Part 2: File System Implementation

(ish, more of a general survey of some of the big topics)

File System Trends

1. most files are small (~2 KB or less)
2. Average file size is growing (~200 KB avg)
3. A few large file use most of the disk space
4. the # of files is large (~100,000 on avg)
5. directories are typically small (< 20 entries)

General File System Structure

1. View disks as composed of blocks |block| ≈ 4 kB
   |sector| ≈ 512 B
2. logically divide blocks into "regions"
   a) data - biggest, stores files contents
   b) Inode
   c) Allocation structures - tracks free blocks
   d) Superblock
      → file system type
      → size file system

The Old UNIX File System

→ stored blocks linearly

sectors 0 → 7 : block 0

8 → 14 : 1

16 → 23 : 2

→ "first fit" allocation

"file" # Fragmentation Example

start: 4 files (A, B, C, D), each need 2 Blocks

- rm B and D
- Allocate E: 4 Blocks

		~~B1~~	~~B2~~			~~D1~~	~~D2~~	
A1	A2	E1	E2	C1	C2	E3	E4	

file fragmentation

→ fragmented files require random access (slow on Disk)

1) Be disk aware

2) keep related things close <u>And</u> unrelated things far apart

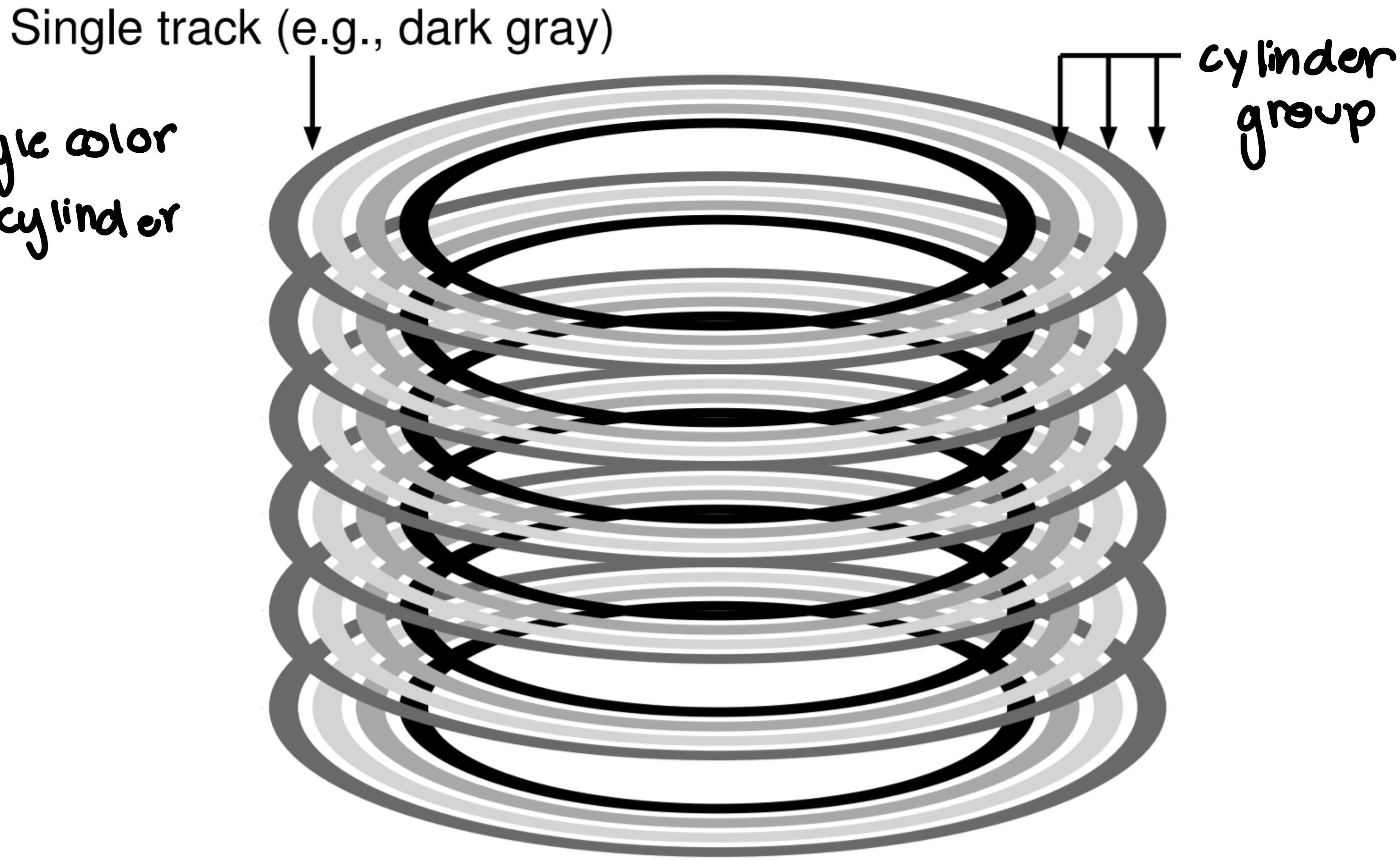
The Big Fix: Be Disk Aware

Cylinder: a set of tracks on different surfaces which are the same distance from the center

Divide disk into cylinder groups.

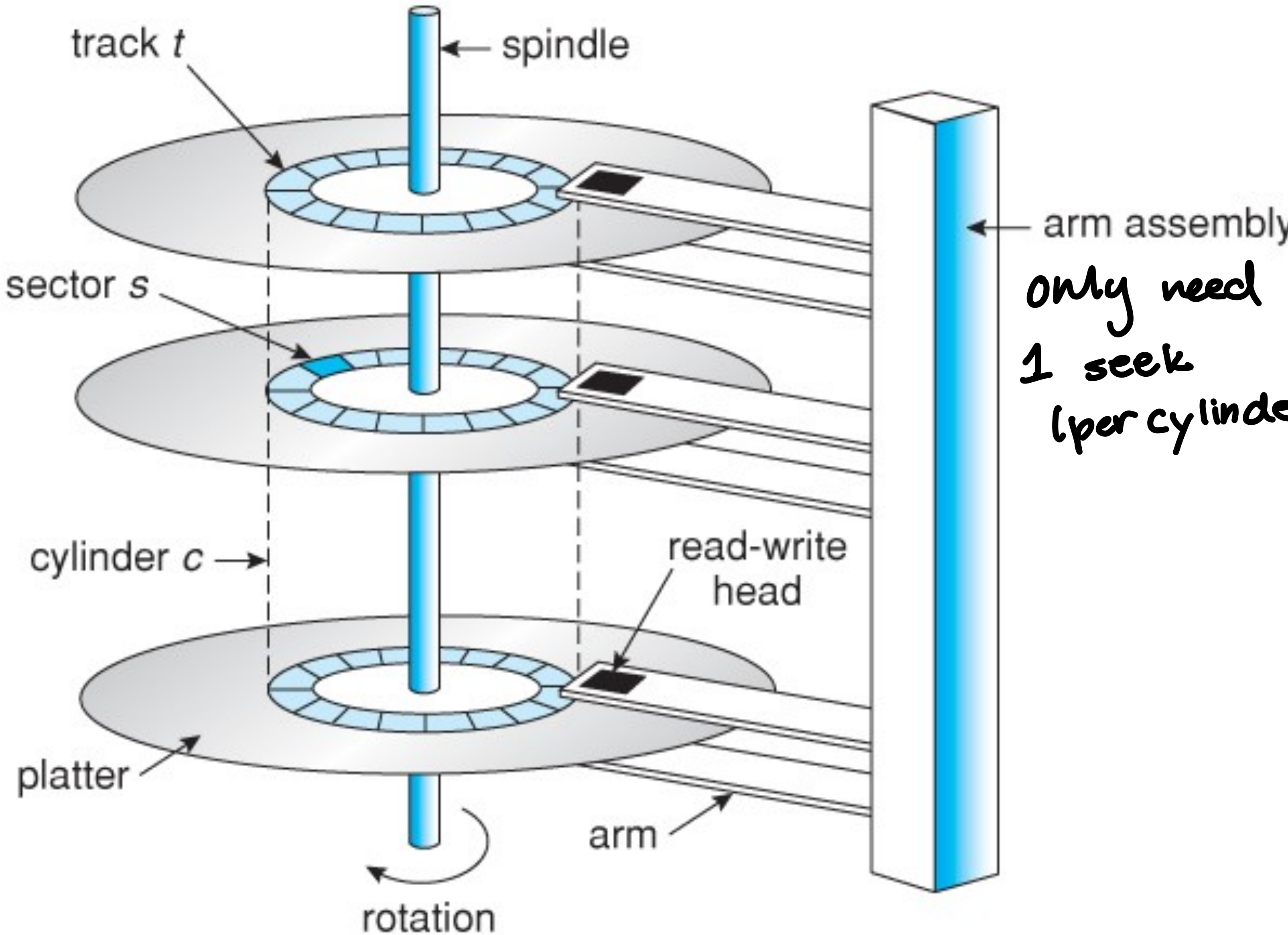
→ set of consecutive cylinders

Visual



single color ⇒ cylinder

Visual #2



only need
1 seek
(per cylinder)

The Inode

Inode: stores the meta data for a file

- length, permissions, location on disk, owner
- referenced by number (inode number, i-number)
- "index node" – originally stored as an array

Example Inode

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed? - permissions
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

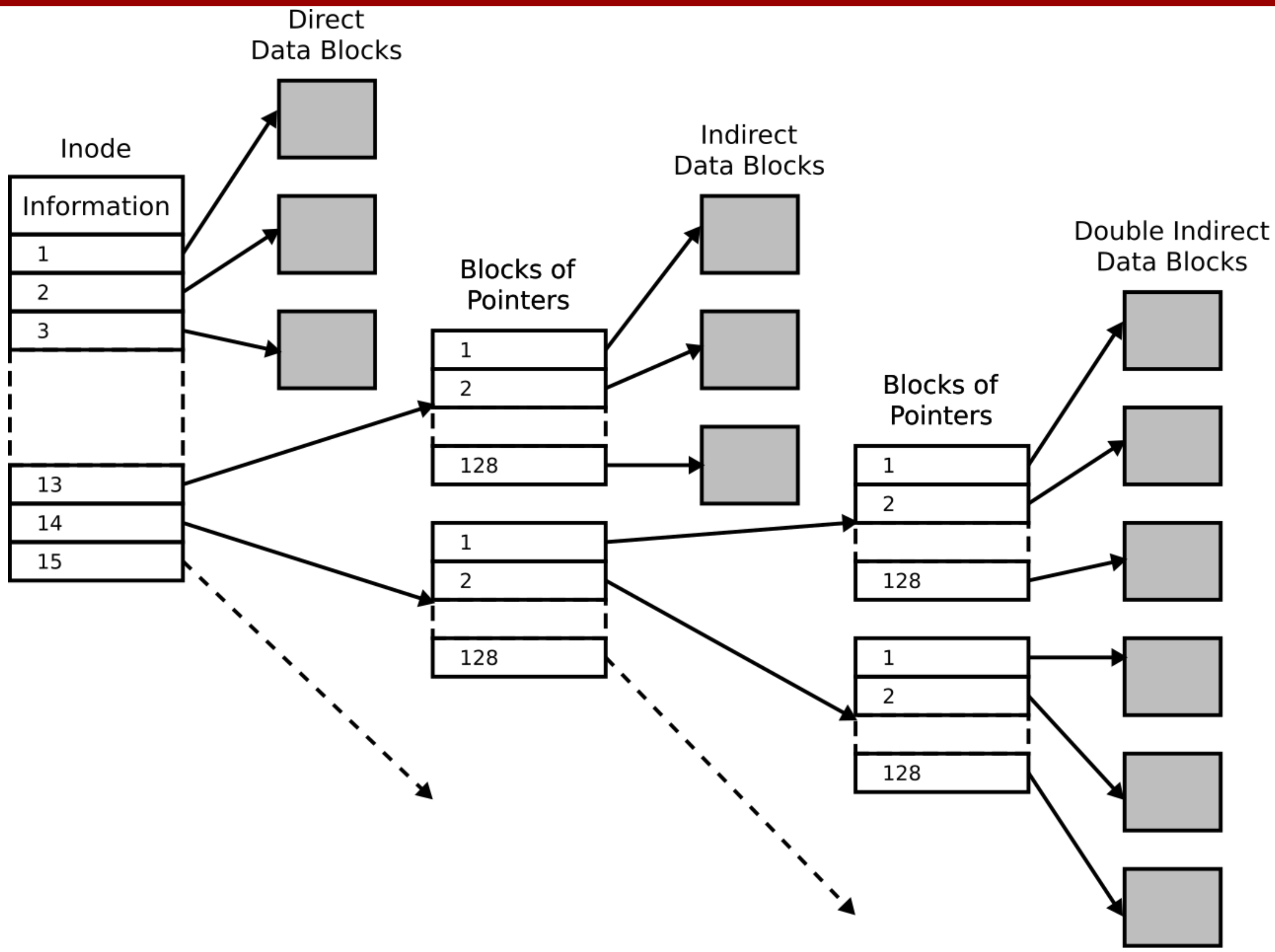
Figure 40.1: **Simplified Ext2 Inode**

Tracking On-Disk Location

<u>Direct Pointer:</u> refers directly to a data block
→ one per block

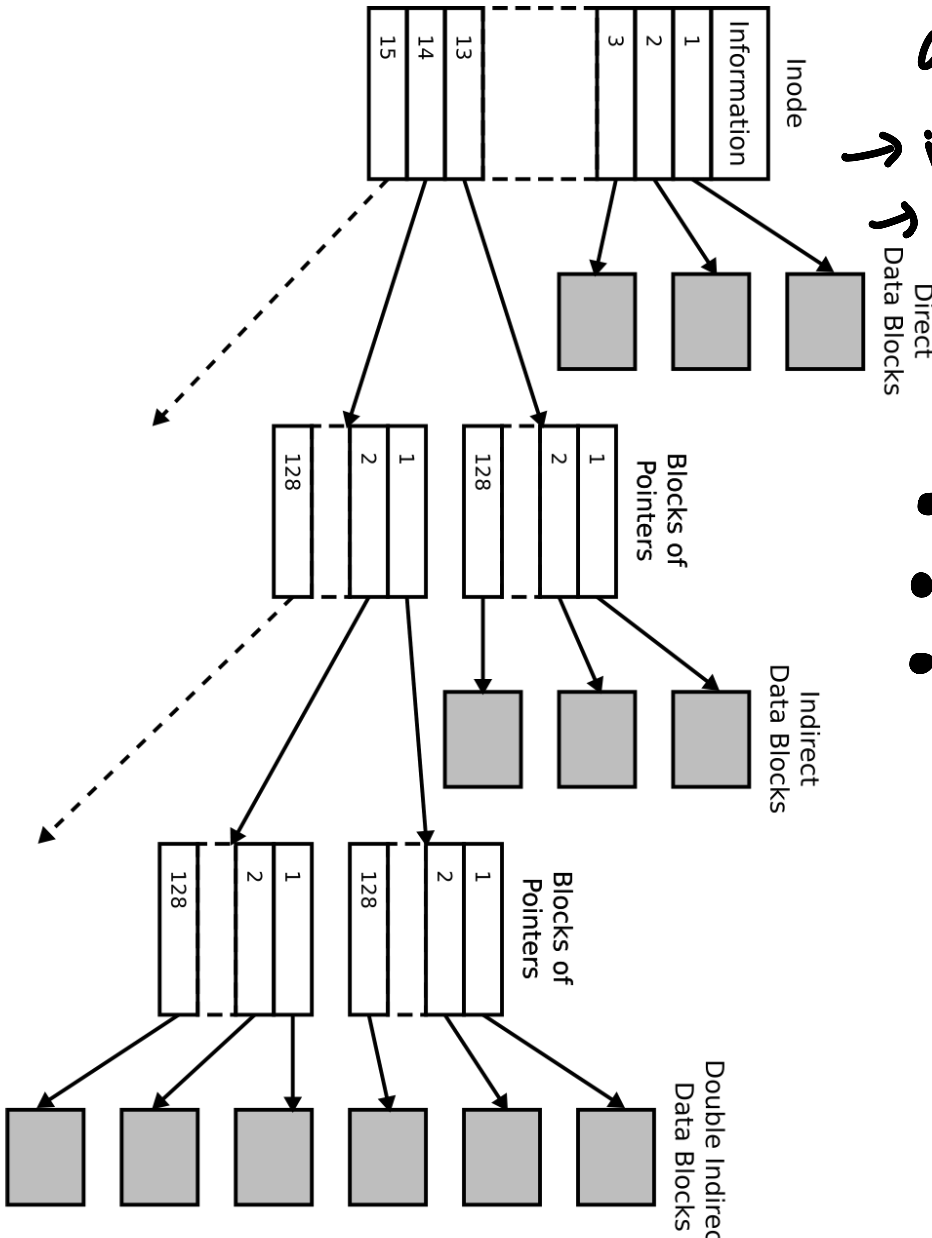
<u>Indirect Pointer:</u> refers to a block which contains direct pointers

Visual



Direct
Data Blocks
Inode
Information
1
2
3
13
14
15
Indirect
Data Blocks
Blocks of
Pointers
1
2
128
1
2
128
Double Indirect
Data Blocks
Blocks of
Pointers
1
2
128
1
2
128

Visual, Rotated



a tree!

→ in balanced tree

→ multi-level index

- pointers
- Extents
- hybrid

Free Space Management

(At least) three approaches: Track which blocks are free?

1. Bitmap
   - store 1 bit per block
   - 0: allocated , 1: free

fast
"is block <10> free?"

slow
"give me a free block"

2. linked list of free blocks
   - supernode contains location of head

fast
"give free block"

slow
"is block free?"

3. B-trees

B-Trees

A B-tree of order *m* is:

— typically wide, not deep

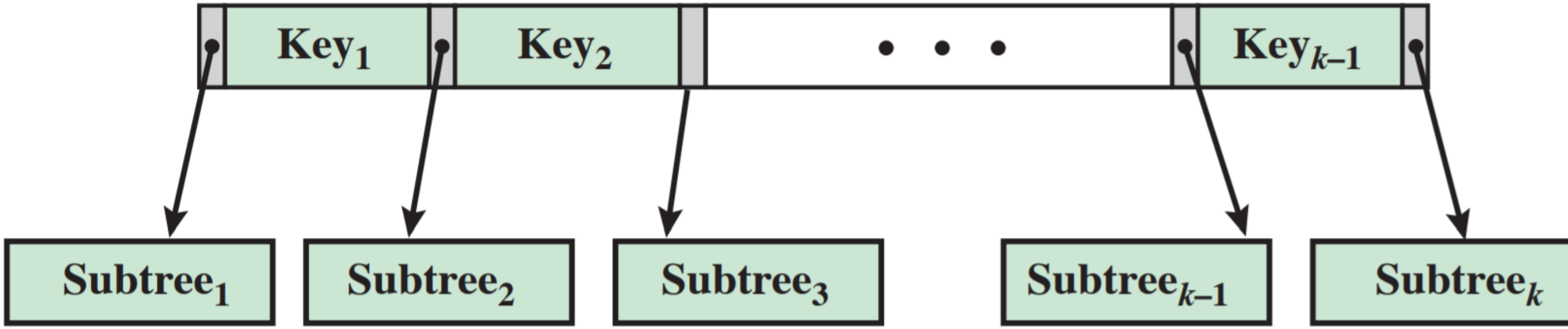
1. a tree (a rooted acyclic graph)
2. Every node has at most m children
3. every internal node has at least $\lceil m/2 \rceil$ children

   not the root
4. the root has at least 2 children
5. a node w/ k children contains (k-1) ordered keys

   ↳ binary search
6. all leaves are at the same depth and contain no information

   ↳ perfectly balanced, gaurantee log(n) ops no only

Visual of a B-Tree Node



- subtree1 < key 1
- key1 < subtree2 < key 2

Benefits of a B-Tree

We can efficiently search a binary-search tree…

…so why use a B-tree?

Space efficiency

→ store 1 node per block

- BST has 1 key per node ⇒ only be using 0.2% of block
- B tree: way more than 1 key per node

Crash Consistency

given that crashes can occur at arbitrary points in time, how does the os garauntee that larger than sector writes are atomic? ie. prevent partial writes/keeping the disk consistent

1. file system checker (fsck)
   - let inconsistencies happen
   - fix them later

search the entirety of Disk ⇒ SLOW

2. Journaling (write-ahead logging)

Journaling

write

Basic Idea: Before a write request jot down a note (in a well known location) describing the operation

Data Journaling

1. Journal write
   a) transaction-begin
   b) the "write" itself
2. Journal commit
   a) |transaction-end| < |sector|
3. "check point"
   → write to original location

same write to 2 diff places

Double write times

Visual

	Journal			File System	
TxB	Contents (metadata)	Contents (data)	TxE	Metadata	Data
issue	**issue**	**issue**			
complete					
	complete				
		complete			
			issue		
			complete		
				issue	**issue**
					complete
				complete	

Figure 42.1: **Data Journaling Timeline**

Recovery

Crash occurs:

1. Before or during step (1): Erase journal entry
2. Before step (2): Ditto
3. Before or during step (3): use journal entry to re-try write

“Real Life” File Systems

Linux:

- the EXT family (e.g., ext3, ext4)

Windows:

- FAT, HPFS, NTFS

Others:

- APFS (Mac), ZFS, XFS, AFS, NFS,