CSCI 136
Data Structures &
Advanced Programming

Lecture 14
Fall 2018
Instructor: Bills
Announcements

- Mid-Term Review Session
  - Monday (10/15), 7:00-8:00 pm in TPL 203
  - No prepared remarks, so bring questions!
- Mid-term exam is Wednesday, October 17
  - During your normal lab session
  - You’ll have 1 hour & 45 minutes (if you come on time!)
  - Closed-book
  - Covers Chapters 1-7 & 9 and all topics up through Linked Lists
  - A “sample” mid-term and study sheet are available online
    - See Handouts & Problem Sets
Last Time

• QuickSort and Sorting Wrap-Up
• Linear Structures
  • The Linear Interface (LIFO & FIFO)
  • The AbstractLinear and AbstractStack classes
• Stack Implementations
  • StackArray, StackVector, StackList,
Today: Linear Structures

- Stack applications
  - Expression Evaluation
  - PostScript: Page Description & Programming
  - Mazerunning (Depth-First-Search)
Evaluating Arithmetic Expressions

• Computer programs regularly use stacks to evaluate arithmetic expressions

• Example: \( x \times y + z \)
  • First rewrite as \( xy \times z + \) (we’ll look at this rewriting process in more detail soon)

• Then:
  • push \( x \)
  • push \( y \)
  • * (pop twice, multiply popped items, push result)
  • push \( z \)
  • + (pop twice, add popped items, push result)
Converting Expressions

• We (humans) primarily use “infix” notation to evaluate expressions
  • \((x+y)*z\)

• Computers traditionally used “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.
  • The top value on the stack is the result of the calculation.
  • Note that results can be left on stack to be used in future computations:
    • Eg: 3 2 * 4 + followed by 5 / yields 2 on top of stack
Example

- \((x*y)+(z*w) \rightarrow xyzw++\)

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 \((x*y)+(z*w)\)
- Result is now on top of stack
Lab Preview: PostScript

- PostScript is a programming language used for generating vector graphics
  - Best-known application: describing pages to printers
- It is a stack-based language
  - Values are put on stack
  - Operators pop values from stack, put result back on
  - There are numeric, logic, string values
  - Many operators
- Let’s try it: The ‘gs’ command runs a PostScript interpreter....
- You’ll be writing a (tiny part of) gs in lab soon....
Lab Preview: PostScript

- Types: numeric, boolean, string, array, dictionary
- Operators: arithmetic, logical, graphic, ...
- Procedures
- Variables: for objects and procedures
- PostScript is just as powerful as Java, Python, ...
  - Not as intuitive
  - Easy to automatically generate
- Example: Recursive factorial procedure
  ```postscript
  /fact { dup 1 gt { dup 1 sub fact mul } if } def
  ```
- Example: Drawing (see picture.ps)
Mazes

• How can we use a stack to solve a maze?

• Properties of mazes:
  • We model a maze as a rectangular grid of cells
  • There is a *start* cell and one or more *finish* cells
  • Goal: Find path of *adjacent* free cells from *start* to *finish*

• Strategy: Consider unvisited cells as “potential tasks”
  • Use linear structure (stack) to keep track of current path being explored
Solving Mazes

- We’ll use two objects to solve our maze:
  - Position: Info about a single cell
  - Maze: Grid of Positions

- General strategy:
  - Use stack to keep track of path from start
  - If we hit a dead end, backtrack by popping location off stack
  - Mark discarded cells to make sure we don’t visit the same paths 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/labyrinth/algrithm.htm
A “Pseudo-Code” Sketch

// Initialization
Read cell data (free/blocked/start/finish) from file data
Mark all free cells as unvisited
Create an empty stack S
Mark start cell as visited and push it onto stack S

While (S isn’t empty && top of S isn’t finish cell)
  current ← S.peek()  // current is top of stack
  If (current has an unvisited neighbor x)
    Mark x as visited; S.push(x)  // x is explored next
  Else S.pop()
If finish is on top of S then success else no solution
Is Pseudo-Code Correct?

• Tools
  • Concepts: adjacent cells; path; simple path; path length; shortest path; distance between cells; reachable from cell
  • Solving a maze: is finish reachable from start?

• Theorem: The pseudo-code will either visit finish or visit every free cell reachable from start

• Proof: Prove that if algorithm does not visit finish then it does visit every free cell reachable from start
  • Do this by induction on distance of free cell from start
  • Base case: distance 0. Easy
  • Induction: Assume every reachable free cell of distance at most \( k \geq 0 \) from start is visited. Prove for \( k+1 \)
Is Pseudo-Code Correct?

• Induction Hyp: Assume every reachable free cell of distance at most \( k \geq 0 \) from start is visited.

• Induction Step: Prove that every reachable free cell of distance \( k+1 \) from start is visited.
  • Let \( c \) be a free cell of distance \( k+1 \) reachable from start
  • Then \( c \) has a free neighbor \( d \) that is distance \( k \) from start and reachable from start
  • But then by induction, \( d \) is visited, so it was put on stack
  • So each free neighbor of \( d \) is visited by algorithm

• Done!
Recursive “Pseudo-Code” Sketch

Boolean RecSolve(Maze m, Position current)

If (current equals finish) return true
Mark current as visited

next ← some unvisited neighbor of current (or null if none left)
While (next does not equal null && recSolve(m, next) is false)
    next ← some unvisited neighbor of current (or null if none left)
Return next ≠ null

• To solve maze, call: Boolean recSolve(m, start)
• To prove correct: Induction on distance from current to finish
• How could we generate the actual solution?
Implementing A Maze Solver

• Iteratively: Maze.java
• Recursively: RecMaze.java
  • Recursive method keeps an implicit stack
    • The method call stack
  • Each recursive call adds to the stack
Implementation: Position class

- Represent position in maze as \((x,y)\) coordinate
- class Position has several relevant methods:
  - Find a neighbor
    - Position getNorth(), getSouth(), getEast(), getWest()
  - boolean equals()
  - Check states of position
    - boolean isVisited(), isOpen()
  - Set states of position
    - void visit(), setOpen(boolean b)
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(), finish()
    • Return start /finish positions
  • Position nextAdjacent(Position p)
    • Return next unvisited neighbor of p---or null if none
  • boolean isClear(Position p)
    • Returns true iff p is a valid move and is not a wall
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
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));
}