

**Universidade Regional Integrada do
Alto Uruguai e das Missões
URI – Campus de Erechim**

**Curso de Graduação em
Ciência da Computação**

Algoritmos e Estrutura de Dados II

Utilizando o Portal

URI Online Judge



www.urionlinejudge.edu.br

Neilor Tonin

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 núm. inteiros **double:** compreende núm. reais **char/string:** compreende 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;

double 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);
}

double media (int vet2[10]){
    double 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:

<pre>struct { char primeiro[10]; char inicialmeio; char ultimo[20]; } nome;</pre>	<pre>typedef struct { char primeiro[10]; char inicialmeio; char ultimo[20]; } TIPONOME; TIPONOME nome, snome, fulano;</pre>
---	--

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.

$$\frac{A0}{A1} + \frac{B0}{B1} = \frac{A0 * B1 + A1 * B0}{A1 * B1}$$

Implementação do tipo RACIONAL

Inicialmente define-se uma estrutura denominada racional, contendo 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){
    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);
}
```

Exercício:

URI Online Judge | 1022

TDA Racional

Por Neilor Tonin, URI  Brasil

Timelimit: 1

A tarefa aqui neste problema é ler uma expressão matemática na forma de dois números Racionais (numerador / denominador) e apresentar o resultado da operação. Cada operando ou operador é separado por um espaço em branco. A sequência de cada linha que contém a expressão a ser lida é: número, caractere, número, caractere, número, caractere, número. A resposta deverá ser apresentada e posteriormente simplificada. Deverá então ser apresentado o sinal de igualdade e em seguida a resposta simplificada. No caso de não ser possível uma simplificação, deve ser apresentada a mesma resposta após o sinal de igualdade.

Entrada

A entrada contém vários casos de teste. A primeira linha de cada caso de teste contém um inteiro N ($1 \leq N \leq 10^4$), indicando a quantidade de casos de teste que devem ser lidos logo a seguir. Cada caso de teste contém um valor racional X ($1 \leq X \leq 1000$), uma operação ($-$, $+$, $*$ ou $/$) e outro valor racional Y ($1 \leq Y \leq 1000$).

Saída

A saída consiste em um valor racional, seguido de um sinal de igualdade e outro valor racional, que é a simplificação do primeiro valor. No caso do primeiro valor não poder ser simplificado, o mesmo deve ser repetido após o sinal de igualdade.

Exemplo de Entrada	Exemplo de Saída
4	10/8 = 5/4
1 / 2 + 3 / 4	-2/8 = -1/4
1 / 2 - 3 / 4	12/18 = 2/3
2 / 3 * 6 / 6	4/6 = 2/3
1 / 2 / 3 / 4	

Exercício:

URI Online Judge | 1028

Figurinhas

Por Neilor Tonin, URI  Brasil

Timelimit: 1



Ricardo e Vicente são aficionados por figurinhas. Nas horas vagas eles arrumam um jeito de jogar um “bafo” ou algum outro jogo que envolva tais figurinhas. Ambos também têm o hábito de trocarem as figuras repetidas com seus amigos e certo dia pensaram em uma brincadeira diferente. Chamaram todos os amigos e propuseram o seguinte. Com as figurinhas em mãos, então cada um tentava fazer uma troca com o amigo que estava mais perto seguindo a seguinte regra: cada um contava quantas figurinhas tinha. Em seguida, eles tinham que dividir as figurinhas de cada um em pilhas do mesmo tamanho, no maior tamanho que fosse possível para ambos. Daí então cada um escolhia uma das pilhas de figurinhas do amigo para receber. Por exemplo, se Ricardo e Vicente fossem trocar as figurinhas e tivessem respectivamente 8 e 12 figuras, ambos dividiam todas as suas figuras em pilhas de 4 figuras (Ricardo teria 2 pilhas e Vicente teria 3 pilhas) e ambos escolhiam uma pilha do amigo para receber.

Entrada

A primeira linha da entrada contém um único inteiro N ($1 \leq N \leq 3000$), indicando o número de casos de teste. Cada caso de teste contém 2 inteiros $F1$ ($1 \leq F1 \leq 1000$) e $F2$ ($1 \leq F2 \leq 1000$) indicando, respectivamente, a quantidade de figurinhas que Ricardo e Vicente têm para trocar.

Saída

Para cada caso de teste de entrada haverá um valor na saída, representando o tamanho máximo da pilha de figurinhas que poderia ser trocada entre dois jogadores.

Exemplo de Entrada	Exemplo de Saída
3 8 12 9 27 259 111	4 9 37

Exercício:

Faça um algoritmo **recursivo** leia 1 número N . Este N é a quantidade de casos de teste que vem a seguir. Para cada caso de teste, leia um valor T que é o termo a calcular de uma sequência de valores, cujo primeiros 2 valores são 1 e 2 respectivamente e cada valor seguinte é igual a $n-1 * n-2 - 0.5$

Exemplo de uma entrada	Saída para o exemplo de entrada
2	2
2	1.5
3	

Tudo o que Você Precisa é Amor

Maratona de Programação da SBC 2001*  Brazil

Timelimit: 1

"All you need is love. All you need is love. All you need is love, love... love is all you need."
The Beatles

Foi inventado um novo dispositivo poderoso pela Beautifull Internacional Machines Corporation chamado de "Máquina do amor!". Dada uma string feita de dígitos binários, a máquina do amor responde se isto é feito somente de amor, ou seja, se tudo o que você irá precisar para construir aquela string for somente amor. A definição de amor para a Máquina do amor é outra string de dígitos binários, fornecida por um operador humano. Vamos supor que nós temos uma string L que representa "love" e forneçamos uma string S para a máquina do amor. Diremos então que tudo o que você precisa é amor para construir S se pudermos repetidamente subtrair L de S até que sobre apenas L. A subtração definida aqui é a mesma subtração aritmética binária na base 2. Por definição é fácil de ver que $L > S$ (em binário), então S não é feito de amor. Se $S = L$ então S é obviamente feito de amor.

Por exemplo, suponha $S = "11011"$ e $L = "11"$. Se repetidamente subtrairmos L de S, obteremos: 11011, 11000, 10101, 10010, 1111, 1100, 1001, 110, 11. Portanto, dado este L, tudo o que você necessita é amor para construir S. Devido a algumas limitações da Máquina do Amor, não será possível lidar com strings com zero à esquerda. Por exemplo "0010101", "01110101", "011111" etc. são string Inválidas. Strings que contenham apenas um dígito também são strings inválidas (isto é outra limitação).

Sua tarefa para este problema é: dadas duas strings binárias válidas, S1 e S2, veja se é possível ter uma string L válida tal que ambas, S1 e S2 possam ser feitas apenas de L (i.e. dadas duas strings válidas S1 e S2, indique se existe pelo menos uma string L válida tal que ambas S1 e S2 sejam feitas apenas de L). Por exemplo, para $S1 = 11011$ e $S2 = 11000$, nós podemos ter $L = 11$ tal que S1 e S2 são feitas ambas somente de L (como pode ser visto no exemplo abaixo).

Entrada

A primeira linha de entrada contém um valor inteiro positivo N N ($N < 10000$) que indica o número de casos de teste. Então, $2*N$ linhas vem a seguir. Cada par de linhas consiste de um caso de teste. Cada par de linhas contém respectivamente S1 e S2 que serão inseridas como entrada para a máquina do amor. Nenhuma string conterà mais do que 30 caracteres. Você pode assumir que as strings de entrada serão válidas e estarão de acordo com as regras acima.

Saída

Para cada par de strings, seu programa deve imprimir uma das seguintes mensagens: Pair #p: All you need is love! Ou Pair #p: Love is not all you need! Onde p representa o número do par de entrada (que inicia em 1). Seu programa deve imprimir a primeira mensagem no caso de existir pelo menos uma string L válida tal que ambas strings S1 e S2 possam ser feitas somente de L. Caso contrário, imprima a segunda linha.

Exemplo de Entrada	Exemplo de Saída
5 11011 11000 11011 11001 111111 100 1000000000 110 1010 100	Pair #1: All you need is love! Pair #2: Love is not all you need! Pair #3: Love is not all you need! Pair #4: All you need is love! Pair #5: All you need is love!

* Maratona de Programação da SBC - ACM ICPC - 2001 - Warmup. Adaptado por Neilor.

Dicas para Resolução:

- Ler o **N** (numero de casos de teste)
- Para cada caso de teste, ler **S1** e **S2**
- Converter ambos de binário para decimal: utilize a função **strtol(, , 2)**
- Se $s1 < s2$ trocar os 2: **swap(s1,s2)**
- $result = GCD(s1,s2)$
- Se $(result > 1)$
 - Apresentar na tela: "All you need is love!";
- senão
 - Apresentar na tela: "Love is not all you need!";

2.1. Busca Binária

A busca binária consiste em posicionar no meio de uma sequência de elementos. A partir daí, busca-se novamente ou para baixo ou para cima, de acordo com o elemento a ser procurado. Ex.: ao se procurar o 47.

-93	-53	-41	-23	12	34	45	47	58	84
0	1	2	3	4	5	6	7	8	9
high=9		low=0		Mid = $0 + (9-0)/2 \rightarrow 4$					

- A partir daí executa-se novamente a rotina, trocando o low por mid+1, ou seja:

high=9 low=5 Mid= $5 + (9-5)/2 \rightarrow 7 \Rightarrow$ achou

-93	-53	-41	-23	12	34	45	47	58	84
0	1	2	3	4	5	6	7	8	9

Busca Binária Recursiva

Segue o código em C++ para busca binária recursiva, lembrando que o mesmo pode ser implementado também de forma não-recursiva.

```
int BS(int vet[], int low, int high, int valor){
    if(high < low){
        return -1;
    }
    int mid = low + ((high - low) / 2);

    if(vet[mid] > valor){
        return BS(vet, low, mid-1, valor);
    }

    if(vet[mid] < valor){
        return BS(vet, mid+1, high, valor);
    }


    return mid;
}
```

Analise a execução do código acima para o exemplo apresentado na página anterior.

Exercício:

URI Online Judge | 1025

Onde está o Mármore?

Por Monirul Hasan Tomal, SEU  Bangladesh

Timelimit: 2

Raju e Meena adoram jogar um jogo diferente com pequenas peças de mármore, chamados Marbles. Eles têm um monte destas peças com números escritos neles. No início, Raju colocaria estes pequenos mármore um após outro em ordem ascendente de números escritos neles. Então Meena gostaria de pedir a Raju para encontrar o primeiro mármore com um certo número. Ele deveria contar 1...2...3. Raju ganha um ponto por cada resposta correta e Meena ganha um ponto se Raju falha. Depois de um número fixo de tentativas, o jogo termina e o jogador com o máximo de pontos vence. Hoje é sua chance de jogar com Raju. Sendo um/a cara esperto/a, você tem em seu favor o computador. Mas não subestime Meena, ela escreveu um programa para monitorar quanto tempo você levará para dar todas as respostas. Portanto, agora escreva o programa, que ajudará você em seu desafio com Raju.

Entrada

A entrada contém vários casos de teste, mas o total de casos é menor do que 65. Cada caso de teste inicia com dois inteiros: N que é o número de mármore e Q que é o número de consultas que Mina deseja fazer. As próximas N linhas conterão os números escritos em cada um dos N mármore. Os números destes mármore não tem qualquer ordem em particular. As seguintes Q linhas will conter Q consultas. Tenha certeza, nenhum dos números da entrada é maior do que 10000 e nenhum deles é negativo.

A entrada é terminada por um caso de teste onde N = 0 e Q = 0.

Saída

Para cada caso de teste de saída deve haver um número serial do caso de teste. Para cada consulta, escreva uma linha de saída. O formato desta linha dependerá se o número consultado estiver ou não escrito em um dos mármore.

Os dois diferentes formatos são descritos abaixo:

'x found at y', se o 1º marble x foi encontrado na posição y. Posições são numeradas de 1, 2,.. a N.
'x not found', se o marble com o número x não estiver presente.

Exemplo de Entrada	Exemplo de Saída
4 1 2 3 5 1 5 5 2 1 3 3 3 1 2 3 0 0	CASE# 1: 5 found at 4 CASE# 2: 2 not found 3 found at 3

Dicas para Resolução:

- Ordene o vetor que contém todos os valores(**marbles**) utilizando o Sort.
- Utilize a busca binária recursiva para encontrar cada um dos **marbles**. Após encontrar o valor, vá retornando uma posição no vetor enquanto o valor for igual ao valor procurado.

Algoritmo de Josephus:

Josephus foi um historiador judeu do primeiro século. A lenda conta que ele e 40 de seus homens estavam presos em uma caverna pelos Romanos. Eles decidiram suicidar-se à ser capturados. Formaram um círculo e começaram a se matar, da 3ª à 3ª a pessoa em ordem da fila. Como Josephus não queria morrer, ele foi capaz de escolher o melhor lugar, a fim de se entregar e sobreviver.

O Problema: Na prática o problema teria N pessoas em um círculo, iria ser percorrido M - 1 pessoas e o M iria ser morto, assim por diante, até sobrar uma única pessoa. Este é um problema trivial de SIMULAÇÃO onde você executa o código até restar apenas um elemento.

A Solução:

Existem duas formas de resolver o problema. A primeira e mais genérica seria utilizando listas ligadas(linked lists). Porém isso somente será visto mais adiante. A outra forma, mais simples, é aplicar uma fórmula recursiva encontrada na página do wikipedia (http://en.wikipedia.org/wiki/Josephus_problem), com complexidade O(n).

A fórmula disponível na página é a : $f(n,k) = (f(n-1,k) + k) \bmod n$, with $f(1,k) = 0$,

Exercício: Resolva agora os problemas UOJ 1030, 1031 e 1032 utilizando a fórmula da página acima. Basta transformar a fórmula para uma recursividade.

```
#include <iostream>
#include <cstdlib>

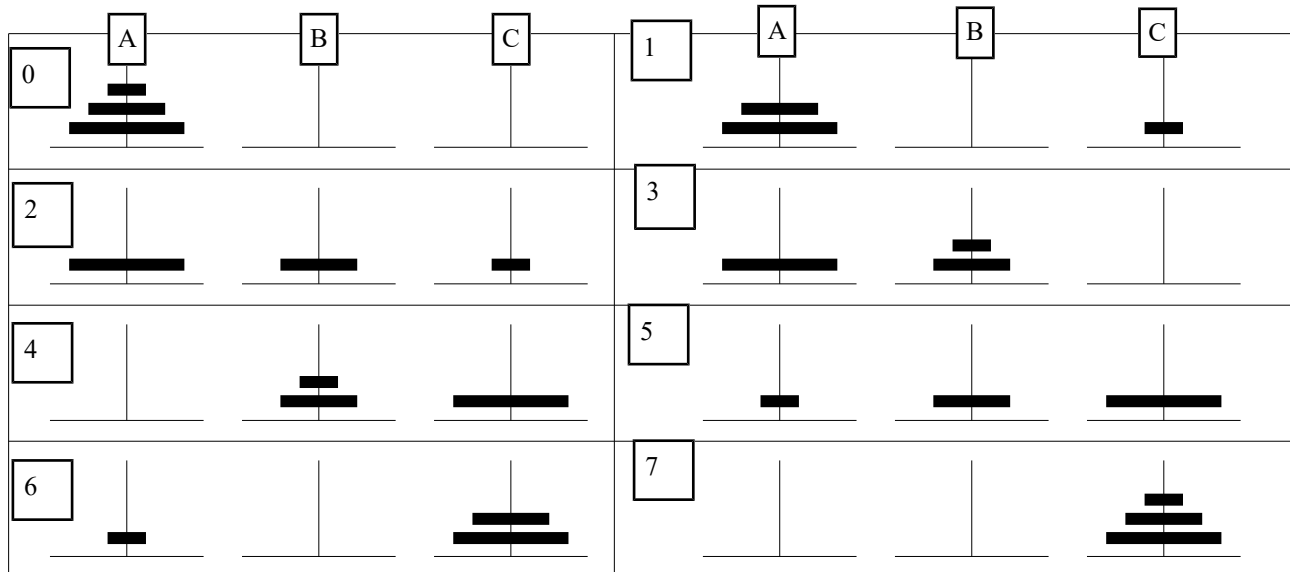
using namespace std;

int f(int n, int k) {
    if (n==1) return 0;
    return ((f(n-1...));
}

int main(void) {
    int casos, N,M;
    cin >> casos;
    for (unsigned int h=1; h<=casos; h++){
        cin >> N >> M;
        printf("Case %d: %d \n", h, f(N,M)+1 ); // corrigir
    }
}
```

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 (origem) para C
senão
    transfira  $n-1$  discos de A (origem) para B (destino), usando C (auxiliar)
    mova o último disco de A (origem) para C (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 sequência das 3 torres, os parâmetros devem ser manipulados na chamada recursiva da rotina Hanoi, respeitando a lógica dos algoritmos:

Programa principal:

Rotina Hanoi:

```

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

Exercício:

A2_R6 - Hanoi

Faça um algoritmo leia 1 número **N**. Este N é a quantidade de casos de teste que vem a seguir. Para cada caso de teste, leia um valor **M** que é a quantidade de discos (1-25) da torre de **Hanoi**. Calcule e mostre então quantos movimentos são necessários para mover todos os discos da torre **A** para a torre **C**, usando a torre **B** como auxiliar

Exemplo de uma entrada	Saída para o exemplo de entrada
2	Mova disco 1 da torre A para B
4	Mova disco 2 da torre A para C
2	Mova disco 1 da torre B para C
	Mova disco 3 da torre A para B
	Mova disco 1 da torre C para A
	Mova disco 2 da torre C para B
	Mova disco 1 da torre A para B
	Mova disco 4 da torre A para C
	Mova disco 1 da torre B para C
	Mova disco 2 da torre B para A
	Mova disco 1 da torre C para A
	Mova disco 3 da torre B para C
	Mova disco 1 da torre A para B
	Mova disco 2 da torre A para C
	Mova disco 1 da torre B para C
	Mova disco 1 da torre A para B
	Mova disco 2 da torre A para C
	Mova disco 1 da torre B para C

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 compreensão 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 contêineres 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 mantêm 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 saída) 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:

D
C
B
A

2) Complete:

a) Entrada: 0,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,R

Saída: D,A,D,O,S

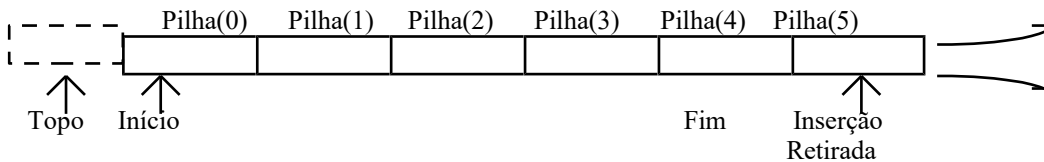
b) Entrada:

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 pilha1,pilha2,pilha3;

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);
}
```

Exercícios:

URI Online Judge | 1069

Diamantes e Areia

Por Neilor Tonin, URI  Brasil

Timelimit: 1

João está trabalhando em uma mina, tentando retirar o máximo que consegue de diamantes "<>". Ele deve excluir todas as partículas de areia "." do processo e a cada retirada de diamante, novos diamantes poderão se formar. Se ele tem como uma entrada .<...<<..>>....>....>>>., três diamantes são formados. O primeiro é retirado de <..>, resultando .<...<>....>....>>>. Em seguida o segundo diamante é retirado, restando .<.....>....>>>. O 3º diamante é daí retirado, restando no final>>>., sem possibilidade de novo diamante.

Entrada

Deve ser lido um valor inteiro N que representa a quantidade de casos de teste. Cada linha a seguir é um caso de teste que contém até 1000 caracteres, incluindo "<, >, ."

Saída

Você deve imprimir a quantidade de diamantes possíveis de serem extraídos em cada caso de entrada.

Exemplo de Entrada	Exemplo de Saída
2 <..><..>> <<<..<.....<<<<.....>	3 1

URI Online Judge | 1068

Balanço de Parênteses I

Por Neilor Tonin, URI  Brasil

Timelimit: 1

Dada uma expressão qualquer com parênteses, indique se a quantidade de parênteses está correta ou não, sem levar em conta o restante da expressão. Por exemplo:

$a + (b * c) - 2 - a$ está correto e

$(a + b * (2 - c) - 2 + a) * 2$ está correto

enquanto

$(a * b - (2 + c)$ está incorreto

$2 * (3 - a))$ está incorreto

$) 3 + b * (2 - c) ($ está incorreto

O seja, todo parênteses que fecha deve ter um outro parênteses que abre correspondente e não pode haver parênteses que fecha sem um previo parenteses que abre e a quantidade total de parenteses que abre e fecha deve ser igual.

Entrada

Como entrada, haverá N expressões ($1 \leq N \leq 10000$), cada uma delas com até 1000 caracteres.

Saída

O arquivo de saída deverá ter a quantidade de linhas correspondente ao arquivo de entrada, cada uma delas contendo as palavras correct ou incorrect de acordo com as regras acima fornecidas.

Exemplo de Entrada

```
a+(b*c)-2-a
(a+b*(2-c)-2+a)*2
(a*b-(2+c)
2*(3-a))
)3+b*(2-c) (
```

Exemplo de Saída

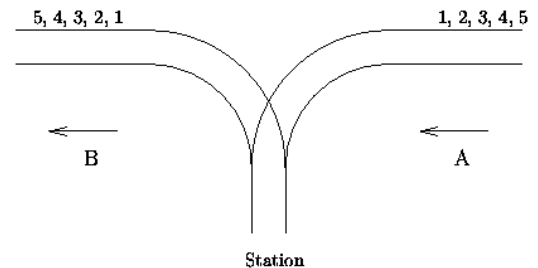
```
correct
correct
incorrect
incorrect
incorrect
```

URI Online Judge | 1062

Trilhos

Timelimit: 1

Há uma famosa estação de trem na cidade PopPush. Esta cidade fica em um país incrivelmente acidentado e a estação foi criada no último século. Infelizmente os fundos eram extremamente limitados naquela época. Foi possível construir somente uma pista. Além disso, devido a problemas de espaço, foi feita uma pista apenas até a estação (veja figura abaixo).



A tradição local é que todos os comboios que chegam vindo da direção A continuam na direção B com os vagões reorganizados, de alguma forma. Suponha que o trem que está chegando da direção A tem $N \leq 1000$ vagões numerados sempre em ordem crescente 1, 2, ..., N. O primeiro que chega é o 1 e o último que chega é o N. Existe um chefe de reorganizações de trens que quer saber se é possível reorganizar os vagões para que os mesmos saiam na direção B na ordem a_1, a_2, a_n .

O chefe pode utilizar qualquer estratégia para obter a saída desejada. No caso do desenho ilustrado acima, por exemplo, basta o chefe deixar que todos os vagões entrem na estação (do 1 ao 5) e depois retirar um a um: retira o 5, retira o 4, retira o 3, retira o 2 e por último retira o 1. Desta forma, se o chefe quer saber se a saída 5, 4, 3, 2, 1 é possível em B, a resposta seria Yes. Vagão que entra na estação só pode sair para a direção B e é possível incluir quantos forem necessários para retirar o primeiro vagão desejado.

Entrada

O arquivo de entrada consiste de um bloco de linhas, cada bloco, com exceção do último, descreve um trem e talvez mais do que uma requisição de reorganização. Na primeira linha do bloco há um inteiro N descrito acima. Cada uma das próximas N linhas conterá uma permutação dos valores 1, 2, ..., N. A última linha contém apenas 0.

Saída

O arquivo de saída contém a quantidade de linhas correspondente às linhas com permutações no arquivo de entrada. Cada linha de saída deve ser Yes se for possível organizar os vagões da forma solicitada e No, caso contrário. Há também uma linha em branco após cada bloco de entrada. No exemplo abaixo, O primeiro caso de teste tem 3 permutações para 5 vagões. O ultimo zero dos testes de entrada não devem ser processados.

Exemplo de Entrada

```
5
5 4 3 2 1
1 2 3 4 5
5 4 1 2 3
0
6
1 3 2 5 4 6
0
0
```

Exemplo de Saída

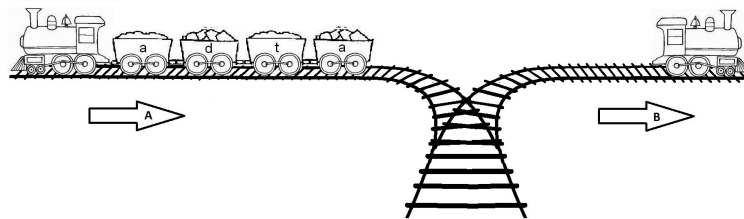
```
Yes
Yes
No
Yes
```

Trilhos Novamente... Traçando Movimentos

Por Neilor Tonin, URI  Brasil

Timelimit: 1

Você lembra daquela estação de trem da cidade PopPush? Apenas para relembrar, existe uma estação de trem em um país incrivelmente acidentado. Além disso, a estação foi construída no século passado e infelizmente os fundos eram muito limitados. Em um determinado trecho foi possível construir apenas uma pista e, a solução encontrada para transportar as cargas nos dois sentidos foi construir uma estação que permitisse desconectar os vagões de uma locomotiva e conectar em outra, que iria em outro sentido.



Cada trem que chega na direção A é manobrado e seus vagões continuam na direção B, reorganizados conforme o chefe da estação deseja. Ao chegar pelo lado A, cada vagão é desconectado e vai até a estação e depois segue para a direção B, para ser conectado na segunda locomotiva. Você pode desconectar quantos trens deseja na estação, mas o vagão que entra na estação só sai pelo lado B e quando ele sai, não pode entrar novamente.

Todos vagões são identificados pelas letras minúsculas (a até z). Isto significa 26 vagões no máximo. O chefe da organização dos vagões precisa agora que você ajude a resolver para ele, através de um programa, qual a sequência de movimentos é necessária para obter a saída desejada após a entrada na estação, seguindo para a direção B. O movimento de e para a estação é descrito pelas letras I e R (Insere e Remove respectivamente). Utilizando a figura dada como exemplo, a entrada a,t,d,a para uma saída desejada d,a,t,a, resulta nos movimentos I,I,I,R,I,R,I,R,R,R

Entrada

A entrada consiste em vários casos de teste, onde cada caso de teste é composto por 3 linhas. A primeira das 3 linhas contém um número inteiro N que representa o número total de vagões. A segunda linha contém a sequência dos vagões que vêm do lado A e a Terceira linha contém a sequência que o chefe de organização deseja como saída para o lado B. A última linha de entrada contém apenas 0, indicando o fim da entrada.

Saída

O arquivo de saída contém a quantidade de linhas correspondente ao número de casos de teste de entrada. Cada linha de saída contém uma sequência de I e R conforme o exemplo. Se não for possível mostrar a saída, as operações devem ser interrompidas e a mensagem "Impossible" deve ser impressa, com um espaço após a sequência.

Exemplo de Entrada	Exemplo de Saída
4 a t d a d a t a 5 a s t a d d a t a s 0	IIIRIRRR IIIIIRRR Impossible

Problema Complementar

A2_P2 – Parentheses Balance

Você recebe uma string consistindo de parênteses do tipo (), [] e {}. Uma string é dita correta se:

(a) se for uma string vazia. (b) se A é correto, (A), [A] ou {a} são corretos

Escreva um programa que pega uma sequência de caracteres e checka se esta sequência é correta. Seu programa pode assumir que o máximo de tamanho que uma string vai ter é 128. O arquivo de entrada contém um inteiro positivo n e uma sequência de n strings de parênteses (), [] e {}, cada string em uma linha.

Exemplo de entrada	Exemplo de saída correspondente
4	Yes
([])	No
(([]()))	Yes
([(){}][()])()	Yes
([{}][()])()	

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 rastrear a expressão a partir de sua extremidade esquerda até o ponto em questão. Se a contagem de parênteses for não-negativa, ela equivale à 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 é 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, conseqüentemente, 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

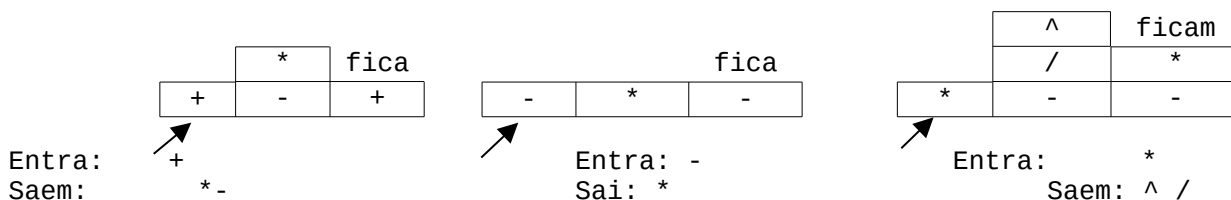
Ex: $A+B*C$ em $ABC*+$

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.



Exercícios:

Transforme as seguintes expressões para a forma posfixa:

- $A/B^c + d*e - a*c$
- $A*(B+C*A-D)*D$
- $X+(A*(B+C*A-D)*D)-2-F$

Exercícios:

URI Online Judge | 1077

Infixa para Posfixa

Por Neilor Tonin, URI  Brasil

Timelimit: 1

O Professor solicitou que você escreva um programa que converta uma expressão na forma infixa (como usualmente conhecemos) para uma expressão na forma posfixa. Como você sabe, os termos in (no meio) e pos (depois) se referem à posição dos operadores. O programa terá que lidar somente com operadores binários +, -, *, /, ^, parênteses, letras e números. Um exemplo seria uma expressão como:

$(A*B+2*C^3)/2*A$. O programa deve converter esta expressão (infixa) para a expressão posfixa: $AB*2C3^*+2/A*$

Entrada

A primeira linha da entrada contém um valor inteiro N ($N < 1000$), que indica o número de casos de teste. Cada caso de teste a seguir é uma expressão válida na forma infixa, com até 300 caracteres.

Saída

Para cada caso, apresente a expressão convertida para a forma posfixa.

Exemplo de Entrada	Exemplo de Saída
3 A*2 (A*2+c-d) / 2 (2*4/a^b) / (2*c)	A2* A2*c+d-2/ 24*ab^/2c*/

URI Online Judge | 1083

LEXSIM - Avaliador Lexico e Sintático

Por Neilor Tonin, URI  Brasil

Timelimit: 1

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:

- Operandos: todas as letras maiúsculas ou minúsculas ('a'..'z', 'A'..'Z') e números (0...9).
- Parênteses.
- Operadores: deverão ser aceitos os seguintes operadores segundo a tabela de prioridades apresentada abaixo:

Operador	Prioridade
\wedge	6
$*, /$	5
$+, -$	4
$>, <, =, \#,$	3
AND (.)	2
OR ()	1

Para facilitar a implementação, será utilizado um ponto para representar o AND (.) e o Pipe (|) para representar o OR.

Obs.: Como restrição, não será permitida a entrada de expressões com operadores unários, como por exemplo o '-' de: $4 * -2$

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, deverá ser impressa a mensagem "Lexical Error!" indicando erro léxico ou "Syntax Error!" indicando o erro de sintaxe, nesta ordem.

Exemplo de Entrada	Exemplo de Saída
(Syntax Error!
(A+	Syntax Error!
(A+B) *c	AB+c*
(A+B) *%	Lexical Error!
(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	Syntax Error!
(ab+*c) /2* (e+a)	Syntax Error!
(a+b*cc) /2* (e+a	Syntax Error!
("a+b*cc) /2* (e+a	Lexical Error!

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.

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.

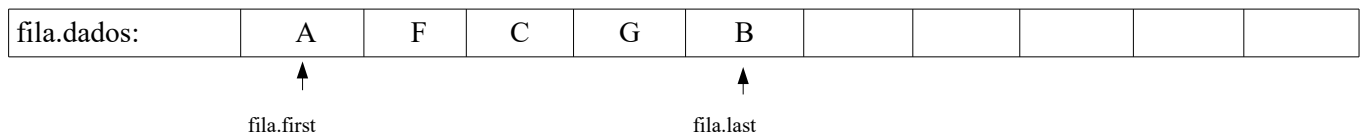
Filas são mais difíceis de implementar do que pilhas, porque as ações ocorrem em ambas as extremidades. A implementação mais simples usa um vetor, inserindo novos elementos em uma extremidade e retirando de outra e depois de uma retirada movendo todos os elementos uma posição para a esquerda. Bem, pode-se fazer melhor utilizando os índices **first** (primeiro) e **last** (último) 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 {
    char dados[SIZE+1];
    int first;
    int last;
    int count;
} queue;
```

```
queue fila;
```

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
}
```

Implementação de fila única (esquerda) e múltiplas filas (direita)

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

using namespace std;

typedef struct {
    int q[SIZE+1]; /* body of queue */
    int first, last, count; /* prim, ult, quantia */
} queue;

queue q;

void inicializa(void){
    q.first = 0;
    q.last = SIZE-1;
    q.count = 0;
}

void enfileira( int x){
    if (q.count >= SIZE)
        cout << "Overflow ao inserir: " << x << endl;
    else {
        q.last = (q.last+1) % SIZE;
        q.q[ q.last ] = x;
        q.count++;
    }
}

int desenfileira(void){
    int x;
    if (q.count <= 0)
        cout << "Underflow na fila!!" << endl;
    else {
        x = q.q[ q.first ];
        q.first = (q.first+1) % SIZE;
        q.count--;
    }
    return(x);
}

void print_queue(void){
    int i,j;
    i=q.first;

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

int main(void){
    inicializa();
    cout <<q.first<<" "<<q.last<<" "<<q.count<<endl;
    enfileira(12);
    enfileira(111);
    enfileira(34);
    cout <<q.first<<" "<<q.last<<" "<<q.count<<endl;
    print_queue();
    desenfileira();
    print_queue();

    return(0);
}
```

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

using namespace std;

typedef struct {
    int q[SIZE+1]; /* body of queue */
    int first, last, count; /* prim, ult, quantia */
} 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 << "Overflow 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 << "Underflow na fila!!" << 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=q->first,j;
    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,*q2;
    init_queue(q); enqueue(q,12); enqueue(q,32);
    print_queue(q);
    dequeue(q); enqueue(q2,2);
    print_queue(q);
    dequeue(q); print_queue(q2);
    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 naipes 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;

```

C++ Queues

back returns a reference to last element of a queue

empty true if the queue has no elements

front returns a reference to the first element of a queue

pop removes the first element of a queue

push adds an element to the end of the queue

size returns the number of items in the queue

Exemplo 1:

```
random_init_decks (&a,&b);
wait (&a,&b);
return (0);
}
queue <int> cartas; //
cartas.push (); //
cartas.size (); //
cartas.front (); // acessa sem retirar
```

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/>)

URI Online Judge | 1110

Jogando Cartas Fora

Folclore, adaptado por Piotr Rudnicki

Timelimit: 1

```
#include <queue>
#include <iostream>

using namespace std;

int main () {
    queue <string> waiting_line;
    string s;
    while ( waiting_line.size() < 3 ) {
        // servindo à fila, por favor, seu nome: ";
        getline ( cin, s );
        waiting_line.push(s);
    }
    while ( waiting_line.empty() ) {}
    cout << "Servindo: " << waiting_line.front() << endl;
    waiting_line.pop();
    return 0;
}
```

Dada uma pilha de n cartas enumeradas de 1 até n com a carta 1 no topo e a carta n na base. A seguinte operação é realizada enquanto tiver 2 ou mais cartas na pilha.

Jogue fora a carta do topo e mova a próxima carta (a que ficou no topo) para a base da pilha.

Sua tarefa é encontrar a sequência de cartas descartadas e a última carta remanescente.

Cada linha de entrada (com exceção da última) contém um número $n \leq 50$. A última linha contém 0 e não deve ser processada. Cada número de entrada produz duas linhas de saída. A primeira linha apresenta a sequência de cartas descartadas e a segunda linha apresenta a carta remanescente.

Entrada

A entrada consiste em um número indeterminado de linhas contendo cada uma um valor de 1 até 50. A última linha contém o valor 0.

Saída

Para cada caso de teste, imprima duas linhas. A primeira linha apresenta a sequência de cartas descartadas, cada uma delas separadas por uma vírgula e um espaço. A segunda linha apresenta o número da carta que restou. Nenhuma linha tem espaços extras no início ou no final. Veja exemplo para conferir o formato esperado.

Exemplo de Entrada Exemplo de Saída

7	Discarded cards: 1, 3, 5, 7, 4, 2
19	Remaining card: 6
10	Discarded cards: 1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 4, 8, 12, 16, 2, 10, 18, 14
6	Remaining card: 6
0	Discarded cards: 1, 3, 5, 7, 9, 2, 6, 10, 8
	Remaining card: 4
	Discarded cards: 1, 3, 5, 2, 6
	Remaining card: 4

Adaptação, entradas e saídas by Neilor Tonin

Remaining card: 4

Outros exemplos (para melhor compreensão)



```
1
Discarded cards:
Remaining card: 1
```

```
2
Discarded cards: 1
Remaining card: 2
```

```
3
Discarded cards: 1, 3
Remaining card: 2
```

```
6
Discarded cards: 1, 3, 5, 2, 6
Remaining card: 4
```

```
7
Discarded cards: 1, 3, 5, 7, 4, 2
Remaining card: 6
```

```
8
Discarded cards: 1, 3, 5, 7, 2, 6, 4
Remaining card: 8
```

URI Online Judge | 1119

A Fila de Desempregados

Autor Desconhecido

Timelimit: 1

Em uma série tentativa de reduzir a fila de desempregados, o novo Partido Nacional Trabalhista dos Rinocerontes Verdes decidiu uma estratégia pública. Todos os dias, todos os candidatos desempregados serão colocados em um grande círculo, voltados para dentro. Alguém é escolhido arbitrariamente como número 1, e os outros são numerados no sentido horário até N (os quais estarão à esquerda do 1°). Partindo do 1° e movendo-se no sentido anti-horário, um contador oficial do laboratório conta k posições e retira um candidato, enquanto outro oficial começa a partir de N e se move no sentido horário, contando m posições e retirando outro candidato. Os dois que são escolhidos são então enviados como estagiários para a reciclagem e se ambos os funcionários escolherem a mesma pessoa, ela (ele) é enviado para se tornar um político. Cada funcionário, em seguida, começa a contar novamente com a pessoa próxima disponível e o processo continua até que não reste ninguém. Note-se que as duas vítimas (desculpe, estagiários) deixam o anel ao mesmo tempo, por isso é possível que um funcionário conte a pessoa já selecionado pelo outro funcionário.

Entrada

Escreva um programa que leia sucessivamente três números (N, k e m; $k, m > 0$, $0 < N < 20$) e determina a ordem no qual os candidatos são retirados para treinamento. Cada conjunto de três números estará em uma linha distinta e o final da entrada de dados é sinalizado por três zeros (0 0 0).

Saída

Para cada conjunto de três números de entrada, imprima uma linha de números especificando a ordem na qual as pessoas são escolhidas. Cada número pode ter até 3 dígitos. Liste o par escolhido partindo da

pessoa escolhida pelo contador do sentido horário. Os pares sucessivos são separados por vírgula (mas não deverá haver vírgula após o último escolhido).

Exemplo de Entrada	Exemplo de Saída
10 4 3 0 0 0	<input type="text"/> 4 <input type="text"/> 8, <input type="text"/> 9 <input type="text"/> 5, <input type="text"/> 3 <input type="text"/> 1, <input type="text"/> 2 <input type="text"/> 6, <input type="text"/> 10, <input type="text"/> 7 onde <input type="text"/> representa um espaço.

4. Classificação (ordenação) de dados

Importante:

- <http://www.sorting-algorithms.com/>
- Métodos de ordenação com dança Húngara

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;

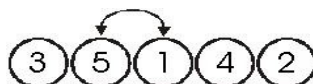
4.1 Bubblesort

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 :



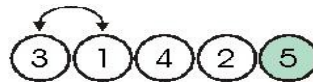
Assim verificamos que neste caso os dois primeiros elementos estão desordenados entre si, logo devemos trocá-los de posição. E assim continuamos com as comparações dos elementos subsequentes :



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 desde o 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.

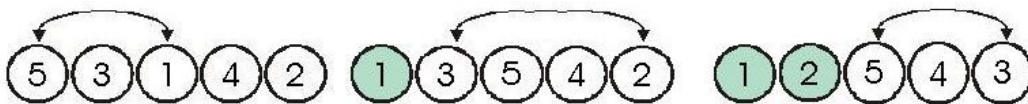


```
long int aux; // Variável auxiliar para fazer a troca, caso necessário
for ( long int i=0; i <= tam-2; i++ ){
    for ( long int j=0; j<= tam-2-i; j++ ) {
        if ( Array[j] > Array[j+1] ) { // Caso o elemento de uma posição menor
            aux = Array[j];           // for maior que um elemento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

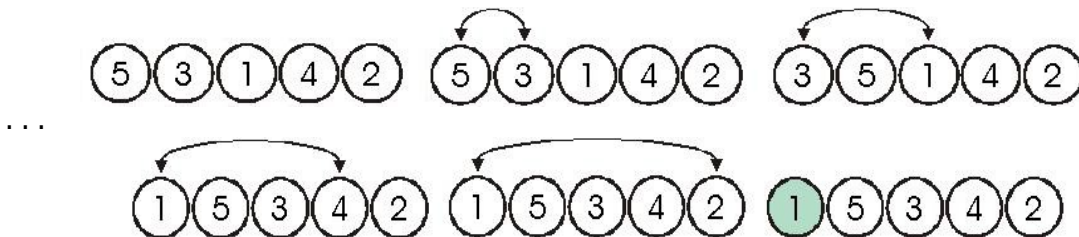
4.2 Seleção Direta

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 início, 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.

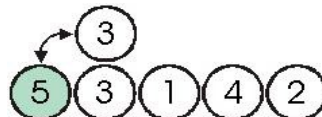
Consideremos inicialmente um arranjo qualquer desordenado:



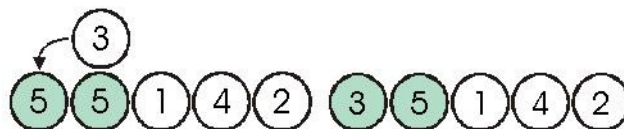
Inicialmente consideramos o primeiro elemento do arranjo como se ele estivesse ordenado, ele será considerado o sub-arranjo ordenado inicial :



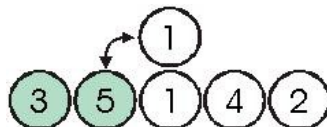
Agora o elemento imediatamente superior ao o sub-arranjo ordenado, no o exemplo o número 3, deve se copiado para uma variável auxiliar qualquer. Após copiá-lo, devemos percorrer o sub-arranjo a partir do último elemento para o primeiro. Assim poderemos encontrar a posição correta da nossa variável auxiliar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



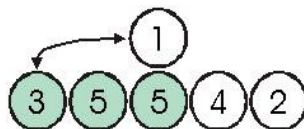
No caso verificamos que a variável auxiliar é menor que o último elemento do o sub-arranjo ordenado (o o sub-arranjo só possui por enquanto um elemento, o número 5). O número 5 deve então ser copiado uma posição para a direita para que a variável auxiliar com o número 3, seja colocada em sua posição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



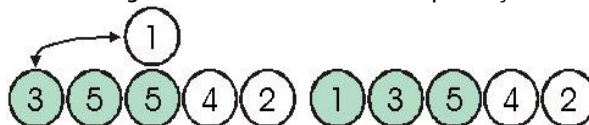
Verifique que o sub-arranjo ordenado possui agora dois elementos. Vamos repetir o processo anterior para que se continue a ordenação. Copiamos então mais uma vez o elemento imediatamente superior ao o sub-arranjo ordenado para uma variável auxiliar. Logo em seguida vamos comparando nossa variável auxiliar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



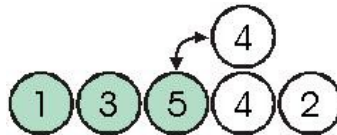
Neste caso verificamos que a nossa variável auxiliar é menor que o último elemento do sub-arranjo. Assim, copiamos este elemento para a direita e continuamos com nossas comparações :



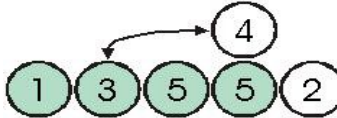
Aqui, mais uma vez a nossa variável auxiliar é menor que o elemento do sub-arranjo que estamos comparando. Por isso ele deve ser copiado para a direita, abrindo espaço para que a variável auxiliar seja colocada em sua posição correta :



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 :

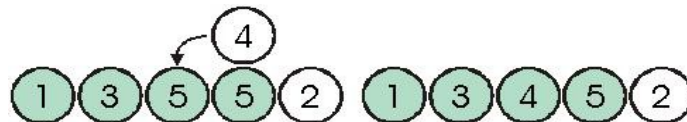


Veja que nossa variável auxiliar é menor que o elemento que está sendo comparado no o sub-arranjo. Então ele deve ser copiado para a direita para que continuemos com nossas comparações :

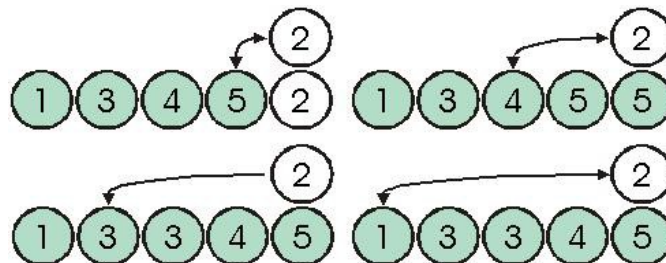


Veja que aqui ocorre o inverso. A variável auxiliar é maior que o elemento do sub-arranjo que estamos comparando. Isto significa que já encontramos a posição correta para a nossa variável auxiliar.

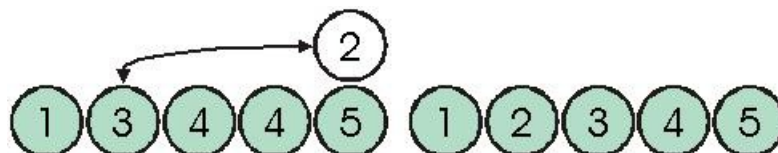
Basta agora copiá-la para sua posição correta, ou seja, copiá-la para o elemento imediatamente superior ao elemento que estava sendo comparado, veja :



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 $h = n / 1,3$ $\text{Salto} = \text{Int}(n / 1,3) = 3$ 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
2	3	24 25 30 28 26	2		não troca
	4	24 25 30 28 26	2		não troca
	5	24 25 30 28 26	2		troca
3	6	24 25 26 28 30	1		não troca
	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 ordenação 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	3 1 4 2 5 5 7 6 8 9 11
5:	3 1 4 2 5 5 7 6 8 9 11
5 11 4 2 8 7 1 6 3 9 5	3 1 4 2 5 5 7 6 8 9 11
5 11 4 2 8 5 1 6 3 9 7	3 1 4 2 5 5 7 6 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1:
5 1 4 2 8 5 11 6 3 9 7	1 3 4 2 5 5 7 6 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1 3 4 2 5 5 7 6 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1 2 3 4 5 5 7 6 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1 2 3 4 5 5 7 6 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1 2 3 4 5 5 7 6 8 9 11
2:	1 2 3 4 5 5 6 7 8 9 11
5 1 4 2 8 5 11 6 3 9 7	1 2 3 4 5 5 7 6 8 9 11
4 1 5 2 8 5 11 6 3 9 7	1 2 3 4 5 5 7 6 8 9 11
4 1 5 2 8 5 11 6 3 9 7	1 2 3 4 5 5 6 7 8 9 11
4 1 5 2 8 5 11 6 3 9 7	1 2 3 4 5 5 6 7 8 9 11
3 1 4 2 5 5 8 6 11 9 7	1 2 3 4 5 5 6 7 8 9 11
3 1 4 2 5 5 7 6 8 9 11	1 2 3 4 5 5 6 7 8 9 11

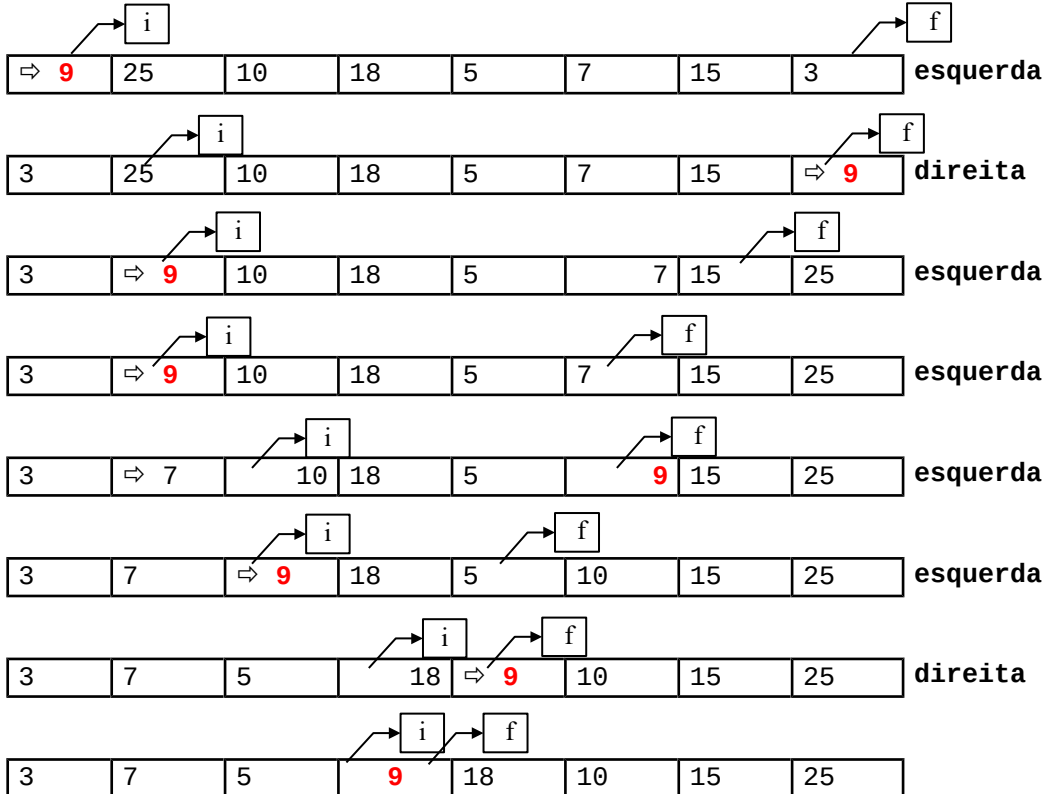
9	25	10	18	5	7	15	3
---	----	----	----	---	---	----	---

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 sequência 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:

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

--	--	--

Exercícios (problemas) utilizando Sort:

a) Mergesort

É importante ressaltar aqui que de forma alguma é importante decorar os muitos tipos de ordenação existentes, mas sim saber aplicá-los. Muitos, mas muitos problemas envolvem a contagem de trocas que devem ser feitas e ao utilizar o bubble, conseguimos apenas TLE. Ou seja o bolha é **muito lento!!!**

A melhor saída é utilizar o **Merge Sort**. Bem codificado e otimizado, sua performance é $n \log n$, a mesma do QuickSort, com a vantagem de se poder fazer a contagem de trocas facilmente.

Segue então o algoritmo foco de nosso trabalho:

/* Merge sort otimizado por NAT. Existem várias opções de Mergesort disponível na Web.
Esta é a mais mais eficiente das 3 variações que eu testei. Eu a otimizei um pouco e ainda
Há uma otimização para ganhar 0.017 segundos. Data: 06/05/2009 Neilor Tonin

```
*/
#include <iostream>
#include <cstdio>

using namespace std;

void merge(int numbers[], int temp[], int left, int mid, int right) {
    int i, left_end, num_elements, tmp_pos;
    left_end = (mid - 1);
    tmp_pos = left;
    num_elements = (right - left + 1);

    while ((left <= left_end) && (mid <= right)) {
        if (numbers[left] <= numbers[mid]) {
            temp[tmp_pos] = numbers[left];
            tmp_pos += 1;
            left += 1;
        } else {
            temp[tmp_pos] = numbers[mid];
            tmp_pos += 1;
            mid += 1;
        }
    }

    while (left <= left_end) {
        temp[tmp_pos] = numbers[left];
        left += 1;
        tmp_pos += 1;
    }

    while (mid <= right) {
        temp[tmp_pos] = numbers[mid];
        mid += 1;
        tmp_pos += 1;
    }

    for (i=0; i <= num_elements; i++) {
        numbers[right] = temp[right];
        right -= 1;
    }
}

void m_sort(int numbers[], int temp[], int left, int right) {
    int mid;

    if (right > left) {
        mid = (right + left) / 2;
        m_sort(numbers, temp, left, mid);
        m_sort(numbers, temp, (mid+1), right);
        merge(numbers, temp, left, (mid+1), right);
    }
}

void mergeSort(int numbers[], int temp[], int array_size) {
    m_sort(numbers, temp, 0, array_size - 1);
}
```

```
int main() {
    int num;
    int arrayOne[10000];
    int arrayTwo[10000];

    scanf ("%d", &num );
    while (num !=0) {
        for (int i=0; i < num; i++) {
            scanf("%d", &arrayOne[i]);
        }
        mergeSort(arrayOne, arrayTwo, num);

        for (int i = 0; i < num; i++) {
            cout << arrayOne[i] << " " << endl;
        }
        scanf ("%d", &num );
    }
    return 0;
}
```

Resolver:

Problema UOJ 1088 – Bolhas e Baldes

Problema UOJ 1162 – Organizador de Vagões

b) Countsort

O counsort consiste conforme o próprio nome sugere, na ideia de contar ao invés de ordenar. Consideremos um pequeno exemplo onde temos que ordenar 10 valores que estão no intervalo 0-5: 0, 3, 1, 2, 5, 2, 3, 4, 4, 1

Ao utilizar um procedimento normal de ordenação, como por exemplo o bubblesort, várias comparações entre estes elementos devem ser efetuadas. Como o algoritmo é de complexidade N^2 , cerca de 90 comparações podem ser feitas no pior caso.

No caso da utilização do **countingSort**, a forma de ordenação é muito mais simples. Deve-se utilizar um vetor para armazenar os elementos que tenha o tamanho máximo referente ao maior valor possível de entrada. Neste exemplo o vetor deveria ter 6 posições (0 a 5).

0	1	2	3	4	5

Inicia-se então zerando todas as posições do vetor. Em seguida, para cada valor inserido, adiciona-se mais 1 na posição equivalente ao número inserido. Para o exemplo fornecido, o vetor ficaria assim:

0	1	2	3	4	5

A partir daí, percorre-se todo o vetor, repetindo na saída **N** vezes o número equivalente à posição de cada elemento, segundo o valor contido em cada posição.

URI Online Judge | 1171

Frequência de Números

Adaptado por Neilor Tonin, URI  Brasil

Timelimit: 1

Neste problema sua tarefa será ler vários números e em seguida dizer quantas vezes cada número aparece na entrada de dados, ou seja, deve-se escrever cada um dos valores distintos que aparecem na entrada por ordem crescente de valor.

Entrada

A entrada contém apenas 1 caso de teste. A primeira linha de entrada contém um único inteiro **N**, que indica a quantidade de valores que serão lidos para **X** ($1 \leq X \leq 2000$) logo em seguida. Com certeza cada número não aparecerá mais do que 20 vezes na entrada de dados.

Saída

Imprima a saída de acordo com o exemplo fornecido abaixo, indicando quantas vezes cada um deles aparece na entrada por ordem crescente de valor.

Exemplo de Entrada	Exemplo de Saída
7	
8	
10	4 aparece 1 vez(es)
8	8 aparece 2 vez(es)
260	10 aparece 3 vez(es)
4	260 aparece 1 vez(es)
10	
10	

Estiagem

Por Neilor Tonin, URI  Brasil

Timelimit: 2

Devido às constantes estiagens que aconteceram nos últimos tempos em algumas regiões do Brasil, o governo federal criou um órgão para a avaliação do consumo destas regiões com finalidade de verificar o comportamento da população na época de racionamento. Este órgão responsável pegará algumas cidades (por amostragem) e verificará como está sendo o consumo de cada uma das pessoas da cidade e o consumo médio de cada cidade por habitante.

Entrada

A entrada contém vários casos de teste. A primeira linha de cada caso de teste contém um inteiro N ($1 \leq N \leq 1 \cdot 10^6$), indicando a quantidade de imóveis. As N linhas contém um par de valores X ($1 \leq X \leq 10$) e Y ($1 \leq Y \leq 200$), indicando a quantidade de moradores de cada imóvel e o respectivo consumo total de cada imóvel (em m^3). Com certeza, nenhuma residência consome mais do que 200 m^3 por mês. O final da entrada é representado pelo número zero.

Saída

Para cada entrada, deve-se apresentar a mensagem “Cidade# n:”, onde n é o número da cidade seguindo a sequência (1, 2, 3, ...) e em seguida deve-se listar, por ordem ascendente de consumo, a quantidade de pessoas seguido de um hífen e o consumo destas pessoas, arredondando o valor para baixo. Na terceira linha da saída deve-se mostrar o consumo médio por pessoa da cidade, com 2 casas decimais sem arredondamento, considerando o consumo real total. Imprimir uma linha em branco entre dois casos de teste consecutivos. No fim da saída não deve haver uma linha em branco.

Exemplo de Entrada

```
3
3 22
2 11
3 39
5
1 25
2 20
3 31
2 40
6 70
0
```

XIII Maratona de Programação IME-USP, 2009.

Exemplo de Saída

```
Cidade# 1:
2-5 3-7 3-13
Consumo medio: 9.00 m3.

Cidade# 2:
5-10 6-11 2-20 1-25
Consumo medio: 13.28 m3.
```

Altura

Por Neilor Tonin, URI  Brazil

Timelimit: 1

Cheio de boas ideias, agora o governo brasileiro resolveu criar a "bolsa altura". Desta forma, você foi incumbido de fazer o levantamento da altura da população de várias cidades e ordenar esta população por ordem crescente de altura. Você sabe que as cidades as quais terá que fazer isso tem menos de 3 milhões de habitantes e que ninguém, segundo o IBGE, tem mais do que 230 cm de altura nestas cidades.

Entrada

A entrada contém vários casos de teste. A primeira linha de entrada contém um inteiro NC ($NC < 100$) que indica a quantidade de casos de teste, ou seja de cidades. Para cada caso de teste, a primeira linha conterá um inteiro N ($1 < N \leq 3000000$), indicando a quantidade de pessoas da cidade. A próxima linha irá conter a altura de cada uma destas pessoas, em centímetros, representado pela letra h ($20 \leq h \leq 230$) e separados por um espaço em branco.

Saída

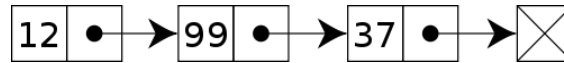
Para cada caso de teste de entrada, imprima uma linha contendo os valores das alturas de todos os moradores da cidade (em cm), por ordem crescente de altura, separados por um espaço em branco.

Obs.: O arquivo de entrada é bastante grande, portanto, utilize um método rápido para leitura / escrita.

Exemplo de Entrada	Exemplo de Saída
6	31 35 37 37 37 57 61 65 72 76
10	21 22 26 45 49 51 51 55 64 78 185 186
65 31 37 37 72 76 61 35 57 37	20 58 64 67 81 93 112 180 203 225
12	32 68 169 180 189 214 220 228
45 186 185 55 51 51 22 78 64 26 49 21	41 55 67 112 133 166
10	38 39 55 120
20 93 203 67 64 225 112 81 58 180	
8	
169 189 220 228 68 32 214 180	
6	
133 55 67 166 112 41	
4	
39 38 120 55	

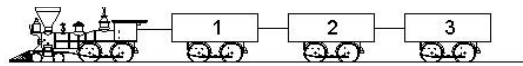
Listas ligadas ou encadeadas

Em Ciência da computação, lista ligada ou **linked list** é uma estrutura de dados que consiste de uma sequência de registros tal que cada registro contém um ou mais campos de dados e uma referência ou link para o próximo registro. (i.e., um *link*) para o próximo registro na sequência (**fonte wikipedia**)

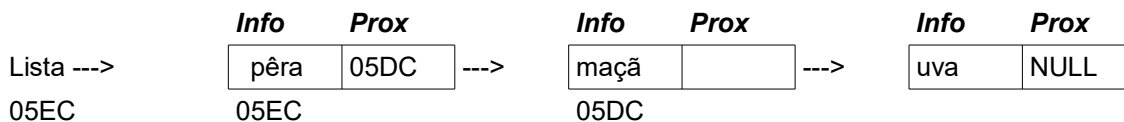


A desvantagem da alocação estática é clara: quando se utiliza uma estrutura linear, uma quantidade de bytes fica alocada mesmo sem ser utilizada. Os dois vetores (INFO e NEXT) são utilizados apenas em parte e a complexidade aumenta exponencialmente quando se deseja implementar 2 ou mais listas sobre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

A estrutura da lista ligada dinâmica é muito simples. Existem somente 2 campos (INFO, NEXT ou PROX), sendo que o campo next é um ponteiro para o próximo elemento da lista. 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. Vejamos o exemplo de um trem. A locomotiva é ligada ao primeiro vagão e cada vagão é ligado ao próximo.



Podemos visualizar 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 número inteiro, com a inserção do número 1:

Passo 1: a inserção de um elemento:

```
#include <iostream>

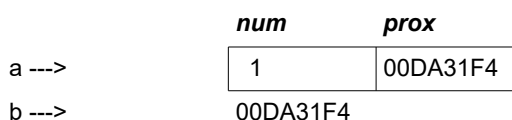
using namespace std;

struct Pessoa {
    int num;
    Pessoa *prox;
    Pessoa(int x, Pessoa *p) {
        cout << "num= " << x << " prox= " << p << endl;
        num = x;
        prox = p;
    }
};

int main(void) {
    /*****
    'a' recebe primeira pessoa criada e em seguida eh linkado
    para si mesmo. Depois 'b' eh criado apontado para 'a'
    *****/
    Pessoa *a = new Pessoa(1, 0);

    a->prox = a;
    Pessoa *b = a;
    cout << "a=" << a << " a->prox: " << a->prox << endl;
    cout << "b=" << b << " b->prox: " << b->prox << endl;
    return 0;
}
```

```
E:\aulas\Algoritmos_ED_2\listadin2
num= 1 prox= 00000000
a=00DA31F4 a->prox: 00DA31F4
b=00DA31F4 b->prox: 00DA31F4
```



Passo 2: a inserção de mais elementos:

```
#include <iostream>

using namespace std;

struct Pessoa {
    int num;
    Pessoa *prox;
    Pessoa(int x, Pessoa *p) {
        cout << "num= " << x << " prox= " << p << endl;
        num = x;
        prox = p;
    }
};

int main(void) {
    /*****
    'a' recebe primeira pessoa criada e em seguida eh linkado
    para si mesmo. Depois 'b' eh criado apontado para 'a'
    *****/
    Pessoa *a = new Pessoa(1, 0);

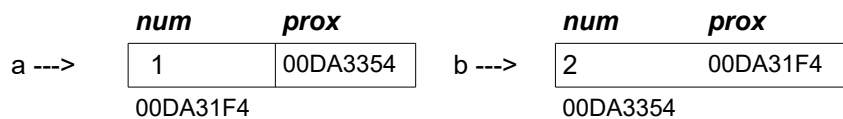
    a->prox = a;
    Pessoa *b = a;
    cout << "a=" << a << " a->prox: " << a->prox << endl;
    cout << "b=" << b << " b->prox: " << b->prox << endl<<endl;

    /// Inserindo mais um elemento
    b = (b->prox = new Pessoa(2, a));

    cout << "a=" << a << " a->prox: " << a->prox << endl;
    cout << "b=" << b << " b->prox: " << b->prox << endl;
    return 0;
}
```

```
num= 1 prox= 00000000
a=00DA31F4 a->prox: 00DA31F4
b=00DA31F4 b->prox: 00DA31F4

num= 2 prox= 00DA31F4
a=00DA31F4 a->prox: 00DA3354
b=00DA3354 b->prox: 00DA31F4
```



Passo 3: Altere o programa fonte para inserir ao todo 5 elementos (do número 1 até o número 5). Para tanto, faça a leitura de um N que é a quantidade de elementos.

```
Digite a quantidade de pessoas:5
num= 1 prox= 00000000
a=00DA3344 a->prox: 00DA3344
b=00DA3344 b->prox: 00DA3344

num= 2 prox= 00DA3344
a=00DA3344 a->prox: 00DA3368
b=00DA3368 b->prox: 00DA3344

num= 3 prox= 00DA3344
a=00DA3344 a->prox: 00DA3368
b=00DA3378 b->prox: 00DA3344

num= 4 prox= 00DA3344
a=00DA3344 a->prox: 00DA3368
b=00DA3388 b->prox: 00DA3344

num= 5 prox= 00DA3344
a=00DA3344 a->prox: 00DA3368
b=00DA3398 b->prox: 00DA3344
```

Passo 4 : Eliminação dos elementos, 1 a 1, iniciando do primeiro

```
#include <iostream>

using namespace std;

struct Pessoa {
    int num;
    Pessoa *prox;
    Pessoa(int x, Pessoa *p) {
        num = x;
        prox = p;
    }
};

int main(void) {
    /*****
    'a' recebe primeira pessoa criada e em seguida eh linkado
    para si mesmo. Depois 'b' eh criado apontado para 'a'
    *****/
    int N;
    cout << "Digite a quantidade de pessoas:";
    cin >> N;

    Pessoa *a = new Pessoa(1, 0);

    a->prox = a;
    Pessoa *b = a;
    cout << "a=" << a << " a->prox: " << a->prox << endl;
    cout << "b=" << b << " b->prox: " << b->prox << endl << endl;

    /*****
    Aloca na memoria todas as pessoas e 'b' recebe a ultima
    (enesima pessoa) pois a contagem comeca pela primeira ('a')
    *****/
    for (int i = 2; i <= N; i++) {
        b = (b->prox = new Pessoa(i, a));
        cout << "inseriu " << i << " no: " << b << endl;
    }

    /*****
    Eliminando os elementos. Quanto b=b->prox significa q tem 1 elemento apenas
    *****/
    cout << "\nEliminando:\n";
    while (b != b->prox) {
        a = b->prox;
        b->prox = a->prox;
        cout << "Eliminando valor=" << a->num << " proximo: " << a->prox << endl;
        delete a; // 'a' é eliminado //
    }

    cout << "restou: " << b->num << endl;
    Return 0;
}
```

```
Digite a quantidade de pessoas:5
a=00DA3344 a->prox: 00DA3344
b=00DA3344 b->prox: 00DA3344
```

```
inseriu 2 no: 00DA3368
inseriu 3 no: 00DA3378
inseriu 4 no: 00DA3388
inseriu 5 no: 00DA3398
```

```
Eliminando:
Eliminando valor=1 proximo: 00DA3368
Eliminando valor=2 proximo: 00DA3378
Eliminando valor=3 proximo: 00DA3388
Eliminando valor=4 proximo: 00DA3398
restou: 5
```

Passo 5: Leia um valor M que é o tamanho do pulo. Elimine de acordo com este pulo. Por exemplo, para uma entrada N=5 e M=3, a saída deve ser:

```

Eliminando:
Eliminando valor=3 proximo: 00DA3388
Eliminando valor=1 proximo: 00DA3368
Eliminando valor=5 proximo: 00DA3368
Eliminando valor=2 proximo: 00DA3388
restou: 4
    
```

	Linked list	Array	Dynamic array
Indexing	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Insertion/deletion at end	$\Theta(1)$	N/A	$\Theta(1)$
Insertion/deletion in middle	$\Theta(1)$	N/A	$\Theta(n)$
Wasted space (average)	$\Theta(n)$	0	$\Theta(n)$

Referência para aprender mais:

http://en.wikipedia.org/wiki/Linked_list

Aplicação:

Uma das aplicações de listas encadeadas é para resolução de problemas de simulação, como por exemplo o problema de Josephus (Josephus problem)

...A good example that highlights the pros and cons of using dynamic arrays vs. linked lists is by implementing a program that resolves the **Josephus problem**. (http://en.wikipedia.org/wiki/Linked_list)

Recapitulando, Josephus foi um historiador judeu do primeiro século. A lenda conta que ele e 40 de seus homens estavam presos em uma caverna pelos Romanos. Eles decidiram suicidar-se à ser capturados. Formaram um círculo e começaram a se matar, da 3ª à 3ª pessoa em ordem da fila. Como Josephus não queria morrer, ele foi capaz de escolher o melhor lugar, afim de sobreviver, e juntar-se os Romanos que o capturaram.

O Problema

Na prática o problema teria N pessoas em um círculo, iria ser percorrido $M - 1$ pessoas e o M iria ser morto, assim por diante, até sobrar uma única pessoa.

A Solução

O interessante é que o algoritmo para solução genérica é bem simples em C++. Usando **listas ligadas(linked lists)** é simples, fácil de entender e super rápido a execução do programa.

a) Quando salto ou o sentido da fila ou lista pode variar, usa-se listas ligadas e duplamente ligadas.

b) Quando o salto é fixo, utiliza-se recursividade:
http://en.wikipedia.org/wiki/Josephus_problem

Passo 6:

```

a = b->prox;
b->prox = a->prox;
cout << "Eliminando" << a->num << endl;
delete a;

/*****
Eliminando os elementos. Quanto b=b->prox significa q tem 1 elemento apenas
*****/
cout << "\nEliminando:\n";
while (b != b->prox) {
    ...
    
```

Pilha dinâmica

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.

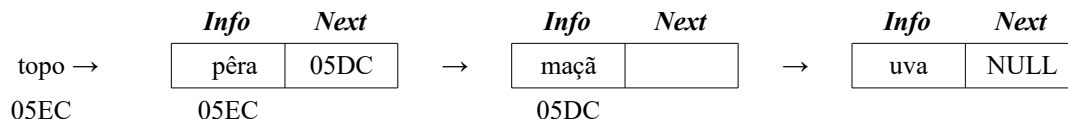
Com relação à manipulação dos dados, os algoritmos de manipulação das pilhas são os mais simples, se comparados com as listas vistas anteriormente.

```
typedef 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.

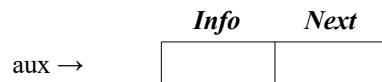
```
topo = aux = NULL;
```

Considerando uma pilha existente com alguns nodos:



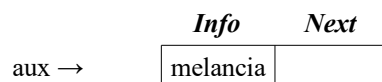
a) O primeiro passo é obter um nodo para inserí-lo na lista:

```
aux = new pilha;
```



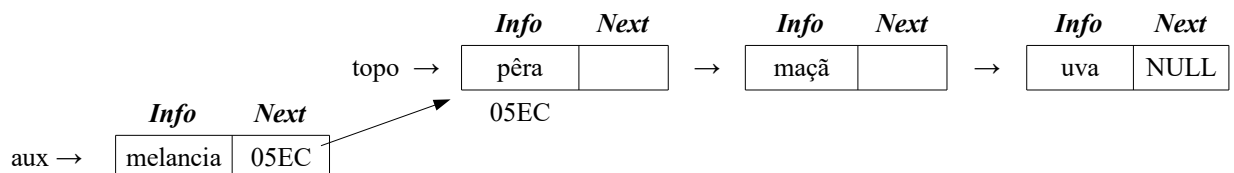
b) O próximo passo é inserir o valor desejado no nodo recém-alocado.

```
cin >> info;
```



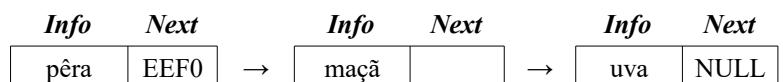
c) Depois deve-se fazer o next do nodo inserido apontar para o início da lista.

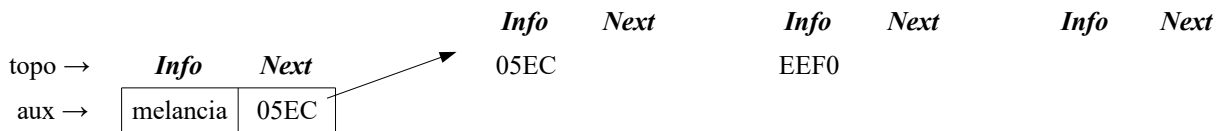
```
aux->next = topo;
```



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;
```

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 ;
```

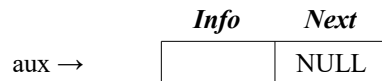
```
Fila *frente, *re, *aux;
```

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

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

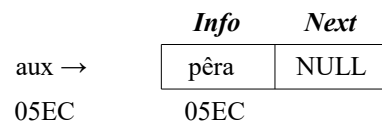
a) O primeiro passo é obter um nodo para inserí-lo na lista.

```
aux = new fila;
```



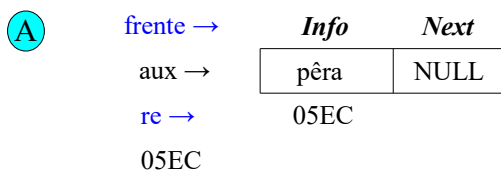
b) Após insere-se o valor desejado no nodo recém-alocado, atualizando o seu campo **Next** para **NULL**:

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



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
    frente = aux;
} else {                //A fila já contém alguns elementos
    re->next = aux;
}
re = aux;
```



(B)

