CSE213L: DSA Lab

LNMIIT, Jaipur

Training Set 02

In this training set, students will use and develop ADT interface operations related to queue and deque data-structures. In addition, class will learn about a way to organise functions in modules. A convenient arrangement for this is based on files and directories. The functions that are closely related to one-another are grouped together in a file and closely related files are placed together in a directory. The goal is to have functions with close interaction and commonality of purpose in a single module/file.

This training set is organised around data structure disciplines deque and queue. To help you start training, we give you a partially developed program for a deque. There are three files provided to you. You may use these files to begin working on the tasks described below. The provided code implements a deque using indices `left` and `right` to represent two ends of data stored in a circular array arrangement.

Training Set 02: Task 01

Read K&R book for more information about the header files and their purposes. Other good C books will also have this information. There are many features which we have not used in the code given to you, but over time as a student in the discipline of IT, you need to learn all these concepts. For now, this code is a decent start.

1. Write codes for functions in file `deque.c` that are incomplete in the provided code.
2. Next, construct a header file `queue.h` to specify ADT functions needed in program `main.c`. At present, function `main()` is using an implementation of deque for its data storage needs. These needs can be better satisfied through a queue ADT interface.
3. Create file `queue.c.` In this file you will implement functions declared in header `queue.h`. The code necessary to implement these functions is quite simple. The codes for functions in file `queue.c` will simply forward the calls to the appropriate functions listed in header file `deque.h`.
4. In your demonstration of this task to your tutor, your program must demonstrate:
    a. function `main()` that includes interface to queue, but not to deque. That is, it has `#include "queue.h"` MUST NOT have `#include "deque.h"`. The program should perform the same tasks that the present code (`main.c`) is able to perform.
    b. There SHOULD NOT be any program file with `#include` directive to include a `.c` file.
    c. ADT queue functions `joinQ()` and `leaveQ()` should be implemented by calls to the deque functions that are unimplemented in the code provided later in this document.
5. Just in case you do not know the compilation commands, use:

```
gcc main.c queue.c deque.c -lm
```

Or, simply use

```
gcc *.c -lm
```

Training Set 02: Task 02

Study and learn the well-known algorithm for finding prime numbers called Sieve of Eratosthenes (https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes) before you lab session. You should be able to write code to compute primes in range 1 to 99999 using the implementation of queue in Task 01. For this you may need to set SIZE to value 100001.

1. Initially, fill the queue with all values from 2 to 99999. This is your initial run (sequence) of candidates for primes. Insert a special marker value into the queue to separate the first run of the prime number candidates from the next run of (surviving) candidates that you will insert in the queue during the computation. Zero (0) is a good number for this purpose. We are now ready to find the prime numbers. In each iteration, a prime will be selected and the selected prime will be used to eliminate candidates values divisible by it. The following step describes the algorithm in detail.
2. Each run of the sieve will read the remaining candidate values from the queue till the run of candidates ends. This end will be noted when the special separator value is read.
    a. First number read from a run is a prime number and it is printed on the output stream (`stdout`).
    b. After finding a successful candidate that qualifies as the next prime number, the remaining numbers in the current run are read to build the next run of the prime candidates. The multiples of the newly found prime number are removed and eliminated. The numbers not eliminated are added to the queue after a run separating value 0.
3. The process of finding prime numbers ends when we cannot build a new run of candidates for primes. As no candidate is available, no new prime can be printed.

Training Set 02: Task 03

Modify the implementation of deque as follows.

We will change the purpose and meaning of indices `left` and `right`. In our implementation of a circular array, these indices were used to mark the leftmost and the rightmost indices of (circular) array locations containing data. The variables mark the positions where data is already stored. If no data is stored in the array, these indices are assigned value -1.

In the new arrangement, indices `left` and `right` indicate the locations in the array where a new data can be placed. If the array has no data, the two indices will refer to two circularly neighbour positions in the array. For example, function `init()` can set `left = 8` and `right = 9`.

In this new arrangement, the array can only hold SIZE – 1 many values and no more.

Re-implement all functions in file `deque.c` for this new interpretation of indices `left` and `right`. After a correct implementation of the ADT functions, Task 02 programs should work correctly *without any* modification in files `deque.h`, `queue.c`, `queue.h` and `main.c`.

Vishv Malhotra
29 April 2020
The LNMIIT, Rupa ki Nangal
Jaipur 302031.

Function main.c()

```
#include <stdio.h>
#include <stdlib.h>
#include "deque.h"

int main(void) {
      int i, s;
      init(); // Start deque

      for (i=0; i< 10; i++) {
            insertLeft(i*i);
            s = size();
            printf("Size = %d Data = %d\n", s, removeRight());
            insertLeft(i*i*i);
      }

      while (size()>0) {
            printf("Emptying deque %d\n", removeRight());
      }

      return 0;
}
```

Header File deque.h

```
/* Header File for deque */
#define SIZE 100
#define ERR_DATA (-302031)

void insertLeft(int d);
void insertRight(int d);
int removeLeft();
int removeRight();
int canWelcome();
int isEmpty();
void init();
int size();
```

Starter code for deque.c

```
/* An implementation of Deque  */
#include "deque.h"

int data[SIZE];
int left = -1;
int right = -1;

void insertLeft(int d) {
     if (size() == SIZE-1)
          return;
     if (left==-1) {
          left=right=0;
          data[left]= d;
          return;
     }
     left = (left-1+SIZE)%SIZE;
     data[left] = d;
}

void insertRight(int d) {
     // Not implemented
}

int removeLeft() {
     return 0; // Not implemented
}

int removeRight() {
     int d, s;

     s = size();

     if (s==0)
          return ERR_DATA; // Error value
     d = data[right];
     right = (right - 1 + SIZE)%SIZE;
     if (s == 1)
          init();
     return d;
}
```

```
int canWelcome() {
     return size() < SIZE;
}

int isEmpty() {
     return size() == 0;
}

void init() {
     left = right = -1;
}

int size() {
     if (left == -1)
          return 0;
     return (right+SIZE-left)%SIZE+1;
}
```

Program output

```
Size = 1 Data = 0
Size = 2 Data = 0
Size = 3 Data = 1
Size = 4 Data = 1
Size = 5 Data = 4
Size = 6 Data = 8
Size = 7 Data = 9
Size = 8 Data = 27
Size = 9 Data = 16
Size = 10 Data = 64
Emptying deque 25
Emptying deque 125
Emptying deque 36
Emptying deque 216
Emptying deque 49
Emptying deque 343
Emptying deque 64
Emptying deque 512
Emptying deque 81
Emptying deque 729
```