CS261 Data Structures

Binary Search Trees II

Bag Implementation

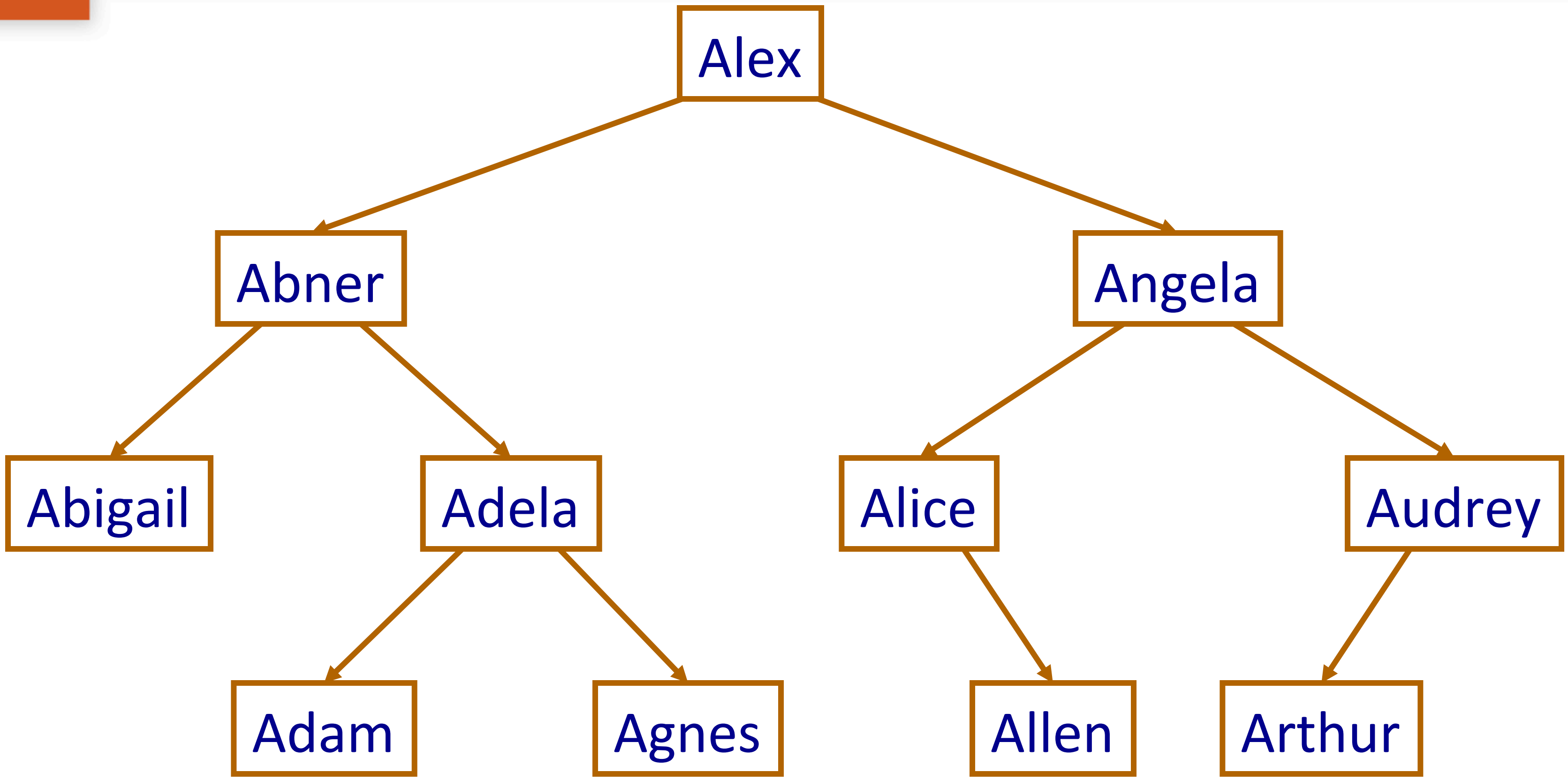
Goals

- BST Representation
- Bag Operations
- Functional-style operations

Binary Search Tree (review)

- Binary search trees are binary trees where every nodes 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 (review)



BST Bag

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

```
struct Node {
  TYPE        val;
  struct Node *left
  struct Node *rght
};
```

```
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);
```

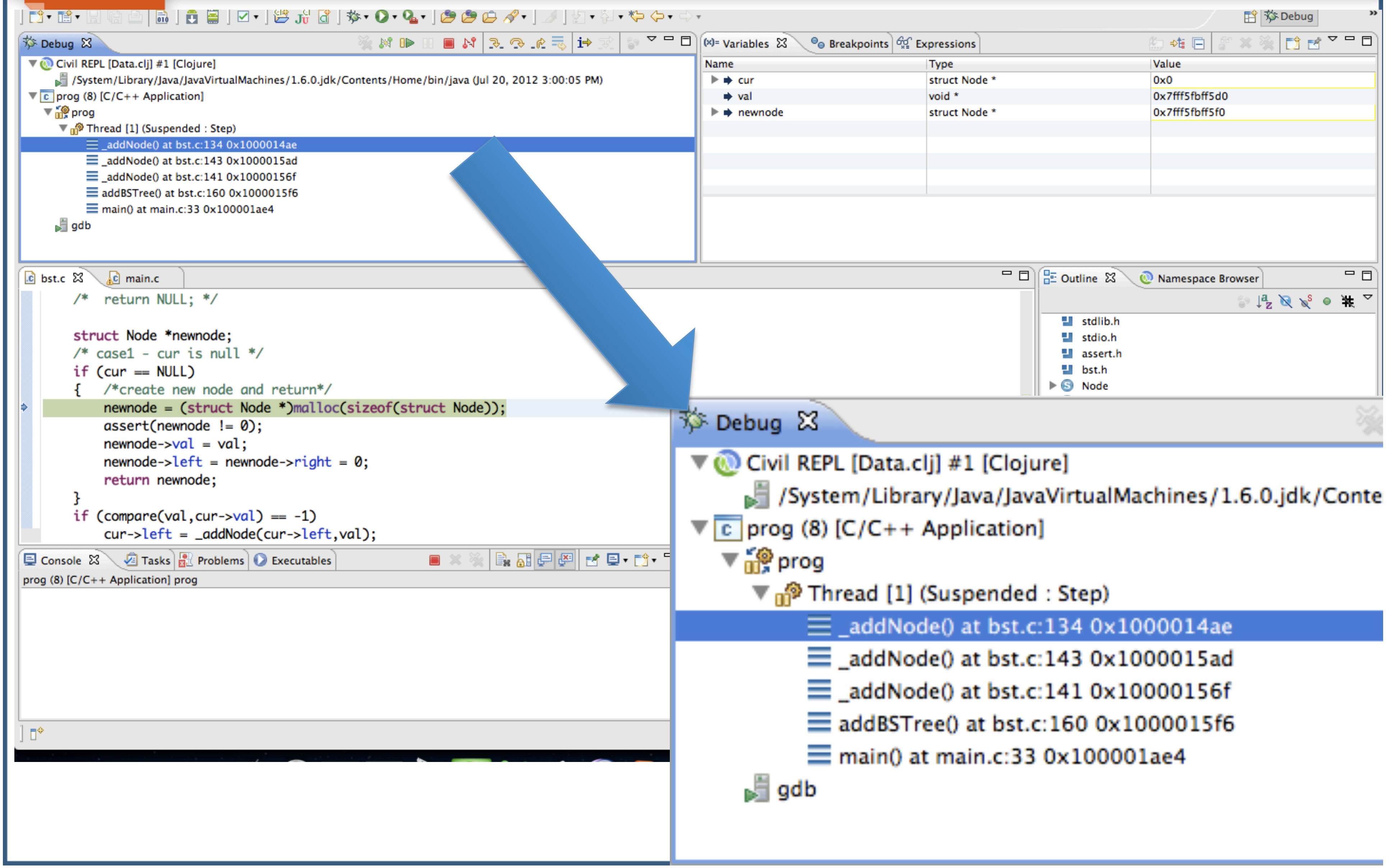
Functional Approach

A useful trick (adapted from the functional programming world): Recursive helper 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
```

Visual Example

Process Stack



BST Add: public facing add

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

Recursive Helper – functional flavor

```
struct Node *_addNode(struct Node *cur, TYPE val){
  struct Node *newnode;
  if (cur == NULL){
      newnode = malloc(sizeof(struct Node));
      assert(newnode != 0);
      newnode->val = val;
      newnode->left = newnode->right = 0;
      return newnode;
  }
   if (val < cur->val)
        cur->left = _addNode(cur->left,val);
   else cur->right = _addNode(cur->right, val);
  return cur;
}
```

Python Functional Version

```
def binary_tree_insert(node, key, value):
  if node is None:
      return TreeNode(None, key, value, None)
  if key == node.key:
      return TreeNode(node.left, key, value, node.right)
  if key < node.key:
      return TreeNode(binary_tree_insert(node.left, key,
             value), node.key, node.value, node.right)
    else:
      return TreeNode(node.left, node.key, node.value,
             binary_tree_insert(node.right, key, value))
```

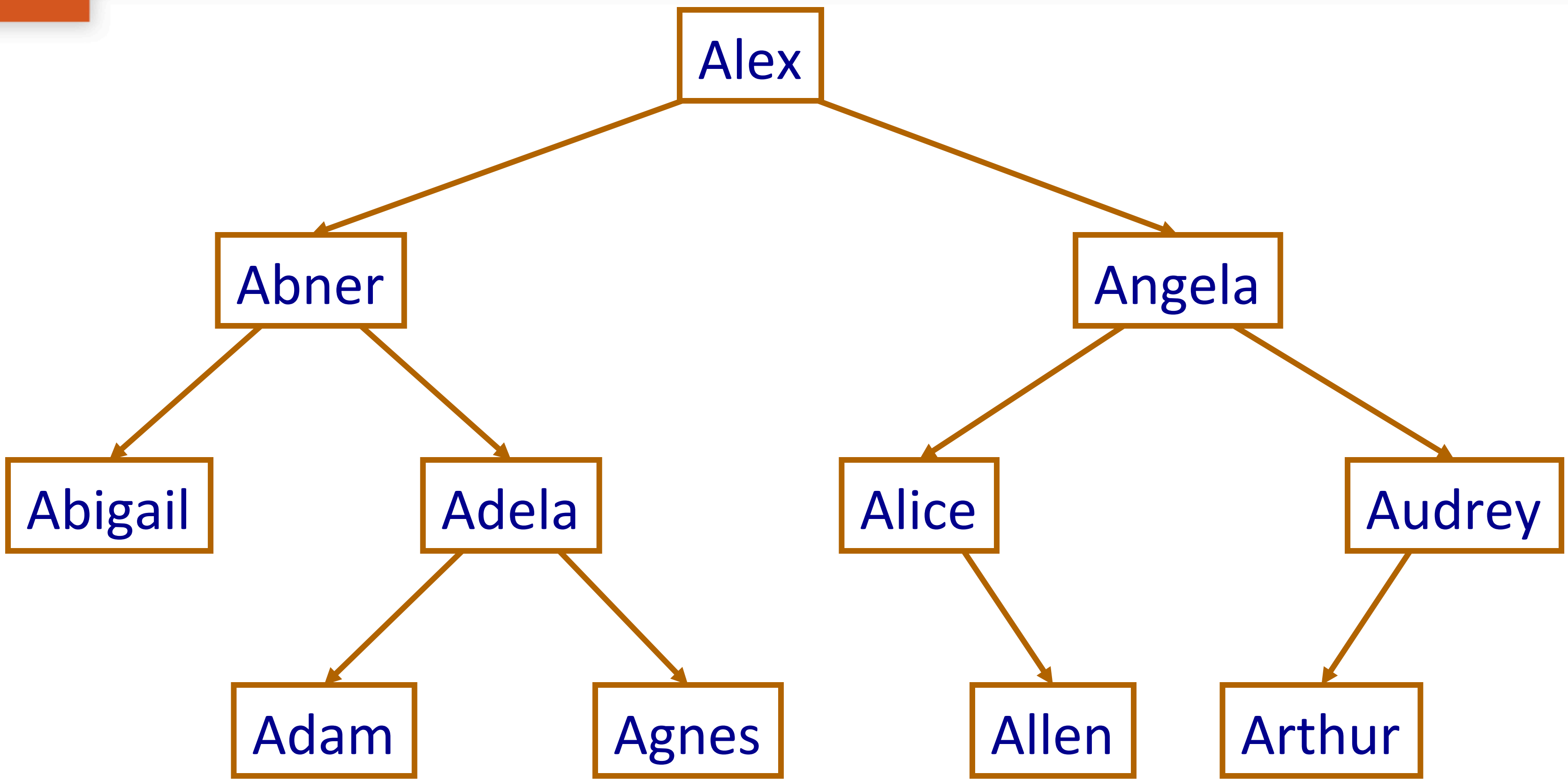
Iterative Flavor - Java

```
public void insert(int data) {
    if (m_root == null) {
        m_root = new TreeNode(data, null, null);
        return;
    }
    Node root = m_root;
    while (root != null) {
      if (data == root.getData()) {
            return;
        } else if (data < root.getData()) {
            if (root.getLeft() == null) {
                root.setLeft(new TreeNode(data, null,
                                                  null));
                return;
            } else {
                root = root.getLeft();
            }
        }
```

Iterative (Java)

```
} else {
            if (root.getRight() == null) {
                root.setRight(new TreeNode(data, null,
                                                 null));
                return;
            } else {
                root = root.getRight();
            }
        }
    }
}
```

BST Remove



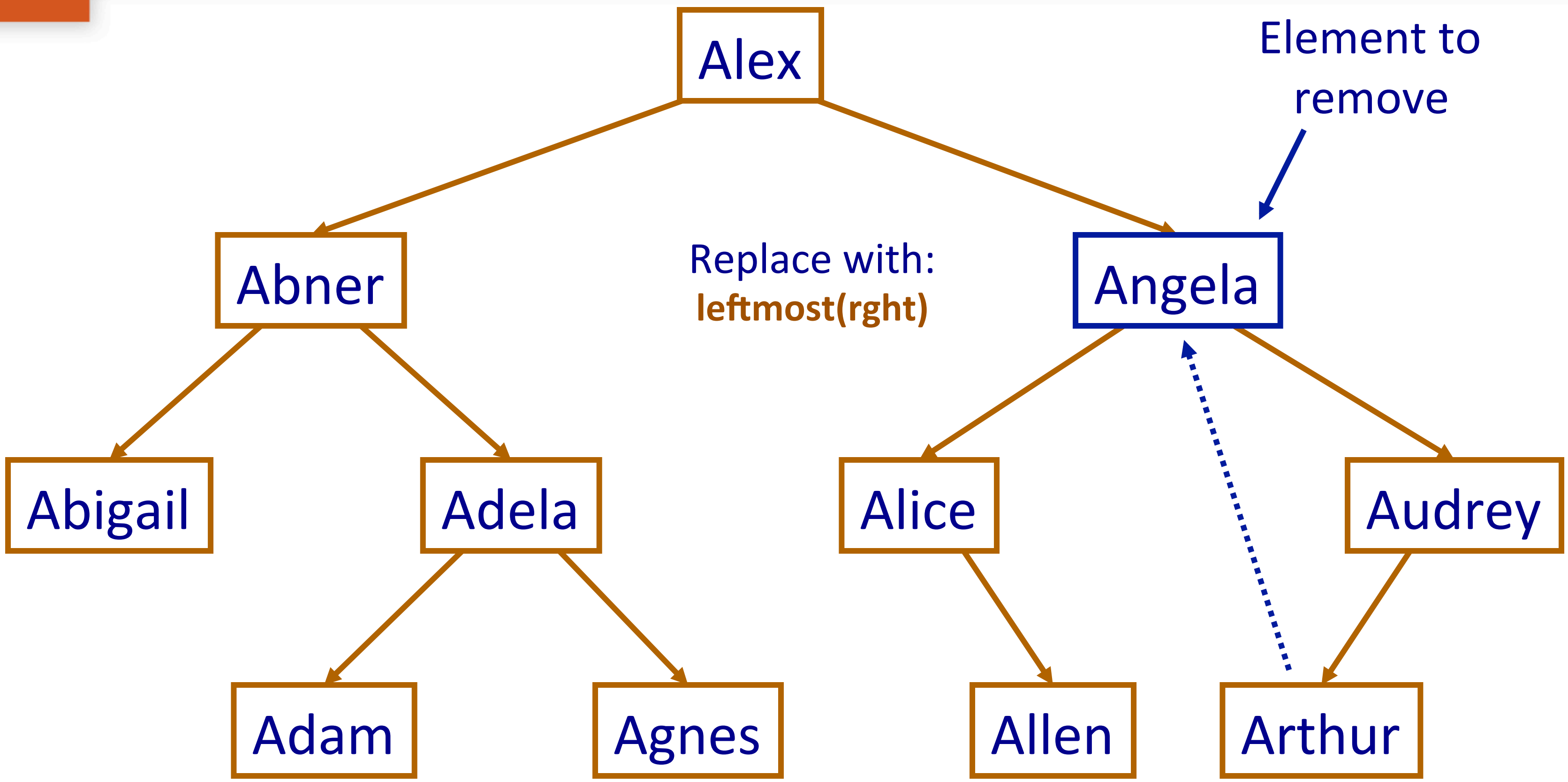
How would you remove Abigail? Audrey? Angela?

Who fills the hole?

- Answer: the leftmost child of the right child
  (smallest element in right subtree)
- Useful to have a couple of private inner routines:

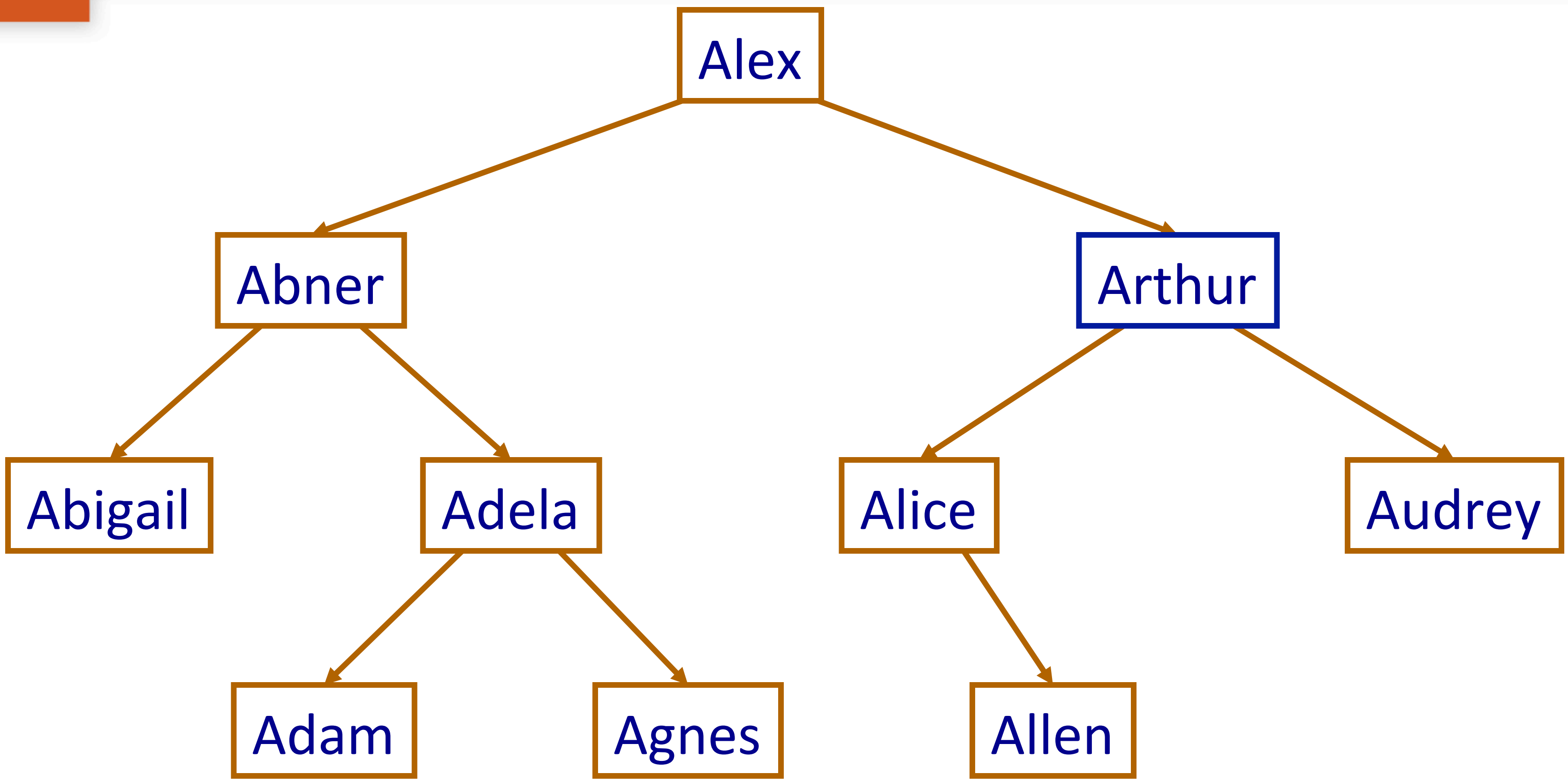
```
TYPE _leftmost(struct Node *cur) {
  ... /* Return value of leftmost child of current node. */
}

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

BST Remove Example



Before call to **remove**

BST Remove Example



After call to **remove**

BST Remove Pseudocode

```
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 subtree
            set right child to result of removeLeftmost(right)
    else if value < current.value
                left child = removeNode(left child, value)
        else right child = removeNode(right child, value)
    return current node
```

Comparison

- Average Case Execution Times

Operation	DynArrBag	LLBag	Ordered ArrBag	BST Bag
Add	O(1+)	O(1)	O(n)	O(logn)
Contains	O(n)	O(n)	O(logn)	O(logn)
Remove	O(n)	O(n)	O(n)	O(logn)

Space Requirements

- Does the functional-style recursive version require more or less space than an iterative version? Think about the call stack?
  - rebuilding as move up from recursion requires O(logn) space on average

Your Turn

- Complete the BST implementation in Worksheet #29