Apostila de Algoritmos e Estrutura de Dados II *30-703* ***4 créditos***

EMENTA: Análise e projeto dos tipos de dados abstratos, estruturas de dados e suas aplicações: listas lineares, pilhas, filas. Métodos e técnicas de classificação de dados.

OBJETIVOS: Ao final desta disciplina o aluno deverá ser capaz de definir formalmente estruturas de dados, manipular estas estruturas, selecioná-las e utilizá-las em suas aplicações.

RELAÇÃO DOS CONTEÚDOS:

- **Conceitos Iniciais**
  - Introdução: tipos primitivos de dados, vetores, matrizes, estruturas (structs).
  - Tipos abstratos de dados (TADs)
  - Representação e implementação de TDA.

- **Recursividade**
  - Definição, exemplos, simulação e implementação de recursividade. Exercícios

- **Listas lineares**
  - Definição, estruturas estáticas e dinâmicas, operações básicas em listas de elementos.

- **Pilhas**
  - Definição do tipo abstrato, aplicações e exemplos
  - Operações básicas em uma pilha
  - Exercícios e Implementações de pilhas

- **Filas**
  - Definição do tipo abstrato, aplicações e exemplos
  - Operações básicas em uma fila
  - Filas circulares
  - Exercícios e Implementações de filas

- **Classificação**
  - Listas ordenadas. Métodos de classificação de dados por:
    - Inserção (direta e incrementos decrescentes)
    - Troca (bolha e partição)
    - Seleção (seleção direta e em árvore)
    - distribuição e intercalação

- **Listas ligadas**
  - Pilhas ligadas
  - Filas lidadas
  - Listas ligadas
  - Listas duplamente ligadas
  - Exercícios e Implementações

BIBLIOGRAFIA BÁSICA (LIVROS TEXTOS):

TENEMBAUM, Aaron M. **Estrutura de Dados Usando C**. São Paulo: Makron Books do Brasil, 1995.
VELLOSO, Paulo. **Estruturas de Dados**. Rio de Janeiro: Ed. Campus, 1991.
VILLAS, Marcos V & Outros. **Estruturas de Dados: Conceitos e Técnicas de implementação.** RJ: Ed. Campus, 1993.

BIBLIOGRAFIA COMPLEMENTAR:

SKIENA, Steven; Revilla, **Miguel. Programming Challenges.** Springer-Verlag New York, 2003.
UVA Online Judge http://icpcres.ecs.baylor.edu/onlinejudge/
AZEREDO, Paulo A. **Métodos de Classificação de Dados**. Rio de Janeiro: Ed. Campus, 1996.

AVALIAÇÃO.: A avaliação consistirá de 3 notas (prova1 + prova2 + trabalho) / 3, sendo que os trabalhos serão os problemas propostos em cada capítulo, que devem ser desenvolvidos em laboratório

1. Conceitos Iniciais

1.1 Tipo de dados

Assume-se que cada constante, variável, expressão ou função é um certo tipo de dados. Esse tipo refere-se essencialmente ao conjunto de valores que uma constante variável, etc. pode assumir.

Tipo primitivos de dados:
Essencialmente são os números, valores lógicos, caracteres, etc que são identificados na maioria das linguagens:
int: compreende os números inteiros **float:** compreende os números reais **char:** compreende os caracteres

1.2 Vetor ou Array

A[10] - Nome do vetor e um índice para localizar.

6	8	4	11	2	13	7	4	9	1
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]

Operações com vetor:
Some o 1º e o último elemento do vetor na variável x: x = a[0] + a[9] x = 6 + 1

Exercício:
Calcule a média dos elementos do vetor:

```
#include <iostream>
using namespace std;

float media (int vet2[10]);

int main(void){
   int i, vet[10];
   for (i=0; i<10; i++)
      cin >> vet[i];
   float x = media(vet);
   cout <<  "Média= " <<  x;
   return (0);
}

float media (int vet2[10]){
   float soma = 0;
   for (int i = 0 ; i <10; i++)
      soma += vet2[i];
   return (soma/10);
}
```

Vetores bidimensionais em C:
`int a[5][2]` (5 linhas e 2 colunas) `int a[3][5][2]` (3 planos, 5 linhas e 2 colunas)

Formas mais simples de declarar uma estrutura em C:

```
struct {
   char  primeiro[10];
   char  inicialmeio;
   char  ultimo[20];
} nome;
```

```
typedef struct {
      char  primeiro[10];
      char  inicialmeio;
      char  ultimo[20];
} TIPONOME;

TIPONOME nome, snome, fulano;
```

Obs: em todos os casos a inicialmeio é apenas um caracter. Exemplo.: Pedro S. Barbosa

```
#include <iostream>

using namespace std;

typedef struct {
   string nome;
   string endereco;
   string cidade;
} CADASTRO;
```

```
int main (void) {
   CADASTRO cliente;

   cout << "Digite o Nome: ";
   getline (cin, cliente.nome);

   cout << "Digite o Endereco: ";
   getline (cin, cliente.endereco);

   cout << "Digite o Cidade: ";
   getline (cin, cliente.cidade);

   cout << cliente.nome << " mora no(a) " << cliente.endereco << " , cidade de " << cliente.cidade;
   return 0;
}
```

1.3 Tipos abstratos de dados

Fundamentalmente, um TDA significa um conjunto de valores e as operações que serão efetuadas sobre esses valores. Não se leva em conta detalhes de implementação. Pode ser até que não seja possível implementar um TDA desejado em uma determinada linguagem.

Como na matemática diferenciamos constantes, funções, etc., na informática identificamos os dados de acordo com as suas características. Por exemplo, se criarmos uma variável do tipo ***fruta***, ela poderia assumir os valores **pera**, **maçã**, etc., e as operações que poderíamos fazer sobre esta variável seriam **comprar, lavar**, etc.

Quando alguém quer usar um TDA "inteiro", ele não está interessado em saber como são manipulados os bits de Hardware para que ele possa usar esse inteiro. O TDA inteiro é universalmente implementado e aceito.

Definição de um TDA Racional

A definição de um TDA envolve duas partes: a definição de valores e a definição de operadores. O TDA Racional consiste em dois valores inteiros, sendo o segundo deles diferente de zero (0). (numerador e denominador).

A definição do TDA racional, por exemplo, inclui operações de criação, adição, multiplicação e teste de igualdade.

$$\frac{A[0]}{A[1]} + \frac{B[0]}{B[1]} = \frac{A[0] * B[1] + A[1] * B[0]}{A[1] * B[1]}$$

Implementação do tipo RACIONAL

Inicialmente deve-se criar uma estrutura denominada racional, que contém um numerador e um denominador e posteriormente cria-se um programa que utiliza as estruturas criadas. Abaixo segue o exemplo. Crie as operações de **soma, multiplicação** e **soma com simplificação** sobre números racionais em laboratório.

```
#include <iostream>

using namespace std;

typedef struct {
      int numerador;
      int denominador;
} RACIONAL;

RACIONAL r1,r2,soma, mult, simpl;

int main(void){
   int i;
   cout << "Digite o 1ro numerador:"; cin >> r1.numerador;
   cout << "Digite o 1ro denominador:"; cin >> r1.denominador;

   cout << "Digite o 2do numerador:"; cin >> r2.numerador;
   cout << "Digite o 2do denominador:"; cin >> r2.denominador;
   ...
   return (0);
}
```

2. Recursividade

Um algoritmo que resolve um problema grande em problemas menores, cujas soluções requerem a aplicação dele mesmo, é chamado **recursivo**.

Existem dois tipos de recursividade: *direta* e *indireta*:
***Direta*:** uma rotina R pode ser expressa como uma composição formada por um conjunto de comandos C e uma chamada recursiva à rotina R: R ◄— [C,R]

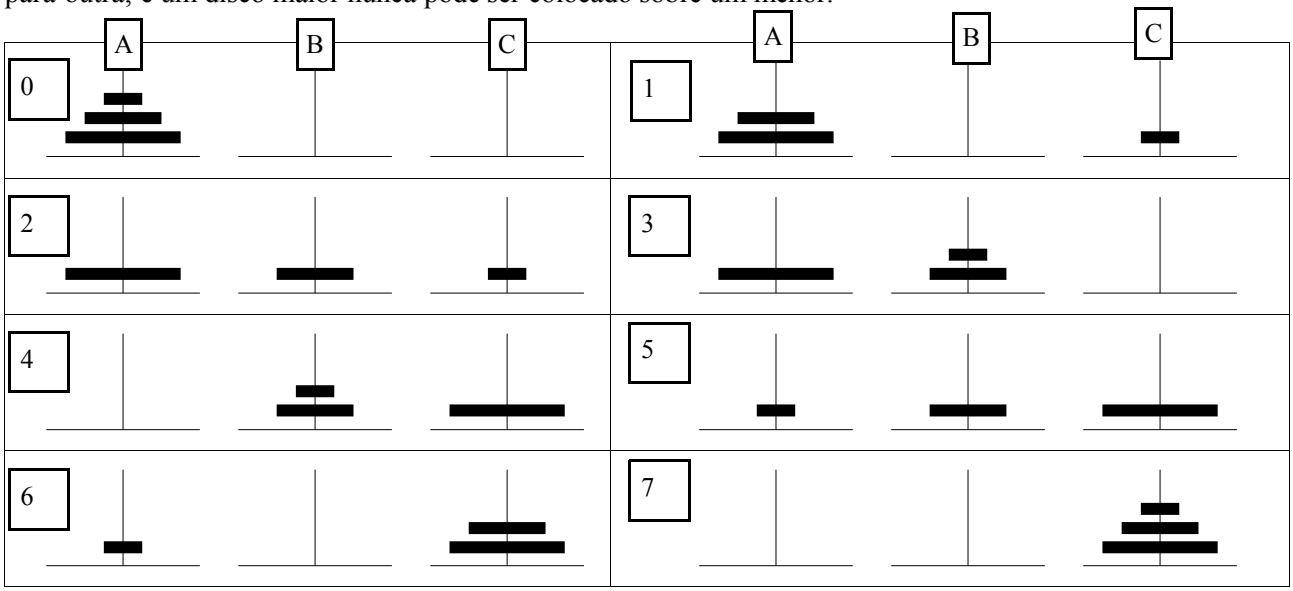
***Indireta*:** as rotinas são conectadas através de uma cadeia de chamadas sucessivas que acaba retornando à primeira que foi chamada:
R1 ← [C1,R2] R2 ← [C2,R13] ... Rn ← [Cn,R1]

Uso da recursividade na solução de problemas:
Ex: cálculo de fatorial:
0! = 1 // dado por definição
n! = n*(n-1) // requer reaplicação da rotina para (n-1)!

Veja a função:

```
Função fat(n)
Início
   Se n=0 então  retorne (1)
   Senão retorne (n*fat(n-1)) //chamada recursiva
Fim;
```

Problema das torres de Hanoi: Considerando três torres, o objetivo é transferir três discos que estão na torre A para a torre C, usando uma torre B como auxiliar. Somente o último disco de cima de uma pilha pode ser deslocado para outra, e um disco maior nunca pode ser colocado sobre um menor.



Dessa forma, para mover n discos da torre A para a torre C, usando a torre B como auxiliar, fazemos:
se n = 1
  mova o disco de A para C

senão
  transfira n-1 discos de A para B, usando C como auxiliar
  mova o último disco de A para C
  transfira n-1 discos de B para C, usando A como auxiliar

Primeira etapa:

se n = 1
  mova o disco de A para C (sem auxiliar) (origem para destino)

senão
  transfira n-1 discos de A (origem) para B (destino), usando C (auxiliar)
  mova disco **n** de A para C (origem para destino)
  transfira n-1 discos de B (origem) para C (destino), usando A (auxiliar)

Segunda etapa: Como a passagem dos parâmetros é sempre: torre A, torre B (auxiliar) e torre C, pois esta é a seqüência das 3 torres, os parâmetros devem ser manipulados na chamada recursiva da rotina Hanoi, respeitando a lógica dos algoritmos:

Programa principal:

```
início
   limpa tela
   hanoi(3,'A','B','C')
fim
```

Rotina Hanoi:

```
                                  A       B       C
Hanoi (int n, char origem, auxiliar, destino);
início
   se (n=1) então                                          A                C
       Escrever("1. Mova disco 1 da torre", origem, " para " , destino)
   senão
       Escrever("2.")   A       B       C
       Hanoi( n-1 , origem , destino , auxiliar)      A                  C
       Escrever("3. Mova disco" ,n, "da torre", origem, " para " ,destino)
       Hanoi( n-1 , auxiliar , origem , destino)
   fim_se                  A       B       C
fim
```

Com a chamada hanoi(3,'A','B','C'), o programa produzirá a seguinte saída:

```
2.
2.
1. Mova disco 1 da torre A para C
3. Mova disco 2 da torre A para B
1. Mova disco 1 da torre C para B
3. Mova disco 3 da torre A para C
2.
1. Mova disco 1 da torre B para A
3. Mova disco 2 da torre B para C
1. Mova disco 1 da torre A para C
```

Série de Fibonacci

A série de Fibonacci é a seguinte: fib(0) 0, fib(1) 1, fib(2) 1, fib(3) 2, fib(4) 3, fib(5) 5, fib(6) 8, ... 13, 21, 34, ...

Para fins de implementação, a série inicia com fib(0) e cada termo n é referenciado por fib(n-1), ou seja o 1º termo é fib(0), o 2º termo é fib(1) e assim sucessivamente. Com exceção dos dois primeiros, cujos valores são pré-determinados (0 e 1), cada elemento da seqüência sempre será a soma dos dois elementos anteriores. Ex:
fib(2) = 0+1 = 2, fib(3) = 1+1 = 2, fib(4) = 2+1 = 3, fib(5) = 2+3 = 5, ...

Podemos então definir a série de Fibonacci através da seguinte definição recursiva:

```
Fib(n) = n se n== 0 ou n==1
Fib(n) = Fib(n-2) + fib(n-1) se n >=2
```

Dessa forma, Fib(4) seria então:

```
Fib(4) = Fib(2)           + Fib(3) =
         Fib(0) + Fib(1)  + Fib(3) =
           0    +    1    + Fib(3) =
                     1    + Fib(1) + Fib(2) =
                     1    +   1    + Fib(0) + Fib(1) =
                          2        +   0    +   1    =    3
```

3. Estruturas de Dados Elementares

Estruturas de dados são o "coração" de qualquer programa mais sofisticado. A seleção de um tipo correto de estrutura de dados fará enorme diferença na complexidade da implementação resultante. A escolha da representação dos dados correta facilitará em muito a construção de um programa, enquanto que a escolha de uma representação errada custará um tempo enorme de codificação, além de aumentar a complexidade de compreenção do código.

Este capítulo apresentará problemas clássicos de programação com estruturas de dados fundamentais, principalmente problemas clássicos envolvidos em jogos de computadores.

Consideramos aqui as operações abstratas sobre as mais importantes estruturas de dados: pilhas, filas, dicionários, filas com prioridades e conjuntos.

Linguagens modernas orientadas a objetos como C++ e Java possuem bibliotecas padrões de estruturas de dados fundamentais. Para um aluno de curso de Ciência de Computação, é importante entender e saber construir as rotinas básicas de manipulação destas estruturas, mas por outro lado, é muito mais importante saber aplicá-las corretamente para solucionar problemas práticos ao invés de ficar reinventando a roda.

3.1. Pilhas

Pilhas e filas são contêiners onde os itens são recuperados de acordo com a inserção dos dados. Uma **pilha** é um conjunto ordenado de itens na qual todas as inserções e retiradas são feitas em uma das extremidades denominada **Topo**. Pilhas mantém a ordem (Last Input First Output). As operações sobre pilhas incluem:

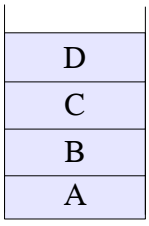
- Push(x,s): insere o item x no topo da pilha s.
- Pop(s): retorna e remove o item do topo da pilha s.
- Inicialize(s): cria uma pilha s vazia.
- Full(s), Empty(s): testa a pilha para saber se ela está cheia ou vazia.

Notadamente não poderá haver operação de pesquisa sobre uma pilha. A definição das operações abstratas acima permite a construção de uma pilha e sua reutilização sem a preocupação com detalhes de implementação. A implementação mais simples utiliza um vetor e uma variável que controla o topo da pilha. A implementação com elementos ligados através de ponteiros é melhor porque não ocasiona overflow.

Um exemplo de pilha seria uma pilha de pratos. Quando um prato é lavado ele é colocado no topo da pilha. Se alguém estiver com fome, irá retirar um prato também do topo da pilha. Neste caso uma pilha é uma estrutura apropriada para esta tarefa porque não importa qual será o próximo prato usado.

Outros casos existem onde a ordem é importante no processamento da estrutura. Isso inclui fórmulas parametrizadas (push em um "(", pop em um ")") - será visto adiante, chamadas recursivas a programas (push em uma entrada de função, pop na saida) e busca em profundidade em grafos transversais (push ao descobrir um vértice, pop quando ele for visitado pela última vez)

Funcionamento:
Ex: Entrada: A, B, C, D, E, F
Seqüência: I I R I I R I R R I R R
Saída: **B D E C F A**



Exercícios:

1) Complete:

a) Entrada: O D D A
Seqüência: I I I R I R R R
Saída:

b) Entrada: I E N S R E
Seqüência: I R I I R I R I I R R R
Saída:

2) Complete:

a) Entrada: O,A,D,D,S
Seqüência:
Saída: D,A,D,O,S

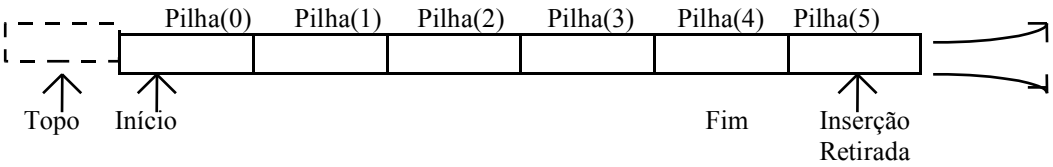
b) Entrada: E,E,X,E,C,E,L,N,T
Seqüência:
Saída: E,X,C,E,L,E,N,T,E

3) Complete:

a) Entrada:
Seqüência: I,I,R,I,I,R,R,I,R
Saída: D,A,D,O,S

b) Entrada:
Seqüência: I,I,I,R,I,I,I,R,I,I,I,R,R,R,R,R,R,R
Saída: E,X,C,E,L,E,N,T,E

Implementação:



Para implementarmos uma pilha seqüencial precisamos de uma variável (Topo) indicando o endereço da mais recente informação colocada na pilha. Por convenção, se a pilha está vazia, o Topo = -1. O tamanho da Pilha é limitado pelo tamanho do vetor.

```
Typedef struct PILHA {
   int topo;
   char dados[10];
}

PILHA p;
```

3.1.1. Inserção

Na inserção deve-se:
* incrementar o topo da pilha
* armazenar a informação X na nova área

3.1.2 Retirada

Na retirada deve-se:
* obter o valor presente em pilha.topo ou pilha->topo.
* decrementar o topo da pilha

O que acontece ao se inserir uma informação quando já usamos toda a área disponível do vetor (topo = fim) ? **Resposta:** ocorre uma situação denominada _____ **Algoritmo de inserção:**	O que acontece quando tentamos retirar um elemento de uma pilha que já está vazia? **Resposta:** ocorre uma situação denominada _____ **Algoritmo de retirada:**

```
#include <iostream>
#define TAM 10

using namespace std;

typedef struct {
       int topo;
       int dados [TAM];
} PILHA;

PILHA p1;

... Implemente o resto.
```

Como poderia ser feita a inserção e retirada no caso de se utilizar mais do que uma pilha?
- deve-se utilizar ponteiros para indicar qual das pilhas será manipulada e qual é o seu endereço.

Diferenças entre o algoritmo que manipula uma pilha e o algoritmo que manipula várias pilhas:
a) deve-se passar 2 parâmetros na inserção: a pilha em que o elemento será inserido e o elemento a ser inserido;
b) deve-se informar a pilha da qual o elemento será retirado;
c) na chamada das funções **insere** e **retira** deve-se passar o endereço da estrutura e dentro da função deve-se indicar que está vindo um ponteiro;
d) deve se usar a referência pilha->topo ao invés de pilha.topo.

```
Pilha p1,p2,p3;

void empilha (struct stack *p, int x){
...
}
int desempilha(struct stack *p ){
...
}
int main(void){
   int x;
   pilha1.topo=-1;      pilha2.topo=-1;      pilha3.topo=-1;
   empilha(&pilha1,4);     empilha(&pilha2,2);      empilha(&pilha3,7);
   x=desempilha(&pilha1); cout << x << endl;
   x=desempilha(&pilha1); cout << x << endl;
   x=desempilha(&pilha2); cout << x << endl;
   return (0);
}
```

Implemente em laboratório:

1) Implemente a rotina empilha e a rotina desempilha, demonstrando visualmente como fica a estrutura (pilha) após cada uma das operações. Obs.: Implemente a pilha sobre um vetor com 10 posições.

2) Construa um programa que leia 10 valores e empilha-os conforme forem pares ou ímpares na pilha1 e pilha2 respectivamente. No final desempilhe os valores de cada pilha mostrando-os na tela.

3.1.3 Utilização pratica de pilhas - Avaliação de Expressões

Agora que definimos uma pilha e indicamos as operações que podem ser executadas sobre ela, vejamos como podemos usar a pilha na solução de problemas. Examine uma expressão matemática que inclui vários conjuntos de parênteses agrupados. Por exemplo:

```
7-(( X *((X + Y) / (J - 3)) + Y) / (4 - 2.5))
```

Queremos garantir que os parênteses estejam corretamente agrupados, ou seja, desejamos verificar se:
a) Existe um número igual de parênteses esquerdos e direitos.
Expressões como "`A((A + B)`" ou "`A + B(`" violam este critério.

b) Todo parêntese da direita está precedido por um parêntese da esquerda correspondente.
Expressões como "`)A+B(-C`" ou "`(A+B))-(C+D`" violam este critério.

Para solucionar esse problema, imagine cada parêntese da esquerda como uma abertura de um escopo, e cada parêntese da direita como um fechamento de escopo. A **profundidade do aninhamento** (ou **profundidade do agrupamento**) em determinado ponto numa expressão é o número de escopos abertos, mas ainda não fechados nesse ponto. Isso corresponderá ao número de parênteses da esquerda encontrados cujos correspondentes parênteses da direita ainda não foram encontrados.

Determinemos a **contagem de parênteses** em determinado ponto numa expressão como o número de parênteses da esquerda menos o número de parênteses da direita encontrados ao rastear a expressão a partir de sua extremidade esquerda até o ponto em questão. Se a contagem de parênteses for não-negativa, ela eqüivalerá á profundidade do aninhamento. As duas condições que devem vigorar caso os parênteses de uma expressão formem um padrão admissível são as seguintes:

1. A contagem de parênteses no final da expressão é 0. Isso implica que nenhum escopo ficou aberto ou que foi encontrada a mesma quantidade de parênteses da direita e da esquerda.

2. A contagem de parênteses em cada ponto na expressão ê não-negativa. Isso implica que não foi encontrado um parêntese da direita para o qual não exista um correspondente parêntese da esquerda.

Na Figura 1, a contagem em cada ponto de cada uma das cinco strings anteriores é dada imediatamente abaixo desse ponto. Como apenas a primeira string atende aos dois critérios anteriormente citados, ela é a única dentre as cinco com um padrão de parênteses correto.

Agora, alteremos ligeiramente o problema e suponhamos a existência de três tipos diferentes de delimitadores de escopo. Esses tipos são indicados por parênteses (e), colchetes [e] e chaves ({e}). Um finalizador de escopo deve ser do mesmo tipo de seu iniciador. Sendo assim, strings como:
(A + B], [(A + B]), {A - (B]} são inválidas.

É necessário rastrear não somente quantos escopos foram abertos como também seus tipos. Estas informações são importantes porque, quando um finalizador de escopo é encontrado, precisamos conhecer o símbolo com o qual o escopo foi aberto para assegurar que ele seja corretamente fechado.

Uma pilha pode ser usada para rastrear os tipos de escopos encontrados. Sempre que um iniciador de escopo for encontrado, ele será empilhado. Sempre que um finalizador de escopo for encontrado, a pilha será examinada. Se a pilha estiver vazia, o finalizador de escopo não terá um iniciador correspondente e a string será, consequentemente, inválida. Entretanto, se a pilha não estiver vazia, desempilharemos e verificaremos se o item desempilhar corresponde ao finalizador de escopo. Se ocorrer uma coincidência, continuaremos. Caso contrário, a string será inválida.

Quando o final da string for alcançado, a pilha deverá estar vazia; caso contrário, existe um ou mais escopos abertos que ainda não foram fechados e a string será inválida. Veja a seguir o algoritmo para esse procedimento. A figura 1 mostra o estado da pilha depois de ler parte da string
`{x + (y -[a + b])*c-[(d + e)]} / (h- (j- (k- [l-n]))).`

```
7 - ( ( X * ( ( X + Y ) / ( J - 3 ) ) + Y ) / ( 4 - 2.5 ) )
0 0 1 2 2 2 3 4 4 4 4 3 3 4 4 4 4 3 2 2 2 1 1 2 2 2 2 2 1 0

( ( A + B )
1 2 2 2 2 1

A + B      (
0 0 0      1

 )  A  +  B ( - C
-1 -1 -1 -1 0 0 0

( A + B )  ) -  ( C + D
1 1 1 1 0 -1 -1 0 0 0 0
```

Figura 1 Contagem de parênteses em vários pontos de strings.

UM EXEMPLO: INFIXO, POSFIXO E PREFIXO

Esta seção examinará uma importante aplicação que ilustra os diferentes tipos de pilhas e as diversas operações e funções definidas a partir delas. O exemplo é, em si mesmo, um relevante tópico de ciência da computação.

Considere a soma de A mais B. Imaginamos a aplicação do operador "+" sobre os operandos A e B, e escrevemos a soma como A + B. Essa representação particular é chamada infixa. Existem três notações alternativas para expressar a soma de A e B usando os símbolos A, B e +. São elas:

- + A B **prefixa**
- A + B **infixa**
- A B + **posfixa**

Os prefixos "pre", "pos" e "in" referem-se à posição relativa do operador em relação aos dois operandos. Na notação prefixa, o operador precede os dois operandos; na notação posfixa, o operador é introduzido depois dos dois operandos e, na notação infixa, o operador aparece entre os dois operandos.

Na realidade, as notações prefixa e posfixa não são tão incômodas de usar como possam parecer a princípio. Por exemplo, uma função em C para retornar a soma dos dois argumentos, A e B, é chamada por Soma(A, B). O operador Soma precede os operandos A e B.

Examinemos agora alguns exemplos adicionais. A avaliação da expressão A + B * C, conforme escrita em notação infixa, requer o conhecimento de qual das duas operações, + ou *, deve ser efetuada em primeiro lugar. No caso de + e *, "sabemos" que a multiplicação deve ser efetuada antes da adição (na ausência de parênteses que indiquem o contrário). Sendo assim, interpretamos A + B * C como A + (B * C), a menos que especificado de outra forma.

Dizemos, então, que a multiplicação tem precedência sobre a adição. Suponha que queiramos rescrever A + B * C em notação posfixa. Aplicando as regras da precedência, converteremos primeiro a parte da expressão que é avaliada em primeiro lugar, ou seja a multiplicação. Fazendo essa conversão em estágios, obteremos:

A + (B * C)	parênteses para obter ênfase
A + (BC *)	converte a multiplicação
A (BC *) +	converte a adição
ABC * +	forma posfixa

As únicas regras a lembrar durante o processo de conversão é que as operações com a precedência mais alta são convertidas em primeiro lugar e que, depois de uma parte da expressão ter sido convertida para posfixa, ela deve ser tratada como um único operando. Examine o mesmo exemplo com a precedência de operadores invertida pela inserção deliberada de parênteses.

(A + B) * C	forma infixa
(AB +) * C	converte a adição
(AB +) C *	converte a multiplicação
AB + C *	forma posfixa

Nesse exemplo, a adição é convertida antes da multiplicação por causa dos parênteses. Ao passar de (A + B) * C para (AB +) * C, A e B são os operandos e + é o operador. Ao passar de (AB +) * C para (AB +)C *, (A.B +) e C são os operandos e * é o operador. As regras para converter da forma infixa para a posfixa são simples, desde que você conheça as regras de precedência.

Consideramos cinco operações binárias: adição, subtração, multiplicação, divisão e exponenciação. As quatro primeiras estão disponíveis em C e são indicadas pelos conhecidos operadores +, -, * e /.

A quinta operação, exponenciação, é representada pelo operador ^. O valor da expressão A ^ B é A elevado à potência de B, de maneira que 3 ^ 2 é 9. Veja a seguir a ordem de precedência (da superior para a inferior) para esses operadores binários:
exponenciação multiplicação/divisão adição/subtração

Quando operadores sem parênteses e da mesma ordem de precedência são avaliados, pressupõe-se a ordem da esquerda para a direita, exceto no caso da exponenciação, em que a ordem é supostamente da direita para a esquerda. Sendo assim, A + B + C significa (A + B) + C, enquanto A ^ B ^ C significa A ^ (B ^ C). Usando parênteses, podemos ignorar a precedência padrão.

Uma questão imediatamente óbvia sobre a forma posfixa de uma expressão é a ausência de parênteses. Examine as duas expressões, A + (B * C) e (A + B) * C. Embora os parênteses em uma das expressões sejam supérfluos [por convenção, A + B * C = A + (B * C)], os parênteses na segunda expressão são necessários para evitar confusão com a primeira.

As formas posfixas dessas expressões são:

Forma Infixa	**Forma Posfixa**
A+(B*C)	ABC *+
(A+B)*C	AB+C*

Considerando a expressão: `a/b^c + d*e - a*c`
Os operandos são: Os operadores são:
O primeiro passo a saber é qual a ordem de prioridade dos operadores:

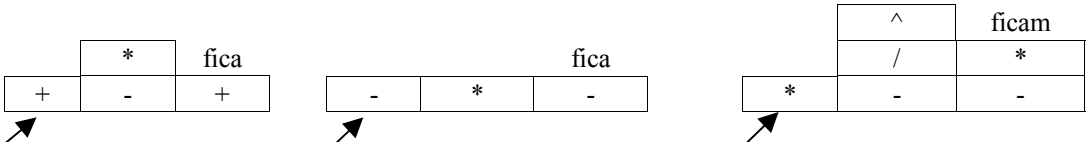
Operador	Prioridade
^, (+ e -) unário	6
*,/	5
+,-	4
>,<,>=,<=,<>	3
AND	2
OR	1

Após, basta utilizarmos uma pilha para transformarmos a expressão
Ex: A+B*C em ABC*+

Elemento	Pilha	Saída
A	-	A
+	+	A
B	+	AB
* (1)	+*	AB
C	+*	ABC

No final, basta juntarmos a saída ao que restou da pilha (desempilhando um a um) à saída: ABC * +
Note que em (1) o operador * foi colocado na pilha sobre o operador +. Isso se deve ao fato de que a prioridade do * é maior do que a do +. Sempre que a prioridade de um elemento a empilhar for maior do que a prioridade do último elemento da pilha, esse elemento deverá ser empilhado.

Quando a prioridade do elemento a empilhar for menor ou igual ao último elemento que está na pilha, deve-se então adotar o seguinte procedimento: desempilhar elementos até que fique como último elemento da pilha, algum elemento com prioridade menor do que o elemento que se está empilhando. Caso não houver na pilha nenhum elemento com prioridade menor do que o elemento que se está empilhando, fica somente o elemento que se está empilhando. Os demais saem para a saída.



Entra: + Entra: - Entra: *
Saem: *- Sai: * Saem: ^ /

Exercícios:
Transforme as seguintes expressões para a forma posfixa:
a) `A/B^C+D*E-A*C`
b) `A*(B+C*A-D)*D`
c) `X+(A*(B+C*A-D)*D)-2-F`

Problema proposto envolvendo pilha

LEXSIN - Avaliador Léxico e Sintático de Expressões (lexsin.cpp)

Uma das formas mais interessantes do uso de pilhas é a na avaliação de uma expressão matemática. Pode-se, através da pilha, fazer a análise léxica de uma expressão (indicar se uma expressão possui um operando inválido, como por exemplo um símbolo qualquer que não está presente nem na tabela de operadores, nem na tabela de operandos) e também a análise sintática. A análise sintática pode indicar que está faltando um ou mais parênteses, sobrando um ou mais parênteses, sobrando operador, 2 operandos sucessivos, etc. A tarefa aqui é determinar se uma expressão está correta ou não.

Entrada
Como entrada, são válidos:
a) Operandos: todas as letras maiúsculas ou minúsculas (ab... ...xz, AB... ...XZ) e números (01..9).
b) Parênteses.
c) Operadores: deverão ser aceitos os seguintes operadores segundo a tabela de prioridades apresentada abaixo:

Operador	***Prioridade***
`^`	6
`*,/`	5
`+,-`	4
`>,<,=,#,`	3
`AND ( . )`	2
`OR ( | )`	1

Para facilitar a implementação, pode-se utilizar um ponto como AND (.) e o Pipe (|) como OR.
Obs.: Como restrição, não será permitida a entrada de expressões com operadores unários. Exemplo: 4 * -2
A finalização da entrada será determinada pelo final do arquivo de entrada EOF().

Saída
Como saída, para cada expressão de entrada deverá ser gerado uma linha indicando o resultado do processamento. Se a expressão estiver correta, esta deverá ser transformada para a forma infixa. Se não for possível, devem ser apresentadas mensagens indicando: Operando Inválido ou Erro de sintaxe, nesta ordem.

Exemplo de entrada	Exemplo de saída
(	Erro de Sintaxe!
(A+	Erro de Sintaxe!
(A+B)*c	AB+c*
(A+B)*%	Erro Léxico!
(a+b*c)/2*e+a	abc*+2/e*a+
(a+b*c)/2*(e+a)	abc*+2/ea+*
(a+b*c)/2*(e+a	Erro de Sintaxe!
(ab+*c)/2*(e+a)	Erro de Sintaxe!
(a+b*cc)/2*(e+a	Erro de Sintaxe!
(“a+b*cc)/2*(e+a	Erro Léxico!

Início da solução

Alguns passos devem ser considerados aqui para a resolução do problema:

- Leia o Problema cuidadosamente: leia cada linha do problema cuidadosamente. Após desenvolver a solução, leia atentamente a descrição do erro. Confira atentamente as entradas e saídas do problema.
- Não pressuponha entradas: sempre há um conjunto de entradas para exemplo de um problema. Para testar, deve-se utilizar mais casos para entrada, números negativos, números longos, números fracionários, strings longas. Toda entrada que não for explicitamente proibida é permitida. Deve se cuidar a ordem de entrada também, pois quando pede-se por exemplo para verificar um intervalo entre 2 números, estes 2 números podem estar em ordem crescente ou decrescente e assim por diante.
- Não se apressar: nem sempre a eficiência é fundamental, portanto não deve-se preocupar com isso a menos que isso seja um predicado do problema. Deve-se ler a especificação para aprender o máximo possível sobre o tamanho da entrada e decidir qual algoritmo pode resolver o problema para aquela determinada entrada de dados. Neste caso específico a eficiência não precisa ser uma grande preocupação.

Sugestão para implementação

Existem muitas maneiras diferentes para verificar a prioridade de um elemento no trabalho de implementação. Uma sugestão seria criar uma estrutura com 2 vetores ELEM[10] e PRI[10]. Procura-se inicialmente o operador no vetor ELEM. Ao encontrá-lo, pega-se a prioridade na mesma posição do vetor PRI. Exemplos:

Rotina para testar prioridade

```
#include <iostream>
#include <string>
using namespace std;

int main(void){
    string operador;
    string expr = "|.><=#+-*/^";
    const int prior[11]= {1,2,3,3,3,3,4,4,5,5,6};
    int prioridade=-1;
    cin >> operador;
    for (int i=0; i<=10; i++) {
        if (expr.substr(i,1)==operador)
           prioridade = prior[i];
    }
    cout << prioridade;
    return (0);
}
```

Operando.cpp – Teste para ver se é um operando – (Letras maiúsculas, minúsculas e números)

//Obs: esta rotina deve ser usada em conjunto com a rotina "prioridade". Inicialmente, faz a leitura da expressão. Por exemplo **a+b*c%**. Faz-se então a verificação de cada caracter da entrada individualmente. O "**a**", por exemplo, é um operando válido. Então ele deve ir direto para a saída. No caso do "**+**", que é o segundo elemento da entrada, a rotina retorna que não é um operando válido. Então, deve-se passá-lo como parâmetro da rotina "prioridade" para verificar se o mesmo é um operador. Em caso afirmativo, deve-se proceder com o empilhamento ou desempilhamento do mesmo, conforme o caso. Ao testar um caracter inválido, por exemplo o "**%**", inicialmente verifica-se que não é um operando válido (rotina "operando"). Em seguida verifica-se também que não é um operador válido (rotina "prioridade). Neste caso, o software deve retornar um erro "operando/operador inválido!"

```
#include <iostream>
#include <string>
using namespace std;

int main(void){
    const string VALIDOS ("ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789");
    string entrada;
    cin >> entrada; //getline (cin,entrada) para o caso de permitir espaços em branco
    for (int i=0; i < entrada.length(); i++) {
        if (VALIDOS.find(entrada[i],0) != -1)
           cout << entrada[i] << " é um operando válido";
        else
           cout << entrada[i] << " não é operando válido: (operador, parênteses ou erro)!";
    }
    return (0);
}
```

3.2. FILA (QUEUE)

Filas mantém a ordem (first in, first out). Um baralho com cartas pode ser modelado como uma fila, se retirarmos as cartas em cima e colocarmos as carta de volta por baixo. Outro exemplo seria a fila de um banco. As operações de inserção são efetuadas no final e as operações de retirada são efetuadas no início. A inserção de um elemento torna-o último da lista (fila).

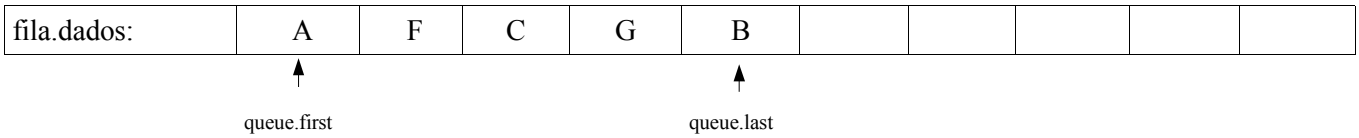
As operações abstratas em uma fila incluem:
- Enqueue(x,q): insere o item x no fim da fila q.
- Dequeue(q): retorna e remove o item do topo da fila q.
- Inicialize(q): cria uma fila q vazia.
- Full(q), Empty(q): testa a fila para saber se ela está cheia ou vazia.

Filas são mais difíceis de implementar do que pilhas, porque as ações ocorrem em ambas as extremidades. A implementação mais simples usa um vetor, inserindo novos elementos em uma extremidade e retirando de outra e depois de uma retirada movendo todos os elementos uma posição para a esquerda. Bem, pode-se fazer melhor utilizando os índices **first (**frente) e **last** (re) ao invés de fazer toda a movimentação. Ao se usar estes índices, fica uma área não usada no início da estrutura. Isso pode ser resolvido através de uma fila circular, onde após a inserção na última posição, inicia-se novamente a inserção pelo início da fila, se tiver posição disponível, é claro.

Dessa forma, teremos a seguinte estrutura:

```
typedef struct {
   int q[SIZE+1];
   int first;
   int last;
   int count;
} queue;

queue q;
```

Uma fila com 5 elementos, sem nenhum deles ter sido retirado é apresentada abaixo:



pseudo-algoritmo de inserção de uma fila:

```
void insere_fila (valor) {
   se fila->count > SIZE então
       OVERFLOW();
   senão
       fila.last++;
       fila.dados[fila.last]=valor;
       fila.count++;
   fim_se
}
```

pseudo-algoritmo de retirada de uma fila:

```
int retra_fila (void) {
   se fila.count <= 0 então
       UNDERFLOW();
   senão
       char x = fila.dados[fila.first];
       fila.first++;
       fila.count--;
       retornar(x);
   fim_se
}
```

```
/* Queue.cpp. Baseado no algoritmo queue.c do livro
   do Steven Skiena (2002) e atualizado por Neilor Tonin
   em 03/01/2008.
   Implementation of a FIFO queue abstract data type.
*/
```

```
#include <iostream>
#include "queue.h"

#define TRUE    1
#define FALSE   0
#define SIZE    3 //red <.

using namespace std;
```

```
#define SIZE    10

typedef struct {
        int q[SIZE+1]; /* body of queue */
        int first;                      /* position of first element */
        int last;                       /* position of last element */
        int count;                      /* number of queue elements */
} queue;
```

```
void init_queue(queue *q){
   q->first = 0;
   q->last = SIZE-1;
   q->count = 0;
}

void enqueue(queue *q, int x){
   if (q->count >= SIZE)
      cout << "Warning: overflow na fila ao inserir!" << x << endl;
   else {
      q->last = (q->last+1) % SIZE;
      q->q[ q->last ] = x;
      q->count = q->count + 1;
   }
}

int dequeue(queue *q){
   int x;
   if (q->count <= 0)
      cout << "Warning: underflow na fila ao retirar!" << endl;
   else {
      x = q->q[ q->first ];
      q->first = (q->first+1) % SIZE;
      q->count = q->count - 1;
   }
   return(x);
}

int empty(queue *q){
   if (q->count <= 0) return (TRUE);
   else return (FALSE);
}

void print_queue(queue *q){
   int i,j;
   i=q->first;

   cout <<endl<< "Elementos da fila:" << endl;
   while (i != q->last) {
      cout << q->q[i] << endl;
      i = (i+1) % SIZE;
  }
  cout << q->q[i] << endl;
}

int main(void){
   queue *q;
   init_queue(q);
   enqueue(q,12);
   enqueue(q,4);
   enqueue(q,32);
   print_queue(q);
   dequeue(q);
   enqueue(q,2);
   print_queue(q);
   dequeue(q);
   print_queue(q);
   return(0);
}
```

3.2.1 Exemplo de um projeto utilizando fila: Going to War

No jogo de cartas infantil “Going to War”, 52 cartas são distribuídas para 2 jogadores (1 e 2) de modo que cada um fica com 26 cartas. Nenhum jogador pode olhar as cartas, mas deve mantê-las com a face para baixo. O objetivo do jogo é recolher (ganhar) todas as 52 cartas.

Ambos jogadores jogam a virando a carta de cima do seu monte e colocando-a na mesa. A mais alta das duas vence e o vencedor recolhe as duas cartas colocando-as embaixo de seu monte. O rank das cartas é o seguinte, da maior para a menor: A,K,Q,J,T,9,8,7,6,5,4,3,2. Naipes são ignoradas. O processo é repetido até que um dos dois jogadores não tenha mais cartas.

Quando a carta virada dos dois jogadores tem o mesmo valor acontece então a guerra. Estas cartas ficam na mesa e cada jogador joga mais duas cartas. A primeira com a face para baixo, e a segunda com a face para cima. A carta virada com a face para cima que for maior vence, e o jogador que a jogou leva as 6 cartas que estão na mesa. Se as cartas com a face para cima de ambos os jogadores tiverem o mesmo valor, a guerra continua e cada jogador volta a jogar uma carta com a face para baixo e outra com a face para cima.

Se algum dos jogadores fica sem cartas no meio da guerra, o outro jogador automaticamente vence. As cartas são adicionadas de volta para os jogadores na ordem exata que elas foram distribuídas, ou seja a primeira carta própria (jogada pelo próprio jogador), a primeira carta do oponente, a segunda carta própria, a segunda carta jogada pelo oponente e assim por diante...

Como qualquer criança de 5 anos, sabe-se que o jogo de guerra pode levar um longo tempo para terminar. Mas quanto longo é este tempo? O trabalho aqui é escrever um programa para simular o jogo e reportar o numero de movimentos.

Construção do monte de cartas

Qual a melhor estrutura de dados para representar um monte de cartas? A resposta depende do que se quer fazer com elas. Está se tentando embaralhá-las? Comparar seus valores? Pesquisar por um padrão na pilha. As intenções é que vão definir as operações na estrutura de dados.

A primeira ação que necessitamos fazer é dar as cartas do topo e adicionar outras na parte de baixo do monte. Portanto, é natural que cada jogador utilize uma fila (estrutura FIFO) definida anteriormente.

Mas aqui têm-se um problema fundamental. Como representar cada carta? Têm-se as naipes (Paus, Copas, Espada e Ouro) e valores na ordem crescente (2-10, valete, rainha, rei e Ás). Têm-se diversas escolhas possíveis. Pode-se representar cada carta por um par de caracteres ou números especificando a naipe e valor. No problema da GUERRA, pode-se ignorar as naipes – mas tal pensamento pode trazer um problema. Como saber que a implementação da Fila está funcionando perfeitamente?

A primeira operação na GUERRA é comparar o valor da face das cartas. Isso é complicado de fazer com o primeiro caracter da representação, porque deve-se comparar de acordo com o ordenamento histórico dos valores da face. Uma lógica Had Hoc parece necessária para se lidar com este problema. Cada carta deverá ter um valor de 0 a 13. Ordena-se os valores das cartas do menor ao maior, e nota-se que há 4 cartas distintas de cada valor. Multiplicação e divisão são a chave para mapeamento de 0 até 51.

```
#include <iostream>
#include <cmath>
#include <cstdlib>
#include "queue2.h"  //Biblioteca com a estrutura da fila e suas funções
#define NCARDS 52    //cartas
#define NSUITS 4     //Naipes
#define TRUE 1
#define FALSE 0
#define MAXSTEPS 100000 //Define o número máximo de jogadas

using namespace std;

char values[] = "23456789TJQKA";
char suits[] = "pceo"; // (p)aus (c)opas (e)spadas e (o)uros

char value(int card){
       return( values[card/NSUITS] );
}

void print_card_queue(queue *q) {
        int i=q->first,j;
        while (i != q->last) {
            cout << value(q->q[i])<<  suits[q->q[i] % NSUITS];
            i = (i+1) % QUEUESIZE;
        }
        cout << value(q->q[i]) <<  suits[q->q[i] % NSUITS];
}

void clear_queue(queue *a, queue *b){
       while (!empty(a))
              enqueue(b,dequeue(a));
}
```

```
void embaralha (int perm[NCARDS+1]){
    randomize();
    int a,b,i,aux;
    for (i = 0; i <=20; i++){
       a= rand()%52;
       b= rand()%52;
       aux = perm[a];
       perm[a]=perm[b];
       perm[b]=aux;
    }
}

void random_init_decks(queue *a, queue *b){
   int i;                              // counter
   int perm[NCARDS+1];
   for (i=0; i<NCARDS; i=i+1) {
               perm[i] = i;
    }
    embaralha(perm);
    init_queue(a);
    init_queue(b);
    for (i=0; i<NCARDS/2; i=i+1) {
       enqueue(a,perm[2*i]);
       enqueue(b,perm[2*i+1]);
    }
    //cout << endl << "CARTAS: " << endl;
    //print_card_queue(a);
    //cout << endl << "CARTAS: " << endl;
    //print_card_queue(b);
}

void war(queue *a, queue *b) {
   int steps=0;                /* step counter */
   int x,y;                    /* top cards */
   queue c;                    /* cards involved in the war */
   bool inwar;                 /* are we involved in a war? */
   inwar = FALSE;
   init_queue(&c);

    while ((!empty(a)) && (!empty(b) && (steps < MAXSTEPS))) {
        print_card_queue(a);
        cout << endl;
        print_card_queue(b);
        cout << endl << endl;
       steps = steps + 1;
       x = dequeue(a);
       y = dequeue(b);
       // x e y possuem valores de 0 até 51
       cout << x <<":"<< value(x)  <<suits[x%NSUITS] <<" "<<y<<":"<<value(y)<< suits[y%NSUITS] << endl;
       enqueue(&c,x);
       enqueue(&c,y);
       if (inwar) {
              inwar = FALSE;
       } else {
              if (value(x) > value(y))
                     clear_queue(&c,a);
              else if  (value(x) < value(y))
                     clear_queue(&c,b);
              else if (value(y) == value(x))
                     inwar = TRUE;
       }
   }
   cout << "Cartas nas pilhas A: " << a->count << " B: " << b->count << endl;
   if (!empty(a) && empty(b))
       cout << "a venceu em " << steps << " jogadas " << endl;
   else if (empty(a) && !empty(b))
       cout << "b venceu em " << steps << " jogadas " << endl;
   else if (!empty(a) && !empty(b))
       cout << "jogo empatado apos " << steps << " jogadas" << endl;
   else
       cout << "jogo empatado apos " << steps << " jogadas" << endl;
}

int main(){
   queue a,b;
   int i;
   random_init_decks(&a,&b);
   war(&a,&b);
   return(0);
}
```

Problemas (exercícios) complementares **10038,** 10315, **10050,** 843, 10205, 10044, 10258 **(páginas 42 até 54)** - livro Steven Skiena e Miguel Revilla (http://icpcres.ecs.baylor.edu/onlinejudge/)

4. Classificação (ordenação) de dados

Classificar é o processo de ordenar os elementos pertencente a uma estrutura de dados em memória (vetor) ou em disco (registros de uma tabela de dados) em ordem ascendente ou descendentes.

Os fatores que influem na eficácia de um algoritmo de classificação são os seguintes:

- o número de registros a serem classificados;
- se todos os registros caberão ou não na memória interna disponível;
- o grau de classificação já existente;
- forma como o algoritmo irá ordenar os dados;

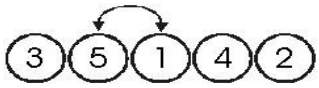
4.1 Bubblesort

Por ser simples e de entendimento e implementação fáceis, o Bubblesort (bolha) está entre os mais conhecidos e difundidos métodos de ordenação de arranjos. Mas não se trata de um algoritmo eficiente, é estudado para fins de desenvolvimento de raciocínio.

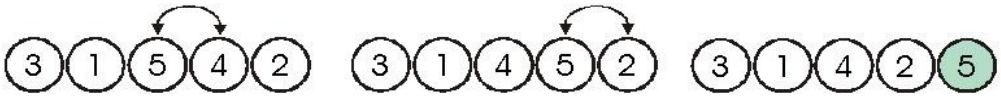
O princípio do Bubblesort é a troca de valores entre posições consecutivas, fazendo com que os valores mais altos (ou mais baixos) "borbulhem" para o final do arranjo (daí o nome Bubblesort). Veja o exemplo a seguir:

Nesta ilustração vamos ordenar o arranjo em ordem crescente de valores. Consideremos inicialmente um arranjo qualquer desordenado. O primeiro passo é se fazer a comparação entre os dois elementos das primeiras posições :

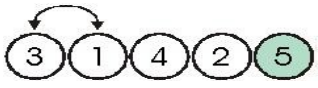
Assim verificamos que neste caso os dois primeiros elementos estão desordenados entre si, logo devemos trocá-los de posição. E assim continuamos com as comparações dos elementos subsequentes :



Aqui, mais uma vez, verificamos que os elementos estão desordenados entre si. Devemos fazer a troca e continuar nossas comparações até o final do arranjo :



Pode-se dizer que o número 5 "borbulhou" para a sua posição correta, lá no final do arranjo. O próximo passo agora será repetir o processo de comparações e trocas desdo início do arranjo. Só que dessa vez o processo não precisará comparar o penúltimo com o último elemento, pois o último número, o 5, está em sua posição correta no arranjo.

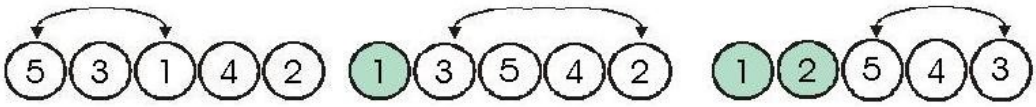



```
long int aux;  // Variável auxiliar para fazer a troca, caso necessário
for ( long int i=0; i <= tam-2; i++ ){
   for ( long int j=0; j<= tam-2-i; j++ ) {
      if ( Array[j] > Array[j+1] ) {  // Caso o elemento de uma posição menor
         aux        = Array[j];       // for maior que um elemento de uma posição
         Array[j]   = Array[j+1];     // maior, faz a troca.
         Array[j+1] = aux;
      }
   }
}
```

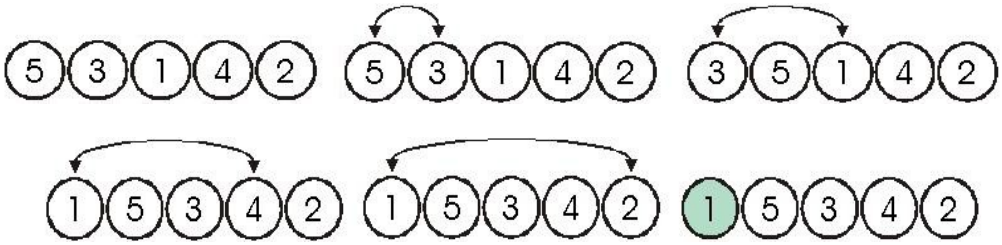
4.2 Seleção Direta

Consiste em encontrar a menor chave por pesquisa sequencial. Encontrando a menor chave, essa é permutada com a que ocupa a posição inicial do vetor, que fica então reduzido a um elemento.

O processo é repetido para o restante do vetor, sucessivamente, até que todas as chaves tenham sido selecionadas e colocadas em suas posições definitivas.



Uma outra variação deste método consiste em posicionar-se no primeiro elemento e aí ir testando-o com todos os outros (segundo)... (último), trocando cada vez que for encontrado um elemento menor do que o que está na primeira posição. Em seguida passa-se para a segunda posição do vetor repetindo novamente todo o processo. Ex:



Exercício: considerando o vetor:

9	25	10	18	5	7	15	3

Ordene-o pelo método de seleção direta:

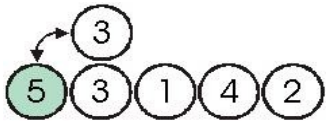
4.3 Inserção Direta

O método de ordenação por Inserção Direta é o mais rápido entre os outros métodos considerados básicos – Bubblesort e Seleção Direta. A principal característica deste método consiste em ordenarmos o arranjo utilizando um sub-arranjo ordenado localizado em seu inicio, e a cada novo passo, acrescentamos a este sub-arranjo mais um elemento, até que atingimos o último elemento do arranjo fazendo assim com que ele se torne ordenado. Realmente este é um método difícil de se descrever, então vamos passar logo ao exemplo.

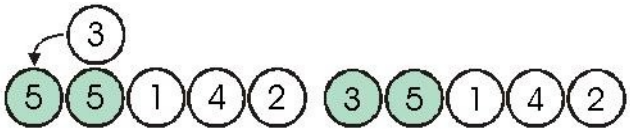
Consideremos inicialmente um arranjo qualquer desordenado:

Inicialmente consideramos o primeiro elemento do arranjo como se ele estivesse ordenado, ele será considerado o o sub-arranjo ordenado inicial :

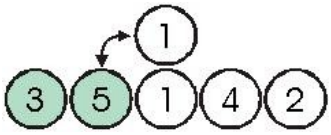
Agora o elemento imediatamente superior ao o sub-arranjo ordenado, no o exemplo o número 3, deve se copiado para uma variável auxiliar qualquer. Após copiá-lo, devemos percorrer o sub-arranjo a partir do último elemento para o primeiro. Assim poderemos encontrar a posição correta da nossa variável auxiliar dentro do sub-arranjo :



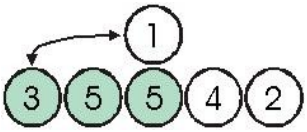
No caso verificamos que a variável auxiliar é menor que o último elemento do o sub-arranjo ordenado (o o sub-arranjo só possui por enquanto um elemento, o número 5). O número 5 deve então ser copiado uma posição para a direita para que a variável auxiliar com o número 3, seja colocada em sua posição correta :



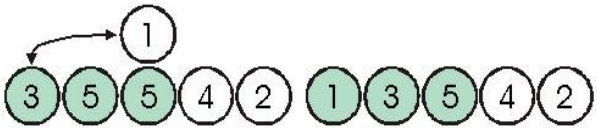
Verifique que o sub-arranjo ordenado possui agora dois elementos. Vamos repetir o processo anterior para que se continue a ordenação. Copiamos então mais uma vez o elemento imediatamente superior ao o sub-arranjo ordenado para uma variável auxiliar. Logo em seguida vamos comparando nossa variável auxiliar com os elementos do sub-arranjo, sempre a partir do último elemento para o primeiro :



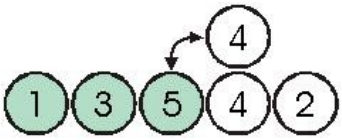
Neste caso verificamos que a nossa variável auxiliar é menor que o último elemento do sub-arranjo. Assim, copiamos este elemento para a direita e continuamos com nossas comparações :



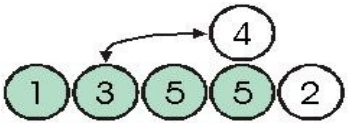
Aqui, mais uma vez a nossa variável auxiliar é menor que o elemento do sub-arranjo que estamos comparando. Por isso ele deve ser copiado para a direita, abrindo espaço para que a variável auxiliar seja colocada em sua posição correta :



Verifique que agora o sub-arranjo ordenado possui 3 elementos. Continua-se o processo de ordenação copiando mais uma vez o elemento imediatamente superior ao o sub-arranjo para a variável auxiliar. Logo em seguida vamos comparar essa variável auxiliar com os elementos do o sub-arranjo a partir do último elemento :

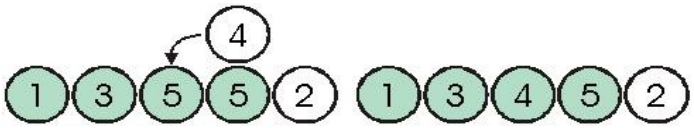


Veja que nossa variável auxiliar é menor que o elemento que está sendo comparado no o sub-arranjo. Então ele deve ser copiado para a direita para que continuemos com nossas comparações :

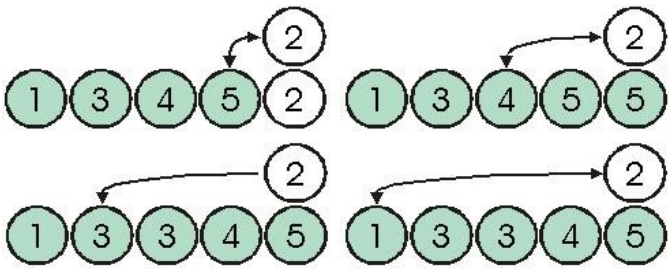


Veja que aqui ocorre o inverso. A variável auxiliar é maior que o elemento do sub-arranjo que estamos comparando. Isto significa que já encontramos a posição correta para a nossa variável auxiliar.

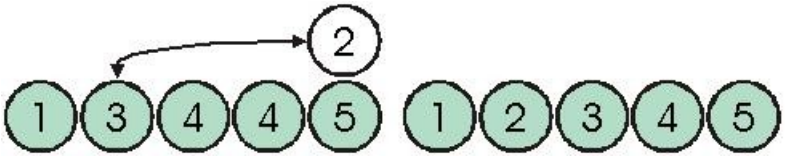
Basta agora copiá-la para sua posição correta, ou seja, copiá-la para o elemento imediatamente superior ao elemento que estava sendo comparado, veja :



o sub-arranjo ordenado possui agora quatro elementos. Repete-se mais uma vez o processo todo de ordenação, copiando o elemento imediatamente superior ao o sub-arranjo para uma variável auxiliar. Aí compara-se essa variável auxiliar com os elementos do o sub-arranjo, lembrando-se que a ordem é do último elemento para o primeiro. Caso seja necessário, copia-se para a direita os elementos que forem maiores que a variável auxiliar até que se encontre um elemento menor ou até que se chegue ao início do arranjo. É simples:



Aqui encontramos um elemento menor que a variável auxiliar. Isto significa que encontramos sua posição correta dentro do sub-arranjo. Basta agora copiá-la para a posição correta :



Exercício: considerando o vetor:

9	25	10	18	5	7	15	3

Ordene-o:

4.4 Pente (CombSort):

Este método de classificação implementa saltos maiores que 1 casa por vez. Suponhamos como exemplo o vetor de chaves abaixo. Como o vetor possui 5 chaves, o salto inicial é igual a 3. OBS.: o salto é dado pelo valor h = n / 1,3 Salto = Int(n / 1,3) = 3 Quando terminar o procedimento com salto = 1 então é utilizado o algoritmo BOLHA para terminar de ordenar

Var.	iter..	vetor					salto	par comparado	ação
1	1	**28**	26	30	**24**	25	3		troca
	2	24	**26**	30	28	**25**	3		troca
	3	**24**	25	**30**	28	26	2		não troca
2	4	24	**25**	30	**28**	26	2		não troca
	5	24	25	**30**	28	**26**	2		troca
	6	**24**	**25**	26	28	30	1		não troca
3	7	24	**25**	**26**	28	30	1		não troca
	8	24	25	**26**	**28**	30	1		não troca
		24	25	26	**28**	**30**	1		não troca

4.5 Shellsort

O algoritmo de ordenação por shel foi criado por Donald L. Shell em 1959. Neste algoritmo, ao invés dos dados serem comparados com os seus vizinhos, é criado um *gap*. O *gap*, no início, é igual à parte inteira da divisão do número de elementos da lista por 2. Por exemplo, se a nossa lista tem 15 elementos, o *gap* inicial é igual a 7. São então comparados os elementos 1º. e 8º., 2º. e 9º., e assim por diante.

O que acontece, então? A maior parte dos elementos já vão para suas posições aproximadas. O 15, por exemplo, já é mandado para o fim da lista na primeira passagem, ao contrário do que acontece na ordenção de troca.

Ao terminar de passar por todos os elementos da lista, o *gap* é dividido por 2 e é feita uma nova passagem. Isso é repetido até que o *gap* seja igual a 0.

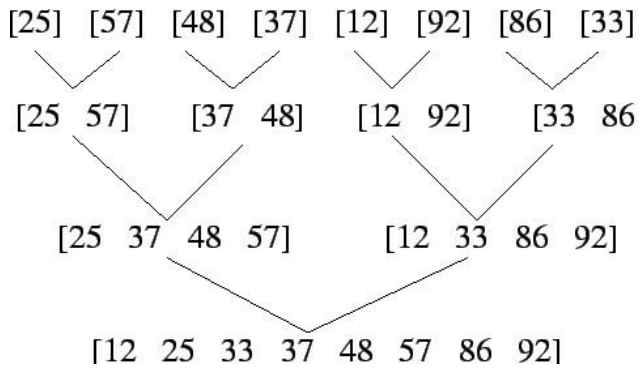
Vamos observar agora todos os passos realizados pelo algoritmo para os valores abaixo:

```
7| 11| 4| 2| 8| 5| 1| 6| 3| 9| 5|
5:
 5| 11| 4| 2| 8| 7| 1| 6| 3| 9| 5|
 5| 11| 4| 2| 8| 5| 1| 6| 3| 9| 7|
 5| 1| 4| 2| 8| 5| 11| 6| 3| 9| 7|
 5| 1| 4| 2| 8| 5| 11| 6| 3| 9| 7|
 5| 1| 4| 2| 8| 5| 11| 6| 3| 9| 7|
 5| 1| 4| 2| 8| 5| 11| 6| 3| 9| 7|

2:
 5| 1| 4| 2| 8| 5| 11| 6| 3| 9| 7|
 4| 1| 5| 2| 8| 5| 11| 6| 3| 9| 7|
 4| 1| 5| 2| 8| 5| 11| 6| 3| 9| 7|
 4| 1| 5| 2| 8| 5| 11| 6| 3| 9| 7|
 3| 1| 4| 2| 5| 5| 8| 6| 11| 9| 7|
 3| 1| 4| 2| 5| 5| 7| 6| 8| 9| 11|
 3| 1| 4| 2| 5| 5| 7| 6| 8| 9| 11|
 3| 1| 4| 2| 5| 5| 7| 6| 8| 9| 11|
 3| 1| 4| 2| 5| 5| 7| 6| 8| 9| 11|

1:
 1| 3| 4| 2| 5| 5| 7| 6| 8| 9| 11|
 1| 3| 4| 2| 5| 5| 7| 6| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 7| 6| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 7| 6| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 7| 6| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 7| 6| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 6| 7| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 6| 7| 8| 9| 11|
 1| 2| 3| 4| 5| 5| 6| 7| 8| 9| 11|
```

4.6 Mergesort

A idéia principal deste algoritmo é combinar duas listas já ordenadas. O algoritmo quebra um array original em dois outros de tamanhos menor recursivamente até obter arrays de tamanho 1, depois retorna da recursão combinando os resultados. Cada um dos dois segmentos possui um ponteiro (x e y) que é incrementado ao se selecionar o valor

Um dos principais problemas com o MergeSort é que ele faz uso de um array auxiliar. A figura abaixo ilustra a ação do MergeSort.



Exemplo:

23	17	8	15	9	12	19	7	4
17	23	8	15	9	12	7	19	4
x ↑		y ↑		x ↑		y ↑		

4.7 Quicksort

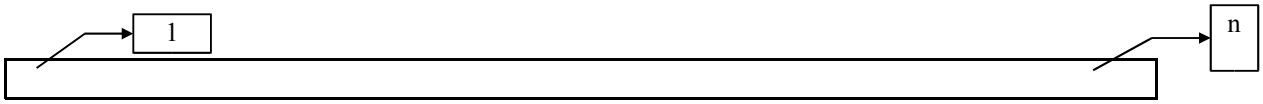
Este método de classificação foi inventado por Hoare [HOA62] e seu desempenho é o melhor na maioria das vezes.

O primeiro elemento da lista a ser classificada é escolhido como o pivô. Depois da primeira fase da classificação, o pivô ocupa a posição que ocupará quando a lista estiver completamente classificada. Os registros com valores de chaves menores do que o valor de chave do registro pivô o precedem na lista e os registros com valores de chaves maiores do que o valor de chave do registro pivô o sucedem na lista.

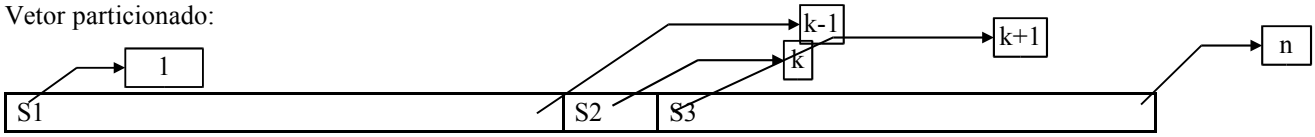
Cada registro é comparado com o registro pivô e suas posições são permutadas se o valor de chave do registro for maior e o registro preceder o registro pivô, ou se o valor de chave do registro for menor e o registro suceder o registro pivô. A posição do registro pivô no final de uma fase divide a lista original em duas sublistas (partições), cada uma delas precisando ser ordenada.

EXEMPLO:

Inicialmente, pega-se um vetor inicial e particiona-se da seguinte forma:



Vetor particionado:



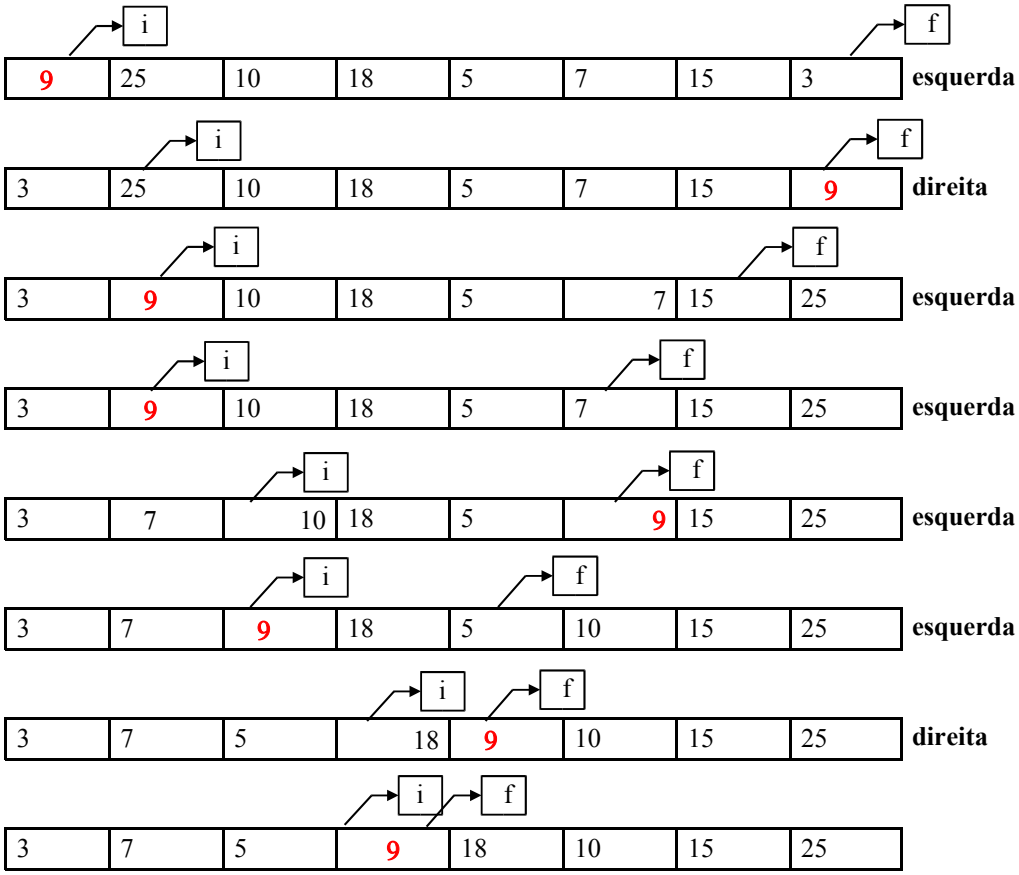
Suponhamos o vetor abaixo:

9	25	10	18	5	7	15	3

escolhemos a chave 9 como particionadora e a guardamos em uma variável **cp**. Com isso, a posição ocupada por ela se torna disponível para ser ocupada por outra chave. Essa situação é indicada pelo símbolo:

	25	10	18	5	7	15	3	**cp**=9

a partir daí, consideremos 2 ponteiros (**i**: início e **f**: fim)



observe então que embora os segmentos S1 e S3 não estejam ainda ordenados, a chave particionadora se encontra na sua posição definitivamente correta.

A seqüência a seguir exibe apenas o final de cada processo de particionamento. As chaves particionadas são mostradas em tipo **negrito**, enquanto os segmentos de apenas um elemento que se formarem são mostrados em tipo itálico.

3	7	5	**9**	15	10	**18**	*25*
3	*5*	**7**	**9**	*10*	**15**	**18**	**25**

Exercícios simples

Ordene, através de todos os métodos aprendidos, os elementos: 7, 11, 4, 2, 8, 5, 1, 6, 3, 9, 10

Seleção direta:

Inserção direta:

Pente (combosort):

Shellsort

Quicksort

Problemas propostos envolvendo classificação (Ordenação)

Os problemas apresentados a seguir foram retirados do livro Programming Challenges de S Skiena e M Revilla. São problemas de classificação que podem utilizar estruturas vistas anteriormente para a sua solução (pilhas, filas, etc).

Staks of Flapjacks (Pilhas de Panquecas) PC/UVA 110402/**120**

Pilhas e filas são frequentemente utilizadas em análise de sintaxe, arquitetura, sistemas operacionais, e simulação de eventos. Pilhas também são importantes na teoria de linguagens formais. Este problema envolve alguns destes conceitos de pilhas e filas aplicados a um método de virar panquecas, de acordo com um conjunto de regras.

Dada uma pilha de panquecas, deve-se escrever um programa que indica como a pilha pode ser classificado de modo que a maior panqueca fique no fundo e a menor panqueca fique no topo. O tamanho de uma panqueca é dado pelo diâmetro da panqueca. **Todas as panquecas em uma pilha têm diferentes diâmetros**.

O ordenamento de uma pilha é feito por uma seqüência de viradas (flips) nas panquecas. Uma virada consiste em inserir uma espátula entre duas panquecas em uma pilha e lançar (reverter) todas panquecas que estão acima da espátula (inversão da sub-pilha). Uma virada é especificada dando a posição da panqueca do fundo da sub-pilha a ser virada (com relação a toda a pilha). A panqueca do fundo da pilha tem posição 1 e a do topo tem posição n.

Uma pilha é especificada dando o diâmetro de cada panqueca na pilha na ordem em que aparecem as panquecas. Por exemplo, considere os três pilhas de panquecas abaixo (no qual a panqueca 7 é a mais alta da pilha de panquecas):

```
  7     8     2
  4     6     5
  6     4     7
  8     7     4
  5     5     6
  2     2     8
```

A pilha à esquerda pode ser transformada para a pilha no meio via flip (3). A pilha do meio pode ser transformada na pilha da direita através do comando flip (1), ou seja 3 1 0.

Entrada
A entrada consiste de uma seqüência de pilhas de panquecas. Cada pilha será composto de entre 1 e 30 panquecas e cada panqueca terá um diâmetro inteiro entre 1 e 100. A entrada é terminada por end-of-file. Cada pilha é dada como uma única linha de entrada com a panqueca do topo aparecendo por primeiro na linha, e a panqueca do fundo aparecendo por último, sendo que todas as panquecas são separadas por um espaço.

Saída
Para cada pilha de panquecas, a saída deve repetir a pilha original em uma linha, seguido por algumas seqüências de **flips** que resultam no ordenamento da pilha de panquecas sendo que a panqueca de maior diâmetro fica no fundo e a panqueca de menor diâmetro fica no topo. Para cada pilha a seqüência de viradas **(flips)** deve ser encerrado por um 0 (indicando que nenhum **flip** é mais necessário). Uma vez que a pilha está classificada, nenhum **flip** é mais necessário.

Sample Imput

```
1 2 3 4 5
5 4 3 2 1
5 1 2 3 4
13
1 3
3 1
2 7 6 3
71 4 12 9 6 5 3 21 121 63 2 8 7
```

Sample Output // Explicação – não faz parte da saída

```
1 2 3 4 5     //Repetição da entrada
0             //Nenhum flip é necessário
5 4 3 2 1     //Repetição da entrada
1 0           //Flip na 1ª panqueca. Após, nenhum flip necessário
5 1 2 3 4     //Repetição da entrada
1 2 0         //Flip 1ª, flip 2ª, Nenhum mais necessário.
13
0
1 3
0
3 1
1 0
2 7 6 3
3 1 3 2 0
71 4 12 9 6 5 3 21 121 63 2 8 7
5 1 9 2 4 9 5 10 6 11 7 9 11 12 0
```

ShellSort PC/UVA 110407/**10152**

“Ele fez cada uma das tartarugas ficar em cima da outra, acumulando todas elas em uma pilha com 9 tartarugas e, em seguida, Yertle subiu. Ele sentou em cima da pilha. Que vista maravilhosa! Ele pôde ver "mais uma milha!”

O rei Yertle pretende reorganizar seu trono de tartarugas colocando os nobres e assessores próximos ao topo. Uma única operação está disponível para alterar a ordem das tartarugas na pilha: uma tartaruga pode sair fora de sua posição na pilha e subir ao longo de outras tartarugas e sentar-se no topo.

Tendo em conta a pilha de tartarugas em sua disposição original e uma disposição exigida para esta mesma pilha, seu trabalho consiste em determinar o nível mínimo de seqüência de operações que transforma (reorganiza) a pilha original transformando-a na pilha exigida.

Entrada
A primeira linha da entrada consiste de um único inteiro K dando o número de casos de teste. Cada teste caso consistem em um inteiro n indicando o número de tartarugas na pilha. As próximas n linhas descrevem a ordem original da pilha de tartarugas. Cada uma das linhas contém o nome de uma tartaruga, que começa com a tartaruga do topo da pilha e segue com os nomes até a tartaruga do fundo da pilha. Tartarugas têm nomes exclusivos, cada um dos quais é uma cadeia de não mais de oitenta caracteres traçada a partir de um conjunto de caracteres consistindo de caracteres alfanuméricos, espaços e o ponto(.) As próximas n linhas na entrada dão a ordenação pretendida da pilha, mais uma vez, por nomeação das tartarugas de cima até embaixo da pilha. Cada caso de teste consiste de exatamente 2n +1 linhas no total. O número de tartarugas (n) será igual ou inferior a 200.

Saída
Para cada caso teste, a saída consiste de uma seqüência de nomes de tartaruga, uma por linha, indicando a ordem em que as tartarugas devem abandonar as suas posições na pilha e subir para o topo. Esta seqüência de operações deve transformar a pilha de tartarugas (da pilha original para a pilha exigida) e deve ser a solução o mais curta possível. Se mais do que uma solução de menor comprimento é possível, qualquer das soluções podem ser reportadas como saída. Imprima uma linha em branco depois de cada teste.

Sample Imput
```
2
3
Yertle
Duke of Earl
Sir Lancelot
Duke of Earl
Yertle
Sir Lancelot
9
Yertle
Duke of Earl
Sir Lancelot
Elizabeth Windsor
Michael Eisner
Richard M. Nixon
Mr. Rogers
Ford Perfect
Mack
Yertle
Richard M. Nixon
Sir Lancelot
Duke of Earl
Elizabeth Windsor
Michael Eisner
Mr. Rogers
Ford Perfect
Mack
```

Sample Output
```
Duke of Earl

Sir Lancelot
Richard M. Nixon
Yertle
```

Longest Nap (Cochilo mais longo) PC/UVA 110404/**10191**

Como você deve saber, existem professores muito ocupados e com um calendário cheio durante o dia. Seu professor, vamos chamá-lo Professor P, é um pouco preguiçoso e quer tirar uma soneca durante o dia, mas, como o seu calendário é muito ocupado, ele não tem muitas chances de fazer isso. Ele realmente deseja tirar apenas uma soneca todos os dias. Como ele vai tirar apenas uma soneca, ele quer tirar a mais longa soneca possível dado o seu calendário. Ele decidiu escrever um programa para ajudá-lo nessa tarefa, mas, como dissemos, Professor P é muito preguiçoso e portanto, ele decidiu finalmente que VOCÊ deve escrever o programa!

A entrada
A entrada consistirá em um número arbitrário de casos de teste, cada caso teste representa um dia. A primeira linha de cada conjunto contém um número inteiro positivo s (não superior a 100), representando o número de reuniões programadas durante esse dia. Nas próximas s linhas existem atividades no seguinte formato: *Time1 time2 atividade*

Onde *time1* representa o tempo que começa a atividade e *time2* o tempo que termina. Todos os horários serão no formato *hh:mm*, *time1* será sempre estritamente inferior a time2, eles serão separados por um espaço simples e todos os tempos será superior ou igual a 10:00 e inferior ou igual a 18:00. Portanto, a resposta deve ser, neste intervalo também, ou seja, nenhuma soneca pode começar antes das 10:00 e depois das 18:00. A *atividade* pode ser qualquer seqüência de caracteres, mas será sempre na mesma linha. Pode-se supor que nenhuma linha será mais longo do que 255 caracteres, que 10 <= hh <= 18 e que 0 <= minutos <60. ***Você não pode assumir, no entanto, que a entrada será, em qualquer ordem específica***. Você deve ler a entrada até chegar ao final do arquivo.

A saída
Para cada caso teste, você deve imprimir a linha a seguir:
`Day #d: the longest nap start at hh:mm and will last for [H hours and] M minutes.`
Onde d significa o número de caso de teste (a partir de 1) e hh:mm é o momento em que o cochilo pode começar.
Para exibir a duração da soneca, siga estas regras simples:
1. Se a duração total X for menos de 60 minutos é, basta imprimir "M minutes.", onde M = X.
2. Se a duração total X em minutos é maior ou igual a 60, imprimir "H hours and M minutes.", onde H = X / 60 (inteiro da divisão, é claro) e M = X mod 60.

Não deve-se preocupar com a concordância (ou seja, deve-se imprimir "1 minutes" ou "1 hours" se for o caso). A duração da soneca é calculada pela diferença entre o início e o fim do tempo livre, ou seja, se uma atividade termina às 14:00 e o próximos começa às 14:47, então tem-se (14:47-14:00)= 47 minutos para um possível cochilo.

Se houver mais de uma maior cochilo com a mesma duração, imprima o que acontece mais cedo. Pode-se supor que não haverá um dia todo ocupado (ou seja, você pode considerar que professor não estará ocupado todo o dia e sempre haverá tempo para pelo menos um cochilo).

Sample Input
```
4
10:00 12:00 Lectures
12:00 13:00 Lunch, like always.
13:00 15:00 Boring lectures...
15:30 17:45 Reading
4
10:00 12:00 Lectures
12:00 13:00 Lunch, just lunch.
13:00 15:00 Lectures, lectures... oh, no!
16:45 17:45 Reading (to be or not to be?)
4
10:00 12:00 Lectures, as everyday.
12:00 13:00 Lunch, again!!!
13:00 15:00 Lectures, more lectures!
15:30 17:15 Reading (I love reading, but should I schedule it?)
1
12:00 13:00 I love lunch! Have you ever noticed it? :)
```

Sample Output
```
Day #1: the longest nap starts at 15:00 and will last for 30 minutes.
Day #2: the longest nap starts at 15:00 and will last for 1 hours and 45 minutes.
Day #3: the longest nap starts at 17:15 and will last for 45 minutes.
Day #4: the longest nap starts at 13:00 and will last for 5 hours and 0 minutes.
```

Vito's Family PC/UVA 110401/**10041**

O gangster mundialmente conhecido Vito Deadstone está indo para Nova York. Ele tem uma grande família lá e todos eles vivem na avenida Lamafia. Uma vez que ele visita todos os seus parentes muitas vezes, ele está procurando uma casa perto deles para morar. Vito pretende minimizar a distância total a todos eles e chantageou você para escrever um programa que resolva o seu problema.

Entrada

A entrada é constituído por vários casos de teste. A primeira linha contém o número de casos de teste. Para cada caso teste ser-lhe-á dado o número inteiro de parentes r (0 <r <500), bem como o número da rua (inteiros também) s1, s2, ... si, ... sr onde vivem (0 < si <30000). Note-se que vários parentes podem viver no mesmo número de rua (na mesma rua).

Saída

Para cada caso de teste seu programa deve escrever a mínima soma das distâncias entre a posição da casa de Vito (que deverá ser uma posição ótima em relação às casas de seus parentes) e a casa de cada um de seus parentes. A distância entre dois números de ruas si e sj é dij = | si-sj |.

Sample Imput

```
2             // 2 casos de teste que vem a seguir:
2 2 4         // 2 parentes, um mora na rua 2 e outro na rua 4
3 2 4 6       // 3 parentes, um mora na rua 2, outro na rua 4 e outro na rua 6
5 6 1 2 4 6   // 5 parentes, um mora na rua 1, outro na rua 2,  outro na rua 4 e 2 na rua 6
4 3 3 3 4     // 4 parentes, 3 moram na rua 3 e um mora na rua 4
```

Sample Output

```
2             // Vito moraria na rua 2 ou 4
4             // Vito moraria na rua 4
9             // Vito moraria na rua 4
1             // Vito moraria na rua 3
```

Problemas (exercícios) complementares (http://icpcres.ecs.baylor.edu/onlinejudge/):

Bridge PC/UVA 110403/**10037**
Shoemaker's Problem PC/UVA 110405/**10026**
CDVII PC/UVA 110406/**10138**
Football PC/UVA 110408/**10194**

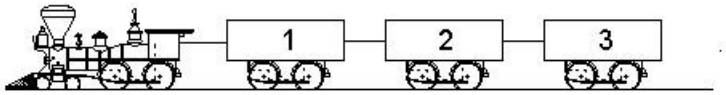
6. Listas ligadas ou encadeadas – Alocação Estática

Existem duas formas para inserir alguns elementos mantendo a ordem lógica destes elementos. São elas:
a) deslocar todos os demais elementos
b) manter um campo NEXT para cada elemento, indicando qual é o próximo

* é como se fosse um trem, com um campo INFO que contém uma ou mais informações e um campo NEXT, que contém o endereço seguinte dentro da estrutura.

INFO	NEXT ou ELO ou LINK
Informações	ligaçao

Vejamos o exemplo de um trem. A locomotiva é ligada ao primeiro vagão e cada vagão é ligado ao próximo.



Existem algumas informações relevantes relacionadas aos vagões, tais como:
a) Tipo: (1) restaurante, (2) passageiros, (3) carga
b) Capacidade: toneladas, pessoas
c) Responsável: quem é o responsável pelo vagão
d) conexão: qual é o vagão seguinte?

Desta forma, devemos pensar como seria a implementação de uma estrutura com os nodos ligados ou encadeados. O encadeamento deve considerar vários aspectos, os quais são relacionados a seguir.

6.1 Implementação de listas em vetores

- A implementação utiliza um vetor (ou matriz) para o campo INFO e um vetor para o campo NEXT.
- A ordem lógica dos elementos é obtida através do campo NEXT, não importando a localização física dentro da estrutura.
- O campo NEXT do último elemento é igual a **-1**, indicando que ele é o último elemento.
- Basta saber em que local está o primeiro elemento da lista para se conhecer todos os demais elementos.
- A lista pode ter um nodo de Controle (**NC**), que seria a locomotiva. Na implementação das listas sobre vetores ele é indispensável.

6.1.1 Estrutura da Lista

Devido ao fato da estrutura ser encadeada, a lista deve ser implementada sobre 2 vetores, e não somente um como era feito na alocação sequencial. Como exemplo, poderemos utilizar os vetores **info** e **next**.

Além dos dois vetores (**info** e **next**), devemos ter a informação **disp**, que indicará qual será o nodo que está disponível para inserção. Uma informação referente ao Nodo de Cabeçalho também pode ser utilizada. Para facilitar, supomos que o Nodo de Cabeçalho será sempre o primeiro nodo do vetor.

```
typedef struct {
   int info [10];
   int next [10];
   int disp, primeiro, ultimo;
} LISTA;

LISTA lista;
```

```
class lista{
   private:
      int info[10], next[10];
      int primeiro, ultimo, disp;
   public:
      void CriaPnd (int elem);
}
```

6.1.2 Pilha dos Nodos Disponíveis

Serve para guardar os nodos que estão disponíveis. Todos os nodos que estão disponíveis, isto é, todas as posições dos vetores **info** e **next** que alocam a lista e não pertençam à mesma, formam uma estrutura à parte do tipo pilha, denominada PND. Voltando ao exemplo do trem, a **PND** controla os vagões vazios que estão no estacionamento.

O topo desta pilha fica apontado por **disp**. Uma rotina denominada CriaPND deve ser criada. O objetivo desta rotina é simplesmente encadear os nodos da estrutura. O **next** do primeiro nodo aponta para o segundo, que aponta para o terceiro, que aponta para o quarto, e assim sucessivamente, até que o último nodo aponta para -1, indicando o término dos nodos disponíveis. Ao final do encadeamento, a estrutura deve ficar como a apresentada abaixo (supondo 7 nodos disponíveis no total para a lista – de 0 a 6).

	Info	Next
0		1
1		2
2		3
3		4
4		5
5		6
6		-1

disp=0 **primeiro**= -1 **ultimo=** -1

Rotina para a Criação da PND (Pilha dos Nodos Disponíveis):

```
void lista::criapnd (int elementos) { // cria a pnd
   for (int i = 0; i<elementos; i++) {
      next[i] = i+1;
   }
   next[elementos-1] = -1;
   disp = 0;
   primeiro = ultimo = -1;
}
```

Sem a criação da **PND**, fica muito difícil saber onde inserir e quantos nodos ainda estão disponíveis.

6.1.3 Inserção em uma lista

A inserção na lista poderá ser feita:
– no início da mesma
– no final da mesma
– numa determinada posição ***n***
– na posição correta para que a lista permaneça ordenada (no caso de uma lista ordenada)

	Info	Next
0	176	-1
1		2
2		3
3		4
4		5
5		6
6		-1

disp=1 **primeiro**= 0 **ultimo=** 0

Como exercício, insira os valores 57, 201, 12 e 19. (Insira cada elemento no final da lista)

	INFO	NEXT
0		
1		
2		
3		
4		
5		
6		

disp= primeiro = ultimo=

	INFO	NEXT
0		
1		
2		
3		
4		
5		
6		

disp= primeiro = ultimo=

	INFO	NEXT
0		
1		
2		
3		
4		
5		
6		

disp= primeiro = ultimo=

	INFO	NEXT
0		
1		
2		
3		
4		
5		
6		

disp= primeiro = ultimo=

Algoritmo para Inserção no **final** de uma lista (válido também para uma fila encadeada):

```
void lista::inserefim (int elem) { //insere no final da pnd








}
```

Algoritmo para Inserção no **início** de uma lista:

```
void lista::insereinicio (int x) { //insere no início da pnd








}
```

Inserção mantendo os elementos ordenados.

Supondo que se deseje inserir os elementos mantendo a lista ordenada logicamente, uma variação da rotina de inserção deve ser criada. Agora insira de forma visual os elementos **12**, **1, 214** e **76**, de modo que ao percorrer a lista, a mesma esteja ordenada por ordem alfabética ascendente.

	INFO	NEXT		INFO	NEXT		INFO	NEXT		INFO	NEXT
0			0			0			0		
1			1			1			1		
2			2			2			2		
3			3			3			3		
4			4			4			4		
5			5			5			5		
6			6			6			6		

disp= primeiro = ultimo= **disp= primeiro = ultimo=** **disp= primeiro = ultimo=** **disp= primeiro = ultimo=**

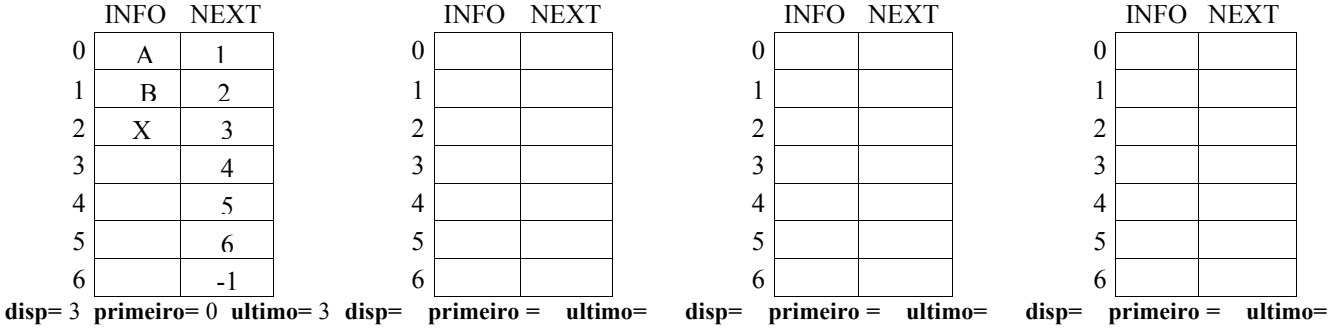
Algoritmo para Inserção na Lista mantendo-a ordenada:

Supondo que se deseje inserir os elementos mantendo a lista ordenada logicamente, desenvolva o algoritmo para tal função.

```
void lista::InsereOrdem (int elem) { //insere mantendo ordem ano final da pnd













} DESENVOLVER EM LABORATÓRIO
```

6.1.4 Retirada em uma lista

Para simplificar, consideramos que o valor a ser retirado da lista seja informado pelo usuário. No caso abaixo, a lista contém alguns elementos. Demonstre graficamente, passo-a-passo como ficaria a retirada dos nodos X, A e B, em sequência.



Crie então o algoritmo para retirada de uma lista encadeada:

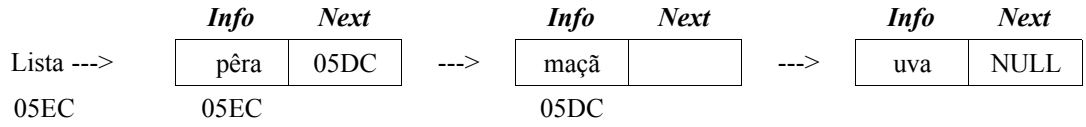
```
void lista::RetiraElem (int elem) { //Retira um elemento específico da lista

}
```

7. Listas ligadas ou encadeadas – Alocação dinâmica

A desvantagem da alocação estática é clara: quando se utiliza uma estrutura linear, uma quantidade de bytes fica alocada mesmo sem ser utilizada. Os dois vetores (INFO e NEXT) são utilizados apenas em parte e a complexidade aumenta exponencialmente quando se deseja implementar 2 ou mais listas sobre a mesma área (vetores).

A estrutura da lista ligada dinâmica é muito simples. Existem somente 2 campos (INFO, NEXT), sendo que o campo next é um ponteiro para o próximo elemento da lista.

Podemos visualisar os nodos como apresentado abaixo



O Info pode ser um inteiro, um caracter ou uma string. Para ilustrar consideraremos abaixo a criação de uma lista cujo info é uma string de tamanho 10:

```
typedef struct LISTA {



} lista;
```

7.1 Inclusão de elementos no início da lista (Pilha)

Ao considerar as estruturas de dados do tipo lista, pilha ou fila, pode-se verificar que elas são idênticas. Serão sempre compostas por um campo ou mais de informações e um campo Next, que aponta para o próximo nodo.

Com relação à manipulação dos dados, os algoritmos de manipulação das pilhas são os mais simples, seguidos pelos algoritmos das filas e listas. Nas listas a complexidade aumenta por possibilitar inserções e retiradas de qualquer nodo que pertença à estrutura.

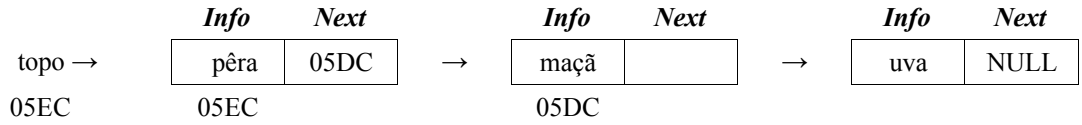
Como os algoritmos da pilha são mais simples, esta será a primeira estrutura apresentada.

```
struct PILHA {



} pilha;

struct pilha *topo, *aux;
```

No início do programa, antes de qualquer operação com a lista, a mesma deve ser inicializada com NULO, indicando que está vazia.

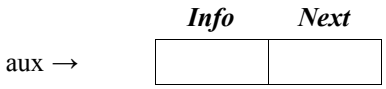
```
topo = aux = NULL;
```

Considerando uma pilha existente com alguns nodos:



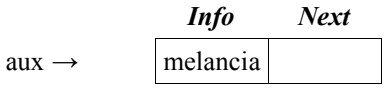
a) O primeiro passo é obter um nodo para inserí-lo na lista:

```
aux = new pilha;
```



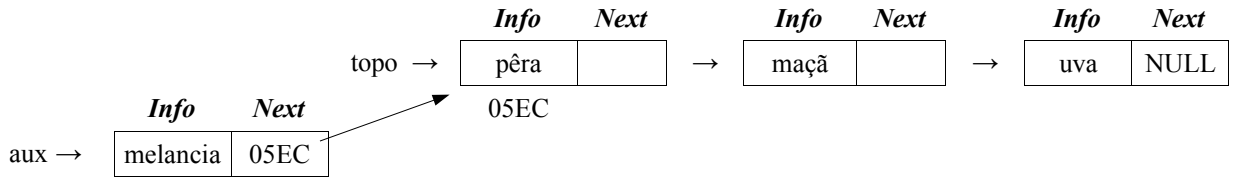
b) O próximo passo é inserir o valor desejado no nodo recém-alocado.

```
Cin >> info;
```



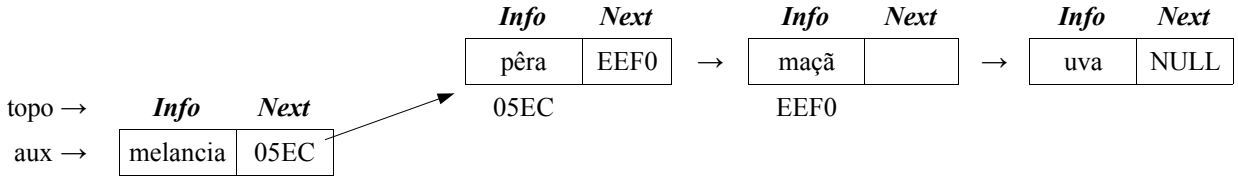
c) Depois deve-se fazer o next do nodo inserido apontar para o início da lista.

```
aux->next = topo;
```



d) por último deve-se fazer o ponteiro pilha apontar para onde o ponteiro `aux` aponta.

```
topo = aux;
```



Reunindo os passos a,b,c e d, temos:

```
aux = new pilha;
cin >> info;
aux->next = topo;
topo = aux;
```

a) Com base na explicação acima, faça a rotina para excluir um elemento da pilha.
b) Faça um programa completo para incluir elementos na pilha. No momento que for digitado um <enter> para a Informação interrompa a leitura programa e mostre todos os elementos que foram inseridos na pilha (`if (strcmp(aux->info,"")==0) break`).

7.2 Inclusão de elementos em uma fila

Na fila, os dados são inseridos no final da estrutura. A definição da estrutura permanece a mesma, alterando somente o nome da mesma.

```
typedef struct FILA {


} fila ;

fila *frente, *re, *aux;
```

No início do programa o fundamental é inicializar os 3 ponteiros para a estrutura:

```
frente = re = aux = NULL;
```

a) O primeiro passo é obter um nodo para inserí-lo na lista.

```
aux = new fila;
```

	Info	*Next*
aux →		NULL

b) Após insere-se o valor desejado no nodo recém-alocado, atualizando o seu campo **Next** para **NULL**:

```
cin >> info;
aux->next = NULL;
```

	Info	*Next*
aux →	pêra	NULL
05EC	05EC	

c) Deve-se encadear em seguida o nodo inserido (A) . Para ilustrar, considera-se aqui a inserção de um segundo elemento (B).

```
if (re=NULL){      //A fila está vazia
A   frente = aux;
else {             //A fila já contém alguns elementos
B   re->next = aux;
}
re = aux;
```

(A)

		Info	*Next*
frente →			
aux →		pêra	NULL
re →		05EC	
05EC			

(B)

	Info	*Next*		*Info*	*Next*
frente →			re →		
aux →	pêra	05DC	→	**melancia**	**NULL**
05EC	05EC		05DC	05DC	

Com base nas explicações acima, faça a rotina para excluir um elemento da pilha. Faça também um programa completo para incluir / excluir elementos da fila, mostrando-a no final.