COMPSCI 107
Computer Science Fundamentals

Lecture 12 – ADTs and Stacks

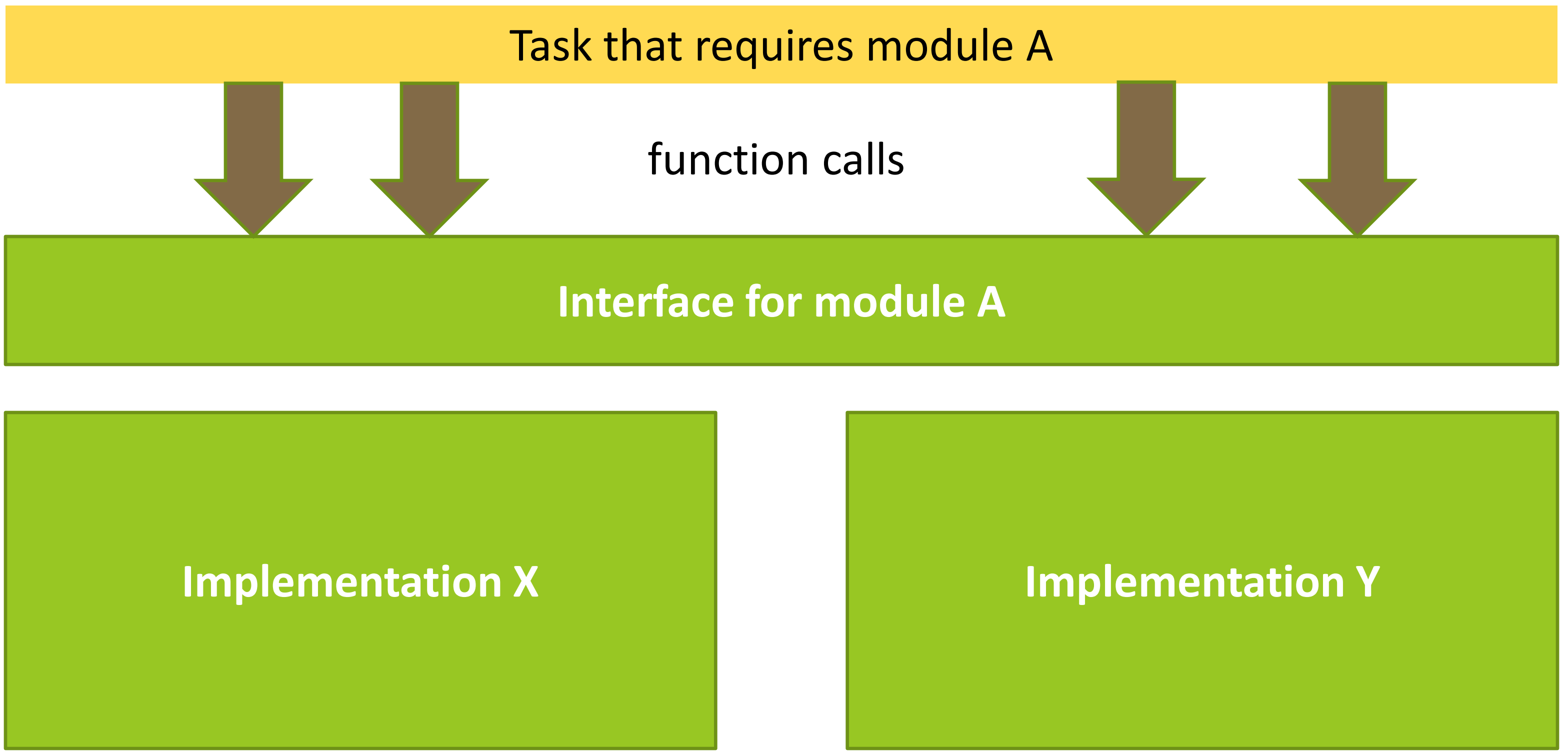
Software Engineering Design Principle

- Modularity
  - Divide the program into smaller parts

- Advantages
  - Keeps the complexity managable
  - Isolates errors (parts can be tested independently)
  - Can replace parts easily
  - Eliminates redundancies (each part takes care of own data)

- Easier to write, Easier to read, Easier to test, easier to modify

Hide implementation within the module



Abstraction

- Separates the purpose from the implementation
- Procedural abstraction
  - Specification (function header and return type) separate from implementation
  - Can replace implementation if required
- Data Abstraction
  - Think about what can be done with the data, separate from how it is done
- Example:
  - Mapping of keys to values
  - Use two parallel lists
  - Use a dictionary

Abstract Data Type (ADT)

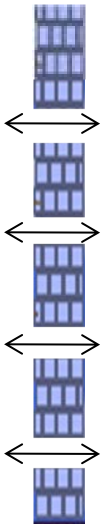
- A collection of data and a set of operations on that data
  - Specifications of an ADT focus on what the operations are
  - Implementation is not specified
- ADT is not a data structure
  - Data structure is a construct within a programming language
  - List, tuple, dictionary

ADT

ADT

- An interface which is visible to the user of the ADT (the client)

ADT



Data structure

A data structure used to implement the ADT

A wall of ADT operations isolates the data structure and the implementation from the program that uses it.

ADT - Integers

- Data
  - [..., -3, -2, -1, 0, 1, 2, 3, ...]

- Operations
  - Addition
  - Subtraction
  - ...
  - Equality
  - Ordering
  - Representation for printing

ADT - Set

- Data
  - An unordered collection of unique elements
- Operations
  - Add
  - Remove
  - Union
  - Intersection
  - Complement

Linear Structure

- Data collection
  - Position of the elements matters and is stable
  - Two ends to the structure (front and back, first and last)

- Different structures has different ways to add and remove elements

ADT - Stack

- Ordered collection of data
  - Addition of items and removal of items happens at the same end
  - Top of the stack
- Remove data in reverse order of data added
  - Last in first out (LIFO)
- Operations
  - Push
  - Pop
  - Peek
  - Is_empty
  - Size

Stack Implementation

- Implementation using Python list
- What is the big-O of push()?
- What is the big-O of pop()?

```
class StackV1:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0,item)

    def pop(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

Stack Implementation

- Implementation using Python list
- What is the big-O of push()?
- What is the big-O of pop()?

```
class StackV2:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Checking braces

- Many computer languages use braces to signify start and end
  - They need to be matched to be correct
  - They need to be nested correctly

- Stacks can be used to determine correct use of braces

- Algorithm:
  - Add each open brace to the stack
  - When a closing brace is encountered, check to see if a matching brace is on the top of the stack
  - When the last token is checked, the stack should be empty

Exercise

- Use a stack to check if the braces are used correctly in the following strings. Show the state of the stack after each stack operation.

- (( ))
- (( ) (
- [( ) { }]
- [[( )] [{ }])

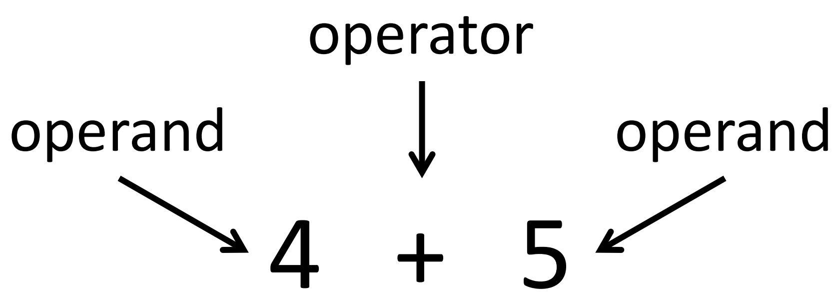
Exercise

- To be done after class: Write a program that uses a Stack to check if an input string has correctly matching braces

- check_braces('(this) [is] {best} ([done] {with} (stacks))')

Postfix expressions

- Standard mathematics represents expressions using infix notation
  - Operators appear between the operands



- Postfix notation puts the operator after the operands
  - No brackets are needed to specify order of precedence

4 5 +

Exercise

- Convert the following infix expressions into postfix notation
- 4 * 5 – 2 * 8 – 1
- 1 – 4 + 2 * 3 * 5
- 9 – 6 / 4 + 2 * (2 + 3)
- 2 * ((4 + 2) * 3 + 2) – 1

Converting from infix to postfix

- A stack can be used in the algorithm to convert infix to postfix
  - Divide expression into tokens
  - Operators: +. -, *, /
  - Operands: single digits
  - Other tokens: brackets

Algorithm for converting infix to postfix

- Create a stack to store operators and a list for the output tokens
- Scan the tokens from left to right
- If the token is an operand, add it to the output list
- If the token is a left parenthesis, push it to the operator stack
- If the token is a right parenthesis, pop the operator stack until the left parenthesis is removed. Append each operator to the output list
- If the token is an operator, push it onto the operator stack. But first, remove any operators that have higher or equal precedence and append them to the output list
- When there are no more tokens, remove operators on the stack and append to the output list

Exercise

- Show the operator stack and the output list at every step as the following infix expression is converted to postfix

12 / (3 + 4) * 2 + 4

Evaluating postfix expressions

- Create an empty stack
- Scan the list of tokens from left to right
- If the token is an operand, push it to the operand stack
- If the token is an operator, pop the stack twice
  - The first element popped is the right operand
  - The second element popped is the left operand
- Apply the operator to the operands and push the result onto the stack
- When there are no more tokens, the stack should contain the result.

Exercise

- Following the algorithm to evaluate postfix expressions, show the operand stack, and the token being processed (at each step) as the following postfix expression is evaluated:

7 12 8 9 - * 3 / +