Dynamic Hash Tables

Selected Topics in Algorithms

ΑΛΜΑ, ΣΗΜΜΥ

Ευάγγελος Μαργέτης

Dynamic Hash Tables

- Dynamic set of keys $S = \{k_1, \ldots, k_n\}$.
- Table has m buckets indexed $0 \ldots m - 1$.
- Hash function $h : U \rightarrow \{0, \ldots, m - 1\}$.
- Insertions and deletions change n over time.
- Performance depends on expected comparisons.
- Chaining used to store collisions per bucket.
- Growth of n increases computational cost.

Dynamic Hash Tables

- Load factor defined as $\alpha = \frac{n}{m}$.
- Expected chain length equals α exactly.
- Successful search: $E_{\text{succ}} = 1 + \frac{\alpha}{2}$.
- Unsuccessful search: $E = \alpha$ under chaining.
- As α increases operations slow linearly.
- Fixed table size implies increasing α.
- Controlling α is fundamental objective.

Dynamic Hash Tables

- Final cardinality n is unknown beforehand.
- We need a mechanism to keep $\alpha \leq \alpha_{max}$.
- A classical solution is to allocate a larger table $m' > m$.
- Then we rehash all keys using $h_{\text{new}}(k_i)$.
- This rehash step has worst-case running time $\Theta(n)$.
- Such pauses are unacceptable in practical systems.
- Therefore we need smooth growth without full rehashing.

Dynamic Hash Tables

- It adapts Linear Hashing for in-memory use.
- It introduces Spiral Storage as an alternative method.
- It defines full data structures and split mechanics.
- It analyzes expected costs under uniform hashing.
- It derives search formulas across a full split cycle.
- It provides experiments comparing all approaches.

Dynamic Hash Tables

- Dynamic hashing uses buckets in memory.
- Keys are mapped by a base hash function.
- Chains store elements inside each bucket.
- The table expands as the load grows.
- The overall load factor is $\alpha = \frac{n}{B}$
- Expansion occurs when α exceeds limits.
- Costs count hash computations and comparisons.

Dynamic Hash Tables

- The table contains m active bucket positions $m = N \cdot 2^L + p$.
- Buckets are organized in segments of size N.
- The active range expands as p increases.
- Each bucket holds a chaining list of keys.
- Addressing uses two hash levels $h_L(k)$ and $h_{L+1}(k)$.
- The structure supports incremental bucket splitting.
- This representation enables smooth predictable growth.

Dynamic Hash Tables

- Linear Hashing stores buckets in fixed-size segments.
- A directory points to segments with capacity N.
- The active bucket count equals $m = N \cdot 2^L + p$.
- Each bucket has a chain storing its records.
- The split pointer p indicates the next bucket to grow.
- Hashing uses functions $h_L(k)$ and $h_{L+1}(k)$ for addresses.
- This structure enables incremental controlled expansion.

Dynamic Hash Tables

- Assume the table has N=4 buckets and the split pointer is p=1.
- A new key hashes to address 5 using h1, which corresponds to the bucket being split.
- We first insert the key using the old address and then split bucket 1.
- Keys that now match h1 move to the new bucket 4.
- The split pointer advances to p=2.
- This illustrates how growth occurs gradually.

Dynamic Hash Tables

- Linear Hashing uses two hash functions during splitting.
- Before splitting: the address is computed as: Insert → Equation → type: $h_0(K) = K \, mod \, 5$
- After splitting: the new address is Insert → Equation → type: $h_1(K) = K \, mod \, 10$
- Keys with last digit 0 satisfy $h_1(K) = 0$ and remain in bucket 0.
- Keys with last digit 5 satisfy $h_1(K) = 5$ and move to the new bucket 5.
- Only bucket 0 is scanned and only half of its keys relocate.

Dynamic Hash Tables

- Linear Hashing supports lookup, insertion and splitting.
- Successful lookup has expected cost $1 + \frac{\alpha}{2}$.
- Unsuccessful lookup has expected cost α.
- Average cost across a split cycle is $\bar{S}(\alpha) = 1 + \frac{5}{6}\alpha$ and $\overline{U}(\alpha) = \frac{11}{12}\alpha$.
- Each insertion performs about 1.5 extra relocations.
- Spiral Storage uses exponential mapping 2^x per access.
- Thus LH is asymptotically optimal and faster in practice than Spiral Storage.

Dynamic Hash Tables

- Successful search in a cycle costs $S(\alpha, x) = 1 + \frac{\alpha}{2}(2 + x - x^2)$.
- Unsuccessful search in a cycle costs $U(\alpha, x) = \frac{\alpha}{2}(2 + x - x^2)$.
- Averaging over all split positions gives $\bar{S}(\alpha) = 1 + \frac{5}{6}\alpha$.
- Unsuccessful average cost becomes $\overline{U}(\alpha) = \frac{11}{12}\alpha$.
- Each insertion causes about 1.5 extra moves.
- Cycle length and split frequency depend on $m = N \cdot 2^L + p$.
- Overall access cost remains $\Theta(\alpha)$ with smooth growth.

Dynamic Hash Tables

- Linear Hashing keeps each key in its correct bucket.
- A key maps to h0 or to h1 depending on the split pointer.
- Before splitting bucket i all keys use h0.
- After splitting, keys satisfying $h_1(K) = h_0(K) + N$.
- move to the new bucket.
- The invariant "p buckets are split" always holds.
- This ensures correct addressing during expansion.

Dynamic Hash Tables

- Let the table size be $m = N \cdot 2^{L} + p$ and p the next bucket to be split.
- After splitting one bucket the pointer p increases by one. Insert → Equation → type: $p := p + 1$
- If p reaches the end of the current round then a new level begins. Insert → Equation → type: $if \quad p = N \cdot 2^{L} \quad then \quad L := L + 1 \quad and \quad p := 0$
- These rules maintain the invariant that exactly p buckets have been split.
- They also ensure that growth proceeds smoothly one bucket at a time.

Dynamic Hash Tables

- We define the global load factor as $\alpha = n/m$.
- Let $x \in [0,1]$ denote the fraction of buckets already split during the expansion cycle.
- Before splitting, a bucket holds all keys addressed by h_0; after splitting, half of its keys move to the new bucket.
- The expected load of an unsplit bucket equals: $z = \alpha(1 + x)$.
- The expected load of a split bucket equals $z/2$.
- These quantities increase smoothly as x grows.

Dynamic Hash Tables

- A successful search examines half of the bucket's chain.
- If the bucket is split (probability x), the expected cost is: 1 + z/4
- If the bucket is unsplit (probability 1 − x), the expected cost is: 1 + z/2
- Thus the total expected successful search cost is: $S(\alpha, x) = x(1 + z/4) + (1 - x)(1 + z/2)$.
- After substituting z = α(1 + x) we obtain: $S(\alpha, x) = 1 + (\alpha/2)(2 + x - x^2)$.

Dynamic Hash Tables

- An unsuccessful search scans the entire chain.
- If the bucket is split, the expected load is z/2; otherwise it is z.
- Hence the total expected unsuccessful search cost is: $U(\alpha, x) = x(z/2) + (1 - x)z$.
- Substituting z = α(1 + x) yields: $U(\alpha, x) = (\alpha/2)(2 + x - x^2)$.

Dynamic Hash Tables

- During one expansion cycle, x increases uniformly from 0 to 1.
- The average successful search cost is: $S(\alpha) = \int_0^1 S(\alpha, x)dx$.
- Evaluating the integral gives: $S(\alpha) = 1 + (5/6)\alpha$.
- Similarly, the average unsuccessful search cost is: $U(\alpha) = \int_0^1 U(\alpha, x)dx = (11/12)\alpha$.

Dynamic Hash Tables

- Insertions cost an unsuccessful search plus split work.
- A split happens on fraction 1/α of insertions.
- The split bucket has load $\alpha(1 + x)$.
- Weighting by probability gives total cost $T_{\text{insert}} = 1.5$.
- Thus insertion runs in constant expected time.

Dynamic Hash Tables

- Insertion computes the bucket using h_L or h_{L+1}.
- If $\alpha > \alpha_{max}$, bucket p splits.
- Splitting moves keys whose new address $\text{addr}(k)$ is larger.
- After splitting, the pointer p increases by one.
- When p reaches $N \cdot 2^L$ it resets to zero.
- At that moment the level L increases by one.
- These invariants ensure smooth predictable expansion.

Dynamic Hash Tables

- Averaging over a full expansion cycle gives $S(\alpha) = 1 + \frac{5\alpha}{6}$ for successful searches.
- The average unsuccessful search cost over the cycle is $U(\alpha) = 11/12 \ * \ \alpha$.
- Only a fraction split frequency $= \frac{1}{\alpha}$ insertions trigger bucket splits.
- The bucket being split has expected load expected split bucket load z $= \alpha(1 + x)$.
- The expected extra hash computations per insertion are extra insert cost $= 1.5$.
- Linear Hashing therefore achieves constant amortized insertion cost.
- Performance is stable even when the table grows large.

Dynamic Hash Tables

- Spiral Storage uses an exponential address map $y = \lfloor d^x \rfloor$ for x ∈ [S, S+1].
- Active buckets lie in the interval $[d^S,\ d^{S+1})$ and growth shifts this interval by increasing S.
- When a bucket leaves the interval its keys move.
- Inverse mapping, let $y = 2^x$ and x = $log_2\, y$ (logical coordinates).
- Its main drawback is costly evaluation of d^x.

Dynamic Hash Tables

- Differentiating the inverse map gives density: $p(y) = \frac{1}{y \ln 2}$ which increases for small values of y.
- The expected bucket load equals: $\lambda(y) = \alpha/(yln2)$.
- The distribution is skewed toward the left side.
- This load profile drives the search costs.

Dynamic Hash Tables

- A successful search inspects half the expected load λ(y): $S(\alpha) = 1 + \int_1^2 (\lambda(y)/$

Dynamic Hash Tables

- Spiral Storage uses the same two-level structure and state variables shift the window right as S grows.
- Relocation uses the logarithmic mapping from $x = \log_d(y)$.
- Address evaluation requires exponentials d^x and inverse mapping requires logarithms $x = \log_d y$. These computations have higher latency than LH.
- Expected search cost remains proportional to α. However, mapping (constant) dominates runtime for large datasets.

Dynamic Hash Tables

- The experiments evaluate hashing performance under load $\alpha = \frac{n}{m}$.
- Random key sets were inserted to reach size $n \approx 10^6$.
- Search tests include successful and unsuccessful probes.
- Linear Hashing uses incremental splits controlled by p.
- Spiral Storage recomputes bucket locations using d^x.
- All methods are compared against rehash-based schemes Rehash.

Dynamic Hash Tables

TABLE II. Theoretically expected and observed average number of comparisons for a successful search in a linear hash table ($\alpha = 5$)

Number of records	Expected value	Observed average		
		File A	File B	File C
2000	3.81	3.84	3.84	3.84
4000	3.81	3.78	3.76	3.80
6000	3.68	3.67	3.66	3.72
8000	3.81	3.84	3.86	3.82
10000	3.56	3.60	3.56	3.52

- The theoretical cost of a successful search in Linear Hashing with load α = 5 is approximately 3.6 comparisons. The observed measurements for Files A, B, and C align extremely closely with this prediction, with deviations below one percent.
- This confirms that the load distribution model and the formula $1 + \frac{5}{6}\alpha$ accurately describe the behavior of Linear Hashing across different table sizes.
- As the number of records increases, the expected cost remains effectively constant, demonstrating that the incremental split mechanism maintains a stable average chain length.

Dynamic Hash Tables

TABLE III. Theoretically expected and observed average number of comparisons for a successful search using spiral storage ($\alpha = 5$)

Number of records	Expected value	Observed average		
		File A	File B	File C
2000	3.61	3.55	3.60	3.60
4000	3.61	3.57	3.56	3.59
6000	3.61	3.61	3.56	3.60
8000	3.61	3.58	3.59	3.59
10000	3.61	3.57	3.60	3.59

- The experimental results for Spiral Storage closely match the theoretical prediction of 3.61comparisons at load $\alpha = 5$.
- The observed values vary only slightly across different files and table sizes, remaining within a narrow band around the expected value.
- This confirms that the skewed load distribution $\lambda(y) = \frac{\alpha}{y \ln 2}$accurately models search cost in practice. Although Spiral Storage is computation-heavier than Linear Hashing, its lookup performance remains stable and predictable.

Dynamic Hash Tables

TABLE IV. Average CPU-time in milliseconds/key for loading and searching in a linear hash table

Test data	Loading $\alpha = 1$	Loading $\alpha = 5$	Loading $\alpha = 10$	Searching $\alpha = 1$	Searching $\alpha = 5$	Searching $\alpha = 10$
File A	0.88	0.97	1.15	0.34	0.41	0.50
File B	0.94	1.02	1.20	0.36	0.44	0.53
File C	1.06	1.23	1.53	0.41	0.53	0.69

TABLE V. Average CPU-time in milliseconds/key for loading and searching in a hash table organized by spiral storage

Test data	Loading $\alpha = 1$	Loading $\alpha = 5$	Loading $\alpha = 10$	Searching $\alpha = 1$	Searching $\alpha = 5$	Searching $\alpha = 10$
File A	1.25	1.17	1.34	0.41	0.48	0.57
File B	1.26	1.20	1.37	0.42	0.49	0.59
File C	1.40	1.43	1.71	0.47	0.59	0.75

- Linear Hashing consistently loads and searches faster than Spiral Storage across all datasets and load factors.
- The difference is most pronounced during loading, where Spiral Storage pays the cost of evaluating exponential and logarithmic functions. Search times also show a uniform advantage for Linear Hashing, reflecting its simpler address computation.
- Both methods scale smoothly as α increases, but Linear Hashing achieves strictly lower constant factors in practice.

Dynamic Hash Tables

- Measured search costs match the predicted values $S(\alpha) = 1 + \frac{\alpha}{2},\ U(\alpha) = \alpha$.
- Insertion times show smooth growth without pauses.
- Linear Hashing performs near the theoretical bounds $\bar{S}(\alpha) = 1 + \frac{5}{6}\alpha$ και

  $\bar{U}(\alpha) = \frac{11}{12}\alpha$
- Spiral Storage behaves correctly but is slower.
- Its overhead comes from evaluating powers like d^x.
- Trees show higher search costs under large α.
- Linear Hashing is the fastest across all experiments.

Dynamic Hash Tables

- Binary search trees build quickly for small key sets.
- However search time depends on the height of the tree.
- Unbalanced trees degrade badly under skewed insertion orders.
- Linear Hashing maintains bounded chain lengths and constant expected search time.
- Tree nodes require two pointers per record, increasing overhead.
- For large tables Linear Hashing clearly dominates search performance.
- Binary trees are competitive only for very small datasets.

Dynamic Hash Tables

- Fixed-size double hashing works best around load factor $\alpha \approx 0.8$.
- To support growth it periodically rehashes the entire table at cost $T_{\text{rehash}} = \Theta(n)$.
- This causes long pauses whenever a full reorganization is triggered.
- Linear Hashing grows by splitting one bucket at a time with amortized cost amortized growth cost $= O(1)$.
- At similar load factors both methods achieve comparable lookup times.
- However Linear Hashing avoids global rebuilds and jitter in response times.
- Dynamic double hashing therefore offers no clear advantage over Linear Hashing.

Dynamic Hash Tables

- Linear Hashing remains the most practical scheme.
- It offers smooth growth and predictable performance.
- Split operations add minimal amortized overhead.
- Spiral Storage is elegant but computationally heavy.
- Its exponential mapping makes it slower in memory.
- Experiments confirm LH is consistently more efficient.
- Dynamic hashing still depends critically on load α.