**Universidade Federal de Minas Gerais** **Algoritmos e Estruturas de Dados II**

**Instituto de Ciências Exatas** **2o Semestre de 2013**

**Departamento de Ciência da Computação**

Trabalho Prático 1 – Skip List – Estruturas de Dados
Valor: 15 pontos

Data de entrega no PRATICO (aeds.dcc.ufmg.br): 14/10/2013

Uma **lista encadeada** ou **lista ligada** é uma estrutura de dados composta por um conjunto de células onde cada célula aponta para a seguinte. Embora seja uma estrutura de dados relativamente simples, a lista encadeada **ordenada** apresenta custos relativamente altos para inserção, busca e remoção. Esse custo ocorre porque, para inserção ou remoção de uma célula, é preciso fazer uma busca prévia. A Figura 1 ilustra uma lista encadeada ordenada.

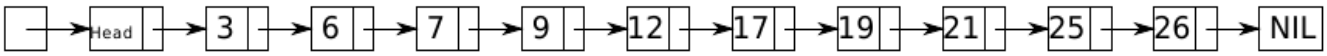

Figura 1: Lista encadeada ordenada

A *Skip List* é uma estrutura de dados voltada para chaves ordenadas cujo custo de inserção, remoção e busca é bem menor que na lista encadeada ordenada tradicional. Uma *Skip List* é implementada como um conjunto de listas encadeadas hierárquicas. Deste modo, a referida estrutura é construída em camadas. A lista mais ao fundo ($h = 0$) é uma lista encadeada ordenada tradicional, enquanto que cada lista na camada superior ($h + k$) contém uma cópia parcial da lista na camada inferior ($h + k$-1).

A quantidade de elementos das listas parciais pode variar, mas para o presente trabalho, a quantidade de elementos da lista superior será aproximadamente, metade da lista inferior. A Fig. 2 ilustra a estrutura básica de uma *Skip List*.

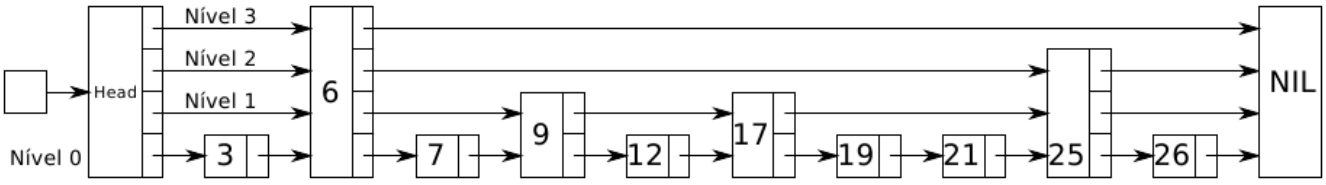

Figura 2: Estrutura de uma *Skip List*.

Note que o nível hierárquico 0 é equivalente a lista encadeada tradicional mostrada na Fig. 1, enquanto que as demais listas nos níveis hierárquicos superiores ($h > 0$) apresentam uma cópia parcial da sua respectiva lista mais inferior.

**Objetivo Geral**: o objetivo geral do Trabalho Prático 1 é a implementação de uma estrutura de dados "*Skip List*".

Objetivos Específicos: a estrutura de dados deverá contemplar as funções **FLVazia**, **Vazia**,

**Insere**, **Remove**, **Busca, Imprime** e **ImprimeTodos** conforme especificado nesse texto.

Os critérios a serem adotados na estrutura *Skip List* deste trabalho são:

- Os níveis hierárquicos serão numerados de baixo para cima começando por 0, vide Fig. 2.
- No presente trabalho, será fixado o número máximo de 5 níveis hierárquicos ($m$=5), ou seja, $0 \leq h \leq 4$ .
- A lista superior ($h$ + 1) deve ter aproximadamente metade dos elementos da lista inferior. Ou seja, $q_{h+1} \approx q_h/2$ , onde $q_h$ representa a quantidade de células no nível hierárquico $h$. Os elementos da lista ($h$ + 1) são selecionados aleatoriamente, contudo, para padronização das saídas no presente trabalho, a hierarquia será fornecida juntamente com a chave.

No TAD você deve criar um tipo (`typedef`) para representar a célula contendo sua chave e os demais atributos e ponteiros que se julgue necessário para a implementação do TAD "*Skip List*". Também devem ser implementadas as funções do TAD *Skip List* conforme lista abaixo:

```
void FLVazia(TipoLista *Lista);
```

A função `FLVazia` inicia a lista criando sua estrutura básica de funcionamento. Considerando a implementação com nó cabeça, a iniciação criará uma estrutura similar à ilustrada na Figura 3.

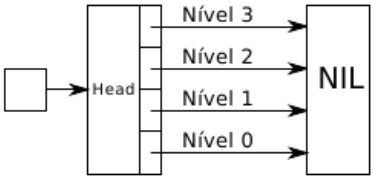


Figura 3: Estrutura de iniciação com nó cabeça para *Skip List*.

É importante destacar que no presente trabalho, sua estrutura terá 5 níveis, ou seja, um nível a mais que o ilustrado na Fig. 3.

```
int Vazia(TipoLista Lista);
```

Verifica se a lista está vazia, retornando `0`, ou se contém algum elemento, retornando `1`. A lista está vazia quando todos os ponteiros de próximo elemento da cabeça apontam para `NIL` como mostrado na Fig. 3.

```
int Busca(TipoChave chave, TipoLista *L, Apontador *ListaH, int
imprimir);
```

A busca será feita pela chave passada por parâmetro, retornando 1 caso o elemento seja encontrado e 0 caso contrário.

Em caso de sucesso, uma lista de ponteiros (`ListaH`) contendo o ponteiro de cada nível deve

ser retornado. Cada ponteiro do nível *h* (`ListaH[h]`), deve apontar para o elemento anterior ao elementos chave buscado, ou o elemento anterior ao primeiro elemento maior que a chave caso a mesma não exista em determinado nível, ou ainda o elemento anterior a `NIL` caso não haja elemento maior que a chave em determinado nível. Ou seja, `ListaH[h]` guarda o ponteiro anterior ao elemento chave buscado ou a posição anterior a possível inserção caso ele não exista. O algoritmo abaixo esclarece melhor a definição.

A função busca é uma das mais importantes neste TP, pois, também serve como base para as funções `insere` e `remove`. No caso da inserção, a função busca fornece os ponteiros das células anteriores em cada nível hierárquico à posição de inserção de uma nova célula. No caso da remoção, a função fornece os ponteiros em cada nível que apontam para a célula anterior a célula a ser excluída. Lembre-se de que lista de ponteiros é obtida pela função `Busca` através do parâmetro `ListaH`.

**Algoritmo da função de busca.**

A função de busca, começa pesquisando o elemento chave na lista mais superior. Se o próximo elemento for `NIL` ou a chave for maior que o elemento de busca, então é preciso descer um nível e continuar a busca horizontalmente até o próximo decremento de nível. A cada "descida", o ponteiro para o próximo elemento deve ser guardado em `ListaH[h]`. A seguir é apresentado o pseudo algoritmo para busca bem como um exemplo ilustrado na Fig. 4.

```
1   int Busca(TipoChave chave, TipoLista *L, Apontador *ListaH, int imprimir){
2
3      x = lista->primeiro;  // primeiro elemento (cabeça ou head)
4      // neste exemplo, max_hierarquia = 3
5      para h = max_hierarquia decremente Ate 0 faça{
6
7         enquanto ((x->prox[h].chave < chave) && (x->prox[h] != NIL)){
8            x = x->prox[h];
9         }
10
11        ListaH[h] = x;
12
13     }
14
15     se (x->prox[h].chave == chave) então
16        retorne 1;
17     senao
18        retorne 0;
19
20  }
```

OBS.: O pseudo algoritmo acima deve ser convertido para linguagem C de acordo com seu TAD. Este pseudo algoritmo não mostra os detalhes de declaração de variáveis nem outros passos considerados triviais.

Para ilustrar a função busca, considere localizar a chave 17. Os passos da busca são ilustrados na Fig. 4 seguindo o caminho tracejado em azul.

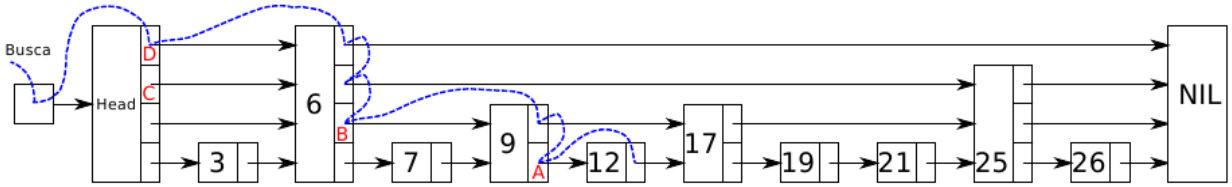

Figura 4: Busca da chave 17.

Neste exemplo (Fig. 4), `ListaH` deve retornar os seguintes ponteiros (na orderm do nível hierárquico de 0 ao 3): {A, B, C, D}.

Caso o pamametro `imprimir` da busca seja igual a 1, a função também deve imprimir no arquivo de saída as chaves e os seus respectivos níveis hierárquicos visitados, sendo uma linha para cada par chave/hierarquia (`c h`) visitada. Para a busca da chave 17, ilustrada na Fig. 4, a saída apresentada deverá ser:

```
6 3
6 2
6 1
9 1
9 0
12 0
17 0
```

**Nota:** A função de busca deverá imprimir os elementos apenas quando a busca for a função principal. Quando for uma função auxiliar como em inserir e remover, a busca **não** deverá imprimir o caminho percorrido nem o resultado `true` ou `false`.

```
int Insere(TipoItem x, int h, TipoLista *Lista);
```

Esta função deve inserir um novo elemento na estrutura do *Skip List* mantendo a consistência da lista. A inserção com sucesso deve retornar `1`, ou `0` caso contrário. Obviamente, o elemento chave não pode ser duplicado, retornando `0` na função insere.

Cada inserção deverá imprimir no arquivo de saída se a operação teve sucesso ou não, sendo, `true` para sucesso e `false` caso contrário.

O primeiro passo antes da inserção é a busca da chave a ser inserida, o que irá retornar a posição em que o elemento deverá ser inserido, caso a chave de fato não exista. Estas posições representadas pelos ponteiros de cada lista são obtidas em `ListaH`.

Para a inserção de um novo elemento na lista, será fornecida tanto a sua chave quanto a sua hierarquia (`c h`).

OBS.: Na busca feita para a função insere, o caminho da busca não será impresso, ou seja, o parâmetro `imprimir` da função de busca será igual a 0.

**Importante:** a hierarquia definida no par de inserção (`c h`) implica na inserção da chave *c* na

hierarquia *h* e em todas as suas listas hierarquias inferiores, até a lista 0. Por exemplo, a inserção da chave 6 na hierarquia 3, implica na inserção da chave em todas as listas como mostrado na Fig. 2.

A inserção de alguns elementos é ilustrada a seguir. Considere o par chave/hierarquia (`c h`).

**a)** Inserção da chave 12 na hierarquia de nível 0 (`12 0`). A busca do elemento 12 retorna `false` e todos os elementos anteriores em `ListaH` apontam para `NIL`. A inserção é feita apenas no nível 0.

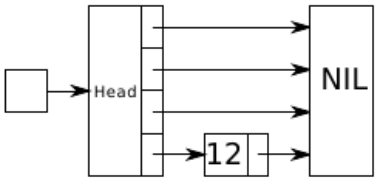

Figura 5: Inserção da chave 12 na lista hierárquica 0.

**b)** Inserção da chave 9 na hierarquia de nível 1 e 0 (`9 1`). Na inserção de (`9 1`), a busca pela chave 9 retorna `false` e `ListaH[0]` aponta para 12 enquanto os demais elementos de `ListaH` apontam para `NIL` (Fig. 5).

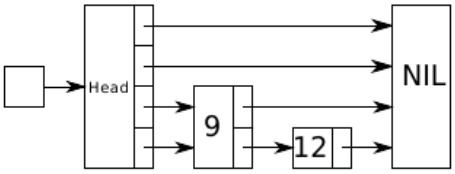

Figura 6: Inserção da chave 9 na lista hierárquica 1 e demais inferiores.

Após a inserção da chave 9, os ponteiros ficam configurados como mostrado na Fig. 6.

**c)** Inserção da chave 26 na hierarquia 0 (`26 0`).

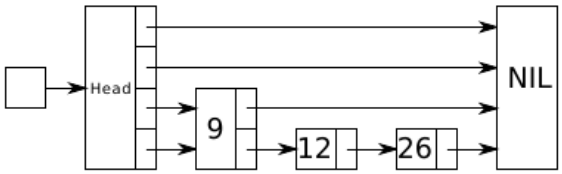

Figura 7: Inserção da chave 9 na lista hierárquica 0.

A inserção da chave 26 é realizada apenas na hierarquia 0, sendo modificada apenas a lista no nível 0.

**d)** Inserção da chave 25 na hierarquia 2 e demais hierarquias inferiores (`25 2`).

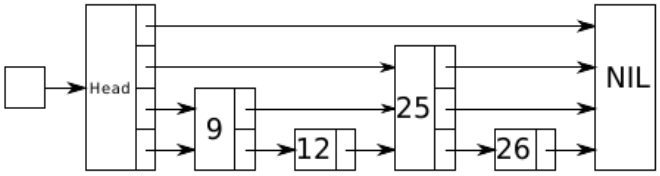

Figura 8: Inserção da chave 25 na lista hierarquica 2 e demais hierarquia(s) inferior(es).

A inserção da chave 25, começa pela hierarquia de nível 2 afetando não só essa lista, mas também as listas inferiores 1 e 0. É preciso conhecer os ponteiros das células anteriores das três ultimas listas.

```
int Remove(TipoChave chave, TipoLista *L);
```

Esta função deve remover o elemento indicado com a `chave`. A remoção deve previamente fazer uma busca pelo elemento chave por dois motivos: (*i*) a busca confirma a existência ou não do elemento indicado pela chave e (*ii*) a busca fornece todos os ponteiros anteriores que apontam para o elemento a ser excluído. A manutenção desses ponteiros deve ser feita para manter a consistência da lista.

A remoção de um elemento deve fazer cada ponteiro, que aponta para o elemento a ser retirado, apontar para a célula apontada pelo elemento retirado em sua respectiva hierarquia, de forma similar ao que é feito para a remoção de um elemento numa lista encadeada comum.

A Fig. 9 ilustra a retirada do elemento de chave igual a 25. Cada ponteiro que aponta para a célula 25 passará a apontar para o elemento apontado por 25 no seu respectivo nível hierárquico. Por exemplo, o ponteiro da célula 12 no nível 0 passará a apontar para 26 enquanto que o ponteiro do elemento 9 no nível 1 passará a apontar para `NIL`, e assim sucessivamente. Na Fig. 9 os ponteiros tracejados que apontam para 25 são redirecionados conforme os ponteiros em verde.

Finalmente, para concluir a função `remove`, a alocação de memória da célula retirada deve ser liberada.

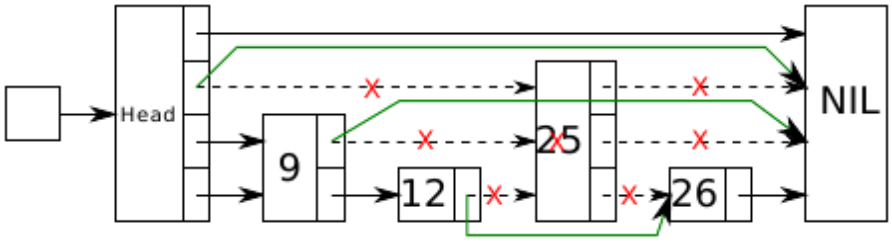


Figura 9: Remoção da chave 25.

Neste exemplo, após a retirada da célula de chave 25, a estrutura deve ser tal como representada na Fig. 7.

Cada tentativa de retirada deverá imprimir no arquivo de saída se a operação teve sucesso ou não, sendo `true` para sucesso e `false` caso contrário.

OBS.: Na busca feita para a função remove, o caminho da busca não será impresso, ou seja, o parâmetro `imprimir` da função de busca será igual a 0.

```
void Imprime(TipoLista Lista, int h);
```

Lista todos os elementos no nível hierárquico *h*. Cada chave deve ser impressa em uma linha

separada contendo o par chave/hierarquia (`c h`).

```
void ImprimeTodos(TipoLista Lista);
```

Lista todos os elementos por nível hierárquico. Primeiramente lista os elementos no nível hierárquico mais alto (4), em seguida no nível inferior, e assim sucessivamente até o nível mais inferior (0). Cada chave deve ser impressa em uma linha separada contendo o par chave/hierarquia (`c h`).

**Modelo de Entrada**

O programa principal deverá ler, da enrada padrão, o arquivo de entrada. Quando executado na linha de comando, usar:

```
./programa < entrada.txt
```

A saída deverá ser direcionada para o dispositivo de saída padrão. (`stdout`).

Os arquivos de entrada serão listas de chaves a serem inseridas ou removidas, podendo conter comandos de impressão e busca. Cada linha tem um comando, caso o comando seja de inserção, remoção ou busca, o mesmo será seguido da chave `c`. Os comandos são:

- `I c h` para inserir a chave `c` na hierarquia `h`;
- `R c` para remover a chave `c`;
- `B c` para buscar a chave `c`;
- `P h` para imprimir a lista na hierarquia *h* usando a função `Imprime`;
- `A` para imprimir todas as listas das hierarquias usando a função `ImprimeTodos`.

Um exemplo de arquivo de entrada é mostrado a seguir:

```
I 12 0
I 9 1
I 26 0
I 25 2
B 26
P 0
R 25
A
```

**Modelo de saída**

Cada saída deverá ser escrita em uma nova linha. Em outras palavras, não é admitido mais que uma saída por linha. As possíveis saídas são listadas a seguir:

| | |
|---|---|
| `true` | Para operação executada com sucesso |
| `false` | Para operação executada com algum erro |

Use EXATAMENTE a saída conforme especificado para manter a conformidade com o PRATICO. A saída referente a entrada mostrada nesse texto é apresentada a seguir:

```
true
true
true
true
25 2
25 1
25 0
26 0
true
9 0
12 0
25 0
26 0
true
9 1
9 0
12 0
26 0
```

**Note que o programa principal não poderá acessar diretamente a estrutura interna do TAD. Se necessário, acrescente novas funções ao seu TAD detalhando-as na documentação do trabalho.**
**O programa criado não deve conter "menus interativos" ou "paradas para entrada de comandos" (como o system ("PAUSE") por exemplo). Ele deve apenas ler os arquivos de entrada, processá-los e gerar os arquivos de saída.**

**Os TPs serão corrigidos em um ambiente Linux. Portanto, o uso de bibliotecas do Windows está PROIBIDO.**

**O que deve ser entregue:**

- Código fonte do programa em C (todos os arquivos .c e .h), bem comentado.
- Arquivo executável.
- Documentação do trabalho. Entre outras coisas, a documentação deve conter, sucintamente:

    1. **Introdução:** descrição sucinta do problema a ser resolvido e visão geral sobre o funcionamento do programa.
    2. **Implementação:** descrição sobre a implementação do programa. Deve ser detalhada a estrutura de dados utilizada (de preferência com diagramas ilustrativos), o funcionamento das principais funções e procedimentos utilizados, o formato de entrada e saída de dados,

compilador utilizado, bem como decisões tomadas relativas aos casos e detalhes de especificação que porventura estejam omissos no enunciado.

3. **Estudo de Complexidade:** estudo da complexidade do tempo de execução das funções implementadas e do programa como um todo (notação *O*), considerando conjuntos de tamanho *n*. Também deve ser apresentado um estudo de complexidade **teórica** para busca, inserção e remoção do *skip list.* Considere uma quantidade arbitrária de níveis hierárquicos, tantos quantos forem necessários, respeitando sempre a quantidade de elementos no nível *h* + 1 igual a metade dos elementos no nível *h* ( $q_{h+1} = q_h / 2$ ). Consiere também que a quantidade de níveis *m* é dada por: $m = \lfloor \log_2(q_0) \rfloor + 1$ .
4. **Testes:** descrição dos testes realizados e listagem da saída (não edite os resultados).
5. **Conclusão:** comentários gerais sobre o trabalho e as principais dificuldades encontradas em sua implementação.
6. **Bibliografia:** bibliografia utilizada para o desenvolvimento do trabalho, incluindo sites da Internet se for o caso.

Obs: Um exemplo de documentação está disponível no Moodle.

Comentários Gerais:

1. Comece a fazer este trabalho logo, enquanto o problema está fresco na memória e o prazo para terminá-lo está tão longe quanto jamais poderá estar.
2. Clareza, indentação e comentários no programa também serão avaliados.
3. O trabalho é individual.
4. A submissão será feita pelo sistema online (`http://aeds.dcc.ufmg.br`).
5. Trabalhos copiados, comprados, doados, etc. serão penalizados conforme anunciado.
6. Na prova 2, uma das questões poderá ser sobre a implementação do trabalho. A nota do trabalho será ponderada pela nota dessa questão.
7. Caso se julgue necessário, poderá ser marcada uma entrevista com o aluno para apresentação do trabalho.
8. Penalização por atraso: $(2^d - 1)$ pontos, onde *d* é o número de dias de atraso.