Árvore de Pesquisa Binária

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). Tente clicar em Search(7) para obter um exemplo de animação na busca de um valor aleatório* $\in [1..99]$ *no BST aleatório acima.*

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)

Para alternar entre a Árvore de Pesquisa Binária padrão e a Árvore AVL (somente comportamento diferente durante a Inserção e Remoção de um Número Inteiro), selecione o respectivo cabeçalho.

Também temos um atalho de URL para acessar rapidamente o modo AVL Tree, que é https://visualgo.net/en/avl *(você pode alterar o 'en' para o idioma de sua preferência de dois caracteres - se disponível).*

3. Motivação

BST (e BST especialmente balanceado como AVL Tree) é uma estrutura de dados eficiente para implementar um certo tipo de ***tabela*** *(ou* ***mapa*** *) Abstract Data Type (ADT).*

Uma Tabela ADT deve suportar ***pelo menos*** *as três operações a seguir da forma mais eficiente possível:*

1. *Search(v) — determina se* v *existe no ADT ou não,*
2. *Insert(v) — insere* v *no ADT,*
3. *Remove(v) — remove* v *do ADT.*

Referência: Ver slide semelhante em Hash Table e-Lecture .

3-1. Que tipo de tabela ADT?

Estamos nos referindo à Tabela ADT onde as chaves precisam ser ordenadas (ao contrário da Tabela ADT onde as chaves não precisam ser desordenadas).

Este requisito especial da Tabela ADT ficará mais claro nos próximos slides.

3-2. Usando matriz/vetor não classificado

Se usarmos matriz/vetor ***não classificado*** *para implementar a Tabela ADT, pode ser ineficiente:*

1. *Search(v) roda em O(* N*), pois podemos acabar explorando todos os* N *elementos do ADT se* v *realmente não existir,*
2. *Insert(v) pode ser implementado em O(* 1 *), apenas coloque* v *no final do array,*
3. *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 classificado

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

1. *Search(v) agora pode ser implementado em O(log **N**), pois agora podemos usar **a pesquisa binária** no array classificado,*
2. *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,*
3. *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?

*O objetivo desta e-Lecture é apresentar a estrutura de dados BST e BST balanceada (Árvore AVL) para que possamos implementar as operações básicas da Table ADT: Search(v), Insert(v), Remove(v) e **algumas outras operações da Tabela ADT** — veja o próximo slide — em tempo O(log **N**) — que é muito menor que **N** .*

PS: Alguns dos leitores mais experientes podem notar que ∃ outra estrutura de dados pode implementar as três operações básicas do Table ADT em um tempo mais rápido, mas continue lendo...

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

3-5. Outras operações de tabela ADT

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

1. *Encontre o elemento Min()/Max(),*
2. *Encontre o elemento Sucessor(v) — 'próximo maior'/Predecessor(v) — 'menor anterior',*
3. *Listar elementos em ordem de classificação,*
4. *Operação X & Y - escondida para fins pedagógicos em um módulo NUS,*
5. *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. A solução

[Este é um slide oculto]

3-7. E quanto à lista encadeada?

A estrutura de dados mais simples que pode ser usada para implementar a Tabela ADT é a Lista Ligada .

*Questionário: **Podemos executar todas as três operações básicas do Table ADT: Pesquisar(v)/Inserir(v)/Remover(v) com eficiência (leia-se: mais rápido que O(N)) usando a Lista encadeada?***

- ○ *Sim*
- ○ *Não*

Discussão: Por quê?

3-8. A solução

[Este é um slide oculto]

3-9. E a tabela de hash?

*Outra estrutura de dados que pode ser usada para implementar a Table ADT é a Hash Table . Ele tem um desempenho muito rápido de Pesquisa(v), Inserção(v) e Remoção(v) (tudo no tempo O(**1**) esperado).*

*Questionário: Então, qual é o objetivo de aprender este módulo BST se a tabela de hash pode fazer as operações cruciais de tabela ADT em **tempo O(1) esperado improvável de ser superado ?***

- ○ *Existem razões válidas, que são ____*
- ○ *Não faz sentido, então este módulo BST pode ser ignorado*

Discuta a resposta acima! Dica: volte aos 4 slides anteriores.

3-10. A solução

[Este é um slide oculto]

4. Visualização

Vamos agora apresentar a estrutura de dados BST. Veja a visualização de um exemplo de BST acima!

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 de Vértice BST

*Cada vértice 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 acima, (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

*Como não permitimos números inteiros duplicados nesta visualização, 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 (essa visualização será atualizada com esse recurso em breve).

5. Operações BST

Fornecemos visualização para as seguintes operações comuns da árvore BST/AVL:

1. *Operações de consulta (a estrutura BST permanece inalterada):*
    1. *Pesquisa(v),*
    2. *Antecessor(v) (e similarmente Sucessor(v)), e*
    3. *Inorder/Preorder Traversal (iremos adicionar Postorder Traversal em breve),*
2. *Operações de atualização (a estrutura BST provavelmente pode mudar):*
    1. *Inserir(v),*
    2. *Remover(v) e*
    3. *Criar BST (vários critérios).*

5-1. Algumas outras operações BST

Existem algumas outras operações de BST (Consulta) que não foram visualizadas no VisuAlgo:

1. *Classificação(v): Dada uma chave* v *, determine qual é sua classificação (índice baseado em 1) na ordem classificada dos elementos BST. Ou seja, Rank(FindMin()) = 1 e Rank(FindMax()) =* N*. Se* v *não existir, podemos relatar -1.*
2. *Select(k): Dado um posto* k*,* $1 \leq k \leq N$*, 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, Select(1) = FindMin() e Select(* N*) = FindMax().*

Os detalhes dessas duas operações estão atualmente ocultos para fins pedagógicos em um determinado módulo NUS.

5-2. Estrutura de dados estática x dinâmica

Estrutura de dados que só é eficiente se não houver (ou rara) atualização, especialmente a(s) operação(ões) de inserção e/ou remoção é chamada de estrutura de dados ***estática*** *.*

A estrutura de dados que é eficiente mesmo se houver muitas operações de atualização é chamada de estrutura de dados ***dinâmica*** *. BST e BST especialmente balanceado (por exemplo, AVL Tree) estão nesta categoria.*

6. Pesquisa(v)

Devido à forma como os dados (números inteiros distintos para esta visualização) 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()

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 mostrado no exemplo aleatório 'distorcido à direita' acima. Tente 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)

*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:*

1. *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).*
2. *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).*
3. *Se **v** é o número inteiro máximo no BST, **v** não tem um sucessor. Tente Successor(71) (deve ser nenhum).*

7-1. Antecessor(v)

*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.

7-2. Complexidade de Tempo O(h)

*As operações predecessoras(v) e sucessoras(v) são executadas em O(**h**), onde **h** é a altura do BST.*

*Mas lembre-se de que esse **h** pode ser tão alto quanto O(**N**) em um BST normal, conforme mostrado no exemplo aleatório 'distorcido à direita' acima. Se chamarmos Successor(FindMax()) , iremos da última folha de volta para a raiz em tempo O(**N**) — não é eficiente.*

8. Travessia Inorder

*Podemos realizar um **Inorder Traversal** desta BST para obter uma lista de números inteiros ordenados dentro desta BST (na verdade, se 'achatarmos' a BST em uma linha, veremos que os vértices são ordenados do menor/mais à esquerda para o maior/mais à direita).*

Inorder Traversal é um método recursivo pelo qual visitamos a subárvore esquerda primeiro, esgotamos todos os itens da subárvore esquerda, visitamos a raiz atual, antes de explorar a subárvore direita e todos os itens da subárvore direita. Sem mais delongas, vamos tentar o Inorder Traversal para vê-lo em ação no exemplo BST acima.

8-1. Complexidade de Tempo O(N)

*Inorder Traversal é executado em O(**N**), independentemente da altura do BST.*

Discussão: Por quê?

*PS: Algumas pessoas chamam a inserção de **N** inteiros não ordenados em um BST em O(**N** log **N**) e, em seguida, executam o O(**N**) Inorder Traversal como ' **BST sort** '. Raramente é usado, pois existem vários algoritmos de classificação mais fáceis de usar (baseados em comparação) .*

8-2. A solução

[Este é um slide oculto]

8-3. Traversal de pré-ordem e pós-ordem

Incluímos a animação para Pré-encomenda, mas não fizemos o mesmo para o método de travessia da árvore Pós-ordem.

Basicamente, no Preorder Traversal, visitamos a raiz atual antes de ir para a subárvore esquerda e depois para a subárvore direita. Para o exemplo BST mostrado em segundo plano, temos: {{15}, {6, 4, 5, 7}, {23, 71, 50}}. PS: Você percebe o padrão recursivo? raiz, membros da subárvore esquerda da raiz, membros da subárvore direita da raiz.

No Postorder Traversal, visitamos primeiro a subárvore esquerda e a subárvore direita, antes de visitar a raiz atual. Para o exemplo BST mostrado em segundo plano, temos: {{5, 4, 7, 6}, {50, 71, 23}, {15}}.

9. Inserir(v)

*Podemos inserir um novo inteiro no BST fazendo uma operação semelhante a **Search(v)** . Mas desta vez, em vez de informar que o novo inteiro não foi encontrado, criamos um novo vértice no ponto de inserção e colocamos o novo inteiro lá. Tente Insert(60) no exemplo acima.*

9-1. Complexidade de Tempo O(h)

***Insert(v)** é executado em O(**h**) onde **h** é a altura do BST.*

*Até agora você deve estar ciente de que este **h** pode ser tão alto quanto O(**N**) em um BST normal, conforme mostrado no exemplo aleatório 'distorcido à direita' acima. Se chamarmos Insert(FindMax()+1) , ou seja, inserirmos um novo inteiro maior que o máximo atual, iremos da raiz até a última folha e então inseriremos o novo inteiro como filho à direita dessa última folha em O (**N**) tempo — não eficiente (observe que só permitimos até h=9 nesta visualização).*

9-2. Mini-questionário

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

- ○ *A altura não pode ser determinada*
- ○ *9*
- ○ *8*
- ○ *10*

Dica profissional: você pode usar o 'Modo de exploração' para verificar a resposta.

10. Remove(v) - Três Casos Possíveis

Podemos remover um número inteiro no BST executando uma operação semelhante a ***Search(v)*** *.*

Se ***v*** *não for encontrado no 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 elaborados em três slides separados (sugerimos que você tente cada um deles um por um).*

10-1. Remover(v) - Caso 1

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*** *).*

10-2. Remover(v) - Caso 2

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*** *).*

10-3. Remover(v) - Caso 3

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 acima (o segundo clique em diante após a primeira remoção fará nada - atualize esta página ou vá para outro slide e retorne a este slide).

Esta parte requer O(***h*** *) devido à necessidade de encontrar o vértice sucessor — além do esforço de busca O(* ***h*** *) anterior.*

10-4. Remover(v) - Discussão do Caso 3

Este caso 3 merece mais discussões:

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

10-5. A resposta

[Este é um slide oculto]

10-6. 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.*

11. Criar BST

Para facilitar a vida no 'Modo de Exploração', você pode criar um novo BST usando estas opções:

1. ***BST vazio** (você pode inserir alguns inteiros um por um),*
2. *Os dois **exemplos de e-Lecture** que você pode ter visto várias vezes até agora,*
3. ***Random** BST (que é improvável que seja extremamente alto),*
4. ***Skewed Left/Right** BST (BST alto com **N** vértices e **N-1** lista encadeada como arestas, para mostrar o comportamento de pior caso das operações BST; desabilitado no modo AVL Tree).*

12. Intermezzo

*Estamos no meio da explicação deste módulo BST. Até agora, notamos que muitas operações básicas da Table ADT executadas em O(**h**) **eh** podem ser tão altas quanto **N-1** arestas, como o exemplo de 'esquerda enviesada' mostrado - ineficiente :(...*

Então, existe uma maneira de fazer nossos BSTs 'não tão altos'?

PS: Se você quiser estudar como essas operações básicas do BST são implementadas em um programa real, você pode baixar este BSTDemo.cpp .

12-1. Experimente o modo de exploração

At this point, we encourage you to press [Esc] or click the X button on the bottom right of this e-Lecture slide to enter the 'Exploration Mode' and try various BST operations yourself to strengthen your understanding about this versatile data structure.

*When you are ready to continue with the explanation of **balanced** BST (we use **AVL Tree** as our example), press [Esc] again or switch the mode back to 'e-Lecture Mode' from the top-right corner drop down menu. Then, use the slide selector drop down list to resume from this slide 12-1.*

13. Balanced BST

*We have seen from earlier slides that most of our BST operations except Inorder traversal runs in O(**h**) where **h** is the height of the BST that can be as tall as **N-1**.*

*We will continue our discussion with the concept of **balanced BST** so that **h** = O(log **N**).*

13-1. AVL Tree

There are several known implementations of balanced BST, too many to be visualized and explained one by one in VisuAlgo.

*We focus on **AVL Tree** (Adelson-Velskii & Landis, 1962) that is named after its inventor: Adelson-Velskii and Landis.*

Other balanced BST implementations (more or less as good or slightly better in terms of constant-factor performance) are: Red-Black Tree, B-trees/2-3-4 Tree (Bayer & McCreight, 1972), Splay Tree (Sleator and Tarjan, 1985), Skip Lists (Pugh, 1989), Treaps (Seidel and Aragon, 1996), etc.

13-2. Extra BST Attribute: height(v)

*To facilitate AVL Tree implementation, we need to **augment** — add more information/attribute to — each BST vertex.*

*For each vertex **v**, we define **height(v)**: The number of edges on the path from vertex **v** down to its deepest leaf. This attribute is saved in each vertex so we can access a vertex's height in O(**1**) without having to recompute it every time.*

13-3. Formal Definition of height(v)

Formally:

```
v.height = -1 (se v for uma árvore vazia)
v.height = max(v.left.height, v.right.height) + 1 (caso contrário)
```

A altura do BST é assim:`raiz.altura`*.*

No exemplo BST acima, altura(11) = altura(32) = altura(50) = altura(72) = altura(99) = 0 (todas são folhas). height(29) = 1, pois há 1 aresta conectando-a à sua única folha 32.

13-4. Mini-questionário

*Questionário: **Quais são os valores de altura(20), altura(65) e altura(41) no BST acima?***

- ☐ *altura(41) = 3*
- ☐ *altura(20) = 2*
- ☐ *altura(20) = 3*
- ☐ *altura(41) = 4*
- ☐ *altura(65) = 2*
- ☐ *altura(65) = 3*

13-5. O limite inferior da altura BST

*If we have **N** elements/items/keys in our BST, the lower bound height **h** > $\log_2$ **N** if we can somehow insert the **N** elements in perfect order so that the BST is perfectly balanced.*

*See the example shown above for **N = 15** (a perfect BST which is rarely achievable in real life — try inserting any other integer and it will not be perfect anymore).*

13-6. Derivation of the Lower Bound

$N \leq 1 + 2 + 4 + ... + 2^h$
$N \leq 2^0 + 2^1 + 2^2 + \ldots + 2^h$
$N < 2^{h+1}$ (sum of geometric progression)
$\log_2 N < \log_2 2^{h+1}$
$\log_2 N < (h+1) * \log_2 2$ ($\log_2 2$ is 1)
$h > (\log_2 N)-1$ (algebraic manipulation)
$h > \log_2 N$

13-7. The Upper Bound of BST Height

*If we have **N** elements/items/keys in our BST, the upper bound height **h** < **N** if we insert the elements in ascending order (to get skewed right BST as shown above).*

The height of such BST is `h = N-1`*, so we have* `h < N`.

Discussion: Do you know how to get skewed left BST instead?

13-8. The Solution

[This is a hidden slide]

13-9. The Combined Bound

*We have seen that most BST operations are in O(**h**) and combining the lower and upper bounds of **h**, we have $\log_2$ **N** < **h** < **N**.*

*There is a dramatic difference between $\log_2$ **N** and **N** and we have seen from the discussion of the lower bound that getting perfect BST (at all times) is near impossible...*

*So can we have BST that has height closer to $\log_2$ **N**, i.e. **c** * $\log_2$ **N**, for a small constant factor **c**? If we can, then BST operations that run in O(**h**) actually run in O(log **N**)...*

14. AVL Tree

Introducing AVL Tree, invented by two Russian (Soviet) inventors: Georgy Adelson-Velskii and Evgenii Landis, back in 1962.

*In AVL Tree, we will later see that its height **h** < 2 * log **N** (tighter analysis exist, but we will use easier analysis in VisuAlgo where **c = 2**). Therefore, most AVL Tree operations run in O(log **N**) time — efficient.*

*Insert(v) and Remove(v) update operations may change the height **h** of the AVL Tree, but we will see **rotation** operation(s) to maintain the AVL Tree height to be low.*

14-1. Step 1: Maintaining height(v) Efficiently

*To have efficient performance, we shall not maintain **height(v)** attribute via the O(**N**) recursive method every time there is an update (Insert(v)/Remove(v)) operation.*

*Instead, we compute O(**1**):* `x.height = max(x.left.height, x.right.height) + 1` *at the back of our Insert(v)/Remove(v) operation as only the height of vertices along the insertion/removal path may be affected. Thus, only O(**h**) vertices may change its **height(v)** attribute and in AVL Tree, **h** < 2 * log **N**.*

Try Insert(37) on the example AVL Tree (ignore the resulting rotation for now, we will come back to it in the next few slides). Notice that only a few vertices along the insertion path: {41,20,29,32} increases their height by +1 and all other vertices will have their heights unchanged.

14-2. Step 2: Define AVL Tree Invariant

Let's define the following important AVL Tree invariant (property that will never change): A vertex ***v*** *is said to be* ***height-balanced*** *if* `|v.left.height - v.right.height| ≤ 1`.

A BST is called height-balanced according to the invariant above if every vertex in the BST is height-balanced. Such BST is called AVL Tree, like the example shown above.

Take a moment to pause here and try inserting a few new random vertices or deleting a few random existing vertices. Will the resulting BST still considered height-balanced?

14-3. Proof - 1

Adelson-Velskii and Landis claim that an AVL Tree (a height-balanced BST that satisfies AVL Tree invariant) with N *vertices has height* $h < 2 * \log_2 N$.

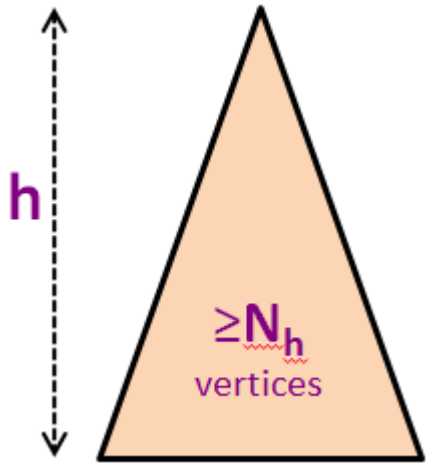
The proof relies on the concept of minimum-size AVL Tree of a certain height h.

Let N_h *be the minimum number of vertices in a height-balanced AVL Tree of height* h.

The first few values of N_h *are* $N_0 = 1$ *(a single root vertex),* $N_1 = 2$ *(a root vertex with either one left child or one right child only),* $N_2 = 4$, $N_3 = 7$, $N_4 = 12$, $N_5 = 20$ *(see the background picture), and so on (see the next two slides).*

14-4. Proof - 2

We know that for any other AVL Tree of N *vertices (not necessarily the minimum-size one), we have* $N \geq N_h$.

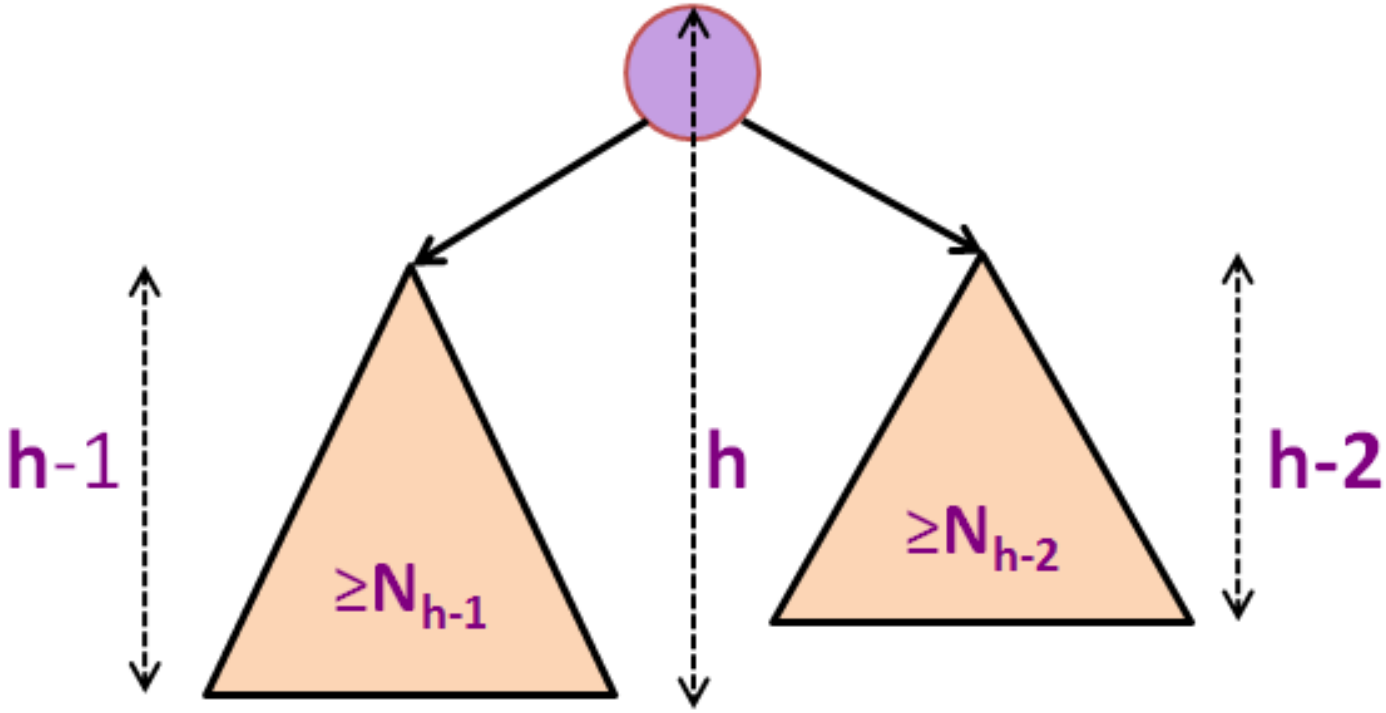


In the background picture, we have $N_5 = 20$ *vertices but we know that we can squeeze 43 more vertices (up to* $N = 63$*) before we have a perfect binary tree of height* $h = 5$.

14-5. Proof - 3

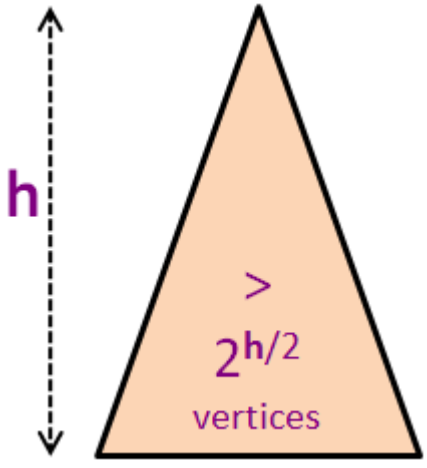
$N_h = 1 + N_{h-1} + N_{h-2}$ (formula for minimum-size AVL tree of height h)
$N_h > 1 + 2*N_{h-2}$ (as $N_{h-1} > N_{h-2}$)
$N_h > 2*N_{h-2}$ (obviously)
$N_h > 4*N_{h-4}$ (recursive)

$N_h > 8*N_{h-6}$ (another recursive step)
... (we can only do this h/2 times, assuming initial h is even)
$N_h > 2^{h/2}*N_0$ (we reach base case)
$N_h > 2^{h/2}$ (as $N_0 = 1$)



14-6. Proof - 4

$N \geq N_h > 2^{h/2}$ (combining the previous two slides)
$N > 2^{h/2}$
$\log_2(N) > \log_2(2^{h/2})$ ($\log_2$ on both sides)
$\log_2(N) > h/2$ (formula simplification)
$2 * \log_2(N) > h$ or $h < 2 * \log_2(N)$
$h = O(\log(N))$ (the final conclusion)



14-7. Step 3: Maintain Invariant

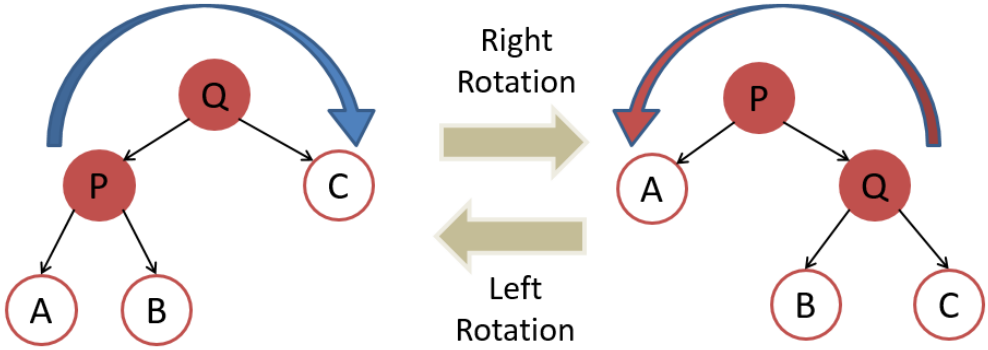
Look at the example BST again. See that all vertices are height-balanced, an AVL Tree.

To quickly detect if a vertex ***v*** *is height balanced or not, we modify the AVL Tree invariant (that has absolute function inside) into:* `bf(v) = v.left.height - v.right.height`.

Now try Insert(37) on the example AVL Tree again. A few vertices along the insertion path: {41,20,29,32} increases their height by +1. Vertices {29,20} will no longer be height-balanced after this insertion (and will

be rotated later — discussed in the next few slides), i.e. bf(29) = -2 and bf(20) = -2 too. We need to restore the balance.

14-8. Introducing Tree Rotation



*See the picture above. Calling **rotateRight(Q)** on the left picture will produce the right picture. Calling **rotateLeft(P)** on the right picture will produce the left picture again.*

***rotateRight(T)/rotateLeft(T)** can only be called if **T** has a left/right child, respectively.*

*Tree Rotation **preserves** BST property. Before rotation, $P \leq B \leq Q$. After rotation, notice that subtree rooted at B (if it exists) changes parent, but $P \leq B \leq Q$ does not change.*

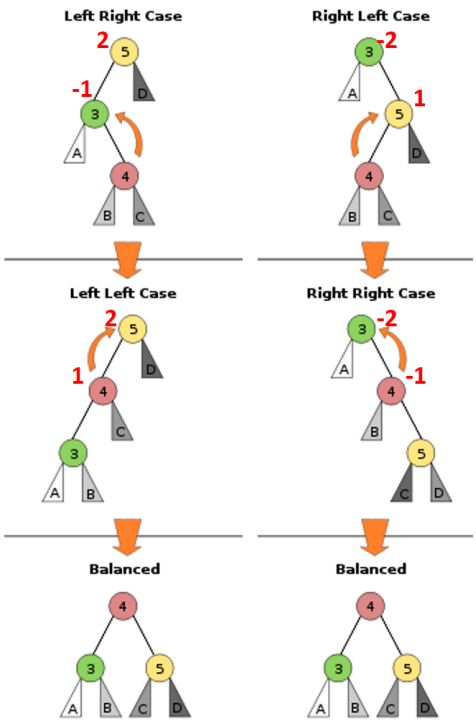
14-9. Non-trivial O(1) Tree Rotation Pseudo-code

```
BSTVertex rotateLeft(BSTVertex T) // pre-req: T.right != null
  BSTVertex w = T.right // rotateRight is the mirror copy of this
  w.parent = T.parent // this method is hard to get right for newbie
  T.parent = w
  T.right = w.left
  if (w.left != null) w.left.parent = T
  w.left = T
  // update the height of T and then w here
  return w
```

14-10. Four Rebalancing Cases

Four Possible Cases

bf(x) = +2 and bf(x.left) = 1
  rightRotate(x)
bf(x) = +2 and bf(x.left) = -1
  leftRotate(x.left)
  rightRotate(x)
bf(x) = -2 and bf(x.right) = -1
  leftRotate(x)
bf(x) = -2 and bf(x.right) = 1
  rightRotate(x.right)
  leftRotate(x)



Pictures from Wikipedia

Basically, there are only these four imbalance cases. We use Tree Rotation(s) to deal with each of them.

14-11. Insert(v) in AVL Tree

1. *Just insert **v** as in normal BST,*
2. *Walk up the AVL Tree from the insertion point back to the root and at every step, we update the height and balance factor of the affected vertices:*
    a. *Stop at the **first** vertex that is out-of-balance (+2 or -2), if any,*
    b. *Use **one** of the four tree rotation cases to rebalance it again, e.g. try Insert(37) on the example above and notice by calling **rotateLeft(29)** once, we fix the imbalance issue.*

Discussion: Is there other tree rotation cases for Insert(v) operation of AVL Tree?

14-12. The Answer

[This is a hidden slide]

14-13. Remove(v) in AVL Tree

1. *Just remove **v** as in normal BST (one of the three removal cases),*
2. *Walk up the AVL Tree from the deletion point back to the root and at every step, we update the height and balance factor of the affected vertices:*
    1. *Now for **every** vertex that is out-of-balance (+2 or -2), we use **one** of the four tree rotation cases to rebalance **them** (can be more than one) again.*

*The main difference compared to Insert(v) in AVL tree is that we may trigger one of the four possible rebalancing cases **several times**, but not more than **h** = O(log **N**) times :O, try Remove(7) on the example above to see two chain reactions **rotateRight(6)** and then **rotateRight(16)+rotateLeft(8)** combo.*

14-14. AVL Tree Summary

*We have now see how AVL Tree defines the height-balance invariant, maintain it for all vertices during Insert(v) and Remove(v) update operations, and a proof that AVL Tree has **h** < 2 * log **N**.*

*Therefore, **all** BST operations (both update and query operations except Inorder Traversal) that we have learned so far, if they have time complexity of O(**h**), they have time complexity of O(log **N**) if we use AVL Tree version of BST.*

This marks the end of this e-Lecture, but please switch to 'Exploration Mode' and try making various calls to Insert(v) and Remove(v) in AVL Tree mode to strengthen your understanding of this data structure.

PS: If you want to study how these seemingly complex AVL Tree (rotation) operations are implemented in a real program, you can download this AVLDemo.cpp (must be used together with this BSTDemo.cpp).

15. Extras

We will end this module with a few more interesting things about BST and balanced BST (especially AVL Tree).

15-1. Those 2 Extra BST Operations

[This is a hidden slide]

15-2. Side Usage of Balanced BST?

[This is a hidden slide]

15-3. Online Quiz

For a few more interesting questions about this data structure, please practice on BST/AVL training module (no login is required).

However, for registered users, you should login and then go to the Main Training Page to officially clear this module and such achievement will be recorded in your user account.

15-4. Online Judge Exercises

We also have a few programming problems that somewhat requires the usage of this ***balanced*** *BST (like AVL Tree) data structure: Kattis - compoundwords and Kattis - baconeggsandspam.*

Try them to consolidate and improve your understanding about this data structure. You are allowed to use C++ STL map/set, Java TreeMap/TreeSet, or OCaml Map/Set if that simplifies your implementation (Note that Python doesn't have built-in bBST implementation).

15-5. The Solution

[This is a hidden slide]