

Data Structures

Stacks

CS284

Structure of this week's classes

Stacks

Applications

Implementing a Stack

Stack

- ▶ 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

Operations on Stacks

- ▶ 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 (**boolean** `empty()`)
 - ▶ inspect the top element (`E peek()`)
 - ▶ retrieve the top element (`E pop()`)
 - ▶ put a new element on the stack (`E push(E obj)`)

StackInt<E> interface

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

An example: a stack of strings `java.util.List` interface

Jonathan
Dustin
Robin
Debbie
Rich

(a)

Dustin
Robin
Debbie
Rich

(b)

Philip
Dustin
Robin
Debbie
Rich

(c)

- ▶ “Rich” is oldest element on stack; “Jonathan” is 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)

Stacks

Applications

Implementing a Stack

Detecting Palindromes

- ▶ Palindrome: a string that reads identically in either direction, letter by letter
 - ▶ “kayak”
 - ▶ “I saw I was I”
 - ▶ “Able was I ere I saw Elba”
 - ▶ “Level, madam, level”
- ▶ We ignore case and whitespace
- ▶ Problem: Write a program that reads a string and determines whether it is a palindrome

Detecting Palindrome

- ▶ How should we tackle this problem?
- ▶ The solution depends on the data structure of that represents the palindrome!

If the Palindrome is a String/Array

- ▶ We can fetch the character based on the index;
- ▶ Use the two pointers approach:
 - ▶ If the char at the left pointer \neq that at the right pointer, return false
 - ▶ Move the left/right pointer;
- ▶ e.g., kayak

What if No Index?

- ▶ For example, detecting palindrome in a single linked list
 $k \rightarrow a \rightarrow y \rightarrow a \rightarrow k$
- ▶ In a single-linked list, can only traverse in one direction
- ▶ Stack can help "reverse" a linked list:

```
public boolean isPalindrome() {  
    Node node = this.head;  
    Stack s = new Stack();  
    while (node != null) {  
        s.push(node.data);  
        node = node.next;  
    }  
    node = this.head;  
    while (node != null) {  
        char this_char = s.pop();  
        if (node.data != this_char)  
            return false;  
        node = node.next;  
    }  
    return true;  
}
```

Balanced Parenthesis

- ▶ 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!

Code

```
public boolean isBalanced(String expression) {  
    Stack<Integer> s = new Stack<Integer>();  
    int index = 0;  
    while (index < expression.length()) {  
        char nextCh = expression.charAt(index);  
        if (nextCh == '(') {  
            s.push(1);  
        }  
        else if (nextCh == ')') {  
            try  
            {  
                s.pop();  
            }  
            catch (Exception e) {  
                return false;  
            }  
        }  
        index++;  
    }  
    return s.empty();  
}
```

Why is `isBalanced` Correct?

Context-free grammar for balanced-parenthesis:

- ▶ Rule 1: $E \rightarrow E E$;
- ▶ Rule 2: $E \rightarrow (E)$;
- ▶ Rule 3: $E \rightarrow \emptyset$;

Assertion 1. If the parenthesis is balanced, `isBalanced` must return true:

- ▶ `isBalanced` returns false only if it pops more times than the pushed times;
- ▶ Using math induction, we can prove any sequence generated by CFG has $\# \text{ pushes} \geq \# \text{ pops}$;
 - ▶ Using rule 1: trivial
 - ▶ Using rule 2: pushes is 1 larger than pushes in shorter sequence;

Why is `isBalanced` Correct?

Context-free grammar for balanced-parenthesis:

- ▶ $E \rightarrow E E$;
- ▶ $E \rightarrow (E)$;
- ▶ $E \rightarrow \emptyset$;

Assertion 2. If `isBalanced` returns true, the parenthesis must be balanced;

- A At each index i , $\text{count}((, i) \geq \text{count}() , i)$;
- B At the last index, $\text{count}((, n - 1) = \text{count}() , n - 1)$;
- ▶ We call a sequence that satisfy both A and B a **mountain** sequence;

Why is isBalanced Correct?

Context-free grammar for balanced-parenthesis:

- ▶ $E \rightarrow E E;$
- ▶ $E \rightarrow (E);$
- ▶ $E \rightarrow \emptyset;$

Theorem 2. Any mountain sequence must be a balanced sequence of parenthesis (i.e., they can be generated using the CFG grammar above).

Proof. If a sequence is a mountain sequence of length $n \geq 2$, the first and last parenthesis must be (and).

- ▶ If the first parenthesis is), it violates A;
- ▶ If the last parenthesis is (, it violates B (why?);

Why is `isBalanced` Correct?

Context-free grammar for balanced-parenthesis:

- ▶ $E \rightarrow E E$; (rule 1)
- ▶ $E \rightarrow (E)$; (rule 2)
- ▶ $E \rightarrow \emptyset$;

Theorem 2. Any mountain sequence must be a balanced sequence of parenthesis (i.e., they can be generated using the CFG grammar above).

Proof (cont.). By math induction, suppose the theorem is true for all sequences of length $n - 1$ or shorter.

- ▶ For any mountain sequence s of length n , the last char s_n must be $)$;
- ▶ If s_{n-1} is $($, use math induction and apply rule 1;
- ▶ If s_2 is $)$, use math induction and apply rule 1;

Why is `isBalanced` Correct?

Context-free grammar for balanced-parenthesis:

- ▶ $E \rightarrow E E$; (rule 1)
- ▶ $E \rightarrow (E)$; (rule 2)
- ▶ $E \rightarrow \emptyset$;

Theorem 2. Any mountain sequence must be a balanced sequence of parenthesis (i.e., they can be generated using the CFG grammar above).

Proof (cont.).

- ▶ If s_2 is (and s_{n-1} is), and at some point P in the middle, the mountain sequence is at the sea level (watch the video recording for definition), we can separate the sequence at P and apply rule 1;
- ▶ If s_2 is (and s_{n-1} is), and the mountain sequence is always above the sea level, we can apply rule 2;



Stacks

Applications

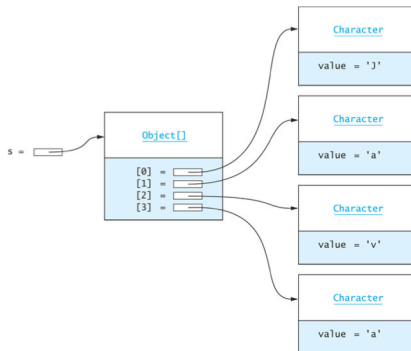
Implementing a Stack

A Stack as an Extension of Vector

- ▶ Java API includes Stack class as part of package `java.util`:

```
public class Stack<E> extends Vector<E>
```

- ▶ Elements of a Vector can be accessed using an integer index and the size can grow or shrink as needed to accommodate the insertion and removal of elements



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

We can use Vector's add method to implement push:

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

pop can be coded as

```
public E pop throws EmptyStackException {  
    try {  
        return remove(size() - 1);  
    } catch (ArrayIndexOutOfBoundsException ex) {  
        throw new EmptyStackException();  
    }  
}
```

Disadvantage of Stack as Subclass of Vector

- ▶ 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](#)

```
Stack<Integer> s = new Stack<Integer>();  
s.push(3);  
s.push(4);  
System.out.println(s.elementAt(0)); // prints 3
```

Implementing a Stack with a List Component

- ▶ Alternative: write a class, `ListStack`, that has a `List` component (in the example below, `theData`)

```
public class ListStack<E> implements StackInt<E>{  
    private List<E> theData;  
    public ListStack() {  
        theData = new ArrayList<E>();  
    }  
    ...  
}
```

Implementing a Stack with a List Component

- ▶ We can use `ArrayList`, `Vector`, or the `LinkedList` classes, as all implement the `List` interface.
- ▶ The `push` method, for example, can be coded as

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

- ▶ A class which adapts methods of another class by giving different names to essentially the same methods (`push` instead of `add`) is called an **adapter class**
- ▶ Writing methods in this way is called **method delegation**

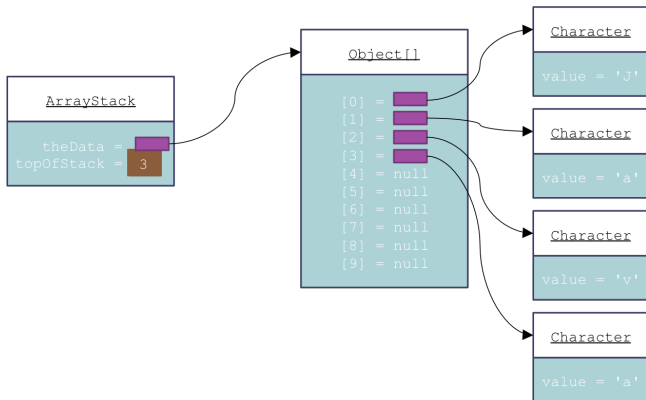
Implementing a Stack Using an Array

If we implement a stack as an array, we would need...

```
public class ArrayStack<E> implements StackInt<E> {  
    private E[] theData;  
    int topOfStack = -1;  
    private static final int INITIAL_CAPACITY = 10;  
  
    @SuppressWarnings("unchecked")  
    public ArrayStack() {  
        theData = (E[])new Object[INITIAL_CAPACITY];  
    }  
}
```

- ▶ Note: The `SuppressWarnings` annotation is just to avoid the compiler from warning: "Type safety: Unchecked cast from `Object[]` to `E[]`"

Implementing a Stack Using an Array

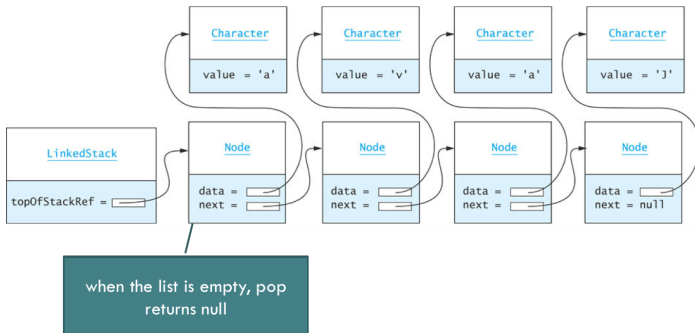


```
public E push(E obj) {  
    if (topOfStack == theData.length-1) { reallocate(); }  
    topOfStack++;  
    theData[topOfStack] = obj;  
    return obj;  
}
```

Implementing a Stack Using an Array

```
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    return theData[topOfStack--];  
}
```

Implementing a Stack as a Linked List



Implementing a Stack as a Linked List

```
import java.util.EmptyStackException;

public class LinkedStack<E> implements StackInt<E> {

    private static class Node<E> {
        private E data;
        private Node next;
    }
}
```

Implementing a Stack as a Linked List

```
// Constructors
private Node(E dataItem) {
    data = dataItem;
    next = null;
}

private Node(E dataItem, Node<E> nodeRef) {
    data = dataItem;
    next = nodeRef;
}
} //end class Node
```

Implementing a Stack as a Linked List

```
// Data Fields
/** The reference to the first stack node. */
private Node<E> topOfStackRef = null;

/** Insert a new item on top of the stack.
    post: The new item is the top item on the stack.
           All other items are one position lower.
    @param obj The item to be inserted
    @return The item that was inserted
 */
public E push(E obj) {
    topOfStackRef = new Node<E>(obj, topOfStackRef);
    return obj;
}
```

Implementing a Stack as a Linked List

```
/** Remove and return the top item on the stack.  
    pre: The stack is not empty.  
    post: The top item on the stack has been  
          removed and the stack is one item smaller.  
    @return The top item on the stack  
    @throws EmptyStackException if stack is empty  
 */  
public E pop() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        E result = topOfStackRef.data;  
        topOfStackRef = topOfStackRef.next;  
        return result;  
    }  
}
```


Implementing a Stack as a Linked List

```
/** Return the top item on the stack.  
    pre: The stack is not empty.  
    post: The stack remains unchanged.  
    @return The top item on the stack  
    @throws EmptyStackException if stack is empty  
 */  
public E peek() {  
    if (empty()) {  
        throw new EmptyStackException();  
    }  
    else {  
        return topOfStackRef.data;  
    }  
}
```

Implementing a Stack as a Linked List

```
/** See whether the stack is empty.  
    @return true if the stack is empty  
    */  
public boolean empty() {  
    return topOfStackRef == null;  
}  
}
```

Comparison of Implementations

- ▶ Extending a `Vector` (as is done by Java) is a poor choice for stack implementation, since all `Vector` methods are accessible
- ▶ The easiest implementation uses a `List` component (`ArrayList` is the simplest) for storing data
 - ▶ An underlying array requires reallocation of space when the array becomes full, and
 - ▶ An underlying linked data structure requires allocating storage for links
 - ▶ As all insertions and deletions occur at one end, they are constant time, $\mathcal{O}(1)$, regardless of the type of implementation used

More Applications

- ▶ Evaluating postfix expressions
- ▶ Converting postfix to infix expressions

Homework 3

- ▶ Given a sequence of balanced parentheses, print the trace of CFG generation rules it has used, e.g., $s_1 = (((()()))$, the output should be $s_1 \rightarrow (s_2), s_2 \rightarrow (s_3), s_3 \rightarrow s_4 s_5, s_4 \rightarrow (s_6), s_5 \rightarrow (s_7), s_6 \rightarrow \emptyset, s_7 \rightarrow \emptyset$

Stack Applications

factorial

n= 3
(return to): main

factorial

n= 2
(return to): factorial

factorial

n = 1
(return to): factorial

factorial

n = 0
(return to): factorial

return value: 1