Árvores - Aula 2

Paulo José da Silva e Silva

15 de maio de 2001

1 Rotações

Vamos ver outras operações que também são interessantes para Árvores Binárias de Busca, que, por preguiça, denotaremos daqui por diante por ABB.

Muitas das operações que vamos ver a seguir baseiam-se em duas rotinas simples que permitem a "troca de posição" entre um nó e um de seus filhos mantendo a propriedade de "ordem" que define uma ABB. Essas operações são chamadas de rotações à direita e à esquerda. Vejamos um exemplo:

Na árvore representada na Figura 1 desejamos "trocar de lugar" o nó com chave 20 e seu filho à esquerda, com chave 5. Em outras palavras, queremos que 5 passe a ser pai de 20. Como $20 > 5$, temos que 20 terá que ser filho à direita de 5. O que fazer então com a sub-árvore ligada à direita de 5, para não perdê-la? É fácil, como todos os elementos nessa sub-árvore estão entre 5 e 20 eles podem passar a formar a sub-árvore à esquerda de 20 após a troca. Veja o resultado na Figura 2. Esta operação é chamada de rotação à direita. Analogamente, podemos estar interessados em analisar como trocar um nó com seu filho à direita. A idéia é a mesma com as posições trocadas (direita e esquerda invertidas). Veja um exemplo nas Figuras 3 e 4. Apresentamos a seguir o código para estas duas operações:

```
/* Rotação à direita */
void rot_dir(no **p) {
  no *aux;
  aux = (*p)->esq;
  (*p)->esq = aux->dir;
  aux->dir = *p;
  *p = aux;
}

/* Rotação à esquerda */
void rot_esq(no **p) {
  no *aux;
  aux = (*p)->dir;
  (*p)->dir = aux->esq;
  aux->esq = *p;
```

```
  *p = aux;
}
```

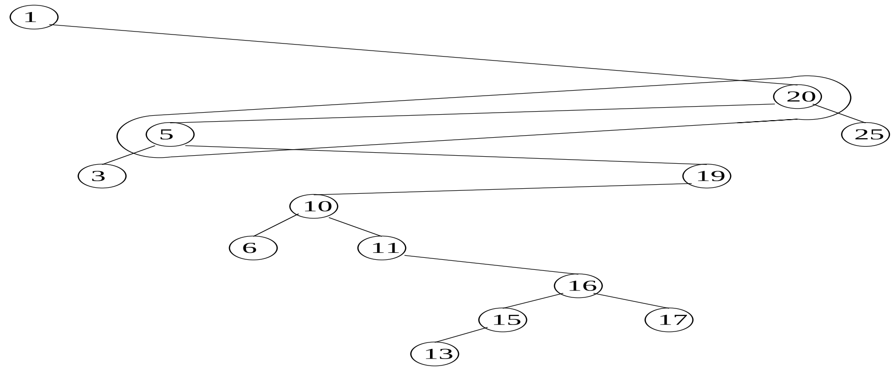

Figura 1: Antes da rotação à direita

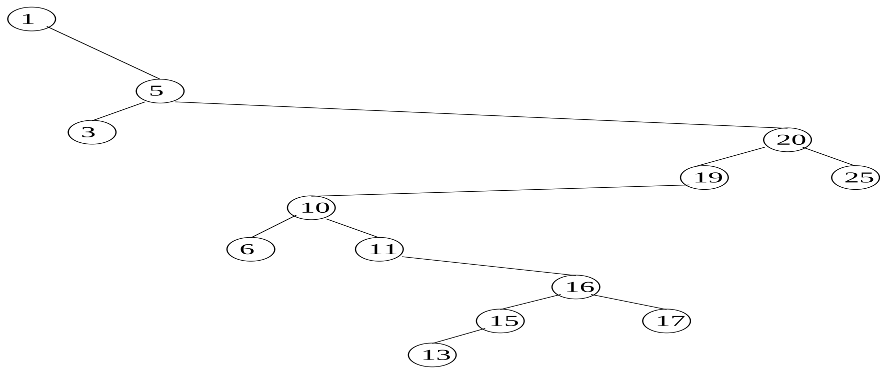

Figura 2: Após da rotação à direita

2 Inserção no topo

Até agora sabemos com inserir e buscar em árvores binárias de busca. Aprendemos também a fazer as tais rotações. Mas para que elas servem? De uma forma geral elas serão peças chaves em novas operações sobre ABB.

Consideremos a seguinte situação hipotética. Nossa ABB está armazenando informações sobre transações no banco de dados comercial de uma empresa. Neste tipo de aplicação, em geral, os dados mais recentes são os mais importantes, pois representam transações que ainda estão se desenvolvendo. Já os dados mais antigos são raramente acessados. Mas por que isso importa?

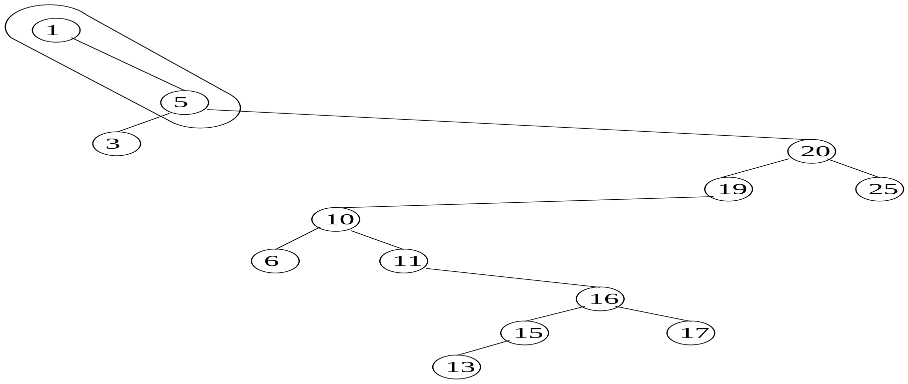

Figura 3: Antes da rotação à esquerda para trocar 1 e 5 de lugar

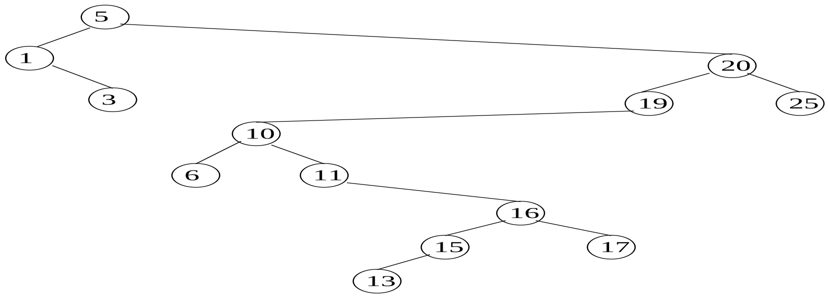

Figura 4: Após da rotação à esquerda

Ora, nossa rotina de inserção de novos nós na ABB sempre insere nós folhas. Isto foi feito simplesmente porque era mais fácil pensar a inserção neste caso. Um efeito colateral é que os nós mais recentes sempre estarão embaixo, sendo então mais "caros" para serem encontrados do que os nós mais antigos que ficam no topo. Isto é exatamente o oposto do que queremos se estivermos no caso descrito no parágrafo anterior. O que precisamos é de uma rotina de que permita a inserção no topo da árvore.

Para isso, vamos pensar recursivamente e usar as rotações. Dada uma nova chave podemos verificar em qual das sub-árvores da raiz ela teria que ser inserida, inserí-la no topo desta sub-arvore e depois fazer uma rotação para trazê-la ao topo da árvore toda. Confuso? Um pouco, mas nestas horas o código fala mais alto:

```
void insere_topo(int x, no **p) {

  /* Se a arvore esta vazia e facil inserir no topo */
  if (*p == NULL) {
```

```
    *p = (no *) malloc(sizeof(no));
    (*p)->chave = x;
    /* Caso houvesse mais dados, copie-os aqui. */
    (*p)->dir = (*p)->esq = NULL;
  }
  else {
    if (x < (*p)->chave) {
      insere_topo(x, &(*p)->esq);
      rot_dir(p);
    }
    else if (x > (*p)->chave) {
      insere_topo(x, &(*p)->dir);
      rot_esq(p);
    }
  }
}
```

O que esta rotina recursiva faz é encontrar o lugar onde o novo nó deveria ser inserido como folha. Em seguida, ela faz uma série de rotações de maneira a trazer o novo nó para o topo. Para aqueles que acham que recurssão não é necessária[1], que tal escrever a rotina acima sem ela?

Por fim, quero destacar que, se as chaves forem inseridas de forma "aleatória", uma ABB criada por inserções no topo também terá altura proporcional ao logaritmo do número de nós.

2.1 Outras operações em ABB

Antes de falarmos em árvores balanceadas vamos apresentar duas novas operações em ABBs.

A primeira é a busca do k-ésimo menor elemento de uma ABB, chamaremos isto de seleção. A rotina para fazer isso é razoavelmente simples. Para buscar o k-ésimo elemento de uma árvore binária de busca começamos verificando o número de elementos na sua sub-árvore direita. Se ele for k então a resposta é a raiz desta sub-árvore. Já se a sub-árvore da esquerda podemos (recursivamente) buscar o k-ésimo elemento nela. Agora se chegarmos à conclusão que ela tem $t < k$ elementos? Então temos que procurar (recursivamente) na árvore da direita, mas não o $k$ésimo e sim o $(k - t - 1)$-ésimo elemento da sub-arvore direita.

O único problema no processo descrito acima é que para que ele seja eficiente é preciso implementar as ABBs com um campo extra que diz quantos elementos tem a sub-árvore associada a cada nó. Isto é simples e fica como exercício (em sala) para vocês. Já a rotina de seleção é apresentada a baixo:

[1]Pois sempre é possível evitá-la. Mesmo pagando o preço de usar explicitamente pilhas, etc.

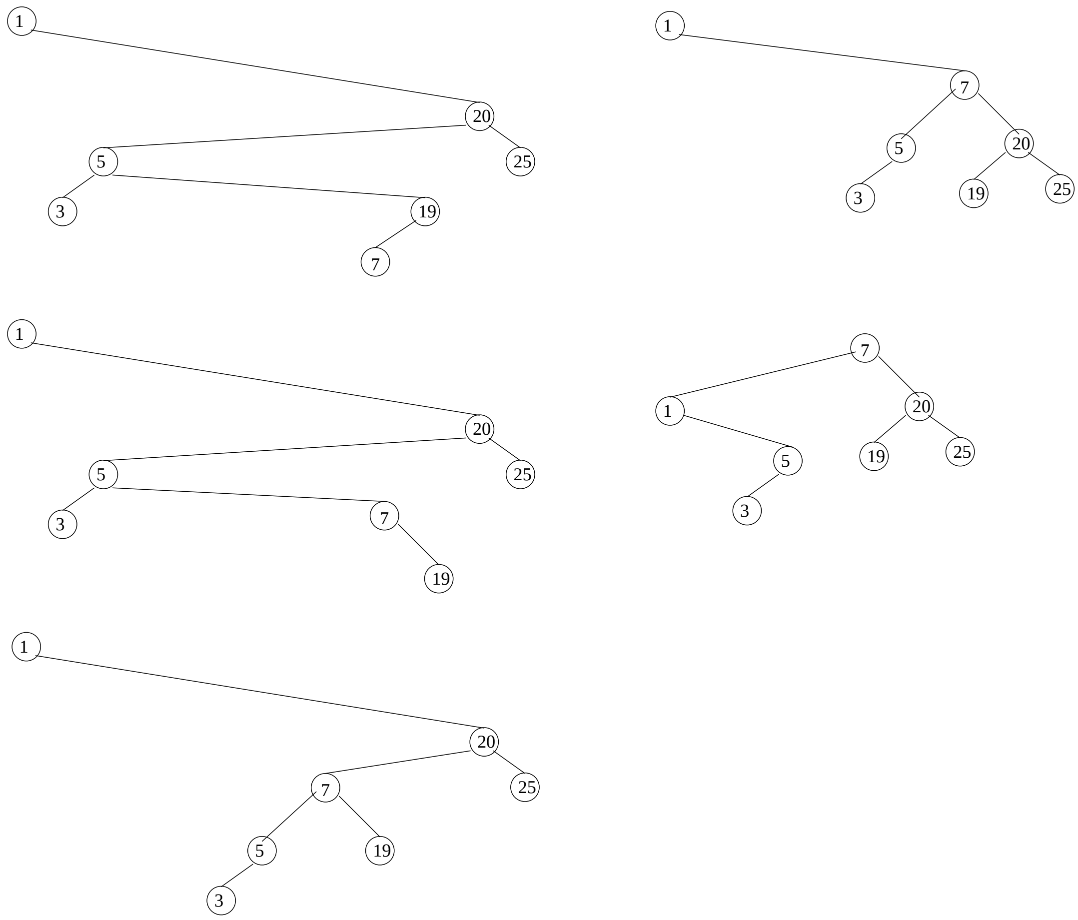

Figura 5: Inserção de 7 no topo

```
/* Consideramos que os nos da arvore possuem um campo extra   */
/* que mantem o numero de elementos das sub-arvores           */
/* associadas a cada no.                                      */
typedef struct _no {
  int chave;
  int n_elem;/* Guarda o numero de elementos nessa sub-arvore */
  /* Outros campos de dados aqui */
  struct _no *esq, *dir;
} no;

no *selecao(int k, no *raiz) {
  int t;
```

```
  /* Pode ser que o k-esimo elemento nao exista. */
  if (raiz->n_elem < k) return NULL;

  t = (raiz->esq == NULL) ? 0 : raiz->esq->n_elem;
  if (t + 1 > k) return selecao(k, raiz->esq);
  if (t + 1 < k) return selecao(k-t-1, raiz->dir);
  return raiz;
}
```

A última operação em ABBs desta aula é a remoção. Infelizmente uma operação tão fundamental é razoavelmente sofisticada e deve ser analisada com calma. Já o uso de rotações ajuda muito.

A idéia é a seguinte: Se o nó que desejamos remover não é a raiz da árvore, nos substituímos (usando recurssão) a sub-arvore onde ele está com o resultado que será obtido pela sua remoção. Já se o nó for a raiz, nós começamos por eliminá-lo. Ficamos então com duas sub-árvores que devem ser unidas. Para isto começamos movendo o menor elemento da sub-árvore direita para o topo usando rotações. Como o nó que trouxemos para o topo da sub-árvore direita é menor, ele não terá elementos ligado a sua sub-árvore esquerda. Ligamos então a sub-árvore esquerda que está sem pai a ele. Vejamos o código:

```
void sobe_menor(no **p) {
  if ((*p)->esq != NULL) {
    sobe_menor(&(*p)->esq);
    rot_dir(p);
  }
}

no *une_dir_esq(no *e, no **d) {

  /* Se a sub-arvore a direita eh vazia, retorna a outra. */
  if (*d == NULL) return e;
  sobe_menor(d);
  (*d)->esq = e;
  return *d;
}

void remove_no(int x, no **p) {
  no *aux;

  /* Verifica se o item esta na arvore. */
  if (*p == NULL) return;

  if (x < (*p)->chave) remove(x, &(*p)->esq);
  if (x > (*p)->chave) remove(x, &(*p)->dir);
```

1	Insere um novo nó com a chave igual a `k`
2	Remove o nó com a chave igual a `k`
3	Faz a seleção do `k`-ésimo menor elemento
4	Sai do programa

Tabela 1: Tabela com operações do exercício 1

```
    if (x == (*p)->chave) {
      aux = *p;
      *p = une_dir_esq((*p)->esq, &(*p)->dir);
      free(aux);
    }
}
```

Existe, porém, um problema em nossa rotina `remove_no`. Como ela sempre escolhe o menor elemento da sub-árvore da esquerda para ser a raiz da sub-árvore, ela acaba quebrando um pouco a aleatoriedade que garante a ABB baixa altura sob condições favoráveis. Pode-se mesmo provar que se a árvore for submetida a uma grande quantidade de inserções e remoções aleatórias a sua altura será proporcional a $\sqrt{n}$ e não mais a $\log(n)$.

Este é um problema intrínseco a estrutura de dados que organizam suas informações para facilitar a busca. Em geral remoções são trabalhosas caso ainda desejemos manter a estrutura "otimizada" pra buscas. Existem algumas técnicas gerais que podem amenizar esse problema sem grandes alterações na remoção. Por exemplo, ao invés de realmente retirar o nó, podemos apenas marcá-lo como inválido, de maneira que ele não seja visto pela busca. Isto é uma boa estratégia se sabemos que o número de remoções será pequeno. Ainda, podemos usar esses nós inválidos em futuras inserções[2] e, por fim, podemos ainda tentar reconstruir toda a estrutura de tempos em tempos usando só inserções e eliminando os nós inválidos.

2.2 Exercícios

1. Faça as adaptações nas rotinas de inserção e remoção para manter cantadores do número nós em cada sub-árvore de modo à seleção funcionar. Implemente então um programa que lê do teclado dois inteiros `op` e `k`. O primeiro inteiro representa uma operação (veja Tabela 1) e outro representa um dado adicional. Após executar cada operação, o programa deve imprimir a árvore atual usando a função `imprime` apresentada na aula anterior.

2. Havia uma outra escolha possível para raiz da sub-árvore obtida a partir união das sub-árvores "filhas" de um nó removido. Qual é esta escolha, re-escreva a rotina para este caso.

[2] Isso é fácil se eles estiverem em folhas.

3. Inspirados pela rotina `sobe_menor` e `selecao` escreva uma rotina que faz com que o k-ésimo menor elemento de uma ABB passe a ser sua raiz. Esta operação é conhecida como *particionamento*.

Referências

[1] Notas de aula da Nami.

[2] Sedgewick, Robert. *Algorithms in C++: Parts 1-4*, 3 edição, Addison Wesley, 1998.

[3] Wirth, Niklaus. *Algoritmos e Estrturas de Dados*, Prentice Hall do Brasil, 1986.