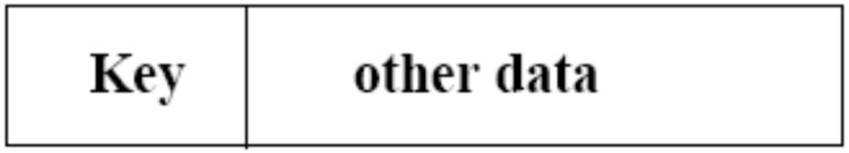
Searching and Hashing

Amiya Ranjan Panda

The Search Problem

- Find items with keys matching a given search key
  - Given an array A, containing n keys, and a search key k, find the index i such as k=A[i]

example of a record



Key	**other data**

Problem: Search

- Given a list of records (elements).
- Each record has an associated key
- Find an efficient algorithm for searching for a record containing a particular key
- Efficiency is quantified in terms of average time analysis (number of comparisons) to retrieve the item

Applications

- Keeping track of customer account information in a bank
  - Search through records to check balances and perform transactions
- Keep track of reservations on flights
  - Search to find empty seats, cancel/modify reservations
- Search engine
  - Looks for all documents containing a given word

Sequential/Linear Search

Step 1 - Read the search element

Step 2 - Compare the search element with the first element in the list

Step 3 - If both are matched, then display "*Given element is found!!!*" and terminate the function

Step 4 - If both are not matched, then compare search element with the next element in the list

Step 5 - Repeat steps 3 and 4 until search element is compared with last element in the list

Step 6 - If last element in the list also doesn't match, then display "*Element is not found!!!*" and terminate the function.

Sequential Search

```
int search(int arr[], int n, int ele) {
   int i;
   for (i = 0; i < n; i++)
      if(arr[i] == ele)
         return i;
   return -1;
}
```

Sequential Search Analysis

- Time complexity is measured on the basis of number of comparisons performed.
- The items have been placed randomly into the list.
- In other words, the probability that the item we are looking for is in any particular position is exactly the same for each position of the list.
- If the item is not in the list, the only way to know it is to compare it against every item present.
- On average, we will find the item about halfway into the list; that is, we will compare against n/2 items.

Worst Case Time for Sequential Search

- For an array of *n* elements, the worst case time for sequential search requires *n* comparisons: O(*n*).
- Consider cases where it must loop over all *n* records:
  - desired record appears in the last position of the array
  - desired record does not appear in the array at all

Average Case for Sequential Search

Assumptions:

1. All keys are equally likely in a search
2. Always search for a key that is in the array

Example:

- Suppose an array of 10 records.
- If search for the first record, then it requires 1 array access; if the second, then 2 array accesses. etc.

The average of all these searches is:

$(1+2+3+4+5+6+7+8+9+10)/10 = 5.5$

Average Case Time for Sequential Search

- Generalize for array size *n.*
- Expression for average-case running time:
- (1+2+…+n)/n = n(n+1)/2n = (n+1)/2
- Therefore, average case time complexity for serial search is O(n).

Comparisons Used in a Sequential Search of an Unordered List

Case	Best Case	Worst Case	Average Case
item is present	O(1)	O(n)	O(n/2)
item is not present	O(n)	O(n)	O(n)

MCQ-1

Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do in order to find the key 18?

(A) 5

(B) 10

(C) 4

(D) 2

(E) 7

MCQ-1

Suppose you are doing a sequential search of the list [15, 18, 2, 19, 18, 0, 8, 14, 19, 14]. How many comparisons would you need to do in order to find the key 18?

(A) 5

(B) 10

(C) 4

(D) 2

(E) 7

Ans: (D) In this case only 2 comparisons were needed to find the key.

MCQ-2

Suppose you are doing a sequential search of the ordered list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18]. How many comparisons would you need to do in order to find the key 13?

(A) 10

(B) 5

(C) 7

(D) 6

(E) 7

MCQ-2

Suppose you are doing a sequential search of the ordered list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18]. How many comparisons would you need to do in order to find the key 13?

(A) 10

(B) 5

(C) 7

(D) 6

(E) 8

Ans: (C) You do not need to search the entire list, since it is ordered you can stop searching when you have compared with a value larger than the key.

Binary Search

- Perhaps it can be done better than O(n) in the average case?
- Assume that there exists an array of records that is sorted. For instance:
  - an array of records with integer keys sorted from smallest to largest (e.g., ID numbers), or
  - an array of records with string keys sorted in alphabetical order (e.g., names).

Binary Search: Iterative

```
int binarySearch(int arr[], int n, int ele)  {
   int low = 0, high = n - 1;
   while (low <= high) {
      int mid = (low + high)/2;
      if (ele == arr[mid])
         return mid;
      else if (ele < arr[mid])
         high = mid - 1;
      else
         low = mid + 1;
   }
   return -1;
}
```

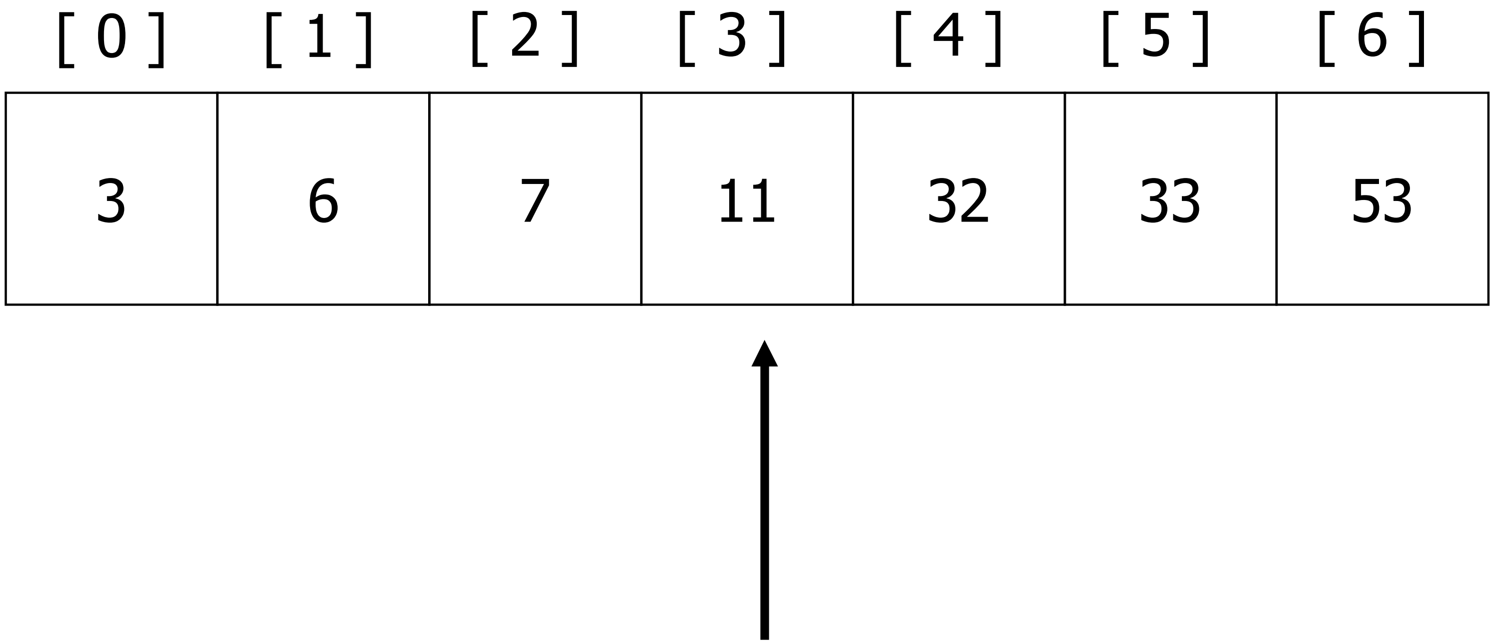
Binary Search

Example: sorted array of integer keys. Target=7.

[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Binary Search

Example: sorted array of integer keys. Target=7.

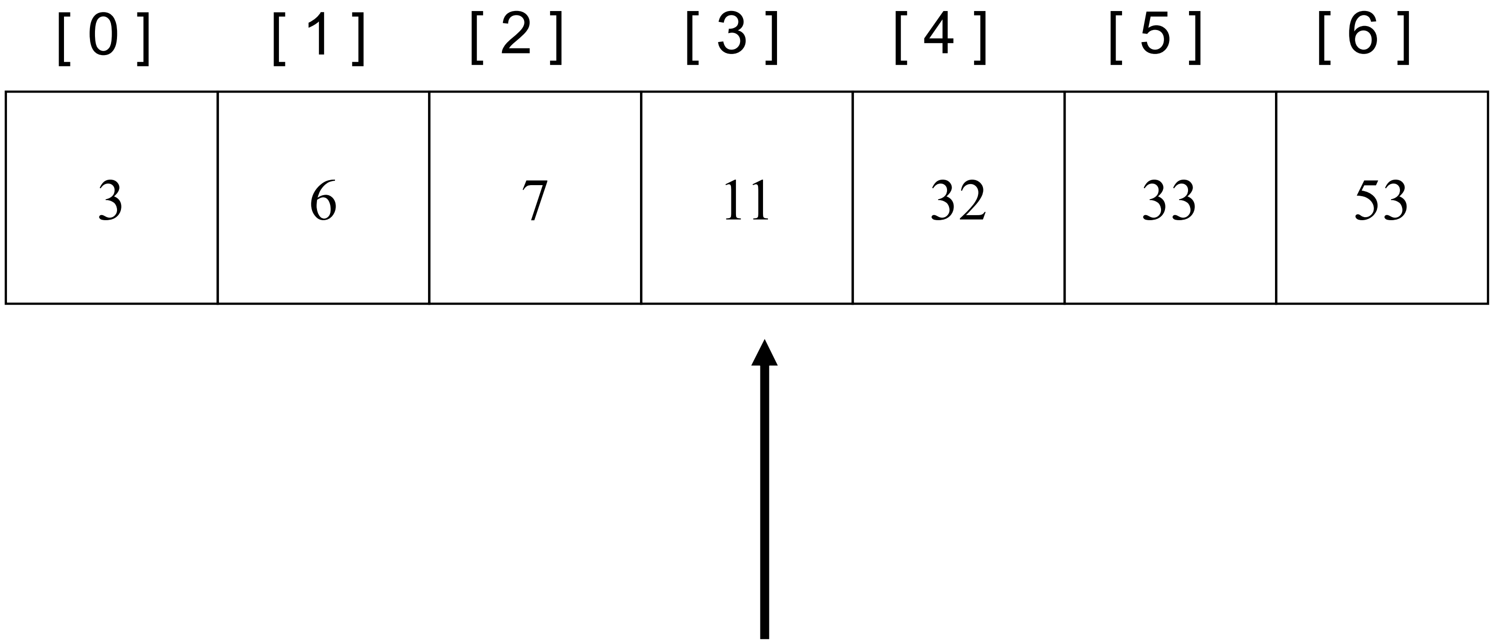


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

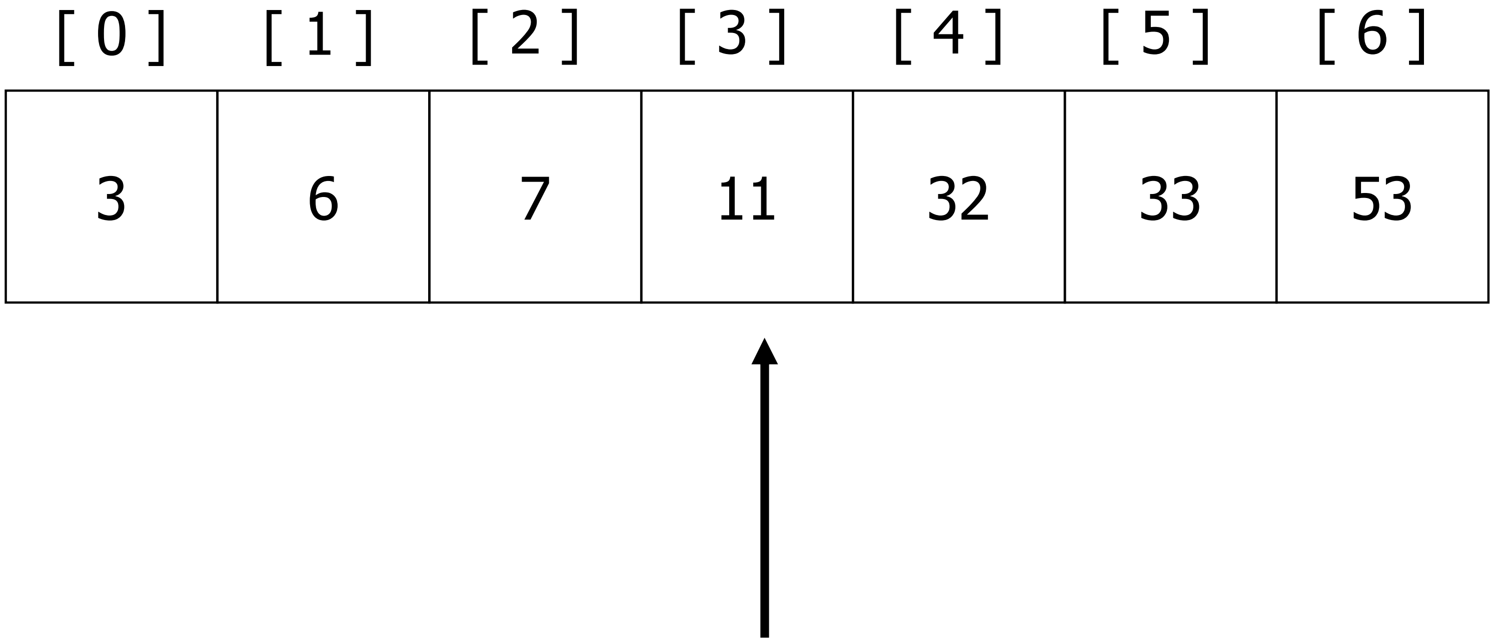


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is target, 7 = midpoint key? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

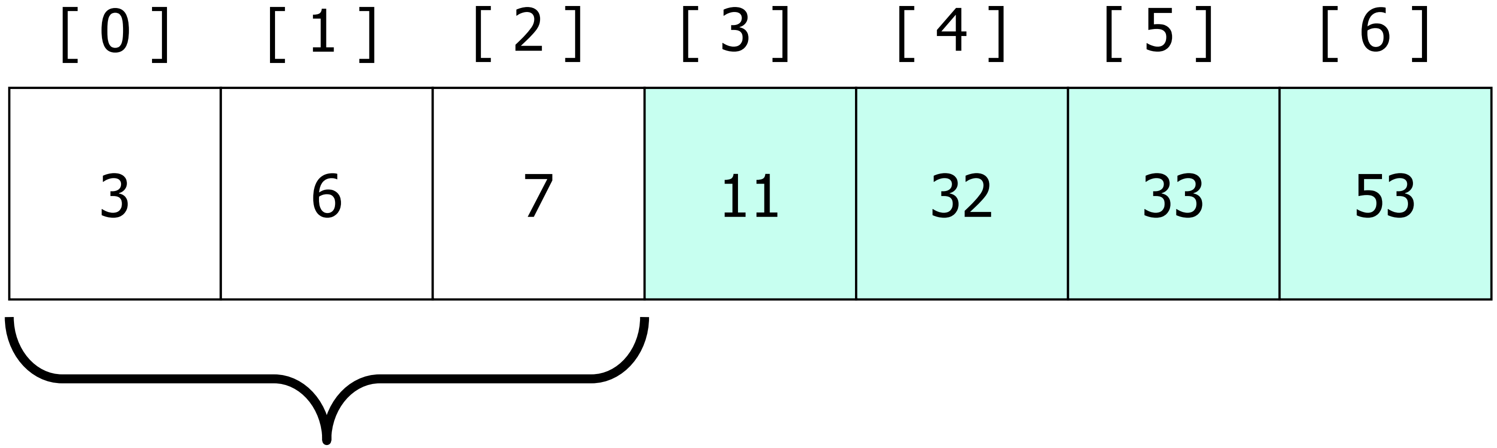


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is target, 7 < midpoint key? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

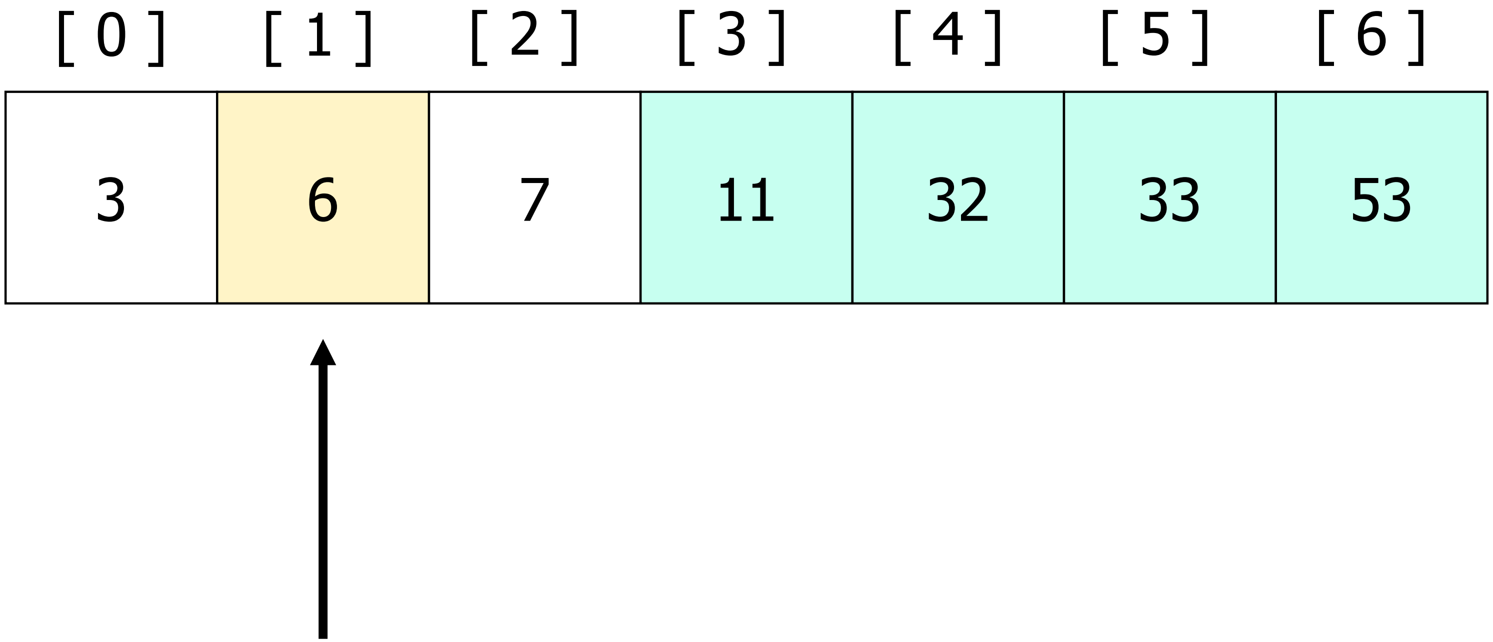


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area before midpoint.

Binary Search

Example: sorted array of integer keys. Target=7.

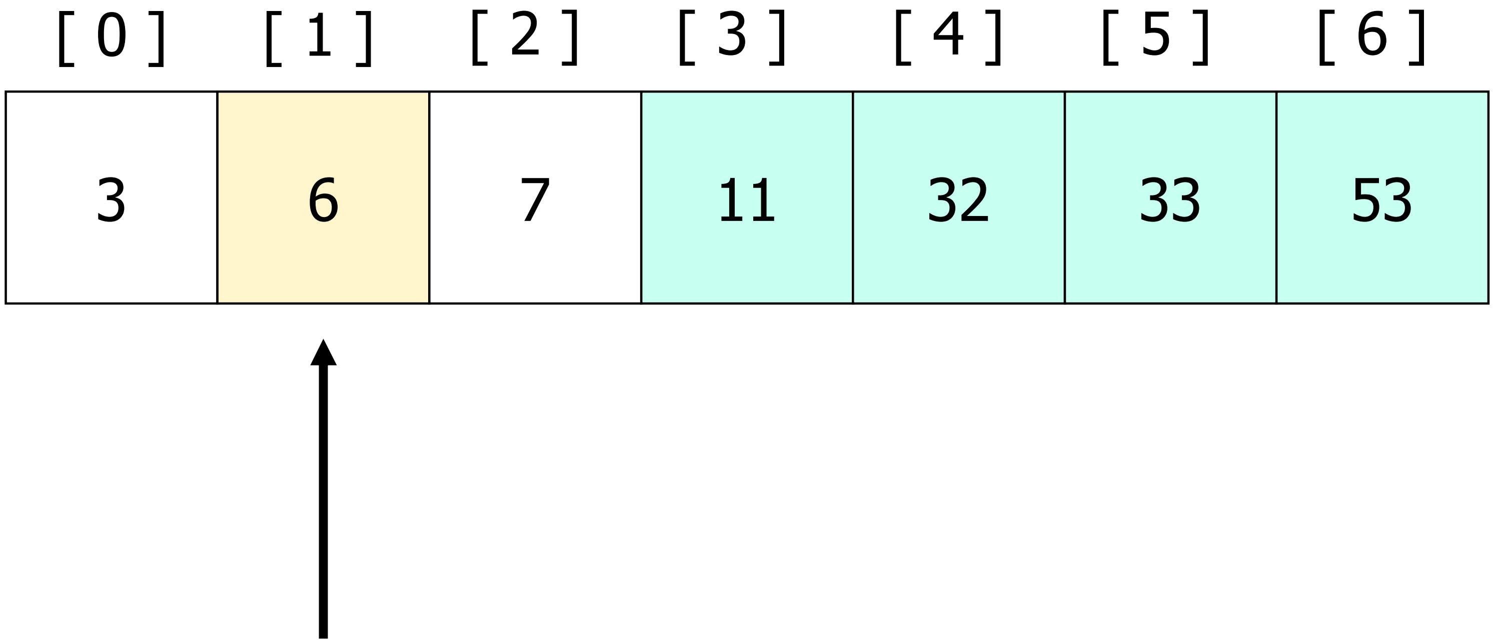


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint

Binary Search

Example: sorted array of integer keys. Target=7.

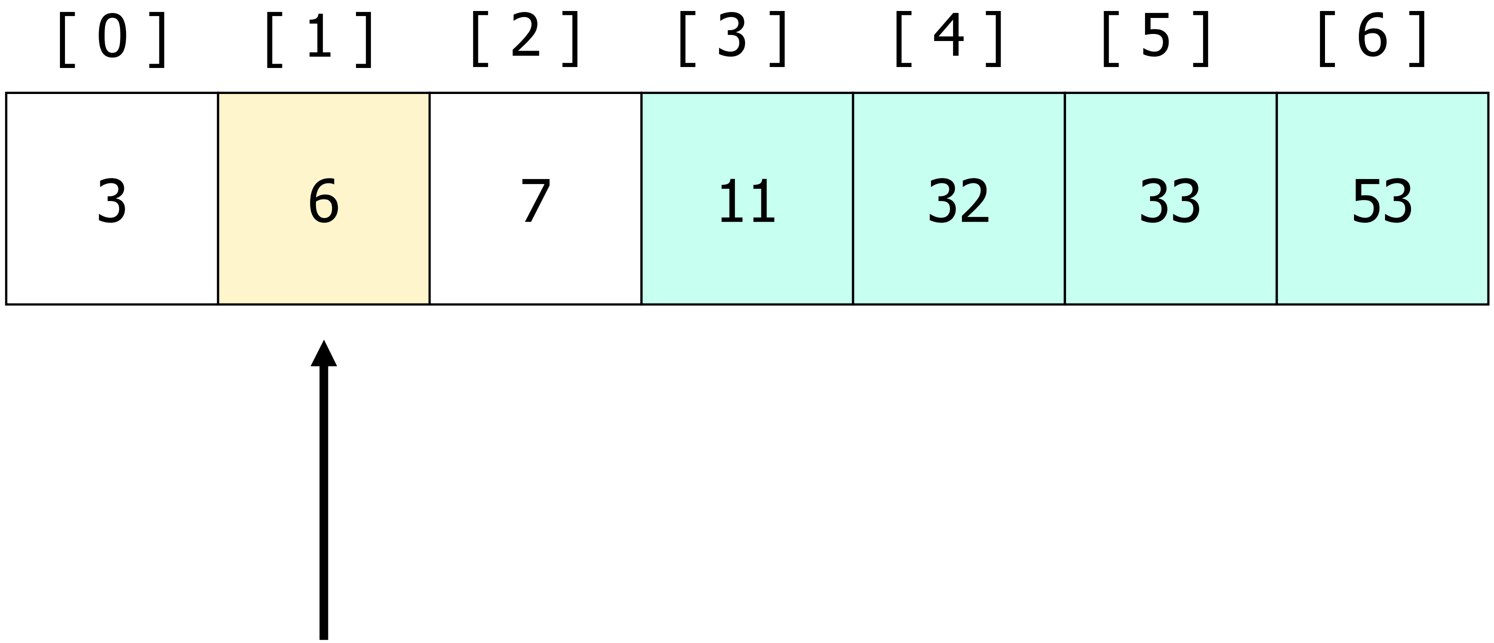


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is target, 7 = key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

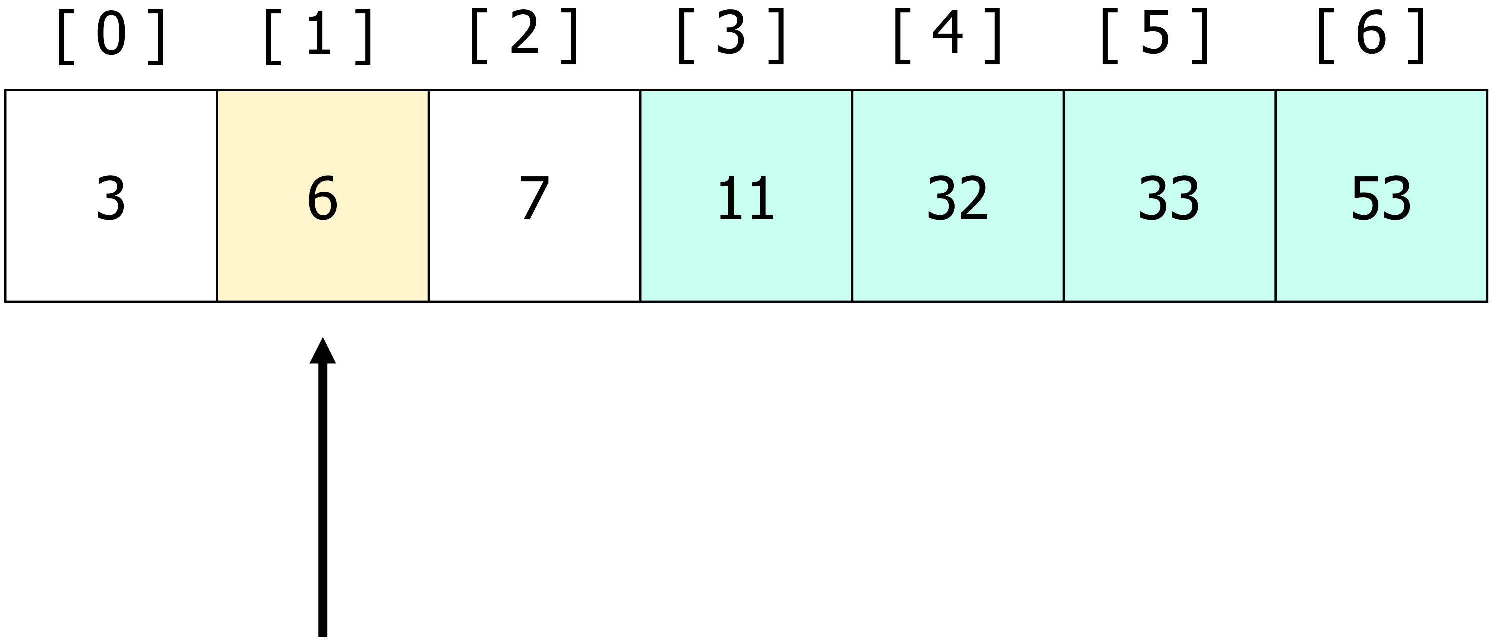


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is target, 7 < key of midpoint? NO.

Binary Search

Example: sorted array of integer keys. Target=7.

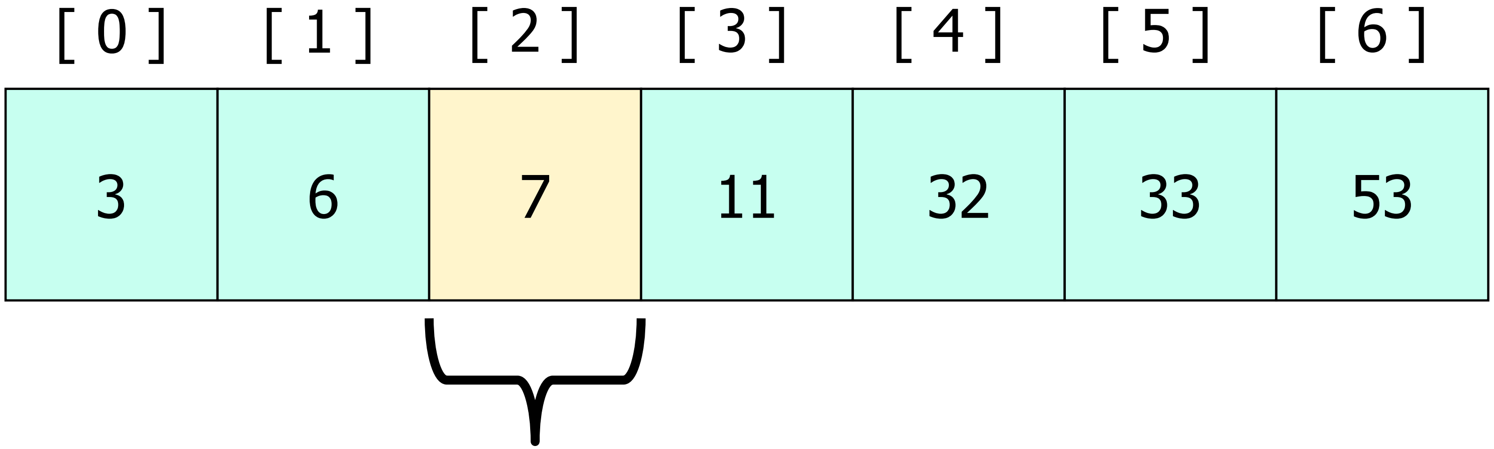


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Is target, 7 > key of midpoint? YES.

Binary Search

Example: sorted array of integer keys. Target=7.

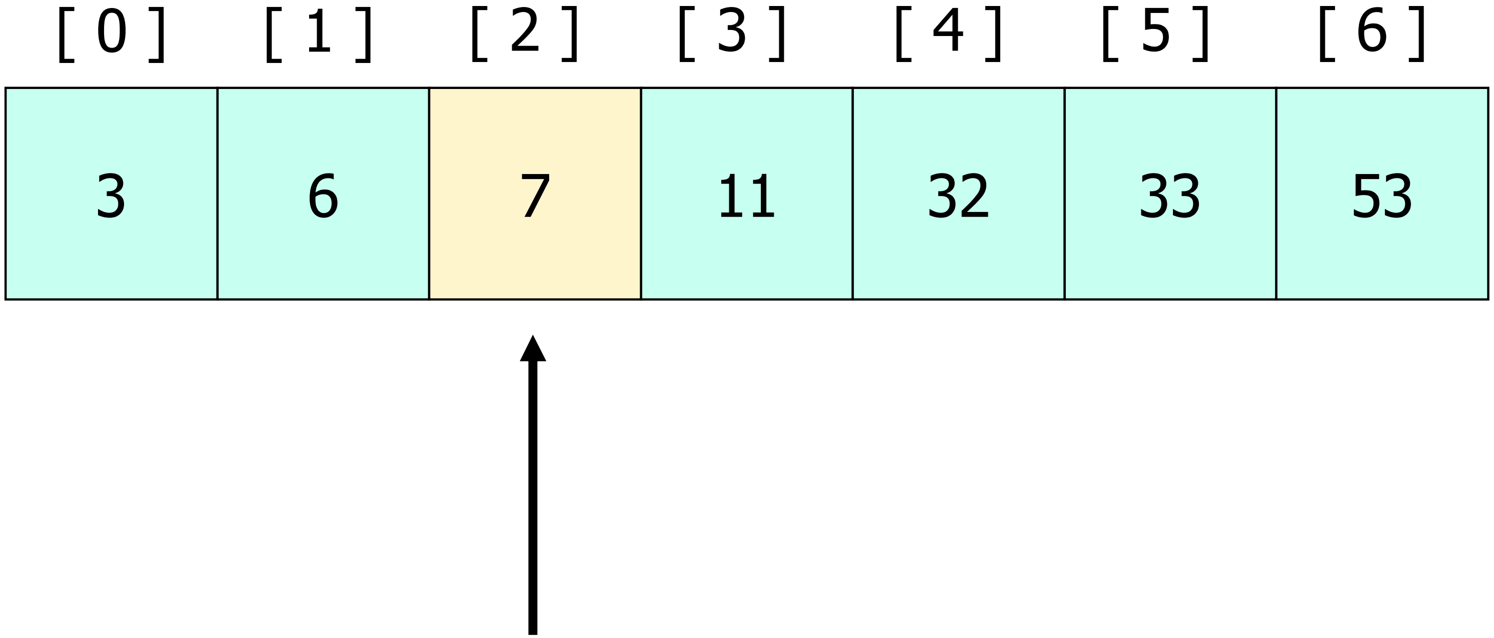


[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Search for the target in the area after midpoint.

Binary Search

Example: sorted array of integer keys. Target=7.



[0]	[1]	[2]	[3]	[4]	[5]	[6]
3	6	7	11	32	33	53

Find approximate midpoint.
Is target, 7 = midpoint key? YES.

Binary Search: Recursive

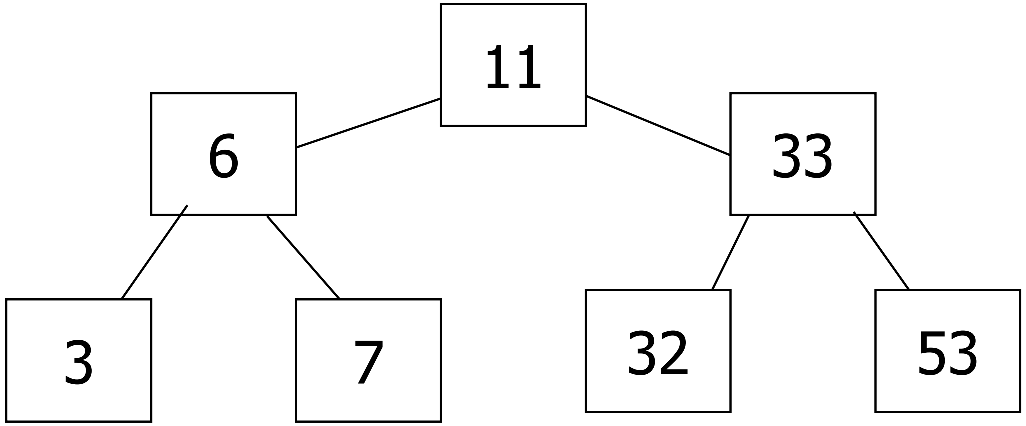
```
int binarySearch(int arr[], int low, int high, int ele) {
   if (low > high)
      return -1;
   int mid = (low + high)/2;
   if (ele == arr[mid])
      return mid;
   else if (ele < arr[mid]) {
        return binarySearch(arr, low, mid-1, ele);
   else
      return binarySearch(arr, mid+1, high, ele);
}
```

Relation to Binary Search Tree

Array of previous example:

3	6	7	11	32	33	53

Corresponding complete binary search tree

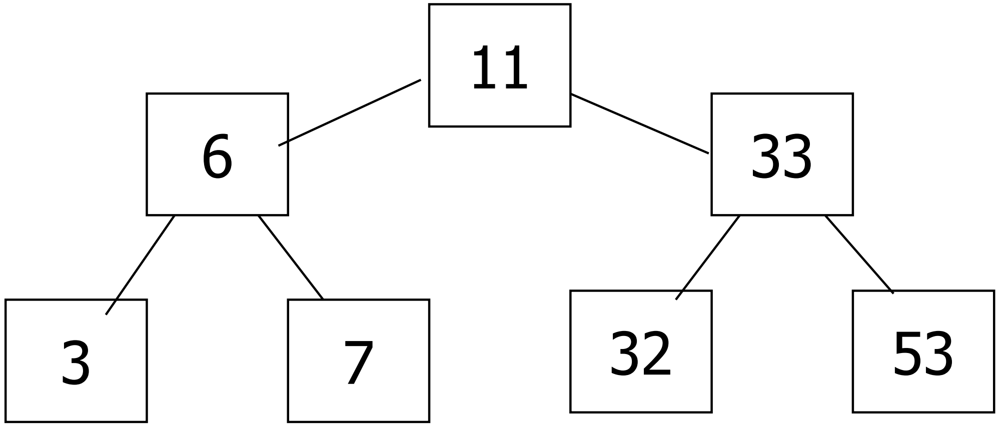


Search for key = 7

Find midpoint:

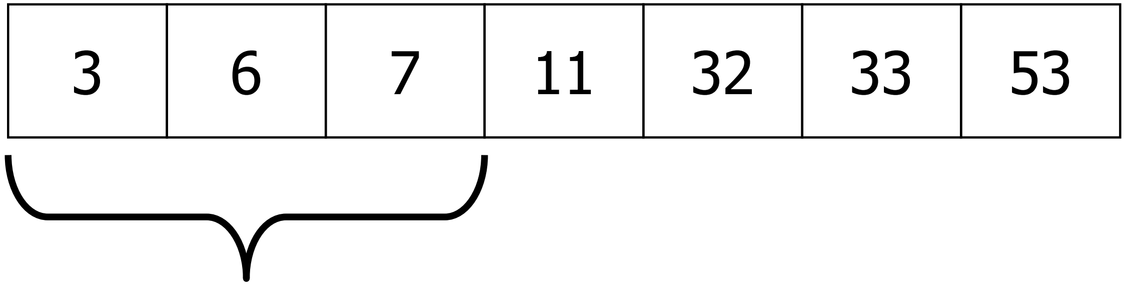
3	6	7	11	32	33	53

Start at root:



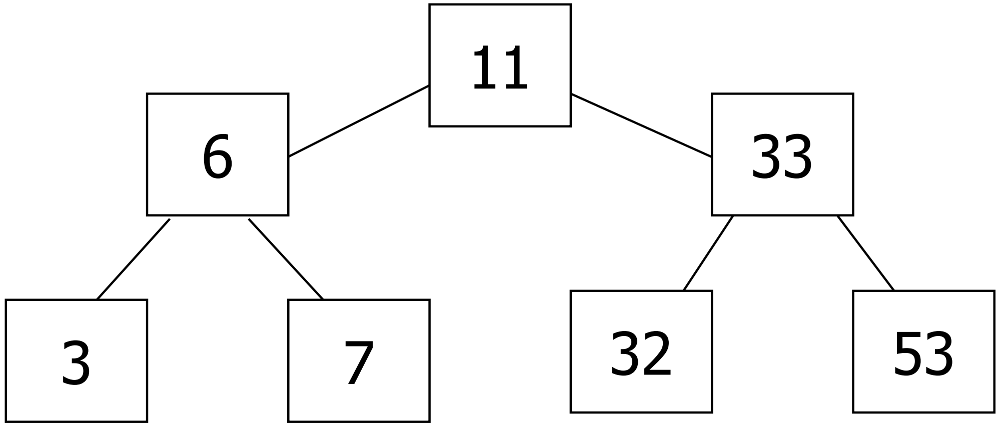
Search for key = 7

Search left subarray:



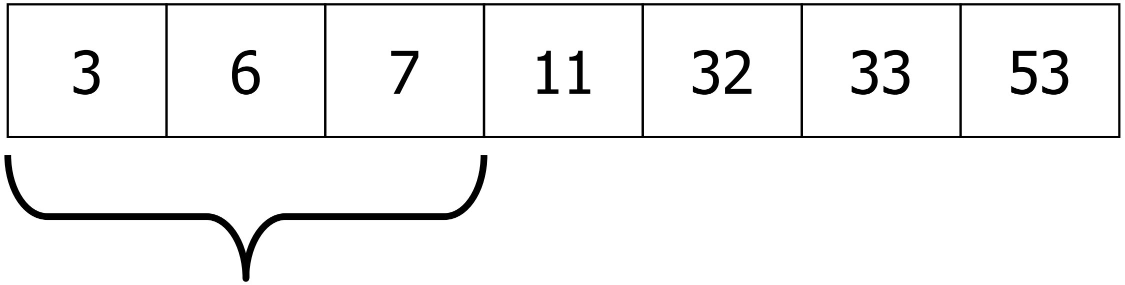
3	6	7	11	32	33	53

Search left subtree:



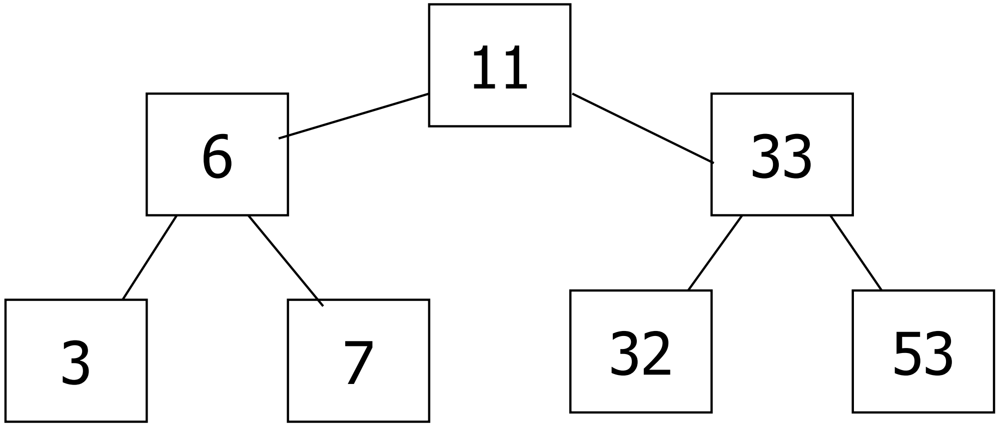
Search for key = 7

Find approximate midpoint of subarray:



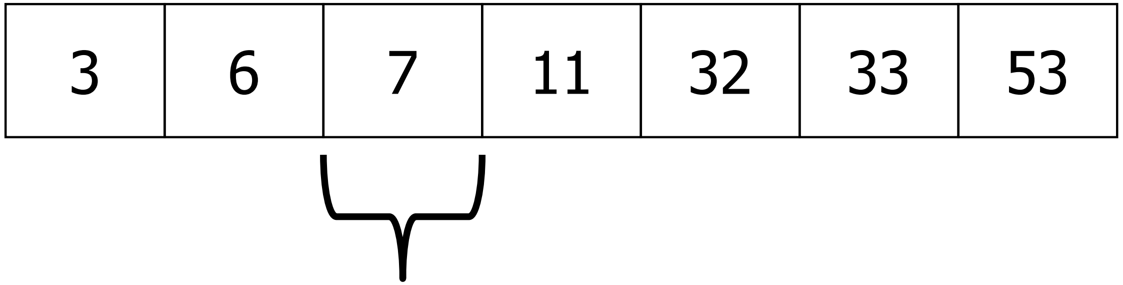
3	6	7	11	32	33	53

Visit root of subtree:



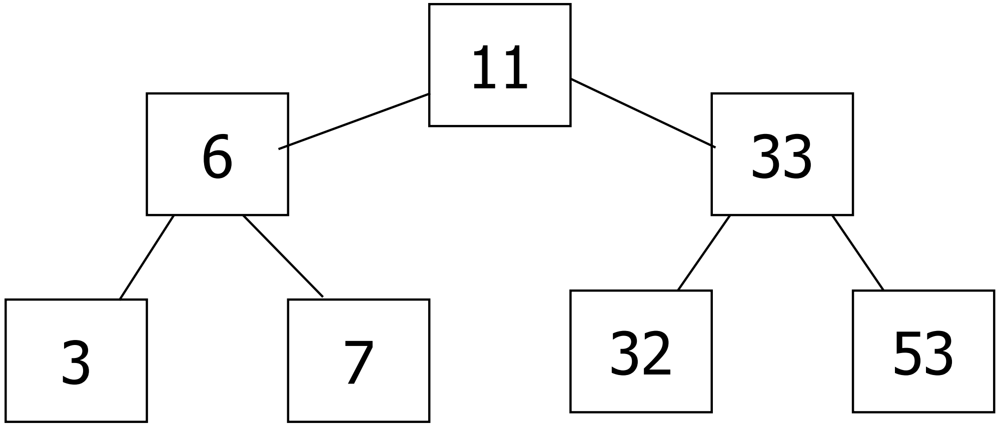
Search for key = 7

Search right subarray:



3	6	7	11	32	33	53

Search right subtree:



Binary Search: Analysis

- Worst case complexity?
- What is the maximum depth of recursive calls in binary search as function of *n*?
- Each level in the recursion, split the array in half (divide by two).
- Therefore maximum recursion depth is floor($\log_2 n$) and worst case = O($\log_2 n$).
- Average case is also = O($\log_2 n$).

Can we do better than O($\log_2 n$)?

- Average and worst case of serial/ linear search = O(n)
- Average and worst case of binary search = O($\log_2 n$)

- Can we do better than this?

  YES. Use a hash table!

What is Hashing

- Hashing is a technique that is used to uniquely search a given key from a group of similar keys.
- The keys are stored in a data structure called hash table.
- Idea: To distribute entries (key/value pairs) uniformly across the hash table.
- Hashing uses a special function called the Hash function.
  - The efficiency of searching a key depends of the efficiency of the hash function used.

Hash Tables

- A hash table is a data structure that is used to store keys/value pairs.
- It uses a hash function to compute an index into an array in which an element will be inserted or searched.
- By using a good hash function, hashing can work well.
- Under reasonable assumptions, the average time required to search for an element in a hash table is O(1).

Hash Tables

Idea:

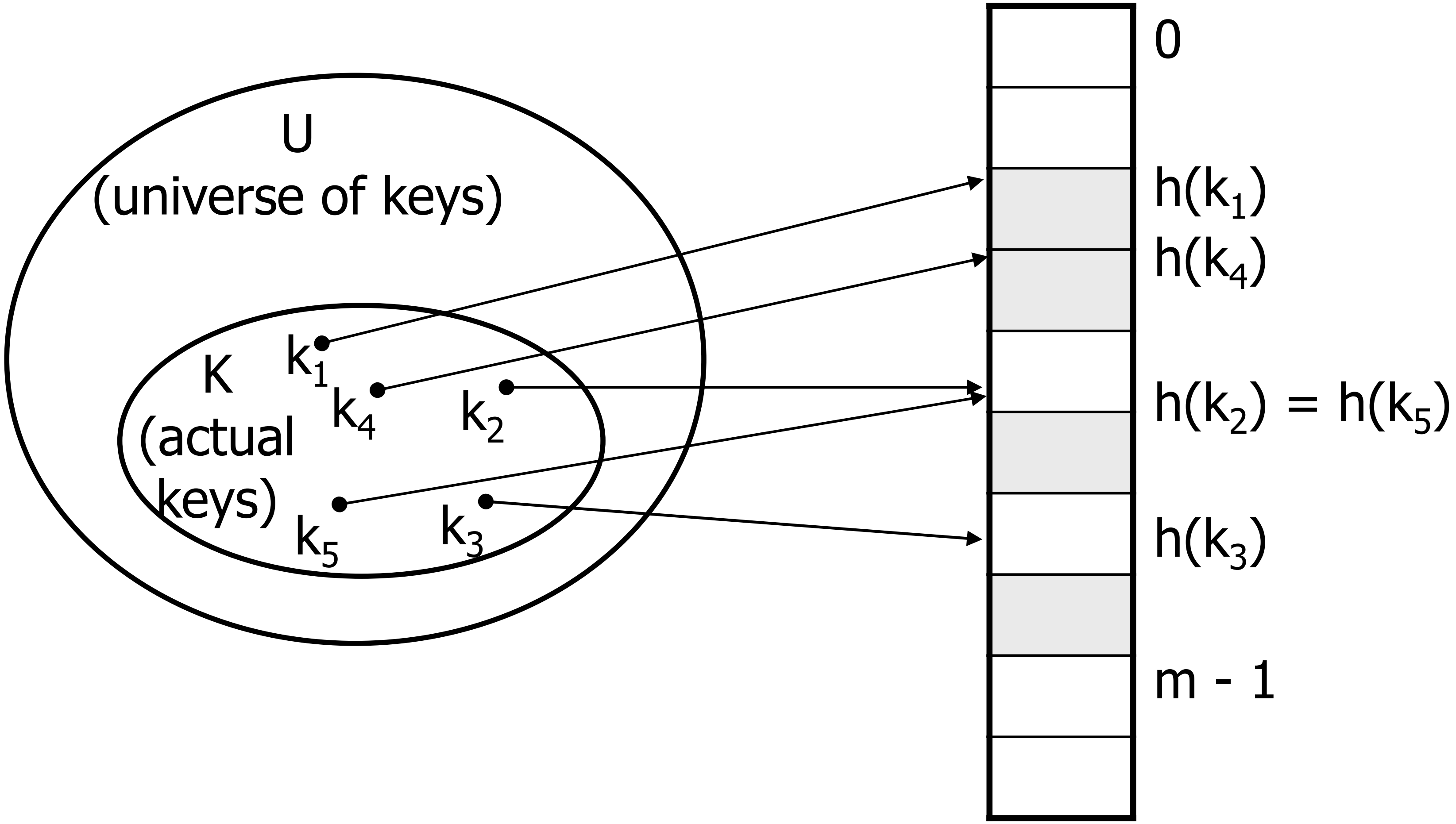
- Use a function *h* to compute the slot for each key in the hash table.
- Store the element in slot *h*(k).

- A hash function *h* transforms a key into an index in a hash table T[0…m-1]:

  $h : U \rightarrow \{0, 1, . . . , m - 1\}$

- Say, k hashes to slot h(k)
- Advantages:
  - Reduce the range of array indices handled: m instead of |U|
  - Storage is also reduced.

Example: Hash Table



Another Example

Suppose that the keys are nine-digit social security numbers

Possible hash function

$h(ssn) = sss$ mod 100 (last 2 digits of ssn)

e.g., if $ssn = 10123411$ then $h(10123411) = 11)$

Hash Functions

- A hash function transforms a key into a table address
- What makes a good hash function?
  1. Easy to compute
  2. Approximates a random function: for every input, every output is equally likely (simple uniform hashing)
- In practice, it is very hard to satisfy the simple uniform hashing property
  - i.e., it is not known in advance the probability distribution that keys are drawn from

Good Approaches for Hash Functions

- Minimize the chance that closely related keys hash to the same slot
  - Strings such as pt and pts should hash to different slots
- Derive a hash value that is independent from any patterns that may exist in the distribution of the keys.

The Division Method

- Idea:
  - Map a key k into one of the m slots by taking the remainder of k divided by m.

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

- Advantage:
  - fast, requires only one operation
- Disadvantage:
  - Certain values of m are bad, e.g.,
    - power of 2
    - non-prime numbers

Example - The Division Method

- If $m = 2^p$, then h(k) is just the least significant p bits of k
  - $p = 1 \Rightarrow m = 2$

    $\Rightarrow$ h(k) = {0, 1} , least significant 1 bit of k
  - $p = 2 \Rightarrow m = 4$

    $\Rightarrow$ h(k) = {0, 1, 2, 3} , least significant 2 bits of k
- Choose m to be a prime, not close to a power of 2
  - Column 2: k mod 97
  - Column 3: k mod 100

	m 97	m 100
16838	57	38
5758	35	58
10113	25	13
17515	55	15
31051	11	51
5627	1	27
23010	21	10
7419	47	19
16212	13	12
4086	12	86
2749	33	49
12767	60	67
9084	63	84
12060	32	60
32225	21	25
17543	83	43
25089	63	89
21183	37	83
25137	14	37
25566	55	66
26966	0	66
4978	31	78
20495	28	95
10311	29	11
11367	18	67

Digit Extraction Method

- Digits are extracted from key and used as address.
- Example: Six digit employee number is used to hash three digit address: range (000-999)
- Select first, third and fourth digits use them as address:

  379452-394

  121267-112

  378845-388

  160252-102

  045128-051

Midsquare Method

- Key is squared and the address is selected from the middle of the squared number.
- Example: Given a key of 9452, midsquare address calculation is shown using four digit address (0000-9999)
- 9452^2=89340304: address is 3403
- Select first three digits and then use midsquare method

379452 : 379^2 = 143641 - 364

121267 : 121^2 = 014641 - 464

378845 : 378^2 = 142884 - 288

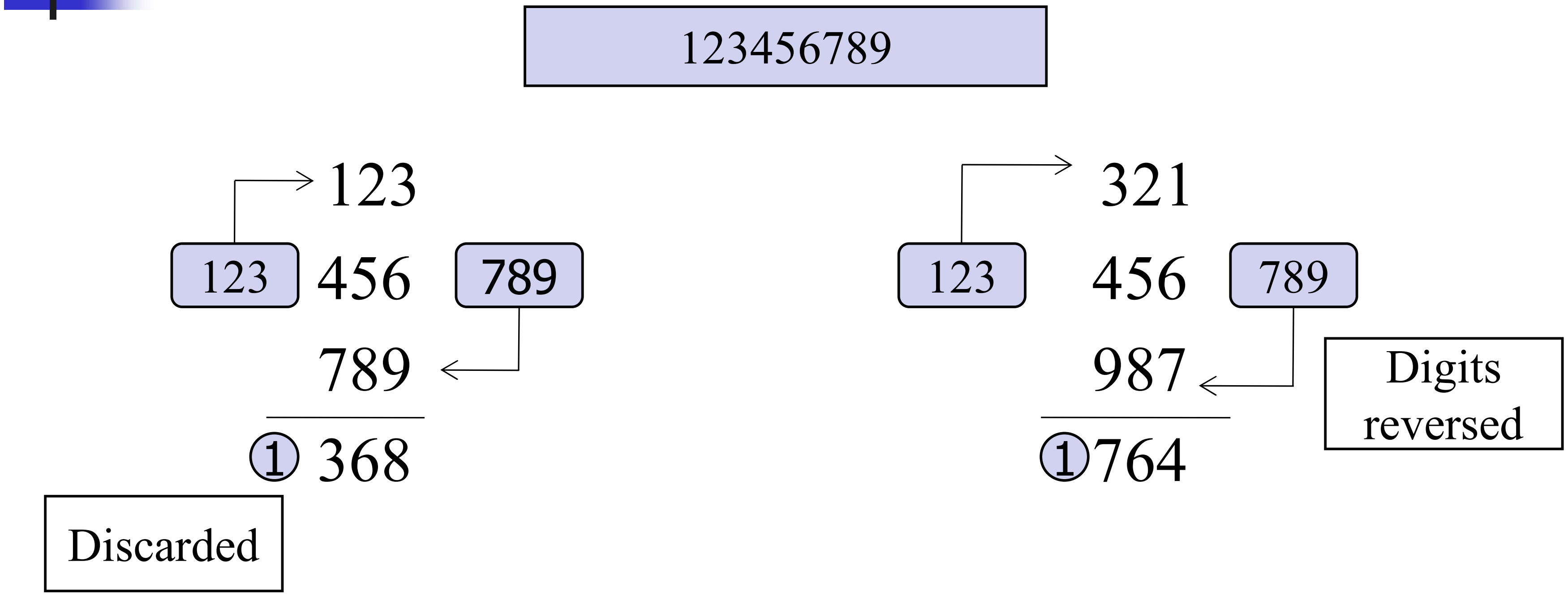
160252 : 160^2 = 025600 - 560

045128 : 045^2 = 002025 - 202

Folding Methods

- Breaks up a key value into precise segments that are added to form a hash value
- Two folding methods:
  - Fold shift
  - Fold boundary
- Fold shift: key value is divided into parts whose size matches size of required address
- Left and right parts are shifted and added with middle part
- Fold boundary: left and right numbers are folded on a fixed boundary between them and center number.

Hash Fold examples



123456789

123
123 456 789
789
1 368
Discarded

321
123 456 789
987
Digits reversed
1764

Fold Shift

Fold boundary

Rotation Method

- Rotate the numbers according to their number of ciphers (123 gets 312, 1234 gets 4123)
- Rotating last character to front of key and minimize the effect.
- Rotation is used in combination with folding and pseudorandom hashing.
- Example: Consider case of six digit employee number that is used in large company.

600101	600101	160010
600102	600102	260010
600103	600103	360010
600104	600104	460010
600105	600105	560010
Original Key	Rotation	Rotated Key

Pseudorandom Hashing

- Key is used as seed in pseudorandom number generator and the random number is scaled into possible address range using modulo-division.
- Pseudorandom number generator generates same number of series.
- A common random number generator is y = ax + c.
- Example:

  Assume a=17, c=7, x=121267, Prime number=307

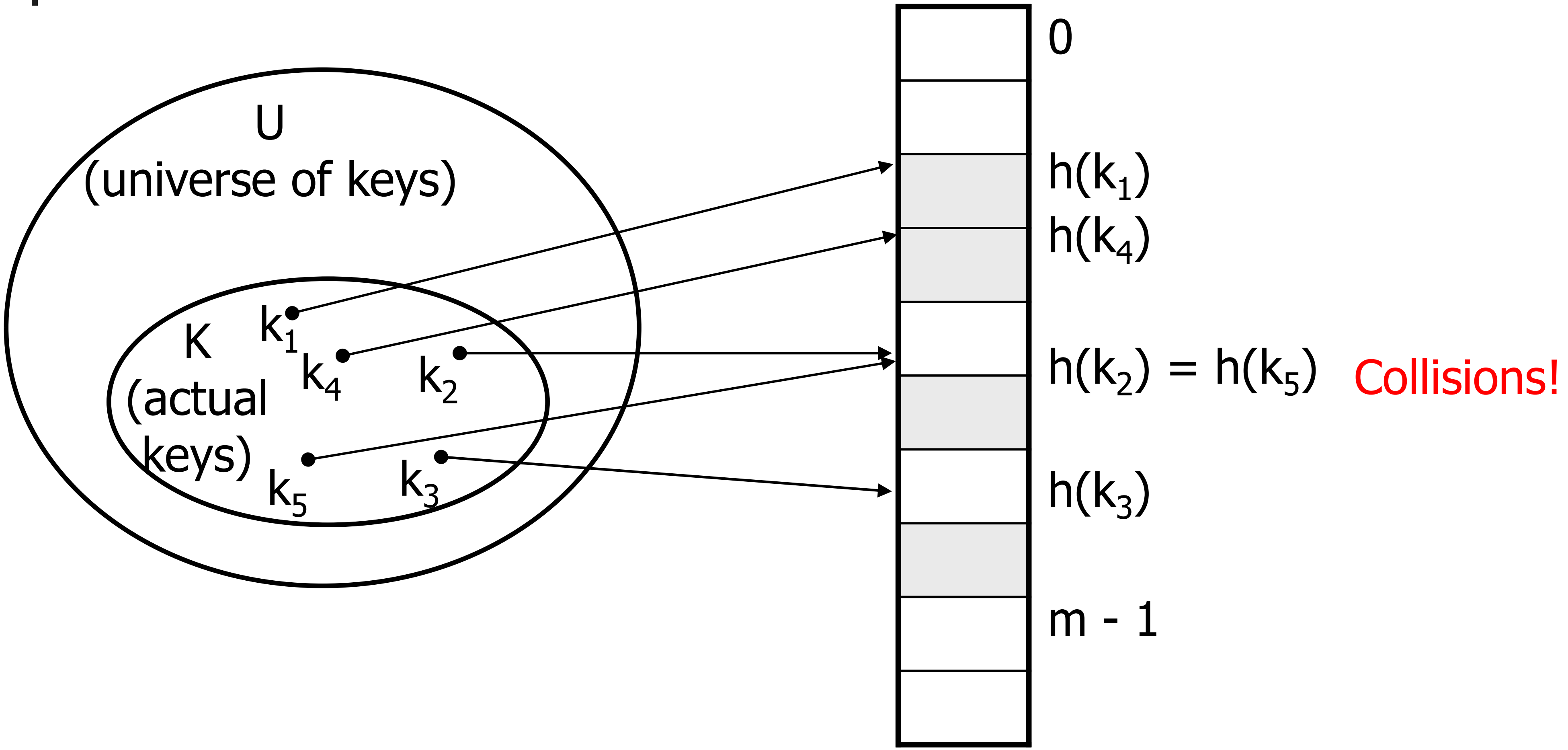
  y= ((17*121267) + 7) % 307

  y= (2061539 + 7) % 307

  y= 2061546 % 307

  y= 41

Do you see any problems with this approach?



Collisions

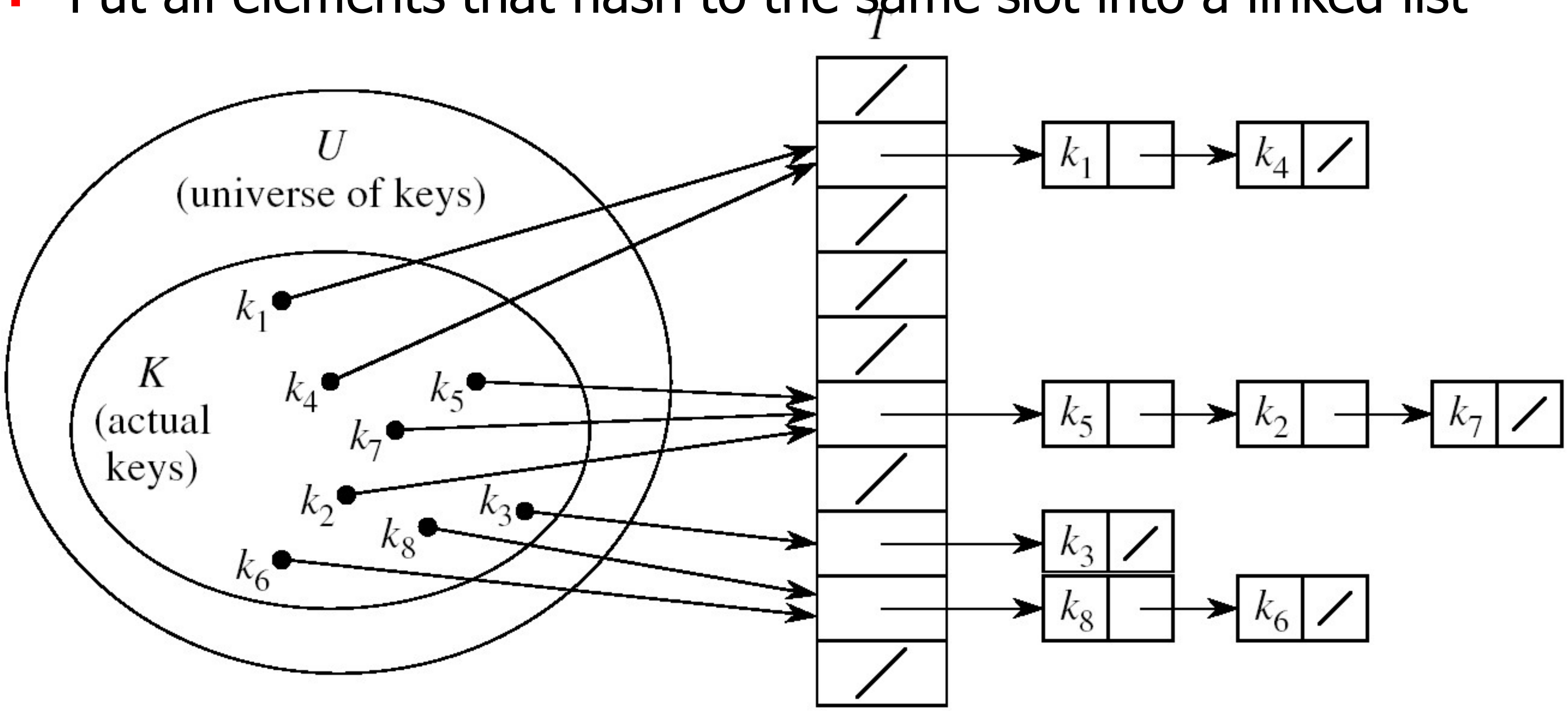
- Two or more keys hash to the same slot!!
- For a given set K of keys
  - If $|K| \leq m$, collisions may or may not happen, depending on the hash function.
  - If $|K| > m$, collisions will definitely happen (i.e., there must be at least two keys that have the same hash value).
- Avoiding collisions completely is hard, even with a good hash function

Handling Collisions

- Methods to handle collision:
  - Chaining: Use a *linked list* to store multiple items that hash to the same slot. Also known as *Open hashing*.
  - Closed Hashing (or *probing*): search for empty slots using a second function and store item in first empty slot that is found. Also known as *Open addressing*.
    - Linear probing
    - Quadratic probing
    - Double hashing

Handling Collisions Using Chaining

- Idea:
  - Put all elements that hash to the same slot into a linked list



  - Slot j contains a pointer to the head of the list of all elements that hash to j

Collision with Chaining

- Choosing the size of the table
  - Small enough not to waste space.
  - Large enough such that lists remain short.
  - Typically 1/5 or 1/10 of the total number of elements.
- How should we keep the lists: ordered or not?
  - Not ordered!
    - Insert is fast
    - Can easily remove the most recently inserted elements

Collision Resolution by Closed Hashing

- Given an item X, try
  cells $h_0(X)$, $h_1(X)$, $h_2(X)$, …, $h_i(X)$
- hash(X) = X % TableSize
- $h_i(X)$ = (hash(X) + F(i)) % TableSize
  - Define F(0) = 0
- F is the collision resolution function.
- Some possibilities:
  - Linear: F(i) = i
  - Quadratic: F(i) = i^2
  - Double Hashing: F(i) = i × $\text{Hash}_2(X)$

Closed Hashing I: Linear Probing

- Main Idea: When collision occurs, scan down the array one cell at a time looking for an empty cell
  - $h_i(X) = (\text{hash}(X) + i) \% \text{TableSize}$ $(i = 0, 1, 2, \ldots)$
  - Compute hash value and increment it until a free cell is found

- Probe sequence:

  0th probe = hash(X) % TableSize

  1th probe = (hash(X) + 1) % TableSize

  2th probe = (hash(X) + 2) % TableSize

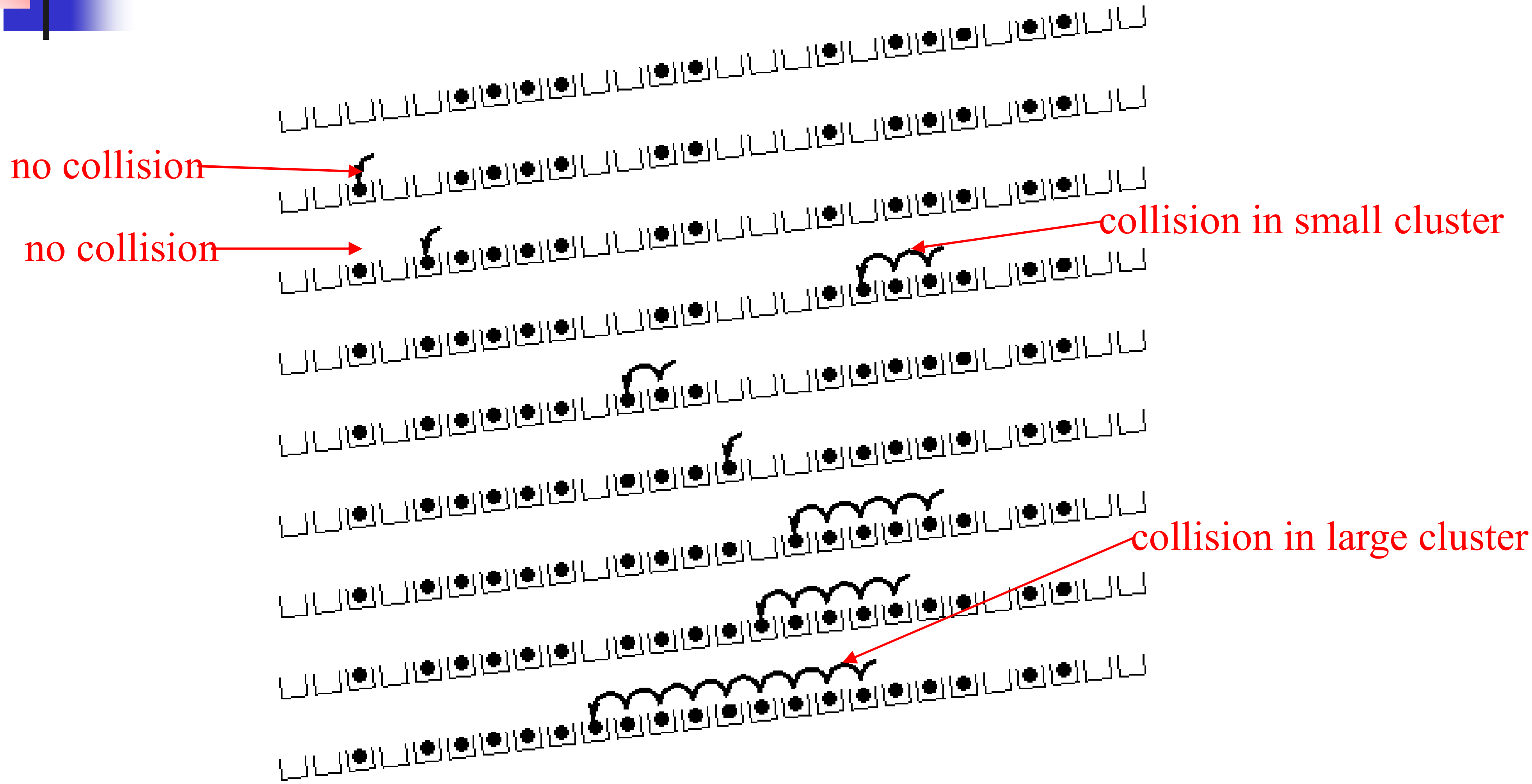
  . . .

  ith probe = (hash(X) + i) % TableSize

Linear Probing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2			21	21
3				2
4				
5				
6				
probes:	1	1	3	2

Linear Probing – Clustering



Drawbacks of Linear Probing

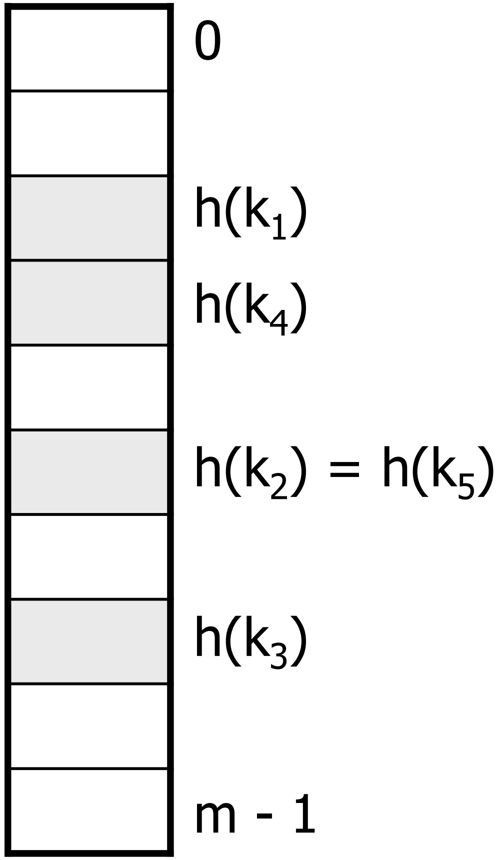
- Works until array is full, but as number of items N approaches TableSize, access time approaches O(N)
- Very prone to cluster formation (as shown example)
  - If a key hashes anywhere into a cluster, finding a free cell involves going through the entire cluster – and making it grow!
  - Primary clustering – clusters grow when keys hash to values close to each other
- Can have cases where table is empty except for a few clusters
  - Does not satisfy good hash function criterion of distributing keys uniformly

Linear probing: Searching for a key

- Three cases:
  i. Position in table is occupied with an element of equal key.
  ii. Position in table is empty.
  iii. Position in table occupied with a different element.

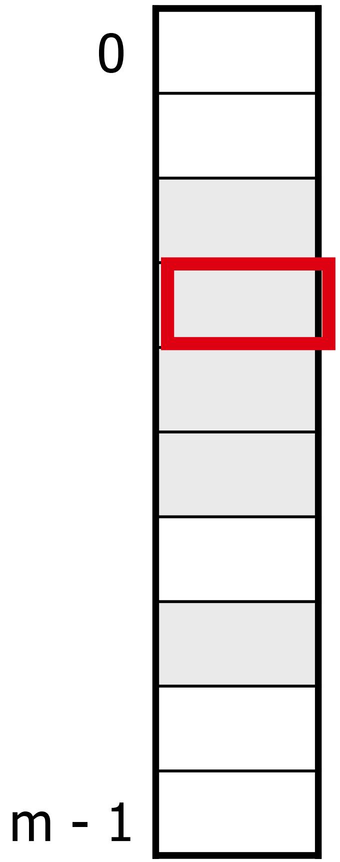
- Case iii: probe the next higher index until the element is found or an empty position is found in a wrap around fashion.

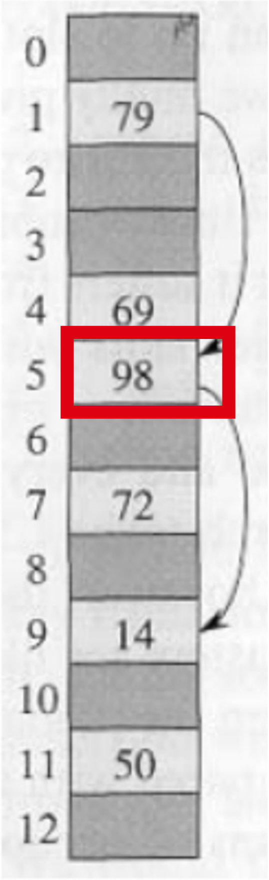
- i.e. Keep moving through the array, wrapping around at the end, until a free spot is found. .



Linear probing: Deleting a key

- Deletions are a bit trickier than in chained hashing.
- We cannot just do a search and remove the element where we find it.
- Problems
  - Cannot mark the slot as empty
  - Impossible to retrieve keys inserted after that slot was occupied
- Solution
  - Mark the slot with a sentinel value DELETED
- The deleted slot can later be used for insertion
- Searching will be able to find all the keys





Linear probing: weakness

- Linear probing exhibits severe performance degradations when the load factor gets high.
- The number of collisions tends to grow as a function of the number of existing collisions.
- This is called primary clustering.

Closed Hashing II: Quadratic Probing

- Main Idea: Spread out the search for an empty slot: Increment by i^2 instead of i

- $h_i(X) = (Hash(X) + i^2)$ % *TableSize*

  $h_0(X) = (Hash(X) + 0)$% TableSize

  $h_1(X) = (Hash(X) + 1)$ % TableSize

  $h_2(X) = (Hash(X) + 4)$ % TableSize

  $h_3(X) = (Hash(X) + 9)$ % TableSize

Quadratic Probing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2
0	14	14	14	14
1		8	8	8
2				2
3				
4			21	21
5				
6				
probes:	1	1	3	1

Problem With Quadratic Probing

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0	insert(2) 2%7 = 2	insert(7) 7%7 = 0
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	3	1	??

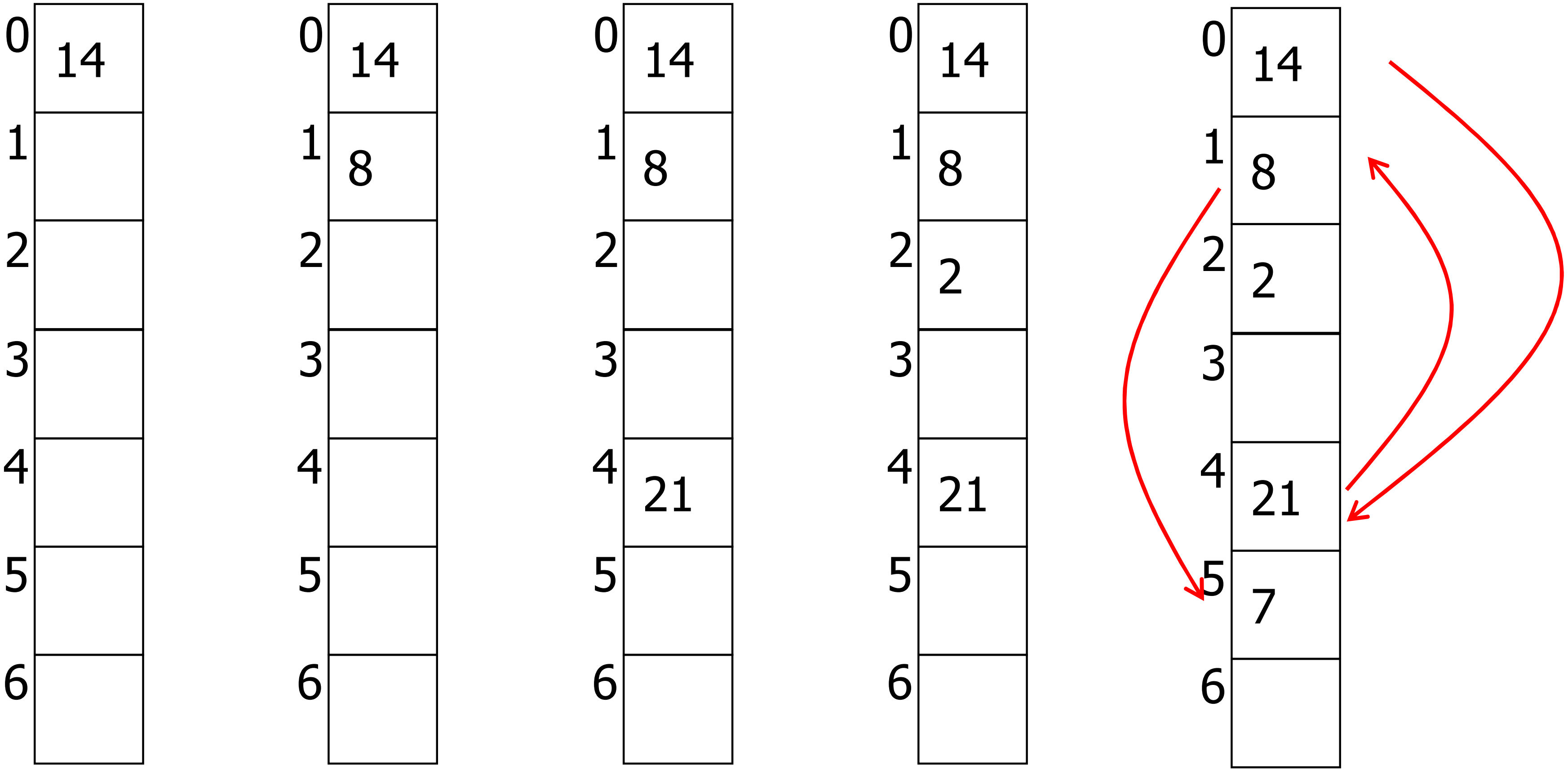
Closed Hashing III: Double Hashing

- Idea: Spread out the search for an empty slot by using a second hash function
  - No primary or secondary clustering
- $h_i(X) = (Hash_1(X) + i \times Hash_2(X))$ % TableSize
  - for i = 0, 1, 2, …
- Good choice of $Hash_2(X)$ can guarantee does not get "stuck" .
  - Integer keys:
    $Hash_2(X) = R - (X \% R)$
    where R is a prime smaller than TableSize

Double Hashing Example

	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0 5-(21%5)=4	insert(2) 2%7 = 2	insert(7) 7%7 = 0 5-(21%5)=4
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					
6					
probes:	1	1	2	1	??

Double Hashing Example



	insert(14) 14%7 = 0	insert(8) 8%7 = 1	insert(21) 21%7 =0 5-(21%5)=4	insert(2) 2%7 = 2	insert(7) 7%7 = 0 5-(21%5)=4
0	14	14	14	14	14
1		8	8	8	8
2				2	2
3					
4			21	21	21
5					7
6					
probes:	1	1	2	1	4

Closed Hashing III: Double Hashing

$h_1(k) = k \% 7$

$h_2(k) = 5 - (k \% 5)$

$h(k, i) = (h_1(k) + i\ h_2(k)\ ) \% 7$

- Insert key 7:

$h_1(7, 0) = 7 \bmod 7 = 0$

$h(7, 1) = (h_1(7) + h_2(21)) \% 7$

$= (0 + 4) \% 7 = 4$

$h(7, 2) = (h_1(7) + 2\ h_2(21)) \% 7$

$= (0 + 8) \% 7 = 1$

$h(7, 3) = (h_1(7) + 3\ h_2(21)) \% 7$

$= (0 + 12) \% 7 = 5$

Double Hashing: Example

$h_1(k) = k \% 13$

$h_2(k) = 1 + (k \% 11)$

$h(k, i) = (h_1(k) + i\, h_2(k)) \bmod 13$

- Insert key 14:

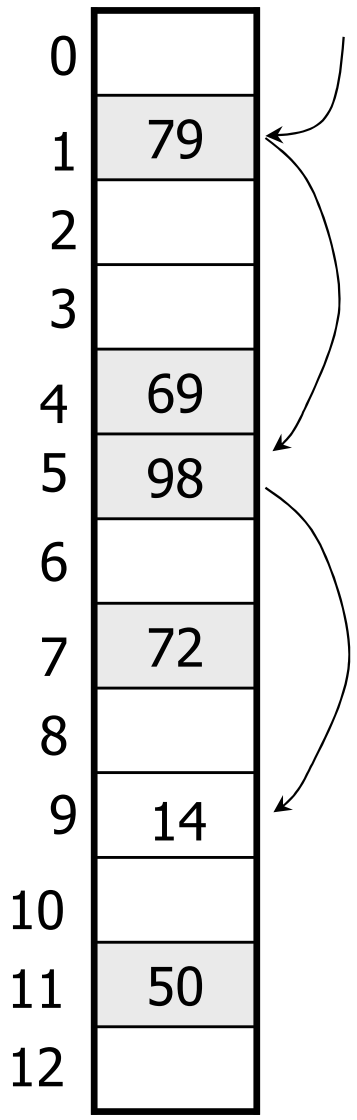
$h_1(14, 0) = 14 \bmod 13 = 1$

$h(14, 1) = (h_1(14) + h_2(14)) \bmod 13$

$= (1 + 4) \bmod 13 = 5$

$h(14, 2) = (h_1(14) + 2\, h_2(14)) \bmod 13$

$= (1 + 8) \bmod 13 = 9$



0	
1	79
2	
3	
4	69
5	98
6	
7	72
8	
9	14
10	
11	50
12	

Double Hashing

- Advantage: avoids clustering
- Disadvantage: harder to delete an element
- Can generate m^2 probe sequences maximum