///\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
///Unsorted List Implemented as a linked structure in C++  
///\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
/// Header file for a list class  
///---------------------------------------------------------------  
/// File: UnsortedListArray.h  
/// Purpose: Header file for a demonstration of an unsorted list  
/// implemented as a linked structure.  
///---------------------------------------------------------------  
  
#include <iostream>  
using namespace std;  
  
/// Define a structure to use as the list item  
struct ListItem  
{  
 int key;  
 float theData;  
 ListItem \*next;  
};  
  
class UnsortedListArray  
{  
 private:  
 ListItem \*head; // Pointer to head of the list  
  
 public:  
 UnsortedListArray(); // Class constructor  
 ~UnsortedListArray(); // Class destuctor  
 void ClearList(); // Remove all items from the list  
 bool Insert(int key, float f);// Add an item to the list  
 bool Delete(int key); // Delete an item from the list  
 bool Search(int key, float \*retVal); // Search for an item in the list  
 int ListLength(); // Return number of items in list  
 bool isEmpty(); // Return true if list is empty  
 bool isFull(); // Return true if list is full  
 void PrintList(); // Print all items in the list  
};  
  
  
///\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
///Implementation (.cpp) file for a list class  
///---------------------------------------------------------------  
/// File: UnsortedListArray.cpp  
/// Purpose: Implementation file for a demonstration of an unsorted  
/// list implemented as a linked structure.  
/// Programming Language: C++  
/// Author: Dr. Rick Coleman  
/// Date: January 8, 2002  
///---------------------------------------------------------------  
///#include "UnsortedListArray.h"  
  
///--------------------------------------------  
/// Function: UnsortedListArray()  
/// Purpose: Class constructor  
/// Returns: void  
///--------------------------------------------  
UnsortedListArray::UnsortedListArray()  
{  
 head = NULL;  
}  
  
///--------------------------------------------  
/// Function: UnsortedListArray()  
/// Purpose: Class destructor  
/// Returns: void  
///--------------------------------------------  
UnsortedListArray::~UnsortedListArray()  
{  
 /// Clear the list to free any memory being used  
 ClearList();  
}  
  
///--------------------------------------------  
/// Function: ClearList()  
/// Purpose: Remove all items from the list  
/// Returns: void  
///--------------------------------------------  
void UnsortedListArray::ClearList()  
{  
 ListItem \*temp;  
  
 if(!isEmpty())  
 {  
 temp = head;  
  
 /// Scan list and free all nodes  
 while(head != NULL)  
 {  
 temp = head;  
 head = head->next;  
 delete temp;  
 }  
 }  
}  
  
///--------------------------------------------  
/// Function: Insert()  
/// Purpose: Insert an item into the list at  
/// the end of the list. See alternate  
/// code below for insert at the beginning  
/// of the list.  
/// Returns: true if insertion was successful  
/// or false if the insertion failed.  
///--------------------------------------------  
bool UnsortedListArray::Insert(int key, float f)  
{  
 ListItem \*temp, \*newNode;  
  
 /// Create a new node and insert the data  
 newNode = new ListItem();  
 /// Check to see if memory allocation failed  
 if(newNode == NULL) return false;  
 /// If all OK then insert the data  
 newNode->key = key;  
 newNode->theData = f;  
 newNode->next = NULL; /// Very import to init this to NULL  
  
 /// Check to see if the list is empty  
 if(isEmpty())  
 {  
 /// Insert new node as first in the list  
 head = newNode;  
 }  
 else  
 {  
 /// Find end of the list  
 temp = head;  
 while(temp->next != NULL)  
 temp = temp->next;  
  
 /// Add this node to the end of the list  
 temp->next = newNode;  
  
 /\*\* Alternate insertion code:  
 \* Since this is an unsorted list an alternate  
 \* insertion approach is to add the new node at  
 \* the head of the list. To do this replace all  
 \* of the code in the else part of this function  
 \* with the following:  
 \*  
 \* newNode->next = head;  
 \* head = newNode;  
 \*/  
 }  
 return true; /// Signal successful insertion  
}  
  
///--------------------------------------------  
/// Function: Delete()  
/// Purpose: Delete an item from the list.  
/// Returns: true if deletion was successful  
/// or false if the deletion failed.  
///--------------------------------------------  
bool UnsortedListArray::Delete(int key)  
{  
 ListItem \*temp, \*back;  
  
 /// Check for empty list  
 if(isEmpty()) return false;  
  
 /// Search the list for the item to delete  
 temp = head;  
 back = NULL;  
 /// The order of the two conditionals in the while()  
 /// look is VERY important. You want to check first  
 /// to see if temp is NULL before trying to reference  
 /// the memory temp is pointing to. If temp is NULL  
 /// then, because this is a && (AND) condition the  
 /// second condition will never be tested. Testing  
 /// the second condition when temp==NULL will result  
 /// in a crash and burn.  
 while((temp != NULL) && (key != temp->key))  
 {  
 back = temp;  
 temp = temp->next;  
 }  
  
 /// Check to see if the item was found  
 if(temp == NULL) return false; // Not found so return false  
 else if(back == NULL) /// Check to see if item is first in list  
 {  
 head = head->next;  
 delete temp; /// Dispose of the node removed from the list  
 }  
 else /// Delete node elsewhere in the list  
 {  
 back->next = temp->next;  
 delete temp; /// Dispose of the node removed from the list  
 }  
 return true; /// Signal successful deletion  
}  
  
  
///--------------------------------------------  
/// Function: Search()  
/// Purpose: Search for an item by key and copy  
/// the value into the variable pointed to  
/// by \*retVal.  
/// Returns: true if search was successful  
/// or false if the search failed.  
///--------------------------------------------  
bool UnsortedListArray::Search(int key, float \*retVal)  
{  
 ListItem \*temp;  
  
 temp = head;  
 /// See note on the order of the conditional in this  
 /// while() loop in Delete() function above.  
 while((temp != NULL) && (key != temp->key))  
 {  
 temp = temp->next;  
 }  
  
 /// If item not found or list is empty return false  
 if(temp == NULL) return false;  
 else  
 \*retVal = temp->theData; // Copy the data  
 return true; /// Signal successful search  
}  
  
///--------------------------------------------  
/// Function: ListLength()  
/// Purpose: Return the number of items in the  
/// list.  
/// Returns: Number of items in list.  
///--------------------------------------------  
int UnsortedListArray::ListLength()  
{  
 ListItem \*temp;  
 int count = 0;  
  
 temp = head;  
 while(temp != NULL)  
 {  
 temp = temp->next;  
 count++;  
 }  
 return count;  
 /// An alternate way to do this is to maintain  
 /// a static variable at the top of this source  
 /// code, e.g. int count. This can be incremented  
 /// each time a node is added and decremented each  
 /// time a node is deleted.  
}  
  
///--------------------------------------------  
/// Function: isEmpty()  
/// Purpose: Return true if the list is empty  
/// Returns: true if empty, otherwise false  
///--------------------------------------------  
bool UnsortedListArray::isEmpty()  
{  
 return (head == NULL);  
}  
  
///--------------------------------------------  
/// Function: isFull()  
/// Purpose: Return true if the list is full  
/// Returns: true if full, otherwise false  
/// Note: In theory a linked list cannot be  
/// full (unless you run out of memory) so  
/// this function defaults to returning false.  
///--------------------------------------------  
bool UnsortedListArray::isFull()  
{  
 return false;  
}  
  
  
///--------------------------------------------  
/// Function: PrintList()  
/// Purpose: Print all items in the list with  
/// their priority.  
/// Returns: void  
///--------------------------------------------  
void UnsortedListArray::PrintList()  
{  
 ListItem \*temp;  
  
 cout << "\n\nItems in the List\n";  
 cout << "-----------------------------------------------------------\n";  
 cout << "Key\t\tData\n";  
 cout << "-----------------------------------------------------------\n";  
  
 if(head == NULL) /// Report no items in the list  
 {  
 cout << "\t List is currently empty.\n";  
 }  
 else  
 {  
 temp = head;  
 while(temp != NULL)  
 {  
 cout << temp->key << "\t\t" << temp->theData << "\n";  
 temp=temp->next;  
 }  
 }  
 cout << "-----------------------------------------------------------\n\n";  
}  
///\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
///Main file used to test the list  
///---------------------------------------------------------------  
/// File: ListMain.cpp  
/// Purpose: Main file with tests for a demonstration of an unsorted  
/// list implemented as a linked structure.  
/// Programming Language: C++  
///---------------------------------------------------------------  
///#include "UnsortedListArray.h"  
  
int main()  
{  
 float f;  
 UnsortedListArray \*theList;  
  
 cout << "Simple List Demonstration\n";  
 cout << "Create a list and add a few tasks to the list";  
  
 theList = new UnsortedListArray(); /// Instantiate a list object  
  
 theList->Insert(5, 3.1f); /// Note: The argument to the funtion should be a float  
 theList->Insert(1, 5.6f); /// A constant real number like 3.1 is interpreted as  
 theList->Insert(3, 8.3f); /// a double unless it is explicitly defined as a float  
 theList->Insert(2, 7.4f); /// by adding an 'f' to the end of the number.  
 theList->Insert(4, 2.5f);  
  
 /// Show what is in the list  
 theList->PrintList();  
  
 /// Test the list length function  
 cout << "\nList now contains " << theList->ListLength() << "items.\n\n";  
  
 /// Test delete function  
 cout << "Testing delete of last item in list.\n";  
 theList->Delete(4);  
 theList->PrintList();  
  
 /// Test delete function  
 cout << "Testing delete of first item in list.\n";  
 theList->Delete(5);  
 theList->PrintList();  
  
 /// Test delete function  
 cout << "Testing delete of a middle item in list.\n";  
 theList->Delete(3);  
 theList->PrintList();  
  
 /// Test delete function with a known failure argument  
 cout << "Testing failure in delete function.\n";  
 if(theList->Delete(4))  
 cout << "Oops! Should not have been able to delete.\n";  
 else  
 cout << "Unable to locate item to delete.\n";  
  
 /// Test search (known failure)  
 cout << "Testing Search function. Search for key 3\n";  
 if(theList->Search(3, &f))  
 cout << "Search result: theData = %f\n", f;  
 else  
 cout << "Search result: Unable to locate item in list\n";  
  
 /// Test search (known success)  
 cout << "Testing Search function. Search for key 2\n";  
 if(theList->Search(2, &f))  
 cout << "Search result: the Data = " << f << "\n";  
 else  
 cout << "Search result: Unable to locate item in list\n";  
  
 cout << "\n\nEnd list demonstration...";  
  
 return 0;  
}