

# Data Searching and Binary Search

Instructor: Meng-Fen Chiang

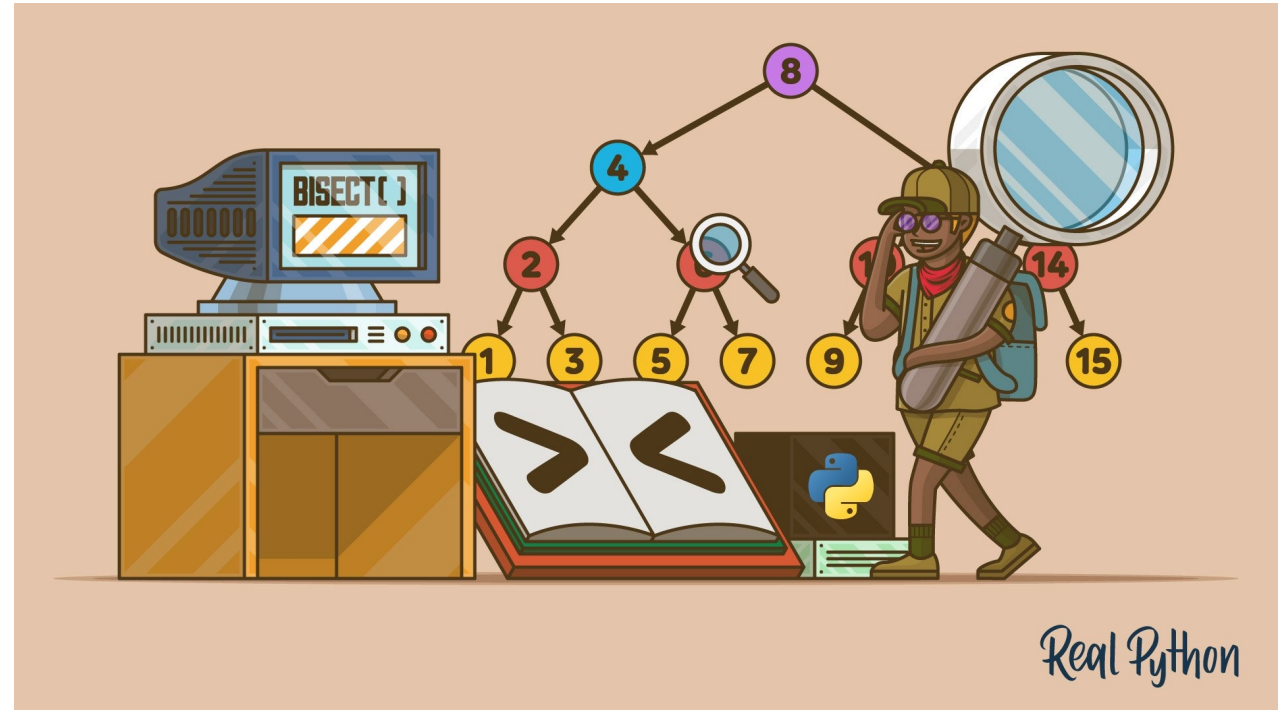
COMPCSI220: WEEK 9



Slides adapted from Kaiqi Zhao

# OUTLINE

- Definition of Search
- Abstract Data Type
  - Unsorted Lists / Sorted Lists
- Types of Search
  - Sequential Search
  - Binary Search
- Time Complexity Analysis



# Data Search in a Large Database

- Searching in a database D of **records**, such that each record has a **key** to use in the search.
- Example – search an album record by **Album ID (as key)**.

Album ID	Album	Song Title	Singer Name	Release Year
1	Courage	"Flying On My Own"	Celine Dion	2019
2	Mind Games	"South Dakota"	Jordy	2019
3	Clarity	"Broken"	Kim Petras	2020
4	Nibiru	"Reggaeton en Paris"	Ozuna	2019
5	Basking in the Glow	"Morning Song"	Oso Oso	2019

# Data Search in a Large Database

- Searching in a database  $D$  of records, such that each record has a key to use in the search.
- The search problem: Given a search key  $k$ , either
  - Return the record associated with  $k$  in  $D$  (a successful search: if  $k$  occurs several times, return any occurrence), or
  - Indicate that  $k$  is not found (an unsuccessful search).
- The purpose of the search
  - To access data in the record for processing, or
  - To update information in the record, or
  - To insert a new record or delete the record found.

# Table ADT

- **Definition** (Table ADT): The table ADT is a set of ordered pairs, or table entries  $(k, v)$  where  $k$  is a unique key and  $v$  is a data value associated with the key  $k$ .
- Types of Operations
  - **RETRIEVE** the entry  $(k, v)$  based on the key  $v$ ; if no entry exist, indicate the search is unsuccessful.
  - **REMOVE** the found entry from the table;
  - **UPDATE** its value  $v$ ;
  - **INSERT** a new entry with key  $k$  if the table has no such entry.

# Types of Search

- **Static search:** unalterable (fixed in advance) databases; no updates, deletions, or insertions.
- **Dynamic search:** alterable databases (allowable insertions, deletions, and updates).

Key		Associated value $v$		
Code	$k$	City	Country	State/Place
AKL	271	Auckland	New Zealand	North Island
DCA	2080	Washington	USA	District of Columbia (D.C.)
FRA	3822	Frankfurt	Germany	Hesse
SDF	12251	Louisville	USA	Kentucky

# Implementation

- **Basic implementations** of the table ADT: lists and trees.
- An **elementary operation**
  - An update of a list element or tree node, or
  - Comparison of two of them.

# Sequential Search in Unsorted Lists

- Starting at the head of a list and examining elements one by one until finding the desired key or reaching the end of the list.
- **Complexity.** Both successful and unsuccessful sequential search have worst-case and average-case time complexity  $\Theta(n)$ .
- **Proof:**
  - The **unsuccessful search** explores each of  $n$  keys, so the worst- and average-case time is  $\Theta(n)$
  - The **successful search** examines  $n$  keys in the worst case and  $\frac{n+1}{2}$  keys on the average, which is still  $\Theta(n)$
- The sequential search is the **ONLY option** for unsorted lists of records.
- A **sorted list implementation** allows for much better search based on the divide-and-conquer paradigm.



# Binary Search in a Sorted List of Records

Given a **sorted list**  $L = \{(k_i, v_i) : i = 1, \dots, n; k_1 < k_2 < \dots < k_n\}$

Recursive binary search for the key  $k$ :

1. If the list is empty, return “not found”, otherwise
  2. Choose the key  $k_m$  of the **middle element** of the list and
    - if  $k_m = k$ , return its record, otherwise
    - if  $k_m > k$ , make a recursive call on the head sublist, otherwise
    - if  $k_m < k$ , make a recursive call on the tail sublist.
- We can do this without recursion!

# Non-recursive (Iterative) Binary Search in Array

- The performance of binary search on an array is much better than on a linked list because of the constant time access to a given element.

---

**Algorithm 1** BinarySearch

---

```
1: function BinarySearch(a sorted integer  $\mathbf{k} = (k_0, k_1, \dots, k_{n-1})$  of  
   keys associated with items, a search key  $k$ )  
2:    $l \leftarrow 0; r \leftarrow n - 1$   
3:   while  $l \leq r$  do  $m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$   
4:     if  $k_m < k$  then  $l \leftarrow m + 1$   
5:     else if  $k_m > k$  then  $r \leftarrow m - 1$   
6:     else return  $m$   
7:   return ItemNotFound
```

---

# Faster Binary Search with Two-way Comparisons

---

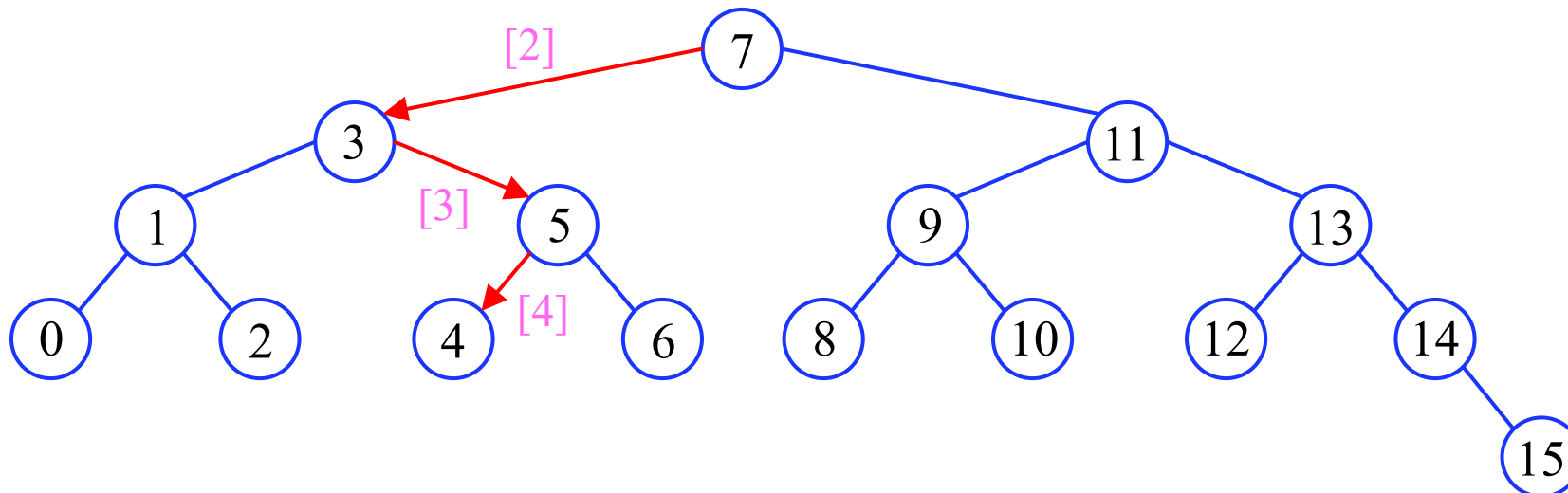
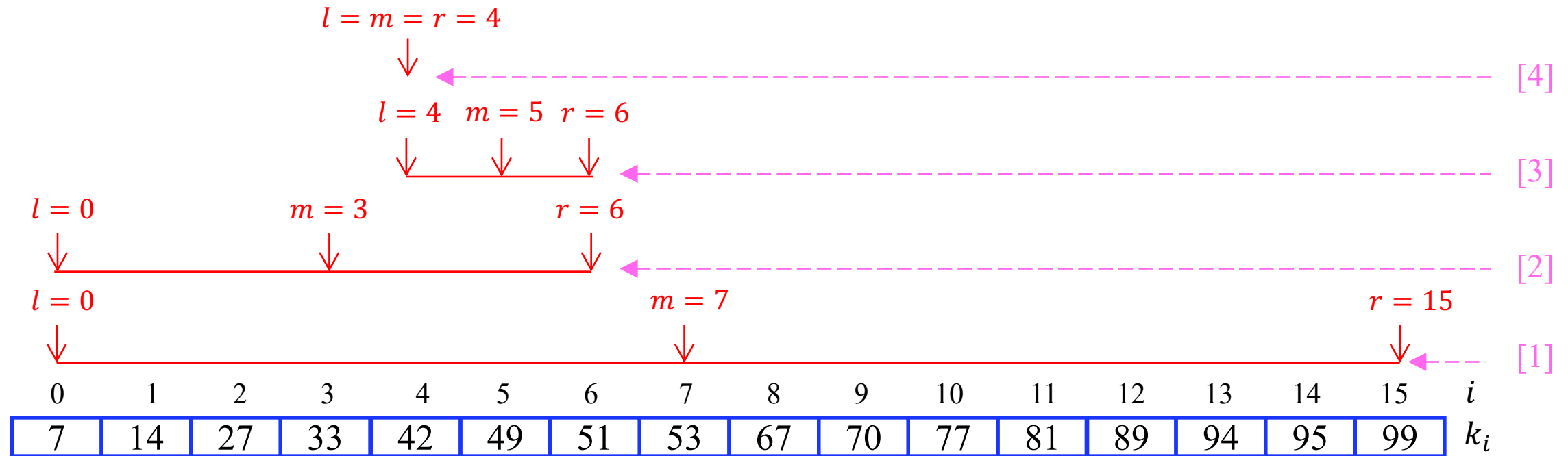
**Algorithm 2** BinarySearch

---

```
1: function BinarySearch2(a sorted integer  $\mathbf{k} = (k_0, k_1, \dots, k_{n-1})$  of  
   keys associated with items, a search key  $k$ )  
2:    $l \leftarrow 0; r \leftarrow n - 1$   
3:   while  $l < r$  do  $m \leftarrow \left\lfloor \frac{l+r}{2} \right\rfloor$   
4:     if  $k_m < k$  then  $l \leftarrow m + 1$   
5:     else  $r \leftarrow m$   
6:   if  $k_l = k$  then return  $l$   
7:   else return ItemNotFound
```

---

Example: Binary Search in Array  $\{k_0 = 7, \dots, k_{15} = 99\}$  for Key  $k=42$



# Time Complexity Analysis: Worst Case

- The worst-case time complexity of unsuccessful and successful binary search is  $\Theta(\log n)$ .
- The full binary tree of height  $h - 1$  has  $n = 2^h - 1$  keys (each internal node has 2 children)
  - The comparison tree height is  $h$  with only the last level not full.
  - $l + 1$  comparisons to find a key at level  $l$ .
  - The worst case:  $h + 1 = \lfloor \log_2 n \rfloor + 1$  comparisons.

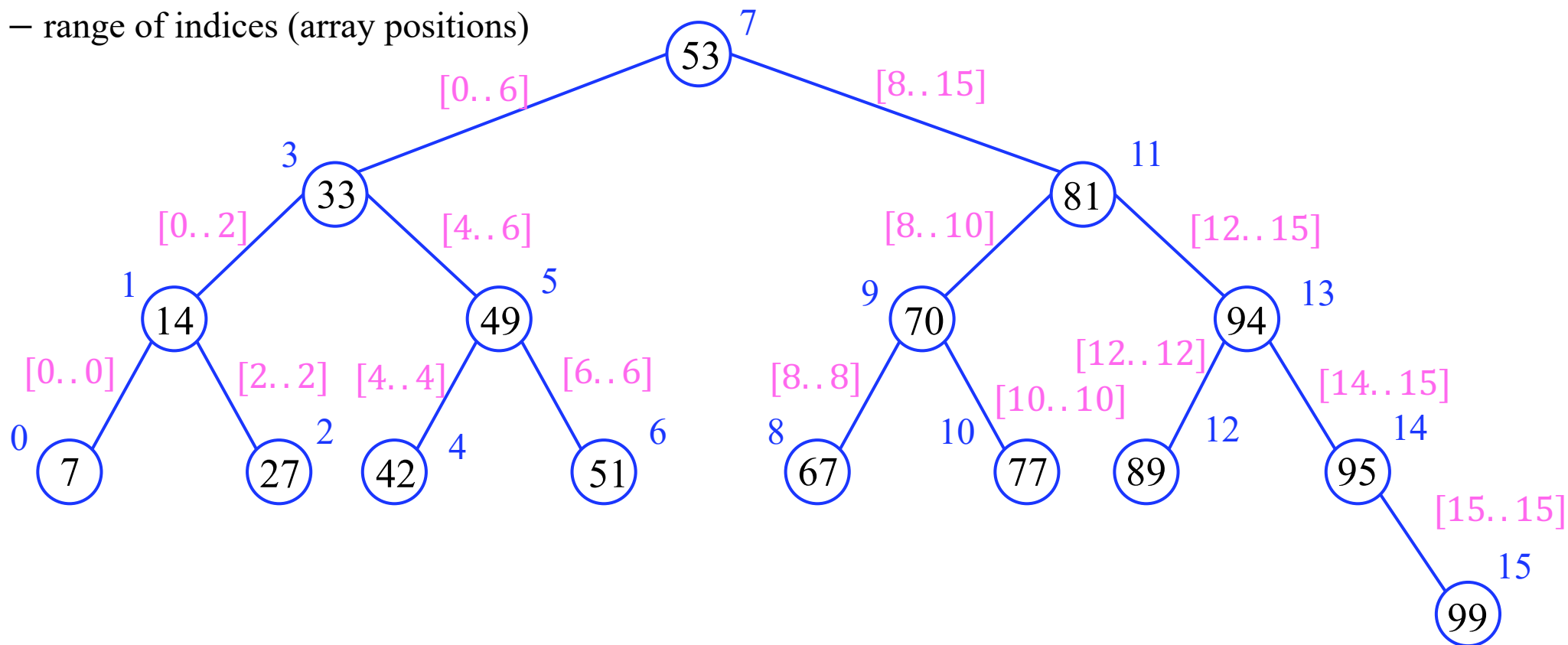
# Time Complexity Analysis: Average Case

- **Lemma:** The average-case time complexity of successful and unsuccessful binary search in a balanced tree is  $\Theta(\log n)$ .
- **Proof:** The height of the tree is  $h = \lfloor \log_2 n \rfloor$ 
  - At least half of the tree nodes have a depth at least  $h - 1$ .
  - The average depth over all nodes is at least  $\frac{h-1}{2}$  and at most  $h$ , so that it is  $\Theta(\log n)$
  - The average depth over all nodes of an arbitrary (not necessarily balanced) binary tree is  $\Omega(\log n)$ .
- The expected search time for an arbitrary balanced tree is equal to the average balanced tree depth  $\Theta(\log n)$ .

# Tree Structure of Binary Search: Binary Search Tree

7	14	27	33	42	49	51	53	67	70	77	81	89	94	95	99
---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

$[l..r]$  – range of indices (array positions)



# SUMMARY

- Definition of Search
- Sequential Search on Unsorted Lists
- Binary Search on Sorted Lists
  - Recursive Binary Search
  - Faster Binary Search
- Time Complexity Analysis

