UNIVERSIDADE DE SÃO PAULO
INSTITUTO DE CIÊNCIAS MATEMÁTICAS E DE COMPUTAÇÃO – USP
DISCIPLINA DE ORGANIZAÇÃO DE ARQUIVOS (SCC02015)
PROF. DRA. CRISTINA DUTRA DE AGUIAR CIFERRI

ÁRVORE B*
TRABALHO PRÁTICO: PARTE 2

Eduardo Aguiar Pacheco
No. USP 8632455

Primeira Edição
São Carlos – SP
12 de junho de 2016

SUMÁRIO

1. INTRODUÇÃO

Conforme requisitado, foi implementada um Árvore B* (B estrela) como índice da tabela de filmes. O modelo proposto para implementação da árvore era o pseudocódigo implementado pelo Cormen[1], porem, ele apenas implementa o pseudocódigo relacionada a Árvore B – não a estrela. Por este motivo, e pelas dificuldades encontradas no decorrer do projeto, as abordagens são distintas.

Também foi implementada a remoção de elementos da árvore. A funcionalidade não foi requisitada pelo trabalho[2], mas, iludido pela simplicidade aparente à deleção, fui tentado a implementa-la, o que resultou em muitas horas, varias frustrações, mas um belo resultado final.

Segue então, os detalhes da implementação da árvore, contida no TAD BSTREE (*bs-tree.h* e *bs-tree.c)* que visa guiar o leitor ao entendimento do código e a perceber suas peculiaridades.

2. ESTRUTURA DA ÁRVORE

A árvore possui três elementos: o cabeçalho, BSTREE; os nós, BSNODE; e o par chave e *offset* do registro correspondente, PAIR.

2.1 BSTREE (CABEÇALHO)

```
struct tree {
	BSNODE *root; //a raiz soh eh liberada quando o indice eh
fechado
	compare_function_declare(compar); // funcao que compara
chaves,
	FILE* file; //arquivo fica aberto, em modo escrita e leitura,
enquanto o indice estiver aberto

	/***** header - persistent data *****/
	int height; // altura da arvore
	unsigned long count; // quantidade de pares armazenados
	size_t keysize; // tamanho das chaves
	size_t blocksize; // tamanho do bloco
	size_t trashstack; // topo da lista de nohs deletados, para
reaproveitamento de espaco
	int degree; // grau da arvore
	//size: 8*4 + 4*2 = 40
};
```

BSTREE, assim redefinido no *bs-tree.h* por um *typedef*, é a estrutura responsável por armazenar as características da árvore. Alguns de seus dados são armazenados somente em memória, durante a execução: o arquivo; a função para comparar chaves; o ponteiro para a raiz, já carregada em memória; o grau da árvore, que é recalculado, por uma função matemática, sempre que o índice é aberto.

---

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein, Introduction to Algorithms, 3rd edition, MIT Press, 2009

[2] Este projeto é, inicialmente, um trabalho acadêmico, como apresentado na capa.

Uma peculiaridade da árvore é que **o nó raiz possui o dobro do tamanho em disco**, ou seja, dois blocos (páginas de disco). Essa característica permite ter mais chaves em memória, **pois a raiz está, durante toda a aplicação, carregada em memória**, e portanto minimiza o número de acessos a disco, porem, o motivo maior pelo qual foi assim implementada é o problema encontrado na remoção de chaves da árvore, que será melhor explanado na seção que trata desse algoritmo.

Entre as características armazenadas pelo cabeçalho, destaca-se a *keysize* (tamanho das chaves armazenadas), e o *blocksize* (tamanho do bloco de disco). Por serem atributos críticos para a construção da árvore, estes não podem ser alterados uma vez que a árvore foi criada. São parâmetros passados para o TAD somente na função *bstcreate* (*bs-tree.h*), que cria um novo índice. Outro atributo que se encaixa nesse contexto crítico é a função que compara as chaves, mas optei por não salvar essa função na memória, para não ter o trabalho de estudar essa abordagem. Dessa forma, ao abrir o índica, *bstopen*, a aplicação precisa informar, novamente, como se compara as chaves.

Foi dada liberdade ao usuários para armazenar qualquer tipo de chave, com qualquer tamanho, desde que esse seja fixo e caiba na árvore, dado o tamanho de bloco informado.

A variável *trashstack* contém o *offset* da pilha de blocos disponíveis para reaproveitamento. Ou seja, quando um nó é removido da árvore, isso ocorre no processo conhecido como *merge*, que será visto no algoritmo de remoção, seu bloco fica disponível para ser reaproveitado. Não foi implementado um algoritmo para compactar o índice. A inserção de um novo bloco na pilha tem um custo constante de um acesso a disco, e também é constante em processamento na *CPU*, pois se trata de uma pilha (*First In, First Out*).

2.2 BSNODE (NÓ)

```
typedef struct node {
	bool changed;	// indica se o noh foi alterado e precisa
ser reescrito
	size_t offset;	// offset do noh

	/*** dados persistidos ***/
	unsigned int count; // quantidade de pares no noh
	PAIR **pairs;
	size_t *children; // filhos do noh, podendo ser NULL
	//size: [int] + [(keysize + size_t) * n] + [size_t * (n+1)]
} BSNODE;
```

Os nós da árvore possuem uma variável *bool*[3] que sinaliza se o nó foi alterado, e, portanto, precisa ser reescrito. **A reescrita ocorre somente quando o nó é liberado da memória** através da função *_bsnode_free*[4].

Os nós armazenam, somente em memória, o seu offset. As demais informações são armazenadas em disco, que são:

[3] O tipo booleano não é nativo em C, então foi simulado por um *typedef* que redefine o tipo *char*, que ocupa um byte, para *bool*.

[4] A nomenclatura de funções assume que, as funções internas (privadas) aos TAD começam com _ (underline).

- Número de pares armazenados no nó;
- Os pares, chave e *offset*, presentes no nó;
- *Offset* dos nós filhos, se houver.

O vetor de pares é, na verdade, um vetor de ponteiros. Essa decisão foi tomada para facilitar a manipulação desses pares e, assim, evitar operações que envolvam movimentar muitos bytes, como seria um atribuição com *struct*, além de que envolveria um maior cuidado com o conteúdo sendo copiado, por tornar o processo menos intuitivo.

O nó não possui uma variável indicando se ele é folha ou não. Essa verificação é feita pelo próprio ponteiro de filhos, *children*: se este for *NULL*, o nó é folha. Os pseudocódigos que serão apresentados nesse documento, por terem caráter instrutivo, trabalham com uma suposta variável *isleaf*, por facilitar o entendimento.

2.3 PAIR (PAR CHAVE/OFFSET)

```
typedef struct pair {
        void *key;
        size_t recoffset; //offset do registro referente aa chave
} PAIR;
```

A estrutura *PAIR* é uma representação em memória do par chave/*offset* que foi, na integra, armazenado em disco. As operações de leitura e escrita do nó (*_bsnode_read*; *_bsnode_*write) são responsáveis por transformar essa representação em memória em bytes escritos no disco, e vice versa. Ou seja, armazenar a chave em memoria, criar o *PAIR*, apontar para ela, e então armazenar o *offset* no *PAIR*, e vice-versa.

Vale a pena citar que a memória alocada para a chave é sempre exclusiva ao TAD. Isso para que a aplicação tenha controle e liberdade quanto ao seu espaço reservado em memória. Implemento esse comportamento na rotina de inserção – copio a chave sendo inserida, e mantenho referencia a copia, permitindo que a original seja liberada sem afetar o comportamento do TAD. É importante notar que os bytes contidos na chave precisam representar a chave na integra, pois, não foi implementado nenhum suporte para escrita recursiva.

3. ALGORITMOS

A árvore-B* possui três funcionalidades básicas. Essas são:

- Inserção
- Remoção
- Busca

3.1 INSERÇÃO

```
//insere um novo par chave/offset na arvore
//parametros
//     t               arvore
//     key        chave a ser inserida
//                     essa chave sera escrita na integra no
disco.
//                     portanto, não deve referenciar endereços
de memória.
//     recoffset  offset do registro referenciado
//retorno
//     se foi inserida com sucesso, retorna a quantidade de
elementos na arvore
//     senao, false
bool bstinsert (BSTREE *t, void *key, size_t recoffset);
```

O algoritmo de inserção é implementado pela rotina *bstinsert* e possui varias sub-rotinas, internas ao TAD. Uma peculiaridade é que seu tratamento de overflow acontece *down-top*, ou seja, depois de inserido. Optei por essa abordagem pois me garante um número preciso de elementos dentro do nó no momento da redistribuição, ou do *split*. A inserção faz uso do tamanho duplicado do nó raiz, o que traz benefícios aquém ao motivo de sua implementação, que será esclarecido na remoção.

Segue um pseudocódigo que demonstra seu comportamento:

```
BST_INSERT (t, key)
1.  seInseriu = _BST_INSERT (t, t.root, key)
2.  if (seInseriu and t.root.n > 2*(t.degree - 1)) // raiz  tem o dobro de espaco
3.      _BST_SPLITROOT (t)

_BST_INSERT (t, node, key)
1.  //busca o ponto de inserção, ou encontra o valor
2.  for (i = 0; i < node.n and key > node.key[i]; i = i+1);
3.  if (i < node.n and key == node.key[i])
4.      return false;
5.  if (node.isleaf == false)
6.      ret = _BST_INSERT (t, node.child[i], key)
7.      if (node.child[i].n > t.degree – 1)
8.              _BST_INSERT_REARRANGE (t, node, i);
9.      return ret;
10. //para inserir, empurra para a direita as chaves presentes, a partir do ponto i
11. SHIFTRIGHT (node.key, i);
12. node.key[i] = key;
13. node.n = node.n+1;
14. return true;

_BST_INSERT_REARRANGE (t, node, i)
1.  if (i > 0 and node.child[i-1].n < t.degree - 1)
2.      //redistribuição normal, como na arvore B, entre dois nos, envolve descer a
    chave separadora do pai, e subir uma cabível. Além de redistribuir o total de
    elementos de forma igualitária entre os nós.
```

```
3.      _BST_REDISTRIBUTE_TOLEFT (t, node, i);
4.      return;
5.  if (i < node.n and node.child[i].n < t.degree – 1)
6.      _BST_REDISTRIBUTE_TORIGHT (t, node, i);
7.
8.  //ao chegar nessa parte, garanto que que os dois possíveis irmãos estão, também,
    lotados.
9.  // na implementação eu sorteio entre as duas possíveis duplas de filhos, aqui será
    simplificado
10. //o split de dois para três consiste em escolher um irmão imediatos, com (t.degree –
    1) filhos, ou seja, lotado, e então criar um terceiro irmão e redistribuir as chaves
    entre eles.
11. //esse processo também envolve a promoção de mais um chave para o pai
12. // 2*(t.degree-1) +1(overflow) +1(chave separadora) são redistribuídos
13. // 3*(2*(t.degree-1)/3) [mínimo] + 2(chaves separadoras)
14. //garanto que todos terão, no mínimo, 2*(t.degree-1)/3 chaves
15. if (i > 0)
16.     _BST_SPLITCHILD_2TO3 (t, node, i – 1);
17. else
18.     _BST_SPLITCHILD_2TO3 (t, node, i);
```

3.2 BUSCA

```
//busca por um elemento atraves da chave
//parametros
//      t               arvore
//      key         chave de busca
//retorno
//      offset correspondente aa chave, se encontrar
//      senao, ULONG_MAX (limits.h)
size_t bstsearch (BSTREE *t, void *key);
```

O algoritmo de busca usufrui da propriedade das árvores B*, onde a sub-arvore à esquerda só possui chaves menores que a chave separadora, e a sub-arvore à direta só possui chaves maiores.

```
BST_SEARCH (t, key)
1.  return _BST_SEARCH (t, t.root, key);

_BST_SEARCH (t, node, key)
1.  //busca o ponto de inserção, ou encontra o valor
2.  for (i = 0; i < node.n and key > node.key[i]; i = i+1);
3.  if (i < node.n and key == node.key[i])
4.      return node.recoffset[i]; // esse atributo não apareceu antes para simplificar
5.  if (node.isleaf == true)
6.      return -1; //ULONG_MAX
7.  return _BST_SEARCH (t, node.child[i], key);
```

3.3 REMOÇÃO

```
//deleta um elementos da arvore
//parametros
//     t               arvore
//     key             chave de busca do elemento
//retorno
//     offset referente a chave excluida, se encontrou
//     senao, ULONG_MAX (limits.h)
size_t bstdelete (BSTREE *t, void *key);
```

A remoção pode não ser tão intuitiva, pois apresenta algumas complicações não evidentes, os pseudocódigos, porem, apresentam de maneira sucinta seu comportamento. Creio que a decisão menos intuitiva, que merece um esclarecimento mais explicito, é a de tornar a raiz um nó duplo, ou seja, que suporta o dobro de chaves. Acompanhe o surgimento do problema:

- Quando ocorre *underflow* em um nó qualquer – sua ocupação ficou menor que dois terços do máximo – eu possuo as opção de redistribuir com um irmão, que possua mais que o necessário, ou de fundi-lo (*merge*) com um irmão (ou dois).
- Para fundi-lo com um único irmão, eu precisaria quebrar a propriedade da árvore-B* e esperar que os dois irmão atingissem metade da ocupação máxima (quando a B* exige *underflow* com $^2/_3$), caso contrário, ao fundi-los teríamos um nó com *overflow* ($^4/_3$). Porém, se eu quebrar a propriedade da árvore, perco controle de sua eficiencia.
- Tudo bem, garanto que os dois irmãos mais próximo possuem no máximo a ocupação mínima, e então fundo com eles, passando a ter apenas dois nós, com a ocupação máxima:
$$2\times {}^{2n}/_3 + \left({}^{2n}/_3 - 1\right) + \ 2 = 2n + 1$$
onde, n é a ocupação máxima, $2\times grau - 1$. O último número ao final de cada lado da igualdade é o número de chaves separatórias.
- Essa foi a solução optada, porem, para ela funcionar preciso garantir que há, pelo menos, dois irmãos. Como o grau mínimo de uma árvore é quatro, a ocupação mínima para o grau mínimo é dois, então isso não é um problema para os nós intermediários, mas é, sim, um problema para os filhos do nó raiz, **pois a raiz não obedece essa regra** – pense que você pode ter somente um elemento na árvore, a raiz precisa comportá-lo.
- A solução, então, foi tratar esse único caso aparte, onde o *underflow* aconteceu nos filhos da raiz e ela só possui uma chave, e então fundi-los, para se tornarem a nova raiz. Mas, como eles possuem $^{2n}/_3$ eu obtenho um novo nó com *overflow*. Então, permito, com exclusividade, que o nó raiz comporte o dobro de chaves, assim $^{4n}/_3 - 1$ não causa *oveflow*.

Graças a Deus por essa solução!

Vamos aos pseudocódigos, que buscam apresentar de uma forma clara como ficou o resultado final.

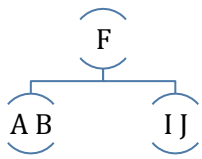
```
BST_DELETE (t, key)
1.  return _BST_DELETE (t, t.root, key);

_BST_DELETE (t, node, key)
1.  integer recoffset = -1; //ou ULONG_MAX
2.  //busca o ponto de inserção, ou encontra o valor
3.  for (i = 0; i < node.n and key > node.key[i]; i = i+1);
4.  if (i < node.n and key == node.key[i])
5.      if (node.isleaf == true)
6.          return _BST_DELETE_ATLEAF (t, node, i); // shift das chaves posteriores, para sobrepô-la e diminuir contador de chaves
7.      PAIR keyAndOffset = _BST_DELETE_SMALLEST (t, node.children[i+1]); // na implementação real eu sorteio entre o predecessor e o sucessor.
8.      recoffset = node.recoffset[i];
9.      node.recoffset[i] = keyAndOffset.offset;
10.     node.key[i] = keyAndOffset.key;
11. else
12.     if (node.isleaf == true)
13.         return recoffset;
14.     recoffset = _BST_DELETE (t, node.child[i], key);
15.
16. if (node.n < 2*(t.degree – 1)/3)
17.     _BST_DELETE_REARRANGE (t, node, i);
18. return recoffset;

_BST_DELETE_REARRANGE (t, node, i)
1.  if (i > 0)
2.      if (node.child[i – 1].n > 2*(t.degree – 1)/3)
3.          _BST_REDISTRIBUTE_TORIGHT (t, node, i - 1);
4.          return;
5.  if (i < node.n)
6.      if (node.child[i + 1].n > 2*(t.degree -1)/3)
7.          _BST_REDISTRIBUTE_TOLEFT (t, node, i);
8.          return;
9.  if (node.n == 1) //só ocorre se for a raiz
10.     _BST_MERGEROOT (t); //o merge da raiz é peculiar, mas a ideia do algoritmo é a mesma
11. else if (i == 0) //para fazer o merge, antes me certifico que tenho três filhos "vazios"
12.     if (node.child[i + 2].n > 2*(t.degree -1)/3)
13.         //busco excedente no segundo irmão – o irmão não imediato
14.         _BST_REDISTRIBUTE_DOUBLELEFT (t, node, i);
15.     else
16.         _BST_MERGE_3TO2 (t, node, i);
17. else if (i == node.n)
18.     if (node.child[i – 2].n > 2*(t.degree -1)/3)
19.         //busca excedente no segundo irmão à esquerda
20.         _BST_REDISTRIBUTE_DOUBLERIGHT (t, node, i – 2);
21.     else
22.         _BST_MERGE_3TO2 (t, node, i – 2);
23. else
24.     //tenho os dois irmãos imediatos, e estão vazios, tudo ok
25.     _BST_MERGE_3TO2 (t, node, i – 1);
```

3.4 ALGORITMOS SUPLEMENTARES

```
//funcao que recebe a chave, o offset do registro referente, e uma
lista de argumentos
//esta lista sao os parametros inviados pela aplicacao, inalterados,
//veja __bstree_debug_dfs
//veja __bstree_debug_bfs
#define runkey_type void(*)(void*,size_t,va_list)
#define runkey_declare(f) void(*f)(void*,size_t,va_list)

//funcao que imprime um chave, em formato texto, na saida padrao
//veja __bstree_debug_printtree
#define printkey_type void(*)(void*)
#define printkey_declare(f) void(*f)(void*)

//percorre todos as chaves da arvore por uma dfs
//parametros
//     arvore
//     funcao para acesso em preordem; pode ser NULL
//     funcao para acesso em inordem; pode ser NULL
//     funcao para acesso em posordem; pode ser NULL
//     parametros para serem repassados aas funcoes de acesso
void __bstree_debug_dfs (BSTREE*, runkey_type, runkey_type,
runkey_type, ...);

//imprime, em modo texto, na saida padrao, o seguinte esquema:
//entre cada chave de um mesmo no ha um espaco ' ';
//ao final de um noh ha um quebra de linha '\n';
//ao final de um nivel da arvore a uma segunda quebra de linha '\n',
//gerando uma linha inteiramante branca entre dois niveis.
//exemplo (os comentarios "--" nao estao presentes)
//F         -- RAIZ
//
//A B       -- FILHO 0 DA RAIZ
//I J       -- FILHO 1 DA RAIZ
//parametros
//     arvore
//     funcao que imprime, em modo texto, na saida padrao, a
chave//parametros
//     arvore
//     funcao que imprime, em modo texto, na saida padrao, a chave
void __bstree_debug_printtree (BSTREE*, printkey_type);
```

Duas outras funções foram acrescentadas ao TAD para facilitar a depuração de seu comportamento: uma *Depth-First Search* (DFS) e um *Breadth-First Search* (BFS), nomeado como *printtree*.

A DFS da liberdade à aplicação para atravessar a árvore tanto em pré-, in- ou pós-ordem, e então processar as chaves de qualquer forma, obedecendo a interface do método. As duas primeiras travessias funcionam de forma intuitiva, já a última, pós-ordem, talvez mereça um esclarecimento: todas as chaves de um nó serão acessadas, em ordem e imediatamente após a anterior, uma vez que todas as sub-arvores filhas deste nó foram recursivamente acessadas. Ou seja, dada a árvore:



A sequencia pré-ordem: F, A, B, I, J
A sequencia in-ordem: A, B, F, I, J

A sequencia pós-ordem: A, B, I, J, F

O BFS foi implementado de maneira engessada, e, por isso, não recebe o nome de BFS, mas sim de *printtree*. Ou seja, não dá liberdade de escolha para a aplicação em como usufruir do BFS. Tomei essa decisão para facilitar o uso do método, pois, para o uso inicialmente proposto, exige muitas informações, como o nível, na árvore, da chave sendo processada. Dessa forma, para a mesma árvore proposta acima, a única saída possível é:

OUTPUT (saída padrão)

```
F

A B
I J
```