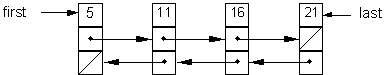
Implementing Doubly Linked Lists

1. The DListNode of a doubly-linked list will contain the information field(s) and two reference fields. One reference will refer to a previous node while the other reference will refer to the next node in the list.
2. Here is a picture of a doubly-linked list, of type DListNode containing Integer objects.



1. A **null** value must be placed at each end of the list to signify the end of the data structure. In the diagram, a **null** is indicated with the diagonal line.
2. A doubly-linked list should have two external references to access the data structure. In the case above, first and last are the two entry points.
3. A doubly-linked list can be traversed in either direction.
4. Inserting values into an ordered doubly-linked list is a similar process to the algorithm used with a singly-linked list. However, the number of reference manipulations will double.
5. The addition of a new node to a position between two existing nodes will require four reference hookups.

As you design and implement your Doubly Linked List, you will implement methods that are be more efficient than your SinglyLinkedList implementation.

1. I will give you the basics of the the DListNode that we will use for this assignment. You will complete the setter/getter and any other helper methods.

**public** **class** DListNode

{

**private** Object value;

**private** DListNode next;

**private** DListNode previous;

**public** DListNode(Object initValue, DListNode initNext, DListNode initPrev)

{

value= initValue;

next = initNext;

previous = initPrev;

}

1. You will need a DLinkedList Class to allow insertion and removal and access to the head and tail of the DLL. You will also include the setter/getter methods.  
     
   **public** **class** DLinkedList {

**private** DListNode firstNode;

**private** DListNode lastNode;

/\*\*

\* Construct an empty list

\*/

**public** DLinkedList() {

firstNode = **null**;

lastNode = **null**;

}

/\*\*

\* Returns true if the list contains no elements

\*/

**public** **boolean** isEmpty()

/\*\*

\* Inserts the argument as the first element of this list.

\*/

**public** **void** addFirst(Object o) {

/\*\*

\* Inserts the argument as the last element of this list.

\*/

**public** **void** addLast(Object o)

/\*\*

\* Removes and returns the first element of this list.

\*/

**public** Object removeFirst()

/\*\*

\* Removes and returns the last element of this list.

\*/

**public** Object removeLast()

/\*\*

\* Returns a String representation of the list.

\*/

**public** String toString()

/\*\*

\* Returns the number of elements in the list as an int.

\*/

**public** **int** size() {

/\*\*

\* Removes all of the elements from this list.

\*/

**private** **void** clear() {

/\*\*

\* Returns a DListIterator.

\*/

**public** DListIterator iterator() {

**return** **new** DListIterator(**this**);

}

1. You will need a DListIterator Class to allow insertion and removal and access to middle nodes of your DLL.

**public** **class** DListIterator {

**private** DListNode current;

**private** DListNode previous;

**private** DLinkedList myList;

**private** **boolean** canRemove; // for remove() method. true if OK to call

// remove()

**public** DListIterator(DLinkedList list) {

myList = list;

current = **null**;

previous = **null**;

canRemove = **false**;

}

**public** String toString()

**public** **boolean** hasPrevious()

**public** **boolean** hasNext()

**public** Object next()

**public** Object previous()

**public** **void** remove()

**public** **void** add(Object element)

**public** **void** set(Object element)

1. Be sure to include many and varied test cases to ensure that your methods work correctly.