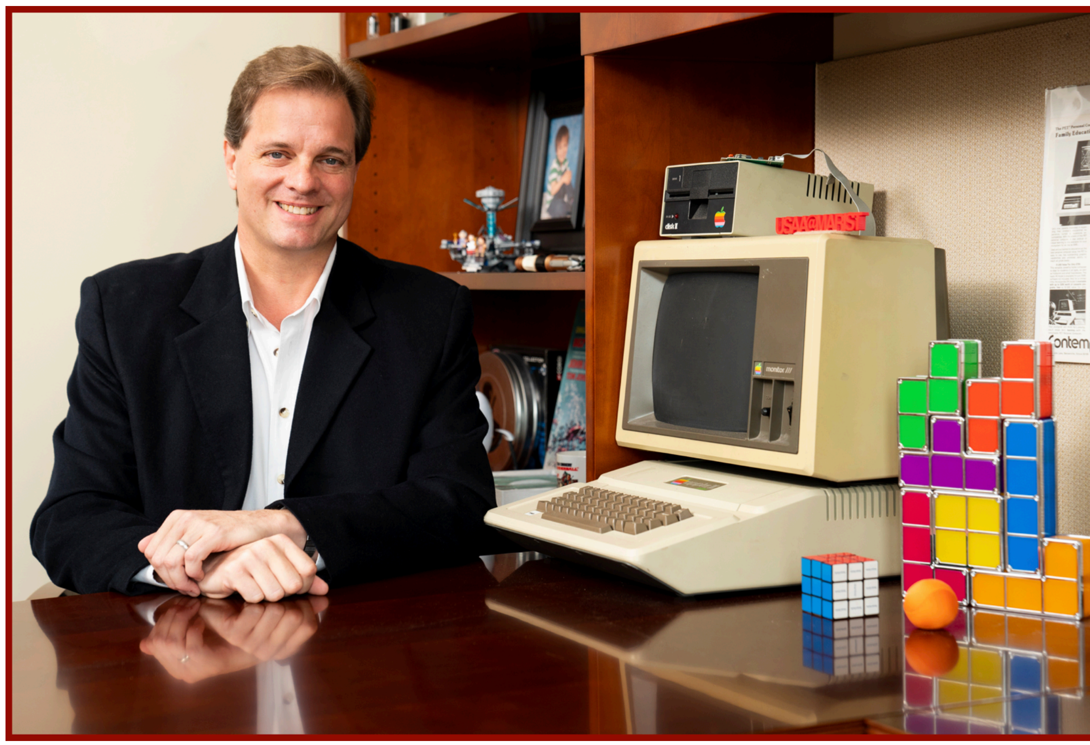
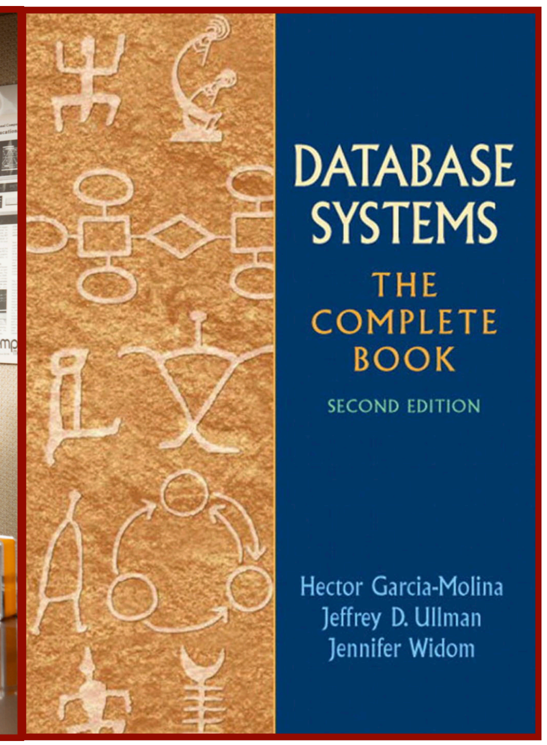
Indexes and Index Structures





Alan G. Labouseur, Ph.D.
Alan.Labouseur@Marist.edu

Our CAP Database

There's not much data in our beloved CAP database.
What if there were more? A lot more?
Like **9 billon** people rather than just 9 ?

People

pid	prefix	firstName	lastName	suffix	homeCity	DOB
1	Dr.	Neil	Peart	Ph.D.	Toronto	1952-09-12
2	Ms.	Regina	Schock		Toronto	1957-08-31
3	Mr.	Bruce	Crump	Jr.	Jacksonville	1957-07-17
4	Mr.	Todd	Sucherman		Chicago	1969-05-02
5	Mr.	Bernard	Purdie		Teaneck	1939-06-11
6	Ms.	Demetra	Plakas	Esq.	Santa Monica	1960-11-09
7	Ms.	Terri Lyne	Carrington		Boston	1965-08-04
8	Dr.	Bill	Bruford	Ph.D.	Kent	1949-05-17
9	Mr.	Alan	White	III	Pelton	1949-06-14

Products

prodId	name	city	qtyOnHand	priceUSD
p01	Heisenberg Compensator	Dallas	47	67.50
p02	Universal Translator	Newark	2399	5.50
p03	Commodore PET	Duluth	1979	65.02
p04	LCARS module	Duluth	3	47.00
p05	Remo drumhead	Dallas	8675309	16.61
p06	Trapper Keeper	Dallas	1982	2.00
p07	Flux Capacitor	Newark	1007	1.00
p08	HAL 9000 memory core	Newark	200	1.25
p09	Red Barchetta	Toronto	1	379000.47

Customers

pid	paymentTerms	discountPct
1	Net 30	21.12
4	Net 15	4.04
5	In Advance	5.50
7	On Receipt	2.00
8	Net 30	10.00

Agents

pid	paymentTerms	commissionPct
2	Quarterly	5.00
3	Annually	10.00
5	Monthly	2.00
6	Weekly	1.00

Orders

orderNum	dateOrdered	custId	agentId	prodId	quantityOrdered	totalUSD
1011	2020-01-23	1	2	p01	1100	58568.40
1012	2020-01-23	4	3	p03	1200	74871.83
1015	2020-01-23	5	3	p05	1000	15696.45
1016	2020-01-23	8	3	p01	1000	60750.00
1017	2020-02-14	1	3	p03	500	25643.89
1018	2020-02-14	1	3	p04	600	22244.16
1019	2020-02-14	1	2	p02	400	1735.36
1020	2020-02-14	4	5	p07	600	575.76
1021	2020-02-14	4	5	p01	1000	64773.00
1022	2020-03-15	1	3	p06	450	709.92
1023	2020-03-15	1	2	p05	500	6550.98
1024	2020-03-15	5	2	p01	880	56133.00
1025	2020-04-01	8	3	p07	888	799.20
1026	2020-05-01	8	5	p03	808	47282.54

Originally from Database Principles, Programming, and Performance by Patrick O'Neil and Elizabeth O'Neil. Modified over and over by Alan G. Labouseur.

Scanning through 9 billion people

Table Scan of **unordered** data

check row 1	Peart
check row 2	Schock
check row 3	Crump
.	
.	
.	
check row 8,999,999,998	White
check row 8,999,999,999	Purdie
check row 9,000,000,000	Bruford

Sometimes we will find the selected person early in the table.
Sometimes we will find the selected person late in the table.

Q: What's the average — or expected — case for *n* rows?

Scanning through 9 billion people

Table Scan (aka Linear Search or Sequential Search)

check row 1	Peart
check row 2	Schock
check row 3	Crump
.	
.	
.	
check row 8,999,999,998	White
check row 8,999,999,999	Purdie
check row 9,000,000,000	Bruford

Sometimes we will find the selected person early in the table.
Sometimes we will find the selected person late in the table.

Q: What's the average — or expected — case for *n* rows?
A: The expected case is ½ *n*, which requires examining 4.5B rows in this example.

Scanning through 9 billion people

Table Scan

check row 1	`Peart`
check row 2	`Schock`
check row 3	`Crump`
.	
.	
.	
check row 8,999,999,998	`White`
check row 8,999,999,999	`Purdie`
check row 9,000,000,000	`Bruford`

That's what we call O(n) in computer science.

Pronounced "Big Oh of n", it means that the time or effort required to complete the task scales in a linear fashion with the number of items being worked on, n. (We ignore constant factors, like ½.)

Sometimes we will find the selected person early in the table.
Sometimes we will find the selected person late in the table.

Q: What's the average — or expected — case for n rows?
A: The expected case is ½ n, which requires examining 4.5B rows in this example.

Scanning through 9 billion people

Table Scan

check row 1	Peart
check row 2	Schock
check row 3	Crump
.	
.	
.	
check row 8,999,999,998	White
check row 8,999,999,999	Purdie
check row 9,000,000,000	Bruford

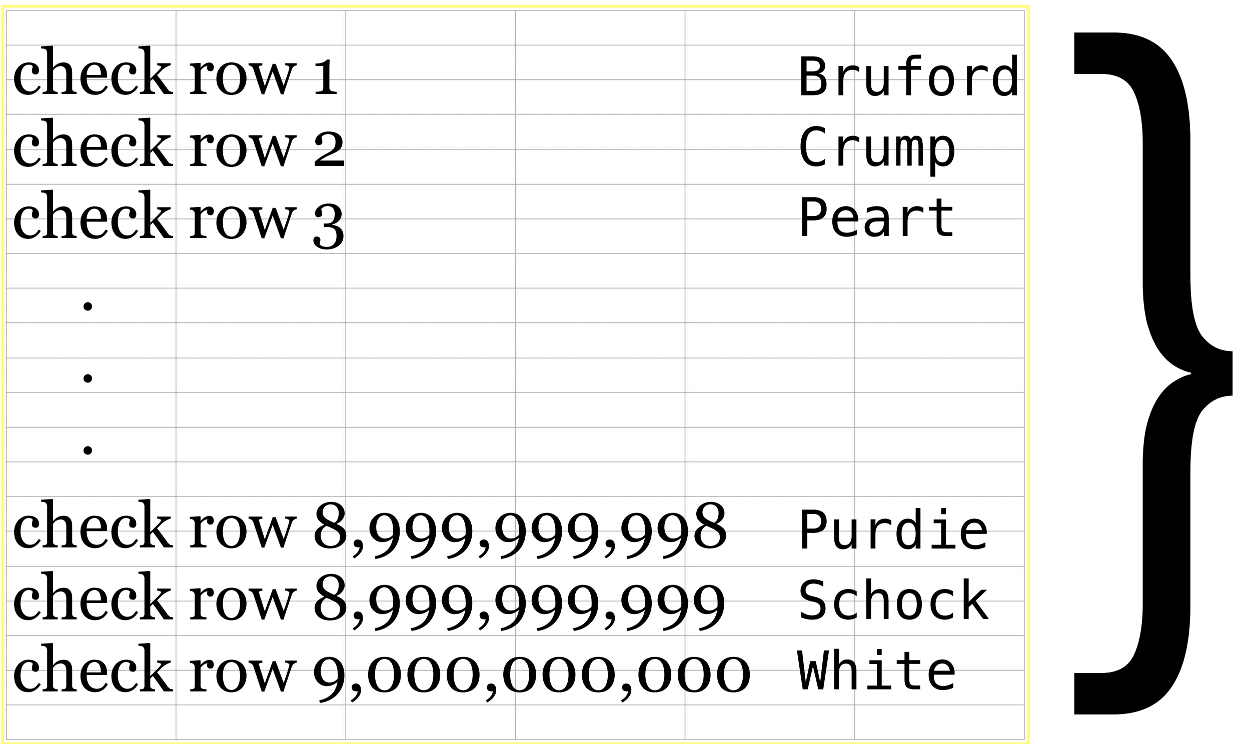
There must be a better way!

Sometimes we will find the selected person early in the table.
Sometimes we will find the selected person late in the table.

Q: What's the average — or expected — case for n rows?
A: The expected case is ½ n, which requires examining 4.5B rows in this example.

Searching 9 billion people

What if we could **search** through **sorted** data?



How would you do it?
What's your strategy?

Want to play a number guessing game?

Searching 9 billion people

What if we could **search** through **sorted** data?

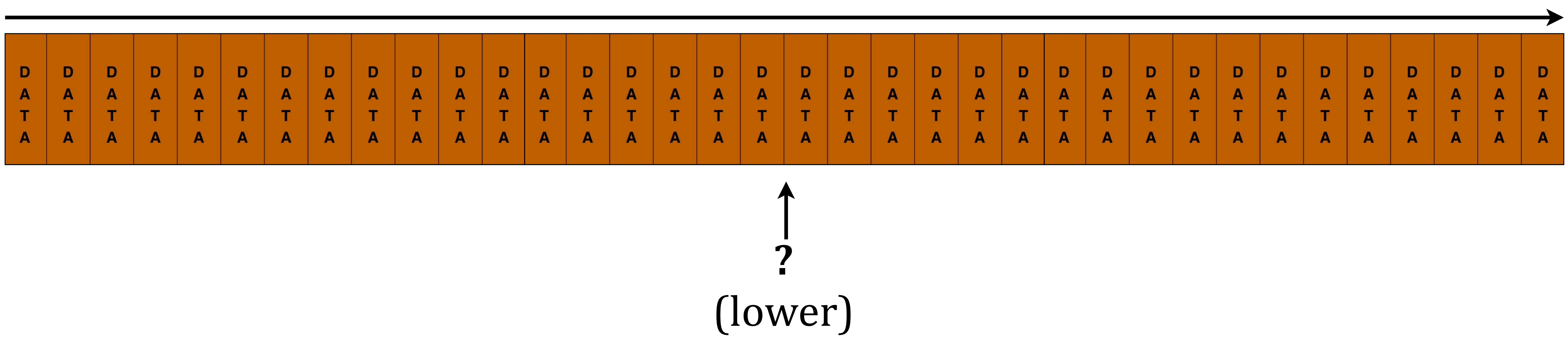
check row 1	Bruford
check row 2	Crump
check row 3	Peart
.	
.	
.	
check row 8,999,999,998	Purdie
check row 8,999,999,999	Schock
check row 9,000,000,000	White

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for *n* rows?

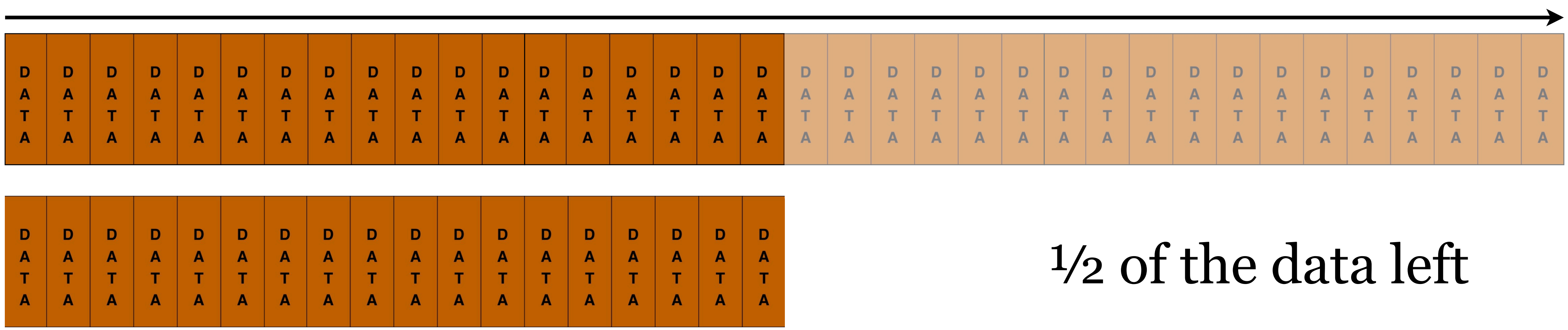
Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Searching 9 billion people

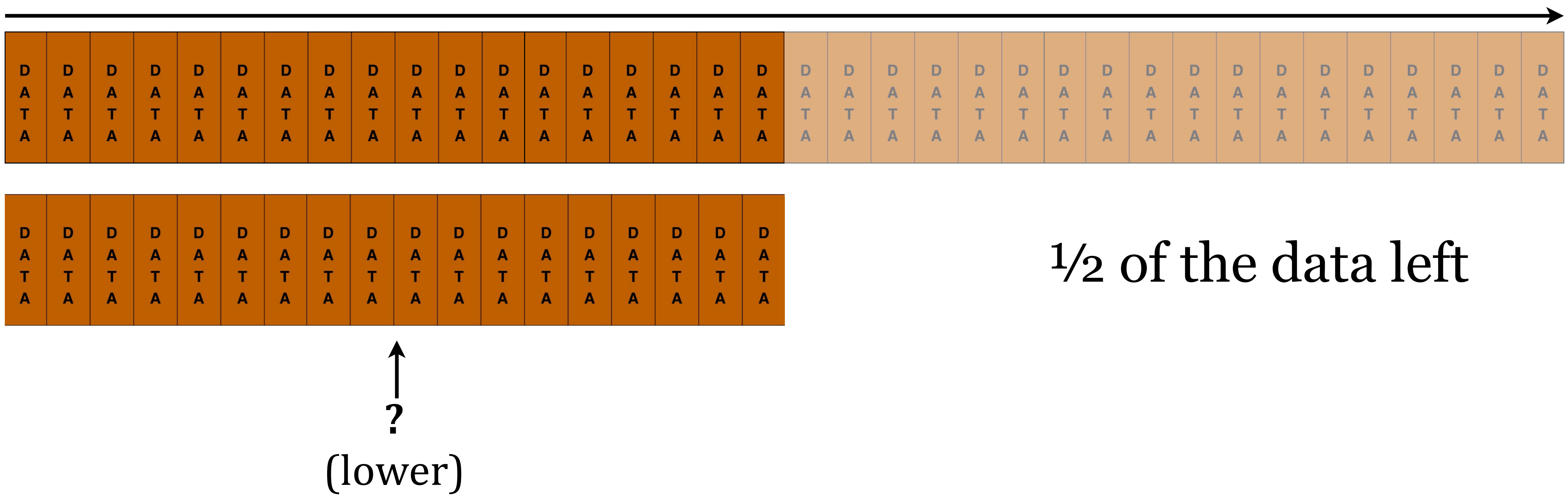
We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



½ of the data left

Searching 9 billion people

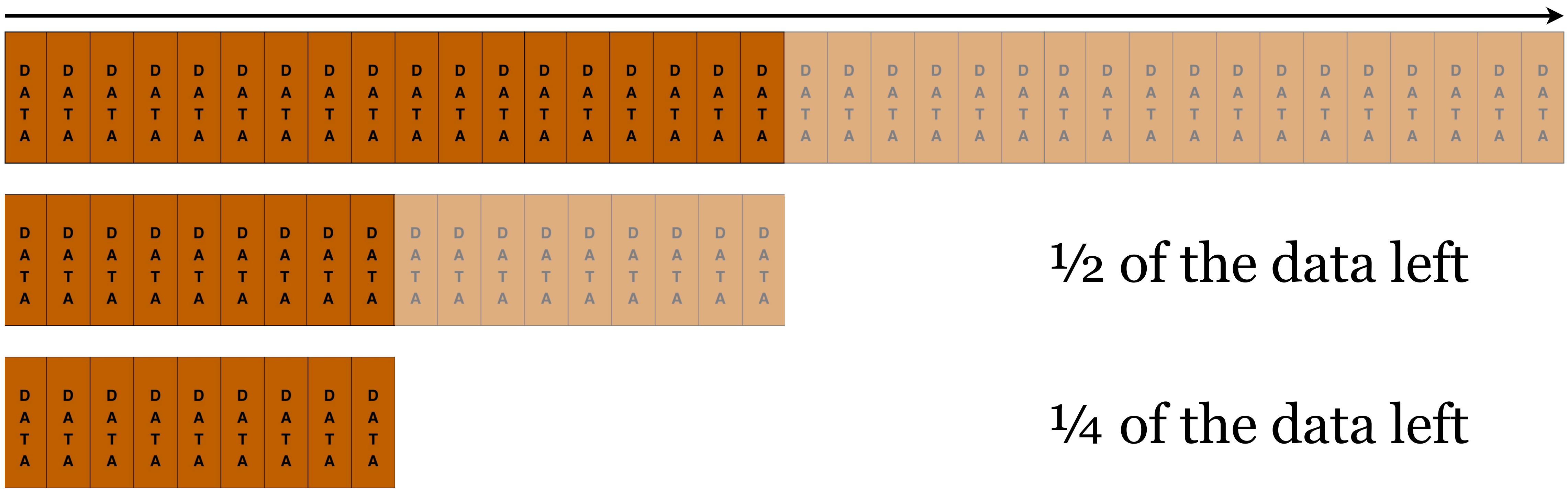
We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



½ of the data left

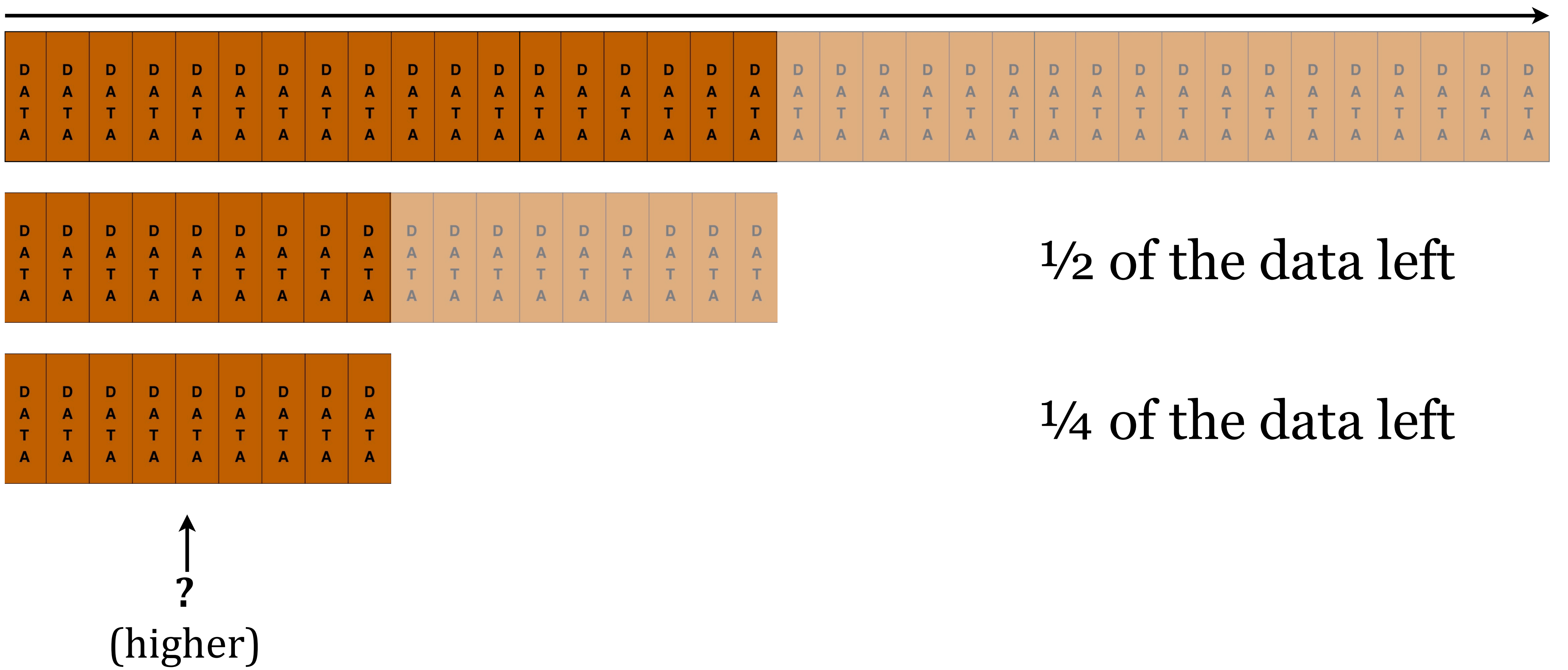
Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



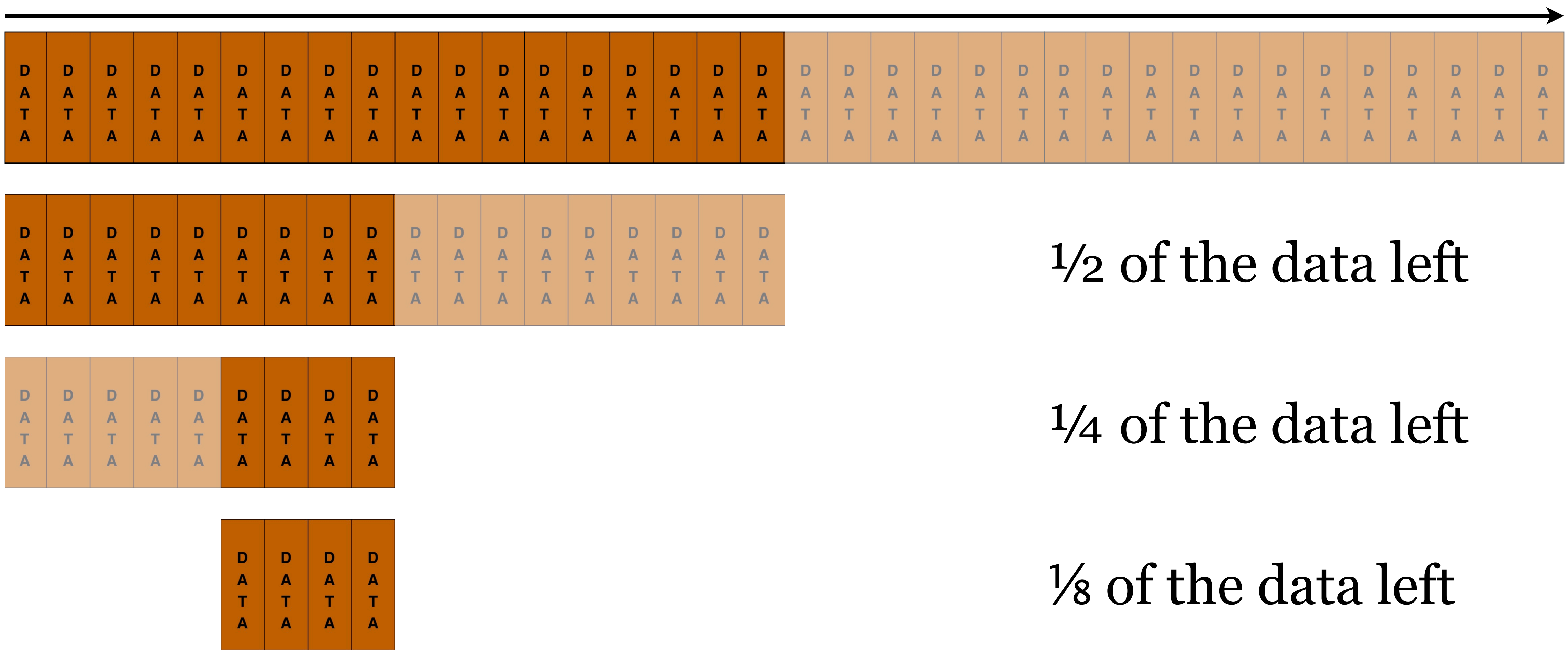
Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



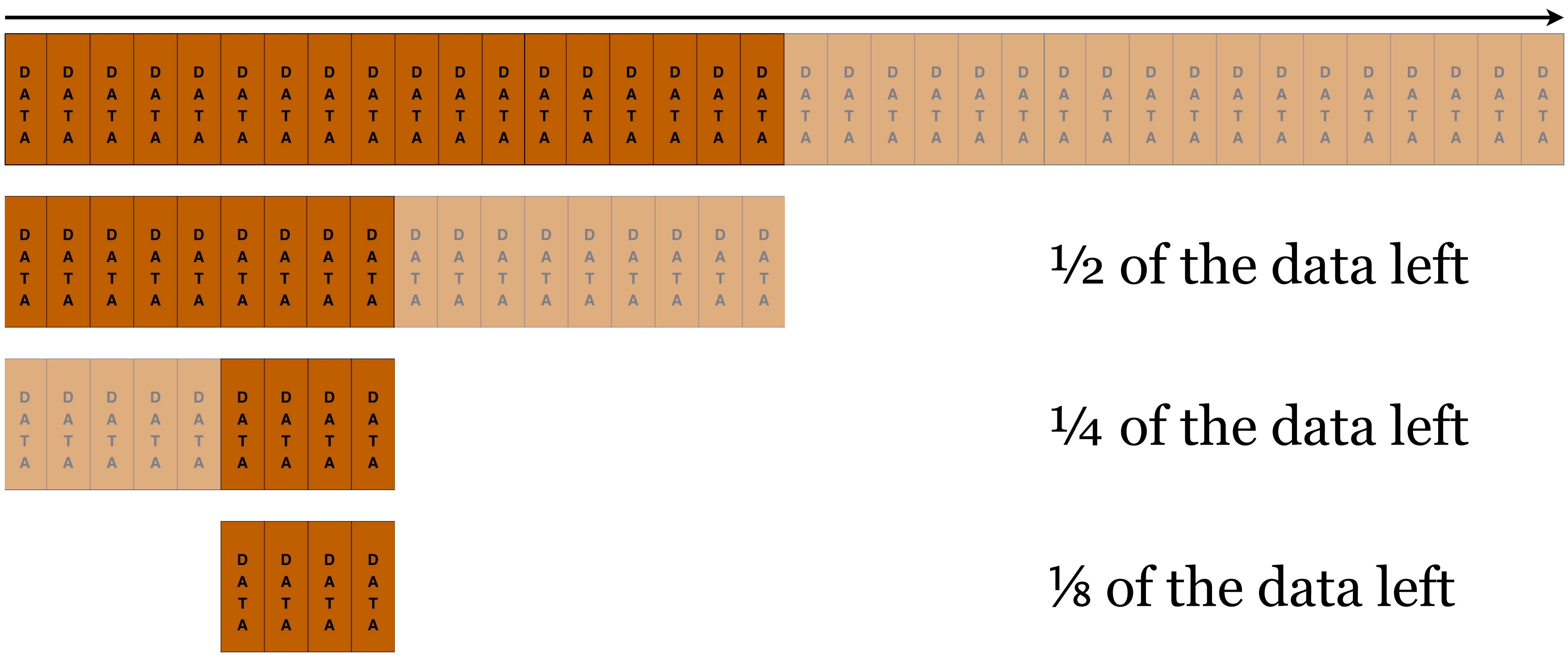
½ of the data left

¼ of the data left

⅛ of the data left

Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



½ of the data left

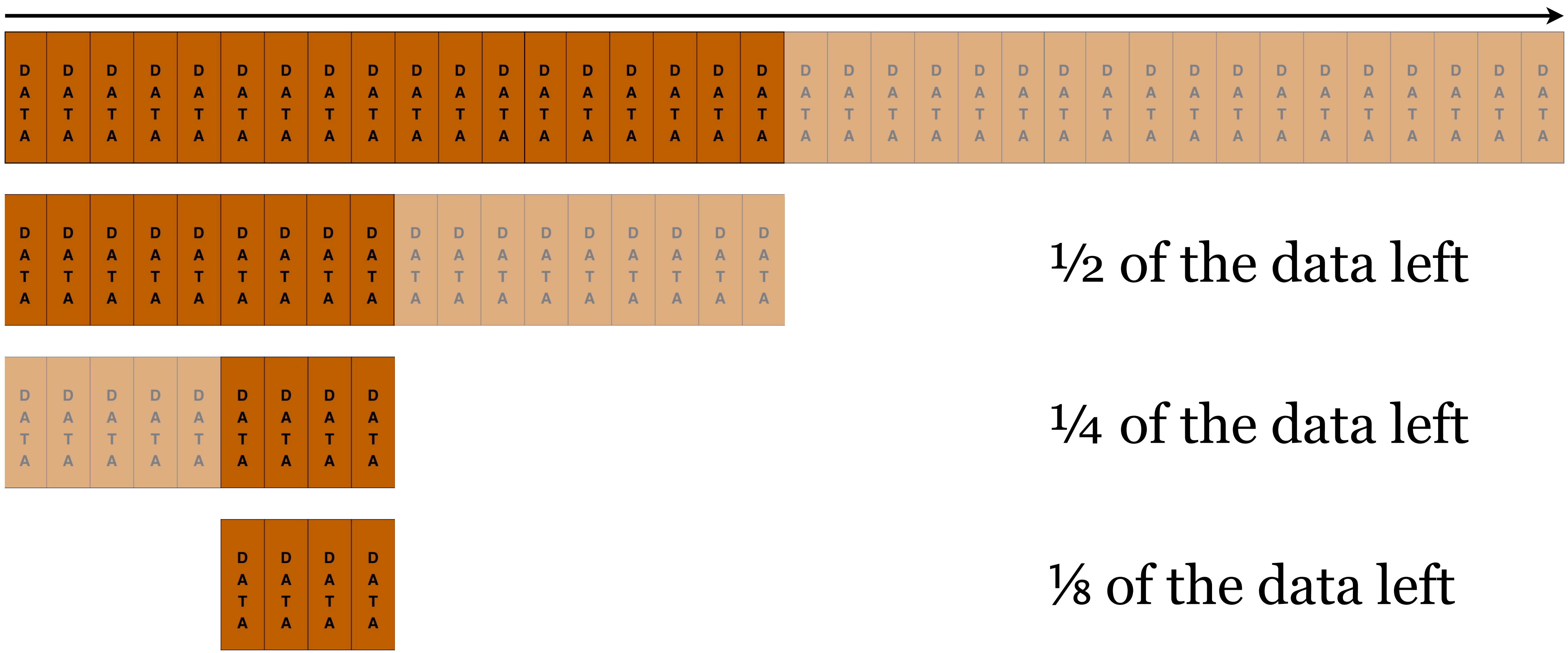
¼ of the data left

⅛ of the data left

Q: What's the average or — expected — case for *n* rows?

Searching 9 billion people

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.



½ of the data left

¼ of the data left

⅛ of the data left

Q: What's the average or — expected — case for n rows?
A: The expected case is **$\log_2 n$**, because we cut it in half each time.

Searching 9 billion people

What if we could search through **sorted** data?

check row 1	Bruford
check row 2	Crump
check row 3	Peart
.	
.	
.	
check row 8,999,999,998	Purdie
check row 8,999,999,999	Schock
check row 9,000,000,000	White

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . ?

Searching 9 billion people

What if we could search through **sorted** data?

check row 1	Bruford
check row 2	Crump
check row 3	Peart
.	
.	
.	
check row 8,999,999,998	Purdie
check row 8,999,999,999	Schock
check row 9,000,000,000	White

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

Q: What's the average or — expected — case for n rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . 33

Searching 9 billion people

What if we could search through **sorted** data?

check row 1	`Bruford`
check row 2	`Crump`
check row 3	`Peart`
.	
.	
.	
check row 8,999,999,998	`Purdie`
check row 8,999,999,999	`Schock`
check row 9,000,000,000	`White`

Now **that** is a better way!
33 < 4.5B

We could pick from the middle. If that's not our target, then we exclude the *lower* or *upper* half of the data, depending on whether our target is greater or lesser than the value we picked. Then we pick the middle of the remaining half. Repeat.

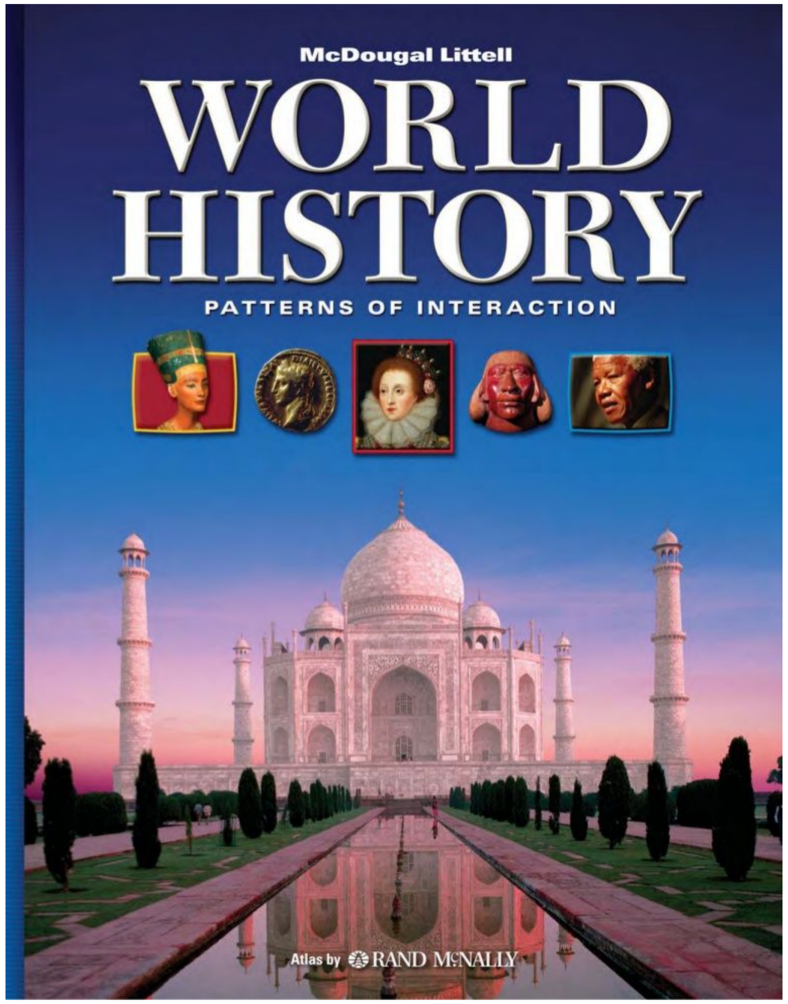
Q: What's the average or — expected — case for *n* rows?
A: The expected case is $\log_2 n$. By the way, $\log_2$ 9B is . . . 33

So . . .

How do we take advantage of sorted data when tables are sets of rows and therefore have no intrinsic order?

Text Books

Consider a text book . . .



… **physically arranged** chronologically from page 1 to *n*.

… with an **index** in the back arranged by topic, with page number references.

… and another **index** arranged by geography with page number references.

Indexes

An **index** is a database object that increases search and lookup speed by imposing order.

Indexes (or indicies) are created with the CREATE INDEX SQL command.

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON [ ONLY ] table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass [ ( opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ INCLUDE ( column_name [, ...] ) ]
    [ WITH ( storage_parameter [= value] [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

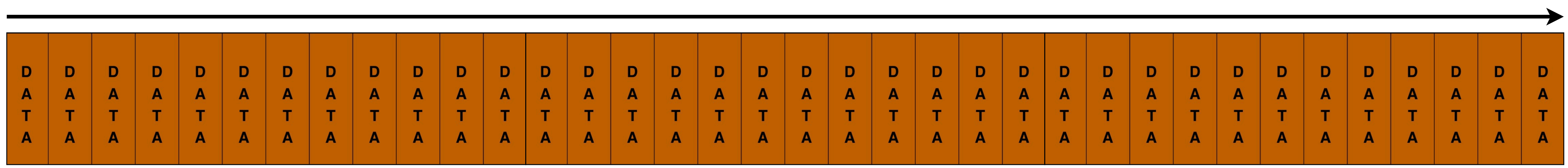
Indexes are created **on** one or more columns in a table. E.g.,

```
CREATE INDEX NameDex ON People (lastName, firstName);
```

There are two kinds of index:
(1) a **clustered** index
(2) a **logical** index

Clustered Index

A **clustered index** is the physical order of the rows of a base table in storage.



Each table can have only one clustered index because it can be stored only in one physical order.

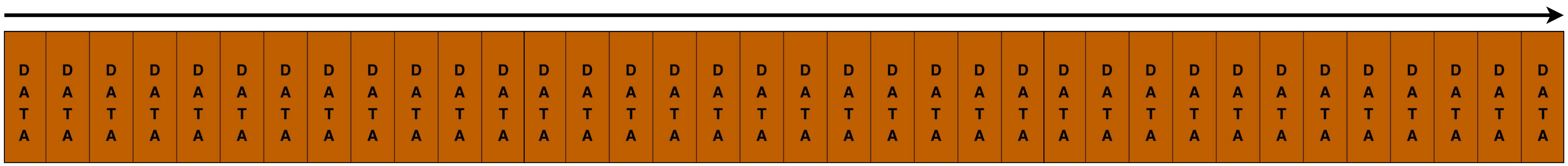
Q: Primary Key values make for nice clustered indexes. Why?

Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

CREATE CLUSTERED INDEX PEOPLE_PKEY ON PEOPLE(*PID*); *// this is T-SQL syntax, not PostgreSQL.*

Clustered Index

A **clustered index** is the physical order of the rows of a base table in storage.



Each table can have only one clustered index because it can be stored only in one physical order.

Q: Primary Key values make for nice clustered indexes. Why?

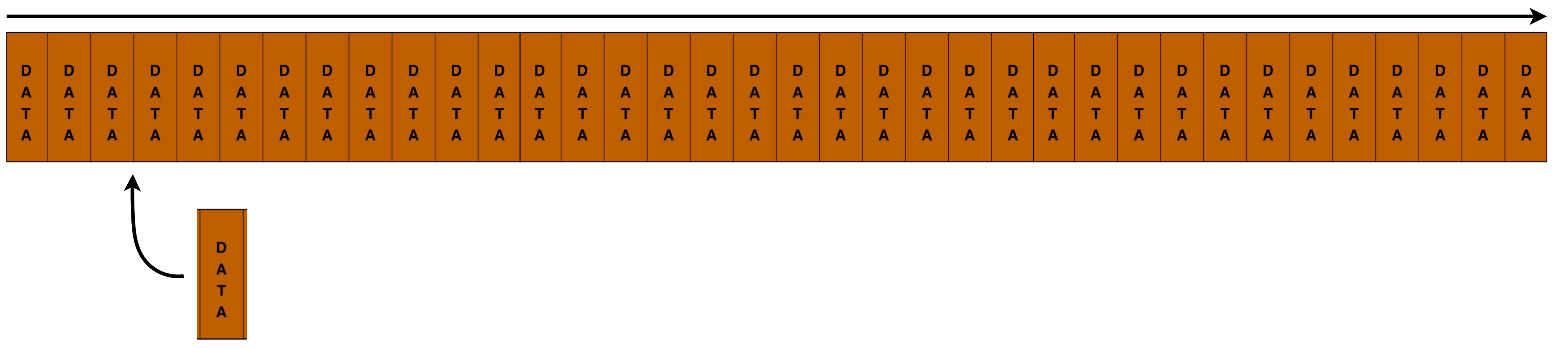
A: Most joins are PK-FK, so the query engine can cross-reference them in log-based lookup time, making joins perform fast.

Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

CREATE CLUSTERED INDEX PEOPLE_PKEY ON PEOPLE(*PID*); *// this is T-SQL syntax, not PostgreSQL.*

Clustered Index

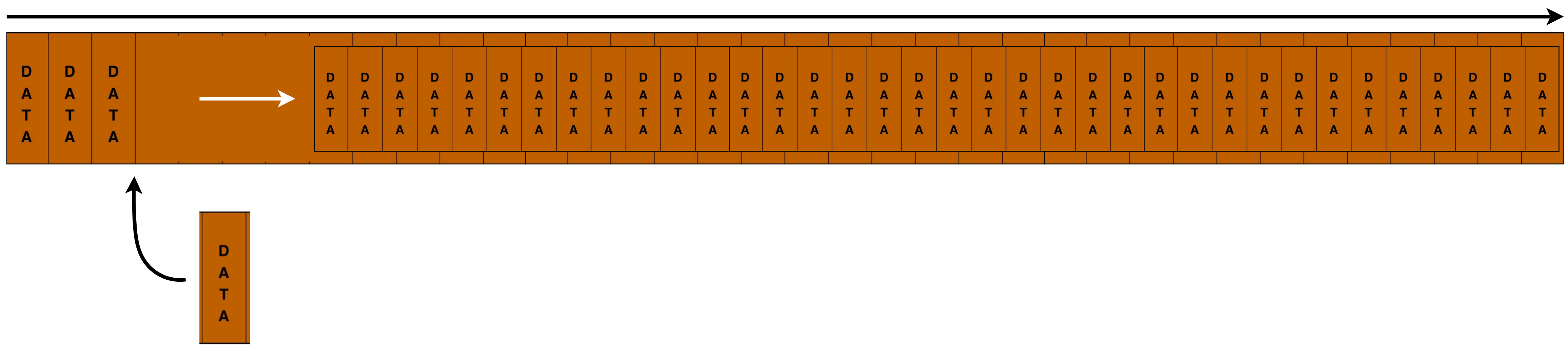
A **clustered index** is the physical order of the rows of a base table in storage.



Q: What happens if we need to add a new value anywhere other than the end of the clustered index?

Clustered Index

A **clustered index** is the physical order of the rows of a base table in storage.

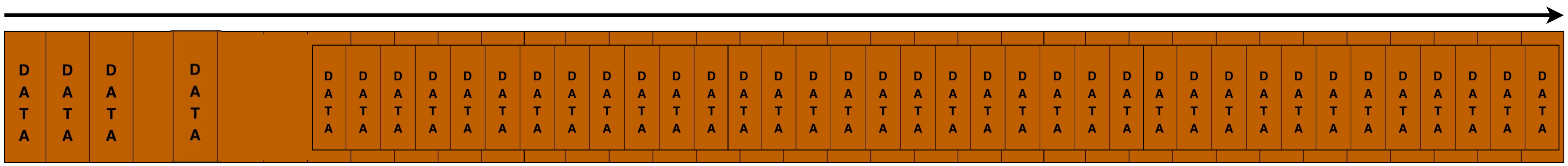


Q: What happens if we need to add a new value anywhere other than the end of the clustered index?

A: We need to re-organize ("smush") everything from that point on to make room in the table.

Clustered Index

A **clustered index** is the physical order of the rows of a base table in storage.



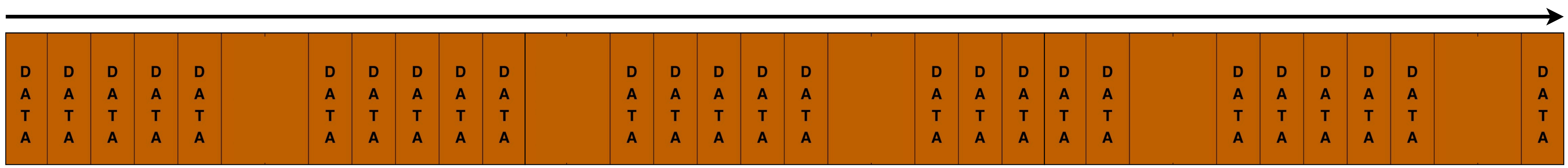
Q: What happens if we need to add a new value anywhere other than the end of the clustered index?

A: We need to re-organize ("smush") everything from that point on to make room in the table.
This can take considerable time; a "stop the world" event inside the database.

Let's not do that.

Clustered Index

A **clustered index** is the physical order of the rows of a base table in storage.



We can trade space for time by setting aside some empty space in the table so that it's not *fully packed*. In this manner there is space available for future inserts and updates.

This is called "fill factor". Fully packed means a fill factor of 100%. Leaving 10% empty space means a 90% fill factor.

`fillfactor` (`integer`)

The fillfactor for a table is a percentage between 10 and 100. 100 (complete packing) is the default. When a smaller fillfactor is specified, `INSERT` operations pack table pages only to the indicated percentage; the remaining space on each page is reserved for updating rows on that page. This gives `UPDATE` a chance to place the updated copy of a row on the same page as the original, which is more efficient than placing it on a different page. For a table whose entries are never updated, complete packing is the best choice, but in heavily updated tables smaller fillfactors are appropriate. This parameter cannot be set for TOAST tables.

Logical Index

A **logical index** is **a tree of pointers** to the physical rows of a base table in storage.

Each table can have many logical indices because they are stored separately.

Consider an index on last name in People:

```
CREATE INDEX NameDex ON People (lastName);
```

Since the clustered index is on *pid* (meaning the rows are stored in *pid* order) we need a different structure to access the People table in a different order, like by *last name* for example. We'll use a tree of pointers for that.

Logical Index

CREATE INDEX NameDex ON People (lastName);

We'll make a tree of pointers based on the *lastName* column of the People table.

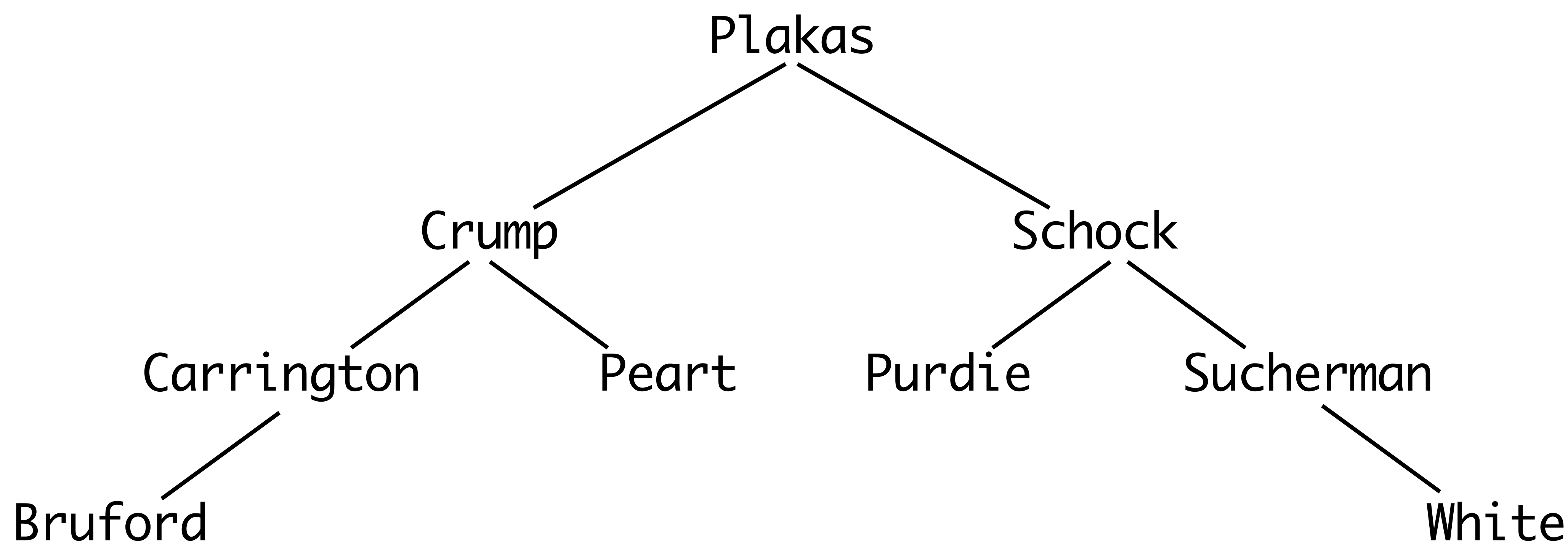
We'll call it a **b**-tree.

Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Clustered index on *pid*.

Logical Index

CREATE INDEX NameDex ON People (lastName);



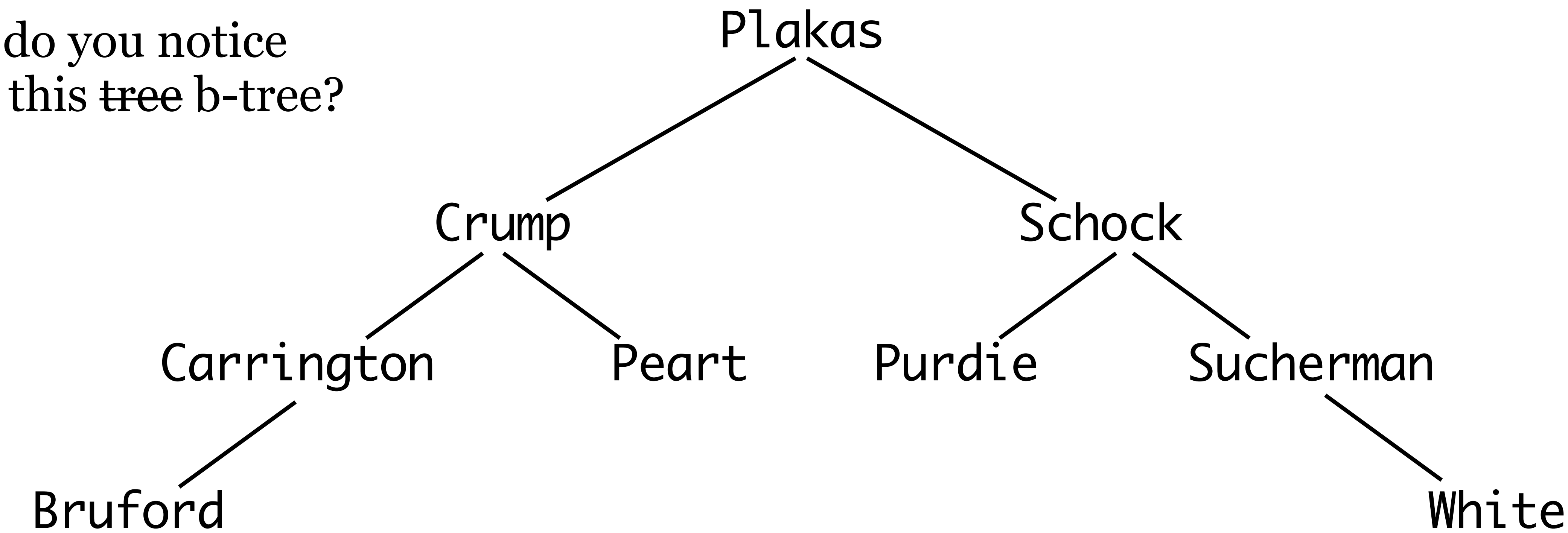
Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Clustered index on *pid*.

Logical Index

CREATE INDEX NameDex ON People (lastName);

What do you notice about this ~~tree~~ b-tree?



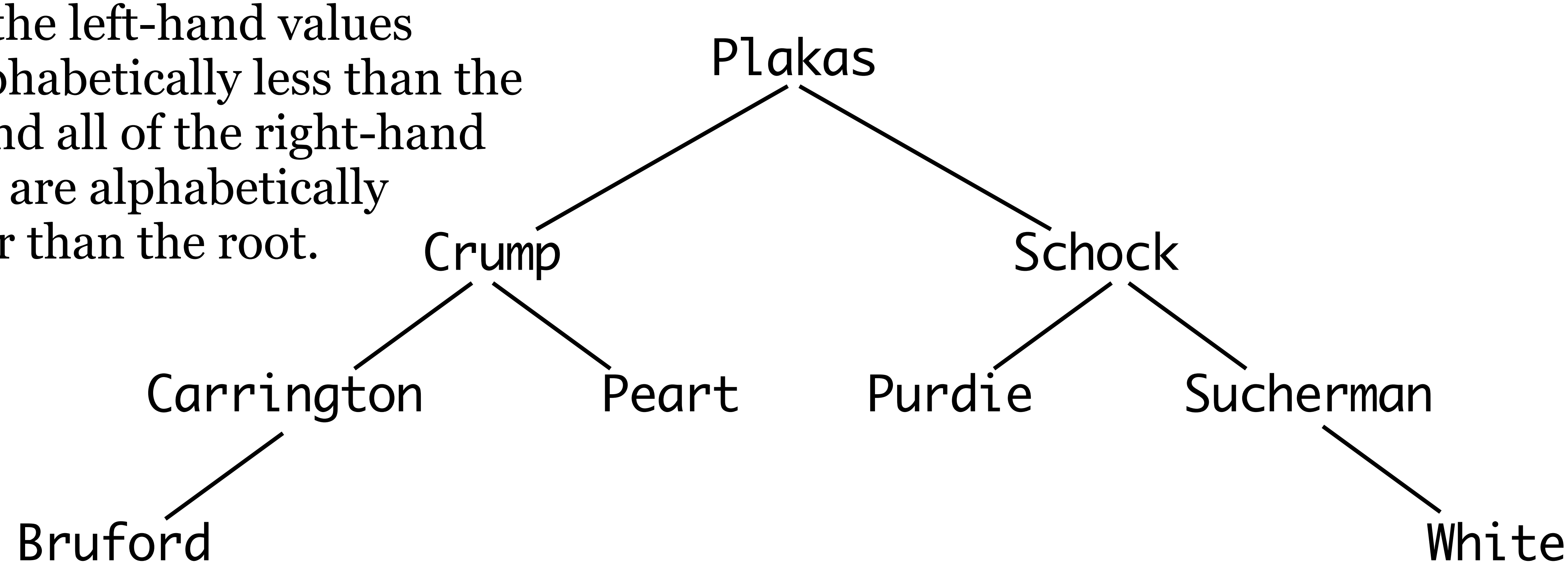
Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Clustered index on *pid*.

Logical Index

CREATE INDEX NameDex ON People (lastName);

All of the left-hand values are alphabetically less than the root and all of the right-hand values are alphabetically greater than the root.

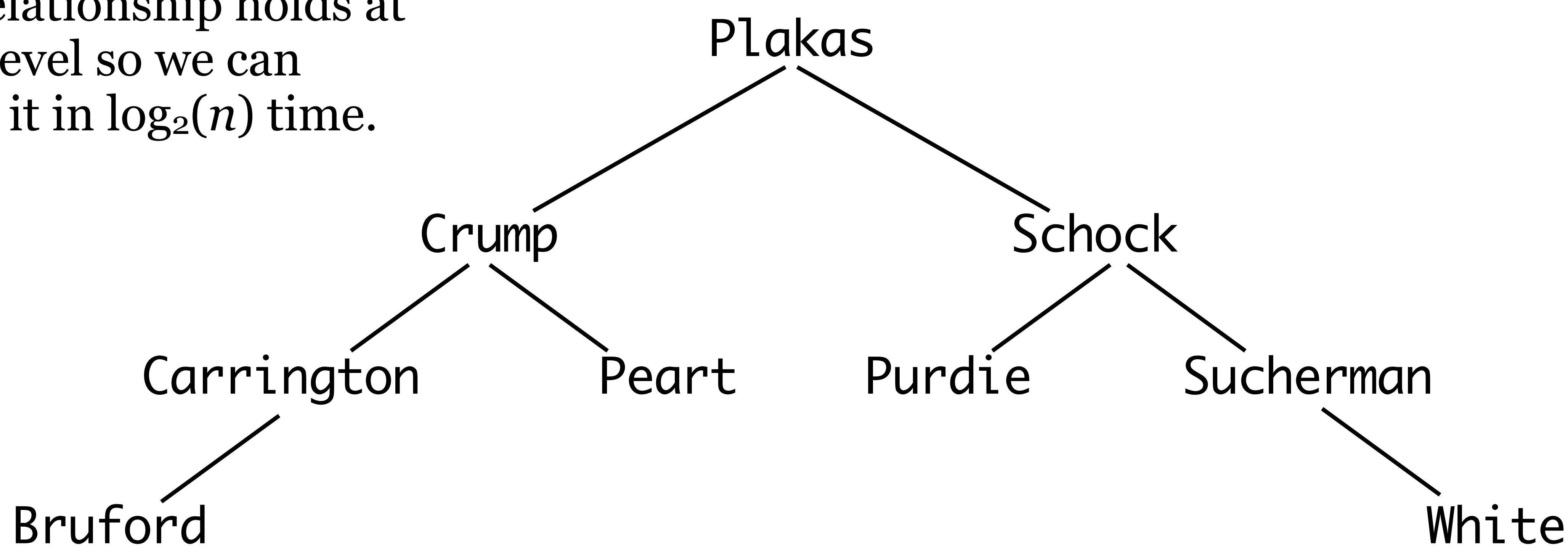


Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Logical Index

CREATE INDEX NameDex ON People (lastName);

This relationship holds at every level so we can search it in $\log_2(n)$ time.

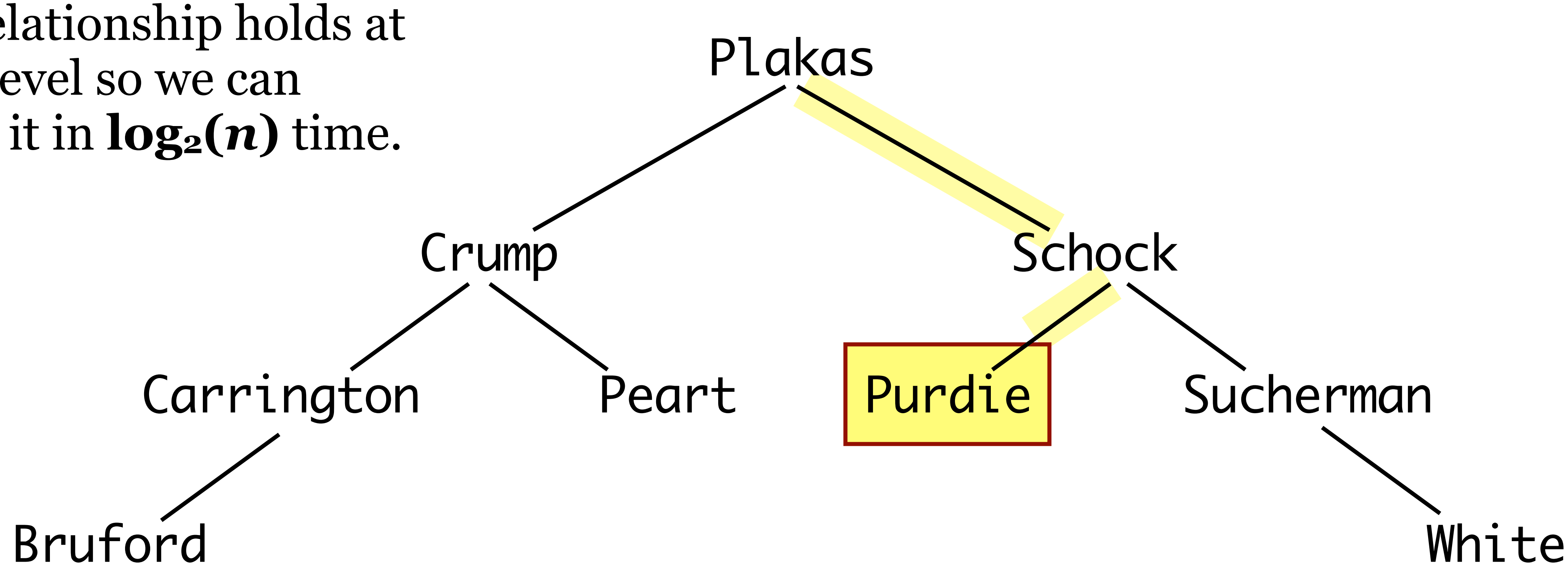


Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Logical Index

CREATE INDEX NameDex ON People (lastName);

This relationship holds at every level so we can search it in **$\log_2(n)$** time.

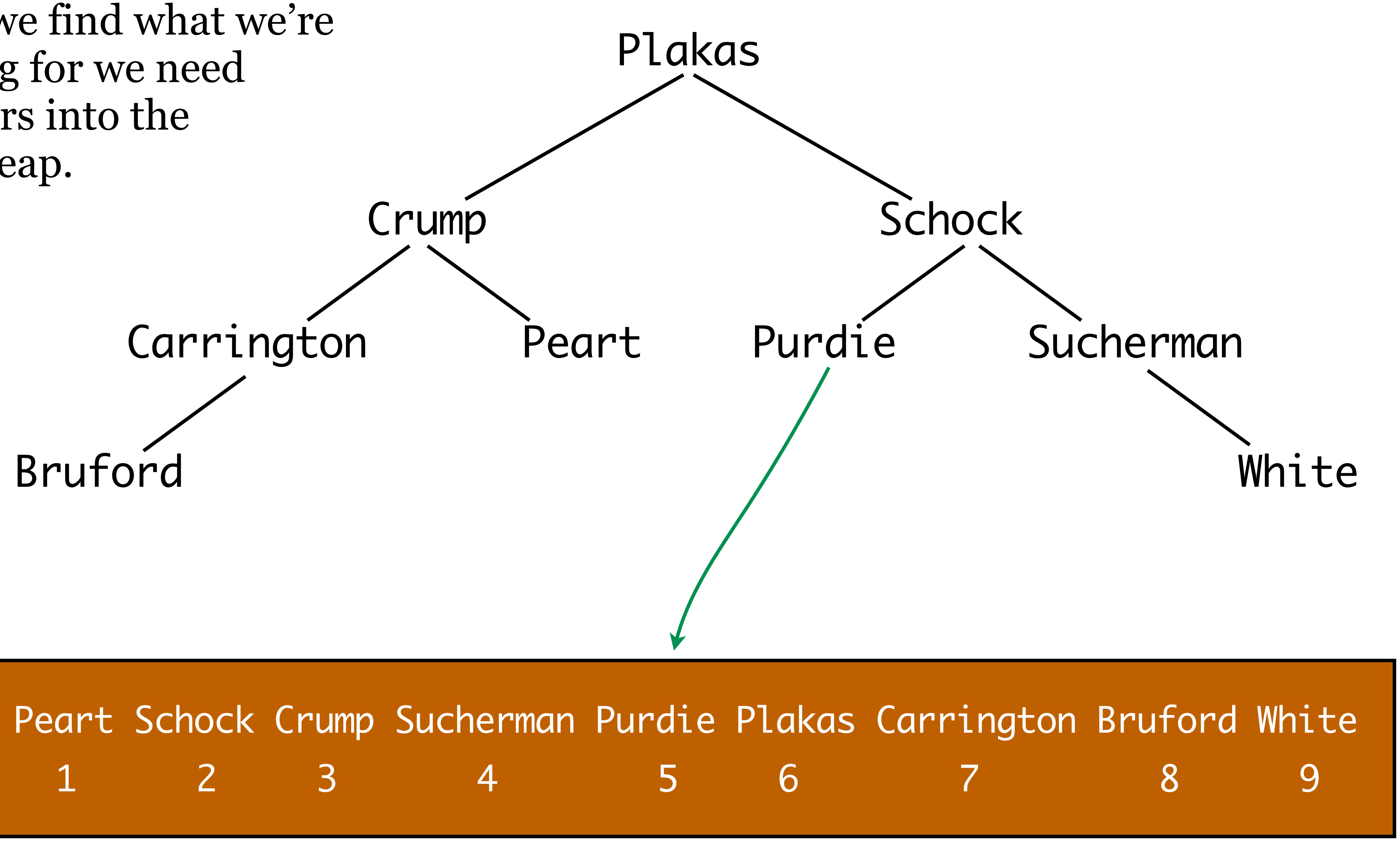


Peart	Schock	Crump	Sucherman	Purdie	Plakas	Carrington	Bruford	White
1	2	3	4	5	6	7	8	9

Logical Index

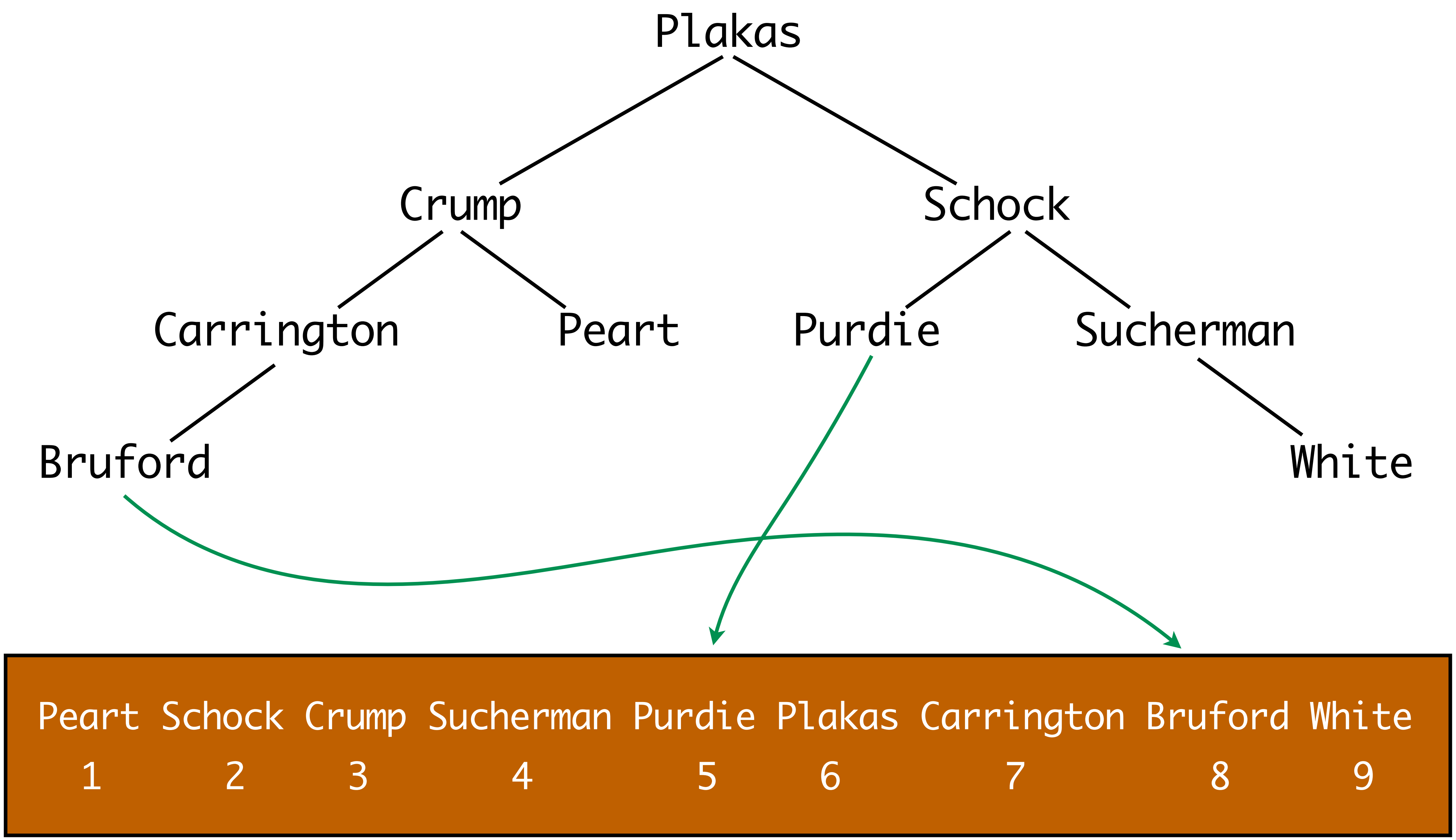
CREATE INDEX NameDex ON People (lastName);

Once we find what we're looking for we need pointers into the data heap.



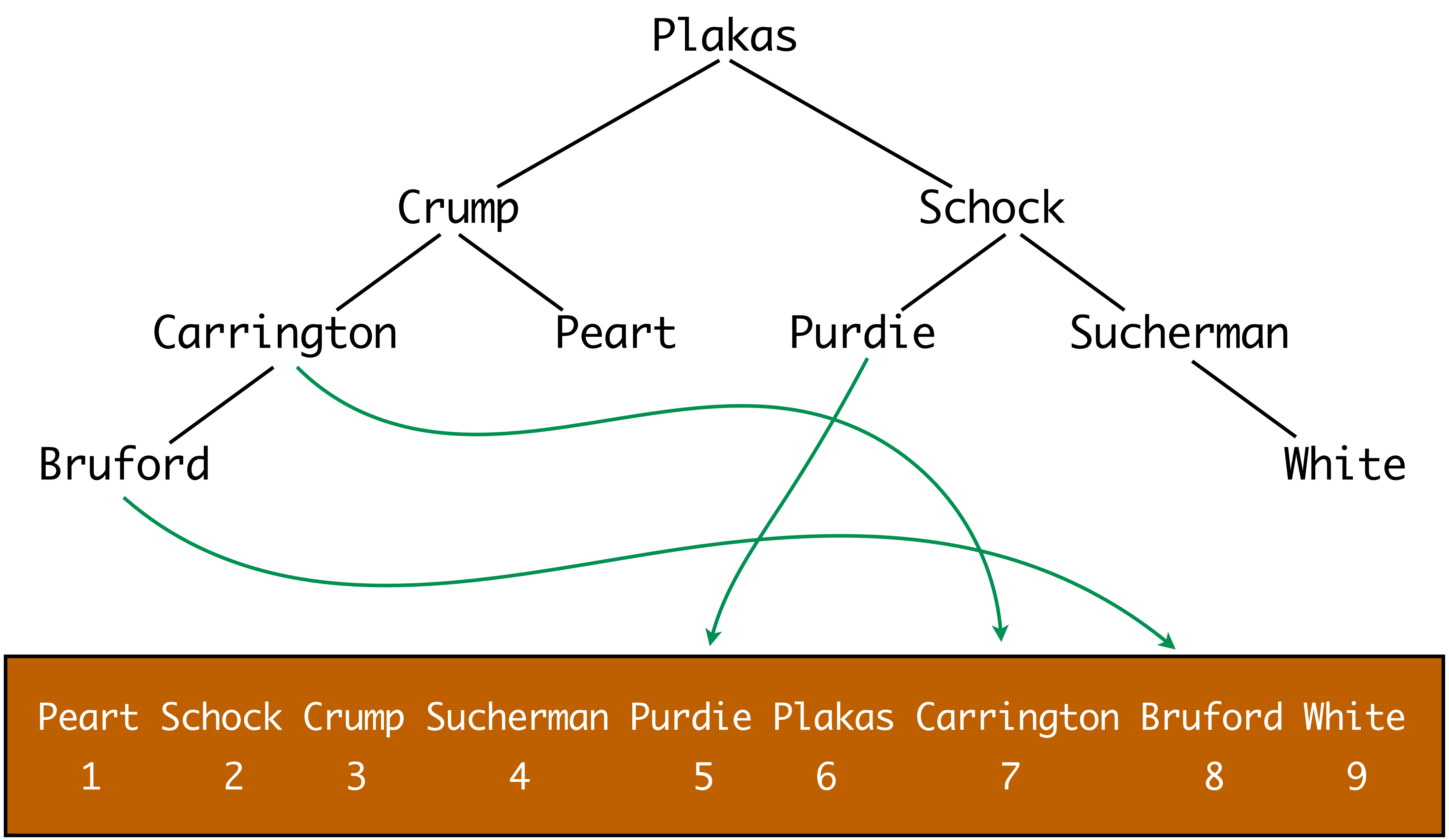
Logical Index

CREATE INDEX NameDex ON People (lastName);



Logical Index

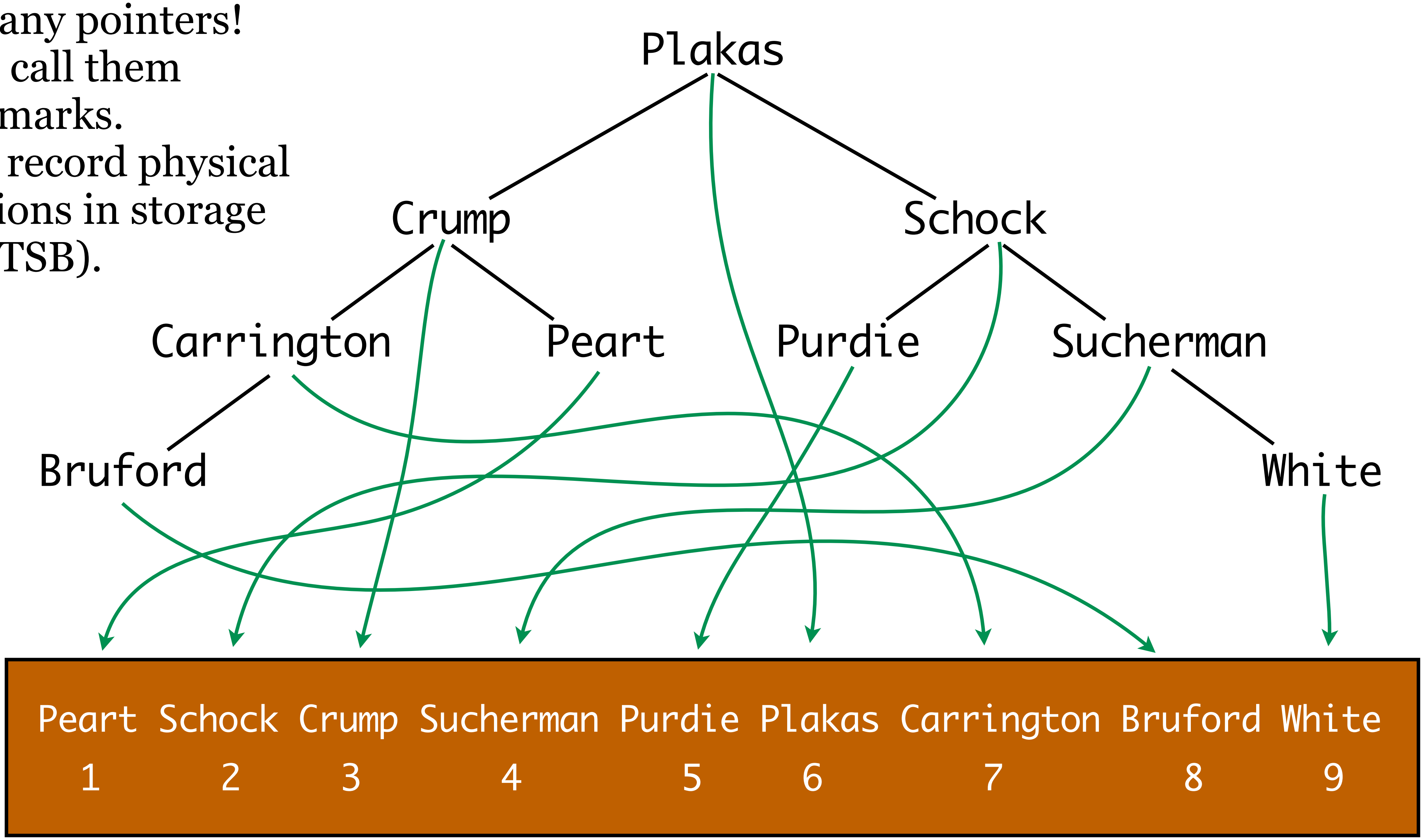
CREATE INDEX NameDex ON People (lastName);



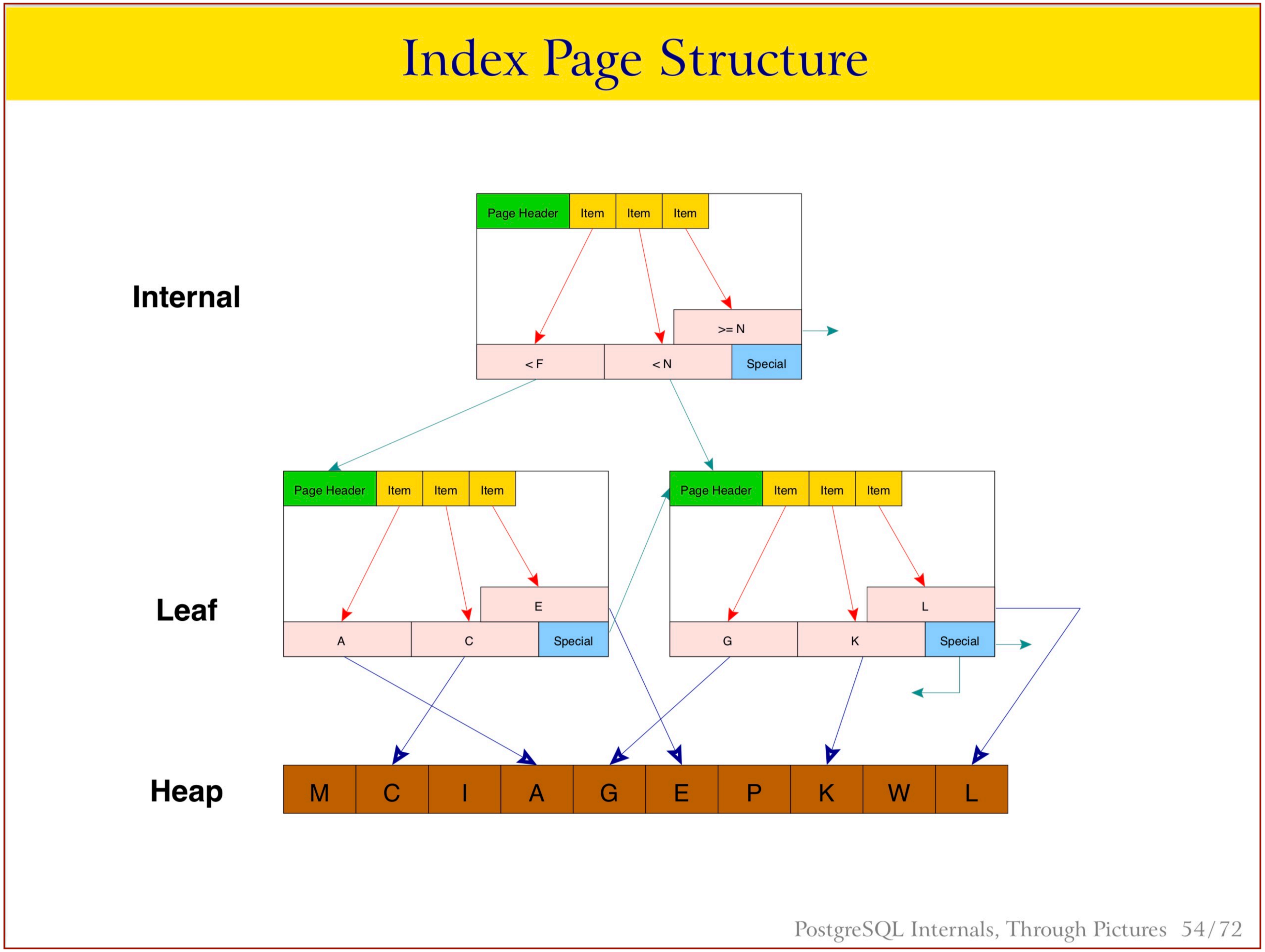
Logical Index

CREATE INDEX NameDex ON People (lastName);

So many pointers!
We'll call them bookmarks.
They record physical locations in storage (like TSB).

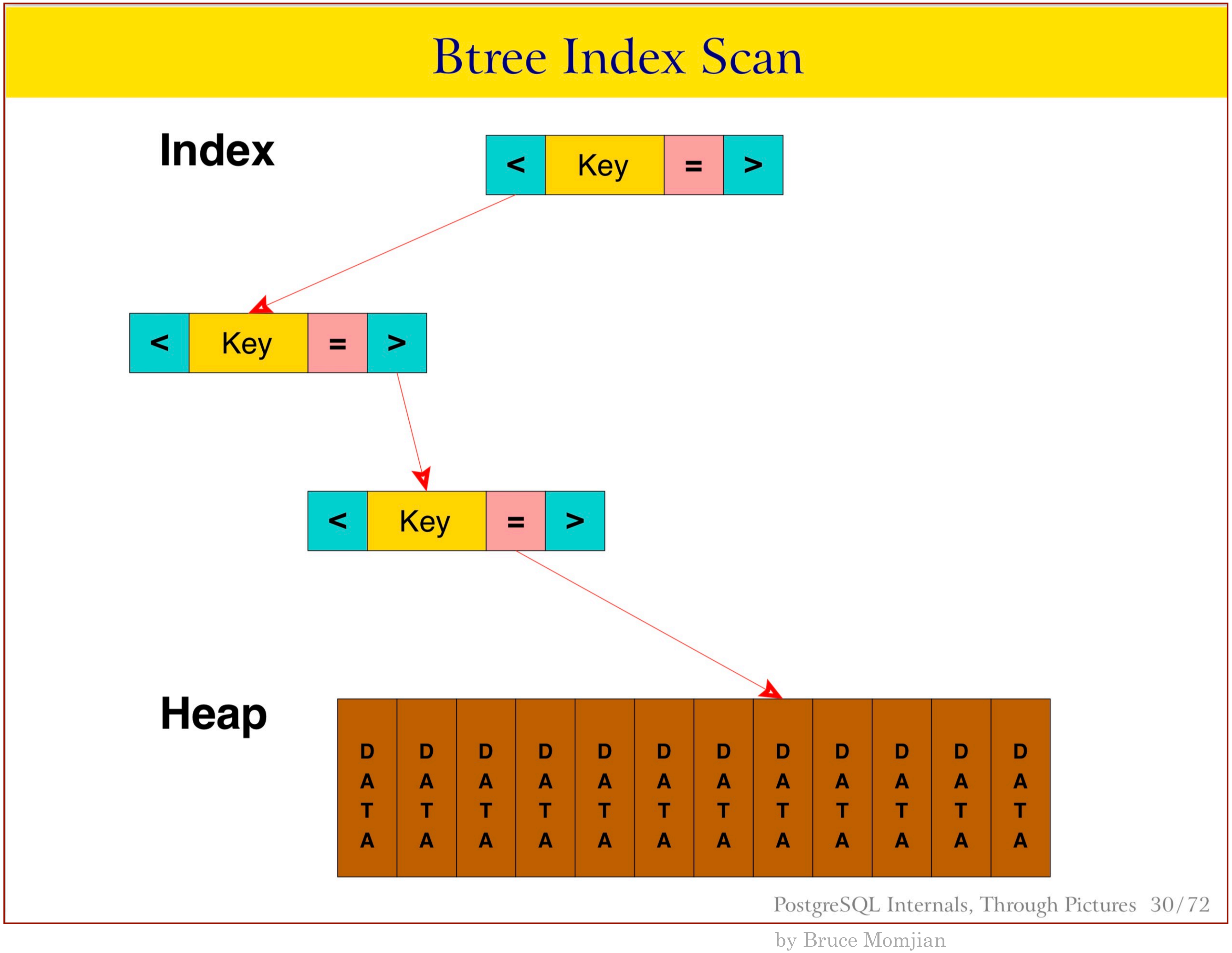


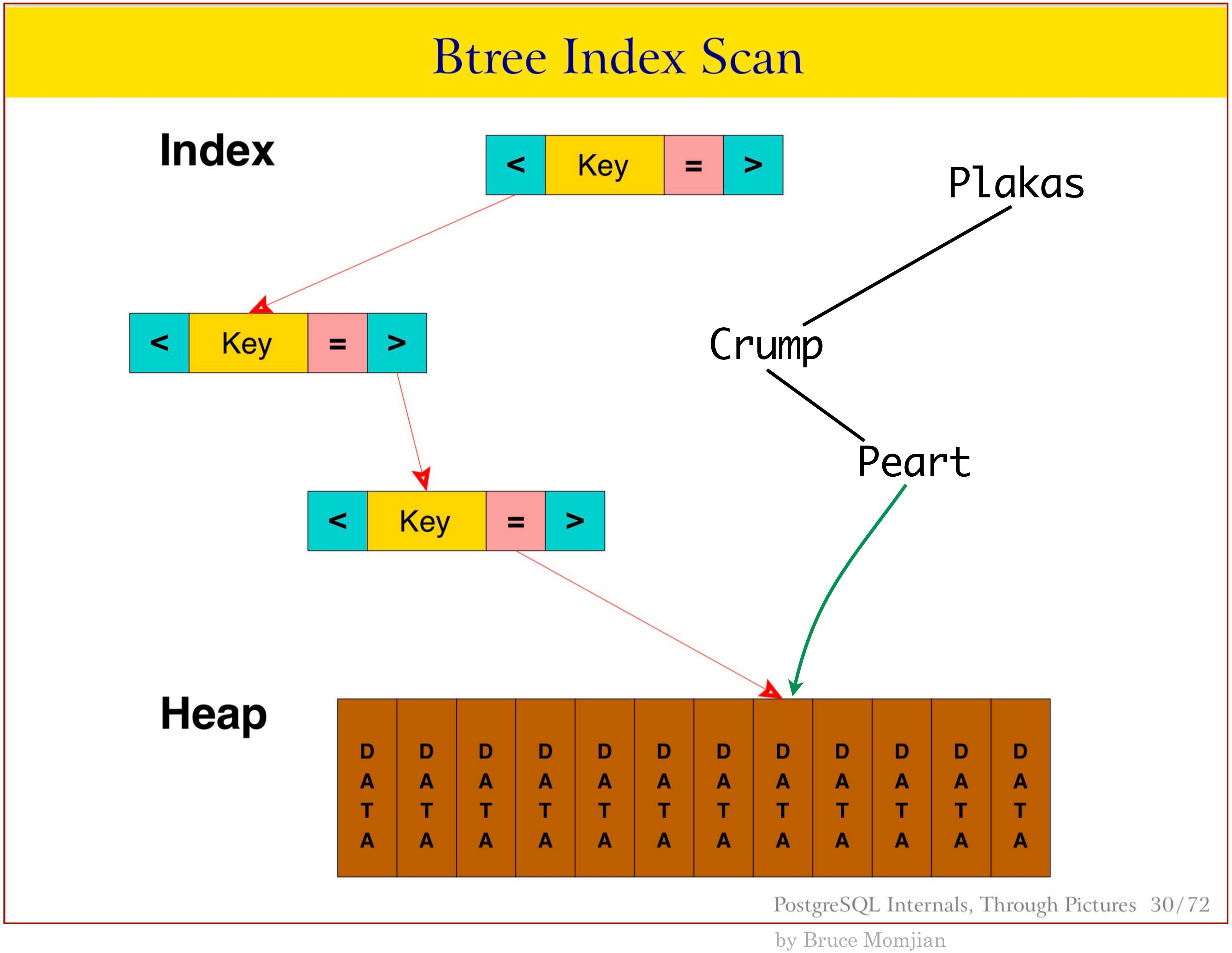
PostgreSQL Internal Index Structures



by Bruce Momjian

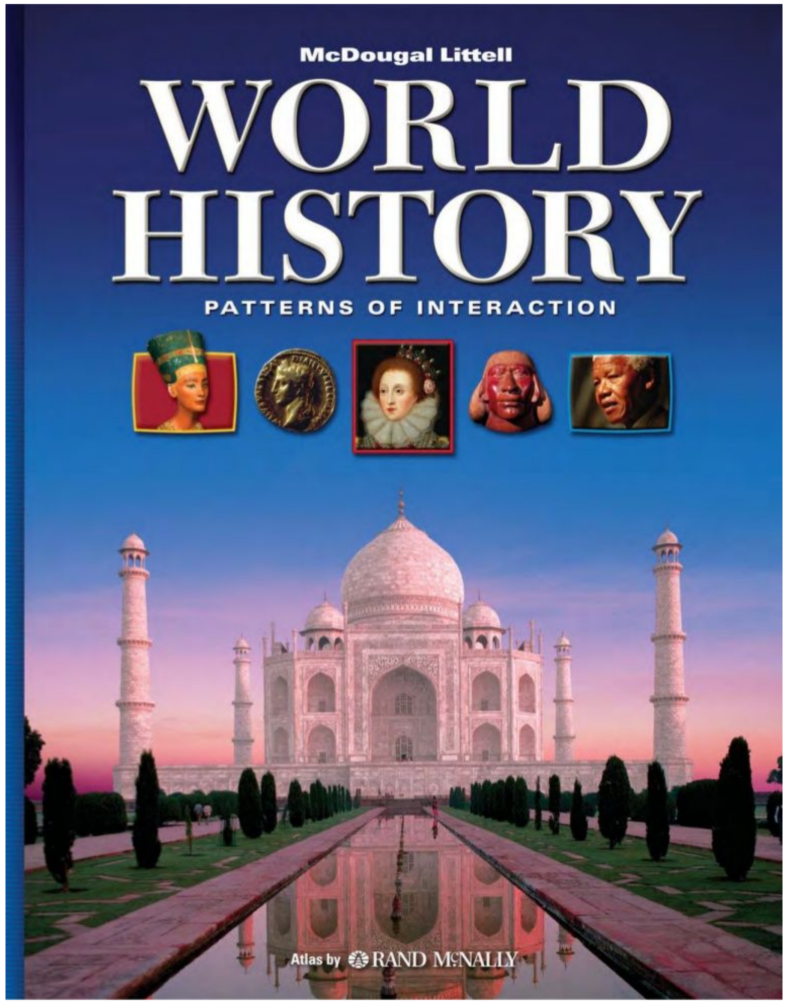
PostgreSQL Internal Index Structures




Btree Index Scan
Index
<
Key
=
>
Plakas
Crump
Peart
Heap
DATA
PostgreSQL Internals, Through Pictures 30/72
by Bruce Momjian

Text Books and Database Systems

Consider a text book . . .



Clustered Index

… physically arranged chronologically from page 1 to n.

Logical Index

… with an index in the back arranged by topic, with page number references.

… and another index arranged by geography with page number references.

Pointers

CAP Default Indexes

```sql
SELECT *
FROM pg_catalog.pg_indexes
WHERE schemaname = 'public'
```

Data Output Explain Messages Notifications

	schemaname name	tablename name	indexname name	tablespace name	indexdef text
1	public	people	people_pkey	[null]	CREATE UNIQUE INDEX people_pkey ON public.people USING btree (pid)
2	public	customers	customers_pkey	[null]	CREATE UNIQUE INDEX customers_pkey ON public.customers USING btree (pid)
3	public	agents	agents_pkey	[null]	CREATE UNIQUE INDEX agents_pkey ON public.agents USING btree (pid)
4	public	products	products_pkey	[null]	CREATE UNIQUE INDEX products_pkey ON public.products USING btree (prodid)
5	public	orders	orders_pkey	[null]	CREATE UNIQUE INDEX orders_pkey ON public.orders USING btree (ordernum)

PostgreSQL created these automatically.
Interestingly, though these are PK indexes, they are logical (because they are *btree* indexes) and not clustered.

(That's why the syntax for creating a clustered index a few slides ago is from SQL Server's T-SQL language.)

Make Your Own Indexes?

When should you create your own indexes?

Do make indexes for frequently accessed columns that have many or mostly unique values (high **selectivity**).

Do **not** make indexes for columns that have few unique values (low selectivity).

Do not make indexes for columns that are frequently updated because indexes need to be updated too. That's a trade-off of using them: slower insert and update but faster retrieval.

What's the other trade-off?