Notas de Aula - AED2 – Árvores: árvores AVL
Prof. Jefferson T. Oliva

Recapitulação sobre árvores binárias de busca (ABB)
- Nó raiz
- Subárvores
- ABB: possui propriedades de ordenação
- n elementos → n – 1 ramificações
- altura mínima: log n
- altura máxima: n
- Representação de uma ABB: estrutura com um campo para o item e dois ponteiros, uma subárvore direita e a outra, esquerda
- Operações: criar, liberar, buscar, inserir e remover, imprimir (pré-, in- e pós-fix)

As ABB são implementadas para agilizar o acesso à informação. Para isso, a árvore deve ser balanceada de modo que garanta que o acesso seja feito no máximo em tempo na ordem de O(log n).

Entretanto, sucessivas inserções de itens podem acarretar no aumento da complexidade de tempo para O(n), que é o caso em que a árvore cresce em apenas uma direção, lembrando uma lista encadeada.

Vimos em uma aula anterior, algumas propriedades de árvores, como altura, profundidade e balanceamento, sendo esta última abordado mais profundamente nessa aula.

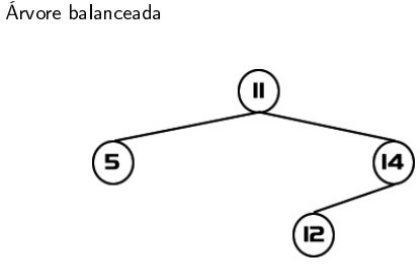
Balanceamento

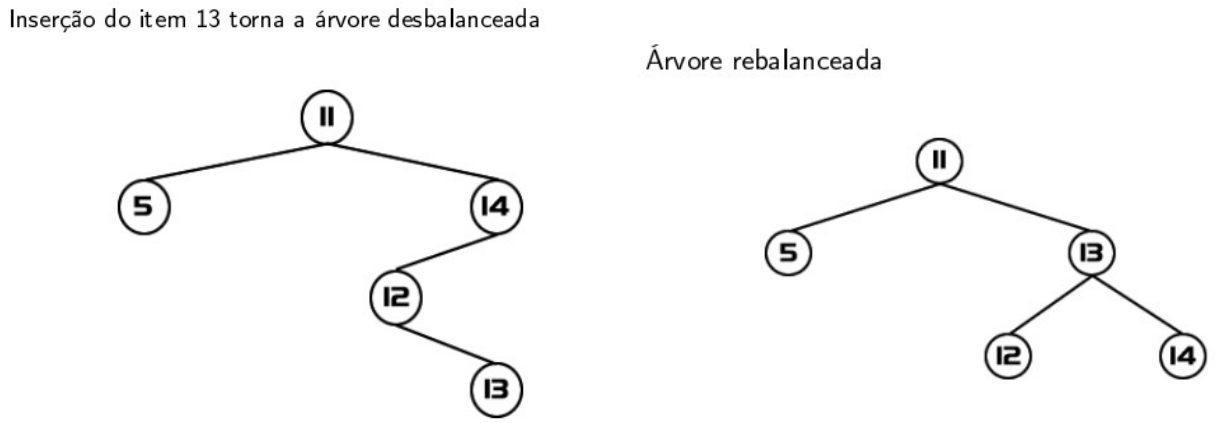
Quando uma árvore está balanceada, o número de comparações em uma operação de busca é minimizada: no máximo O(log n).

Em outras palavras, a altura da árvore é mantida baixa em relação à quantidade de itens após as operações de inserção e remoção. Assim, uma árvore balanceada de altura h pode conter, no máximo, 2^{h + 1} – 1 elementos.

Diz-se que uma árvore é balanceada se a altura entre as subárvores diferem no máximo em 1.

Portanto, manter árvores balanceadas é uma tarefa complexa.





Existem dois tipos de árvores que são balanceadas após as operações de inserção e de remoção: AVL e vermelha-preta.

Árvores AVL

Proposta por Adelson, Velsky e Landis em 1962, AVL é um tipo de ABB que são rebalanceadas, caso a mesma se torna desbalanceada.

Em uma árvore AVL, as operações de busca, inserção e remoção têm a complexidade de tempo na ordem de O(log n) tanto para o caso médio quanto para o pior caso.

Uma ABB vazia ou de altura entre 0 e 1 é uma árvore AVL.

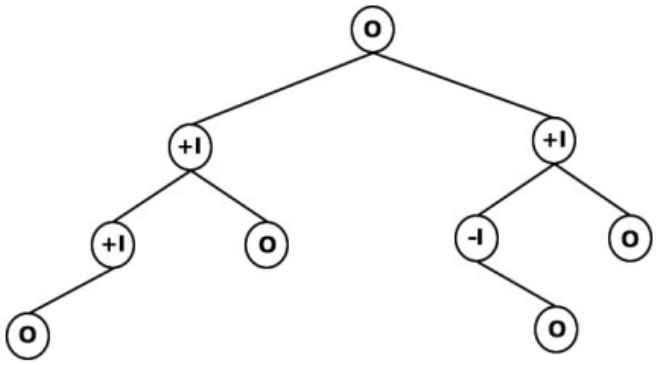
Como podemos ver se uma ABB é balanceada?
Resp.: através do uso do fator de balanceamento

O fator de balanceamento (FB) é a diferença de altura entre a subárvore esquerda (He) e a direita (Hd). Em uma árvore AVL, cada subárvore deve ter altura equilibrada (de acordo com o fator de balanceamento).

Assim, cada nó de uma árvore AVL deve conter um valor de FB:
→ -1: a altura da subárvore direita é maior que a da esquerda
→ 0: as subárvores direita e esquerda são iguais
→ +1: a altura da subárvore esquerda é maior que a da direita

Em uma operação de inserção ou remoção, caso uma subárvore fique com altura menor que -1 ou maior que +1, a árvore deve ser rebalanceada.

Exemplo de árvore balanceada com fator de balanceamento em cada nó



Caso o balanceamento, de acordo com FB, seja violado, alguma das seguintes operações deve ser realizada:
- Left-left (LL): rotação simples da subárvore esquerda
- Right-right (RR): rotação simples da subárvore direita
- Left-right (LR): a subárvore esquerda deve ser rotacionada para subárvore direita
- Right-left (RL): vice-versa

Exemplos interessantes entre os slides 17 e 25

No slide 21: a rotação é aplicado no nó desequilibrado mais profundo primeiro. Depois, o problema é resolvido para o nó pai. No exemplo, ao rebalancear o nó “maio”, o nó “março” fica balanceado, sem a necessidade de operações adicionais.

No slide 22, começa a complicar, pois uma rotação simples não resolve o problema.
- Vocês lembram da operação de exclusão? Suponha que mudamos o código para utilizar o maior valor da subárvore esquerda para substituir o nó excluído. Se fôssemos excluir o item “março”, como ficaria? O nó “maio” iria substituir “março”, certo? O lado esquerdo de “maio” seria “agosto” e o direito “novembro”. Também, antes da atualização, o lado direito de “maio” (NULL) passaria a ser lado esquerdo de “novembro”, e o lado esquerdo de “maio” passaria ser lado direito de “agosto”.
- Na rotação LR, não ocorre remoção, mas alguns movimentos são parecidos em relação à exclusão. No exemplo apresentado, grande partes das operações coincidem com a maior parte dos movimentos em uma remoção: o maior nó da subárvore esquerda (em relação ao nó com FB desbalanceado) passaria a ser o nó raiz (irá ocupar o lugar do nó com FB desbalanceado na hierarquia), e o antigo nó raiz irá compor a subárvore direita. No exemplo apresentado, o lado direito de agosto passará a apontar para o lado direito de “maio”, que é “agosto”. O lado esquerdo de “março” passará a apontar para o lado direito de “maio”, que é NULL. Por fim, o lado esquerdo de “maio” passará a ser “agosto” e o direito, “março”.

Por mais que a movimentação da rotação possa lembrar a operação de remoção em uma ABB, na realidade, na rotação LR não é procurado o maior elemento da subárvore esquerda. A operação é bem mais simples: o que substituirá o nó desbalanceado será o primeiro nó a direita da subárvore esquerda. Por que isso? Apenas fazendo isso, seria com se eu tivesse puxando a parte responsável pelo desbalanceamento para cima. Por que não fazer igual à operação de remoção (buscar o menor

ou o maior elemento em sub-árvores)? Como o FB pode ser desbalanceado em qualquer nível, o custo pode ser elevado para busca. Também, essa abordagem não resolveria o problema, pois o nó em que foi constatado o desbalanceamento pode apenas mudar o sinal do FB, ou seja, continuaria desbalanceado e seria necessária mais uma aplicação de rebalanceamento. Essa repetição poderia ser necessária até que operação chegasse perto dos nós folhas.

Assim, considerar apenas um filho da subárvore seria a estratégia mais econômica computacionalmente, além de resolver o problema eficientemente.

Em cada inserção e remoção em árvores AVL é feita uma busca, como em remoção? Resposta: não, pois não é necessária a busca, pois os nós que serão utilizados para a rotação já estão próximos ao nó com o FB desequilibrado.

Ver até o slide 26.

Agora iremos ver alguns fragmentos de código de árvore AVL.

Começaremos pela estrutura:

```
typedef struct Pointer{
        int item;
        int fb;
        struct Pointer* right;
        struct Pointer* left;
}Node;
```

Qual a diferença dessa estrutura em relação à ABB das aulas anteriores?

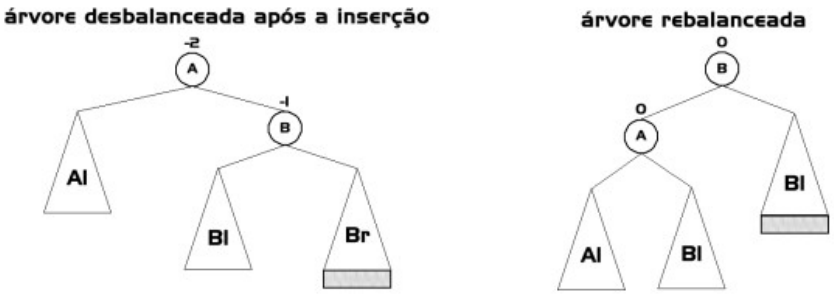
Sobre as rotações, começamentos com os exemplos mais simples.

<u>Quando um nó possui tamanho maior ou igual a +2, significa que o lado esquerdo está desbalanceado. Quando a raiz da subárvore tem FB = +1, significa que o movimento que deve ser feito é o LL. Se FB = -1, então a operação deverá ser LR.</u>

Começando com a rotação LL

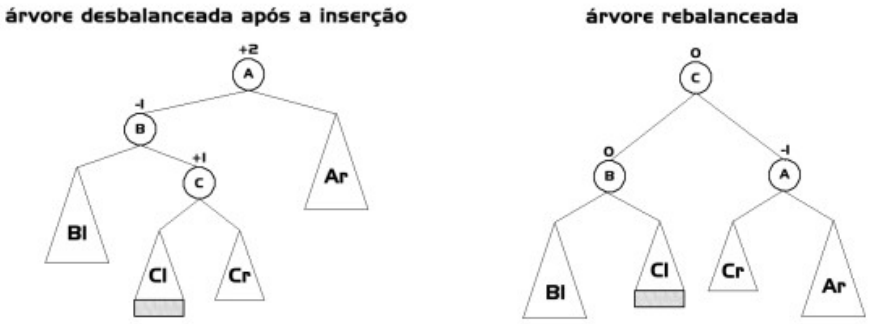
```
Node *aux = tree->left;
tree->left = aux->right;
aux->right = tree;
tree = aux;
```

Rotação RR

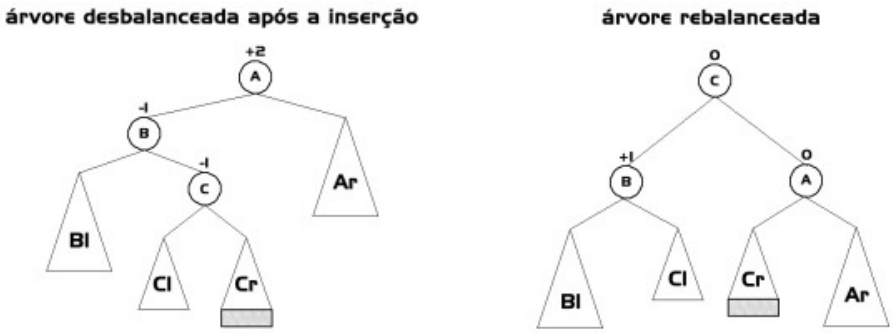



```
Node *aux = tree->right;
tree->right = aux->left;
aux->left = tree;
tree = aux;
```

Rotação LR

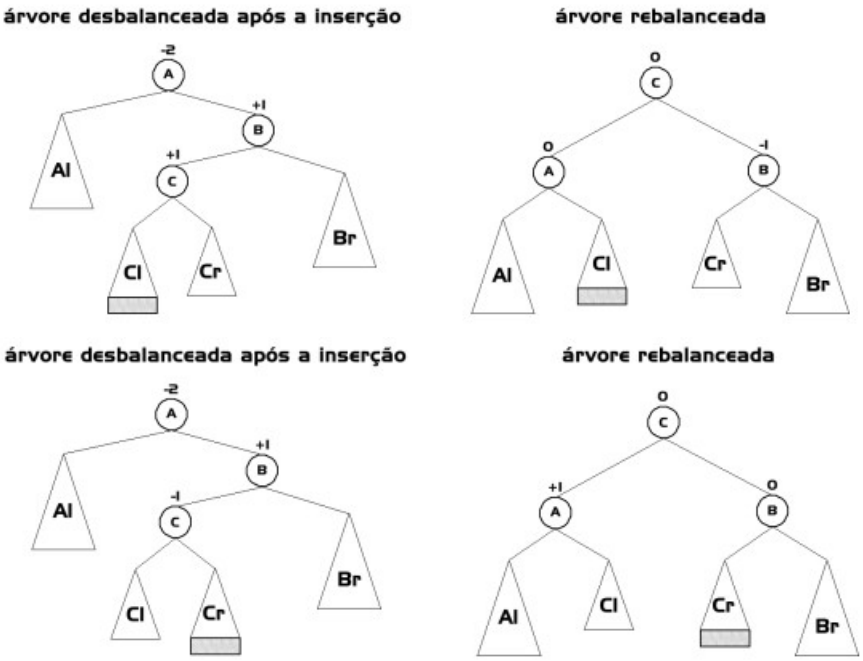



```
Node *auxA = tree->left;
Node *auxB = auxA->right;
auxA->left = auxB->left;
auxB->left =  auxA;
tree->left = auxB->right;
auxB->right = tree;
tree = auxB;
```



Rotação RL

```
Node *auxA = tree->right;
Node *auxB = auxA->left;
auxA->left = auxB->right;
auxB->right = auxA;
tree->right = auxB->left;
auxB->left = tree;
tree = auxB;
```



A inserção e a remoção de itens em árvores AVL são realizadas da mesma forma que em árvores binárias de busca apresentadas na aula anterior, mas a diferença é que pode ser necessário rebalanceamento da árvore após essas operações.

A complexidade de busca é O(log n)

Implementação da operação de inserção da árvore AVL

```
void rotateL(Node *tree){
        Node *auxA = tree->left; *auxB;

        if (auxA->fb == +1){
                tree->left = auxA->right;
                auxA->right = tree;
                tree->fb = 0;
                tree = auxA;
        }else{
                auxB = auxA->right;
                auxA->right = auxB->left;
                auxB->left = auxA;
                tree->left = auxB->right;
                auxB->right = tree;

                if (auxB->fb == +1)  tree->fb = -1;
                else  tree->fb = 0;

                if (auxB->fb == -1)  auxA->fb = +1;
                else  auxA->fb = 0;

                tree = auxB;
        }
}


void rotateR(Node *tree){
        Node *auxA = tree->right; *auxB;

        if (auxA->fb == -1){
                tree->right = auxA->left;
                auxA->left = tree;
                tree = auxA;
        }else{
                auxB = auxA->left;
                auxA->left = auxB->right;
                auxB->right = auxA;
                tree->right = auxB->left;
                auxB->left = tree;

                if (auxB->fb == -1)
                        tree->fb = +1;
                else
                        tree->fb = 0;

                if (auxB-> fb == +1)
                        auxA->fb = -1;
                else
                        auxA->fb = 0;

                tree = auxB;
        }
}
```

```
Node* insert(Node* tree, int value, int grown){
	Node *auxA, *auxB;

	if( tree == NULL){
		create(tree);

		tree->item = value;
		tree->bf = 0;
		grown = true;

		return tree;
	}else if (value < tree->item){
		return tree->left = insert(tree->left, value, &grown);

		if (grown){ // verificar se aumentou a árvore pelo lado esquerdo
			switch (tree->fb){
				case -1: tree->fb = 0; grown = false; break; // AVL balanceada
				case 0: tree->fb = +1; break;// AVL balanceada
				case +1: rotationL(tree); tree->bf = 0; grown = false;// AVL desbalanceada
			}
		}
	}else if (value < tree->item){
		return tree->left = iinsert(tree->right, value, &grown);

		if (grown){
			switch (tree->bf){ // verificar se aumentou a árvore pelo lado direto
				case +1: tree->bf = 0; grown = false; break;
				case 0: tree->bf = -1; break;
				case -1: rotationR(tree); tree->bf = 0; grown = false;
			}
		}
	}
}
```

Implementação da operação de Remoção da árvore AVL

```
static Node* rotateLRM(Node *tree, int *reduced){
          Node *auxA, *auxB;

          switch (tree->fb){
      case +1: tree->fb = 0; break; // AVL balanceada
      case 0: tree->fb = -1; *reduced = 0; break;// AVL balanceada
      case -1:
         auxA = tree->right;

         if (auxA->fb <= 0){ // rotação RR
            tree->right = auxA->left;
            auxA->left = tree;

            if(auxA->fb == 0){
               tree->fb = -1;
               auxA->fb = +1;
               *reduced = 0;
            }else
               tree->fb = auxA->fb = 0;

            tree = auxA;
         }else{ // Rotação RL
            auxB = auxA->left;
            auxA->left = auxB->right;
            auxB->right = auxA;
            tree->right = auxB->left;
            auxB->left = tree;

            if (auxB->fb == -1) tree->fb = +1;
            else
               tree->fb = 0;

            if (auxB->fb == +1)
               auxA->fb = -1;
            else
               auxA->fb = 0;

            tree = auxB;
            auxB->fb = 0;
         }
   }

          return tree;
}
```

```
static Node* rotateRRM(Node *tree, int *reduced){
        Node *auxA, *auxB;

        switch (tree->fb){
    case -1: tree->fb = 0; *reduced = 0; break;
    case 0: tree->fb = 1; break;
    case +1:
      auxA = tree->left;

      if (auxA->fb >= 0){ // rotação LL
        tree->left = auxA->right;
        auxA->right = tree;

        if(auxA->fb == 0){
          tree->fb = +1;
          auxA->fb = -1;
          *reduced = 0;
        }else
          tree->fb = auxA->fb = 0;

        tree = auxA;
      }else{ // Rotação LR
        auxB = auxA->right;
        auxA->right = auxB->left;
        auxB->left = auxA;
        tree->left = auxB->right;
        auxB->right = tree;

        // Atualização dos fatores de balanceamento
        if (auxB->fb == +1)
          tree->fb = -1;
        else
          tree->fb = 0;

        if (auxB->fb == -1)
          auxA->fb = +1;
        else
          auxA->fb = 0;

        tree = auxB;
        auxB->fb = 0;
      }
  }

        return tree;
}
```

```
Node* remover(Node* tree, int valor, int *reduced){
   Node *aux, *auxP, *auxF;

   if (tree != NULL){
      if (valor < tree->item){
         tree->left = remover(tree->left, valor, reduced);

         if (*reduced)
            tree = rotateLRM(tree, reduced);
      }else if (valor > tree->item){
         tree->right = remover(tree->right, valor, reduced);

         if (*reduced)
            tree = rotateRRM(tree, reduced);
      }else{
         aux = tree;

         if (aux->left == NULL)
            tree = tree->right;
         else if (aux->right == NULL)
            tree = tree->left;
          else{
            auxP = aux->right;
            auxF = auxP;

            while (auxF->left != NULL){
               auxP = auxF;
               auxF = auxF->left;
            }

            if (auxP != auxF){
               auxP->left = auxF->right;
               auxF->left = aux->left;
            }

            auxF->right = aux->right;

            tree = auxF;
         }

         *reduced = 1;
         free(aux);
      }
   }

   return tree;
}
```

Referências

Baranauskas, J. A. Árvores AVL - Algoritmos e Estruturas de Dados I. Slides. Ciência da Computação FFCLRP-USP, Ribeirão Preto, 2013.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. Introduction to Algorithms. Third edition, The MIT Press, 2009.

Marin, L. O. Árvores AVL. AE23CP - Algoritmos e Estrutura de Dados II. Slides. Engenharia de Computação. Dainf/UTFPR/Pato Branco, 2017.

Szwarcfiter, J.; Markenzon, L. Estruturas de Dados e Seus Algoritmos. LTC, 1994.

Tenenbaum, A.; Langsam, Y. Estruturas de Dados usando C. Pearson, 1995.

Ziviani, N. Projeto de Algoritmos - com implementações em Java e C++. Thomson, 2007.