Indexing

Prof. Hyuk-Yoon Kwon
https://sites.google.com/view/seoultech-bigdata

Most parts are based on slides used in Stanford (http://web.stanford.edu/class/cs145)

B+ Trees:
An IO-Aware Index Structure

Today's Lecture

1. Indexes: Motivations & Basics

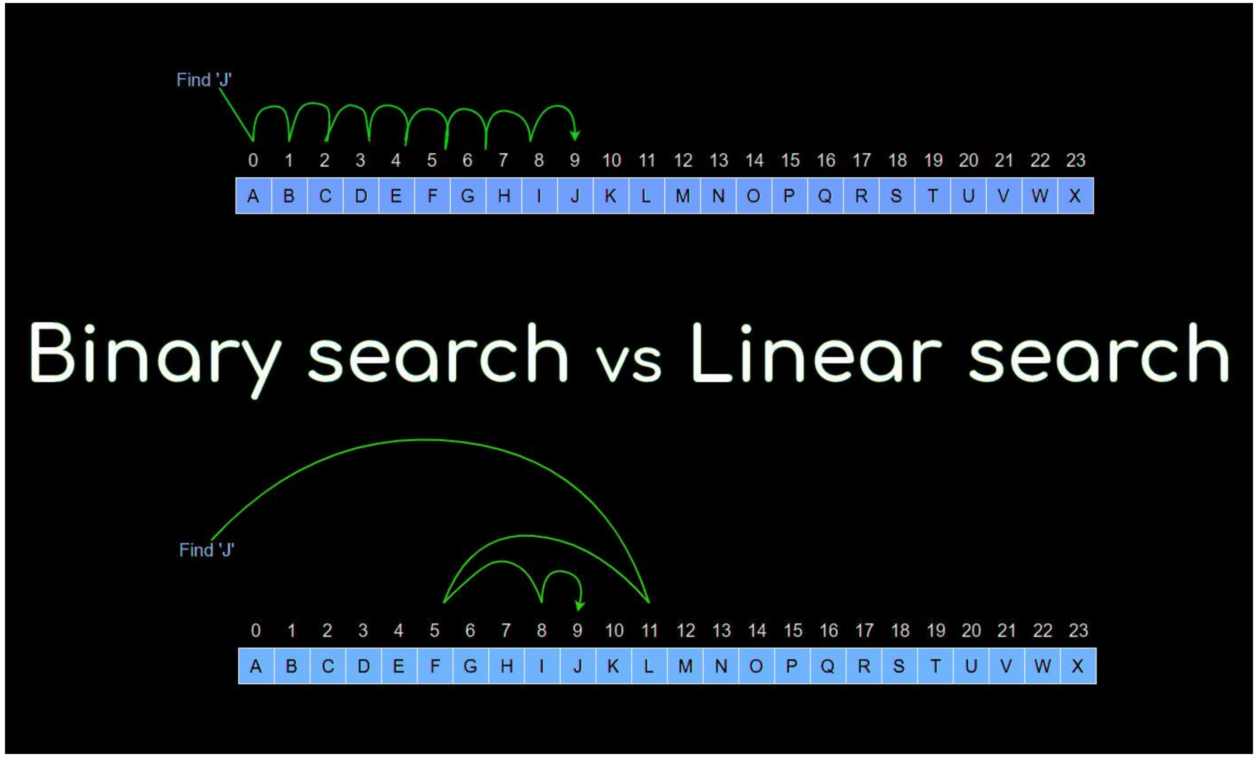
2. B+ Trees

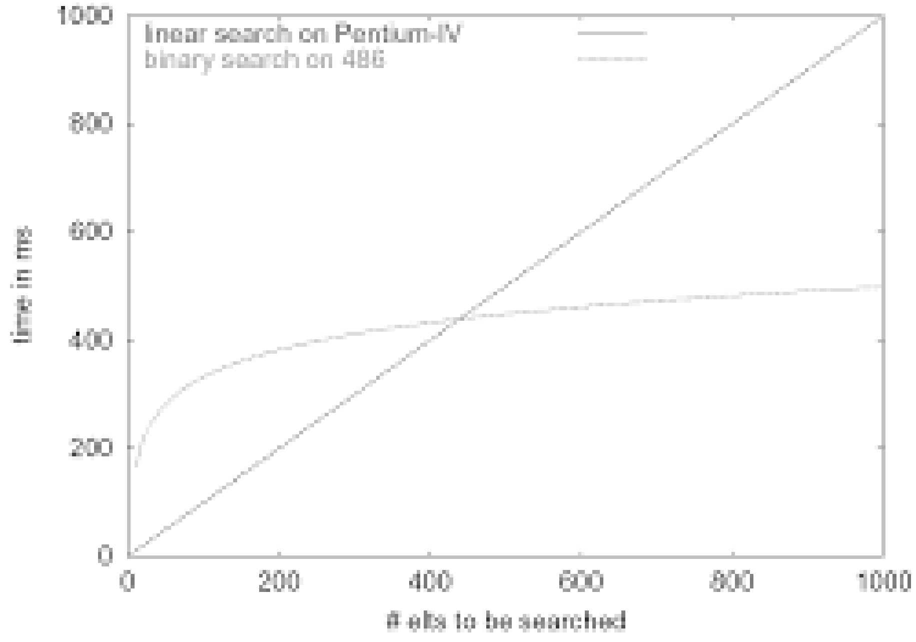
Index Motivation

Person(<u>name</u>, age)

- Suppose we want to search for people of a specific age

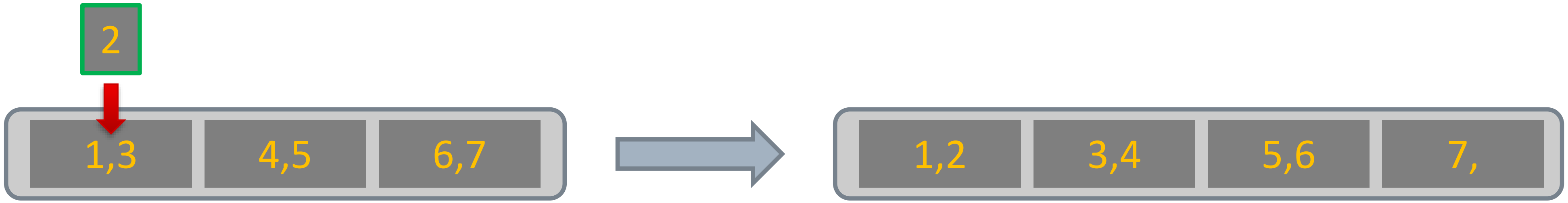
- *First idea:* Sort the records by age… we know how to do this fast!





Index Motivation

- What about if we want to insert a new person, but keep the list sorted?



- We would have to potentially shift *N* records
  - We could leave some “slack” in the pages…

Could we get faster insertions?

Index Motivation

- **What about if we want to be able to search quickly along multiple attributes (e.g., not just age)?**
  - We could keep multiple copies of the records, each sorted by one attribute set… this would take a lot of space

Can we get fast search over multiple attribute (sets) without taking too much space?

We'll create separate data structures called ***indexes*** to address all these points

Further Motivation for Indexes: NoSQL!

- **NoSQL engines are (basically) *just indexes!***

  - A lot more is left to the user in NoSQL… one of the primary remaining functions of the DBMS is still to provide index over the data records, for the reasons we just saw!

Indexes are critical across all DBMS types

Indexes: High-level

- **An _index_ on a file speeds up selections on the _search key fields_ for the index.**
  - Search key properties
    - Any subset of fields
    - is **not** the same as *key of a relation*

- ***Example:***

Product(name, maker, price)

On which attributes would you build indexes?

More precisely

- **An *index* is a data structure mapping search keys to sets of rows in a database table**
  - Provides efficient lookup & retrieval by search key value- usually much faster than searching through all the row s of the database table
- **An index can store the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)**
  - We'll mainly consider secondary indexes

Conceptual Example

What if we want to return all books published after 1867? The above table might be very expensive to search over row-by-row…

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

```
SELECT *
FROM Russian_Novels
WHERE Published > 1867
```

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	***001***
1877	***003***

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punish ment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

Maintain an index for this, and search over that!

Why might just keeping the table sorted by year not be good enough?

Conceptual Example

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punish ment*	Dostoyevsky	1866	…
003	*Anna Karenina*	Tolstoy	1877	…

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punish ment	002
Tolstoy	Anna Karenina	003
Tolstoy	War and Peace	001

Can have multiple indexes to support multiple search keys

Indexes shown here as tables, but in reality we will use more efficient data structures…

Covering Indexes

By_Yr_Index

Published	BID
1866	002
1869	001
1877	003

We say that an index is **<u>covering</u>** *for a specific query* if the index contains all the needed attributes- ***meaning the query can be answered using the index alone!***

The "needed" attributes are the union of those in the SELECT and WHERE clauses…

Example:

```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

High-level Categories of Index Types

- **B-Trees *(covered next)***
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will look at a variant called **B+ Trees***

- **Hash Tables *(not covered)***

Practice – Create Index

- **Use the sample table in the previous lecture, which includes 100,000 tuples**
- **Compare the execution of the following SQL query for the cases of using index and not using index**
  - SELECT t1.value1

    FROM SampleT t1, SampleT t2

    WHERE t1.value2 > t2.value2

    GROUP BY t1.value1

    HAVING t1.value1 > 9990;
  - How to build index
    - CREATE INDEX value2_idx ON SampleT(value2);
  - How to remove index
    - DROP INDEX value2_idx;

B+ Trees

B+ Trees

Search trees

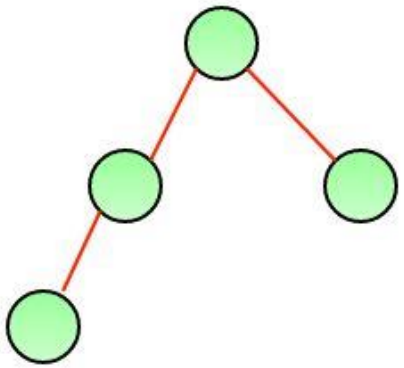
- B does not mean binary!

Idea in B Trees:

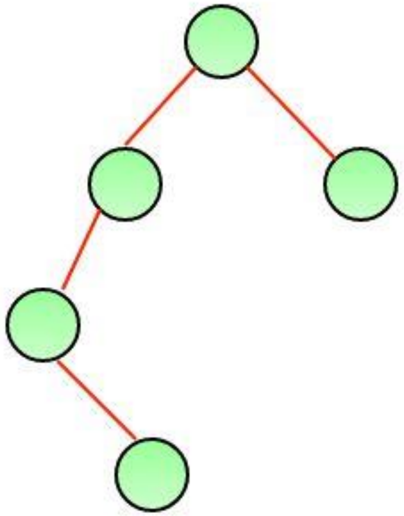
- make 1 node = 1 physical page
- Balanced tree

Idea in B+ Trees:

- Make leaves into a linked list (for range queries)



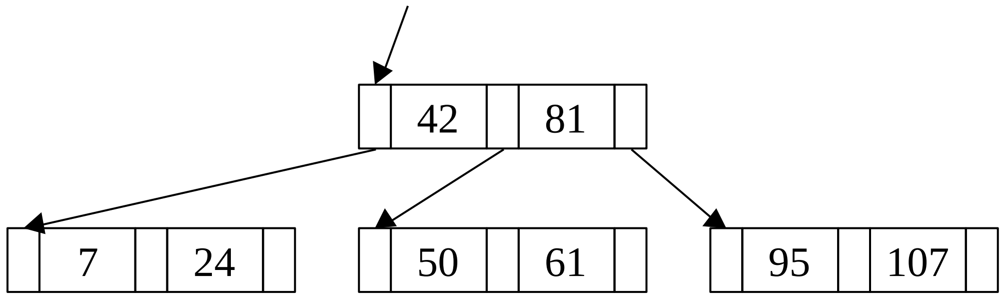
A height balanced tree



Not a height balanced tree

B-Tree

- **Keys are stored in either leaf nodes or non-leaf nodes**
  - Keys stored in a non-leaf are not duplicated in a leaf
- **B-tree of order d**
  - Each node contains slots for 2d keys and (2d + 1) pointers



Operations on B-trees (1/5)

Insertion

1. Search to the leaf node

2. If there is an empty key slot then

   insert a new element

3. Otherwise (i.e., if the node is to split)

   a. Create a new leaf node

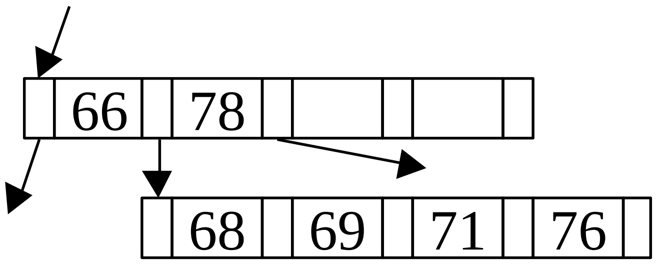
   b. Evenly distribute keys to two nodes (old + new)

   c. Promote the center value to the higher-level node
      (erase from the leaf : insert at the higher-level node)

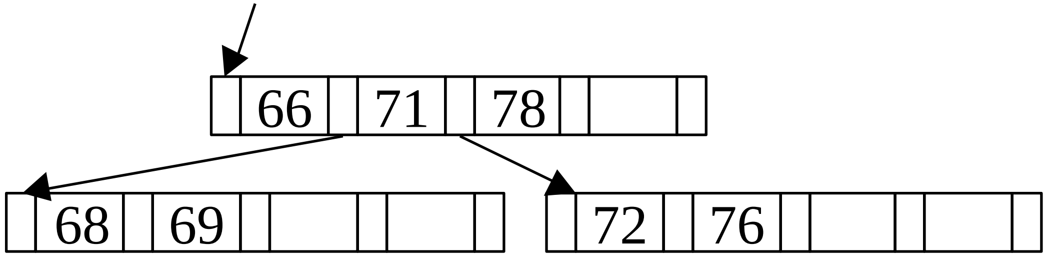
   d. Propagate split if the higher-level node gets full

Operations on B-trees (2/5)

- Example



insert 72



Operations on B-trees (3/5)

- **Deletion**

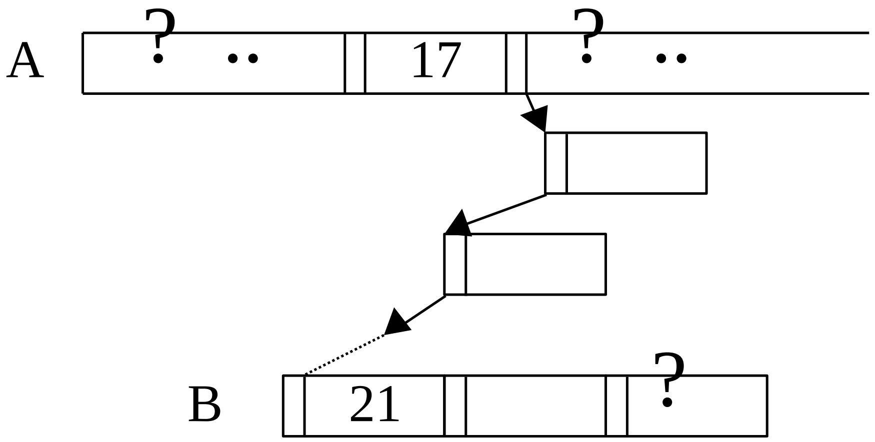
1. Find the key to be deleted
2. Deletion

   if the key is in a leaf then delete that entry
   else if the key is in a nonleaf node then

   a. Delete this key
   b. Promote the next higher value to this slot
      (**We assume** that the next-higher value is always in the leaf that is the leftmost leaf in the right subtree)
      - To maintain the correct structure (pointers + ordering)

Operations on B-trees (4/5)

- Example



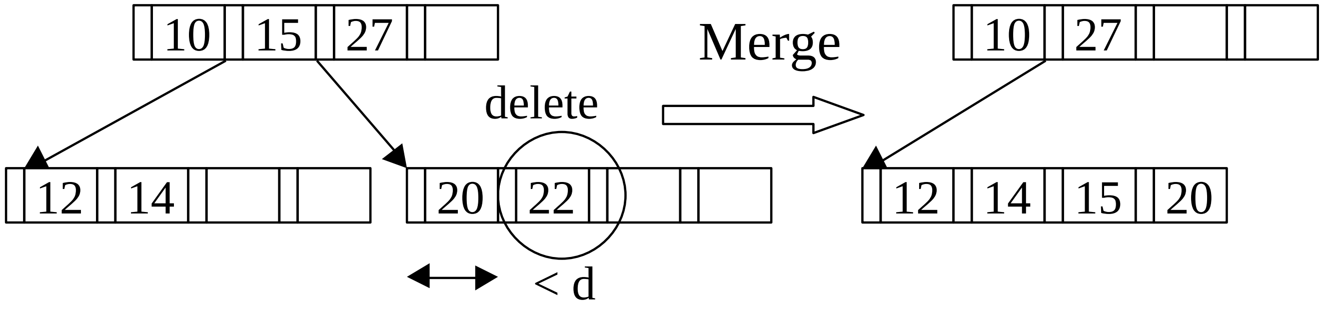
Delete 17 and promote 21 to node A

Then, effectively, one key is deleted from B (a leaf)

Operations on B-trees (5/5)

3. If Step 2 causes underflow (i.e., # keys + # keys of an adjacent node < # of maximum entries in the node), then

   a. merge

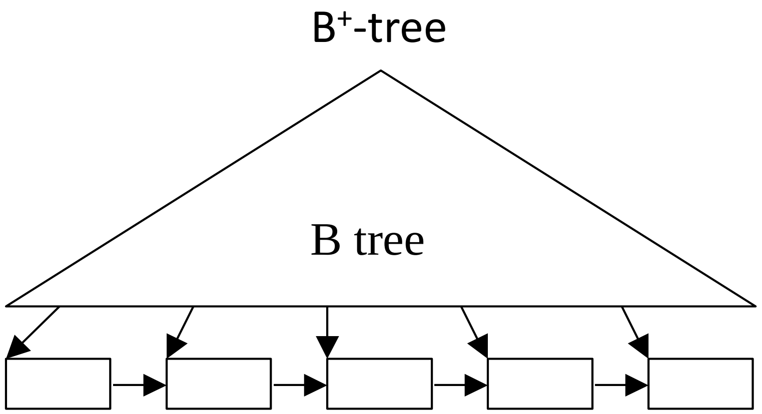


Note : The previous center value (15) has been deleted from the parent node

B+-Tree: Variants of the B-tree

- Similar to B-tree, but
- Leaf-level always has all the keys

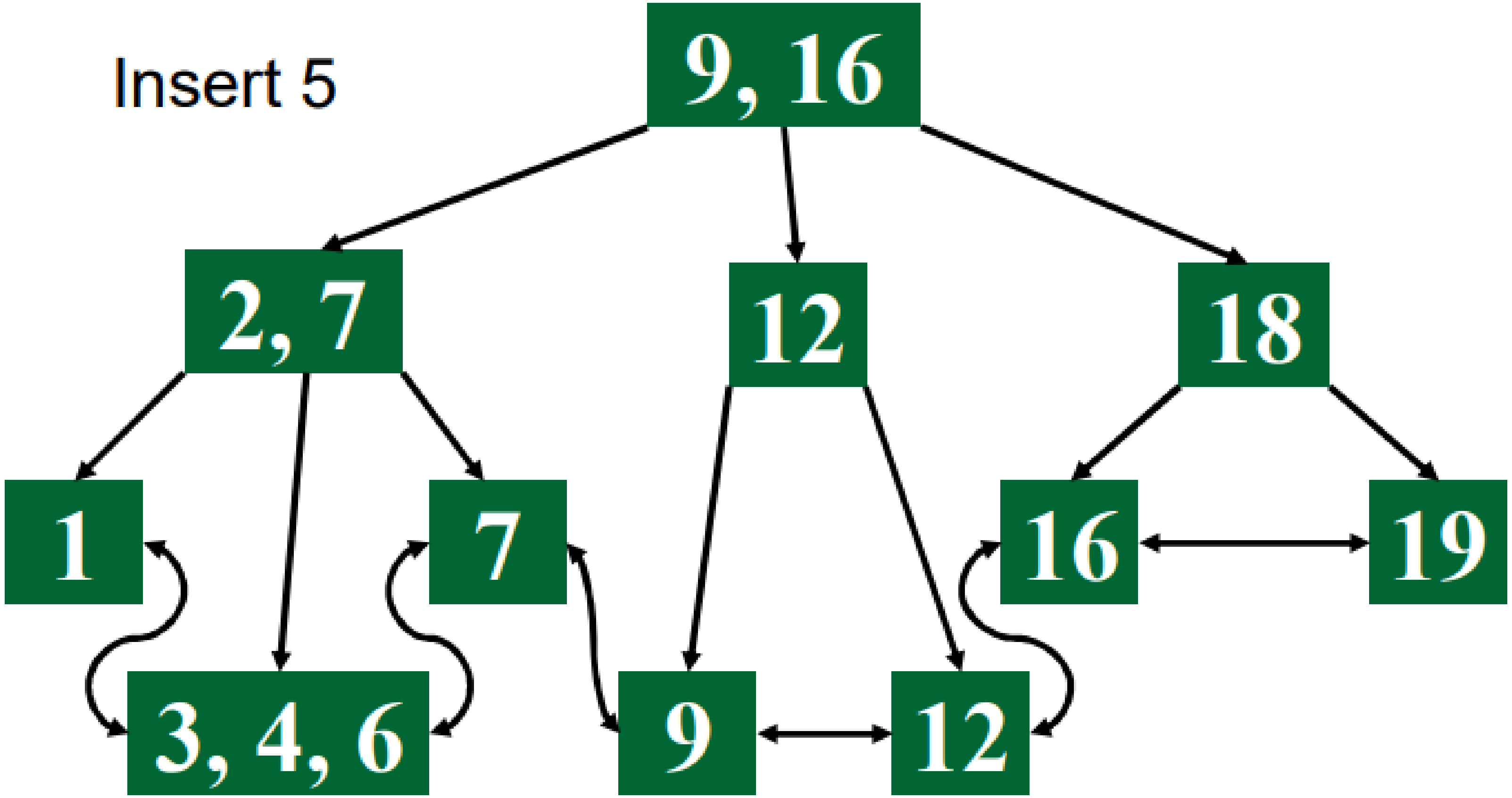
  (Thus, some keys are duplicated)
- Leaf-level nodes are chained by pointers according to the sort order
- Advantages
  - Faster sequential processing
- Disadvantages
  - Duplicate key values
  - More maintenance cost



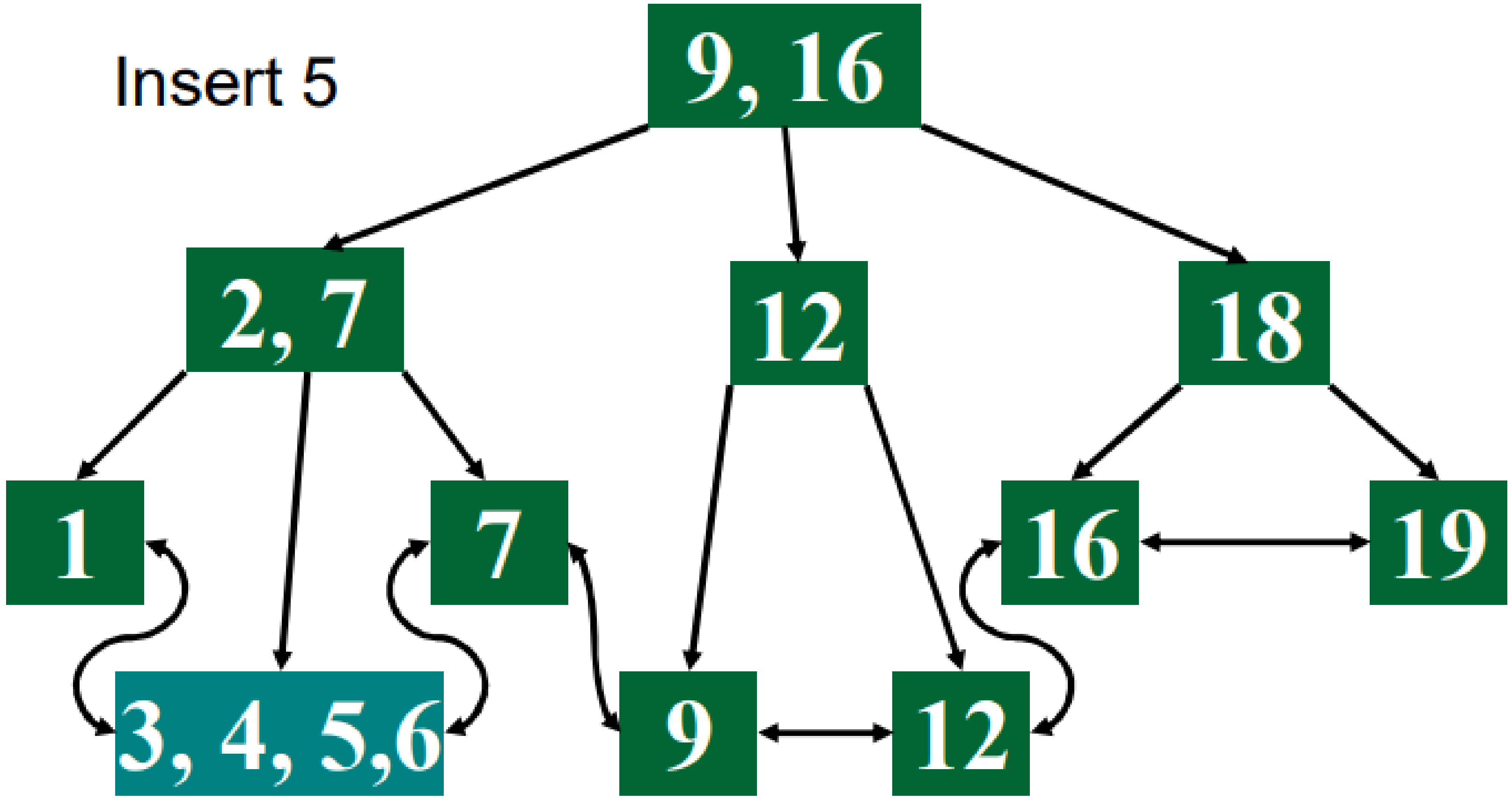
B+-Tree Insertion

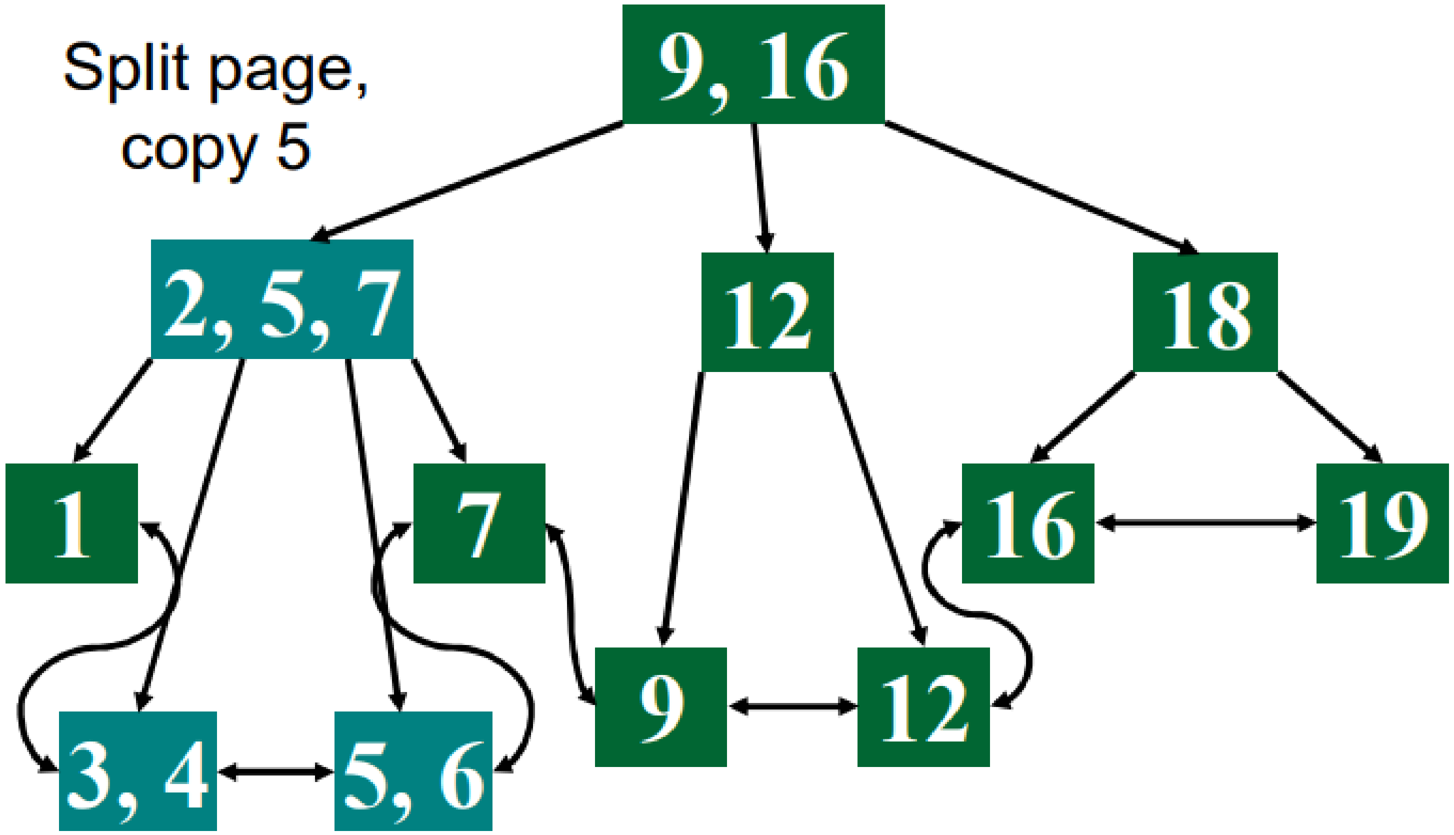
- **Procedures**
  - Insert at bottom level
  - If leaf node overflows (i.e., no empty slot), split the node and **copy** middle element to next higher non-leaf node
  - If non-leaf node overflows, split page and **move** middle element to next higher non-leaf node

Insert 5



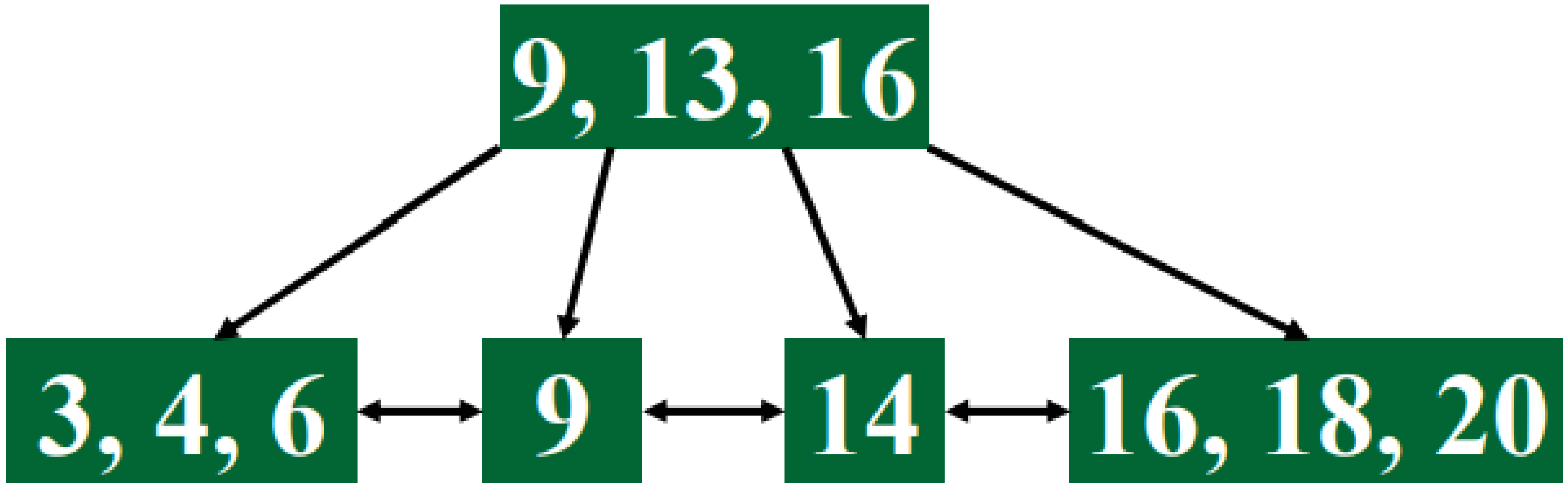
Insert 5



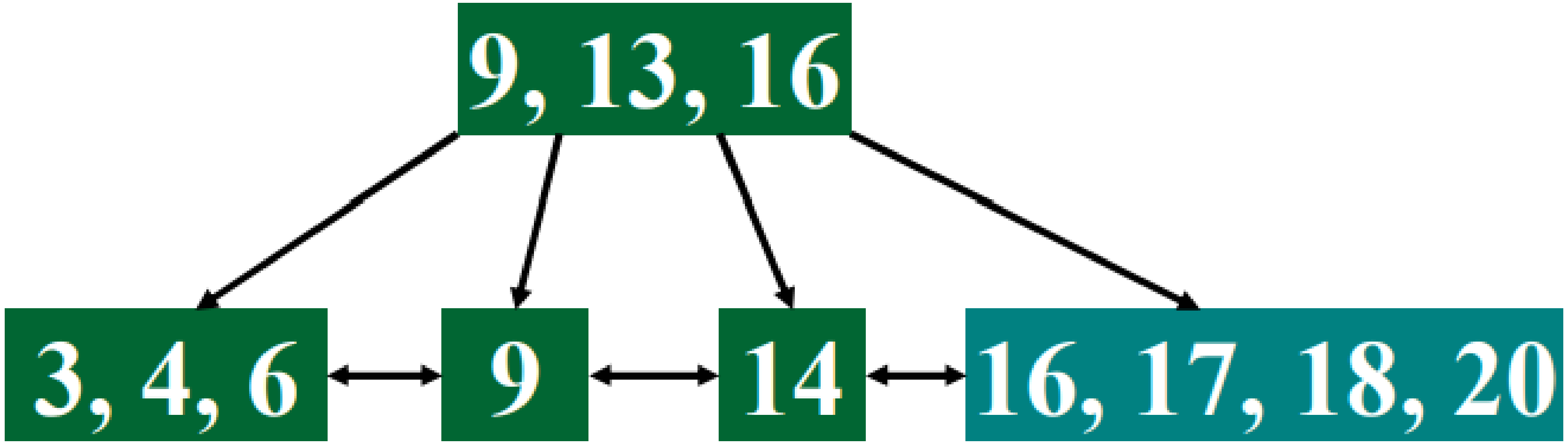

Split page,
copy 5
9, 16
2, 5, 7
12
18
1
7
16
19
9
12
3, 4
5, 6

Practice: Insert a new value in B+-tree

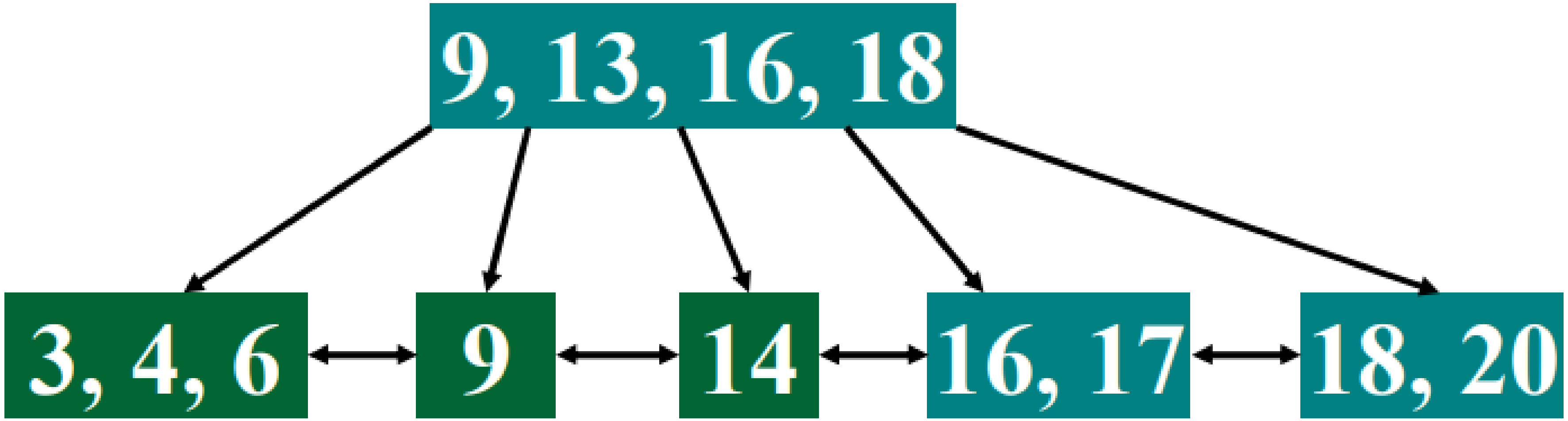
Insert 17



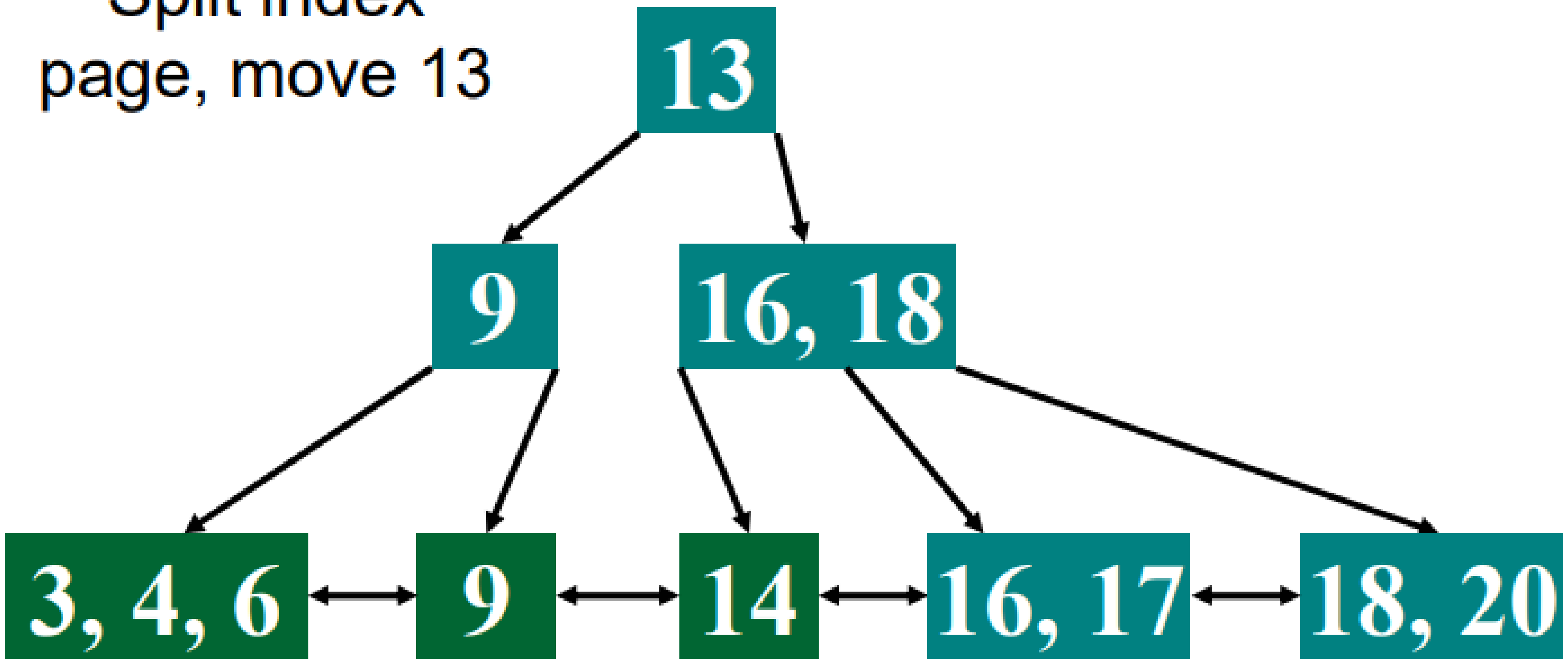
Insert 17



Split leaf
page, copy 18



Split index page, move 13

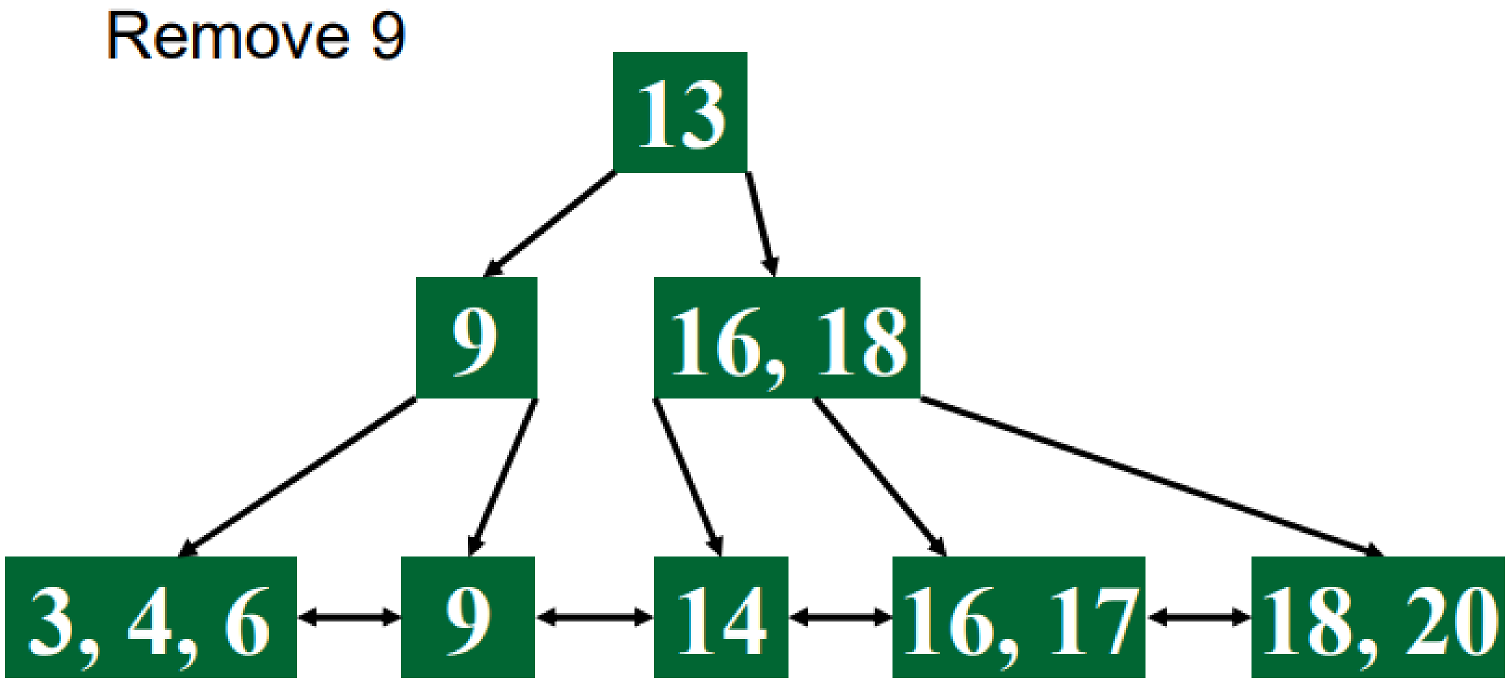


B+-Tree Deletion

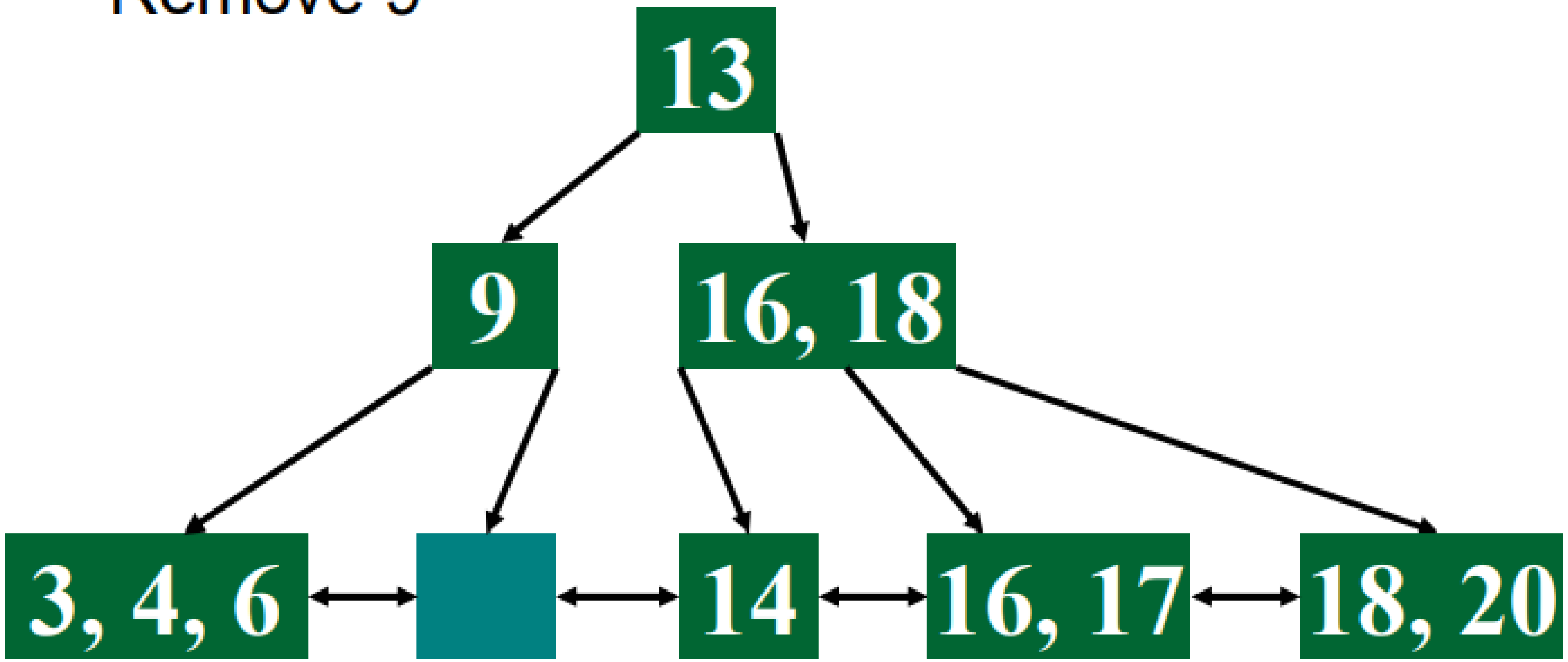
Procedures

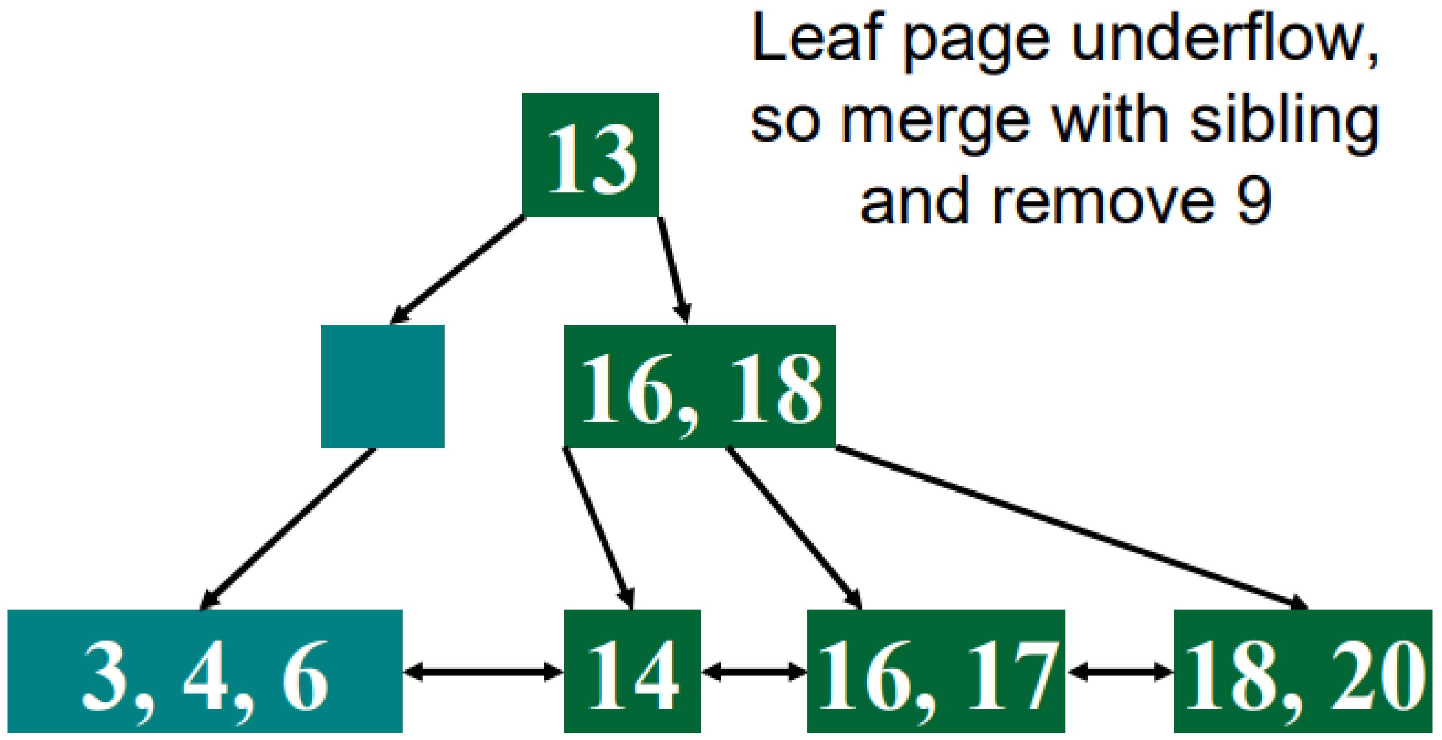
- Delete key from leaf node
- If leaf node underflows (i.e., # keys + # keys of an adjacent node <= # of maximum entries in the node), merge with adjacent leaf nodes
  - Delete the key in the non-leaf node
- If non-leaf node underflows (i.e., # keys + # keys of an adjacent node < # of maximum entries in the node), merge with adjacent non-leaf nodes
  - Move down the key from the next non-leaf node

Practice: Delete the value from B+-tree

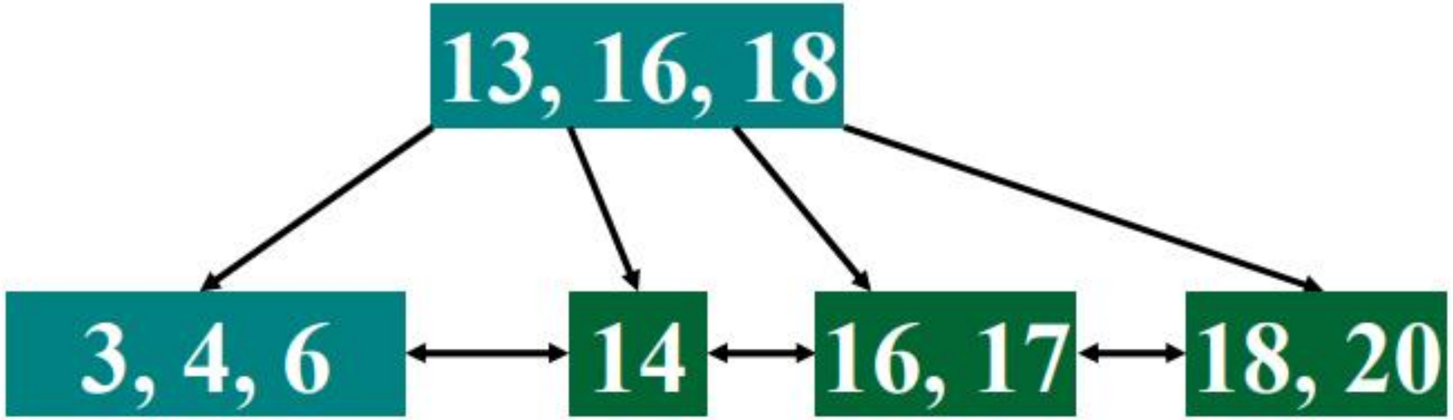


Remove 9




Leaf page underflow,
so merge with sibling
and remove 9
13
16, 18
3, 4, 6
14
16, 17
18, 20

Index page underflow, so merge with sibling and demote 13



Searching a B+ Tree

- **For exact key values:**
  - Start at the root
  - Proceed down, to the leaf

```
SELECT name
FROM   people
WHERE  age = 25
```

- **For range queries:**
  - As above
  - *Then sequential traversal*

```
SELECT name
FROM   people
WHERE  20 <= age
  AND  age <= 30
```

B+ Tree Exact Search Animation

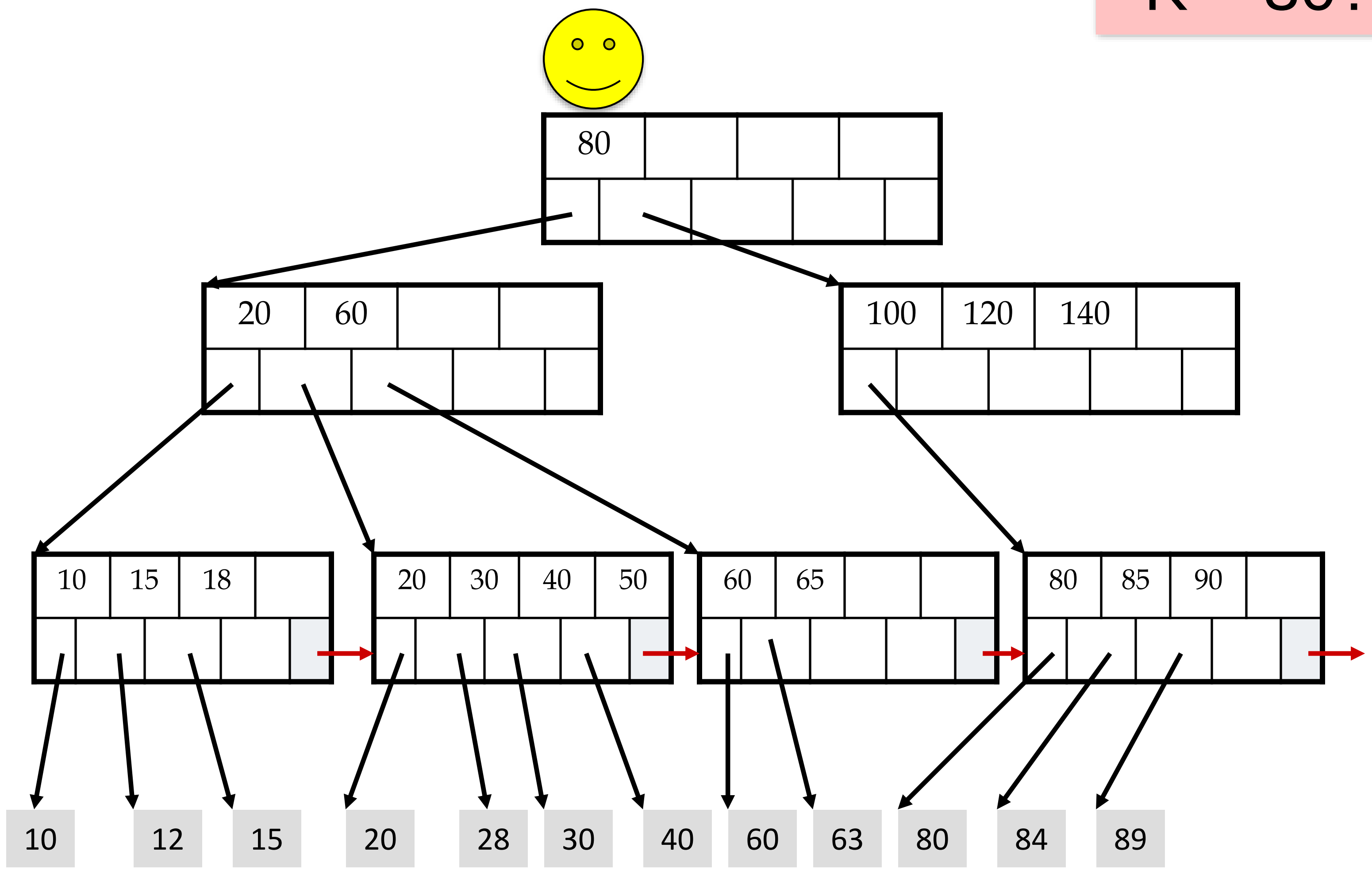
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



B+ Tree Range Search Animation

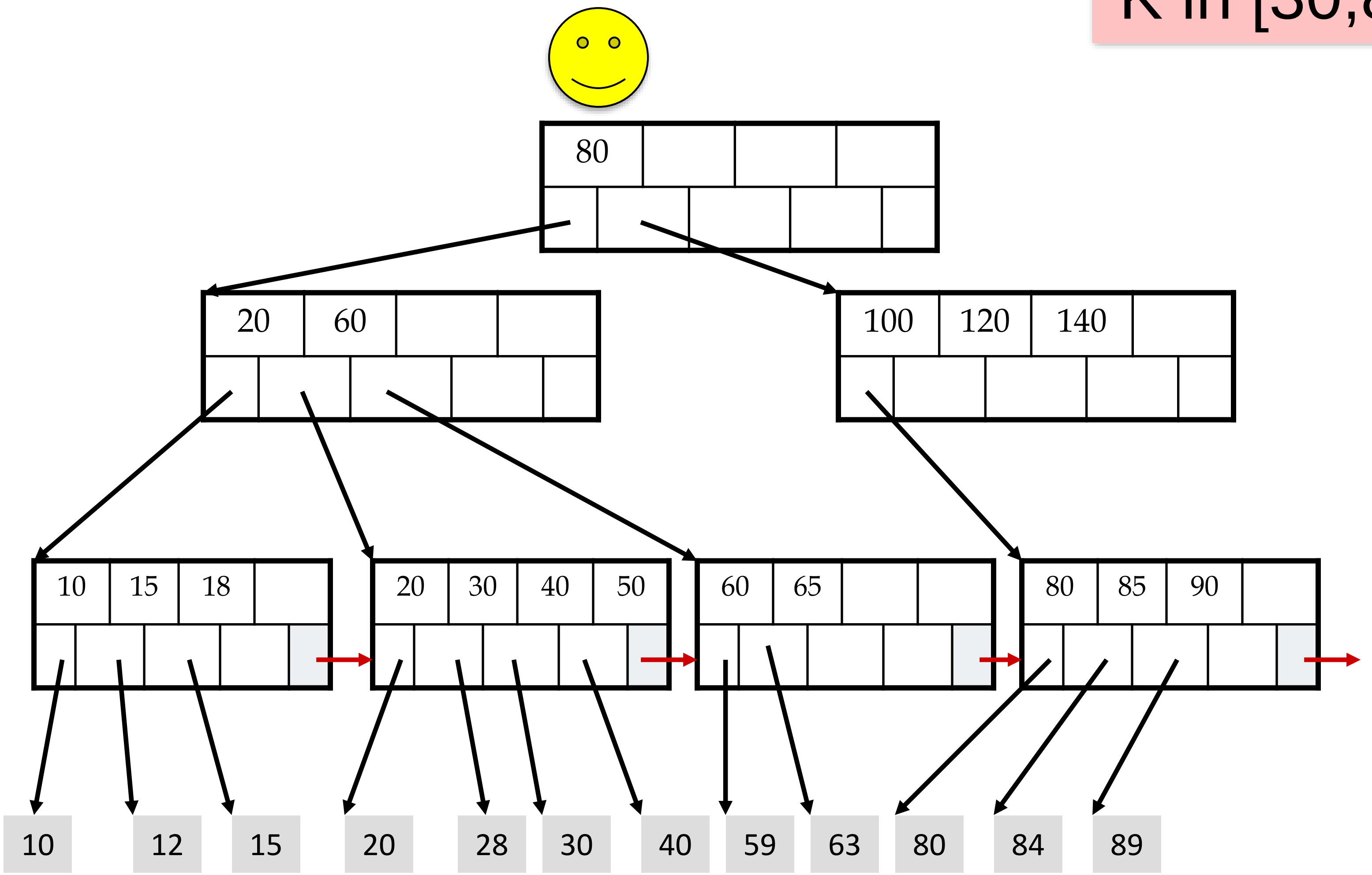
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Summary

- **We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently**

- **We create indexes over tables in order to support *exact and range search* and *insertion* over *multiple s earch keys***

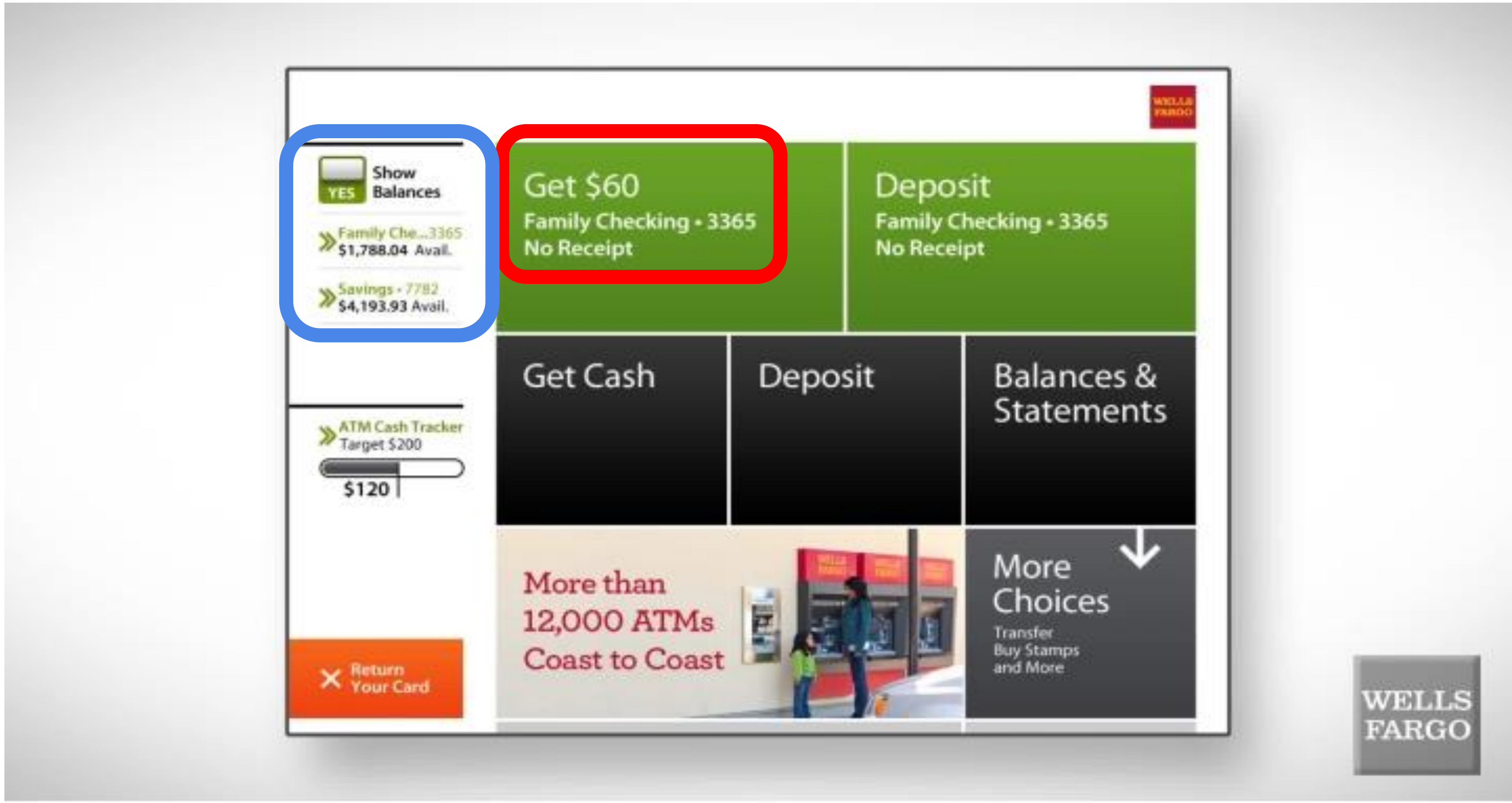
- **B+ Trees are one index data structure which support very fast exact and range search & insertion**

Transactions (Intro)

Example

Unpack ATM DB:

Transaction



Read Balance Give money Update Balance	vs	Read Balance Update Balance Give money

Goals for this lecture

Transactions are a programming abstraction that enables the DBMS to handle *recovery and concurrency* for users.

Application: Transactions are critical for users

Fundamentals: The <u>basics</u> of how TXNs work

- **Transaction processing is part of debates around SQL, noSQL, NewSQL systems**
- **How do TXNs work? What are tradeoffs?**

What you will learn about in this section

1. Our “model” of the DBMS / computer
2. Transactions basics
3. Motivation: Recovery & Durability
4. Motivation: Concurrency

Our model

	Shared
Main Memory (RAM)	1 3
Disk	2 3

1. **Shared:** Each process can read from / write to shared data in main memory

2. **Disk:** Global memory can read from / flush to disk

3. **Log*:*** Assume on stable disk storage- spans both main memory and disk…

Log is a *sequence* from main memory -> disk

<u>“Flushing</u> to disk” = writing to disk from main memory

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

```
START TRANSACTION
    UPDATE Product
    SET Price = Price – 1.99
    WHERE pname = 'Gizmo'
COMMIT
```

Transactions: Basic Definition

A **transaction ("TXN")** is a sequence of one or more ***operations*** (reads or writes) which reflects ***a single real-world transition***.

In the real world, a TXN either happened completely or not at all

Examples:

- Transfer money between accounts
- Purchase a group of products
- Register for a class (either waitlist or allocated)

Transactions in SQL

- In “ad-hoc” SQL:
  - Default: each statement = one transaction
- In a program, multiple statements can be grouped together as a transaction:

```
START TRANSACTION
        UPDATE Bank SET amount = amount – 100
        WHERE name = ‘Bob’
        UPDATE Bank SET amount = amount + 100
        WHERE name = ‘Joe’
COMMIT
```

Motivation for Transactions

Grouping user actions (reads & writes) into *transactions* helps with two goals:

- **Recovery & Durability**: Keeping the DBMS data consistent and durable in the face of crashes, aborts, system shutdowns, etc.

- **Concurrency:** Achieving better performance by parallelizing TXNs *without* creating anomalies

Motivation

1. Recovery & Durability of user data is essential for reliable DBMS usage

- The DBMS may experience crashes (e.g. power outages, etc.)
- Individual TXNs may be aborted (e.g. by the user)

Idea: Make sure that TXNs are either **durably stored in full**, **or not at all**; keep log to be able to “roll-back” TXNs

Protection against crashes / aborts

```
Client 1:
        INSERT INTO SmallProduct(name, price)
        SELECT pname, price
        FROM Product
        WHERE price <= 0.99

        DELETE Product
        WHERE price <=0.99
```

Crash / abort!

What goes wrong?

Protection against crashes / aborts

```
Client 1:
        START TRANSACTION
        INSERT INTO SmallProduct(name, price)
                SELECT pname, price
                FROM Product
                WHERE price <= 0.99

        DELETE Product
                WHERE price <=0.99
        COMMIT OR ROLLBACK
```

Now we’d be fine! We’ll see how / why this lecture

Motivation

2. Concurrent execution of user programs is essential for good DBMS performance.

- Disk accesses may be frequent and **slow**- optimize for throughput (# of TXNs), trade for latency (time for any one TXN)
- Users should still be able to execute TXNs as if in **isolation** and such that **consistency** is maintained

Idea: Have the DBMS handle running several user TXNs concurrently, in order to keep CPUs humming…

Multiple users: single statements

```
Client 1: UPDATE Product
                    SET Price = Price – 1.99
                    WHERE pname = ‘Gizmo’

Client 2: UPDATE Product
                    SET Price = Price*0.5
                    WHERE pname=‘Gizmo’
```

Two managers attempt to discount products *concurrently-*
What could go wrong?

Multiple users: single statements

```
Client 1: START TRANSACTION
                    UPDATE Product
                    SET Price = Price – 1.99
                    WHERE pname = ‘Gizmo’
          COMMIT

Client 2: START TRANSACTION
                    UPDATE Product
                    SET Price = Price*0.5
                    WHERE pname=‘Gizmo’
          COMMIT
```

Now works like a charm-

Properties of Transactions

What you will learn about in this section

- Atomicity
- Consistency
- Isolation
- Durability

Transaction Properties: ACID

- **A**tomic
  - State shows either all the effects of txn, or none of them
- **C**onsistent
  - Txn moves from a state where integrity holds, to another where integrity holds
- **I**solated
  - Effect of txns is the same as txns running one after another (ie looks like batch mode)
- **D**urable
  - Once a txn has committed, its effects remain in the database

ACID: Atomicity

- TXN's activities are atomic: **all or nothing**
  - Intuitively: in the real world, a transaction is something that would either occur *completely* or *not at all*
- Two possible outcomes for a TXN
  - It *commits*: all the changes are made
  - It *aborts*: no changes are made

ACID: Consistency

- The tables must always satisfy user-specified ***integrity constraints***
  - *Examples:*
    - Account number is unique
    - Stock amount can't be negative
    - Sum of *debits* and of *credits* is 0

- How consistency is achieved:
  - Programmer makes sure a txn takes a consistent state to a consistent state
  - *System* makes sure that the txn is **atomic**

ACID: Isolation

- A transaction executes concurrently with other transactions
- **Isolation**: the effect is as if each transaction executes in *isolation* of the others.
  - E.g. Should not be able to observe changes from other transactions during the run

ACID: Durability

- The effect of a TXN must continue to exist (***"persist"***) after the TXN
  - And after the whole program has terminated
  - And even if there are power failures, crashes, etc.
  - And etc…

- Means: Write data to **disk**

Challenges for ACID properties

- In spite of failures: Power failures, but not media failures
- Users may abort the program: need to “rollback the changes”
  - Need to *log* what happened
- Many users executing concurrently
  - Can be solved via locking

And all this with… Performance!!

A Note: ACID is contentious!

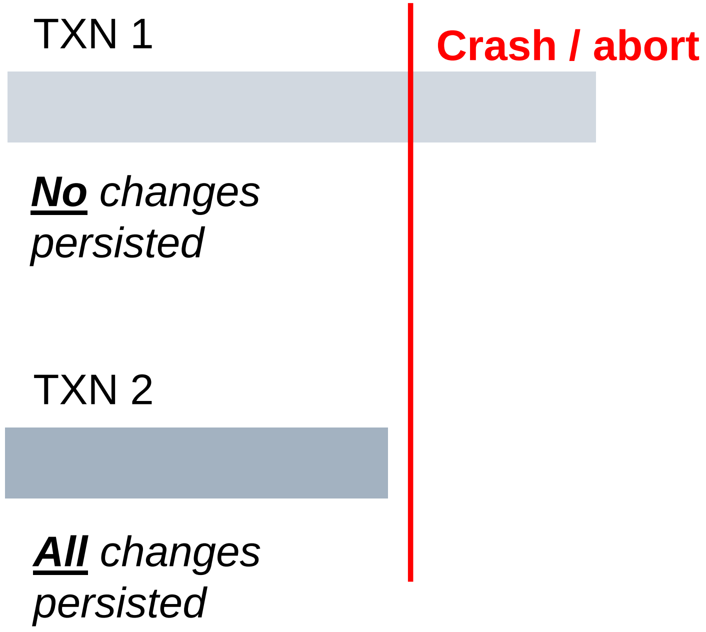
- Many debates over ACID, both **historically** and **currently**
- Many newer “NoSQL” DBMSs relax ACID
- In turn, now “NewSQL” reintroduces ACID compliance to NoSQL-style DBMSs…

ACID is an extremely important & successful paradigm, but still debated!

Goal for this lecture: Ensuring Atomicity & Durability

ACID

- **Atomicity:**
  - TXNs should either happen completely or not at all
  - If abort / crash during TXN, *no* effects should be seen



- **Durability:**
  - If DBMS stops running, changes due to completed TXNs should all persist
  - *Just store on stable disk*

We'll focus on how to accomplish atomicity (via logging)

The Log

- Is a list of modifications

- Log is *duplexed* and *archived* on stable storage.

- Can **force write** entries to disk
  - A page goes to disk.

- All log activities ***handled transparently*** the DBMS.

Basic Idea: (Physical) Logging

- Record UNDO information for every update!
  - Sequential writes to log
  - Minimal info (diff) written to log

- The **log** consists of **an ordered list of actions**
  - Log record contains:
    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!

Why do we need logging for atomicity?

- Couldn't we just write TXN to disk **only** once whole TXN complete?
  - Then, if abort / crash and TXN not complete, it has no effect- atomicity!
  - *With unlimited memory and time, this could work…*

- However, we **need to log partial results of TXNs** because of:
  - Memory constraints (enough space for full TXN??)
  - Time constraints (what if one TXN takes very long?)

We need to write partial results to disk!
…And so we need a **log** to be able to ***undo*** these partial results!