CSCI 136:
Data Structures
and
Advanced Programming

Lecture 23

Trees, part 3

Instructors: Dan & Bill

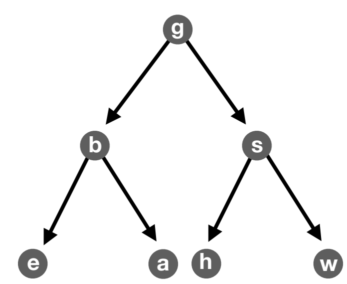
Williams

Outline

- Tree balance
- Big-O
- Implicit BST

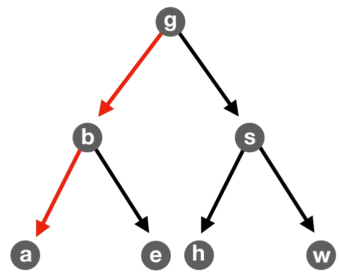
Tree balance

In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter e.

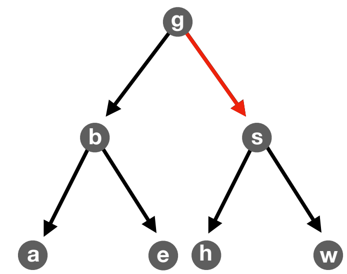
In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter a.

Finding a takes **two steps**.

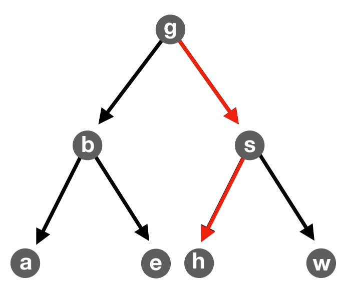
In the **worst case**, how long does it take to find an element in this binary search tree?



Suppose it is the letter s.

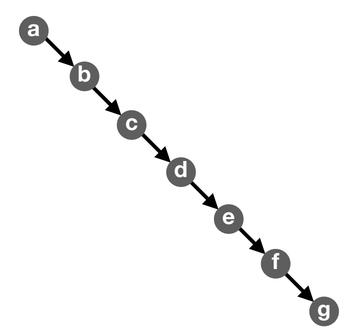
Finding s takes **one step**.

In the **worst case**, how long does it take to find an element in this binary search tree?



In the **worst case**, the time depends on the **length** of the **longest path**.

Suppose a friend gives you the following sequence of values: `[a,b,c,d,e,f,g]`



Ouch!!!

Worst case:
O(n)

And asks you to store them in a binary tree to "make accessing them fast."

Is access **guaranteed** to be **fast**?

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

`isBalanced(t):`

`t` is balanced if and only if

- `t` is empty, or
- **all** of the following
  - `isBalanced(t.left)` is `true` **and**
  - `isBalanced(t.right)` is `true` **and**
  - `|height(t.left) - height(t.right) |≤ 1`

Keep in mind: we know that the worst case has something to do with **height**.

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

Clearly a balanced tree.

Yeah, sure, there's no tree. Details, details...

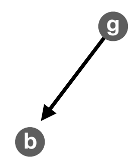
Time to access an element ~ **0 steps**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

Balanced? **Yes.**

Max time to access an element ~ **0 steps**

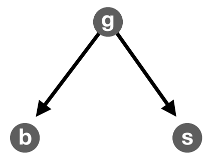
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **1 step**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)

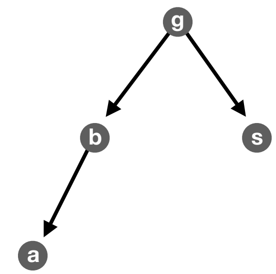


Balanced? **Yes.**

Changes nothing.

Max time to access an element: **1 step**

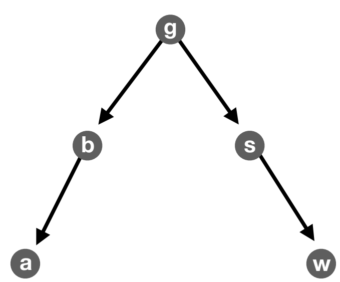
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

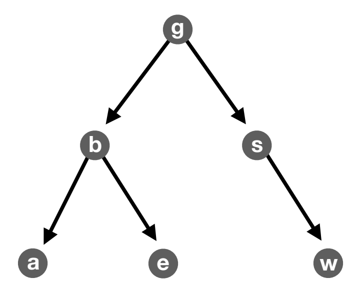
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

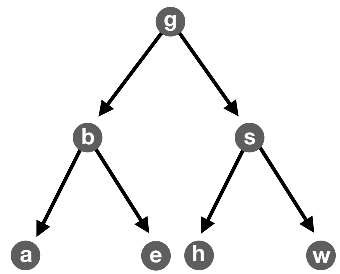
But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

Max time to access an element: **2 steps**

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



Balanced? **Yes.**

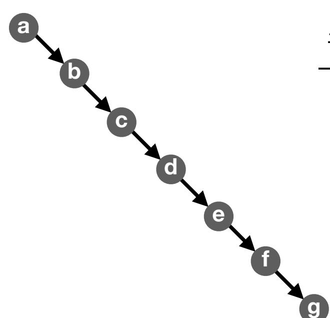
Max time to access an element: **2 steps**

# nodes	max time
1	**0 steps**
2	**1 step**
3	**1 step**
4	**2 steps**
5	**2 steps**
6	**2 steps**
7	**2 steps**
8	**3 steps**
...	**...**

This looks like **time** = $\log_2$(# nodes)

But does this hold up?

But what if your tree maintained the following property **on insertion**? (i.e., it is always true)



# nodes	max time
7	**6 steps**

Clearly **not** a balanced tree.

Logarithmic worst-case access time has something to do with the **compactness** of a tree; **height matters**.

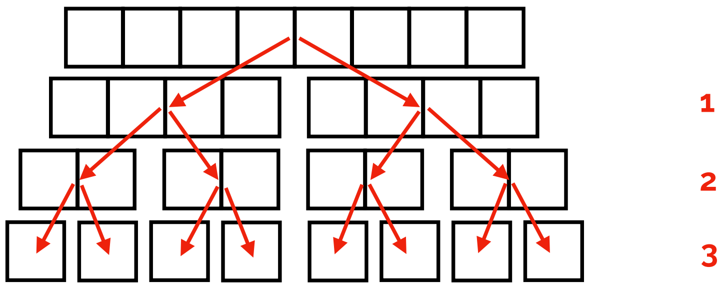
BST Big-O

Worst case time is **$O(\log_2(n))$** for a **balanced binary tree**.

Why?

What is min. binary tree height needed to store **n** nodes?

Cute theorem: **height ≥ $\lfloor \log_2(n) \rfloor$**



Intuition: $\log_2(n)$ is the number of times you can **divide n nodes in halves.**

Implicit Data Structures

Recall: binary search tree

A **binary search tree** is a binary tree that maintains the **binary search property** as elements are added or removed. In other words, the **key** in each node:

- must be ≥ any **key** stored in the left subtree, and
- must be ≤ any **key** stored in the right subtree.

As with other ordered structures, order is maintained **on insertion.**

BST is an ADT

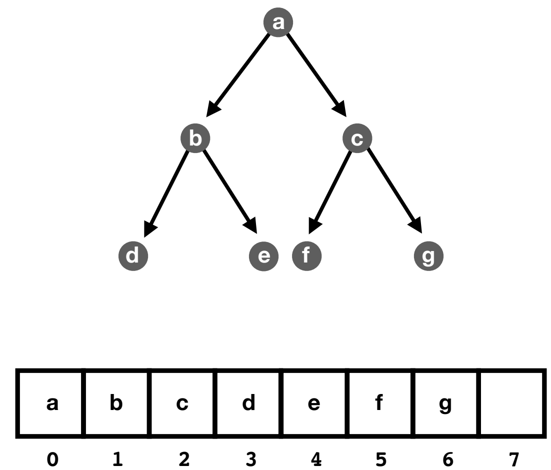
Do we actually need a **tree** to store a **tree**?

No. We can use an **implicit data structure** instead.

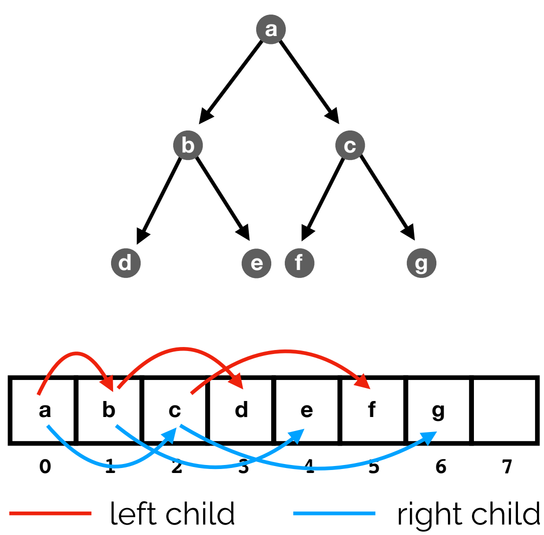
Implicit data structure

A **implicit data structure** or **space-efficient data structure** is a data structure that stores only **necessary** information. Instead of explicitly representing relationships between elements of the structure using references, an implicit structure **uses the relative positions of elements**.

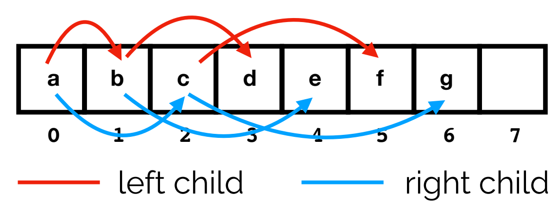
Implicit binary tree



Implicit binary tree



Implicit relationship



$$\text{leftChild}(i) = 2 \times i + 1$$

$$\text{rightChild}(i) = 2 \times i + 2$$

$$\text{parent}(i) = \left\lfloor (i - 1) / 2 \right\rfloor$$

Implicit Binary Search Tree

Let's implement an implicit BST.

Recap & Next Class

This lecture:

Tree balance

Big-O

Implicit BST

Next lecture:

Priority queues

Heaps