6.3 Directories and Naming

- Directories
- Name-Space Management

Desired Properties of Directories

- No restrictions on names
- Fast
- Space-efficient

S5FS Directories

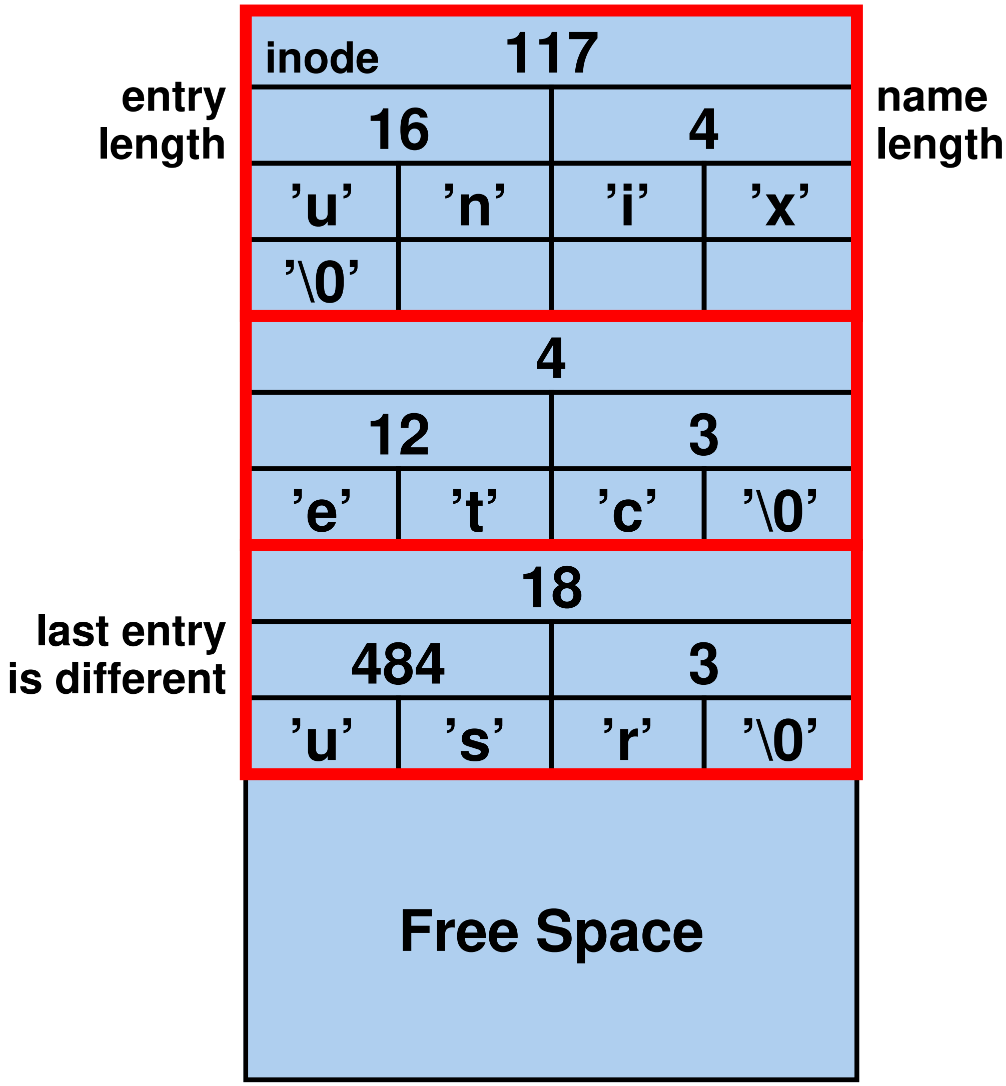
Component Name	Inode Number

directory entry

.	1
..	1
unix	117
etc	4
home	18
pro	36
dev	93

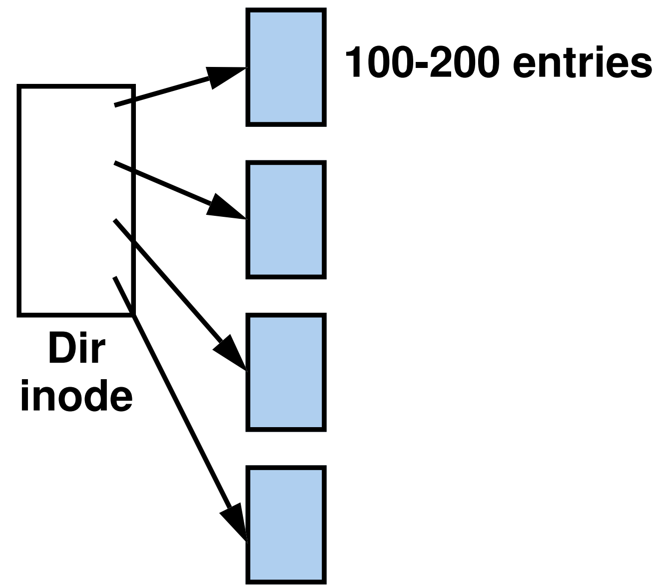
- each entry is 32 bytes long in S5FS (see "`s5fs.h`" in **`weenix`**)
- this is what get stored inside the "data blocks" of a *directory file*
  - i.e., a directory is implemented as an inode and the disk map inside the inode points directly or indirectly to its "data blocks"

FFS Directory Format



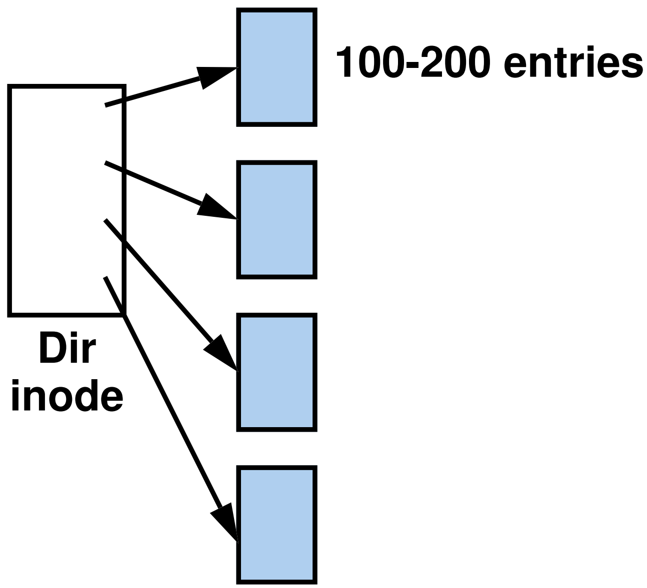
Content of Directory File

- unsorted entries



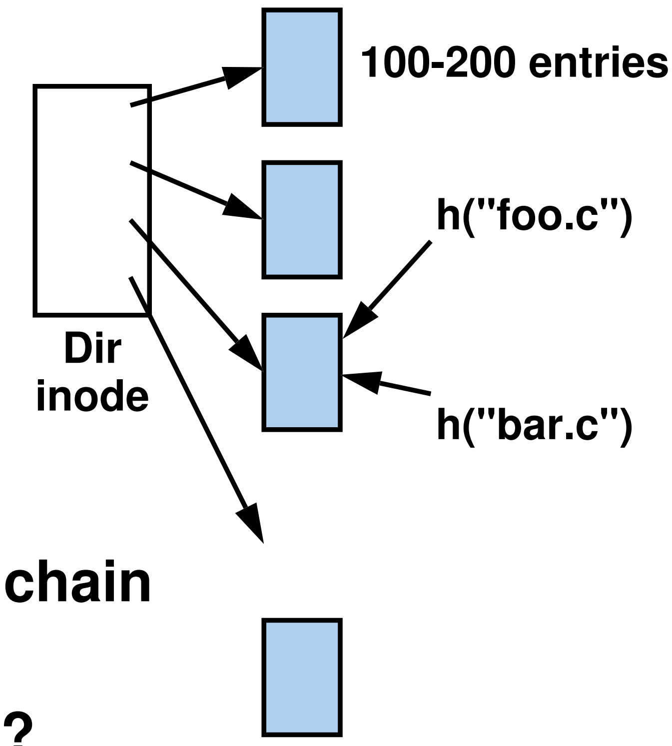
Look Up Filename In A Directory

- How to look for a file named "foo.c"?
  - *O(n)* is no good
    - linear search
  - *O(log n)* is desirable
    - B+ tree
  - *O(1)* is great
    - hash table



Hash Table

- Hash file names to disk blocks
  - don't treat the directory contents as a sequential file
  - treat it as an array of hash buckets
    - each ***bucket*** is stored in a ***disk block***
  - h(filename) tells you which bucket (i.e., block) the file information is in
    - ***one disk read***

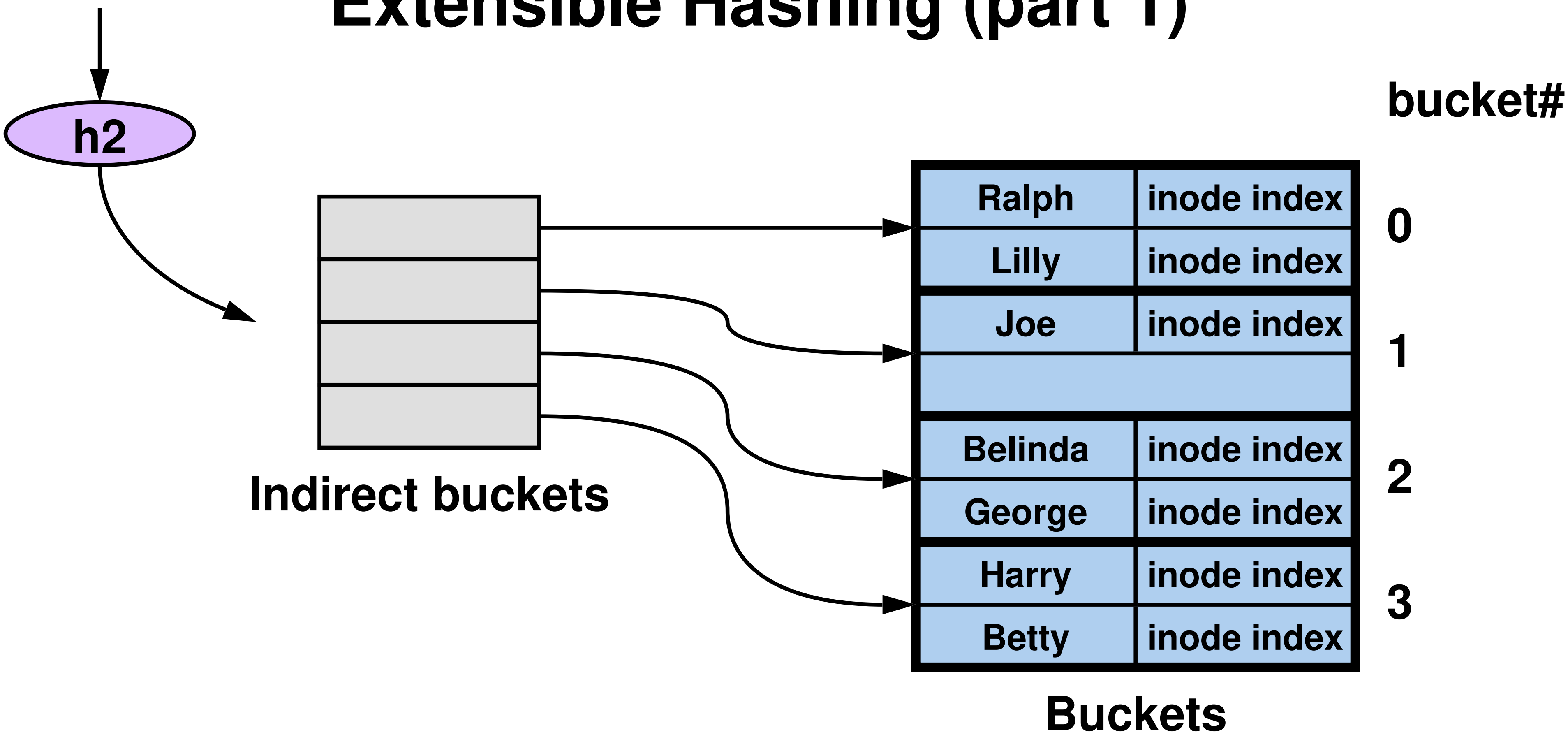


- The usual problem with hash tables
  - ***collisions***
    - length of conflict/collision resolution chain is determined by block size
    - what do you do when the block is full?
      - traditional approach is to get a new hash function and rehash every entry
      - too expensive

Extensible Hashing

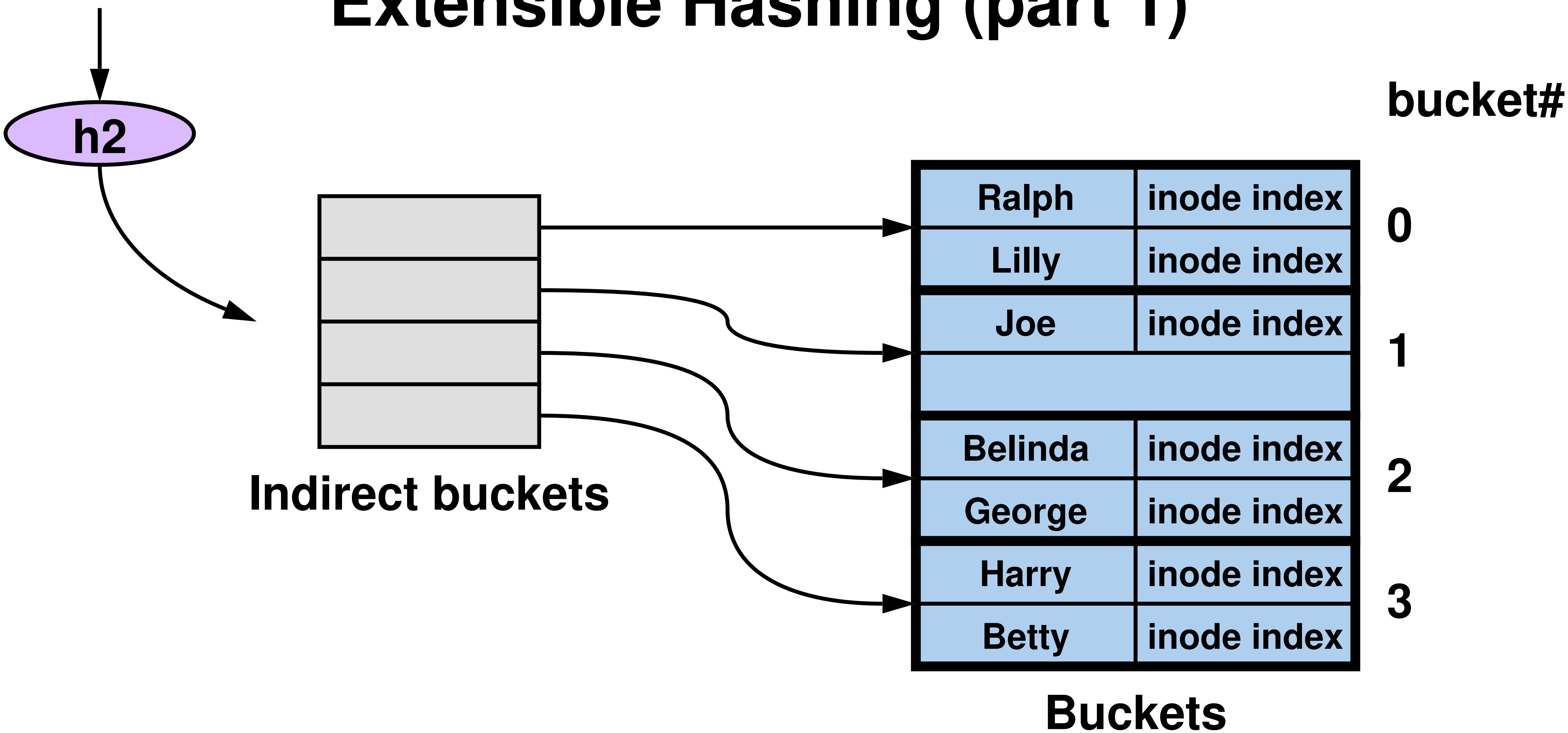
- Uses a sequence of hash functions, h_0, h_1, h_2, ...
  - where h_i hashes names to 2^i buckets
  - for any name x, the low-order i bits of $h_i(x)$ are the same in $h_{i+1}(x)$
    - h_{i+1} is an *extension* on h_i
- To use extensible hashing, also need to add a level of *indirection*

Extensible Hashing (part 1)



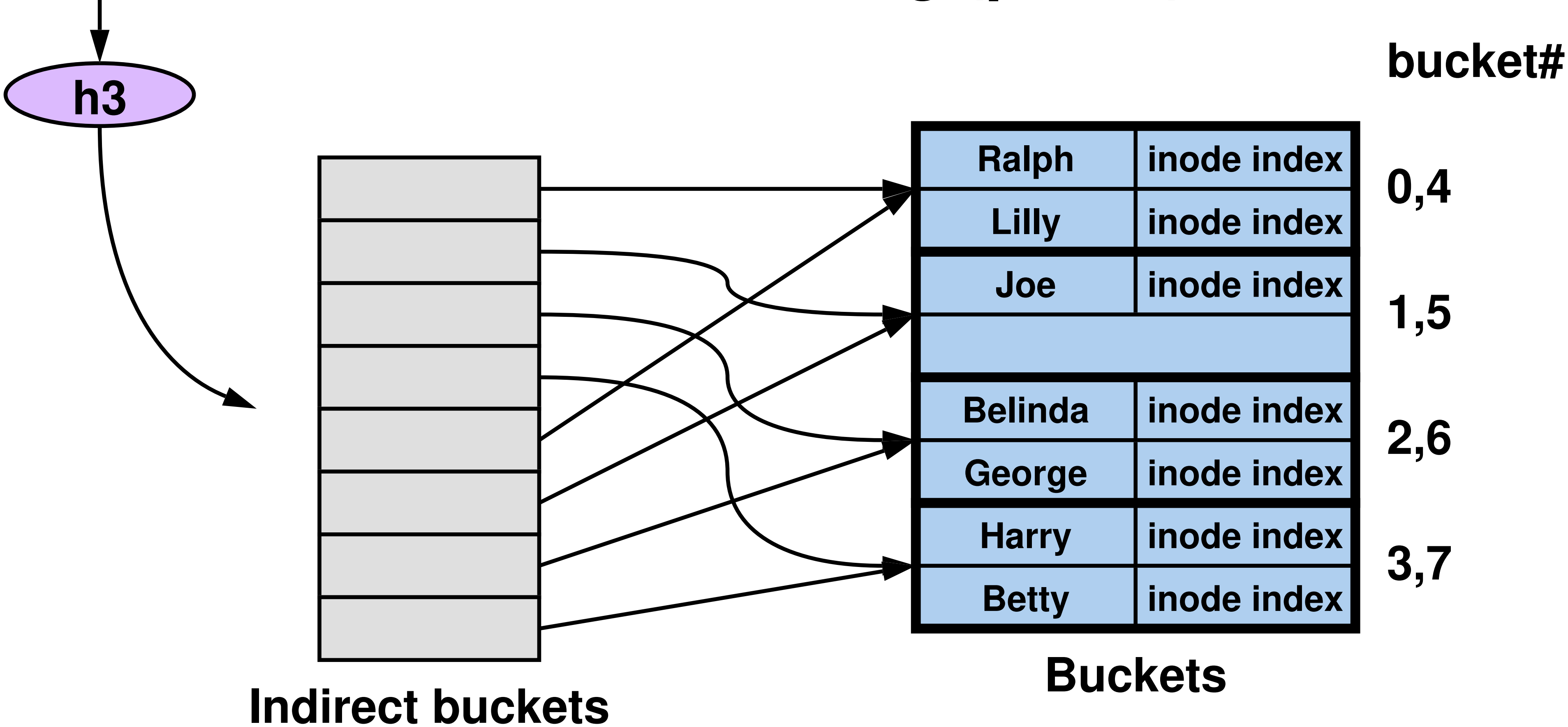
- Bucket = block

Extensible Hashing (part 1)



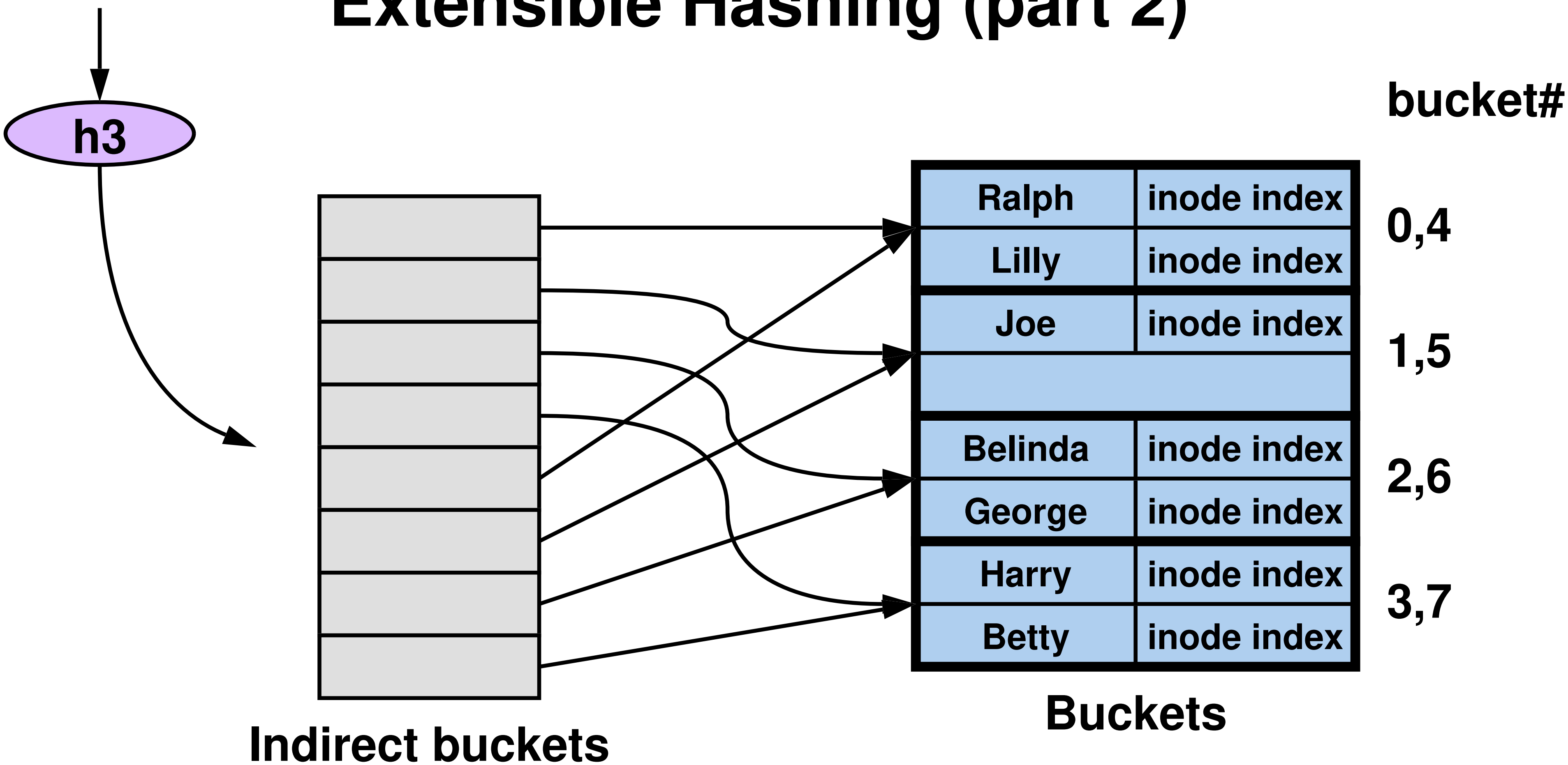
- insert(Fritz)
  - h2(Fritz) = 2
    - but bucket 2 is full

Extensible Hashing (part 2)



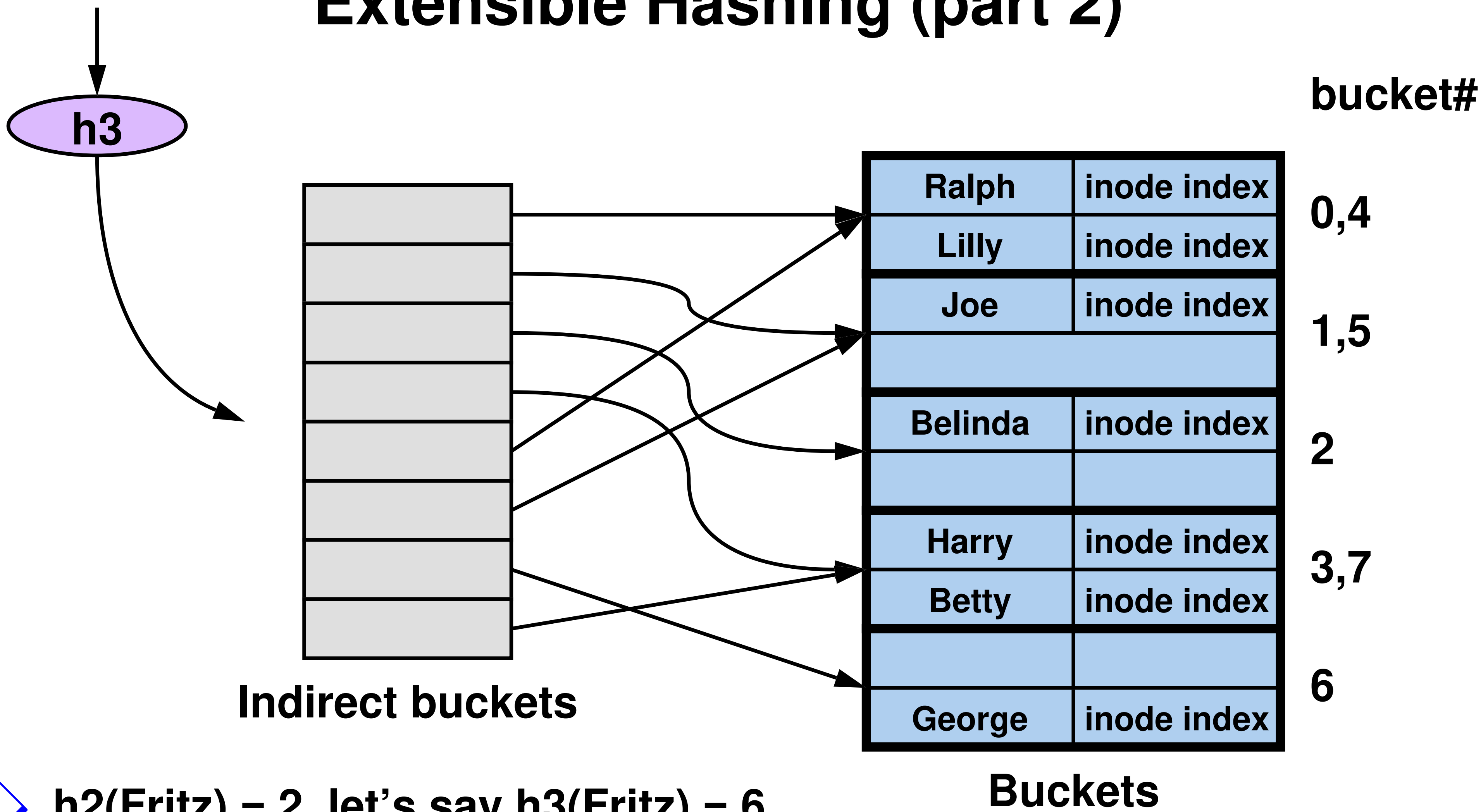
- First you double the number of indirect buckets
  - since there are 8 buckets, you need h_3
  - then you rehash all the entries using h_3
    - easily accomplished for ***extensible hashing with indirection***

Extensible Hashing (part 2)



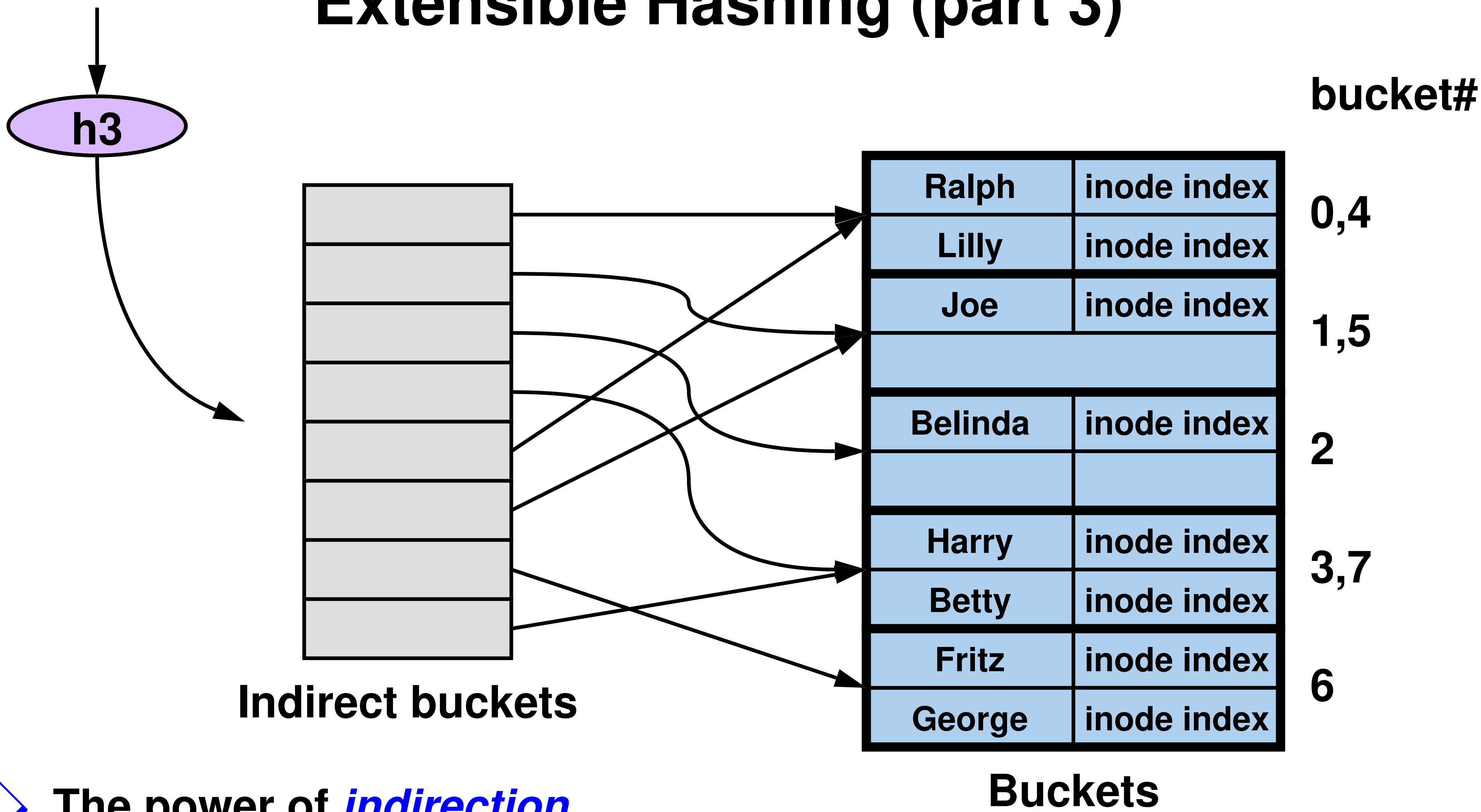
- h2(Fritz) = 2, let's say h3(Fritz) = 6, bucket 2 is full, get a new bucket, and rehash all items in bucket 2 with h3
  - on the average half of them should go to bucket 2 and the other half should go to bucket 6
    - e.g., h3(Belinda) = 2, h3(George) = 6

Extensible Hashing (part 2)



- h2(Fritz) = 2, let’s say h3(Fritz) = 6

Extensible Hashing (part 3)



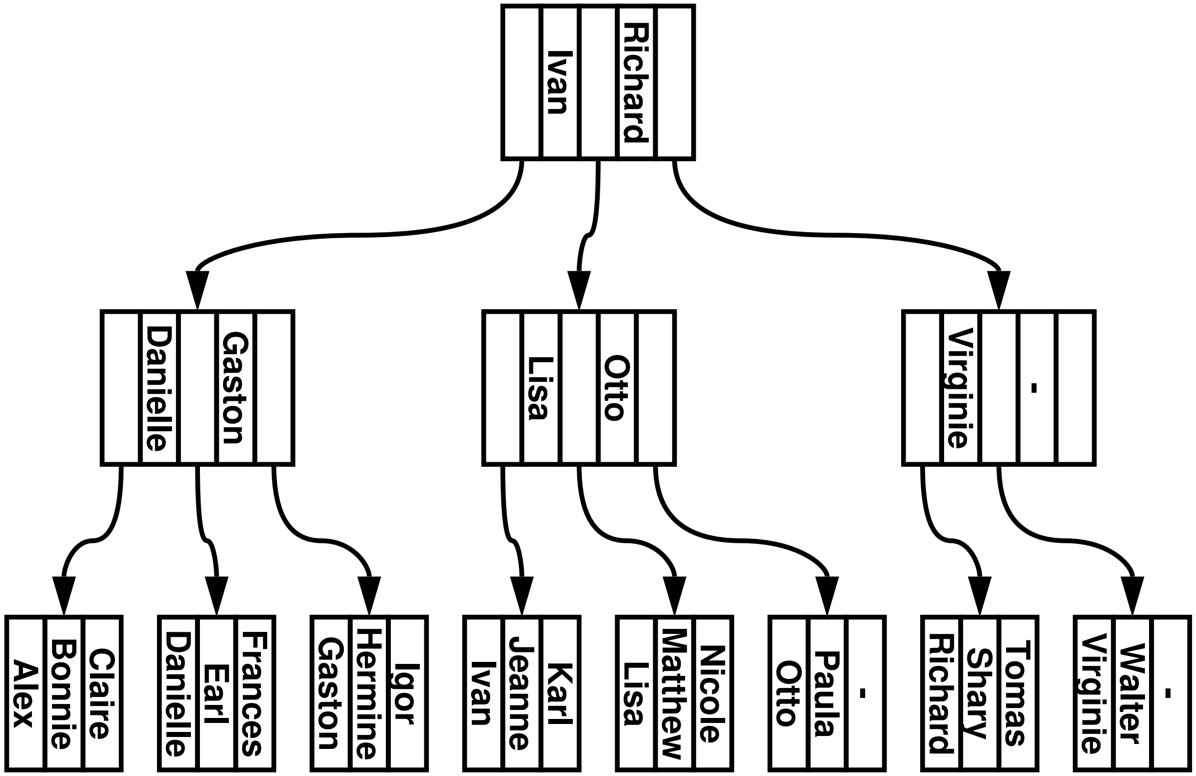
- The power of *indirection*
  - without the indirect buckets, it will be expensive, i.e., *O(n)*
  - with indirect buckets, *O(1)*

B Trees

- A ***B tree*** of order ***m*** has the following properties:
  1) every node has ≤ *m* children
  2) every node (except root & leaves) has ≥ *ceil(m/2)* children
  3) the root has at least 2 children (unless it is also a leaf)
  4) all leaves appear at the ***same level*** and carry no keys
     - usually omitted from the drawings
  5) a non-leaf node with *k* children contains *k-1* keys
  - guarantee that each node of size *m* is at least 50% full
  - *m* is often large to reduce the number of disk accesses
- Each ***node*** in a ***B tree*** corresponds to one ***disk block***
  - ***B+ tree:***
    - internal nodes contain no data, just keys
    - leaf nodes are linked to ease sorted sequential traversal
  - there are a lot of B tree variations in various application areas
- We will illustrate the basic idea with an example
  - real B+ tree algorithm is more complicated than depicted

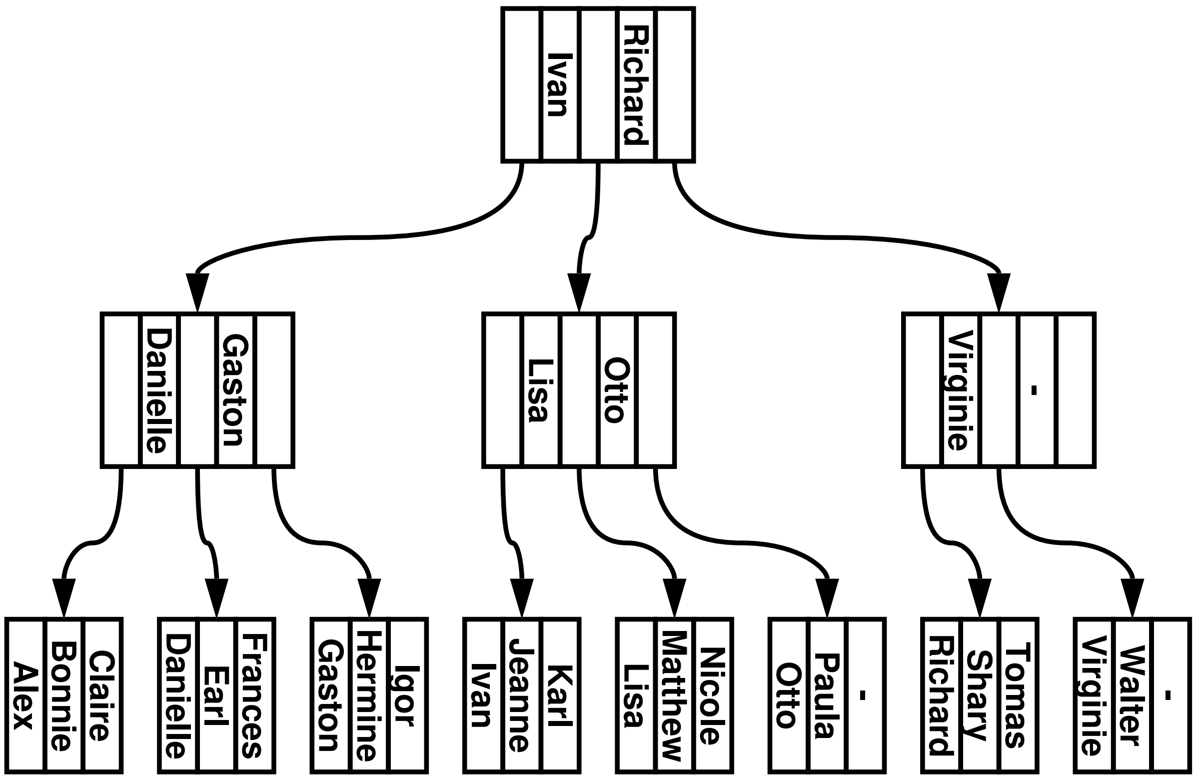
B+ Trees (part 1)

- Ex: a *B+ tree* of order 3
  - every internal node has either 2 or 3 children
    - if a node has > 3 children, it needs to split into two nodes
    - if a node has < 2 children, it needs to merge with its neighbor
    - we will assume that this is also true for the leaf nodes



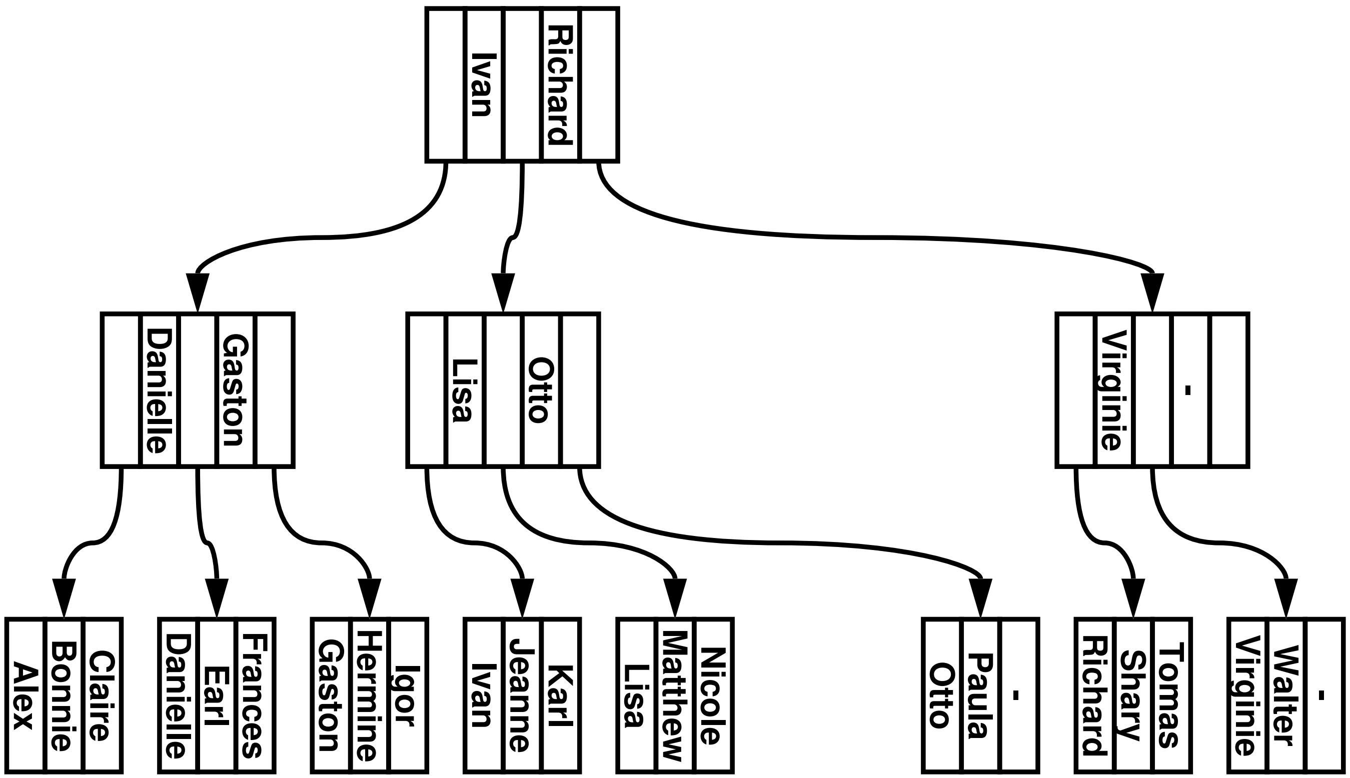
B+ Trees (part 1)

- ***Insertion:*** **e.g., "Lucy"**
  - **let's say you search left if < key, and search right if ≥ key**



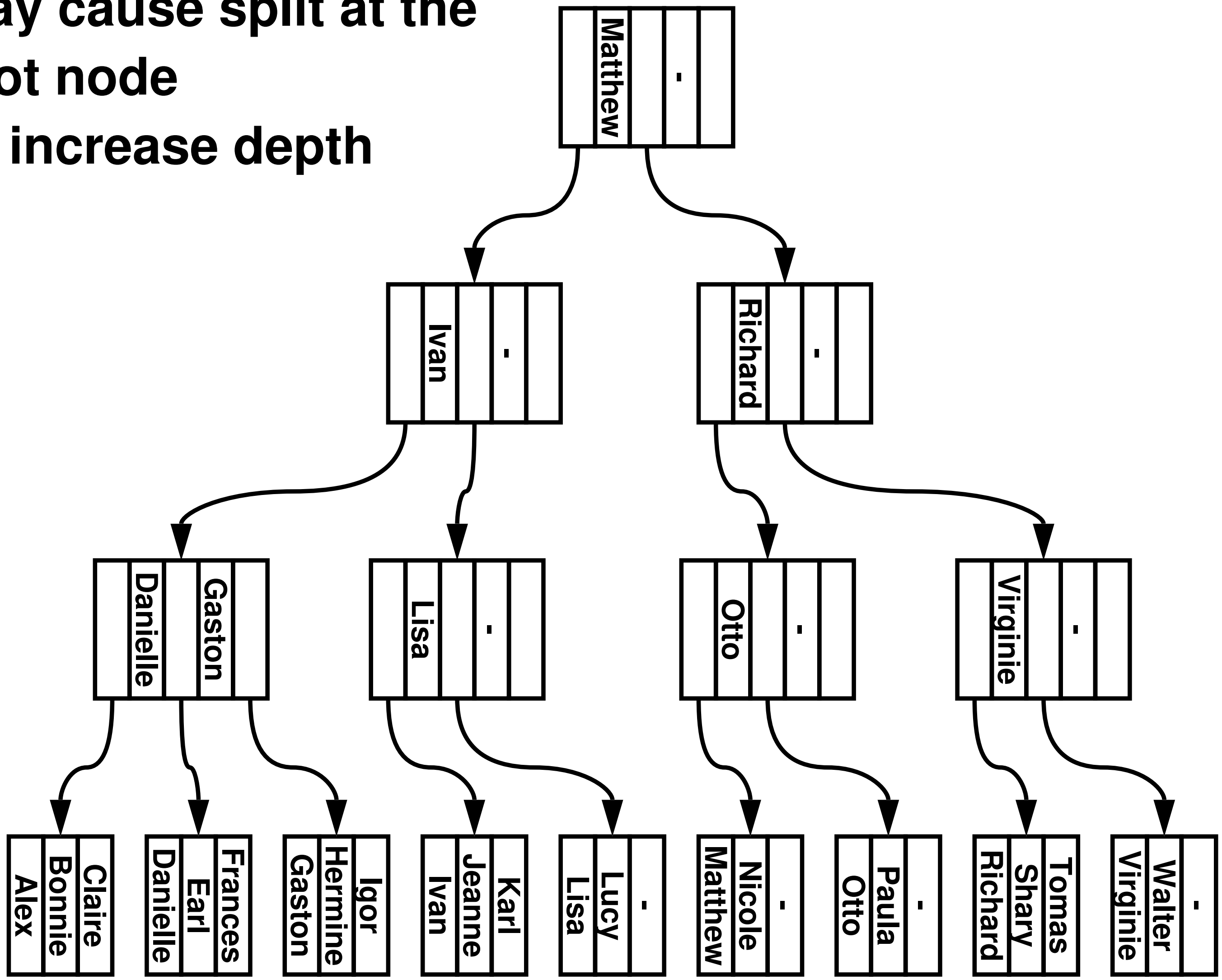
B+ Trees (part 1)

- ***Insertion:*** e.g., "Lucy"
  - let's say you search left if < key, and search right if ≥ key



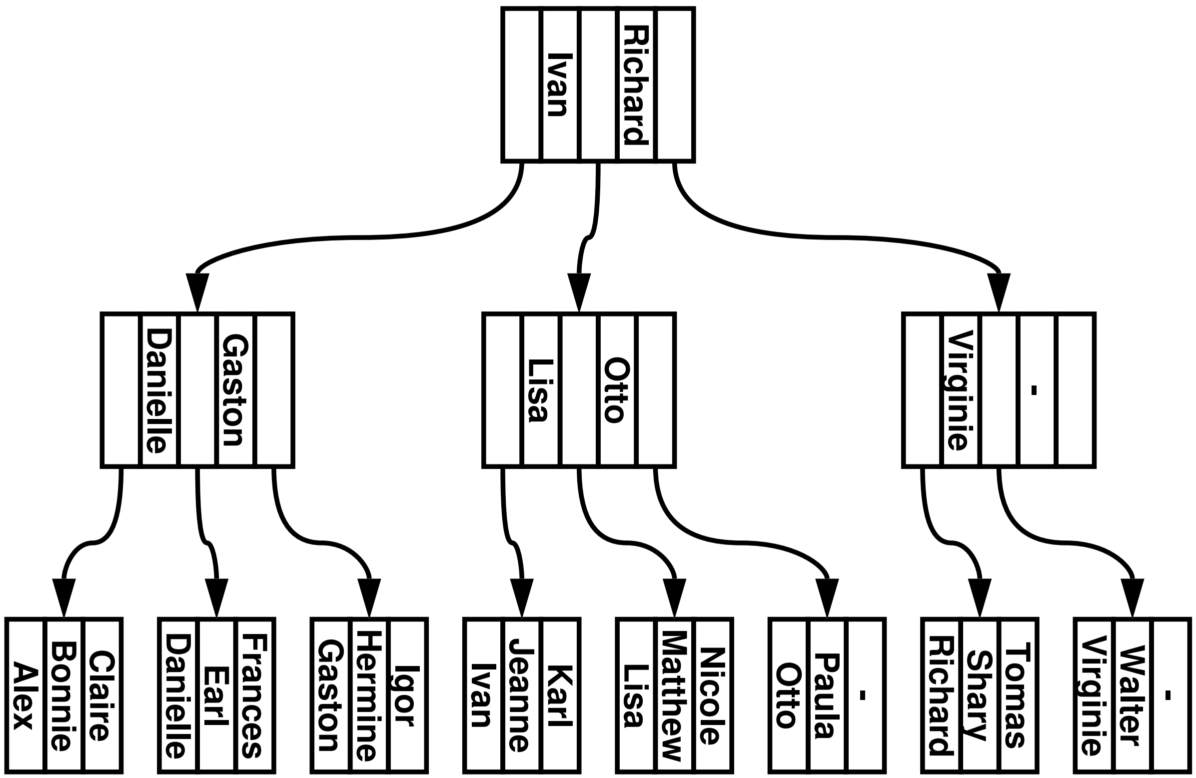
B+ Trees (part 2)

- ***Insertion:*** e.g., "Lucy"
  - may cause split at the root node
    - increase depth



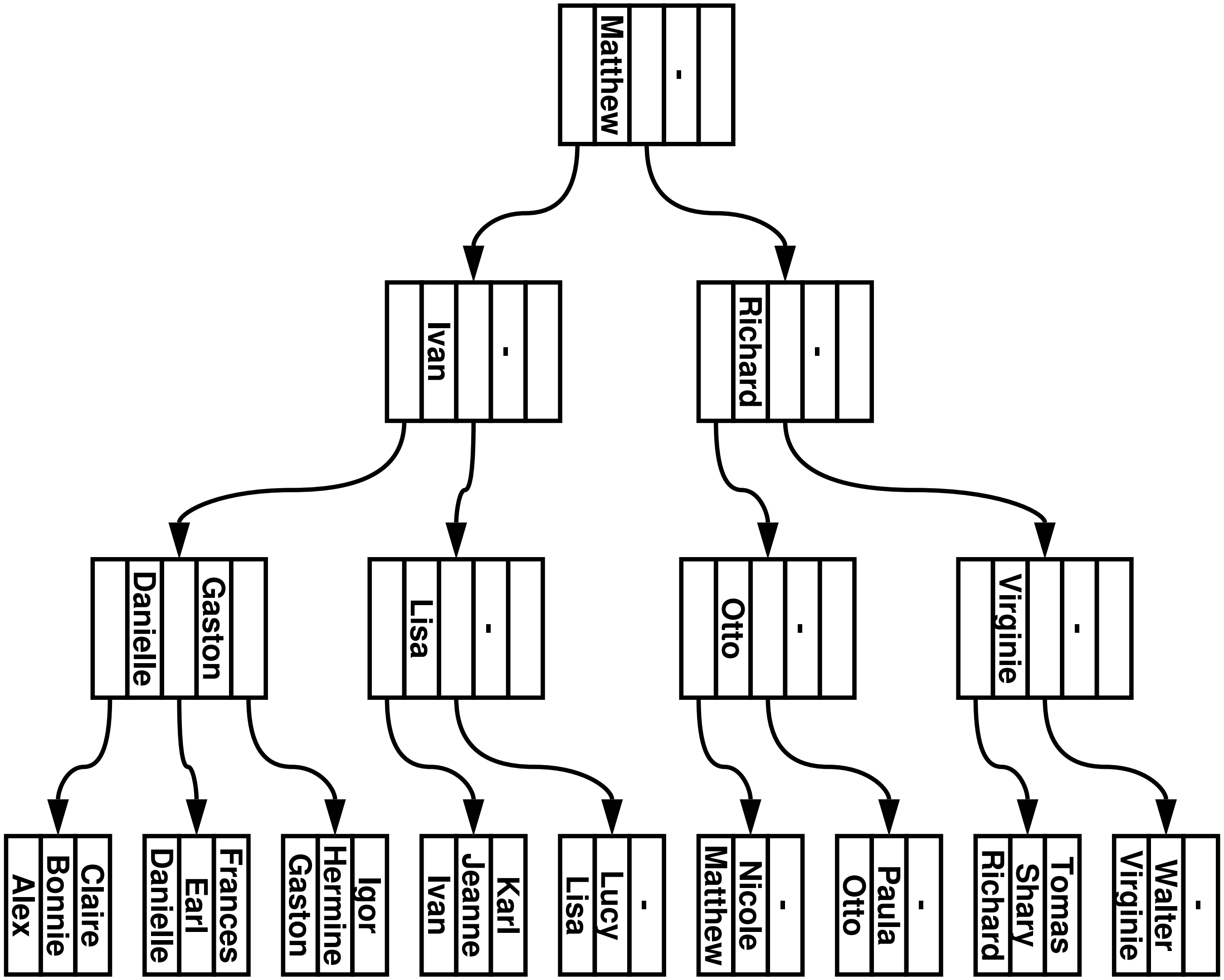
B+ Trees (part 2)

- ***Insertion:*** e.g., "Lucy"
  - there is another way
    - "rotate" Nicole to the right (since cannot "rotate" Lisa to the left)



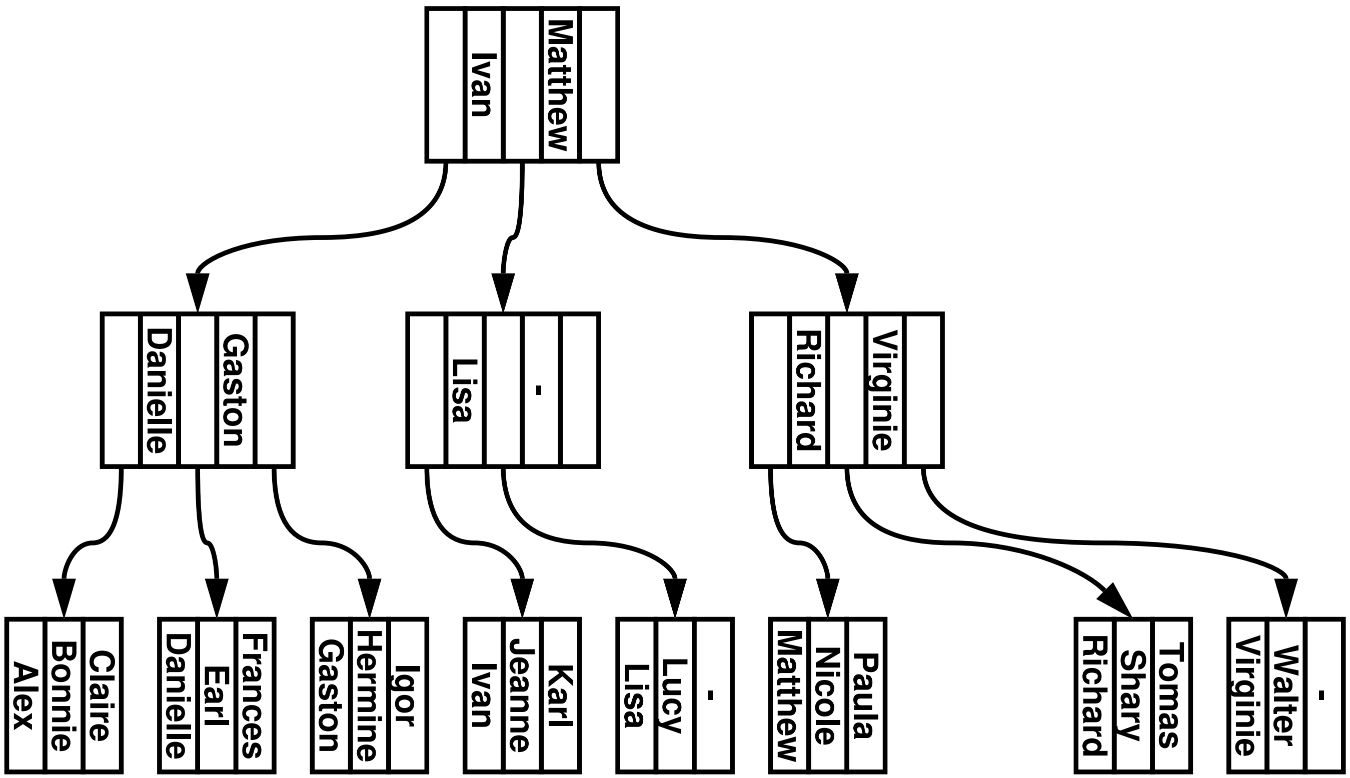
B+ Trees (part 3)

- *Deletion:* e.g., "Otto"

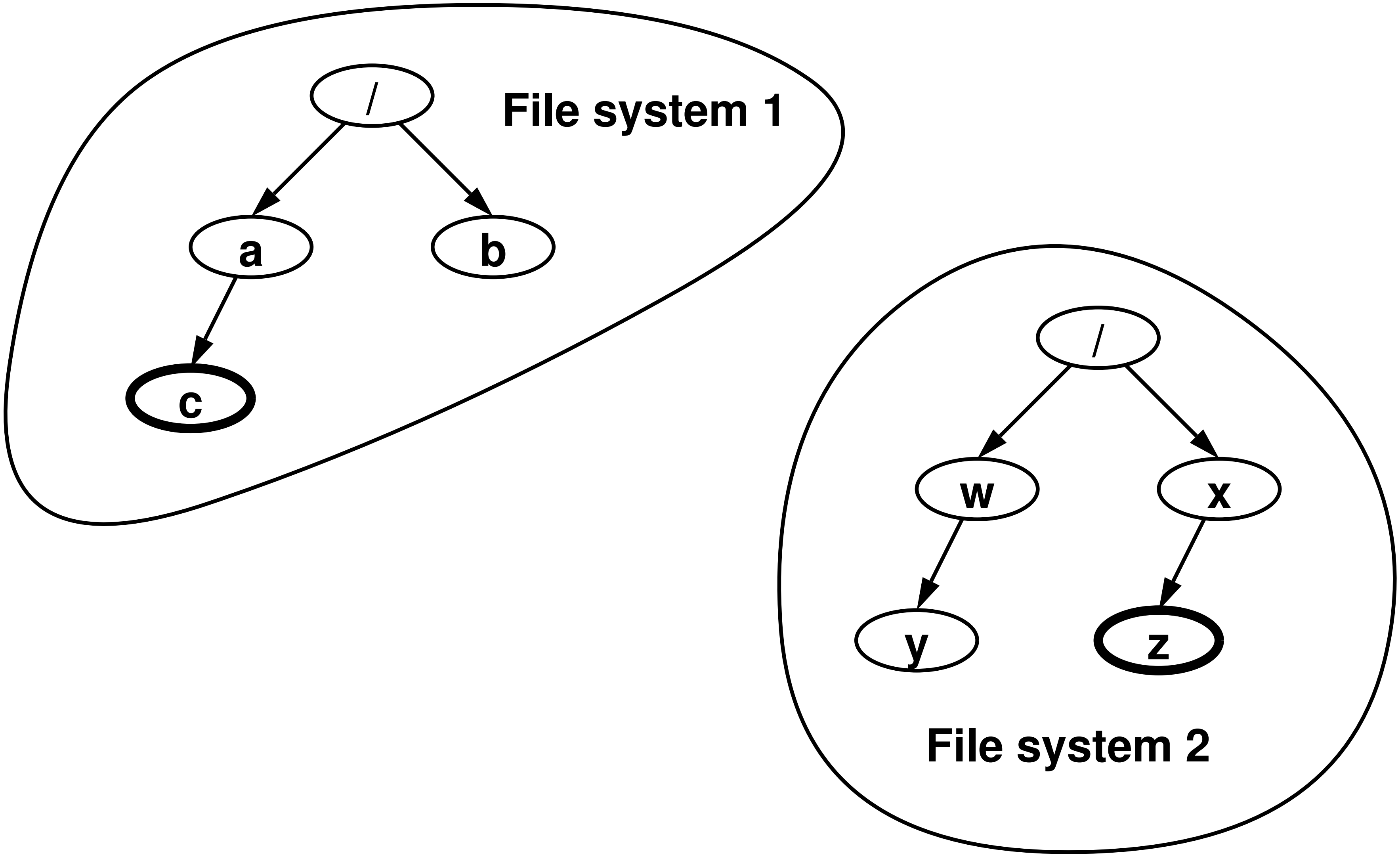


B+ Trees (part 3)

- ***Deletion:*** **e.g., "Otto"**
  - **merge nodes if necessary**
    - **may decrease depth**



Name-Space Management



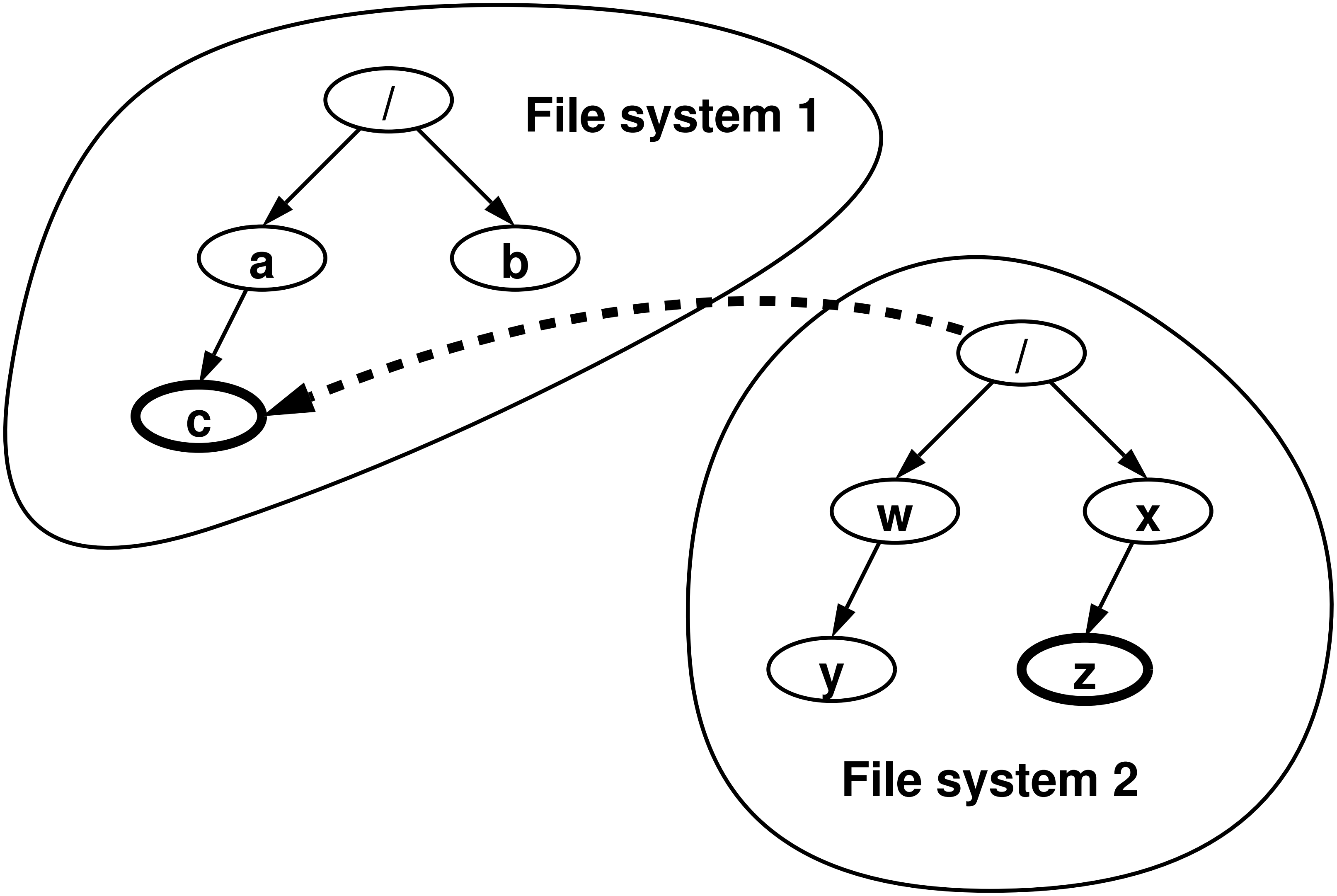
Name-Space Management

- **Multiple partitions of hard drive**
  - on Windows, you have C:, D:, etc.
  - on Unix, you have /dev/sda0 (ext2), /dev/sda1 (ext3), etc.

- **Main challenge for name-space management**
  - how do you make the name-space appear uniform?
    - Windows: drives
    - Unix: file system *mounting*
      - create an entry in the inode to point to the file system
      - only happen in *memory* and not on disk

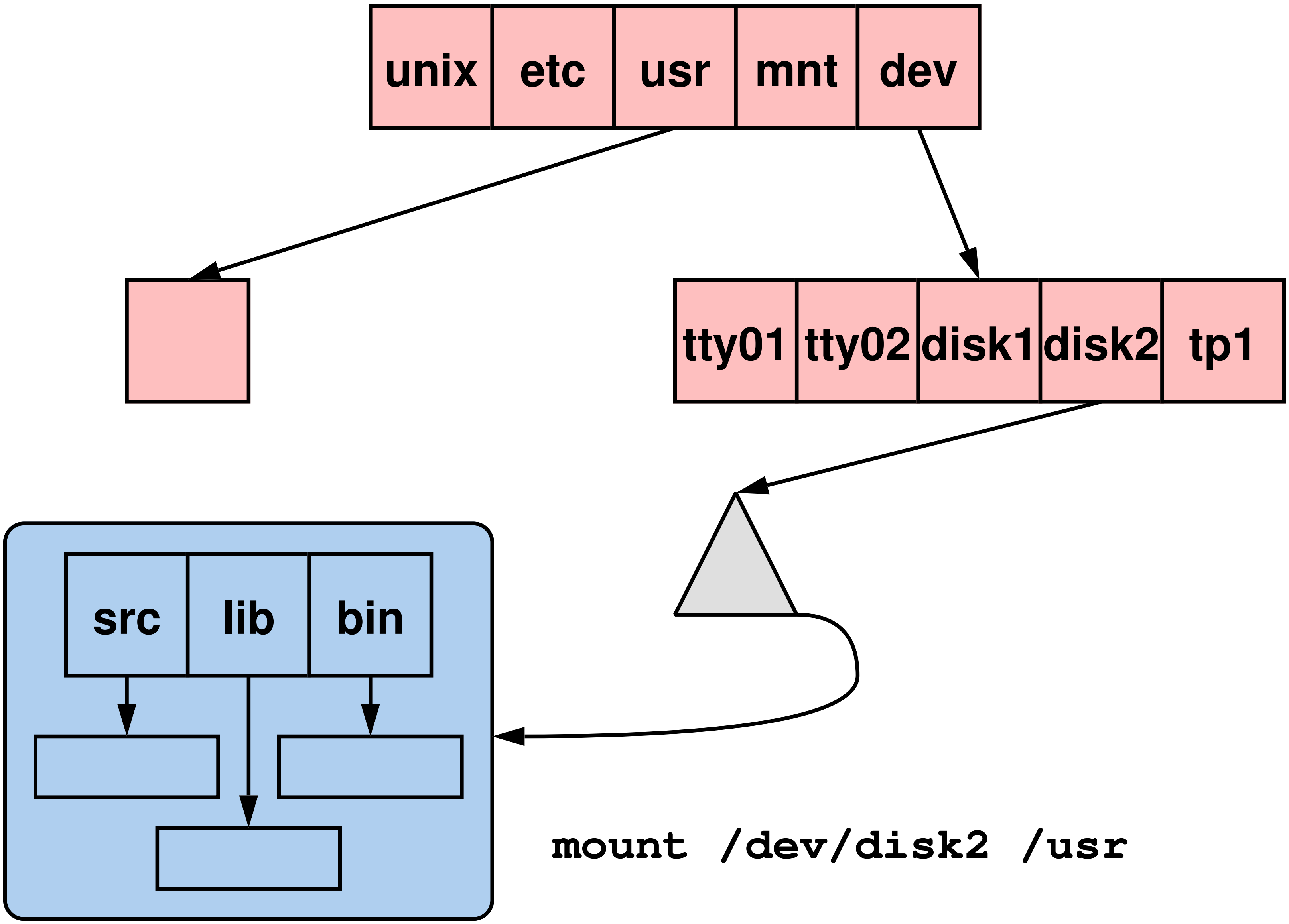
Unix Name-Space Management

- Unix: file system *mounting*
  - mount one file system on a *directory* of another file system



Mount Points

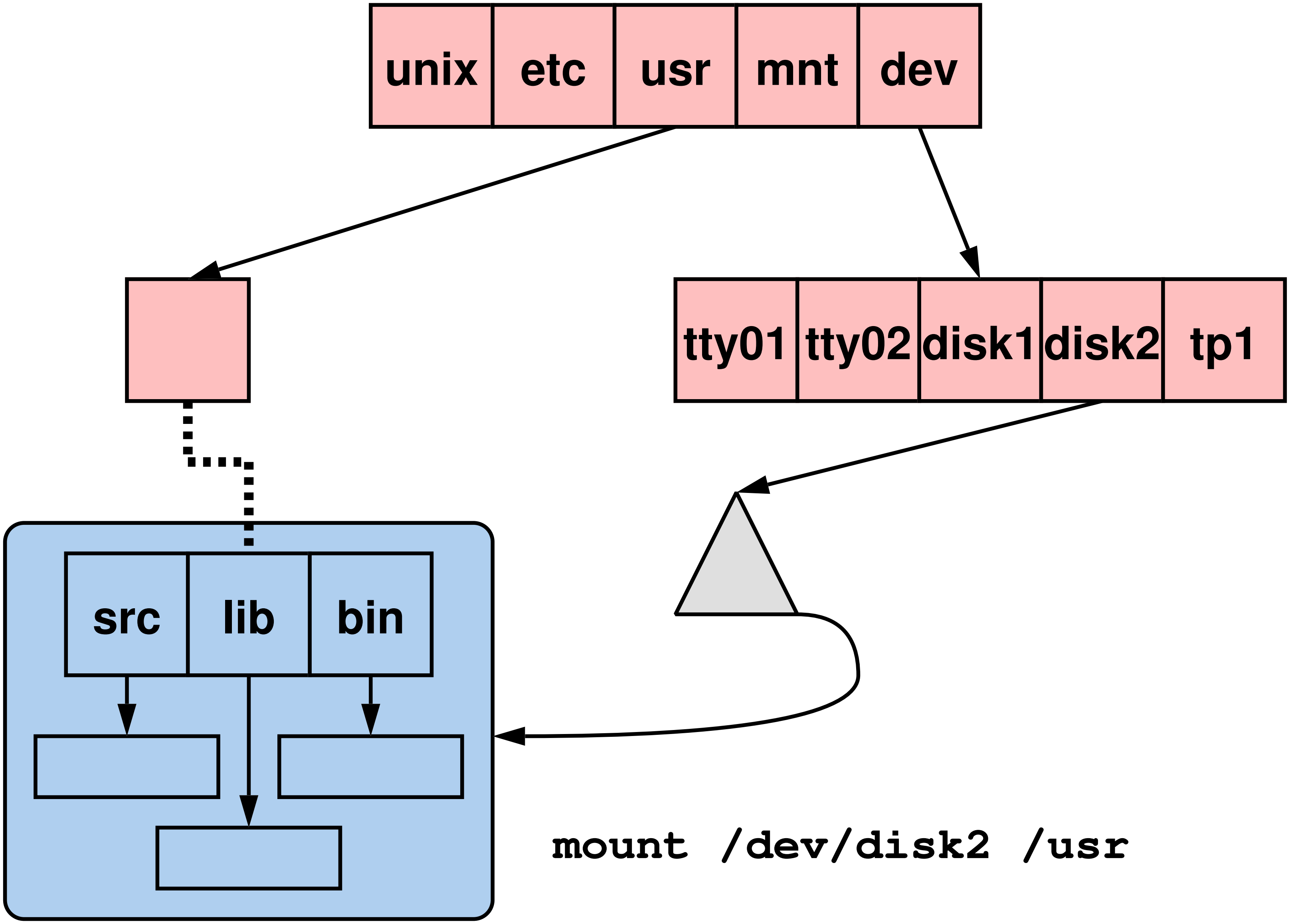
- A directory can be a *mount point*




```
mount /dev/disk2 /usr
```

Mount Points

- A directory can be a *mount point*



`mount /dev/disk2 /usr`

File System Mounting

- `mount /dev/disk2 /usr`
  - `/usr/bin/bash` is to be located at `/dev/disk2/bin/bash`
- `mount /dev/usb0 /mnt/usb`
  - on Mac OS X, instead of `/mnt`, it uses `/Volumes`

Weenix

- Try `MOUNTING=1` in `Config.mk`
  - may be after the semester is over
  - polymorphism
- It should be clear that mouting a file system is not the same as using an *alias* (i.e., string substitution)