




Section 1: MRQ (Each question = 0.5 x 4 = 2 marks)

Question 1 to 5 are **multiple response question (MRQ)**. You need to evaluate each suboption given in the question to either "1" (True) or "0" (False). Write **only "1" or "0"** in the answer sheet. **Each suboption is worth 0.5 marks.**

1. Each of the following cases insert zero or more statements at the Point α and β . Evaluate whether the described behavior is T (true, i.e. correct) or F (false, i.e. incorrect).

```
int main( )
{
    //This is process P
    if ( fork() == 0 ){
        //This is process Q
        if ( fork() == 0 ) {
            //This is process R
            .....
            return 0;
        }
        <Point  $\alpha$ >
    }
    <Point  $\beta$ >
    return 0;
}
```

	Point α	Point β	Behavior
a	nothing	wait(NULL);	Process Q always terminate before P. Process R can terminate at any time w.r.t. P and Q. 
b	wait(NULL);	nothing	Process Q always terminate before P. Process R can terminate at any time w.r.t. P and Q. 
c	wait(NULL);	wait(NULL);	Process P never terminates. 

			<div></div> <div></div> <div></div>
d	<code>execl(valid executable....);</code>	<code>wait(NULL);</code>	Process Q always terminate before P. Process R can terminate at any time w.r.t. P and Q. <div></div> <div></div>

2. Evaluate each of the following statements regarding **Process Control Block (PCB)**.

a	The hardware context in the PCB always reflect the actual register values in the processor. <div></div>
b	The memory context in the PCB is not the actual memory space used by the process. <div></div> <div></div>
c	The PCBs themselves are stored in the memory. <div></div>
d	The OS context of a PCB may contains information used for scheduling, e.g. priority, time quantum allocated, etc. <div></div>

3. Evaluate each of the following statements regarding **Process Scheduling**.

a	The interval between timer interrupt is always an integer multiple of time quantum. <div></div>
b	The MLFQ algorithm favors (i.e. give higher priority to) IO Intensive process. <div></div>
c	Non-preemptive scheduling algorithm cannot cause starvation. <div></div> <div></div>
d	Given the same period of time, smaller interval between timer interrupt lengthen task turn-around time. <div></div> <div></div>

4. Evaluate each of the following statements regarding **Thread** and **POSIX Thread (pthread)**.

a	All pthreads must execute the same function when they start. [REDACTED]
b	Multi-threaded program using pure user threads can never exploit multi-core processors. [REDACTED]
c	If we use a 1-to-1 binding in a hybrid thread model, the end result is the same as pure-user thread model. [REDACTED]
d	On a single-core processor that support simultaneous multithreading, we can execute more than one thread at the same time. [REDACTED]

5. Evaluate each of the following statements regarding **Unix Signal**.

a	A process can install user-define handler for multiple different signals . [REDACTED]
b	We can install user-define handler for all signals. [REDACTED]
c	A parent process can force the child processes to execute any part of their code by sending signal to them. [REDACTED]
d	The "kill" signal (sent by the "kill" command) is different from the "interrupt" signal (sent by pressing "ctrl-c"). [REDACTED]

Section 2: Short Questions (30 marks)

Question 6 (10 marks)

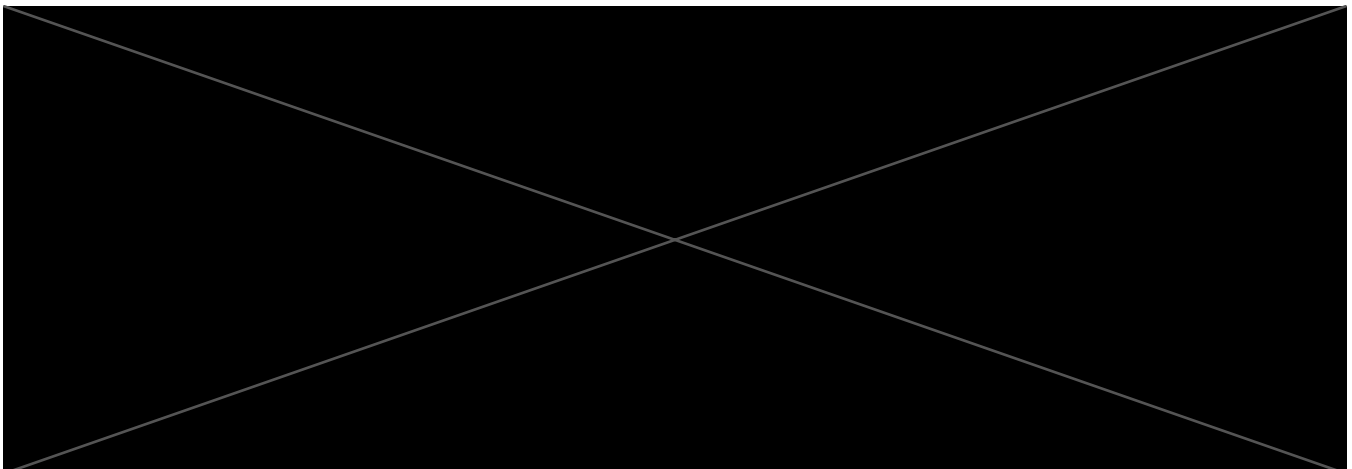
Below is a program **P** sketch:

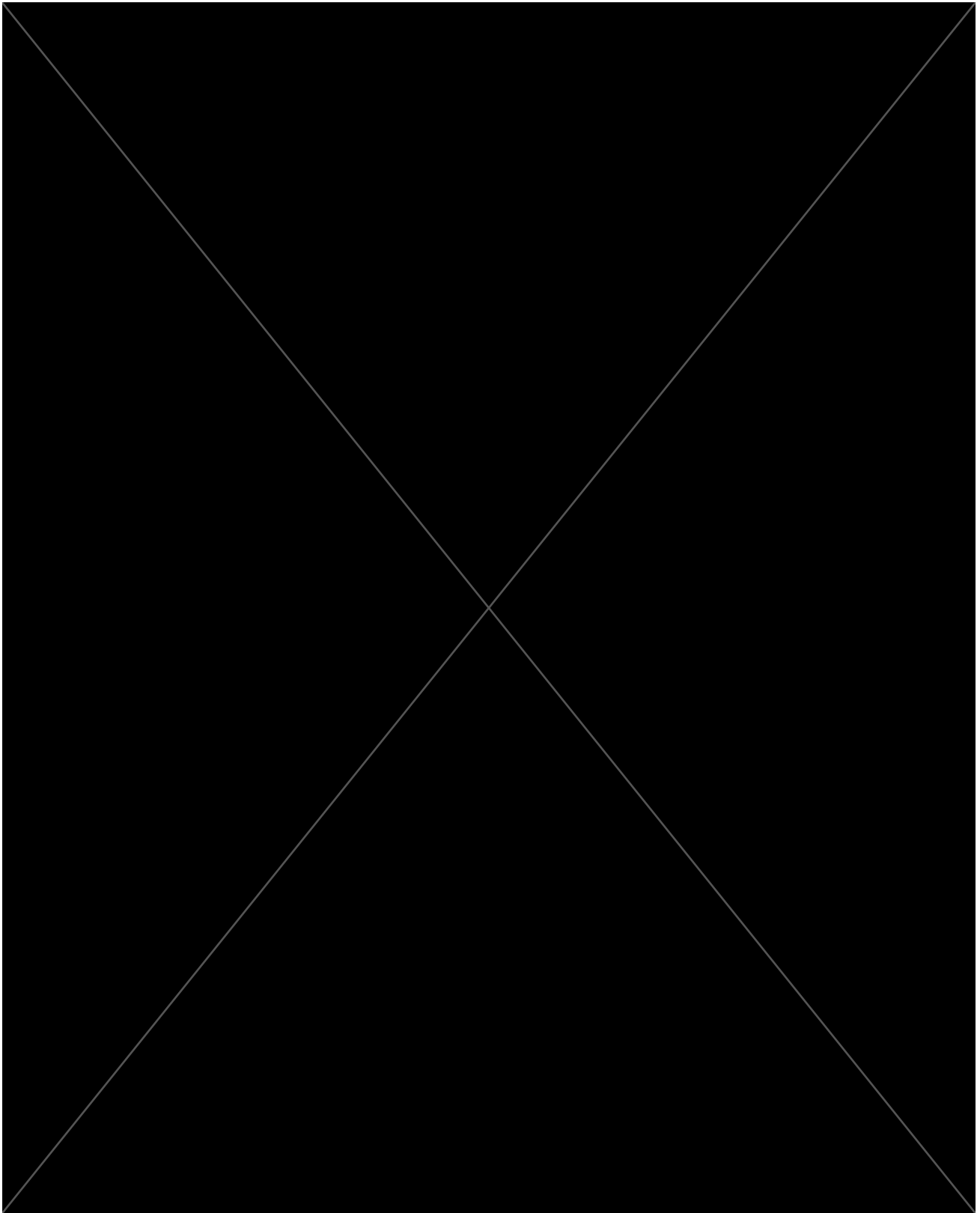
```
int main() {
    int i, result[100000];    //100,000 results

    for (i = 0; i < 100000; i++)
        result[i] = compute(i);    //compute takes ~1ms per call
    <Write the result[] array into a file "secret.txt">
}
```

Suppose **P** is executed on an OS with **standard 3 levels MLFQ scheduler** with a **100ms time quantum**. For consistency, let's assume **priority 2 is the highest** and priority 0 is the lowest.

- Briefly describe the priority change of **P** overtime on this system. **[1 mark]**
- Briefly describe with **pseudo-code** on how to maximize processor usage **unfairly** on this system to complete the computation as fast as possible. **[3 marks]**
- As discussed in tutorial, exploits like (b) can be mitigated by keep track of processor usage **across timeslices**. By using **only topics learned in CS2106 so far**, briefly describe with **pseudo-code** on how you are still able to maximize processor usage **unfairly**. You need to state all relevant assumptions and explain how the **result[]** array is maintain properly (contains all result, in the same order as the original code) through the "cheat". **[4 marks]**
- Suggest one simple scheduler fix for your exploit in (c). **[2 marks]**





Question 7 (8 marks)

In many programming languages, function parameter can be **passed by reference**. Consider this fictional C-like language example:

```
void change( int<Ref> i ) { //i is a pass-by-reference parameter
    i = 1234; //this changes main's variable myInt in this case
}

int main() {
    int myInt = 0;
    change( myInt );    //myInt become 1234 after the function call
    ..... //other variable declarations and code
}
```

Mr. Holdabeer feels that he has the perfect solution **that works for this example** by relying on **stack pointer and frame pointer**. The key idea is to load main's local variable "myInt" whenever the variable "i" is used in the change() function.

Given that the stack frame arrangement shown **independently** as follows:

For Main()			For change()		
		← \$SP			← \$SP
...	...		Saved SP	-8	← \$FP
myInt	-12		Saved FP	-4	
Saved SP	-8	← \$FP	Saved PC	0	
Saved FP	-4				
Saved PC	0				

- Suppose the main()'s and change()'s stack frame has been properly setup, and change() is now executing, show how to store the value "1234" into the right location. You only need pseudo-instructions like below. **[3 marks]**
 - Register_D ← Load Offset(Register_S)
Load the value at memory location [Register_S] + Offset and put into Register_D
e.g. \$R1 ← Load -4(\$FP)
 - Offset(Register_S) ← Store Value
Put the value into memory location [Register_S] + Offset,
e.g. -4(\$FP) ← Store 1234
- Briefly describe another usage scenario for pass-by-reference parameter that **will not work with** this approach. **[2 marks]**
- Briefly describe a better, universal approach to handle pass-by-reference parameter on stack frame. Sketch the stack frame for the change() function to illustrate your idea. **[3 marks]**

[REDACTED]

[REDACTED]

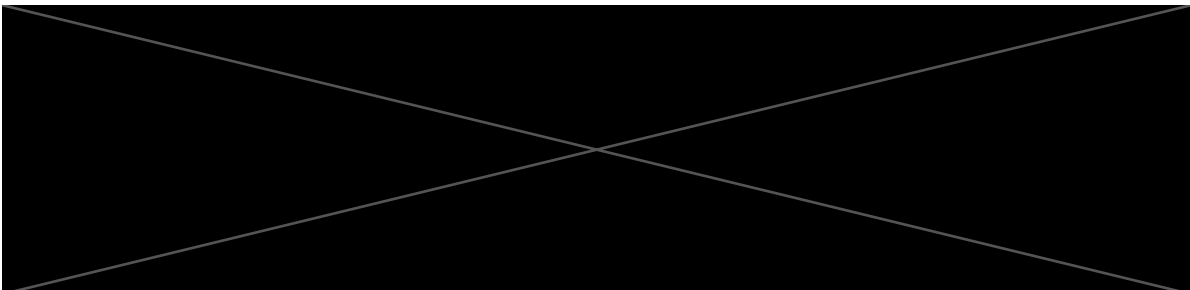
[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]

[REDACTED]



Question 8 (12 marks)

The code used by this question is on the next page so that you can **tear it out (up?)** to facilitate your answering.

For (a) and (b), we will study the program behavior of **multiple threads**. If the described behavior is possible, give the execution pattern using `t<thread number>.<line number>-<line number>` ,e.g. `t1.23-25` (thread 1 executing line 23 to 25), `t2.23-24` (thread 2 then execute line 23 and 24. Otherwise, give your answer as "impossible".

- a. Suppose we have initialized the global linked list to $2 \rightarrow 1$ (1 is the last node). Is the following output **possible** if we execute **2 threads on the printList() function?**

[3 marks]

```
2 Nodes
2 Nodes
2 2 1 1
```

- b. Suppose we start with an **empty** global linked list, we then execute **2 threads** on the `insertNumbers(2)` function. Can the global linked list contain only $2 \rightarrow 1$ (i.e. only 2 values) after both threads finish?

[3 marks]

For (c) and (d), use general semaphore to ensure the indicated program behavior. You should i) **use the least number of semaphore(s)** and ii) protect the **least number of lines of code**. Show the semaphore declarations clearly then use the line number to indicate where you want to insert the semaphore(s) operations (i.e. `wait(...)` and `signal(...)`).

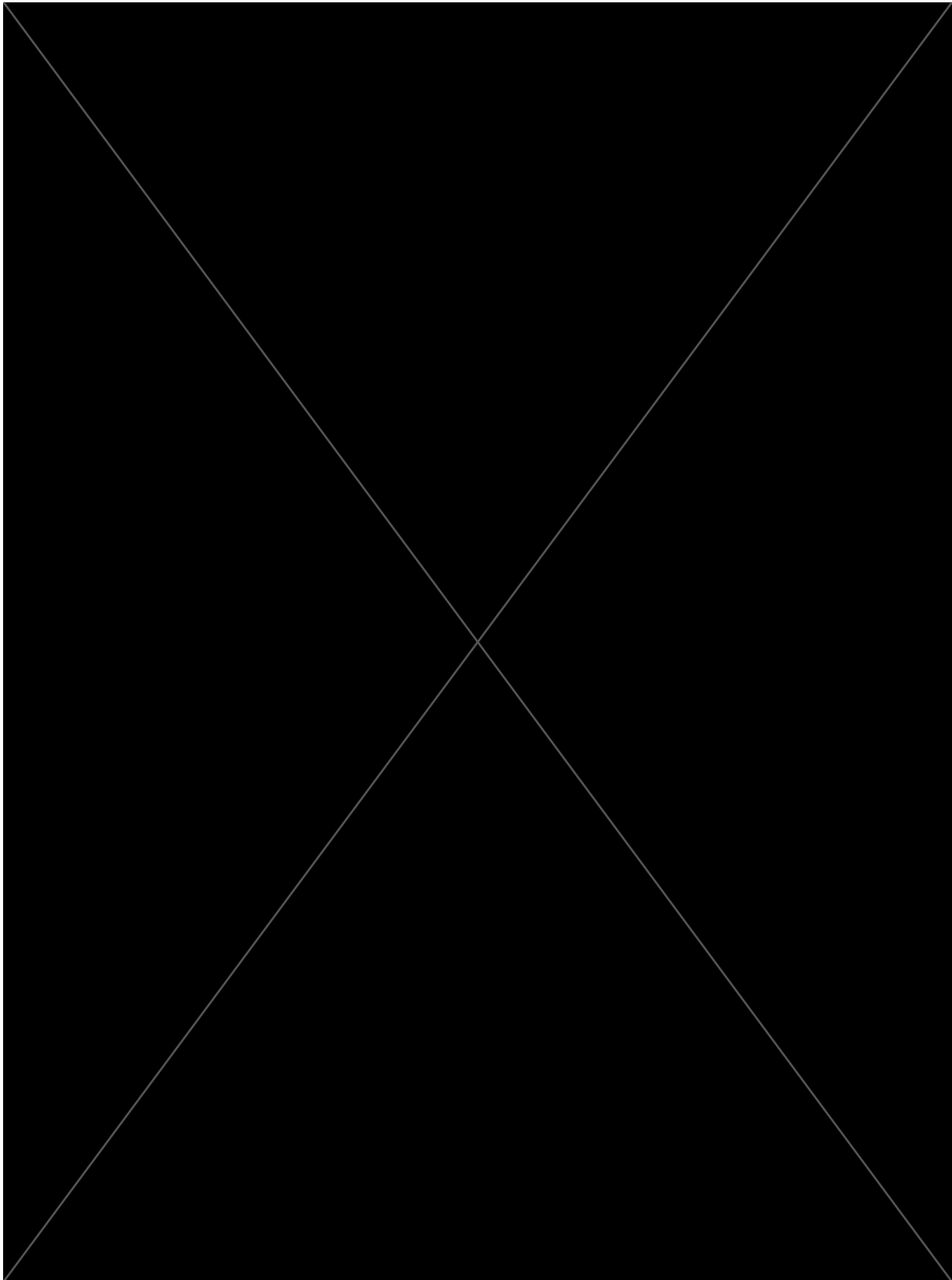
- c. Suppose we run **10 threads** on the `insertNumbers(4)` function. Use semaphore to ensure that the global linked list has 40 nodes and the global node count = 40 after all threads finish.

[3 marks]

- d. Suppose we run multiple threads on the `insertNumbers()` function and multiple threads on the `printList()` function. Ensure the `printList()` function:

- The printed number of nodes (i.e. line 24) always matches the total number of node values printed.

[3 marks]



[REDACTED]

[REDACTED]

Section 3: Bonus Question (1 mark)

9. "***This is a massacre!***": Predict the **class average score of this paper** (excluding this bonus question). If your prediction is with ± 1 mark of actual average, you will get a bonus "*I can see the future*" 1 mark. 😊

[REDACTED]

[REDACTED]

Code for Question 8 (You can tear out this page for ease of answering)

Below is a modified code from the "linked list lab". The main differences are:

- The linked list is now maintained by a **global head pointer gHead** (line 6).
- There is a global count of number of nodes, **gTotalNode** (line 7).
- The insertion always insert at head position (i.e. first position) (line 9 to 15).
- There is a helper function to insert a series of number 1...N at the head of linked list (line 17 to 21).

```

1  typedef struct NODE {
2      int data;
3      struct NODE* next;
4  } node;
5
6  node* gHead = NULL;
7  int gTotalNode = 0;
8
9  void insertHead( int newData ) {
10     node* newNode = (node*)malloc(sizeof(node));
11     newNode->data = newData;
12     newNode->next = gHead;
13     gHead = newNode;
14     gTotalNode++;
15 }
16
17 void insertNumbers( int N ) {
18     for (int i = 1; i <= N; i++){
19         insertHead(i);
20     }
21 }
22
23 void printList() {
24     printf("%d Nodes\n", gTotalNode);
25     for ( node* ptr = gHead; ptr != NULL; ptr = ptr->next)
26         printf("%i ", ptr->data);
27     printf("\n");
28 }
29

```