Universidade Federal de Pernambuco
Cin – Centro de Informática
Prof: Adriano Sarmento
Data: 29/05/2012
Data de Entrega: 11/06/2012

Considerações:

É proibido usar a biblioteca conio.h;
Leia a lista toda o quanto antes, para evitar más interpretações e muitas dúvidas em cima da hora;
Envie uma prévia da lista, pelo menos um dia antes da data final de entrega, para o caso de acontecer algum imprevisto;
A lista é para ser feita individualmente. Qualquer tentativa de cópia acarretará o zeramento da lista de todos os envolvidos;
Em caso de dúvida, envie email para **listaip@googlegroups.com**;
Atenção para a liberação de memória no final dos programas. Será cobrado que o espaço alocado no decorrer do programa seja totalmente liberado no final do mesmo.

Sexta Lista – IP/Eng. Da Computação – 2012.1

Questão 1) Torre de Hanói

A Torre de Hanói é um "quebra-cabeça" que consiste em uma base contendo três pinos, em um dos quais são dispostos alguns discos uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo. O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação. O número de discos pode variar sendo que o mais simples contém apenas três.
Sabendo disso, crie um software que simule uma torre de Hanói. Cada pino da torre será representada por uma pilha de discos, onde cada disco armazena um inteiro, recebido pelo usuário.
A cada movimento, será impressa a configuração atual das 3 torres.

A entrada consistirá de:

```
N                  //Número de discos
X1                 //Tamanho do menor disco
X2                 //Tamanho do segundo menor disco
.
.
.
XN                 //Tamanho do maior disco
//Fim da entrada
Saída:

Estado inicial:
Torre A: XN XN-1 ... X1 //Primeira torre
Torre B:                //Vazia no estado inicial
Torre C:                //Vazia no estado inicial

Movimento 1:
Torre A: XN XN-1 ... X2 //Configuração das torres
Torre B: X1             //Após o primeiro movimento
Torre C:

.
.
```

.

```
Movimento Z:

Torre A:                //Vazia no estado final
Torre B:                //Vazia no estado final
Torre C: XN XN-1 ... X1 //Completamente movida
```

Questão 2) Estou com Sort

Listas encadeadas permitem que termos sejam adicionados ou retirados em qualquer parte da sua extensão, no entanto trocar os nós de posição é um tanto quanto complicado. Requer a mudança de até 4 referencias, o anterior ao primeiro nó, o anterior ao segundo nó e os próprios dois nós que precisam ser trocados de ordem. Normalmente se controla a entrada dos termos para que já entrem na ordem devida, no entanto, programadores descuidados não o fizeram. Você terá que corrigir o erro deles.

Receba um arquivo de entrada .in contendo em cada linha um quantidade qualquer de números (unsigned int) separados por espaços, aloque os termos em uma lista encadeada na mesma ordem e depois organize-os em ordem crescente de valor. Não troque o conteúdo de cada nó, e mantenha cada nó no mesmo endereço da alocação inicial, você deve unicamente mexer nas suas referencias.

Uma solução possível é a criação de uma função swap(troca de termo) com a assinatura próxima a esta:

List* swap(List* list, Node* one, Node* two)

A ser chamada dentro de outra função que organizará os termos, desse jeito será mais fácil adaptar a implementação já conhecida por vocês de ordenação de vetor a lista encadeada.

Para esta função quando for preciso trocar o 2° termo com o 4°, por exemplo, faça agora o 1° termo apontar o 4°, o 4° apontar o 3°, o 3° apontar o 2° e o 2° apontar o 5° termo. E assim para cada caso. Preste atenção quando for mudar o 1° termo, nesse caso, o próprio ponteiro da lista mudará seu valor para um novo primeiro termo a ser apontado.

Qualquer algoritmo de ordenação de container (vetor, lista, set, arvore) é bem vindo: selection sort, quick sort, buble sort ou qualquer outro. Escolham o que preferirem.

Não é obrigatório o uso de swap, qualquer solução que altere unicamente as referencias de próximo termo é aceitável.

Ao fim, leia a lista agora organizada e a coloque em uma linha do arquivo .out. Faça o mesmo na próxima linha do arquivo de entrada até o seu fim. Entre cada linha, desaloque os nós desnecessários ou aloque novos termos necessários, sempre mantendo o mínimo necessário de nós alocados por linha a ser organizada.

```
Exemplo de In:
3 5 6 0 1 8
2 0 9
10 999 22 10192 99 19282 2928328 29292 373 32 13 54 13 23 45 32


Out :
0 1 3 5 6 8
0 2 9
10 13 13 22 23 32 32 45 54 99 373 999 10192 19282 29292 2928328
```

OBS: O algoritmo distribuído na internet pelo nome de "EstouComSort" esta proibido.

Questão 3) Fórmula 1

Um programador fã de Formula 1, após muitas manhãs de domingo sem poder checar o resultado das corridas no seu OS de linha de comando preferido, resolveu criar um programa para poder ficar a par das corridas.
Porém, no meio do desenvolvimento do programa, ele percebeu que, por ser muito atarefado, não conseguiria tempo para completar o programa.
Então, o programador pediu aos alunos de IP de EC-CIn que completassem seu programa.

Até agora, o programa recebe da internet eventos das corridas, e manda através de um arquivo texto comandos para a segunda parte a ser desenvolvida pelos alunos.

Existem 4 comandos do arquivo que foram implementados, cada um em uma linha:

"X passou Y" onde X e Y são corredores, X antes do comando estando logo atrás de Y, e depois do comando X estará a frente de Y.
"Volta" o corredor mais proximo da largada completa mais uma volta.
"Lider" escreve na saída o nome do corredor que lidera a corrida
"X Quebrou" O carro do corredor X quebrou

Os corredores, segundo especificação do programador, deverão ser estruturas com:

- Nome(20 caracteres)
- Numero de voltas restantes a ser completadas (inteiro)

Postas em uma lista encadeada circular.

Antes de começar os comandos, o arquivo de entrada conterá:

```
N                 //Um inteiro, que será o numero de corredores.
V                 //Um inteiro, que será o numero de voltas da corrida.
NOME1
NOME2
.
.
.
NOMEN             //E os nomes dos corredores, linha por linha
COMANDO1
COMANDO2
COMANDO3                          //Comandos
.
.
.
ULTIMOCOMANDO
//FIM DE ARQUIVO
```

Ao quebrar, ou passar pela largada e completar o numero de voltas total, o corredor sai da corrida (e da lista encadeada circular)
O programa encerra quando todos os corredores completarem a corrida ou quebrarem.

No fim do programa, devem ser impressos no arquivo de saída o nome dos corredores, em ordem de chegada, e os nomes dos que tiveram seus carros quebrados, em ordem reversa de quebra.
Antes de serem impressos, os corredores que completaram a corrida devem ser armazenados em uma FILA, e os que quebraram em uma PILHA.

```
//Saida
1 - NOMEDOVENCEDOR1                          //X linhas, com os nomes de quem
completou a corrida, em ordem de chegada.
2 - NOMEDOVENCEDOR2
3 - NOMEDOVENCEDOR3
.
.
.
X - NOMEDOVENCEDORX

X+1 - NOMEDEQUEMQUEBROUY - Quebrou              //Y linhas, com os nomes de quem
quebrou, em ordem reversa de quebra.
X+2 - NOMEDEQUEMQUEBROUY-1 - Quebrou
X+3 - NOMEDEQUEMQUEBROUY-2 - Quebrou
X+4 - NOMEDEQUEMQUEBROUY-3 - Quebrou
.
.
.
X+Y - NOMEDEQUEMQUEBROU1 - Quebrou
//FIM DE ARQUIVO
```

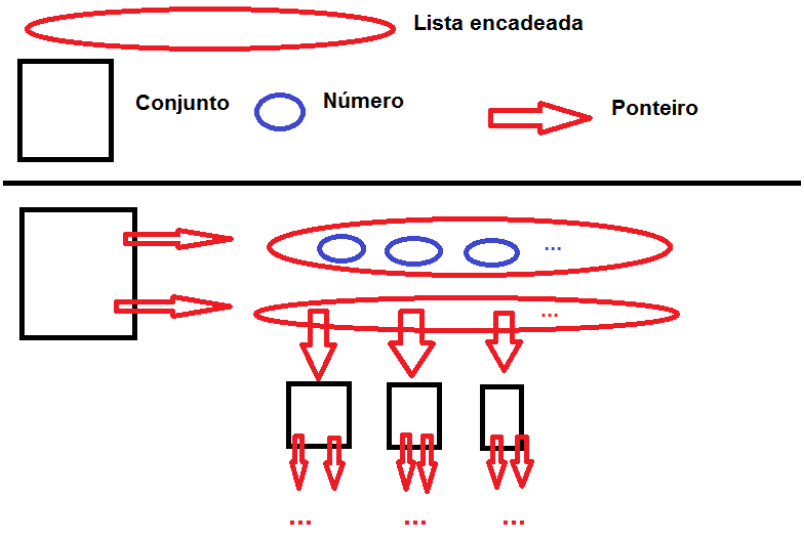
Questão 4) Brincando de conjuntos

Um conjunto pode conter ou não números, e pode conter ou não outros conjuntos.
Sendo assim, um mestre Jedi da matemática pediu pra você implementar em C um tipo estruturado de conjunto, sendo este formado por uma lista encadeada de inteiros e outra lista encadeada de conjuntos.

Dessa forma é recursivo já na definição, e a melhor forma de implementar as funções de adição de termo, remoção ou comparação é também por recursividade. Repare que o tipo estruturado conjunto não deve apontar pra outro conjunto. Ele deve ter dois ponteiros, um para uma lista encadeada de inteiros, e outro para uma lista de conjuntos (que não é um conjunto).
E cada nó da lista sendo um conjunto pode novamente conter outras listas, afinal,"{{1}}" existe e é diferente de {1}.

Adicionar termos já existentes ou remover termos não existentes não faz nada.
O vazio sempre estará presente em qualquer conjunto.
Ordem não importa, são conjuntos.

Exemplo de implementação de conjunto:



A entrada conterá um numero qualquer de linhas que podem seguir os seguintes comandos:

[char] = {...}
Inicializa um conjunto com o rotulo dito pela letra em char, guarde a referencia para possível acesso. Se a letra já existir, atualize o valor daquele conjunto. Imprima "[char] foi inicializado" caso ele não existisse anteriormente e "[char] teve seu valor modificado" caso ele já exista.

[char] < { ... }
Inclui os termos do conjunto escrito no conjunto já inicializado de rotulo [char]
Imprima "[char] teve termos adicionados" se termos foram realmente adicionados.

[char1] < [char2]
 Inclui todos os termos do conjunto char2 no conjunto char1, de forma há modificar futuramente char2 não alterar char1
Imprima "Os termos de [char2] foram adicionados a [char1]" se termos foram realmente adicionados

[char] > { ... }
Remove de char os termos contidos no conjunto descrito.
Imprima "[char] teve termos removidos" se algum termo foi de fato removido.

[char1] > [char2]
Remove os termos contidos em char2 de char1.
Imprima "Os termos de [char2] foram removidos de [char1]" se de fato algum termo foi removido.

Caso alguma das operações anteriores de fato nenhum termo for adicionado ou removido, imprima "[char(1)] nao foi modificado"

[char] ? { ... }
Imprima "[char] contem o conjunto descrito" na saida se o conjuto [char] contem todos os termos descritos dentro do outro conjunto nele. Imprima "[char] nao contem o conjunto descrito" caso contrario.

Pela definição matemática de conjuntos, {{1}} não contem "1" e se perguntado {{1}} ? {1} deve retornar falso. Porém contem "{1}" e deve responder positivo para {{1}} ? {{1}}. Também deve responder positivamente para {{1}} ? {}, afinal todo conjunto contém o vazio.

[char1] ? [char2]
Imprima "[char1] contem [char2]" ou "[char1] não contem [char2]" seguindo as mesmas regras anteriores.

[char]$ //delete da memoria o conjunto de rotulo char. Imprima "[char] foi deletado"

Tirando o comando de inicialização "=", todos os outros o(s) rotulo(s) já estara(ão) inicializado(s).
[char] é apenas uma letra, diferencie minúsculo de maiúsculo.
Em todas as operações { termo } é diferente de { { termo } }.
Desaloque tudo que restar ao final, e desaloque na linha aquilo que foi pedido. Não perca referencias.

```
Exemplo de in :

A  =  {1, 2, 3, {1,2}}
A  ? {1}
A  ? {{1}}
A  ?  {{1,2}}
B = {{1}, 1}
A ? B
B ? A
B >  {{1}}
A ? B
B $
A > {{1}}
//Fim de entrada.
// As entradas podem conter espaços, trate isso.
// Cada linha sempre terminara com um "\n" incluindo a ultima.

Out :
A foi inicializado
A contem o conjunto descrito
A nao contem o conjunto descrito
A contem o conjunto descrito
B foi inicializado
A não contem B
B não contem A
B teve termos removidos
A contem B
B foi deletado
A nao foi modificado
//Fim da saida
//A também será deletado, fim de arquivo.
```

Questão 5) Enigma Machine

Enigma é o nome por que é conhecida uma máquina electro-mecânica de criptografia com rotores, utilizada tanto para criptografar como para descriptografar mensagens secretas, usada em várias formas na Europa a partir dos anos 1920. A sua fama vem de ter sido adaptada pela maior parte das forças militares alemãs a partir de cerca de 1930. A facilidade de uso e a suposta indecifrabilidade do código foram as principais razões para a sua popularidade. O código foi, no entanto, decifrado, e a informação contida nas mensagens que ele não protegeu é geralmente tida como responsável pelo fim da Segunda Guerra Mundial pelo menos um ano antes do que seria de prever.

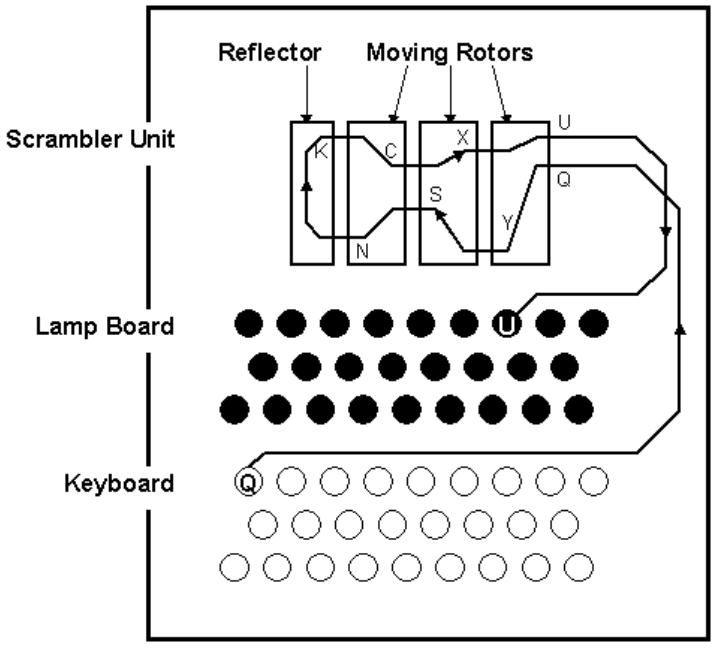
O monitores de IP-EC, precisando de uma maneira de conversar com os alunos sem que os maléficos alunos e monitores de IP-CC vejam os conteúdos de suas mensagens, solicitaram que você, um calouro extremamente habilidoso na linguagem C, implemente o seu algoritmo, de forma a simular o funcionamento de uma máquina enigma.

A máquina enigma consiste de vários rotores, com ligações elétricas, que permutam caracteres. Um rotor recebe um sinal, permuta a entrada, e repassa a informação para o rotor adjacente. Ao passar por todos os rotores, o sinal é permutado num refletor, e volta a passar por todos os rotores, resultando ao final no caractere encriptado.

Para fortalecer a encriptação, movimenta os rotores de permutação, seguindo uma lógica simples e eficaz.

O primeiro rotor é movido uma posição a cada caractere encriptado. Ao completar uma volta completa, o segundo rotor também é rotacionado em uma posição. Quando esse segundo rotor completar uma volta, o terceiro irá se mover uma posição, e assim sucessivamente.

Definindo a posição inicial de cada rotor(O quanto se rotacionava cada um antes de começar a encriptação), tinha-se uma chave de criptografia, o que aumentava ainda mais o número de possibilidades.



Estrutura:

Para a implementação, cada rotor e o refletor será representado por uma lista duplamente encadeada circular, onde cada nó interliga a margem esquerda com a margem direita do rotor. Essa ligação é determinada por um valor inteiro, que representa o deslocamento de posições equivalente à permutação.

Cada nó nos rotores possuirá duas informações, o valor de deslocamento direto(para baixo) e o de deslocamento inverso(sinal pós-refletor, para cima).

Já no refletor, cada nó guardará apenas um valor de deslocamento(para baixo)

Entrada:

A entrada consistirá de:

Um inteiro N, representando o número de rotores.

Um inteiro K, representando o tamanho do alfabeto.

K linhas, cada linha com um caractere do alfabeto.

N grupos de K linhas, onde cada grupo representa um rotor, e cada linha possui dois inteiros, o indíce de deslocamento direto e inverso.

K linhas, representando os deslocamentos do refletor.

Uma linha com N inteiros, definindo a chave de criptografia.

Uma mensagem a ser encriptada.

Saída:

A saída consistirá da mensagem encriptada.

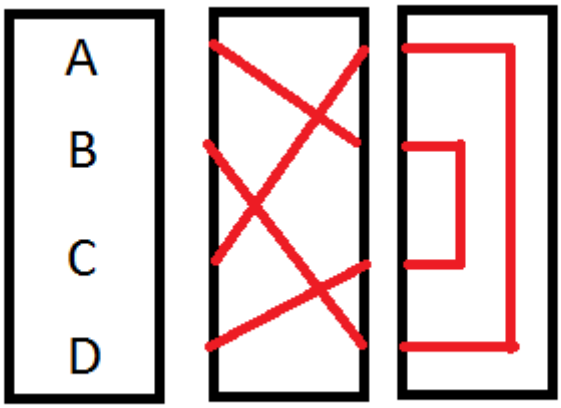
Considerando a seguinte entrada:

```
1
4
A
B
C
D
1 2
2 1
2 3
3 2
3
1
3
1
0
AB
```

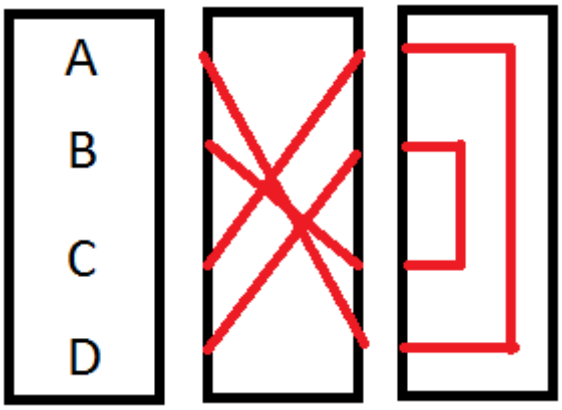
Obtemos a saída:

```
DD
```

Ao montarmos o rotor e o reflector, eles terão as seguintes ligações:

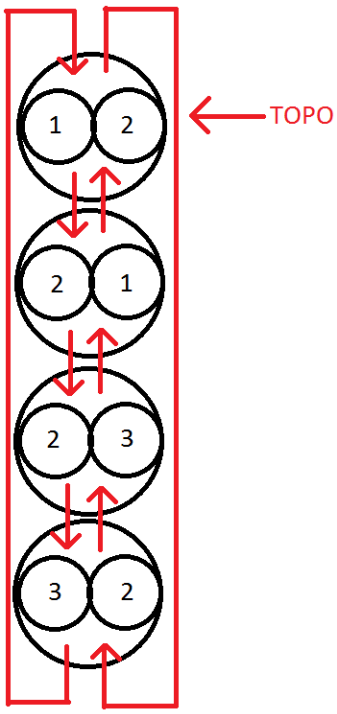


Dessa maneira, a entrada ‘A’ será encriptada para ‘D’.
Após essa encriptação, o rotor irá se mover em uma posição, ficando da seguinte forma:



De tal forma que, a letra ‘B’ também será encriptada para ‘D’.

Como uma lista encadeada, o rotor pode ser representado da seguinte forma:



Assim, cada nó desloca o fluxo **para baixo**, antes de passar pelo refletor, e **para cima**, depois. A movimentação do rotor consistirá simplesmente na mudança de seu topo(Ou seja, o head da lista)

Se, por exemplo, o fluxo deve entrar na segunda posição do rotor(Onde as posições começam em zero), a partir do topo, faz-se o acesso a dois elementos para baixo, acessando então o nó (2,3). Para saber qual posição do próximo rotor precisa ser acessada, somamos a posição atual(2), com o deslocamento direto(2), logo, é preciso acessar a o próximo rotor na quarta posição. Essa mesma lógica é aplicada no refletor.

Durante o caminho inverso, o funcionamento é praticamente igual, diferenciando-se apenas na movimentação pela lista, que agora ocorrerá para cima.