10.A. Stacks

Abstraction (Last-In, First-Out) and Operations

<table>
<thead>
<tr>
<th>PUSH</th>
<th>POP</th>
<th>TOP</th>
<th>EMPTY</th>
</tr>
</thead>
</table>

Policies Correspond to Code (Θ(1) for all operations)

1. Direction of growth in array
2. What does stack pointer indicate?
   a. Next available element:
      
      ![Diagram of stack pointer and array]

      ```plaintext
      EMPTY: return sp==0;
      PUSH(x): A[sp++]=x;
      POP: return A[--sp];
      TOP: return A[sp-1];
      ```

      b. Most recently pushed element:
      
      ![Diagram of stack pointer and array]

      ```plaintext
      EMPTY: return sp==(-1);
      PUSH(x): A[++sp]=x;
      POP: return A[sp--];
      TOP: return A[sp];
      ```

Also easy to implement using a linked list (CLRS exercise 10.2-2):
Applications

1. Run-time environment for programming languages.
2. Compilers/parsing
3. Depth-first search on graphs (Notes 14).

10.B. RAT-IN-A-MAZE USING A STACK (DEPTH-FIRST SEARCH)

Array initially contains 0/1 for each position.
0=open (" "), 1=wall (".").

Stack contains positions on current path.

Array entries change to reflect search status:
2=discovered ("^"), 3=solution ("#)

http://ranger.uta.edu/~weems/NOTES2320/VERMIN/ratDFSrec.c

```c
int DFS(int row,int col)
{
    if (maze[row][col]!=0)
        return 0; // report failure
    if (row==stopRow && col==stopCol)
    {
        maze[row][col]=3;
        return 1; // report success
    }
    maze[row][col]=2;  // Mark slot as discovered
    if (!DFS(row-1,col))       // Try North
        if (!DFS(row,col+1))     // Try East
            if (!DFS(row+1,col))   // Try South
                if (!DFS(row,col-1)) // Try West
                    return 0;
    maze[row][col]=3;  // On final path
    return 1;  // Propagate success through recursion
}

int main()
{
    readInput();
    printf("Initial maze:\n");
    printMaze();
    if (DFS(startRow,startCol))
        printf("Success:\n");
    else
        printf("Failure:\n");
    printMaze();
}
```
typedef enum {init,north,east,south,west} direction;
typedef struct {
    int row,col;
    direction current;
} stackEntry;

int DFS(int row,int col)
{
    stackEntry work;
    int returnValue;
    work.row=row;
    work.col=col;
    work.current=init;
    pushStack(work);
    while (!emptyStack())
    {
        work=popStack();
        if (work.current==init)  // Just arrived here?
        {
            if (maze[work.row][work.col]!=0)  // Not an open slot?
            {
                returnValue=0;
                continue;
            }
        }
        if (work.current==west)  // No other directions to try
        {
            returnValue=0;
            continue;
        }
        // Try next direction. Push current position and new position
        work.current++;
        pushStack(work);
        switch (work.current) {
            case north: work.row--; break;
            case east: work.col++; break;
            case south: work.row++; break;
            case west: work.col--; break;
        }
        work.current=init;
        pushStack(work);
    }
    return returnValue;
}
10.C. **Evaluating Postfix Expressions Using a Stack**

Infix: $$(1 + 2) \times (3 + 1) / (1 + 1 + 1)$$

Postfix: $$1 2 + 3 1 + * 1 1 + 1 + /$$

Prefix: $$/ * + 1 2 + 3 1 + + 1 1 1$$

Evaluating Postfix – Store operands on stack until popped for operator

```c
while (unprocessed input tokens)
{
    get token;
    if (token is an operand)
        stack.push(token);
    else // token is an operator
    {
        operand2=stack.pop();
        operand1=stack.pop();
        stack.push(result of (operand1 token operand2));
    }
}
result=stack.pop();
if (!stack.empty())
    <error>
```

<table>
<thead>
<tr>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: 1</td>
</tr>
<tr>
<td>2: 1 2</td>
</tr>
<tr>
<td>+: 3</td>
</tr>
<tr>
<td>3: 3 3</td>
</tr>
<tr>
<td>1: 3 3 1</td>
</tr>
<tr>
<td>+: 3 4</td>
</tr>
<tr>
<td>*: 12</td>
</tr>
<tr>
<td>1: 12 1</td>
</tr>
<tr>
<td>1: 12 1 1</td>
</tr>
<tr>
<td>+: 12 2</td>
</tr>
<tr>
<td>1: 12 2 1</td>
</tr>
<tr>
<td>+: 12 3</td>
</tr>
<tr>
<td>/: 4</td>
</tr>
</tbody>
</table>

10.D. **Queues**

Abstraction (First-In, First-Out) and Operations

ENQUEUE (at tail)  DEQUEUE (from head)  EMPTY
Applications

1. Huffman coding using two queues
   (http://ranger.uta.edu/~weems/NOTES2320/huffman2Q.c)

   ![Diagram of Huffman coding using two queues]

   - Input trees in weight order
   - Merged trees in weight order
   - Each step: Merge lowest pair

2. Data communications

3. Message-based concurrent programming

4. Event-interrupt handlers

5. Breadth-first search
   a. Graphs (Notes 14)
   b. Rat in a maze (http://ranger.uta.edu/~weems/NOTES2320/VERMIN/ratBFSqueue.c)

   ![Rat in a maze]

   Demo of both versions of search:

   http://ranger.uta.edu/~weems/NOTES2320/VERMIN/rat.drag.html
Implementation using $A[0] \ldots A[n-1]$

<table>
<thead>
<tr>
<th>Non-Reusable</th>
<th>Circular</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Initialize</strong></td>
<td>$\text{tail} = \text{head} = 0$;</td>
</tr>
<tr>
<td><strong>EMPTY</strong></td>
<td>$\text{return tail} == \text{head}$;</td>
</tr>
<tr>
<td><strong>ENQUEUE(x)</strong></td>
<td>$A[\text{tail}++] = x$;</td>
</tr>
<tr>
<td>if (tail == n)</td>
<td>if (tail == n)</td>
</tr>
<tr>
<td>$&lt;$ error $&gt;$</td>
<td>tail = 0;</td>
</tr>
<tr>
<td></td>
<td>if (tail == head)</td>
</tr>
<tr>
<td></td>
<td>$&lt;$ confused $&gt;$</td>
</tr>
<tr>
<td><strong>DEQUEUE</strong></td>
<td>if (tail == head)</td>
</tr>
<tr>
<td>if (tail == head)</td>
<td>$&lt;$ empty $&gt;$</td>
</tr>
<tr>
<td>$&lt;$ empty $&gt;$</td>
<td>temp = $A[\text{head}++]$;</td>
</tr>
<tr>
<td>return $A[\text{head}++]$;</td>
<td>if (head == n)</td>
</tr>
<tr>
<td></td>
<td>head = 0;</td>
</tr>
<tr>
<td></td>
<td>return temp;</td>
</tr>
</tbody>
</table>

Implementation using a linked list (CLRS exercise 10.2-3):

![Diagram of a linked list](image)

Aside: Suppose a queue has pointers to outgoing messages. How would you maintain:

1. The average length of an outgoing message?
2. The maximum length of an outgoing message?
What if messages are in a *stack* instead?

Solution for queue is to use *two* stacks (CLRS exercise 10.1-6):

Initialize:

```
Initialize inStack
Initialize outStack
```

Enqueue(message, length):

```
if inStack.empty or length > inMaximum
   inMaximum=length
   inStack.push(message, length)
```

Dequeue:

```
if outStack.empty
   if inStack.empty
      <ERROR>
      (message, length)=inStack.pop
      outStack.push(message, length)
   while !inStack.empty
      (message, length)=inStack.pop
      outStack.push(message, max(outStack.top.length, length))

   (message, length)=outStack.pop
   return message
```

MaxLength:

```
if inStack.empty and outStack.empty
   <ERROR>
if outStack.empty
   return inMaximum
if inStack.empty
   return outStack.top.length
return max(inMaximum, outStack.top.length)
```

Amortized vs. actual cost of operations