

Chapter 13

Searching

Objectives

Upon completion you will be able to:

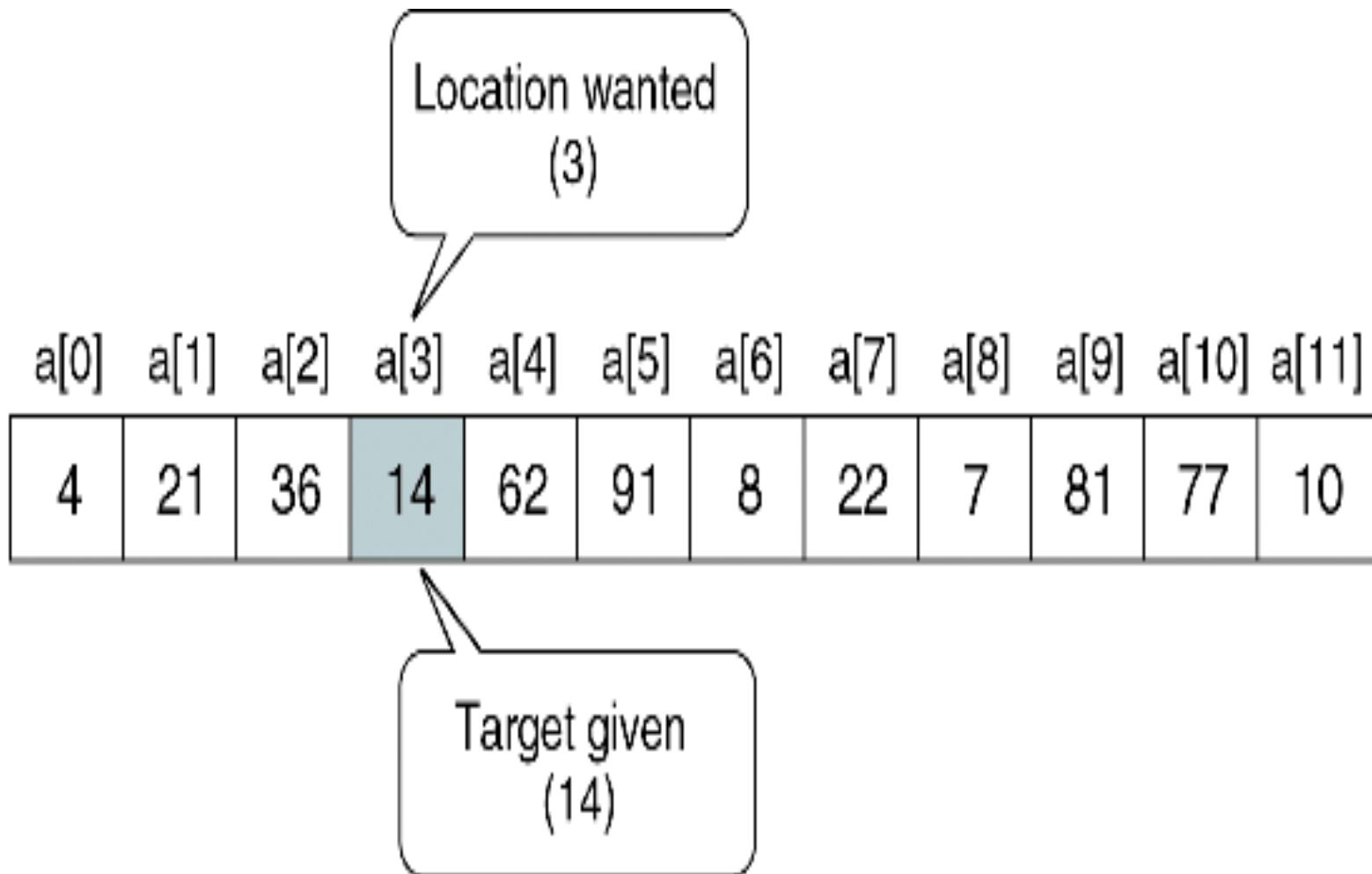
- Design and implement sequential searches
- Discuss the relative merits of different sequential searches
- Design and implement binary searches
- Design and implement hash-list searches
- Discuss the merits of different collision resolution algorithms

13-1 List Searches

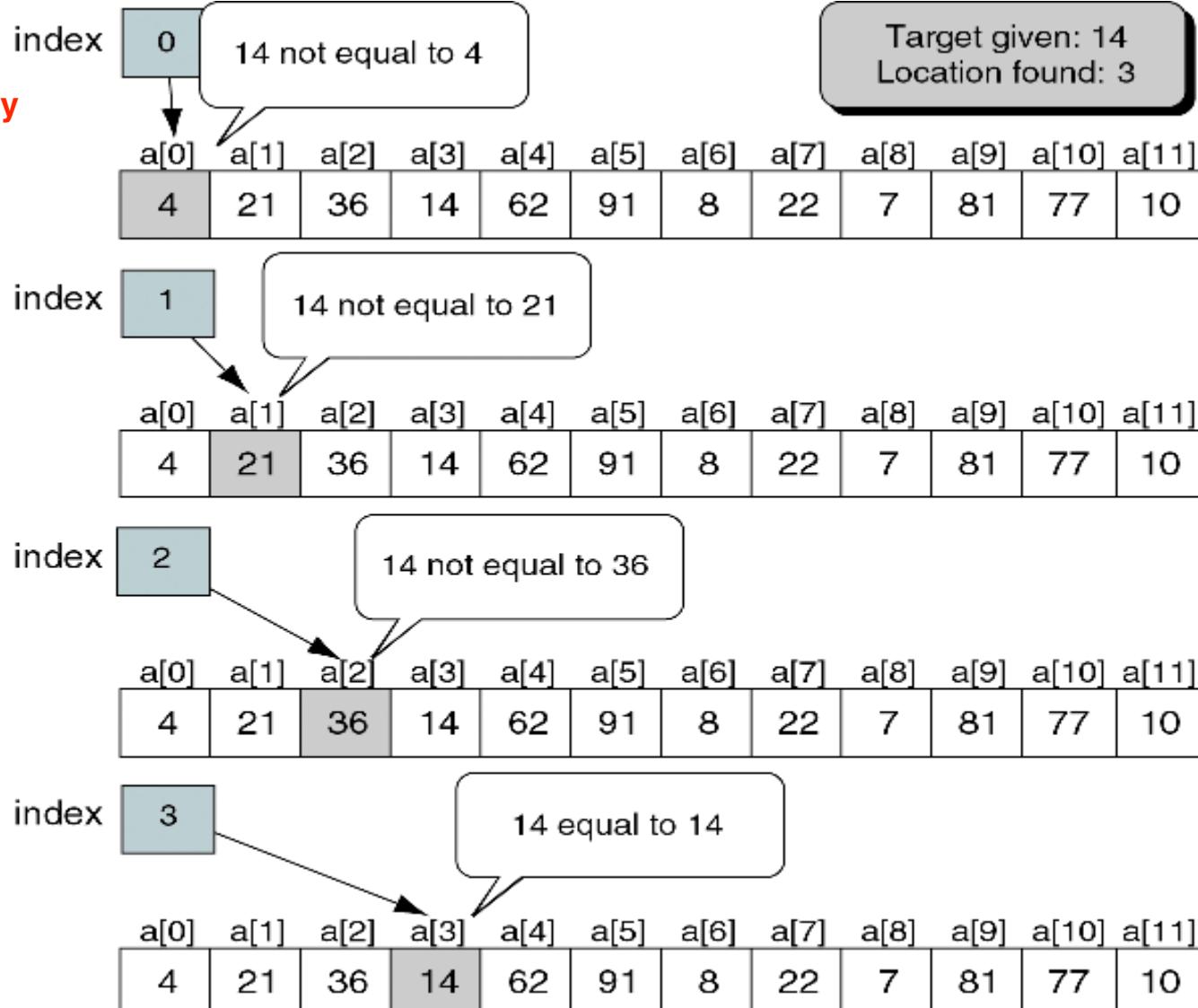
In this section we study searches that work with arrays. The two basic searches for arrays are the sequential search and the binary search.

- **Sequential Search**
- **Variations on Sequential Searches**
- **Binary Search**
- **Analyzing Search Algorithms**

Search Concept

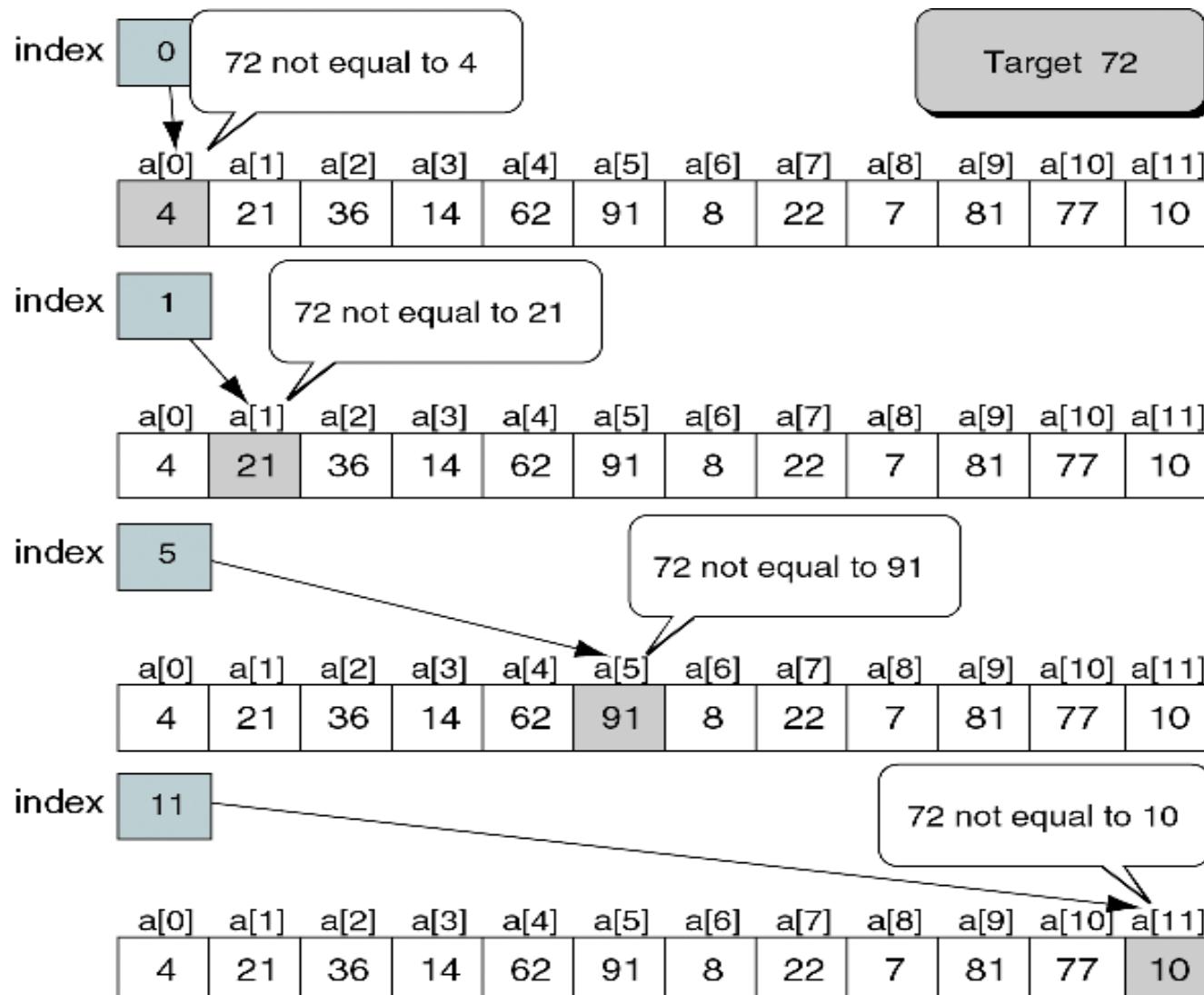


Successful Search of Unordered List



Unsuccessful Search of Unordered List

Worst case



Note: Not all test points are shown.

Sequential Search

```
Algorithm seqSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
    Pre    list must contain at least one element
           last is index to last element in the list
           target contains the data to be located
           locn is address of index in calling algorithm
    Post   if found: index stored in locn & found true
           if not found: last stored in locn & found false
    Return found true or false
1 set looker to 0
2 loop (looker < last AND target not equal list[looker])
    1 increment looker
3 end loop
4 set locn to looker
5 if (target equal list[looker])
    1 set found to true
6 else
    1 set found to false
7 end if
8 return found
end seqSearch
```

Variation 1: Sentinel Search

```
Algorithm SentinelSearch (list, last, target, locn)
Locate the target in an unordered list of elements.
Pre    list must contain element at the end for sentinel
       last is index to last data element in the list
       target contains the data to be located
       locn is address of index in calling algorithm
Post   if found--matching index stored in locn & found
       set true
       if not found--last stored in locn & found false
Return found true or false
1 set list[last + 1] to target
2 set looker to 0
3 loop (target not equal list[looker])
1   increment looker
4 end loop
5 if (looker <= last)
1   set found to true
2   set locn to looker
6 else
1   set found to false
2   set locn to last
7 end if
8 return found
end SentinelSearch
```

Variation 2: Probability Search

```
Algorithm ProbabilitySearch (list, last, target, locn)
Locate the target in a list ordered by the probability of each
element being the target--most probable first, least probable
last.

Pre    list must contain at least one element
       last is index to last element in the list
       target contains the data to be located
       locn is address of index in calling algorithm
Post   if found--matching index stored in locn,
       found true, and element moved up in priority.
       if not found--last stored in locn & found false
Return found true or false

1 find target in list
2 if (target in list)
   1 set found to true
   2 set locn to index of element containing target
   3 if (target after first element)
      1 move element containing target up one location
   4 end if
3 else
   1 set found to false
4 end if
5 return found
end ProbabilitySearch
```

Ordered List Search

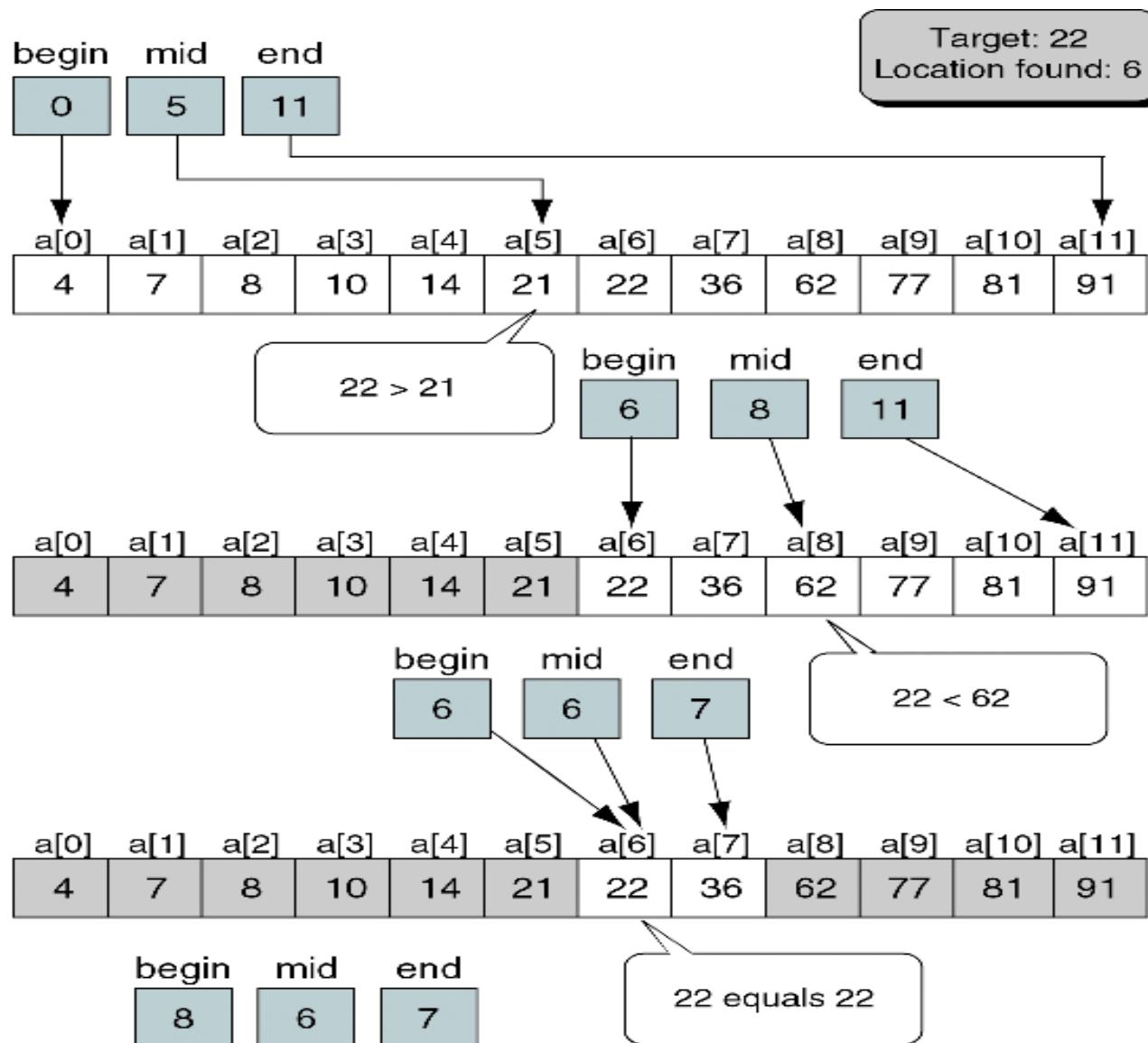
- Pure sequential search: $O(n)$ and two conditions in loop
- Sentinel search: $O(n)$ and single one condition in loop
- Ordered list search
 - Seq. search with target = list [loc]: $O(n)$ with 2 tests
 - Seq. search with target $>$ or \leq list[loc]: $O(n/2)$ with 2 tests

```
1 if (target less than last element in list)
    1 find first element less than or equal to target
    2 set locn to index of element
2 else
    1 set locn to last
3 end if
4 if (target in list)
    1 set found to true
5 else
    1 set found to false
6 end if
7 return found
end OrderedListSearch
```

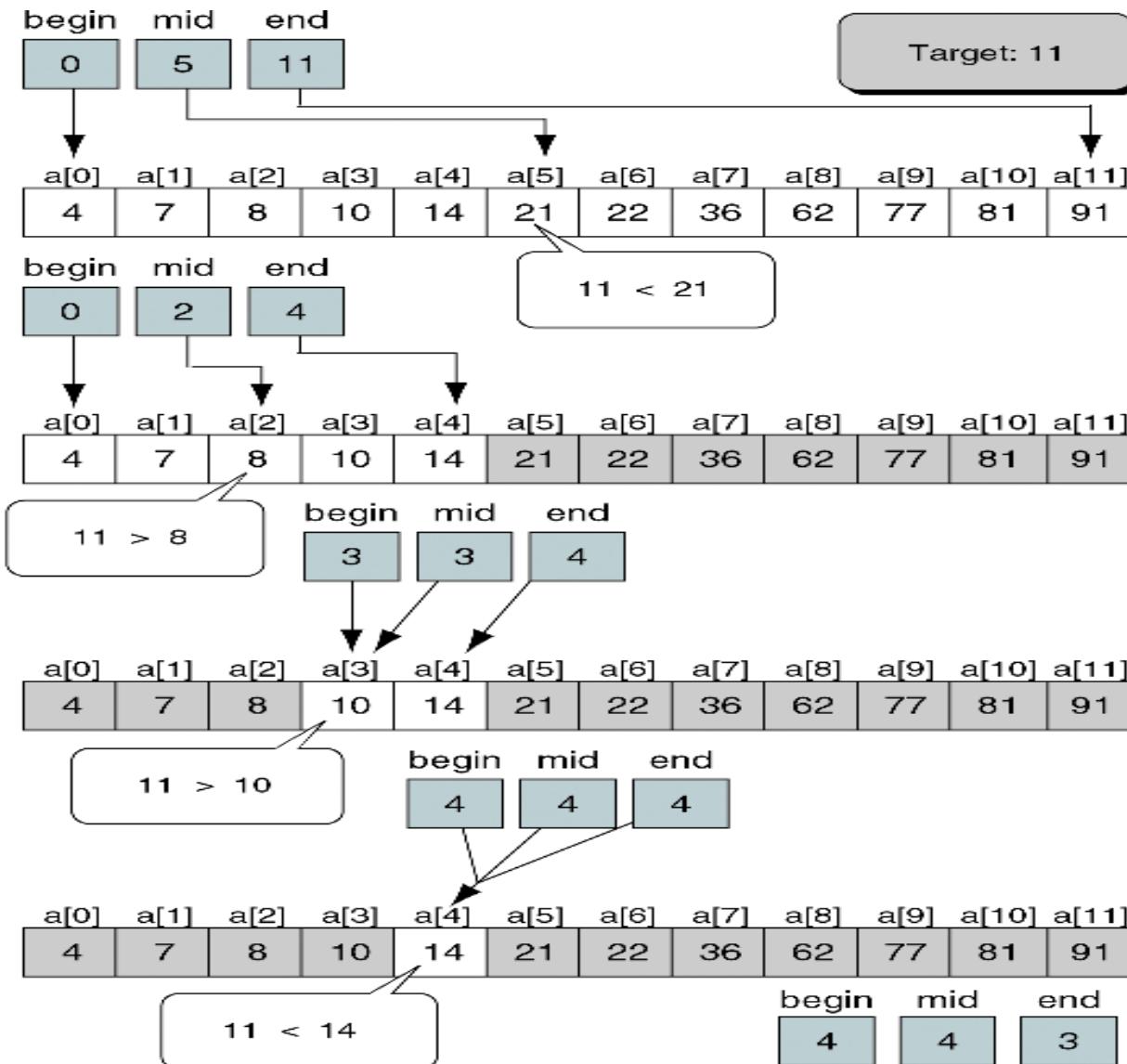
$O(n/2)$ with one test !



Successful Binary Search Example



Unsuccessful Binary Search Example



Binary Search

```
Algorithm binarySearch (list, last, target, locn)
Search an ordered list using Binary Search
    Pre    list is ordered; it must have at least 1 value
           last is index to the largest element in the list
           target is the value of element being sought
           locn is address of index in calling algorithm
    Post   FOUND: locn assigned index to target element
           found set true
           NOT FOUND: locn = element below or above target
           found set false
    Return found true or false
1  set begin to 0
2  set end to last
3  loop (begin <= end)
    1  set mid to (begin + end) / 2
    2  if (target > list[mid])
        Look in upper half
        1  set begin to (mid + 1)
```

Binary Search (cont.)

```
3 else if (target < list[mid])
    Look in lower half
    1 set end to mid - 1
4 else
    Found: force exit
    1 set begin to (end + 1)
5 end if
4 end loop
5 set locn to mid
6 if (target equal list [mid])
    1 set found to true
7 else
    1 set found to false
8 end if
9 return found
end binarySearch
```

Efficiency (time complexity): $O(\log n)$

13-2 Search Implementations

Having discussed the basic concepts of array searches, we write C implementations of the two most common.

- **Sequential Search in C**
- **Binary Search In C**

Sequential Search

```
1  /* Locate target in an unordered list of size elements.  
2   Pre  list must contain at least one item  
3       last is index to last element in list  
4       target contains the data to be located  
5       locn is address of index in calling function  
6   Post FOUND: matching index stored in locn address  
7       return true (found)  
8   NOT FOUND: last stored in locn address  
9       return false (not found)  
10 */  
11 bool seqSearch (int list[ ], int last,  
12                  int target, int* locn)  
13 {  
14 // Local Definitions  
15     int looker;
```

Sequential Search (cont.)

```
16
17 // Statements
18     looker = 0;
19     while (looker < last && target != list[looker])
20         looker++;
21
22     *locn = looker;
23     return ( target == list[looker] );
24 } // seqSearch
```

Binary Search

```
1  /* Search an ordered list using Binary Search.  
2      Pre  list must contain at least one element  
3              last is index to largest element in list  
4              target is value of element being sought  
5              locn is address of index in calling function  
6      Post FOUND: locn = index to target element  
7                      return true (found)  
8      NOT FOUND: locn = index below/above target  
9                      return false (not found)
```

Binary Search (cont.)

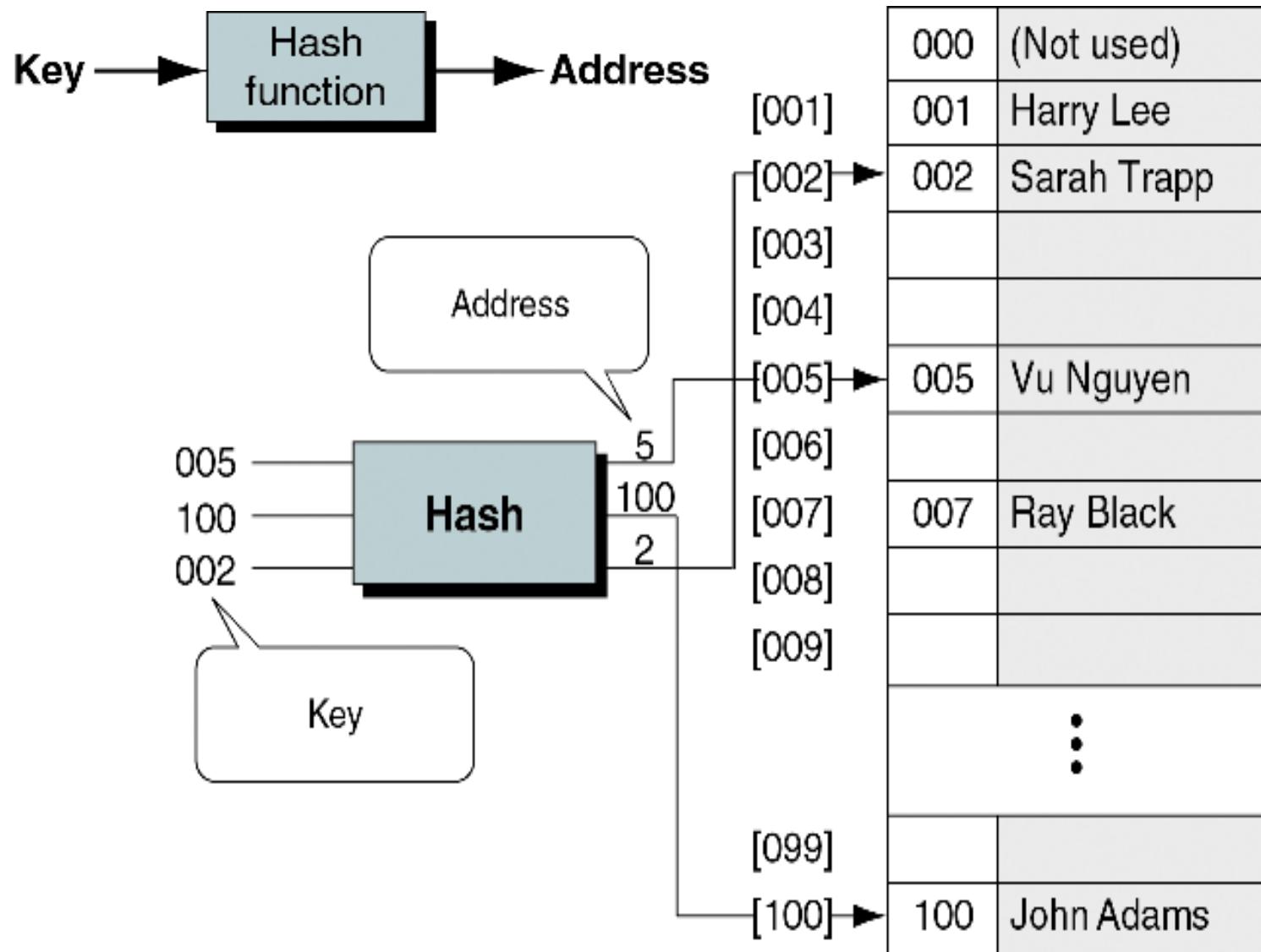
```
10  */
11 bool binarySearch    (int list[ ], int last,
12                           int target, int* locn )
13 {
14 // Local Definitions
15     int begin;
16     int mid;
17     int end;
18
19 // Statements
20     begin = 0;
21     end   = last;
22     while (begin <= end)
23     {
24         mid = ( begin + end ) / 2;
25         if ( target > list[ mid ] )
26             // look in upper half
27             begin = mid + 1;
28         else if ( target < list[ mid ] )
29             // look in lower half
30             end = mid - 1;
31         else
32             // found: force exit
33             begin = end + 1;
34     } // end while
35     *locn = mid;
36     return (target == list [mid]);
37 } // binarySearch
```

13-3 Hashed List Searches

The goal of a hashed search is to find the target data in only one test. After discussing some basic hashed list concepts, we discuss eight hashing methods. At the end of the section, we develop a hashing algorithm that uses several of the methods discussed in the section.

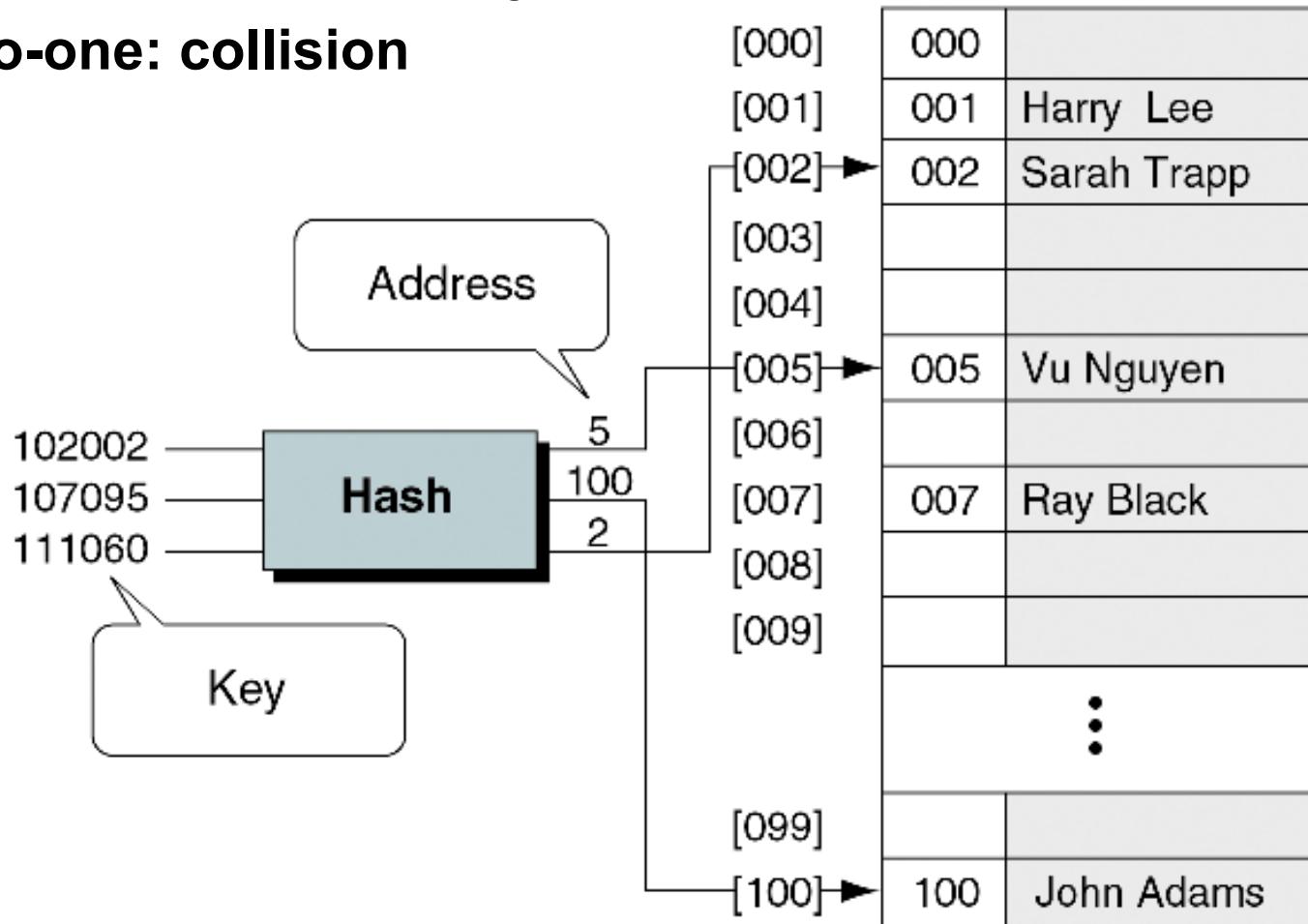
- Basic Concepts
- Hashing Methods
- One Hashing Algorithm

Hash Concept

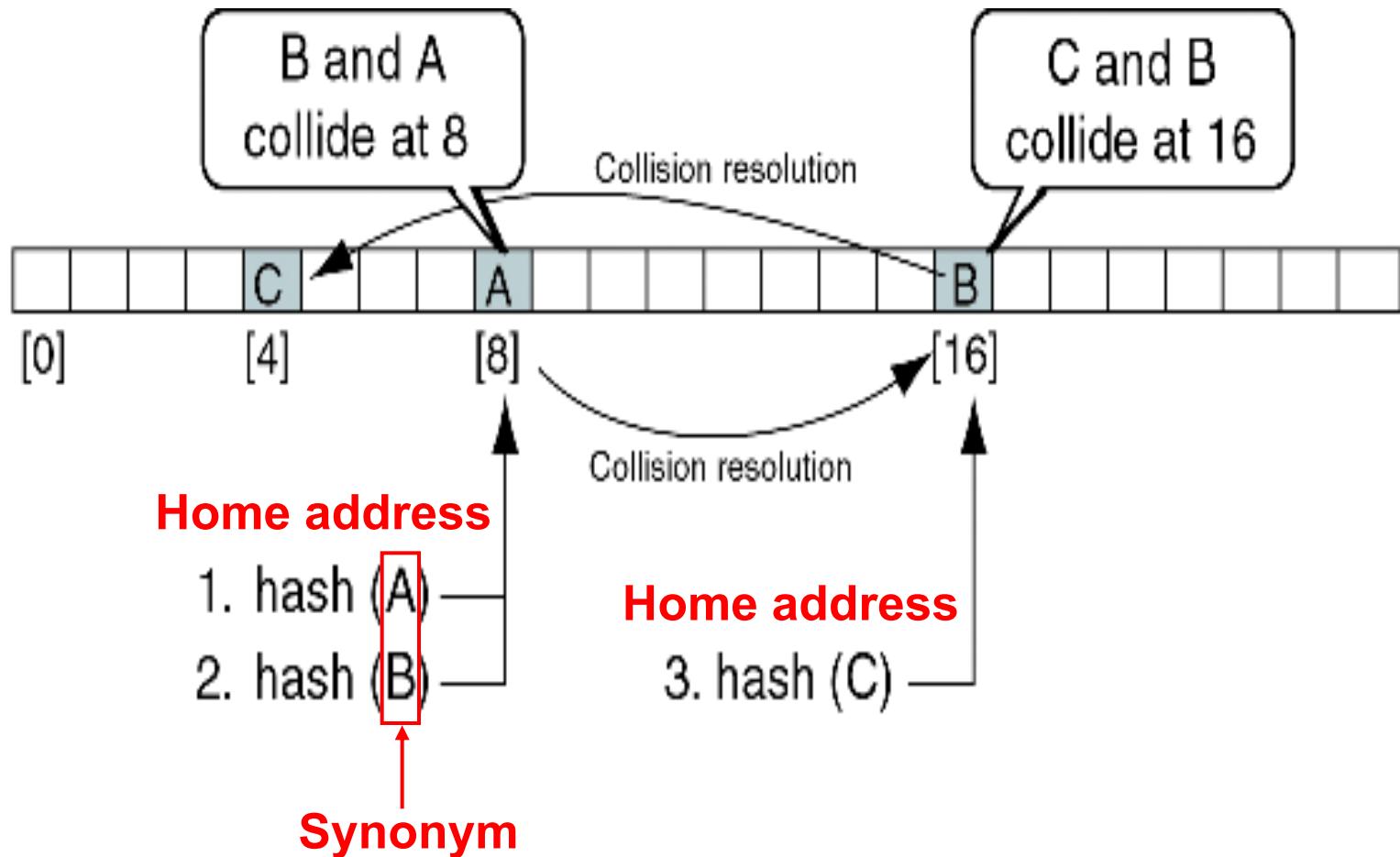


Hash Concept (cont.)

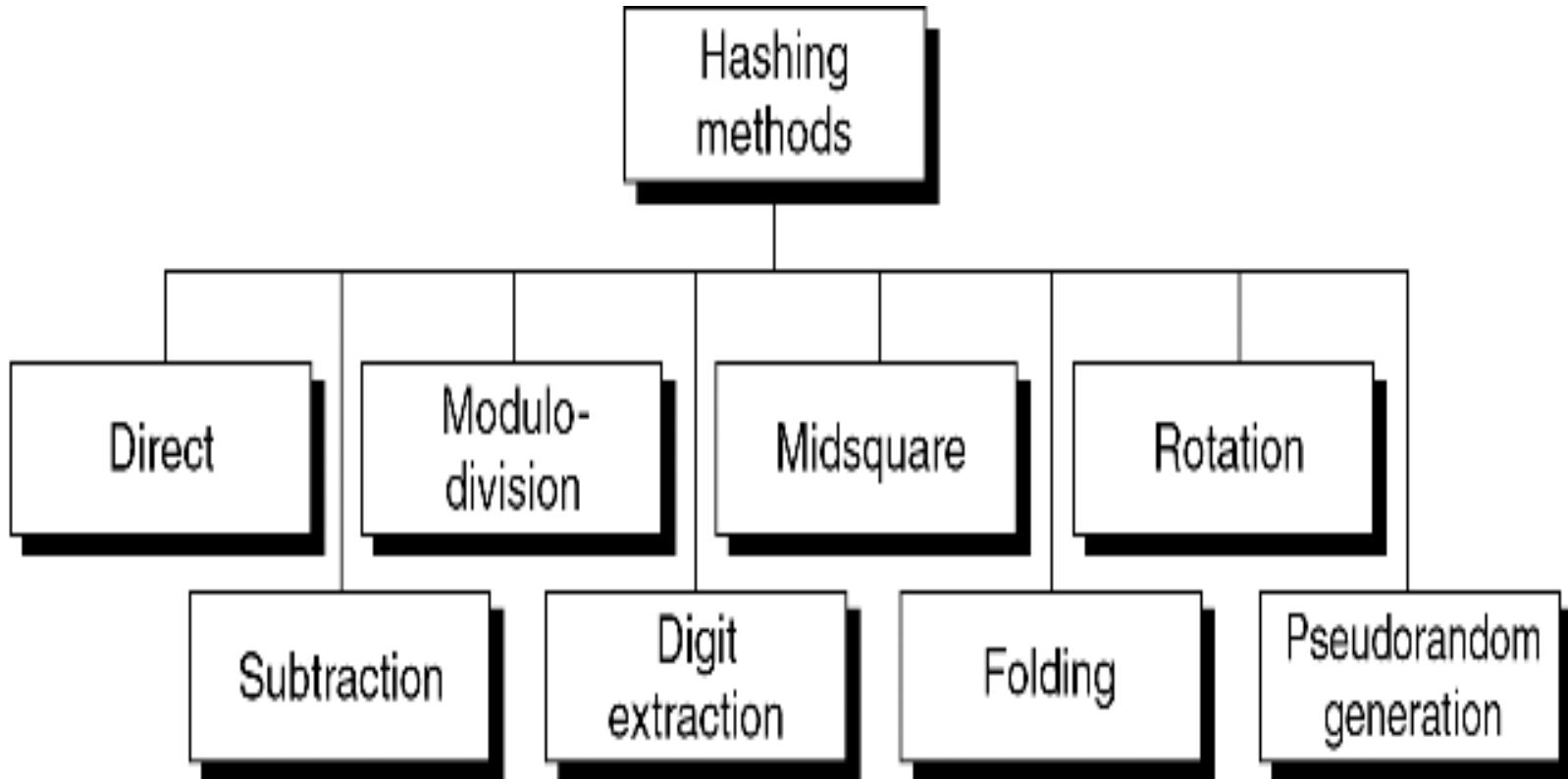
- What about number of key (e.g. employee ID) **>>** number of address (employee) ?
 - One-to-one: waste memory space
 - Many-to-one: collision



Collision Resolution Concept

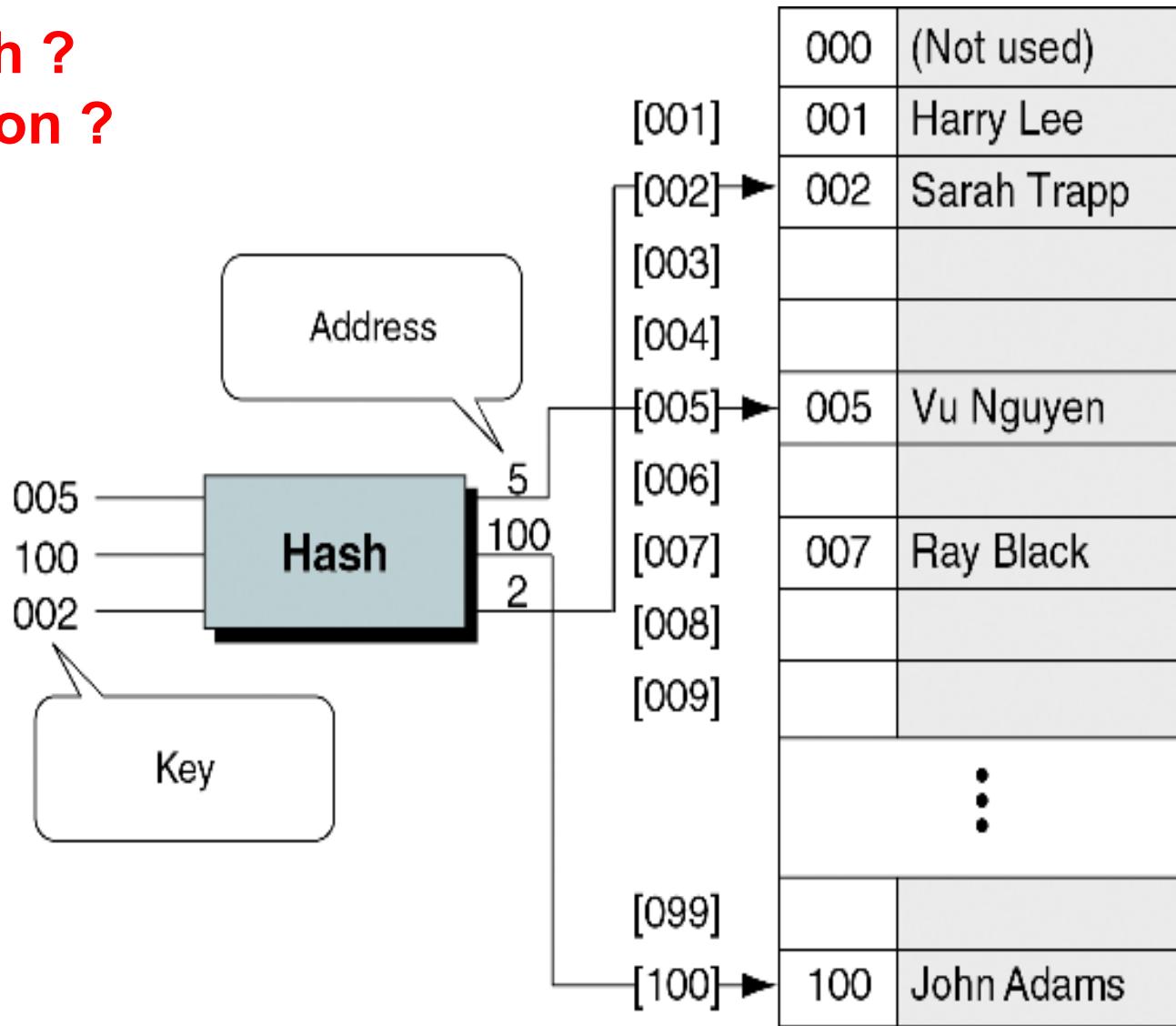


Basic Hash Techniques



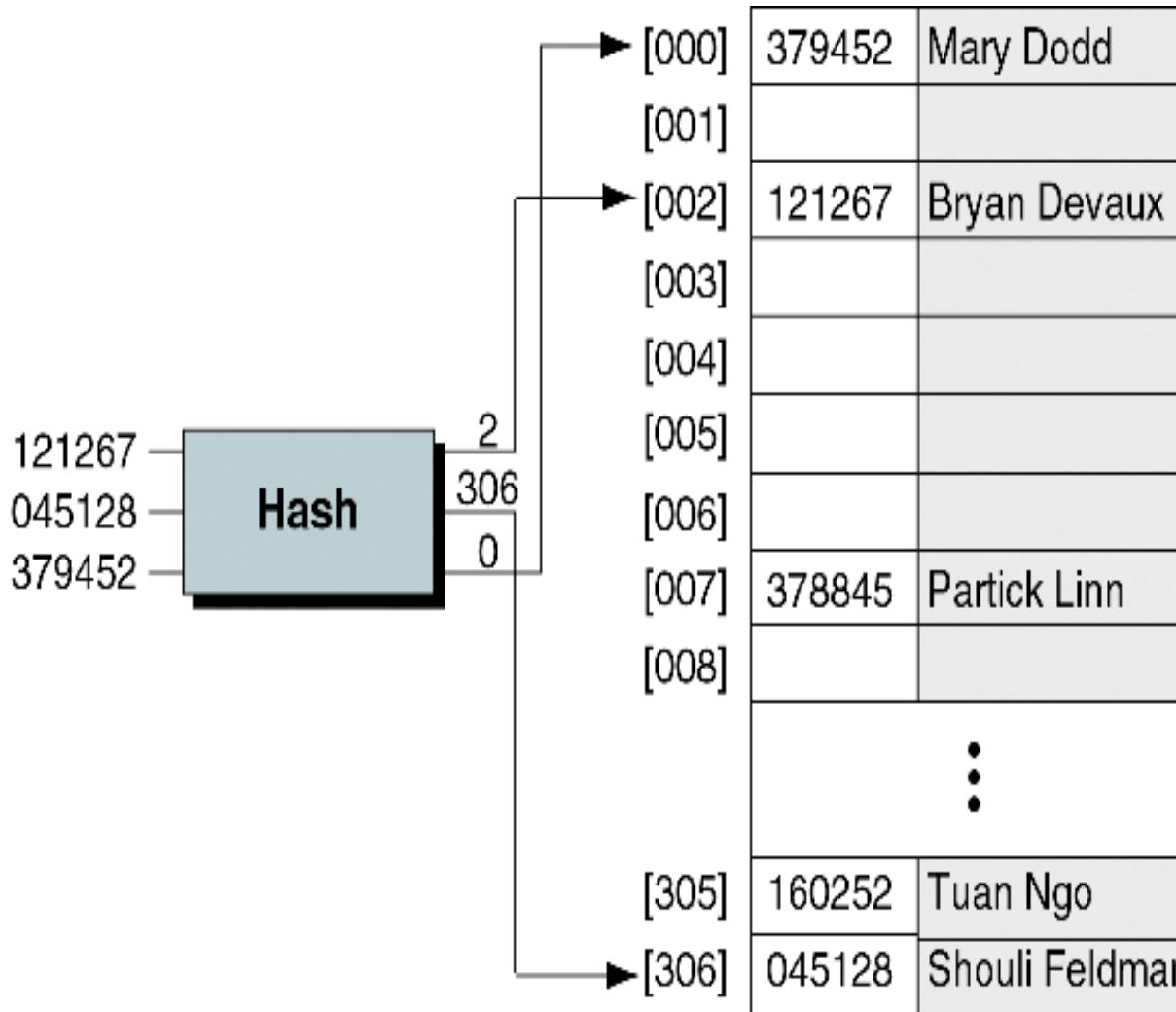
Direct Hashing

Strength ?
Limitation ?



Modulo-division Hashing

- Address = key *mod* listsize e.g., $2 = 121267 \bmod 307$
- A listsizethat is **prime number** produces fewer collisions

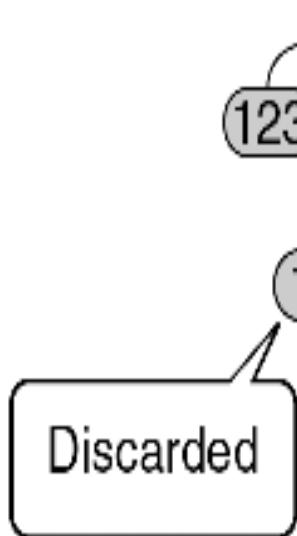


Digit-extraction and Midsquare Hashing

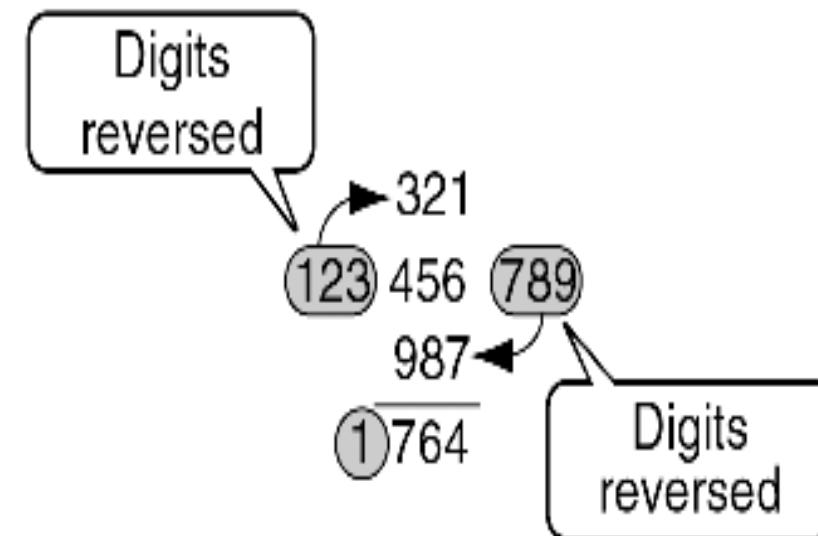
- Digit – extraction (address space 1 ~ 999)
 - 379452 → 394
 ✓ ✓ ✓ ✓ ✓ ✓ ✓
 - 121267 → 112
 ✓ ✓ ✓ ✓ ✓ ✓ ✓
 - 378845 → 388
 ✓ ✓ ✓ ✓ ✓ ✓ ✓
- Midsquare (address space 1 ~ 999)
 - $\text{Sqr}(9452) = 89\boxed{3403}04 \rightarrow \text{address: } 3403$
 - $\boxed{379}452 \rightarrow \text{sqr}(379) = 14\boxed{364}1 \rightarrow \text{address: } 364$

Folding Hashing

Key
123456789



(a) Fold shift



(b) Fold boundary

Rotation Hashing

- Rotation hashing is usually incorporated in combination with other hashing methods, especially when keys are assigned serially
- In the example, it does not work well with *mod* method
- It works well with simple fold shift (two digit address)

Original key	Rotation	Rotated key
600101	62	160010 26
600102	63	260010 36
600103	64	360010 46
600104	65	460010 56
600105	66	560010 66

Pseudorandom Hashing

- In pseudorandom hashing the key is used as the seed in a pseudorandom-number generator
- A common random-number generator
 - $y = ax + c$, where x is key, a and c are predefined constant
- For example ($a = 17$, $c = 7$)
 - $121267 \rightarrow y = ((17 \times 121267) + 7) \bmod 307 = 41$

Hashing Algorithm Example

```
Algorithm hash (key, size, maxAddr, addr)
```

This algorithm converts an alphanumeric key of size characters into an integral address.

Pre key is a key to be hashed
 size is the number of characters in the key
 maxAddr is maximum possible address for the list

Post addr contains the hashed address

1 set looper to 0

2 set addr to 0

Hash key

3 for each character in key **fold-shift**

 1 if (character not space)
 1 add character to address
 2 rotate addr 12 bits right

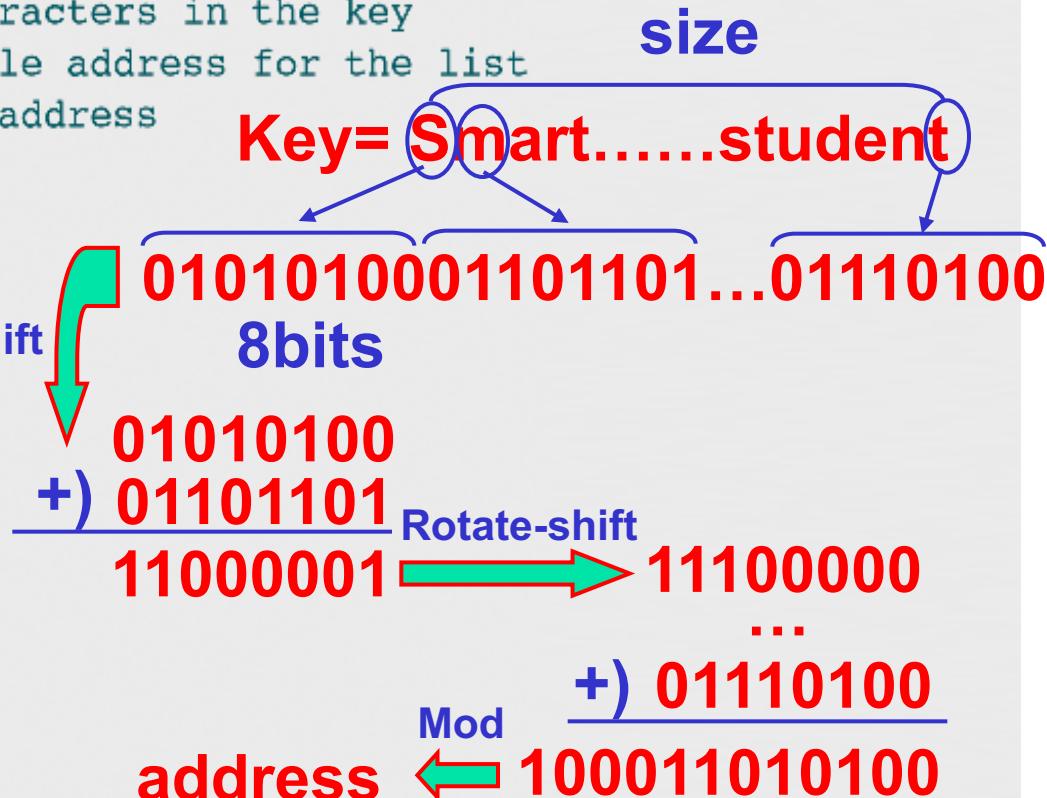
 2 end if

4 end loop

Test for negative address

5 if (addr < 0)
 1 addr = absolute(addr)
6 end if
7 addr = addr modulo maxAddr

end hash

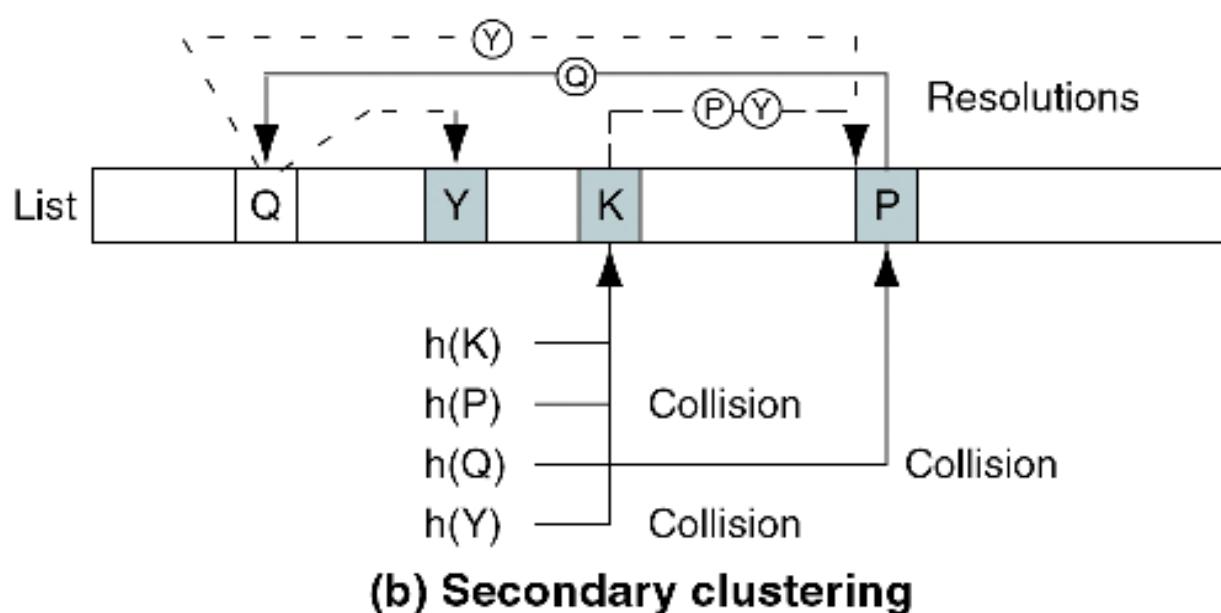
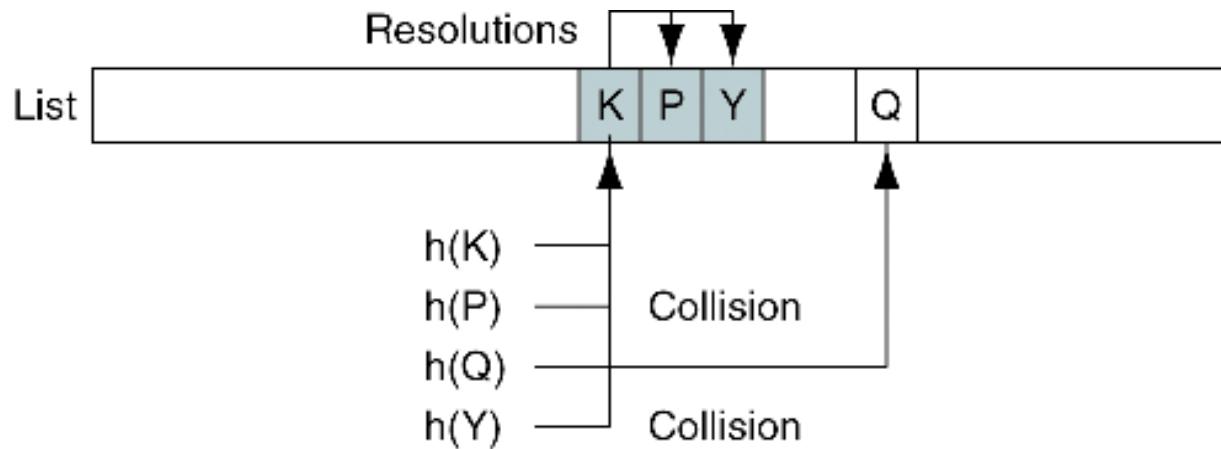


13-4 Collision Resolution

We discuss three structural approaches and four specific collision resolution algorithms. At the end of the section, we develop a C program that builds and searches a hashed list.

- **Open Addressing**
- **Linked List Collision Resolution**
- **Bucket Hashing**
- **Combination Approaches**
- **Hashed List Example**

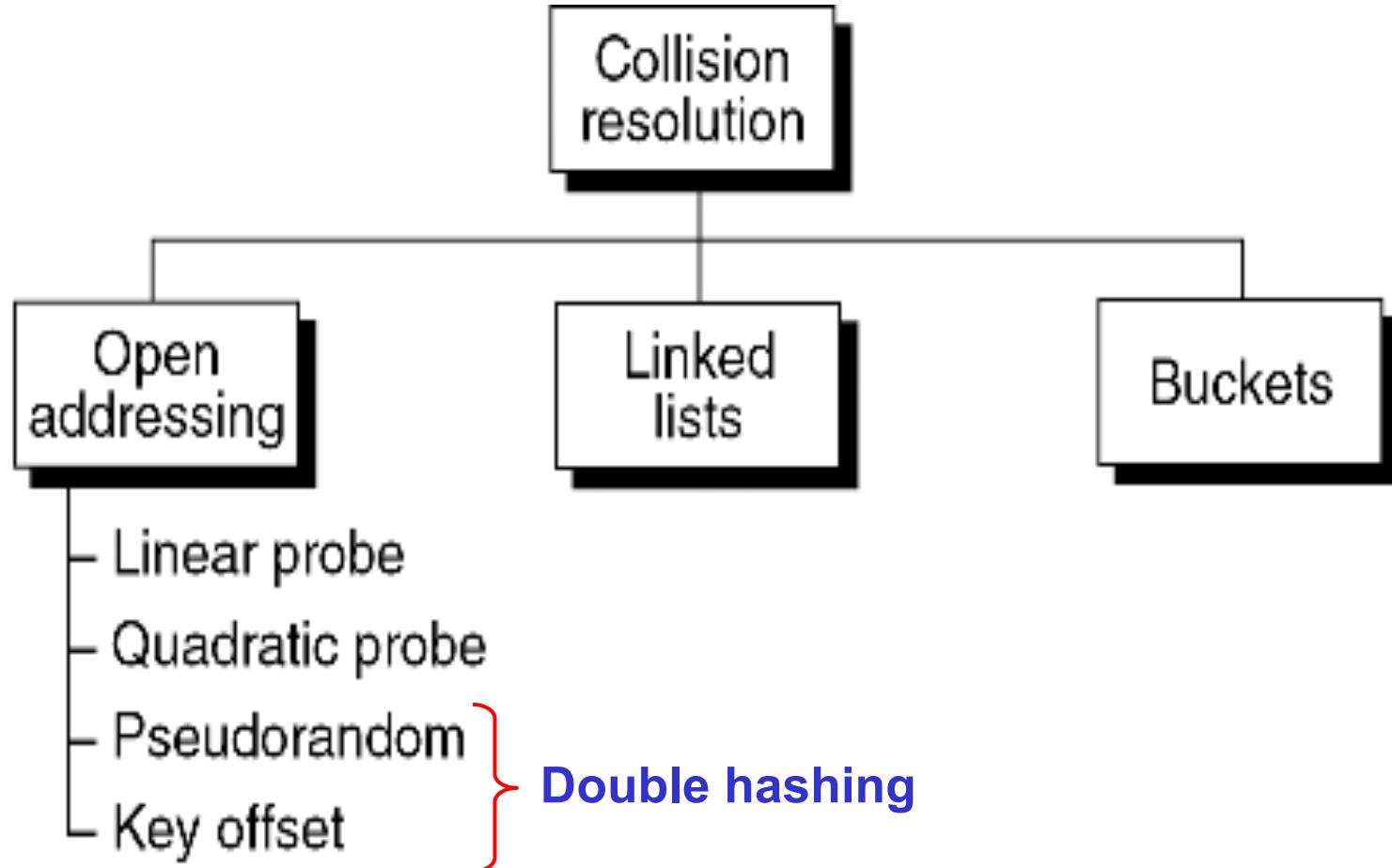
Clustering and Collisions



Hashing Design Principles

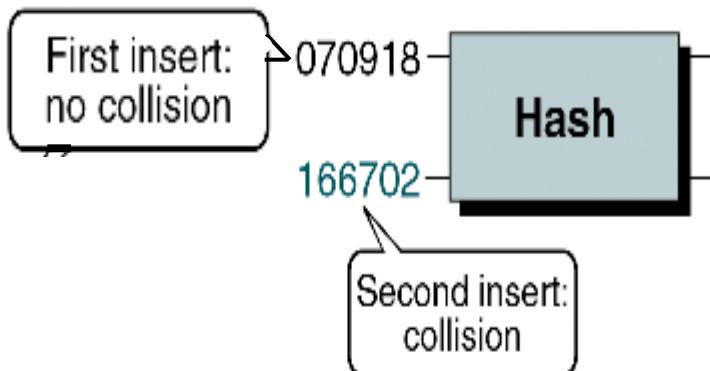
- The *load factor* (# of elements divided by the # of physical elements allocated for the list) no more than 75% full
- Minimize the clustering (primary and secondary clustering)
- Maximize the number of elements to be inserted into a list before a collision occurs
- Number of elements examined in the search must be limited
- Traditional limit difficulties
 - Finding the ends doesn't mean that every element has been tested
 - Examining every element would be extensive time-consuming
 - Some of the collision resolution techniques cannot physically examine all of the elements

Collision Resolution Methods



Linear Probe Collision Resolution

- Disadvantages
 - Not suitable for search, especially after data have been deleted
 - Create primary clustering
- Advantages
 - Simple
 - Suitable for disk address



[000]	379452	Mary Dodd
[001]	070918	Sarah Trapp
[002]	121267	Bryan Devaux
[003]	166702	Harry Eagle
[004]		
[005]		
[006]		
[007]	378845	Patrick Linn
[008]		
⋮		
[305]	160252	Tuan Ngo
[306]	045128	Shouli Feldman

Probe 1
Probe 2

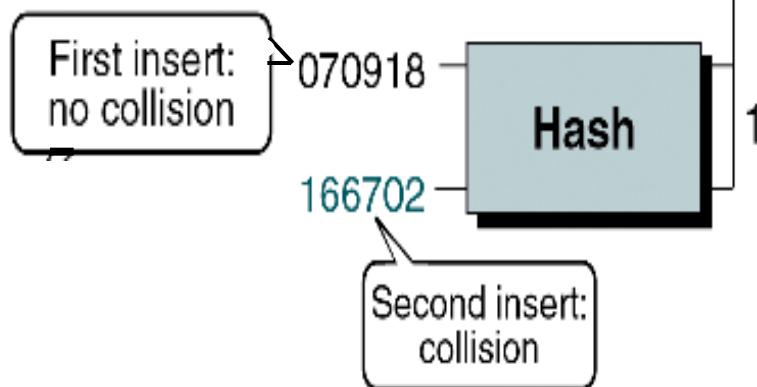
Quadratic Collision Resolution

- Advantage: prevents primary clustering
- Disadvantages
 - Squaring the number is time-consuming → use “add”
 - Impossible to generate a new address for every element
→ make list size prime number

Probe number	Collision location	Probe ² and increment	New address
1	1	$1^2 = 1$	$1 + 1 \rightarrow 02$
2	2	$2^2 = 4$	$2 + 4 \rightarrow 06$
3	6	$3^2 = 9$	$6 + 9 \rightarrow 15$
4	15	$4^2 = 16$	$15 + 16 \rightarrow 31$
5	31	$5^2 = 25$	$31 + 25 \rightarrow 56$
6	56	$6^2 = 36$	$56 + 36 \rightarrow 92$
7	92	$7^2 = 49$	$92 + 49 \rightarrow 41$
8	41	$8^2 = 64$	$41 + 64 \rightarrow 05$
9	5	$9^2 = 81$	$5 + 81 \rightarrow 86$
10	86	$10^2 = 100$	$86 + 100 \rightarrow 86$

Pseudorandom Collision Resolution

- Advantages (double hashing)
 - Prevent primary clustering
- Disadvantages
 - All keys follow one collision resolution path → secondary clustering (same as linear and quadratic probe)



[000]	379452	Mary Dodd
[001]	070918	Sarah Trapp
[002]	121267	Bryan Devaux
[003]		
[004]		
[005]		
[006]		
[007]	378845	Patrick Linn
[008]	166702	Harry Eagle
⋮		
[305]	160252	Tuan Ngo
[306]	045128	Shouli Feldman

A vertical line labeled 'Probe 1' connects the [001] through [008] slots. An arrow points from the [008] slot back up to the [001] slot, indicating the probe sequence. In the [008] slot, the value 166702 is highlighted in blue, and the name Harry Eagle is in black. A callout bubble points to this entry with the text 'Pseudorandom $y = 3x + 5$ '. A red circle highlights the term 'address' at the bottom right of the slide.

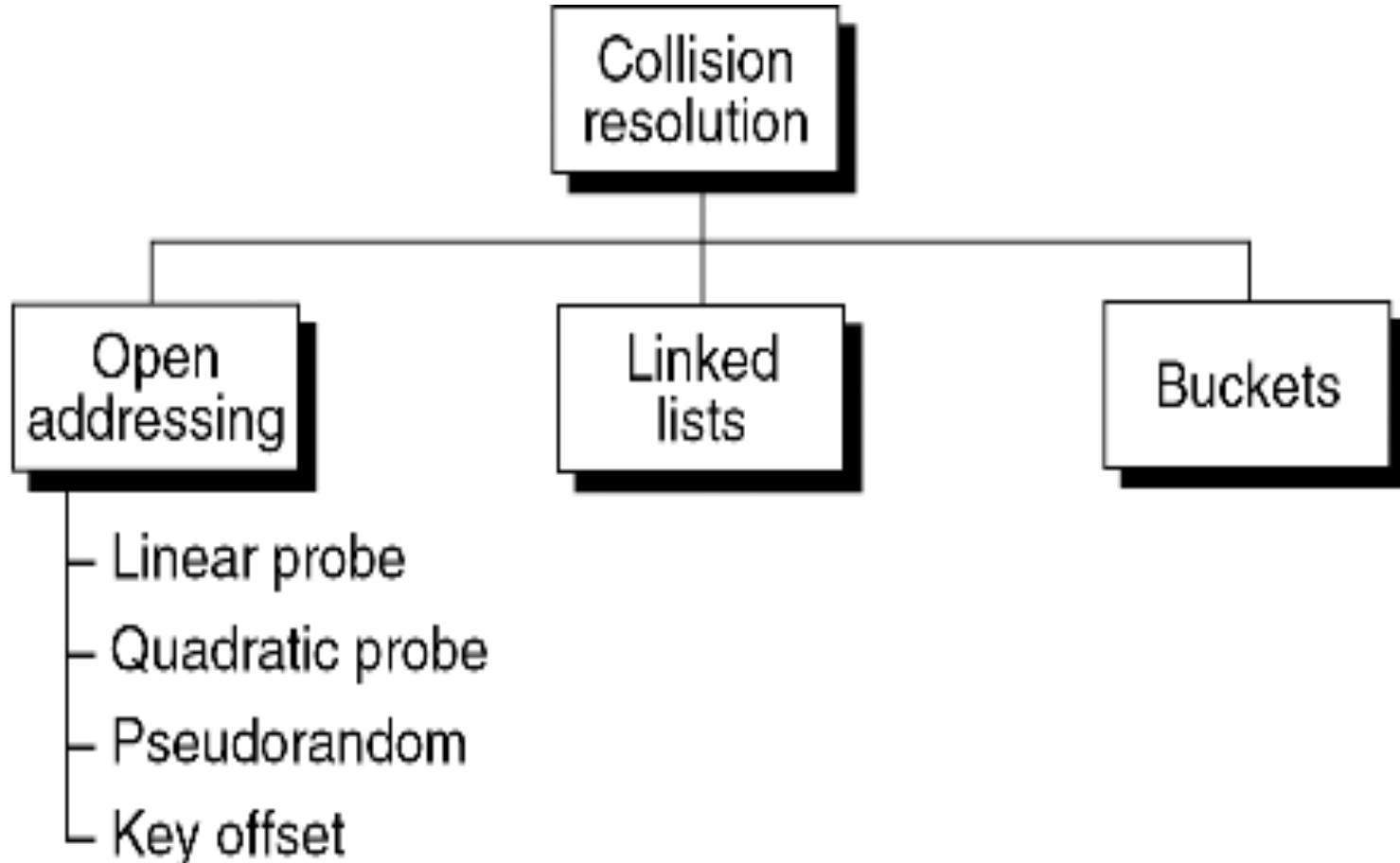
Key-offset Collision Resolution

- Calculates the new address as a function of the **old address** and the **key**
- Advantages
 - Double hashing that produces different collision paths for different keys
- $\text{offset} = \text{key}/\text{listsize}$, $\text{addr.} = ((\text{offset} + \text{old addr.}) \bmod \text{listsize})$
- For example: $\text{key} = 166702 \rightarrow \text{offset} = 166702/307 = 543$
 $\text{addr.} = ((543+001)\bmod 307) = 237$

If 237 were collision $\rightarrow \text{addr.} = ((543+237) \bmod 307) = 166$

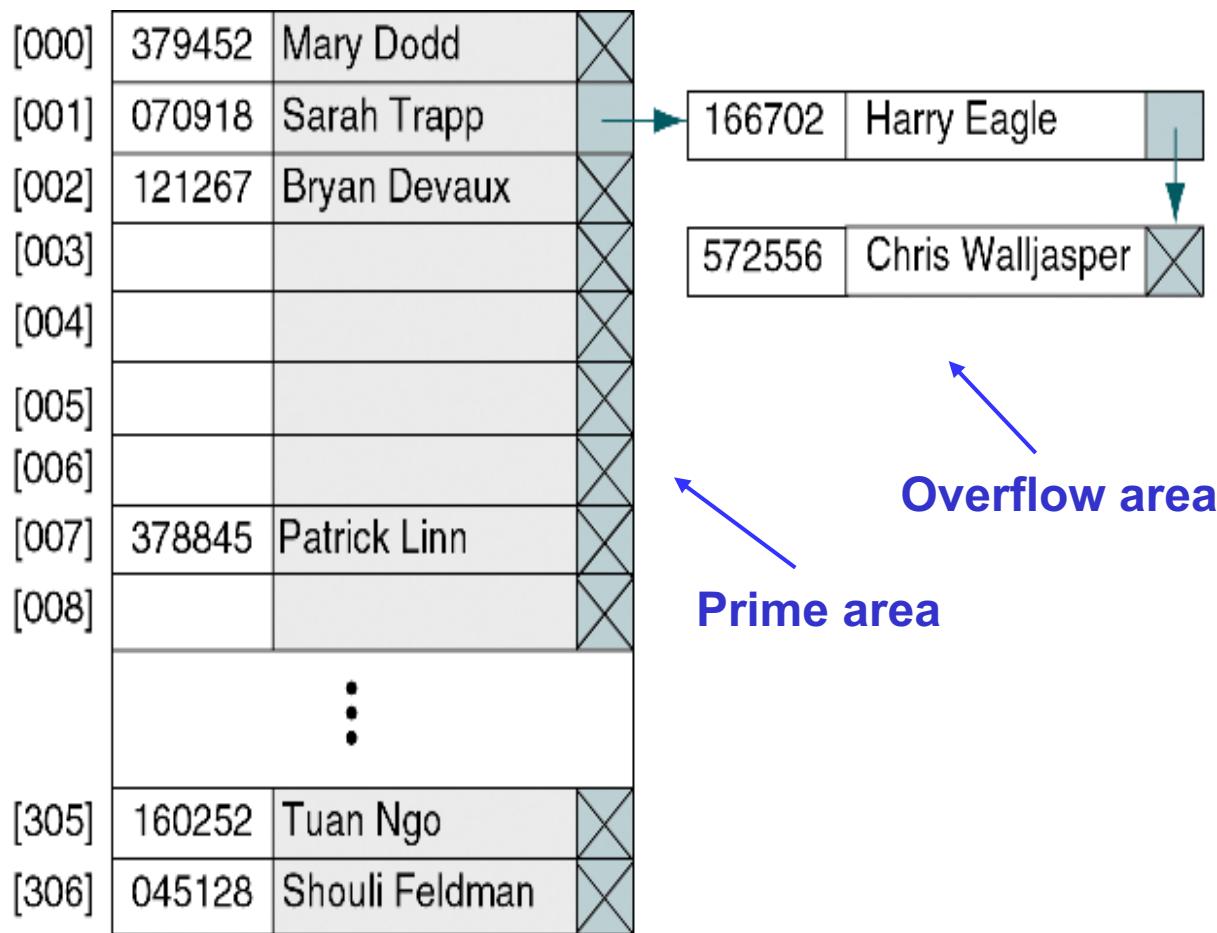
Key	Home address	Key offset	Probe 1	Probe 2
166702	1	543	237	166
572556	1	1865	024	047
067234	1	219	220	132

Collision Resolution Methods



Linked List Collision Resolution

- Advantage: eliminate the drawback of open addressing method in which each collision resolution increase the probability of future collisions



Bucket Collision Resolution

- **Disadvantage:**
 - **Uses significant more space when the buckets are partially empty at given time**
 - **Does not completely resolve the collision problem**

[000]	Bucket 0	379452	Mary Dodd
[001]	Bucket 1	070918	Sarah Trapp
		166702	Harry Eagle
		367173	Ann Giorgis
[002]	Bucket 2	121267	Bryan Devaux
		572556	Chris Walljasper
		:	Linear probe placed here
[307]	Bucket 307	045128	Shouli Feldman

Hashed List Search Example

```
1  /* The telephone lookup program using hashing.  
2   The input is a file of names and phone numbers.  
3   Written by:  
4   Date:  
5 */  
6 #include <stdio.h>  
7 #include <stdlib.h>  
8 #include <string.h>  
9  
10 // Global Declarations  
11 typedef struct  
12 {  
13     char name [31];  
14     char phone[16];  
15 } LISTING;  
16  
17 const int cMax_Size = 53;  
18  
19 // Prototype Declarations  
20 void buildList (LISTING phoneList[], int* last);
```

Hashed List Search Example (cont.)

```
21     void hashSearch (LISTING* phoneList,      int   last);
22     int hashKey       (char*      key,           int   last);
23     int collision    (int       last,          int   locn);
24
25 int main (void)
26 {
27 // Local Definitions
28     LISTING  phoneList[cMax_Size];
29     int       last;
30
31 // Statements
32     printf("Begin Phone Listing\n");
33
34     last = cMax_Size - 1;
35     buildList (phoneList, &last);
36     hashSearch(phoneList,  last);
37
38     printf("\nEnd Phone Listing\n");
39     return 0;
40 } // main
41
```

Hashed List Search Example (cont.)

```
42  /* ===== buildList ===== */
43  Read phone number file and load into array.
44  Pre  phoneList is array to be filled
45  last is index to last element loaded
46  Post array filled
47 */
48 void buildList (LISTING phoneList[], int* last)
49 {
50 // Local Definitions
51 FILE* fpPhoneNums;
52 LISTING aListing;
53 int locn;
54 int cntCol;
55 int end;
56
```

Hashed List Search Example (cont.)

```
57 // Statements
58 fpPhoneNums = fopen ("P13-03.TXT", "r");
59 if (!fpPhoneNums)
60 {
61     printf("Can't open phone file\n");
62     exit (100);
63 } // if
64
65 // Set keys to null
66 for (int i = 0; i <= *last; i++)
67     phoneList[i].name[0] = '\0';
68
```

Hashed List Search Example (cont.)

```
69     while (!feof(fpPhoneNums))
70 {
71     fscanf(fpPhoneNums, " %30[^;]*c %[^;]*c",
72             aListing.name, aListing.phone);
73     locn = hashKey(aListing.name, *last);
74
75     if (phoneList[locn].name[0] != '\0')
76     {
77         // Collision
78         end    = *last;
79         cntCol = 0;
80         while (phoneList[locn].name[0] != '\0'
81                && cntCol++ <= *last)
82             locn = collision(*last, locn);
83
84         if (phoneList[locn].name[0] != '\0')
85         {
86             printf("List full. Not all read.\n");
87             return;
88         } // if full list
89     } // if collision
90     phoneList[locn] = aListing;
91 } // while
92 return;
93 } // buildList
```

Hashed List Search Example (cont.)

```
94
95  /* ===== hashKey =====
96   Given key, hash key to location in list.
97   Pre phoneList is hash array
98   last is last index in list
99   key is string to be hashed
100  Post returns hash location
101 */
102 int hashKey (char* key, int last)
103 {
104 // Local Definitions
105     int addr;
106     int keyLen;
107 }
```

Hashed List Search Example (cont.)

```
108 // Statements
109     keyLen = strlen(key);
110     addr    = 0;
111
112     for (int i = 0; i < keyLen; i++)
113         if (key[i] != ' ')
114             addr += key[i];
115     return (addr % last + 1);
116 } // hashKey
```

Hashed List Search Example (cont.)

```
117
118 /* ===== collision =====
119     Have a collision. Resolve.
120     Pre  phoneList is hashed list
121         last is index of last element in list
122         locn is address of collision
123     Post returns next address in list
124 */
125 int collision (int  last, int  locn)
126 {
127 // Statements
128     return locn < last ? ++locn : 0;
129 } // collision
130
```

Hashed List Search Example (cont.)

```
131 /* ===== hashSearch =====
132   Prompt user for name and lookup in array.
133   Pre phoneList has been initialized
134   Post User requested quit
135 */
136 void hashSearch (LISTING* phoneList, int last)
137 {
138 // Local Definitions
139   char srchName[31];
140   char more;
141   int locn;
142   int maxSrch;
143   int cntCol;
144 }
```

Hashed List Search Example (cont.)

```
145 // Statements
146     do
147     {
148         printf("Enter name: ");
149         scanf ("%s", srchName);
150
151         locn = hashKey (srchName, last);
152         if (strcmp(srchName, phoneList[locn].name) != 0)
153         {
154             // treat as collision
155             maxSrch = last;
156             cntCol = 0;
157             while (strcmp (srchName,
158                             phoneList[locn].name) != 0
159                         && cntCol++ <= maxSrch)
160                 locn = collision(last, locn);
161         } // if
162
163         // Test for success
164         if (strcmp (srchName, phoneList[locn].name) == 0)
```

Hashed List Search Example (cont.)

```
165         printf("%-32s (%02d) %-15s\n",
166                 phoneList[locn].name,
167                 locn,
168                 phoneList[locn].phone);
169     else
170         printf("%s not found\n", srchName);
171
172     printf("\nLook up another number <Y/N>? ");
173     scanf (" %c", &more);
174 } while (more == 'Y' || more == 'y');
175 } // hashSearch
```

Hashed List Search Example (cont.)

Results:

```
Begin Phone Listing
Enter name: Julie
Julie (38) (555) 916-1212

Look up another number <Y/N>? y
Enter name: Chris
Chris (39) (555) 946-2859

Look up another number <Y/N>? y
Enter name: Wyan
Wyan (52) (555) 866-1234

Look up another number <Y/N>? y
Enter name: Wayn
Wayn (0) (555) 345-0987

Look up another number <Y/N>? y
Enter name: Bill
Bill not found

Look up another number <Y/N>? n
End Phone Listing
```

幾個分別了多年的同學相約去拜訪大學的老師。老師很高興，問他們生活得怎麼樣。不料，一句話就勾出了大家的滿腹牢騷，大家紛紛訴說著生活的不如意：工作壓力大呀，生活煩惱多呀，做生意的商戰失利呀，當官的仕途受阻呀…彷彿大家都成了時代的棄兒。

老師笑而不語，從廚房拿出一大堆杯子，擺在茶几上。

這些杯子各式各樣，形態各異，有瓷器的，有玻璃的，有塑膠的，有的杯子看起來豪華而高貴，有的則顯得普通而簡陋……

老師說：「我就不把你們當客人看了。你們要是渴了，就自己倒水喝吧。」

七嘴八舌，大家說得口乾舌燥了，便紛紛拿了自己看中的杯子倒水喝。

等我們手裏都端了一杯水時，老師又發言了。

他指著茶几上剩下的杯子說：「你們有沒有發現，你們手裏的杯子是最好看最別緻的杯子，

而像這些塑膠杯沒有人選中它。」

當然，我們並不覺得奇怪，誰都希望自己拿著的是一隻好看的杯子。

老師說：「這就是你們煩惱的根源」

大家需要的是水，而非杯子，但我們有意無意地會去選擇漂亮的杯子。

這就如我們的生活---如果生活是水的話，那麼，工作、金錢、地位 這些東西就是杯子，

它們只是我們盛起生活之水的工具。

其實，杯子的好壞並不影響水的品質。

如果將心思花在杯子上，大家還有心情去品嘗水的苦甜，這不就是自尋煩惱嗎？