Department of Computer Science & Engineering
Indian Institute of Technology Kharagpur

Practice Sheet #13 with Solutions

Topic: Stack and Queue

Date: 04-04-2017

1. Consider the following sequence of push and pop operations on an initially empty stack S.
S = push(S,1);
S = push(S,2);
S = pop(S);
S = push(S,3);
S = push(S,4);
S = pop(S);
S = pop(S);
S = pop(S);

Which of the following is the correct order in which elements are popped?

(a) 1, 2, 3, 4
(b) 2, 1, 3, 4
(c) 2, 3, 4, 1
(d) 2, 4, 3, 1
Ans. d

2. Consider the following sequence of push and pop operations on an initially empty stack S.
S = push(S,1);
S = pop(S);
S = push(S,2);
S = push(S,3);
S = pop(S);
S = push(S,4);
S = pop(S);
S = pop(S);

Which of the following is the correct order in which elements are popped?

(a) 1, 2, 3, 4
(b) 1, 3, 2, 4
(c) 1, 3, 4, 2
(d) 4, 3, 2, 1
Ans. c

3. (a) Give a suitable **typedef** for representing a stack of integers using *either* an array *or* a linked list.

(b) Write C functions for implementing the following stack operations. If you have opted for the linked list representation, clearly indicate whether a dummy header node is used.
init – which constructs and returns an empty stack,
empty – which returns a value indicating whether a given stack is empty or not,
push – which pushes a given integer on to a given stack,
top – which returns the top element of a given stack,
delete – which deletes the top element of a given stack.
(c) Write an *iterative* C function which takes an unsigned integer and prints its representation to the base 5 *using a stack*; the function should use the data type defined by you for representing a stack and only the functions of Part (b).

Ans. ??

4. (a) Give a suitable **typedef** with a brief explanation for representing a job which comprises a positive integer identification number and another positive integer for job size given as the time needed for completing the job.

(b) Give a suitable **typedef** with a brief explanation for representing a queue of jobs using *either* an array *or* a linked list.

(c) Give prototypes for the following functions on a queue of jobs (*no need* to write the function bodies).
empty – which returns a value indicating whether a given queue is empty or not,
enqueue – which puts a given job in a given queue,
front – which returns the job from the front of a given queue if the queue is not empty; otherwise it returns a job with a negative identification number,
dequeue – which deletes the front of a given queue.

(d) One of the ways to process a collection of jobs, called *Round-Robin strategy*, is as follows: There is a predetermined period of time, called *time slice*. Each job is taken from the queue; if its time is less than or equal to the *time slice*, then it is processed until completion; otherwise, it is processed for a period equal to the *time slice*, its remaining time requirement is accordingly updated and then the job is put back into the queue for taking up in future after processing other jobs in a similar manner.
Write a C function which takes a queue of jobs and a time slice and processes the jobs using the Round- Robin strategy until the completion of each of them. The function should use the functions of Part (c).
Ans. ??

5. You have an $m \times n$ maze of rooms. Each adjacent pair of rooms has a door that allows passage between the rooms. At some point of time some of the doors are locked, the rest are open. A mouse sits at room number (s, t), and there is fabulous food for the mouse at room number (u, v). Your task is to determine whether there exists a route for the mouse from room (s, t) to room (u, v) through the open doors. The idea is to start a search at room no (s, t), then investigate adjacent rooms (s1, t1), . . . , (s_k, t_k) that can be reached from (s, t), and then those adjacent rooms that can be reached from each (s_i, t_i), and so on.
The rooms are numbered (i, j), where i grows horizontally (along the x direction), and j grows in the vertical direction (along the y axis). The four walls of the (i, j)-th room are numbered as shown to the right of the maze. If a horizontal or vertical wall has an open door, we indicate this by the value 1; otherwise, we use the value 0. That is, $H_{i,j} = 1$ if the horizontal door connecting the rooms $(i, j - 1)$ and (i, j) is open; $H_{i,j} = 0$ otherwise. Similarly, $V_{i,j}$ is 1 or 0 depending upon whether the vertical door between the rooms $(i - 1, j)$ and (i, j) is open or not.
This numbering scheme also applies to the walls of the boundary of the maze. However, we assume that the mouse cannot go out of the house, that is, all the walls on the boundary are closed. H is available as an $m \times (n+1)$ array, whereas V is available as an $(m+1) \times n$ array.
We use a stack to implement the search (from (s, t) to (u, v) given the arrays H and V). Since there is no need to revisit a room during the search, we maintain an $m \times n$ array of flags in order to keep track of the rooms that are visited—the value 1 means "visited", and 0 means "not visited so far". The stack contains a list of rooms, that is, it is capable of storing pairs of indices (i, j). The stack ADT is supplied as follows.
S = init();   /* Create an empty stack */

S = push(S,i,j);   /* Push the pair (i, j) **to the top of the stack S */**
S = pop(S);   /* Pop an element (a pair) from the top of the stack */
(i,j) = top(S);   /* Return the top (i, j) **of the stack S */**
isEmpty(S);   /* Returns true if and only if the stack S is empty */

(a) Fill out the following **main()** function that pushes the source room (s, t) to an initially empty stack, and subsequently calls the search function with appropriate arguments.

```
main ()
{
stack S;
int m, n, s, t, u, v, H[MAX][MAX], V[MAX][MAX], visited[MAX][MAX], status;
/* Assume that m, n, s, t, u, v, H[][], V[][] and visited[][] are
appropriately initialized here. You do not have to write these. */
. . .
S = ______________________; /* Initialize the stack S */
S = push( S, ____________________,______________________ ); /* Push source room to
stack */
visited[ ___________][ _______________] = 1; /* Mark source room as visited */
status = search(m,n,s,t,u,v,H,V,visited,S); /* Call the search function */
printf("Search %s\n", (status == 1) ? "successful" : "unsuccessful");
}
```

Ans. ??

(b) Complete the search function whose skeleton is provided below. The function returns 1 if the destination node is ever reached. Otherwise, it returns 0.

```
int search ( int m, int n, int s, int t, int u, int v,
int H[][MAX], int V[][MAX], int visited[][MAX], stack S )
/* m × n is the size of the maze, (s, t) is the source node,
(u, v) is the destination node, H[][] and V[][] are door arrays,
visited[][] is the array to store which nodes are visited so far,
and S is the stack to be used in the search. */
{
pair room; /* pair is a structure of two integer values x and y */
int i, j;
/* So long as the stack is not empty */
while (____________________ ) {
room =___________________ ; /* Read the top element from the stack */
i = room.x; j = room.y ; /* Retrieve x and y coordinates of room */
/* Delete the top from the stack */
_______________________________/* If (i, j) is the destination node, return success */
if ( _________________) return___________________________ ;
/* Otherwise, look at the four adjacent rooms one by one */
/* Left room: If left door is open and left room is not yet visited */
if (_______________________ ) {
/* Push left room to the stack and mark left room as visited */
S = push(S,________________ , ___________________);
visited[ __________________][___________________ ] = ___________________;
}
/* Analogously process the adjacent room to the right */
___________________________________________
/* Process the adjacent room at the bottom */
_______________________________________
/* Process the adjacent room at the top */
______________________________________________
}
return __________________________;
}
```
Ans. ??

6. In a doubly linked list, each node has a link in the forward direction and another link in the backward direction. The forward link of the last node and the backward link in the first node of the list are **NULL**.

```
typedef struct _node {
int data;
struct _node *flink; /* forward link pointing to the next node */
struct _node *blink; /* backward link pointing to the previous node */
} node;
```

A double-ended queue is an ordered list in which insertion and deletion can occur at both the ends. Let us represent a double-ended queue by a doubly linked list as follows. This is an array of two pointers, the first pointer (at index zero) pointing to the first node of the linked list and the second (at index one) the last node of the list. In an empty queue **Q**, both the pointers **Q[0]** and **Q[1]** are **NULL**.

```
typedef node *d_e_queue[2];
```

(a) Complete the following insert function which takes three arguments: a double-ended queue **Q**, the integer **a** to be inserted, and the **end** (zero or one) where insertion would take place.

```
void insert ( d_e_queue Q, int a, int end )
{
node *p;
p = _________________________ ; /* Allocate memory */
p -> data = a;
if ( __________________ ) { /* Insertion in an empty queue */
______________________________ /* Set the links of p */
____________________________ /* Set the pointers in Q */
return;
}
if (end == 0) { /* Insertion at the beginning of the list */
_____________________________ /* Set the links of p */
Q[0] -> blink = p; __________________ /* Set Q[0] */
} else { /* Insertion at the end of the list */
_____________________________ /* Set the links of p */
_______________________________ /* Adjust Q[1] */
}
}
```

Ans.

```
(node *)malloc(sizeof(node))
Q[0] == NULL
p -> flink = p -> blink = NULL;
Q[0] = Q[1] = p;
p -> flink = Q[0]; p -> blink = NULL;
Q[0] = p;
p -> blink = Q[1]; p -> flink = NULL;
Q[1] -> flink = p; Q[1] = p;
```

(b) Complete the following deletion function that takes two arguments: a double-ended queue **Q** and an **end** (zero/one) at which deletion occurs. Free the node being deleted.

```
void delete ( d_e_queue Q, int end )
{
if (Q[0] == NULL) return; /* Deletion from an empty queue */
if (Q[0] == Q[1]) { /* Deletion from a queue with one element */
_____________________________________
return;
}
if (end == 0) { /* Deletion at the beginning of the list */
___________________________________________
} else { /* Deletion at the end of the list */
_______________________________________________
}
}
```

Ans.

```
free(Q[0]); Q[0] = Q[1] = NULL;
Q[0] = Q[0] -> flink; free(Q[0] -> blink); Q[0] -> blink = NULL;
Q[1] = Q[1] -> blink; free(Q[1] -> flink); Q[1] -> flink = NULL;
```

7. You start with an empty stack S, and then perform n push and n pop operations on S such that: You push the integers 1;2;3; : : : ;n in that order. Assume that the stack has enough memory to store n elements, that is, no push operation fails. You never pop from an empty stack, that is, no pop operation fails, so after the n push and the n pop operations S becomes empty again. Immediately before each pop operation, you print the top of the stack, that is, the element which is going to be popped. The above rules indicate that the integers 1;2;3; : : : ;n are printed in some order. The printed sequence is said to be realized by a stack.
For example, let n = 5. The permutation 2;4;3;5;1 can be realized by the following sequence of push and pop operations (the print statements are not shown): push(1), push(2), pop(), push(3), push(4), pop(), pop(), push(5), pop(), pop(). Observe that this example sequence satisfies the specified rules. We push the integers in the order 1;2;3;4; 5. We perform the same number of pop operations and never pop from an empty stack. Not every permutation of 1;2;3; : : : ;n can be realized by a stack. For example, convince yourself that for n = 5 the permutation 2;4;1;3;5 cannot be realized by a stack.
The following function takes as input n and an array **seq[]** storing a permutation of 1;2;3; : : : ;n, and checks whether the input sequence can be realized by a stack. There are no external **push()** and **pop()** functions. We instead use a local array **S[]** as the stack, and write the codes for push and pop explicitly in the function body. The variable **top** stores the index of the stack top in **S[]**. The variable **a** stores what element is to be inserted in the stack in the next push operation. Finally, the variable **i** stands for how many integers are printed (that is, how many pop operations are carried out). The function also prints the maximum-length prefix of the input sequence that can be realized using the stack. Complete the function.

```
void check ( int *seq, int n )
{
int a, top, i, *S;
S = ________________________ ; /* Allocate memory */
a = 1; top = -1; i = 0; /* Initialize */
/* Repeat the following loop until explicitly broken in the body */
while (1) {
/* First check whether it is necessary to print and pop now */
if ( _________________________________________ ) {
printf("%d ", ____________________ );
_______________ ; /* Pop from S */
_____________________ ; /* Another integer printed */
```

} else
/* Then check whether it is allowed to push another element to S */
if (____________________) {
____________________ ; /* Push a to S */
_____________________ ; /* Prepare for the next push */
} else ______________________ ;
}
if (______________________________)
printf("+++ The input sequence can be realized by a stack...\n");
else
printf("+++ The input sequence cannot be realized by a stack...\n");
_________________________________ ; /* Clean locally allocated dynamic memory */
}
Ans.
(int *)malloc(n * sizeof(int))
(top >= 0) && (seq[i] == S[top])
S[top]
--top
++i
a <= n
S[++top] = a
++a
break ;
i == n
free(S)

8. Following stack operations are performed on an initially empty stack:
push(5); push(1); push(4); push(3); pop( ); push(2); pop( ); pop( ); push(8); push(7); pop( );
What is the value stored in *top of the stack* after the above operations?
Ans. 8

9. Following queue operations are performed on an initially empty queue:
enqueue(6); enqueue(12); enqueue(13); dequeue( ); dequeue( ); enqueue(19); enqueue(21); enqueue(22); dequeue( ); enqueue(20);
What is the value of the *queue front* after the above operations?
Ans. 19

10. (a) Consider the stack data type: struct stack{ int data[MAXSIZE]; int top;};
Complete the following functions to (i) *create* an empty stack, (ii) check if a stack *is empty*, (iii) *push* into a non-full stack, and (iv) *pop* from a non-empty stack. We define an empty stack to have *top = -1*.
(i)
struct stack create(){
struct stack s;
s.top = ____________;
return s; }
Ans. -1

(ii)
int isempty(struct stack *s){
return (___________) ;

```
}
```
Ans. s->top == -1

(iii)
```
void push(struct stack *s, int x){
____________________;
_________________________;
}
```
Ans. s->top ++, s->data[s->top] = x

(iv)
```
int pop(struct stack *s){
int x =___________________;
s-> top --;
return x; }
```
Ans. s->data[s->top]

(b) A decimal number is converted to a binary number by continually divide-by-2 to give a result and a remainder of either a “1” or a “0” until the final result equals zero. For example, the binary equivalent of the decimal number 9 is 1001. Complete the program below which uses a stack to print the binary equivalent of the decimal number *n*. You can use the stack functions defined above.
```
struct stack create(){
struct stack s;
s.top = -1;
return s; }
int isempty(struct stack *s){
return (s->top == -1) ;
}
#include <stdio.h>
#define MAXSIZE 128
void main(){
struct stack s; int n;
scanf(“%d”, &n);
s = create();
while(____________){
push(______________); /* divide-by-2, and store remainder in stack */
n =______________; /* continue with result of division */
}
printf(“The binary number is: \n“); /*get binary number from stack */
while(__________________) printf(“%d”,________________);
}
```
Ans.
```
n != 0
&s, n%2
n/2
!isempty(&s)
pop(&s)
```

11. Write C program segments/statements to serve the following purposes.
(a) Define a structure for implementing a stack of integers using an array.
(b) Write a function for creating a stack.

(c) Write a function for pushing an integer into the stack.
Ans. ??

12. Let S be a stack of size n ≥ 1. Starting with the empty stack, suppose we push the first n natural numbers in sequence, and then perform n pop operations. Assume that Push and pop operation take X seconds each, and Y seconds elapse between the end of one such stack operation and the start of the next operation. For m ≥ 1, define the stack-life of m as the time elapsed from the end of Push(m) to the start of the pop operation that removes m from S. The average stack-life of an element of this stack is:

(a) n (X + Y)
(b) 3Y + 2X
(c) n (X + Y) - X
(d) Y + 2X
Ans. c

13. The best data structure to check whether an arithmetic expression has balanced parentheses is a:
(a) queue
(b) stack
(c) tree
(d) List
Ans. b

14. A circularly linked list is used to represent a Queue. A single variable p is used to access the Queue. To which node should p point such that both the operations enQueue and deQueue can be performed in constant time?
(a) rear node
(b) front node
(c) not possible with a single pointer
(d) node next to front
Ans. a

15. An implementation of a queue Q, using two stacks S1 and S2, is given below:

```
void insert(Q, x) {
  push (S1, x);
}

void delete(Q){
  if(stack-empty(S2)) then
    if(stack-empty(S1)) then {
        print("Q is empty");
        return;
    }
    else while (!(stack-empty(S1))){
        x=pop(S1);
        push(S2,x);
    }
  x=pop(S2);
}
```

Let n insert and m (<=n) delete operations be performed in an arbitrary order on an empty queue Q. Let x and y be the number of push and pop operations performed respectively in the process. Which one of the following is true for all m and n?
(a) n+m <= x < 2n and 2m <= y <= n+m
(b) n+m <= x < 2n and 2m<= y <= 2n
(c) 2m <= x < 2n and 2m <= y <= n+m
(d) 2m <= x <2n and 2m <= y <= 2n

Ans. a

16. Consider the following operation along with Enqueue and Dequeue operations on queues, where k is a global parameter.

```
MultiDequeue(Q){
  m = k
  while (Q is not empty and m  > 0) {
    Dequeue(Q)
    m = m - 1
  }
}
```

What is the worst case time complexity of a sequence of n MultiDequeue() operations on an initially empty queue?

(a) \Theta(n)
(b) \Theta(n + k)
(c) \Theta(nk)
(d) \Theta(n^2)
Ans. a

--*--