ESCOLA SUPERIOR DE TECNOLOGIA - IPS

ANO LETIVO 2015 / 2016

ENGENHARIA DE INFORMÁTICA

ALGORITMOS E TIPO ABSTRATOS DE DADOS

PROFª ROSSANA SANTOS

MINI-PROJETO 2
READY-WEAR SOFT-WARE

MIGUEL FURTADO 120221006

Índice

Tipo de dados usados e respetiva Implementação

Stack Estático

Foi implementado 1 stack estáticos no projeto, sendo este para guardar clientes dentro de ficheiros.

Stack Estático .H

```
#ifndef STACK_H_INCLUDED
#define STACK_H_INCLUDED
#define N 20
#include "element.h"

struct stack
{
   TElem table[N];
   unsigned int size;
};

typedef struct stack *PtStackE;

PtStackE stackCreateEstatico(void);
int stackDestroyEstatico(PtStackE *pstack);
int stackPushEstatico(PtStackE pstack, TElem pelem);
int stackPopEstatico(PtStackE pstack, TElem *pelem);
int stackPeekEstatico(PtStackE pstack, TElem *pelem);
int stackIsEmptyEstatico(PtStackE pstack);
int stackIsFullEstatico(PtStackE pstack);
int stackSizeEstatico(PtStackE pstack, unsigned int *pelement);

#endif // STACK_H_INCLUDED
```

Stack Estático .C

```
#include <stdlib.h>
#include <stdio.h>
#include "stackEstatico.h"

#define OK         0
#define NULL_PTR    1
#define NO_STACK    2
#define STACK_EMPTY 4
#define STACK_FULL  5


PtStackE stackCreateEstatico()
{
```

```
   PtStackE myStack;

   if((myStack = (PtStackE) malloc (sizeof (struct stack))) == NULL)
      return NULL;

   myStack->size = 0;

   return myStack;
}

int stackDestroyEstatico(PtStackE *pstack)
{
   if(*pstack == NULL) return NO_STACK;
   free(*pstack);
   *pstack = NULL;

   return OK;
}

int stackPushEstatico(PtStackE pstack, TElem pelem)
{
   if(pstack == NULL) return NO_STACK;
   if(pstack->size == N) return STACK_FULL;

   pstack->table[pstack->size++] = pelem;
   return OK;
}

int stackPopEstatico(PtStackE pstack, TElem *pelem)
{
   if(pstack == NULL) return NO_STACK;
   if(pstack->size == 0) return STACK_EMPTY;
   if(pelem == NULL) return NULL_PTR;

   *pelem = pstack->table[--pstack->size];

   return OK;
}

int stackPeekEstatico(PtStackE pstack, TElem *pelem)
{
   if(pstack == NULL) return NO_STACK;
   if(pstack->size == 0) return STACK_EMPTY;
   if(pelem == NULL) return NULL_PTR;

   *pelem = pstack->table[pstack->size];
   return OK;
}
```

```
int stackIsEmptyEstatico(PtStackE pstack)
{
   if(pstack == NULL) return NO_STACK;
   if(pstack->size == 0) return STACK_EMPTY;

   return OK;
}

int stackIsFullEstatico(PtStackE pstack)
{
   if(pstack == NULL) return NO_STACK;
   if(pstack->size == N) return STACK_FULL;

    return OK;
}

int stackSizeEstatico(PtStackE pstack, unsigned int *pelement)
{
   if(pstack == NULL) return NO_STACK;
   if(pelement == NULL) return NULL_PTR;

   *pelement = pstack->size;
   if(pstack->size == 0) return STACK_EMPTY;

   return OK;
}
```

Stack Dinâmico

Foi implementado 1 stack dinamico no projeto, sendo este para guardar ficheiros

Stack Dinâmico .H

```
#ifndef STACKDINAMICO_H_INCLUDED
#define STACKDINAMICO_H_INCLUDED
#include "elementFicheiro.h"

typedef struct node *PtNode;

struct node
{
   TElemF *ptElem;
   PtNode ptPrev;
};

 struct stackd
{
   PtNode top;
   unsigned int size;
};

typedef struct stackd *PtStackD;

PtStackD stackCreateDinamico(void);                    /* criar um stack dinamico*/
int stackDestroyDinamico(PtStackD *pStack);            /* destruir um stack */
int stackPushDinamico(PtStackD pStack, TElemF pelem);  /* adicionar um elemento à
stack*/
int stackPopDinamico(PtStackD pStack, TElemF *pelem);  /* remover um elemento da
stack*/
int stackPeekDinamico(PtStackD pStack, TElemF *pelem);  /* mostra qual é o elemento no
top da stack*/
int stackIsEmptyDinamico(PtStackD pStack);             /* esta vazio*/
int stackIsFullDinamico(PtStackD pStack);              /* esta cheio*/
int stackSizeDinamico(PtStackD pStack, unsigned int *pnelem);   /* tamanha da stack*/

#endif // STACKDINAMICO_H_INCLUDED
```

Stack Dinâmico .C

```
#include <stdio.h>
#include <stdlib.h>
#include "stackDinamico.h"

#define OK         0
#define NULL_PTR    1
```

```
#define NULL_SIZE   2
#define NO_MEM      3
#define STACK_EMPTY 4
#define NO_STACK    5

PtStackD stackCreateDinamico()
{
   PtStackD myStack;

   if((myStack = (PtStackD) malloc (sizeof (struct stackd))) == NULL)
   {
      return NULL;
   }

   myStack->top = NULL;
   myStack->size = 0;

   return myStack;
}

int stackPushDinamico(PtStackD pStack, TElemF pelem)
{
   PtNode tmpNode;

   if(pStack == NULL) return NO_STACK;
   if((tmpNode = (PtNode)malloc(sizeof (struct node))) == NULL)return NO_MEM;
   if((tmpNode->ptElem = (TElemF *)malloc (sizeof (TElemF))) == NULL)
   {
      free(tmpNode);
      return NO_MEM;
   }

   *tmpNode->ptElem = pelem;
   tmpNode->ptPrev = pStack->top;

   pStack->top = tmpNode;
   pStack->size++;

   return OK;
}

int stackPopDinamico(PtStackD pStack, TElemF *pelem)
{
   PtNode tmpNode;

   if(pStack == NULL)return NO_STACK;
   if(pStack->top == NULL)return STACK_EMPTY;
   if(pelem == NULL)return NULL_PTR;
```

```
   *pelem = *pStack->top->ptElem;
   tmpNode = pStack->top;
   pStack->top = pStack->top->ptPrev;
   pStack->size--;
   free(tmpNode->ptElem);
   free(tmpNode);

   return OK;
}

int stackDestroyDinamico(PtStackD *ptStack)
{
   PtStackD tmpStack = *ptStack;
   PtNode tmpNode;

   if(tmpStack == NULL)return NO_STACK;

   while(tmpStack->top != NULL)
   {
      tmpNode = tmpStack->top;
      tmpStack->top = tmpStack->top->ptPrev;
      free(tmpNode->ptElem);
      free(tmpNode);
   }

   free(tmpStack);
   *ptStack = NULL;

   return OK;
}

int stackPeekDinamico(PtStackD pStack, TElemF *pelem)
{
   if(pStack == NULL)
   {
      return NO_STACK;

   }
   if(pStack->top == NULL)
   {
      return STACK_EMPTY;
   }
   if(pelem == NULL)
   {
      return NULL_PTR;
   }
```

```
   *pelem = *pStack->top->ptElem;

   return OK;
}

int stackIsEmptyDinamico (PtStackD pStack)
{

   if(pStack->top == NULL)
   {
      return 1;
   }
   return 0;
}

int stackSizeDinamico(PtStackD pstack, unsigned int *pnelem)
{
   if(pstack == NULL)
   {
      return NO_STACK;
   }
   if(pnelem == NULL)
   {
      return NULL_PTR;
   }

   *pnelem = pstack->size;

   return OK;
}
```

Queue Estático

A Queue foi usada de apoio para a implemetação a àrvore binaria.

Queue Estático.H

```
#ifndef FILAESTATICA_H_INCLUDED
#define FILAESTATICA_H_INCLUDED
#define MAX 100
#include "elementNo.h"

struct queue     /* definição da fila */
{
   TElemN *table;          /* área de armazenamento */
   unsigned int begin,end;         /* cabeça e cauda da fila */
   unsigned int nElem;
   unsigned int N;
};

typedef struct queue *PtQueue;

PtQueue queueCreate(int max);
int queueDestroy(PtQueue *pQueue);
int queueEnqueue(PtQueue pQueue, TElemN pelem);
int queueDequeue(PtQueue pQueue, TElemN *pelem);
int queuePeek(PtQueue pQueue, TElemN *pelem);
int queueIsEmpty(PtQueue pQueue);
int queueIsFull(PtQueue pQueue);
int queueSize(PtQueue pQueue, unsigned int *pNElem);

#endif // FILAESTATICA_H_INCLUDED
```

Queue Estático .C

```
#include <stdlib.h>
#include <stdio.h>
#include "filaEstatica.h"

#define OK         0
#define NO_QUEUE    1
#define NO_MEM      2
#define NULL_PTR    3
#define QUEUE_EMPTY 4
#define QUEUE_FULL  5

PtQueue queueCreate (int max)
{
   PtQueue queue;
   if(max < 1)
```

```
    max = MAX;

  if ((queue = (PtQueue) malloc (sizeof (struct queue))) == NULL)
    return NULL;          /* alocar memória para a fila */

  queue->table = calloc(max,sizeof(TElemN));
  queue->N = max;
  queue->begin = 0;     /* inicializar a cabeça da fila */
  queue->end = 0;       /* inicializar a cauda da fila */
  queue->nElem = 0;

  return queue;         /* devolver a referência da fila acabada de criar */
}

int queueDestroy (PtQueue *pqueue)
{
  PtQueue Tmpqueue = *pqueue;

  if (Tmpqueue == NULL) return NO_QUEUE;
  free (Tmpqueue);      /* libertar toda a memória ocupada pela fila */
  *pqueue = NULL;       /* colocar a referência da fila a NULL */
  return OK;
}

int queueEnqueue (PtQueue pqueue, TElemN elem)
{
  if (pqueue == NULL)
  {
    return NO_QUEUE;
  }

  if(queueIsFull(pqueue))
    return QUEUE_FULL;

  pqueue ->table[pqueue->end] = elem;
  pqueue -> end = (pqueue -> end+1) % pqueue->N;
  pqueue ->nElem++;
  return OK;

}

int queueDequeue (PtQueue pqueue, TElemN *pElem)
{
  if (pqueue == NULL)
  {
    return NO_QUEUE;
  }
```

```
   if(queueIsEmpty(pqueue))
      return QUEUE_EMPTY;
   if (pElem == NULL)
      return NULL_PTR;

   *pElem = pqueue ->table[pqueue->begin];
   pqueue -> begin = (pqueue -> begin+1) % pqueue->N;
   pqueue ->nElem--;
   return OK;
}

int queuePeek (PtQueue pqueue, TElemN *pElem)
{
   if (pqueue == NULL)
      return NO_QUEUE;

   if(queueIsEmpty(pqueue))
      return QUEUE_EMPTY;

   if (pElem==NULL)
      return NULL_PTR;

   *pElem = pqueue ->table[pqueue->begin];
   return OK;
}

int queueIsEmpty (PtQueue pqueue)
{
   if (pqueue == NULL)
      return NO_QUEUE;
   if (pqueue->nElem == 0)
      return 1;
   return 0;
}

int queueIsFull (PtQueue pqueue)
{
   if (pqueue == NULL)
      return NO_QUEUE;
   if (pqueue->nElem == pqueue->N)
      return 1;
   return 0;
}



int queueSize (PtQueue pqueue, unsigned int *pNElem)
{
```

```
    if (pqueue == NULL)
        return NO_QUEUE;

    if (pNElem == NULL)
        return NULL_PTR;

    *pNElem = pqueue->nElem;

    return 0;
}
```

Árvore Binária de Pequisa

Árvore Binária .H

```
#ifndef BINARYTREE_H_INCLUDED
#define BINARYTREE_H_INCLUDED
#include "element.h"

typedef struct abpnode *PtBSTnode;

struct abpnode
{
   PtBSTnode ptLeft;
   PtBSTnode ptRight;
   TElem *ptElem;
};

PtBSTnode BSTnodeCreate (TElem pelem);              /* criar um nó da árvore */
unsigned int BSTSize (PtBSTnode proot);             /* saber o tamanho da àrvore */
unsigned int BSTHeight (PtBSTnode proot);           /* saber a altura da àrvore */

PtBSTnode BSTSearchRec (PtBSTnode proot, TElem pelem) ; /* pesquisa recursiva */
PtBSTnode BSTMinRec (PtBSTnode proot);              /* pesquisa de minimo recursiva */
PtBSTnode BSTMaxRec (PtBSTnode proot);              /* pesquisa de máximo recursiva */

void BSTInsertRec (PtBSTnode *proot, TElem pelem);     /* inserção recursiva */
void BSTDeleteRecTotal (PtBSTnode *proot, TElem *pelem);/* remoção recursiva Total*/
PtBSTnode FindMin (PtBSTnode proot);                /* encontrar o minimo */
void BSTDestroy (PtBSTnode *proot);                 /* destruir a àrvore */
void nodeRemove (PtBSTnode *proot);
void BSTDeleteRec (PtBSTnode *proot, TElem *pelem);    /* remoção recursiva  sem apagar
elemento da memória*/

void BSTPreOrderRec (PtBSTnode proot);              /* travessia em pré-ordem recursiva */
void BSTInOrderRec (PtBSTnode proot);               /* travessia em ordem recursiva */
void BSTPostOrderRec (PtBSTnode proot);             /* travessia em pós-ordem recursiva */

void BSTByLevel (PtBSTnode proot);        /* mostrar em largura  - Precisa de confirmação, não
tenho a certeza se imprime*/
void BSTDisplay (PtBSTnode proot, unsigned int plevel); /* mostrar em profrundidade */

void writeElem(PtBSTnode node);


#endif // BINARYTREE_H_INCLUDED
```

Árvore Binária .C

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>
#include "binaryTree.h"
#include "filaEstatica.h"

void writeElem(PtBSTnode node)
{
   printf("%s, %d", node->ptElem->nome, node->ptElem->nVisitas);
}

PtBSTnode BSTnodeCreate (TElem pelem)        /* alocação do nó decomposto */
{
   PtBSTnode node; /* Criar um nó */

   if ((node = (PtBSTnode) malloc (sizeof (struct abpnode))) == NULL) /* Alocar o nó na
memória, se não for possível retorna NULL */
      return NULL;

   if ((node->ptElem = (TElem *) malloc (sizeof (TElem))) == NULL) /* Alocar o elemento na
memória, se não for possível liberta o nó anteriormene guardado e retorna NULL */
   {
      free (node);
      return NULL;
   }

   *node->ptElem = pelem;       /* copiar a informação */
   node->ptLeft = NULL; /* apontar para a esquerda para NULL */

   node->ptRight = NULL;          /* apontar para a direita para NULL */

   return node;
}

void BSTnodeDestroy (PtBSTnode *pelem)       /* libertação do nó decomposto */
{
   free ((*pelem)->ptElem);       /* libertação do elemento */
   free (*pelem);          /* libertação do nó */
   *pelem = NULL;         /* colocar o ponteiro a nulo */
}

unsigned int BSTSize (PtBSTnode proot)  /* cálculo do número de elementos recursiva */
{
   if (proot == NULL)
      return 0;    /* árvore vazia */
```

```
   else
      return 1 + BSTSize (proot->ptLeft) + BSTSize (proot->ptRight);
}

unsigned int BSTHeight (PtBSTnode proot)
{
   unsigned int leftHeight, rightHeight;

   if (proot == NULL)
      return 0;    /* nó externo no nível 0 */

   else
   {
      leftHeight = BSTHeight (proot->ptLeft);       /* subárvore esquerda */
      rightHeight = BSTHeight (proot->ptRight);   /* subárvore direita */

      if (leftHeight > rightHeight)
         return leftHeight + 1;

      else
         return rightHeight + 1;
   }
}

PtBSTnode BSTSearchRec (PtBSTnode proot, TElem pelem)   /* pesquisa recursiva */
{
   if (proot == NULL)
      return NULL;          /* pesquisa sem sucesso */

   if (strcmp(proot->ptElem->nome, pelem.nome) == 0)
      return proot;         /* pesquisa com sucesso */

   else if (strcmp (proot->ptElem->nome, pelem.nome) > 0)
      return BSTSearchRec (proot->ptLeft, pelem);

   else
      return BSTSearchRec (proot->ptRight, pelem);
}

PtBSTnode BSTMinRec (PtBSTnode proot)  /* pesquisa de mínimo recursiva */
{
   if (proot == NULL)
      return NULL;

   else if (proot->ptLeft == NULL)
      return proot;
```

```
   else
      return BSTMinRec (proot->ptLeft);
}

PtBSTnode BSTMaxRec (PtBSTnode proot)  /* pesquisa de máximo recursiva */
{
   if (proot == NULL)
      return NULL;

   else if (proot->ptRight == NULL)
      return proot;

   else
      return BSTMaxRec (proot->ptRight);
}

void BSTInsertRec (PtBSTnode *proot, TElem pelem)  /* inserção recursiva */
{
   if (*proot == NULL)
   {
      if ((*proot = BSTnodeCreate (pelem)) == NULL)
         return;
   }

   else if (strcmp((*proot)->ptElem->nome, pelem.nome) > 0)    /* subárvore esquerda */
      BSTInsertRec (&(*proot)->ptLeft, pelem);

   else if (strcmp((*proot)->ptElem->nome, pelem.nome) < 0)    /* subárvore direita */
      BSTInsertRec (&(*proot)->ptRight, pelem);

   else
      return;       /* o elemento já existe */
}

void nodeDelete (PtBSTnode *proot)
{
   PtBSTnode node = *proot;

   if ((*proot)->ptLeft == NULL && (*proot)->ptRight == NULL)
      BSTnodeDestroy (proot);    /* nó folha - eliminar o elemento */

   else if ((*proot)->ptLeft == NULL)       /* com subárvore direita */
   {
      *proot = (*proot)->ptRight;           /* ligar à direita */
      BSTnodeDestroy (&node);  /* eliminar o elemento */
   }

   else if ((*proot)->ptRight == NULL)     /* com subárvore esquerda */
```

```
  {
    *proot = (*proot)->ptLeft;  /* ligar à esquerda */
    BSTnodeDestroy (&node);  /* eliminar o elemento */
  }

  else   /* com subárvores direita e esquerda */
  {
    /* substituir pelo menor elemento da subárvore direita */
    *(*proot)->ptElem = *FindMin ((*proot)->ptRight)->ptElem;
    /* remover o menor elemento da subárvore direita */
    BSTDeleteRec (&(*proot)->ptRight, (*proot)->ptElem);
  }
}

PtBSTnode FindMin (PtBSTnode proot)
{
  PtBSTnode node = proot;

  while (node->ptLeft != NULL)
    node = node->ptLeft;

  return node;/* devolver um ponteiro para o elemento */
}

void BSTDeleteRecTotal (PtBSTnode *proot, TElem *pelem)      /* remoção recursiva */
{
  if (*proot == NULL)
    return;      /* árvore vazia ou elemento inexistente */

  if (strcmp((*proot)->ptElem->nome, pelem->nome) > 0)
    BSTDeleteRec (&(*proot)->ptLeft, pelem);

  else if (strcmp((*proot)->ptElem->nome, pelem->nome) < 0)
    BSTDeleteRec (&(*proot)->ptRight, pelem);

  else
  {
    *pelem = *(*proot)->ptElem;         /* copiar o elemento */
    nodeDelete(proot);           /* eliminar o elemento */
  }
}

void BSTDestroy (PtBSTnode *proot)
{
  if (*proot != NULL)
  {
    BSTDestroy (&(*proot)->ptLeft);     /* destruir a subárvore esquerda */
    BSTDestroy (&(*proot)->ptRight);   /* destruir a subárvore direita */
```

```
      BSTnodeDestroy (proot);    /* eliminar o elemento */
   }
}

void BSTPreOrderRec (PtBSTnode proot)  /* travessia em pré-ordem recursiva */
{
   if (proot != NULL)
   {
      writeElem(proot);  /* imprimir o conteúdo do elemento */
      BSTPreOrderRec (proot->ptLeft);
      BSTPreOrderRec (proot->ptRight);
   }
}

void BSTInOrderRec (PtBSTnode proot)  /* travessia em ordem recursiva */
{
   if (proot != NULL)
   {
      BSTInOrderRec (proot->ptLeft);
      writeElem(proot);  /* imprimir o conteúdo do elemento */
      BSTInOrderRec (proot->ptRight);
   }
}

void BSTPostOrderRec (PtBSTnode proot)  /* travessia em pós-ordem recursiva */
{
   if (proot != NULL)
   {
      BSTPostOrderRec (proot->ptLeft);
      BSTPostOrderRec (proot->ptRight);
      writeElem(proot);  /* imprimir o conteúdo do elemento */
   }
}

void BSTByLevel (PtBSTnode proot)
{
   PtBSTnode node;
   PtQueue queue;
   int nodesInLevel = 1;
   int altura = BSTHeight(proot);
   int alturaDouble = (double) altura;
   int numeroEspacos;

   FILE* arvore;
   arvore = fopen("arvore.txt", "w+");
   if(arvore == NULL)perror("arvore.txt");

   if (proot == NULL)
```

```
{
   printf("\nArvore vazia\n");
   return;       /* árvore vazia */
}
queue = queueCreate (100);

if ((queue == NULL)) return;

queueEnqueue (queue, proot);          /* armazenar a raiz */

while (!queueIsEmpty (queue))
{
   queueDequeue (queue, &node);     /* retirar o nó */

   numeroEspacos = pow(2, alturaDouble);

   while( numeroEspacos != 0)
   {
      printf(" ");
      numeroEspacos--;
   }
   alturaDouble--;

   printf("%s, %d", node->ptElem->nome, node->ptElem->nVisitas);
   fprintf(arvore,"%s, %d", node->ptElem->nome, node->ptElem->nVisitas);

   nodesInLevel--;

   if(nodesInLevel == 0)
   {
      printf("\n");
      fprintf(arvore,"\n");
   }

   /* armazenar a raiz da subárvore esquerda */
   if (node->ptLeft != NULL)
   {
      queueEnqueue (queue, node->ptLeft);
      nodesInLevel++;
   }

   /* armazenar a raiz da subárvore direita */
   if (node->ptRight != NULL)
   {
      queueEnqueue (queue, node->ptRight);
      nodesInLevel++;
   }
}
```

```
   queueDestroy (&queue);       /* destruir a fila */
   fclose(arvore);
}

void listInOrder (PtBSTnode proot, TElem plist[], unsigned int *pcount)
{
   if (proot != NULL)
   {
      listInOrder (proot->ptLeft, plist, pcount);    /* árvore esquerda */
      plist[(*pcount)++] = *proot->ptElem;          /* colocar elemento */
      listInOrder (proot->ptRight, plist, pcount);  /* árvore direita */
   }
}

void BSTDisplay (PtBSTnode proot, unsigned int plevel)
{
   unsigned int i;

   if (proot == NULL)
   {
      for (i = 0; i < plevel; i++) printf ("\t");
      printf ("*\n");
      return;
   }


   BSTDisplay (proot->ptRight, plevel + 1);

   for (i = 0; i < plevel; i++) printf ("\t");

   writeElem(proot);
   printf("\n");

   BSTDisplay (proot->ptLeft, plevel + 1);
}

void nodeRemove (PtBSTnode *proot)
{

   if ((*proot)->ptLeft == NULL)  /* com subárvore direita */
   {
      *proot = (*proot)->ptRight;            /* ligar à direita */
   }

   else if ((*proot)->ptRight == NULL)     /* com subárvore esquerda */
   {
      *proot = (*proot)->ptLeft;  /* ligar à esquerda */
```

```
   }

   else   /* com subárvores direita e esquerda */
   {
      /* substituir pelo menor elemento da subárvore direita */
      *(*proot)->ptElem = *FindMin ((*proot)->ptRight)->ptElem;
      /* remover o menor elemento da subárvore direita */
      BSTDeleteRec (&(*proot)->ptRight, (*proot)->ptElem);
   }
}

void BSTDeleteRec (PtBSTnode *proot, TElem *pelem)  /* remoção recursiva  sem apagar
elemento da memória*/
{
   if (*proot == NULL)
      return;       /* árvore vazia ou elemento inexistente */

   if (strcmp((*proot)->ptElem->nome, pelem->nome) > 0)
      BSTDeleteRec (&(*proot)->ptLeft, pelem);

   else if (strcmp((*proot)->ptElem->nome, pelem->nome) < 0)
      BSTDeleteRec (&(*proot)->ptRight, pelem);

   else
   {
      nodeDelete(proot);            /* eliminar o elemento */
   }
}
```

Lista

A lista foi implementada para guardar uma lista de clientes.

Lista .H

```
#ifndef LIST_H_INCLUDED
#define LIST_H_INCLUDED
#include "element.h"

typedef struct ptNode *PtNodeL;

struct ptNode /* definição de um nó da lista */
{
   TElem *ptElem; /* ponteiro para o elemento */
   PtNodeL ptPrev; /* ponteiro para o nó seguinte */
   PtNodeL ptNext; /* ponteiro para o nó seguinte */
```

```
};

struct list
{
   PtNodeL head;
   PtNodeL tail;
   unsigned int nElems;
};

typedef struct list *PtList;

PtList listCreate ();                                   /*Criar a lista*/
int listDestroy (PtList *ptList);                       /*Destruir a lista*/
int listAdd (PtList ptList, unsigned int rank, TElem elem);          /*Adicionar um element à
lista*/
int listSet (PtList ptList, unsigned int rank, TElem elemIn, TElem* pElemOut);/*Retirar um
elemento da lista e substituir por outro*/
int listRemove (PtList ptList, unsigned int rank, TElem* pElem);     /*Remover um
elemento da lista*/
int listGet (PtList ptList, unsigned int rank, TElem* pElem);        /*Obter um elemento da
lista*/
int listSize (PtList ptList);                           /*Tamanho da lista*/


#endif // LIST_H_INCLUDED
```

Lista .C

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

#define OK 0          /* operação realizada com sucesso */
#define NO_LIST -1    /* lista inexistente */
#define NO_MEM -2     /* memória esgotada */
#define NULL_PTR -3   /* ponteiro nulo */
#define LIST_EMPTY -4  /* lista vazia */
#define OUT_OF_RANK -5  /* fora do range da lista */


void destroyNode(PtNodeL ptNode)
{
   free(ptNode->ptElem);
   free(ptNode);
}

PtList listCreate ()
{
```

```
   PtList ptList;
   if ((ptList = (PtList) malloc (sizeof (struct list))) == NULL)
      return NULL; /* alocar memória */

   ptList->head = NULL; /* inicializar a cabeça */
   ptList->tail = NULL; /* inicializar a cauda */
   ptList->nElems = 0;

   return ptList; /* devolver a referência da lista acabada de criar */
}

int listDestroy (PtList *ptList)
{
   PtList tmptList = *ptList;
   PtNodeL tmpNode;

   if (tmptList == NULL) return NO_LIST;

   while (tmptList->head != NULL)
   {
      tmpNode = tmptList->head;
      tmptList->head = tmptList->head->ptNext;
      free (tmpNode->ptElem);
      free (tmpNode);
   }

   free (tmptList);
   *ptList = NULL;
   return OK;
}


PtNodeL ptNodeAtRank(PtList ptList, unsigned int rank)
{
   int i;

   if(rank < ptList->nElems/2)
   {
      PtNodeL ptNode = ptList->head;
      for (i = 0; i <= rank; i++)
         ptNode = ptNode->ptNext;

      return ptNode;
   }
   else
   {
      PtNodeL ptNode = ptList->tail;
      for (i = ptList->nElems - 1; i > rank; i--)
```

```
            ptNode = ptNode->ptPrev;

        return ptNode;
    }
}

int listAdd (PtList ptList, unsigned int rank, TElem elem)
{
    PtNodeL tmpNode;

    if (ptList == NULL) return NO_LIST;

    if(rank < 0 || rank > ptList->nElems)
        return OUT_OF_RANK;

    if ((tmpNode = malloc (sizeof (struct ptNode))) == NULL)
        return NO_MEM;

    if ((tmpNode->ptElem = malloc (sizeof (TElem)))== NULL)
    {
        free (tmpNode);
        return NO_MEM;
    }

    *tmpNode->ptElem = elem;
    tmpNode->ptPrev = NULL;
    tmpNode->ptNext = NULL;
    ptList->nElems++;

    if(ptList->nElems == 1)
    {
        ptList->head = tmpNode;
        ptList->tail = tmpNode;
    }

    if(rank == 0)
    {
        tmpNode->ptNext = ptList->head;
        tmpNode->ptNext->ptPrev = tmpNode;
        ptList->head = tmpNode;

        return OK;
    }

    if(rank == ptList->nElems - 1)
    {
        tmpNode->ptPrev = ptList->tail;
        tmpNode->ptPrev->ptNext = tmpNode;
```

```
        ptList->tail = tmpNode;

        return OK;
    }

    PtNodeL ptNextNode = ptNodeAtRank(ptList,rank);
    PtNodeL ptNodePrev = ptNextNode->ptPrev;

    tmpNode->ptPrev = ptNodePrev;
    tmpNode->ptNext = ptNextNode;

    ptNextNode->ptPrev = tmpNode;
    ptNodePrev->ptNext = tmpNode;

    return OK;
}

int listSet (PtList ptList, unsigned int rank, TElem elemIn, TElem* pElemOut)
{
    if (ptList == NULL) return NO_LIST;

    if(listSize(ptList) == 0) return LIST_EMPTY;

    if(rank < 0|| rank >= ptList->nElems) return OUT_OF_RANK;

    if(pElemOut == NULL) return NULL_PTR;

    PtNodeL ptNode =ptNodeAtRank(ptList, rank);
    *pElemOut = *ptNode->ptElem;

    *ptNode->ptElem = elemIn;

    return OK;
}

int listRemove (PtList ptList, unsigned int rank, TElem* pElem)
{
    if (ptList == NULL) return NO_LIST;

    if(rank < 0 || rank >= ptList->nElems) return OUT_OF_RANK;

    if(pElem == NULL) return NULL_PTR;

    PtNodeL ptNode = ptNodeAtRank(ptList,rank);

    *pElem = *ptNode->ptElem;
    ptList->nElems--;
```

```
    if(ptList->nElems == 0)
    {
        ptList->head = NULL;
        ptList->tail = NULL;
        destroyNode(ptNode);
    }

    PtNodeL ptNodePrev = ptNode->ptPrev;
    PtNodeL ptNodeNext = ptNode->ptNext;

    if(rank == ptList->nElems)
    {
        ptList->tail = ptNodePrev;
        ptNodePrev->ptNext = ptNodeNext;
    }

    else if(rank == 0)
    {
        ptNodeNext->ptPrev = ptNodePrev;
        ptList->head = ptNodeNext;
    }
    else
    {
        ptNodePrev->ptNext = ptNodeNext;
        ptNodeNext->ptPrev = ptNodePrev;
    }
    destroyNode(ptNode);
    return OK;
}

int listGet (PtList ptList, unsigned int rank, TElem* pElem)
{
    if (ptList == NULL) return NO_LIST;

    if(listSize(ptList) == 0) return LIST_EMPTY;

    if(rank < 0|| rank >= ptList->nElems) return OUT_OF_RANK;

    if(pElem == NULL) return NULL_PTR;

    PtNodeL ptNode = ptNodeAtRank(ptList, rank);
    *pElem = *ptNode->ptElem;

    return OK;
}
int listSize (PtList ptList)
{
    if(ptList == NULL) return NO_LIST;
```

```
    return ptList->nElems;
}
```

Complexidade Algorítmica

Inserção de um novo registo na árvore

Sempre que um elemento é adicionado à lista de clientes da loja, é também adicionado à árvore. A outra única situação que este é inserido sem ser quando ainda não existe o cliente, é quando o cliente foi removido posteriormente da árvore por já não entrar na loja à mais de 30 dias, como tal independentemente de estar na lista de clientes terá que ser adicionado novamente à árvore. Para adicionar um elemento à árvore é usado o método BSTInsertRec que existe no header da binaryTree, em que a complexidade deste é de **O(n).** Este método é chamado dentro de um outro método, logo a complexidade é de **O(n^2).**

Algoritmo:

```
void gerirListaClientes(Loja loja, PtStackE stackClientes)
{
   PtStackE aux = stackCreateEstatico();

   Cliente clienteAux = criarCliente();
   Cliente aux2 = criarCliente();

   while(!stackIsEmptyEstatico(stackClientes))
   {
      stackPopEstatico(stackClientes, &clienteAux);
      stackPushEstatico(aux, clienteAux);

      if(listSize(loja->lista) == 0)
      {
         decompoeClienteNaoExiste(&clienteAux, loja);
         listAdd(loja->lista, loja->lista->nElems, clienteAux);
         loja->clienteNo = BSTnodeCreate(clienteAux);
         strcpy(aux2.nome, clienteAux.nome);
      }

      else
      {
         if(encontrarCliente(loja, clienteAux, aux2) == 0)
         {
            decompoeClienteNaoExiste(&clienteAux, loja);
            listAdd(loja->lista, loja->lista->nElems, clienteAux);
            BSTInsertRec(&loja->clienteNo, clienteAux);
            strcpy(aux2.nome, clienteAux.nome);
         }
         else
         {
            strcpy(aux2.nome, clienteAux.nome);
         }
      }
```

```
    }

    while(!stackIsEmptyEstatico(aux))
    {
        stackPopEstatico(aux, &clienteAux);
        stackPushEstatico(stackClientes, clienteAux);
    }

    stackDestroyEstatico(&aux);
    confirmarAtividade(loja);
}


int encontrarCliente(Loja loja, Cliente cliente, Cliente global)
{
    PtListIterator ptIt = listIteratorCreate(loja->lista);
    Cliente *clienteAux;
    PtBSTnode clienteNo;



    if(strcmp(cliente.nome, global.nome) != 0)
    {
        if(listSize(loja->lista) == 1)
        {
            clienteAux = nextElem(ptIt);
            clienteNo = BSTSearchRec(loja->clienteNo, cliente);

            if(strcmp(clienteAux->nome, cliente.nome) == 0)
            {
                if(cliente.valorDeCompras == 0)
                {
                    clienteAux->nVisitas++;

                    if(clienteAux->estado == 1)
                    {
                        clienteNo->ptElem->nVisitas++;
                    }

                    loja->nrClienteVisitaram++;
                    loja->nrClientesSemCompras++;

                    clienteAux->dia = loja->dataLoja->dia;
                    clienteAux->mes = loja->dataLoja->mes;
                    clienteAux->ano = loja->dataLoja->ano;
                }
                else
                {
```

```
            clienteAux->nVisitas++;
            if(clienteAux->estado != 1)
            {
               BSTInsertRec(&loja->clienteNo, cliente);
               clienteAux->estado = 1;
               clienteNo = BSTSearchRec(loja->clienteNo, cliente);
            }

            if(clienteAux->estado == 1)
            {
               clienteNo->ptElem->nVisitas++;
            }

            clienteAux->nVisitasComCompras++;
            clienteAux->valorDeCompras = clienteAux->valorDeCompras +
cliente.valorDeCompras;
            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
            loja->nrClienteVistaramCompras++;
         }
         return 1;
      }
   }

   while(hasNextElem(ptIt))
   {
      clienteAux = nextElem(ptIt);
      clienteNo = BSTSearchRec(loja->clienteNo, cliente);

      if(strcmp(clienteAux->nome, cliente.nome) == 0)
      {
         if(cliente.valorDeCompras == 0)
         {
            clienteAux->nVisitas++;

            if(clienteAux->estado == 1)
            {
               clienteNo->ptElem->nVisitas++;
            }

            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
```

```
            loja->nrClientesSemCompras++;
          }
          else
          {
            clienteAux->nVisitas++;

             if(clienteAux->estado != 1)
            {
              BSTInsertRec(&loja->clienteNo, cliente);
              clienteAux->estado = 1;
              clienteNo = BSTSearchRec(loja->clienteNo, cliente);
            }

            if(clienteAux->estado == 1)
            {
              clienteNo->ptElem->nVisitas++;
            }

            clienteAux->nVisitasComCompras++;
            clienteAux->valorDeCompras = clienteAux->valorDeCompras +
cliente.valorDeCompras;
            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
            loja->nrClienteVistaramCompras++;
          }
          return 1;
        }
      }
    }
    else
    {
      if(listSize(loja->lista) == 1)
      {
        clienteAux = nextElem(ptIt);

        if(strcmp(clienteAux->nome, cliente.nome) == 0)
        {
          clienteAux->valorDeCompras = clienteAux->valorDeCompras +
cliente.valorDeCompras;

          return 1;
        }
      }

      while(hasNextElem(ptIt))
```

```
    {
      clienteAux = nextElem(ptIt);

      if(strcmp(clienteAux->nome, cliente.nome) == 0)
      {
        clienteAux->valorDeCompras = clienteAux->valorDeCompras +
cliente.valorDeCompras;

        return 1;
      }
    }
  }
  return 0;
}
```

Remoção de um registo da árvore

A remoção de um registo da árvore acontece quando um cliente já não visita a loja à 30 dias. Para este efeito foi criado um método que confirma a atividade dos clientes e que conforme se estes têm visitado a loja ou não, vai mudar o seu estado para 0, o valor de compras para 0 e vai remover da árvore de registo. Este algoritmo tem como complexidade **O(n).**

Algoritmo:

```
void confirmarAtividade(Loja loja)
{
  PtListIterator ptIt = listIteratorCreate(loja->lista);
  Cliente *aux;
  int diasDescontar;

  while(hasNextElem(ptIt))
  {
    diasDescontar = 30;
    aux = nextElem(ptIt);
    if(aux->mes == loja->dataLoja->mes)
    {
      if((aux->dia - loja->dataLoja->dia) >= 30)
      {
        aux->estado = 0;
        aux->valorDeCompras = 0;
        BSTDeleteRec(&loja->clienteNo, aux);
      }
      else
      {
        continue;
      }
```

```
        }
        else
        {
            if((loja->dataLoja->mes - aux->mes) >= 2)
            {
                aux->estado = 0;
                aux->valorDeCompras = 0;
                BSTDeleteRec(&loja->clienteNo, aux);
            }
            else
            {
                if(loja->dataLoja->mes & 1)
                {
                    diasDescontar = diasDescontar - loja->dataLoja->dia;
                    if(aux->dia < (30 - diasDescontar))
                    {
                        aux->estado = 0;
                        aux->valorDeCompras = 0;
                        BSTDeleteRec(&loja->clienteNo, aux);
                    }
                    else
                    {
                        continue;
                    }
                }
                else
                {
                    diasDescontar = diasDescontar - loja->dataLoja->dia;
                    if(aux->dia < (31 - diasDescontar))
                    {
                        aux->estado = 0;
                        aux->valorDeCompras = 0;
                        BSTDeleteRec(&loja->clienteNo, aux);
                    }
                    else
                    {
                        continue;
                    }
                }
            }
        }
    }
}
```

Inserção de uma nova visita à loja de um cliente

Para a inserção de uma visita à loja foi usado um método que também foi usado na inserção de um novo registo na árvore. A complexidade algorítmica deste é de **O(n),** pois de cada vez que regista um cliente a partir do ficheiro ele fez uma visita então altera nos dados do Cliente e como o cliente pode estar ou não registado na lista, pode implicar se é preciso percorrer toda a lista de clientes para encontrar o cliente certo e aumentar o número de visitas ou não.

Algoritmo:

```
void gerirListaClientes(Loja loja, PtStackE stackClientes)
{
   PtStackE aux = stackCreateEstatico();

   Cliente clienteAux = criarCliente();
   Cliente aux2 = criarCliente();

   while(!stackIsEmptyEstatico(stackClientes))
   {
      stackPopEstatico(stackClientes, &clienteAux);
      stackPushEstatico(aux, clienteAux);

      if(listSize(loja->lista) == 0)
      {
         decompoeClienteNaoExiste(&clienteAux, loja);
         listAdd(loja->lista, loja->lista->nElems, clienteAux);
         loja->clienteNo = BSTnodeCreate(clienteAux);
         strcpy(aux2.nome, clienteAux.nome);
      }

      else
      {
         if(encontrarCliente(loja, clienteAux, aux2) == 0)
         {
            decompoeClienteNaoExiste(&clienteAux, loja);
            listAdd(loja->lista, loja->lista->nElems, clienteAux);
            BSTInsertRec(&loja->clienteNo, clienteAux);
            strcpy(aux2.nome, clienteAux.nome);
         }
         else
         {
            strcpy(aux2.nome, clienteAux.nome);
         }
      }
   }

   while(!stackIsEmptyEstatico(aux))
   {
      stackPopEstatico(aux, &clienteAux);
      stackPushEstatico(stackClientes, clienteAux);
   }

   stackDestroyEstatico(&aux);
```

```
   confirmarAtividade(loja);
}

void decompoeClienteNaoExiste(Cliente *cliente, Loja loja)
{
   cliente->id = loja->lista->nElems + 1;

   if(cliente->valorDeCompras == 0)
   {
      cliente->nVisitas++;

      loja->nrClienteVisitaram++;
      loja->nrClientesSemCompras++;
   }
   else
   {
      cliente->nVisitas++;
      cliente->nVisitasComCompras++;

      loja->nrClienteVisitaram++;
      loja->nrClienteVistaramCompras++;
   }
}

int encontrarCliente(Loja loja, Cliente cliente, Cliente global)
{
   PtListIterator ptIt = listIteratorCreate(loja->lista);
   Cliente *clienteAux;
   PtBSTnode clienteNo;



   if(strcmp(cliente.nome, global.nome) != 0)
   {
      if(listSize(loja->lista) == 1)
      {
         clienteAux = nextElem(ptIt);
         clienteNo = BSTSearchRec(loja->clienteNo, cliente);

         if(strcmp(clienteAux->nome, cliente.nome) == 0)
         {
            if(cliente.valorDeCompras == 0)
            {
               clienteAux->nVisitas++;

               if(clienteAux->estado == 1)
               {
                  clienteNo->ptElem->nVisitas++;
               }

               loja->nrClienteVisitaram++;
               loja->nrClientesSemCompras++;
```

```
            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;
        }
        else
        {
            clienteAux->nVisitas++;
            if(clienteAux->estado != 1)
            {
                BSTInsertRec(&loja->clienteNo, cliente);
                clienteAux->estado = 1;
                clienteNo = BSTSearchRec(loja->clienteNo, cliente);
            }

            if(clienteAux->estado == 1)
            {
                clienteNo->ptElem->nVisitas++;
            }

            clienteAux->nVisitasComCompras++;
            clienteAux->valorDeCompras = clienteAux->valorDeCompras + cliente.valorDeCompras;
            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
            loja->nrClienteVistaramCompras++;
        }
        return 1;
    }
}

while(hasNextElem(ptIt))
{
    clienteAux = nextElem(ptIt);
    clienteNo = BSTSearchRec(loja->clienteNo, cliente);

    if(strcmp(clienteAux->nome, cliente.nome) == 0)
    {
        if(cliente.valorDeCompras == 0)
        {
            clienteAux->nVisitas++;

            if(clienteAux->estado == 1)
            {
                clienteNo->ptElem->nVisitas++;
            }

            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
```

```
            loja->nrClientesSemCompras++;
        }
        else
        {
            clienteAux->nVisitas++;

             if(clienteAux->estado != 1)
            {
                BSTInsertRec(&loja->clienteNo, cliente);
                clienteAux->estado = 1;
                clienteNo = BSTSearchRec(loja->clienteNo, cliente);
            }

            if(clienteAux->estado == 1)
            {
                clienteNo->ptElem->nVisitas++;
            }

            clienteAux->nVisitasComCompras++;
            clienteAux->valorDeCompras = clienteAux->valorDeCompras + cliente.valorDeCompras;
            clienteAux->dia = loja->dataLoja->dia;
            clienteAux->mes = loja->dataLoja->mes;
            clienteAux->ano = loja->dataLoja->ano;

            loja->nrClienteVisitaram++;
            loja->nrClienteVistaramCompras++;
        }
        return 1;
      }
    }
}
else
{
    if(listSize(loja->lista) == 1)
    {
      clienteAux = nextElem(ptIt);

      if(strcmp(clienteAux->nome, cliente.nome) == 0)
      {
        clienteAux->valorDeCompras = clienteAux->valorDeCompras + cliente.valorDeCompras;

        return 1;
      }
    }

    while(hasNextElem(ptIt))
    {
      clienteAux = nextElem(ptIt);

      if(strcmp(clienteAux->nome, cliente.nome) == 0)
      {
        clienteAux->valorDeCompras = clienteAux->valorDeCompras + cliente.valorDeCompras;
```

```
            return 1;
          }
        }
      }
    return 0;
}
```

Determinação do género que mais visita a loja

A complexidade algorítmica deste é de **O(n)**, pois dependendo do número de clientes registados na lista teremos que percorrer mais ou menos clientes.

Algoritmo:

```
void generoQueMaisVisitou(Loja loja)
{
   PtListIterator ptIt = listIteratorCreate(loja->lista);
   Cliente *aux;
   int masculino = 0, feminino = 0;

   while(hasNextElem(ptIt))
   {
      aux = nextElem(ptIt);
      if(strcmp(aux->genero, "masculino") == 0)
         masculino++;
      else
         feminino++;
   }

   if(masculino > feminino)
      strcpy(loja->generoMaisVisitou, "masculino");
   else if(masculino < feminino)
      strcpy(loja->generoMaisVisitou, "feminino");
   else
      strcpy(loja->generoMaisVisitou, "igual");
}
```

Conclusão

Com o desenvolvimento deste mini-projeto foi-me possível aplicar muitos dos conteúdos lecionados durante as aulas.

Uma das grandes dificuldades sentidas e que não foi concluída no desenvolvimento deste projeto foi a impressão da árvore em consola ou em ficheiro. Apesar das inúmeras tentativas este foi um grande desafio que precisava de mais tempo e talvez de mais uma cabeça com uma perspetiva diferente para me ajudar a resolver e chegar ao resultado desejado.

Apesar desta grande dificuldade, todos os outros objetivos foram alcançados com sucesso e com este novo conhecimento para o futuro irá ser possível resolver todo o tipo de situações semelhantes.