Lecture 4

Hashing Methods

Lecture Content

1. Basics

2. Collision Resolution Methods

2.1 Linear Probing Method

2.2 Quadratic Probing Method

2.3 Double Hashing Method

2.4 Coalesced Chaining Method

2.5 Separate/Direct Chaining Method

2.6 Cuckoo Hashing Method

1. Basics

- Search time in array is $O(n)$.
- Search time in tree is $O(\log n)$.
- Ideally, search time in array-based hash table is constant $O(1)$ in the average case.
- Array-based, linked list based, or tree-based hash table can be used.

1. Basics

- Some applications of hash tables are as follows.

- Hashing is a technique often used to implement an ADT dictionary.

- Compilers use hash tables to keep track of declared variables in source code. The data structure is called **symbol table**.

- Spelling checker: entire dictionary can be prehashed and words can be checked in constant time.

1. Basics

- In this lecture, we examine a data structure (i.e., hash table) which is designed specifically with the objective of providing efficient insertion and searching operations (deletion is not our primary objective).
- In order to meet the design objective (i.e., constant time performance), we do not require that there be any specific ordering of the items in the container.

1. Basics

• To achieve constant time performance objective, the implementation must be based in some way on an array rather than a linked list.

• This is because we can access the *k*-th element of an array in constant time, whereas the same operation in a linked list takes longer time.

1. Basics

- We are designing a container which will be used to hold some number of items of a given set K. We call the elements of the set K **keys** and K is called the **key space**.
- The general approach is to store the keys in an array. The position (also called location or index) i of a key k in the array is given by a function $f(k)$, called a **hash function**, which determines the position of a given key directly from that key

(i.e., $i = f(k)$) is called the **hash code** of k).

1. Basics

- In the general case, we expect the size of the set of keys, denoted as $|K|$, to be relatively large in comparison with the number of items stored in the container M (denoted as $M << |K|$).

- In other words, the number of items stored in the container is significantly less than $|K|$. We use an array of size M to contain items.

1. Basics

- Consequently, what we need is a function

$$f\colon K \rightarrow \{0, 1, \ldots, M - 1\}.$$

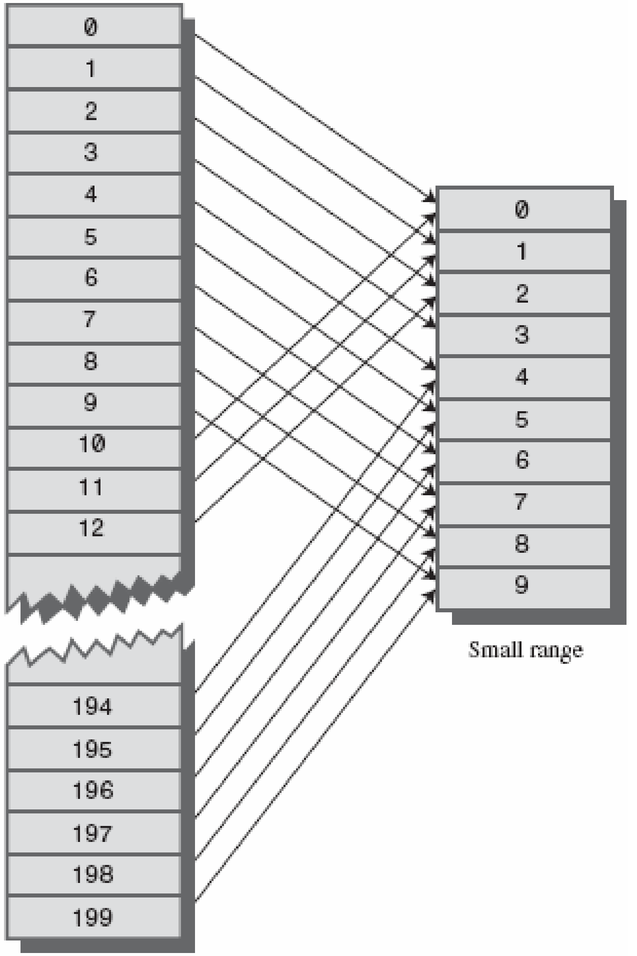
- This function maps (or transforms) the set of key values to be stored in the container to subscripts (indexes) in an array of length M.
- This function is called a **hash function**.

1. Basics

- In general, since $|K| >> M$, the mapping defined by a hash function will be a many-to-one mapping.
- That is, there will exist many pairs of distinct keys x and y, such that $x \neq y$, for which $f(x) = f(y)$.
- This situation is called a **collision** and f is not a **perfect hash function**.
- Several approaches for dealing with collisions are explored in the following sections.

1. Basics

- Example: $K = \{0, 1, ..., 199\}$, $M = 10$, for each key k in K, $f(k) = k \% M$.



1. Basics

- The characteristics of a good hash function are as follows.
  - It avoids collisions.
  - It tends to spread keys evenly in the array.
  - It is easy to compute (i.e., computational time of a hash function should be $O(1)$).

Lecture Content

1. Basics

2. Collision Resolution Methods

2.1 Linear Probing Method

2.2 Quadratic Probing Method

2.3 Double Hashing Method

2.4 Coalesced Chaining Method

2.5 Separate/Direct Chaining Method

2.6 Cuckoo Hashing Method

2. Collusion Resolution Methods

- Three methods in open addressing are linear probing, quadratic probing, and double hashing.
- These methods are of the **division hashing method** because the hash function is $f(k) = k \% M$.
- Some other hashing methods are middle-square hashing method, multiplication hashing method, and Fibonacci hashing method, and so on.

2.1 Linear Probing Method

- The hash table in this case is implemented using an array (i.e., adjacent list) containing M nodes (or elements), each node of the hash table has a field k used to contain the key of the node.
- M can be any positive integer but M is often chosen to be a prime number.
- When the hash table is initialized, all fields k are assigned to -1 (i.e., empty or vacant).

2.1 Linear Probing Method

- When a node with the key k needs to be added into the hash table, the hash function

$$f(k) = k \% M$$

will specify the address $i = f(k)$ (i.e., an index of an array) within the range $[0, M - 1]$.

2.1 Linear Probing Method

- If there is no conflict (i.e., the cell i is unoccupied), then this node is added into the hash table at the address i.

- If a conflict takes place, then the hash function rehashes first time f_1 to consider the next address (i.e., $i + 1$). If conflict occurs again, then the hash function rehashes second time f_2 to examine the next address (i.e., $i + 2$). This process repeats until the available address found then this node will be added at this address.

2.1 Linear Probing Method

- The rehash function at the time t (i.e., the collision number t = 1, 2, ...) is presented as follows

$$f_t(k) = (f(k) + t)\ \%\ M = (i + t)\ \%\ M$$

- When searching a node, the hash function $f(k)$ will identify the address i (i.e., $i = f(k)$) falling between 0 and M - 1.

2.1 Linear Probing Method

- **Example**: insert keys 32, 53, 22, 92, 17, 34, 24, 37, and 56 into a hash table of size M = 10.

(a)

0	-1
1	-1
2	32
3	53
4	-1
5	-1
6	-1
7	-1
8	-1
9	-1

(b)

0	-1
1	-1
2	32
3	53
4	22
5	92
6	-1
7	-1
8	-1
9	-1

(c)

0	-1
1	-1
2	32
3	53
4	22
5	92
6	34
7	17
8	-1
9	-1

(d)

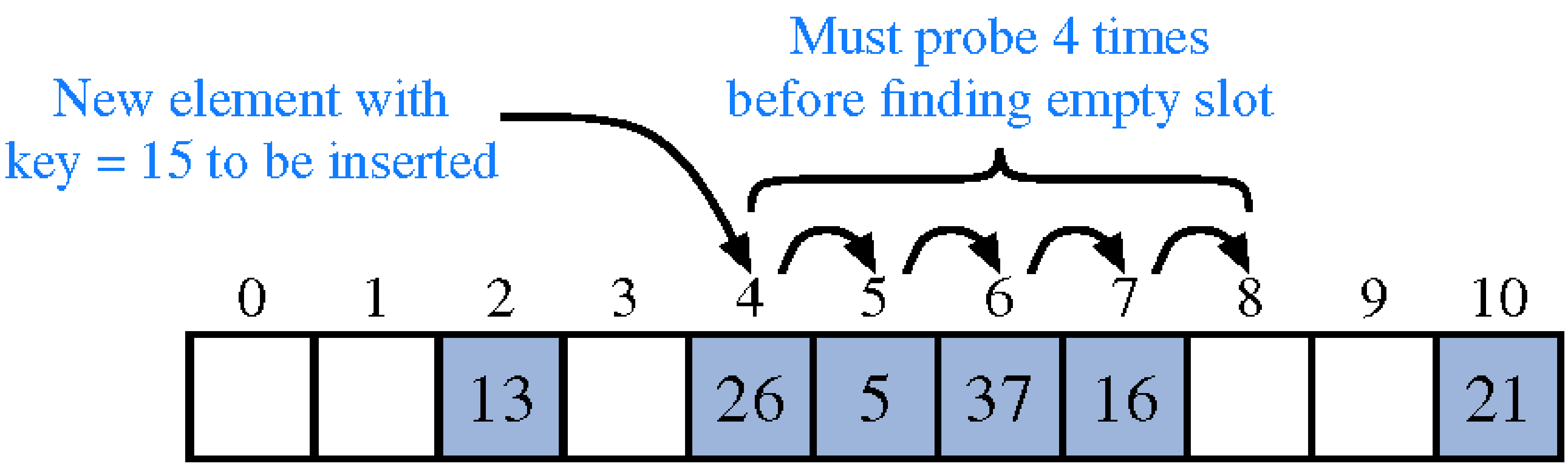
0	-1
1	-1
2	32
3	53
4	22
5	92
6	34
7	17
8	24
9	37

(e)

0	56
1	-1
2	32
3	53
4	22
5	92
6	34
7	17
8	24
9	37

2.1 Linear Probing Method

- **Example**: Insertion the keys 13, 26, 5, 37, 16, and 15 into a hash table with integer keys using hash function $f(k) = k \bmod 11$. The value $f(k)$ is called the **hash code** of the key k.



2.1 Linear Probing Method

- The drawback of the linear probing method is that it causes **the primary clustering problem**.

- That is, it creates a long sequence of filled slots.

- In other words, primary clustering occurs when the keys hashed to different locations trace the same sequence in looking for an empty location.

2.2 Quadratic Probing Method

- Quadratic probing is an attempt to keep clusters from forming.

- The idea is to probe more widely separated cells, instead of those adjacent to the primary hash site.

2.2 Quadratic Probing Method

- The hash table in this case is implemented using an array (i.e., adjacent list) containing M nodes, each node of the hash table has a field k used to contain the key of the node.

- When the hash table is initialized, all fields k are assigned to -1.

2.2 Quadratic Probing Method

• When a node with the key k needs to be added into the hash table, the hash function

$$f(k) = k \% M$$

will specify the address i within the range $[0, M - 1]$ (i.e., $i = f(k)$).

2.2 Quadratic Probing Method

- If there is no conflict, then this node is added into the hash table at the address *i*.

- If a conflict takes place, then the hash function rehashes first time f_1 to consider the address $f(k) + 1^2$. If conflict occurs again, then the hash function rehashes second time f_2 to examine the address $f(k) + 2^2$. This process repeats until the available address found then this node will be added at this address.

2.2 Quadratic Probing Method

- The rehash function at the time t (i.e., the collision number t = 1, 2, ...) is presented as follows.

$$f_t(k) = (f(k) + t^2)\ \%\ M = (i + t^2)\ \%\ M$$

- When searching a node, the hash function $f(k)$ will identify the address i (i.e., $i = f(k)$) falling between 0 and M - 1.

2.2 Quadratic Probing Method

- **Example**: insert the keys 10, 15, 16, 20, 30, 25, 26, and 36 into a hash table of size $M = 10$.

(a)

0	10
1	-1
2	-1
3	-1
4	-1
5	15
6	16
7	-1
8	-1
9	-1

(b)

0	10
1	20
2	-1
3	-1
4	-1
5	15
6	16
7	-1
8	-1
9	-1

(c)

0	10
1	20
2	-1
3	-1
4	30
5	15
6	16
7	-1
8	-1
9	25

(d)

0	10
1	20
2	-1
3	-1
4	30
5	15
6	16
7	26
8	-1
9	25

(e)

0	10
1	20
2	36
3	-1
4	30
5	15
6	16
7	26
8	-1
9	25

2.2 Quadratic Probing Method

- The problem with quadratic probing method is that it causes **the secondary clustering problem**.

- Secondary clustering occurs when keys which hash to the same location trace the same sequence in looking for an empty location.

2.3 Double Hashing Method

- To eliminate secondary clustering problem, we can use another approach: **double hashing**.
- Double hashing is one of the most effective methods of probing an open addressed hash table.
- It works by adding the **hash codes** of two hash functions. This method uses two functions f and g as the hash functions.

2.3 Double Hashing Method

• When a node with the key k needs to be added into the hash table, the first hash function

$$f(k) = k \,\%\, M = i$$

will specify the address i within a range $[0, M - 1]$ (i.e., $i = f(k)$), where M is a prime number.

• A prime number M produces fewer collisions.

2.3 Double Hashing Method

- If there is no conflict, this node is added into the hash table at the address i.

- If a conflict takes place, the second hash function $g(k) = c - (k \% c) = j$, where the constant c is a prime number less than hash table size M and j is called **step size**, is used.

2.3 Double Hashing Method

Then, the first hash function rehashes the first time to consider the address $f_1(k) = (f(k) + j) \% M = (i + j) \% M = i_1$.

If conflict happens again, the first hash function rehashes the second time to consider the address $f_2(k) = (f_1(k) + j) \% M = i_2$. This process repeats until the available address found then this node will be added at this address.

2.3 Double Hashing Method

• The rehash function at the time t (i.e., the collision number t = 1, 2, ...) is presented as follows.

$$f_t(k) = (f_{t\text{-}1}(k) + j) \; \% \; M = (i_{t\text{-}1} + j) \; \% \; M,$$

where $f_0(k) = i_0 = i = f(k)$, $j = g(k) = c - (k \% c)$, c and M are primes, $c < M$.

• Double hashing requires that the size of the hash table is a prime number (e.g., $M = 11$).

2.3 Double Hashing Method

- **Example**: insert the keys 14, 17, 25, 37, 34, 16, and 26 into a hash table of size $M = 11$, $c = 5$.

0	-1
1	-1
2	-1
3	14
4	-1
5	-1
6	17
7	-1
8	-1
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	26
10	-1

2.3 Double Hashing Method

	Probes
14 mod 11 = 3 ✓	1
17 mod 11 = 6 ✓	1
25 mod 11 = 3 ×	**2**
37 mod 11 = 4 ✓	1
34 mod 11 = 1 ✓	1
16 mod 11 = 5 ✓	1
26 mod 11 = 4 ×	**5**

0	-1
1	-1
2	-1
3	14
4	-1
5	-1
6	17
7	-1
8	-1
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	26
10	-1

- $j = 5 - (25 \% 5) = 5$,

$$i_1 = (i_0 + j) \% 11 = (i + j) \% 11 = (3 + 5) \% 11 = 8$$

2.3 Double Hashing Method

	Probes
14 mod 11 = 3 ✓	1
17 mod 11 = 6 ✓	1
25 mod 11 = 3 ×	**2**
37 mod 11 = 4 ✓	1
34 mod 11 = 1 ✓	1
16 mod 11 = 5 ✓	1
26 mod 11 = 4 ×	**5**

0	-1
1	-1
2	-1
3	14
4	-1
5	-1
6	17
7	-1
8	-1
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	-1
10	-1

0	-1
1	34
2	-1
3	14
4	37
5	16
6	17
7	-1
8	25
9	26
10	-1

- $j = 5 - (26 \% 5) = 4$,

$i_1 = (i_0 + j) \% 11 = (4 + 4) \% 11 = 8$, ×

$i_2 = (i_1 + j) \% 11 = (8 + 4) \% 11 = 1$, ×

$i_3 = (i_2 + j) \% 11 = (1 + 4) \% 11 = 5$, ×

$i_4 = (i_3 + j) \% 11 = (5 + 4) \% 11 = 9$ ✓

2.4 Coalesced Chaining Method

- The hash table in this case is implemented using an array containing *M* nodes.
- Each node of the hash table is a class consisting of two fields as follows.

  Field *k*: contains the key of the node.

  Field *next*: contains a reference to next node if conflict occurs.
- When the hash table is initialized, all fields *k* and *next* are assigned to -1.

2.4 Coalesced Chaining Method

• When a node with the key k needs to be added into the hash table, the hash function $f(k) = k \% M = i$ will identify the address i within the range $[0, M - 1]$.

- If there is no conflict, then this node will be added into the hash table at address i.

2.4 Coalesced Chaining Method

- If a conflict happens, then this node will be added into the hash table at the first available address, say *j*, from the bottom of the hash table. The field *next* of **the last node** at index *i* **in the chain** will be updated to *j*.

2.4 Coalesced Chaining Method

- **Example**: insert the keys 10, 15, 26, 30, 25, and 35 into a hash table of size 10.

(a)

0	10	-1
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	-1
6	26	-1
7	-1	-1
8	-1	-1
9	-1	-1

(b)

0	10	9
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	-1
6	26	-1
7	-1	-1
8	-1	-1
9	30	-1

2.4 Coalesced Chaining Method

- **Example**: insert the keys 10, 15, 26, 30, 25, and 35 into a hash table of size 10.

(b)

0	10	9
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	-1
6	26	-1
7	-1	-1
8	-1	-1
9	30	-1

(c)

0	10	9
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	8
6	26	-1
7	-1	-1
8	25	-1
9	30	-1

2.4 Coalesced Chaining Method

- **Example**: insert the keys 10, 15, 26, 30, 25, and 35 into a hash table of size 10.

(c)

0	10	9
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	8
6	26	-1
7	-1	-1
8	25	-1
9	30	-1

(c)

0	10	9
1	-1	-1
2	-1	-1
3	-1	-1
4	-1	-1
5	15	8
6	26	-1
7	35	-1
8	25	7
9	30	-1

2.5 Separate (or Direct) Chaining Method

- The hash table is implemented by using singly linked list.

- Nodes on the hash table are hashed into M (e.g., $M = 10$) singly linked lists (from list 0 to list M - 1).

- Nodes conflicted at the address i are directly connected in the list i, where $0 \le i \le M - 1$.

2.5 Direct (or Separate) Chaining Method

- When a node with the key k is added into the hash table, the hash function $f(k) = k \% M$ will identify the address i between 0 and $M - 1$ corresponding the singly linked list i where this node will be added.

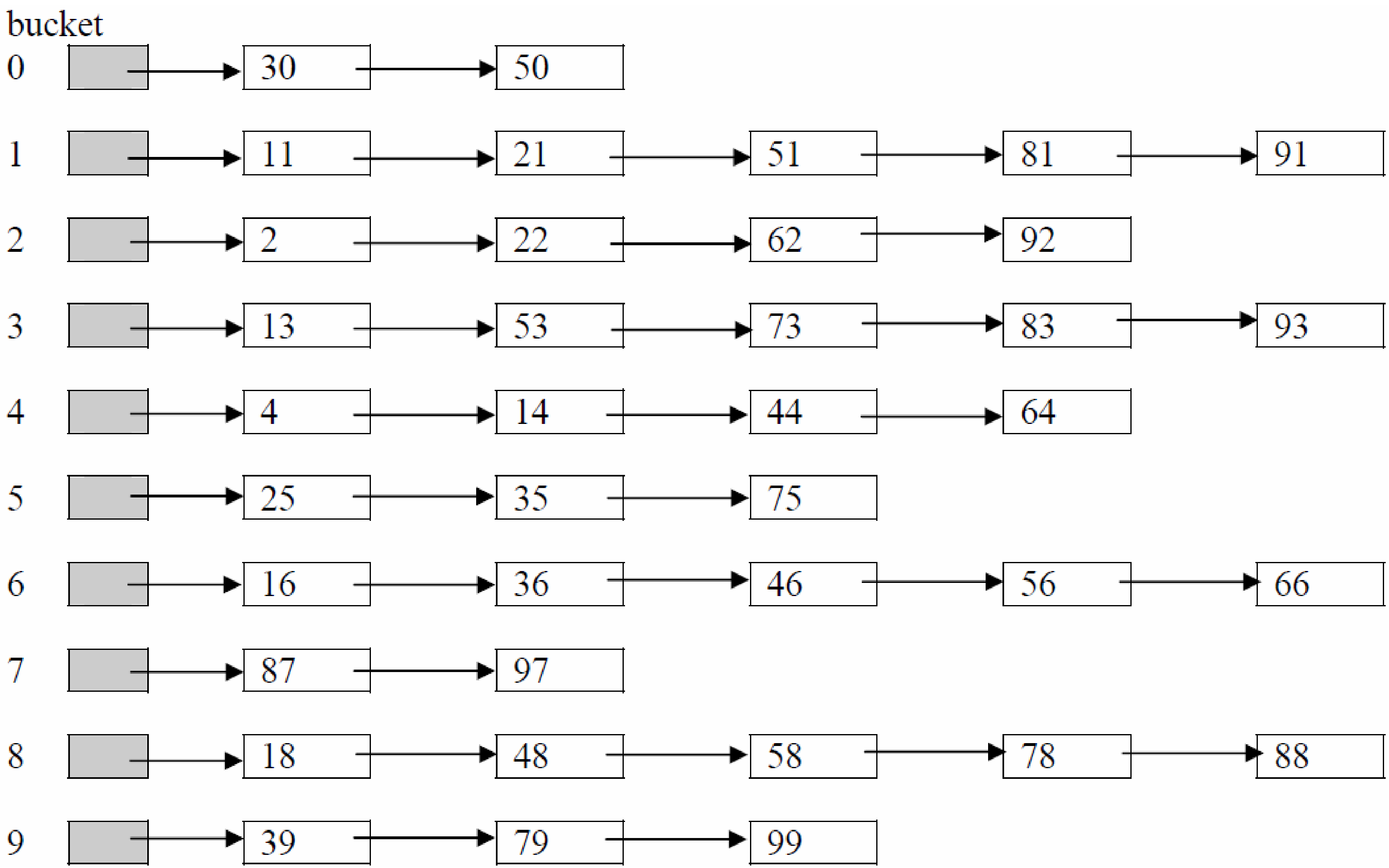
2.5 Direct (or Separate) Chaining Method

- When a node needs to be searched on the hash table, the hash function $f(k) = k \% M$ will specify the address i within the range $[0, M - 1]$ corresponding the singly linked list i that may contain the node.

- Searching a node on the hash table turns out the problem of searching an element on the singly linked list.

2.5 Direct (or Separate) Chaining Method

- **Example**: insert 13, 4, 16, 87, 11, 30, 2, 18, 25, ...



2.6 Cuckoo Hashing Method

- In the cuckoo hashing scheme, we use two hash tables T_0 and T_1, each of size M.
- In addition, we use a hash function h_0 for T_0 and a different hash function h_1 for T_1.
- For any given key $k \in K$, there are two possible places where we can store the key k, namely, either in $T_0[h_0(k)]$ or $T_1[h_1(k)]$.
- For example, the key $k = \mathrm{A}(0, 2)$ will be stored at index 0 in T_0 and at index 2 in T_1 (i.e., $h_0(k) = 0$, $h_1(k) = 2$).

2.6 Cuckoo Hashing Method

procedure insert(k)

if ($k = T_0[h_0(k)]$) **or** ($k = T_1[h_1(k)]$) **then return**;

$i \leftarrow 0$

2.6 Cuckoo Hashing Method

loop t times

if $T_i[h_i(k)]$ is empty **then**

$T_i[h_i(k)] \leftarrow k$

return

$temp \leftarrow T_i[h_i(k)]$

$T_i[h_i(k)] \leftarrow k$ // cuckoo eviction, kicking out, displace

$k \leftarrow temp$

$i \leftarrow (i + 1) \bmod 2$ // $i = 0, 1$

Rehash all the items including k using new hash functions h_0 and h_1.

2.6 Cuckoo Hashing Method

function *search*(k)

  return ($k = T_0[h_0(k)]$) **or** ($k = T_1[h_1(k)]$)

end;

2.6 Cuckoo Hashing Method

- **Example**: insert the keys A(0, 2), B(0, 0), C(1, 4), D(1, 0), E(3, 2), F(3, 4) into the hash tables T_0 and T_1

Table 0	
0	
1	
2	
3	
4	

Table 1	
0	
1	
2	
3	
4	

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

2.6 Cuckoo Hashing Method

Table 0	
0	A
1	
2	
3	
4	

Table 1	
0	
1	
2	
3	
4	

A: 0, 2

Cuckoo hash table after insertion of A

Table 0	
0	B
1	
2	
3	
4	

Table 1	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0

Cuckoo hash table after insertion of B
// *temp* = A, k = A, i = 1,

Table 0	
0	B
1	C
2	
3	
4	

Table 1	
0	
1	
2	A
3	
4	

A: 0, 2
B: 0, 0
C: 1, 4

Cuckoo hash table after insertion of C

Table 0	
0	B
1	D
2	
3	E
4	

Table 1	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2

Cuckoo hash table after insertion of D, and then E
// *temp* = C, k = C, i = 1,

2.6 Cuckoo Hashing Method

Table 0	
0	B
1	D
2	
3	F
4	

Table 1	
0	
1	
2	A
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Cuckoo hash table starting the insertion of F into the hash table. First, F displaces E.
// $temp = E$, $k = E$, $i = 1$,

Table 0	
0	B
1	D
2	
3	F
4	

Table 1	
0	
1	
2	E
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Continue the insertion of F into the hash table. Next, E displaces A.
// $temp = A$, $k = A$, $i = 0$,

Table 0	
0	A
1	D
2	
3	F
4	

Table 1	
0	
1	
2	E
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Continue the insertion of F into the hash table. Next, A displaces B.
// $temp = B$, $k = B$, $i = 1$,

Table 0	
0	A
1	D
2	
3	F
4	

Table 1	
0	B
1	
2	E
3	
4	C

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Completing the insertion of F into the hash table. Miraculously, B finds an empty position in Table 1.

2.6 Cuckoo Hashing Method

- We cannot successfully insert *G* with hash locations (1, 2).
- *G* displaces *D*, *D* displaces *B*, *B* displaces *A*, *A* displaces *E*, *E* displaces *F*, *F* displaces *C*, and *C* displaces *G* which was placed there at the start.
- *G* is inserted into its alternate in Table 1 (location 2). *G* displaces *A*, *A* displaces *B*, *B* displaces *D*, *D* displaces *C*, *C* displaces *F*, *F* displaces *E*, and *E* displaces *G* from position 2.
- At this point, the initial hash tables appear again. A cycle is detected. A rehash is required.

2.6 Cuckoo Hashing Method

- A family H of hash functions is 2-**universal** if for any two distinct keys x and y in K, where $x \neq y$, and for a hash function h chosen uniformly at random from H, we have

$$\text{Probability}(h(x) = h(y)) \leq 1 / M,$$

where M is the hash table size.

2.6 Cuckoo Hashing Method

- The universal family H of hash functions is defined as follows.

$$H = \{h_{a,b}(x) = ((ax + b) \bmod p) \bmod M, \text{ where } 1 \le a \le p - 1, 0 \le b \le p - 1\},$$

where a and b are chosen randomly and p is a prime larger than the largest input key.

- H has $p(p - 1)$ possible hash functions.

2.6 Cuckoo Hashing Method

- For example, three random choices of (a, b) yield three different hash functions:

$$h_{3,7}(x) = ((3x + 7) \bmod p) \bmod M$$

$$h_{4,1}(x) = ((4x + 1) \bmod p) \bmod M$$

$$h_{8,0}(x) = ((8x) \bmod p) \bmod M$$

Exercises

Write the following complete Java programs

- `LP.java` for linear probing method,
- `QP.java` for quadratic probing method,
- `DH.java` for double hashing method,
- `CC.java` for coalesced chaining method, and
- `SC.java` separate/direct chaining method.

Hash Code for a String

- The hash code of a given string s can be computed as follows.

$$f(s) = a_0x^{n-1} + a_1x^{n-2} + \ldots + a_{n-2}x + a_{n-1},$$

// a polynomial of degree n - 1

where a_i is the Unicode of s_i for $0 \leq i \leq n - 1$ and x is a positive constant (e.g., $x = 31$).

Hash Code for a String

```
int f = 0, n = length(s);
for (int i = 0; i < n; i++)
    f = f * x + a[i];
```

References

1. Adam Drozdek. 2005. *Data Structures and Algorithms in Java*. 2Ed. Cengage Learning.

ISBN: 0534492525.

2. Mark Allen Weiss. 2011. *Data Structures and Algorithm Analysis in Java*. 3Ed. Prentice Hall. ISBN: 0132576279.

3. Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser . 2014. *Data Structures and Algorithms in Java*. 6 Ed. Wiley. ISBN: 978-1-118-77133-4.

References

4. Bruno R. Preiss. 1999. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*. Wiley. ISBN: 0471346136.