Exercícios Comentados
Árvores B (*B-tree*)

Waldemar Celes
Departamento de Informática, PUC-Rio

16 de Abril de 2014

Neste exercício, vamos considerar a representação de árvores B (*B-tree*). Árvores B são eficientes estruturas de busca, sendo também apropriada para armazenamento em memória secundária. A árvore B de ordem m, denotada por *árvore* $(m-1)$-m, possui nós com até m sub-árvores filhas $(p_0, p_1, \ldots, p_{m-1})$ e com até $(m-1)$ chaves $(k_0, k_1, \ldots, k_{m-2})$. Os nós da sub-árvore p_i têm informações menores que a chave k_i; e os nós da última sub-árvore (p_{m-1}) têm informações maiores que a última chave (k_{m-2}). Em resumo, a sub-árvore à esquerda de uma chave tem valores menores que a chave e a sub-árvore à direita tem valores maiores, mas menores que a chave seguinte. A figura abaixo ilustra a representação esquemática de um nó de uma árvore B de ordem 5: *árvore 4-5*. É comum o uso de árvores B de ordem superiores, como *árvore 100-101*, etc.

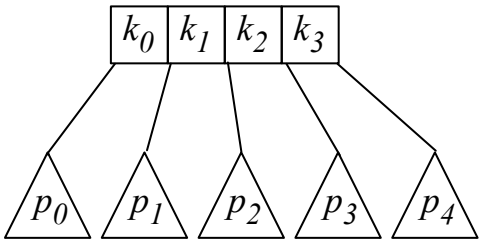

Figura 1: Representação esquemática de um nó da árvore B de ordem 5.

Numa árvore B de ordem m, cada nó interno (com exceção da raiz) tem um número mínimo de sub-árvore filhas igual a $m/2+1$ e, consequentemente, um número mínimo de chaves igual a $m/2$. Assim, uma *árvore 4-5*, cada nó interno tem de 3 a 5 sub-árvores filhas. Além disso, todas as folhas da árvore estão no mesmo nível. Com isso, garante-se que a altura da árvore é $O(\log n)$, onde n é o número de chaves na árvore (numa *árvore 4-5*, no pior caso, todos os nós internos terão "apenas" 3 filhos resultando em $h = \log_3 n$).

Interface

Como se trata de uma estrutura dinâmica para buscas, a interface de programação deve incluir funções para criar e destruir a estrutura, inserir e remover elementos da estrutura e fazer busca de elementos na estrutura. Além disso, para visualizar o conteúdo da estrutura, podemos incluir uma função que imprime as chaves. A função de busca tem como valor de retorno o nó da árvore que contém a chave em questão, fornecendo também a posição (índice) da chave dentro do nó. Assim, na interface, incluímos uma função que retorna o valor da chave dados o nó e a posição. Neste exemplo, a chave é representada por um número inteiro.

```c
#ifndef BTREE_H
#define BTREE_H

typedef struct btree BTree;

BTree* bt_create (void);
void bt_destroy (BTree* a);

BTree* bt_search (BTree* a, int x, int* pos);
int bt_key (BTree* a, int pos);

BTree* bt_insert (BTree* a, int x);
BTree* bt_remove (BTree* a, int x);

void bt_print (BTree* a, int indent);

#endif
```

Representação em C

A representação de um nó da árvore em C é discutida nesta seção. Como veremos, a função de inserção cuidará para que um nó não tenha mais filhos (e mais chaves) do que o permitido pela definição. No entanto, para facilitar a implementação, como será discutido, é comum permitir que um nó tenha um filho (e uma chave) a mais que o permitido, temporariamente. Portanto, iremos dimensionar nosso nó com a capacidade de armazenar um filho e uma chave a mais que o permitido, e usaremos este espaço extra apenas temporariamente na função de inserção.

Nesta nossa implementação, definimos a constante N que define a ordem da árvore. Além dos vetores de chaves e filhos (com as posições extras), armazenamos também junto ao nó o número de chaves efetivamente armazenado.

```c
#define N 5 /* order (an odd number) */

struct btree {
  int n;          /* number of keys */
  int k[N];
  BTree *p[N+1];
};
```

Função de busca

A função de busca recebe a raiz da árvore e a chave a ser buscada, e tem como valor de retorno o nó da árvore onde a chave se encontra, ou NULL se a chave não for encontrada. Caso a chave exista (valor de retorno diferente de NULL), a função também preenche o endereço fornecido com a posição (índice) da chave dentro do nó. Dados o nó e a posição, pode-se recuperar a chave armazenada através da função key, também ilustrada.

Note que um nó folha tem chaves mas não tem filhos. Para identificar se um nó é folha, verificaremos o valor da primeira sub-árvore filha (p [0]); se for NULL, significa que o nó é folha. Usamos então uma função auxiliar interna que verifica se um nó é folha. Ainda, definimos uma função interna que busca a posição de uma chave dentro de um nó específico: se a chave existir dentro do nó, a função tem 1 como valor de retorno e preenche a posição do nó. Se a chave não existir, a função tem 0 como valor de retorno, mas ainda assim preenche a posição, que passa a

indicar a sub-árvore filha onde a chave, se existir na árvore, estará presente. A função de busca usa estas funções auxiliares como ilustrado a seguir.

```
static int isleaf (BTree* a)
{
  return (a->p[0] == NULL);
}

static int findpos (BTree* a, int x, int* pos)
{
  for ((*pos)=0; (*pos)<a->n; ++(*pos))
    if (x == a->k[*pos])
      return 1;
    else if (x < a->k[*pos])
      break;
  return 0;
}

BTree* bt_search (BTree* a, int x, int* pos)
{
  int found = findpos(a,x,pos);
  if (found)
    return a;
  else if (isleaf(a))
    return NULL;
  else
    return bt_search(a->p[*pos],x,pos);
}

int bt_key (BTree* a, int pos)
{
  return a->k[pos];
}
```

Funções para criar e destruir uma árvore

A função para criar uma árvore vazia deve alocar o nó raiz e defini-lo inicialmente com zero chaves. Outra inicialização importante é atribuir NULL à primeira sub-árvore filha, sinalizando que não tem filhos. A função para liberar a memória de uma árvore é convencional, liberando as sub-árvores para então liberar a memória do nó raiz.

```
BTree* bt_create (void)
{
  BTree* a = (BTree*) malloc(sizeof(BTree));
  a->n = 0;
  a->p[0] = NULL;
  return a;
}

void bt_destroy (BTree* a)
{
  int i;
  if (!isleaf(a)) {
    for (i=0; i<=a->n; ++i)
      bt_destroy(a->p[i]);
```

```
  }
  free(a);
}
```

Função para inserção de elementos

A implementação da função de inserção faz uso do espaço extra reservado em cada nó. A inserção de um novo elemento numa árvore B sempre ocorre em uma das folhas da árvore. Com o espaço extra de memória, garantimos que um novo elemento sempre poderá ser inserido em um nó. No entanto, este nó pode ter sua capacidade de armazenamento excedida (a função auxiliar overflow verifica esta condição). Se este for o caso, o nó deve ser subdividido, criando-se um novo nó para o qual transferimos metade das chaves e suas respectivas sub-árvores filhas. Esta subdivisão é feita pela função auxiliar split . Esta função recebe o nó com informações excedentes e faz a divisão criando um novo nó. Para este novo nó são transferidas as $n/2$ maiores chaves com suas respectivas sub-árvores. No nó original, permanece as $n/2$ menores chaves e suas sub-árvores. A chave do meio (posição $n/2$) é um valor que deve ser repassado pela função, tendo como valor de retorno o ponteiro do novo nó criado. A figura abaixo ilustra este operação de subdivisão de um nó.

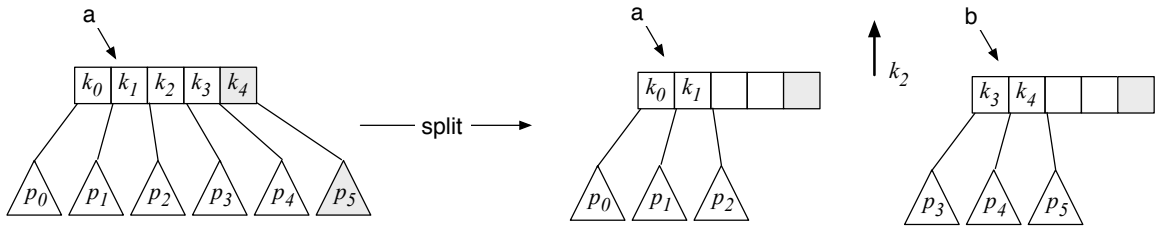

Figura 2: Ilustração de subdivisão de um nó da árvore B de ordem 5.

Uma possível implementacão desta função auxiliar para subdivisão de um nó é mostrada a seguir, junto com a função que verifica se um nó tem informações excedentes.

```
static int overflow (BTree* a)
{
  return (a->n == N);
}

static BTree* split (BTree* a, int* m)
{
  int i;
  BTree* b = bt_create();
  int q = a->n/2;
  b->n = a->n - q - 1;
  a->n = q;
  *m = a->k[q];
  b->p[0] = a->p[q+1];
  for (i=0; i<b->n; ++i) {
    b->k[i] = a->k[q+1+i];
    b->p[i+1] = a->p[q+1+i+1];
  }
  return b;
}
```

Quem deve administrar a subdivisão de um nó com informações excedentes é o nó pai em questão, pois a subdivisão de um nó filho cria um filho novo que deve ser inserido no pai. Naturalmente, será necessário acrescentar uma nova chave em um nó folha (inserção de um novo elemento) ou um par (chave, sub-árvore) num nó interno (resultante de uma operação de subdivisão). Note que para inserir numa folha, só inserimos a chave, e num nó interno, inserimos a chave e um novo filho, que sempre será o filho à direita (maior) que a chave, dada a nossa função de subdivisão. Podemos então escrever uma única função auxiliar que atende a folhas e nós internos.

```
static void addright (BTree* a, int pos, int k, BTree* p)
{
  int j;
  for (j=a->n; j>pos; --j) {
    a->k[j] = a->k[j-1];
    a->p[j+1] = a->p[j];
  }
  a->k[pos] = k;
  a->p[pos+1] = p;
  a->n++;
}
```

Vamos ilustrar agora a função que faz a inserção de um novo elemento na árvore. Inicialmente, vamos considerar uma função interna de inserção, que insere o novo elemento dado um nó da árvore. Usando recursão, se o nó for folha, basta inserir a nova chave no nó e retornar; se o nó não for folha, chamamos recursivamente a função para inserir na sub-árvore filha e então verificamos se a raiz da sub-árvore filha ficou com informações excedentes. Se este for o caso, devemos subdividir este nó raiz, inserindo um novo par chave-sub-árvore, retornados pela função de subdivisão, no nó corrente. A implementação desta função é ilustrada a seguir. Note que, nesta implementação, optamos por aceitar chaves repetidas, inserindo-as na sub-árvore à esquerda.

```
static void insert (BTree* a, int x)
{
  int pos;
  findpos(a,x,&pos);   /* insert even if already exists */
  if (isleaf(a)) {
    addright(a,pos,x,NULL);
  }
  else {
    insert(a->p[pos],x);
    if (overflow(a->p[pos])) {
      int m;
      BTree* b = split(a->p[pos],&m);
      addright(a,pos,m,b);
    }
  }
}
```

Por fim, devemos tratar o caso em que o nó raiz da árvore fica com informações excedentes. Neste caso, devemos subdividir o nó raiz e criar um novo nó raiz que aponta para os nós filhos subdivididos. Neste caso, a raiz da árvore é alterada. Por isso, a função de inserção exportada tem como valor de retorno o (novo) ponteiro da raiz.

```
BTree* bt_insert (BTree* a, int x)
{
```

```
  insert(a,x);
  if (overflow(a)) {
    int m;
    BTree* b = split(a,&m);
    BTree* r = bt_create();
    r->k[0] = m;
    r->p[0] = a;
    r->p[1] = b;
    r->n = 1;
    return r;
  }
  return a;
}
```

Função para imprimir a árvore

Podemos pensar numa função para imprimir as chaves de uma árvore B em forma textual no console. Esta função é útil para validarmos os resultados obtidos. Nesta função o texto com o valor da chave é indentado de forma proporcional à profundidade do nível da árvore. Assim, é possível visualizar a hierarquia no console. A árvore é impressa no sentido decrescente para facilitar a leitura no console.

```
#define INDENT(x) for (int j=0; j<x; ++j) printf(" ");
void bt_print (BTree* a, int indent)
{
  int i;
  if (isleaf(a)) {
    for (i=a->n-1; i>=0; --i) {
      INDENT(indent);
      printf("%d\n",a->k[i]);
    }
  }
  else {
    bt_print(a->p[a->n],indent+2);
    for (i=a->n-1; i>=0; --i) {
      INDENT(indent);
      printf("%d\n",a->k[i]);
      bt_print(a->p[i],indent+2);
    }
  }
}
```

Se inserirmos os valores de 0 a 9, nesta ordem, na árvore, a função imprime a estrutura abaixo, com o valor 3 como raiz.

```
    9
    8
  7
    6
  5
    4
3
    2
  1
    0
```

Um possível programa de teste é apresentado a seguir.

```
#include "btree.h"

#include <stdio.h>

#define N 100

int main (void)
{
  int i;
  int v[N];
  BTree* a = bt_create();
  for (i = 0; i<N; ++i) {
    v[i] = rand();
    a = bt_insert(a,v[i]);
  }
  bt_print(a,0);
  bt_print(a,0);
  printf("0 1 2 3 4 5 6 7 8 9\n");  /* indica o nível das informações no console */
  bt_destroy(a);
  return 0;
}
```

Função para remoção de elementos

Vamos agora discutir a função que retira uma chave da estrutura. Da mesma forma que só podemos inserir elementos nas folhas, também só podemos retirar elementos de folhas. Se o elemento a ser removido já estiver numa folha, a remoção é direta; no entanto, se o elemento que desejamos remover estiver em um nó interno, precisamos transformá-lo num elemento de folha. Para isso, podemos achar o maior elemento (chave) na sub-árvore à esquerda da chave que queremos remover.Trocamos as duas chaves de posição: a chave que queremos remover com a maior chave da sub-árvore à esquerda. Temporariamente, estaremos violando a propriedade de ordenação da estrutura, mas agora podemos remover a chave de interesse, que está numa folha da sub-árvore à esquerda, e então a ordenação da estrutura é restabelecida. A figura abaixo ilustra esta troca de chaves.

Precisamos portanto de uma função para achar a chave máxima de uma sub-árvore:

```
static BTree* findmax (BTree* a)
{
  while (!isleaf(a))
    a = a->p[a->n];    // last child
  return a;
}
```

Recaímos então sempre no caso de remover uma chave de uma folha. No entanto, a folha em questão pode ficar com menos chaves do que é permitido (o número mínimo de chaves em um nó é $m/2$, onde m é a ordem da árvore B). Se este for o caso, temos duas estratégias: (i) se um número de filhos do nó irmão acrescido do número de filhos do nó em questão acrescido de uma unidade não ultrapassar o limite do número de filhos de um nó, podemos concatenar os dois nós, junto com a chave que os separa; (ii) se a soma dos filhos acrescida da chave exceder o número máximo de chaves, significa que este nó irmão tem muitos filhos, e pode ceder um para restabelecer o número mínimo de filhos no nó em questão. Neste caso, a chave que separa os

nós é transferida para o nó em questão e um dos filhos do nó irmão ocupa a posição da chave, de forma a não violar a ordenação da estrutura. Note que a estratégia (i) implica em remover também uma chave (e sub-árvore) do nó pai, podendo se propagar até a raiz, caso todos os nós fiquem com menos chaves do que o permitido.

Podemos então identificar algumas funções auxiliares que devem ser implementadas. Inicialmente, vamos considerar uma função para retirar uma chave (e sua sub-árvore) de um dado nó. O objetivo não é liberar memória, apenas deslocar os elementos dos vetores para preencher o espaço antes ocupado pela chave (e sub-árvore) que não estão mais no nó. No caso de nó folha, precisamos retirar apenas a chave; no entanto, no caso de um nó interno, temos que retirar a chave e a sub-árvore. Aqui, ocorrem duas situações: podemos estar interessados em remover a chave com sua sub-árvore à esquerda ou com sua sub-árvore à direita. Podemos então fazer duas funções distintas, e na folha aplicar qualquer uma das duas (já que não nos interessa as sub-árvores).

```
static void delleft (BTree* a, int pos)
{
  int i;
  for (i=pos; i<a->n-1; ++i) {
    a->k[i] = a->k[i+1];
    a->p[i] = a->p[i+1];
  }
  a->p[i] = a->p[i+1];
  a->n--;
}

static void delright (BTree* a, int pos)
{
  int i;
  for (i=pos; i<a->n-1; ++i) {
    a->k[i] = a->k[i+1];
    a->p[i+1] = a->p[i+2];
  }
  a->n--;
}
```

Similar ao que fizemos na inserção, o nó pai será responsável por verificar se o filho ficou inválido; neste caso, verificar se o filho ficou com menos chaves do que o permitido. Podemos escrever uma função simples para fazer esta verificação.

```
static int underflow (BTree* a)
{
  return (a->n < N/2);
}
```

Se o nó filho ficou em condição inválida, cabe ao nó pai re-arrumar os filhos para restabelecer a consistência da estrutura. Como discutido, temos duas estratégias: redistribuição de chaves (e sub-árvores) ou concatenação de nós, com a liberação de um deles. O primeiro passo é identificar um nó irmão ao nó que ficou inconsistente. Vamos adotar a estratégia de escolher sempre o irmão à direita (a menos que este não existe; neste caso, escolhemos o irmão à esquerda). Temos então sempre um par de nós: o nó à direita de uma chave e o nó à esquerda de uma chave, um deles com menos filhos do que o permitido. Verificamos então se é possível concatenar os nós, isto é, se a soma dos filhos acrescido da chave não excede o limite de filhos permitido para um nó. Se este for o caso, vamos adicionar no nó à esquerda as chaves e sub-árvores do nó à direita, começando com a chave do pai e a primeira sub-árvore do nó da direita. Note então que estamos

sempre inserindo uma chave e sua sub-árvore à direita; logo, podemos usar a função addright já usada na inserção. Em seguida, temos que remover a chave e sua sub-árvore da direita do nó pai que está re-arrumando os filhos. O trecho de código que faz essa concatenação é apresentado a seguir:

```
addright(left,left->n,a->k[pos],right->p[0]);
for (int i=0; i<right->n; ++i)
  addright(left,left->n,right->k[i],right->p[i+1]);
free(right);
delright(a,pos);
```

Se a concatenação não for possível, temos que transferir os filhos de um nó para o outro, sempre "rotacionando" com o valor da chave: a chave é migrada para o nó com poucos filhos e uma chave do outro nó ocupa a posição da chave no nó pai, até que o número de filhos seja o suficiente. Note que essa migração pode ser da esquerda para a direita como da direita para a esquerda; por isso, precisamos incluir uma função addleft que adiciona a chave com sua sub-árvore da esquerda em um nó:

```
static void addleft (BTree* a, int pos, int k, BTree* p)
{
  int j;
  for (j=a->n; j>pos; --j) {
    a->k[j] = a->k[j-1];
    a->p[j+1] = a->p[j];
  }
  a->p[pos+1] = a->p[pos];
  a->k[pos] = k;
  a->p[pos] = p;
  a->n++;
}
```

A função completa que rearruma os filhos de um nó com menos chaves do que o necessário fica então:

```
static void rearrange (BTree* a, int pos)
{
  if (pos==a->n)  /* convert child index into key index */
    pos--;
  BTree* left = a->p[pos];
  BTree* right = a->p[pos+1];
  if (left->n + right->n + 1 >= N) {
    /* redistribute */
    while (underflow(left)) {
      addright(left,left->n,a->k[pos],right->p[0]);
      a->k[pos] = right->k[0];
      delleft(right,0);
    }
    while (underflow(right)) {
      addleft(right,0,a->k[pos],left->p[left->n]);
      a->k[pos] = left->k[left->n-1];
      delright(left,left->n-1);
    }
  }
  else {
    /* concatenate */
    addright(left,left->n,a->k[pos],right->p[0]);
```

```
    for (int i=0; i<right->n; ++i)
      addright(left,left->n,right->k[i],right->p[i+1]);
    free(right);
    delright(a,pos);
  }
}
```

A função de remoção interna é dada por:

```
static int rem (BTree* a, int x)
{
  if (isleaf(a)) {
    int pos;
    if (findpos(a,x,&pos))
      delleft(a,pos);  /* or delright */
    else
      return 0;
  }
  else {
    int pos;
    if (findpos(a,x,&pos)) {
      BTree* l = findmax(a->p[pos]);
      a->k[pos] = l->k[l->n-1];
      l->k[l->n-1] = x;
    }
    if (!rem(a->p[pos],x))
      return 0;
    if (underflow(a->p[pos]))
      rearrange(a,pos);
  }
  return 1;
}
```

E a função para remoção exportada trata do caso em que a raiz fica com apenas um único filho restante:

```
BTree* bt_remove (BTree* a, int x)
{
  if (rem(a,x)) {
    if (a->n == 0 & a->p[0] != NULL) {  /* root with one child */
      BTree* t = a;
      a = a->p[0];
      free(t);
    }
  }
  return a;
}
```

Exercícios propostos

1.

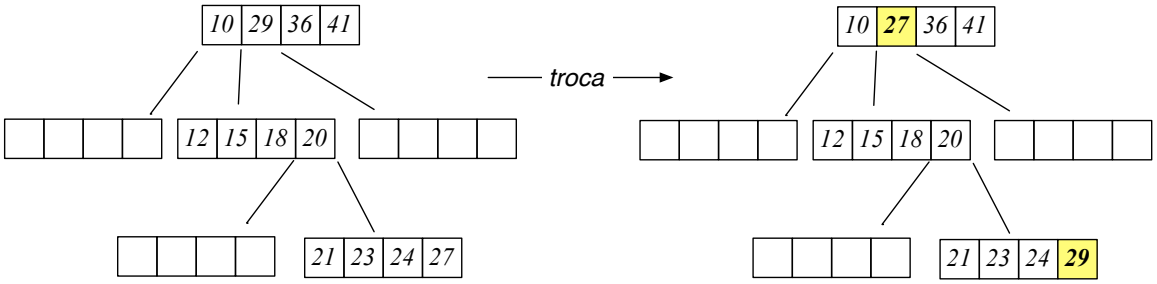

Figura 3: Troca de chaves para transferir para uma folha.