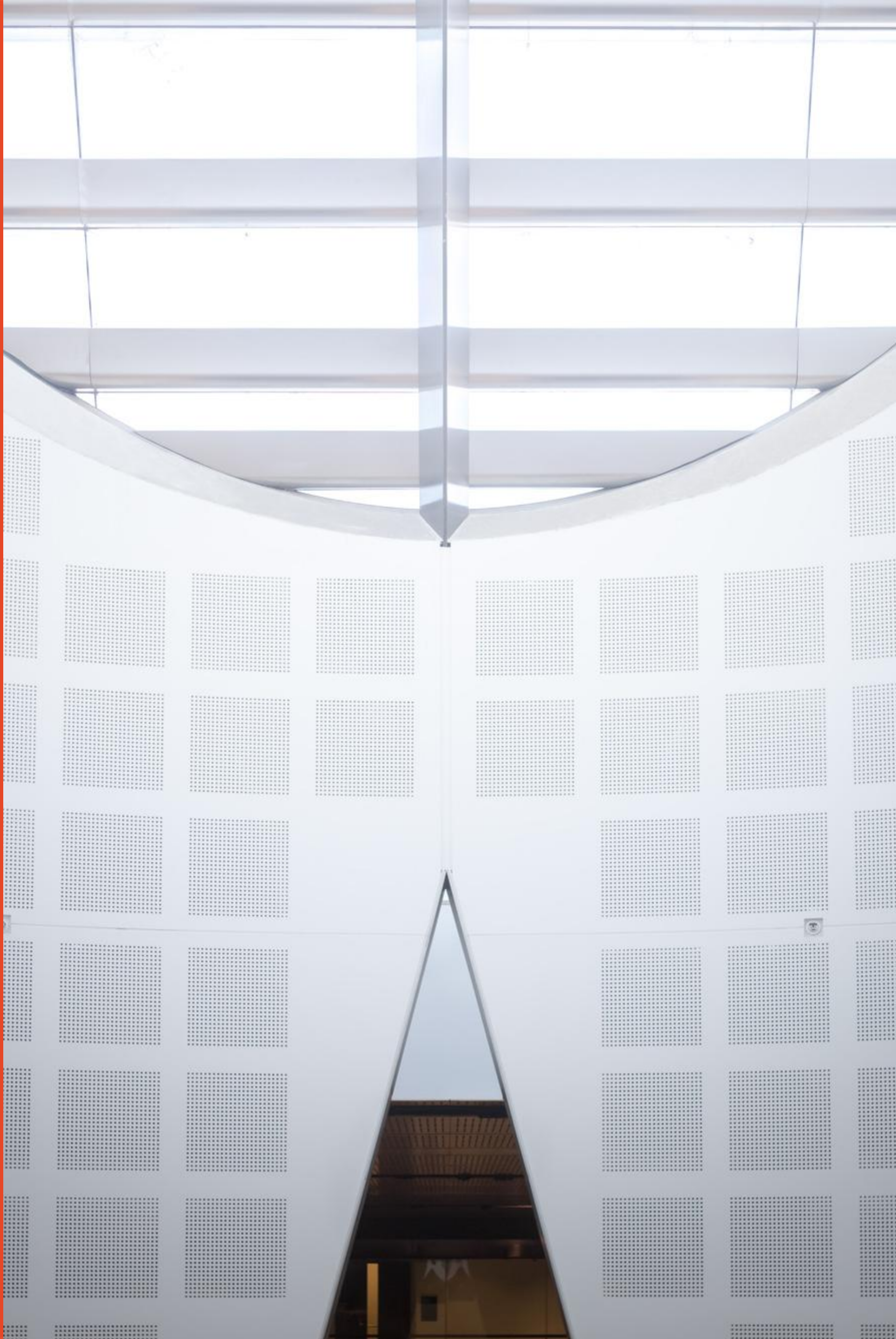
9123: Data Structures & Algorithms

Abstract Data Types and Algorithm Analysis

Dr. Ravihansa Rajapakse
Dr. Karlos Ishac
School of Computer Science

THE UNIVERSITY OF SYDNEY



Recap

- We discussed the structure of linked lists and operations
- List types
  - Singly linked lists
  - Doubly linked lists
  - Circular lists
- Operations
  - Insertion
  - Deletion
  - Traversal

Abstract Data Types

THE UNIVERSITY OF SYDNEY

Abstract Data Types & Data Structures

An abstract data type (ADT) is a specification of the desired behaviour from the point of view of the user of the data.

A data structure is a concrete representation of data, and this is from the point of view of an implementer, not a user.

Distinction is subtle but similar to the difference between a computational problems and an algorithm.

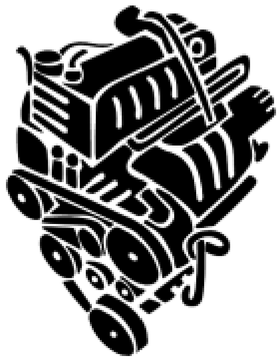
Abstract Data Types (ADT)

Type defined in terms of its data items and associated operations, not its implementation.

Simple example: Driving a car

interface

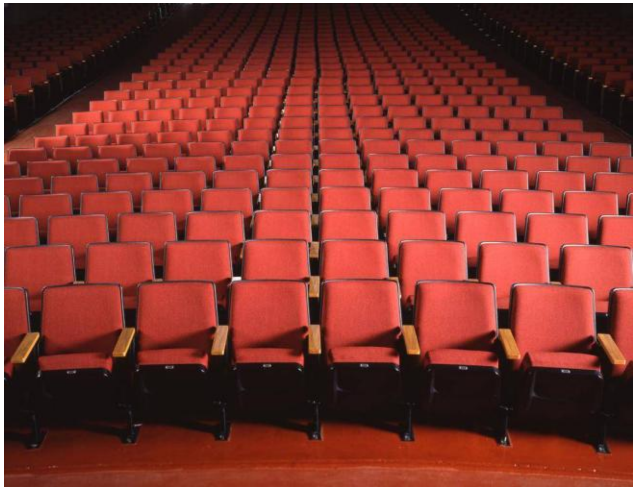
implementation



Benefits of ADT Approach

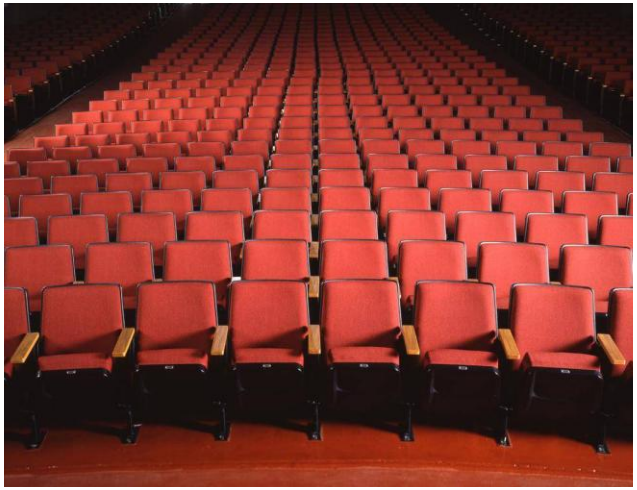
- Code is easier to understand if different issues are separated into different places.
- Client can be considered at a higher, more abstract, level.
- Many different systems can use the same library, so only code tricky manipulations once, rather than in every client system.
- There can be choices of implementations with different performance tradeoffs, and the client doesn't need to be rewritten extensively to change which implementation it uses.

Example: Reservation System



- We have a theatre with 500 named seats, e.g., “N31”
- What kind of data should be stored?
  - Seats names
  - Seats reserved or available.
  - If reserved, name of the person who reserved the seat.
- Operations needed?

Example: Reservation System



- Operations needed?
  - capacity_available() : number of available seats (integer)
  - capacity_sold() : number of seats with reservations
  - customer(x) : name of customer who bought seat x
  - release(x) : make seat x available (ticket returned)
  - reserve(x, y) : customer y buys ticket for seat x
  - add(x) : install new seat whose id is x
  - get_available(): access available seats

ADT Challenges

- Specify how to deal with the boundary cases
  - what to do if reserve(x, y) is invoked when x is already occupied?
  - what other cases can you think of?

- Do we need a new ADT? Could we use an existing one, perhaps by renaming the operations and tweaking the error-handling?

Stacks and Queues

These ADTs are restricted forms of List, where insertions and removals happen only in particular locations:

- stacks follow last-in-first-out (LIFO)
- queues follows first-in-first-out (FIFO)

So why should we care about a less general data structures?

- operations and names are part of computing culture
- numerous applications
- simpler/more efficient implementations than Lists

Stacks

THE UNIVERSITY OF SYDNEY

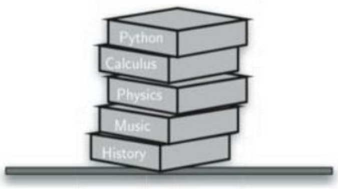
What is a Stack?

– A stack(sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end.

– This end is commonly referred to as the “top.”

– Stack principle: Last In First Out (LIFO) which means the last element inserted is the first one to be removed

– Example: Which is the first element to pick up?

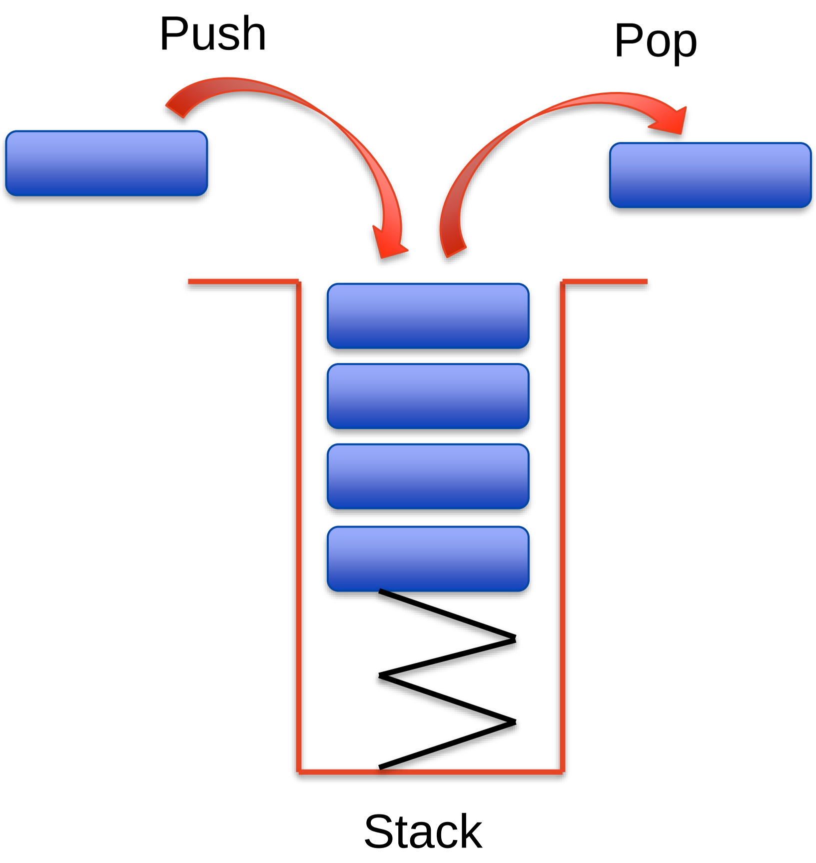


Stack Example

– A common model of a stack is a plate or coin stacker.

– Plates are "pushed" onto the top and "popped" off from the top

– Stacks form Last-In-First-Out (LIFO)



Stack Operations

Main stack operations:

- push(e): inserts an element, e
- pop(): removes and returns the last inserted element

Auxiliary stack operations:

- top(): returns the last inserted element without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Stack Operations Example

Operation	Returns	Stack
push(5)	-	[5]
push(3)	-	[5, 3]
size()	2	[5, 3]
pop()	3	[5]
isEmpty()	False	[5]
pop()	5	[]
isEmpty()	True	[]
push(7)	-	[7]
push(9)	-	[7, 9]
top()	9	[7, 9]
push(4)	-	[7, 9, 4]
pop()	4	[7, 9]

Applications of Stacks

Direct applications

- Keep track of a history that allows undoing such as Web browser history or undo sequence in a text editor
- Chain of method calls in a language supporting recursion
- Parentheses checker-examine a file to see if its braces {} and other operators are matching

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Method Stacks

The runtime environment keeps track of the chain of active methods with a stack, thus allowing recursion

When a method is called, the system pushes on the stack a frame containing

- Local variables and return value
- Program counter

When a method ends, we pop its frame and pass control to the method on top

```
def main()
    i = 5;
    foo(i);

def foo(j)
    k = j+1;
    bar(k);

def bar(m)
    ...
```

bar
PC = 1
m = 6

foo
PC = 2
j = 5
k = 6

main
PC = 2
i = 5

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!

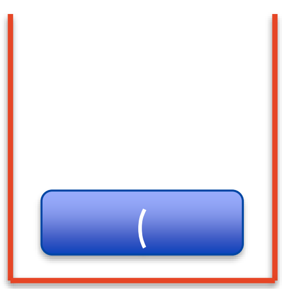
Steps to Check for Balanced Parentheses

- Initialize an empty stack.
- Iterate through each character in the expression.
  - If the character is an **opening bracket** ((, { , [ ), push it onto the stack
  - If the character is a **closing bracket** ( ) , } ,]):
    - Check if the stack is empty. If yes, return **false** (unbalanced).
    - Otherwise, **pop** the top element from the stack.
    - Check if the popped opening bracket **matches** the current closing bracket. If not, return **false** (unbalanced).
- After iteration, check the stack:
  - If the stack is **empty**, return **true** (balanced).
  - If not, return **false** (unbalanced).

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

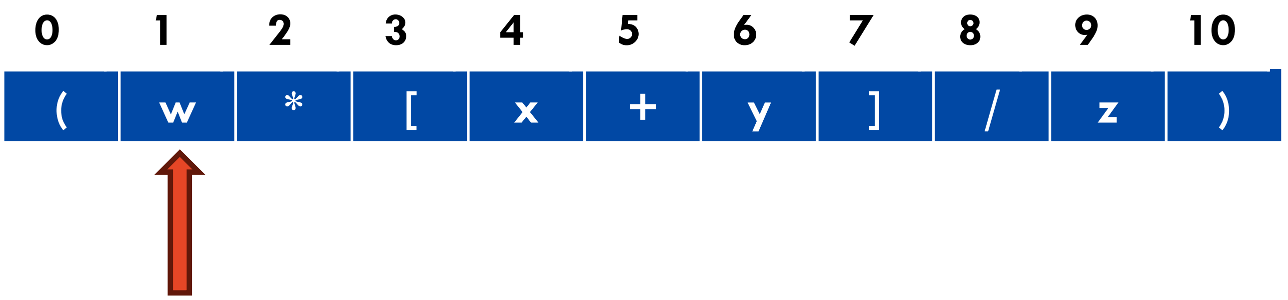


Stack

Balanced : true
Index : 0

Balanced Parentheses (Example 1)

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

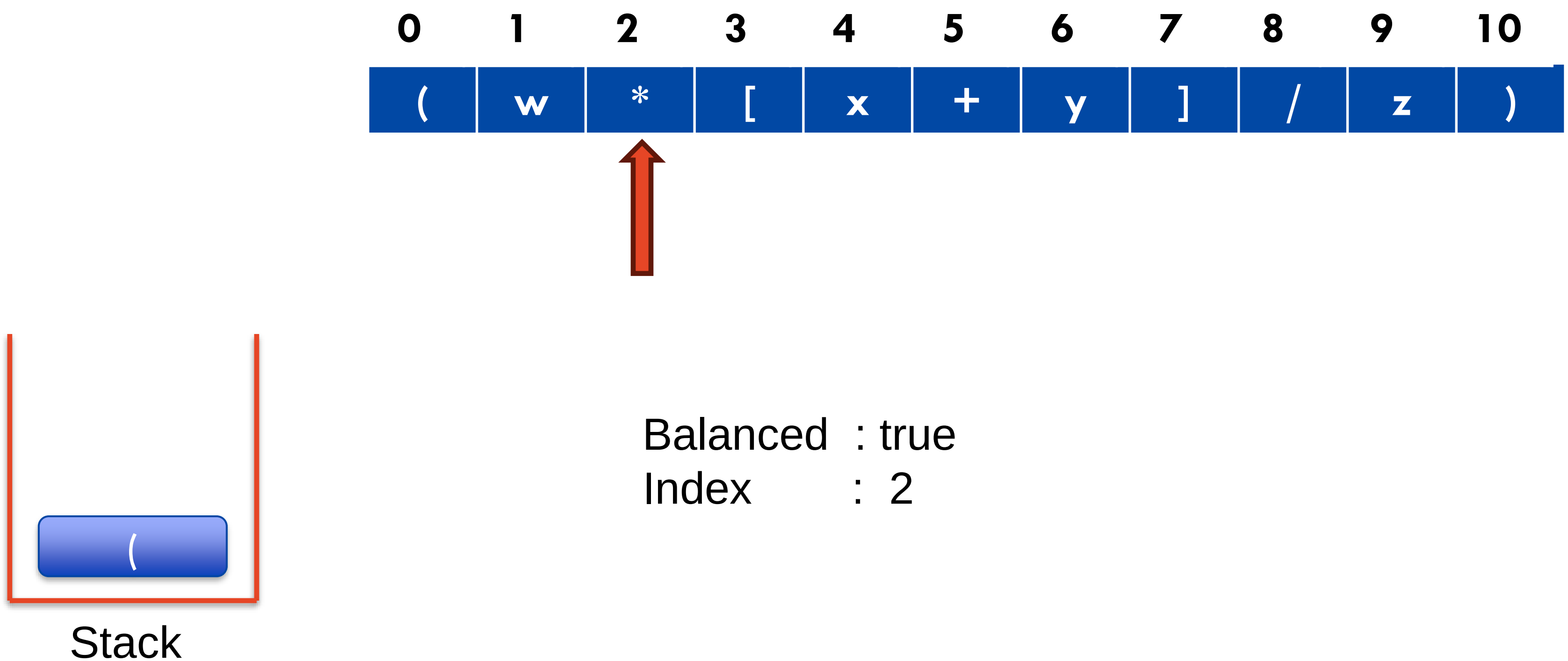


Stack

Balanced : true
Index : 1

Balanced Parentheses (Example 1)

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

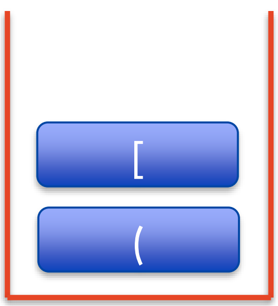


Balanced : true
Index : 2

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



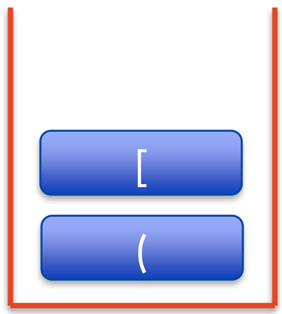
Stack

Balanced : true
Index : 3

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



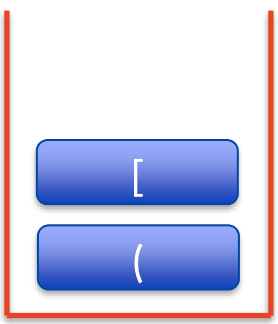
Stack

Balanced : true
Index : 4

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



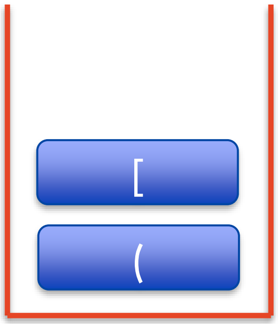
Stack

Balanced : true
Index : 5

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)

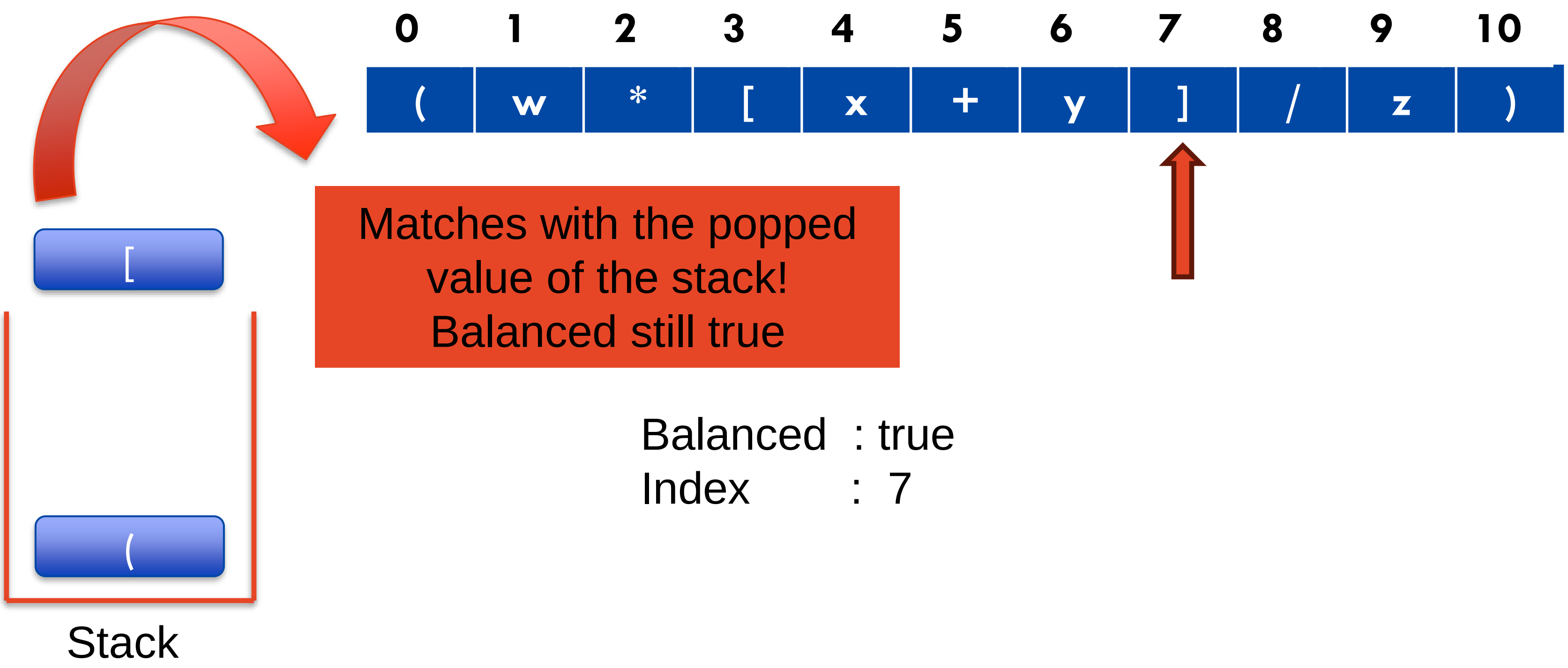


Stack

Balanced : true
Index : 6

Balanced Parentheses (Example 1)

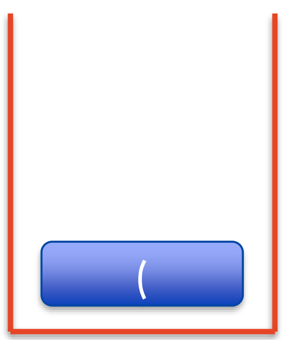
Expression: (w * [x + y] / z)



Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



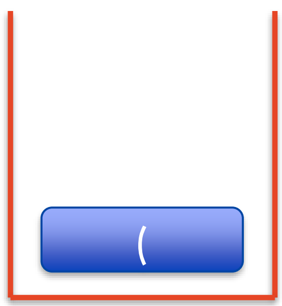
Stack

Balanced : true
Index : 8

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



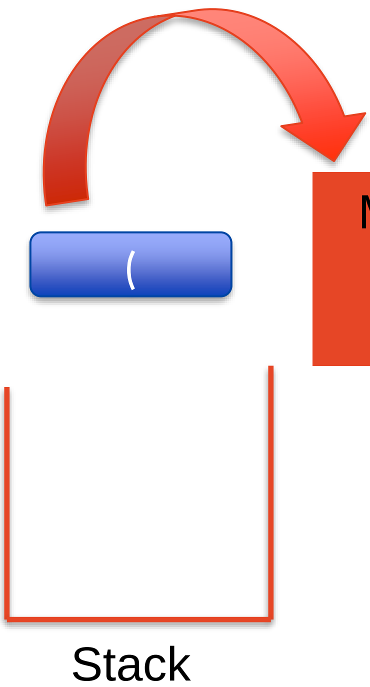
Stack

Balanced : true
Index : 9

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



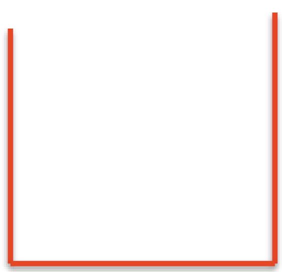
Matches with the popped value of the stack! Balanced still true

Balanced : true
Index : 10

Balanced Parentheses (Example 1)

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

0	1	2	3	4	5	6	7	8	9	10
(	w	*	[	x	+	y	]	/	z	)



Stack

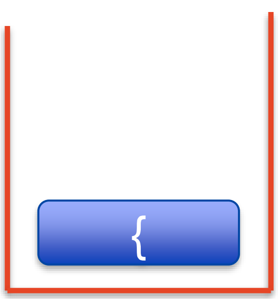
Balanced : true
Index : 10

Since the stack is empty at the end, the expression is **balanced**

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }

0	1	2	3	4	5	6	7	8	9
{	[	x	+	y	)	]	-	z	}

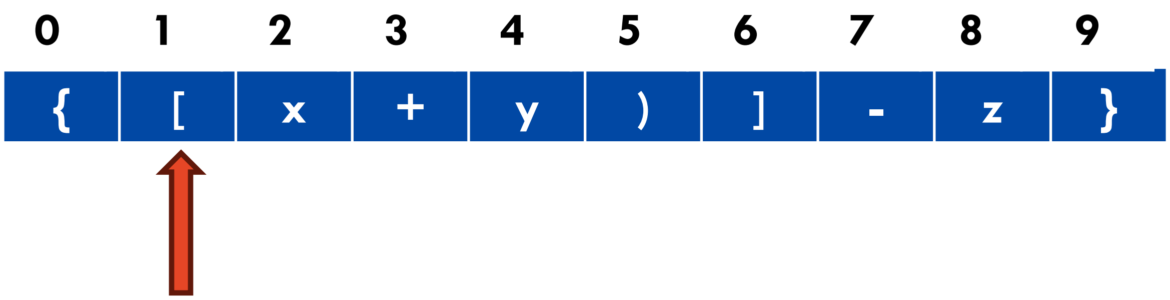


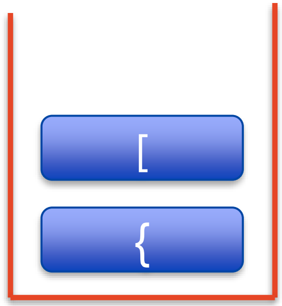
Stack

Balanced : true
Index : 0

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }



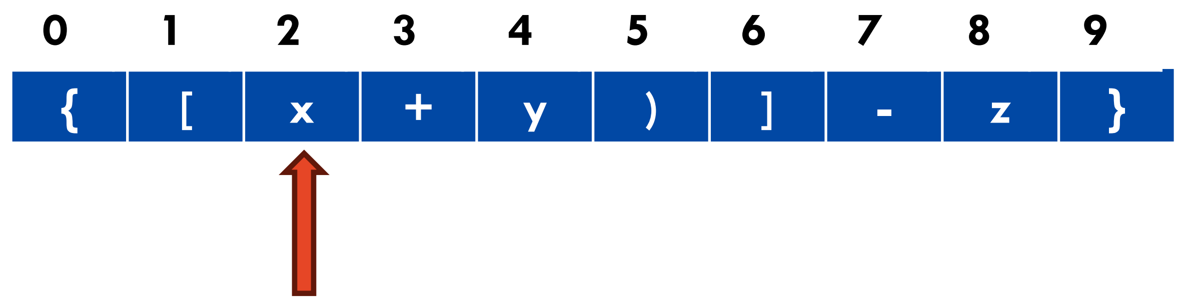


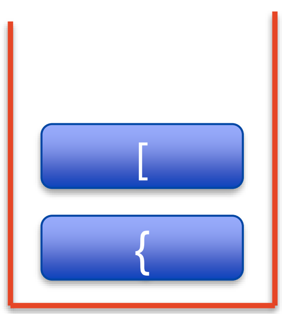
Stack

Balanced : true
Index : 1

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }





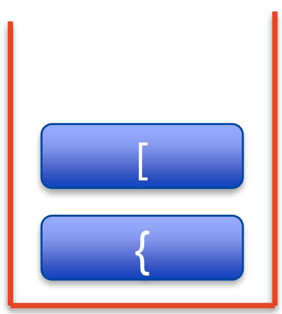
Stack

Balanced : true
Index : 2

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }

0	1	2	3	4	5	6	7	8	9
{	[	x	+	y	)	]	-	z	}



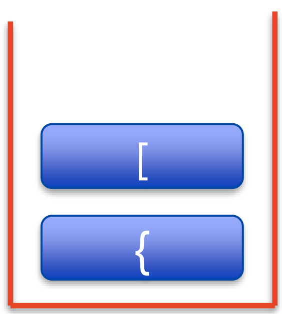
Stack

Balanced : true
Index : 3

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }

0	1	2	3	4	5	6	7	8	9
{	[	x	+	y	)	]	-	z	}

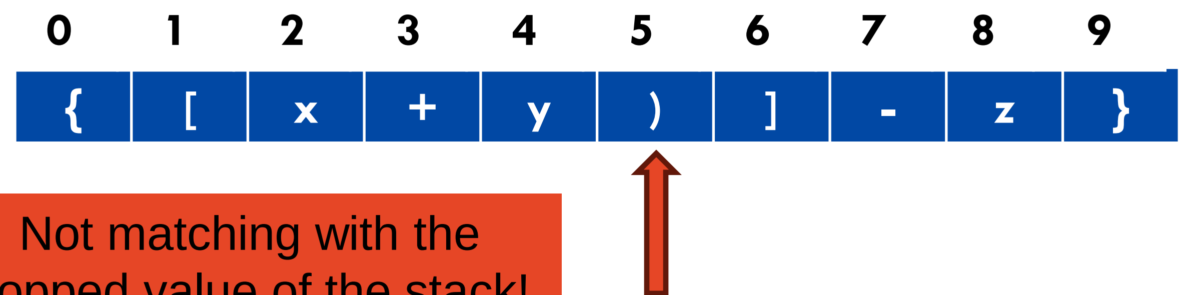


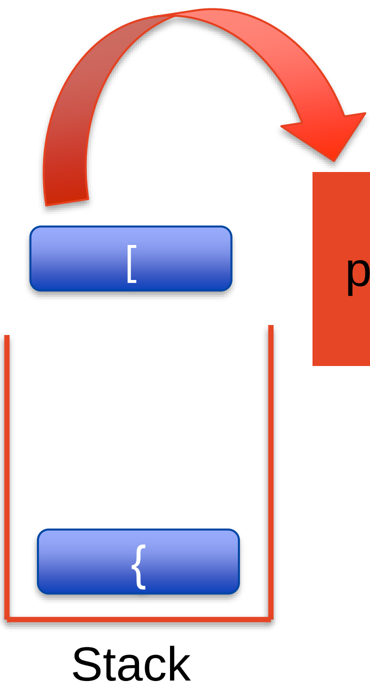
Stack

Balanced : true
Index : 4

Balanced Parentheses (Example 2)

Expression: { [x + y)] - z }





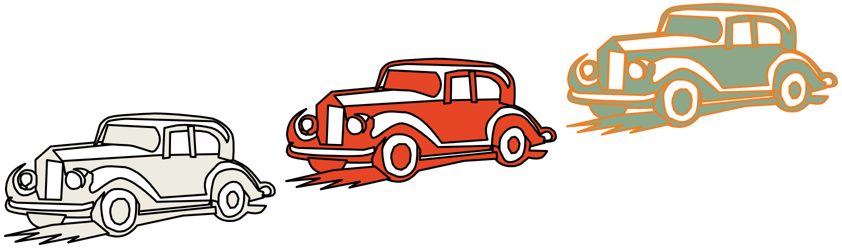
Not matching with the popped value of the stack!
Balanced is false

Balanced : false
Index : 5

Since the Balanced is false, the expression is **unbalanced**

Queues

Queue ADT



Main queue operations:

- enqueue(e): inserts an element, e, at the end of the queue
- dequeue(): removes and returns element at the front of the queue

Auxiliary queue operations:

- first(): returns the element at the front without removing it
- size(): returns the number of elements stored
- isEmpty(): indicates whether no elements are stored

Queue Example

Operation	Output	Queue
enqueue(5)	-	(5)
enqueue(3)	-	(5, 3)
dequeue()	5	(3)
enqueue(7)	-	(3, 7)
dequeue()	3	(7)
first()	7	(7)
dequeue()	7	()
isEmpty()	true	()
enqueue(9)	-	(9)
enqueue(7)	-	(9, 7)
size()	2	(9, 7)
enqueue(3)	-	(9, 7, 3)
enqueue(5)	-	(9, 7, 3, 5)
dequeue()	9	(7, 3, 5)

Queue applications

Buffering packets in streams, e.g., video or audio

Direct applications

- Waiting lists
- Access to shared resources (e.g., printer)
- Multiprogramming

Indirect applications

- Auxiliary data structure for algorithms
- Component of other data structures

Queue Application: Ticket Counter

Imagine a queue at a ticket counter where people stand in line to buy tickets. The first person in line gets served first, following the **FIFO (First in, First Out)** principle.

Operations in the queue:

1. Enqueue (Add to queue) : A person joins the end of the queue
2. Dequeue (Remove from queue) : The person at the front gets served and leaves the queue

Algorithm Analysis

THE UNIVERSITY OF
SYDNEY

Why Do We Need a Performance Measure for Algorithms?

- **Multiple Ways to Solve a Problem** – There are many different algorithms for the same task, and we need a way to evaluate them.
- **Finding the Best Approach** – Algorithm analysis helps compare different solutions to determine which one is the most efficient.
- **Estimating Resource Usage** – It allows us to predict how much time and memory an algorithm will require as input size grows.
- **Ensuring Scalability & Performance** – Helps choose algorithms that work well for both small and large datasets without unnecessary slowdowns.

Early Attempts to Measure Algorithm Efficiency

Measuring Execution Time

- Initially, execution time was used to determine how efficient an algorithm was.

- Developers timed how long an algorithm took to complete a task and compared results.

- While straightforward, this method had significant drawbacks for reliable evaluation.

Why Execution Time is Not a Good Measure?

– **Hardware Dependent –** Performance varies across different CPUs, RAM, and system configurations.
– **Implementation Dependent –** Execution time is affected by programming language, compiler, and system optimizations.
– **Input Size Variation –** Small inputs may run fast, but execution time doesn't predict behavior for large datasets.
– **External Factors –** Background processes and multi-threading can cause inconsistent results.

The Need for a Better Measure

Why do we need a new way to analyze algorithms?

- We need a method that is independent of hardware and implementation.
- It should focus on how an algorithm scales with input size.
- It must allow us to compare different algorithms objectively.

Introduction to Big-O Notation

What is Big-O Notation?

– A mathematical way to describe how the runtime of an algorithm grows with input size (n).

– Focuses on the worst-case scenario to ensure performance reliability.

– Ignores constant factors and lower-order terms. (e.g., $O(2n+4n^2) \rightarrow O(n^2)$)

Why is it useful?

– Provides a standardized method for analyzing efficiency.

– Helps choose the best algorithm for large-scale problems.

Understanding Growth Factor in Big-O

- The **growth factor** in time complexity refers to how the runtime of an algorithm increases as the input size (**n**) grows.
- Example:

$$T_{(n)} = \mathrm{n}c_1 + n^2c_2 + n^3c_3 + n^4c_4$$

n=1

$$T_1 = c_1 + c_2 + c_3 + c_4$$

n=10

$$T_{10} = 10c_1 + 100c_2 + 1000c_3 + 10000c_4$$

n=100

$$T_{100} = 100c_1 + 10000c_2 + 1000000c_3 + 100000000c_4$$

Big-O = O(n^4)

Why Worst-Case Matters?

- **Predictability & Reliability** – Ensures the algorithm performs within known limits, crucial for real-time and critical applications.
- **Avoids Unexpected Slowdowns** – Some algorithms perform well on average but degrade in the worst case (e.g., QuickSort: $O(n \log n)$ avg, $O(n^2)$ worst).
- **Helps Choose the Right Algorithm** – Algorithms like Merge Sort ($O(n \log n)$) are preferred over Bubble Sort ($O(n^2)$) due to consistent worst-case performance.
- **Optimizes Resource Allocation** – Knowing the worst-case complexity helps developers allocate the right amount of computational power, memory, and bandwidth, preventing system crashes and inefficiencies.

Big-O Notation

- Instead of exact times or operations, **Big-O describes growth trends.**
- It tells us the **upper bound (worst-case scenario)** of an algorithm's efficiency.

```python
def print_items(n):
    for i in range(n):
        print(i)  # Runs n times
```

O(n) → Linear time complexity

- This gives a **hardware-independent** way to compare algorithms.

Scanning Items at a Supermarket (O(n)) – Linear Time

Scenario:

– A cashier scans items at checkout, and you have 50 items in your cart.

Approach:

– Each item is scanned one by one into the system. The total time taken grows directly with the number of items.

– If you double the items (100 items), it takes twice as long.

Complexity Analysis:

– The time required increases proportionally with the number of items. Works fine for moderate inputs, but scales linearly.

– Big-O Complexity: O(n) (Good, but not ideal for very large inputs).

Finding a Word in a Physical Dictionary (O(log n)) – Logarithmic Time

Scenario:

- You are looking for the word "Algorithm" in a 1,000-page dictionary.

Approach:

- You don't flip through each page one by one.
- Instead, you open the middle and check:
  - If the word comes before, search the left half.
  - If the word comes after, search the right half.
- You repeat this process until you find the word.

Finding a Word in a Physical Dictionary (O(log n)) – Logarithmic Time

Complexity Analysis:

– Each time, the search space halves (1,000 → 500 → 250 → 125 → ...).
– The number of searches needed grows logarithmically with the number of pages.
– Efficiency: Even with a million pages, you'd only need about 20 searches!
– Big-O Complexity: O(log n) (Very efficient for large datasets).

Checking for Duplicate Transactions in a Bank ($O(n^2)$) – Quadratic Time

Scenario:

– A bank needs to check for duplicate transactions in a list of 1,000 payments.

Approach:

– The system compares each transaction with every other transaction to see if they match.
– This requires nested loops:
  – The first loop picks a transaction.
  – The second loop checks all other transactions for a duplicate.

Checking for Duplicate Transactions in a Bank ($O(n^2)$) – Quadratic Time

Complexity Analysis:

- If there are 1,000 transactions, the system performs 1,000 × 1,000 = 1,000,000 comparisons.
- If transactions double to 2,000, comparisons become 4,000,000—this scales poorly!
- Big-O Complexity: $O(n^2)$ (Becomes too slow for large datasets).

Understanding Time & Space Complexity

Time Complexity

- Measures how execution time grows as input size (n) increases.
- Helps analyze algorithm efficiency.
- Example: Searching a name in an unsorted list $O(n)$ vs. binary search in a phonebook $O(\log n)$.

Space Complexity

- Measures how much memory an algorithm needs as input size grows.
- Includes variables, recursion, and extra data structures.
- Example: Sorting an array in place $O(1)$ vs. using extra memory for a copy $O(n)$.

Common Big-O Examples with Real-World Analogies

Complexity	Example	Real-World Analogy
O(1) (Constant)	Accessing arr[i]	Finding a book by its shelf number
O(log n) (Logarithmic)	Binary search	Looking for a word in a dictionary
O(n) (Linear)	Looping through an array	Checking every page in a book
O(n log n) (Linearithmic)	Merge Sort	Efficiently organizing pizza orders
$O(n^2)$ (Quadratic)	Bubble Sort	Pairwise comparisons in a tournament
$O(2^n)$ (Exponential)	Recursive Fibonacci	Brute-force password cracking

Big-O Complexity Growth Rates