CX1107
Data Structures and Algorithms

Searching

College of Engineering

School of Computer Science and Engineering

Dr. Loke Yuan Ren

Lecturer

yrloke@ntu.edu.sg

Overview

- Exhaustive Algorithm: Sequential Search
- Decrease-and-conquer Algorithm: Binary Search
- Data Structures:
  - Hashing
    - Open Hashing
    - Closed Hashing

Sequential Search

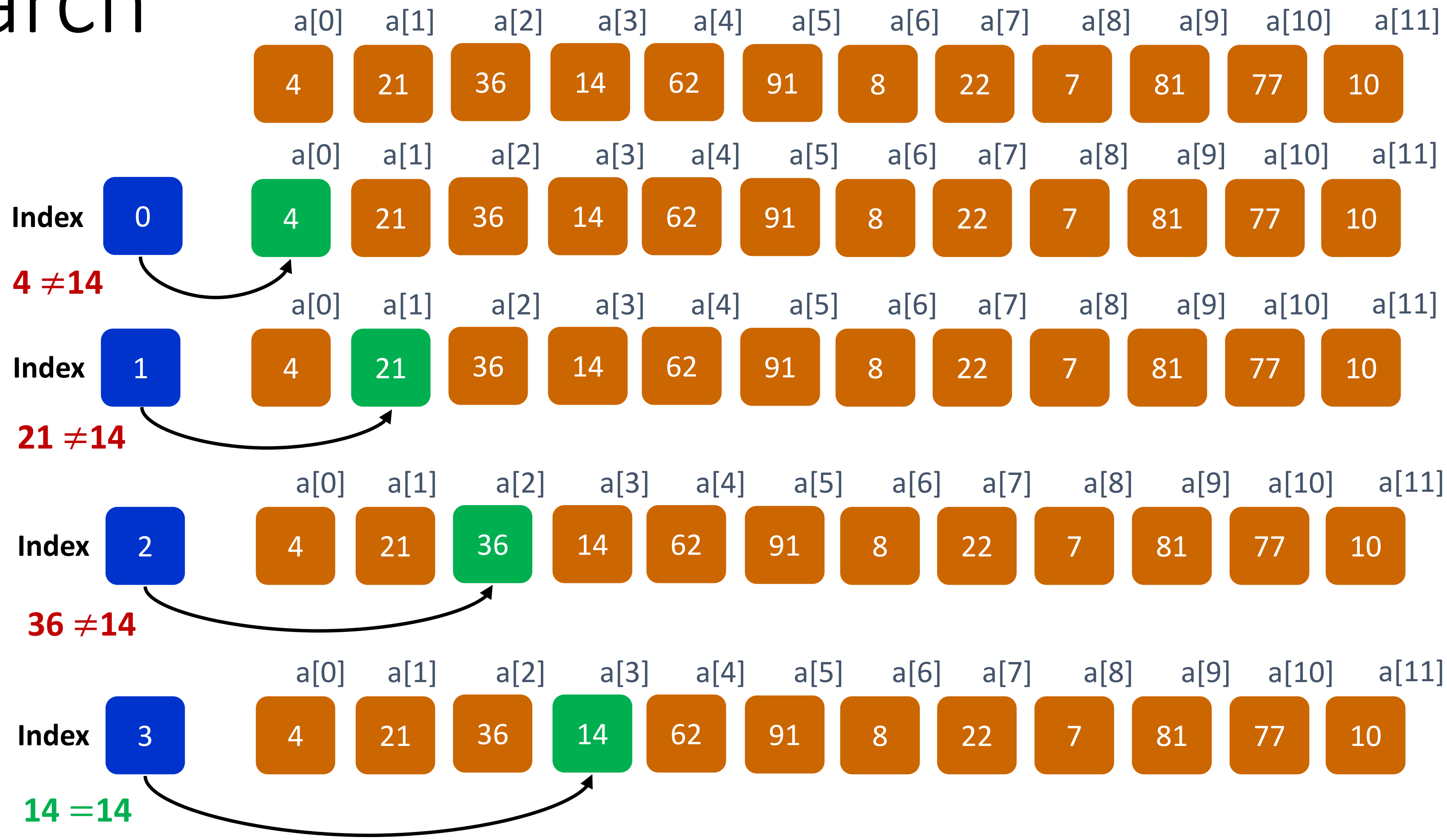
- The array is unordered
- The approach is a brute-force approach or a näive algorithm
- Every element in the array is required to be read and compare

Sequential Search

Target Given	14
Location Wanted	3

Algorithm 2 Sequential Search

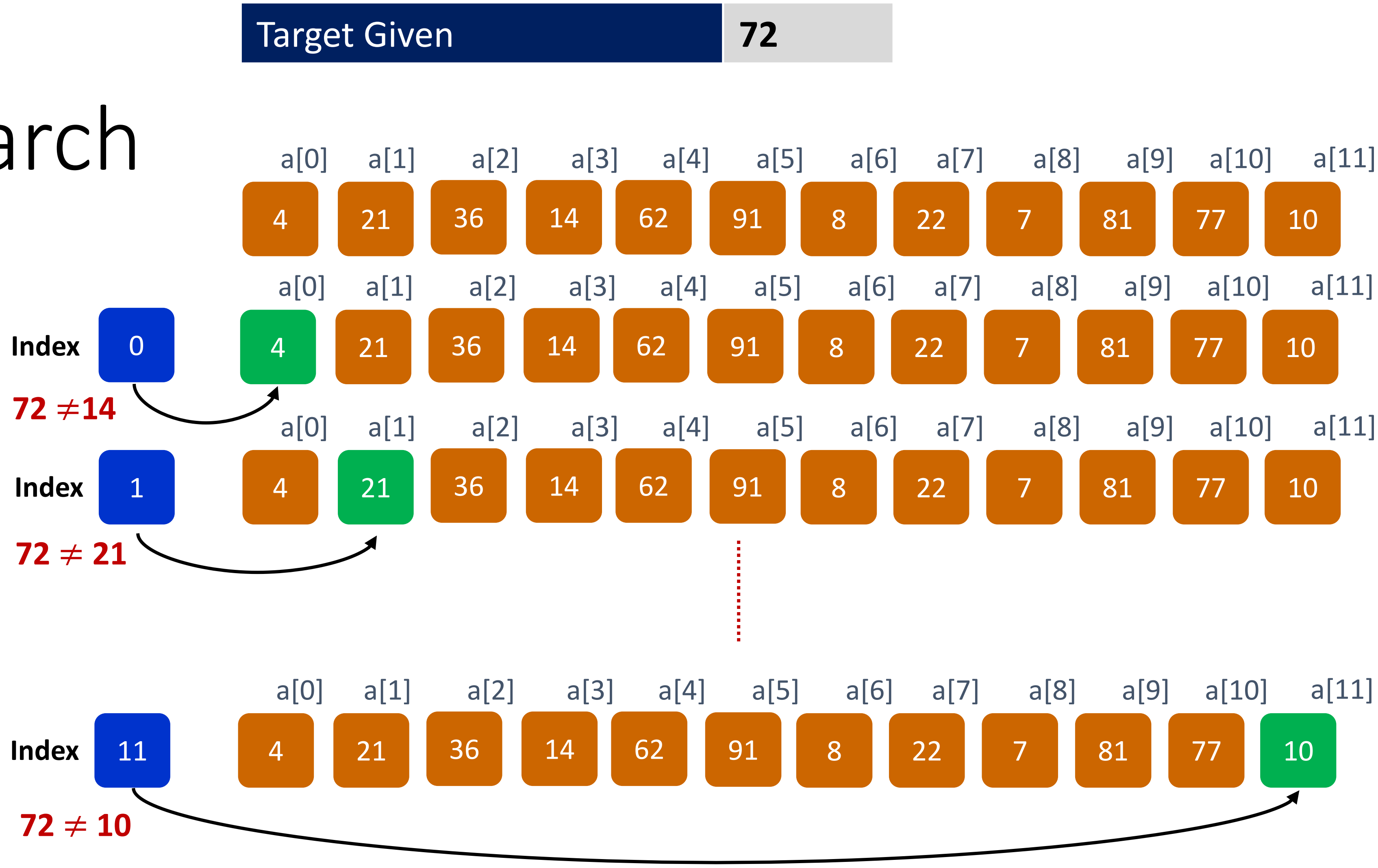
```
function seqSearch(int[] Data, int n, int key)
begin
for index = 0 to n − 1 do
    begin
    if Data[index] == key then
        return index;      ⟸ Success
    end
return -1;   ⟸ Failure
end
```



Sequential Search

Algorithm 2 Sequential Search

```
function seqSearch(int[] Data, int n, int key)
begin
for index = 0 to n − 1 do
    begin
    if Data[index] == key then
        return index;          Success
    end
return -1;          Failure
end
```



Time Complexity of Sequential Search

- Best-case complexity: Θ (1) , 1 comparison against key (the first item is the search key)
- Worst-case complexity: Θ (n) , n comparison against key (Either the last item or no item is the search key)
- Average-case complexity:

Key is in the search 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
- Average complexity: $A_s(n) = \sum_{i=1}^{n} (\frac{1}{n})(i)$
- $= \frac{1}{n}\left(\frac{n(n+1)}{2}\right) = \frac{n+1}{2} = \Theta(n)$

Key is not in the search array:

- n comparisons (if it is an linked list, you may need to take an extra comparison)
- $A_s(n) = n = \Theta(n)$

Pr(succ) $A_s(n)$ + Pr(fail) $A_f(n)$ $= q\frac{n+1}{2} + (1-q)n$

Both worst and average complexity are $\Theta(n)$

Binary Search

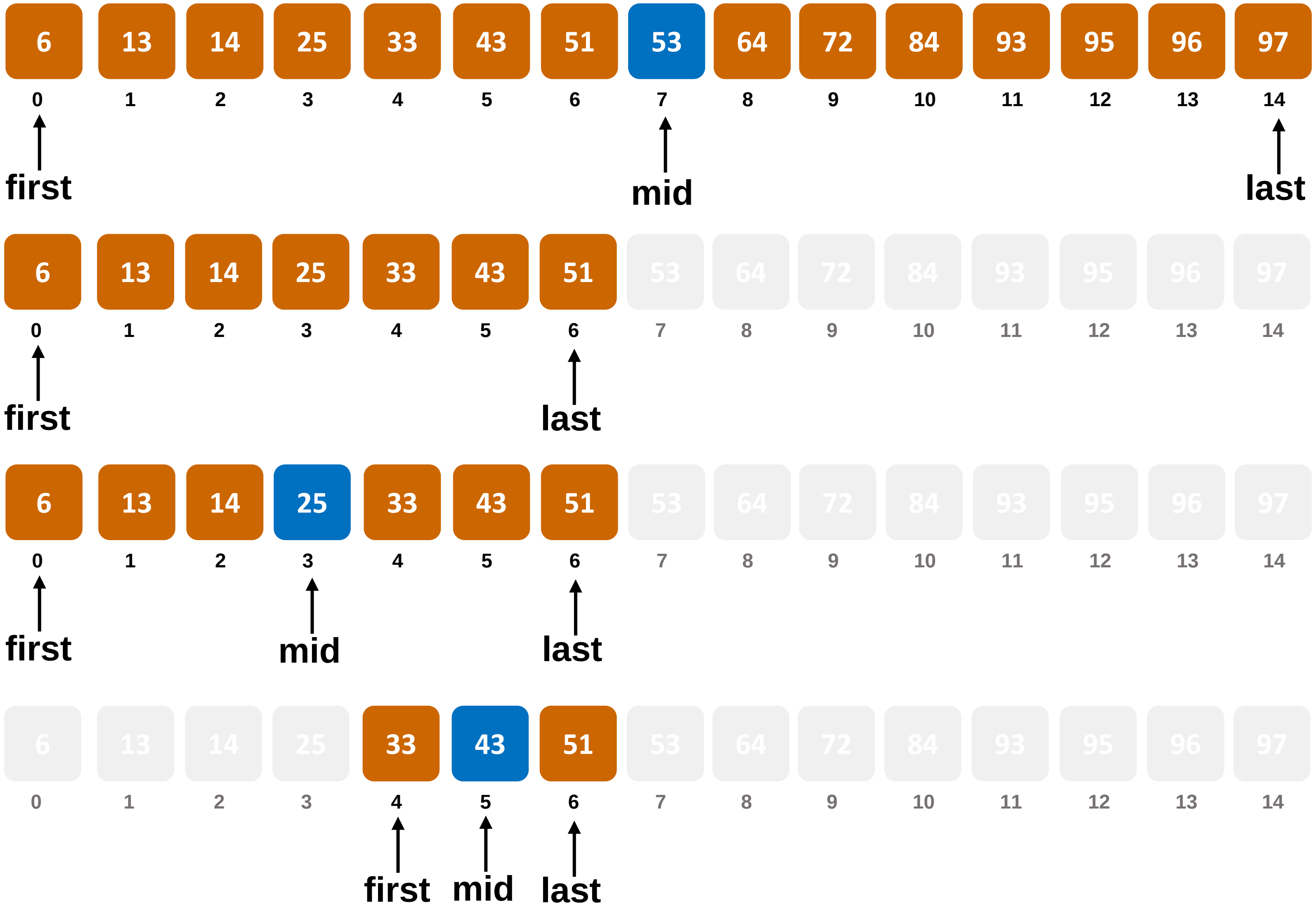
- The array is ordered
- The approach is a decrease-and-conquer approach
- A problem is divided into two smaller and similar sub-problem, one of which does not even have to be solved
- The method uses the information of the order to reduce the search space.

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);
    }
}
```

Recursive Version

Binary search for target **33**

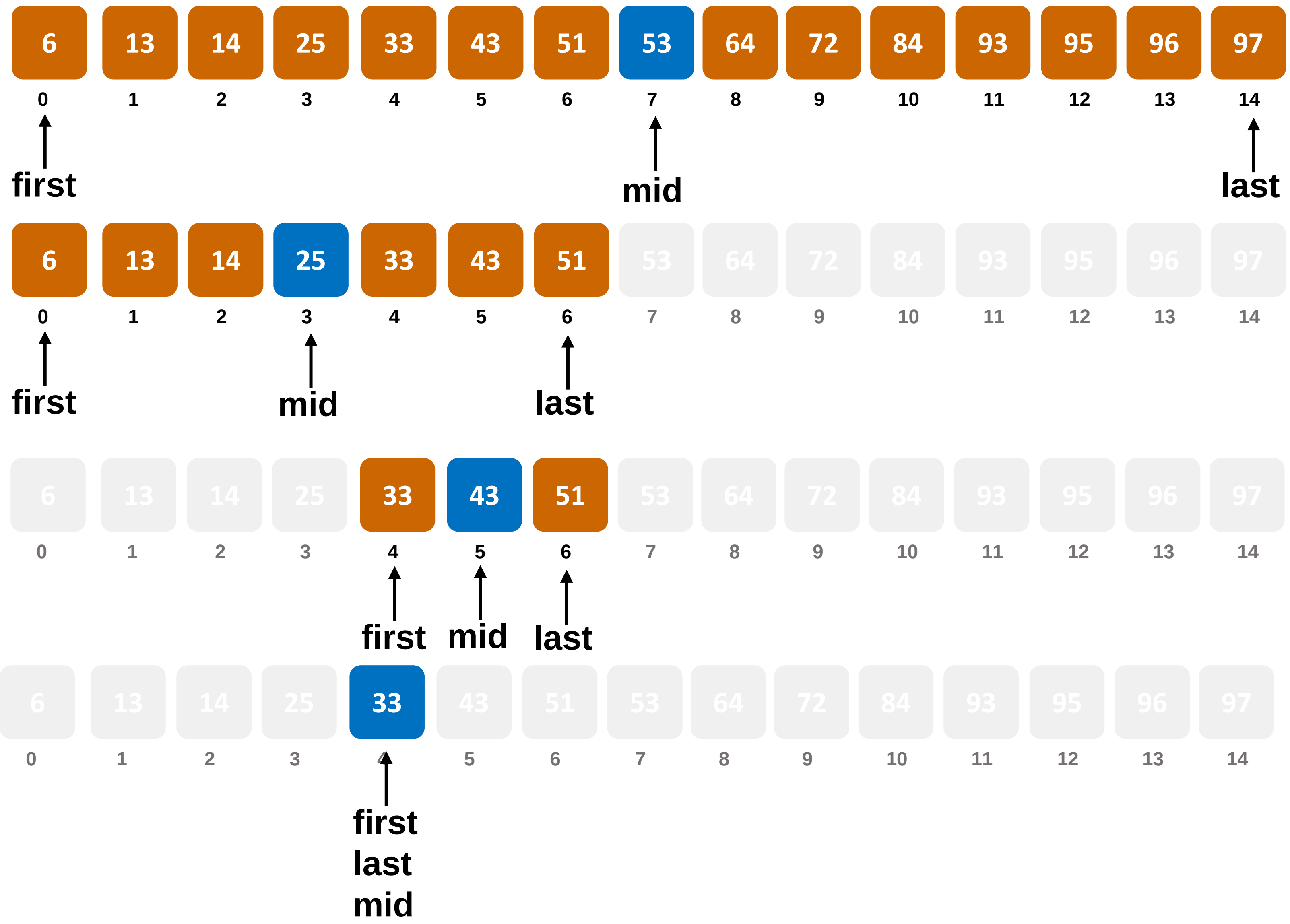


Binary Search

```
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);
    }
}
```

Recursive Version

Binary search for target **33**



Binary Search

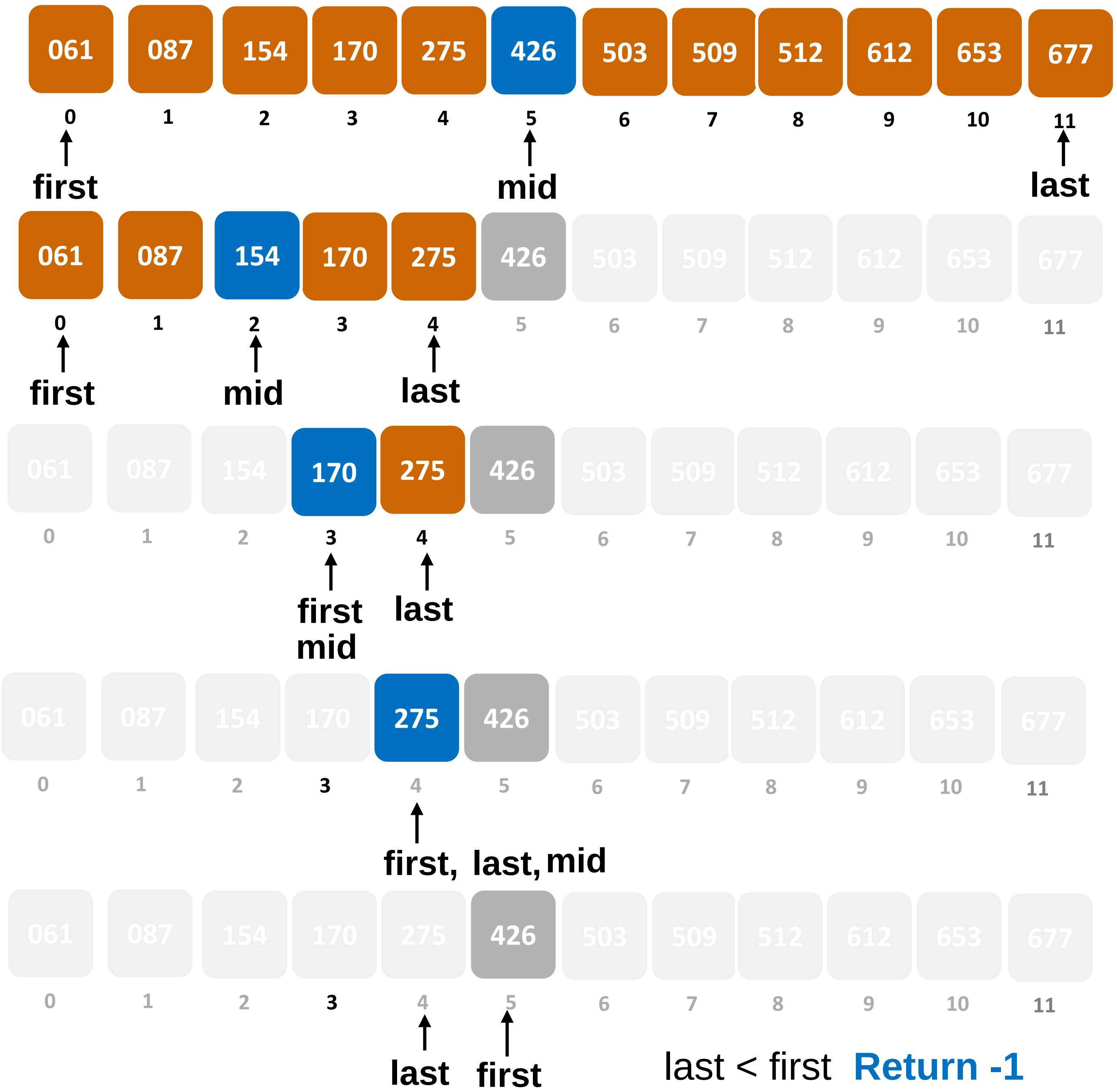
```
int binarySearch (int E[], int first, int last, int k)
{
    if(last < first)
        return -1;
    else {
        int mid = (first + last)/2;
        if(k == E[mid])
            return mid;
        else if(k < E[mid])
            return binarySearch(E,first, mid-1,k);
        else
            return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

Recursive Version

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

Iterative Version

Binary search for target **400**



Time Complexity of Binary Search: **Worst Case**

```
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);
    }
}
```

T(n)

Constant c

$\boldsymbol{T(n/2)}$

$\boldsymbol{T(n/2)}$

Recursive Version

$$
\begin{aligned}
T(n) &= T\left(\frac{n}{2}\right) + c \\
&= T\left(\frac{n}{2^2}\right) + 2c \\
&= T\left(\frac{n}{2^3}\right) + 3c \\
&\dots \\
&= T\left(\frac{n}{2^k}\right) + kc \\
&= T(1) + kc \\
&= T(0) + (k+1)c \\
&= (k+1)c \\
&= (\lfloor \log_2 n \rfloor + 1)c \\
&= (\lceil \log_2 (n+1) \rceil)c \\
&= \Theta(\log_2 n)
\end{aligned}
$$

$$
\begin{aligned}
&1 \le \frac{n}{2^k} < 2 \\
&2^k \le n < 2^{k+1} \\
&k \le \log_2 n < k+1 \\
&k = \lfloor \log_2 n \rfloor
\end{aligned}
$$

$$
\begin{array}{llll}
k & \le \log_2 n & & < k+1 \\
2^k & \le \; n & & < 2^{k+1} \\
2^k & < \; n+1 & & \le 2^{k+1} \\
k & < \log_2(n+1) & & \le k+1
\end{array}
$$

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

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

Eg. k=4 and n is integer

$2^4 \le n < 2^5$

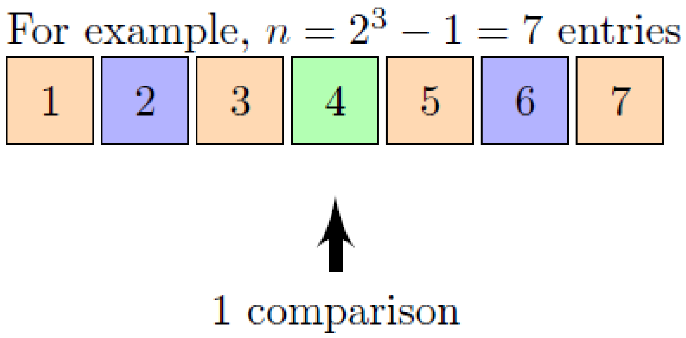
$2^4 < n+1 \le 2^5$

Time Complexity of Binary Search: Average Case

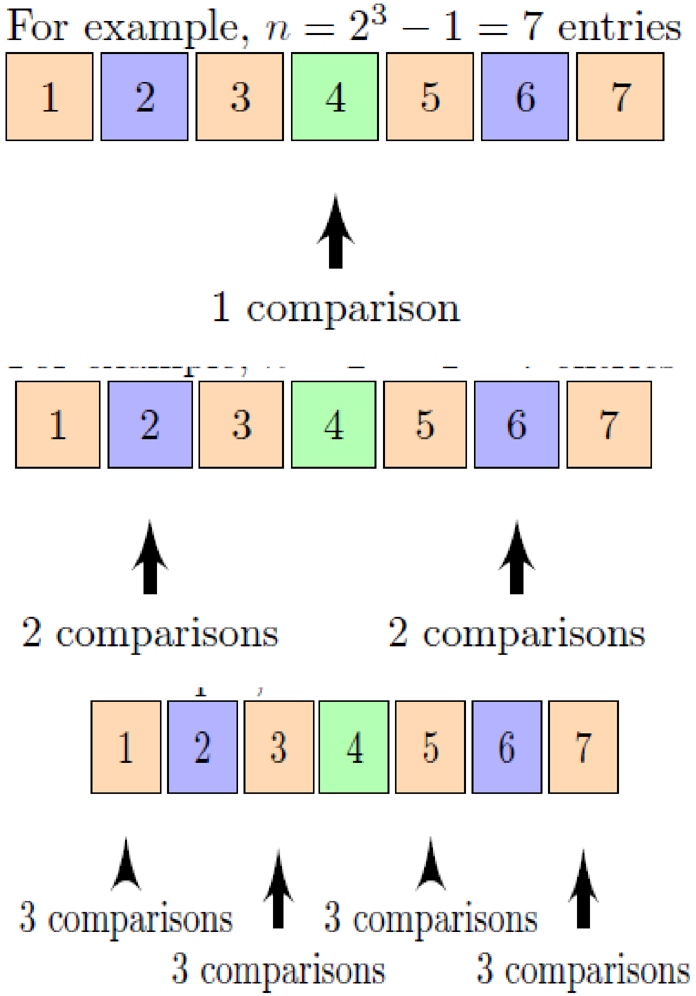
$$A(n) = qA_s(n) + (1-q)A_f(n)$$

- $A_s(n)$: # of comparisons for successful search
- $A_f(n)$: # of comparisons for unsuccessful search (worst case): $\Theta(\log_2 n)$

- For $A_s(n)$, we assume $n = 2^k - 1$ first



Time Complexity of Binary Search: Average Case



- We can observe that:
  - 1 position requires 1 comparison
  - 2 positions requires 2 comparisons
  - 4 positions requires 3 comparisons
  - … …
  - 2^{t-1} positions requires t comparisons
- n=2^k-1, we have

$$A_s(n) = \frac{1}{n}\sum_{t=1}^{k} t2^{t-1} = \frac{(k-1)2^k + 1}{n} = \frac{[\log_2(n+1) - 1](n+1) + 1}{n} = \log_2(n+1) - 1 + \frac{\log_2(n+1)}{n}$$

$$\begin{aligned}
\sum_{t=1}^{k} t2^{t-1} &= 1\cdot 1 + 2\cdot 2 + 3\cdot 4 + 4\cdot 8 + \ldots + k\cdot 2^{k-1} \\
2\sum_{t=1}^{k} t2^{t-1} &= \quad 1\cdot 2 + 2\cdot 4 + 3\cdot 8 + \ldots + (k-1)\cdot 2^{k-1} + k\cdot 2^{k} \\
(2-1)\sum_{t=1}^{k} t2^{t-1} &= -1\cdot 1 - 1\cdot 2 - 1\cdot 4 - 1\cdot 8 - \ldots - 1\cdot 2^{k-1} + k\cdot 2^{k} \quad \triangleright \text{eq. 2 - eq. 1} \\
\sum_{t=1}^{k} t2^{t-1} &= -2^k + 1 + k\cdot 2^k \quad \triangleright \text{geometric series} \\
&= 2^k(k-1) + 1
\end{aligned}$$

Time Complexity of Binary Search: Average Case

- The average complexity is

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

- Binary search does approximately $\log_2(n+1)$ comparisons on average for n entries.
  - q is probability which is always ≤ 1
  - $\frac{\log_2(n+1)}{n}$ is very small especially when n >> 1

Hashing

- A typical space and time trade-off in algorithm
- To achieve search time in O(1), memory usage will be increased

- What is hashing?
  - Hash functions
  - Collision and its resolutions
    - Closed Address Hashing
    - Open Address Hashing
      - Linear Probing
      - Quadratic Probing
      - Double Hashing
  - Delete a key from a hash table
  - Dynamic hash tables

What is hashing?

Direct-Address Table

- Assume that the keys of elements K drawn from the universe of possible keys U
- No two elements have the same key

- Search time is O(1) but …
  - The array size is enormous
  - |U| >> |K|

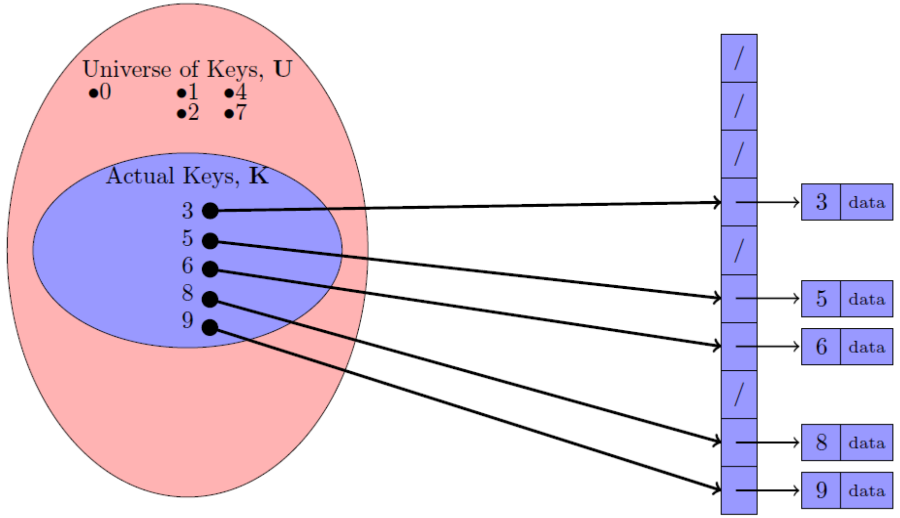


Figure 4.1: Direct Address Table

What is hashing?

- To reduce the key space to a reasonable size
- Each key is mapped to a unique index (hash value/code/address)
- Search time remains O(1) on the average

hash function: {all possible keys} → {0, 1, 2, …, h-1}

- The array is called a hash table
- Each entry in the hash table is called a hash slot
- When multiple keys are mapped to the same hash value, a collision occurs
- If there are n records stored in a hash table with h slots, its load factor is $\alpha = \frac{n}{h}$

Hash Functions

- Must map all possible value within the range of the hash table uniquely
- Mapping should achieve an even distribution of the keys
- Easy and fast to compute
- Minimize collision

1. Modulo Arithmetic
2. Folding
3. Mid-square
4. Multiplicative Congruential Method
5. Etc.

Hash Functions

1. Modulo Arithmetic: $\mathrm{H}(k) = k \bmod h$

For keys are chosen from

- decimal number → h avoid to use powers of 10
- unknown lower p-bit patterns → h avoid to use powers of 2
- “real” data → h should be a prime number but not too close to any power of 2

2. Folding
   - Partition the key into several parts and combine the parts in a convenient way
   - Shift folding: Divide the key into a few parts and added up these parts

Hash Functions

3. Mid-square
   - The key is squared and the middle part of the result is used as the hash address
     - Eg. k=3121, $k^2 = 3121^2 = 9740641 \rightarrow H(k) = 406$

4. Multiplicative Congruential Method
   - Pseudo-random number generator
     - $a = 8\left\lfloor \frac{h}{23} \right\rfloor + 5$
     - $H(k) = (a \times k)\ mod\ h$

```
>> h = 31
>> a= 8*floor(h/23) +5

a =

    13

>> k = 1:15

k =

     1     2     3     4     5     6     7     8     9    10    11    12    13    14    15

>> fk = mod((a*k),h)

fk =

    13    26     8    21     3    16    29    11    24     6    19     1    14    27     9

```

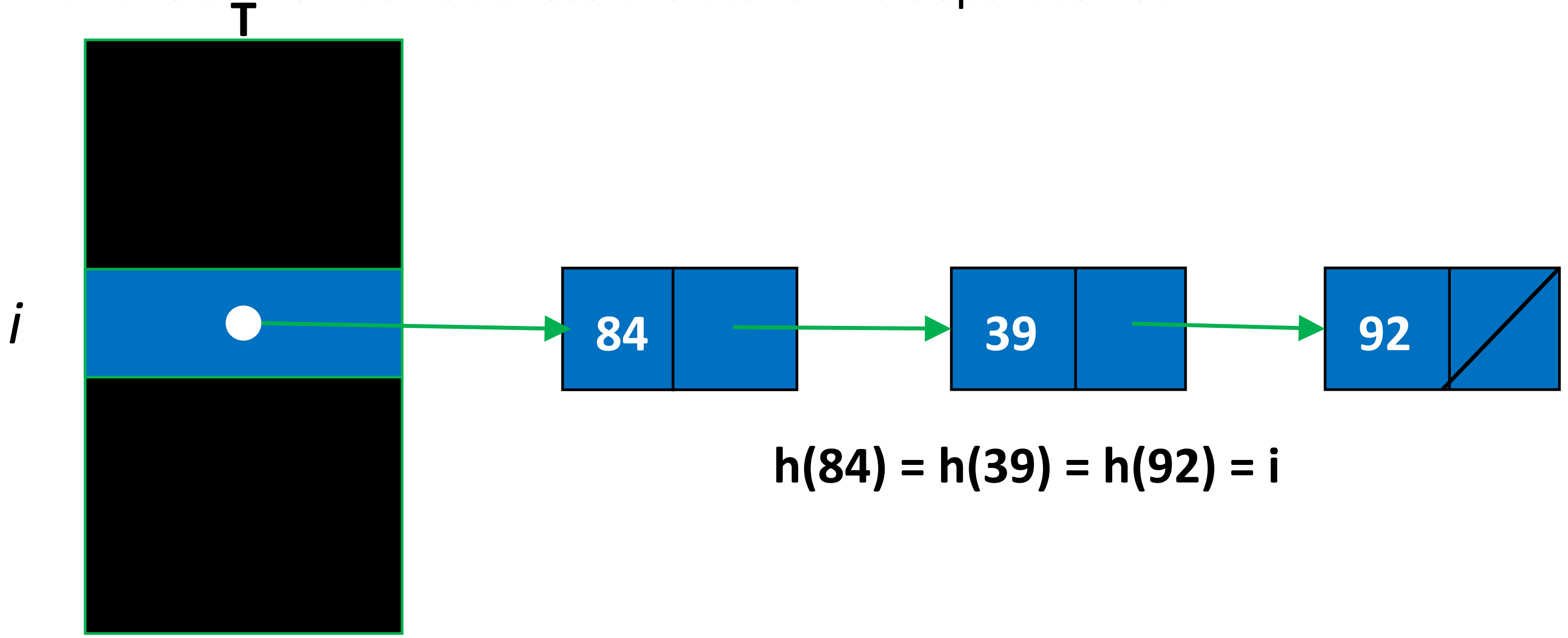
Collision Resolutions

- Closed Addressing Hashing – a.k.a separate chaining

- Open Addressing Hashing
  - Linear Probing
  - Quadratic Probing
  - Double Probing

Closed Addressing: Separate Chaining

- Keys are not stored in the table itself
- All the keys with the same hash address are store in a separate list



- During searching, the searched element with hash address i is compared with elements in linked list H[i] sequentially
- In closed address hashing, there will be α number of elements in each linked list on average.

Closed Addressing: Separate Chaining

Time complexity in the worst-case analysis:

- When all elements are hashed to the same slot
- A linked list contains all n elements
- Its unsuccessful search takes n key comparisons, $\Theta(n)$
- Its successful search, assuming the probability of searching for each item is $\frac{1}{n}$
  $\frac{1}{n}\sum_{i=1}^{n} i = \frac{n+1}{2} = \Theta(n)$
  - It is just like a sequential search

Closed Address Hashing: Separate Chaining

Time complexity in the average-case analysis:

- All elements are equally likely hashed into h slots.
- Its unsuccessful search takes $\frac{n}{h}$ key comparisons, $\Theta(\alpha)$
- Its successful search takes 1 more than the number of comparisons done when the sought after item was inserted into the hash table
  - Before the i^th item is inserted into the hash table, the average length of all lists is $\frac{i-1}{h}$
  - When the ith item is sought for, the no. of comparisons is $(1+\frac{i-1}{h})$
  - The average number of key comparisons over n items
  - If α is constant (n is proportional to h), then $\Theta(1)$
  - Searching takes constant time averagely

$$\boxed{\begin{aligned}\frac{1}{n}\sum_{i=1}^{n}\left(1+\frac{i-1}{h}\right) &= \frac{1}{n}\left(\sum_{i=1}^{n}1+\frac{1}{h}\sum_{i=1}^{n}(i-1)\right)\\ &= 1+\frac{1}{nh}\left(\frac{n}{2}(n-1)\right)\\ &= 1+\frac{n-1}{2h} = \Theta(1+\alpha)\end{aligned}}$$

Open Addressing

- Keys are stored in the table itself
- α cannot be greater than 1
- When collision occurs, probe is required for the alternate slot
  - Ideally, the probing approach can visit every possible slot.

1. Linear Probing: probe the next slot

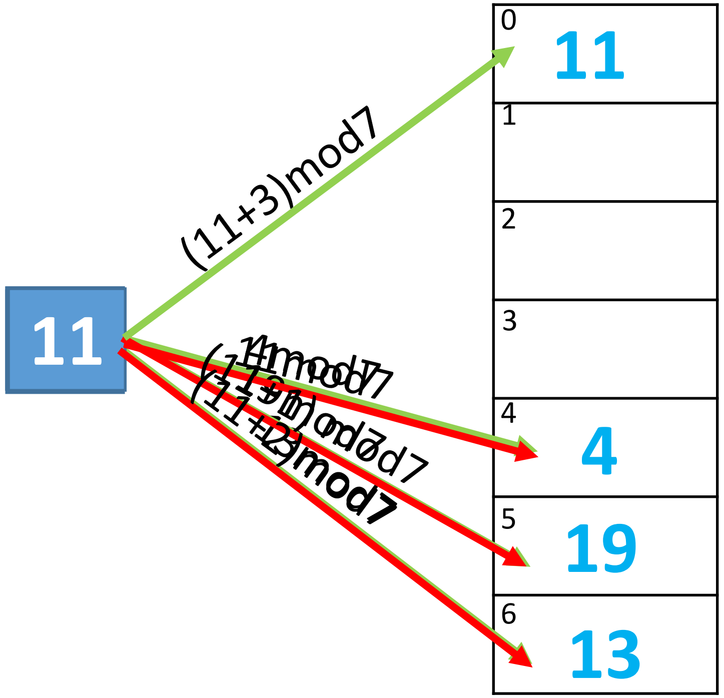
$H(k, i) = (k + i) \bmod h$ where $i \in [0, h - 1]$

eg. $H(k, i) = (k + i) \bmod 7$

$k \in \{4, 13, 19, 11\}$

Primary clustering:

- A long runs of occupied slots
- Average search time is increased



Open Addressing

2. Quadratic Probing

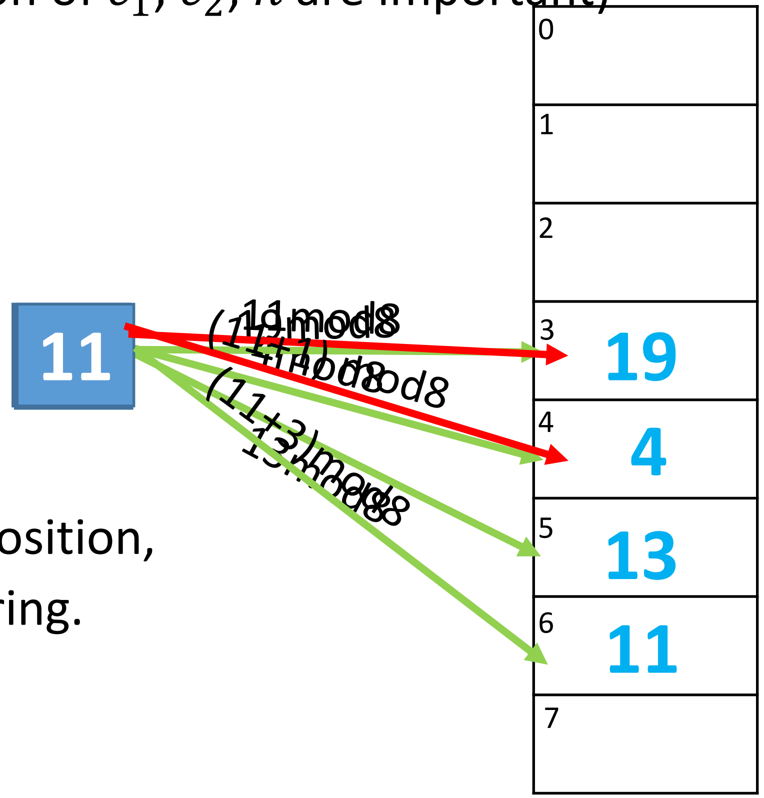
$$H(k,i) = (k + c_1 i + c_2 i^2) \bmod h \quad \text{where } c_1 \text{ and } c_2 \text{ are constants, } c_2 \neq 0$$

- May not all hash table slots be on the probe sequence (selection of c_1, c_2, h are important)
- For $h = 2^n$, a good choice for the constants are $c_1 = c_2 = \frac{1}{2}$

eg. $H(k,i) = \left(k + \frac{1}{2}i + \frac{1}{2}i^2\right) \bmod 8$

$k \in \{4, 13, 19, 11\}$

$(\frac{1}{2}i + \frac{1}{2}i^2)$mod 8
1
3
6
2
7
5
4



- Secondary Clustering: if two keys have the same initial probe position, their probe sequences will be the same. This will form a clustering.
    - Inserting k=3 in the previous example.

Open Addressing

3. Double Hashing: a random probing method

$H(k,i) = (k + iD(k)) \bmod h$ where $i \in [0, h-1]$ and $D(k)$ is another hash function

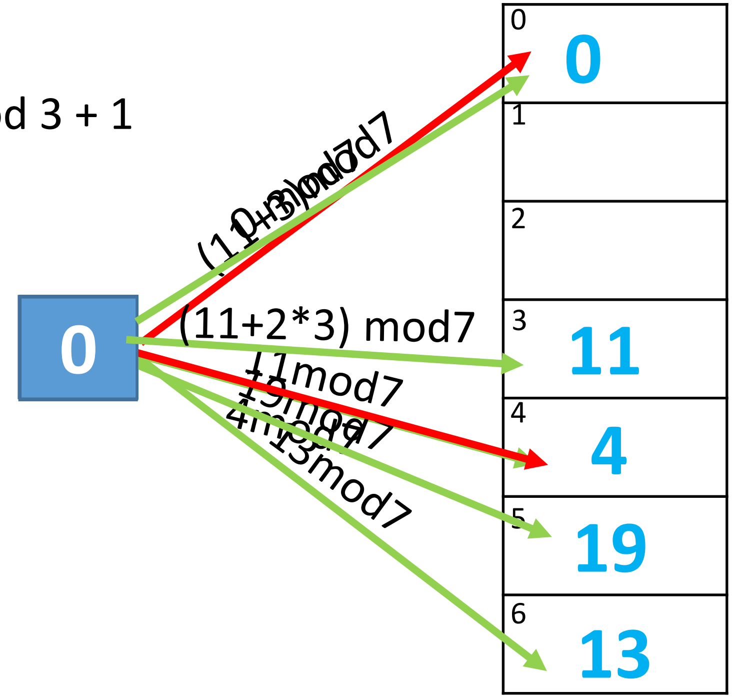
- The hash table size h should be a prime number

eg. $H(k,i) = (k + iD(k)) \bmod 7$

$\mathrm{D}(k) = (k) \bmod 3 + 1$

$k \in \{0, 4, 13, 19, 11\}$

D(11) = 11 mod 3 + 1



Time Complexity

Linear Probing

- Successful Search: $\frac{1}{2}(1 + \frac{1}{1-\alpha})$
- Unsuccessful Search: $\frac{1}{2}(1 + \left(\frac{1}{1-\alpha}\right)^2)$

Double Hashing

- Successful Search: $\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$
- Unsuccessful Search: $\frac{1}{1-\alpha}$

*Proof can be found in The Art of Computer Programming by Knuth Donald (1973)

Delete A Key Under Open Addressing

- Leave the deleted key in the table
- Make a marker indicating that it is deleted
- Overwrite it when a new key is inserted to the slot
- May need to do a “garbage collection” when a large number of deletions are done
  - To improve the search time

Rehashing: Expanding the Hash Table

- As α increases, the time complexity also increases

Solution:

- Increase the size of hash table (doubled)
- Rehash all keys into new larger hash table

Summary

- Exhaustive Algorithm: Sequential Search: O(n)
- Decrease-and-conquer Algorithm: Binary Search: $O(\log_2 n)$
- Data Structures: Hashing
    - Closed Hashing: Separate Chaining: $O(\alpha)$ on average
    - Open Hashing
        - Linear Probing
        - Quadratic Probing
        - Double Probing
    - Delete keys
    - Rehashing