ECE2810J

Data Structures and Algorithms

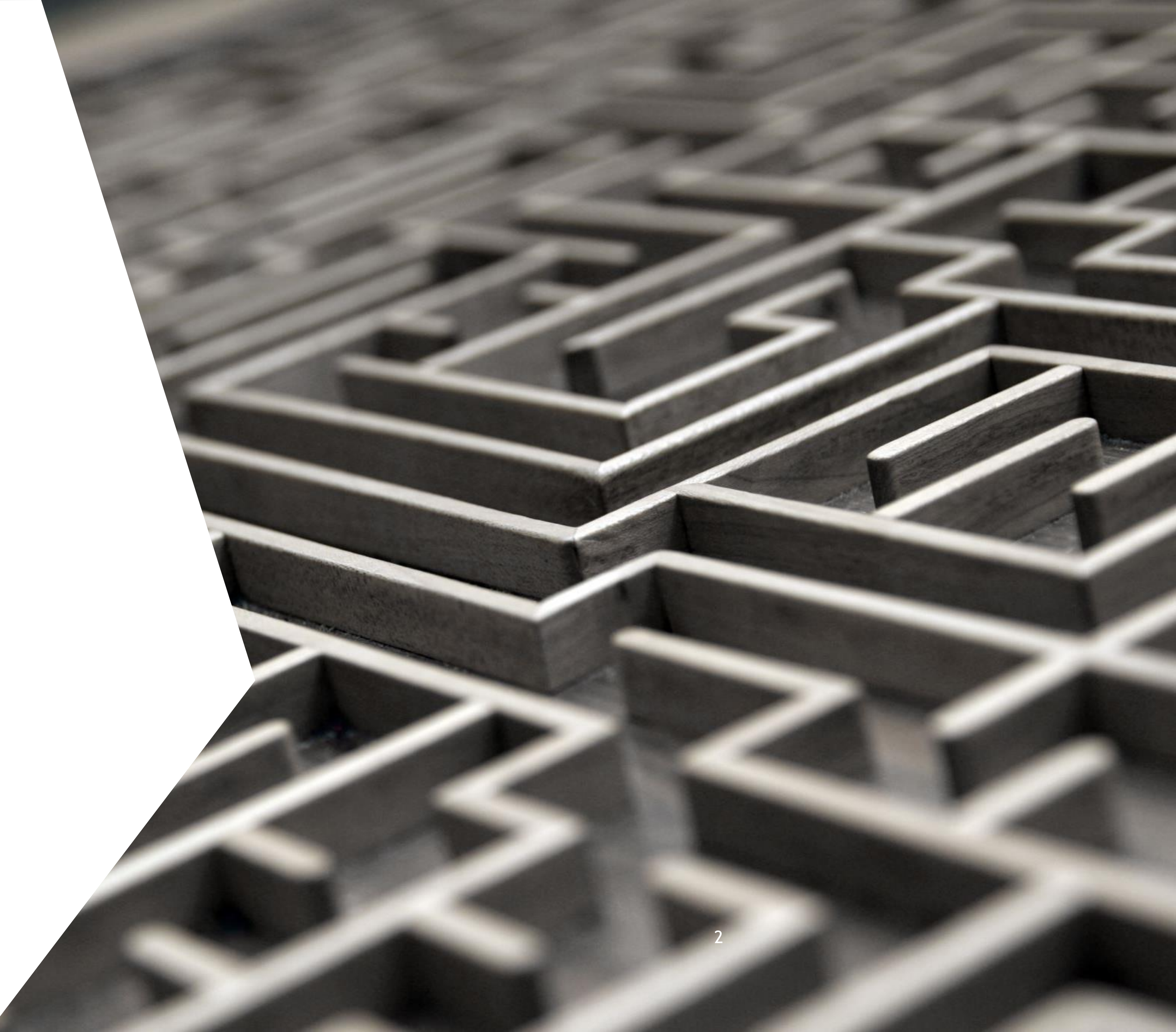
RC2

Topics:

- Non-comparison Sort
- Linear Time Selection
- Hashing Table

Outline

- Non-comparison Sort
  - Counting Sort
  - Bucket Sort
  - Radix Sort
- Linear Time Selection
  - Randomized selection algorithm
  - Deterministic selection algorithm
- Hashing Table
  - Hashing Basics
  - Hash Function
  - Collision Resolution



Counting Sort
A General Version

- A general version (allow additional data and guarantee the stability):

1. Allocate an array **`C[k+1]`**
2. Scan array **`A`**. For i=**`1`** to **`N`**, increment **`C[A[i]]`**
3. For **`i=1`** to **`k`**, **`C[i]=C[i-1]+C[i]`**
   - **`C[i]`** now contains number of items less than or equal to **`i`**
4. For **`i=N`** downto **`1`**, put **`A[i]`** in new position **`C[A[i]]`** and decrement **`C[A[i]]`**

Counting Sort
Example (General, allows additional data in A)

1. Allocate an array `C[k+1]`.
2. Scan array **A**. For **i=1** to **N**, increment `C[A[i]]`.
3. For `i=1` to `k`, `C[i]= C[i-1]+C[i]`
4. For `i=N` downto `1`, put `A[i]` in new position `C[A[i]]` and decrement `C[A[i]]`.

k=5

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	0	2	2	4	7	7

	1	2	3	4	5	6	7	8
A	0	0	2	2	3	3	3	5

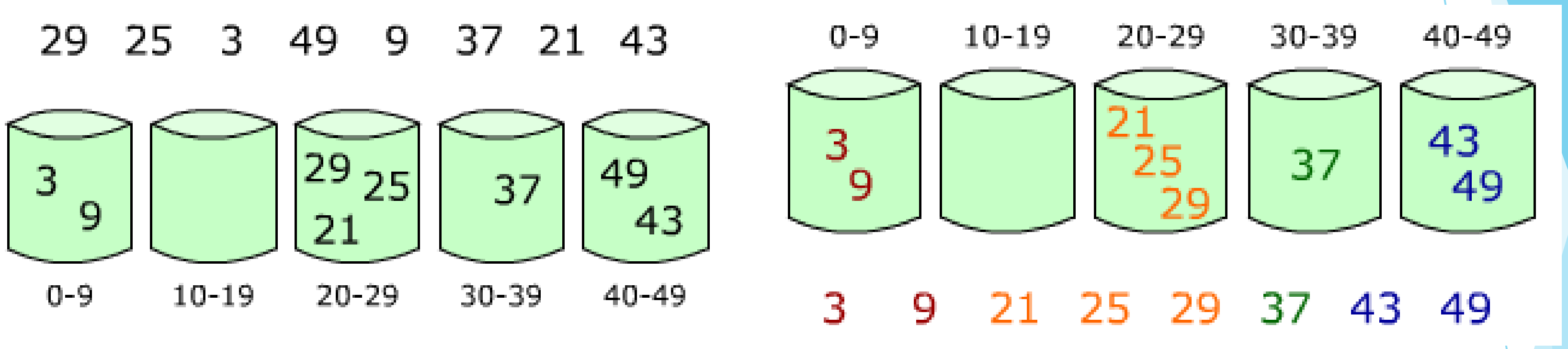
Bucket Sort

- Instead of simple integer, each key can be a complicated record, such as a real value.
- Then instead of incrementing the count of each bucket, **distribute** the records **by their keys** into appropriate buckets.
- Algorithm:

1. Set up an array of initially empty “buckets”.
2. Scatter: Go over the original array, putting each object in its bucket.
3. Sort each non-empty bucket <u>by a comparison sort</u>.
4. Gather: Visit the buckets in order and put all elements back into the original array.

Bucket Sort

- Example



- Time complexity
  - Suppose we are sorting cN items and we divide the entire range into N buckets.
  - Assume that the items are **uniformly distributed** in the entire range.
  - The average case time complexity is $O(N)$.

Radix Sort

- **Radix sort** sorts integers by looking at one digit at a time.
- Procedure: Given an array of integers, from the least significant bit (LSB) to the most significant bit (MSB), repeatedly do **stable** bucket sort according to the current bit.
- For sorting base-b numbers, bucket sort needs b buckets.
  - For example, for sorting decimal numbers, bucket sort needs 10 buckets.

Radix Sort
Example

- Sort 815, 906, 127, 913, 098, 632, 278.
- Bucket sort 815, 906, 127, 913, 098, 632, 278 according to the least significant bit:

0	1	2	3	4	5	6	7	8	9
		632	913		815	906	127	098 278	

- Bucket sort 632, 913, 815, 906, 127, 098, 278 according to the second bit.

Radix Sort
Example

- Bucket sort 632, 913, 815, 906, 127, 098, 278 according to the second bit.

0	1	2	3	4	5	6	7	8	9
906	913 815	127	632				278		098

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

Radix Sort
Example

- Bucket sort 906, 913, 815, 127, 632, 278, 098 according to the most significant bit.

0	1	2	3	4	5	6	7	8	9
098	127	278				632		815	906 913

- The final sorted order is: 098, 127, 278, 632, 815, 906, 913.

Radix Sort
Time Complexity

- Let k be the maximum number of digits in the keys and N be the number of keys.
- We need to repeat bucket sort k times.
  - Time complexity for the bucket sort is $O(N)$.
- The total time complexity is $O(kN)$.

Radix Sort

- Radix sort can be applied to sort keys that are built on **positional notation**.
  - **Positional notation**: all positions uses the same set of symbols, but different positions have different weight.
  - Decimal representation and binary representation are examples of positional notation.
  - Strings can also be viewed as a type of positional notation. Thus, radix sort can be used to sort strings.
- We can also apply radix sort to sort records that contain multiple keys.
  - For example, sort records (year, month, day).

Randomized Selection

```
Rselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Choose pivot p from A uniformly at random;
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Rselect(1st part of A, j-1, i);
  else return Rselect(2nd part of A, n-j, i-j);
}
```

Deterministic Selection Algorithm

```
Dselect(int A[], int n, int i) {
// find i-th smallest item of array A of size n
  if(n == 1) return A[1];
  Break A into groups of 5, sort each group;
  C = n/5 medians;
  p = Dselect(C, n/5, n/10);
  Partition A using pivot p;
  Let j be the index of p;
  if(j == i) return p;
  if(j > i) return Dselect(1st part of A, j-1, i);
  else return Dselect(2nd part of A, n-j, i-j);
}
```

Choose Pivot

Same as Rselect

The function has two recursive calls

Deterministic Selection Algorithm

- In deterministic selection, assume groups are made up of 9 elements instead of 5. Will there be more or less recursive calls to *DSelect* **within** the "finding the median of medians" steps?

- Fewer recursive calls.
- larger buckets -> less number of buckets

Hashing

“Algorithm” -> A -> … -> find it

An element -> hash function -> find it

(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

(3,c)	(33,b)	(22,a)	(85,e)	(73,d)			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

What Can Go Wrong?

(3,c)		(22,a)	(33,b)			(73,d)	(85,e)
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]

- Where does (35, g) go?
- Problem: The home bucket for (35, g) is already occupied!
  - This is a "**collision**".

Hash Function Design Criteria

- Must compute a bucket for every key in the universe.
- Must compute the same bucket for the same key.
- Should be easy and quick to compute.
- Minimizes collision ← The hardest criterion
  - Spread keys out evenly in hash table
  - **Gold standard**: **completely random hashing**
    - The probability that a randomly selected key has bucket i as its home bucket is $1/n,\ 0 \leq i < n$.
    - Completely random hashing **minimizes** the likelihood of a collision when keys are selected at random.
    - However, completely random hashing is **infeasible** due to the need to remember the random bucket.

Hash Functions

- Hash function ($h(key)$) maps key to buckets in two steps:

1. Convert key into an integer in case the key is not an integer.
   - A function $t(key)$ which returns an integer value, known as **hash code**.
2. **Compression map**: Map an integer (hash code) into a home bucket.
   - A function $c(hashcode)$ which gives an integer in the range $[0, n-1]$, where n is the number of buckets in the table.

- In summary, $h(key) = c(t(key))$, which gives an index in the table.

Hash function criteria: Should be easy and quick to compute.

Compression Map

- Map an integer (hash code) into a home bucket.
- The most common method is by **modulo arithmetic**.

  `homeBucket = c(hashcode) = hashcode % n`
  where n is the **number of buckets** in the hash table.

- Example: Pairs are (22,a), (33,b), (3,c), (55,d), (79,e). Hash table size is 7.

	(22,a)	(79,e)	(3,c)		(33,b)	(55,d)
[0]	[1]	[2]	[3]	[4]	[5]	[6]

Hashing by Modulo

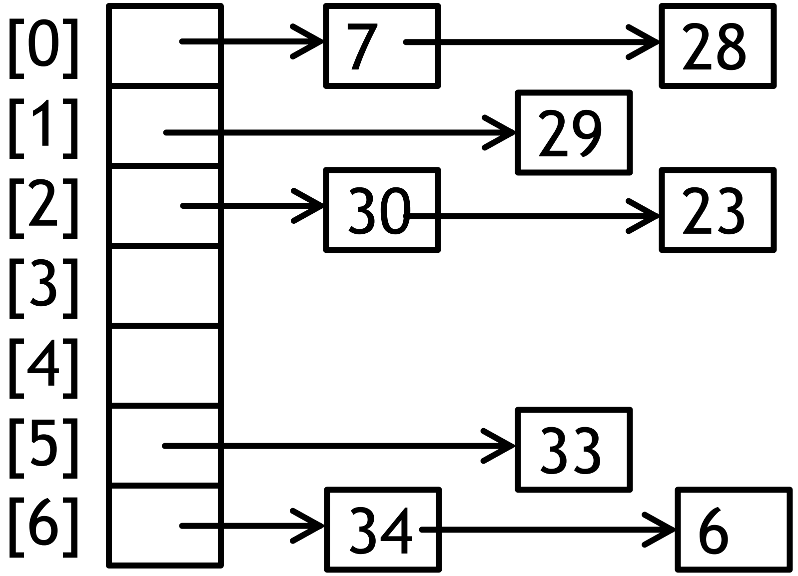
- The choice of the hash table size n will affect the distribution of home buckets.
- Suppose the keys of an application are more likely to be mapped into even integers.
  - E.g., memory address is always a multiple of 4.
- When the hash table size n is an **even** number, **even** integers are hashed into **even** home buckets.
  - E.g., n = 14: 20%14 = 6, 32%14 = 4, 8%14 = 8
- So **do not** use an even hash table size n.
- Ideally, choose the hash table size n as a **large prime number**.

Collision Resolution

- Separate Chaining
- Open Addressing
  - Linear Probing
  - Quadratic Probing and Double Hashing
  - Performance of Open Addressing

Separate Chaining

- Each bucket keeps a **linked list** of all items whose home buckets are that bucket.
- Example: Put pairs whose keys are 6, 23, 34, 28, 29, 7, 33, 30 into a hash table with $n = 7$ buckets.
  - `homeBucket = key % 7`
  - **Note**: we insert object at the beginning of a linked list.



Separate Chaining

- **`Value find(Key key)`**
  - Compute **`k = h(key)`**
  - Search in the linked list located at the k-th bucket with the key
    - Check every entry
- **`void insert(Key key, Value value)`**
  - Compute **`k = h(key)`**
  - Search in the linked list located at the k-th bucket
    - If found, update its value;
    - Otherwise, insert pair at the beginning of the linked list in O(1) time

Separate Chaining

- `Value remove(Key key)`
  - Compute `k = h(key)`
  - Search in the linked list located at the k-th bucket
    - If found, remove that pair

Open Addressing

- Reuse empty space in the hash table to hold colliding items.

- Search hash table in systematic way for an empty bucket
  - Idea: use a sequence of hash functions h_0, h_1, h_2, . . . to **probe** the hash table until we find an empty slot.
    - I.e., we **probe** the hash table buckets mapped by h_0(key), h_1(key), …, in sequence, until we find an empty slot.
    - Generally, we could define $h_i(x) = h(x) + f(i)$

Open Addressing Methods

- Linear probing:
  $h_i(x) = (h(x) + i) \% n$
- Quadratic probing:
  $h_i(x) = (h(x) + i^2) \% n$
- Double hashing:
  $h_i(x) = (h(x) + i*g(x)) \% n$

n is the hash table size

Linear Probing

$h_i(key) = (h(key)+i) \% n$

- Apply hash function h_0, h_1, …, in sequence until we find an empty slot.
  - This is equivalent to doing a linear search from **h(key)** until we find an empty slot.

- Example: Hash table size $n = 9$, $h(key) = key\%9$
  - Thus $h_i(key) = (key\%9+i)\%9$
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence

	1	11			5			
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

How about 2?

Linear Probing

Example

- Hash table size `n = 9`, `h(key) = key%9`
  - Thus h_i`(key) = (key%9+i)%9`
  - Suppose we insert 1, 5, 11, 2, 17, 21, 31 in sequence.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	1	11	2	21	5	31		17

- $h_0(2) = 2$. Not empty!
- So we try $h_1(2) = 3$. It is empty, so we insert there!
- $h_0(21) = 3$. Not empty!
- $h_1(21) = 4$. It is empty, so we insert there!
- $h_0(31) = 4$. Not empty!
- $h_1(31) = 5$. Not empty!
- $h_2(31) = 6$. It is empty, so we insert there!

Linear Probing

find()

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- With linear probing `h_i(key) = (key%9+i)%9`
  - How will you **search** an item with key = 31?
  - How will you **search** an item with key = 10?

- Procedure: probe in the buckets given by h_0(key), h_1(key), …, in sequence **until**
  - we find the key,
  - or we find an empty slot, which means the key is not found.

Linear Probing

remove()

1	11	2	21	5	31		17

	1	11	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- With linear probing `h`$_i$ `(key) = (key%9+i)%9`
  - How will you **remove** an item with key = 11?
  - If we just find 11 and delete it, will this work?

	1		2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

What is the result for searching key = 2 with the above hash table?

Linear Probing

remove()

cluster

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	1		2	21	5	31		17

- After deleting 11, we need to **rehash** the following “cluster” to fill the vacated bucket.
- However, we cannot move an item **beyond** its **actual** hash position. In this example, 5 cannot be moved ahead.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	1		2	21	5	31		17

Linear Probing

Alternative implementation of remove()

	1	del	2	21	5	31		17
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]

- **Lazy deletion**: we mark deleted entry as “**deleted**”
  - “deleted” is not the same as “empty”
  - Now each bucket has three states:
    - “occupied”, “empty”, and “deleted”
- We can overwrite the “deleted” entry when inserting
- When we **search**, we will keep looking if we encounter a “deleted” entry

Quadratic Probing

$$h_i(key) = (h(key)+i^2) \% n$$

- It is less likely to form large clusters.
- Example: Hash table size $n = 7$, $h(key) = key\%7$
  - Thus $h_i(key) = (key\%7+i^2)\%7$
  - Suppose we insert 9, 16, 11, 2 in sequence.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
		9	16	11		2

  - $h_0(16) = 2$. Not empty!
  - $h_1(16) = 3$. It is empty, so we insert there.
  - $h_0(2) = 2$. Not empty!
  - $h_1(2) = 3$. Not empty!
  - $h_2(2) = 6$. It is empty, so we insert there.

Problem of Quadratic Probing

- However, may never find an empty slot even if the table isn't full!
  - Highly filled table
- Luckily, if the **load factor** $L \leq 0.5$, guaranteed to find an empty slot
  - Table size must be a **prime** number!
  - Definition: given a hash table with n buckets that stores m objects, its **load factor** is

$$L = \frac{m}{n} = \frac{\#\text{objects in hash table}}{\#\text{buckets in hash table}}$$

More on Load Factor of Hash Table

- Question: which collision resolution strategy is feasible for load factor larger than 1?
  - Answer: separate chaining.
  - Note: for open addressing, we require $L \leq 1$.
- Claim: $L = O(1)$ is a necessary condition for operations to run in constant time.

More on Load Factor of Hash Table

- Question: A hash table of size 100 has 40 empty elements and 25 deleted elements. What is its load factor?

- Answer: 0.35
- $L = \frac{100-40-25}{100} = \frac{35}{100} = 0.35$

Double Hashing

$$h_i(x) = (h(x) + i*g(x)) \% n$$

- Uses 2 distinct hash functions.
- Increment **differently** depending on the key.
  - If h(x) = 13, g(x) = 17, the probe sequence is 13, 30, 47, 64, …
  - If h(x) = 19, g(x) = 7, the probe sequence is 19, 26, 33, 40, …
  - For linear and quadratic probing, the incremental probing patterns are **the same** for all the keys.

Double Hashing
Example

- Hash table size `n = 7, h(key) = key%7, g(key) = (5-key)%5`
  - Thus `hᵢ(key) = (key%7+(5-key)%5*i)%7`
  - Suppose we insert 9, 16, 11, 2 in sequence.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
		9		11	2	16

  - $h_0(16) = 2$. Not empty!
  - $h_1(16) = 6$. It is empty, so we insert there.
  - $h_0(2) = 2$. Not empty!
  - $h_1(2) = 5$. It is empty, so we insert there.

Expected Number of Comparisons

- Chaining (assume completely random hash)
  - First check whether empty or not, count as 1 operation
  - Average length is L in each bucket
    - U(L) = 1+L
    - S(L) = 1+L/2 (average search of filled bucket is half expected length)

Which Strategy to Use?

- Both separate chaining and open addressing are used in real applications
- Some basic guidelines:
  - If resizing is frequent, better to use open addressing
  - If need removing items, better to use separate chaining
    - **`remove()`** is tricky in open addressing
  - In mission critical application, prototype both and compare

Exercises

- Suppose you have a hash table of size $M = 7$ that uses the hash function $H(n) = n$ and the compression function $C(n) = n \bmod M$. Quadratic probing is used to resolve collisions. You enter the following six elements into this hash table in the following order: *{*24*,* 11*,* 17*,* 21*,* 10*,* 4*}*. No resizing is done. After all collisions are resolved, which index of the hash table remains empty (the first index is 0)?

- Answer: 2

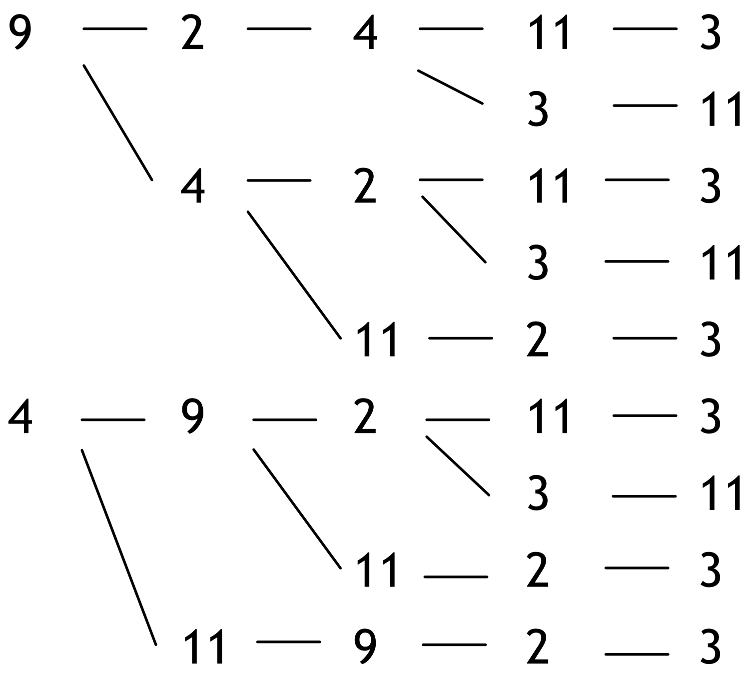
17	21		24	11	10	4
0	1	2	3	4	5	6

Exercises

- How many possbile inserting sequences for the hash table using quadratic probing with hash function $h_i(x) = (x + i^2) \bmod 7$ would lead to a hash table like this?

0	1	2	3	4	5	6
3		9	2	4	11	

- Answer: 9



Exercises

- https://leetcode.cn/problems/two-sum/
- https://leetcode.cn/problems/longest-consecutive-sequence/