CSE213L: DSA Lab

LNMIIT, Jaipur

Training Set 02

In this training, students will learn basic ADT operations on queue and deque. In addition, class will learn about a way to collect functions into modules. A convenient arrangement for this is based on files and directories. The functions that are closely related to one-another may be grouped together in a file (or module). The idea is to have strong cohesion and commonality of purpose among the functions in a single module/file.

The training program in this set is organised around a deque and a queue. To help you start, we give a partial implantation of a deque. There are three files provided to you. You may use them to quickly start working on the tasks described below.

Training Set 02: Task 01

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

1. Write the code for the functions in module `deque.c` that are incomplete.
2. Next, construct a header file `queue.h` with ADT functions needed by program `main.c`. At present, function `main()` is using an implementation of deque for data storage needs that are better satisfied through ADT interface for a queue.
3. Create a file `queue.c` in which you will implement functions declared in header `queue.h`. The new code needed is very simple. The functions will simply forward the call to the appropriate functions of the deque.
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"` but does not have `#include "deque.h"`. It should perform the same tasks that the present code (`main.c`) is able to perform.
    b. You queue functions `joinQ()` and `leaveQ()` should be implemented by calls to the deque functions that are unimplemented in the start-up code provided to the students.
5. Just in case you do not know the compilation command:

    ```
    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 candidates from the next run of (surviving) candidates that you will insert in the queue during computation. 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 unsuitable candidates.
2. Each run of the sieve will read the candidate values from the queue till the run of candidates ends. This end will be noted when the special separator value is read. (Zero is good number for this purpose in this problem).
    a. First number read from a run is a prime and it is printed on the output stream (`stdout`).
    b. After that the numbers are read and inserted into the queue to build the next (new) run of candidates for primes. The multiples of the prime number are removed and eliminated. The numbers not eliminated are added to the queue.
3. The process ends when we can not build a new run of candidates for primes. No new candidate is available.

Training Set 02: Task 03

Modify the implementation of deque as follows.

We will change the purpose and meaning of indices `left` and `right`. These will be indices where the new data can be placed. Note this is different from how we used these variables in the implementation given to you (and in our textbook), In the implementation given to you, these index point at the position where data is already stored.

In the new arrangement, indices `left` and `right` will point at the positions in the array that are available for storing the next data. For example, function `init()` can set `left = 8` and `right = 9`. In this new arrangement, clearly the array can only hold SIZE – 1 many values and no more.

Re-implement all functions for this new approach. After a correct implementation of the module, Task 02 should work correctly *without any* modification in files `deque.h`, `queue.c`, `queue.h` and `main.c`.

That is all for today.

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
```