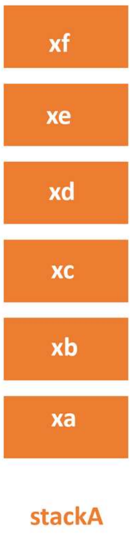
Stacks

Stack *overflow*

Every stack has a size that determines how many nodes it can accommodate. Attempting to push a node in a full stack will result in a stack overflow. The program may crash due to a stack overflow.
A stack is illustrated in the given image.
`stackA.push(xg)` will result in a stack overflow since the stack is already full.



The *stack* data structure

A *stack* is a data structure that follows a last in, first out (LIFO) protocol. The latest node added to a stack is the node which is eligible to be removed first. If three nodes (`a` , `b` and, `c`) are added to a stack in this exact same order, the node `c` must be removed first. The only way to remove or return the value of the node `a` is by removing the nodes `c` and `b` .

Main methods of a *stack* data structure

The stack data structure has three main methods: `push()` , `pop()` and `peek()` . The `push()` method adds a node to the top of the stack. The `pop()` method removes a node from the top of the stack. The `peek()` method returns the value of the top node without removing it from the stack.

Java Stack: pop()

The `.pop()` method of the Java `Stack` class removes the node at the top of the stack. It does so using the `LinkedList` method `.removeHead()` . It then decreases the stack `size` variable, returns the data of the removed node, and throws an error if the stack is empty. The helper method `.isEmpty()` is used to verify this.

```
public String pop() {
  if (!this.isEmpty()) {
      String data =
this.stack.removeHead();
      this.size--;
      return data;
  } else {
      throw new Error("Stack is empty!");
  }
}
```

Java Stack: Constructors

The main Java `Stack` class constructor instantiates these variables:

- A `stack` in the form of a `LinkedList`
- A `maxSize` integer that determines the maximum number of nodes in the stack
- A `size` integer that keeps track of the current size of the stack

Another `Stack` constructor can be written that does not take any arguments. This constructor is used if no `maxSize` value is specified as a parameter. It sets the `maxSize` variable to `Integer.MAX_VALUE` , which is the maximum integer in Java. This is stored in a variable `DEFAULT_MAX_VALUE` . This effectively creates an unbounded stack.

```
public Stack() {
    this(DEFAULT_MAX_SIZE);
}

public Stack(int maxSize) {
    this.stack = new LinkedList();
    this.size = 0;
    this.maxSize = maxSize;
}
```

Java Stack: Helper Methods

A Java `Stack` class can implement two helper methods to determine actions that should be taken with the stack:

- `.hasSpace()` returns a `boolean` representing if there is space left in a bounded stack. It is used within the `Stack` `.push()` method.
- `.isEmpty()` returns a `boolean` representing whether the stack is empty or not. It is used within the `Stack` `.pop()` and `.peek()` methods.

```
public boolean hasSpace() {
    return this.size < this.maxSize;
}

public boolean isEmpty() {
    return this.size == 0;
}
```

Java Stack: peek()

The `.peek()` method of the Java `Stack` class examines, but does not remove, the top node of the stack. It returns the top node value if there is one, and returns `null` if the stack is empty. The top node is accessed using the `head` attribute of the linked list `stack` . The top node's data is accessed using the `data` attribute of the `head` node.

```
public String peek() {
    if (this.isEmpty()) {
        return null;
    } else {
        return this.stack.head.data;
    }
}
```

Java Stack: push()

A key method of the Java `Stack` class is `.push()` . This method takes a single String argument, `data` , and adds this value to the top of the stack using the `LinkedList` method `.addToHead()` . It then increases the `size` variable and throws an error if the stack is full to ensure that the stack does not overflow with nodes. It verifies this using the helper method `.hasSpace()` .

```
public void push(String data) {
    if (this.hasSpace()) {
        this.stack.addToHead(data);
        this.size++;
    } else {
        throw new Error("Stack is full!");
    }
}
```

Java Stack Behavior

A stack can be implemented in Java by creating a `Stack` class with these methods. Each adjusts the stack in a different way.

- One constructor initializes an internal `LinkedList` for storage and a `size` and `maxSize` for tracking stack size.
- The other constructor initializes a stack with a `maxSize` property value of

```
public class Stack {

    public LinkedList stack;
    public int size;
    static final int DEFAULT_MAX_SIZE =
Integer.MAX_VALUE;
```

Integer.MAX_VALUE by default.

- .hasSpace() determines if the stack has space for more data.
- .isEmpty() determines if the stack has any data.
- .push() adds new data to the top of the stack.
- .pop() removes the top element of the stack and return its value.
- .peek() looks at but does not remove the element at the top of the stack.

```
    public int maxSize;

    public Stack() {
        this(DEFAULT_MAX_SIZE);
    }

    public Stack(int maxSize)

    public boolean hasSpace()

    public boolean isEmpty()

    public void push(String data)

    public String pop()

    public String peek()
}
```