PROJECT

Web Navigator

This project, Web Navigator, simulates the navigational operations of a web browser such as :

- opening a new web page,
- navigating back a page and
- going forward a page. We will use the `Stack` class to maintain the history of visited pages with a `backPages` stack and a `nextPages` stack.

When we open a new page, we push the previous page on the `backPages` stack. When we revisit an old page and then visit a new page from there, we clear any content in the `nextPages` stack.

When we revisit a back page, we push the current page on the `nextPages` stack. Like the back button and the next button on a web browser, the back page and next page operations can be enabled or disabled depending on the state of the two stacks. For example, if the `backPages` stack is empty, the back operation is disabled and will be enabled only when the stack has content.

User input is required to:

- enter a new page to be visited,
- navigate backward or forward a page, and
- to quit the program.

The option to navigate forward or backward is conditional depending on user input and the state of the stacks. We will explain this in more detail in the relevant tasks.

At every operation other than quitting, we display information about the current page and the top element of the two stacks

Tasks

20/20 Complete

Mark the tasks as complete by checking them off

Initialization

1.

The Web Navigator needs two stacks to maintain the history of visited pages.

- Create a `const backPages` variable and assign it to a new `Stack` class to model the history of visited pages
- Create a `const nextPages` variable and assign it to a new `Stack` class to model the pages that get moved when an old page from the `backPages` stack is revisited.

Hint

```
const backPages = new Stack();
const nextPages = new Stack();
```

2.

Set the default page to be anything you like. Assign this to a global variable `currentPage`.

Hint

```
let currentPage = 'Start Page';
```

Helper Functions

3.

We will need several helper functions to help us implement the basic operations of this program. Define a function, `showCurrentPage()` to display the following:

- the action taken, based on user input
- the current page,
- the top element of the `backPages` stack, and
- the top element of the `nextPages` stack.

This function takes a string parameter, `action` based on user input.

Hint

An example of `showCurrentPage()` is provided below.

```
showCurrentPage = (action) => {
  console.log(`\n${action}`);
  console.log(`Current page = ${currentPage}`);
  console.log('Back page = ', backPages.peek());
  console.log('Next page = ', nextPages.peek());
}
```

4.

Next, we want to implement what happens when we visit a new page. The new page replaces the current page, hence, the current page has to be moved to the `backPages` stack as history. The first time we open a new page, the `nextPages` stack is empty. However, this is not always so when we have

navigated back and forth. The expected behavior of a web browser is to always clear the `nextPages` stack when a new page is visited.

Define a `newPage()` function with one parameter, `page`. Implement the following inside the function:

- push `currentPage` to the `backPages` stack
- update `currentPage` to be `page`
- clear the `nextPages` stack
- show the current page by calling the helper function defined in Task 1

Hint

An example of `newPage()` is provided below.

```
newPage = (page) => {
  backPages.push(currentPage);
  currentPage = page;

  // clear the nextPages stack
  while (!nextPages.isEmpty()) {
    nextPages.pop();
  }

  showCurrentPage("NEW: ");
}
```

5.

The next helper function we want to define is `backPage()` which is called when we navigate backward a page. This function does not accept any parameter. The steps to implement this function are:

- push the current page on the `nextPages` stack as we will no longer display it,
- remove the top item from the `backPages` stack and set it as the current page, and
- display the new current page using the helper function we created in Task 1 and pass an argument to it to denote the back operation.. Hint

Hint

An example of `backPage()` is implemented as follows:

```
backPage = () => {
  nextPages.push(currentPage);
  currentPage = backPages.pop();
```

```
  showCurrentPage("BACK: ");
}
```

6.

The last helper function we need is parallel to the `backPage()` function in Task 5. We will call this function, `nextPage()`. Try and implement this function as follows:

- push the current page on the `backPages` stack as we will no longer display it,
- remove the top item from the `nextPages` stack and set it as the current page, and
- display the new current page using the helper function we created in Task 1 and pass an argument to it to denote the next operation..

Hint

You can check your implementation with ours here.

```
nextPage = () => {
  backPages.push(currentPage);
  currentPage = nextPages.pop();
  showCurrentPage("NEXT: ");
}
```

User Interface Part 1

7.

Our user interface will be text-driven from the `bash` terminal. It will look something like this at the beginning of a program:

```
DEFAULT:
Current page = Start Page
Back page =  null
Next page =  null

Enter a url, Q|q for quit.
Where would you like to go today?
```

And something like this in the middle of the program where we have a history of back pages and forward pages:

```
BACK:
Current page = amazon.com
Back page =  Start Page
Next page =  yahoo.com

Enter a url, B|b for back page, N|n for next page, Q|q for quit.
Where would you like to go today?
```

The displayed text varies depending on the state of the navigation process. Like on a web browser, if there is a history of visited pages indicated by a non-empty `backPages` stack, the **Back** button will be enabled; if there is no history, as

indicated by an empty `backPages` stack, it will be disabled.. The *Next* button behaves similarly.

Define a global variable, `finish`, that controls the termination of a `while` loop that takes in user input. We will implement the `while` loop later in Task 10. Initialize `finish` to `false`.
Hint

```
let finish = false;
```

8.

We want to control when the back navigation and front navigation operations are enabled. Define two global variables, `showBack` and `showNext` and initialize them to `false`.
Hint

```
let showBack = false;
let showNext = false;
```

9.

When the program is started, it shows a default page. Call the helper function that does this with an appropriate argument.
Hint

We would call `showCurrentPage()` with an argument such as `'DEFAULT'`.

```
showCurrentPage('DEFAULT: ');
```

10.

The majority of the code that controls the processing of user input is executed in a `while` loop. Define a `while` loop that utilizes the `finish` global variable as a condition.
Hint

```
while (finish === false) {

}
```

11.

The processes inside the `while` loop are broken up into 3 parts:

- display the instructions to the user
- prompt the user for input
- process user input

We have declared strings that contain user input instructions called `baseInfo`, `backInfo`, `nextInfo` and `quitInfo` that will be referenced in the `while` loop.

- Define a local variable inside the `while` loop called `instructions` and initialize it to `baseInfo`.
- If `backPages` isn't empty, we want to
    - append `backInfo` to `instructions` separated by a comma
    - enable backward navigation using `showBack`
- Otherwise, we want to disable backward navigation

Hint

An example implementation is as follows:

```
  let instructions = baseInfo;
  if (backPages.peek() != null) {
    instructions = `${instructions}, ${backInfo}`;
    showBack = true;
  } else {
    showBack = false;
  }
```

12.

Parallel to Task 11, this task will implement a similar logic to the `nextPages` stack. If `nextPages` has content, we want to

- append `nextInfo` to `instructions` separated by a comma, and
- enable forward navigation Otherwise, we want to disable forward navigation

Hint

Example code is provided below:

```
  if (nextPages.peek() != null) {
    instructions = `${instructions}, ${nextInfo}`;
    showNext = true;
  } else {
    showNext = false;
  }
```

13.

Finally, we want to enable the user to quit the program by adding `quitInfo` to `instructions` and display the final format of `instructions` to the user.
Hint

Here's how we did it:

```
  instructions = `${instructions}, ${quitInfo}.`;
  console.log(instructions);
```

User Interface Part 2

14.

The next section of the user interface focuses on prompting the user for input and processing user input while inside the `while` loop. The code to prompt for user input is as follows:

```
  const response = prompt('How are you today?');
```

Perform a similar action using `question` as the prompt and save the response in a local variable, `answer`.

Hint

```
  const answer = prompt(question);
```

15.

Since we accept inputs in both lower and upper cases, we want to simplify our input processing by lower-casing our response but not override the original response. Define a local variable `lowerCaseAnswer` and initialize it to the lower-cased conversion of `answer`

Hint

To convert a string to lower case, we do the following:

```
const mixedCase = 'Hello World';
const lowerCase = mixedCase.toLowerCase();
```

16.

We are left with the task of processing user input that has been lower-cased. Our choices are: `b`, `n`, `q` or a url string. Write a conditional statement to process only the url string and display a new page based on the original typed url.

Hint

```
  if ((lowerCaseAnswer !== 'n') && (lowerCaseAnswer !== 'b') &&
(lowerCaseAnswer !== 'q')) {
    // we create a new page based on the url
    newPage(answer);
  }
```

17.

Write corresponding `else if` statements to process navigating back a page and forward a page.. Remember to check if we can actually navigate forward or backward utilizing the `showNext` and `showBack` statuses in our `else` logic.

Hint

```
  } else if ((showNext === true) && (lowerCaseAnswer === 'n')) {
    // we navigate forward a page
    nextPage();
  } else if ((showBack === true) && (lowerCaseAnswer === 'b')) {
    // we navigate back a page
    backPage();
  }
```

18.

Add additional checks if the user tries to go forward or backward a page even when the option is not available to them. Provide a user-friendly message to let the user know that they can't proceed with that option.
Hint

Add these checks after the `else if` statements in Task 17.

```
  else if (lowerCaseAnswer === 'b') {
    // invalid input to a non-available option
    console.log('Cannot go back a page. Stack is empty.');
  } else if (lowerCaseAnswer === 'n') {
    // invalid input to a non-available option
    console.log('Cannot go to the next page. Stack is empty.');
  }
```

19.

Before we finish coding, we also need to evaluate the user's desire to quit the program. Write an `else if` statement to process this input. Consider how we should terminate the `while` loop.
Hint

```
  else if (lowerCaseAnswer === 'q') {
    // we quit the program
    finish = true;
  }
```

20.

Remember to save your code at this juncture. Lastly, in the terminal, execute `node script.js` and test it out. Congratulations for a job well done!