Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 14 –Árvore AVL (Remoção e Implementação)

Agenda

- Remoção
- Estrutura de dados
- Implementação da Inserção
- Implementação da Remoção

PS: Parte do conteúdo retirado do material do Prof. Flávio B. Gonzaga

Remoção

- Como a Árvore AVL é uma árvore binária de busca, ela segue as mesmas regras para a remoção de um elemento.

- Existem três cenários (os mesmos discutidos na aula 9):
  i. Remoção de um nó sem filhos (somente remove o nó)
  ii. Remoção de um nó com um filho (substitui o nó pelo filho)
  iii. Remoção de um nó com dois filhos (substitui pelo menor da subárvore direita)

Remoção

- Ao fazer a remoção de uma Árvore AVL usando o mesmo método de remoção de uma árvore binária de busca "tradicional", nós garantimos que as propriedades da árvore binária de busca estarão mantidas.
  - Cada nó possuirá exatamente 0 (zero), 1 (um) ou 2 (dois) filhos e;
  - Dado um nó *v*, todos os nós na subárvore esquerda são menores do que ele e todos os nós na subárvore direita são maiores do que ele.

Remoção

- Apesar das propriedades da árvore binária de busca estarem mantidas, a remoção não garante que as propriedades da Árvore AVL sejam mantidas.
  - Ou seja, após a remoção de uma Árvore AVL, não é possível garantir que o fator de balanceamento de todos os nós continuará com –1, 0 ou +1.
  - Após realizar a remoção de um nó, vamos precisar revisitar nossa árvore binária de busca, atualizando o fator de balanceamento após a remoção.

Remoção

- Depois da remoção, a árvore pode ficar desbalanceada. Portanto, será preciso:

1. Voltar do nó removido até a raiz, atualizando alturas.
2. A cada nó, verificar o fator de balanceamento:
    1. Se for diferente de –1, 0 ou +1, a árvore está desbalanceada.
3. Dependendo do caso, aplicar as rotações apropriadas.

Remoção

- As rotações para a remoção são as mesmas que vimos na aula passada para a inserção. São elas:
  - **RSE** – Rotação Simples à Esquerda (ou Esquerda Simples)
  - **RSD** – Rotação Simples à Direita (ou Direita Simples)
  - **RDE** – Rotação Dupla à Esquerda (ou Esquerda Dupla)
  - **RDD** – Rotação Dupla à Direita (ou Direita Dupla)

Remoção

- Como as operações de rotação são as mesmas, o que muda?
  - O que muda é a regra com que se determina quais os nós estarão envolvidos nas rotações.
  - Partindo-se do nó desbalanceado, escolhe-se a sub-árvore cujo nó possui a maior altura (h).

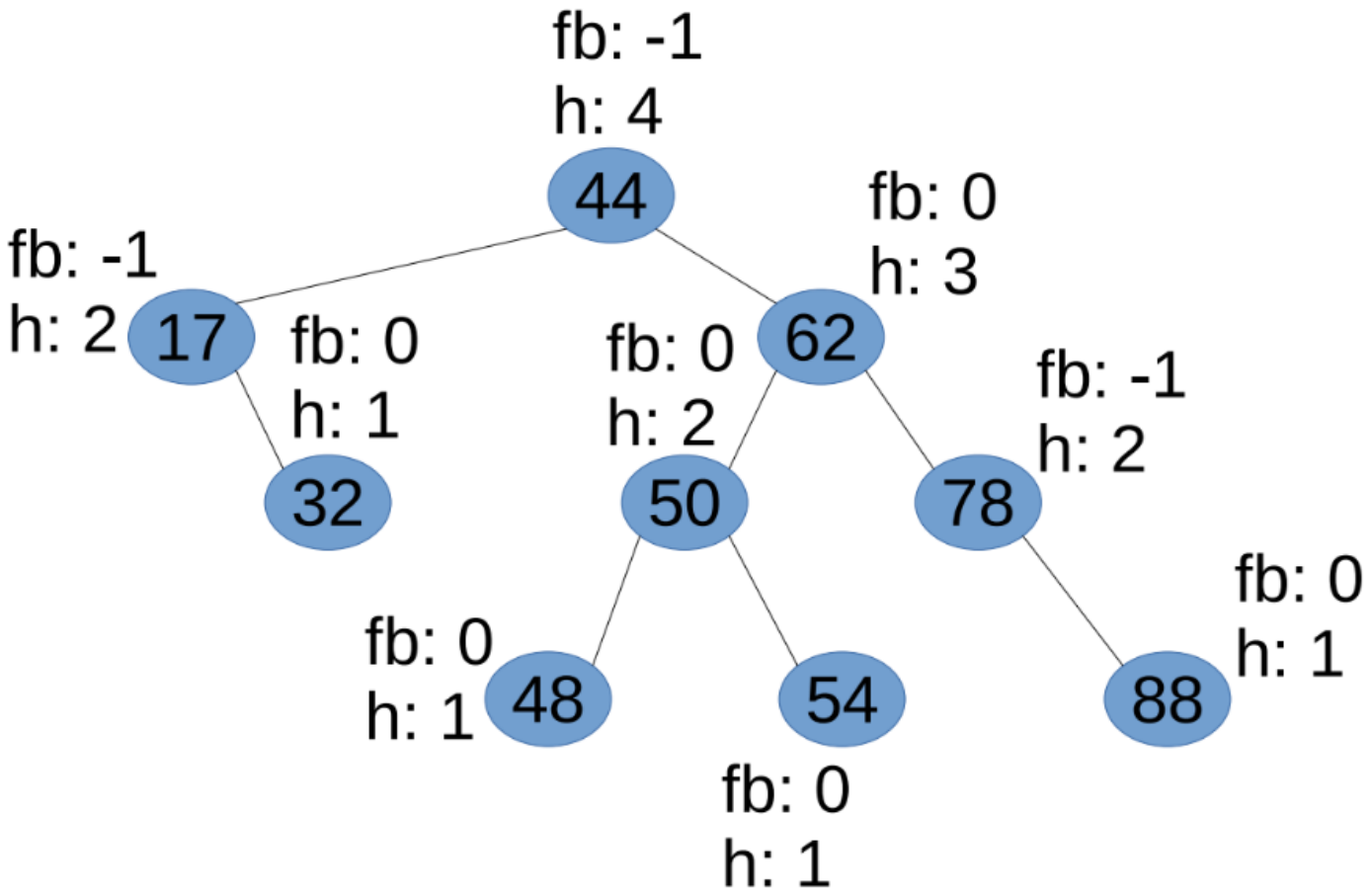
Remoção

- Pode ser necessária a realização de mais de uma operação (considerando rotações simples ou duplas como sendo uma operação), até que a árvore esteja balanceada.

- Para o exemplo de remoção, vamos supor que tenhamos uma Árvore AVL com os seguintes valores:

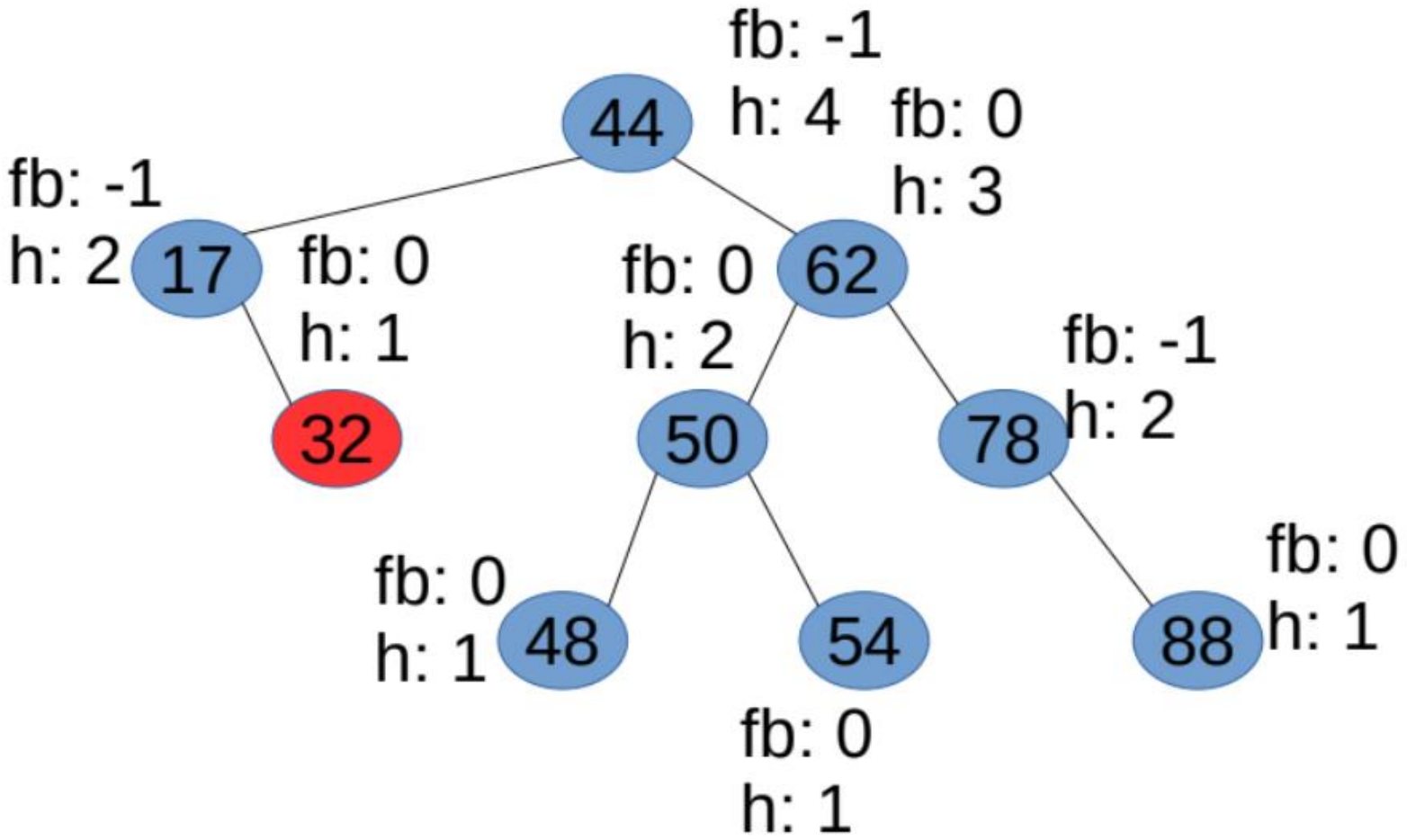
44 – 17 – 62 – 32 – 50 – 78 – 48 – 54 – 88

Remoção

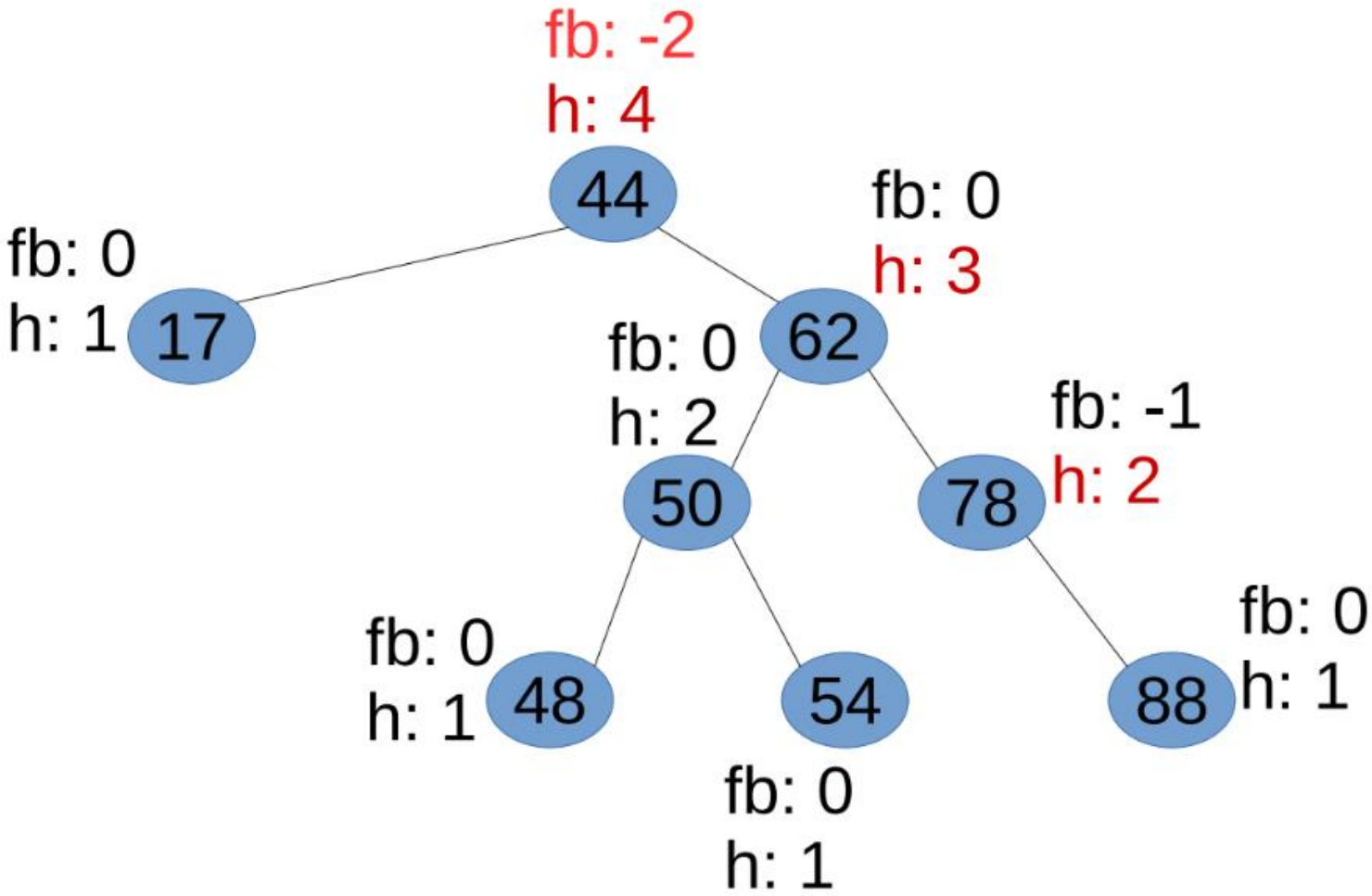


- Para este exemplo, vamos **remover o nó 32**

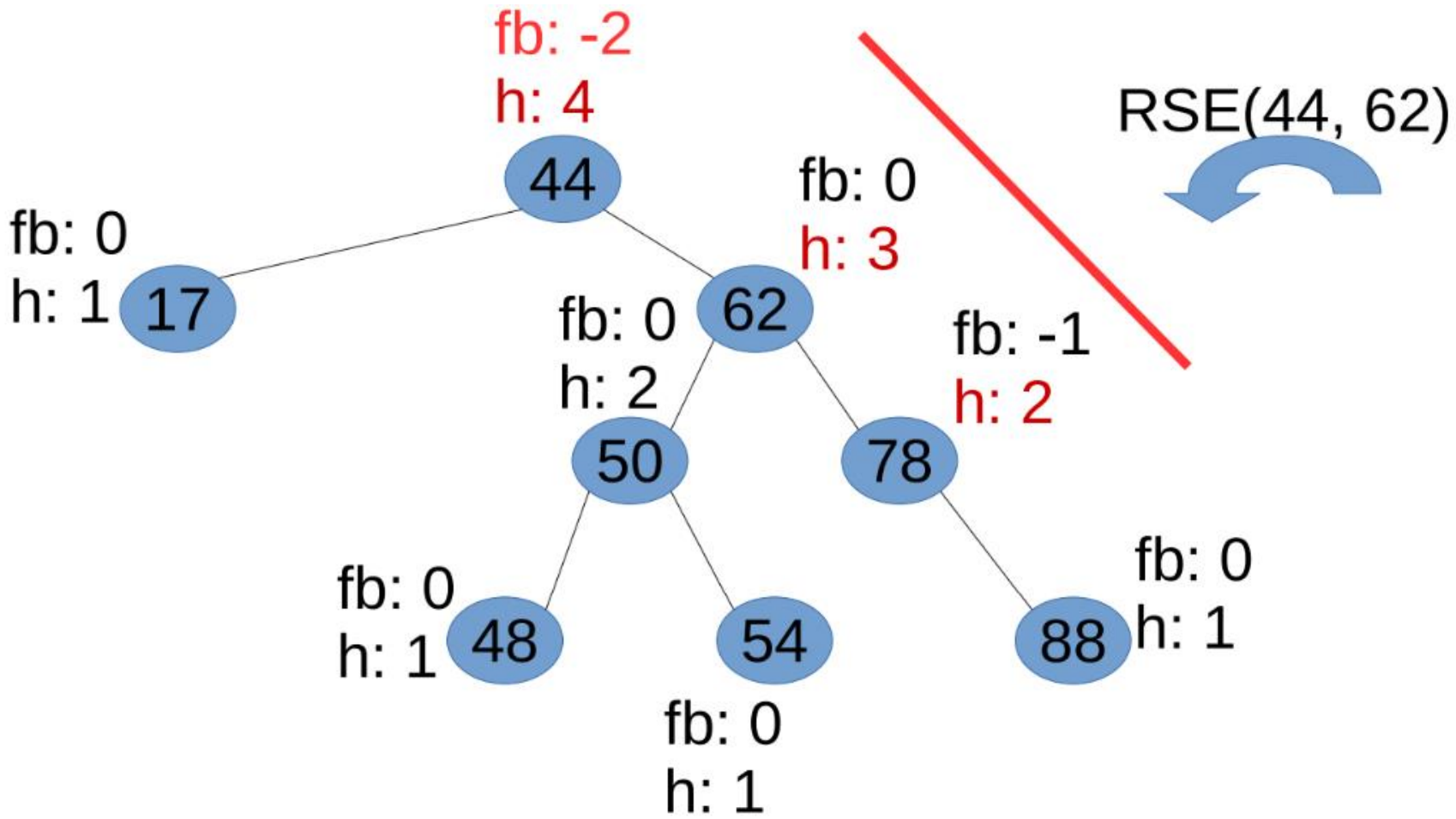
Remoção


fb: -1
h: 4
44
fb: 0
h: 3
fb: -1
h: 2
17
fb: 0
h: 1
fb: 0
h: 2
62
fb: -1
h: 2
32
50
78
fb: 0
h: 1
48
54
88
fb: 0
h: 1
fb: 0
h: 1

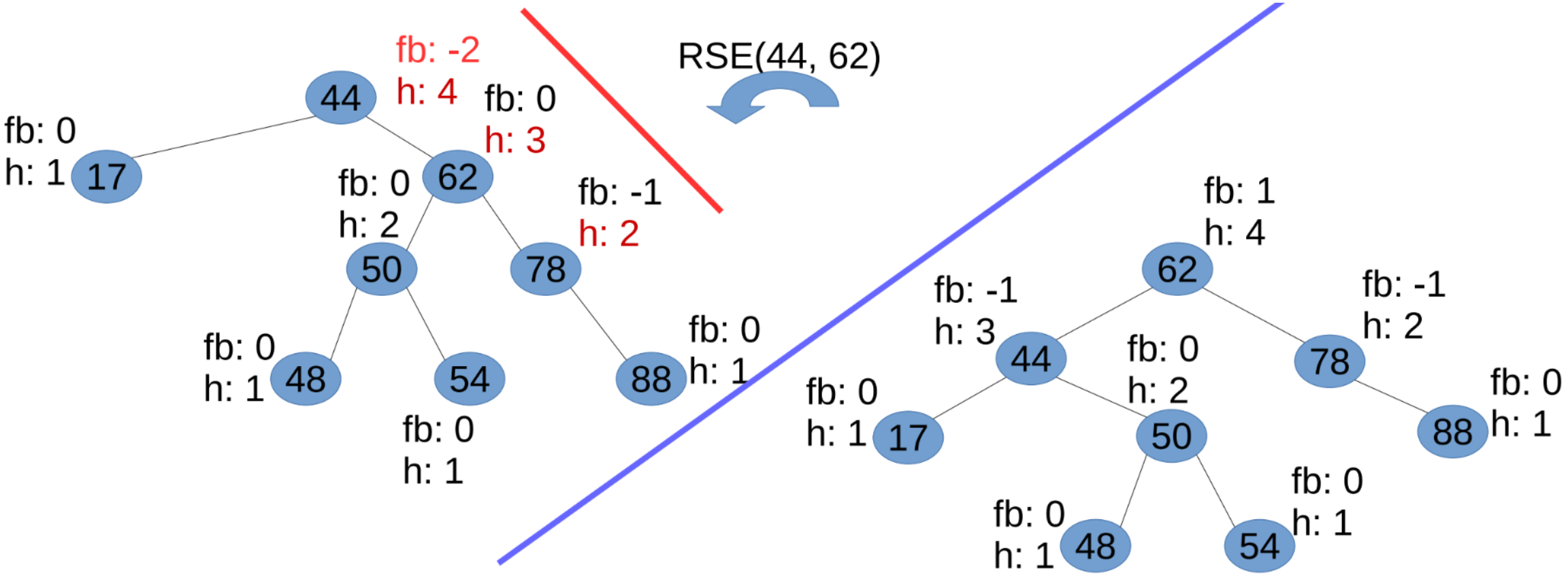
Remoção


fb: -2
h: 4
44
fb: 0
h: 3
fb: 0
h: 1
17
fb: 0
h: 2
62
fb: -1
h: 2
50
78
fb: 0
h: 1
48
54
88
fb: 0
h: 1
fb: 0
h: 1

Remoção


fb: -2
h: 4
44
RSE(44, 62)
fb: 0
h: 1
17
fb: 0
h: 3
62
fb: 0
h: 2
50
fb: -1
h: 2
78
fb: 0
h: 1
48
54
fb: 0
h: 1
88
fb: 0
h: 1

Remoção


fb: -2
h: 4
fb: 0
h: 3
44
fb: 0
h: 1
17
fb: 0
h: 2
62
fb: -1
h: 2
50
78
fb: 0
h: 1
48
54
fb: 0
h: 1
88
fb: 0
h: 1
RSE(44, 62)
fb: 1
h: 4
62
fb: -1
h: 3
44
fb: -1
h: 2
78
fb: 0
h: 2
50
fb: 0
h: 1
17
fb: 0
h: 1
88
fb: 0
h: 1
48
54

Remoção

- Caso queiram testar alguns exemplos de inserção & remoção de elementos em uma Árvore AVL, podem acessar o site:

https://www.cs.usfca.edu/~galles/visualization/AVLtree.html

Estrutura de Dados

- Vamos utilizar a mesma estrutura de dados da árvore binária de busca. A diferença é que vamos acrescentar mais uma informação: O fator de balanceamento (do tipo inteiro).

```
typedef struct no {
    int info;
    int fb;
    struct no *esq;
    struct no *dir;
} Node;
```

Estrutura de Dados

- Além disso, também fornecemos as implementações de cada uma das rotações.

- Todos os algoritmos de rotações utiliza uma função auxiliar, chamada de *fatorBalanceamento* (que faz o cálculo do fator de balanceamento utilizando a altura).
  - A implementação da altura é a mesma da árvore binária de busca!

Estrutura de Dados

```c
int fatorBalanceamento(Node* no) {
    if (no == NULL) {
        return 0;
    }
    return altura(no->esq) - altura(no->dir);
}
```

- O fator de balanceamento é obtido subtraindo a altura da subárvore direita da altura da subárvore esquerda do nó passado como parâmetro.
- Se o ponteiro no for NULL, indicando que o nó não existe, a função retorna 0.
- Caso contrário, retorna a diferença de altura dos nós esquerdo e direito.

Estrutura de Dados

```
Node* rotacaoEsquerda(Node* raiz) {
    Node* temp = raiz->dir;
    raiz->dir = temp->esq;
    temp->esq = raiz;

    raiz->fb = fatorBalanceamento(raiz);
    temp->fb = fatorBalanceamento(temp);

    return temp;
}
```

- A função *rotacaoEsquerda* recebe o nó raiz como argumento e realiza as seguintes operações:
    i. Ela armazena o filho direito de *raiz* no ponteiro *temp*, pois ele se tornará o novo nó *raiz* da subárvore após a rotação.
    ii. Em seguida, o filho esquerdo de *temp* passa a ser o filho direito de *raiz*.

Estrutura de Dados

```
Node* rotacaoEsquerda(Node* raiz) {
    Node* temp = raiz->dir;
    raiz->dir = temp->esq;
    temp->esq = raiz;

    raiz->fb = fatorBalanceamento(raiz);
    temp->fb = fatorBalanceamento(temp);

    return temp;
}
```

- A função *rotacaoEsquerda* recebe o nó raiz como argumento e realiza as seguintes operações (*cont.*):

  iii. *temp* se torna o novo nó raiz, com raiz sendo movida para esquerda de *temp*.

  iv. Após a rotação, os fatores de balanceamento (fb) dos nós *raiz* e *temp* são recalculados, garantindo que a árvore AVL continue balanceada.

Estrutura de Dados

```
Node* rotacaoDireita(Node* raiz) {
    Node* temp = raiz->esq;
    raiz->esq = temp->dir;
    temp->dir = raiz;

    raiz->fb = fatorBalanceamento(raiz);
    temp->fb = fatorBalanceamento(temp);

    return temp;
}
```

- A função *rotacaoDireita* recebe o nó raiz como argumento e executa os seguintes passos:
    i. O filho esquerdo de raiz é armazenado no ponteiro *temp*, pois ele se tornará o novo nó raiz da subárvore após a rotação.
    ii. O filho direito de *temp* passa a ser o filho esquerdo de raiz.

Estrutura de Dados

```
Node* rotacaoDireita(Node* raiz) {
    Node* temp = raiz->esq;
    raiz->esq = temp->dir;
    temp->dir = raiz;

    raiz->fb = fatorBalanceamento(raiz);
    temp->fb = fatorBalanceamento(temp);

    return temp;
}
```

- A função *rotacaoDireita* recebe o nó raiz como argumento e executa os seguintes passos (*cont.*):

  iii. *temp* se torna o novo nó raiz, sendo movida para o lado direito de *temp*.

  iv. Após a rotação, os fatores de balanceamento (fb) dos nós raiz e *temp* são recalculados, garantindo que a árvore AVL continue balanceada.

Estrutura de Dados

```
Node* rotacaoDuplaDireita(Node* raiz) {
    raiz->esq = rotacaoEsquerda(raiz->esq);
    return rotacaoDireita(raiz);
}

Node* rotacaoDuplaEsquerda(Node* raiz) {
    raiz->dir = rotacaoDireita(raiz->dir);
    return rotacaoEsquerda(raiz);
}
```

- A rotação dupla direita nada mais é do que a rotação esquerda seguida da rotação direita.
- De maneira análoga, a rotação dupla esquerda é a rotação direita seguida da rotação esquerda.

Implementando a inserção

- Para a inserção e remoção, vamos utilizar a função auxiliar *balancear*.
- Essa função é responsável por garantir que a árvore AVL permaneça balanceada após a inserção ou remoção de um nó.
- Ela começa recalculando o fator de balanceamento (fb) da raiz, utilizando a função *fatorBalanceamento*. Se fb for maior que 1, isso indica que a subárvore esquerda está "pesada", e a árvore pode precisar de uma rotação.

```
Node* balancear(Node* raiz) {
    raiz->fb = fatorBalanceamento(raiz);

    if (raiz->fb > 1) {
        if (fatorBalanceamento(raiz->esq) >= 0) {
            return rotacaoDireita(raiz);
        } else {
            return rotacaoDuplaDireita(raiz);
        }
    }

    if (raiz->fb < -1) {
        if (fatorBalanceamento(raiz->dir) <= 0) {
            return rotacaoEsquerda(raiz);
        } else {
            return rotacaoDuplaEsquerda(raiz);
        }
    }

    return raiz;
}
```

Implementando a inserção

- Se o fb da subárvore esquerda também for positivo ou zero, uma rotação simples à direita é suficiente.
- Caso contrário, realiza-se uma rotação dupla à direita (uma rotação à esquerda seguida de uma rotação à direita).

```
Node* balancear(Node* raiz) {
    raiz->fb = fatorBalanceamento(raiz);

    if (raiz->fb > 1) {
        if (fatorBalanceamento(raiz->esq) >= 0) {
            return rotacaoDireita(raiz);
        } else {
            return rotacaoDuplaDireita(raiz);
        }
    }

    if (raiz->fb < -1) {
        if (fatorBalanceamento(raiz->dir) <= 0) {
            return rotacaoEsquerda(raiz);
        } else {
            return rotacaoDuplaEsquerda(raiz);
        }
    }

    return raiz;
}
```

Implementando a inserção

- Se o fb for menor que -1, indicando que a subárvore direita está "pesada", e se o fator de balanceamento da subárvore direita for negativo ou zero, uma rotação simples à esquerda é aplicada.
- Caso contrário, realiza-se uma rotação dupla à esquerda (uma rotação à direita seguida de uma rotação à esquerda).

```
Node* balancear(Node* raiz) {
    raiz->fb = fatorBalanceamento(raiz);

    if (raiz->fb > 1) {
        if (fatorBalanceamento(raiz->esq) >= 0) {
            return rotacaoDireita(raiz);
        } else {
            return rotacaoDuplaDireita(raiz);
        }
    }

    if (raiz->fb < -1) {
        if (fatorBalanceamento(raiz->dir) <= 0) {
            return rotacaoEsquerda(raiz);
        } else {
            return rotacaoDuplaEsquerda(raiz);
        }
    }

    return raiz;
}
```

Implementando a inserção

- Se o fator de balanceamento da raiz estiver entre -1 e 1, a árvore já está balanceada, e a função simplesmente retorna a raiz sem alterações.

```
Node* balancear(Node* raiz) {
    raiz->fb = fatorBalanceamento(raiz);

    if (raiz->fb > 1) {
        if (fatorBalanceamento(raiz->esq) >= 0) {
            return rotacaoDireita(raiz);
        } else {
            return rotacaoDuplaDireita(raiz);
        }
    }

    if (raiz->fb < -1) {
        if (fatorBalanceamento(raiz->dir) <= 0) {
            return rotacaoEsquerda(raiz);
        } else {
            return rotacaoDuplaEsquerda(raiz);
        }
    }

    return raiz;
}
```

Implementando a inserção

- Primeiro, ela verifica se a árvore está vazia (se raiz é NULL).
- Se estiver, cria um novo nó, inicializa seus valores (informação e fator de balanceamento) e o retorna como a nova raiz da árvore.
- Caso contrário, a função compara o valor a ser inserido com o valor do nó atual.

```c
Node* inserirAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        Node* novo = (Node*)malloc(sizeof(Node));
        novo->info = valor;
        novo->fb = 0;
        novo->esq = novo->dir = NULL;
        return novo;
    }

    if (valor < raiz->info) {
        raiz->esq = inserirAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = inserirAVL(raiz->dir, valor);
    } else {
        return raiz; // Valor duplicado não inserido
    }
    return balancear(raiz);
}
```

Implementando a inserção

- Se o valor for menor que o do nó, a função recursivamente tenta inserir o valor na subárvore esquerda;
- Se for maior, tenta na subárvore direita. Se o valor já existir na árvore (caso de duplicata), a função retorna a raiz sem fazer alterações.

```
Node* inserirAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        Node* novo = (Node*)malloc(sizeof(Node));
        novo->info = valor;
        novo->fb = 0;
        novo->esq = novo->dir = NULL;
        return novo;
    }

    if (valor < raiz->info) {
        raiz->esq = inserirAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = inserirAVL(raiz->dir, valor);
    } else {
        return raiz; // Valor duplicado não inserido
    }
    return balancear(raiz);
}
```

Implementando a inserção

- Após inserir o valor, a função chama balancear para garantir que a árvore AVL permaneça balanceada, atualizando os fatores de balanceamento dos nós afetados e realizando rotações, se necessário.
- A função retorna a raiz da árvore (que pode ter sido alterada durante o balanceamento).

```c
Node* inserirAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        Node* novo = (Node*)malloc(sizeof(Node));
        novo->info = valor;
        novo->fb = 0;
        novo->esq = novo->dir = NULL;
        return novo;
    }

    if (valor < raiz->info) {
        raiz->esq = inserirAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = inserirAVL(raiz->dir, valor);
    } else {
        return raiz; // Valor duplicado não inserido
    }
    return balancear(raiz);
}
```

Implementando a remoção

- A função começa verificando se a árvore está vazia (raiz == NULL), caso em que retorna NULL.
- Se o valor a ser removido for menor que o valor do nó atual, a remoção é feita recursivamente na subárvore esquerda, e se for maior, na subárvore direita.

```
Node* removeAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        return NULL;
    }
    if (valor < raiz->info) {
        raiz->esq = removeAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = removeAVL(raiz->dir, valor);
    } else {
        if (raiz->esq == NULL || raiz->dir == NULL) {
            Node* temp = raiz->esq ? raiz->esq : raiz->dir;
            if (temp == NULL) {
                temp = raiz;
                raiz = NULL;
            } else {
                *raiz = *temp;
            }
            free(temp);
        } else {
            Node* temp = encontrarMinimo(raiz->dir);
            raiz->info = temp->info;
            raiz->dir = removeAVL(raiz->dir, temp->info);
        }
    }
    if (raiz == NULL) {
        return NULL;
    }
    return balancear(raiz);
}
```

Implementando a remoção

- Quando o valor a ser removido é encontrado, a função verifica se o nó tem um ou nenhum filho.
- Se tiver apenas um filho (ou nenhum), o nó é substituído pelo seu filho (se houver), e a memória do nó removido é liberada com free.

```
Node* removeAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        return NULL;
    }
    if (valor < raiz->info) {
        raiz->esq = removeAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = removeAVL(raiz->dir, valor);
    } else {
        if (raiz->esq == NULL || raiz->dir == NULL) {
            Node* temp = raiz->esq ? raiz->esq : raiz->dir;
            if (temp == NULL) {
                temp = raiz;
                raiz = NULL;
            } else {
                *raiz = *temp;
            }
            free(temp);
        } else {
            Node* temp = encontrarMinimo(raiz->dir);
            raiz->info = temp->info;
            raiz->dir = removeAVL(raiz->dir, temp->info);
        }
    }
    if (raiz == NULL) {
        return NULL;
    }
    return balancear(raiz);
}
```

Implementando a remoção

- Se o nó a ser removido tiver dois filhos, o nó com o menor valor na subárvore direita (sucessor) é encontrado e seu valor é copiado para o nó a ser removido, seguido da remoção desse sucessor.
- Após a remoção, a função verifica se a árvore ficou vazia e, caso contrário, chama a função balancear para garantir que a árvore AVL esteja balanceada.

```
Node* removeAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        return NULL;
    }
    if (valor < raiz->info) {
        raiz->esq = removeAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = removeAVL(raiz->dir, valor);
    } else {
        if (raiz->esq == NULL || raiz->dir == NULL) {
            Node* temp = raiz->esq ? raiz->esq : raiz->dir;
            if (temp == NULL) {
                temp = raiz;
                raiz = NULL;
            } else {
                *raiz = *temp;
            }
            free(temp);
        } else {
            Node* temp = encontrarMinimo(raiz->dir);
            raiz->info = temp->info;
            raiz->dir = removeAVL(raiz->dir, temp->info);
        }
    }
    if (raiz == NULL) {
        return NULL;
    }
    return balancear(raiz);
}
```

Implementando a remoção

- Se necessário, a árvore é balanceada com rotações. Por fim, a função retorna a raiz (que pode ter mudado durante o processo de remoção e balanceamento).

```
Node* removeAVL(Node* raiz, int valor) {
    if (raiz == NULL) {
        return NULL;
    }
    if (valor < raiz->info) {
        raiz->esq = removeAVL(raiz->esq, valor);
    } else if (valor > raiz->info) {
        raiz->dir = removeAVL(raiz->dir, valor);
    } else {
        if (raiz->esq == NULL || raiz->dir == NULL) {
            Node* temp = raiz->esq ? raiz->esq : raiz->dir;
            if (temp == NULL) {
                temp = raiz;
                raiz = NULL;
            } else {
                *raiz = *temp;
            }
            free(temp);
        } else {
            Node* temp = encontrarMinimo(raiz->dir);
            raiz->info = temp->info;
            raiz->dir = removeAVL(raiz->dir, temp->info);
        }
    }
    if (raiz == NULL) {
        return NULL;
    }
    return balancear(raiz);
}
```

Algoritmos e Estrutura de Dados II

Prof. Fellipe Guilherme Rey de Souza

Aula 14 –Árvore AVL (Remoção e Implementação)