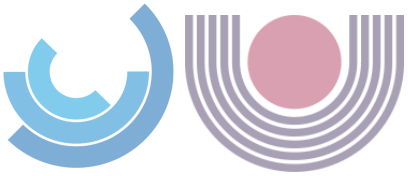


Universidade Estadual do Oeste do Paraná
Bacharelado em Ciência da Computação

Atividade Prática 02

Gabriel Norato Claro, Lionel Rodel, Maria Eduarda Crema Carlos
gabriel.claro@unioeste.br, lionel.rodel@unieste.br, maria.carlos3@unioeste.br

Cascavel, 2022
08/08/2022

1. Apresentação

A atividade prática 02 contém três códigos, grafo TAD, classe fila com herança de lista encadeada e método de ordenação MergeSort. Com o intuito de praticar grafos, programação orientada a objetos e métodos de ordenação

O github de acesso aos códigos fontes é *https://github.com/iMaGiNaTrOn/Grafos-Fila-Ordenacao*.

2. Métodos

2.1. Grafo

De maneira mais formal, podemos dizer que um grafo é um terno (V, A, f), onde V e A são conjuntos finitos arbitrários e f é a função que associa a cada elemento de A um par ordenado de elementos de V. O código está escrito e comentado nos arquivos “grafo.c” e “funcoes-grafo.h”. A seguir temos o código fonte do grafo TAD.

2.1.1. “grafo.c”

```
#include <stdio.h>
#include <stdlib.h>
#include "funcoes-grafo.h"

// Função principal
int main()
{
    // Cria um grafo
    int V = 5;
    Grafo *grafo = createGrafo(V);
    addEdge(grafo, 0, 1);
    addEdge(grafo, 0, 4);
    addEdge(grafo, 1, 2);
    addEdge(grafo, 1, 3);
    addEdge(grafo, 1, 4);
    addEdge(grafo, 2, 3);
    addEdge(grafo, 3, 4);

    printf("Grafo\n");
    // printa o grafo
    printGrafo(grafo);

    printf("\nChecagem de existência de arestas\n");
    checkEdge(0, 1, grafo);
    checkEdge(0, 3, grafo);
    checkEdge(5, 1, grafo);

    printf("\nRemovendo aresta de um grafo\n");
```

```
    printf("No nodo 1, valor 3 deve ser removido");
    removeEdge(1, 3, grafo);

    printf("\nGrafo atualizado\n");
    printGrafo(grafo);

    printf("\nRemovendo todo o grafo\n");
    removeGraph(grafo);

    printGrafo(grafo);

    return 0;
}
```

2.1.2. “funcoes-grafo.h”

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

// Nodo de uma lista de adjacência
typedef struct Node
{
    int dest;
    struct Node *next;
} Node;

// Estrutura de uma lista de adjacência
typedef struct AdjList
{
    Node *head;
} AdjList;

// Grafo como uma lista
typedef struct Grafo
{
    int V;
    AdjList *array;
} Grafo;

// Função para criar uma nova lista de adjacência
Node *newAdjListNode(int dest)
{
    Node *newNode = (Node *)malloc(sizeof(Node));
    newNode->dest = dest;
    newNode->next = NULL;
    return newNode;
}
```

```
// Função que cria grafo com V vertices
Grafo *createGrafo(int V)
{
    Grafo *grafo = (Grafo *)malloc(sizeof(Grafo));
    grafo->V = V;

    // Cria um array de listas de adjacência
    grafo->array = (AdjList *)malloc(V * sizeof(AdjList));

    // Inicializa cada lista com a head nula
    int i;
    for (i = 0; i < V; ++i)
        grafo->array[i].head = NULL;

    return grafo;
}

// Adiciona uma aresta
void addEdge(Grafo *grafo, int src, int dest)
{
    // Adiciona um vertice na lista
    Node *check = NULL;
    Node *newNode = newAdjListNode(dest);

    if (grafo->array[src].head == NULL)
    {
        newNode->next = grafo->array[src].head;
        grafo->array[src].head = newNode;
    }
    else
    {

        check = grafo->array[src].head;
        while (check->next != NULL)
        {
            check = check->next;
        }

        check->next = newNode;
    }

     // Como o grafo é não-direcionado, adiciona o nodo de src em dest
tbm
    newNode = newAdjListNode(src);
    if (grafo->array[dest].head == NULL)
    {
        newNode->next = grafo->array[dest].head;
        grafo->array[dest].head = newNode;
```

```
    }
    else
    {
        check = grafo->array[dest].head;
        while (check->next != NULL)
        {
            check = check->next;
        }
        check->next = newNode;
    }
}

void checkEdge(int v1, int v2, Grafo *graph)
{
    // indice de erro
    int erro = 0;
    Node *check;
    // verifica se o nodo requisitado existe dentro do grafo
    if (v1 >= graph->V)
    {
        printf("valor esta fora do alcance do grafo\n");
        erro = 1;
        // se ele existir, check recebe o ponteiro do valor q ele referencia
    }
    else
    {
        check = graph->array[v1].head;
    }

    // aqui, se ele nn houve erro, ele vai checar a aresta
    if (erro == 0)
    {
        while (check)
            { // enquanto check for diferente de NULL, procura no nodo
seguinte
            if (check->dest != v2)
            {
                check = check->next;
            }
            else
            {
                printf("Aresta esta presente\n");
                break;
            }
        }
         // se chegou aqui, com certeza check == NULL, então aresta nn
estava presente
        if (check == NULL)
```

```
            printf("Aresta nao esta presente\n");
        }
    }

    void removeEdge(int v1, int v2, Grafo *grafo)
    {
        Node *node;
        Node *aux;

        // checa se v1 está dentro do escopo de tamanho de grafo
        if (v1 >= grafo->V)
        {
            printf("Valor de v1 está fora do alcance do grafo\n");
            // checa se v2 está dentro do escopo do tamanho de grafo
        }
        else if (v2 >= grafo->V)
        {
            printf("Valor de v2 está fora do alcance do grafo\n");
        }
        else
        {
                // ambos v1 e v2 estão dentro do escopo, então começa a
    checar se tem
            node = grafo->array[v1].head; // node recebe o valor da cabeça
            while (node)
            {
                if (node->dest == v2)
                { // checa se valor de destino de nodo é v2
                    // se for, remove ele do nodo
                    aux->next = node->next;
                    free(node);
                    printf("Aresta v1:v2 removida\n");
                    break;
                }
                else if (node->next == NULL)
                    {                                       // senão, ele checa se o
    proximo ponteiro é nulo
                        printf("Nao existe aresta entre v1 e v2\n"); // se for, o v2 nn
    estava feito no grafo
                    break;
                }
                aux = node;
                node = node->next;
            }
        }
    }

    // remove todo o grafo
```

```
void removeGraph(Grafo *grafo)
{
   // Vtotal é a qtd de vertices total
   int Vtotal = (grafo->V) - 1;
     // enquanto Vtotal é um valor dentro do tamanho, ele faz um laço
for
   for (Vtotal; Vtotal > -1; Vtotal--)
   {
      Node *nodo;
      Node *aux;
        // enquanto a cabeça da lista no indice de Vtotal for diferente de
nulo
      // ele vai fazendo limpeza de cada vertice do grafo
      while (grafo->array[Vtotal].head != NULL)
      {
         nodo = grafo->array[Vtotal].head;

          // condição de parada pra quando ele atinge o ultimo nodo, na
cabeça no indice Vtotal
         if (nodo->next == NULL)
         {
            free(nodo);
            break;
         }

         // varrendo todo o array no indice de Vtotal
         while (nodo)
         {
            // condição de limpeza de aresta
            if (nodo->next == NULL)
            {
               aux->next = NULL;
               free(nodo);
               break;
            }
            aux = nodo;
            nodo = nodo->next;
         }
      }
   }
   // libera espaço do grafo
   free(grafo);
   printf("Grafo removido com sucesso!\n");
}

// Printa grafo
void printGrafo(Grafo *graph)
{
```

```
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        Node *pCrawl = graph->array[v].head;
        printf("\n Lista de Adjacência do Nodo %d\n Inicio ", v);
        while (pCrawl)
        {
            printf("-> %d ", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}
```

2.2 Classe Fila com herança

O objetivo deste código é utilizar a classe de lista para realizar as funções na classe de fila, utilizando assim a herança.

Como os códigos estão separados em quatro partes, o “node.hpp”, “lista.hpp”, “fila.hpp” e “interface.cpp”. Abaixo temos os códigos fontes da fila.

2.2.1. “node.hpp”

```
// código que forma a classe nodo onde armazenamos os dados da
nossa fila e armazenamos o ponteiro para o próximo elementos
#ifndef NODE_HPP
#define NODE_HPP
template <typename TipoInfo>
// classe
class Node
{
private:
    TipoInfo dados;
    Node<TipoInfo> *proximo;

public:
    Node();  // construtor
    ~Node(); // destruidor

    // funções de get e set
    TipoInfo getDados();
    void setDados(TipoInfo dados);
    Node<TipoInfo> *getProximo();
    void setProximo(Node<TipoInfo> *proximo);
};
// construtor
template <typename TipoInfo>
Node<TipoInfo>::Node()
{
    dados = TipoInfo(0);
```

```
    proximo = nullptr;
}
// destruidor
template <typename TipoInfo>
Node<TipoInfo>::~Node()
{
    proximo = nullptr;
}
// gets
template <typename TipoInfo>
TipoInfo Node<TipoInfo>::getDados()
{
    return dados;
}
template <typename TipoInfo>
Node<TipoInfo> *Node<TipoInfo>::getProximo()
{
    return proximo;
}
// sets
template <typename TipoInfo>
void Node<TipoInfo>::setDados(TipoInfo dados)
{
    this->dados = dados;
}
template <typename TipoInfo>
void Node<TipoInfo>::setProximo(Node<TipoInfo> *proximo)
{
    this->proximo = proximo;
}
#endif
```

2.2.2. “lista.hpp”

```
// Código com a classe lista, onde receberá o início, fim e tamanho dela. Foi implementado apenas as funções que uma fila precisa, como append no fim e remover no início
#ifndef LISTA_HPP
#define LISTA_HPP
#include "node.hpp"
template <typename TipoInfo>

class Lista
{
private:
    Node<TipoInfo> *inicio;
    Node<TipoInfo> *fim;
    int tamanho;
```

```
public:
    // construtor e destruidor
    Lista();
    ~Lista();
    // get
    Node<TipoInfo> *getInicio();
    Node<TipoInfo> *getFim();
    int getTamanho();
    // set
    void setInicio(Node<TipoInfo> *inicio);
    void setFim(Node<TipoInfo> *fim);
    void setTamanho(int tamanho);
    // funções
    void append(TipoInfo dados);
    TipoInfo removeFirst();
    void showList();
};
// construtor
template <typename TipoInfo>
Lista<TipoInfo>::Lista()
{
    inicio = nullptr;
    fim = nullptr;
    tamanho = 0;
}
// destruidor
template <typename TipoInfo>
Lista<TipoInfo>::~Lista()
{
    inicio = nullptr;
    fim = nullptr;
}
// get
template <typename TipoInfo>
Node<TipoInfo> *Lista<TipoInfo>::getInicio()
{
    return inicio;
}
template <typename TipoInfo>
Node<TipoInfo> *Lista<TipoInfo>::getFim()
{
    return fim;
}
template <typename TipoInfo>
int Lista<TipoInfo>::getTamanho()
{
    return tamanho;
}
```

```
// set
template <typename TipoInfo>
void Lista<TipoInfo>::setInicio(Node<TipoInfo> *inicio)
{
   this->inicio = inicio;
}
template <typename TipoInfo>
void Lista<TipoInfo>::setFim(Node<TipoInfo> *fim)
{
   this->fim = fim;
}
template <typename TipoInfo>
void Lista<TipoInfo>::setTamanho(int tamanho)
{
   this->tamanho = tamanho;
}
// funções
// adiciona no final
template <typename TipoInfo>
void Lista<TipoInfo>::append(TipoInfo dado)
{
   Node<TipoInfo> *node = new Node<TipoInfo>();
   node->setDados(dado);
   node->setProximo(nullptr);
   if (tamanho == 0)
   {
      inicio = node;
      fim = node;
   }
   else
   {
      fim->setProximo(node);
      fim = node;
   }
   tamanho++;
}
// remove o primeiro
template <typename TipoInfo>
TipoInfo Lista<TipoInfo>::removeFirst()
{
   Node<TipoInfo> *node;
   TipoInfo aux;
   if (tamanho == 0)
   {
      std::cout << "Lista vazia" << std::endl;
      return (TipoInfo(0));
   }
   else if (tamanho == 1)
```

```
    {
        node = inicio;
        aux = inicio->getDados();
        inicio = nullptr;
        fim = nullptr;
        tamanho = 0;
        delete node;
    }
    else if (tamanho > 1)
    {
        node = inicio;
        aux = inicio->getDados();
        inicio = inicio->getProximo();
        tamanho--;
        delete node;
    }

    return aux;
}
// mostra lista
template <typename TipoInfo>
void Lista<TipoInfo>::showList()
{
    Node<TipoInfo> *node = inicio;
    if (tamanho == 0)
    {
        std::cout << "FILA VAZIA!!";
    }
    else
    {
          std::cout << "===============FILA===============" <<
std::endl;
        for (int i = 0; i < tamanho; i++)
        {
            std::cout << node->getDados() << std::endl;
            node = node->getProximo();
        }
          std::cout << "==================================" <<
std::endl;
    }
}
#endif
```

2.2.3. “fila.hpp”

```
// classe fila, que recebe os atributos e funções da nossa classe lista
#ifndef FILA_HPP
#define FILA_HPP
#include "lista.hpp"
template <typename TipoInfo>
```

```
class Fila : private Lista<TipoInfo>
{
public:
   Fila();
   ~Fila();

   TipoInfo remover();
   void enfileirar(TipoInfo dado);
   void showFila();
};
// construtor
template <typename TipoInfo>
Fila<TipoInfo>::Fila()
{
}
// destruidor
template <typename TipoInfo>
Fila<TipoInfo>::~Fila()
{
}
// função de remover
template <typename TipoInfo>
TipoInfo Fila<TipoInfo>::remover()
{
   return Lista<TipoInfo>::removeFirst();
}
// função de enfileirar
template <typename TipoInfo>
void Fila<TipoInfo>::enfileirar(TipoInfo dado)
{
   Lista<TipoInfo>::append(dado);
}
// mostrar fila
template <typename TipoInfo>
void Fila<TipoInfo>::showFila()
{
   Lista<TipoInfo>::showList();
}

#endif
```

2.2.4. “interface.hpp”

```
// executar: make run
// Interface do código
#include <iostream>
#include "fila.hpp"
```

```
int main()
{
   Fila<int> *fila;        // cria fila
   fila = new Fila<int>(); // inicia
   int num;
   int op;

   while (1)
   {
      std::cout << "\n1 - Enfilerar" << std::endl;
      std::cout << "2 - Remover" << std::endl;
      std::cout << "3 - Mostrar Fila" << std::endl;
      std::cout << "0 - Sair" << std::endl;
      std::cin >> op;

      switch (op)
      {
      case 0:
         exit(0);
      case 1:
         std::cout << "Insira um numero na Fila: " << std::endl;
         std::cin >> num;
         fila->enfileirar(num);
         break;
      case 2:
               std::cout << "Removido : " << fila->remover() << " " <<
std::endl;
         break;
      case 3:
         fila->showFila();
         break;
      default:
          std::cout << "O número digitado não corresponde a nenhuma
opção";
      }
   }
   return 0;
}
```

2.3. Mergesort

A ordenação Mergesort foi construído da seguinte maneira:
A parte inicial da função ocorre a leitura de um vetor de 8 números e os manda para a função mergesort. O código está escrito nos arquivos chamados "MergeSort.c" e “functionMergeSort.h”.

2.3.1. “MergeSort.c”

```
#include <stdio.h>
```

```
#include <stdlib.h>

int main()
{
  int i;
  int vetor[8];

  for (i = 0; i < 8; i++)
  {
    printf("[%d]: ", i);
    scanf("%d", &vetor[i]);
  }

  printf("\n");
  mergesort(vetor, 8);

  printf("Vetor Ordenado: [");
  for (i = 0; i < 8; i++)
    if (i == 7)
    {
      printf("%d", vetor[i]);
    }
    else
    {
      printf("%d, ", vetor[i]);
    }
  printf("]");
  return 0;
}
```

2.3.2. “functionMergeSort.h”

Aqui é determinado os parâmetros para o sort e merge.

```
void mergesort(int *vetor, int tamanho)
{
  int *aux = malloc(sizeof(int) * tamanho);
  sort(vetor, aux, 0, tamanho - 1);

  free(aux);
}
```

A função sort com recursividade de duas partes do vetor, início e fim, além de dividir o vetor, e retorna suas metades.

```
void sort(int *vetor, int *aux, int inicio, int fim)
{
  if (inicio >= fim)
    return;

  int meio = (inicio + fim) / 2;
  // entradas separadas até inicio >=fim
```

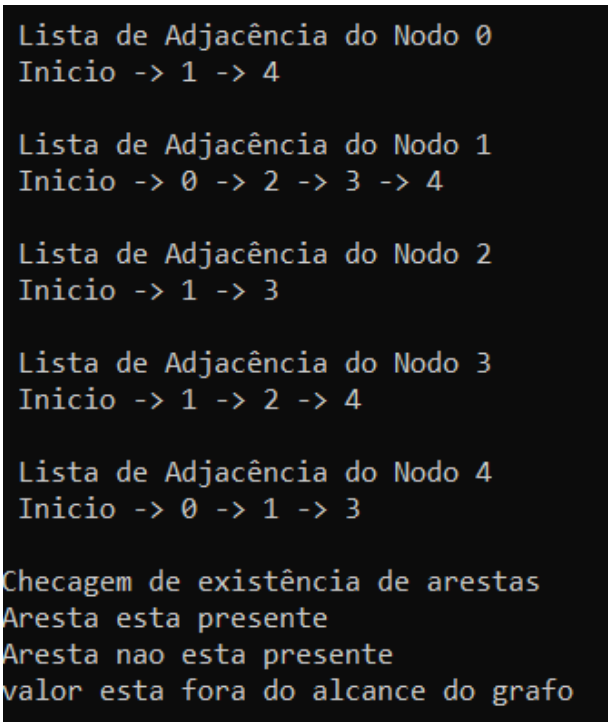
```
  sort(vetor, aux, inicio, meio);
  sort(vetor, aux, meio + 1, fim);

  if (vetor[meio] <= vetor[meio + 1])
    return;

  interloca(vetor, aux, inicio, meio, fim);
}
```

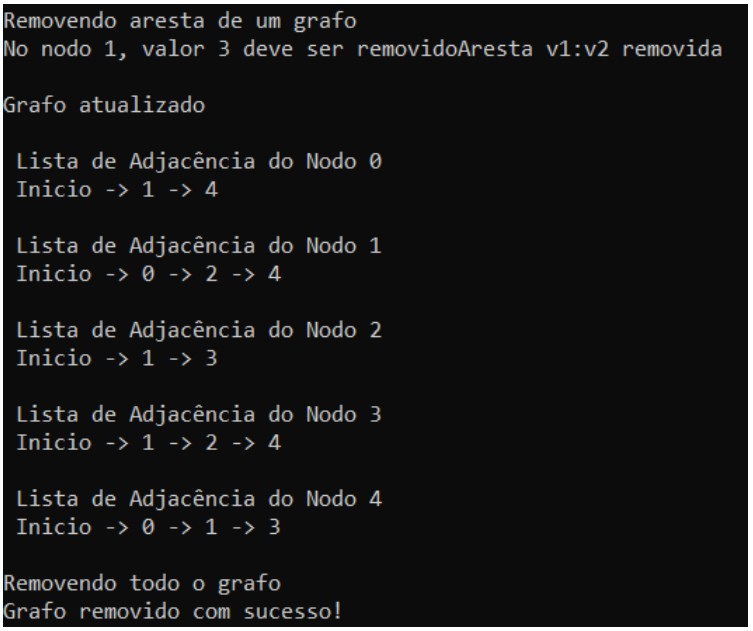
A função merge (interloca) recebe vetores separados e os une.

```
void interloca(int *vetor, int *aux, int inicio, int meio, int fim)
{
  int i,
      iv = inicio,
      im = meio + 1;
  // printando movimento do sort
  printf("|");
  for (i = 0; i < 8; i++)
    printf("%d ", vetor[i]);

  printf("|\n");
  for (i = inicio; i <= fim; i++)
    aux[i] = vetor[i];

  i = inicio;

  while (iv <= meio && im <= fim)
  {

    if (aux[iv] <= aux[im])
      vetor[i++] = aux[iv++];
    else
      vetor[i++] = aux[im++];
  }

  while (iv <= meio)
    vetor[i++] = aux[iv++];

  while (im <= fim)
    vetor[i++] = aux[im++];
}
```

3. Testes

3.1. Grafos





3.2. Fila com herança

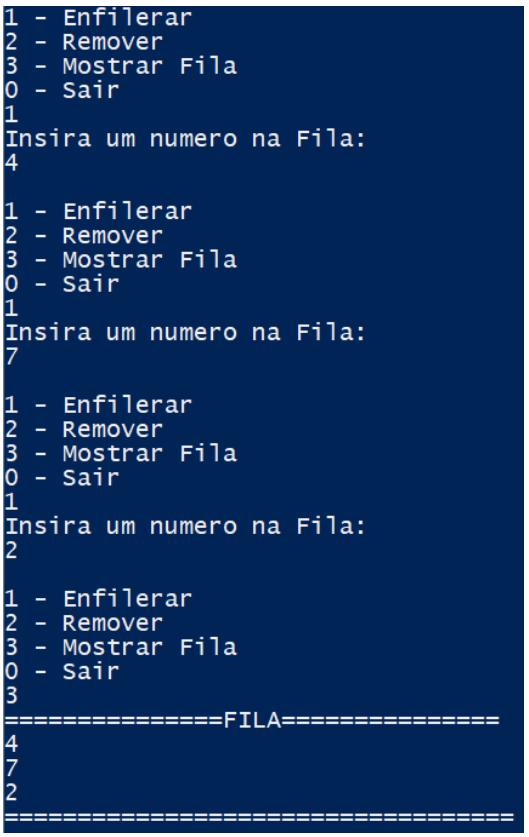
O primeiro teste será mostrar que a fila está vazia no início quando selecionamos a opção remover ou mostrar fila.

```
1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
2
Removido : Lista vazia
0

1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
3
FILA VAZIA!!
1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
```

O segundo teste é adicionar um item na lista e remover ele.

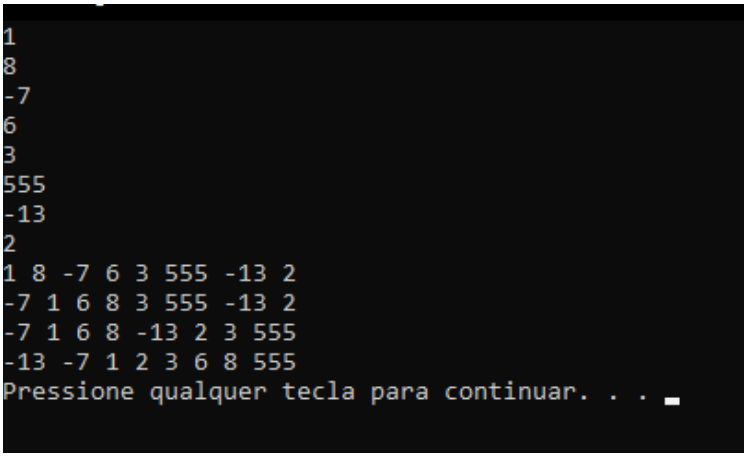
```
1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
1
Insira um numero na Fila:
3

1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
2
Removido : 3

1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
3
FILA VAZIA!!
1 - Enfilerar
2 - Remover
3 - Mostrar Fila
0 - Sair
```

O terceiro e último teste será adicionar três itens e remover dois itens da fila.



3.3. Mergesort

A seguir está o teste com números negativos e inteiros, onde é demonstrado o processo de ordenação e com o resultado final totalmente ordenado.


1
8
-7
6
3
555
-13
2
1 8 -7 6 3 555 -13 2
-7 1 6 8 3 555 -13 2
-7 1 6 8 -13 2 3 555
-13 -7 1 2 3 6 8 555
Pressione qualquer tecla para continuar. . .