Trabalho Prático 1:
Banco de Dados com Árvores B^+

Algoritmos e Estruturas de Dados III – 2016/1
Entrega: 28/04/2016

1 Introdução

Árvores B são estruturas balanceadas em que um nó pode ter um número vasto de filhos. Assim, ao se buscar por um elemento, é necessário percorrer menos níveis, o que as torna ideais para armazenar grandes volumes de dados e buscá-los de forma eficiente.

As árvores B^+ são um tipo de árvores B, e são comumente usadas em sistemas de bancos de dados reais. As principais diferenças entre a árvore B e a B^+ se dão nos nós de armazenamento dos dados e no acesso de uma folha a outra. Na árvore B, os registros são armazenados juntamente com as suas chaves associadas, inclusive em nós intermediários. Por outro lado, em árvores B^+ os registros são armazenados somente em nós folhas, dessa forma, permitindo que nós intermediários possam armazenar mais chaves (*fanout* maior). Consequentemente, árvores B^+ possuem alturas menores, para uma mesma quantidade de memória para os nós, o que permite que registros sejam acessados em um número menor de passos. Além de um *fanout* maior, árvores B^+ possuem outra diferença importante: suas folhas possuem apontadores para as próximas folhas, o que permite caminhar eficientemente, de forma sequencial, pelos registros da árvore.

Neste trabalho, vocês terão de simular um banco de dados simplificado, com operações simples de busca e inserção usando árvores B^+.

2 Árvores B^+

As árvores B foram propostas por [Bayer and McCreight, 1972] para a organização de informação em discos magnéticos e outros meios de armazenamento secundário. Em 1979, o uso de árvores B já era praticamente o padrão adotado em sistemas de arquivos de propósito geral para a manutenção de

índices para bases de dados. As árvores B^+ foram especialmente projetadas para pesquisa de informação, portanto, elas buscam minimizar o número de operações de movimentação de dados (escrita / leitura) em uma pesquisa ou alteração. Um exemplo em alto nível de uma árvore B^+ pode ser visto na Figura 1.

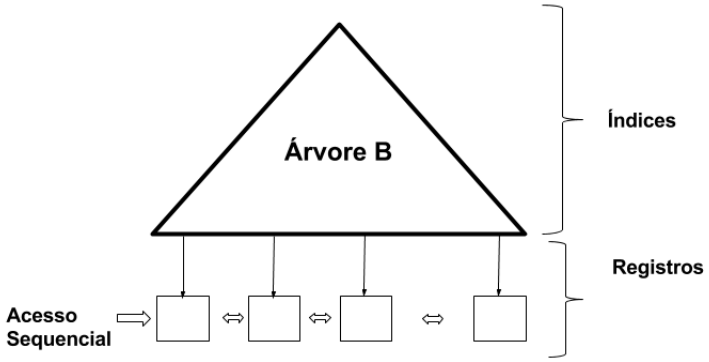

Figura 1: Ilustração alto nível de uma árvore B^+.

A árvore B^+ permite as seguintes operações: Inserção, Busca e Remoção. Neste trabalho explicaremos apenas as duas primeiras, uma vez que somente suas implementações serão cobradas.

Genericamente, em uma árvore B^+, sendo m a sua ordem, cada um dos nós internos, com exceção da raiz, possuem entre $\lceil m/2 \rceil$ e m sub-árvores, e a quantidade de elementos varia entre $\lceil m/2 \rceil - 1$ e $m - 1$ (pode variar em um elemento, dependendo da implementação). Faremos a seguir uma breve descrição das operações que serão cobradas, apresentando breves exemplos. Destacamos que os tamanhos utilizados para as árvores são para facilitar a explicação, e que existem pequenas divergências de acordo com a referência utilizada.

Inserção – Exemplo

A Figura 2 ilustra uma árvore B^+, cuja ordem é igual a 5. Para demostrar a operação de inserção iremos adicionar um registro, fazendo com que posteriormente tenhamos de balancear a árvore.

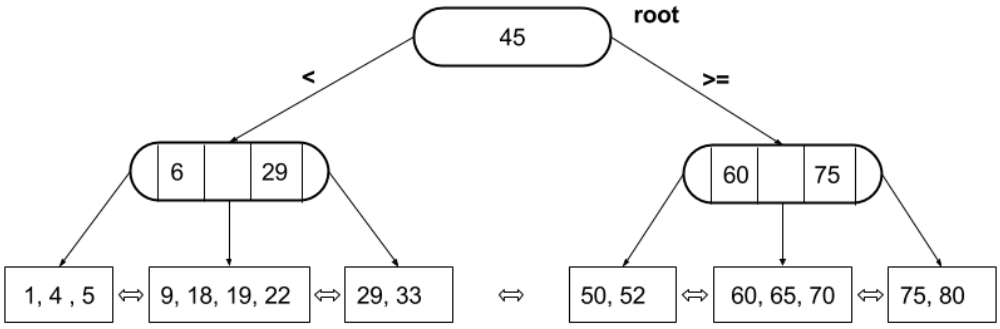

Figura 2: Árvore B^+.

Passo 1 Para inserir um novo registro, devemos primeiramente localizar a folha em que ele deve ser inserido. Uma vez que já possuímos essa informação, devemos inseri-lo. Na Figura 3, ilustramos o bloco que foi modificado pela inserção, e ainda, destacamos o novo registro.

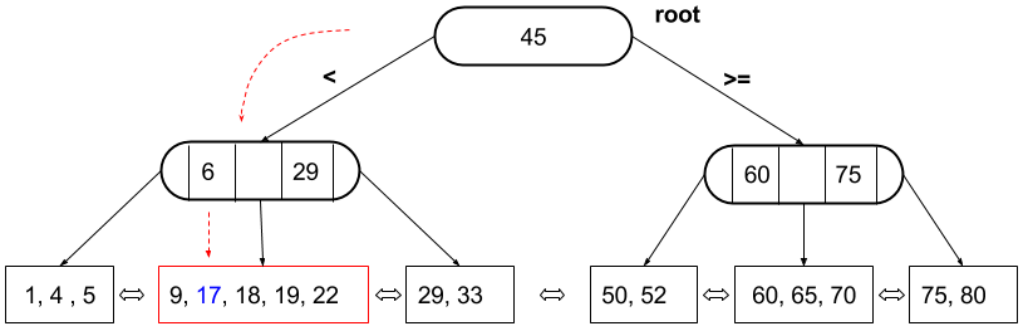

Figura 3: Árvore modificada pela inserção de um registro.

Passo 2 Após a inserção, percebemos que o novo registro causa um acúmulo de registro maior do que é permitido para um nó. Assim, a inserção do elemento requer uma divisão do nó. Por ser o elemento central, uma cópia do índice 18 é movido para o nó pai e é criada uma nova folha para poder particionar o nó antigo. Uma demostração deste processo pode ser vista na Figura 4.

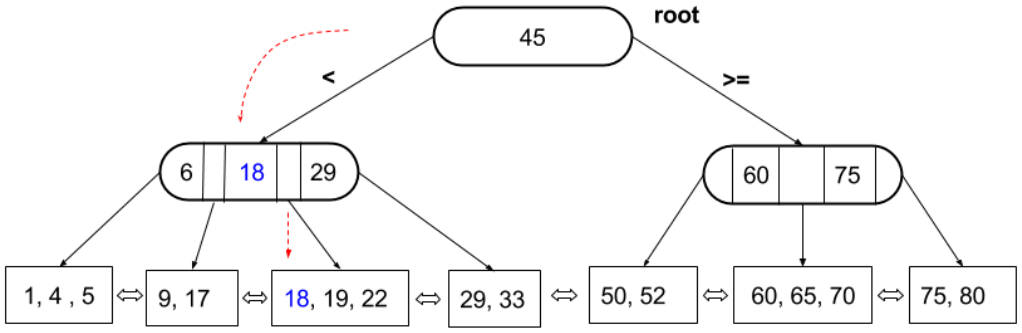

Figura 4: Estado da árvore após o balanceamento dos registros.

Busca

A busca em uma árvore B^+ é realizada da mesma forma que em uma árvore binária, entretanto, ao invés de duas escolhas, agora temos múltiplas decisões pelo fato de estarmos lidando com uma árvore n-ária. Inicia-se a busca na raiz da árvore e a pesquisa continua até que a chave procurada seja encontrada em um nó folha, uma vez que todos os registros residem neles.

3 Banco de Dados Simplificado

O banco de dados a ser implementado deverá ser capaz de gerenciar GBs de dados e, portanto, terá que utilizar memória secundária. Para permitir que operações de inserção e busca sejam executadas de forma rápida, apesar dos registros estarem armazenados em disco, o seu banco de dados simplificado irá indexar os registros utilizando árvores B^+. A informação de qual coluna dos dados deverá ser utilizada como índice será fornecida junto da entrada (Seção 5). Estas colunas: (i) serão campos numéricos, para facilitar a comparação de chaves; (ii) terão valores únicos. Por exemplo, em um banco de dados de clientes esse campo poderia ser o CPF. Você pode assumir que o valor dos campos numéricos que serão usados como índice cabem em uma variável **unsigned long long int**.

No restante desta seção iremos descrever as operações que o sistema de banco de dados deve suportar e como elas se relacionam com o índice que será construído.

3.1 Sintaxe das operações

No banco de dados simplificado, apenas três operações existirão: busca, inserção e impressão. Cada operação será descrita por uma linha contendo:

$$\langle operacao \rangle \langle tab \rangle \langle arg_1 \rangle \langle tab \rangle \langle arg_2 \rangle \ \ldots \ \langle arg_n \rangle$$

3.2 Busca

Na busca exata, apenas o registro com exatamente o valor fornecido deve ser retornado. Note que um índice em um banco de dados não pode ter dois registros com chaves iguais. A palavra-chave da busca será *search* e o restante da sintaxe é descrita abaixo:

$$search \langle tab \rangle \langle key \rangle$$

onde $\langle tab \rangle$ corresponde à tabulação, que no editor de texto é emitido pela tecla tab e no código corresponde ao $\backslash t$. Essa operação deverá imprimir um resultado em uma linha separada. Caso um registro seja encontrado, ele deve ser impresso preservando a ordem dos campos no arquivo **.csv** e os campos devem ser separados entre si por um $\langle tab \rangle$. Caso não seja encontrado nenhum registro, deve ser impresso *null*.

3.3 Inserção

O objetivo da inserção é incluir um registro completo no banco de dados. Um registro será composto por vários campos. A palavra-chave da operação será *add* e cada campo será passado como um argumento:

$$add \langle tab \rangle \langle field_1 \rangle \langle tab \rangle \langle field_2 \rangle \langle tab \rangle \ldots \langle tab \rangle \langle field_n \rangle$$

onde $\langle field_i \rangle$ é o dado do campo i do registro. Você pode assumir que os dados sempre estarão na ordem correta dos campos e que eles serão sempre válidos, ou seja, eles terão chave única, terão o tipo correto e caberão no espaço devido. Essa operação não imprimirá nenhum valor, uma vez que seu objetivo é modificar a árvore como um todo.

3.4 Impressão

O objetivo dessa operação é imprimir a estrutura da árvore no momento atual. A operação não terá nenhum parâmetro e será indicada pela pela palavra chave *dump*.

dump

A impressão começará a partir da raiz e deverá ser feita em *largura*, ou seja, todos os nodos do nível i serão impressos antes dos nodos do nível $i+1$. Dentro de um nível, os nodos serão impressos da esquerda para a direita, um nodo por linha. A impressão de um nodo consistirá em imprimir somente as chaves armazenadas neste, com chaves adjacentes separadas por vírgula. Por exemplo, se fôssemos imprimir a árvore mostrada no exemplo da figura 4, o *dump* seria o seguinte:

```
45,
6,18,29,
60,75,
1,4,5,
9,17,
18,19,22,
29,33,
50,52,
60,65,70,
75,80,
```

Note no final de cada linha tem uma vírgula. Não só isso facilita a impressão do nodo mas também é indispensável para que o script de correção consiga avaliar a estrutura da árvore de forma adequada. Para que vocês tenham uma ideia melhor de como caminhar pela árvore e realizar a impressão da sua estrutura, usem o algoritmo de Busca em Largura abaixo.

Busca em Largura

Busca em largura é uma estratégia geral de caminhamento em grafos, que nesse trabalho será usada para imprimir os nós da árvore B^+ em níveis. No Algoritmo 1, é apresentada a estratégia desse caminhamento. Ele funciona basicamente da seguinte forma: partindo de um nó inicial, ele explora todos os nós filhos, colocando os ponteiros para eles em uma fila para que seja possível manter uma ordem dos próximos nós que serão visitados. À medida em que ele vai desenfileirando os nós, ele repete o processo de visitar os filhos e enfileirá-los. Assim, o algoritmo continua até que todos os nós sejam desenfileirados. Note que a implementação desse algoritmo dará exatamente a impressão em largura pedida.

Algoritmo 1: Busca em largura adaptado de [Cormen, 2009].

```
   Data: Tree
 1 begin
 2 |   Escolha a raiz root em Tree
 3 |   Insira root em Fila
 4 |   while Fila não estiver vazia do
 5 |   |   Seja n o primeiro nó em Fila
 6 |   |   for cada f ∈ filho de n do
 7 |   |   |   Insira f em Fila
 8 |   |   end
 9 |   |   Remova n de F
10 |   |   Imprima n
11 |   end
12 end
```

3.5 [Extra]

Múltiplos índices Nos testes obrigatórios, a cada execução, apenas um campo do registro será considerado como índice. Aqueles que fizerem o trabalho de forma a suportar as operações para múltiplos índices ganharão ponto(s) extra(s). Para isso, pense em todas as implicações que isso pode gerar e adapte o seu algoritmo. Assim, todos os campos que serão usados como índices serão passados como argumentos do programa, e a sintaxe da operação de busca passa a ser:

$$search\langle tab\rangle\langle index\_id\rangle\langle tab\rangle\langle key\rangle$$

onde *index_key* é o índice do campo pelo qual deve ser feita a busca. Além disso, a operação de impressão passa a receber um parâmetro, que é o índice a ser considerado na hora de imprimir a árvore:

$$dump\langle tab\rangle\langle index\_id\rangle$$

Busca por intervalo Na busca por intervalo, todos os registros cujas chaves estejam dentro do intervalo fechado devem ser retornados. Sua sintaxe é:

$$search\langle tab\rangle\langle min\_key\rangle\langle tab\rangle\langle max\_key\rangle$$

e devem ser impressos todos os registros encontrados, um por linha, da forma descrita na seção 3.2.

Note que ambos os tópicos podem ser implementados. Nesse caso, é só acrescentar o parâmetro *index_key* à sintaxe da busca por intervalo:

$$search\langle tab\rangle\langle index\_key\rangle\langle tab\rangle\langle min\_key\rangle\langle tab\rangle\langle max\_key\rangle$$

4 O que deve ser implementado

Você deve implementar uma simulação de um sistema de banco de dados simplificado que suporte as operações descritas na seção 3 e que siga o formato de entrada e saída descritos em 5. Além disso, o seu sistema de banco de dados **deverá utilizar uma árvore B$^+$ para indexar os registros** e armazenar **todos os registros em disco**. A implementação específica da árvore B$^+$ será livre, com exceção de dois pontos:

1. **Somente o nó raiz pode ficar em memória**: Entre duas operações, o único nodo que pode estar em memória é o nó raiz, e os demais deverão estar armazenados em disco. Para processar as operações, você pode manter os nodos em memória **temporariamente**, conforme for necessário.

2. Dependendo da referência utilizada para estudar a árvore, a nomenclatura e a implementação podem mudar. Seja m **a ordem da árvore**, existem referências cujo número máximo de elementos é m e outras em que o máximo é $m-1$. E também existem referências cujo número mínimo de elementos em nós não-raiz são $\lceil \frac{m}{2} \rceil$ e outras em que esse número é $\lceil \frac{m}{2} \rceil - 1$. Será checado se os nós da sua árvore têm, no mínimo, $\lceil \frac{m}{2} \rceil - 1$ itens (exceto a raiz) e se têm, no máximo, m itens. Além disso, outras características serão verificadas para checar a consistência da sua árvore B$^+$.

Siga esses critérios fielmente, pois durante os testes iremos restringir a quantidade máxima de memória que o seu programa poderá usar, de forma que se a árvore não estiver em disco ou se não estiver devidamente balanceada o seu TP não executará.

O `Makefile` que será submetido com o código deve permitir que o programa seja compilado ao digitar o comando *'make all'* no terminal e o executável deverá ser nomeado: `tp1.exe`. É de extrema importância que esses critérios sejam seguidos à risca, para que seu código seja compatível com os scripts de teste. Caso contrário, o seu TP não poderá ser testado e, consequentemente, você irá tirar 0.

5 Entrada e Saída

A entrada será passada como um arquivo **.csv**, em que cada linha corresponde a uma operação. Para a operação de busca, a saída deve ser impressa da seguinte forma:

search
$\langle saida\_operacao_i \rangle$
search

e para a operação de impressão:

dump
$\langle saida\_operacao_i \rangle$
dump

de forma a separar as saídas de operações diferentes, pois elas podem ocupar mais de uma linha, cada. Ao final do arquivo, deverá ser impressa uma linha vazia. Isso é para facilitar a impressão, já que haverá uma quebra de linha ao final da saída de todas as operações.

O seu programa deverá ler da linha de comando cinco parâmetros, **nessa ordem:** (i) nome do arquivo de saída; (ii) nome do arquivo de entrada; (iii) ordem da árvore; (iv) número de campos no registro; (v) id do campo a ser usado como índice (começando de zero). A seguir, há um exemplo de como seu programa será executado pelo sistema de teste:

```
./tp1.exe saida teste.csv 100 4 3
```

Note que a extensão dos arquivos de entrada e saída não é relevante, seu programa deve tratar com arquivos de texto de quaisquer extensões.

Caso seu trabalho suporte múltiplos índices, o seu programa receberá mais de cinco parâmetros. Você deve checar o número de índices passados, eles estarão sempre no final.

6 O que deve ser entregue

Deverá ser submetido um arquivo *.zip* contendo somente uma pasta chamada `tp1` e dentro desta deverá ter: (i) Documentação e (ii) Implementação.

Documentação Poderá ter no máximo 10 páginas e deverá seguir tanto os critérios de avaliação discutidos na Seção 7.1, bem como as diretrizes sobre a elaboração de documentações disponibilizadas no *moodle*. Além desses requisitos básicos, a documentação do tp1 deverá ter:

1. Experimentos mostrando a variação no tempo de execução e na profundidade máxima atingida da árvore, para diversos valores de ordem da árvore.

2. Discussão sobre a utilização de árvores B^+ para bancos de dados, baseada nos resultados dos experimentos.

3. Discussão sobre o motivo da ordem da árvore não poder ser maior que o tamanho de um página do sistema de memória virtual.

4. Como esse trabalho poderia ser melhorado.

Implementação Código fonte do seu TP (*.c* e *.h*) e um arquivo *Makefile* para realizar a compilação do seu código, atendendo aos critérios descritos na Seção 4.

7 Avaliação

Seu trabalho será avaliado de acordo com a qualidade do código e documentação submetidos. Eis uma lista **não exaustiva** dos critérios de avaliação que serão utilizados.

7.1 Documentação

Introdução Inclua uma breve explicação do problema que está sendo resolvido no seu trabalho e um resumo da sua solução.

Solução do Problema Você deve descrever a solução do problema de maneira clara e precisa, detalhando e justificando os algoritmos e estruturas de dados utilizados. Para tal, artifícios como pseudo-códigos, exemplos ou diagramas podem ser úteis. Note que documentar uma solução não é o mesmo que documentar seu código. **Não** é necessário incluir trechos de código em sua documentação nem mostrar detalhes de sua implementação, exceto quando estes influenciem o seu algoritmo principal, o que se torna interessante.

Análise de Complexidade Inclua uma análise de complexidade de tempo e espaço dos principais algoritmos e estrutura de dados utilizados. Cada complexidade apresentada deverá ser devidamente **justificada** para que seja aceita.

Avaliação Experimental Sua documentação deve incluir os resultados de experimentos que avaliem o tempo de execução de seu código em função de características da entrada. Cabe a você gerar entradas para esses experimentos. Por exemplo: se esse trabalho fosse sobre ordenação, seria interessante mostrar como o tempo de execução de cada algoritmo varia quando o número de itens a serem ordenados aumenta. Para tal, um gráfico mostrando o tempo de execução em função do tamanho da entrada pode ser interessante. Você também deve interpretar os resultados obtidos. Comente sobre cada gráfico ou tabela que você apresentar mostrando o que é possível concluir a partir dele. **DICA:** Use o guia e exemplos de documentação disponíveis no moodle.

7.2 Implementação

Linguagem & Ambiente O seu programa deverá ser implementado na linguagem **C** e poderá fazer uso de funções da biblioteca padrão da linguagem. Trabalhos que utilizem qualquer outra linguagem de programação e/ou que façam uso de outras bibliotecas que não a padrão serão zerados. Além disso, certifique-se que seu código compile e funcione corretamente nas máquinas **LINUX** dos laboratórios do DCC.

Casos de teste A sua implementação passará por um processo de correção automatizado e, portanto, o formato da saída do seu programa deve ser idêntico àquele descrito nessa especificação. Saídas com qualquer divergência serão consideradas erradas, mesmo que as divergências sejam *whitespaces*. e.g. espaços, *tabs*, quebras de linha, etc. Para auxilia-lo na depuração do seu código, será fornecido um pequeno, **não-exaustivo**, conjunto de entradas e suas respectivas saídas. É seu dever certificar-se que seu código funciona corretamente para qualquer entrada válida.

Alocação Dinâmica Algoritmos e estruturas de dados deverão fazer uso de memória alocada dinâmicamente (`malloc()` ou `calloc()`). Certifique-se que seu programa utiliza essas regiões de memória corretamente, pois os monitores penalizarão implementações que realizam *out-of-bounds access* e que tenham vazamento de memória (não desalocar memoria dinâmica).
DICA: Utilize `valgrind` antes de submeter o seu TP.

Qualidade do código Seu código também será avaliado no quesito de legibilidade, dando atenção, porém não limitando-se, aos seguintes items: (i) **INDENTAÇÃO**; (ii) nomes de variável e função descritivos e claros; (iii) Modularização adequada; (iv) Comentários dentro de funções, explicando o

que certos trechos mais complicados fazem; (v) Comentários fora de funções, explicando, em alto-nível, o que as funções mais importantes fazem; (vi) funções concisas que desempenham somente uma tarefa; (vii) **Proibido uso de variáveis globais**; (viii) Quem usar `goto` zera a matéria =].
DICA: Consulte o exemplo de código no moodle

8 Considerações Finais

8.1 Dicas

Registros O tipo dos campos dos registros pode variar de acordo com o teste, e pode ser *int*, *double* ou uma string de até 30 caracteres. Uma boa forma de implementá-lo é ler e escrever tudo como strings e apenas converter o campo índice quando necessário.

Gravar e ler nodos do disco Para que seja possível salvar um nodo em disco, o programa deve converter a *struct* que representa e armazena os dados do nodo em uma sequência de bytes. Esse processo de conversão é chamado de serialização. Para reconstruir o *struct* do nodo, com as mesmas informações de antes, o processo inverso deve ser feito, ou seja, deve ser feita uma desserialização. Por exemplo, considere o nodo abaixo:

```
struct Leaf{
  // numero de chaves armazenadas
  int key_num_;
  // posicao no arquivo para a folha a esquerda
  long long unsigned next_;
  // vetor de chaves
  Vector* ls_keys_;
};
typedef struct Leaf Leaf;
```

Esse *struct* tem dois tipos de dados: (i) primitivos, como por exemplo o `int` e o `long long unsigned` que podem ser codificados em bytes diretamente; (ii) compostos, como por exemplo o tipo `Vector`, que assim como o próprio nodo, tem que ter cada um de seus membros serializados individualmente. Para poder serializar esta estrutura, podemos utilizar o seguinte código:

```
void LeafSerialize (Leaf* l, FILE* f){
  // crie um buffer para armazenar todas variaveis primitivas
```

```
  // de forma sequencial. O tipo do buffer vai ser o maior dos
  // tipos primitivos envolvido. Nesse caso: long long
  long long unsigned metadata [2];

  metadata [0] = l->key_num_;
  metadata [1] = l->next_;

  // o arquivo f tem que ser um arquivo BINARIO. Escreva o
  // buffer preenchido no arquivo
  fwrite (&(metadata [0]), sizeof(long long unsigned), 2, f);

  // chame o metodo de serializacao do vetor
  VectorSerialize (l->ls_keys_, f);
}
```

Para deserializar o nodo, precisamos de duas coisas: (i) **Saber onde (a partir de qual byte) o nodo foi escrito no arquivo**; (ii) Ler os bytes na mesma ordem em que foram escritos e saber o que cada um significa. Por exemplo, para desserializar o nodo:

```
void LeafDeserialize (Leaf* l, FILE* f){
  // use o mesmo tipo de vetor utilizado na serializacao
  long long unsigned metadata [2];

  // IMPORTANTE: o ponteiro de acesso do arquivo ja
  // deve estar apontando para o inicio da sequencia
  // de bytes(serializacao) no arquivo, que representa o
  // nodo que vc quer desserializar. Use fseek
  fread (&(metadata [0]), sizeof(long long unsigned), 2, f);

  // Decodifica os bytes lidos. Sabemos que os primeiro
  // 8 bytes (long long u) sao o tamanho OBS: downcasting
  // de long long para int
  l->key_num_ = metadata [0];
  // o segundo long long sabemos que e o endereco da proxima
  // folha no arquivo
  l->next_ = metadata [1];

  // chame o metodo de serializacao do vetor.
  // IMPORTANTE: o fread ja move o ponteiro de acesso do
  // arquivo. Logo, nesse momento, f ja esta apontando
  // para o inicio da serializacao do vetor.
  // Nao precisa chamar fseek
  VectorDeserialize (l->ls_keys_, f);
}
```

8.2 Outros

Atrasos Trabalhos poderão ser entregues após o prazo estabelecido, porém sujeitos a uma penalização regida pela seguinte fórmula:

$$\Delta_p = \frac{2^{d-1}}{0.32}\,\%$$

Por exemplo, se a nota dada pelo corretor for 70 e você entregou o TP com 4 dias corridos de atraso, sua penalização será de $\Delta_p = 25\%$ e, portanto, a sua nota final será: $N_f = 70 \cdot (1 - \Delta_p) = 52.2$. Note que a penalização é exponencial e 6 dias de atraso resultam em uma penalização de 100%.

Referências

[Bayer and McCreight, 1972] Bayer and McCreight (1972). Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189.

[Cormen, 2009] Cormen, T. H. (2009). *Introduction to algorithms*. MIT press.