Chapter 17

Linked Data
Structures

Learning Objectives

- Nodes and Linked Lists
  - Creating, searching
- Linked List Applications
  - Stacks, queues, sets, hash tables
  - Friend classes, alternatives
- Iterators
  - Pointers as iterators
- Trees

Introduction

- Linked list
  - Constructed using pointers
  - Grows and shrinks during run-time
  - Doubly Linked List : A variation with pointers in both directions
- Trees also use pointers
- Pointers backbone of such structures
  - Use dynamic variables

Approaches

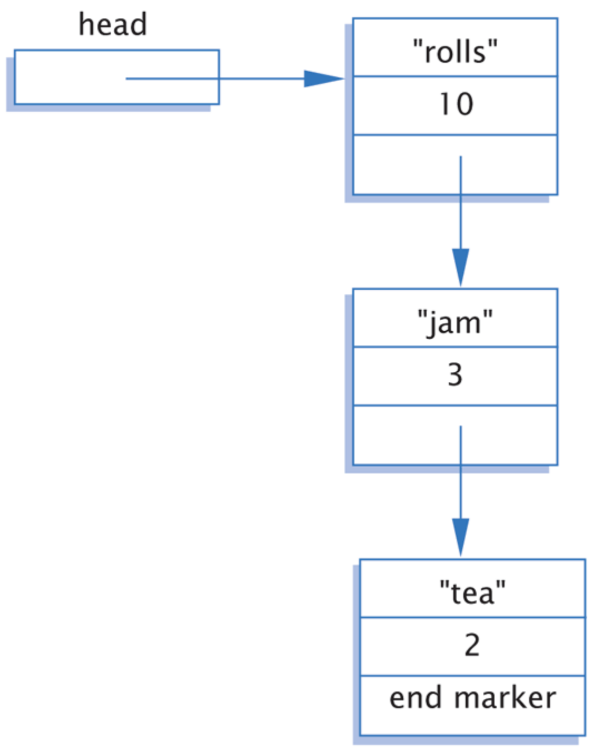
- Three ways to handle such data structures:
  1. C-style approach: global functions and structs with everything public
  2. Classes with private member variables and accessor and mutator functions
  3. Friend classes
- Linked lists will use method 1
- Stacks, queues, sets, and hash tables will use method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

Display 17.1 Nodes and Pointers

Display 17.1 Nodes and Pointers



Node Definition

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

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

Head Pointer

- Box labeled "head" not a node:
  ListNodePtr head;
  - A simple pointer to a node
  - Set to point to 1st node in list
- Head used to "maintain" start of list
- Also used as an *argument* to functions

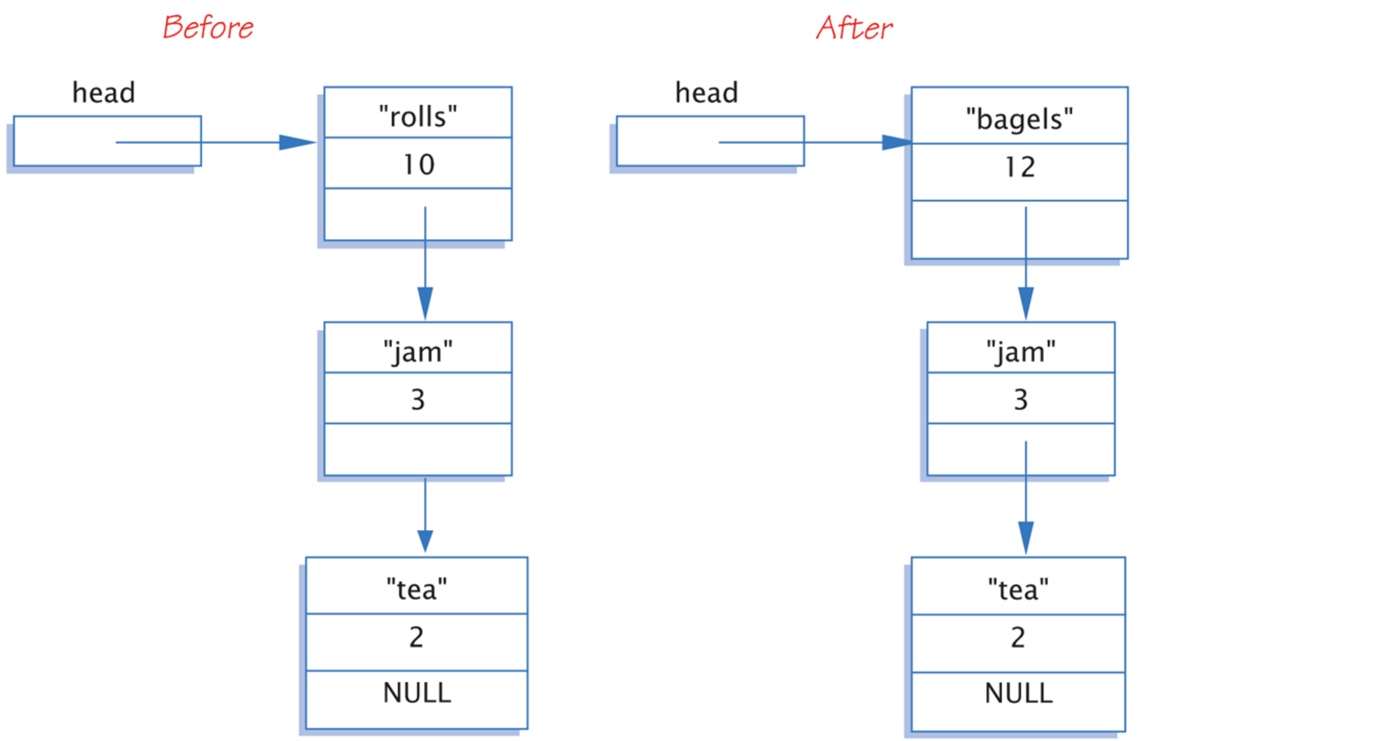
Example Node Access

- (*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

Display 17.2 Accessing Node Data

Display 17.2 Accessing Node Data

```
head->count = 12;
head->item = "bagels";
```



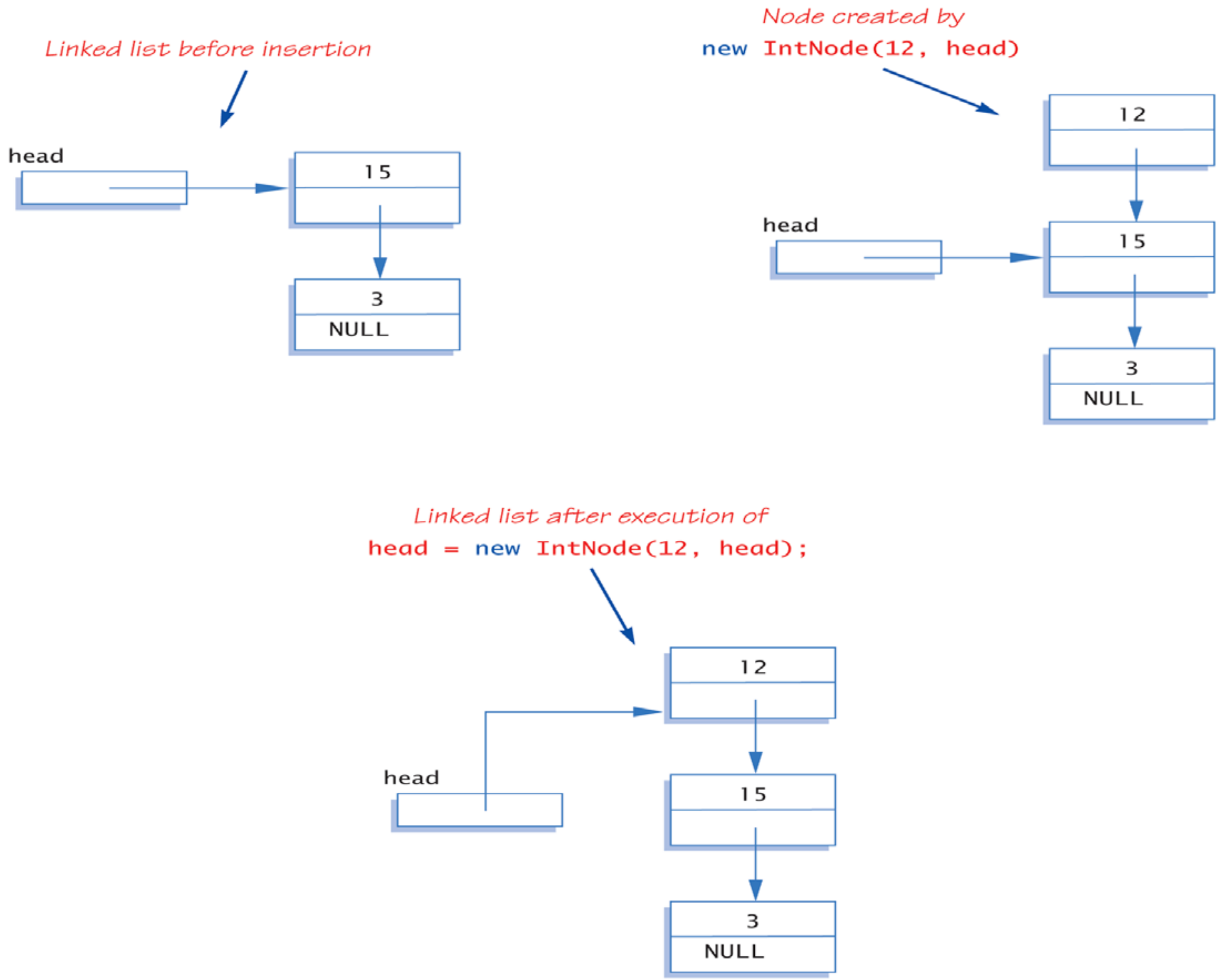
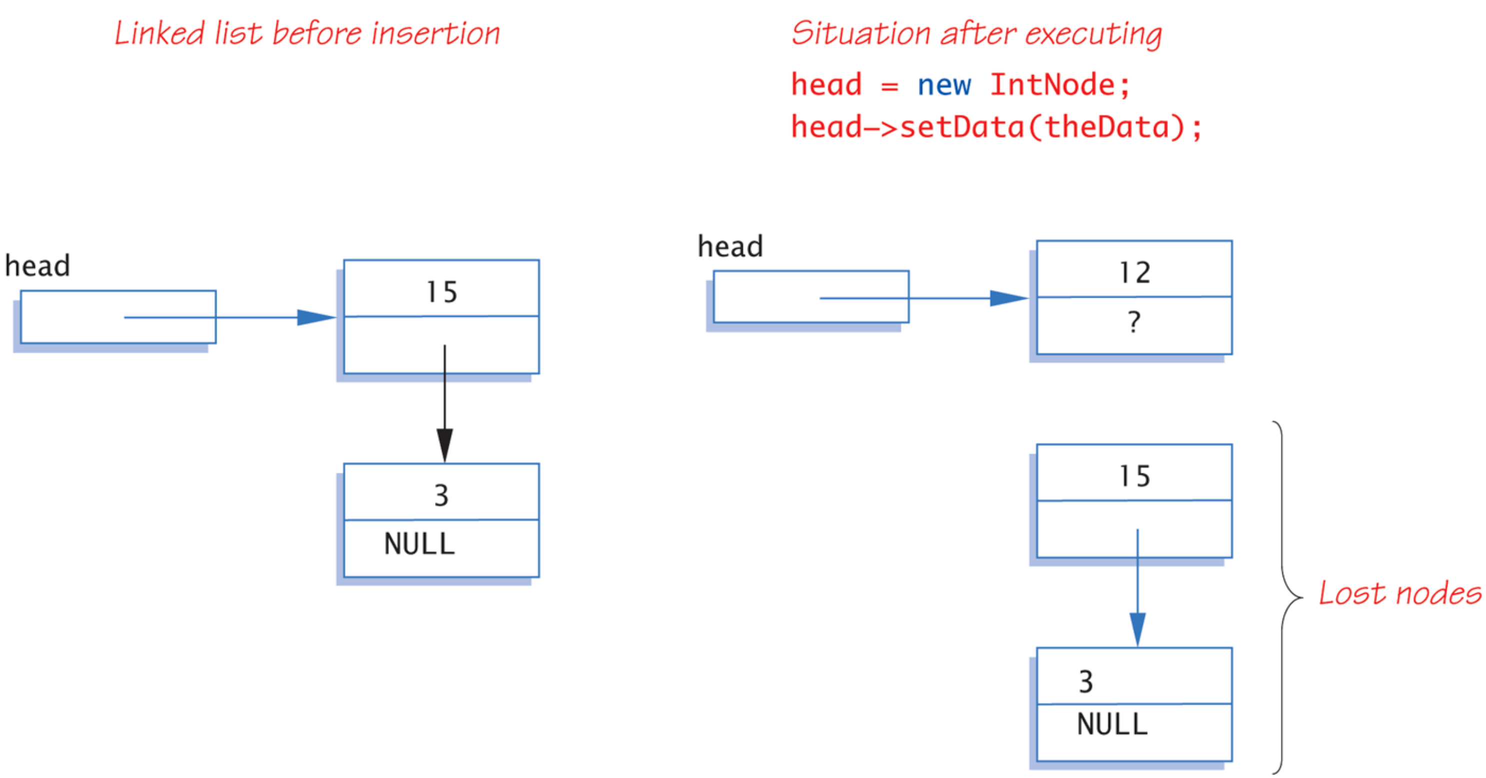
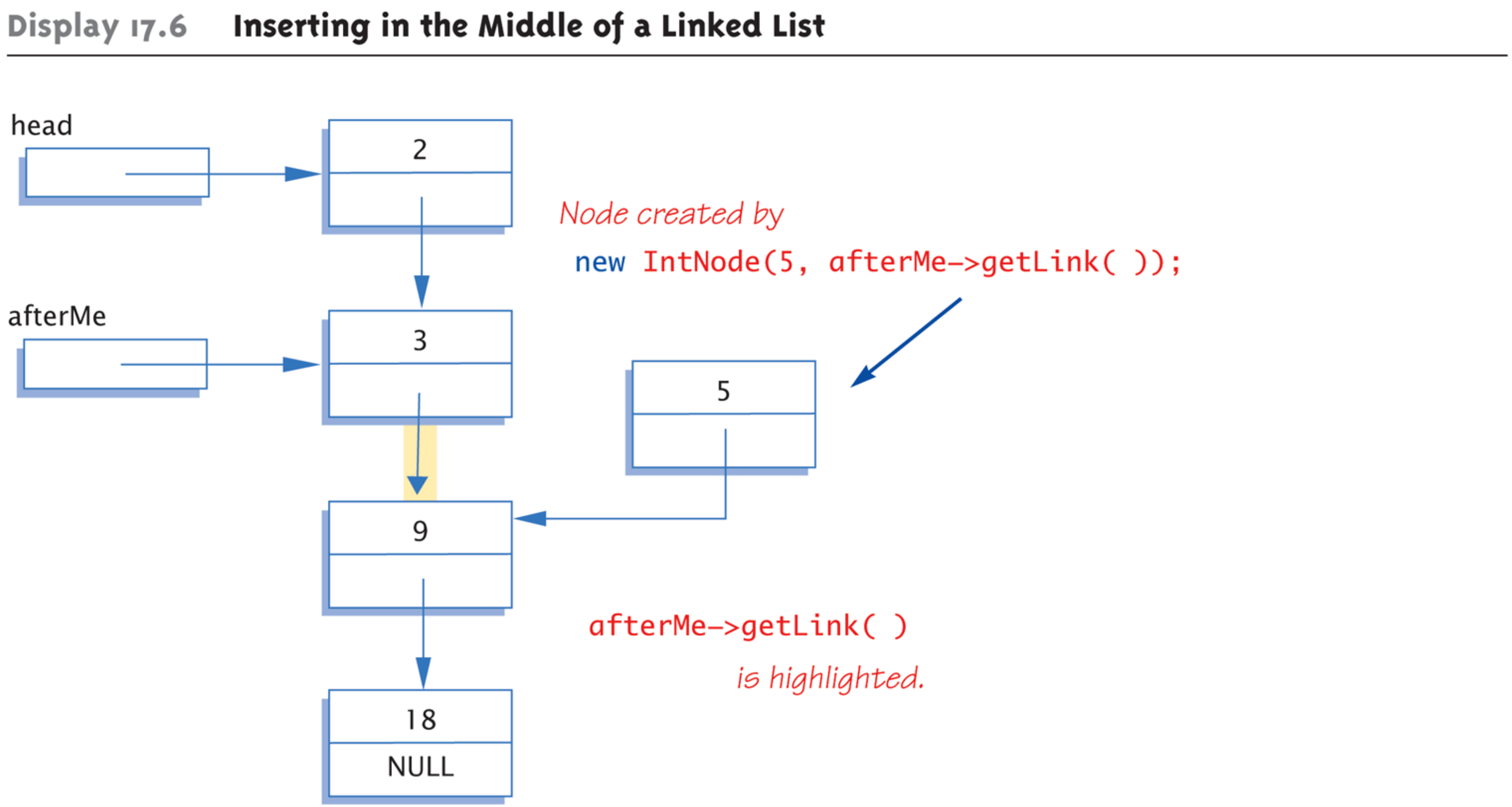
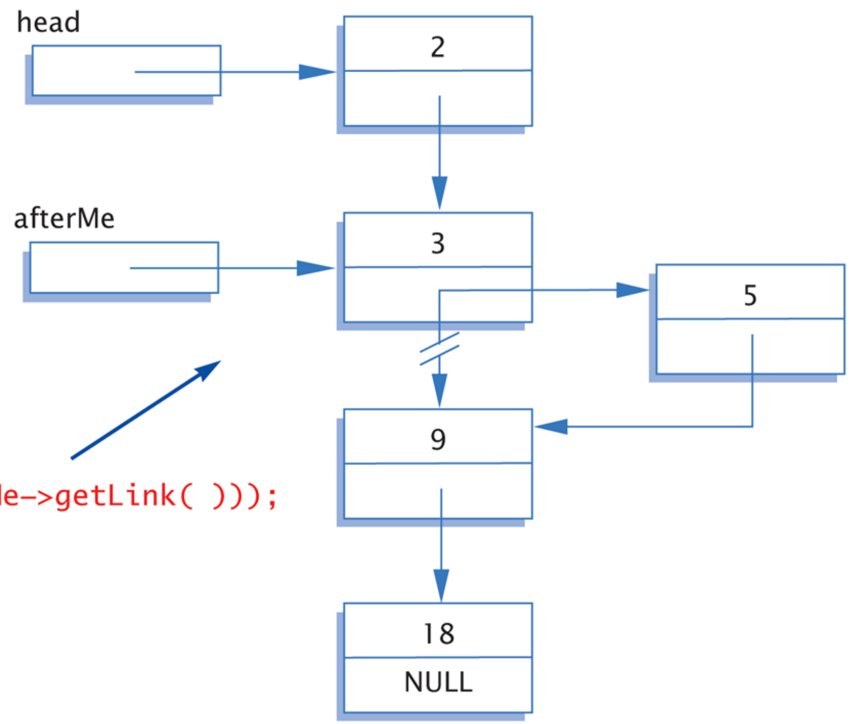
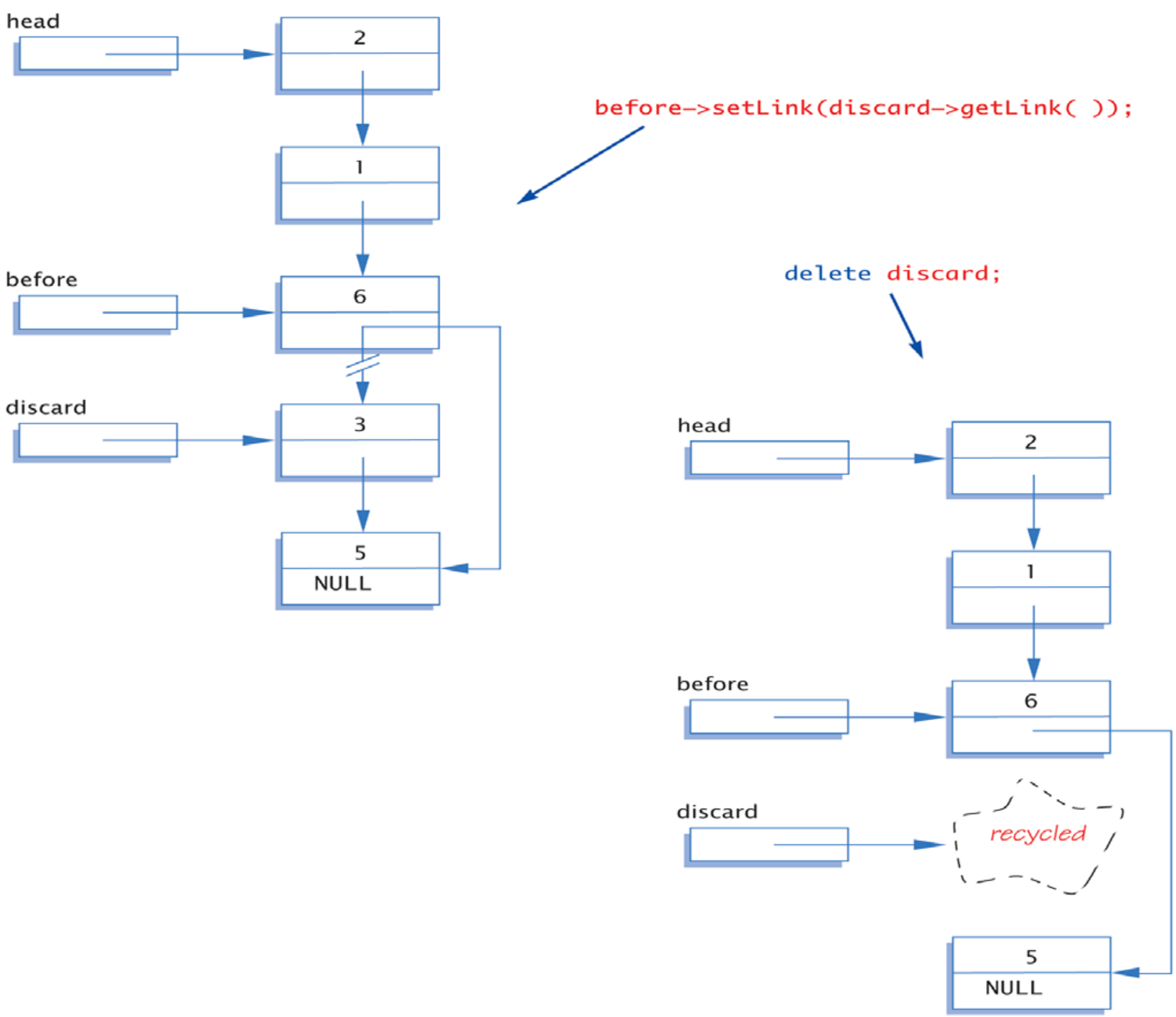
End Markers

- Use NULL for node pointer
  - Considered "sentinel" for nodes
  - Indicates no further "links" after this node
- Provides end marker similar to how we use partially-filled arrays

Linked List

- Lists as illustrated called linked lists
- First node called *head*
  - Pointed to by pointer named *head*
- Last node special also
  - It's member pointer variable is NULL
  - Easy test for "end" of linked list

Linked List Class Definition

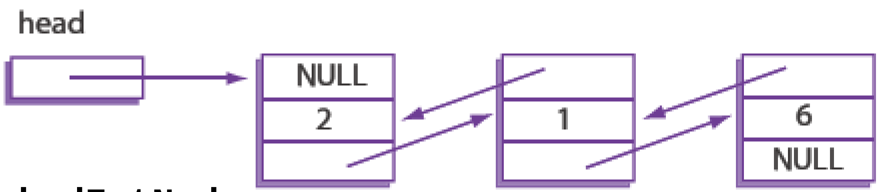
- ```
  class IntNode
  {
  public:
          IntNode() { }
          IntNode(int theData, IntNOde* theLink)
                      : data(theData), link(theLink) { }
          IntNode* getLink()  {return link;}
          int getData()  {return data;}
          void setData(int theData)            {data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

Linked List Class

- Notice all member function definitions are inline
  - Small and simple enough
- Notice two-parameter constructor
  - Allows creation of nodes with specific data value and specified link member
  - Example:
    IntNodePtr p2 = new IntNode(42, p1);

Create 1st Node

- IntNodePtr head;
  - Declares pointer variable *head*
- head = new IntNode;
  - Dynamically allocates new node
  - Our 1st node in list, so assigned to head
- head->setData(3);
  head->setLink(NULL);
  - Sets head node data
  - Link set to NULL since it's the only node!

Display 17.3 **Adding a Node to the Head of a Linked List**



Lost Nodes Pitfall: **Display 17.5** Lost Nodes

Display 17.5 Lost Nodes



Display 17.6 Inserting in the Middle of a Linked List (1 of 2)



Display 17.6 Inserting in the Middle of a Linked List (2 of 2)



Final result of
```
afterMe->setLink(
            new IntNode(theData, afterMe->getLink( )));
```

Removing a Node

Display 17.7 **Removing a Node**



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

Pseudocode for search Function

- while (here doesn't point to target node and here is not pointing to the last node)

```
{
    Make here point to next node in list
}
if (here node points to target)
    return here;
else
    return NULL;
```

Algorithm for search Function

- ```
  while (here->getData() != target &&
                  here->getLink() != NULL)
          here = here->getLink();

  if (here->getData() == target)
          return here;
  else
          return NULL;
  ```
- Must make "special" case for empty list
  - Not done here
  ```

Doubly Linked Lists

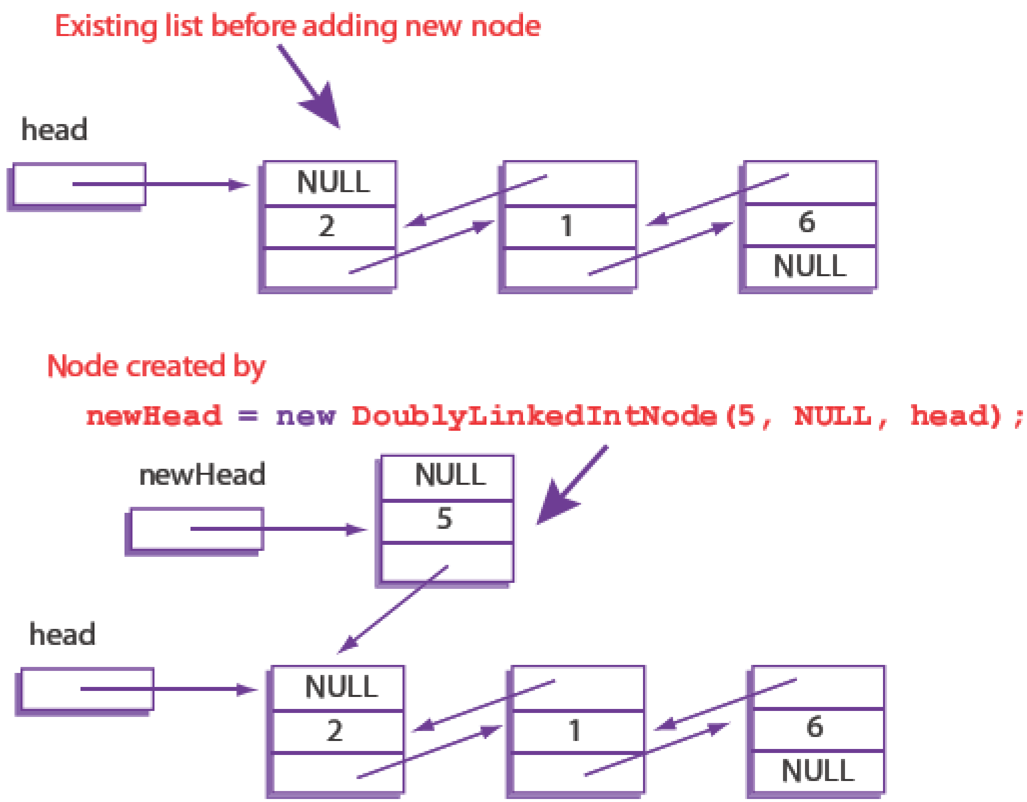
- What we have described is a singly linked list
  - Can only follow links in one direction
- Doubly Linked List
  - Links to the next node and another link to the previous node
  - Can follow links in either direction
  - NULL signifies the beginning and end of the list
  - Can make some operations 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;
```

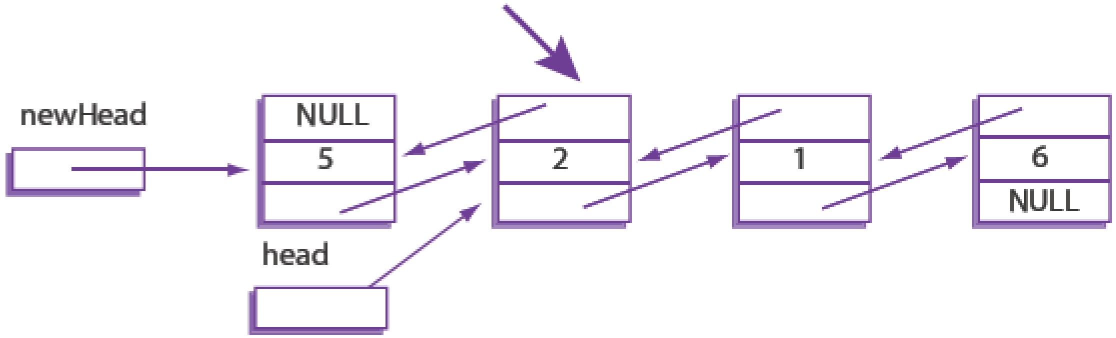
Adding a Node to the Front of a Doubly Linked List (1 of 2)



Adding a Node to the Front of a Doubly Linked List (2 of 2)

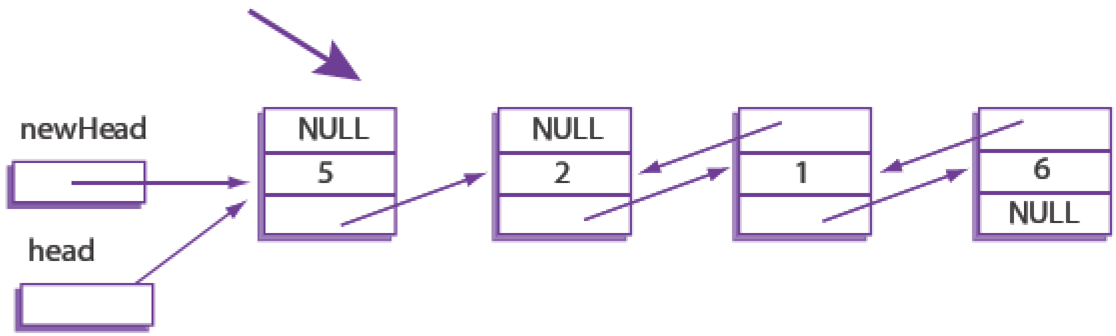
Set the previous link of the original head node

```
head->setPreviousNode(newHead);
```



Set head to newHead

```
head = newHead;
```

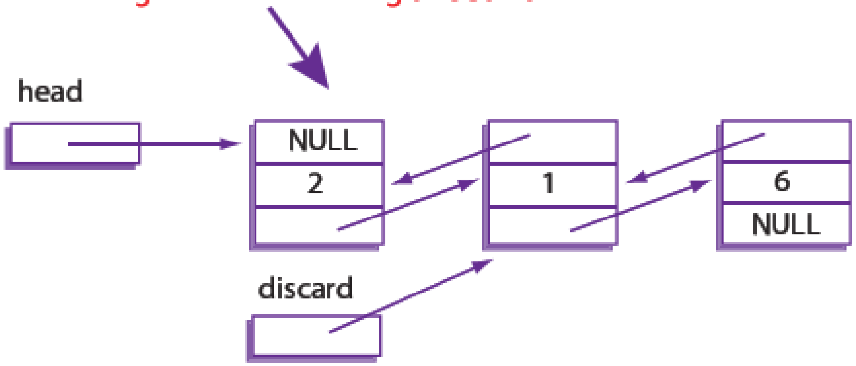


Deleting a Node from a Doubly Linked List

- Removing a node requires updating references on both sides of the node we wish to delete
- Thanks to the backward link we do not need a separate variable to keep track of the previous node in the list like we did for the singly linked list
  - Can access via node->previous

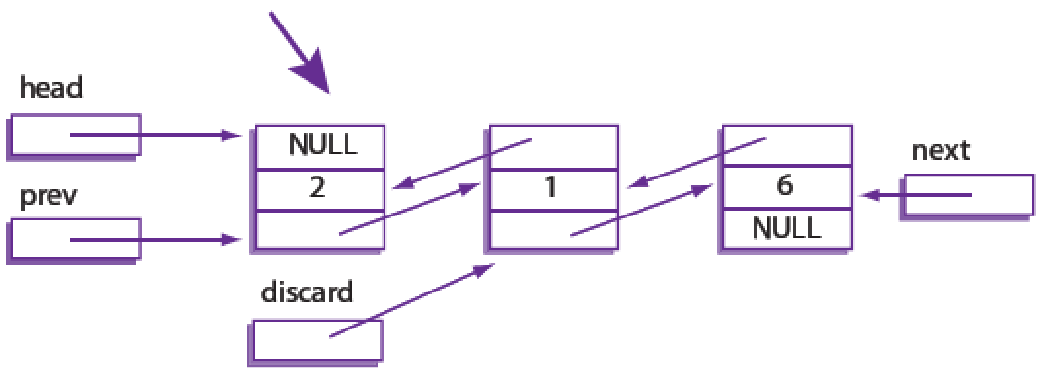
Deleting a Node from a Doubly Linked List (1 of 2)

Existing list before deleting `discard`

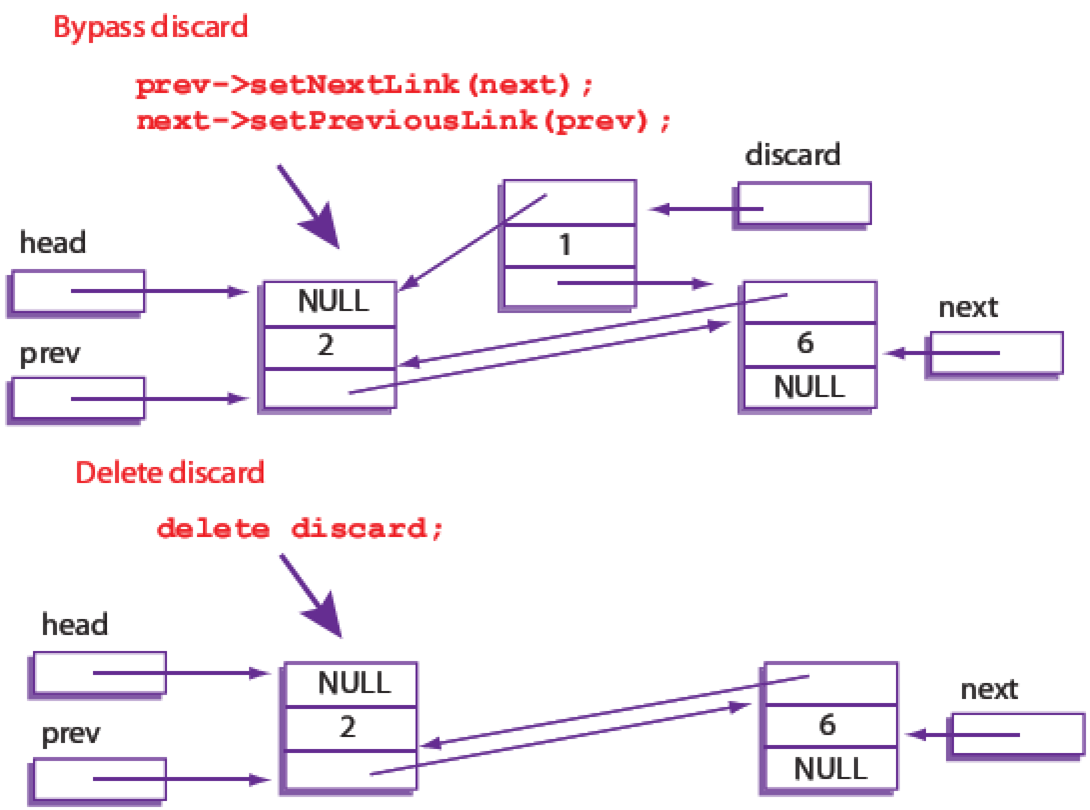


Set pointers to the previous and next nodes

```
DoublyLinkedIntNodePtr prev = discard->getPreviousLink( );
DoublyLinkedIntNodePtr next = discard->getNextLink( );
```



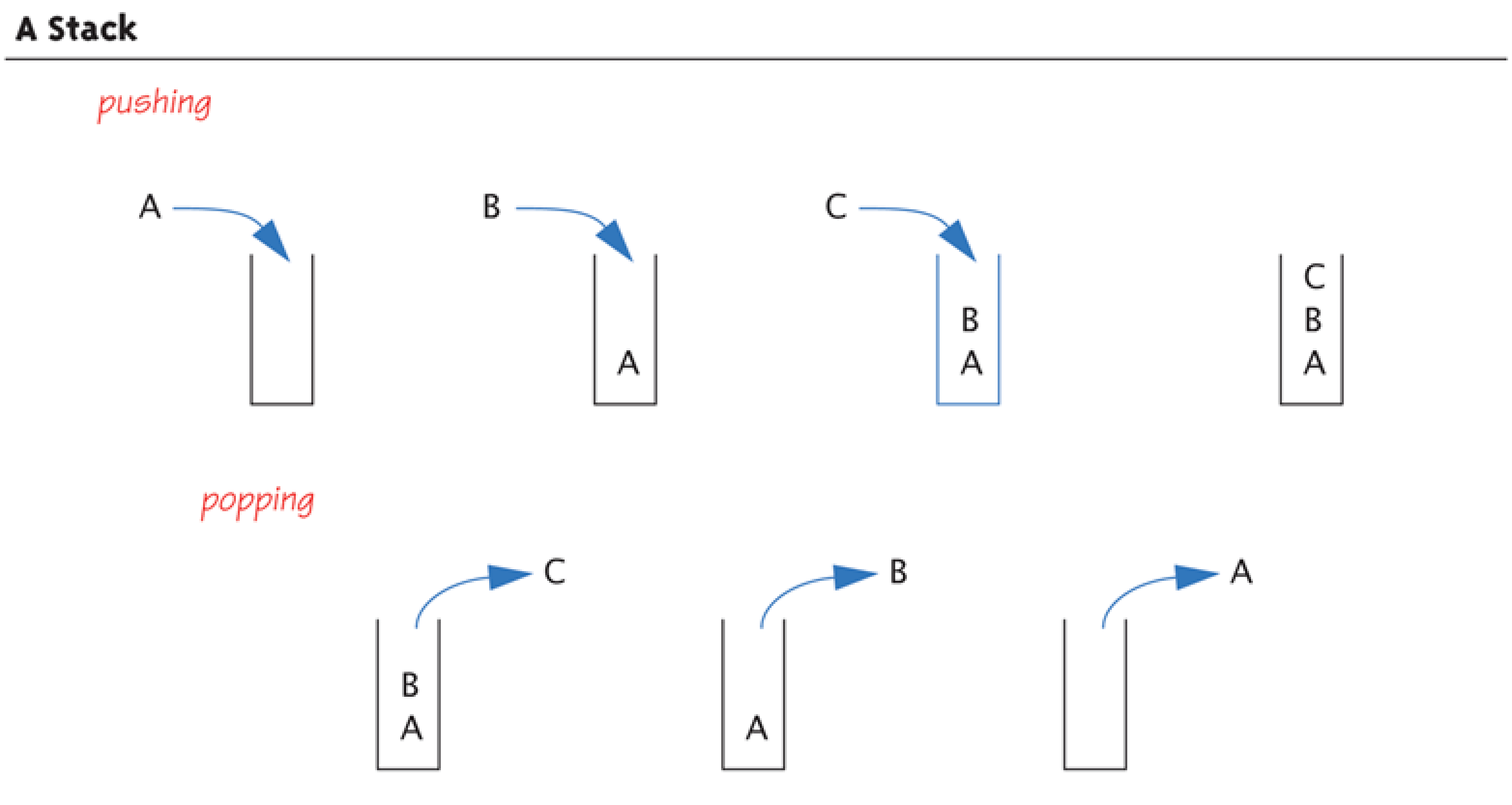
Deleting a Node from a Doubly Linked List (2 of 2)



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

A Stack—Graphic: **Display 17.12** A Stack



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
```

Stack Push and Pop

- Adding data item to stack → push
  - Considered "pushing" data onto stack
  - Recall: goes to "top" of stack
- Removing data item from stack → pop
  - Considered "popping" item off stack
  - Recall: removed from "top" of stack

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

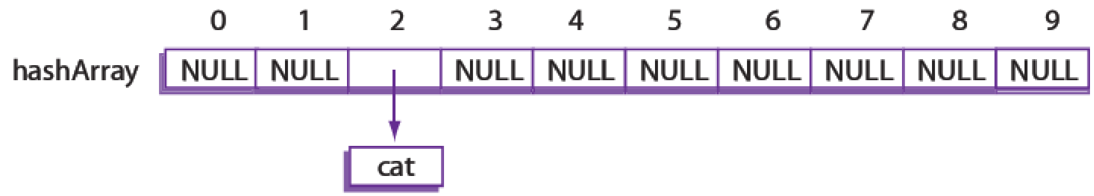
- Storage
  - Make an array of fixed size, say 10
  - In each array element store a linked list
  - To add an item, map (i.e., hash) it to one of the 10 array elements, then add it to the linked list at that location
- Retrieval
  - To look up an item, determine its hash code then search the linked list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

Constructing a Hash Table

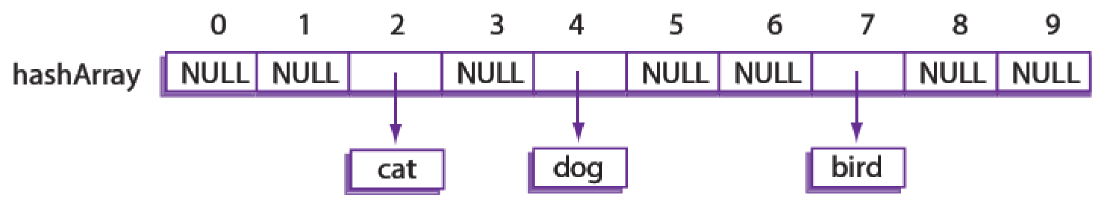
Existing hash table with 10 empty linked lists

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

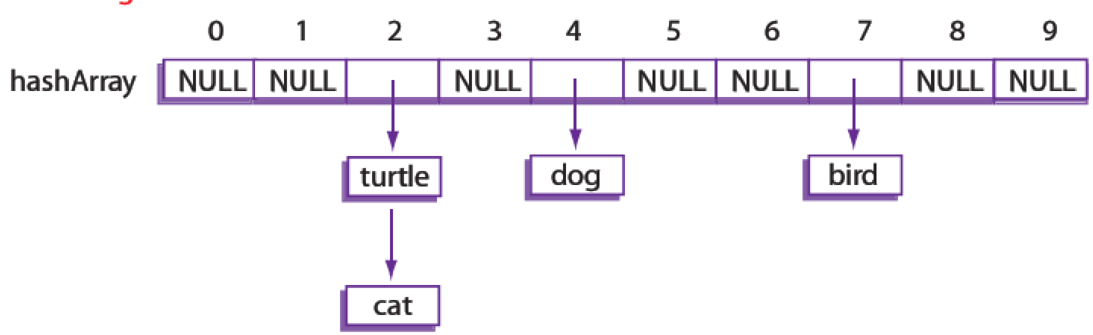
After adding "cat" with a hash of 2



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
```

Hash Table Demonstration

```
// Program to demonstrate use of the HashTable class
#include <string>
#include <iostream>
#include "hashtable.h"
#include "listtools.cpp"
#include "hashtable.cpp"
using std::string;
using std::cout;
using std::endl;
using HashTableSavitch::HashTable;

int main()
{
  HashTable h;
  cout << "Adding dog, cat, turtle, bird" << endl;
  h.put("dog");
  h.put("cat");
  h.put("turtle");
  h.put("bird");
  cout << "Contains dog? " << h.containsString("dog") << endl;
  cout << "Contains cat? " << h.containsString("cat") << endl;
  cout << "Contains turtle? " << h.containsString("turtle") << endl;
  cout << "Contains bird? " << h.containsString("bird") << endl;
  cout << "Contains fish? " << h.containsString("fish") << endl;
  cout << "Contains cow? " << h.containsString("cow") << endl;
  return 0;
}
```

SAMPLE DIALOGUE

```
Adding dog, cat, turtle, bird
Contains dog? 1
Contains cat? 1
Contains turtle? 1
Contains bird? 1
Contains fish? 0
Contains cow? 0
```

Hash Table Efficiency

- Worst Case
  - Every item inserted into the table has the same hash key, the find operation may have to search through all items every time (same performance as a linked list)
- Best Case
  - Every item inserted into the table has a different hash key, the find operation will only have to search a list of *size 1*, very fast
- Can decrease the chance of collisions with a better hash function
- Tradeoff: Lower chance of collision with bigger hash table, but more wasted memory space

Friend Classes

- Recall constant use of getLink and setlink accessor and mutator functions
  - Somewhat of a nuisance
  - Similar to making data public?!
    - Public makes available to ALL!
- Use friend class
  - Make queue template class "friend" of node template class
  - All private link members directly available in member functions of queue class!

Forward Declaration

- Class friendships typically require classes reference each other
  - Presents problem
  - How can "both" be declared at same time?
- Requires forward declaration
  - Simple class heading given inside other:
    class Queue;   //Forward Dec.
  - Announces "class Queue will exist"

Iterators

- Construct for cycling through data
  - Like a "traversal"
  - Allows "whatever" actions required on data
- Pointers typically used as iterators
  - Seen in linked list implementation

Pointers as Iterators

- Recall: linked list: "prototypical" data structure
- Pointer: "prototypical" example of iterator
  - Pointer used as iterator by moving thru linked list node by node starting at head:
  - Example:

```
Node_Type *iterator;
for (iterator = Head; iterator != NULL;
                                iterator=iterator->Link)
  Do_Action
```

Iterator Classes

- More versatile than pointer
- Typical overloaded operators:

  ++ advances iterator to next item
  -- retreats iterator to previous item
  == Compares iterators
  != Compare for not equal
  * Accesses one item
- Data structure class would have members:

  begin(): returns iterator to 1^{st} item in structure
  end(): returns iterator to test if at end

Iterator Class Example

- Cycle through data structure named *ds*:

  ```
  for (i=ds.begin();i!=ds.end();i++)
      process *i  //*i is current data item
  ```

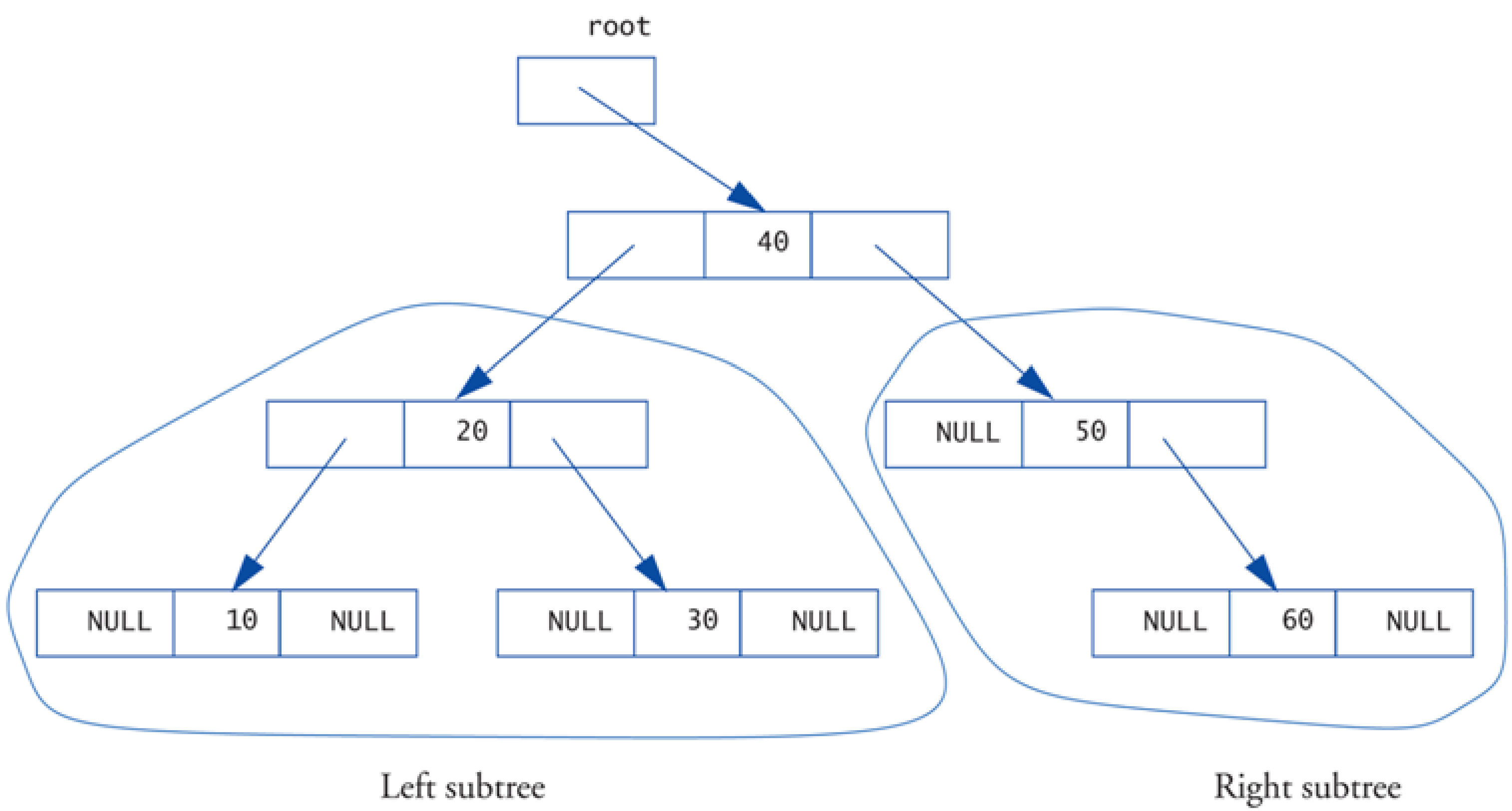
- i is name of iterator

Trees Introduction

- 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)

A Binary Tree



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;
```

Tree Properties

- Notice paths
  - From top to any node
  - No "cycles" – follow pointers, will reach "end"
- Notice here each node has two links
  - Called ***binary tree***
  - Most common type of tree
- Root node
  - Similar to linked list's head
- Leaf nodes
  - Both link variables are NULL (no subtrees)

Trees and Recursion

- Note tree’s "recursive structure"
- Each node has two subtrees
  - Each subtree has two subtrees
    - Etc., etc.
- Makes trees amenable to recursive algorithms
  - For searching especially!

Tree Processing

- **Preorder Processing:**
  1. Process data in root node
  2. Process left subtree
  3. Process right subtree
- **In-order Processing:**
  1. Process left subtree
  2. Process data in root
  3. Process right subtree
- **Postorder Processing:**
  1. Process left subtree
  2. Process right subtree
  3. Process data in root

Tree Storage

- Our example stored values in special way:
  - Called binary search tree storage rule:
    1. values in left subtree less than root value
    2. values in right subtree greater than root
    3. rule applies recursively to each subtree
- Trees using this storage mechanism:
  - Called binary search tree (BST)
  - Traversals:
    In-order processing → values "in order“ (from smallest to largest)

Summary 1

- Node is struct or class object
  - One or more members is pointer
  - Nodes connected by member pointers
    - Produce structures that grow and shrink at runtime
- Linked list
  - List of nodes where each node points to next
  - In a doubly linked lists there are pointers in both directions
- End of linked list marked with NULL pointer

Summary 2

- Stack is LIFO data structure
- Queue is FIFO data structure
- Hash Tables are data structures for quick storage and retrieval; can be implemented with a linked list
- Sets can be implemented with linked lists
- Iterator construct allows cycling through data items in given data structure
- Tree data structures
  - Nodes have two member pointers
  - Each point to other nodes/subtrees
- Binary search tree
  - Special storage rules allow rapid searches