CSC 212: Data Structures and Abstractions

Hash Tables

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

Data Structure	Worst-case			Average-case			Ordered?
	insert at	delete	search	insert at	delete	search	
sequential (unordered)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	No
sequential (ordered) binary search	O(n)	O(n)	O(log n)	O(n)	O(n)	O(log n)	Yes
BST	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)	Yes
2-3-4	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes

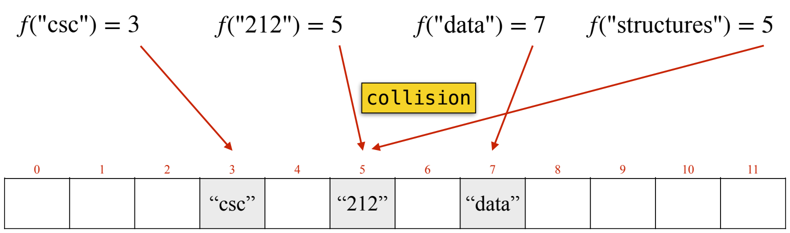
Can we do better?

Random access memory

- Random Access Memory (RAM) represents a fundamental principle in computer science
  - ✓ it allows the retrieval of any element in constant time $O(1)$, regardless of its position within the memory block
  - ✓ this principle is most commonly observed through arrays
- Arrays in C++
  - ✓ contiguous memory allocation
  - ✓ homogeneous elements (same data type)
  - ✓ fixed-length (traditional arrays have predetermined size)
  - ✓ zero-based indexing

Hash tables

- A hash table is a data structure that implements an associative array
  - ✓ the array can store keys (**set**), or key-value pairs (**map**)
  - ✓ a **hash** function is used to compute an index, that can be used to find a desired key in the array
  - ✓ provides an efficient way to implement sets or dictionaries



Hash function

Hash function

- A hash function is a function that maps an input key to some integer value
  - ✓ must be <u>deterministic</u> (same input produces the same output)
  - ✓ should be <u>well-distributed</u> (the numbers produced are as spread out as possible)
  - ✓ should be <u>efficient</u> to compute

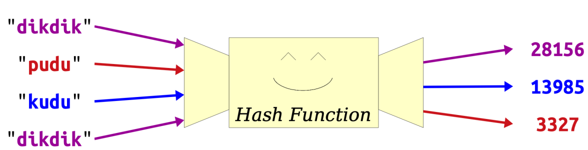


Image credit: CS106B @ Stanford

Hash functions

- Space efficiency
  - ✓ making all keys equally possible requires a huge array, even if we only have a couple of elements
  - ✓ **idea**: use a hash function, but modify the result to be within a smaller range (the **<u>capacity</u>** of the array)

```
// if hash() returns non-negatives
index = hash(key) % capacity

// if hash() returns any integer
index = abs(hash(key) % capacity)
```

`The load factor in a hash table is the ratio of N, the number elements, to M, the total capacity`

Practice

- Which of the following tables is a better choice?
- What is the load factor?

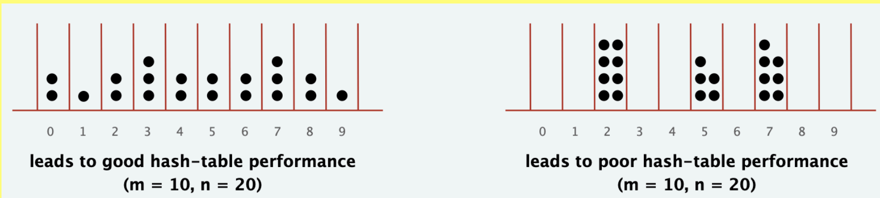


Image credit: COS 226 @ Princeton

Collisions

- Occurs when two different keys hash to the same index in the hash table
- Resolution:
  - ✓ **separate chaining**: each slot in the hash table contains a collection of all the keys that hash to that index
  - ✓ **open addressing**: if a collision occurs, the algorithm searches for the next available slot in the hash table
  - ✓ open addressing is more space-efficient than chaining, but it can be slower

Designing hash functions

- Hash functions on different data types
  - ✓ **integers**: use the integer value as the hash value
  - ✓ **floats**: convert to binary and use the integer value as the hash value, or manipulate the bits (e.g. XOR the mantissa and exponent)
  - ✓ **strings**: use the $31x + y$ rule or other variants
  - ✓ **compound objects**: use the $31x + y$ rule or other variants
- Mapping hash values into $[0, M - 1]$
  - ✓ **M is prime**: helps distributing keys more uniformly, minimizing collisions, `hash % M`
  - ✓ **M is a power of two**: modulo operation can be replaced with a faster bitwise AND operation, `hash & (M−1)`

Hash functions (other uses)

- Storing passwords
  - ✓ hash the password and store the hash in the database
- File verification
  - ✓ hash the file (checksum) and compare the hash with the stored hash
  - ✓ e.g. when downloading a file, vendors publish a hash value, client checks whether hash matches, otherwise file is corrupted
- Examples (one-way hashes)
  - ✓ hash functions that are difficult to reverse or to find two keys that map to the same value
  - ✓ MD5, SHA-1, SHA-256, SHA-512, SHA3-512, …

Separate chaining

Separate chaining

- Idea
  - solve collisions by storing a linked list at each index
  - assume duplicated keys are not allowed
- Operations (assume a hash function *h*)
  - **insert**: add the new key to the linked list at *h(key)*
    - insert at front of the list for faster operations, no need to keep the keys on each list in sorted order
  - **search**: search the linked list at *h(key)*
  - **delete**: remove the key from linked list at *h(key)*
- Comments
  - the linked list can be replaced by a balanced tree or another data structure to improve access time

Practice

- Perform the following operations
  - insert(L, 11), delete(D), insert(M, 12), delete(E), search(C)
    - assume: insertions occur at front of the lists, hash(L)=3, hash(M)=0

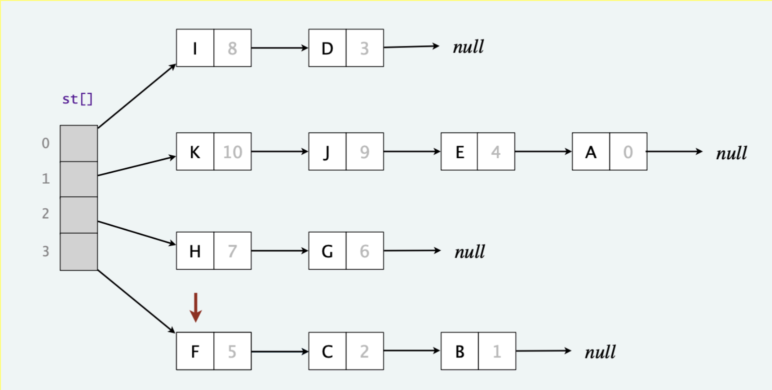
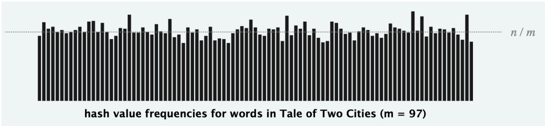


Image credit: COS 226 @ Princeton

Analysis

- Uniform hashing assumption
  - assume the hash function is a good one, and all keys are uniformly distributed



- Load factor (α)
  - the ratio of the number of keys (N) to the number of slots (M)

$$\alpha = \frac{N}{M}$$

- Time complexity
  - average case for search and delete is $O(c + \alpha)$, where c is the cost of the hash function
  - worst case for search and delete is $O(c + N)$ — all the keys hash to the same index
  - insert in all cases can be done in $O(c)$ if inserting at front of the list

Resizing a hash table

- Growing to a larger array when α exceeds a threshold
  - ✓ create a new table with larger capacity and rehash all the keys

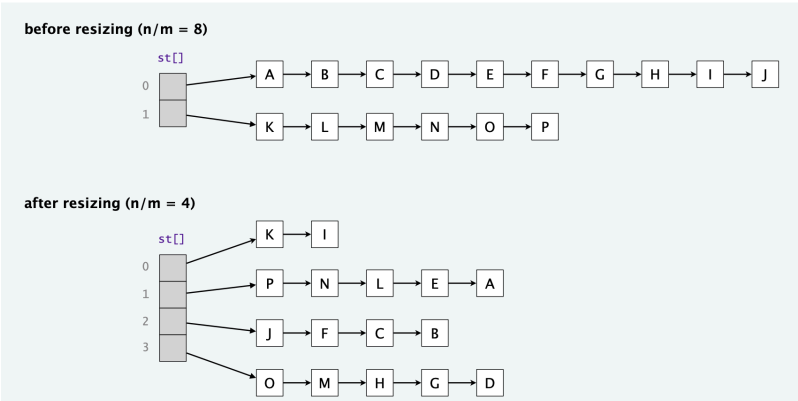


Image credit: COS 226 @ Princeton

Practice

- Insert the following keys into a hash of size M=4
  - 4, 2, 1, 10, 21, 32, 43, 3, 51, 71
- Resize the table to M=11

Considerations

- Choices for α
  - ✓ too small, the hash table will be too large and waste space
  - ✓ too large, the hash table will be too small and cause collisions
- Typical values
  - ✓ between 0.5 and 1.0 often provide a reasonable balance of space efficiency and lookup performance
  - ✓ higher load factors (>1.0) remain functional but with degraded performance characteristics
  - ✓ for performance-critical applications, implementers should conduct benchmarks with representative data sets to determine the optimal load factor for their specific use case

Data Structure	Worst-case			Average-case			Ordered?
	insert at	**delete**	**search**	**insert at**	**delete**	**search**	
sequential (unordered)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	No
sequential (ordered) binary search	O(n)	O(n)	O(log n)	O(n)	O(n)	O(log n)	Yes
BST	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)	Yes
2-3-4	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Hash table (separate chaining)	O(1)	O(n)	O(n)	O(1)*	O(1)*	O(1)*	No

(*) assumes uniform hashing and appropriate load factor

Open addressing

Open addressing

- Idea
  - solve collisions by "*probing*"
    - searching for the next available slot in the hash table
    - each slot holds a single element — if using *key-value* pairs, maintain separate arrays
  - assume duplicated keys are not allowed and $M \geq N$
- Operations (assume a hash function h)
  - **insert**: if *h(key)* is empty, place the new key there, otherwise, probe the table using a predetermined sequence until a slot is found
  - **search**: if h(key) contains the key then return successfully, if not, probe the table using the established sequence until either finding the key or an empty slot, which indicates that the key is not present in the table
  - **delete**: upon finding the key, cannot mark the slot as empty, as this would disrupt future search operations by prematurely terminating probe sequences, instead, mark the slot as ***"deleted"***
- Comments
  - approach is more space-efficient than chaining, but it can be slower (better with $\alpha \approx 0.5$)

Probing

- $h(k, i)$ is the position for the i-th probe
- $h'(k)$ is the initial hash value of key k
- i is the probe number ($i = 0,1,2,\ldots$)
- m is the table size
- $h_1(k)$ is the primary hash function
- $h_2(k)$ is the secondary hash function

- Linear probing
  - moves to the next available index
  - $h(k, i) = (h'(k) + i) \mod m$
- Quadratic probing
  - moves to the next available index using a quadratic function
  - $h(k, i) = (h'(k) + i^2) \mod m$
- Double hashing
  - moves to the next available index using a secondary hash function h_2 (should not evaluate to 0)
  - $h(k, i) = (h_1(k) + i \cdot h_2(k)) \mod m$

Practice

- Perform the following operations (assume linear probing)
  - search(w), delete(z), delete(w), search(r), insert(c), insert(d), insert(e)
    - assume: h(z)=2, h(x)=7, h(r)=7, h(w)=7, h(y)=14, h(a)=12, h(c)=8, h(d)=15, h(e)=14

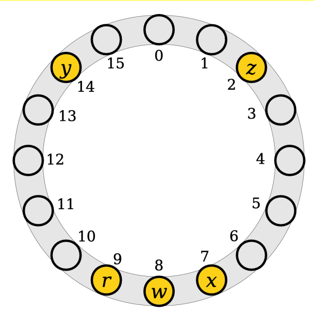


Image credit: CS106B @ Stanford

Practice

- Insert the following keys into a hash of size M=7
  - 4, 2, 1, 10, 21, 32, 43, 3, 51, 71, 17, 24

✓ use linear probing

✓ use quadratic probing

✓ use double hashing with $h_2(k) = 1 + (k \mod 10)$

Image credit: CS106B @ Stanford

Data Structure	Worst-case			Average-case			Ordered?
	insert at	**delete**	**search**	**insert at**	**delete**	**search**	
sequential (unordered)	O(n)	O(n)	O(n)	O(n)	O(n)	O(n)	No
sequential (ordered) binary search	O(n)	O(n)	O(log n)	O(n)	O(n)	O(log n)	Yes
BST	O(n)	O(n)	O(n)	O(log n)	O(log n)	O(log n)	Yes
2-3-4	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Red-Black	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)	Yes
Hash table (separate chaining)	O(n)	O(n)	O(n)	O(1)*	O(1)*	O(1)*	No
Hash table (open addressing)	O(n)	O(n)	O(n)	O(1)*	O(1)*	O(1)*	No

(*) assumes uniform hashing and appropriate load factor