

# Unit #7: B<sup>+</sup>-Trees

CPSC 221: Basic Algorithms and Data Structures

Anthony Estey, Ed Knorr, and Mehrdad Oveis

2016W2

# Unit Outline

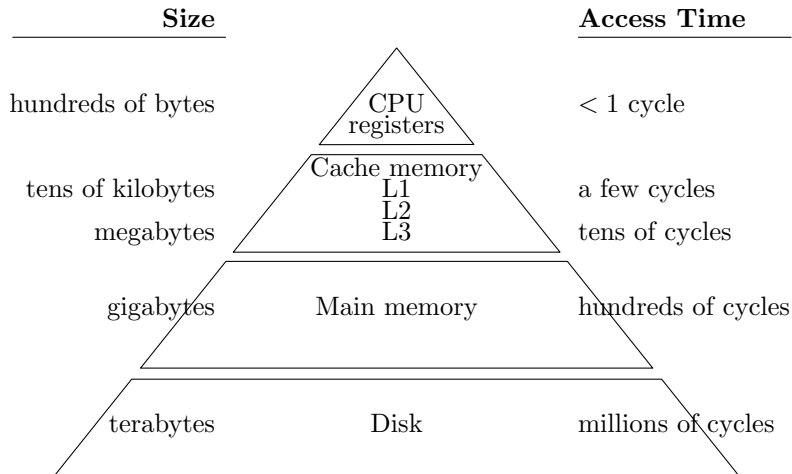
- ▶ Minimizing disk I/Os
- ▶ B<sup>+</sup>-Tree properties
- ▶ Implementing B<sup>+</sup>-Tree insert and delete
- ▶ Some final thoughts on B<sup>+</sup>-Trees

# Learning Goals

- ▶ Describe the structure, navigation and time complexity of a B<sup>+</sup>-Tree.
- ▶ Insert and delete keys from a B<sup>+</sup>-Tree.
- ▶ Relate  $M$ ,  $L$ , the number of nodes, and the height of a B<sup>+</sup>-Tree.
- ▶ Compare and contrast B<sup>+</sup>-Trees with other data structures.
- ▶ Justify why the number of I/Os becomes a more appropriate complexity measure (than the number of CPU operations) when dealing with large datasets and their indexing structures (e.g., B<sup>+</sup>-Trees).
- ▶ Explain the difference between a B-Tree and a B<sup>+</sup>-Tree

# Memory Hierarchy

Why worry about the number of disk I/Os?



# Time Cost: Processor to Disk

## Processor

- ▶ Operates at a few GHz (gigahertz = billion cycles per second)
- ▶ Several instructions per cycle
- ▶ Average time per instruction  $< 1$  ns (1 nanosecond =  $10^{-9}$  seconds)

## Disk (HDD = Hard (spinning) Disk Drive)

- ▶ Seek time  $\approx 10$  ms (1 millisecond =  $10^{-3}$  seconds)
- ▶ Note: Solid State Drives (SSDs) have “seek time”  $\approx 0.03$  ms

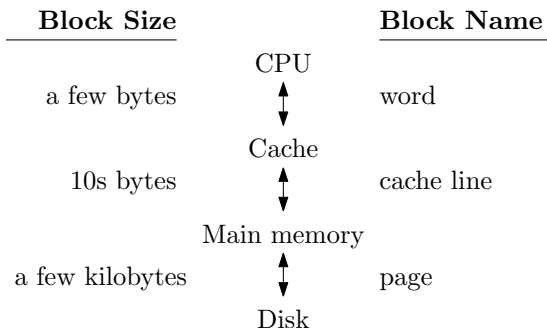
**Result:**  $\approx 10$  million instructions for each disk read!

**Hold on...** How long does it take to read a 1 TB (1 terabyte =  $10^{12}$  bytes) disk?  $1 \text{ TB} \times 10 \text{ ms} = 10 \text{ billion seconds} > 300 \text{ years?}$

**What's wrong?** Each disk read/write moves more than a byte (e.g., 4 KB, 8 KB, ... block sizes). Continuous HDD disk access is about the same speed as on an SSD.

# Memory Blocks

Each memory access to a slower level of the hierarchy fetches a block of data.

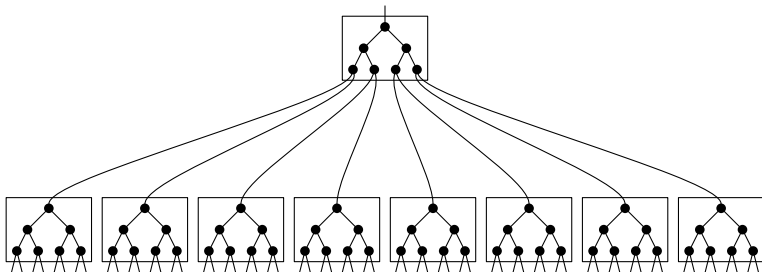


A block is the contents of **consecutive** memory locations.  
So random access between levels of the hierarchy is very slow.

# Chopping Trees into Blocks

## Idea

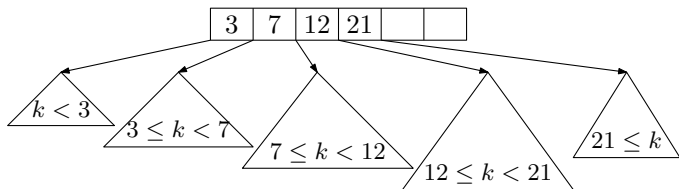
Store data for many adjacent nodes in consecutive memory locations.



## Result

One memory block access provides keys to determine many (more than two) search directions.

# M-ary Search Tree



## M-ary tree property

- ▶ Each node has  $\leq M$  children.

**Result:** Complete  $M$ -ary tree with  $n$  nodes has height  $\Theta(\log_M n)$

## Search tree property

- ▶ Each node has  $\leq M - 1$  search keys:  $k_1 < k_2 < k_3 \dots$
- ▶ All keys  $k$  in  $i$ th subtree obey  $k_i \leq k < k_{i+1}$  for  $i = 0, 1, \dots$

Disk I/O's (runtime) for find:



# B<sup>+</sup>-Trees

B<sup>+</sup>-Trees of order  $M$  are specialized  $M$ -ary search trees:

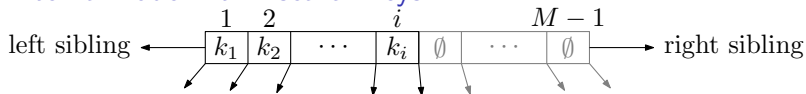
- ▶ ALL leaves are at the same depth!
- ▶ Internal nodes have between  $\lceil M/2 \rceil$  and  $M$  children.
- ▶ Keys and values are stored only in the leaves. Search keys in internal nodes only direct traffic. **B-Trees store (key, value) pairs at internal nodes.**
- ▶ Leaves hold between  $\lceil L/2 \rceil$  and  $L$  (key, value) pairs.
- ▶ The root is special. If it is an internal page, it has between 2 and  $M$  children. If it is a leaf page, it holds at most  $L$  (key, value) pairs.

## Result

- ▶ Height is  $\Theta(\log_M n)$
- ▶ insert, delete, and find operations visit  $\Theta(\log_M n)$  nodes.
- ▶  $M$  and  $L$  are chosen so that each (full) node fills one page of memory. Each node visit (e.g., disk I/O operation) retrieves about  $M/2$  to  $M$  keys or  $L/2$  to  $L$  (key, value) pairs at a time.

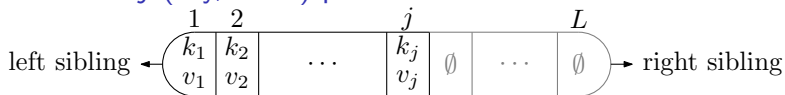
# B<sup>+</sup>-Tree Nodes

Internal node with  $i$  search keys



- ▶  $i + 1$  subtree pointers
- ▶ parent and left & right sibling pointers

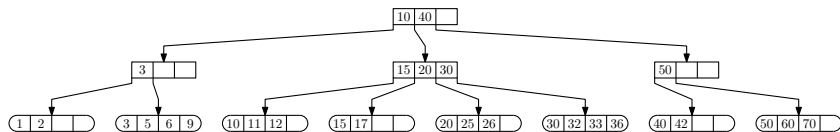
Leaf with  $j$  (key, value) pairs



- ▶ parent and left & right sibling pointers
- ▶ values may be pointers to disk records

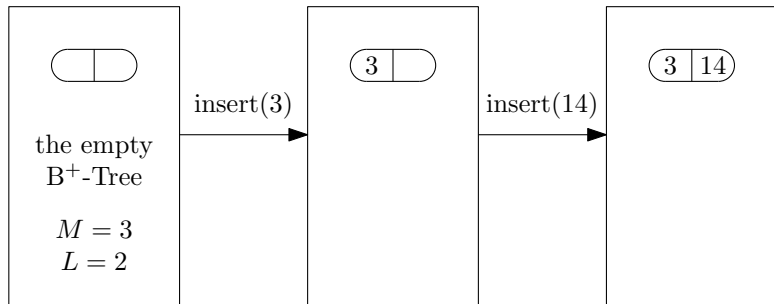
Each node may hold a different number of items.

## Example B<sup>+</sup>-Tree with $M = 4$ and $L = 4$



Values in leaf nodes are not shown.

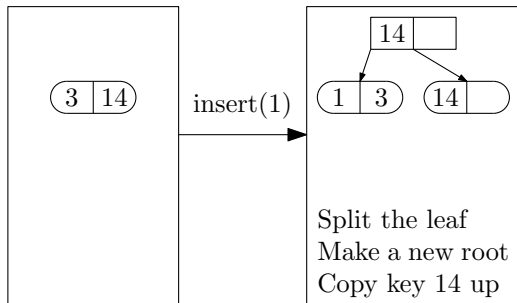
## Making a B<sup>+</sup>-Tree



The root is a leaf.

What happens when we now insert(1)?

## Splitting the Root

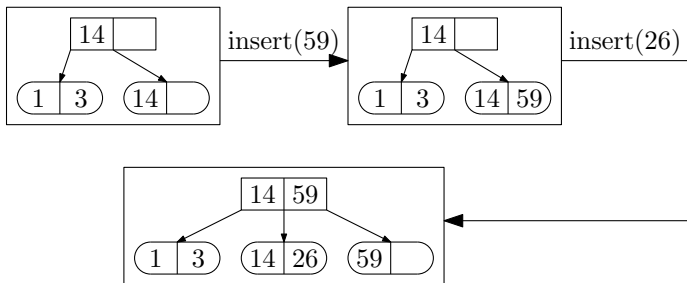


Too many keys for one leaf!

So, make a new leaf and create a parent (the new root) for both.

Why is key 14 duplicated?

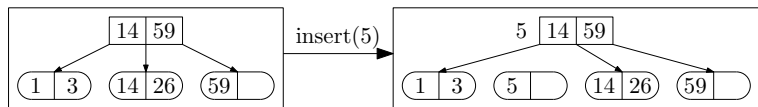
## Splitting a Leaf



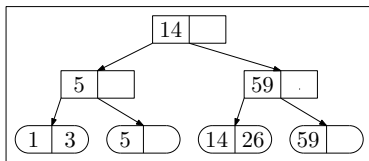
insert(26) causes too many keys for the (14 | 59) leaf to store.

So, make a new leaf and **copy** the “middle” key (the smallest key in the new leaf holding the larger keys) up to the common parent.

# Propagating Splits



Add a new leaf  
Copy key 5 to parent  
There's no room!



Split the internal node  
Add a new parent  
Move key 14 up

insert(5) causes too many keys for [1 | 3] leaf

Copy up key 5 causes too many keys for [14 | 59] node

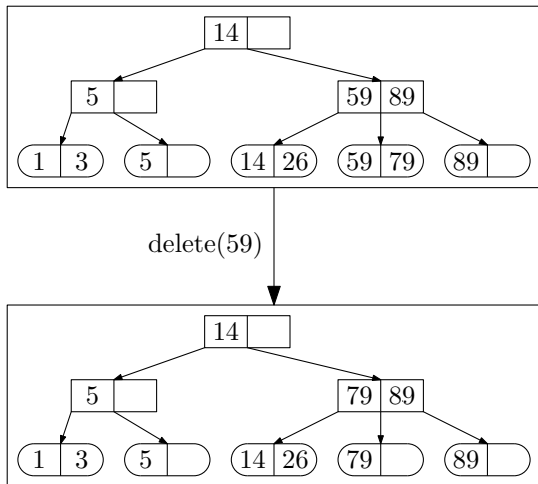
So, make a new internal node and **move up** the middle key.

# Insertion Algorithm

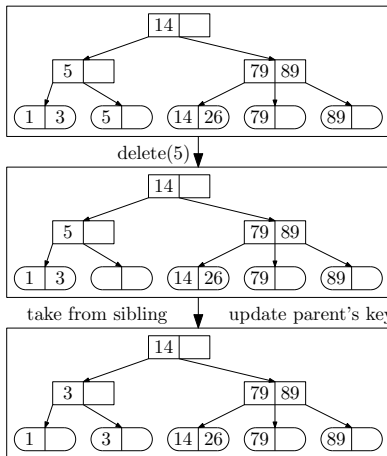
1. Insert (key, value) pair in the target leaf page
2. If the leaf now has  $L + 1$  pairs: // overflow
  - ▶ Split the leaf into two leaves:
    - ▶ Original holds the  $\lceil (L + 1)/2 \rceil$  small key pairs
    - ▶ New one holds the  $\lfloor (L + 1)/2 \rfloor$  large key pairs
  - ▶ **Copy** smallest key in new leaf (the middle key) up to parent
3. If an internal node now has  $M$  keys: // overflow
  - ▶ Split the node into two nodes:
    - ▶ Original holds the  $\lceil (M - 1)/2 \rceil$  small keys
    - ▶ New one holds the  $\lfloor (M - 1)/2 \rfloor$  large keys
  - ▶ If root, hang the new nodes under a new root. Done.
  - ▶ **Move** the remaining middle key up to parent & go to 3



# Delete

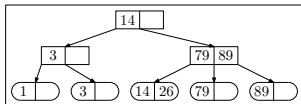


## Delete: Take from a sibling

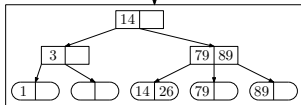


Take 3 from (1 | 3). It has enough items that it can spare one. Update the parent's search key. This is not optional.

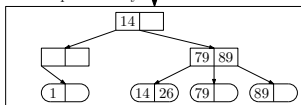
# Delete: Merge



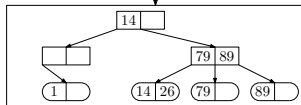
delete(3)



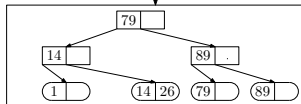
sibling has no spare  
merge with sibling  
delete parent's key



Now parent is underfull

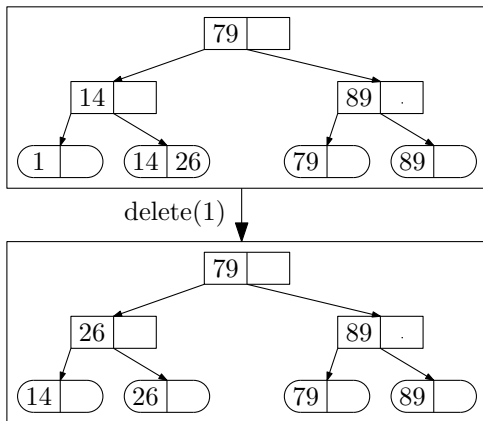


take from its sibling  
update its parent's key

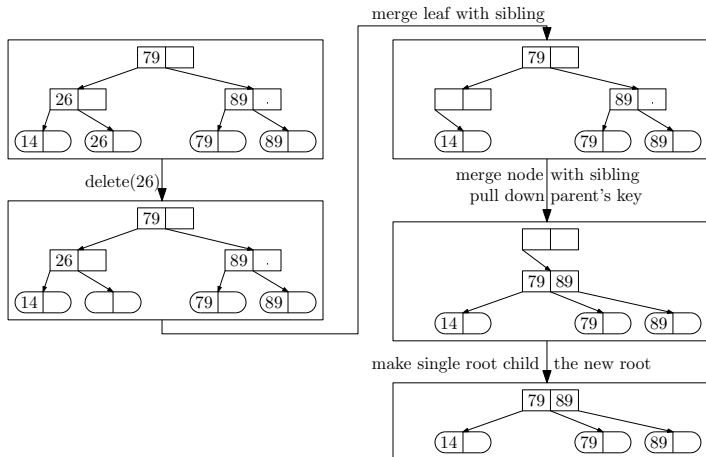


**WARNING:** A leaf is underfull if it holds fewer than  $\lceil L/2 \rceil$  items.  
For  $L > 2$ , an underfull leaf is not empty!

## Delete: Take from a sibling



# Deleting the Root



The root only gets deleted when it has just one subtree (no matter how big  $M$  is).

# Deletion Algorithm

1. Remove the (key, value) pair from the correct leaf.
2. If the leaf now has  $\lceil L/2 \rceil - 1$  items: // underflow
  - ▶ If a sibling has a spare item then take it (smallest from right sibling or largest from left sibling) & update parent's key
  - ▶ Else merge with a sibling & **delete** parent's key
3. If internal non-root node now has  $\lceil M/2 \rceil - 2$  keys: // underflow
  - ▶ If a sibling has a spare child then take it (leftmost from right sibling or rightmost from left sibling) & update parent's key
  - ▶ Else merge with a sibling & **pull down** parent's key & go to 3
4. If the root now has only one child, make that child the new root.

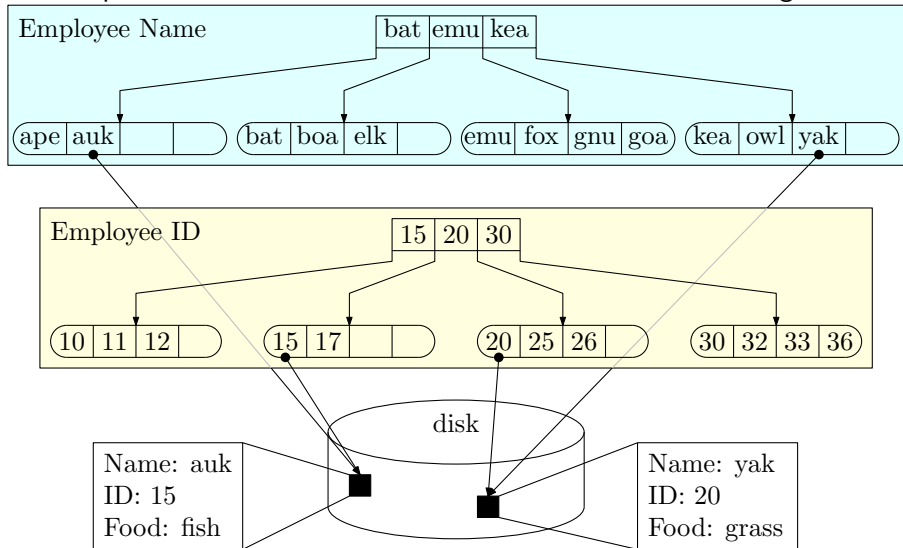
Note: Merge never creates a node with too many items. Why?

## Thinking about B<sup>+</sup>-Trees

- ▶ Deletion is fast if the leaf doesn't underflow or if we can take a (key, value) pair from a sibling. Merging and propagation take more time.
- ▶ Insertion is fast if the leaf doesn't overflow (could we give to a sibling?) Splitting and propagation take more time.
- ▶ Propagation is rare if  $M$  and  $L$  are large. (Why?)
- ▶ Repeated insertions and deletions can cause thrashing.
- ▶ If  $M = L = 128$ , then a B<sup>+</sup>-Tree of height 4 will store at least 30,000,000 items.
- ▶ Range queries (i.e., `findBetween(key1, key2)`) are fast thanks to the sibling pointers in the leaves.

# B<sup>+</sup>-Trees in Practice

Multiple B<sup>+</sup>-Trees can **index** the same data records. This is good.





## A Tree by Any Other Name...

- ▶ B-Trees with  $M = 3$  are called 2-3 trees
- ▶ B-Trees with  $M = 4$  are called 2-3-4 trees