int checkOnesSequence(unsigned long data, int minSize) {

    int currentSequenceLength = 0;

    while (data > 0) {

        if (data & 1) {

            currentSequenceLength++;

            if (currentSequenceLength >= minSize) {

                return 1;

            }

        } else {

            currentSequenceLength = 0;

        }

        data >>= 1;

    }

    return 0;

}

Flight \*create\_flight(char \*text) {

    if(text == NULL) return NULL;

    Flight \*flight = malloc(sizeof(Flight));

    if(flight == NULL) return NULL;

    flight->desc = break\_flight(text, &flight->code, flight->start, flight->end);

    return flight;

}

void append\_flight(Flight \*flight, char \*duration) {

    if(flight == NULL || duration == NULL) return;

    flight->desc = realloc(flight->desc, strlen(flight->desc) + strlen(duration) + 4);

    if(flight->desc == NULL) return;

    strcat(flight->desc, " [");

    strcat(flight->desc, duration);

    strcat(flight->desc, "]");

}

char \*break\_flight(char \*text, int \*code, char \*start, char \*end) {

    if (text == NULL) return NULL;

    char \*temp = strdup(text);

    if (temp == NULL) return NULL;

    char \*token = strtok(replica, "|");

    if (token == NULL) {

        free(temp);

        return NULL;

    }

    \*code = atoi(token);

    token = strtok(NULL, "-");

    if (token == NULL) {

        free(temp);

        return NULL;

    }

    strcpy(start, token);

    token = strtok(NULL, "|");

    if (token == NULL) {

        free(temp);

        return NULL;

    }

    strcpy(end, token);

    token = strtok(NULL, "");

    if (token == NULL) {

        free(temp);

        return NULL;

    }

    char \*desc = strdup(token);

    free(temp);

    return desc;

}

int bstinsert (BstEvPtr \*\*rootAddr, Event \*event){

    if (rootAddr == NULL){ //

        \*rootAddr = malloc(sizeof(BstEvPtr));

        if(\*rootAddr== NULL) return 0;

        (\*rootAddr)->ptr = event;

        (\*rootAddr)->left = (\*rootAddr)->right = NULL;

        return 1;

    }

    int cmp = strcmp(event->desc, (\*rootAddr)->ptr->desc);

    if(cmp == 0) return 0; //Já existe

    if (cmp < 0) return bstinsert(&(\*rootAddr)->left, event);

    if (cmp > 0) return bstinsert(&(\*rootAddr)->right, event);

}

MAKEFILE :  
CC=gcc

CFLAGS= -g -Wall -lm

OBJ= programas para compilação

main: main.o $(OBJ)

$(CC) $(CFLAGS) $^ -o $@

%.o: %.c

$(CC) $(CFLAGS) -c $<

clean:

rm -f \*.o

rm -f main

gcc minArr.c min2.c test.c -c gcc minArr.o min2.o test.o -o test

int lSortAdd (LNode \*\*headPtr, DynFileInfo \*info, int (\*compare) (DynFileInfo \*e1, DynFileInfo \*e2)) {

    int comparison = \*headPtr == NULL ? 1 : compare((\*headPtr)->info, info);

    if (comparison == 0) return 0;

    if (comparison > 0) {

        LNode \*AuxNode = malloc(sizeof(LNode));

        if (AuxNode == NULL) {

            return -1;

        }

        AuxNode->info = info;

        AuxNode->next = \*headPtr;

        \*headPtr = AuxNode;

        return 1;

    } else {

        return lSortAdd(&((\*headPtr)->next), info, compare);

    }

    return 0;

}

T -> Função pública

t -> Função privada (static)

U -> Undefined (printf)

typedef struct listStaff{

  struct listStaff \*next;  // ligação na lista

  Member \*ref; // referência: ponteiro para o elemento de dados

} ListStaff;

void listInvert(ListStaff \*\*headAddr) {

    ListStaff \*prev = NULL;

    ListStaff \*current = \*headAddr;

    ListStaff \*next = NULL;

    while (current != NULL) {

        next = current->next;

        current->next = prev;

        prev = current;

        current = next;

    }

    \*headAddr = prev;

}

typedef struct{

  int number;        // Número de identificação

  char \*name;        // Nome (string alojada dinamicamente)

  char \*function;    // Cargo desempenhado (string alojada dinamicamente)

} Member;

typedef struct bstStaff{

  struct bstStaff \*left, \*right; // ligações na árvore

  Member \*ref;             // referência: ponteiro para o elemento de dados

} BstStaff;

int bstGetNumber(BstStaff \*root, char \*name) {

    if (root == NULL) return -1;

    int cmp = strcmp(name, root->ref->name);

    if (cmp == 0) return root->ref->number;

    if (cmp < 0) return bstGetNumber(root->left, name);

    return bstGetNumber(root->right, name);

}

int verifyWord(char \*word, char \*phrase) {

const char \*delimiters = " \t\n\r\v\f";

char \*token;

char phrase\_copy[strlen(phrase) + 1];

strcpy(phrase\_copy, phrase);

token = strtok(phrase\_copy, delimiters);

while (token != NULL) {

if (strcmp(word, token) == 0) {

return 1;

}

token = strtok(NULL, delimiters);

}

return 0;

}

Int (\* compare) (const void \* p1, const void \* p2){

[Struct name] \*ptr 1 = (Struct name\**) p1;*

[Struct name] \*ptr 2 = (Struct name\*) p2;

Return strcmp (ptr1->char, ptr2->char);

}

void activateBits (long long \*value, int step) {

    for (int i = 0; i < sizeof(\*value) \* CHAR\_BIT; i += step) {

        \*value |= 1LL << i;

    }

}

int countTrailingZeros(unsigned long long data) {

    int count = 0;

    int word\_size = sizeof(data) \* CHAR\_BIT;

    for (int i = 0; i < word\_size; ++i) {

        if ((data & 1) == 0) {

            count++;

        } else {

            break;

        }

        data >>= 1;

    }

    return count;

}

void mnemonic (char \*str) {

    int word = 0;

    int j = 0;

    for (int i = 0; str[i] != '\0'; i++) {

        if(isalpha(str[i]) && word == 0) {

            str[j++] = str[i];

            if(isspace(str[i+1])) {

                str[j++] = ' ';

                word = 1;

            }

        }

        if(word) {

            if (isspace(str[i]) && isalpha(str[i+1])) {

                str[j++] = toupper(str[i+1]);

            }

        }

    }

    str[j] = '\0';

}