Raymond Chen

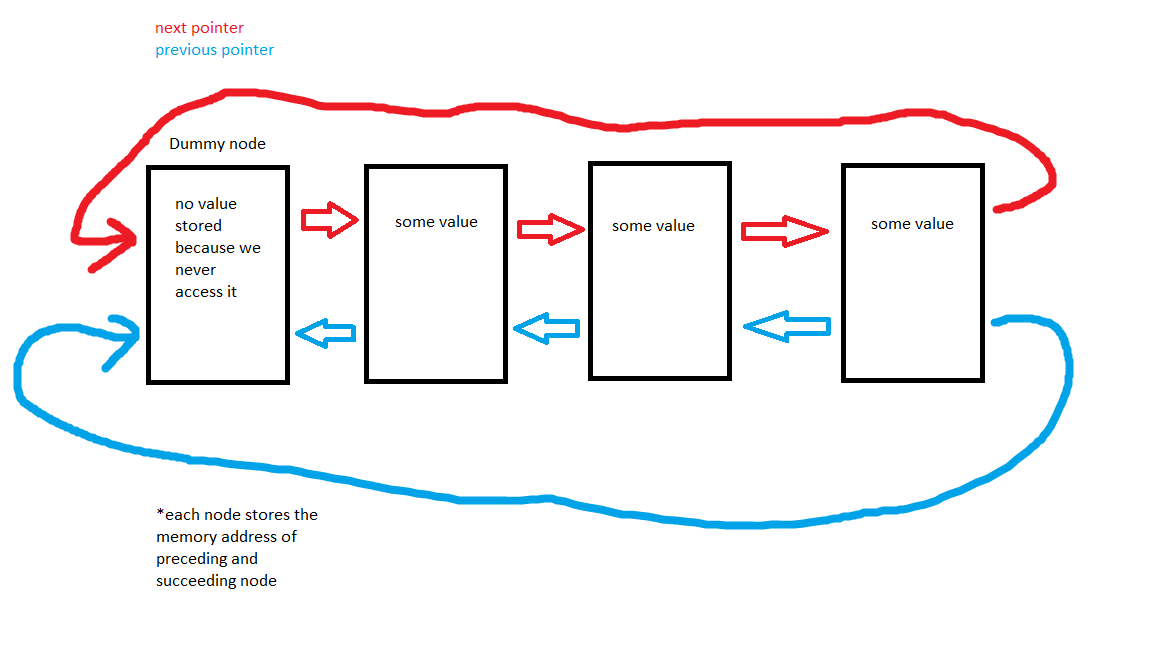
David Smallberg

CS32 Spring 2023

Project 2 Writeup

**Design Description**

I decided to utilize a circular doubly linked list that had a dummy node. The dummy node allows me to eliminate the use of head and tail pointers and allows for the implementation of some of the functions to be easier. An empty linked list would only consist of the dummy node with the previous and next pointers pointing to itself. The behavior of my linked list is illustrated below.



**Pseudocode**

Destructor Implementation

//find the first node of the linked list using a pointer

//have another pointer be assigned the task of deleting the dynamically allocated node

//iterate through entire linked list to an delete each node

//set the pointers of the dummy node to point to itself and set the size to 0 as a way of “resetting” the linked list to its default parameters

Assignment Operator

//check for aliasing

//iterate through the sequence that we want to copy into backwards and delete each node (deleting it backwards prevents the nodes from shifting one position to the left and thus prevents the erase function from skipping over a node)

//have another loop that iterates through the sequence we want to copy from

//use get function and insert function to transfer assign new nodes

Insert (both functions)

//for the first insert implementation check if the position entered by the caller is valid eg. invalid position

would be greater than size or less than 0, do not have to check this in second insert function

//then dynamically allocate memory for a new node using the “new” keyword

//set the value of the new node

//adjust the four pointers to ensure the sequence is linked properly

Erase

//create a new pointer variable of Node type and get it to point at the node BEFORE the one we are wanting to delete

//create another pointer variable that will point to the node want to delete

//adjust pointers so that none of the previous and next pointers are pointing to the node that will be deleted

//delete the node at pos

Remove

//have another tracker node that points to the node before the one we want to delete

//iterate through the sequence and check for matching values

//if values match we will create a pointer that points to the node we want to delete

//adjust the pointers so that none of the previous and next pointers are pointing to the node that will be deleted

//delete node and increment a counter that keeps track of how many values have been removed

Subsequence

//store the value of sequence 2’s first node

//iterate through sequence 1 to see if any values match the first value of sequence 2

//this loop iterates at most the number of times as the size of sequence 1 - size of sequence 2

//if a matching value is found, then we check if the consecutive values match using another loop

//compare the consecutive values, if they don’t match then we break out of the nested loop and continue iterating

//if all the values match, return pos of sequence 1 at which the first values matched

concatReverse

//if the result sequence is not empty we traverse backwards through the result sequence and delete the nodes

//traverse backwards through sequence 1 to grab the values and insert them into the result sequence

//then traverse backwards through sequence 2 to grab the values and insert them into the result sequence

//when inserting the sequence 2 values into the result sequence, we have to use the size of result as our starting index

**Test Cases**

Sequence b;

assert(b.size() == 0); //empty sequence

assert(b.empty() == true); //empty sequence

assert(b.insert("a") == 0); //checks if position

assert(b.insert("b") == 1);

assert(b.insert("c") == 2);

assert(b.insert("a") == 0); //makes sure that insert function compares the values and inserts them accordingly

assert(b.insert("z") == 4); //ensures that largest values goes to end of the sequence

assert(b.size() == 5); //makes sure size is updating properly

assert(b.empty() == false);

Sequence g;

assert(g.insert(1, "apple") == -1); //cannot insert if pos > size

assert(g.insert(-1, "apple") == -1); //cannot insert if pos < 0

assert(g.insert(0, "apple") == 0); //insert at beginning

assert(g.size() == 1); //size updates

assert(g.insert(1, "banana") == 1);

assert(g.size() == 2); //size updates

assert(g.insert(1, "minion") == 1); //inserting minion at pos 1 should move banana to pos 2

assert(g.find("banana") == 2); //banana should be pos 2

Sequence o;

assert(o.insert(0, "dog") == 0);

assert(o.insert(1, "dog") == 1);

assert(o.insert(2, "dog") == 2);

assert(o.insert(3, "dog") == 3);

assert(o.insert(4, "dog") == 4);

assert(o.insert(5, "dog") == 5);

assert(o.insert(6, "dog") == 6);

assert(o.size() == 7);

assert(o.remove("dog") == 7);

assert(o.size() == 0); //check if sequence is empty after removing

assert(o.insert("a") == 0);

assert(o.insert("b") == 1);

assert(o.insert("c") == 2);

assert(o.insert("d") == 3);

assert(o.insert("e") == 4);

assert(o.erase(0) == true);

assert(o.erase(4) == false); //sequence o only has 4 items, cannot erase in 4th index

Sequence z;

z.insert("water");

z.insert("OJ");

z.insert("coke");

z.insert("apple juice");

z.insert("tea");

ItemType value;

assert(z.get(2, value) == true && value == "coke"); //get function copies correct value

assert(z.set(2, "sprite") == true && z.find("sprite") == 2); //set function replace proper value

assert(z.find("dog") == -1);

Sequence j;

assert(j.get(-1, value) == false);

assert(j.set(-1, value) == false);

assert(j.get(0, value) == false);

assert(j.set(0, value) == false);

Sequence seq1;

Sequence seq2;

seq1.insert("a");

seq1.insert("b");

seq1.insert("c");

seq2.insert("d");

seq2.insert("e");

seq2.insert("f");

seq1.swap(seq2); //test for successful swap

assert(seq1.find("d") == 0);

assert(seq1.find("e") == 1);

assert(seq1.find("f") == 2);

assert(seq2.find("a") == 0);

assert(seq2.find("b") == 1);

assert(seq2.find("c") == 2);

Sequence seq3; //swap with empty sequence

Sequence seq4;

seq3.insert("a");

seq3.swap(seq4);

assert(seq3.size() == 0);

assert(seq4.size() == 1);

Sequence seq5; //swap with empty sequence

seq5.insert("a");

seq5.swap(seq5);

assert(seq5.find("a") == 0);

Sequence one;

Sequence two;

one.insert(0, "y");

one.insert(1, "a");

one.insert(2, "r");

two.insert(0, "d");

two.insert(1, "n");

two.insert(2, "o");

two.insert(3, "m");

Sequence result;

result.insert("o");

result.insert("o");

result.insert("o");

assert(result.size() == 3);

concatReverse(one, two, result);

assert(result.find("r") == 0); //test if concatReverse actually reversed the letters

assert(result.find("a") == 1);

assert(result.find("y") == 2);

assert(result.find("m") == 3);

assert(result.find("o") == 4);

assert(result.find("n") == 5);

assert(result.find("d") == 6);

assert(result.size() == (one.size() + two.size())); //size of result should be the cumulative size of sequence one and sequence two

Sequence rea;

rea.insert("a");

rea.insert("b");

rea.insert("c");

rea.insert("d");

rea.erase(2);

assert(rea.find("c") == -1);

rea.dump();

cout << "passed all cases!" << endl;

//subsequence test

//Sequence seqP;

//Sequence seqJ;

//seqP.insert(0, 30);

//seqP.insert(1, 21);

//seqP.insert(2, 63);

//seqP.insert(3, 42);

//seqP.insert(4, 17);

//seqP.insert(5, 17);

//seqP.insert(6, 17);

//seqP.insert(7, 29);

//seqP.insert(8, 8);

//seqP.insert(9, 32);

//seqJ.insert(0, 17);

//seqJ.insert(1, 17);

//seqJ.insert(2, 17);

//assert(subsequence(seqP,seqJ) == 4);