Muzaffar Sharapov

PART B

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.

Deleting a pointer twice happens in case two classes share the same heap object. Deleting the same memory twice can lead to unexpected errors. It could not even show it right away and crash later, show runtime error, or it could go through without any errors. People call that "undefined behavior". Int *a = new int(2);

```
string *str_1 = new string;
auto str_2 = str_1; *str_1
= "AAA";
cout<<*str_2<<endl;
delete str_1; delete str_2;
```

Double deleting could happen using share_ptr as well.

```
int *a = new int(2); std::shared_ptr<int> b(a);
std::shared_ptr<int> c(a); Solution to the above code is,
int *a = new int(2); std::shared_ptr<int> b(a);
std::shared_ptr<int> c = b;
```

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.

2. 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).

```
void exFunc() {
unique_ptr<int> uptr (new int(3)); }
```

Which is similar to

```
void exFunc() { int* int
= new int(3);
unique_ptr<int> uptr
(intPtr)
```

When it’s out of exFunc function’s scope then the object is deleted.

4. Converting unique_ptr to shared_ptr is easy. Use unique_ptr needed.

It is easy by using std::move function. For example from a line of code that I used for part C in method addEnd340

unique_ptr<Node<ItemType>> uniqueNodePtr = make_unique<Node<ItemType>>();

Here we can convert uniqueNodePtr to shared ptr by using move. Exemple

shared_ptr<Node<ItemType>> convertedToSharedPtr = move(uniqueNodePtr);

5. Use weak_ptr for shared_ptr like pointers that can dangle

Smart pointers are excellent methods to handle dangling pointer as by using raw pointer it is hard to tell whether the refernced date has been deallocated. One of the way to check dangling pointer in weak_ptr is by using .lock() function. We could also check the data if we call expired() func. Thus, it’s better to use smart pointers than raw pointers. We cannot have a dangling pointer without weak pointer as weak pointer lets us check dangling by using lock().