CS302 - Data Structures *using C++*

Topic: Tree Implementations

Kostas Alexis

AUTONOMOUS ROBOTS LAB

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

Array-based Representation

- As tree changes (additions, removals) …
  - Nodes may not be in contiguous array elements
- Thus, need a list of available nodes
  - Called a **free list**

Array-based Representation

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

Array-based Representation

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

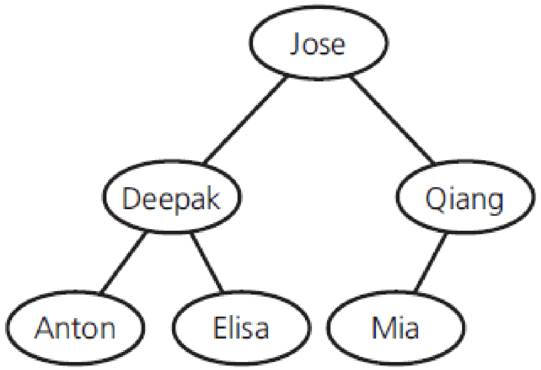
```
template<class ItemType>
class TreeNode
{
private:
     ItemType item;   // Data portion
     int leftChild;   // Index to left child
     int rightChild; // Index to right child
public:
     TreeNode();
     TreeNode(const ItemType& nodeItem, int left, int right);

// Declarations of the methods setIte, getItem, setLeft, getLeft,
// setRight, and getRight are here.
   ...
}; // end TreeNode
```

Array-based Representation

A binary tree of names

(a)



Its implementation using the array tree

(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

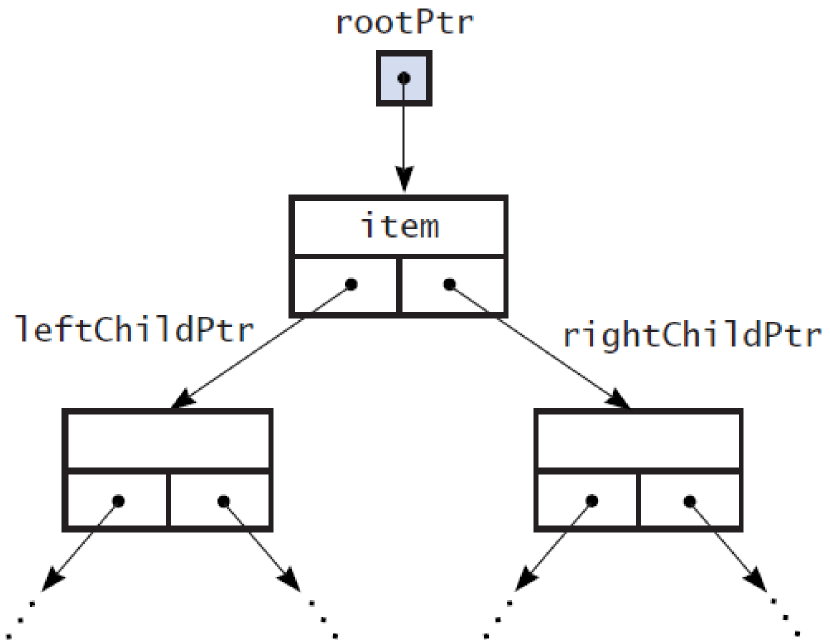
free

6

Free list

Link-based Representation

- A link-based implementation of a binary tree



Link-based Representation

- Header file containing the class BinaryNode for a link-based implementation of ADT binary Tree

```
#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);
```

Link-based Representation

- Header file containing the class **BinaryNode** for a link-based implementation of ADT binary Tree

```
    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

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

```
#ifndef BINARY_NODE_TREE_
#define BINARY_NODE_TREE_

#include "BinaryTreeInterface.h"
#include "BinaryNode.h"
#include "PrecondViolatedExcept.h"
#include "NotFoundException.h"

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

The Header File

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

```
// 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);
```

The Header File

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

```
    // 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();
```

The Header File

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

```
// PUBLIC BINARY_TREE_INTERFACE METHODS SECTION
bool isEmpty() const;
int getHeight() const;
int getNumberOfNodes() const;
ItemType getRootData() const throw(PrecondViolatedExcept);
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;
```

The Header File

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

```
    // PUBLIC TRAVERSAL 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
```

The Header File

- Invoke example

```
BinaryNodeTree<std::string> tree1;
auto tree2Ptr = std::make_shared<BinaryNodeTree<std::string>>(“A”);
auto tree3Ptr = std::make_shared<BinaryNodeTree<std::string>>(“B”);
auto tree4Ptr = std::make_shared<BinaryNodeTree<std::string>>(“C”);
```

The Implementation

- **Constructors**

```
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
```

The Implementation

- **Constructors**

```
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
```

The Implementation

- **Constructors**

```
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
```

Requires implicit use of traversal!

The Implementation

- **Constructors**

```
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
```

Requires implicit use of traversal!
Role of the Protected method copyTree

The Implementation

- Protected method **copyTree** called by copy constructor

```
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
```

The Implementation

- Protected method **copyTree** called by copy constructor
  - Must be linked together by using new pointers. You cannot simply copy the pointers in the nodes of the original tree. Deep Copy

```cpp
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
```

The Implementation

- **Copy constructor**

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

The Implementation

- **Copy constructor**

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

Deep Copy

The Implementation

- **destroyTree** used by destructor which simply calls this method

```
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
```

The Implementation

- **destroyTree** used by destructor which simply calls this method

```
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
```

postorder.
Why?

The Implementation

- Protected method **getHeightHelper** called by method **getHeight**

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

The Implementation

- Protected method **getHeightHelper** called by method **getHeight**
  - Height of a subtree rooted at particular node is 1 – for the node itself – plus the height of node's tallest tree

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

The Implementation

- Protected method **getHeightHelper** called by method **getHeight**
    - Height of a subtree rooted at particular node is 1 – for the node itself – plus the height of node's tallest tree

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

Recursive

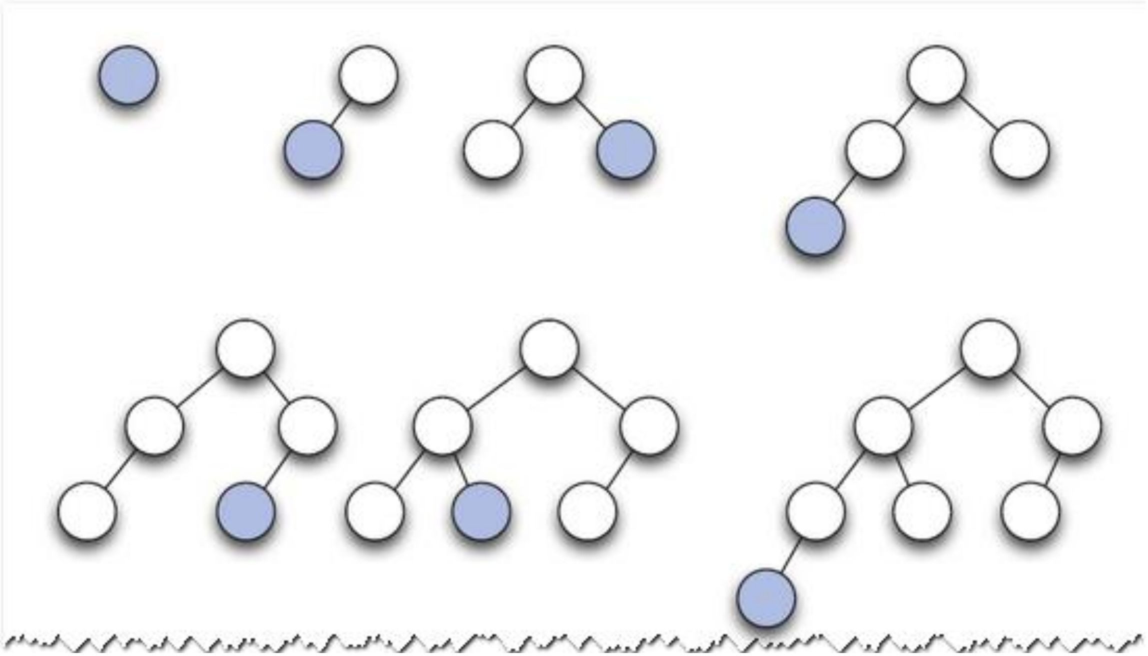
The Implementation

- Method **add**

```
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
```

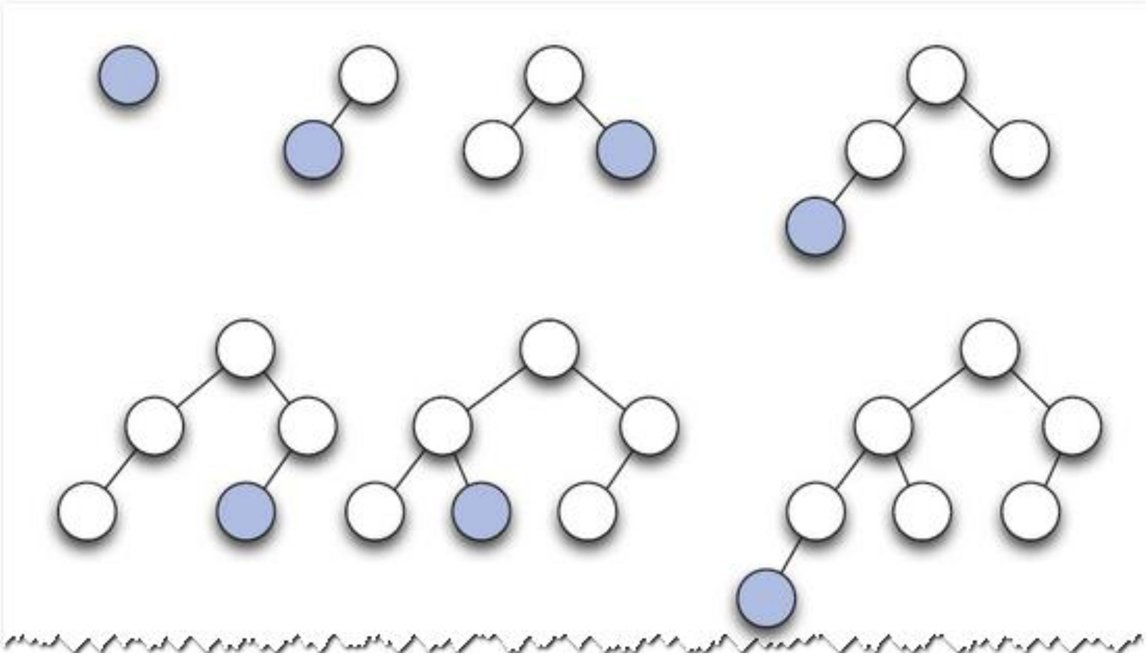
The Implementation

- Adding nodes to an initially empty binary tree



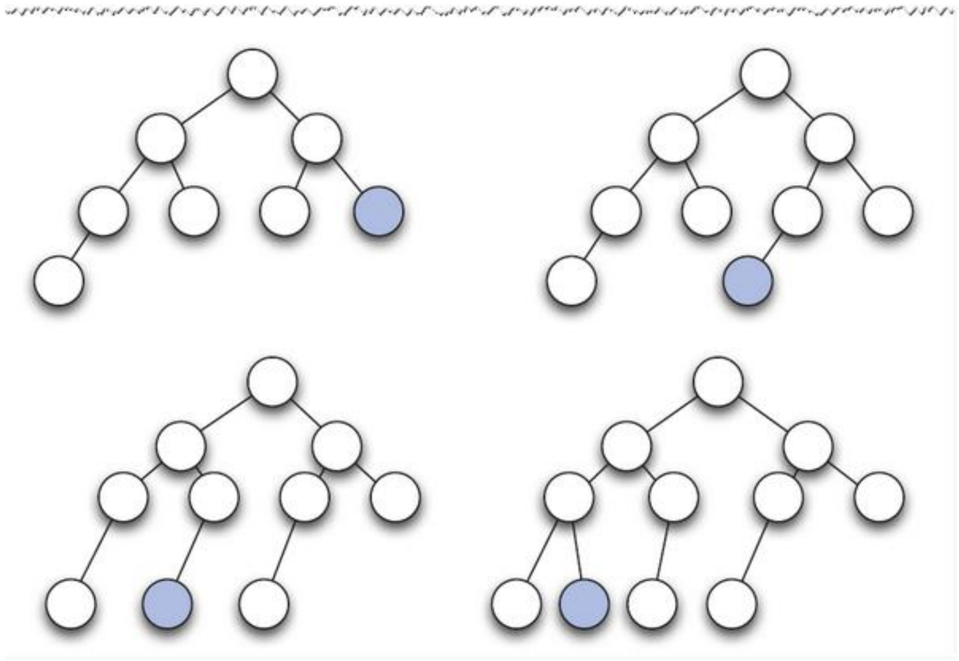
The Implementation

- Adding nodes to an initially empty binary tree: at a minimum ensure its balanced



The Implementation

- Adding nodes to an initially empty binary tree (cont)



The Implementation

- Method **balancedAdd**

```
template<class ItemType>
auto BinaryNodeTree<ItemType>::balancedAdd(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                                       std::shared_ptr<BinaryNode<ItemType>> newNodePtr)
{
     if (subTreePtr == nullptr)
          return newNodePtr;
     else
     {
          auto leftPtr = subTreePtr->getLeftChildPtr();
          auto rightPtr = subTreePtr->getRightChildPtr();

          if (getHeighetHelper(leftPtr) > getHeightHelper(rightPtr))
          {
                    rightPtr = balancedAdd(rightPtr, newNodePtr);
                    subTreePtr->setRightChildPtr(rightPtr);
          else
          {
                    leftPtr = balancedAdd(leftPtr, newNodePtr);
                    subTreePtr->setLeftChild(leftPtr);
          } // end if
          return subTreePtr;
     } // end if
} // end balancedAdd
```

The Implementation

- Method **balancedAdd**

```
template<class ItemType>
auto BinaryNodeTree<ItemType>::balancedAdd(std::shared_ptr<BinaryNode<ItemType>> subTreePtr,
                                           std::shared_ptr<BinaryNode<ItemType>> newNodePtr)
{
     if (subTreePtr == nullptr)
          return newNodePtr;
     else
     {
          auto leftPtr = subTreePtr->getLeftChildPtr();
          auto rightPtr = subTreePtr->getRightChildPtr();

          if (getHeighetHelper(leftPtr) > getHeightHelper(rightPtr))
          {
                    rightPtr = balancedAdd(rightPtr, newNodePtr);
                    subTreePtr->setRightChildPtr(rightPtr);
          else
          {
                    leftPtr = balancedAdd(leftPtr, newNodePtr);
                    subTreePtr->setLeftChild(leftPtr);
          } // end if
          return subTreePtr;
     } // end if
} // end balancedAdd
```

The recursive call to balancedAdd adds the new node and returns a pointer to the revised subtree. However, we need to link this subtree to the rest of the tree. This is done through setRightChildPtr.

The Implementation

- Protected method that enables **recursive traversals**
  - Public traversal methods will call these protected methods

```
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
```

The Implementation

- Protected method that enables **recursive traversals**
  - Public traversal methods will call these protected methods

```cpp
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
```

Client not only can access but also modify it

The Implementation

- The definition of the public method inorderTraverse only contains the cal

```
inorder(visit, rootPtr);
```

The Implementation

- Examine nonrecursive traversal – better understand relation between stacks and recursion

The Implementation

- Examine nonrecursive traversal – better understand relation between stacks and recursion
- Conceptual difficulty for nonrecursive traversal: where to go next after a particular node has been visited?

The Implementation

- Examine nonrecursive traversal – better understand relation between stacks and recursion
- Conceptual difficulty for nonrecursive traversal: where to go next after a particular node has been visited?
  - Consider how the recursive method works

```
if(treePtr!=nullptr)
{
    inorder(visit, treePtr->getLeftChildPtr()); // Point 1
    ItemType theItem = treePtr->getItem();
    visit(theItem);
    inorder(visit, treePtr->getRightChildPtr()); // Point 2
} // end if
```

The Implementation

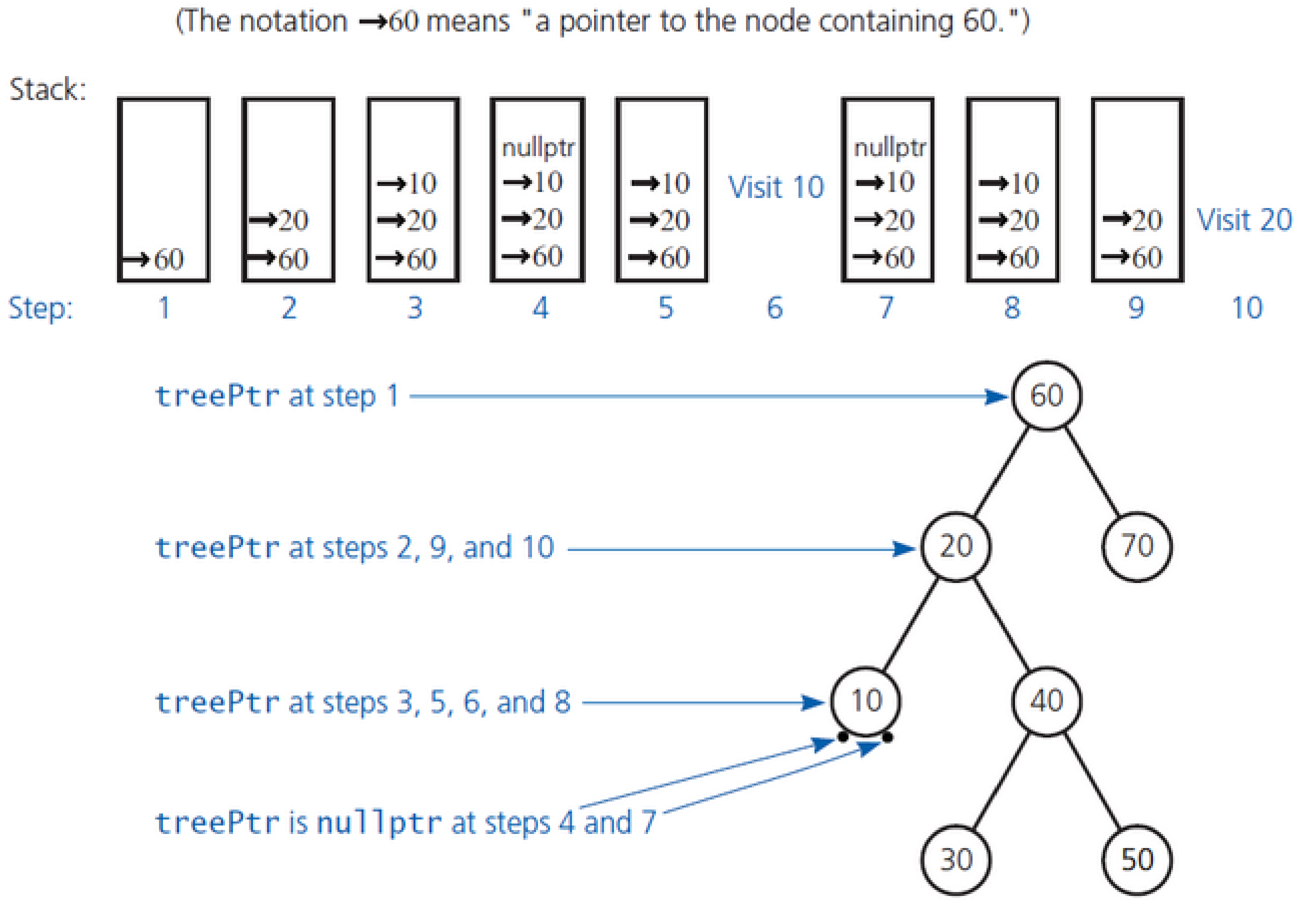
- Examine nonrecursive traversal – better understand relation between stacks and recursion
- Conceptual difficulty for nonrecursive traversal: where to go next after a particular node has been visited?
  - Consider how the recursive method works

```
if(treePtr!=nullptr)
{
    inorder(visit, treePtr->getLeftChildPtr()); // Point 1
    ItemType theItem = treePtr->getItem();
    visit(theItem);
    inorder(visit, treePtr->getRightChildPtr()); // Point 2
} // end if
```

  - During execution, the value of the pointer treePtr marks the current position of the tree.
  - Each time inorder makes a recursive call, the traversal moves to another node.
  - In terms of the stack that is implicit to recursive methods, a call to inorder pushes the new value of treePtr – that is, a pointer to the new current node – onto the stack.
  - At any given time, the stack contains pointers to the nodes along the path from the tree's root to the current node n, with the pointer to the root at the bottom.

The Implementation

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



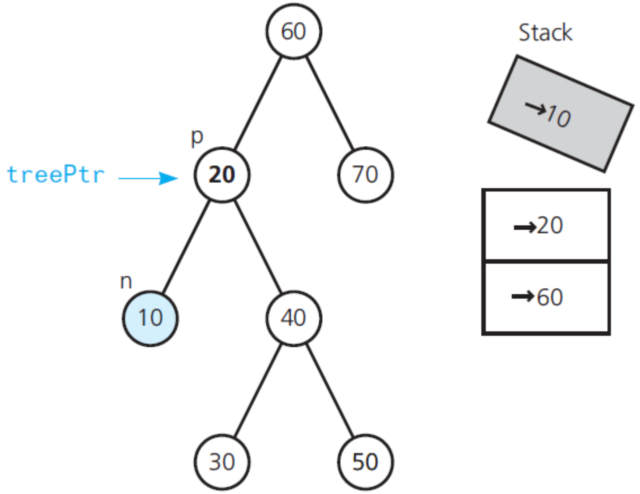
The Implementation

- Examine nonrecursive traversal – better understand relation between stacks and recursion
- Conceptual difficulty for nonrecursive traversal: where to go next after a particular node has been visited?
  - What happens when inorder returns from a recursive call?
  - The traversal retraces its steps by backing up the tree from a node n to its parent p, from which the recursive call to n was made.
  - Therefore, the pointer to n is popped from the stack and the pointer to p comes to the top of the stack.

The Implementation

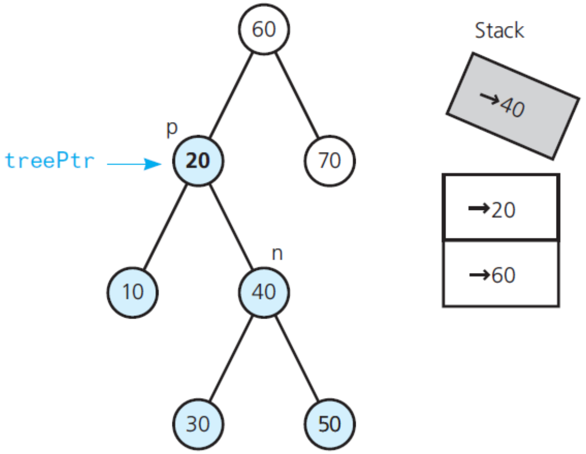
- Steps during an inorder traversal of the subtrees of 20

(a) Traversing 20's left subtree
(steps 9 and 10 in Figure 16-4)



Left subtree of 20 has been
traversed. Pop the reference
to 10 from the stack, visit 20.

(b) Traversing 20's right subtree



Right subtree of 20 has been
traversed. Pop the reference
to 40 from stack.

The Implementation

- Examine nonrecursive traversal – better understand relation between stacks and recursion
- Conceptual difficulty for nonrecursive traversal: where to go next after a particular node has been visited?
  - Two facts emerge from the recursive version of inorder when a return is made from a recursive call:
    - The implicit recursive stack of pointers is used to find the node p that the traversal must go back to
    - Once the traversal backs up to node p, it either visits p (for example, displays its data) or back farther up the tree.
    - It visits p if p’s left subtree has just been traversed;
    - it backs up if its right subtree has just been traversed.
    - The appropriate action is taken simply as a consequence of the point – 1 or 2 – to which control is returned.

The Implementation

- **Nonrecursive** inorder traversal (pseudocode)

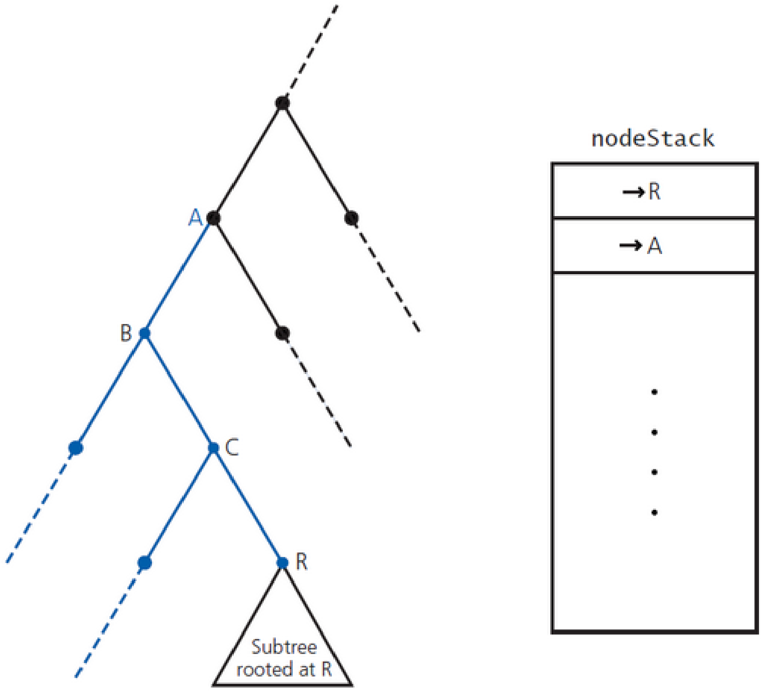
```
traverse(visit(item: ItemType): void): void
{
     // Initialize
     nodeStack = A new, empty stack
     curPtr = rootPtr // Start at root
     done = false
     while (!done)
     {
          if (curPtr != nullptr)
          {
                    // Place pointer to node on stack before traversing 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 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()
                    }
          }
     }
}
```

The Implementation

- Avoiding returns to nodes B and C



Thank you