

CSE 3318 Notes 10: Stacks and Queues

(Last updated 1/5/24 11:26 AM)

CLRS 10.1

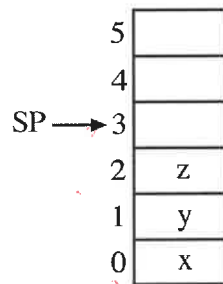
10.A. STACKS

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

PUSH POP TOP EMPTY

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:



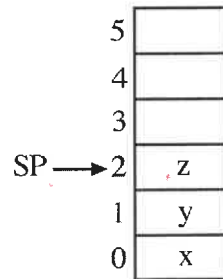
EMPTY: `return sp==0;`

PUSH(x): `A[sp++] = x;`

POP: `return A[--sp];`

TOP: `return A[sp-1];`

- b. Most recently pushed element:



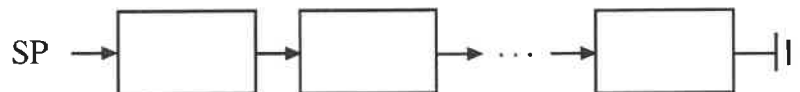
EMPTY: `return sp==(-1);`

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):



<https://ranger.uta.edu/~weems/NOTES3318/VERMIN/ratDFSstack.c>

```

...
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;
}

```

$$(1 + 2) * (3 + 4)$$

10.C. EVALUATING POSTFIX EXPRESSIONS USING A STACK

Infix: $(1 + 2) * (3 + 1) / (1 + 1 + 1)$

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

```
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>
```

	Stack
1:	1
2:	1 2
+	3
3:	3 3
1:	3 3 1
+	3 4
*	12
1:	12 1
1:	12 1 1
+	12 2
1:	12 2 1
+	12 3
/:	4

(Aside: https://en.wikipedia.org/wiki/Shunting-yard_algorithm)

10.D. QUEUES

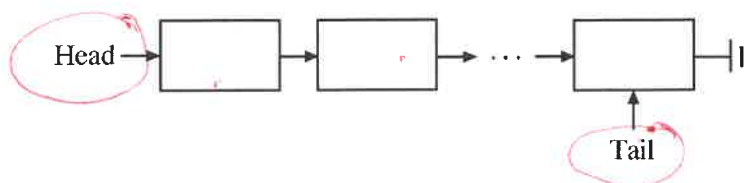
Abstraction (First-In, First-Out) and Operations

ENQUEUE (at *tail*)

DEQUEUE (from *head*)

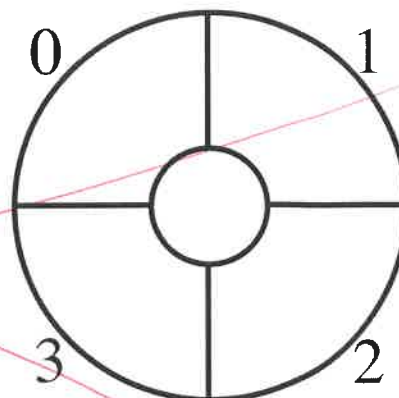
EMPTY

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



Aside: Implementation using $A[0] \dots A[n-1]$

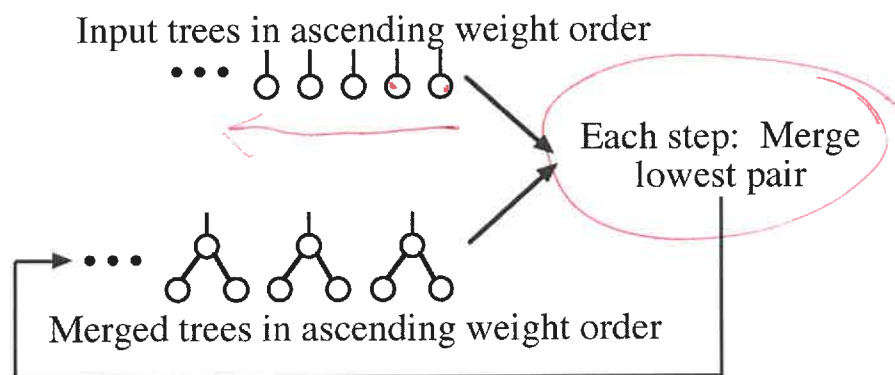
	<u>Non-Reusable</u>	<u>Circular</u>
Initialize	<code>tail=head=0;</code>	<code>tail=head=0;</code>
EMPTY	<code>return tail==head;</code>	<code>return tail==head;</code>
ENQUEUE(x)	<code>A[tail++]=x;</code> <code>if (tail==n)</code> <code>< error ></code>	<code>A[tail++]=x;</code> <code>if (tail==n)</code> <code>tail=0;</code> <code>if (tail==head)</code> <code>< confused ></code>
DEQUEUE	<code>if (tail==head)</code> <code>< empty ></code> <code>return A[head++];</code>	<code>if (tail==head)</code> <code>< empty ></code> <code>temp=A[head++];</code> <code>if (head==n)</code> <code>head=0;</code> <code>return temp;</code>



Applications

1. Huffman coding using two queues

(<https://ranger.uta.edu/~weems/NOTES3318/huffman2Q.c>)



2. Data communications

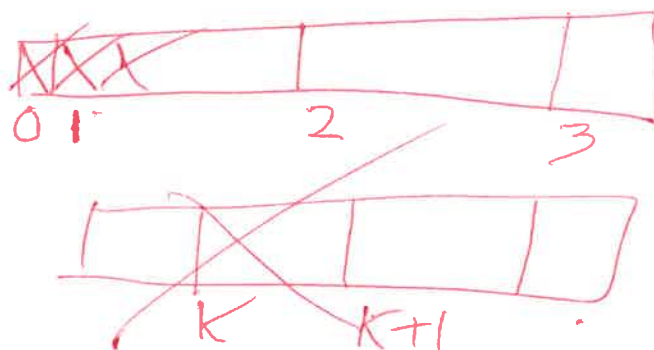
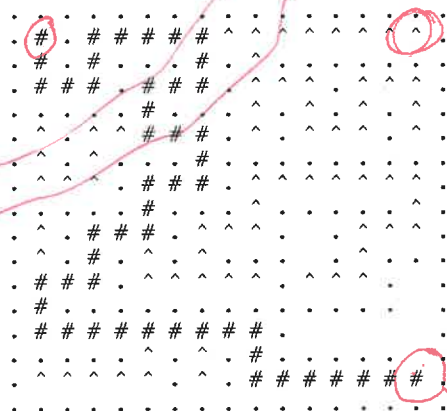


3. Message-based concurrent programming

(Aside: <https://dl.acm.org/doi/pdf/10.1145/3406678.3406690>)

4. Breadth-first search

a. Graphs (Notes 13)

b. Rat in a maze (<http://ranger.uta.edu/~weems/NOTES3318/VERMIN/ratBFSQueue.c>)

Demo of both versions of search:

<http://ranger.uta.edu/~weems/NOTES3318/VERMIN/rat.drag.html>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

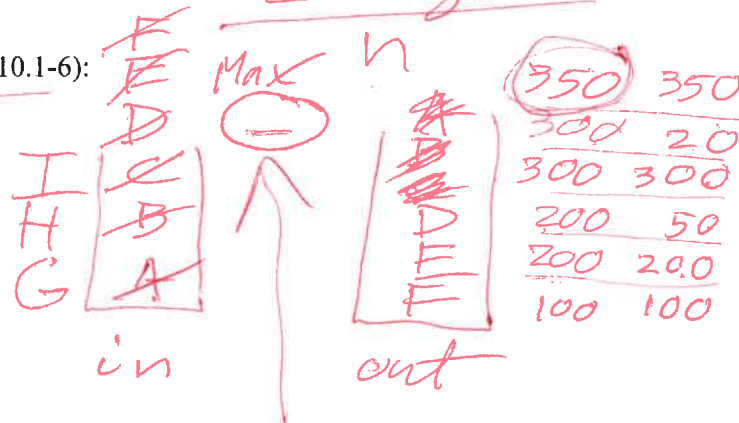
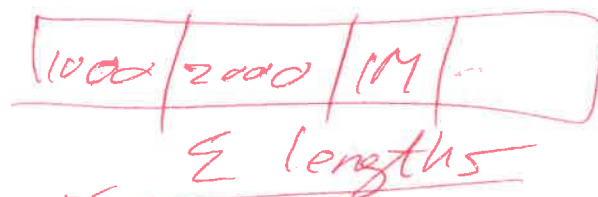
Initialize outStack

Enqueue(message, length):

if inStack.empty or length > inMaximum

inMaximum=length

inStack.push(message, length)



Deque:

```

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