Linear Hashing & Spiral Storage

Gkanios Antonios

Per-Ake Larson, Dynamic Hash Tables

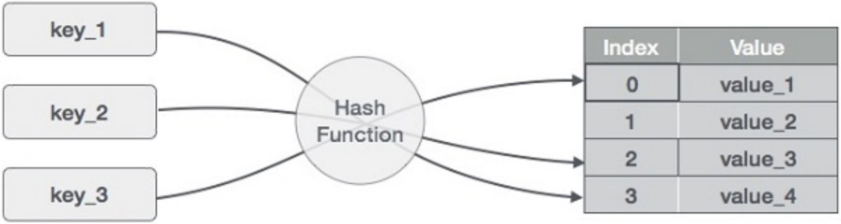
12/11/2020

Contents

Hashing in General

- A hash table is an in-memory data structure that associates keys with values.
- The primary operation it supports efficiently is a lookup: given a key, find the corresponding value.
- It works by transforming the key using a hash function into a hash, a number that is used as an index in an array to locate the desired location where the values should be.
- Multiple keys may be hashed to the same bucket.
- All hash table implementations have some collision resolution strategy.
- Hash tables are often used to implement associative arrays, sets and caches.

Hashing in General



Index	Value
0	value_1
1	value_2
2	value_3
3	value_4

Static Hashing

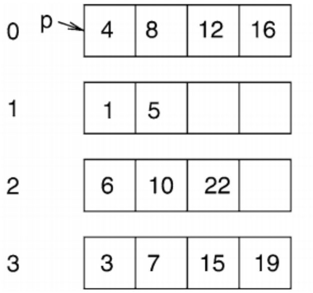
- In the static hashing, the hash function maps search-key values to a fixed set of buckets.This has some disadvantages:
    - Hash tables grow with time. Once buckets start to overflow, performance will degrade.
    - If we attempt to anticipate some future size and allocate sufficient buckets for that expected size when we build the hash table initially, we will waste lots of space.
    - If the hash table ever shrinks, space will be wasted.
    - We can try some workarounds, like reorganizing records, but is very expensive
- To avoid these problems, we would like to use some techniques that allow us to modify dynamically the number of buckets in our hash table.
- **Linear Hashing** is the first in a number of schemes known as **dynamic hashing**.

Linear Hashing in General

- Linear Hashing scheme was invented by Witold Litwin in 1980.
- Linear Hashing is a dynamic data structure which implements a hash table and grows or shrinks one bucket at a time.
- The name Linear Hashing is used because the number of buckets grows or shrinks in a linear fashion.
- When an overflow occurs, it is not always the overflown bucket that is split.
- Overflows are handled by creating a chain of pages under the overflown bucket.
- The hashing function changes dynamically and at any given instant there are two hashing functions used by the scheme.

Initial Layout

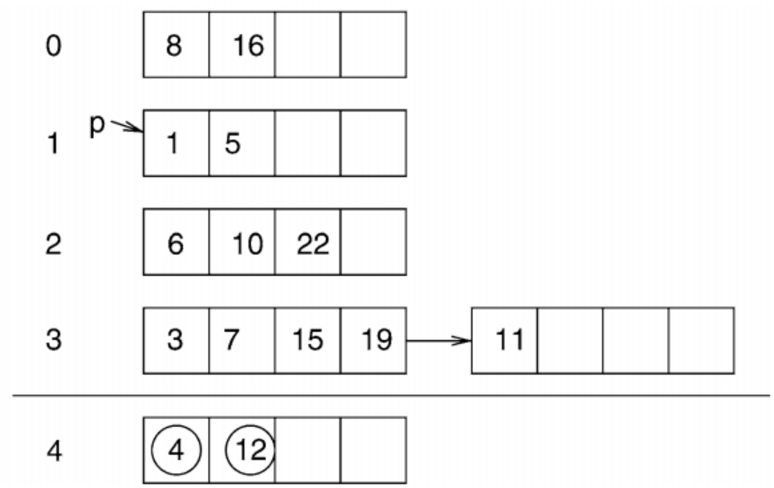
- The Linear Hashing scheme has N initial buckets labelled 0 through $N - 1$.
- Initial hashing function $h_0(k) = f(k)\%N$ (used to map any key k into one of the N buckets).
- Index p which points to the bucket to be split next whenever an overflow page is generated (p = 0).



Bucket Split

- When the first overflow occurs (it can occur in any bucket), bucket 0, which is pointed by p, is split (rehashed) into two buckets (0 and N).
- A new empty page is also added in the overflown bucket to accommodate the overflow.
- Splitting a bucket involves moving approximately half of the records from the bucket to a new bucket at the end of the table using a new hash function h_1.
- A crucial property of h_1 is that search values that were originally mapped by h_0 to some bucket j must be remapped either to bucket j or bucket $j + N$.

Bucket Split (2)



$N = 4,\ h_0(k) = k \ \% \ 2^2,\ h_1(k) = k \ \% \ 2^3.$

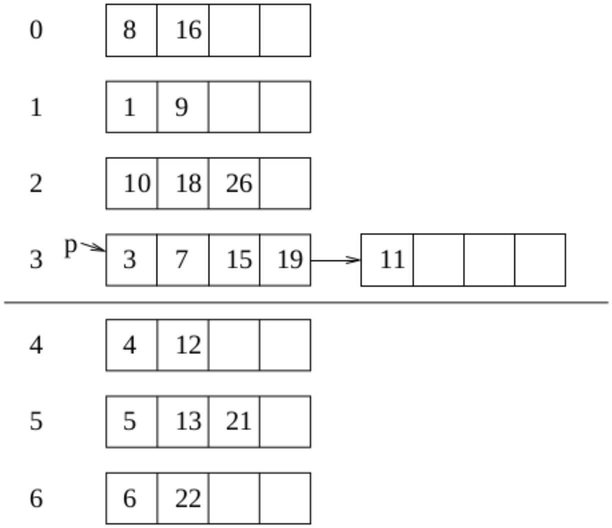
Round and Hash Function Advancement

- After enough overflows, all original N buckets will be split. This marks the end of splitting-round 0.
- At the end of round 0 the Linear Hashing scheme has a total of 2N buckets.
- Hashing function h_0 is no longer needed as all $2N$ buckets can be addressed by hashing function h_1.Index p is reset to 0 and a new round, namely splitting-round 1, starts. A new hash function h_2 will start to be used.

Round and Hash Function Advancement (2)

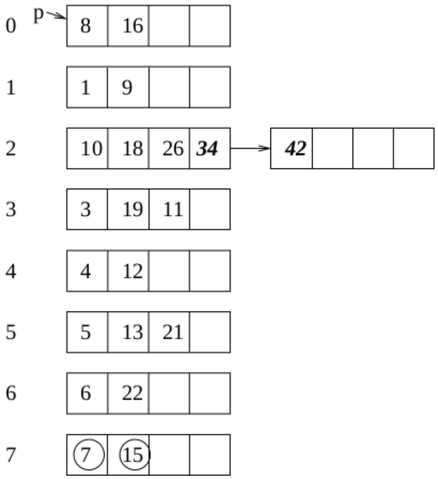
- Linear Hashing scheme involves a family of hash functions $h_0, h_1, h_2, \ldots$.
- $h_i(k) = f(k) \% 2^i * N$. This way, it is guaranteed that if h_i hashes a key to bucket $j \in [0 \ldots 2^i * N - 1]$, h_{i+1} will hash the same key to either bucket j or bucket $j + 2^i * N$.
- In splitting round i, the hash functions h_i and h_{i+1} are used.
- At the beginning of round i, $p = 0$ and there are $2^i * N$ buckets.
- When all of these buckets are split, splitting round $i + 1$ starts and hash functions h_{i+1} and h_{i+2} will start to be used.

Round and Hash Function Advancement (3)



$$N = 4,\ h_0(k) = k \ \%\ 2^2,\ h_1(k) = k \ \%\ 2^3.$$

Round and Hash Function Advancement (4)



$N = 4,\ h_1(k) = k \ \%\ 2^3,\ h_2(k) = k \ \%\ 2^4.$

Search Scheme

- A search scheme is needed to map a key k to a bucket, either when searching for an existing record or when inserting a new record. The search scheme works as follows:
  - If $h_i(k) \geq p$, then choose bucket denoted by $h_i(k)$ since the bucket has not been split yet in the current round.
  - If $h_i(k) < p$, then choose bucket denoted by $h_{i+1}(k)$ which can be either bucket denoted by $h_i(k)$ or its spit image bucket $h_i(k) + 2^i * m$.

Linear Hashing Variations

- A split performed whenever a bucket overflow occurs is an uncontrolled split.
- The overall load factor is defined as the number of records in the table divided by the (current) number of buckets. In our case, the overall load factor equals the average chain length.
- We fix a lower and an upper bound on the overall load factor and expand (contract) the table whenever the overall load factor goes above (below) the upper (lower).
- In practice, higher storage utilization is achieved if a split is triggered not by an overflow, but when the load factor becomes greater than some upper threshold.
- This is called a controlled split.

Performance Analysis

- We analyze the expected performance of a growing linear hash table under the assumption that the table is expanded as soon as the overall load factor exceeds $\alpha, \alpha > 0$.
- It is also assumed that there are no deletions.
- The expected cost of retrieval and insertion depends on what fraction of the buckets has already been split during the current round.
- The performance is best at the end of an expansion because the load is uniform over the whole table.
- The performance varies cyclically where a cycle corresponds to a doubling of the table.

Performance Analysis (2)

- A linear hash table can be viewed as consisting of two traditional hash tables:
    - The buckets that have not yet been split during the current round.
    - The buckets that have been split plus the new buckets created during the current round.
- Within each part, the expected load is the same for every bucket.

Performance Analysis (3)

- Let's consider a traditional hash table with load factor λ, where records are stored using chaining.
- Expected number of key comparisons for a for a successful search: $s(\lambda) = 1 + \frac{\lambda}{2}$.
- Expected number of key comparisons for a for an unsuccessful search: $u(\lambda) = \lambda$.
- Let $0 \leq x \leq 1$, denote the fraction of buckets that have been split during the current round.
- Let z, denote the expected number of records in an unsplit bucket.

Performance Analysis (4)

- For the overall load factor to be equal to α, the following must hold:
  $2xz/2 + (1-x)z = a(2x+1-x) \implies z = a(1+x)$.
  In other words, the expected number of records in an unsplit bucket grows linearly from α to 2α.
- Let $S(\alpha, x)$, denote the expected number of key comparisons for a successful search when a fraction x of the buckets have been split.
- Let $U(\alpha, x)$, denote the expected number of key comparisons for an unsuccessful search when a fraction x of the buckets have been split.
- $S(\alpha, x) = xs(\frac{\alpha(1+x)}{2}) + (1-x)s(\alpha(1+x)) = 1 + \frac{\alpha}{4}(-x^2+x+2)$
- $U(\alpha, x) = xu(\frac{\alpha(1+x)}{2}) + (1-x)u(\alpha(1+x)) = \frac{\alpha}{2}(-x^2+x+2)$

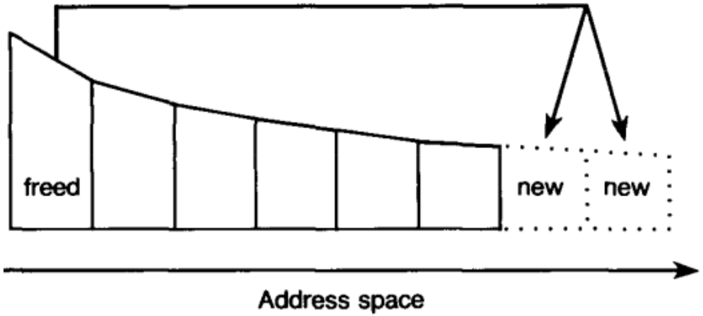
Performance Analysis (5)

- The minimum expected search lengths occur when the load is uniform over the whole table, that is, when $x = 0$ or $x = 1$.Then:
  - $S(\alpha, 0) = 1 + \frac{\alpha}{2}$.
  - $U(\alpha, 0) = \alpha$.
- The average search cost over a cycle can be computed by integrating the expected search length over a split round. Then:
  - $\bar{S}(\alpha, x) = \int_0^1 S(\alpha, x) dx = 1 + \frac{\alpha}{2}\frac{13}{12}$.
  -
  - $\bar{U}(\alpha, x) = \int_0^1 U(\alpha, x) dx = \alpha\frac{13}{12}$

Problem with Linear Hashing and Workaround

- When using linear hashing the expected cost of retrieving, inserting or deleting a record varies cyclically.
- Spiral storage overcomes this undesirable feature and exhibits uniform performance regardless of the table size
- The load is high at the beginning of the (active) address space and tapers off towards the end.
- To expand the table additional space is allocated at the end of the address space, and at the same time a smaller amount of space is freed at the beginning.

Spiral Storage Address Space



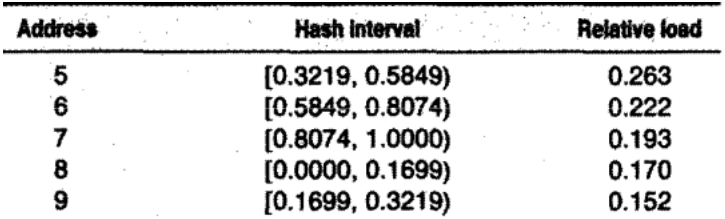
Spiral Storage Scheme

- Spiral storage requires a hashing function that maps keys uniformly into $[0, 1)$. $0 \leq h(k) < 1$.
- h(K) is then mapped into a value x in $[S, S+1)$. x is uniquely determined by requiring that its fractional part must agree with h(K): $(x = \lceil S - h(k) \rceil + h(k))$
- The final address is computed as $y = \lfloor d^x \rfloor$.
- Currently active address space extends from $\lfloor d^S \rfloor$ to $\lceil d^{S+1} \rceil - 1 \approx d^S * (d-1)$ addresses.
- $d \geq 1$ is a constant, called expansion factor
- d^x is called expansion function (most convenient value for computations, is $d = 2$).
- There are many possible expansion functions.
- The expansion function 2^x has the property that the expansion rate is constant.

Spiral Storage Example (1)

- Let's assume that we start from an active address space of 5 addresses. Let $d = 2$.
- We look for a value of S, that gives us 5 active addresses. $(d^S * (d-1) = 5 \implies 2^S = 5 \implies S = \log_2 5 = 2.3219)$
- First active address $= \lfloor d^S \rfloor = \lfloor 2^{2.3219} \rfloor = 5$.
- Last active address $= \lceil d^{S+1} \rceil - 1 = \lceil 2^{3.3219} \rceil - 1 = 9$.
- We want to see, where all keys will be distributed according to their hash value.
- For example, if $h(k) = 0.75$, $x = \lceil 2.3219 - 0.75 \rceil + 0.75 = 2.75 \implies y = \lfloor 2^{2.75} \rfloor = 6$.
- The resulting distribution according to hash values is given in the table below.

Spiral Storage Example (1)

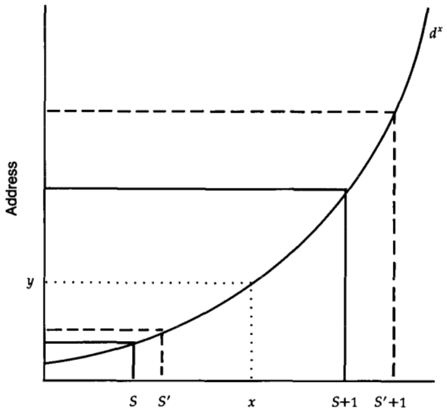


Address	Hash interval	Relative load
5	[0.3219, 0.5849)	0.263
6	[0.5849, 0.8074)	0.222
7	[0.8074, 1.0000)	0.193
8	[0.0000, 0.1699)	0.170
9	[0.1699, 0.3219)	0.152

Spiral Storage Expansion

- To increase the active address space we simply increase S to $S^{'}$.
- The keys that previously mapped into the range $[S, S^{'})$, now map into the range $[S + 1, S^{'} + 1)$.
- The new address range is approximately d times the old address range.
- The records stored in the bucket that disappears are relocated to the new buckets and the expansion is complete.
- The value $S^{'}$ is normally chosen so that exactly one bucket disappears.
- Most of the time an expansion creates two new buckets, but occasionally either one or three buckets are created.

Spiral Storage Expansion (2)



Spiral Storage Setback

- The most expensive part of the address calculation is the computation of 2^x.
- A function of this type is normally computed by approximating it with a polynomial of a fairly high degree.
- Let $f(x)$ be a function that approximates 2^x, $0 \leq x \leq 1$.
- Most suggested function is: $f(x) = \frac{a}{b-x} + c$, $0 \leq x \leq 1$.
- The values of the parameters a, b and c can be determined by fixing the value of $f(x)$ at three points.
- The resulting performance is very close to the performance obtained when using 2^x (The expansion rate is almost, but not exactly constant).

Performance Analysis

- As before, it is assumed that the overall load factor is kept constant and equal to $\alpha, \alpha > 0$.
- Expansion function 2^x is used.
- It is assumed that there are no deletions.
- For convenience, we can consider only the normalized address range $[1, 2)$.
- $p(y)$, denotes the probability that a key hashes to a (normalized) address in $[y, y + dy] \subseteq [1, 2)$.

Performance Analysis (2)

- $p(y) = \log_2(y + dy) - \log_2(y) = \log_2(1 + \frac{dy}{y}) = \frac{dy}{y ln 2}$.
- The insertion probability density function is: $\frac{1}{y ln 2}$.
- Over the normalized address range $[1, 2)$, the insertion probability density function is: $\frac{1}{y ln 2}$.
- The expected load factor of a bucket at address y is proportional to the insertion probability at y.
- Load factor: $\lambda(y) = \frac{c_1}{y ln 2}$, c_1 is a normalizing constant.
- Average load factor must equal α: $\int_1^2 \frac{c_1 dy}{y ln 2} = \alpha \implies c_1 = \alpha$.

Performance Analysis (3)

- The probability of a successful search hitting a bucket is proportional to the load factor of the bucket
- The probability of hitting a bucket with an address in $[y, y + dy)$ is $\frac{c}{y} dy$.
- The constant c is determined by the fact that the probability of a search hitting some bucket is one.
- $\int_1^2 \frac{c}{y} dy = 1 \implies c = \frac{1}{ln2}$.
- If a successful search hits a bucket with a load factor of λ, the expected cost is $s(\lambda) = 1 + \frac{\lambda}{2}$.

Performance Analysis (4)

- The expected cost of a successful search is:
  $\bar{S}(\alpha) = \int_1^2 s(\frac{\alpha}{y ln2})\frac{dy}{y ln2} = \int_1^2 (1 + \frac{\alpha}{2y ln2})\frac{dy}{y ln2} = \cdots = 1 + \frac{\alpha}{2}1.0407$
  .
- Respectively the expected cost of an unsuccessful search in a bucket with load factor λ is $u(\lambda) = \lambda$.
- The expected cost of an unsuccessful search is
  $\bar{U}(\alpha) = \int_1^2 u(\frac{a}{y ln2})\frac{dy}{y ln2} = \int_1^2 (\frac{a}{(ln2)^2})\frac{dy}{y^2} = \cdots = \alpha 1.0407$.