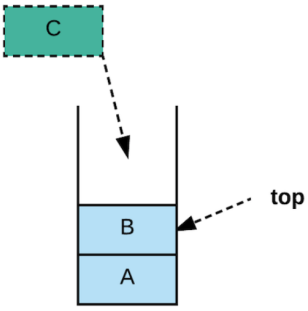
Data Structures - Stacks and Queues

Stacks

A stack is a collection of elements in which the storage and retrieval follow the **Last in First Out** (LIFO) ordering. This means that the last element added is the element on the top, and the first element added is at the bottom.

A stack is similar to an array with one significant difference: elements are only accessible from one end, the top. This means we can't randomly access elements. We can add elements to the stack and we can remove elements from the stack. But if we want to access an element mid-way in the stack, we need to pop the elements above it off the stack.

Stacks are usually thought of as vertical data structures, unlike link lists and arrays, which are horizontal. Hence, the first item—the only directly accessible item on the stack—is referred to as top of the stack.



A real-life example of Stack could be a stack of books. To get the book that's somewhere in the middle, you will have to remove all the books placed at the top of it. Also, the last book you added to the Stack of books is at the top!

A typical implementation of stack supports the following operations:

- push(): adds an element to the stack
- pop(): removes the last inserted element from the stack

What are stacks used for?

There are many famous algorithms, such as Depth First Search and the Expression Evaluation Algorithm, which harness the functionality of stacks.

Stacks are used:

- To backtrack to the previous task/state, for example in recursive code.

- To store a partially completed task, for example, when you are exploring two different paths on a Graph from a point while figuring out the smallest path to the target.

Benefits of Stacks

Stacks allow for constant-time adding and removing of the top item. Constant-time, or O(1), is a very performant runtime.

These actions are constant-time because we don’t need to shift any items around to pop off the top item in the stack.

Downsides of Stacks

Stacks, unfortunately, don't offer constant-time access to the nth item in the stack, unlike an array.

So if we want to access the third item, we would have to pop off each item in our stack until we reach the 3rd item, and if we want to access the first item in the stack (the bottom element) we have to iterate through each book on top of it and pop it off; this has a worst-case runtime of O(n) where n is the number of items in the stack.

Example of Using a Stack During an Interview

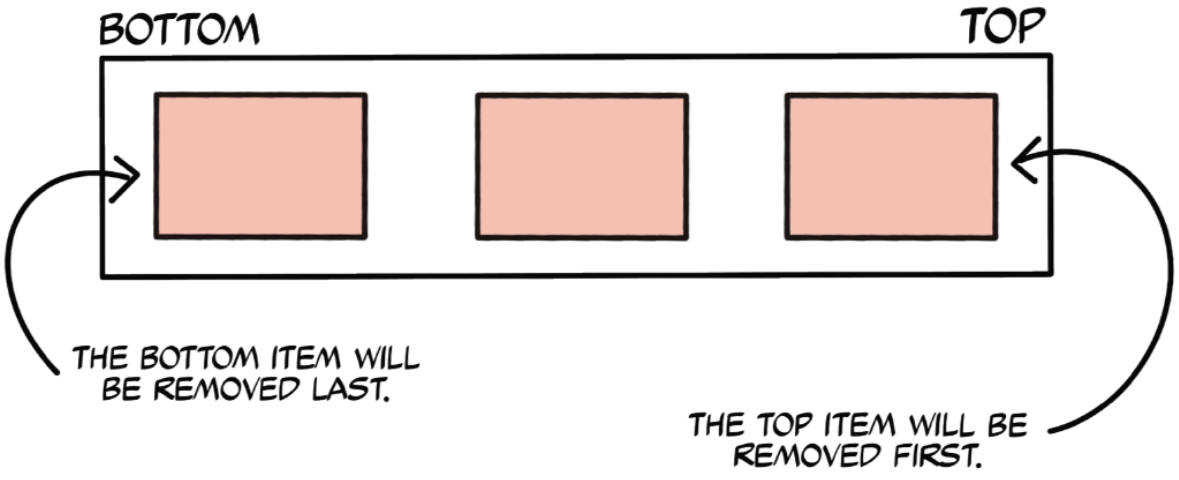
Imagine you're building a navigation component for a product website we'll call "Rainforest". Users can navigate from the home page to different product pages within the site.

For example, a user flow might be Home > Kitchen > Small Appliances > Toasters. Build a navigation component that displays the list of previously visited pages as well as a back button which lets the user go back to the previous page.

A stack would be an optimal solution for this question because the top-most item in a stack is the most recently added item (in this case the last page that was visited).

So when we want to navigate backwards, we can pop off the last item in the stack and render the state of the previous page.

Your solution should display a navigation with four links, a history list that displays the previously visited pages, a current page paragraph which displays the name of the currently displayed page, and a go back button which, when clicked, navigates to the previously visited page.



Implementation with Array:

```
class Stack {
   constructor() {
      this.store = [];
      this.top = 0;
   }

   push(element) {
      return this.store.push(element);
   }

   pop() {
      return this.store.pop();
   }

   getTop() {
        if (this.store.length == 0) return null;
        return this.top;
   }

   isEmpty() {
        return this.store.length == 0;
   }

   size() {
       return this.store.length;
   }

}
```

Arrays can be used to implement Stacks because we are only concerned with the very end of the array and push/pop functionality which is O(1) time complexity.

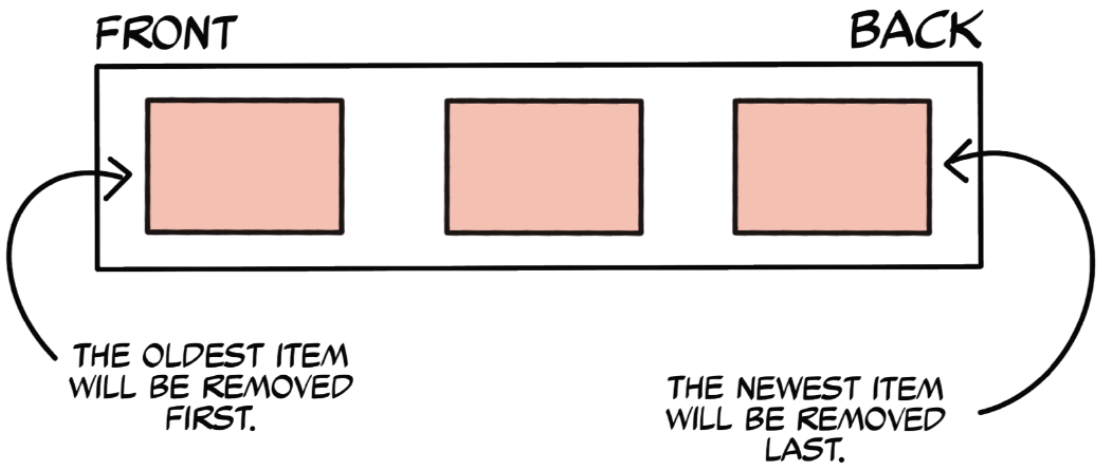
Implementation with Linked List:

```
class Node {
  constructor(value, next) {
    this.value = value;
    this.next = next;
  }
}

class Stack {
   constructor() {
      this.top = 0;
   }

   push(element) {
      this.top = new Node(value, this.top);
      return this;
   }

   pop() {
      const popped = this.top;
      this.top = popped.next;
      return popped.value;
   }
}
```

Queue

Similar to the stack, a queue is another linear data structure that stores the elements in a sequential manner. The storage and retrieval in a queue follows the **FIFO** principle, which is short for First in First Out.

In other words, in a queue, the first element inserted is the one that comes out first. You can think of a queue as a pipe with both ends open. Elements enter from one end (back) and leave from the other (front).



An example of a queue is the fast-food service at McDonald's. You line up, and service is provided in the order that you (and everyone else) lined up. If you are first to line up, you get served first.

A more practical example of a queue is the event loop of a web browser. As different events are being triggered (for example, the click of a button), they are added to an event loop's queue and handled in the order they entered the queue.

What are queues used for?

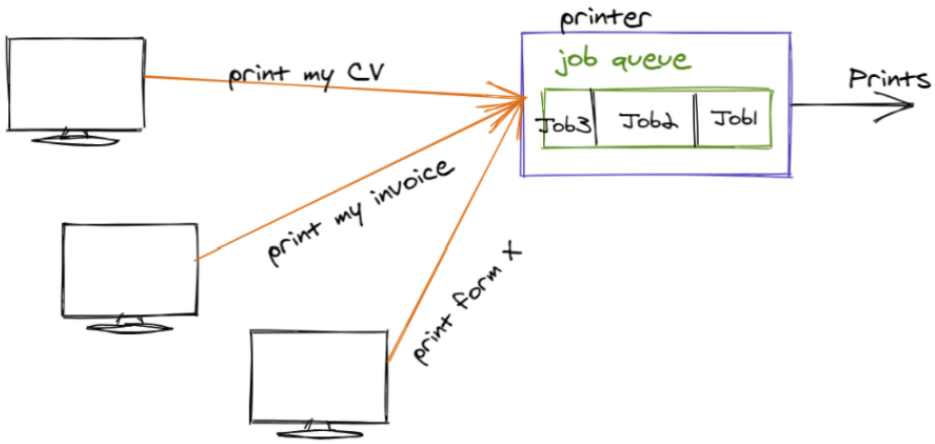
Queues are also used as a component in many other algorithms. Queues are used in graph or tree breadth-first search, for example, when we need to store a list of the nodes that we need to process later. Each time we process a node, we add its adjacent nodes or children to the end of the queue.

- We want to prioritize something over another
- A resource is shared between multiple devices

Most operating systems also perform operations based on a **Priority Queue** that allows operating systems to switch between appropriate processes. They are also used to store packets on routers in a certain order when a network is congested. Implementing a cache also heavily relies on queues.

Queues are also found in different messaging applications where messages are often generated faster than they are processed.

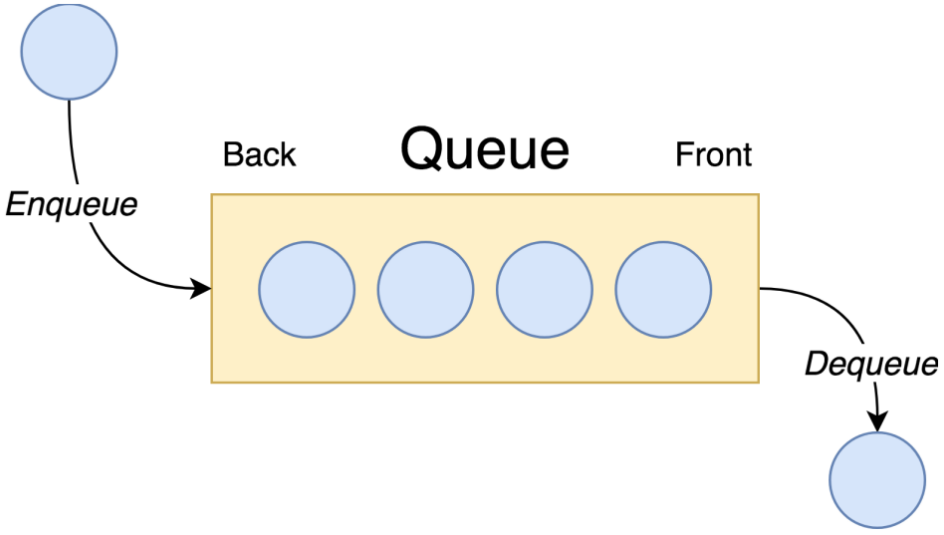
For example, when you want to parse a lot of web pages or send a lot of emails, you might want to put those web pages or emails into a queue for further processing and process them one by one without worrying that some of the data is being lost or messages dropped.



The main functions of a Queue include the following:

- enqueue(data):
    - Adds data to a queue (insertion) - **back**

- dequeue():
    - Removes the oldest data added to a queue (deletion) - **front**



Note: Operations such as searching or updating elements are not common for queues. There are only two things we're interested in when working with queues: the beginning and the end of it.

Implementation with an Array:

```
class Queue {
  constructor() {
    this.queue = [];
  }

  enqueue(item) {
    this.queue.push(item);
  }

  dequeue() {
    return this.queue.shift();
  }

  get length() {
    return this.queue.length;
  }

  isEmpty() {
    return this.length === 0;
  }

}
```

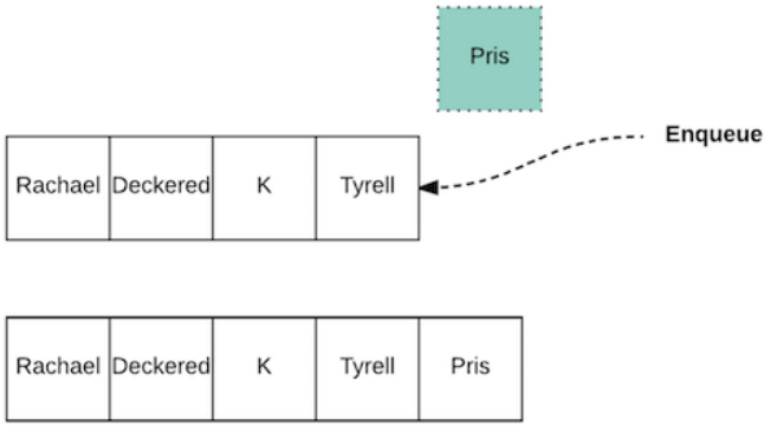
Implementation with Linked List:

```
class Node {
  constructor(value) {
    this.value = value;
    this.next = null;
  }
}

class Queue {
  constructor() {
    this.first = null;
    this.last = null;
 }
}
```

Queue insertion

The insertion operation in a queue is limited to only one place: the end of the queue. This operation is called **enqueue()**.

For example, in the following illustration, suppose that you have an existing queue, and you want to add an item Pris in the queue. The only place where Pris will be added is the end of the queue.

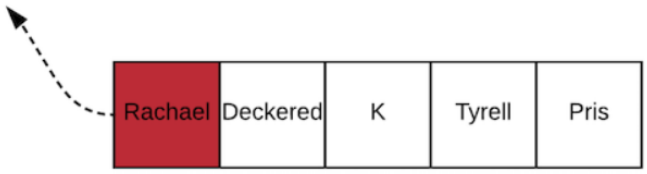


```
enqueue(value) {
    const newNode = new Node(value);

    if (this.first) {
       this.last.next = newNode;
    } else {
       this.first = newNode;
    }
    this.last = newNode;
}
```

Queue removal

The removal operation in a queue is limited to only one place as well: the beginning of the queue. This operation is called **dequeue()**.

For example, in the following illustration, suppose that you have an existing queue. You can remove an item from the beginning of the queue only. So, if you want to remove something from this queue, Rachael will be removed.



```
dequeue() {
  if (this.first) {
    const dequeued = this.first;
    this.first = dequeued.next;

    if (dequeued === this.last) {
      this.last = null;
    }
    return dequeued.value;
  }
}
```

Stacks and Queues Problems

Problem #1: Check for Palindrome

Write an algorithm that uses a stack to determine whether a given input is a palindrome.

If a string has an odd number of characters, you will essentially ignore the middle character. This is because the middle character doesn't determine whether the string is a palindrome. It's the characters on either side of the middle value that determine whether the sentence is a palindrome.

```
const isPalindrome = (sentence) => {

  let middle = Math.floor(sentence.length / 2);

  const stack = new Stack();

  for (let index = 0; index < middle; index++) {
    stack.push(sentence[index]);
  }

  middle += sentence.length % 2 === 0 ? 0 : 1;

  for (let index = middle, limit = sentence.length;
index < limit; index++) {
    if (sentence[index] !== stack.pop()) {
      return false;
    }
  }

  return true;
};
```

Problem #2: Balanced Matching Parentheses

Write an algorithm that uses a Stack that accepts an expression as a string and returns true if the parentheses in the expression match and false otherwise.

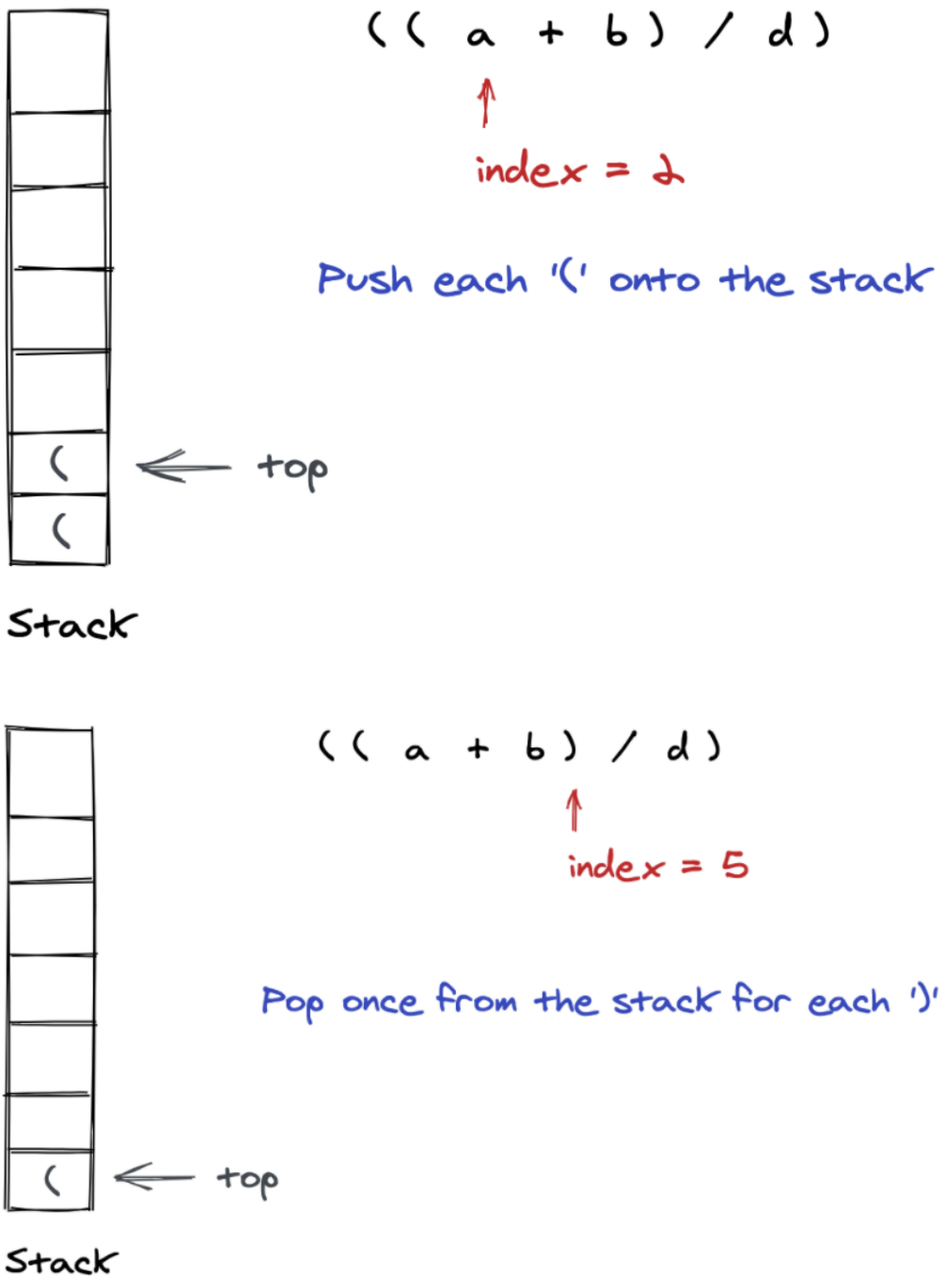
Approach:

Each closing parenthesis must be matched to an opening parenthesis for this to work. This is where using a stack is useful.

The basic concept is to iterate through each character in the expression. If the character is (, then push it onto the stack.

When iterating through a string, when you reach a) character, you will pop one (character off of the stack. This is how you match opening and closing parentheses. As illustrated below, index=5 is a closing parenthesis, so you pop one opening parenthesis off of the stack.

If the stack is empty and you come across a) in your string, then you know that the expression is invalid, so return false.



Video review: https://www.youtube.com/watch?v=CCyEXcNamC4

```
balancedParentheses(expression) {
     const stack = new Stack();

    for (let index = 0, limit = expression.length;
index < limit; index++) {
      if (expression[index] === "(") {
        stack.push("(");
      } else {
          if (expression[index] === ")") {
             if (stack.top) {
               stack.pop();
             } else {
               return false;
             }
          }
      }
    }

    return !stack.top;
 }
```

Quiz Questions:

1. Which principle are elements in a stack inserted and removed by?

Last in, First Out

2. What is Stack underflow and Stack overflow caused by?

Stack underflow is when we try to pop an item from an empty stack. Stack overflow happens when we try to push more items on a stack than its max capacity.

3. When dequeuing an element, the front pointer points to which element?

The front pointer points to the first element.

4. Using a linked list to implement a queue has a time complexity of:

O(n)

Resources

https://www.youtube.com/watch?v=1AJ4ldcH2t4

https://www.youtube.com/watch?v=wtynhUwS5hI&t=230s

https://www.youtube.com/watch?v=A3ZUpyrnCbM

https://www.youtube.com/watch?v=Wg8IiY1LbII