Stacks

Professor Frank Kreimendahl

School of Computing and Data Science

Wentworth Institute of Technology

September 26, 2022

Stack ADT

Stack ADT

- A stack is a fundamental data structure in computer science
- A stack behaves similarly to a Pez dispenser:
  - Only the top item can be accessed
  - Only one item can be inserted or extracted at a time
- The top of the stack is the most recently added item in the stack
- The stack is a *Last-in, First-out* (LIFO) data structure

Stack Specification

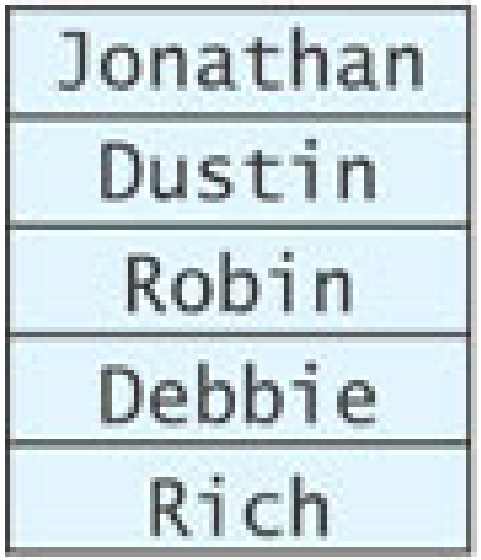
- We can only interact with the top of the stack (no random access), so there are not many operations possible
- Stack operations:
  - `empty()`: determine if a stack is empty
  - `peek()`: get the top item on the stack
  - `pop()`: remove and return the top item on the stack
  - `push(E)`: put a new item on top of the stack and return that item

Stack Interface

```
public interface StackInt<E> {
  E push(E obj);
  E peek();
  E pop();
  boolean isEmpty();
}
```

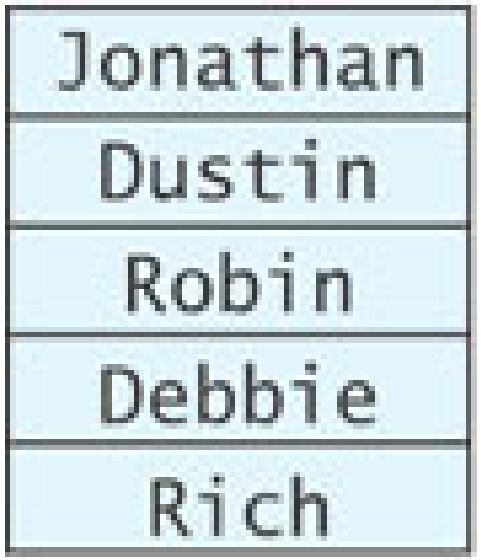
Example Peek

- `Rich` was added longest ago, and `Jonathan` most recently
- `Jonathan` is at the *top* of the stack, which is where every interaction takes place
- `String last = names.peek();` would result in `last` referencing the "Jonathan" string



Stack of Strings

Example Pop



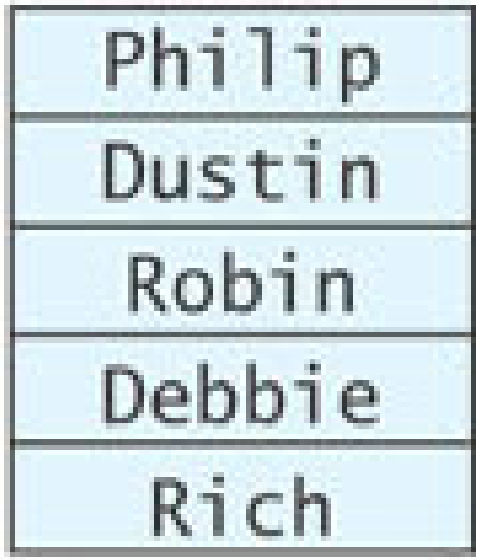
Before `pop()`

After `pop()`

- `String temp = names.pop();` modifies the stack and results in `temp` referencing the "Jonathan" string

Example Push

Before `push("Philip")`



After `push("Philip")`

- `names.push("Philip");` modifies the stack

Array Implementation

Array Implementation

- We can use an array to store the data held in a stack
- Treat the end of the list as the head of the stack – it is the most efficient to modify
- What list operation is similar to push?
- What list operation is similar to pop?
- We will actually use an `ArrayList` to keep the data, since it has operations we can translate

Class Definition

```
public class ListStack<E> implements StackInt<E> {
  private List<E> theData;

  public ListStack() {
    theData = new ArrayList<>();
  }

  // stack interface implementations
}
```

push and pop

```
public E push(E obj) {
  theData.add(obj);
  return obj;
}

public E pop() {
  if (isEmpty())
    throw new NoSuchElementException();
  return theData.remove(theData.size() - 1);
}
```

peek and isEmpty

```
public E peek() {
  if (isEmpty())
    throw new NoSuchElementException();
  return theData.get(theData.size() - 1);
}

public boolean isEmpty() {
  return theData.isEmpty();
}
```

Stack Applications

Checking for Balanced Parentheses

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

- Computers are good at reading and solving arithmetic expressions
- We need to describe to a program what expressions are valid or invalid
- Balanced parentheses are important in an expression being valid – we can validate an expression's parenthesis use
- We can use stacks to verify if an expression is valid or not

isBalanced Algorithm

ISBALANCED(expression)

```
 1: s ← empty stack
 2: index ← 0
 3: while index < expression.len() do
 4:     if next character == '(' then
 5:         s.push(next character)
 6:     else if next character == ')' then
 7:         if s.isEmpty() then
 8:             return false
 9:         s.pop()
10:     increment index
11: return s.isEmpty()
```

isBalanced Testing

- Test a variety of both valid and invalid inputs
- Test both nested and sequential parentheses
- Test unbalanced parentheses
- Test no parentheses

Infix To Postfix Converter

- Infix notation is standard expression notation – binary operators are between their operands
- Postfix is easier for a computer to process
- Postfix has another useful property: no need for parentheses!
- We will use a stack to convert from infix to postfix

Postfix Expression	Infix Expression	Value
4 7 *	4 * 7	28
4 7 2 + *	4 * (7 + 2)	36
4 7 * 20 -	(4 * 7) - 20	8
3 4 7 * 2 / +	3 + ((4 * 7) / 2)	17

Conversion Example

- Given the infix equation $w - 5.1/sum * 2$, convert to postfix
- What is the postfix form?
- How can we build it in an automated way?
- How can we build it so that we only have to scan through the infix equation once?

convert pseudocode

CONVERT(expression)

```
 1: postfix ← empty StringBuilder
 2: operator stack ← empty Stack
 3: while expression has tokens left do
 4:     t ← next token
 5:     if t is an operand then
 6:         append t to postfix
 7:     else if t is an operator then
 8:         PROCESSOPERATOR(t)
 9:     else Syntax error
10: pop all operators off stack and append to postfix
11: return postfix
```

processOperator Pseudocode

PROCESSOPERATOR(t)

```
 1: if operator stack is empty then
 2:     push t onto operator stack
 3: else
 4:     topOp ← top of operator stack
 5:     if t precedence > topOp precedence then
 6:         push t onto operator stack
 7:     else
 8:         while stack is not empty and t precedence ≤ topOp
    precedence do
 9:             pop topOp and append to postfix
10:             if operator stack is not empty then
11:                 topOp ← top of operator stack
12:         push t onto operator stack
```

Solution Breakdown

- Mostly the code is if/else statements in a loop:
  - operands go directly to output
  - operators get pushed onto the stack
  - operators already on the stack might get popped
  - the stack is emptied to the output at the end

Running the Converter

Next Token	Action	Effect on operatorStack	Effect on postfix
w	Append w to postfix.	[ ]	w
-	The stack is empty Push - onto the stack	[-]	w
5.1	Append 5.1 to postfix	[-]	w 5.1
/	precedence(/) > precedence(-), Push / onto the stack	[/] [-]	w 5.1
sum	Append sum to postfix	[/] [-]	w 5.1 sum
*	precedence(*) equals precedence(/) Pop / off of stack and append to postfix	[-]	w 5.1 sum /

Running the Converter

Next Token	Action	Effect on operatorStack	Effect on postfix
*	precedence(*) > precedence(-), Push * onto the stack		w 5.1 sum /
2	Append 2 to postfix		w 5.1 sum / 2
End of input	Stack is not empty, Pop * off the stack and append to postfix		w 5.1 sum / 2 *
End of input	Stack is not empty, Pop - off the stack and append to postfix		w 5.1 sum / 2 * -

Evaluate Postfix Expressions

- Now that we can convert infix to postfix, how can we find the results of the expression?
- Using a stack, of course!
- We will scan through the postfix expression only once to calculate the result
- This means that we can evaluate any infix expression by scanning through an expression just twice – no jumping around in the expression or repeated scans

Postfix Evaluation Algorithm

```
EVALUATE(expression)
 1: operand stack ← empty Stack
 2: while expression has tokens left do
 3:     t ← next token
 4:     if t is an operand then
 5:         push t onto operand stack
 6:     else if t is an operator then
 7:         pop right operand off stack
 8:         pop left operand off stack
 9:         evaluate operator with two operands
10:         push result onto stack
11: return popped stack result
```