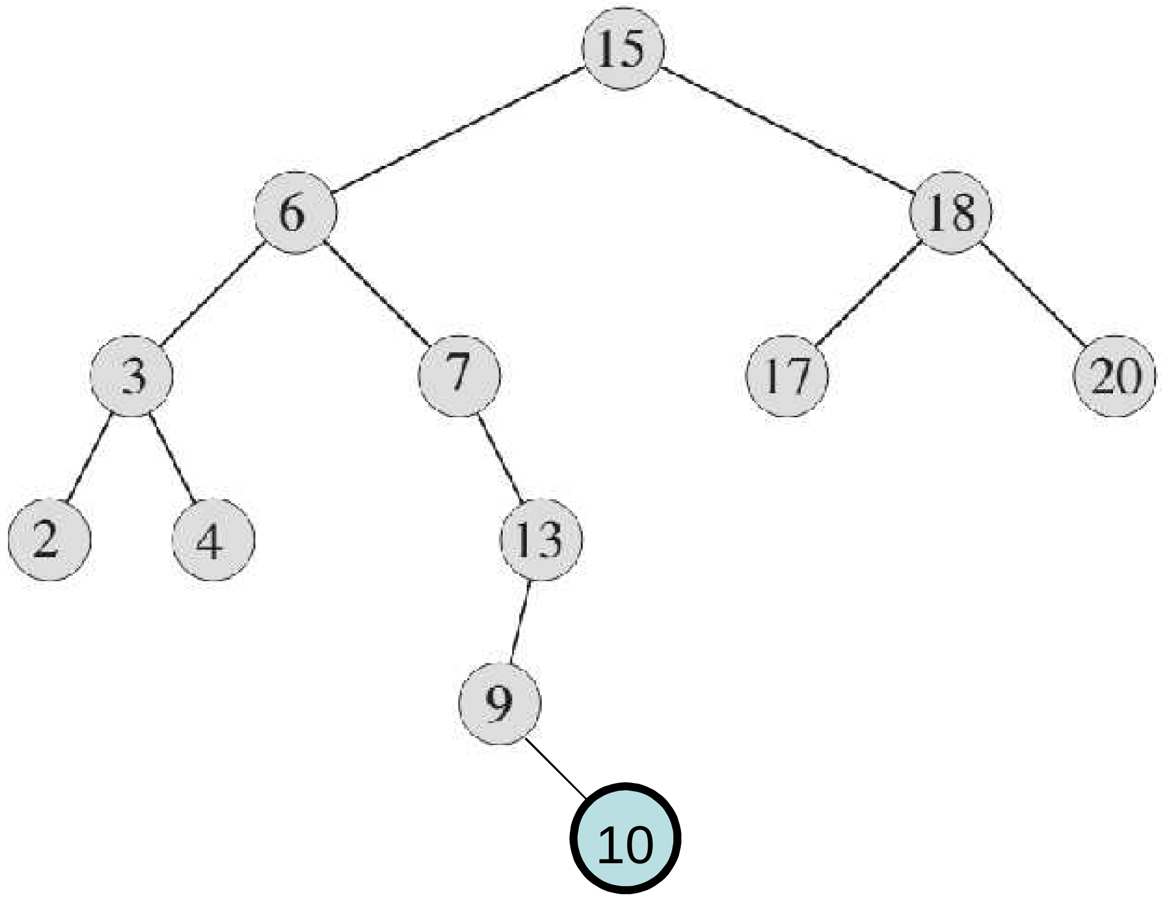
- Complexity is $O(h)$, where h is the (maximum) height of the tree
- Average case complexity: $O(\log n)$
  - Balanced tree
- Worst case complexity: $O(n)$
  - "Stick" tree

Insert

- How do we insert a new key into the tree?
- Similar to search
- Start at the root, and trace a path downwards, looking for a null pointer to append the node

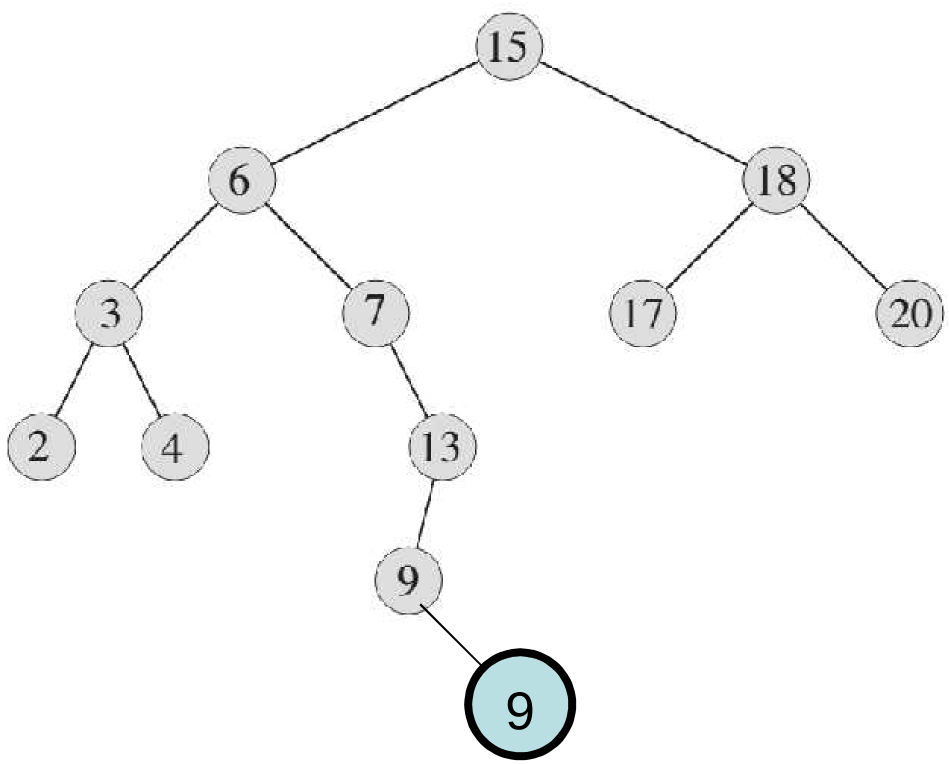
Insert Example

```
tree_insert(10);
```



Insert with Duplicates

- For sets with no duplicates, use (<, >)
- For duplicates, need deterministic policy
  - Use (<= , >) or (<, >=)



uses (<, >=)

Insert

```
void tree_insert(Node *&x, Key k) {
  if (x == nullptr) {
    x = new Node;
    x->key = k;
    x->left = x->right = nullptr;
  } // if
  else if (k < x->key)
    tree_insert(x->left, k);
  else
    tree_insert(x->right, k);
} // tree_insert()
```

- New node inserted at leaf
- Note the nifty(?) use of reference-to-pointer-to-Node
- Exercise: modify this code to set the parent pointer

Insert

```
void tree_insert(Node *&x, Node *x_parent, Key k) {
  if (x == nullptr) {
    x = new Node;
    x->key = k;
    x->left = x->right = nullptr;
    x->parent = x_parent;
  } // if
  else if (k < x->key)
    tree_insert(x->left, x, k);
  else
    tree_insert(x->right, x, k);
} // tree_insert()
```

- This version sets the parent pointer

Insert [CLRS]

```
void tree_insert_CLRS(Node *&t, Key k) {
  Node *y = nullptr;
  Node *x = t;
  while (x != nullptr) { //find location for new node
    y = x;
    x = (k < x->key) ? x->left : x->right;
  } // while
  Node *z = new Node(k, y); //y is parent of new node

  if (y == nullptr)
    t = z; //tree t was empty
  else if (z->key < y->key) y->left = z;
  else y->right = z;
} // tree_insert()
```

Exercise

- Start with an empty tree
- Insert these keys, in this order:

  12, 5, 18, 2, 9, 15, 19, 17, 13
- Draw the tree
- Write a new order to insert the same keys which generates a worst-case tree
- How many worst-case case trees are possible for *n* nodes?

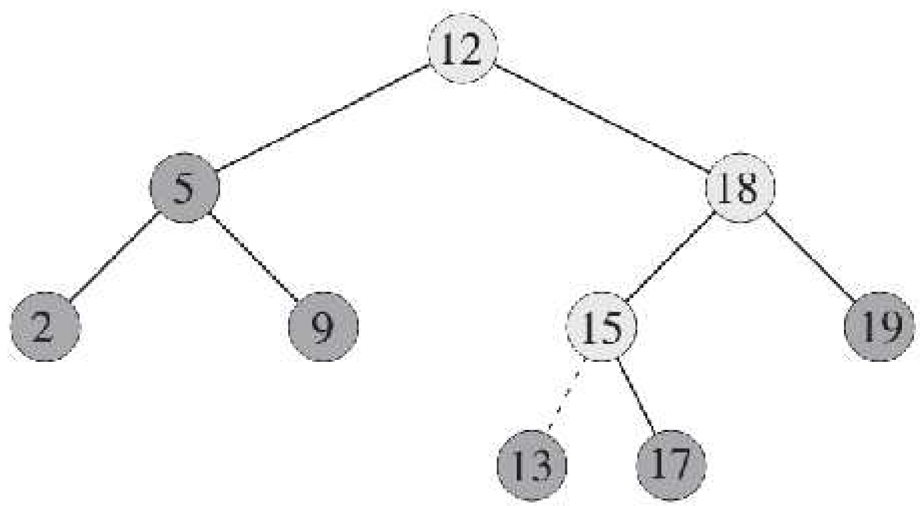
Complexity

- The complexity of insert (and many other tree functions) depends on the height of the tree
- Average case (balanced): $O(\log n)$
- Worst case (unbalanced "stick"): $O(n)$
- Average case:
  – Random data
  – Likely to be well-balanced

Exercise

- Write a function to find the Node with the smallest key
- What are the average and worst-case complexities?

```
//returns a pointer to the Node with the min key
Node *tree_min(Node *x);
```



Exercise

```
//return a pointer to the min key node
Node *tree_min(Node *x) {
  if (x == nullptr)
    return nullptr;
  while (x->left)
    x = x->left;
  return x;
} // tree_min()
```

- Average case complexity: $O(\log n)$
- Worst case complexity: $O(n)$

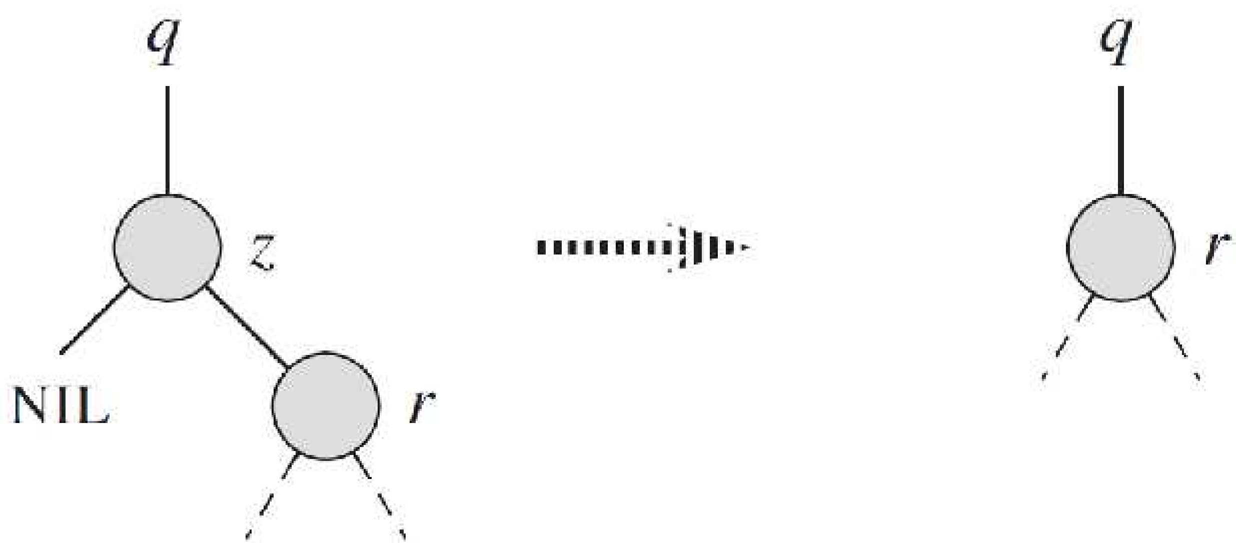
Delete

- What if we want to delete a node?
- To delete node *z*:
  1. *z* has no children (trivial)
  2. *z* has no left child
  3. *z* has no right child
  4. *z* has two children

- Complete algorithm is in CLRS 12.3

Delete

- *z* has no left child: replace *z* by right child



Delete

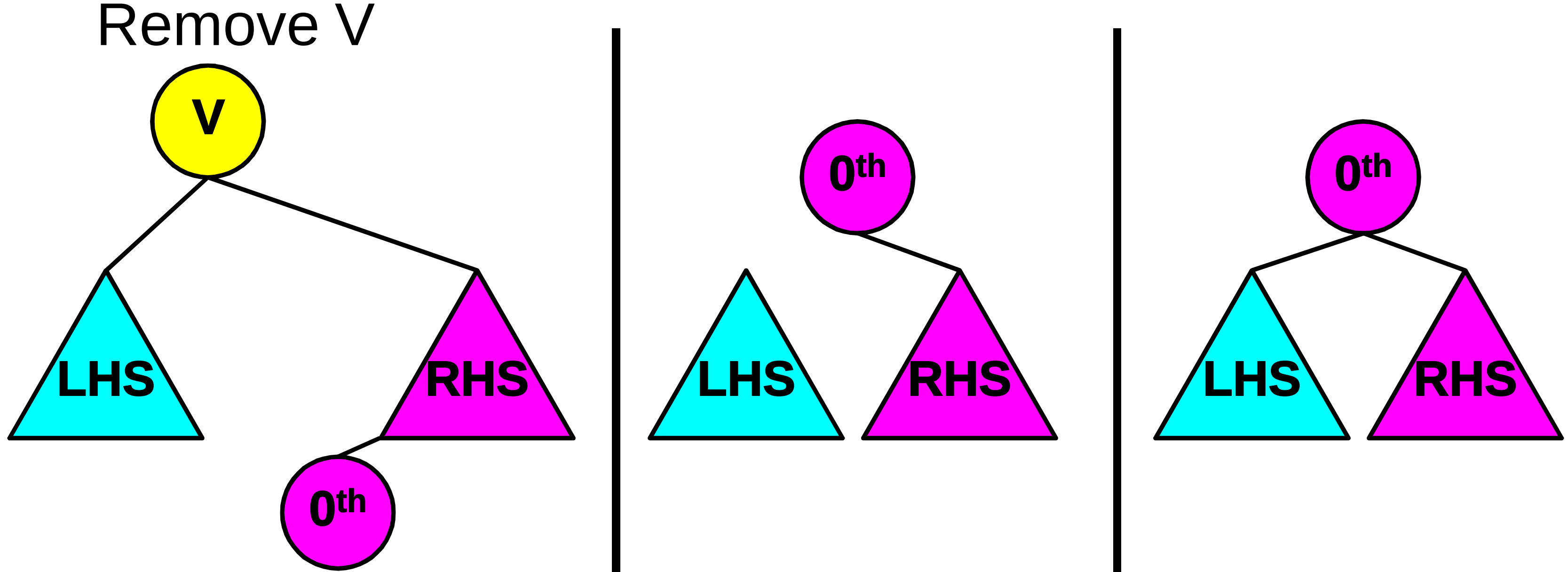
- *z* has no right child: replace *z* by left child

Delete

- *z* has left and right children
- Replace with a “combined” tree of both
- Key observation
  - All in LHS subtree ≤ all in RHS subtree
  - Transplant smallest RHS node to root
    - Called the inorder successor
    - Must be some such node, since RHS is not empty
    - New root might have a right child, but no left child
  - Make new root’s left child the LHS subtree

Delete

Remove V



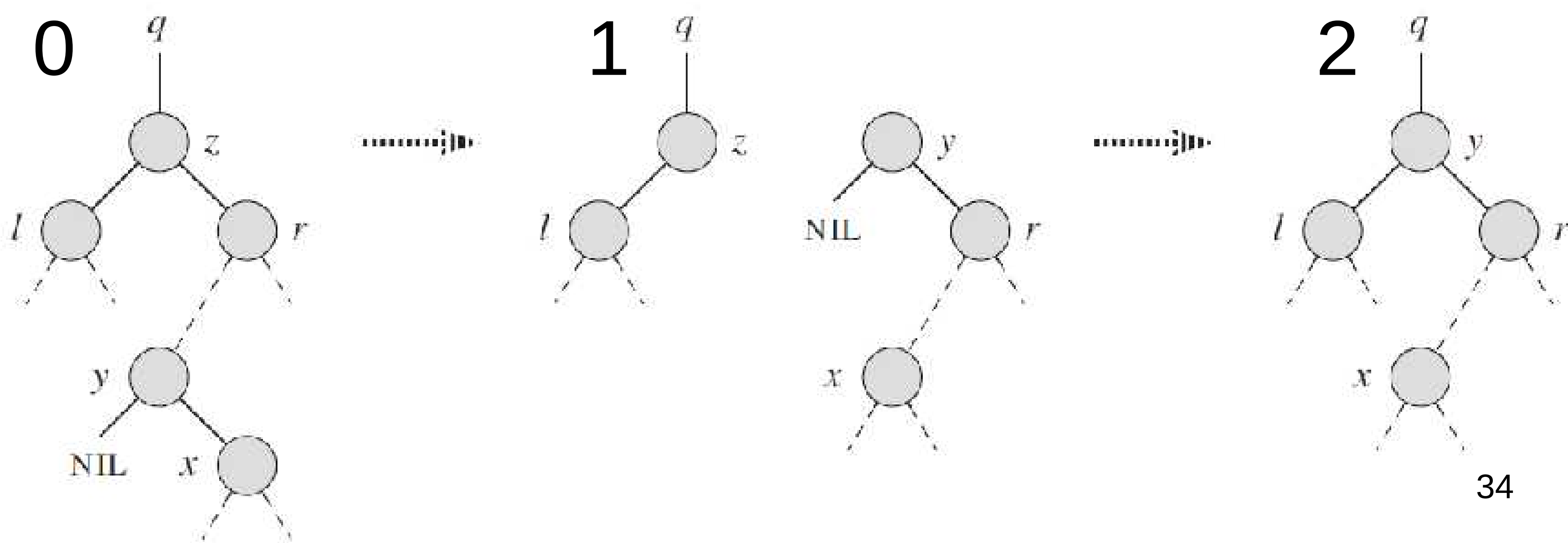
Find smallest node in RHS

Transplant smallest RHS node to root

Combine two trees

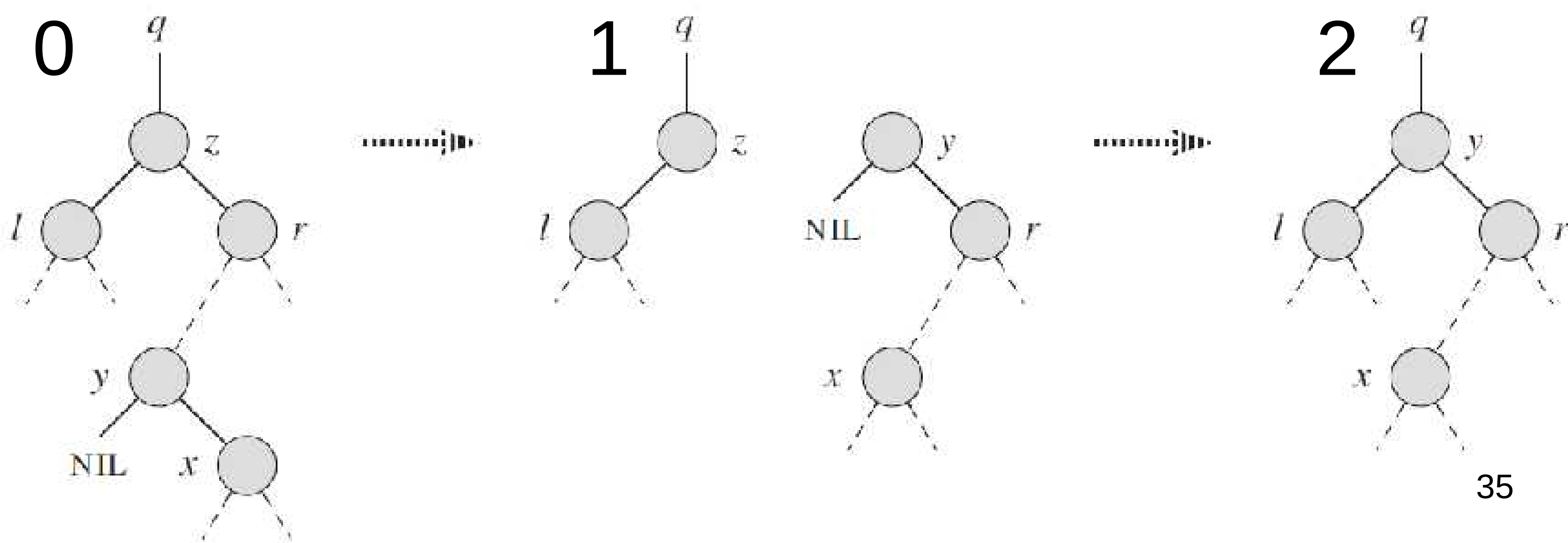
Delete

1. Transplant smallest RHS node to root
   – Must be some such node, since RHS is not empty
   – New root might have a right child, but no left child
2. Make new root's left child the LHS subtree



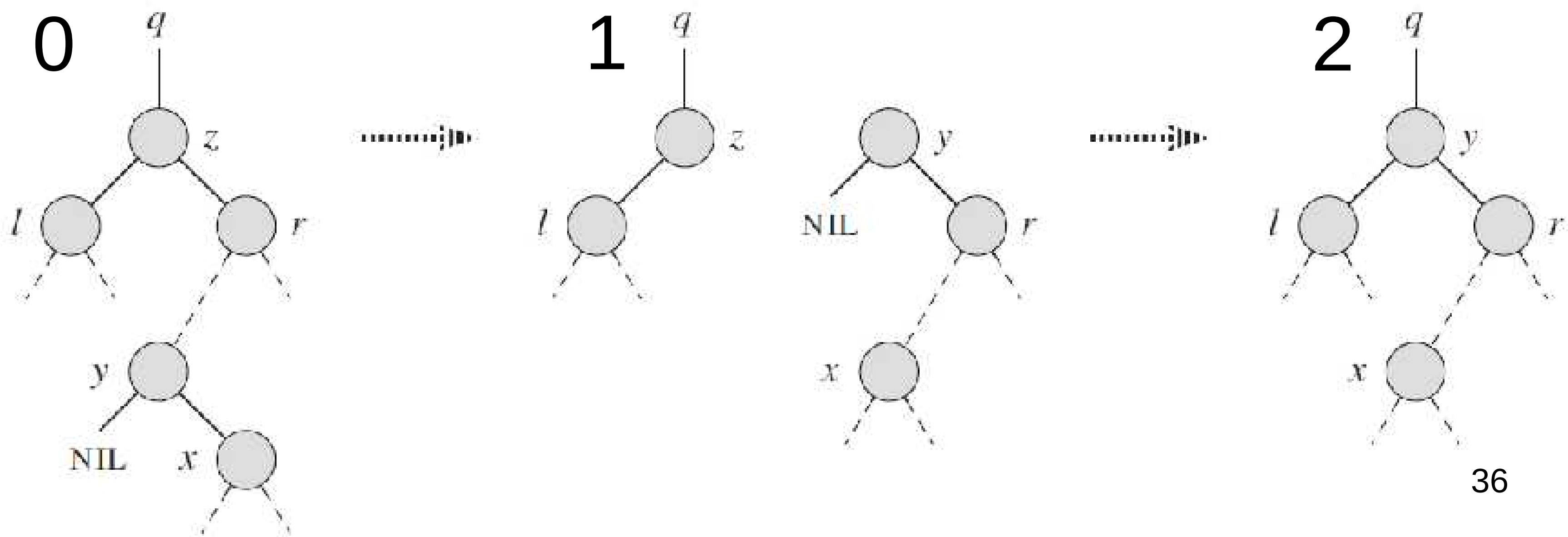
Delete

- This is where parent pointers are especially helpful: the "transplant" is $O(1)$
- What is the complexity of finding the smallest node in the right subtree?



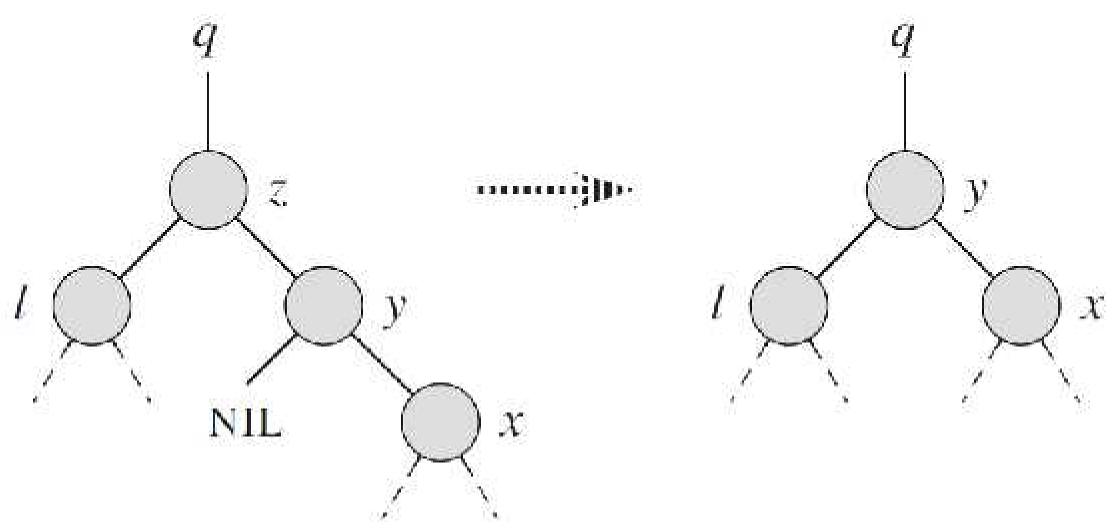
Delete

- This is where parent pointers are especially helpful: the "transplant" is $O(1)$
- What is the complexity of finding the smallest node in the right subtree?
  - $O(n)$ worst case, $O(\log n)$ average case, which dominates the complexity of delete



Delete

- Easier case: *z*'s right child has no left child



Delete Helper Function [CLRS]

```
void tree_transplant(Node *&t, Node *u, Node *v) {
  if (u->parent == nullptr)
    t = v;
  else if (u == u->parent->left)
    u->parent->left = v;
  else
    u->parent->right = v;

  if (v != nullptr)
    v->parent = u->parent;
  delete u;
} // tree_transplant()
```

Delete [CLRS]

```
void tree_delete(Node *&t, Node *z) {
  if (z->left == nullptr)
    tree_transplant(t, z, z->right);
  else if (z->right == nullptr)
    tree_transplant(t, z, z->left);
  else {
    Node *y = tree_min(z->right);
    if (y->parent != z) {
      tree_transplant(t, y, y->right); // Don't delete!
      y->right = z->right;
      y->right->parent = y;
    } // if
    tree_transplant(t,z,y);
    y->left = z->left;
    y->left->parent = y;
  } // else
} // tree_delete()
```

Single-Function Delete 1/3

```
template <typename T>
void BinaryTree<T>::remove(Node *&tree, const T &val)
{
    Node *nodeToDelete = tree;
    Node *inorderSuccessor;

    // Recursively find the node containing the value to delete
    if (tree == nullptr)
        return;
    else if (val < tree->value)
        remove(tree->left, val);
    else if (tree->value < val)
        remove(tree->right, val);
    else {
```

Single-Function Delete 2/3

```
        // Check for simple cases where one subtree is empty
        if (tree->right == nullptr)
        {
            tree = tree->left;
            delete nodeToDelete;
        } // if
        else if (tree->left == nullptr)
        {
            tree = tree->right;
            delete nodeToDelete;
        } // else if
```

Single-Function Delete 3/3

```
        else {
            // Node to delete has both left and right subtrees
            inorderSuccessor = tree->right;

            while (inorderSuccessor->left != nullptr)
                inorderSuccessor = inorderSuccessor->left;

            // Replace value with one from inorder successor
            nodeToDelete->value = inorderSuccessor->value;
            // Remove the inorder successor from right subtree
            remove(tree->right, inorderSuccessor->value);
        } // else
    } // else
} // BinaryTree::remove()
```

Summary: Binary Search Trees

- Each node points to two children (left, right), and possibly a parent
- All nodes are ordered: left ≤ root ≤ right
- Modification of nodes
  - External is easy
  - Internal is more complicated
- In general, operations on BSTs are:
  - $O(\log n)$ average
  - $O(n)$ worst case