

Course: CSC 340.04

Student: Kullathon “Mos” Sitthisarnwattanachai, **SFSU ID:** 921425216

Teammate: n/a, **SFSU ID:** n/a

Assignment Number: 04

Assignment Due Date & Time: 05-06-2022 at 11:55 PM

Important

This project was built and tested using CMake 3.21 and C++20. For instructions on how to build and run each part, please **refer to the README file** in the submission folder.

All parts have been tested to produce identical output except where otherwise noted. Screenshots are provided as a sample for each part.

PART A – Linked Bag, 40 points

- Please change only files: **LinkedList340.cpp** and **Include.h**, no other files.
- We are to implement 8 small additional functions and 2 helper functions to the Linked Bag.
 - See PartA_LinkedBag340 folder.
- Our programs must produce **identical** output to the output in the 2 sample runs:
Asmt04_Run1.txt and **Asmt04_Run2.txt**
 - Our Test 9's output must also be **identical** to the sample output excepts the random values.
 - Our Test 9's random values in our 2 sample runs' output must be **different**.
 - See below for screenshots demonstrating the program.

```
Run - CSC340Asmt04
Run: PartA_LinkedBag340 x
"/Volumes/GoogleDrive/My Drive/School/San Francisco State University/Spring 2022/Classes/CSC
340/Assignments/Assignment 04/CSC340Asmt04/cmake-build-debug/PartA_LinkedBag340
/PartA_LinkedBag340"
----- 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
```

```
Run - CSC340Asmt04
Run: PartA_LinkedBag340 x
!Display bag: 4-FOUR 5-FIVE #-END #-BEGIN 4-FOUR 9-NINE 0-ZERO
-----> 7 item(s) total

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

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

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

Process finished with exit code 0
```

Descriptions of the 8 functions:

Please ask questions, if any, during the in-class discussions and demos for this assignment.

1. **removeSecondNode340** deletes the second node in the Linked Bag. **4 pts**
2. **addEnd340** inserts the new node at the end of the Linked Bag. **4 pts**
3. **getCurrentSize340Iterative** counts the number of nodes in the Linked Bag iteratively. **4 pts**
4. **getCurrentSize340Recursive** counts the number of nodes in the Linked Bag recursively. Use 1 helper function:
getCurrentSize340RecursiveHelper. 4 pts
5. **IMMEDIATE RECURSION: getCurrentSize340RecursiveNoHelper**
counts the number of nodes in the Linked Bag recursively. This recursive function **does not** use any helper functions. **8 pts**
6. **getFrequencyOf340Recursive** recursively counts the number of times an entry appears in the Linked Bag. Use 1 helper function:
getFrequencyOf340RecursiveHelper. 4 pts
7. **IMMEDIATE RECURSION: getFrequencyOf340RecursiveNoHelper**
recursively counts the number of times an entry appears in the Linked Bag. This recursive function does not use any helper functions. **8 pts**
8. **removeRandom340** removes a random entry from the Linked Bag. **4 pts**

PART B – Smart Pointers, 15 points

- For each of the following statements, please:
 - Explain the statement in 5 or more sentences. Please think Interviews. And
 - Create a new code experiment to demonstrate our understanding.
 - *Please remember to submit our code and document our experiment in our assignment report.*

See the code experiment in the PartB_Experiments/main.cpp file.
Each function demonstrates the answers for parts (B)(1) through (B)(5).
Refer to the README file for instructions on how to run 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.
 - As raw pointers are unable to discern whether the object it is pointing to is still valid, having multiple pointers pointing to the same object increases the chance for a segmentation fault to occur. Raw pointers rely on its declaration to know the type of its pointee. It cannot, however,

know whether the pointee still exists in the memory. If two pointers `a` and `b` points to an object in the memory and `delete` is applied to either one of them, then both of them will point to deallocated memory. Attempting to dereference or invoke `delete` again on either one will result in a segmentation fault.

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 program exits that function or block.

- Typically, objects that are instantiated with the `new` keyword will be kept alive on the free-store unless deallocated using the `delete` keyword. Failing to use the `delete` keyword when the object is no longer needed consumes unnecessary space on the free-store. This issue can be alleviated by the use of smart pointers. If multiple access is needed across different scopes, one can avoid using the `new` keyword entirely and construct the required object using the `std::make_shared()` function. Instantiating an object this way allows for an object to be kept alive beyond a particular scope. The pointers are “smart” in the sense that they keep track of the number of references and destruct the object automatically when no longer in use.

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).

- The usage of smart pointers can be expanded to class members to create a domino-like effect for when an instance goes out of scope. Normally, to heap-allocate memory for a class member, we would need to make sure that the member is properly deallocated at the end of the instance's lifetime. This would require the programmer to implement some sort of clean up code in the class' destructor. With the use of smart pointers, we can do away with special clean up code in the destructor. Similar to its usage with a class instance, using smart pointers with class members will automatically keep track of the number of references to the member, and destroy it when no longer in use. When an instance goes out of scope (and therefore becomes no longer accessible), then its members would also be out of scope. This means that when an instance is about to be destroyed, its members would be destroyed too since they would also be inaccessible.

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

- One can convert a `unique_ptr` to a `shared_ptr` by using the `std::move()` function. It is more preferable to use `unique_ptr` over `shared_ptr` as there is less overhead. `unique_ptr`s, as the name suggests, deals with managing resources with exclusive ownership. `shared_ptr`s on the other hand, are typically twice in size and incur additional overhead for its control blocks. In addition, as `shared_ptr`s

deal with multiple ownerships, it uses atomic reference count manipulations, which are much slower than non-atomic manipulations. As such, when using smart pointers, one should consider using `unique_ptr` first, and only when needed, convert it to a `shared_ptr`.

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

- `weak_ptr`s are “weak” in the sense that they do not have ownership over its reference. Unlike `unique_ptr` and `shared_ptr`, `weak_ptr` do not directly “manage” an object, but rather *observe* it. They can be considered as a utility of `shared_ptr`, since they are non-owning references to `shared_ptr`s. And because they do not provide ownership over an object, they cannot affect the lifetime of an object and do not use reference counting. This means that `weak_ptr`s are able to “dangle” – in other words, it can persist even after the object is destroyed. Unlike raw pointers however, `shared_ptr`s provide a way to check whether the ownership has expired using the `expired()` member function.

PART C – Linked Bag, Smart Pointers Version, 20 points

- Create a Smart Pointers version of our PART B’s Linked Bag:

1. Please create a copy of our entire PART B solution and name it:

PartC_SmartPointers.

2. Then go through all the files, not just **LinkedBag340.cpp**, and use smart pointers properly where it is possible.
3. In our assignment report, **list the file names and the line numbers in which we use smart pointers. For each smart pointer, explain in 5 or more sentences why it is a proper use.**

- [Node.h, line 24](#)

This line declares the next member to be a unique pointer to a Node instance. In a linked bag, each node points to the next, forming a chain.

As such, in order to traverse a bag or access a particular (non-head) node, one must travel through all the intermediary nodes before arriving at the desired node. It makes sense to use a unique pointer here because each node is owned exclusively by its parent. If a node is no longer accessible, then all of the nodes following it would also become inaccessible. With the use of unique pointers, we can ensure that the resources allocated to nodes that are out of reach are automatically freed in a cascading manner when they can no longer be accessed.

- The next member is mostly used to refer to the next node for traversal. In most cases, it does not need to be changed after initialization, so the `Node::getNext()` function returns the raw

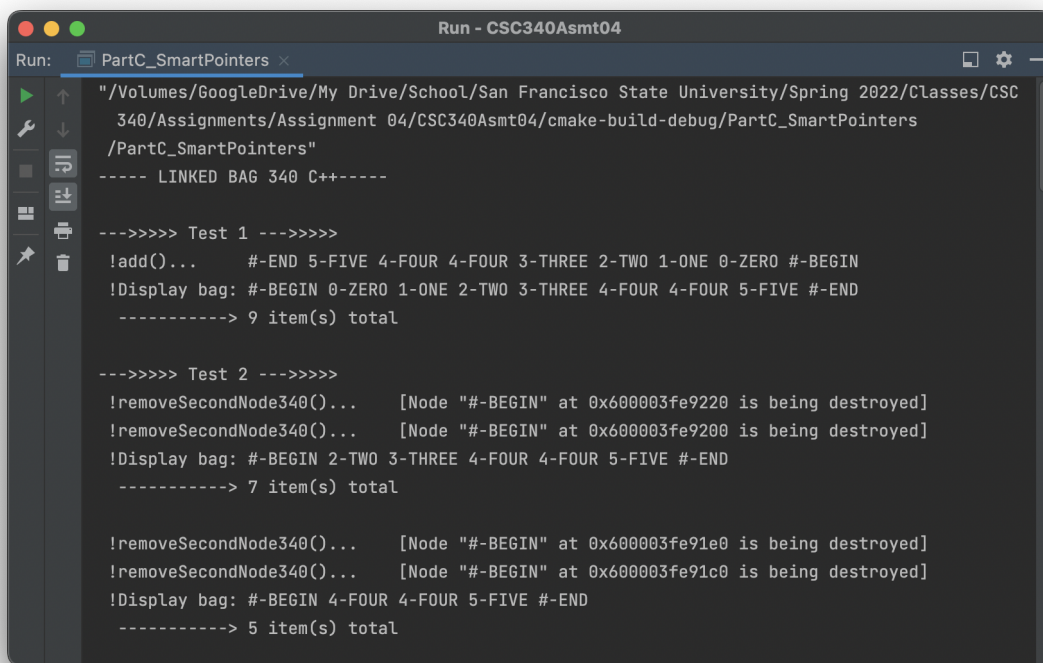
pointer to the following node instead. One notable exception is when a node is being deleted (see [Node::remove\(\)](#)) as the targeted node isn't actually "deleted" but overwritten with data of the head node. The head node is then replaced with its child using `std::move()`.

- [LinkedBag.h, line 42](#)

This line declares the head pointer of the linked bag to be a unique pointer to a Node instance. Similar to how a Node instance exclusively owns the one following it through the `next` member, a linked bag instance exclusively owns the first node ("head node") in the chain. A unique pointer is appropriate here because a linked bag instance should hold exclusive ownership to the head node. As the head node and all nodes in the chain exclusively own each subsequent node, the exclusive ownership to the head node allows a linked bag instance to control all of its data. When a head pointer is no longer accessible or set to `nullptr`, all of the nodes in the chain will automatically be destroyed.

- For the most part, the head pointer is not written to directly. Its primary purpose is to ensure proper memory deallocation when the head pointer is no longer accessible. When traversing through the linked bag, a raw pointer is used instead as it needs to refer to each node during traversal, but does not change it.

4. This Smart Pointers version must work properly and produce identical output like that of our PART B version.
5. In addition, please **update** and add **destructor(s)** so that the program displays more information (in addition to the output required and described above) when object(s) get destroyed.
 - See below for screenshots demonstrating the program.



```
Run - CSC340Asmt04
Run: PartC_SmartPointers x
"/Volumes/GoogleDrive/My Drive/School/San Francisco State University/Spring 2022/Classes/CSC
340/Assignments/Assignment 04/CSC340Asmt04/cmake-build-debug/PartC_SmartPointers
/PartC_SmartPointers"
---- 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()... [Node "#-BEGIN" at 0x600003fe9220 is being destroyed]
!removeSecondNode340()... [Node "#-BEGIN" at 0x600003fe9200 is being destroyed]
!Display bag: #-BEGIN 2-TWO 3-THREE 4-FOUR 4-FOUR 5-FIVE #-END
-----> 7 item(s) total

!removeSecondNode340()... [Node "#-BEGIN" at 0x600003fe91e0 is being destroyed]
!removeSecondNode340()... [Node "#-BEGIN" at 0x600003fe91c0 is being destroyed]
!Display bag: #-BEGIN 4-FOUR 4-FOUR 5-FIVE #-END
-----> 5 item(s) total
```

```
Run - CSC340Asmt04
PartC_SmartPointers x
--->>>> 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
```

```
Run - CSC340Asmt04
PartC_SmartPointers x
-----> 7 item(s) total

!removeRandom340() ---> [Node "4-FOUR" at 0x600003fe9160 is being destroyed] 5-FIVE
!removeRandom340() ---> [Node "4-FOUR" at 0x600003fe9140 is being destroyed] 4-FOUR
!Display bag: 4-FOUR 9-NINE 4-FOUR 9-NINE 0-ZERO
-----> 5 item(s) total

!removeRandom340() ---> [Node "4-FOUR" at 0x600003fe9120 is being destroyed] 9-NINE
!removeRandom340() ---> [Node "4-FOUR" at 0x600003fe91c0 is being destroyed] 4-FOUR
!Display bag: 4-FOUR 9-NINE 0-ZERO
-----> 3 item(s) total

!removeRandom340() ---> [Node "4-FOUR" at 0x600003fe91e0 is being destroyed] 4-FOUR
!removeRandom340() ---> [Node "9-NINE" at 0x600003fe9200 is being destroyed] 9-NINE
!Display bag: 0-ZERO
-----> 1 item(s) total

[LinkedList instance with 1 item(s) is being destroyed] [Node "0-ZERO" at
0x600003fe9220 is being destroyed]
Process finished with exit code 0
```

6. Please remember to submit our code of this part. Save the code under a folder named **"PartC_SmartPointers"** and include this folder in the assignment submission ZIP. *Please remember to document this part in the assignment report.*

- See PartC_SmartPointers folder.

PART D – Linked Bag, Creativity, 5 Extra Credit points

- Please create a copy of our entire PART C solution and name it: **PartD_IamCreative**
- This part is to show off our creative mind. Please implement a new function for Part C's LinkedBag. We need to add code to **LinkedBag340.cpp** and **Include.h** and write **PartD.cpp** to demonstrate how this new function works.
- **Requirements**, this function shall:
 1. Perform **one** meaningful task. Please use the first paragraph of at least 5 sentences in PART D to explain why it is a meaningful task.
 2. Modify the LinkedBag's content every time it runs.
 3. Use Smart Pointers in its parameter list, in its implementation, and as return value(s).

- Our graders expect higher quality in this part: creativity, a meaningful task, clean code, and clear documentation and report.