Assignment 4 Part A and C

Part A

1. Deleting the same memory twice: This error can happen when two pointers address the same dynamically allocated object. If **delete** is applied to one of the pointers, then the object's memory is returned to the Free-store. If we subsequently delete the second pointer, then the Free-store may be corrupted.

When two pointers reference the same data, they both point to the same memory address that the object occupies. Therefore, if you use the delete command on one of the pointers, the memory address will be deleted. Since the other pointer referenced the same memory address, the second pointer is now a floating pointer that doesn't point to anything. If you make the mistake of using the same delete call on that pointer, you will be attempting to delete nothing. This could cause corruption in the Free-store.

Code:

```
string* ptrOne;
string* ptrTwo;
string str{ "I'm a little tea pot" };
ptrOne = str;
ptrTwo = str;
delete ptrOne; //This deletes the memory address of str. Which also deletes it from ptrTwo!
delete ptrTwo; //This could cause corruption in the Free-store
ptrOne = nullptr;
ptrTwo = nullptr;
```

2. Use smart pointers… Objects that must be allocated with **new**, but you like to have the same lifetime as other objects/variables on the Run-time stack. Objects assigned to smart pointers will be deleted when the program exits that function or block.

When a smart pointer is created, it is done so through a constructor that needs to include an object as a parameter. This benefits you because smart pointers are able to delete themselves when the program terminates. When they do this, the object they referenced is deleted as well. If you have an object or some type of data that could cause a hacker to enter your code, you could forget to delete it at the end, giving them easy access to your data. With smart pointers, these objects are automatically deleted when the program terminates, closing any holes that could allow potential hackers from entering.

Code:

```
unique_ptr<string> uptr{ make_unique<String>("Unique Pointer")};
shared_ptr<vector> sptr{ make_shared<vector>({7, 3, 2, 6})};
```

weak_ptr<vector> wptr{ sptr };

3. Use smart pointers… Data members of classes, so when an object is deleted all the owned data is deleted as well (without any special code in the destructor).

Normally, when objects are created in programs, the programmer must manually delete the objects that are no longer in use and must remember to set raw pointers to null. If they forget to do this, it could cause a potential breakpoint for hackers. A useful feature of Smart Pointers is that when the object the pointer is referencing is deleted, the smart pointer is automatically deleted as well. This removes the risk of a programmer forgetting to set their pointer to null. It also removes the cybersecurity risk that would occur because of a hanging pointer.

Code:

```
Unique_ptr<string> uptr{ make_unique<string>("I am UNIQUE!")};
//extra code where uptr is used!
uptr.reset();
//uptr's reference was deleted and now uptr is deleted automatically
```

4. Converting unique_ptr to shared_ptr is easy. Use unique_ptr first and convert unique_ptr to shared_ptr when needed.

It's always the best practice to start your smart pointers as unique pointers because of their smaller size and functionality. Most of the time, only one pointer references a single object, so using a unique_ptr is always the smartest option. However, if you do come to the situation where two pointers reference the same object, then you can change a unique_ptr to a shared_ptr using the move() function. Another way you can do this automatically is by creating a shared_ptr from the unique_ptr that you are initializing.

Code:

```
//option 1:
unique_ptr<string> uptr( make_unique<string>("Silly String"));
shared_ptr<string> sptr(move(uptr));

//option 2:
shared_ptr<string> sptr( make_unique<string>("Sillier String") );
```

5. Use **weak_ptr** for **shared_ptr** like pointers that can dangle.

Weak pointers can only be implemented as a subordinate of a shared_ptr. They don't increment the shared_ptr's incrementation but do serve other purposes. Weak_ptr's don't get dereferenced automatically. To dereference them, you must change them into shared pointers.

This is because weak_ptrs are used to see if the object they point to has been destroyed. They do this using the expire() function. So, if the object they reference has been destroyed, the weak_ptr acts as a dangling pointer and allows you to see if the object has been destroyed yet.

Code:

```
shared_ptr<String> sptr{ make_shared<string>("I will be shared")};
weak_ptr<string> wptr{sptr};
sptr.reset();
wptr.expired();
```

Part C

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

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

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

- shared_ptr<Node<ItemType>> headPtr{ nullptr }: Line 63 in .h
- shared_ptr<Node<ItemType>> 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