AST20105 Data Structures & Algorithms

CHAPTER 11 – HASHING

Instructed by Garret Lai

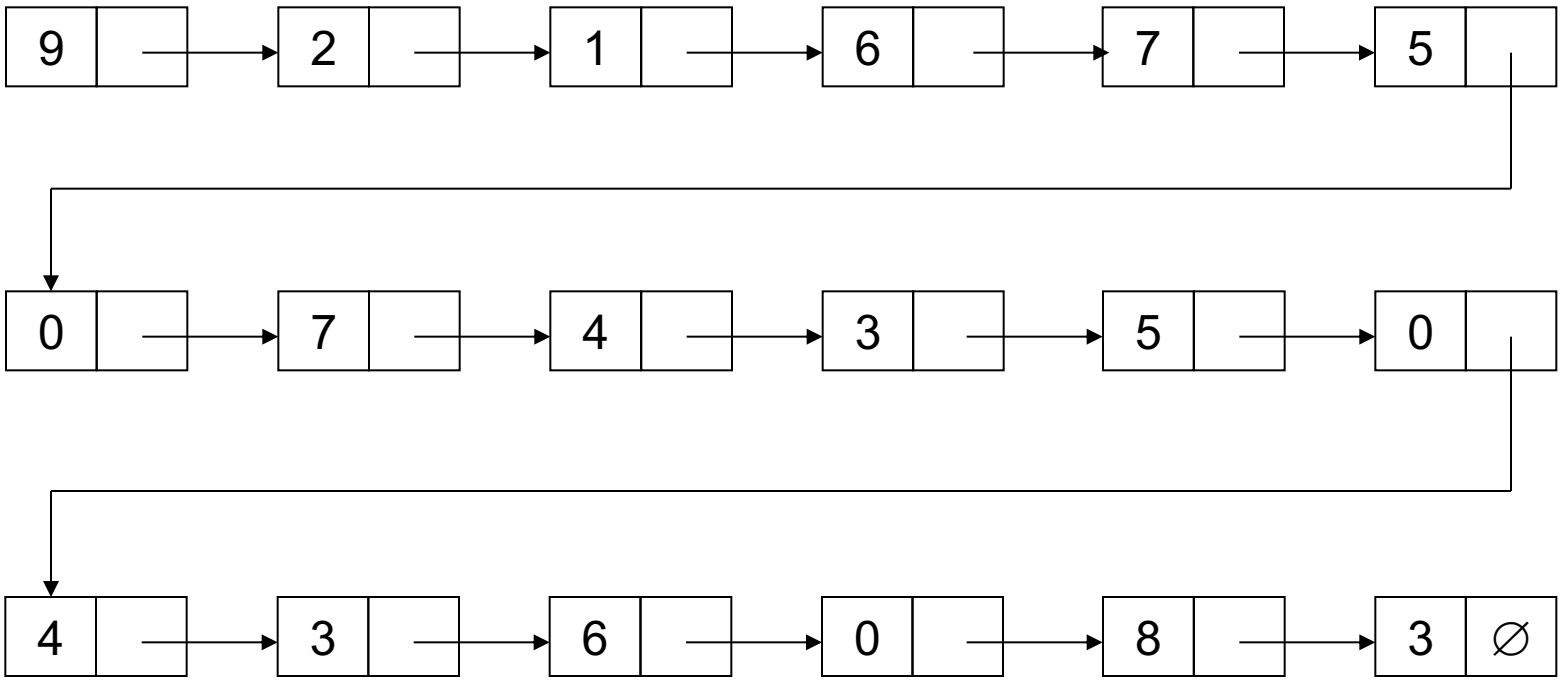
Before Start

- How many steps is needed to find 8 in the array?

12	5	18	19	2	13	10	20	1	6	17	15	3	4	16	11	9	7	8	14

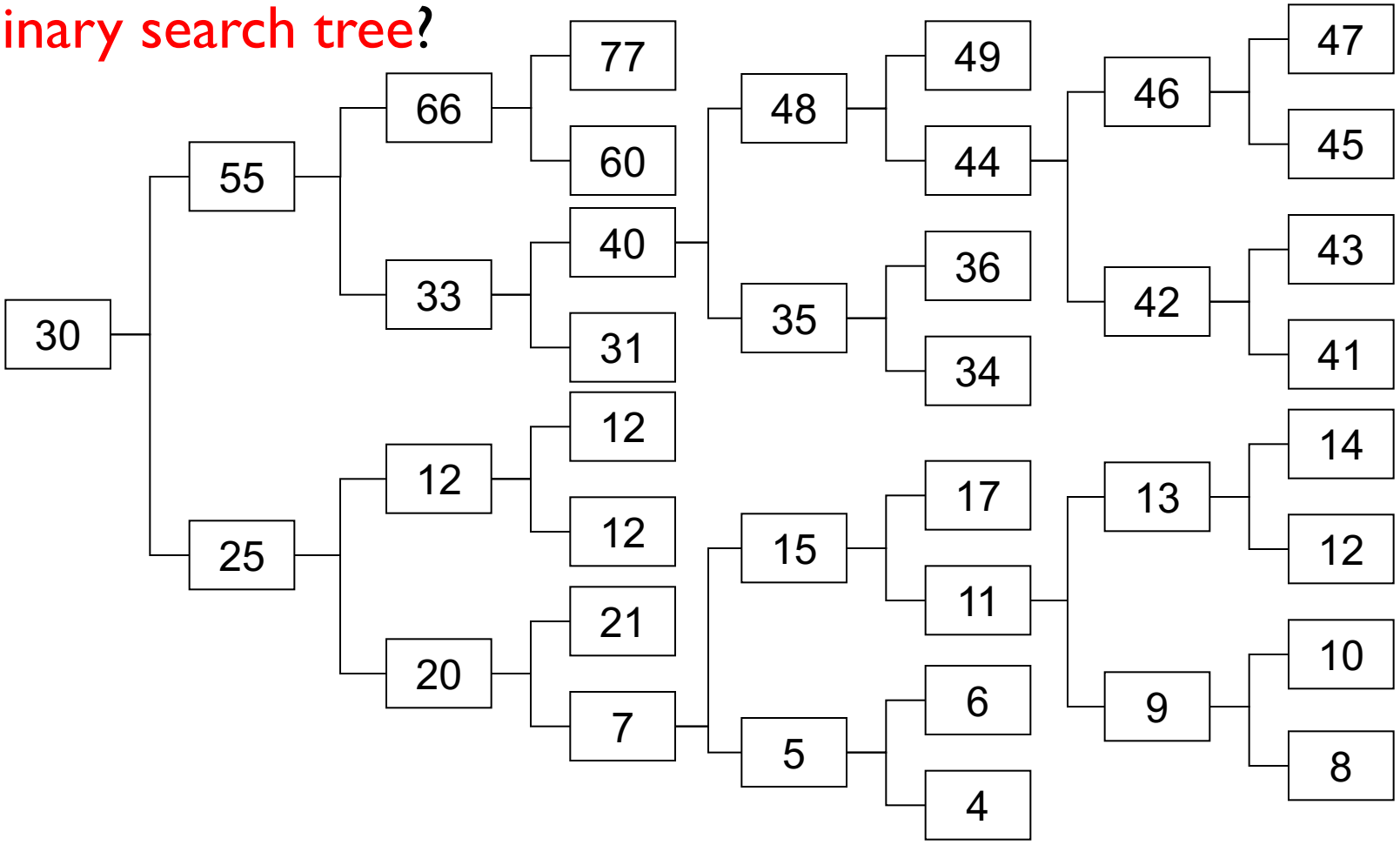
Before Start

- How many steps is needed to find 8 in the linked list?



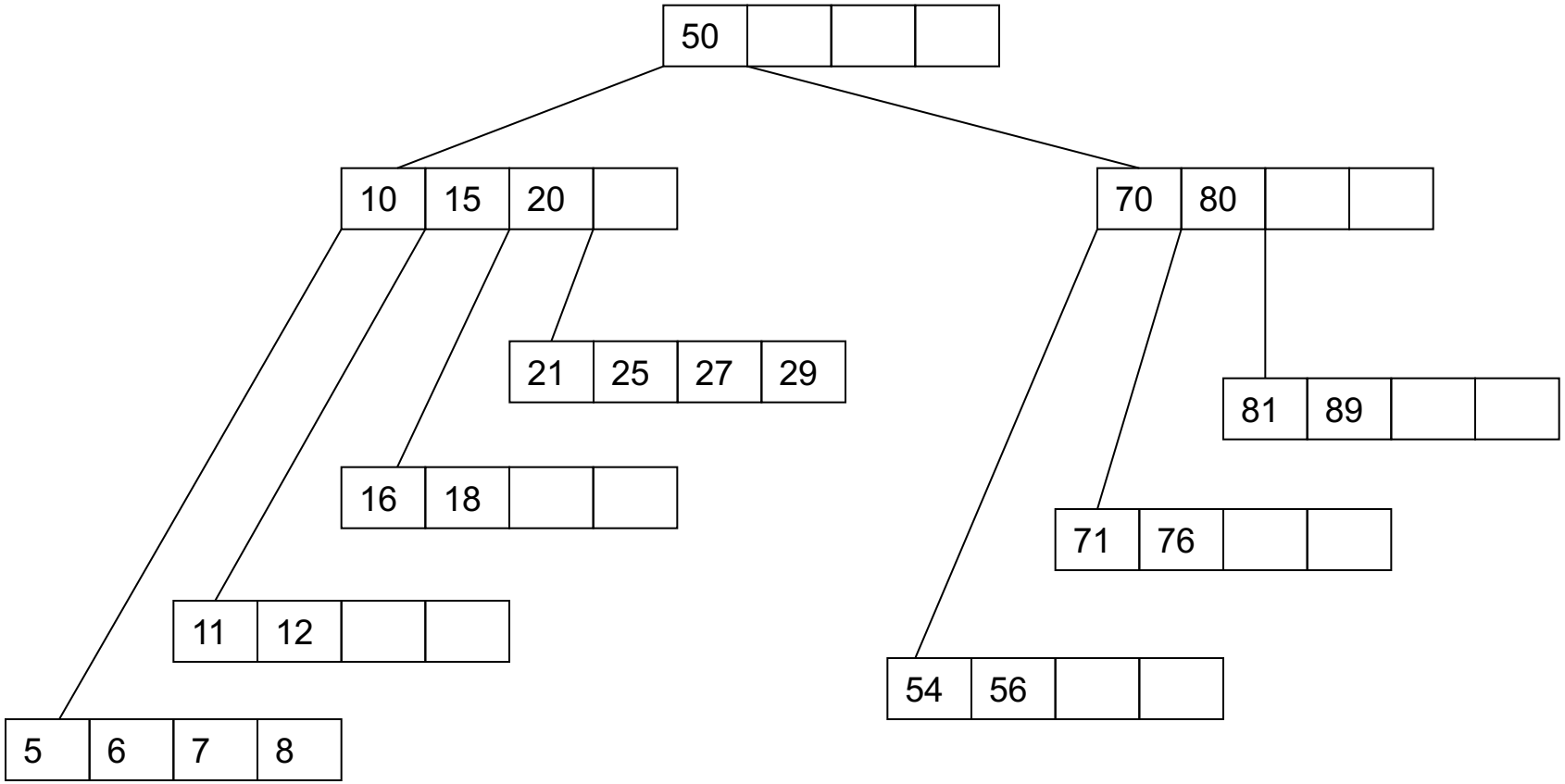
Before Start

- How many steps is needed to find 8 in the binary search tree?



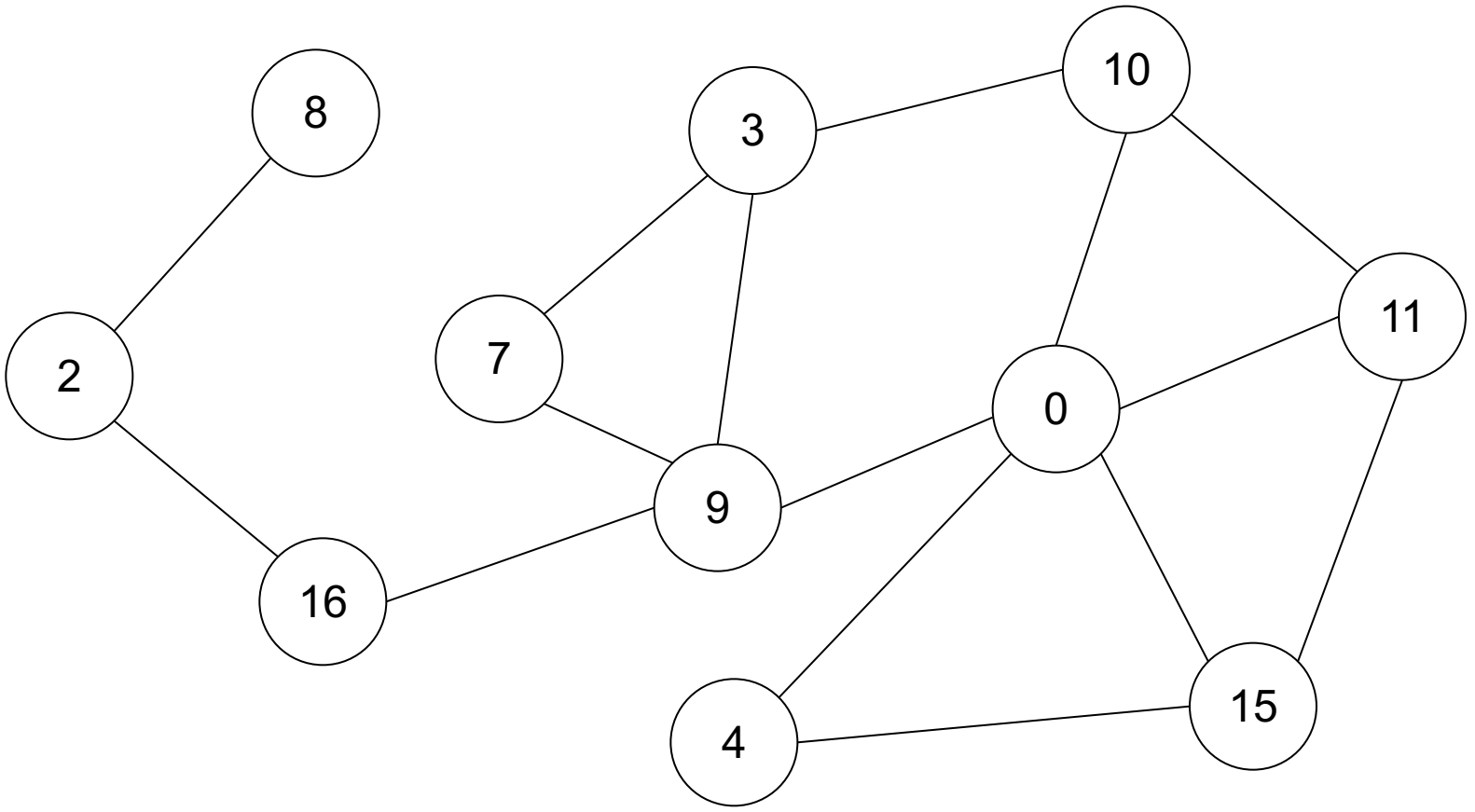
Before Start

- How many steps is needed to find 8 in the B-Tree?



Before Start

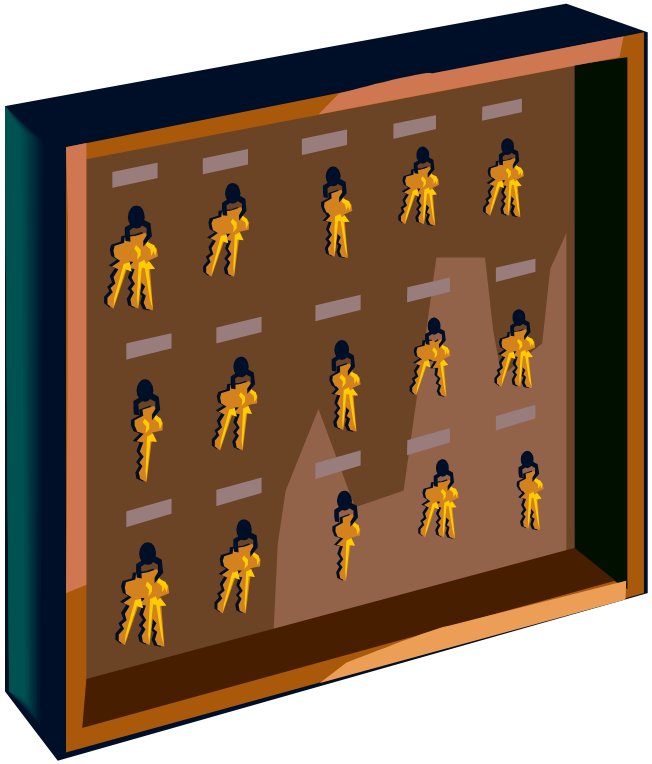
- How many steps is needed to find 8 in a Graph?



Before Start

- Can we speed up the searching process?

Hashing



Hash Table

- A hash table is a data structure storing keys / data in an indexed table

- Basically hash table maps unique keys to associated values.

- In the view of implementation, hash table is an array-based data structure, which uses hash function to convert the key into the index of an array element, where associated value is to be sought.

Hash Function

- Hash function is very important part of hash table design.
- A hash function, denoted as h, provides a method to find the table index from key / data
  - In other words, it maps keys into locations of a hash table
- A key k hashes to location h(k), where h(k) is the hash value of k
- Hashing refers to a process of inserting keys / data to hash table with the help of hash function

Hash Function

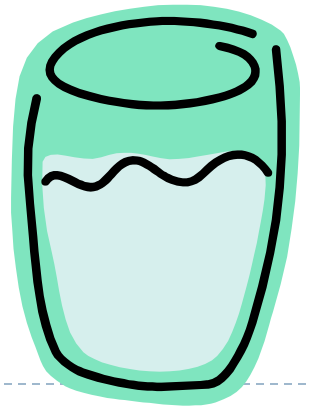
- Hash function is considered to be good, if it provides uniform distribution of hash values.

- The reason, why hash function is a subject to the principal concern, is that poor hash functions cause collisions and some other unwanted effects, which badly affect hash table overall performance.

Load Factor

- Basic underlying data structure used to store hash table is an array.

- The load factor is the ratio between the number of stored items and array's size.



Load Factor

- Hash table can whether be of a constant size or being dynamically resized, when load factor exceeds some threshold.

- Resizing is done before the table becomes full to keep the number of collisions under certain amount and prevent performance degradation.

Collision

- What happens, if hash function returns the same hash value for different keys?
- It yields an effect, called *collision*.
- Solutions:
  - Design a good hash function that can be computed efficiently and minimize the number of collision
  - Design collision resolution algorithm

Collision

- Collisions are practically unavoidable and should be considered when one implements hash table.
- Due to collisions, keys are also stored in the table, so one can distinguish between key-value pairs having the same hash.
- There are various ways of *collision resolution*. Basically, there are two different strategies:
  - Closed addressing
  - Open addressing

Closed Addressing

- **Closed addressing (open hashing).**
  - Each slot of the hash table contains a link to another data structure (i.e. linked list), which stores key-value pairs with the same hash.
  - When collision occurs, this data structure is searched for key-value pair, which matches the key.

Open Addressing

- **Open addressing (closed hashing).**
  - Each slot actually contains a key-value pair.
  - When collision occurs, open addressing algorithm calculates another location (i.e. next one) to locate a free slot.
  - Hash tables, based on open addressing strategy experience drastic performance decrease, when table is tightly filled (load factor is 0.7 or more).

Hash Table Example

Hash Table Example

- In the following, we show the very simple hash table example.
  - It uses simple hash function,
  - collisions are resolved using linear probing (open addressing strategy) and
  - hash table has constant size.
- This example clearly shows the basics of hashing technique.

Hash Table

- Underlying array has constant size to store 16 elements and each slot contains key-value pair.

- Key is stored to distinguish between key-value pairs, which have the same hash.

Hash Function

- Table allows only integers as values.
- Hash function to be used is the remainder of division by 16.
- In the view of implementation, this hash function can be encoded using remainder operator or using bitwise AND with 15.

Collision Resolution Strategy

- Linear probing is applied to resolve collisions.

- In case the slot, indicated by hash function, has already been occupied, algorithm tries to find an empty one by probing consequent slots in the array.

Collision Resolution Strategy

- *Note.*
  - Linear probing is not the best technique to be used when table is of a constant size.
  - When load factor exceeds particular value (appr. 0.7), hash table performance will decrease nonlinearly.
  - Also, the number of stored key-value pairs is limited to the size of the table (16).

Example 1

- Put "ALGORITHMS" into the Hash Table while the key of 'A' is 65, 'B' is 66, … etc.

Value	Key
A	65
L	76
G	71
O	79
R	82
I	73
T	84
H	72
M	77
S	83

Example 1

- Put 'A' into the Hash Table
  - key%16 = 65%16 = 1
  - 'A' is put at index 1

Index	Key	Value
0		
1	65	A
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Example 1

- Put 'L' into the Hash Table
  - key%16 = 76%16 = 12
  - 'L' is put at index 12

Index	Key	Value
0		
1	65	A
2		
3		
4		
5		
6		
7		
8		
9		
10		
11		
12	76	L
13		
14		
15		

Example 1

- And so on…
  - ‘G’ at 7
  - ‘O’ at 15
  - ‘R’ at 2
  - ‘I’ at 9
  - ‘T’ at 4
  - ‘H’ at 8
  - ‘M’ at 13
  - ‘S’ at 3

Index	Key	Value
0		
1	65	A
2	82	R
3	83	S
4	84	T
5		
6		
7	71	G
8	72	H
9	73	I
10		
11		
12	76	L
13	77	M
14		
15	79	O

Example 2

- Put "COMPUTERS" into the Hash Table while the key of 'A' is 65, 'B' is 66, … etc.

Value	Key
C	67
O	79
M	77
P	80
U	85
T	84
E	69
R	82
S	83

Example 2

- Put 'C' into the Hash Table
  - key%16 = 67%16 = 3
  - 'C' is put at index 3

Index	Key	Value
0		
1		
2		
3	67	C
4		
5		
6		
7		
8		
9		
10		
11		
12		
13		
14		
15		

Example 2

- And so on…
  - ‘O’ @ 15
  - ‘M’ @ 13
  - ‘P’ @ 0
  - ‘U’ @ 5
  - ‘T’ @ 4

Index	Key	Value
0	80	P
1		
2		
3	67	C
4	84	T
5	85	U
6		
7		
8		
9		
10		
11		
12		
13	77	M
14		
15	79	O

Example 2

- When put 'E' into the Hash Table
  - key@16 = 69%16 = 5
  - While 5 is occupied by 'U' already, (Collision occurred)
  - So, 'E' is put at 6 (Linear probing)

Index	Key	Value
0	80	P
1		
2	82	R
3	67	C
4	84	T
5	85	U
6	69	E
7		
8		
9		
10		
11		
12		
13	77	M
14		
15	79	O

Example 2

- When put 'S' into the Hash Table
  - key@16 = 83%16 = 3
  - While 3 is occupied by 'C' already, (Collision occurred)
  - 4, 5, 6 are also occupied
  - So, 'S' is put at 7 (Linear probing)

Index	Key	Value
0	80	P
1		
2	82	R
3	67	C
4	84	T
5	85	U
6	69	E
7	83	S
8		
9		
10		
11		
12		
13	77	M
14		
15	79	O

Implementation

- This implementation suffers one bug.
  - When there is no more place in the table, the loop, searching for empty slot, will run infinitely.
  - It won't happen in real hash table based on open addressing, because it is most likely dynamic-sized.
- Also the removal's implementation is omitted to maintain simplicity.

Implementation

```
1.  class HashEntry {
2.  private:
3.      int key;
4.      int value;
5.  public:
6.      HashEntry(int key, int value) {
7.          this->key = key;
8.          this->value = value;
9.      }
10.     int getKey() {
11.         return key;
12.     }
13.     int getValue() {
14.         return value;
15.     }
16. };
```

Implementation

```
17. const int TABLE_SIZE = 16;
18.
19. class HashMap {
20. private:
21.     HashEntry **table;
22. public:
23.     HashMap() {
24.         table = new HashEntry*[TABLE_SIZE];
25.         for (int i = 0; i < TABLE_SIZE; i++)
26.             table[i] = NULL;
27.     }
```

Implementation

```
28.     int get(int key) {
29.         int hash = (key % TABLE_SIZE);
30.         while (table[hash] != NULL &&
                        table[hash]->getKey() != key)
31.             hash = (hash + 1) % TABLE_SIZE;
32.         if (table[hash] == NULL)
33.             return -1;
34.         else
35.             return table[hash]->getValue();
36.     }
```

Implementation

```
37.     void put(int key, int value) {
38.         int hash = (key % TABLE_SIZE);
39.         while (table[hash] != NULL &&
                        table[hash]->getKey() != key)
40.             hash = (hash + 1) % TABLE_SIZE;
41.         if (table[hash] != NULL)
42.             delete table[hash];
43.         table[hash] = new HashEntry(key, value);
44.     }
```

Implementation

```
45.     ~HashMap() {
46.         for (int i = 0; i < TABLE_SIZE; i++)
47.             if (table[i] != NULL)
48.                 delete table[i];
49.         delete[] table;
50.     }
51. };
```

Hash Functions

Hash Functions

- The number of hash functions that can be used to assign positions to n items in a table of m positions (for n <= m) is equal to m^n.
  - Most of these functions are too unwieldy for practical applications and cannot be represented by a concise formula.
- This section discusses some specific types of hash functions.

Division

- A hash function must guarantee that the number it returns is a valid index to one of the table cells.
- The simplest way to accomplish this is to use division modulo
  - TSize = sizeof(table), as in
  - h(K) = K mod TSize, if K is a number.

Folding

- In this method, the key is divided into several parts.
- These parts are combined or folded together and are often transformed in a certain way to create the target address.
- There are two types of folding:
  - Shift folding
  - Boundary folding

Folding

- The key is divided into several parts and these parts are then processed using a simple operation such as addition to combine them in a certain way.

- In shift folding, they are put underneath one another and then processed.

Folding

- For example, a social security number (SSN) 123-45-6789 can be divided into three parts, 123, 456, 789, and then these parts can be added.

- The resulting number, 1,368, can be divided modulo TSize or, if the size of the table is 1,000, the first three digits can be used for the address.

Folding

- With boundary folding, the key is seen as being written on a piece of paper that is folded on the borders between different parts of the key.

- In this way, every other part will be put in the reverse order.

Folding

- Consider the same three parts of the SSN: 123, 456 and 789.
  - The first part, 123, is taken in the same order,
  - then the piece of paper with the second part is folded underneath it so that 123 is aligned with 654, which is the second part, 456, in reverse order.
  - When the folding continues, 789 is aligned with the two previous parts. The result is 123 + 654 +789 = 1,566.

Folding

- In both versions, the key is usually divided into even parts of some fixed size plus some remainder and then added.
- This process is simple and fast, especially when bit patterns are used instead of numerical values.
  - A bit-oriented version of shift folding is obtained by applying the exclusive- or operation, ^.

Mid-square Method

- In the mid-square method, the key is squared and the middle or mid part of the result is used as the address.

- If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding.

Mid-square Method

- In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys.

- For example,

-
  - if the key is 3,121, then $3{,}121^2 = 9{,}740{,}641$,
  - and for the 1,000-cell table, h(3,121) = 406, which is the middle part of $3{,}121^2$.

Extraction

- In the extraction method, only a part of the key is used to compute the address.
- For the social security number 123-45-6789, this method might use
  - the first four digits, 1234;
  - the last four, 6,789;
  - the first two combined with the last two, 1,289; or
  - some other combination.

Extraction

- Each time, only a portion of the key is used.

- If this portion is carefully chosen, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in an insignificant way.

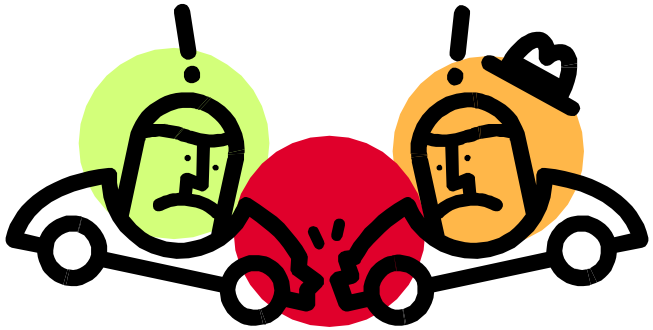
Radix transformation

- Using the radix transformation, the key K is transformed into another number base;
- K is expressed in a numerical system using a different radix.
- If K is the decimal number 345, then its value in base 9 (nonal) is 423. This value is then divided modulo TSize, and the resulting number is used as the address of the location to which K should be hashed.

Radix transformation

- Collisions, however, cannot be avoided.
  - If TSize = 100,
    - then although 345 and 245 (decimal) are not hashed to the same location,
    - 345 and 264 are because 264 decimal is 323 in the nonal system, and both 423 and 323 return 23 when divided modulo 100.

Handling the collisions



Handling the collisions

- In the small number of cases, where multiple keys map to the same integer, then elements with different keys may be stored in the same "slot" of the hash table.

- It is clear that when the hash function is used to locate a potential match, it will be necessary to compare the key of that element with the search key.

Handling the collisions

- But there may be more than one element which should be stored in a single slot of the table.

- Various techniques are used to manage this problem:
  - chaining,
  - re-hashing,
  - using neighboring slots (linear probing),
  - quadratic probing,
  - random probing,
  - overflow areas, ...

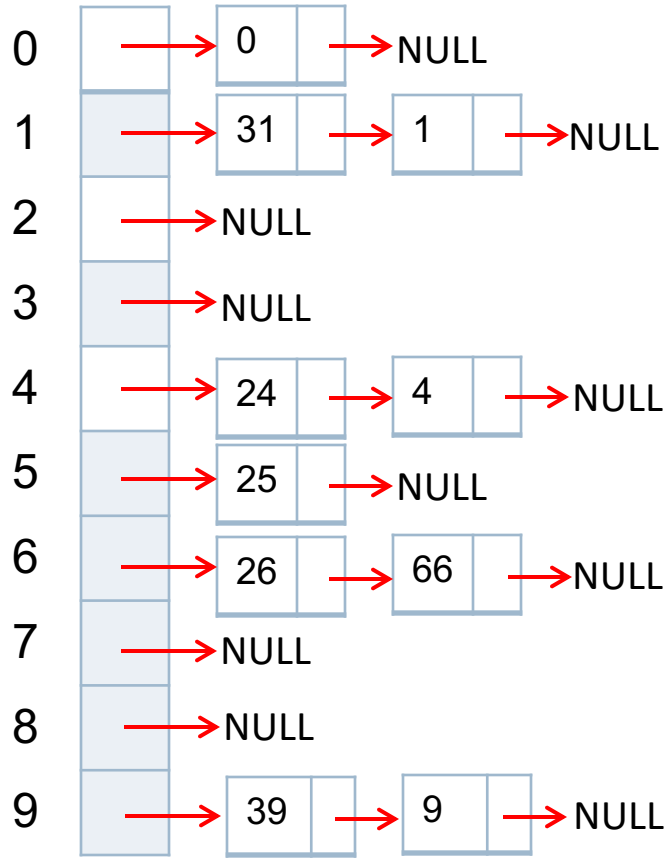
Chaining

- One simple scheme is to chain all collisions in lists attached to the appropriate slot.
- This allows an unlimited number of collisions to be handled and doesn't require *a priori* knowledge of how many elements are contained in the collection.
- The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.

Chaining

- To insert key k to hash table — Worst case: O(1)
  - Compute h(k) to determine where to insert the element
  - If T[h(k)] is NULL, make this table cell to point to a node contains k
  - Otherwise, add a node contains k to the beginning of the list
- To search for a key k — Worst case: O(n)
  - Compute h(k) and search within the list at T[h(k)]
- To delete a key k from the hash table T — Worst case: O(n)
  - Compute h(k) to determine where to remove the element
  - Search within the list at T[h(k)] and delete the node contains k if it is found

Chaining



Insert 9, 25, 66, 4, 24, 1, 31, 0, 9, 39
in sequence

Hash Function:
h(k) = k mod 10

Chaining

- Pros:
  - The number of keys in each linked list is a small constant
    (Assume n keys will be stored and m is chosen as the next larger prime number) and this facilitates constant time, i.e. O(1) for searching, insertion and deletion of elements on average
  - Deletion is easy

- Cons:
  - More space is needed as linked list is used
  - Memory allocation of node and manipulation of pointers slow down the program

Open Addressing / Probing

- Open addressing or probing refers to finding other available place if collision occurs

- Hash function of open addressing is as follows:
  - $h(k, i) = (h_i(k) + f(i)) \bmod size$,
    where f(i) is the collision resolution function
  - Typically f(0) = 0

- Hash functions for different open addressing schemes:
  - Linear probing: $f(i) = i$
  - Quadratic probing: $f(i) = i^2$
  - Re-hashing: $f(i) = i * h_2(k)$

Open Addressing / Probing

- To insert key k to hash table
  - Probe hash table until an empty slot is found
- To search for a key k
  - Probe hash table until the key is found or confirmed that it is not found
- To delete a key k from the hash table T
  - Problem:
    - If the key is deleted, but there are keys that hash to the same location stored in other locations in the table, then searches for those keys will be treated as unsuccessful
  - Must be "lazy" delete
    - Keep the key in the table, but mark it as deleted
    - New key will overwrite the location marked as deleted

Linear Probing

- One of the simplest re-hashing functions is
  +1 (or -1), *ie* on a collision,
  look in the neighboring slot in the table.

- It calculates the new address extremely quickly and may be extremely efficient.

Linear Probing

- **Clustering**
  - Linear probing is subject to a **clustering** phenomenon.
  - Re-hashes from one location occupy a block of slots in the table which "grows" towards slots to which other keys hash.
  - This exacerbates the collision problem and the number of re-hashed can become large.

Linear Probing – Example

	Initial	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Insert 89, 18, 49, 58, 69
in sequence

$h(k, i) = (h(k) + i) \bmod 10$
$h(k) = k \bmod 10$

Probe sequence:

- 89: 9
- 18: 8
- 49: 9 → (9+1) mod 10 = 0
- 58: 8 → (8+1) mod 10 = 9
  → (8+2) mod 10 = 0
  → (8+3) mod 10 = 1
- 69: 9 → (9+1) mod 10 = 0
  → (9+2) mod 10 = 1
  → (9+3) mod 10 = 2

Linear Probing

- Pros:
  - Easy to implement
  - Use less memory than separate chaining
  - Fast when table is sparse

- Cons:
  - Hash table has fixed size
  - Likely with block of contiguously occupied entries, i.e. a cluster (we call this primary clustering) and this causes bad performance since
    - It increases the chance of having collision for the next insertion
    - It increases the searching time of elements

Quadratic Probing

	Initial	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Insert 89, 18, 49, 58, 69
in sequence

$h(k, i) = (h(k) + i^2) \bmod 10$
$h(k) = k \bmod 10$

Probe sequence:

- 89: 9
- 18: 8
- 49: 9 → $(9+1^2) \bmod 10 = 0$
- 58: 8 → $(8+1^2) \bmod 10 = 9$
  → $(8+2^2) \bmod 10 = 2$
- 69: 9 → $(9+1^2) \bmod 10 = 0$
  → $(9+2^2) \bmod 10 = 3$

Quadratic Probing

- Pros:
  - Easy to implement
- Cons:
  - Keys that hash to the same initial location will probe the same alterative cells and this causes clustering around the probe sequences (we call this second clustering)

Re-hashing

- Re-hashing schemes use a second hashing operation when there is a collision.
- If there is a further collision, we *re-hash* until an empty "slot" in the table is found.
- The re-hashing function can either be a new function or a re-application of the original one.
- As long as the functions are applied to a key in the same order, then a sought key can always be located.

Re-hashing

	Initial	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Insert 89, 18, 49, 58, 69
in sequence

$h(k, i) = (h(k) + ih_2(k)) \bmod 10$
$h(k) = k \bmod 10$
$h_2(k) = 7 - (k \bmod 7)$

Probe sequence:

- 89: 9
- 18: 8
- 49: 9 → $(9 + i*(7 - 49 \bmod 7)) \bmod 10 = 6$
- 58: 8 → $(8 + i*(7 - 58 \bmod 7)) \bmod 10 = 3$
- 69: 9 → $(9 + i*(7 - 69 \bmod 7)) \bmod 10 = 0$

Re-hashing

- How to choose second hash function?
  - Shouldn't evaluate to zero
  - Relatively prime to the size of table .
    Otherwise, only a fraction of table entries will be examined
- Pros:
  - Eliminate secondary clustering
- Cons:
  - Time consuming to compute two hash functions

Overflow Area

- Another scheme will divide the pre-allocated table into two sections:
  - the *primary area* to which keys are mapped and
  - an area for collisions, normally termed the *overflow area*

Overflow Area

- When a collision occurs, a slot in the overflow area is used for the new element and a link from the primary slot established as in a chained system.
  - This is essentially the same as chaining, except that the overflow area is pre-allocated and thus possibly faster to access.
  - As with re-hashing, the maximum number of elements must be known in advance, but in this case, two parameters must be estimated: the optimum size of the primary and overflow areas.

SUMMARY

Organization	Advantages	Disadvantages
Chaining	∘Unlimited number of elements ∘Unlimited number of collisions	∘Overhead of multiple linked lists
Re-hashing	∘Fast re-hashing ∘Fast access through use of main table space	∘Maximum number of elements must be known ∘Multiple collisions may become probable
Overflow area	∘Fast access ∘Collisions don't use primary table space	∘Two parameters which govern performance need to be estimated

CHAPTER 11 END