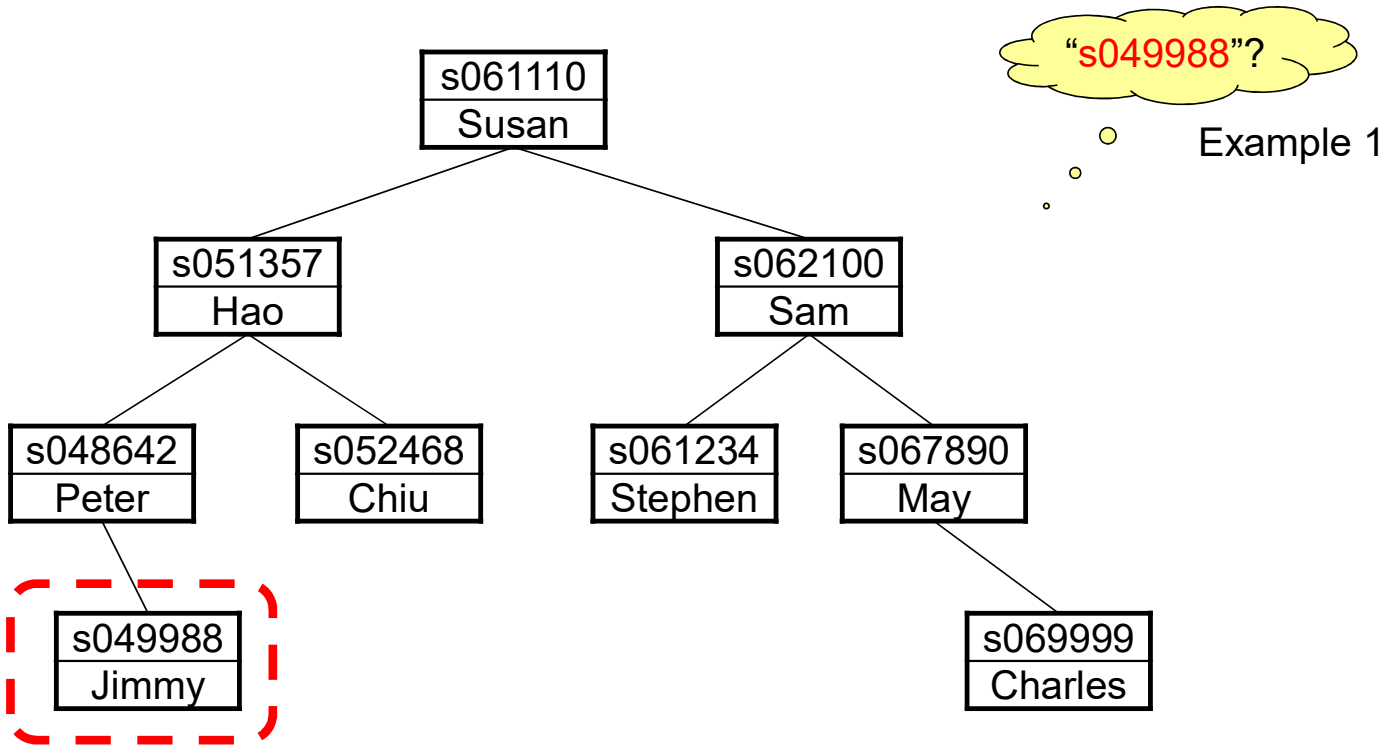
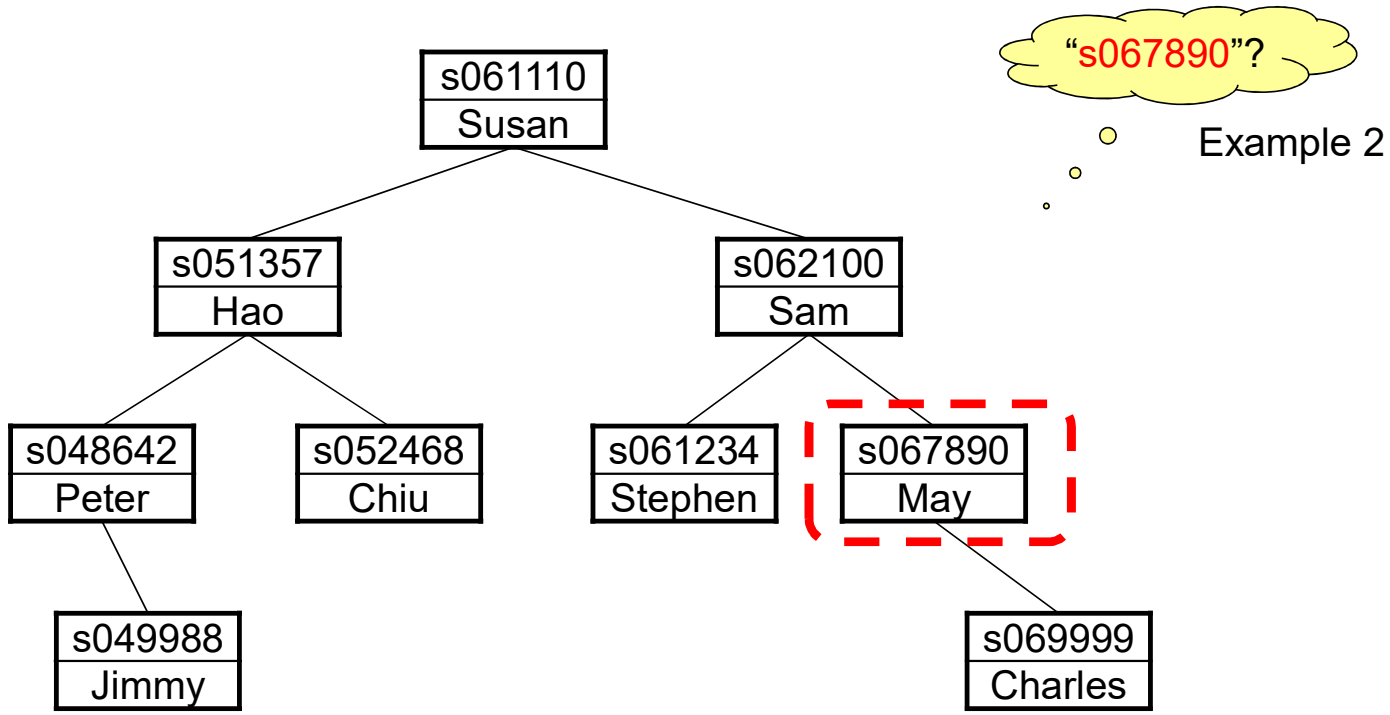
BST Operation: Searching

- Finds a node that matches a particular search key.



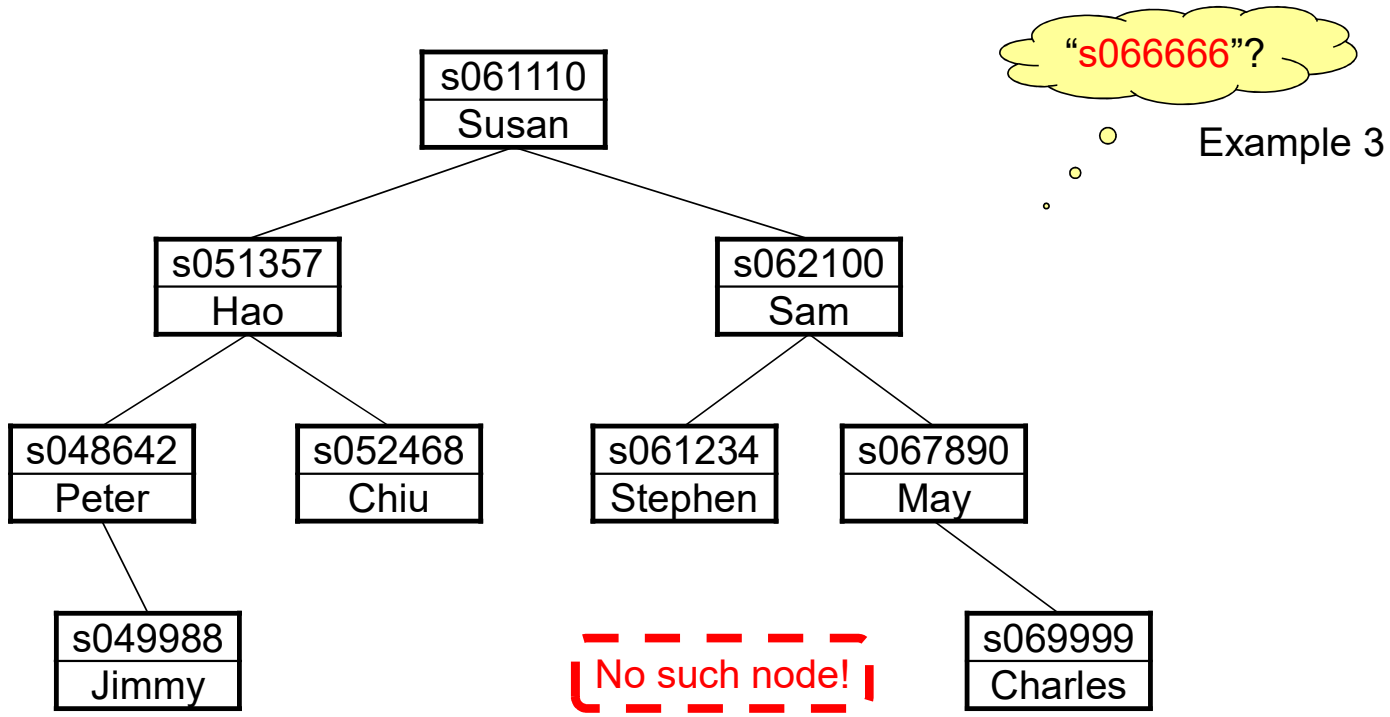
BST Operation: Searching

- Finds a node that matches a particular search key.

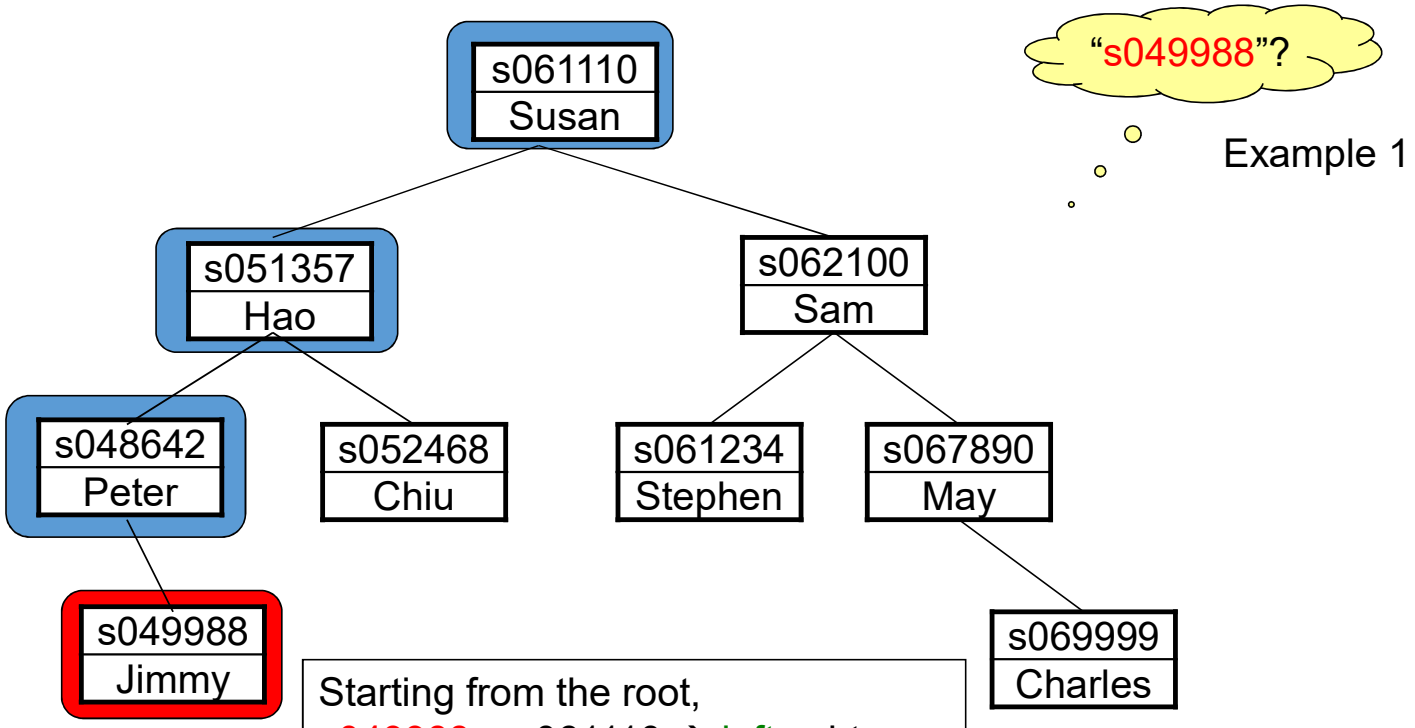


BST Operation: Searching

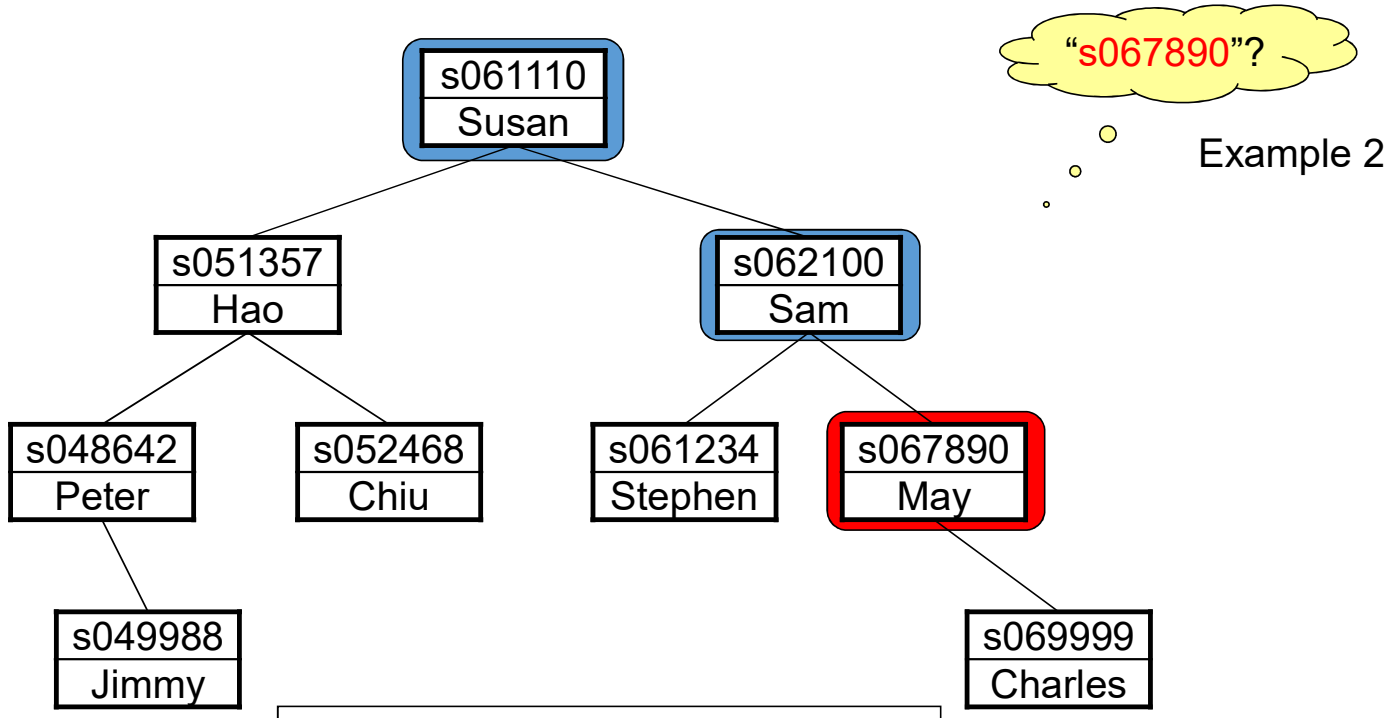
- Finds a node that matches a particular search key.



Searching a BST in Details



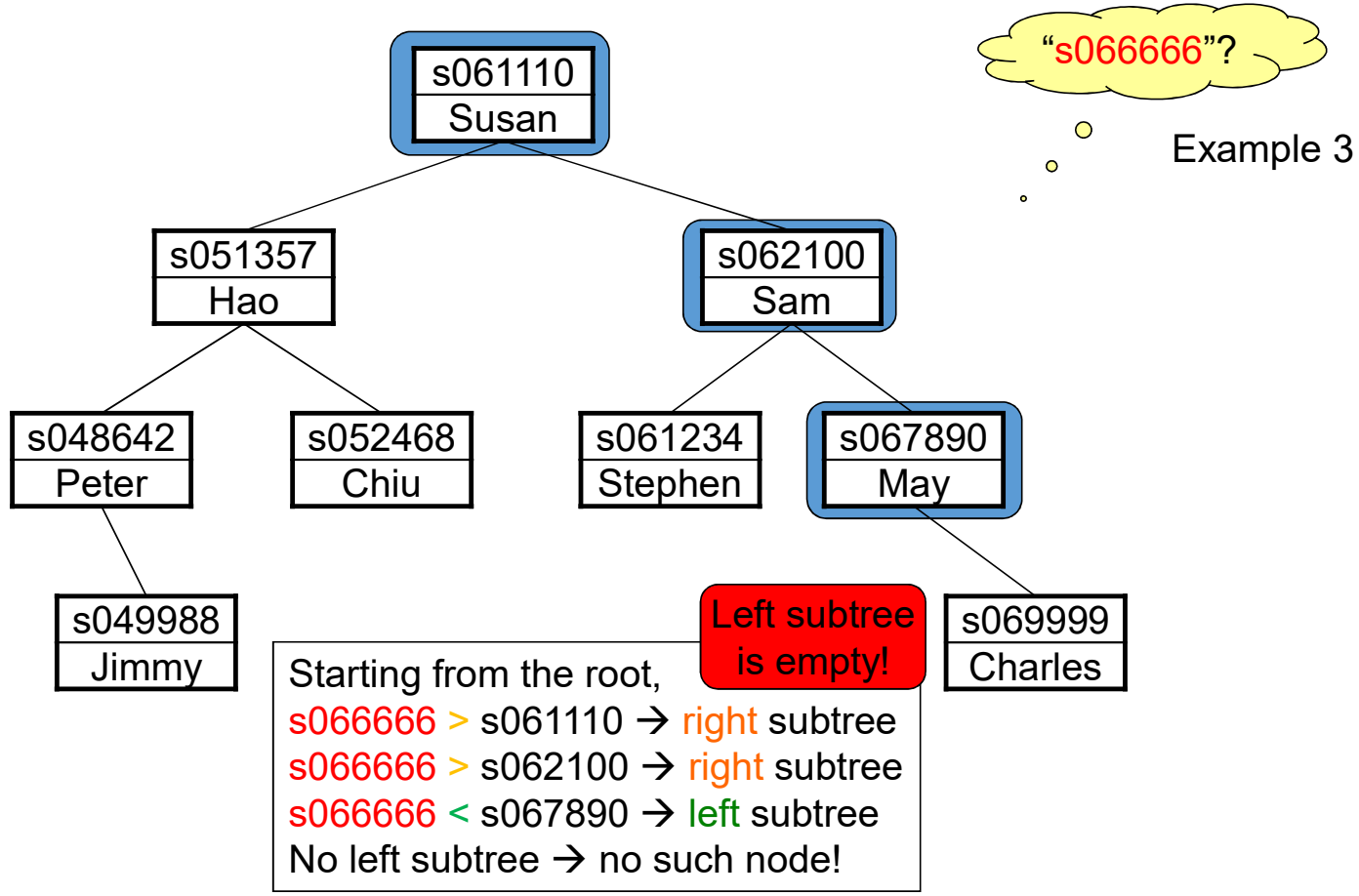
Searching a BST in Details



Starting from the root,
s067890 > s061110 → right subtree
s067890 > s062100 → right subtree
s067890 = s067890 → found!

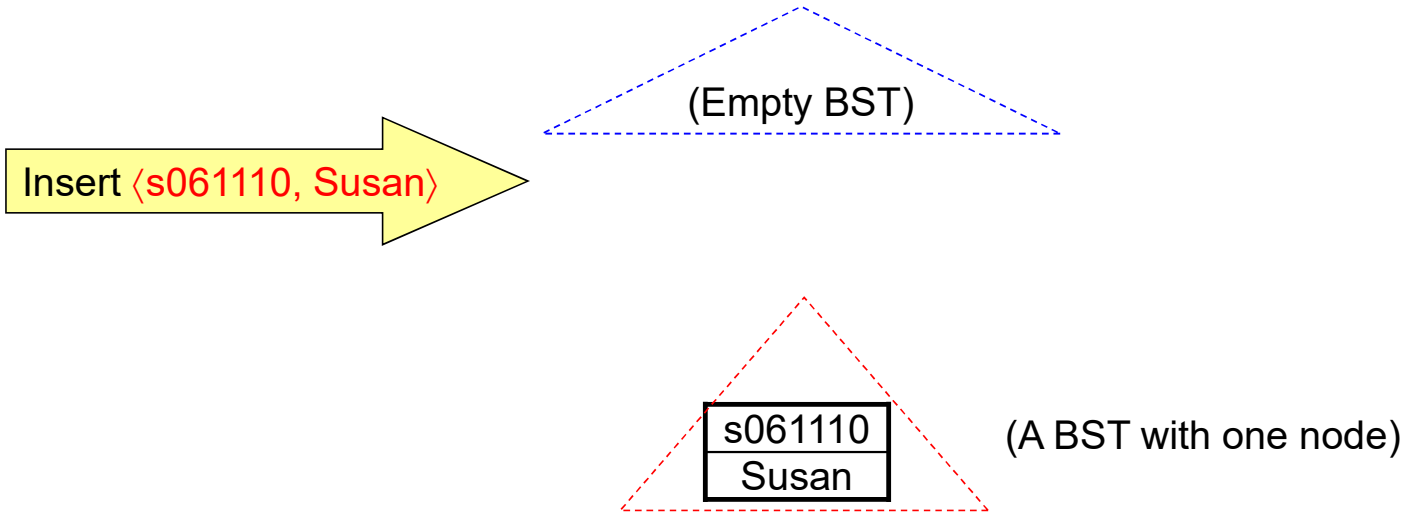
Searching a BST in Detailsb

The average searching time in a BST is $O(\log N)$



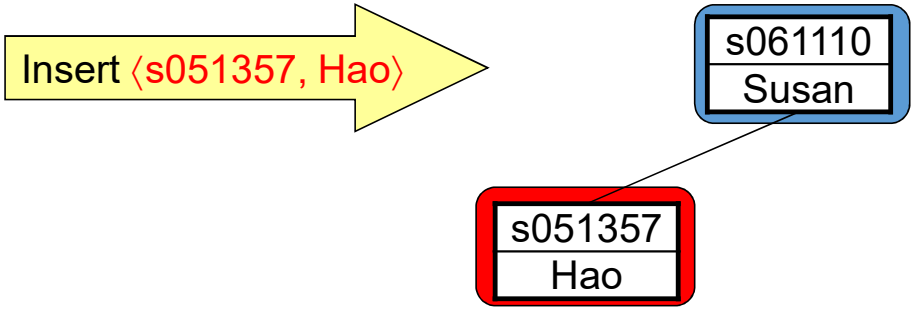
BST Operation: Insertion

- Inserts a node to an existing BST.



BST Operation: Insertion

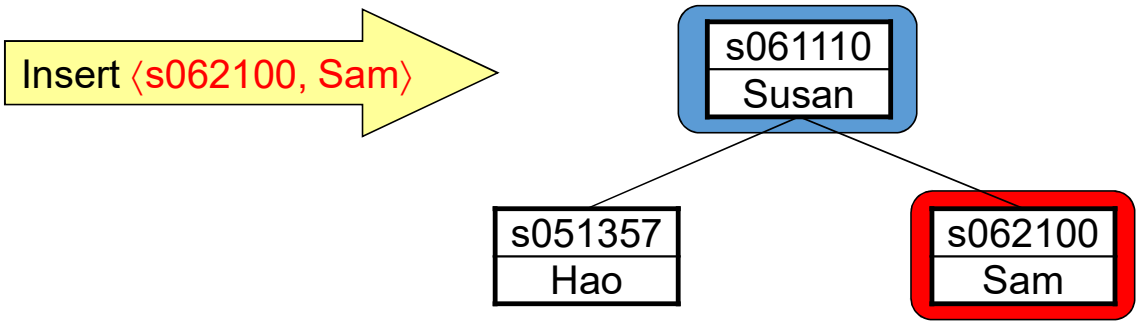
- Inserts a node to an existing BST.



s051357 < s061110 → left subtree

BST Operation: Insertion

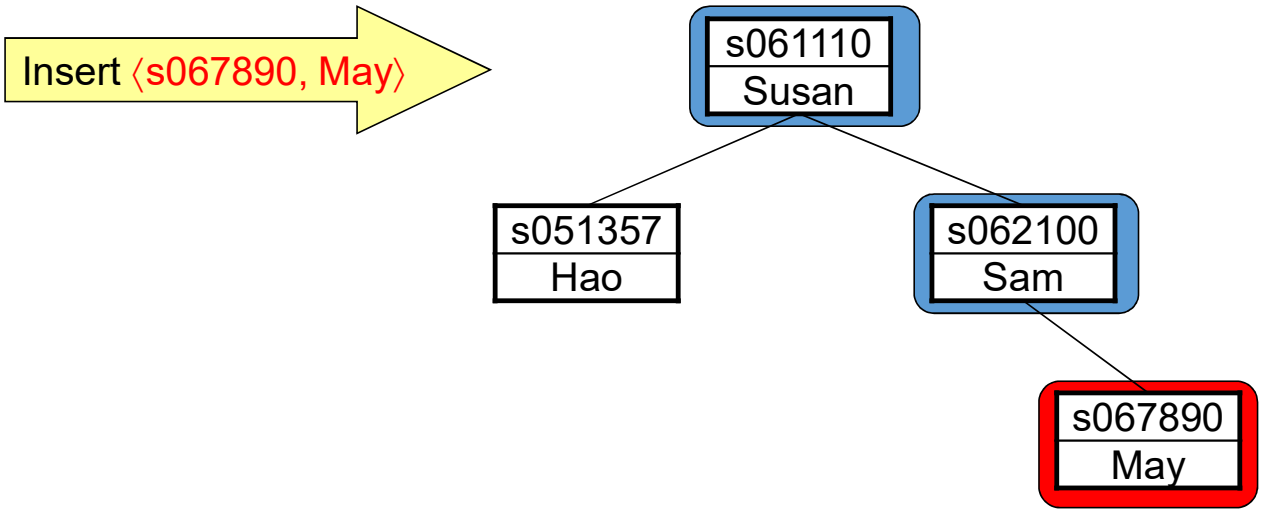
- Inserts a node to an existing BST.



s062100 > s061110 → right subtree

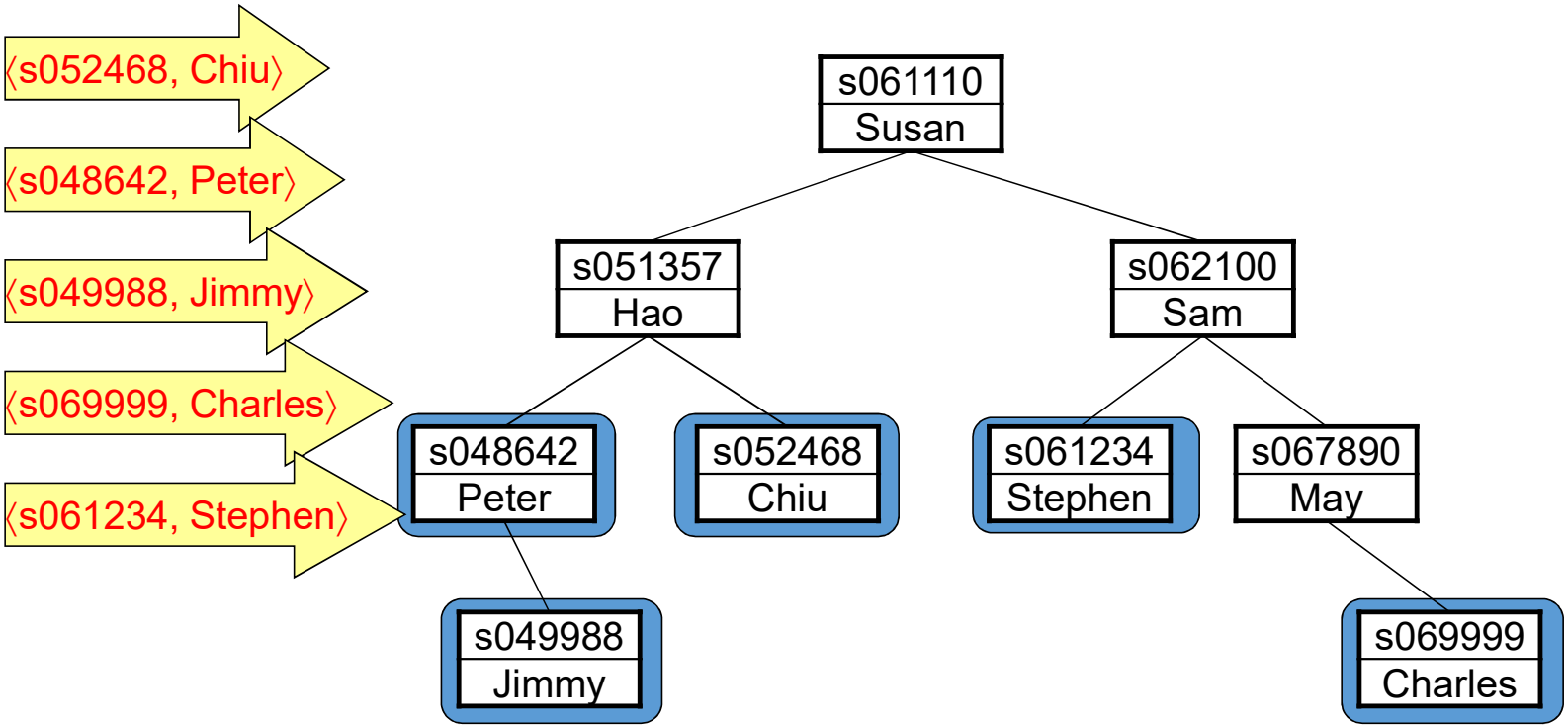
BST Operation: Insertion

- Inserts a node to an existing BST.



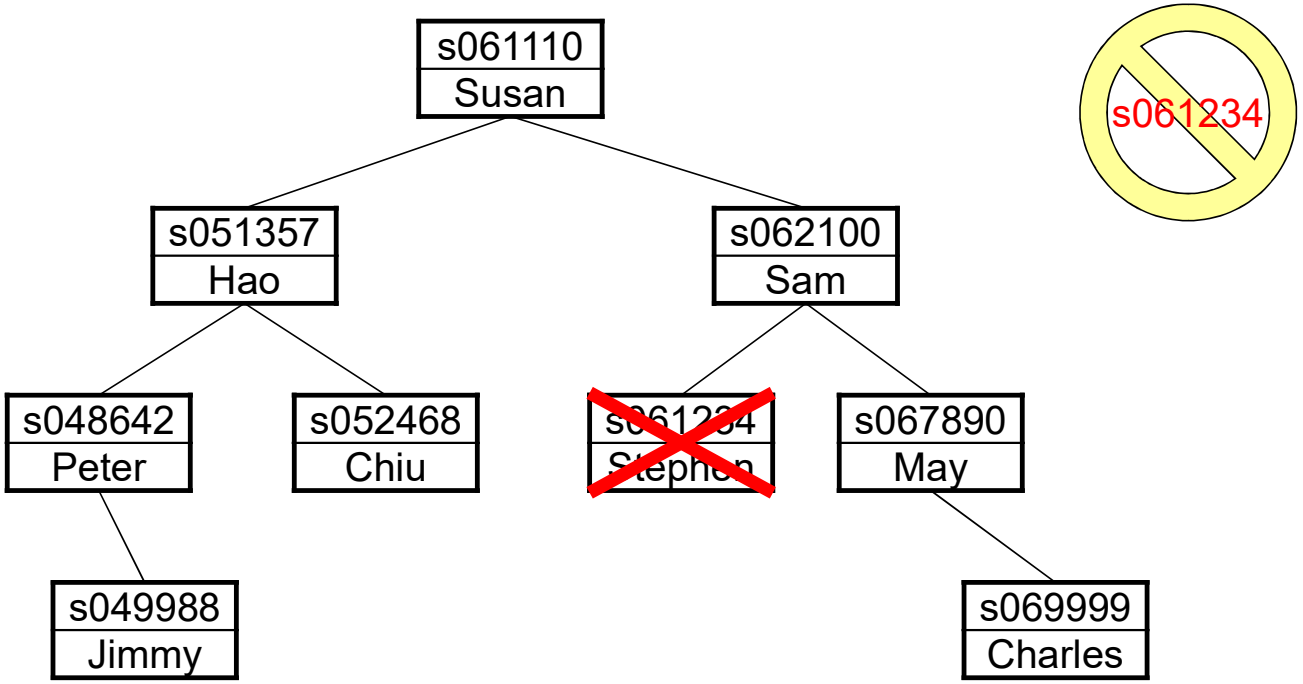
s067890 > s061110 → right subtree
s067890 > s062100 → right subtree

Inserting Nodes to a BST

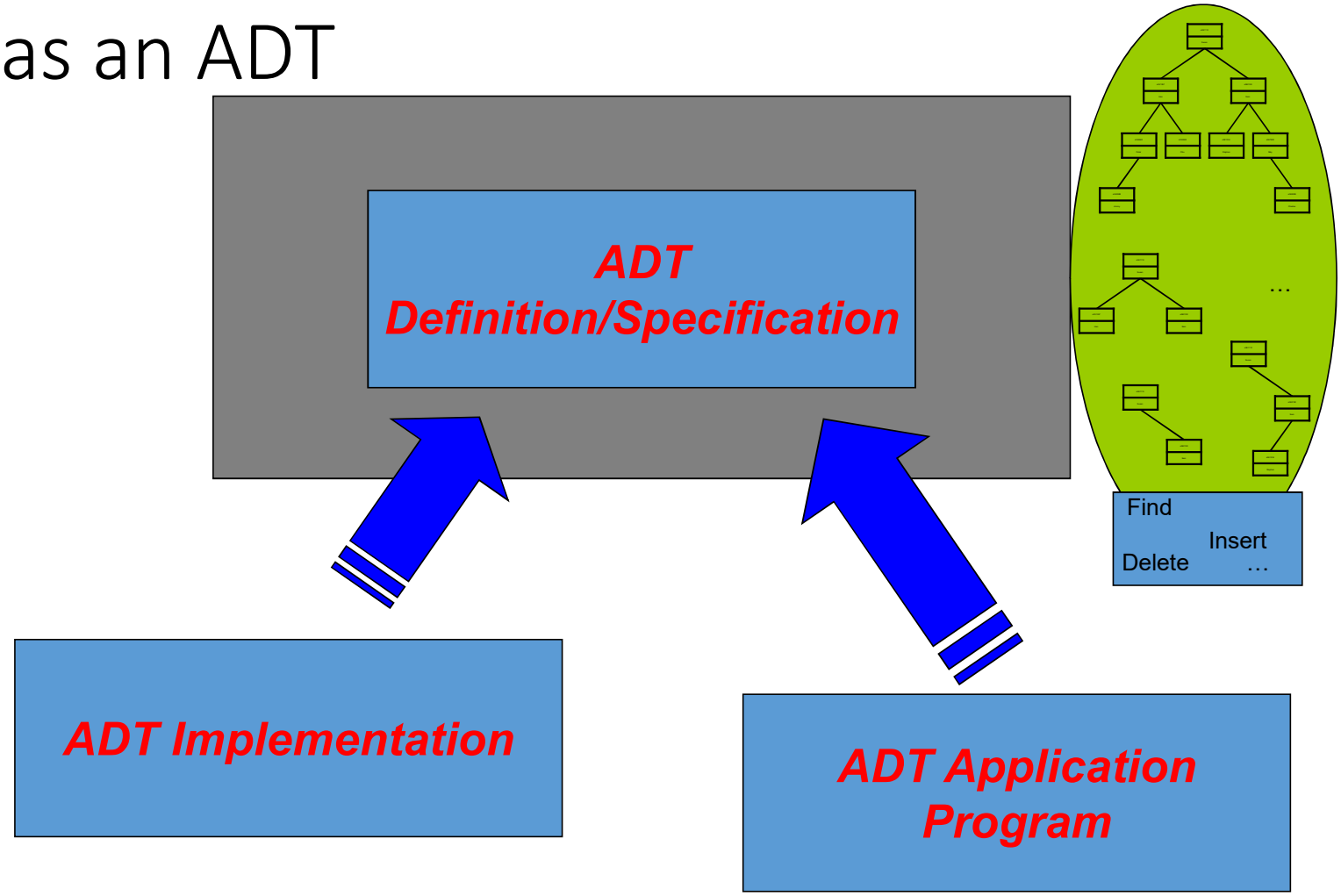


BST Operation: Deletion

- Deletes a node from a BST.

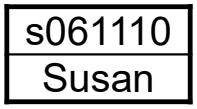


BST as an ADT

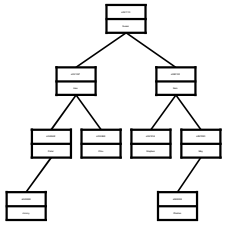

ADT
Definition/Specification
...
Find
Insert
Delete
…
ADT Implementation
ADT Application
Program

Defining Two ADTs

- We define two ADTs, one for tree nodes and one for BSTs.



treeNodeADT



bstADT

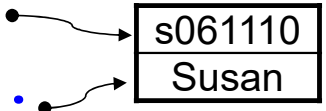
Defining a Tree Node ADT: treenode.h

treenode.h

```
typedef struct treeNodeCDT *treeNodeADT;

treeNodeADT NewNode(char *key, void *value);
char *GetNodeKey(treeNodeADT n);
void *GetNodeValue(treeNodeADT n);
void DelNode(treeNodeADT n);
treeNodeADT CopyNode(treeNodeADT n);
```

`treeNodeADT NewNode(char *key,`
`void *value);`



- Create and return a new tree node which corresponds to ⟨**`key`**, **`value`**⟩.

Defining a Tree Node ADT: treenode.h

```
char *GetNodeKey(treeNodeADT n);
```

- Return the key (which is a string) of the tree node `n`.

```
void *GetNodeValue(treeNodeADT n);
```

- Return the value of the tree node `n`.

```
void DelNode(treeNodeADT n);
```

- Delete the tree node `n`.

```
treeNodeADT CopyNode(treeNodeADT n);
```

- Copy the tree node `n` and return the new copy.

Defining a BST ADT: bst.h

bst.h

```
#include "treenode.h"

typedef struct bstCDT *bstADT;

bstADT EmptyBST();
int BSTIsEmpty(bstADT t);

bstADT MakeBST(treeNodeADT root, bstADT left, bstADT right);

treeNodeADT Root(bstADT t);
bstADT LeftSubtree(bstADT t);
bstADT RightSubtree(bstADT t);

treeNodeADT FindNode(bstADT t, char *key);
bstADT InsertNode(bstADT t, treeNodeADT n);
bstADT DeleteNode(bstADT t, char *key);
```

Defining a BST ADT: bst.h

```
#include "treenode.h"
```

- This is needed because the interface contains the type treeNodeADT.

```
bstADT EmptyBST();
```

- Return the empty BST.

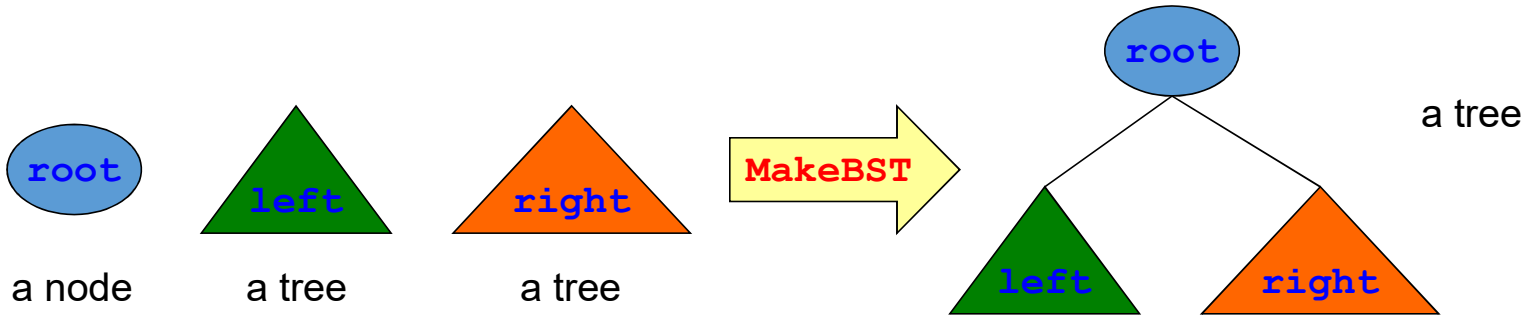
```
int BSTIsEmpty(bstADT t);
```

- Return 1 if `t` is the empty BST; 0 otherwise.

Defining a BST ADT: bst.h

```
bstADT MakeBST(treeNodeADT root,
               bstADT left,
               bstADT right);
```

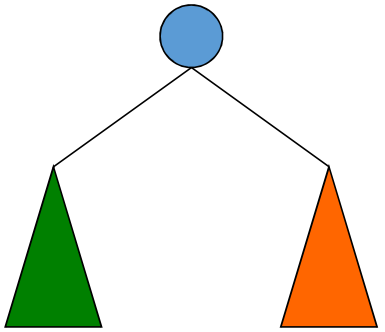
- Create and return a new BST rooted at the node **`root`** with left and right subtrees **`left`** and **`right`** respectively.



Defining a BST ADT: bst.h

```
treeNodeADT Root(bstADT t);
```

- Return the root of the BST `t`.



```
bstADT LeftSubtree(bstADT t);
```

- Return the left subtree of the BST `t`.

```
bstADT RightSubtree(bstADT t);
```

- Return the right subtree of the BST `t`.

Defining a BST ADT: bst.h

```
treeNodeADT FindNode(bstADT t, char *key);
```

- Return the node in the BST `t` whose key is the string `key`, or `NULL` if `key` does not exist in `t`.

```
bstADT InsertNode(bstADT t, treeNodeADT n);
```

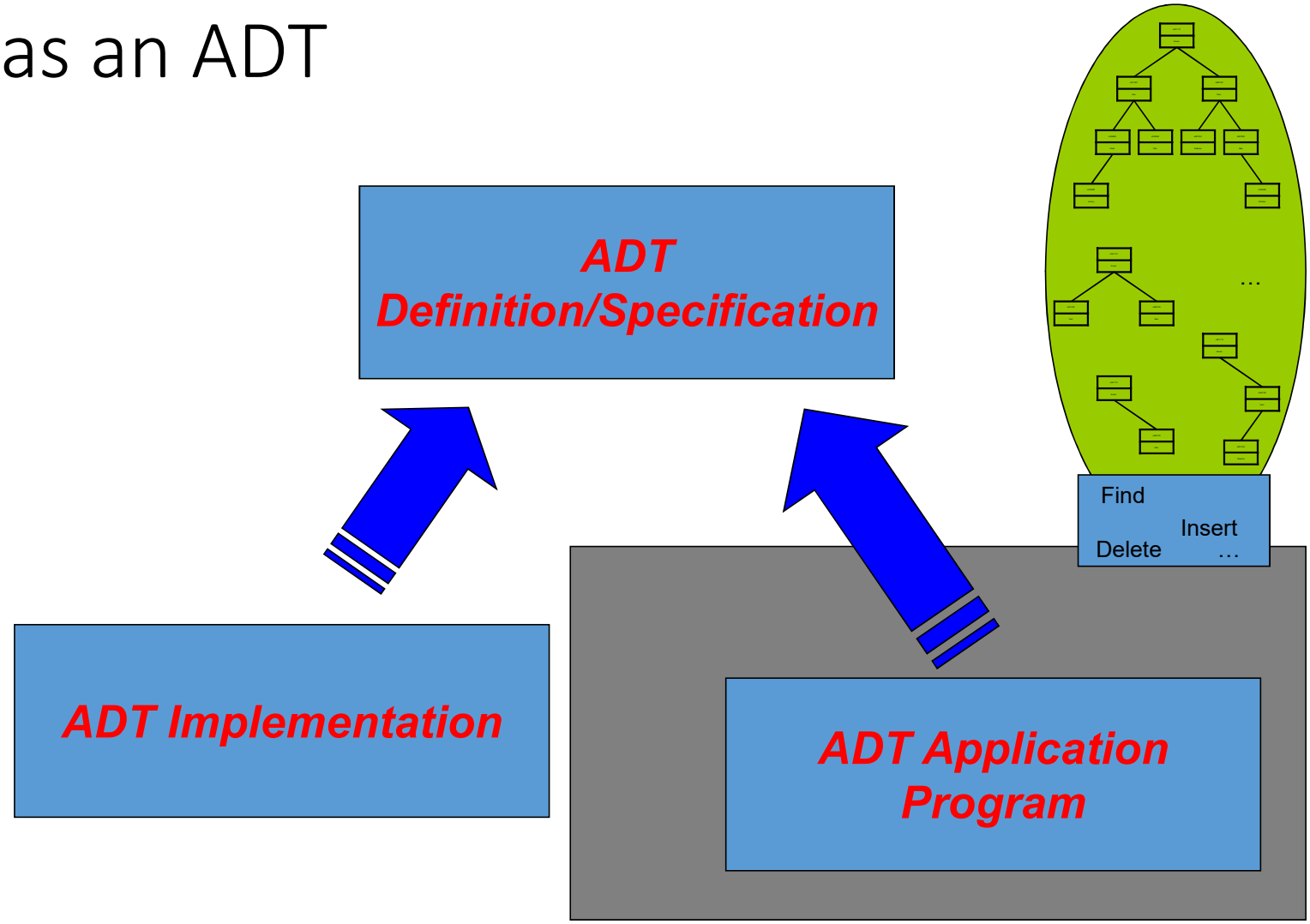
- Return a BST which has the node `n` inserted to the BST `t`.
- If `t` contains a node which has the same key as `n`, then the return tree should be `t` with the node replaced by `n`.

Defining a BST ADT: bst.h

```
bstADT DeleteNode(bstADT t,
                  char *key);
```

- Return a BST which has the node with key **`key`** deleted from the BST **`t`**.
- Return **`t`** if **`key`** does not exist in **`t`**.

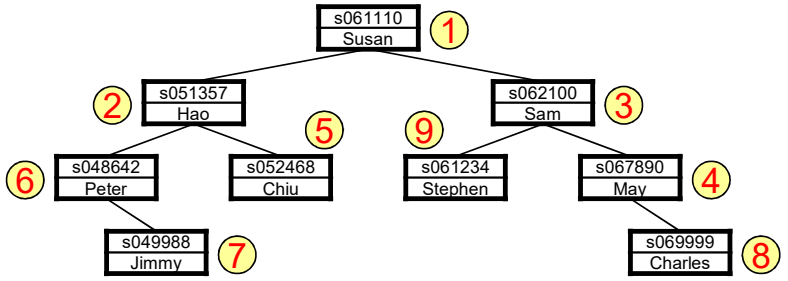
BST as an ADT


ADT
Definition/Specification
...
Find
Insert
Delete
…
ADT Implementation
ADT Application
Program

Using the BST ADT

- Building a BST from an empty one.

```
   bstADT bst;
   bst = EmptyBST();
1  bst = InsertNode(bst, NewNode("s061110", "Susan"));
2  bst = InsertNode(bst, NewNode("s051357", "Hao"));
3  bst = InsertNode(bst, NewNode("s062100", "Sam"));
4  bst = InsertNode(bst, NewNode("s067890", "May"));
5  bst = InsertNode(bst, NewNode("s052468", "Chiu"));
6  bst = InsertNode(bst, NewNode("s048642", "Peter"));
7  bst = InsertNode(bst, NewNode("s049988", "Jimmy"));
8  bst = InsertNode(bst, NewNode("s069999", "Charles"));
9  bst = InsertNode(bst, NewNode("s061234", "Stephen"));
```



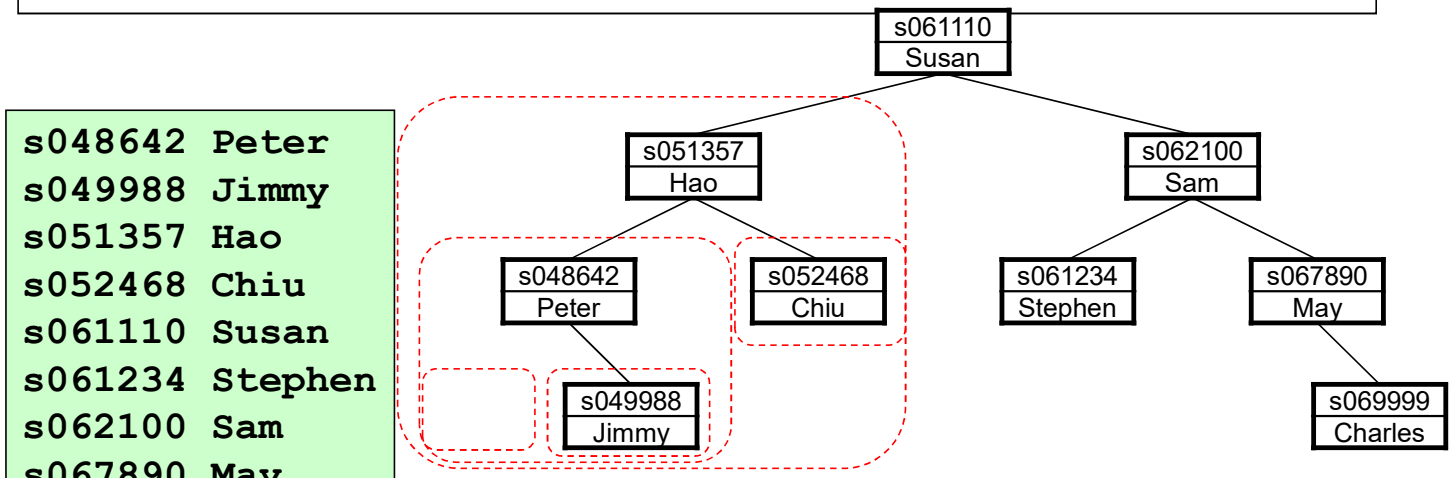
Height of a BST

```
int BSTHeight(bstADT t) {
   int lh, rh;
   if (BSTIsEmpty(t))
      return 0;
   else {
      lh = BSTHeight(LeftSubtree(t));
      rh = BSTHeight(RightSubtree(t));
      if (lh < rh)
         return rh + 1;
      else
         return lh + 1;
   }
}
```

The height of a non-empty BST is one more than the height of its left or right subtrees, whichever is larger.

Displaying a BST

```
void DisplayBST(bstADT t) {
   if (!BSTIsEmpty(t)) {
      DisplayBST(LeftSubtree(t));
      printf("%s\t%s\n", GetNodeKey(Root(t)),
                         (char *)GetNodeValue(Root(t)));
      DisplayBST(RightSubtree(t));
   }
}
```




```
s048642 Peter
s049988 Jimmy
s051357 Hao
s052468 Chiu
s061110 Susan
s061234 Stephen
s062100 Sam
s067890 May
s069999 Charles
```

Note: the output is sorted according to the keys.

Traversing a Tree

- The process of going through the nodes of a tree and performing some operation at each node is called ***traversing*** or ***walking*** the tree.
- The **`DisplayBST`** function traverses a BST.
- It is an ***inorder*** traversal, in which we process a node (displaying it in this example) *between* the recursive calls.

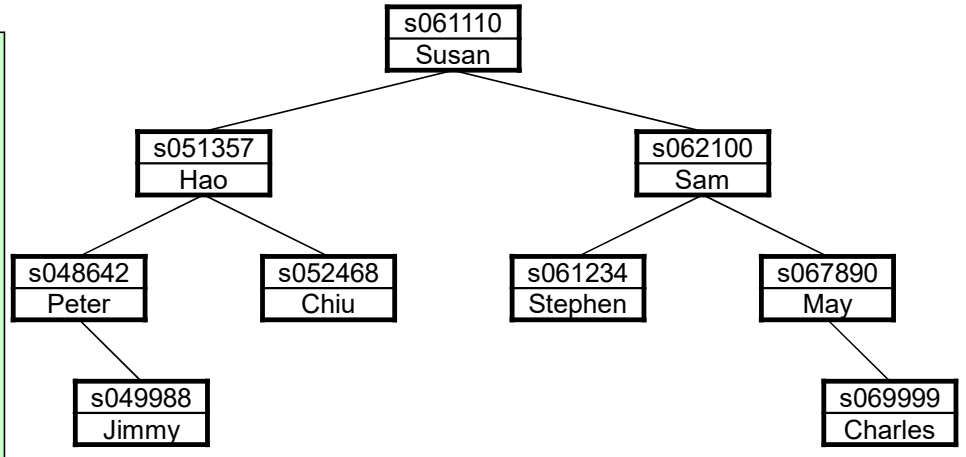
```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   printf(…);    // between recursive calls
   DisplayBST(RightSubtree(t));
}
```

Traversing a Tree: Preorder

- ***Preorder*** traversal processes a node *before* the recursive calls.

```
if (!BSTIsEmpty(t)) {
   printf(…);   // before recursive calls
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
}
```

```
s061110 Susan
s051357 Hao
s048642 Peter
s049988 Jimmy
s052468 Chiu
s062100 Sam
s061234 Stephen
s067890 May
s069999 Charles
```

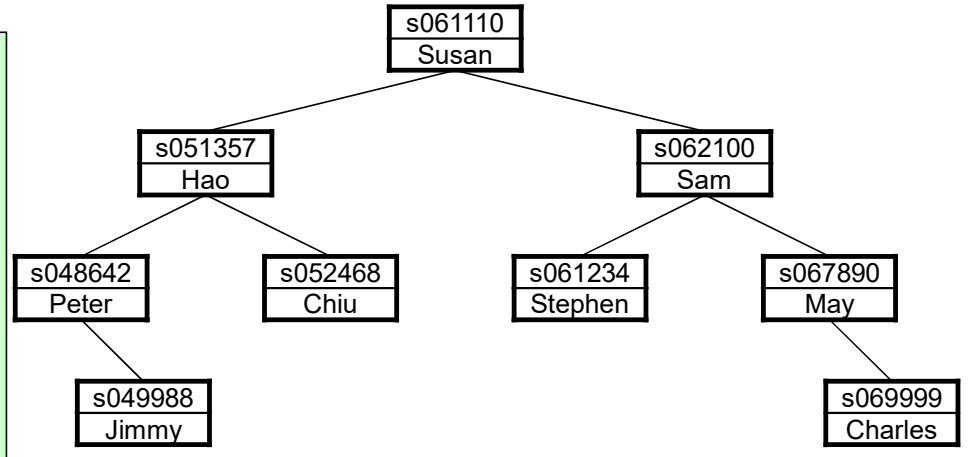


Traversing a Tree: Postorder

- ***Postorder*** traversal processes a node *after* the recursive calls.

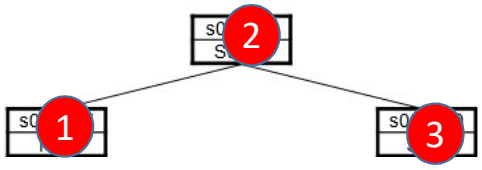
```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
   printf(…);    // after recursive calls
}
```

```
s049988 Jimmy
s048642 Peter
s052468 Chiu
s051357 Hao
s061234 Stephen
s069999 Charles
s067890 May
s062100 Sam
s061110 Susan
```



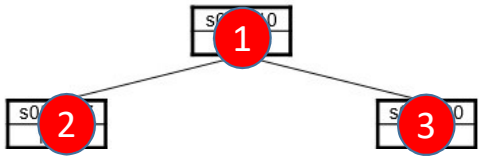
Inorder traversal

```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   printf(…);    // between recursive calls
   DisplayBST(RightSubtree(t));
}
```



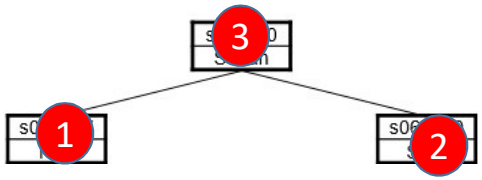
Preorder traversal

```
if (!BSTIsEmpty(t)) {
   printf(…);    // before recursive calls
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
}
```



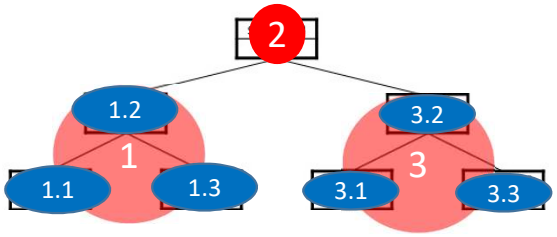
Postorder traversal

```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
   printf(…);    // after recursive calls
}
```



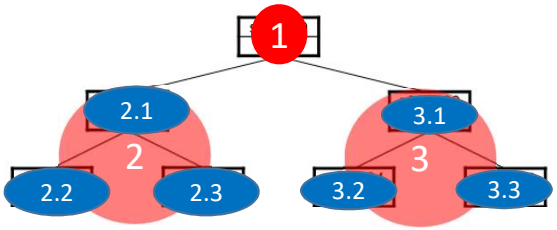
Inorder traversal

```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   printf(…);    // between recursive calls
   DisplayBST(RightSubtree(t));
}
```



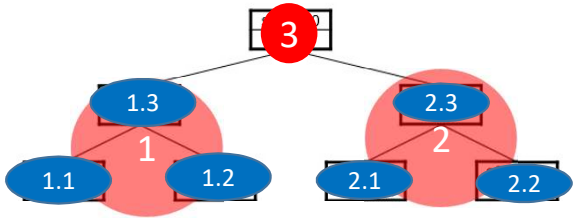
Preorder traversal

```
if (!BSTIsEmpty(t)) {
   printf(…);    // before recursive calls
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
}
```

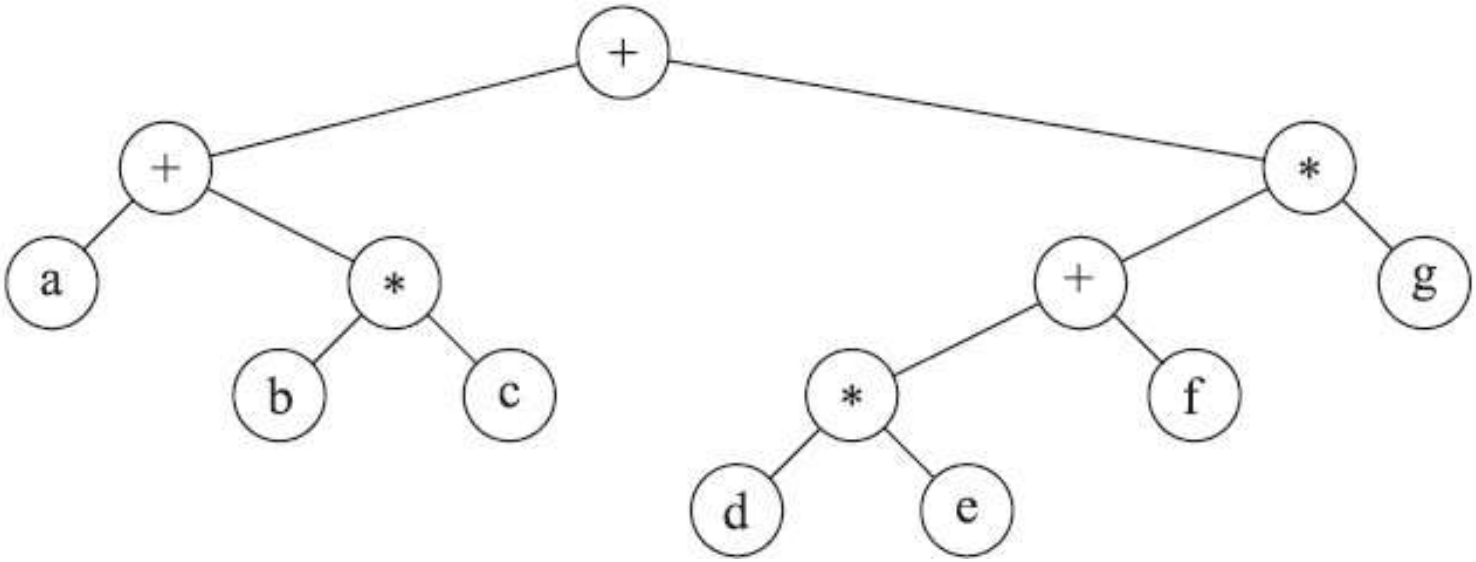


Postorder traversal

```
if (!BSTIsEmpty(t)) {
   DisplayBST(LeftSubtree(t));
   DisplayBST(RightSubtree(t));
   printf(…);    // after recursive calls
}
```



Expression tree

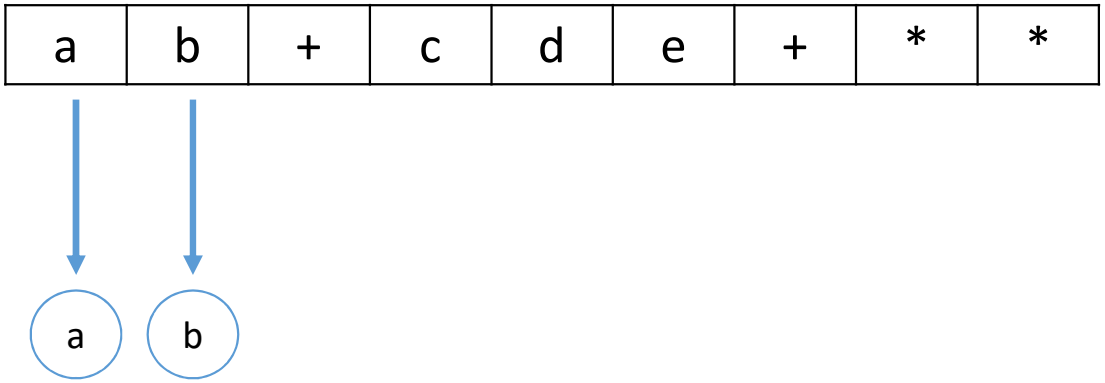


Inorder traversal : (a+b*c) + ((d*e + f) * g)

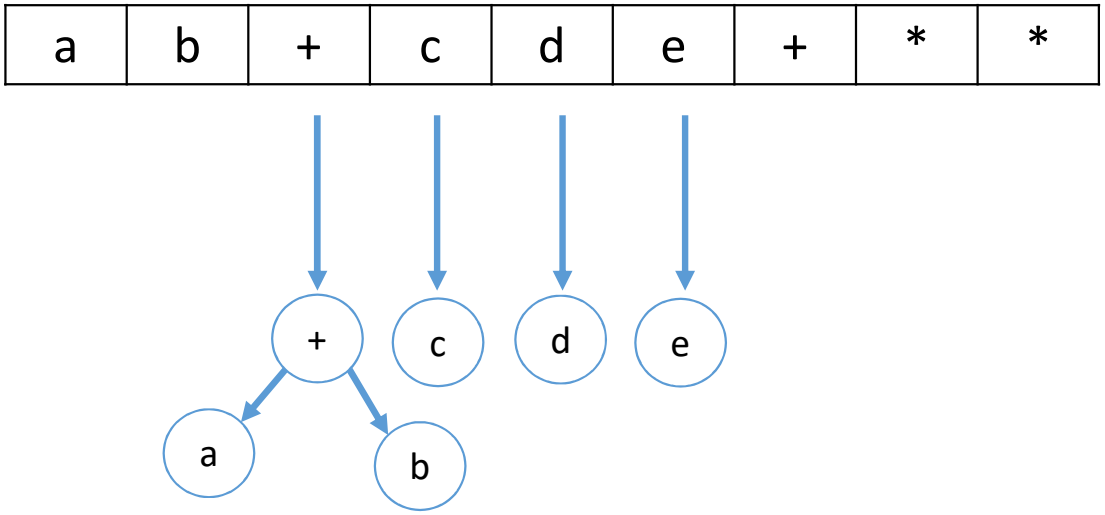
postorder traversal : a b c * + d e * f + g * +

preorder traversal : + + a * b c * + * d e f g

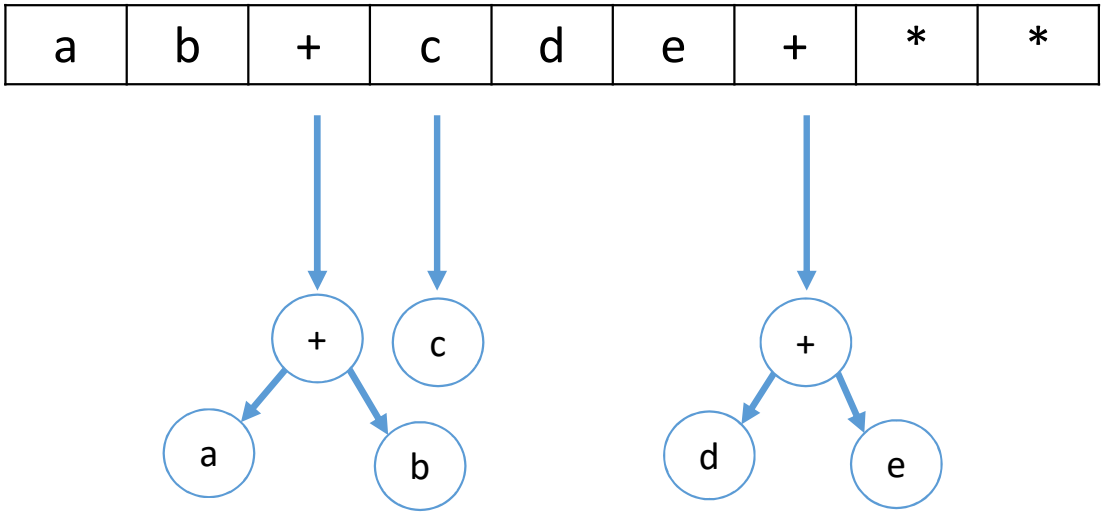
From Stack to Expression Tree



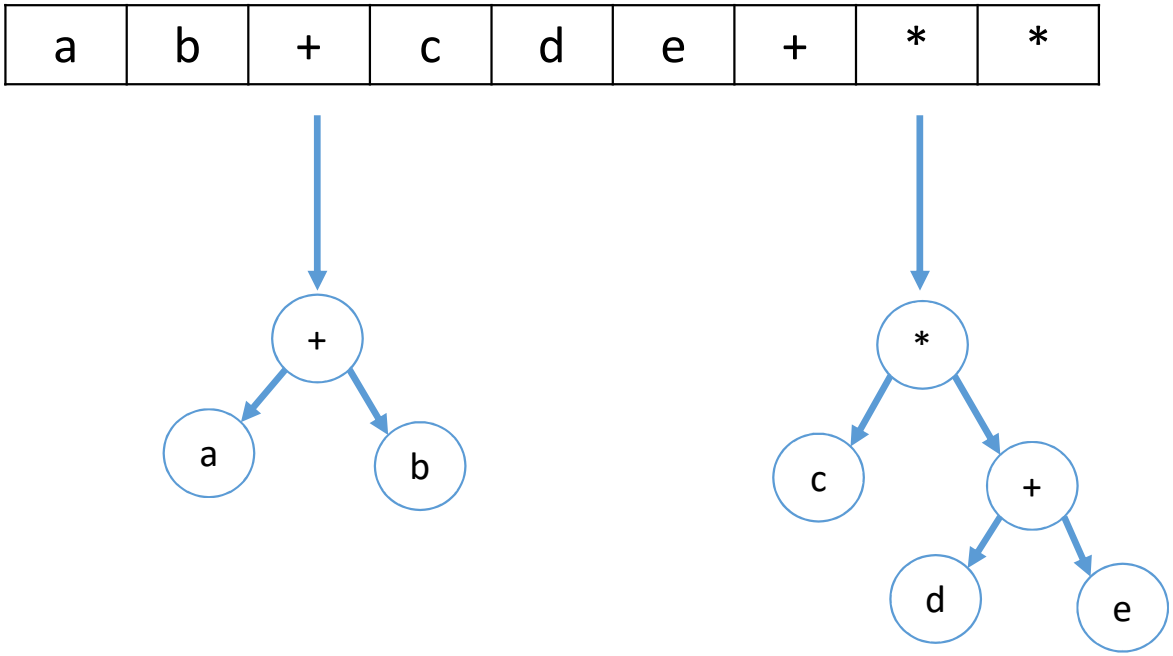
From Stack to Expression Tree



From Stack to Expression Tree



From Stack to Expression Tree



From Stack to Expression Tree