Course: **CSC 340.02**
Student: **Abbas Mahdavi** **SFSU ID: 918345420**
Teammate: **Berke Melisa Sever** **SFSU ID: 921662115**
Assignment Number: **04**
Assignment Due Date & Time: **Jul 23, 2022 at 11:59pm**

PART A: Linked Bag

1. **Output:**

```
C:\Users\abbas\OneDrive\Desktop\Assignment_04_PartA\cmake-build-debug\Assignment_04_PartA.exe
----- LINKED BAG 340 C++-----

--->>>>> Test 1 --->>>>>
 !add()...     #-END 5-FIVE 4-FOUR 4-FOUR 3-THREE 2-TWO 1-ONE 0-ZERO #-BEGIN
 !Display bag: #-BEGIN 0-ZERO 1-ONE 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
  ----------> 9 item(s) total

--->>>>> Test 2 --->>>>>
 !removeSecondNode340()...
 !removeSecondNode340()...
 !Display bag: #-BEGIN 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
  ----------> 7 item(s) total

 !removeSecondNode340()...
 !removeSecondNode340()...
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END
  ----------> 5 item(s) total

--->>>>> Test 3 --->>>>>
 !addEnd340()...
 !addEnd340()...
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR
  ----------> 7 item(s) total

 !addEnd340()...
 !addEnd340()...
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  ----------> 9 item(s) total

--->>>>> Test 4 --->>>>>
 !getCurrentSize340Iterative - Iterative...
  ---> Current size: 9
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  ----------> 9 item(s) total

--->>>>> Test 5 --->>>>>
 !getCurrentSize340Recursive() - Recursive...
  ---> Current size: 9
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  ----------> 9 item(s) total

--->>>>> Test 6 --->>>>>
 !getCurrentSize340RecursiveNoHelper() - Recursive...
  ---> Current size: 9
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  ----------> 9 item(s) total
```

```
--->>>>> Test 7 --->>>>>
 !getFrequencyOf()...
  ---> 0-ZERO:  1
  ---> 1-ONE:   0
  ---> 2-TWO:   0
  ---> 4-FOUR:  3
  ---> 9-NINE:  2
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  -----------> 9 item(s) total

 !getFrequencyOf340Recursive() - Recursive...
  ---> 0-ZERO:  1
  ---> 1-ONE:   0
  ---> 2-TWO:   0
  ---> 4-FOUR:  3
  ---> 9-NINE:  2
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  -----------> 9 item(s) total

--->>>>> Test 8 --->>>>>
 !getFrequencyOf340RecursiveNoHelper() - Recursive...
  ---> 0-ZERO:  1
  ---> 1-ONE:   0
  ---> 2-TWO:   0
  ---> 4-FOUR:  3
  ---> 9-NINE:  2
 !Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END 9-NINE 4-FOUR 9-NINE 0-ZERO
  -----------> 9 item(s) total

--->>>>> Test 9 --->>>>>
 !removeRandom340() ---> 9-NINE
 !removeRandom340() ---> #-END
 !Display bag: 4-FOUR 5-FIVE 4-FOUR #-BEGIN 4-FOUR 9-NINE 0-ZERO
  -----------> 7 item(s) total

 !removeRandom340() ---> 9-NINE
 !removeRandom340() ---> 4-FOUR
 !Display bag: 5-FIVE #-BEGIN 4-FOUR 4-FOUR 0-ZERO
  -----------> 5 item(s) total

 !removeRandom340() ---> #-BEGIN
 !removeRandom340() ---> 4-FOUR
 !Display bag: 5-FIVE 4-FOUR 0-ZERO
  -----------> 3 item(s) total

 !removeRandom340() ---> 4-FOUR
 !removeRandom340() ---> 0-ZERO
 !Display bag: 5-FIVE
  -----------> 1 item(s) total


Process finished with exit code 0
```

Part B: Smart Pointers

1. Deleting the same memory for the second time returns an error. First a pointer to int is created. Thus memory is allocated for it. The first delete pt will not give error. Since the pointer was already freed up, the second delete statement will give an error.

```
#include <iostream>
using namespace std;
int main() {
std::cout << "Hello World!\n";
int *ptr = new int;
 delete ptr;
 delete ptr;
}
```

Here first, creating a pointer to int, allocating memory to it, and then trying to delete ptr. The first statement won't give an error. The second statement gives an error as the pointer is already freed.

2. The program below uses a unique pointer for example. This pointer is a smart pointer. This means the smart pointer will only point to a specific object. In this case int data type is used. In this case, whenever the unique ptr goes out of scope, the memory for that pointer is automatically released, preventing from memory leakage.

```
#include <iostream>
//memory header for using smart pointers
#include using namespace std;
int main() {
//declare smart pointer
unique_ptr intPtr(new int(12));
cout << *intPtr << endl;
 return 0;
//now we are returning without deleting memory , here when unique_ptr goes out of scope ,
memory is deleted without causing memory leak
}
```

3. This smart pointer in particular is used to point to the object. In this case it is of the class pointer. This is the same kind of concept as the last example. Whenever the object goes out of scope, the memory allocated for that data is deleted. In this example, a class is being used to create that pointer.

```
 #include
//memory header for using smart pointers
#include using namespace std;
class example {
int *ptr;
public:
example() {
ptr = new int;
```

```
}
~example() { }
};
int main() {
//now declare smart pointer pointing to object of a class
unique_ptr objPtr;
}
```

A smart pointer is used to point to the object of the class. When the object goes out of scope, the memory allocated for data will automatically be deleted.

4. This code is the demonstration of converting a shared pointer to a unique pointer. A shared and unique pointer have different properties. To convert or make the same, the move method is used. This is already implemented into the program. This alone converts that object to the same properties as the shared pointer.

```
#include <iostream>
#include<memory>
Using namespace std;
class conversion {
int *point;
public:
conversion()
{
point = new int;
```

```
}
~conversion()
{
}
};
int main() {
unique_ptr<conversion> objectPointer
unique_ptr <conversion> sharedObjectPointer = move(objectPointer);
}
```

5.
```
#include <iostream>
#include<memory>
using namespace std;
int main() {
shared_ptr<int>sharedPoint(new int);
weak_ptr<int>wk1 = sharedPoint;
if(wk1.expired()) {
cout << "weak pointer number 1 is expired" << endl;
}
cout << "reference count weak 1: " wk1.use_count() << endl;
}
```

This code is example of using weak pointer and making them dangle. A weak pointer is different from the other pointers.

This pointer points to an object to a deleted object. This tends to leave that pointer with incomplete information to work

properly. C++ identifies weak pointers as shared pointers with nothing to point to.

Part C: Linked Bag, Smart Pointers Version

- The PartC_SmartPointers folder is included in the Code folder, which also includes the main code for part A.
- We have done the smart pointer changes properly, but was not able to run the program or get an output, and got a lot of errors.
- Therefore, we have provided the files we made changes to in the PartC_SmartPointers folder, for your viewing.
- The part A work perfectly fine and runs well, and should receive full points. However, partC is NOT RUNABLE, and we are hoping to receive partial credit.

Node.h (renamed SPNode.h) and Node.cpp (remaned SPNode.cpp)

- SetNext(shared_ptr): Line 34 in .h and Line 38 in .cpp
- shared_ptr getNext() const: Line 38 in .h and Line 49 in .cpp
- shared_ptr next{ nullptr }: Line 43 in .h

LinkedBag.h (Renamed SPLinkedBag.h) and LinkedBag.cpp (Renamed SPLinkedBag.cpp)

- shared_ptr<Node> headPtr{ nullptr }: Line 63 in .h
- shared_ptr<Node> getPointerTo(const ItemType&) const: Line 68 in .h
- Line 148 and 151 in .cpp - getFrequencyOf: Line 130 in .cpp
- clear(): Lines 112 - 124 in .cpp