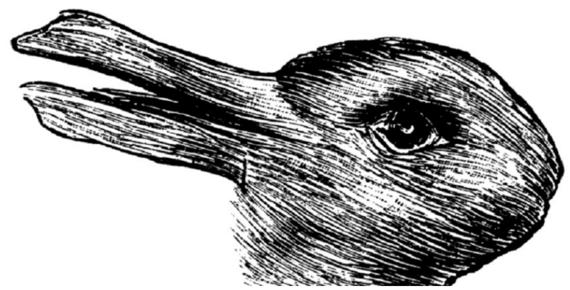
One time I was a worrying duck.
But now I just don't give

you just gotta love a duck

The duck that was...

The platypus quest - I hope you guys still remember it from 2A. If you had aced it, this quest should be a piece of cake. Otherwise, this is your first major challenge of 2B. Make sure you're solid with what it takes to solve it before proceeding to the next one.



As you solve this quest, you'll find yourself leveraging a lot of the knowledge and skills you learned in CS2A. You will also learn a few new concepts (e.g. inner classes, overloading of operators, etc.) Where necessary, I simply apply these concepts in starter code I give you. You are free to dive into it to any level of detail and clarify them in the forums (I'll help you), or you can simply copy my starter code as-is and follow the directions in the spec for now. The skipped concepts will make more sense in about a week.

Some miniquests in this quest will give you rewards for meeting a minimum requirement, and give you additional surprise rewards for doing something else right that wasn't explicitly asked.

Overview

In this quest you will implement a class called `Playlist` as a singly linked list of nodes with `Song_Entry` objects as their payloads. Your code should be organized into a header file (`Playlist.h`) and a separate implementation file (`Playlist.cpp`).

Your `Playlist` class will encapsulate things the client (the user of the class) doesn't have to know about. Make them inner classes. An inner class is simply a class that is defined inside another class - duh! If an inner class is private, then only the containing class can create or manipulate objects of that class. If it is public, then anyone can instantiate the inner class and invoke its public methods. But they'd have to access the class name with the outer-class qualifier.

For example, if `Song_Entry` is a public inner class of `Playlist`, a user, in their `main()` method, can refer to it as `Playlist::Song_Entry`. This kind of structuring also gives a nice namespace-like separation of the type from other classes elsewhere that may have the same name.

You will have two inner classes in your `Playlist` class:

- `Playlist::Song_Entry`, which is public
- `Playlist::Node`, which is private

You don't have to modify anything in the structure (declaration) of the above inner classes. But you will have to flesh out some of the method implementations in them.

Keep the payload class, `Playlist::Song_Entry`, simple because that's not the focus of this quest. Define it as follows:

```
// This class def should be placed in the public section of Playlist

class Song_Entry {
private:
    int _id;
    string _name;

public:
    Song_Entry(int id = 0, string name = "Unnamed")
        : _id(id), _name(name) {}

    int get_id() const { return _id; }
    string get_name() const { return _name; }

    bool set_id(int id);
    bool set_name(string name);

    bool operator==(const Song_Entry& that) {
        return this->_id == that._id && this->_name == that._name;
    }
    bool operator!=(const Song_Entry& that) {
        return !(*this == that);
    }

    friend std::ostream& operator<<(ostream& os, const Song_Entry& s) {
        return os << "{ id: "<< s.get_id() << ", name: " << s.get_name() << " }";
    }
    friend class Tests; // Don't remove this line
};
```

Perform some basic data validation in the setters. Don't set negative IDs or empty song names. Make the setters return true on success and false on failure.

Define the `Playlist::Node` inner class as follows:

```
// This class def should be placed in the private section of Playlist
// at an "appropriate" location. (What might be an inappropriate location?)
class Node {
private:
    Song_Entry _song;
    Node *_next;

public:
    Node(const Song_Entry& song = Song_Entry()) : _song(song), _next(nullptr) {}
    ~Node(); // Do not do recursive free

    Song_Entry& get_song() { return _song; }
    Node *get_next() { return _next; }
    Node *insert_next(Node *p);
    Node *remove_next();

    friend class Tests; // Don't remove this line
};
```

Let's discuss the preceding fragment briefly before moving on. It defines a class called `Node`, which contains a `Song_Entry` member called `_song` and a pointer member (a memory address) called `_next`.

What does `_next` point to? Its value is the memory address at which a `Node` can be found. It might even be its own address - that is, point to itself. That's the cool thing about creating linked data structures. Not only can you jump all over the place in your memory, you can also define *self-referential structures*, which is a big deal.

> **Important Note**
>
> In a previous version of this spec, when p is a pointer, I used to say "*p points to*". But that made me say things like the "header points to" and this confused students who couldn't tell if I meant _head->next or _head itself. To avoid this confusion, I've decided to use the word *equals*.
>
> In this document, when I say the _prev_to_current equals something, I mean that the contents of the pointer, _prev_to_current, equals it. You can think of it as *overloading* our current understanding of the word *equals* with an added meaning.
>
> c++ lets us do cool things like this in programming too!

A few special things about Node

`Node::get_song()` returns a reference to its `Song_Entry` object - not a copy. Why? Because that's the only way you can easily modify the contents of an existing node in this implementation. Extra credit rewards await meaningful discussion of this point, as well as thoughts on other ways in which you might change, say, the 5th song in a list from A to B.

`Node::get_next()` simply returns the node's `_next` pointer.

Besides a `Node's` getters and setters, note the two special methods: `insert_next()` and `remove_next()`.

You will invoke these from your `Playlist` instance methods. Since the idea of a `Node` needs to be *abstracted away* from your end-user (the programmer who uses your `Playlist` class), none of the `Playlist's` public methods will ever communicate in terms of Nodes with the outside world. Instead they will interface using `Song_Entry` objects - inserting songs, removing songs, and so on. However, these methods that insert and remove songs will, opaquely to the user, themselves use their private `Node` insertion and removal methods.

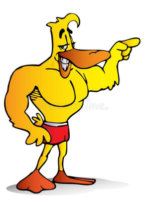
`Node::insert_next(Playlist::Node *p)`

should insert the given node, `p`, directly in front of itself. That is, after the operation, its own `_next` member should be pointing at `p`. It should return `p`, the pointer to the node that just got inserted.

`Playlist::Node *remove_next()` should unlink and remove the node pointed to by `_next`. Refer to a diagram later in this document to see what needs to get done. But here is an important question before you start cutting its code. What is the node that `remove_next()` returns? Is it the node that just got unlinked so the caller can extract and use the value that it pulled out of the list?

No. It returns a pointer to itself (the `this` value). What is a good reason for doing so?

Remember that the remove operation also deletes the node it unlinks (frees its memory). So you cannot return a node which doesn't belong to you any more. But one may object, saying that it makes sense to return the unlinked node to the caller and also shift the responsibility of deleting the node to the caller. Does this alternative approach make sense? Discuss the pros and cons (including esthetic reasons) for doing it one way versus the other (extra rewards may await insightful discussions).



After you delete a successor node for a given node, make sure to set its value to `NULL` for easy debugging. This way you're sure to know which nodes are pointing to allocated memory. (There's one more important thing to keep in mind here. Look for my picture elsewhere in this document to know what that is.)

Finally, here is the `Playlist` class. Heed the header comment

```
// Important implementation note: With the exception of to_string() and find...()
// all Playlist methods below should operate in a constant amount of time
// regardless of the size of the Playlist instance.
//
// The semantics of prev_to_current is such that it always points to the
// node *BEFORE* the cursor (current). This makes the manipulations easy because
// we can only look forward (and not back) in singly linked lists.

class Playlist {
public:
    // Inner public class ---------------------------------------------------
    // The client can refer to it by using the qualified name Playlist::Song_Entry
    class Song_Entry {
        // TODO - your code here
    };

private:
    // This is going to be our inner private class. The client doesn't need to
    // know.
    class Node {
```

```
        // TODO - your code here
    };

private:
    Node *_head, *_tail, *_prev_to_current;
    size_t _size;

public:
    Playlist();
    ~Playlist();

    size_t get_size() const { return _size; }
    Song_Entry& get_current_song() const;

    // The following return "this" on success, null on failure. See the spec
    // for why.
    Playlist *clear();
    Playlist *rewind();
    Playlist *push_back(const Song_Entry& s);
    Playlist *push_front(const Song_Entry& s);
    Playlist *insert_at_cursor(const Song_Entry& s);
    Playlist *remove_at_cursor();
    Playlist *advance_cursor();
    Playlist *circular_advance_cursor();

    // The following return the target payload (or sentinel) reference on success
    Song_Entry& find_by_id(int id) const;
    Song_Entry& find_by_name(string songName) const;

    string to_string() const;

    friend class Tests; // Don't remove this line
};
```

On returning a pointer to the list

Note that some of the public list manipulation methods return a pointer to the current list object. Why do we do that? We do it because it allows us to program using a very nice pattern:

```
my_list
     .push_back(song_1)
    ->push_back(song_2)
    ->push_back(song_3)
    -> . . .
;
```

In fact, it would be better if you returned a reference to the current object rather than a pointer. Then you can write the even nicer pattern:

```
my_list
    .push_back(song_1)
    .push_back(song_2)
    .push_back(song_3)
    . ...
;
```

I'll leave you to experiment with the esthetics and find out more. But in this quest, you're gonna have to return a pointer to the current object (essentially **`this`**).

On being clear where your head is

Note that you are guaranteed to find a head node in any valid list with this approach. The head node is always equal to the very first actual node of the list. It is not a data node, and we will use it to double as a sentinel (see next section).

`_prev_to_current` is always equal to the node immediately *before* what we call the current node (by design). You can think of it as the *cursor* in this linked list. It always points to (or has a next element that is equal to) the node that we define as *current*. All list operations are done at the location of this cursor.

I think one of the best ways you'll get to appreciate the utility of this member is to try and program this quest without using it. This member saves you from having to scan the list from the beginning just to find where "here" is. If you use this member, then "here" (the current location) is always directly in front of your `_prev_to_current` node. This will make a whole lot more sense once you've programmed this functionality both ways. I highly recommend you do so (for your own edification).

On sentinels

Some of the `Playlist` methods return a `Song_Entry` object. What would you do if the requested operation failed and you don't have a `Song_Entry` object to return? For example, what would you return if I invoked your `find_by_id()` and passed it an non-existent song ID?

The most common way to handle such a situation is using what are called exceptions. However, we don't cover exceptions until a week or so from now. So until then, we'll use a stopgap measure and make the method return a sentinel. A sentinel is a known invalid object.

In this case, conveniently use the header node in a linked list as a sentinel. When you create a header node, make sure the object in it has an id of -1. This is the sentinel and will never change. The user only gets to see what `_head->next` points to and downstream from there.

On to the miniquests ...

Your first miniquest - Know your `songs, nodes and lists`

Implement the following public methods:

- `Song_Entry::set_id()`
- `Song_Entry::set_name()`
- `Node constructor`
- `Playlist constructor`

Make sure you read about the special things to do with `Nodes` (earlier in this document) before you start implementing.

The `Playlist` constructor should create an empty `Playlist` that matches Figure 1. So it's easy rewards. Go for it.

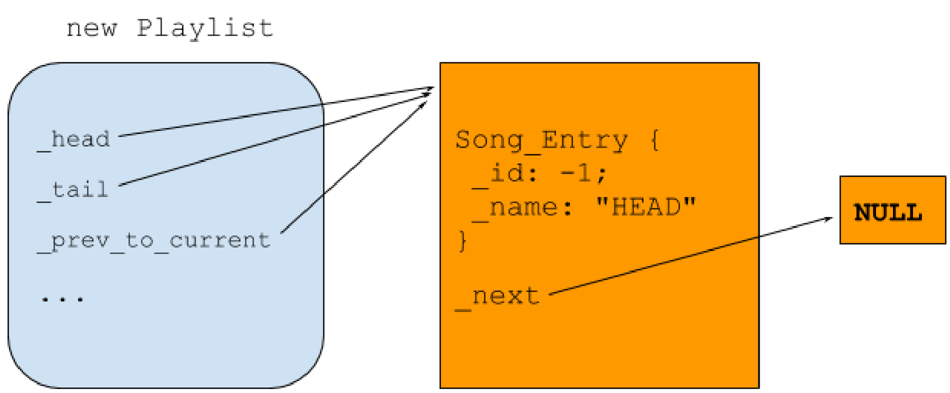


Figure 1

The head node should be a `Song_Entry` object with an id of -1. Remember that you cannot set negative ids in a `Song_Entry` object using a setter on it. The only way to do it would be at construction time (that is, when you create the head node).

Your second miniquest - Destructors

Since your `Playlist` constructor is going to allocate memory on the heap, you must have a destructor or you will suffer memory leaks. Implement

```
Playlist::~Playlist();
```

Make sure your destructor clears out the linked list by freeing all downstream nodes first and then deleting `_head` (which got allocated in the constructor). To accomplish this you would delete `_head`, which should trigger `Playlist::~Node`, the `Node` destructor. However, once invoked, the `Node` destructor will be called, and it should iteratively peel off one node at a time (adjacent to `_head`) and `delete` (free) it. When all downstream nodes have been released thus, the `Node` destructor should delete `_head` and return.

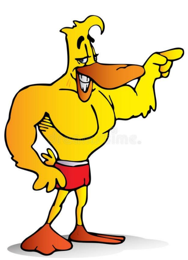
Note that there is no guarantee a memory location won't be overwritten the moment you delete the object to which it belongs. Thus, before you delete a node's descendants (starting at its `_next` member), make sure that you have taken a copy of that member into your local variable. Don't try to access it AFTER the node has been deleted. Then you will likely encounter unpredictable errors that cause your code to work sometimes and not other times.

Carelessness here has resulted in forum posts that went "*I'm sure I had it working, but something must have changed on the testing site because it's not working any more.*"

Esthetics

A good rule of thumb I use when I have questions about where to deallocate memory is this: If the memory was allocated in the constructor, then it should only be deallocated in the destructor. In general, I try and match up my allocations and deallocations into symmetric locations in the causal chain leading up to some action and back.

Also, keep in mind that you should not delete the "`this`" object. It is the deletion of the current object that triggers its destructor to be called. In here, you should only delete downstream nodes.



This paragraph will likely save you several hours of debugging frustration: When you destroy a node, you will have to delete its `_next` member. This means that a node's deletion will auto-invoke the delete on its _next pointer and so on. You want to make sure you delete no more than you should. Therefore make sure to (1) set every pointer you delete to `nullptr` so you can check before you attempt to double-delete something and (2) clear out a `Node's` _next pointer (set it to null) before you delete the node. This will prevent the delete from cascading to affect all downstream nodes.

Your third miniquest - Insert song at current location

Implement:

```
Playlist *Playlist::insert_at_cursor(const Song_Entry& s);
```

When I invoke it, I will pass it a `Song_Entry` object as its argument. It should insert *a copy of* that object at the *current location* of the `Playlist` object. It should leave the current location unchanged. This means that your `_prev_to_current` member will now end up pointing to this newly inserted `Node`.

This is a tricky method and is worth getting correct before you move on. Refer to the picture in Figure 2 to see what needs to happen:

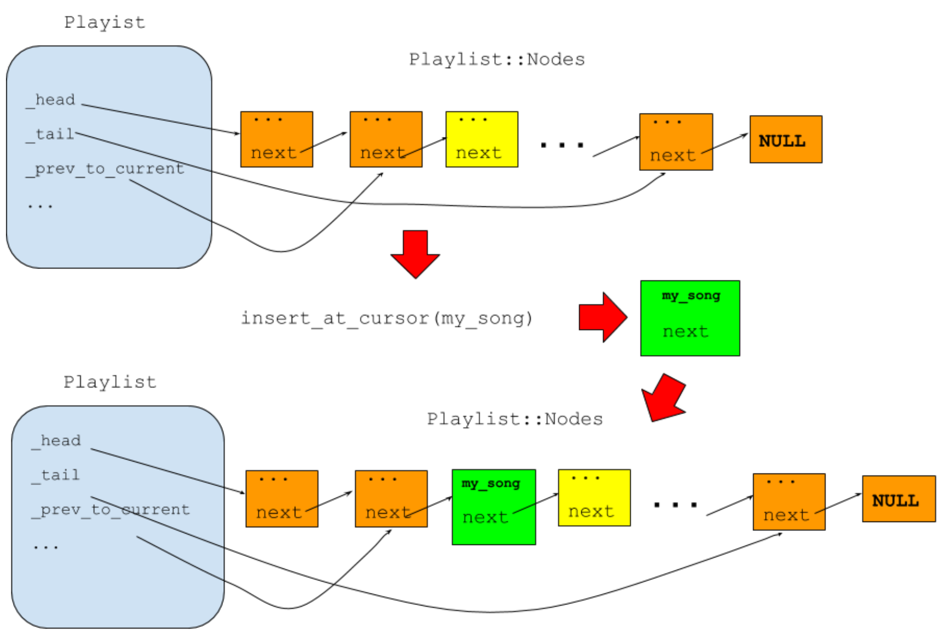


Figure 2 - insert at cursor

Your fourth miniquest - Push back

Implement:

```
Playlist *push_back(const Song_Entry& s);
```

This method must insert (a copy of) the song `s` as a brand new node at the end of the list. That is, this newly allocated `Node` will become the new `_tail`.

Since you have already implemented `insert_at_cursor()`, simply use it to complete this mini quest.

1. Save the current value of `_prev_to_current`, then set it to your `_tail`.
2. Then insert the given string at the current position (which will now be the tail).
3. Finally restore the value of `_prev_to_current` to your saved value.

Before you implement this, convince yourself that this works by simulating the above sequence of steps on a piece of paper with a pencil.

Your fifth miniquest - Push front

Implement:

```
Playlist *Playlist::push_front(const Song_Entry& s);
```

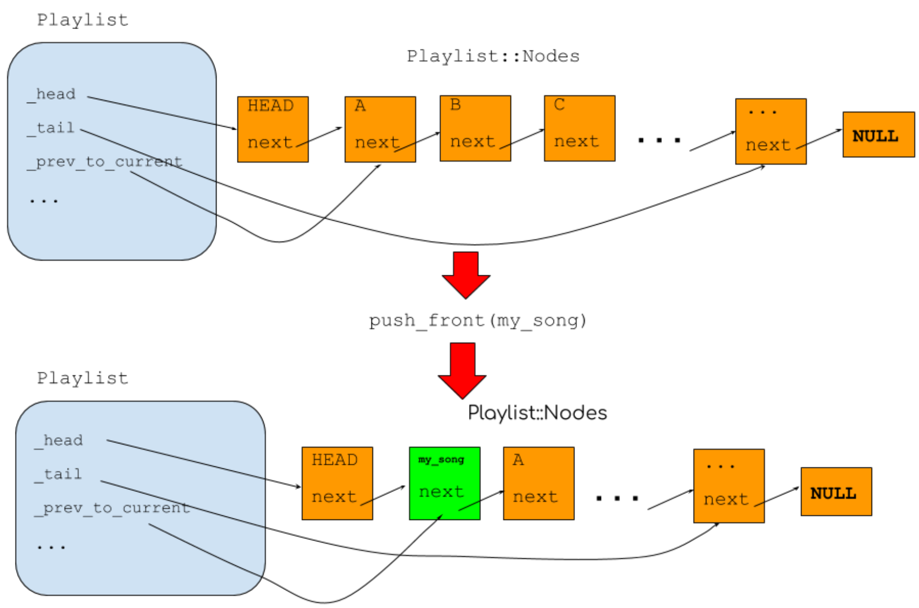


Figure 3 - push front

This method must insert the song s as a brand new data node at the front of the list. That is, this newly allocated `Node` will become the node that `_head->next` points to.

Implement it using the same strategy as for `push_back()`

Your sixth miniquest - Advance

Implement:

```
Playlist *Playlist::advance_cursor();
```

If `_prev_to_current` is the same as the tail node, then obviously you can't advance to the next element (there is none). In that case you would return a null pointer (`nullptr`). Otherwise, make `_prev_to_current` point to whatever its `_next` member was pointing to.

Important! Remember that `_prev_to_current` can never be null because it's at least pointing to the list header. The current element visible to the user is always the element AFTER prev_to_current. So, when you're trying to advance the cursor and it's pointing to the element before `_tail`, you return null (Why?)

Your seventh miniquest - Circular Advance

Implement:

```
Playlist *Playlist::circular_advance_cursor();
```

This is essentially the same as advance_cursor() EXCEPT for the fact that if the cursor is pointing to the very last data node, the advance will silently reset to point to the first node. Refer to the picture below to see what needs to happen:

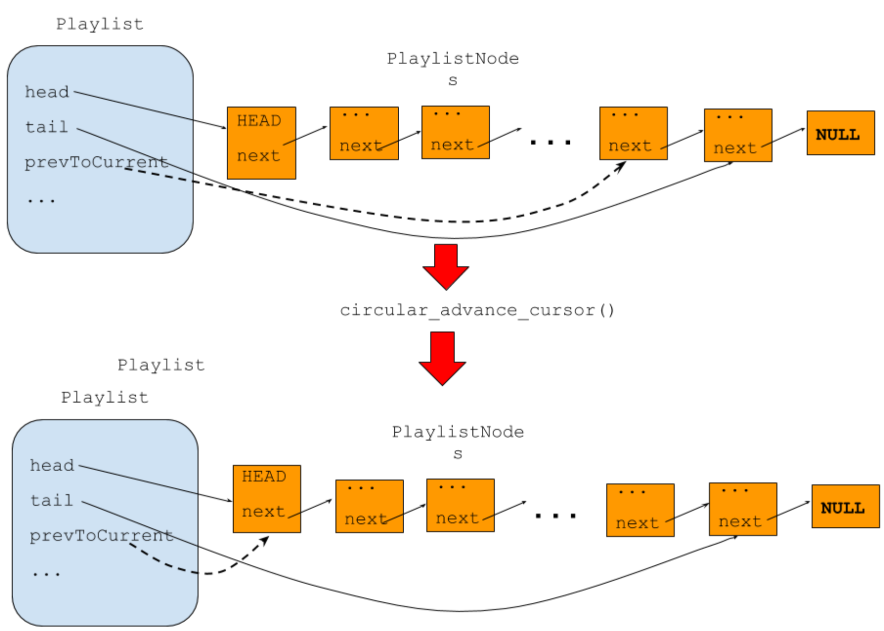


Figure 4 - circular advance

Your eighth miniquest - Get current song

Implement:

```
SongEntry Playlist::get_current_song();
```

If a `_next` element exists, then simply return its _song member.

If no next element exists (this could happen if `_prev_to_current` is the same as your tail node), then you must return the sentinel (`this->_head`->_song)

> This method is only a few lines, but make sure you understand it fully. Ask in the forums if any of it is unclear.
> The current item is the song member of whichever node is equal to `_prev_to_current->_next`.
>
> If `_prev_to_current->_next == nullptr`, which should not happen, we would ideally throw an exception. But since we haven't covered exceptions yet, we will return the sentinel.

Your ninth miniquest - Remove song at cursor

Implement:

```
Playlist *Playlist::remove_at_cursor();
```

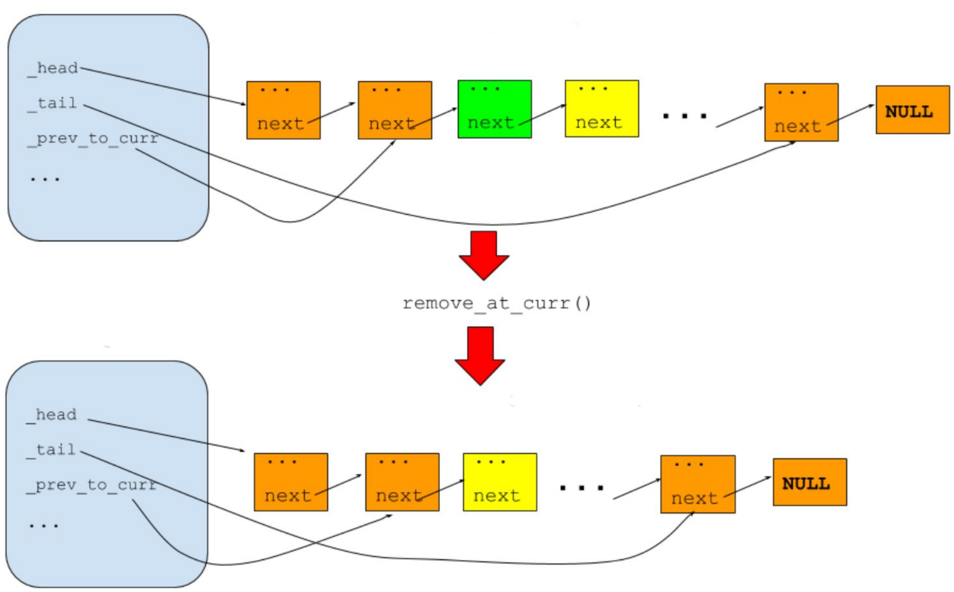


Figure 5 - remove at current position

Again, this is a tricky method and important to get right. Refer to the picture in Figure 3 to decide exactly what needs to happen in your code.

Your tenth miniquest - Get size

Implement:

```
size_t Playlist::get_size() const;
```

I can't believe you're gonna get a reward for just reporting the value of size!

Still, I guess this is your first quest this quarter and you gotta have some freebies…

Your eleventh miniquest - Rewind (but not relax just yet)

Implement:

```
Playlist *Playlist::rewind();
```

This should reset `_prev_to_current` back to the `_head` node.

Your twelfth miniquest - Clear

Implement:

```
void Playlist::clear();
```

Yet another tricky method to get right. When invoked, it must

1. Iteratively (not recursively) delete all the non-null nodes in the chain starting at `_head->_next` (Talking points: What might the problem be with a recursive delete instead?) Consider using the `Node` destructor you wrote.
2. Reset _prev_to_current and `_tail` back to `_head`
3. Set the value of `_head->next` to `nullptr`

Note that you are not deleting the memory associated with the _head node itself. That is the job of the destructor. What the constructor created, only the destructor should undo.

Your thirteenth miniquest - Find an item

Implement the following *linear-search* methods:

```
Song_Entry& Playlist::find_by_id(int id) const;

Song_Entry& Playlist::find_by_name(string id) const;
```

Note an important aspect of the signature. They return references to Song_Entry objects. Not copies. If the requested song exists in the list, then what gets returned is a reference to the actual data element. That means that if I assign something to this reference, it will change the contents of the list node that contains that song.

It's important to understand exactly what that means. Please do discuss it in the forums and help each other out whenever possible. Ask me if you're stuck.

What will this method return if the requested song is not found in the list? In that case, return a sentinel in the same way you did for `get_current_song()`.

Your fourteenth miniquest - Stringify

Implement:

```
string Playlist::to_string() const;
```

This method must return a string representation of the list of strings in the following exact format. I've colored in the spaces. Each colored rectangle stands for exactly one space.

```
Playlist: [N] entries.
{ id: [id1], name: [Name of 1st song] }
{ id: [id2], name: [Name of 2nd song] }
. . .
{ id: [idk], name: [Name of kth song] } [P]
. . .
{ id: [idN], name: [Name of last song] } [T]
```

The parts in red above (also in square brackets) must be replaced by you with the appropriate values.

The `_prev_to_current` element, *if visible*, should be marked with a `[P]` tag (see above).

Similarly The very last element (the tail), if visible, should be marked with a `[T]` tag.

You would print a maximum of 25 elements this way. If the list has more than 25 strings, you must print first 25 and then print a single line of ellipses (`...`) in place of the remaining elements.

There is one newline after the last line (which may be ellipses).

Here's a point to ponder and discuss. Why do we need to say "*if visible*" when talking about the cursor above?

Review

Now I recommend that you rewind and review the specs one more time making notes along the way. It will help you when you start cutting code.

Starter code

There is no starter code in this quest. You're on your own in the wildies (actually, that's not true, if you've been reading along).

Testing your own code

You should test your methods using your own `main()` function in which you try and call your methods in many different ways and cross-check their return value against your expected results. But when you submit you must NOT submit your `main()`. I will use my own and invoke your methods in many creative ways. Hopefully you've thought of all of them.

Submission

When you think you're happy with your code and it passes all yourown tests, it is time to see if it will also pass mine.

1. Head over to https://quests.nonlinearmedia.org
2. Enter the secret password for this quest in the box.
3. Drag and drop your `Playlist.h` and `Playlist.cpp` files into the button and press it. (Make sure you're not submitting a `main()` function)
4. Wait for me to complete my tests and report back (usually a minute or less).

Points and Extra Credit Opportunities

Extra credit points await well thought out and helpful discussions.

May the best coders win. That may just be all of you.

Happy Hacking,

&