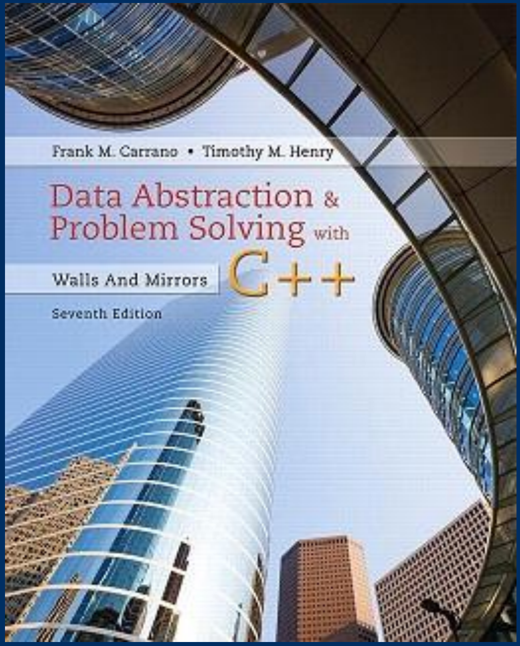
Chapter 16
Tree Implementations

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors' slides

Reminders

- Assignment 5 is available
  – Due April 11th at 2pm
- TA
  – Athanasia Katsila,
    Email: akatsila [at] nevada {dot} unr {dot} edu,
    Office Hours: Tuesday, 10:30 am - 12:30 pm at SEM 211
- Quiz 8 is available
  – Today between 4pm to 11:59pm

Nodes in a Binary Tree

- Representing tree nodes
  - Must contain both data and “pointers” to node’s children
  - Each node will be an object
- Array-based
  - Pointers will be array indices
- Link-based
  - Use C++ pointers

Array-Based Representation

- Class of array-based data members

```
TreeNode<ItemType>  tree[MAX_NODES]; // Array of tree nodes
int                 root;            // Index of root
int                 free;            // Index of free list
```

  - Variable root is index to tree's root node within the array tree
  - If tree is empty, root = -1

Array-Based Representation

- As tree changes (additions, removals) …
  - Nodes may not be in contiguous array elements
- Thus, need list of available nodes
  - Called a free list
- Node removed from tree
  - Placed in free list for later use

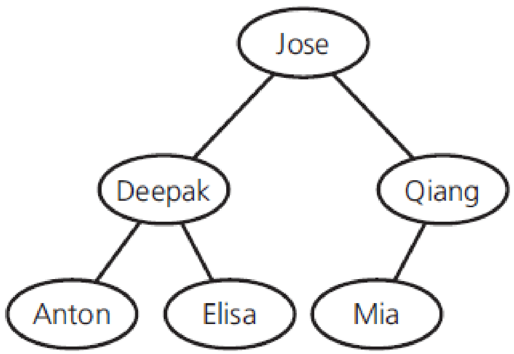
Array-Based Representation

```
template<class ItemType>
class TreeNode
{
private:
   ItemType  item;         // Data portion
   int       leftChild;    // Index to left child
   int       rightChild;   // Index to right child

public:
   TreeNode();
   TreeNode(const ItemType& nodeItem, int left, int right);

// Declarations of the methods setItem, getItem, setLeft, getLeft,
// setRight, and getRight are here.

   . . .

}; // end TreeNode
```

- The class TreeNode for an array-based implementation of the ADT binary tree

Array-Based Representation

(a)



(b)

The array tree

	item	leftChild	rightChild
0	Jose	1	2
1	Deepak	3	4
2	Qiang	5	–1
3	Anton	–1	–1
4	Elisa	–1	–1
5	Mia	–1	–1
6	?	–1	7
7	?	–1	8
8	?	–1	9
.	.	.	.
.	.	.	.
.	.	.	.

root: 0

free: 6

Free list (rows 6 onward)

(a) A binary tree of names;
(b) its implementation using the array tree

Link-Based Representation

```
/** A class of nodes for a link-based binary tree.
 @file BinaryNode.h */

#ifndef BINARY_NODE_
#define BINARY_NODE_
#include <memory>

template<class ItemType>
class BinaryNode
{
private:
   ItemType                             item;           // Data portion
   std::shared_ptr<BinaryNode<ItemType>> leftChildPtr;  // Pointer to left child
   std::shared_ptr<BinaryNode<ItemType>> rightChildPtr; // Pointer to right child

public:
   BinaryNode();
   BinaryNode(const ItemType& anItem);
   BinaryNode(const ItemType& anItem,
              std::shared_ptr<BinaryNode<ItemType>> leftPtr,
              std::shared_ptr<BinaryNode<ItemType>> rightPtr);
```

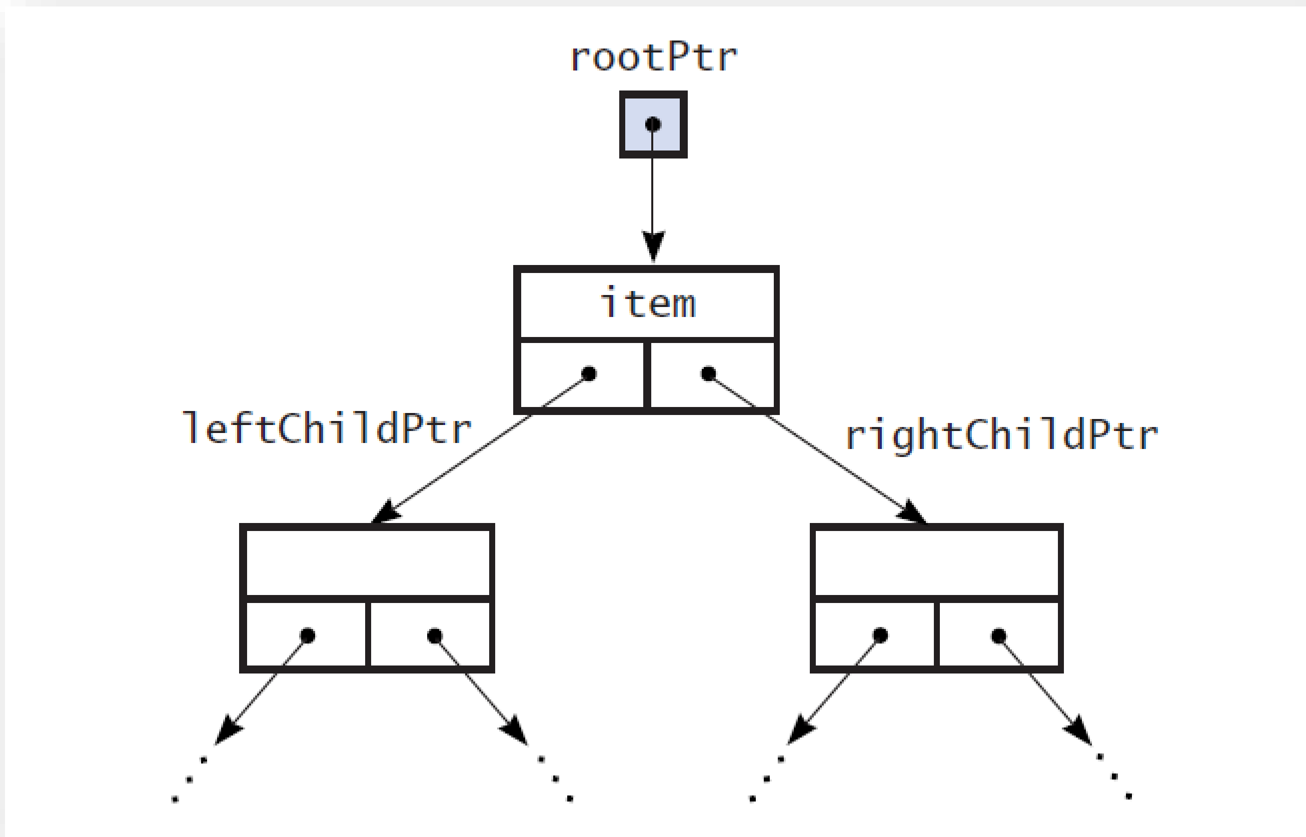
- The header file containing the class BinaryNode for a link-based implementation of the ADT binary tree

Link-Based Representation

```cpp
    void setItem(const ItemType& anItem);
    ItemType getItem() const;

    bool isLeaf() const;

     auto getLeftChildPtr() const;
     auto getRightChildPtr() const;

    void setLeftChildPtr(std::shared_ptr<BinaryNode<ItemType>> leftPtr);
    void setRightChildPtr(std::shared_ptr<BinaryNode<ItemType>> rightPtr);
}; // end BinaryNode

#include "BinaryNode.cpp"
#endif
```

- The header file containing the class BinaryNode for a link-based implementation of the ADT binary tree

Link-Based Representation



- A link-based implementation of a binary tree

The Header File

```cpp
/** ADT binary tree: Link-based implementation.
 @file BinaryNodeTree.h */

#ifndef BINARY_NODE_TREE_
#define BINARY_NODE_TREE_

#include "BinaryTreeInterface.h"
#include "BinaryNode.h"
#include "PrecondViolatedExcept.h"
#include "NotFoundException.h"
#include <memory>

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

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

The Header File

```
protected:
//------------------------------------------------------------
//      Protected Utility Methods Section:
//      Recursive helper methods for the public methods.
//------------------------------------------------------------
   int getHeightHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const;
   int getNumberOfNodesHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const;

   // Recursively adds a new node to the tree in a left/right fashion to keep tree balanced.
   auto balancedAdd(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                    std::shared_ptr<BinaryNode<ItemType>> newNodePtr);
```

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

The Header File

```
// Removes the target value from the tree.
virtual auto removeValue(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                         const ItemType target, bool& isSuccessful);

// Copies values up the tree to overwrite value in current node until
// a leaf is reached; the leaf is then removed, since its value is stored in the parent.
auto moveValuesUpTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);

// Recursively searches for target value.
virtual auto findNode(std::shared_ptr<BinaryNode<ItemType>> treePtr,
                      const ItemType& target, bool& isSuccessful) const;

// Copies the tree rooted at treePtr and returns a pointer to the root of the copy.
auto copyTree(const std::shared_ptr<BinaryNode<ItemType>> oldTreeRootPtr) const;

// Recursively deletes all nodes from the tree.
void destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr);
```

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

The Header File

```
   // Recursive traversal helper methods:
   void preorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;
   void inorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;
   void postorder(void visit(ItemType&), std::shared_ptr<BinaryNode<ItemType>> treePtr) const;

public:
//------------------------------------------------------------
//        Constructor and Destructor Section.
//------------------------------------------------------------
   BinaryNodeTree();
   BinaryNodeTree(const ItemType& rootItem);
   BinaryNodeTree(const ItemType& rootItem,
                  const std::shared_ptr<BinaryNodeTree<ItemType>> leftTreePtr,
                  const std::shared_ptr<BinaryNodeTree<ItemType>> rightTreePtr);
   BinaryNodeTree(const std::shared_ptr<BinaryNodeTree<ItemType>>& tree);
   virtual ~BinaryNodeTree();
```

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

The Header File

```
//------------------------------------------------------------
//       Public BinaryTreeInterface 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& newData); // Adds an item to the tree
   bool remove(const ItemType& data); // Removes specified item from the tree
   void clear();
   ItemType getEntry(const ItemType& anEntry) const throw(NotFoundException);
   bool contains(const ItemType& anEntry) const;
```

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

The Header File

```
//------------------------------------------------------------
//        Public Traversals Section.
//------------------------------------------------------------
   void preorderTraverse(void visit(ItemType&)) const;
   void inorderTraverse(void visit(ItemType&)) const;
   void postorderTraverse(void visit(ItemType&)) const;

//------------------------------------------------------------
//        Overloaded Operator Section.
//------------------------------------------------------------
   BinaryNodeTree& operator=(const BinaryNodeTree& rightHandSide);
}; // end BinaryNodeTree

#include "BinaryNodeTree.cpp"
#endif
```

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

The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::BinaryNodeTree() : rootPtr(nullptr)
{
} // end default constructor

template<class ItemType>
BinaryNodeTree<ItemType>::
BinaryNodeTree(const ItemType& rootItem)
      :rootPtr(std::make_shared<BinaryNode<ItemType>>(rootItem, nullptr, nullptr))
{
}  // end constructor
```

- Constructors

The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::
BinaryNodeTree(const ItemType& rootItem,
               const std::shared_ptr<BinaryNodeTree<ItemType>> leftTreePtr,
               const std::shared_ptr<BinaryNodeTree<ItemType>> rightTreePtr)
   :rootPtr(std::make_shared<BinaryNode<ItemType>>(rootItem,
                                                   copyTree(leftTreePtr->rootPtr),
                                                   copyTree(rightTreePtr->rootPtr))
{
}  // end constructor
```

- Constructors

The Implementation

```
template<class ItemType>
std::shared_ptr<BinaryNode<ItemType>> BinaryNodeTree<ItemType>::copyTree(
     const std::shared_ptr<BinaryNode<ItemType>> oldTreeRootPtr) const
{
   std::shared_ptr<BinaryNode<ItemType>> newTreePtr;

   // Copy tree nodes during a preorder traversal
   if (oldTreeRootPtr != nullptr)
   {
      // Copy node
      newTreePtr = std::make_shared<BinaryNode<ItemType>>(oldTreeRootPtr->getItem(),
                                                           nullptr, nullptr);
      newTreePtr->setLeftChildPtr(copyTree(oldTreeRootPtr->getLeftChildPtr()));
      newTreePtr->setRightChildPtr(copyTree(oldTreeRootPtr->getRightChildPtr()));
   }  // end if
   // Else tree is empty (newTreePtr is nullptr)

   return newTreePtr;
}  // end copyTree
```

- Protected method copyTree called by copy constructor

The Implementation

```
template<class ItemType>
BinaryNodeTree<ItemType>::
                     BinaryNodeTree(const BinaryNodeTree<ItemType>& treePtr)
{
   rootPtr = copyTree(treePtr.rootPtr);
}  // end copy constructor
```

- Copy constructor

The Implementation

```
template<class ItemType>
void BinaryNodeTree<ItemType>::
     destroyTree(std::shared_ptr<BinaryNode<ItemType>> subTreePtr)
{
   if (subTreePtr != nullptr)
   {
      destroyTree(subTreePtr->getLeftChildPtr());
      destroyTree(subTreePtr->getRightChildPtr());
      subTreePtr.reset(); // Decrement reference count to node
   }  // end if
}  // end destroyTree
```

- destroyTree used by destructor which simply calls this method

The Implementation

```
template<class ItemType>
int BinaryNodeTree<ItemType>::
    getHeightHelper(std::shared_ptr<BinaryNode<ItemType>> subTreePtr) const
{
   if (subTreePtr == nullptr)
      return 0;
   else
      return 1 + max(getHeightHelper(subTreePtr->getLeftChildPtr()),
                     getHeightHelper(subTreePtr->getRightChildPtr()));
}  // end getHeightHelper
```

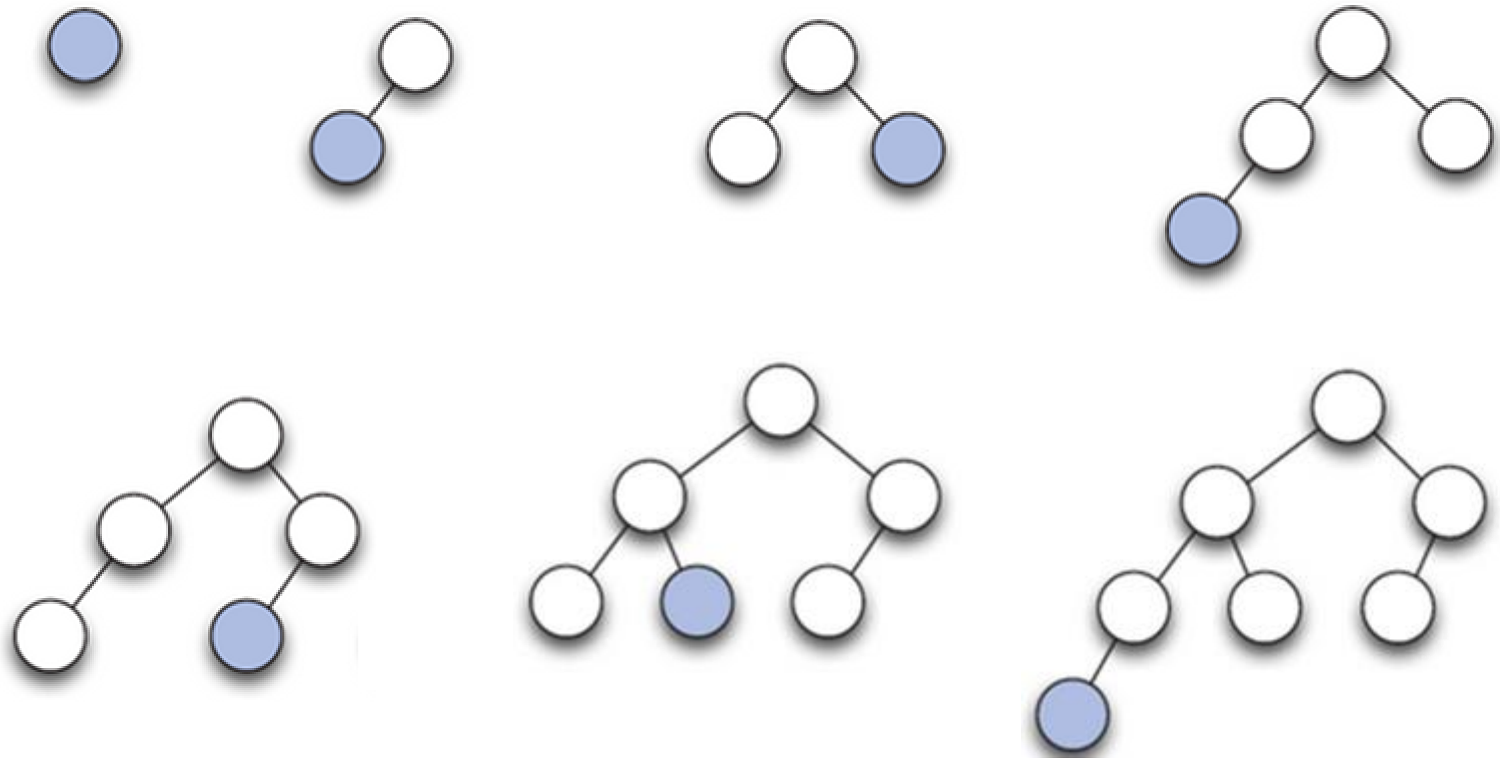
- Protected method getHeightHelper

The Implementation

```
template<class ItemType>
bool BinaryNodeTree<ItemType>::add(const ItemType& newData)
{
   auto newNodePtr = std::make_shared<BinaryNode<ItemType>>(newData);
   rootPtr = balancedAdd(rootPtr, newNodePtr);

   return true;
}  // end add
```

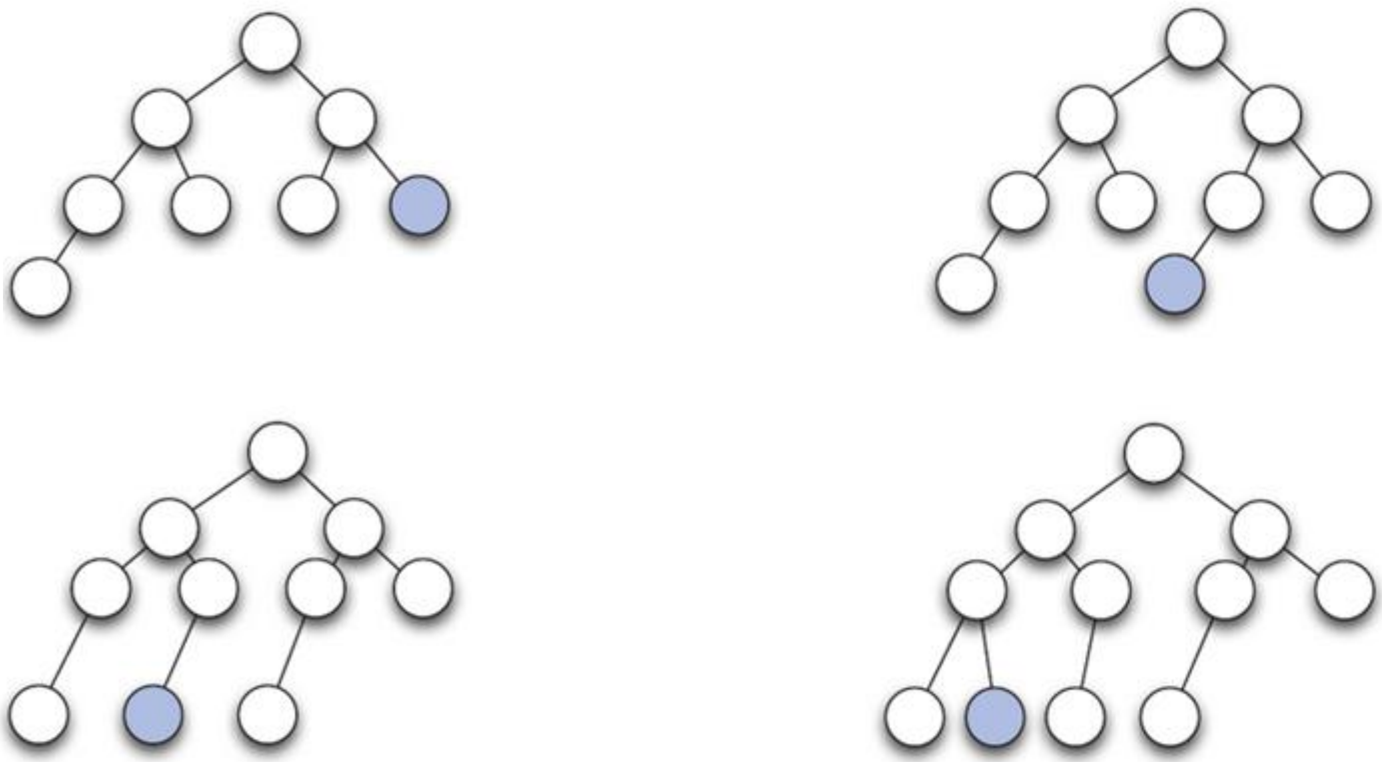
- Method add

The Implementation



- Adding nodes to an initially empty binary tree

The Implementation



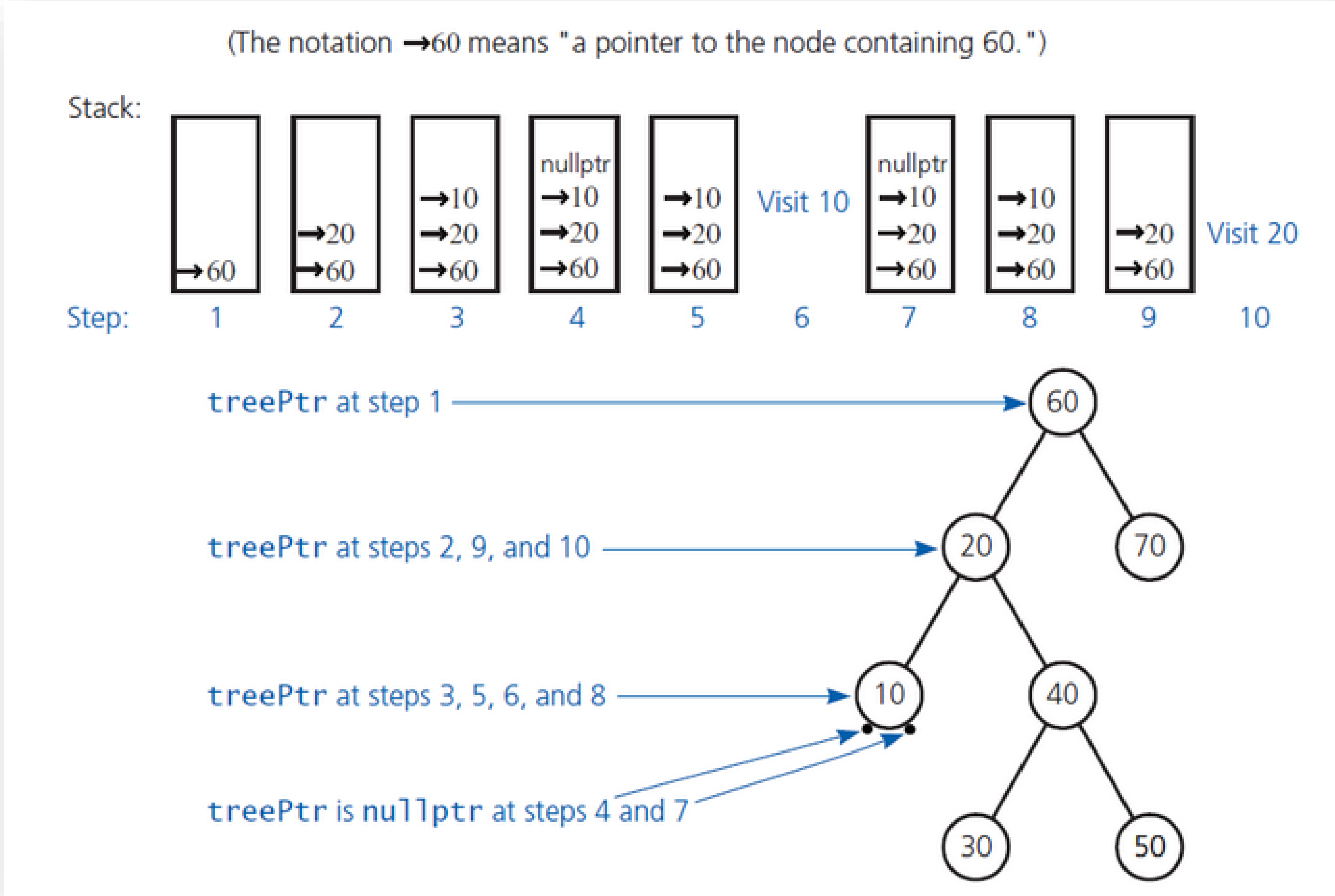
- Adding nodes to an initially empty binary tree

The Implementation

```
template<class ItemType>
void BinaryNodeTree<ItemType>::
     inorder(void visit(ItemType&),
             std::shared_ptr<BinaryNode<ItemType>> treePtr) const
{
   if (treePtr != nullptr)
   {
      inorder(visit, treePtr->getLeftChildPtr());
      ItemType theItem = treePtr->getItem();
      visit(theItem);
      inorder(visit, treePtr->getRightChildPtr());
   }  // end if
}  // end inorder
```

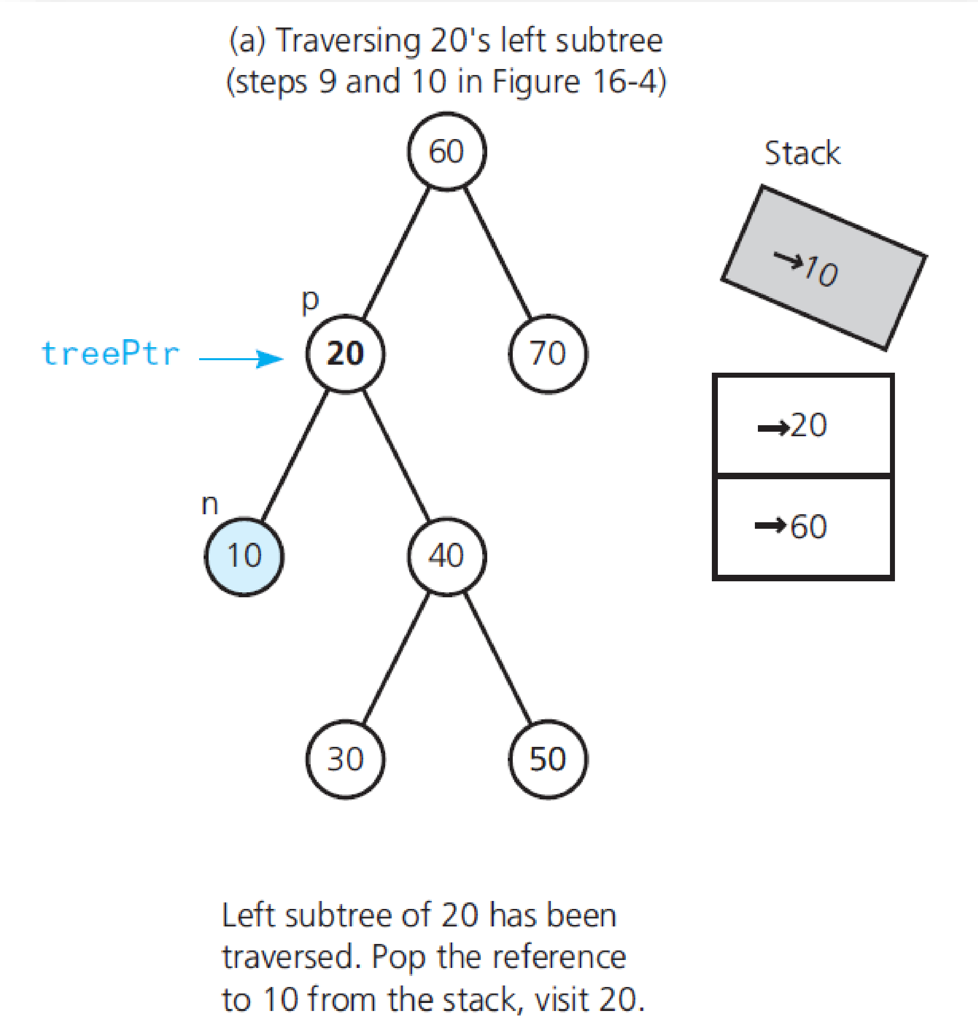
- Protected method that enables recursive traversals.

The Implementation



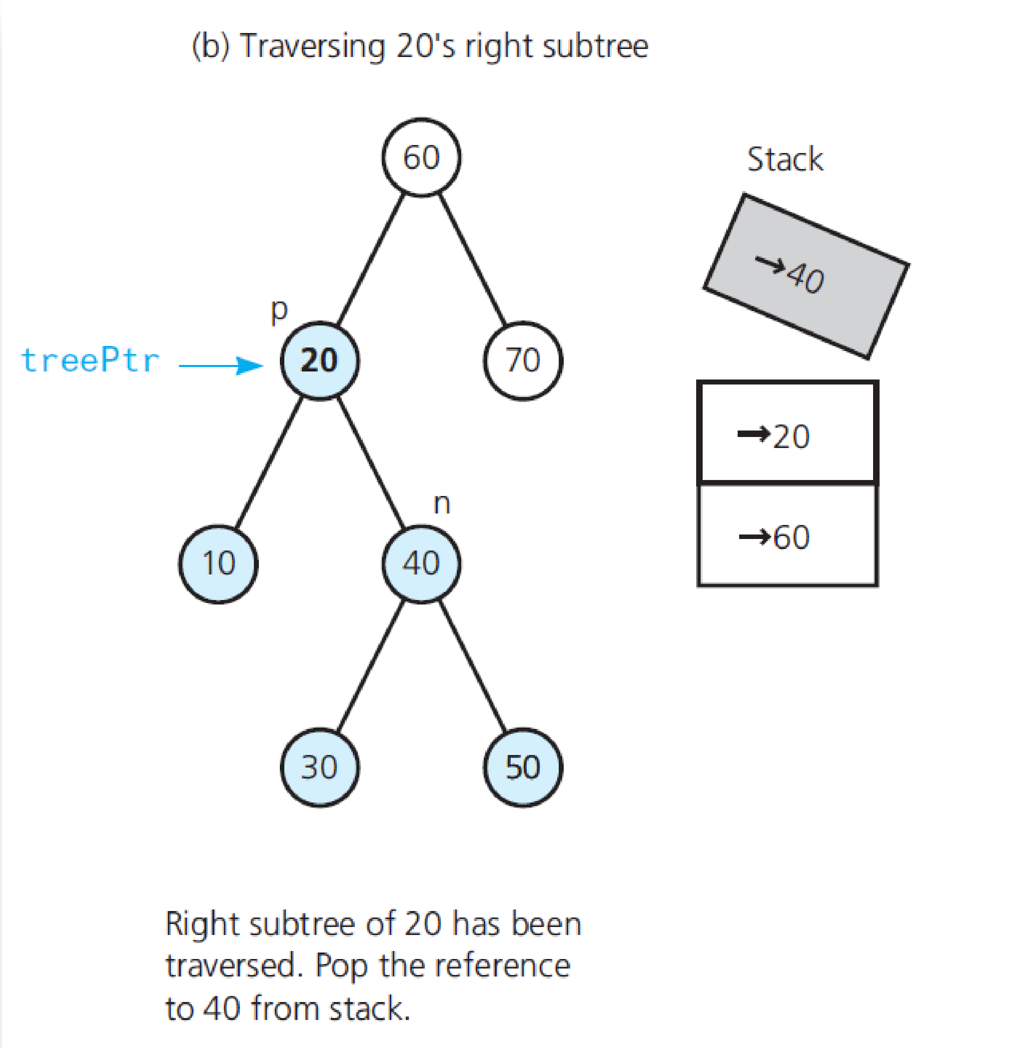
- Contents of the implicit stack as treePtr progresses through a given tree during a recursive inorder traversal

The Implementation



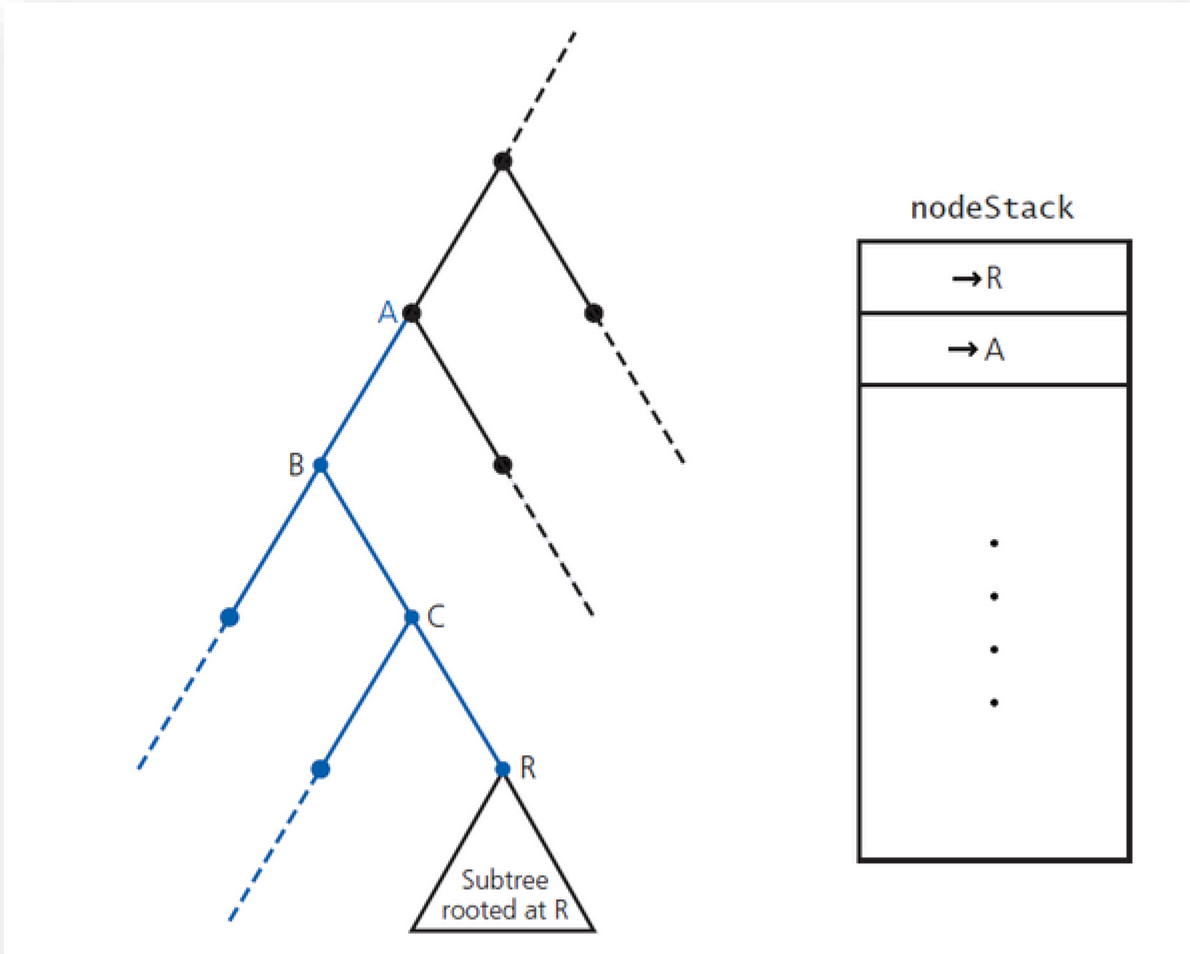
- Steps during an inorder traversal of the subtrees of 20

The Implementation



- Steps during an inorder traversal of the subtrees of 20

The Implementation



- Avoiding returns to nodes B and C

The Implementation

```
// Nonrecursively traverses a binary tree in inorder.
traverse(visit(item: ItemType): void): void
{
    // Initialize
    nodeStack = A new, empty stack
    curPtr = rootPtr // Start at root
    done = false

    while (!done)
    {
        if (curPtr != nullptr)
        {
            // Place pointer to node on stack before traversing the node's left subtree
            nodeStack.push(curPtr)

            // Traverse the left subtree
            curPtr = curPtr->getLeftChildPtr()
        }
        else  // Backtrack from the empty subtree and visit the node at the top of
```

- Nonrecursive inorder traversal

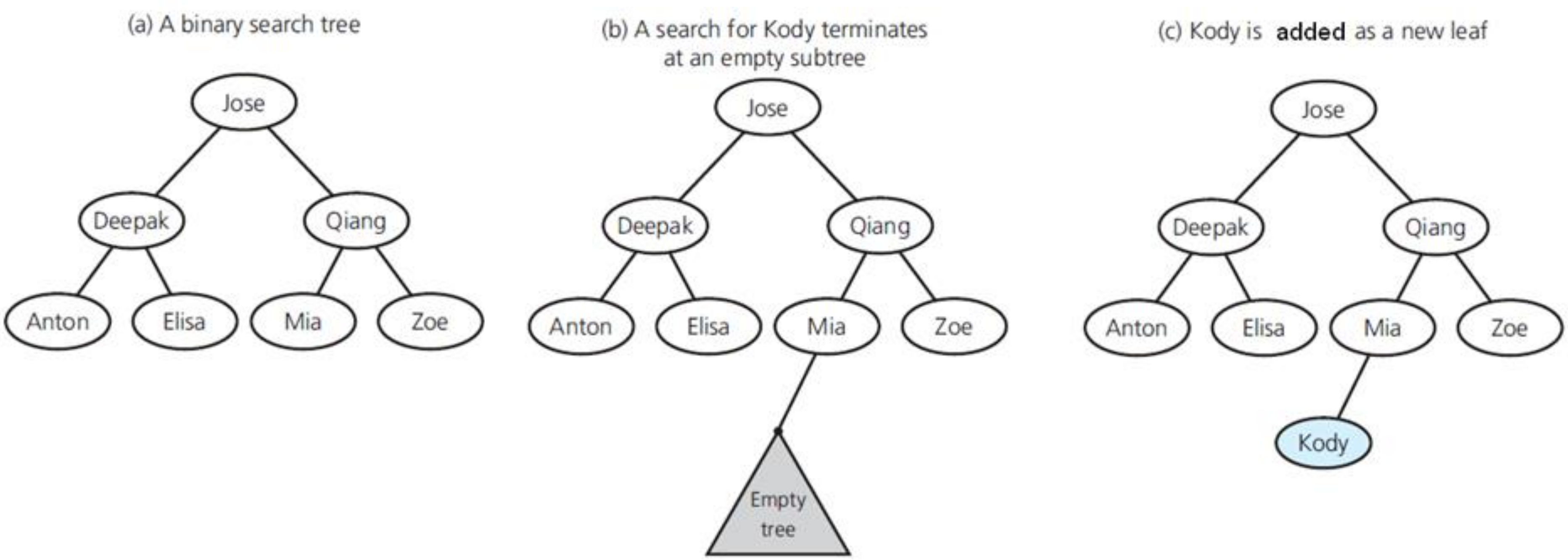
The Implementation

```
        }
        else    // Backtrack from the empty subtree and visit the node at the top of
                // the stack; however, if the stack is empty, you are done
        {
            done = nodeStack.isEmpty()
            if (!done)
            {
                nodeStack.peek(curPtr)
                visit(curPtr->getItem())
                nodeStack.pop()

                // Traverse the right subtree of the node just visited
                curPtr = curPtr ->getRightChildPtr()
            }
        }
    }
}
```

- Nonrecursive inorder traversal

- Uses same node objects as for binary-tree implementation
- Class BinaryNode from Listing16-2 will be used
- Recursive search algorithm from Section15.3.2 is basis for operations

- Adding Kody to a binary search tree



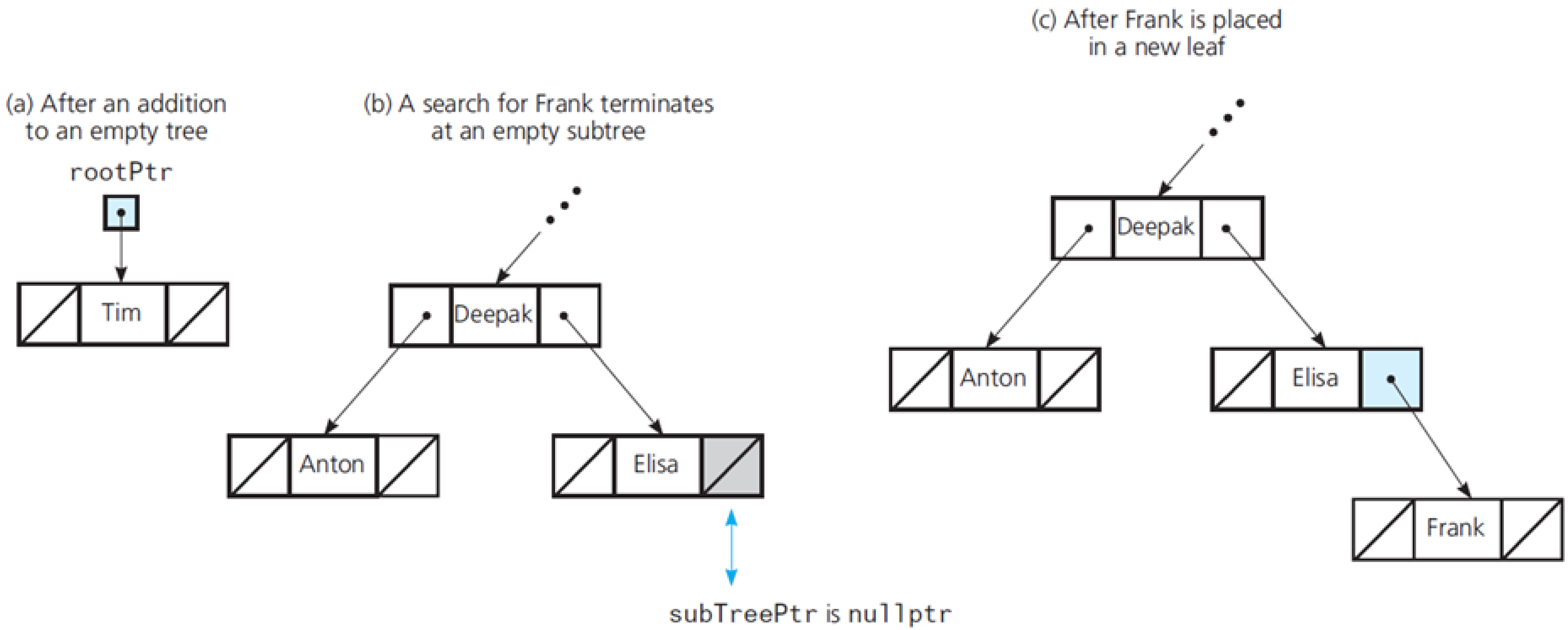
- Method add

```
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
```

- Refinement of addition algorithm

```
// Recursively places a given new node at its proper position in a binary search tree.
placeNode(subTreePtr: BinaryNodePointer,
          newNodePtr: BinaryNodePointer): BinaryNodePointer
{
   if (subTreePtr is 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
}
```

- Adding new data to a 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
}
```

- Cases for node *N* containing item to be removed

1. *N* is a leaf
   - Remove leaf containing target
   - Set pointer in parent to nullptr

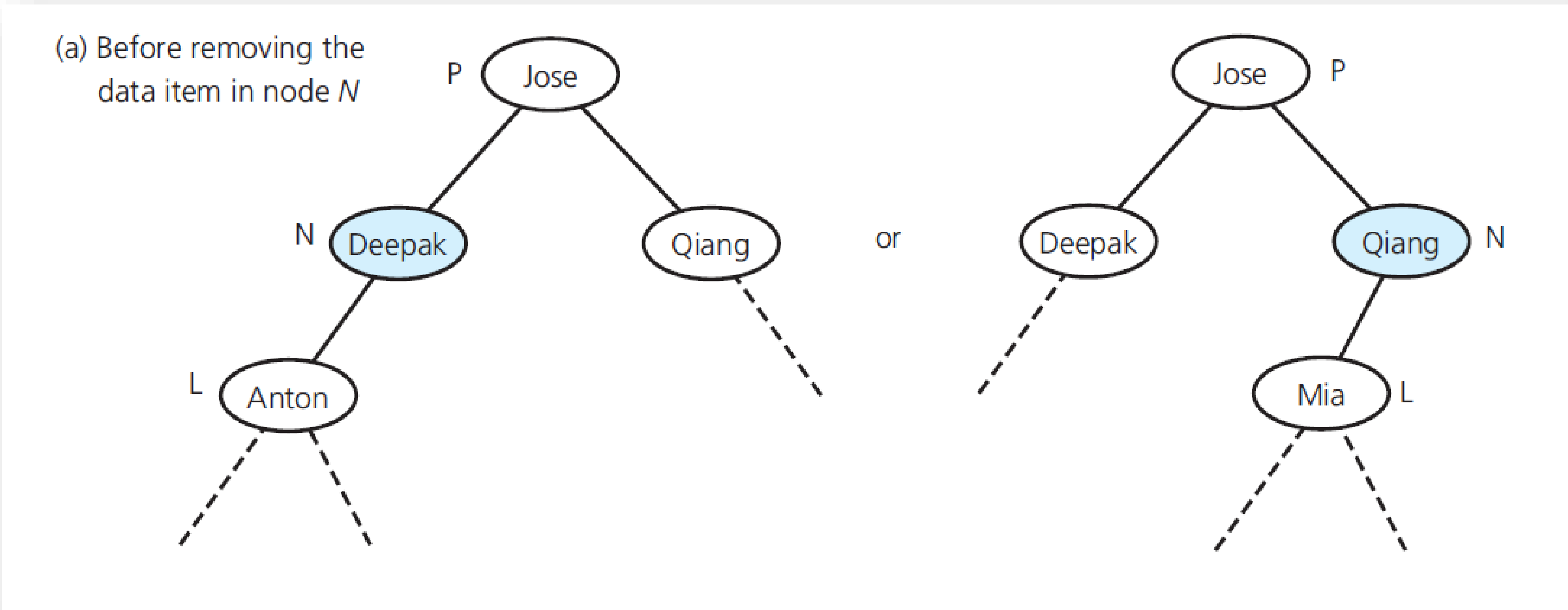
- Cases for node *N* containing item to be removed

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

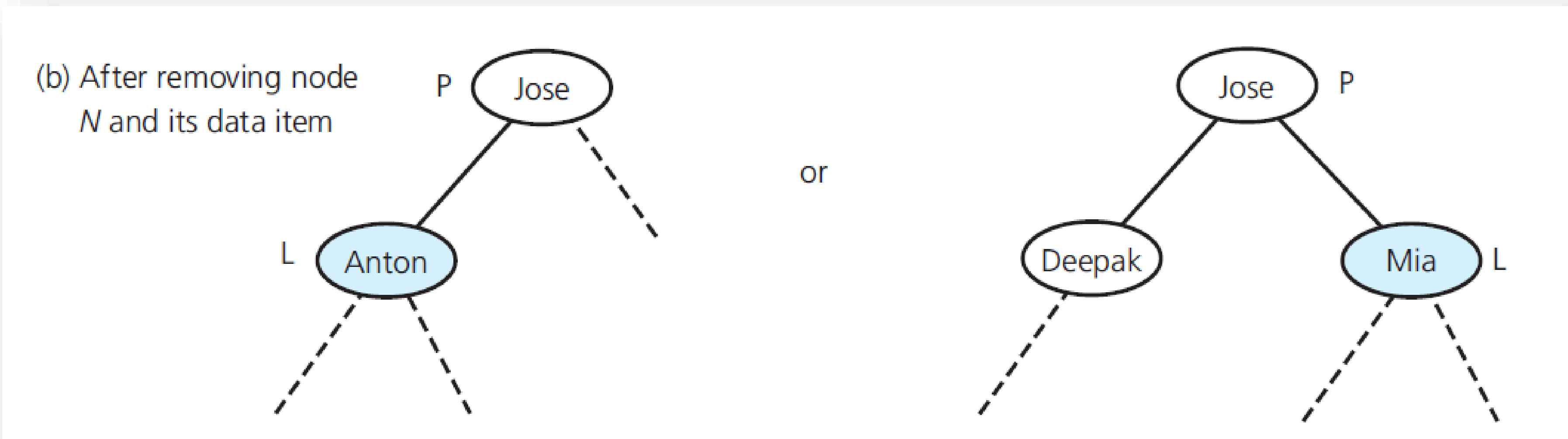
- Cases for node *N* containing item to be removed

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

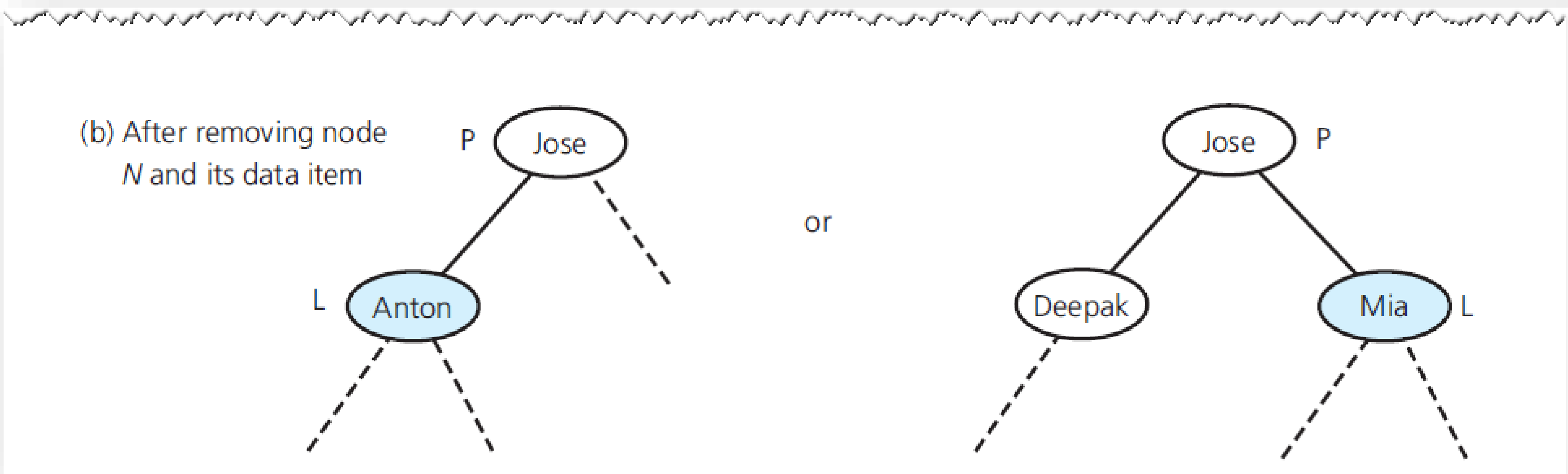
Link-Based Implementation of the ADT Binary Search Tree



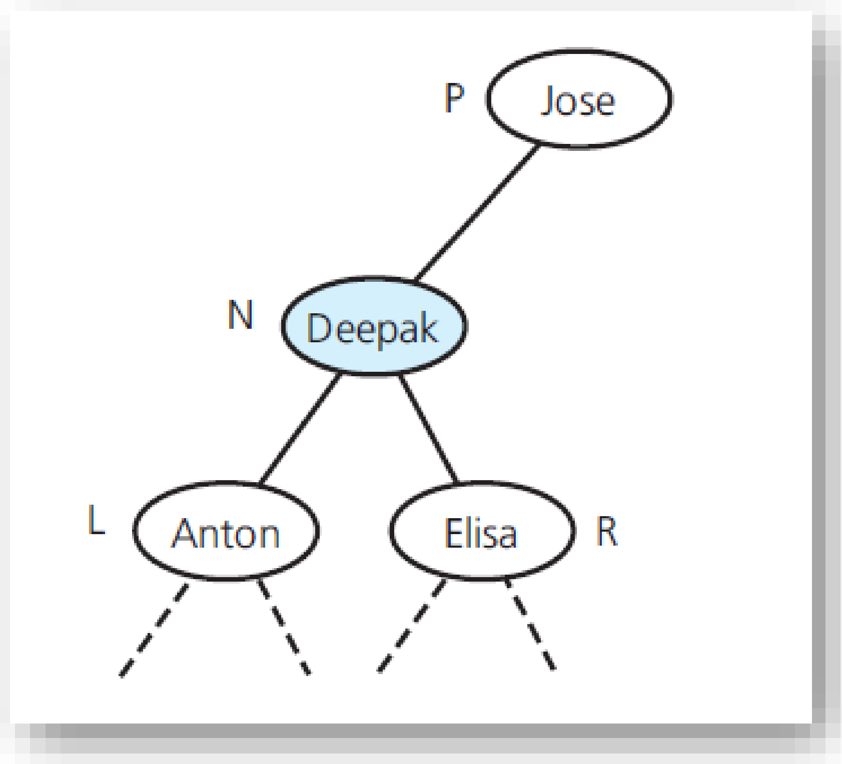
- Case 2 for removeValue: The data item to remove is in a node N that has only a left child and whose parent is node P



- 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

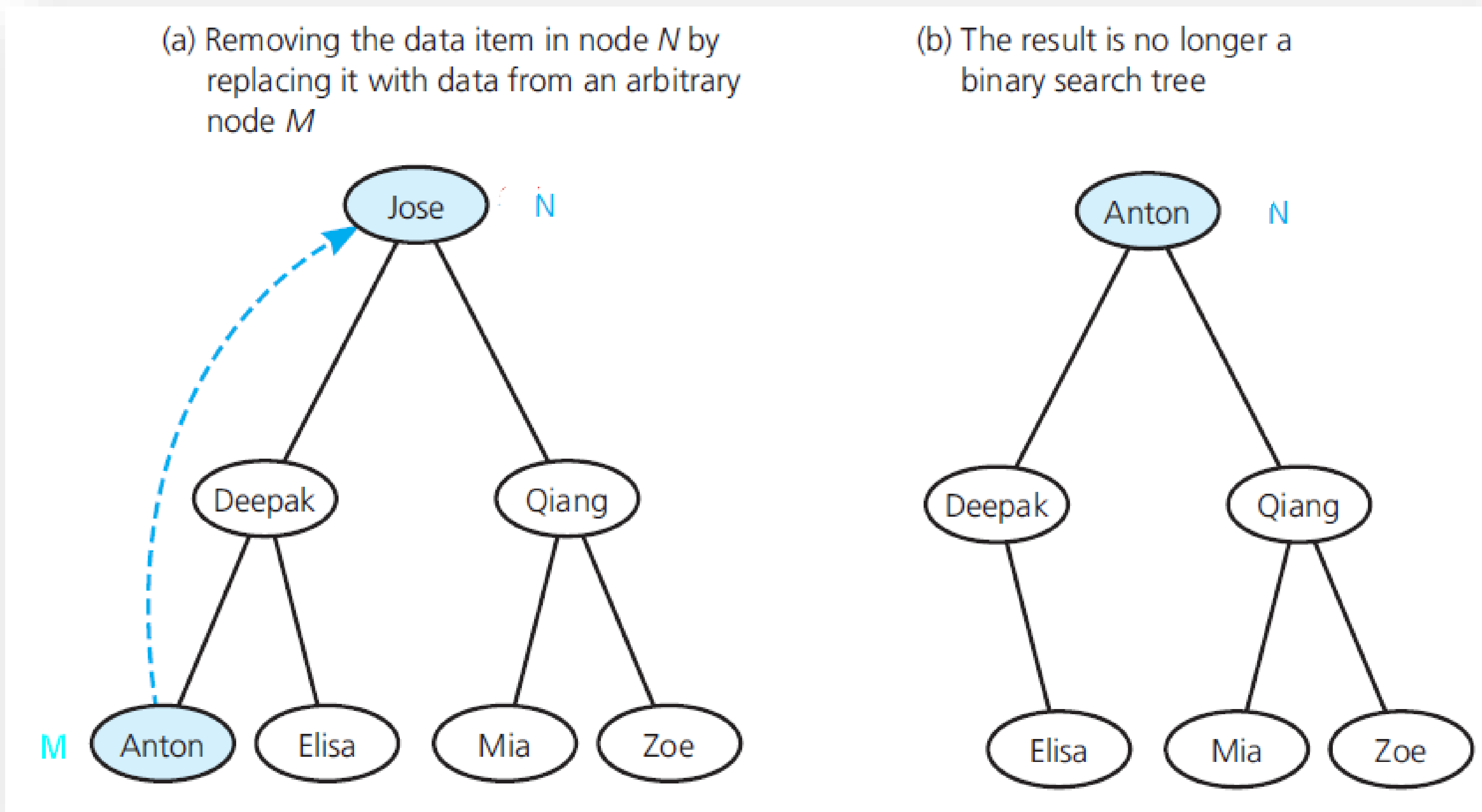


- Case 2 for removeValue: The data item to remove is in a node *N* that has only a left child and whose parent is node *P*

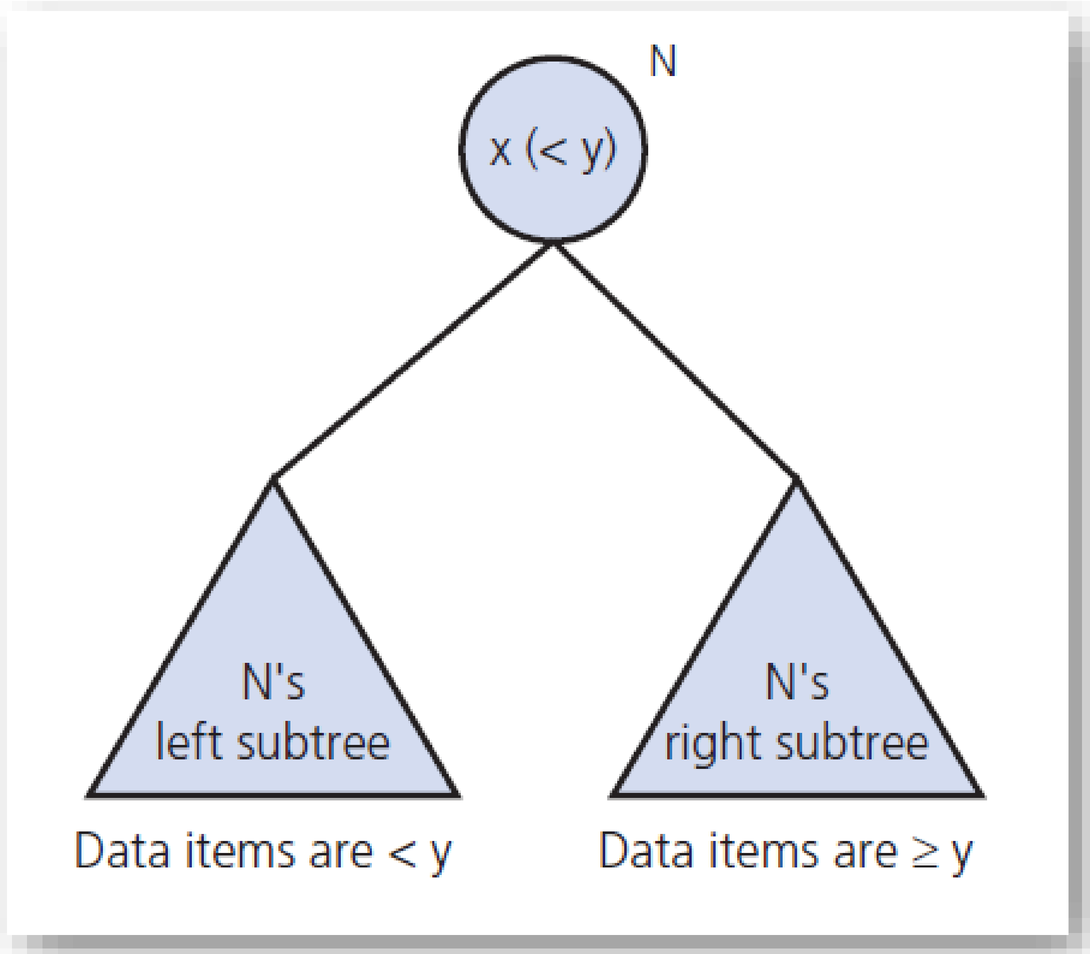


- Case 3: The data item to remove is in a node *N* that has two children

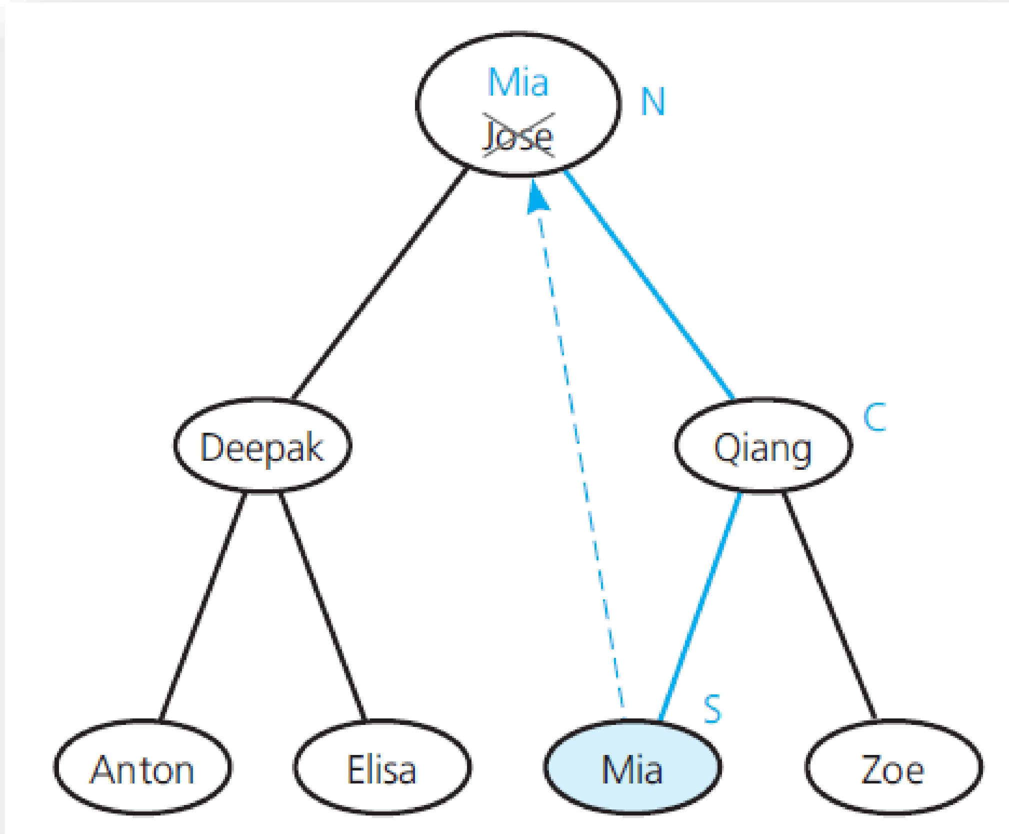
- Not any node will do



Link-Based Implementation of the ADT Binary Search Tree



Link-Based Implementation of the ADT Binary Search Tree



- Replacing the data item in node *N* with its inorder successor

Link-Based Implementation of the ADT Binary Search Tree

```
// Removes the given target from the binary search tree to which subTreePtr points.
// Returns 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
```

- Final draft of the removal algorithm

Link-Based Implementation of the ADT Binary Search Tree

```
        }
        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)
```

- Final draft of the removal algorithm

Link-Based Implementation of the ADT Binary Search Tree

```
}
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
}
```

- Final draft of the removal algorithm

Link-Based Implementation of the ADT Binary Search Tree

```
        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 node N
            return nodePtr
        }
    }

    // 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.
    removeLeftmostNode(nodePtr: BinaryNodePointer,
                       inorderSuccessor: ItemType&): BinaryNodePointer
    {
```

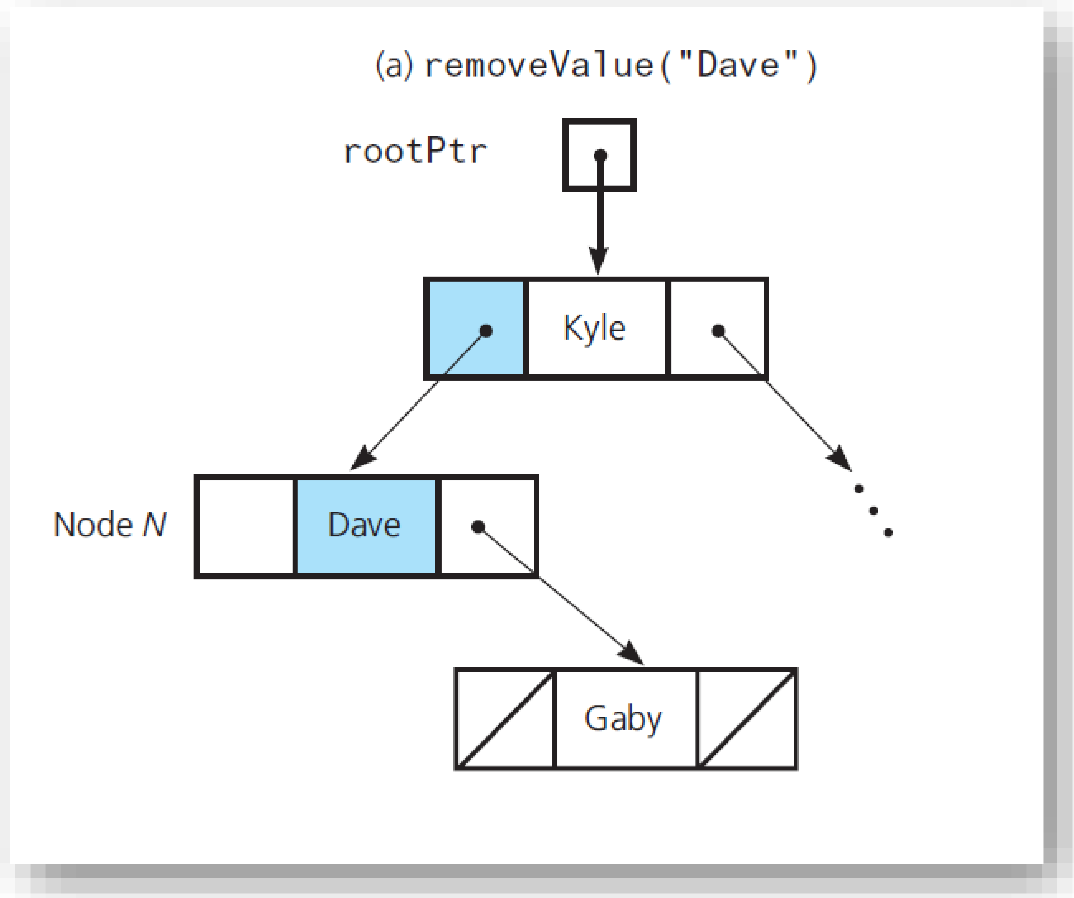
- Final draft of the removal algorithm

Link-Based Implementation of the ADT Binary Search Tree

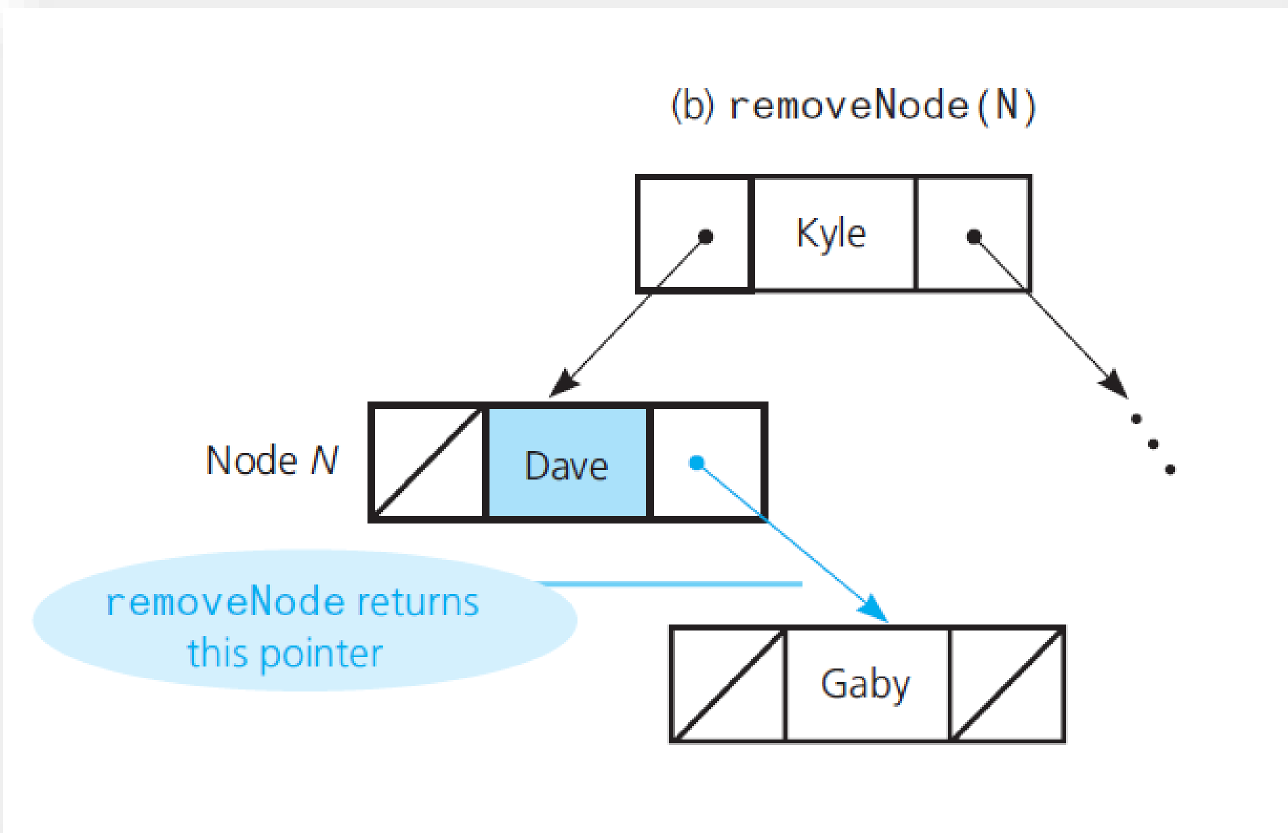
```
                      inorderSuccessor: ItemType&): BinaryNodePointer
{
   if (nodePtr->getLeftChildPtr() == nullptr)
   {
      // This is the node you want; it has no left child, but it might have a right subtree
      inorderSuccessor = nodePtr->getItem()
      return removeNode(nodePtr)
   }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

- Final draft of the removal algorithm

Link-Based Implementation of the ADT Binary Search Tree

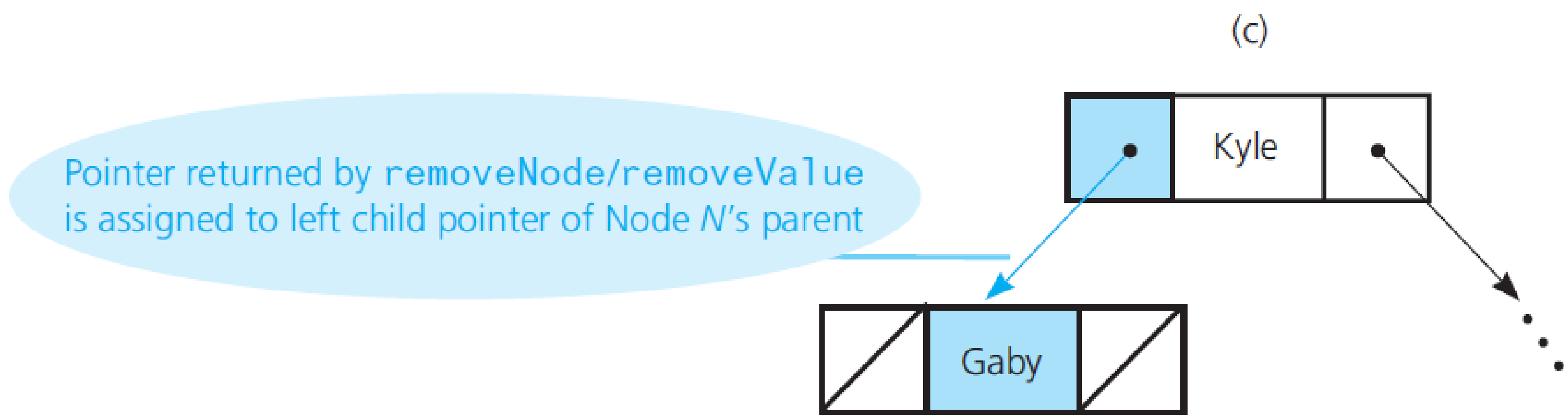


- Recursive removal of node N



- Recursive removal of node N

Link-Based Implementation of the ADT Binary Search Tree



- Recursive removal of node N

- Algorithm for 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)
 }
```

The Class BinarySearchTree

```
/** Link-based implementation of the ADT binary search tree.
 @file BinarySearchTree.h */

#ifndef BINARY_SEARCH_TREE_
#define BINARY_SEARCH_TREE_

#include "BinaryTreeInterface.h"
#include "BinaryNode.h"
#include "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;
```

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

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);
```

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

The Class BinarySearchTree

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

    // 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();
```

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

The Class BinarySearchTree

```
BinarySearchTree(const ItemType& rootItem);
BinarySearchTree(const BinarySearchTree<ItemType>& tree);
virtual ~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);
```

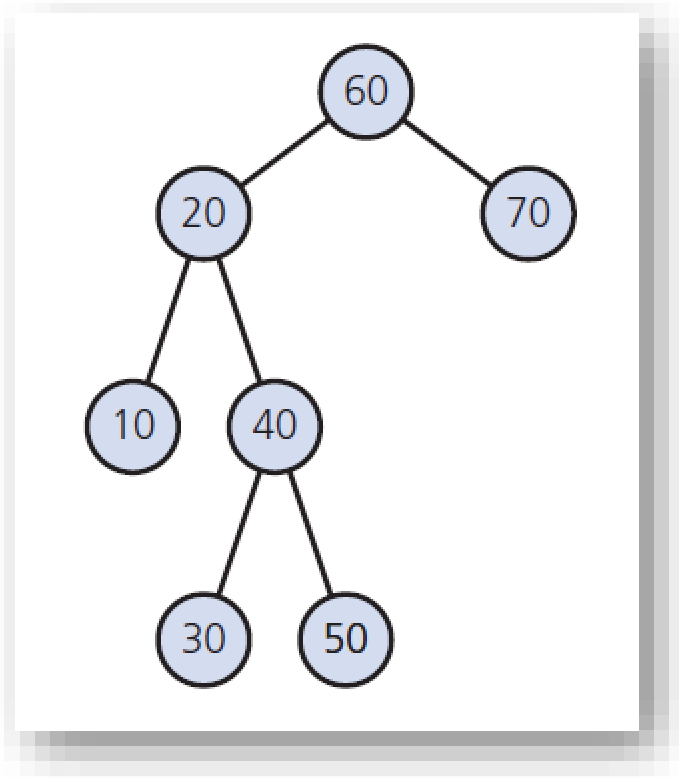
- A header file for the link-based implementation of the class BinarySearchTree

The Class BinarySearchTree

```
    bool contains(const ItemType& anEntry) const;

    //------------------------------------------------------------
    //  Public Traversals Section.
    //------------------------------------------------------------
    void preorderTraverse(void visit(ItemType&)) const;
    void inorderTraverse(void visit(ItemType&)) const;
    void postorderTraverse(void visit(ItemType&)) const;

    //------------------------------------------------------------
    //   Overloaded Operator Section.
    //------------------------------------------------------------
    BinarySearchTree<ItemType>&
                operator=(const BinarySearchTree<ItemType>& rightHandSide);
}; // end BinarySearchTree
#include "BinarySearchTree.cpp"
#endif
```

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

Saving a Binary Search Tree in a File



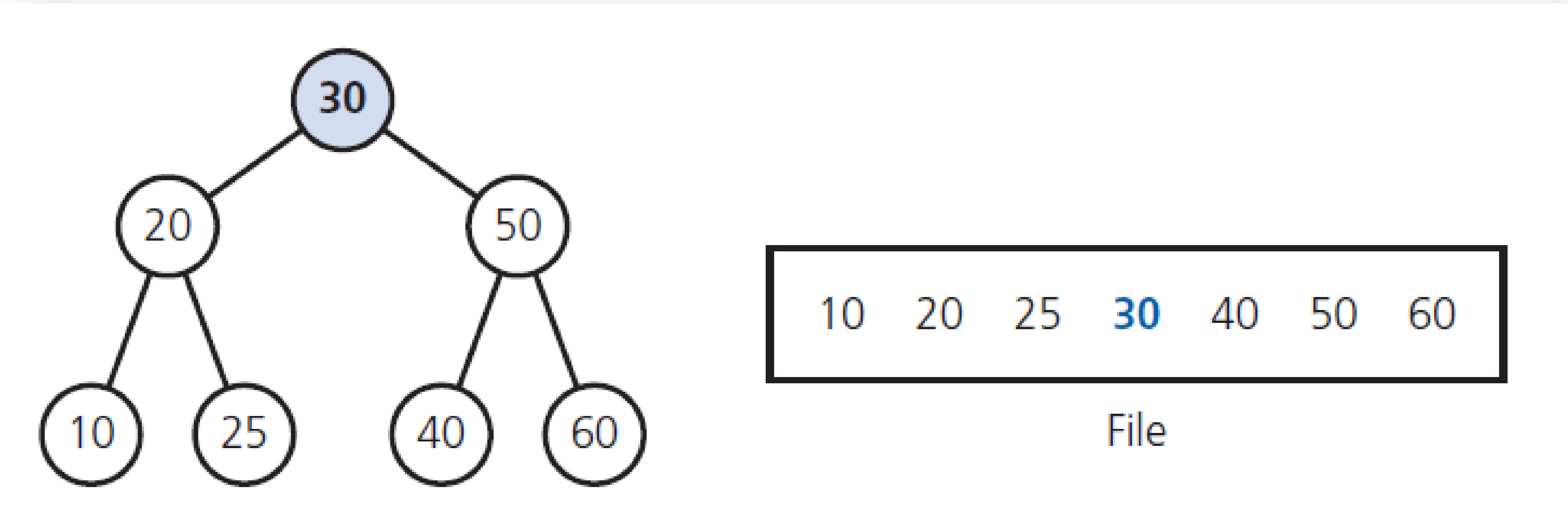
- An initially empty binary search tree after the addition of 60, 20, 10, 40, 30, 50, and 70

Saving a Binary Search Tree in a File

- 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

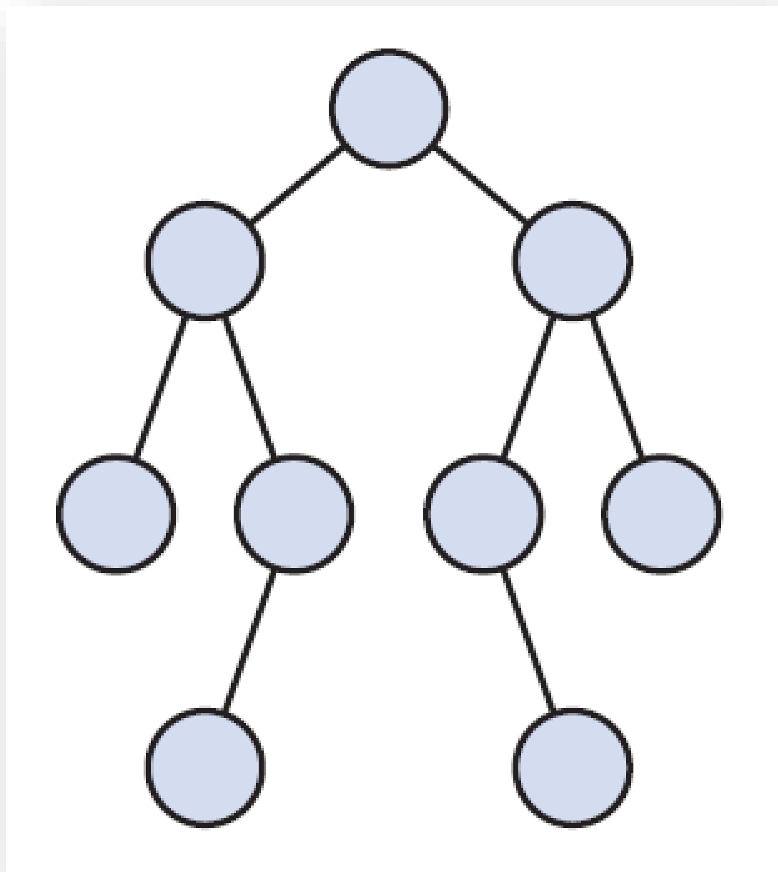
Saving a Binary Search Tree in a File

- A full tree saved in a file by using inorder traversal



Saving a Binary Search Tree in a File

- A tree of minimum height that is not complete



Saving a Binary Search Tree in a File

- 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)
```

Saving a Binary Search Tree in a File

- Building a minimum-height binary search tree

```
      // 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->setRightChildPtr(rightPtr)

      return treePtr
   }
   else
      return nullptr
}
```

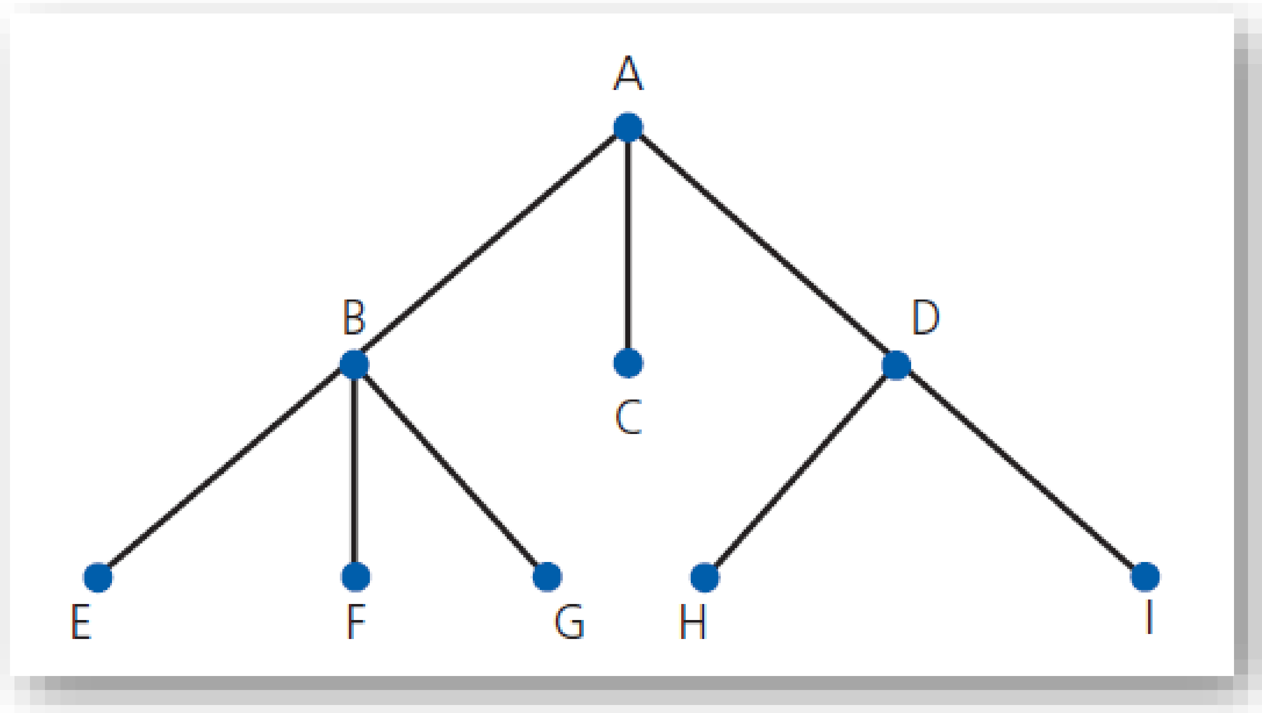
Tree Sort

- Tree sort uses a binary search tree.

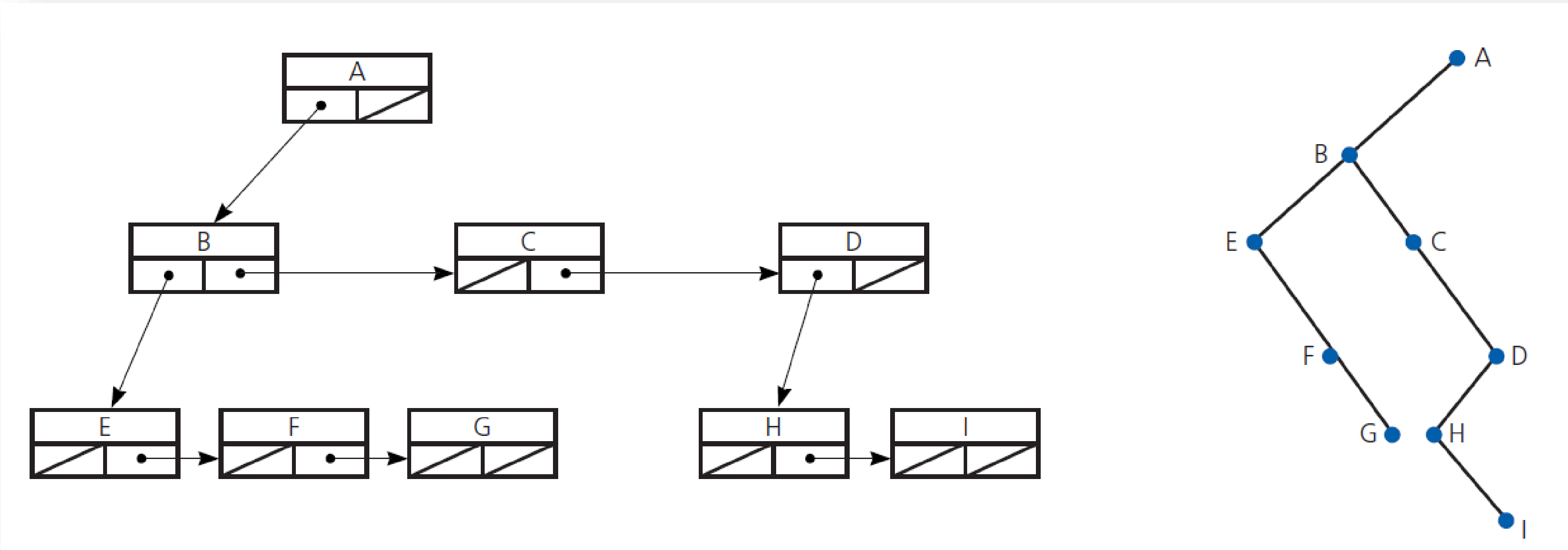
```
// Sorts the integers in an array into ascending order.
treeSort(anArray: array, n: integer)
{
    Add anArray's entries to a binary search tree bst
    Traverse bst in inorder. As you visit bst's nodes, copy their data items into successive
      locations of anArray
}
```

General Trees

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

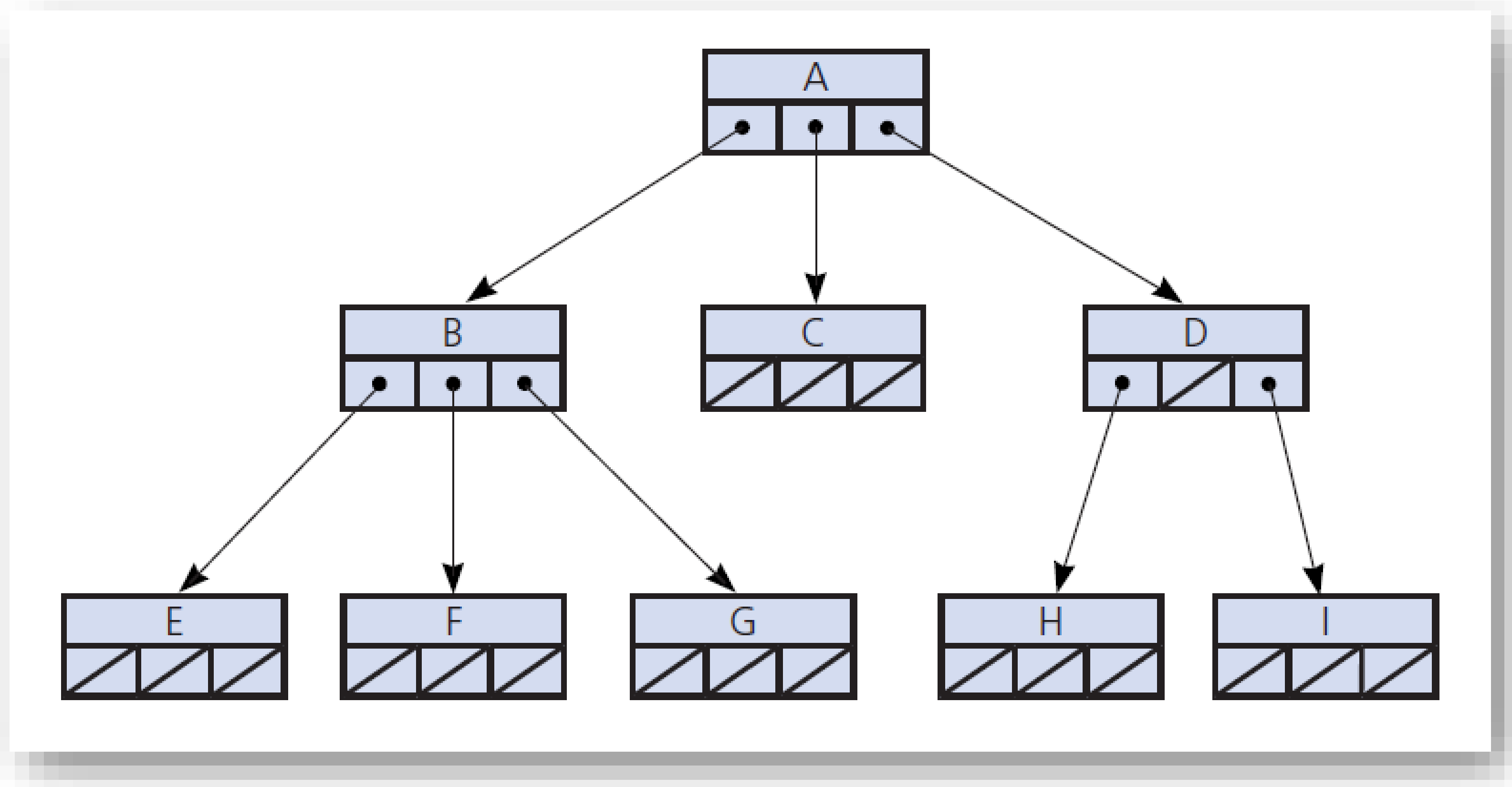


General Trees



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

General Trees



- An implementation of the *n*-ary tree