# COMP 1633 Assignment 4 ("Hockey List", Revisited) – Part 2

| | | |
|---|---|---|
| **Given:** | Monday, March 14, 2022 | |
| **Phase 2 Electronic Copy Due:** | Monday, March 21, 2022 before 11:59pm | ***LATES*** |
| **Phase 2 Hard Copy Due:** | Tuesday, March 22, 2022 before 1:00pm | ***ACCEPTED*** |

## Part 2 Requirements

## In Part 2 you will implement the `HockeyList` class according to the following specifications.

## Unit and Coverage Testing

Note that the submission for this assignment *will* involve the use of `GoogleTest`, `gcov` and `valgrind` to demonstrate code correctness.

You are NOT required to submit a test plan for this assignment, nor a `GoogleTest` module. However, a significant portion of your mark for this assignment **will** be based on the functional and coverage testing of your. Part of the submission process will involve your `HockeyList` class being tested ***via a supplied test module whose source code you will NOT have access to and which provides limited feedback***. If you have not seriously attempted to test your class on your own and it fails the submission testing, you will be *sorely* pressed to determine what is failing! Therefore, it is recommended that you use the following testing strategy. *(Naturally, most of you will likely ignore this excellent advice.)*

If you didn't complete the **`GoogleTest`** (unit testing) lab or the **`gcov`** (coverage testing) lab, you should go back and do so before continuing with this assignment.

While it is tempting to jump right to implementation the `HockeyList` class and then do testing as an afterthought. This can and usually leads to both poor class design and poor testing.

When creating a new class, you should follow the "test first" methodology: develop tests for your public methods in `GoogleTest` **before** actually coding those methods and then make sure your public method code is covered by your tests using `gcov`.

Follow the "test first" methodology: begin Part 2 by creating a test module called `test_HockeyList.cpp`, adding tests for the various public methods of the `HockeyList` class. In order to run your tests you will need to create the list implementation but with the public class methods stubbed out. As such, most, if not all, of your tests will fail. Once you think you a method is fully tested you can implement that method. Ideally your tests will pass and `gcov` will indicate full code coverage. However, the farther you progress in your implementation, the more tests should start to pass. Assuming you have created a comprehensive set of tests, if your tests are all passing, your `HockeyList` is all done!

As you implement your list class, you will think of additional tests. Each time you do, add them to the test module. Also, you will discover bugs in your code as you work. When this happens, add a test to the module which exposes the bug. Then, implement a fix and check that the test now passes. If the bug ever reappears and you're running your module regularly, then you will know immediately.

You should also set your project up so that you can use gcov to analyze the line-by-line coverage of the source code in your "**.**cpp" files. Regularly checking your coverage is a good idea, as it will show sections of your code which are not being tested – and untested code is a breeding ground for bugs.

Running your code with valgrind every so often to check for dynamic memory issues and any other problems valgrind finds is also a good idea.

## Starting Code

In the directory /users/library/comp1633/assignments/a4/p2_start, you will find a solution to Part 1 and a new version of the HockeyList header file. **These files MUST be used as the starting point for Part 2.** The reason for using a common a4p2**.**cpp is so that everyone starts with a working version of the client program. The list header file has been modified to include private data members and a private method that you must use. In Part 2, you will implement the HockeyList class according to the specified implementations.

**Absolutely NO changes can be made to the existing a4p2.cpp nor to the existing items in the provided interface (both public and private)**. ***You may add additional helper functions to ONLY the private section of the class definition. Any added function must be useful and must not duplicate any existing code***. **Failure to follow this REQUIREMENT will result in a significant grade deduction!**

In the HockeyList class, the public member functions must use appropriate private member functions to achieve their goal. Public member functions must not duplicate code that is in an existing private member function.

## Your Makefile

Your makefile ***must*** be set up so that you can build the following targets:

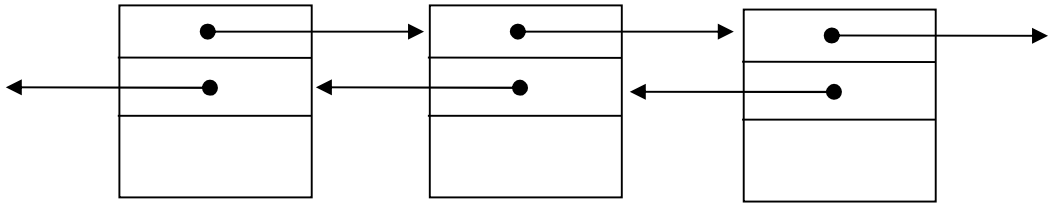
- a4p2 builds your a4p2 program
- clean removes all files which can be auto-generated
- TESTING builds a GoogleTest program for test_HockeyList and executes it, and then executes gcov to display the coverage percentage for HockeyList.cpp
- all does a "clean", then accomplishes the "a4p2" and "TESTING" targets

Hint: pattern your makefile after the example from the unit testing lab.

## Design & Implementation

The `HockeyList` implementation will use ***a doubly-linked list***.

In a doubly-linked list, each node contains a pointer to the "`next`" node *and* a pointer to the "`previous`" node:



Because there are several advantages of doubly-linked lists over singly-linked lists, doubly-linked lists are used much more often in practice. To see one of these advantages, consider having a pointer to an arbitrary node. If the list is *singly*-linked, it is impossible to remove that node or to insert before it (these operations would require updating the *previous* node, but there is no way of backing up to find it). However, if the list is *doubly*-linked, it is possible to remove the node or to insert before it.

As with singly-linked lists, you must be careful inserting/removing at the head/tail. However, there is a common trick to avoid writing extra code to handle working with these boundary nodes: use "dummy" head and tail nodes! These nodes do not actually store meaningful data; instead, they ensure that each "meaningful" node has a neighbouring node (as opposed to "null") on each side.

Because the `HockeyList` class is a container for items of three teams – Canada, Finland, US – as well as an Injury list, we will actually manage items using an array of four lists, one list for each team, plus one for the injured players. This is why the team field was removed from the `PlayerNode` in part 1.

### The Private Section of the Class

The new class declaration contains the following private section of the `HockeyList` class:

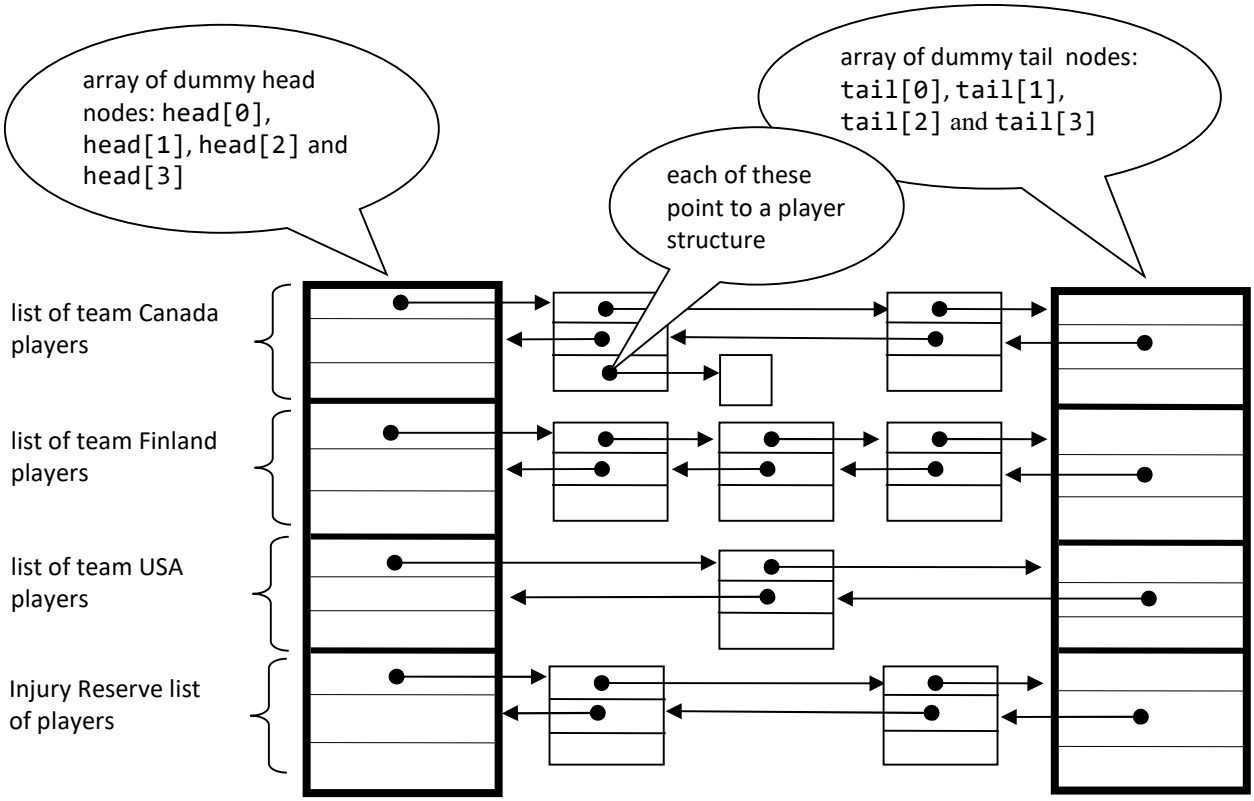
```
struct ListNode
{
  ListNode *next;
  ListNode  *prev;
  PlayerPtr thePlayer;
};

typedef struct ListNode *ListPtr;

int findTeamNumber(char teamLetter);
void search(int pos, ListPtr &item, int &team, bool &found) const;
ListPtr createNode(const char name[], char position, int goals, int assists,
                   int goalsAgainst) const;
void link(ListPtr item, int team);
void unlink(ListPtr item);
void displayTeam(int team, int &position, ostream &out) const;

ListNode head[NUM_TEAMS];
ListNode tail[NUM_TEAMS];
```

## The Private Type Definition and Data Members

Notice an appropriate node type declaration and a `typedef` are nested inside the private section of the class. Being placed inside the private section means that the outside world will not have access to this type. Next, six private helper methods are declared and will be discussed in the next section. Finally, two parallel array data members called `head` and `tail` are declared. Each is an array of "dummy" boundary nodes. Some notes about `head` / `tail`:

- both are arrays of actual `ListNodes` and NOT arrays of pointers to `ListNodes`.
- they are not allocated dynamically, and therefore they must never be deallocated back to the heap. Only the "meaningful" nodes are dynamically allocated from the heap!
- making the `head` and `tail` arrays of ListNodes allows these nodes to be processed like any other in the lists.
- ***The "dummy" boundary HEAD nodes only need to use the next field and the TAIL nodes only need to use the previous field. Using any of the unused fields in the "dummy" boundary nodes to store any information to assist in list processing - including NULLs in pointer fields - is not allowed and violating this <u>will result</u> in a significant grade deduction. Since you always have direct access to the head and tail nodes, identifying these nodes for purposes of terminating processing can always be done by their addresses.***

The following diagram shows how an actual HockeyList looks:




The use of list nodes and player nodes allows separation of the linked structure from the player information. In this way the lookup method can return a copy of the player information to the client program that has no trace of any list information. Thus, the details of list implementation is hidden from the client program and the client is also unable to inadvertently alter the list structure.

Review the previous diagram and determine what an empty list will look like: your constructor needs to create this situation.

**Very important**: when working out the detailed design of the doubly linked list algorithms, *be sure to draw pictures*! Code that rewires "`next`" and "`previous`" pointers during node insertion/removal can be hard to follow unless it is traced out with the aid of a diagram.

**The Private Helper Methods**

The private helper methods are not part of the class interface from the client's perspective. Instead, they exist to be used by other methods of the class:

| Method | Description |
|---|---|
| `findTeamNumber` | *Uses the constant team letters array to return the team number (i.e. array index) for a given team letter. You can assume the provided letter is valid.* |
| `search` | *Searches for a list node at the given position. This function has three by reference parameters: a found flag, a pointer to the node at the specified position and team number corresponding to the team where this node is found. If a node is found at the specified position then the found flag will have the value of true and the node pointer and the team number will be valid; otherwise found will be false and the node pointer and team number are undefined.* |
| `createNode` | *Creates a new list node that points to a new player node and returns a pointer to the list node (at this time this new node will not be linked into the list data structure).* |
| `link` | *Links an existing item node into the data structure.* |
| `unlink` | *Unlinks an item node from the data structure, but does not deallocate memory.* |
| `displayTeam` | *Displays all players of the given team to the given output stream. Notice that there is a public* `display` *method and this private* `displayTeam` *method. The public method is responsible for generating the entire list of all team names and players. This method is responsible for printing the players on a single team. Observe that the position parameter, the list position of the players, is passed by reference.This is necessary so that the list positions can continue in correct order from one team to the next.* |

If the above functions are working properly, then implementing the public methods by calling the above should be straightforward. Therefore, invest lots of time getting these correct and testing them. Realize that these are **private** methods, so can NOT directly be tested via your unit tests, since test code only has access to the **public** interface. However, as novice unit testers, there is a simple workaround for this restriction: temporarily make the private methods public! Once satisfied they are working correctly, make the methods private again, remove the test cases from the unit testing module **AND THEN add test cases to the public methods so that your coverage remains unaltered!** Remember from the first section that the goal is 100% coverage of the class being tested.

## More on Documentation

The following documentation is required:

- NO changes are to be made to the `a4p2.cpp` file, this includes NOT updating the ID block.
- An ID block is required to be added to the specification file, i.e. the `HockeyList.h` file, and the implementation file, i.e. the `HockeyList.cpp` file.
- As well, a header block **is** required at the top of the implementation file that briefly describes the private data members. Also, a method description, like the definition class, but only for each private member function.

## Submission

Submission instructions will be provided on Friday March 18, 2022.