Gabriel Brehm

CS260

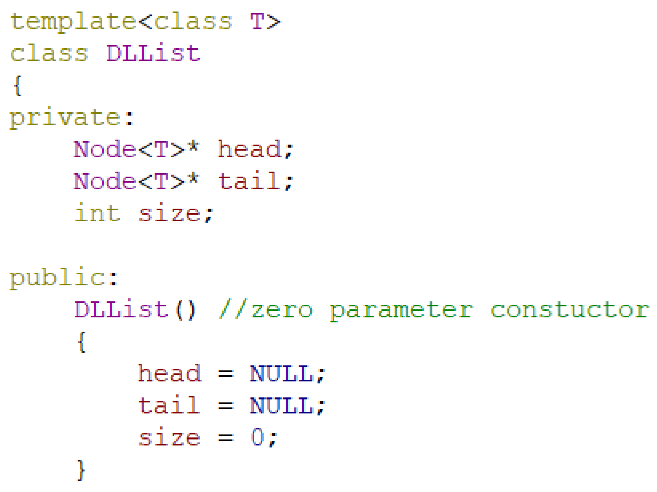
Dr. Breeann Flesch

6 May 2018

Lab 2 Report

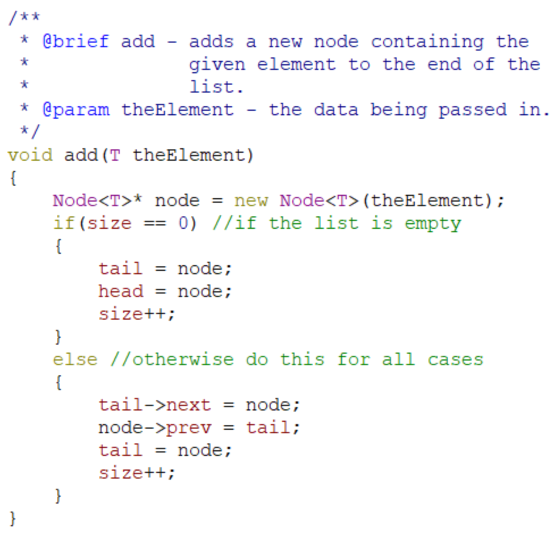
For this lab, I wrote four different header files to create and use a doubly linked list (DLL) with a queue. This report will be going over the function written for the DLL, while the other three header files are able to be seen in the actual code.

First, we have the templated class declaration and the no argument constructor. The class has private member variables for a head and a tail, which are both pointers to nodes, and a size which is an int. The first public function is a constructor which sets the head and tail to NULL and sets the size to zero.



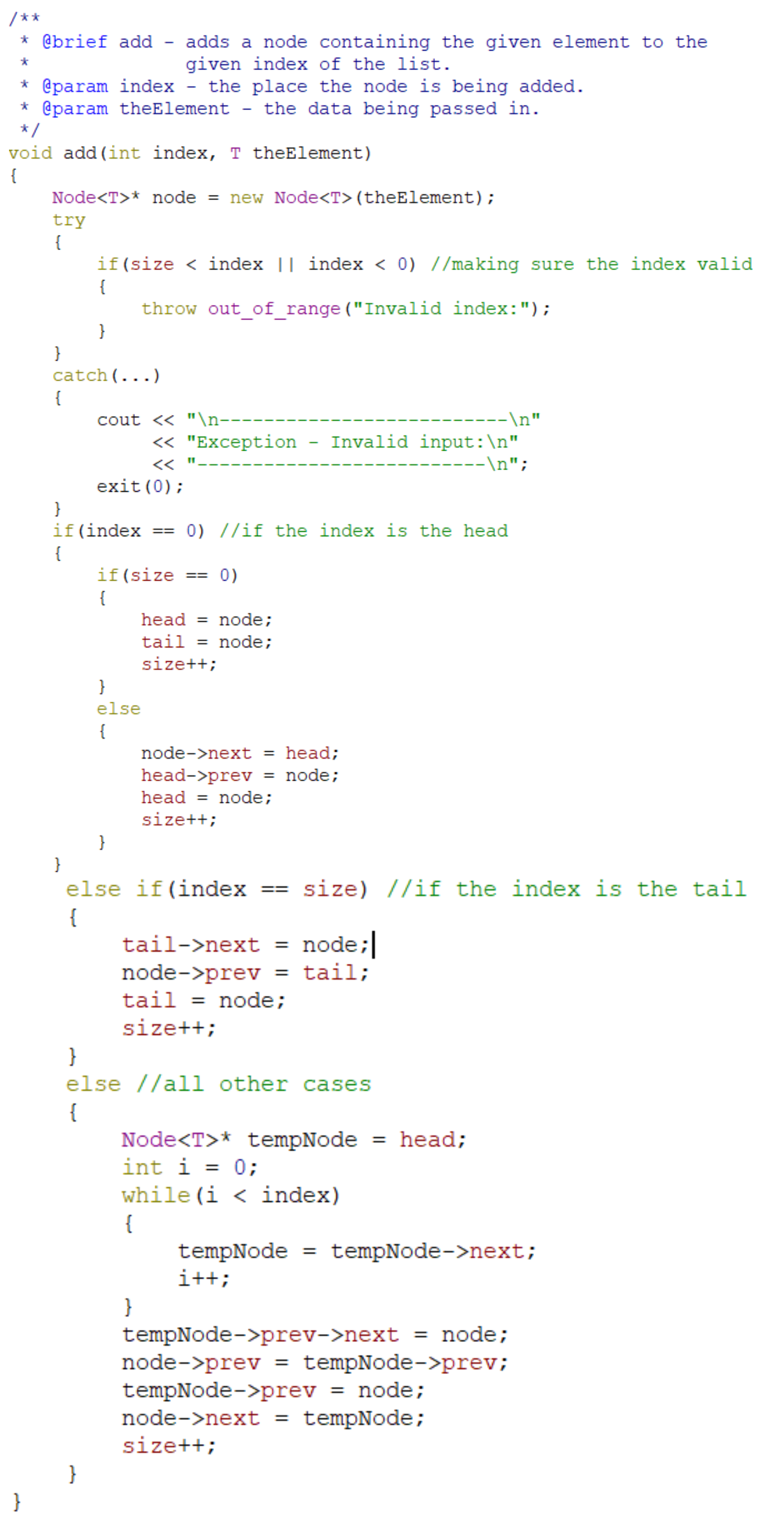
```
template<class T>
class DLList
{
private:
    Node<T>* head;
    Node<T>* tail;
    int size;

public:
    DLList() //zero parameter constuctor
    {
        head = NULL;
        tail = NULL;
        size = 0;
    }
```

Next, we have the add function, which creates a new pointer to a node on the heap, containing the given element, and places it at the end of the list. Of course, if the size of the list is zero, meaning it is empty, the function simply sets both the head and the tail to that pointer to the new node, and increments the size.

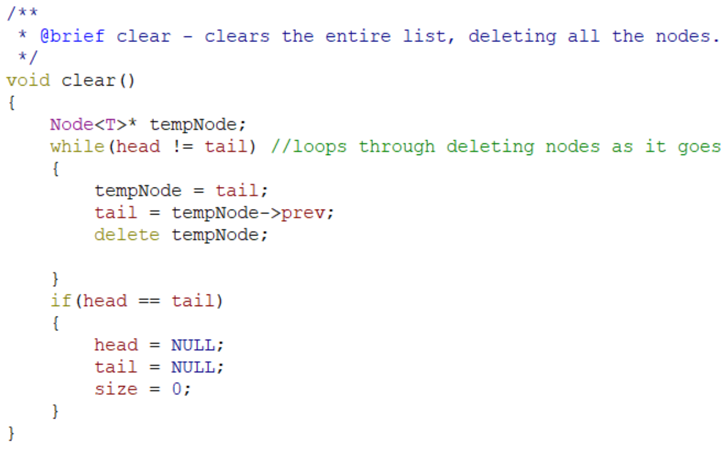


```
/**
 * @brief add - adds a new node containing the
 *              given element to the end of the
 *              list.
 * @param theElement - the data being passed in.
 */
void add(T theElement)
{
    Node<T>* node = new Node<T>(theElement);
    if(size == 0) //if the list is empty
    {
        tail = node;
        head = node;
        size++;
    }
    else //otherwise do this for all cases
    {
        tail->next = node;
        node->prev = tail;
        tail = node;
        size++;
    }
}
```

Third, we have the add at an index function, which is basically a more complex version of the add function previously discussed. First, the function creates a pointer to a new node on the heap, and then it checks if the index given as a parameter is valid. If it is not, and exception is thrown, and the function terminates. Otherwise, if the index is 0, we add the node to the head and increment the size, and if it is equal to the size, we add the node to the tail and increment the size. In the likely case that it is neither of those, we create a temporary pointer to a node, and use it to walk through the array until we find the index we are looking for, then we proceed with some complex pointer reallocation, and then increment the size.

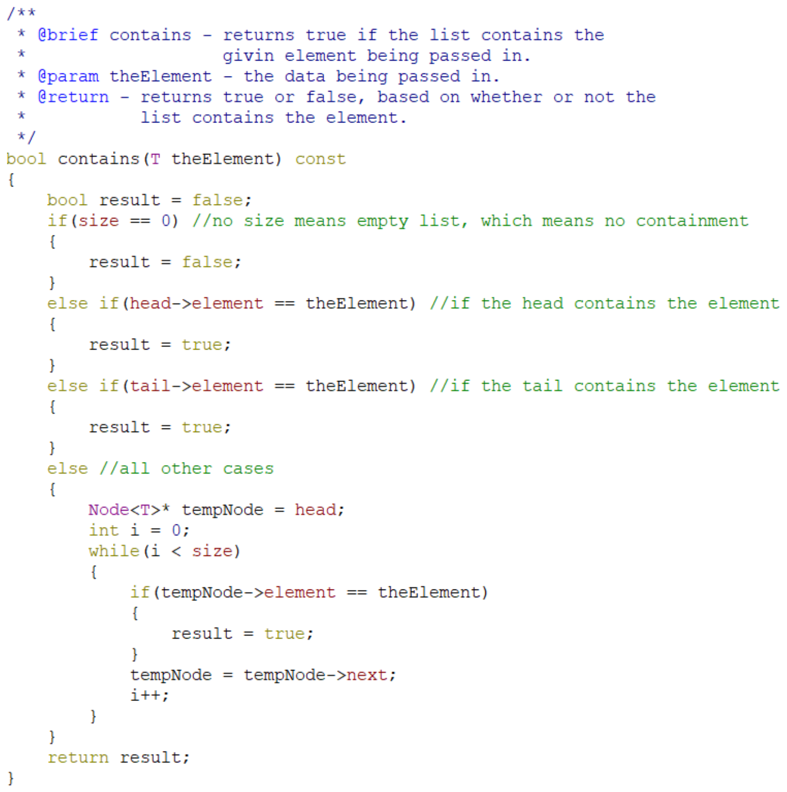


```
/**
 * @brief add - adds a node containing the given element to the
 *              given index of the list.
 * @param index - the place the node is being added.
 * @param theElement - the data being passed in.
 */
void add(int index, T theElement)
{
    Node<T>* node = new Node<T>(theElement);
    try
    {
        if(size < index || index < 0) //making sure the index valid
        {
            throw out_of_range("Invalid index:");
        }
    }
    catch(...)
    {
        cout << "\n--------------------------\n"
             << "Exception - Invalid input:\n"
             << "--------------------------\n";
        exit(0);
    }
    if(index == 0) //if the index is the head
    {
        if(size == 0)
        {
            head = node;
            tail = node;
            size++;
        }
        else
        {
            node->next = head;
            head->prev = node;
            head = node;
            size++;
        }
    }
    else if(index == size) //if the index is the tail
    {
        tail->next = node;
        node->prev = tail;
        tail = node;
        size++;
    }
    else //all other cases
    {
        Node<T>* tempNode = head;
        int i = 0;
        while(i < index)
        {
            tempNode = tempNode->next;
            i++;
        }
        tempNode->prev->next = node;
        node->prev = tempNode->prev;
        tempNode->prev = node;
        node->next = tempNode;
        size++;
    }
}
```



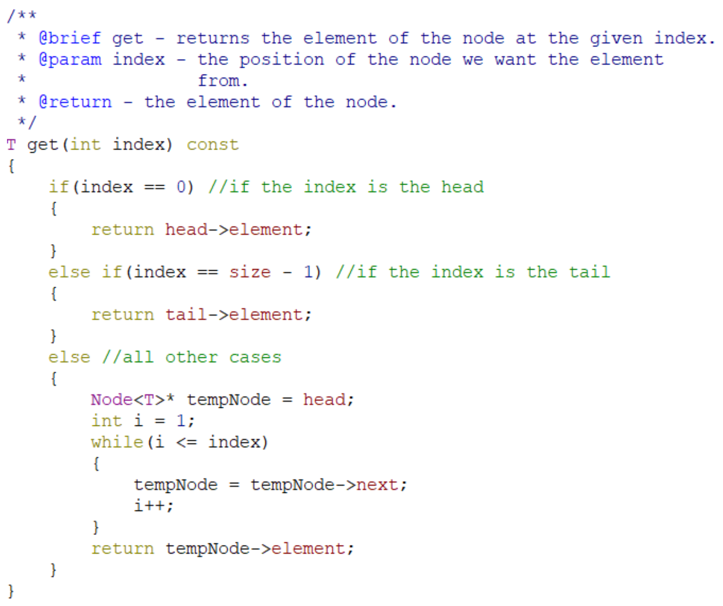
```
/**
 * @brief clear - clears the entire list, deleting all the nodes.
 */
void clear()
{
    Node<T>* tempNode;
    while(head != tail) //loops through deleting nodes as it goes
    {
        tempNode = tail;
        tail = tempNode->prev;
        delete tempNode;

    }
    if(head == tail)
    {
        head = NULL;
        tail = NULL;
        size = 0;
    }
}
```

Next, there is the clear function, which, does exactly what you would think. First, it creates a temporary pointer to a node, and then it loops through the list deleting each node off the heap as it goes. Afterwards, if the head and the tail are pointing to the same node, then it sets the head and tail to NULL and sets the size back to zero.



```
/**
 * @brief contains - returns true if the list contains the
 *                   givin element being passed in.
 * @param theElement - the data being passed in.
 * @return - returns true or false, based on whether or not the
 *           list contains the element.
 */
bool contains(T theElement) const
{
    bool result = false;
    if(size == 0) //no size means empty list, which means no containment
    {
        result = false;
    }
    else if(head->element == theElement) //if the head contains the element
    {
        result = true;
    }
    else if(tail->element == theElement) //if the tail contains the element
    {
        result = true;
    }
    else //all other cases
    {
        Node<T>* tempNode = head;
        int i = 0;
        while(i < size)
        {
            if(tempNode->element == theElement)
            {
                result = true;
            }
            tempNode = tempNode->next;
            i++;
        }
    }
    return result;
}
```

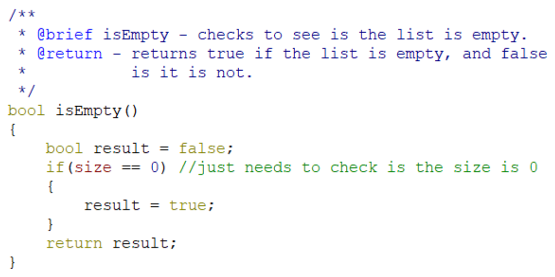
The fifth function is called contains, and it is a Boolean that checks if the list contains the given element. It starts by creating a Boolean variable and sets it to false. Then it checks all of the possibilities for the placement of the element. If the size is zero, the list contains nothing, so the result says false. If the element of the head node is equal to the given element, then the result is true. If the element of the tail is equal to the given element, then the result is true. Otherwise, the function loops through the list until it finds the given element. If it finds it the result is true, if it does not find it, then the result is false, and the function returns the result.



```
/**
 * @brief get - returns the element of the node at the given index.
 * @param index - the position of the node we want the element
 *               from.
 * @return - the element of the node.
 */
T get(int index) const
{
    if(index == 0) //if the index is the head
    {
        return head->element;
    }
    else if(index == size - 1) //if the index is the tail
    {
        return tail->element;
    }
    else //all other cases
    {
        Node<T>* tempNode = head;
        int i = 1;
        while(i <= index)
        {
            tempNode = tempNode->next;
            i++;
        }
        return tempNode->element;
    }
}
```

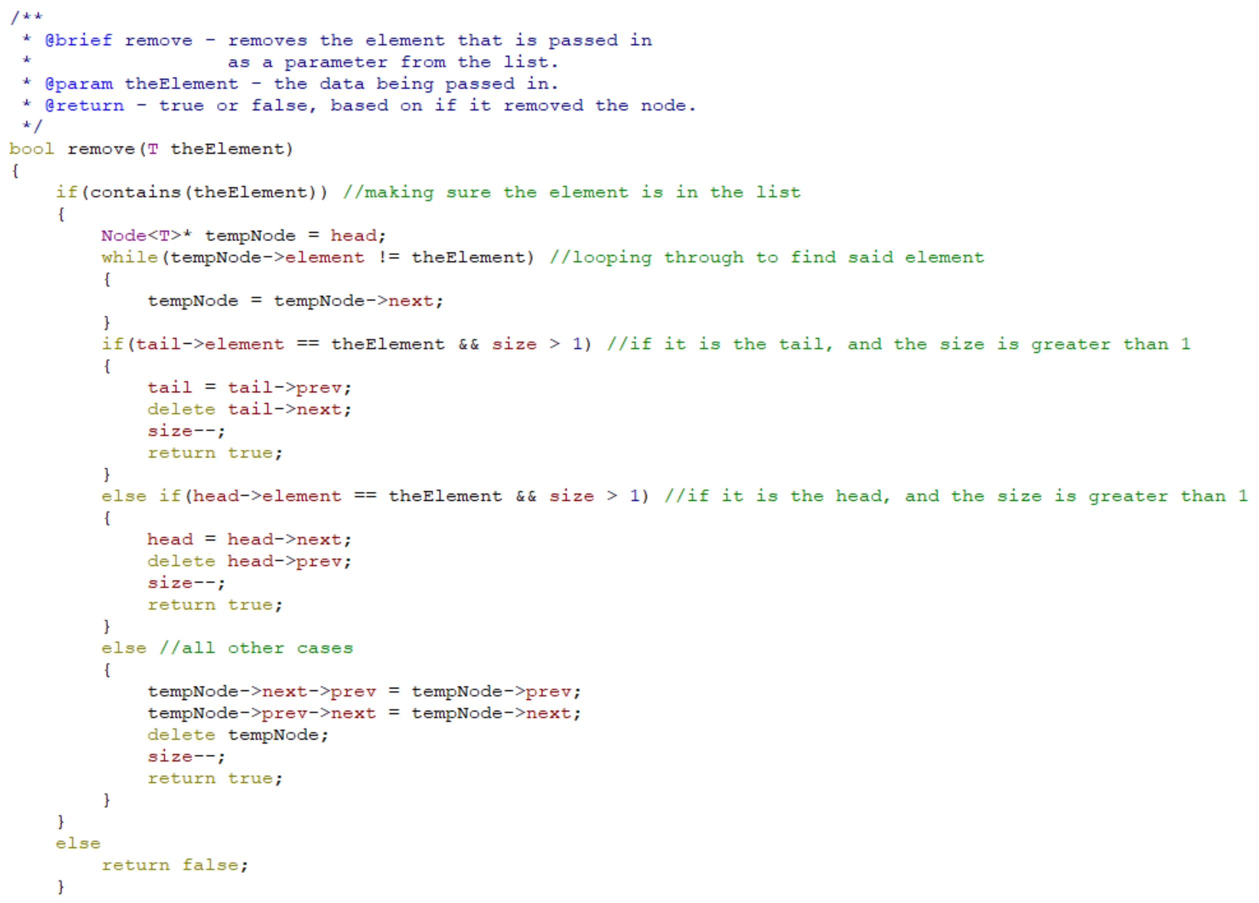
Next, we have the get function, which gets the element of the node of the given index and returns it to the user. If the index is equal to 0, then the function returns the head node's element. If the index is equal to the size of the list minus 1, then the function returns the tail node's element. Otherwise, the function loops though the list until if finds that index and returns the element of the node at that index.

The seventh function checks if the list is empty, simply by checking if the size of the list is zero. If it is, the function returns true, if the it is not, then the function returns false;



```
/**
 * @brief isEmpty - checks to see is the list is empty.
 * @return - returns true if the list is empty, and false
 *           is it is not.
 */
bool isEmpty()
{
    bool result = false;
    if(size == 0) //just needs to check is the size is 0
    {
        result = true;
    }
    return result;
}
```

Next is the remove function, which removes from the list a given element, and returns true or false depending on of the element was removed. First, this function uses the contains function to see if the element we want to remove is in the list. Then it loops through the array until it finds the node that contains the element we want to remove. If that element is at the tail, and the size of the array is greater than 1, we remove the last node, reset the pointers, decrement the size, and return true. If the that element is at the head, we remove the first node, reset the pointers, decrement the size, and return true. Otherwise, we make the two adjacent nodes point to each other, and delete the node that has the element in between them. Then we decrement the



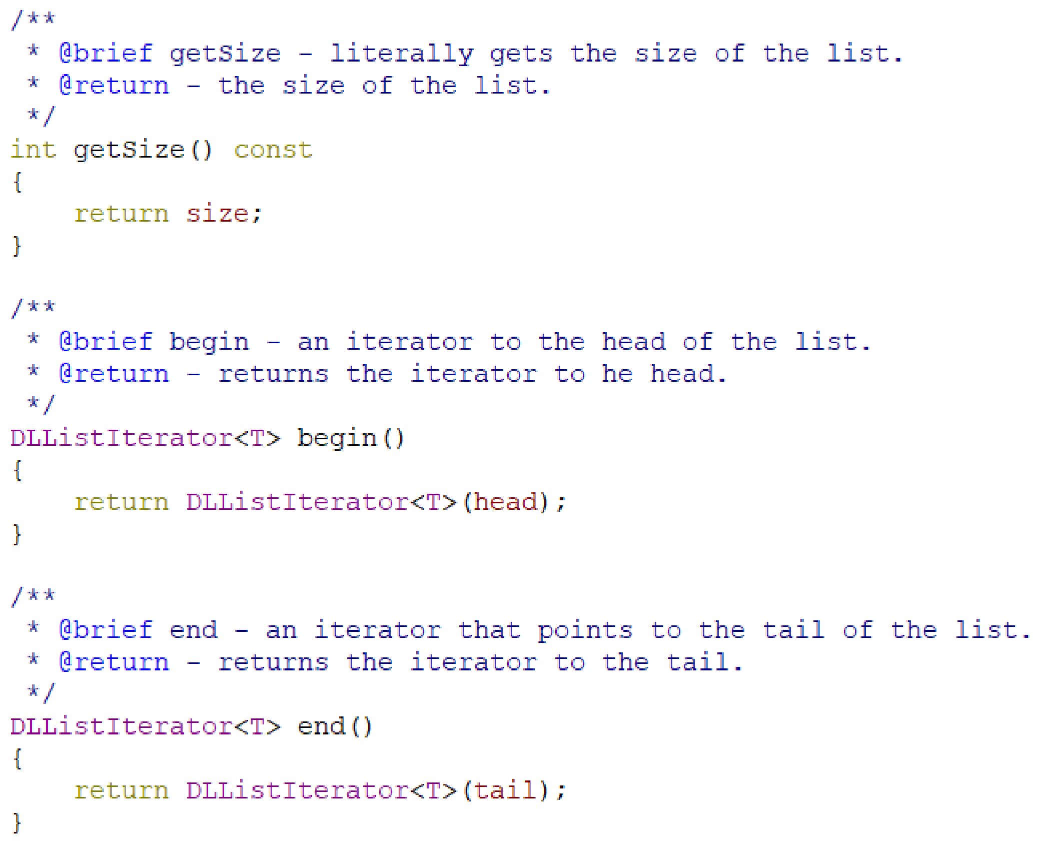
```
/**
 * @brief remove - removes the element that is passed in
 *                 as a parameter from the list.
 * @param theElement - the data being passed in.
 * @return - true or false, based on if it removed the node.
 */
bool remove(T theElement)
{
    if(contains(theElement)) //making sure the element is in the list
    {
        Node<T>* tempNode = head;
        while(tempNode->element != theElement) //looping through to find said element
        {
            tempNode = tempNode->next;
        }
        if(tail->element == theElement && size > 1) //if it is the tail, and the size is greater than 1
        {
            tail = tail->prev;
            delete tail->next;
            size--;
            return true;
        }
        else if(head->element == theElement && size > 1) //if it is the head, and the size is greater than 1
        {
            head = head->next;
            delete head->prev;
            size--;
            return true;
        }
        else //all other cases
        {
            tempNode->next->prev = tempNode->prev;
            tempNode->prev->next = tempNode->next;
            delete tempNode;
            size--;
            return true;
        }
    }
    else
        return false;
    }
```

size and return true. If none of these conditions were met, that is, if the contains function returned false, we just return false as well.

The ninth function does a similar thing to the previous function, except instead of taking in an element to remove, it takes in an index, and it returns the element in the node at that index. Just like the second add function, we check to make sure that the index is valid and throw and out of bounds exception if it is not. If the index is valid, we check if it is the tail of the list. If it is, we reset the pointers, delete the node, decrement the size, and return the element. If the index is the head of the list, we do the same thing, except it is at the beginning of the list. Otherwise, we loop through the list until we reach that index, make the two adjacent nodes point at each other, decrement the size, delete the node and return the element.

```
/**
 * @brief removeAt - removes the node at the specified index.
 * @param index - the index of the node that is being deleted.
 * @return - the element of the node.
 */
T removeAt(int index)
{
    try
    {
        if(size <= index || index < 0) //making sure we have a valid index
        {
            throw out_of_range("Invalid index:");
        }
    }
    catch(...)
    {
        cout << "\n--------------------------\n"
             << "Exception - Invalid input:\n"
             << "--------------------------\n";
        exit(0);
    }
    Node<T>* tempNode = head;
    T theElement;
    if(index == size - 1) //if the tail is the index, delete the tail and reset pointers
    {
        tempNode = tail;
        tail = tempNode->prev;
        tempNode->prev->next = NULL;
        theElement = tempNode->element;
        delete tempNode;
        size--;
        return theElement;
    }
    else if(index == 0) //if the head is the index, delete the head and reset pointers
    {
        head = tempNode->next;
        tempNode->next->prev = NULL;
        theElement = tempNode->element;
        delete tempNode;
        size--;
        return theElement;
    }
    else //all other cases, and generic pointer reset
    {
        int i = 0;
        while(i < index)
        {
            tempNode = tempNode->next;
            i++;
        }
        tempNode->prev->next = tempNode->next;
        tempNode->next->prev = tempNode->prev;
        size--;
        theElement = tempNode->element;
        delete tempNode;
        return theElement;
    }
}
```

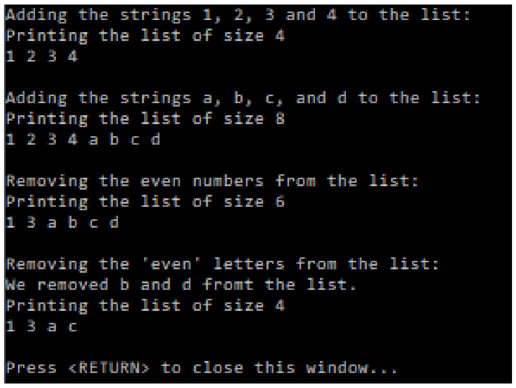
The last three functions here are the size accessor, the begin iterator, and the end iterator. The size accessor does exactly what you would expect; returns the size of the list. It is probably my favorite function. The begin function creates an iterator to the head of the list. With the operator overloads that exist in the iterator header file, this can make it quite easy to traverse through a list. The end function does the same thing as the beginning one, except it makes an iterator to the tail of the list. While these two functions were not used in the other functions listed here, they could easily be implemented every time we traversed the list.



```
/**
 * @brief getSize - literally gets the size of the list.
 * @return - the size of the list.
 */
int getSize() const
{
    return size;
}

/**
 * @brief begin - an iterator to the head of the list.
 * @return - returns the iterator to he head.
 */
DLListIterator<T> begin()
{
    return DLListIterator<T>(head);
}

/**
 * @brief end - an iterator that points to the tail of the list.
 * @return - returns the iterator to the tail.
 */
DLListIterator<T> end()
{
    return DLListIterator<T>(tail);
}
```

In addition to the test cases, I wrote some tests in the main for the functions that were giving me specific trouble. What I did was I made a list of strings and used the first add method to add the strings 1, 2, 3 and 4 into the list. Then I used the other add method to add the strings a, b, c and d to the list. Then I removed the even numbers with the remove element function, and the even letters with the remove at index function. In this, I also used the print function to see my results.

```cpp
int main()
{
    DLList<string> myList;

    cout << "Adding the strings 1, 2, 3 and 4 to the list:\n";
    myList.add("1");
    myList.add("2");
    myList.add("3");
    myList.add("4");
    printList(myList);

    cout << "Adding the strings a, b, c, and d to the list:\n";
    myList.add(4, "a");
    myList.add(5, "b");
    myList.add(6, "c");
    myList.add(7, "d");
    printList(myList);

    cout << "Removing the even numbers from the list:\n";
    myList.remove("2");
    myList.remove("4");
    printList(myList);

    cout << "Removing the 'even' letters from the list:\n";
    string temp1 = myList.removeAt(3);
    string temp2 = myList.removeAt(4);
    cout << "We removed "
         << temp1
         << " and "
         << temp2
         << " fromt the list.\n";
    printList(myList);
```



```
Adding the strings 1, 2, 3 and 4 to the list:
Printing the list of size 4
1 2 3 4

Adding the strings a, b, c, and d to the list:
Printing the list of size 8
1 2 3 4 a b c d

Removing the even numbers from the list:
Printing the list of size 6
1 3 a b c d

Removing the 'even' letters from the list:
We removed b and d fromt the list.
Printing the list of size 4
1 3 a c

Press <RETURN> to close this window...
```

Work Cited

Peer Collaboration:

I worked with Jacob Malmstadt, Megan T, Walker M, Mike D on this project. The test cases were provided by Mike D.

Websites:

http://www.cplusplus.com/

https://stackoverflow.com

http://en.cppreference.com/w/