Midterm 2
Review

Parameter Passing Methods

- Efficiency of parameter passing
  - Call-by-value
    - Requires copy be made → Overhead
  - Call-by-reference
    - Placeholder for actual argument
    - Most efficient method
  - Negligible difference for simple types
  - For class types → clear advantage
- Call-by-reference desirable
  - Especially for "large" data, like class types

The const Parameter Modifier

- Large data types (typically classes)
  - Desirable to use call-by-reference
  - Even if function *will not* make modifications
- Protect argument
  - Use constant parameter
    - Also called constant call-by-reference parameter
  - Place keyword *const* before type
  - Makes parameter "read-only"
  - Attempts to modify result in compiler error

Static Members

- Static member variables
  - All objects of class "share" one copy
  - One object changes it → all see change
- Useful for "tracking"
  - How often a member function is called
  - How many objects exist at given time
- Place keyword *static* before type

Static Functions

- Member functions can be static
  - If no access to *object data* needed
  - And still "must" be member of the class
  - Make it a static function
- Can then be called outside class
  - From non-class objects:
    - E.g., Server::getTurn();
  - As well as via class objects
    - Standard method: myObject.getTurn();
- Can only use static data, functions!

Static Members Example:
Display 7.6 Static Members (1 of 4)

Display 7.6 Static Members

```
#include <iostream>
using namespace std;

class Server
{
public:
    Server(char letterName);
    static int getTurn( );
    void serveOne( );
    static bool stillOpen( );
private:
    static int turn;
    static int lastServed;
    static bool nowOpen;
    char name;
};

int Server:: turn = 0;
int Server:: lastServed = 0;
bool Server::nowOpen = true;
```

Vector Basics

- Similar to an array:
  - Has base type
  - Stores collection of base type values
- Declared differently:
  - Syntax: vector<Base_Type>
    - Indicates template class
    - Any type can be "plugged in" to Base_Type
    - Produces "new" class for vectors with that type
  - Example declaration:
    `vector<int> v;`

Vector Use

- vector<int> v;
  - "v is vector of type int"
  - Calls class default constructor
    - Empty vector object created
- Indexed like arrays for access
- But to add elements:
  - Must call member function push_back
- Member function size()
  - Returns current number of elements

Vector Efficiency

- Member function capacity()
  - Returns memory currently allocated
  - Not same as size()
  - Capacity typically > size
    - Automatically increased as needed
- If efficiency critical:
  - Can set behaviors manually
    - v.reserve(32); //sets capacity to 32
    - v.reserve(v.size()+10); //sets capacity to 10 more than size
    - v.resize(10);

Overloading Basics

- Overloading operators
  - VERY similar to overloading functions
  - Operator itself is "name" of function
- Example Declaration:

```
const Money operator +(const Money& amount1,
                       const Money& amount2);
```

  - Overloads + for operands of type Money
  - Uses constant reference parameters for efficiency
  - Returned value is type Money
    - Allows addition of "Money" objects

Overloaded "+"

- Given previous example:
  - Note: overloaded "+" NOT member function
  - Definition is "more involved" than simple "add"
    - Requires issues of money type addition
    - Must handle negative/positive values
- Operator overload definitions generally very simple
  - Just perform "addition" particular to "your" type

Money "+" Definition: **Display 8.1** Operator Overloading

- Definition of "+" operator for Money class:

```
const Money operator +(const Money& amount1, const Money& amount2)
{
    int allCents1 = amount1.getCents( ) + amount1.getDollars( )*100;
    int allCents2 = amount2.getCents( ) + amount2.getDollars( )*100;
    int sumAllCents = allCents1 + allCents2;
    int absAllCents = abs(sumAllCents); //Money can be negative.
    int finalDollars = absAllCents/100;
    int finalCents = absAllCents%100;

    if (sumAllCents < 0)
    {
        finalDollars = -finalDollars;
        finalCents = -finalCents;
    }

    return Money(finalDollars, finalCents);
}
```

If the **return** *statements puzzle you, see the tip entitled* ***A Constructor Can Return an Object.***

Overloading as Member Functions

- Previous examples: standalone functions
  - Defined outside a class
- Can overload as "member operator"
  - Considered "member function" like others
- When a binary operator is a member function:
  - Only ONE parameter, not two!
  - Calling object serves as 1st parameter

Member Operator in Action

- Money cost(1, 50), tax(0, 15), total;
  total = cost + tax;
  - If "+" overloaded as member operator:
    - Object cost is calling object
    - Object tax is single argument
  - Think of as: total = cost.+(tax);
- Declaration of "+" in class definition:
  - const Money operator +(const Money& amount);
  - Notice only ONE argument

Overloading Operators: Which Method?

- Object-Oriented-Programming
  - Principles suggest member operators
  - Many agree, to maintain "spirit" of OOP
- Member operators more efficient
  - No need to call accessor & mutator functions
- At least one significant disadvantage
  - Lose automatic type conversion of the first operand

Friend Functions

- Nonmember functions
  - Recall: operator overloads as nonmembers
    - They access data through accessor and mutator functions
    - Very inefficient (overhead of calls)
- Friends can directly access private class data
  - No overhead, more efficient
- So: best to make nonmember operator overloads friends!

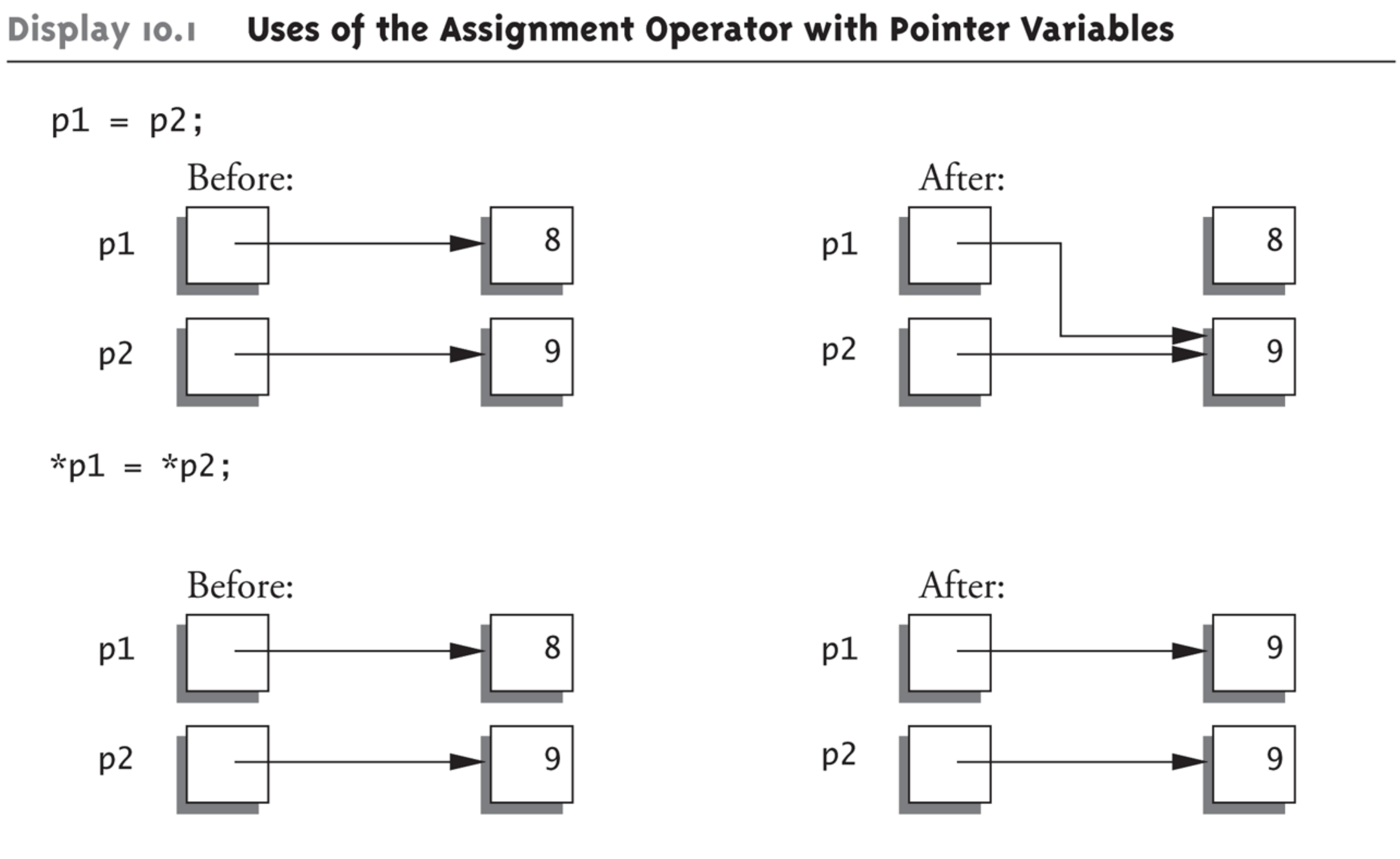
Friend Functions

- Friend function of a class
  - Not a member function
  - Has direct access to private members
    - Just as member functions do
- Use keyword *friend* in front of function declaration
  - Specified IN class definition
  - But they're NOT member functions!

Pointer Assignments

- Pointer variables can be "assigned":
  int *p1, *p2;
  p2 = p1;
  - Assigns one pointer to another
  - "Make p2 point to where p1 points"
- Do not confuse with:
  *p1 = *p2;
  - Assigns "value pointed to" by p2, to "value pointed to" by p1

Pointer Assignments Graphic: **Display 10.1** Uses of the Assignment Operator with Pointer Variables



delete Operator

- De-allocate dynamic memory
  - When a dynamic variable is no longer needed
  - Returns memory to freestore
  - Example:
    ```
    int *p;
    p = new int(5);
    … //Some processing…
    delete p;
    ```
  - De-allocates dynamic memory "pointed to by pointer p"
    - Literally "destroys" memory

Define Pointer Types

- Can "name" pointer types
- To be able to declare pointers like other variables
  - Eliminate need for "*" in pointer declaration
- typedef int* IntPtr;
  - Defines a "new type" alias
  - Consider these declarations:
    IntPtr p;
    int *p;
    - The two are equivalent

Creating Dynamic Arrays

- Very simple!
- Use new operator
  - Dynamically allocate with pointer variable
  - Treat like standard arrays
- Example:

```
typedef double * DoublePtr;
DoublePtr d;
d = new double[10];    //Size in brackets
```

  - Creates dynamically allocated array variable *d,* with ten elements, base type double

Deleting Dynamic Arrays

- Allocated dynamically at run-time
  - So should be destroyed at run-time
- Simple again. Recall Example:

```
d = new double[10];
… //Processing
delete [] d;
```

  - De-allocates all memory for dynamic array
  - Brackets indicate "array" is there
  - Recall: *d* still points there!
    - Should set `d = NULL;`

Constructors in Derived Classes

- Base class constructors are NOT inherited in derived classes!
  - But they can be invoked within derived class constructor
    - Which is all we need!
- Base class constructor must initialize all base class member variables
  - Those inherited by derived class
  - So derived class constructor simply calls it
    - "First" thing derived class constructor does

Derived Class Constructor Example

- Consider syntax for HourlyEmployee constructor:

```
HourlyEmployee::HourlyEmployee(string theName,
                    string theNumber, double theWageRate,
                    double theHours)
            : Employee(theName, theNumber),
            wageRate(theWageRate), hours(theHours)
{
    //Deliberately empty
}
```

- Portion after : is "initialization section"
  - Includes invocation of Employee constructor

The protected: Qualifier

- New classification of class members
- Allows private data of base class access "by name" in derived class
  - But nowhere else
  - Still no access "by name" in other classes
- In class it's defined → acts like private
- Considered "protected" in derived class
  - To allow future derivations
- Many feel this "violates" information hiding

Redefining vs. Overloading

- Very different!
- Redefining in derived class:
  - SAME parameter list
  - Essentially "re-writes" same function
- Overloading:
  - Different parameter list
  - Defined "new" function that takes different parameters
  - Overloaded functions must have different signatures

Functions Not Inherited

- All "normal" functions in base class are inherited in derived class
- Exceptions:
  - Constructors (we've seen)
  - Destructors
  - Copy constructor
    - But if not defined, generates "default" one
    - Recall need to define one for pointers!
  - Assignment operator
    - If not defined → default

Protected and Private Inheritance

- New inheritance "forms"
  - Both are rarely used
- Protected inheritance:
  class SalariedEmployee : protected Employee
  {…}
  - Public members in base class become protected in derived class
- Private inheritance:
  class SalariedEmployee : private Employee
  {…}
  - All members in base class become private in derived class

Linked Data Structures

- Three ways to handle such data structures:
  1. C-style approach: global functions and structs with everything public
  2. Classes with private member variables and accessor and mutator functions
  3. Friend classes
- Linked lists will use method 1
- Stacks, queues, sets, and hash tables will use method 2
- Trees will use method 3

Nodes and Linked Lists

- Linked list
  - Simple example of "dynamic data structure"
  - Composed of nodes
- Each "node" is variable of struct or class type that's dynamically created with new
  - Nodes also contain pointers to other nodes
  - Provide "links"

Node Definition

- struct ListNode
```
{
    string item;
    int count;
    ListNode *link;
};

typedef ListNode* ListNodePtr;
```
- Order here is important!
  - Listnode defined 1st, since used in typedef

Example Node Access

- (*head).count = 12;
  - Sets *count* member of node pointed to by *head* equal to 12
- Alternate operator, ->
  - Called "arrow operator"
  - Shorthand notation that combines * and .
  - head->count = 12;
    - Identical to above
- cin >> head->item
  - Assigns entered string to *item* member

Linked List Class Definition

- ```
  class IntNode
  {
  public:
          IntNode() { }
          IntNode(int theData, IntNOde* theLink)
                  : data(theData), link(theLink) { }
          IntNode* getLink()  {return link;}
          int getData()  {return data;}
          void setData(int theData)          {data = theData;}
          void setLink(IntNode* pointer)     {link=pointer;}
  private:
          int data;
          IntNode *link;
  };
  typedef IntNode* IntNodePtr;
  ```
  ```

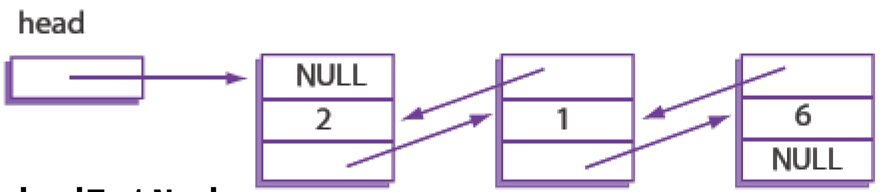
Searching a Linked List

- Function with two arguments:
  IntNodePtr search(IntNodePtr head, int target);
  //Precondition: pointer head points to head of
  //linked list. Pointer in last node is NULL.
  //If list is empty, head is NULL
  //Returns pointer to 1st node containing target
  //If not found, returns NULL
- Simple "traversal" of list
  - Similar to array traversal

Doubly Linked Lists

- What we have described is a singly linked list
  - Can only follow links in one direction
- Doubly Linked List
  - Links to the next node and another link to the previous node
  - Can follow links in either direction
  - NULL signifies the beginning and end of the list
  - Can make some operations easier, e.g. deletion since we don’t need to have a *before variable* to remember the node that links to the node we wish to discard.

Doubly Linked Lists




```
class DoublyLinkedIntNode
{
public:
    DoublyLinkedIntNode ( ){}
    DoublyLinkedIntNode (int theData, DoublyLinkedIntNode* previous,
                         DoublyLinkedIntNode* next)
            : data(theData), nextLink(next), previousLink(previous) {}
    DoublyLinkedIntNode* getNextLink( ) { return nextLink; }
    DoublyLinkedIntNode* getPreviousLink( ) { return previousLink; }
    int getData( ) { return data; }
    void setData(int theData) { data = theData; }
    void setNextLink(DoublyLinkedIntNode* pointer) { nextLink = pointer; }
    void setPreviousLink(DoublyLinkedIntNode* pointer)
         { previousLink = pointer; }
private:
    int data;
    DoublyLinkedIntNode *nextLink;
    DoublyLinkedIntNode *previousLink;
};
typedef DoublyLinkedIntNode* DoublyLinkedIntNodePtr;
```

Stacks

- Stack data structure:
  - Retrieves data in reverse order of how stored
  - LIFO – last-in/first-out data structure
  - Think of like “a hole in the ground"
- Stacks used for many tasks:
  - Track C++ function calls
  - Memory management
- Our use:
  - Use linked lists to implement stacks

Display 17.17 Interface File for a Stack Template Class (1 of 2)

Interface File for a Stack Template Class

```
//This is the header file stack.h. This is the interface for the class
//Stack, which is a template class for a stack of items of type T.
#ifndef STACK_H
#define STACK_H

namespace StackSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
        Node<T> *link;
    };
```

You might prefer to replace the parameter type T with `const T&`.

Display 17.17 Interface File for a Stack Template Class (2 of 2)

Interface File for a Stack Template Class

```
        template<class T>
    class Stack
    {
public:
        Stack( );
        //Initializes the object to an empty stack.

        Stack(const Stack<T>& aStack);
```

Copy constructor

```
        Stack<T>& operator =(const Stack<T>& rightSide);

        virtual ~Stack( );
```

The destructor destroys the stack and returns all the memory to the freestore.

```
        void push(T stackFrame);
        //Postcondition: stackFrame has been added to the stack.

        T pop( );
        //Precondition: The stack is not empty.
        //Returns the top stack frame and removes that top
        //stack frame from the stack.

        bool isEmpty( ) const;
        //Returns true if the stack is empty. Returns false otherwise.
    private:
        Node<T> *top;
    };

}//StackSavitch
#endif //STACK_H
```

Queues

- Another common data structure:
  - Handles data in first-in/first-out manner (FIFO)
  - Items inserted to end of list
  - Items removed from front
- Representation of typical "line" forming
  - Like bank teller lines, movie theatre lines, etc.

Display 17.20 Interface File for a Queue Template Class (1 of 3)

Interface File for a Queue Template Class

```
//This is the header file queue.h. This is the interface for the class
//Queue, which is a template class for a queue of items of type T.
#ifndef QUEUE_H
#define QUEUE_H

namespace QueueSavitch
{
    template<class T>
    class Node
    {
    public:
        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
        Node<T>* getLink( ) const { return link; }
        const T getData( ) const { return data; }
        void setData(const T& theData) { data = theData; }
        void setLink(Node<T>* pointer) { link = pointer; }
    private:
        T data;
```

This is the same definition of the template class Node *that we gave for the stack interface in Display 17.13. See the tip "A Comment on Namespaces" for a discussion of this duplication.*

(continued)

Hash Tables

- A hash table or hash map is a data structure that efficiently stores and retrieves data from memory
- Here we discuss a hash table that uses an array in combination with singly linked lists
- Uses a hash function
  - Maps an object to a key
  - In our example, a string to an integer

Simple Hash Function for Strings

- Sum the ASCII value of every character in the string and then compute the modulus of the sum using the size of the fixed array.

```
int computeHash(string s)
{
  int hash = 0;
  for (int i = 0; i < s.length( ); i++)
  {
    hash = hash + s[i];
  }
  return hash % SIZE;  // SIZE = 10 in example
}
```

```
Example:  “dog” = ASCII 100, 111, 103
Hash = (100 + 111 + 103) % 10      =  4
```

Hash Table Idea

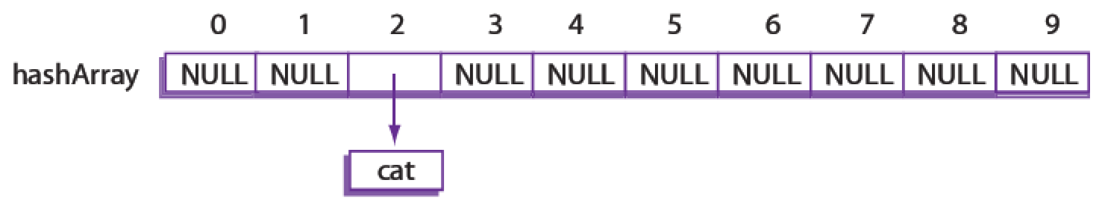
- Storage
  - Make an array of fixed size, say 10
  - In each array element store a linked list
  - To add an item, map (i.e., hash) it to one of the 10 array elements, then add it to the linked list at that location
- Retrieval
  - To look up an item, determine its hash code then search the linked list at the corresponding array slot for the item

Constructing a Hash Table

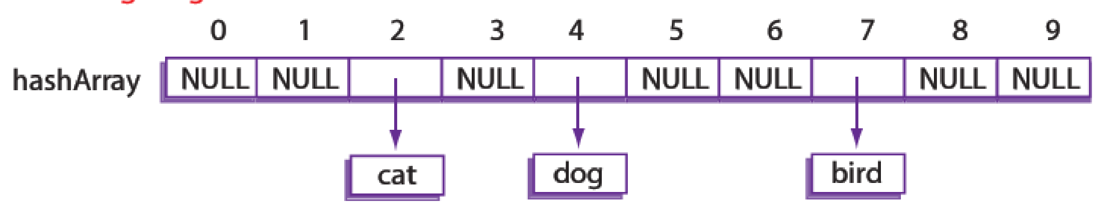
Existing hash table with 10 empty linked lists

```
Node<string> *hashArray[10];
for (int i=0; i<10; i++) hashArray[i] = NULL;
```

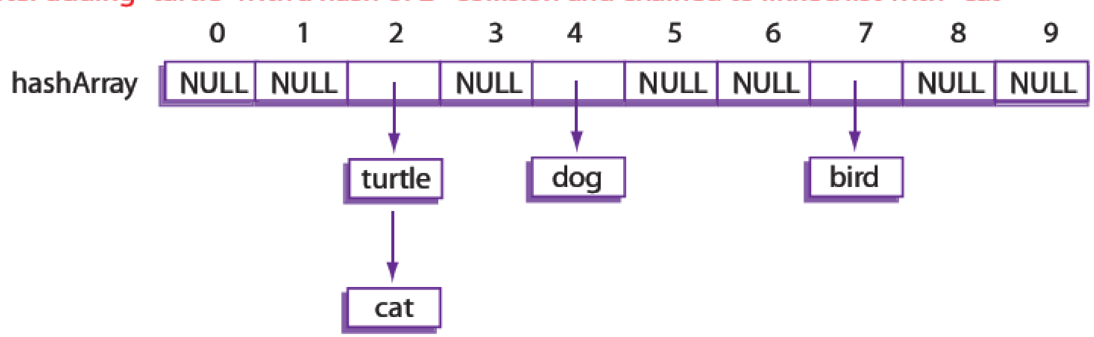
After adding "cat" with a hash of 2



After adding "dog" with a hash of 4 and "bird" with a hash of 7



After adding "turtle" with a hash of 2 - collision and chained to linked list with "cat"



Interface File for a HashTable Class (1 of 2)

```
// This is the header file hashtable.h.  This is the interface
// for the class HashTable, which is a class for a hash table
// of strings.
#ifndef HASHTABLE_H
#define HASHTABLE_H

#include <string>
#include "listtools.h"
```

The library "listtools.h" is the linked list library interface from Display 17.14.

```
using LinkedListSavitch::Node;
using std::string;

namespace HashTableSavitch
{
 const int SIZE = 10;  // Maximum size of the hash table array
```

Interface File for a HashTable Class (2 of 2)

```
 class HashTable
 {
  public:
      HashTable(); // Initialize empty hash table
      // Normally a copy constructor and overloaded assignment
      // operator would be included.  They have been omitted
      // to save space.
      virtual ~HashTable();  // Destructor destroys hash table

      bool containsString(string target) const;
      // Returns true if target is in the hash table,
      // false otherwise

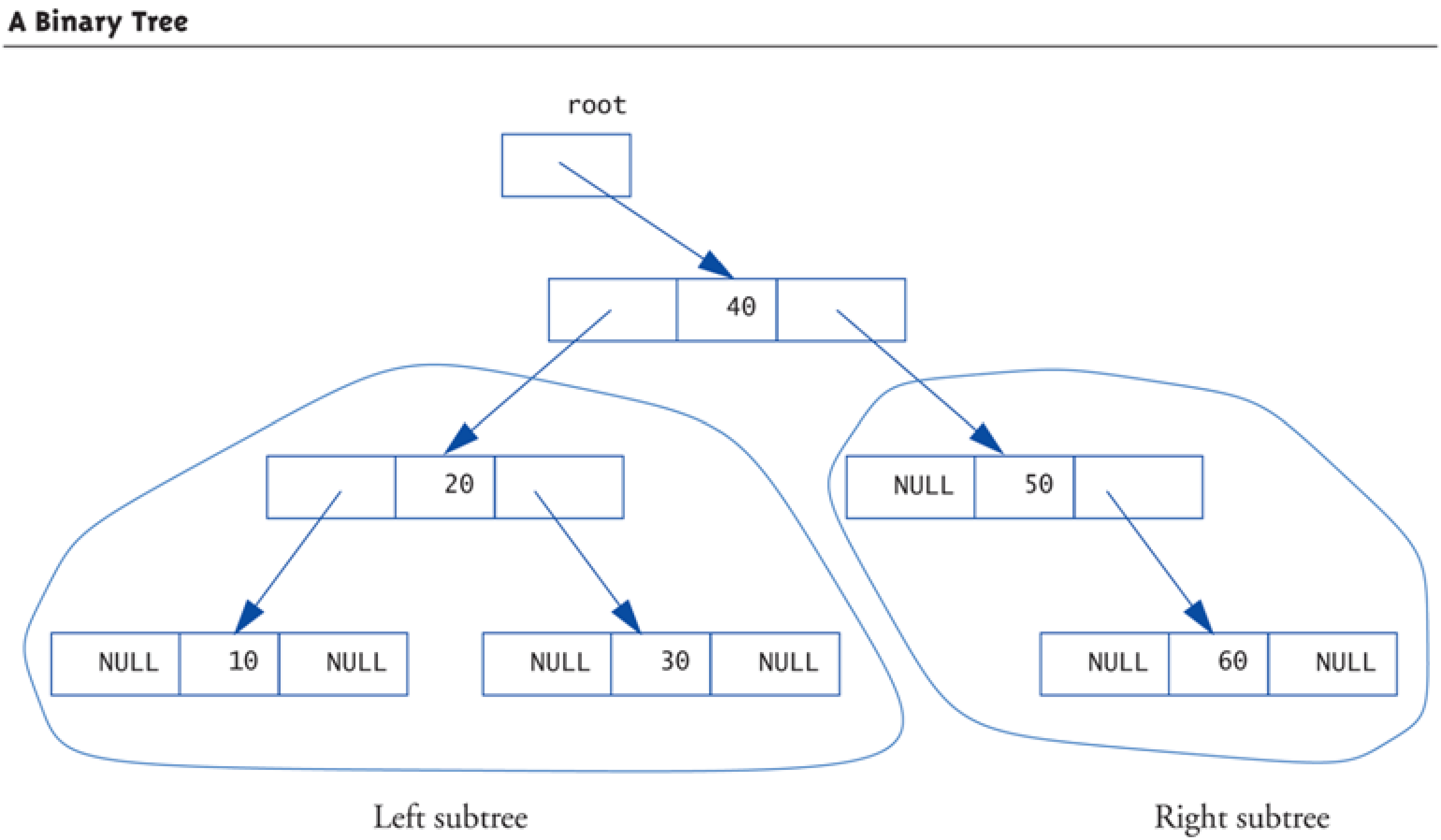
      void put(string s);
      // Adds a new string to the hash table

  private:
      Node<string> *hashArray[SIZE];      // The actual hash table
      static int computeHash(string s);   // Compute a hash value
 }; // HashTable
} // HashTableSavitch
#endif // HASHTABLE_H
```

Tree Structures

- Trees can be complex data structures
- Only basics here:
  - Constructing, manipulating
  - Using nodes and pointers
- Recall linked list: nodes have only one pointer → next node
- Trees have two, & sometimes more, pointers to other nodes

Tree Structure: **Display 17.35** A Binary Tree (1 of 2)



Tree Structure:
Display 17.35 A Binary Tree (2 of 2)

```
class IntTreeNode
{
public:
    IntTreeNode(int theData, IntTreeNode* left, IntTreeNode* right)
        : data(theData), leftLink(left), rightLink(right){}
private:
    int data;
    IntTreeNode *leftLink;
    IntTreeNode *rightLink;
};


IntTreeNode *root;
```