Fernando Salazar

Assignment 4

Assignment 4 (Arbitrary list operations)

Completion requirements

Create a linked-list that allows:

an add function that takes a value and inserts it into a given position into the list

(example: myList.add(someValue, somePosition))

a remove function that takes a position and removes the value stored at that position of the list and returns it

(example: myList.remove(somePosition))

a get function that takes a position and returns that value without removing it

(example: myList.get(somePosition))

Be sure to include at least one test function for each piece of functionality that should verify that your code is working! This should be at least one test per behavior, likely more. You can make these tests in a source file with a main where your tests are either directly in the main or inside their own standalone functions (please do not neglect the importance of testing!)

Once you have implemented and tested your code, add to the README file what line(s) of code or inputs and outputs show your work meeting each of the above requirements (or better, include a small screen snip of where it meets the requirement!).

(Note: we will cover the analysis of some of this in class next week, then we will have you analyze the next ones!)

Attempt to analyze the complexity of your implementation with line-by-line analysis,

Note: This assignment is to get you to think about the trade-offs that we may have to weigh before using one structure over another

// Design

an add function that takes a value and inserts it into a given position into the list

a remove function that takes a position and removes the value stored at that position of the list and returns it

a get function that takes a position and returns that value without removing it

test functions for each function

//requirements

//add function

```
// Add function that lets you pick location in list
void add(int value, int position) {  // needs data and location

    Node* newNode = new Node();  //  new node
    newNode->data = value;       // Sets the dat in node.

    // sets postion at front of list
    if (position == 0) {
        newNode->next = head;    //  points to current head
        head = newNode; // becomes new head
    } else {
        Node* current = head;
        // loops through the list to the point to just before you want it
        for (int i = 0; i < position - 1; ++i) {
            // this makes the  node poin to the next in the list
            current = current->next;
        }

        newNode->next = current->next;
        current->next = newNode;  // points to new newnode
    }

    counter++;  // adds one to the count
}
```

//remove function

```
// Function to remove an element at any postion
// very similar to add function
int remove(int position) {

    Node* temp; // pointer for node to be deleted
    int value; // for data in node to be deleted

    if (position == 0) {
        temp = head; // saves
        value = head->data; // saves data
        head = head->next; // updates head pointer to  next node
    } else {
        Node* current = head;
        // loops through the list to the point to just before you want it
        for (int i = 0; i < position - 1; ++i) {
            current = current->next;
        }
        temp = current->next;  // saves the node data etc
        value = temp->data;
        current->next = temp->next;
    }

    delete temp;  // Deletes the node
    counter--;      // lowers count

    return value; // Returns the removed node value
}
```

// get function

```
// gets the value at a given position
int get(int position) {

    Node* current = head; // places pointer at head of ist
    // loops through the list till it get to right before you want it.
    for (int i = 0; i < position; ++i) {
        current = current->next;
    }

    return current->data;  // Return the value
}
```

// test functions

```
void testadd() {
    NumbersList list;
    // tesing add function
    std::cout << "Testing add function:\n";
    //adds 3 items to the list
    list.add(1, 0);
    list.add(2, 1);
    //should replace 2
    list.add(3, 1);
    //results
    std::cout << "Expected list: 1, 3, 2, real list: ";
    list.print();
    std::cout << std::endl;

}
```

```
// testing remove function
void testRemove() {
    NumbersList listRemove;

    //adds 3 items to the list
    listRemove.add(1, 0);
    listRemove.add(2, 1);
    //should replace 2
    listRemove.add(3, 1);

    // Test remove function
    //removes postion 1
    std::cout << "Testing remove function:\n";
    std::cout << "Expected remove(1): 3, value removed: " << listRemove.remove(1) << std::endl;
    std::cout << "Expected list after remove: 1 , 2, end, real list: ";
    listRemove.print();
    std::cout << std::endl;

}
```

```cpp
// testing get function
void testGet() {
    NumbersList listGet;

    //adds 3 items to the list
    listGet.add(1, 0);
    listGet.add(2, 1);
    //should replace 2
    listGet.add(3, 1);

    // Test get function
    std::cout << "Testing get function:\n";
    //checks  positions
    std::cout << "Expected get(0): 1, Actual get(0): " << listGet.get(0) << std::endl;
    std::cout << "Expected get(1): 3, Actual get(1): " << listGet.get(1) << std::endl;
    std::cout << "Expected get(2): 2, Actual get(2): " << listGet.get(2) << std::endl;
    std::cout << std::endl;

}
```

// results from test functions

```
Microsoft-MIEngine-Error-1xczrueh.1r1' '--pid=Microsoft-MIEngine-Pid-v
ter=mi'
Testing add function:
Expected list: 1, 3, 2, real list: 1, 3, 2, end

Testing remove function:
Expected remove(1): 3, value removed: 3
Expected list after remove: 1 , 2, end, real list: 1, 2, end

Testing get function:
Expected get(0): 1, Actual get(0): 1
Expected get(1): 3, Actual get(1): 3
Expected get(2): 2, Actual get(2): 2
```

// line analysis

```
/ line-by-line analysis of the three main functions
// not sure if I did it right but the bulk of my code is going to be  o(1) in the
complexity because it only does the one thing,
olny when it as to loop through in 3 parts does it get higher  o(n)
```

```
 // Add function that lets you pick location in list
    void add(int value, int position) {  // needs data and location
 * o(1)      if (position < 0) { // checks if the position is in range //less
then zero
            throw std::out_of_range("position out of range");
        }
 * o(1)       if (position > count) { // checks if the position is in range //
greater than count
            throw std::out_of_range("position out of range");
        }

* o(1)        Node* newNode = new Node();  //  new node
* o(1)       newNode->data = value;       // Sets the dat in node.

        // sets postion at front of list
 * o(1)       if (position == 0) {
            newNode->next = head;    //  points to current head
            head = newNode; // becomes new head
 * o(n)       } else {
            Node* current = head;
            // loops through the list to the point to just before you want it
            for (int i = 0; i < position - 1; ++i) {
                // this makes the  node poin to the next in the list
                current = current->next;
            }

* o(1)            newNode->next = current->next;
* o(1)           current->next = newNode;  // points to new newnode
        }

* o(1)        count++;  // adds one to the count
    }

    // Function to remove an element at any postion
    // very similar to add function
    int remove(int position) {
* o(1)        if (position < 0) { // checks if the position is in range //less
then zero
            throw std::out_of_range("position out of range");
        }
* o(1)        if (position > count) { // checks if the position is in range //
greater than count
            throw std::out_of_range("position out of range");
        }
```

```
* o(1)         Node* temp; // pointer for node to be deleted
* o(1)         int value; // for data in node to be deleted

* o(1)         if (position == 0) {
            temp = head; // saves
            value = head->data; // saves data
            head = head->next; // updates head pointer to  next node
* o(n)         } else {
            Node* current = head;
            // loops through the list to the point to just before you want it
            for (int i = 0; i < position - 1; ++i) {
                current = current->next;
            }
* o(1)             temp = current->next;  // saves the node data etc
* o(1)             value = temp->data;
* o(1)             current->next = temp->next;
        }

* o(1)         delete temp;  // Deletes the node
* o(1)        count--;      // lowers count

* o(1)         return value; // Returns the removed node value
    }

    // gets the value at a given position
    int get(int position) {
* o(1)         if (position < 0) { // checks if the position is in range //less
then zero
            throw std::out_of_range("position out of range");
        }
* o(1)         if (position > count) { // checks if the position is in range //
greater than count
            throw std::out_of_range("position out of range");
        }

        Node* current = head; // places pointer at head of ist
        // loops through the list till it get to right before you want it.
* o(n)         for (int i = 0; i < position; ++i) {
            current = current->next;
        }

* o(1)         return current->data;  // Return the value
    }
```