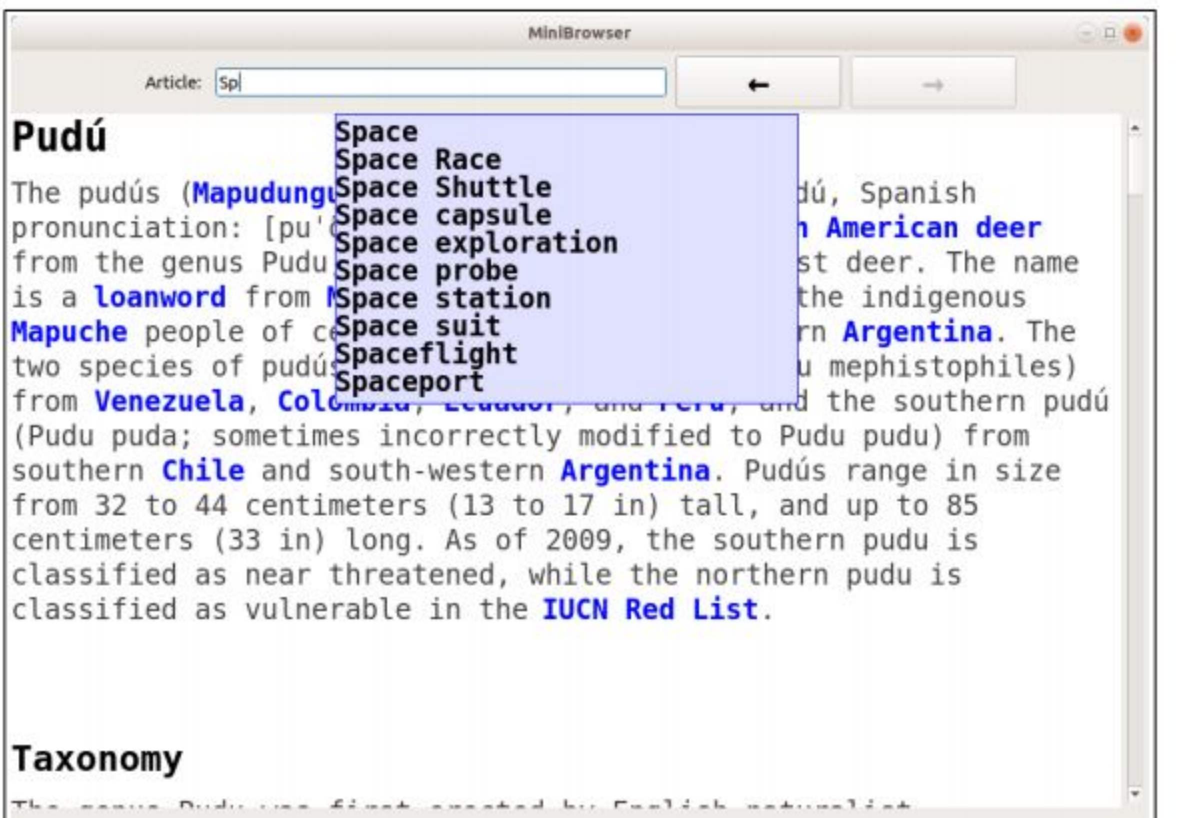
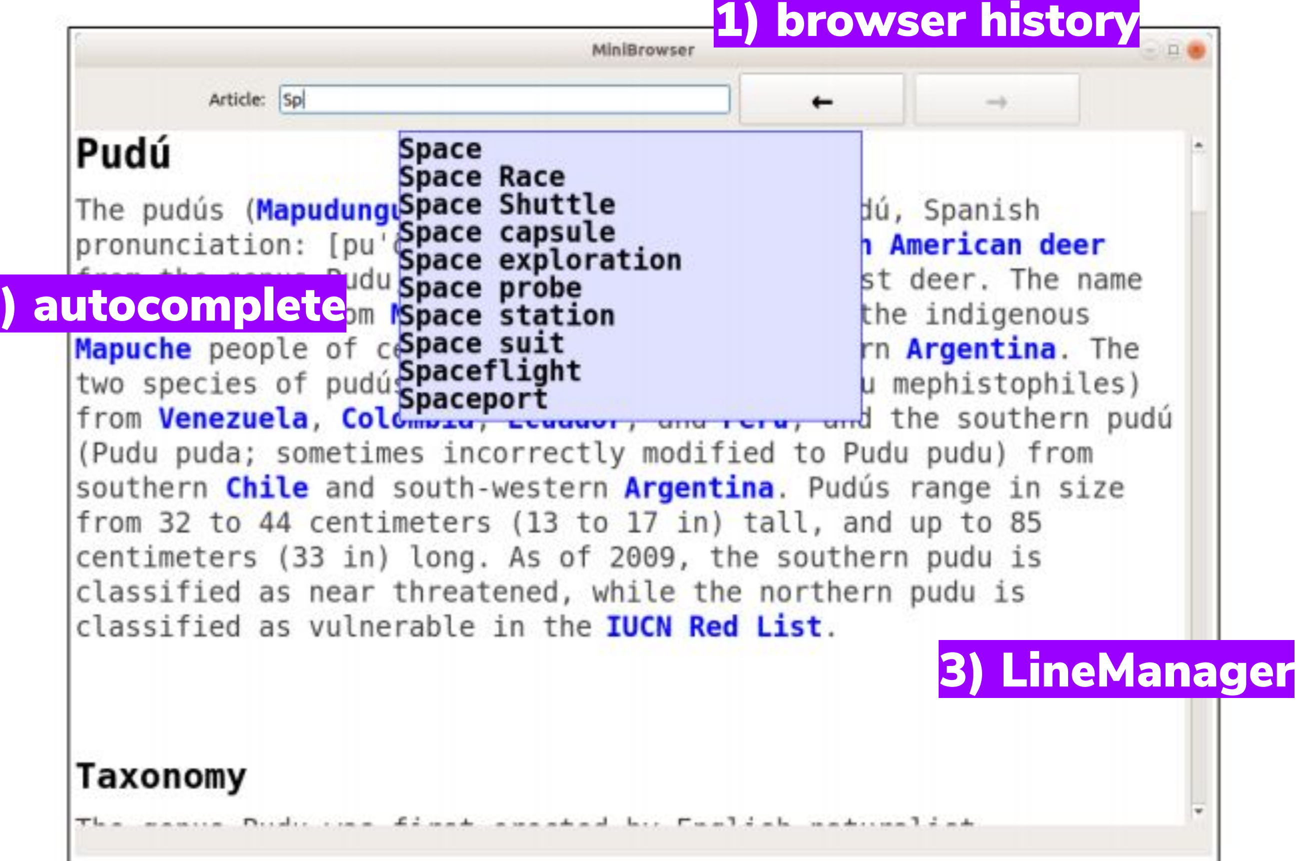
Assignment 6: "Minibrowser"

Juliette Woodrow, Ethan Chi

What is MiniBrowser?

- You build your own “mini” browser that loads Wikipedia articles!
- We handle:
  - the downloading articles from Wikipedia part
- You handle:
  - the browser history
  - the autocomplete search function
  - rendering lines on the screen!
- Uses **linked lists** and **pointers** heavily


MiniBrowser
Article:
Sp
←
→
Pudú
Space
Space Race
Space Shuttle
Space capsule
Space exploration
Space probe
Space station
Space suit
Spaceflight
Spaceport
The pudús (Mapudungu
dú, Spanish
pronunciation: [pu'
American deer
from the genus Pudu
st deer. The name
is a loanword from
the indigenous
Mapuche people of c
rn Argentina. The
two species of pudús
u mephistophiles)
from Venezuela, Col
d the southern pudú
(Pudu puda; sometimes incorrectly modified to Pudu pudu) from
southern Chile and south-western Argentina. Pudús range in size
from 32 to 44 centimeters (13 to 17 in) tall, and up to 85
centimeters (33 in) long. As of 2009, the southern pudu is
classified as near threatened, while the northern pudu is
classified as vulnerable in the IUCN Red List.
Taxonomy



1) browser history
2) autocomplete
3) LineManager
MiniBrowser
Article:
Sp
Pudú
Space
Space Race
Space Shuttle
Space capsule
Space exploration
Space probe
Space station
Space suit
Spaceflight
Spaceport
(Pudu puda; sometimes incorrectly modified to Pudu pudu) from
southern Chile and south-western Argentina. Pudús range in size
from 32 to 44 centimeters (13 to 17 in) tall, and up to 85
centimeters (33 in) long. As of 2009, the southern pudu is
classified as near threatened, while the northern pudu is
classified as vulnerable in the IUCN Red List.
Taxonomy

Part 1: Browser History

Overview

Overview

Goal: Implement a history object to keep track of web browser history

Overview

Goal: Implement a history object to keep track of web browser history

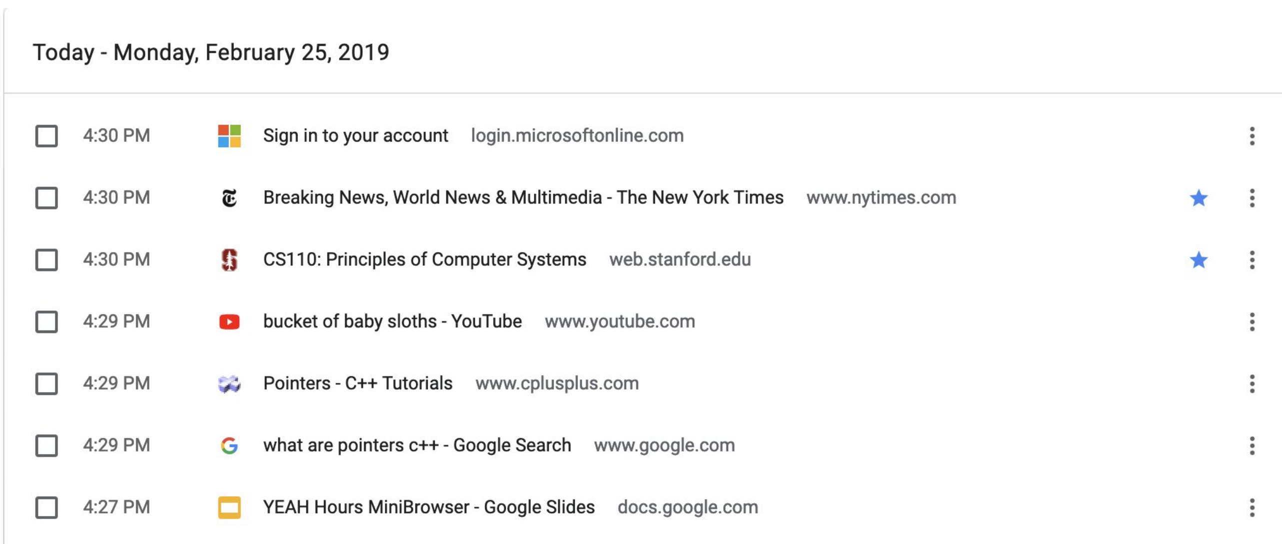
Kind of like what Google Chrome has for each user!

Overview

Goal: Implement a history object to keep track of web browser history

Kind of like what Google Chrome has for each user!

Check out my history from Monday!



Overview

Goal: Implement a **history object** to keep track of web browser history

Overview

Goal: Implement a **history object** to keep track of web browser history

```
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! ... */
};
```

Overview

Goal: Implement a **history object** to keep track of web browser history

```
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! ... */
};
```

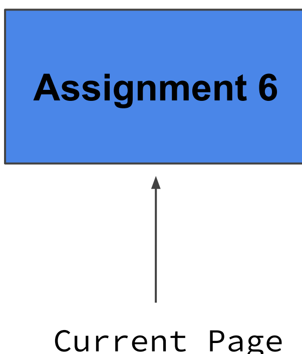
- Constructor
- ~Destructor
- 5 public methods

Let’s Look at an Example…

Imagine a 106b student is using our browser

Imagine a 106b student is using our browser

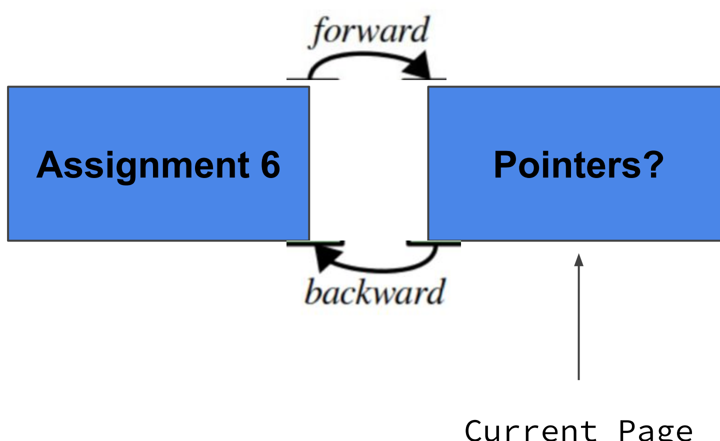
history.goToNewPage("cs106b assignment 6");



Imagine a 106b student is using our browser

history.goToNewPage(“cs106b assignment 6”);

history.goToNewPage(“pointer”);

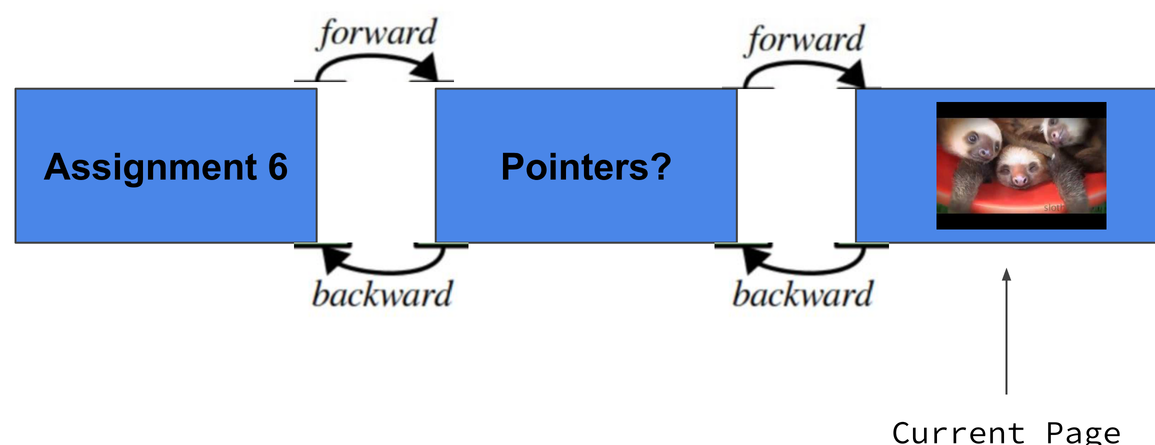


Imagine a 106b student is using our browser

history.goToNewPage("cs106b assignment 6");

history.goToNewPage("pointer");

history.goToNewPage("bucket of baby sloths");



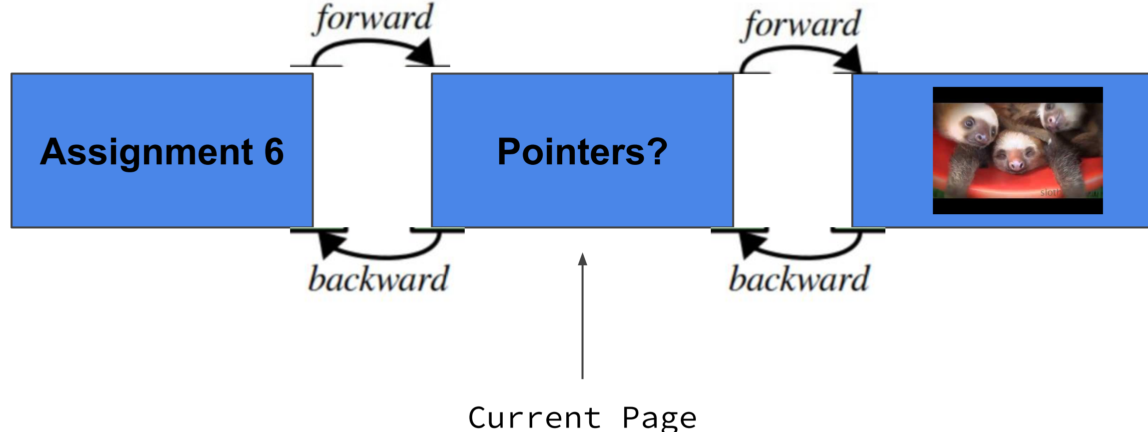
Imagine a 106b student is using our browser

history.goToNewPage("cs106b assignment 6");

history.goToNewPage("pointer");

history.goToNewPage("bucket of baby sloths");

history.goBack();



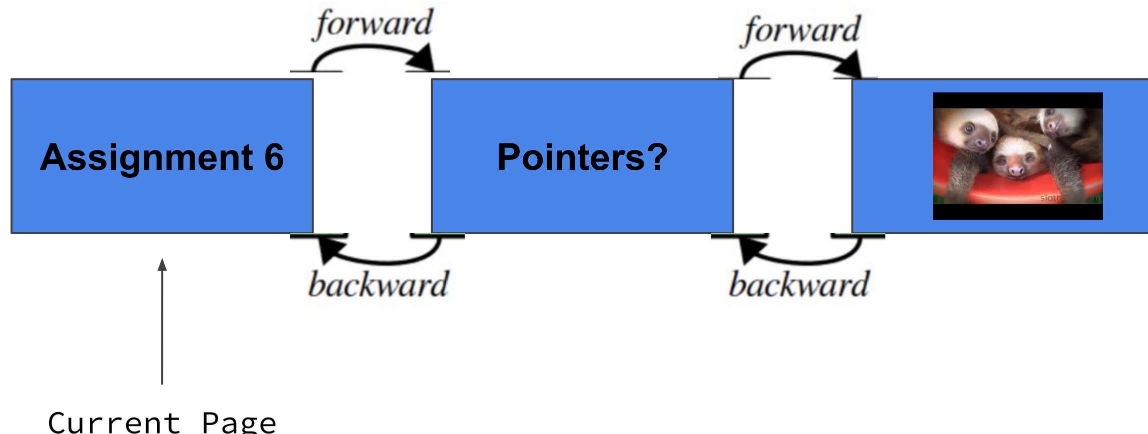
Imagine a 106b student is using our browser

history.goToNewPage(“cs106b assignment 6”);

history.goToNewPage(“pointer”);

history.goToNewPage(“bucket of baby sloths”);

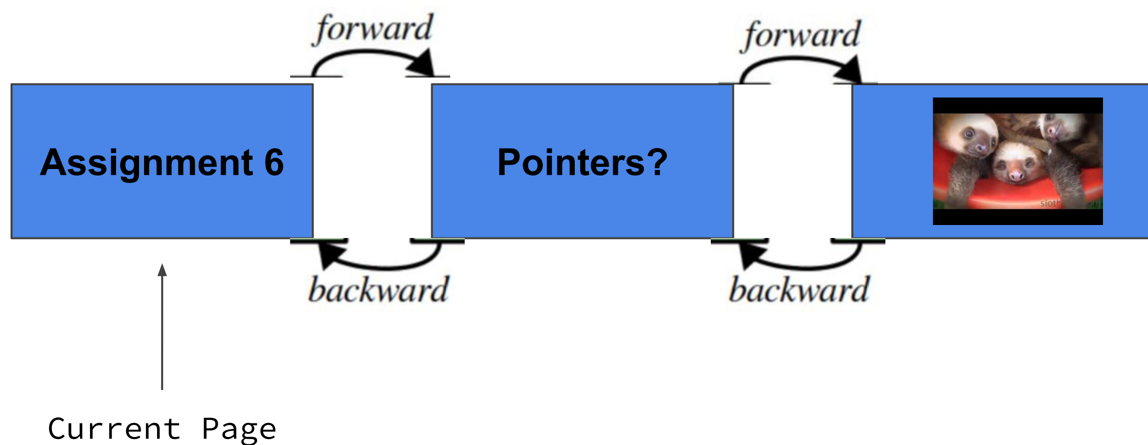
history.goBack(); x2



Imagine a 106b student is using our browser

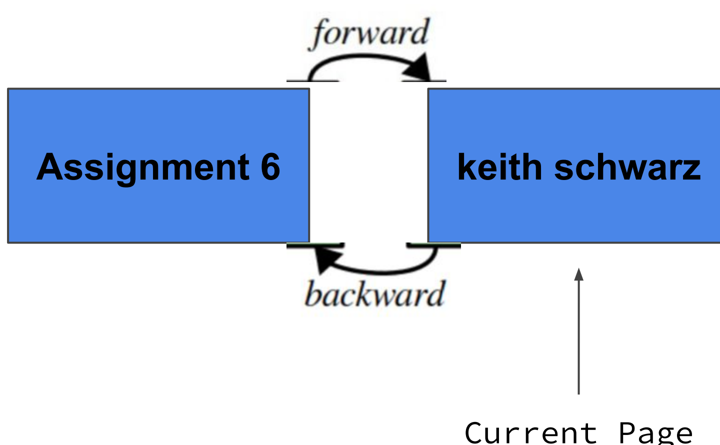
What will happen if they type:

history.goToNewPage(“keith schwarz”); ???



Imagine a 106b student is using our browser

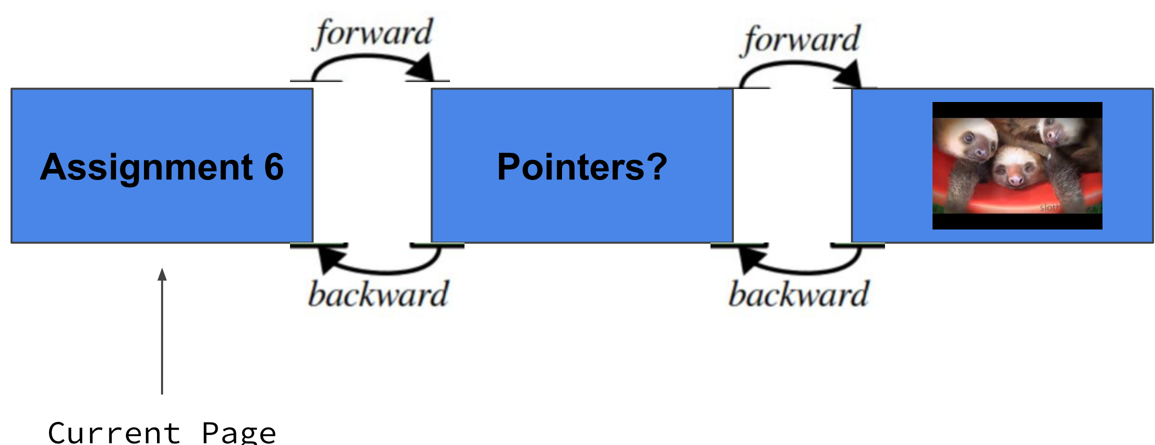
history.goToNewPage("keith schwarz");



Let’s look at the Public Methods

goToNewPage(const std::string& page)

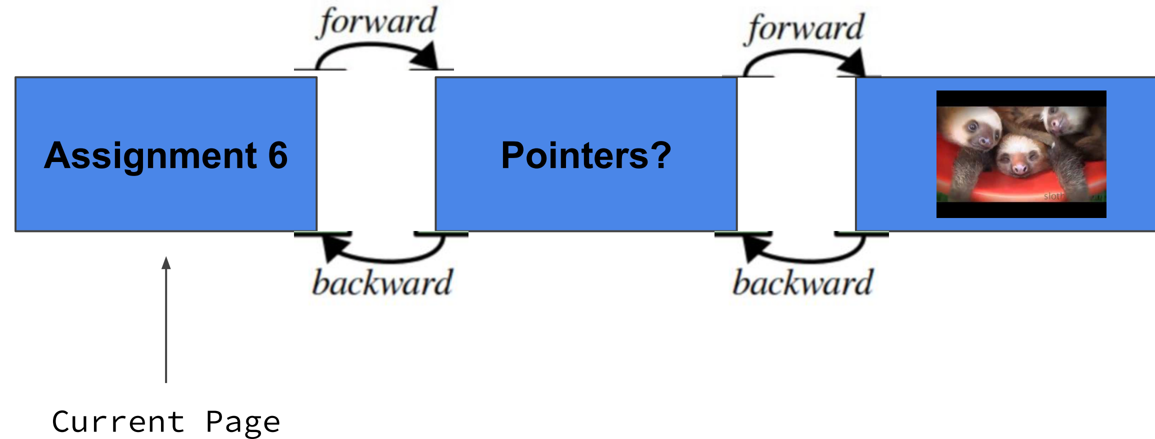
goToNewPage(const std::string& page)



goToNewPage(const std::string& page)

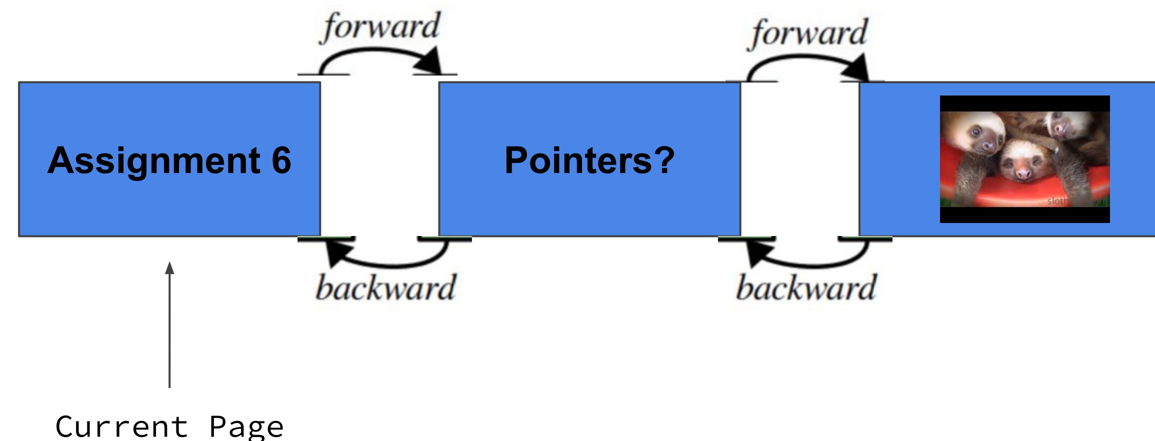
We are going to walk through what happens when the user attempts:

history.goToNewPage("keith schwarz");



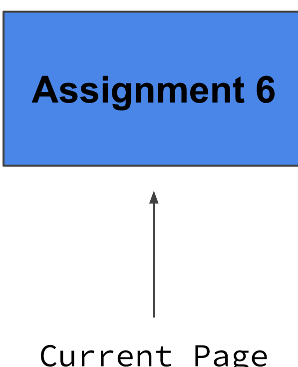
goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.



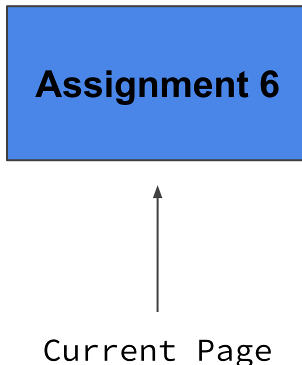
goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.



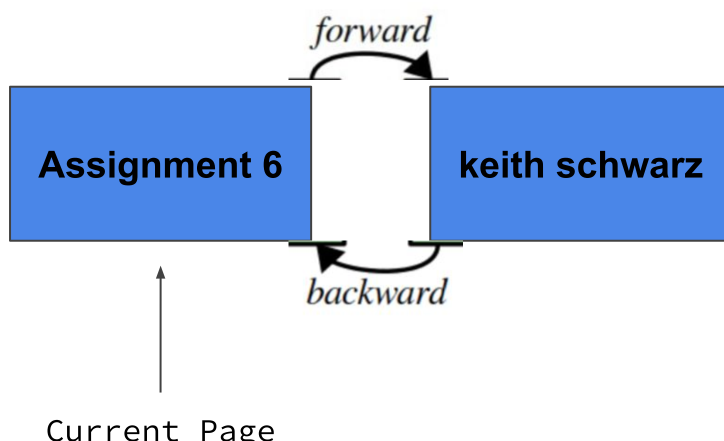
goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.
- Append the new page to the end of the history.



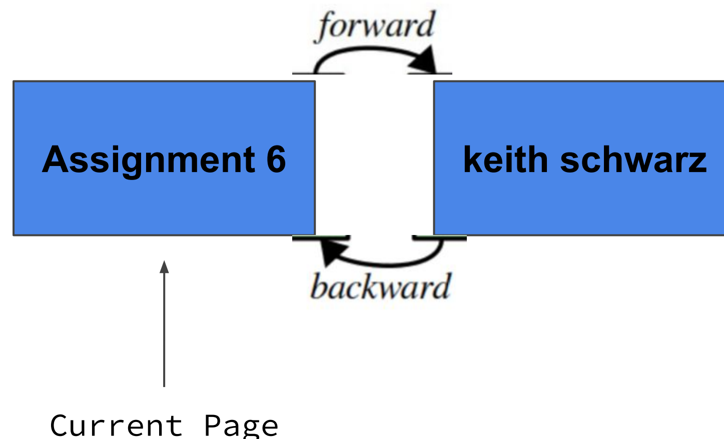
goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.
- Append the new page to the end of the history.



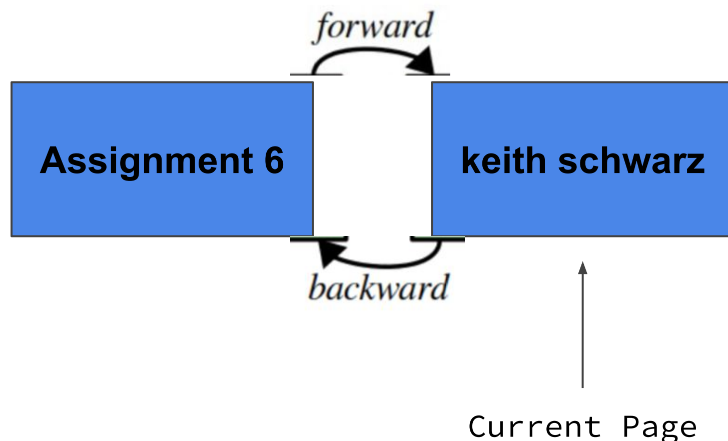
goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.
- Append the new page to the end of the history.
- Update the current page to be the new page



goToNewPage(const std::string& page)

- If there are any pages forward of the current page in history clear them out.
- Append the new page to the end of the history.
- Update the current page to be the new page



hasForward()

hasForward()

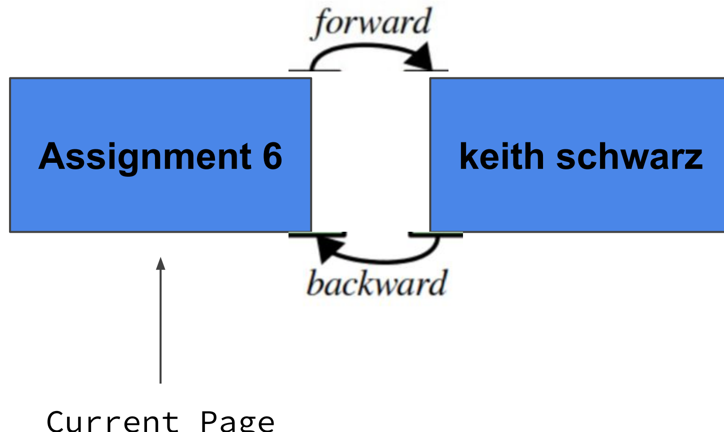
- If there is a page after the current one, return true

hasForward()

- If there is a page after the current one, return true
- If this is the last in the list, return false

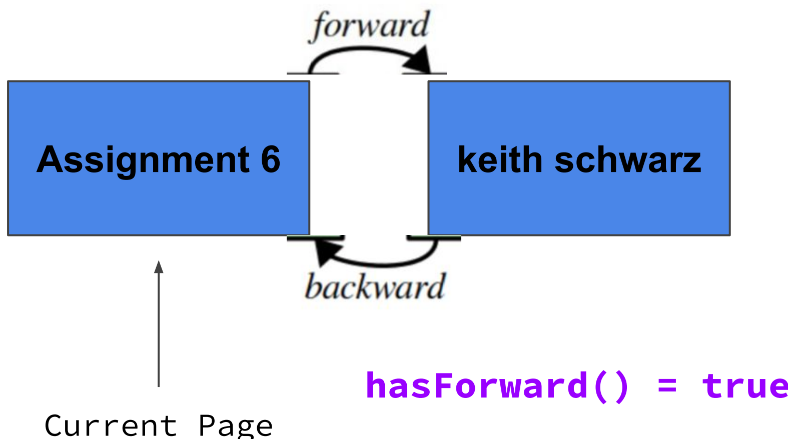
hasForward()

- If there is a page after the current one, return true
- If this is the last in the list, return false



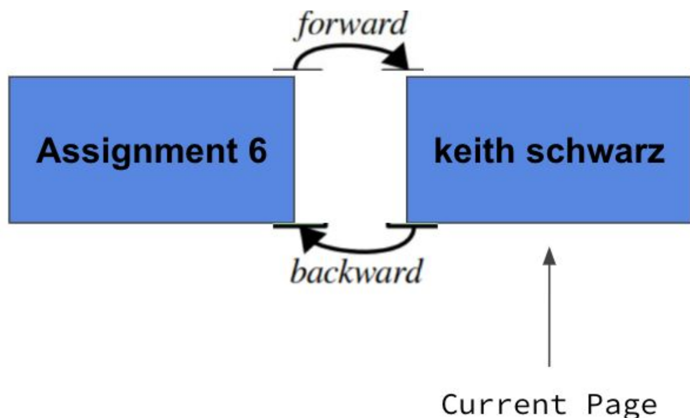
hasForward()

- If there is a page after the current one, return true
- If this is the last in the list, return false



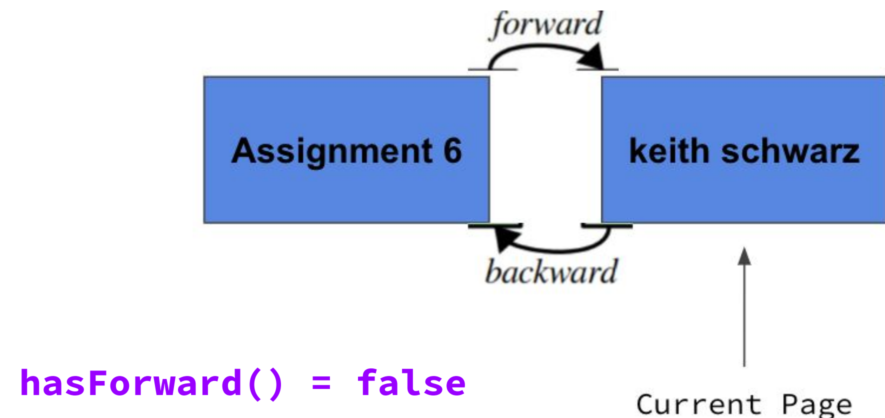
hasForward()

- If there is a page after the current one, return true
- If this is the last in the list, return false



hasForward()

- If there is a page after the current one, return true
- If this is the last in the list, return false



hasBackward()

hasBackward()

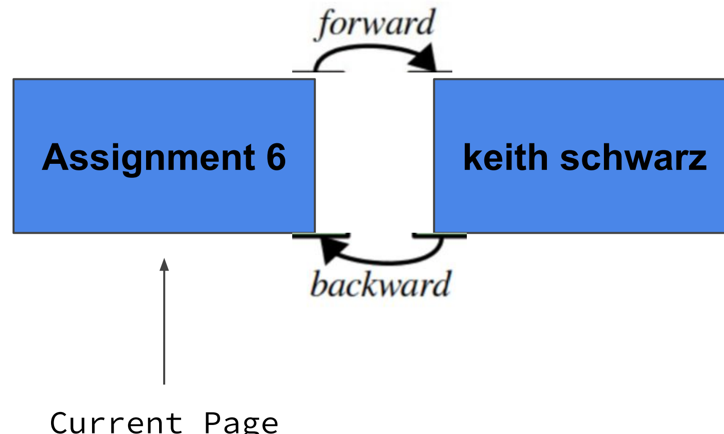
- If there is a page before the current one, return true

hasBackward()

- If there is a page before the current one, return true
- If this is the first in the list, return false

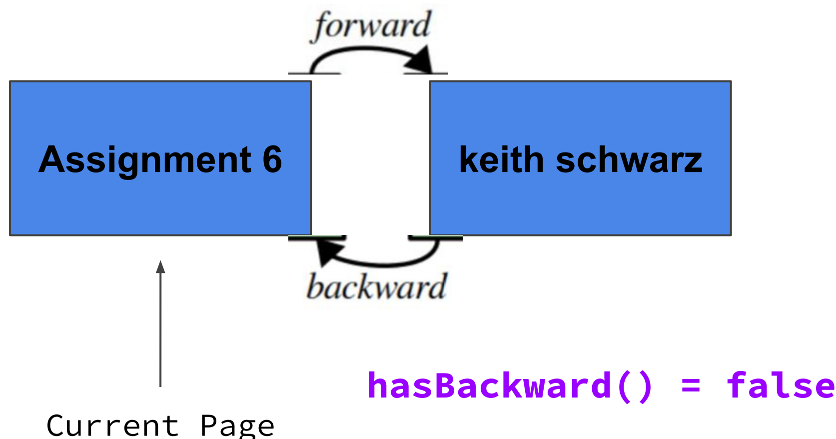
hasBackward()

- If there is a page before the current one, return true
- If this is the first in the list, return false



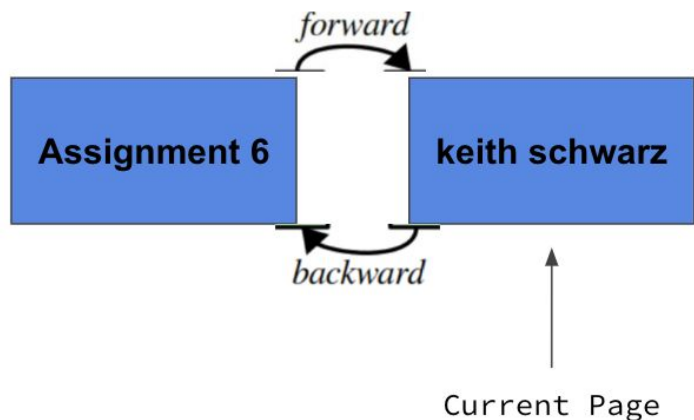
hasBackward()

- If there is a page before the current one, return true
- If this is the first in the list, return false



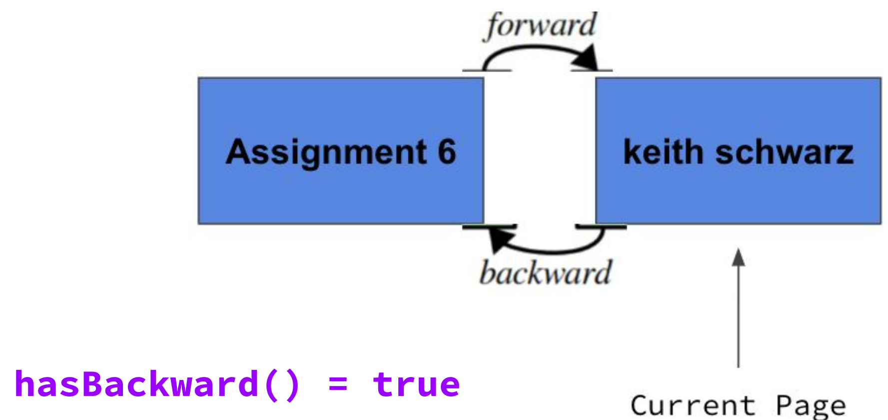
hasBackward()

- If there is a page before the current one, return true
- If this is the first in the list, return false



hasBackward()

- If there is a page before the current one, return true
- If this is the first in the list, return false



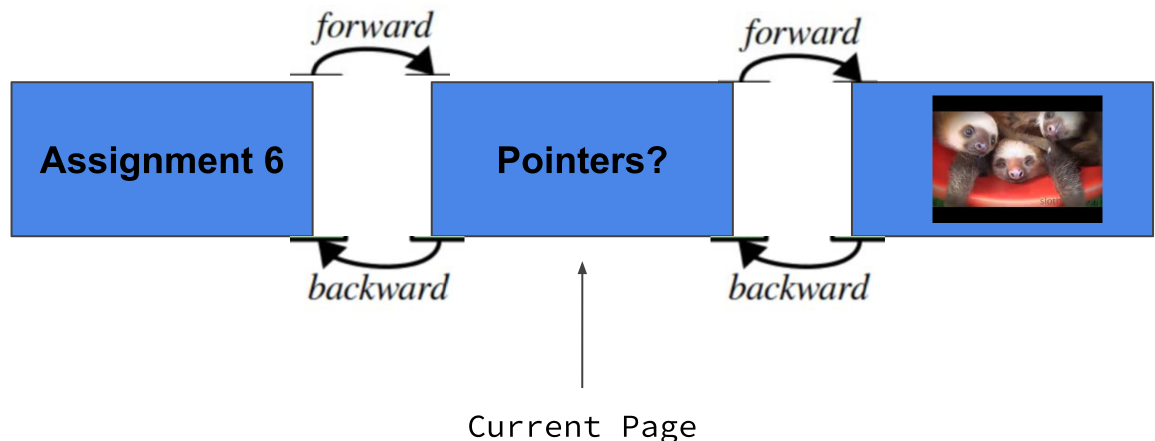
goForward()

goForward()

- Moves current page forward one step

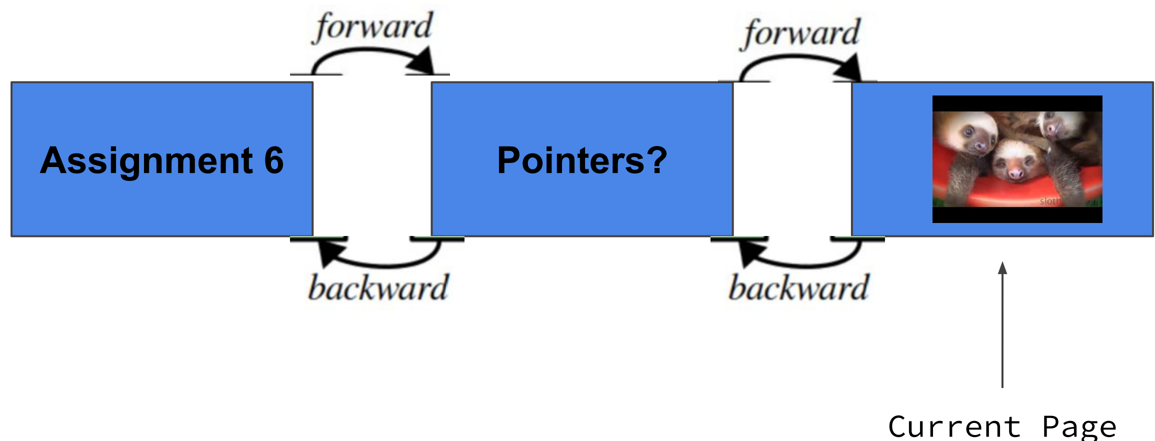
goForward()

- Moves current page forward one step



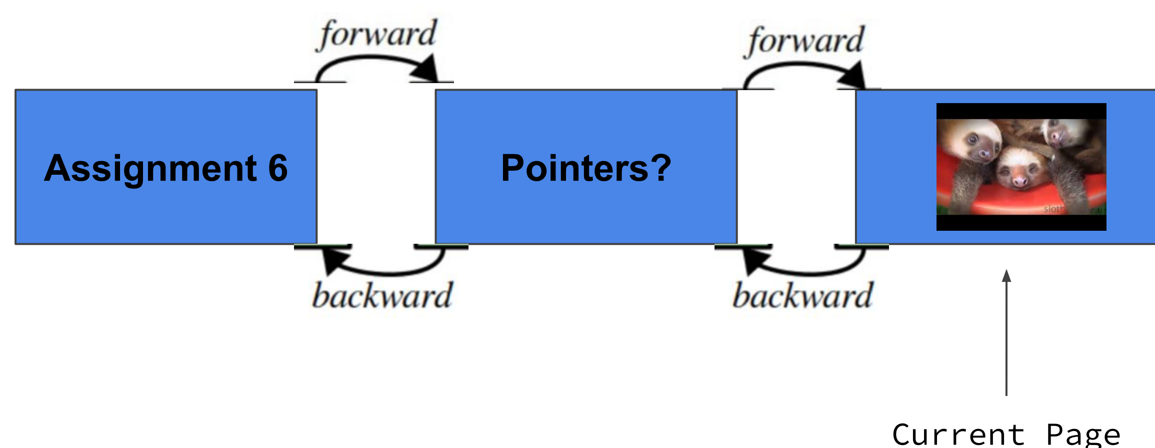
goForward()

- Moves current page forward one step



goForward()

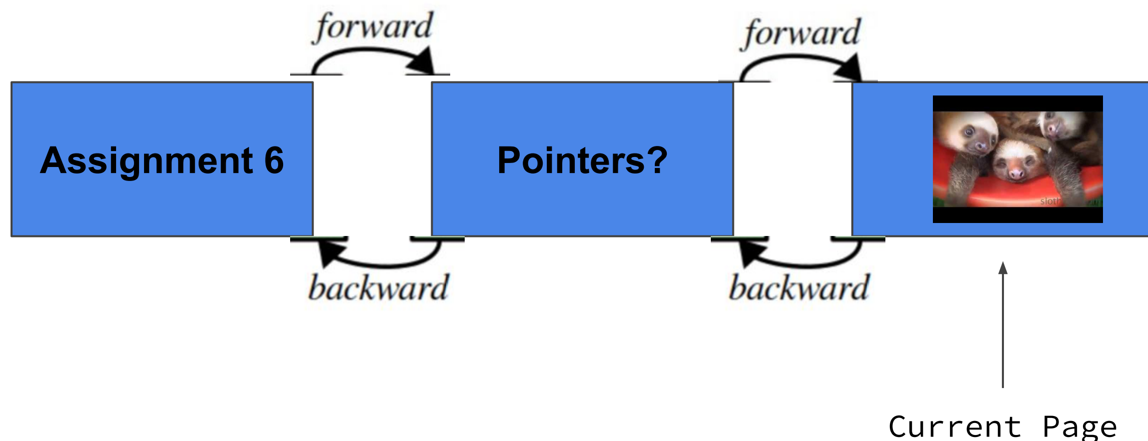
- Moves current page forward one step
- Returns what this page is



goForward()

- Moves current page forward one step
- Returns what this page is

Return: “bucket of baby sloths”

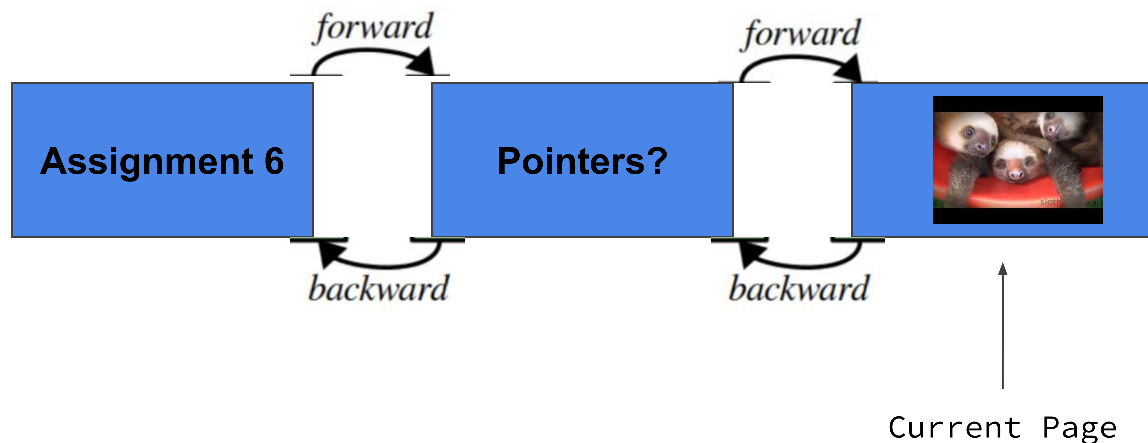


goForward()

- Moves current page forward one step
- Returns what this page is

Return: “bucket of baby sloths”

****If it is not possible to go forward, report an error****



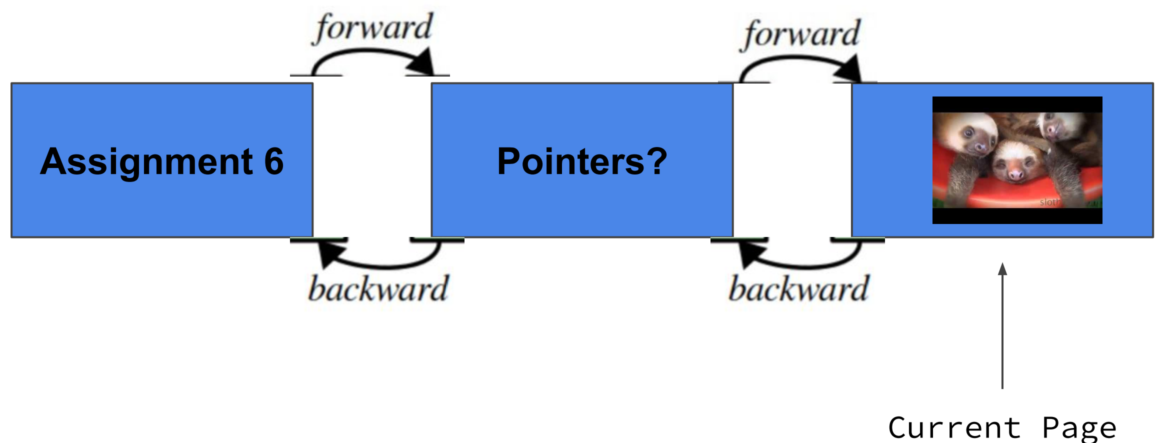
goBackward()

goBackward()

- Moves current page backward one step

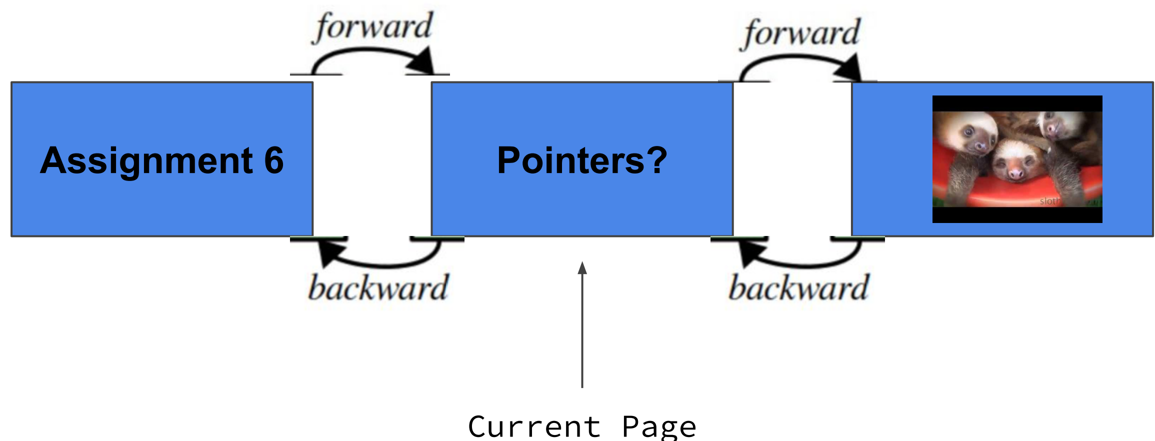
goBackward()

- Moves current page backward one step



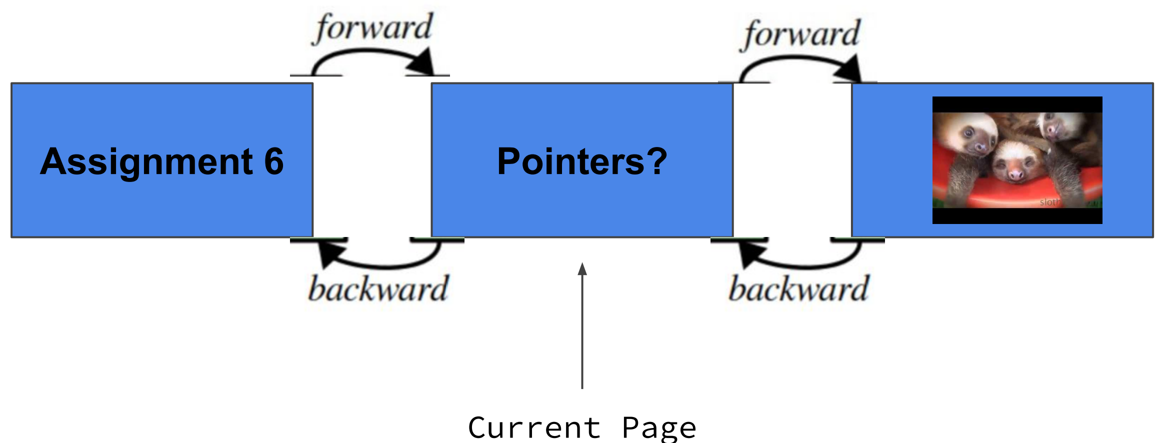
goBackward()

- Moves current page backward one step



goBackward()

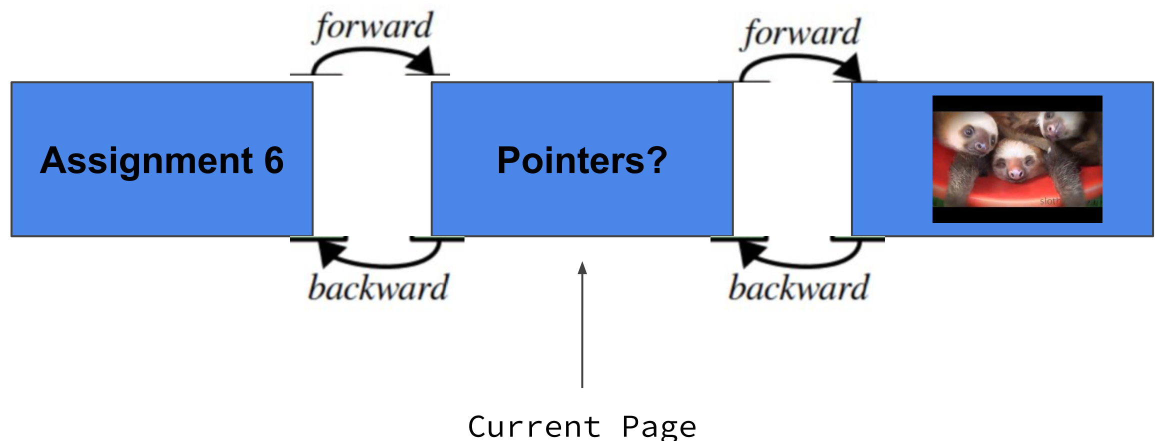
- Moves current page backward one step
- Returns what this page is



goBackward()

- Moves current page backward one step
- Returns what this page is

Return: “pointer”

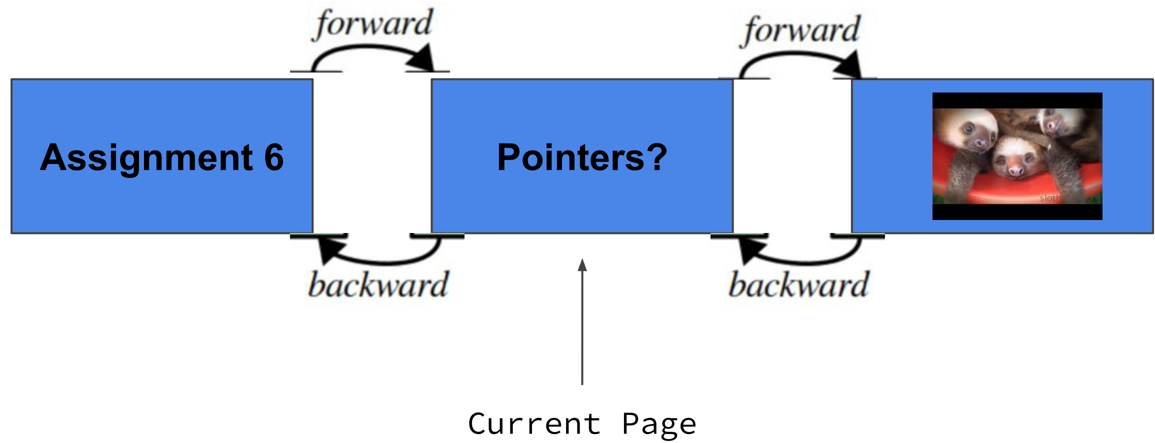


goBackward()

- Moves current page backward one step
- Returns what this page is

Return: “pointer”

****If it is not possible to go backward, report an error****



Restrictions on the History Class Implementation

Restrictions

- You MUST implement this using a doubly-linked list, along the lines of the example shown in the assignment handout

Restrictions

- You MUST implement this using a doubly-linked list, along the lines of the example shown in the assignment handout

- You MUST meet the time bounds set out in the header
    - Every option EXCEPT for goToNewPage should run in O(1) time
    - goToNewPage should run in O(n)
        - n is the number of elements in the history

Resources to help Understand Doubly Linked Lists

Resources for Doubly Linked Lists

- See section handout six (and solutions)

Resources for Doubly Linked Lists

- See section handout six (and solutions)

- Before starting, it may be worth it to go through this section problem:

Problem Seven: Doubly-Linked Lists

The linked lists we talked about in lecture are called ***singly-linked lists*** because each cell just stores a single link pointer, namely, one to the next element in the list. A common variant on linked lists is the ***doubly-linked list***, where each cell stores two pointers – a pointer to the next element in the list (as before) and a pointer to the previous element in the list.

Let's begin by modifying some of the existing code from lecture to account for this case. Assuming you have a `Cell` type representing a cell in a doubly-linked list, write a function

```
Cell* readList();
```

that reads a list of values from the user, then returns a new doubly-linked list containing those values in the order they were entered.

Doubly-linked lists have one really nice property: it is *really* easy to splice a new element into or out of a doubly-linked list. Write a function

```
void insertBefore(Cell*& head, Cell* beforeMe, Cell* newCell);
```

that takes as input a pointer to the first element in a doubly-linked list, a pointer to a cell somewhere in the linked list (`beforeMe`), and a newly-allocated `Cell` object, then splices the new cell into the doubly-linked list right before the cell `beforeMe`. Your function should update head so that when the function returns, it still points at the first cell in the linked list. (Why is it necessary to pass in the head of the list?) You can assume that `beforeMe` is not null.

Tips, Cautions, and Notes

Tips/Cautions

- Make sure every new has exactly one delete
- Need to write std::string in .h files
- Member functions that return nested types also need the :: but use class name to tell compiler where these types come from
- ONLY use new when you are positive that you want to make a new linked list cell. DON'T use new when you just want a <u>pointer</u> to an <u>existing</u> linked list cell
- Similarly, only delete if you are positive that you want to <u>permanently</u> delete this cell

Notes

- You MUST write 3 custom test cases

- When finished with Browser History:
  - History buttons (forward and backward) should work.
  - You won't see text until you implement line manager
  - But if you type text into the address bar and hit enter you should be able to navigate between pages

Part 2: Autocomplete

Pr

Pr
Proxy server
President of the United States
Prince Philip, Duke of Edinburgh
Premier League
Prince (musician)
Prince Harry, Duke of Sussex
Production of the James Bond films
Prince William, Duke of Cambridge
Prague

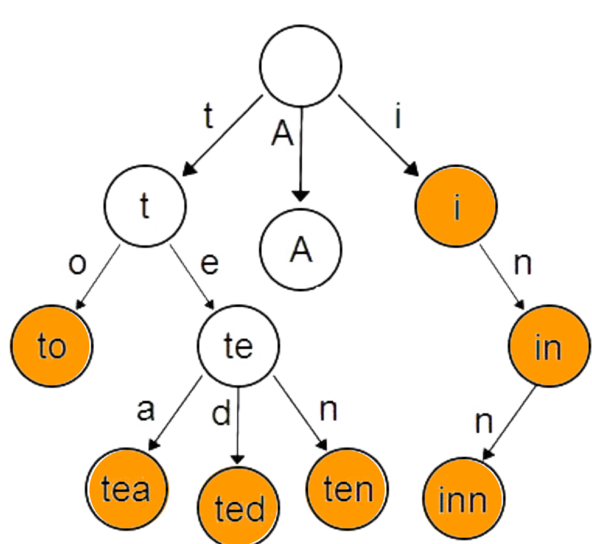
Given a prefix, autocomplete returns a list of articles that start with the prefix.

Implemented using a **trie**

What is a trie?

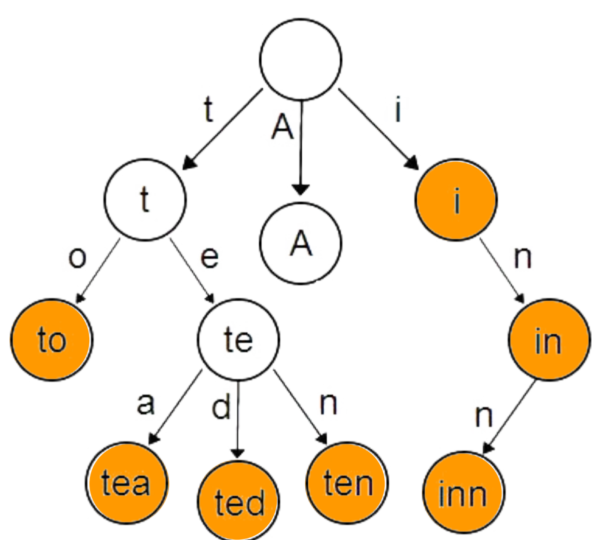
Tries

- Pronounced [tɹaɪ] ("try")*
- A trie is a special kind of tree made up of nodes
- Each trie has:
  - a **boolean value** representing whether the node represents a word
  - some sort of mapping of **characters** to `TrieNode*` **pointers**
  - Each node represents a sequence of characters that's given by its position in the trie.



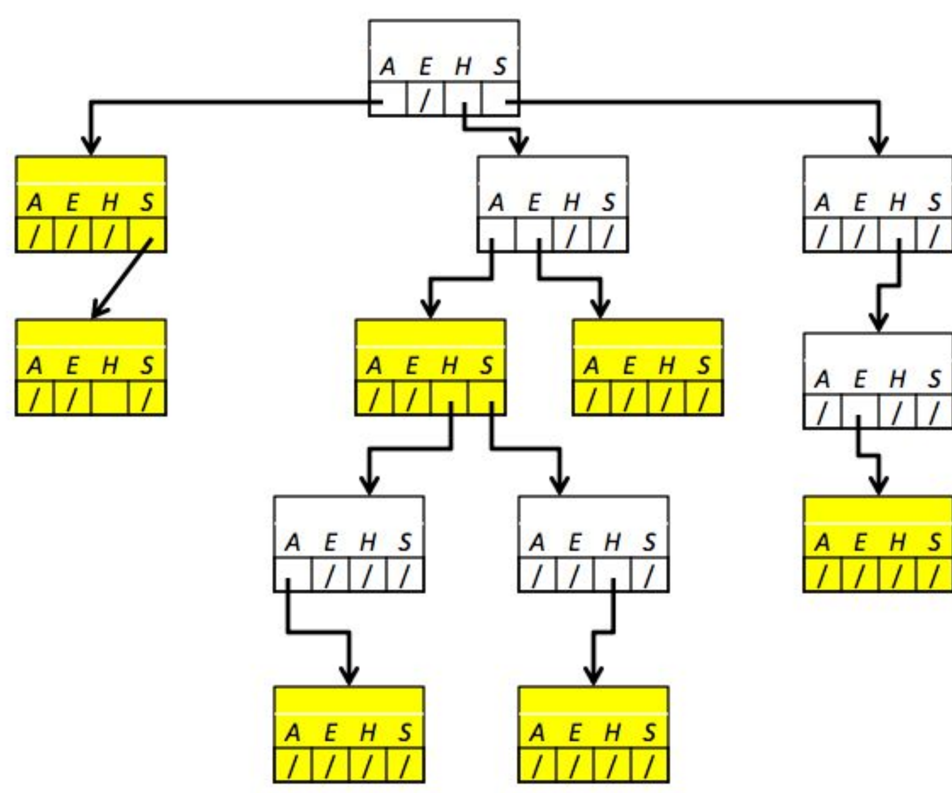
Tries

- **Note:** each node doesn't know what letter it represents! The word that a node represents depends only on its position in the trie.
  - (The words written inside the nodes aren't really part of the `TrieNode`; they're just included in the diagram to make what word each node represents more clear.)



What words exist in the trie?

Yellow = word in the trie



(credit: Nick Troccoli)

What words exist in the trie?

Yellow = word in the trie

a

as

ha

haha

hash

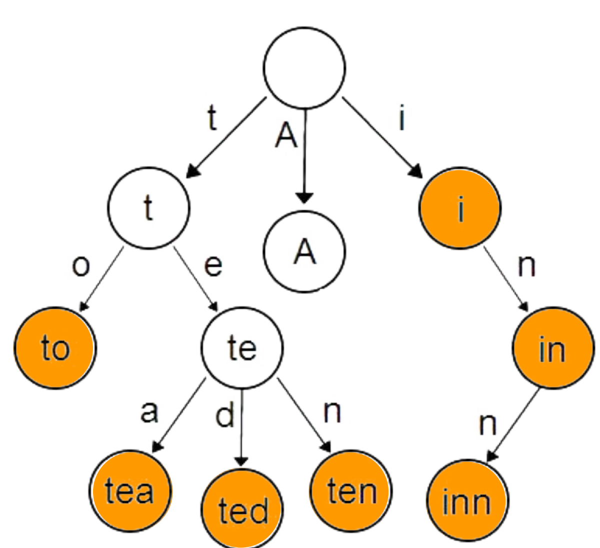
he

she

(credit: Nick Troccoli)

And one more thing...

- Tries are optimized for prefix searches.
- The `Lexicon` class is built using a trie!
- How would we go about searching for all words that begin with 'te' in this trie? →



Autocomplete

- A class to provide autocomplete results, using a trie to store Wikipedia titles

```
class Autocomplete {
    public:
        Autocomplete();
        ~Autocomplete();
        void add(const std::string& word);
        Vector suggestionsFor(const std::string& prefix, int maxHits) const;
    private:
        /* … up to you to decide how to store the trie! … */
};

struct TrieNode {
    bool isWord;
    /* store your mappings here! */
}
```

Autocomplete

- Uses a **trie** to store the most common article titles from Wikipedia
- Has two functions:
  - `void add(const std::string& word);`
    - Adds a new title to the trie.
  - `Vector<std::string> suggestionsFor(const std::string& prefix, int maxHits);`
    - Called when the user types some text in the search bar.
    - Returns a list of titles that begin with the prefix.
    - If there are `maxHits` or fewer suggestions, return all of them. If there are > `maxHits` suggestions, return only `maxHits` of them.

Autocomplete — Hints

- You can return autocomplete suggestions in any order you like.
- **Do not assume** that the `char` values in the titles will be English letters.
    - `NAK` (non-acknowledgement), `BEL` (bell sound!), and `TAB` are all characters!
    - With non-Roman scripts, a `char` may represent part of a glyph.
    - Don't assume anything about what's in your `char`s and you'll be fine. :)

Autocomplete — Hints

- Is there any programming strategy that might be useful for traversing a trie?
  - (one whose name starts with R, perhaps?)
- Efficiency:
  - Avoid spending time gathering more strings than you'll be allowed to return.
  - Don't go down branches of the trie that don't produce words with the given prefix.
- Be sure not to leak any memory! Use the **`TRACK_ALLOCATIONS_OF`** macro to record how many times your nodes are allocated and deallocated, and make sure that everything balances.
- This is a tricky assignment—we recommend you write **at least 4** test cases!

Part 3: Line Manager

Lines

- Each Wikipedia page loaded by MiniBrowser is broken down into lines of text.

US Constitution

We the People of the United
States, in Order to form a more
perfect Union, establish Justice,
insure domestic Tranquility,
provide for the common defence,
promote the general Welfare, and
secure the Blessings of Liberty to
ourselves and our Posterity, do
ordain and establish this
Constitution for the United States
of America.

US Constitution

We the People of the United
States, in Order to form a more
perfect Union, establish Justice,
insure domestic Tranquility,
provide for the common defence,
promote the general Welfare, and
secure the Blessings of Liberty to
ourselves and our Posterity, do
ordain and establish this
Constitution for the United States
of America.

More on Lines

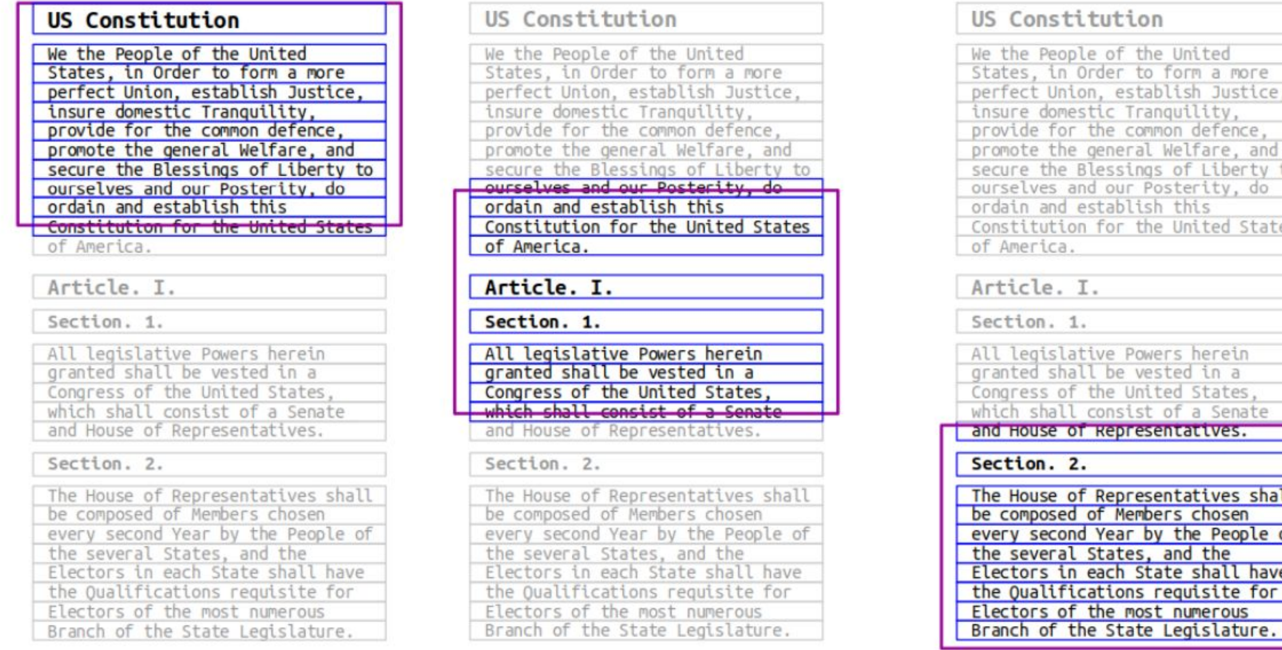
- All lines have the same width
- Not all lines have the same height.
- There can be vertical space between lines.
- Lines **cannot** overlap each other.

US Constitution

We the People of the United
States, in Order to form a more
perfect Union, establish Justice,
insure domestic Tranquility,
provide for the common defence,
promote the general Welfare, and
secure the Blessings of Liberty to
ourselves and our Posterity, do
ordain and establish this
Constitution for the United States
of America.

US Constitution

We the People of the United
States, in Order to form a more
perfect Union, establish Justice,
insure domestic Tranquility,
provide for the common defence,
promote the general Welfare, and
secure the Blessings of Liberty to
ourselves and our Posterity, do
ordain and establish this
Constitution for the United States
of America.



- We don’t always need to render all of the lines…
- Which ones do we really need? Only the ones visible in our window

More on Lines

Lines are represented by a struct:

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

More on Lines

Lines are represented by a struct:

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

Article. I.

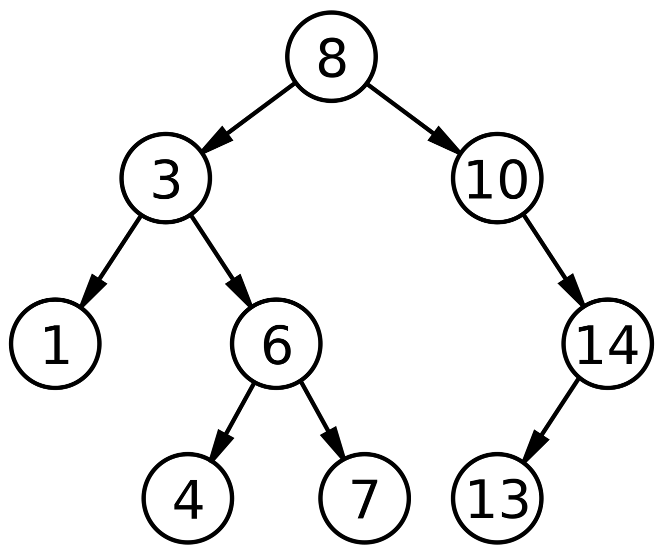
» **Numerically lower Y-coordinates are higher on the display!** «

What is LineManager?

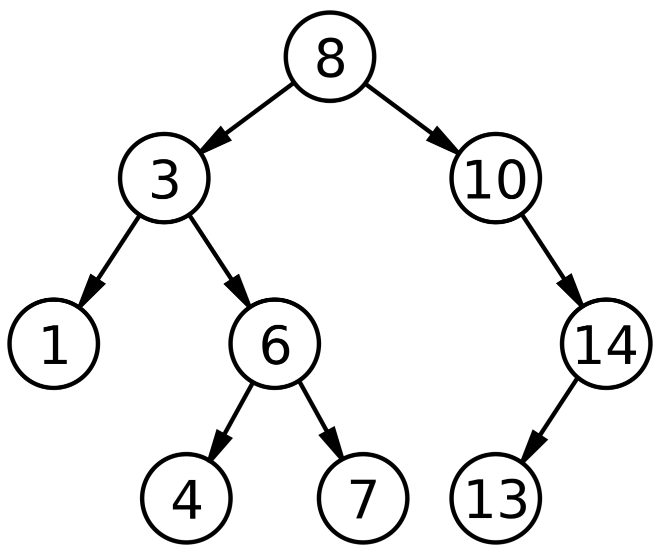
- **LineManager** is a class that stores lines
- Lines must be stored as a **binary search tree**

```
class LineManager {
     public:
          LineManager(const Vector& lines);
          ~LineManager();
          double contentHeight() const;
          Line* lineAt(double y) const;
          Vector linesInRange(double topY, double bottomY) const;
     private:
          /* binary tree stuff!*/
};
```

Binary Search Trees

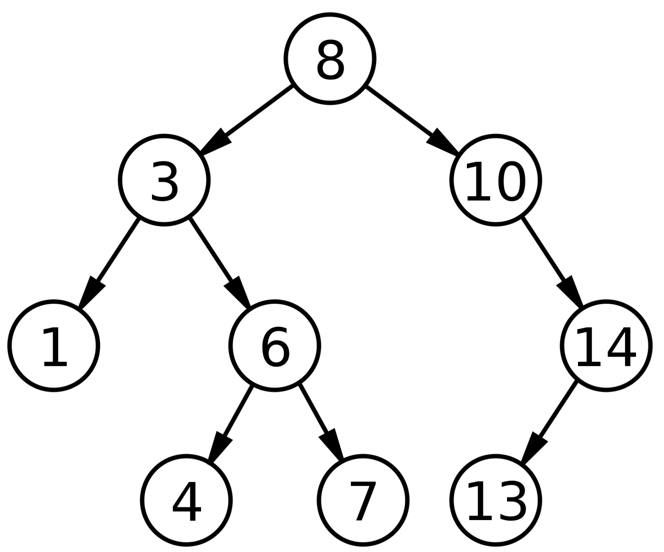


Binary Search Trees



- Left child must be less than parent
- Right child must be greater than parent

Binary Search Trees — LineManager!



- Left child must be **above** parent
- Right child must be **below** parent

Building Your Tree

- How do we build a binary tree?
  - Structs containing a value and pointers to left and right children
  - If the left pointer is `nullptr`, there's no left child; same for the right pointer.
  - Here's an example of a binary tree node that stores an int:

```
struct BinaryTreeNode {
    int value;
    BinaryTreeNode* left;
    BinaryTreeNode* right;
}
```

Building Your Tree — Things to Think About

- You should aim to construct as balanced of a tree as possible here, since, operations on a binary search tree get really slow when the tree is imbalanced.
- (Hint: since the lines come in sorted order, what line do you want at the top of the tree? Then what strategy do we use to fill out the left and right sides?)
- **Don't compare `Line*`s—this doesn't work as expected.** Compare their coordinates instead.
- Your LineManager class should store the binary tree as a pointer. (To what?)
- There are mandatory **time bounds** — check `LineManager.h` for more details!

contentHeight()

- Returns the y-coordinate of the bottom of the last line.

```
class LineManager {
     public:
          LineManager(const Vector& lines);
          ~LineManager();
          double contentHeight() const;
          Line* lineAt(double y) const;
          Vector linesInRange(double topY, double bottomY) const;
};
```

contentHeight()

- Returns the y-coordinate of the bottom of the last line.
- Given the last line, what `line` function do we want here? **bottomY()**
- How can we traverse the tree to find the bottom line?

```
class LineManager {
    public:
        LineManager(const Vector& lines);
        ~LineManager();
        double contentHeight() const;
        Line* lineAt(double y) const;
        Vector linesInRange(double topY, double bottomY) const;
};
```

lineAt()

- Receive a Y-coordinate
- Return the line that contains the coordinate, or nullptr if none exists
- If two lines both have the Y-coordinate on their border, either is OK

```
class LineManager {
     public:
          LineManager(const Vector& lines);
          ~LineManager();
          double contentHeight() const;
          Line* lineAt(double y) const;
          Vector linesInRange(double topY, double bottomY) const;
};
```

linesInRange()

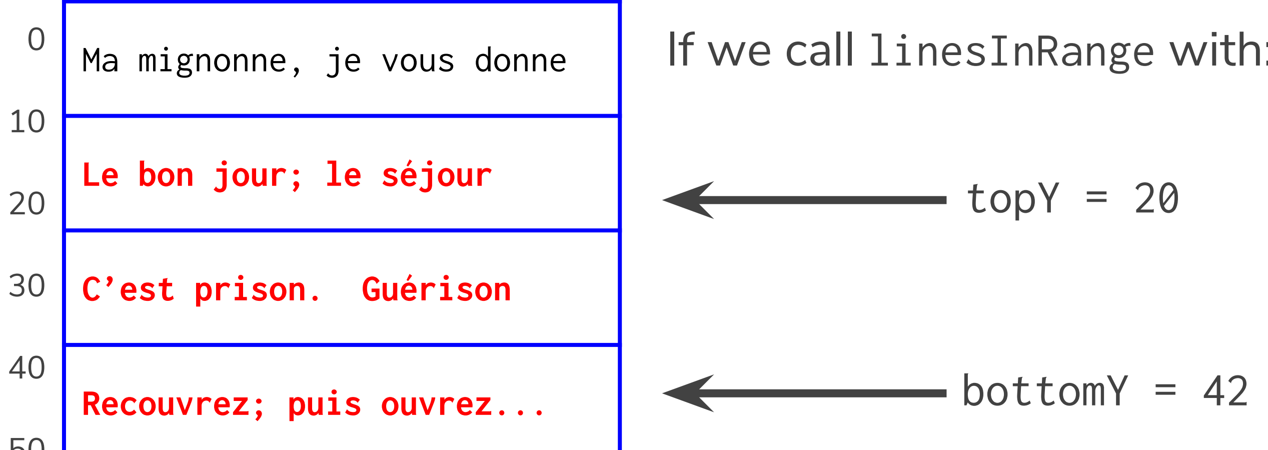
- Receive two Y-coordinates
- Return a Vector of lines that are at least partially between the two Y-coordinates

```
class LineManager {
     public:
          LineManager(const Vector& lines);
          ~LineManager();
          double contentHeight() const;
          Line* lineAt(double y) const;
          Vector linesInRange(double topY, double bottomY) const;
};
```

linesInRange(double topY, double bottomY)

- Receive two Y-coordinates
- Return a Vector of lines that are at least partially between the two Y-coordinates

`linesInRange(double topY, double bottomY)`



`linesInRange(13, 42)` should include lines #2, 3, and 4.

Assignment Due:

Friday, March 8th

pair programming is ***permitted*** :)

Any questions?