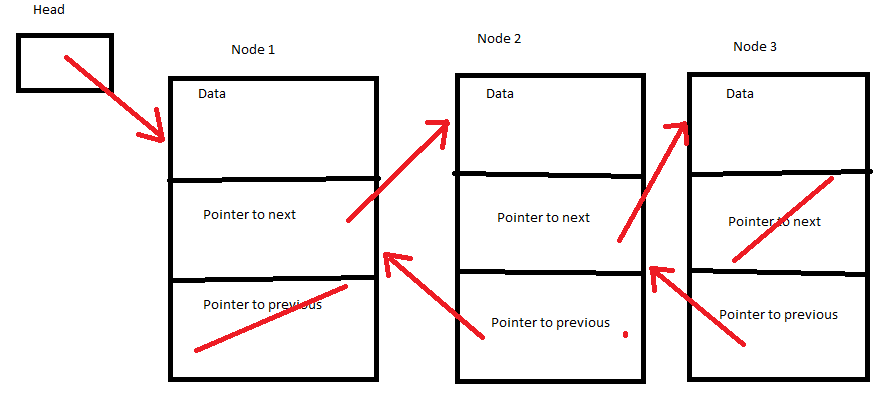
**Doubly-Linked List**

My doubly-linked list implementation is one in which the Sequence contains Nodes that are in a linked list. The nodes contain data of ItemType and two pointers each, one that points to the next Node and one that points to the previous Node. The beginning Node is pointed to by a head pointer.



**Pseudocode**

Sequence::~Sequence()

If there no Nodes in the sequence, you don’t delete anything.

If there is only the one node, delete it.

If there are multiple modes, set two pointers, one to the current item and one to the next.

Delete the current item and make the “current” pointer point to the next item and the next to the “next-next” item.

Repeat this process until you are on the last item, then delete it. (Items = Nodes)

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

If the two Sequences are the same, do nothing.

Get the value of each node and create a new node in a new sequence for it.

Switch the new node with this one and assign the sizes as well.

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

If the pos number is negative or larger than the size, return invalid.

If you want to create a node at the top of the sequence linked list:

Create a new Node and set its pointers to point at the rest of the stack as well as nullptr.

Add size so that we can implement it.

If we are adding the first node, make the node and then increment the size.

If we’re adding at the end, we make a new node, point it at the previous one and point the previous end at the new end.

If in the middle of the list, create the new Node and make sure the pointers are arranged properly to point to the next and previous one.

int Sequence::insert(const ItemType& value)

If the sequence is empty, just insert a new Node and make head point to it.

Otherwise, see if the data member is greater than the value if so, insert new Node here.

While the data member is less than the value, move onto the next pointer, making sure that the next “Node” is not the nullptr. If you get a position that matches the specifications in the list then add the new node and adjust it on both sides. If it happens to be at the end of the list then add Node and adjust the pointers.

bool Sequence::erase(int pos)

If it tries to erase a negative position or position outside size of the linked list, return false. Also return false if the Linked List is empty.

For position 0 deletions: If the size of the linked list is only 1 then it can only delete the first Node (pointed to by head), decrement the size, and return true.

If the linked list is larger than one, we create a pointer to the next item, change it’s previous pointer to null and delete the first node. We then repoint the head to the “new” first one and decrement the size.

For all other, move the pointer through the array, counting with an integer.

If the pointer is the last item in the Linked List, create a pointer to the previous Node, set its next pointer to null and delete the last Node.

If the pointer is not the last item in the list, create a pointer to the previous one and make it point to the current pointer’s next item. It then makes the “next item” point to the “previous item.” After they are linked to one another, we delete our pointer’s Node.

int Sequence::remove(const ItemType& value)

Using the counter and find function, we loop through the sequence, and erase whenever we find our value, incrementing our counter.

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

If the number is less than zero or larger than the size, return false.

Otherwise, we run through the items until we get to the numbered Node that we wanted. At this node we copy its data value into the “value.”

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

If the number is less than zero or larger than the size, return false.

Otherwise we count through the items until we get to the numbered Node and then we replace this data value with the value that we have.

int Sequence::find(const ItemType& value) const

If the Sequence has no values, we return false.

If the value is our first item, we return 0.

We then run through the pointer’s next item. The next item will then be tested for the value, if it is not the value then the next pointer is the tested for nullptr and if not, the pointer is moved to the next one. A counter keeps track of which value the value is found at.

void Sequence::swap(Sequence& other)

Create a temporary pointer that points to this Sequence’s head. Create a temporary Size and set it equal to this Sequence’s size.

Set this Sequence’s head and size to the other’s head and size.

Set the other’s head and size to the temporary ones.

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

If sequence 2 is larger than 1, return false. If sequence 2 is empty then return 0.

Create two items such that we can set their values to the Sequence’s values at certain points.

Using a for loop, cycle through until we have a match in seq1 for seq2 position 0 (utilizing the get function to compare our newly created item). Then use another for loop to compare the subsequent Sequence values in seq 2 to seq 1. If there is a mismatch, leave this loop, if it completes, return the original value where the sequence 1 started matching.

If sequence 1 goes beyond the point where it can fit a subsequence of seq 2, return -1.

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

If one of the sequences is empty, set the result to the other sequence.

Find both sizes of the sequences and set maxsize to the larger one.

Use a for loop to cycle through the sequences to get the values of the sequence. If you are able to get a value (the sequence has not ended for example if one was smaller than the other), then insert that value into the temporary result. After you have finished writing the temporary result, delete whatever is in the result now and set it to the temporary one. This will help if the result is actually one of the sequences as well. By not deleting it until the end, we are still able to interweave its values.

**Test Cases**

Sequence::~Sequence();

Exit out of a loop so that the destructor would be called.

Sequence& Sequence::operator= (const Sequence& value);

Sequence a;

insert(0);

Sequence b;

b = a; //Set the existing Sequence b to a.

Sequence(const Sequence& other);

Sequence a;

insert(0);

Sequence b = a; //Set a new Sequence b to a.

bool empty() const;

Sequence a;

assert(a.empty() == 1); //Properly returns true for an empty sequence

insert(0);

assert(a.empty() == 0); //Properly returns false for a non-empty sequence

int size() const;

Sequence a;

assert(a.size() == 0); //Properly returns 0 for an empty sequence

insert(0);

assert(a.size() == 1); //Properly returns the size for a non-empty sequence

bool insert(int pos, const ItemType& value);

Sequence a;

assert(a.insert(0, 66) == 1); //Properly returns true for inserting into an empty sequence

assert(a.insert(0, 66) == 1); //Properly returns true for inserting at the top of the list

assert(a.insert(1, 66) == 1); //Properly returns true for inserting at the end of the list

assert(a.insert(1, 66) == 1); //Properly returns true for inserting in the middle of the list

assert(a.insert(-1, 66) == 0); //Properly returns false for inserting at a negative number

assert(a.insert(5, 66) == 0); //Properly returns false for inserting at Node past the size

int insert(const ItemType& value);

Sequence a;

assert(a.insert(1) == 0); //Properly returns 0 for inserting into a empty list

assert(a.insert(-1) == 0); //Properly returns 0 for inserting at the top of the list

assert(a.insert(3) == 2); //Properly returns 2 for inserting at the end of the list

assert(a.insert(0) == 1); //Properly returns 1 for inserting in the middle of the list

bool erase(int pos);

Sequence a;

a.insert(1);

a.insert(2);

a.insert(3);

assert(a.erase(-1) == 0); //Properly returns false for trying to erase a negative number

assert(a.erase(5) == 0); //Properly returns false for trying to erase some Node that is larger than our list.

assert(a.erase(2) == 1); //Properly returns true for erasing at the end of the list.

assert(a.erase(0) == 1); //Properly returns true for erasing at the beginning of the list.

a.insert(3);

a.insert(3);

assert(a.erase(1) == 1); //Properly returns true for erasing in the middle of the list.

int remove(const ItemType& value);

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

Sequence a;

a.insert(17);

a.insert(26);

a.insert(2);

assert(a.remove(22) == 0); //Confirm that there is no 22 to remove

assert(a.remove(26) == 1); //Confirm that there is only one 26 to remove

assert(s.remove(66 == 2); //Confirm that it is capable of removing multiple values

bool get(int pos, ItemType& value) const;

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

ItemType value = 99;

assert(s.get(5, value) == 0); //Check if get failed because there is no 5th position

assert(s.get(-2, value) == 0); //Check if get failed because there is no negative position

assert(s.get(4, value) == 1); //Check if get passed because there is a 4th position

bool set(int pos, const ItemType& value);

Sequence a;

a.insert(17);

a.insert(26);

a.insert(2);

assert(a.set(2, 27) == 1); //Check if it passed since there is a 2nd position

assert(a.set(-1, 27) == 0); //Check if it failed since there is no negative position

assert(a.set(5, 27) == 0); //Check if it failed since there is a 5th position

int find(const ItemType& value) const;

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

assert(s.find(22) == -1); //Return -1 because there is no 22 in the Sequence.

assert(s.find(66) == 1); //Return 1, the smallest of the 66 data points.

assert(s.find(12) == 4); //It is able to find something at the end of the Seq.

assert(s.find(35) == 0); //It is able to find something at the beginning of the Seq.

void swap(Sequence& other);

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

Sequence a;

a.insert(17);

a.insert(26);

a.insert(2);

a.swap(s);

s.dump(); //Prints out the s value which should be what you inputted in a.

cerr << endl;

a.dump();//Prints out the a value which should be what you inputted in s.

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

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

Sequence a;

a.insert(17);

a.insert(26);

a.insert(2);

Sequence v;

assert(subsequence(s, a) == -1); //Returns -1 if the a doesn’t match up with s anywhere

assert(subsequence(s, v) == 0); //Returns 0 if seq2 is an empty sequence.

Sequence b;

b.insert(0, 35);

b.insert(1, 66);

assert(subsequence(s, b) == 3); //Returns 3, able to find if it matches at the beginning.

Sequence c;

c.insert(0, 45);

c.insert(1, 23);

assert(subsequence(s, c) == 0); //Returns 0, able to find if it matches end/middle of sequence 1.

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

Sequence s;

s.insert(0, 35);

s.insert(1, 66);

s.insert(2, 66);

s.insert(3, 45);

s.insert(4, 12);

Sequence a;

a.insert(17);

a.insert(26);

a.insert(2);

Sequence v;

interleave(s, a, v);

v.dump(); //Able to interweave different lengthed sequences into an empty sequence.

interleave(s, s, v);

v.dump(); //Able to interweave same lengthed and the same sequences into a non-empty seq.

interleave(v, a, v);

v.dump(); //Able to interweave the sequence if it is the input and the output.