# CSC 212: Data Structures and Abstractions

## Hash Tables

Prof. Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Fall 2025

| 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?

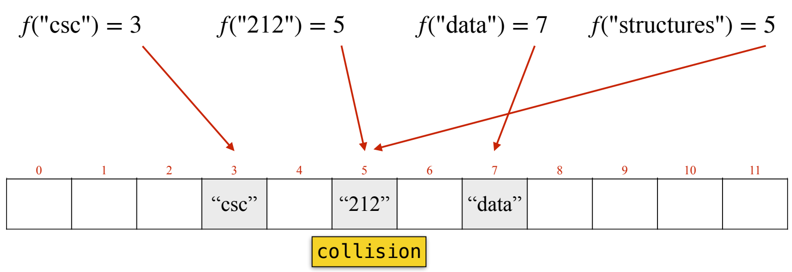


# Random access memory

- Random Access Memory (RAM)
  - ✓ fundamental principle in computer science
  - ✓ enables constant-time access to any memory location, given its address — foundation for efficient array operations
  - ✓ arrays exploit RAM by mapping indices directly to memory addresses
- Arrays in C++
  - ✓ contiguous memory allocation
  - ✓ homogeneous elements (same data type)
  - ✓ fixed size (determined at creation, cannot be modified)
  - ✓ zero-based indexing
  - ✓ $O(1)$ random access

# Hash tables

- A hash table is a data structure that implements an associative array
  - ✓ stores keys (**set**), or key-value pairs (**map**)
  - ✓ uses a **hash** function to compute an array index from a key
  - ✓ provides average-case $O(1)$ for search, insertion, deletion






# Hash function

# Hash function

- A hash function maps an input key to an integer value
  - ✓ hash value is then mapped to a valid array index using modulo
- Essential properties
  - ✓ **deterministic**: same key must always produce the same hash value
  - ✓ **uniform distribution**: hash values should be spread evenly
  - ✓ **efficient to compute**: execute in $O(1)$ time

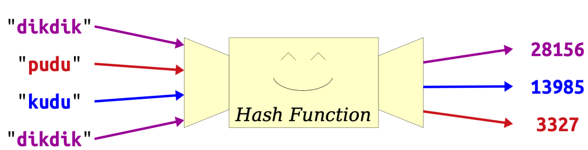



Image credit: CS106B @ Stanford



# Hash functions

- Space efficiency
  - ✓ supporting all possible keys directly would require enormous arrays => use a hash function to map keys to a much smaller array of size $m$ (the **capacity** 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** $\alpha$ 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 $\alpha$?

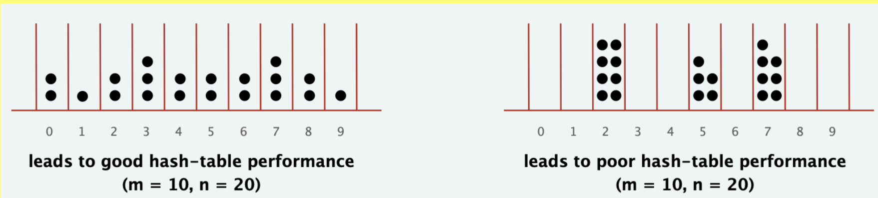



Image credit: COS 226 @ Princeton



# Collisions

- Definition
  - ✓ occurs when two distinct keys hash to the same index in the table
    - inevitable consequence when $\alpha > 1$
- Resolution strategies:
  - ✓ **separate chaining**: each table slot contains a linked list (or other secondary structure) of all elements hashing to that index
    - insertion $O(1)$, search/delete $O(1 + \alpha)$
  - ✓ **open addressing**: all elements stored directly in the table, collisions trigger probe sequences
    - open addressing is more space-efficient than chaining, but it can be slower



# Designing hash functions

- Design principles
  - ✓ good hash functions satisfy: determinism, efficiency, and uniform distribution
- Hash functions on different data types
  - ✓ **integers**: may use the integer value as the hash value, or may apply transformations to break patterns
  - ✓ **floats**: convert to binary and treat as an integer, or manipulate the bits (e.g. XOR the mantissa and exponent)
  - ✓ **strings**: use the polynomial rolling hash ( $31x + y$ rule) or other variants
  - ✓ **compound objects**: combine hash values of individual fields, may use the $31x + y$ rule or others



# Designing hash functions

- Importance of a hash function
  - ✓ determines hash table's storage capacity
  - ✓ directly impacts collision probability
  - ✓ influences overall data structure performance
- Size selection strategies — mapping hashes into $[0, M-1]$
  - ✓ **M is prime**: safe default, helps distributing keys more uniformly, minimizing collisions
    - hash % $M$
  - ✓ **M is a power of two**: faster, enables fast modulo operation via bitwise AND
    - hash & $(M - 1)$

# Hash functions (beyond hash tables)

- Storing passwords
  - ✓ never store plaintext passwords — use cryptographic hash functions
- File verification
  - ✓ checksum mechanism — detect file corruption and provide data integrity validation
  - ✓ verification process: generate and publish file hash, client recomputes hash of downloaded file, compare computed and published hashes
- Cryptographic hash functions
  - ✓ one-way property: computationally infeasible to reverse the hash, find input producing specific hash, generate collision
  - ✓ MD5, SHA-1, SHA-256, SHA-512, SHA3-512, …



# Separate chaining

# Separate chaining

- Collision resolution mechanism
  - ✓ utilizes a linked list at each hash table index (array of linked lists)
  - ✓ guarantees $O(1)$ average-case and $O(n)$ worst-case search times
  - ✓ supports dynamic memory allocation
  - ✓ maintains key uniqueness constraint
- Core operations (assume a hash function $h$)
  - ✓ **insert**: add a new key (or key/value pair) to the linked list at $h(key)$
    - insert at front 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 (or key/value pair) from linked list at $h(key)$
- Alternative collision resolution
  - ✓ replace linked list with a self-balancing tree (e.g. Red-Black, AVL), guarantees $O(\log n)$ worst-case search



# 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

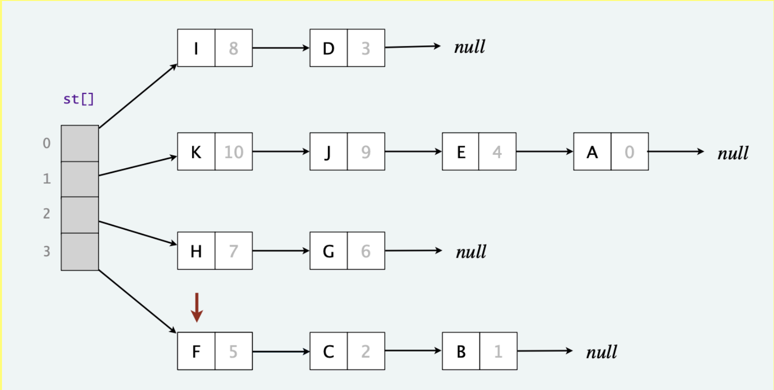



Image credit: COS 226 @ Princeton

```cpp
#include <string>
#include <iostream>
#include <list>
#include <vector>

const size_t tableSize = 101;
typedef std::string Key;
typedef int Value;
typedef std::pair<std::string, int> HashEntry;
typedef std::list<HashEntry> HashChain;
typedef std::vector<HashChain> HashTable;

size_t hash(const Key& k) {
    size_t hashValue = 0;
    for (char ch : k) {
        hashValue = hashValue * 31 + ch;
    }
    return hashValue;
}
```



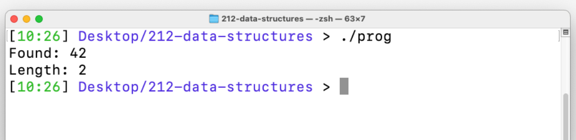
```cpp
void insert(HashTable& table, const Key& key, const Value& value) {
    size_t index = hash(key) % tableSize;
    table[index].push_front({key, value});
}

Value* search(HashTable& table, const Key& key) {
    size_t index = hash(key) % tableSize;
    for (auto& entry : table[index]) {
        if (entry.first == key)
            return &entry.second;
    }
    return nullptr;
}

bool remove(HashTable& table, const Key& key) {
    size_t index = hash(key) % tableSize;
    auto& list = table[index];
    for (auto it = list.begin(); it != list.end(); ++it) {
        if (it->first == key) {
            list.erase(it);
            return true;
        }
    }
    return false;
}
```



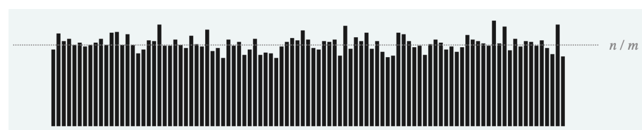
```cpp
int main() {
    HashTable table(tableSize);
    insert(table, "example", 42);
    insert(table, "test", 84);
    insert(table, "csc", 126);
    insert(table, "212", 168);
    Value* val = search(table, "example");
    if (val) {
        std::cout << "Found: " << *val << std::endl;
    }
    remove(table, "example");
    remove(table, "test");
    size_t count = 0;
    for (const auto& chain : table) {
        count += chain.size();
    }
    std::cout << "Length: " << count << std::endl;
    return 0;
}
```






# Analysis

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




hash value frequencies for words in Tale of Two Cities (m = 97)

- Load factor ($\alpha$)
  - the ratio of the number of keys ($n$) to the number of slots ($m$)

$$\alpha = \frac{n}{m}$$

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

# Resizing a hash table

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

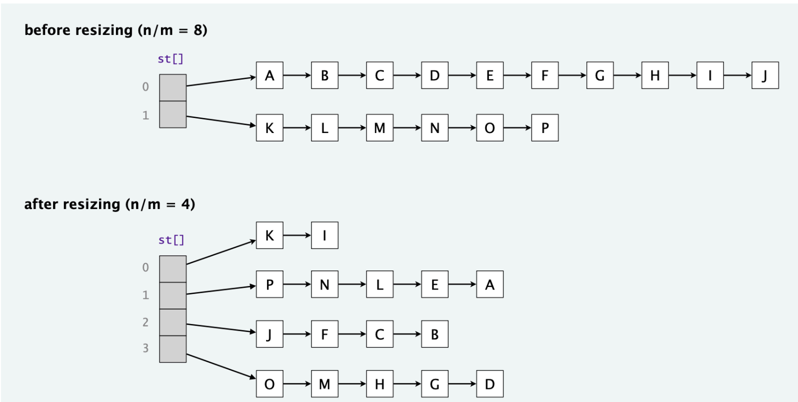



Image credit: COS 226 @ Princeton



# 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 $\alpha$
  - ✓ too small, results in excessive table size with inefficient space utilization
  - ✓ too large, results in insufficient table size, increasing collision probability and degrading lookup performance
- 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
  - ✓ for performance-critical applications, conduct benchmarks with representative data sets to determine the optimal load factor



| **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

- Collision resolution mechanism
  - searching for next available slot (*probing*)
  - single-element per slot constraint, however requires careful deletion handling
  - assume duplicated keys are not allowed and $M \geq N$
- Core operations (assume a hash function $h$)
  - **insert**: if $h(key)$ is empty, place the new key (or key/value pair) 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 a predetermined 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

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

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



# 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(w)=7, h(r)=7, h(y)=14, h(a)=12, h(c)=8, h(d)=15, h(e)=14

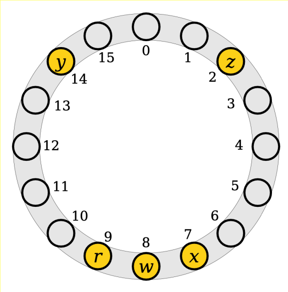



Image credit: CS106B @ Stanford

# Practice

- Insert the following keys into a hash of size M=13
  - 4, 2, 1, 10, 21, 32, 43, 3, 51, 71, 17

✓ 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



# Unordered associative containers (STL)

Unordered associative containers implement data structures that can be quickly searched – $O(1)$ average-case complexity

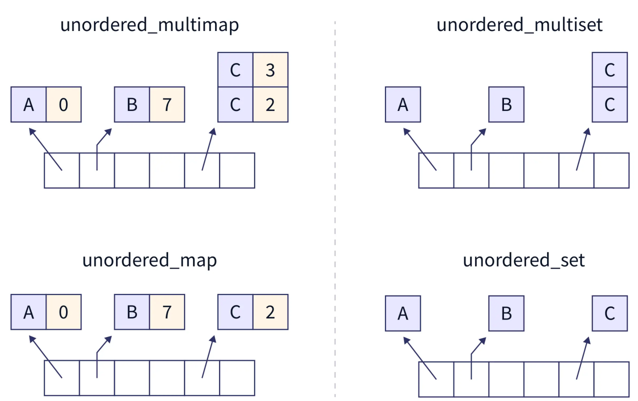



Image credit: https://www.scaler.com/topics/cpp/containers-in-cpp/