20.2 Self-Referential Classes

• Self-referential class
  – Contains a pointer member that points to an object of the same class type
  – Example
    ```cpp
    class Node {
    ...
    Node *nextPtr;
    };
    ```
  – Pointer data member `nextPtr` is a link
    • Can tie a `Node` to another `Node`
Fig. 20.1 | Two self-referential class objects linked together.
20.4 Linked Lists

• Linked list
  – Linear collection of self-referential class objects
    • Called nodes
    • Connected by pointer links
  – Accessed via a pointer to the first node
    • Subsequent nodes are accessed via previous node’s link
  – By convention, link in last node is set to null pointer 0
  – Additional nodes are dynamically allocated as necessary
20.4 Linked Lists (Cont.)

- Linked list (Cont.)
  - Advantages over arrays
    - Linked lists are dynamic
      - Length can increase or decrease as necessary
    - Efficient insertion of new elements into a sorted list
      - Existing list elements do not need to be moved
**Fig. 20.2** | A graphical representation of a list.
// Fig. 2Q3: Listnode.h
// Template ListNode class definition.
#ifndef LISTNODE_H
#define LISTNODE_H

// forward declaration of class List required to announce that class exists so it can be used in the friend declaration on at line 13
template<typename NODETYPE>
class List;

template<typename NODETYPE>
class ListNode{
    friend class List< NODETYPE >; // make List a friend

public:
    ListNode(const NODETYPE & ); // constructor
    NODETYPE getData() const; // return data in node

private:
    NODETYPE data; // data
    ListNode< NODETYPE > *nextPtr; // next node in list
}; // end class ListNode

// constructor
template<typename NODETYPE>
ListNode< NODETYPE >::ListNode( const NODETYPE &info )
    : data( info ), nextPtr( 0 )
{ // empty body
} // end ListNode constructor

Outline

Listnode.h
(1 of 2)

Declare class List< NODETYPE > as a friend

Member data stores a value of type parameter NODETYPE

Member nextPtr stores a pointer to the next ListNode object in the linked list
// Fig. 20.4: List.h
// Template List class definition.
#ifndef LIST_H
#define LIST_H

#include <iostream>
using std::cout;

#include "listnode.h" // ListNode class definition

template<typename NODETYPE>
class List
{
public:
    List(); // constructor
    ~List(); // destructor
    void insertAtFront(const NODETYPE &);
    void insertAtBack(const NODETYPE &);
    bool removeFromFront(NODETYPE &);
    bool removeFromBack(NODETYPE &);
    bool isEmpty() const;
    void print() const;

private:
    ListNode<NODETYPE> *firstPtr; // pointer to first ListNode in a List
    ListNode<NODETYPE> *lastPtr; // pointer to last ListNode in a List

    // utility function to allocate new node
    ListNode<NODETYPE> *getNewNode(const NODETYPE &);
}; // end class List
31 // default constructor
32 template< typename NODETYPE >
33 List< NODETYPE >::List()
34 : firstPtr( 0 ), lastPtr( 0 )
35 {
36     // empty body
37 } // end List constructor
38
39 // destructor
40 template< typename NODETYPE >
41 List< NODETYPE >::~List()
42 {
43     if ( isEmpty() )  // List is not empty
44     {
45         cout << "Destroying nodes ... \n";
46
47         ListNode< NODETYPE > *currentPtr = firstPtr;
48         ListNode< NODETYPE > *tempPtr;
49
50         while ( currentPtr != 0 )  // delete remaining nodes
51             {
52             tempPtr = currentPtr;
53             cout << tempPtr ->data << '\n';
54             currentPtr = currentPtr ->nextPtr;
55             delete tempPtr;
56         } // end while
57     } // end if
58
59     cout << "All nodes destroyed \n\n";
60 } // end List destructor
// end function removeFromBack

// is List empty?
template< typename NODETYPE >
bool List< NODETYPE >::isEmpty() const
{
    return firstPtr == 0;
} // end function isEmpty

// return pointer to newly allocated node
template< typename NODETYPE >
ListNode< NODETYPE > *List< NODETYPE >::getNewNode(
    const NODETYPE &value )
{
    return new ListNode< NODETYPE >( value );
} // end function getNewNode

// display contents of List
template< typename NODETYPE >
void List< NODETYPE >::print() const
{
    if ( isEmpty() ) // List is empty
    {
        cout << "The list is empty  
        return;
    } // end if

// Fig. 20.5: Fig20_05.cpp
// List class test program.
#include <iostream>
using std::cin;
using std::cout;
using std::endl;

#include <string>
using std::string;

#include "List.h" // List class definition

// function to test a List
template< typename T >
void testList( List< T >& listObject, const string &typeName )
{
    cout << "Testing a List of " << typeName << " values \n";
    instructions(); // display instructions

    int choice; // store user choice
    T value;    // store input value

    do // perform user-selected actions
    {
    cout << "? ";
    cin >> choice;
}
cout << "End list test \n\n";
} // end function testList

// display program instructions to user
void instructions()
{
    cout << "Enter one of the following:  \n"
        << "  1 to insert at beginning of list  \n"
        << "  2 to insert at end of list  \n"
        << "  3 to delete from beginning of list  \n"
        << "  4 to delete from end of list  \n"
        << "  5 to end list processing  \n";
} // end function instructions

int main()
{
    // test List of int values
    List<int> integerList;
testList( integerList, "integer" );

    // test List of double values
    List<double> doubleList;
testList( doubleList, "double" );
return 0;
} // end main
Fig. 20.6 | Operation \texttt{insertAtFront} represented graphically.
Fig. 20.7 | Operation `insertAtBack` represented graphically.
Fig. 20.8 | Operation `removeFromFront` represented graphically.
Fig. 20.9 | Operation **removeFromBack** represented graphically.
20.4 Linked Lists (Cont.)

• Linked list (Cont.)
  – Circular, singly linked list
    • Pointer in last node points back to first node
  – Doubly linked list
    • Each node has a link to next node and a link to previous node
    • Two “start pointers”
      – One to first node, one to last node
    • Allows traversals both forward and backward
  – Circular, doubly linked list
    • Forward link of last node points back to first node
    • Backward link of first node points to last node
Fig. 20.10 | Circular, singly linked list.
Fig. 20.11 | Doubly linked list.
Fig. 20.12 | Circular, doubly linked list.
Tree of life  
(mid 19th century)

http://www.ucmp.berkeley.edu/education/events/eukevol.html
Carl Woese

Early 16S rRNA tree

Woese’s new tree of life. The position and length of each branch is determined by comparisons of ribosomal RNA.
Interesting study: AIDS transmission

• http://www.pnas.org/content/99/22/14292

• First time phylogenetics was used in a criminal court case in the U.S (2002).
Goal

• **Input:**
  – *Data from a set of genes/species*

• **Output:**
  – *A phylogenetic tree that accurately characterizes the right lineages*
• *We infer trees because we don’t really know all the species, esp. ancestors represented by internal nodes.*

• *Also, tree finding is a hard problem: there are 34,459,425 unrooted trees for only ten taxa.*

• *Today, we’ll discuss simple approaches for phylogenetic tree construction that are fast under certain assumptions.*
Species tree

http://members.aol.com/darwinpage/trees.htm
Old example

Source:
Warren Ewens
U. of Penn
20.7 Trees

• Tree
  – Nonlinear, two-dimensional data structure
  – Tree nodes contain two or more links
    • Binary tree nodes contain exactly two links
20.7 Trees (Cont.)

• Tree (Cont.)
  – Terminology
    • Root node
      – First node in a tree
    • Child node
      – Node linked to by another node (its parent)
      – In a binary tree, there is a left child and a right child
    • Subtree
      – Tree defined by treating a child node as the root of its own tree
    • Siblings
      – Multiple children of a single parent node
    • Leaf node
      – Node with no children
Fig. 20.18 | A graphical representation of a binary tree.
20.7 Trees (Cont.)

• Tree (Cont.)
  – Binary search tree
    • Values in any left subtree are less than value in its parent node
    • Values in any right subtree are greater than value in its parent node
    • Can be recursively traversed in three ways
      – Inorder
        • Left subtree, then current node, then right subtree
      – Preorder
        • Current node, then left subtree, then right subtree
      – Postorder
        • Left subtree, then right subtree, then current node
Fig. 20.23 | A binary search tree.
20.7 Trees (Cont.)

• Tree (Cont.)
  – Applications of binary search trees
    • Duplicate elimination
      – Inserting duplicate will follow same path as original
      – Duplicate can be discarded when compared with original
    • Searching (in a balanced binary search tree)
      – Has $O(\log n)$ runtime
        • Each comparison of a node to search key eliminates half the nodes
        • Maximum of $\log_2 n$ comparisons are required
    • Sorting (binary tree sort)
      – Inorder traversal of a binary search tree results in processing the values in ascending order