CSC 261/461 – Database Systems
Lecture 17

Fall 2017

Announcement

- Quiz 6
  - Due: Tonight at 11:59 pm
- Project 1 Milepost 3
  - Due: Nov 10
- Project 2 Part 2 (Optional)
  - Due: Nov 15

The IO Model & External Sorting

Today's Lecture

1. Chapter 16 (Disk Storage, File Structure and Hashing)
2. Chapter 17 (Indexing)

Self-Study

- Buffering of Blocks
  - Pin Count and Dirty Bit
  - Buffer Replacement Strategies
    - Least recently used (LRU)
    - Clock Policy
    - First in First Out (FIFO)
    - MRU (Most recently used)

RECORDS AND FILES

Records and Record Types

Data in Database

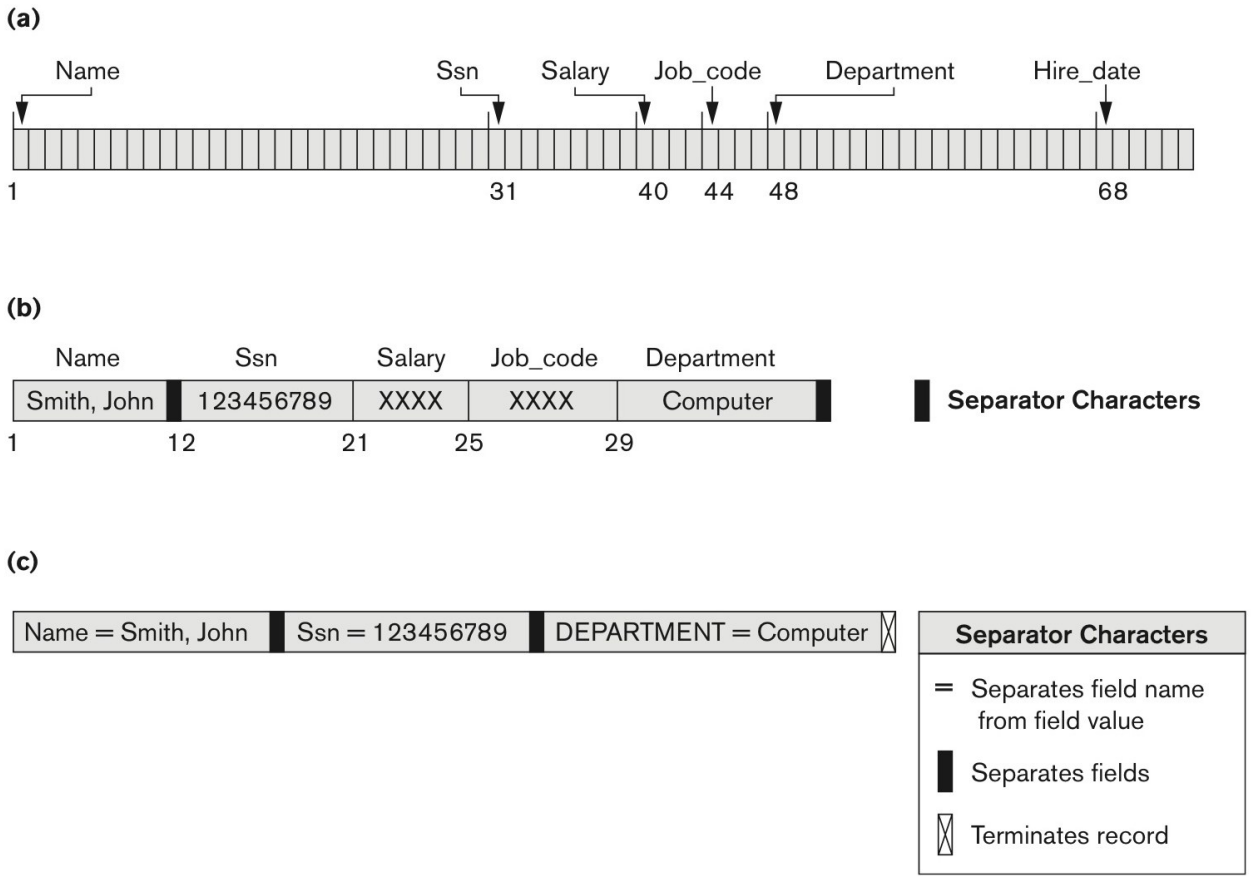
=

A set of **records** organized into a set of **files**

- Each record consists of a collection of related data values or items
- Each value corresponds to a particular attribute
  - Takes one or more bytes

Fixed Length Records vs Variable Length Records

- Fig 16.5 (from textbook)



Record Blocking

- Block Size = B bytes
- Fixed Record Size = R bytes

- With $B > R$:
  - Records per block = $bfr = \lfloor \frac{B}{R} \rfloor$
  - Unused space per block = $B - bfr * R = B \bmod R$

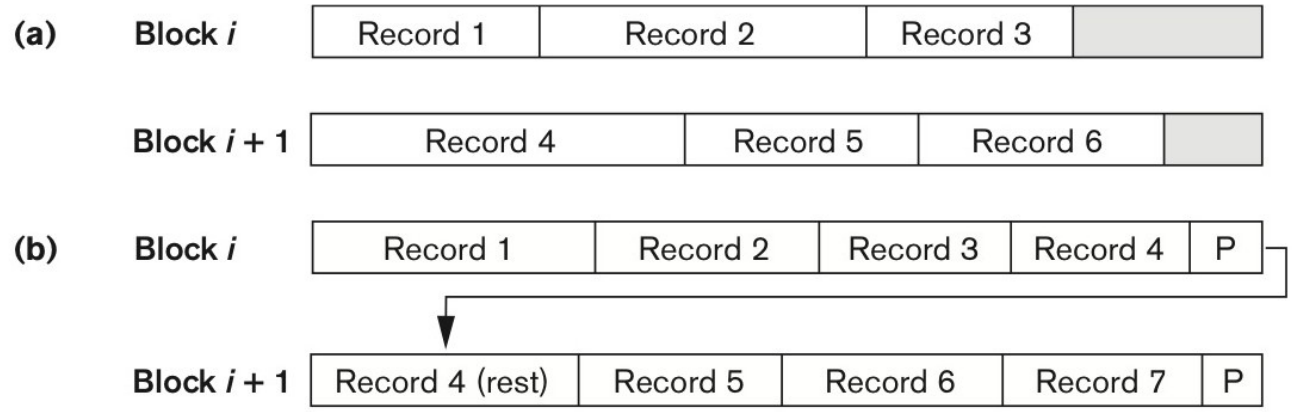
- Number of Blocks required = $\lceil (R/bfr) \rceil$

- Two scenarios:
  - Spanned: records can span more than one block
  - Unspanned: records can't span more than one block

Unspanned vs Spanned

Unspanned

Spanned



Heap Files vs Sorted Files

- Insertion
  - (Heap vs Sorted)
  - For Sorted files: Overflow
- Deletion
  - Deletion Marker
- Modifying
  - For Sorted files: May consist of Deletion and Insertion
- Searching
  - How many block access?
  - (by record number) What if the records are numbered and are of fixed size?
  - Searching by range (Heap vs Sorted)

Another variety

- Another type of files beyond:
  - Heap Files and Sorted Files
  - **Hash Files**

2. HASHING TECHNIQUES

- Sears, Roebuck and Co., Consumers Guide, 1897

"IF YOU DON'T FIND IT IN THE INDEX, LOOK VERY CAREFULLY THROUGH THE ENTIRE CATALOG"

- Hashing first proposed by Arnold Dumey (1956)
- Hash codes
- Chaining
- Open addressing

HASING – GENERAL IDEAS

Example

Java HashCode

hashCode

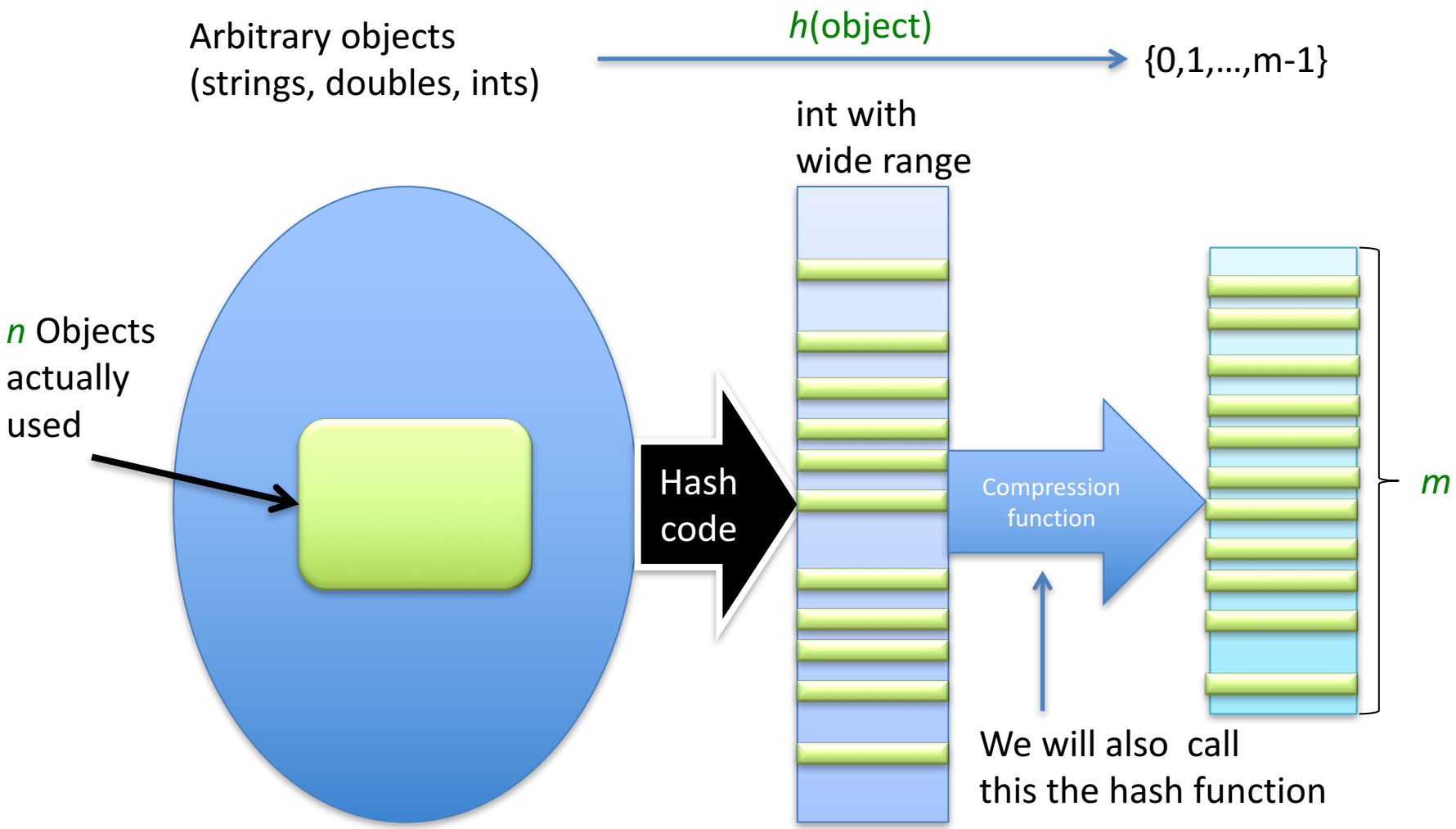
```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The hash code for a String object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

Top level view



Example (Once again!)

Java HashCode

hashCode

```
public int hashCode()
```

Returns a hash code value for the object. This method is supported for the benefit of hash tables such as those provided by `HashMap`.

The hash code for a String object is computed as

```
s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]
```

True / False

- Unequal objects must have different hash codes
- Objects with the same hash code must be equal
- Both Wrong
  - We would like this but it's impossible to achieve
  - Still possible for a particular set of values but impossible if input values are unknown prior to applying the hashcode.

Good Hash Function

- If $\text{key}_1 \neq \text{key}_2$, then it's extremely unlikely that $h(\text{key}_1) = h(\text{key}_2)$
  - Collision problem!

- Pigeonhole principle
  - K+1 pigeons, K holes → at least one hole with ≥ 2 pigeons

Division method

$$h(s) = s \mod m$$

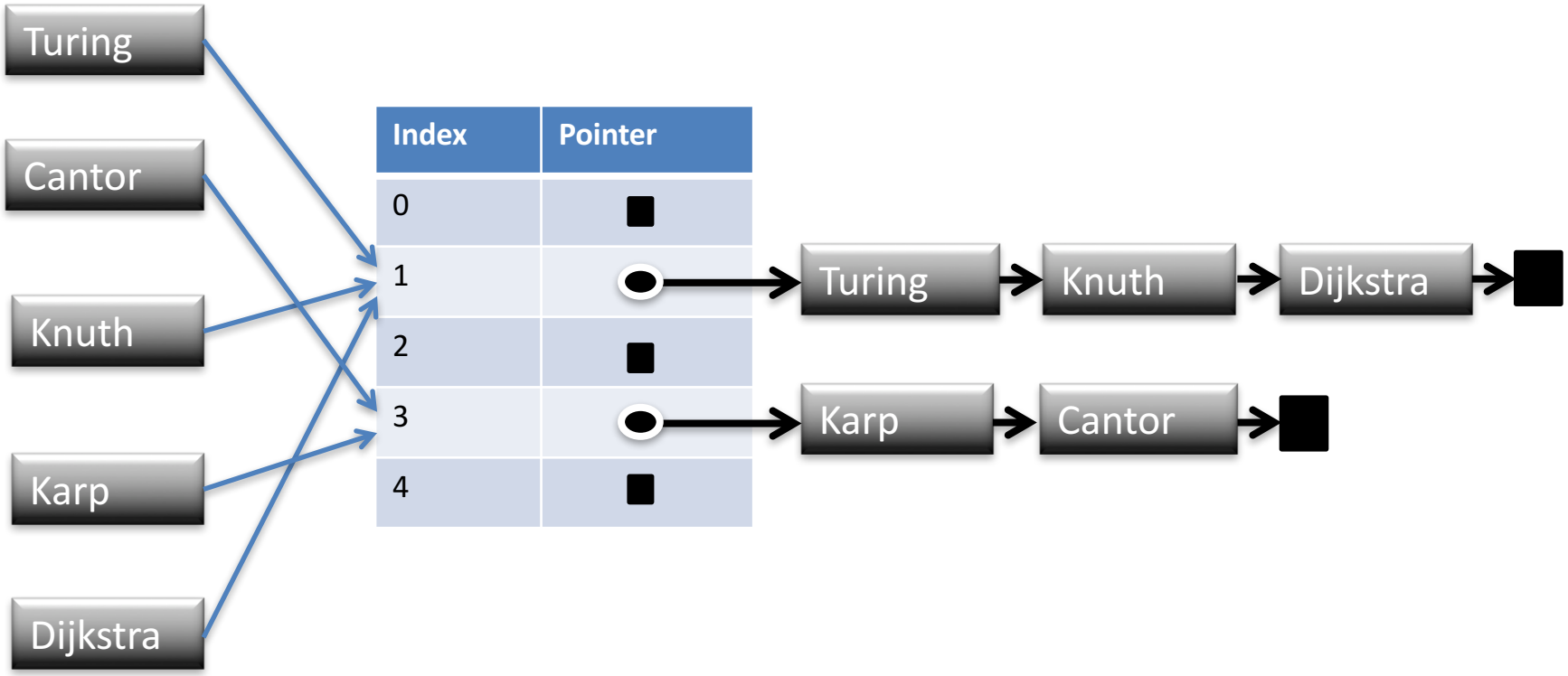
- How does this function perform for different m?

Separate chaining

Open addressing

COLLISION RESOLUTION

Separate Chaining



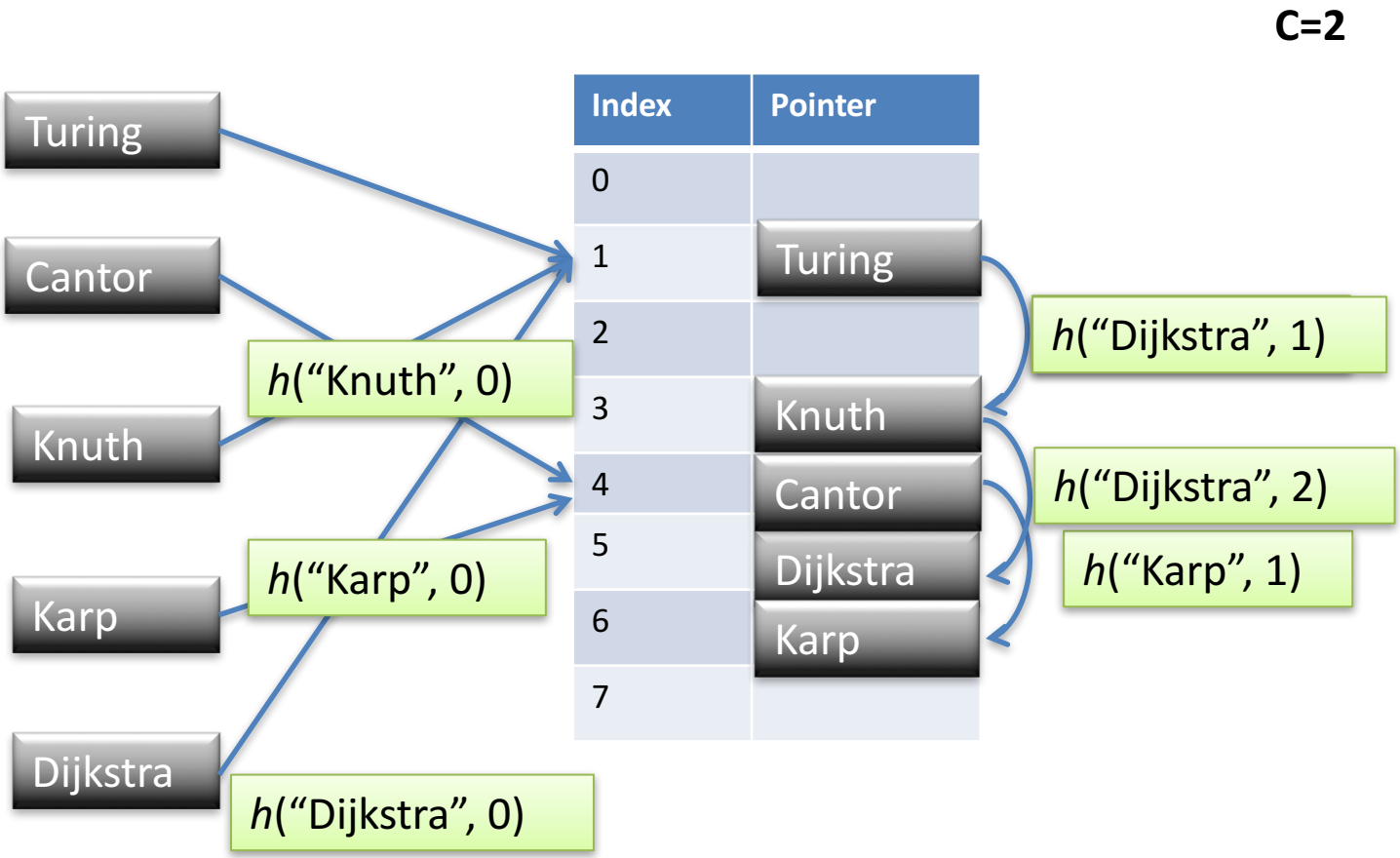
Open Addressing

- Store all entries in the hash table itself, no pointer to the "outside"
  - If a particular index **i** is full, try the next index (i+C) where C is a constant.

- Advantage
  - Less space waste
  - Perhaps good cache usage

- Disadvantage
  - More complex collision resolution
  - Slower operations

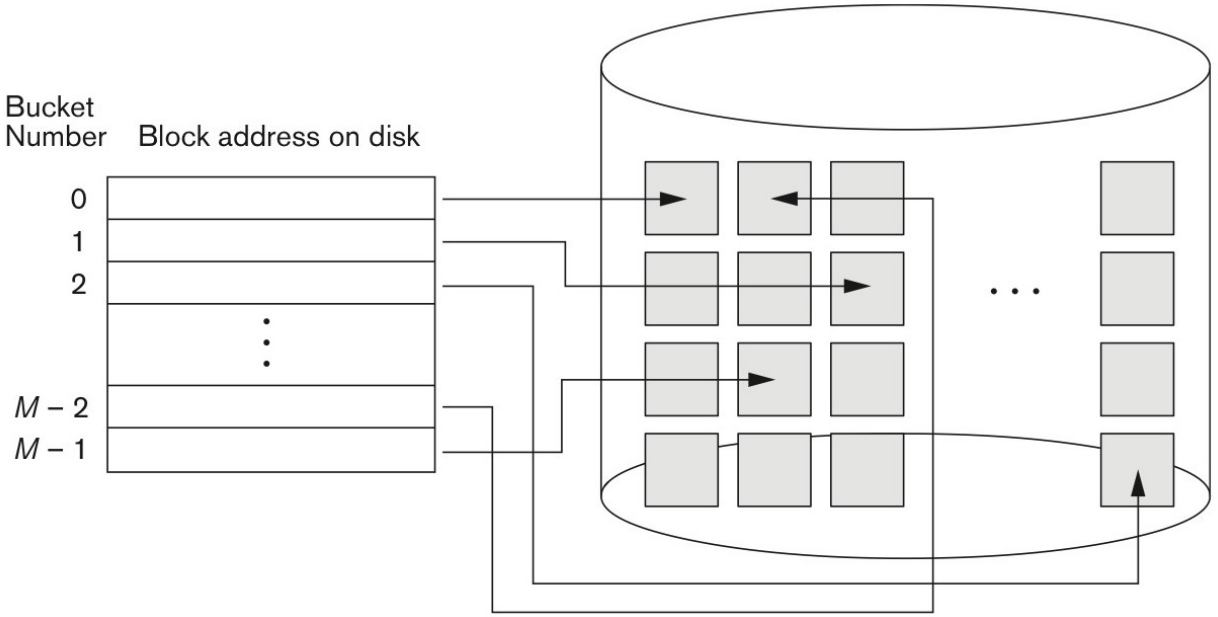
Open Addressing



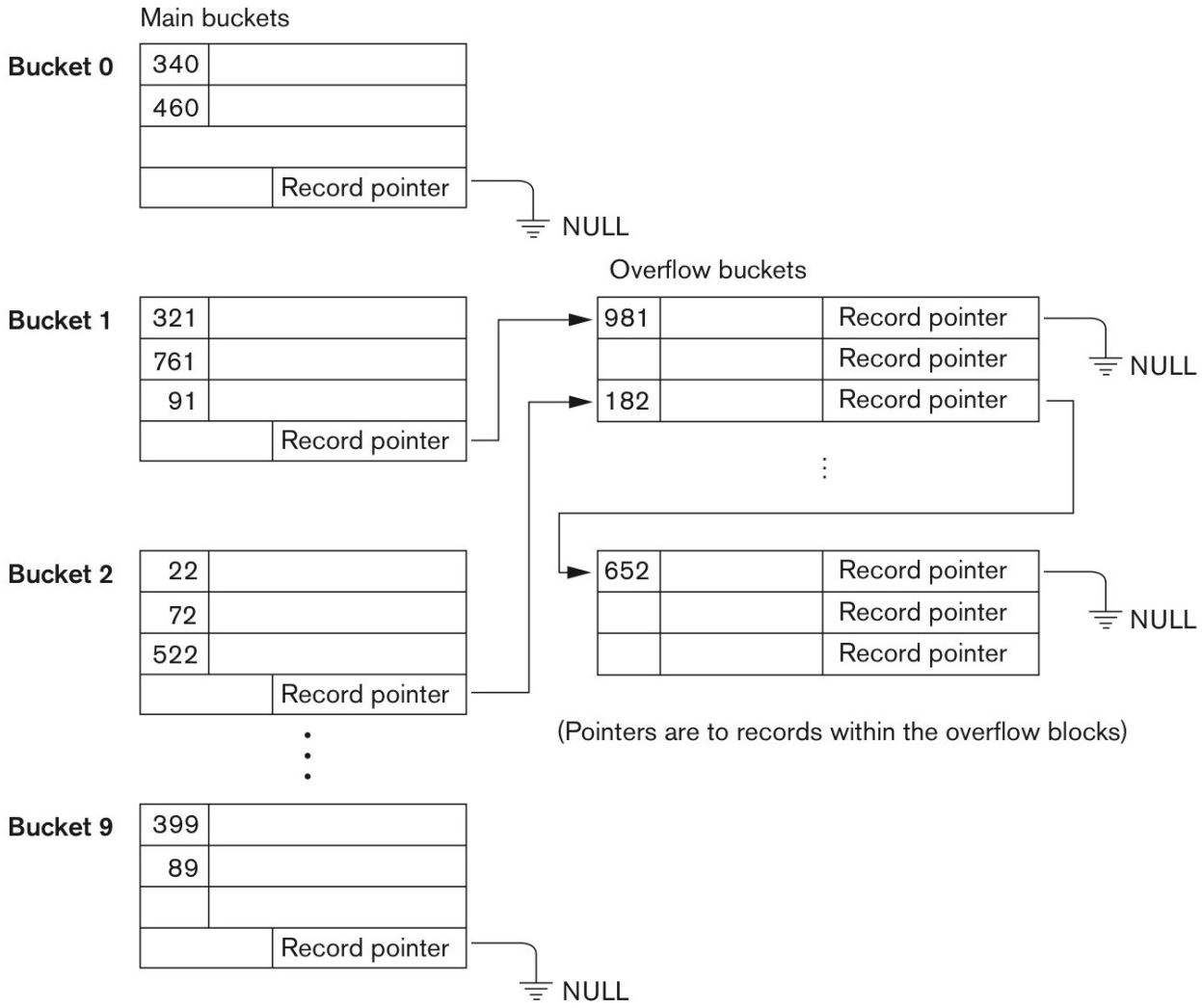
External Hashing

- Hashing for disk files
- Target address space is made of buckets
- Hashing function maps a key into relative bucket number
- Convert the bucket number into corresponding disk block address

Bucket Number to Disk Block address



Bucket Number to Disk Block address



REVIEW

What Did We Learn

- Disk Storage
  - Hardware Description of Disk Devices
    - Sections to study: 16.2.1 and 16.2.2
- Buffering of Blocks
  - Buffer Management
  - Buffer Replacement Strategies
  - Sections to study: 16.3
- Placing File Records on Disk
  - Records and Record Types
  - Fixed-Length, Variable-Length, Spanned, Unspanned
  - Sections to study: 16.4.1 to 16.4.4

What did we learn

- Operations on Files
  - Insert, Modify, and Delete
  - And others.
  - Sections to study: 16.5
- Heap Files vs Sorted Files
  - Sections to study: 16.6 and 16.7
- Hash Files
  - Internal Hashing and External Hashing
  - Sections to study: 16.8.1 and 16.8.2

INDEXING

Types of Indexing

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
- Multilevel Indexes
  - Dynamic Multilevel Indexes
- Hash Indexes

What you will learn about in this section

1. Indexes: Motivation

2. Indexes: Basics

Index Motivation

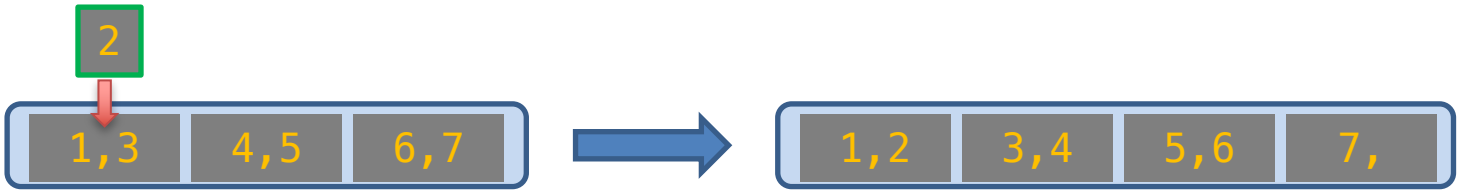
```
Person(name, age)
```

- 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!
- How many IO operations to search over N ***sorted*** records?
  - Simple scan: $O(N)$
  - Binary search: $O(\log_2 N)$

Could we get even cheaper search? E.g. go from $\log_2 N \rightarrow \log_{200} N$?

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, requiring up to ~ **2*N/P** IO operations (where P = # of records per page)!
  - 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!
  - Sometimes use B+ Trees, sometimes hash indexes

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 rows of the database table
- An index can store:
  - Full rows it points to (primary index) or
  - Pointers to those rows (secondary index)

Operations on an Index

- Search: Quickly find all records which meet some condition *on the search key attributes*
  - More sophisticated variants as well. Why?

Indexing is one the most important features provided by a database for performance

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***

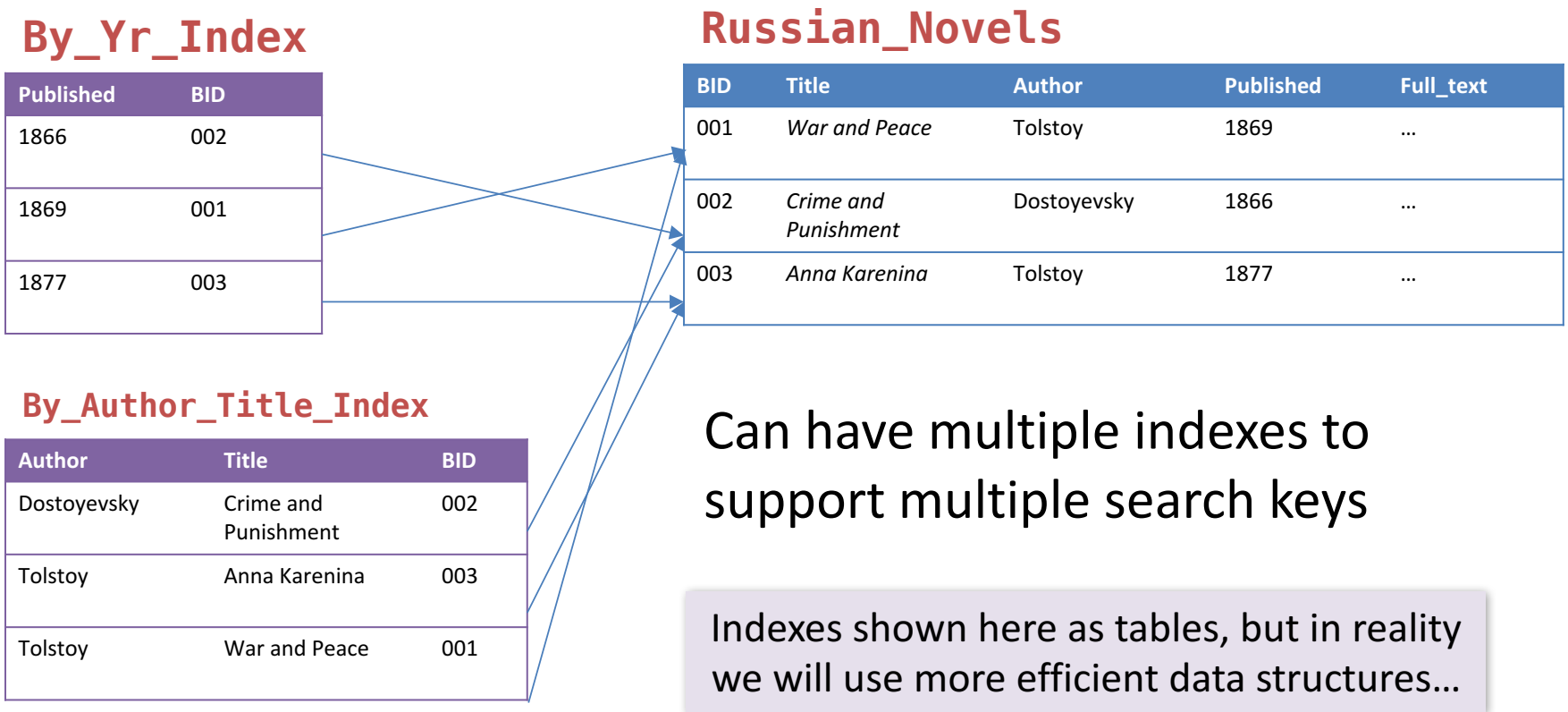
Russian_Novels

BID	Title	Author	Published	Full_text
001	*War and Peace*	Tolstoy	1869	…
002	*Crime and Punishment*	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



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 **covering** *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:

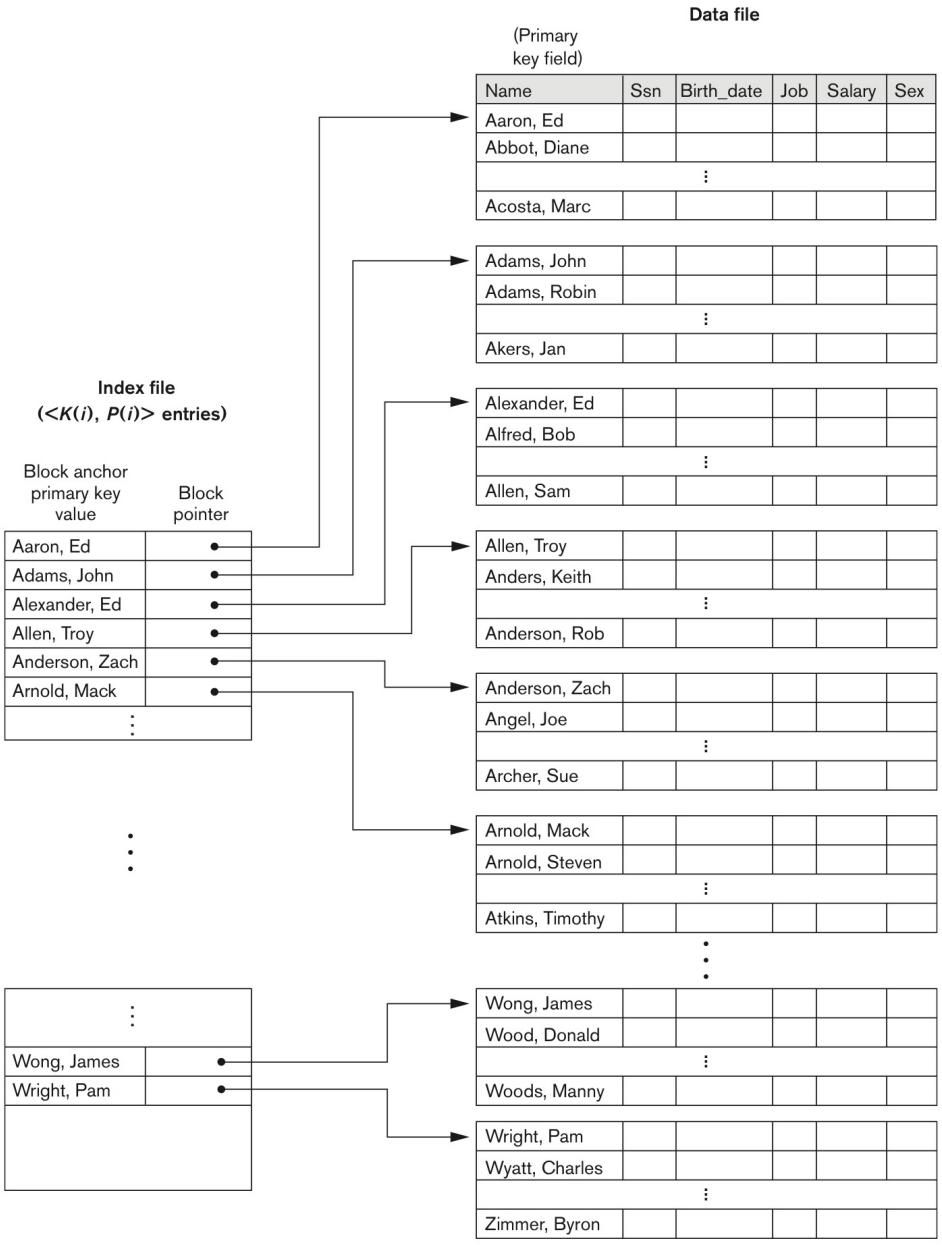
```
SELECT Published, BID
FROM Russian_Novels
WHERE Published > 1867
```

TYPES OF INDEXES

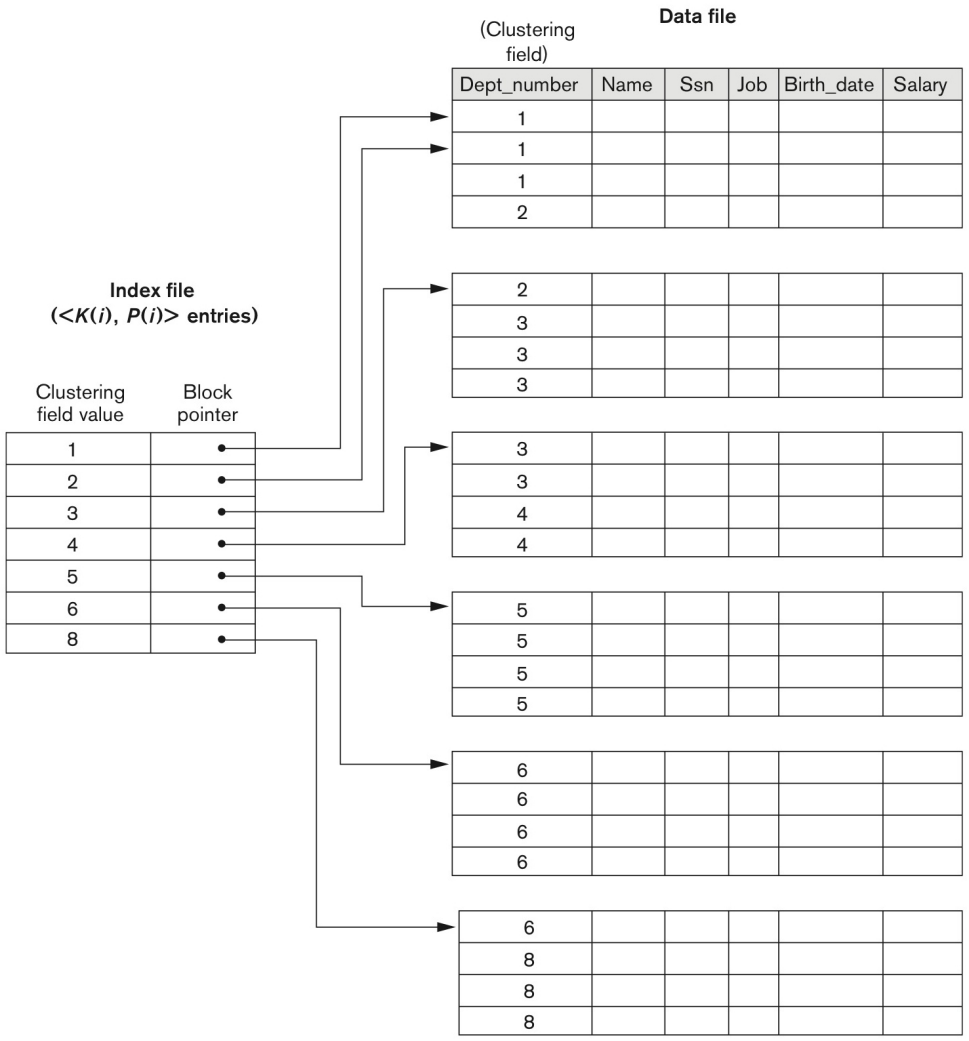
Types of Indexing

- Primary Indexes
- Clustering Indexes
- Secondary Indexes
- Multilevel Indexes
  - Dynamic Multilevel Indexes
- Hash Indexes

Primary Indexes: Index for Sorted (Ordered) Files

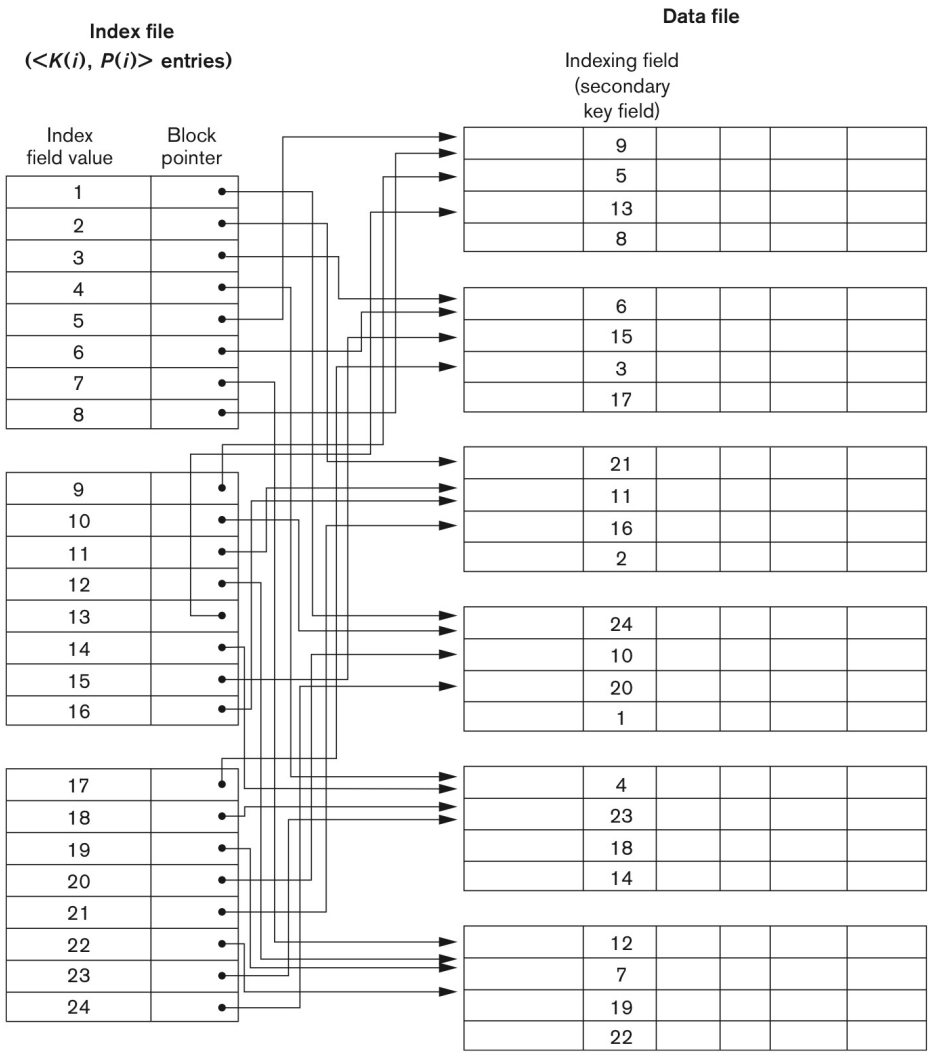


Clustering Indexes (Index for Sorted (on non-key) Files)



Secondary Indexes (on a key field)

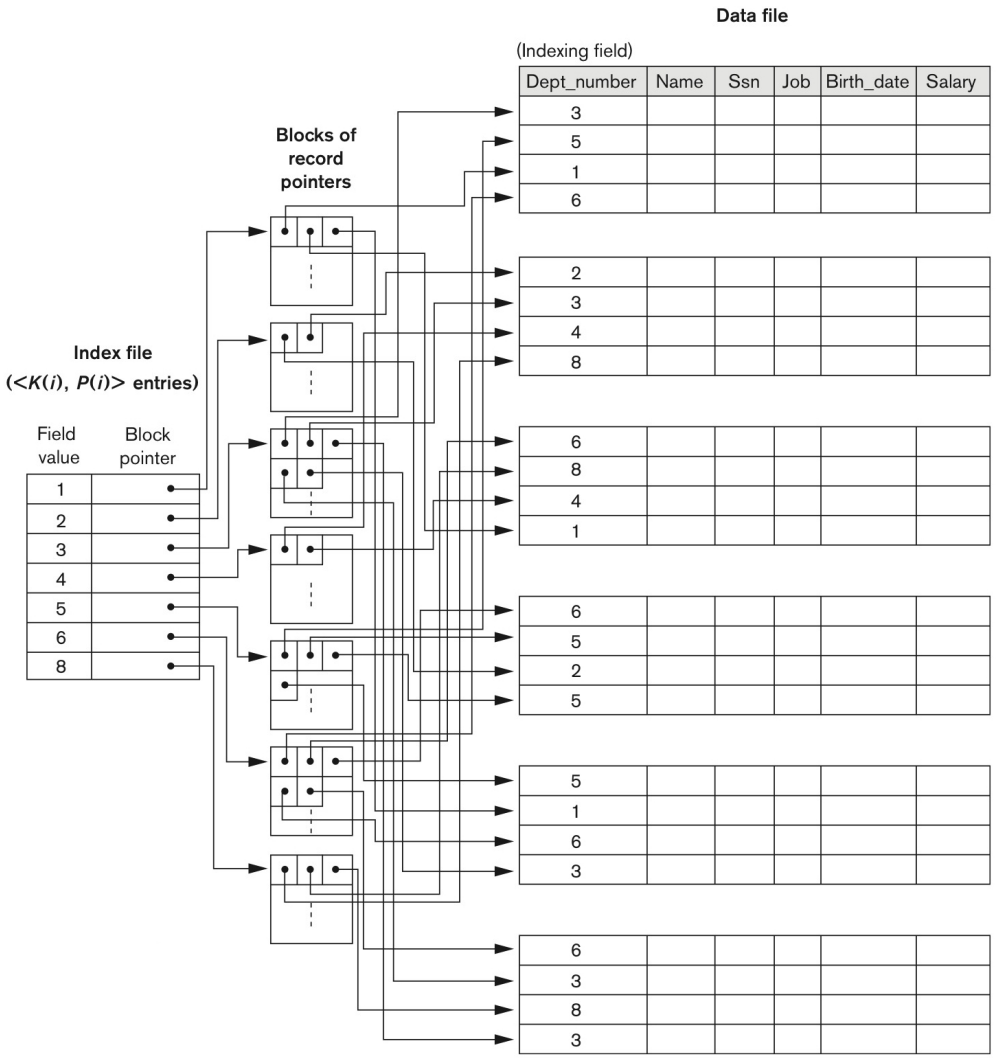
- Secondary means of accessing a data file
- File records could be ordered, unordered, or hashed



Secondary Indexes (on a non-key field)

Extra level of indirection

- Provides logical ordering
  - Though records are not physically ordered



High-level Categories of Index Types

- Multilevel Indexes
  - Very good for range queries, sorted data
  - Some old databases only implemented B-Trees
  - *We will mostly look at a variant called* ***B+ Trees***

- Hash Tables
  - Very good for searching

Real difference between structures: costs of ops *determines which index you pick and why*

MULTILEVEL INDEXES

What you will learn about in this section

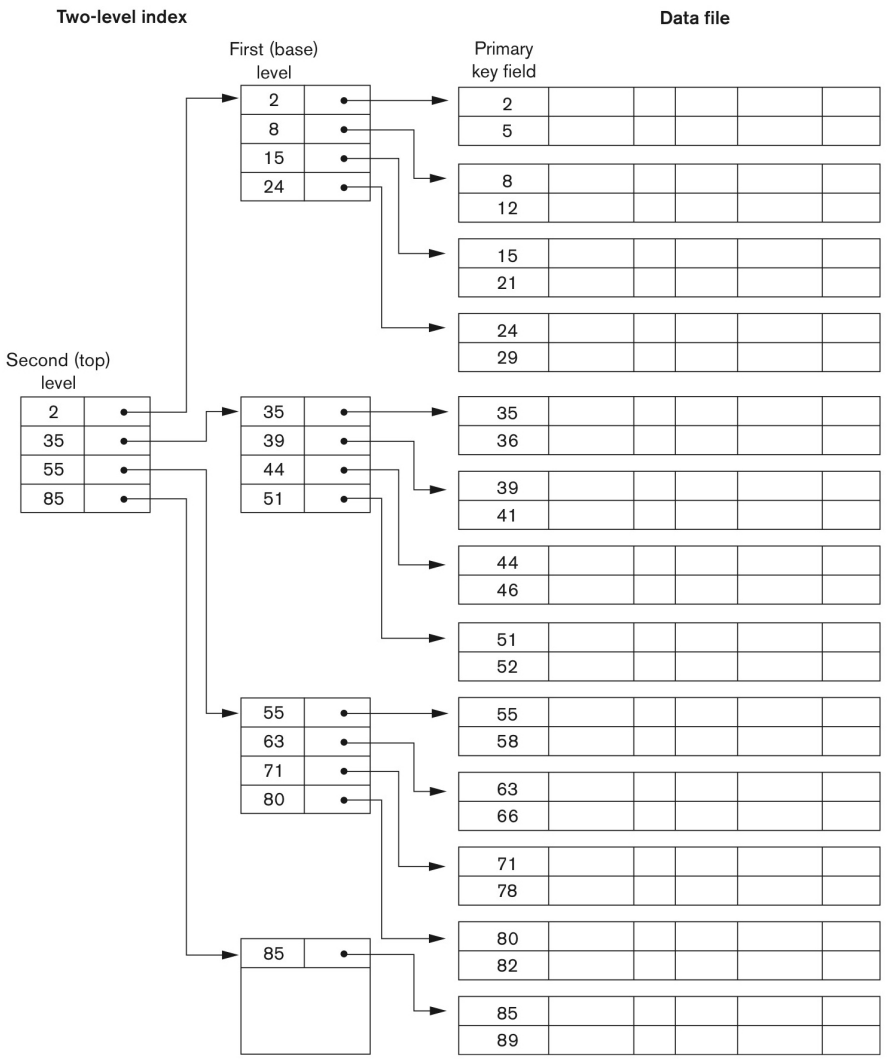
1. ISAM

2. B+ Tree

1. ISAM

ISAM

- Indexed Sequential Access Method
  - For an index with b_i blocks
    - Earlier: $\log_2 b_i$ block access
    - Now: $\log_{fo} b_i$ block access
    - $(fo = fanout)$



1. B+ TREES

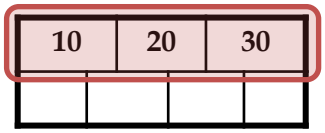
What you will learn about in this section

1. B+ Trees: Basics
2. B+ Trees: Design & Cost
3. Clustered Indexes

B+ Trees

- Search trees
  - B does not mean binary!
- Idea in B Trees:
  - make 1 node = 1 physical page
  - Balanced, height adjusted tree (not the B either)
- Idea in B+ Trees:
  - Make leaves into a linked list (for range queries)

B+ Tree Basics

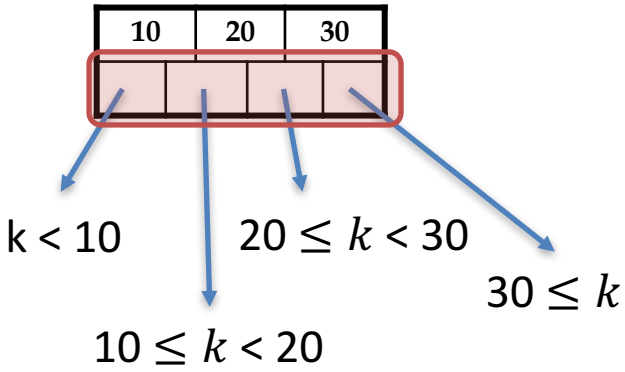


Parameter d = degree
The minimum number of key an interior node can have

Each *non-leaf ("interior")* ***node*** has $\geq$ d and $\leq$ 2d ***keys****

except for root node, which can have between **1 and 2d keys*

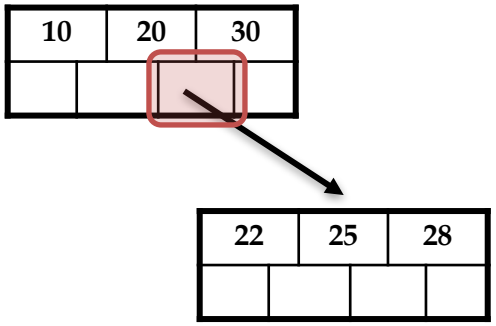
B+ Tree Basics



The *n* keys in a node define *n+1* ranges

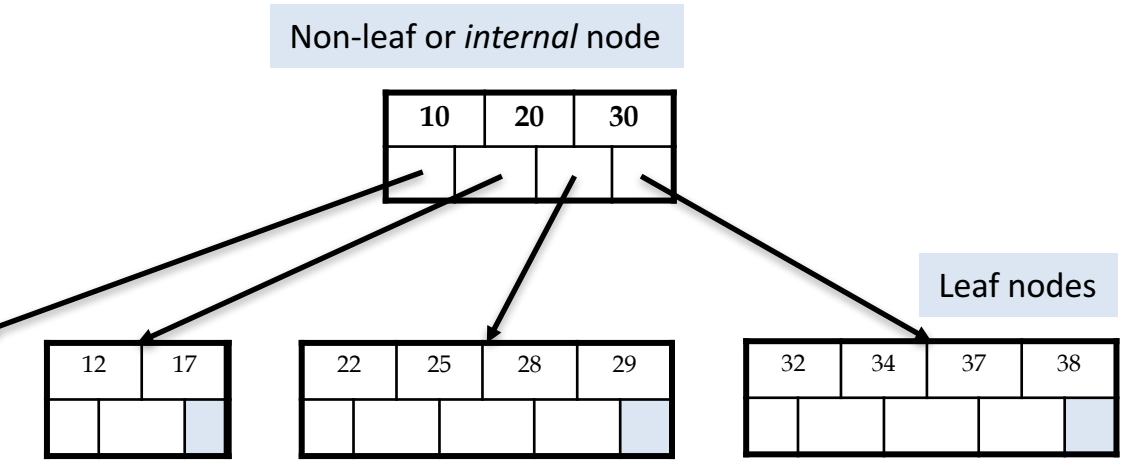
B+ Tree Basics

Non-leaf or *internal* node



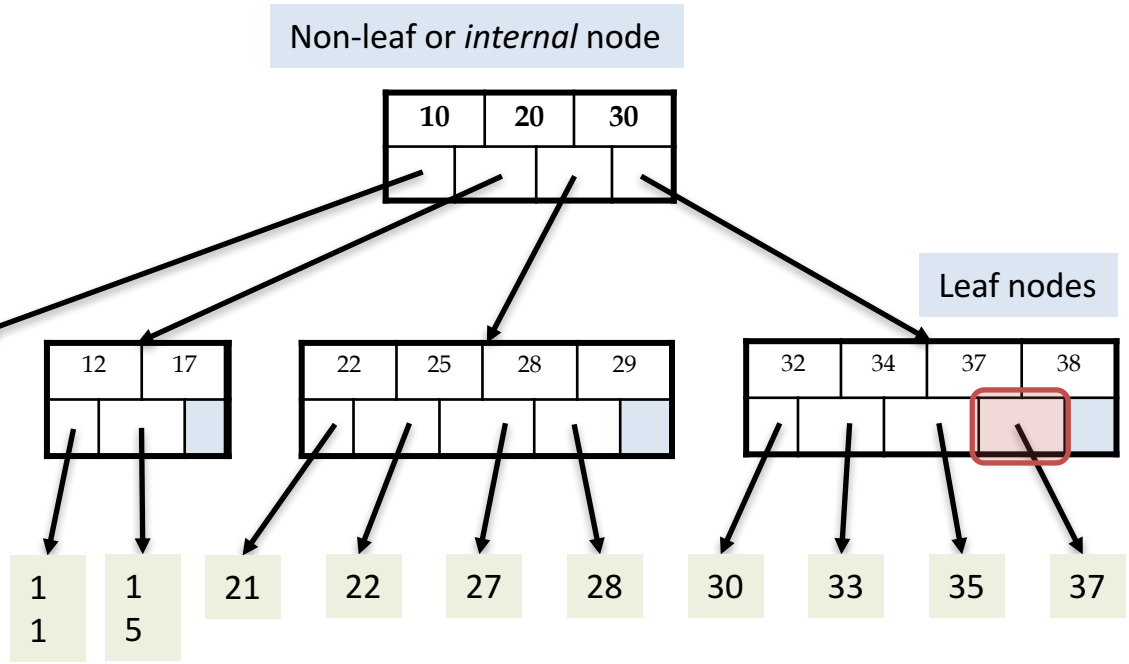
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

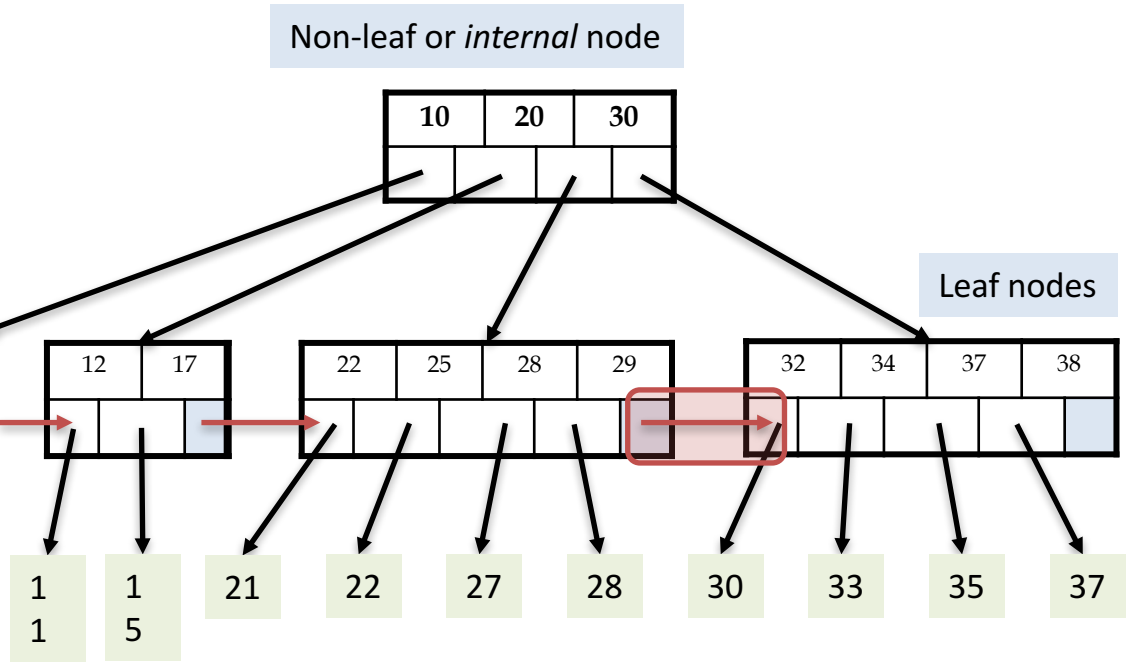
B+ Tree Basics



Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

B+ Tree Basics

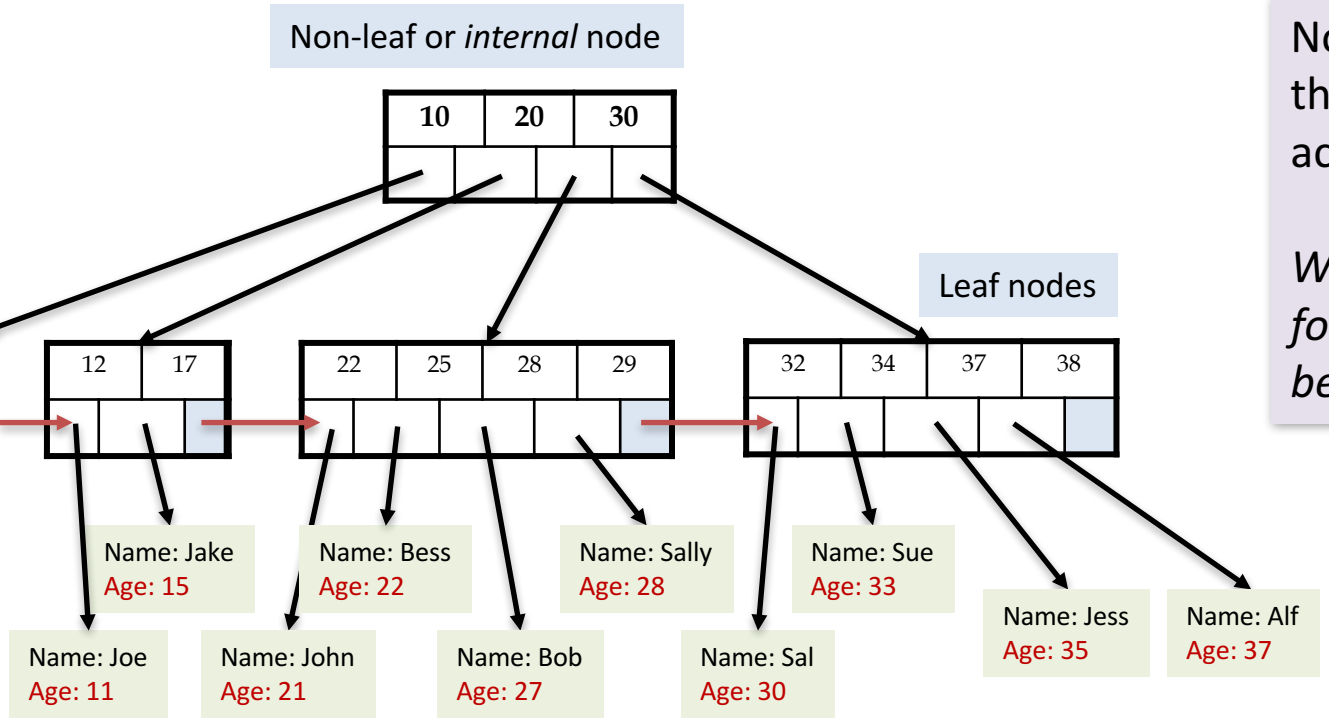


Leaf nodes also have between *d* and *2d* keys, and are different in that:

Their key slots contain pointers to data records

They contain a pointer to the next leaf node as well, ***for faster sequential traversal***

B+ Tree Basics



Note that the pointers at the leaf level will be to the actual data records (rows).

We might truncate these for simpler display (as before)…

Some finer points of B+ Trees

Searching a B+ Tree

- For exact key values:
  - Start at the root
  - Proceed down, to the leaf
- For range queries:
  - As above
  - *Then sequential traversal*

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

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

B+ Tree Exact Search Animation

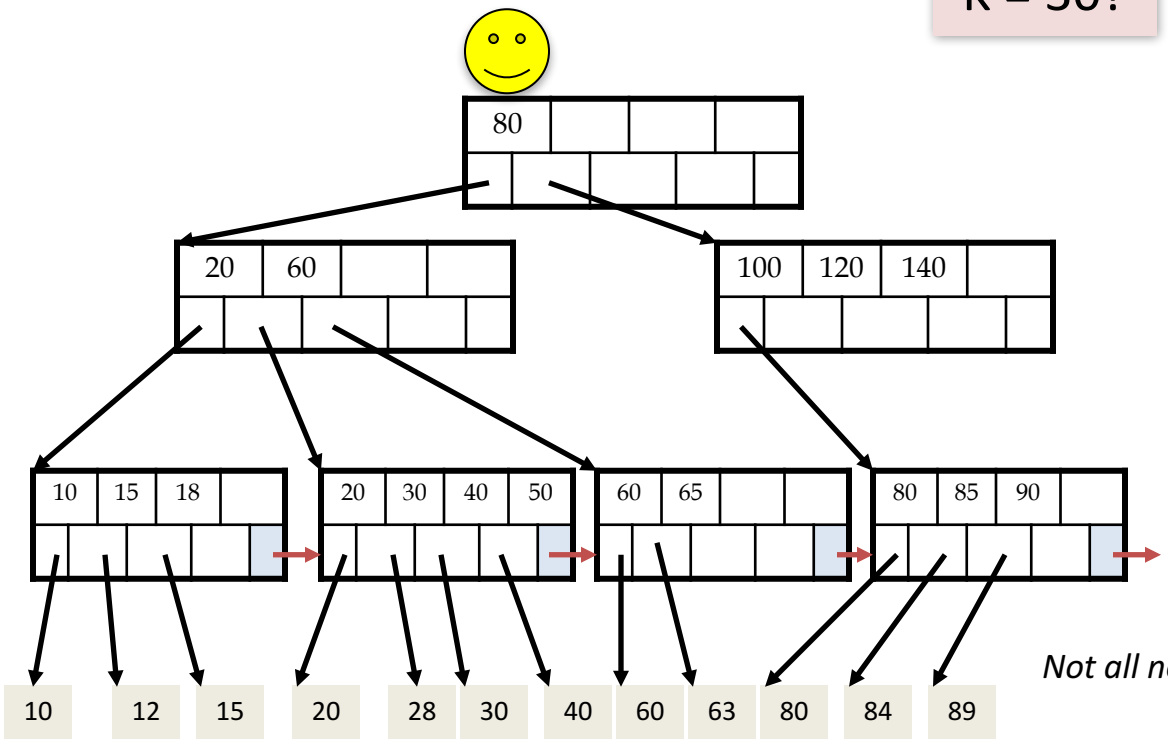
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

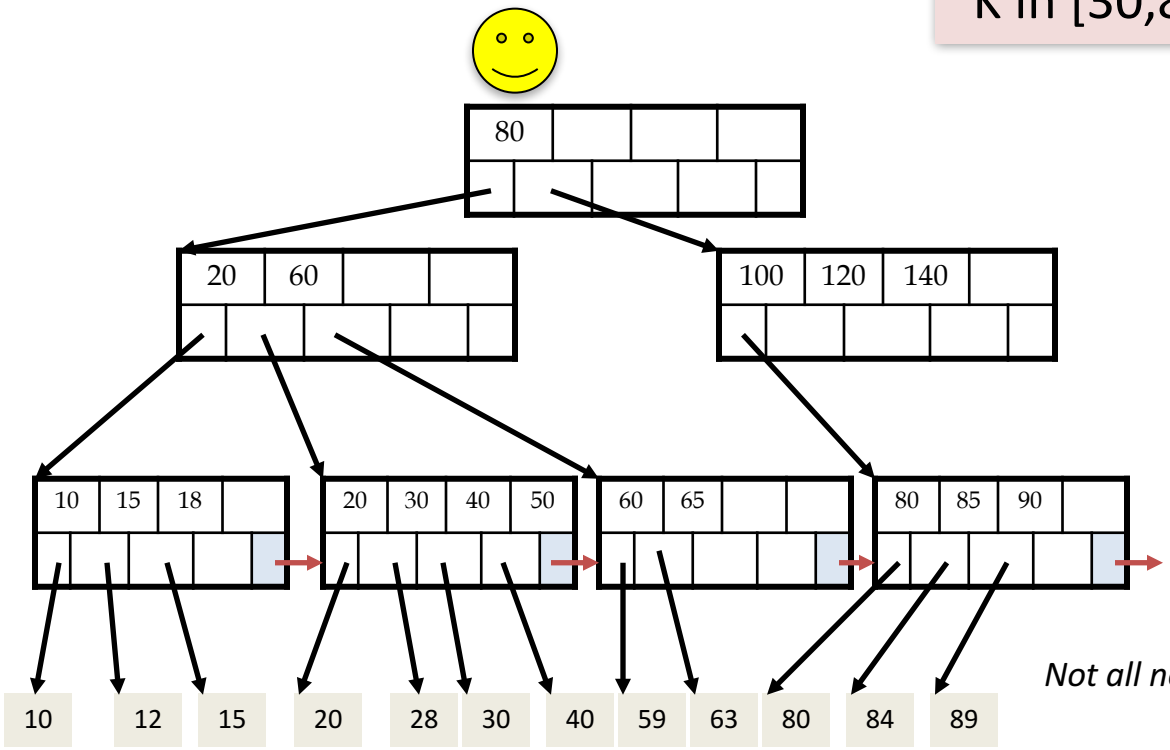
K in [30,85]?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Design

- How large is ***d***?
- Example:
  - Key size = 4 bytes
  - Pointer size = 8 bytes
  - Block size = 4096 bytes
- We want each *node* to fit on a single *block/page*
  - 2d x 4 + (2d+1) x 8 <= 4096 → ***d*** <= ***170***

B+ Tree: High Fanout = Smaller & Lower IO

- As compared to e.g. binary search trees, B+ Trees have **high *fanout*** (***between d+1 and 2d+1***)
- This means that the **depth of the tree is small** → getting to any element requires very few IO operations!
  - Also can often store most or all of the B+ Tree in main memory!

The **fanout** is defined as the number of pointers to child nodes coming out of a node

Note that fanout is dynamic- we'll often assume it's constant just to come up with approximate eqns!

Simple Cost Model for Search

- Let:
  - f = fanout, which is **in** $[d+1, 2d+1]$ *(**we'll assume it's constant for our cost model…**)*
  - N = the total number of *pages* we need to index
  - F = fill-factor (usually ~= 2/3)

- Our B+ Tree needs to have room to index N / F pages!
  - We have the fill factor in order to leave some open slots for faster insertions

- What height (h) does our B+ Tree need to be?
  - h=1 → Just the root node- room to index f pages
  - h=2 → f leaf nodes- room to index f^2 pages
  - h=3 → f^2 leaf nodes- room to index f^3 pages
  - …
  - h → f^{h-1} leaf nodes- room to index f^h pages!

→ We need a B+ Tree of height h = $\left\lceil \log_f \frac{N}{F} \right\rceil$

Fast Insertions & Self-Balancing

- **Same cost as exact search**
- ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

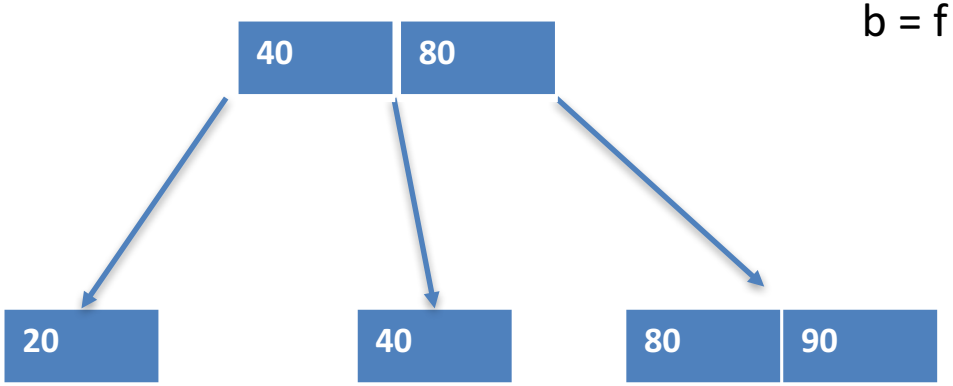
B+ Trees also (relatively) fast for single insertions!
However, can become bottleneck if many insertions (if fill-factor slack is used up…)

Insertion

- Perform a search to determine what bucket the new record should go into.
- If the bucket is not full (at most (b-1) entries after the insertion), add the record.
- Otherwise, split the bucket.
  - Allocate new leaf and move half the bucket's elements to the new bucket.
  - Insert the new leaf's smallest key and address into the parent.
  - If the parent is full, split it too.
    - Add the middle key to the parent node.
  - Repeat until a parent is found that need not split.
- If the root splits, create a new root which has one key and two pointers. (That is, the value that gets pushed to the new root gets removed from the original node)

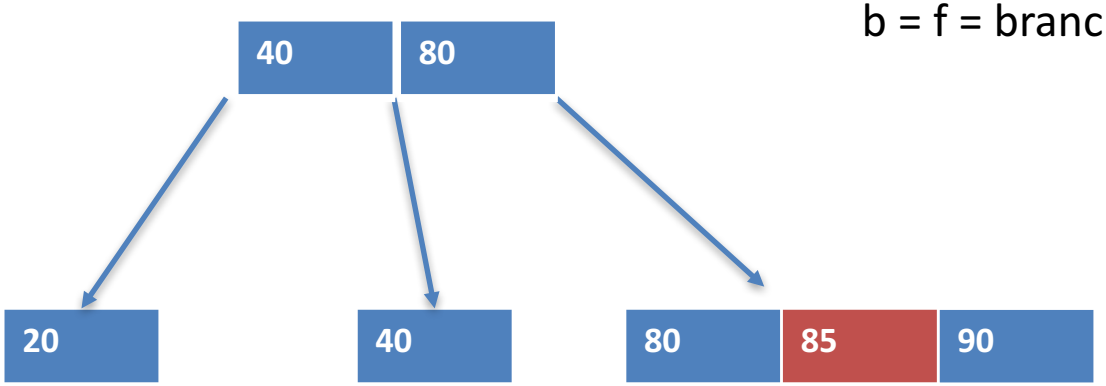
- Note: B-trees grow at the root and not at the leave

Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

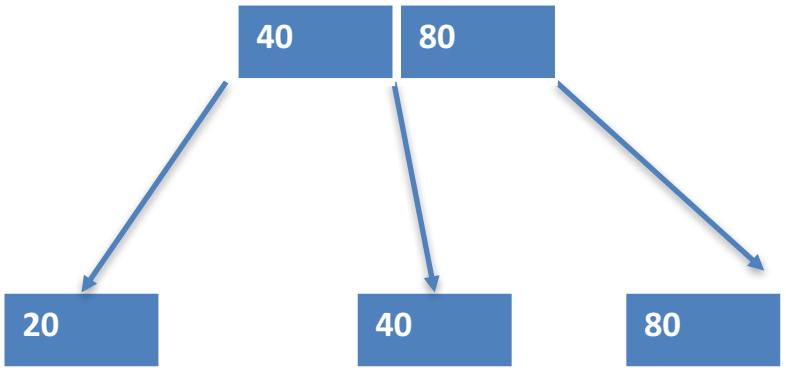
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

This is what we would like. But the maximum number of keys in any node is (3-1) = 2
So, split.

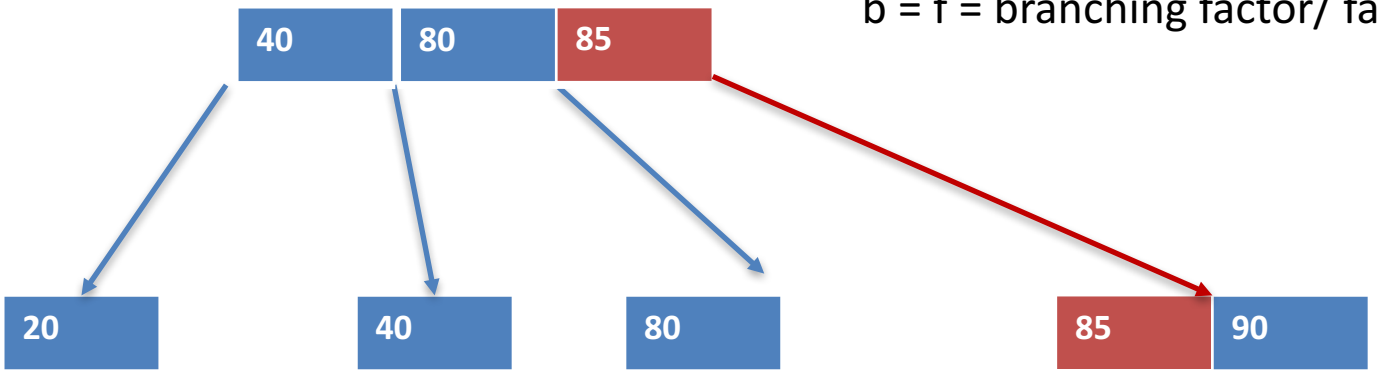
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

Allocate new leaf and move half the bucket's elements to the new bucket.

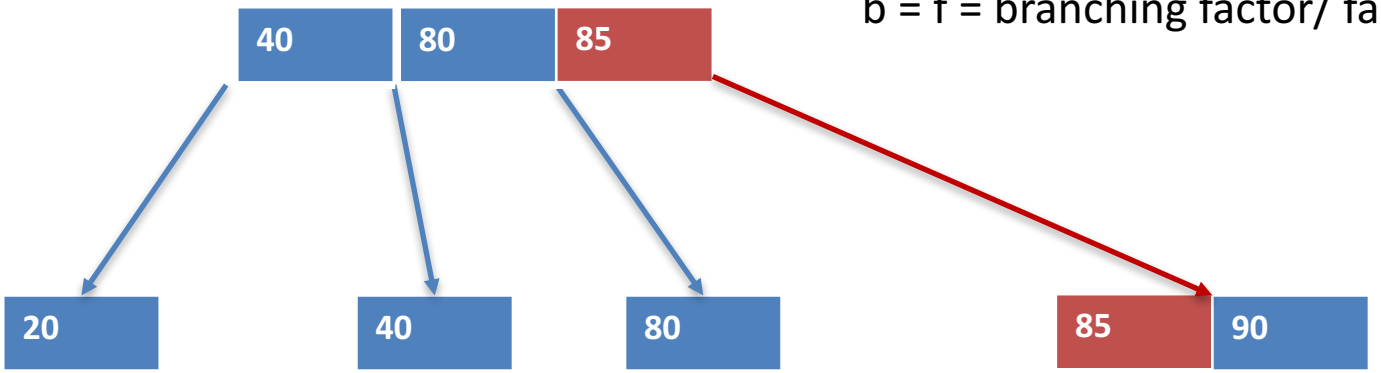
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

Insert the new leaf's smallest key and address into the parent.

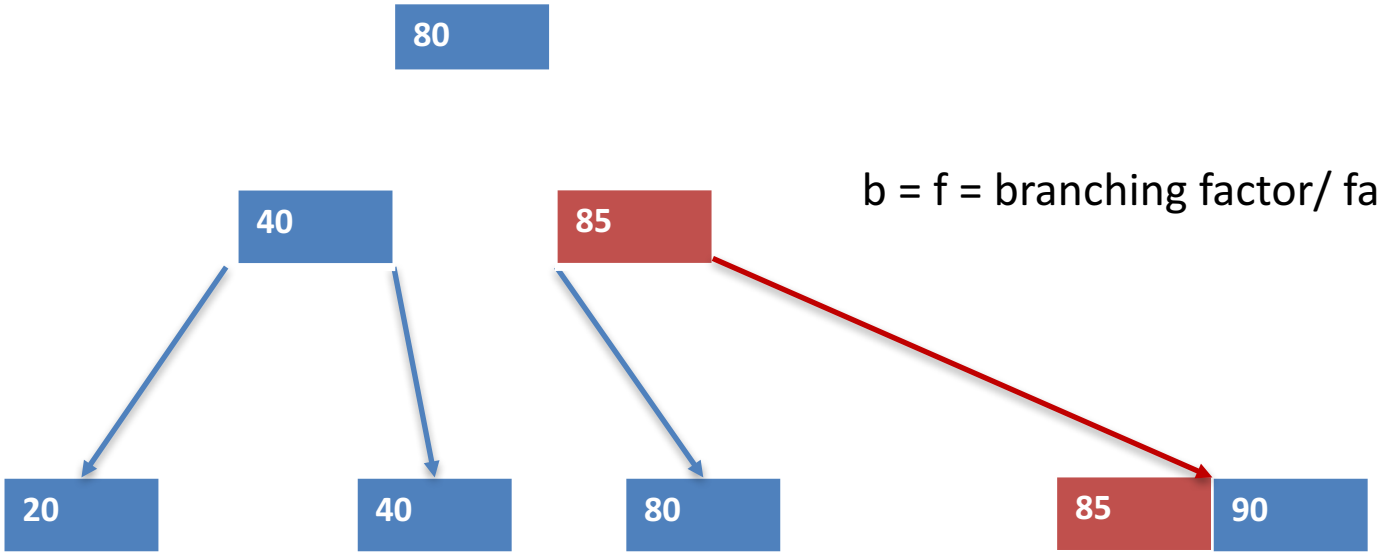
Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

This is not allowed, as the parent is full. Need to split

Insertion (Insert 85)



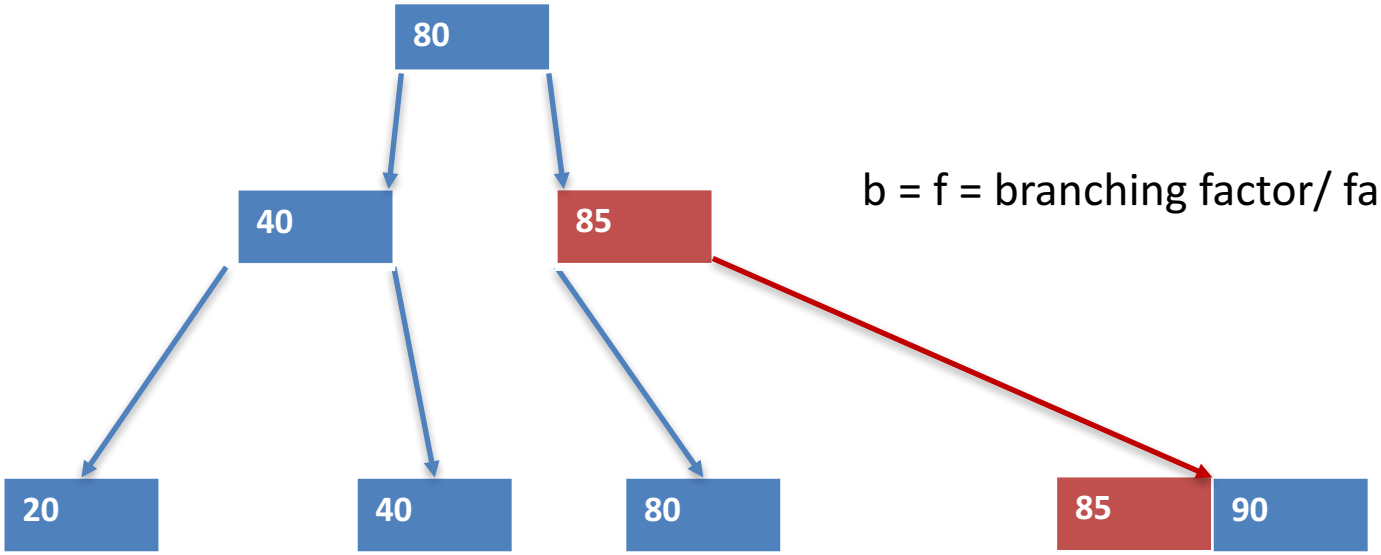
b = f = branching factor/ fan-out = 3

If the parent is full, split it too.

Add the middle key to the parent node.

Repeat until a parent is found that needs no spliting

Insertion (Insert 85)



b = f = branching factor/ fan-out = 3

If the root splits, create a new root which has one key and two pointers.

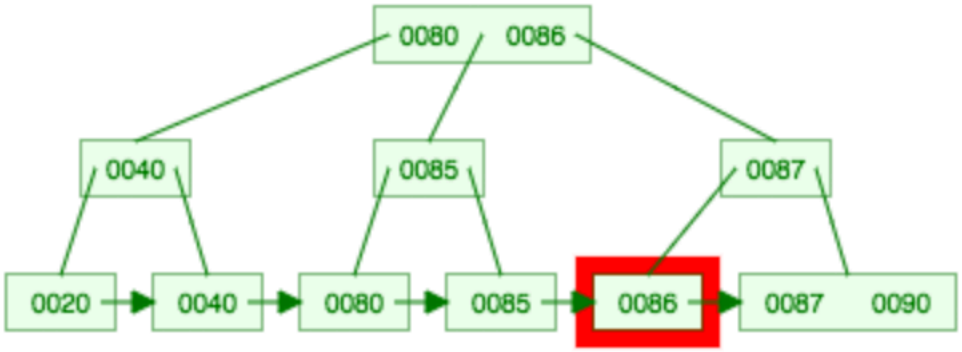
(That is, the value that gets pushed to the new root gets removed from the original node)

Deletion

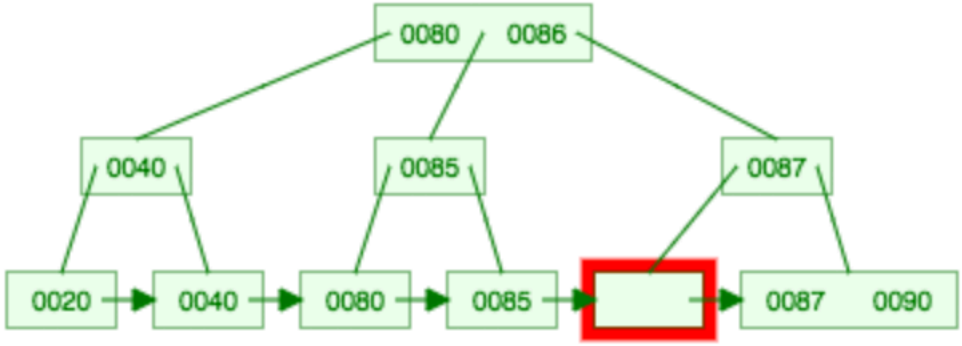
- Start at root, find leaf L where entry belongs.
- Remove the entry.
  - If L is at least half-full, done!
  - If L has fewer entries than it should,
    - If sibling (adjacent node with same parent as L) is more than half-full, re-distribute, borrowing an entry from it.
    - Otherwise, sibling is exactly half-full, so we can merge L and sibling.
- If merge occurred, must delete entry (pointing to L or sibling) from parent of L.
- Merge could propagate to root, decreasing height.

- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- The degree in this visualization is actually fan-out f or branching factor b.

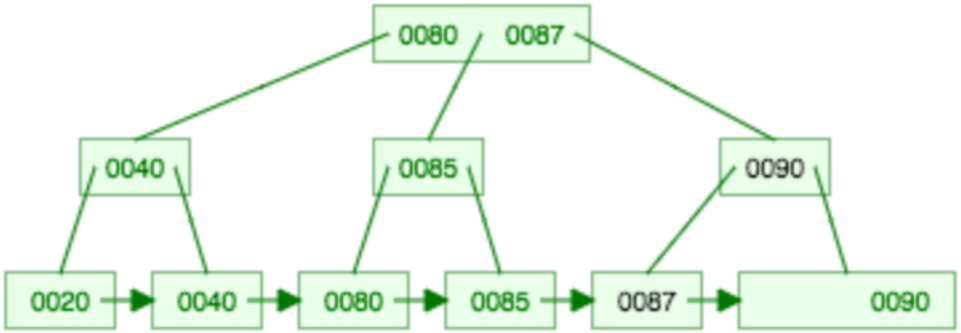
- Find 86



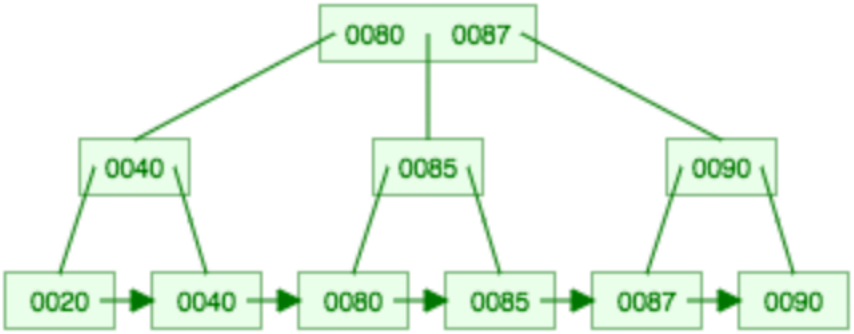
- Delete it



- Stealing from right sibling (redistribute).
- Modify the parent node



- Finally,



Acknowledgement

- Some of the slides in this presentation are taken from the slides provided by the authors.
- Many of these slides are taken from cs145 course offered by Stanford University.