CSCI 136
Data Structures & Advanced Programming

Jeannie Albrecht

Lecture 16

March 17, 2014

Administrative Details

- Handout: Lab 6
  - You may work with a partner again this week
  - Due after break (but do yourself a favor and finish this week!)
  - Due TUESDAY instead of Monday
    - But watch out for 134 conflicts on Monday night
    - Also no 136 TAs on duty on Mondays
    - But I have office hours on Monday afternoon
- You'll get Labs 3 and 4 back on Wed in lab
- You'll get midterm back after break

Last Time

- Learned about DoublyLinkedLists
- Started talking about stacks

Today's Outline

- Continue discussing stacks
- Learn about infix and postfix
- Talk about how stacks can be used to solve mazes

Note about Stack Implementations

- structure5.StackArray
  - int top, Object data[ ]
  - Add/remove from index top

  + all operations are O(1)
  – wasted/run out of space

- structure5.StackVector
  - Vector data
  - Add/remove from tail

  +/– most ops are O(1) (add is O(n) in worst case)
  – wasted space

- structure5.StackList
  - SLL data
  - Add/remove from head

  + all operations are O(1)
  +/– O(n) space overhead

Note about Terminology

- When using stacks:
  - pop = remove
  - push = add
  - peek = get
- In Stack interface, pop/push/peek methods call add/remove/get methods that are defined in Linear interface
- But add does not really exist in Stack interface (it is inherited from Linear)

Recall: Evaluating Arithmetic Expressions

- Computer processes use stacks to evaluate arithmetic expressions
- Example: x*y+z
  - First rewrite as xy*z+
  - Then:
    - push x
    - push y
    - mult (pop twice, multiply, push result)
    - push z
    - add (pop twice, add, push result)

Converting Expressions

- We (i.e., humans) primarily use "infix" notation to evaluate expressions
  - (x+y)*z
- Computers use "postfix" (also called Reverse Polish) notation
  - xy+z*
  - Operators appear after operands, parentheses not necessary
- How do we convert between the two?
  - Compilers do this for us

Converting Expressions

- Example: x*y+z*w
- Conversion
  1) Add full parentheses to preserve order of operations
     (x*y)+(z*w)
  2) Move all operators (+-*/) after operands
     (xy*)(zw*)+
  3) Remove parentheses
     xy*zw*+

Use Stack to Evaluate Postfix Exp

- While there are input "tokens" (i.e., symbols) left:
  - Read the next token from input.
  - If the token is a value, push it onto the stack.
  - Else, the token is an operator that takes n arguments.
    - (It is known a priori that the operator takes n arguments.)
    - If there are fewer than n values on the stack → error.
    - Else, pop the top n values from the stack.
      - Evaluate the operator, with the values as arguments.
      - Push the returned result, if any, back onto the stack.
  - If there is only one value on the stack, that value is the result of the calculation.
  - Else if there are more values in the stack w/o operators, there are too many input values → error.

Example

- (x*y)+(z*w) -> xy*zw*+
- Evaluate:
  - Push x
  - Push y
  - Mult (Pop y, Pop x, Push x*y)
  - Push z
  - Push w
  - Mult (Pop w, Pop z, Push z*w)
  - Add (Pop x*y, Pop z*w, Push (x*y)+(z*w))
  - One value left, so we're done.

Mazes

- How can we use a stack to solve a maze?
  - http://www.cs.williams.edu/~jeannie/cs136/lectures/lecture15/Kim/index.html
- Properties of mazes:
  - A maze is simply a matrix of cells
  - There is a start cell and finish cell
  - Want to find a path of adjacent cells between start and finish
- Strategy: Consider unvisited cells as "potential tasks"
  - Use linear structure (stack) to keep track of *outstanding* tasks (i.e., unvisited cells that are adjacent to visited cells)

Solving Mazes

- We'll use two classes to solve our maze:
  - Position
  - Maze
- General strategy:
  - Use stack to keep track of path from start
  - If we hit a dead end, backtrack by popping location off stack
  - Leave "bread crumbs" to make sure we don't visit the same place twice

Backtracking Search

- Try one way (favor north and east)
- If we get stuck, go back and try a different way
- We will eventually either find a solution or exhaust all possibilities
- Also called a "depth first search"

- Lots of other algorithms that we will not explore: http://www.astrolog.org/labyrnth/algrithm.htm

Position class

- Represent position in maze as (x,y) coordinate
- class Position has 5 relevant methods:
  - `Position getNorth()`
  - `Position getSouth()`
  - `Position getEast()`
  - `Position getWest()`
  - `boolean equals()`

Maze class

- Relevant Maze methods:
  - `Maze(String filename)`
    - Constructor; takes file describing maze as input
  - `void visit(Position p)`
    - Visit position p in maze
  - `boolean isVisited(Position p)`
    - Returns true iff p has been visited before
  - `Position start()`
    - Return start position
  - `Position finish()`
    - Return finish position
  - `boolean isClear(Position p)`
    - Returns true iff p is a valid move and is not a wall

Implementing Maze

- Iteratively: Maze.java
- Recursively: RecMaze.java
  - Recursive methods keep an implicit stack
  - Each recursive call adds another layer to the stack

Method Call Stacks

- In JVM, need to keep track of method calls
- JVM maintains stack of method invocations (called frames)
- Stack of frames
  - Receiver object, parameters, local variables
- On method call
  - Push new frame, fill in parameters, run code
- Exceptions print out stack
- Example: StackEx.java
- Recursive calls recurse too far: StackOverflowException
  - Overflow.java

Recursive Call Stacks

```
public static long factorial(int n) {
  if (n <= 1) // base case
     return 1;
  else
     return n * factorial(n - 1);
}

public static void main(String args[]) {
  System.out.println(factorial(3)};
}
```