CS213/293 Data Structure and Algorithms 2023

Lecture 5: Dictionary

Instructor: Ashutosh Gupta

IITB India

Compile date: 2023-08-12

Topic 5.1

Problem of dictionary

Storing maps/dictionary

Definition 5.1

A *Dictionary* stores values so that they can be found efficiently using *keys*.

Example 5.1

A dictionary may contain bank accounts.

- ▶ *Bank account number is the key*
- ▶ *The information about your account is the value*
    - ▶ *current amount, name, address, etc*
- ▶ *To take any action on an account, one needs the key*

Dictionary (Map) container

In C++ and many languages, dictionaries are called maps.

map supports the following interface.

▶ `map<Key,T> m` : allocates new map m
▶ `m.at(e)` : access specified value (throws an exception when value is missing)
▶ `m[key] = e` : Inserts key-value pair.
▶ `m.erase(key)` : removes key-value pair.

Some support functions

▶ `m.empty()` : checks whether the map is empty
▶ `m.size()` : returns the number of key-value pairs

Order over keys

Two kinds of keys:

- ▶ Ordered: keys are compared using less than, greater than, and equality
  - ▶ The default map in C++ assumes keys are ordered.
    Reference: https://en.cppreference.com/w/cpp/container/map

- ▶ Unordered: keys are compared only using equality
  - ▶ For unordered keys, use unordered_map in C++.
    Reference: https://en.cppreference.com/w/cpp/container/unordered_map

Since all data is bit-vector, we can always define order over keys. However, the user decides if the keys are ordered or unordered.

Implementation choices

- arrays, linked lists
- Hash table (`unordered_map` in C++)
- Binary trees
- Red/black trees (`map` in C++)
- AVL trees
- B-trees

Actions on dictionary

We need to design a dictionary data structure keeping in mind the following three important actions on dictionaries.

- ▶ Insertion
- ▶ Deletion
- ▶ Search

Topic 5.2

Design choices for dictionaries

Cost of searching for keys

We have seen in lecture 1 the cost of searching for the position of a key.

Ordered keys

▶ Binary search is $O(log\ n)$

Unordered keys

▶ Linear search is $O(n)$

Dictionaries via unordered keys

[2,10,8,19,34,23]

► Searching and deletion is $O(n)$

► Insertion is $O(1)$

Application: Log files, (frequent insertion, but rare searches and deletion)

Dictionaries via ordered keys on arrays

$$[2,8,10,19,23,34]$$

- Searching is $O(\log n)$
- Insertion and deletion is $O(n)$
  - Need to shift keys before insertion/after deletion

Application: Look-up tables (e.g. precomputed values for trigonometric functions), (frequent searches, but rare insertion and deletion)

Exercise 5.1
Can we use a linked list?

One crazy idea: direct addressing!

Consider application: caller id. We need a map from phone numbers to names.

We have 10-digit-long phone numbers. So let us allocate an array A of size 10^{10}.

Names are stored at the phone number index.

Null	Ashutosh	Null	Null	Divya	Null
9898927391	9898927392	9898927393	9898927394	9898927395	9898927396

▶ All operations are $O(1)$
▶ Huge waste of space.

Exercise 5.2
Do we have $O(1)$ cost in the above?

Topic 5.3

Hash table

Can we improve direct addressing?

Can we somehow avoid the waste of space and still get the benefit of direct addressing?

Let the table size is m and the number of keys is n.

We will design a data structure, where $O(1)$ expected time for all operations and storage is $O(m + n)$.

m is roughly of same size as n.

Hashing

We choose a function, called the hash function,

$$h : Keys \rightarrow HashValues$$

such that $|HashValues| = m$.

We use $h(key)$ to index the storage array instead of *keys*.

We assume the time to compute $h(key)$ is $\Theta(1)$.

Example: Hashing

Example 5.2

Let us suppose we want to store called IDs of phone numbers from your contacts in your phone.

You probably have less than 1000 contacts.

Let us use $h(number) = (number \bmod 1000)$.

We create an array of 1000 entries and store the contact names as follows. Let us suppose Ashutosh's phone number is 9898927392 and Divya's phone number is 9869755395.

Null	Ashutosh	Null	Null	Divya	Null
391	392	393	394	395	396

One problem: Let us suppose Akhil's phone number is 9868733392. We have a collision.

Collision resolution: chaining

In the case of $h(k_1) = h(k_2)$, we cannot store two values in the same place on the array.
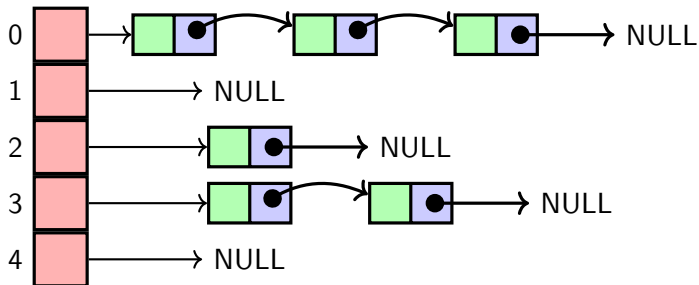
We maintain a linked list for key-value pairs that have the same hash value of their keys and a table (array) indexed by the hash values points to the linked lists.
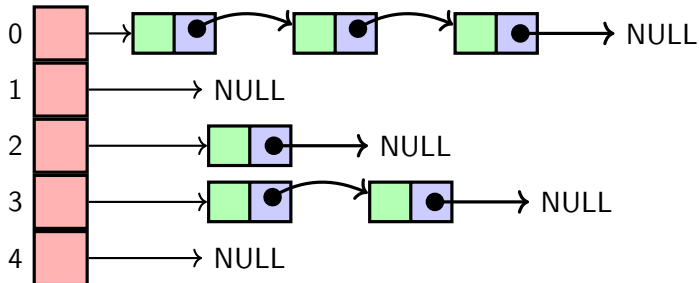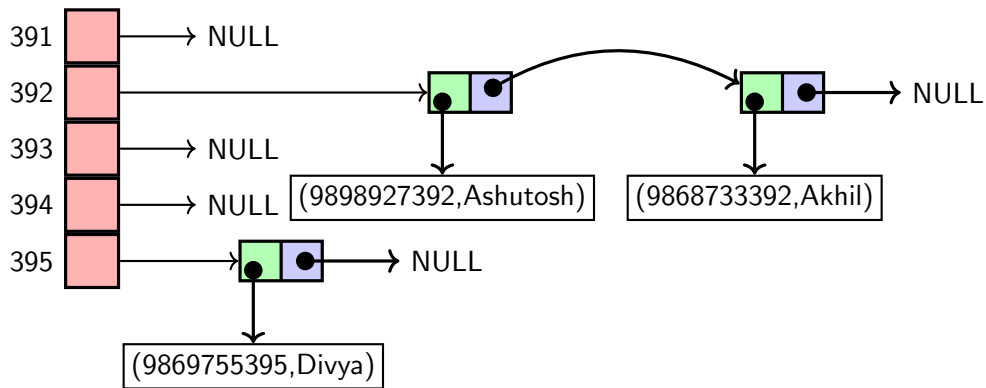
Collision resolution: chaining(2)

To search/insert/delete a (key,value) pair

- using $h(key)$ find position in the table
- search/insert/delete the pair in the linked list of the position.

Example: telephone directory

Topic 5.4

Analysis of hash functions

A good hash function

A good hash function

- ▶ distributes keys evenly amongst the positions.
- ▶ has a low probability of collision.
- ▶ is quick to compute.

Good hash functions are rare - Birthday paradox!

Exercise 5.3
What is a bad hash function?

Load factor

If n is large as compared to m, there are a greater number of collisions.

We define load factor $\alpha = \dfrac{n}{m}$.

Keep α roughly around 1.

- If α is too small, we are wasting space.
- If α is too large, we have long chains.

Exercise 5.4
What to do if α is not known upfront?

Topic 5.5

Designing hash functions

Hash function design

$$h : Keys \rightarrow \{0, ..., m-1\}$$

m is the size of hash table!

Keys can be of a variety of types.

- ▶ Biometric fingerprints
- ▶ Addresses
- ▶ Words of language dictionaries

Usually, h is the composition of the following functions.

- ▶ *code* : $Keys \rightarrow \mathbb{Z}$
- ▶ *compression* : $\mathbb{Z} \rightarrow \{0, ..., m-1\}$

$$h = code \circ compression$$

Useful functions for code

▶ Integer cast: Interpret the bit representation of the key as an integer, if the representation is less than the size of a word (32 bits/64 bits)

▶ Component sum: If the representation is longer than a word, sum the blocks of 8-bits to compute the integer code.

Example 5.3

$code($ "Disaster"$) = $ 'D'$+$ 'i'$+$'s'$+$'a'$+$'s'$+$'t'$+$'e'$+$'r'
$$= 0x44 + 0x69 + 0x73 + 0x61 + 0x73 + 0x74 + 0x65 + 0x72 = 0x33F$$

Example 5.4

Is this a good coding scheme?

Useful functions for code: polynomial accumulation

▶ Let $a_0, ..., a_k$ be the list of 8-bit blocks of the binary representation of the *key*.

$$code(a_0 a_1 .... a_k) = a_0 + a_1 x + a_2 x^2 + ... + a_k x^k$$

where x is a constant.

▶ The idea is borrowed from error-correcting codes (e.g. Reed-Solomon codes)

▶ Observation: the choice of $x = 33, 37, 39,$ or 41 gives at most 6 collisions in English vocabulary of 50K+ words. (Please check the claim!)

Exercise 5.5
How can we efficiently compute the polynomial?

Commentary: Usually the polynomial is computed using Horner's rule or precomputed values of x^k. This kind of encoding is widely used in a_i. Overflow is ignored in the computation.

unordered_map in C++ uses Murmurhash2 for code

```cpp
size_t _Hash_bytes(const char* buf, size_t len, size_t seed) {
    const size_t m = 0x5bd1e995;
    size_t hash = seed ^ len;
    while(len >= 4) { // Mix 4 bytes at a time into the hash.
        size_t k = *((const size_t*)buf);
        k *= m; k ^= k >> 24; k *= m;
        hash *= m; hash ^= k;
        buf += 4; len -= 4;
    }
    size_t k;
    switch(len) { // Handle the last few bytes of the input array.
        case 3: k = buf[2]; hash ^= k << 16;
        case 2: k = buf[1]; hash ^= k << 8;
        case 1: k = buf[0]; hash ^= k; hash *= m;
    };
    hash ^= hash >> 13; hash *= m; hash ^= hash >> 15; //Do final mixes.
    return hash;
}
```

Commentary: The above code is from https://github.com/gcc-mirror/gcc/blob/master/libstdc++-v3/libsupc++/hash_bytes.cc

Design of compression

Remainder compression:

$$compression(e) = e \bmod m$$

Here the size of the table matters.

▶ If $m = 2^k$, the least significant bits of e determine the position in the table. If the output of *code* is not uniformly distributed, we do not have enough randomization.

▶ If m is a prime, *compression*(e) will return uniformly distributed output. Rule of thumb: stay away from powers of 2.

Example 5.5

Let us suppose, we want to store 2000 *keys and we are ok with three collisions.*

A good choice of m is 701, *which is* prime *near* 2000/3 *and* away *from powers of* 2.

Design of compression(2)

Multiplicative compression:

$$compression(e) = \lfloor m\{ae\} \rfloor,$$

where $a \in (0, 1)$ is a constant.

▶ Here the size of the table does not matter.

▶ However, some as work better than others. Folklore, $\dfrac{\sqrt{5}-1}{2}$ (golden ratio) works well!

Exercise 5.6
Show compression(e) $\in \{0, .., m-1\}$

Commentary: For extended discussion look at The Art of Computer Programming. Volume 3. Sorting and Searching, by Donald Knuth

Design of compression(3)

MAD(multiplication, add, divide) compression:

$$compression(e) = |ak + b| \bmod m,$$

where $a, b \in \mathbb{Z}$ are constants.

▶ Eliminates patterns in input keys if m does not divide a.

▶ The technique is borrowed from pseudo-random generators!

Topic 5.6

Open addressing: an alternative to chaining!

Open addressing

Open addressing is another way of handling collision.

- ▶ The method needs $\alpha \leq 1$
- ▶ Each table entry has a key or Null
- ▶ We may have to examine many positions for the search

Hash function for open addressing

A slight modification of the hash function.

$$h : Keys \times \{0, .., m-1\} \to \{0, .., m-1\}$$

such that $h(k, 0), ..., h(k, m-1)$ is a permutation of $0, ..., m-1$ for any key k.

Example 5.6
Let $m = 5$.

For some key k,

$$h(k, 0), ..., h(k, 4) = 3, 0, 2, 4, 1.$$

Hash function for open addressing(2)

- $h(key, 0)$ is our usual hash function to place the key.

- $h(key, i)$ is an alternative available choice to place the key if earlier choices $h(key, j)$ for each $j < i$ are occupied.

Open addressing insert

Algorithm 5.1: OpenAddressInsert(key)

---
1 **if** *Table is full* **then**
2 | error;
3 i := 0;
4 **do**
5 | probe := h(k,i);
6 | i = i + 1;
7 **while** *table[probe] is occupied and i < m*;
8 table[probe] = k;

Linear probing

Linear probing is a special case of open addressing.

In linear probing, we chose h as follows

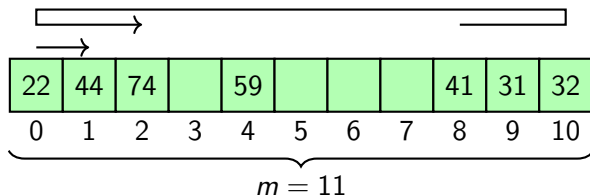$$h(k, i) = (h(k, 0) + i) \bmod m \qquad \text{for each } i > 0.$$

If a position is occupied, take the next one.

Example: insertion in linear probing

Example 5.7

Let $m = 11$ and $h(k, 0) = k \bmod 11$.

Let us consider the following sequence of insertions: 41, 22, 44, 59, 32, 31, 74

$$m = 11$$

Open addressing search

Algorithm 5.2: OpenAddressSearch(key)

---
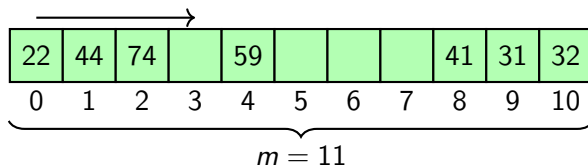
1 i := 0;
2 **do**
3   probe := h(k,i);
4   **if** table[probe] == k **then**
5   **return** probe;
6   i = i + 1;
7 **while** (table[probe] is occupied or *has tombstone*) and i < m;
8 **return** -1;

Example: search in linear probing

Example 5.8

Let $m = 11$ and $h(k, 0) = k \bmod 11$.

Let us search for 33 in the following table. We will examine locations from 0 to 3.

22	44	74		59				41	31	32
0	1	2	3	4	5	6	7	8	9	10

$$m = 11$$

Exercise 5.7

How many locations will we examine for the following searches?
- *74*
- *61*
- *44*
- *43*

Example: deletion in open addressing

Example 5.9

Let $m = 11$ and $h(k, 0) = k \bmod 11$.

Let us the delete key at position 1 in the following table. Will it be correct?

We need to place a marker (tombstone) to indicate that something was here such that we continue to search 74 correctly.

22	X	74		59				41	31	32
0	1	2	3	4	5	6	7	8	9	10

$$m = 11$$

Deletion in open addressing

Algorithm 5.3: OpenAddressDelete(key)

1 probe = OpenAddressSearch(key);
2 **if** *probe ≥ 0* **then**
3   table[probe] = 'X'   // Tombstone marker 'X' indicates that the place was occupied!

> We can reuse the tombstone location for insertion
> but assume it is occupied for search.

Exercise 5.8
After many deletions, the performance of the search degrades. How can we recover performance?

Topic 5.7

Problems

Problem: birthday paradox

Exercise 5.9

Given that k elements have to be stored using a hash function with target space n. What is the probability of the hash function having an inherent collision? What is an estimate of the probability of a collision in the insertion of N elements?

Hint: Stirling's approximation $\sqrt{2\pi n} \left(\dfrac{n}{e}\right)^n e^{\frac{1}{12n+1}} < n! < \sqrt{2\pi n} \left(\dfrac{n}{e}\right)^n e^{\frac{1}{12n}}$

Problem: analysis of linear probing

Exercise 5.10

Let $C(i)$ be the chain of array indices that are queried to look for a key k in linear probing where $h(k) = i$.

a. How does this chain extend by an insertion, and how does it change by a deletion?

b. A search for a key k ends when an empty cell is encountered. What if we mark the end of $C(i)$ with an end marker. We stop the search when this marker is encountered. Would this work? Would this be efficient?

c. Is there a way of not using tombstones?

Exercise: Double hashing

Exercise 5.11

Let $m = 11$, $h_1(k) = (k \bmod 11)$, $h_2 = 6 - (k \mod 6)$.

Let us use the following hash function for an open addressing scheme.

$$h(k, i) = h_1(k) + i * h_2(k).$$

What will be the state of the table after the following sequence of insertions?

41, 22, 44, 59, 32, 31, 74

Commentary: Double hashing avoids the problem of bunching up the keys, therefore improving search.

Problem: searchable by both keys and values

Exercise 5.12

Suppose you want to store a large set of key-value pairs, for example, (name,address). You have operations, which are addition, deletion, and search of elements in this set. You also have queries whether a particular name or address is there in the set, and if so then count them and delete all such entries. How would you design your hash tables?

Topic 5.8

Extra slides: Binary search in recursive representation!

Search for ordered keys

If keys are stored in order, then we use the binary search that we have discussed in lecture 1.

Algorithm 5.4: BinarySearch(A, key, low, high)

---

1 if $low > high$ **then**
2 | **return** *-1*

3 mid := (low+high)/2;
4 if $A[mid] == key$ **then**
5 | **return** *mid*

6 if $key < A[mid]$ **then**
7 | **return** *BinarySearch(A,key, low, mid-1)*

8 return *BinarySearch(A,key, mid+1, high)*

Exercise 5.13

We earlier saw the iterative version of the Binary search. Can we write any recursive algorithm as iterative algorithm?

Topic 5.9

Extra slides: Performance analysis for chaining.

Simple uniform fictional hash function

- An ideal hash function would pick a position uniformly at random and assign the key to it.
- However, this is not a real hash function, because we will not be able to search later.
- Only for our analysis, we will use this simple uniform hash function

Cost of unsuccessful search

► Simple uniform hashing will result in the average list length of α
► Number of elements traversed is α
► Search time is $O(1 + \alpha)$

Cost of successful search

▶ Assume that a new key-value pair is inserted at the end of the linked list

▶ Upon insertion of ith key-value pair the expected length of the list is $\dfrac{i-1}{m}$

▶ In the case of a successful search of the ith key, the expected number of keys examined is 1 more than the number of keys examined when the ith key-value pair was inserted.

▶ Expected number of key-value pairs examined for each key search

$$\frac{1}{n}\sum_{i=1}^{n}(1+\frac{i-1}{m}) = 1 + \frac{1}{mn}\sum_{i=1}^{n}(i-1) = 1 + \frac{1}{mn}\frac{n(n-1)}{2} = 1 + \frac{n}{2m} - \frac{1}{2m}$$

▶ Including the time for computing the hash function we obtain

$$2 + \frac{n}{2m} - \frac{1}{2m} \in \Theta(1+\alpha)$$

End of Lecture 5