MSiA-413 Introduction to Databases and Information Retrieval

Lecture 10
Indexes

Instructor: Nikos Hardavellas

Slides adapted from Steve Tarzia, Silberschatz

Last Lecture: introduced the *storage hierarchy*

Larger, but slower

delay		*capacity*
0.3ns	CPU Registers	1 kB (kilobyte)
5ns	CPU Caches (L2)	16 MB
50ns	Random Access Memory (RAM)	16 GB
100µs	Flash Storage (SSD)	1 TB
5ms	Magnetic Disk	8 TB

- Disk is about *ten billion* times larger than registers, but has about *ten million* times larger delay (latency)
- Goal is to work as much as possible in the top levels
- Large, rarely-needed data is stored at the bottom level

Sorting and Binary Search

- We know it's easy to find data if it's in a *sorted* list
  - That's why printed dictionaries and phone books are alphabetical
- **Binary Search** is how computers find entries in a sorted list
  - Let's say you're looking for the word "key" in a list of 10,000 words
    1. Compare "key" to the word in the middle position (5,000th word)
    2. If you're lucky and that middle word is *equal to* "key", then you're done!
    3. If the middle word is *greater than* "key" then go back to step 1, but use just the left half of the list (words 0 through 4,999)
    4. If the middle word is *less than* "key" then go back to step 1, but use just the right half of the list (words 5,001 through 10,000)
  - It will take at most $\log_2 N$ steps to find the entry, where N is the list size
    - e.g., 32 steps for binary search in a list of 4 billion (because $2^{32} \cong$ 4 billion)

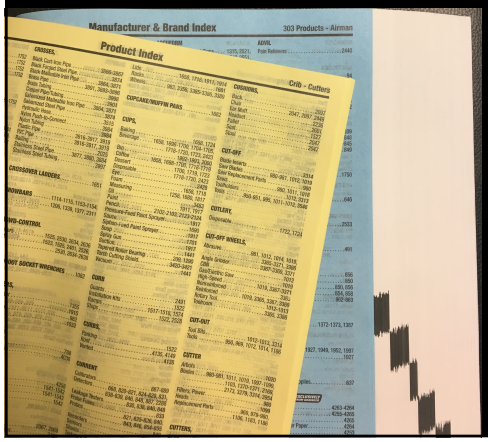
Why sorting is not enough

1. You can't *insert new data* without shuffling everything over to make room

2. You can't sort in *multiple dimensions*
   - Let's say you want to find a product quickly according to either it's name, manufacturer, or price. You can only sort by one of the there three columns

3. It doesn't take advantage of the hardware's storage hierarchy
   - The binary search will have to access the disk in every step because the index is distributed over the full data set
   - It would be better to put all the index data close together

A printed catalog handles multiple indexes by adding more pages



- Real-life example: Grainger catalog
- White pages: data (unsorted)
- Additional yellow pages
  - Index of products
  - Sorted by product type name
- Additional blue pages
  - Index of manufacturers
  - Sorted by manufacturer name
- Easy to locate products and manufacturers
  - …without sorting the data in either dimension

Index data use the storage hierarchy more effectively

- All of the index data may fit in a higher level of the storage hierarchy
- For example, might be able to fit this one column entirely in RAM and avoid using magnetic disk (*ns* instead of *ms* access latencies)
- Arrows mean that you're storing a *reference* to another storage location

In an ideal world:

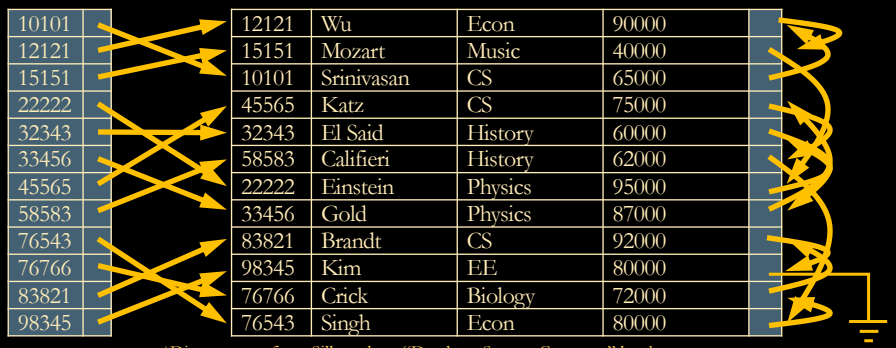
10101	10101	Srinivasan	CS	65000
12121	12121	Wu	Econ	90000
15151	15151	Mozart	Music	40000
22222	22222	Einstein	Physics	95000
32343	32343	El Said	History	60000
33456	33456	Gold	Physics	87000
45565	45565	Katz	CS	75000
58583	58583	Califieri	History	62000
76543	76543	Singh	Econ	80000
76766	76766	Crick	Biology	72000
83821	83821	Brandt	CS	92000
98345	98345	Kim	EE	80000

*Diagrams are from Silberschatz "Database System Concepts" book

Index data use the storage hierarchy more effectively

- All of the index data may fit in a higher level of the storage hierarchy
- For example, might be able to fit this one column entirely in RAM and avoid using magnetic disk (*ns* instead of *ms* access latencies)
- Arrows mean that you're storing a *reference* to another storage location

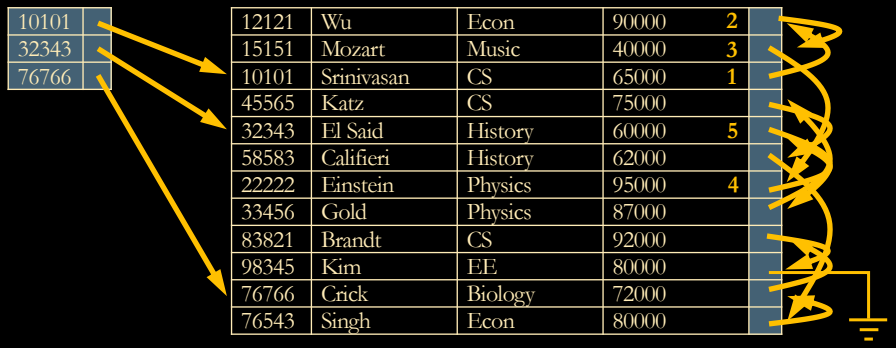
In the real world:



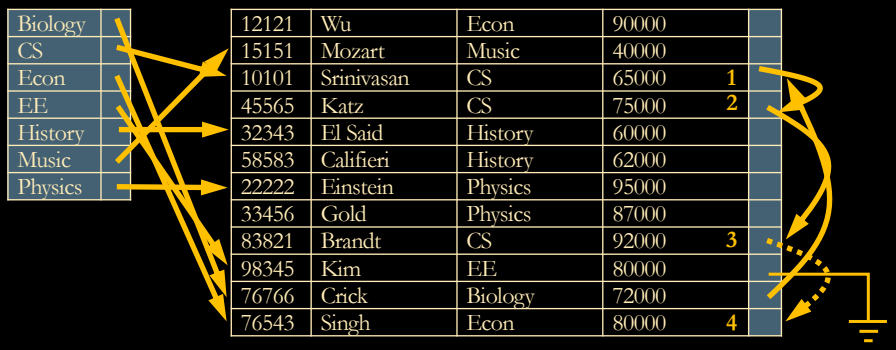
*Diagrams are from Silberschatz "Database System Concepts" book

A ***Sparse*** Index may fit in an even higher storage level

- However, we have to do some one-by-one scanning to find the particular row of interest

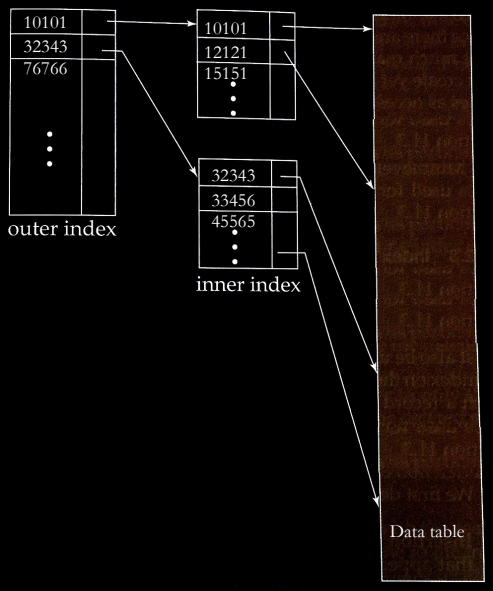


Sparse Index can be used for repeated values

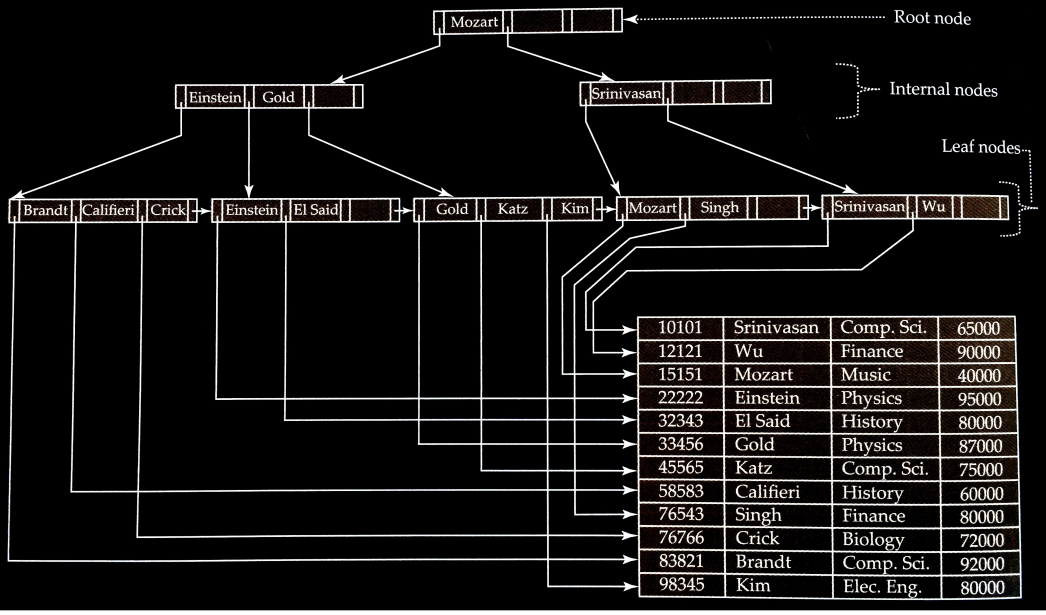


Two-level Index gives us the best of both

- The outer level is small and fits in fast memory, like the sparse index
- The inner level is larger but still fits in a faster memory level than main data, and it tells us exactly where to find data. No need to scan one-by-one



B+Trees generalize the concept of multi-level indexes



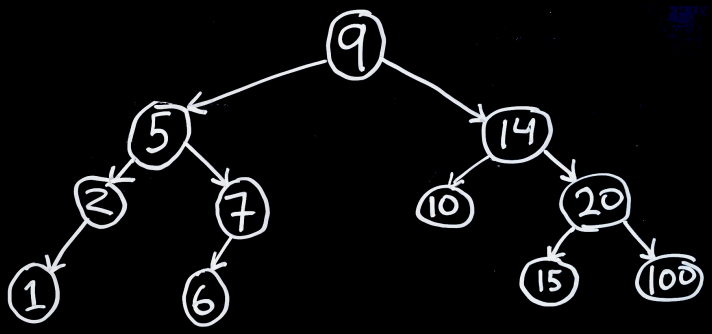
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Detour: Binary Search Tree

- A "Data Structure" that allows you to search for items quickly.
- Instead of sorting data like this:

1	2	5	6	7	9	10	14	15	20	100

- Organize data into a tree like this:

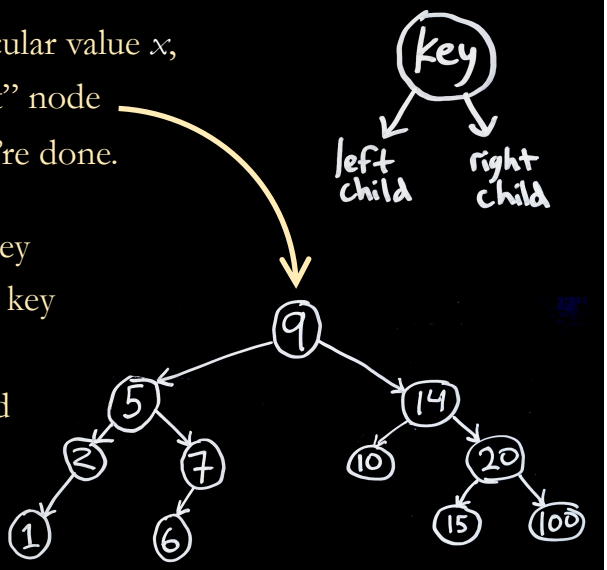


Binary Search Tree mechanics

Given that you are searching for a particular value x,

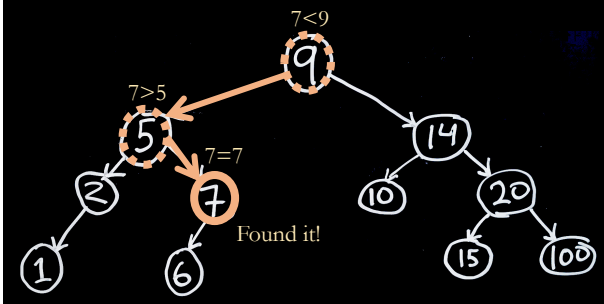
1. Always start your search at the "root" node
2. If this node's key equals x, then you're done. You have found it!
3. Move to the *left child** if $x <$ node's key
4. Move to the *right child** if $x >$ node's key
5. Go to step 2

* In steps 3 & 4, if there is no such child then the key is absent

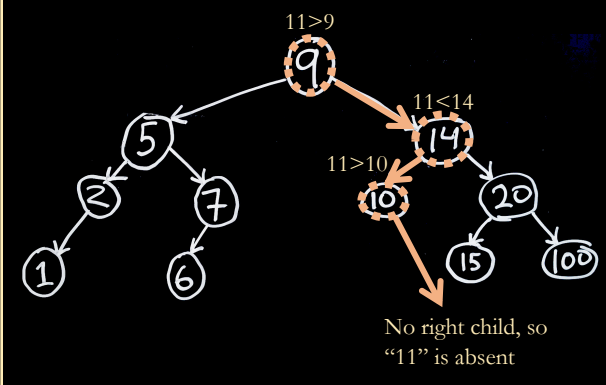


BST Search Examples

Searching for "7":



Searching for "11"



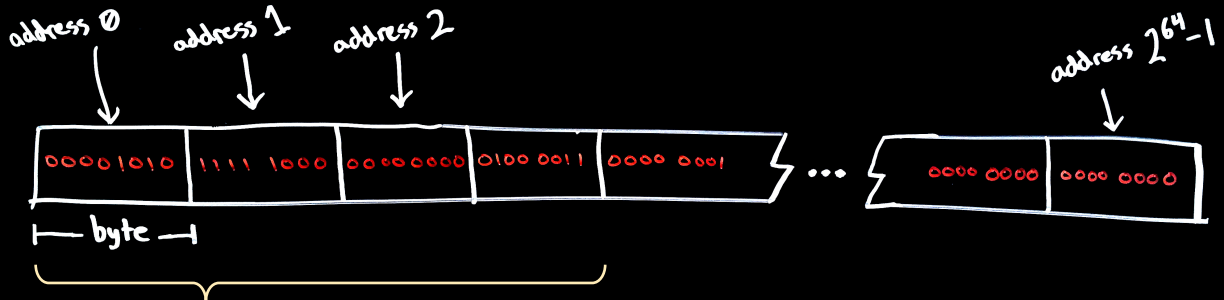
Building a Binary Search Tree

- Works just like search
- Find where key *should be* then add it

So, let's talk about memory…

Computer memory is basically just a big array

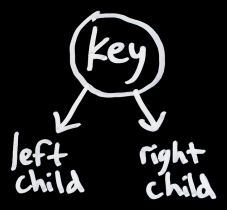
- All data is stored as a sequence of zeros and ones
- Central Processing Unit (CPU) specifies which *address* to read and write
  - Addresses refer to a byte-sized sequence of bits (8 bits)

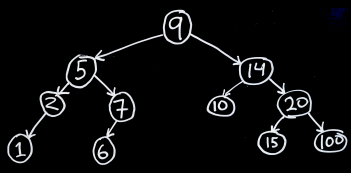


These 4 memory addresses could represent many different things.
For example, a single 32-bit float, or a few characters of UTF-8 text.

A Tree is a type of "graph"
A collection of nodes referring to other nodes

Address	Key	Left Child	Right Child
101	**20**	310	105
102	**5**	104	103
103	**7**	108	*NULL*
104	**2**	107	*NULL*
105	**100**	*NULL*	*NULL*
106	**9**	102	891
107	**1**	*NULL*	*NULL*
108	**6**	*NULL*	*NULL*
…			
310	**15**	*NULL*	*NULL*
311	**10**	*NULL*	*NULL*
…			
891	**14**	311	101

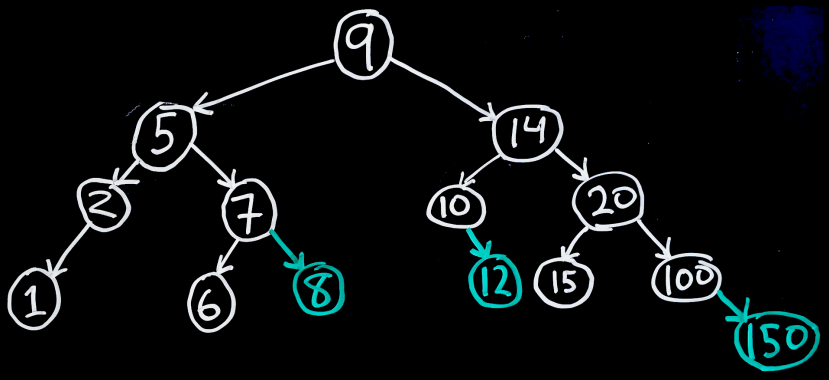




- In this case the root is at address 106
- It's a *linked* data structure
  - Allows us to use a one-dimensional array of computer memory to represent something with complex geometry
  - The tree can be reshaped very quickly by just changing a couple of child values

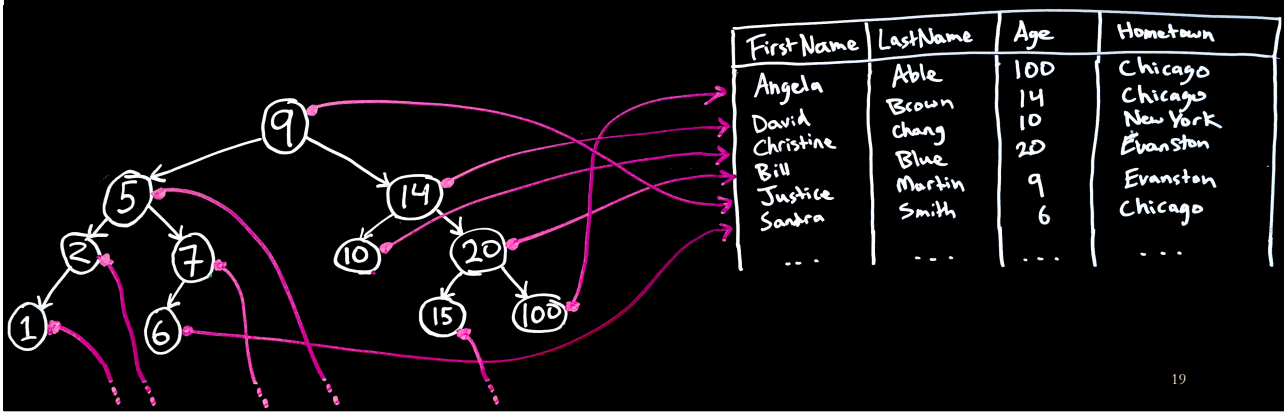
Quick insertions and deletions in a tree

- There is no need to shift all data, just change a few nodes

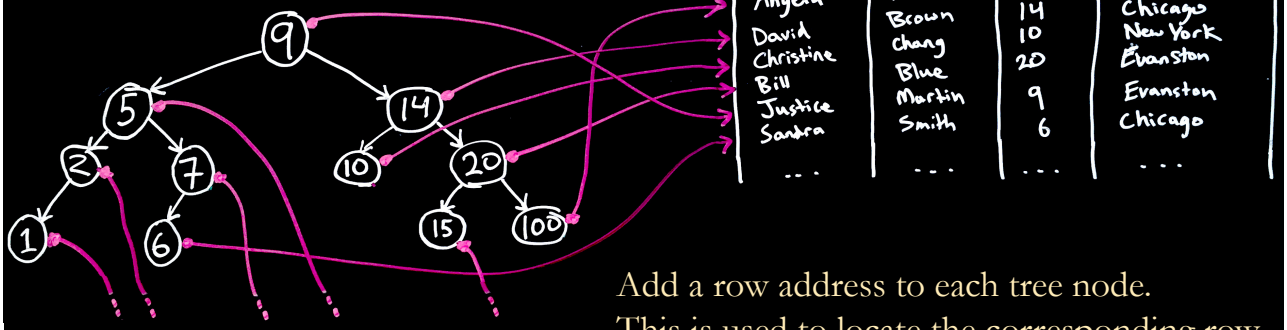


Using a BST as an index

- Add one piece of additional info to each node – a row address
- Below, a BST is an index on a table's *Age* column
- Search for an age in the tree, then the tree node directs you to a row



In more detail



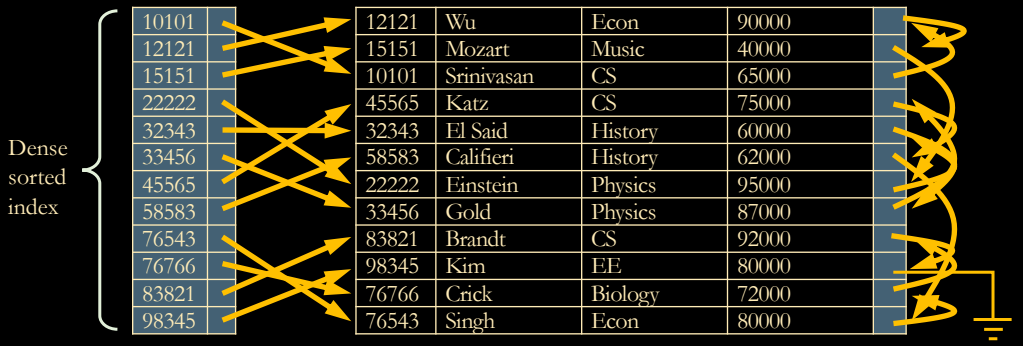
Add a row address to each tree node.
This is used to locate the corresponding row.

Tree representation in memory:

Address	Key	Left Child	Right Child	Row Address
101	20	310	105	**10500**
102	5	104	103	**10200**
103	7	108	*NULL*	**10300**
104	2	107	*NULL*	**20200**
105	100	*NULL*	*NULL*	**20800**

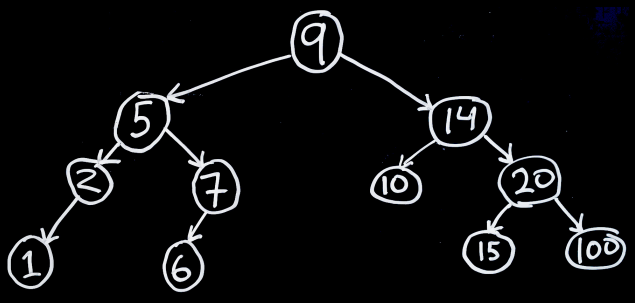
Reminder: Dense sorted index

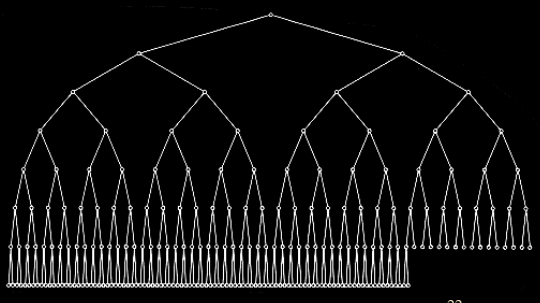
- Cannot add or delete items without shifting data
- Might think about leaving some open space, but it's impossible to predict exactly where space will be needed, and this will change over time



Searching a BST can be fast if you eliminate half of the remaining options in each step

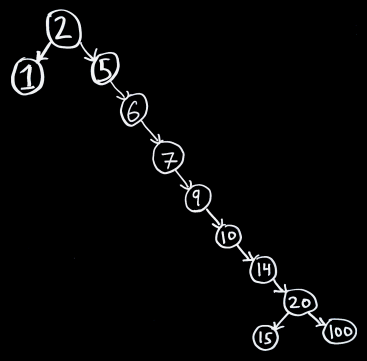
- Like Binary Search of a sorted list, x steps lets you scan 2^x data values
- Each additional level can index twice as much data





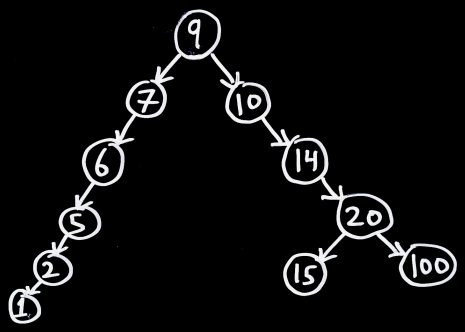
But an *unbalanced* tree is very inefficient

- In the worst case, you would look at every node to find what you're looking for
- The problem is that each step is not eliminating much candidate data
  - The "path not taken" is empty



Maintaining balance in a tree

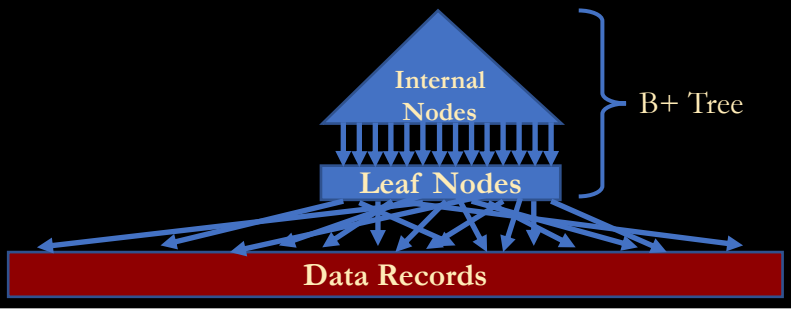
- It's not enough to just choose your root node wisely.
  This tree is better, but still not well balanced:



- *Every node* must have a value near the middle of all its descendants

Computer scientists have invented many different self-balancing trees:

- For example: red-black trees, AVL trees, splay trees, and b-trees
- They all require some quick maintenance work to restore balance after each insertion or deletion.
- Databases mostly use B^+-trees for indexes
  - We won't go too deep into the details of B^+-trees in this course



The Database Reality: B+Tree Indexes