

CE2001/ CZ2001: Algorithms

Searching

Dr. Loke Yuan Ren

A Generic Search Routine

```
while (there is more possible data to examine){  
    examine one datum;  
    if (this datum == search key)  
        return succeed;  
    }  
return fail;
```

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the basic principle of searching
- Apply the concept of sequential search algorithm using working examples
- Analyse the time complexity of sequential search algorithm

A Generic Search Routine

```
while (there is more possible data to examine){  
    examine one datum;  
    if (this datum == search key)  
        return succeed;  
    }  
return fail;
```

A Generic Search Routine

```
while (there is more possible data to examine){  
    examine one datum;  
    if (this datum == search key)  
        return succeed;  
    }  
return fail;
```

5

A Generic Search Routine

```
while (there is more possible data to examine){  
    examine one datum;  
    if (this data == search key)  
        return succeed;  
    }  
return fail;
```

6

Sequential Search (Unordered Array)

Sequential Search (Unordered Array)

```
int seqSearch (int [] Data, int n, int key)  
{  
    int index;  
  
    for (index = 0; index < n; index++)  
        if (key == Data[index])  
            return index; // succeed  
    return -1; // fail  
}
```

8

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
    int index;

    for (index = 0; index < n; index++)
        if (key == Data[index])
            return index; // succeed
    return -1; // fail
}
```

9

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
    int index;

    for (index = 0; index < n; index++)
        if (key == Data[index])
            return index; // succeed
    return -1; // fail
}
```

11

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
    int index;

    for (index = 0; index < n; index++)
        if (key == Data[index])
            return index; // succeed
    return -1; // fail
}
```

10

Sequential Search (Unordered Array)

```
int seqSearch (int [ ] Data, int n, int key)
{
    int index;

    for (index = 0; index < n; index++)
        if (key == Data[index])
            return index; // succeed
    return -1; // fail
}
```

12

Successful Search

Target Given	14
Location Wanted	3



13

Successful Search

Target Given	14
Location Wanted	3



14

Successful Search

Target Given	14
Location Wanted	3



15

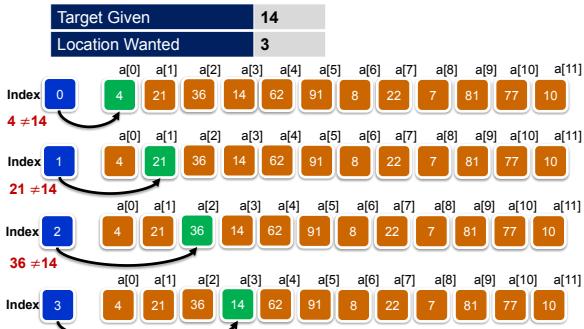
Successful Search

Target Given	14
Location Wanted	3



16

Successful Search



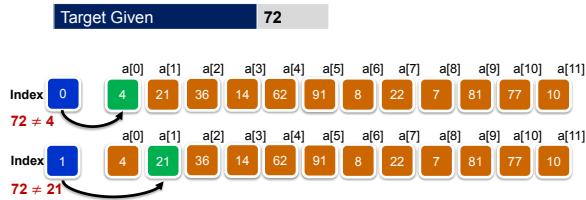
Unsuccessful Search



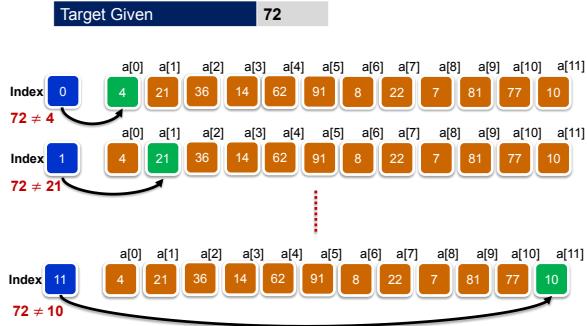
Unsuccessful Search



Unsuccessful Search



Unsuccessful Search



21

Complexity of Sequential Search

- Best-case complexity:** 1 comparison against key
- Worst-case complexity:** n comparisons against key
- Average-case complexity:**

- Consider the situation that the **key** is found in array
- e_i represents the event that the key appears in i^{th} position of array, so its probability $P(e_i) = 1/n$
- $T(e_i)$ is the no. of comparisons done
- In this case, we have $0 \leq i < n$, $T(e_i) = i + 1$
- If key is in array, Avg. complexity $A_s(n)$:

$$= \sum_{i=0}^{n-1} P(e_i)T(e_i) = \sum_{i=0}^{n-1} \left(\frac{1}{n}\right)(i+1) = \frac{1}{n} \sum_{i=1}^n i = \frac{n+1}{2}$$

formula for arithmetic

$$S_n = \frac{n}{2} (2a + (n-1)d)$$

$$= \frac{n}{2} (a + L)$$

if element to be found at pos 2 no. of comparison

$i+1$

i

\vdots

n

Find no. of comparisons upto n

23

$$= \frac{1}{n} \sum_{i=0}^{n-1} (i+1) = \frac{1}{n} [1+2+3+\dots+n]$$

$$= \frac{1}{n} \times \frac{n(n+1)}{2}$$

Complexity of Sequential Search

- On the other hand, if **key** is not in array, number of comparisons $A_f(n)$ is n.
- Combining the above:
 - $\Pr(\text{succ}) A_s(n) + \Pr(\text{fail}) A_f(n) = q * (n+1)/2 + (1-q) * n$
 - If there is a 50-50 chance that key is not in the array, (i.e. $q = 1/2$),
 - Then **average no. of key comparisons** is $3n/4 + 1/4$ (about 3/4 of entries examined)
- Both **worst** and **average complexity** are $\Theta(n)$.

You have the array

• 50% chance found by half-fit

• 50% chance search 1/2

$$= \frac{1}{2} \frac{(n+1)}{2} + (1 - \frac{1}{2}) \frac{n}{2}$$

24

Recap

- A generic search routine ↗
- Sequential search
 - Examine data item sequentially
 - Average and worst time complexities are both $\theta(n)$

25

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the Binary Search Algorithm
- Apply the binary search algorithm using working examples
- Analyse worst-case time complexity
- Analyse average-case time complexity

27

Binary Search

Binary Search

- Applicable when the array is **ordered**.
- This search method **uses the information of the order** of the elements and tries to do less work.
- This is another example of the **divide and conquer** approach.
- Here a problem is **divided into 2 smaller sub-problems**, one of which does not even have to be solved.

28

Binary Search



```
int binarySearch (int [] E, int first, int last, int k) // k: Item to search
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

29

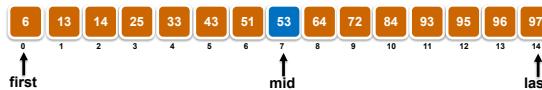
Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

31

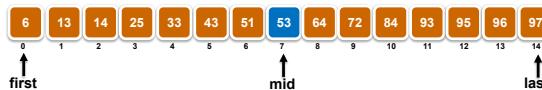
Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

30

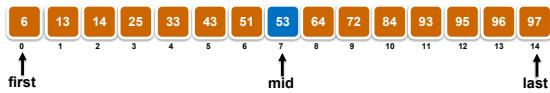
Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

32

Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
```

```
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

33

Binary Search

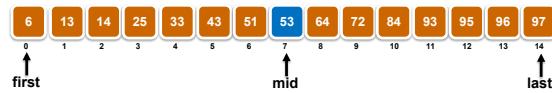


```
int binarySearch (int [] E, int first, int last, int k)
```

```
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

35

Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
```

```
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

34

Binary Search



```
int binarySearch (int [] E, int first, int last, int k)
```

```
{
    if (last < first)
        return -1;
    else {
        int mid = (first + last) / 2;
        if (k == E[mid]) return mid;
        else if (k < E[mid])
            return binarySearch (E, first, mid - 1, k);
        else
            return binarySearch (E, mid + 1, last, k);
    }
}
```

} Non-recursive version possible

36

Binary Search (Example 1)

Binary Search (Example 1)

Binary search for target 33



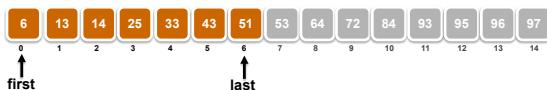
Binary Search (Example 1)

Binary search for target 33



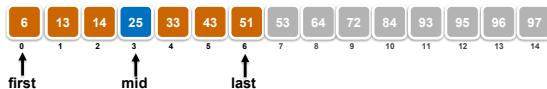
Binary Search (Example 1)

Binary search for target 33



Binary Search (Example 1)

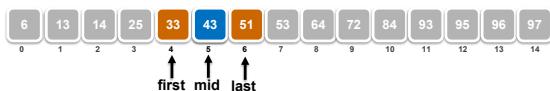
Binary search for target 33



41

Binary Search (Example 1)

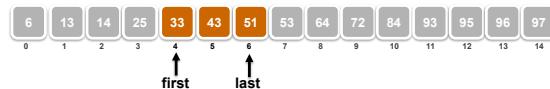
Binary search for target 33



43

Binary Search (Example 1)

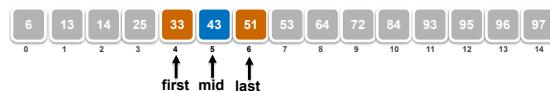
Binary search for target 33



42

Binary Search (Example 1)

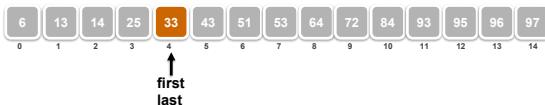
Binary search for target 33



44

Binary Search (Example 1)

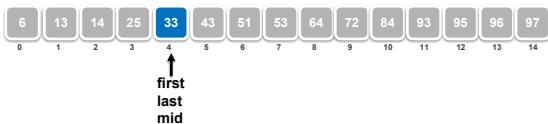
Binary search for target 33



45

Binary Search (Example 1)

Binary search for target 33

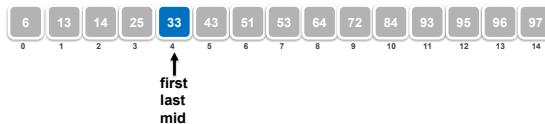


Successfully found 33!

47

Binary Search (Example 1)

Binary search for target 33

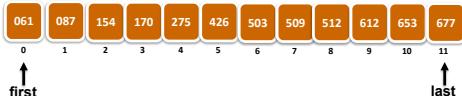


46

Binary Search (Example 3)

Example: Searching for 400

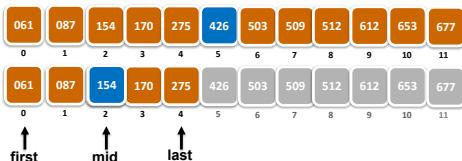
Binary search for target 400



55

Example: Searching for 400

Binary search for target 400



57

Example: Searching for 400

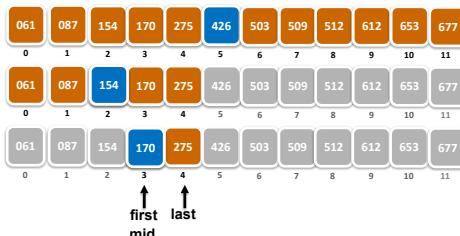
Binary search for target 400



56

Example: Searching for 400

Binary search for target 400



58

Example: Searching for 400

Binary search for target 400

061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11

first
last
mid

59

Example: Searching for 400

Binary search for target 400

061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11
061	087	154	170	275	426	503	509	512	612	653	677
0	1	2	3	4	5	6	7	8	9	10	11

last < first Return -1

Search fails!

60

Best case:
When search key is right at the middle.

Worst-Case Time Complexity

Worst-Case Time Complexity

int binarySearch (int[] E, int first, int last, int k)

T(n)

← time complexity

{

if (last < first) return -1;

else {

 int mid = (first + last) / 2;

 if (k == E[mid]) return mid;

 else if (k < E[mid])

 return binarySearch(E, first, mid - 1, k);

 else return binarySearch(E, mid + 1, last, k);

}

} So, T(n) = c + T(n/2)

Constant c

Adjusted

$T(\frac{n-1}{2})$

$T(\frac{n}{2})$

T(n/2)

Worst Case is failure case T(0) = 0

every recursive call → constant time, c + A recursive call.

62

Worst-Case Time Complexity

Solve recurrence equation $T(n) = T(n/2) + c$

$$\begin{aligned} T(n) &= T(n/2) + c \\ &= T(n/4) + 2c \\ &= T(n/8) + 3c \\ &= \dots \\ &= T(1) + dc \\ &= T(0) + (d+1)c \end{aligned}$$

$$\left. \begin{aligned} T(1) &= T(0) + c \\ T(0) &= 0 \end{aligned} \right\} \Rightarrow$$

$$\begin{aligned} n &\rightarrow n/2 \rightarrow n/2^2 \rightarrow n/2^3 \rightarrow \dots \rightarrow n/2^d \\ n/2^d &\geq 1 \Rightarrow d \leq \log_2 n \\ &= \lfloor \log_2 n \rfloor \end{aligned}$$

Suppose to have one more recursive call for the last step doesn't affect much

$d = \max$ times to divide n by 2 before going below 1

So, worst-case running time $T(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$

$$\begin{aligned} \frac{n}{2^d} &> 1 \\ \text{Not equal to } 1 &\\ \text{because eq. } &\\ n=7 & 2^3 \\ n=3 & 2^2 \\ n=1, 2 & 2^1 \\ n=1, 2, 2^2 & (\text{Stop here}) \\ & x1 \end{aligned}$$

63

Average Number of Comparisons

- Law of Expectations:
- $A_q(n) = q A_s(n) + (1 - q) A_f(n)$,
- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search
- For simplicity, let $n=2^k-1$,
 - $A_f(n)$ is the worst case discussed in slide 63: $\lceil \log_2(n+1) \rceil$
 - $A_s(n)$?

For example, $n = 2^3 - 1 = 7$ entries



↑
1 comparison

65

n must be an integer.

Prove $\lfloor \log_2 n \rfloor + 1 = \lceil \log_2(n+1) \rceil$

- Let k be some integer number that $k = \lfloor \log_2 n \rfloor$, then

$$k \leq \log_2 n < k + 1$$

$$2^k \leq n < 2^{k+1}$$

$$2^k < n + 1 \leq 2^{k+1}$$

$$k < \log_2(n+1) \leq k + 1$$

Eg. $k=4$,
 $2^4 \leq n < 2^5$
 $2^4 < n + 1 \leq 2^5$
 n is integer

$$\lceil \log_2(n+1) \rceil = k + 1$$

$$\lceil \log_2(n+1) \rceil = \lfloor \log_2 n \rfloor + 1$$

64

Average Number of Comparisons

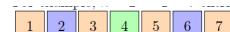
For example, $n = 2^3 - 1 = 7$ entries



↑
1 comparison



↑
2 comparisons



↑
2 comparisons



↑
3 comparisons



↑
3 comparisons

↑
3 comparisons

↑
3 comparisons

We can observe that:

- 1 position requires 1 comparison
- 2 positions require 2 comparisons
- 4 positions require 3 comparisons
- ...
- 2^{t-1} positions require t comparisons

$n=2^k-1$, we have

$$\begin{aligned} n+1 &= 2^k \\ A_s(n) &= \sum_{i=1}^k i 2^{i-1} \\ K &= \log_2(n+1) \end{aligned}$$

Assume each position get equal prob

$$= \frac{(k-1)2^k + 1}{n}$$

$$= \frac{\lfloor \log_2(n+1) - 1 \rfloor (n+1) + 1}{n}$$

$$= \log_2(n+1) - 1 + \frac{\log_2 n}{n}$$

66

Average Number of Comparisons

- The average complexity is

$$\begin{aligned}
 A_q(n) &= qA_s(n) + (1-q)A_f(n) \\
 &= q[\log_2(n+1) - 1 + \frac{\log_2 n}{n}] + (1-q)(\log_2(n+1)) \\
 &= \log_2(n+1) - q + q\frac{\log_2 n}{n} \\
 &= \Theta(\log_2(n))
 \end{aligned}$$

- Binary search does approximately $\log_2(n+1)$ comparisons on average for n entries.

- q is probability which is always ≤ 1
- $\frac{\log_2 n}{n}$ is very small especially when $n \gg 1$

67

Recap

- Binary Search

- Divide and conquer strategy
- Examine data item in the middle and search recursively on single half
- Average and worst time complexities are both $\Theta(\log(n))$

70

Learning Objectives

At the end of this lecture, students should be able to:

- Explain the concept of Hashing
- Explain the requirement of Hash Functions
- Use different hashing functions

72

CE2001/ CZ2001: Algorithms

Hashing

Dr. Loke Yuan Ren

Direct-Address Table

- Idea: Suppose that the set of actual keys is $K \subseteq \{0, 1, \dots, m-1\}$ and keys are distinct.

Set up an array $T[0 .. m-1]$:

$$T[x, k] = \begin{cases} x, & \text{if } k \in K \wedge x.k = k; \\ NIL, & \text{otherwise.} \end{cases}$$

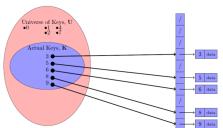


Figure 4.1: Direct Address Table

- Then, operations take $O(1)$ time

- Issue:** The range of keys can be large ($m \gg |K|$), e.g. 64-bit numbers (range $m \approx 18.45 \times 10^{18}$)

73

Hashing (Recap)

Motivation: To be able to assign a unique array index to every possible key that could occur in an application.

- Key space may be too *large* for an array on the computer while only a *small* fraction of the key values will appear.
- The purpose of hashing is to translate an extremely large key space into a reasonably small set of integers.
- A hash function f : key space \rightarrow hash codes

75

Hash Table

- A **hash table** H is an array on indexes $0 .. m-1$, each entry in the array is called a **hash slot**.

Example: a hash table of 200 entries.

A possible hash function:

$$f(k) = k \bmod 200$$

Note: Not a good hash function.

When multiple keys are mapped to the same hash code, a **collision** occurs.

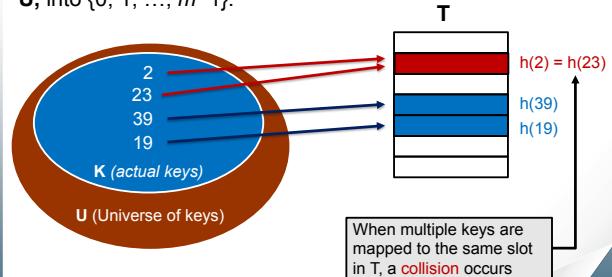
Two main issues in designing a hash table:

- (1) What hash function to use?
- (2) How to handle collisions?

76

Direct-Address Table (Solution)

Use a **hash function** h to map the all keys of the universe, U , into $\{0, 1, \dots, m-1\}$.



74

Hash Functions

- A hash function **MUST** return a value within the hash table range.
- It should achieve an even distribution of the keys that actually occur across the range of indices
- It should be easy and quick to compute

Tips

- If we know nothing about the incoming key distribution, evenly distribute the key range over the hash table slots while avoiding obvious opportunities for clustering.
- If we have knowledge of the incoming distribution, use a distribution-dependant hash function.

77

Hash Function (Example 2)

```
int h(char x[10])
{
    int i, sum;
    for (sum=0, i=0; i < 10; i++)
        sum += (int) x[i];
    return(sum % M);
}
```

Only good if the sum is large compared to M

iii. Multiplicative Congruential method (pseudo-number generator):

Step 1: Choose the hash table size, h .

Step 2: Choose the multiplier, a

$$a = 8 \lfloor h/23 \rfloor + 5$$

Step 3: Define the hash function, f

$$f(k) = (a * k) \bmod h$$

```

1. h = 20
2. a = 8 * floor(h/23) + 5
3. a = 13
4. k = 1..15
5. k = 13
6. f = 0
7. for k = 1 to 15
8.   f = (a * k) % h
9.   print(f)
10.  k = k + 1
11. endfor
12. print(f)
13. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
14. f = 0
15. k = 13
16. for k = 1 to 15
17.   f = (a * k) % h
18.   print(f)
19.   k = k + 1
20. endfor
21. print(f)
22. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
23. f = 0
24. k = 13
25. for k = 1 to 15
26.   f = (a * k) % h
27.   print(f)
28.   k = k + 1
29. endfor
30. print(f)
31. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
32. f = 0
33. k = 13
34. for k = 1 to 15
35.   f = (a * k) % h
36.   print(f)
37.   k = k + 1
38. endfor
39. print(f)
40. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
41. f = 0
42. k = 13
43. for k = 1 to 15
44.   f = (a * k) % h
45.   print(f)
46.   k = k + 1
47. endfor
48. print(f)
49. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
50. f = 0
51. k = 13
52. for k = 1 to 15
53.   f = (a * k) % h
54.   print(f)
55.   k = k + 1
56. endfor
57. print(f)
58. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
59. f = 0
60. k = 13
61. for k = 1 to 15
62.   f = (a * k) % h
63.   print(f)
64.   k = k + 1
65. endfor
66. print(f)
67. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
68. f = 0
69. k = 13
70. for k = 1 to 15
71.   f = (a * k) % h
72.   print(f)
73.   k = k + 1
74. endfor
75. print(f)
76. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
77. f = 0
78. k = 13
79. for k = 1 to 15
80.   f = (a * k) % h
81.   print(f)
82.   k = k + 1
83. endfor
84. print(f)
85. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
86. f = 0
87. k = 13
88. for k = 1 to 15
89.   f = (a * k) % h
90.   print(f)
91.   k = k + 1
92. endfor
93. print(f)
94. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
95. f = 0
96. k = 13
97. for k = 1 to 15
98.   f = (a * k) % h
99.   print(f)
100.  k = k + 1
101. endfor
102. print(f)
103. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
104. f = 0
105. k = 13
106. for k = 1 to 15
107.   f = (a * k) % h
108.   print(f)
109.   k = k + 1
110. endfor
111. print(f)
112. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
113. f = 0
114. k = 13
115. for k = 1 to 15
116.   f = (a * k) % h
117.   print(f)
118.   k = k + 1
119. endfor
120. print(f)
121. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
122. f = 0
123. k = 13
124. for k = 1 to 15
125.   f = (a * k) % h
126.   print(f)
127.   k = k + 1
128. endfor
129. print(f)
130. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
131. f = 0
132. k = 13
133. for k = 1 to 15
134.   f = (a * k) % h
135.   print(f)
136.   k = k + 1
137. endfor
138. print(f)
139. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
140. f = 0
141. k = 13
142. for k = 1 to 15
143.   f = (a * k) % h
144.   print(f)
145.   k = k + 1
146. endfor
147. print(f)
148. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
149. f = 0
150. k = 13
151. for k = 1 to 15
152.   f = (a * k) % h
153.   print(f)
154.   k = k + 1
155. endfor
156. print(f)
157. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
158. f = 0
159. k = 13
160. for k = 1 to 15
161.   f = (a * k) % h
162.   print(f)
163.   k = k + 1
164. endfor
165. print(f)
166. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
167. f = 0
168. k = 13
169. for k = 1 to 15
170.   f = (a * k) % h
171.   print(f)
172.   k = k + 1
173. endfor
174. print(f)
175. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
176. f = 0
177. k = 13
178. for k = 1 to 15
179.   f = (a * k) % h
180.   print(f)
181.   k = k + 1
182. endfor
183. print(f)
184. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
185. f = 0
186. k = 13
187. for k = 1 to 15
188.   f = (a * k) % h
189.   print(f)
190.   k = k + 1
191. endfor
192. print(f)
193. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
194. f = 0
195. k = 13
196. for k = 1 to 15
197.   f = (a * k) % h
198.   print(f)
199.   k = k + 1
200. endfor
201. print(f)
202. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
203. f = 0
204. k = 13
205. for k = 1 to 15
206.   f = (a * k) % h
207.   print(f)
208.   k = k + 1
209. endfor
210. print(f)
211. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
212. f = 0
213. k = 13
214. for k = 1 to 15
215.   f = (a * k) % h
216.   print(f)
217.   k = k + 1
218. endfor
219. print(f)
220. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
221. f = 0
222. k = 13
223. for k = 1 to 15
224.   f = (a * k) % h
225.   print(f)
226.   k = k + 1
227. endfor
228. print(f)
229. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
230. f = 0
231. k = 13
232. for k = 1 to 15
233.   f = (a * k) % h
234.   print(f)
235.   k = k + 1
236. endfor
237. print(f)
238. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
239. f = 0
240. k = 13
241. for k = 1 to 15
242.   f = (a * k) % h
243.   print(f)
244.   k = k + 1
245. endfor
246. print(f)
247. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
248. f = 0
249. k = 13
250. for k = 1 to 15
251.   f = (a * k) % h
252.   print(f)
253.   k = k + 1
254. endfor
255. print(f)
256. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
257. f = 0
258. k = 13
259. for k = 1 to 15
260.   f = (a * k) % h
261.   print(f)
262.   k = k + 1
263. endfor
264. print(f)
265. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
266. f = 0
267. k = 13
268. for k = 1 to 15
269.   f = (a * k) % h
270.   print(f)
271.   k = k + 1
272. endfor
273. print(f)
274. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
275. f = 0
276. k = 13
277. for k = 1 to 15
278.   f = (a * k) % h
279.   print(f)
280.   k = k + 1
281. endfor
282. print(f)
283. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
284. f = 0
285. k = 13
286. for k = 1 to 15
287.   f = (a * k) % h
288.   print(f)
289.   k = k + 1
290. endfor
291. print(f)
292. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
293. f = 0
294. k = 13
295. for k = 1 to 15
296.   f = (a * k) % h
297.   print(f)
298.   k = k + 1
299. endfor
300. print(f)
301. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
302. f = 0
303. k = 13
304. for k = 1 to 15
305.   f = (a * k) % h
306.   print(f)
307.   k = k + 1
308. endfor
309. print(f)
310. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
311. f = 0
312. k = 13
313. for k = 1 to 15
314.   f = (a * k) % h
315.   print(f)
316.   k = k + 1
317. endfor
318. print(f)
319. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
320. f = 0
321. k = 13
322. for k = 1 to 15
323.   f = (a * k) % h
324.   print(f)
325.   k = k + 1
326. endfor
327. print(f)
328. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
329. f = 0
330. k = 13
331. for k = 1 to 15
332.   f = (a * k) % h
333.   print(f)
334.   k = k + 1
335. endfor
336. print(f)
337. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
338. f = 0
339. k = 13
340. for k = 1 to 15
341.   f = (a * k) % h
342.   print(f)
343.   k = k + 1
344. endfor
345. print(f)
346. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
347. f = 0
348. k = 13
349. for k = 1 to 15
350.   f = (a * k) % h
351.   print(f)
352.   k = k + 1
353. endfor
354. print(f)
355. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
356. f = 0
357. k = 13
358. for k = 1 to 15
359.   f = (a * k) % h
360.   print(f)
361.   k = k + 1
362. endfor
363. print(f)
364. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
365. f = 0
366. k = 13
367. for k = 1 to 15
368.   f = (a * k) % h
369.   print(f)
370.   k = k + 1
371. endfor
372. print(f)
373. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
374. f = 0
375. k = 13
376. for k = 1 to 15
377.   f = (a * k) % h
378.   print(f)
379.   k = k + 1
380. endfor
381. print(f)
382. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
383. f = 0
384. k = 13
385. for k = 1 to 15
386.   f = (a * k) % h
387.   print(f)
388.   k = k + 1
389. endfor
390. print(f)
391. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
392. f = 0
393. k = 13
394. for k = 1 to 15
395.   f = (a * k) % h
396.   print(f)
397.   k = k + 1
398. endfor
399. print(f)
400. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
401. f = 0
402. k = 13
403. for k = 1 to 15
404.   f = (a * k) % h
405.   print(f)
406.   k = k + 1
407. endfor
408. print(f)
409. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
410. f = 0
411. k = 13
412. for k = 1 to 15
413.   f = (a * k) % h
414.   print(f)
415.   k = k + 1
416. endfor
417. print(f)
418. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
419. f = 0
420. k = 13
421. for k = 1 to 15
422.   f = (a * k) % h
423.   print(f)
424.   k = k + 1
425. endfor
426. print(f)
427. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
428. f = 0
429. k = 13
430. for k = 1 to 15
431.   f = (a * k) % h
432.   print(f)
433.   k = k + 1
434. endfor
435. print(f)
436. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
437. f = 0
438. k = 13
439. for k = 1 to 15
440.   f = (a * k) % h
441.   print(f)
442.   k = k + 1
443. endfor
444. print(f)
445. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
446. f = 0
447. k = 13
448. for k = 1 to 15
449.   f = (a * k) % h
450.   print(f)
451.   k = k + 1
452. endfor
453. print(f)
454. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
455. f = 0
456. k = 13
457. for k = 1 to 15
458.   f = (a * k) % h
459.   print(f)
460.   k = k + 1
461. endfor
462. print(f)
463. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
464. f = 0
465. k = 13
466. for k = 1 to 15
467.   f = (a * k) % h
468.   print(f)
469.   k = k + 1
470. endfor
471. print(f)
472. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
473. f = 0
474. k = 13
475. for k = 1 to 15
476.   f = (a * k) % h
477.   print(f)
478.   k = k + 1
479. endfor
480. print(f)
481. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
482. f = 0
483. k = 13
484. for k = 1 to 15
485.   f = (a * k) % h
486.   print(f)
487.   k = k + 1
488. endfor
489. print(f)
490. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
491. f = 0
492. k = 13
493. for k = 1 to 15
494.   f = (a * k) % h
495.   print(f)
496.   k = k + 1
497. endfor
498. print(f)
499. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
500. f = 0
501. k = 13
502. for k = 1 to 15
503.   f = (a * k) % h
504.   print(f)
505.   k = k + 1
506. endfor
507. print(f)
508. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
509. f = 0
510. k = 13
511. for k = 1 to 15
512.   f = (a * k) % h
513.   print(f)
514.   k = k + 1
515. endfor
516. print(f)
517. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
518. f = 0
519. k = 13
520. for k = 1 to 15
521.   f = (a * k) % h
522.   print(f)
523.   k = k + 1
524. endfor
525. print(f)
526. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
527. f = 0
528. k = 13
529. for k = 1 to 15
530.   f = (a * k) % h
531.   print(f)
532.   k = k + 1
533. endfor
534. print(f)
535. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
536. f = 0
537. k = 13
538. for k = 1 to 15
539.   f = (a * k) % h
540.   print(f)
541.   k = k + 1
542. endfor
543. print(f)
544. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
545. f = 0
546. k = 13
547. for k = 1 to 15
548.   f = (a * k) % h
549.   print(f)
550.   k = k + 1
551. endfor
552. print(f)
553. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
554. f = 0
555. k = 13
556. for k = 1 to 15
557.   f = (a * k) % h
558.   print(f)
559.   k = k + 1
560. endfor
561. print(f)
562. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
563. f = 0
564. k = 13
565. for k = 1 to 15
566.   f = (a * k) % h
567.   print(f)
568.   k = k + 1
569. endfor
570. print(f)
571. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
572. f = 0
573. k = 13
574. for k = 1 to 15
575.   f = (a * k) % h
576.   print(f)
577.   k = k + 1
578. endfor
579. print(f)
580. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
581. f = 0
582. k = 13
583. for k = 1 to 15
584.   f = (a * k) % h
585.   print(f)
586.   k = k + 1
587. endfor
588. print(f)
589. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
590. f = 0
591. k = 13
592. for k = 1 to 15
593.   f = (a * k) % h
594.   print(f)
595.   k = k + 1
596. endfor
597. print(f)
598. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
599. f = 0
600. k = 13
601. for k = 1 to 15
602.   f = (a * k) % h
603.   print(f)
604.   k = k + 1
605. endfor
606. print(f)
607. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
608. f = 0
609. k = 13
610. for k = 1 to 15
611.   f = (a * k) % h
612.   print(f)
613.   k = k + 1
614. endfor
615. print(f)
616. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
617. f = 0
618. k = 13
619. for k = 1 to 15
620.   f = (a * k) % h
621.   print(f)
622.   k = k + 1
623. endfor
624. print(f)
625. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
626. f = 0
627. k = 13
628. for k = 1 to 15
629.   f = (a * k) % h
630.   print(f)
631.   k = k + 1
632. endfor
633. print(f)
634. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
635. f = 0
636. k = 13
637. for k = 1 to 15
638.   f = (a * k) % h
639.   print(f)
640.   k = k + 1
641. endfor
642. print(f)
643. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
644. f = 0
645. k = 13
646. for k = 1 to 15
647.   f = (a * k) % h
648.   print(f)
649.   k = k + 1
650. endfor
651. print(f)
652. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
653. f = 0
654. k = 13
655. for k = 1 to 15
656.   f = (a * k) % h
657.   print(f)
658.   k = k + 1
659. endfor
660. print(f)
661. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
662. f = 0
663. k = 13
664. for k = 1 to 15
665.   f = (a * k) % h
666.   print(f)
667.   k = k + 1
668. endfor
669. print(f)
670. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
671. f = 0
672. k = 13
673. for k = 1 to 15
674.   f = (a * k) % h
675.   print(f)
676.   k = k + 1
677. endfor
678. print(f)
679. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
680. f = 0
681. k = 13
682. for k = 1 to 15
683.   f = (a * k) % h
684.   print(f)
685.   k = k + 1
686. endfor
687. print(f)
688. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
689. f = 0
690. k = 13
691. for k = 1 to 15
692.   f = (a * k) % h
693.   print(f)
694.   k = k + 1
695. endfor
696. print(f)
697. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
698. f = 0
699. k = 13
700. for k = 1 to 15
701.   f = (a * k) % h
702.   print(f)
703.   k = k + 1
704. endfor
705. print(f)
706. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
707. f = 0
708. k = 13
709. for k = 1 to 15
710.   f = (a * k) % h
711.   print(f)
712.   k = k + 1
713. endfor
714. print(f)
715. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
716. f = 0
717. k = 13
718. for k = 1 to 15
719.   f = (a * k) % h
720.   print(f)
721.   k = k + 1
722. endfor
723. print(f)
724. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
725. f = 0
726. k = 13
727. for k = 1 to 15
728.   f = (a * k) % h
729.   print(f)
730.   k = k + 1
731. endfor
732. print(f)
733. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
734. f = 0
735. k = 13
736. for k = 1 to 15
737.   f = (a * k) % h
738.   print(f)
739.   k = k + 1
740. endfor
741. print(f)
742. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
743. f = 0
744. k = 13
745. for k = 1 to 15
746.   f = (a * k) % h
747.   print(f)
748.   k = k + 1
749. endfor
750. print(f)
751. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
752. f = 0
753. k = 13
754. for k = 1 to 15
755.   f = (a * k) % h
756.   print(f)
757.   k = k + 1
758. endfor
759. print(f)
760. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
761. f = 0
762. k = 13
763. for k = 1 to 15
764.   f = (a * k) % h
765.   print(f)
766.   k = k + 1
767. endfor
768. print(f)
769. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
770. f = 0
771. k = 13
772. for k = 1 to 15
773.   f = (a * k) % h
774.   print(f)
775.   k = k + 1
776. endfor
777. print(f)
778. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
779. f = 0
780. k = 13
781. for k = 1 to 15
782.   f = (a * k) % h
783.   print(f)
784.   k = k + 1
785. endfor
786. print(f)
787. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
788. f = 0
789. k = 13
790. for k = 1 to 15
791.   f = (a * k) % h
792.   print(f)
793.   k = k + 1
794. endfor
795. print(f)
796. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
797. f = 0
798. k = 13
799. for k = 1 to 15
800.   f = (a * k) % h
801.   print(f)
802.   k = k + 1
803. endfor
804. print(f)
805. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
806. f = 0
807. k = 13
808. for k = 1 to 15
809.   f = (a * k) % h
810.   print(f)
811.   k = k + 1
812. endfor
813. print(f)
814. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
815. f = 0
816. k = 13
817. for k = 1 to 15
818.   f = (a * k) % h
819.   print(f)
820.   k = k + 1
821. endfor
822. print(f)
823. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
824. f = 0
825. k = 13
826. for k = 1 to 15
827.   f = (a * k) % h
828.   print(f)
829.   k = k + 1
830. endfor
831. print(f)
832. 20 8 21 3 16 29 11 24 6 19 1 14 27 9
833. f = 0
834
```

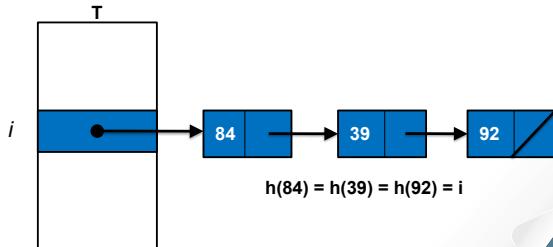
CE2001/ CZ2001: Algorithms

Closed Address Hashing

Dr. Loke Yuan Ren

Collision Handling 1: Closed Address Hashing

- Maintains the original hashed address
- Records hashed to the same slot are linked into a list
- Also called **Chained Hashing**



*more
how many
data in
each slot*

Learning Objectives

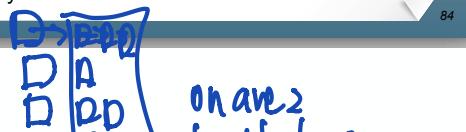
At the end of this lecture, students should be able to:

- Explain and apply closed address hashing method for collision handling
- Analyze worst-case and average-case complexity

82

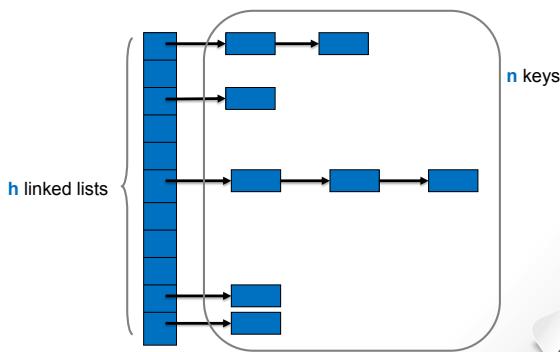
Closed Address Hashing

- Initially, all entries in the hash table are empty lists.
- All elements with hash address i will be inserted into the linked list $H[i]$.
- If there are n records to store in the hash table, then n/h is the **load factor** of the hash table.
- In closed address hashing, there will be n/h number of elements in each linked list on average.
- During searching, the searched element with hash address i is compared with elements in linked list $H[i]$ sequentially



84

Example of Hash Table with Chained Hashing



85

Average Case Analysis (Unsuccessful Search)

If we assume that any given item is equally likely to hash into any of the h slots, an **unsuccessful search** on average does n/h key comparisons.

Proof:

1. An unsuccessful search means searching to the end of the list.
2. The number of comparisons is equal to the length of the list.
3. The average length of all lists is the load factor, n/h .
4. Thus the expected number of comparisons in an unsuccessful search is n/h .

87

2 DD load factor 2

Analysis of chained hashing

- The **worst case** behaviour of hashing happens when all elements are hashed to **the same slot**. In this case,
 - The linked list contains all n elements
 - An **unsuccessful search** will do n key comparisons
 - A **successful search**, assuming the probability of searching for each item is $1/n$, will take

$$\frac{1}{n} \sum_{i=1}^n i = \frac{(n+1)}{2} = \Theta(n)$$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

Note:
Checking if link list is **NULL** is not counted as a key comparison

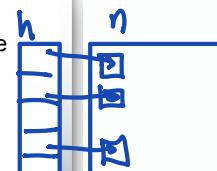
86

Average Case Analysis (Successful Search)

Assume that any given item is equally likely to hash into any of the h slots, a **successful search** on average does $\Theta(1 + n/h)$ comparisons.

Proof (Optional):

1. We assume that the items are inserted at the end of the list in each slot.
2. The expected no. of comparisons in a successful search is 1 more than the no. of comparisons done when the sought after item was inserted into the hash table. **insertion limit**
3. When the i^{th} item is inserted into the hash table, the average length of all lists is $(i-1)/h$. So when the i^{th} item is sought for, the no. of comparisons is $\left(1 + \frac{i-1}{h}\right)$.



$$\sum_{i=1}^{i-1} \frac{i-1}{h} + 1$$

first you did
 $\frac{i-1}{h}$ unsuccessful search
 try to search for i^{th} item

88

Average Case Analysis (Successful Search)

So the average number of comparisons over n items is:

$$\begin{aligned}
 & \frac{1}{n} \sum_{i=1}^n \left(1 + \frac{i-1}{h} \right) \\
 & = \frac{1}{n} \sum_{i=1}^n (1) + \frac{1}{nh} \sum_{i=1}^n (i-1) \\
 & = 1 + \frac{1}{nh} \sum_{i=0}^{n-1} i \quad \text{Arithmetic series} \\
 & = 1 + \frac{n-1}{2h}
 \end{aligned}$$

Therefore, a successful search on average does $\Theta(1 + n/h)$ comparisons (end of proof).

If n is proportional to h, i.e. $n = O(h)$, then $n/h = O(h)/h = O(1)$.

Thus, each successful search with chained hashing takes constant time averagely.

89

but not the end . measured $\frac{i-1}{h}$

Recap

- Closed address hashing
 - Handle collision through linked lists
 - Load factor = n/h
 - Worst case takes n comparisons
 - Average case
 - Unsuccessful search n/h comparisons
 - Successful search $O(1)$

90

Learning Objectives

At the end of this lecture, students should be able to:

- Describe how open address hashing works
- Apply open address hashing
- Explain and apply two types of rehashing:
 - Linear probing
 - Double hashing

92

CE2001/ CZ2001: Algorithms

Open Address Hashing

Dr. Loke Yuan Ren

Open Address Hashing

- To store all elements in the hash table
 - The load factor n/h is never greater than 1
 - When collision occurs, rehash/ probe the alternative empty slot
 - Linear Probing: $h(k,i) = (h'(k) + i) \text{ mod } m$
 - Hash function, $h(k) = k \text{ mod } m$
 - If $h(k)$ is non-empty slot
 - Rehash: $h'(h(k), i) = (h(k) + i) \text{ mod } m, i=1, \dots, m-1$
 - RPFAT till empty slot is found

*k: key
m: hash table size*

Searching A Key with Rehashing

```

compute the hash code, code = h(k);
loc = code; ans = blank structure;
while (H[loc] is not empty) {
    if (H[loc].key == k) {
        ans = H[loc];
        break;
    }
    else {
        loc = rehash(loc);
        if (loc == code) break;
    }
}
ans = H[loc]

```

Key found!

Compute another location

Note:
Checking if $H[loc]$ is empty is not counted as a key comparison

Open Address Hashing (Example)

- Consider the linear probing policy for storing the following keys: 1055, 1492, 1776, 1812, 1918, 1942. The hash function:

$$h(k) = k \bmod 10$$

Number of species	Number of genera
1055	1492
1492	1812
1776	1942
1812	1055
1918	1776
1942	1918

Searching A Key with Rehashing

```

compute the hash code, code = h(k);
loc = code; ans = blank structure;
while (H[loc] is not empty) {
    if (H[loc].key == k) {
        ans = H[loc];
        break;
    }
    else {
        loc = rehash(loc);
        if (loc == code) break;
    }
}
ans = H[loc]

```

Key found!

Compute another location

Note:
Checking if $H[loc]$ is empty is not counted as a key comparison

Behaviour of Rehashing

- **Three outcomes of searching:**

1. key found at $h(k)$ – success!
2. position empty – fail!
3. probe table (downwards subject to mod. table size) until key found or empty slot met or whole table searched.

- **Deletion under open addressing**

A slot becomes empty when an element is deleted: should be marked as 'obsolete' or 'tombstone' instead, so that searching will not stop there.

97

Double Hashing

- **Double hashing:** a random rehashing method

- Hash function, $h(k) = k \bmod m$

- $d = \text{hashIncr}(k)$;

- $\text{rehash}(j, d) = (h(k) + i \cdot d) \bmod m$,
 $i=0,1,2,\dots,m-1$

where **hashIncr()** is another hash function.

- The hash table size, **m**, should be a prime number.

99

Limitations of Rehashing

- **Issue with linear probing:** Searching is expensive when the load factor approaches 1.

- **Primary clustering:** Long runs of occupied slots

98

Linear Probing Vs Double Hashing (Example)

To store the following keys:

1051, 1492, 1776, 1812, 1918, 1561, **523**, 1340

- **Linear probing**

$h(k) = k \bmod 11$ and $\text{rehash}(j) = (j+i) \bmod 11$, $i=1, 2, \dots, 10$

- **Double hashing**

$h(k) = k \bmod 11$, $d = \text{hashIncr}(k) = k \bmod 8 + 1$;
 $\text{rehash}(j, d) = (j+i \cdot d) \bmod 11$, $i=1, 2, \dots, 10$

100

Linear Probing Vs Double Hashing (Example)

To store the following keys:

1051, 1492, 1776, 1812, 1918, 1561, **523**, 1340

- **Linear probing**

$$h(k) = k \bmod 11 \text{ and } \text{rehash}(j) = (j+i) \bmod 11, i=1, 2\dots 10$$

- **Double hashing**

$$h(k) = k \bmod 11, d = \text{hashIncr}(k) = k \bmod 8 + 1; \\ \text{and } \text{rehash}(j, d) = (j+i*d) \bmod 11, i=1, 2\dots 10$$

101

Double Hashing

- $d = \text{hashIncr}(k) = k \bmod 8 + 1$
 - Why do I **+1**?
- The size of hash table in double hashing must be prime number?

Double hashing

$$h(k) = k \bmod 11, d = \text{hashIncr}(k) = k \bmod 8 + 1; \\ \text{and } \text{rehash}(j, d) = (j+i*d) \bmod 11, i=1, 2\dots 10$$

103

Linear Probing Vs Double Hashing (Example)

Linear Probing

0	1340
1	
2	
3	
4	1918
5	1776
6	1051
7	1492
8	1812
9	523
10	1561

0	1051
1	1492
2	1776
3	1812
4	1918
5	1561
6	523
7	1340
8	
9	
10	

Double Hashing

0	0
1	1
2	2
3	523
4	1918
5	1776
6	1051
7	1492
8	1812
9	1340
10	1561

Recap

- Collision Handling
 - Open Address Hashing – not fixed address (hash function)
 - Closed Address Hashing – fixed address

104