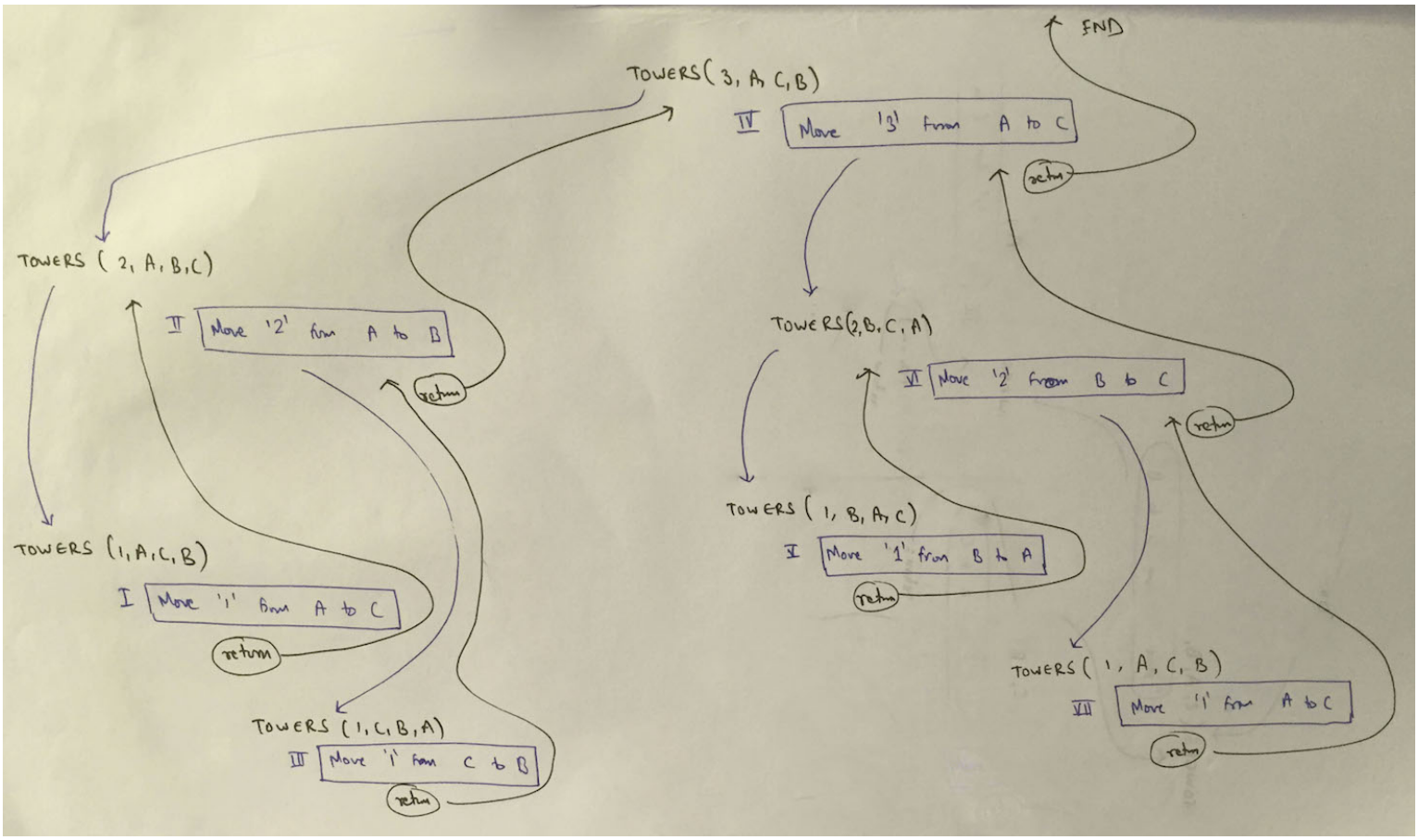
DS Assignment II

1. A) For the Towers of Hanoi problem, show the call tree during the recursive call Towers(3, A, C, B). In the tree, label the root node as Towers (3, A, C, B) while marking all the intermediate nodes. Show clearly the sequence of moves beginning from step1.

```
void Towers(int n, char *a, char *c, char *b) {
    if (n == 1)
      printf("\n\tMove Disk '%d' from Peg '%s' to Peg '%s'", n, a, c);
    else {
      Towers (n-1, a, b, c);
      printf("\n\tMove Disk '%d' from Peg '%s' to Peg '%s'", n, a, c);
      Towers (n-1, b, c, a);
    }
    return;
}
```



Full Sized Image

```
Output:
    Move Disk '1' from Peg 'A' to Peg 'C'
    Move Disk '2' from Peg 'A' to Peg 'B'
    Move Disk '1' from Peg 'C' to Peg 'B'
    Move Disk '3' from Peg 'A' to Peg 'C'
    Move Disk '1' from Peg 'B' to Peg 'A'
    Move Disk '2' from Peg 'B' to Peg 'C'
    Move Disk '1' from Peg 'A' to Peg 'C'

                                          ^ No need to copy this crap .
```

1. B) How many steps will be generated for a call of Towers (n, A, C, B) ? Justify your answer.

For 'n' pegs, the number of steps generated will be $2^n - 1$.

For one peg, the number of steps is exactly 1.
The recursive algorithm solves twice for "n-1" moves and prints once for the last n^{th} peg.
Hence the number of moves = 2 x 2 x 2 x … 2 (n times) - 1

2. Write the procedure for checking validity of an expression containing nested parenthesis. Give the stack content for [X/(Y-Z)+ D] and A * (B-C)} +D.

Loop through the expression in ascending order:

1. If an opening parenthesis '(', '{', or '[' is found, push it onto the stack.

2. If an closing parenthesis is found:

    2A. If the stack is not empty:

        2A1. If a matching closing parenthesis is at the top of the stack, pop the element from the top of the stack.

        2A2. If the parenthesis on the top of the stack does not match with the closing parenthesis, the expression is not balanced - bracket mismatch.

    2B. If the stack is empty, the expression is not balanced on the right.

If the stack is empty, the expression is balanced

Else the expression is not balanced on the left.

[X / (Y - Z) + D]

ELEMENT ENCOUNTERED	OPERATION	STACK CONTENT
[	PUSH	[
X		[
/		[
(	PUSH	[(
Y		[(
-		[(
Z		[(
)	CHECK, PASSED: POP	[
+		[
D		[
]	CHECK, PASSED: POP	

Stack empty at the end: Expression is balanced.

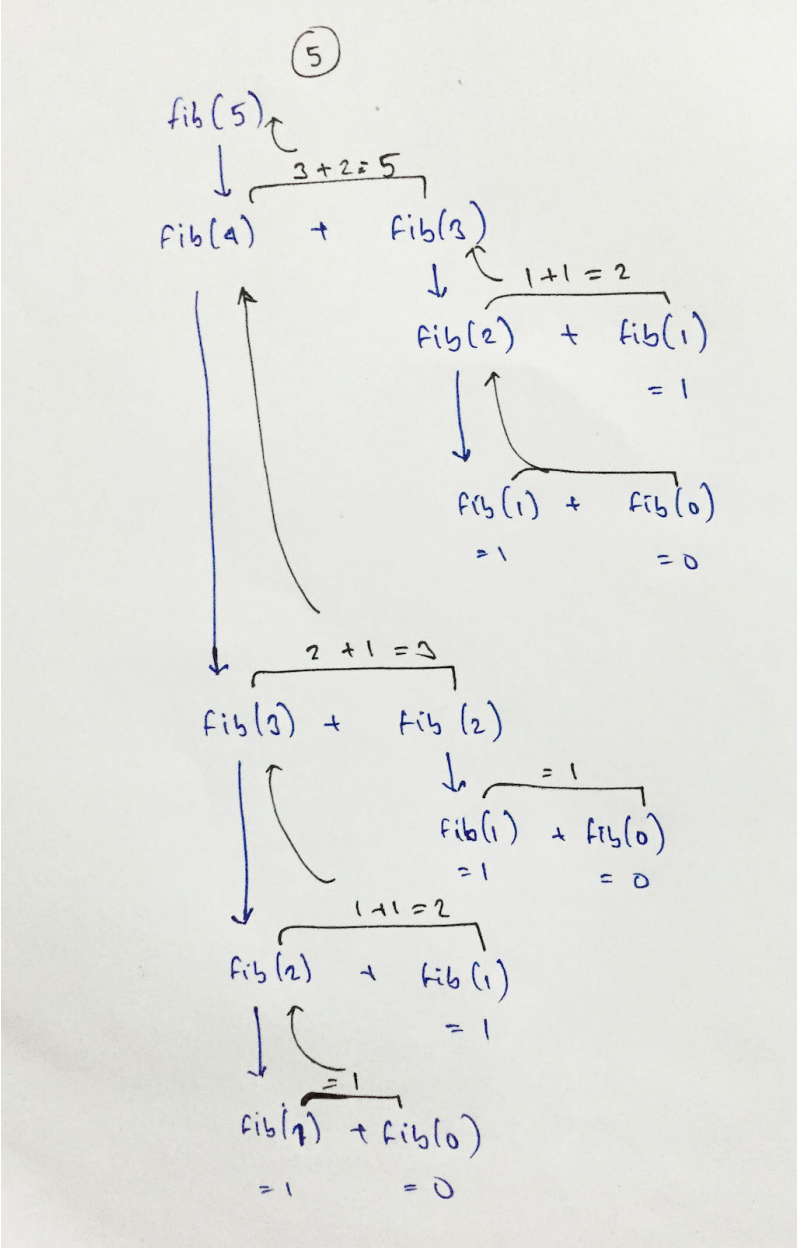
A * (B - C)} + D

ELEMENT ENCOUNTERED	OPERATION	STACK CONTENT
A		
*		
(	PUSH	(
B		(
-		(
C		(
)	CHECK, PASSED: POP	
}	STACK EMPTY UNDERFLOW	

Stack was empty when closing bracket was found, hence the expression is not balanced on the right.

3. Write recursive function to compute nth Fibonacci number. Show the status of the system stack after each function call to compute Fibonacci (5) i.e n=4. You need not show the stack frame itself for each function call. Simply add the name of the function and parameter to the stack to show its invocation and remove the name from the stack to show its termination.

```
long fibonacci (long n) {
      if (n == 0) return 0;
      if (n == 1) return 1;
      return fib(n - 1) + fib(n - 2);
}
```



4. Create a structure CircularQueue with integer pointer `arr` to store integer elements, `front` index of element to be deleted, `rear` index where the element is to be inserted, `capacity` the no of elements in the queue currently and max which is `max` size that the queue can have.

Write a program that contains the following:

a) A function Create a CircularQueue of max size(taken as input from the user) and initialize the variables appropriately.

b) A function to insert into the CircularQueue. All exceptions must be handled and variables updated.

c) A function to delete an element from the CircularQueue. All exceptions must be handled.

d) A function isFullQueue which checks if the CircularQueue is full and doubles the size of the queue(max) and updates all variables and data appropriately.

e) A function isEmptyQueue which checks if the CircularQueue is empty.

f) A main program to create a variable of CircularQueue and call all these functions.

```
//
//  CircularQueue.c
//  Circular Queue with size expanding option. (Version 2)
//
//  Note. While the older version works fine for small values, this one is
more efficient and short.
//

#include <stdio.h>
#include <stdlib.h>

#define UNDERFLOW_INT -32767

/// Boolean type, just for readability

typedef enum {
    NO = 0,
    YES = 1,
} BOOL;

typedef struct CircularQueue {
    int * arr;
    int front, rear, capacity, maxSize;
} CQUEUE_t;

typedef CQUEUE_t * CQUEUE_p_t;
```

```
// Queue methods

void createQueue (CQUEUE_p_t queue) {
    printf("\n\tEnter size of the circular queue: ");
    scanf("%d", &queue->maxSize);
    queue->arr = (int *)calloc(queue->maxSize, sizeof(int));
    queue->capacity = 0;
    queue->rear = queue->front = -1;
}

/**
 *  Check for overflow, if full, prepare to increase the size of the queue.
 *
 *  Increase the size of the queue using realloc.
 *
 *  We need to reset the queue only if the queue is full and front is greater
than rear (i.e. the queue is circularly arranged)
 *
 *  Get the current queue elements and put them in 'currentElements' array.
 *
 *  Set front to 0 and rear to capacity - 1
 *
 *  Put elements back in the queue, and increase the max size.
 *
 *  Correction: realloc() is frakking inefficient. Use free() and calloc()
instead.
 */

BOOL isFullQueue (CQUEUE_p_t queue) {
// Check for overflow

    if ((queue->front == queue->rear + 1) || (queue->front == 0 &&
queue->rear == queue->maxSize - 1)) {

// Get all the elements in a temp array

      int i, k;
      int currentElements[queue->capacity + 1];

      for (i = queue->front, k = 0; i < queue->capacity; ++i)
           currentElements[k++] = queue->arr[i];
      for (i = 0; i <= queue->rear; ++i)
           currentElements[k++] = queue->arr[i];


// Set front to 0 and rear to capacity - 1

      queue->front = 0;
      queue->rear = queue->capacity - 1;
```

```
// Free the memory of queue->arr, increase the maxSize, use calloc() to
increase the size

      free(queue->arr);
      queue->maxSize *= 2;
      queue->arr = (int *)calloc(queue->maxSize, sizeof(int));

// Put the elements back into the queue->arr.

      for (i = queue->rear; i >= 0; --i)
            queue->arr[i] = currentElements[--k];

      return YES;
    }
    return NO;
}

BOOL isEmptyQueue (CQUEUE_t queue) {
    if (queue.front == -1)
      return YES;
    return NO;
}

/**
 *  Insertions are done from the rear end.
 *
 *  If the queue is full, increase size and proceed.
 *
 *  If the queue is empty, set front and rear to 0 and insert at index 0.
 *
 *  If the rear end is at the end of the array, set rear to 0 (circulate) and
insert at index 0
 *
 *  Else increment rear end and insert at the position.
 *
 *  Increment the capacity.
 */

void insert (CQUEUE_p_t queue, int item) {
    if (isFullQueue(queue))
      printf("\nQueue overflow... expanding size...");

    if (isEmptyQueue(*queue))
      queue->front = queue->rear = 0;
    else
      queue->rear = (queue->rear + 1)%(queue->maxSize);

    *(queue->arr + queue->rear) = item;

    queue->capacity += 1;
}
```

```
/**
 *  Deletions are done from the front end.
 *
 *  Check for underflow, return underflow value if true.
 *
 *  Get the item at the front end.
 *
 *  If front == rear, i.e. there's only one element, set both to -1.
 *
 *  If front is at the end of the array, set front to 0.
 *
 *  Else increment front end.
 *
 *  Decrement the capacity.
 */

int delete (CQUEUE_p_t queue) {
    if (isEmptyQueue(*queue)) {
      printf("\nQueue Underflow!\n\n");
      return UNDERFLOW_INT;
    }

    int item = *(queue->arr + queue->front);

    if (queue->front == queue->rear)
      queue->front = queue->rear = -1;
    else
      queue->front = (queue->front + 1)%(queue->maxSize);

    queue->capacity -= 1;

    return item;
}

/**
 *  If the queue is empty, display appropiate message.
 *
 *  If rear end is greater the front end, print from the front end to the
rear end.
 *
 *  Else print from front to the end of the queue, then from starting of the
queue to the rear end.
 */

void display (CQUEUE_t queue) {
    if (isEmptyQueue(queue))
      printf("\nEmpty Queue.\n");
    else {
      printf("\nCurrent Queue [%d]: ", queue.capacity);
      int i;
```

```
      for (i = queue.front; i != queue.rear; i = (i+1)%(queue.maxSize))
            printf("\t%d", *(queue.arr + i));
      printf("\t%d\n", *(queue.arr + i));

      printf("\n");
    }
}


int main(int argc, const char * argv[]) {

    CQUEUE_t queue;

    createQueue(&queue);

    char choice;
    int item;

    do {
      printf("\n\t1. Insert\n\t2. Delete\n\t3. Display Queue.\n\tQ.
Quit\nEnter Choice : ");
      scanf(" %c", &choice);

      if (choice == '1') {
            printf("\tEnter item to be inserted: ");
            scanf("%d", &item);
            insert(&queue, item);
      }
      if (choice == '2') {
            item = delete(&queue);
            if (item != UNDERFLOW_INT)
                  printf("\tDeleted item: %d\n", item);
      }
      display(queue);

    } while (choice == '1' || choice == '2' || choice == '3');

    return 0;
}
```

5. Give an algorithm to convert an infix expression to postfix and using the same convert the given expression A+B*(C-D) EF/ (G+H) to its postfix equivalent expression by showing the stack content at each scan.

The algorithm goes as follows.

Parse the inputs one by one:

1. If the input is an operand, then place it in the output buffer/ stack.

2. If the input is an operator, push it into the operator stack.

3. While the stack is not empty and operator in stack has higher precedence than input operator, then pop the operator present in stack and add it to output buffer. Add the input operator to the stack.

4. If the input is an open brace, push it into the operator stack.

5. If the input is a close brace, pop elements in stack one by one until we encounter close brace. Discard braces while writing to output buffer/stack.

INPUT	OPERATOR STACK	OUTPUT BUFFER
A		A
+	+	A
B	+	A B
*	+ *	A B
(	+ * (	A B
C	+ * (	A B C
-	+ * (-	A B C
D	+ * (-	A B C D
)	+ *	A B C D -
$	+ * $	A B C D -
E	+ * $	A B C D - E
$	+ * $ $	A B C D - E
F	+ * $ $	A B C D - E F
/	+ * /	A B C D - E F $ $
(	+ * / (	A B C D - E F $ $
G	+ * / (	A B C D - E F $ $ G
+	+ * / (+	A B C D - E F $ $ G
H	+ * / (+	A B C D - E F $ $ G H
)	+ * /	A B C D - E F $ $ G H +
/0'		A B C D - E F $ $ G H + / * +

Sorry for the bad handwriting.
Here's some potatoes.