Algoritmos e Estrutura de Dados

Árvores Binárias de Busca

Baseado no material dos Prof: Wesley Romão

Prof. Nilton Luiz Queiroz Jr.

Remoção

- A remoção em uma ABB não pode alterar as propriedades que a ABB leva em consideração sobre as chaves dos nós;
  - Ou seja, a remoção deve preservar a ordem;
  - Após remover um nó, ainda deve continuar valendo as seguintes afirmações sobre qualquer nó:
    - As chaves de todos os nós da subárvore esquerda são menores que a chave da raiz;
    - As chaves de todos os nós da subárvore direita são maiores que a chave da raiz;

Remoção

- A remoção é a operação mais complexa sobre ABBs;
- Existem 3 casos na remoção em uma ABB:
  - Remoção de um nó folha;
  - Remoção de um nó com apenas um filho;
  - Remoção de um nó com dois filhos;

Remoção

- Remoção de nó folha:
  - É o caso mais simples da remoção em uma árvore binária;
    - Não é necessário tomar decisões quanto as subárvores esquerda e direita do nó que está sendo removido;
  - A única coisa que é necessário fazer na remoção do nó folha é eliminar o nó;

Remoção

- Remoção do nó com apenas um filho:
  - Basta substituir o nó pelo seu filho;
    - Seja ele filho direito ou esquerdo;

Remoção

- Remoção de um nó com dois filhos não vazios;
  - Duas possíveis soluções:
    - Fusão:
      - Busca-se o maior elemento da subárvore esquerda e faz com que o pai do nó removido aponte para ele, e seu filho direito passa a ser a subárvore direita do nó removido;
      - O tamanho da árvore cresce muito;
    - Substituição;
      - Procura-se o menor elemento da subárvore direita (ou o maior da esquerda), o substitui pelo nó e faz uma remoção na folha;

Remoção

- A remoção em uma ABB com n elementos tem complexidade:
  - $O(\log_2 n)$ no melhor caso;
- Para implementar a remoção por substituição é interessante usar um algoritmo para buscar o maior elemento na subárvore esquerda (ou o menor na subárvore direita) e substituí-lo pelo elemento do nó;
- Assim, iremos implementar a função que encontra o maior elemento de uma subárvore e retorna o um ponteiro para esse nó;
- Uma alternativa seria implementar uma função que além de achar o maior elemento ainda o substituísse pelo nó atual e removesse da árvore, o que faria que não fosse necessário “descer” na árvore mais de uma vez;
  - Uma função antecessor;

Antecessor

- Função maior

```
struct tipo_no *maior(struct tipo_no *n){
    /*algoritmo*/
}
```

Antecessor

- Função maior

```
struct tipo_no *maior(struct tipo_no *n){
    if(n->dir != NULL){
        return maior(n->dir);
    }else{
        return n;
    }
}
```

Remoção

- Com a função maior implementada, a remoção pode ser feita da seguinte maneira:
  - Nó folha:
    - Apenas remova o nó e ajuste o ponteiro do seu pai;
  - Nó com somente um filho:
    - Apenas faça o ponteiro de seu pai apontar para o seu filho;
  - Nó com dois filhos:
    - Encontre o maior elemento da sua subárvore esquerda e armazene-o no nó;
    - Remova o elemento encontrado da subárvore esquerda;

Remoção

- Remoção

```
struct tipo_no *remove_no(struct tipo_no *n, int ch){
    /*algoritmo*/
}
```

Remoção

- Remoção

```
struct tipo_no *remove_no(struct tipo_no *n, int ch){
    struct tipo_no *aux;
    if(n!=NULL){
        if(ch < n->dado.chave){
            n->esq = remove_no(n->esq,ch);
        }else if( ch > n->dado.chave){
            n->dir = remove_no(n->dir,ch);
        }else{
            if(n->dir == NULL){
                n=n->esq;
            }else if(n->esq == NULL){
                n=n->dir;
            }else{
                aux=maior(n->esq);
                n->dado = aux->dado;
                n->esq=remove_no(n->esq,aux->dado.chave);
            }
        }
    }
    return n;
}
```

Remoção

- Uma alternativa para a remoção seria retornar se foi ou não possível remover, e retornar do dado removido em uma passagem por referência;
  - Como no caso da remoção de um nó com dois filhos a remoção é chamada recursivamente após a substituição e o dado a ser removido é “alterado”, é importante usar algo para definir se o dado foi ou não alterado;
    - Uma variável que sinalize esse evento;
  - Ou seja, só pode ser feita uma cópia do dado;

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

Remoção

- Remoção com retorno

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

Remoção

```
15.                 }else{
16.                         aux = *n;
17.                         if((*n)->esq == NULL){
18.                                 *n = ((*n)->dir);
19.                         }else{
20.                                 *n = ((*n)->esq);
21.                         }
22.                         free(aux);
23.                 }
24.                 return 1;
25.         }
26.     }else{
27.         return 0;
28.     }
29. }
```

Exercícios

1. Faça uma função que receba um nó de uma árvore binária e retorne sua altura.
2. Faça uma função que receba uma árvore binária de busca e encontre o menor elemento da árvore.
3. Altere a remoção, no caso do nó que tem dois filhos, para que ao invés de substituir o maior elemento da subárvore esquerda seja substituído o menor elemento da subárvore direita.
4. Altere o algoritmo de remoção, no caso do nó com dois filhos, para que seja gerado um número aleatório, 1 ou 0, e a subárvore escolhida para substituição seja a esquerda caso o número gerado seja 1, ou seja a esquerda caso o número gerado seja 0.