**Chelsea Biala**

**CS32**

**Smallberg Lecture 2**

1. My linked list has a pointer to a head node, which is initialized in the constructor as a node with an arbitrary value (there is no dummy node or tail pointer). An empty list consists of a pointer to an arbitrary node. A populated list has a pointer to that same head node (which has been assigned a value). Each node has a value and a pointer to its next and previous nodes.
2. Sequence(const Sequence& s)

Copies sizes

Copies head node and pointers

Repeatedly:

Copies next node in list

Sets last next to null

~Sequence()

Repeatedly:

Deletes each node in order

Sequence& operator=(const Sequence& n)

If not the same sequence

Creates temp copy of n

Swaps n with this

void uncheckedInsert(int pos, const ItemType& value)

If list is empty

Set head to value

Create new node

If inserting at the beginning

Set head to new node

Find position to insert

If inserting at the end

Add new node, adjust pointers

Set next to null

If inserting in the middle

Add new node, adjust pointers

Increment size

int insert(int pos, const ItemType& value)

Check if pos is out of bounds

Insert (see uncheckedInsert(pos, value))

int insert(const ItemType& value)

Repeatedly

Check if value should be stored at position

Increment position

Insert (see uncheckedInsert(position, value))

bool erase(int pos)

Check if pos is out of bounds

If deleting first node in sequence

Create temp node for head

Move head forward

If there are other nodes in sequence

Delete pointer to head

If deleting only node in sequence

Create new arbitrary head node

Delete old head

Find pos of node to be deleted

Create temp node

Adjust pointers

Delete old node

int remove(const ItemType& value)

Find position of node to be deleted

Check if not found

Repeatedly

Go to position of node to be deleted

Erase(pos)

Increment counter

Find the next node with value

Until no more nodes with value

bool get(int pos, ItemType& value) const

Check if pos is out of bounds

Go to node at position

Assign value at position to value

bool set(int pos, const ItemType& value)

Check if pos is out of bounds

Go to node at position

Assign value to node at position

int find(const ItemType& value) const

Repeatedly

If value is equal to element

Return pos

Go to next node

Until traversed all elements

Return -1

void swap(Sequence& other)

Swap head pointers

Swap sizes

void dump() const

Repeatedly

Print node value

Move to next node

Until traversed all elements

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

Check if both lists are empty

Repeatedly

Repeatedly

If values at position of both sequences match

Iterate through seq2 and see if seq1 continues to match

Set pos to position in seq1

Else

Break

Until through seq2

Until through seq1

Return pos + 1 - size of seq2

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

Until result is empty

Erase node in result

If seq1 and seq2 are empty

Return result empty

If seq1 is bigger or the same size as seq2

Repeatedly

Put value of seq1 into result twice

Increment element counter

Until seq2 size

Repeatedly

Put value of seq2 in between seq1 values

Increment element counter

Until seq2 size

Repeatedly

Add rest of values of seq1 onto end of result

Until seq1 size

Else (seq2 is bigger)

Repeatedly

Put value of seq1 into result twice

Increment element counter

Until seq1 size

Repeatedly

Put value of seq2 in between seq1 values

Increment element counter

Until seq1 size

Repeatedly

Add rest of values of seq2 onto end of result

Until seq2 size

3. Sequence s1;

assert(s1.empty()); // Test empty

assert(s1.size() == 0); // Test size when empty

assert(s1.remove("a") == 0); // Tests removing nothing

Sequence s2(s1); // Copy constructor of empty

assert(s2.empty()); // Test copy of empty

Sequence s3 = s1; // Assignment operator of empty

assert(s3.empty()); // Test assignment of empty

assert(s1.insert("b") == 0); // Test inserting first element

assert(s1.insert("a") == 0); // Test inserting element that should go first

assert(s1.insert("d") == 2); // Test inserting element that should go last

assert(s1.insert("c") == 2); // Test inserting element that should go in middle

assert(s1.size() == 4); // Test size when not empty

assert(s1.insert(0, "1") == 0); // Test when inserting at beginning

assert(s1.insert(8, "1") == -1); // Test when you can't insert

assert(s1.insert(5, "1") == 5); // Test when inserting at end

assert(s1.insert(1, "1") == 1); // Test when inserting in middle

assert(s1.remove("1") == 3); // Test when removing multiple nodes at beginning, end, and middle

assert(s1.remove("1") == 0); // Test when nothing to rmemove

assert(s1.erase(0) == **true**); // Test when erasing first node

assert(s1.erase(1) == **true**); // Test when erasing middle node

assert(s1.erase(1) == **true**); // Test when erasing last node

assert(s1.erase(4) == **false**); // Test when erasing out of bounds

ItemType t;

s1.get(0,t);

assert(t == "b"); // Test get

assert(s1.get(1,t) == **false**); // Test get when out of bounds

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

assert(s1.insert("d") == 2);

assert(s1.set(2, "c") == **true**); // Test set

assert(s1.set(3,"a") == **false**); // Test set when out of bounds

assert(s1.find("a") == 0); // Test find at beginning

assert(s1.find("b") == 1); // Test find in middle

assert(s1.find("c") == 2); // Test find at end

assert(s2.insert("4") == 0);

assert(s2.insert("3") == 0);

assert(s2.insert("2") == 0);

assert(s2.insert("1") == 0);

s1.swap(s2);

assert(s1.size() == 4 && s2.size() == 3); // Test swapping sizes

s1.get(3,t);

assert(t == "4"); // Test swapped

s2.get(2,t);

assert(t == "c"); // Test swapped

s1.insert(0,"c"); s1.insert(0,"b"); s1.insert(0,"a");

assert(subsequence(s1, s2) == 0); // Test when sub starts at 0

s1.insert(0,"0");

assert(subsequence(s1,s2) == 1); // Test when sub starts in middle

s1.erase(4); s1.erase(3); s1.erase(2); s1.erase(1);

assert(subsequence(s1,s2) == -1); // Test when is no sub

s1.insert(4,"a"); s1.insert(5,"b"); s1.insert(6,"c");

assert(subsequence(s1,s2) == 4); // Test when sub ends Sequence

s1.erase(5);

assert(subsequence(s1,s2) == -1); // Test when partial sub at end

s1.insert(6, "f");

assert(subsequence(s1,s2) == -1); // Test when partial sub in middle

assert(subsequence(s1,s3) == -1); // Test when first is empty

assert(subsequence(s3,s1) == -1); // Test when second is empty

assert(subsequence(s3,s3) == -1); // Test when both are empty

assert(subsequence(s1, s1) == 0); // Test when they're the same

Sequence result;

interleave(s1,s2,result); // Test when first is longer

//result.dump();

interleave(s2,s1,result); // Test when second is longer

//result.dump();

interleave(s2,s2,result); // Test when same size (and same Sequence)

//result.dump();

interleave(s2,s3,result); // Test when second is empty

//result.dump();

interleave(s3,s2,result); // Test when first is empty

//result.dump();

interleave(s3,s3,result); // Test when both are empty

//result.dump();

result.insert("1");

interleave(s2,s3,result); // Test when result isn't empty

//result.dump();

cout << "Passed all tests" << endl;