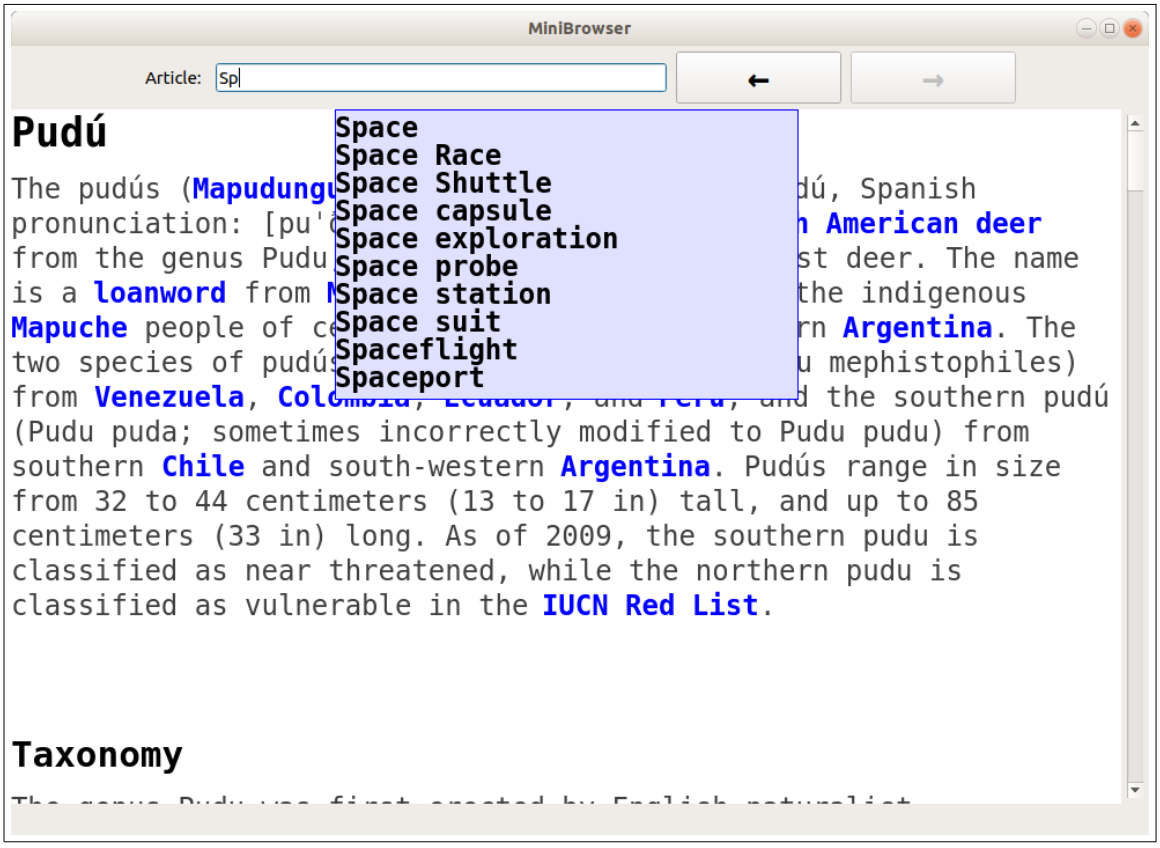
Assignment 6: MiniBrowser

Assignment by Ashley Taylor and Keith Schwarz with additions by Nick Troccoli

Web browsers are remarkable pieces of software built using clever algorithms data structures. In this assignment, you'll implement three linked structures that, collectively, power the major components of a small web browser hooked up to Wikipedia. By the time you've finished this assignment, you'll have a much better understanding of how linked structures get used in practice and will get a little window into the beautiful ideas that power modern software systems.



This assignment consists of three smaller pieces:

- ***Browser History***: Web browsers have forward and back buttons you can use to navigate across the chain of pages you've visited. How is that information stored? By being creative with linked lists, it's possible to implement this simply and elegantly.
- ***Autocomplete***: Most web browsers (and many websites) can look at what you're typing in a search bar and provide suggestions about what you might be searching for. Through some creative use of the trie data structure, it's possible to provide search suggestions almost instantaneously.
- ***Line Manager***: Have you ever stopped to think about how computers draw text on a screen and let you click on hyperlinks? Somewhere in the browser there's logic to determine what you're hovering over and what text needs to get displayed based on the position of the scrollbar. Your task is to implement this using modified binary search trees.

As always, we recommend making slow and steady progress on this assignment. And, of course, feel free to ask questions in the LaIR!

Due Friday, March 8th at the start of class.
You are encouraged to work in pairs on this assignment.

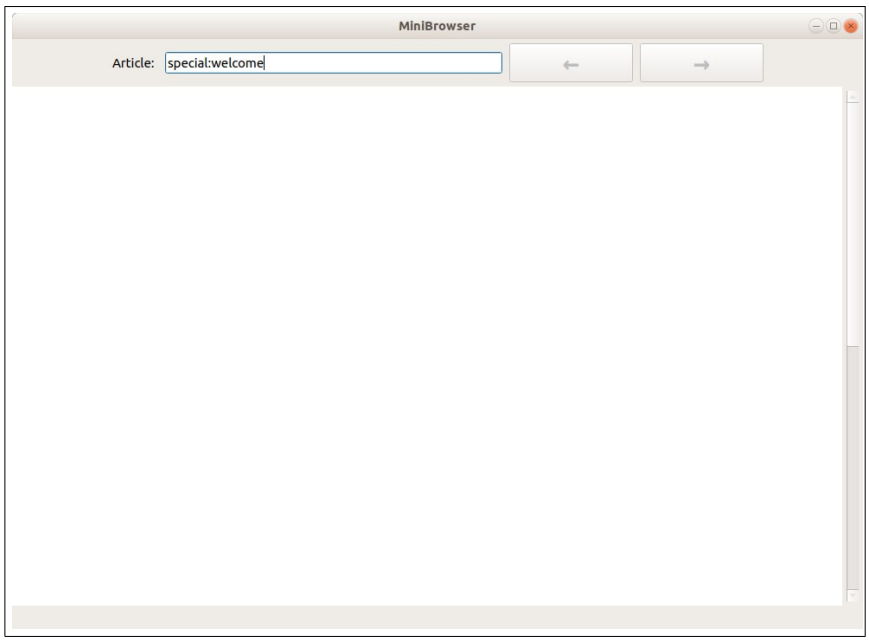
The Starter Files

The starter files we've provided for this assignment are arranged slightly differently than in previous assignments, since we'd like you to end up with a super flashy final product at the end.

When you run the starter files as-is, you'll be presented with a window containing all the results from the test cases. By default, running the program will trigger tests rather than opening the browser itself.

If you'd like to activate the web browser, open `Main.cpp`. You'll see that it contains a `bool` called `shouldRunTests`, which is, by default, set to `true`. If you change this to `false` and run the program, it'll run the web browser instead.

Without anything implemented, the web browser will look something like this:



You'll see the start page (`special:welcome`) in the Article bar. You won't see text yet, since you haven't implemented the logic to display text on the screen. The forward and back buttons will be disabled, since you haven't implemented the browser history. Additionally, as you type text into the Article bar, you won't see any autocomplete suggestions – again, because you haven't implemented autocomplete.

This assignment has three parts, and you can actually implement them in any order that you'd like.

- ***Browser history***: Using doubly-linked lists, keep track of which pages should be visited when the user hits the backward and forward buttons. This part of the assignment is probably the simplest, so we recommend starting here. You'll want to edit the `History.h` and `History.cpp` files. Plan to spend at most two days on this part of the assignment.
- ***Autocomplete***: Implement the browser's autocomplete by using the trie data structure you saw in class. This one has a nice "blast from the past" feel to it in that it combines recursive enumeration with data structures. You might want to implement this one second, though if you'd like more of a warm-up on trees, feel free to start with the line manager. The autocomplete system is in the `Autocomplete.h` and `Autocomplete.cpp` files. We expect this will take you about three or four days to complete.
- ***Line manager***: Write some clever binary search tree manipulations to determine what section of the article should be displayed in the window and where the mouse is at any given time. The coding difficulty here is roughly on par with autocomplete, but getting this one working requires a bit more conceptual understanding of how the browser works. We suspect most of you will implement this one last, but it's certainly possible to do it right after getting `History` working. The code for this section goes in `LineManager.h` and `LineManager.cpp`. As with `Autocomplete`, this should take about three or four days to complete.

A Note: On Building Larger Systems

Every assignment we've provided to you ships with a good amount of starter code (the testing harness, the graphical display, etc.), but this one has a good deal more than usual. That's because a web browser has a bunch of different components in it. There's logic to download pages from the internet, logic to lay out text on a page, logic to display that text, etc. This might seem overwhelming when you're first settling into this assignment, but it's actually not that bad once you get acclimated. Real web browsers can run at hundreds of millions of lines of code – way beyond what any one person can keep in their head – yet people still manage to write and maintain them. The reason why is that the code is broken down into smaller components, each of which is managed by a different team. Those teams don't need to know how everything else works; they just need to know what's expected of their pieces and how those pieces interface with others.

In this assignment, you'll be tasked with implementing three subsystems within the browser. Some of those pieces will operate more or less independently of the other components, and some will require you to have some knowledge of how the rest of the browser works. This handout goes into detail about what pieces are responsible for doing what.

Working on a larger system can sometimes give you a feeling of "I'm not sure how all these parts come together." You'll get your component working, and suddenly the whole thing is operational. Or perhaps you break something, and the impact is bizarre behavior downstream. That's normal – it's just a part of contributing to a larger codebase.

The rules of how your code should work are the same as before. Before you start writing any code, make sure you understand what the code is supposed to do. Test like crazy using unit tests to make sure that your code behaves as expected. And then admire your creation at work!

A Note: Where's the Documentation?

This assignment handout contains an overview of what you need to do for the assignment, including some specifics about what various functions need to do. However, this handout is not designed to be a complete, standalone guide to the assignment. Much of the documentation of how the types you'll implement are supposed to work can be found in the header files for those types. When in doubt, check the headers!

A useful technique: in Qt Creator, if you place the text cursor over the name of a class, function, or type and then hit `F2` (on Macs, `fn + F2`), it'll jump you to the place where that object was declared. This can make it a lot easier to pull up information about a particular member function or nested type.

Browser History

The `History` object which keeps track of the browser history. Here's the interface:

```
class History {
public:
    History();
    ~History();

    void goToNewPage(const std::string& page);

    bool hasForward() const;
    bool hasBackward() const;

    std::string goForward();
    std::string goBackward();

private:
    /* ... discussed below; mostly up to you! ... */
};
```

The rest of this section explores what these functions are supposed to do.

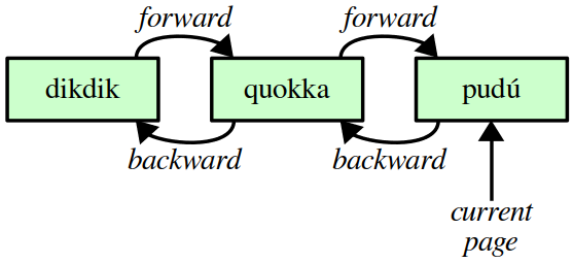
> ***Endearing C++ Quirks, Part 1: `string` versus `std::string`***
>
> Inside header files, you have to refer to the string type as `std::string` rather than just `string`. Turns out that what we've been calling `string` is really named `std::string`. The line `using namespace std;` that you've placed at the top of all your `.cpp` files essentially says "I'd like to be able to refer to things like `std::string`, `std::cout`, etc. using their shorter names `string` and `cout`." The convention in C++ is to not include the `using namespace` line in header files, so in the header we have to use the full name `std::string`.
>
> Think of it like being really polite. Imagine that the `string` type is a Supreme Court justice, a professor, a doctor, or some other job with a cool honorific, and she happens to be your sister. At home (in your `.cpp` file), you call just call her `string`, but in public (in the `.h` file), you're supposed to refer to her as `std::string`, the same way you'd call her Dr. `string`, Prof. `string`, Justice `string`, or whatever other title would be appropriate.

Whenever the user visits a new page, the browser will call the `goToNewPage` function, passing in the name of that page, for your browser history to record. The remaining four functions allow us to navigate forwards and backwards through the history. The functions `hasForward` and `hasBackward` return whether there is a page to go forward or back to. The functions `goForward` and `goBackward` advance forward and backward in the history, returning what page was accessed in each case.

To give you a better sense for what these functions do, let's trace through an example. Imagine the user visits the pages dikdik, quokka, and pudú, in that order. The browser would then make these three calls:

```
history.goToNewPage("dikdik");
history.goToNewPage("quokka");
history.goToNewPage("pudú");
```

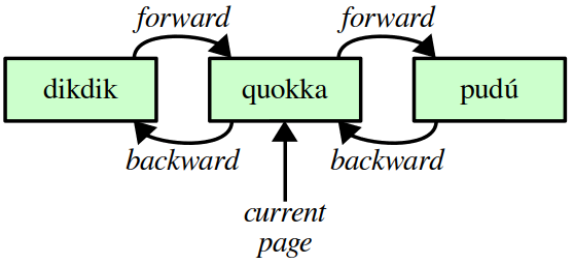
and the state of the history would look like this:



Notice that the last page, pudú, is marked as being the current page. Now, suppose the user presses the "back" button. The browser will then execute code along these lines:

```
string pageToLoad = history.goBackward(); // Returns "quokka"
doSomethingThatLoadsAPage(pageToLoad);
```

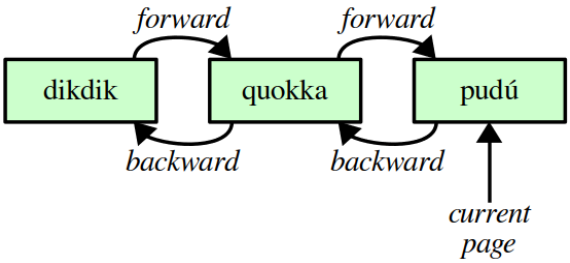
At this point, the browser is displaying the page about quokkas, and the history should look like this:



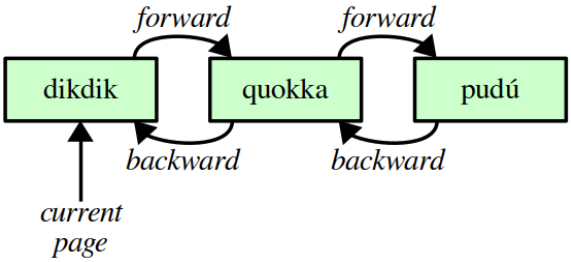
Notice that the pudú page is still there in the history, but isn't the active page. And that's a good thing, because if the user hits the forward button, the browser will execute code to the effect of

```
string pageToLoad = history.goForward(); // Returns "pudú"
doSomethingThatLoadsAPage(pageToLoad);
```

which would return to the pudú page and leave the history looking like this:



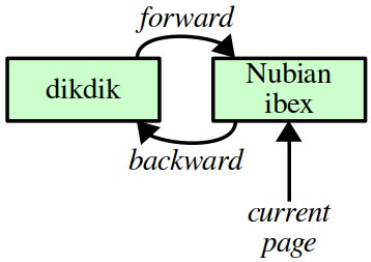
Things become a bit more interesting when the user navigates backward and then decides to visit a new page that isn't the one they returned from. For example, let's suppose the user hits the back button twice, triggering two calls to `history.goBackward()`, and leaving the history looking like this:



If the user now decides to visit a different page – say, Nubian ibex – then the browser will execute

```
history.goToNewPage("Nubian ibex");
```

so the history knows about the new page. Now, what should the history look like? Well, since the user came from the page "dikdik", hitting "back" should take them back there. But since the user is exploring a new path, the forward button won't be enabled. Internally, the history should look like this:



Gone are the days of quokkas and pudús. We can never return to that state of animal hipsterdom.

For a full description of what all the member functions in the `History` type do, along with the required time bounds, ***please refer to the `History.h` header file***. As a reminder, throughout this assignment, if you're curious what a function is supposed to do, check the appropriate header. The member functions are commented describing what inputs they should take in, what assumptions you can make about those inputs, what they should do, and what big-O runtimes they should have.

You are free to implement this class however you'd like, subject to the following restrictions:

- ***You must implement this type using a doubly-linked list***, along the lines of what's shown in the diagrams above.
- ***You must meet the time bounds set out in the header***. Specifically, every option except for `goToNewPage` should run in time O(1), and `goToNewPage` should run in time O(n), where n is the number of elements in the history.

You're free to implement that doubly-linked list however you'd like. Section Handout 6 discusses several ways of representing doubly-linked lists, and we invite you to think about which of those strategies you think would be best here.

You have practice writing classes and code that works with dynamically-allocated memory from Assignment 5, where you implemented the `HeapPQueue` type using dynamic arrays. The main difference here is that you're now dynamically allocating individual linked list cells rather than large arrays.

Something to keep an eye on here: make sure everything you allocate with `new` is balanced by exactly one call to `delete`. If you forgot to deallocate memory, you get what's called a ***memory leak***. Memory leaks generally don't manifest in a program unless you leave that program open and running for a long time, but they are serious concerns. Firefox 2 (wow, that was a while back!) famously had a memory leak that would cause it to crash if it were left running for too long, since the browser would slowly use up more and more memory until none was left.

To make it easier for you to see whether your code leaks memory, we've added some instrumentation to the `Cell` type that you can use to see how many `Cell`s have been created with new but not yet freed with delete. Check the "Destructor frees memory" test case in `History.cpp` for an example of how this works. (Pro tip: hover over the `instancesOf` function and the `TRACK_ALLOCATIONS_OF` macro and press `F2` (`fn` + `F2` on Macs) to jump to the documentation.) Feel free to add in other tests like this one to make sure you've got all your cases covered!

Some notes on this problem:

- Don't use the `new` keyword unless you're certain that you want to create a new linked list cell. You should only say `new` if you specifically want to make new linked list cells, not if you want to make a pointer to an existing linked list cell.
- Similarly, be intentional about freeing memory. You should only `delete` memory if you specifically want to permanently get rid of a specific linked list cell.
- In C++, if you declare a pointer variable and don't give it a value, it does *not* default to `nullptr`. Instead, it defaults to "I'm pointing at some random spot in memory, and chances are it's not `nullptr` and probably isn't even something you own." If you try dereferencing a pointer like this, it'll almost certainly crash your program. This is also true of pointer data members of `struct`s. Consequently, make sure you explicitly give each pointer a value.
- Want to learn more about doubly-linked lists? Check Section Handout 6 for details.

You must write ***at least three*** custom test cases for this part of the assignment. Be intentional about what you test. What sorts of things do you need to look out for? What edge cases would be tricky to handle?

Once you've finished this section, the history buttons (forward and backward) should work correctly in the browser. You won't see any text displaying until you've gotten the line manager working, but if you type text into the address bar and hint enter, you should be able to navigate from page to page.

Endearing C++ Quirks, Part 2: Returning Nested Types

There's another charming personality trait of C++ that pops up when implementing member functions that return nested types. For example, suppose that you define a `Cell` type inside your `History` class and then make a private helper function whose signature is

```
private:
    Cell* ebrateGoodTimes();
```

In the `.cpp` file, when you're implementing this function, you need to give the full name of the `Cell` type when specifying the return type:

```
History::Cell* History::ebrateGoodTimes() {
    // Come on!
}
```

Alas, the pun that worked so well in the header lost a *lot* in translation.

While you need to use the full name `History::Cell` in the *return type* of an implementation of a helper function, you don't need to do this anywhere else. For example, this code is perfectly legal:

```
History::Cell* History::ebrateGoodTimes() {
    Cell* cell = new Cell; // Totally fine!
    cell->next = new Cell; // Also fine!
    return cell;
}
```

Similarly, you don't need to do this if the function takes a `Cell` as a parameter. For example, imagine you have this member function:

```
private:
    void oohLooksItsA(Cell* ebrity);
```

then you could implement it without issue as

```
void History::oohLookItsA(Cell* ebrity) {
    // Specifically, it's Megan Smith!
}
```

Autocomplete

The MiniBrowser has an autocomplete system that's hooked up to its Article bar. As the user starts typing an article name, the autocomplete will return lists of suggested articles, populated by a combination of pages they've visited in the past, plus a list of the most commonly-visited articles on Wikipedia. This is done through the `Autocomplete` type, which has this interface:

```
class Autocomplete {
public:
    Autocomplete();
    ~Autocomplete();

    void add(const std::string& word);

    Vector<std::string> suggestionsFor(const std::string& prefix,
                                       int maxHits) const;

private:
    /* … up to you to decide! … */
};
```

The two major operations, as you can see, are `add` and `suggestionsFor`. As the name implies, the `add` function adds the given string to the collection of article titles that should appear in future autocomplete searches. For example, if you call

```
ac.add("Nubian ibex");
```

then in an autocomplete search for `"Nub"` or `"N"`, the article `"Nubian ibex"` will be a candidate.

The `suggestionsFor` function does the autocomplete search. It takes two parameters, a prefix and a maximum number of hits, then returns a `Vector<string>` of articles that begin with the specified prefix. If there are `maxHits` or fewer suggestions available, then `suggestionsFor` should return all of them. If there are more than `maxHits` options, `suggestionsFor` can return any `maxHits` of them.

You are free to implement this however you'd like provided, subject to the restriction that ***your implementation must be based on a trie***. You'll need to think through how to represent a trie in code and whether there are any tradeoffs available in the representation.

> ***Endearing C++ Quirks, Part 3: Signed and Unsigned Characters***
>
> Depending on how you implement your trie, you might want to use the fact that `char` values each have a unique integer value. (Didn't know that? No worries! It's something we didn't explicitly cover this quarter in lecture.) Be careful – on some systems `char`s range from 0 to 255, inclusive, and on others they range from -128 to +127, inclusive – and your section leader might be testing your code on a system that works differently than yours.
>
> On modern systems the `unsigned char` type (yes, that is the name of the type) always runs from 0 to 255, inclusive. Feel free to use this type if you want to use character values numerically. It's entirely possible to complete this assignment without running into this issue, in which case, feel free to ignore this section. This is more of a CS107 topic anyway. 😄

Some notes on this problem:

- As always, refer to the header file for more information about the `Autocomplete` type, including information about the time bounds and the return value of `suggestionsFor`.
- Your trie should be case-sensitive. Treat `"APPLE"` and `"apple"` as different strings.
- Be sure not to leak any memory! Use the `TRACK_ALLOCATIONS_OF` macro along with the handy `instancesOf` function and make sure everything balances.
- Member functions can call other member functions, but `const` member functions can only call other `const` member functions.

As is usual for our assignments, you'll need to write some custom test cases. We'd specifically like you write at least ***four*** different test cases for this part of the assignment. Again, take this job seriously. Really make sure that your trie handles edge cases properly, that it runs efficiently, and that it's not leaking memory.

Once you've finished this section, the autocomplete feature of the browser should be working correctly. As you type text into the address bar, you should see suggestions pop up. Additionally, if you visit new pages that weren't in the original list of suggestions, you should see those suggestions now pop up in the autocomplete window.

Endearing C++ Quirks, Part 4: Unicode and UTF-8

As you're writing and testing your code, you might find that some of the strings you're storing in your trie contain bizarre characters that aren't present in the original. This tends to happen when you're storing strings containing characters from non-English alphabets. To understand why this is, we need to take a quick detour to the wonderful world of Unicode and UTF-8.

On most systems, the C++ `char` type can take on one of 256 different values. However, there are *way* more than 256 possible characters that can appear in article titles – think of characters from different alphabets, mathematical symbols, and, yes, even emojis.

To represent this many characters, the strings inside MiniBrowser are represented using a format called ***UTF-8***. This representation works by having some glyphs represented using a sequence of multiple `char` values, while having other glyphs (like the question mark) represented using a single `char` value. For example, a `string` representing the string 🕌 (the mosque emoji) would be four `char`s long, and those `char`s would have the values 240, 159, 149, and 140, in that order. (Unless, of course, you're on a system in which the `char` type ranges from -128 to +127, in which case they'll have different values.)

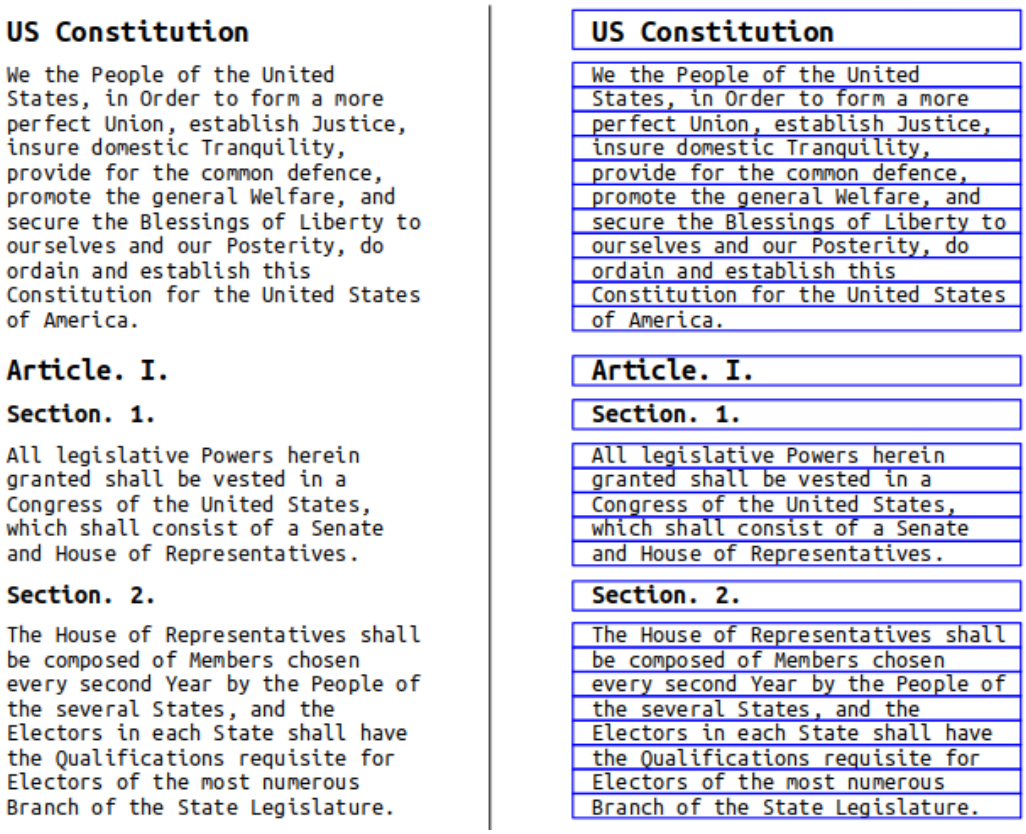
The good news is that, from your perspective, you shouldn't find yourself needing to do anything special to represent strings encoded in UTF-8. If you process `string` objects one `char` at a time, then the trie will still work properly, even if a single glyph takes up multiple `char` objects. However, where you *do* need to be careful is in making assumptions about the `char`s you see. UTF-8 uses almost all 256 possible `char` values, so you should not, for example, assume that you're always seeing English letters in your strings. Similarly, functions like `isalpha`, `toupper`, `tolower`, etc. might not work properly on the `char`s you're processing. But that's fine, since you're not supposed to do any case conversions anyway.

Line Manager

You may have noticed that no text is displaying inside of the browser window when you first start up the program. The reason for that is that you haven't implemented the ***line manager***, the component of the MiniBrowser that's responsible for displaying content.

As its name suggests, the line manager manages lines. To understand what that means, we need to take a quick break to take about what a *line* is, and then what about them needs to be managed.

Each page in the MiniBrowser is broken down into a number of lines of text, or just "lines" for short. For example, below is an example of a document (the first bit of the US Constitution). To the left is the way that it would render in the MiniBrowser. To the right is the same document, with the pieces of text grouped into lines.



There are a few important properties of lines that are relevant here. First, notice that ***all lines have the same width***. That width, specifically, is the full width of the document. Those lines might not use up all of the horizontal space allocated to them, but for simplicity we've included this unused space as part of the same line.

Second, ***not all lines have the same height***. For example, the line for the title ("US Constitution") is taller than the lines for the text of the constitution, since it has a larger font size.

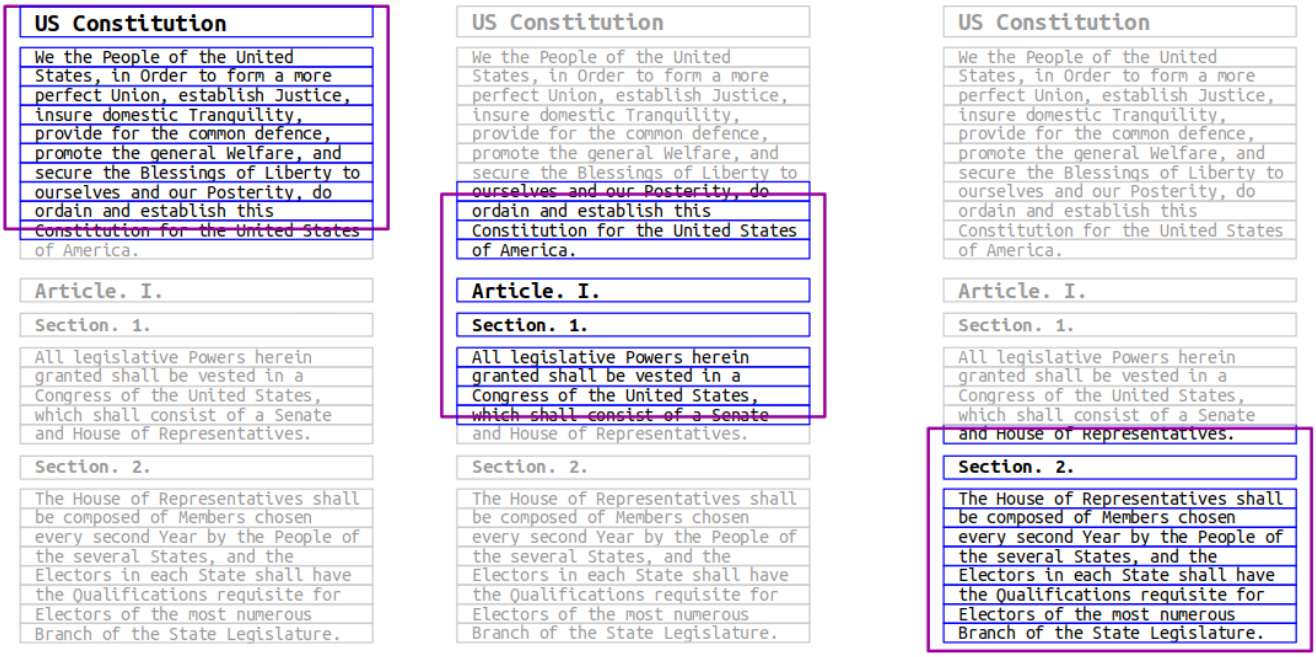
Third, ***there can be vertical spaces between the lines***. These spaces might correspond to whitespace between paragraphs, or extra vertical space between headings and the text body to make things easier to read.

Fourth, and finally, ***lines cannot overlap one another***. The purpose of breaking the document into lines is to make it easier to render text on the screen. Since we don't want lines of text drawing on top of one another, each line is allocated its own region of vertical space that doesn't overlap with any other line. Two lines might just barely touch one another, as might be the case inside a paragraph, but that's it.

The starter code for the MiniBrowser already contains the logic to take the text of an article and break it apart into lines. (When the status line says "Laying out page…," that's what it's spending its time doing.) However, once the text has been broken apart into lines, there's still some work to be done.

The first of these jobs is making the scrollbar work correctly. Most articles are too long for the browser to display all at once, and so the user will have to work the scrollbar to read the whole article. You've probably not stopped to think much about how scrolling works, but the techniques that go into fast scrolling are actually quite clever.

When you scroll up and down on the scrollbar, the effect you have is that the browser window stays fixed and that the content moves up and down behind it. But it actuality, it's more common for things to work the other way. We imagine that the content of the page stays fixed, and the browser moves a region called the ***viewport*** up and down over that content. (Think of it like the principle of relativity, but for software. You can't distinguish between a fixed viewport and scrolling content and a scrolling viewport with fixed content). For example, as our user scrolls up and down over the Constitution, she's essentially moving the browser window up and down over the lines that make up that document, as shown below.



In this picture, the purple window represents the viewport of the web browser. As you can see, depending on the position of the browser viewport, some lines (blue borders, black text) are visible inside the window and need to be drawn so the user can see them. Some lines (gray borders, gray text) aren't inside the viewport, and therefore don't need to be drawn. After all, the user can't see them.

Notice that a line of text needs to get drawn if any part of that line overlaps with the viewport. For example, in the leftmost example, the line `"Constitution for the United States"` is partially cut off, but still needs to be drawn so that the user can see the tops of the letters. In the center example, the line `"ourselves and our Posterity, do"` is partially visible at the top of the window, so it needs to be drawn as well. However, any line that doesn't overlap with the viewport doesn't need to be shown.

It takes time to render text on the screen – there's a bunch of logic to decode the bytes that make up the text into individual glyphs, then to load the images for those glyphs, then to figure out the spacing between glyphs, then to apply styles, etc. – so a major optimization in getting the browser to appear snappy and responsive is to only display the lines that overlap that viewport. This is one of the major tasks of the line manager: figuring out a way to store the lines so that queries of the form "what are all the lines that need to get displayed given where the viewport is?" can be implemented as efficiently as possible.

If you're thinking "hey, isn't that kinda sortaish like the lower bound searches we did in Data Sagas?," the answer is "yes!" The details are a bit different, and we're going to ask you to implement things in a slightly different way than last time, but the idea is conceptually quite similar.

There's another major task delegated to the line manager, and that's determining where the mouse is. As the user moves her mouse around the viewport, the browser needs to highlight links that are being hovered over. If she clicks on a link, the browser needs to change which page it's on to follow that link.

How should the browser go about doing this? One option, which would not be very efficient, would be to scan over all the lines, checking if the mouse is contained within any of them. That's not going to be very fast, especially if you have a long article with thousands of lines in it. If you store the lines in an intelligent fashion, though, you can dramatically speed up this sort of search. If you think about it, we have a bunch of items (here, lines) that are sorted by *y* coordinate, and we're trying to see which of those items is at a particular *y* coordinate. That sounds like a lot like what you'd do with a binary search – or, more relevant here, with a binary search *tree*.

What You Need To Do

Your task is to implement the `LineManager` type, which is defined below:

```
class LineManager {
public:
    LineManager(const Vector<Line *>& lines);
    ~LineManager();

    double contentHeight() const;
    Line*  lineAt(double y) const;
    Vector<Line *> linesInRange(double topY, double bottomY) const;

private:
    /* … more on that later … */
};
```

Here's how this type works. Once the browser has downloaded a page and broken that page apart into lines, it'll take those lines, stash them in a `Vector<Line *>` (sorted by *y* coordinate) and hand that `Vector` off to a new `LineManager` using the constructor. Whenever the browser needs information about the lines on a page, it can make one of three types of queries:

- ***Content height***: The very first line always starts at *y* coordinate 0. What is the *y* coordinate of the bottom of the bottommost line? This information is used to determine how the scrollbar should react when the user scrolls it. (Notice that this is not necessarily the sums of the heights of all the lines, because there might be spaces between the lines. Do you see why?)
- ***Line at***: When the user moves the mouse, we need to know which link to highlight, and when the user clicks the mouse, we need to know which link to traverse. Rather than searching all lines and seeing if we can find one that matches, the browser will ask the `LineManager` to tell it which line, if any, is at the specified *y* coordinate.
- ***Lines in range***: As mentioned above, to render the content inside the browser, the browser needs to know what lines are in the range of the viewport. Given a range of *y* coordinates, which lines should we display?

You may have noticed that these operations reference the `Line` type. Here, `Line` is a type defined in `Browser/Line.h` with a bunch of fields, of which only a few are of interest to you:

```
class Line {
public:
    double topY() const;
    double bottomY() const;

    /* … other things you don't need … */
};
```

Our graphics coordinate system has $y = 0$ at the top of the page, increasing downward. Here, the `Line`'s `topY()` and `bottomY()` return the *y* coordinates that define the line. Because *y* increases as you move down, `topY()` will always be less than or equal to the value of `bottomY()`, and you can rely on this.

The `topY()` and `bottomY()` values are inclusive, so a line with top *y* coordinate 137 and bottom *y* coordinate 154 would include *y* coordinate 151 (which is within the range), 137 and 154 (the endpoints), but not 136 or 154.001 (which are outside the range).

You may have noticed that this assignment is themed around linked structures, and that continues here. You're free to make most of the design decisions for this type on your own, but ***you must implement the `LineManager` as a binary search tree***. Here, the keys in the nodes will be `Line*`s, with the children corresponding to "lines that are above this line" and "lines that are below this line."

Some notes on this problem:

- If you have any questions about what the member functions are supposed to do, what assumptions they should make about their inputs, what specifically the output should be, or what the required time complexities are, check the `LineManager.h` header file.
- The graphics coordinate system is set up so that the top of the article is at *y* coordinate 0, with *y* coordinates increasing from top to bottom. The unit of measurement is the pixel, so *y* coordinate 137 means "137 pixels below the top of the article." The `contentHeight()` function therefore returns the height of all the content, measured in pixels, hence the name.
- This part of the assignment requires you to interface with other parts of the browser in ways that the other parts of the assignment did not. Additionally, you'll need to make sure you understand the conceptual model of lines and viewports. If you're shaky on how these things fit together, stop by the CLaIR to ask for a clarification. Writing code when you aren't sure what it is that you need to do is an easy way to introduce bugs.
- The constructor for the `LineManager` can assume that the `Line*`s that appear in the `Vector` are provided to you in sorted order. It's the job of the `LineManager` to actually build the BST holding all of the `Line*`s. You should aim to construct as balanced of a tree as possible here, since, as you've seen, operations on a binary search tree get really, really slow when the tree is imbalanced. As a hint for how to do this, since you already have the lines provided in sorted order, consider building a BST for them by placing the middle element in the array up at the root, then recursively processing the left and right subarrays.
- Notice that there are no functions on `LineManager` that allow the client to add or remove lines once the manager has been set up. And that makes sense – the contents of an article don't change once you've loaded the page.
- ***Draw lots of pictures***. You actually don't need all that much code to get this part of the assignment up and running. However, to determine *what* code to write, you'll need to have a good grasp of what that code needs to do. For example, when you're determining which `Line` is at a given *y* coordinate, draw a picture of a bunch of lines and overlay a BST on top of them. Think about how you'd decide which direction of the tree to search.

Testing is key for this part of the assignment. We'd like you to implement at least ***six*** custom test cases, each of which should look for some different edge case of the functions you need to implement. Why six? Because it shouldn't be too tough to find that many different edge cases that you need to test. So test thoroughly! Also, feel free to use the `TRACK_ALLOCATIONS_OF` macro and the `instancesOf` function to diagnose memory errors.

Once you've finished this part of the assignment, you should see content in each page, and the scrollbars should work properly. You should also be able to click links to change which page you're on. Nifty!

> ***Endearing C++ Quirks, Part 5: Pointer Comparisons***
>
> In C++, it's legal to compare pointers using the `<`, `<=`, `>=`, and `>` operators. These operators are useful if you have multiple pointers into the same array, and they tell you how those pointers are ordered within that array. This is something you'll see if you take CS107.
>
> Binary search trees are based on the idea that you compare elements against one another to determine which subtrees to explore. Throughout this part of the assignment, you will need to see how different *y* coordinates relate to different `Line` objects. Make sure that you are not comparing `Line*` pointers against one another using these relational operators – the results of those comparisons are not guaranteed to have any relation whatsoever to where those lines are in space.

(Optional) Part Four: Extensions!

Once you've gotten everything up and running, we encourage you to tinker around with the MiniBrowser and find cool extensions that you can add in. Here are some suggests to get you started:

- ***History***: Could you store the browser history on-disk so that when the browser starts up, it contains the history from last time? Or could you perhaps edit the starter files so that the browser begins at the most-frequently-visited-page that the user has been to?
- ***Autocomplete***: We haven't specified the order in which you should return items when doing autocomplete searches, nor have we told you how to break ties when there are more possible items to return than there are spots available. Could you somehow determine which pages are more important than others and, if so, uprank them? Could you prioritize pages that have been visited before?
- ***Line Manager***: The sorts of searches that are performed in the line manager have a property called *locality*. For example, if the user is scrolling the page down, there's a very good chance that most, if not all, of the lines that are returned are going to be the same as the ones returned earlier. Similarly, in queries for which line is under the mouse, chances are there are only a few possible lines that might be returned – namely, the ones that are currently displayed. The ***splay tree*** is a type of BST that reshapes itself whenever a query is made to hoist up whatever was found up to the top of the tree. By being strategic with how those restructurings are done, the splay tree ends up having much, much better performance than a balanced tree when access patterns aren't uniform. Try implementing a splay tree instead of a regular BST for this part of the assignment.

Submission Instructions

To submit your work, please turn in the files

- `History.h` and `History.cpp`,
- `Autocomplete.h` and `Autocomplete.cpp`, and
- `LineManager.h` and `LineManager.cpp`.

If you've edited any of the other starter files, please feel free to submit those as well. Make sure that you've written the appropriate numbers of tests for each part of this assignment. And that's it! You're done!

Good luck, and have fun!