Amir Ali Yazdi-Nejad

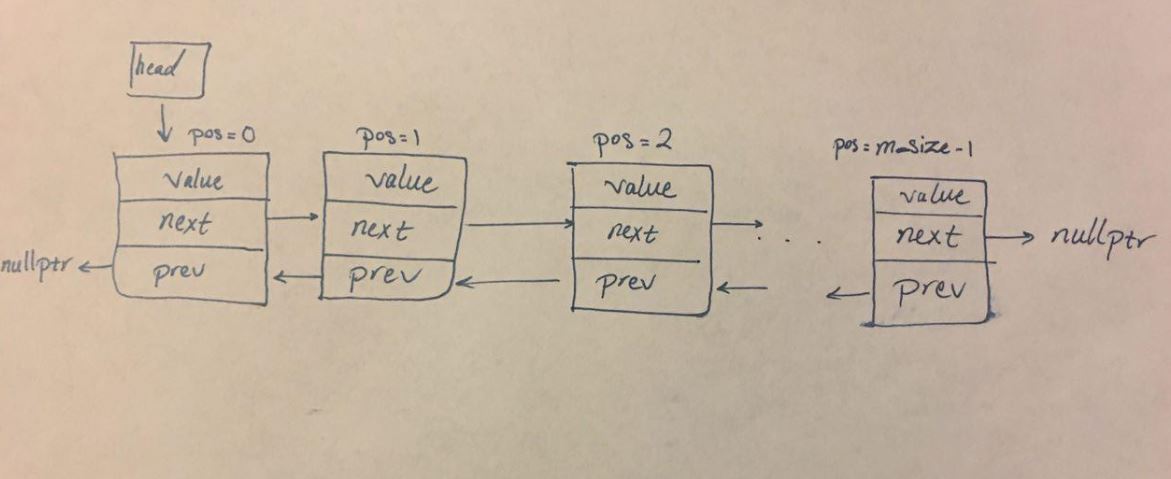
UID: 204846462

CS 32

Professor Nachenberg

**Project 2 Report**

1. In this project, I used Doubly Linked List to construct a sequence. I chose linked list over array, since linked lists reduce the amount of work and special cases when we want to add, erase, swap, or exchange data. A doubly linked list consists a sequence of nodes, which are connected to the previous and next nodes by two pointers called previous and next pointers. Linked lists do not have a fixed size, which makes them better than arrays in some cases. This picture demonstrates the functionality of a doubly linked list of sized m\_size.



1. [**Pseudocode**](http://web.cs.ucla.edu/classes/winter17/cs32/pseudocode.html)**for non-trivial algorithms:**

Sequence::Sequence()

{

Set the head equal to nullptr

Set m\_size equal to zero

}

Sequence::Sequence(const Sequence &old)

{

Set the head equal to nullptr

Set m\_size equal to 0

Loop through from new\_pos = 0 to new\_pos < old.size() (size of the old sequence)

{

Make a new ItemType called new\_value; Get the value at position new\_pos from the old sequence and store it in new\_value

Insert new\_value to the new sequence at position new\_pos

}

}

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

{

if the old sequence is the same as the current sequence

return a pointer to this sequence

else make a copy of the old sequence in a new Sequence called temp using the copy constructor

Switch the data between the current sequence and the temporary sequence and then get rid of the temporary sequence when you exit the function

return a pointer to this sequence

}

Sequence::~Sequence()

{

Make a new node called p which points to the top node

while p is not equal to nullptr

{

Make a new node called n which point to the node after p

Get rid of p

Pass value of n to p

}

}

bool Sequence::empty() const

{

return true if the sequence is empty, otherwise return false

}

int Sequence::size() const // Return the number of items in the sequence.

{

return the number of items in the sequence

}

bool Sequence::insert(int pos, const ItemType& value) // Insert the given value in a specific position

{

if position is less than 0 or greater than the size of the sequence

return false

otherwise call addItem function and give it a position and a value

then return true

}

int Sequence::insert(const ItemType& value) // Insert the given value in an ascending order

{

Declare a new integer called p

Point p to head

Loop through from pos = 0 to pos is less than the size of the sequence and value is greater than the value which p is pointing at

Move pointer p to the next node

Call addItem function and give it a position and a value

return the position where value was inserted

}

bool Sequence::erase(int pos) // Erase the value in the given position

{

if position is less than 0 or greater than or equal to the size of the sequence

return false

Make a pointer which points to the head of the sequence

Declare count as an integer and set it equal to 0

while killMe is not equal to nullptr

{

if count is equal to pos

{

if killMe’s previous pointer is not a nullptr

{

if killMe’s next pointer is not a nullptr

{

Connect the previous pointer to the node after killMe

Connect the next pointer to the node before killMe

}

Otherwise

Set the previous pointer to nullptr

}

Otherwise

{

if the pointer before killMe is not a nullptr

{

Set head equal to the pointer after killMe

Set the next node’s previous pointer to nullptr

}

Otherwise

Set head equal to nullptr

}

Get rid of killMe

Decrement m\_size by one

Break out of the loop

}

Increment count by one

Move killMe so it points to the next node

}

return true

}

int Sequence::remove(const ItemType& value)

{

Declare an integer called count and set it equal to 0

Declare and integer called pos and set it equal to 0

Create a new pointer which points to the first node

While p is not equal to nullptr

{

if the value of node that p is pointing to is equal to value

{

Move p to the next node

Increment count by one

call the erase function at position pos

}

Otherwise

{

Increment pos by one

Move p to the next node

}

}

return count

}

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

{

if position is less than 0 or greater than or equal to the size of the sequence

return false

Create a new pointer called p which points to the first node

Loop through for integer i = 0 to i is less than pos

Move p to the next node

Set value equal to the value which p is pointing to

return true

}

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

{

if position is less than 0 or greater than or equal to the size of the sequence

return false

Create a pointer which points to the first node

Loop through for i = 0 to i is less than pos

Move p to the next node

Set value which p points at to the given value

return true

}

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

{

Create a new integer called pos and set it equal to 0

Create a new pointer which points to the first node

while p is not equal nullptr

{

if the value which p points at is equal to value

return pos

Move p to the next node

Increment pos by one

}

return -1

}

void Sequence::swap(Sequence& other)

{

Create a pointer called p which points at the first node of sequence other

Set the value of other’s head to the current sequence’s head

Set the head of the current sequence to p

Declare a new integer called t\_size and set it equal to size of the other sequence

Set the size of other to the size of current sequence

Set the size of the current sequence to t\_size

}

void Sequence::dump() const

{

Create a new pointer, which points to the first node of the sequence

Print out “sequence: ”

while p is not equal to nullptr

{

Print out the value which p is pointing at plus “ ”

Move p to the next node

}

Jump to the next line of compiler

Print out “size: ” and the value of the size function

}

void Sequence::addToFront(ItemType value)

{

Create a new node and a pointer p which points at it

Set the value of the new node to the given value

if head is not nullptr

Connect head’s node’s previous pointer to p

Connect the next pointer of the node which p is pointing at to head

Connect the previous pointer of the node which p is pointing at to nullptr

Set head equal to p

}

void Sequence::addItem(int pos, ItemType value)

{

if pos is equal to 0

Call the addToFront and give it the given value

Otherwise

{

Declare a new integer called count and set it equal to 0

Create a new pointer which points to the first node

while the next pointer of the node which p is pointing at is not equal to nullptr

{

if pos - 1 is not equal to count

break out of the loop

Otherwise move p to the next node

Increment count by one

}

Create a new node and a pointer called latest

Set the value of the node which latest is pointing at equal to value

Set the next pointer of latest’s node to the next pointer of p’s node

Connect the previous pointer of latest’s node to p

Set the next pointer of latest’s node equal to latest

if latest’s next pointer is not equal to nullptr

Connect the previous pointer of the node after the one latest is pointing at to latest

}

Increment m\_size by one

}

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

{

if size of seq1 is greater than or equal to size of seq2 and seq2 is not empty

{

Declare a new Boolean called subFound and set it equal to false

Declare two new ItemTypes called firstValue and secondValue

Declare a new integer called firstPos

Loop through for pos is equal to 0 to pos less than size of seq1

{

Get the value at position 0 of seq2 and store it in secondValue

Get the value at position pos of seq1 and store it in firstValue

if firstValue is equal to secondValue

{

Set firstPos equal to pos

Loop through for n = 0 to n is less than size of seq2

{

Declare a new integer m and set it equal to firstPos

Add n values to m

Set subFound equal to true

Get the value at position n of seq2 and store it in secondValue

Get the value at position m of seq1 and store it in firstValue

if firstValue is not equal to secondValue

{

Set subFound equal to false

Break out of loop

}

}

if subFound is equal to true

return firstPos

}

}

}

Otherwise

return -1

}

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

{

Declare a new Sequence called temp

Declare two new ItemTypes and call them test1 and test2

Declare a new Boolean called test and set it equal to false

Declare a new Boolean called secondTest and set it equal to false;

if seq1 is empty

Set temp equal to seq2

Otherwise if seq2 is empty

Set temp equal to seq1;

Otherwise if size of seq1 is equal to size of seq2

{

Loop through for i = 0 to i is less than size of seq1

{

Set test equal to true

Get the value at position i of seq1 and store it in test1

Get the value at position i of seq2 and store it in test2

if test1 is not equal to test2

{

Set test equal to false

Break out of loop

}

}

if test is equal to true

Set temp equal to seq1;

}

Otherwise

Set secondTest equal to true

if secondTest is true

{

Declare two new integers called pos1 and pos2 and set them equal to 0

Declare a new integer called pos\_temp and set it equal to 0

Declare two ItemTypes called firstValue and secondValue

while pos1 is less than size of seq1 or pos2 is less than size of seq2

{

if pos1 is less than size of seq1

{

Get the value at pos1 of seq1 and store it in firstValue

Insert firstValue into pos\_temp of the temp sequence

Increment pos by one

Increment pos\_temp by one

}

if pos2 is less than size of seq2

{

Get the value at pos2 of seq2 and store it in secondValue

Insert secondValue into pos\_temp of the temp sequence

Increment pos2 by one

Increment pos\_temp by one

}

}

}

Set result equal to temp

}

1. **Test Cases:**

#include <iostream>

#include <cassert>

#include "Sequence.h"

using namespace std;

int main()

{

ItemType x;

Sequence s;

s.insert(0,56); // Insert values in positions 0-9

s.insert(1,74);

s.insert(2,21);

s.insert(3,64);

s.insert(4,43);

s.insert(5,54);

s.insert(6,99);

s.insert(7,13);

s.insert(8,14);

s.insert(9,19);

s.insert(99); // Insert value by ascending order

Sequence s2;

s2.insert(0, 54); // Insert values into positions 0-2

s2.insert(1, 99);

s2.insert(2, 13);

//s2.remove(76);

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

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

assert(s.remove(35) == 0); // nothing to remove

Sequence s3;

s3.insert(0, 55); // Insert values into positions 0-2

s3.insert(1, 46);

s3.insert(2, 76);

Sequence s4(s2); // Check copy constructor

s4 = s; // Check assignment operator

//s2.set(5, 33); // Test set function

//s2.get(2, x); // Test get function

//cerr << x << endl;

//s2.erase(3); // Test erase function

//s2.remove(76); // Test remove function

//s.swap(s2); // Test swap function

cerr << subsequence(s, s2) << endl; // Test subsequence function

Sequence s5;

interleave(s, s2, s5); // Test interleave function

s.dump(); // Test dump function for s, s2, s3, s4, s5

s2.dump();

s3.dump();

s4.dump();

s5.dump();

cerr << s2.find(99) << endl; // Test find function

cerr << "ALL TEST PASSED" << endl;

return 0;

}