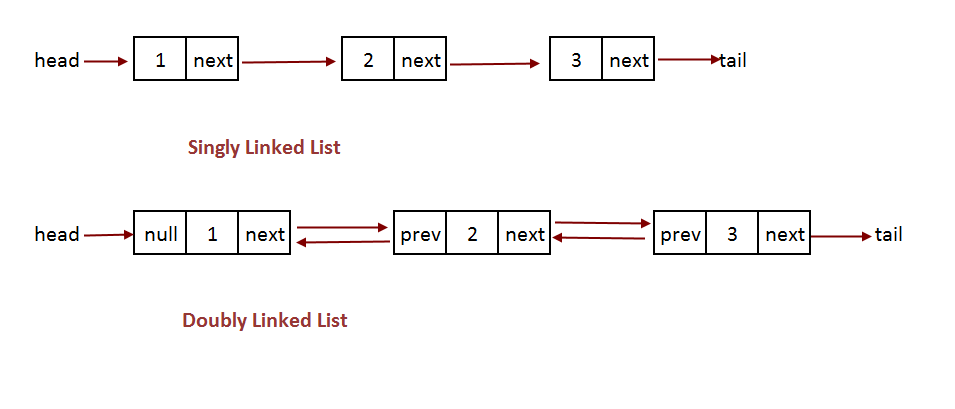
Description:

My Linked List is doubly-linked which means it can be traversed starting from either end (head or tail). It is not circular and does not include any dummy nodes. Each list node contains a value of type ItemType, a Node pointer to the next node, and a Node pointer to the previous node.



This image pulled off of Google Images (<http://algorithms.tutorialhorizon.com/files/2016/03/Doubly-Linked-List.png>) illustrates the difference between a standard singly linked list and the doubly linked list that I implemented.

Pseudocode:

Sequence::Sequence(const Sequence &copy){

Sets values for empty list

While copy node pointer isn’t nullptr{

Create new node and set previous node’s next to that

Traverse both node pointer and copy node pointer

}

Copies tail and size

}

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

if empty, then insert new node at beginning

while (p isn’t nullptr) {

if 0 and not empty, then insert new node at front of list, set next node prev to this node

if pos found & next node isn’t nullptr, insert new node with value and change pointers around to point to this node

if pos found & next node is nullptr, insert new node at end, and change prev node next to point to this node

p = next node, and increment index

}

}

int Sequence::insert(const ItemType& value) follows similar logic to above insert

bool Sequence::erase(int pos) {

if empty return true

if first node, delete and move up next node to head

while (p isn’t nullptr){

if pos found and next node isn’t nullptr, then set nodes around this node each other, and delete this node

if pos found and next node is nullptr, set tail to prev node, and delete this node

p = next node, and increment index

}

}

int Sequence::remove(const ItemType& value) {

while (p isn’t nullptr){

if value found, traverse p, erase(index), increment count

else traverse p, and increment index

}

return count

}

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

if seq2 is empty, return -1

for each element i in seq1

for each element in seq2 starting at i in seq1

if either sequence fails to get the element or that seq1 element != seq2 element, then break out of inner loop

if end of seq2 is reached, then all of seq2 was found in seq1 so return i

return -1

}

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

create copies of seq1 and seq2 to prevent aliasing issues

set result to new empty sequence to eliminate previous elements

for i < size of smaller sequence

insert element at i index from seq1 then seq2 into result

if seq1>seq2, insert rest of seq1 at end of result

else insert rest of seq2 at end of result

}

Test Cases:

// test insert methods

sq.insert("hello"); // test insert into empty list

sq.insert("goodbye"); // test insert into front of list

sq.insert(0, "hi"); // test insert at beginning

sq.insert(1, "again"); // test insert at middle of list

sq.insert(0, "he");

sq.insert("aaaa"); // test insert into front

sq.insert("zzzz"); // test insert into end

sq.insert("hh");

sq.insert(11, "hi"); // test out of bounds insert

sq.insert(8, "hi"); // test end insert

sq.dump();

// test erase & default constructor & empty & size

Sequence testErase; // test default constructor

cerr << testErase.empty() << endl;

cerr << testErase.size() << endl;

testErase.erase(0); // erase empty list

sq.erase(0); // erase first element

sq.erase(10); // erase element out of bounds

sq.erase(6); // erase last element

cerr << sq.size() << endl;

sq.dump();

// test remove

sq.insert("hello");

sq.insert(7, "hello");

sq.remove("hello"); // remove repeated incidents of “hello”

sq.remove("hi"); // remove repeated incidents of “hi”

sq.remove(“smallberg”); // attempt to remove non-existent item in Sequence

sq.dump();

// test find

cerr << sq.find("hh")<< endl; // find end item

cerr << sq.find("goodbye") << endl; // find beginning item

cerr << sq.find(“again”) << endl; // find middle item

cerr<< sq.find("hello") << endl; // try to find non-existent item

// test swap

Sequence sq2;

sq.swap(sq2); // swap with empty sequence

sq.swap(sq2); // swap back

sq2.insert("hello");

sq2.insert("he");

sq2.insert("hh");

sq.dump();

sq2.dump();

cerr << endl;

sq.swap(sq2); // standard swap of 2 lists

sq.dump();

sq2.dump();

// test copy & assignment

Sequence sq3(sq); // create copy of sq

sq.erase(0); // ensure modifying sq does not modify sq3

sq3.dump();

sq.dump();

cerr << sq3.size() << sq.size() << endl << endl;

sq3 = sq2; // assign sq2 to sq3

sq2.erase(0); // ensure modifying sq2 does not change sq3

sq3.dump();

sq2.dump();

sq.dump();

// test get & set

ItemType temp;

sq.get(0, temp); // get first value

cerr << temp << endl << endl;

sq.get(10, temp); // ensure temp doesn’t change from out of bounds get

cerr << temp << endl << endl;

temp = "booger";

sq.set(0, temp); // set first value

sq.get(0, temp);

cerr << temp << endl << endl;

sq.set(10, temp); // ensure nothing in sq changes

cerr << temp << endl << endl;

sq.dump();

// test subsequence

sq.remove("hello");

sq2.insert(0, "hello");

sq2.set(0, temp);

sq2.insert(0, "hello");

sq2.insert(0, "hello");

sq2.insert(0, "hello");

sq2.remove("again");

sq2.remove("goodbye");

sq2.insert(4,"interrupt");

sq2.dump();

sq2.dump();

sq.dump();

cerr << endl;

cerr << subsequence(sq2, sq) << endl; // standard subsequence check

Sequence sq9;

cerr << subsequence(sq2, sq9) << endl; // subsequence with empty sequence

cerr << subsequence(sq, sq2) << endl; // second sequence > first sequence

sq2.remove(“hello”);

cerr << subsequence(sq2, sq) << endl; // when no subsequence found

// test interleave

Sequence result = sq, sq5, sq6;

sq2.remove("hello");

sq2.remove("booger");

sq2.remove("hh");

sq2.insert("unique");

interleave(sq2, sq, result); // standard interleave check

result.dump();

interleave(sq2, sq5, result); // interleave with empty sequence

result.dump();

interleave(sq2, sq2, sq2); // check for aliasing

sq2.dump();