CS 261: Data Structures

Binary Tree Traversals

Binary Search Trees

Binary Tree Traversals

- How to examine nodes in a tree?
- A list is a simpler structure:
  - can be traversed either forward or backward
- What order do we visit nodes in a tree?
- Most common tree traversal orders:
  - Pre-order
  - In-order
  - Post-order

Binary Tree Traversals

- All traversal algorithms have to:
  - Process node
  - Process left subtree
  - Process right subtree

Traversal order is determined by the order these operations are done

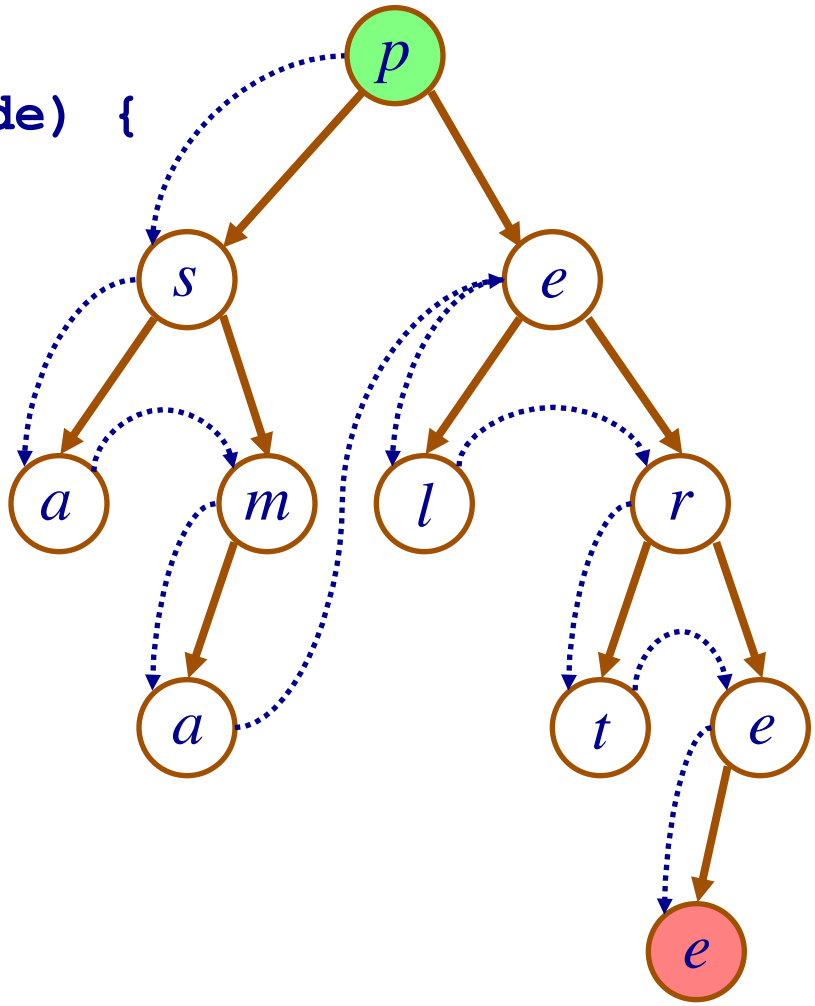
Binary Tree Traversals – Most Common

1. Node, left, right → Pre-order
2. Left, node, right → In-order
3. Left, right, node → Post-order

Pre-Order Binary Tree Traversal

- Process order → Node, Left subtree, Right subtree

```
void preorder(struct Node *node) {
  if (node != 0){
    process (node->val);
    preorder(node->left);
    preorder(node->rght);
  }
}
```

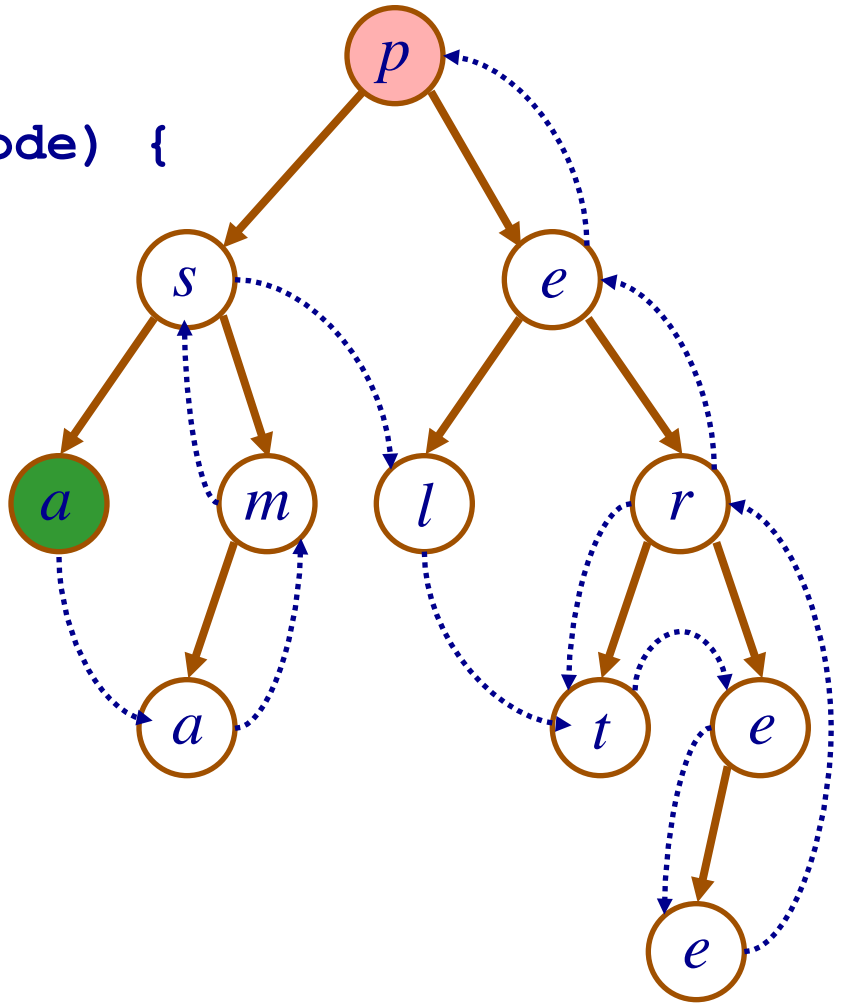


Example result: p s a m a e l r t e e

Post-Order Binary Tree Traversal

- Process order → Left subtree, Right subtree, Node

```
void postorder(struct Node *node) {
  if (node != 0){
    postorder(node->left);
    postorder(node->rght);
    process(node->val);
  }
}
```

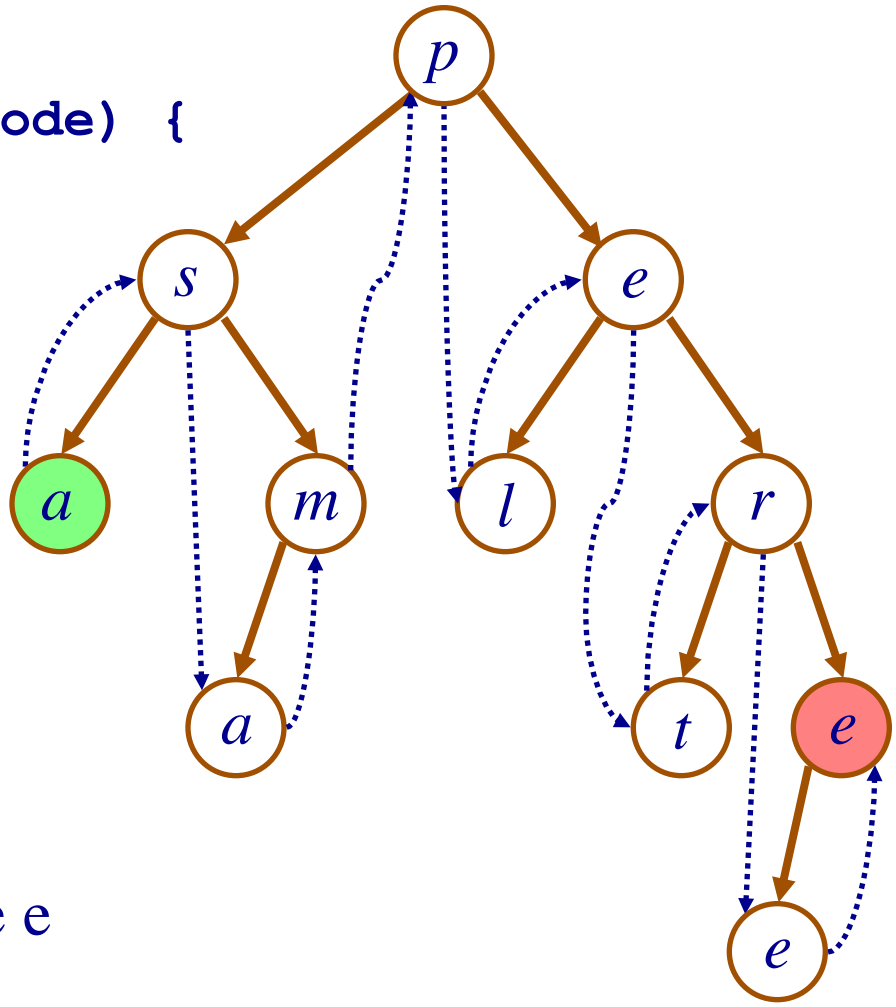


Example result: a a m s l t e e r e p

In-Order Binary Tree Traversal

- Process order → Left subtree, Node, Right subtree

```
void inorder(struct Node *node) {
  if (node != 0){
    inorder(node->left);
    process(node->val);
    inorder(node->rght);
  }
}
```



Example result: a s a m p l e t r e e

Complexity of Binary Tree Traversals

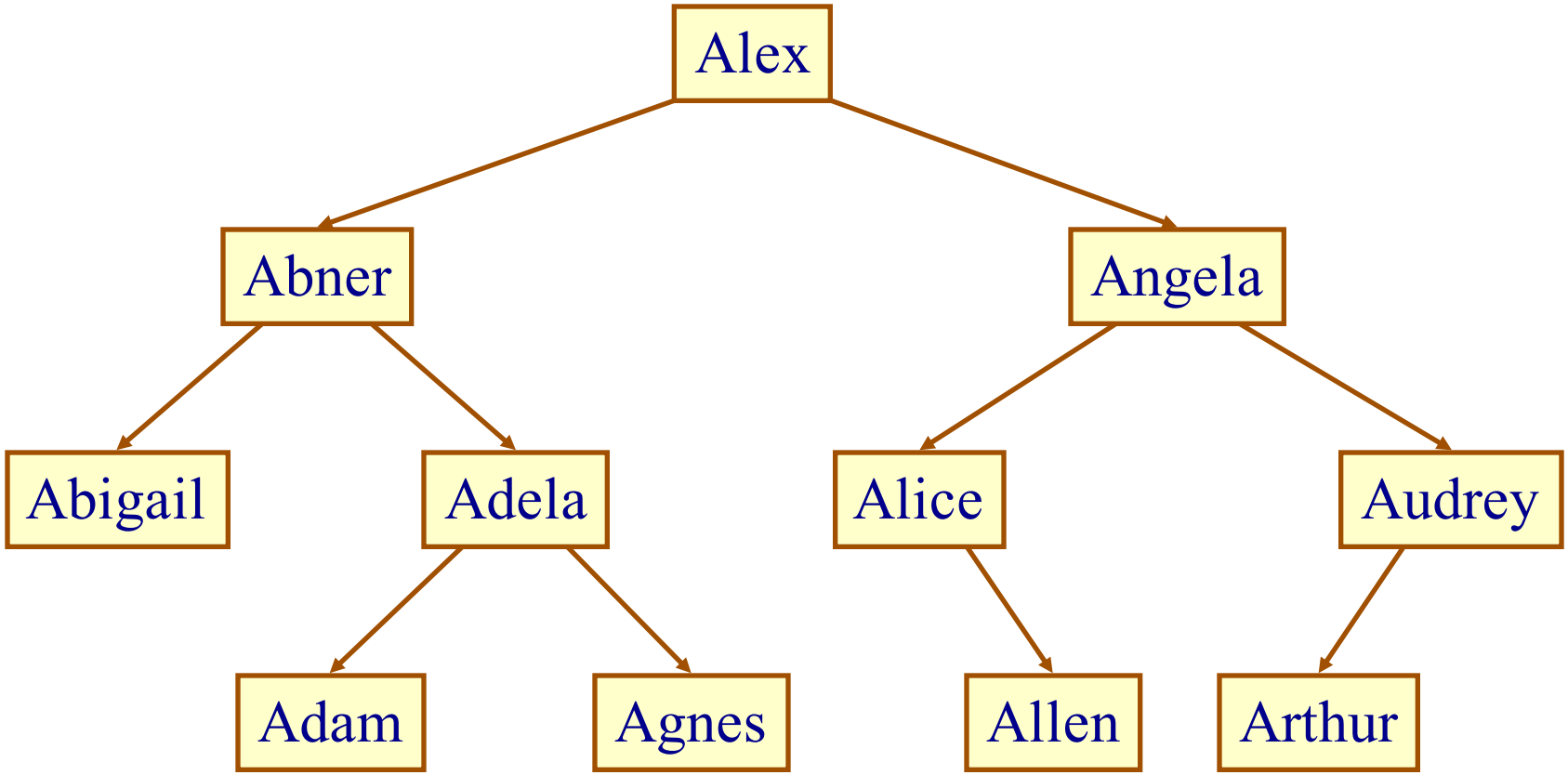
- Each traversal requires constant work at each node (not including recursive calls)
- Order not important
- Iterating over all n elements in a tree requires $O(n)$ time

Binary Search Trees

Binary Search Tree

- = Binary trees where every node value is:
  - **Greater than** all its **left** descendants
  - **Less than or equal to** all its **right** descendants
  - 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



A Node in BST

```
struct Node {
   TYPE   val;
   struct Node *left;    /* Left  child */
   struct Node *right;   /* Right child */
};
```

Binary Search Tree (BST): Implementation

```
struct BST {
  struct Node *root;
  int size;
};


void initBST(struct BST *tree);
int containsBST(struct BST *tree, TYPE val);
void addBST(struct BST *tree, TYPE val);
void removeBST(struct BST *tree, TYPE val);
int  sizeBST(struct BST *tree);
```

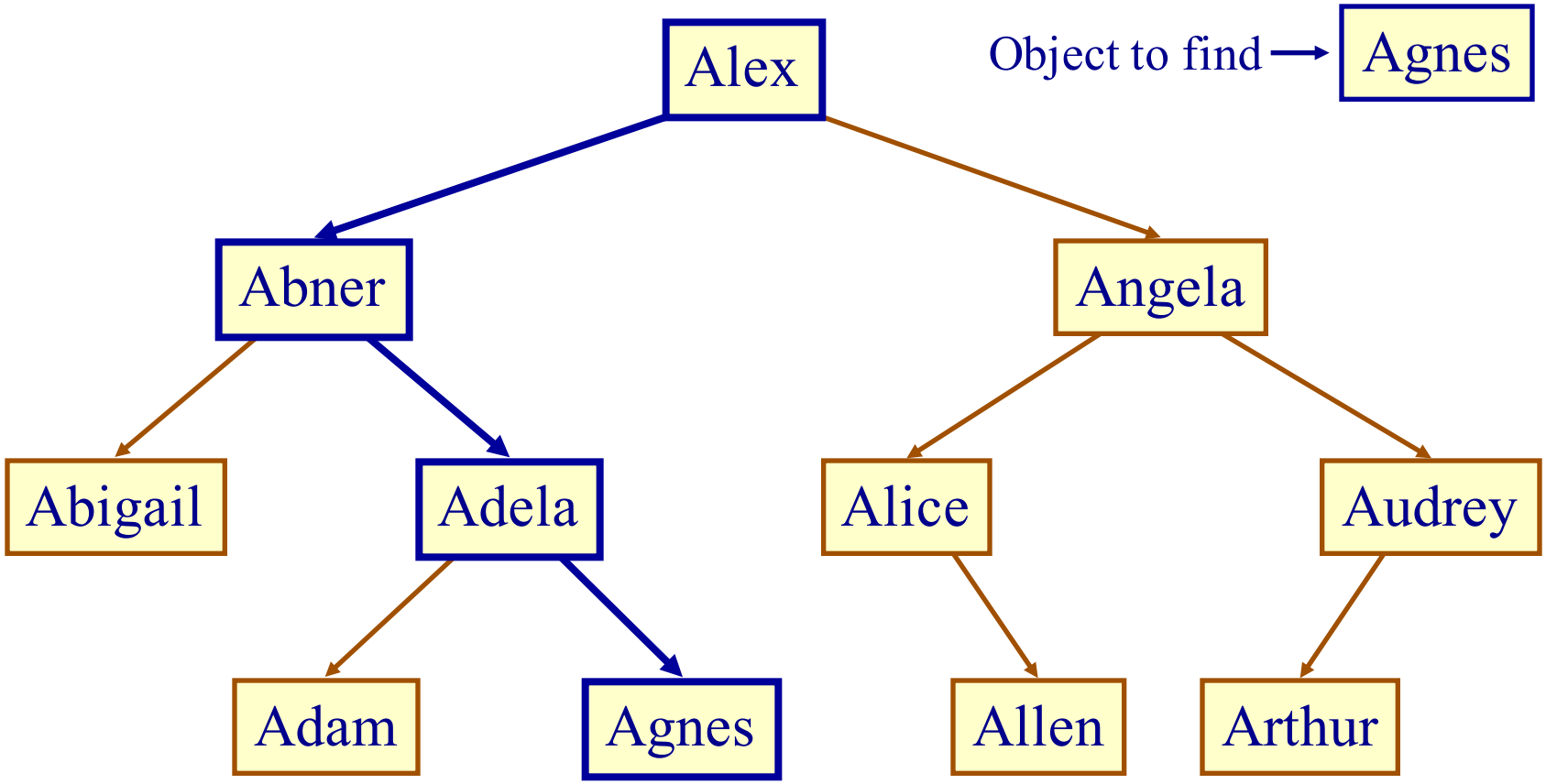
Sorted Bag Implementation: Contains

- 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

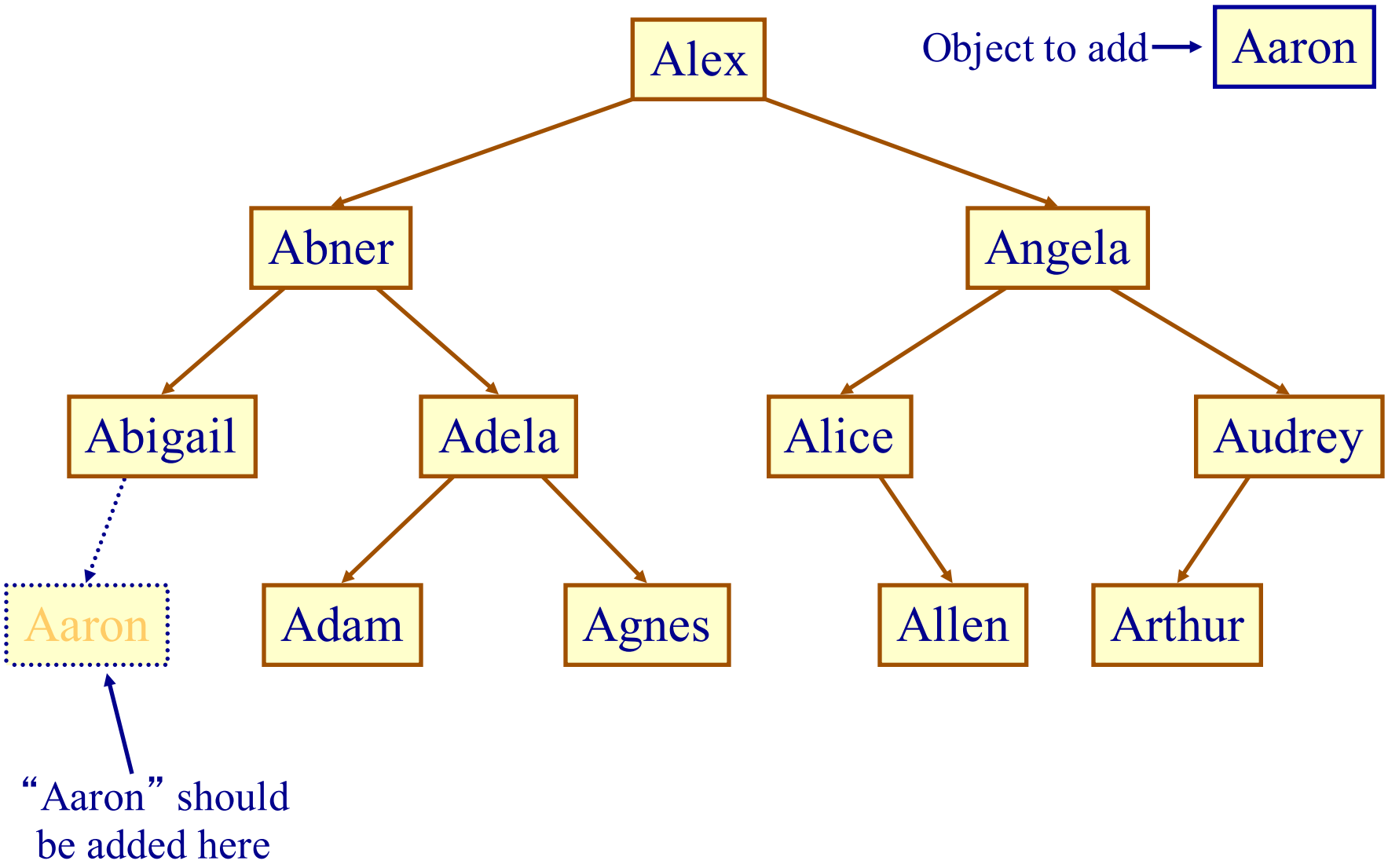
Bag Implementation: Contains

- Traverses a path from the root to the leaf
- Therefore, if tree is *reasonably full*, execution time is O(??)

BST: Contains/Find Example



BST: Add Example



Add: Calls a Utility Routine

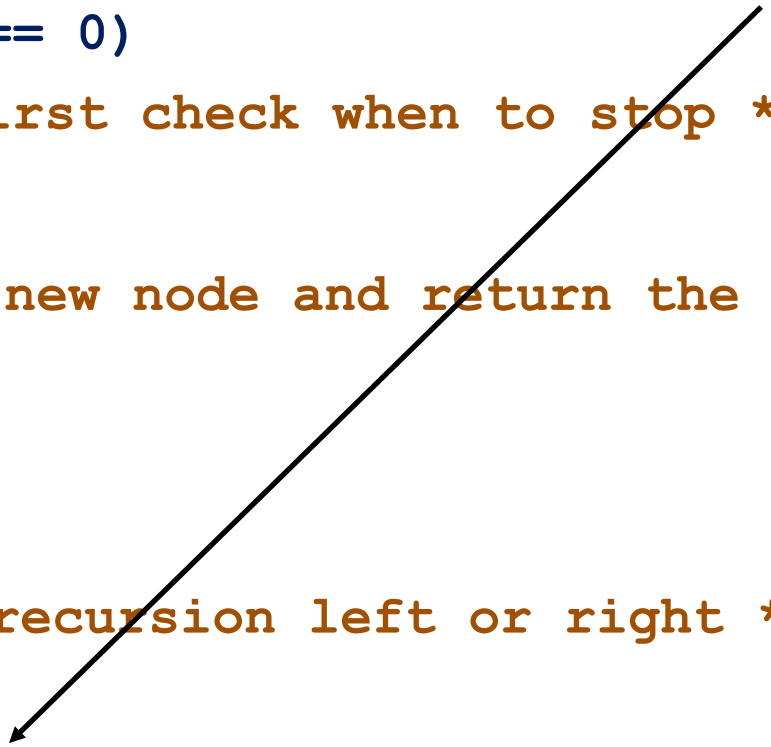
```
void addBST(struct BST *tree, TYPE val) {
  assert(tree);
  tree->root = _addNode(tree->root, val);
  tree->size++;
}
```

What is the complexity? O(??)

Useful Auxiliary Function -- Recursion

```
struct Node * _addNode(struct Node * current, TYPE val){
  if (current == 0)
  {/* always first check when to stop */

  ...  /*make a new node and return the pointer to it*/
  }
  else
  {
      /* call recursion left or right */
  }
  return current;
}
```



return the same pointer

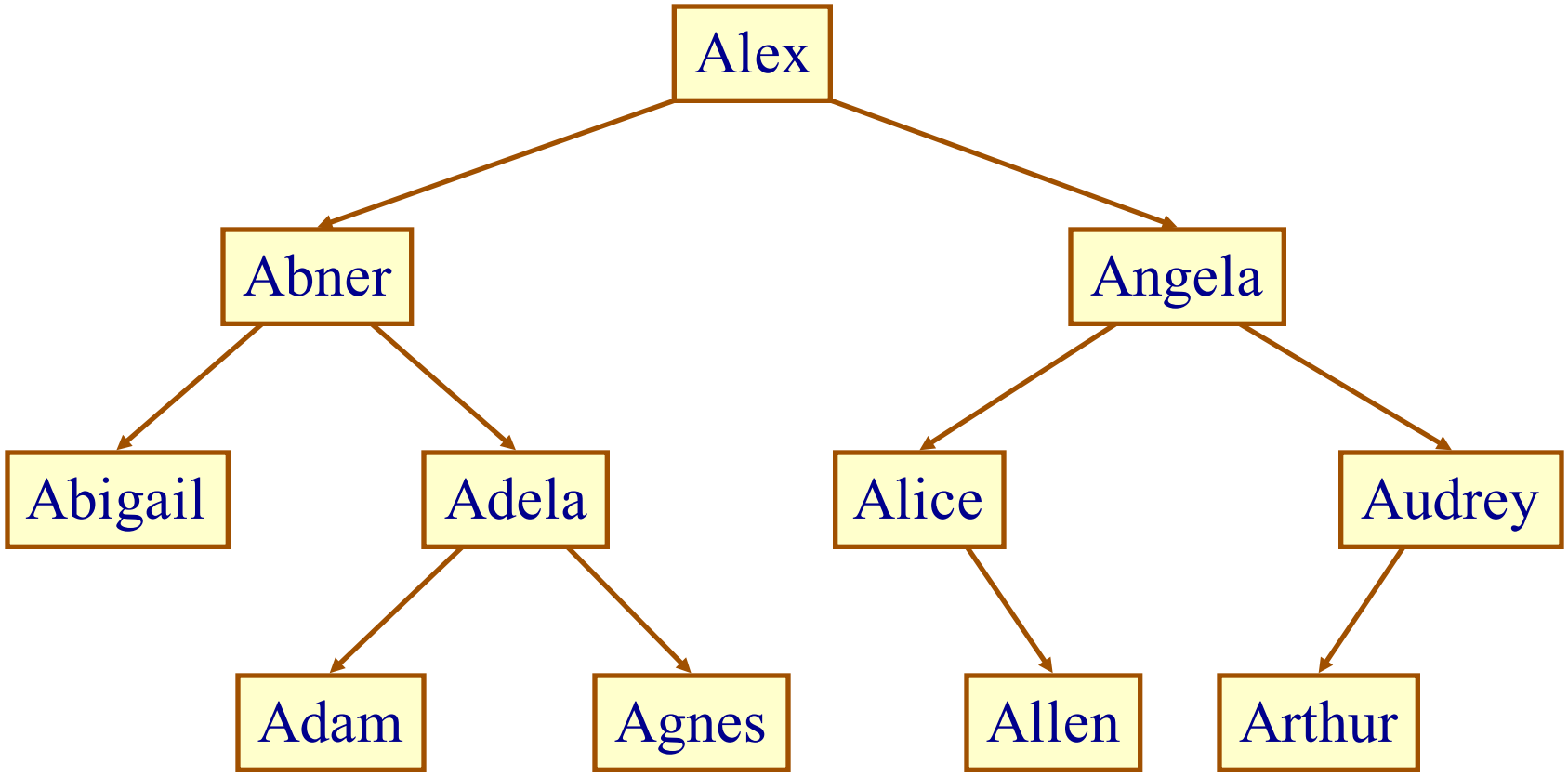
Useful Auxiliary Function -- Recursion

```
struct Node * _addNode(struct Node * current, TYPE val){
  if (current == 0){
  ...  /*make a new node and return the pointer to it*/
  }else{
     if (val < current->val)
       /* recursion left */
       current->left = _addNode(current->left, val);
     else
       /* recursion right */
       current->right = _addNode(current->right, val);
  }
  return current;
}
```

Useful Auxiliary Function

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

BST: Remove



How would you remove: Audrey? Angela?

Who Fills the Hole in BST?

- Answer:

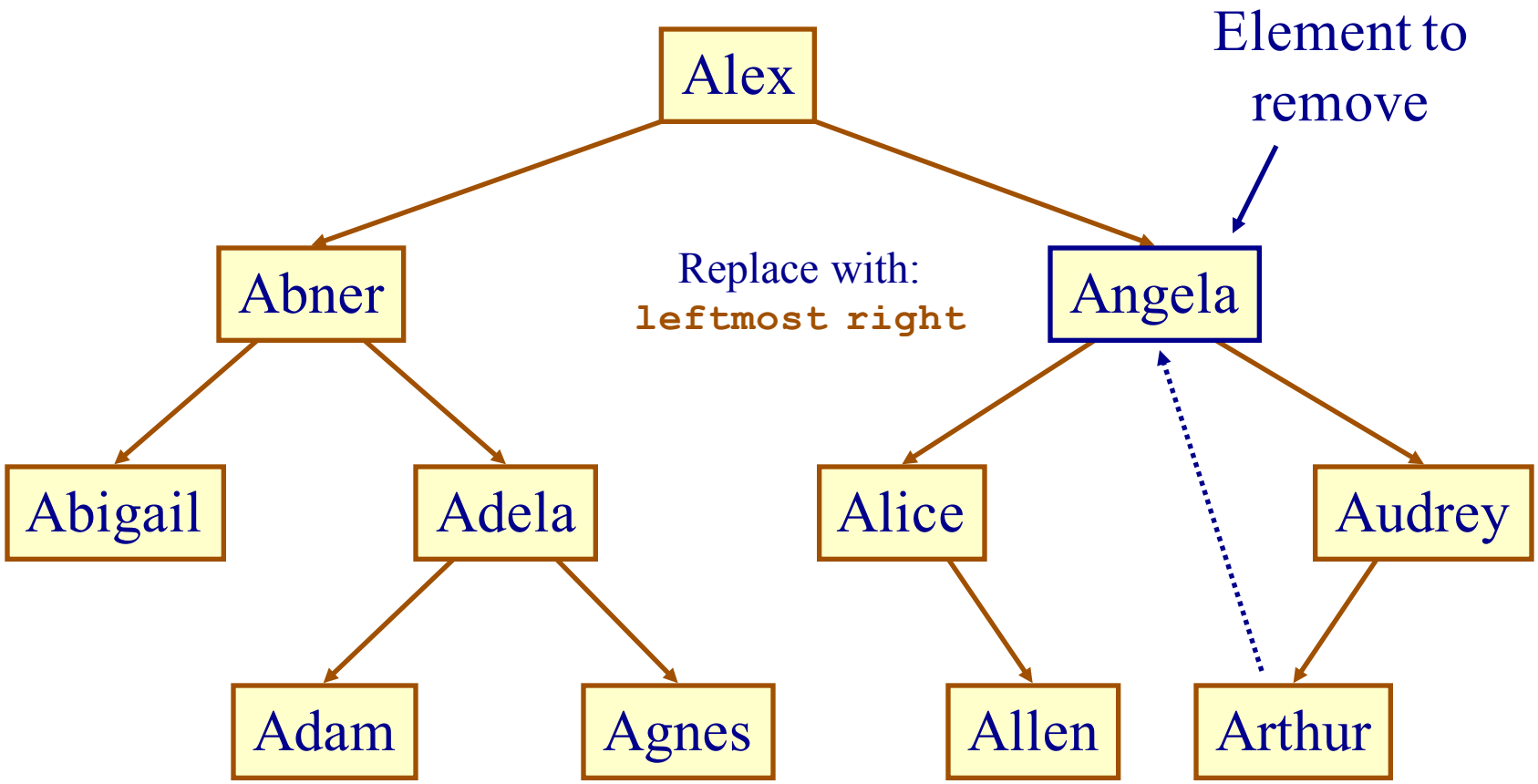
the leftmost descendant

of the right child

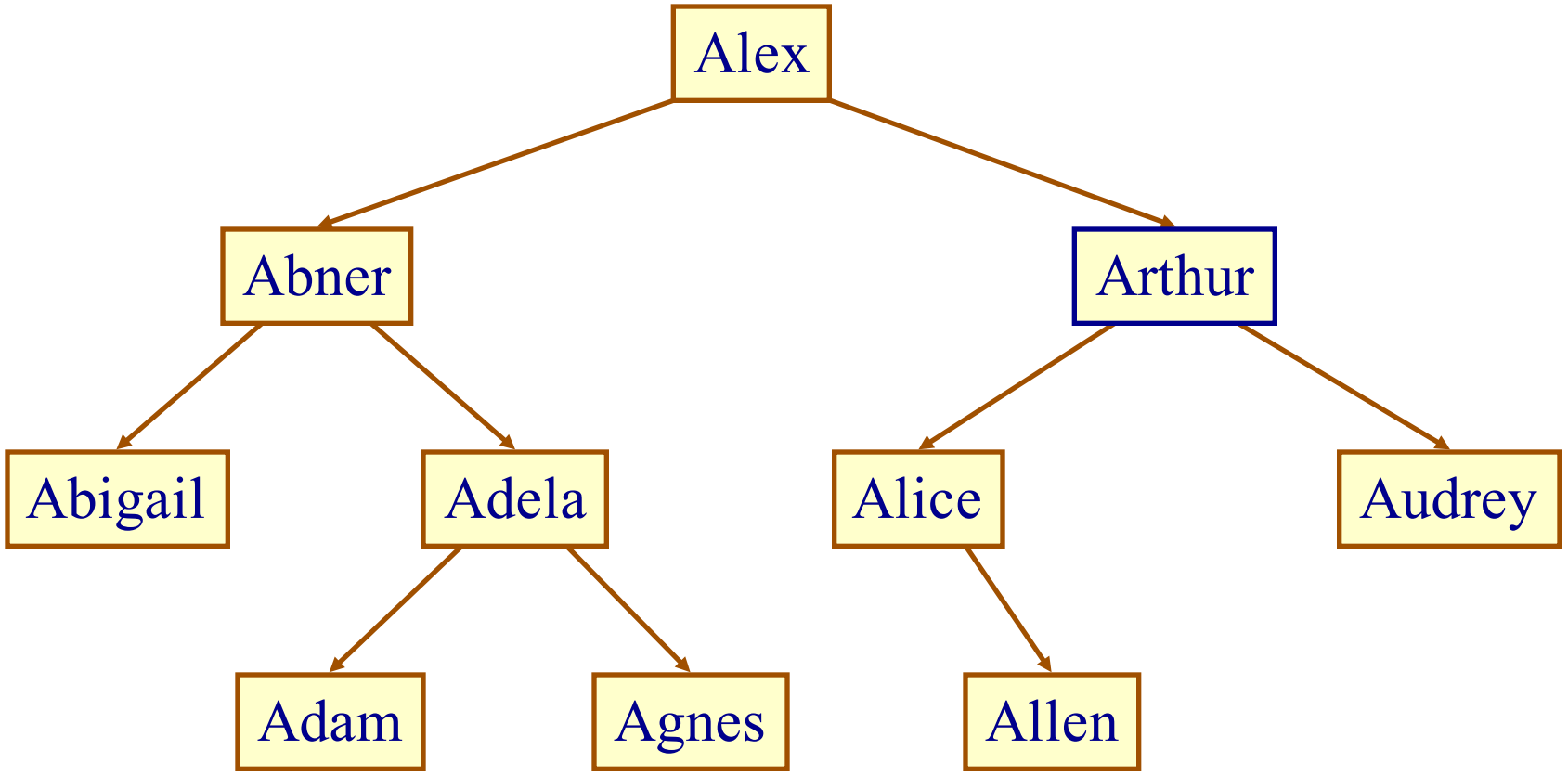
= smallest element in the right subtree

- Try this on a few values

BST: Remove Example



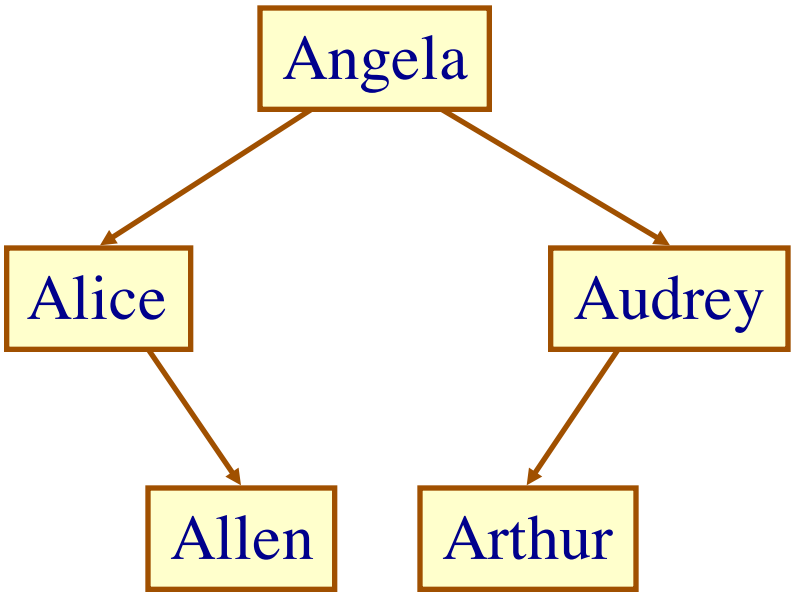
BST: Remove Example



After call to **`remove`**

Special Case

- What if you don't have the right child?
- Try removing "Audrey"
  - Simply, return the left child



Remove

```
void removeBST (struct BST *tree, TYPE e) {
  if (containsBST(tree, e)) {
    tree->root = _removeNode(tree->root, e);
    tree->size--;
  }
}
```

Remove Node

```
struct Node *_removeNode(struct Node *current, TYPE e){
  if (current->val == e)
     /* Found it, remove it */
  else
     /* not found, so recursive call */
}
```

Remove Node

```
struct Node *_removeNode(struct Node *current, TYPE e){
  struct Node *node;
  assert(current);
  if (current->val == e) /* Found it, remove it */
    if (current->right == NULL){
       node = current->left;
       free(current);
       return node; /* returns the left child */
    }
    else
        /* replace current->val with the value of
        the leftmost descendant of the right child */
        current->val = _leftMost(current->right);
        current->right = _removeLeftmost(current->right);
  else
     /* not found, so recursive call */
   ...
}
```

Remove Node

```
struct Node *_removeNode(struct Node *current, TYPE e){
  if (current->val == e) /* Found it, remove it */
     ...
  else /* not found, so recursive call */
     if( e < current->val)
       current->left = _removeNode(current->left, e);
     else
       current->right = _removeNode(current->right, e);

  return current;
}
```

Useful Routines: `_leftMost`

```
/*Returns value of leftmost child of current node*/

TYPE _leftMost(struct Node *current) {

   while(current->left != 0)

      current = current->left;

   return current->val;

}
```

Useful Routines: _removeLeftmost

```
/* Removes the leftmost descendant of current */
struct Node *_removeLeftmost(struct Node *current) {
   struct Node *temp;
   if(current->left != 0){
      current->left = _removeLeftmost(current->left);
      return current;
   }
   temp = current->right;
   free(current);
   return temp;
}
```

Complexity

- Running down a path from root to leaf
- What is the time complexity? O(??)