[TAP:RTNHE] Where?
Administrative Details

• Spring Break!
Today’s Outline

• Linear Structures
  • Stack
  • Applications
• Queue
  • Applications
Mazes

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

• Properties of mazes:
  • We model a maze as a 2-d array of cells
  • There is a start cell and one or more finish cells
  • Goal: Find path from start to finish
Solving Mazes

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

- General strategy (backtracking search):  
  - Use stack to keep track of path from start  
  - Go one way (“push”)  
  - If we get stuck, go back (“pop”) and try a different way  
  - We will eventually either find a solution or exhaust all possibilities
Position Class

- Represent position in maze as \((x, y)\) coordinate
- Instance variables: int row, int col, boolean visited, boolean open
- Methods:
  - Getters and setters
  - equals()
Maze Class

- Represent position in maze as \((x, y)\) coordinate
- Instance variables: Position start, Position finish, Position[][] board
- Methods:
  - Getters and setters
  - toString()
  - Position nextAdjacent(Position current)
public Position nextAdjacent(Position cur) {
    Position next = board[cur.getRow()-1][cur.getCol()];  // W
    if (next.isOpen() && !next.isVisited()) {
        return next;
    }

    next = board[cur.getRow()][cur.getCol()+1];  // E
    if (next.isOpen() && !next.isVisited()) {
        return next;
    }

    next = board[cur.getRow()+1][cur.getCol()];  // S
    if (next.isOpen() && !next.isVisited()) {
        return next;
    }

    next = board[cur.getRow()][current.getCol()-1];  // W
    if (next.isOpen() && !next.isVisited()) {
        return next;
    }

    return null;
}
public static boolean solve(Maze maze, Position cur) {
    if (cur.equals(maze.finish())) {
        return true;
    }
    cur.visit();
    Position next = maze.nextAdjacent(cur);
    while (next != null) {
        if (solve(maze, next)) {
            System.out.print(next + " ");
            return true;
        }
        next = maze.nextAdjacent(cur);
    }
    return false;
}

solve(maze, maze.start());
IterSolver Class

```java
public static boolean solve (Maze maze)
{
    Stack <Position> path = new StackList <Position> ();

    Position cur = maze.start();
    cur.visit();
    path.push (cur);

    while (!path.empty () && !path.peek ().equals (maze.finish ()))
    {
        Position next = maze.nextAdjacent (path.peek ());
        if (next != null)
        {
            next.visit();
            path.push (next);
        }
        else
        {
            path.pop();
        }
    }

    if (!path.empty ())
    {
        System.out.println (path);
        return !path.empty ();
    }
}
```
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 \( y \)
    • \( \times \) (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
     $((x*y)(z*w)+)$
  3) Remove parentheses
     $x*y+z*w+$
Today’s Outline

- Linear Structures
  - Stack
    - Applications
  - Queue
    - Applications
Queue

- Examples: Lines at grocery store
- What methods do we need to define?
  - Queue interface methods
- New terms (only) associated with stacks
  - Enqueue = insert from the back
  - Dequeue = remove from the front
  - peek = look up the first element
Implementation (in structure 5)

- Queue interface
  - Defines enqueue/dequeue/peek methods
- 3 classes implementing the stack interface:
  - QueueArray
    - Object[] data, int head, int count
    - tail = (head + count) % data.length
  - QueueVector
    - Vector data
    - Add/remove from tail
  - QueueList
    - SLL data
    - Add/remove from head
Position Class

- Represent position in maze as (x,y) coordinate
- Instance variables: int row, int col, boolean visited, boolean open
- Methods:
  - Getters and setters
  - equals()
  - toString()
Maze Class

• Represent position in maze as \((x,y)\) coordinate

• Instance variables: Position start, Position finish, Position[][] board

• Methods:
  • Getters and setters
  • toString()
  • Position nextAdjacent(Position current)
Today’s Outline

- Linear Structures
  - Stack
    - Applications
  - Queue
    - Applications
public static boolean solve(Maze maze) {
    Queue<Position> queue = new QueueList<Position>();

    Position current = maze.start();
    queue.enqueue(current);
    current.visit();

    while(!queue.isEmpty()) {
        current = queue.dequeue();
        if(current.equals(maze.finish())){
            System.out.println(current.toPathString());
            return true;
        }

        //enqueue all neighbors
        Position neighbor=null;
        while((neighbor=maze.nextAdjacent(current))!=null) {
            neighbor.setParent(current);
            neighbor.visit();
            queue.enqueue(neighbor);
        }
    }

    return false;
}