[TAP:VMUGW] Merge vs Quick

- Which of the following is false?
  A. Both sorting algorithms have the same *best case* run time complexity
  B. Both sorting algorithms have the same *average case* run time complexity
  C. Both sorting algorithms have the same *worst case* run time complexity
  D. They are all true.
  E. Whatever

Administrative Details

- Mid-term exam is Wednesday, March 14
  - During your normal lab session
  - You’ll have approximately 1 hour & 45 minutes (if you come on time!)
  - Closed-book: Covers Chapters 1-7 & 9, handouts, and all topics up through Sorting
  - A “sample” mid-term **and** study sheet are available online

Today’s Outline

- Linear Structures
  - Stack
    - Applications

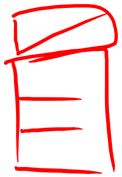
Linear Structures

- What if the application you're working on restricts where elements are inserted and removed?
  - lines at DMV ← FIFO (queue) (first in first out)
  - web browser ("[previous]") ← LIFO (stack) (last in first out)
  - data packets arriving at a router

Linear Structures

- Approaches
  - Use existing structures (vector, linked list)
  - Define new simplified structures
- Less functionality can result in:
  - Simpler implementation
  - Greater efficiency
  - Less room for error?

Stacks



- Examples: stack of trays or cups
  - Can only take tray/cup from top of stack
- What methods do we need to define?
  - Stack interface methods
- New terms (only) associated with stacks
  - Push = add to the top
  - Pop = remove from the top
  - Peek = "look" at the top

Implementation (in structure5)

- Stack interface
  - Defines pop/push/peek methods
- 3 classes implementing the stack interface:
  - StackArray (array-based)
    - int top, Object data[ ]
    - Add/remove from index top
  - StackVector (vector-based)
    - Vector data
    - Add/remove from tail
  - StackList (LL-based)
    - SLL data
    - Add/remove from *head*

– fixed size (potentially wasted space)
+ O(1) operations

+ resizeable
– potentially wasted space
+/– O(1) operations
O(n) to "ensureCapacity"

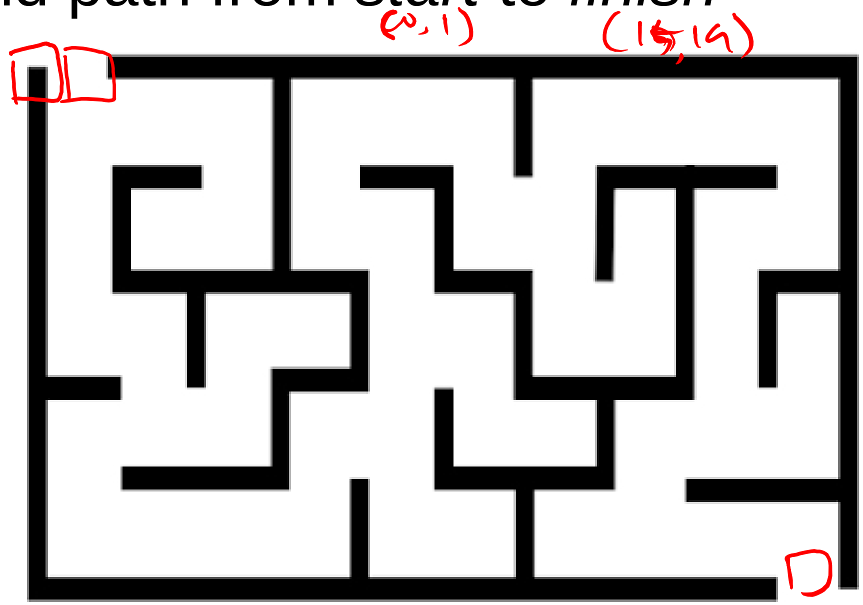
+ resizeable
+ O(1) operations

Today's Outline

- Linear Structures
  - Stack
    - 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

↑ path to finish

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)`

```
public Position nextAdjacent(Position cur) {
    Position next = board[cur.getRow()-1][cur.getCol()]; // N
    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;
}
```

RecSolver Class

```
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;
}
```

```
main() {
    solve(maze, maze.start());
}
```