Tutorial 4 — Storage, B+ Trees

Richard Wong
rk2wong@edu.uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

February 9, 2018

Exercise 4-1

Does the data in a heap file necessarily lie in a contiguous region in secondary memory (disk)?

Exercise 4-1 Solution

No, recall that a file is a logical abstraction. The OS filesystem manages the blocks that make up our files. These blocks can exist apart from one another in physical memory.

Exercise 4-2

Why would we use a B+ tree file instead of a sorted file to hold our database records?

Exercise 4-2 Solution

Suppose our database contains more data than fits in main memory.

How would you search the sorted file? The binary search algorithm requires us to know the midpoint of our sorted collection.

The B+ tree allows us to perform O(log n) lookups/insertions/deletions without a O(n) memory footprint.

The B+ tree also lets us narrow down lookup results by more than a factor of 2 with each iteration of the lookup algorithm.

Exercise 4-3

When might we want to use a hash file instead of a B+ tree file to hold our database records?

Exercise 4-3 Solution

The hash file grants us constant-time lookup by key, but not much else.

If we expect a significant portion of our queries to be lookups by key equality, this could be good (but comparisons, ordering, etc. will be slooooooow...)

Exercise 4-4

When would we prefer MRU over LRU as a database cache replacement algorithm?

Exercise 4-4 Solution

This depends on memory access patterns.

MRU is good under the assumption that blocks we have used recently are less likely to be accessed soon. This is typically true with database systems.

This is the opposite assumption that LRU makes (temporal locality). We might want to use LRU in a more generalized setting (OS memory management, where the OS doesn't know memory access patterns of applications).

Exercise 4-5

Why shouldn’t we make lots of indices over our database tables?

Exercise 4-5 Solution

Each index on our table introduces a time overhead with each record update, insertion, and deletion, if we want to keep our indices up to date (which is likely, in our single-machine environment).

Indices also take up space. Ideally, all of our indices fit in main memory; otherwise, we pay the cost of disk swap.

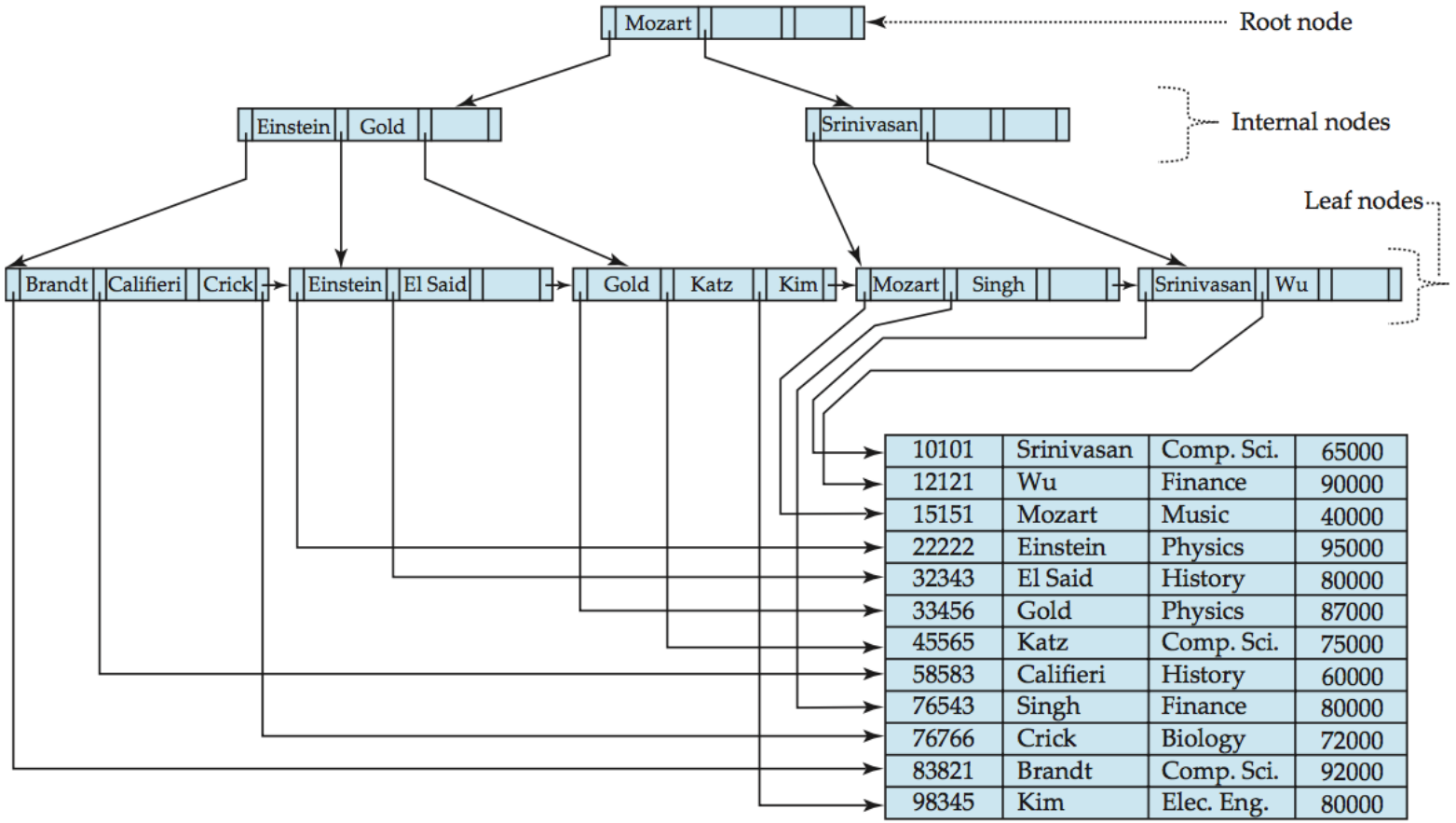
In short, database writes become increasingly expensive with more indices.

Exercise 4-6

Insert the following keys into the B+ tree below with degree d=2:

`Adams`

`Lamport`



Exercise 4-6 Solution (1/2)

As an aside, we know the degree is 2 because the nodes/blocks have at most $4 = 2d$ child pointers.

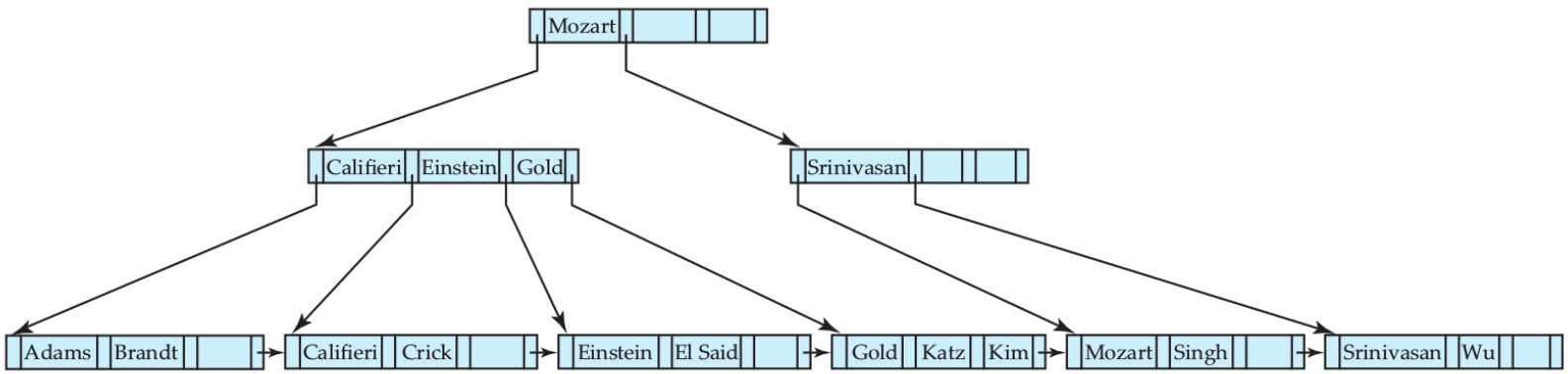
`insert(Adams)`

Follow the tree to find out where to insert Adams.

This ends up being left of Brandt/Califieri/Crick, but that block is full. (Conceptually) add it anyways.

Split the oversized block into Adams/Brandt and Califieri/Crick.

Promote Califieri (leftmost key of right split) as a key in the parent such that the parent becomes Califieri/Einstein/Gold.



Exercise 4-6 Solution (2/2)

`insert(Lamport)`

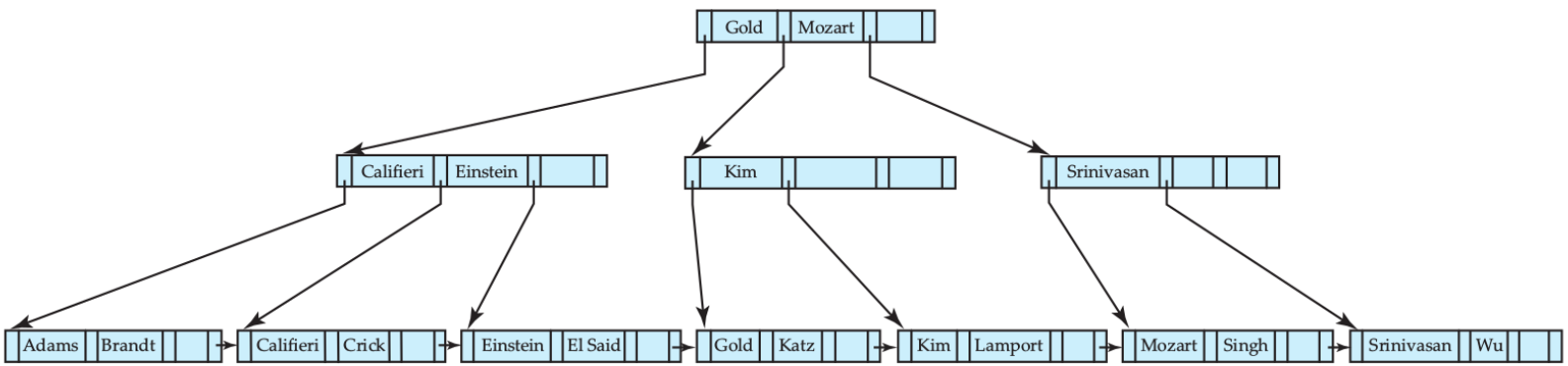
Follow the tree to find out where to insert Lamport.

It goes right of Gold/Katz/Kim, but the block is full.

As before, add Lamport conceptually, split the block into Gold/Katz and Kim/Lamport, and promote Kim.

Now the split blocks' parent (Califieri/Einstein/Gold/Kim) is oversized. Repeat the process.

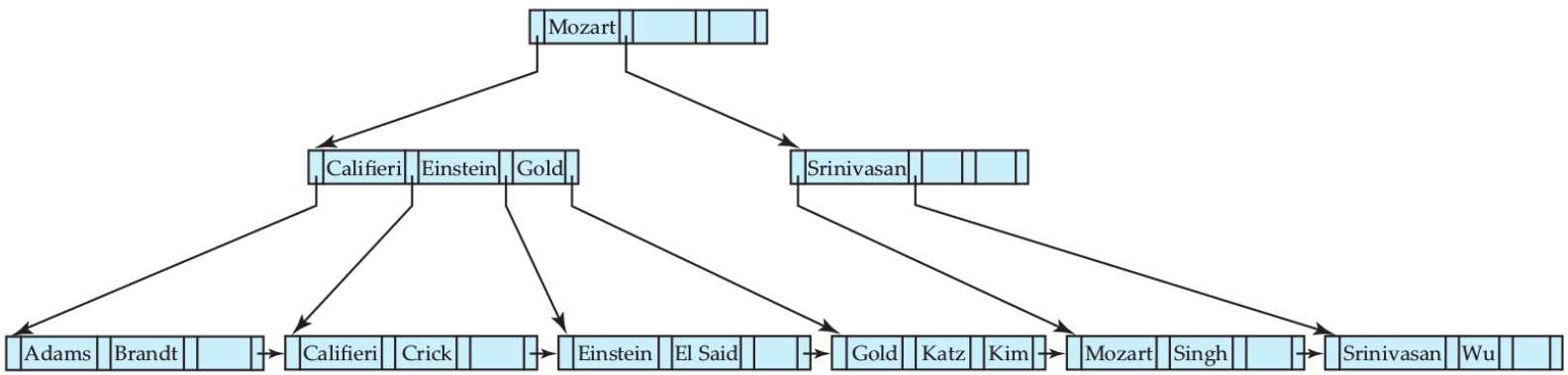
Split into Califieri/Einstein and Gold/Kim, and promote Gold to the root, which becomes Gold/Mozart.



Exercise 4-7

Delete the following keys from the B+ tree below with degree *d*=2. Note that this tree does not have the value Lamport inserted, only Adam.

```
Srinivasan
Singh
Wu
Gold
```



Exercise 4-7 Solution (1/3)

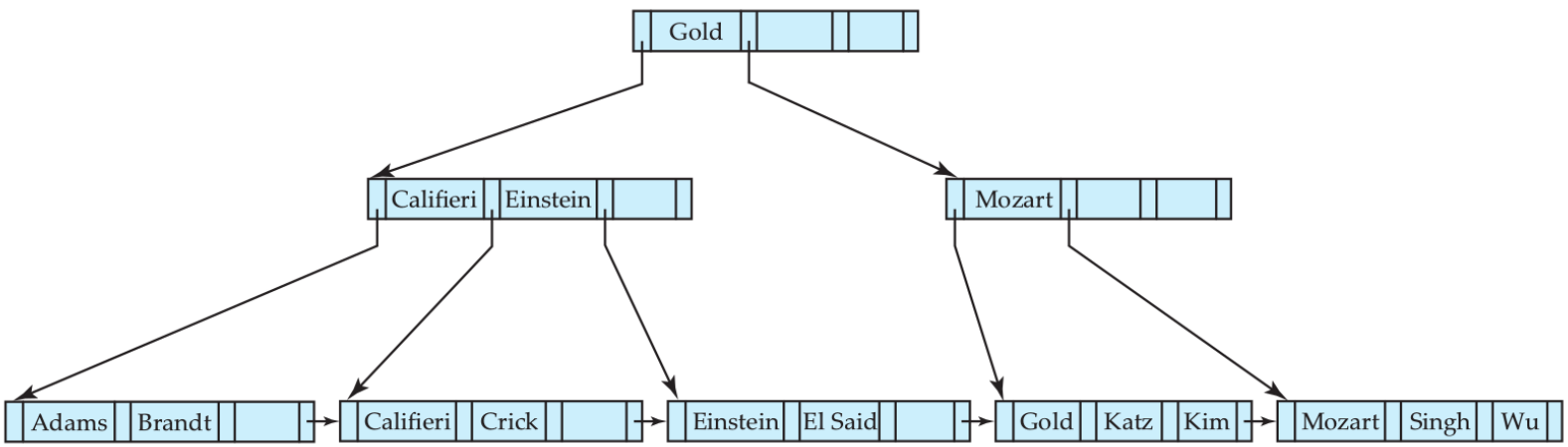
`delete(Srinivasan)`

Remove Srinivasan from the Srinivasan/Wu node to leave Wu.

It is undersized and can merge with a sibling, so we merge Wu with Mozart/Singh to get Mozart/Singh/Wu.

The intermediate node Srinivasan only has one child, but cannot merge with its sibling Califieri/Einstein/Gold, so it steals a child of its sibling, Gold/Katz/Kim.

Update the keys of the intermediate nodes such that lookup still works.

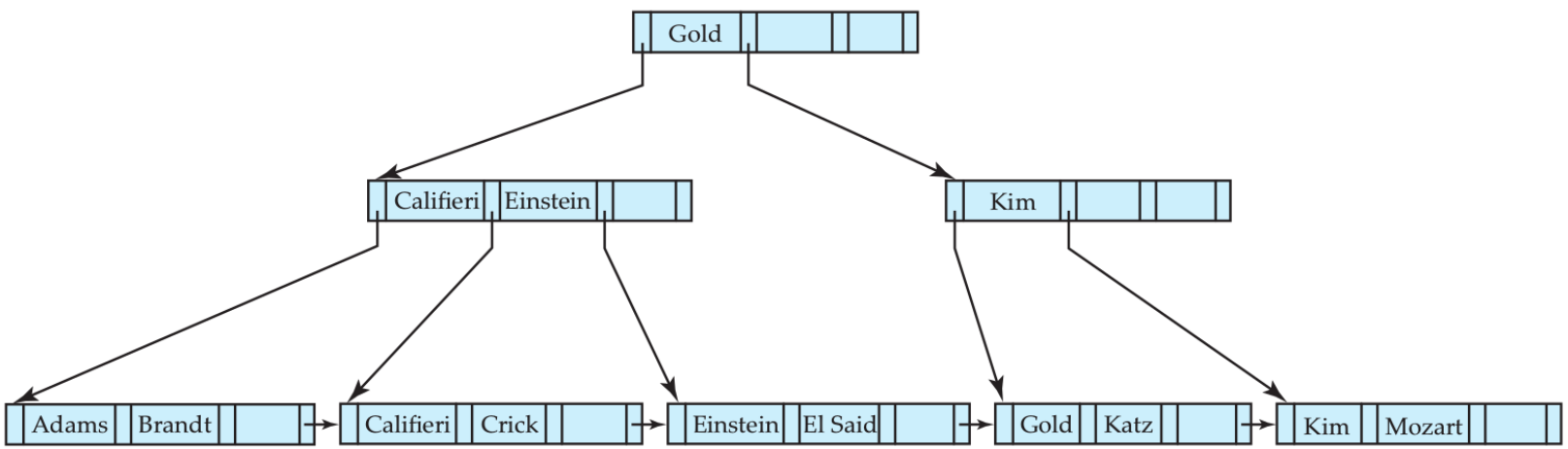


Exercise 4-7 Solution (2/3)

```
delete(Singh)
delete(Wu)
```

Removal of Singh and Wu are easy, since nothing complicated needs to happen.

The leaves after their removal violate no invariants and cannot be merged.



Exercise 4-7 Solution (3/3)

`delete(Gold)`

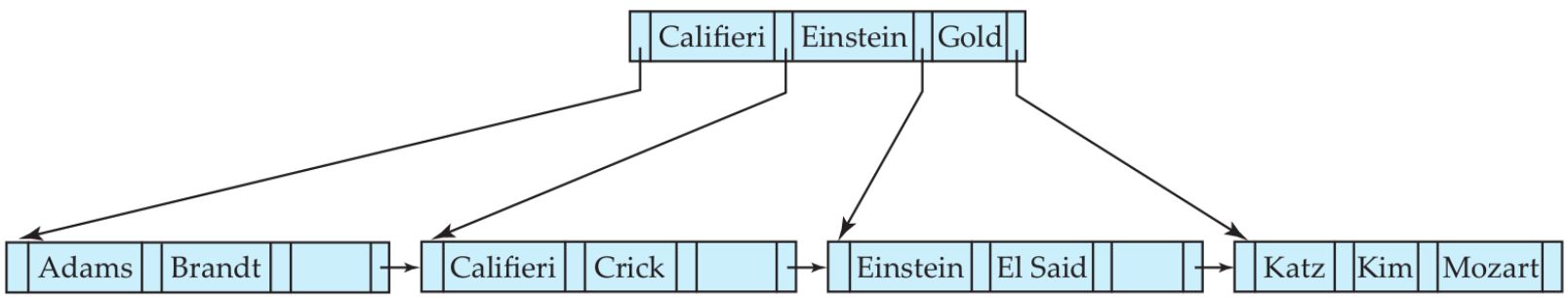
Removing Gold from Gold/Katz leaves a leaf that is undersized and can merge with Kim/Mozart to get Gold/Kim/Mozart.

The intermediate node Kim now is undersized, and is **able to merge** with its sibling, Califieri/Einstein, so it does.

We are left with one intermediate node, Califieri/Einstein/Kim, under the root node Gold.

The key replacement here merges keys from the root and the other intermediate node, leaving us with a new root, Califieri/Einstein/Gold.

An algorithm that leaves us with Califieri/Einstein/Katz would also be correct.



Resources

The B+ tree example discussed in these slides are sourced from `Database System Concepts, Sixth Edition`, **starting on page 491**. In case you don't have the book, Professor Google is kind.