1. *Single-ended list*.
2. Abstract Data Types (ADTs: stack using *single-ended list* and queue using *double-ended list*).
3. Specialized list (*doubly linked list*).

1. Single-ended list

The single-ended list is a structure that characterizes quickly inserting and deleting a data. From exercise no. 3 we know data structure such as an array and a sorted array. Both structures have some disadvantages. In the case of an unsorted arrays, the search is slow, while in sorted arrays slow is inserting new items. It is also time-consuming for deleting an item in both types of arrays.

In the single-ended list, data is inside the object **Link**. Since there are usually many similar objects (**link**s) in the list (list), it is reasonable to create a separate class for them. Each object of type Link contains (as a class field) pointer (**pNex**) to the next object of type Link.

Example 1 (C++; single-ended list = (singly) linked list) First approach to data structure: linked list.

[linkList.cpp]

```
//linkList.cpp
//demonstrates linked list
#include <iostream>
using namespace std;

class Link
{
    public:
        int iData;                  //data item
        double dData;               //data item
        Link* pNext;                //ptr to next link in list

        Link(int id, double dd) :   //constructor
        iData(id), dData(dd), pNext(NULL)
        {  }

        void displayLink()          //display ourself {22, 2.99}
        {
        cout << "{" << iData << ", " << dData << "} ";
        }

};  //end class Link
//--------------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;                       //ptr to first link on list

    public:
        LinkList()  : pFirst(NULL)          //constructor
        {  }                                //(no links on list yet)
```

```
        bool isEmpty()                      //true if list is empty
        { return pFirst==NULL; }

        //insert at start of list
        void insertFirst(int id, double dd)
        {                                   //make new link
            Link* pNewLink = new Link(id, dd);
            pNewLink->pNext = pFirst;       //newLink-->old first
            pFirst = pNewLink;              //first-->newLink
        }

        Link* getFirst()                    //return first link
        { return pFirst; }

        void removeFirst()                  //delete first link
        {                                   //(assumes list not empty)
            Link* pTemp = pFirst;           //save first
            pFirst = pFirst->pNext;         //unlink it: first-->old next
            delete pTemp;                   //delete old first
        }

        void displayList()
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;        //start at beginning of list
            while(pCurrent != NULL)         //until end of list,
            {
                pCurrent->displayLink();//print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }

};  //end class LinkList
//-----------------------------------------------------------------
int main()
{
    LinkList theList;                       //make new list

    theList.insertFirst(22, 2.99);          //insert four items
    theList.insertFirst(44, 4.99);
    theList.insertFirst(66, 6.99);
    theList.insertFirst(88, 8.99);

    theList.displayList();                  //display list

    while( !theList.isEmpty() )             //until it's empty,
    {
        Link* pTemp = theList.getFirst();   //get first link
                                            //display its key
        cout << "Removing link with key " << pTemp->iData << endl;
        theList.removeFirst();              //remove it
    }
    theList.displayList();                  //display empty list
    return 0;
}  //end main()
```

Finding and removing the specific objects (Links)

The next example will include two additional methods. These methods can be applied to:

- search list item that contains a variable with a specific value,
- deleting the item of the list containing a variable with the specified value.

Example 2 (C++; (singly) linked list) Second version of data structure: linked list.

[linkList2.cpp]

```
//linkList2.cpp
//demonstrates linked list
#include <iostream>
using namespace std;

class Link
{
    public:
        int iData;                  //data item (key)
        double dData;               //data item
        Link* pNext;                //next link in list

        Link(int id, double dd) :   //constructor
        iData(id), dData(dd), pNext(NULL)
        {  }

        void displayLink()          //display ourself: {22, 2.99}
        {
            cout << "{" << iData << ", " << dData << "} ";
        }
};  //end class Link
//-----------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;               //ptr to first link on list

    public:
        LinkList()  : pFirst(NULL)  //constructor
        {  }                        //(no links on list yet)

        ~LinkList()                 //destructor (deletes links)
        {
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                Link* pOldCur = pCurrent;    //save current link
                pCurrent = pCurrent->pNext;  //move to next link
                delete pOldCur;              //delete old current
            }
        }

        void insertFirst(int id, double dd)
        {                                //make new link
            Link* pNewLink = new Link(id, dd);
            pNewLink->pNext = pFirst;    //it points to old first link
            pFirst = pNewLink;           //now first points to this
        }
```

```
        Link* find(int key)             //find link with given key
        {                               //(assumes non-empty list)
            Link* pCurrent = pFirst;            //start at 'first'
            while(pCurrent->iData != key)       //while no match,
            {
                if(pCurrent->pNext == NULL)      //if end of list,
                    return NULL;                 //didn't find it
                else                             //not end of list,
                    pCurrent = pCurrent->pNext; //go to next link
            }
            return pCurrent;                     //found it
        }

        bool remove(int key)                //remove link with given key
        {                                   //(assumes non-empty list)
            Link* pCurrent = pFirst;        //search for link
            Link* pPrevious = pFirst;
            while(pCurrent->iData != key)
            {
                if(pCurrent->pNext == NULL)
                    return false;           //didn't find it
                else
                {
                    pPrevious = pCurrent;  //go to next link
                    pCurrent = pCurrent->pNext;
                }
            }                               //found it
            if(pCurrent == pFirst)          //if first link,
                pFirst = pFirst->pNext;     //change first
            else                                    //otherwise,
                pPrevious->pNext = pCurrent->pNext; //bypass it
            delete pCurrent;                //delete link
            return true;                    //successful removal
        }

        void displayList()                  //display the list
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;        //start at beginning of list
            while(pCurrent != NULL)         //until end of list,
            {
                pCurrent->displayLink();     //print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }
};  //end class LinkList
//-----------------------------------------------------------------
int main()
{
    LinkList theList;                       //make list

    theList.insertFirst(22, 2.99);          //insert 4 items
    theList.insertFirst(44, 4.99);
    theList.insertFirst(66, 6.99);
    theList.insertFirst(88, 8.99);

    theList.displayList();                  //display list

    int findKey = 44;                       //find item
    Link* pFind = theList.find(findKey);
```

```
    if( pFind != NULL)
        cout << "Found link with key " << pFind->iData << endl;
    else
        cout << "Can't find link" << endl;

    int remKey = 66;                        //remove item
    bool remOK = theList.remove(remKey);
    if( remOK )
        cout << "Removed link with key " << remKey << endl;
    else
        cout << "Can't remove link" << endl;

    theList.displayList();                  //display list
    return 0;
}  //end main()
```

Member function find()

The **find()** works like displayList() from program linkList.cpp. Initially pCurrent pointer is set to point to the same object as pFirst. Every time element in the list we need to set pCurrent on pCurrent ->pNext. Each list item is checked if storage search value. If the searched value is found, it returns a pointer to the list item. If the end of the list is reach without finding the item with storage value NULL pointer is returned.

2. Abstract Data Types

What are Abstract Data Types (ADTS)? Roughly speaking, it is a way of looking at the data structure: focus on what needs to be done (with the data structure), ignoring the method of operation of data structures.

ADT is the concept of data structures separated from implementation. As examples of abstract data types can be stacks and queues. Separating data structures from their implementation will be shown also by stacks and queues, but now we'll use a linked list not as previously arrays.

Implementation of stack based on a linked list

Implementation push() and pop()

On the classes 04-05 to create a stack, we used class vector (for data storage). Operations on the stack have been done using function push() and pop() as operations on vectors, ie. The operation of adding (pushing) to the stack

stackVect[++top] = data;

and operation removing (poping) from the stack

data = stackVect[top--];

Data storage on stack may also be implemented by using the linked list. In this case, the operations pop() and push() are implemented by operations such as

theList.insertFirst(data)

and

data = theList.deleteFirst()

User of the stack class calls the push() and pop() to add and to delete an element, without knowledge (or the need for knowledge) about whether the stack is implemented as an array or as a linked list.

Example 3 (C++; (singly) linked list) Example of implementation of the stack (class LinkStack) using a class LinkList instead of an array.

[linkStack.cpp]

```
//linkStack.cpp
//demonstrates a stack implemented as a list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;               //data item
        Link* pNext;                //next link in list

        Link(double dd) : dData(dd), pNext(NULL)
        {  }                        //constructor

        void displayLink()          //display ourself
        { cout << dData << " "; }
};  //end class Link
//-----------------------------------------------------------
class LinkList
{
    private:
        Link* pFirst;                   //ptr to first item on list

    public:
        LinkList() : pFirst(NULL)       //constructor
        {  }

        ~LinkList()                     //destructor (deletes links)
        {
            Link* pCurrent = pFirst;    //start at beginning of list
            while(pCurrent != NULL)     //until end of list,
            {
                Link* pOldCur = pCurrent;    //save current link
                pCurrent = pCurrent->pNext;  //move to next link
                delete pOldCur;              //delete old current
            }
        }

        bool isEmpty()                  //true if list is empty
        { return (pFirst==NULL); }

        void insertFirst(double dd)     //insert at start of list
        {                               //make new link
            Link* pNewLink = new Link(dd);
            pNewLink->pNext = pFirst;   //newLink --> old first
            pFirst = pNewLink;          //first --> newLink
        }

        double deleteFirst()            //delete first item
```

```
        {                                   //(assumes list not empty)
            Link* pTemp = pFirst;           //save old first link
            pFirst = pFirst->pNext;         //remove it: first-->old next
            double key = pTemp->dData;      //remember data
            delete pTemp;                   //delete old first link
            return key;                     //return deleted link's data
        }

        void displayList()                  //display all links
        {
            Link* pCurrent = pFirst;        //start at beginning of list
            while(pCurrent != NULL)         //until end of list,
            {
                pCurrent->displayLink(); //print data
                pCurrent = pCurrent->pNext;  //move to next link
            }
            cout << endl;
        }
};  //end class LinkList
//---------------------------------------------------------------
class LinkStack
{
    private:
        LinkList* pList;                    //pointer to linked list
    public:
        LinkStack()                         //constructor
        { pList = new LinkList; }           //make a linked list

        ~LinkStack()                        //destructor
        { delete pList; }                   //delete the linked list

        void push(double j)                 //put item on top of stack
        { pList->insertFirst(j); }

        double pop()                        //take item from top of stack
        { return pList->deleteFirst(); }

        bool isEmpty()                      //true if stack is empty
        { return ( pList->isEmpty() ); }

        void displayStack()
        {
            cout << "Stack (top-->bottom): ";
            pList->displayList();
        }
};  //end class LinkStack
//---------------------------------------------------------------
int main()
{

    LinkStack theStack;             //make stack

    theStack.push(20);              //push items
    theStack.push(40);

    theStack.displayStack();        //display stack (40, 20)

    theStack.push(60);              //push items
    theStack.push(80);

    theStack.displayStack();        //display (80, 60, 40, 20,)
```

```
    theStack.pop();                 //pop items (80, 60)
    theStack.pop();

    theStack.displayStack();        //display stack (40, 20)
    return 0;
}  //end main()
```

Relation between classes

Function main() is associated only with the class LinkStack. LinkStack class in turn is only associated with the class LinkList. Between the main() and class LinkList there is no direct communication.

Implementation of the stack with a linked list instead of arrays shows how interface can be disconnected (visible/available to the user class) from implementation of (hidden) stack.

Double-ended lists

The implementation of the queue will be done using double-ended list. Double-ended list is similar to the usual (singly) linked list (single-ended list) but now object list has a reference to the first pFirst and last pLast item of the list. Reference to the last item in the list allow us to attach easily a new item at the end of it as in the case of attaching an element to the beginning of the list (of course, it is possible to attach the element to the end of single-ended list going through all the elements of the list but it is more difficult and less efficient). Easy access to the first and last element of the the list is appropriate in situations where the standard single-ended list is not effective. An example here is the implementation of the queue.

Example 4 (C++) Implementation of double-ended list.

[firstLastList.cpp]

```
//firstLastList.cpp
//demonstrates list with first and last references
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                   //data item
        Link* pNext;                    //ptr to next link in list

        Link(double d) : dData(d), pNext(NULL)  //constructor
        {  }

        void displayLink()              //display this link
        { cout << dData << " "; }
};  //end class Link
//--------------------------------------------------------------
class FirstLastList
{
    private:
```

```
        Link* pFirst;                   //ptr to first link
        Link* pLast;                    //ptr to last link

    public:
        FirstLastList() : pFirst(NULL), pLast(NULL)   //constructor
        {  }

        ~FirstLastList()                        //destructor
        {                                       //  (deletes all links)
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                Link* pTemp = pCurrent;         //remember current
                pCurrent = pCurrent->pNext;     //move to next link
                delete pTemp;                   //delete old current
            }
        }

        bool isEmpty()                          //true if no links
        { return pFirst==NULL; }

        void insertFirst(double dd)             //insert at front of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pLast = pNewLink;               //newLink <-- last
            pNewLink->pNext = pFirst;           //newLink --> old first
            pFirst = pNewLink;                  //first --> newLink
        }

        void insertLast(double dd)              //insert at end of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pFirst = pNewLink;              //first --> newLink
            else
                pLast->pNext = pNewLink;        //old last --> newLink
            pLast = pNewLink;                   //newLink <-- last
        }

        void removeFirst()                      //remove first link
        {                                       //(assumes non-empty list)
            Link* pTemp = pFirst;               //remember first link
            if(pFirst->pNext == NULL)           //if only one item
                pLast = NULL;                   //NULL <-- last
            pFirst = pFirst->pNext;             //first --> old next
            delete pTemp;                       //delete the link
        }

        void displayList()
        {
            cout << "List (first-->last): ";
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                pCurrent->displayLink();        //print data
                pCurrent = pCurrent->pNext;     //move to next link
            }
            cout << endl;
        }
};  //end class FirstLastList
```

```
//--------------------------------------------------------------
int main()
{
    FirstLastList theList;              //make a new list

    theList.insertFirst(22);            //insert at front
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);             //insert at rear
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayList();              //display the list

    cout << "Deleting first two items" << endl;
    theList.removeFirst();              //remove first two items
    theList.removeFirst();

    theList.displayList();              //display again
    return 0;
}  //end main()
```

Implementation of the queue using double-ended list

Example 5 (C++; double-ended list) Implementation of data structure: the queue (you can compare with the program Queue.cpp from classes No. 5 example 2).

[linkQueue.cpp]

```
//linkQueue.cpp
//demonstrates queue implemented as double-ended list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                   //data item
        Link* pNext;                    //ptr to next link in list

        Link(double d) :dData(d), pNext(NULL)  //constructor
        {  }

        void displayLink()              //display this link
        { cout << dData << " "; }
};  //end class Link
//--------------------------------------------------------------
class FirstLastList
{
    private:
        Link* pFirst;                   //ptr to first link
        Link* pLast;                    //ptr to last link

    public:
        FirstLastList() : pFirst(NULL), pLast(NULL)  //constructor
        {  }
```

```
        ~FirstLastList()                        //destructor
        {                                       //  (deletes all links)
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                Link* pTemp = pCurrent;         //remember current
                pCurrent = pCurrent->pNext;     //move to next link
                delete pTemp;                   //delete old current
            }
        }

        bool isEmpty()                          //true if no links
        { return pFirst==NULL; }

        void insertLast(double dd)              //insert at end of list
        {
            Link* pNewLink = new Link(dd);      //make new link
            if( isEmpty() )                     //if empty list,
                pFirst = pNewLink;              //first --> newLink
            else
                pLast->pNext = pNewLink;        //old last --> newLink
            pLast = pNewLink;                   //newLink <-- last
        }

        double removeFirst()                    //delete first link
        {                                       //(assumes non-empty list)
            Link* pTemp = pFirst;               //remember first link
            double temp = pFirst->dData;
            if(pFirst->pNext == NULL)           //if only one item
                pLast = NULL;                   //null <-- last
            pFirst = pFirst->pNext;             //first --> old next
            delete pTemp;                       //delete the link
            return temp;
        }

        void displayList()
        {
            Link* pCurrent = pFirst;            //start at beginning
            while(pCurrent != NULL)             //until end of list,
            {
                pCurrent->displayLink();        //print data
                pCurrent = pCurrent->pNext;     //move to next link
            }
            cout << endl;
        }
};  //end class FirstLastList
//--------------------------------------------------------------
class LinkQueue
{
    private:
        FirstLastList theList;

    public:
        bool isEmpty()                  //true if queue is empty
        { return theList.isEmpty(); }

        void insert(double j)           //insert, rear of queue
        { theList.insertLast(j); }

        double remove()                 //remove, front of queue
```

```
        {  return theList.removeFirst();  }

        void displayQueue()
        {
            cout << "Queue (front-->rear): ";
            theList.displayList();
        }
};  //end class LinkQueue
//-----------------------------------------------------------------
int main()
{
    LinkQueue theQueue;         //make a queue

    theQueue.insert(20);        //insert items
    theQueue.insert(40);

    theQueue.displayQueue();    //display queue (20, 40)

    theQueue.insert(60);        //insert items
    theQueue.insert(80);

    theQueue.displayQueue();    //display queue (20, 40, 60, 80)

    cout << "Removing two items" << endl;
    theQueue.remove();          //remove items (20, 40)
    theQueue.remove();

    theQueue.displayQueue();    //display queue (60, 80)
    return 0;
}  //end main()
```

3. Specialized list

Implementation of doubly linked list

Doubly linked list provides the ability to pass the list in both directions. In the doubly linked list each element of the list has two pointers (instead of one as in the singly linked list). The first pointer points to the next element (as in the usual list). The second pointer points to the preceding element.

The beginning of the declaration of the doubly linked list:

```
class Link
{
    public:
        double dData;       //data item
        Link* pNext;        //next link in list
        Link* pPrevious;    //previous link in list
};
```

Inserting an element to the front of the list

```
void insertFirst(double dd)          //insert at front of list
{
    Link* pNewLink = new Link(dd);   //make new link
    if( isEmpty() )                  //if empty list,
        pLast = pNewLink;            //newLink <-- last
    else
        pFirst->pPrevious = pNewLink; //newLink <-- old first
    pNewLink->pNext = pFirst;        //newLink --> old first
    pFirst = pNewLink;               //first --> newLink
}
```

Inserting an element to the list after element containing searched data

```
if(pCurrent==pLast)              //if last link,
{
    pNewLink->pNext = NULL;      //newLink --> null
    pLast = pNewLink;            //newLink <-- last
}
else                             //not last link,
{                                //newLink --> old next
    pNewLink->pNext = pCurrent->pNext;
                                 //newLink <-- old next
    pCurrent->pNext->pPrevious = pNewLink;
}
pNewLink->pPrevious = pCurrent; //old current <-- newLink
pCurrent->pNext = pNewLink;     //old current --> newLink
```

Example 6 (C++) Implementation of data structure of type: doubly linked list.

[doublyLinked.cpp]

```
//doublyLinked.cpp
//demonstrates doubly-linked list
#include <iostream>
using namespace std;

class Link
{
    public:
        double dData;                   //data item
        Link* pNext;                    //next link in list
        Link* pPrevious;                //previous link in list

    public:
        Link(double dd) :               //constructor
            dData(dd), pNext(NULL), pPrevious(NULL)
        {  }

        void displayLink()              //display this link
        { cout << dData << " "; }
};  //end class Link
//--------------------------------------------------------------
class DoublyLinkedList
{
    private:
        Link* pFirst;                   //pointer to first item
```

```
    Link* pLast;                    //pointer to last item

public:
    DoublyLinkedList() :            //constructor
        pFirst(NULL), pLast(NULL)
    {  }

    ~DoublyLinkedList()             //destructor (deletes links)
    {
        Link* pCurrent = pFirst;    //start at beginning of list
        while(pCurrent != NULL)     //until end of list,
        {
            Link* pOldCur = pCurrent;   //save current link
            pCurrent = pCurrent->pNext; //move to next link
            delete pOldCur;             //delete old current
        }
    }

    bool isEmpty()                  //true if no links
    { return pFirst==NULL; }

    void insertFirst(double dd)         //insert at front of list
    {
        Link* pNewLink = new Link(dd);  //make new link
        if( isEmpty() )                 //if empty list,
            pLast = pNewLink;           //newLink <-- last
        else
            pFirst->pPrevious = pNewLink; //newLink <-- old first
        pNewLink->pNext = pFirst;       //newLink --> old first
        pFirst = pNewLink;              //first --> newLink
    }

    void insertLast(double dd)          //insert at end of list
    {
        Link* pNewLink = new Link(dd);  //make new link
        if( isEmpty() )                 //if empty list,
            pFirst = pNewLink;          //first --> newLink
        else
        {
            pLast->pNext = pNewLink;    //old last --> newLink
            pNewLink->pPrevious = pLast;//old last <-- newLink
        }
        pLast = pNewLink;               //newLink <-- last
    }

    void removeFirst()                  //remove first link
    {                                   //(assumes non-empty list)
        Link* pTemp = pFirst;
        if(pFirst->pNext == NULL)       //if only one item
            pLast = NULL;               //null <-- last
        else
            pFirst->pNext->pPrevious = NULL;  //null <-- old next
        pFirst = pFirst->pNext;         //first --> old next
        delete pTemp;                   //delete old first
    }

    void removeLast()                   //remove last link
    {                                   //(assumes non-empty list)
        Link* pTemp = pLast;
        if(pFirst->pNext == NULL)       //if only one item
            pFirst = NULL;               //first --> null
```

```
        else
            pLast->pPrevious->pNext = NULL;  //old previous --> null
        pLast = pLast->pPrevious;        //old previous <-- last
        delete pTemp;                    //delete old last
    }

                                         //insert dd just after key
    bool insertAfter(double key, double dd)
    {                                    //(assumes non-empty list)
        Link* pCurrent = pFirst;         //start at beginning
        while(pCurrent->dData != key)    //until match is found,
        {
            pCurrent = pCurrent->pNext; //move to next link
            if(pCurrent == NULL)
                return false;            //didn't find it
        }
        Link* pNewLink = new Link(dd);   //make new link
        if(pCurrent==pLast)              //if last link,
        {
            pNewLink->pNext = NULL;      //newLink --> null
            pLast = pNewLink;            //newLink <-- last
        }
        else                             //not last link,
        {                                //newLink --> old next
            pNewLink->pNext = pCurrent->pNext;
                                         //newLink <-- old next
            pCurrent->pNext->pPrevious = pNewLink;
        }
        pNewLink->pPrevious = pCurrent; //old current <-- newLink
        pCurrent->pNext = pNewLink;      //old current --> newLink
        return true;                     //found it, did insertion
    }

    bool removeKey(double key)           //remove item w/ given key
    {                                    //(assumes non-empty list)
        Link* pCurrent = pFirst;         //start at beginning
        while(pCurrent->dData != key)    //until match is found,
        {
            pCurrent = pCurrent->pNext; //move to next link
            if(pCurrent == NULL)
                return false;            //didn't find it
        }
        if(pCurrent==pFirst)             //found it; first item?
            pFirst = pCurrent->pNext;    //first --> old next
        else                             //not first
                                         //old previous --> old next
            pCurrent->pPrevious->pNext = pCurrent->pNext;

        if(pCurrent==pLast)              //last item?
            pLast = pCurrent->pPrevious;  //old previous <-- last
        else                             //not last
                                         //old previous <-- old next
            pCurrent->pNext->pPrevious = pCurrent->pPrevious;
        delete pCurrent;                 //delete item
        return true;                     //successful deletion
    }

    void displayForward()
    {
        cout << "List (first-->last): ";
        Link* pCurrent = pFirst;         //start at beginning
```

```
            while(pCurrent != NULL)          //until end of list,
            {
                pCurrent->displayLink();     //display data
                pCurrent = pCurrent->pNext; //move to next link
            }
            cout << endl;
        }

        void displayBackward()
        {
            cout << "List (last-->first): ";
            Link* pCurrent = pLast;          //start at end
            while(pCurrent != NULL)          //until start of list,
            {
                pCurrent->displayLink();     //display data
                pCurrent = pCurrent->pPrevious; //go to previous link
            }
            cout << endl;
        }
};  //end class DoublyLinkedList
//-----------------------------------------------------------------
int main()
{
    DoublyLinkedList theList;      //make a new list

    theList.insertFirst(22);       //insert at front
    theList.insertFirst(44);
    theList.insertFirst(66);

    theList.insertLast(11);        //insert at rear
    theList.insertLast(33);
    theList.insertLast(55);

    theList.displayForward();      //display list forward
    theList.displayBackward();     //display list backward

    cout << "Deleting first, last, and 11" << endl;
    theList.removeFirst();         //remove first item
    theList.removeLast();          //remove last item
    theList.removeKey(11);         //remove item with key 11

    theList.displayForward();      //display list forward

    cout << "Inserting 77 after 22, and 88 after 33" << endl;
    theList.insertAfter(22, 77);  //insert 77 after 22
    theList.insertAfter(33, 88);  //insert 88 after 33

    theList.displayForward();      //display list forward
    return 0;
}  //end main()
```

Based on:
Robert Lafore – "Teach Yourself Data Structures And Algorithms In 24 Hours"