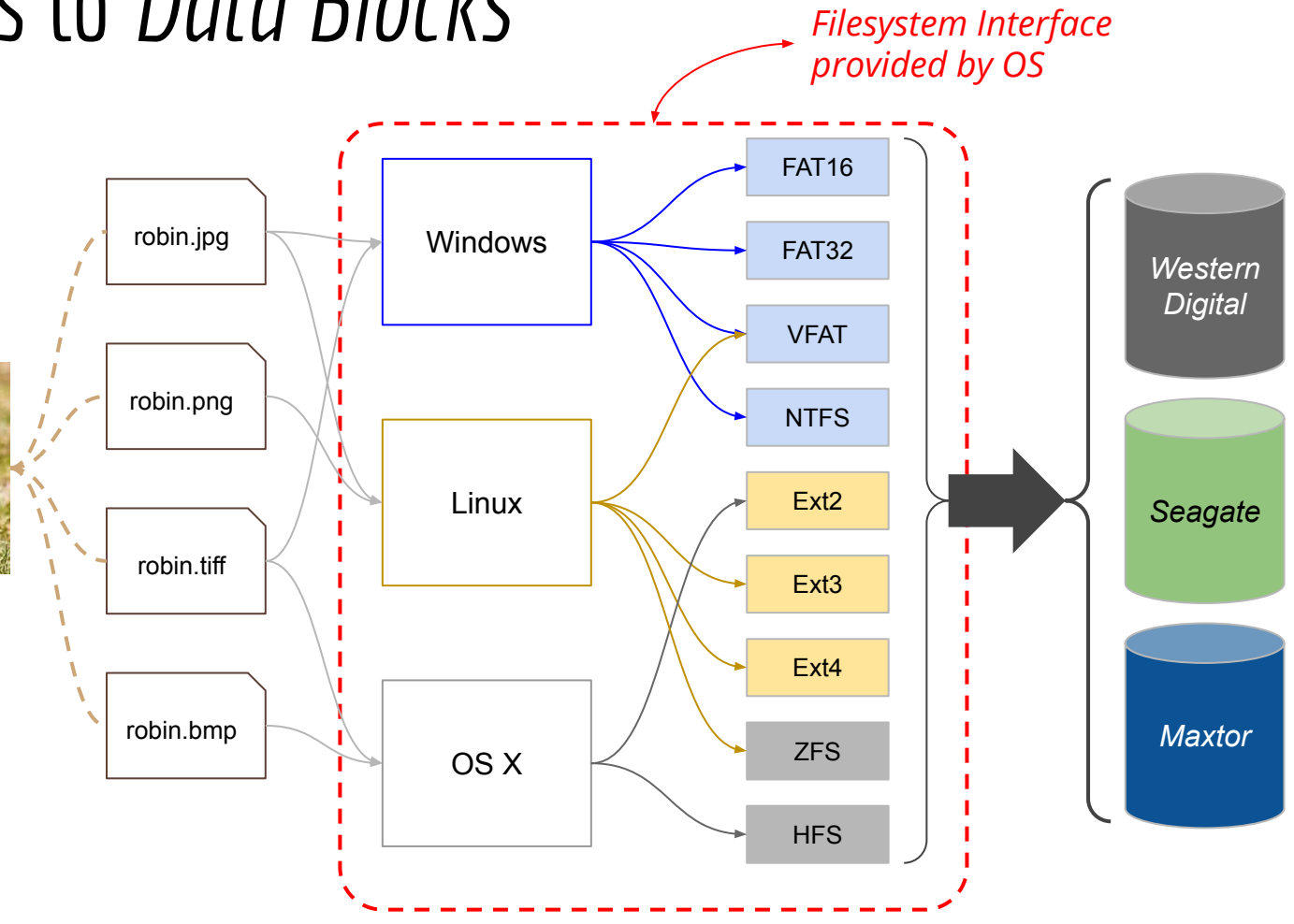
Filesystem Interface

Popular File Systems

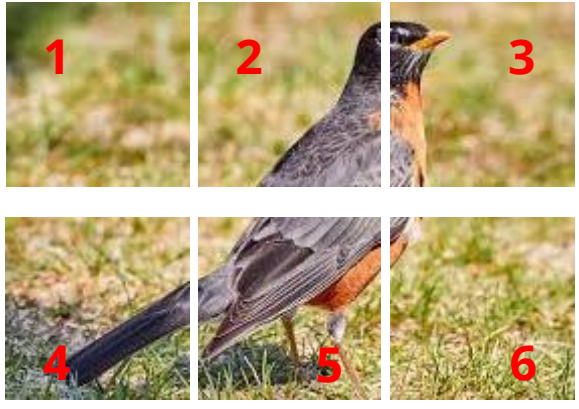
- DOS/Windows
  - FAT (FAT12, FAT16, FAT32, VFAT, NTFS
- Linux
  - Ext, Ext2, Ext3, Ext4,
- Mac OS X
  - HFS, HFS+
- And many more
  - ZFS, BTRFS
  - IPFS (Inter-Planetary File System 2015, Stable Release Aug 2019)

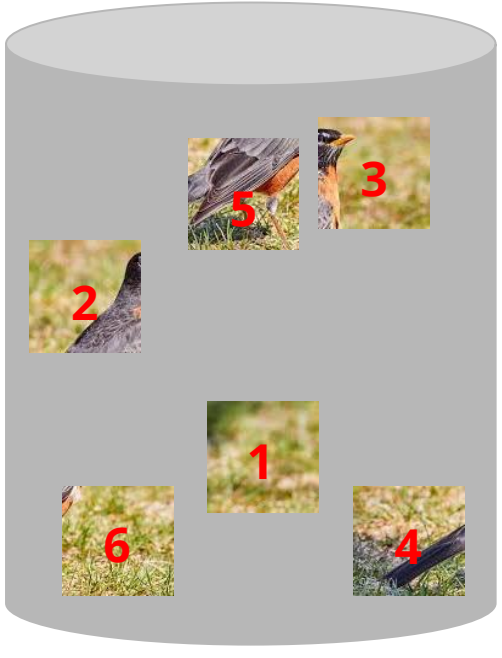
Why do we need these filesystems?

User Files to *Data Blocks*



Logical vs. Physical Representation





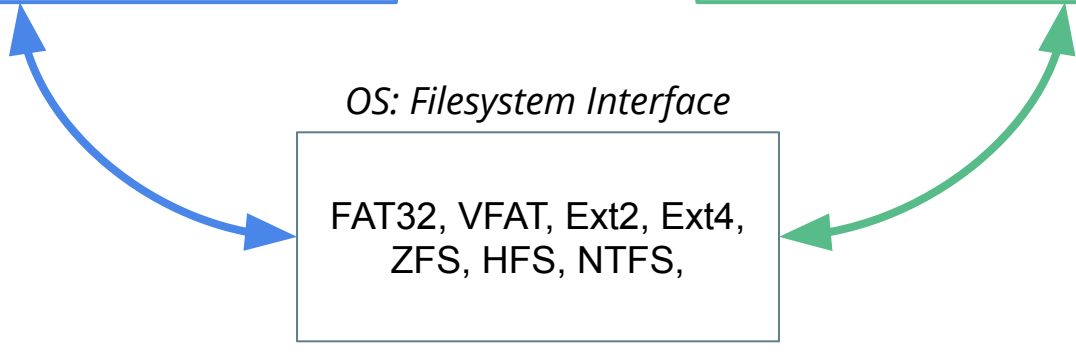
Fixed size disk blocks

Analogy:
contiguous **logical** address space **vs.** non-contiguous arrangement in **physical** RAM

(User) Files vs. (Storage) Devices

- Records, database, ...
- File formats
- Binary vs. text files
- JPG, PNG, TIFF, ZIP, ...
- C++ source, ELF executable, ...

- Block of bytes
- Track Number, Sector Number,
- Error Checking/Correction
- Parity Block
- DVD, IDE Drive, USB, CD-ROM, ...



Cloud Storage (Remote Devices)

Techniques in *memory management* are also applicable to *filesystems*

Memory Manager

- **transient** data in memory
- Processes access the physical RAM without OS intervention

Filesystem Manager

- **persistent** data on “disk”
- Process accesses the storage device via system calls

File services expected from the OS?

Functionalities / Services of a File System

- Assembly instructions related to I/O operations are typically **privilege** instructions
- Basic Unix system calls (CRUD operations)
  - File operations: `open()`, `close()`, `read()`, `write()`, `seek()`, `unlink()`
  - Directory operations: `opendir()`, `readdir()`, `...`
- Persistent Storage
  - With(out) encryption
- Access Control: Sharing & Protection
- Data Recovery

What does the OS do?

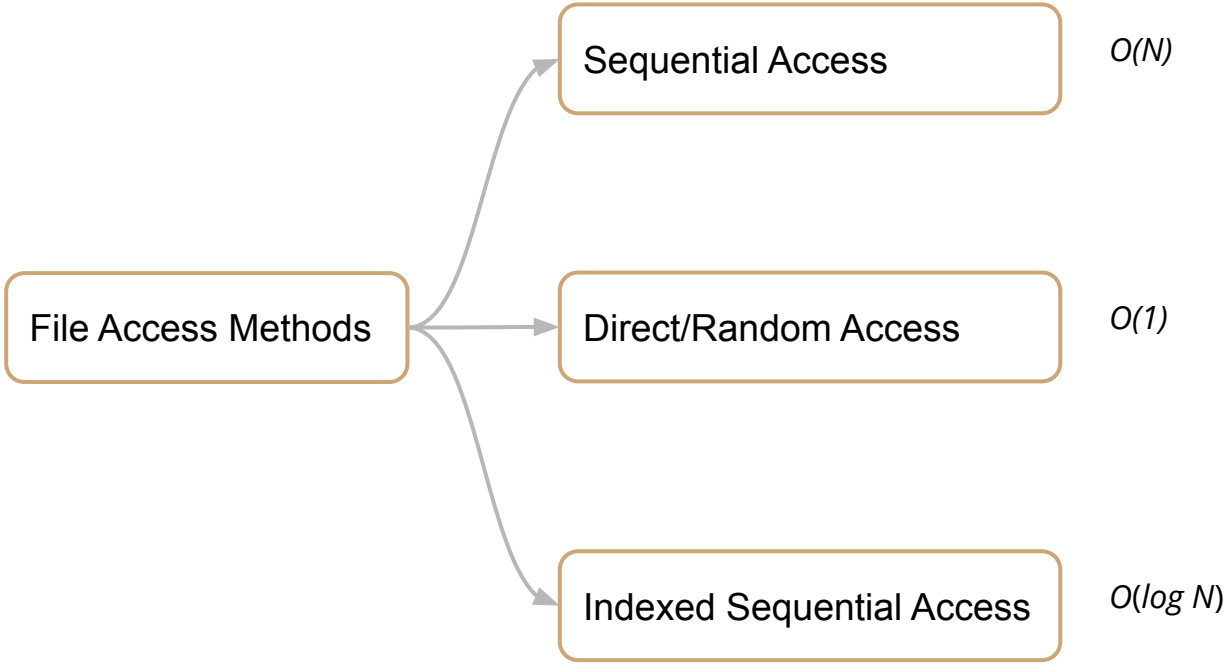
```
unsigned char buff[2048];
int fd = open(_______, _____);

while (_____) {
    int nbytes = read (fd, buff, 2048);
    lseek (fd, _____);
}
close (fd);
```

FS Related Data Structures?

- Data structures in user processes?
  - **Per-process open-file table**
- Data structure(s) used by OS?
  - **System wide (global) open-file table**
- Open a shared file?
- Locations
  - **On Disk Data Structures** (persistent)
  - **On RAM** (transient): initialized at boot time by reading on disk data structure

Access Methods



Variable vs. Fixed-Length Records

```
#include<iostream>
int main() {
  cout << “Hello world” << endl;
  return 0;
}
```

20 chars
2 chars
14 chars
31 chars
13 chars
3 chars

```
----+----1----+----2----+
#include<iostream>·····
·························
int main()··············
··cout·<<·”Hello”;·····
··return·0;·············
}·······················
```

25 chars
25 chars
25 chars
25 chars
25 chars
25 chars

Cursor “next line” = advance 25 bytes

In storage device: sequence of bytes

```
----+----1----+----2----+----3----+----4----+----5----+----6----+----7----+----8----+
#include<iostream>↵↵int main() {↵  cout << “Hello world” << endl;↵  return 0;↵}↵
```

Sequential Access Method

- Data in file are processed sequentially
  - To move the file pointer to offset 10,000, the program must read the first 10,000 bytes
  - Variable-length logical records
- Important interfaces: `read_next(___)` and `write_next(___)`
- OS system calls to skip the file pointer backward/forward
  - Interface: `skip(num_bytes)` or `lseek()` ⇒ O(N) operation
  - Skip ≠ jump: can't "jump" to a specific logical record
- The cost of inserting new data into a file?
  - at the end of the file?
  - anywhere else in the file?

Direct/Random Access Method

- **Fixed-length** logical records
- File pointer can “jump” to anywhere within the file
- Interfaces:
  - Sequential access: `read_next()`, `write_next()`
  - Random access: `position_file(rec_number)`: position the file pointer to a specific record ⇒ O(1) operation

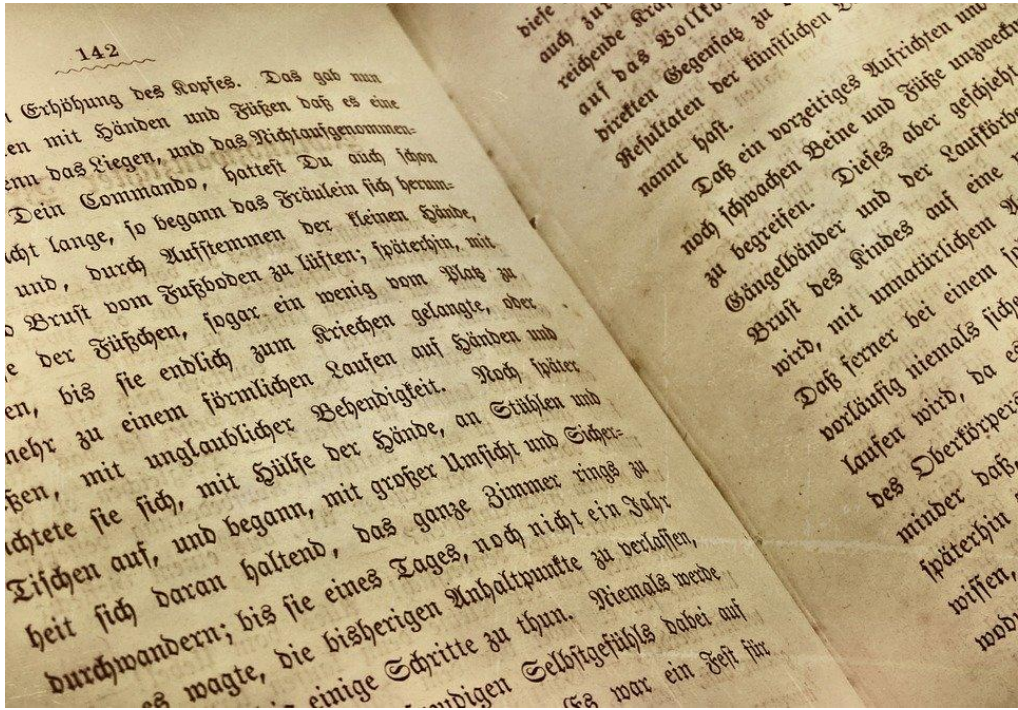
Why Indexed Sequential?

500-page book
100 words/page

Cost of locating one word?

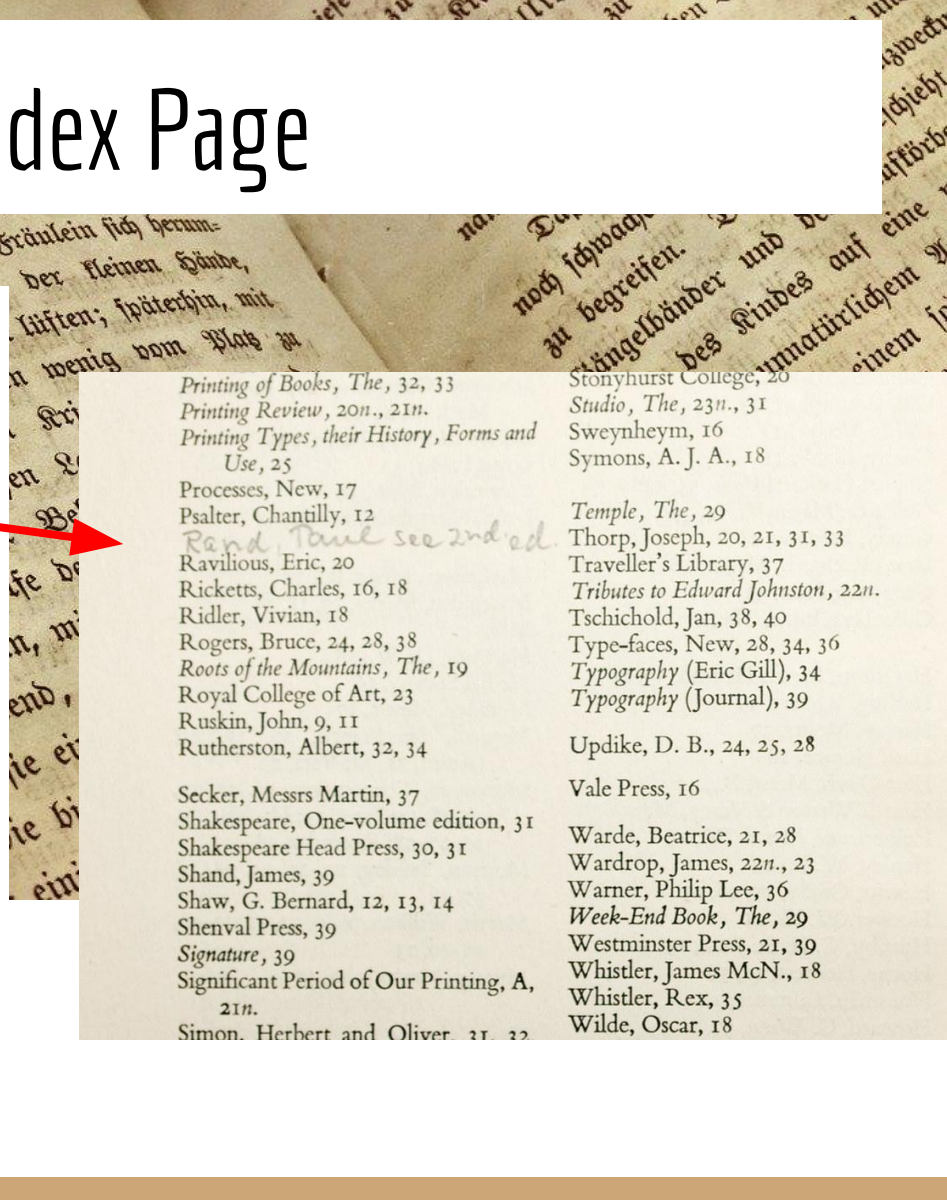
Word search in a book

- Total 500 pages
- 100 words/page
- Total 50,000 words to search
  - A **sequential** search costs an average of 25,000 comparisons (*across all 500 pages*)



Word search in a book + Index Page

- Total 500 pages
  - 100 words / page
- 10 **index pages**
  - 40 words / index page
  - Total 400 indexed words
- Search a word
  - Sequential search within **10** index pages ⇒ avg **200 comparisons**
  - Sequential search in the *target page* ⇒ avg **50 comparisons**
  - Total 250 comparisons across only **11 pages**



Index (+Sequential) Access Method

- Index provides a **random access** capability to quickly reach the **vicinity** of the desired record
  - The desired record itself must be searched in a sequential manner

Words index in a printed textbook

- Components required: data file + index file
  - Index file contains key and a pointer to a “small section” in the main file
  - Index file is search to find a key equal (or close) to the desired key value
  - Search continues (sequentially) within the “small section” as directed by the pointer
- Example: B+ Trees

Access Methods

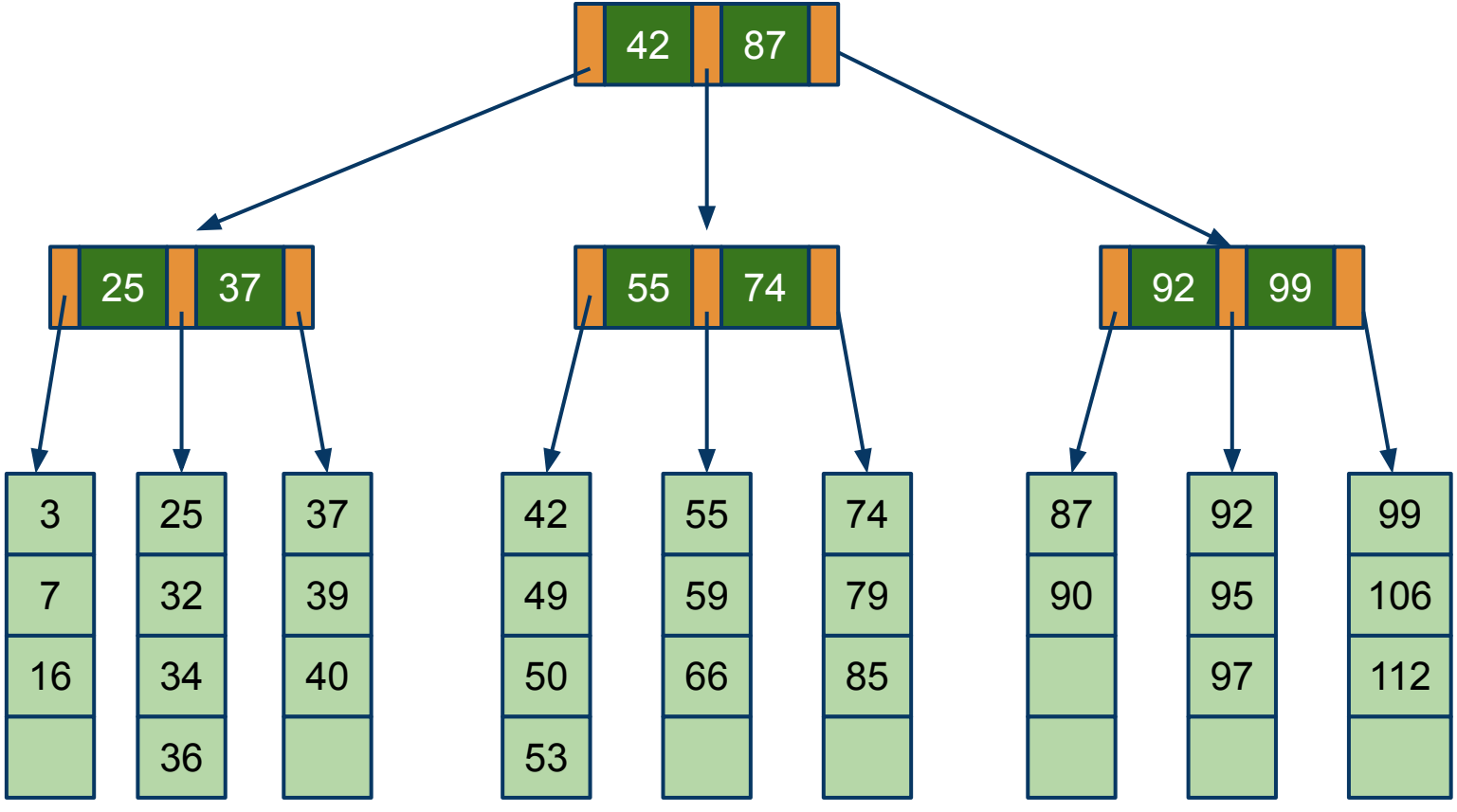
	Interfaces	Data Layout	Cost

Sequential	`read_next (`*`toBuff`*`)`
	`write_next (`*`fromBuff`*`)`
	`skip (`*`num_bytes`*`)`
		Variable length records	O(N)

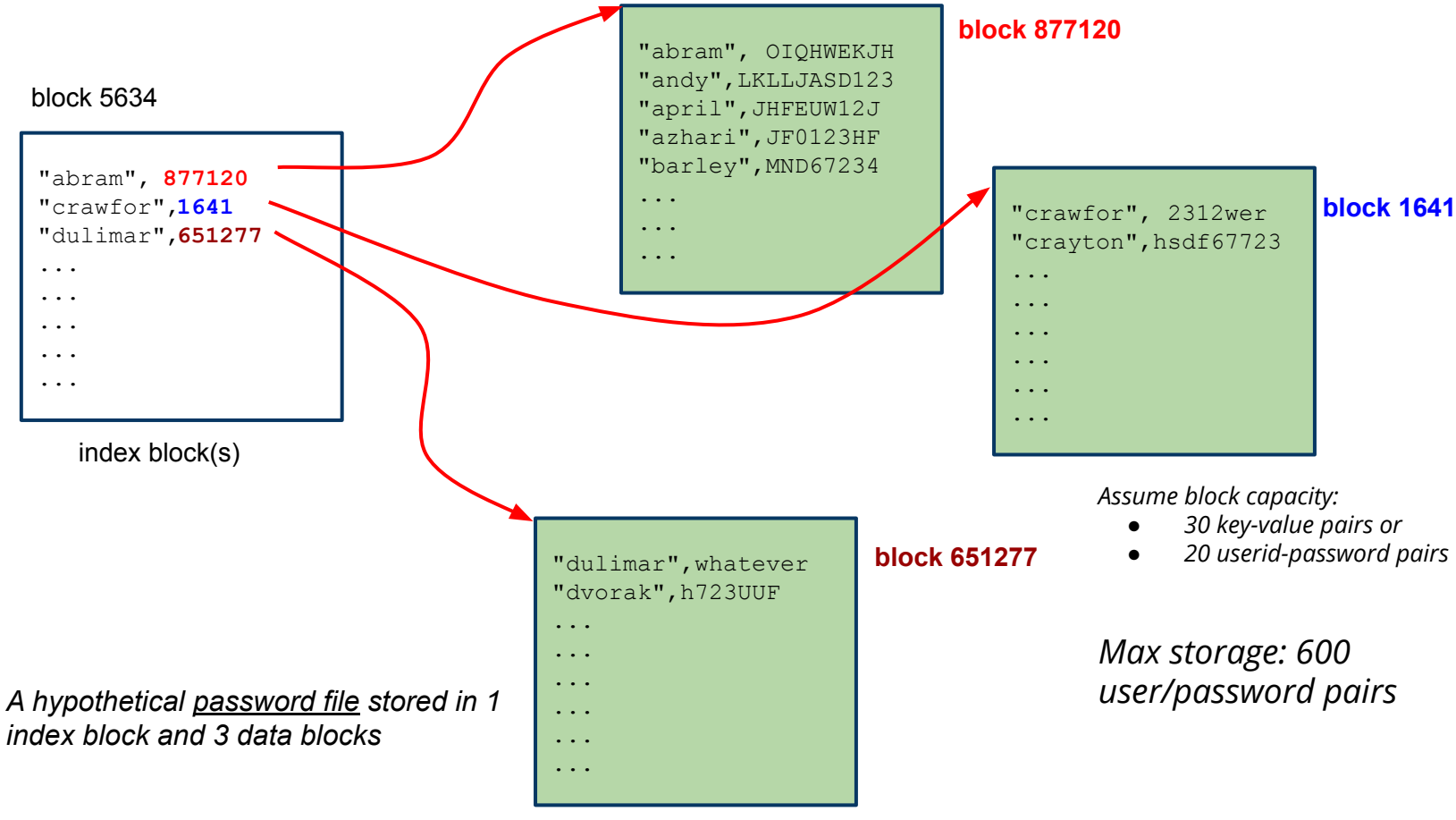
Direct/Random	`skip_to (`*`record_number`*`)`
	`read_next (`*`toBuff`*`)`
	`write_next (`*`fromBuff`*`)`
		Fixed length records	O(1)

Indexed Sequential	`find(`*`key`*`)`
	`read_next (`*`fromBuff`*`)`
	`write_next (`*`fromBuff`*`)`
		Tree structure & Variable length records	O(log N)

Data Structures & Algorithms: B^+ Trees



Indexed Sequential Access (Single-Level Index)



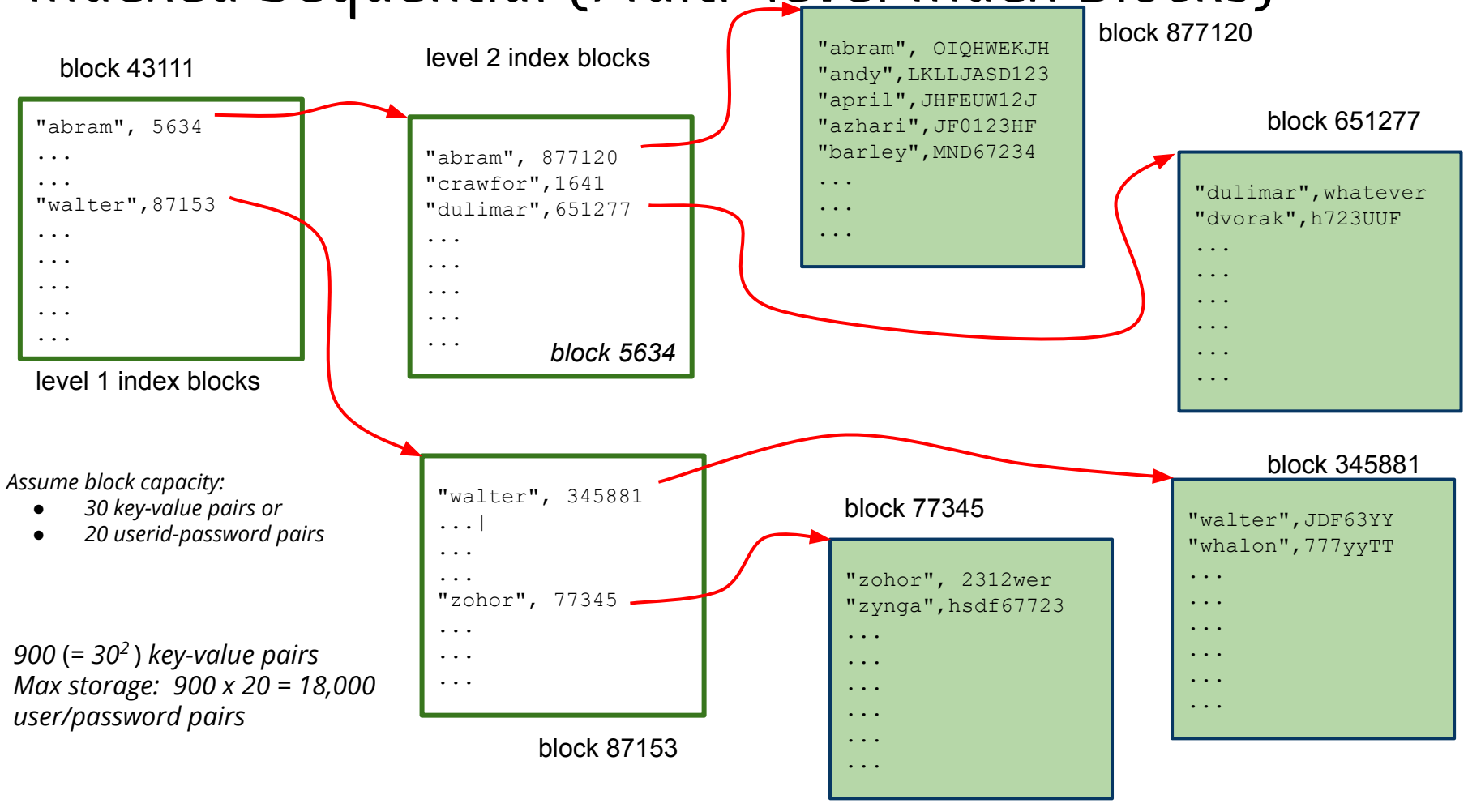
Assume block capacity:

- *30 key-value pairs or*
- *20 userid-password pairs*

Max storage: 600 user/password pairs

A hypothetical password file stored in 1 index block and 3 data blocks

Indexed Sequential (Multi-level Index Blocks)



Assume block capacity:

- *30 key-value pairs or*
- *20 userid-password pairs*

900 (= 30^2) key-value pairs
Max storage: 900 x 20 = 18,000 user/password pairs