CS 261 – Data Structures

Binary Search Trees
Binary Search Tree

• Binary search trees are binary trees where every node’s object value is:
  – *Greater than* all its descendents in the *left subtree*
  – *Less than or equal to* all its descendents in the *right subtree*

• In-order traversal returns elements in sorted order

• If tree is reasonably full (*well balanced*), searching for an element is $O(\log n)$
Binary Search Tree: **Example**

- **Alex**
  - **Abner**
    - **Abigail**
    - **Adam**
  - **Adela**
    - **Agnes**
  - **Angela**
    - **Alice**
    - **Allen**
    - **Arthur**
Binary Search Tree (BST): Implementation

```c
struct BSTree {
    struct Node *root;
    int          cnt;
};

void    initBSTree(struct BSTree *tree);
void     addBSTree(struct BSTree *tree, TYPE val);
int containsBSTree(struct BSTree *tree, TYPE val);
void  removeBSTree(struct BSTree *tree, TYPE val);
int     sizeBSTree(struct BSTree *tree);
```
Binary Node: Reminder

```c
struct Node {
    TYPE      val;
    struct Node *left;  /* Left child. */
    struct Node *right; /* Right child. */
};
```
Bag Implementation: **Contains**

- Start at root

- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false

- Traverses a path from the root to the leaf

- Therefore, if tree is *reasonably full* (an important if), execution time is $O(??)$
Before first call to `add`
Useful Trick

A useful trick (adapted from the functional programming world):
Secondary routine that returns tree with the value inserted

Node addNode(Node current, TYPE value)
  if current is null then return new Node with value
  otherwise if value < Node.value
    left child = addNode(left child, value)
  else
    right child = addNode(right child, value)
return current node
Add: Calls Utility Routine

```c
void add(struct BSTree *tree, TYPE val) {
    tree->root = _addNode(tree->root, val);
    tree->cnt++;
}
```

What is the complexity? O( ?? )
How would you remove Abigail? Audrey? Angela?
Who fills the hole?

• Answer: the leftmost child of the right child (smallest element in right subtree)

• Try this on a few values

• Useful to have a couple of private inner routines:

```c
TYPE _leftmost(struct Node *cur) {
    ... /* Return value of leftmost child of current node. */
}

struct Node *_removeLeftmost(struct Node *cur) {
    ... /* Return tree with leftmost child removed. */
}
```
BST: Remove Example

Before call to remove

Replace with: \texttt{leftmost(rght)}
BST: Remove Example

After call to remove
Special Case

• What if you don’t have a right child?

• Try removing “Audrey”
  – Think about it
  – Can just return left child
Remove: Easy if you return a tree

```java
Node removeNode(Node current, TYPE value)
    if value = current.value
        if right child is null
            return left child
        else
            replace value with leftmost child of right child
            set right child to be removeLeftmost(right)
    else if value < current.value
        left child = removeNode(left child, value)
    else right child = removeNode(right child, value)
    return current node
```
Complexity

• Running down a path from root to leaf

• What is the complexity? $O(??)$

• Questions?? Now the worksheet