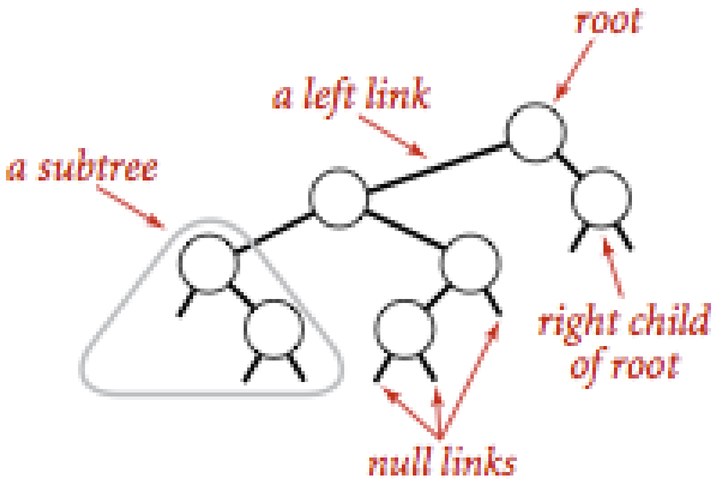
CS302 - Data Structures
using C++

Topic: Binary Search Tree Implementation

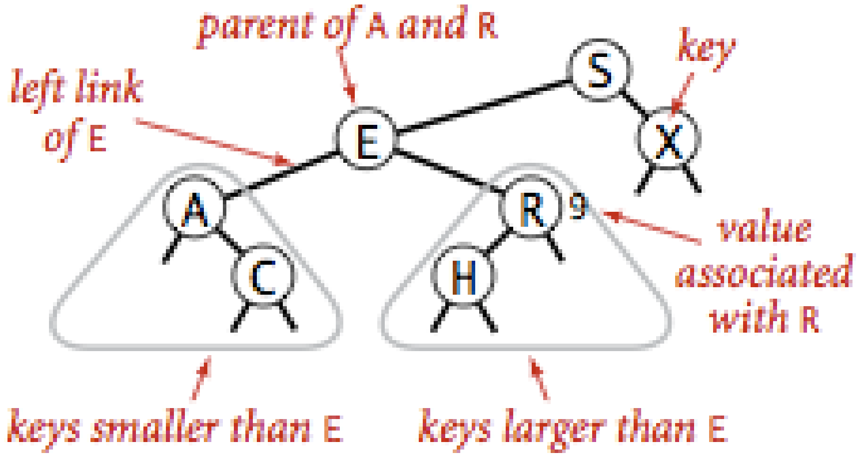
Kostas Alexis

Binary Search Trees

- Definition: A Binary Search Tree (BST) is a binary tree where each node has a Comparable key (and an associated value) and satisfies the restriction that the key in any node is larger than the keys in all nodes in that node's left subtree and smaller than the keys in all nodes in that node's right subtree.



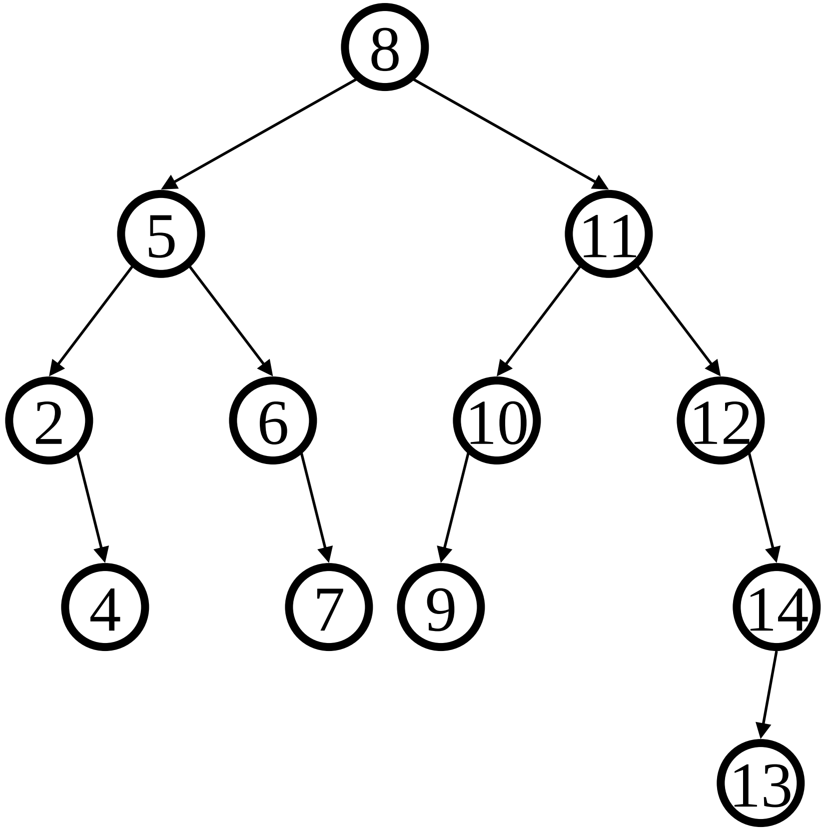
Anatomy of a binary tree



Anatomy of a binary search tree

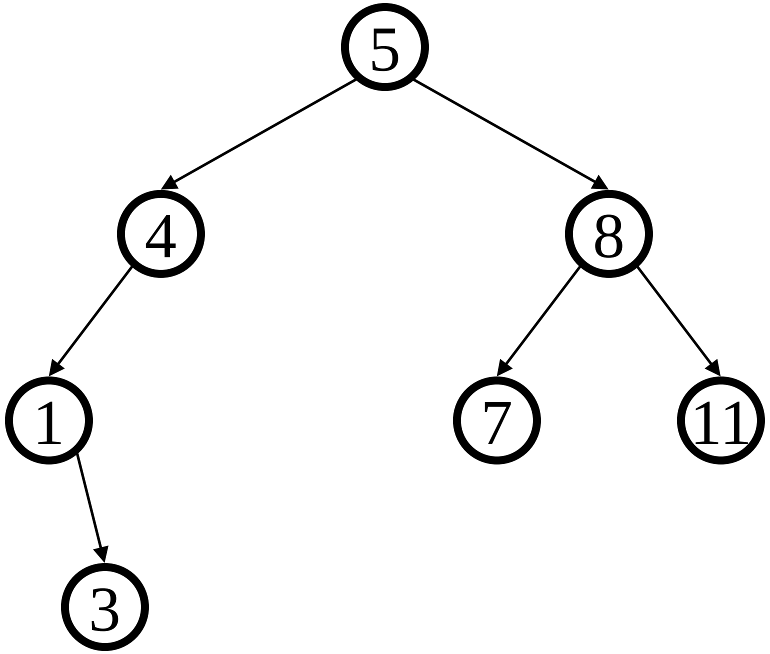
Binary Search Trees

- Binary Tree property
    - Each node has <= 2 children
    - Result
        - Storage is small
        - Operations are simple
        - Average depth is small
- Search Tree property
    - All keys in left subtree smaller than root’s key
    - All keys in right subtree larger than root’s key
    - Result
        - Easy to find any given key
        - Insert/delete by changing links

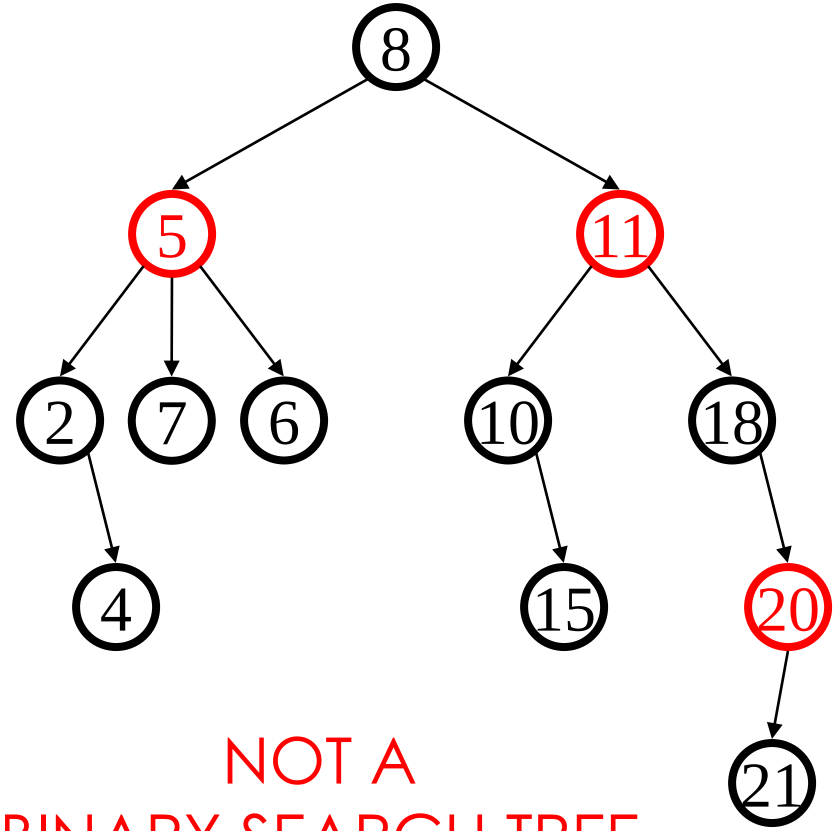


Binary Search Trees

- Example and Counter-Example



BINARY SEARCH TREE



NOT A
BINARY SEARCH TREE

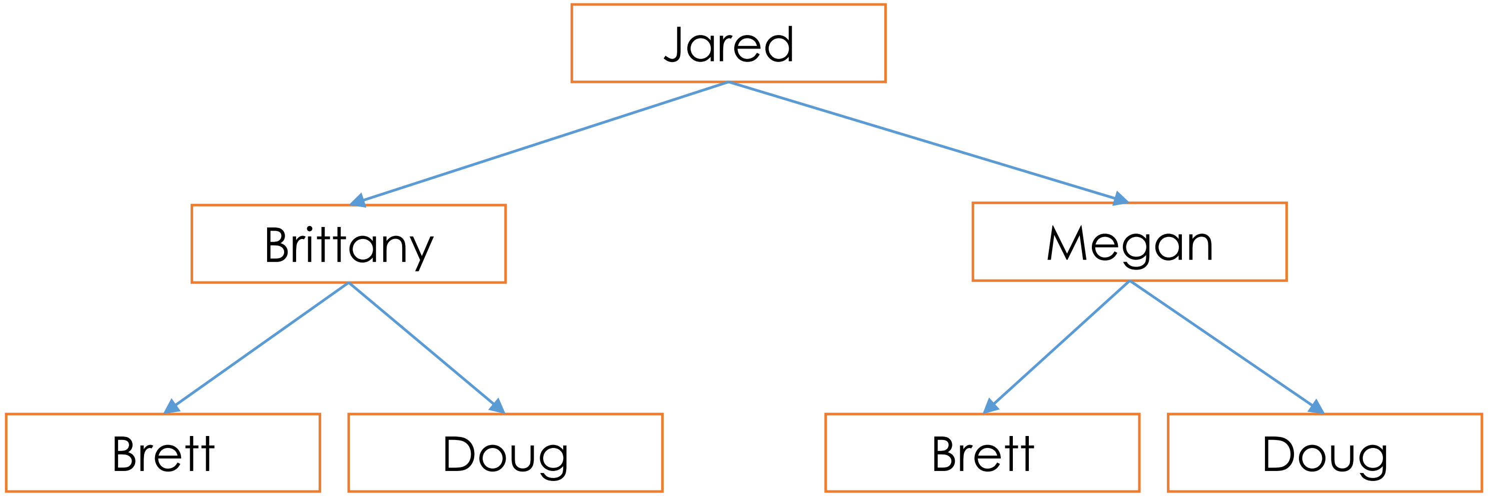
Binary Search Trees

- Binary Search Tree (BST)
  - Binary tree that has the following properties for each node **n**:
    - **n**’s value is > all values in **n**’s left subtree $\mathbf{T_L}$
    - **n**’s value is < all values in **n**’s right subtree $\mathbf{T_R}$
    - Both $\mathbf{T_L}$ and $\mathbf{T_R}$ are binary search trees

Binary Search Trees

- Binary Search Tree (BST)
  - Binary tree that has the following properties for each node **n**:
    - **n**’s value is > all values in **n**’s left subtree $\mathbf{T_L}$
    - **n**’s value is < all values in **n**’s right subtree $\mathbf{T_R}$
    - Both $\mathbf{T_L}$ and $\mathbf{T_R}$ are binary search trees
- A binary tree whose nodes contain objects and
  - Data in a node is greater than the data in the node’s left child
  - Data in a node is less than the data in the node’s right child

Binary Search Trees



Jared

Brittany

Megan

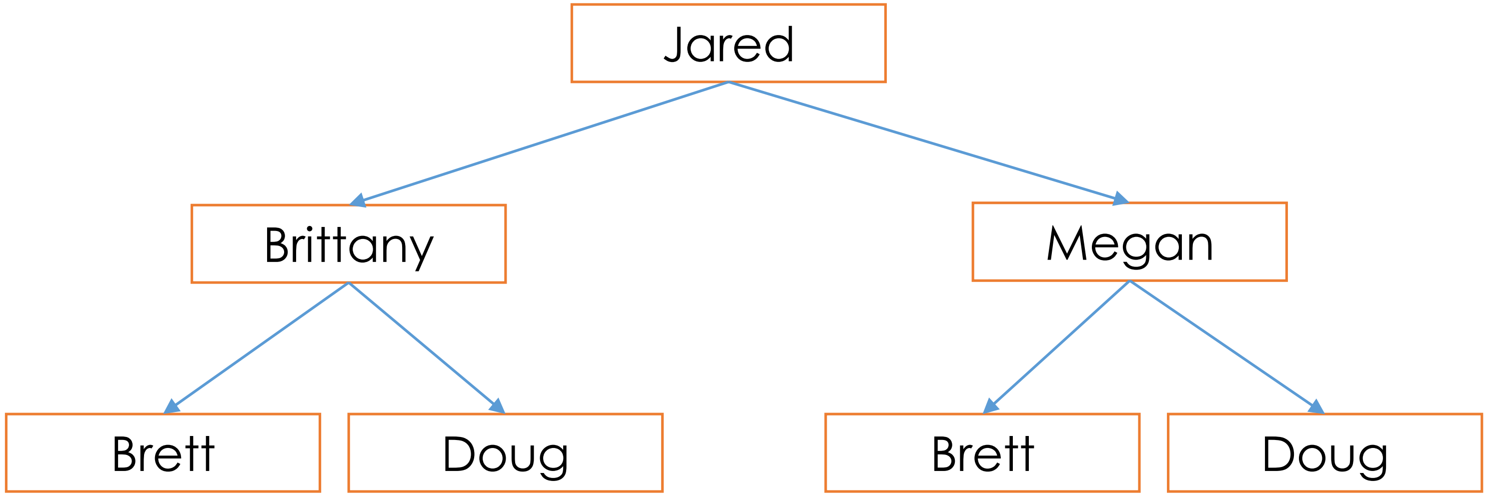
Brett

Doug

Brett

Doug

Binary Search Trees



An In-Order Traversal is "In Order" for a Binary Search Tree

Link-based Implementation of the ADT Binary Search Tree

- Uses same node objects as for binary-tree implementation

Link-based Implementation of the ADT Binary Search Tree

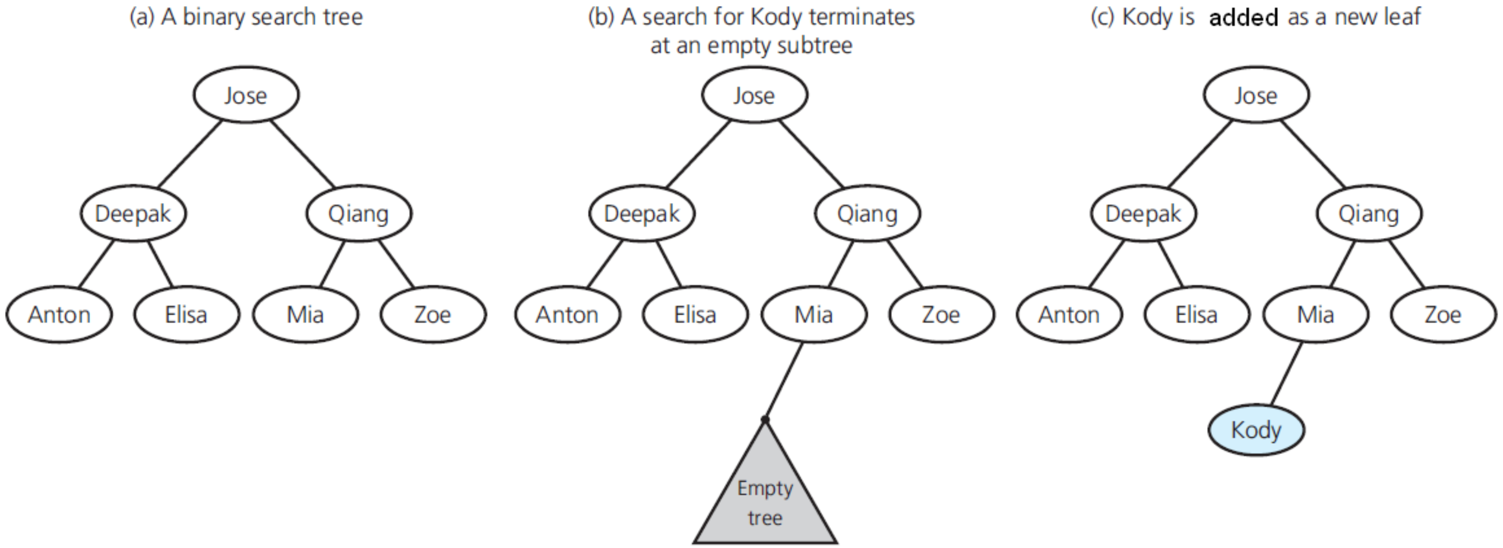
- Uses same node objects as for binary-tree implementation
- Class BinaryNode will be used

Link-based Implementation of the ADT Binary Search Tree

- Uses same node objects as for binary-tree implementation
- Class BinaryNode will be used
- Recursive search algorithm is the basis for operations

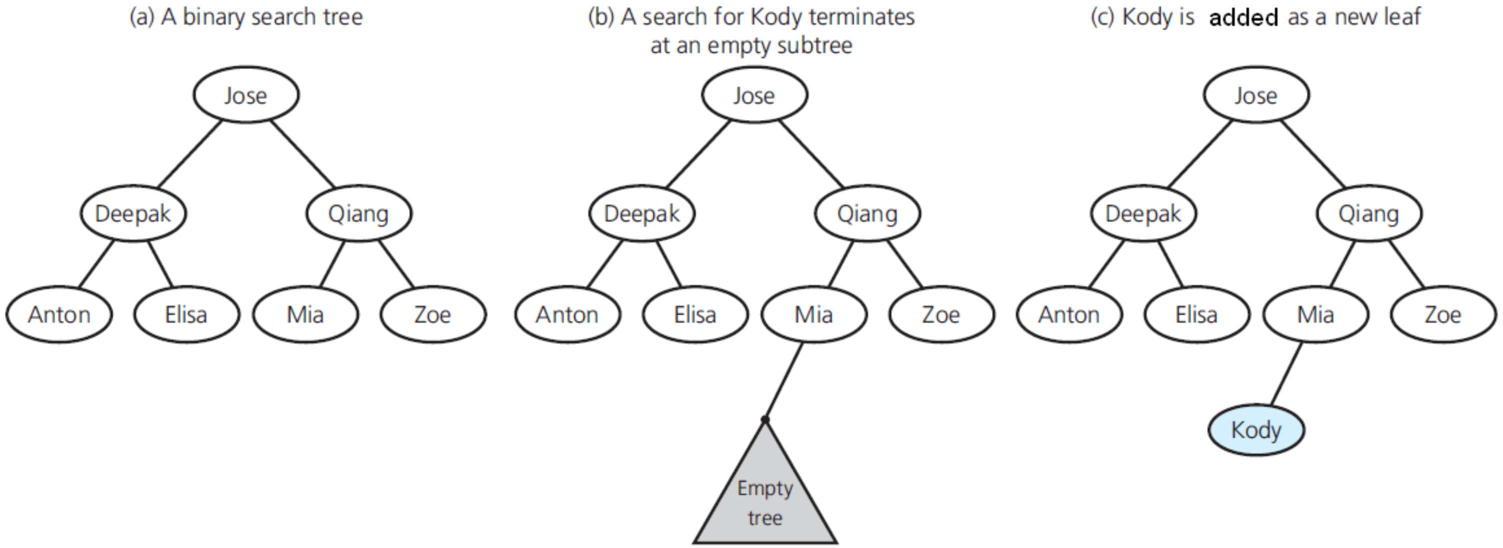
Link-based Implementation of the ADT Binary Search Tree

- Adding “Kody” to a Binary Search Tree



Link-based Implementation of the ADT Binary Search Tree

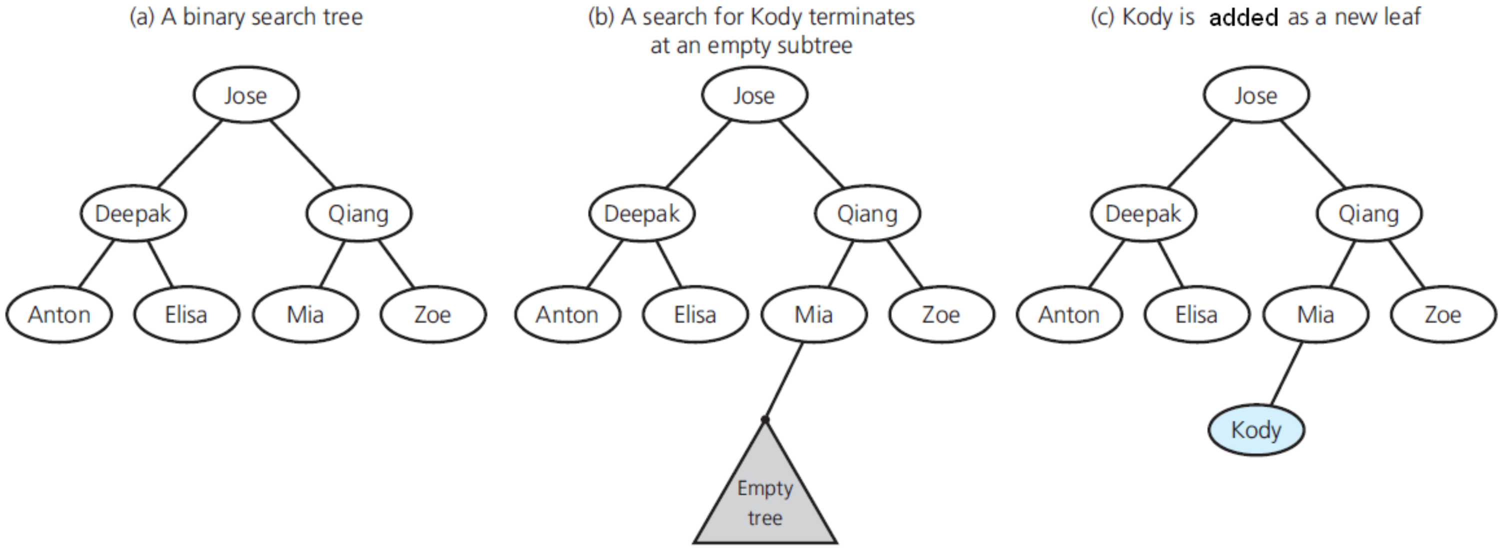
- Adding “Kody” to a Binary Search Tree



- Addition is “like search” we first find where Kody should be if it was there.

Link-based Implementation of the ADT Binary Search Tree

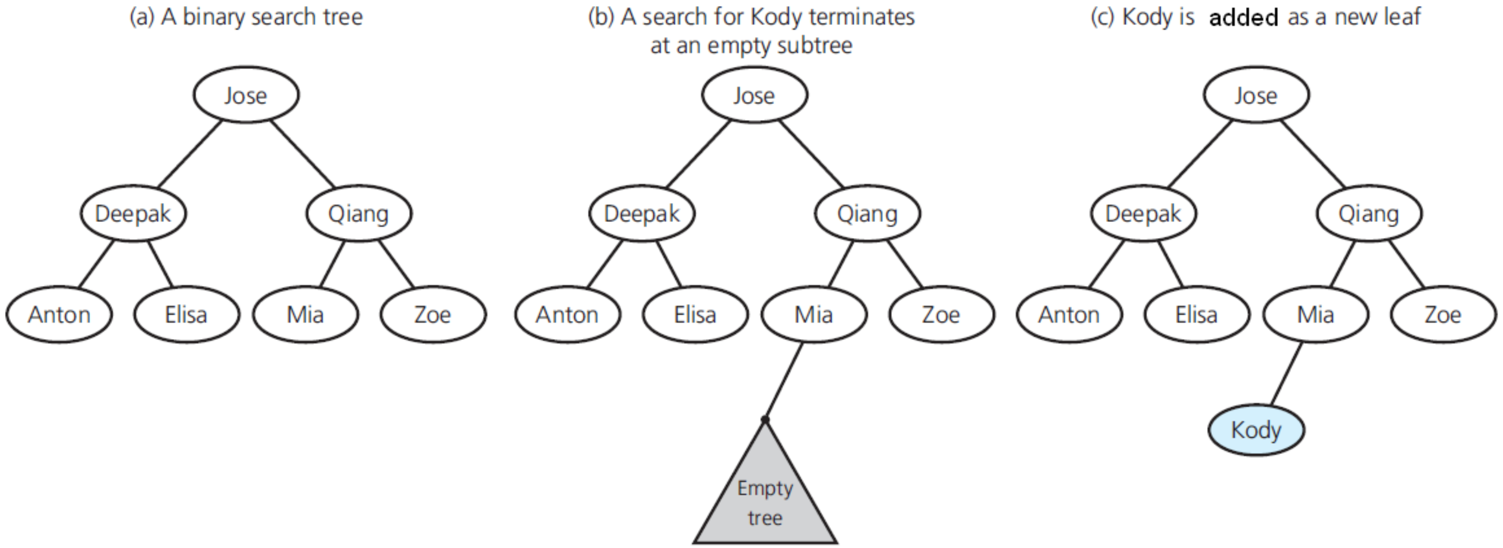
- Adding “Kody” to a Binary Search Tree



- Addition is “like search” we first find where Kody should be if it was there.
  - Terminates at Mia

Link-based Implementation of the ADT Binary Search Tree

- Adding “Kody” to a Binary Search Tree



- Addition is “like search” we first find where Kody should be if it was there.
  - Terminates at Mia
- As Mia has no left child, the addition is simple, requiring only that Mia’s left child pointer points to a new node that contains Kody.

Notes for the class BinarySearchTree

- **In this lecture-approach:** based on BinaryNodeTree
- Tree Root Data Field **rootPtr**
- “Disable” any methods that could change a value
  - BinaryNode Tree
    - **void** setRootData(**const** ItemType& newData)

Link-based Implementation of the ADT Binary Search Tree

- Method **add**

```
template<class ItemType>
bool BinarySearchTree<ItemType>::add(const ItemType& newData)
{
    auto newNodePtr = std::make_shared<BinaryNode<ItemType>>(newData);
    rootPtr = placeNode(rootPtr, newNodePtr);
    return true;
} // end add
```

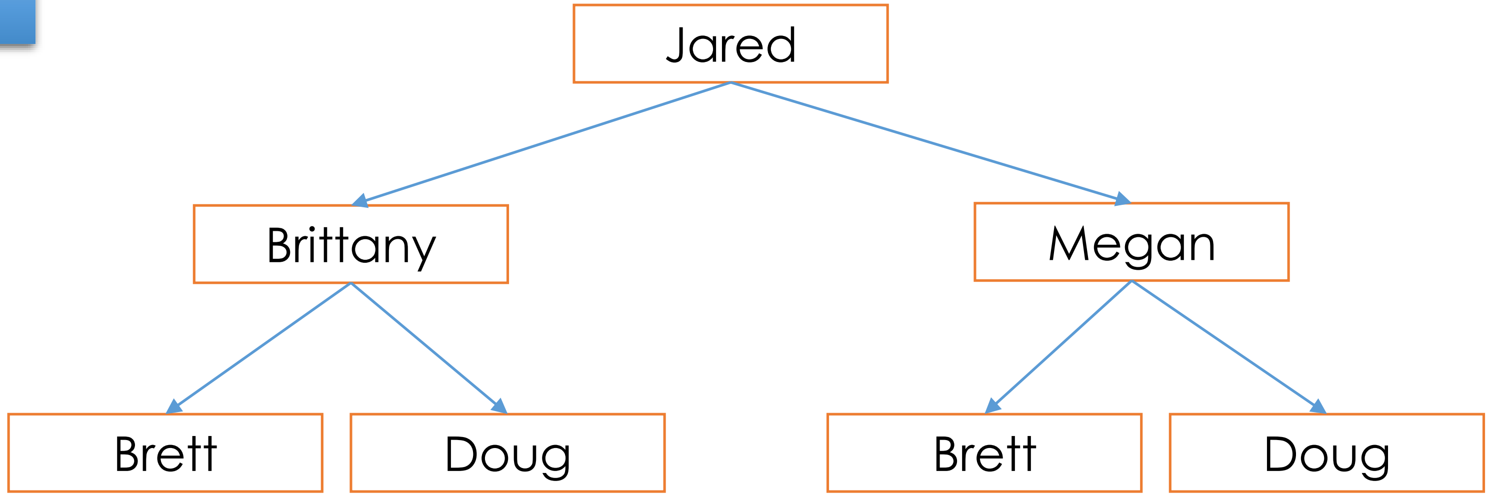
Link-based Implementation of the ADT Binary Search Tree

- Method **add**
    - Must maintain binary search tree structure
    - Every addition to a binary search tree adds a new leaf to the tree

Link-based Implementation of the ADT Binary Search Tree

- Method **add**
  - Must maintain binary search tree structure
  - Every addition to a binary search tree adds a new leaf to the tree

add (Chad)



Jared

Brittany

Megan

Brett

Doug

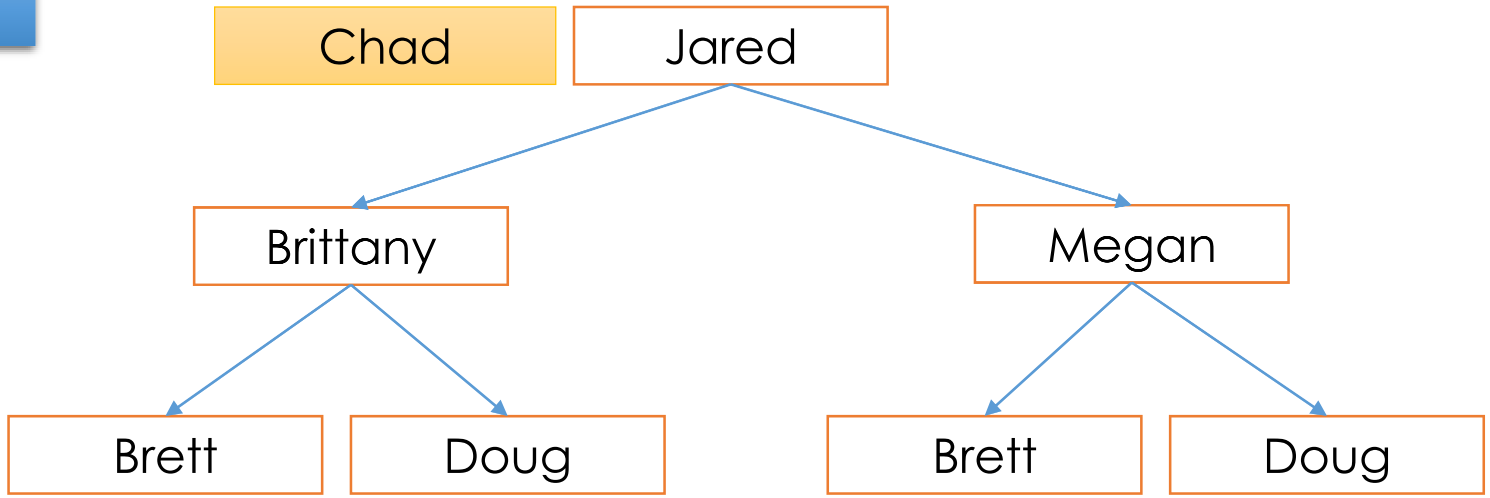
Brett

Doug

Link-based Implementation of the ADT Binary Search Tree

- Method **add**
  - Must maintain binary search tree structure
  - Every addition to a binary search tree adds a new leaf to the tree

add (Chad)



Chad

Jared

Brittany

Megan

Brett

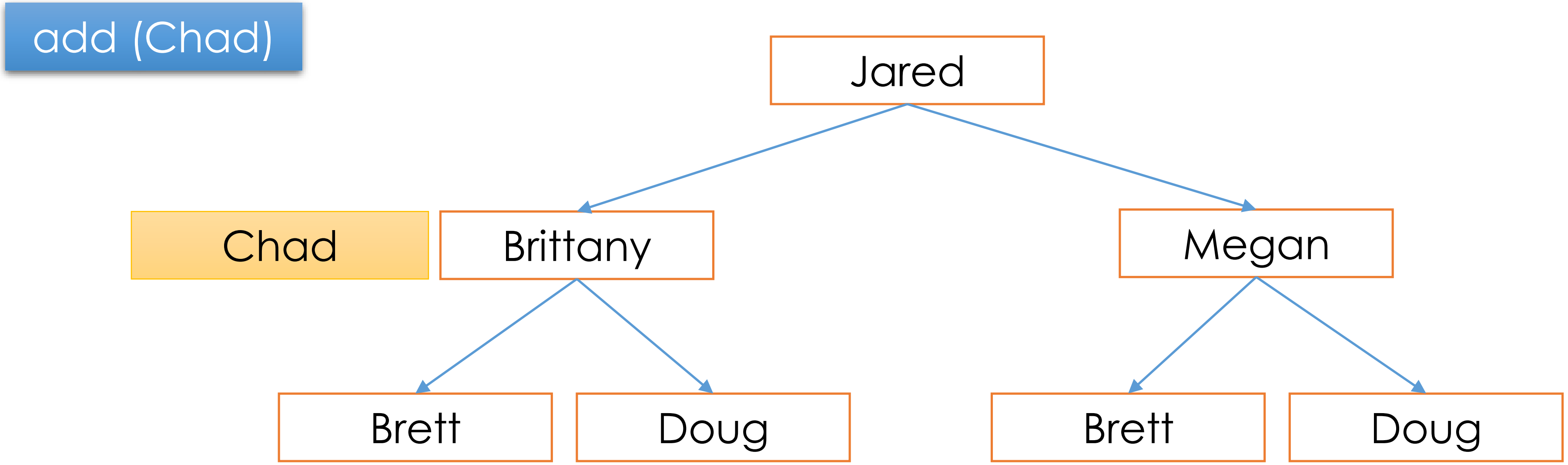
Doug

Brett

Doug

Link-based Implementation of the ADT Binary Search Tree

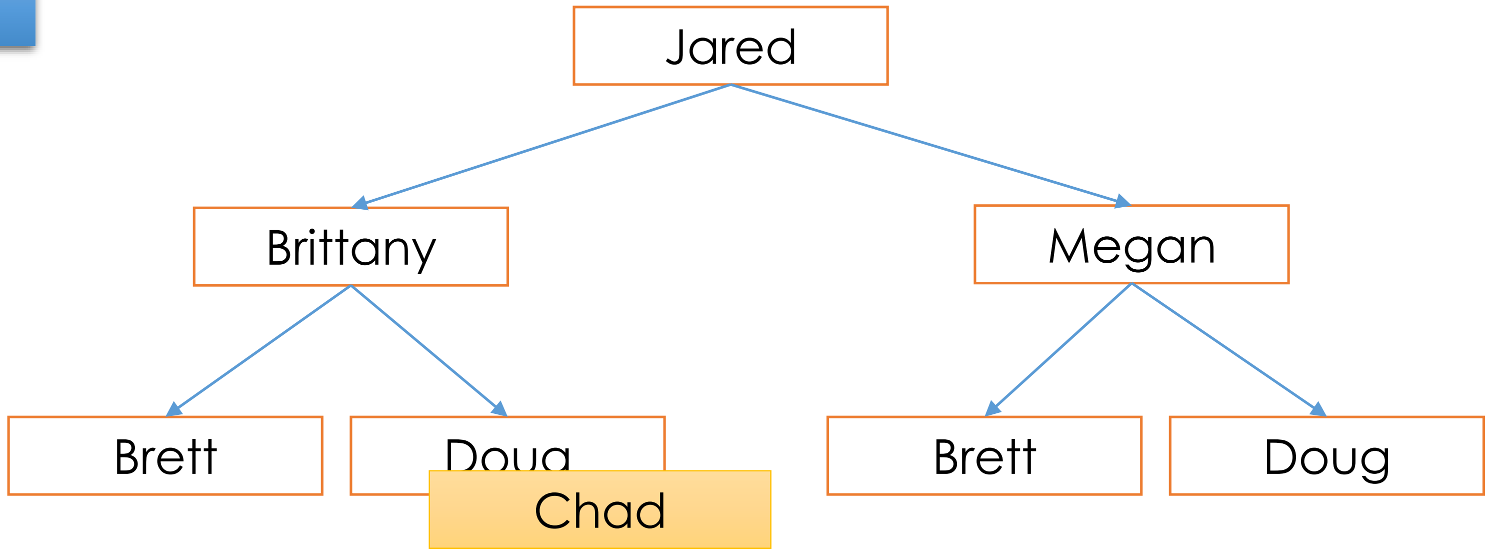
- Method **add**
  - Must maintain binary search tree structure
  - Every addition to a binary search tree adds a new leaf to the tree



Link-based Implementation of the ADT Binary Search Tree

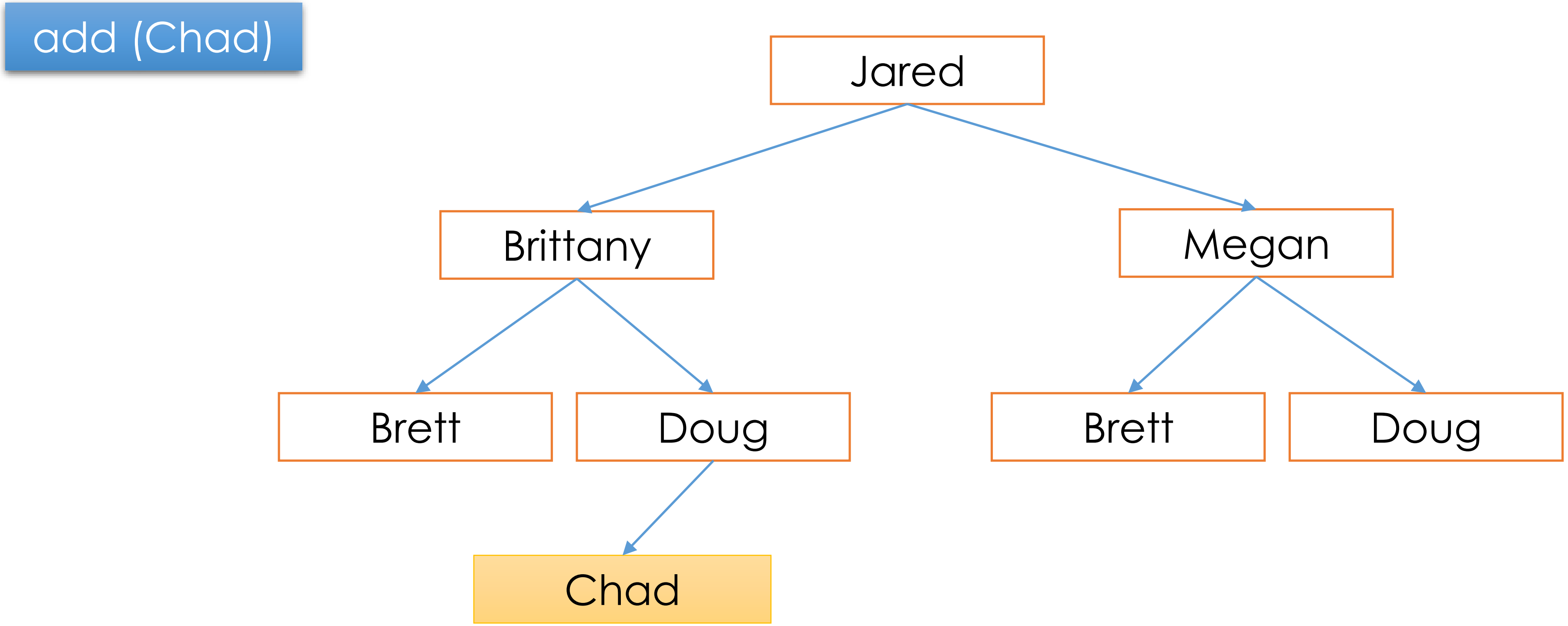
- Method **add**
  - Must maintain binary search tree structure
  - Every addition to a binary search tree adds a new leaf to the tree

add (Chad)



Link-based Implementation of the ADT Binary Search Tree

- Method **add**
  - Must maintain binary search tree structure
  - Every addition to a binary search tree adds a new leaf to the tree



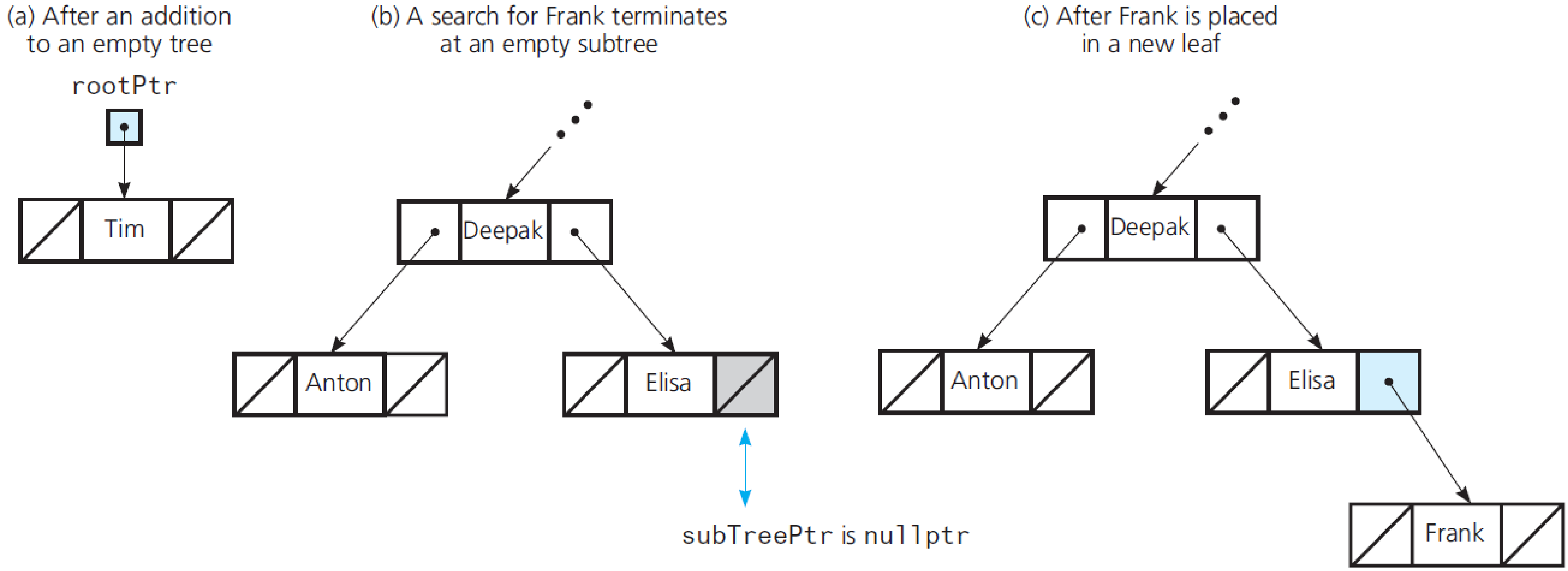
Link-based Implementation of the ADT Binary Search Tree

- Refinement of **addition** algorithm

```
// Recursively places a given new node at its proper position in a binary search tree
placeNode(subTreePtr: BinaryNodePointer, newNodePtr: BinaryNodePointer): BindayNodePointer
{
    if (subTreePtr == nullptr)
        return newNodePtr
    else if (subTreePtr->getItem() > newNodePtr->getItem())
    {
        tempPtr = placeNode(subTreePtr->getLeftChildPtr(), newNodePtr)
        subTreePtr->setLeftChildPtr(tempPtr)
    }
    else
    {
        tempPtr = placeNode(subTreePtr->getRightChildPtr(), newNodePtr)
        subTreePtr->setRightChildPtr(tempPtr)
    }
    return subTreePtr
}
```

Link-based Implementation of the ADT Binary Search Tree

- Adding new data to a binary search tree



Link-based Implementation of the ADT Binary Search Tree

- Removing an entry.
  - More involved than the adding process
  - First use the search algorithm to locate the specified item and then, if it is found, you must remove it from the tree
  - While doing so, maintain a Binary Search Tree structure.

Link-based Implementation of the ADT Binary Search Tree

- First draft of the **removal** algorithm

```
// Removes the given target from a binary search tree
// Returns true if the removal is successful or false otherwise
removeValue(target: ItemType): boolean
{
    Locate the target by using the search algorithm
    if (target is found)
    {
        Remove target from the tree
        return true
    }
    else
        return false
}
```

Link-based Implementation of the ADT Binary Search Tree

- Must maintain binary search tree structure

Link-based Implementation of the ADT Binary Search Tree

- Cases for node N containing item to be removed

1. **Case 1: N is a leaf**
2. **Case 2: N has only one child**
3. **Case 3: N has two children**

Link-based Implementation of the ADT Binary Search Tree

- Cases for node N containing item to be removed
- **Case 1: N is a leaf**
  - Remove leaf containing target
  - Set pointer in parent nullptr

Link-based Implementation of the ADT Binary Search Tree

- Cases for node N containing item to be removed
- **Case 2: N has only left (or right) child – cases are symmetrical**
    - After N removed, all data items rooted at L (or R) are adopted by root of N
    - All items adopted are in correct order, binary search tree property preserved

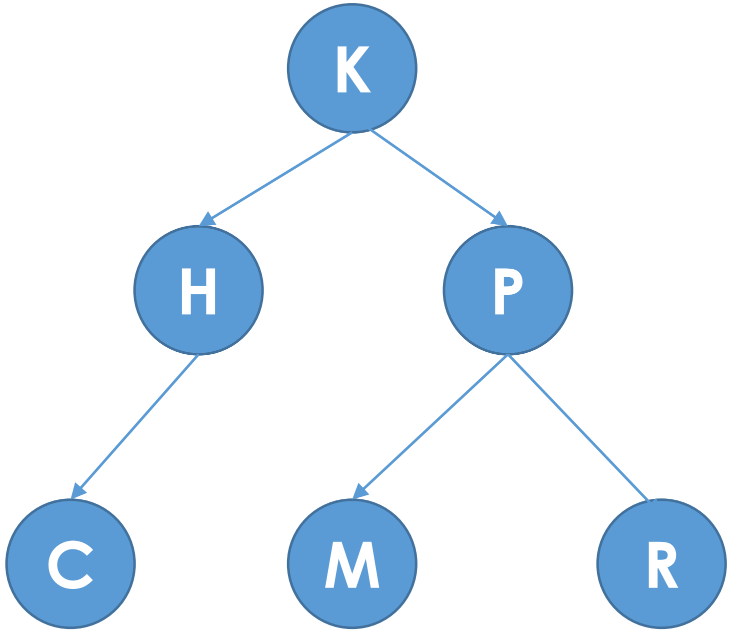
Link-based Implementation of the ADT Binary Search Tree

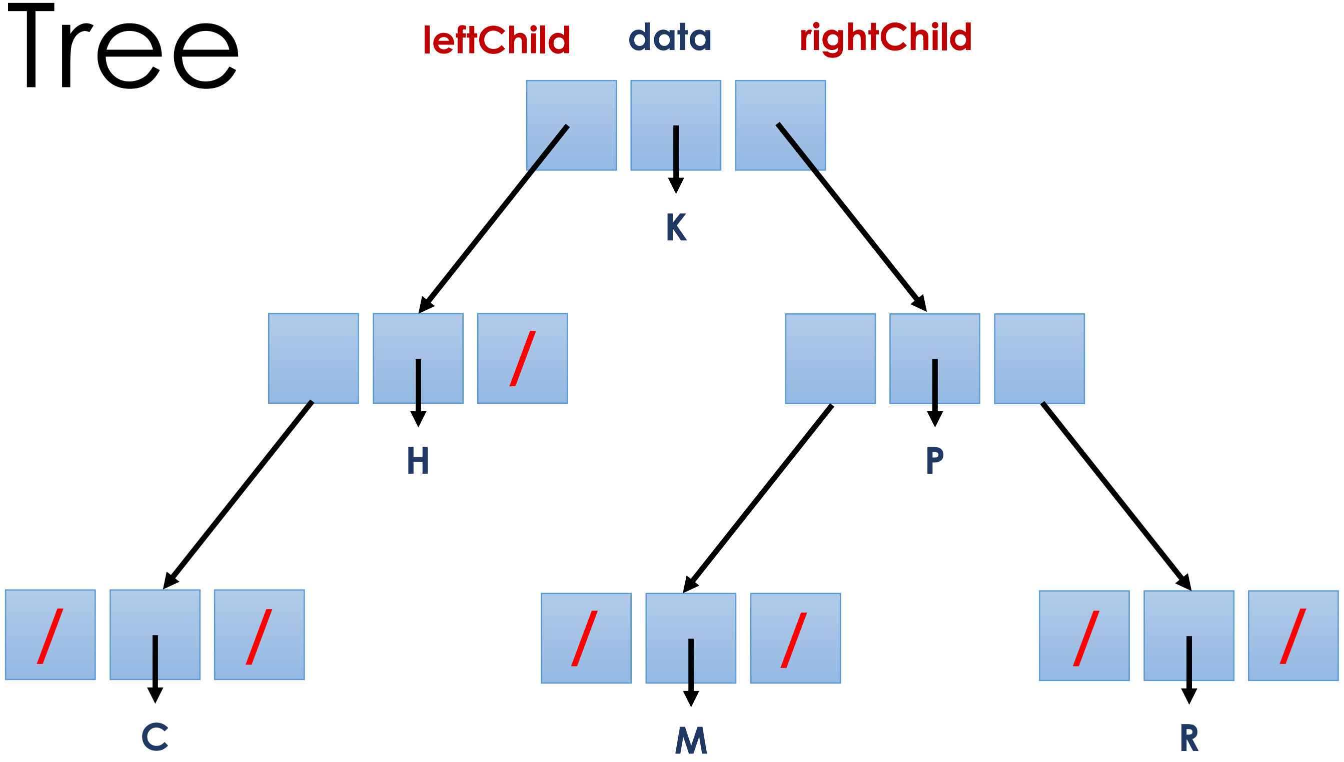
- Cases for node N containing item to be removed
- **Case 3: N has two children**
  - Harder case.
  - One cannot simply remove the parent node.
  - Find a node easier to remove and replace the node elements.

Link-based Implementation of the ADT Binary Search Tree

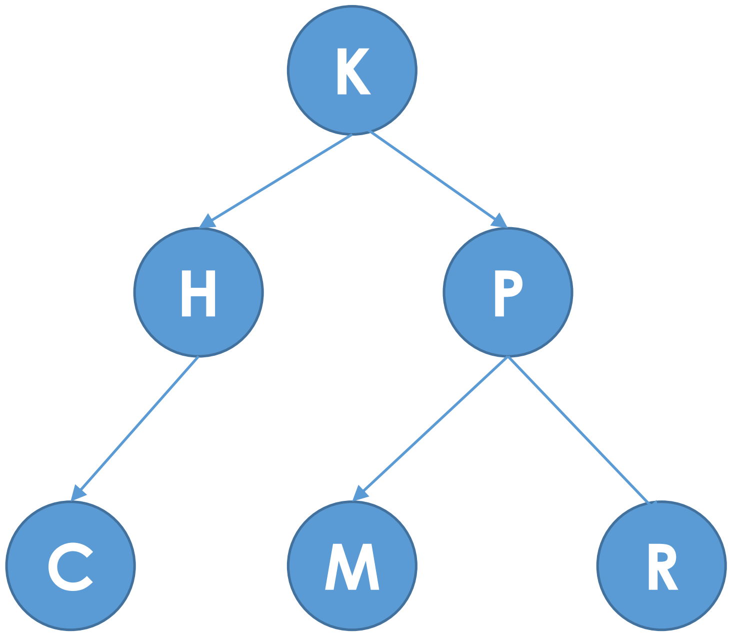
- Cases for node N containing item to be removed
- **Case 3: N has two children**
  - Locate another node M easier to remove from tree than N
  - Copy item that is in M to N
  - Remove M from tree

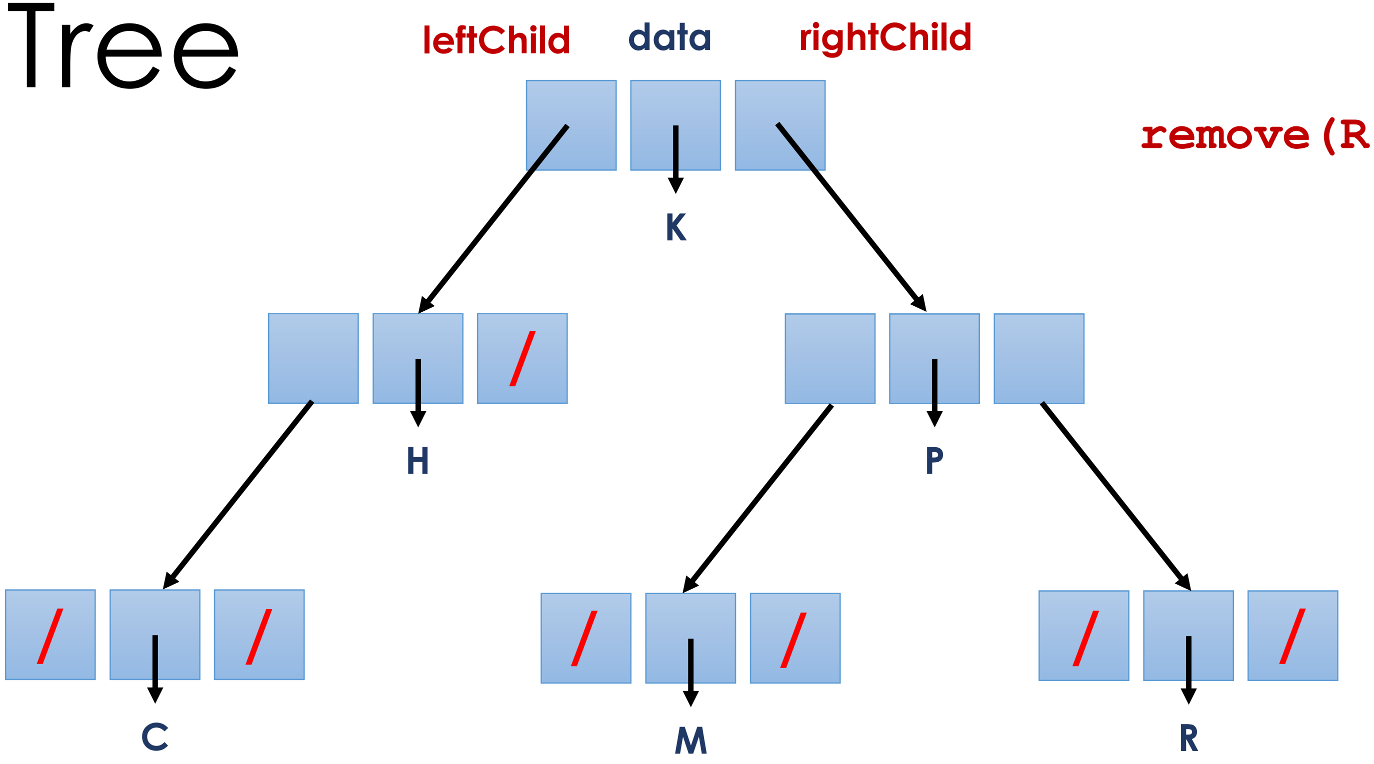
Link-based Implementation of the ADT Binary Search Tree





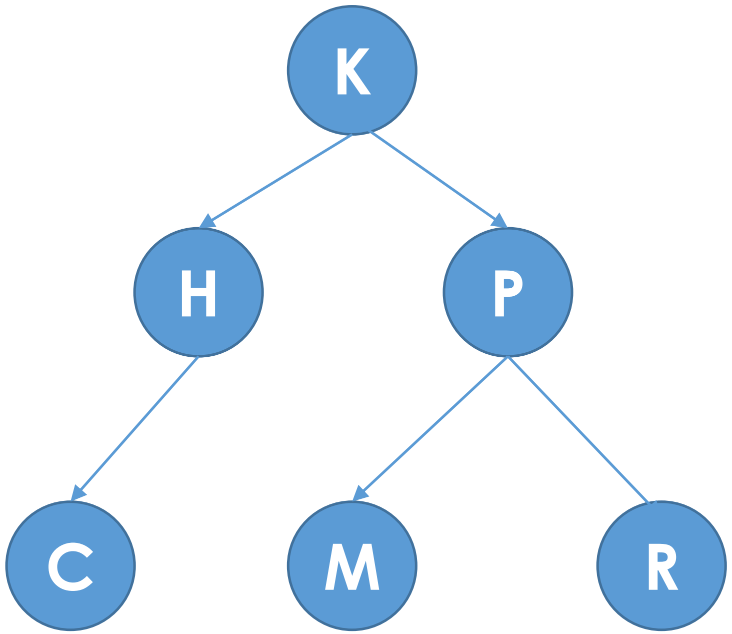
Link-based Implementation of the ADT Binary Search Tree

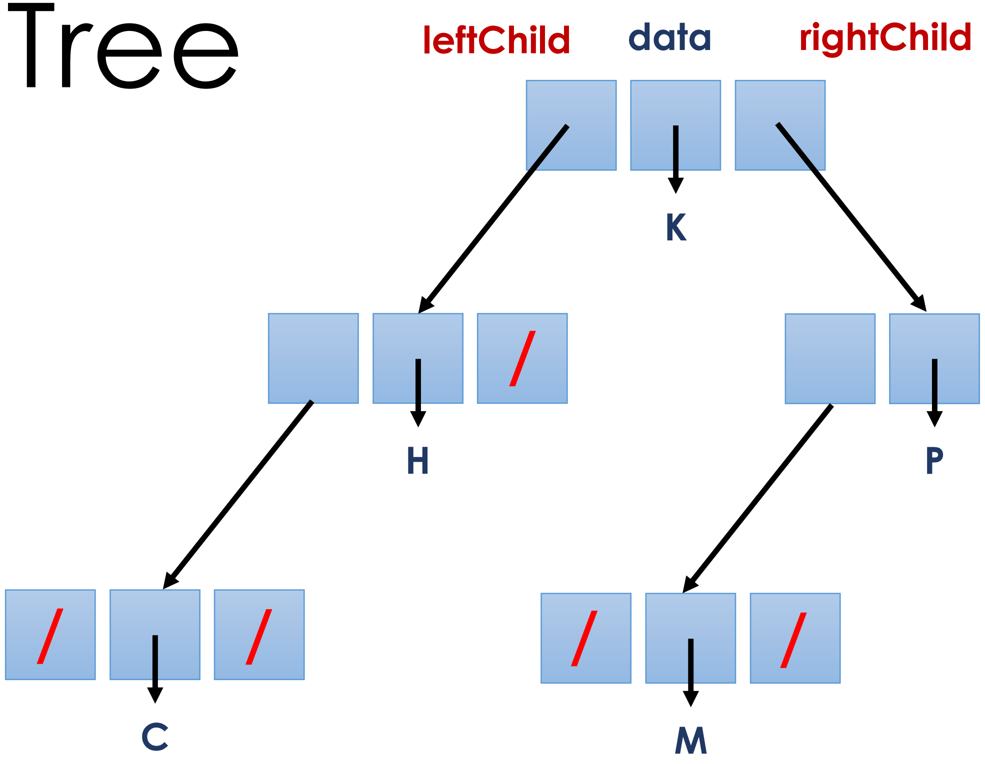




`remove(R)`

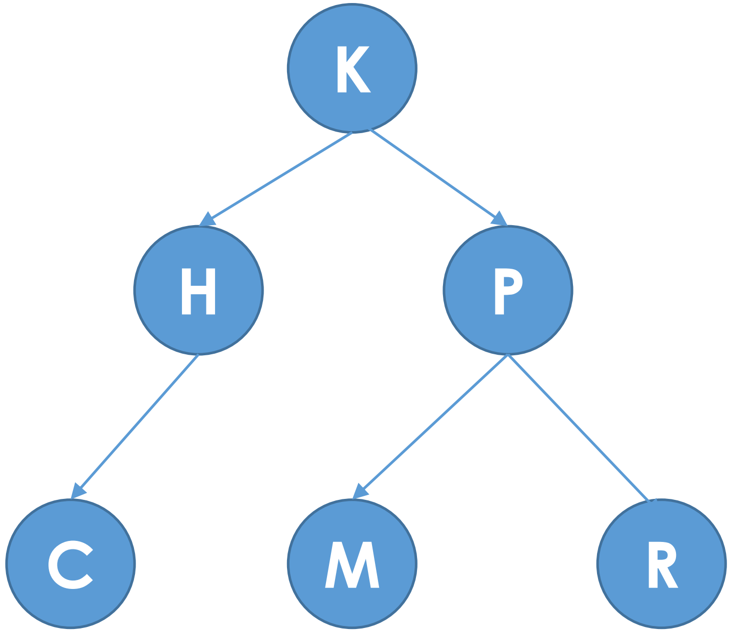
Link-based Implementation of the ADT Binary Search Tree

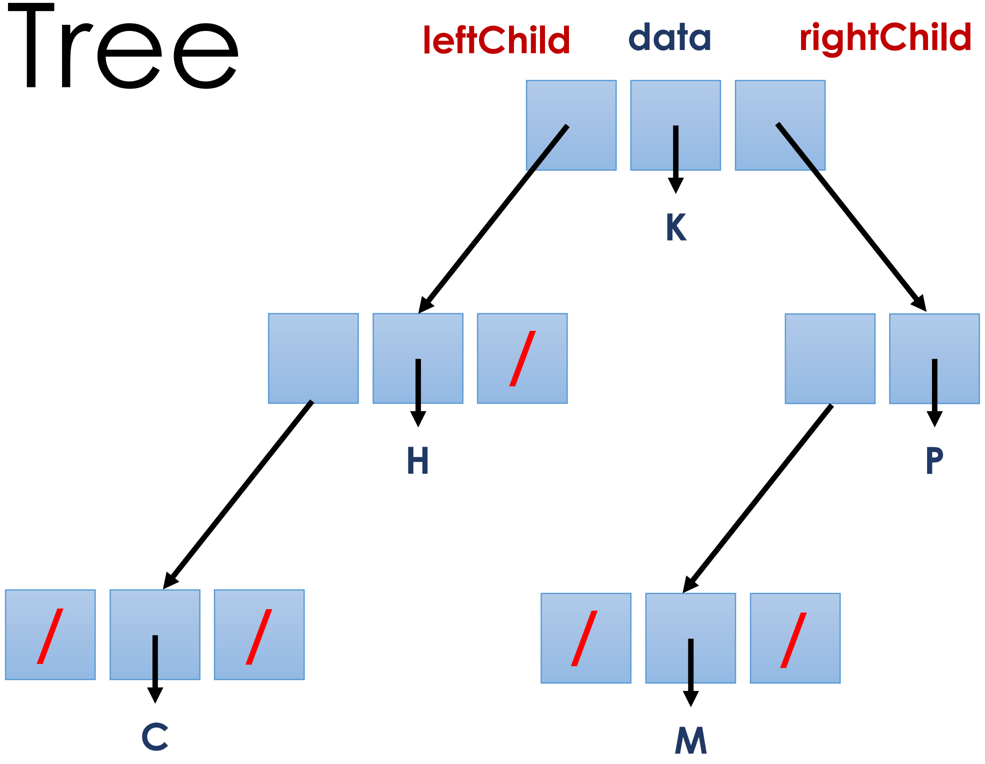




`remove(R)`

Link-based Implementation of the ADT Binary Search Tree

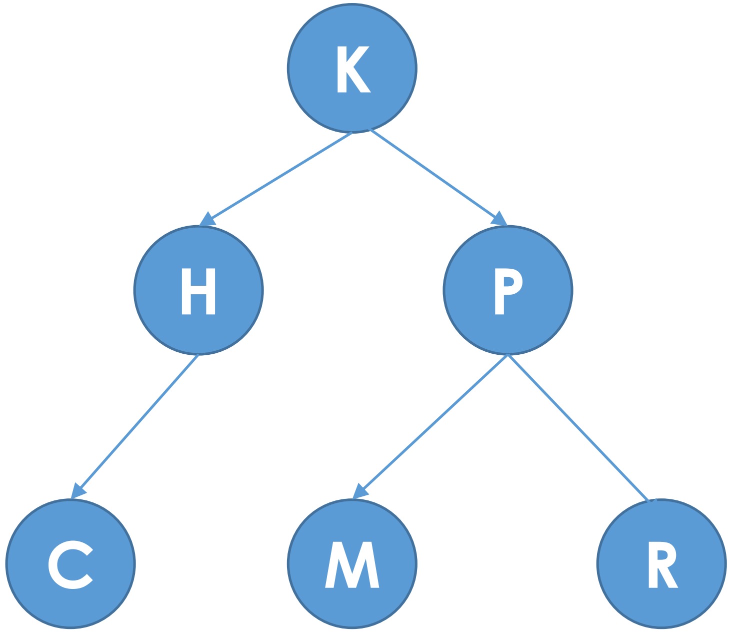


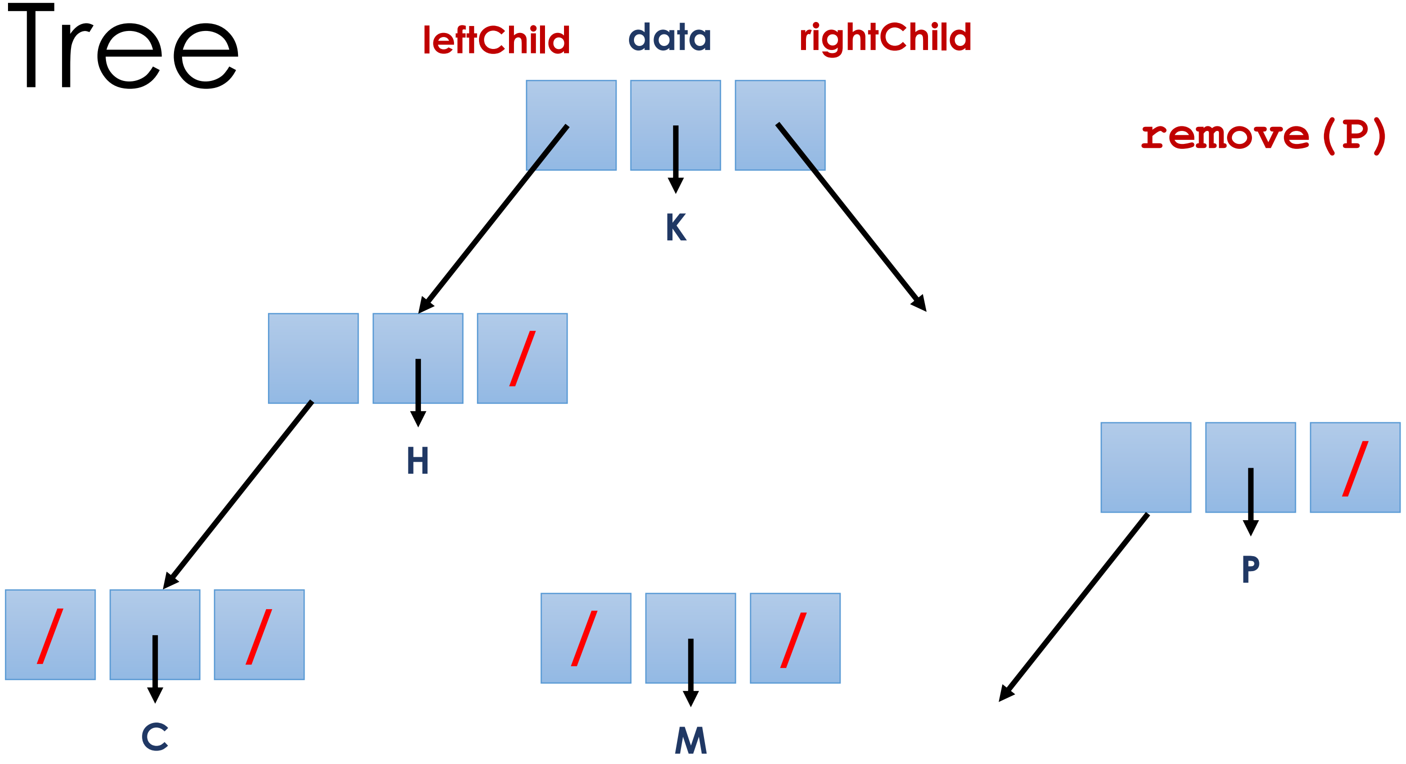


`remove(P)`

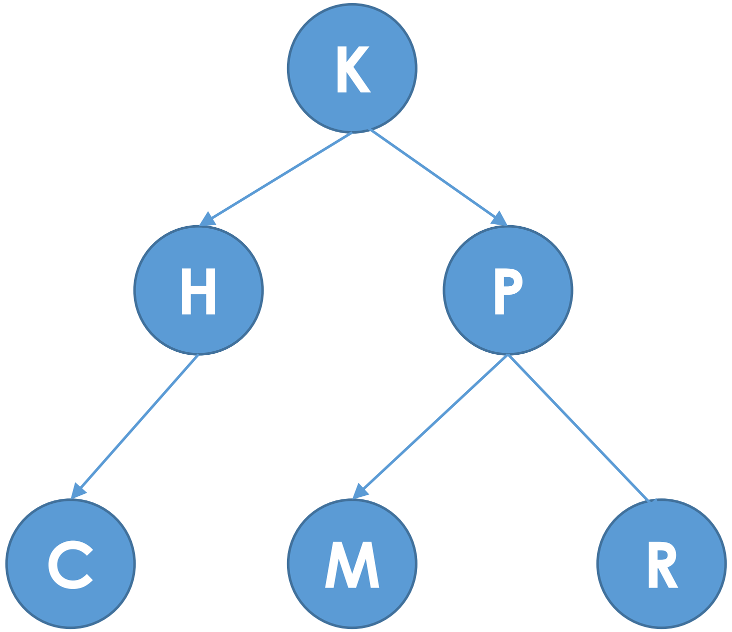
Link-based Implementation of the ADT Binary Search Tree

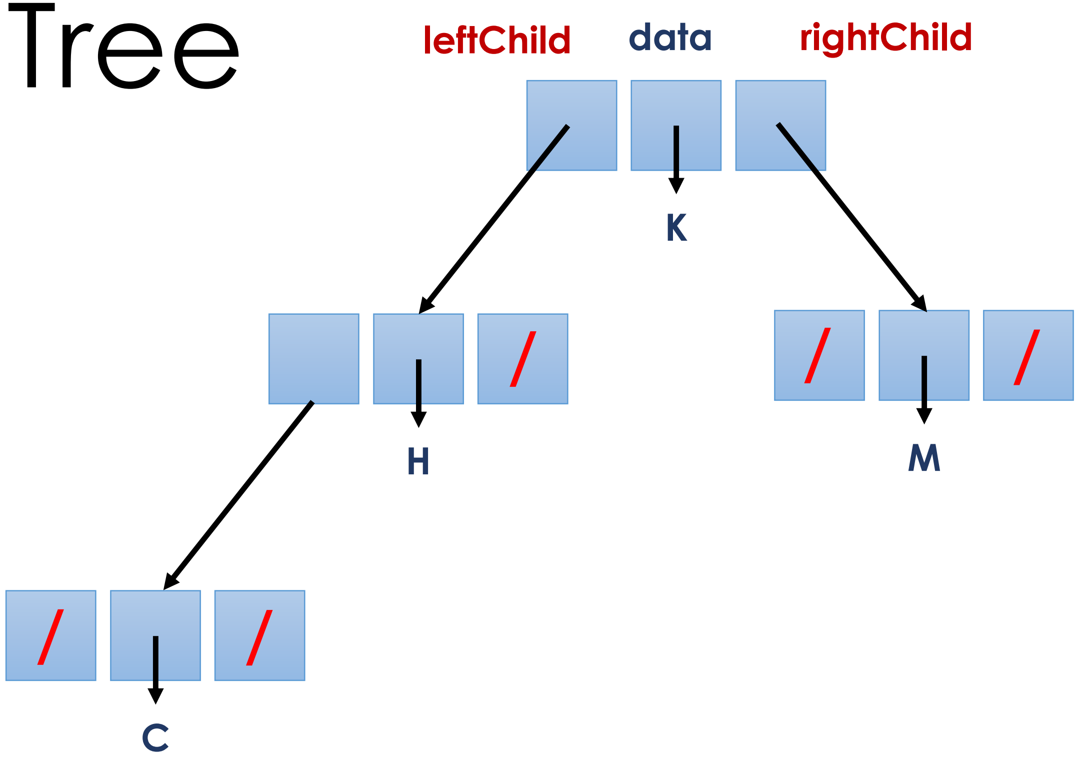
`remove(P)`





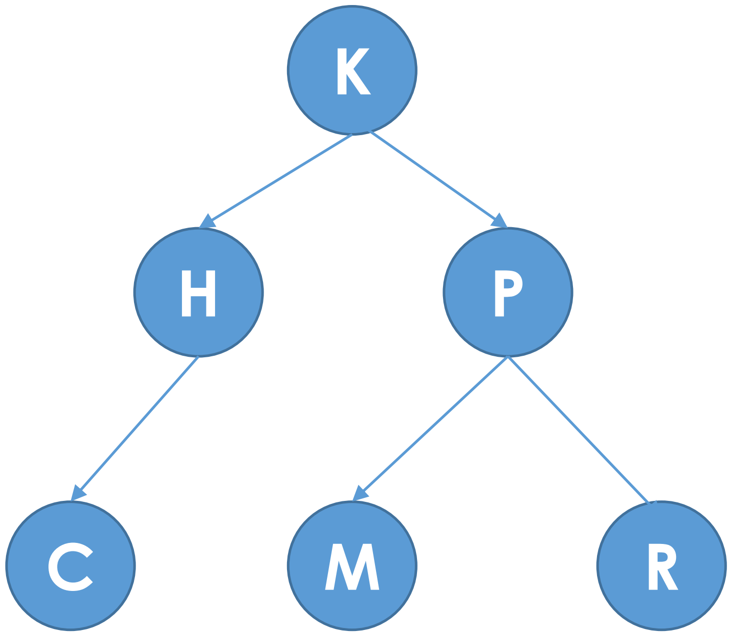
Link-based Implementation of the ADT Binary Search Tree

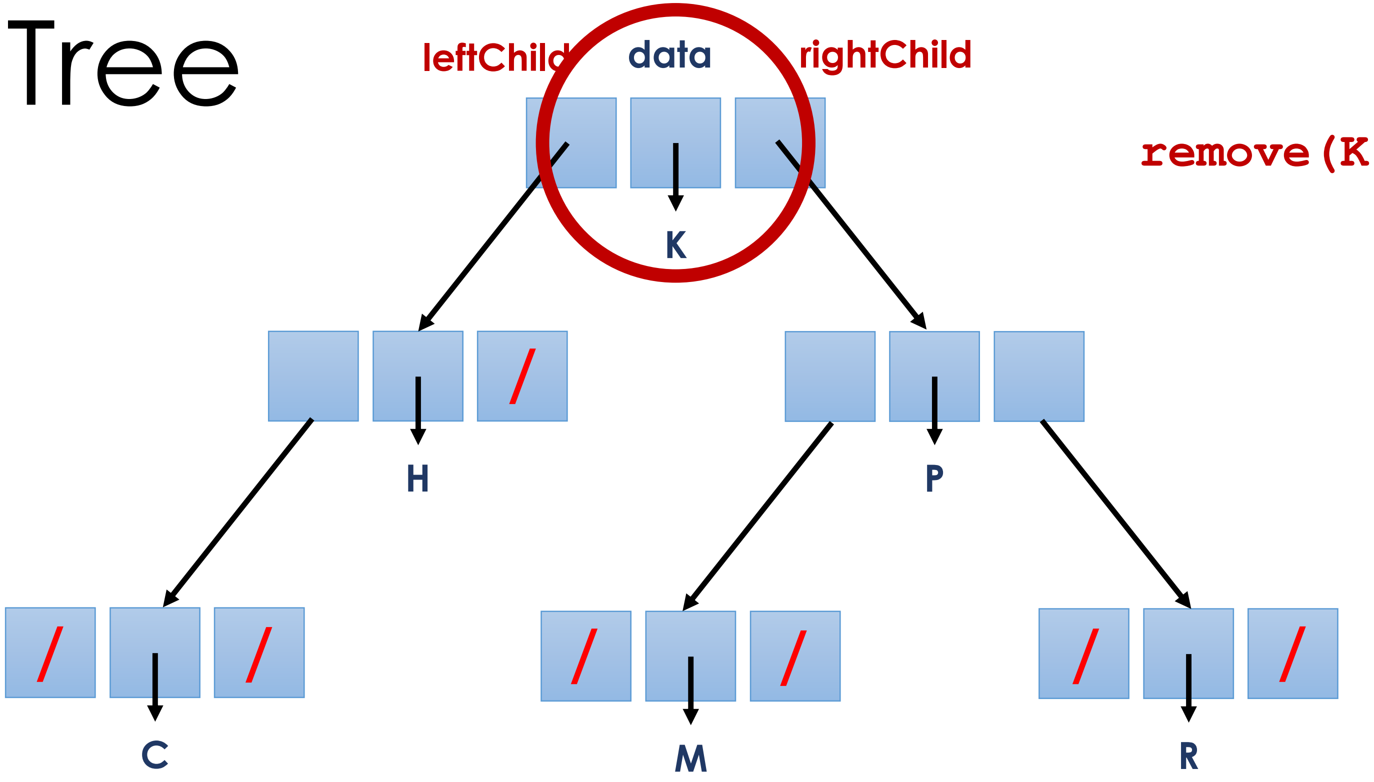




`remove(P)`

Link-based Implementation of the ADT Binary Search Tree

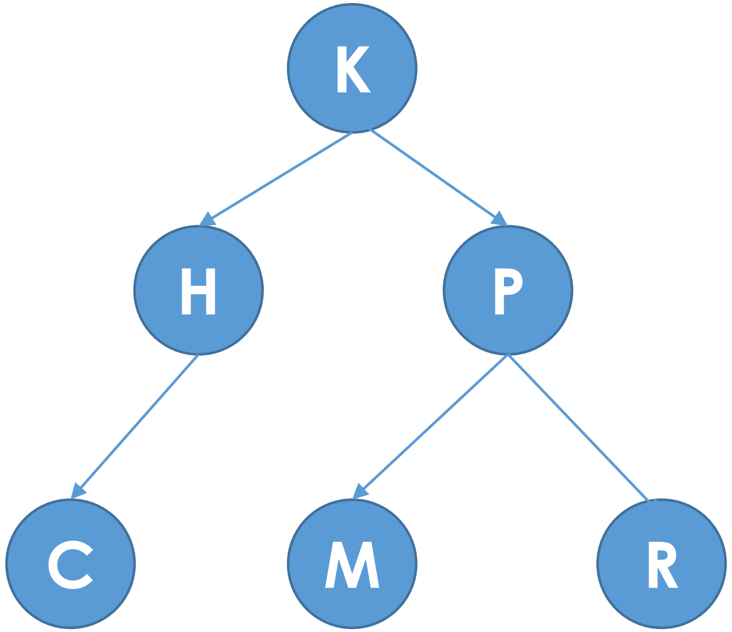


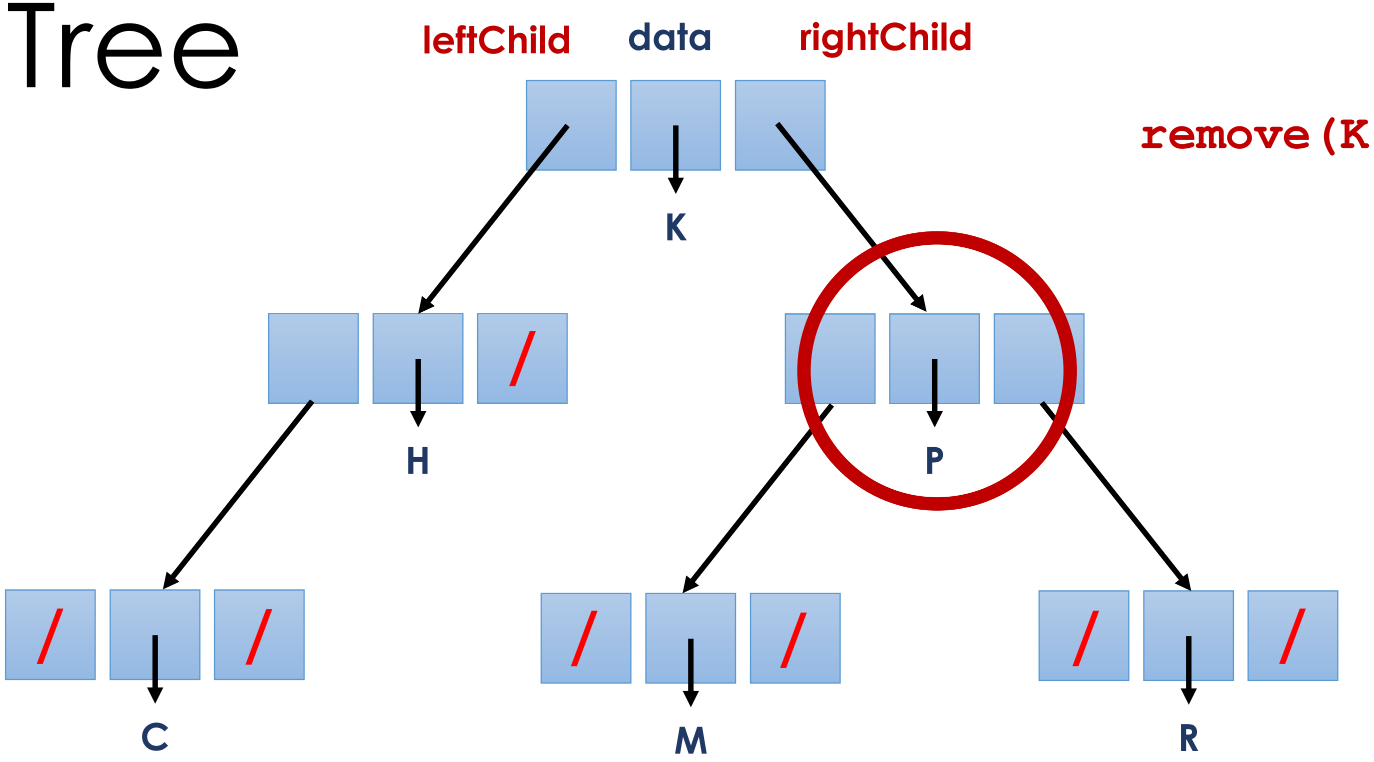


`remove(K)`

Link-based Implementation of the ADT Binary Search Tree

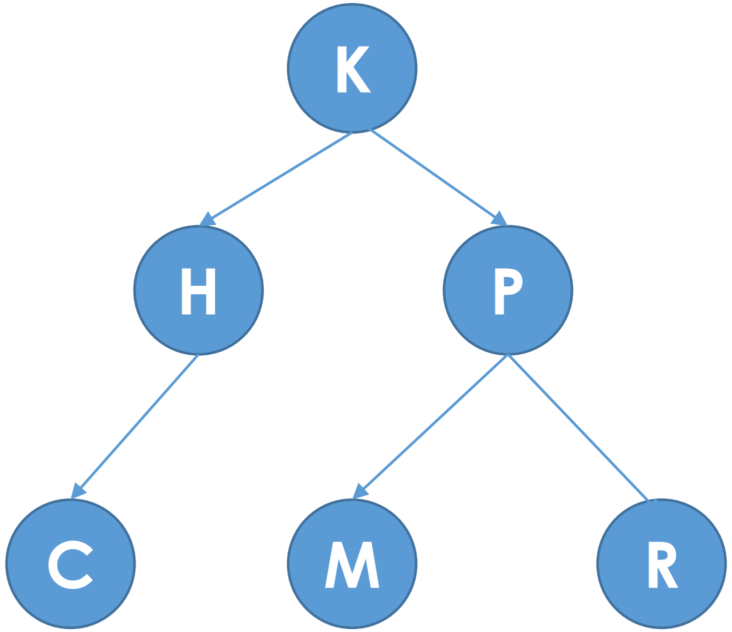
`remove(K)`

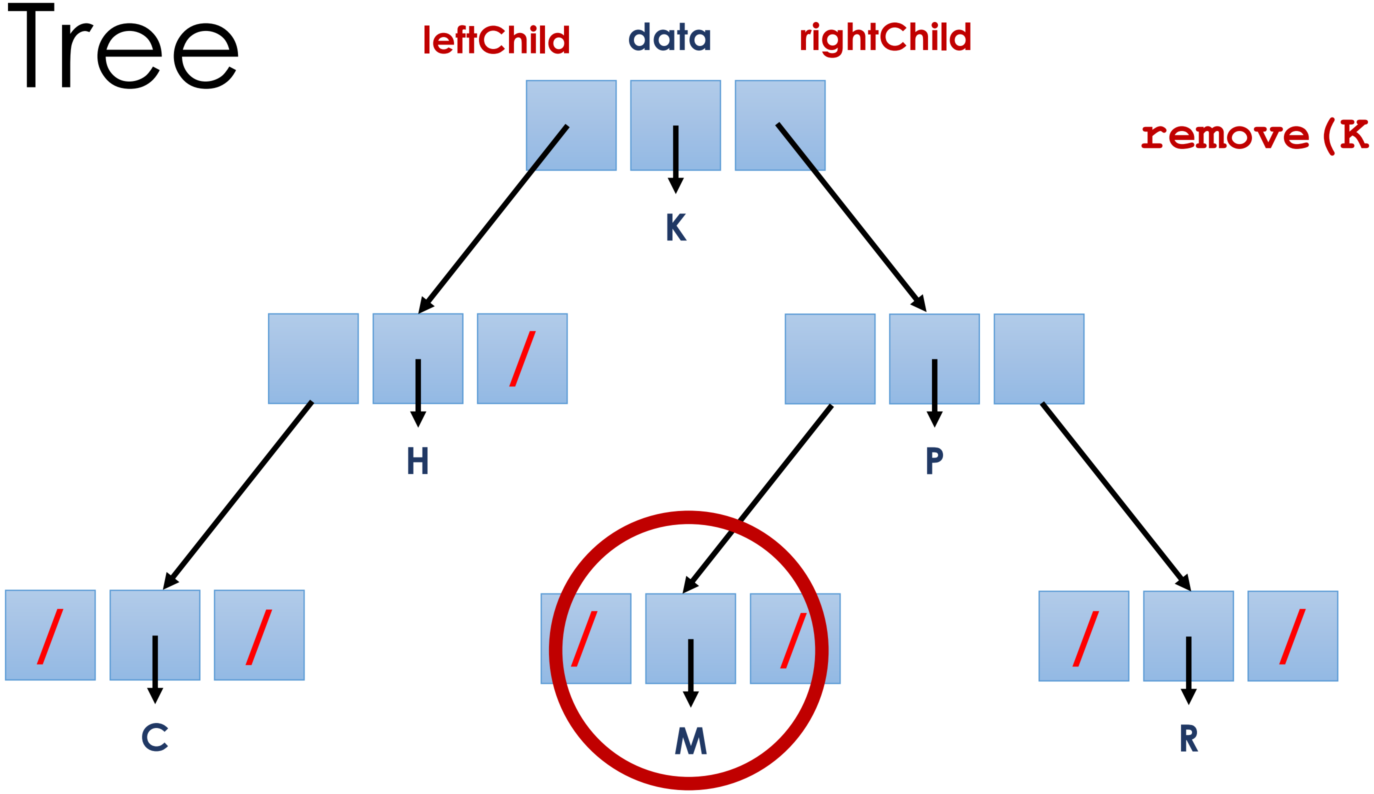




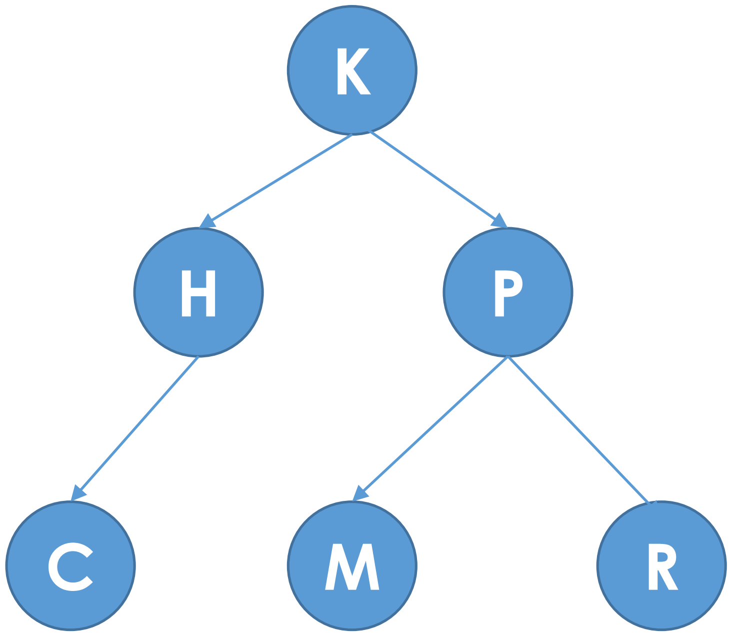
Link-based Implementation of the ADT Binary Search Tree

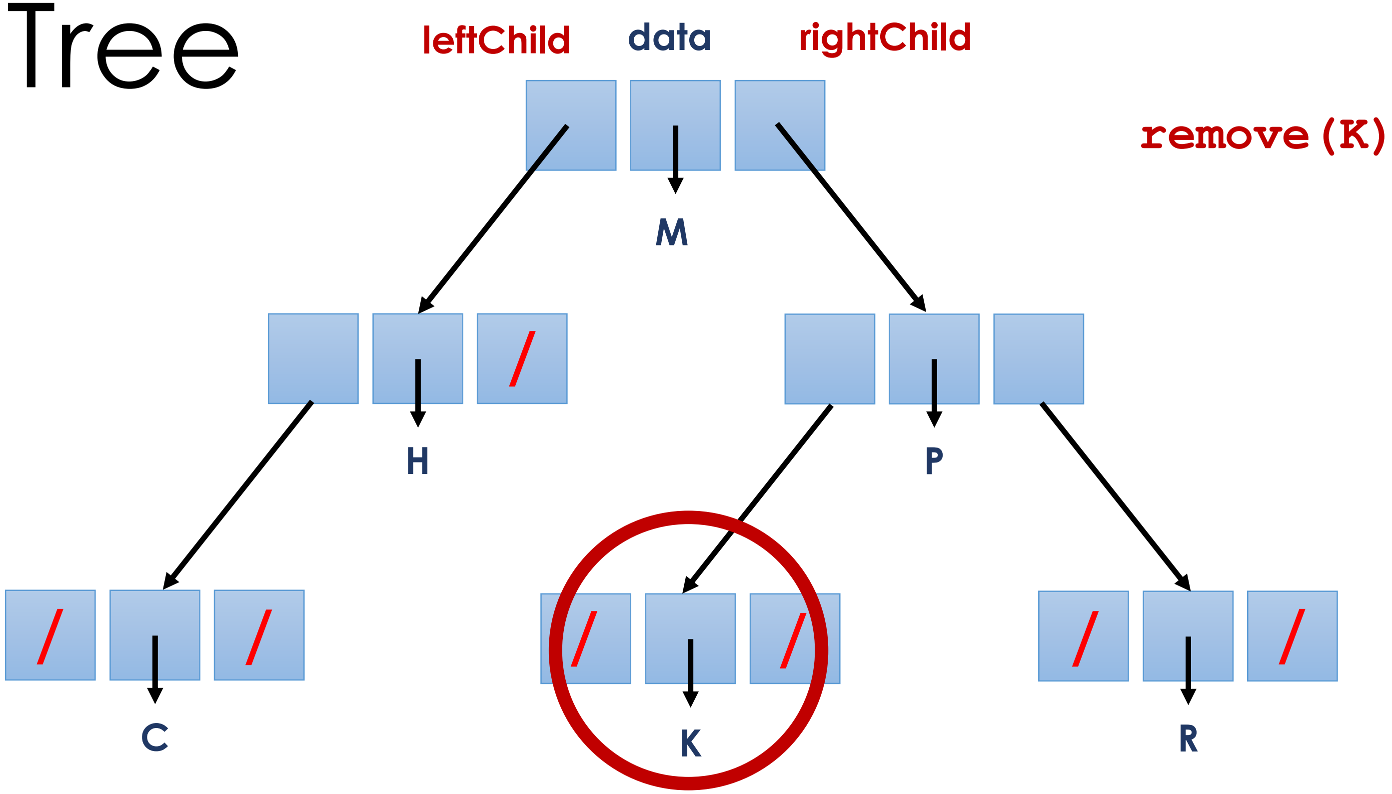
`remove(K)`



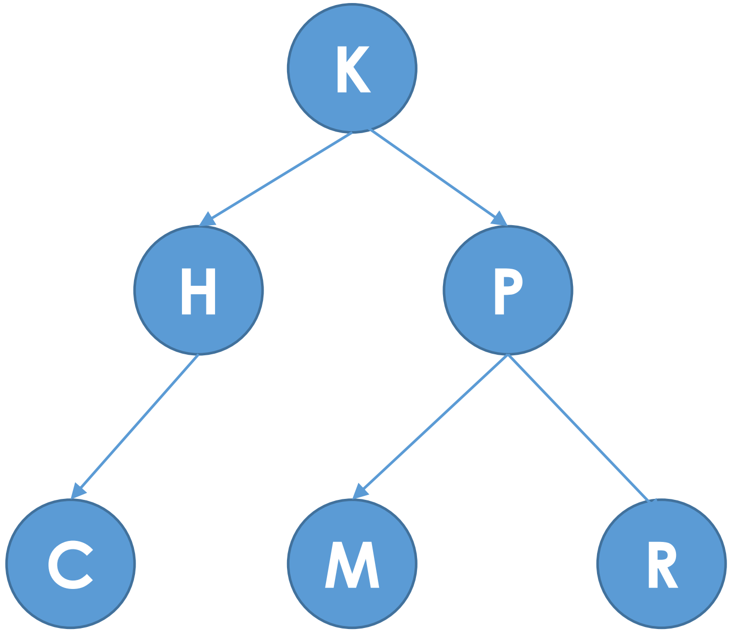


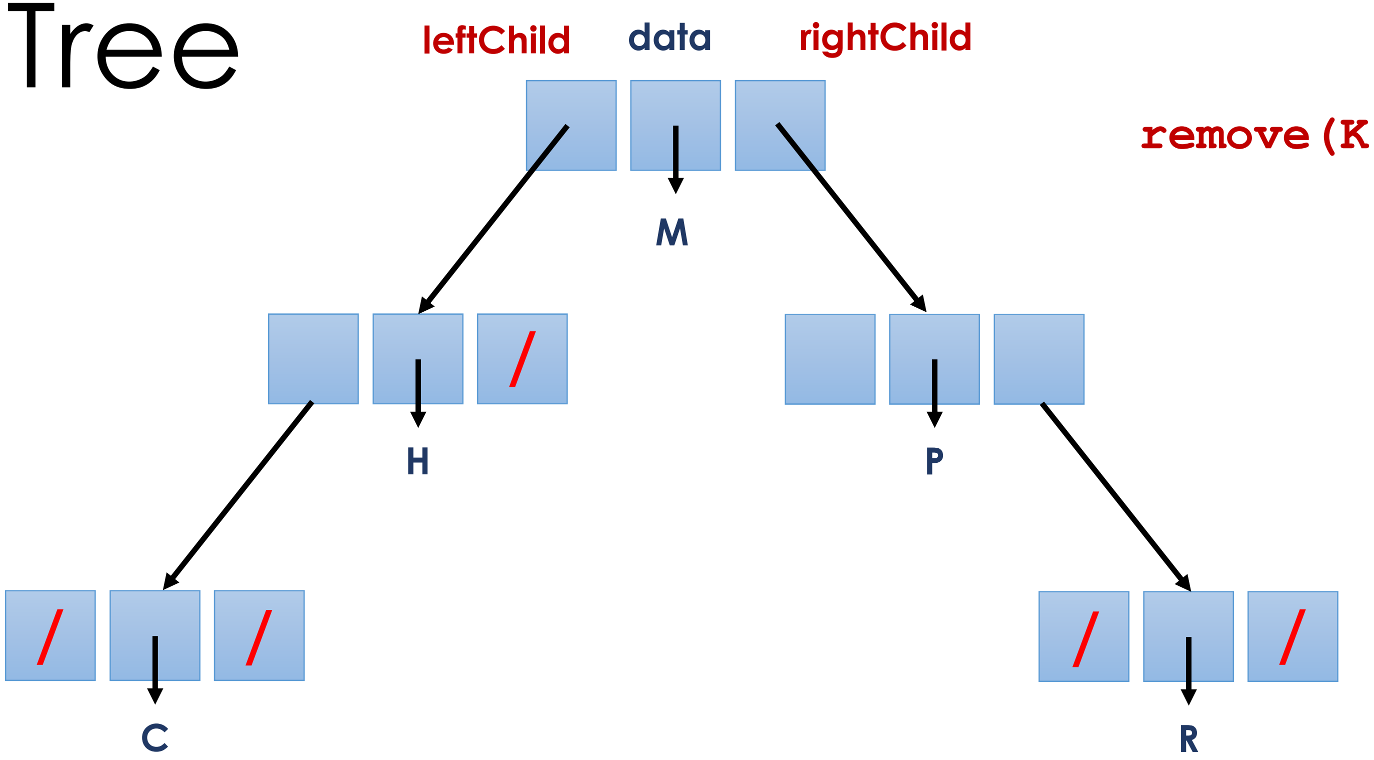
Link-based Implementation of the ADT Binary Search Tree





Link-based Implementation of the ADT Binary Search Tree

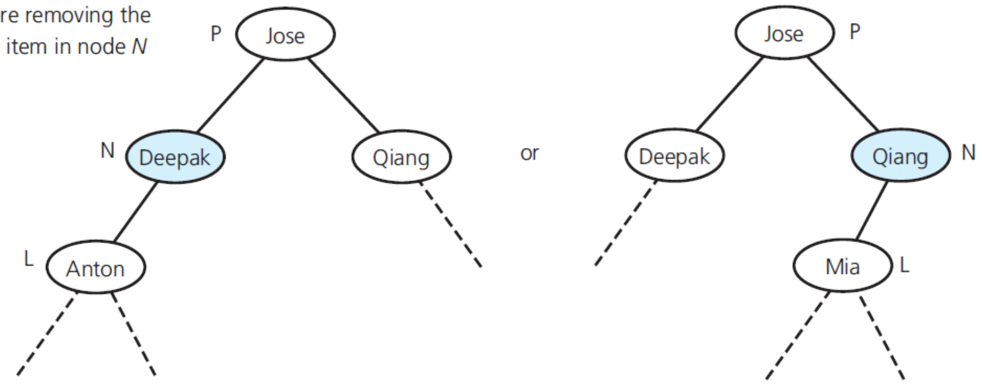




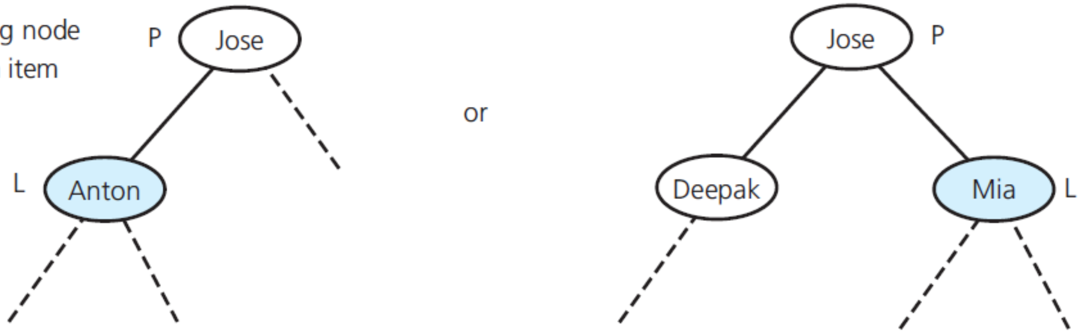
`remove(K)`

Link-based Implementation of the ADT Binary Search Tree

- Case 2 for removeValue: The data item to remove is in a node N that has only a left child and whose parent is node P



(a) Before removing the data item in node *N*



(b) After removing node *N* and its data item

Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)

```
// Removes the given target from the binary search tree to which subTreePtr points.
// Removes a pointer to the node at this tree location after the value is removed.
// Sets isSuccessful to true if the removal is successful, or false otherwise
removeValue(subTreePtr: BinaryNodePointer, target: ItemType, isSuccessful: boolean&): BinaryNodePointer
{
    if (subTreePtr == nullptr)
    {
        isSuccessful = false
    }
    else if (subTreePtr->getItem() == target)
    {
        // Item is in the root of some subtree
        subTreePtr = removeNode(subTreePtr) // Remove the item
        isSuccessful = true
    }
    else if (subTreePtr->getItem() > target)
    {
        // Search the left subtree
        tempPtr = removeValue(subTreePtr->getLeftChildPtr(), target, isSuccessful)
        subTreePtr->setLeftChildPtr(tempPtr)
    }
    else
```

Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)

```
    else
    {
        // Search the right subtree
        tempPtr = removeValue(subTreePtr->getRightChildPtr(), target, isSuccessful)
        subTreePtr->setRightChildPtr(tempPtr)
    }
    return subTreePtr
}

// Removes the data item in the node N to which nodePtr points.
// Returns a pointer to the node at this tree location after the removal
removeNode(nodePtr: BinaryNodePointer): BinaryNodePointer
{
    if (N is a leaf)
    {
        // Remove leaf from the tree
        Delete the node to which nodePtr points (done for us if nodePtr is a smart pointer)
        return nodePtr
    }
    else if (N has only one child C)
```

Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)

```
    else if (N has only one child C)
    {
        // C replaces N as the child of N's parent
        if (C is a left child)
                nodeToConnectPtr = nodePtr->getLeftChildPtr()
        else
                nodeToConnectPtr = nodePtr->getRightChildPtr()

        Delete the node to which nodePtr points (done for us if nodePtr is a smart pointer)
        return nodeToConnectPtr
    }
    else // N has two children
```

Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)

```
     else // N has two children
     {
          // Find the inorder successor of the entry in N: it is in the left subtree rooted at N’s right child
          tempPtr = removeLeftmostNode(nodePtr->getRightChildPtr(), newNodeValue)
          nodePtr->setRightChildPtr(tempPtr)
          nodePtr->setItem(newNodeValue) // Put replacement value in nodeN
          return nodePtr
     }
}

// Removes the leftmost node in the left subtree of the node pointed to by nodePtr
// Sets inorder Successor to the value in this node
// Returns a pointer to the revised subtree
removeLeftmostNode(nodePtr: BinaryNodePointer, inorderSuccessor: ItemType&): BinaryNodePointer
```

Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)

```
removeLeftmostNode(nodePtr: BinaryNodePointer, inorderSuccessor: ItemType&): BinaryNodePointer
{
    if (nodePtr->getLeftChildPtr() == nullptr)
    {
        // This is the node you want; it has no left child, but it might have a right subtree
        inorderSuccessor = nodePtr->getItem()
        return removeNode(nodePtr)
    }
    else
    {
        tempPtr = removeLeftmostNode(nodePtr->getLeftChildPtr(), inorderSuccessor)
        nodePtr->setLeftChildPtr(tempPtr)
        return nodePtr
    }
}
```

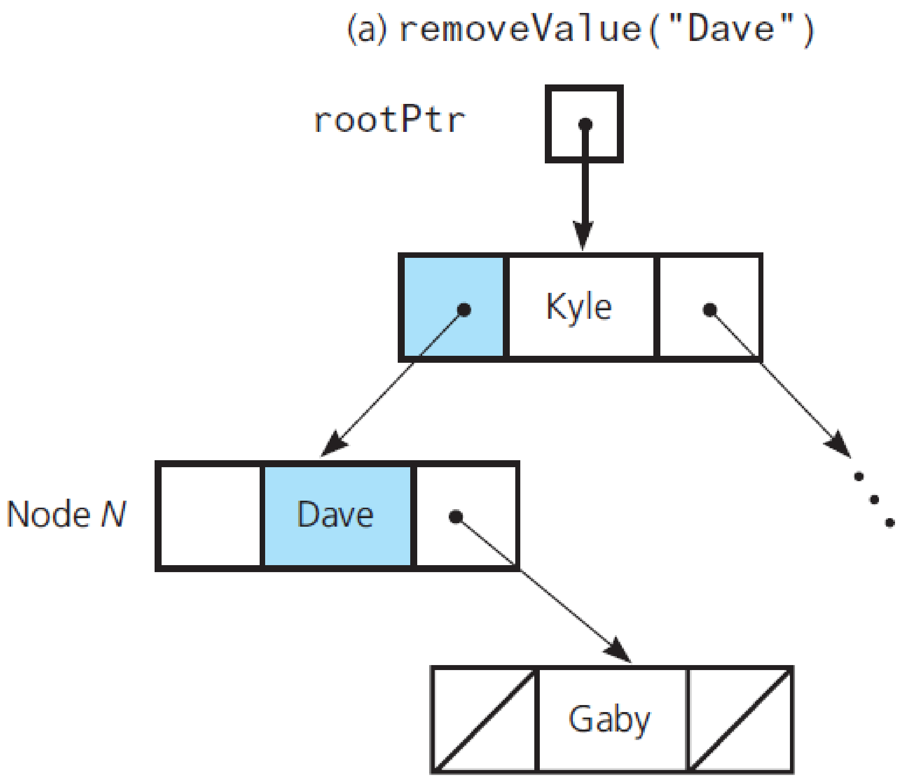
Link-based Implementation of the ADT Binary Search Tree

- Final draft of the **removal** algorithm (method **removeNode** uses **findSuccesorNode**)
- Public method remove

```
// Removes the given data from this binary search tree
remove(target: ItemType): boolean
{
    isSuccessful = false
    rootPtr = removeValue(rootPtr, target, isSuccessful)
    return isSuccessful
}
```

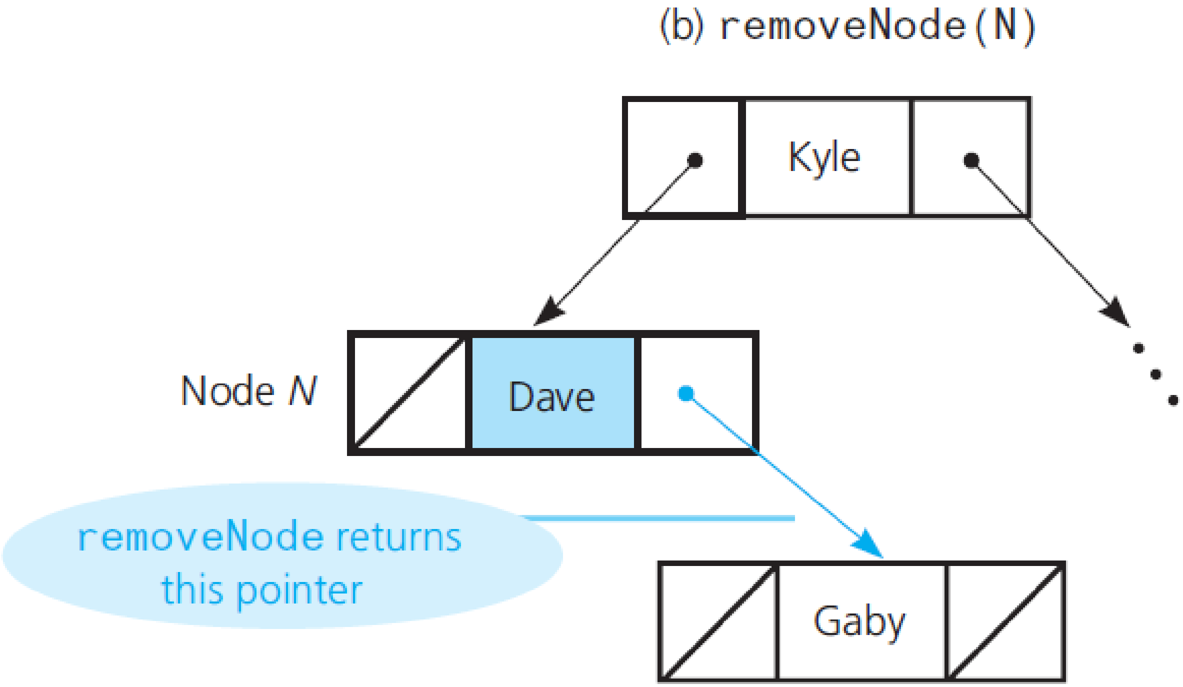
Link-based Implementation of the ADT Binary Search Tree

- Recursive removal of node N



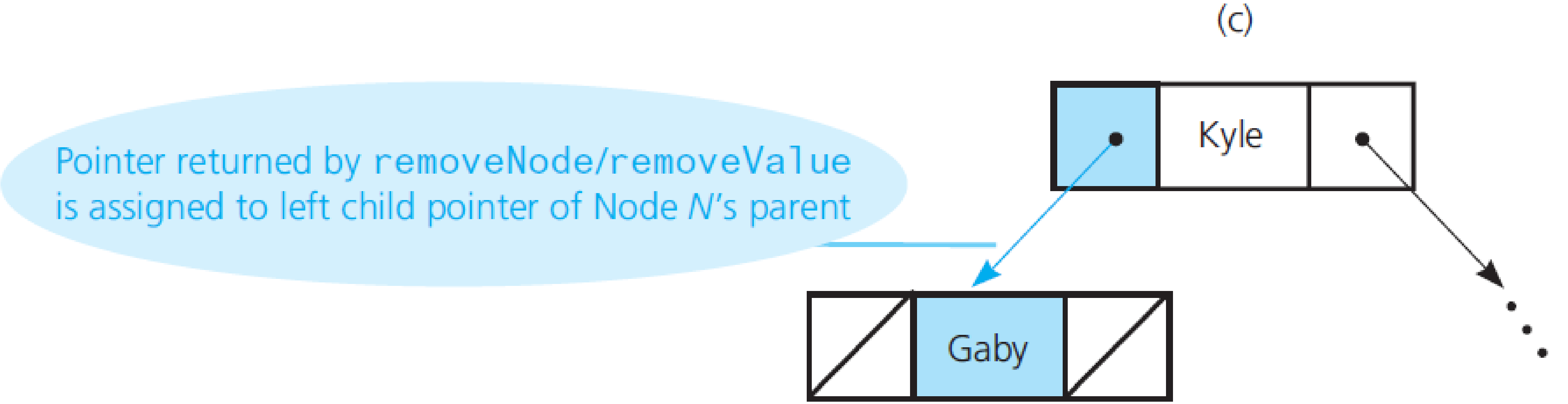
Link-based Implementation of the ADT Binary Search Tree

- Recursive removal of node N (**cont**)



Link-based Implementation of the ADT Binary Search Tree

- Recursive removal of node N (**cont**)



Link-based Implementation of the ADT Binary Search Tree

- Algorithm **findNode**

```
// Locates the node in the binary search tree to which subTreePtr points and that contains the value target.
// Returns either a pointer to the located node or nullptr if such a node is not found.

findNode(subTreePtr: BinaryNodePointer, target: ItemType): BinaryNodePointer
{
    if (subTreePtr == nullptr)
        return nullptr            // Not found

    else if (subTreePtr->getItem() == target)
        return subTreePtr;        // Found
    else if (subTreePtr->getItem() > target)
        // Search left subtree
        return findNode(subTreePtr->getLeftChildPtr(), target)
    else
        // Search right subtree
        return findNode(subTreePtr->getRightChildPtr(), target)
}
```

Link-based Implementation of the ADT Binary Search Tree

- Algorithm **findNode**
- The method **findNode** is designed to be utilized by a method **getEntry**.
- The operation **getEntry** must return the data item with the desired value if it exists; otherwise it must throw an exception **NotFoundException**.
- The method, therefore, calls **findNode** and checks its return value.
- If the desired target is found, **getEntry** returns it.
- If **findNode** returns **nullptr**, **getEntry** throws an exception.

The Class BinarySearchTree

- A header file for the link-based implementation of the class BinarySearchTree

```
// Link-based implementation of the ADT binary search tree.

#ifndef BINARY_SEARCH_TREE_
#define BINARY_SEARCH_TREE_

#include “BinaryTreeInterface.h”
#include “BinaryNode.h”
#inlucde “BinaryNodeTree.h”
#include “NotFoundException.h”
#include “PrecondViolatedExcept.h”
#include <memory>

template<class ItemType>
class BinarySearchTree: public BinaryNodeTree<ItemType>
{
private:
    std::shared_ptr<BinaryNode<ItemType>> rootPtr;
```

The Class BinarySearchTree

- A header file for the link-based implementation of the class BinarySearchTree

```
protected:
    // PROTECTED UTILITY METHODS SECTION:
    // RECURSIVE HELPER METHODS FOR THE PUBLIC METHODS

    // Places a given new node at its proper position in this binary search tree
    auto placeNode(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                       std::shared_ptr<BinaryNode<ItemType>> newNode);

    // Removes the given target value from the tree while maintaining a binary search tree
    auto removeValue(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                       const ItemType target,
                       bool& isSuccessful) override;

    // Removes a given node from a tree while maintaining a binary search tree
    auto removeNode(std::shared_ptr<BinaryNode<ItemType>> nodePtr);
```

Why protected? Why not private?

The Class BinarySearchTree

- A header file for the link-based implementation of the class BinarySearchTree

```
    // Removes the leftmost node in the left subtree of the node pointed to by nodePtr
    // Sets inorderSuccessor to the value in this node
    // Returns a pointer to the revised subtree
    auto removeLeftmostNode(std::shared_ptr<BinaryNode<ItemType>>subTreePtr,
                                  ItemType& inorderSuccessor);

    // Returns a pointer to the node containing the given value, or nullptr if not found
    auto findNode(std::shared_ptr<BinaryNode<ItemType>> treePtr,
                          const ItemType& target) const;

public:
    // CONSTRUCTOR AND DESTRUCTOR SECTION
    BinarySearchTree();
    BinarySearchTree(const ItemType& rootItem);
    BinarySearchTree(const BinarySearchTree<ItemType>& tree);
    virtual ~BinarySearchTree();
```

The Class BinarySearchTree

- A header file for the link-based implementation of the class BinarySearchTree

```
// PUBLIC METHODS SECTION
bool isEmpty() const;
int getHeight() const;
int getNumberOfNodes() const;
ItemType getRootData() const throw(PrecondViolatedExcept);
void setRootData(const ItemType& newData);
bool add(const ItemType& newEntry);
bool remove(const ItemType& target);
void clear();
ItemType getEntry(const ItemType& anEntry) const throw(NotFoundException)
bool contains(const ItemType& anEntry) const;
```

The Class BinarySearchTree

- A header file for the link-based implementation of the class BinarySearchTree

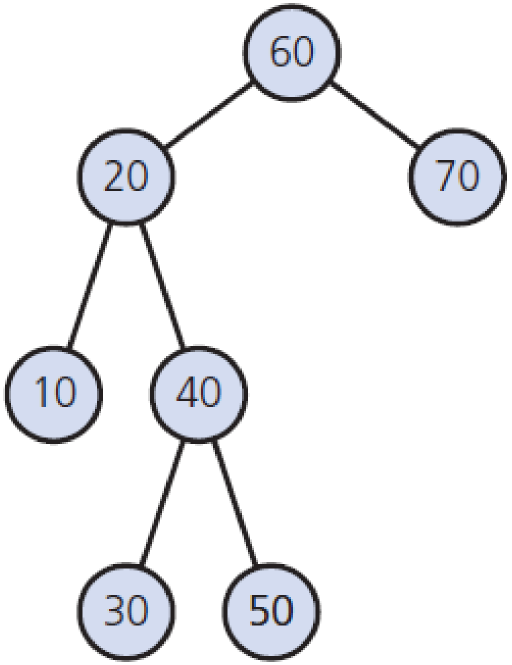
```
    // PUBLIC TRAVERSALS SECTION
    void preorderTraverse(void visit(ItemType&)) const;
    void inorderTraverse(void visit(ItemType&)) const;
    void postorderTraverse(void visit(ItemType&)) const;

    // OVERLOADED OPERATION SECTION
    BinarySearchTree<ItemType>& operator = (const BinarySearchTree<ItemType>& rightHandSide);
}; // end BinarySearchTree
#include “BinarySearchTree.cpp”
#endif
```

Saving a Binary Search Tree in a File

- Saving the binary search tree involves saving its values and restoring it correctly.
- Saving a binary search tree and then restoring it to its original shape:
    - First algorithm restores a binary search tree to exactly the same shape it had before it was saved

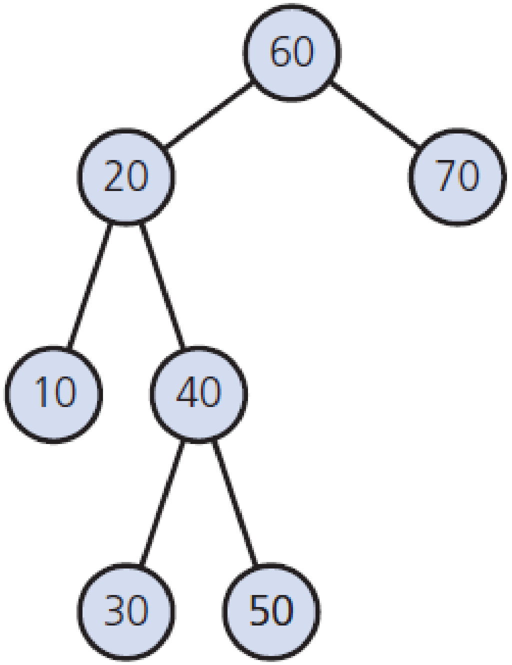
Saving a Binary Search Tree in a File

- An initially empty binary search tree after the addition of 60, 20, 10, 40, 30, 50, and 70



Saving a Binary Search Tree in a File

- An initially empty binary search tree after the addition of 60, 20, 10, 40, 30, 50, and 70



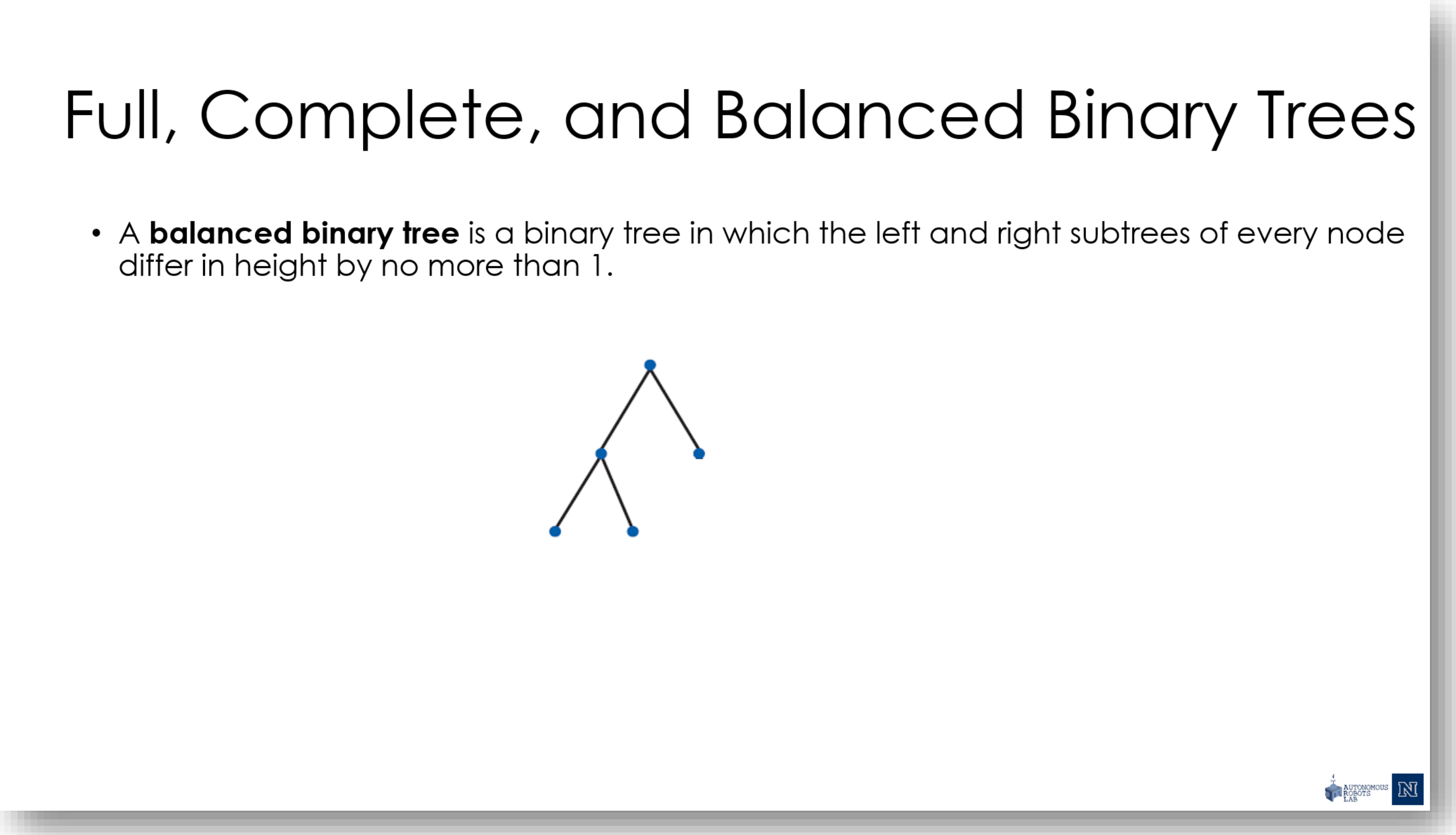
- If you now use the add method to add these values to a BST you recover the original tree.

Saving a Binary Search Tree in a File

- Saving a binary search tree and then restoring it to a balanced shape:
  - Do we necessarily want to recover its original shape?
  - We can use the same data to crate set of BSTs
  - The shape of the BST affects efficiency of operations
  - We decide to recover a balanced tree.

Saving a Binary Search Tree in a File

- Saving a binary search tree and then restoring it to a balanced shape:
  - Do we necess
  - We can use th
  - The shape of t
  - We decide to
  - **Remember:**



Full, Complete, and Balanced Binary Trees

- A **balanced binary tree** is a binary tree in which the left and right subtrees of every node differ in height by no more than 1.

Saving a Binary Search Tree in a File

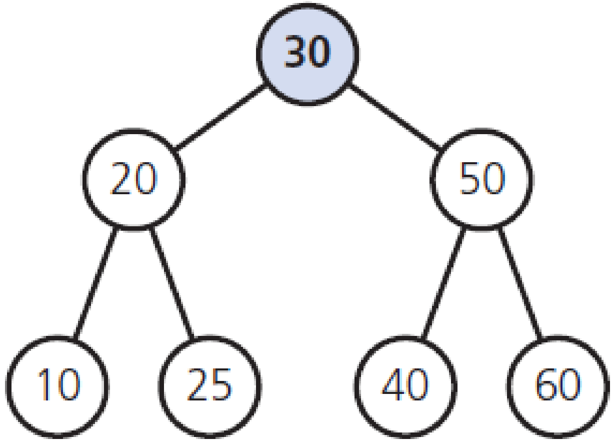
- Use preorder traversal to save binary search tree in a file
  - Restore to original shape by using method add

Saving a Binary Search Tree in a File

- Use preorder traversal to save binary search tree in a file
    - Restore to original shape by using method add
- Balanced binary search tree increases efficiency of ADT operations

Saving a Binary Search Tree in a File

- **Gaining Insight:** A **full tree** saved in a file by using inorder traversal

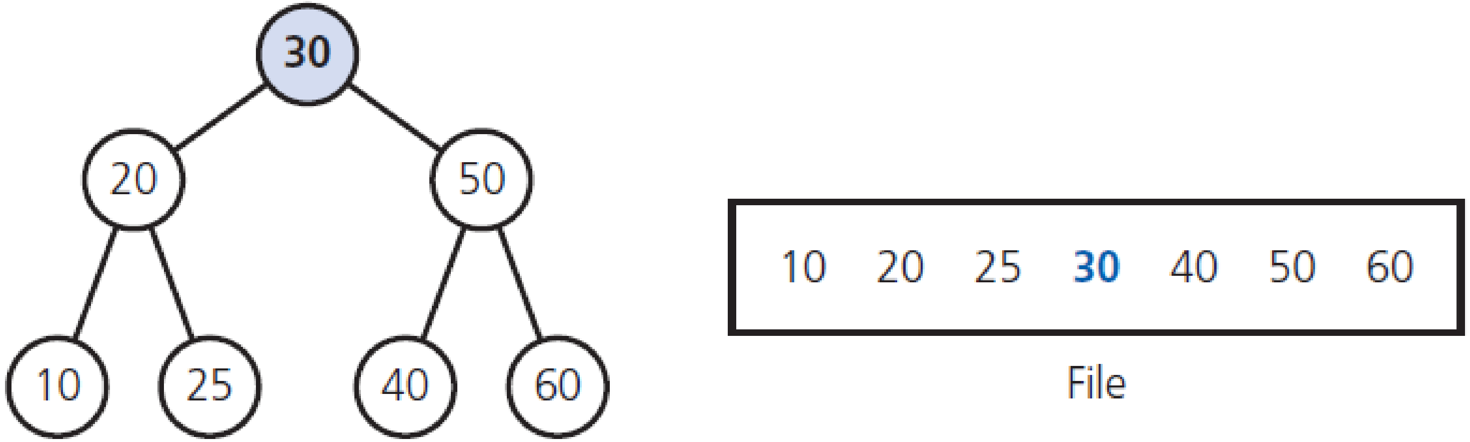


10 20 25 **30** 40 50 60

File

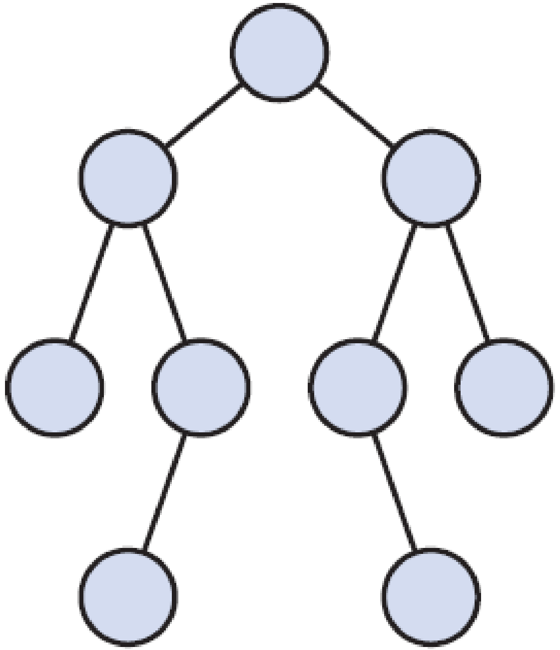
Saving a Binary Search Tree in a File

- **Gaining Insight:** A **full tree** saved in a file by using inorder traversal
  - But trees are not always full
  - What we care most is to have a tree of minimum height



Saving a Binary Search Tree in a File

- **Gaining Insight:** A tree of minimum height that is not complete



The Class BinarySearchTree

- Building a minimum-height binary search tree

```
// Builds a minimum-height binary search tree from n sorted values in a file.
// Returns a pointer to the tree's root.
readTree(treePtr: BinaryNodePointer, n: integer): BinaryNodePointer
{
    if (n > 0)
    {
        treePtr = pointer to new node with nullptr as its child pointers

        // Construct the left subtree
        leftPtr = readTree(treePtr->getLeftChildPtr(), n / 2)
        treePtr->setLeftChildPtr(leftPtr)

        // Get the data item for this node
        rootItem = next data item from file
        treePtr->setItem(rootItem)

        // Construct the right subtree
        rightPtr = readTree(treePtr->getRightChildPtr(), (n-1) / 2)
        treePtr->getRightChildPtr(rightPtr)

        return treePtr
    }
        return nullptr
}
```

The Class BinarySearchTree

- Building a minimum-height binary search tree
- In conclusion:
  - Easy to build a balanced tree if the data in the file is sorted.
  - You need n so that you can determine the middle and, in turn, the number of nodes in the left and right subtrees of the tree’s root.
  - Knowing these numbers is a simple matter of counting nodes as you traverse the tree and then saving the number in a file that the restore operation can read.

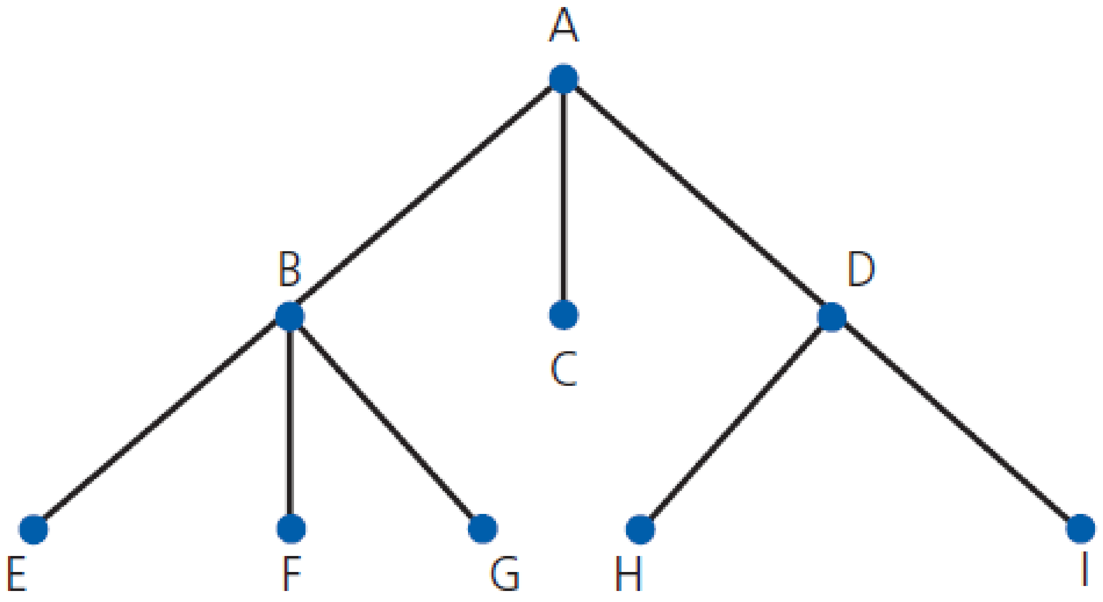
Tree Sort

- Tree sort uses a binary search tree

```
// Sorts the integers in an array into ascending order
treeSort(anArray: array, n: integer)
{
    Add anArray’s entries to a binary search tree bst
    Traverse bst in inorder. As you visit bst’s nodes, copy their data items into successive locations of anArray
}
```

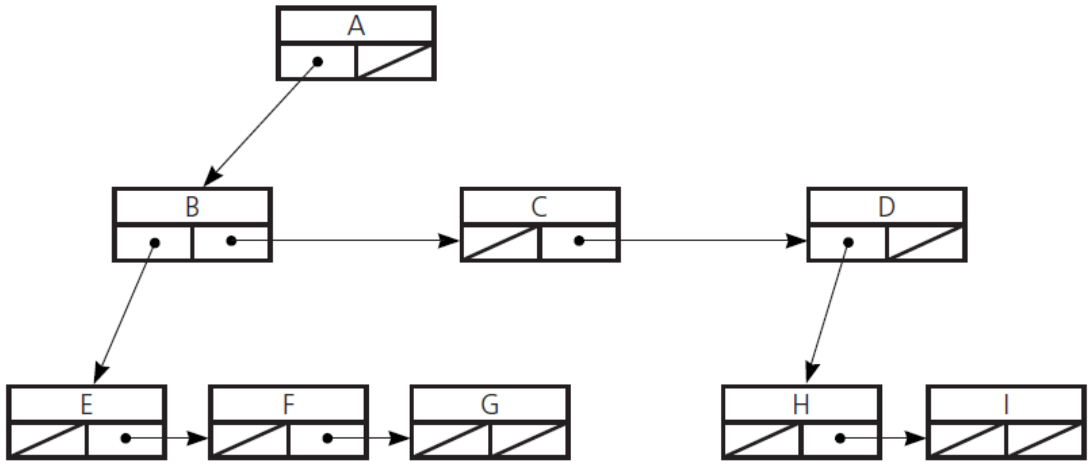
General Trees

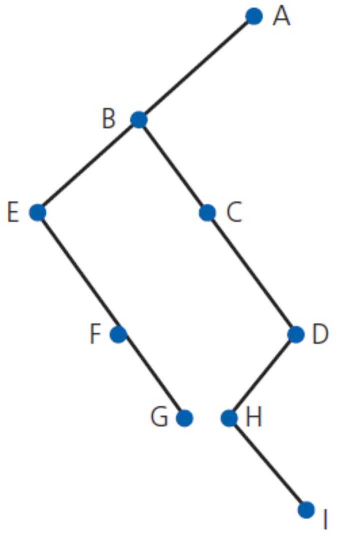
- A general tree or an n-ary tree with n=3



General Trees

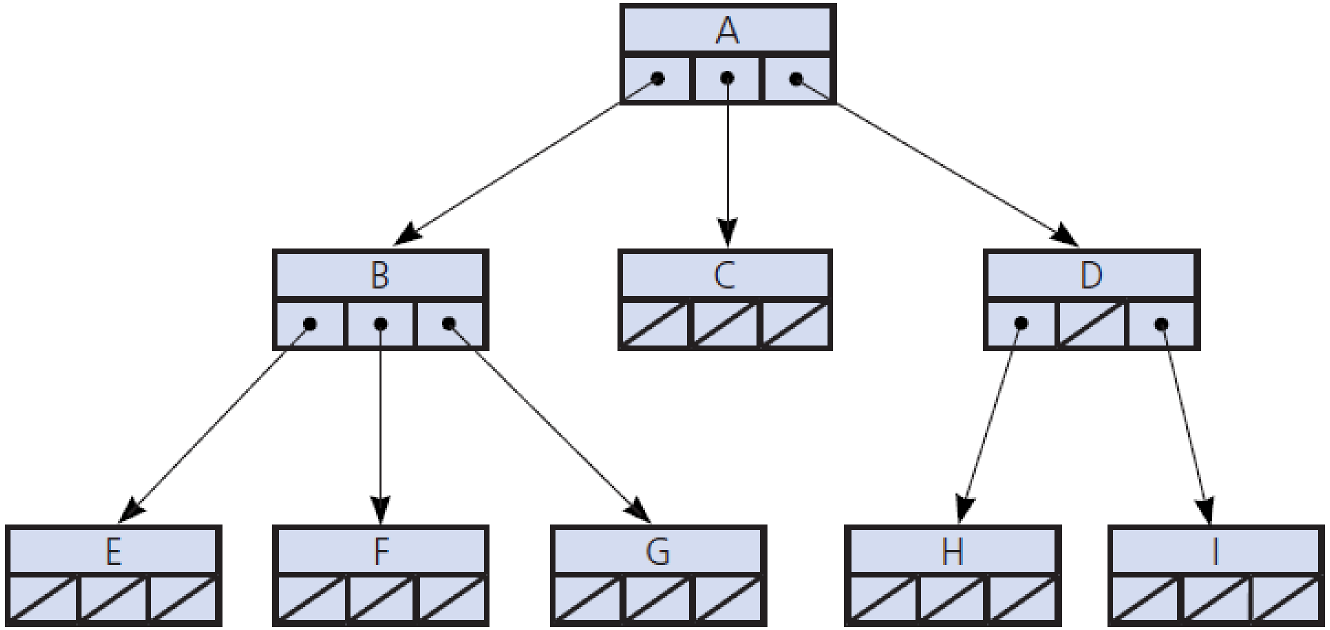
- An implementation of a general tree and its equivalent binary tree





General Trees

- An implementation of the n-ary tree



Thank you