ESC101: Fundamentals of Computing (End Semester Exam)

Version A

Total Number of Pages: 17

Total Points 105

Question	Points	Score
1	5	
2	5	
3	30	
4	35	
5	30	
Total:	105	

Instructions

1. Read these instructions carefully.
2. Write you name, section and roll number on all the pages of the answer book, **including the ROUGH pages**. You will be penalised if you fail to write the name, roll number and correct section.
3. Write the answers cleanly in the space provided. Space is given for rough work in the answer book.
4. Using pens (blue/black ink) and not pencils. Do not use red pens for answering.
5. Do not exchange question books or change the seat after obtaining question paper.
6. Even if no answers are written, the answer book has to be returned back with name and roll number written.
7. Sign the attendance sheet.

I PLEDGE MY HONOUR THAT DURING THE EXAMINATION I HAVE NEITHER GIVEN NOR RECEIVED ASSISTANCE.

....
Signature

Question 1. (5 points) Consider the program given below.

```
#include <stdio.h>
#include <stdlib.h>
typedef struct node
{
        int data;
        struct node *next;
}N;
int main(){
  N *p, *q;
  p = (N*) malloc (sizeof(N));
  p->data = 10;
  q = (N*) malloc(sizeof(N));
  q->data = p->data*2;
  p->next = q; q->next = p;
  p->next->next->data = q->next->next->data*2;
  q->next->next->data = p->next->next->data*2;
  printf("%d %d",p->data, q->data);
  free(p); free(q);
  return 0;
}
```

What is the output of the program? If the program results in an error, mention the type of error.

Output:

```
40 80
```

Question 2. (5 points) Consider the program given below.

```
#include <stdio.h>
int foo(int a, int f){
  int b=0, c;
  if(a> 0)
    {
      c = a%2;
      c = c*f; f = f*10;
      b = foo(a/2,f) +c;
    }
  return b;
}
int main(){
  int n=10;
  printf("%d\n", foo(n, 1) );
  return 0;
}
```

What is the output of the program? If the program results in an error, mention the type of error.

Output:

```
1010
```

Solution

Question 3. (30 points) Pattern matching refers to checking the occurrence of a specific pattern in a given sequence. Pattern matching in text sequences often uses REGular EXpressions (or regex) to specify the pattern to be matched.

Most of the commonly used Linux command line tools including, but not limited to `ls` and `grep` use pattern matching with regex. `ls` is used to list all the files whose names match the given pattern. `grep` is used to search within files and return the lines which contain a match for the given pattern.

Given below is a partially filled code which does pattern matching using regex. The rules for defining patterns using regex for the purpose of this code are as following:

- *Symbol set:* The symbol set is the set of all ASCII characters (excluding the newline(\n) character). Any string comprised of symbols from the symbol set is a regex pattern. Thus, `MAXSIZE` will match all strings, which are the same as `MAXSIZE`.
- *Dot operator:* The `dot` operator (.) in a regex pattern is a placeholder for any symbol from the symbol set. Thus, `M..SIZE` will match all strings, which start with `M`, followed by exactly two symbols and end in `SIZE`.
- *Star operator:* The `star` operator (*) in a regex pattern is a placeholder for any string of length 0 or more, comprised of symbols from the symbol set. A string of length 0 means no string. Thus, `M*SIZE` will match all strings, which start with `M`, followed by any number of symbols (or no symbols) and end in `SIZE`.
- *Escape sequences:* If backslash (\) occurs in a pattern, the character following it is escaped and considered literally. Thus, `round_*\.txt` will match all strings, which start with `round_`, then contains any number of symbols (or no symbols) and end with `.txt`.

Fill in the blanks in the following code to complete the implementation of the regex pattern matcher.

Note: The blanks should be filled so that the arrays use the minimum possible memory which ensures that all possible strings of length less than or equal to MAX_LEN. Also, the blanks should be filled to guarantee that if someone wants to change MAX_LEN sometime later, they should only change line 12 and the code still works correctly.

```
1  /*
2     This program is a regex pattern matcher
3  
4     It takes as input n in first line
5     Next line takes a valid regex as input
6     Next n lines contain strings to match
7  
8     Outputs the matched strings in separate lines
9  */
10 #include<stdio.h>
11 #include<stdlib.h>
12 #define MAX_LEN 500
13 void read_line(char *in)
14 {
15         //This function reads a line from stdin
16         //ended by '\n' into the string pointed
17         //by in. It is assumed that in has enough
18         //memory allocated
19         char c = getchar();
20         int i=0;
```

```c
        while(c != '\n')
        {
                in[i++] = c;
                c = getchar();
        }
        in[i] = '\0';
}
int memo[MAX_LEN][MAX_LEN];
void init_memo()
{
        //This function initialises memo matrix to -1
        for(int i=0;i<MAX_LEN;++i)
                for(int j=0;j<MAX_LEN;++j)
                        *(&(memo[i][0])+j) = -1;
}
int match(char *str, char *regex, int i, int j)
{
        //This function returns 1 if the str matches regex
        //i and j store the length of string/regex matched (used in memo)

        //Base cases for recursion
        //If str is empty
        if(*str == '\0')
        {
                //Now regex should only contain '*'s.
                //Anything else can't match an empty string
                while(*regex != '\0')
                {
                        if(*regex != '*') return 0;
                        regex++;
                }
                return 1;
        }

        //Now we know that string is not empty
        //If regex is empty then it's not a match
        if(*regex == '\0')
                return 0;

        //Check if call already took place
        if(memo[i][j] != -1)
                return memo[i][j];

        int answer = 0;

        //Now we try to match first character of str and regex
        if(*regex == '\\')
        {
                //next character is escaped
                if(*str == *(regex+1))
                {
                        answer = match(str+1,regex+2, i+1, j+2);
                }
```

```c
        }
        else
        {
                if(*regex == '*')
                {
                        //Either match a character with '*' and let * remain for future matching
                        //Or Don't match a character and end '*'
                        answer = match(str+1, regex, i+1, j) || match(str, regex+1, i, j+1);
                }
                else if(*regex == '.')
                {
                        answer = match(str+1, regex+1, i+1, j+1);
                }
                else
                {
                        if(*str == *regex)
                        {
                                answer = match(str+1,regex+1, i+1, j+1);
                        }
                }
        }
        return memo[i][j]=answer;
}
int main()
{
        int n;
        char* regex;
        char** strings;

        scanf("%d\n", &n);      //Note: "\n" is important here

        regex = (char *)malloc((MAX_LEN+1) * sizeof(char));
        strings = (char **)calloc(n,  sizeof(char *));

        read_line(regex);
        for(int i=0; i<n; ++i)
        {
                *(strings +i) = (char *)malloc((MAX_LEN+1) * sizeof(char));
                read_line(strings[i]);
        }

        printf("Matched Strings\n");
        for(int i=0; i<n; ++i)
        {
                init_memo();
                if(match(strings[i], regex, 0, 0))
                        printf("%s\n", strings[i]);
        }

        free(regex);
        for(int i=0; i<n; ++i)
```

```
                free(strings[i]);
        free(strings);

        return 0;
}
```

Solution

Question 4. (35 points) **Binary tree**

```
#define TRUE 1
#define FALSE 0

struct node {
    int data;
    struct node* left;
    struct node* right;
};

/*
 Helper function that allocates a new node
 with the given data and NULL left and right
 pointers.
*/
struct node* NewNode(int data) {
  struct node* node = (struct node *) malloc (sizeof(struct node));
  node->data = data;
  node->left = NULL;
  node->right = NULL;
  return node;
}

/*
 Given a node, return the length of the longest
 path to a leaf node (a node which does
 not have any children)
*/
int getNodeHeight(struct node* node){
    if (node == NULL) return 0;
    else{
        // find the maximum heights of right and
        // left subtrees. Calculate
        // the height of the tree rooted at node
        // using these two heights
        int leftHeight, rightHeight;
        leftHeight = getNodeHeight(node->left);
        rightHeight = getNodeHeight(node->right);
        if(leftHeight > rightHeight)
                return leftHeight + 1;
        else
                return rightHeight + 1;
    }
}

/*
 Given two trees, return true if they are
 structurally identical.
*/
```

```
int sameTree(struct node* a, struct node* b) {
  // both empty -> true
  if (a==NULL && b==NULL) return TRUE;
  // both non-empty -> compare them
  else if (a!=NULL && b!=NULL) {
    return
      (a->data == b->data &&
      sameTree(a->left, b->left) &&
      sameTree(a->right, b->right)
      );
  }
  // 3. one empty, one not -> false
  else return FALSE;
}

/*
 Change a tree so that the roles of the
 left and right pointers are swapped at every node.
 So the tree...
       4
      / \
     2   5
    / \
   1   3

 is changed to...
       4
      / \
     5   2
        / \
       3   1
*/
void mirror(struct node* node) {
  if (node==NULL) {
    return;
  }
  else {
    struct node* temp;

    // do the subtrees
    mirror(node->left);
    mirror(node->right);

    // swap the pointers in this node
    temp = node->left;
    node->left = node->right;
    node->right = temp;
  }
}

/*
 Given a tree and a sum, return true if there is a path from the root
 down to a leaf, such that adding up all the values along the path
```

```
 equals the given sum.
 Strategy: compute the sum value till you reach the leaf
 and then compare the computed value with required value
*/
int hasPathSum(struct node* node, int sumSoFar, int reqSum) {
  // return true if we run out of tree
  // and required sum reached
  if (node == NULL) {
    if (sumSoFar == reqSum) return 1;
    else return 0;
  }
  else {
  // otherwise check both subtrees

    sumSoFar =  sumSoFar + node->data;
    // check whether the left or right subtrees have
    // the required sum
    int isLeftSum = hasPathSum(node->left, sumSoFar, reqSum);
    int isRightSum = hasPathSum(node->right, sumSoFar, reqSum);

    if( isLeftSum || isRightSum )
        return 1;
    else
        return 0;
  }
}

/*
 Recursive helper function -- given a node, and an array containing
 the path from the root node up to but not including this node,
 print out all the root-leaf paths.
*/
void printPathsRecur(struct node* node, int path[], int lenSoFar) {
  if (node==NULL) {
        // we have reached the end of a path
        for(int i = 0; i < lenSoFar; ++i){
        printf("%d ", path[i]);
    }
    printf("\n");
    return;
  }

  // append this node to the path array
  path[lenSoFar] = node->data;
  lenSoFar++;

  // try both subtrees
    printPathsRecur(node->left, path, lenSoFar);
    printPathsRecur(node->right, path, lenSoFar);
}

/*
 Given a binary tree, print out all of its root-to-leaf
```

```
 paths, one per line. Uses a recursive helper to do the work.
*/
void printPaths(struct node* node) {
  int height = getNodeHeight(node);
  int path[height];
  printPathsRecur(node, path, 0);
}
```

Binary Search tree

```
/*
 Given a binary tree, return true if a node
 with the target data is found in the tree. Recurs
 down the tree, chooses the left or right
 branch by comparing the target to each node.
*/
int lookup(struct node* node, int target) {
  // Base case == empty tree
  // in that case, the target is not found so return false
  if (node == NULL) {
    return FALSE;
  }
  else {
    // see if found here
    if (target == node->data) return TRUE;
    else {
      // otherwise recur down the correct subtree
      if (target < node->data) return lookup(node->left, target);
      else return lookup(node->right, target);
    }
  }
}

/*
 Give a binary search tree and a number, inserts a new node
 with the given number in the correct place in the tree.
*/
struct node* insert(struct node* node, int data) {
  // If the tree is empty, return a new, single node
  if (node == NULL) {
    return NewNode(data);
  }
  else {
    // Otherwise, recur down the tree
    // and insert into the correct location
    if (data <= node->data) node->left = insert(node->left, data);
    else node->right = insert(node->right, data);

    return node;
  }
}

```

```c


/*
 Given a non-empty binary search tree,
 return the minimum data value found in that tree.
 Note that the entire tree does not need to be searched.
*/
int minValue(struct node* node) {
  struct node* current = node;
  // loop down to find the leftmost leaf
  while (current->left != NULL) {
    current = current->left;
  }

  return current->data;
}

/*
 Given a non-empty binary search tree,
 return the maximum data value found in that tree.
 Note that the entire tree does not need to be searched.
*/
int maxValue(struct node* node) {
  struct node* current = node;
  // loop down to find the leftmost leaf
  while (current->right != NULL) {
    current = current->right;
  }

  return current->data ;
}

/*
 Returns true if a binary tree is a binary search tree.
*/
int isBST(struct node* node) {
  if (node==NULL) return TRUE;
  // false if the max of the left is > than us

  if (node->left!=NULL && maxValue(node->left) > node->data)
    return FALSE;

  // false if the min of the right is <= than us
  if (node->right!=NULL && minValue(node->right) <= node->data)
    return FALSE;

  // false if, recursively, the left or right is not a BST
  if (!isBST(node->left) || !isBST(node->right))
    return FALSE;

  // passing all that, it's a BST
  return TRUE;
}
```

Question 5. Jim is practising programming these days. She has not seriously written code for a long time and is likely to make a lot of mistakes. She wants to implement some operations for linked lists. Help her in debugging the code.

Note: In the line number mentioned if there is no error then write "OK". If there is some error then write the correct code which should be a single statement. **Unnecessary edits will be penalized.**

```
1 struct node
2 {
3         int val;
4         struct node * next;
5 };
```

(a) (9 points) 3+2+1+2+1 Using the above structures, she wants to create a singly linked list. The variable head should point to the first node of the linked list. The linked list is empty when head = NULL. A new node will always be added at the end of linked list. The following function should add node at the end of a singly linked list and return the head of the resulting list.

```
1  /* Insert data at the end of linked list */
2  struct node * insert(int data, struct node * head)
3  {
4          struct node * q, * r;
5          struct node * tmp;
6          tmp -> val = data;
7          tmp -> next = NULL;
8          if(head == 0)
9          {
10                 head = tmp;
11         }
12         else
13         {
14             q = head;
15             r = head -> next;
16             while(r != NULL)
17             {
18                 q = r -> next;
19                 if((r = r -> next) && 0)
20                         r = q -> next;
21             }
22             q -> next = head;
23         }
24     return head;
25 }
```

Suspicion in	**Your response**
line 5	
line 8	
line 18	
line 19	
line 22	

(b) (4 points) 1+2+1 Assume a correct implementation of the insert node function.
Jim now wants to reverse the linked list. She writes a function which should return the head after reversing the linked list. Help her in debugging the following function.

```
1  struct node * reverse(struct node * head)
2  {
3          struct node *p1 = NULL, *p2 = head, *p3 = head -> next;
4          while(p2 != NULL)
5          {
6                  p2 -> next = p3;
7                  p1 = p2;
8                  p2 = p3;
9                  if((p3 == NULL))
10                         continue;
11                 p3 = p3 -> next;
12         }
13         head = p2;
14         return head;
15 }
```

Suspicion in	**Your response**
line 6	
line 10	
line 13	

(c) (6 points) 3+2+1 Jim now wants to write a function to reverse every group of k nodes in a singly linked list. It is not necessary that the number of nodes in the linked list should be a multiple of k i.e. the last group of nodes might have less than k nodes, you still have to reverse that group and return the head of

the resulting linked list.
Example : For the given linked list : $1 \to 2 \to 3 \to 4 \to 5 \to 6 \to 7$ and k = 2, the output should be $2 \to 1 \to 4 \to 3 \to 6 \to 5 \to 7$

```
1  struct node * kreverse(struct node * head, int k)
2  {
3          struct node * cur = head;
4          struct node * next = NULL;
5          struct node * prev = NULL;
6          int count = 0;
7          while(cur && count <= k)
8          {
9                  next = cur -> next;
10                 cur -> next = prev;
11                 prev = cur;
12                 cur = next;
13         }
14         if(next != NULL)
15         {
16                 head = kreverse(next, k);
17         }
18         return head;
19 }
```

Suspicion in	**Your response**
line 7	
line 16	
line 18	

(d) (11 points) 2+2+2+1+2+2 Jim now wants to write a function which will delete the p^{th} node from the end of linked list and returns head of the resulting linked list. p=1 means the last node of linked list.

```
struct node * removePthfromLast(struct node * head, int p)
{
        struct node *cur = head;
        struct node * H_cur;
        H_cur = &head;
        int i = 1;
        while(i < p)
        {
                i++;
                cur = cur -> next;
        }
        if(i < p)
                printf("Linked list has less than p nodes\n");
        else
        {
```

```
16                  while(cur != NULL)
17                  {
18                          H_cur = &(*H_cur -> next);
19                          cur = cur -> next;
20                  }
21                  *H_cur = *H_cur -> next;
22          }
23          return head;
24  }
```

Suspicion in	**Your response**
`line 4`	
`line 7`	
`line 12`	
`line 16`	
`line 18`	
`line 21`	

Part(a)

Suspicion in	Your response
line 5	* tmp = (struct node *)malloc(sizeof(struct node));
line 8	OK
line 18	q = r;
line 19	OK
line 22	q $\rightarrow$ next = tmp;

Part(b)

Suspicion in	Your response
line 6	p2 $\rightarrow$ next = p1
line 10	OK
line 13	head = p1

Part (c)

Suspicion in	Your response

line 7	while(cur && ((count = count+1) <= k))
	while(cur && (++count <= k))
	while(cur && (count++ < k))

line 16	head $\rightarrow$ next = kreverse(next,k); head $\rightarrow$ next = kreverse(curr,k);
line 18	return prev;

Part (d)

Suspicion in	**Your response**
`line 4`	struct node ** H_cur;
`line 7`	while(cur != NULL && i < p)
`line 12`	if(i < p || cur == NULL) if(cur == NULL)
`line 16`	while(cur → next != NULL)
`line 18`	H_cur = &((*H_cur) → next);
`line 21`	*H_cur = (*H_cur) → next;