Easy difficulty (5 minutes)

```
#include <stdio.h>

// Functions
void printPyramid(int rows);

int main() {
   printPyramid(5);

   return 0;
}

/**
Function: print pyramid 'like' pattern, increment numbers throughtout
the positions of the pyramid
Parameters: rows - number of rows.
Comments:
Out: void
***/
void printPyramid(int rows){
   int i, j = 1; // counters
   int spaces = 1; // spaces and rows for user input
   int k, numb = 1;  // k and t for spaces

   spaces = rows + 4 - 1;  // initial number of spaces

   for (i = 1; i < rows; i++) {  // generate rows // BUG -> for (i = 1; i <=
rows; i++)
      for (k = spaces; k >= 1; k--) {  // print spaces before numbers
         printf(" ");
      }

      for (j = 1; j <= i; j++) {  // loop to print numbers based on the current
row.
         printf("%d ", numb++); // -> BUG
         numb++;
      }

      printf("\n");  // move to the next line for the next row.
      spaces--;  // decrement the number of spaces for the next row.
   }
}
```

Medium difficulty (10 minutes)

```
/*
This C program is designed to manage employee data. It dynamically allocates
memory for an array of employees, takes user input for each employee's ID, name,
and salary, updates the salaries by increasing them by 10% (due to the
intentional bug, this part won't work correctly), prints the employee details
before and after the salary update, calculates and prints the average salary of
all employees, and then frees the allocated memory.
 */

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

typedef struct {
    int id;
    char name[50];
    float salary;
} Employee;
```

```
void updateSalary(Employee *employee, float percent) {
    // Bug: Missing division by 100
    employee->salary *= (1 + percent);  // Bug: Missing division by 100
    //employee->salary *= (1 + percent / 100);
}

void printEmployeeDetails(const Employee *employee) {
    printf("ID: %d\n", employee->id);
    printf("Name: %s\n", employee->name);
    printf("Salary: $%.2f\n", employee->salary);
}

float calculateAverageSalary(const Employee *employees, int count) {
    float totalSalary = 0;

    for (int i = 0; i < count; i++) {
        totalSalary += employees[i].salary;
    }

    return totalSalary / count;
}

int main() {
    int employeeCount;
    printf("Enter the number of employees: ");
    scanf("%d", &employeeCount);

    if (employeeCount <= 0) {
        printf("Please enter a positive number of employees.\n");
        return 1;
    }

    Employee *employees = (Employee *)malloc(sizeof(Employee) * employeeCount);

    if (!employees) {
        printf("Memory allocation failed.\n");
        return 1;
    }

    for (int i = 0; i < employeeCount; i++) {
        printf("Enter details for employee %d:\n", i + 1);
        employees[i].id = i + 1;

        printf("Name: ");
        scanf("%s", employees[i].name);

        printf("Salary: $");
        scanf("%f", &employees[i].salary);

        printf("\n");
    }

    printf("Employee details before salary update:\n");
    for (int i = 0; i < employeeCount; i++) {
        printEmployeeDetails(&employees[i]);
    }

    // Update salaries
    for (int i = 0; i < employeeCount; i++) {
        updateSalary(&employees[i], 10.0);  // Bug: Missing division by 100
    }

    printf("\nEmployee details after salary update:\n");
```

```
    for (int i = 0; i < employeeCount; i++) {
        printEmployeeDetails(&employees[i]);
    }

    float averageSalary = calculateAverageSalary(employees, employeeCount);
    printf("\nAverage Salary: $%.2f\n", averageSalary);

    free(employees);

    return 0;
}
```

Hard difficulty (15 minutes)

```
/*This program uses a binary tree structure for displaying
      a tournament of tennis. The main function shows information
      about the winner, the structure of the tournament, the
      matches played, etc. It reads this information from a textfile
      with this format: [player][num_of_sets]
*/
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

#define MAX_NAME 20
#define STAGES 15
#define EMPTY NULL
#define NO_LINK NULL

// Define Structs
typedef struct _PLAYER{
    char name[MAX_NAME];
    int sets;
}PLAYER;

typedef struct _BTREE_NODE{
    void *data;
    struct _BTREE_NODE* left;
    struct _BTREE_NODE* right;
}BTREE_NODE;

// Define Macros
typedef BTREE_NODE* BTREE;
typedef enum _BTREE_LOCATION{BTREE_LEFT, BTREE_RIGHT} BTREE_LOCATION;
typedef enum _BOOLEAN {FALSE = 0, TRUE = 1} BOOLEAN;
typedef enum _STATUS {ERROR = 0, OK = 1} STATUS;

#define DATA(node) ((node)->data)
#define LEFT(node) ((node)->left)
#define RIGHT(node) ((node)->right)

// Functions
BTREE_NODE* newBTreeNode(void* data);
BTREE_NODE* addBTreeNode(BTREE_NODE* upnode, BTREE_NODE* node, BTREE_LOCATION
where);
BTREE_NODE* initNode(void* ptr_data, BTREE_NODE* node1,BTREE_NODE* node2);
BTREE_NODE* createBtree(void** v,int i,int size);
int bTreeSize(BTREE btree);
int bTreeDeep(BTREE btree);
BOOLEAN bTreeLeaf(BTREE_NODE* node);
BOOLEAN isLeaf(BTREE btree);
```

```
void printNode(BTREE btree, int space);
void printLeafs(BTREE btree);
void printTree(BTREE btree);
void bTreeFree(BTREE);

STATUS readPlayersFromFile(void** ,char* );

int QualifierS(BTREE);
int countTotalSets(BTREE);
int countWinnerSets(BTREE btree);
void printAllGames(BTREE);
void printWinnerGames(BTREE btree);


int main(){
      BTREE Btree;
      void* players [STAGES];
      char file_name [MAX_NAME];

      printf("Nome do ficheiro: ");
      scanf("%s", file_name);

      if (readPlayersFromFile(players, file_name))
      {
            Btree = createBtree(players, 0, STAGES);

            printf("\nParticipant List:\n");
            printLeafs(Btree);

            printf("\nGames List:\n");
            printAllGames(Btree);

            printf("\nNumber of qualifiers: %d", bTreeDeep(Btree) - 1);

            printf("\nNumber of Games: %d", bTreeSize(Btree) / 2);

            printf("\nNumber of Sets: %d", countTotalSets(Btree));

            printf("\nTournament Winner: %s\n", ((PLAYER*)Btree->data)->name);

            printf("\nGames played by the Tournament Winner: %d\n");
            printWinnerGames(Btree);

            printf("\nSets won by the Tournament Winner: %d\n",
countWinnerSets(Btree));

        printTree(Btree);

            bTreeFree(Btree);
      }
      else
            printf("Error Reading File\n");
    return 0;
}


/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BTREE_NODE* newBTreeNode(void* data){
```

```
    BTREE_NODE* tmp_pt;
    if((tmp_pt = (BTREE_NODE*)malloc(sizeof(BTREE_NODE))) != NULL){
        tmp_pt->data = data;
        tmp_pt->left = tmp_pt->right = NULL;
    }
    return tmp_pt;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BTREE_NODE* addBTreeNode(BTREE_NODE* upnode, BTREE_NODE* node, BTREE_LOCATION
where){
    BTREE_NODE* tmp_pt = upnode;
    if(where == BTREE_LEFT){
        if(upnode->left == NULL){
            upnode->left = node;
        } else {
            tmp_pt = NULL;
        }
    } else {
        if(upnode->right == NULL){
            upnode->right = node;
        } else {
            tmp_pt = NULL;
        }
    }
    return tmp_pt;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BTREE_NODE* initNode(void* ptr_data, BTREE_NODE* node1,BTREE_NODE* node2){
    BTREE_NODE *tmp_pt = NULL;
    tmp_pt = newBTreeNode(ptr_data);
    tmp_pt->left = node1;
    tmp_pt->right = node2;
    return tmp_pt;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BTREE_NODE* createBtree(void **v, int i, int size){
    if(i >= size){
        return NULL;
    } else {
            return(initNode(*(v + i), createBtree(v, 2 * i + 1, size),
createBtree(v, 2 * i + 2, size)));
    }
}
/*************************************************************
* Function:
*
```

```
* Parameters:
* Comments:
* Out:
*************************************************************/
int bTreeSize(BTREE btree){
    int count = 0;
    if(btree != NULL){
        count = 1 + bTreeSize(btree->left) + bTreeSize(btree->right);
    }
    return count;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
int bTreeDeep(BTREE btree){
    int deep = 0;
    int left, right;
    if(btree!= NULL){
        left = bTreeDeep(btree->left);
        right = bTreeDeep(btree->right);
        deep = 1 + ((left > right) ? left : right);
    }
    return deep;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BOOLEAN bTreeLeaf(BTREE_NODE* btree)
{
      if ((btree->left == NULL) && (btree->right == NULL))
            return(TRUE);
      else
            return(FALSE);
}/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
BOOLEAN isLeaf(BTREE btree)
{
      return ((btree->left == NULL) && (btree->right == NULL)) ? TRUE : FALSE;
}
/*************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*************************************************************/
void printNode(BTREE btree, int space)
{
  if(btree == NULL)
  {
    return ;
```

```
  }
  space += 10;

  printNode(btree->right, space);

  for(int i = 0; i < space; i++)
    {
      printf(" ");
    }
  printf("%s\n", ((PLAYER*)btree->data)->name);

  printNode(btree->left, space);
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
void printLeafs(BTREE btree)
{

      if (btree != NULL) {

            printLeafs(btree->left);
            if(isLeaf(btree)){
            printf("%s\n", ((PLAYER*)btree->data)->name);
            }
            printLeafs(btree->right);
      }
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
void printTree(BTREE btree)
{
  printNode(btree, 0);
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
void bTreeFree(BTREE btree)
{
      if (btree != NULL)
      {
            bTreeFree(btree->left);
            bTreeFree(btree->right);
            free(btree);
      }
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
```

```
* Out:
*************************************************************/
STATUS readPlayersFromFile(void** players, char* file_name)
{
      FILE* fp;
      int j, i = 0;
      void* ptr_data;
      if ((fp = fopen(file_name, "r")) != NULL)
      {
            while (!feof(fp))
            {
                  if ((ptr_data = malloc(sizeof(PLAYER))) != NULL)
                  {
                        fscanf(fp, "%[^;];", ((PLAYER*)ptr_data)->name);
                        fscanf(fp, "%d\n", &(((PLAYER*)ptr_data)->sets));
                        players[i] = ptr_data;
                        i++;
                  }
                  else
                  {
                        for (j = i; j >= 0; j--)
                              free(players[j]);
                        return(ERROR);
                  }
            }
            fclose(fp);
            return(OK);
      }
      else
            return(ERROR);
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
int Qualifiers(BTREE_NODE* BTREE)
{
      int count = 0;
      if (BTREE == NULL)
            return 0;
      if (BTREE != NULL) {
             count = 1 + Qualifiers(BTREE->left) + Qualifiers(BTREE->right);
             return count;
      }

}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
int countTotalSets(BTREE btree)
{
      int count = 0;
      while (btree != NULL) {
            countTotalSets(LEFT(btree));

            count = ((PLAYER*)btree->data)->sets + countTotalSets(btree->left) +
countTotalSets(btree->right);
```

```
            countTotalSets(btree->right);
            return count;
      }
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
int countWinnerSets(BTREE btree)
{
    int count = 0;
    BTREE BT=btree;
    count += ((PLAYER*)DATA(BT))->sets;
    if (btree != NULL && !bTreeLeaf(btree)) {
        if (!strcmp(((PLAYER*)DATA(btree->left))->name,
((PLAYER*)DATA((btree)))->name)){
            count +=  countWinnerSets(BT->left);
        }
        else
        {
            count += countWinnerSets(BT->right);
        }
    }
    return (count);
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
void printAllGames(BTREE btree)
{

            if ((btree) != NULL && !bTreeLeaf(btree))
            {
              printAllGames(btree->left);
              printAllGames(btree->right);
              printf("%s sets -> %d : %s sets -> % d => Winner : %s \n",
((PLAYER*)btree->left->data)->name, ((PLAYER*)btree->left->data)->sets,
((PLAYER*)btree->right->data)->name, ((PLAYER*)btree->right->data)->sets,
((PLAYER*)btree->data)->name);

            }
}
/*****************************************************************
* Function:
*
* Parameters:
* Comments:
* Out:
*****************************************************************/
void printWinnerGames(BTREE btree)
{
    if (btree != NULL && !bTreeLeaf(btree)) {
        printf("%s sets -> %d : %s sets -> % d => Winner : %s \n",
((PLAYER*)DATA(LEFT(btree)))->name, ((PLAYER*)DATA(LEFT(btree)))->sets,
((PLAYER*)DATA(RIGHT(btree)))->name, ((PLAYER*)DATA(RIGHT(btree)))->sets,
((PLAYER*)DATA(btree))->name);
```

```
        if (!strcmp(((PLAYER*)DATA(LEFT(btree)))->name,
((PLAYER*)DATA((btree)))->name))
            printWinnerGames(LEFT(btree));
        else
            printWinnerGames(RIGHT(btree));
    }
      return;
}
```

Torneio.txt

```
Jogador4;0
Jogador4;3
Jogador6;1
Jogador1;2
Jogador4;3
Jogador6;3
Jogador7;2
Jogador1;3
Jogador2;0
Jogador3;2
Jogador4;3
Jogador5;1
Jogador6;3
Jogador7;3
Jogador8;2
```