Indian Institute of Technology Tirupati
B.Tech in Computer Science & Engineering
Semester 2, Academic Year 2017-18

CS2810 Advanced Programming Lab

Instructor	**: Dr. Venkata Ramana Badarla**
L-T-P-C	**: 0-0-3-2**
Lab Slot	**: Tue 3:00 to 5:50pm (Slot E)**
Lab	**: Computer Center**

Lab Experiments:

Week	*Exercise*	*Experiment*
1	*1*	Algorithm efficiency and Time complexity – Learn through simple examples
2	*2*	Sorting algorithms and Time complexity
3	*3. & 4*	Linear Abstract data structures – LIST, STACK, and their Applications
4	*5*	QUEUE / Cir QUEUE and Event driven Simulator
5		*Buffer Week*

Exercise 1

1. Algorithm efficiency and Time complexity – Learn through simple examples

Objective: To show the impact of algorithm efficiency on the time complexity through simple examples

(A) Algorithm to check the given number N, is prime or not (primality test) implementing in three different ways where number N is checked against all the numbers up to, i) N – 1, ii) N/2 iii) SQRT (N). Test the time taken for the primality test in these approaches for different values of N. Finally the time taken for each of the approach is to be plotted in a graph.

(B) Algorithm to generate Fibonacci numbers up to a given number N implementing with and without recursion and checking the performance in both cases for large values of input N.

(C) Find Kth largest number in a given group of N numbers

(D) Given two numbers x and n, write a program to compute x power n, using recursive doubling and without using recursive doubling.

(E) Removing all the duplicate elements in an array and print only unique numbers.

Exercise 2

1. Sorting algorithms and Time complexity

Objective: To learn sorting algorithms and measuring their time complexity

Three sorting algorithms Selection, Insertion and Quick sorting algorithms are to be implemented. The performance comparison of these three algorithms for different input sizes, 1K, 10K, 50K, 100K, is to be done separately for the following cases

(A) Input data generate at random – to study the average case
(B) Input data is already sorted – to study the best case
(C) Input data is sorted in reverse order – to study the worst case

Finally a separate comparative performance graph is to be plotted for each case and algorithm.

Exercise 3

1. LIST Data Structure

Objective: To implement a generic linear data structure, LIST, to store the objects using dynamic memory allocation

LIST data structure to support the following operations to be implemented.

(A) initialize – to initialize the list variables of a given list L
(B) insert - to insert an element 'e' at position 'p' in a given list L
(C) delete – to delete an element for a given position p in a given list L
(D) locate – to locate an element e in a given list L
(E) retrieve – to retrieve an element at position p in a given list L
(F) print – to print the elements of a given list L
(G) first – to return the position of the first element in the list
(H) last – to return the position of the last element in the list

Exercise 4

Stack and its applications

Objective: To implement Stack data structures from a generic LIST data structure (Code Reusability)

This exercise includes implementation of stack data structure using List and development of an application for evaluating arithmetic expression with round parentheses using the stack data structure.

Evaluation of express has two parts

(A) Converting the given expression into postfix format
(B) Evaluating the postfix expression

Implementation will be verified for any general arithmetic expression with round parentheses.

Exercise 5

EVENT DRIVEN SIMULATOR – Grocery Store

Objective: To learn Queue data structure and implementing its application

This exercise provides a basic framework for an event driven simulator. You need to simulate the operations of a grocery store with checkout queues. Assume that there are **n** checkout queues (1, 2, 3 … n). There are two events that could happen in the system.

(A) Arrival of a customer into any one of the shortest length checkout queue
(B) Departure of a customer from a queue

In this case you require **n+1** event clocks (one for each checkout queue and one for arrival event). Let us take an array EVENTCLOCK that contains these event clocks.

(A) EVENTCLOCK[1] - time at which next customer departs from queue 1
(B) EVENTCLOCK[2] - time at which next customer departs from queue 2
(C) …..
(D) ….
(E) EVENTCLOCK[n] - time at which next customer departs from queue 2
(F) EVENTCLOCK[n+1] - time at which next customer arrives and enters into shortest length queue

MAINCLOCK is used to record the current time in the simulation. At each time instance you need to pick an event, if any, scheduled to take place at that time instance. The parameter SIMULATION_TIME represents the maximum simulation time.

Outline of the activities:

(A) Next customer arrives at a random time selected from the range [0,300] seconds
(B) The number of items purchased by each customer is also a random number selected from the range [1,25]
(C) Assume that each item requires 30 seconds of service time by the checkout person. So if a customer purchases 3 items, then he/she requires 3x30 = 90 seconds overall service time at the respective checkout queue.

Outline of the Algorithm:

```
1. Instantiate N departure queues OR array of departure queues, QLIST
2. Initialize all event clocks of departure events AND arrival event

MAIN_CLOCK = 0        /* reset the simulation start time */
EVENT_CLOCK[N+1] = 0  /* reset the event clock of arrival event */
```

```
for I = 1, 2, ..... N,    (where N is number of events)
           EVENT_CLOCK[I] = INFINITY (a large value)
Note: Initially arrival event should happen before scheduling any
```
3. While (MAIN_CLOCK < SIMULATION_TIME)
```
      J = MIN (EVENT_CLOCK)              /* determine the next event */
      MAIN_CLOCK = EVENT_CLOCK[J]
      IF J = N+1
      /* add new customer to one of the shortest departure queue */
           ARRIVAL(QLIST)
      ELSE
      /* Customer from Queue J will leave now from the system */
           DEPART (J, QLIST)
```
4. Output relevant statistics for simulation

Requirement: This will require to develop Queue ADT with member functions, enqueue( ), dequeue( ) , front( ) and printQueue( ) etc. Although you will instantiate two queues, each one of these will consist the same kind of queue nodes. Define a queue node in the following manner:

```
typedef struct queueNode {
      int custNum;          /* Unique identifier;
                               starts at 1; after 24 hours
                               should be reset to 1 */
      int totalItems;       /* random number between [1 to 25]  */
      int serviceTime;      /* (total Items x 30 seconds) / 60;
                               units in minutes */
      int depTime;          /* depTime = serviceTime+sum of
                               serviceTimes of customers in line
                               before this customer */
    struct queueNode *pNext;
} QueueNode;
```

Your simulator should determine the following for the above **normal** checkout queue system:

(A) Number of customers serviced,
(B) Number of customers remaining,
(C) Average time spent by each customer in the line,
(D) Average waiting time for each customer, and
(E) Average idle time for the checkout clerks

Further, assuming that the Grocery stores wanted to open one more checkout queue of type either express (where customers having less than 5 times only will enter into the queue) or normal, suggest which of these queue types is better to open.