|  |  |
| --- | --- |
| **CSCI 2100E Spring 2020** | Student Name: \_Tam Yi Ki\_\_ Student Id: \_\_1155126127\_\_ |

# Lab Exercise #3

*Lab exercises are always due 2 weeks after day of the lab. Submitting this lab exercise gives you 10% bonus to your next test if you are below mean.*

*Please fill in the lab sheet and submit the completed Word doc file to blackboard.* *Places you need to fill in or work on are marked in red.*

## Demo

Here we see a Stack ADT implemented using array. We would like the stack to be usable for different max sizes though, so we need to use dynamic memory allocation for our array as well.

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  typedef struct {  int \*data; // stack data, we assume integer for simplicity  int top; // top of the stack  int maxSize; // max size of the stack  } Stack;  void StackInit(Stack\* stack, int size) {  // this function initializes a stack for first use  printf("Initializing stack to hold %d integers...\n",size);  stack->maxSize = size;  stack->top = -1;  stack->data = (int\*) malloc(sizeof(int) \* size);  }  void StackPush(Stack\* stack, int data) {  if (stack->top >= stack->maxSize - 1) {  printf("Stack already full, returning\n");  return;  }  printf("Pushing %d\n",data);    stack->top++;  stack->data[stack->top] = data;  }  int StackPeek(Stack\* stack) {    // return what's on the top without removing    // TODO: implement  }  int StackPop(Stack\* stack) {    // return what's on the top and remove it    // TODO: implement  }  int StackDestroy(Stack\* stack) {  // free up memory used up by data    // TODO: implement  }  int main()  {  Stack myStack;    StackInit(&myStack,8);  StackPush(&myStack,3);  StackPush(&myStack,4);  StackPush(&myStack,6);  StackPush(&myStack,1);  printf("Peek: %d\n",StackPeek(&myStack));  StackPush(&myStack,2);  StackPush(&myStack,5);  StackPush(&myStack,8);  StackPush(&myStack,9);  StackPush(&myStack,7);  printf("Peek: %d\n",StackPeek(&myStack));  printf("Pop: %d\n",StackPop(&myStack));  printf("Pop: %d\n",StackPop(&myStack));  printf("Pop: %d\n",StackPop(&myStack));  printf("Pop: %d\n",StackPop(&myStack));  StackPush(&myStack,11);  StackPush(&myStack,10);  printf("Peek: %d\n",StackPop(&myStack));  StackDestroy(&myStack);    return 0;  } |

## 

## Problem 1: [Finishing the Stack]

Please complete the above implementation so that the three functions, StackPeek, StackPop and StackDestory can work properly. The correct output, if you do not modify the main, should look like this:

|  |
| --- |
| Initializing stack to hold 8 integers... Pushing 3 Pushing 4 Pushing 6 Pushing 1 Peek: 1 Pushing 2 Pushing 5 Pushing 8 Pushing 9 Stack already full, returning Peek: 9 Pop: 9 Pop: 8 Pop: 5 Pop: 2 Pushing 11 Pushing 10 Peek: 10 |

Notice that:

1. You may find the Stack session of the lecture notes on "Abstract Data Type, Stack/Queue" helpful.
2. **Do NOT modify the Stack struct;** if you modify it, there will be **NO MARKS** for the exercises
3. If a stack is empty, we can assume Peek and Pop to return -1;

Please paste your finished program here:

|  |
| --- |
| int StackPeek(Stack\* stack) {  if(stack->top == -1)  return -1;  else  return stack->data[stack->top];  }  int StackPop(Stack\* stack) {    // return what's on the top and remove it  if(stack->top == -1)  return -1;  else {  int pop = stack->data[stack->top];  stack->top--;  return pop;  }  // TODO: implement  }  int StackDestroy(Stack\* stack) {  // free up memory used up by data  free(stack->data);  // TODO: implement  } |

## Problem 2: [Queue with Circular Array]

Please implement a Queue ADT using circular array, as demonstrated in the lecture notes. Please complete the code in the following template:

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  typedef struct {  int \*circularArray;  int maxSize; // max size of queue; size of circular array  int frontIndex; // index of the front of queue  int rearIndex; // index of the end of queue  int queueLength; // lenght of queue at the moment; 0 means no one is in queue  } Queue;  void QueueInit(Queue\* queue, int size) {  printf("Initializing queue of max size %d\n",size);    queue->circularArray = (int\*) malloc(sizeof(int) \* size);  queue->maxSize = size;  queue->frontIndex = 0;  queue->rearIndex = -1;  queue->queueLength = 0;  }  void Enqueue(Queue\* queue, int data) {  printf("Enqueueing %d\n",data);    if(queue->queueLength == queue->maxSize)  printf("Queue full!\n");  else {  queue->rearIndex++;  queue->queueLength++;  queue->circularArray[(queue->rearIndex) % (queue->maxSize)] = data;  }  // TODO: implement  // if queue is full, print "Queue full!" and do not enqueue  }  int Dequeue(Queue\* queue) {  // TODO: implement  // if queue is empty, print "Queue empty!" and return -1  if(queue->frontIndex > queue-> rearIndex){  printf("Queue empty!\n");  return -1;  }  else {  int dequeueData = queue->circularArray[(queue->frontIndex) % (queue->maxSize)];  queue->frontIndex++;  queue->queueLength--;  return dequeueData;  }  }  void QueuePrint(Queue\* queue) {  printf("Print queue: ");  if (queue->queueLength == 0) {  printf("(empty)\n");  return;  } else if (queue->queueLength <0) {  printf("Queue length is negative! Something's wrong!\n");  return;  }    int i;  for (i=0;i<queue->queueLength;i++) {  int circularIndex = (queue->frontIndex+i)%queue->maxSize;  printf("%d ",queue->circularArray[circularIndex]);  }  printf("\n");  }  int main()  {  Queue myQueue;    QueueInit(&myQueue,6);  QueuePrint(&myQueue);  Enqueue(&myQueue,10);  QueuePrint(&myQueue);  Enqueue(&myQueue,11);  Enqueue(&myQueue,12);  Enqueue(&myQueue,13);  Enqueue(&myQueue,14);  Enqueue(&myQueue,15);  Enqueue(&myQueue,16);  QueuePrint(&myQueue);  int i;  for (i=0;i<4;i++) {  int result = Dequeue(&myQueue);  if (result != -1)  printf("Dequeueing %d\n",result);  }  QueuePrint(&myQueue);  Enqueue(&myQueue,16);  for (i=0;i<5;i++) {  int result = Dequeue(&myQueue);  if (result != -1)  printf("Dequeueing %d\n",result);  }  QueuePrint(&myQueue);  } |

There are multiple ways to implement a circular array. Please follow the implementation in the template. In particular:

1. QueuePrint has already been done for you; reading it will be essential for you to understand this template!
2. You may find the Queue session of the lecture notes on "Abstract Data Type, Stack/Queue" helpful if you forgot what circular array is already.
3. **Do NOT modify the Queue struct;** you will receive **NO MARKS** if you modify it
4. Enqueue should **print "Queue full!"** if the queue is already full and you want to add one more to the queue
5. Similarly, Dequeue should **print "Queue empty!" and return -1** if queue has nothing to dequeue from
6. The main output, if you do not modify the main, should look like this:

|  |
| --- |
| Initializing queue of max size 6 Print queue: (empty) Enqueueing 10 Print queue: 10  Enqueueing 11 Enqueueing 12 Enqueueing 13 Enqueueing 14 Enqueueing 15 Enqueueing 16 Queue full! Print queue: 10 11 12 13 14 15  Dequeueing 10 Dequeueing 11 Dequeueing 12 Dequeueing 13 Print queue: 14 15  Enqueueing 16 Dequeueing 14 Dequeueing 15 Dequeueing 16 Queue empty! Queue empty! Print queue: (empty) |

## Problem 3 [Revision: Accessing Arrays with Pointer]

We have the following incomplete program:

|  |
| --- |
| #include <stdio.h>  void PrintArray(int\* arrayHead) {  // the array is assumed to be terminated by a zero element  int\* current; // pointer for traversing the array!  current = arrayHead;  for(int i = 0; i < 8; i++){  printf("%d ", \*current);  current++;  }  // insert your code here  }    int main()  {  int a[] = {5,4,3,2,1,2,3,4,0};    PrintArray(a);    return 0;  } |

Please complete the program, **without using array notation or changing the main program and parameters,** so that the output will become:

|  |
| --- |
| 5 4 3 2 1 2 3 4 |

Hint: Do you recall the relation of arrays with pointers? Please check Lecture Notes Part 1/1b if you don't have a clue what I am talking about!

## Problem 4 [Revision: Matrix Display]

Here you will find an incomplete program that can display a matrices with different rows and columns. Please complete the functions PrintMatrix and PrintMatrixTranspose so that it can function properly:

* PrintMatrix will print the matrix on screen properly given a 1D array that contains integer elements of the matrix, the number of rows, and the number of columns.
* Similarly, PrintMatrixTranspose will print the transpose of the matrix on screen, with the same parameter setup of PrintMatrix
* **Please do NOT change the function parameters** of PrintMatrix and PrintMatrixTranspose. HINT: 2D arrays are stored in a continuous block of memory in C as well

|  |
| --- |
| #include <stdio.h>  void PrintMatrix(int\* matrix, int numRows, int numColumns) {  int \*current;  current = matrix;  // insert your code here  for(int i = 0; i < numRows; i++){  for(int j = 0; j< numColumns; j++){  printf("%d ", \*current);  current++;  }  printf("\n");  }  }  void PrintMatrixTranspose(int\* matrix, int numRows, int numColumns) {  //int \*transpose;  //transpose = (int\*)malloc(numRows \* numColumns \* sizeof(int));  for(int i = 0; i < numColumns; i++){  for(int j = 0; j < numRows; j++){  printf("%d ", \*(matrix + (j \* numColumns + i)));  }  printf("\n");  }  }  int main() {  int a[] = {1,4  ,2,5  ,3,6};  int b[] = {1,4,7,10,  2,5,8,11,  3,6,9,12  };    printf("a: \n");  PrintMatrix(a,3,2);  printf("b: \n");  PrintMatrix(b,3,4);  printf("a^T: \n");  PrintMatrixTranspose(a,3,2);  printf("b^T: \n");  PrintMatrixTranspose(b,3,4);      return 0; } |

The correct output of the above program is:

|  |
| --- |
| a:   1 4   2 5   3 6  b:   1 4 7 10   2 5 8 11   3 6 9 12  a^T:   1 2 3   4 5 6  b^T:   1 2 3   4 5 6   7 8 9  10 11 12 |

|  |
| --- |
| Please finish other Lab Exercises if you haven't already! They can give significant bonuses to your tests and exam scores. Late submission will also always give higher marks than no submission at all. |