1 Stacks

A stack is an abstract data type which is defined as an ordered collection of objects such that:

1. You can insert or remove one object at at time
2. When you remove an object, you always remove the object that has been inserted most recently

A good example from everyday life is a stack of trays in a fast-food restaurant. When you add a tray to the stack it usually goes to the top of the stack and when you get a tray from the stack you usually get the most recently placed tray.

The basic operations on a stack are inserting an element, removing and element and checking if a stack is empty. The operation of inserting an element in a stack is usually called ”pushing an element in the stack” and the operation of removing an element from a stack is usually called ”popping and element from the stack”. We will therefore call the functions performing these operations in our examples *push()*, *pop()* and *empty()*.

A stack can be implemented in a computer program in a number of ways. The most common implementations are through a linked list or an array. Both implementations have their advantages and disadvantages.

1.1 Linked List Implementation of a Stack

A stack can be implemented very easily with a linked list by applying the following restrictions:

1. When you insert an element, it is always inserted at the beginning of the list
2. When you remove an element, it is always the first element in the list

Exercise 1: Implement a stack of integers in C based on the following definitions:

```
struct node
{
   int data;
   struct node * next;
};

typedef struct node * stack;

/* initialize a stack */
void initstack(stack *s);
```

```
/* function to check if a stack is empty */
int empty(stack s);

/* pushes a value on the stack, prints an error */
/* message if out of memory */
void push(stack * s, int value);

/* pops a value from a stack, prints an error message */
/* if the stack is empty and returns -1 */
int pop(stack * s);
```

Exercise 2: Trace the execution of the following code showing the state of the stack at each point:

```
   initstack(&s);
   push(&s, 1);
   push(&s, 2);
   printf("%d\n", pop(&s));
   push(&s, 3);
   printf("%d\n", pop(&s));
   printf("%d\n", pop(&s));
```

1.2 Array Implementation of a Stack

When the maximum size of a stack is known in advance it makes sense to implement a stack by an array. The advantage of such an implementation is that it is much faster that a linked list implementation since there is no dynamic memory allocation involved (dynamic memory allocation is computationally expensive). You can define a stack to be a structure containing an array of integers and an integer variable (*top*) which is the index of the top entry in the array. *top = -1* would indicate an empty stack;

Exercise 3: Implement a stack of integers in C based on the following definitions:

```
#define maxstack 100

typedef struct
{
   int   data[maxstack];
   int   top;
} stack;

/* initialize a stack */
void initstack(stack *s);

/* function to check if a stack is empty */
int empty(stack * s);

/* function to check if a stack is full */
```

```
int full(stack * s);

/* pushes a value on the stack, prints an error */
/* message if out of memory */
void push(stack * s, int value);

/* pops a value from a stack, prints an error message */
/* if the stack is empty and returns -1 */
int pop(stack * s);
```

Exercise 4: Repeat exercise 2 this time with an array implementation of a stack.

1.3 Stack Examples

Exercise 5: Implement an Postfix expression evaluator based on the following pseudo-code given in class:

```
initialize stack

while there is something to read
   read input
   if input is an operand push on the stack
   if input is an operator pop two operands from ...
      ... the stack, apply operator and push the result back

pop final result from stack and print it
```

Exercise 6: Implement an Infix to Postfix translator based on the following pseudo-code:

```
initialize stack

while there is something to read
   read input
   if input is an operand print it
   if input is a '(' then push it on the stack
   if input is a ')' then
      repeatedly pop from stack and print until ...
         ... you see a '(', do not print the '('
   if input is an operator
        while stack not empty, and top of stack is not a '(', and ...
               ... priority of top of stack >= priority of input
            pop from stack and print
        push input on the stack
while stack not empty
     pop from stack and print
```

Exercise 7: In your second homework, you implemented the minesweeper game with a recursive *uncover()* function. Here is an example of such a function:

```
struct cell
{
   int mine;         /* 1 indicates a mine, 0 no mine */
   int visible;      /* 1 indicates visible (uncovered), 0 indicates invisible) */
   int adjacent;      /* number of adjacent mines (0-8) */
};

struct cell board[9][9];

/* a function to check if x and y are valid coordinates */
int valid(int x, int y)
{
   return ((x >= 0) && (y >= 0) && (x < 9) && (y < 9));
}

void uncover(struct cell board[9][9], int x, int y)
{
   if (valid(x,y) && (board[x][y].visible == 0)) {
      board[x][y].visible = 1;
      if (board[x][y].adjacent == 0) {
            uncover(board, x+1, y+1);
            uncover(board, x  , y+1);
            uncover(board, x-1, y+1);
            uncover(board, x+1, y  );
            uncover(board, x-1, y  );
            uncover(board, x+1, y-1);
            uncover(board, x  , y-1);
            uncover(board, x-1, y-1);
         }
      }
   }
}
```

Change the recursive *uncover()* function into a non-recursive function that uses a stack.

2 Queues

A queue is an abstract data type which is defined as an ordered collection of objects such that:

1. You can insert or remove one object at at time
2. When you remove an object, you always remove the object that has been inserted least recently

A good example from everyday life is a queue of people in a fast-food restaurant waiting waiting in line to give their orders. When a person is being served it is usually the one that came earliest and when a new person arrives at the restaurant he usually goes to the end of the queue and will be served only after everybody that came before him is served.

The basic operations on a queue are inserting an element, removing and element and checking if a queue is empty. The operation of inserting an element in a queue is usually called ”enqueueing an element in the queue” and the operation of removing an element from a queue is usually called ”dequeueing and element from the queue”. We will therefore call the functions performing these operations in our examples *enqueue()*, *dequeue()* and *empty()*.

A queue can be implemented in a computer program in a number of ways. The most common implementations are through a linked list or an array. Both implementations have their advantages and disadvantages.

2.1 Linked List Implementation of a Queue

A queue can be implemented very easily with a linked list by applying the following restrictions:

1. When you insert an element, it is always inserted at the end of the list
2. When you remove an element, it is always the first element in the list that is removed

Exercise 8: Implement a queue of integers in C based on the following definitions:

```
struct node
{
   int data;
   struct node * next;
};

typedef struct node * nodeptr;
typedef struct {
   nodeptr head;
   nodeptr tail;
} queue;

/* initialize a queue */
void initqueue(queue *q);

/* function to check if a queue is empty */
int empty(queue q);

/* enqueues a value in the queue, prints an error */
/* message if out of memory */
void enqueue(queue * q, int value);

/* dequeues a value from a queue, prints an error message */
/* if the queue is empty and returns -1 */
int dequeue(queue * q);
```

2.2 Array Implementation

When the maximum size of a queue is known in advance it makes sense to implement it with a fixed-size array. The advantage of such an implementation is that it is much faster that a linked list implementation since there is no dynamic memory allocation involved (dynamic memory allocation is computationally expensive). You can implement a queue by defining it as a structure containing an array and two extra integer variables (*head* and *count*). *head* would be an array index to the beginning of the queue and *count* will be the number of elements in the queue. Note that the index will wrap, so that when you reach the end of the array you continue from the beginning.

Exercise 9: Implement a queue of integers in C based on the following definitions:

```
#define maxqueue 3

typedef struct {
   int   data[maxqueue];
   int   head;
   int   count;
} queue;

/* initialize a queue */
void initqueue(queue *q);

/* function to check if a queue is empty */
int empty(queue * q);

/* function to check if a queue is full */
int full(queue * q);

/* enqueues a value in the queue, prints an error */
/* message if queue is full */
void enqueue(queue * q, int value);

/* dequeues a value from a queue, prints an error message */
/* if the queue is empty and returns -1 */
int dequeue(queue * q);
```

2.3 Queue Examples

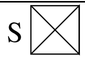
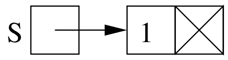
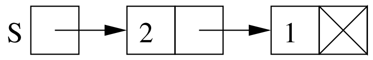
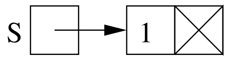
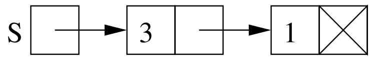
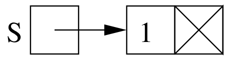
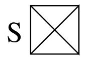
Exercise 10: Change the non-recursive *uncover()* function that used stacks in Exercise 7 into a non-recursive function that uses queues. The queue-based *uncover()* function will uncover the board in a different order - starting from the user-selected position and spreading uniformly around it.

3 Solutions

Solution to Exercise 1:

```
/*********************************************************************/
/* Linked List implementation of a stack                             */
/*********************************************************************/

#include <stdio.h>
#include <stdlib.h>

struct node
{
   int data;
   struct node * next;
};

typedef struct node * stack;

/* initialize a stack */
void initstack(stack *s)
{
   *s = NULL;
}

/* function to check if a stack is empty */
int empty(stack s)
{
   return (s == NULL);
}

/* pushes a value on the stack, prints an error */
/* message if out of memory */
void push(stack * s, int value)
{
   stack temp = (stack) malloc(sizeof(struct node));
   if (temp == NULL) {
      printf("\nERROR: Out of memory\n");
   } else {
      temp->data = value;
      temp->next = *s;
      *s = temp;
   }
}

/* pops a value from a stack, prints an error message */
/* if the stack is empty and returns -1 */
int pop(stack * s)
{
   stack temp;
   int value;

   if (empty(*s)) {
      printf("\nERROR: Stack underflow\n");
      return -1;
   } else {
      temp = (*s);
      (*s) = (*s)->next;
      value = temp->data;
      free(temp);
      return value;
   }
}
```

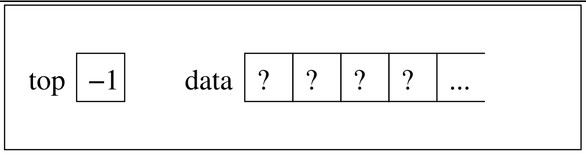
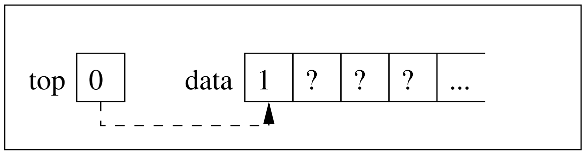
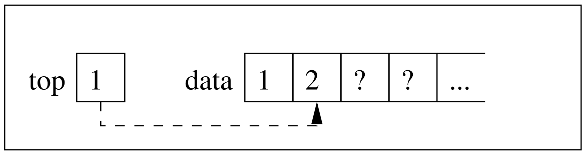
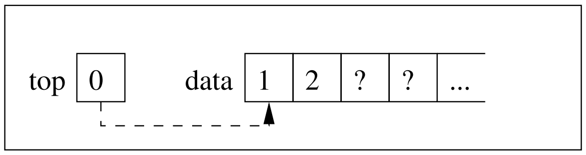
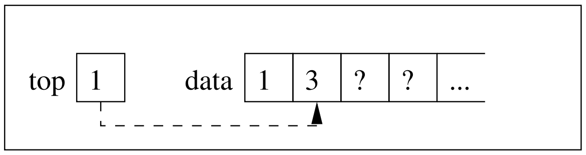
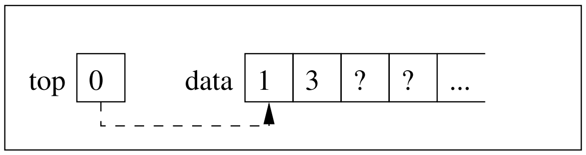
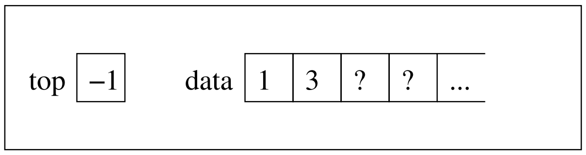
Solution to Exercise 2:

Operation	State after execution of Operation
`nitstack(&s);`	
`push(&s, 1);`	
`push(&s, 2);`	
`printf("%d\n", pop(&s));`	
`push(&s, 3);`	
`printf("%d\n", pop(&s));`	
`printf("%d\n", pop(&s));`	

Solution to Exercise 3:

```
/**********************************************************************/
/* Array implementation of a stack                                    */
/**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define maxstack 100

typedef struct
{
   int   data[maxstack];
   int   top;
} stack;

/* initialize a stack */
void initstack(stack *s)
{
   s->top = -1;
}

/* function to check if a stack is empty */
int empty(stack * s)
{
   return (s->top == -1);
}

/* function to check if a stack is full */
int full(stack * s)
{
   return (s->top == (maxstack-1));
}

/* pushes a value on the stack, prints an error */
/* message if out of memory */
void push(stack * s, int value)
{
   if (full(s)) {
      printf("\nERROR: stack overflow\n");
   } else
      s->data[++(s->top)] = value;
}

/* pops a value from a stack, prints an error message */
/* if the stack is empty and returns -1 */
int pop(stack * s)
{
   if (empty(s)) {
      printf("\nERROR: stack underflow\n");
      return -1;
   } else
      return s->data[(s->top)--];
}
```

Solution to Exercise 4:

Operation	State after execution of Operation
`initstack(&s);`	
`push(&s, 1);`	
`push(&s, 2);`	
`printf("%d\n", pop(&s));`	
`push(&s, 3);`	
`printf("%d\n", pop(&s));`	
`printf("%d\n", pop(&s));`	

Solution to Exercise 5:

```
/**********************************************************************/
/* Evaluates an Postfix expression using a stack                      */
/**********************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

... stack definitions go here ...

/* reads a postfix expression from the standard input */
/* and calculates the result, expression is terminated */
/* by "end" or End-Of-File condition. Assumes a valid */
/* input */
int main(int argc, char *argv[])
{
   stack s;
   char buf[128];
   int a, b;

   initstack(&s);

   while ( scanf("%s", buf) != EOF) {
      if (strcmp(buf, "+") == 0) {
         a = pop(&s);
         b = pop(&s);
         push(&s, b+a);
      } else if (strcmp(buf, "-") == 0) {
         a = pop(&s);
         b = pop(&s);
         push(&s, b-a);
      } else if (strcmp(buf, "*") == 0) {
         a = pop(&s);
         b = pop(&s);
         push(&s, b*a);
      } else if (strcmp(buf, "/") == 0) {
         a = pop(&s);
         b = pop(&s);
         push(&s, b/a);
      } else if (strcmp(buf, "end") == 0)
         break;
      else push(&s, atoi(buf));
   }

   printf("result = %d\n", pop(&s));
   return 0;
}
```

Solution to Exercise 6:

```
/**********************************************************************/
/* Infix to Postfix converter using stacks                            */
/**********************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

... linked list implementation of a stack goes here ...

/* returns what is on the top of the stack without actually removing it */
int peek(stack * s)
{
   if (empty(*s)) {
      printf("\nERROR: Stack underflow\n");
      return -1;
   } else
      return (*s)->data;
}

/* returns the precedence of an operator */
int precedence(char c)
{
   if ((c=='-') || (c=='+')) return 1;
   if ((c=='*') || (c=='/')) return 2;
   return 3; /* i.e. ^ */
}

/* checks if a character is an operator */
int is_operator(char c)
{
   return ( (c=='+') || (c=='-') || (c=='*') || (c=='/') || (c=='^'));
}

int main(int argc, char *argv[])
{
   stack s;
   char ch, op;

   initstack(&s);

   while ((ch = getchar()) != '\n') {
      if (isspace(ch)) /* skip white space */
         continue;

      if (ch == '(') {
         push(&s, ch);
      } else if (ch == ')') {
         while ((op = pop(&s)) != '(')
            printf("%c ", op);
      } else if (is_operator(ch)) {
         while ( !empty(s) &&
                 (peek(&s)!='(') &&
                 (precedence(ch) <= precedence(peek(&s)))) {
            printf("%c ", pop(&s));
         }
         push(&s, ch);
      } else
         printf("%c ", ch);
   }

   while (!empty(s)) printf("%c ", pop(&s));

   printf("\n");
   return 0;
}
```

Solution to Exercise 7:

```
/**********************************************************************/
/* Uncovering Mines with a stack                                      */
/**********************************************************************/

void uncover(struct cell board[9][9], int x, int y)
{
   stack s;

   initstack(&s);
   push(&s, x);
   push(&s, y);
   while (!empty(s)) {
      y = pop(&s); /* note the reverse order when popping */
      x = pop(&s); /* coordinates from the stack */
      if (valid(x,y) && (board[x][y].visible == 0)) {
         board[x][y].visible = 1;
         if (board[x][y].adjacent == 0) {
            push(&s, x+1, y+1);
            push(&s, x  , y+1);
            push(&s, x-1, y+1);
            push(&s, x+1, y  );
            push(&s, x-1, y  );
            push(&s, x+1, y-1);
            push(&s, x  , y-1);
            push(&s, x-1, y-1);
         }
      }
   }
}
```

Solution to Exercise 8:

```
/*********************************************************************/
/* Linked List implementation of a queue                             */
/*********************************************************************/
#include <stdio.h>
#include <stdlib.h>

struct node {
   int data;
   struct node * next;
};

typedef struct node * nodeptr;

typedef struct {
   nodeptr head;
   nodeptr tail;
} queue;

/* initialize a queue */
void initqueue(queue *q)
{
   q->head = NULL;
}

/* function to check if a queue is empty */
int empty(queue q)
{
   return (q.head == NULL);
}

/* enqueues a value in the queue, prints an error message if out of memory */
void enqueue(queue * q, int value)
{
   nodeptr temp = (nodeptr) malloc(sizeof(struct node));
   if (temp == NULL) {
      printf("\nERROR: Out of memory\n");
   } else {
      temp->data = value;
      temp->next = NULL;
      if (empty(*q)) {
         q->head = q->tail = temp;
      } else {
         q->tail->next = temp;
         q->tail = q->tail->next;
      }
   }
}

/* dequeues a value from a queue, prints an error message if the queue is empty and returns -1 */
int dequeue(queue * q)
{
   nodeptr temp;
   int value;
   if (empty(*q)) {
      printf("\nERROR: Stack underflow\n");
      return -1;
   } else {
      temp = q->head;
      q->head = q->head->next;
      value = temp->data;
      free(temp);
      return value;
   }
}
```

Solution to Exercise 9:

```
/**********************************************************************/
/* Array implementation of a queue                                    */
/**********************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define maxqueue 3

typedef struct {
   int   data[maxqueue];
   int   head;
   int   count;
} queue;

/* initialize a queue */
void initqueue(queue *q)
{
   q->head = 0;
   q->count = 0;
}

/* function to check if a queue is empty */
int empty(queue * q)
{
   return (q->count == 0);
}

/* function to check if a queue is full */
int full(queue * q)
{
   return (q->count >= maxqueue);
}

/* enqueues a value in the queue, prints an error */
/* message if queue is full */
void enqueue(queue * q, int value)
{
   if (full(q)) {
      printf("\nERROR: queue is full\n");
   } else {
      q->data[ (q->head + q->count) % maxqueue ] = value;
      q->count++;
   }
}

/* dequeues a value from a queue, prints an error message */
/* if the queue is empty and returns -1 */
int dequeue(queue * q)
{
   int value;

   if (empty(q)) {
      printf("\nERROR: queue is empty\n");
      return -1;
   } else {
      value = q->data[q->head];
      q->head = (q->head + 1) % maxqueue;
      q->count--;
      return value;
   }
}
```

Solution to Exercise 10:

```
/*********************************************************************/
/* Uncovering Mines with a queue                                     */
/*********************************************************************/

void uncover(struct cell board[9][9], int x, int y)
{
   queue q;

   initqueue(&q);
   enqueue(&q, x);
   enqueue(&q, y);

   while (!empty(q)) {
      x = dequeue(&q);
      y = dequeue(&q);
      if (valid(x,y) && (board[x][y].visible == 0)) {
         board[x][y].visible = 1;
         if (board[x][y].adjacent == 0) {
            enqueue(&q, x+1); enqueue(&q, y+1);
            enqueue(&q, x  ); enqueue(&q, y+1);
            enqueue(&q, x-1); enqueue(&q, y+1);
            enqueue(&q, x+1); enqueue(&q, y  );
            enqueue(&q, x-1); enqueue(&q, y  );
            enqueue(&q, x+1); enqueue(&q, y-1);
            enqueue(&q, x  ); enqueue(&q, y-1);
            enqueue(&q, x-1); enqueue(&q, y-1);
         }
      }
   }
}
```