CSE 2320 Notes 7: Stacks and Queues

(Last updated 9/26/06 8:56 PM)

CLRS, 10.1

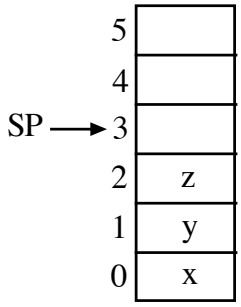
STACKS

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

PUSH POP TOP EMPTY

Policies Correspond to Code (usually $\Theta(1)$)

1. Direction of growth in array
2. What does *stack pointer* indicate?
   a. Next available element:



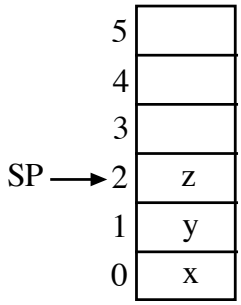
EMPTY: `return sp==0;`

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];`

Applications

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

Rat in a Maze

```
. . . . . . . . . . . . . . . . . . . .
. # . # # # # # ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ .
. # . # . . . # . ^ . . . . . . . . .
. # # # . # # # . ^ ^ ^ . ^ ^ ^ . . .
. . . . . # . . . ^ . ^ . ^ . ^ . . .
.   .     # # # . ^ . ^ ^ ^ . ^ . . .
.   .   . . . # . . . . . . . . . . .
.       . # # # . ^ ^ ^ ^ ^ ^ ^ ^ . .
. . . . . # . . . ^ . . . . . ^ . . .
. ^ . # # # . ^ ^ ^ . ^ . ^ ^ ^ . . .
. ^ . # . ^ . ^ . . . ^ . ^ . . . . .
. # # # . ^ ^ ^ ^ ^ . ^ ^ ^ .   . . .
. # . . . . . . . . . . . . .   . . .
. # # # # # # # # # .           . . .
. . . . .   .   . # . . . . .   . . .
.           .   . # # # # # # # . . .
. . . . . . . . . . . . . . . . . . .
```

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

Stack contains: Coordinates for each position on current path.

Array entries may change to reflect search status: 2=dead end (“^”), 3=solution (“#”).

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;
maze[row][col]=3;  // On final path
return 1;  // Propagate success through recursion
}

main()
{
readInput();
printf("Initial maze:\n");
printMaze();
if (DFS(startRow,startCol))
  printf("Success:\n");
else
  printf("Failure:\n");
printMaze();
}
```

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;
}
```

Evaluating/Converting Expressions

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 Prefix – Store operators and first operands on stack of pairs

```
while (unprocessed input tokens)
{
  get token;
  if (token is an operator)
    stack.push((token,NONE));
  else // token is an operand
  {
    while (1)
    {
      if (stack.empty())
        if (unprocessed input tokens)
          <error>
        else
          return token;

      (operator,operand)=stack.pop();
      if (operand==NONE)
      {
        stack.push((operator,token));
        break;
      }
      else
        token=result of evaluating (operand operator token);
    }
  }
}
<error>
```

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

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		

Infix – Can't evaluate directly, but can produce intermediate postfix:

Operands are immediately appended to postfix

Operators go to stack and are popped when a lower/equal precedence operator, right parentheses, or end-of-expression is encountered.

operator	precedence
+	1
-	1
*	2
/	2
unary -	3

```
while (unprocessed input tokens)
{
  get token;
  if (token is an operand)
    Append token to postfix output;
  else if (token is "(")
    stack.push(token);
  else if (token is ")")
    while((topToken=stack.pop())!="(")
      Append topToken to postfix output;
  else // token is an operator
  {
    precedence=findPrecedence(token);
    while (!stack.empty() && stack.top()!="(" &&
           findPrecedence(stack.top())>=precedence)
      Append stack.pop() to postfix output;
    stack.push(token);
  }
}
while (!stack.empty())
  Append stack.pop() to postfix output;
```

	Stack	Postfix
(:	(	
1:	(	1
+:	(+	1
2:	(+	1 2
):		1 2 +
*:	*	1 2 +
(:	* (	1 2 +
3:	* (	1 2 + 3
+:	* (+	1 2 + 3
1:	* (+	1 2 + 3 1
):	*	1 2 + 3 1 +
/:	/	1 2 + 3 1 + *
(:	/ (	1 2 + 3 1 + *
1:	/ (	1 2 + 3 1 + * 1
$+_1$:	/ ($+_1$	1 2 + 3 1 + * 1
1:	/ ($+_1$	1 2 + 3 1 + * 1 1
$+_2$:	/ ($+_2$	1 2 + 3 1 + * 1 1 $+_1$
1:	/ ($+_2$	1 2 + 3 1 + * 1 1 $+_1$ 1
):	/	1 2 + 3 1 + * 1 1 $+_1$ 1 $+_2$
EOE		1 2 + 3 1 + * 1 1 $+_1$ 1 $+_2$ /

Postfix may be processed immediately by using a second stack for just the operands.

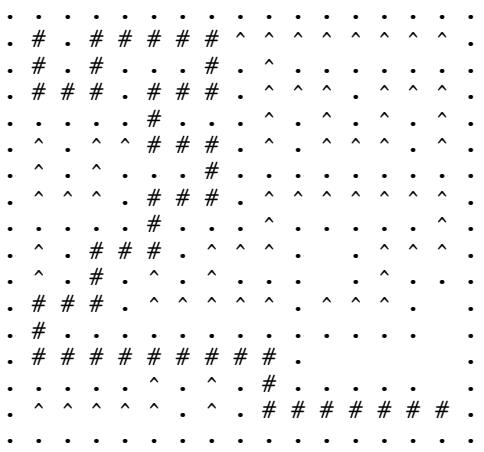
QUEUES

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

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

Applications

1. Data communications
2. Message-based concurrent programming
3. Event-interrupt handlers
4. Algorithms – breadth-first search
   a. Graphs
   b. Rat in a maze (`ratBFSqueue.c`)



Implementation using `A[0]...A[n-1]`

	Non-Reusable	Circular
Initialize	`tail=head=0;`	`tail=head=0;`
EMPTY	`return tail==head;`	`return tail==head;`

ENQUEUE(x)	`A[tail++]=x;`
	`if (tail==n)`
	`  < error >`
		`A[tail++]=x;`
		`if (tail==n)`
		`  tail=0;`
		`if (tail==head)`
		`  < confused >`

DEQUEUE	`if (tail==head)`
	`  < empty >`
	`return A[head++];`
		`if (tail==head)`
		`  < empty >`
		`temp=A[head++];`
		`if (head==n)`
		`  head=0;`
		`return temp;`