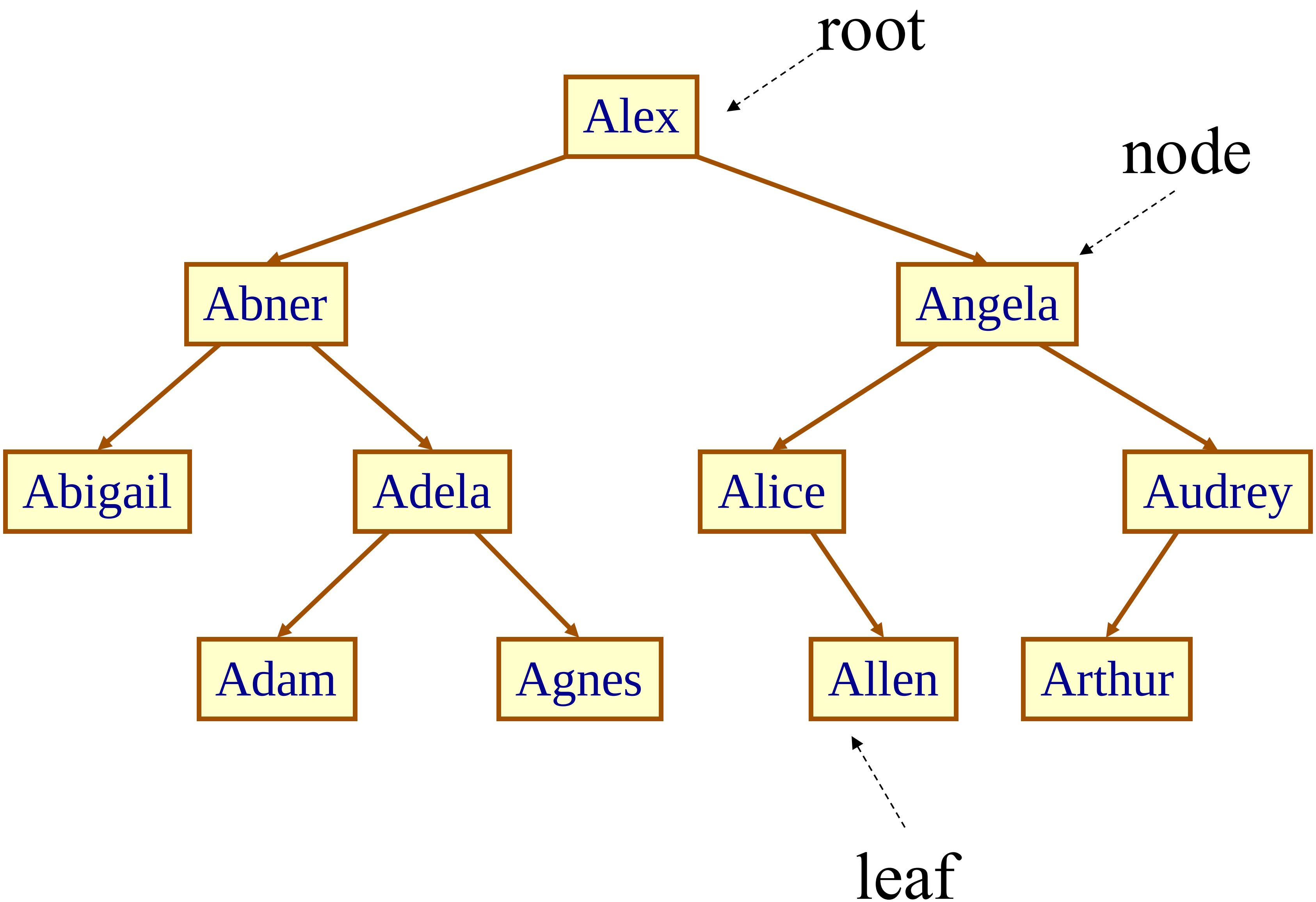
CS 261: Data Structures

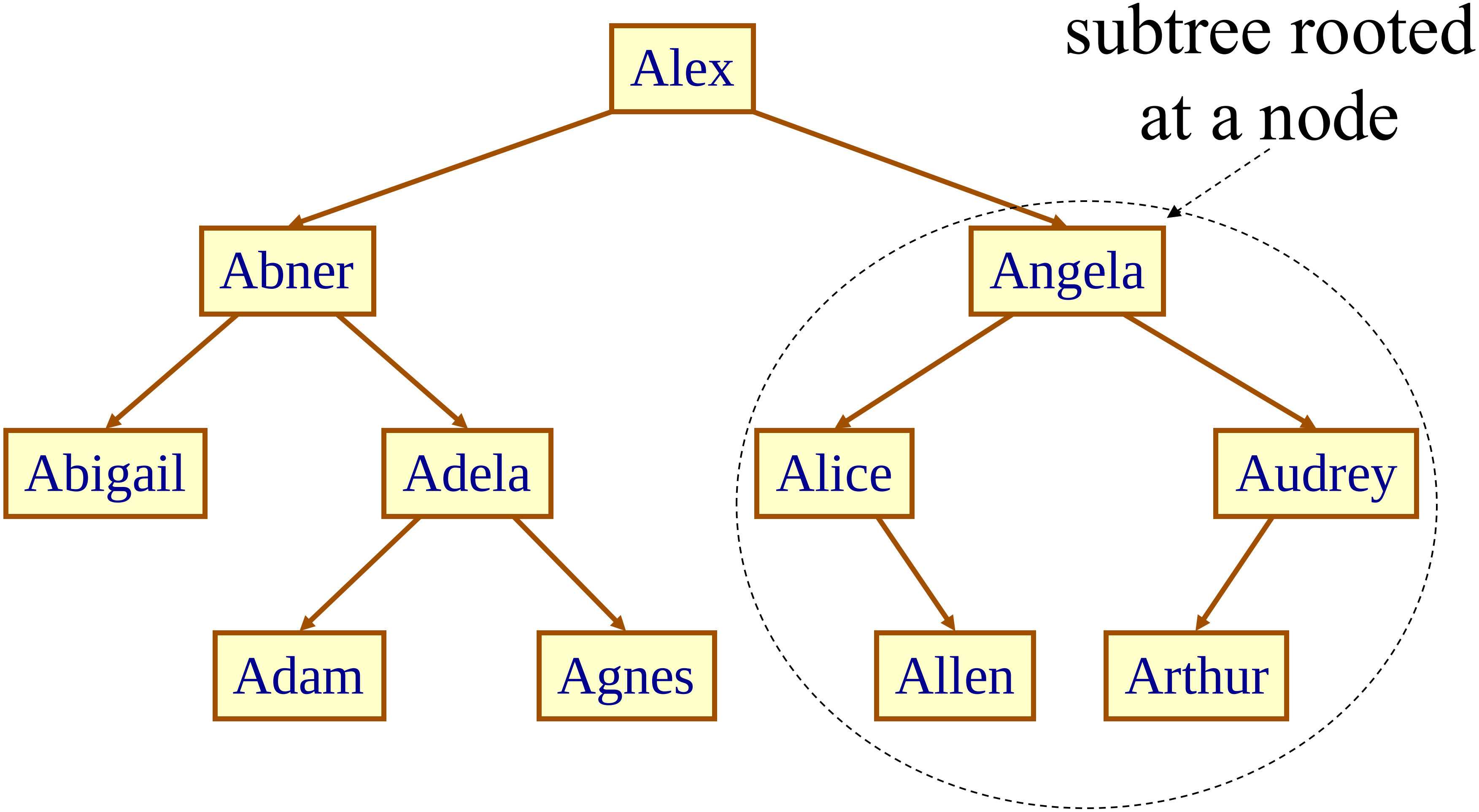
Binary Tree Traversals

Binary Search Trees

Binary Trees



Binary Trees



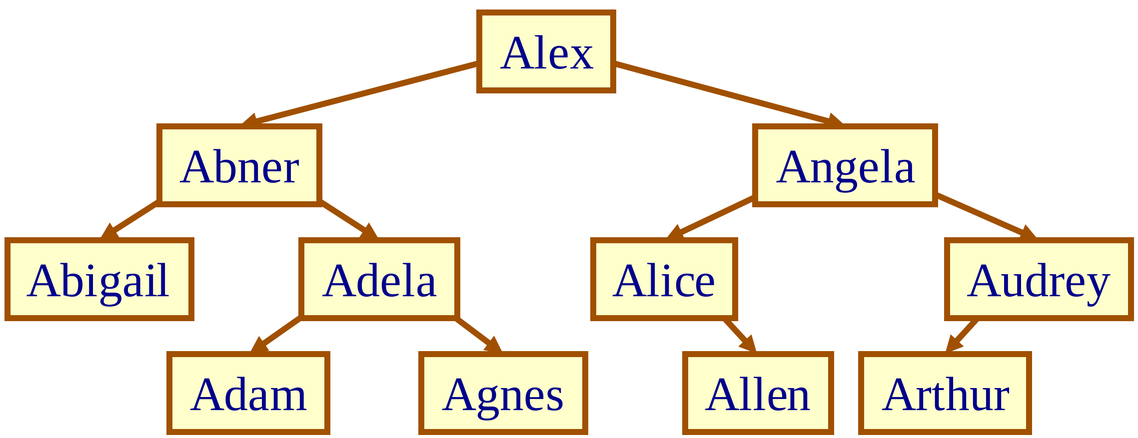
Node Structure Type

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

Like the **`Link`** structure in a linked list

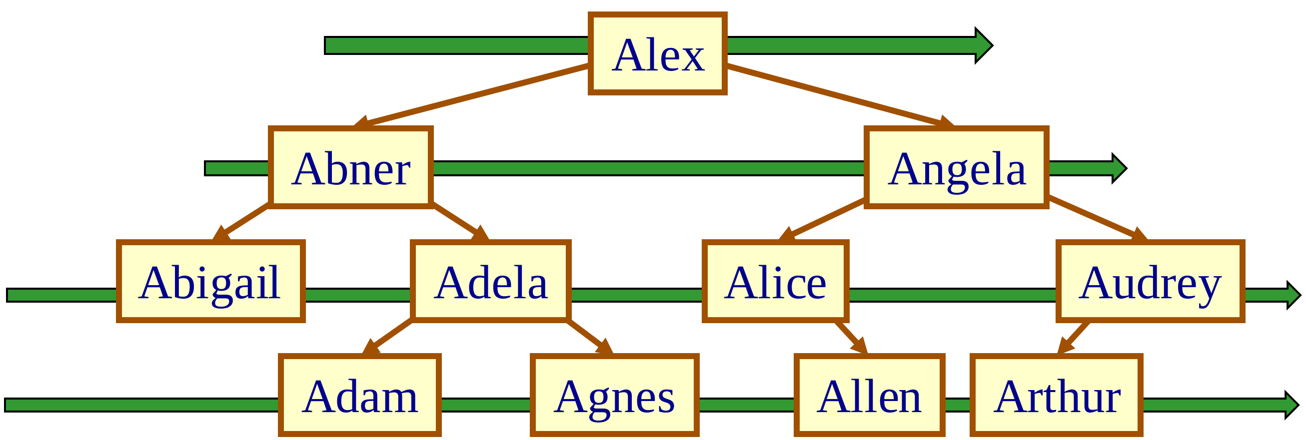
Tree Traversals

- How to access nodes in a tree?
- What order do we visit nodes in a tree?
  - Example:



Tree Traversals: Breadth-First

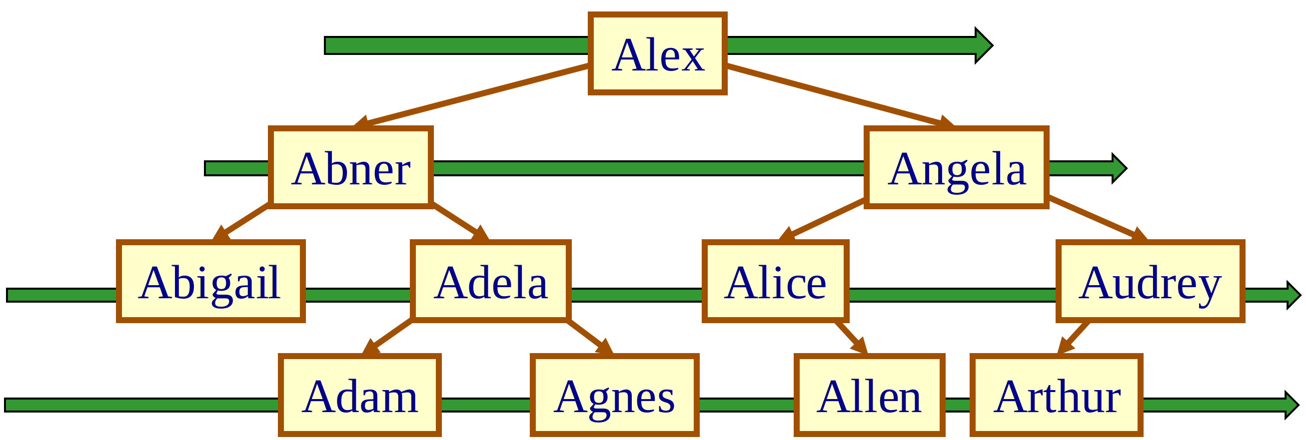
- Level-order or breadth-first traversals
  - **top-down**, or bottom-up
  - **left-to-right**, or right-to-left



Output: {Alex, Abner, Angela, Abigail, Adela, Alice, Audrey, Adam, Agnes, Allen, Arthur}

Tree Traversals: Breadth-First

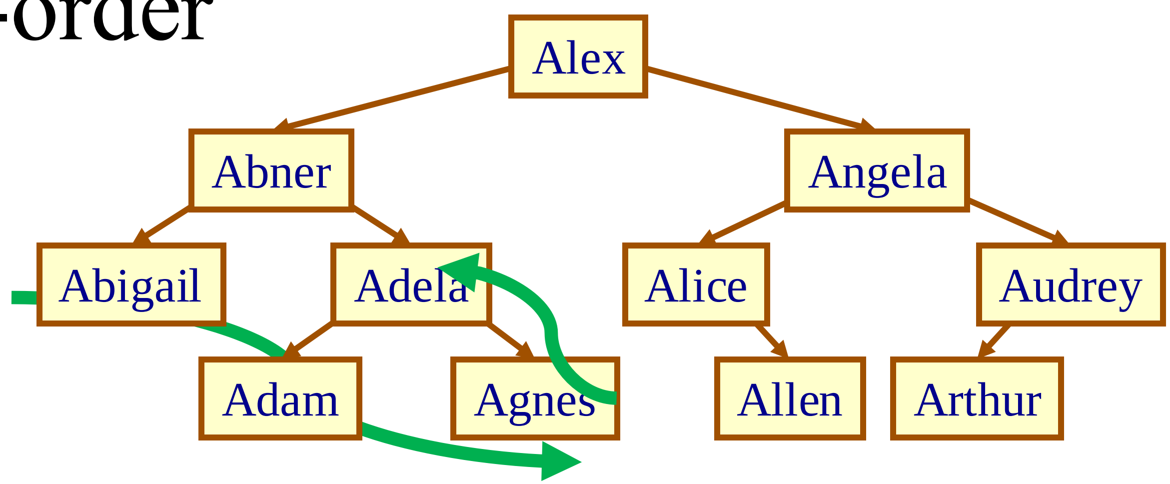
- Level-order or breadth-first traversals
  - –top-down, or **bottom-up**
  - –**left-to-right**, or right-to-left



Output: {Adam, Agnes, Allen, Arthur, Abigail, Adela, Alice, Audrey, Abner, Angela, Alex}

Tree Traversals: Depth-First

- Depth-first traversals:
  - –pre-order
  - –**post-order**
  - –in-order



Output: {Abigail, Adam, Agnes, Adela, Abner, Allen, Alice, Arthur, Audrey, Angela, Alex}

Tree Traversals

- Recursive algorithms
- Iterative algorithms

You need to know how to implement both types of traversals in C.

Recursive Functions: Important Checks

1. Must have a **stopping criterion.**
2. The recursive call uses a **different input.**

```
TYPE recursive_f(input1) {
    if( stop ){
        process_stop;
    else{
        process(input1);
        recursive_f(input2);
    }
     return some_result;
  }
```

Depth-First Traversals

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

Pre-Order Traversal

Processing order: Node → Left → Right

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

Check correctness!

Pre-Order Traversal

Processing order: Node → Left → Right

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

Stopping criterion ✅

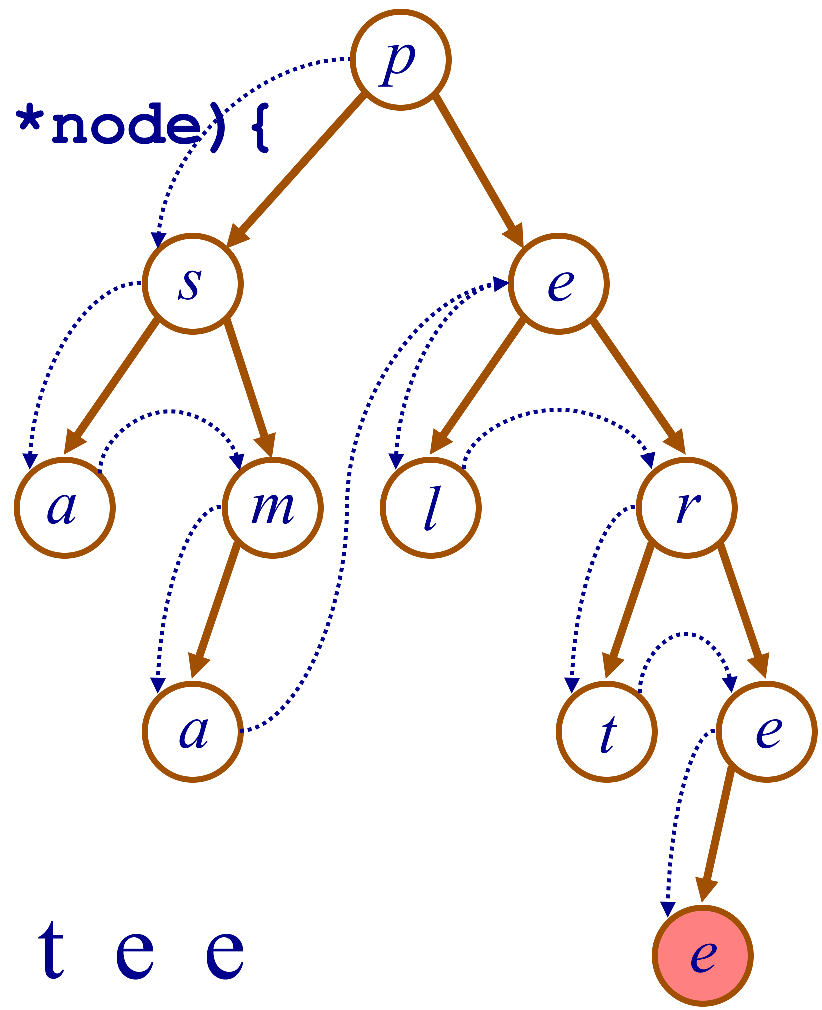
different input ✅

Pre-Order Traversal

Processing order: Node → Left → Right

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

COMPLEXITY?

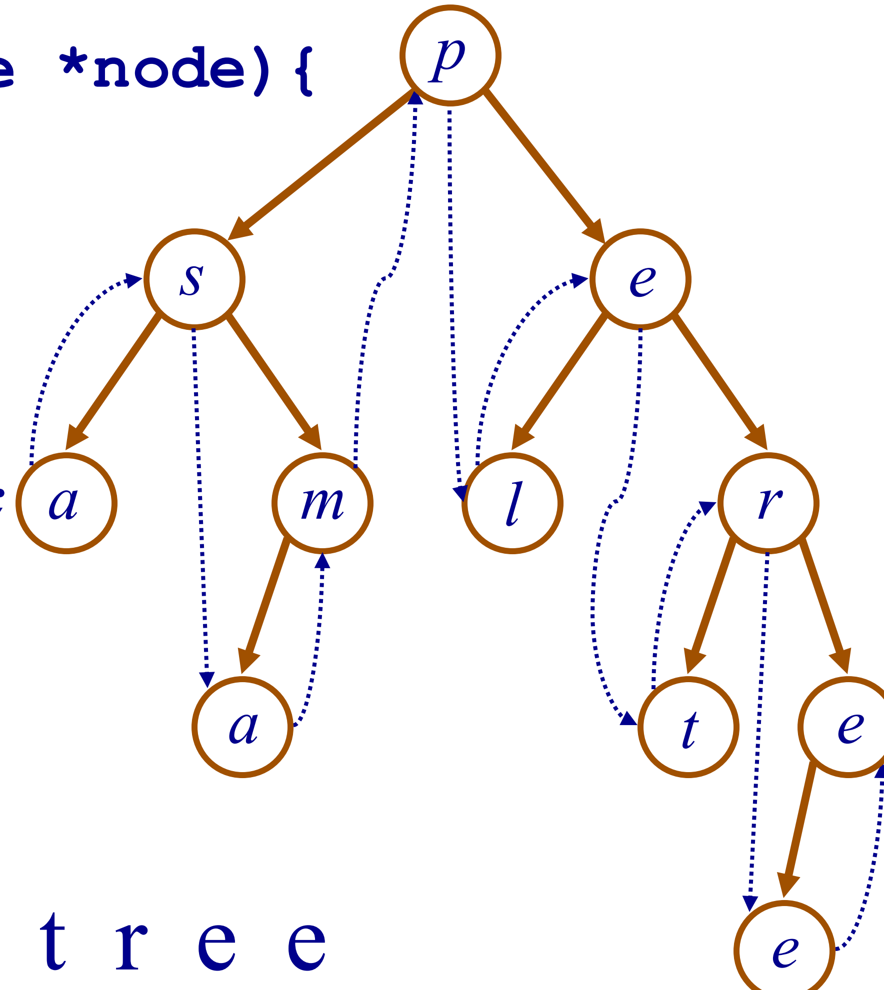


Result: p s a m a e l r t e e

In-Order Traversal

Processing order: Left → Node → Right

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



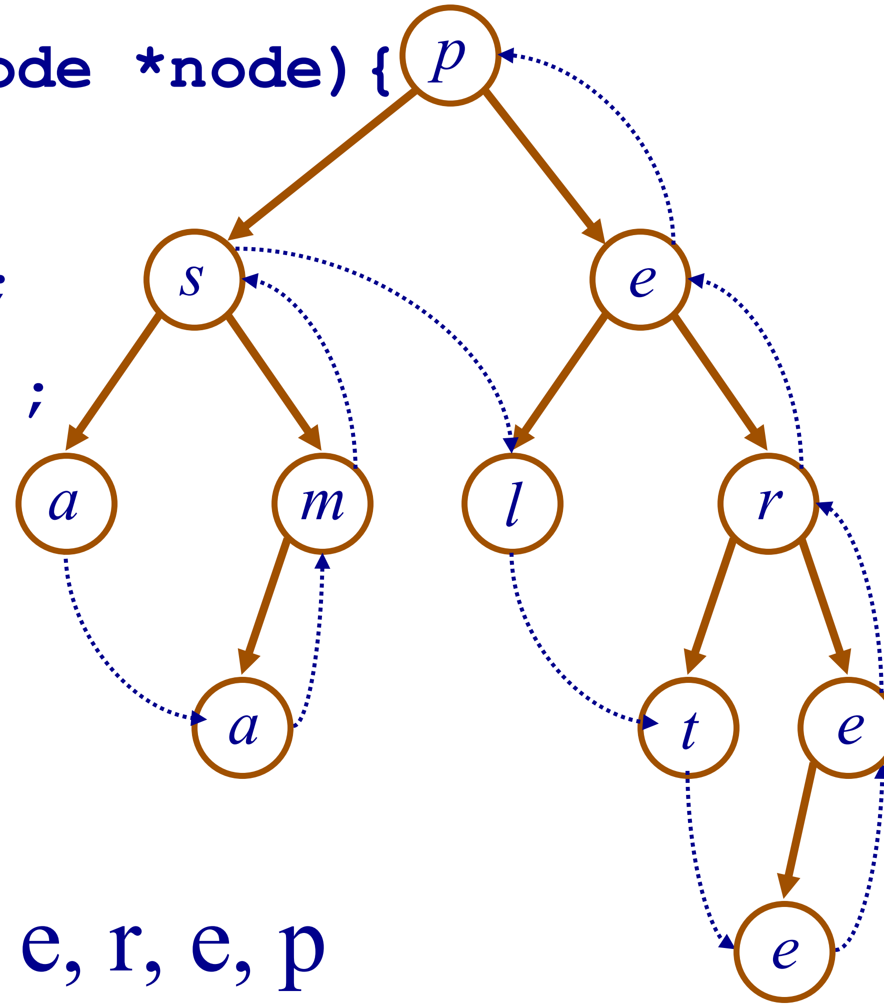
COMPLEXITY?

Result: a s a m p l e t r e e

Post-Order Traversal

Processing order: Left → Right → Node

```
void postorder(struct Node *node){
  if (node != NULL){
    postorder(node->left);
    postorder(node->right);
    process(node);
  }
}
```



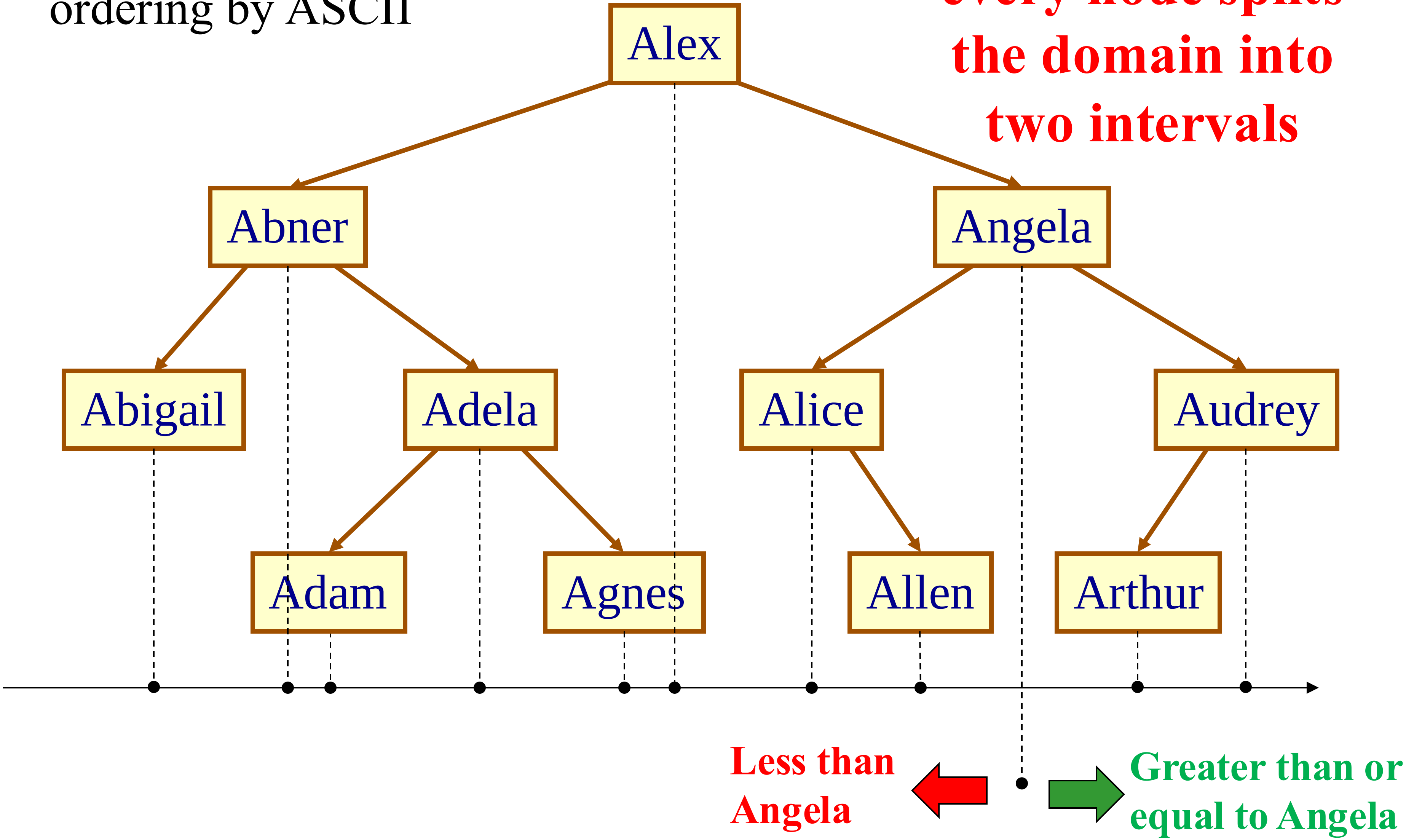
COMPLEXITY?

Result: a, a, m, s, l, t, e, e, r, e, p

Binary Search Tree: Example

ordering by ASCII

every node splits the domain into two intervals

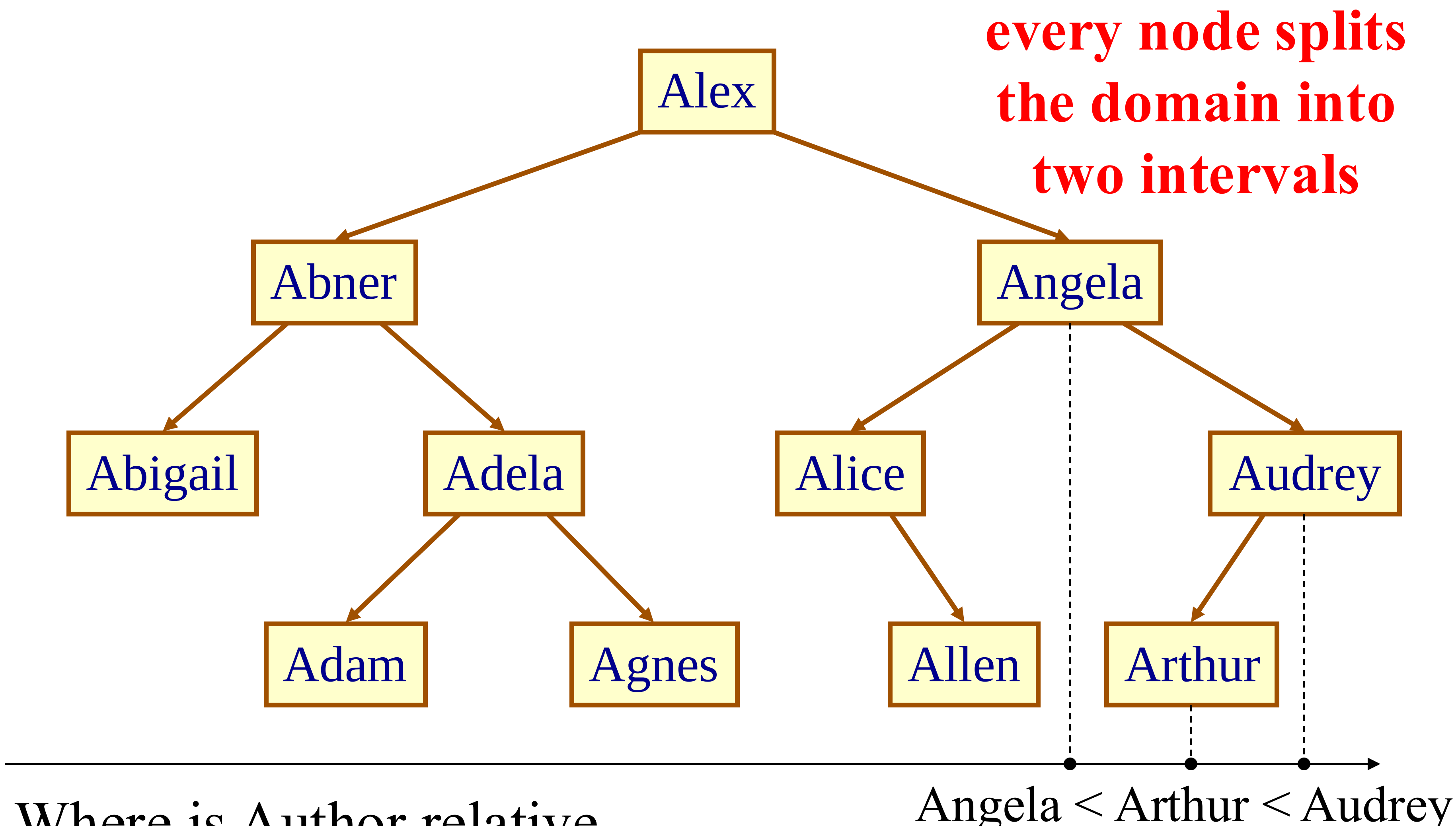


Binary Search Tree (BST)

A binary tree where every node value is:

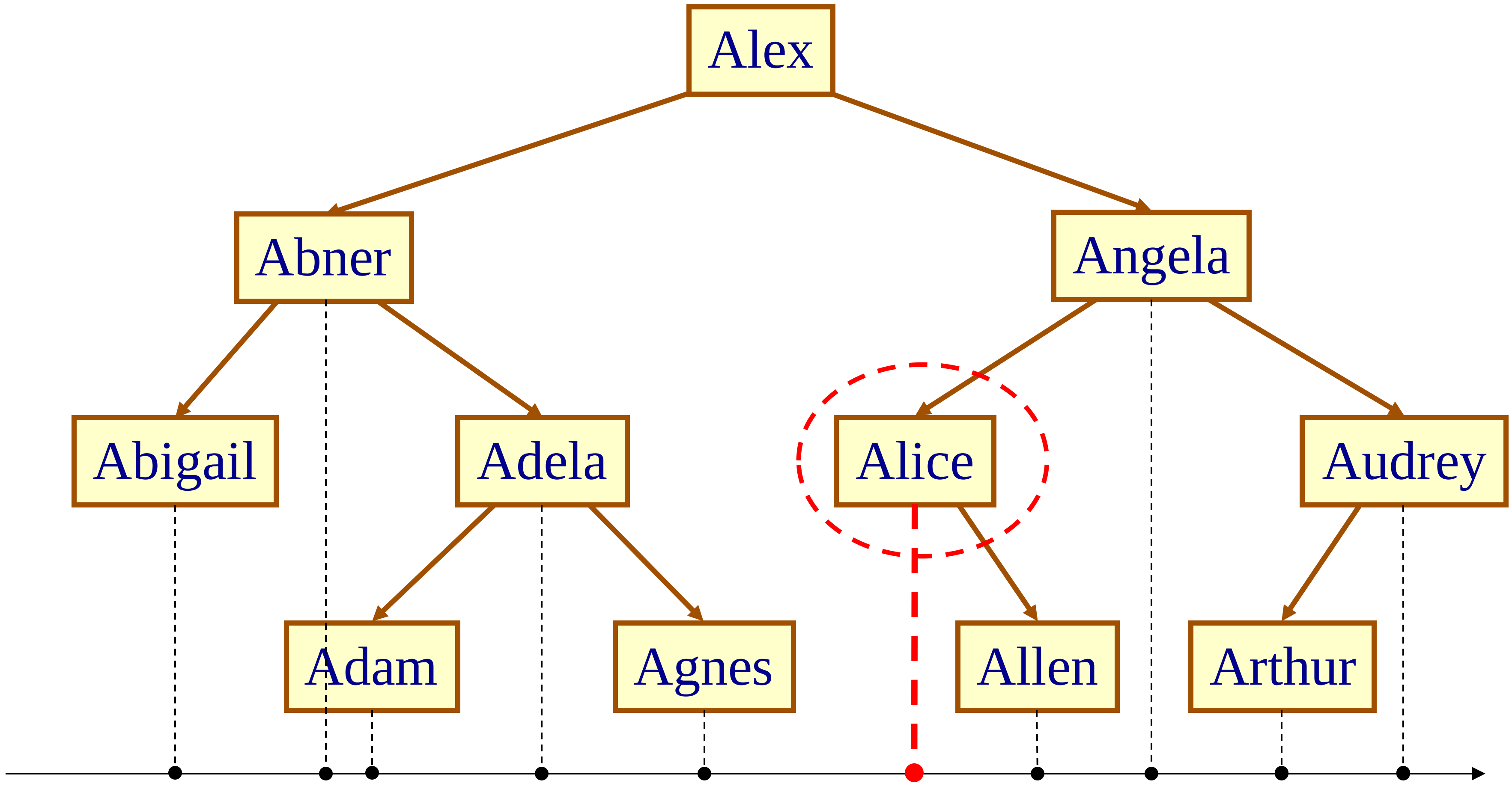
- **Greater than** all of its **left** descendants
- **Less than or equal to** all of its **right** descendants

Binary Search Tree: Example



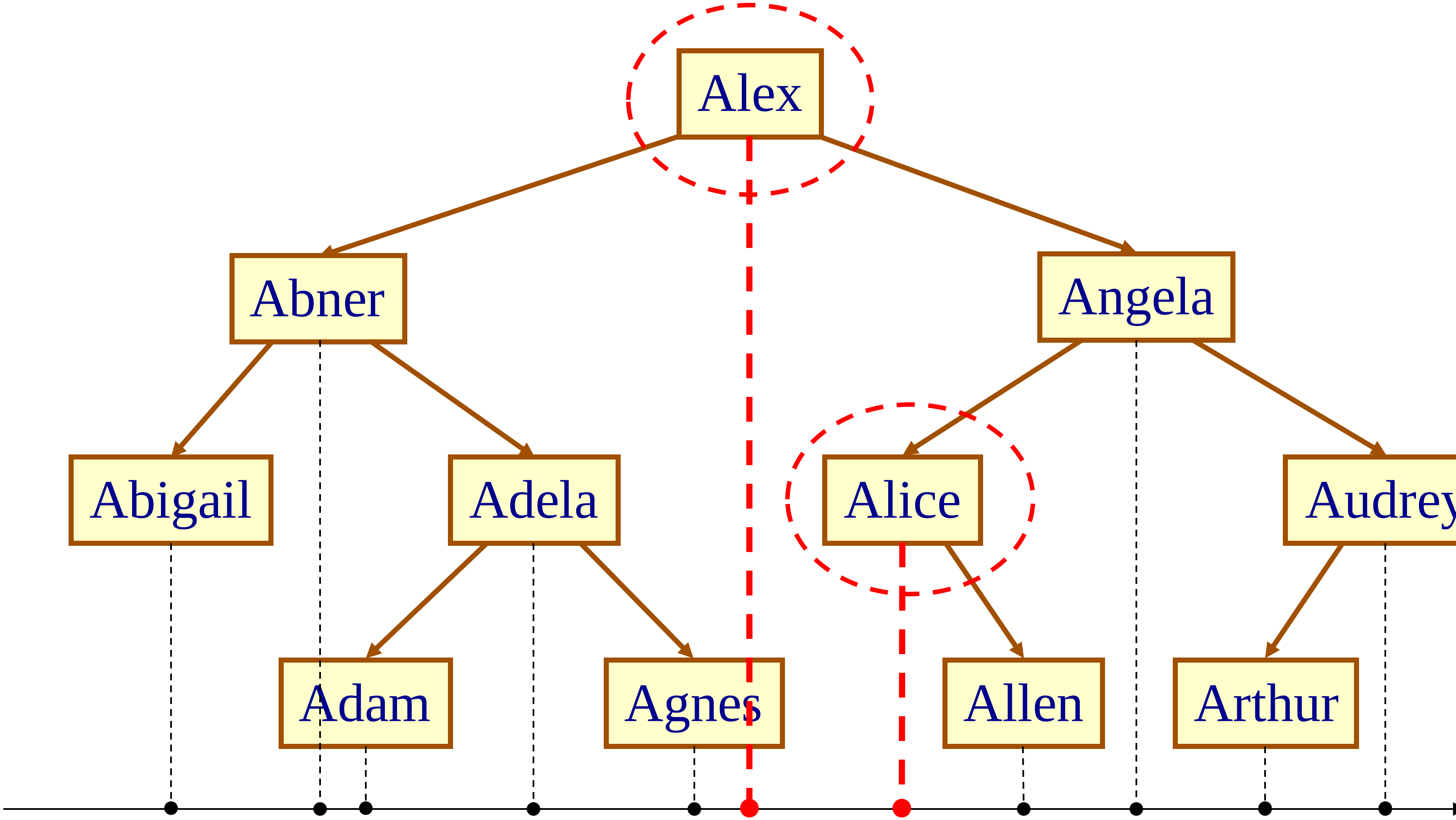
Where is Author relative to Audrey on the axis?

Binary Search Tree: Example



What is the leftmost descendant of Angela?

Binary Search Tree: Example



The leftmost descendant of my right child is my next higher node!

BST: Interface

```
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);
```

BST Uses Two Struct Types

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

struct BST {
  struct Node *root;
  int size; /*number of nodes*/
};
```

Initialize BST

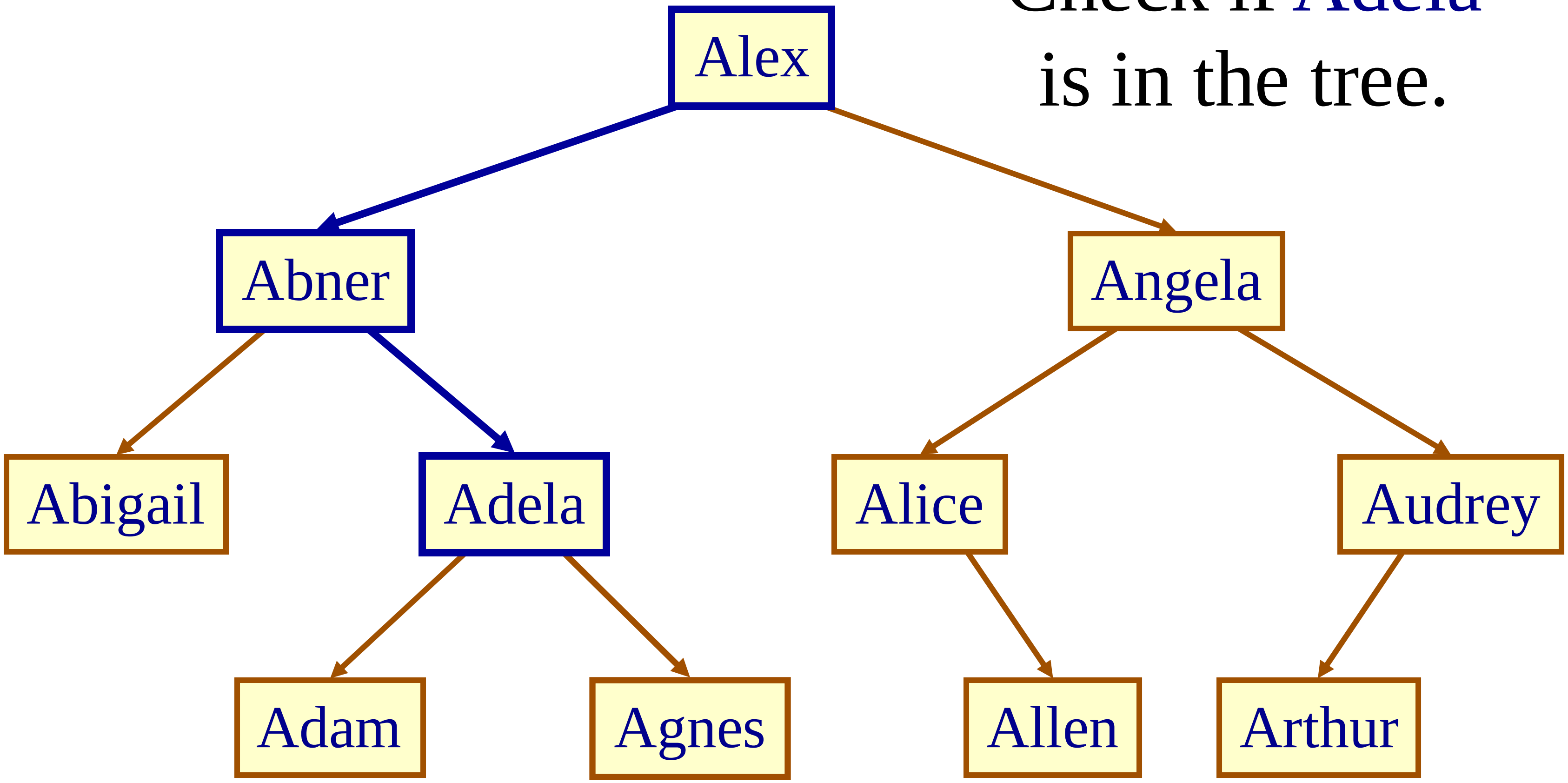
```c
struct Node {
  TYPE val;
  struct Node *left;
  struct Node *right;
};
```

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

```c
void initBST(struct BST *tree){
   assert(tree);
   /*initially, the tree is empty */
   tree->root = NULL; /* important */
   tree->size = 0;
}
```

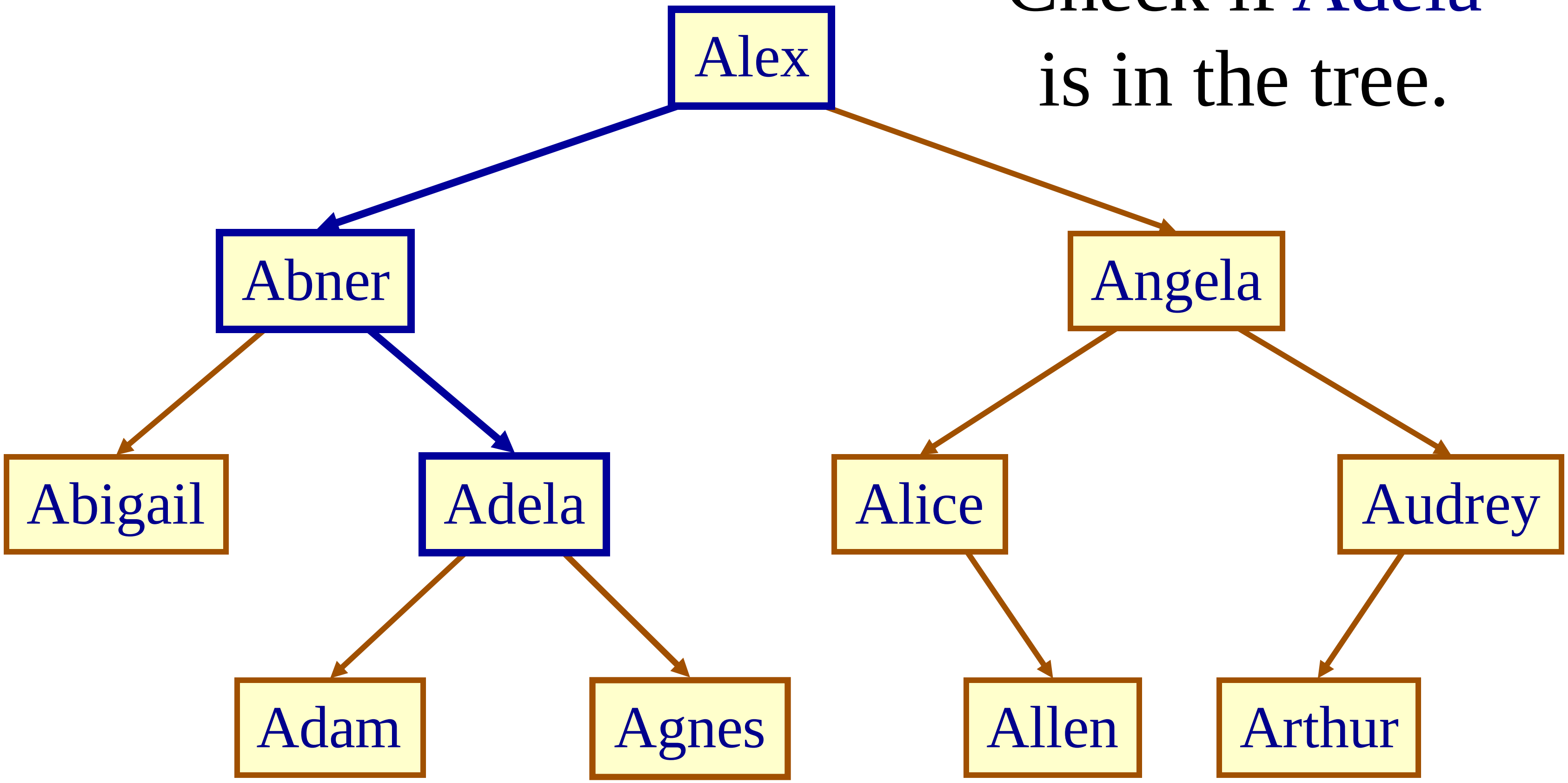
Contains BST

Check if Adela is in the tree.



We will leverage the BST property to efficiently traverse the tree to the node containing Adela.

Contains BST



Check if Adela is in the tree.

If BST is balanced, complexity of **`containsBST`** is proportional to the tree height O(log *n*)

Contains BST

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

```
 /* returns 1 or 0 */
int containsBST(struct BST *tree, TYPE val){
  assert(tree);




}
```

Contains BST

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

```
/* returns 1 or 0 */
int containsBST(struct BST *tree, TYPE val){
   assert(tree);
   if (tree->root)
      return containsNode(tree->root, val);
   else
      return 0; /* tree is empty */
}
```

Keep in mind, we will always need a helper function for a recursive tree traversal.

```
int containsNode(struct Node *node, TYPE val){
    /*Ensure the stopping criterion exists*/
    if(!node) return 0;

}
```

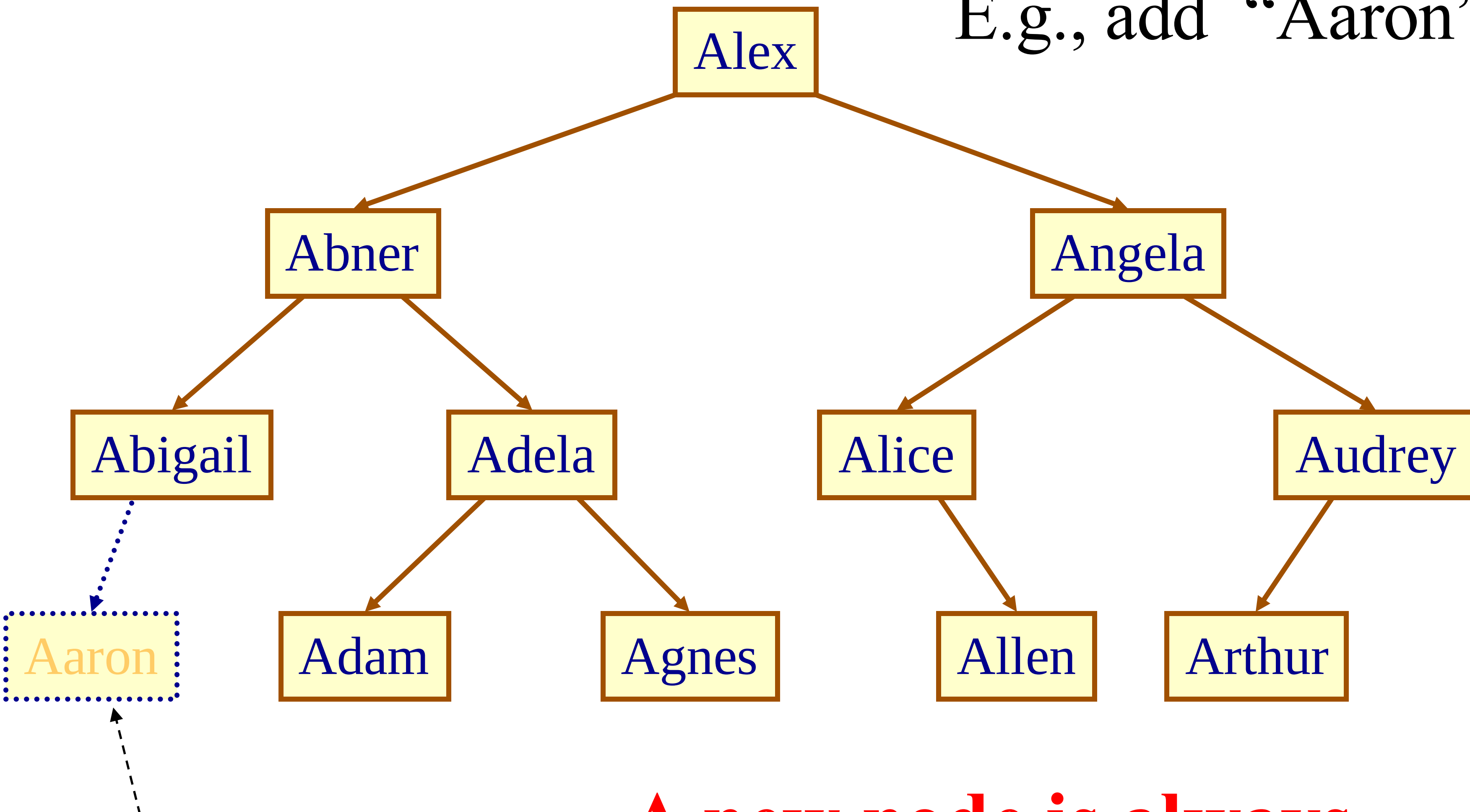
In our top-down traversal, we reached beyond the leaf node, so we couldn't find val in the tree.

```
int containsNode(struct Node *node, TYPE val){
   /*Ensure the stopping criterion exists*/
   if(!node) return 0;
   if( EQ(val, node->val) )
      return 1; /* found it */
   else
    /* Recursive calls of containsNode() */




}
```

```
int containsNode(struct Node *node, TYPE val){
   /*Ensure the stopping criterion exists*/
   if(!node) return 0;
   if( EQ(val, node->val) )
      return 1; /* found it */
   else if( LT(val, node->val) )
      /*Recursion must use different input*/
      return containsNode(node->left, val);
   else
     return containsNode(node->right, val);
}
```

Add BST

E.g., add "Aaron"

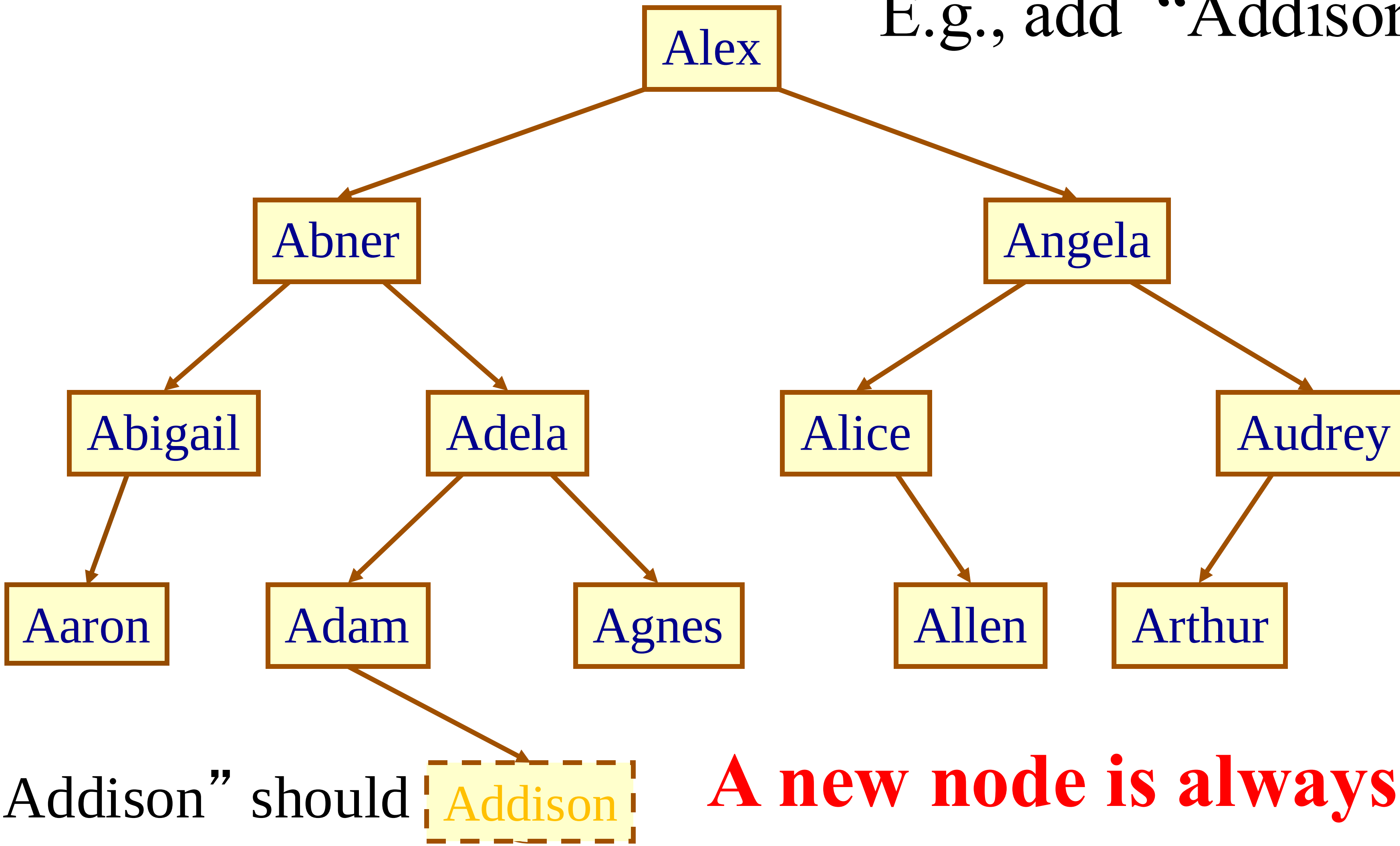


"Aaron" should be added here

A new node is always inserted as a leaf !

Add BST

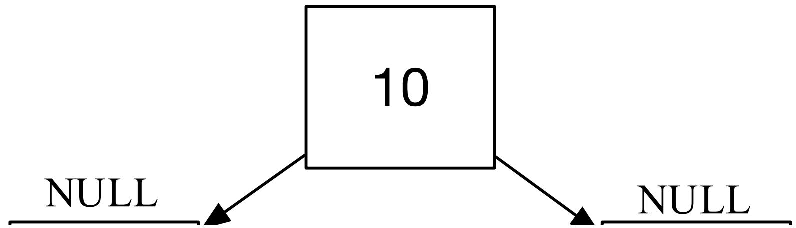
E.g., add "Addison"



"Addison" should be added here

A new node is always inserted as a leaf !

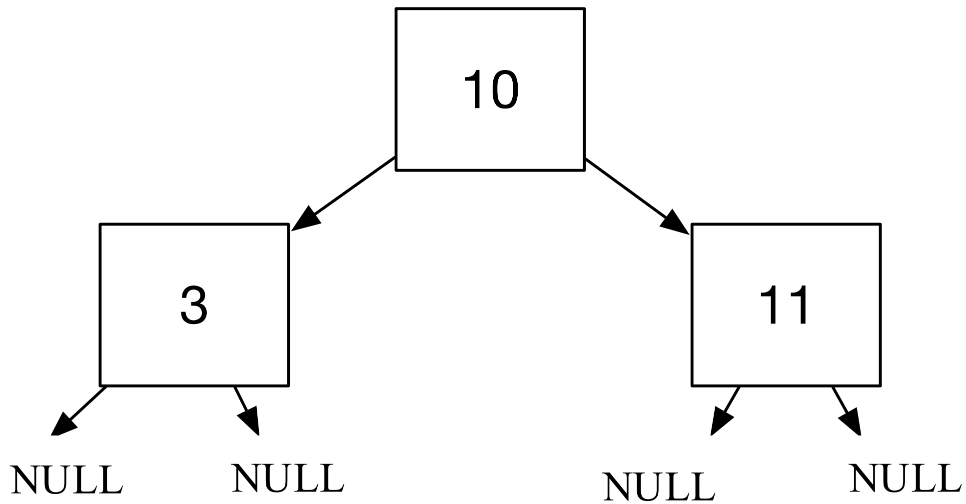
Example



Add the sequence:

10,
3,
11,
2,
10,
5,
6

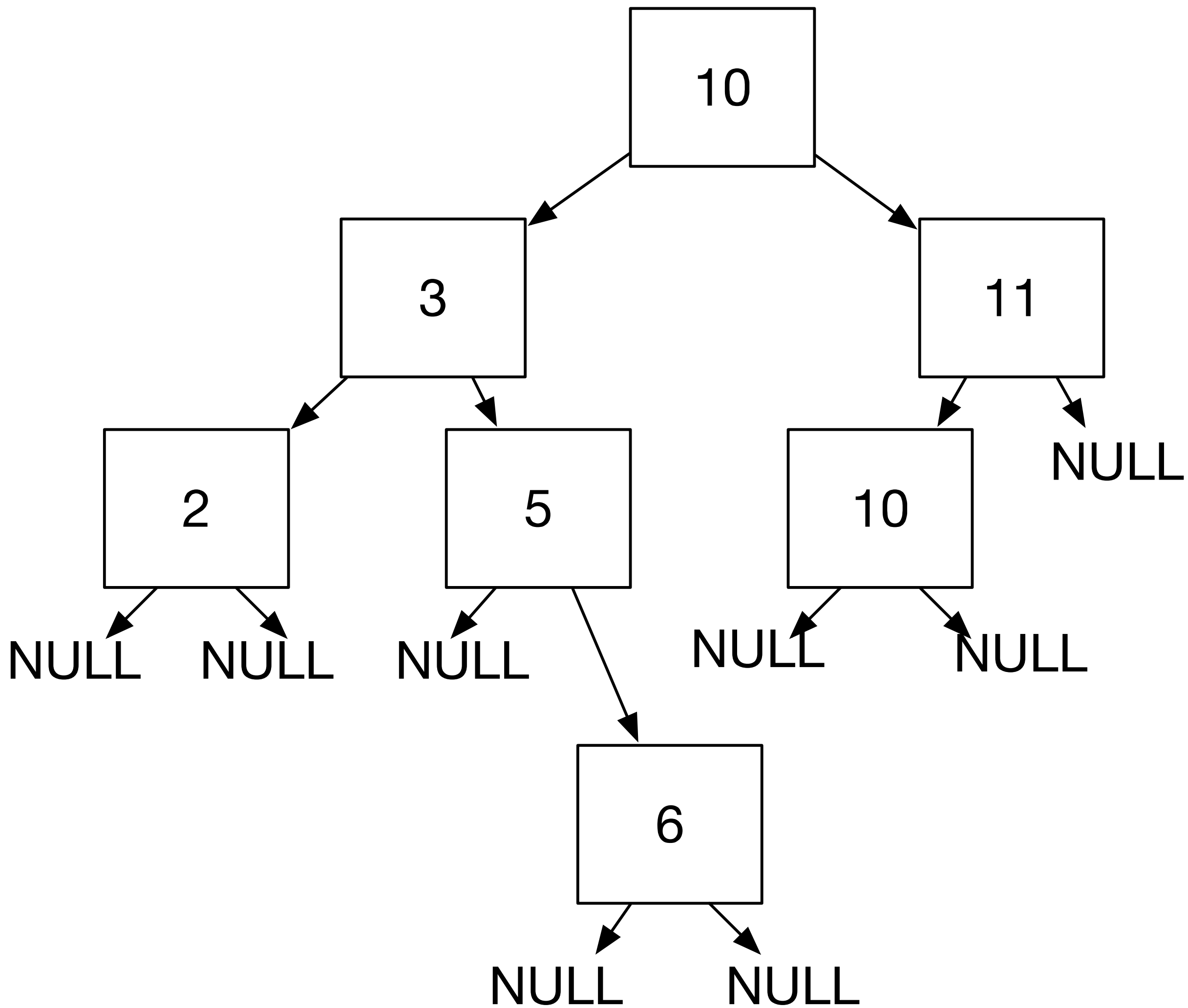
Example



Add the sequence:

10,
3,
11,
2,
10,
5,
6

Example



Add the sequence:
10,
3,
11,
2,
10,
5,
6

Add BST

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

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

Keep in mind, we will always need a helper function for a recursive tree traversal.

What is complexity of addBST? O(??)

```c
void initBST(struct BST *tree){
    tree->root = NULL;
    tree->size = 0;
}
```

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

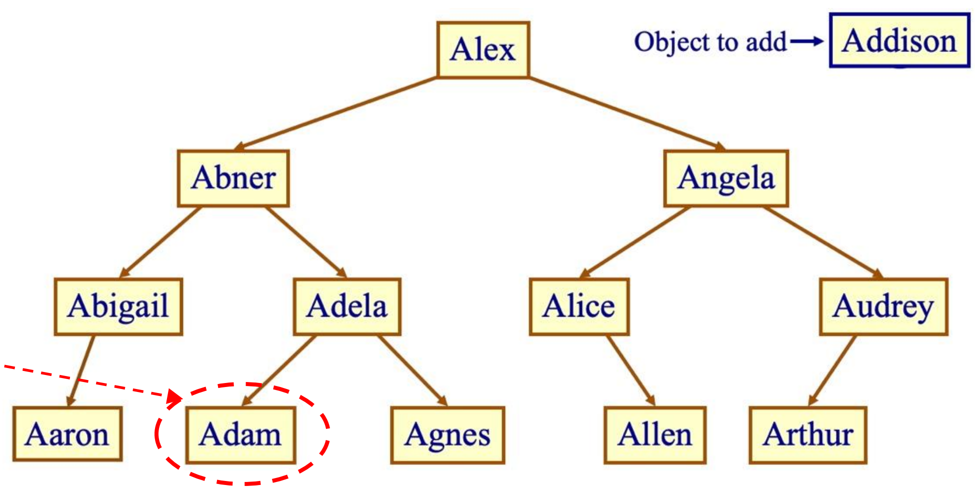
The function returns a pointer to the specified input node to propagate the changes up the tree.

```c
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
   ...




   return node;
}
```

return the same pointer

```
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
  if(!node){ /* stopping criterion */
  /*reached beyond a leaf, insert a new node*/
  }
  else{
  /* recursion left or right to find
     where we can insert val */
  }
  return node;
}
```

“Addison” should be added here



```c
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
   if(!node){ /* stopping criterion */
       /* insert a new node */
   }
   else{/* recursion left or right */
    if (LT(val, node->val))
      node->left = addNode(node->left, val);
    else
      node->right = addNode(node->right, val);
   }
   return node;
}
```

```c
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
   if(!node){ /* stopping criterion */
       /* insert a new node */
   }
   else{/* recursion left or right */
    if (LT(val, node->val))
      node->left = addNode(node->left, val);
    else
      node->right = addNode(node->right, val);
   }
   return node;
}
```

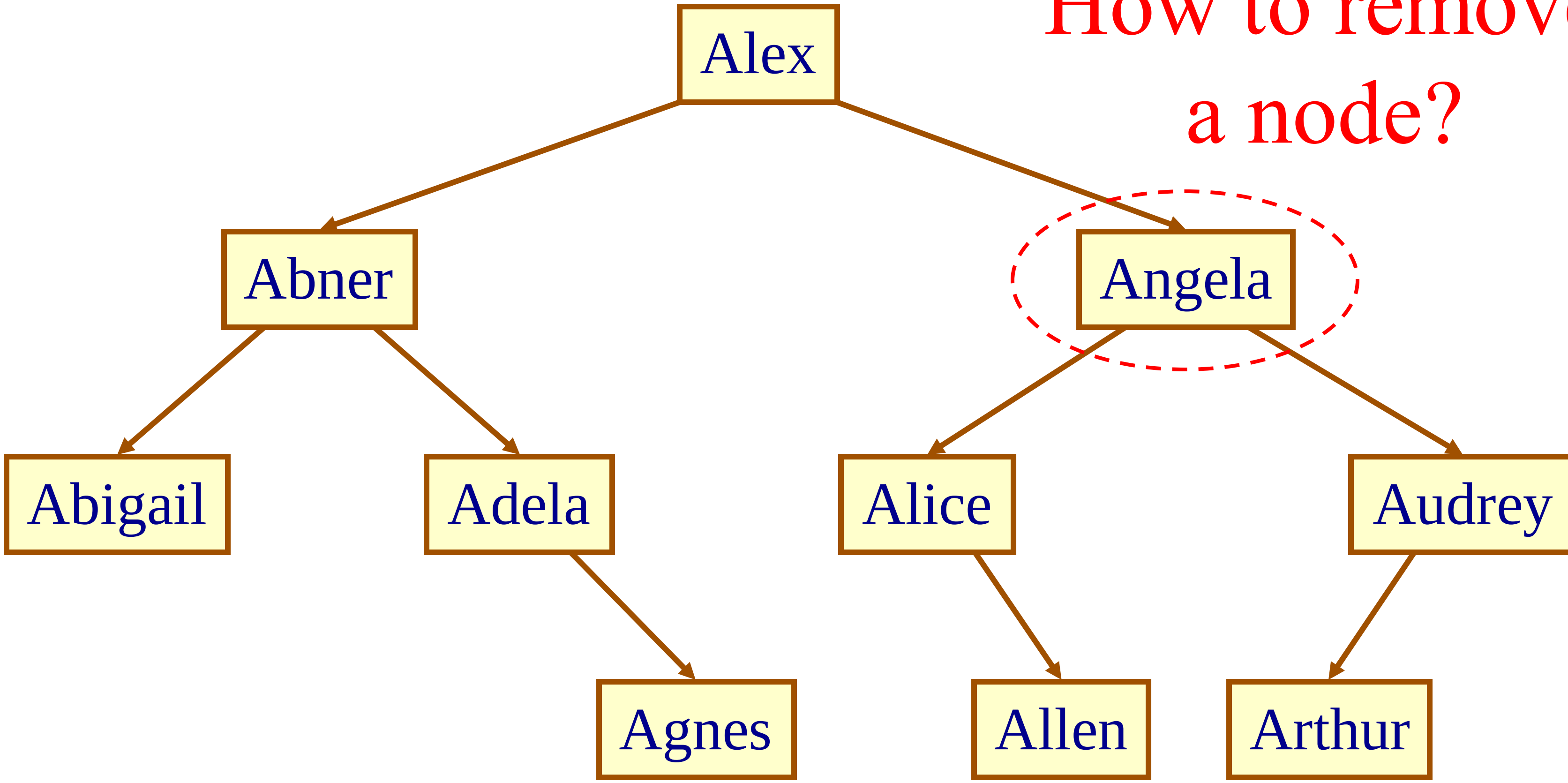
The function returns a pointer to the specified input node to propagate the changes up the tree.

```c
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
   if(!node){ /* stopping criterion */
       /* insert a new node */
   }
   else{/* recursion left or right */
    if (LT(val, node->val))
      node->left = addNode(node->left, val);
    else
      node->right = addNode(node->right, val);
   }
   return node;
}
```

the recursion must use a different input

```c
#define NODE struct Node

NODE *addNode(NODE *node, TYPE val){
  if(!node){ /* stopping criterion */
    NODE *new = (NODE *) malloc(sizeof(NODE));
    assert(new);
    new->val = val;
    new->left = new->right = NULL; /*leaf*/
    return new; /*propagate the changes up*/
  }
  else{/* recursion left or right */}
  return node;
}
```

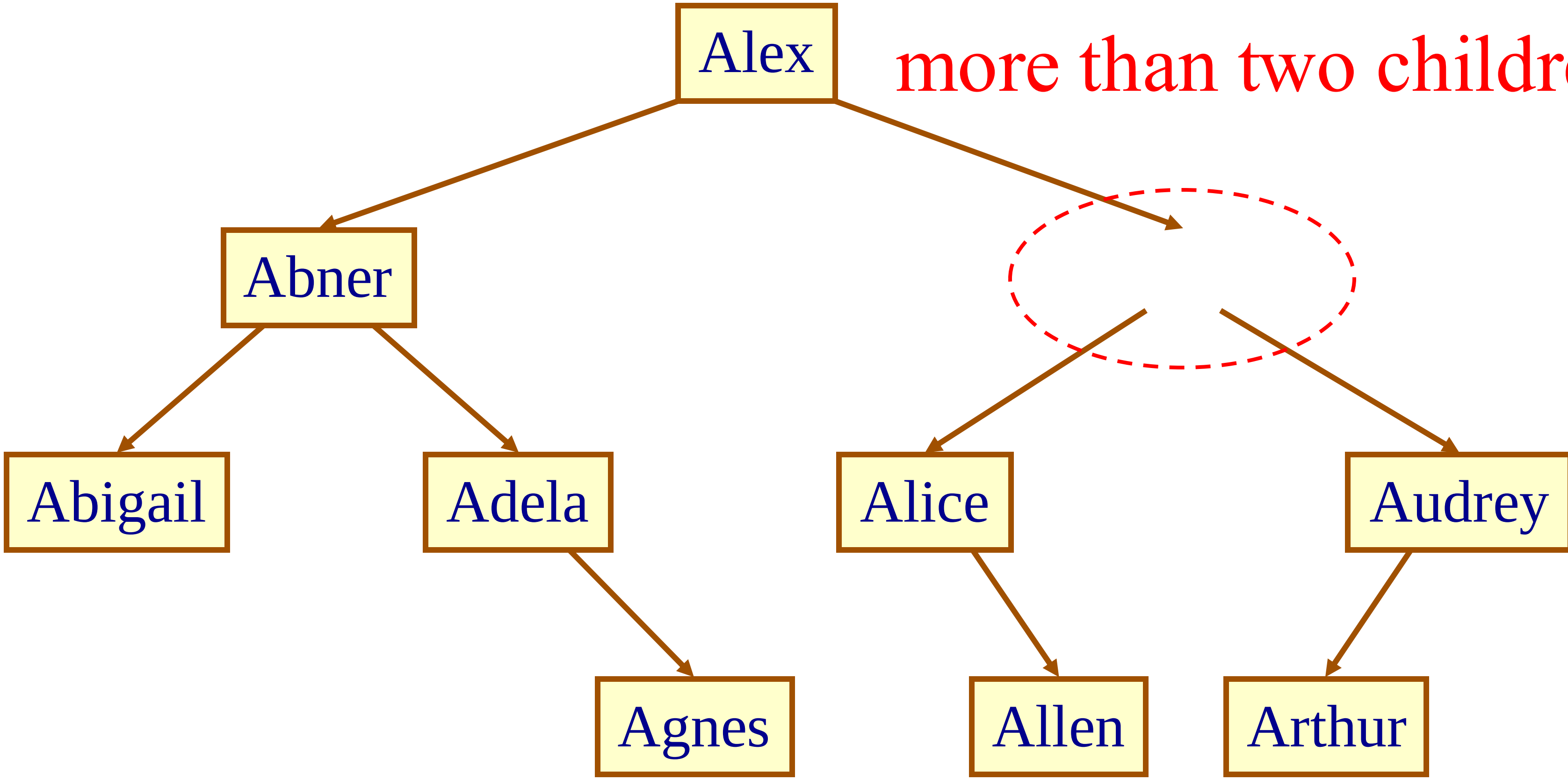
BST: Remove



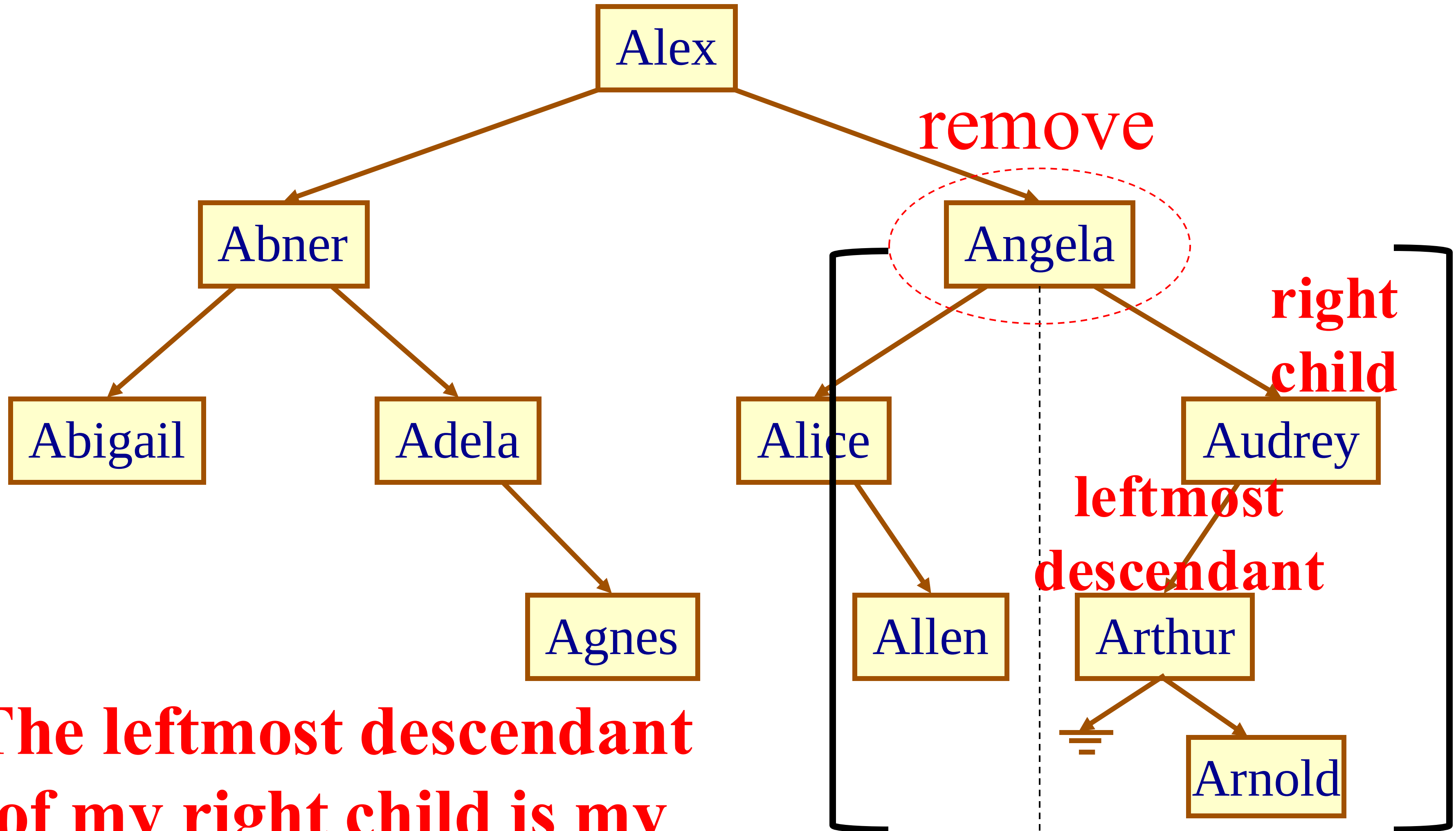
How to remove a node?

BST: Remove

A node cannot have
more than two children



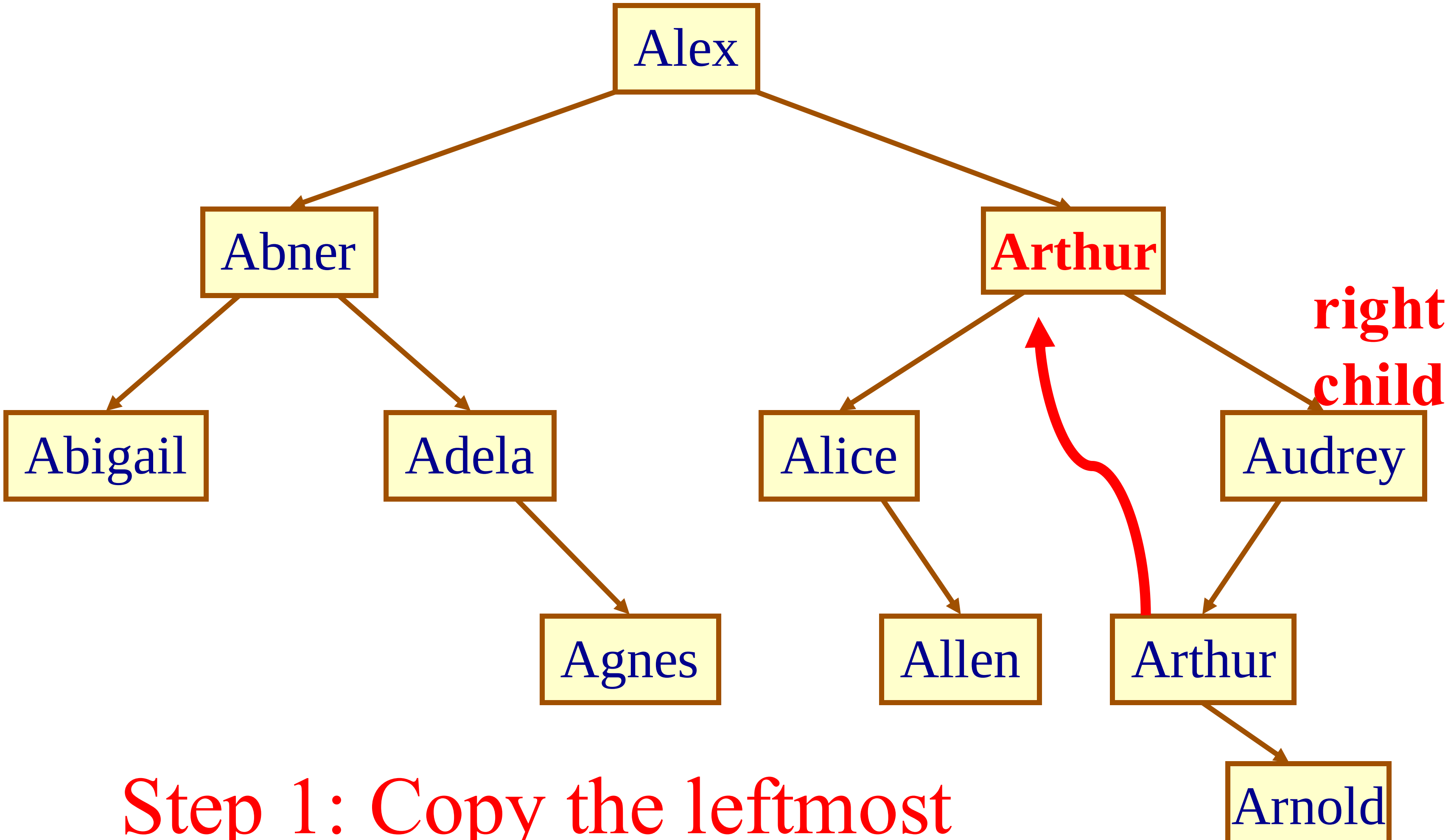
Case 1: Both the right child and its leftmost descendant exist



The leftmost descendant of my right child is my next higher node!

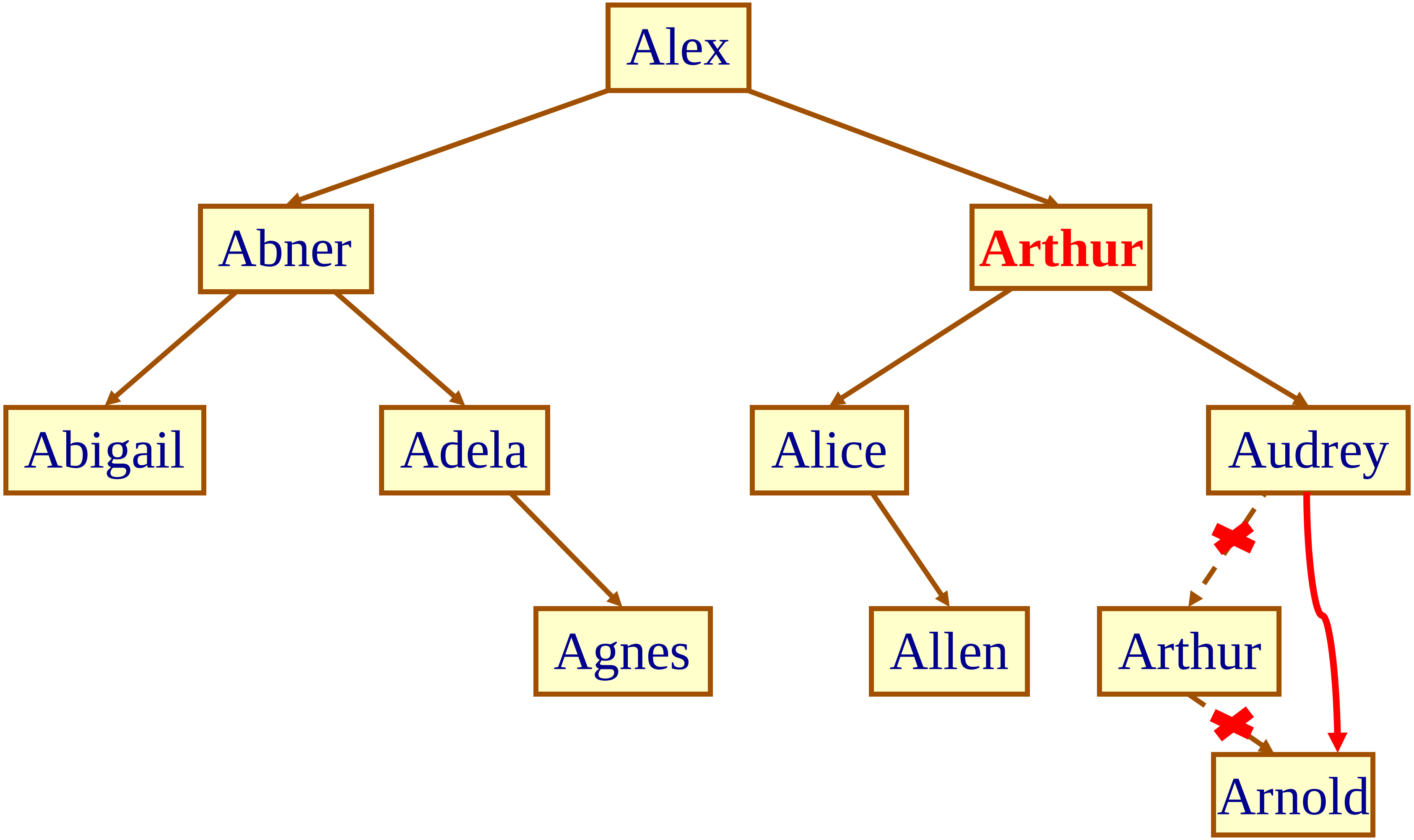
Allen < Angela < Arthur < Arnold

Case 1: Both the right child and its leftmost descendant exist



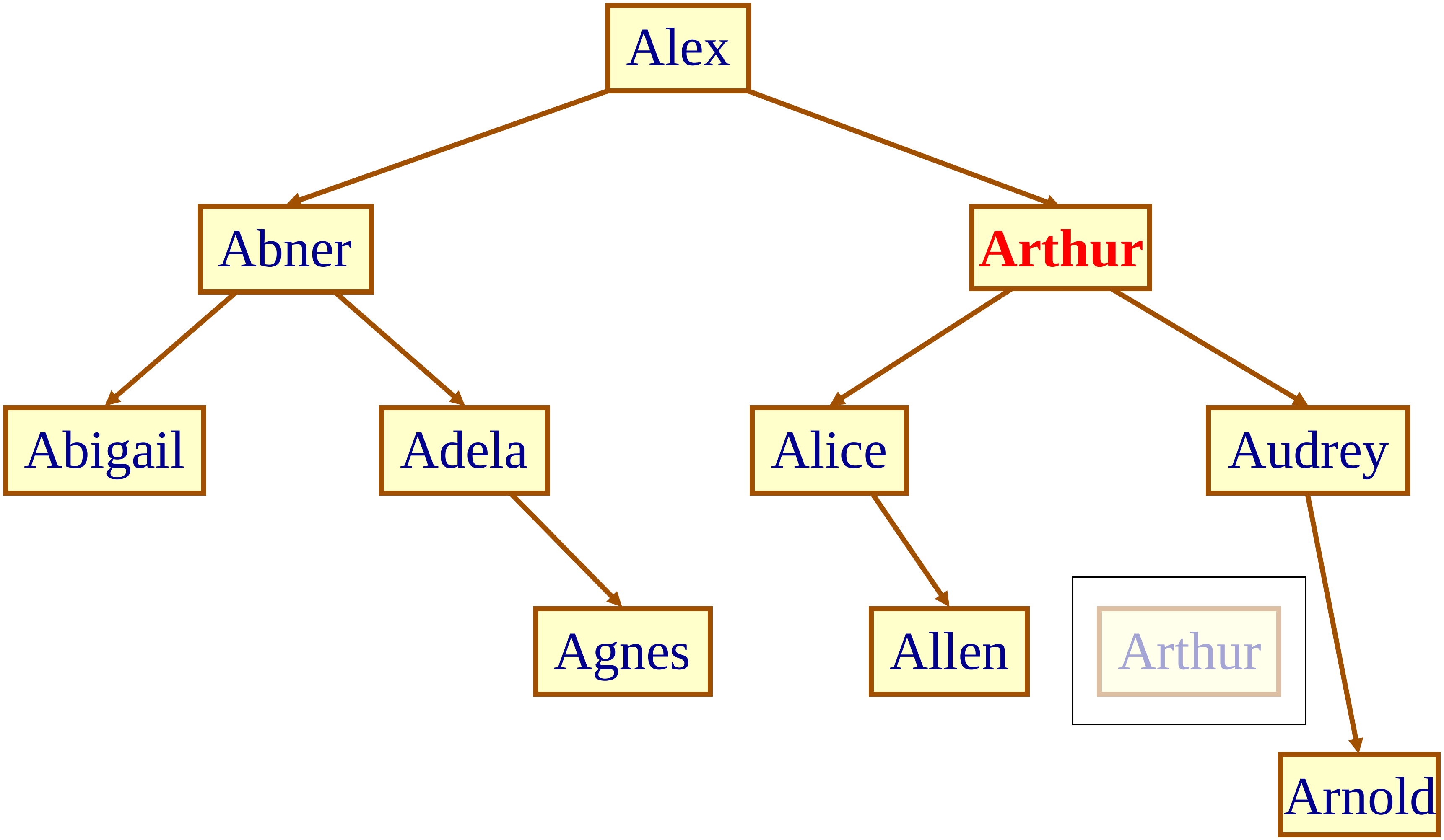
Step 1: Copy the leftmost descendant of the right child

Case 1: Both the right child and its leftmost descendant exist



Step 2: Disconnect the leftmost descendant

Case 1: Both the right child and its leftmost descendant exist

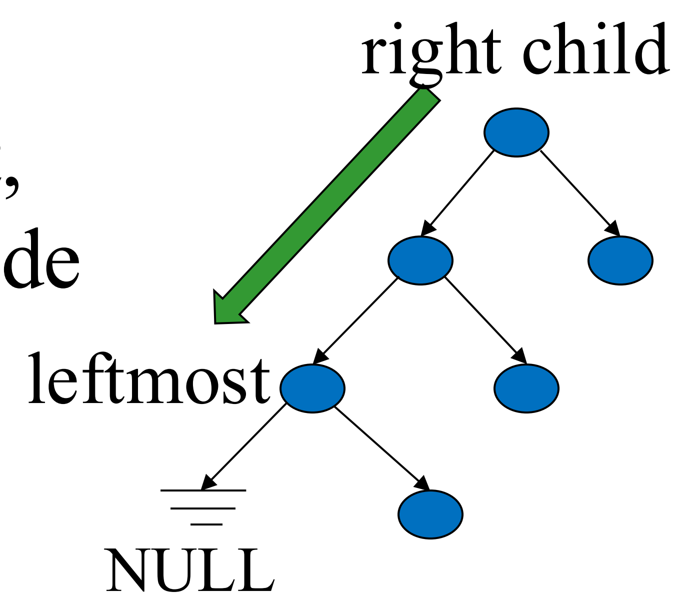


Step 3: Free the leftmost descendant

Leftmost Value: Iterative

```
/*Returns value of the leftmost child*/
TYPE _leftmostVal(struct Node *node) {
  while(node->left != NULL) node=node->left;
  return node->val;
}
```

If the left child does not exist, return the value of the input node

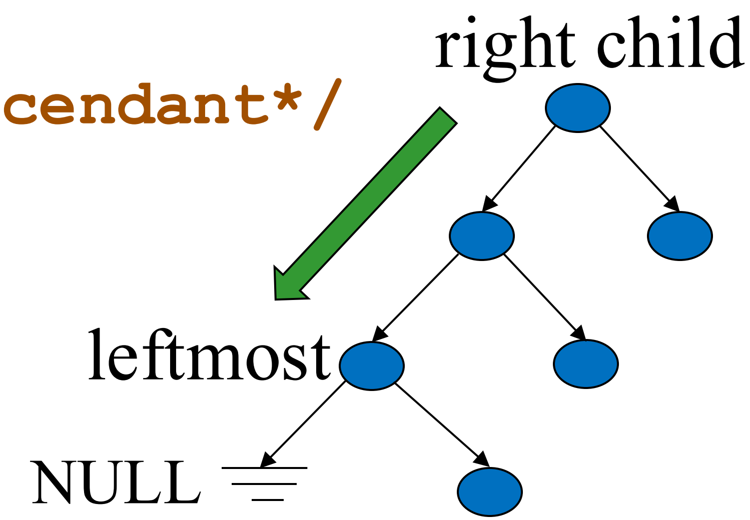


We can use this function for Step 1 in Case 1

Remove Leftmost: Recursive

```
struct Node *_removeLeftmost(struct Node *node)
{  /* find the leftmost descendant */
   if(node->left != NULL){
      node->left = _removeLeftmost(node->left);
      return node;
   }

   /*remove the leftmost descendant*/
   ...
}
```

recursively slide down to the left

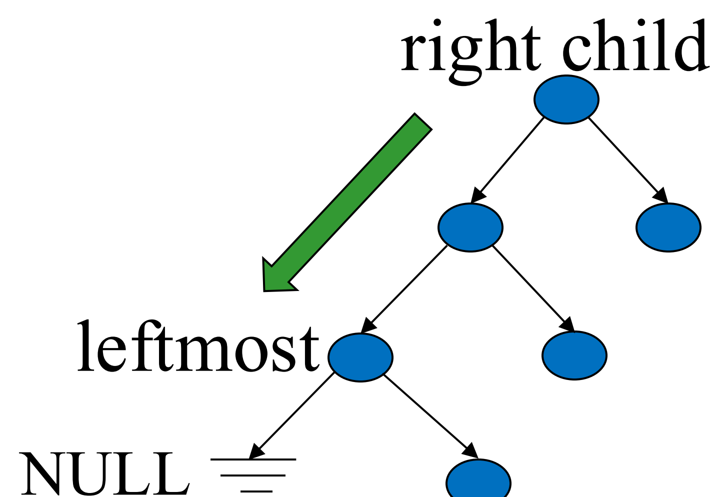


Remove Leftmost: Recursive

```
struct Node *_removeLeftmost(struct Node *node)
{   /* find the leftmost descendant */
    if(node->left != NULL){
        node->left = _removeLeftmost(node->left);
        return node;
    }
    /*remove*/

}
```

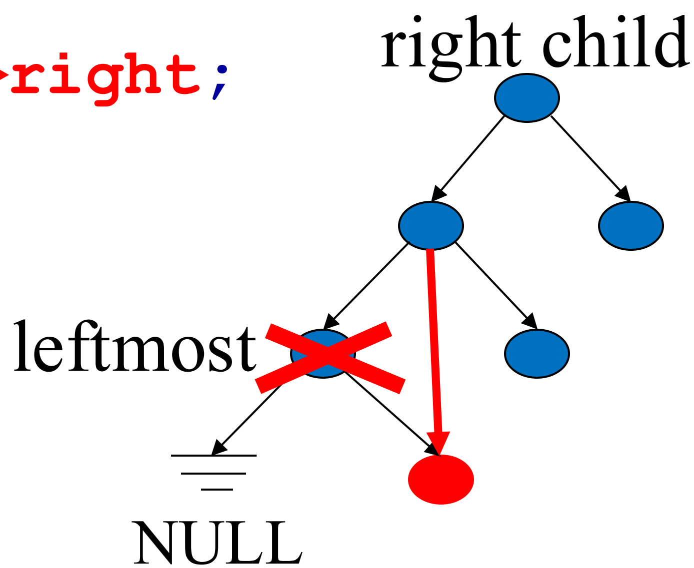
recursively slide down to the left

propagate the changes up the tree



Remove Leftmost: Recursive

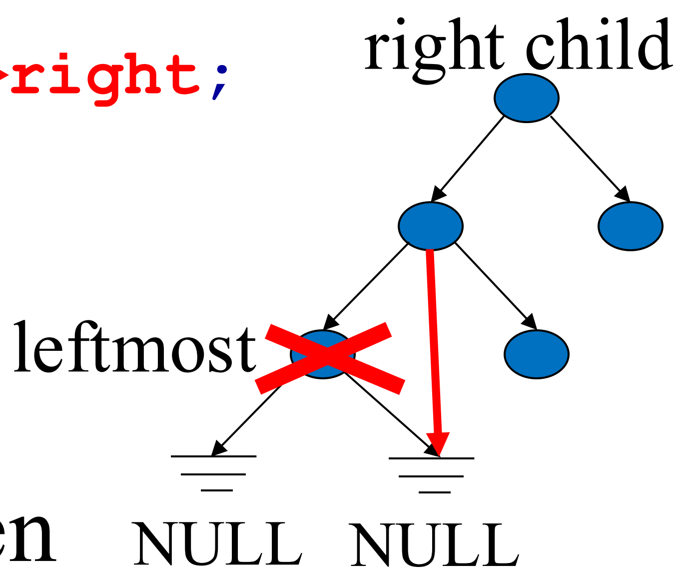
```
struct Node *_removeLeftmost(struct Node *node)
{
    if(node->left != NULL){
        node->left = _removeLeftmost(node->left);
        return node;
    }
    struct Node *temp = node->right;
    free(node);
    return temp;
}
```



return the right child of the leftmost descendant

Remove Leftmost: Recursive

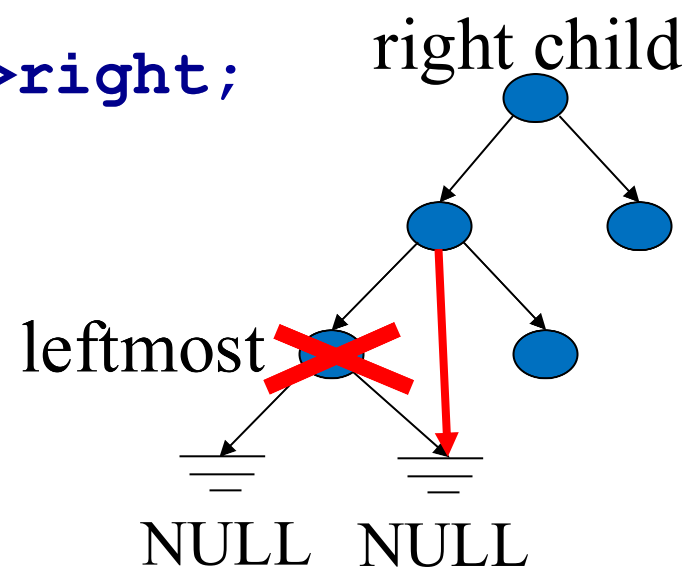
```
struct Node *_removeLeftmost(struct Node *node)
{
    if(node->left != NULL){
        node->left = _removeLeftmost(node->left);
        return node;
    }
    struct Node *temp = node->right;
    free(node);
    return temp;
}
```



Special case: no children

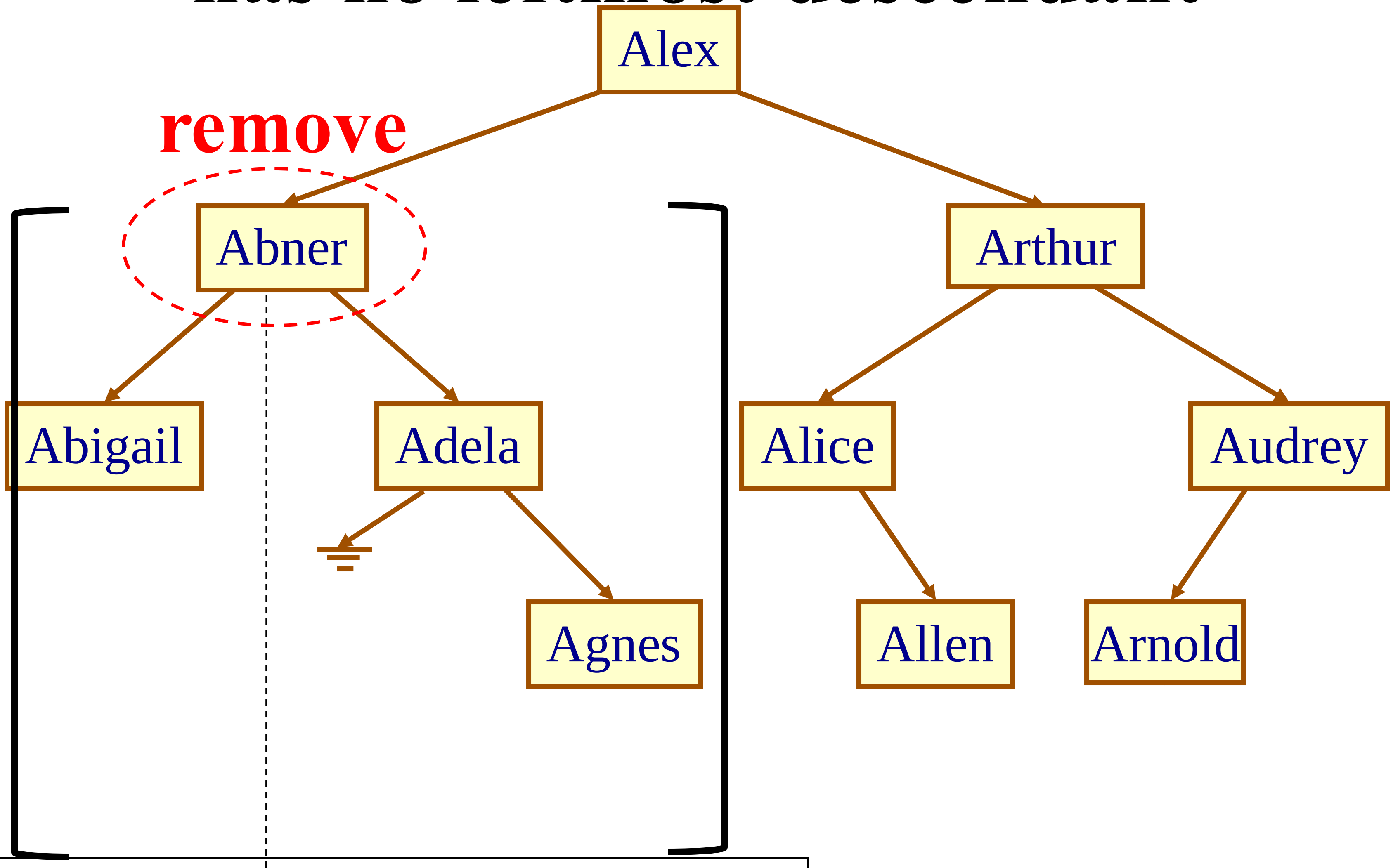
Remove Leftmost: Recursive

```
struct Node *_removeLeftmost(struct Node *node)
{
    if(node->left != NULL){
        node->left = _removeLeftmost(node->left);
        return node;
    }
    struct Node *temp = node->right;
    free(node);
    return temp;
}
```



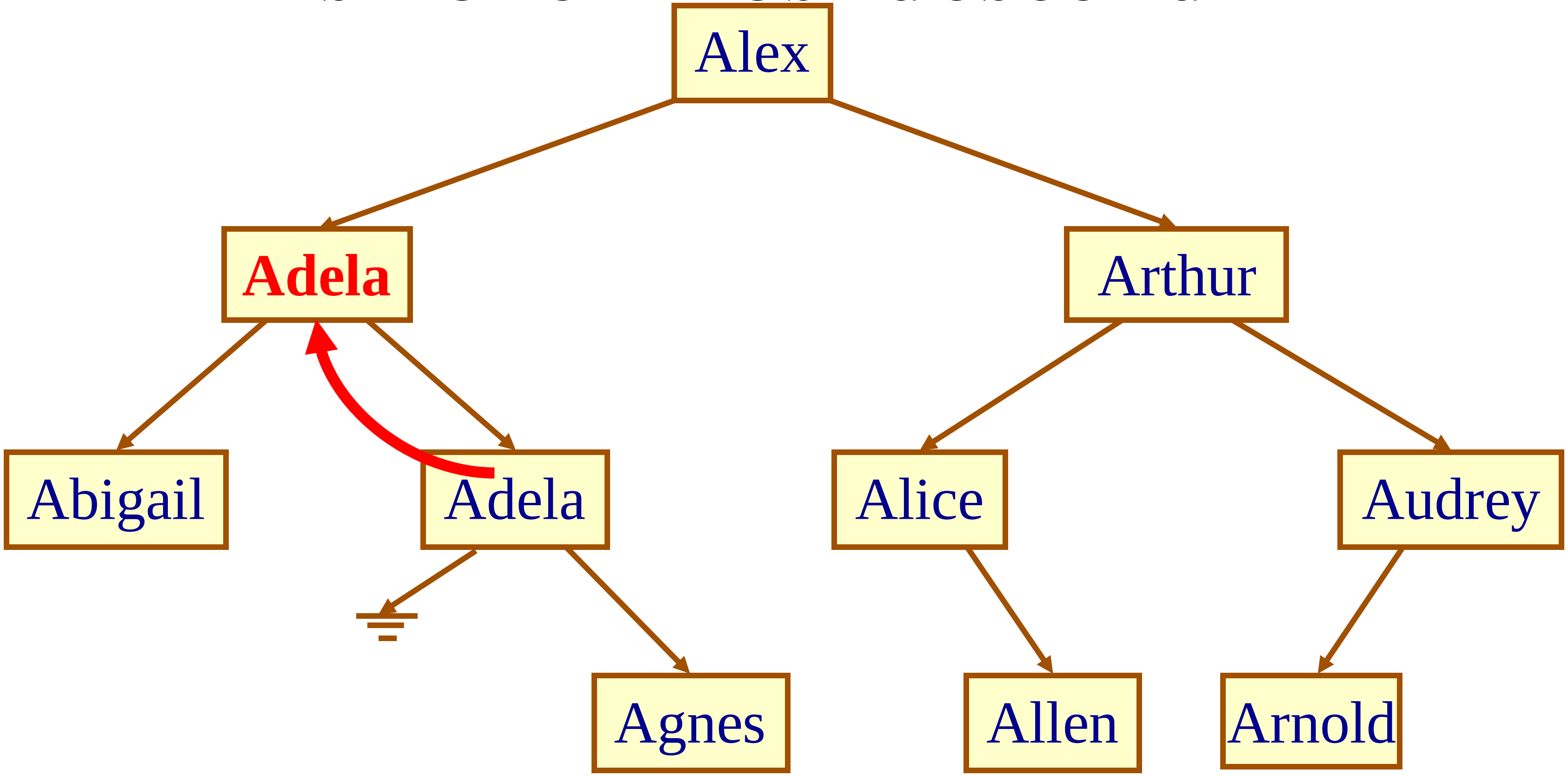
We can use this function for Steps 2-3 in Case 1

Case 2: The right child exists but has no leftmost descendant



Abigail < Abner < Adela < Agnes

Case 2: The right child exists but has no leftmost descendant

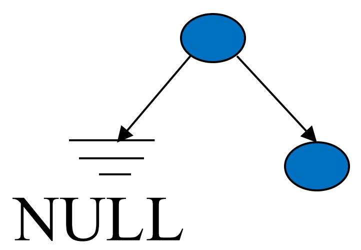


Step 1: Copy the right child

Use the Same Function in Case 2

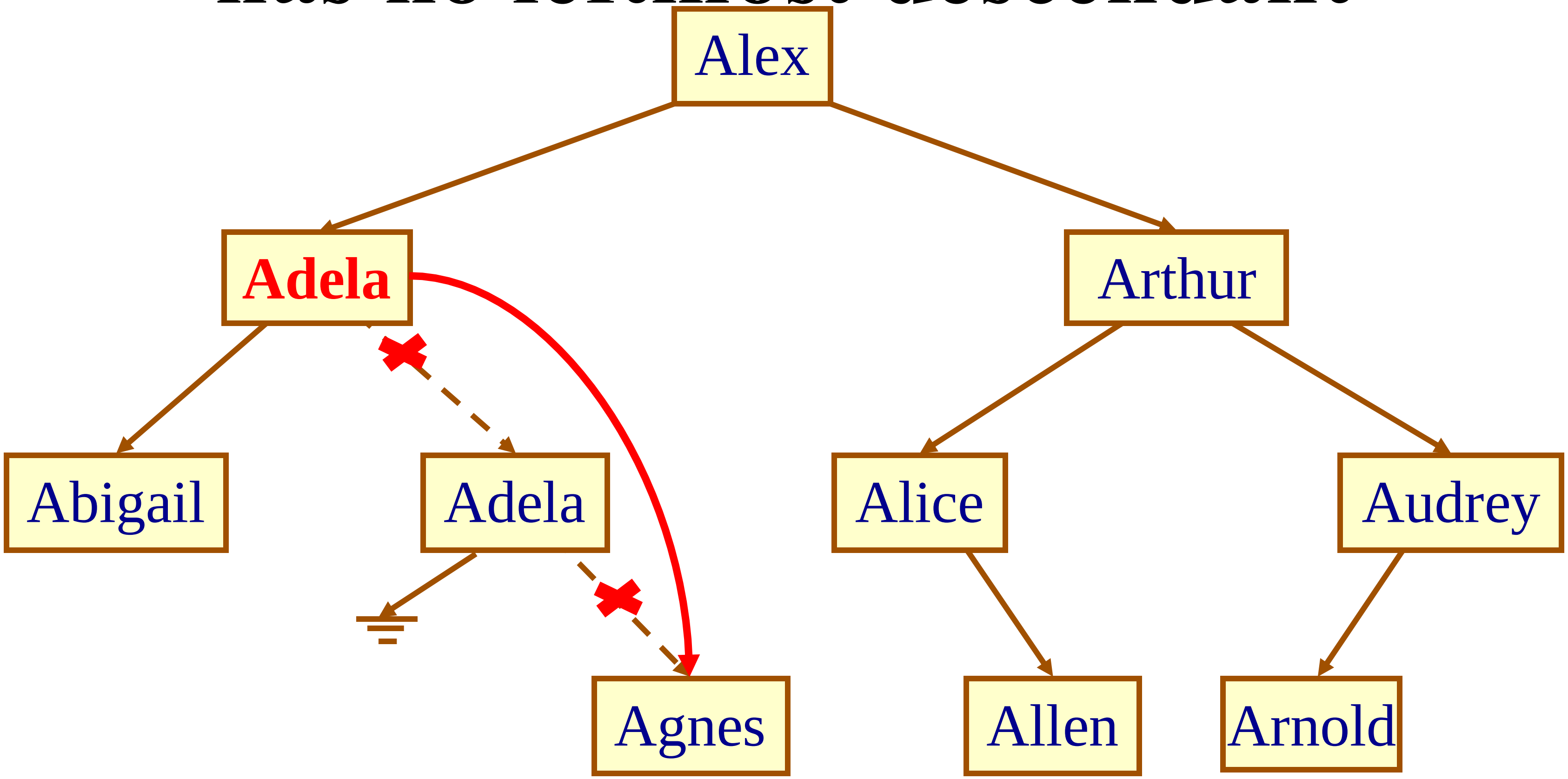
```
/*Returns value of the leftmost child*/
TYPE _leftmostVal(struct Node *node) {
  while(node->left != NULL) node=node->left;
  return node->val;
}
```

If the leftmost descendant of the right child does not exist, we will return the value of the right child



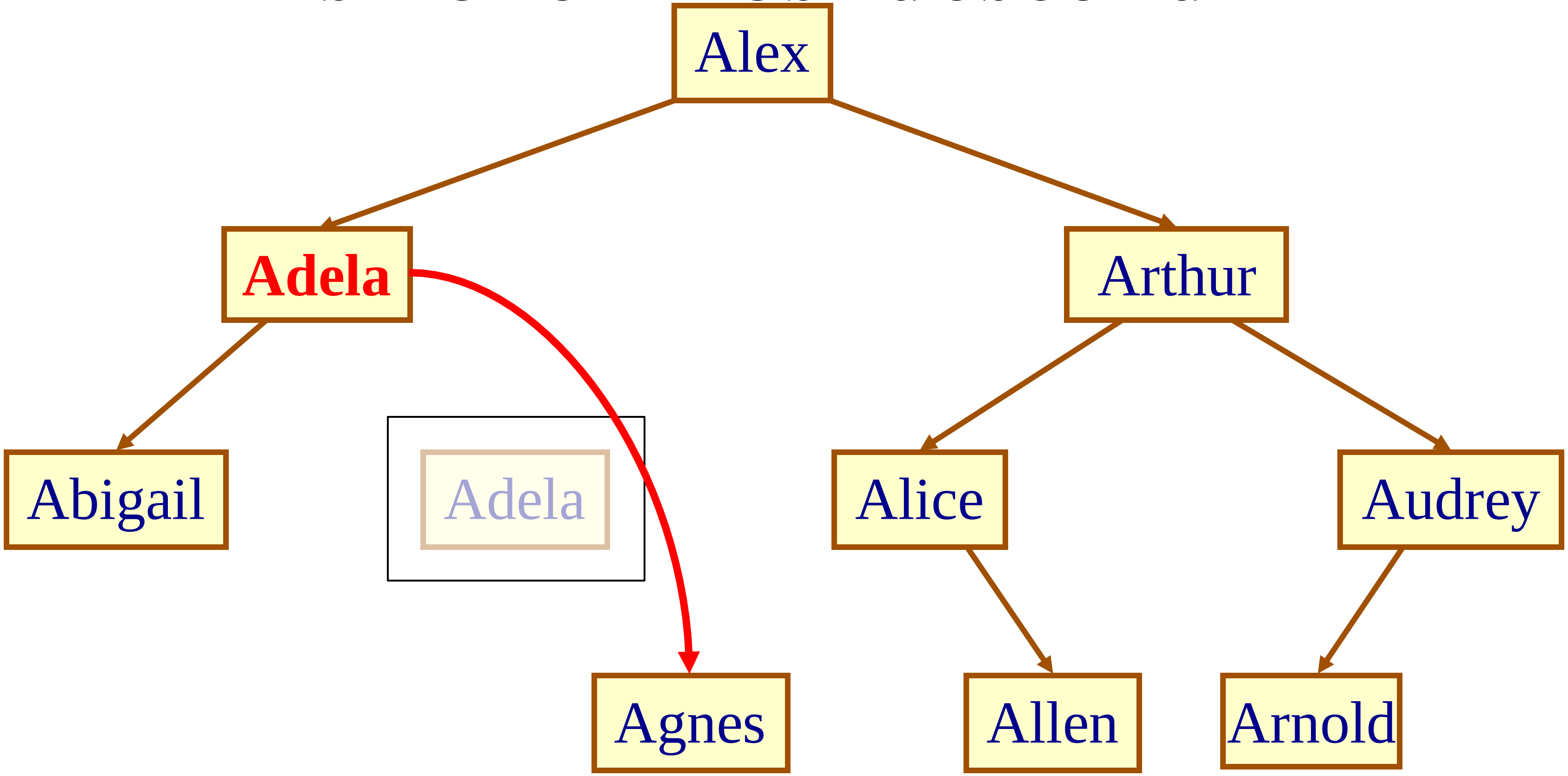
We can use this function for Step 1 in Case 2

Case 2: The right child exists but has no leftmost descendant



Step 2: Disconnect the right child

Case 2: The right child exists but has no leftmost descendant

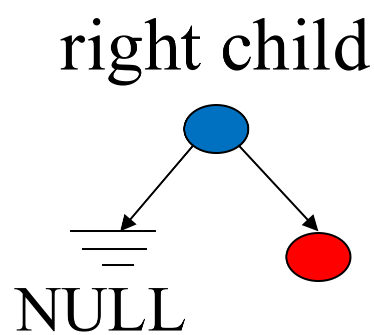


Step 3: Free the right child

Use the Same Function in Case 2

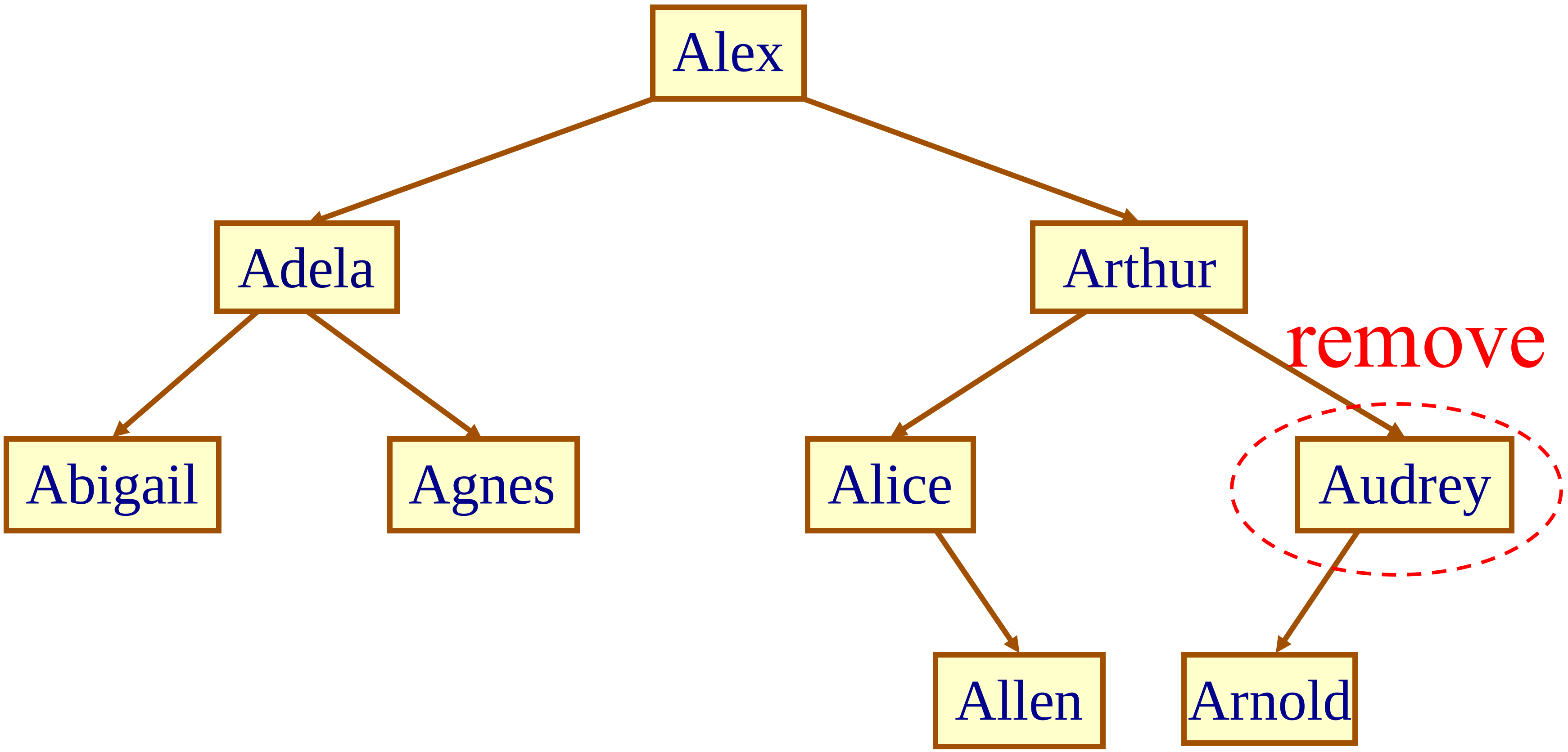
```
NODE *_removeLeftmost(NODE *node) {
    if(node->left != NULL){
        node->left = _removeLeftmost(node->left);
        return node;
    }
    struct Node *temp = node->right;
    free(node);
    return temp;
}
```

If the leftmost descendant of the right child does not exist, we will return the **right grandchild**

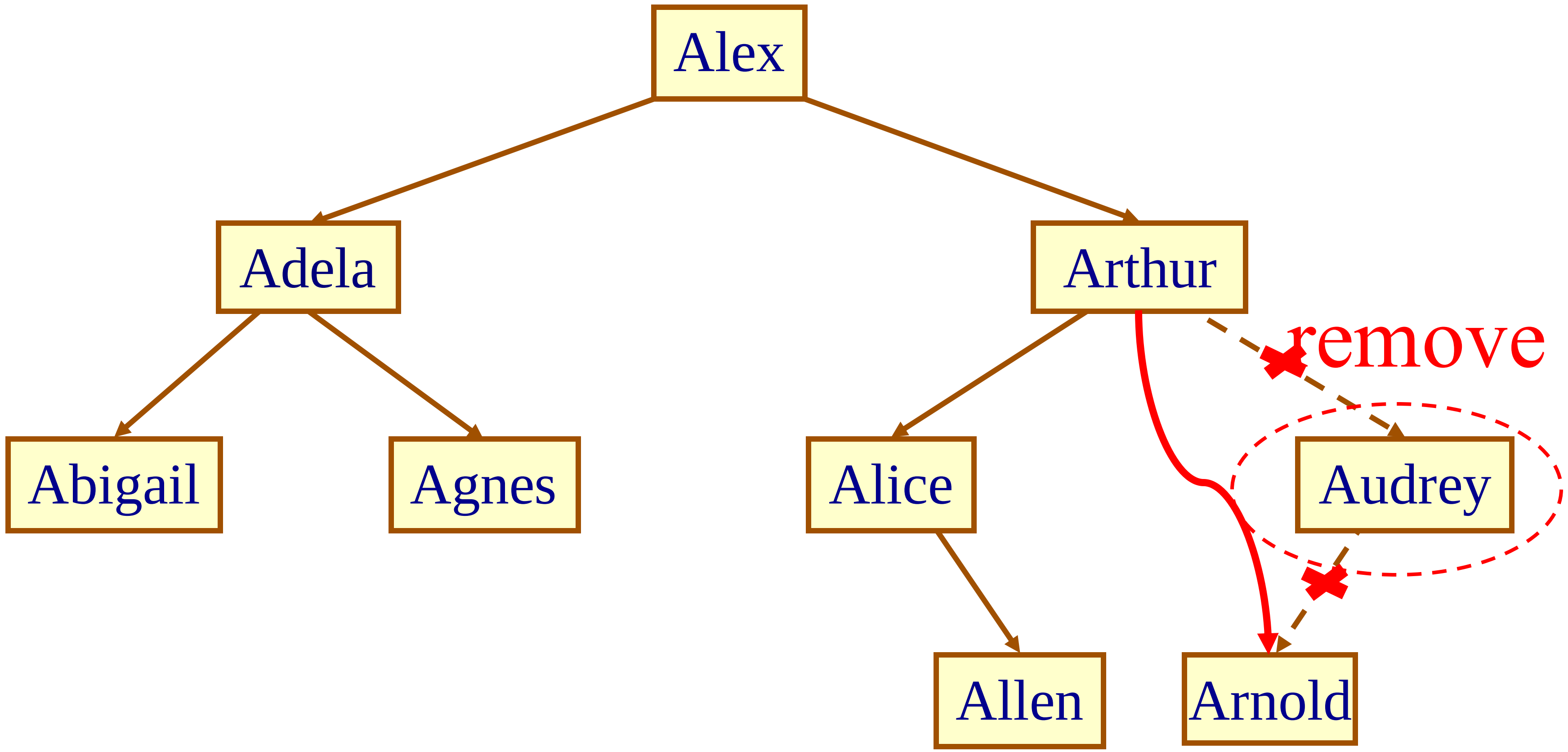


We can use this function for Steps 2-3 in Case 2

Case 3: The right child does not exist

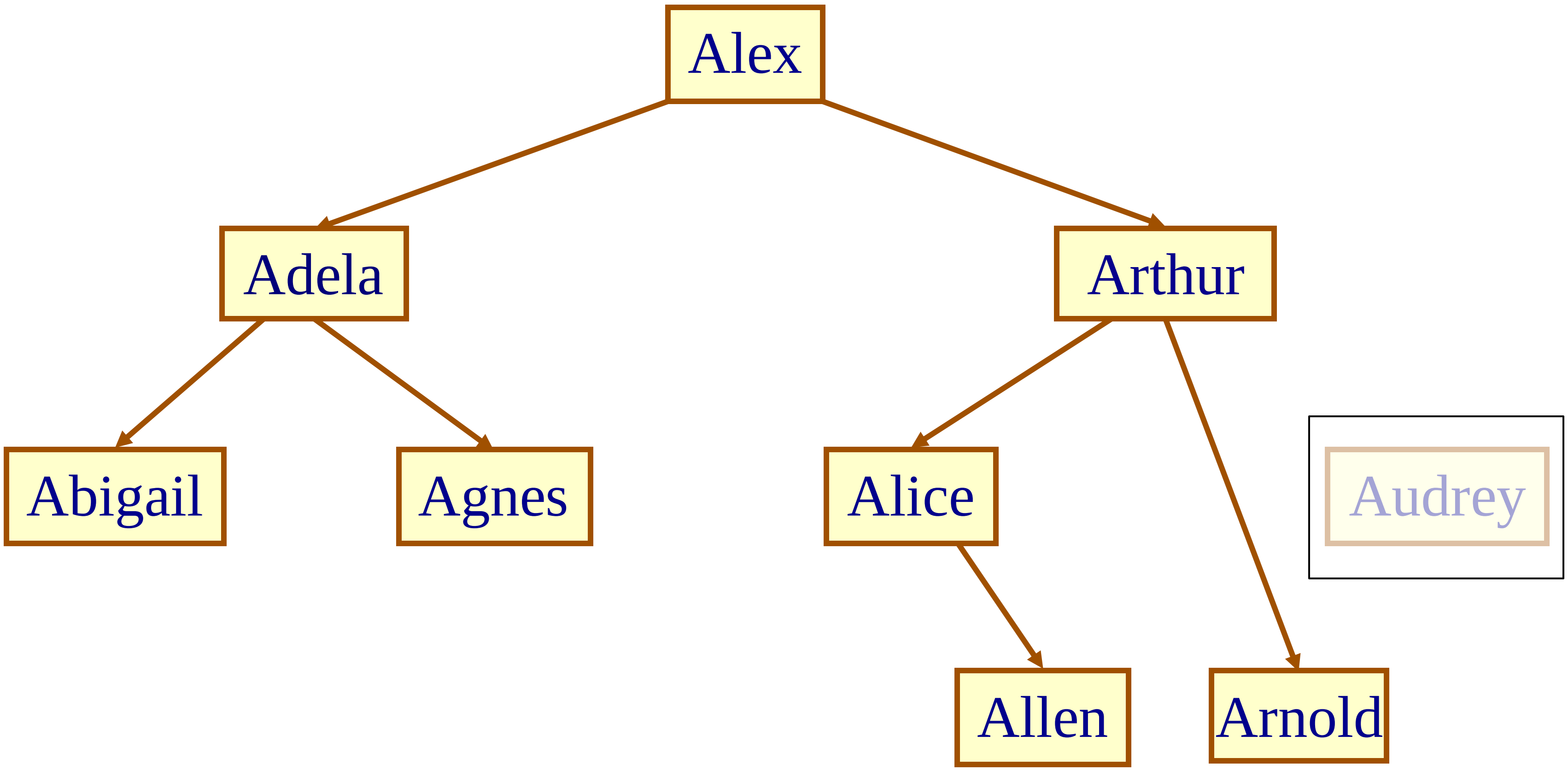


Case 3: The right child does not exist



Step 1: Disconnect the node

Case 3: The right child does not exist



Step 2: Free the node

We Need Two Functions for Remove

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

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

same pointer

Complexity: O(?)

We Need Two Functions for Remove

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

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

same pointer

In case, if `containsBST()` is not available

```
#define NODE struct Node
NODE *removeNode(NODE *node, TYPE e){
  if(!node) return NULL;
  if (EQ(node->val, e))
  {
    ... /*remove node; consider the 3 cases*/
  }
  else{
    ... /*recursive calls*/
  }
  return node;
}
```

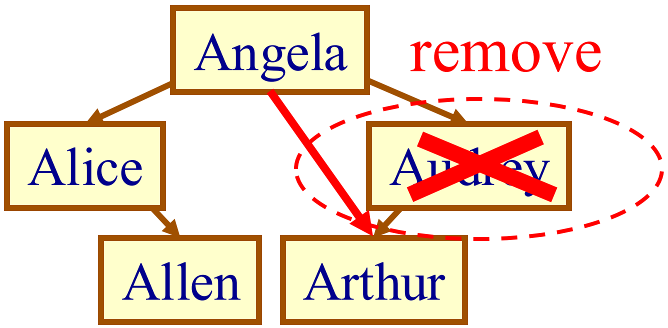
same pointer

```
NODE *removeNode(NODE *node, TYPE e){
  if(!node) return NULL;
  struct Node *temp;
  if (EQ(e, node->val))
    /* remove node; consider the 3 cases */
  else
    /* not found, so recursive calls */
    if(LT(e, node->val))
      node->left = removeNode(node->left, e);
    else
      node->right = removeNode(node->right, e);

  return node;
}
```

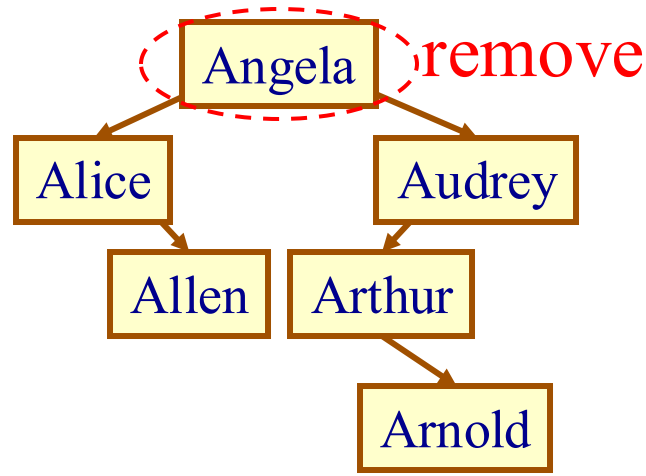
same pointer

```
/* val is in the node, remove the node */
if (EQ(e, node->val)){
  if(node->right == NULL)

    /* Case 3: no right child */

  else

    /* Cases 1 & 2: right child exists */

}
else{
   /* not found, so recursive calls */
...
```

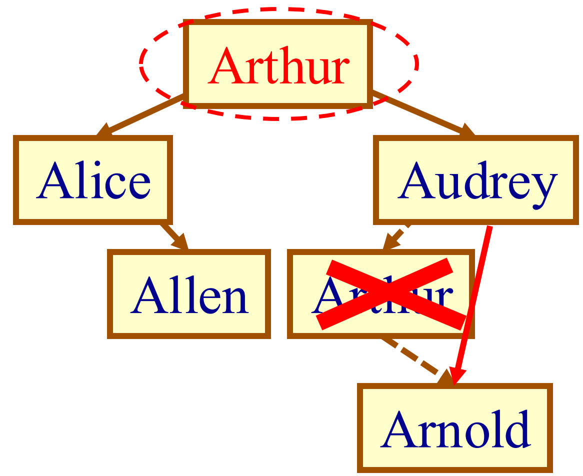
```
if (EQ(e, node->val)){/*val found*/
    if(node->right == NULL){
        /* Case 3: no right child*/
        /* return the left child */
        temp = node->left;
        free(node);
        return temp;
    }
    else
    {
      /*Cases 1 & 2: right child exists*/
    }
   ...
```



```
if (EQ(e, node->val)){/*val found*/
  if(node->right == NULL){ /*Case 3*/
    temp = node->left;
    free(node);
    return temp;
  }
  else{/*Cases 1 & 2*/
    /*copy the leftmost descendant*/
    node->val = _leftmostVal(node->right);
    /*remove the leftmost descendant*/
    node->right =
               _removeLeftmost(node->right);
  } ...
```



```c
if (EQ(e, node->val)){/*val found*/
  if(node->right == NULL){ /*Case 3*/
    temp = node->left;
    free(node);
    return temp;
  }
  else{/*Cases 1 & 2*/
    /*copy the leftmost descendant*/
    node->val = _leftmostVal(node->right);
    /*remove the leftmost descendant*/
    node->right =
              _removeLeftmost(node->right);
  } ...
```



Remove All BST

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

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

```
void removeAllBST(BST *tree, TYPE e){
  assert(tree);
  if(tree->root) /*check if tree is empty*/
    tree->root =
    removeAllNode(tree->root, e, &(tree->size));
  }
}
```

We must maintain tree size for each node removal

Remove All BST

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

same pointer

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

```
void removeAllBST(BST *tree, TYPE e){
   assert(tree);
   if(tree->root) /*check if tree is empty*/
      tree->root =
      removeAllNode(tree->root, e, &(tree->size));
   }
}
```

same pointer

Recursive Remove All Node

```
struct Node *removeAllNode
 (struct Node *node, TYPE e, int *treesize){

 /* We'll use the pre-order depth-first
    traversal to search for e top-down,
    until the node == NULL. */
    if(node == NULL) return NULL;

   ...

   return node;
}
```

to maintain tree size for each node removal

Recursive Remove All Node

```
struct Node *removeAllNode
  (struct Node *node, TYPE e, int *treesize){

  /* We'll use the pre-order depth-first
     traversal to search for e top-down,
     until the node == NULL. */
     if(node == NULL) return NULL;

   ...

   return node;
}
```

to propagate the changes up to the parent node

```
Node *removeAllNode(Node *node, TYPE e, int *treesize){
  while(node != NULL && EQ(e, node->val)){

     /*remove this node; consider all edge cases*/

     (*treesize)--; /* for each removal */
  }
  if( node != NULL && LT(e, node->val))
    node->left = removeAllNode(node->left, e, treesize);
  else if (node != NULL)
    node->right = removeAllNode(node->right, e, treesize);

   return node;
}
```

```
struct Node *temp;
while(node != NULL && EQ(e, node->val)){
      /*remove this node; consider all edge cases*/
      if(node->right == NULL){
            /* Case 3: there is no right child */
            temp = node->left;
            free(node);
            node = temp;
      }
      else
            /* Cases 1 & 2: the right child exists */




      (*treesize)--; /* for each removal */
}
```

```
struct Node *temp;
while(node != NULL && EQ(e, node->val)){
      /*remove this node; consider all edge cases*/
      if(node->right == NULL){/* Case 3 */
         temp = node->left;
         free(node);
         node = temp;
      }
      else{/* Cases 1 & 2 */
         node->val = _leftmostVal(node->right);
         node->right = _removeLeftmost(node->right);
      }
      (*treesize)--; /* for each removal */

}/* end while loop */
```