B+ Trees: An IO-Aware Index Structure

"If you don't find it in the index, look very carefully through the entire catalog"

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

Today’s Lecture

1. *[Moved from 12-3]: External Merge Sort & Sorting Optimizations*

2. Indexes: Motivations & Basics

3. B+ Trees

1. External Merge Sort

What you will learn about in this section

1. External merge sort

2. External merge sort on larger files

3. Optimizations for sorting

Recap: External Merge Algorithm

- Suppose we want to merge two **sorted** files both much larger than main memory (i.e. the buffer)

- We can use the **external merge algorithm** to merge files of ***arbitrary length*** in **2*(N+M) IO** operations with only **3 buffer pages**!

Our first example of an "IO aware" algorithm / cost model

External Merge Sort

Why are Sort Algorithms Important?

- Data requested from DB in sorted order is **extremely common**
  - e.g., find students in increasing GPA order

- **Why not just use quicksort in main memory??**
  - What about if we need to sort 1TB of data with 1GB of RAM…

A classic problem in computer science!

More reasons to sort…

- Sorting useful for eliminating *duplicate copies* in a collection of records (Why?)
- Sorting is first step in *bulk loading* B+ tree index.

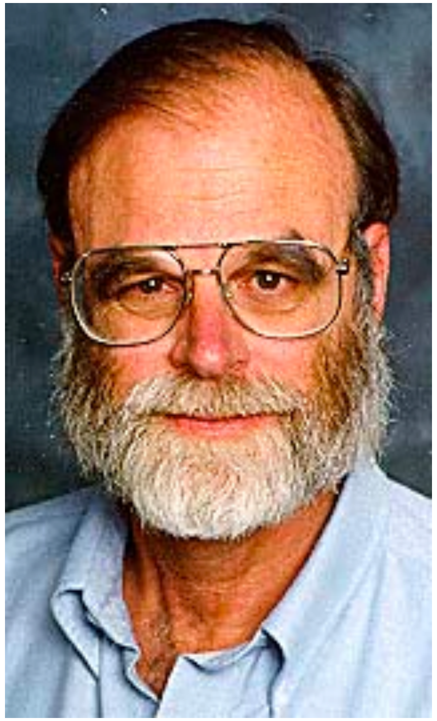
Coming up…

Next lecture

- *Sort-merge* join algorithm involves sorting

Do people care?

http://sortbenchmark.org



Sort benchmark bears his name

So how do we sort big files?

1. Split into chunks small enough to **sort in memory *("runs")***

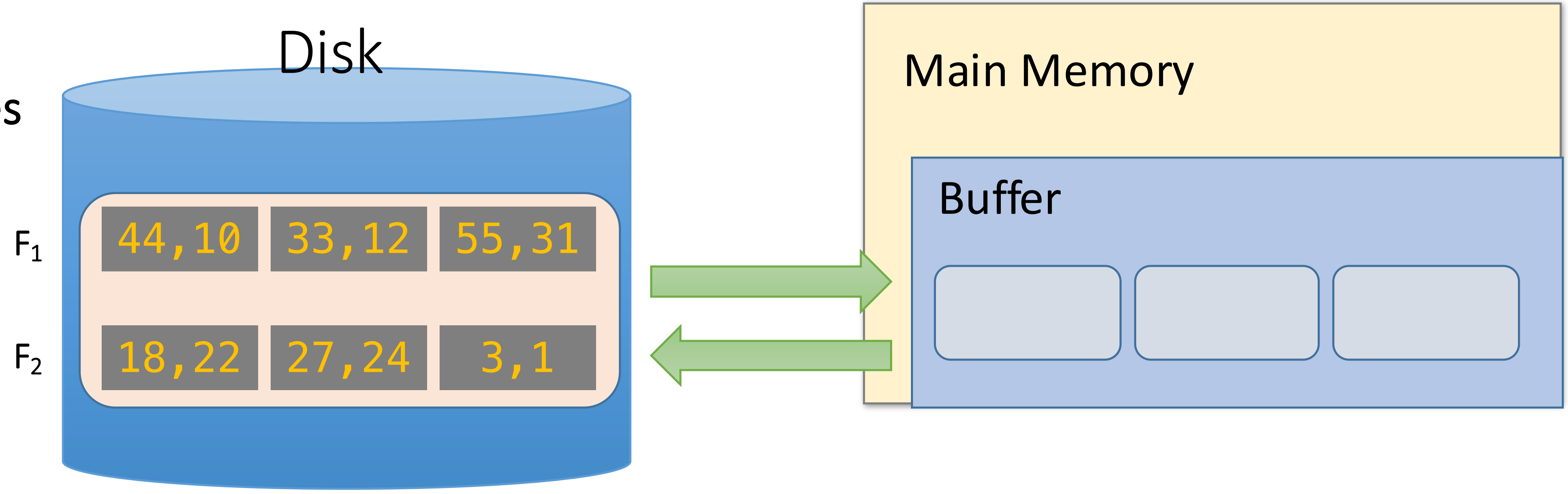
2. **Merge** pairs (or groups) of runs ***using the external merge algorithm***

3. **Keep merging** the resulting runs ***(each time = a "pass")*** until left with one sorted file!

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Orange file = unsorted



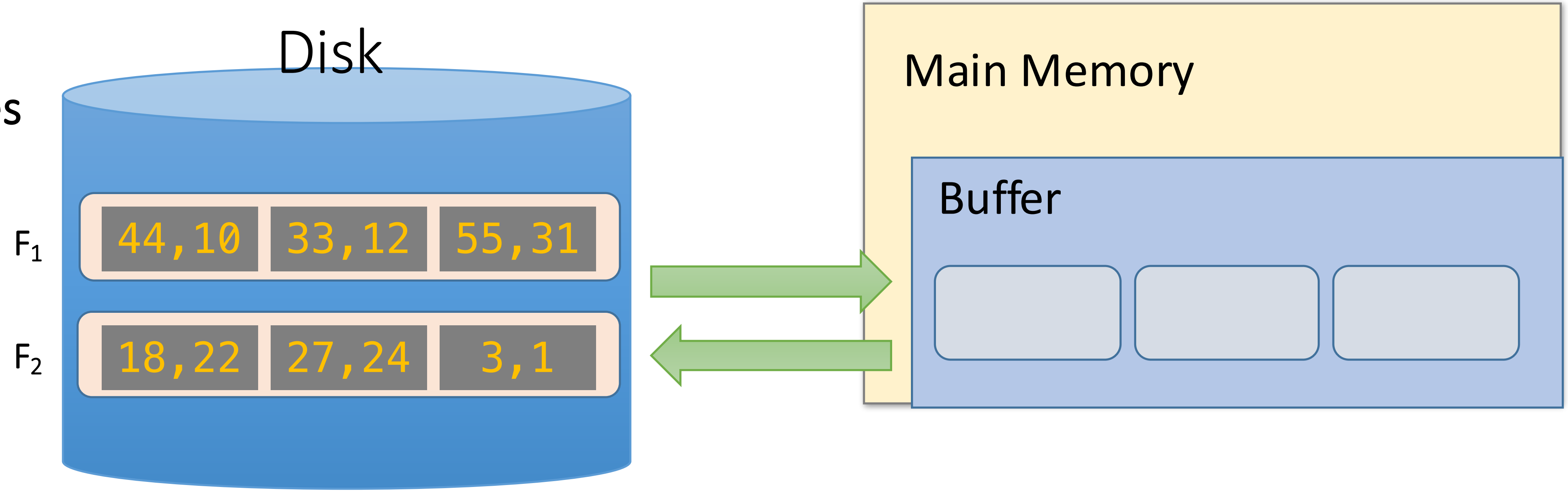
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted

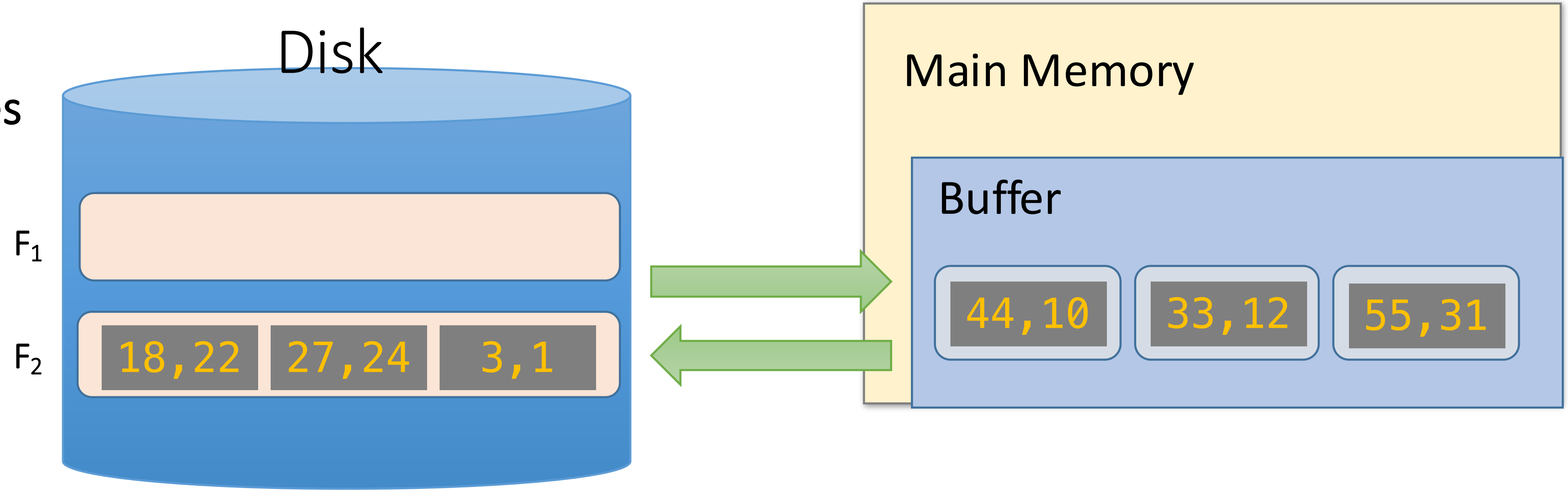


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file



Orange file = unsorted

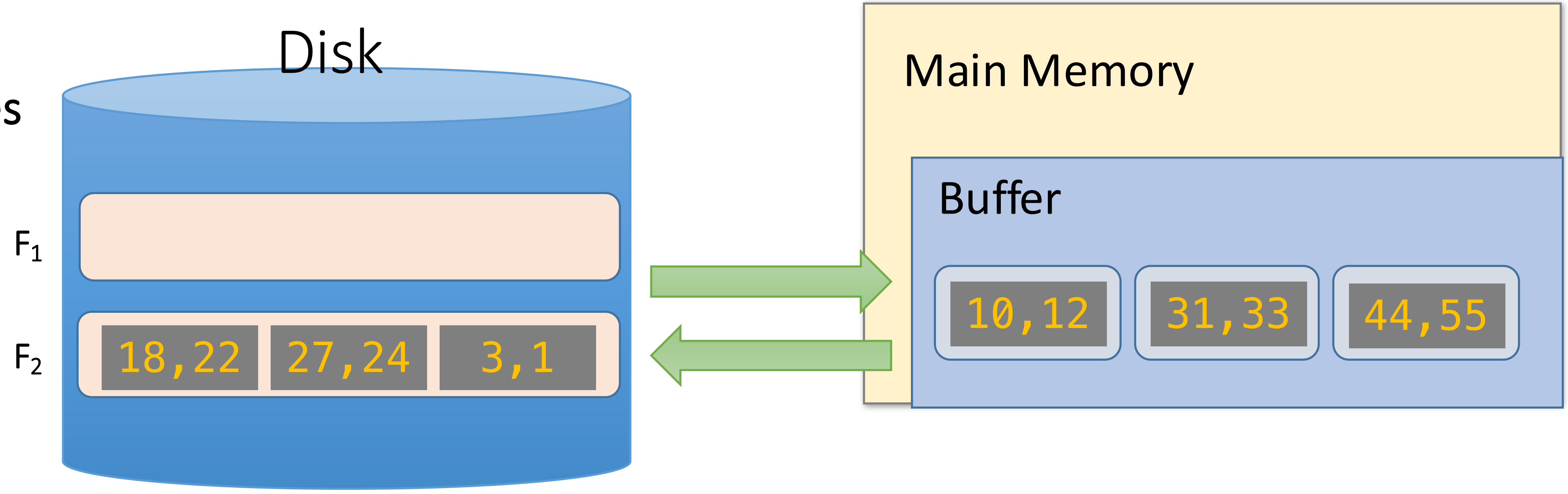
1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file

Orange file = unsorted

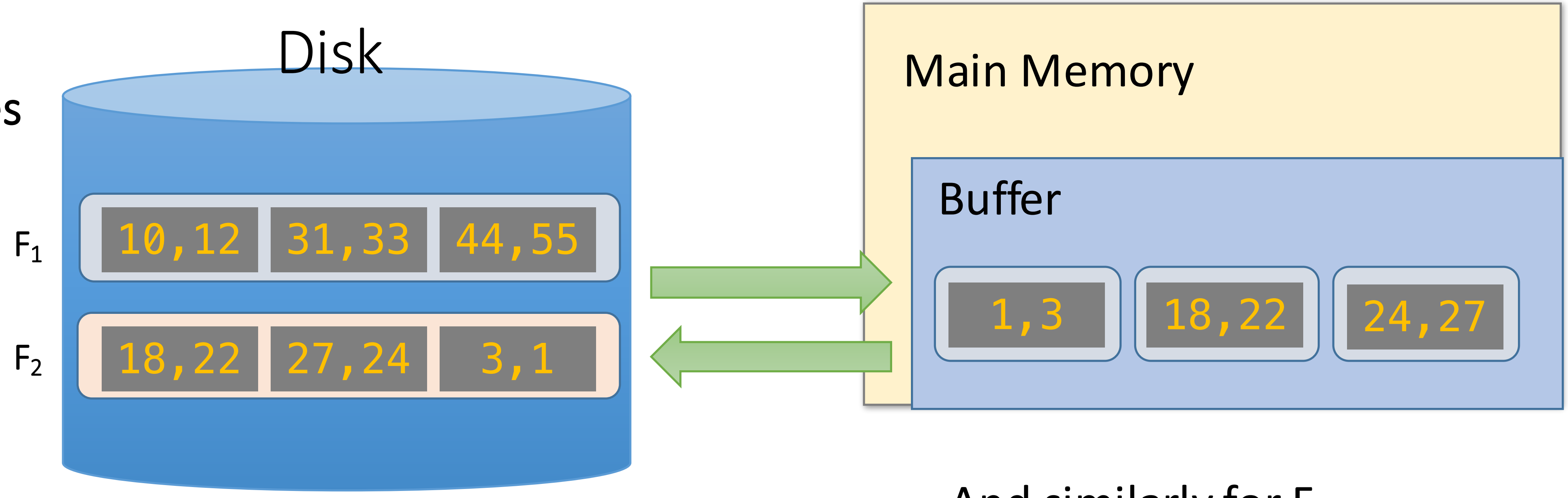


1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:
- 3 Buffer pages
- 6-page file

Each sorted file is a called a *run*



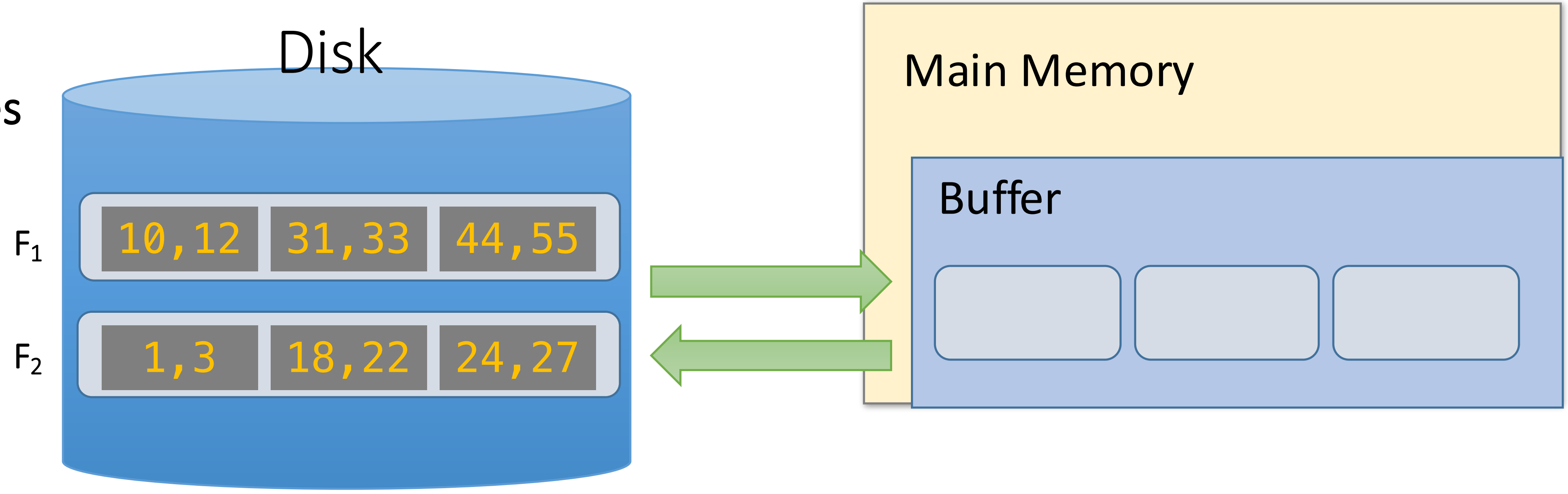
And similarly for F_2

1. Split into chunks small enough to **sort in memory**

External Merge Sort Algorithm

Example:

- 3 Buffer pages
- 6-page file



2. Now just run the **external merge** algorithm & we're done!

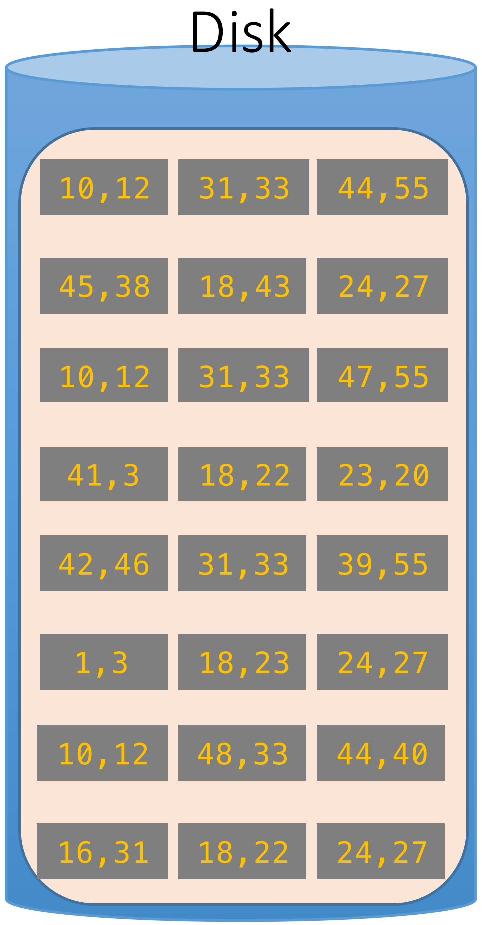
Calculating IO Cost

For 3 buffer pages, 6 page file:

1. Split into **two 3-page files** and **sort in memory**
   1. **= 1 R + 1 W for each file = 2*2 = 4 IO operations**

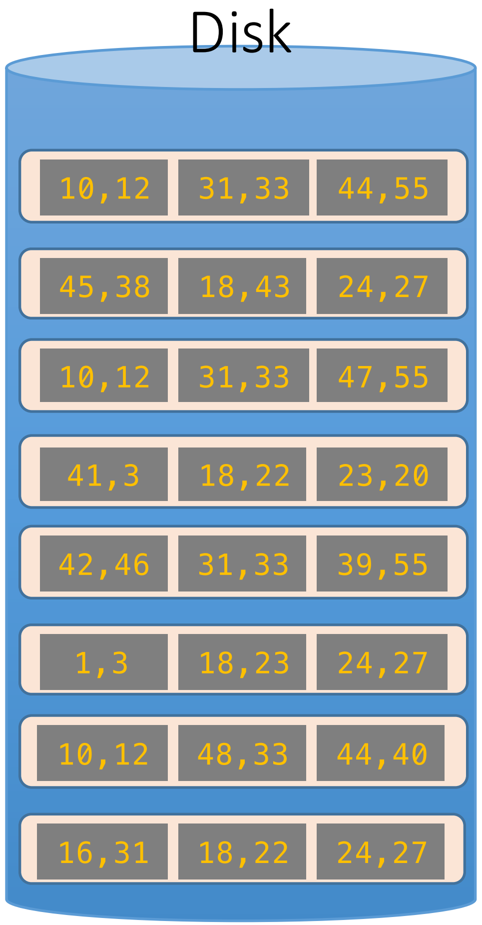
2. **Merge** each pair of sorted chunks ***using the external merge algorithm***
   1. **= 2*(3 + 3) = 12 IO operations**

Running External Merge Sort on Larger Files



Assume we still only have *3* buffer pages *(Buffer not pictured)*

Running External Merge Sort on Larger Files



1. Split into files small enough to sort in buffer…

Assume we still only have *3* buffer pages *(Buffer not pictured)*

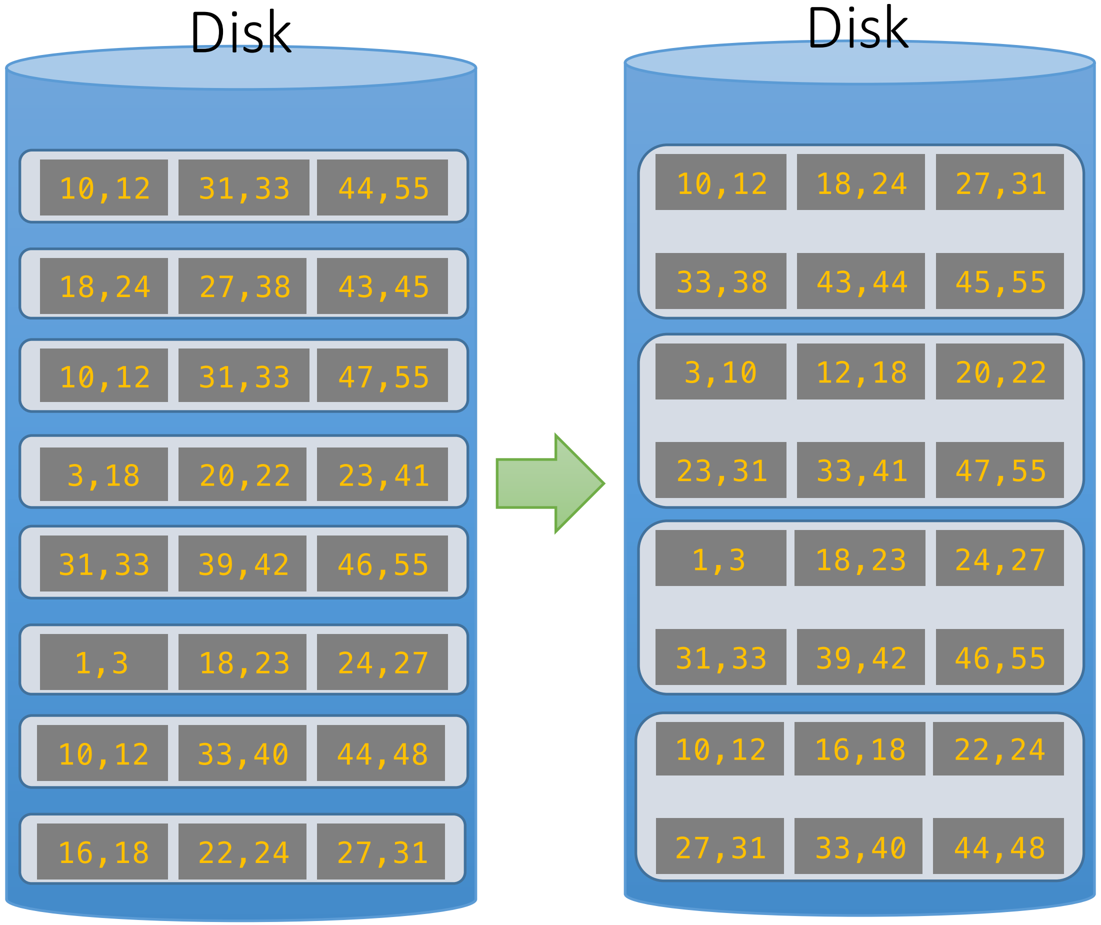
Running External Merge Sort on Larger Files

1. Split into files small enough to sort in buffer… and sort

Assume we still only have *3* buffer pages *(Buffer not pictured)*

Call each of these sorted files a ***run***

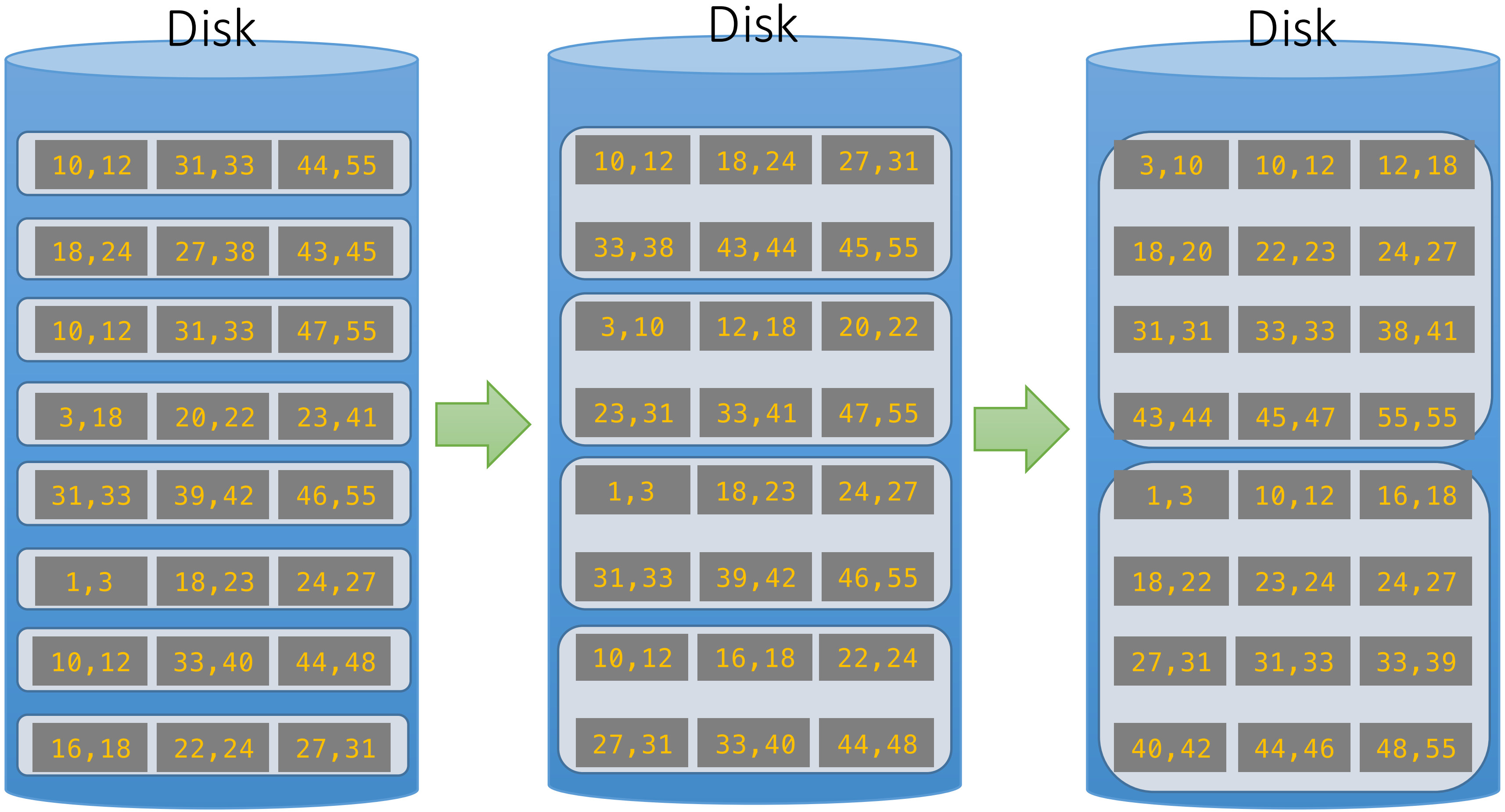
Running External Merge Sort on Larger Files



Assume we still only have *3* buffer pages *(Buffer not pictured)*

2. Now merge pairs of (sorted) files... **the resulting files will be sorted!**

Running External Merge Sort on Larger Files

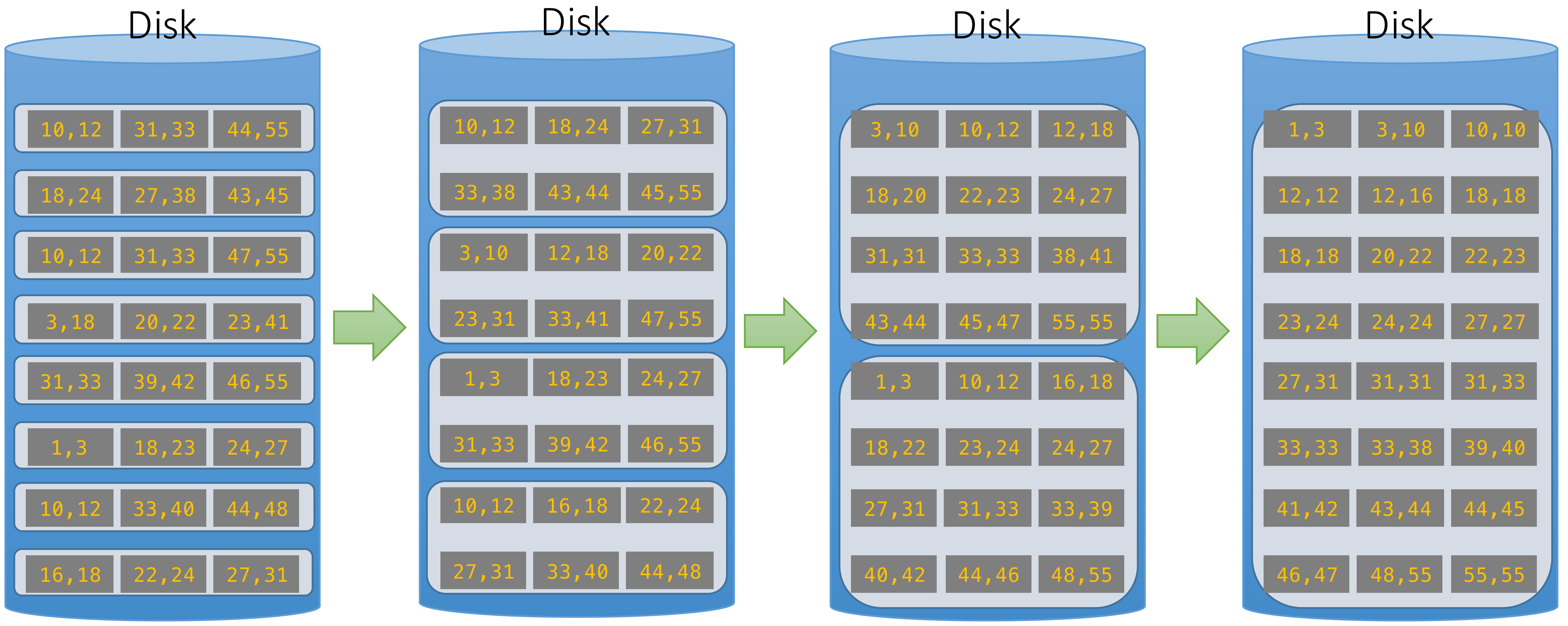


Assume we still only have *3* buffer pages *(Buffer not pictured)*

3. And repeat…

Call each of these steps a ***pass***

Running External Merge Sort on Larger Files

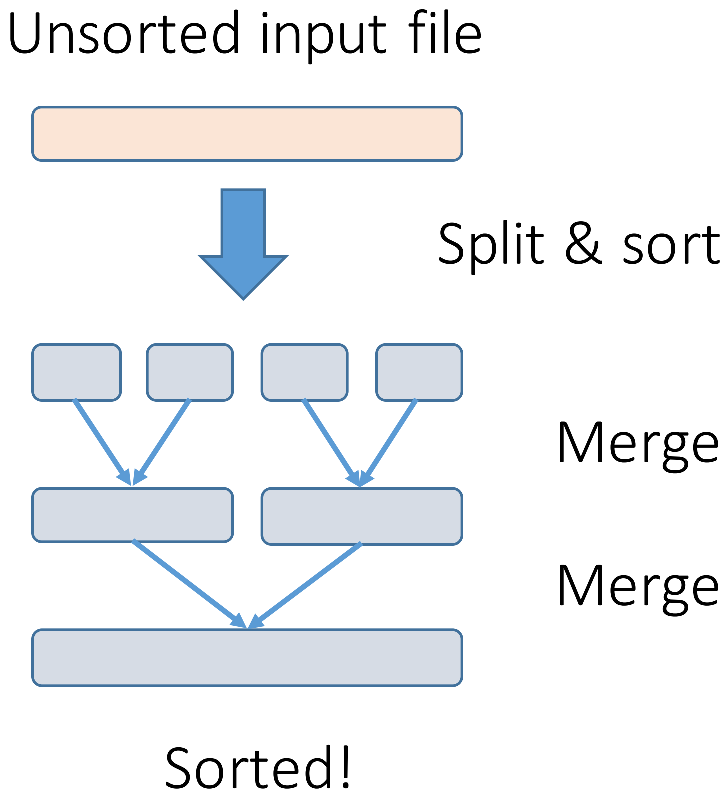


4. And repeat!

Simplified 3-page Buffer Version

Assume for simplicity that we split an N-page file into N single-page ***runs*** and sort these; then:

- We merge **N/2 runs** of length **1 page**
  - Each one takes 2*(1+1) = **4 IO**
- Then we merge **N/4 runs** of length **2 pages**
  - Each one takes 2*(2+2) = **8 IO**
- … In general, each pass will take ***2N*** IO!



For N pages, there will be $\lceil\log_2 N\rceil$+1 passes
→ 2N*($\lceil \boldsymbol{log_2 N} \rceil$+1) total IO cost!

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

1. **Increase length of initial runs**. Sort B+1 at a time!

At the beginning, we can split the N pages into runs of length B+1 and sort these in memory

IO Cost:

$$2N(\lceil \log_2 N \rceil + 1) \Rightarrow 2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length 1

Starting with runs of length ***B+1***

Using B+1 buffer pages to reduce # of passes

Suppose we have B+1 buffer pages now; we can:

2. Perform a B-way merge.

On each pass, we can merge groups of ***B*** runs at a time (vs. merging pairs of runs)!

IO Cost:

$$2N(\lceil \log_2 N \rceil + 1)$$

Starting with runs of length 1

$$2N\left(\left\lceil \log_2 \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Starting with runs of length ***B+1***

$$2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1\right)$$

Performing ***B***-way merges

Repacking

Repacking for even longer initial runs

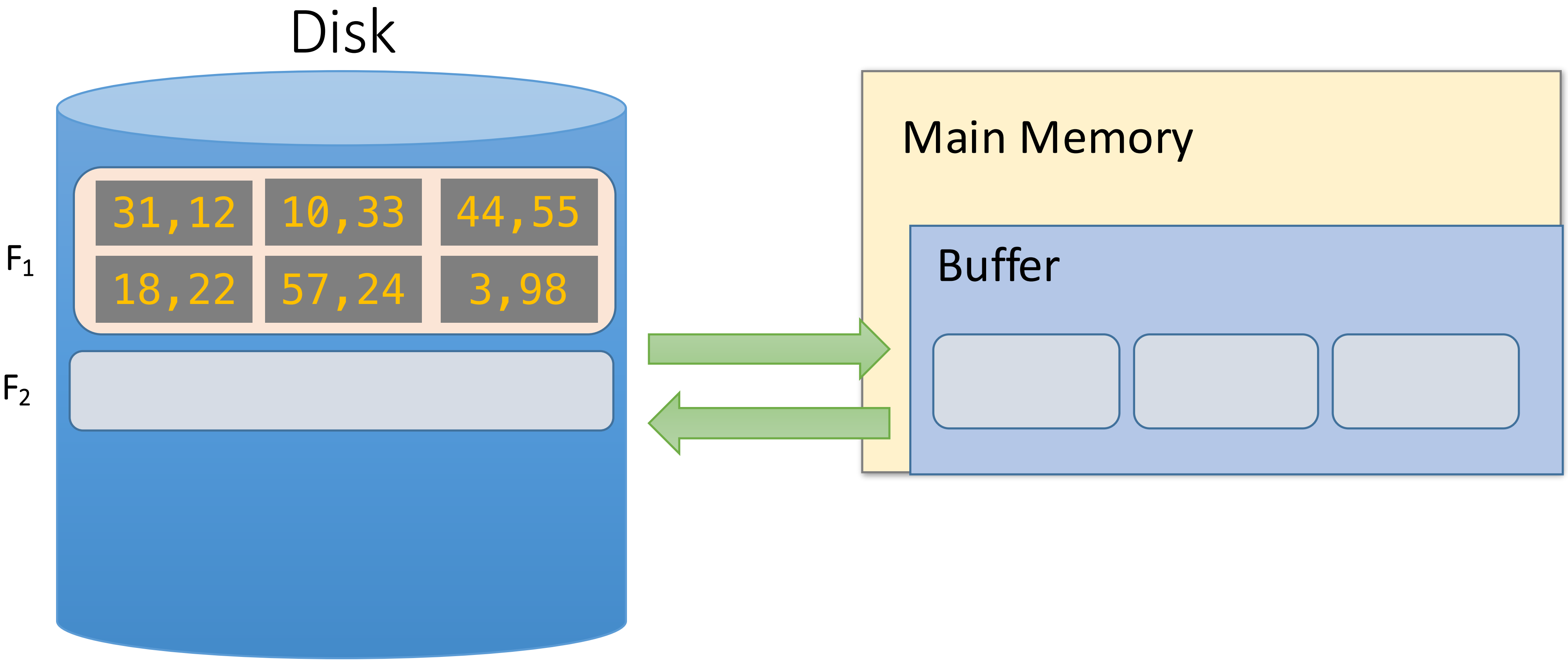
- With B+1 buffer pages, we can now start with ***B+1-length initial runs*** (and use ***B-way merges***) to get $2N(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{B+1}} \right\rceil + 1)$ IO cost…

- Can we reduce this cost more by getting even longer initial runs?

- Use <u>**repacking**</u>- produce longer initial runs by "merging" in buffer as we sort at initial stage

Repacking Example: 3 page buffer

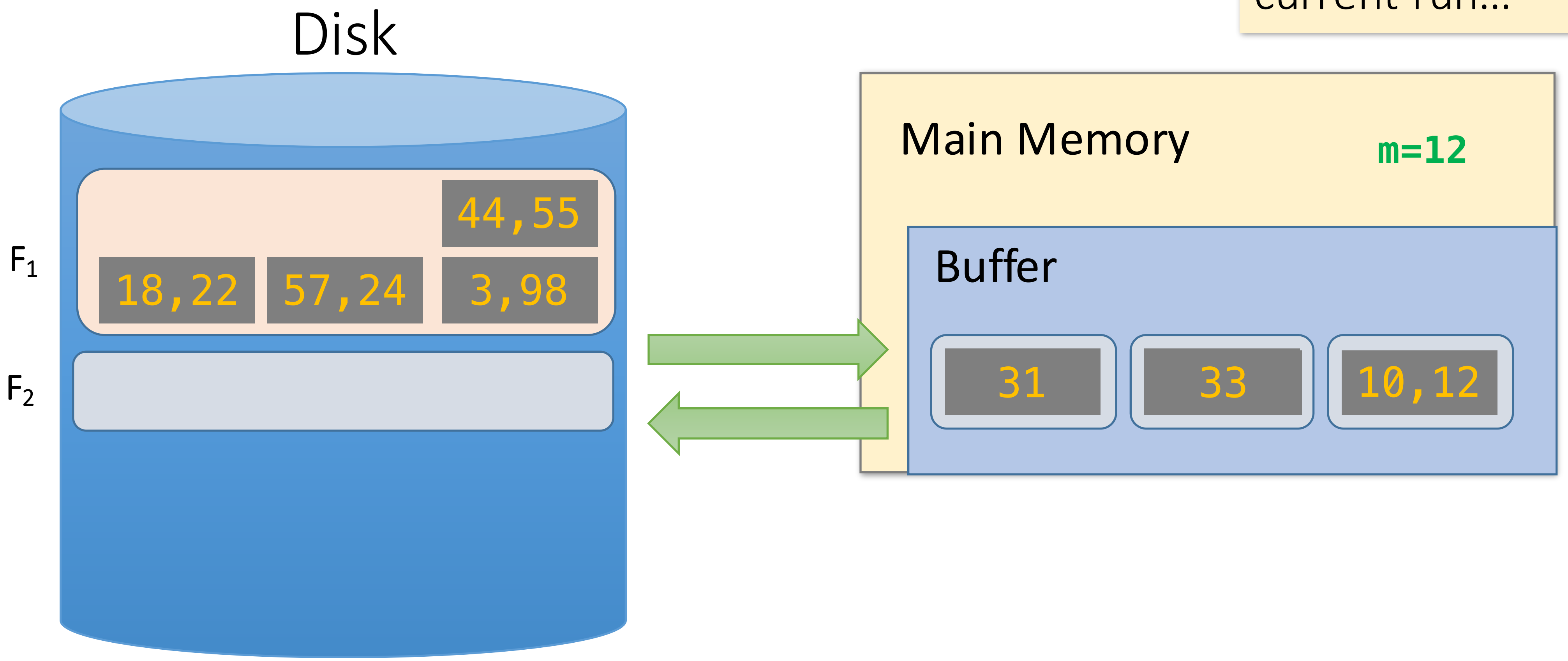
- Start with unsorted single input file, and load 2 pages



Repacking Example: 3 page buffer

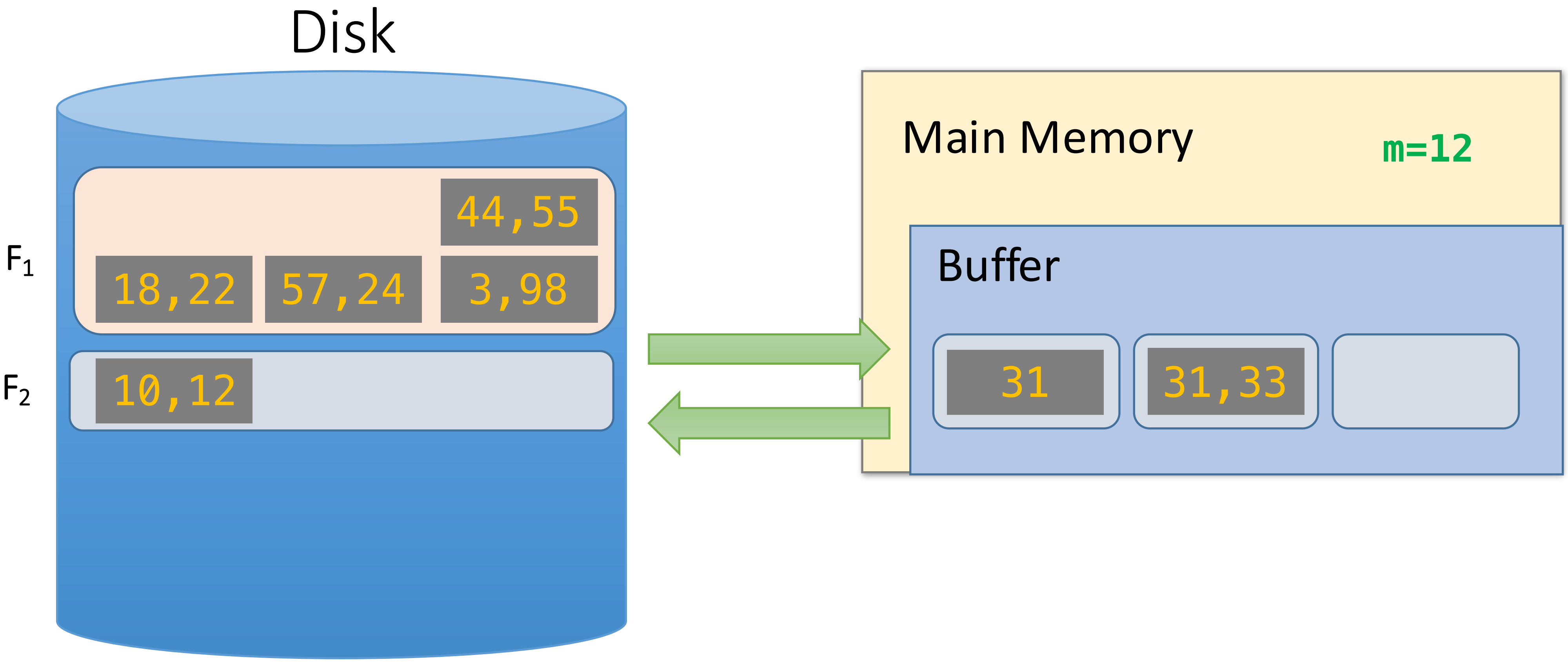
- Take the minimum two values, and put in output page

Also keep track of max (last) value in current run…



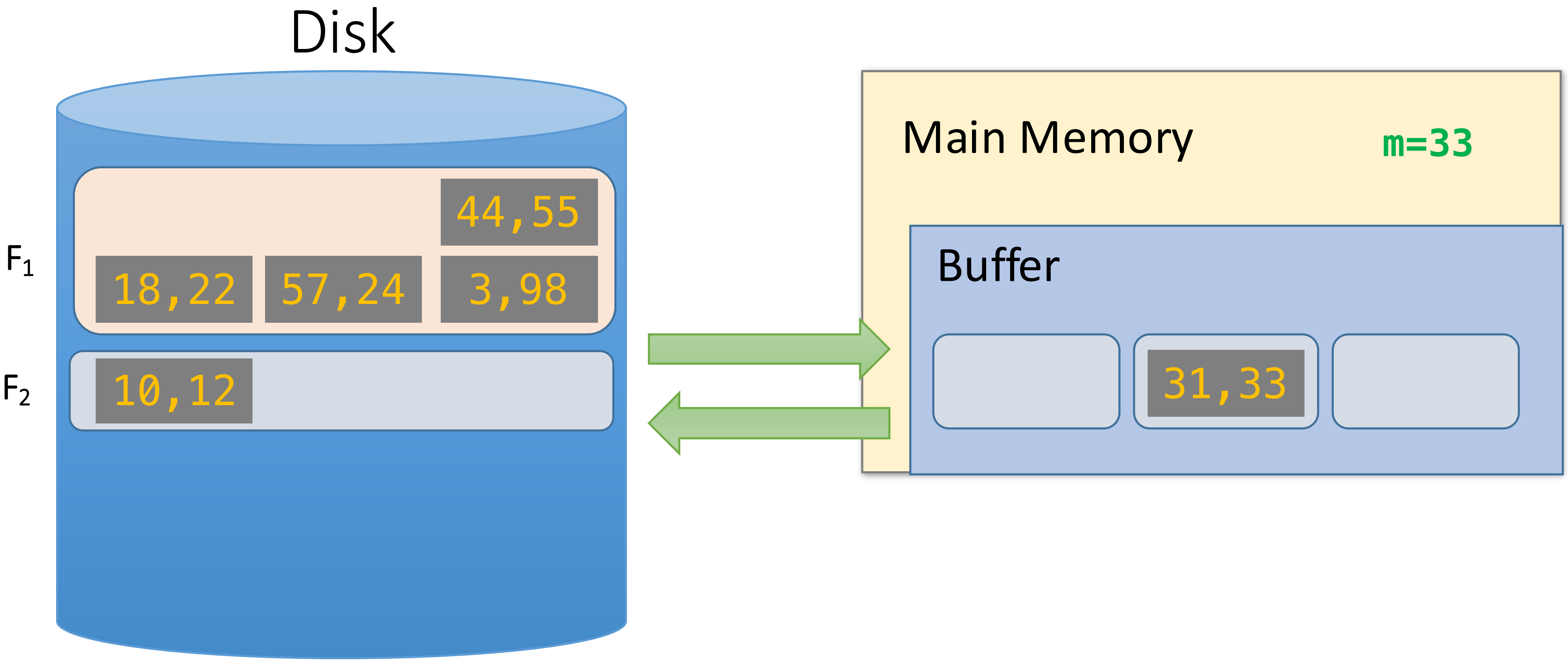
Repacking Example: 3 page buffer

- Next, ***repack***



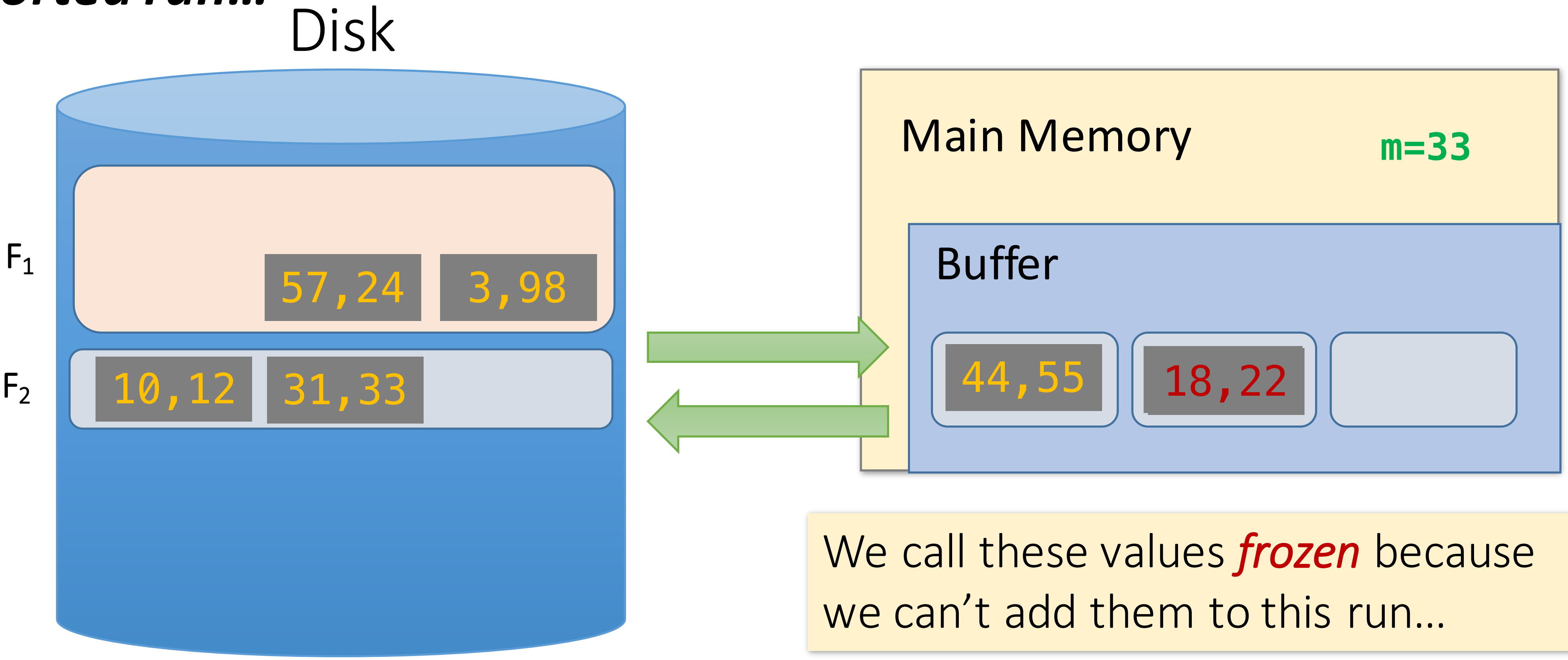
Repacking Example: 3 page buffer

- Next, ***repack***, then load another page and continue!



Repacking Example: 3 page buffer

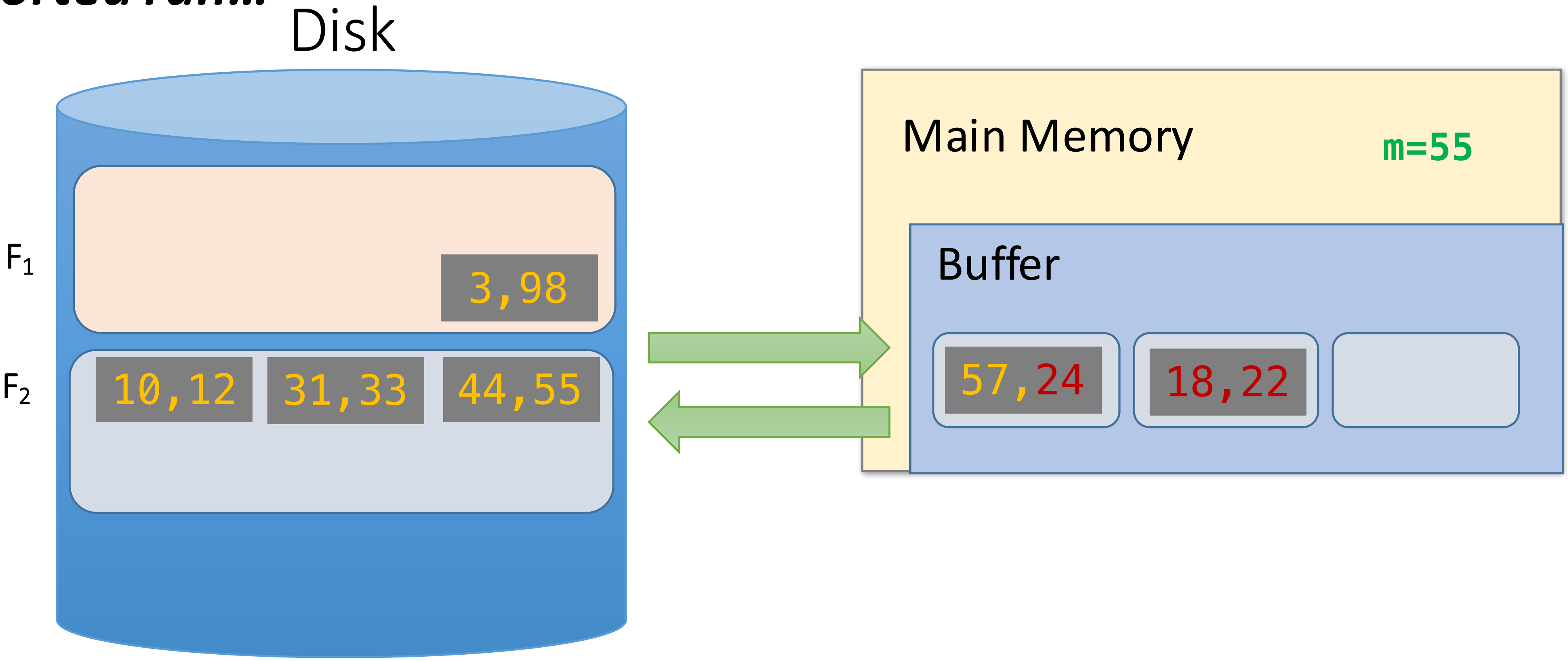
- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



We call these values *frozen* because we can't add them to this run…

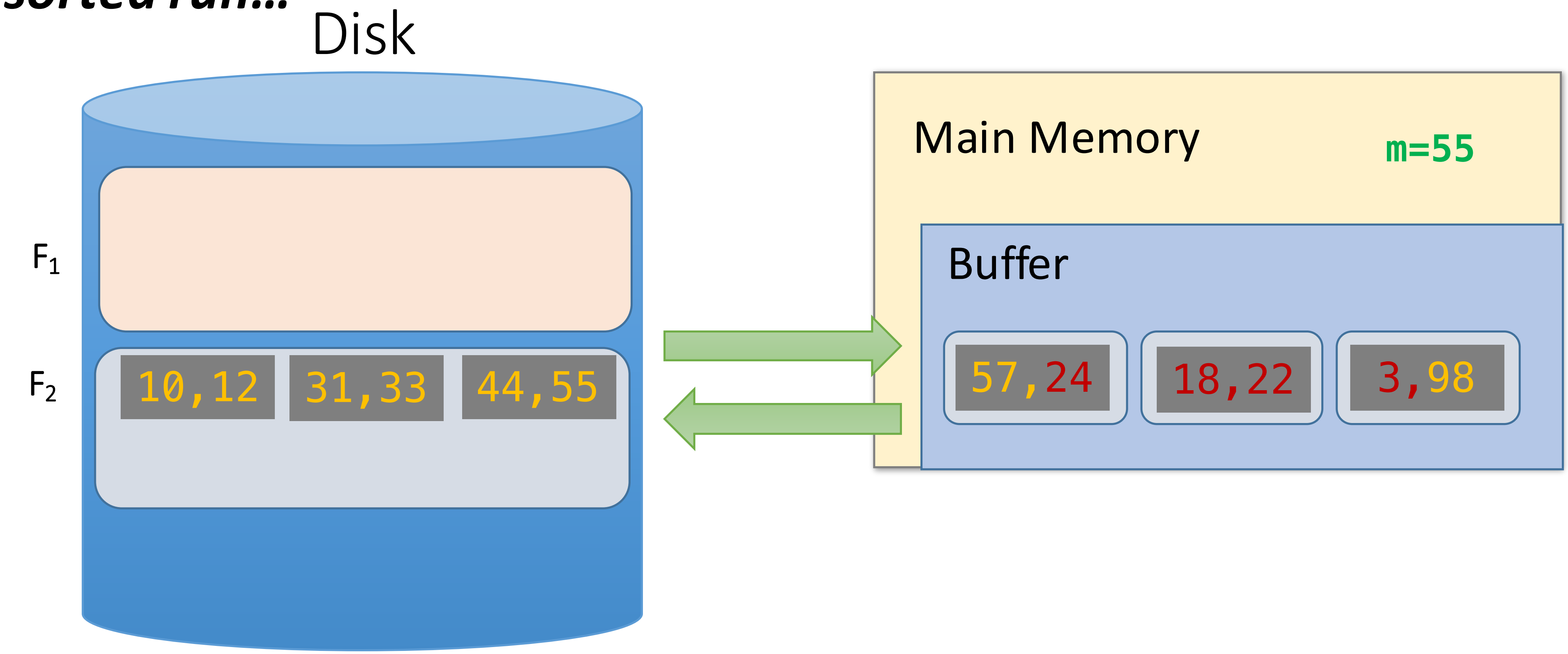
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



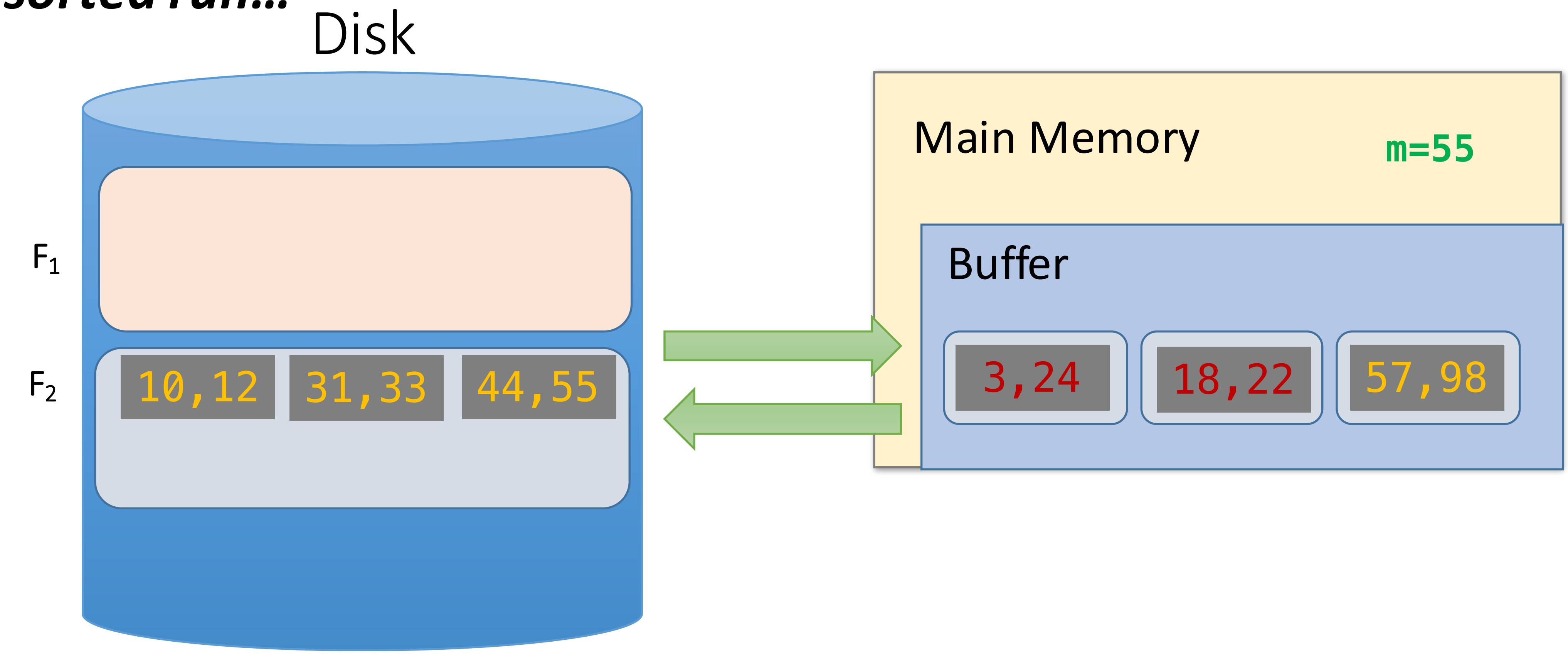
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



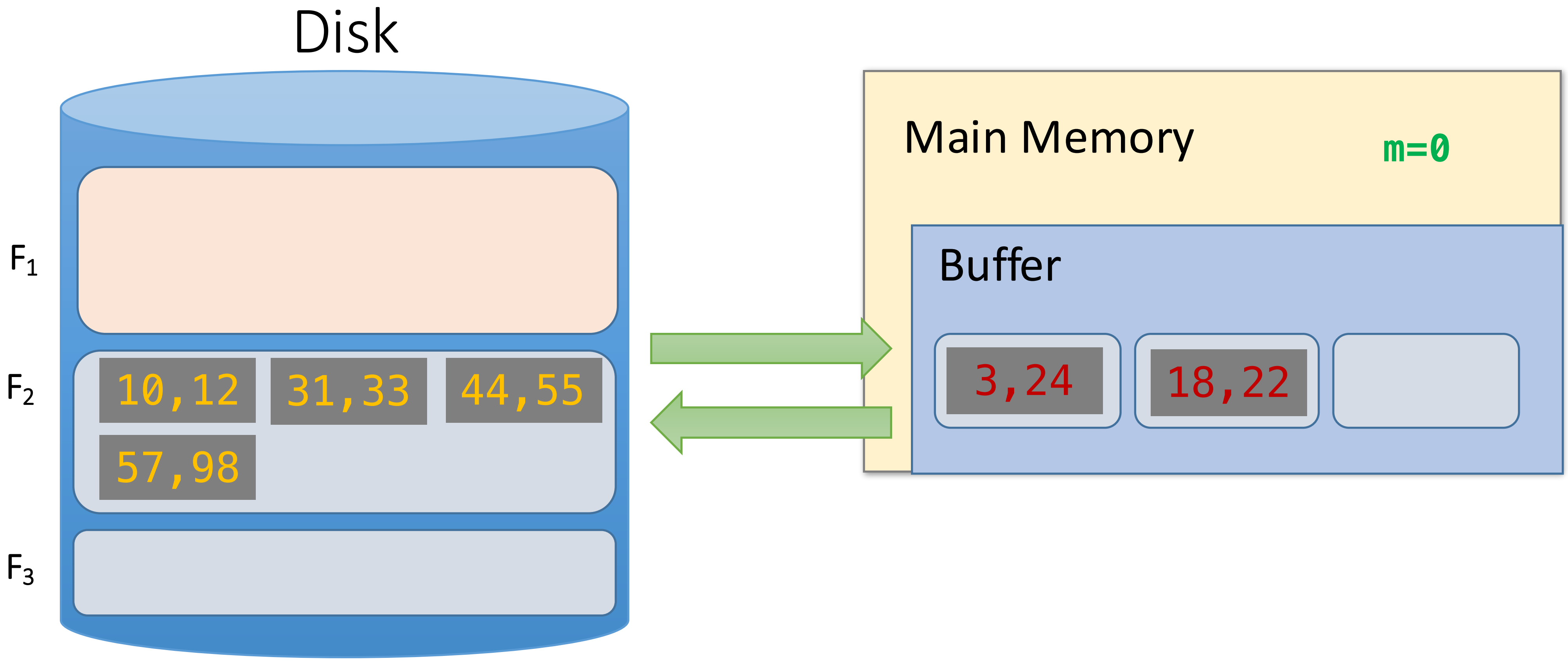
Repacking Example: 3 page buffer

- Now, however, ***the smallest values are less than the largest (last) in the sorted run…***



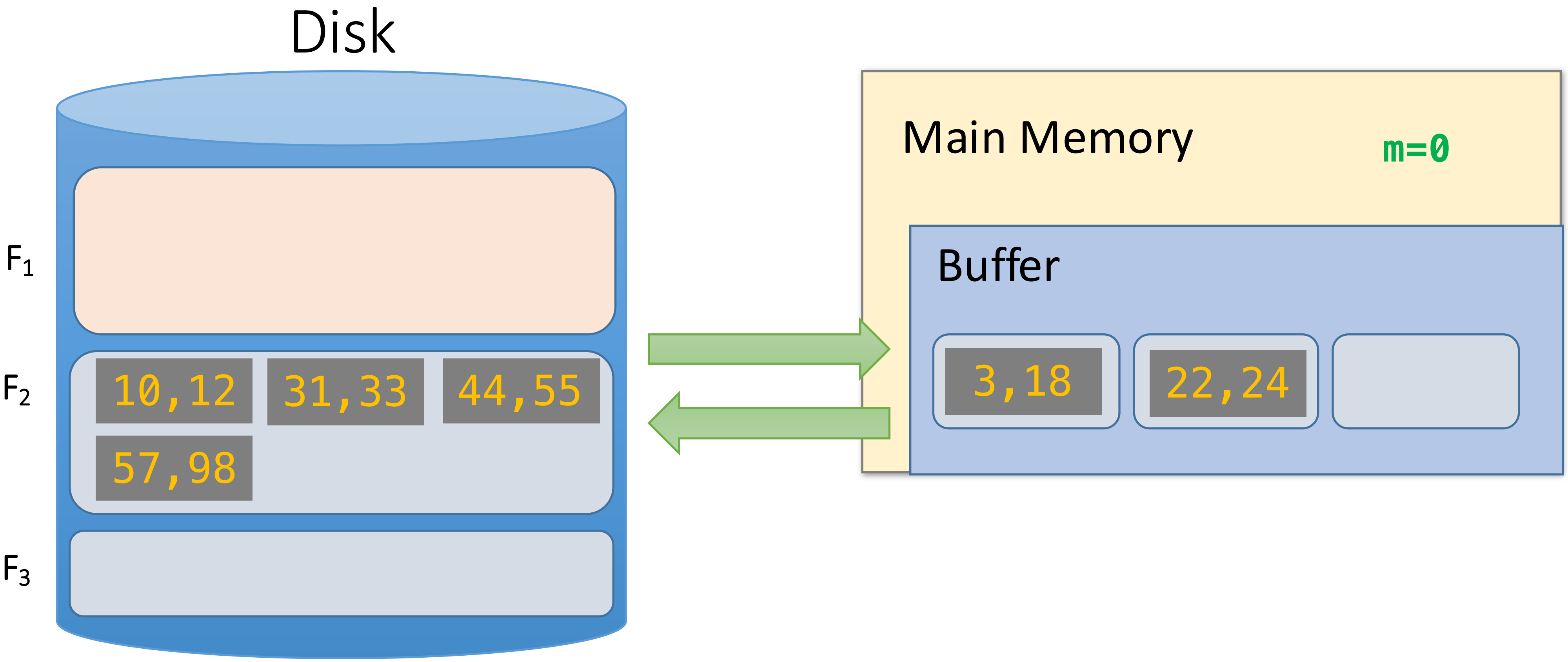
Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking Example: 3 page buffer

- Once ***all buffer pages have a frozen value,*** or input file is empty, start new run with the frozen values



Repacking

- Note that, for buffer with B+1 pages:
  - If input file is sorted → nothing is frozen → we get **a single** run!
  - If input file is reverse sorted (worst case) → everything is frozen → we get runs of length **B+1**

- In general, with repacking we do **no worse** than without it!

- What if the file is already sorted?

- Engineer's approximation: runs will have **~2(B+1)** length

$$\sim 2N\left(\left\lceil \log_B \frac{\boldsymbol{N}}{\boldsymbol{2}(\boldsymbol{B}+\boldsymbol{1})} \right\rceil + 1\right)$$

Summary

- Basics of IO and buffer management.
  - See notebook for more fun! (Learn about *sequential flooding*)
- We introduced the IO cost model using **sorting**.
  - Saw how to do merges with few IOs,
  - Works better than main-memory sort algorithms.
- Described a few optimizations for sorting

2. Indexes

What you will learn about in this section

1. Indexes: Motivation

2. Indexes: Basics

3. ACTIVITY: Creating indexes

Index Motivation

`Person(name, age)`

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

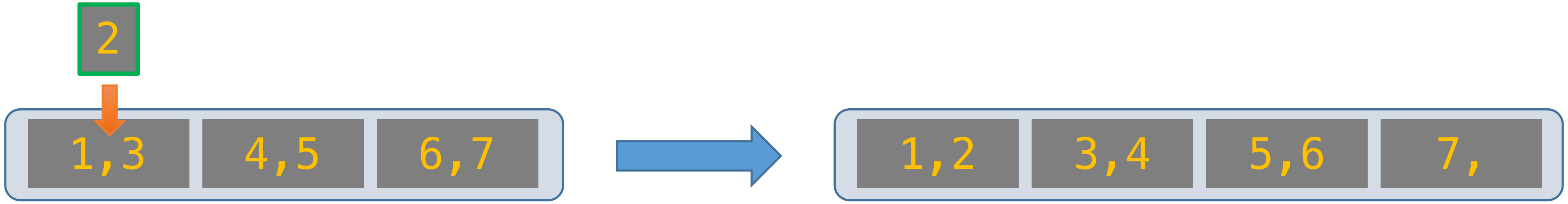
- ***First idea:*** Sort the records by age… we know how to do this fast!

- 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 (covered next), sometimes hash indexes (not covered here)

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 the full rows it points to (*primary index*) or pointers to those rows (*secondary index*)
  - We'll mainly consider secondary indexes

Operations on an Index

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

- Insert / Remove entries
  - Bulk Load / Delete. 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

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	…

By_Author_Title_Index

Author	Title	BID
Dostoyevsky	Crime and Punishment	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 **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
```

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***

The data structures we present here are "IO aware"

- Hash Tables *(not covered)*
  - There are variants of this basic structure to deal with IO
  - Called ***linear*** or ***extendible hashing-*** IO aware!

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

[Activity-13.ipynb]

2. B+ Trees

What you will learn about in this section

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

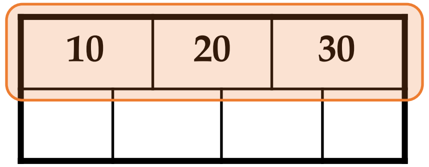
B+ Trees

- 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

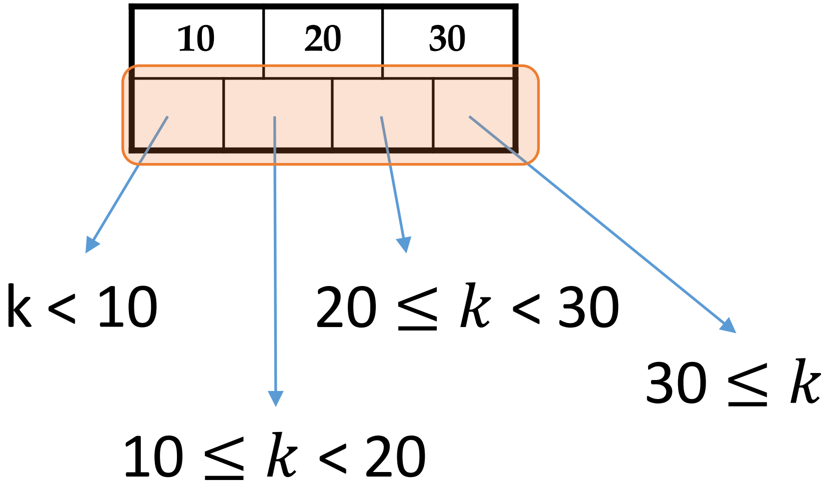


Parameter ***d*** = the degree

Each *non-leaf ("interior")* ***node*** has ≥ d and ≤ 2d ***keys****

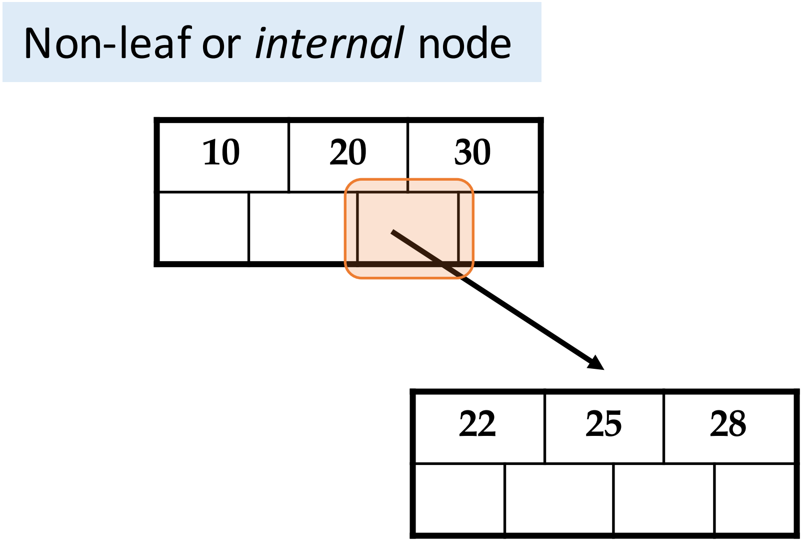
except for root node, which can have between **2 and 2d keys*

B+ Tree Basics



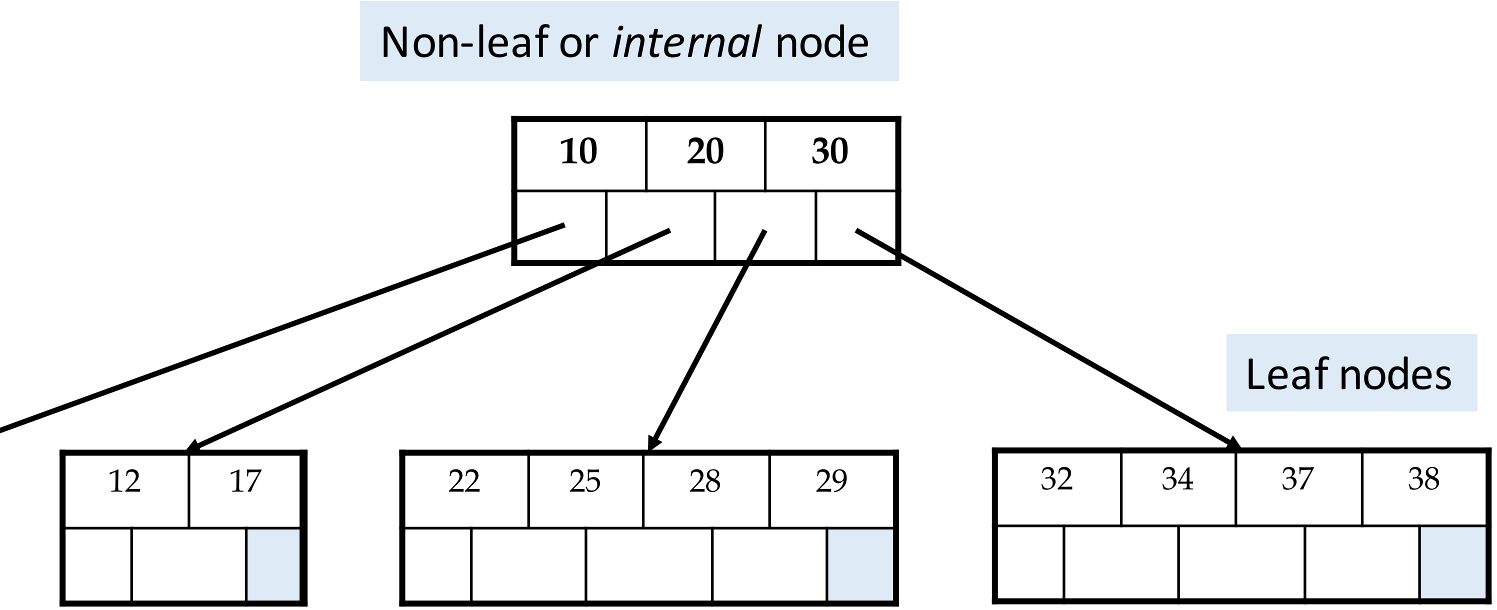
The *n* keys in a node define *n+1* ranges

B+ Tree Basics



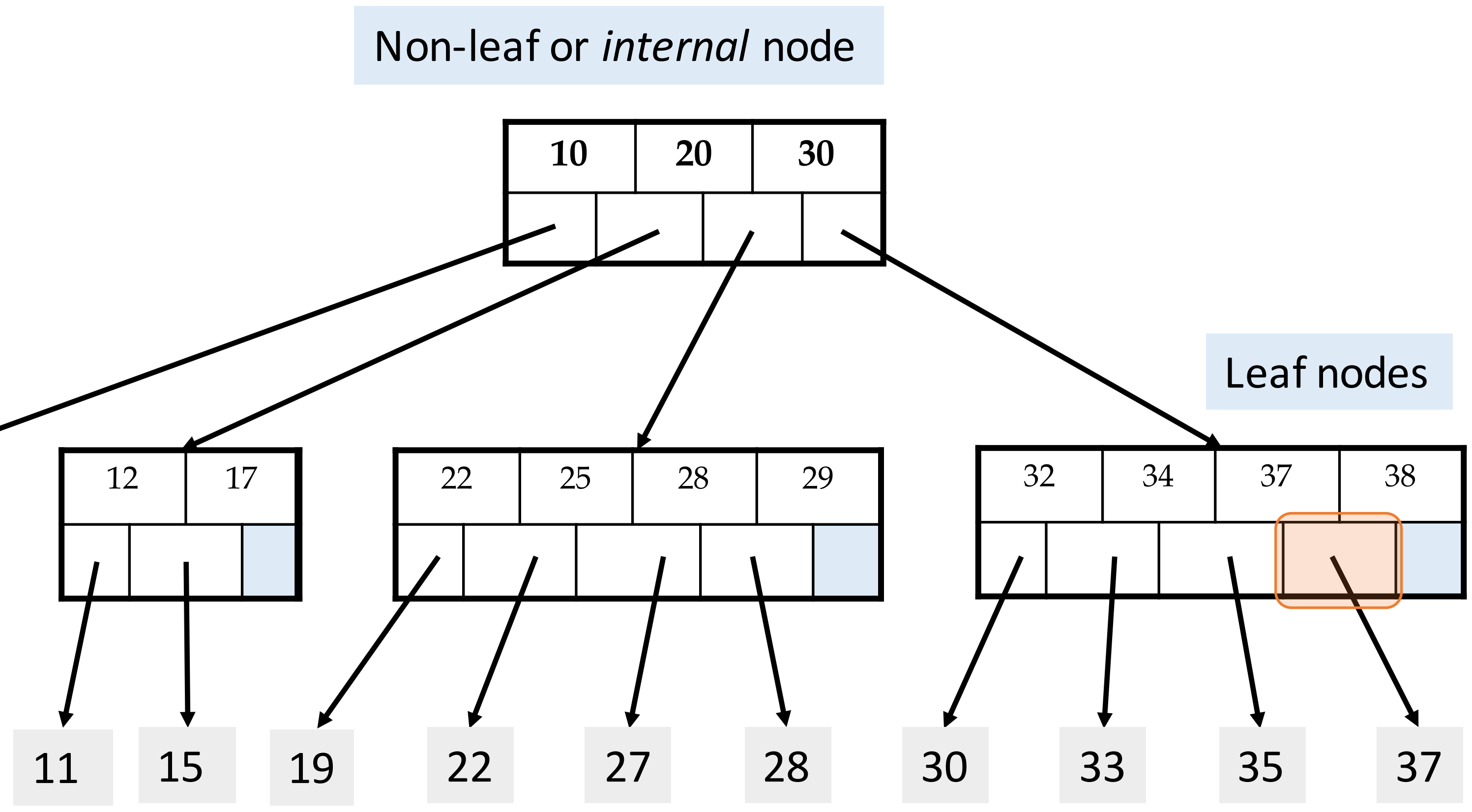
For each range, in a *non-leaf* node, there is a **pointer** to another node with keys in that range

B+ Tree Basics



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

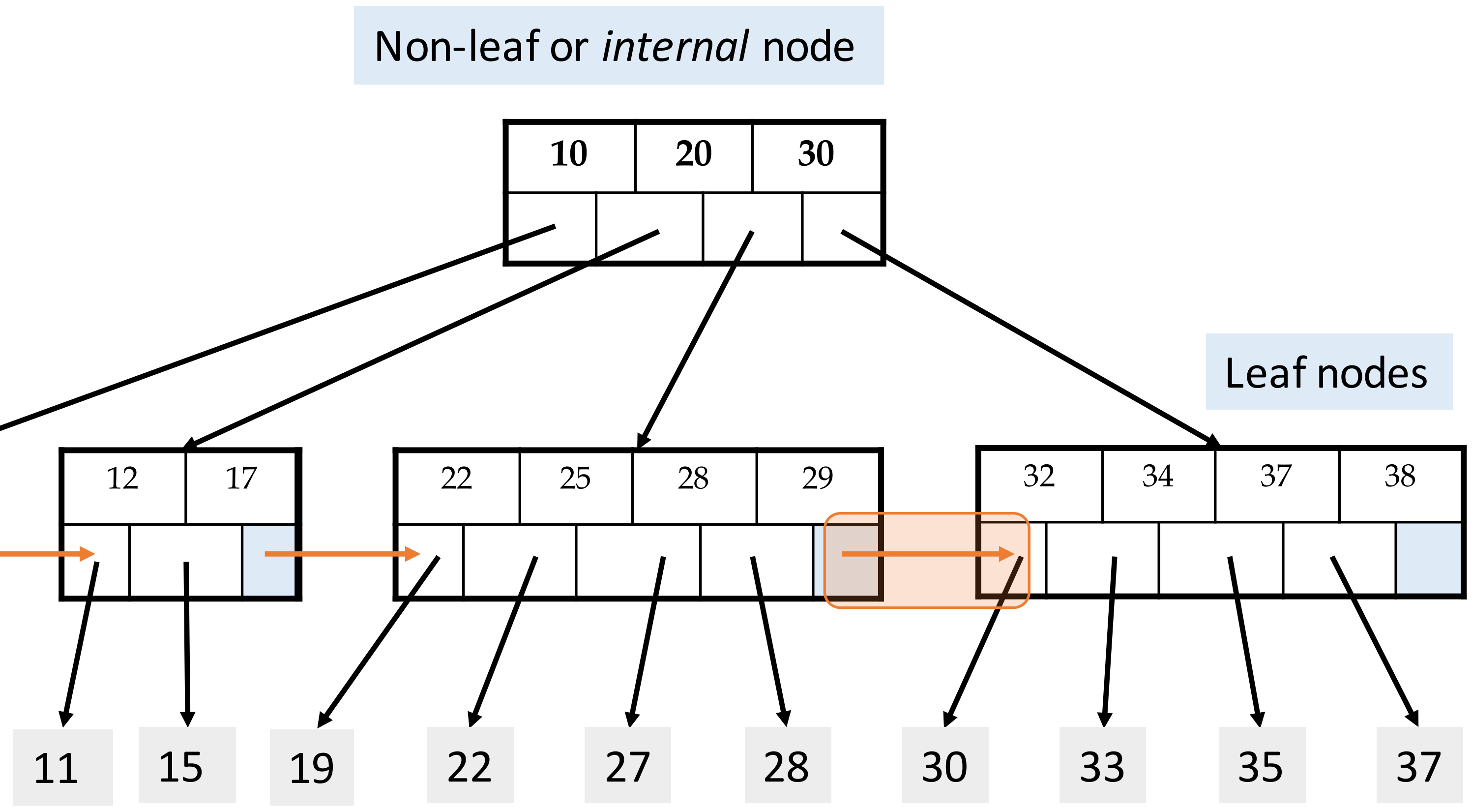
B+ Tree Basics



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

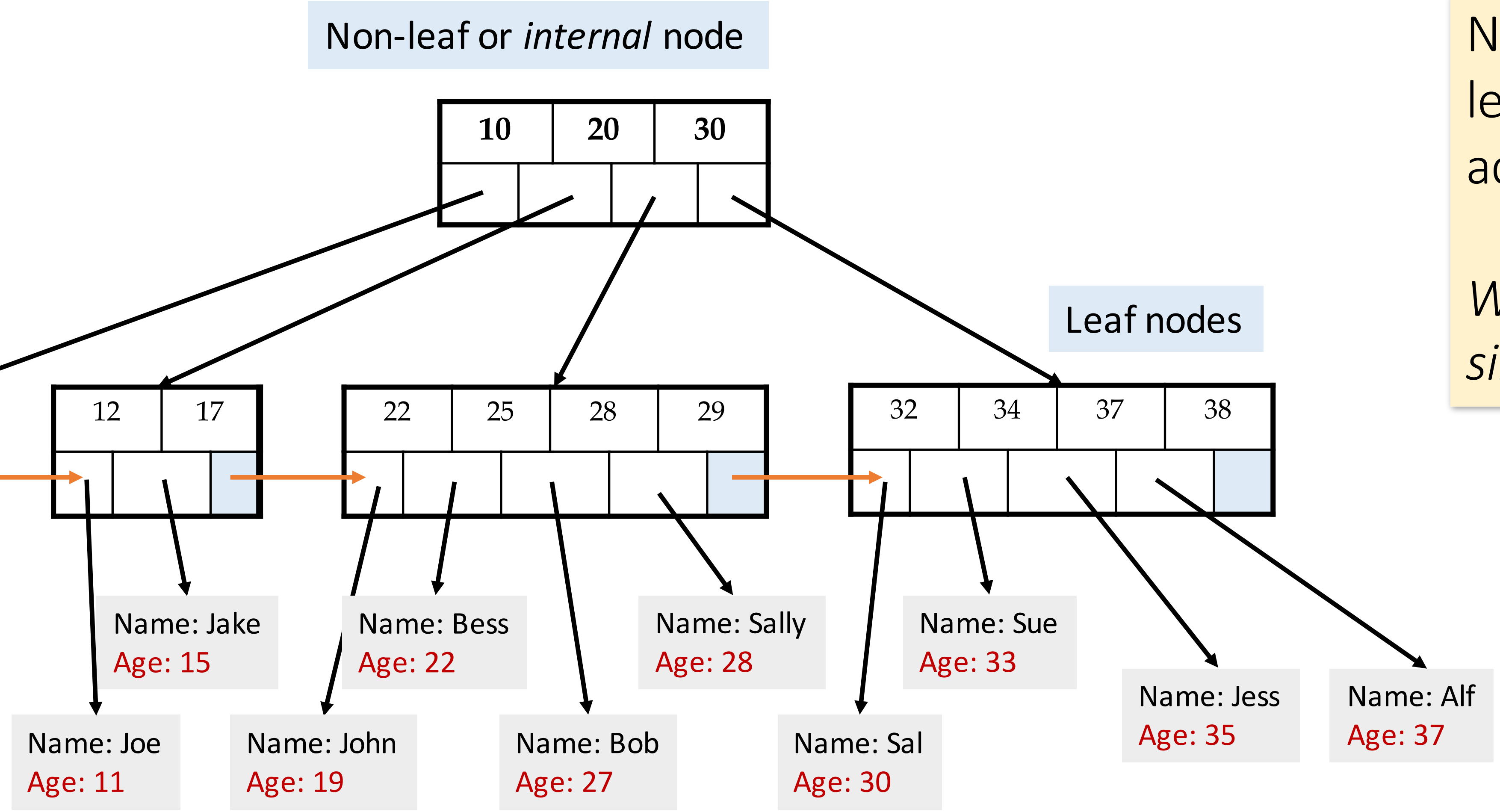


Leaf nodes also have between *d* and *2d* keys, and are different in that:

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

```
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

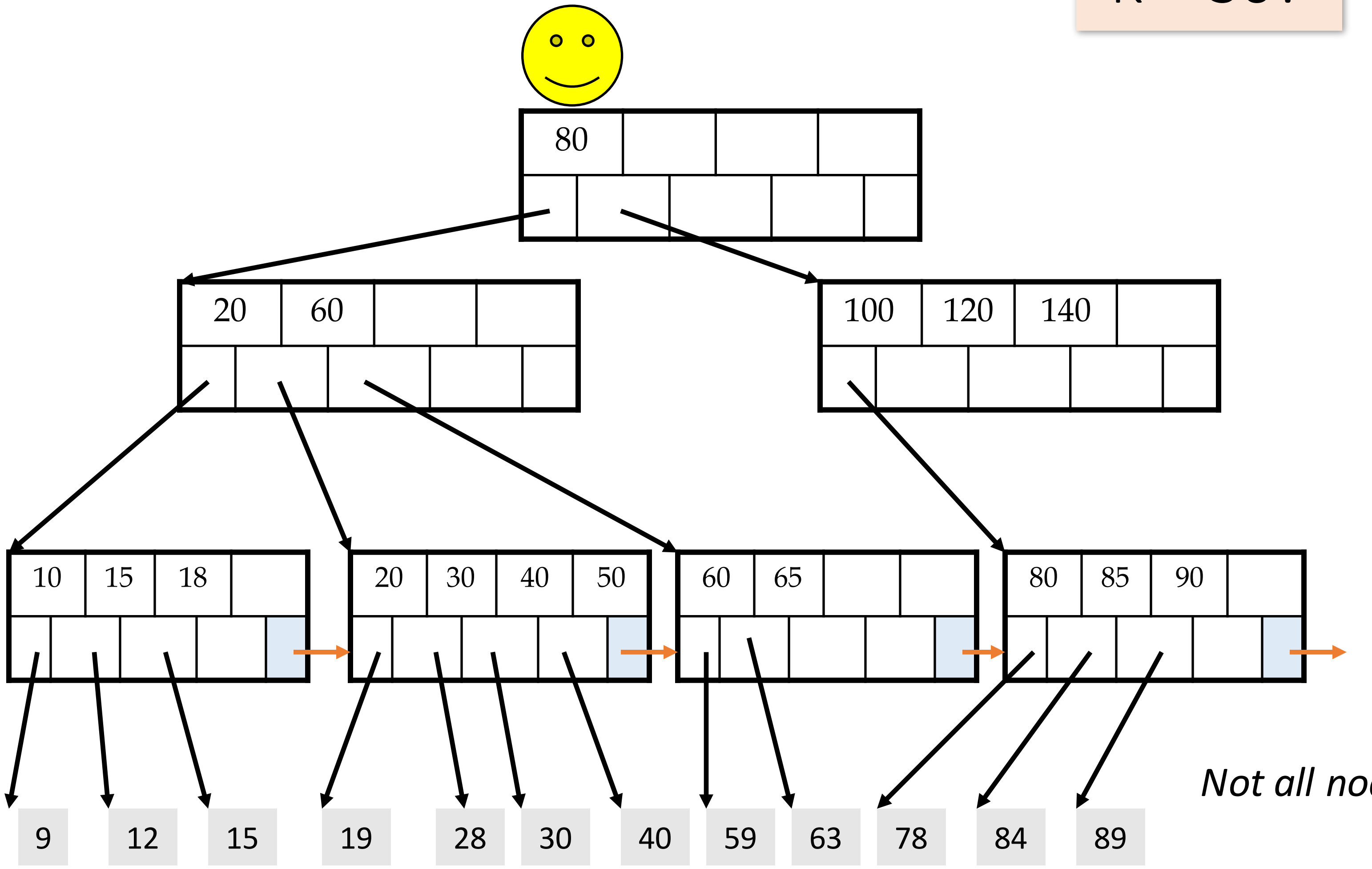
K = 30?

30 < 80

30 in [20,60)

30 in [30,40)

To the data!



Not all nodes pictured

B+ Tree Range Search Animation

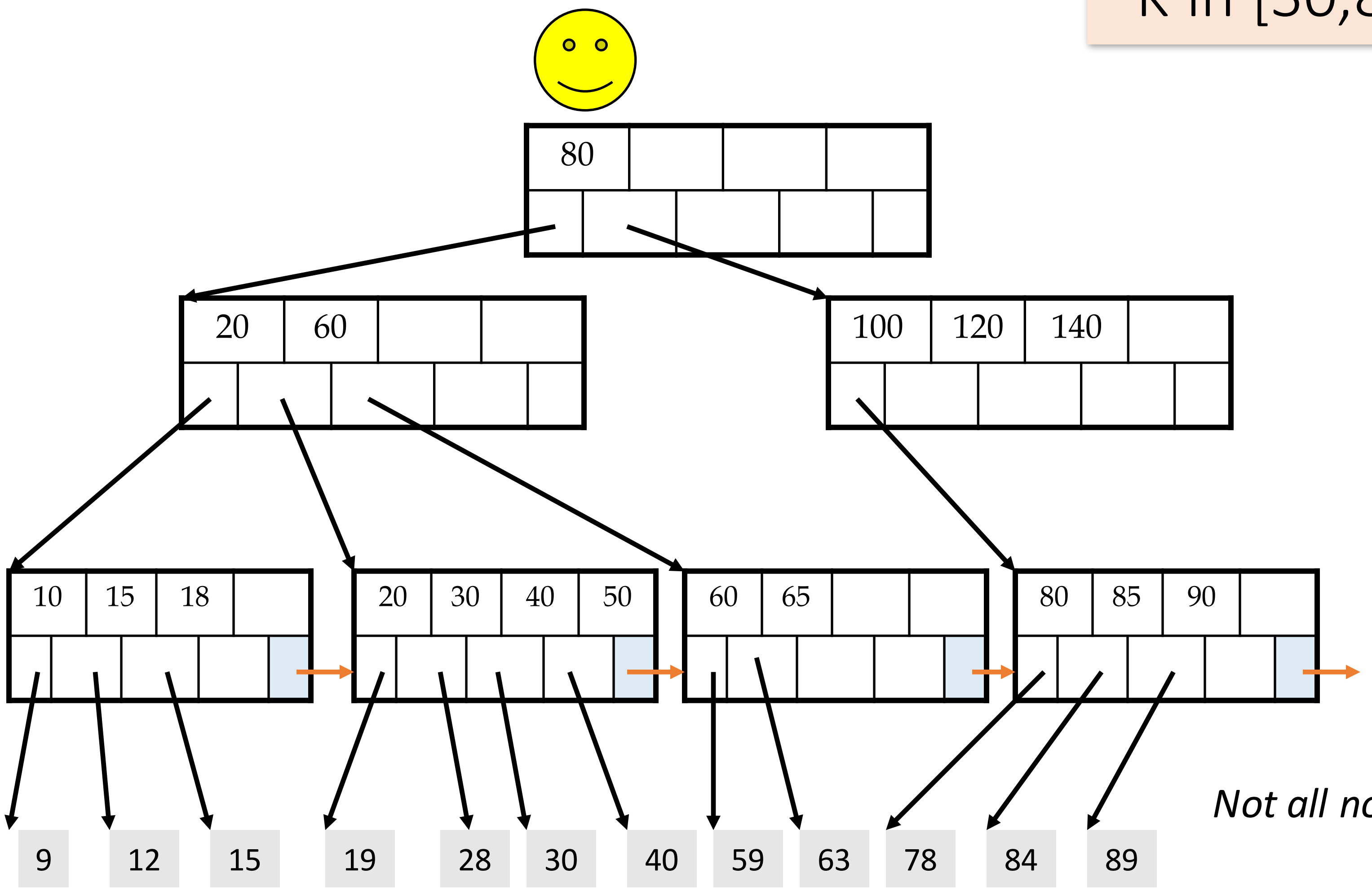
K in [30,85]?

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 byes

NB: Oracle allows 64K blocks
→ d <= 2666

- 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*** (= ***2d***)

- 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!

- A TiB = 2^{40} Bytes. What is the height of a B+ Tree that indexes it (with 64K pages)?
  - $(2*2666)^h = 2^{40}$ → ***h = 4***

The <u>fanout</u> is defined as the maximum number of pointers to child nodes per node

The known universe contains ~10^{80} particles… what is the height of a B+ Tree that indexes these?

B+ Trees in Practice

- Typical order: d=100. Typical fill-factor: 67%.
  - average fanout = 133
- Typical capacities:
  - Height 4: 133^4 = 312,900,700 records
  - Height 3: 133^3 = 2,352,637 records
- Top levels of tree sit *in the buffer pool*:
  - Level 1 = 1 page = 8 Kbytes
  - Level 2 = 133 pages = 1 Mbyte
  - Level 3 = 17,689 pages = 133 MBytes

<u>Fill-factor</u> is the percent of available slots in the B+ Tree that are filled; is usually < 1 to leave slack for (quicker) insertions

Typically, only pay for one IO!

Simple Cost Model for Search

- Let:
  - F = fanout
  - N = the total number of records
  - fill-factor = 2/3
    - → 1.5N is effective # of records B+ Tree needs to have room for
  - L_B = # of levels of the B+ Tree in main memory

- For exact search: $\mathbf{\log_F 1.5N - L_B + 1}$

Only pay for reading from B+ Tree nodes on disk (plus reading the actual record)

Simple Cost Model for Search

- Let:
  - F = fanout
  - N = the total number of records
  - fill-factor = 2/3
    - → 1.5N is effective # of records B+ Tree needs to have room for
  - L_B = # of levels of the B+ Tree in main memory
- For searching a range R: $\mathbf{\log_F 1.5N - L_B + R}$

Fast Insertions & Self-Balancing

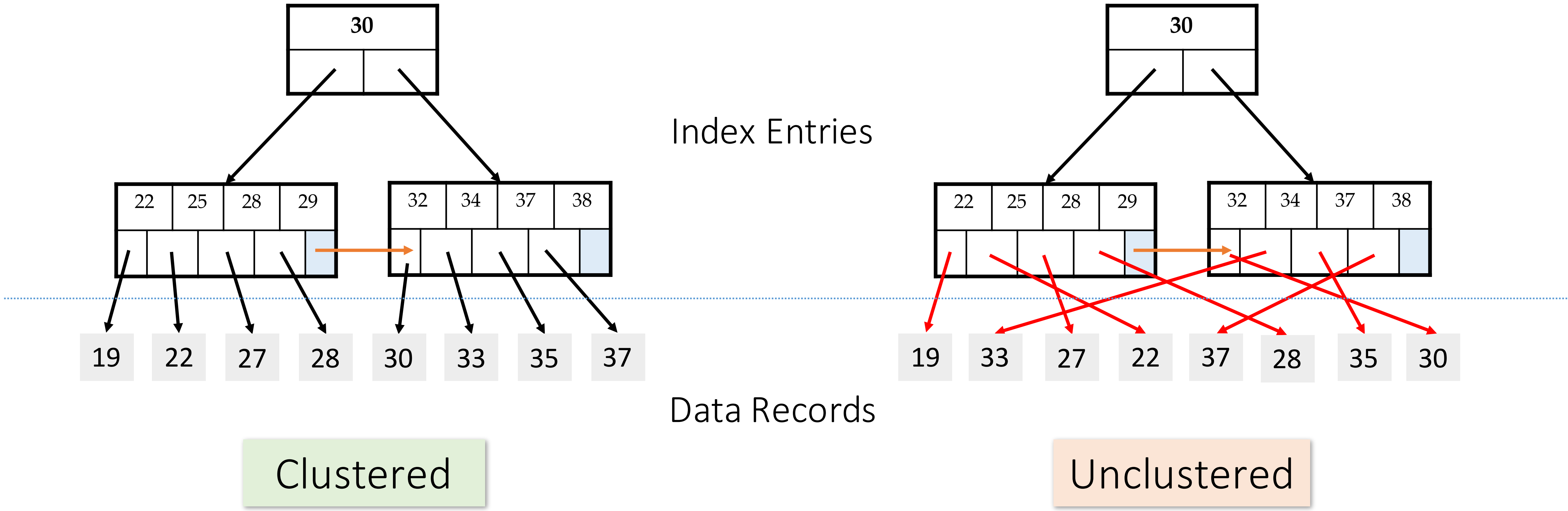
- We won’t go into specifics of B+ Tree insertion algorithm, but has several attractive qualities:
  - **~ Same cost as exact search**
  - ***Self-balancing:*** B+ Tree remains **balanced** (with respect to height) even after insert

B+ Trees also (relatively) fast for insertions!

Clustered Indexes

An index is ***clustered*** if the underlying data is ordered in the same way as the index's data entries.

Clustered vs. Unclustered Index



Clustered vs. Unclustered Index

- Recall that for a disk with block access, **sequential IO is much faster than random IO**
- For exact search, no difference between clustered / unclustered
- For range search over R values: difference between **1 random IO + R sequential IO**, and **R random IO**:
  - For 100,000 records- difference between ~10ms and ~10min!

Summary

- We covered an algorithm + some optimizations for sorting larger-than-memory files efficiently
  - An ***IO aware*** algorithm!
- We create **indexes** over tables in order to support ***fast (exact and range) search*** and ***insertion*** over ***multiple search keys***
- **B+ Trees** are one index data structure which support very fast exact and range search & insertion via ***high fanout***
  - ***Clustered vs. unclustered*** makes a big difference for range queries too