Database Indexing Overview

ΠΜΣ "Ερευνητικές Κατευθύνσεις στην Πληροφορική"

Επεξεργασία και Ανάλυση Δεδομένων

SPRING SEMESTER 2020

Material taken from 15-415 - Database Applications class @Carnegie Mellon

C. Faloutsos

Indexing- overview

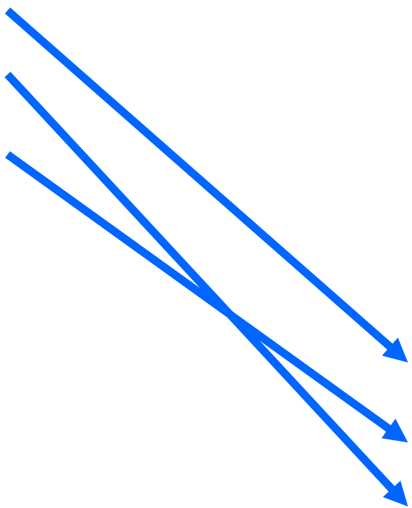
- primary / secondary indices
- index-sequential (ISAM)
- B - trees, B+ - trees
- Hashing

Indexing

- once the records are stored in a file, how do you search efficiently?
- brute force: retrieve all records, report the qualifying ones
- better: use indices (pointers) to locate the records directly

Indexing – main idea:

123
125
234



STUDENT		
Ssn	Name	Address
123	smith	main str
234	jones	forbes ave
125	tomson	main str

Measuring ‘goodness’

- range queries?
- retrieval time?
- insertion / deletion?
- space overhead?
- reorganization?

Main concepts

- search keys are sorted in the index file and point to the actual records
- primary vs. secondary indices
- Clustering (sparse) vs non-clustering (dense) indices

Indexing

Primary key index: on primary key (no duplicates)

123
234
345
456
567

STUDENT

<u>Ssn</u>	Name	Address
123	smith	main str
234	jones	forbes ave
678	tomson	main str
456	stevens	forbes ave
345	smith	forbes ave

Indexing

secondary key index: duplicates may exist

forbes ave	
main str	

Address-index

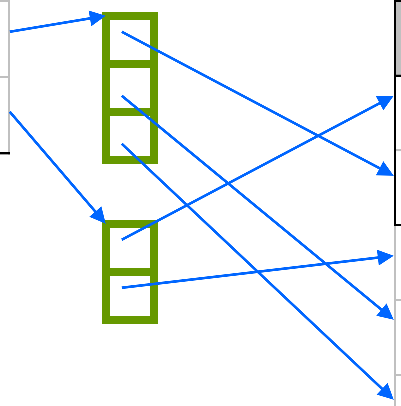
STUDENT		
<u>Ssn</u>	Name	Address
123	smith	main str
234	jones	forbes ave
345	tomson	main str
456	stevens	forbes ave
567	smith	forbes ave

Indexing

secondary key index: typically, with 'postings lists'

forbes ave
main str

Postings lists



STUDENT		
Ssn	Name	Address
123	smith	main str
234	jones	forbes ave
345	tomson	main str
456	stevens	forbes ave
567	smith	forbes ave

Main concepts – cont'd

- Clustering (= sparse) index: records are physically sorted on that key (and not all key values are needed in the index)
- Non-clustering (=dense) index: the opposite
- E.g.:

Indexing

Clustering/sparse index on ssn

123
456
…

>=123

>=456

STUDENT

Ssn	Name	Address
123	smith	main str
234	jones	forbes ave
345	tomson	main str
456	stevens	forbes ave
567	smith	forbes ave

Indexing

Non-clustering / dense index

123
234
345
456
567

Ssn	Name	Address
345	tomson	main str
234	jones	forbes ave
567	smith	forbes ave
456	stevens	forbes ave
123	smith	main str

Summary

- **All combinations are possible**

	Dense	Sparse
Primary		usual
secondary	usual	rare

- **at most one sparse/clustering index**
- **as many as desired dense indices**
- **usually: one primary-key index (maybe clustering) and a few secondary-key indices (non-clustering)**

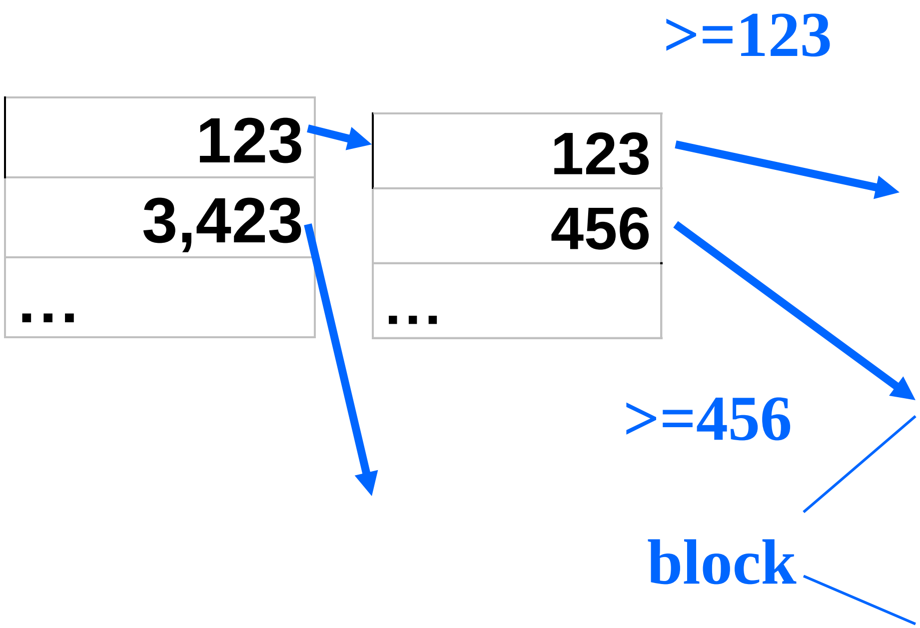
Indexing- overview

- primary / secondary indices
- index-sequential (ISAM)
- B - trees, B+ - trees
- hashing
  - static hashing
  - dynamic hashing

ISAM

- What if index is too large to search sequentially?

ISAM



STUDENT

Ssn	Name	Address
123	smith	main str
234	jones	forbes ave
345	tomson	main str
456	stevens	forbes ave
567	smith	forbes ave

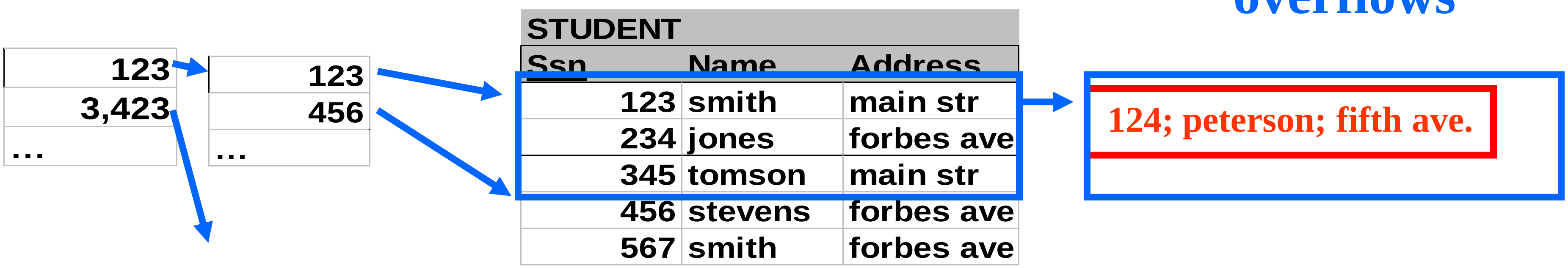
ISAM - observations

- if index is too large, store it on disk and keep index-on-the-index
- usually two levels of indices, one first- level entry per disk block (why?)

ISAM - observations

- What about insertions/deletions?

overflows



- **overflow chains may become very long - thus:**
  - **shut-down & reorganize**
  - **start with ~80% utilization**

So far

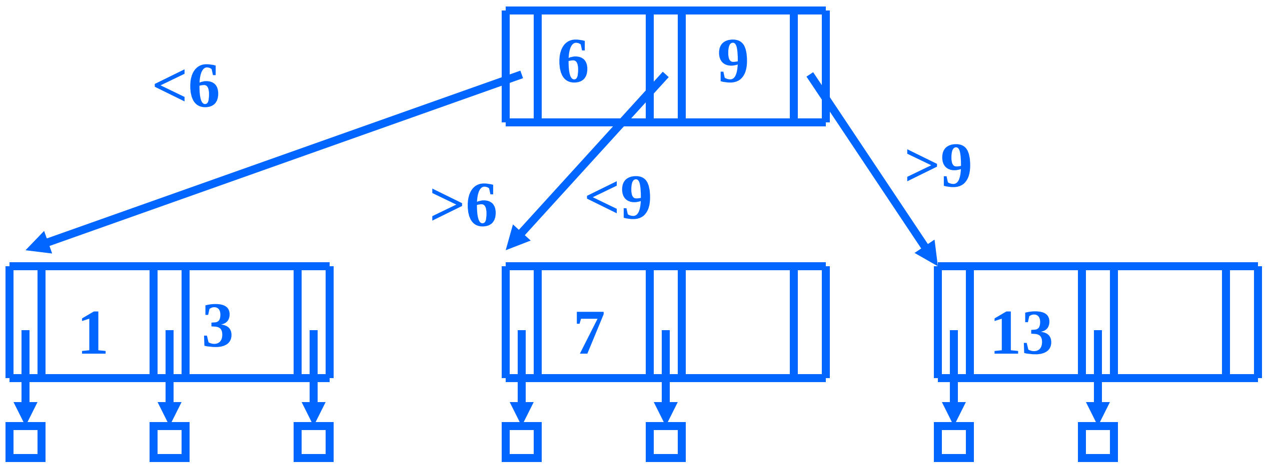
- … indices (like ISAM) suffer in the presence of frequent updates
- alternative indexing structure: **B - trees**

B-trees

- the **most successful** family of index schemes (B-trees, B^{+}-trees, B^{*}-trees)
- Can be used for primary/secondary, clustering/non-clustering index.
- balanced "n-way" search trees

B-trees

Eg., B-tree of order 3:

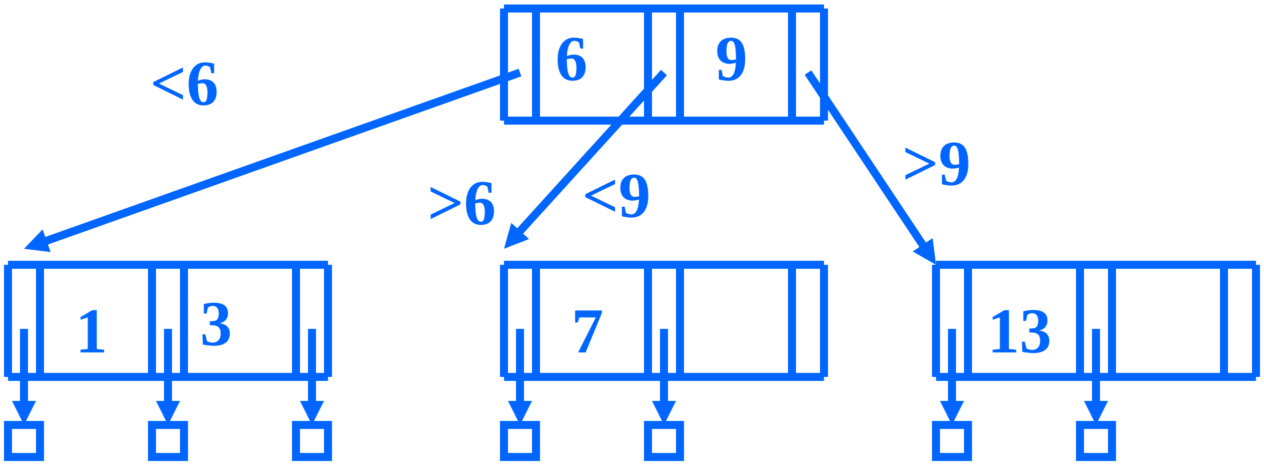


Properties

- “block aware” nodes: each node -> disk page
- O(log (N)) for everything! (ins/del/search)
- typically, if m = 50 - 100, then 2 - 3 levels
- utilization >= 50%, guaranteed; on average 69%

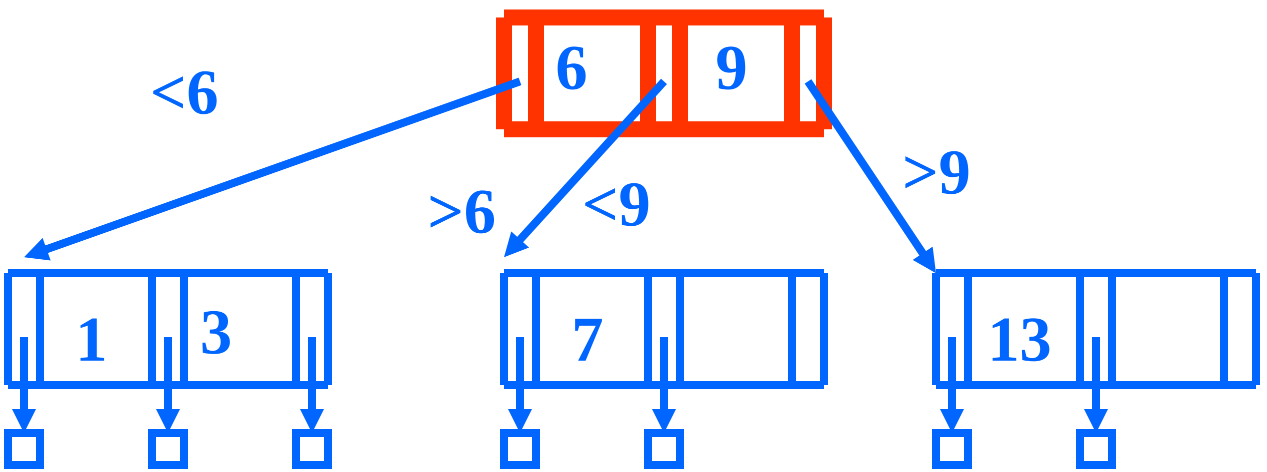
Queries

- Algo for exact match query? (eg., ssn=8?)



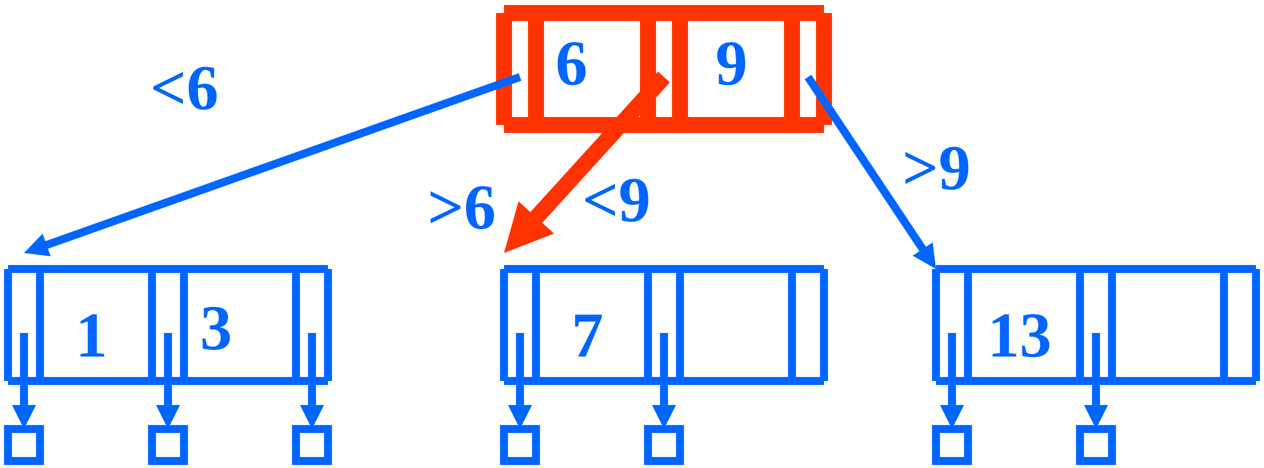
Queries

- Algo for exact match query? (eg., ssn=8?)



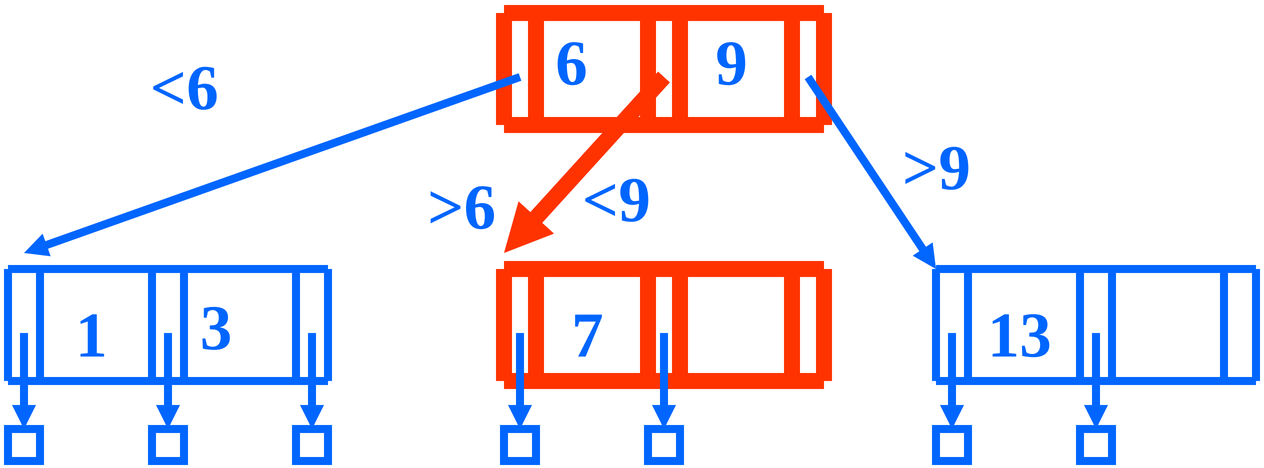
Queries

- Algo for exact match query? (eg., ssn=8?)



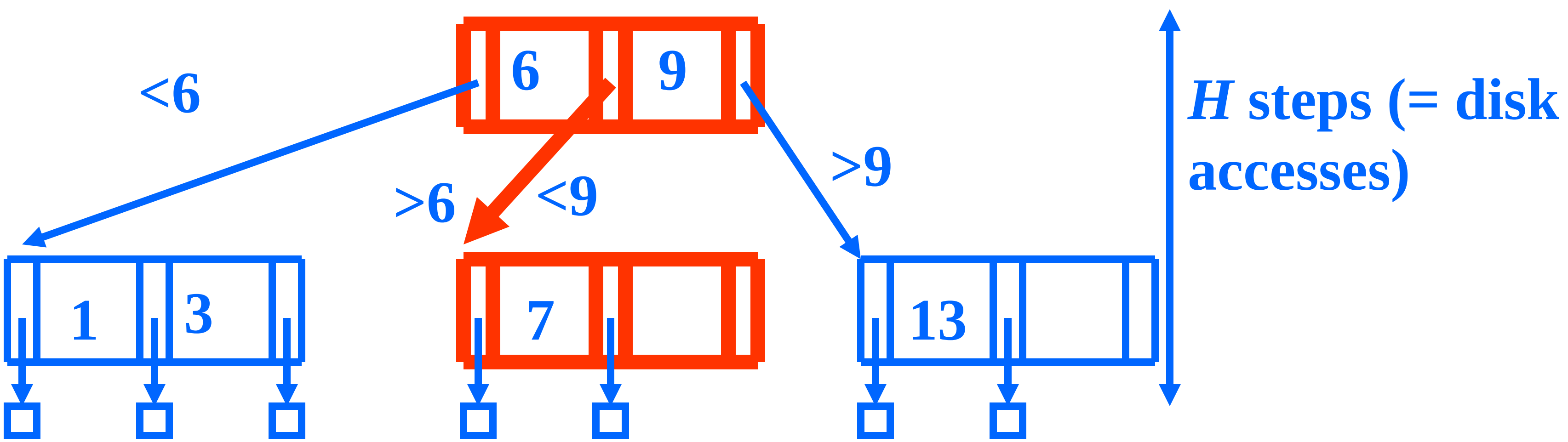
Queries

- Algo for exact match query? (eg., ssn=8?)



Queries

- Algo for exact match query? (eg., ssn=8?)

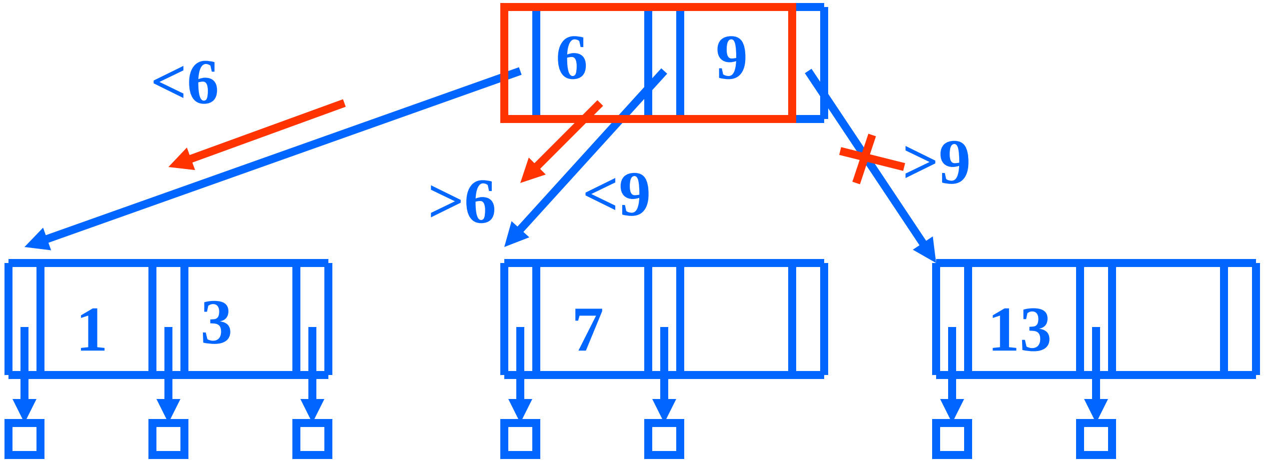


Queries

- what about range queries? (eg., $5 < salary < 8$)
- Proximity/ nearest neighbor searches? (eg., $salary \sim 8$)

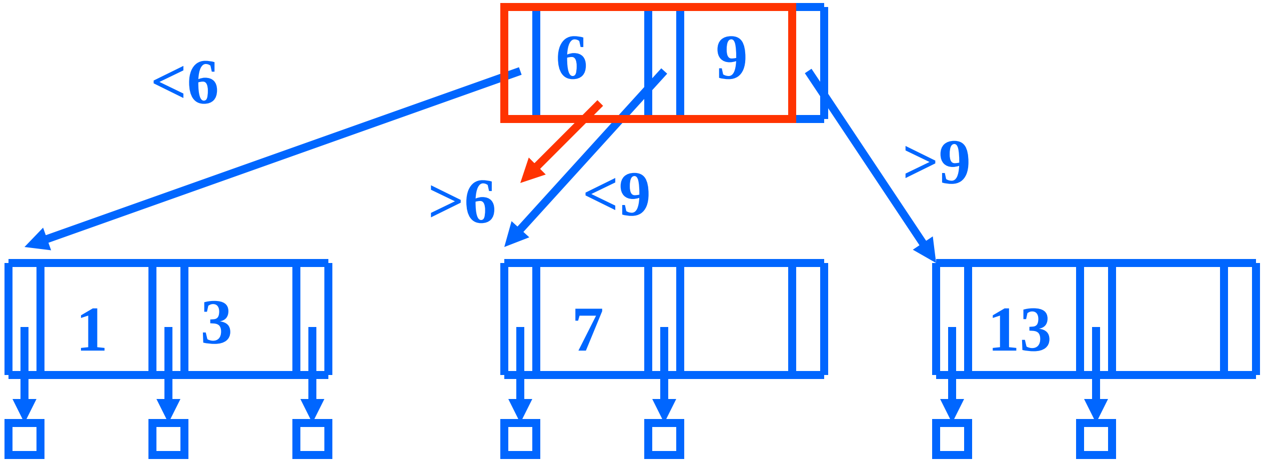
Queries

- what about range queries? (eg., ***5<salary<8***)
- Proximity/ nearest neighbor searches? (eg., *salary ~ 8*)



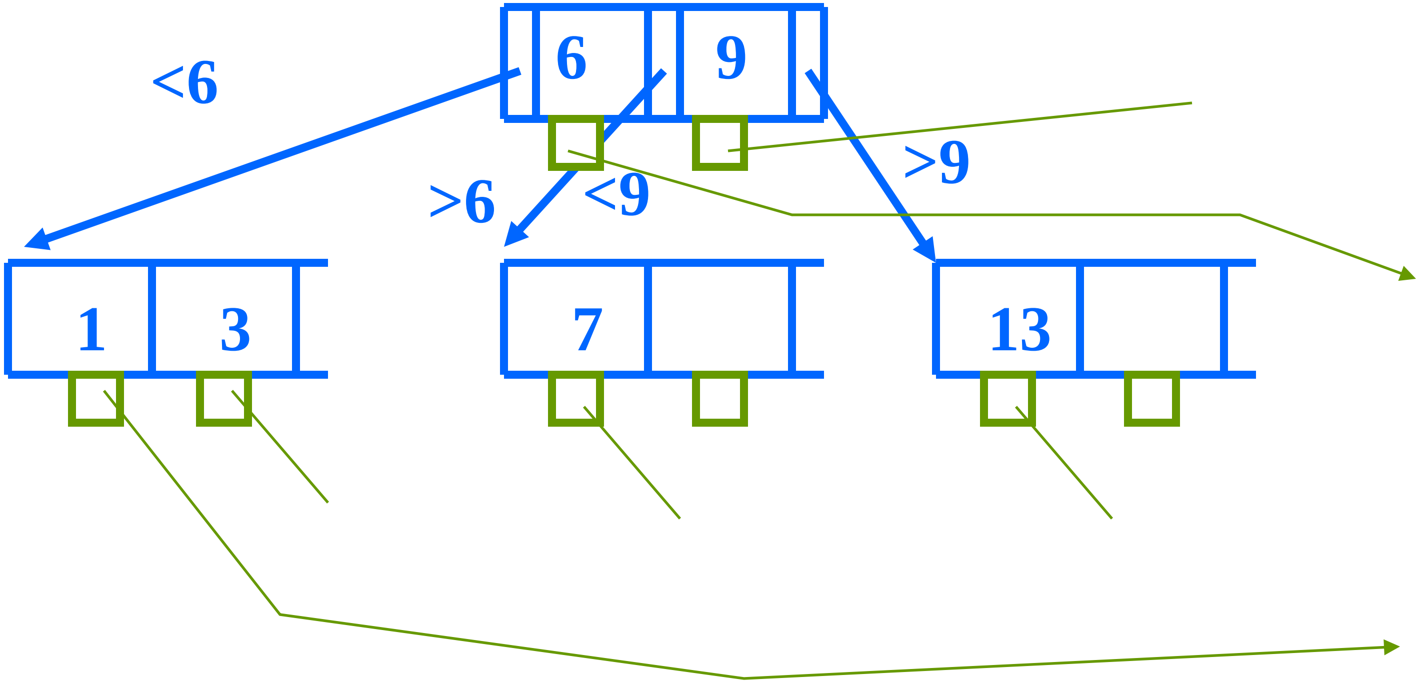
Queries

- what about range queries? (eg., *5<salary<8*)
- Proximity/ nearest neighbor searches? (eg., ***salary ~ 8***)



B-trees in practice

In practice:

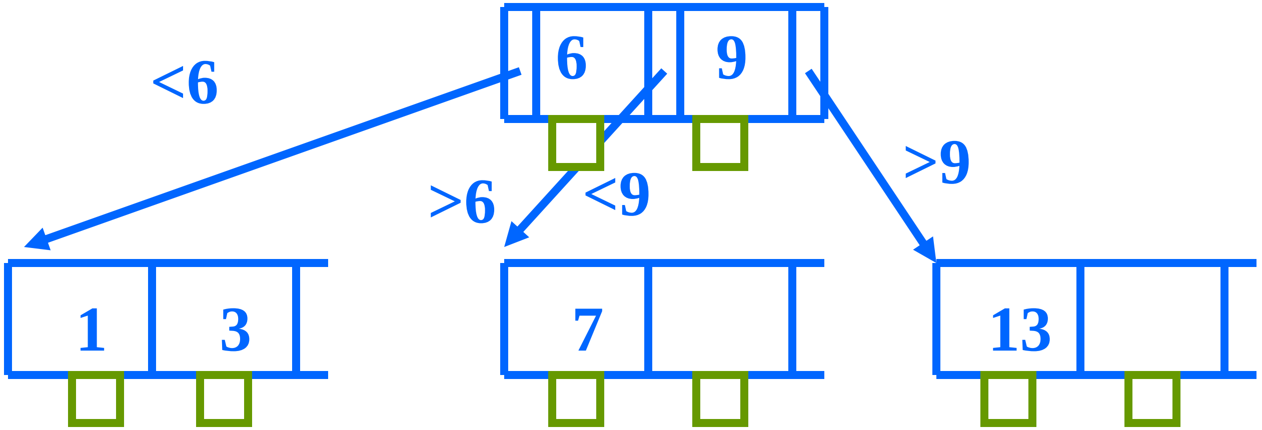


Ssn	……	
3		
7		
6		
9		
1		

B-trees in practice

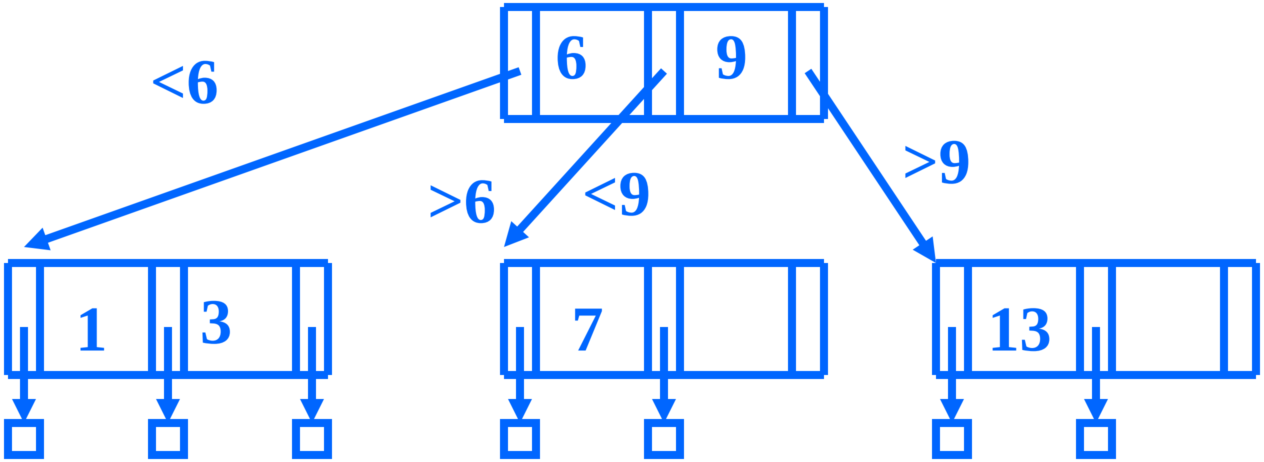
In practice, the formats are:

- leaf nodes: (v1, rp1, v2, rp2, … vn, rpn)
- Non-leaf nodes: (p1, v1, rp1, p2, v2, rp2, …)



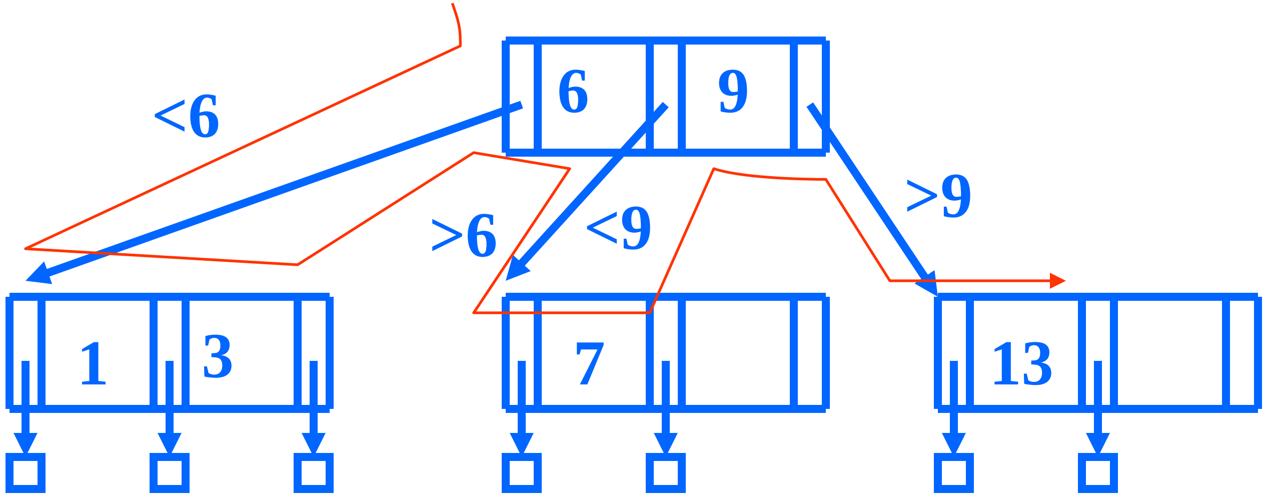
B+ trees - Motivation

B-tree – print keys in sorted order:



B+ trees - Motivation

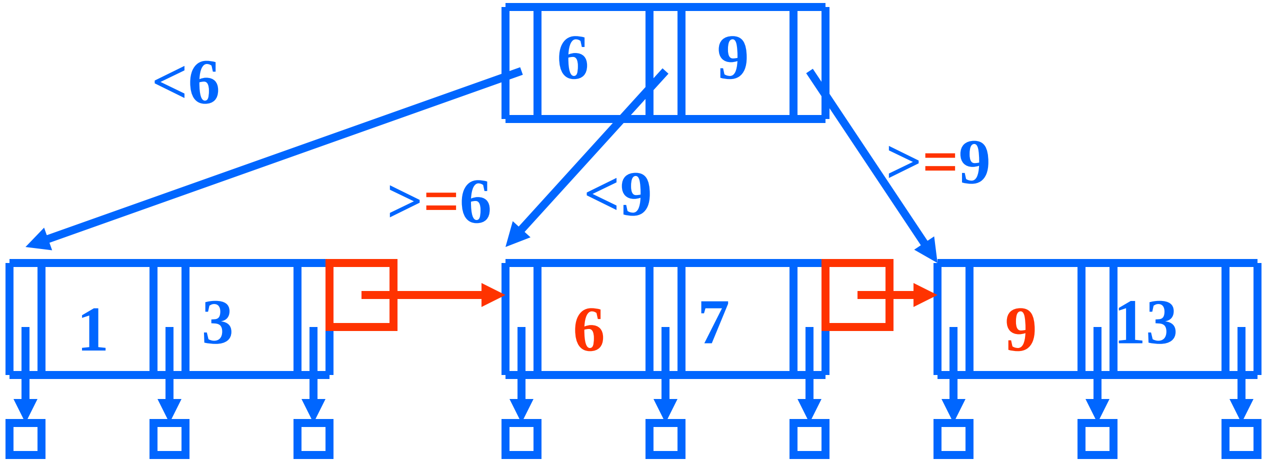
B-tree needs back-tracking – how to avoid it?



Solution: B^+ - trees

- facilitate sequential ops
- They string all leaf nodes together
- AND
- replicate keys from non-leaf nodes, to make sure every key appears at the leaf level

B+ trees



Conclusions

- all B – tree variants can be used for any type of index: primary/secondary, sparse (clustering), or dense (non-clustering)
- All have excellent, O(logN) worst-case performance for ins/del/search
- It's the prevailing indexing method

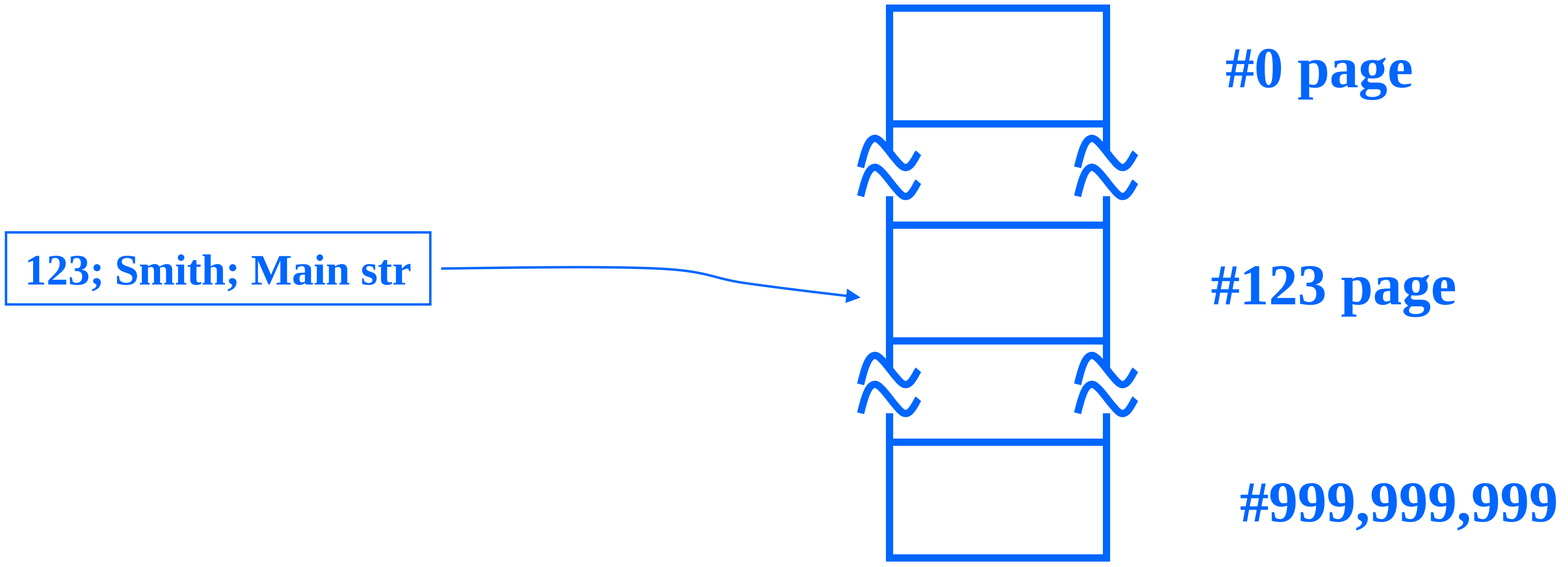
(Static) Hashing

Problem: "*find EMP record with ssn=123*"

What if disk space was free, and time was at premium?

Hashing

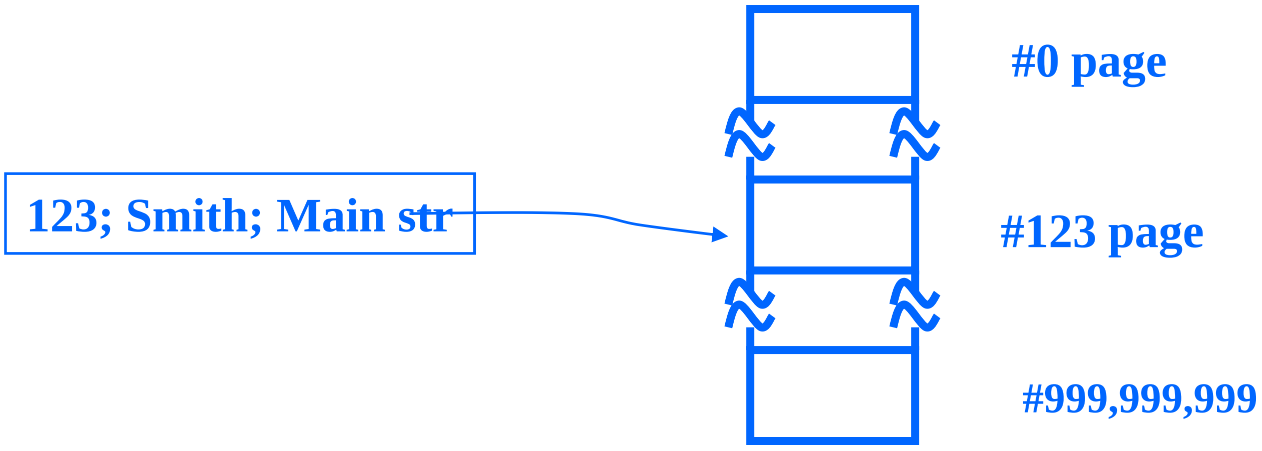
A: Brilliant idea: key-to-address transformation:



Hashing

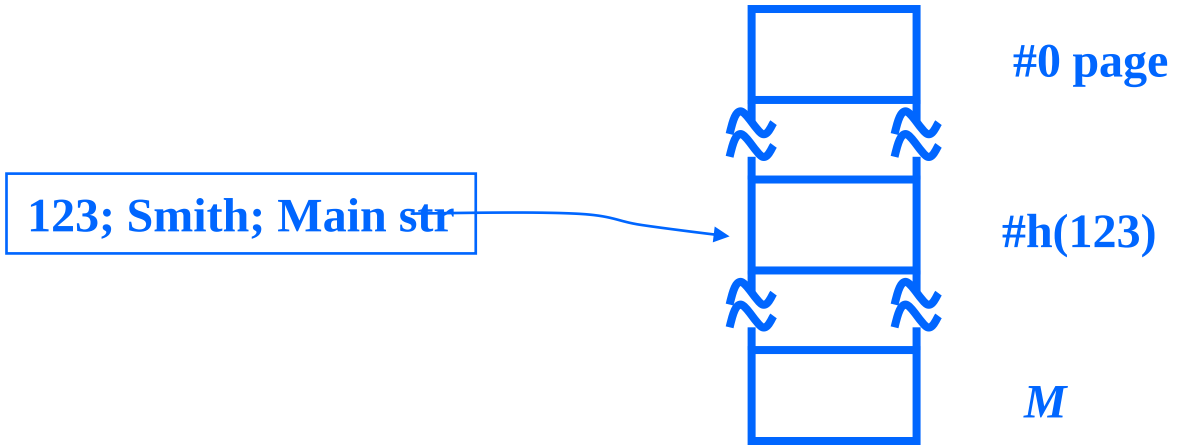
Since space is NOT free:

- use *M*, instead of 999,999,999 slots
- hash function: $h(key) = slot\text{-}id$



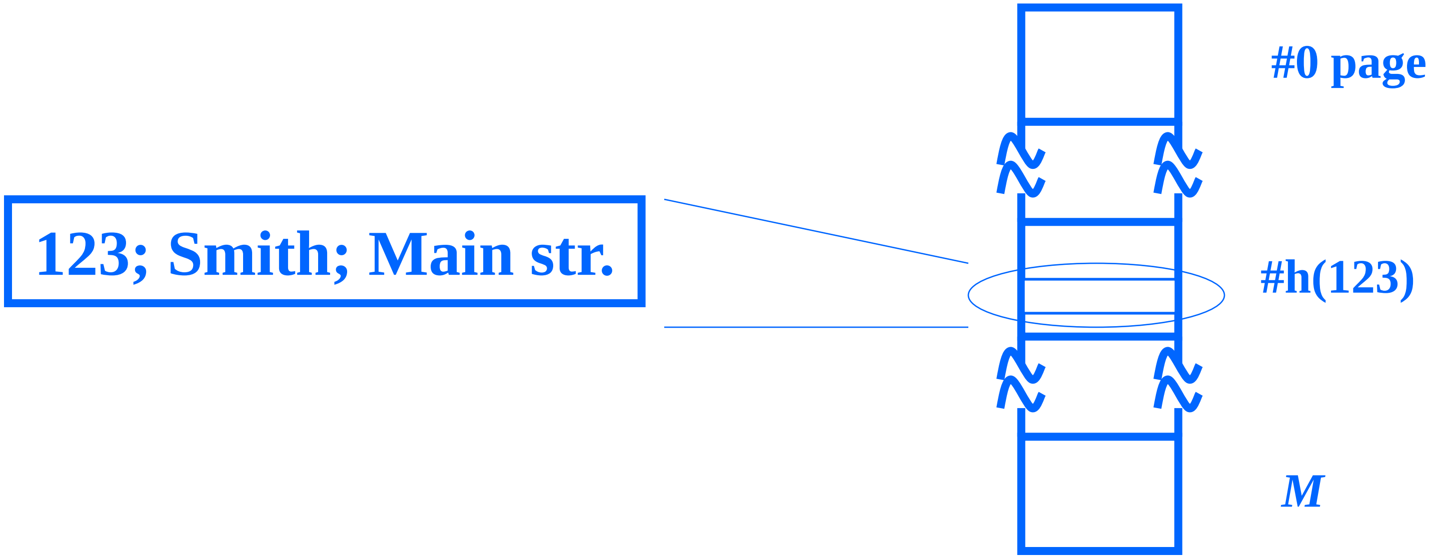
Hashing

Typically: each hash bucket is a page, holding many records:



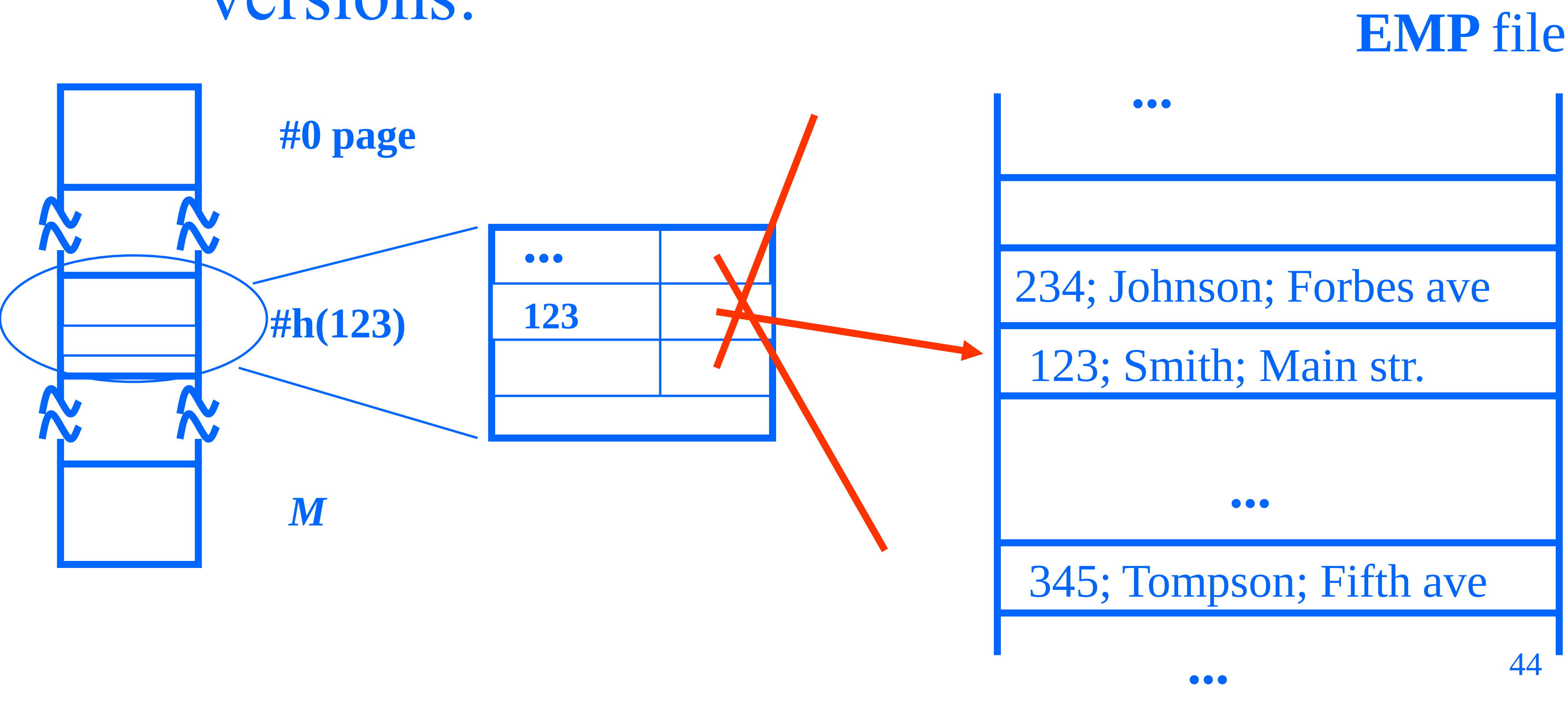
Hashing

Notice: could have **clustering**, or non-clustering versions:



Hashing

Notice: could have clustering, or **non-clustering** versions:



Design decisions

1) formula *h()* for hashing function
2) size of hash table *M*
3) collision resolution method

Design decisions - functions

- Goal: **uniform** spread of keys over hash buckets
- Popular choices:
  - Division hashing
  - Multiplication hashing

Division hashing

$$h(x) = (a*x+b) \mod M$$

- eg., $h(ssn) = (ssn) \mod 1{,}000$
  - gives the last three digits of ssn
- M: size of hash table - choose a prime number, defensively (why?)

Division hashing

- eg., M=2; hash on driver-license number (dln), where last digit is 'gender' (0/1 = M/F)
- in an army unit with predominantly male soldiers
- Thus: avoid cases where M and keys have common divisors - prime M guards against that!

Size of hash table

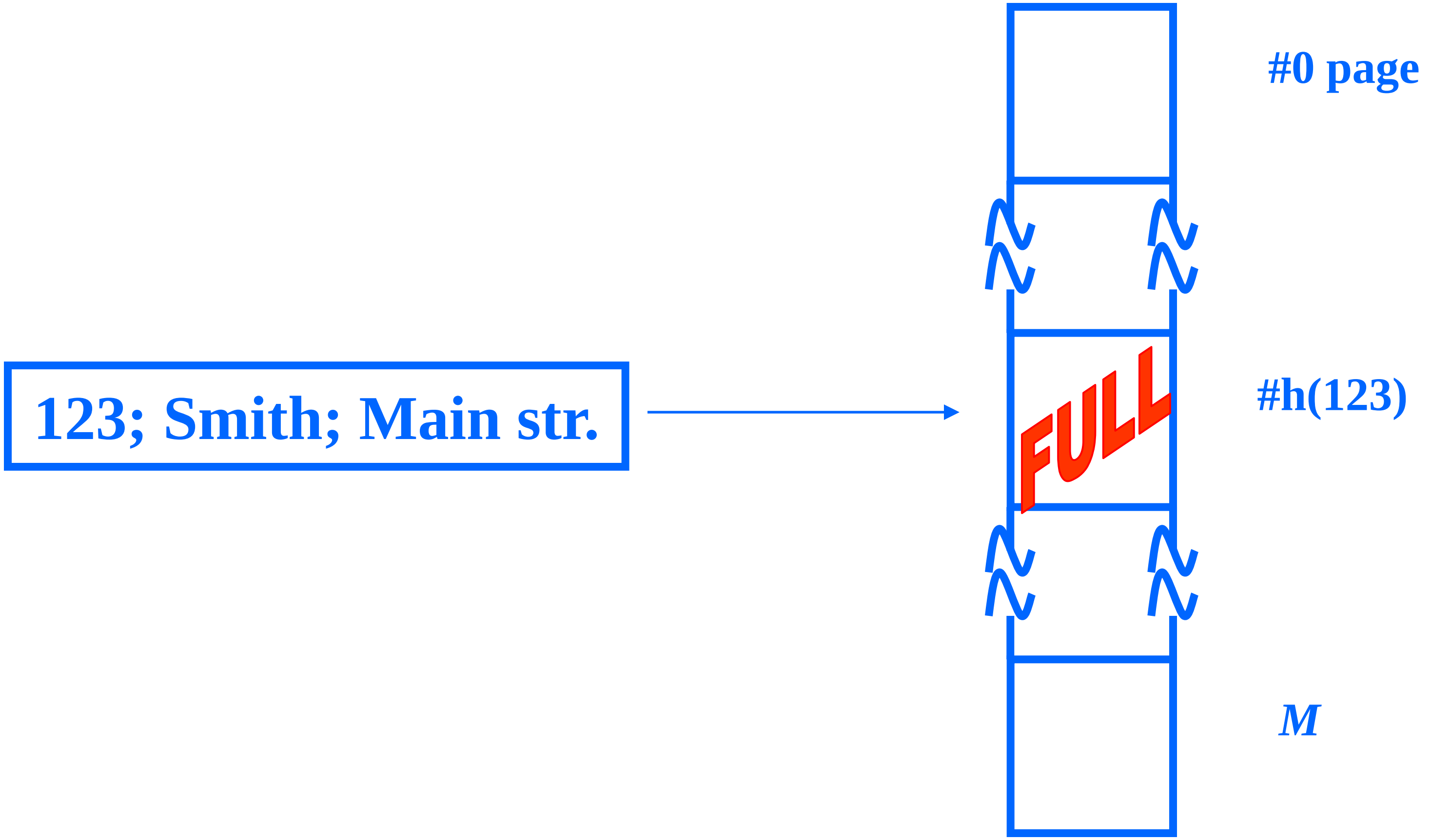
- eg., 50,000 employees, 10 employee-records / page
- Q: *M*=?? pages/buckets/slots

Size of hash table

- eg., 50,000 employees, 10 employees/page
- Q: M=?? pages/buckets/slots
- A: utilization ~ 90% and
  - M: prime number

Eg., in our case: M= closest prime to 50,000/10 / 0.9 = 5,555

Collision resolution

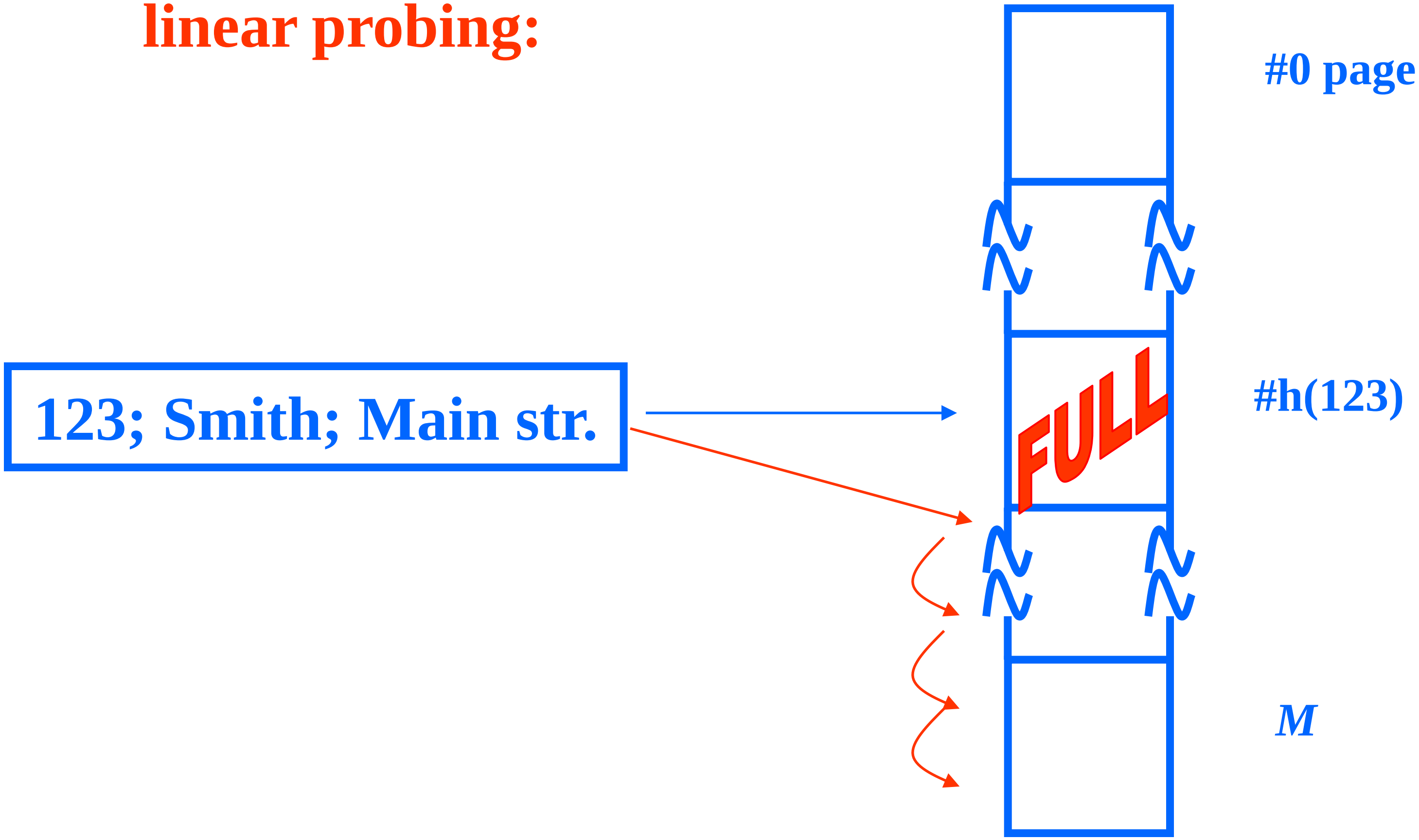


Collision resolution

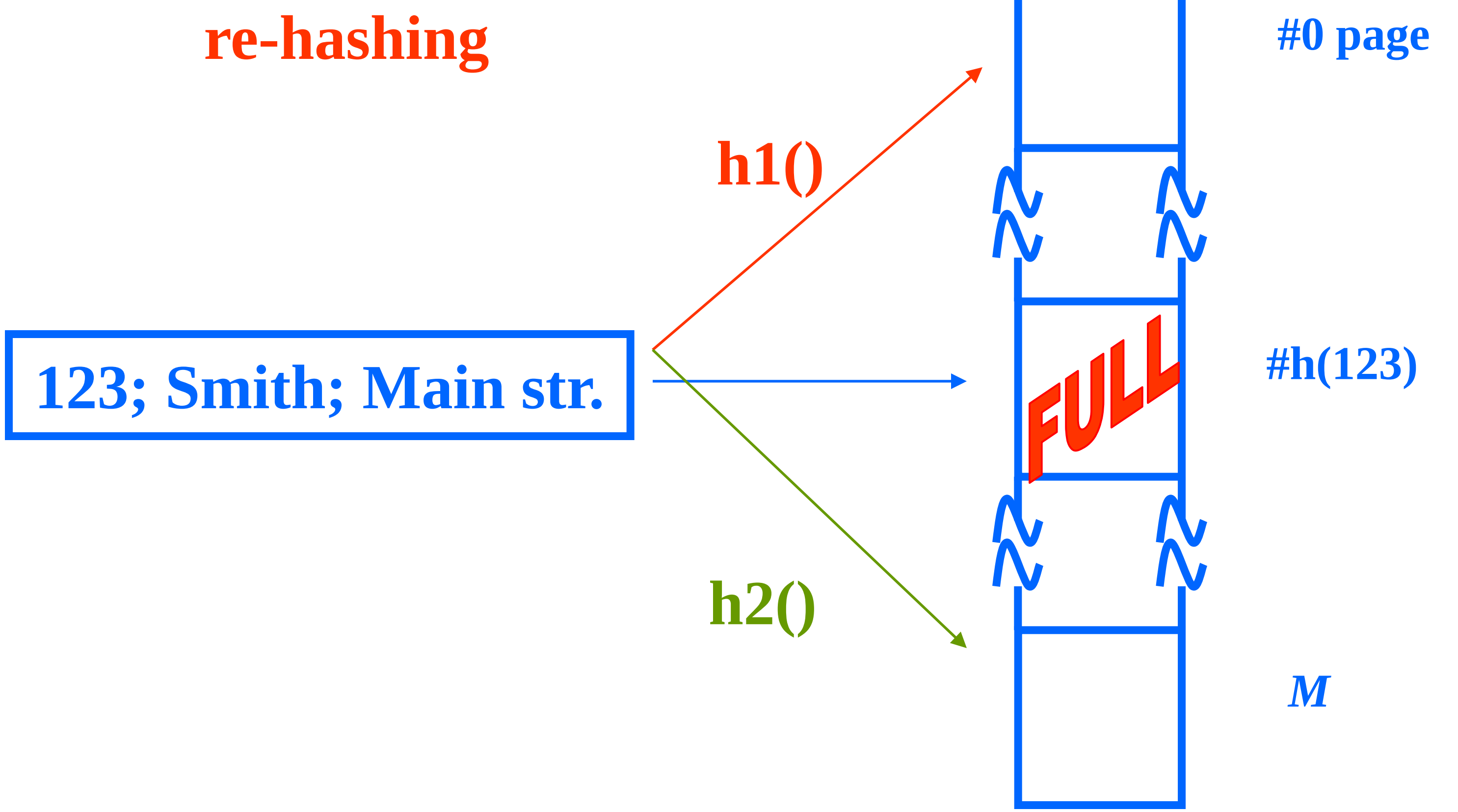
- Q: what is a ‘collision’?
- A: ??
- Q: why worry about collisions/overflows? (recall that buckets are ~90% full)
- A: ‘birthday paradox’

Collision resolution

linear probing:



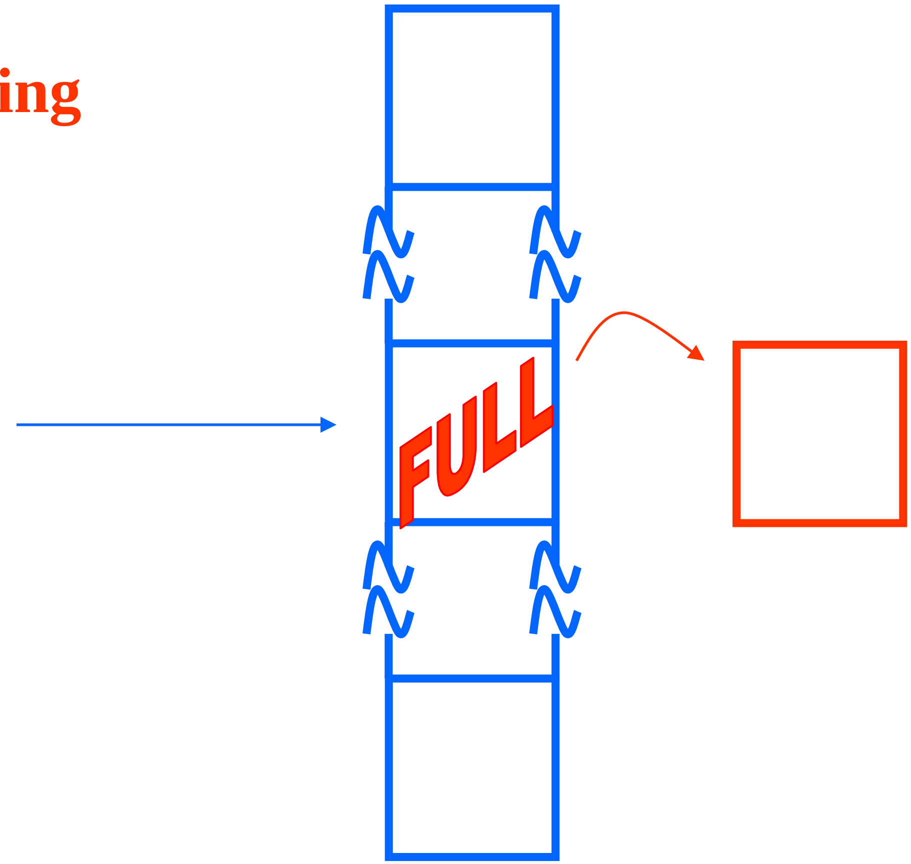
Collision resolution



Collision resolution

separate chaining

123; Smith; Main str.



Design decisions - conclusions

- function: division hashing
  - $h(x) = (a*x+b) \bmod M$
- size *M*: ~90% util.; prime number.
- collision resolution: separate chaining
  - easier to implement (deletions!);
  - no danger of becoming full

Hashing vs B-trees:

Hashing offers

- speed ! (O(1) **avg**. search time)

..but:

Hashing vs B-trees:

..but B-trees give:

- key ordering:
  - **range queries**
  - **proximity queries**
  - **sequential scan**
- O(log(N)) guarantees for search, ins./del.
- graceful growing/shrinking