Report for Project 2:

1. I used circularly doubly-linked list because with the head pointing to a sentinel node and looping to the end of the list. For instance, it would make it easier to implement the

Set ::erase() since there is no need to be worried about the edge cases like erasing from the front or the end. I implemented dummy node, so I can use circularly doubly linked list. So in the beginning for the construction of the Node, head \*next, \*prev pointers point to the itself, and keep going till the last node point head again.

2.

**Set :: Set(const Set& other):**

**{**

Copy the “other” size to m\_size

Create a new head node that points to itself

Then create pointer p that points to other.head->next; loop through “other” until it gets back to its head

create a new node called n

n gets p's value

connect n forward to the sentinel

connect n backward to what initially lay there

connect what initally lay there forward to n

connect the sentinel backward to n

move p forward

**}**

**Set :: ~Set()//looping backward to remove each node**

**{**

while the sentinel node’s prev pointer doesn’t point to itself

create a node p that has points to the end

connect what is behind the current end to sentinel

connect the sentinel with what is behind the current end

delete the p pointer

at the end of the while loop, delete head

**}**

**Set& Set :: operator = (const Set &other)**

**{**

if the same object has been passed, aka assigned,

then do simply return itself, \*this

other wise

create a temperoray object Set calling it temp

pass temp to the swap() function

at the end return newly assigned Set, \*this

**}**

**bool Set :: insert(const ItemType& value)**

**{**

First check if the node is not empty

Create a temp node that has the address of the node after sentinel

Do a while loop until temp points back to sentinel node

Check whether “value” exist in the List

Return false if does

Otherwise keep going through list, if value was not in the List, it will exit the while loop

Now after the while loop create another node,\*itr, that has the address the node after sentinel node

Do a while loop in the list with the “itr”

If the value of the current node, itr->value, is the greater/eqaul to as the passed value

Create a new node

Store “value” in its ItemType part of the node

Connect the new node forward the current node,itr

Conncet the new node backward to the node before the current node

Connect the node before the current node forwardly to the new node

Conncet the current node backwardly the new node

Increase the size

Return true as the sign of adding a new node the list

Go the the next node in the list

Outside of the while loop, if the “value” is bigger than values in the List, then insert the new node at the end of the list.

Creat a new node

Make pointer of List that has the address of the last node,\*p

Store “value” in its ItemType part of the node

Make the new node points to sentinel forwardly

Make the new the new node points to the node before sentinel, backwardly

Connect the node before sentinel, p, to the new node forwardly

Connect the sentinel node to the new node backwardly

Increase the size of the list

Return true as a sign of adding a new node

Other wise if was empty

Create a new node

Assigned value to its ItemType value

Make it to point to the head node both with forward and backward

Store the address of the newly created node in the sentinel next and prev pointer

Increase the size of the node

Return true as the result of adding a new node the List

**}**

**bool Set :: contains(const ItemType& value) const**

**{**

Loop through the list by creating pointer of List that has the address of the sentinel node

Loop through till points to the sentinel node

If the passed value is the same as value stored in the current node

Return true as a sign of having the node

Otherwise return false

**}**

**bool Set :: erase(const ItemType& value)**

**{**

Create a new pointer that has the address of what sentinel points to, \*itr

Loop through the List with itr till it points back the sentinel node

If there is node with value of “value”

Create a new pointer that has the address of itr

Connect the node before the current node forwardly to node after the current node

Connect the node after the current node backwardly to the node before current node

Delete the current node

Decrease the size of the list

Return true as a sign of deleting the node with the value of “value”

If there is no such a node yet, keep going through the loop

If did not find the node return false

**}**

**bool Set :: get(int i, ItemType& value) const**

**{**

if “i” is out of range, smaller than zero and greater than the size of the List

return false

other wise

create a pointer that has the address of the sentinel node, \*p

loop through starting from zero till the size of the list

make p points to the next node

if “i” was the same as the current counter

copy the value of the current node into “value”

return true as a sign of doing so

otherwise keep looping through the node

return false if couldn’t find/copy

**}**

**void Set :: swap(Set& other)**

**{**

Create a temp pointer that has the address of the other sentinel node

copy address of the sentinel of node into the sentinel of head

copy the address of the temp pointer to the sentinel node

create an integer data type called tempSize that stores size of” other”

copy the current size into the other size

copy the address of the tempSize into the current size

**}**

**void unite(const Set& s1, const Set& s2, Set& result)**

**{**

Create a temp Set object to copy result in it to avoid aliasing

Do a while loop and call empty() function for result to delete all the nodes in tempResult

If result was not empty

Call get(I,data) function by passing zero for I and ItemType data the first node in the list

Then call erase(data) to delete that node

If s1 was empty

Copy s2 in tempResult

Else if s2 was empty

Copy s1 in tempResult

Otherwise

Do a for loop from zero until the size of s1

Call get function to copy data of each node in s1 in data

Then insert data with insert function in tempResult

Do a for loop from zero until the size of s2

Call get function to copy data of each node in s2 in data

Then insert data with insert function in tempResult (note, insert() avoids inserting repeated nodes)

At the end called the assignment operator to assign tempResult to result

**}**

**void subtract(const Set& s1, const Set& s2, Set& result)**

**{**

Create a temp Set object to copy result in it to avoid aliasing

Do a while loop and call empty() function for result to delete all the nodes in tempResult

If result was not empty

Call get(I,data) function by passing zero for I and ItemType data the first node in the list

Then call erase(data) to delete that node

If s1 was empty

Copy s2 in tempResult

Else if s2 was empty

Copy s1 in tempResult

Otherwise

Do a for loop from zero to the size of s1

Use get function to get the data of each node in s1 and store it in ItemType of data

Check if s2 has that node with contain function

If it does, continue the loop

If s2 does not have that data

Insert it in tempResult

At the end use assignment operator to copy tempResult in result

**}**

**Test Cases:**

int main()

{

//for this testcases; ItemType in Set.h should be changed to **unsigned long**

Set ss, pp;

ItemType x;

assert(!pp.get(0,x));//there is nothing in pp to get

assert(ss.size()==0); //check size function works

assert(ss.empty()); // check to see if ss set is empty

assert(pp.empty()); // check to see if pp set is empty

for (int i = 0; i < 50; i++)

{

ss.insert(i); // Insert 50 items in ss

}

assert(ss.size() == 50); // Check to see if ss contains 50 nodes

ss.swap(pp); // Swap the random values from a for numbers 0-50 into pp

assert(pp.size() == 50); // Check to see if the size is 50 for pp

for (int k = 0; k < 20; k++)

{

pp.erase(k); //remove the first 20 numbers

}

assert(pp.size() == 30); // Checks if size is now 30

Set set1;

set1.insert(30); // The following inserts test if the data is correctly inserted in the sequence

set1.insert(21);

set1.insert(63);

set1.insert(42);

set1.insert(17);

set1.insert(10);

cout <<"set1:"<<endl;

set1.dump();//check whether insert works correctly and they have been added in order

assert(!set1.erase(654));//nothing to erase

Set set2;

set2.insert(17);

set2.insert(63);

set2.insert(29);

cout<< "set2 before the assignment operator:"<<endl;

set2.dump();

ItemType data;

assert(set1.get(1,data) && data==17);//check get works and gets the correct data

Set result;

cout << "Result in Unite:" << endl;

unite(set1, set2, result);//check the unite function whether gives out 10, 17, 21, 29, 30, 42, 63

result.dump();

cout<<endl<< endl;

cout << "Result in Subtract:" << endl;

subtract(set1, set2, result);//check the unite function whether gives out 10, 21, 30 , 42

result.dump();

set2=set1; //check assignment operator works

assert(set1.size()==set2.size());//if the assignment operator worked correctly they should both have the same size

assert(set1.contains(21) && set2.contains(21));// they should both contain 21

cout<<"set2 after the assignment:"<<endl;

set2.dump();

Set setNew(pp);

assert(setNew.size()==30);//Check that copy constructor works by checking whether newSet has the same size as pp =30

}