CS315: Principles of Database Systems
Indexing

Arnab Bhattacharya
arnabb@cse.iitk.ac.in

Computer Science and Engineering,
Indian Institute of Technology, Kanpur
http://web.cse.iitk.ac.in/~cs315/

2nd semester, 2013-14

Tue, Fri 1530-1700 at CS101

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
  1. Search key: Attribute that is used for searching
  2. Pointer to the entire object or tuple
- Index files should be smaller than data files

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
  1. Search key: Attribute that is used for searching
  2. Pointer to the entire object or tuple
- Index files should be smaller than data files
- Index evaluation metrics
  - Search time
  - Modification overhead
  - Space overhead

Basics

- Indexing is used to speed up search
- A search key is used
- An index file consists of records or index entries which has two fields
  1. Search key: Attribute that is used for searching
  2. Pointer to the entire object or tuple
- Index files should be smaller than data files
- Index evaluation metrics
  - Search time
  - Modification overhead
  - Space overhead
- Two basic types of indices
  1. Ordered index: search keys are organized according to some order
  2. Hash index: search keys are organized according to a hash function

Static hashing

- A hash function maps a key to a bucket
- A bucket is a unit of storage
- It is typically a disk block
- A key may need to be searched sequentially inside a bucket
- Results in hash file organization
- Example: mod n where n is the number of buckets

Hash function

- Two important qualities of an ideal hash function
- Uniform: Total number of keys from the domain is spread uniformly over all the buckets
- Random: Number of keys in each bucket is same irrespective of the actual distribution of keys

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing:

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing:

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing:

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing:

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing:

Bucket overflow

- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing: Interval is computed using another hash function h'
    - $h(k, i) = (h(k) + h'(k).i) \bmod m$

Bucket overflow

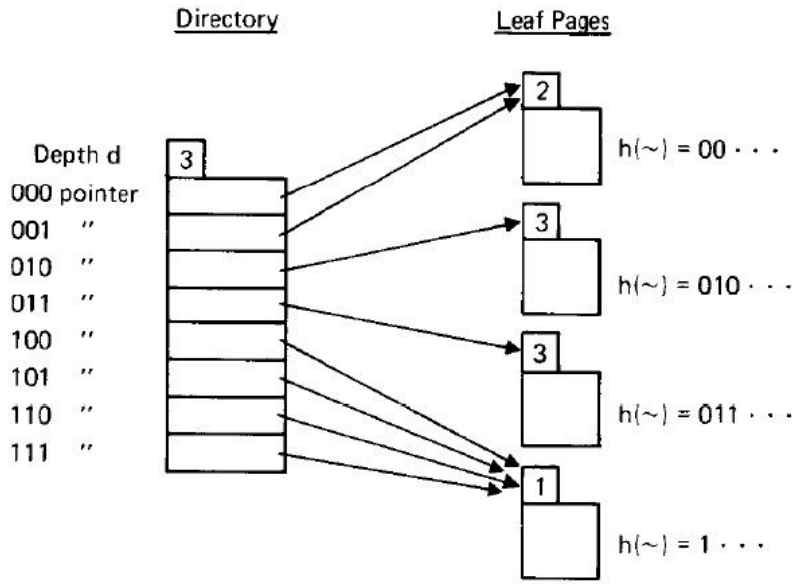
- Buckets may overflow at runtime due to
  - Skew in distribution of keys
  - Non-uniformity of hash function
- Probability can only be reduced, but not eliminated
- Handled using overflow buckets
- Closed addressing: Overflow buckets are linked together
  - Also called separate chaining or chaining
- Open addressing: Excess keys assigned to some other bucket and total number of buckets is kept fixed
  - Address at i^{th} attempt for key k is $h(k, i) = (h(k) + o_i.i) \bmod m$
  - Linear probing: Interval between buckets is fixed
    - $h(k, i) = (h(k) + c.i) \bmod m$
  - Quadratic probing: Interval between buckets increase linearly
    - $h(k, i) = (h(k) + c.i.i) \bmod m$
  - Double hashing: Interval is computed using another hash function h'
    - $h(k, i) = (h(k) + h'(k).i) \bmod m$
- Changing size of a database is a problem
- Periodic re-hashing is the only solution
- Dynamic hashing: h changes dynamically but deterministically

Dynamic hashing

- Organize overflow buckets as binary trees
- m binary trees for m primary pages
- $h_0(k)$ produces index of primary page
- Particular access structure for binary trees
- Family of functions $g(k) = \{h_1(k), \ldots, h_i(k), \ldots\}$
- Each $h_i(k)$ produces a bit
- At level i, if $h_i(k) = 0$, take left branch, otherwise right branch
- Example: bit representation

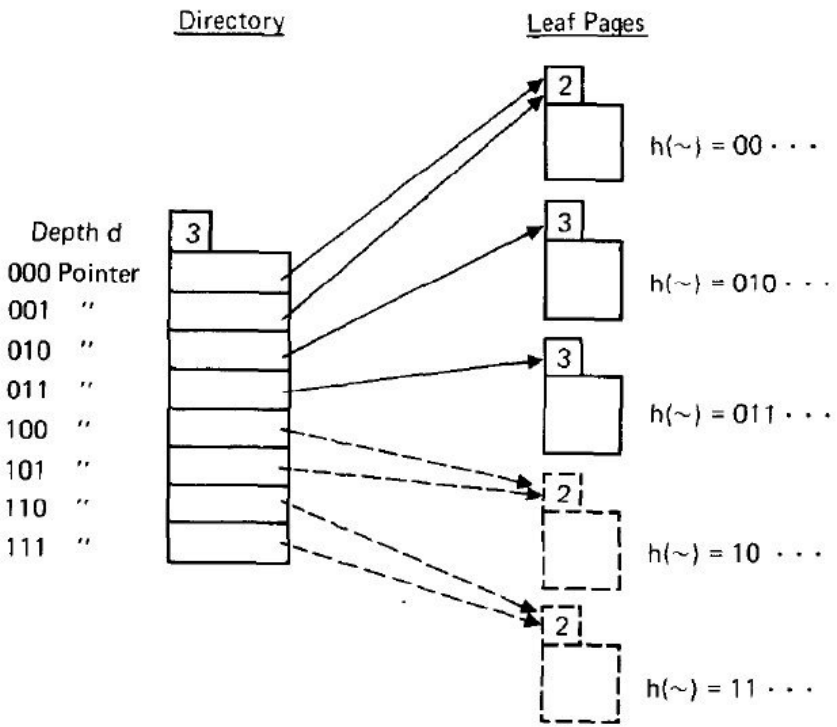
Extendible hashing

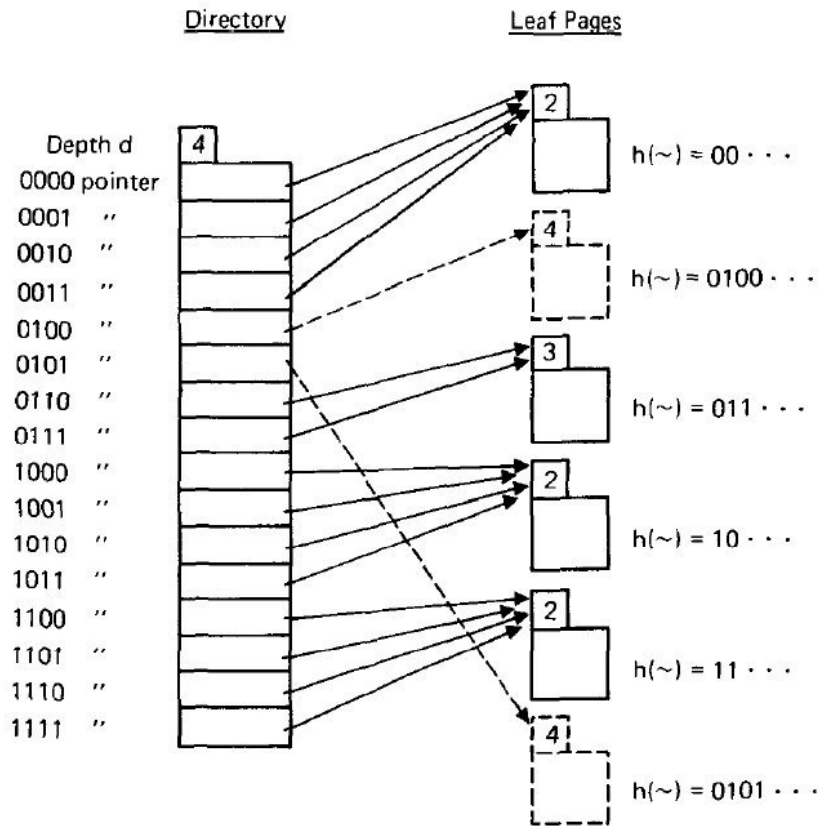
- Directory of pointers to buckets (leaf pages)
- Directory has *global depth d*
- 2^d pointers to leaf pages
- Pointer *i* contains keys starting with bit string *i*
- Leaf page has *local depth* $l \leq d$
- Leaf page *j* contains keys starting with bit string *j*



Insertion

- When leaf page overflows
  - If $l < d$, leaf page split into two and l is incremented for both new leaf pages
  - If $l = d$, directory doubles in size, d is incremented and leaf page splits





Linear hashing

- Number of buckets grow by at most 1
  - Linear growth
- Both primary and overflow buckets
  - Overflow buckets are chained
- Family g of hash functions $\{h_0, \ldots, h_i, \ldots\}$
  - $h_i(k) = h(k) \mod (2^i n)$
  - n is initial number of buckets
  - h_{i+1} doubles the range of h_i

Linear hashing

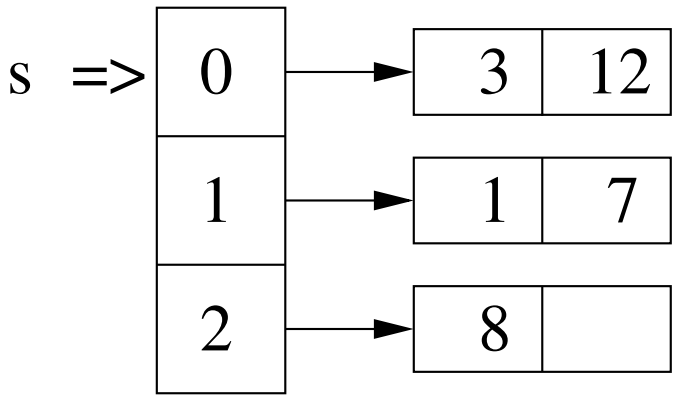
- Number of buckets grow by at most 1
  - Linear growth
- Both primary and overflow buckets
  - Overflow buckets are chained
- Family g of hash functions $\{h_0, \ldots, h_i, \ldots\}$
  - $h_i(k) = h(k) \mod (2^i n)$
  - n is initial number of buckets
  - h_{i+1} doubles the range of h_i
- Load factor decides when to split
- Split pointer s decides which bucket to split
  - s is independent of overflowing bucket
  - At level i, s is between 0 and $2^i - 1$
  - s is incremented and if at end, is reset to 0
- Records in splitting bucket are rehashed using h_{i+1}
  - Equal chance of being in old and new buckets

Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full

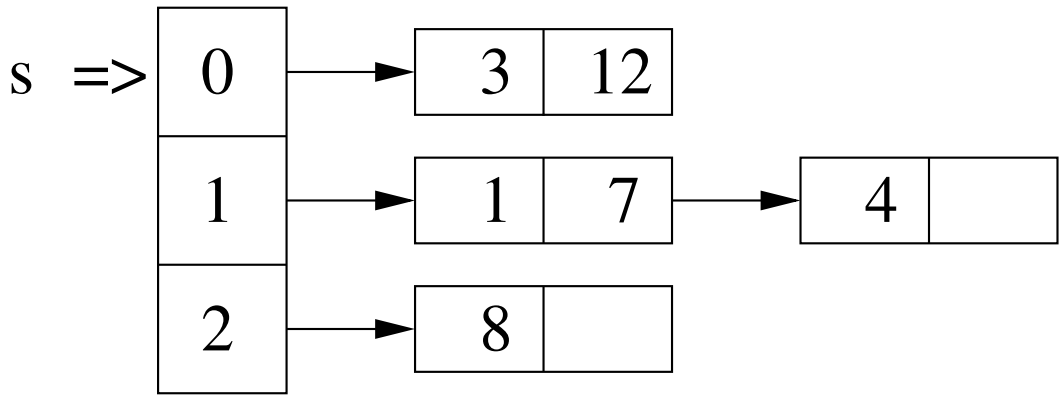
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



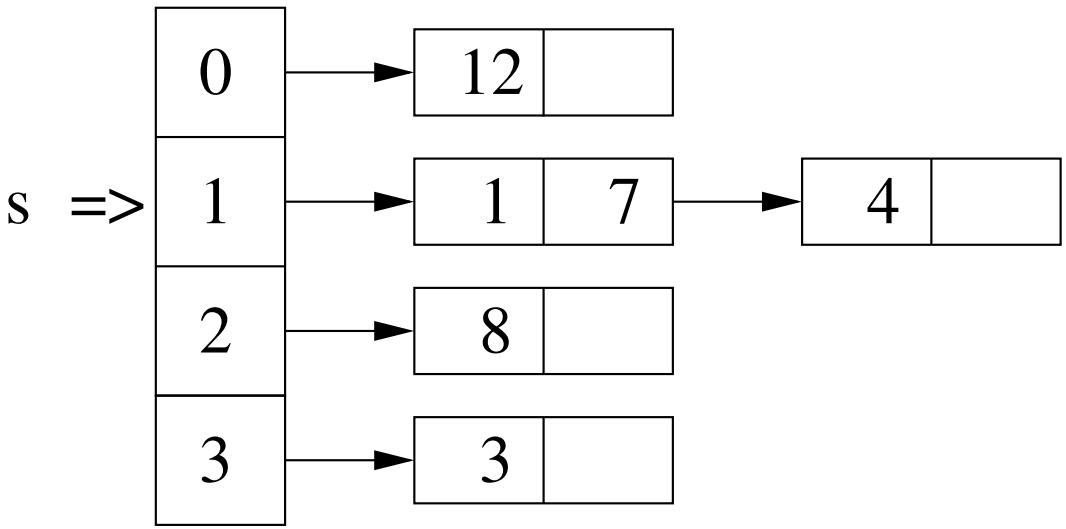
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



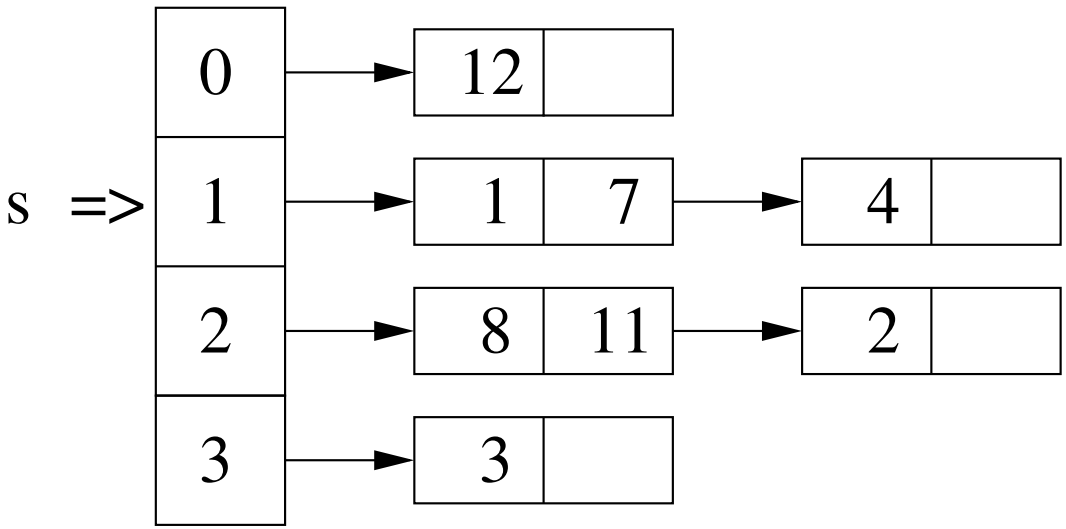
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



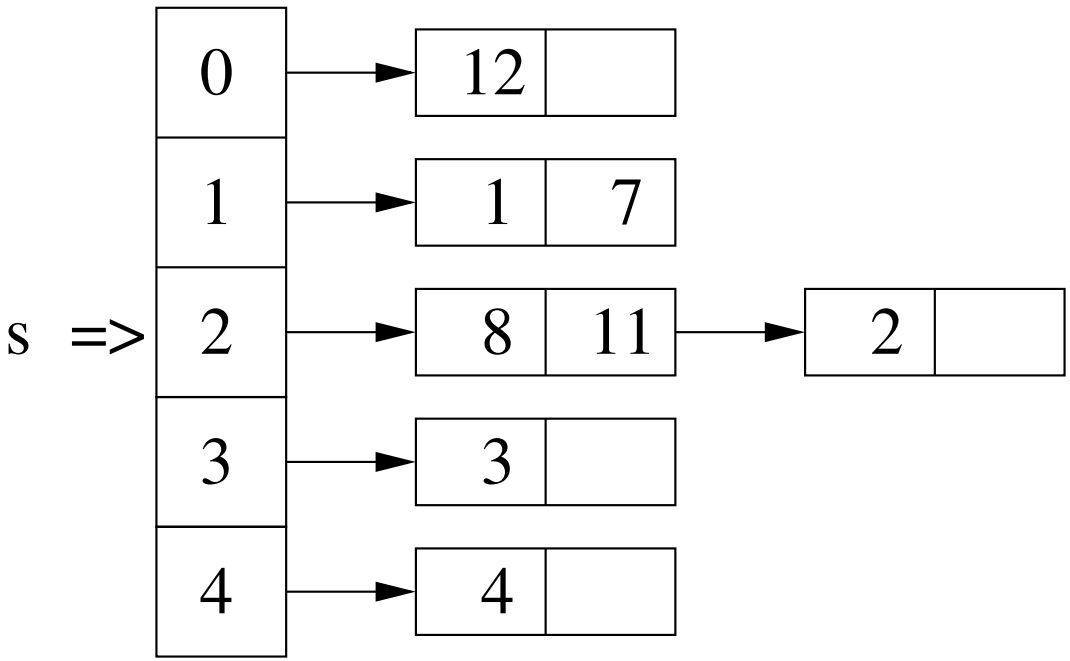
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



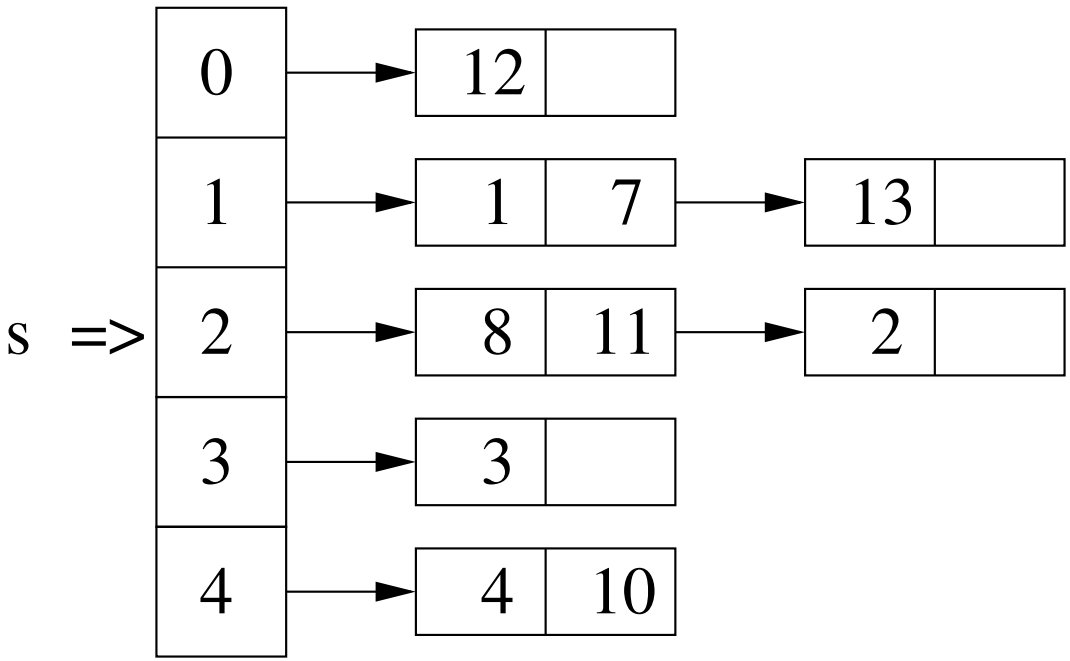
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



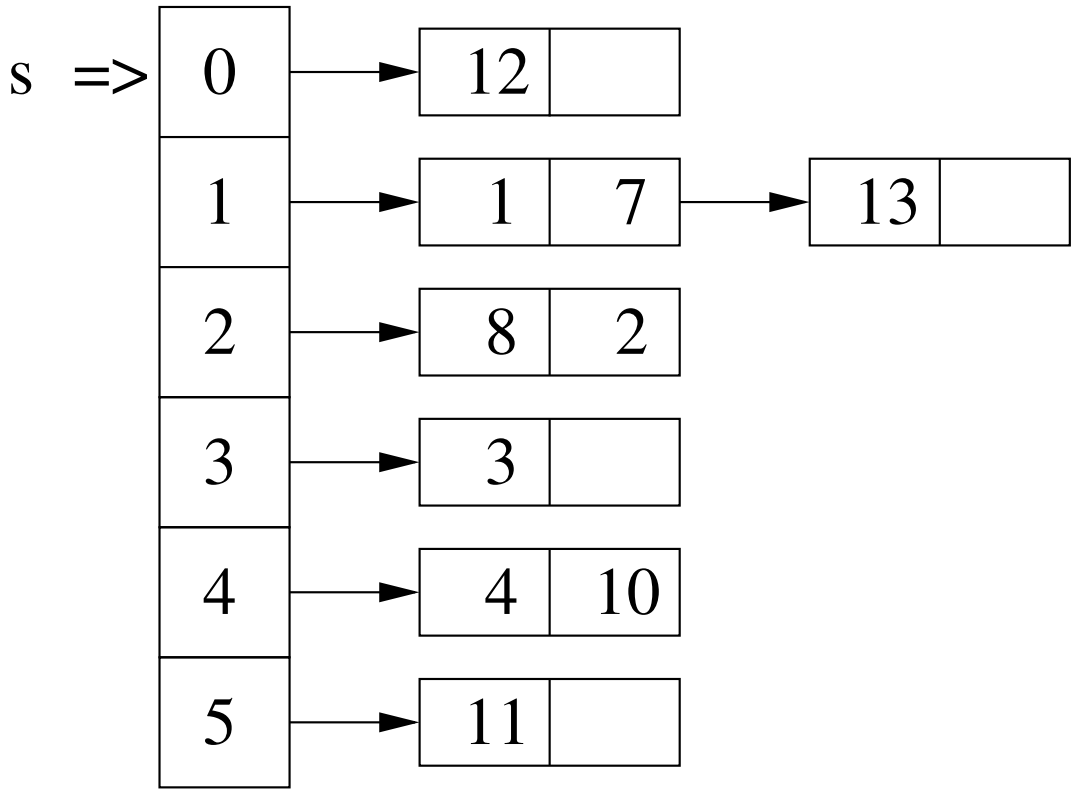
Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



Principle

- Full buckets are *not necessarily* split
- Buckets split are *not necessarily* full
- Principle: Every bucket will be split sooner or later and so all overflows will be reclaimed and rehashed



Searching

- Searching key r
- Find bucket $b = h_l(r)$, l being final level
- If $b > s$, then bucket b has not been split and r must be here
- Otherwise r may be in b or $b + 2^l$
- Apply $h_{l+1}(r)$ to find out

Ordered index

- Index sequential file: ordered sequential file with an index

Ordered index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index

Ordered index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there

Ordered index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there
  - Deletion: Replaced by next key
  - Insertion: Inserted only if entries exceed a block
- Sparse index requires less space and less maintenance but more searching time

Ordered index

- Index sequential file: ordered sequential file with an index
- Primary index or clustering index: index whose search key specifies the sequential order of the file
  - Generally primary key
- Secondary index or non-clustering index: any other index
- Dense index: index record appears for every key
  - Secondary index must be dense
- Sparse index: index record appears for only some keys
  - Records must be sequentially ordered by key
  - To locate k, find largest key $< k$, and then sequential search from there
  - Deletion: Replaced by next key
  - Insertion: Inserted only if entries exceed a block
- Sparse index requires less space and less maintenance but more searching time
- Multilevel index: primary index does not fit in memory
  - Outer index: Sparse primary index
  - Inner index: Dense primary index file

B-tree

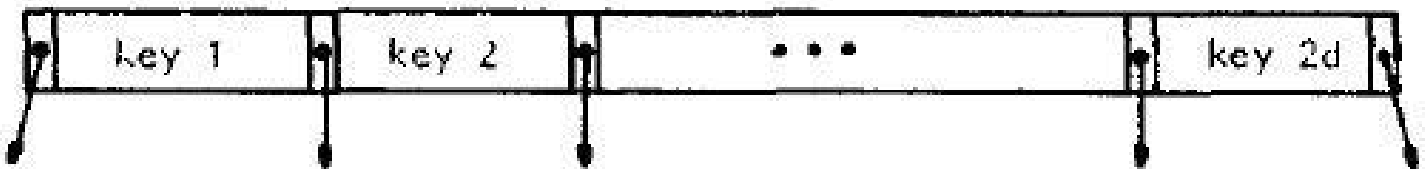
- A B-tree of order m has the following properties:
  1. Leaf nodes are in same level (balanced) and contain no data
  2. Internal nodes (not the root) have between m and $2m$ keys
  3. Root has at least 1 key
  4. An internal node with r keys have $r + 1$ children
  5. Child pointers in leaf nodes are null
- Branching factor = $m + 1$

B+-tree

- Balanced
- All data are in the leaf nodes
- Leaf nodes have $m \leq r \leq 2m$ keys and $r + 1$ data pointers
- Internal nodes contain $m \leq r \leq 2m$ keys and $r + 1$ child pointers
- Keys define range of values for children
- Often siblings are connected by pointers to avoid parent traversal

Order

- Branching factor or order determined by page size, size of key and size of pointer



- Page size = 4 kB
- Size of key = 8 bytes
- Size of pointer = 4 bytes
- If order is m, then $8 \times 2m + 4 \times (2m + 1) = 4 \times 1024$
- Therefore, $m = 170$
- A tree of height 3 can, therefore, store 5×10^6 records and that of height 4 can store 8×10^8 records

Index on multiple attributes

- Search keys having more than one attribute are called composite search keys
- Separate indices may be used
  - Union, intersection, etc. of individual results
- Multi-dimensional indexing
  - Quadtree: Extension of BST
  - R-tree: Extension of B+-tree
  - Indexing is specified by hyper-rectangles

Bitmap index

- Attribute domain consists of a small number of distinct values
- A bitmap or a bit vector is an array of bits
- Each distinct value has an array of the size of the number of tuples
  - If the *i*-th bit is 1, tuple *i* has that value

Gender	Grade
Male	C
Female	A
Female	C
Male	D
Male	A

- Two sets of bit vectors
  - Male = (10011), Female = (01100)
  - A = (01001), B = (00000), C = (10100), D = (00010)

Bitmap operations

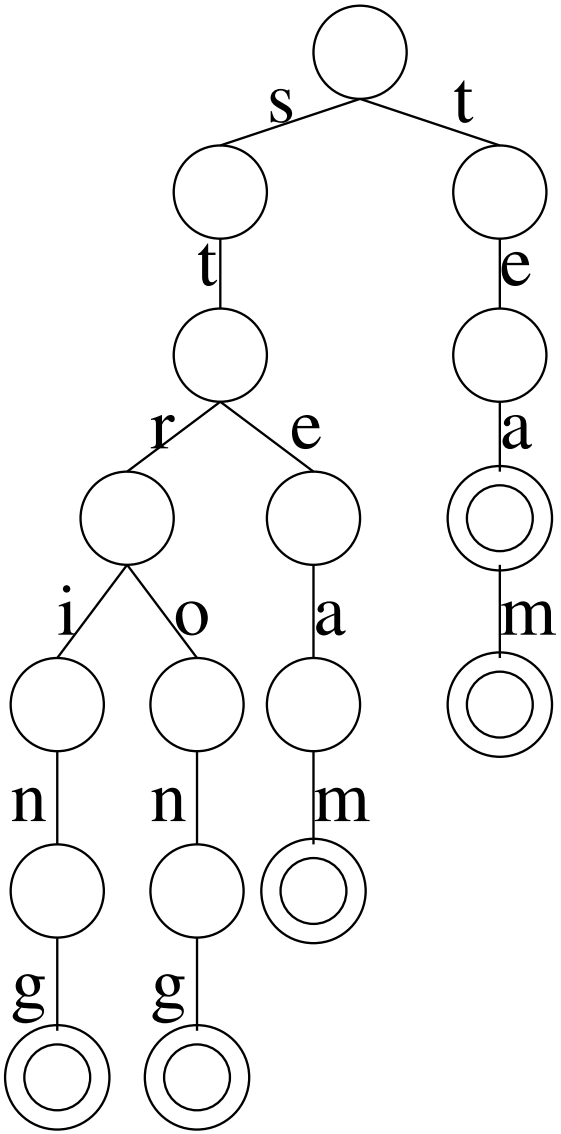
- Queries are answered using bitmap operations
- Example: Find the male student who got ‘D’
  - Bitmap(Male) AND Bitmap(D)
- Null values require a special bitmap for null
- O/S allows efficient bitmap operations when they are packed in word sizes

Tries

- Comes from the word retrieval
- Mostly used for strings
- Structure
  - Root represents null string
  - Each edge defines the next character
  - Each node stores a string or a prefix of a string
  - Strings with same prefix share the path
- Advantages over binary search trees
  - Search time is $O(m)$ where m is the length of the query
  - Size is generally less
  - Independent of database size
- Related structures: prefix tree, radix tree, suffix tree

Example

- string, strong, steam, team



Indices in SQL

- create index *i* on *r* (*a*) creates an index named *i* on the attribute *a* of the relation *r*

```
create index bindex on branch (bname)
```

- Index will be used implicitly whenever attribute *a* of relation *r* is queried
- drop index *i* deletes the index

```
drop index bindex
```