File Structures

Ch08. B. Sorting of Large Files

2020. Spring

Instructor: Joonho Kwon

jhkwon@pusan.ac.kr

Data Science Lab @ PNU

data science laboratory

Slides are based on the textbook.

Outline

- 8.1 Cosequential operations
- 8.2 Application of the Model to a General Ledger Program
- **8.3 Extension of the Model to Include Multiway Merging**
- 8.4 A Second Look at Sorting in Memory
- 8.5 Merging as a Way of Sorting Large Files on Disk

- Skipped
  - 8.6 Sorting Files on Tape
  - 8.7 Sort-Merge Packages
  - 8.8 Sorting and Cosequential Processing in Unix

A K-way Merge Algorithm (1/3)

- K-way merge
  - A very general form of cosequential file processing
  - Merge K sorted input lists to create a single sorted output list
- Adapt 2-way merge algorithm
  - Instead of List1 and Lists2 keep an array of lists: List[1], List[2], ..., List[k]
  - Instead of item(1) and item(2) keep an array of items: item[1], item[2], ..., item[k]

A K-way Merge Algorithm (2/3)

- The synchronization step for 2 lists

```
if item(1) < item(2) then ...
if item(1) > item(2) then ...
if item(1) = item(2) then ...
```

- Modify

```
(1) minitem = index of miminum item in item[1],
          item[2], ..., item[k]
(2) output item[minitem] to output list
(3) for i=1 to K do
(4)      if item[i] = item[minitem] then
(5)          get next item from List[i]
```

  - If there are no repeated items among different lists, lines (3)-(5) can be simplified to:

```
get next item from List[minitem]
```

A K-way Merge Algorithm (3/3)

- C++/c style

```
// find an index of minimum item
int minItem = MinIndex(Item,k)
// Item(minItem) is the next output
ProcessItem(minItem);
for(i=0; i<k; i++)        // look at each list
    if( Item(minItem) == Item(i))    // advance list i
        MoreItems[i] = NextIemInList(i);
```

  - No repeated items

```
// find an index of minimum item
int minI = MinIndex(Item,k)
// Item(minItem) is the next output
ProcessItem(minI);
MoreItems[minI] = NextIemInList(minI);
```

Review: Ledger code

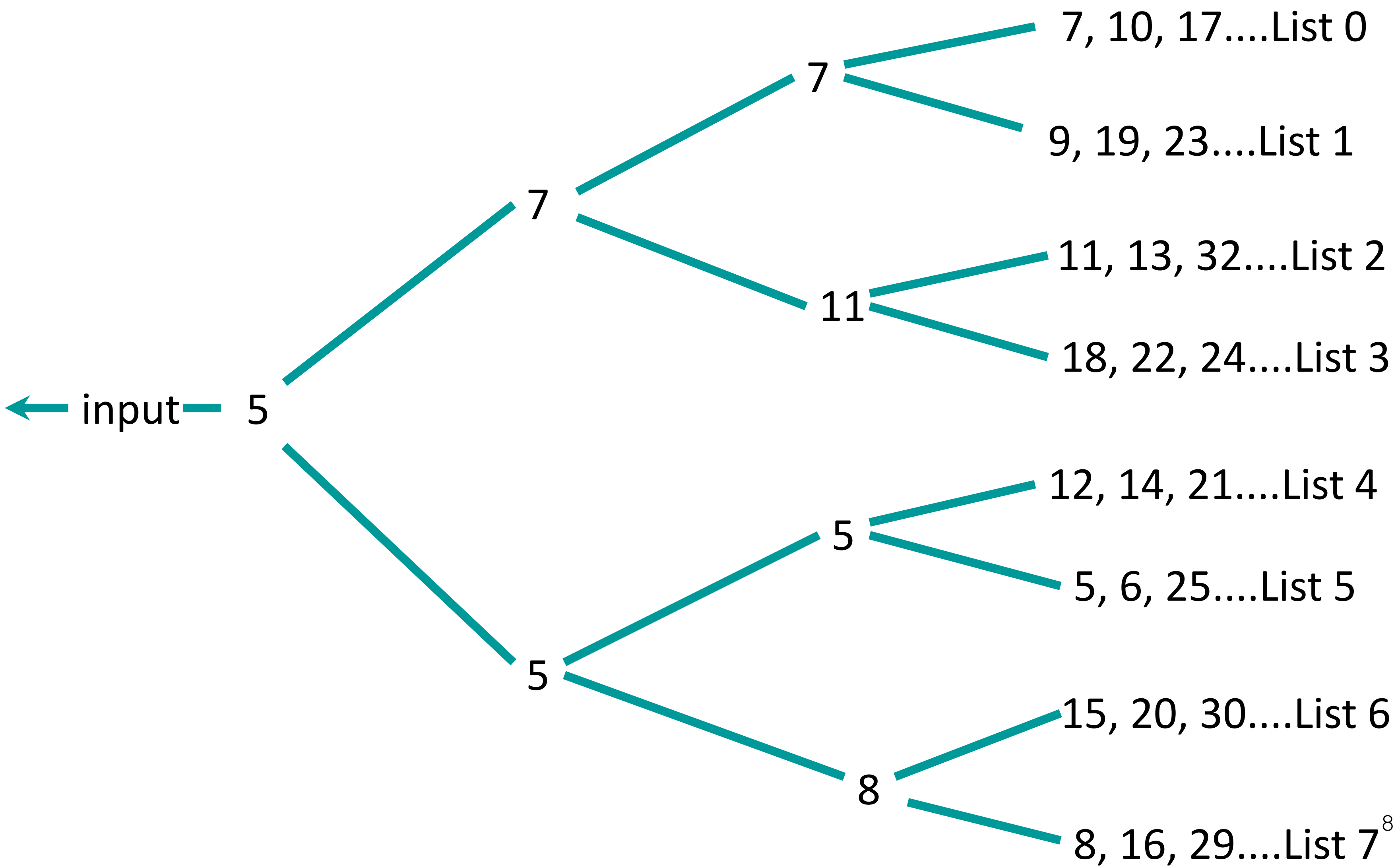
```
// return current item from this list
int LedgerProcess::Item (int ListNumber)
{   return AccountNumber[ListNumber];}

// process the item in this list when it first appears
int LedgerProcess::ProcessItem (int ListNumber)
{
    switch (ListNumber)
    {
        case 1: // process new ledger object
            ledger.PrintHeader(OutputList);
        case 2: // process journal file
            journal.PrintLine(OutputList);
    }
    return TRUE;
}
```

```
//get next item from this list
int LedgerProcess::NextItemInList (int ListNumber)
{
    switch (ListNumber)
    {
    case 1: return NextItemInLedger ();
    case 2: return NextItemInJournal ();
    }
    return FALSE;
}
```

Selection Tree for Merging Large Number of Lists

- K-way merge
  - nice if K is no larger than 8 or so
  - if K > 8, the set of comparisons for minimum key is expensive
  - loop of comparison (computing)
- Selection Tree (if K > 8)
  - time vs. space trade off
  - a kind of “tournament” tree
  - the minimum value is at root node
  - the depth of tree is $log_2 K$

Selection Tree



CosequentialProcess class

- A single, simple model that can be the basis for the construction of any kind of consequential process
  - supports processing of any type of list
  - Includes operations to match and merge lists
  - Defines the list processing operations required for cosequential processing as virtual methods

Outline

- 8.1 Cosequential operations
- 8.2 Application of the Model to a General Ledger Program
- 8.3 Extension of the Model to Include Multiway Merging
- **8.4 A Second Look at Sorting in Memory**
- 8.5 Merging as a Way of Sorting Large Files on Disk

- Skipped
  - 8.6 Sorting Files on Tape
  - 8.7 Sort-Merge Packages
  - 8.8 Sorting and Cosequential Processing in Unix

A Second Look at Sorting in Memory

- Read the whole file from into memory, perform sorting, write the whole file into disk
- Can we improve on the time that it takes for this RAM sort?
  - perform some of parts in parallel
  - selection sort is good but cannot be used to sort entire file
- Using Heap technique!
  - processing and I/O can occur in parallel
  - keep all the keys in heap
- Heap building while reading a block
- Heap rebuilding while writing a block

Overlapping processing and I/O

- Heap
  - a kind of binary tree, complete binary tree
  - each node has a single key, that key is less than or equal to the key at its parent node
  - storage for tree can be allocated sequentially
  - so there is no need for pointers or other dynamic overhead for maintaining the heap
  - Details: Skipped

Outline

- 8.1 Cosequential operations
- 8.2 Application of the Model to a General Ledger Program
- 8.3 Extension of the Model to Include Multiway Merging
- 8.4 A Second Look at Sorting in Memory
- **8.5 Merging as a Way of Sorting Large Files on Disk**
  - **External Merge from Database book**

- Skipped
  - 8.6 Sorting Files on Tape
  - 8.7 Sort-Merge Packages
  - 8.8 Sorting and Cosequential Processing in Unix

Challenge

- Merging Big Files with Small Memory

How do we *efficiently* merge two sorted files when both are much larger than our main memory buffer?

External Merge Algorithm

- **Input**: 2 **sorted** lists of length M and N
- **Output:** 1 sorted list of length M + N
- **Required:** At least 3 Buffer Pages
- **IOs**: 2(M+N)

Key (Simple) Idea

- To find an element that is no larger than all elements in two lists, one only needs to compare minimum elements from each list.

If:

$$A_1 \leq A_2 \leq \cdots \leq A_N$$
$$B_1 \leq B_2 \leq \cdots \leq B_M$$

Then:

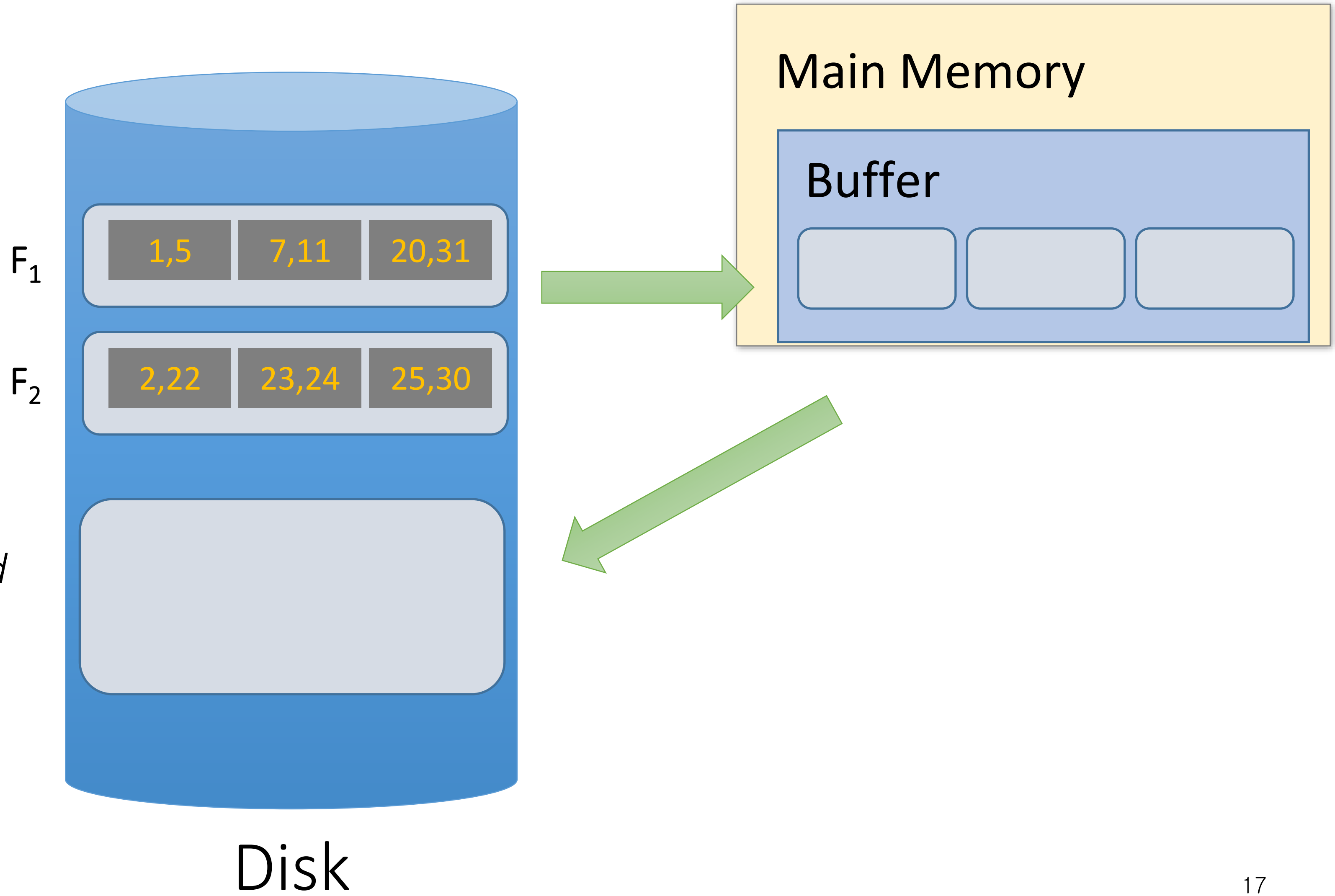
$$Min(A_1, B_1) \leq A_i$$
$$Min(A_1, B_1) \leq B_j$$

for i=1….N and j=1….M

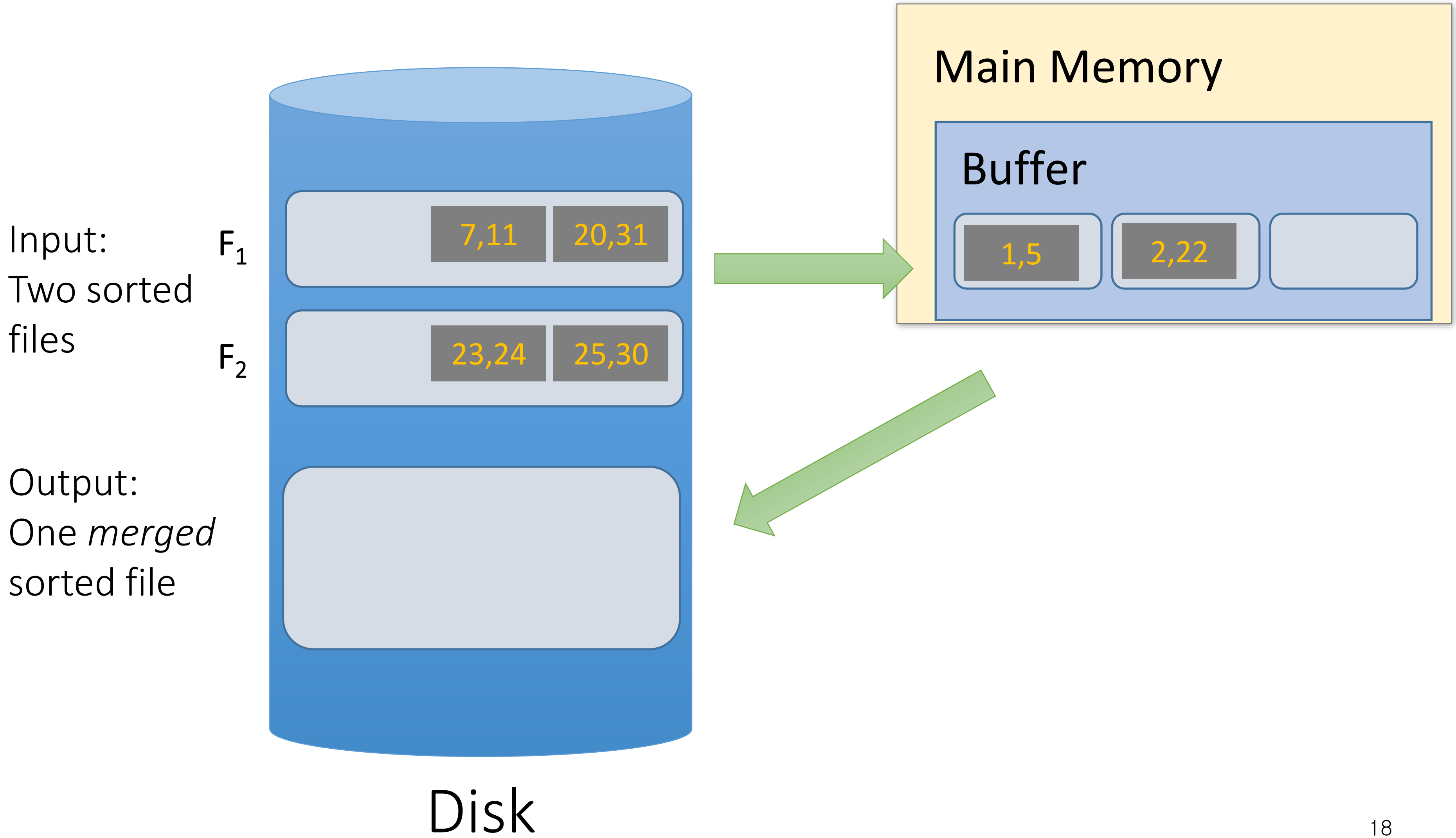
Review: External Merge Algorithm (1/10)

Input:
Two sorted files

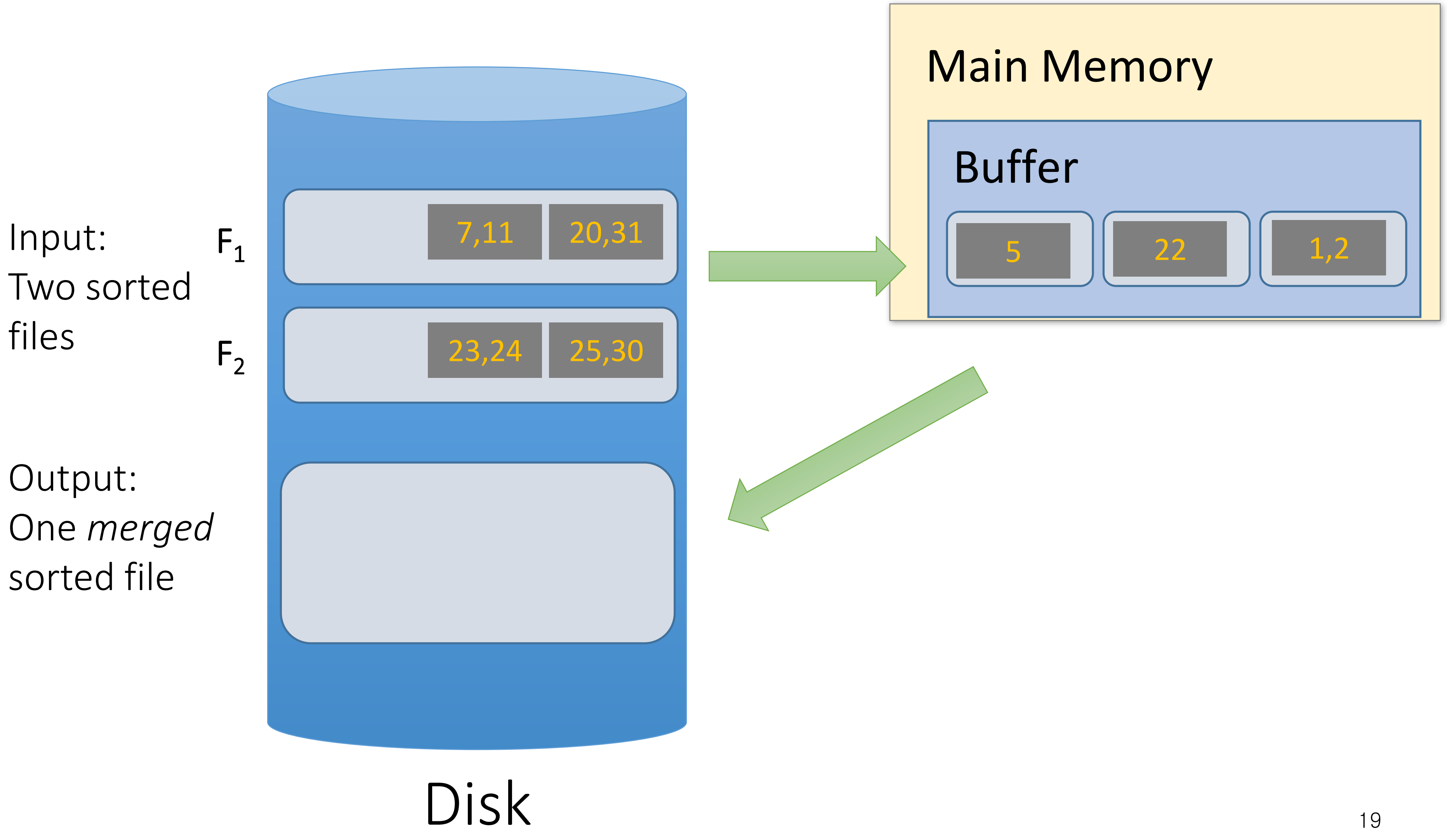
Output:
One *merged* sorted file



Review: External Merge Algorithm (2/10)



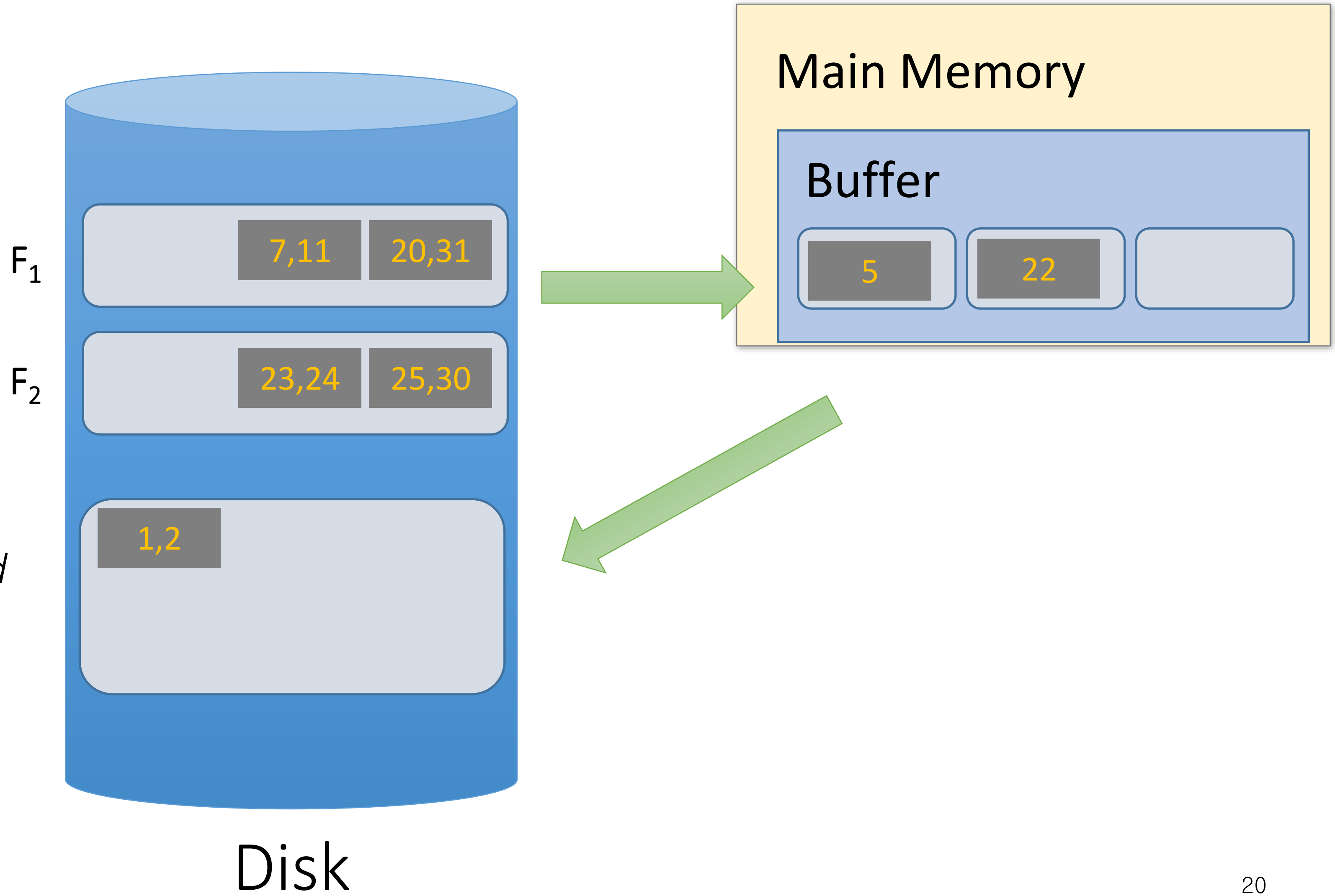
Review: External Merge Algorithm (3/10)



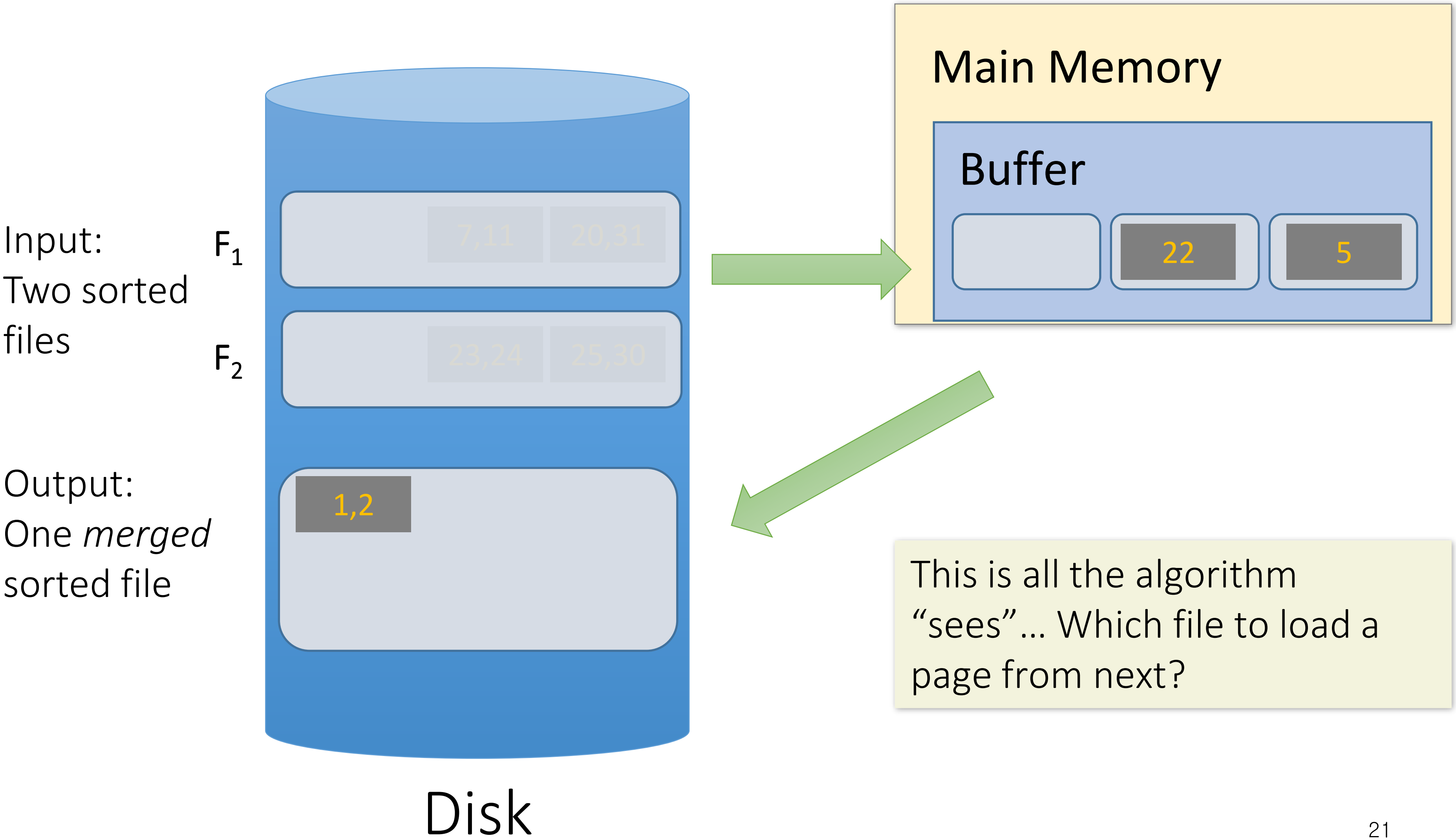
Review: External Merge Algorithm (4/10)

Input:
Two sorted files

Output:
One *merged* sorted file



Review: External Merge Algorithm (5/10)

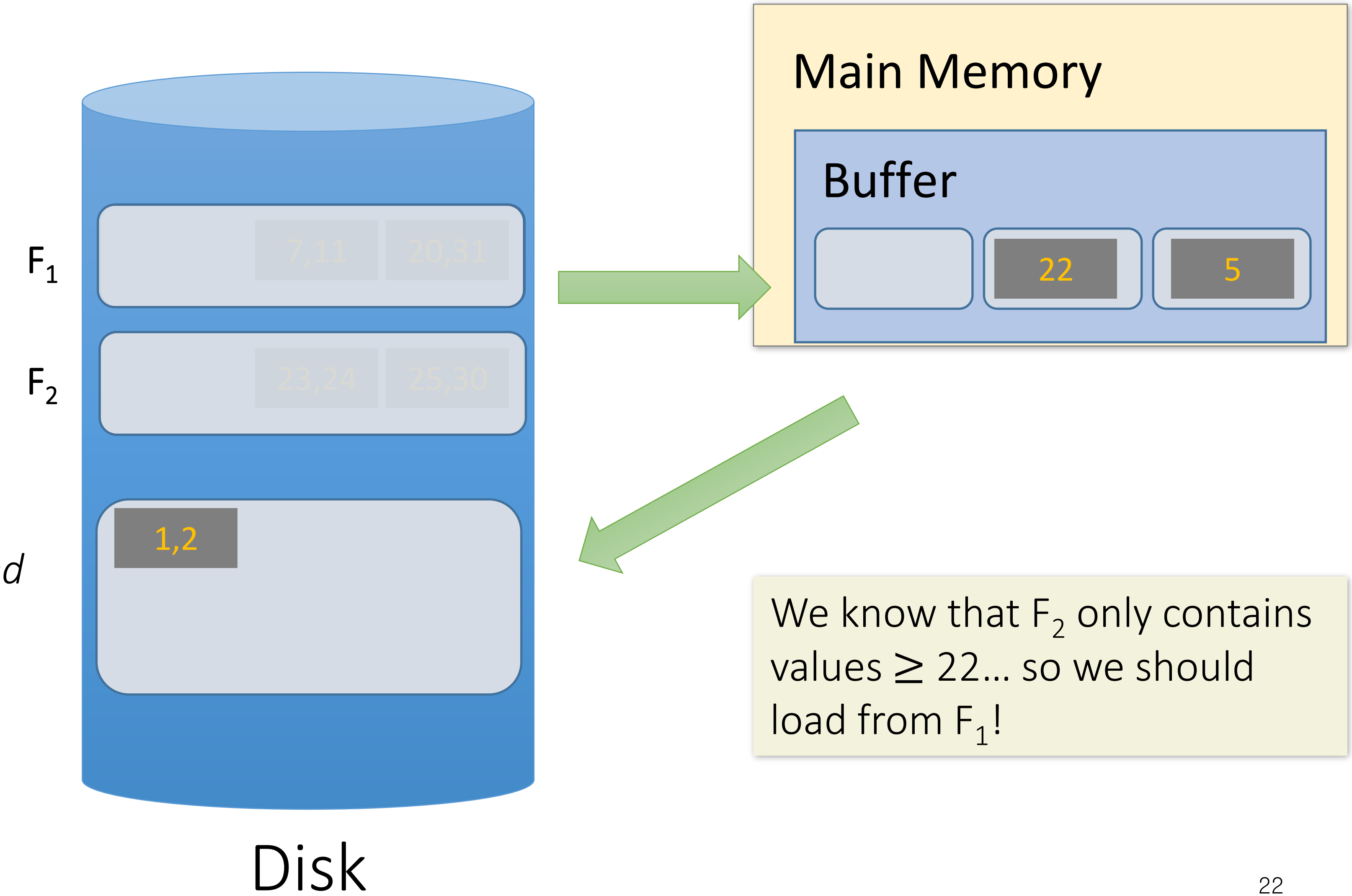


This is all the algorithm "sees"… Which file to load a page from next?

Review: External Merge Algorithm (6/10)

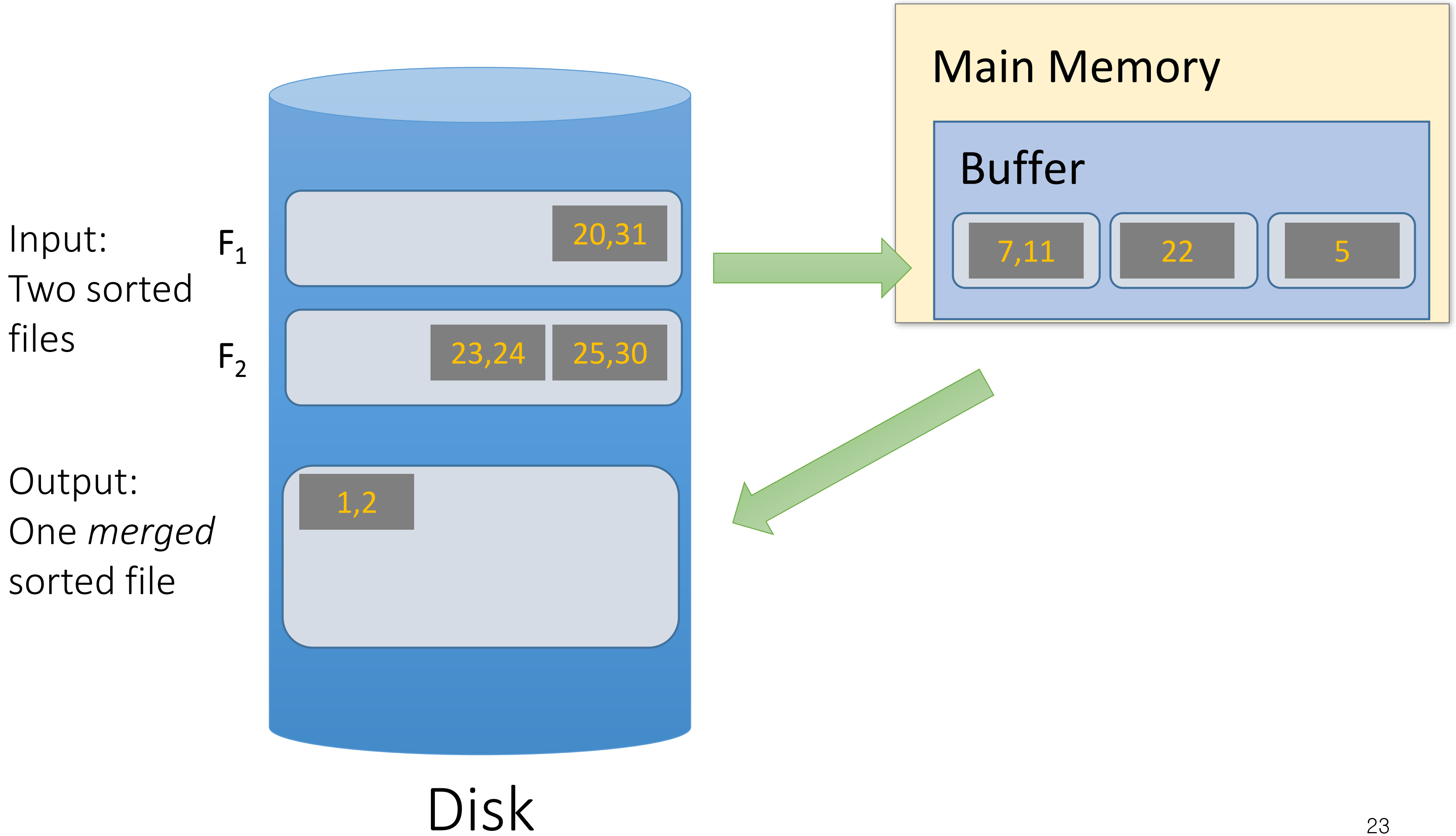
Input:
Two sorted files

Output:
One *merged* sorted file

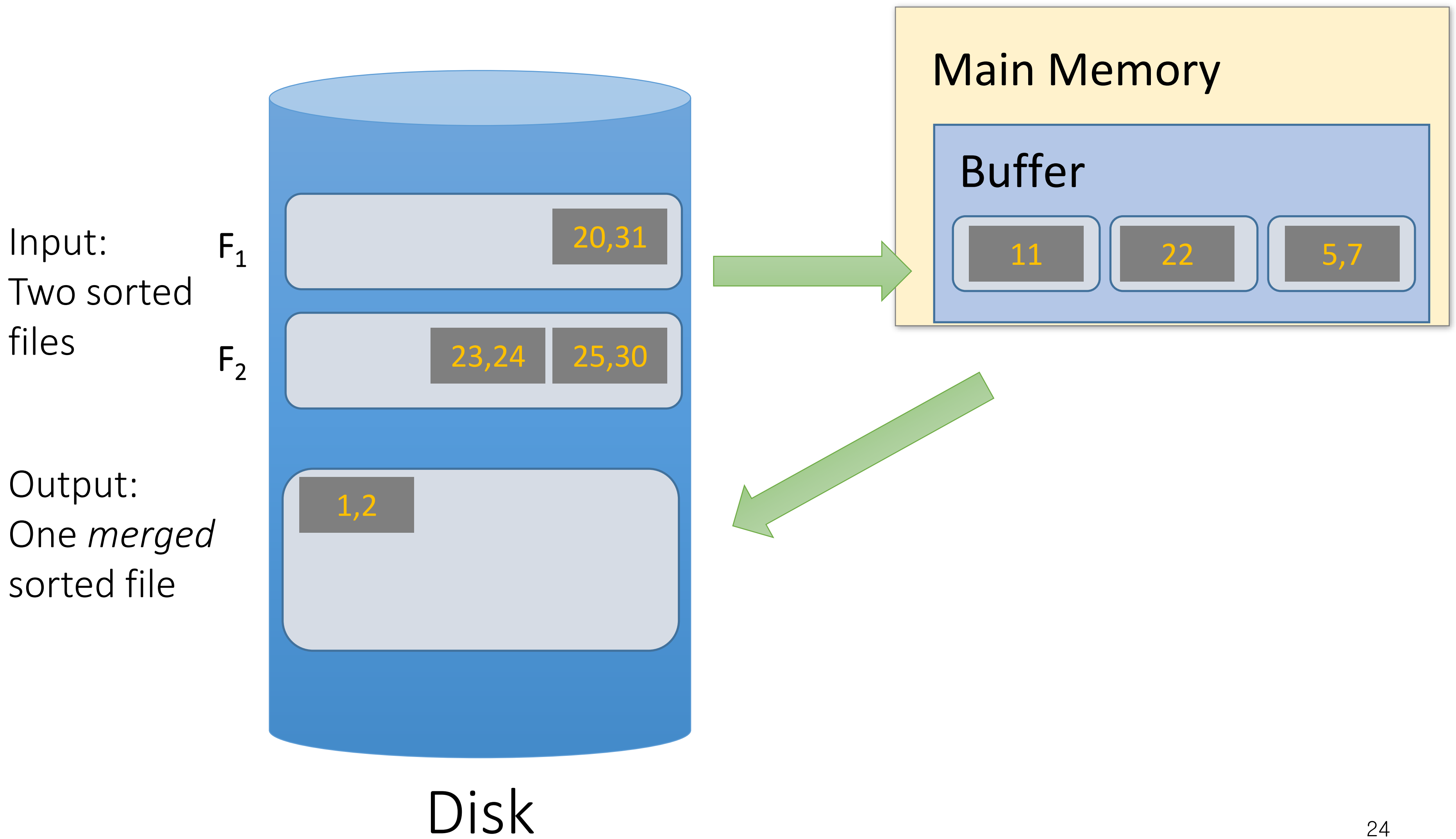


We know that F_2 only contains values $\geq$ 22… so we should load from F_1!

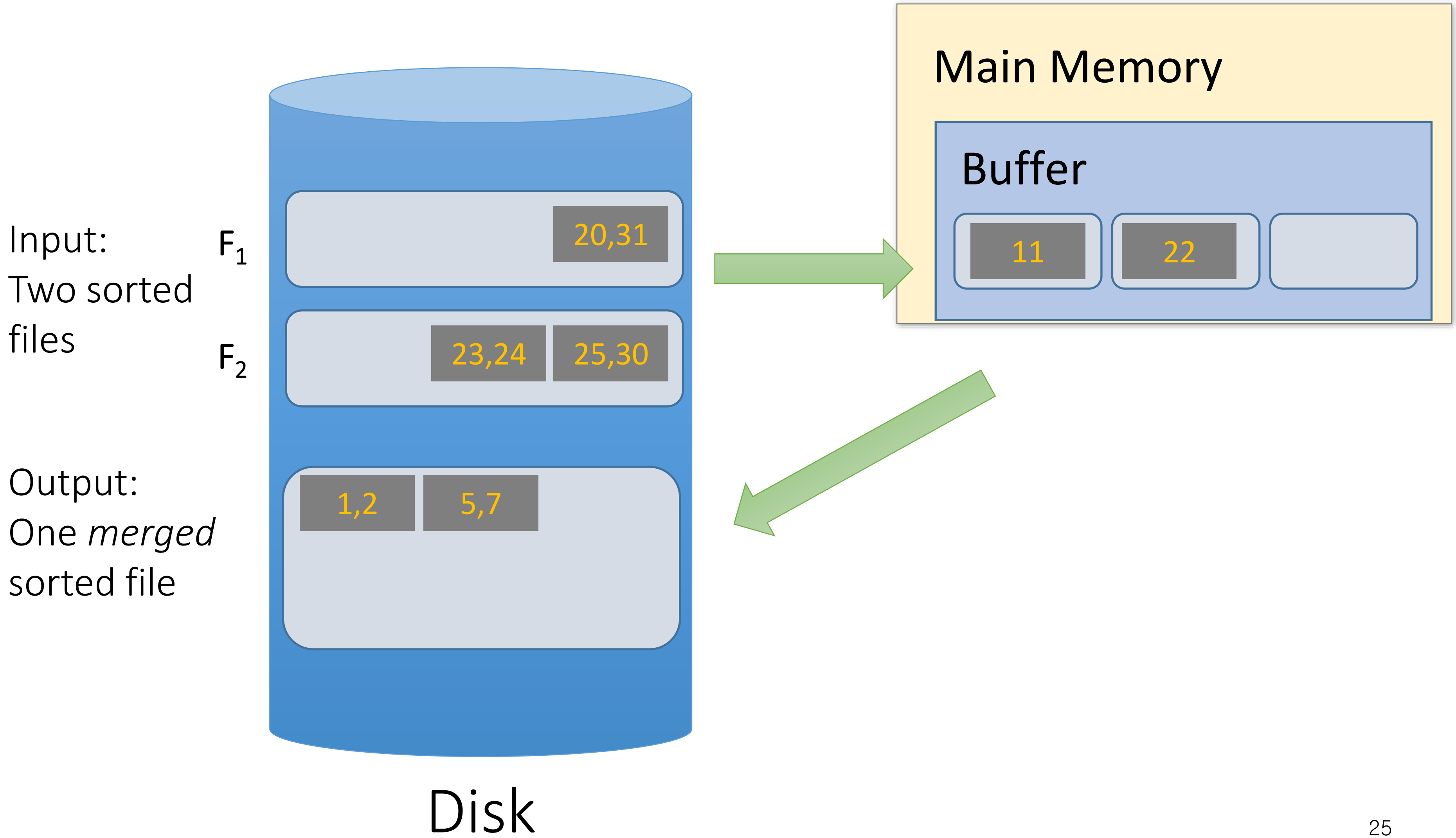
Review: External Merge Algorithm (7/10)



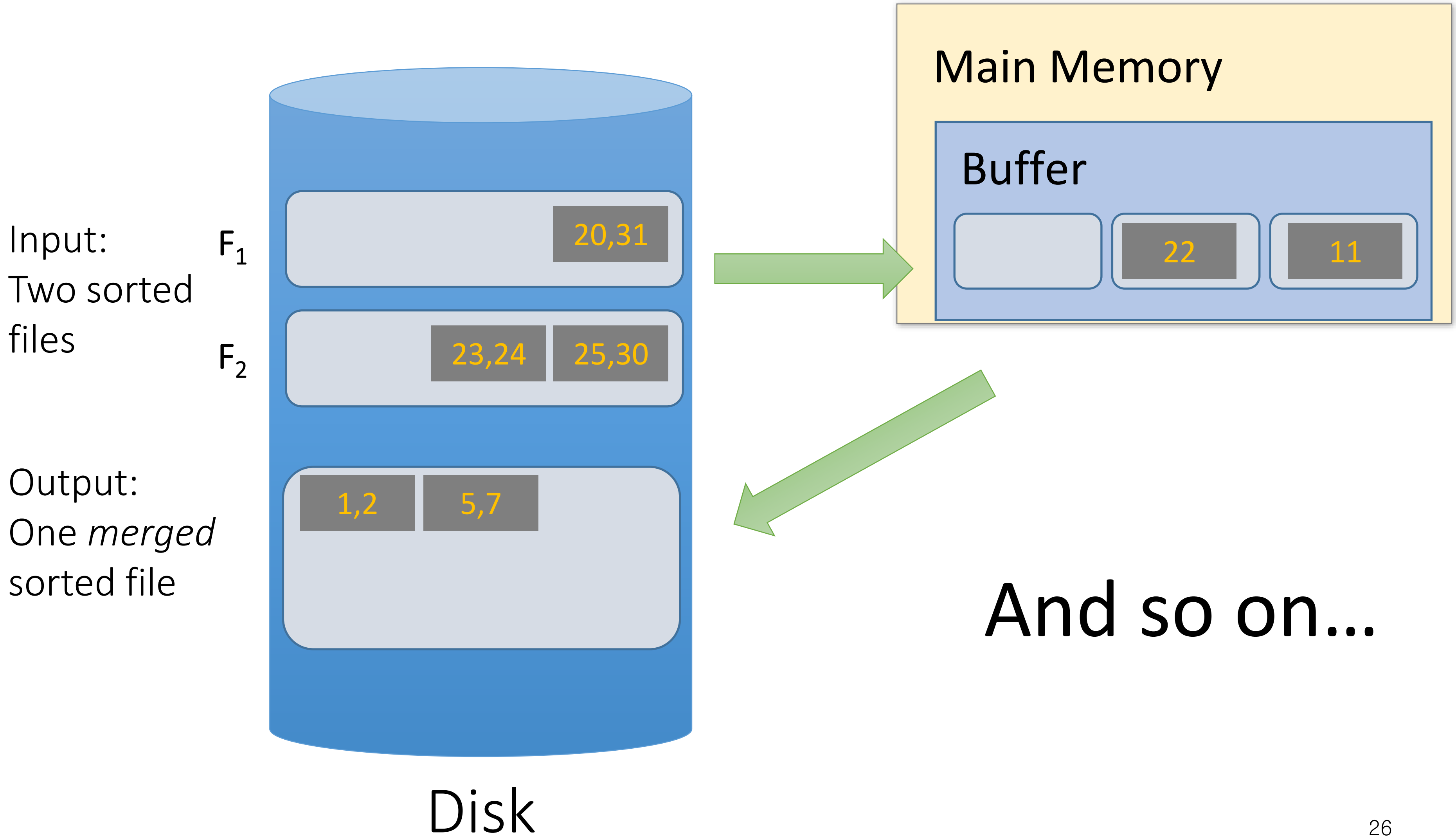
Review: External Merge Algorithm (8/10)



Review: External Merge Algorithm (9/10)



Review: External Merge Algorithm (10/10)



And so on…

Summary of external merging

- We can merge 2 lists of **arbitrary length** with *only* 3 buffer pages.

If lists of size M and N, then
Cost: 2(M+N) IOs
Each page is read once, written once

With B+1 buffer pages, can merge B lists. How?

Outline

- 8.1 Cosequential operations
- 8.2 Application of the Model to a General Ledger Program
- 8.3 Extension of the Model to Include Multiway Merging
- 8.4 A Second Look at Sorting in Memory
- **8.5 Merging as a Way of Sorting Large Files on Disk**
  - **External Merge Sort from Database book**

- Skipped
  - 8.6 Sorting Files on Tape
  - 8.7 Sort-Merge Packages
  - 8.8 Sorting and Cosequential Processing in Unix

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

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…

- *Sort-merge* join algorithm involves sorting

Coming up…

So how do we sort big files?

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

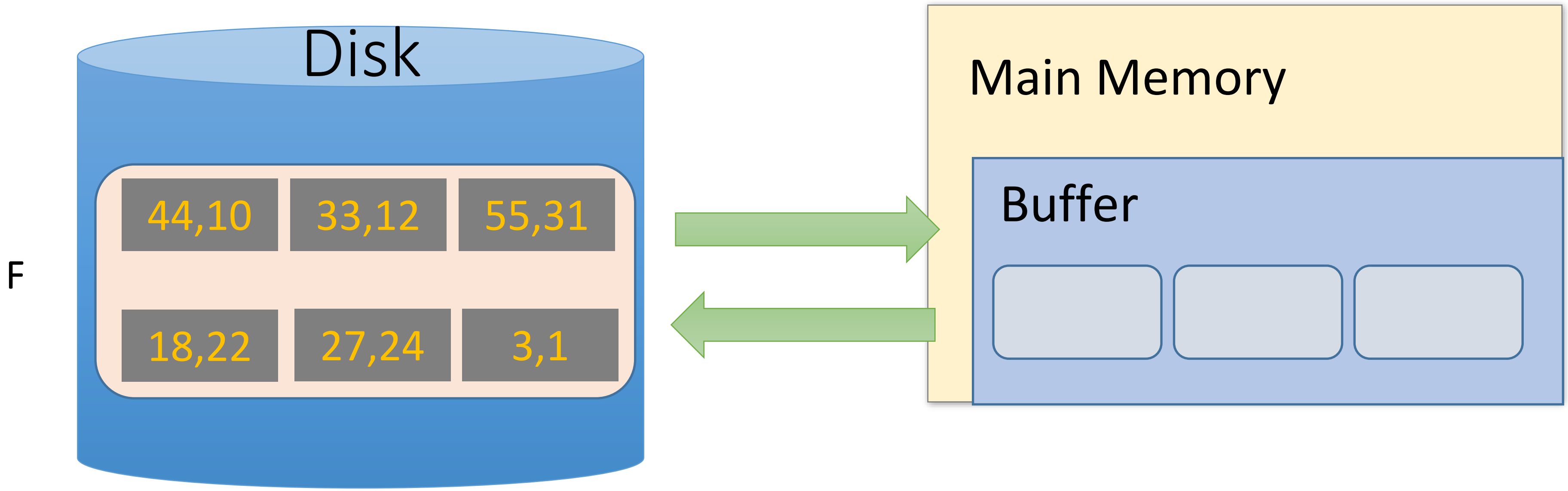
2. **Merge** pairs (or groups) of runs ***using the external merge algorithm***

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

External Merge Sort Algorithm (2-way sort) (1/6)

- Example: 3 Buffer pages, 6-page file

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

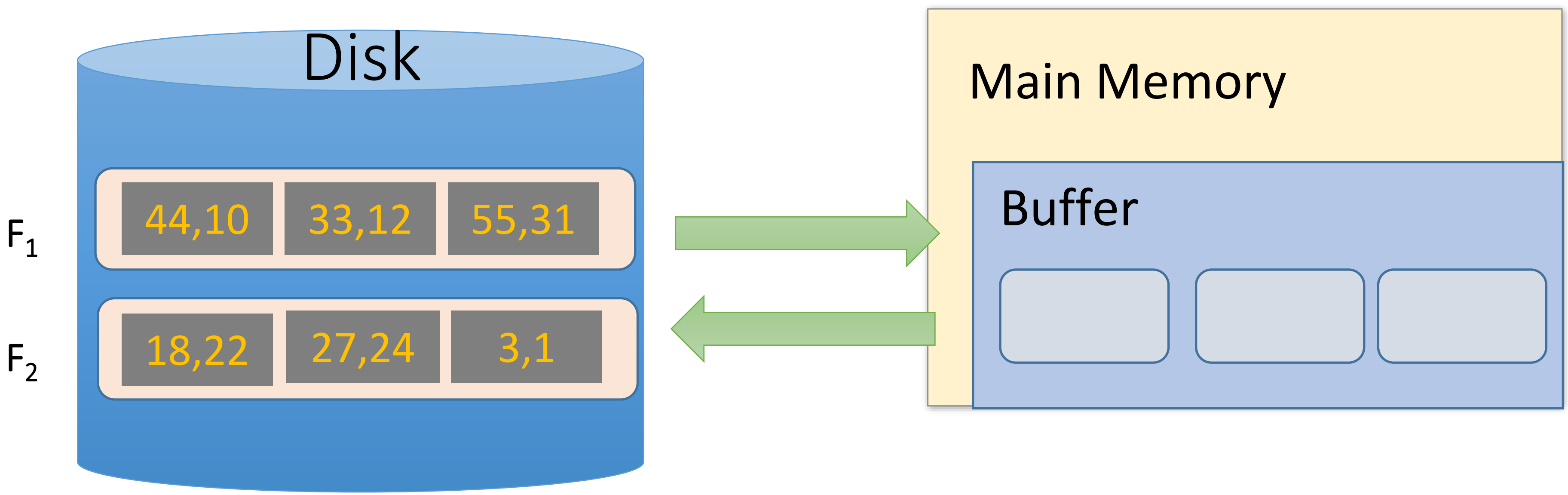


Orange file
= unsorted

External Merge Sort Algorithm (2-way sort) (2/6)

- Example: 3 Buffer pages, 6-page file

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

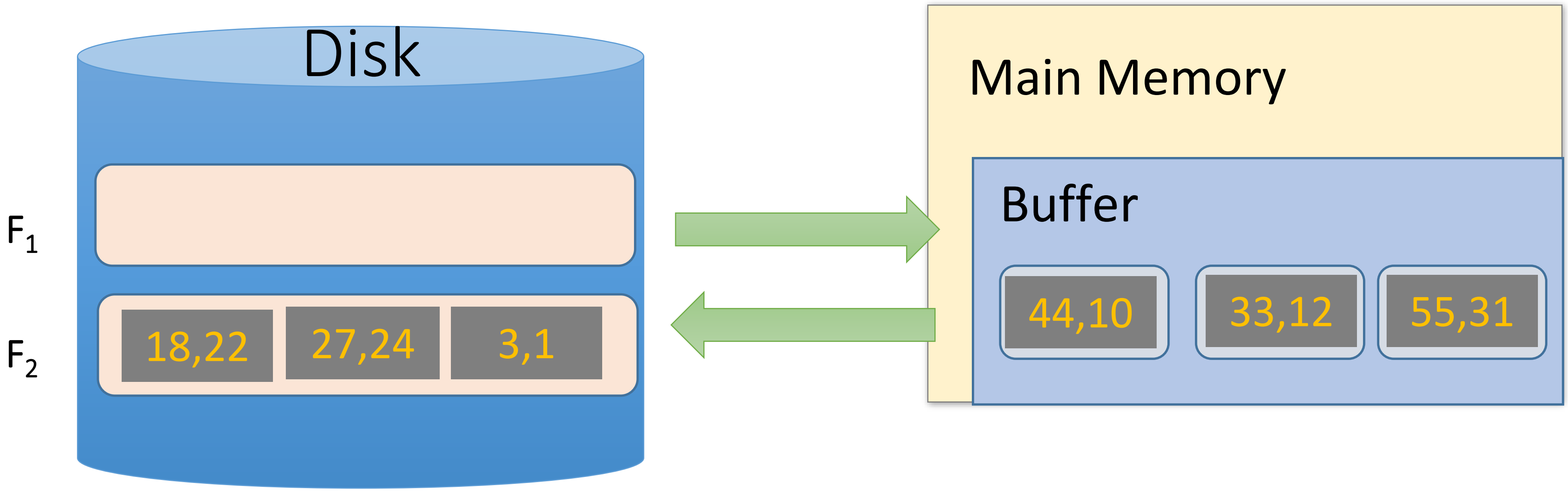


Orange file
= unsorted

External Merge Sort Algorithm (2-way sort) (3/6)

- Example: 3 Buffer pages, 6-page file

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

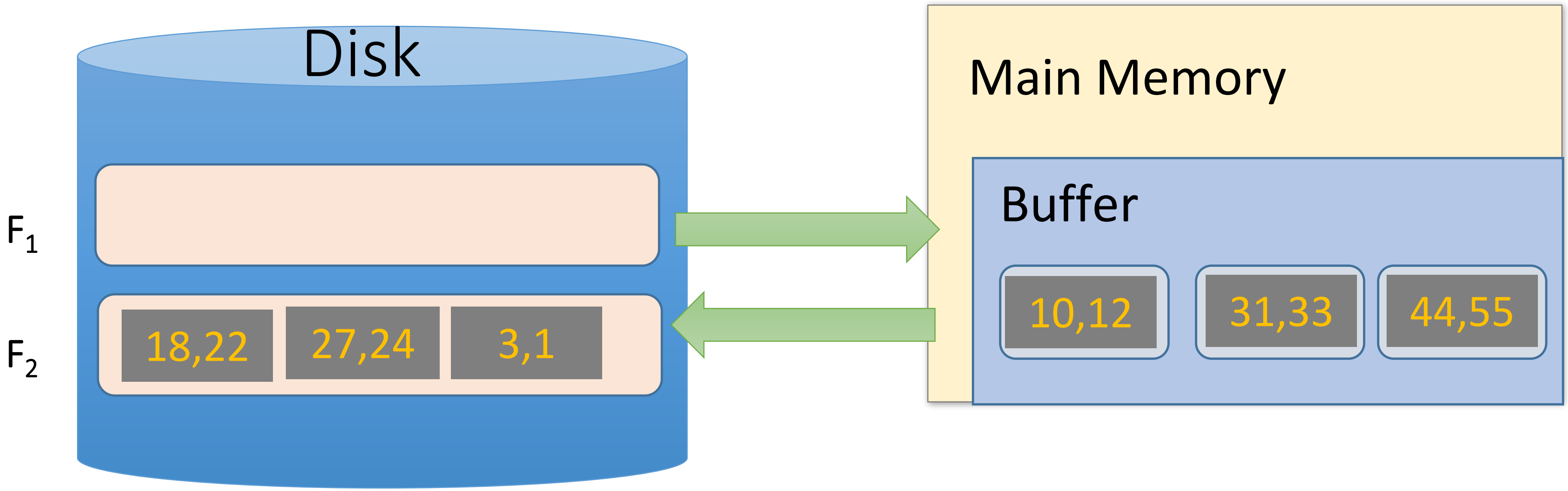


Orange file
= unsorted

External Merge Sort Algorithm (2-way sort) (4/6)

- Example: 3 Buffer pages, 6-page file

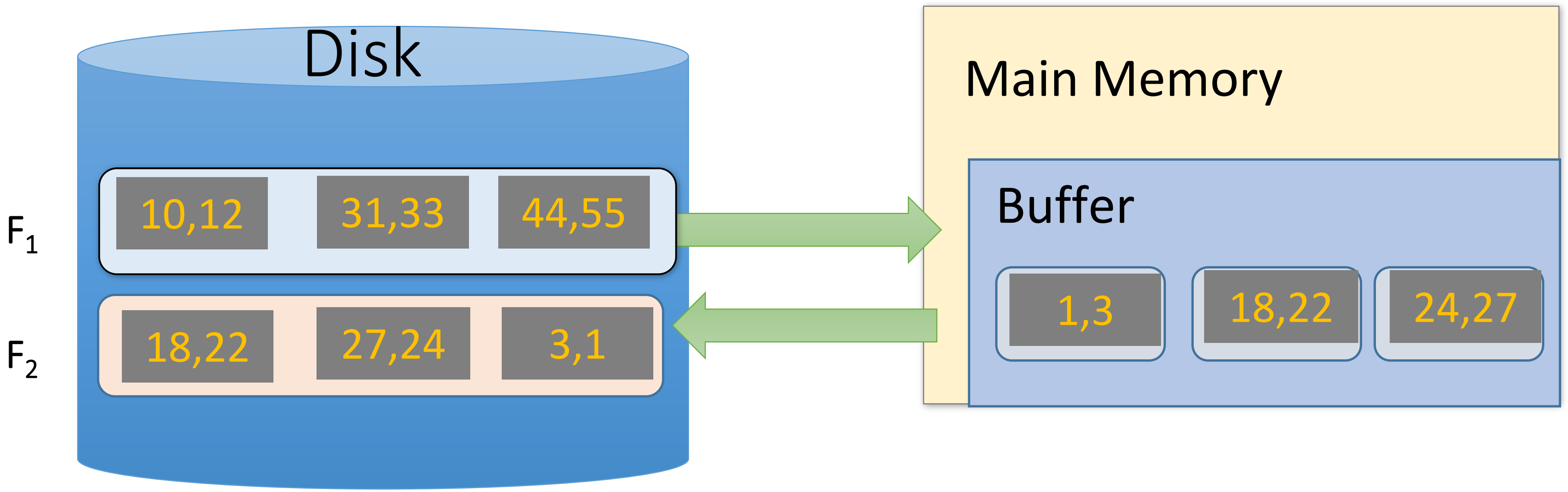
1. Split into chunks small enough to **sort in memory**



External Merge Sort Algorithm (2-way sort) (5/6)

- Example: 3 Buffer pages, 6-page file

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

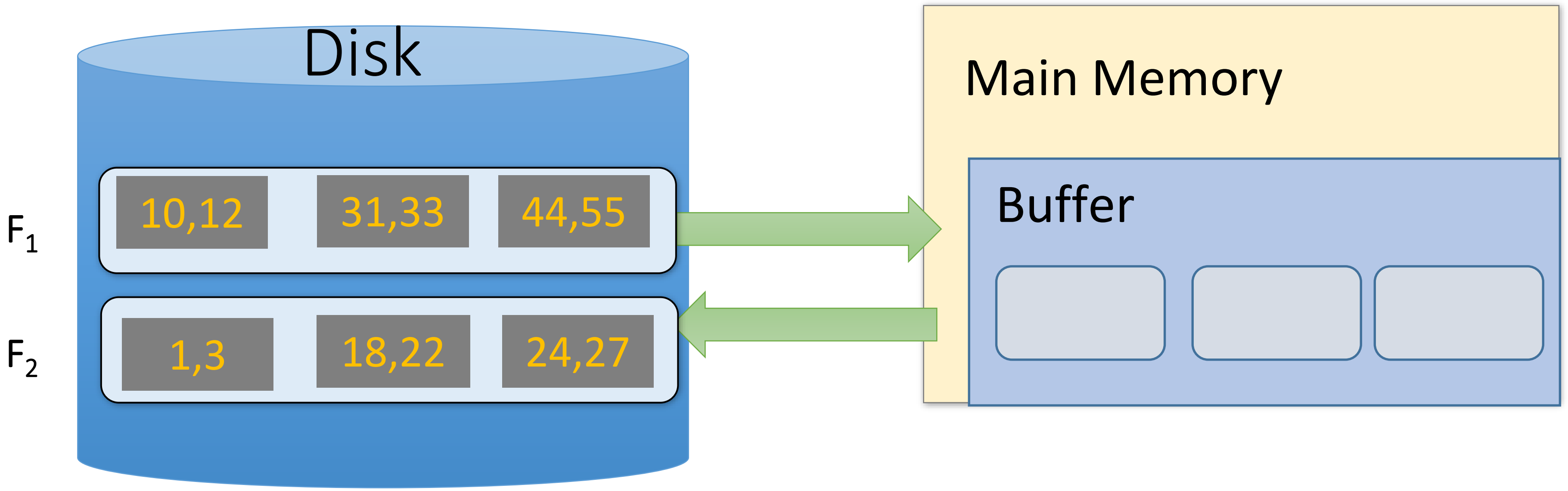


And similarly for F$_2$

Each sorted file is a called a *run*

External Merge Sort Algorithm (2-way sort) (6/6)

- Example: 3 Buffer pages, 6-page file



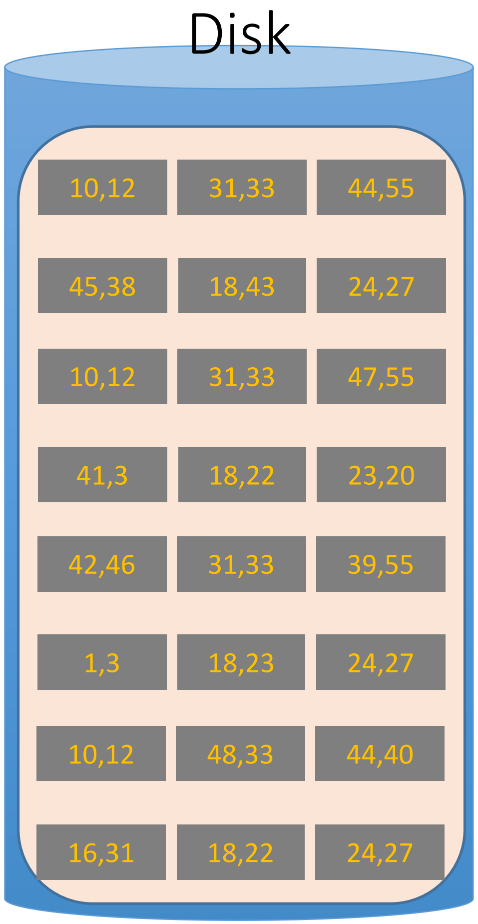
2. Now just run the **external merge** algorithm & we're done!

Calculating IO Cost

- For 3 buffer pages, 6 page file:
  - Split into **two 3-page** files and **sort in memory**
    - = **1 R + 1 W** for each file = 2*(3 + 3) = 12 IO operations
  - Merge each pair of sorted chunks using the external merge algorithm
    - = 2*(3 + 3) = 12 IO operations
  - **Total cost = 24 IO**

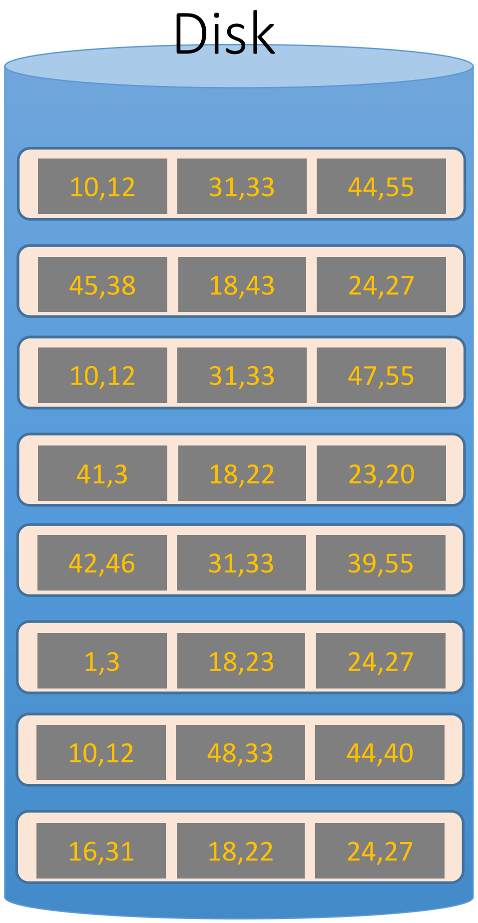
Running External Merge Sort on Larger Files (1/6)

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



Running External Merge Sort on Larger Files (2/6)

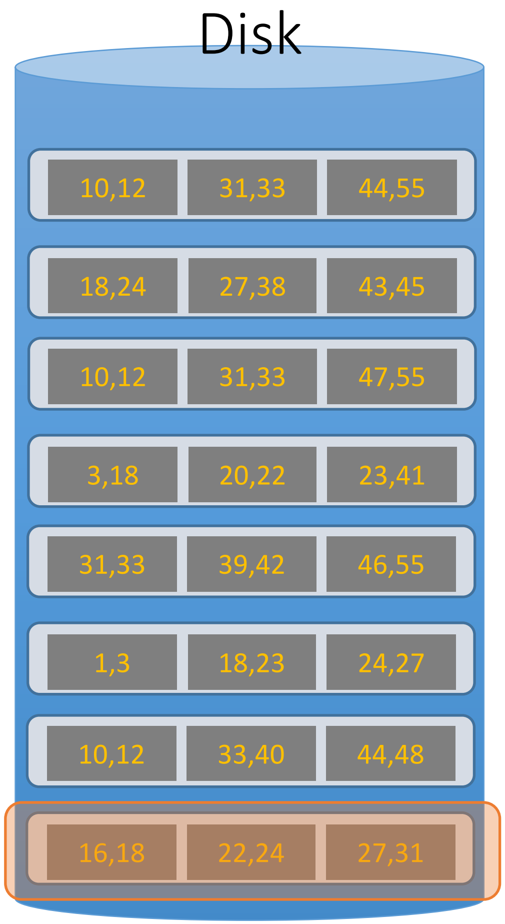
- Assume we still only have *3* buffer pages *(Buffer not pictured)*



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

Running External Merge Sort on Larger Files (3/6)

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

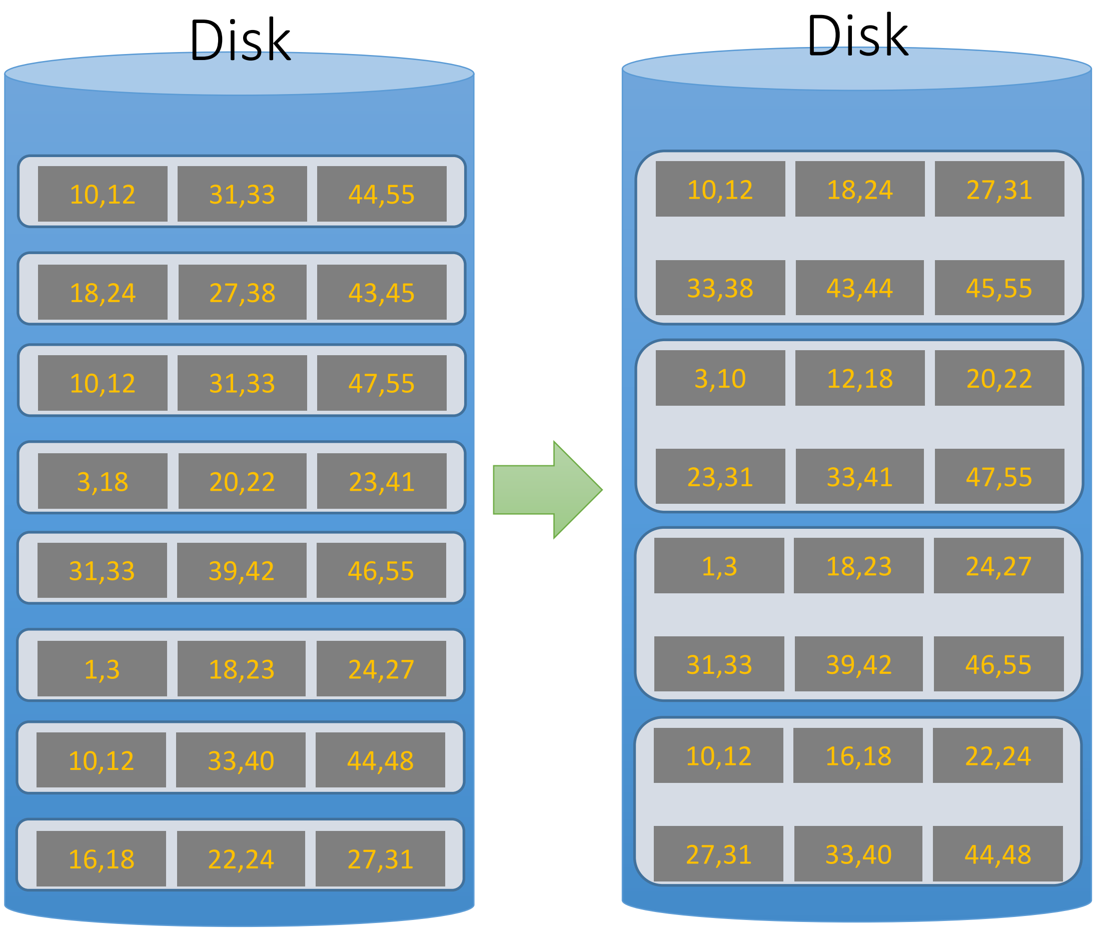


1. Split into files small enough to sort in buffer… and sort

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

Running External Merge Sort on Larger Files (4/6)

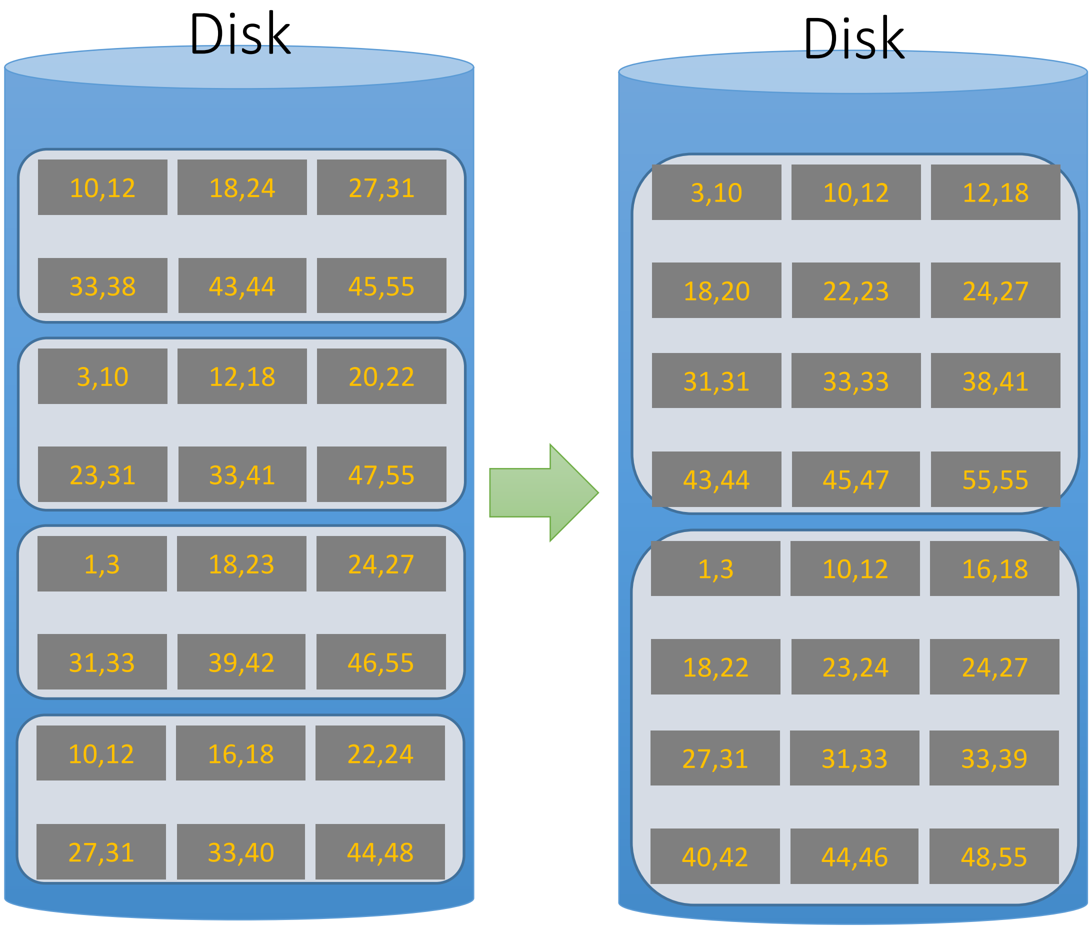
- Assume we still only have *3* buffer pages *(Buffer not pictured)*



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

Running External Merge Sort on Larger Files (5/6)

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

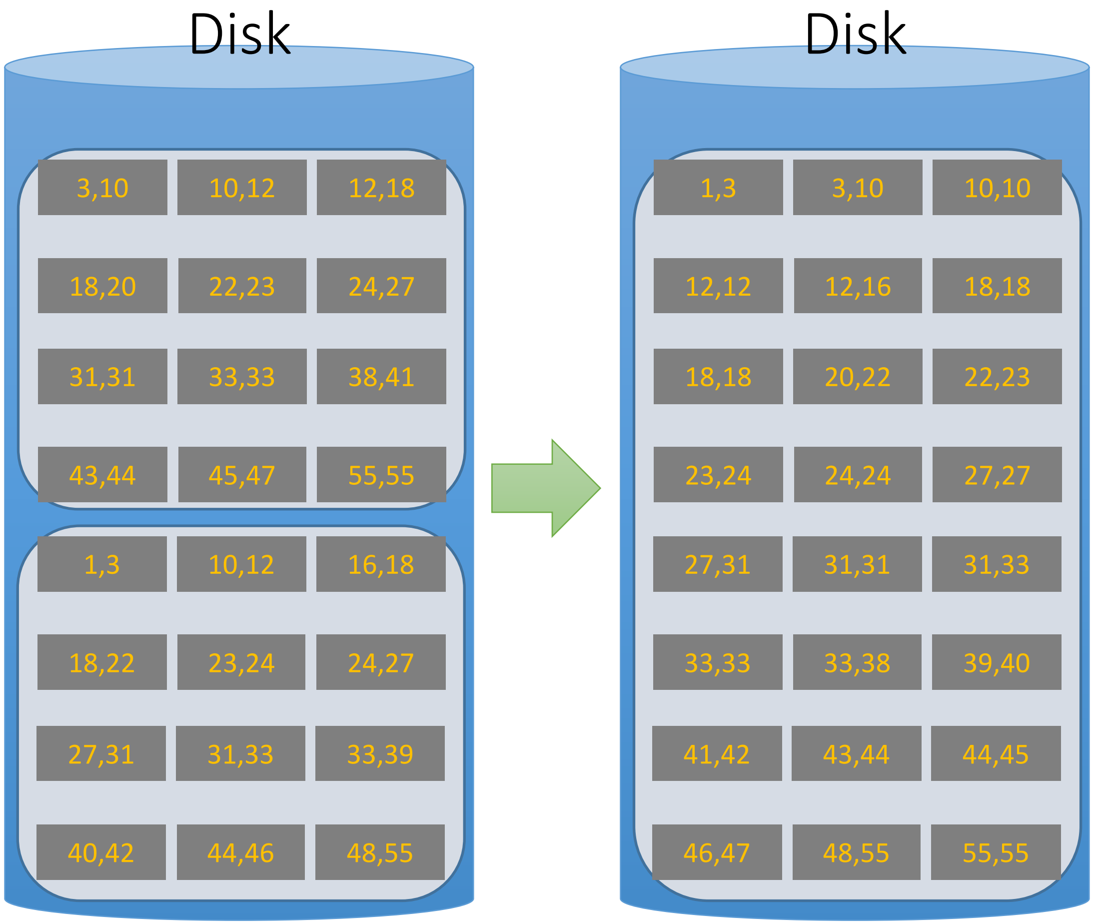


3. And repeat…

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

Running External Merge Sort on Larger Files (6/6)

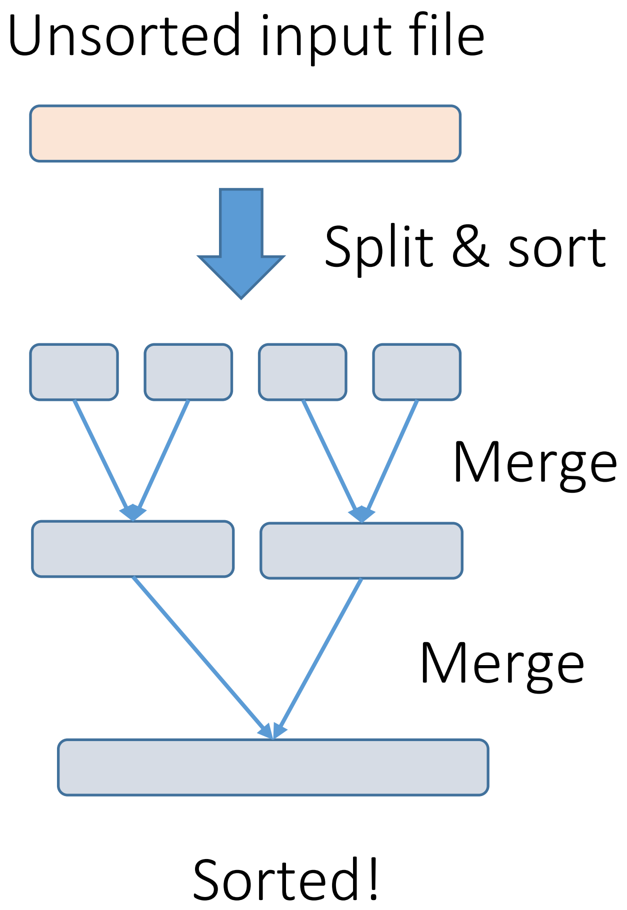
- Assume we still only have *3* buffer pages *(Buffer not pictured)*



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:
  - First pass: Merge N/2 pairs of runs each of length 1 page
  - Second pass: Merge N/4 pairs of runs each of length 2 pages
  - In general, for N pages, we do $\lceil log_2 N \rceil$ passes
    - +1 for the initial split & sort
  - Each pass involves reading in & writing out all the pages = $2N$ IO



→ **2N*($\lceil \boldsymbol{log_2 N} \rceil$+1)** total IO cost!

Using B+1 buffer pages to reduce # of passes (1/2)

- 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 (2/2)

- 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)!

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

Starting with runs of length 1

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

Starting with runs of length *B+1*

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

Performing ***B***-way merges

Q&A