香港中文大學(深圳)
The Chinese University of Hong Kong

CSC3100 Data Structures
Lecture 18: Hashing

Li Jiang
School of Data Science (SDS)
The Chinese University of Hong Kong, Shenzhen

Search problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key x, find the index i such as $x = A[i]$

- A record is often represented by a key-value pair
  - Example of key-value pair:
    - Key: student ID
    - Value: other information like name, major, age, sex…

example of a record

Key	other data

Applications

- Keeping track of customer account information at a bank
  - Query customer account by name, ID, account number, etc.

- Keep track of reservations on flights
  - Cancel/modify reservations

- Search engine
  - Looks for all documents containing a given word

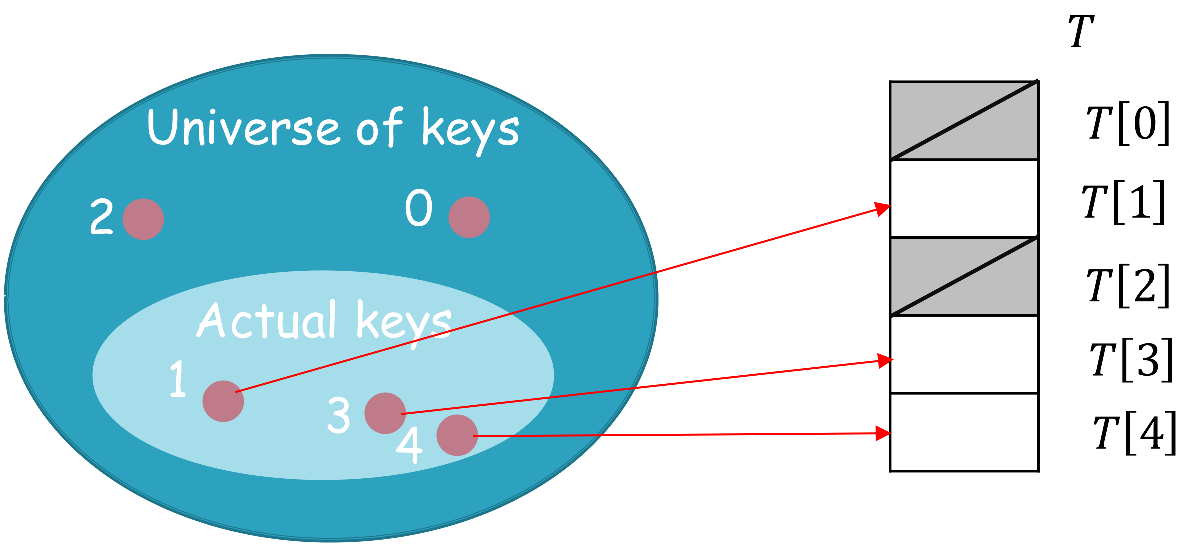
- Applications need a lot of queries
  - Once the data are inserted, deletion operations are not very often

First solution: direct addressing

- Assumptions:
  - Key values are distinct
  - Each key is drawn from a universe $U = \{0, 1, . . . , m - 1\}$
- Idea:
  - Store the items in an array, indexed by keys
- Direct-address table representation:
  - An array $T[0 . . . m - 1]$
  - Each corresponds to a key in U
  - T[k] stores a pointer to x (or x itself) with key k
  - T[k] may be empty

Direct-address tables

- If the keys of the records are integers from $[U] = \{0,1,2,\cdots,U-1\}$
  - We maintain an array T of size U
  - To insert a record r: $T[r.key] = r$
  - To delete a record r: $T[r.key] = NULL$
  - To search the record with key k: return $T[k]$



Limitation of direct-address table

- The universe of the keys are usually very large
  - All the integers in the range $[0, 2^{31} - 1]$, i.e., $U = 2^{31}$
  - But the number of records may be far less than U

- Example
  - Let $U = 2^{31} - 1 \approx 2.1\ billion$
  - Let the number of distinct keys be only 1 million
  - We need to create a direct-address table of size 2.1 billion when there are only 1 million records, so 99% of the space is wasted!

Comparing different solutions

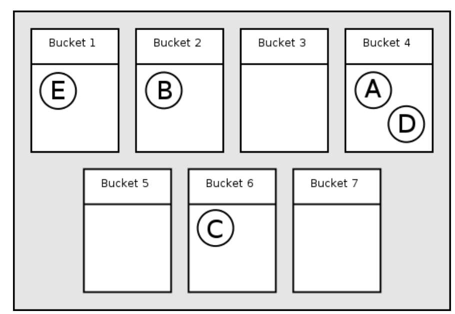
- Solving search using:
  - Direct addressing
  - Ordered/unordered arrays
  - Ordered/unordered linked lists
  - Binary search tree

	Insert	Search
direct addressing (keys are the indexes)	O(1)	O(1)
ordered array (keys are not indexes)	O(N)	O(logN)
ordered linked list	O(N)	O(N)
unordered array (keys are not indexes)	O(N)	O(N)
unordered linked list	O(1)	O(N)
binary search tree	O(logN)	O(logN)

Hashing: the main idea

- In direct-addressing, the record with key k is stored in slot k of the array T, i.e., $T[k]$

- In hashing, the element is stored in slot $h(k)$, i.e., $T[h(k)]$, where h is a hash function
  - Assume keys are integers in the range of $[0, U-1]$
  - Denote by $[x]$ the set of integers from 0 to $x-1$
  - A hash function h is a function from $[U]$ to $[m]$
    - For any integer k, $h(k)$ returns an integer in $[m]$
    - The value $h(k)$ is called the hash value of k
    - $U > m$



Simple hash functions

- For numeric keys, one simple hash function is

  Key mod *TableSize*,

  where *TableSize* is a prime number

E.g., select *TableSize* to be 4999, a prime number close to 5000

Key Value	Address		Key Value	Address	
123456789	1485		987654118	1688	**
123456790	1486		555555555	1688	**
000000504	0504	*	101129183	4412	
200120472	0504	*	200120473	0505	
118920912	4700		010600010	2130	
200120000	0032		027001191	1592	

* and ** indicate collisions

Example of hash functions

- If we have a set S of keys $\{1,2,3,5,7,8\}$, and the hash function is $h(x) = x\%3$ (module)
  - What are the hash values of the integers in S?
  - $h(1) = 1\ \%3 = 1$
  - $\boxed{h(2) = 2\%\ 3 = 2}$
  - $h(3) = 3\ \%\ 3 = 0$
  - $\boxed{h(5) = 5\ \%\ 3 = 2}$
  - $h(7) = 7\%\ 3\ = 1$
  - $\boxed{h(8) = 8\ \%\ 3 = 2}$
  - What are U and m?

Collision

Hash table ADT

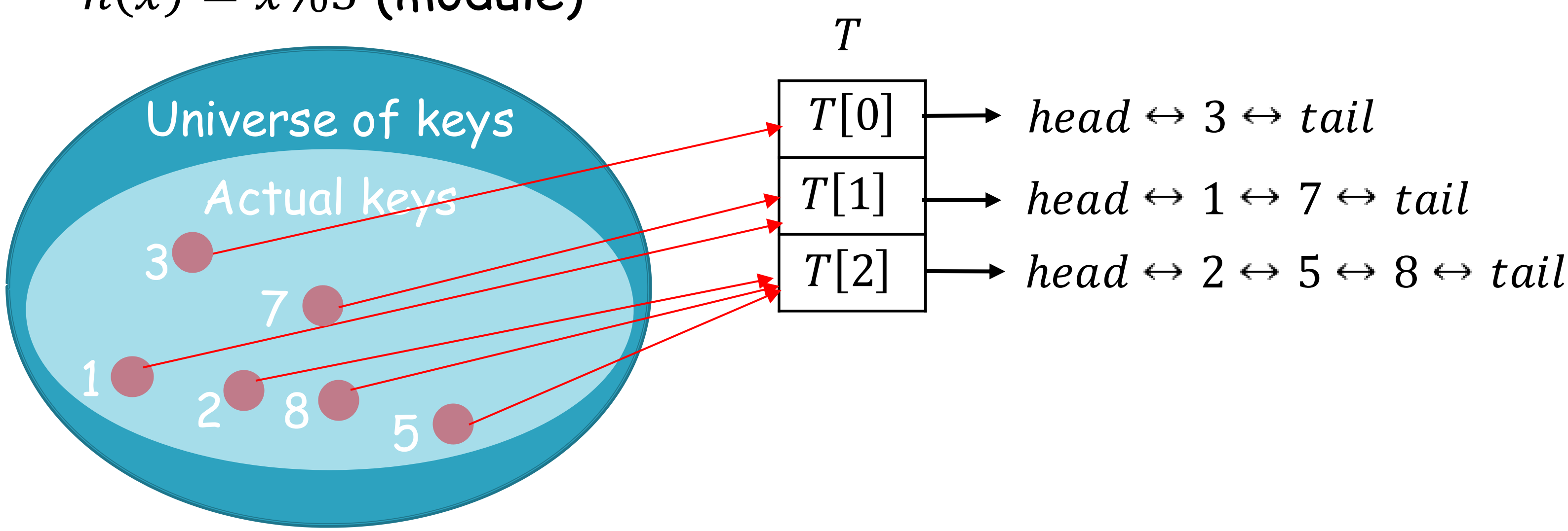
- Operations:
  - createTable(sizem): create a hash table of size sizem
  - search(hashtable, key): return the value if the hashtable contains the key, otherwise return NULL
  - insert(hashtable, key, value): insert the key value pair to the hashtable
  - delete(hashtable, key): remove the key and values stored on the hashtable
  - isFull(hashtable): return true if no element can be inserted to hashtable, otherwise return false

Collision resolution 1: chaining

- Collision: two keys hash to the same slot

- Chaining: we place all elements that hash to the same slot into the same linked list

$h(x) = x\%3$ (module)



Hash table initialization with chaining

- Assume $h(x)$ hashes keys to the range $[0, m-1]$
  - Create an array of size m with each entry storing a linked list

Algorithm: createTable(sizem)

```
hashtable←allocate an array of size sizem
for i from 0 to sizem-1
    hashtable[i] <- allocate an empty linkedlist
return hashtable
```

- search(hashtable, key): search record with key key:
  - Retrieve linked list $L_{h(key)}$ and search from $L_{h(key)}$

Algorithm: search(hashtable, key)

```
hashid = h(key)
node=hashtable[hashid].head.next
while node != hashtable[hashid].tail
    if node.data.key == key
        return node.data.value
    node = node.next
return NULL
```

Collision resolution 1: chaining

- insert(hashtable, key, record): insert record r with key key:
  - L_i: the linked list containing elements hashes to i
  - Find the linked list $L_{h(key)}$
  - Insert r to the end of $L_{h(key)}$

Algorithm: insert(hashtable, key, record)

```
hashid = h(key)
keyValuePair← (key, record)
Insert_linklist(hashtable[hashid], keyvaluepair)
```

```
Or simply:
Insert_linklist(hashtable[hashid], key, record)
```

Collision resolution 1: chaining

- delete(hashtable, key): delete record with key key
  - Retrieve linked list $L_{h(key)}$
  - Search from $L_{h(key)}$ and get the node containing key key, and delete this node

Algorithm: delete(hashtable, key)

```
hashid = h(key)
node=hashtable[hashid].head.next
while node != hashtable[hashid].tail
    if node.data.key == key
        break
    node = node.next
if node != hashtable[hashid].tail
  delete_linkedlist(hashtable[hashid], node)
```

- isFull(hashtable): always return false

Practice

- Given a hash function $h(k) = k\%7$, show the hash table after inserting 1, 3, 9, 20, 30, 51, 25, 23, 36
  - T[0]:
  - T[1]: 1, 36
  - T[2]: 9, 30, 51, 23
  - T[3]: 3
  - T[4]: 25
  - T[5]:
  - T[6]: 20

Analysis of hashing with chaining

- Load factor α: the average number of elements stored in a chain
  - If the hash table has size m and stores n elements in it, then $\alpha = \frac{n}{m}$
  - Assumption: uniform hashing of $h(k)$
    - Elements are equally likely to hash into any of the m slots
    - L_i: the linked list containing the elements hashes to i

> Theorem 1: In a hash table with collision resolved by chaining, the search/insertion/deletion takes $O(1+\alpha)$ time in expectation if $h(k)$ is uniform hashing.

- Proof: check appendix in the slides
- By choosing m so that $\frac{n}{m} = \alpha = O(1)$, the query time is $O(1)$

All theorems and proofs of hashing will not be tested in the exam

Collision resolution 2: open addressing

- All elements are stored in the hash table
  - Unlike chaining, no elements are stored outside the table
  - The hash table may "fill up" such no insertion can be made
    - Load factor α is always smaller than 1
- For insertion, we examine, or probe, the hash table until an empty slot is found to put the key
  - How to probe the hash table?
    - Linear probing, double Hashing
    - Quadratic probing (not discussed in the lecture)
- Deletion: not efficiently supported (why?)

Linear probing

- For insertion, we probe $h(k)$, $h(k)+1$, $h(k)+2, \cdots$, $h(k)+(m-1)$ one by one until we find an empty slot, and insert the record to this slot
  - Formally we probe $h(k,i)=(h(k)+i)\%m$ from $i=0$ to $i=m-1$ until we find an empty slot to insert

- When searching for a record with a certain key,
  - Compute $h(k)$
  - Examine the hash table buckets in order $T[h(k,i)]$ for $0 \leq i \leq m-1$ until one of the following happens:
    - $T[h(k,i)]$ has the record with key equal to k
    - $T[h(k,i)]$ is empty, then no record contains key k in the hash table

A running example: linear probing

- If we have a hash table with size $m = 17$ and the hash function $h(k) = k \% 17$
  - Consider inserting the following records: 6, 12, 34, 29, 28, 11, 23, 7, 0, 33, 30, 45
  - $h(6,0) = 6\%17 = 6$ → empty, insert here
  - $h(12,0) = 12\%17 = 12$ → empty, insert here
  - $h(34,0) = 34\%17 = 0$ → empty, insert here
  - $h(29,0) = 29\%17 = 12$ → not empty
    Try inserting at $h(29,1)$, empty, insert here
  - …

0				4				8				12				16
34	0	45				6	23	7			28	12	29	11	30	33

Practice

- If we have a hash table with size $m = 17$ and the hash function $h(k) = k \% 17$
  - Assume that we use linear probing to address collisions
    - Given the following hash table, show the hash table after inserting 21 and 13
    - Given the following hash table, show the records examined when searching for 14

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
34	0	45				6	23	7			28	12	29	11	30	33

Double hashing

- Deficiency of linear probing
  - Long sequence of occupied slots, which degrades the query efficiency

- Double hashing
  - We have an additional hash function $h' > 0$
  - Insertion: we probe $h(k, i) = \big(h(k) + i \cdot h'(k)\big)\%m$ one by one for i from 0 to $m - 1$ until an empty slot is found
  - Search: we search $h(k, i)$ for i from 0 to $m - 1$ until one of the following happens:
    - $T[h(k, i)]$ has the record with key equal to k
    - $T[h(k, i)]$ is empty, then no record contains key k in the hash table

A running example: double hashing

- If we have a hash table with size $m = 17$ and the hash function $h(k) = k \ \% \ 17$ and $h'(k) = 1 + k\%5$
- Insert the following records: 6, 12, 34, 29, 28 using double hashing to resolve collisions
  - $h(6,0) = 6\%17 = 6$ → empty, insert here
  - $h(12,0) = 12\%17 = 12$ → empty, insert here
  - $h(34,0) = 34\%17 = 0$ → empty, insert here
  - $h(29,0) = 29\%17 = 12$ → not empty,

  try $h(29,1) = (12 + 1 + 29\%5)\%17 = 0$, not empty,

  try $h(29,2) = (12 + 2 \cdot (1 + 29\%5))\%17 = 5$, empty, insert here
  - $h(28,0) = 28\%17 = 11$ → empty, insert here

0				4				8				12				16
34					29	6					28	12				

Practice

- Consider a hash table with size $m = 17$ and the hash function $h(k) = k \,\%\, 17$ and $h'(k) = 1 + k\%5$
  - Assume that we use double hashing,
    - Given the following hash table, show the hash table after inserting 11 and 27
    - After inserting 11 and 27, show the records examined when searching for 23

0				4				8				12				16
34					29	6				27	28	12	11			

Analysis of open addressing

- Load factor α: If the hash table T has size m and we store n elements in the hash table
  - $\alpha = \frac{n}{m}$

- Assumption: uniform hashing of $h(k,i)$
  - The probing sequence $\langle h(k,0), h(k,1), h(k,2), \cdots, h(k,m-1)\rangle$ used to insert or search for each key k is equally likely to be any permutation of $\langle 0,1,2,\cdots,m-1\rangle$

> Theorem 2: In a hash table with collision resolved by open addressing, the search takes $O(\frac{1}{1-\alpha})$ time in expectation if $h(k,i)$ is uniform hashing.

  - Proof: check appendix at the end of the slides
  - If $\alpha = O(1)$, then the query time is $O(1)$

All theorems and proofs of hashing will not be tested in the exam

Chaining vs open addressing

- Pros of chaining
  - Less sensitive to hash functions and load factors (α can be larger than 1), while open addressing requires to avoid long probes, and its load factor $\alpha < 1$
  - Support deletion, while open addressing is difficult to support deletion
- Pros of open addressing
  - Usually much faster than chaining

What makes a good hash function?

- A good hash function satisfies the uniform hashing property:
  - Each key is equally likely to hash to any of the m slots, independent of where other keys will hash to

- We learn two hashing functions: division and universal hashing
  - Division is effective in practice without any theoretical guarantee on $O(1)$ query time

  - Universal hashing provides theoretical guarantees on $O(1)$ query time

Hash function: division

- $h(k) = k \bmod m = k \% m$
  - If $m = 10^p$, then $h(k)$ only uses the lowest-order p digits of the key value k
    - We cannot use all digits to generate hash keys, and we should not choose such an m
    - In a similar way, we should not choose $m = 2^p$
  - Option of m: choose a prime number not close to the power of 2 or 10
    - Example: $U = 2000$, we choose $m = 701$

Universal hashing

- Let $\mathcal{H}$ be a family of hash functions from $[U]$ to $[m]$
- $\mathcal{H}$ is called universal if the following condition holds:

> Let k_1, k_2 be two distinct integers from $[U]$. By picking a function $h \in \mathcal{H}$ uniformly at random, we guarantee that
> $$\Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$$

  - Then, we choose one from $\mathcal{H}$ uniformly at random and use it as the hash function h for all operations

- Theoretical guarantee with universal hashing
  - With a universal hash function h
    - Query time of chaining: $O(1 + \alpha)$ (Proof: check appendix)
    - Query time of open addressing: more complicated (Omit), see [1]

[1] http://www.cs.tau.ac.il/~zwick/Adv-Alg-2015/Linear-Probing.pdf

Designing a family of universal functions

- We construct a universal family $\mathcal{H}$ of hash function from $[U]$ to $[m]$
  - Pick a prime number p such that $p > U$
  - For every $a \in \{1,2,\cdots,p-1\}$, and every $b \in \{0,1,2,\cdots,p-1\}$
    - $h_{a,b}(k) = \big((a \cdot k + b)\ mod\ p\big) mod\ m$
  - This defines $p \cdot (p-1)$ functions, which constitutes $\mathcal{H}$
    - Proof of universal: Omitted. Interested readers may refer to textbook, Chapter 11.3.3, pages 265-268
  - We then randomly select one function from these $p \cdot (p-1)$ functions

Example of universal function

- If $U = 10$ and $m = 5$
  - First select a prime number $p = 11$ $(p > U)$
  - We then have $11 \cdot 10 = 110$ functions in this universal family $\mathcal{H}$
  - For $a \in \{1,2,\cdots,10\}$, *and* $b \in \{0,1,2,\cdots,10\}$, we have
    - $h_{a,b}(k) = \big((a \cdot k + b)\ mod\ 11\big)\ mod\ 5$

- Does such p always exists? $(p > U)$
  - We can always find a prime number between $[U, 2U]$ according to number theorem (the proof is out of the scope, you may refer to [2] if you are interested)

[2]. https://en.wikipedia.org/wiki/Proof_of_Bertrand%27s_postulate

Recommended reading

- Reading this week
  - Textbook Chapters 11.1-11.4

- Next week
  - Graphs, Textbook Chapters 22.1-22.3

Appendix: Proof of Theorem 1

What we have: the uniform hashing assumption.
For any two keys k_i and k_j, $\Pr\left[h(k_i) = h\left(k_j\right)\right] = \frac{1}{m}$.

If we have query key q, the query cost is: $1 + \left|L_{h(q)}\right|$.
Recap: $L_{h(q)}$ is the linked list containing elements hashes to $h(q)$.

Define a random variable X_i to be 1 if the i-th element of the stored n elements has the same hash value as $h(q)$.

$$\left|L_{h(q)}\right| = \sum_{i=1}^{n} X_i$$

$E\left[\left|L_{h(q)}\right|\right] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$ (By linearity of expectation)

$E[X_i] = 1 \cdot \Pr[h(q) = h(k_i)] + 0 \cdot \Pr[h(q) \neq h(k_i)] = \frac{1}{m}$
$\Rightarrow E\left[\left|L_{h(q)}\right|\right] = \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$. We prove search can be finished in $O(1 + \alpha)$ time. We can similarly prove for insertion and deletion.

Appendix: Proof of Theorem 2

Given a query q, denote X_q be the number of probes, i.e., $h(q,0)$, $h(q,1), \cdots, h(q, X_q - 1)$, are occupied while $h(q, X_q)$ is not occupied.

The expected search cost is then:

$$E[X_q] = 1 \cdot \Pr[X_q = 1] + 2 \cdot \Pr[X_q = 2] + \cdots + m \cdot \Pr[X_q = m]$$
$$= \sum_{i=1}^{m} i \cdot \Pr[X_q = i] = \sum_{i=1}^{m} i \cdot (\Pr[X_q \geq i] - \Pr[X_q \geq i+1])$$
$$= \sum_{i=1}^{m} \Pr[X_q \geq i]$$

The probability that $\Pr[X_q \geq i]$?

The 1st one is occupied by one of the n elements. Probability: $\frac{n}{m}$

The 2nd one is occupied by one of the remaining n-1 elements.

Probability: $\frac{n-1}{m-1}$,

…

The $(i-1)$-th one is occupied by one of the remaining $n-i+2$ elements: $\frac{n-i+2}{m-i+2}$.

Proof of Theorem 2

Therefore: $\Pr[X_q \geq i] = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2} \leq \left(\frac{n}{m}\right)^{i-1} = \alpha^{i-1}$

$$E[X_q] = \sum_{i=1}^{m} \Pr[X_q \geq i] \leq 1 + \alpha + \alpha^2 + \cdots \alpha^{m-1} \leq \frac{1}{1-\alpha}$$

Hence, the expected search cost can be bounded by $O(\frac{1}{1-\alpha})$.

Appendix: Query time with universal hashing

What we have: the universal hashing assumption.
For any two keys k_i and k_j, $\Pr\left[h(k_i) = h(k_j)\right] \leq \frac{1}{m}$.

If we have query key q, the query cost is: $1 + \left|L_{h(q)}\right|$.
Recap: $L_{h(q)}$ is the linked list containing elements hashes to $h(q)$.

Define a random variable X_i to be 1 if the i-th element of the stored n elements has the same hash value as $h(q)$.

$$\left|L_{h(q)}\right| = \sum_{i=1}^{n} X_i$$

$E\left[\left|L_{h(q)}\right|\right] = E\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} E[X_i]$ (By linearity of expectation)

$E[X_i] = 1 \cdot \Pr[h(q) = h(k_i)] + 0 \cdot \Pr[h(q) \neq h(k_i)] \leq \frac{1}{m}$
Therefore, $E\left[\left|L_{h(q)}\right|\right] \leq \sum_{i=1}^{n} \frac{1}{m} = \frac{n}{m} = \alpha$. Proof done.