



# 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
- Standard Template Library
  - Has predefined versions of some structures

# 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

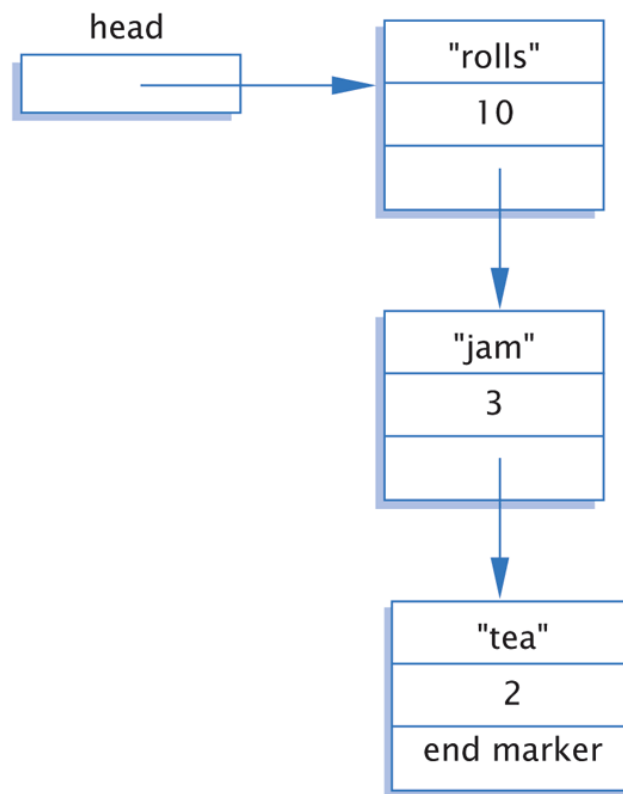
# Nodes and Linked Lists

- 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 1<sup>st</sup>, since used in typedef
- Also notice "circularity"

# Head Pointer

- Box labeled "head" not a node:  
ListNodePtr head;
  - A simple pointer to a node
  - Set to point to 1<sup>st</sup> node in list
- Head used to "maintain" start of list
- Also used as 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

# End Markers

- Use NULL or nullptr (in C++11) 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

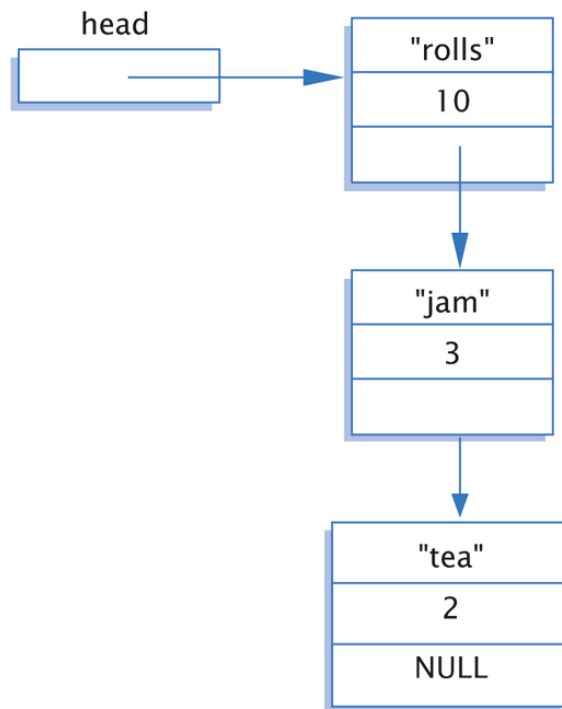
# Display 17.2 Accessing Node Data

## Display 17.2 Accessing Node Data

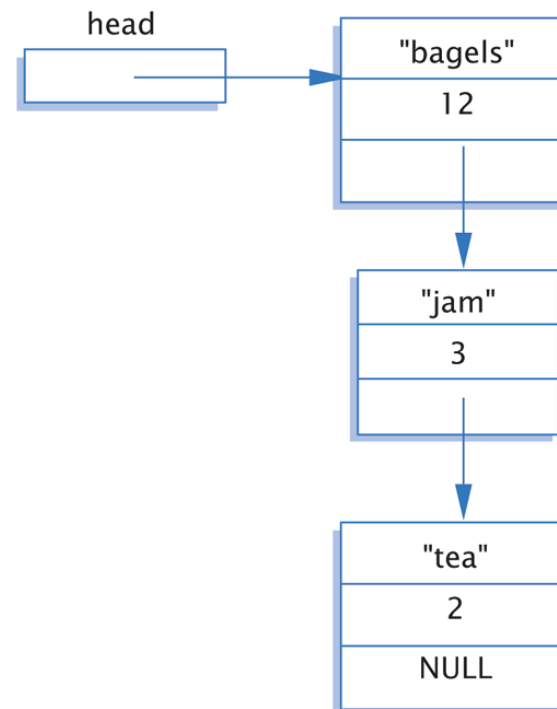
---

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

*Before*



*After*



# 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 (or nullptr in C++11)
  - 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() const {return link;}
    int getData() const {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);`

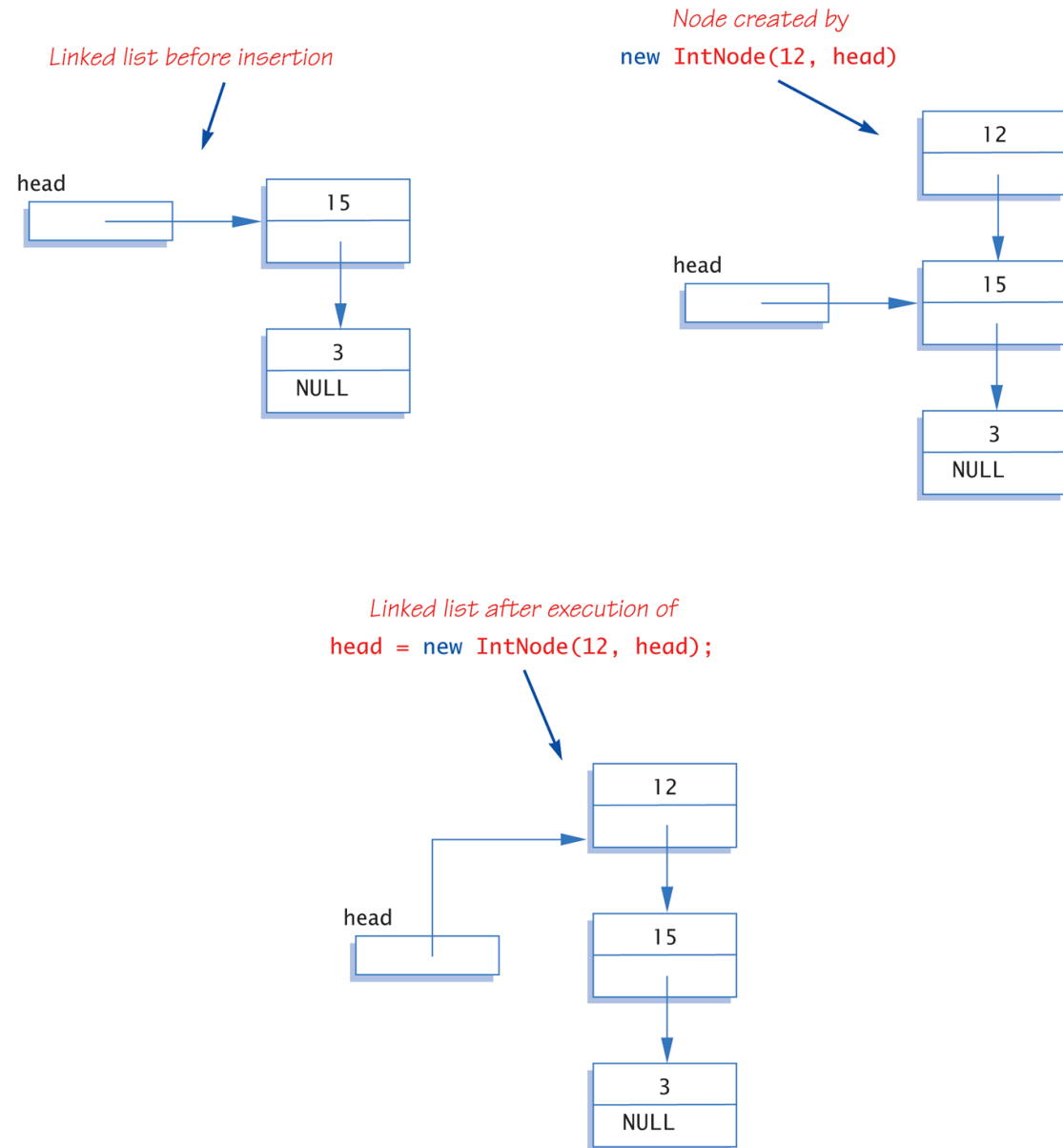
# Create 1<sup>st</sup> Node

- `IntNodePtr head;`
  - Declares pointer variable *head*
- `head = new IntNode;`
  - Dynamically allocates new node
  - Our 1<sup>st</sup> 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

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





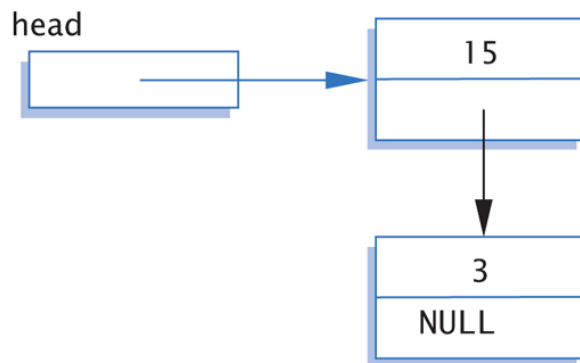
# Lost Nodes Pitfall:

## Display 17.5 Lost Nodes

**Display 17.5** Lost Nodes

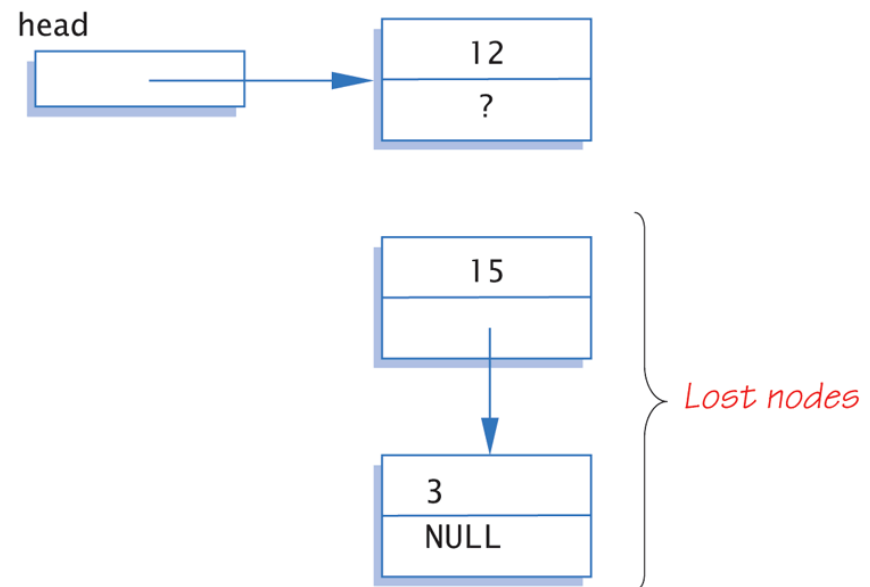
---

*Linked list before insertion*



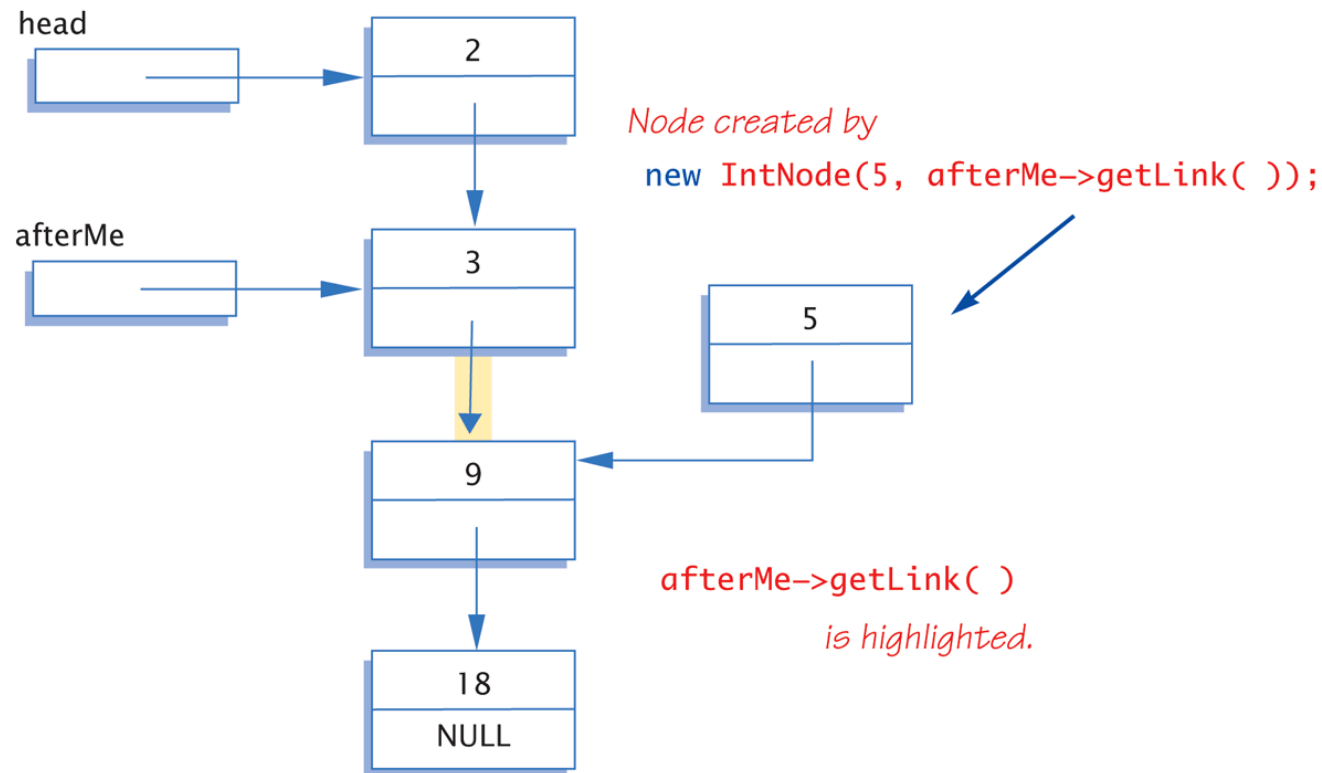
*Situation after executing*

```
head = new IntNode;  
head->setData(theData);
```

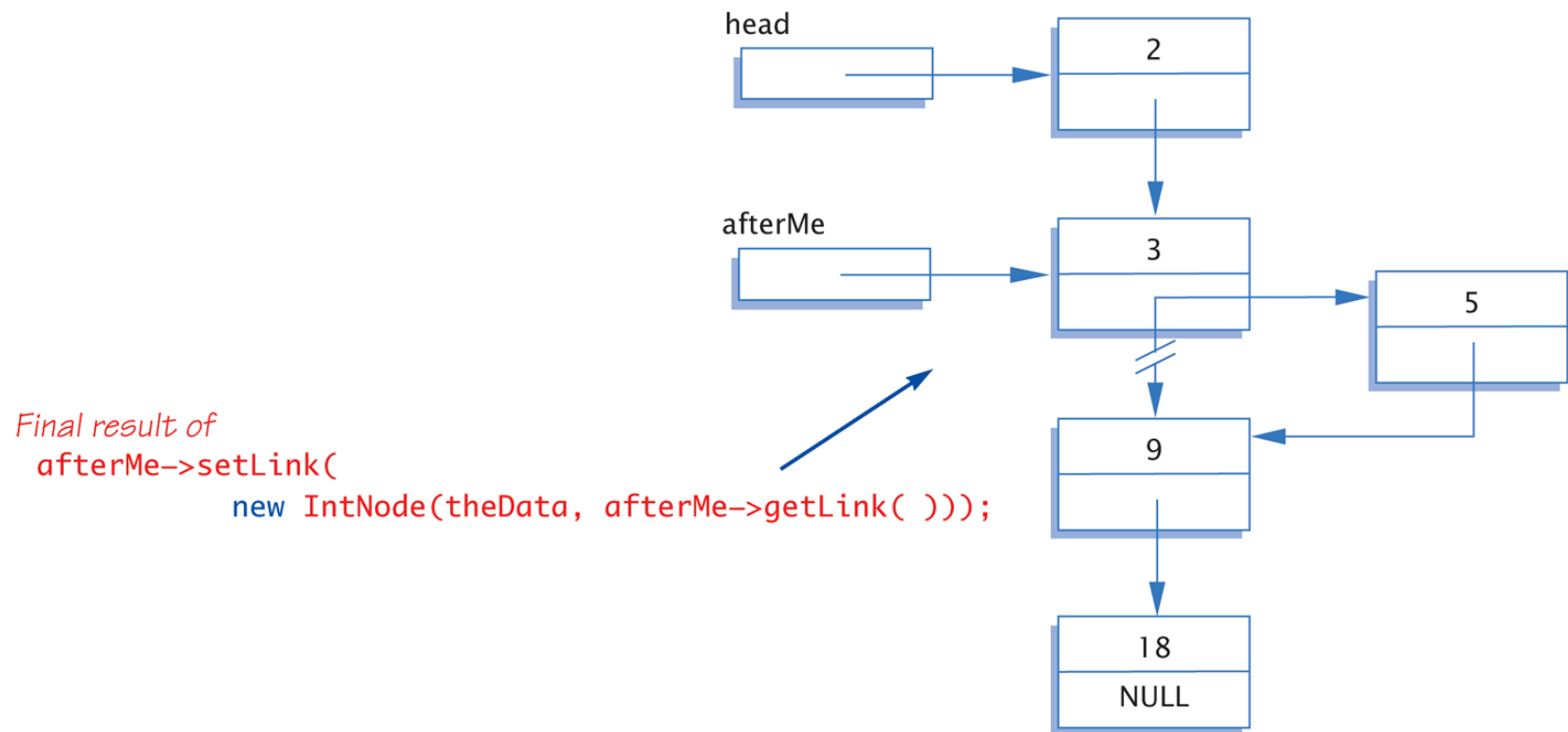


# Display 17.6 Inserting in the Middle of a Linked List (1 of 2)

Display 17.6 Inserting in the Middle of a Linked List



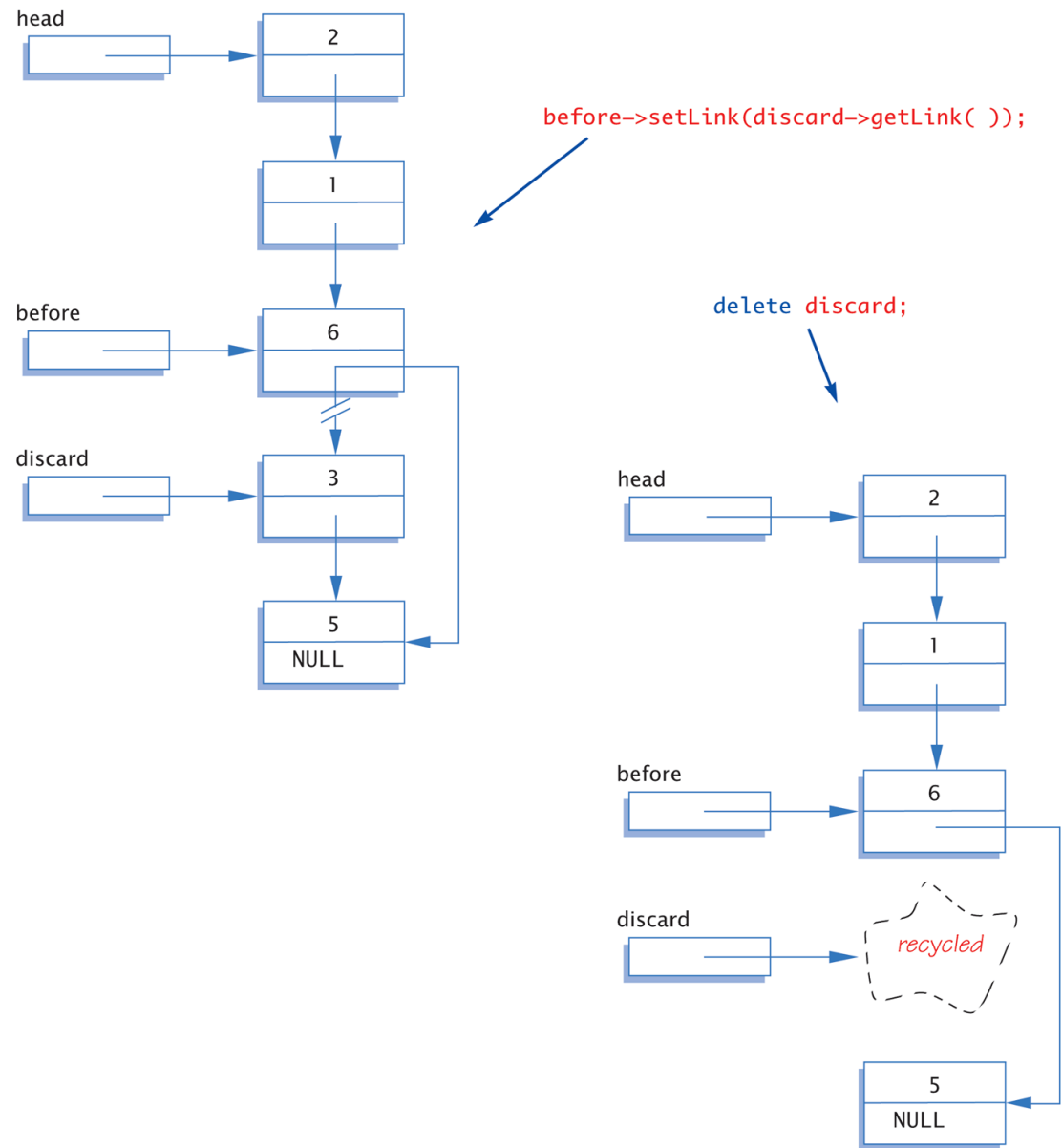
# Display 17.6 Inserting in the Middle of a Linked List (2 of 2)



# Display 17.7

## 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 or 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;
```

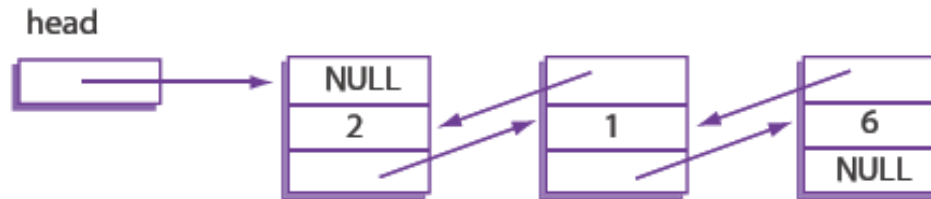
- Must make "special" case for empty list
  - Not done here

# Doubly Linked Lists

- What we just 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 search the list to find the node before the one we want to remove

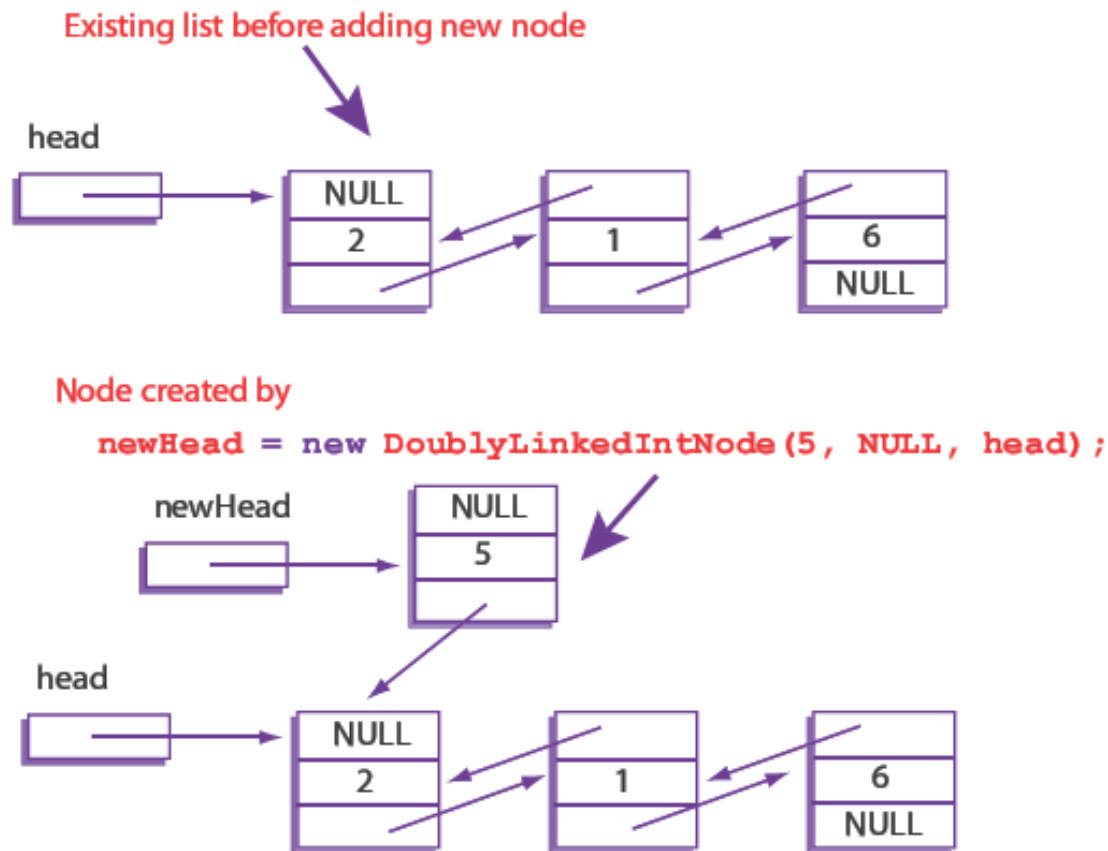


# Doubly Linked Lists



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

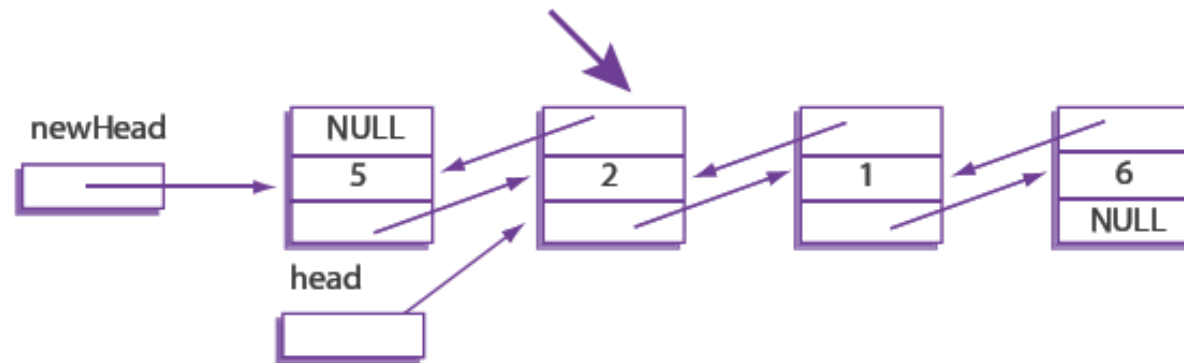
# Adding a Node to the Front of a Doubly Linked List (1 of 2)



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

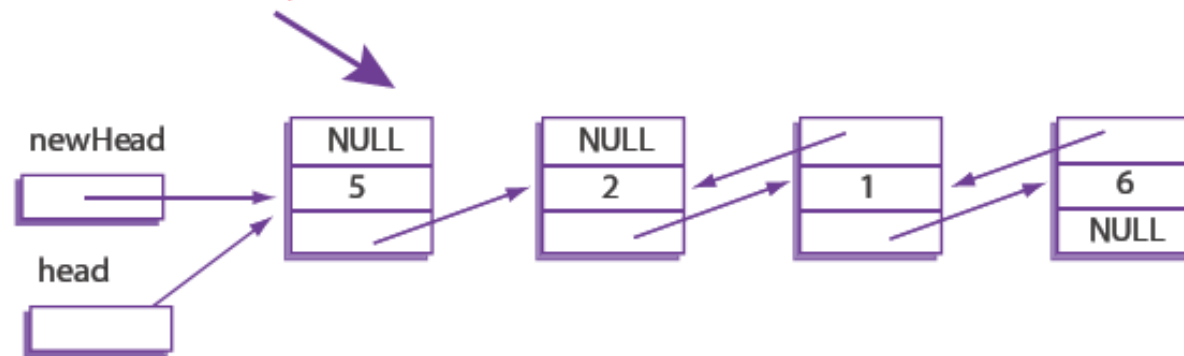
Set the previous link of the original head node

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



Set head to newHead

```
head = newHead;
```

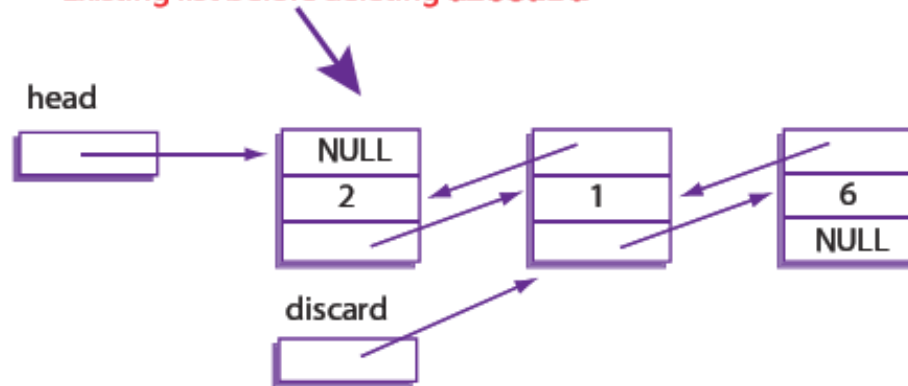


# Deleting a Node from a Doubly Linked List

- 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`

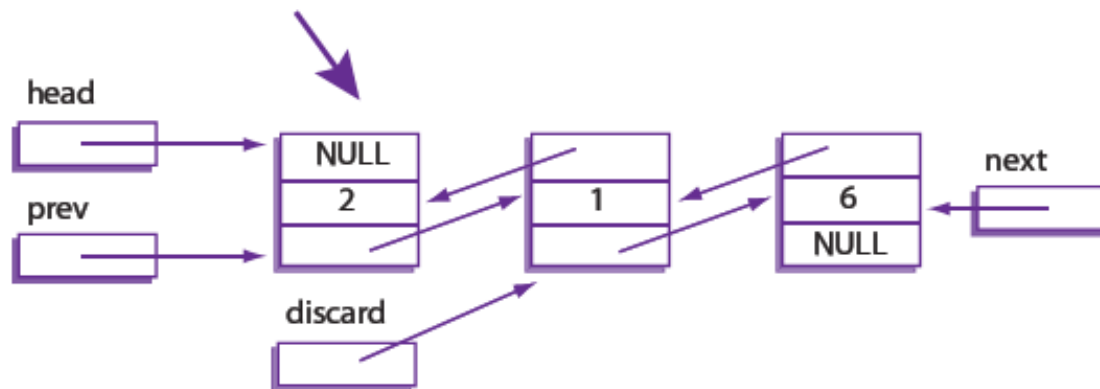
# Deleting a Node from a Doubly Linked List (1 of 2)

Existing list before deleting discard



Set pointers to the previous and next nodes

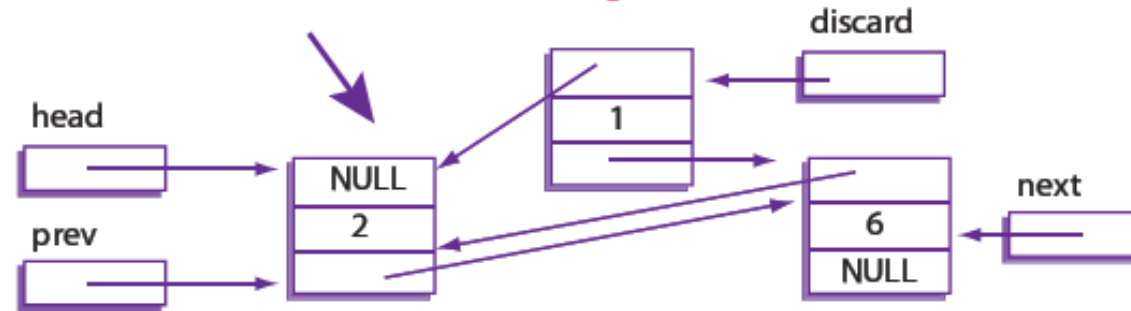
```
DoublyLinkedListNodePtr prev = discard->getPreviousLink( );  
DoublyLinkedListNodePtr next = discard->getNextLink( );
```



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

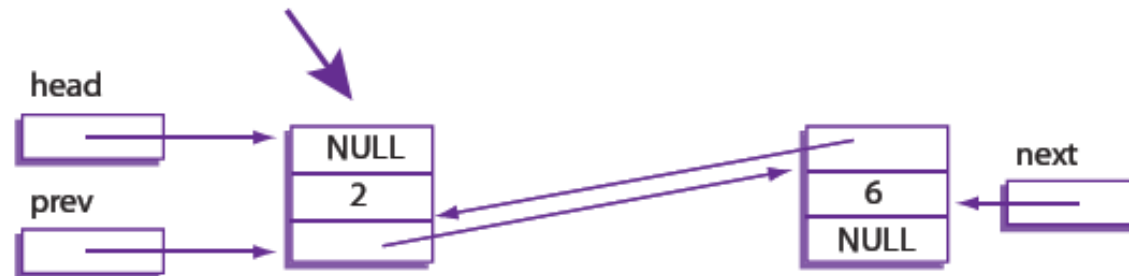
Bypass discard

```
prev->setNextLink(next);  
next->setPreviousLink(prev);
```



Delete discard

```
delete discard;
```



# Stacks

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

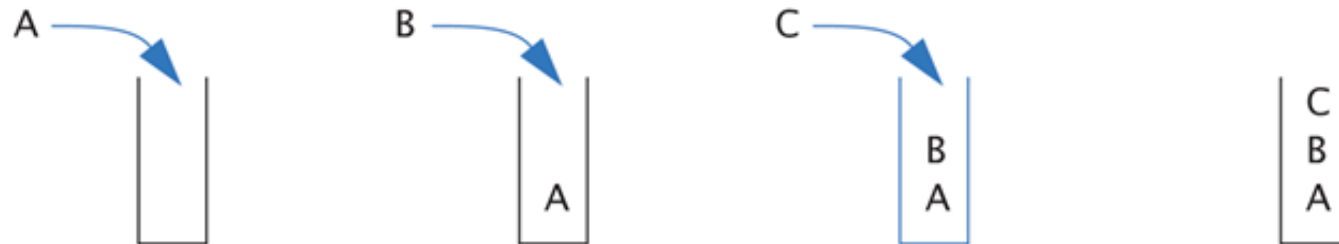
# A Stack—Graphic:

## Display 17.12 A Stack

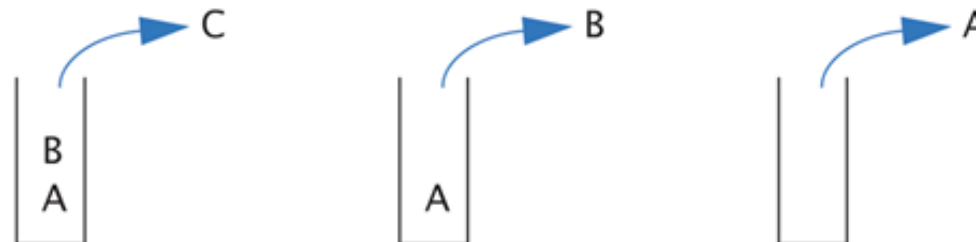
### A Stack

---

*pushing*



*popping*





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

## Interface File for a Stack Template Class

---

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

5  namespace StackSavitch
6  {
7      template<class T>
8      class Node
9      {
10     public:
11         Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
12         Node<T>* getLink( ) const { return link; }
13         const T getData( ) const { return data; }
14         void setData(const T& theData) { data = theData; }
15         void setLink(Node<T>* pointer) { link = pointer; }
16     private:
17         T data;
18         Node<T> *link;
19     };
```

*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

---

```
20     template<class T>
21     class Stack
22     {
23     public:
24         Stack();
25         //Initializes the object to an empty stack.
26         Stack(const Stack<T>& aStack); ← Copy constructor
27         Stack<T>& operator =(const Stack<T>& rightSide);
28         virtual ~Stack(); ← The destructor destroys the stack
                               and returns all the memory to the
                               freestore.
29         void push(T stackFrame);
30         //Postcondition: stackFrame has been added to the stack.
31         T pop();
32         //Precondition: The stack is not empty.
33         //Returns the top stack frame and removes that top
34         //stack frame from the stack.
35         bool isEmpty() const;
36         //Returns true if the stack is empty. Returns false otherwise.
37     private:
38         Node<T> *top;
39     };
40 } //StackSavitch
41 #endif //STACK_H
```

---

# Stack Template Class Driver:

## Display 17.18 Program Using the Stack Template Class (1 of 3)

### Program Using the Stack Template Class

---

```
1  //Program to demonstrate use of the Stack template class.
2  #include <iostream>
3  #include "stack.h"
4  #include "stack.cpp"
5  using std::cin;
6  using std::cout;
7  using std::endl;
8  using StackSavitch::Stack;
```

(continued)

# Stack Template Class Driver:

## Display 17.18 Program Using the Stack Template Class (2 of 3)

### Program Using the Stack Template Class

---

```
9  int main( )
10 {
11     char next, ans;
12     do
13     {
14         Stack<char> s;
15         cout << "Enter a line of text:\n";
16         cin.get(next);
17         while (next != '\n')
18         {
19             s.push(next);
20             cin.get(next);
21         }
```

# Stack Template Class Driver:

## Display 17.18 Program Using the Stack Template Class (3 of 3)

```
22         cout << "Written backward that is:\n";
23         while ( ! s.isEmpty() )
24             cout << s.pop();
25         cout << endl;

26         cout << "Again?(y/n): ";
27         cin >> ans;
28         cin.ignore(10000, '\n');
29     }while (ans != 'n' && ans != 'N');

30     return 0;
31 }
```

### SAMPLE DIALOGUE

Enter a line of text:  
**straw**  
Written backward that is:  
warts  
Again?(y/n): **y**  
Enter a line of text:  
**I love C++**  
Written backward that is:  
++C evol I  
Again?(y/n): **n**

*The ignore member of cin is discussed in Chapter 9. It discards input remaining on the line.*

# 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

---

```
1
2 //This is the header file queue.h. This is the interface for the class
3 //Queue, which is a template class for a queue of items of type T.
4 #ifndef QUEUE_H
5 #define QUEUE_H
6 namespace QueueSavitch
7 {
8     template<class T>
9     class Node
10    {
11    public:
12        Node(T theData, Node<T>* theLink) : data(theData), link(theLink){}
13        Node<T>* getLink( ) const { return link; }
14        const T getData( ) const { return data; }
15        void setData(const T& theData) { data = theData; }
16        void setLink(Node<T>* pointer) { link = pointer; }
17    private:
18        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)



# Display 17.20 Interface File for a Queue Template Class (2 of 3)

## Interface File for a Queue Template Class

---

```
19      Node<T> *link;  
20  };
```

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

```
21  template<class T>  
22  class Queue  
23  {  
24  public:  
25      Queue( );  
26      //Initializes the object to an empty queue.
```

```
27      Queue(const Queue<T>& aQueue);
```

*Copy constructor*

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

```
29      virtual ~Queue( );
```

*The destructor destroys the queue and returns all the memory to the freestore.*

```
30
```

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

```
31     void add(T item);
32     //Postcondition: item has been added to the back of the queue.

33     T remove( );
34     //Precondition: The queue is not empty.
35     //Returns the item at the front of the queue
36     //and removes that item from the queue.

37     bool isEmpty( ) const;
38     //Returns true if the queue is empty. Returns false otherwise.
39 private:
40     Node<T> *front; //Points to the head of a linked list.
41                     //Items are removed at the head
42     Node<T> *back; //Points to the node at the other end of the linked list.
43                     //Items are added at this end.
44 };

45 } //QueueSavitch

46 #endif //QUEUE_H
```

# Queue Template Class Driver: **Display 17.21** Program Using the Queue Template Class

```
12 do
13 {
14     Queue<char> q;
15     cout << "Enter a line of text:\n";
16     cin.get(next);
17     while (next != '\n')
18     {
19         q.add(next);
20         cin.get(next);
21     }

22     cout << "You entered:\n";
23     while ( ! q.isEmpty() )
24         cout << q.remove();
25     cout << endl;

26     cout << "Again?(y/n): ";
27     cin >> ans;
28     cin.ignore(10000, '\n');
29 }while (ans != 'n' && ans != 'N');
```

# 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

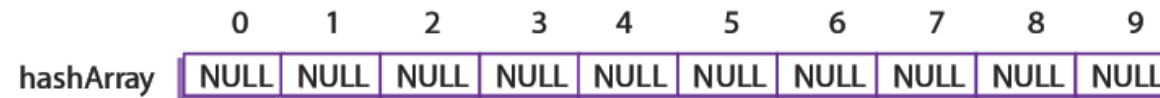
# Hash Table Idea

- 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

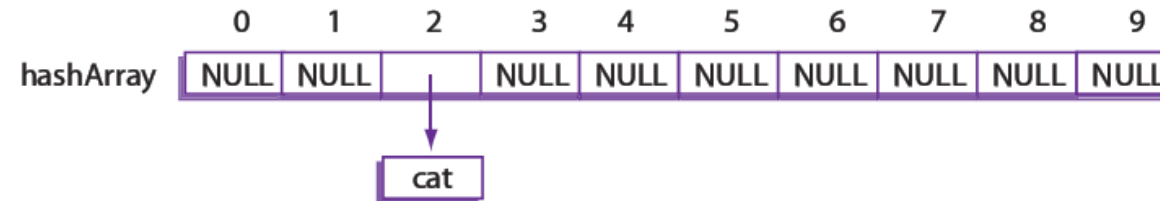
# Constructing a Hash Table

Existing hash table with 10 empty linked lists

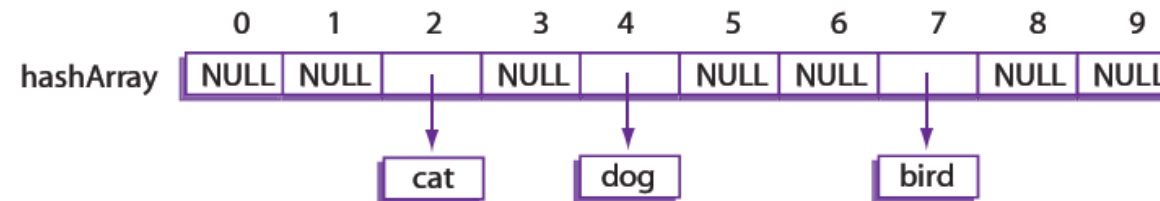
```
Node<string> *hashArray[10];  
for (int i=0; i<10; i++) hashArray[i] = NULL;
```



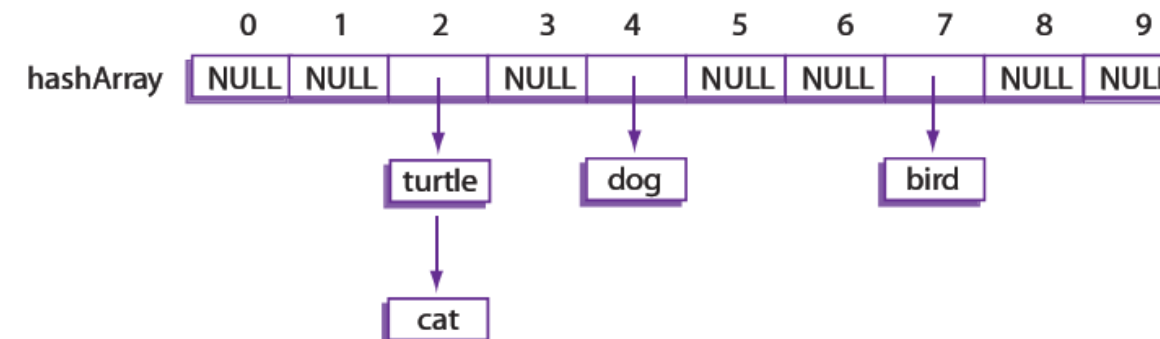
After adding "cat" with a hash of 2



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)

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

6  #include <string>
7  #include "listtools.h"
8  The library "listtools.h" is the linked list library
9  interface from Display 17.14.

10 using LinkedListSavitch::Node;
11 using std::string;

12 namespace HashTableSavitch
13 {
14     const int SIZE = 10;  // Maximum size of the hash table array
```



# Interface File for a HashTable Class

## (2 of 2)

```
13  class HashTable
14  {
15      public:
16          HashTable(); // Initialize empty hash table
17          // Normally a copy constructor and overloaded assignment
18          // operator would be included. They have been omitted
19          // to save space.
20          virtual ~HashTable(); // Destructor destroys hash table

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

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

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

# Implementation File for Hash Table Class (1 of 3)

```
1  // This is the implementation file hashtable.cpp.
2  // This is the implementation of the class HashTable.

3  #include <string>
4  #include "listtools.h"
5  #include "hashtable.h"

6  using LinkedListSavitch::Node;
7  using LinkedListSavitch::search;
8  using LinkedListSavitch::headInsert;
9  using std::string;

10 namespace HashTableSavitch
11 {
12     HashTable::HashTable()
13     {
14         for (int i = 0; i < SIZE; i++)
15         {
16             hashArray[i] = NULL;
17         }
18 }
```

# Implementation File for Hash Table Class (2 of 3)

```
19     HashTable::~~HashTable()
20     {
21         for (int i=0; i<SIZE; i++)
22         {
23             Node<string> *next = hashArray[i];
24             while (next != NULL)
25             {
26                 Node<string> *discard = next;
27                 next = next->getLink( );
28                 delete discard;
29             }
30         }
31     }

32     int HashTable::computeHash(string s)
33     {
34         int hash = 0;
35         for (int i = 0; i < s.length( ); i++)
36         {
37             hash = hash + s[i];
38         }
39         return hash % SIZE;
40     }
```

# Implementation File for Hash Table Class (3 of 3)

```
41     void HashTable::put(string s)
42     {
43         int hash = computeHash(s);
44         if (search(hashArray[hash], s)==NULL)
45         {
46             // Only add the target if it's not in the list
47             headInsert(hashArray[hash], s);
48         }
49     }
50 } // HashTableSavitch
```

# Hash Table Demonstration

```
1  // Program to demonstrate use of the HashTable class
2  #include <string>
3  #include <iostream>
4  #include "hashtable.h"
5  #include "listtools.cpp"
6  #include "hashtable.cpp"
7  using std::string;
8  using std::cout;
9  using std::endl;
10 using HashTableSavitch::HashTable;

11 int main()
12 {
13     HashTable h;
14     cout << "Adding dog, cat, turtle, bird" << endl;
15     h.put("dog");
16     h.put("cat");
17     h.put("turtle");
18     h.put("bird");
19     cout << "Contains dog? " << h.containsString("dog") << endl;
20     cout << "Contains cat? " << h.containsString("cat") << endl;
21     cout << "Contains turtle? " << h.containsString("turtle") << endl;
22     cout << "Contains bird? " << h.containsString("bird") << endl;
23     cout << "Contains fish? " << h.containsString("fish") << endl;
24     cout << "Contains cow? " << h.containsString("cow") << endl;
25     return 0;
26 }
```

## SAMPLE DIALOGUE

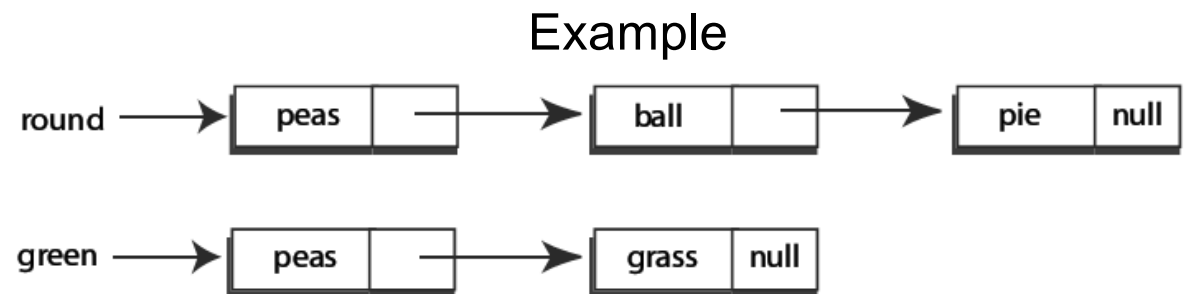
Adding dog, cat, turtle, bird  
Contains dog? 1  
Contains cat? 1  
Contains turtle? 1  
Contains bird? 1  
Contains fish? 0  
Contains cow? 0

# 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

# Set Template Class

- A set is a collection of elements in which no element occurs more than once
- We can implement a simple set that uses a linked list to store the items in the set
- Fundamental set operations we will support:
  - Add
  - Contains
  - Union
  - Intersection



# Interface File for a Set Template Class (1 of 2)

```
1 // This is the header file set.h. This is the interface
2 // for the class Set, which is a class for a generic set.
3 #ifndef SET_H
4 #define SET_H
5 #include "listtools.h"
6 // "listtools.h" is the linked list library interface from Display 17.14.
7 using LinkedListSavitch::Node;

8 namespace SetSavitch
9 {
10     template<class T>
11     class Set
12     {
13     public:
14         Set() { head = NULL; } // Initialize empty set

15         // Normally a copy constructor and overloaded assignment
16         // operator would be included. They have been omitted
17         // to save space.

18         virtual ~Set(); // Destructor destroys set
```



# Interface File for a Set Template Class (2 of 2)

```
18     bool contains(T target) const;
19     // Returns true if target is in the set, false otherwise

20     void add(T item);
21     // Adds a new element to the set

22     void output();
23     // Outputs the set to the console

24     Set<T>* setUnion(const Set<T>& otherSet);
25     // Union calling object's set with otherSet
26     // and return a pointer to the new set

27     Set<T>* setIntersection(const Set<T>& otherSet);
28     // Intersect calling object's set with otherSet
29     // and return a pointer to the new set
30 private:
31     Node<T> *head;
32 }; // Set
33 } // SetSavitch
34 #endif // SET_H
```

# Implementation File for a Set Template Class (1 of 4)

```
1 // This is the implementation file set.cpp.
2 // This is the implementation of the class Set.
3 #include <iostream>
4 #include "listtools.h"
5 #include "set.h"
6 using std::cout;
7 using std::endl;
8 using LinkedListSavitch::Node;
9 using LinkedListSavitch::search;
10 using LinkedListSavitch::headInsert;
11 namespace SetSavitch
12 {
13     template<class T>
14     Set<T>::~~Set()
15     {
16         Node<T> *toDelete = head;
17         while (head != NULL)
18         {
19             head = head->getLink( );
20             delete toDelete;
21             toDelete = head;
22         }
23     }
```

# Implementation File for a Set Template Class (2 of 4)

```
24     template<class T>
25     bool Set<T>::contains(T target) const
26     {
27         Node<T>* result = search(head, target);
28         if (result == NULL)
29             return false;
30         else
31             return true;
32     }

33     void Set<T>::output()
34     {
35         Node<T> *iterator = head;
36         while (iterator != NULL)
37         {
38             cout << iterator->getData( ) << " ";
39             iterator = iterator->getLink( );
40         }
41         cout << endl;
42     }
```

# Implementation File for a Set Template Class (3 of 4)

```
43     template<class T>
44     void Set<T>::add(T item)
45     {
46         if (search(head, item)
47             ==NULL)
48         {
49             // Only add the target if
50             // it's not in the list
51             headInsert(head, item);
52         }
53     }
```

```
52     template<class T>
53     Set<T>* Set<T>::setUnion(const
54                               Set<T>& otherSet)
55     {
56         Set<T> *unionSet = new Set<T>();
57         Node<T>* iterator = head;
58         while (iterator != NULL)
59         {
60             unionSet->add(iterator->getData(
61             ));
62             iterator = iterator->getLink( );
63         }
64         iterator = otherSet.head;
65         while (iterator != NULL)
66         {
67             unionSet->add(iterator->getData(
68             ));
69             iterator = iterator->getLink( );
70         }
71         return unionSet;
72     }
```

# Implementation File for a Set Template Class (4 of 4)

```
70     template<class T>
71     Set<T>* Set<T>::setIntersection(const Set<T>& otherSet)
72     {
73         Set<T> *interSet = new Set<T>();
74         Node<T>* iterator = head;
75         while (iterator != NULL)
76         {
77             if (otherSet.contains(iterator->getData( )))
78             {
79                 interSet->add(iterator->getData( ));
80             }
81             iterator = iterator->getLink( );
82         }
83         return interSet;
84     }
85 } // SetSavitch
```

# Set Demonstration (1 of 3)

```
1 // Program to demonstrate use of the Set class
2 #include <iostream>
3 #include <string>
4 #include "set.h"
5 #include "listtools.cpp"
6 #include "set.cpp"
7 using std::cout;
8 using std::endl;
9 using std::string;
10 using namespace SetSavitch;
11 int main()
12 {
13     Set<string> round;    // Round things
14     Set<string> green;    // Green things
15
16     round.add("peas");    // Sample data for both sets
17     round.add("ball");
18     round.add("pie");
19     round.add("grapes");
20     green.add("peas");
21     green.add("grapes");
22     green.add("garden hose");
23     green.add("grass");
```

# Set Demonstration (2 of 3)

```
23     cout << "Contents of set round: ";
24     round.output();
25     cout << "Contents of set green: ";
26     green.output();

27     cout << "ball in set round? " <<
28           round.contains("ball") << endl;
29     cout << "ball in set green? " <<
30           green.contains("ball") << endl;

31     cout << "ball and peas in same set? ";
32     if ((round.contains("ball") && round.contains("peas")) ||
33         (green.contains("ball") && green.contains("peas")))
34         cout << "  true" << endl;
35     else
36         cout << "  false" << endl;

37     cout << "pie and grass in same set? ";
38     if ((round.contains("pie") && round.contains("grass")) ||
39         (green.contains("pie") && green.contains("grass")))
40         cout << "  true" << endl;
41     else
42         cout << "  false" << endl;
```

# Set Demonstration (3 of 3)

```
43     cout << "Union of green and round: " << endl;
44     Set<string> *unionset = round.setUnion(green);
45     unionset->output();
46     delete unionset;

47     cout << "Intersection of green and round: " << endl;
48     Set<string> *interiset = round.setIntersection(green);
49     interiset->output();
50     delete interiset;

51     return 0;
52 }
```

## SAMPLE DIALOGUE

Contents of set round: grapes pie ball peas

Contents of set green: grass garden hose grapes peas

ball in set round? 1

ball in set green? 0

ball and peas in same set? true

pie and grass in same set? false

Union of green and round:

garden hose grass peas ball pie grapes

Intersection of green and round:

peas grapes



# 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<sup>st</sup> 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

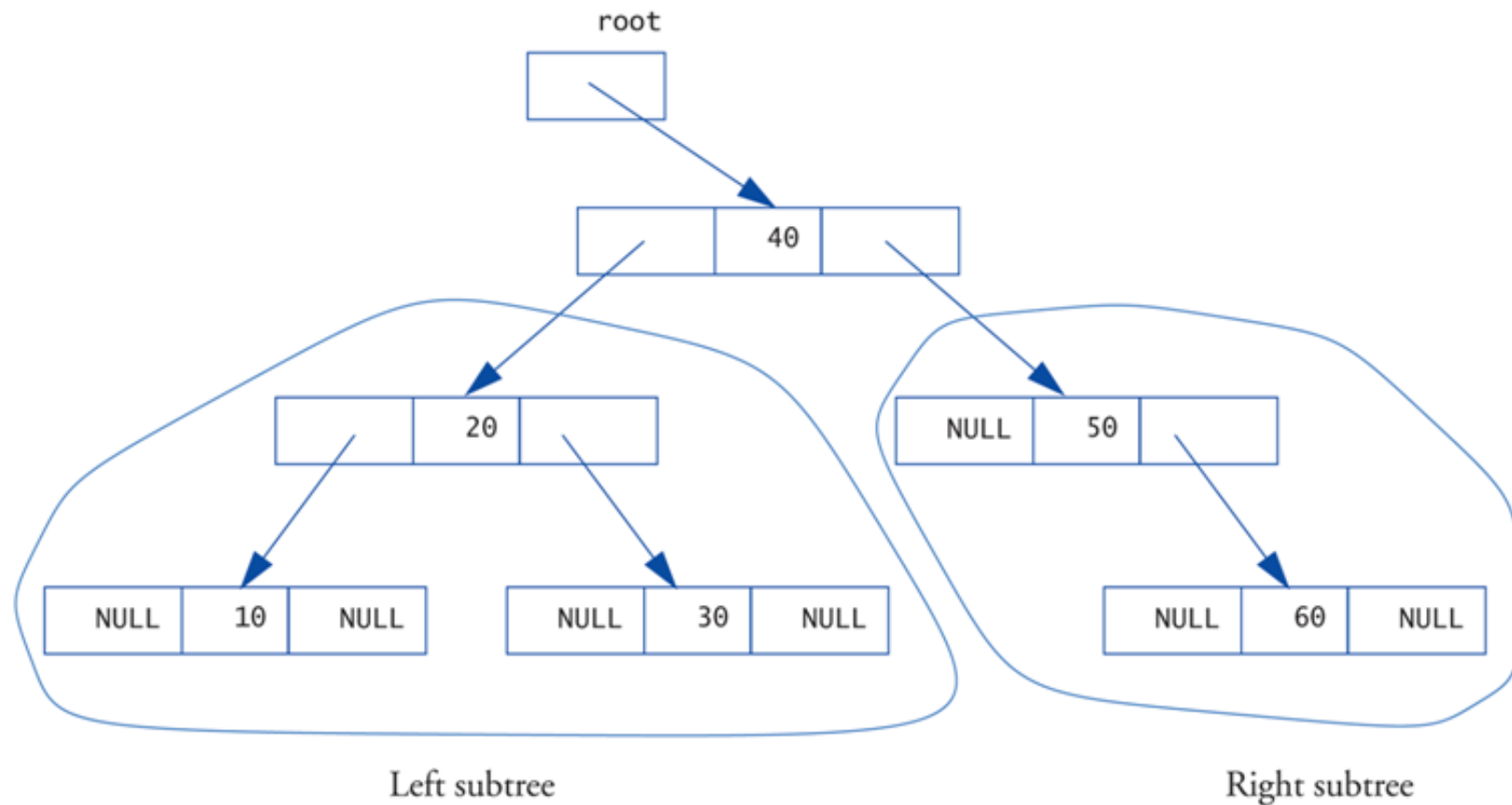
# Trees Introduction

- 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 tree 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:
    - Inorder → values "in order"
    - Preorder → "prefix" notation
    - Postorder → "postfix" notation

# 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