CLRS 10.1

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 ($\Theta(1)$ for all operations)

1. Direction of growth in array

2. What does stack pointer indicate?

   a. Next available element:

   ![Diagram of stack operations](image)

      | 5 | 4 | 3 | 2 | 1 | 0 |
      |   |   |   | 2 | y | x |
      |   |   |   | 3 | z |   |
      |   |   |   |   |   |   |
      | SP|   |   |   |   |   |

      Empty: return sp==0;
      Push(x): A[sp]=x;
      Pop: return A[sp--];
      Top: return A[sp];

   b. Most recently pushed element:

      ![Diagram of stack operations](image)

      | 5 | 4 | 3 | 2 | 1 | 0 |
      |   |   |   | 2 | y | x |
      |   |   |   | 3 | z |   |
      |   |   |   |   |   |   |
      | SP|   |   |   |   |   |

      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

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;       // On final path
    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.row==stopRow && work.col==stopCol)  // At destination?
            {
                maze[work.row][work.col]=3;
                returnValue=1;
                continue;
            }
            maze[work.row][work.col]=2;  // Mark slot as discovered
        }
        else if (returnValue==1)  // Backtracking from successful search?
        {
            maze[work.row][work.col]=3;
            continue;
        }
        else 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

```plaintext
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>
<th>1:</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>+:</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3:</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1:</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>+:</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>*:</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>1:</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>1:</td>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>+:</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>1:</td>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>+:</td>
<td>12</td>
<td>3</td>
</tr>
<tr>
<td>/:</td>
<td>4</td>
<td></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)

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

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)

   Demo of both versions of search:
   http://ranger.uta.edu/~weems/NOTES2320/VERMIN/rat.drag.html
Implementation using $A[0] \ldots A[n-1]$ (aside starting with Spring 2020)

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

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