Aula 6: BST e BST Balanceado (AVL)

DCC405-Estrutura de Dados II

Prof. Me. Acauan C. Ribeiro

1. Introdução

- Uma árvore de busca binária (BST) é uma árvore binária em que cada vértice tem apenas até 2 filhos que satisfazem a propriedade BST:
  - todos os vértices na subárvore esquerda de um vértice devem conter um valor menor que o seu próprio e todos os vértices na subárvore direita de um vértice deve conter um valor maior que o seu próprio (assumimos que todos os valores são inteiros distintos nesta visualização e um pequeno ajuste é necessário para atender a duplicatas/não inteiros).
- Fazer exemplo Search(7) para obter um exemplo de animação na busca de um valor aleatório $\in [1..99]$ no BST.
- Uma árvore **Adelson-Velskii Landis (AVL)** é um BST autobalanceado que mantém sua altura em $O(\log N)$ quando possui N vértices na árvore AVL.

2. BST e BST balanceado (árvore AVL)

- A diferença Árvore de Pesquisa Binária padrão e a Árvore AVL ocorre no momento da Inserção e Remoção de elementos da árvore.
- O objetivo da árvore AVL é manter a arvore sempre distribuida de maneira organizada para diminuir sua altura. Neste sentido as ações de custo maior numa árvore binária sempre vão ser O(log N), devido as regras da AVL.

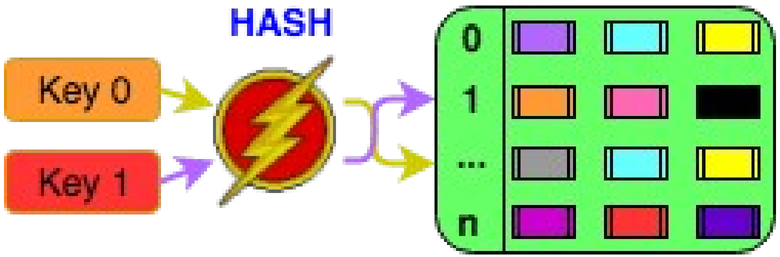
Georgy Maksimovich Adelson-Velsky was born in Samara (Russia) on January 8, 1922. In 1940 he enrolled in the Faculty of Mechanics and Mathematics at Moscow State University.

3. Motivação

- BST (e BST especialmente balanceado como AVL Tree) é uma estrutura de dados eficiente para implementar um certo tipo de tabela – tabela hash (ou map) sendo assim um Tipo Abstrato de Dado (TAD).
- Uma Tabela com essa estrutura deve suportar pelo menos as três operações da forma mais eficiente possível:
  - **Search(v)** — determina se v existe no TAD ou não,
  - **Insert(v)** — insere v no TAD,
  - **Remove(v)** — remove v do TAD.

3-1. Que tipo de Tabela TAD?

- Estamos nos referindo à Tabela TAD onde as chaves precisam ser ordenadas (ao contrário da Tabela TAD onde as chaves não precisam ser desordenadas como o caso base de Tabelas Hash).
- Este requisito especial da Tabela ficará mais claro nos próximos slides.



3-2. Usando matriz/vetor não ordenado

Se usarmos matriz/vetor não classificado para implementar a Tabela TAD descrita, podemos ter métodos ineficientes:

- **Search(v)** roda em O(N), pois podemos acabar explorando todos os N elementos do TAD se v realmente não existir,
- **Insert(v)** pode ser implementado em O(1), apenas coloque v no final do array,
- **Remove(v)** também é executado em O(N), pois primeiro temos que procurar por v que já é O(N) e depois fechar a lacuna resultante após a exclusão — também em O(N).

3-3. Usando matriz/vetor ordenado

Se usarmos array/vetor classificado para implementar a Table, podemos melhorar o desempenho de **Search(v)**, mas enfraquecemos o desempenho de **Insert(v)**:

- **Search(v)** agora pode ser implementado em O(log N), pois agora podemos usar a pesquisa binária no array classificado,
- **Insert(v)** agora é executado em O(N), pois precisamos implementar uma estratégia de classificação por inserção para fazer com que o array permaneça classificado,
- **Remove(v)** é executado em O(N) porque mesmo que Search(v) seja executado em O(log N), ainda precisamos fechar a lacuna após a exclusão — que está em O(N).

3-4. O(log N) Complexidades?

- Um dos objetivos desta aula é apresentar a estrutura de dados BST e BST balanceada (Árvore AVL) para que possamos implementar as operações básicas da Table TAD: Search(v), Insert(v), Remove(v) e algumas outras operações — veja o próximo slide — em tempo O(log N) — que é muito menor que N .
- OBS: Alguns dos alunos mais atentos podem notar que vimos outra estrutura de dados que pode implementar as três operações básicas do Table-TAD-BST em um tempo mais rápido

N	≈ 1 000	≈ 1 000 000	≈ 1 000 000 000
log **N**	10	Only 20 :O	Only 30 :O:O

3-5. Outras operações de tabela TAD

Além das três básicas, existem algumas outras operações possíveis do Table ADT:

- Encontre o elemento **Min()/Max()**,
- Encontre o elemento **Sucessor(v)** — 'próximo maior'/**Predecessor(v)** — 'menor anterior',
- Listar elementos em ordem de classificação,
- Existem outras operações possíveis.

Discussão: Qual é a melhor implementação possível para as três primeiras operações adicionais se estivermos limitados a usar array/vetor [classificados|não classificados]?

3-6. Discussão

- Para um array não ordenado, encontre as execuções **Min()/Max() em O(N)**, pois não sabemos onde o elemento min/max reside. Também é O(N) para localizar o Sucessor(v)/Predecessor(v) pelo mesmo motivo. A listagem de elementos na ordem de classificação envolve uma chamada para o algoritmo de classificação, portanto, pode ser O(N log N).

- Para um array ordenado, encontre as execuções **Min()/Max() em O(1)**, pois o elemento min/max reside na primeira/última posição, respectivamente. Também é O(1) para encontrar o Sucessor(v)/Predecessor(v) — basta voltar/avançar um índice de v, respectivamente. Listar elementos em ordem de classificação é apenas O(N) da primeira à última enumeração, pois a matriz já está classificada. Isso parece bom, mas quando houver mais inserções e/ou exclusões, o array ordenado não será mais eficaz.

3.7 – Tabela Hash vs BST (AVL)

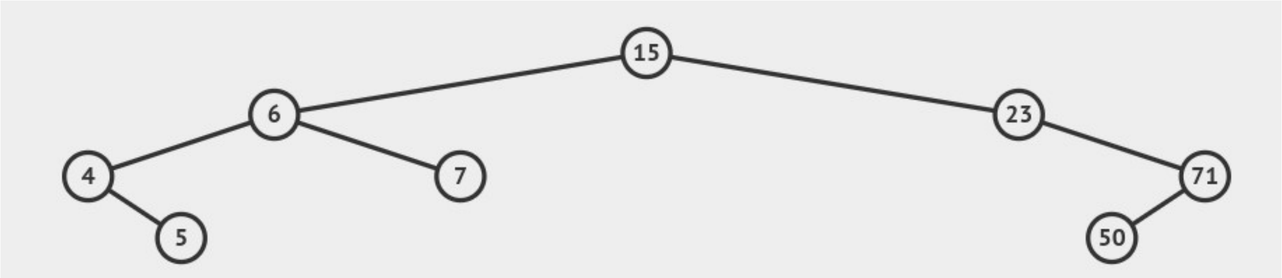
- Outra estrutura de dados que pode ser usada para implementar a Table discutida é a **Hash Table** . Ela tem um desempenho muito rápido de **Pesquisa(v)**, **Inserção(v)** e **Remoção(v)** (tudo no tempo O(1) esperado).
- **Pergunta:** Então, qual é o objetivo de aprender este módulo BST se a tabela hash pode fazer as operações cruciais de tabela TAD em tempo O(1) esperado improvável de ser superado ?

3-7. Discussão

- Na lista de outras operações Table TAD, existem três operações Table TAD adicionais que requerem a ordenação dos elementos na Table.
- Se os elementos não precisam ser ordenados, podemos usar a estrutura de dados da **tabela hash** não ordenada que é muito rápida, ou seja, O(1) Search(v), Insert(v) e Remove(v).
- Porém, se precisarmos **realizar consultas envolvendo a ordem dos elementos** na Tabela, essa estrutura de dados BST, principalmente a versão balanceada, é o caminho a seguir.

4. Revisar - BST

- O vértice raiz não tem um pai. Só pode haver um vértice raiz em um BST. O vértice da folha não tem filhos. Pode haver mais de um vértice folha em uma BST. Os vértices que não são folha são chamados de vértices internos. Às vezes, o vértice raiz não é incluído como parte da definição de vértice interno, pois a raiz de um BST com apenas um vértice pode realmente se encaixar na definição de uma folha também.



- No exemplo acima, o vértice **15** é o vértice raiz, o vértice **{5, 7, 50}** são as folhas, o vértice **{4, 6, 15 (também a raiz), 23, 71}** são os vértices internos.

4-1. Atributos do Vértice BST

- Cada vértice (nó) tem pelo menos **4 atributos**: pai, esquerda, direita, chave/valor/dados (existem outros atributos em potencial). Nem todos os atributos serão usados para todos os vértices, por exemplo, o vértice raiz terá seu atributo pai = NULL. Alguma outra implementação separa a chave (para ordenação de vértices no BST) com os dados de satélite reais associados às chaves.
- O filho esquerdo/direito de um vértice (exceto folha) é desenhado à esquerda/direita e abaixo desse vértice, respectivamente. O pai de um vértice (exceto a raiz) é desenhado acima desse vértice. A chave (inteira) de cada vértice é desenhada dentro do círculo que representa aquele vértice. No exemplo anterior, (chave) 15 tem 6 como filho esquerdo e 23 como filho direito. Assim, o pai de 6 (e 23) é 15.

4-2. Propriedade BST

- Caso a árvore não permita chaves repetidas, a propriedade BST é a seguinte: Para cada vértice X, todos os vértices na subárvore esquerda de X são estritamente menores que X e todos os vértices na subárvore direita de X são estritamente maiores que X .
- No exemplo acima, os vértices da subárvore esquerda da raiz 15: $\{4, 5, 6, 7\}$ são todos menores que 15 e os vértices da subárvore direita da raiz 15: $\{23, 50, 71\}$ são todos maiores que 15. Você também pode verificar recursivamente a propriedade BST em outros vértices.
- Para uma implementação mais completa, devemos considerar inteiros duplicados também. A maneira mais fácil de suportar isso é adicionar mais um atributo em cada vértice: a frequência de ocorrência de X.

5. Operações BST

Operações comuns da árvore BST/AVL:

- Operações de consulta (a estrutura BST permanece inalterada):
  - Pesquisa(v),
  - Antecessor(v) (e similarmente Sucessor(v)), e
  - Inorder/Preorder Traversal/Postorder Traversal
- Operações de atualização (a estrutura BST provavelmente pode mudar):
  - Inserir(v),
  - Remover(v) e
  - Criar BST (vários critérios).

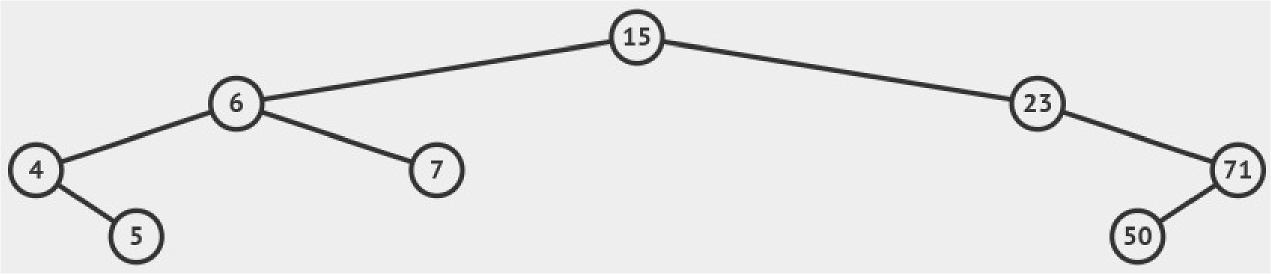
5-1. Algumas outras operações BST

Existem algumas outras operações de BST (Consulta) que não foram aprofundadas:

- Classificação(v): Dada uma chave v , determine qual é sua classificação (índice baseado em 1) na ordem classificada dos elementos BST. Ou seja, $\text{Rank}(\text{FindMin}()) = 1$ e $\text{Rank}(\text{FindMax}()) = N$. Se v não existir, podemos relatar -1.
- Select(k): Dado um posto **k , 1 ≤ k ≤ N** , determine a chave v que tem esse posto k no BST. Ou, em outras palavras, encontre o k-ésimo menor elemento no BST. Ou seja, $\text{Select}(1) = \text{FindMin}()$ e $\text{Select}(N) = \text{FindMax}()$.

6. Pesquisa(v)

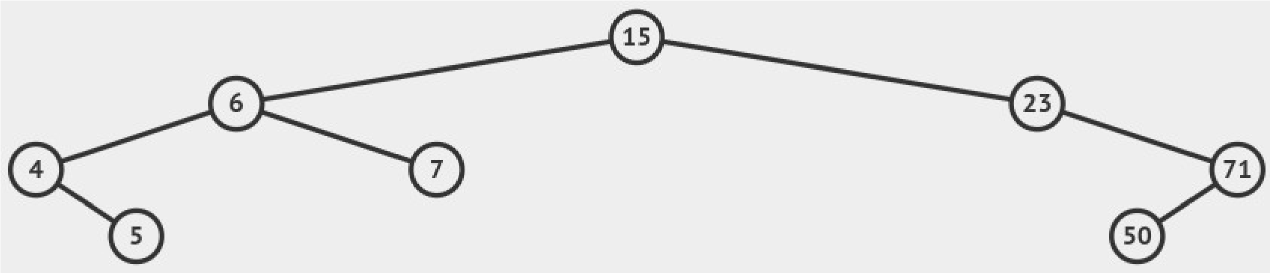
- Devido à forma como os dados (números inteiros distintos para este exemplo) são organizados dentro de uma BST, podemos fazer uma busca binária por um número inteiro v de forma eficiente (daí o nome Árvore de Pesquisa Binária).
- Primeiro, definimos o vértice **atual** = **root** e, em seguida, verificamos se o vértice atual é menor/igual/maior que o inteiro v que estamos procurando. Em seguida, vamos para a subárvore direita/paramos/vamos para a subárvore esquerda, respectivamente. Continuamos fazendo isso até encontrarmos o vértice necessário ou não.



- No exemplo BST acima, tente clicar em **Search(23)** (encontrado após 2 comparações), **Search(7)** (encontrado após 3 comparações), **Search(21)** (não encontrado após 2 comparações - neste ponto, perceberemos que não podemos encontrar 21).

6-1. FindMin() e FindMax() - Implementar

- Da mesma forma, devido à forma como os dados são organizados dentro de um BST, podemos encontrar o elemento mínimo/máximo (um número inteiro nesta visualização) começando pela raiz e seguindo para a subárvore esquerda/direita, respectivamente.



- Tente clicar em **FindMin() e FindMax()** no exemplo BST mostrado acima. As respostas devem ser 4 e 71 (ambas após comparação com 3 inteiros da raiz ao vértice mais à esquerda/vértice mais à direita, respectivamente).

6-2. Complexidade de Tempo O(h)

- As operações **Search(v)/FindMin()/FindMax()** são executadas em **O(h)**, onde **h** é a **altura do BST**.
- Mas observe que este h pode ser tão alto quanto O(N) em um BST normal, conforme comentado em aulas anteriores 'degenerado à direita'.
- Fazer exemplo **Search(100)** (este valor não deve existir, pois usamos apenas números inteiros aleatórios entre [1..99] para gerar esse BST aleatório e, portanto, a rotina de pesquisa deve verificar todo o caminho da raiz até a única folha em O(N) tempo - não eficiente.

7. Sucessor(v) - Implementar

- Devido às propriedades do BST, podemos encontrar o **Sucessor** de um inteiro v (suponha que já sabemos onde o inteiro v está localizado na chamada anterior de Search(v)) da seguinte maneira:
  - Se v tiver uma subárvore direita, o inteiro mínimo na subárvore direita de v deve ser o sucessor de v . Tente **Successor(23)** (deve ser 50).
  - Se v não tiver uma subárvore à direita, precisamos percorrer o(s) ancestral(es) de v até encontrarmos 'uma curva à direita' para o vértice w (ou alternativamente, até encontrarmos o primeiro vértice w que é maior que o vértice v). Assim que encontrarmos o vértice w , veremos que o vértice v é o elemento máximo na subárvore esquerda de w . Tente **Successor(7)** (deve ser 15).
  - Se v é o número inteiro máximo no BST, v não tem um sucessor. Tente **Successor(71)** (deve ser nenhum).

7-1. Antecessor(v) - Implementar

- As operações para o **Antecessor** de um inteiro v são definidas de forma semelhante (apenas o espelho das operações do Sucessor).
- Tente os mesmos três casos de canto (mas espelhados): **Predecessor(6)** (deve ser 5), **Predecessor(50)** (deve ser 23), **Predecessor(4)** (deve ser nenhum).
- Neste ponto, pare e pondere sobre esses três casos Sucessor(v)/Predecessor(v) para garantir que você entenda esses conceitos.

Questão

- **Quiz:** Inserir números inteiros [1, 10, 2, 9, 3, 8, 4, 7, 5, 6] um por um, nessa ordem, em uma BST inicialmente vazio resultará em um BST de altura:

  ( ) A altura não pode ser determinada

  ( ) 9

  ( ) 8

  ( ) 10

8. Remove(v) - Três Casos Possíveis

- Podemos **remover** um número inteiro no BST executando uma operação semelhante a **Search(v)** .
- Se v não for encontrado na BST, simplesmente não fazemos nada.
- Caso v seja encontrado na BST, não informamos que o inteiro existente v foi encontrado, mas, em vez disso, realizamos um dos três possíveis casos de remoção que serão: (ver próximos slides)

8-1. Remover(v) - Caso 1 (nó folha)

- O primeiro caso é o mais fácil: o vértice v é atualmente um dos **vértices folha** da BST.
- A exclusão de um vértice folha é muito fácil: apenas removemos esse vértice folha — tente Remover(5) no exemplo BST acima (o segundo clique em diante após a primeira remoção não fará nada — atualize esta página ou vá para outro slide e volte para este slide).
- Esta parte é claramente $O(1)$ — além do esforço de busca anterior $O(h)$.

8-2. Remover(v) - Caso 2 - Nó interno com 1 filho

- O segundo caso também não é tão difícil: o vértice v é um vértice (interno/raiz) do BST e tem exatamente um filho . Remover v sem fazer mais nada desconectará o BST.
- A exclusão de um vértice com um filho não é tão difícil: conectamos o único filho desse vértice com o pai desse vértice - tente Remove(23) no exemplo BST acima (o segundo clique em diante após a primeira remoção não fará nada - atualize esta página ou vá para outro slide e retorne a este slide).
- Esta parte também é claramente $O(1)$ — além do esforço de busca anterior $O(h)$.

8-3. Remover(v) - Caso 3 – Exatamente 2 filhos

- O terceiro caso é o mais complexo dos três: o vértice v é um vértice (interno/raiz) da BST e **possui exatamente dois filhos**. Remover v sem fazer mais nada desconectará o BST.
- A exclusão de um vértice com dois filhos é a seguinte: Substituímos esse vértice por seu sucessor e, em seguida, excluímos seu sucessor duplicado em sua subárvore direita — tente Remove(6) no exemplo BST (de exemplo)
- Esta parte requer O(h) devido à necessidade de encontrar o vértice sucessor — além do esforço de busca O(h) anterior.

8-4. Remover(v) - Discussão do Caso 3

Este caso 3 merece mais discussões:

- Por que substituir um vértice B que tem dois filhos por seu sucessor C é sempre uma estratégia válida?
- Podemos substituir o vértice B que tem dois filhos por seu predecessor A? Por que ou por que não?

8-4. Remover(v) - Discussão do Caso 3

- Afirmamos que o vértice C, que é o sucessor do vértice B que tem dois filhos, deve ter no máximo um filho (o que é um caso de remoção mais fácil).
- O vértice B tem dois filhos, então B deve ter um filho certo. Vamos chamá-lo de R. O sucessor de B deve ser o vértice mínimo da subárvore com raiz em R. Lembre-se de que o elemento mínimo de uma subárvore em BST não tem filho à esquerda (pode ter filho à direita). Assim, C, o sucessor de B tem no máximo um filho.
- Antes da remoção, **temos X (pode estar vazio) $<$ A $<$ B $<$ C $<$ Z (pode estar vazio) no BST**. Substituir B por seu sucessor C e, em seguida, excluir o C antigo e duplicado manterá as propriedades BST de todos os vértices envolvidos. Da mesma forma, substituir B por seu predecessor A também alcançará o mesmo resultado. Só precisamos ser consistentes.

8-6. Complexidade de Tempo O(h)

- Remove(v) é executado em O(h) onde h é a altura do BST. Caso de remoção 3 (a exclusão de um vértice com dois filhos é a 'mais pesada', mas não é maior que O(h)).
-
- Como você deve ter entendido completamente até agora, h pode ser tão alto quanto O(N) em um BST normal, conforme mostrado no exemplo aleatório 'distorcido à direita' acima. Se chamarmos Remove(FindMax()) , ou seja, removermos o inteiro máximo atual, iremos da raiz até a última folha em tempo O(N) antes de removê-lo — não é eficiente.

Exercício

- Converter o código do arquivo **BSTDemo.cpp** para um **Projeto em Python**