Hash Indexes

COSC 404 – Database System Implementation

Hash Indexes Overview

B-trees reduce the number of block accesses to 3 or 4 for large data sets. The goal of hash indexes is to make all operations require **only 1 access.**

Hashing is a technique for mapping key values to locations.

Hashing requires a hash function f(*x*), that takes the key value *x* and computes *y*=f(*x*) which is the location of where the key should be stored.

A ***collision*** occurs when try to store two different keys in the same location.

- $f(x_1) = y$ and $f(x_2) = y$ for two keys $x_1 != x_2$

Handling Collisions

A ***perfect hash function*** is a function that:

- For any two key values x_1 and x_2, $f(x_1)$!= $f(x_2)$ for all x_1 and x_2, where x_1 != x_2.
- That is, no two keys map to the same location.
- It is not always possible to find a perfect hash function for a set of keys depending on the situation.
  - Recent research on perfect hash functions is useful for databases.

We must determine how to handle collisions where two different keys map to the same location.

One simple way of handling collisions is to make the hash table really large to minimize the probability of collisions.

- This is not practical in general. However, we do want to make our hash table moderately larger than the # of keys.

Open Addressing

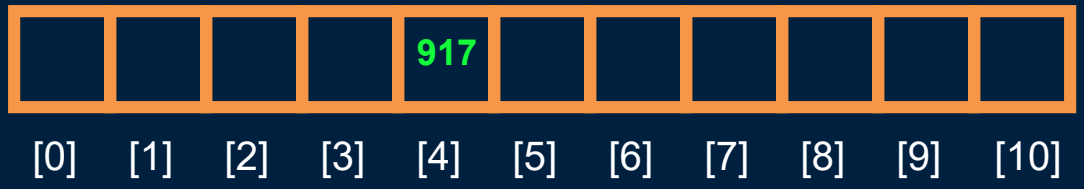
Open addressing with linear probing is a method for handling hash collisions.

Open addressing:

- Computes *y=f(x)* and attempts to put key in location *y*.
- If location *y* is occupied, scan the array to find the next open location. Treat the array as circular.

Open Addressing Example

Open addressing on a 11 element array with f(x) = x % 11:



Insert 917 at location 4.

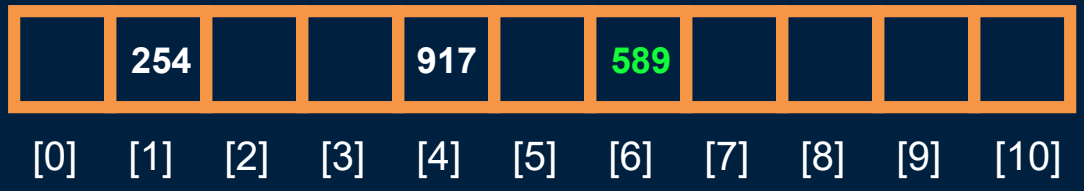
Open Addressing Example (2)

Open addressing on a 11 element array with f(x) = x % 11:

Insert 254 at location 1.

Open Addressing Example (3)

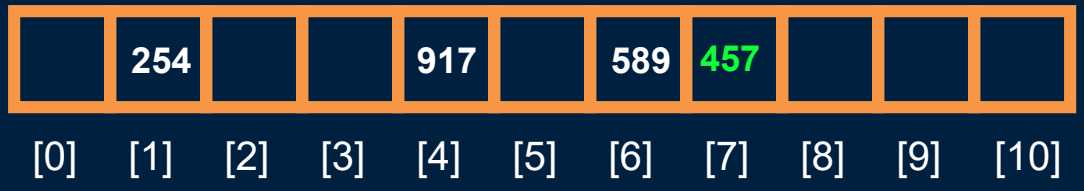
Open addressing on a 11 element array with f(x) = x % 11:



Insert 589 at location 6.

Open Addressing Example (4)

Open addressing on a 11 element array with f(x) = x % 11:

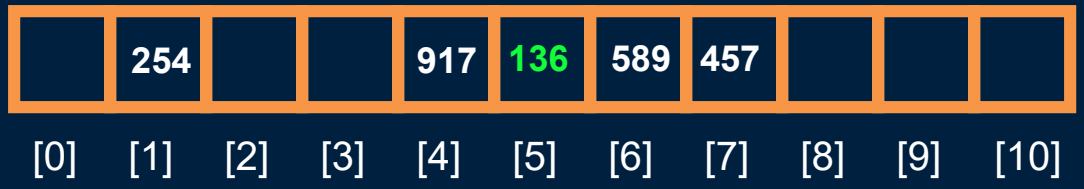


Insert 457 at location 6.

- Collision with 589.
- Next open location is 7, so insert there.

Open Addressing Example (5)

Open addressing on a 11 element array with f(x) = x % 11:

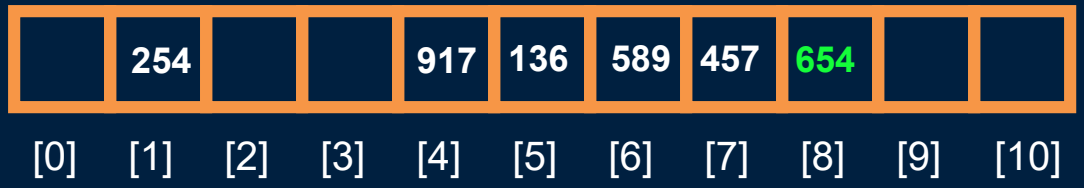


Insert 136 at location 4.

- Collision with 917.
- Next open location is 5, so insert there.

Open Addressing Example (6)

Open addressing on a 11 element array with f(x) = x % 11:

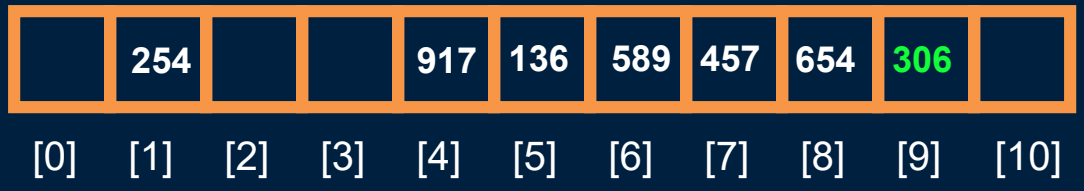


Insert 654 at location 5.

- Collision with 136.
- Note that a collision occurs with a key that did not even originally hash to location 5.
- Keep going down array until find location to insert which is 8.

Open Addressing Example (7)

Open addressing on a 11 element array with f(x) = x % 11:



Insert 306 at location 9.

Open Addressing Example Summary

Insert

- 917 1 probe(s)
- 589 1
- 254 1
- 457 2
- 136 2
- 654 4
- 306 1

Average number of probes = 12 / 7 = 1.7

Open Addressing
Insert and Delete

Insert using linear probing creates the potential that a key may be inserted many locations away from its original hash location.

What happens if an element is then deleted in between its proper insert location and the location where it was put?

- How does this affect insert and delete?

Example: Delete 589 (f(589)=6), then search for 654 (f(654)=5).

- Problem! Search would normally terminate at empty location 6!

Solution: Have special constants to mark when a location is empty and never used OR empty and was used.

Open Address Hashing

Question: What location is **19** inserted at?

A) 8

B) 9

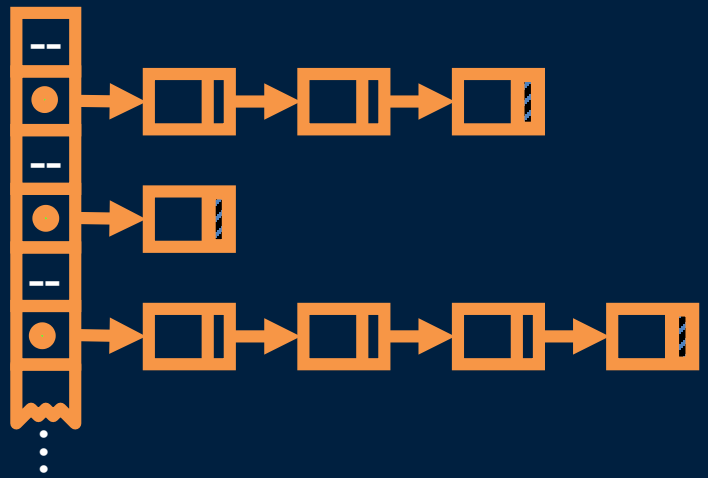
C) 6

D) 0

Separate Chaining

Separate chaining resolves collisions by having each array location point to a linked list of elements.

- Algorithms for operations such as insert, delete, and search are straightforward.
- As with open addressing, separate chaining may degenerate into a linear algorithm if the hash function does not distribute the keys evenly in the array.



Hash Limitations and Analysis

Hashing gives very good performance when the hash function is good and the number of conflicts is low.

- If the # of conflicts is high, then the performance of hashing rapidly degrades. The worse case is O(n).
- Collisions can be reduced by minimizing the:

  load factor = # of occupied locations/size of hash table.

However, on average, inserts, searches, and deletes are O(1)!

The limitations of hashing are:

- Ordered traversals are difficult without an additional structure and a sort. (hashing randomizes locations of records)
- Partial key searches are difficult as the hash function must use the entire key.

Hash Limitations and Analysis (2)

The ***hash field space*** is the set of all possible hash field values for records.

- i.e. It is the set of all possible key values that we may use.

The ***hash address space*** is the set of all record slots (or storage locations).

- i.e. Size of array in memory or physical locations in a file.

Tradeoff:

- The larger the address space relative to the hash field space, the easier it is to avoid collisions, BUT
- the larger the address space relative to the number of records stored, the worse the storage utilization.

Hashing Questions
How to handle real data?

Determine your own hash function for each of the following set of keys. Assume the hash table size is 100,000.

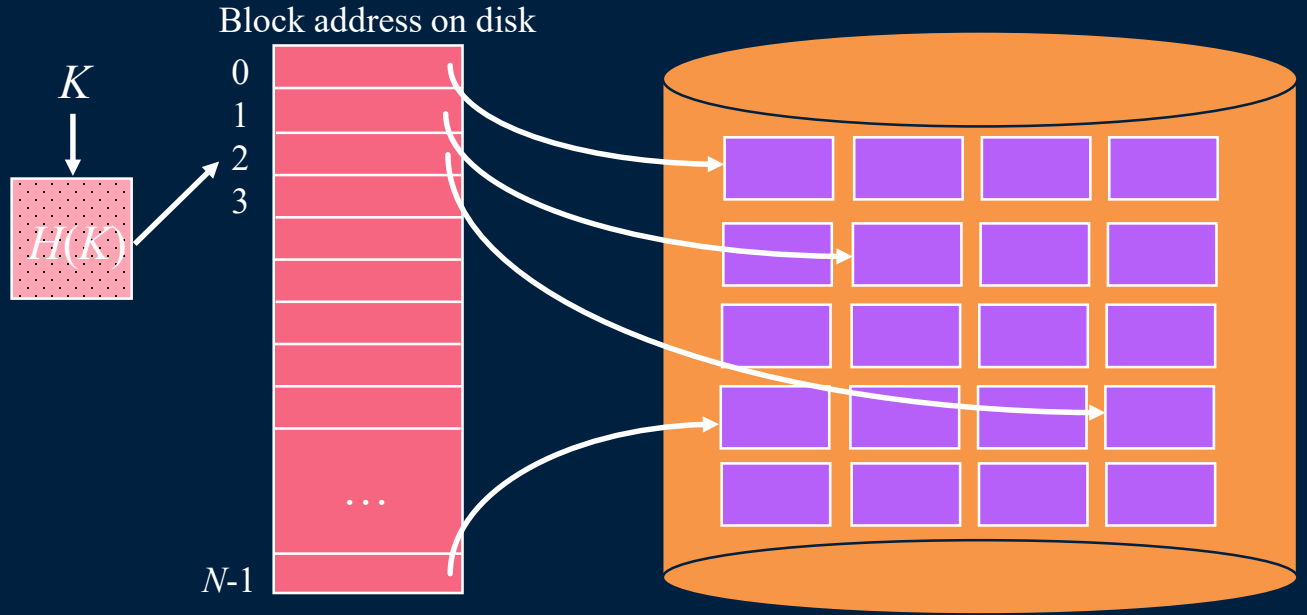
1) The keys are part numbers in the range 9,000,000 to 9,099,999.

2) The keys are people's names.

- E.g. "Joe Smith", "Tiffany Connor", etc.

External Hashing Overview

External hashing algorithms allocate records with keys to blocks on disk rather than locations in a memory array.

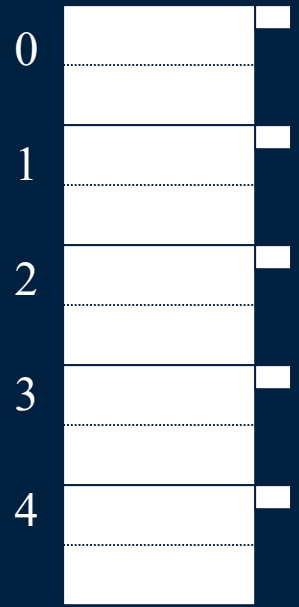


Hash file has relative bucket numbers 0 through N-1.
Map ***logical*** bucket numbers to ***physical*** disk block addresses.
Disk blocks are buckets that hold several data records each.

External Hashing Example

External Hash Table

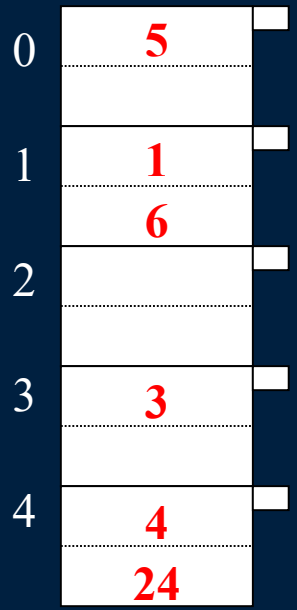
- 5 buckets
- 2 records per bucket
- use overflow blocks
- f(x) = x % 5



External Hashing Example Insertion

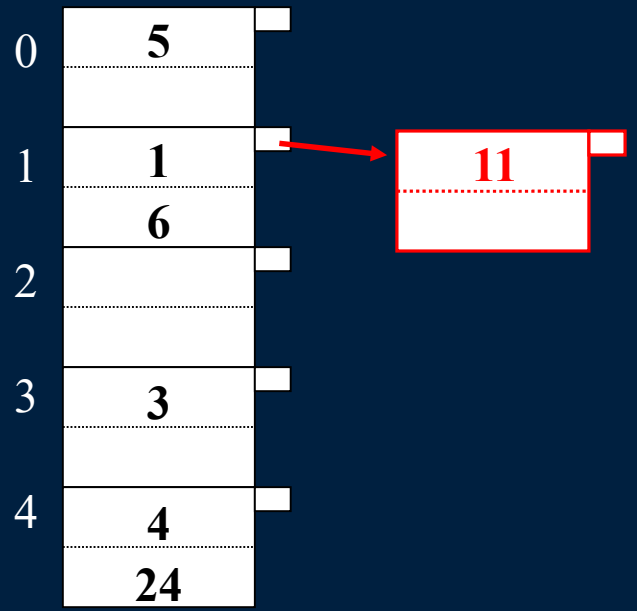
Insert:

1,5,3,6,4,24



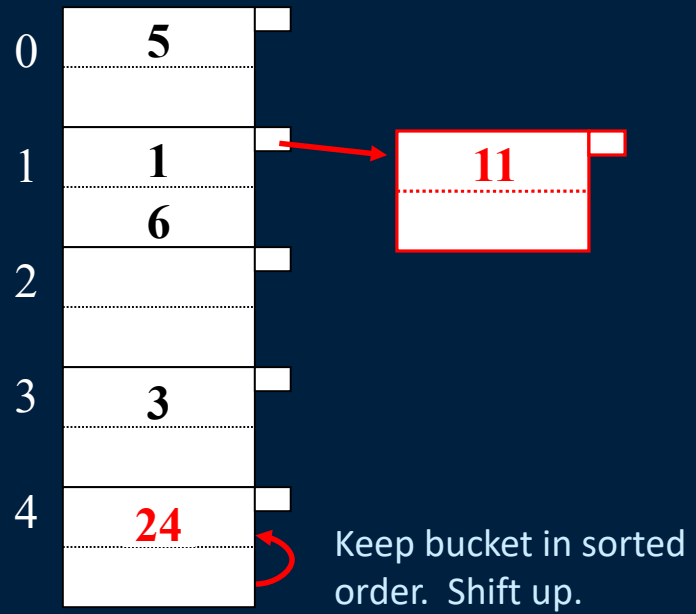
External Hashing Example
Insertion with Overflow

Insert: 11



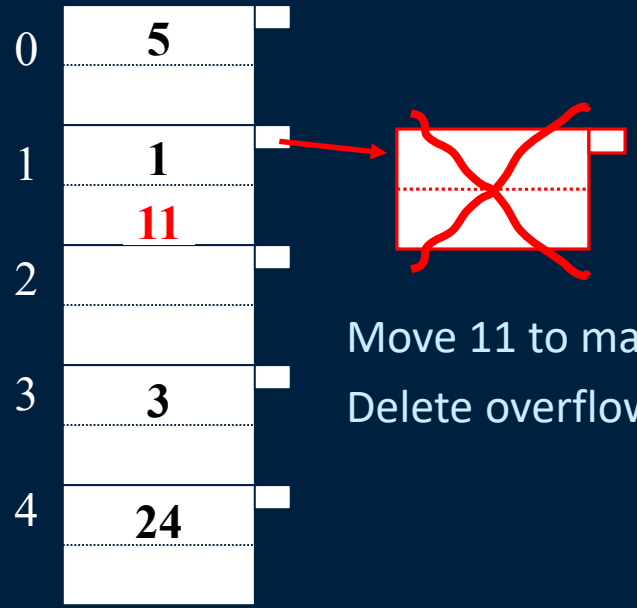
External Hashing Example
Deletion

Delete: 4



External Hashing Example
Deletion with Overflow

Delete: 6



Move 11 to main bucket.

Delete overflow block.

Deficiencies of Static Hashing

In static hashing, the hash function maps keys to a fixed set of bucket addresses. However, databases grow with time.

- If initial number of buckets is too small, performance will degrade due to too many overflows.
- If file size is made larger to accommodate future needs, a significant amount of space will be wasted initially.
- If database shrinks, again space will be wasted.
- One option is periodic re-organization of the file with a new hash function, but it is very expensive.
- ***Bottom line:*** Must determine optimal utilization of hash table.
  - Try to keep utilization between 50% and 80%. Hard when data changes.

These problems can be avoided by using techniques that allow the number of buckets to be modified dynamically.

Linear Hashing

Linear hashing allows a hash file to expand and shrink dynamically.

A linear hash table starts with 2^d buckets where ***d*** is the # of bits used from the hash value to determine bucket membership.

- Take the ***last d*** bits of ***H*** where ***d*** is the current # of bits used.

The growth of the hash table can either be triggered:

- 1) Every time there is a bucket overflow.
- 2) When the load factor of the hash table reaches a given point.

We will use the load factor method.

- Since bucket overflows may not always trigger hash table growth, overflow blocks are used.

Linear Hashing
Load Factor

The ***load factor lf*** of the hash table is the number of records stored divided by the number of possible storage locations.

- The initial number of blocks ***n*** is a power of 2.
  - As the table grows, it may not always be a power of 2.
- The number of storage locations ***s*** = #blocks X #records/block.
- The initial number of records in the table ***r*** is 0 and is increased as records are added.
- Load factor = ***r / s*** = ***r / n*** * #records/block

We will expand the hash table when the load factor > 85%.

Linear Hashing Load Factor

Question: A linear hash table has 5 blocks each with space for 4 records. There are currently 2 records in the hash table. What is its load factor?

A) 10%

B) 40%

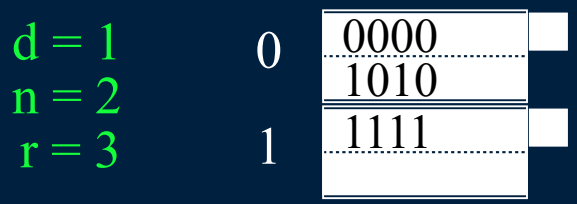
C) 50%

D) 0%

Linear Hashing Example

Assume each hashed key is a sequence of four binary digits.

Store values 0000, 1010, 1111.



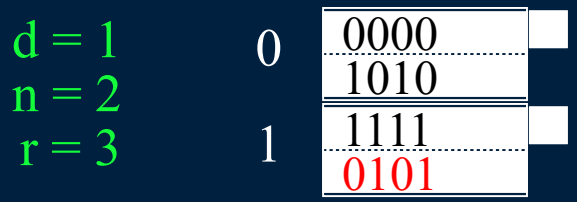
Linear Hashing
Insertions

Insertion algorithm:

- Insert a record with key ***K*** by first computing its hash value ***H***.
- Take the ***last*** ***d*** bits of ***H*** where ***d*** is the current # of bits used.
- Find the bucket ***m*** where ***K*** would belong using the ***d*** bits.
- If ***m < n***, then bucket exists. Go to that bucket.
  - If the bucket has space, then insert ***K***. Otherwise, use an overflow block.
- If ***m >= n***, then put ***K*** in bucket ***m - 2^{d-1}***.
- After each insert, check to see if the load factor ***lf*** < threshold.
- If ***lf >=*** threshold perform a split:
  - Add new bucket ***n***. (Adding bucket ***n*** may increase the directory size ***d***.)
  - Divide the records between the new bucket n=***$1b_2 \ldots b_d$*** and bucket ***$0b_2 .. b_d$***.
  - **Note that the bucket split may not be the bucket where the record was added!** Update ***n*** and ***d*** to reflect the new bucket.

Linear Hashing Insertion Example

Insert 0101.



4/4 = 100% full.
Above threshold triggers split.

Linear Hashing Insertion Example (2)

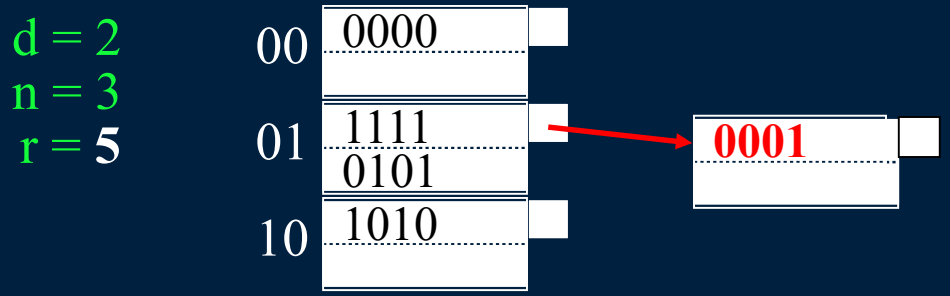
Added new bucket 10. (2 in binary - old n!)

Divide records of bucket 00 and 10.

d = **2**
n = **3**
r = **4**

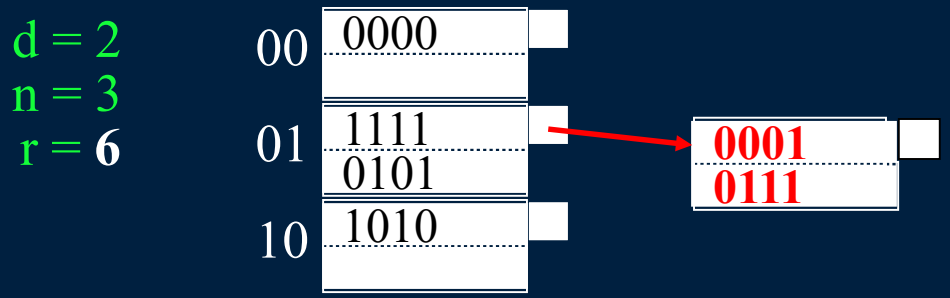
Linear Hashing Insertion Example (3)

Insert 0001. Use overflow block.



Linear Hashing Insertion Example (4)

Insert 0111.



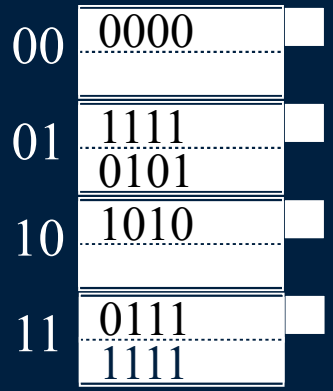
6/6 = 100% full.

Above threshold triggers split.

Linear Hashing Insertion Example (5)

Create bucket 11. Split records between 01 and 11.

d = 2
n = 4
r = 6



Linear Hashing Question

1) Show the resulting hash directory when hashing the keys: 0, 15, 8, 4, 7, 12, 10, 11 using linear hashing.

- Assume a bucket can hold two records (keys).
- Assume 4 bits of hash key.
- Add a new bucket when utilization is >= 85%.

Clicker: What bucket is 11 in?

A) 000

B) 001

C) 1011

D) 011

B+-trees versus Linear Hashing

B+-trees versus linear hashing: which one is better?

Factors:

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?

Expected type of queries:

- Hashing is generally better at retrieving records having a specified value for the key.
- If range queries are common, B+-trees are preferred.

Real-world result: PostgreSQL implements both B+-trees and linear hashing. Currently, linear hashing is not recommended for use.

Hash Indexes Summary

Hashing is a technique for mapping key values to locations.

- With a good hash function and collision resolution, insert, delete and search operations are O(1).
- Ordered scans and partial key searches however are inefficient.
- Collision resolution mechanisms include:
  - open addressing with linear probing - linear scan for open location.
  - separate chaining - create linked list to hold values and handle collisions at an array location.

Dynamic hashing is required for databases to handle updates.

Linear hashing performs dynamic hashing and grows the hash table one bucket at a time.

Major Objectives

The "One Things":

- Perform open address hashing with linear probing.
- Perform linear hashing.

Major Theme:

- Hash indexes improve average access time but are not suitable for ordered or range searches.

Other objectives:

- Define: hashing, collision, perfect hash function
- Calculate load factor of a hash table.
- Compare/contrast external hashing and main memory hashing.
- Compare/contrast B+-trees and linear hashing.


UBC
THE UNIVERSITY OF BRITISH COLUMBIA