Data on Disk and B+-trees

Storage Media: Players

- **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- **Main memory**:
  - fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
  - generally too small (or too expensive) to store the entire database
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.
  - But… CPU operates only on data in main memory

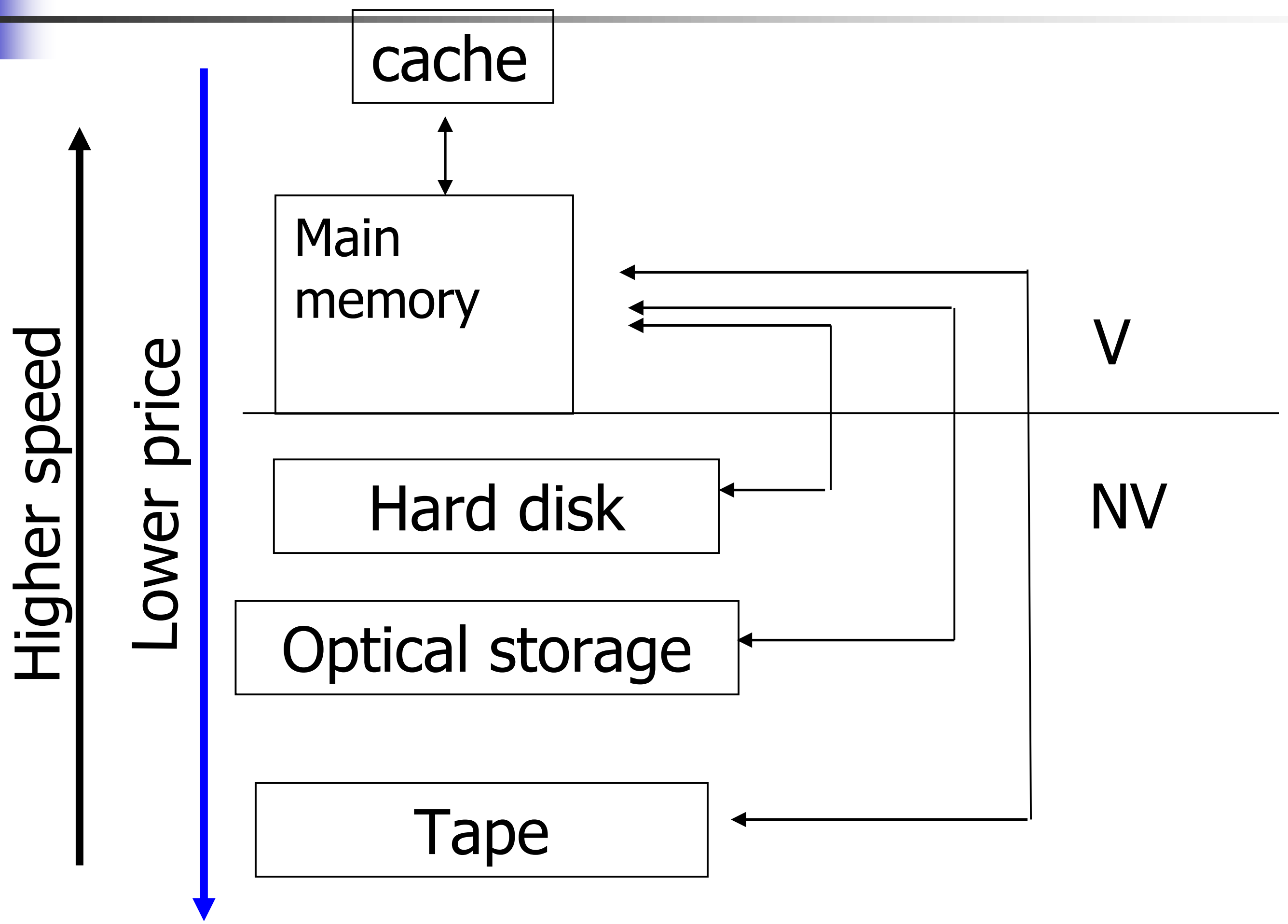
Storage Media: Players

- **Disk**
  - Primary medium for the long-term storage of data; typically stores entire database.
  - random-access – possible to read data on disk in any order, unlike magnetic tape
  - Non-volatile: data survive a power failure or a system crash, disk failure less likely than them
- New technology: Solid State Disks and Flash disks

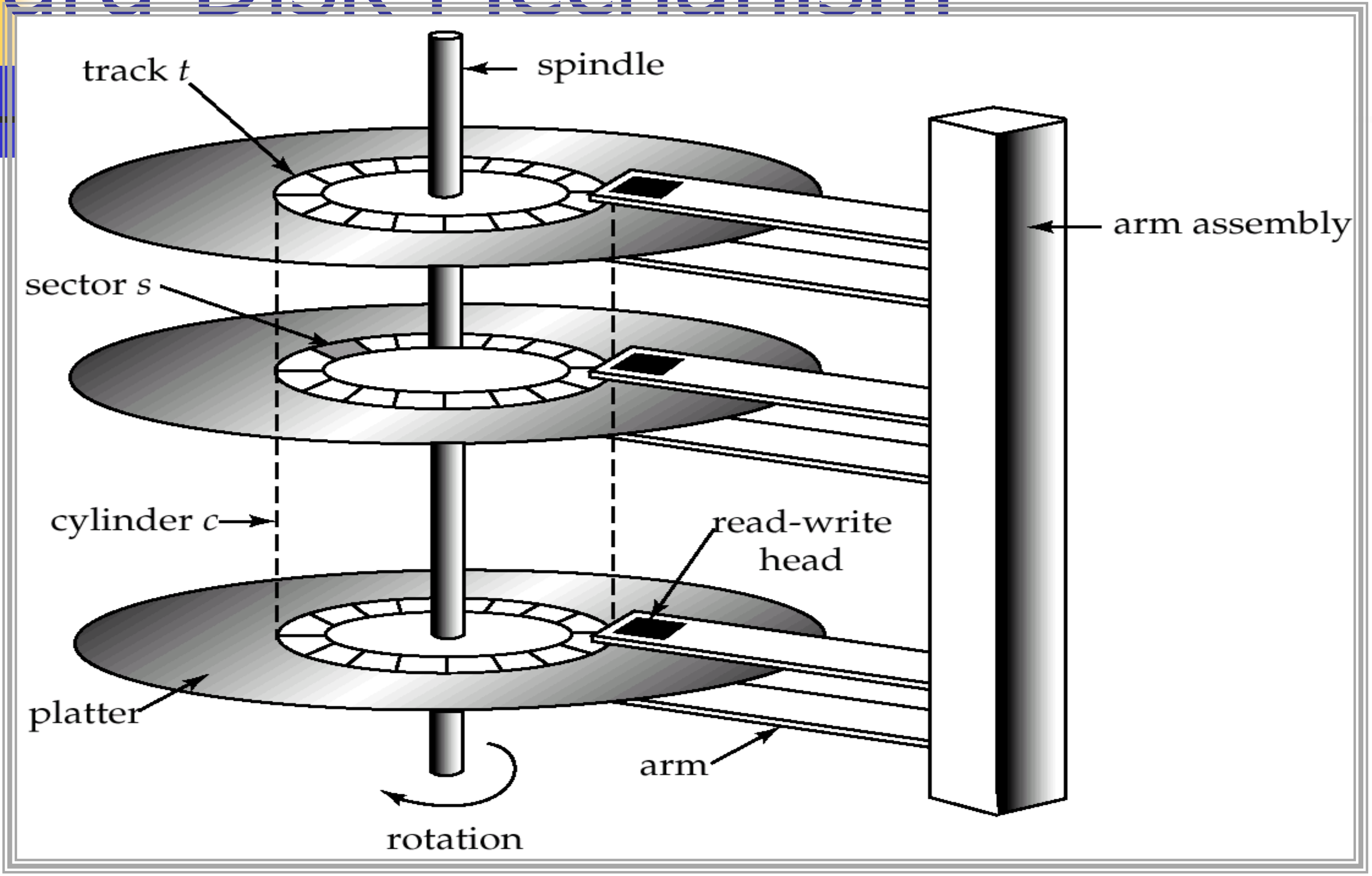
Storage Media: Players

- **Optical storage**
  - non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R and DVD-R)
  - Multiple write versions also available (CD-RW, DVD-RW, and DVD-RAM)
  - Reads and writes are slower than with magnetic disk
- **Tapes**
  - Sequential access (very slow)
  - Cheap, high capacity

Memory Hierarchy



Hard Disk Mechanism



- **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
- Surface of platter divided into circular **tracks**
- Each track is divided into **sectors**.
  - A sector is the smallest unit of data that can be read or written.
- To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- **Block: a sequence of sectors**
- **Cylinder** *i* consists of *i*th track of all the platters

“Typical” Values

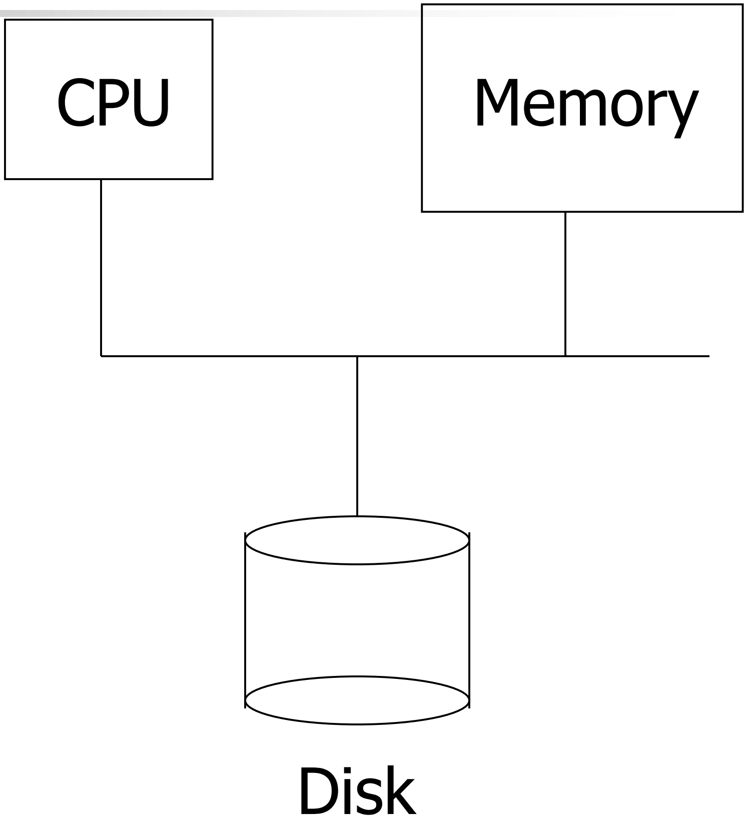
Diameter:	1 inch $\rightarrow$ 10 inches
Cylinders:	100 $\rightarrow$ 2000
Surfaces:	1 or 2
(Tracks/cyl)	2 $\rightarrow$ 30
Sector Size:	512B $\rightarrow$ 50K
Capacity:	4 MB $\rightarrow$ 8 TB

Data Access

- Data Access Method (aka Index)
  - data structure that allows efficient access of data stored on hard disk for specific queries (query types)
- Query types:
  - Exact match
  - Range Query
  - …. more
- With New Data types and Applications =>
  - New Indexing and Query algorithms!!
  - New Systems!!!

Model of Computation

- Data stored on disk(s)
- Minimum transfer unit: a page = b bytes or B records (or block)
- N records -> N/B = n pages
- I/O complexity: in number of pages



I/O complexity

- An ideal index has space $O(N/B)$, update overhead $O(1)$ or $O(\log_B(N/B))$ and search complexity $O(\alpha/B)$ or $O(\log_B(N/B) + \alpha/B)$ where α is the number of records in the answer
- But, sometimes CPU performance is also important… minimize cache misses -> don't waste CPU cycles

B+-tree

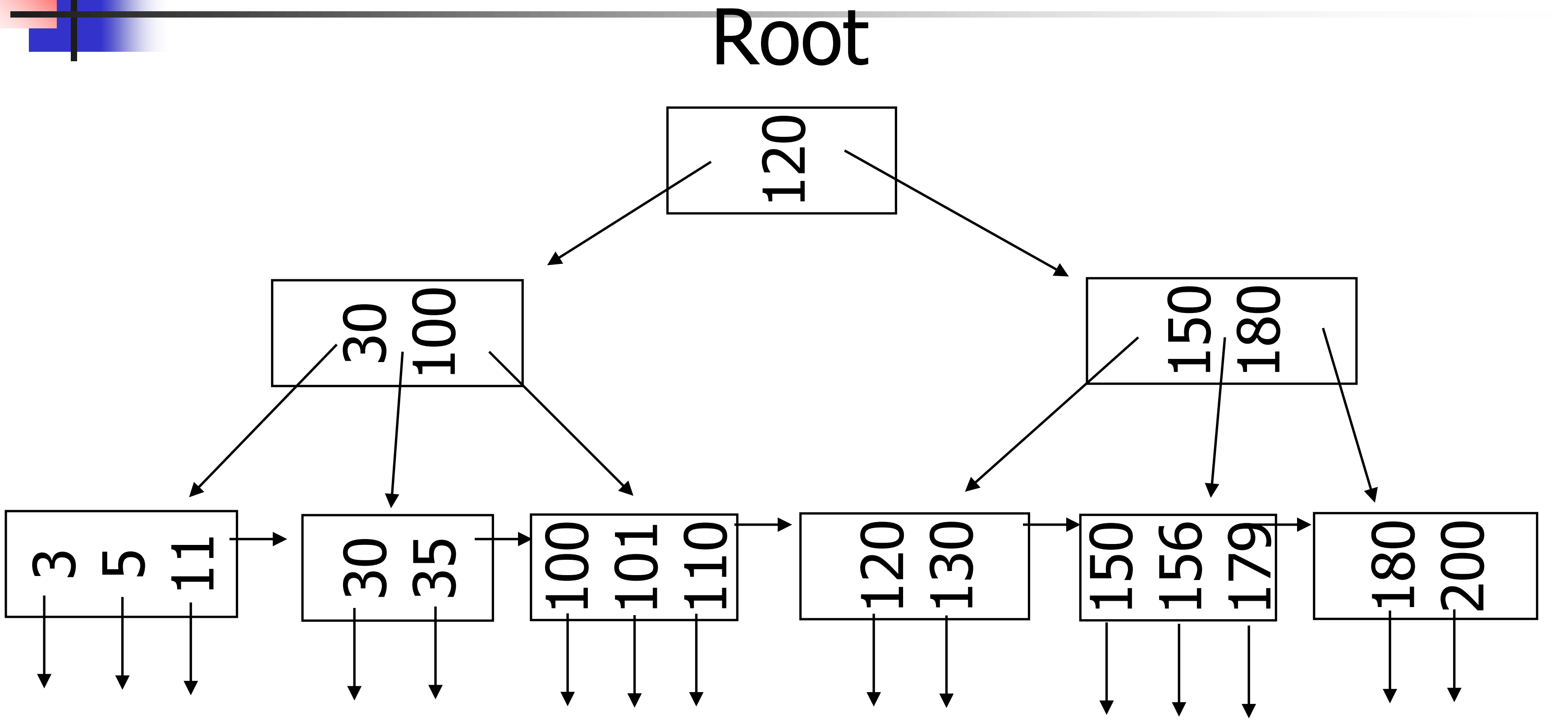
- Records must be <u>ordered</u> over an attribute, SSN, Name, etc.
- Queries: exact match and range queries over the indexed attribute: “find the name of the student with ID=087-34-7892” or “find all students with gpa between 3.00 and 3.5”

- Optimal 1-dimensional Index for range queries!

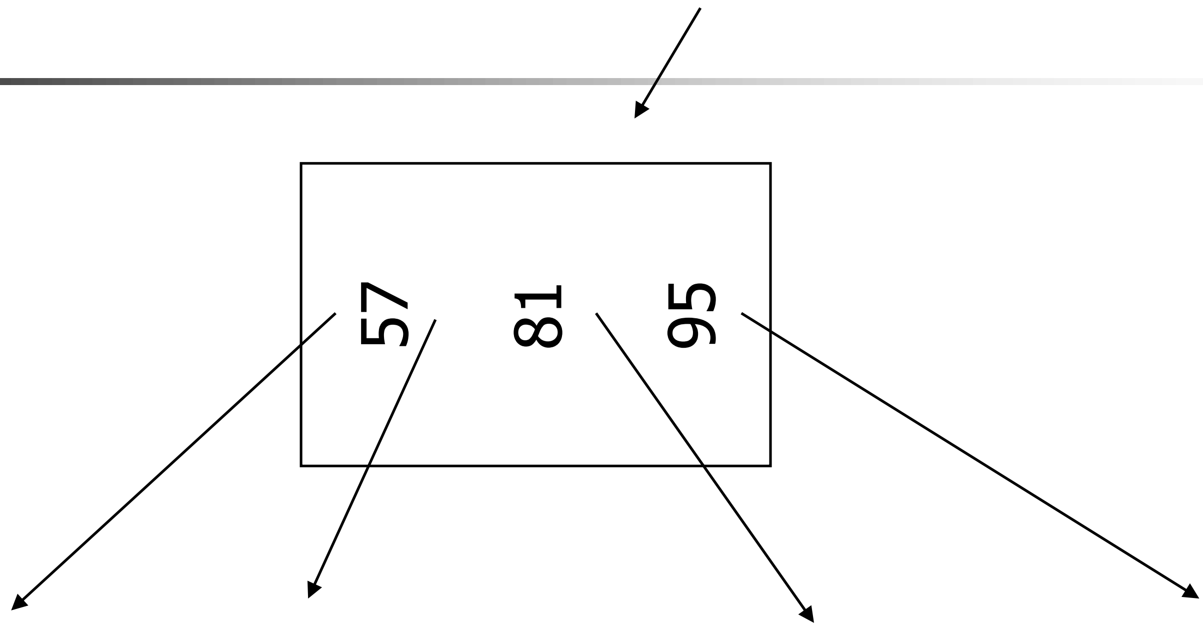
B+-tree:properties

- Insert/delete at log $_F$ (N/B) cost; keep tree *height-balanced*. (F = fanout)
- Minimum 50% occupancy (except for root). Each node contains **d** <= *m* <= 2**d** entries/pointers.
- Two types of nodes: **index** nodes and **data** nodes; each node is 1 page (disk based method)

Example


Root
120
30 100
150 180
3 5 11
30 35
100 101 110
120 130
150 156 179
180 200

Index node



to keys

< 57

to keys

57≤ k<81

to keys

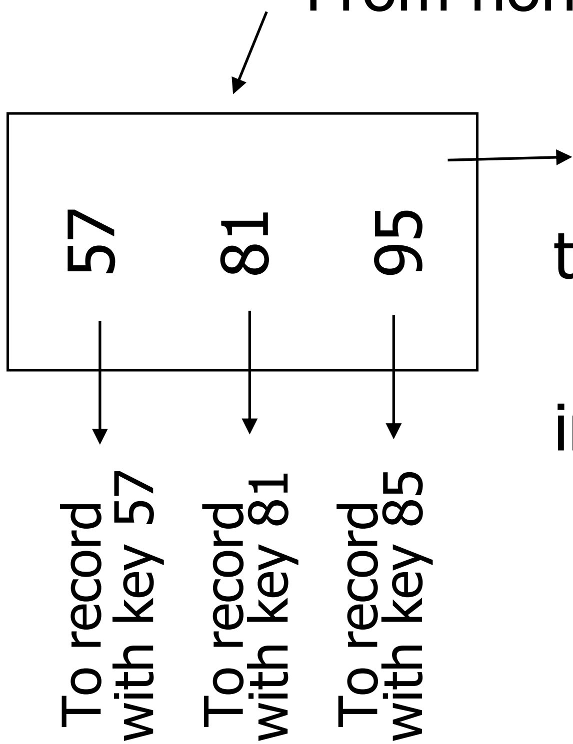
81≤k<95

to keys

95≤

Data node

From non-leaf node



to next leaf

in sequence

```
Struct {
Key   real;
Pointr *long;
} entry;

Node entry[B];
```

Insertion

- Find correct leaf *L.*
- Put data entry onto *L*.
  - If *L* has enough space, *done*!
  - Else, must *split* *L (into L and a new node L2)*
    - Redistribute entries evenly, **copy up** middle key.
    - Insert index entry pointing to *L2* into parent of *L*.
- This can happen recursively
  - To split index node, redistribute entries evenly, but **push up** middle key. (Contrast with leaf splits.)
- Splits "grow" tree; root split increases height.
  - Tree growth: gets *wider* or *one level taller at top.*

Deletion

- Start at root, find leaf *L* where entry belongs.
- Remove the entry.
  - If L is at least half-full, *done!*
  - If L has only **d-1** entries,
    - Try to **re-distribute**, borrowing from **sibling** *(adjacent node with same parent as L)*.
    - If re-distribution fails, **merge** *L* and sibling.
- If merge occurred, must delete entry (pointing to *L* or sibling) from parent of *L*.
- Merge could propagate to root, decreasing height.

Characteristics

- Optimal method for 1-d range queries:

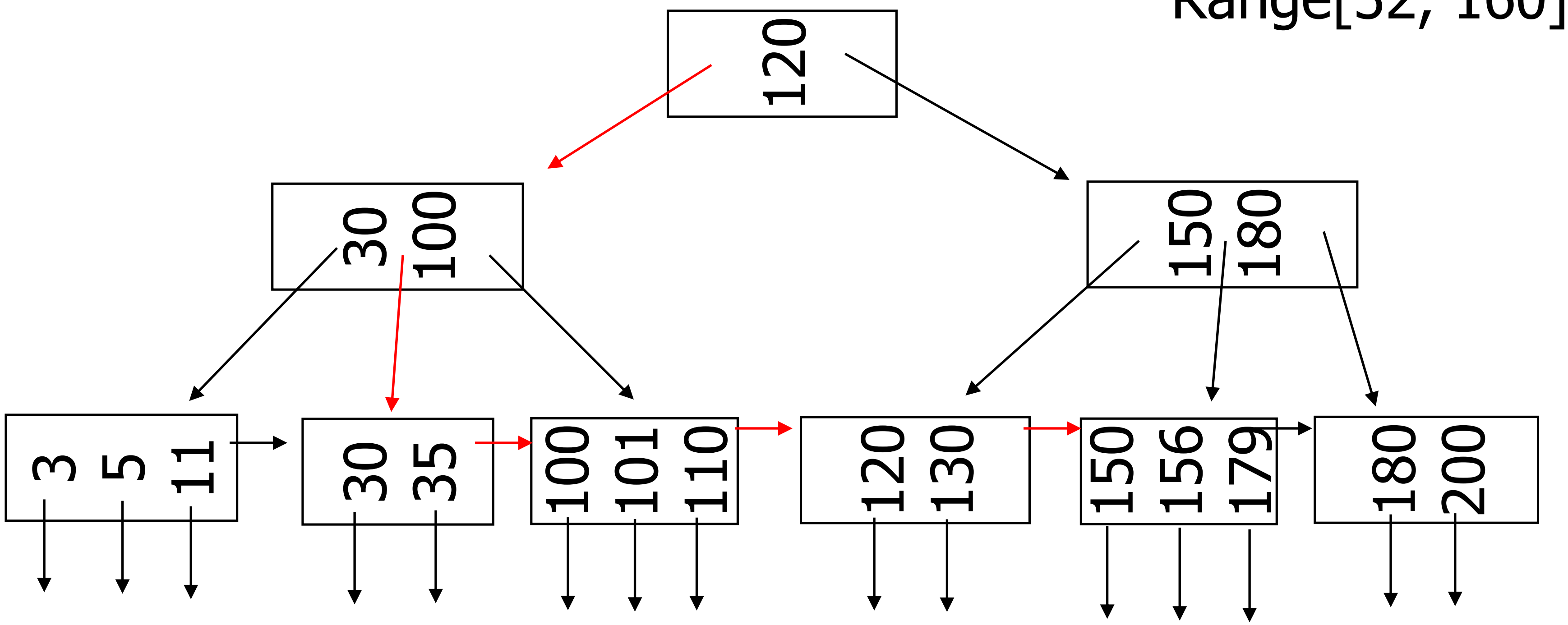
Space: $O(N/B)$, Updates: $O(\log_B(N/B))$, Query: $O(\log_B(N/B) + \alpha/B)$

- Space utilization: 67% for random input
- B-tree variation: index nodes store also pointers to records

Example

Root

Range[32, 160]



Other issues

- Internal node architecture [Lomet01]:
  - Reduce the overhead of tree traversal.
    - Prefix compression: In index nodes store only the prefix that differentiate consecutive sub-trees. Fanout is increased.
- Cache sensitive B+-tree
  - Place keys in a way that reduces the cache faults during the binary search in each node.
  - Eliminate pointers so a cache line contains more keys for comparison.

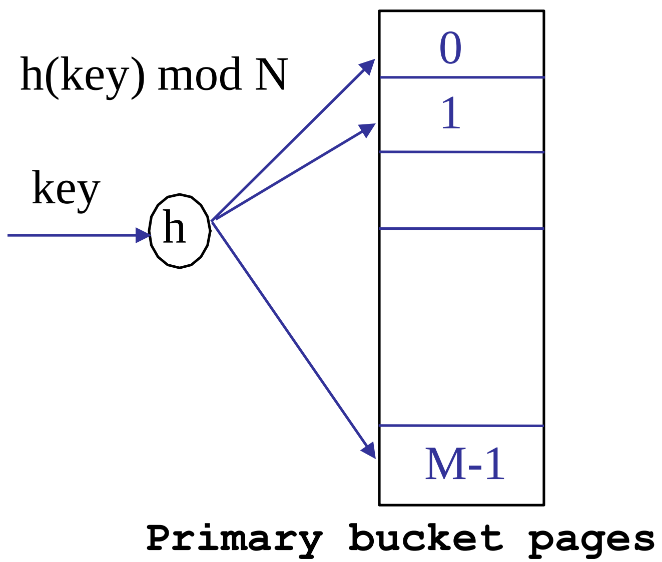
External Memory Hashing

Hashing

- Hash-based indices are best for exact match queries. Faster than B+-tree!
- Typically 1-2 I/Os per query where a B+-tree requires 4-5 I/Os
- But, cannot answer range queries…

Idea

- Use a function to direct a record to a page
- h(k) mod M = bucket to which data entry with key k belongs. (M = # of buckets)



Design decisions: Functions

*- h(x) = ((a*x+b) mod p)mod M*

$a>0$, $a,b < p$, p is a prime, and $p>M$

(this is a Universal Hash family for different (a,b) values)

*- h(x) = [fractional-part-of (x * φ)] * M,*

φ: golden ratio (0.618... = (sqrt(5)-1)/2)

- Size of hash table M
- Overflow handling: open addressing or chaining : problem in dynamic databases

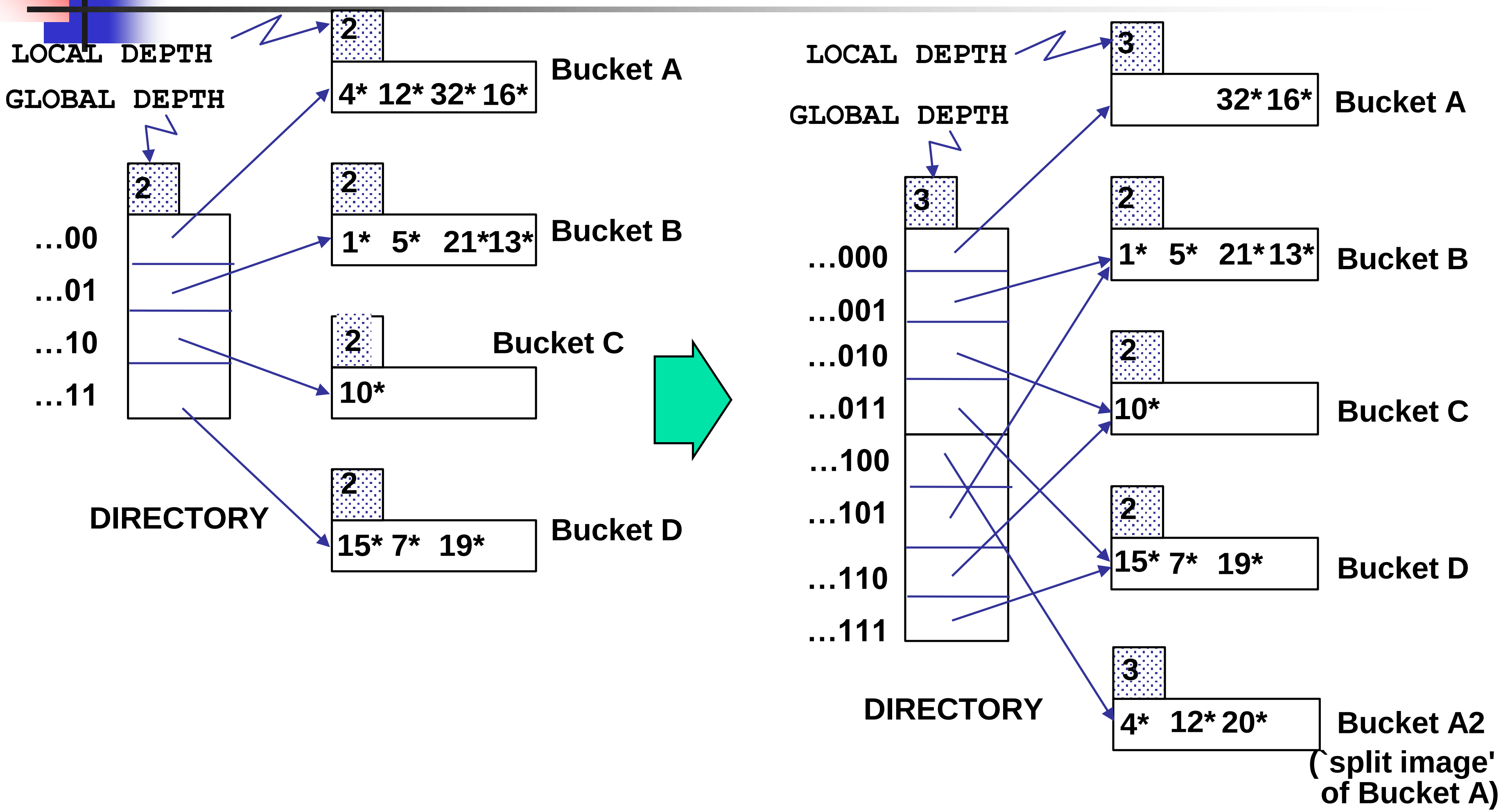
Dynamic hashing schemes

- Extensible hashing: uses a directory that grows or shrinks depending on the data distribution. No overflow buckets
- Linear hashing: No directory. Splits buckets in linear order, uses overflow buckets

Extensible Hashing

- Bucket (primary page) becomes full. Why not re-organize file by doubling # of buckets (changing the hash function)?
  - Reading and writing all pages is expensive!
  - Idea: Use directory of pointers to buckets, double # of buckets by doubling the directory, splitting just the bucket that overflowed!
  - Directory much smaller than file, so doubling it is much cheaper. Only one page of data entries is split.
  - Trick lies in how hash function is adjusted!

Insert h(k) = 20 10100→ 00



Linear Hashing

- This is another dynamic hashing scheme, alternative to Extensible Hashing.
- Motivation: Ext. Hashing uses a directory that grows by doubling… Can we do better? (smoother growth)
- LH: split buckets from left to right, regardless of which one overflowed (simple, but it works!!)

Linear Hashing (Contd.)

- Directory avoided in LH by using overflow pages. (chaining approach)
  - Splitting proceeds in `rounds'. Round ends when all M_R initial (for round R) buckets are split. Buckets 0 to Next-1 have been split; Next to M_R yet to be split.
  - Current round number is Level.
  - **Search:** To find bucket for data entry r, find $\mathbf{h}_{Level}(r)$:
    - If $\mathbf{h}_{Level}(r)$ in range `Next to M_R', r belongs here.
    - Else, r could belong to bucket $\mathbf{h}_{Level}(r)$ or bucket $\mathbf{h}_{Level}(r) + M_R$; must apply $\mathbf{h}_{Level+1}(r)$ to find out.

Linear Hashing: Example

Initially: $h(x) = x \bmod M$ (M=4 here)

Assume 3 records/bucket

Insert 17 = 17 mod 4 $\longrightarrow$ 1

Bucket id	0	1	2	3
	4 8	5 9 13	6	7 11

$hi(x) = x \bmod 2^{Level} * M$

Level=0

Linear Hashing: Example

Initially: $h(x) = x \bmod N$ (N=4 here)

Assume 3 records/bucket

Overflow for Bucket 1

Insert 17 = 17 mod 4 → 1

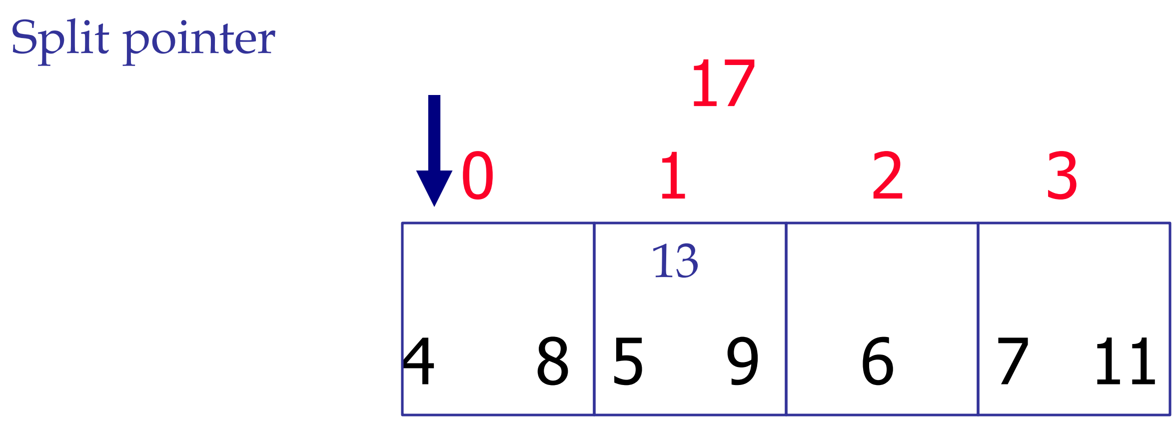
Bucket id	0	1	2	3
	4 8	5 9 13	6	7 11

Split bucket 0, anyway!!

Linear Hashing: Example

To split bucket 0, use another function h1(x):

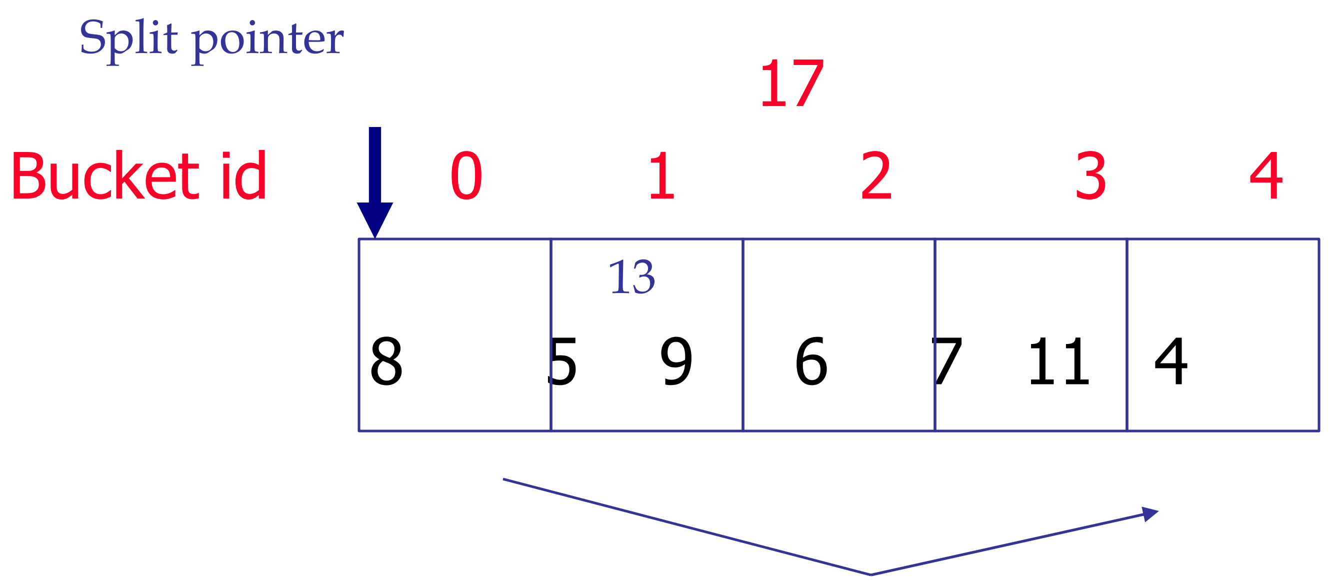
$h0(x) = x \bmod N$, $h1(x) = x \bmod (2*N)$



Linear Hashing: Example

To split bucket 0, use another function h1(x):

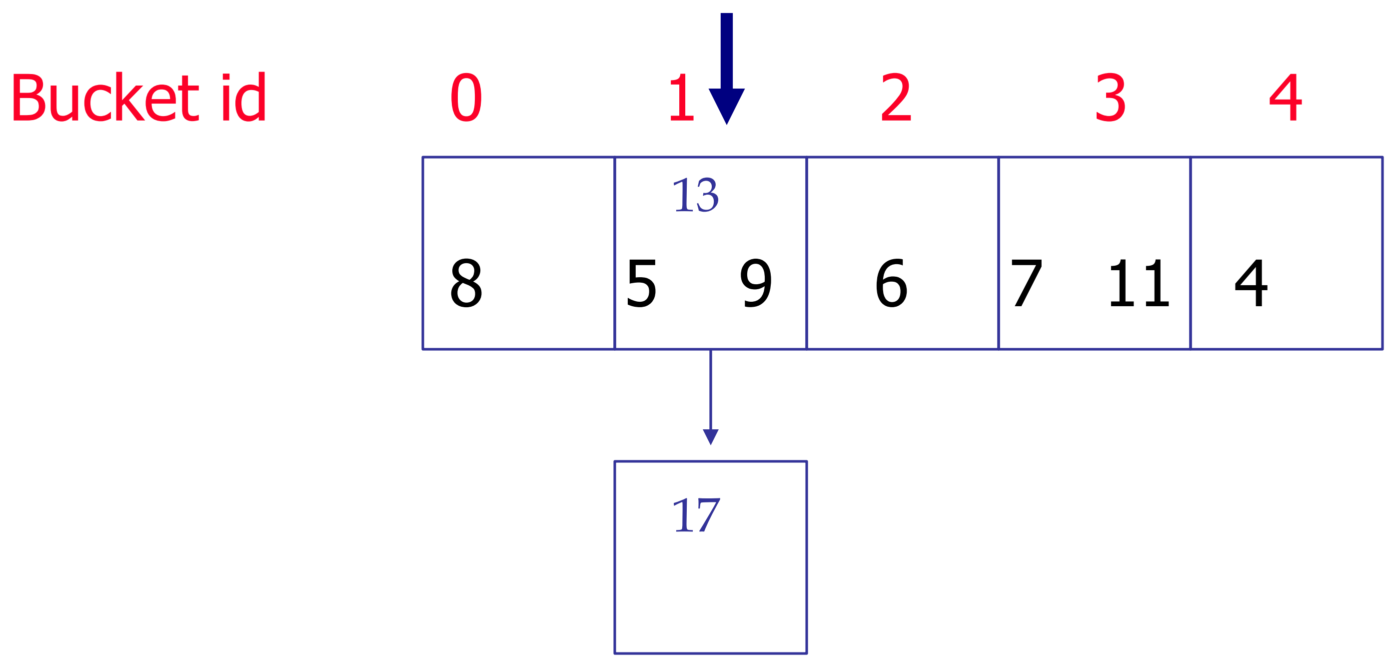
$h0(x) = x \bmod N$, $h1(x) = x \bmod (2*N)$



Linear Hashing: Example

To split bucket 0, use another function h1(x):

$$h0(x) = x \bmod N, \quad h1(x) = x \bmod (2*N)$$

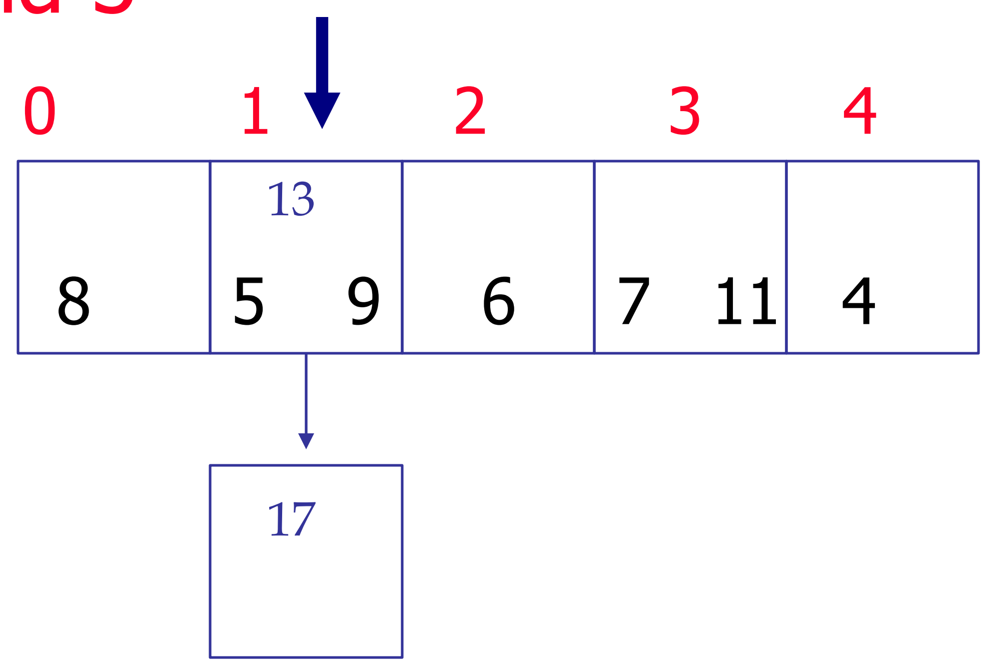


Linear Hashing: Example

h0(x) = x mod N , h1(x) = x mod (2*N)

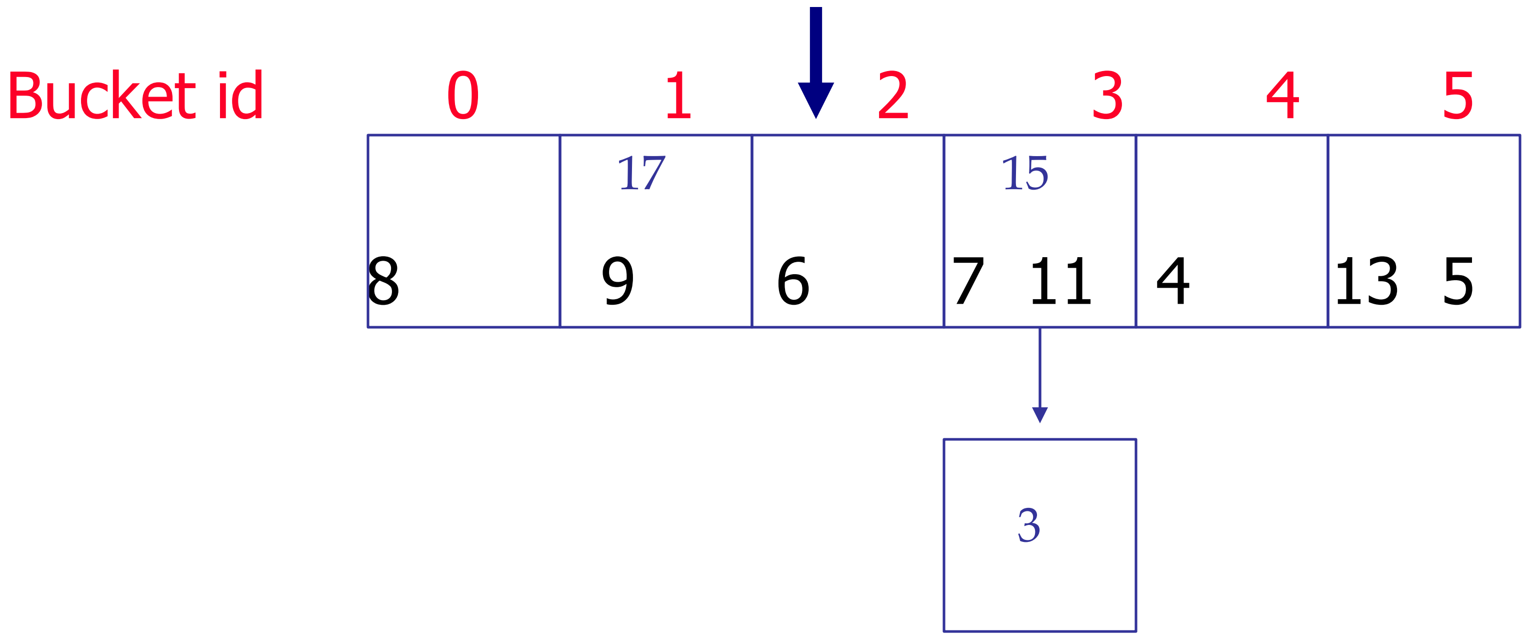
Insert 15 and 3

Bucket id



Linear Hashing: Example

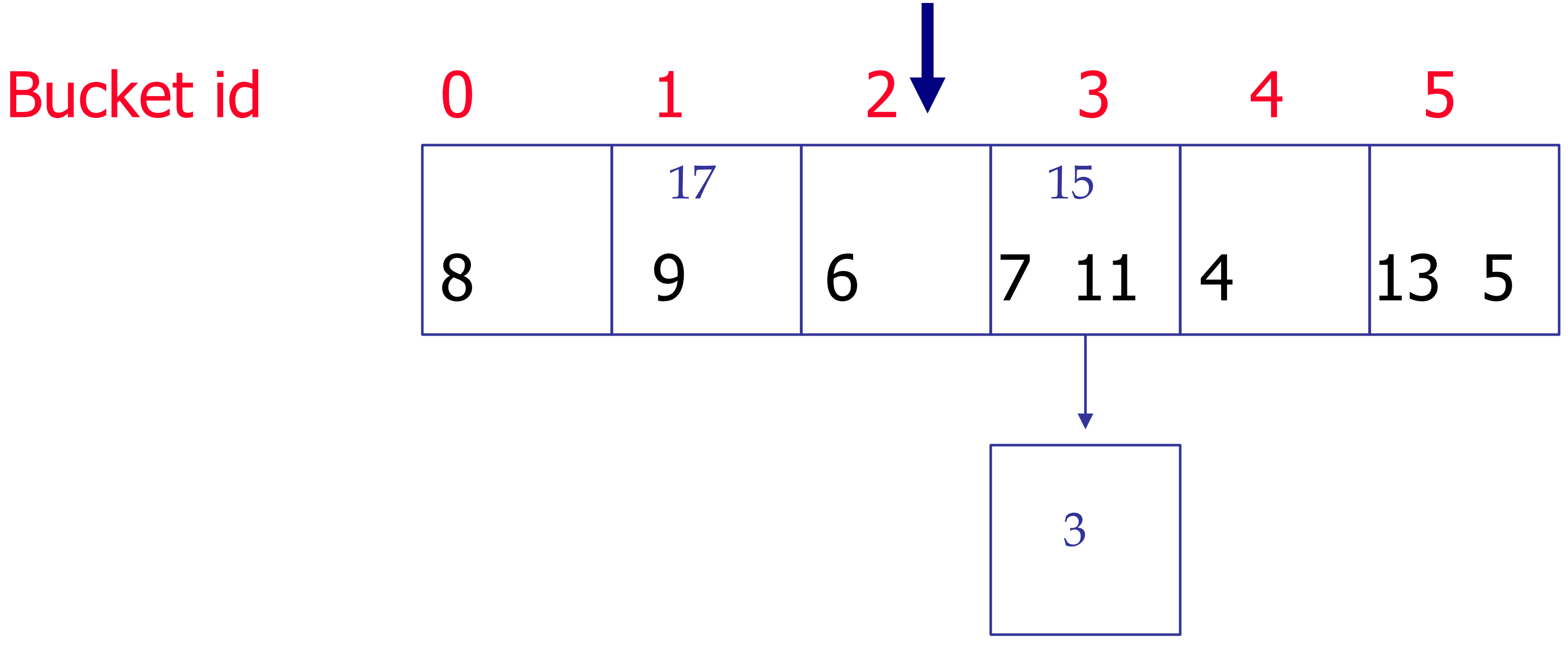
h0(x) = x mod N , h1(x) = x mod (2*N)



Linear Hashing: Search

h0(x) = x mod N (for the un-split buckets)

h1(x) = x mod (2*N) (for the split ones)



Linear Hashing: Search

h1(x) = x mod 8 (for the un-split buckets)

h2(x) = x mod 16 (for the split ones)

0	1	2	3	4	5	6	7
8	17 9	6	3 11	4	13 5		15 7

After we split the Nth bucket (3), we reset the Next pointer to 0 and we start a new round. The two hash functions are now h1 and h2.

Level =1

Linear Hashing: Search

Algorithm for Search:

```
Search(k)
    b = h0(k)
    if b < split-pointer then
        b = h1(k)
    read bucket b and search there
```

References

[Litwin80] Witold Litwin: Linear Hashing: A New Tool for File and Table Addressing. VLDB 1980: 212-223

http://www.cs.bu.edu/faculty/gkollios/ada01/Papers/linear-hashing.PDF