Algoritmos e Estruturas de Dados

Árvores AVL
Implementação

Prof. Nilton Luiz Queiroz Jr.

Árvores AVL

- Para implementar uma árvore AVL de maneira eficiente é necessária a alteração na estrutura usada, se comparada na árvore binária;
  - Se o cálculo da altura de um nó não for realizado em tempo constante, a eficiência do algoritmo, em relação ao tempo de execução, não será a esperada;
- Quais alterações poderiam ser feitas na estrutura dos nós de uma ABB para atender tais requisitos?

Árvores AVL

- Para implementar uma árvore AVL de maneira eficiente é necessário incrementar, no mínimo, um campo altura para armazenar a altura do nó

```
struct tipo_item{
    int chave;
};
struct tipo_no{
    struct tipo_item dado;
    int altura;
    struct tipo_no *esq;
    struct tipo_no *dir;
};
struct tipo_arvore{
    struct tipo_no *raiz;
};
```

Árvores AVL

- Além da alteração na estrutura é interessante o uso de funções auxiliares para:
  - Calcular a altura da árvore
    - Com custo constante;
  - Fazer a rotação para a direita;
  - Fazer a rotação para esquerda;
  - Balancear um nó;
- E também das funções de inserção e remoção;
  - Essas funções devem executar as rotações quando necessário;

Árvores AVL

- Para que a implementação de uma árvore AVL seja eficiente é necessário que o cálculo da altura de um único nó seja feito em tempo constante;
- Como calcular a altura de um nó em tempo constante?

Árvores AVL

- Para que a implementação de uma árvore AVL seja eficiente é necessário que o cálculo da altura de um único nó seja feito em tempo constante;
- Como calcular a altura de um nó em tempo constante?
  - Como cada nó tem sua altura armazenada, basta obter a altura do maior nó filho e somar 1;

```
int altura_no(struct tipo_no *n){
    /*algoritmo*/
}
```

Calcular altura

```
int altura_no(struct tipo_no *n){
    int altura_sad=0, altura_sae=0;
    if(n!=NULL){
        if(n->esq != NULL){
            altura_sae = n->esq->altura;
        }
        if(n->dir != NULL){
            altura_sad = n->dir->altura;
        }
        if (altura_sad>altura_sae)
            return altura_sad+1;
        else
            return altura_sae+1;
    }else{
        return 0;
    }
}
```

Rotações

- Como as rotações mais complexas são uma combinação das duas rotações mais simples, basta implementar as duas rotações simples;
  - Quando for necessário realizar uma rotação mais complexa, basta fazer a chamada das duas rotações simples na ordem correta;

Rotação simples à esquerda

```
void rotaciona_esquerda(struct tipo_no **p){
    /*algoritmo*/
}
```

Rotação simples à esquerda

```
void rotaciona_esquerda(struct tipo_no **p){
    struct tipo_no *q,*aux;
    q=(*p)->dir;
    (*p)->dir = q->esq;
    q->esq = *p;
    aux=*p;
    *p=q;
    q=aux;
    q->altura = altura_no(q);
    (*p)->altura=altura_no(*p);
}
```

Rotação simples à esquerda

```
void rotaciona_direita(struct tipo_no **p){
    /*algoritmo*/
}
```

Rotação simples à esquerda

```
void rotaciona_direita(struct tipo_no **p){
    struct tipo_no *q,*aux;
    q=(*p)->esq;
    (*p)->esq = q->dir;
    q->dir = *p;
    aux=*p;
    *p=q;
    q=aux;
    q->altura = altura_no(q);
    (*p)->altura=altura_no(*p);
}
```

Balanceamento de um nó

- Sendo RSAD o nó raiz da subárvore direita do nó desbalanceado e RSAE o nó raiz da subárvore esquerda do nó desbalanceado temos as seguintes situações de desbalanceamento:
  - Altura(RSAE) > Altura(RSAD)+1 e a subárvore esquerda de RSAE maior que a subárvore direita de RSAE;
  - Altura(RSAD) > Altura(RSAE)+1 e a subárvore direita de RSAD maior que subárvore esquerda de RSAD;
  - Altura(RSAE) > Altura(RSAD)+1 e a subárvore esquerda de RSAD maior que a subárvore direita de RSAD;
  - Altura(RSAD) > Altura(RSAE)+1 e a subárvore direita de RSAE maior que subárvore esquerda de RSAE;

Balanceamento de um nó

```
void balanceia(struct tipo_no **n){
    /*algoritmo*/
}
```

Balanceamento de um nó

```
void balanceia(struct tipo_no **n){
    int asd,ase;
    asd = altura_no((*n)->dir);
    ase = altura_no((*n)->esq);
    if( asd > ase+1){
        if(altura_no((*n)->dir->esq) > altura_no((*n)->dir->dir)){
            rotaciona_direita(&(*n)->dir);
        }
        rotaciona_esquerda(n);
    }else if(ase > asd+1){
        if(altura_no((*n)->esq->dir) > altura_no((*n)->esq->esq)){
            rotaciona_esquerda(&(*n)->esq);
        }
        rotaciona_direita(n);
    }
}
```

Inserção na árvore AVL

- A inserção em uma AVL é feita da mesma maneira que em uma ABB, porém quando se insere um elemento podem ocorrer duas coisas quanto a altura de um nó que faz parte do caminho percorrido até o nó folha que acabou de ser inserido:
  - Sua altura ter aumentado em uma unidade;
  - Sua altura não ter aumentado;
- Dessa forma, é necessário recalcular a altura de todos os nós no caminho da inserção;
- Após o cálculo da altura, é necessário balancear o nó caso ele tenha sido desbalanceado;
- É importante observar que tanto o cálculo quanto o balanceamento são feitos **após** a inserção do nó!

Inserção na árvore AVL

- Para inserir em uma árvore AVL, iremos adotar a seguinte função:

```
int insere(struct tipo_arvore *a, struct tipo_item x){
    return insere_no(&(a->raiz),x);
}
```

- Dessa forma, a questão é como implementar a função insere_no;
  - Essa implementação é muito parecida com a da ABB;

Inserção na árvore AVL

```
int insere_no(struct tipo_no **n, struct tipo_item x){
    int ret;
    if(*n==NULL){
        *n=cria_no(x);
        ret=1;
    }else if( (*n)->dado.chave < x.chave ){
        ret=insere_no(&((*n)->dir),x);
    }else if( (*n)->dado.chave > x.chave){
        ret=insere_no(&((*n)->esq),x);
    }else{
        ret=0;
    }
    (*n)->altura = altura_no(*n);
    balanceia(n);
    return ret;
}
```

Inserção na árvore AVL

```
int insere_no(struct tipo_no **n, struct tipo_item x){
    /*algoritmo*/
}
```

Remoção

- Assim como na inserção, todos os nós que fazem parte do caminho entre o nó raiz e o nó removido devem ser balanceados;
  - Sua altura pode ser reduzida;
- Para implementar a remoção usaremos a seguinte função:

```
int rem(struct tipo_arvore *a, int ch, struct tipo_item *x){
    remove_no(&(a->raiz),ch,x,0);
}
```

- Como implementar a função remove_nó?
  - Assim como a insere_no, essa função é muito parecida com a função de remoção de nó da ABB;

Remoção

```
1.  int remove_no(struct tipo_no **n,int ch, struct tipo_item *x,int flag){
2.      struct tipo_no *aux;
3.      int ret,asd,ase;
4.      if(*n!=NULL){
5.          if(ch > (*n)->dado.chave){
6.              ret=remove_no(&((*n)->dir),ch,x,flag);
7.          }else if(ch < (*n)->dado.chave){
8.              ret=remove_no(&((*n)->esq),ch,x,flag);
9.          }else if(ch == (*n)->dado.chave){
10.             if(!flag){
11.                 *x=(*n)->dado;
12.             }
13.             if(((*n)->esq != NULL) && ((*n)->dir != NULL)){
14.                 aux=maior((*n)->esq);
15.                 (*n)->dado = aux->dado;
16.                 remove_no(&(*n)->esq,aux->dado.chave,x,1);
17.             }else{
```

Remoção

```
17.                 }else{
18.                         aux = *n;
19.                         if((*n)->esq == NULL){
20.                                 *n = ((*n)->dir);
21.                         }else{
22.                                 *n = ((*n)->esq);
23.                         }
24.                         free(aux);
25.                 }
26.                 ret= 1;
27.         }
28.         if(*n!=NULL){
29.                 (*n)->altura = altura_no(*n);
30.                 balanceia(n);
31.         }
32.     }
33.     return ret;
34. }
```

Exercícios

1. Altere a função de inserção na árvore AVL de modo que possa existir inserção de elementos com chaves repetidas;
2. Utilizando a função de inserção do exercício 1, e outras que julgar necessárias, faça um algoritmo que receba um vetor, seu tamanho e retorne o vetor ordenado, utilizando uma árvore AVL.