Data and File Structures Laboratory

Hashing

Malay Bhattacharyya

Assistant Professor

Machine Intelligence Unit
Indian Statistical Institute, Kolkata

September, 2018

1 Basics

2 Hash Collision

3 Hashing in other applications

What is hashing?

Definition (Hashing)

Hashing is the process of indexing and retrieving data items in a data structure to provide faster way (preferably $O(1)$) of finding the element using the hash function.

Hash function

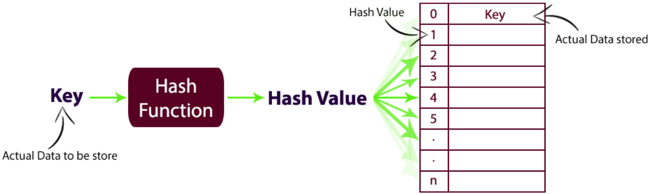
Definition (Hash function)

A hash function h projects a value from a set with many (or even an infinite number of) data items to a value from a set with a fixed number of (fewer) data elements.

Hash function

Definition (Hash function)

A hash function *h* projects a value from a set with many (or even an infinite number of) data items to a value from a set with a fixed number of (fewer) data elements.



Note: The hashed values are kept in a data structure known as *hash tables*.

Hash function

Some popular examples of hash function are given below.

- $k^* = h(M) = k\%10$,
- $k^* = h(M) = k/2$,
- $k^* = h(M) = \lceil \log k \rceil$,
- etc.

Hash function

Some popular examples of hash function are given below.

- $k^* = h(M) = k\%10$,
- $k^* = h(M) = k/2$,
- $k^* = h(M) = \lceil \log k \rceil$,
- etc.

Definition (Preimage)

For a hash value $k^* = h(k)$, k is defined as the preimage of k^*.

What makes a good hash function?

We want to design a hash function $h : [n] \rightarrow [m]$ $(n > m)$ that satisfies the following requirements:

- Searching (lookup) is worst-case O(1).
- Deletions are worst-case O(1).
- Insertions are amortized, expected O(1).
- Each data item is equally likely to hash to any of the m positions
- The function h is computationally collision free.

What makes a good hash function?

We want to design a hash function $h : [n] \rightarrow [m]$ $(n > m)$ that satisfies the following requirements:

- Searching (lookup) is worst-case O(1).
- Deletions are worst-case O(1).
- Insertions are amortized, expected O(1).
- Each data item is equally likely to hash to any of the m positions
- The function h is computationally collision free.

Note: Depending on the application, there might be additional requirements.

Family of hash functions

Definition (k-independent family of hash functions)

A family of hash functions is said to be k-independent if selecting a function at random from the family guarantees that the hashed values of any designated k keys are independent random variables.

Family of hash functions

Definition (k-independent family of hash functions)

A family of hash functions is said to be k-independent if selecting a function at random from the family guarantees that the hashed values of any designated k keys are independent random variables.

Note: Some efficient approaches are required to satisfy k-independence property.

What is hash collision?

If multiple data items (keys) hash to the same position then it is termed as a hash collision.

Formally, a pair of keys k_1 and k_2 are said to have hash collision with respect to the hash function $h()$ if

$$h(k_1) = h(k_2).$$

For example, the keys 121 and 1234321 will have hash collision with respect to the hash function $h(k) = k\%11$.

Dealing with hash collision

- **Strategy 1:** Resolution
  - Closed addressing: Store all the elements with hash collisions in an auxiliary data structure (e.g., linked list, BST, etc.) outside the hash table.
  - Open addressing: Store all the elements with hash collisions by strategically moving them from preferred to the other positions in the hash table itself.

Dealing with hash collision

- **Strategy 1:** Resolution
  - Closed addressing: Store all the elements with hash collisions in an auxiliary data structure (e.g., linked list, BST, etc.) outside the hash table.
  - Open addressing: Store all the elements with hash collisions by strategically moving them from preferred to the other positions in the hash table itself.
- **Strategy 2:** Avoidance
  - Perfect hashing: Ensure that collisions do not happen and if happen relocate the other elements.

Dealing with hash collision

- **Strategy 1:** Resolution
  - Closed addressing: Store all the elements with hash collisions in an auxiliary data structure (e.g., linked list, BST, etc.) outside the hash table.
  - Open addressing: Store all the elements with hash collisions by strategically moving them from preferred to the other positions in the hash table itself.
- **Strategy 2:** Avoidance
  - Perfect hashing: Ensure that collisions do not happen and if happen relocate the other elements.

Note: Closed addressing is also termed as *chaining*.

Closed addressing

Closed addressing uses an auxiliary data structure whose domain D is defined as follows

$$D := \{k_i | \exists k_j \in H : k_i \neq k_j, h(k_i) = h(k_j)\}$$

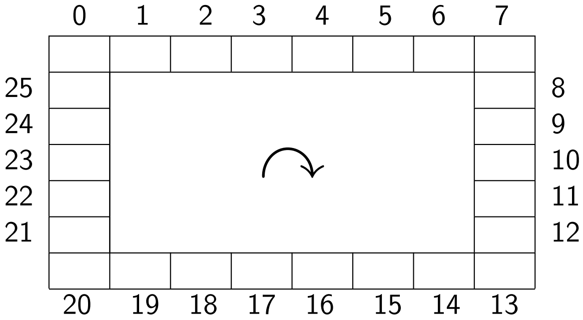
Here, H denotes the hash table.

Note: Use of auxiliary data structures include additional burdens (of pointer dereferencing) that are not cache-friendly,

Closed addressing – Insertion

Let $h(k) = k\%26$.

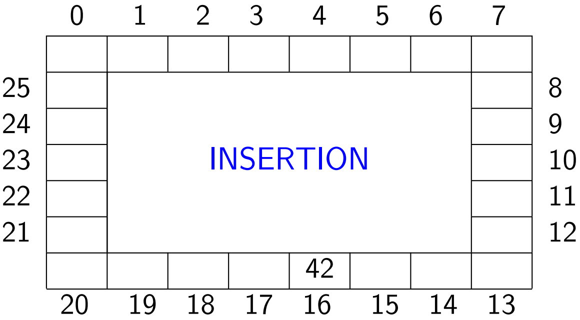
k
42 $\rightarrow$
94
11
16
68
37



Closed addressing – Insertion

Let $h(k) = k\%26$.

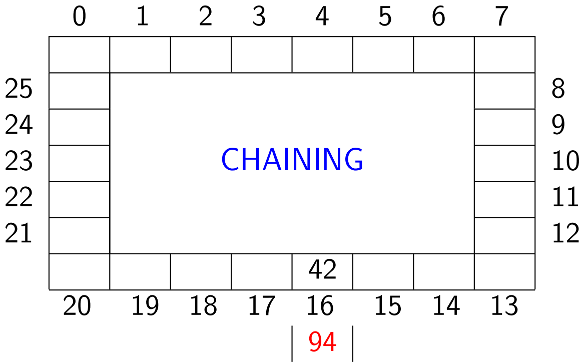
k
42
94 →
11
16
68
37



Closed addressing – Insertion

Let $h(k) = k\%26$.

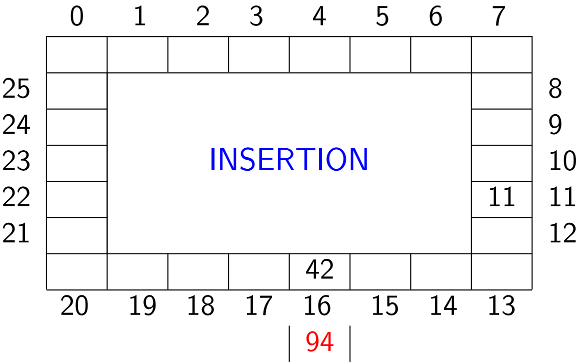
k
42
94
11 →
16
68
37



Closed addressing – Insertion

Let $h(k) = k\%26$.

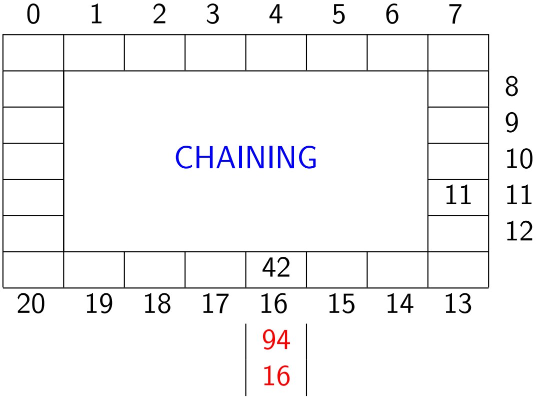
k
42
94
11
16 →
68
37



Closed addressing – Insertion

Let $h(k) = k\%26$.

k
42
94
11
16
68 $\rightarrow$
37

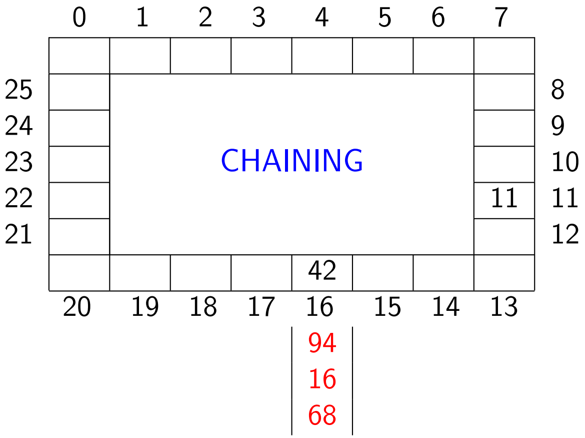


CHAINING

Closed addressing – Insertion

Let $h(k) = k\%26$.

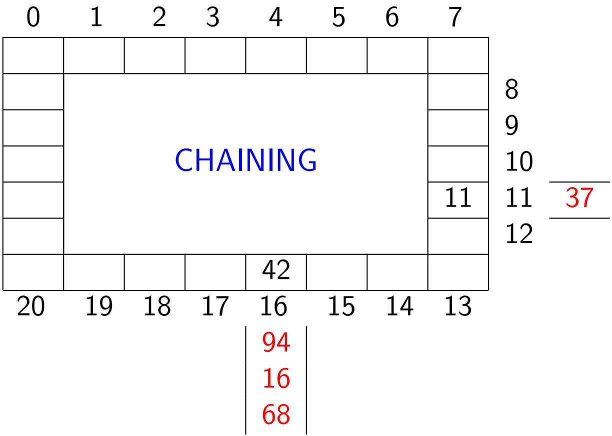
k
42
94
11
16
68
37 →



Closed addressing – Insertion

Let $h(k) = k\%26$.

k
42
94
11
16
68
37



0 1 2 3 4 5 6 7

8 9 10 11 12

13 14 15 16 17 18 19 20

21 22 23 24 25

CHAINING

11

37

42

94

16

68

Closed addressing – Implementation

Traditional:

```
typedef struct node{
    unsigned hash_val;
    DATA data;
    struct node *next;
}HNODE;
```

Closed addressing – Implementation

Traditional:

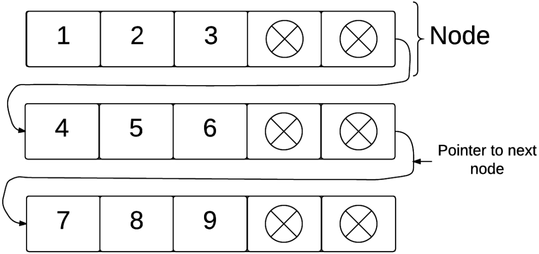
```
typedef struct node{
    unsigned hash_val;
    DATA data;
    struct node *next;
}HNODE;
```

Alternative: Using unrolled linked lists!!!

```
#define HASH_UNROLL 10
typedef struct node{
    unsigned hash_val[HASH_UNROLL];
    DATA data[HASH_UNROLL]; // Array of elements at a node
    struct node *next;
}HNODE;
```

Closed addressing – Implementation

Unrolled linked list: This is a variant of linked list containing nodes of small arrays (of same size), which are large enough to fill the cache line. An iterator pointing into the list comprises both a pointer to a node and an index into that node containing an array.



Note: Unrolled linked lists are conceptually related to B-trees.

Open addressing with linear probing

Linear probing uses a hash function of the form

$$h(k,i) = (h'(k) + i)\%m.$$

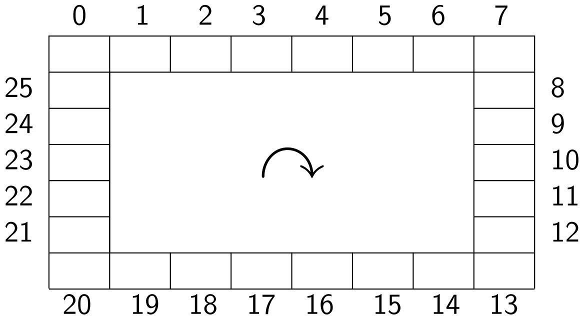
Here, h' is an auxiliary hash function and $i = 0, 1, \ldots, m-1$.

Note: The number of collisions tends to grow as a function of the number of existing collisions. This problem is known as *primary clustering*. It increases the average search time in a hash table.

Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

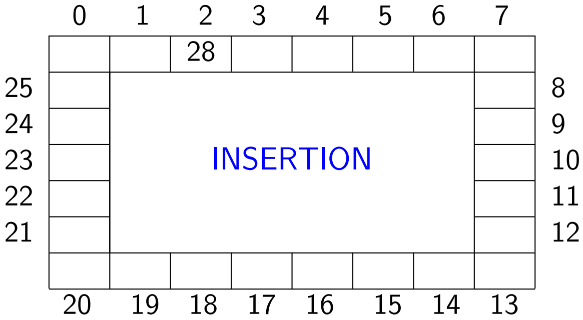
k
28 →
61
99
35
9
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

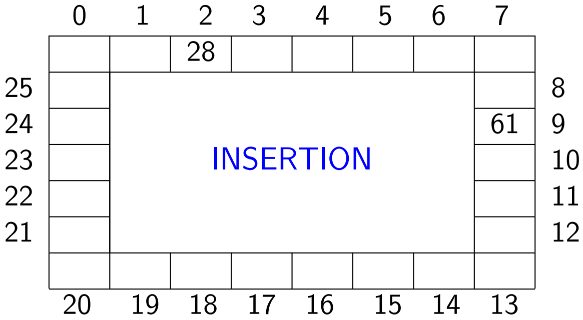
k
28
61 →
99
35
9
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

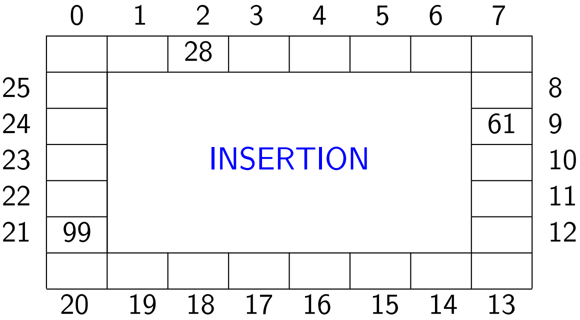
k
28
61
99 →
35
9
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

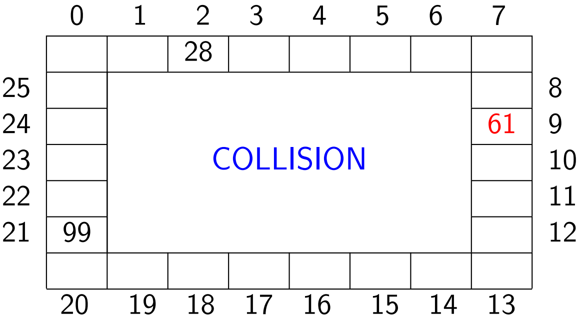
k
28
61
99
35 →
9
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

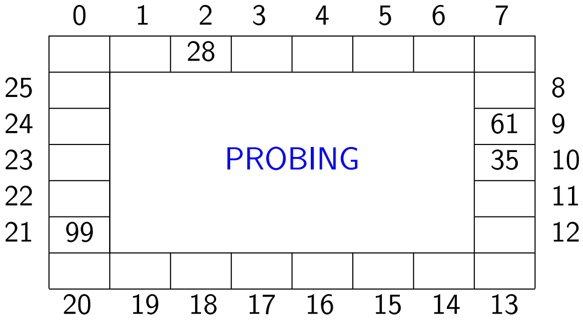
k
28
61
99
35 →
9
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

k
28
61
99
35
9 →
55



0 1 2 3 4 5 6 7

28

25 8

24 61 9

23 PROBING 35 10

22 11

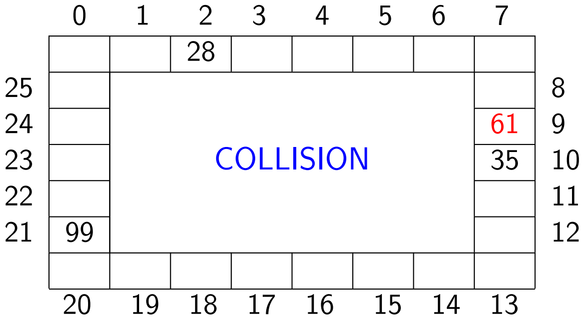
21 99 12

20 19 18 17 16 15 14 13

Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

k
28
61
99
35
9 →
55



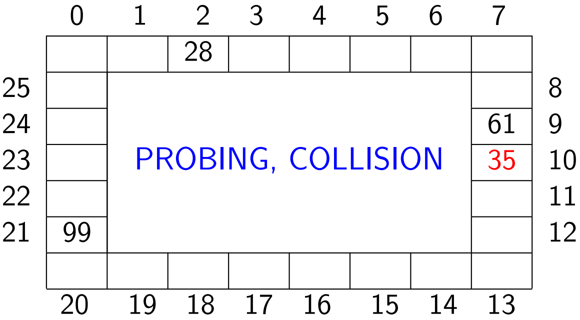
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
		28							61	35											99				

COLLISION

Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

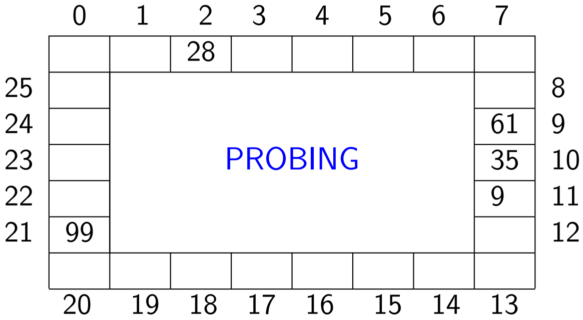
k
28
61
99
35
9 →
55



Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

k
28
61
99
35
9
55 →



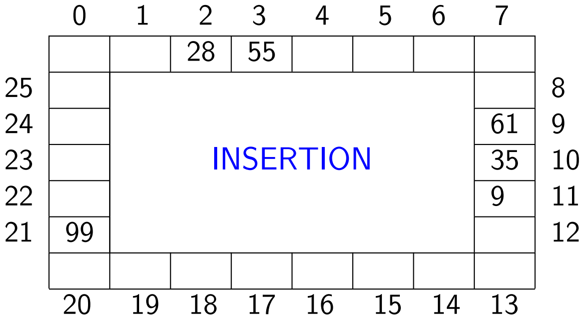
PROBING

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
		28							61	35	9										99				

Open addressing with linear probing – Insertion

Let $h'(k) = k\%26$.

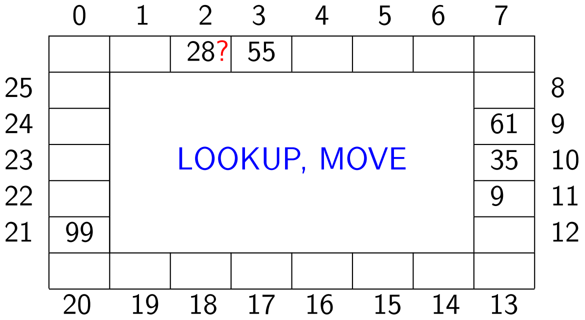
k
28
61
99
35
9
55



Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

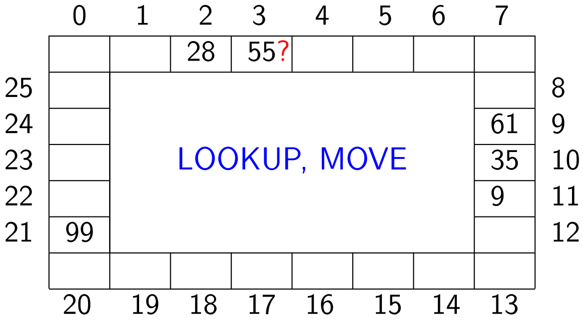
s
80



Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

s
80

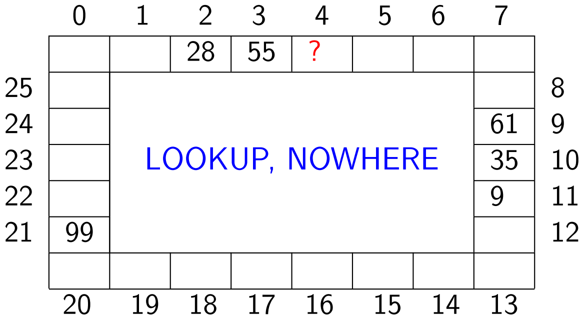


LOOKUP, MOVE

Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

s
80



0 1 2 3 4 5 6 7

28 55 ?

25

24

23

22

21 99

8

61 9

35 10

9 11

12

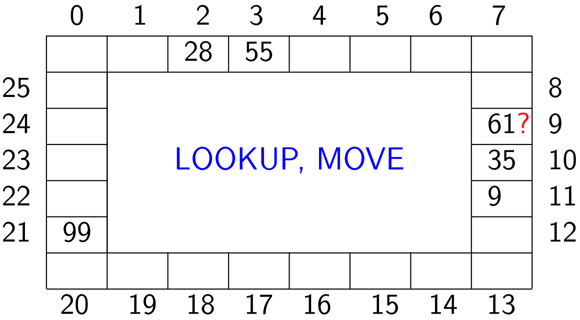
LOOKUP, NOWHERE

20 19 18 17 16 15 14 13

Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

s
35



0 1 2 3 4 5 6 7

28 55

25 8

24 61? 9

23 LOOKUP, MOVE 35 10

22 9 11

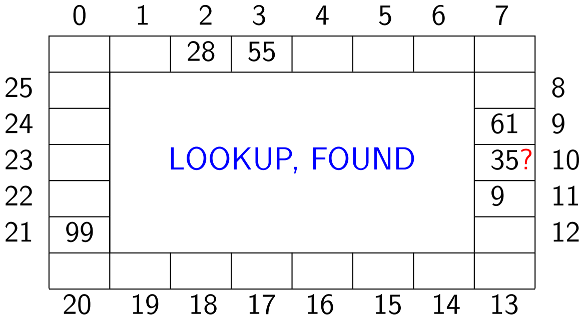
21 99 12

20 19 18 17 16 15 14 13

Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

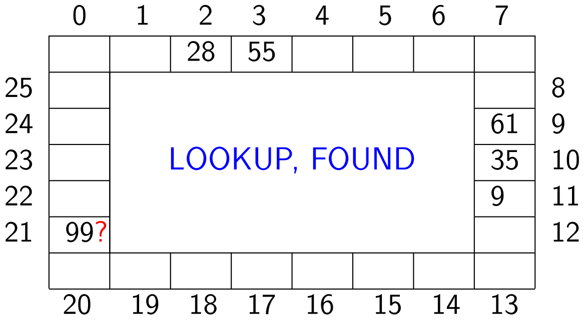
s
35



Open addressing with linear probing – Searching

Let $h'(k) = k\%26$.

s
99



0 1 2 3 4 5 6 7

28 55

25 8

24 61 9

23 LOOKUP, FOUND 35 10

22 9 11

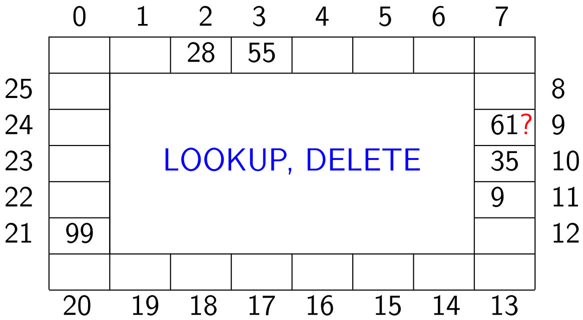
21 99? 12

20 19 18 17 16 15 14 13

Open addressing with linear probing – Deletion

Let $h'(k) = k\%26$.

s
61 ×



0	
1	
2	28
3	55
4	
5	
6	
7	
8	
9	61?
10	35
11	9
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	99
22	
23	
24	
25	

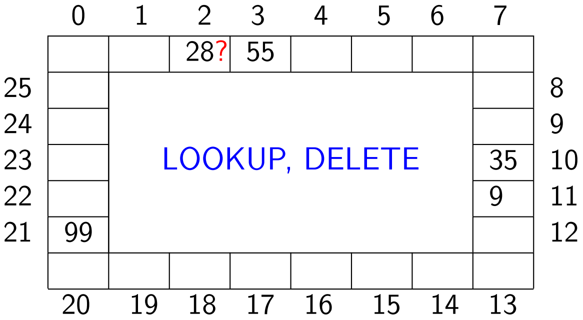
LOOKUP, DELETE

Note: The symbol represents the tombstones.

Open addressing with linear probing – Deletion

Let $h'(k) = k\%26$.

s
28 ×



0 1 2 3 4 5 6 7

28? 55

25 8

24 9

23 35 10

22 9 11

21 99 12

LOOKUP, DELETE

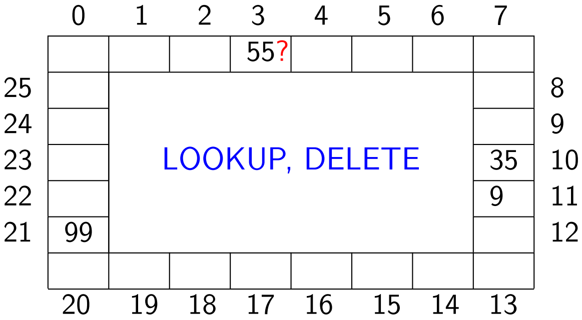
20 19 18 17 16 15 14 13

Open addressing with linear probing – Deletion

Let $h'(k) = k\%26$.

s

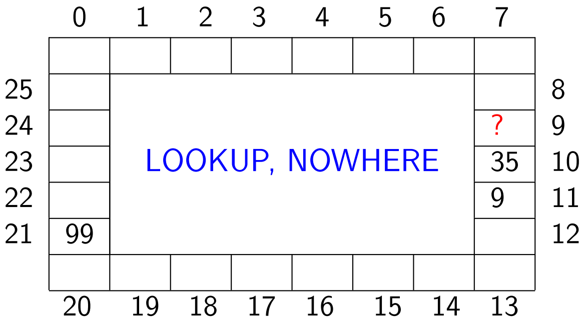
55 ×



Open addressing with linear probing – Deletion

Let $h'(k) = k\%26$.

s
35 ×



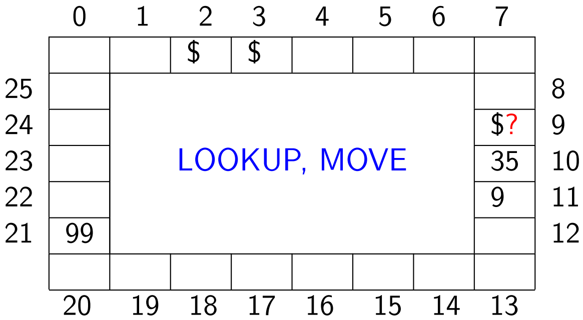
LOOKUP, NOWHERE

To trace linearly probed items, we have to keep track of the positions from where items have been deleted!!!

Open addressing with linear probing – Deletion

Let $h'(k) = k\%26$.

s
35 ×



Note: The *tombstones* (denoted with the symbol $) are used to keep track of the positions of deleted items.

Open addressing with quadratic probing

Quadratic probing uses a hash function of the form

$$h(k, i) = (h'(k) + (c_1 * i^2 + c_2 * i))\%m.$$

Here, h' is an auxiliary hash function, c_1 and c_2 are auxiliary constants, and $i = 0, 1, \ldots, m1$.

Note: It suffers from a problem known as *secondary clustering*.

Open addressing with double hashing

Double hashing uses a hash function of the form

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

The permutations produced have many of the characteristics of randomly chosen permutations.

Note: It has the only disadvantage that as soon as the hash table fills up the performance degrades.

Robin Hood hashing

Robinhood hashing is a variation of open addressing where keys can be moved after they are placed.

When an existing key is found during an insertion that is closer to its preferred location than the new key, it is displaced (relocation) to make room for it.

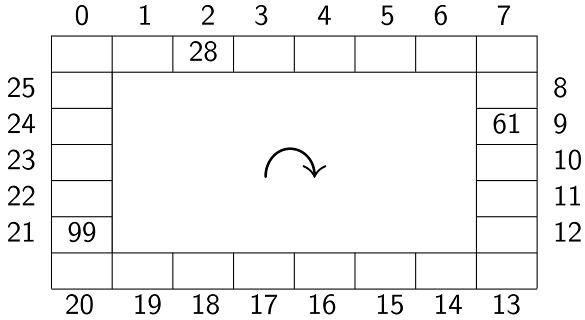
- This dramatically decreases the variance in the expected number of searchers (lookups).
- It also makes it possible to terminate searches (lookups) early.

Note: Assuming truly random hash functions, the variance of the expected number of probes required in Robin Hood hashing is $O(\log \log n)$.

Robin Hood hashing – Insertion

Let $h'(k) = k\%26$.

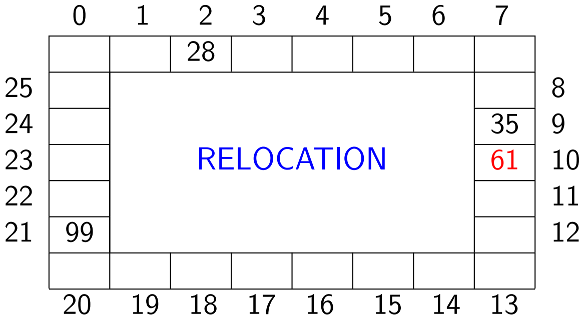
k
28
61
99
35 →
9
55



Robin Hood hashing – Insertion

Let $h'(k) = k\%26$.

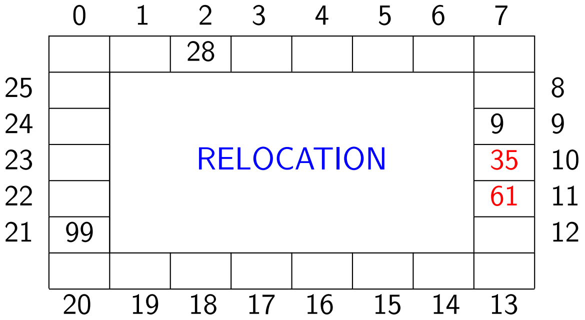
k
28
61
99
35
9 $\rightarrow$
55



Robin Hood hashing – Insertion

Let $h'(k) = k\%26$.

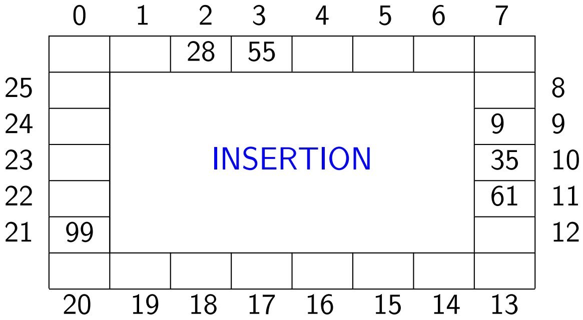
k
28
61
99
35
9
55 →



Robin Hood hashing – Insertion

Let $h'(k) = k\%26$.

k
28
61
99
35
9
55



Cuckoo hashing

We choose a pair of hash functions h_1 and h_2 such that $h_1 : [n] \to [m]$ and $h_2 : [n] \to [m]$.

We use two tables, each of which can accommodate m items. Every item $k \in R$ will either be at position $h_1(k)$ in the first table or at $h_2(k)$ in the second.

Cuckoo hashing

We choose a pair of hash functions h_1 and h_2 such that $h_1 : [n] \rightarrow [m]$ and $h_2 : [n] \rightarrow [m]$.

We use two tables, each of which can accommodate m items. Every item $k \in R$ will either be at position $h_1(k)$ in the first table or at $h_2(k)$ in the second.

Note: New hash functions might be required to be introduced in case of critical conditions.

Cuckoo hashing – Insertion

1. To insert an item k, start by inserting it into Table 1.
2. If $h_1(k)$ is empty, place k there.
3. Otherwise, place k there, taking out the old item k' and relocating it into Table 2 at $h_2(k')$.
4. Repeat this process, bouncing between the tables, until all the items stabilize.
5. If the same position is revisited with the same item to insert (known as a *cycle*), perform rehashing by choosing a new pair of hash functions and insert all items back into the tables.

Cuckoo hashing – Insertion

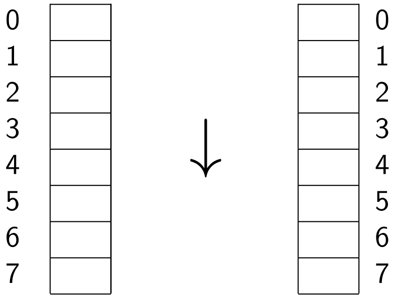
1. To insert an item k, start by inserting it into Table 1.
2. If $h_1(k)$ is empty, place k there.
3. Otherwise, place k there, taking out the old item k' and relocating it into Table 2 at $h_2(k')$.
4. Repeat this process, bouncing between the tables, until all the items stabilize.
5. If the same position is revisited with the same item to insert (known as a *cycle*), perform rehashing by choosing a new pair of hash functions and insert all items back into the tables.

Note: Multiple rehashes might be necessary before it succeeds.

Cuckoo hashing – Insertion

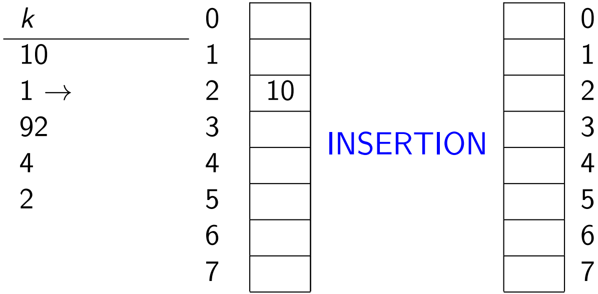
Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.

k
10 $\rightarrow$
1
92
4
2



Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.



k
10
1 →
92
4
2

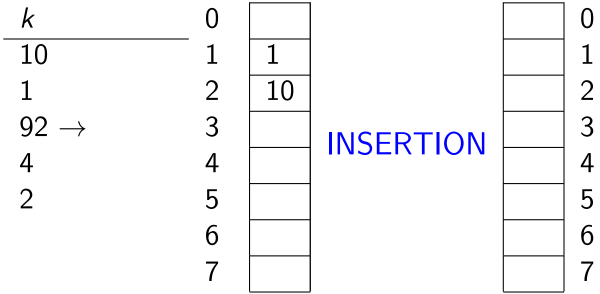
0	
1	
2	10
3	
4	
5	
6	
7	

INSERTION

	0
	1
	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.



k
10
1
92 →
4
2

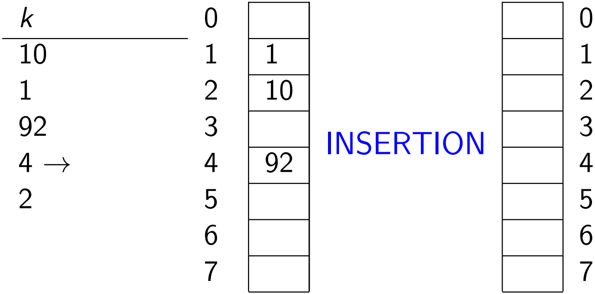
0	
1	1
2	10
3	
4	
5	
6	
7	

INSERTION

	0
	1
	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.



k
10
1
92
4 →
2

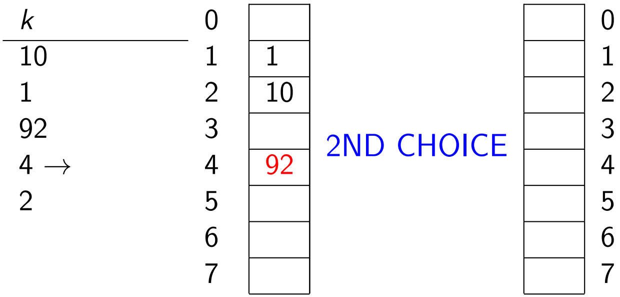
0	
1	1
2	10
3	
4	92
5	
6	
7	

INSERTION

	0
	1
	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10} k]$.



k
10
1
92
4 →
2

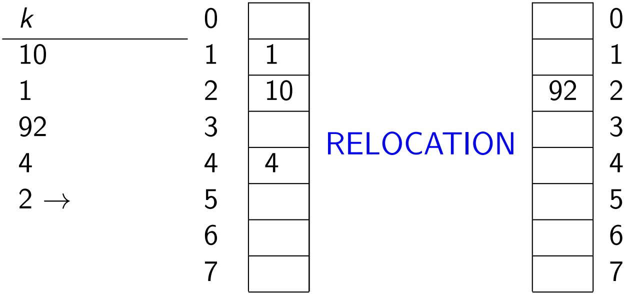
0	
1	1
2	10
3	
4	92
5	
6	
7	

2ND CHOICE

	0
	1
	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.



k
10
1
92
4
2 →

0	
1	1
2	10
3	
4	4
5	
6	
7	

RELOCATION

	0
	1
92	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.

k
10
1
92
4
2 $\rightarrow$

0	
1	1
2	10
3	
4	4
5	
6	
7	

2ND CHOICE

	0
	1
92	2
	3
	4
	5
	6
	7

Cuckoo hashing – Insertion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.

k
10
1
92
4
2

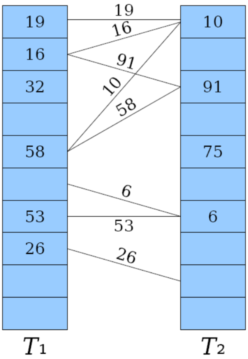
0	
1	1
2	2
3	
4	4
5	
6	
7	

RELOCATION

	0
10	1
92	2
	3
	4
	5
	6
	7

Cuckoo hashing – The cuckoo graph

- The ***cuckoo graph*** is a bipartite multigraph derived from a cuckoo hash table.
- Each table slot is a node.
- Each element is an edge.
- Edges link slots where each element can be.
- Each insertion introduces a new edge into the graph.

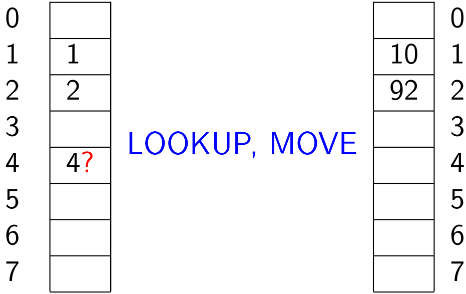


Note: An insertion in cuckoo hash tables traces a path through the cuckoo graph. An insertion succeeds iff the connected component containing the inserted item contains at most one cycle.

Cuckoo hashing – Deletion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.

k
92 ×



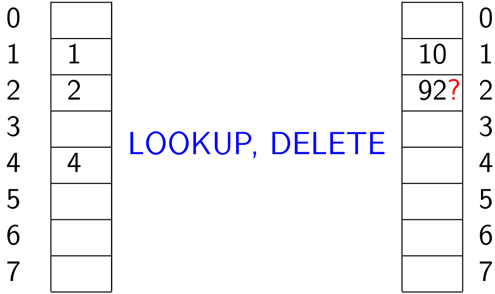
0			0
1	1	10	1
2	2	92	2
3			3
4	4?		4
5			5
6			6
7			7

LOOKUP, MOVE

Cuckoo hashing – Deletion

Let $h_1(k) = k\%8$ and $h_2(k) = [log_{10}k]$.

k
92 ×



0	
1	1
2	2
3	
4	4
5	
6	
7	

LOOKUP, DELETE

	0
10	1
92?	2
	3
	4
	5
	6
	7

Cryptography

Definition (One-way function)

A function $y = f(x)$ that satisfies the following two properties is known as a one-way function.

1. Given x, it is easy to compute $y = f(x)$.
2. Given y, it is computationally infeasible to compute $x = f^{-1}(y)$.

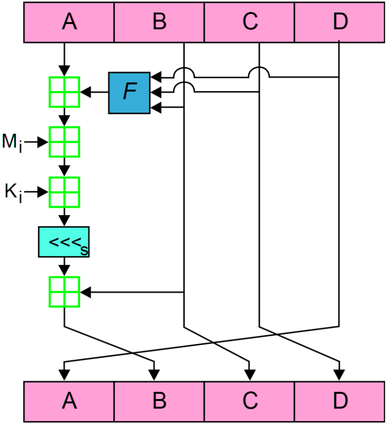
E.g., $y = f(x_1, x_2) = x_1 \times x_2$, where x_1 and x_2 are two large prime numbers, is a one-way function.

Note: A one-way function is not necessarily a single-variable function.

Cryptography

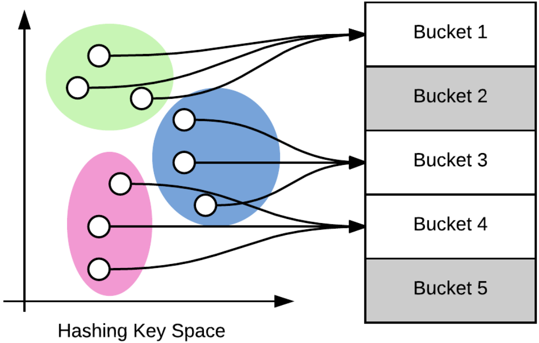
Requirement	Description
Variable input size	h can be applied to a block of data of any size
Fixed output size	h produces a fixed-length output
Efficiency	$h(k)$ is relatively easy to compute for any given k, making both hardware and software implementations practical
Preimage resistant (one-way property)	For any given hash value k^*, it is computationally infeasible to find k such that $k^* = h(k)$
Second preimage resistant (weak collision resistant)	For any given key k_1, it is computationally infeasible to find $k_2 \neq k_1$ with $h(k_1) = h(k_2)$
Collision resistant (strong collision resistant)	It is computationally infeasible to find any pair (k_1, k_2) such that $h(k_1) = h(k_2)$
Pseudorandomness	Output of k meets standard tests for pseudorandomness

Cryptography



The MD5 hashing mechanism

Dimensionality reduction



The locality sensitive hashing (LSH)

Problems – Day 19

1. Given a string with English alphabets as input, write a program to return its largest substring that contains equal number of vowels and consonants. The only constraint is that the time consumed by your program must be linear to the length of the string.
2. Write a program to implement Cuckoo hashing for storing some given numbers within $[1, n]$ with the following hash functions.

$$h_1(k) = k/2$$

$$h_2(k) = k\%(n/2)$$

   Consider that n is a user input.

Problems – Day 19

3. Given an integer matrix and a column number as input, write a program to efficiently find out all the columns that are permutation of the given column.

   Input Format:

   ```
   4 4 0 # No. of rows, No. of columns, input column
   30 70 40 20
   20 90 10 30
   40 30 20 10
   10 50 30 40
   ```

   Output: 2, 3

4. Write a program that will take a set of n distinct integers as user input and find out the largest x_4 in $O(n^2)$ time such that $x_1 + x_2 + x_3 = x_4$, where x_1, x_2, x_3 and x_4 are all distinct elements from the input set.