Report

Design of double linked list implementation

My Sequence was implemented using a linear double-linked list with a head pointer points to the first node, as shown on the diagram below:

HEAD

Value

Next

previous

Value

Next

Previous

|  |
| --- |
| Value |
| Pointer Next |
| Pointer Previous |

NULL

**Pseudocode:**

bool Sequence::insert(int pos, const ItemType& value)

{

If pos is out of bound return false;

Increase m\_size by 1;

If the sequence is empty, put the value in the first node and return true;

If the sequence is not empty, and pos == 0 then

Create a new node and put the value in

Connect head node to the newly created node

Name the newly created node as the head node and return true

Else

find the pos-1 node on the sequence

create a new node and put the value in

if next node is not NULL then

connect this new node with previous node and next node

else only connect the previous node to the new node

return true;

}

int Sequence::insert(const ItemType& value)

{

increase m\_size by one;

if the sequence is empty then insert the node as the head node and return 0

else

if value <= value in head node then

create a new node the put the value in

connect this node with the head node

make this new node the new head node

traversal the Sequence to find the first node such that value <= the value in the node

create a new node the put the value in

connect this node with the previous node

if the next node is not nullptr then connect the new node with next node

return the position we put the newly created node.

}

bool Sequence::erase(int pos)

{

if pos is out of bound then retuan false;

decrease the size by one

if pos == 0 then

make the head points to the next node

delete the original head node and return true

else

find the pos-1 node in the sequence

if the pos+1 node is not NULL, connect pos-1 node with pos+1 node

delete the pos node and return true

}

int Sequence::remove(const ItemType& value)

{

For every node in the sequence

If the value in the node == value

call erase function to delete the node

total++

return total

}

void Sequence::swap(Sequence& other)

{

Create a temporary Sequence object

Let temporary = other

Let other = \*this

Let \*this = temporary

}

Sequence::Sequence(const Sequence &other)

{

let m\_size = other.m\_size

for every node in Sequence other

copy it into the newly created sequence object starting from head node

}

Sequence& Sequence::operator=(const Sequence &other)

{

If “this” points to the address different from the address of other

Delete the node originally pointed by “this” pointer

Copy every node of Sequence other in to the Sequence pointed by “this vector”

}

Sequence::~Sequence()

{

Let a pointer A points to the head node

While A is not points to a NULL

Let a new pointer B points to the next node

Delete this node

A = B

}

int subsequence(const Sequence& seq1, const Sequence& seq2)

{

For every node in seq1

If current node in Seq1 s.t. its value equals to the value of the head node in seq2

Starting from this node, check if next seq2.size()-1 node is identical with the corresponding node in seq2.

If it is identical the return the current node position in Seq1

If there is no subsequence return -1

}

void interleave(const Sequence& seq1, const Sequence& seq2, Sequence& result)

{

Create a new sequence called seq3

Let i and j points to 0 pos of seq1 and seq2 respectively

While there is still node in seq1 AND there is still node in seq2

Copy the ith node in seq1 to seq3 and increase i

Copy the jth node in seq2 to seq3 and increase j

If there is still nodes left in seq1

Copy them into the seq3

If there is still nodes left in seq2

Copy them into the seq3

Let result = seq3

}

**Test Cases:**

//default constructor

Sequence s;

// For an empty sequence:

assert(s.size() == 0); // test size

assert(s.empty()); // test empty

assert(s.remove("paratha") == 0); // nothing to remove

// test function insert & find

assert(s.insert(1, "a") == false);

s.insert(0, "a");

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

s.insert(1, "c");

s.insert(1, "b");

assert(s.find("a") == 0 && s.find("b") == 1 && s.find("c") == 2);

assert(!s.insert(4, "d"));

s.insert(0, "aa");

assert(s.find("aa") == 0 && s.find("a") == 1 && s.find("b") == 2 && s.find("c") == 3);

//test function set, find and size

assert(s.find("zz") == -1);

assert(s.set(-1, "wrong") == false && s.set(4, "wrong") == false);

s.set(1, "good");

assert(s.find("aa") == 0 && s.find("good") == 1 && s.find("b") == 2 && s.find("c") == 3);

s.set(0, "good2");

assert(s.find("good2") == 0 && s.find("good") == 1 && s.find("b") == 2 && s.find("c") == 3);

s.set(s.size() - 1, "good3");

assert(s.find("good2") == 0 && s.find("good") == 1 && s.find("b") == 2 && s.find("good3") == 3);

//test another insert function

Sequence s1;

s1.insert("b");

s1.insert("f");

s1.insert("d");

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2);

s1.insert("a");

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2 && s1.find("a") == 3);

s1.insert("c");

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("c") == 2 && s1.find("b") == 3 && s1.find("a") == 4);

s1.insert("g");

assert(s1.find("g") == 0 && s1.find("f") == 1 && s1.find("d") == 2 && s1.find("b") == 4 && s1.find("a") == 5);

//test function erase

assert(!s1.erase(-1) && !s1.erase(s1.size()));

s1.erase(0);

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("c") == 2 && s1.find("b") == 3 && s1.find("a") == 4);

s1.erase(2);

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2 && s1.find("a") == 3);

s1.erase(3);

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2);

//test function remove and insert

assert(s1.remove("a") == 0);

s1.insert("a");

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2 && s1.find("a") == 3);

assert(s1.remove("a") == 1);

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2 && s1.size() == 3);

s1.insert("a");

s1.insert(0, "a");

s1.insert(1, "a");

assert(s1.find("a") == 0 && s1.find("f") == 2 && s1.find("d") == 3 && s1.find("b") == 4);

s1.insert(3, "a");

s1.insert(4, "a");

s1.insert("a");

assert(s1.remove("a") == 6);

assert(s1.find("f") == 0 && s1.find("d") == 1 && s1.find("b") == 2 && s1.size() == 3);

//test function get

ItemType temp;

assert(s1.get(-1, temp) == false && !s1.get(s1.size(), temp));

s1.get(0, temp);

assert(temp == "f");

s1.get(1, temp);

assert(temp == "d");

s1.get(s1.size() - 1, temp);

assert(temp == "b");

//copy constructor

Sequence ss = s1;

assert(ss.find("f") == 0 && ss.find("d") == 1 && ss.find("b") == 2 && ss.size() == 3);

//assignment operator

Sequence sss;

sss = s1;

assert(sss.find("f") == 0 && sss.find("d") == 1 && sss.find("b") == 2 && sss.size() == 3);

//test function swap

Sequence t;

t.insert("IamAgenius");

s1.swap(t);

assert(t.find("f") == 0 && t.find("d") == 1 && t.find("b") == 2 && t.size() == 3 && s1.find("IamAgenius") == 0 && s1.size() == 1);

//test function subsequence

Sequence a;

Sequence b;

a.insert("a");

a.insert("b");

a.insert("a");

a.insert("c");

b.insert("a");

b.insert("b");

assert(subsequence(a, b) == 1);

b.erase(0);

assert(subsequence(a, b) == 2);

b.insert("a");

assert(subsequence(a, b) == 2);

assert(b.remove("a") == 2);

b.insert("c");

b.insert("a");

assert(subsequence(a, b) == -1);

b.insert("b");

a.insert(3, "c");

a.insert(4, "b");

assert(subsequence(a, b) == 0);

//test function interleave

assert(ss.find("f") == 0 && ss.find("d") == 1 && ss.find("b") == 2 && ss.size() == 3);

Sequence result;

interleave(ss, b, result);

assert(result.find("f") == 0 && result.find("c") == 1 && result.find("d") == 2);

assert(result.find("b") == 3 && result.find("a") == 5 && result.size() == 6);

interleave(ss, b, ss);

assert(ss.find("f") == 0 && ss.find("c") == 1 && ss.find("d") == 2);

assert(ss.find("b") == 3 && ss.find("a") == 5 && ss.size() == 6);