More Linear ADTs: Stacks and Queues

Chapter 6: Sections 6.1, 6.3: 6.3.1 - 6.3.2, and 6.4: 6.4.1 - 6.4.2

CS 2334
University of Oklahoma
Brian F. Veale
Linear ADTs

- **Linear Abstract Data Types**
  - Store elements in a sequential manner
  - All elements except first have a specific predecessor
  - All elements except last have a specific successor
  - There is a 1-to-1 relationship between elements

- **Examples of Linear ADT’s**
  - String and StringBuffer
  - List
  - Stack
  - Queue and PriorityQueue

- We’re interested in using all these ADT’s but not implementing them.
Stacks

- Places restrictions on where insertion, deletions, and retrieval operations can occur
- Stack – a last-in, first-out (LIFO) linear data structure
  - The only object that can be accessed is the last one placed on the stack
    - Can only retrieve, change, or remove the object that is on the top of the stack.
  - Ex: A stack of plates
- java.util.Stack<E>
  - Not a part of the JCF
  - [http://java.sun.com/j2se/1.5.0/docs/api/java/util/Stack.html](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Stack.html)
java.util.Stack<E>

- Extends Vector with five operations that allow a vector to be treated as a stack.

- Interface:
  - E peek()
    - Looks at the object at the top of this stack without removing it from the stack.
  - E pop()
    - Removes the object at the top of this stack and returns that object as the value of this function.
  - E push(E item)
    - Pushes an item onto the top of this stack.
  - boolean empty()
    - Tests if this stack is empty.
Applications of Stacks

• Common applications of stacks include:
  • Backtracking algorithms
  • Histories
  • Evaluating Postfix Expressions
  • Converting Infix to Postfix
    • Covered in the text
  • Undo Functions
  • System Stack
    • Manages method calls and memory
    • Mentioned in the text
  • Parsing expressions for programming languages
    • Used by compilers such as javac
Infix to Postfix Conversion

• **Infix expression**
  • Operators are located between their operands
  • $3 + 2 \times 4 - 1$

• **Postfix Expression**
  • The operator immediately follows its operand(s)
  • $3 2 4 \times + 1 -$
  • Also known as RPN (Reverse Polish Notation)

• **Can use a stack to convert an infix expression to a postfix expression**
  • Most calculators do this conversion
Infix to Postfix Conversion

Idea: Locate the highest precedence to determine the order of operations using an operator stack:

Scan expression from left to right

If next token is an operand: Add it to the postfix expression.

If the next token is an operator:
  If operatorStack.top().precedence() >= operator.precedence()
    Pop operators off of the stack and add them to the postfix expression until the stack is empty or
    operatorStack.top().precedence() < operator.precedence().
    Push the current operator on to the top of the stack.
  Else push the operator on to the top of the stack.

Repeat until the entire expression has been processed

Pop operator(s) off of the top of the stack, adding them to the expression until the stack is empty.
Postfix Expression Evaluation

Create a new stack
While there are more tokens in the expression
  Get the next token
  If the token is an operand
    Push the operand onto the stack
  Else if the token is an operator
    Pop the top two operands from the stack
    Use the operator to evaluate the two operands just popped
    Push the resulting expression onto the stack
  End if
End while
Return the value at the top of the stack
Infix/Postfix Expression Demo

• *Infix to postfix conversion*
  • *Infix2Postfix*

• *Postfix evaluation*
  • *PostFixEval*
Backtracking Algorithms

- **What is a backtracking algorithm?**
  - Begin in a predefined state
  - Move from state to state searching for the desired ending state
  - If you have to choose between several alternatives, pick one.
  - Randomly or not
  - If you reach an undesired state (a dead end), go back to the last point with an unexplored state
  - Repeat until you have exhausted all states or reached desired ending state
Two Implementation Approaches for Backtracking

• Recursion
  • We might discuss recursion later in the course.
  • Covered in Data Structures
• Use a stack
Solving a Maze

- Can use backtracking
- Maze Demo:
  - Maze
  - MazeGUI
  - TextGridPanel
  - maze1.txt
  - maze2.txt
  - maze3.txt
Backtracking in the Maze Demo

- $s$ denotes the starting position
- $$ denotes the goal position
- # denotes a wall
- The program moves from one square in the maze to another looking for the square with $\$$$. As it does this, it changes from state to state.
- When a state is visited the first time, it is marked with an 'o'. When a state is visited again and the algorithm backtracks to the state before it, the state is marked with a '.'.
- The first time a state has been visited, it's position in the maze is stored onto a stack. When the algorithm is in a state and cannot make any legal moves, it backtracks to the previous state by 'popping' it's position off of the stack.
Backtracking Algorithm for the Maze

Push the starting state onto the stack.
done = stack.empty()
while( done == false ) {
    use top of stack to find next state and push it onto the stack
    if we cannot move forward to another state, then pop the top state off of the stack
    If the goal has been reached, done = true
    If stack.empty(), done = true
}
If stack is empty, maze is unsolvable
Queues

• Queue – A first-in, first-out (FIFO) linear data structure
  • The only value we can access is the value at the front of the queue
  • Models a waiting line
  • Can be used to store information about objects waiting to access resources or request services

• java.util.Queue<E>
  • In the Java API, Queue is an interface not a implementation
  • We have to provide our own queue class.
  • Part of the JCF
java.util.Queue<E>

• E element()
  • Retrieves, but does not remove, the head of this queue.
• boolean offer(E o) - (aka enqueue)
  • Inserts the specified element into this queue, if possible.
• E peek()
  • Retrieves, but does not remove, the head of this queue, returning null if this queue is empty.
• E poll() - (aka dequeue)
  • Retrieves and removes the head of this queue, or null if this queue is empty.
• E remove() - (aka dequeue)
  • Retrieves and removes the head of this queue.
• Has other methods defined, from Collection, such as add(), isEmpty(), and size().
Applications of Queues

• Common applications of Queues
  • Controlling access to shared resources
    • Shared printers
    • Network access
    • CPU usage
    • Disk access
      • Discussed in the text
  • Simulations
    • Real world:
      • Lines at a bank, store, etc.
    • Computers
      • Simulating network scenarios in network design
Queue Demo

• Queue.java
Priority Queue

- Items enqueued to a Priority Queue are assigned a priority or rank
- When dequeued, items of a higher priority are removed before those of a lower priority
- Items of equal priority are dequeued in a FIFO order
- The interface is the same as a normal queue.
  - `java.util.PriorityQueue<E>`
    - Part of the JCF
- Examples:
  - Emergency Room
  - Processor sharing with different level users