Advanced Programming Techniques in Java

COSI 12B

Recursion III & Stacks

Lecture 22

Class Objectives

- Backtracking (Section 12.5)
- Stacks (Chapter 14)

Review: Recursive Algorithm for Towers of Hanoi

Recursive Algorithm for n -Disk Problem: Move *n* Disks from the Starting Peg to the Destination Peg

`if` *n* is 1
    move disk 1 (the smallest disk) from the starting peg to the destination peg
`else`
    move the top *n* – 1 disks from the starting peg to the temporary peg (neither starting nor destination peg)
    move disk *n* (the disk at the bottom) from the starting peg to the destination peg
    move the top *n* – 1 disks from the temporary peg to the destination peg

Review: Counting Cells in a Blob - Design

Algorithm for `countCells(x, y)`

```
if the cell at (x, y) is outside the grid
    the result is 0
else if the color of the cell at (x, y) is not the abnormal color
    the result is 0
else
    set the color of the cell at (x, y) to a temporary color
    the result is 1 plus the number of cells in each piece of the blob that
    includes a nearest neighbor
```

Review: Counting Cells in a Blob -Testing

- Verify that the code works for the following cases:
  - A starting cell that is on the edge of the grid
  - A starting cell that has no neighboring abnormal cells
  - A starting cell whose only abnormal neighbor cells are diagonally connected to it
  - A "bull's-eye": a starting cell whose neighbors are all normal but their neighbors are abnormal
  - A starting cell that is normal
  - A grid that contains all abnormal cells
  - A grid that contains all normal cells

Backtracking

Backtracking

- Backtracking is an approach to implementing a systematic trial and error search for a solution
- An example is finding a path through a maze
- If you are attempting to walk through a maze, you will probably walk down a path as far as you can go
  - Eventually, you will reach your destination, or you won't be able to go any farther
  - If you can't go any farther, you will need to consider alternative paths
- Backtracking is a systematic, nonrepetitive approach to trying alternative paths and eliminating them if they don't work

Backtracking (cont.)

- If you never try the same path more than once, you will eventually find a solution path if one exists
- Problems that are solved by backtracking can be described as a set of choices made by some method
- Recursion allows you to implement backtracking in a relatively straightforward manner
  - Each activation frame is used to remember the choice that was made at that particular decision point
- A program that plays chess may involve some kind of backtracking algorithm

Finding a Path through a Maze

- Problem
  - Use backtracking to find and display the path through a maze
  - From each point in a maze, you can move to the next cell in a horizontal or vertical direction, if the cell is not blocked

Finding a Path through a Maze (cont.)

- Analysis
  - The maze will consist of a grid of colored cells
  - The starting point is at the top left corner (0,0)
  - The exit point is at the bottom right corner `(getNCols() - 1, getNRows() -1)`
  - All cells on the path will be `BACKGROUND` color
  - All cells that represent barriers will be `ABNORMAL` color
  - Cells that we have visited will be `TEMPORARY` color
  - If we find a path, all cells on the path will be set to `PATH` color

Recursive Algorithm for Finding Maze Path

Recursive Algorithm for `findMazePath(x, y)`

`if` the current cell is outside the maze
    return `false` (you are out of bounds)
`else if` the current cell is part of the barrier or has already been visited
    return `false` (you are off the path or in a cycle)
`else if` the current cell is the maze exit
    recolor it to the path color and return `true` (you have successfully completed the maze)
`else //` *Try to find a path from the current path to the exit:*
    mark the current cell as on the path by recoloring it to the path color
    `for` each neighbor of the current cell
        `if` a path exists from the neighbor to the maze exit
            return `true`
    // *No neighbor of the current cell is on the path*
    recolor the current cell to the temporary color (visited) and return `false`

Testing

- Test for a variety of test cases:
  - Mazes that can be solved
  - Mazes that can't be solved
  - A maze with no barrier cells
  - A maze with a single barrier cell at the exit point

Stack Abstract Data Type

Stack Abstract Data Type

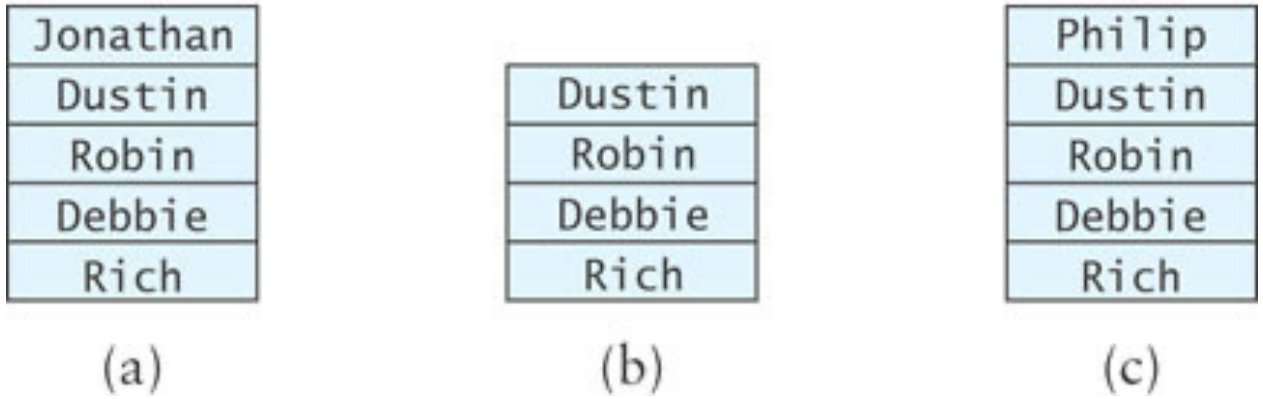
- A stack is one of the most commonly used data structures in computer science
- A stack can be compared to a Pez dispenser
  - Only the top item can be accessed
  - You can extract only one item at a time
- The top element in the stack is the one added to the stack most recently
- The stack's storage policy is *Last-In, First-Out*, or *LIFO*

Specification of the Stack Abstract Data Type

- Only the top element of a stack is visible; therefore, the number of operations performed by a stack are few
- We need the ability to
  - test for an empty stack (empty)
  - inspect the top element (peek)
  - retrieve the top element (pop)
  - put a new element on the stack (push)

Methods	Behavior
`boolean empty()`	Returns **`true`** if the stack is empty; otherwise, returns **`false`**.
`E peek()`	Returns the object at the top of the stack without removing it.
`E pop()`	Returns the object at the top of the stack and removes it.
`E push(E obj)`	Pushes an item onto the top of the stack and returns the item pushed.

A Stack of Strings



- "Rich" is the oldest element on the stack and "Jonathan" is the youngest (Figure a)
- `String last = names.peek();` stores a reference to "Jonathan" in `last`
- `String temp = names.pop();` removes "Jonathan" and stores a reference to it in `temp` (Figure b)
- `names.push("Philip");` pushes "Philip" onto the stack (Figure c)

Stack Applications

Finding Palindromes

- Palindrome: a string that reads identically in either direction, letter by letter (ignoring case)
  - kayak
  - "I saw I was I"
  - “Able was I ere I saw Elba”
  - "Level, madam, level"

- Problem: Write a program that reads a string and determines whether it is a palindrome

Finding Palindromes (cont.)

Data Fields	Attributes
`private String inputString`	The input string.
`private Stack<Character> charStack`	The stack where characters are stored.
Methods	**Behavior**
`public PalindromeFinder(String str)`	Initializes a new `PalindromeFinder` object, storing a reference to the parameter `str` in `inputString` and pushing each character onto the stack.
`private void fillStack()`	Fills the stack with the characters in `inputString`.
`private String buildReverse()`	Returns the string formed by popping each character from the stack and joining the characters. Empties the stack.
`public boolean isPalindrome()`	Returns **`true`** if `inputString` and the string built by `buildReverse` have the same contents, except for case. Otherwise, returns **`false`**.

Finding Palindromes (cont.)

```
import java.util.*;


public class PalindromeFinder {
  private String inputString;
  private Stack<Character> charStack = new Stack<Character>();


 public PalindromeFinder(String str) {
   inputString = str;
   fillStack();
 }
 ...
```

Finding Palindromes (cont.)

- Solving using a stack:
  - Push each string character, from left to right, onto a stack

k a y a k

```
private void fillStack() {
  for(int i = 0; i < inputString.length(); i++) {
    charStack.push(inputString.charAt(i));
  }
}
```

Finding Palindromes (cont.)

- Solving using a stack:
  - Pop each character off the stack, appending each to the `StringBuilder result`

k a y a k

```
private String buildReverse(){
 StringBuilder result = new StringBuilder();
 while(!charStack.empty()) {
  result.append(charStack.pop());
 }
 return result.toString();
}
```

Finding Palindromes (cont.)

```
   ...


    public boolean isPalindrome() {
      return inputString.equalsIgnoreCase(buildReverse());
    }
}
```

Testing

- We can test this class using the following inputs:
  - a single character (always a palindrome)
  - multiple characters in a word
  - multiple words
  - different cases
  - even-length strings
  - odd-length strings
  - the empty string (considered a palindrome)

Balanced Parentheses

- When analyzing arithmetic expressions, it is important to determine whether an expression is balanced with respect to parentheses

```
( a + b * ( c / ( d - e ) ) ) + ( d / e )
```

- The problem is further complicated if braces or brackets are used in conjunction with parentheses
- The solution is to use stacks!

Balanced Parentheses (cont.)

Method	Behavior
`public static boolean isBalanced(String expression)`	Returns **`true`** if `expression` is balanced with respect to parentheses and **`false`** if it is not.
`private static boolean isOpen(char ch)`	Returns **`true`** if `ch` is an opening parenthesis.
`private static boolean isClose(char ch)`	Returns **`true`** if `ch` is a closing parenthesis.

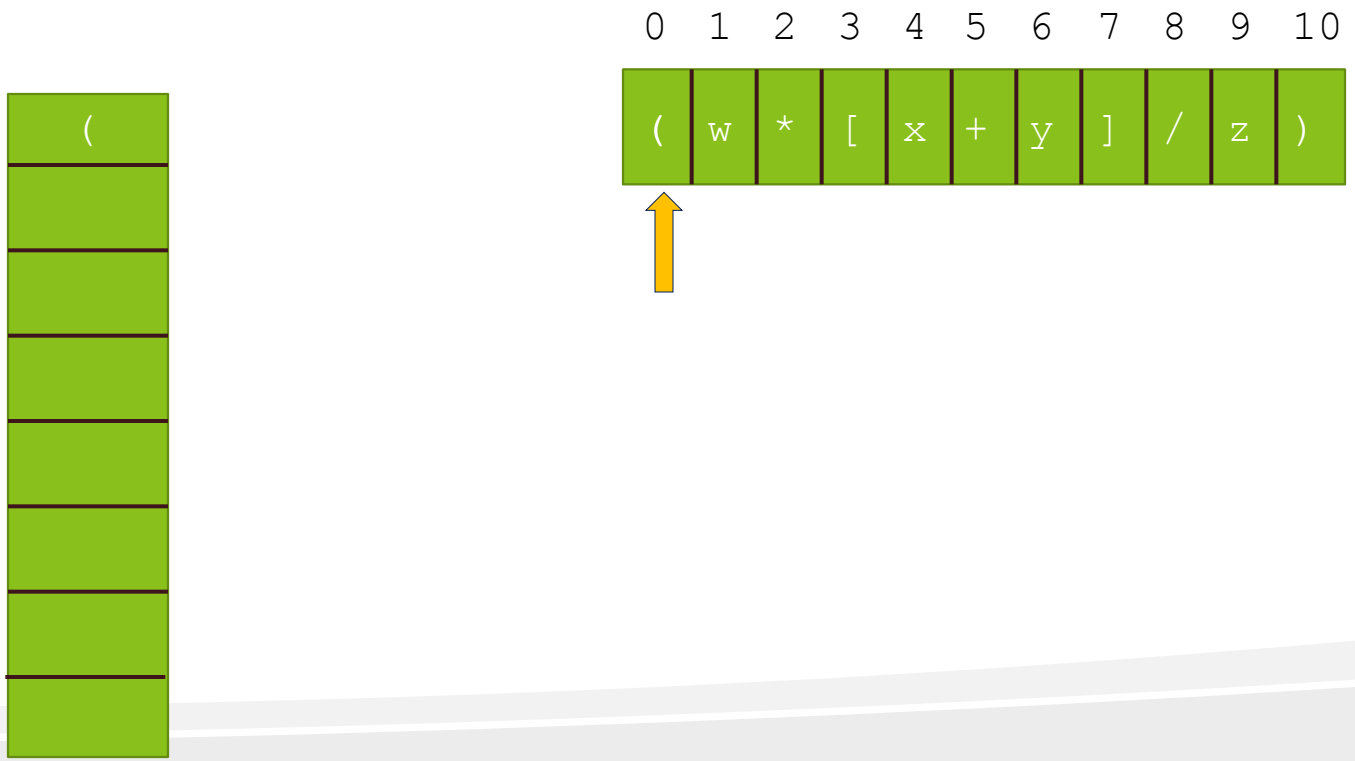
Balanced Parentheses (cont.)

Algorithm for method `isBalanced`

1. Create an empty stack of characters.
2. Assume that the expression is balanced (`balanced` is `true`).
3. Set index to 0.
4. `while balanced` is `true` and index < the expression's length
5. Get the next character in the data string.
6. `if` the next character is an opening parenthesis
7. Push it onto the stack.
8. `else if` the next character is a closing parenthesis
9. Pop the top of the stack.
10. `if` stack was empty or its top does not match the closing parenthesis
11. Set balanced to `false`.
12. Increment index.
13. Return `true` if `balanced` is `true` and the stack is empty.

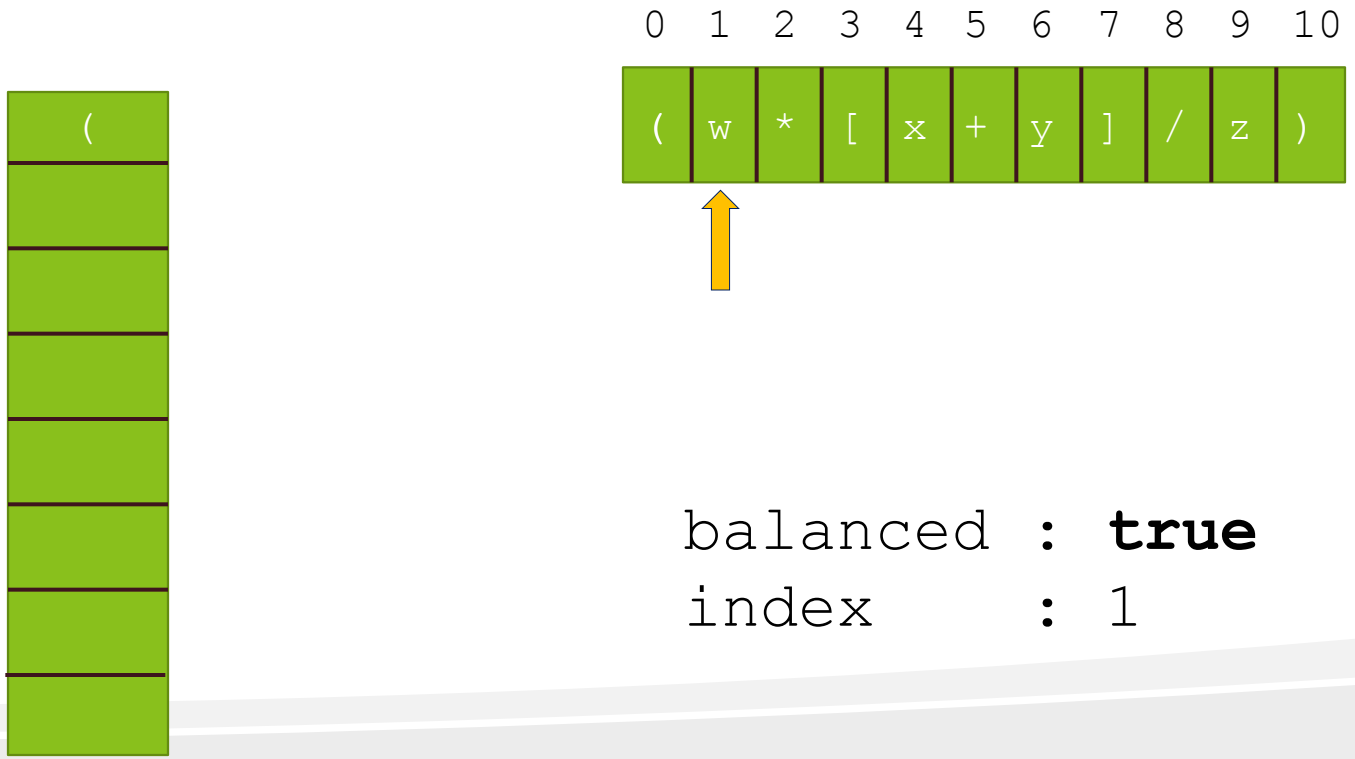
Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



Balanced Parentheses (cont.)

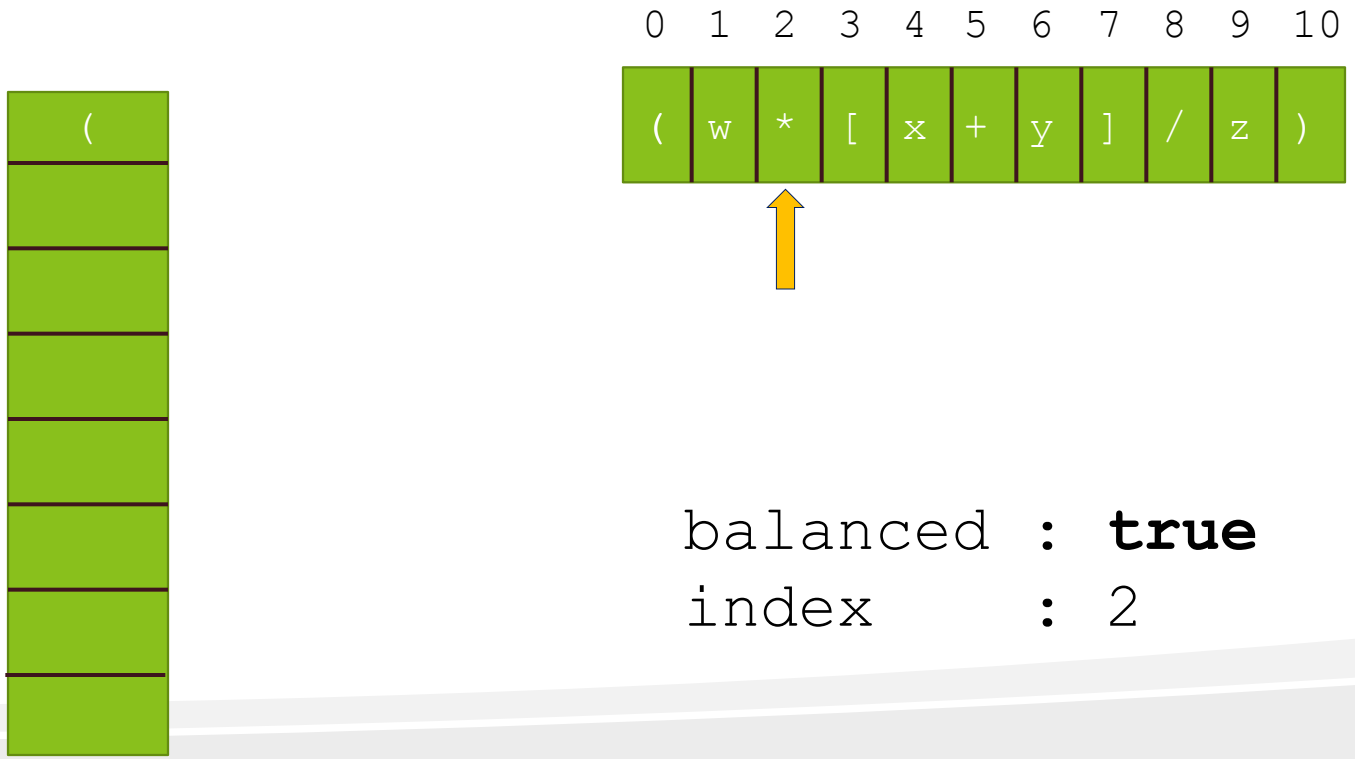
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 1
```

Balanced Parentheses (cont.)

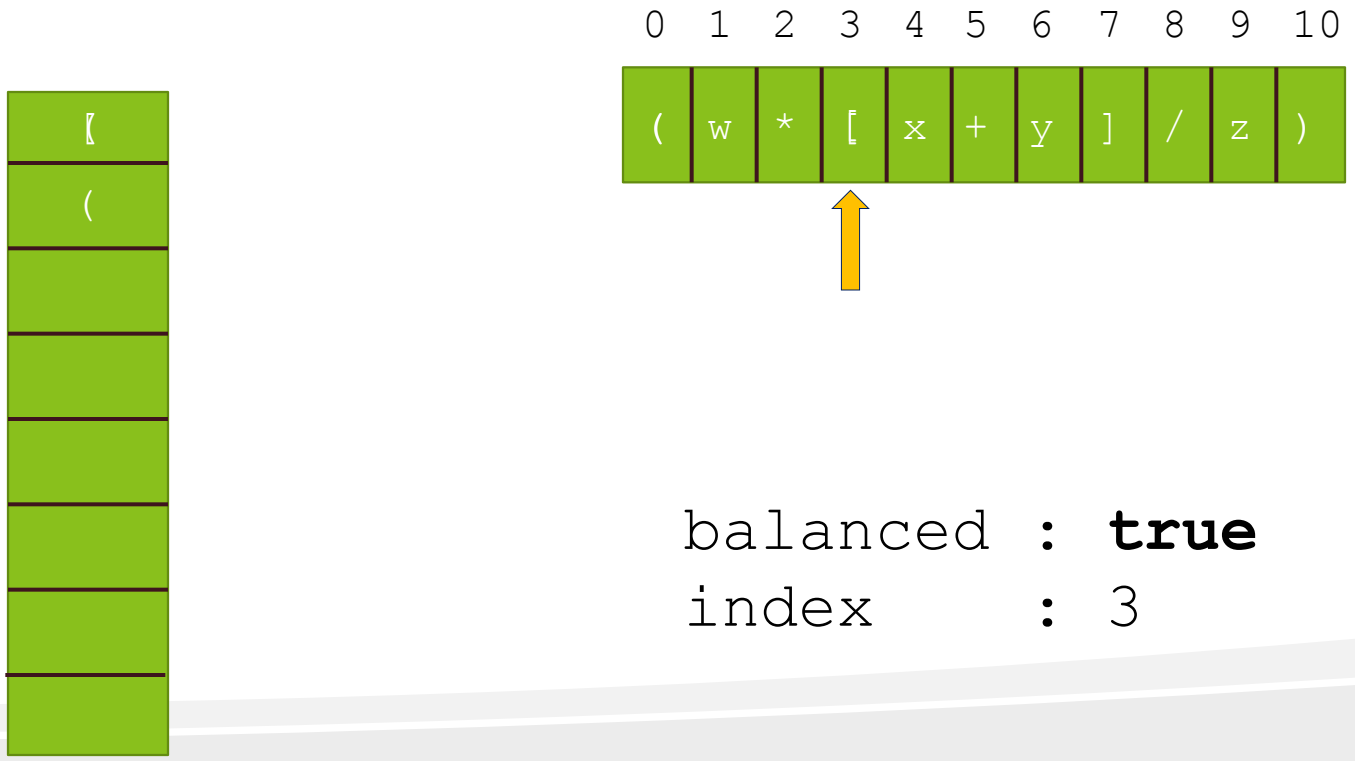
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 2
```

Balanced Parentheses (cont.)

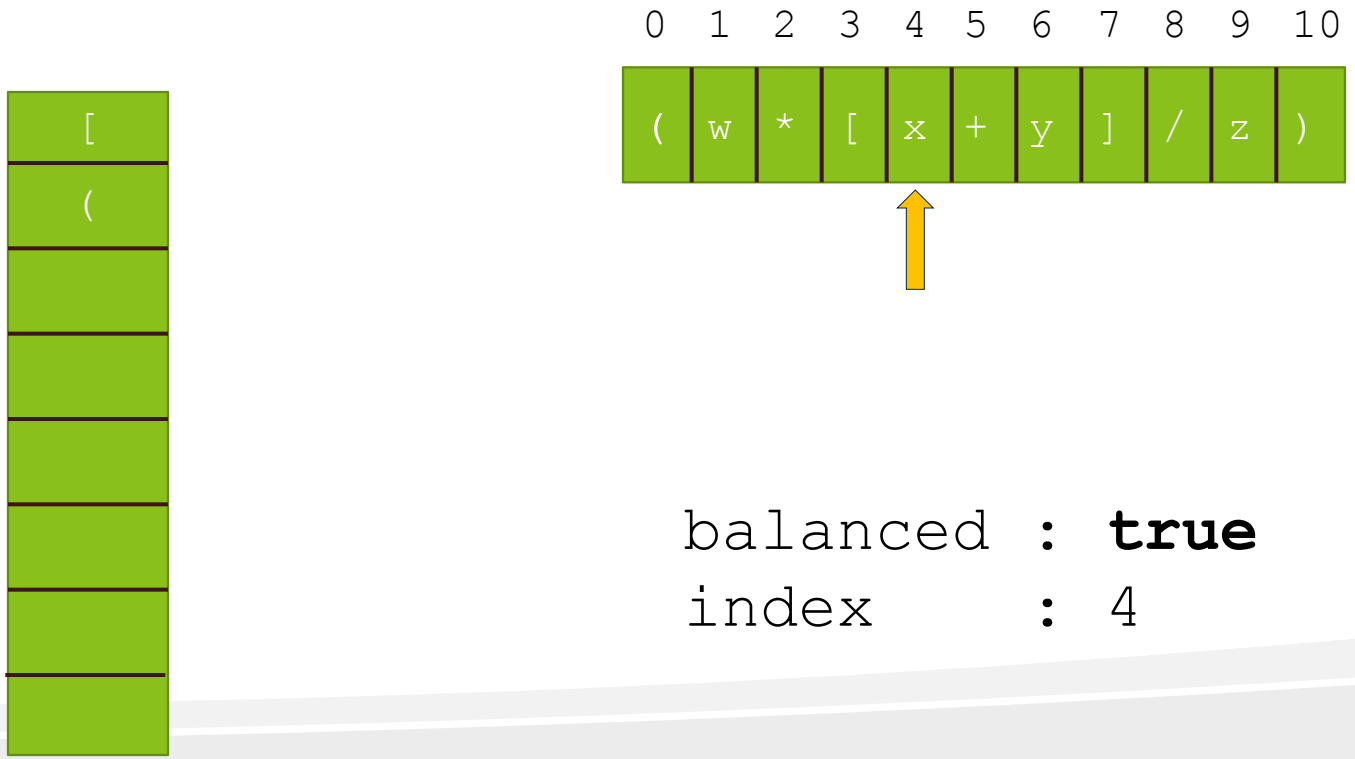
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 3
```

Balanced Parentheses (cont.)

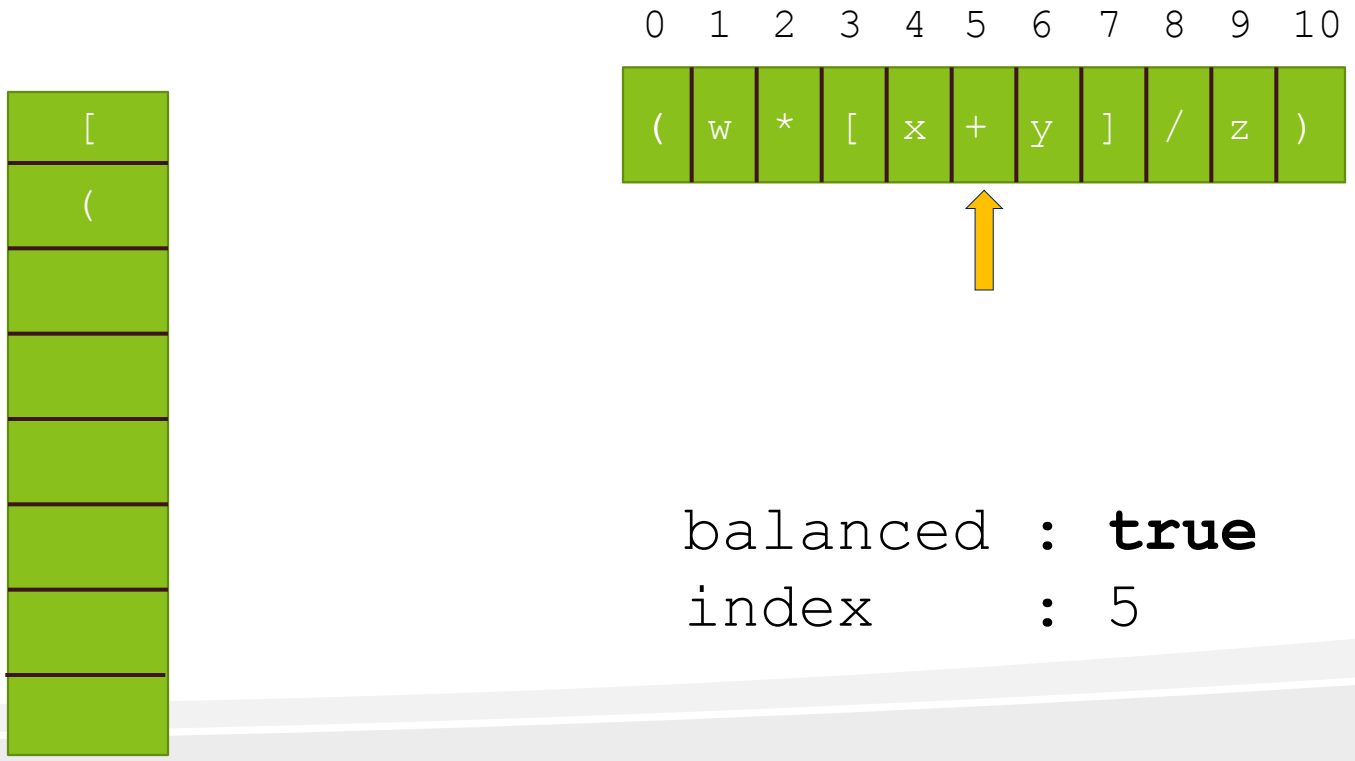
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 4
```

Balanced Parentheses (cont.)

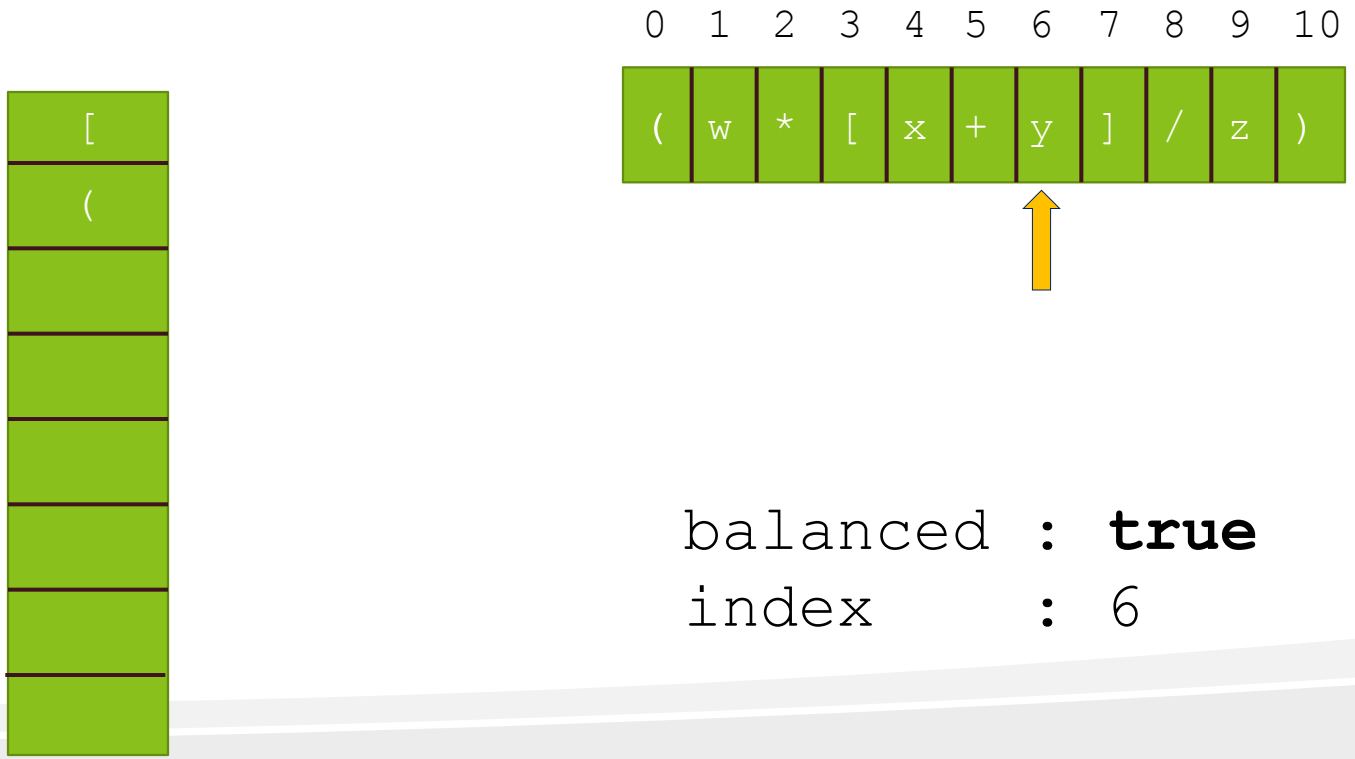
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 5
```

Balanced Parentheses (cont.)

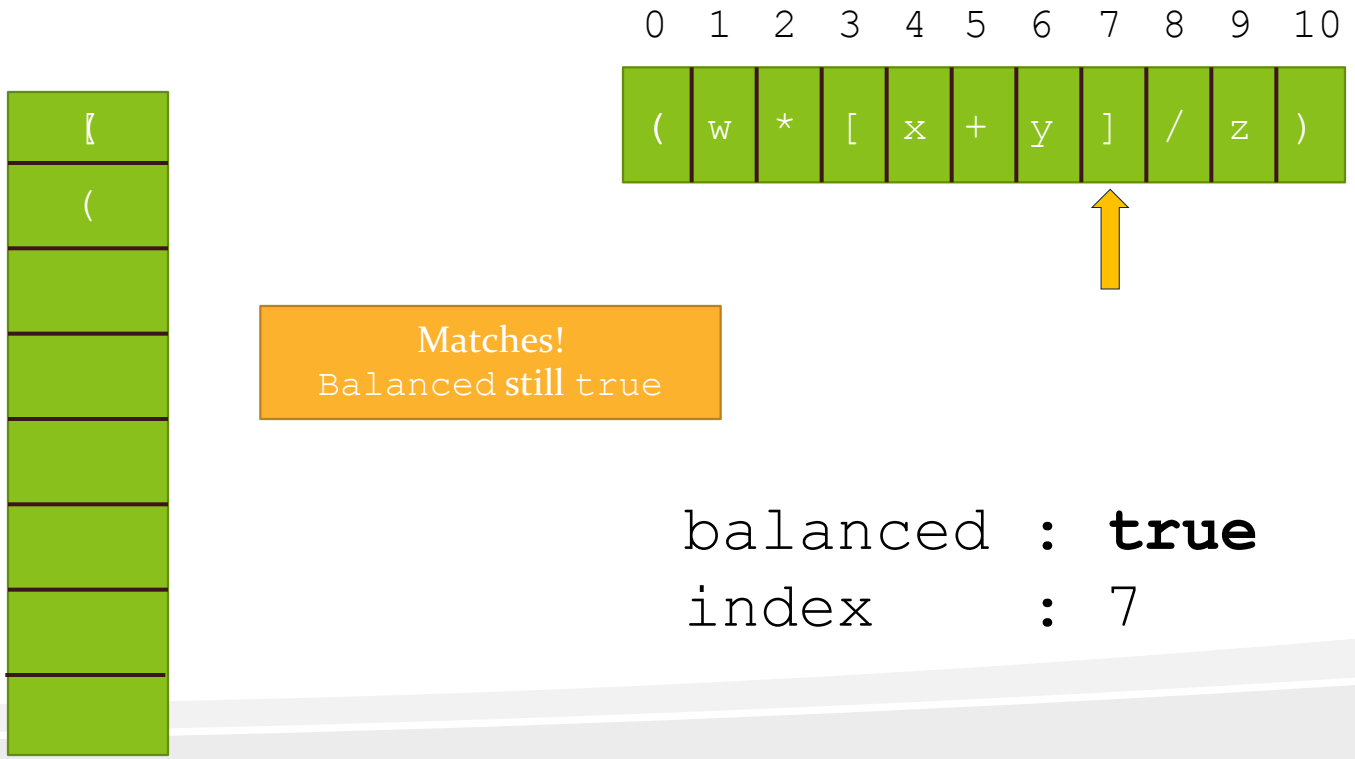
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 6
```

Balanced Parentheses (cont.)

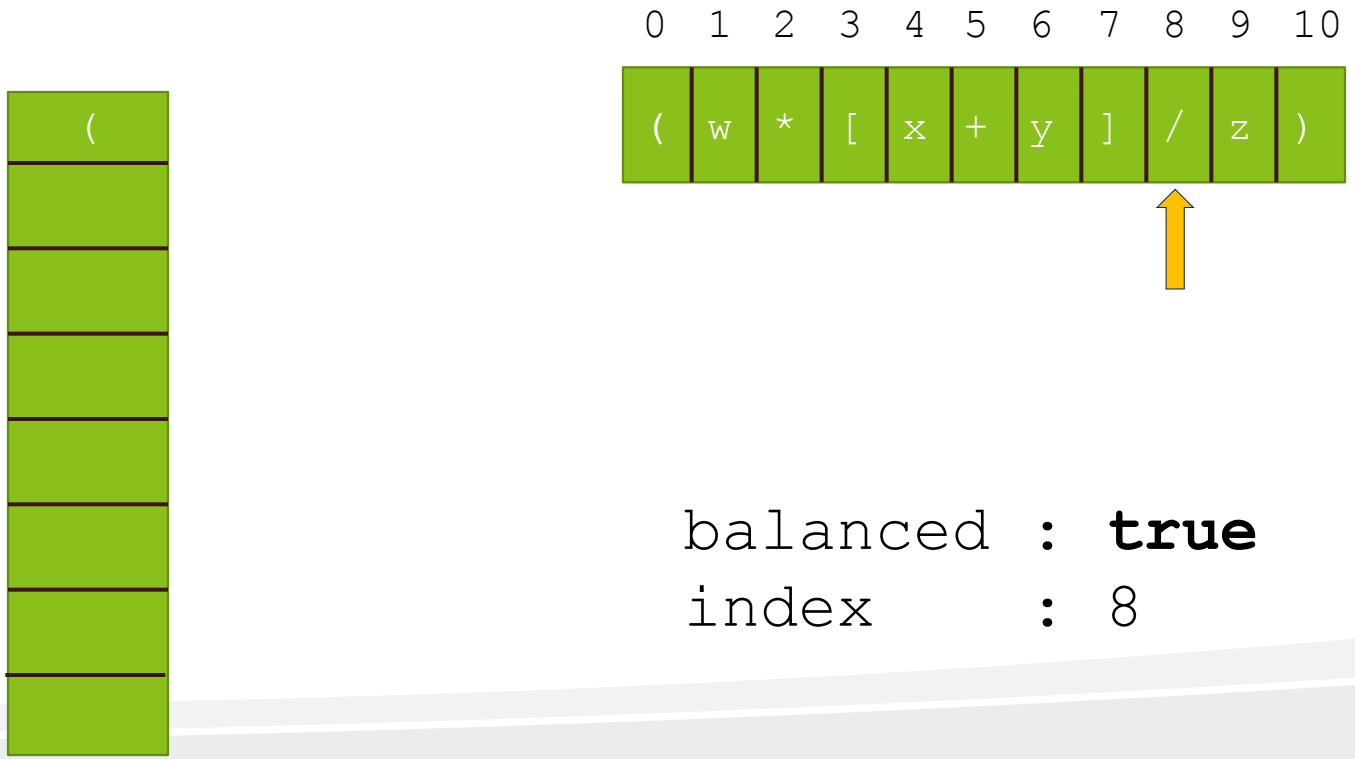
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 7
```

Balanced Parentheses (cont.)

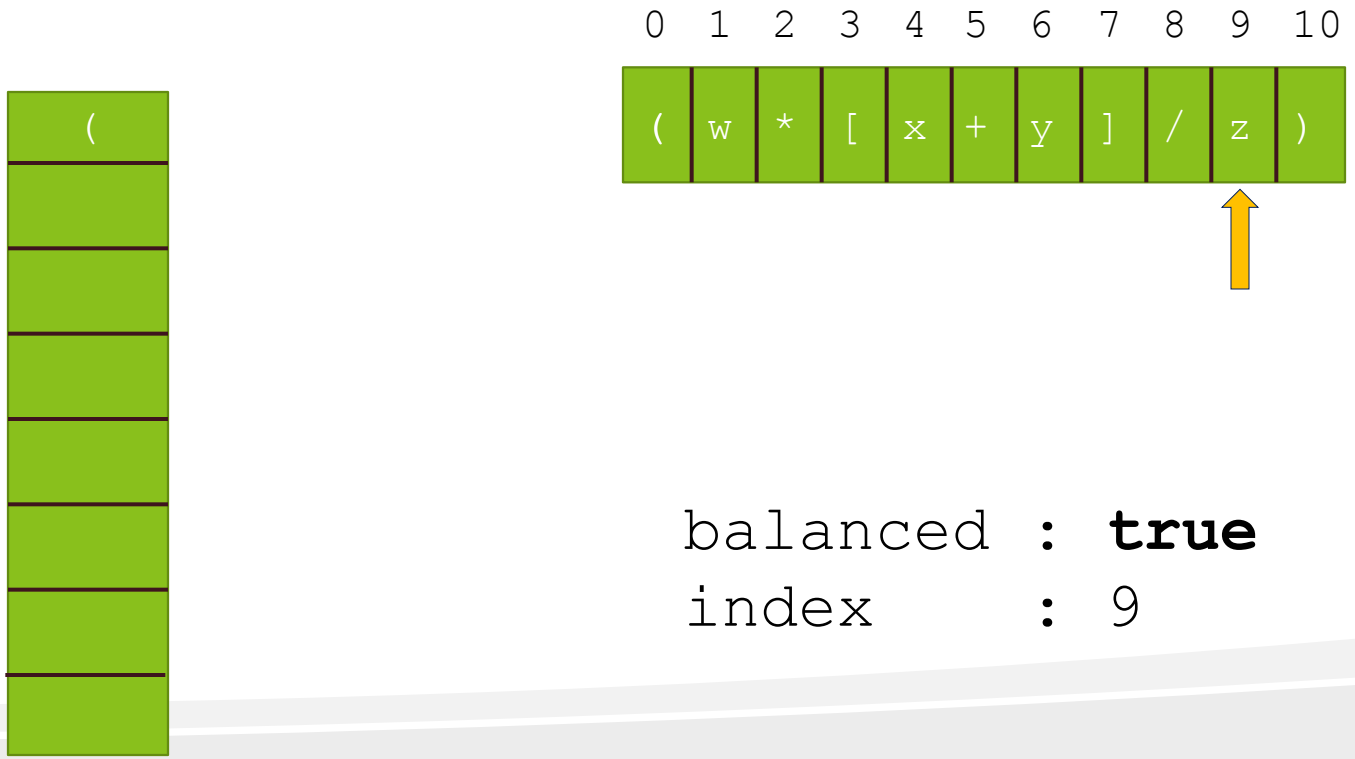
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 8
```

Balanced Parentheses (cont.)

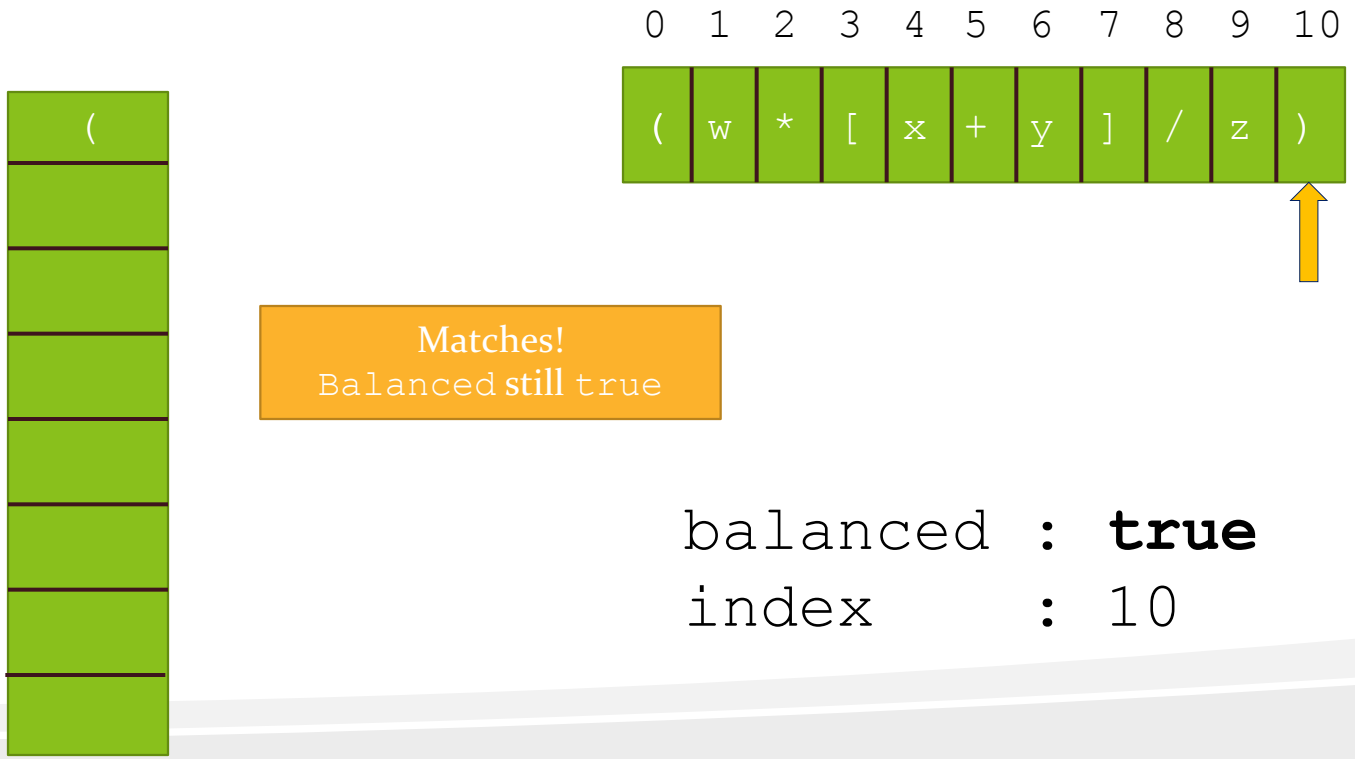
Expression: `(w * [x + y] / z)`




```
balanced : true
index    : 9
```

Balanced Parentheses (cont.)

Expression: `(w * [x + y] / z)`



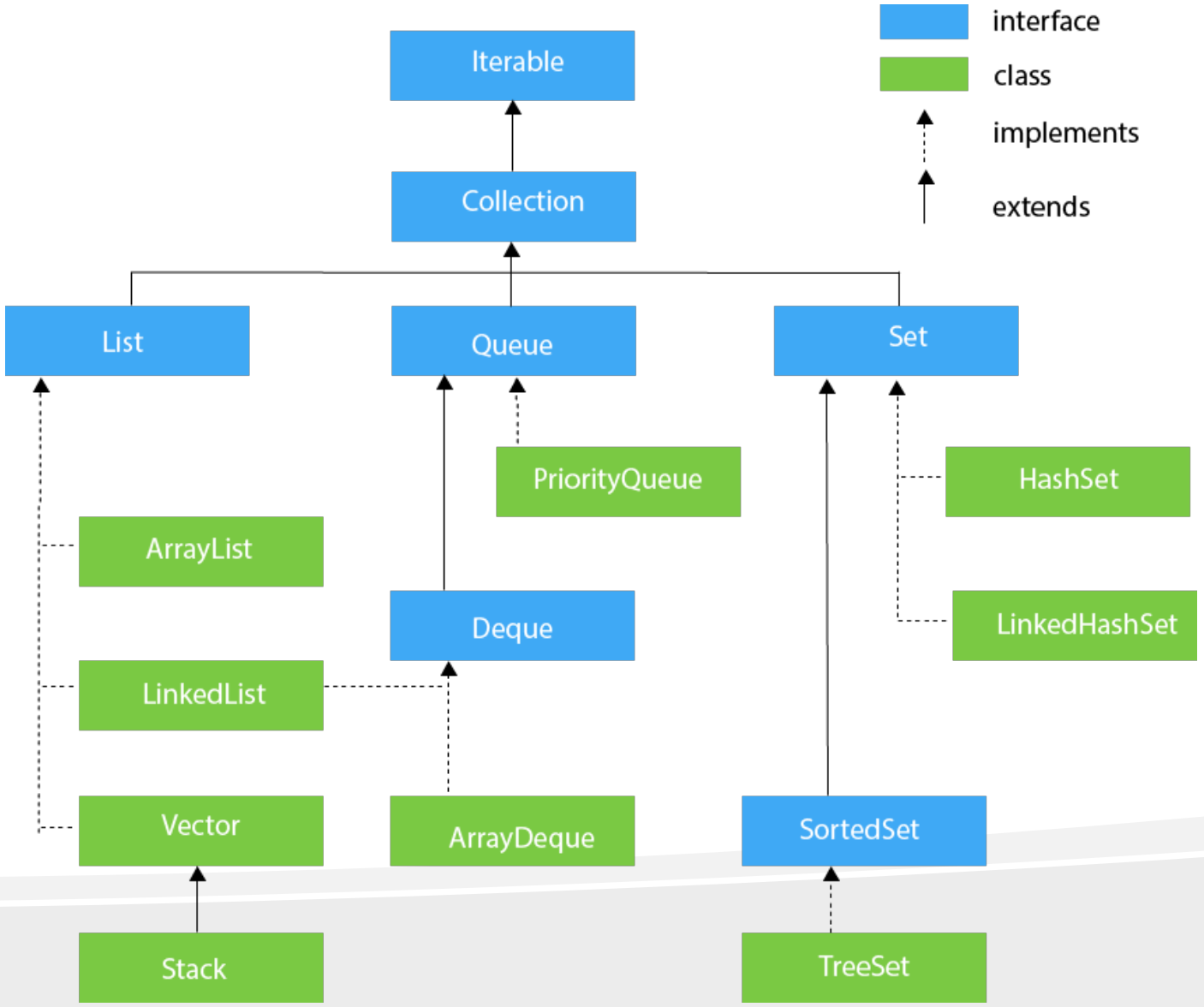

```
balanced : true
index    : 10
```

Testing

- Provide a variety of input expressions displaying the result `true` or `false`
- Try several levels of nested parentheses
- Try nested parentheses where corresponding parentheses are not of the same type
- Try unbalanced parentheses

- PITFALL: attempting to pop an empty stack will throw an `EmptyStackException`. You can guard against this by either testing for an empty stack or catching the exception

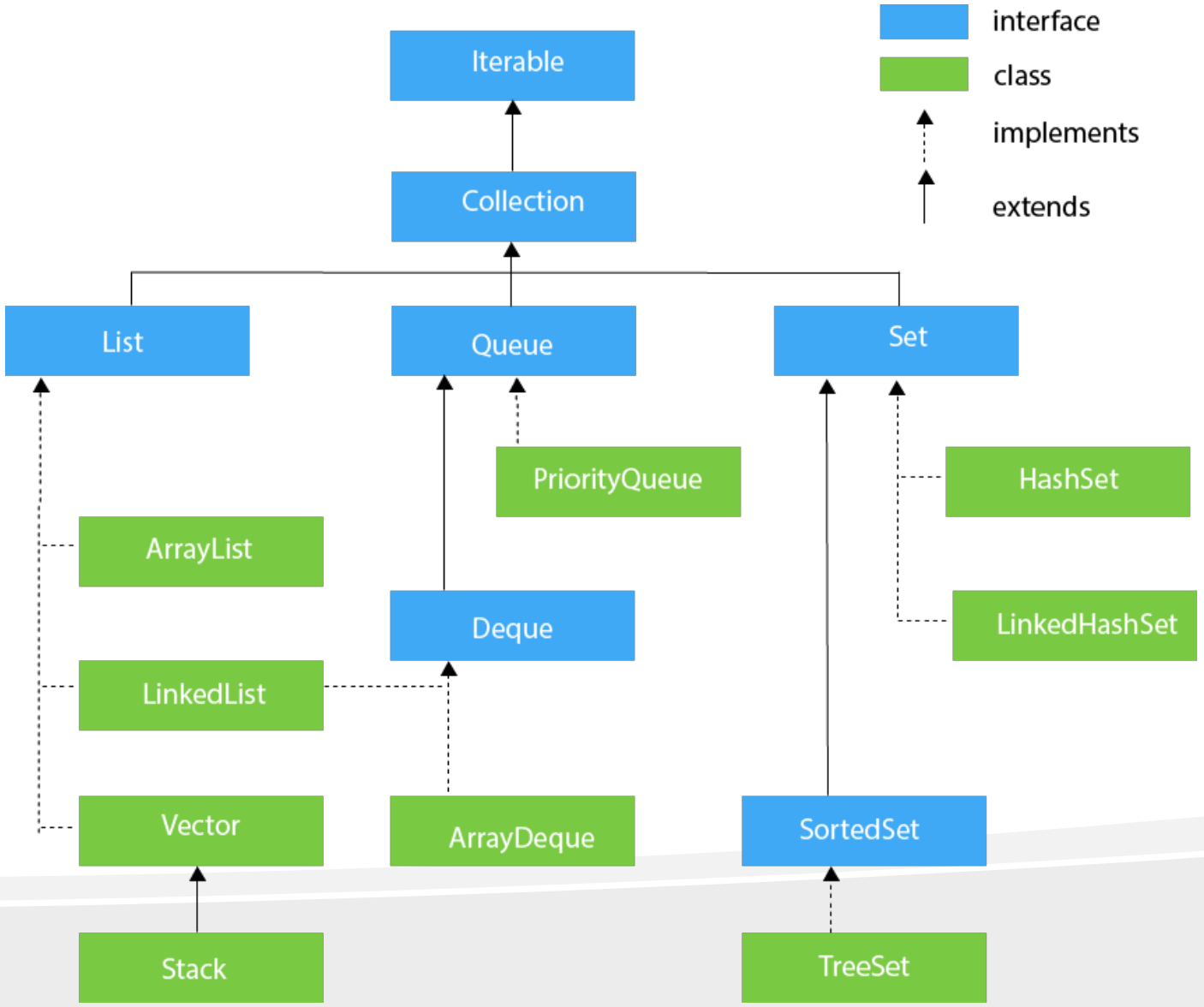
Collections Framework Diagram



Implementing a Stack as an Extension of Vector (cont.)

- Because a `Stack` *is a* `Vector`, all of `Vector` operations can be applied to a `Stack` (such as searches and access by index)
- But, since only the top element of a stack should be accessible, this violates the principle of information hiding

Review: Collections Framework Diagram



Sets

Words in a book

- Write an application that reads in the text of a book (say, Moby Dick) and then lets the user type words, and tells whether those words are contained in Moby Dick or not
  - How would we implement this with a List?

Sets

- **Set**: A collection of unique values (no duplicates allowed) that can perform the following operations efficiently:
  - `add`, `remove`, search (`contains`)
  - We don't think of a set as having indexes; we just add things to the set in general and don't worry about order