Part 2. Stacks

Instructor: Sangmi Pallickara (sangmi@cs.colostate.edu)
Department of Computer Science
Colorado State University

Outline

- **Stacks?**
- Applications using Stacks
- Implementing Stacks
- Relationship between Stacks and Recursive Programming

Linear, time ordered structures

- Data Structures that reflects *a temporal relationship*
  - *Order* of **removal** based on the *order* of **insertion**
- We will consider
  - ***"First come, first serve"***
    - First Come, First Out: *FIFO (queue)*
  - ***"take from the top of the pile"***
    - Last In, First Out: *LIFO (stack)*

What can we do with Coin dispenser?

- "**push**" a coin into the dispenser.
- "**pop**" a coin from the dispenser.
- "**see**" the coin on top.
- "**check**" whether this dispenser is empty or not.

Stacks

- ***Last In First Out (LIFO***) structure
- ***Add/Remove*** from the **same** end

Stack Operations

- **`isEmpty()`**: determine whether stack is empty
- **`push()`**: add a new item to the stack
- **`pop()`**: remove the item added most recently
- **`peek()`**: retrieve the item added most recently
- **`createStack()`**: create an empty stack
- **`removeAll()`**: remove all the items

7

Outline

- Stacks?
- **Applications using Stacks**
- Implementing Stacks
- Relationship between Stacks and Recursive Programming

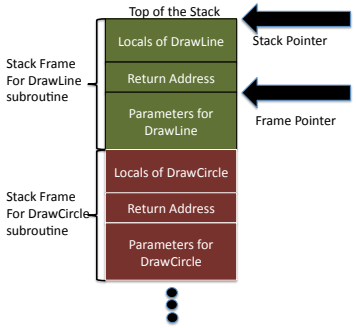
8

Applications – *Call Stack*

- Stack data structure to store information about ***active subroutines*** or ***computer program***.
- Memory allocation for tracking the execution of functions such as a nested function.
  - First method that returns is the last one invoked
- Element of call stack – activation record
  - *Parameters*
  - *Local variables*
  - *Return address*: pointer to the next instruction to be executed in calling method

9

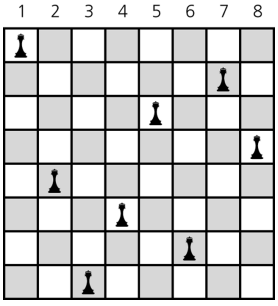
Call Stack



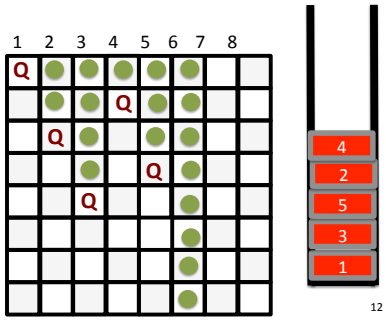
10

Applications – Backtracking

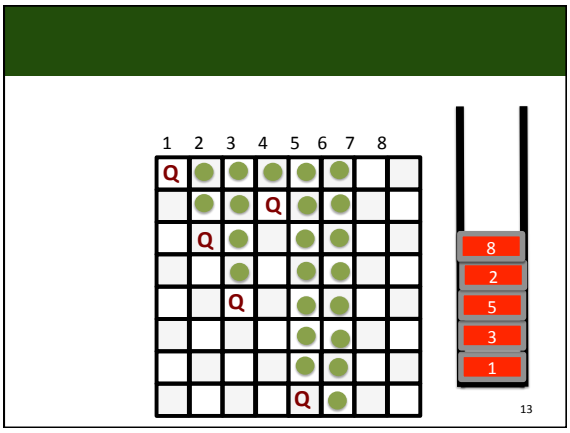
- Looking for a solution based on a ***guess***.
- Keeping the guesses (temporary answers) in the stack.
- When the process faces deadend, go back one step (visit the first one in the stack) and repeat the process.



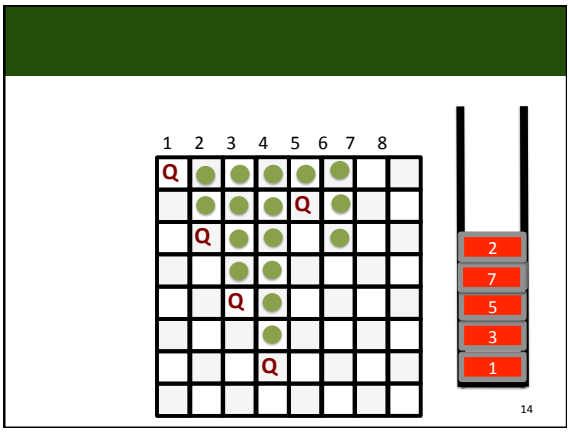
11



12



13



14

Checking for Balanced Braces

1. Each time you **encounter a "}"**, it matches an **already encountered "{"**
2. When you reach the end of the string, you have **matched all of the "{"** you have encountered

15

Peudocode

```
while ( not at the end of the string){
     if (the next character is a "{"){
          aStack.push("{")
     }
     else if (the character is a "}") {
          openBrace = aStack.pop()
     }
}
```

16

String "abc{{def}ijm}" docode with example

"a" "b" "c" "{" "{" "d" "e" "f" "}" "i" "j" "m"

```
while ( not at the end of the String ){
     if (the next character is a "{")
          aStack.push("{")
     }
     else if (the character is a "}") {
          openBrace = aStack.pop()
     }
}
```

{ {

17

Algebraic Expressions

- Evaluating *Postfix* Expressions
  - Assumptions:
    - String is a syntactically correct postfix expression
    - No unary operator
    - No exponentiation operators
    - Operands are single lowercase letters

18

Peudocode

String "2 3 4 + *"

"2" "3" "4" "+" "*"

```
for ( each character ch in the string){
  if(ch is an operand){
     push value that operand ch
        represents onto stack
  } else{
     operand2 = Pop the top of the stack
     operand1 = Pop the top of the stack
     result = operand1 op operand2
     push result onto stack
  }
}
```

WHY?

19

Outline

- Stacks?
- Applications using Stacks
- **Implementing Stacks**
- Relationship between Stacks and Recursive Programming

20

Implementing Stacks

- Should perform the ***stack operations***
- *Array-based* implementation
- *Reference-based* implementation
- *List-based* implementation

21

Array-Based Implementation

- Use an array of Objects (called items)

```
public class StackArrayBased implements
kInterface{
inal int MAX_STACK = 50;
   private Object items[];
   private int top;
   public StackArayBased(){
     items = new Object [MAX_STACK];
     top = -1;
   } end default constructor
```

WHY?

22

isEmpty()

```
public boolean isEmpty(){
   return top < 0;
}
```

23

push()

```
public void push (Object newItem) throws
  StackException{
  if (!isFull()) {
     item[++top] = newItem;
  }else{
     throw new StackException(your_error_message);
  }
}
```

24

pop( )

```
public Object pop() throws StackException{
    if (!isEmpty()){
        return items[top--];
    } else {
      throw new StackException(your_error_message);
    }
}
```

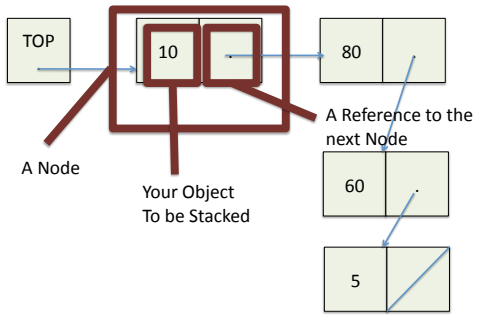
25

peek( )

```
public Object peek() throws StackException{
  if(! isEmpty()){
        return items[top];
  }else {
    throw new StackException(your_error_message);
  }
}
```

26

Reference-Based Implementation



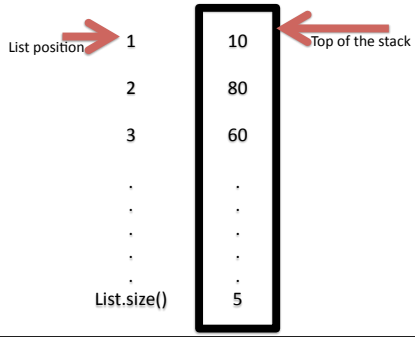
27

```
private Node top;
public void push (Object newItem){
    top = new Node(NewItem,top)
}

public Object pop() throws StackException{
    if (!isEmpty()){
        Node temp = top;
        top = top.next;
        return temp.item;
    } else {… exception handling}
}
```

28

Implementation that uses *List*



29

```
public void push(Object newItem){
     list.add(0, newItem);
}
public Object pop() throws StackException{
     if (!list.isEmpty()){
          Object temp = list.get(0);
          list.remove(0);
          return temp;
     } else exception handling
}
public void popAll(){
  list.removeAll();
}
```

30

Comparison of Implementation

- What are the pros and cons for different implementation styles?
- *Array based?*
- *Reference based?*
- *List based?*

-- Array based implementation can provide fixed size stack

-- List based: Why not just List?

31

How about java.util.Stack?

- Derived from Vector.
- `empty(), peek(), pop(), push(), search(Object o)`

32

Relationship between Stacks and Recursion

- Most implementations of recursion can maintain a stack of activation records.
- Backtracking example
- Within recursive calls, the most recently executed call is stored at the top of the stack.
- Store and access the same point of the data structure.

33

Next Reading

- Chap. 8 Queues

34