Selection, Searching, Hash Maps, Sorting

CMPE 180A Data Structures and Algorithms in Python

Week 7

Week 7 Outline

- Term Project: data set selection, groups, deliverables
- Homework: HW3, HW4
- Searching:
    - sequential search
    - binary search tree
    - hashing
- Sorting algorithms:
    - selection sort
    - bubble sort
    - merge sort
    - quick sort
    - insertion sort
    - shell sort

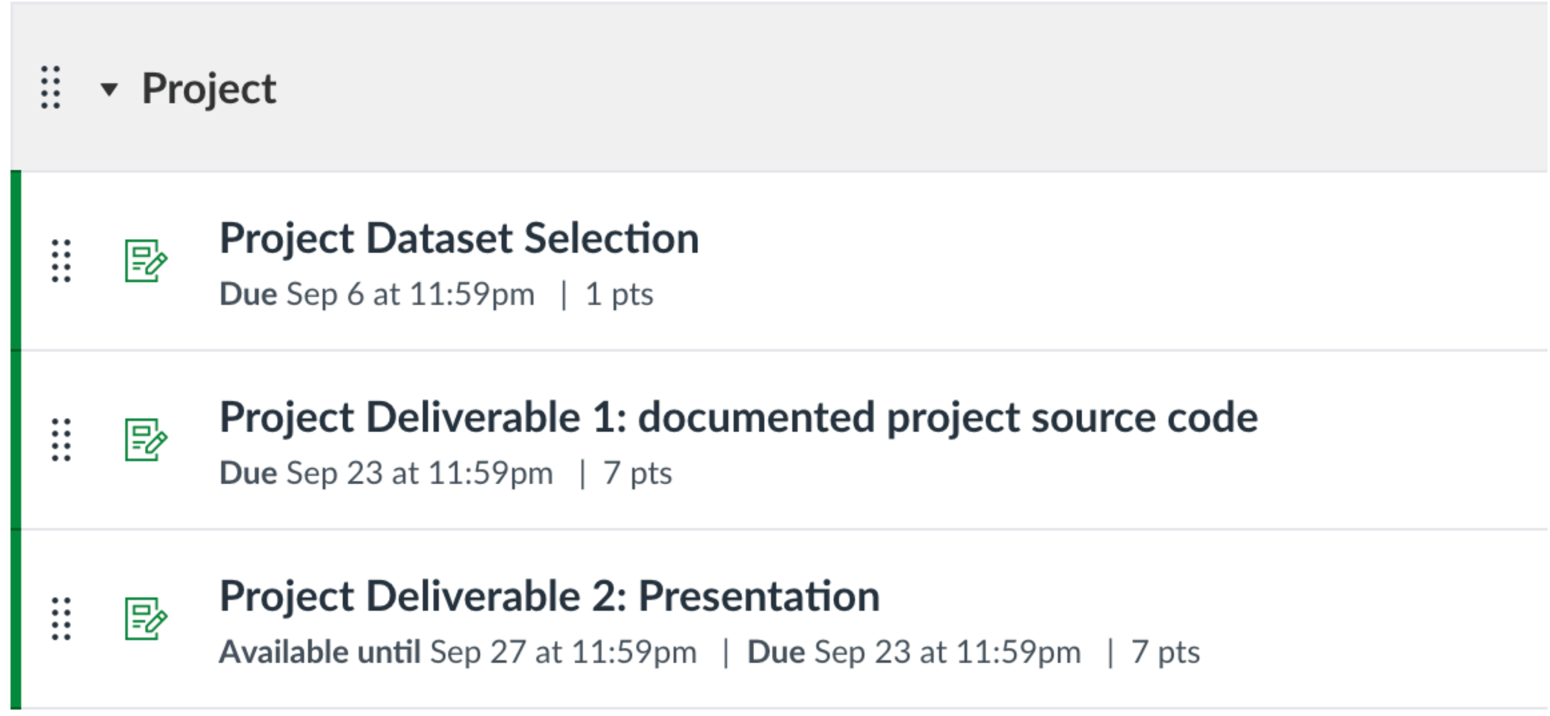
Project Datasets fro EDA

Select one for the term project

- **2024 Fortune 1000 Companies** (https://www.kaggle.com/datasets/jeannicolasduval/2024-fortune-1000-companies)
- **Payment Card Fraud Detection 2025 (** https://www.kaggle.com/datasets/pratyushpuri/payment-card-fraud-detection-with-ml-models-2025)
- **Electric Vehicle Population Data 2025** (https://www.kaggle.com/datasets/yanghu583/electric-vehicle-population-data-2025)
- **League of Legends Champions** (Season 15, 25.13) LoL Champ Statistics from the Wiki (https://www.kaggle.com/datasets/laurenainsleyhaines/league-of-legends-champions-season-15-25-13)
- **Superstore Dataset** (https://www.kaggle.com/datasets/vivek468/superstore-dataset-final)
- **Spotify Tracks DB** (https://www.kaggle.com/datasets/zaheenhamidani/ultimate-spotify-tracks-db)
- **2024 Fortune 1000 Companies** (https://www.kaggle.com/datasets/jeannicolasduval/2024-fortune-1000-companies)
- **Premier League Matches** (https://https://www.kaggle.com/datasets/mhmdkardosha/premier-league-matches)
- **US Airline Flight Routes and Fares** (https://https://www.kaggle.com/code/nitikagupta29/us-airline-flight-routes-and-fares)
- **Elite College Admissions** (https://https://www.kaggle.com/datasets/mexwell/elite-college-admissions)
- **Alzheimer's Disease Patient Data** (https://https://www.kaggle.com/datasets/muhammadehsan02/alzheimers-disease-patient-data)
- **NBA Players Stats 23/24** (https://https://www.kaggle.com/datasets/orkunaktas/nba-players-stats-2324)
- **Facebook Metrics Dataset** (https://https://www.kaggle.com/datasets/dileeppatchaone/facebook-metrics-dataset-of-cosmetic-brand)

Term Project
15 points

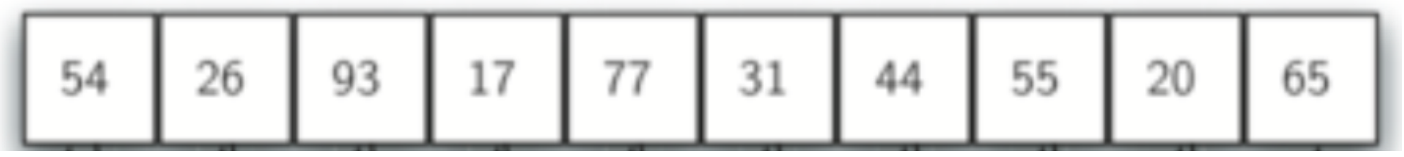
- Groups created in CMPE180A Canvas LMS
- Term project group submission points created
- Action points:
  - select the data set to work on
  - Agree on time/way to align
  - All: understand the data set
  - Analysis and EDA: split the work or in parallel
- Deliverables: 1) Data set selection, 2) Documented code, 3) Presentation
- Prepare the presentation slides
  - Everyone should present
  - Explain the data set: what it represents
  - Explain features in the data set
  - Visualize the dataset
  - Predict



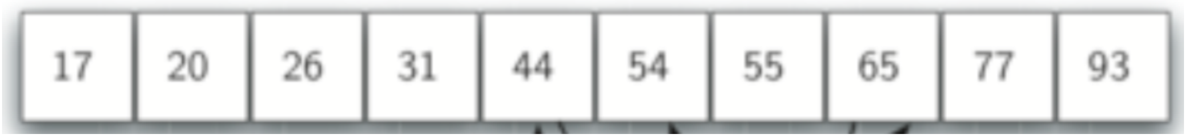
Searching

Searching

- Searching is the algorithmic process of finding a particular item in a collection of items and returning **True** or **False**



Unordered list of integers



Ordered list of integers

Searching Unordered List

- Sequential search: O(n)

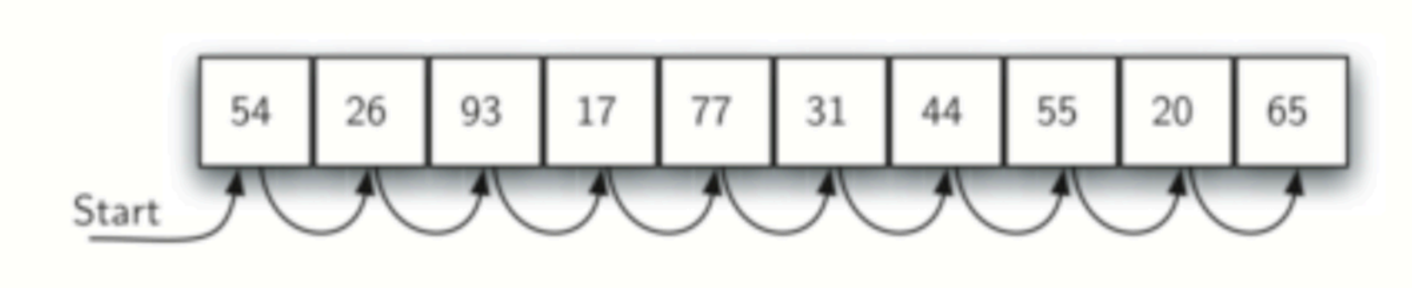


Figure 1: Sequential Search of a List of Integers

Table 1: Comparisons Used in a Sequential Search of an Unordered List

Case	Best Case	Worst Case	Average Case
item is present	1	n	$\frac{n}{2}$
item is not present	n	n	n

Searching Ordered List

- Sequential: O(n)
- Binary Search: O(log n)

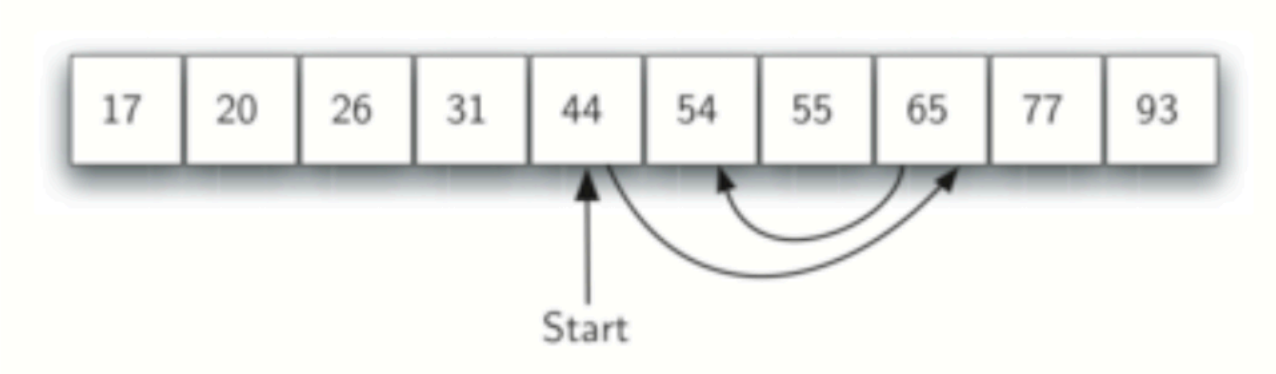


Table 2: Comparisons Used in Sequential Search of an Ordered List

item is present	1	n	$\frac{n}{2}$
item not present	1	n	$\frac{n}{2}$

https://runestone.academy/ns/books/published/pythonds3/SortSearch/TheSequentialSearch.html

Q-3: 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**

Q-4: 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**

https://runestone.academy/ns/books/published/pythonds3/SortSearch/TheBinarySearch.html

```
midpoint = len(a_list) // 2
```

Q-3: Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to find the key 8.

- **A. 11, 5, 6, 8**
- **B. 12, 6, 11, 8**
- **C. 3, 5, 6, 8**
- **D. 18, 12, 6, 8**

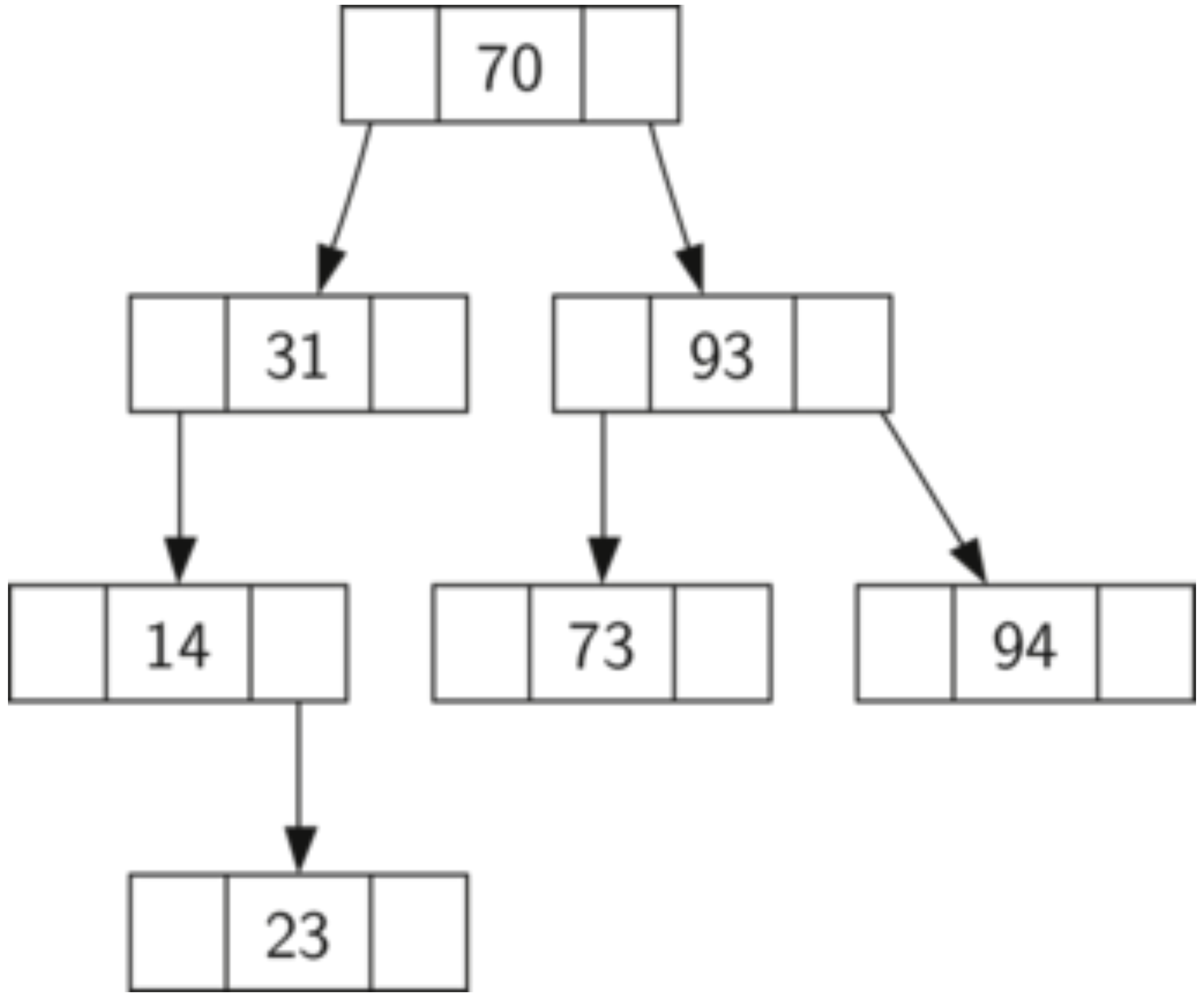
Q-4: Suppose you have the following sorted list [3, 5, 6, 8, 11, 12, 14, 15, 17, 18] and are using the recursive binary search algorithm. Which group of numbers correctly shows the sequence of comparisons used to search for the key 16?

- **A. 11, 14, 17**
- **B. 18, 17, 15**
- **C. 14, 17, 15**
- **D. 12, 17, 15**

Binary Search Tree

Binary Search Tree (BST) Property

- Keys that are less than the parent are found in the left subtree, and keys that are greater than the parent are found in the right subtree
- Showing **keys** only

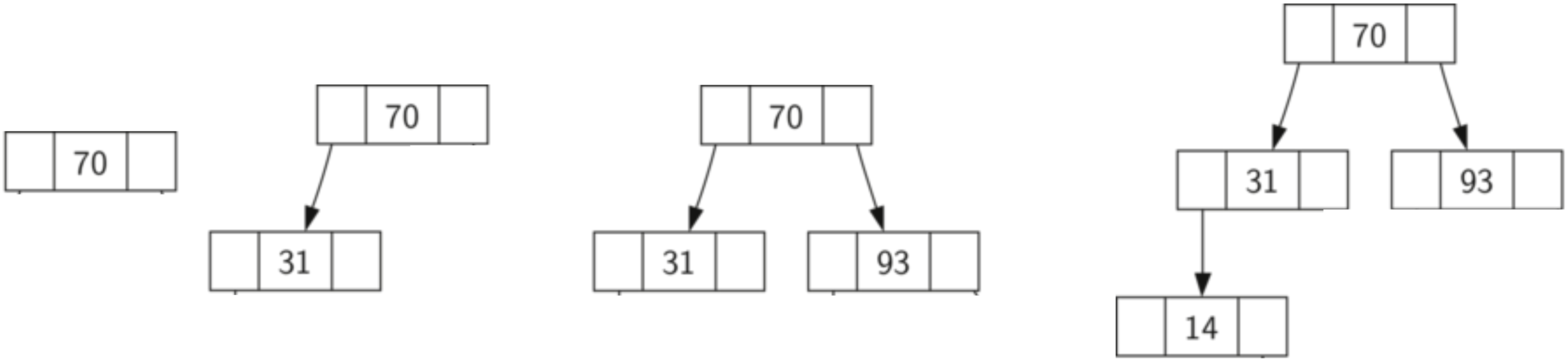


Note: Play Slideshow

BST creation

70, 31, 93, 14, 23, 73, 94

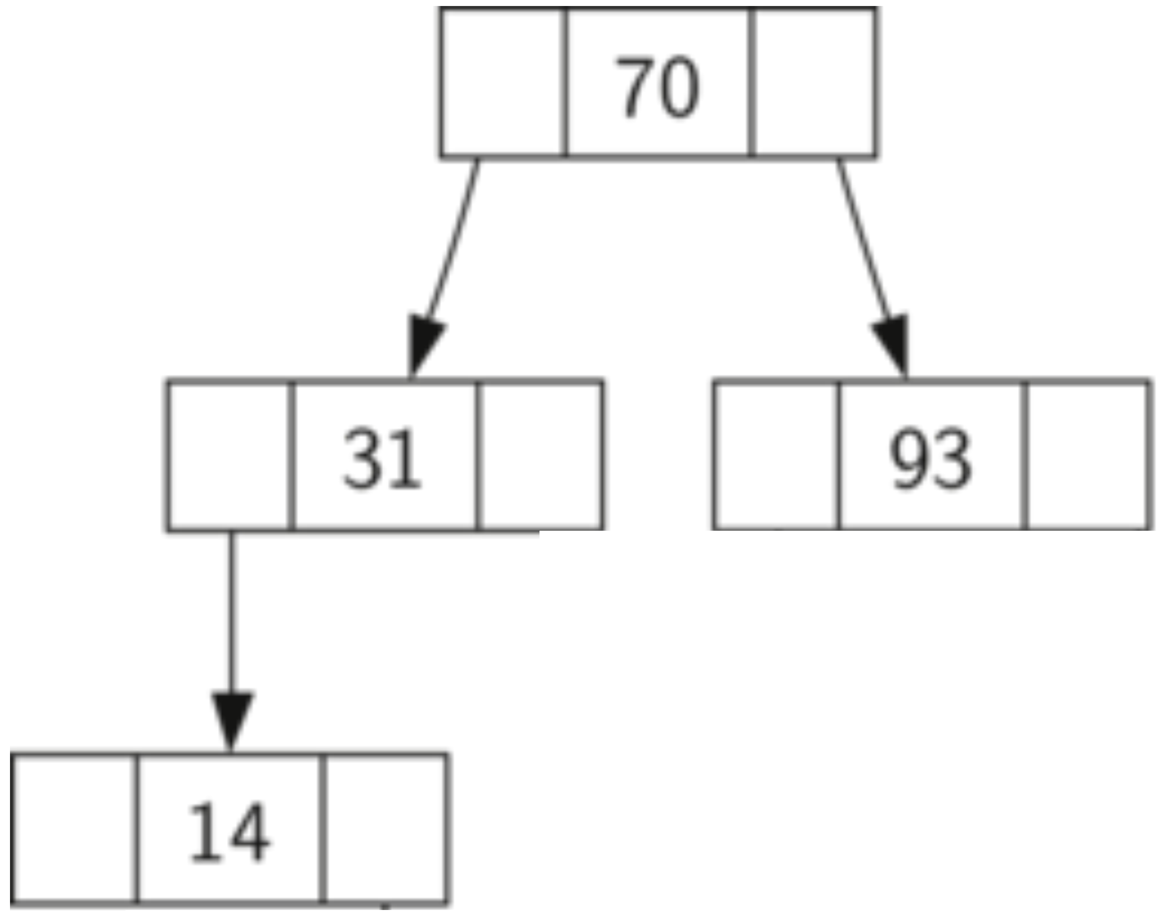
- Inserting nodes one by one:

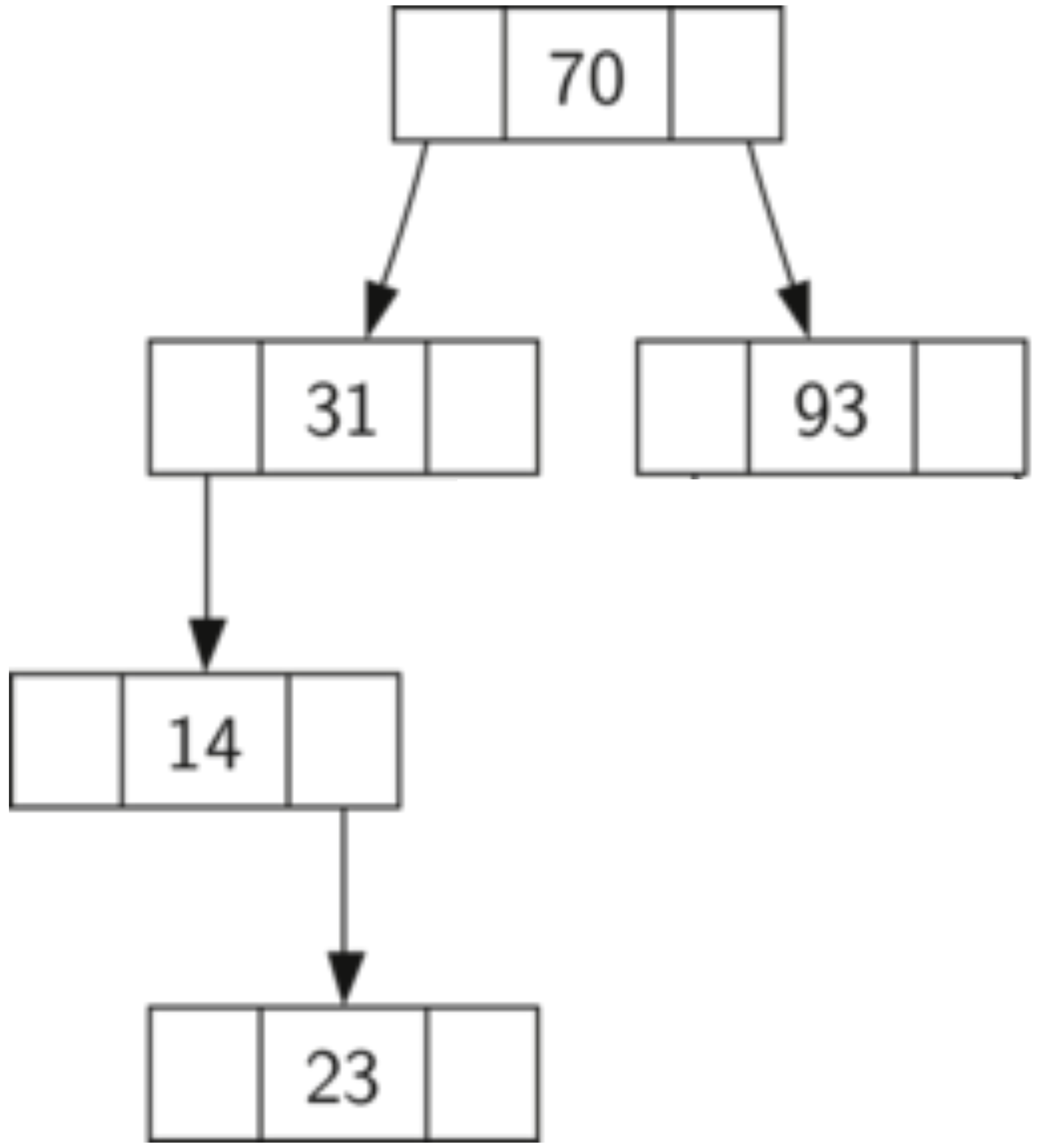


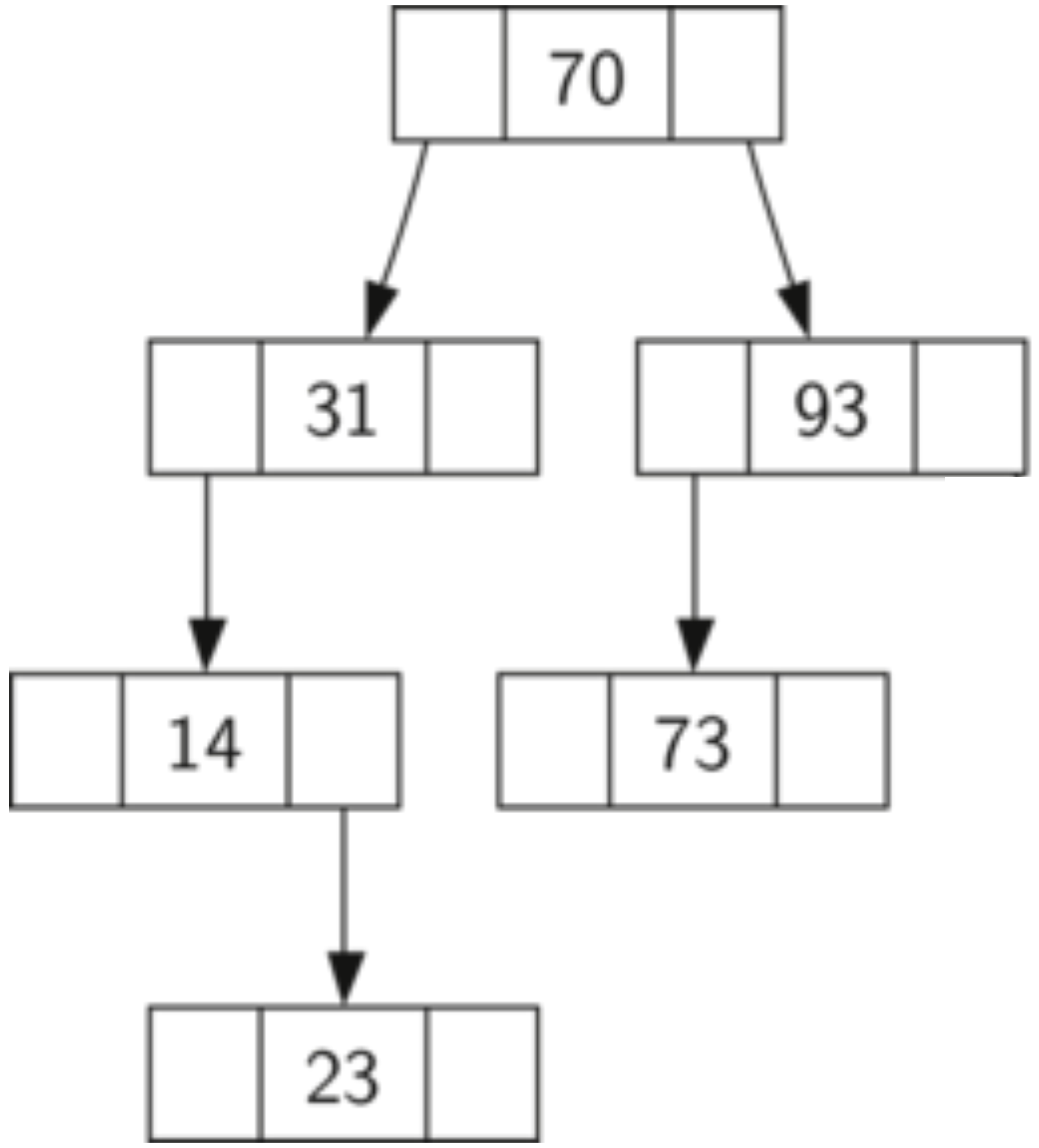
BST creation

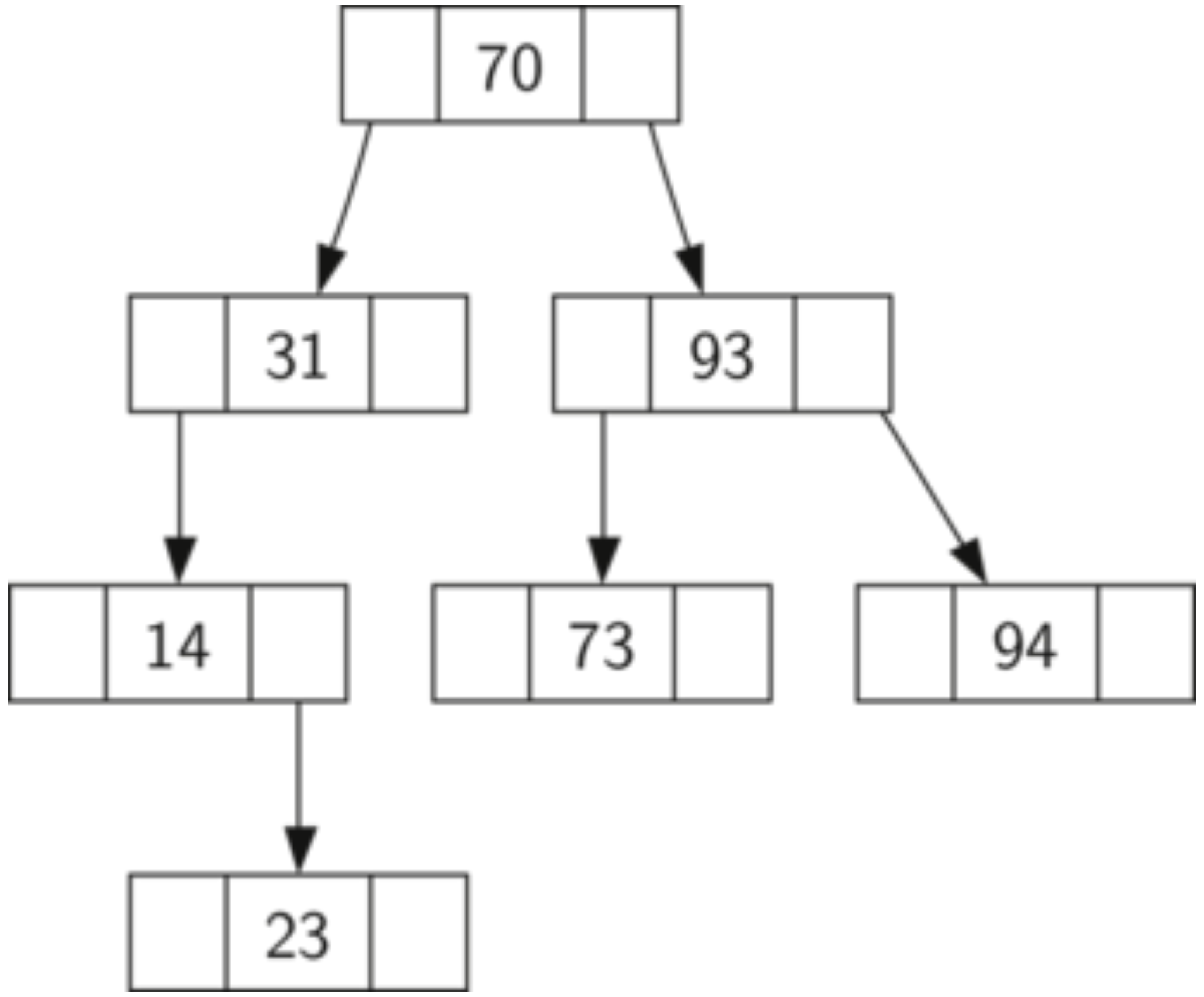
70, 31, 93, 14, 23, 73, 94

- Inserting nodes one by one:





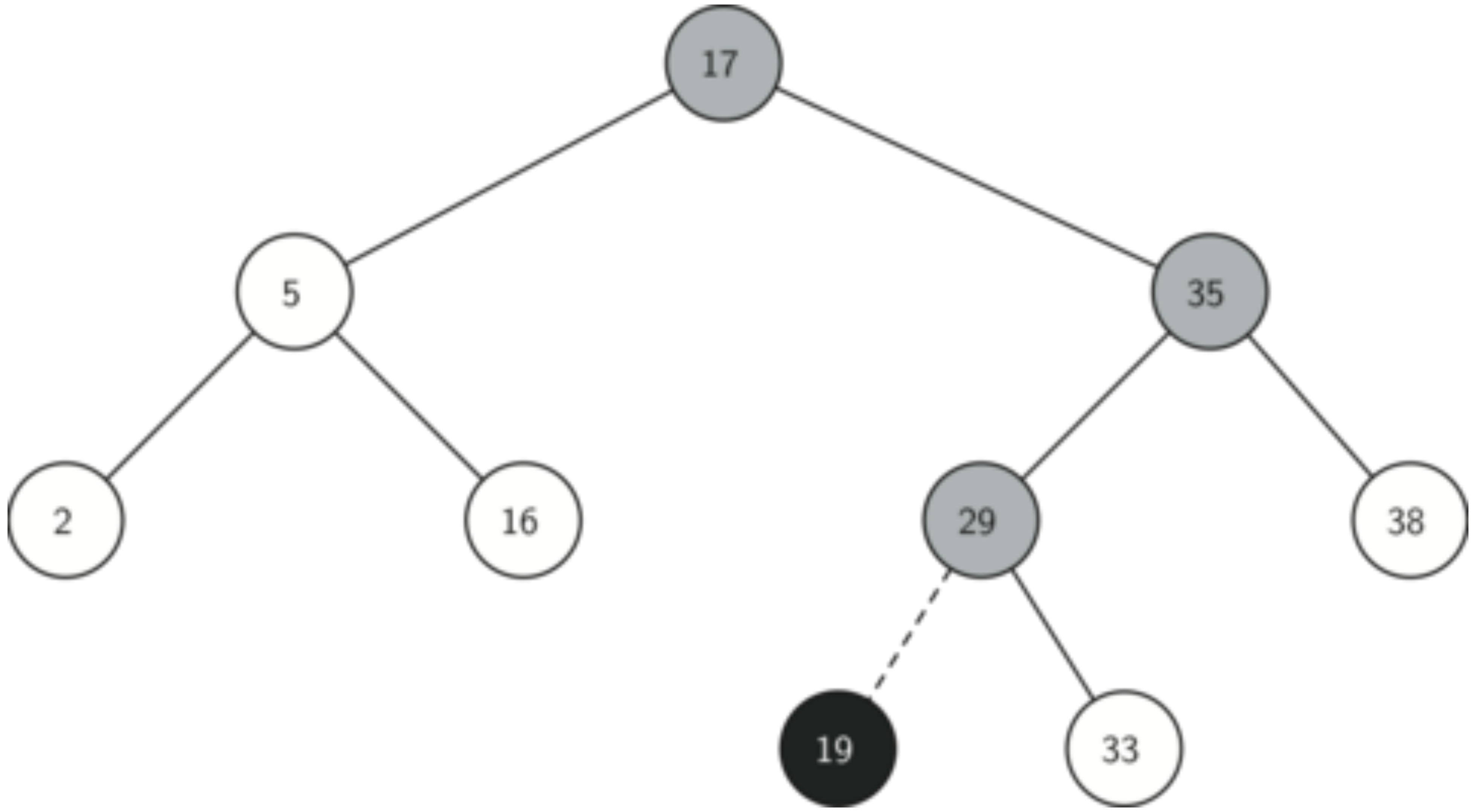




Search Tree Operations

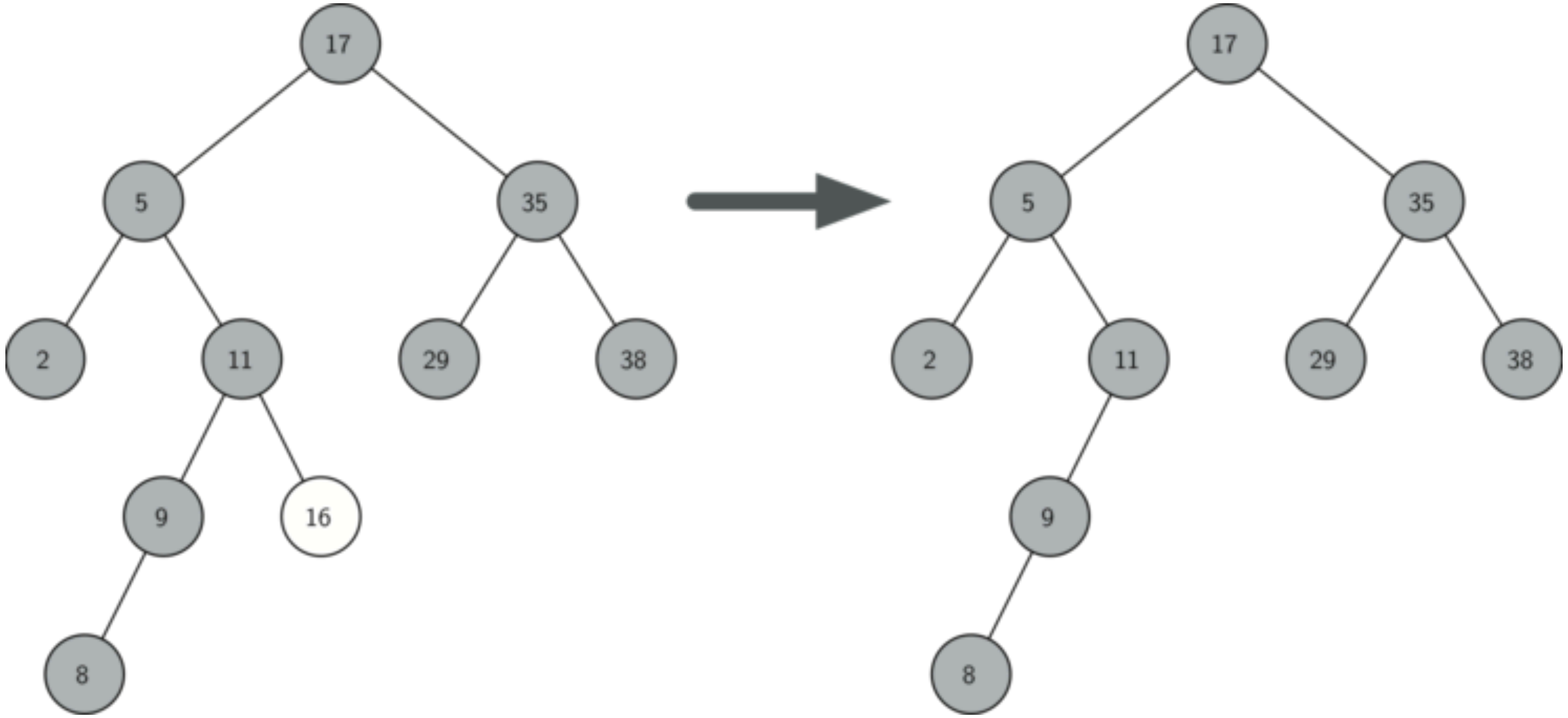
- `Map()` Create a new, empty map.
- `put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
- `get(key)` Given a key, return the value stored in the map or `None` otherwise.
- `del` Delete the key-value pair from the map using a statement of the form `del map[key]`.
- `len()` Return the number of key-value pairs stored in the map.
- `in` Return `True` for a statement of the form `key in map`, if the given key is in the map.

Inserting a Node with Key = 19



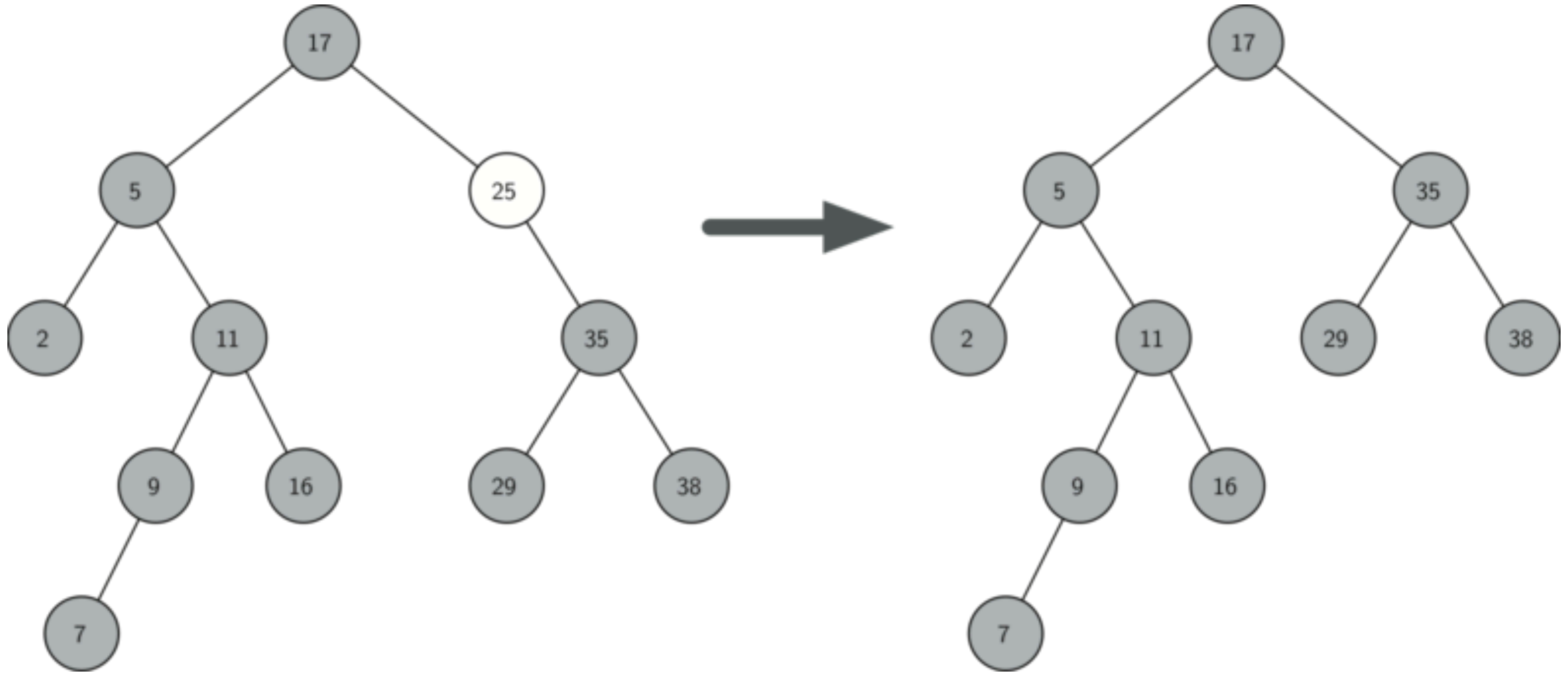
Deleting Node 16, a Node without Children

-



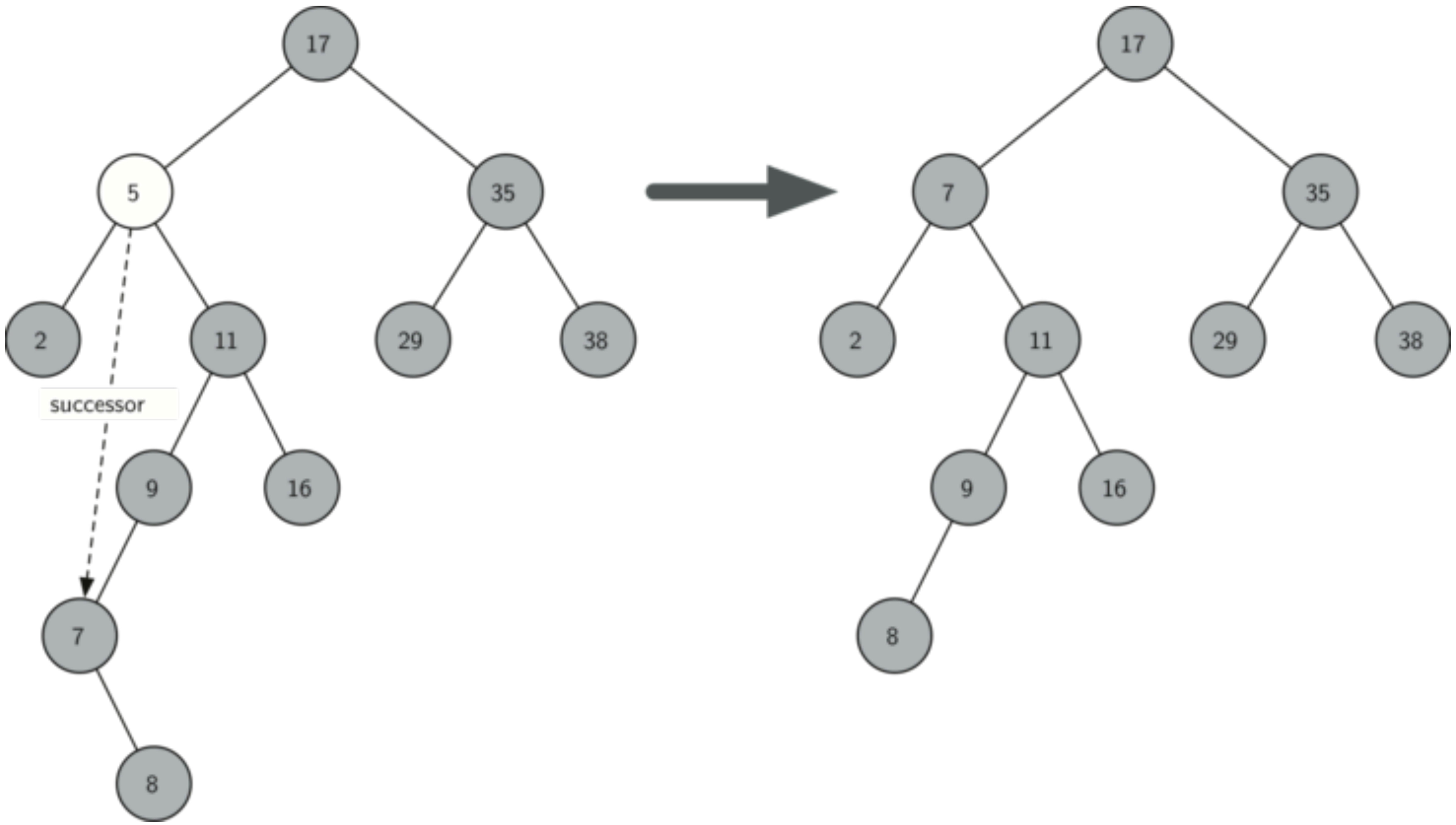
Deleting Node 25, a Node That Has a Single Child

-



Deleting Node 5, a Node with Two Children

- Replace the node with the smallest element from the right subtree (or the largest element from the left subtree)



Hash Tables

Hash table

- a data structure that can be searched in O(1) time
- a collection of items which are stored in such a way as to make it easy to find them later
- **slot -** position of the hash table; hash function: mappings to slots

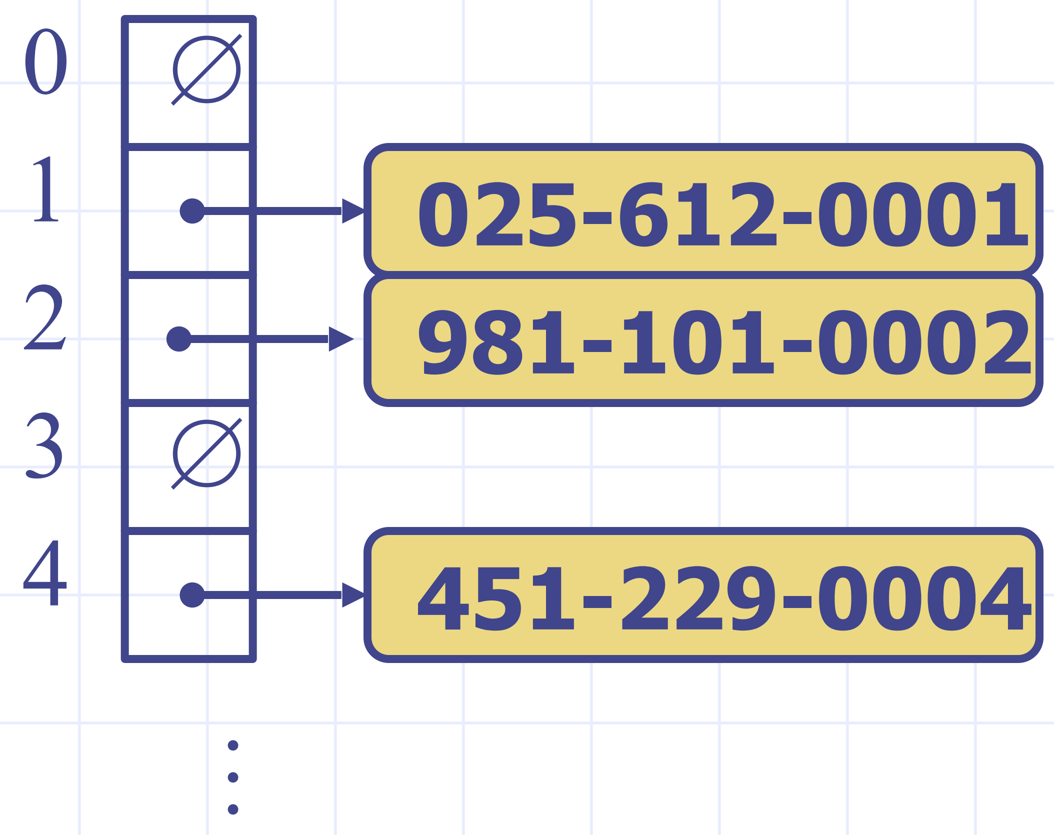
0	1	2	3	4	5	6	7	8	9	10
None	None	None	None	None	None	None	None	None	None	None

- Hash Table with 11 empty slots
- Load factor: $\lambda = \frac{number\_of\_items}{table\_size}$
- Hash function: $h(item) = item \,\%\, 11$

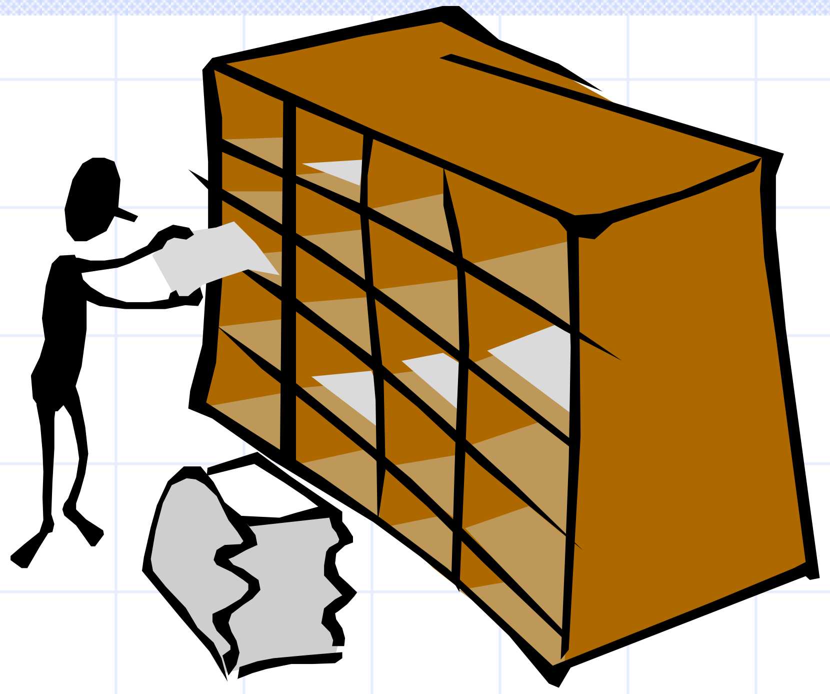
0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

More General Kinds of Keys

- But what should we do if our keys are not integers in the range from 0 to N – 1?
  - Use a **hash function** to map general keys to corresponding indices in a table.
  - For instance, the last four digits of a Social Security number.



Hash Functions and Hash Tables



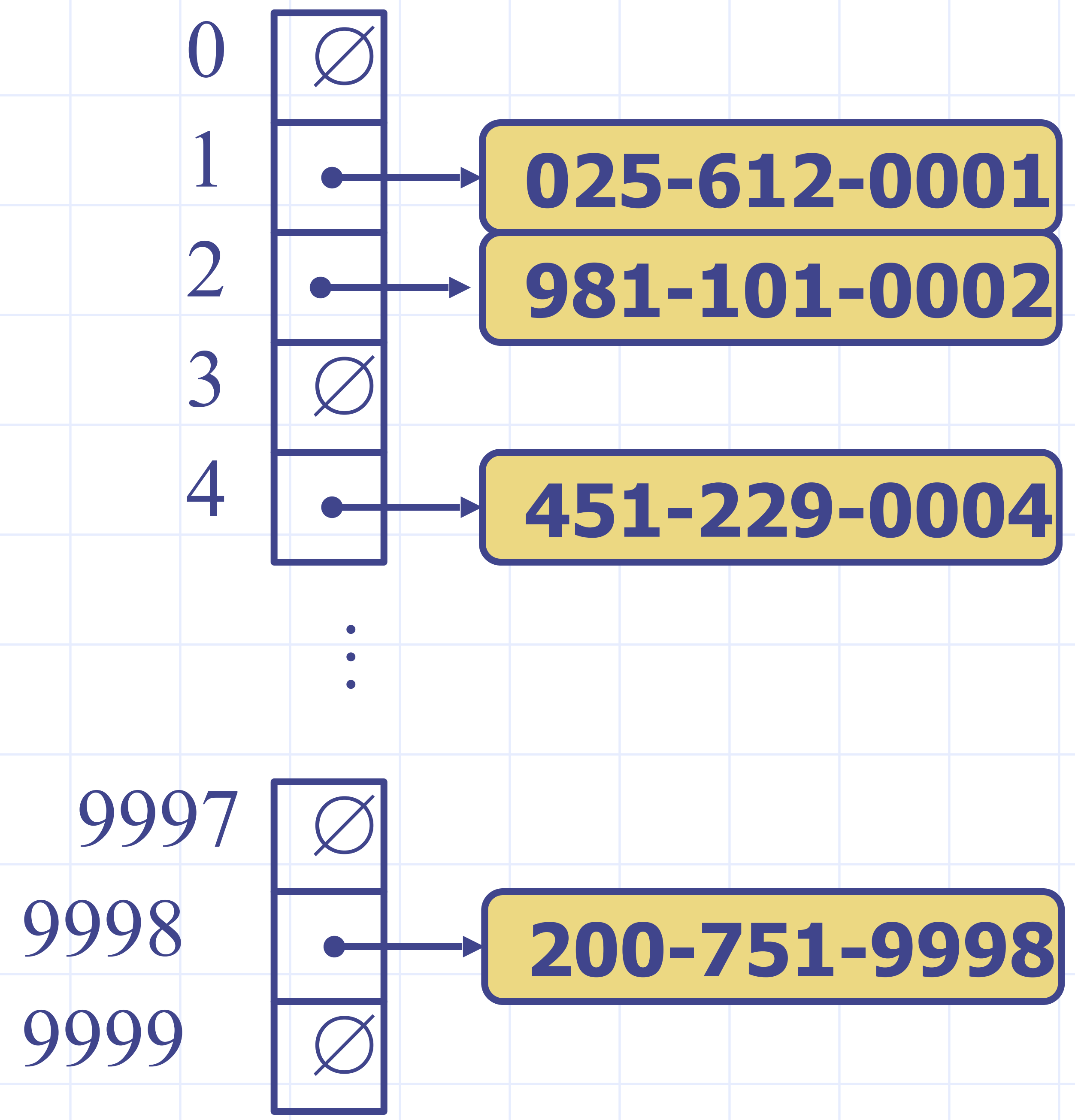
- A hash function h maps keys of a given type to integers in a fixed interval $[0, N-1]$
- Example:

  $$h(x) = x \bmod N$$

  is a hash function for integer keys
- The integer $h(x)$ is called the hash value of key x
- A hash table for a given key type consists of
  - Hash function h
  - Array (called table) of size N
- When implementing a map with a hash table, the goal is to store item (k, o) at index $i = h(k)$

SSN Example

- We design a hash table for a map storing entries as (SSN, Name), where SSN (social security number) is a nine-digit positive integer
- Our hash table uses an array of size $N = 10{,}000$ and the hash function
  $h(x)$ = last four digits of x



Hash Functions

- A hash function is usually specified as the composition of two functions:

  Hash code:

  h_1: keys $\rightarrow$ integers

  Compression function:

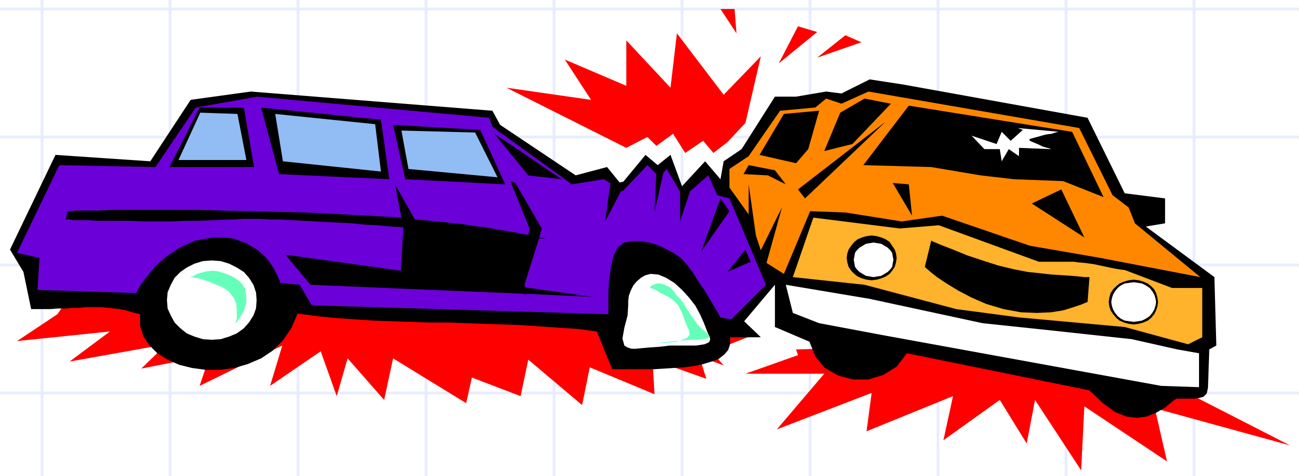
  h_2: integers $\rightarrow [0, N-1]$

- The hash code is applied first, and the compression function is applied next on the result, i.e.,

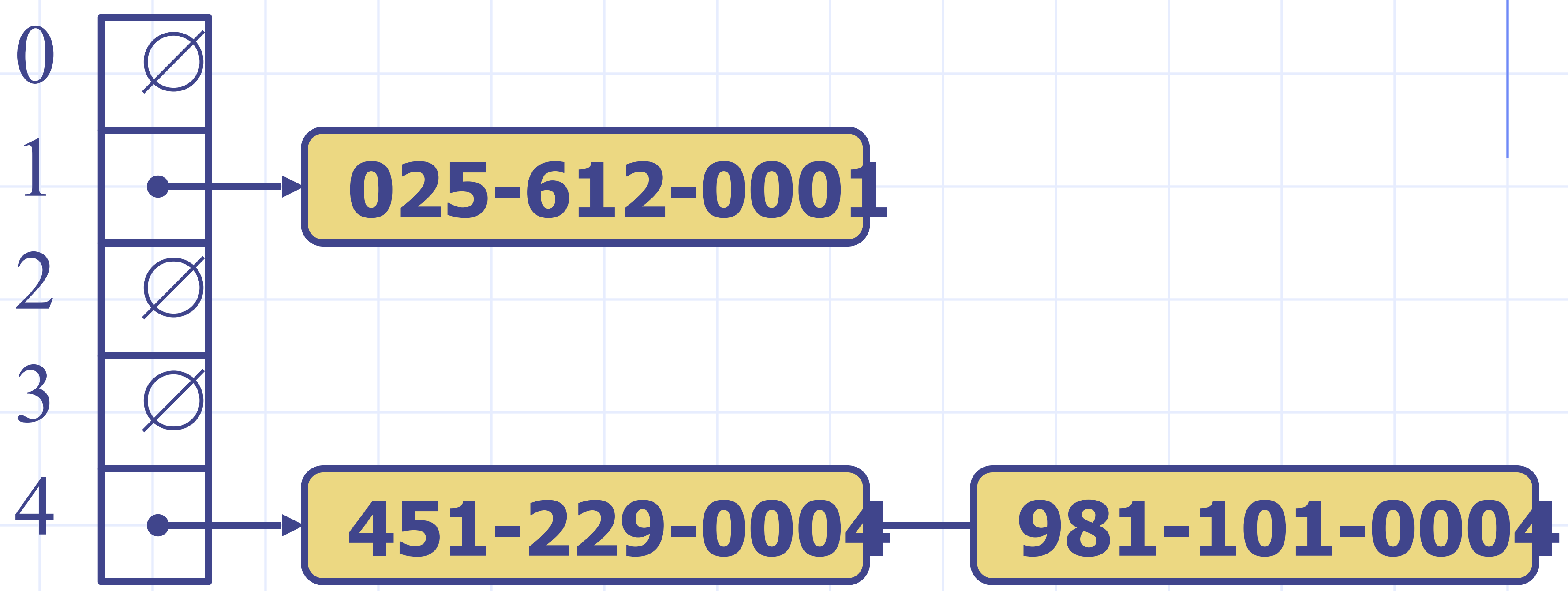
  $$h(x) = h_2(h_1(x))$$

- The goal of the hash function is to "disperse" the keys in an apparently random way

Collision Handling



- Collisions occur when different elements are mapped to the same cell
- Separate Chaining: let each cell in the table point to a linked list of entries that map there



- Separate chaining is simple, but requires additional memory outside the table

Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table
- **Linear probing**: handles collisions by placing the colliding item in the next (circularly) available table cell - **rehashing**
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes - clustering
- Search: probing until found or empty slot

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

Assignment:

insert given keys into the hash table

Linear Probing

- **Open addressing**: the colliding item is placed in a different cell of the table
- **Linear probing**: handles collisions by placing the colliding item in the next (circularly) available table cell - **rehashing**
- Each table cell inspected is referred to as a "probe"
- Colliding items lump together, causing future collisions to cause a longer sequence of probes - clustering
- Search: probing until found or empty slot

- Example:
  - $h(x) = x \bmod 13$
  - Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

0	1	2	3	4	5	6	7	8	9	10	11	12

⇩

		41			18	44	59	32	22	31	73	
0	1	2	3	4	5	6	7	8	9	10	11	12

https://runestone.academy/ns/books/published/pythonds3/SortSearch/Hashing.html

Q-1: In a hash table of size 13 which index positions would the following two keys map to? 27, 130

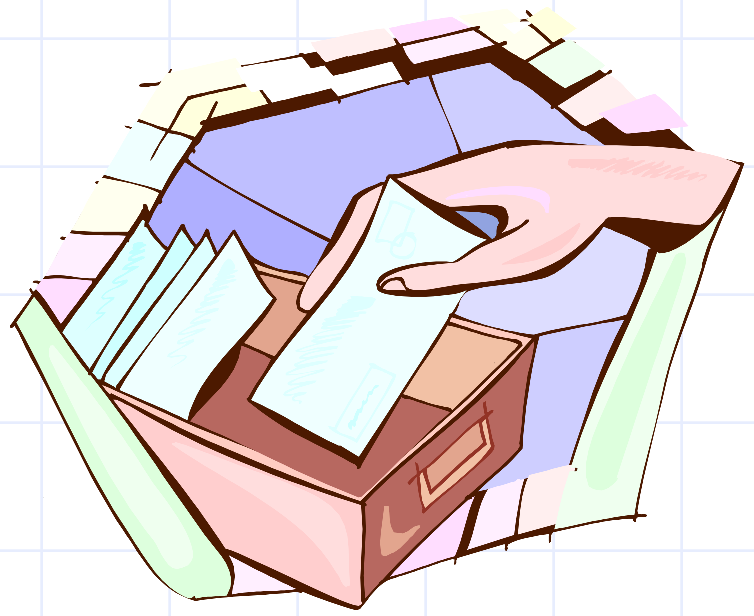
- **A. 1, 10**
- **B. 13, 0**
- **C. 1, 0**
- **D. 2, 3**

Q-2: Suppose you are given the following set of keys to insert into a hash table that holds exactly 11 values: 113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99 Which of the following best demonstrates the contents of the hash table after all the keys have been inserted using linear probing?

- **A. 100, __, __, 113, 114, 105, 116, 117, 97, 108, 99**
- **B. 99, 100, __, 113, 114, __, 116, 117, 105, 97, 108**
- **C. 100, 113, 117, 97, 14, 108, 116, 105, 99, __, __**
- **D. 117, 114, 108, 116, 105, 99, __, __, 97, 100, 113**

a collision resolution technique used in open addressing hash tables

Double Hashing



- Double hashing uses a secondary hash function $d(k)$ and handles collisions by placing an item in the first available cell of the series
$$(i + jd(k)) \bmod N$$
for $j = 0, 1, \ldots, N-1$
- The secondary hash function $d(k)$ cannot have zero values
- The table size N must be a prime to allow probing of all the cells
- ***k*****: probe,** ***i*****: the key,** ***N*****: the size of the hash table,** ***j*****: the probe attempt**

- Common choice of compression function for the secondary hash function:
$$d_2(k) = q - k \bmod q$$
where
  - $q < N$
  - q is a prime
- The possible values for $d_2(k)$ are
$$1, 2, \ldots, q$$

Hashing Analysis

For a successful search using open addressing with linear probing, the average number of comparisons is approximately

$$\frac{1}{2}\left(1 + \frac{1}{1-\lambda}\right)$$

and an unsuccessful search gives

$$\frac{1}{2}\left(1 + \left(\frac{1}{1-\lambda}\right)^2\right)$$

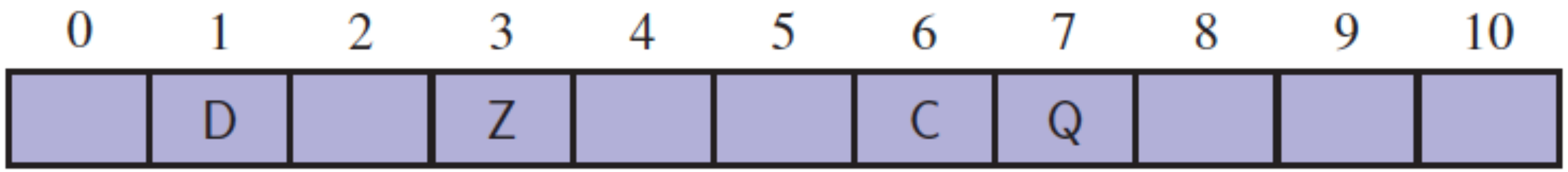
If we are using chaining, the average number of comparisons is

$$1 + \frac{\lambda}{2}$$

for the successful case, and simply the load factor λ comparisons if the search is unsuccessful.

Recall the notion of a Map

- Intuitively, a map M supports the abstraction of array using keys as indices with a syntax such as M[k].
- a map with n items uses keys that are known to be integers in a range from 0 to N − 1, for some $N \geq n$.



ADT Map

`Map()` Create a new, empty map. It returns an empty map collection.
`put(key,val)` Add a new key-value pair to the map. If the key is already in the map then replace the old value with the new value.
`get(key)` Given a key, return the value stored in the map or `None` otherwise.
`del` Delete the key-value pair from the map using a statement of the form `del map[key]`.
`len()` Return the number of key-value pairs stored in the map.
`in` Return `True` for a statement of the form `key in map`, if the given key is in the map, `False` otherwise.

Hash table

- A data structure used to store keys, with corresponding values
- Inserts, deletes, and lookups run O(1) in time
- Key is stored in the array locations - “slots” based on “hash code”
- “hash code” is an integer computed from the key by a hash functions
- Good hash function: objects are distributed uniformly across the array locations I.e. keys are spread uniformly
- Collision: mapping to same location - maintaining linked list of objects
- Different from sorted array: keys do not have to appear in order
- Equal keys should have equal hash codes
- Keys should be immutable objects
- Key update: first remove it, then update it and add it back

Sorting

Sorting

- the process of placing elements from a collection in some kind of order
- rearranging a collection of items into increasing or decreasing order
- preprocessing step to make searching the collection faster or identifying similar items

Sorting algorithms

- Naive sorting algorithms run in $O(n^2)$
- Heapsort, merge sort, quick sort: ~$O(n*log(n))$

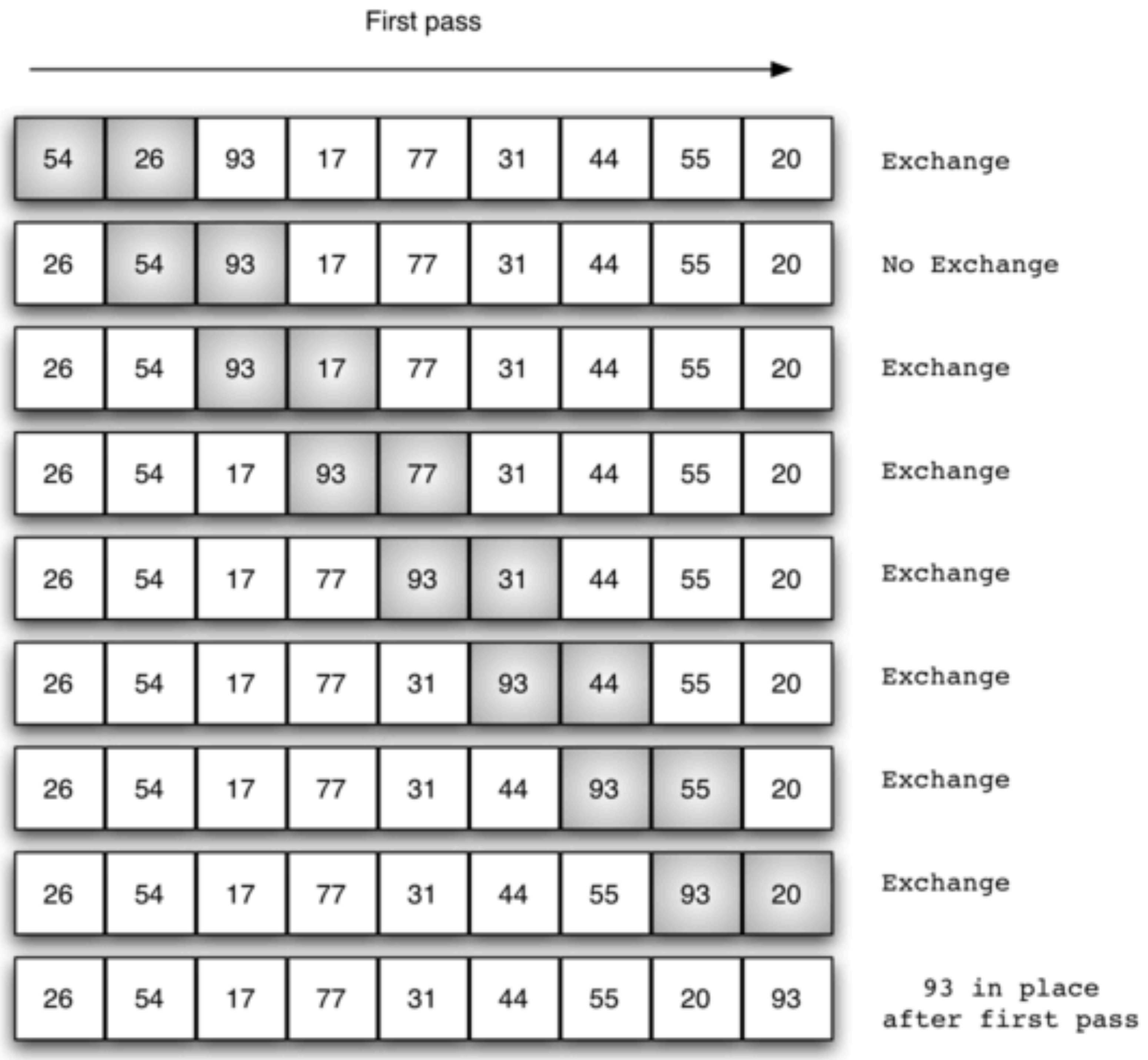
Python functions

- sort() method: stable in-place sort for list objects
- sorted(): returns a new list
- Complexity

Method	Algorithm	Best Case	Average Case	Worst Case	Space Complexity
`list.sort()`	Timsort	O(n)	O(n log n)	O(n log n)	O(n) (typ. O(log n))
`sorted()`	Timsort	O(n)	O(n log n)	O(n log n)	O(n) (typ. O(log n))

The Bubble Sort

- Makes multiple passes through a list
- It compares adjacent items and exchanges those that are out of order
- Each pass through the list places the next largest value in its proper place
- Each item bubbles up to the location where it belongs
- $O(n^2)$

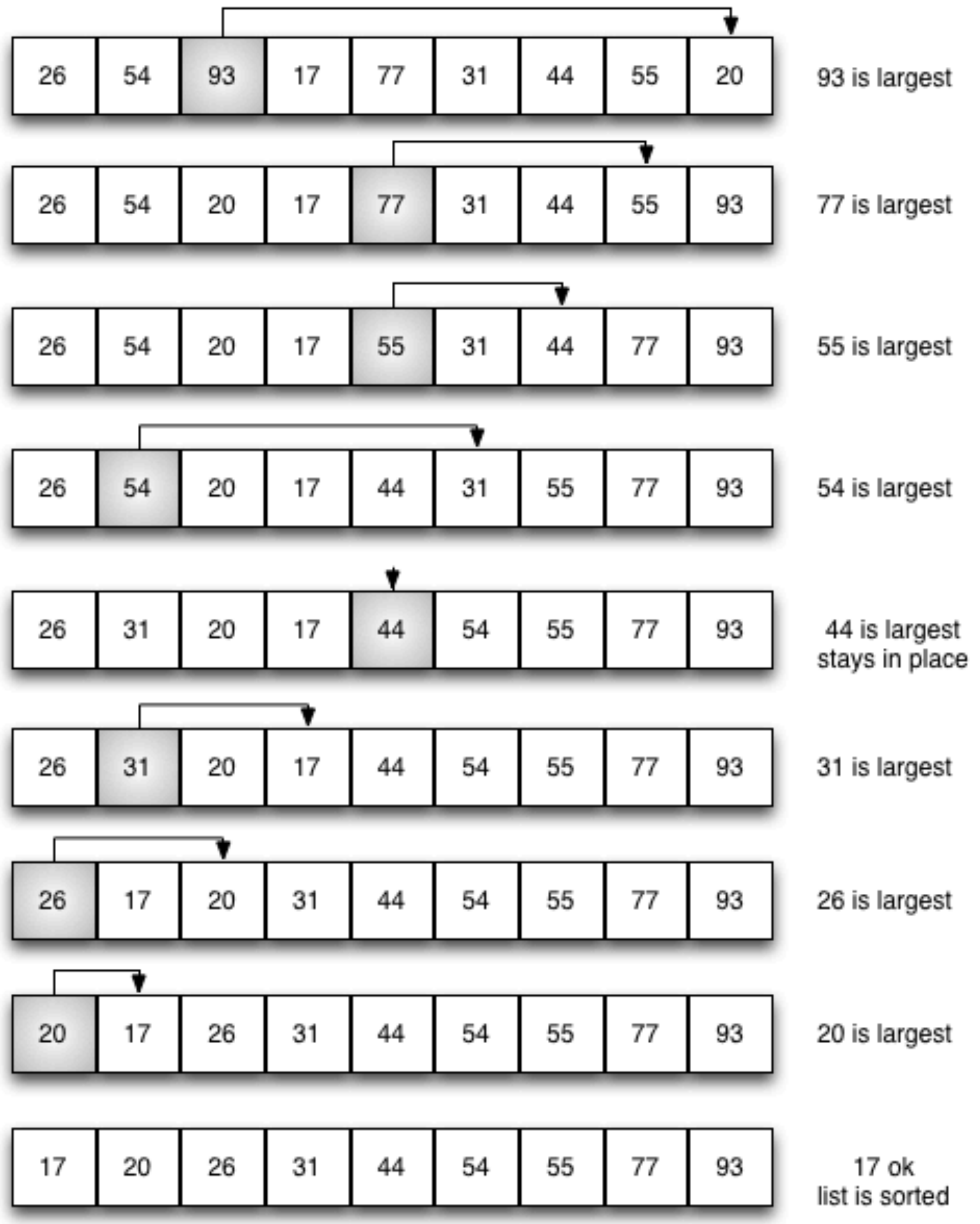


Q1: What is the number of comparison/switches on a sorted array?

Q2: How could the implementation be improved in this case?

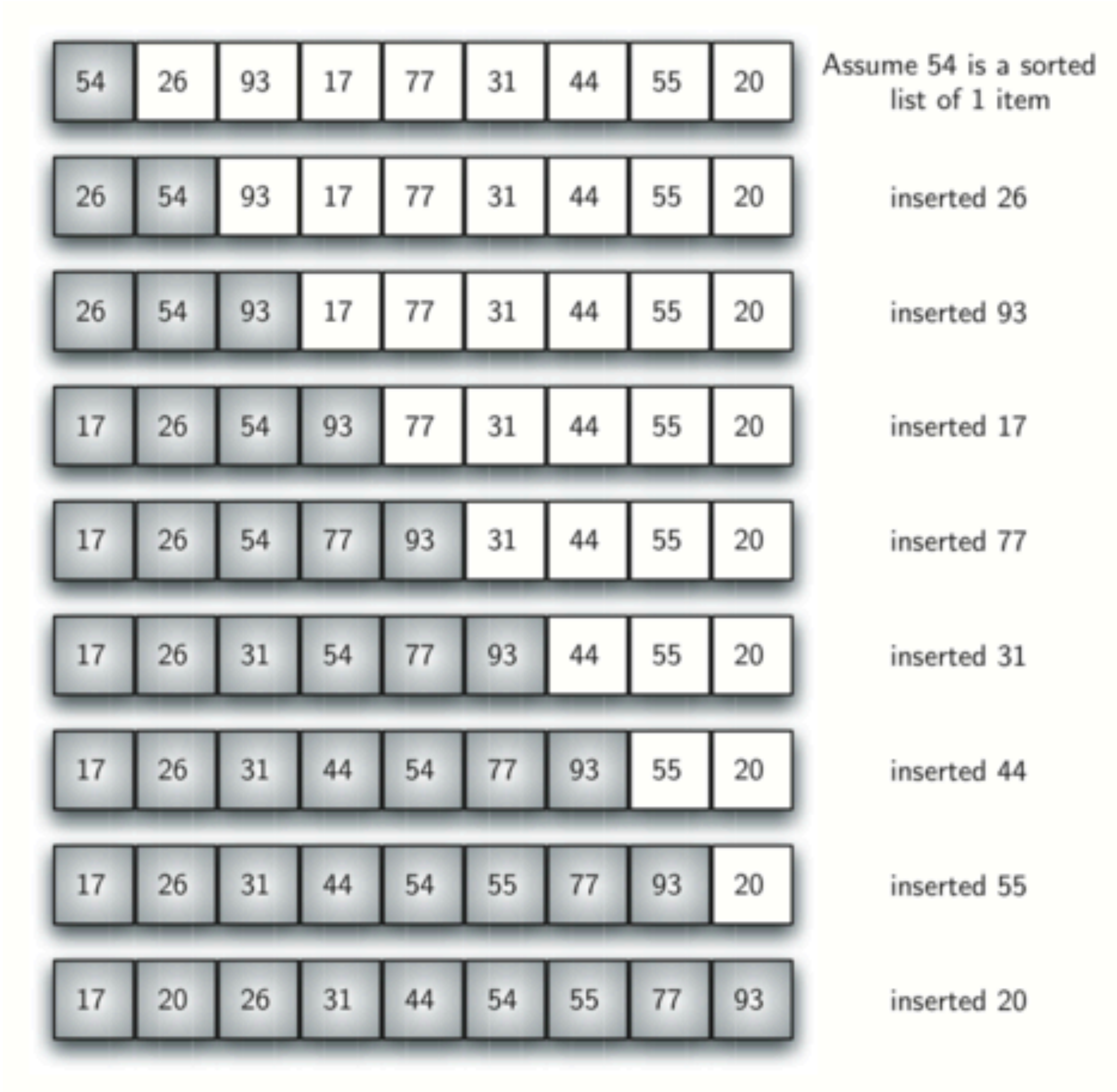
The Selection Sort

- The **selection sort** improves on the bubble sort: by making only one exchange for every pass through the list
- It looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location
- As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place,.
- $O(n^2)$



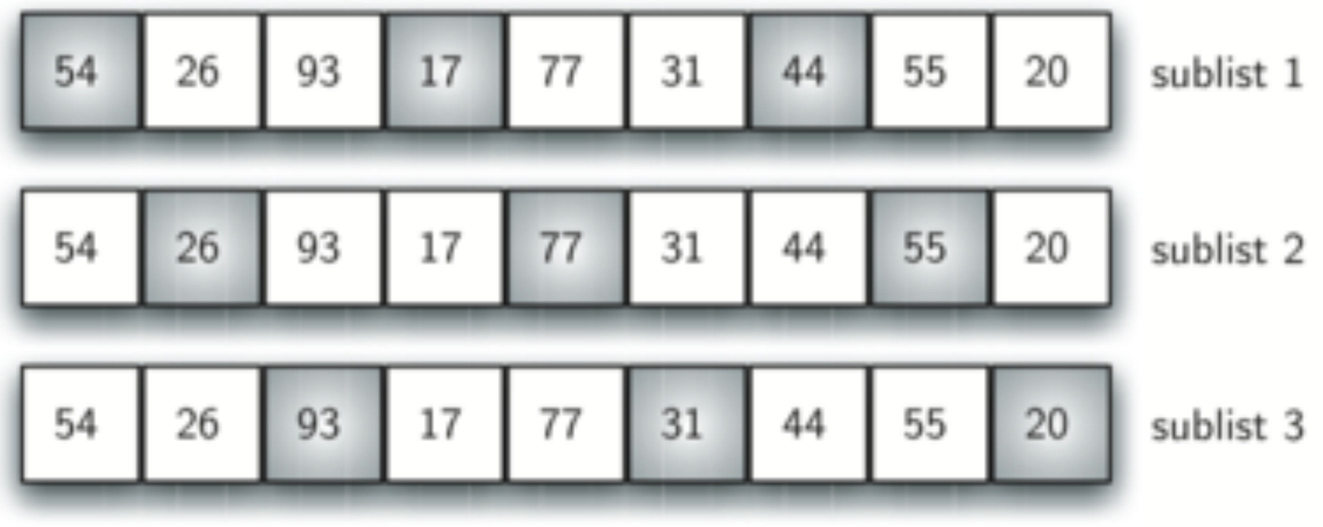
The Insertion Sort

- The **insertion sort**, $O(n^2)$
- It always **maintains a sorted sublist** in the lower positions of the list
- Each new item is then inserted back into the previous sublist such that the sorted sublist is one item larger

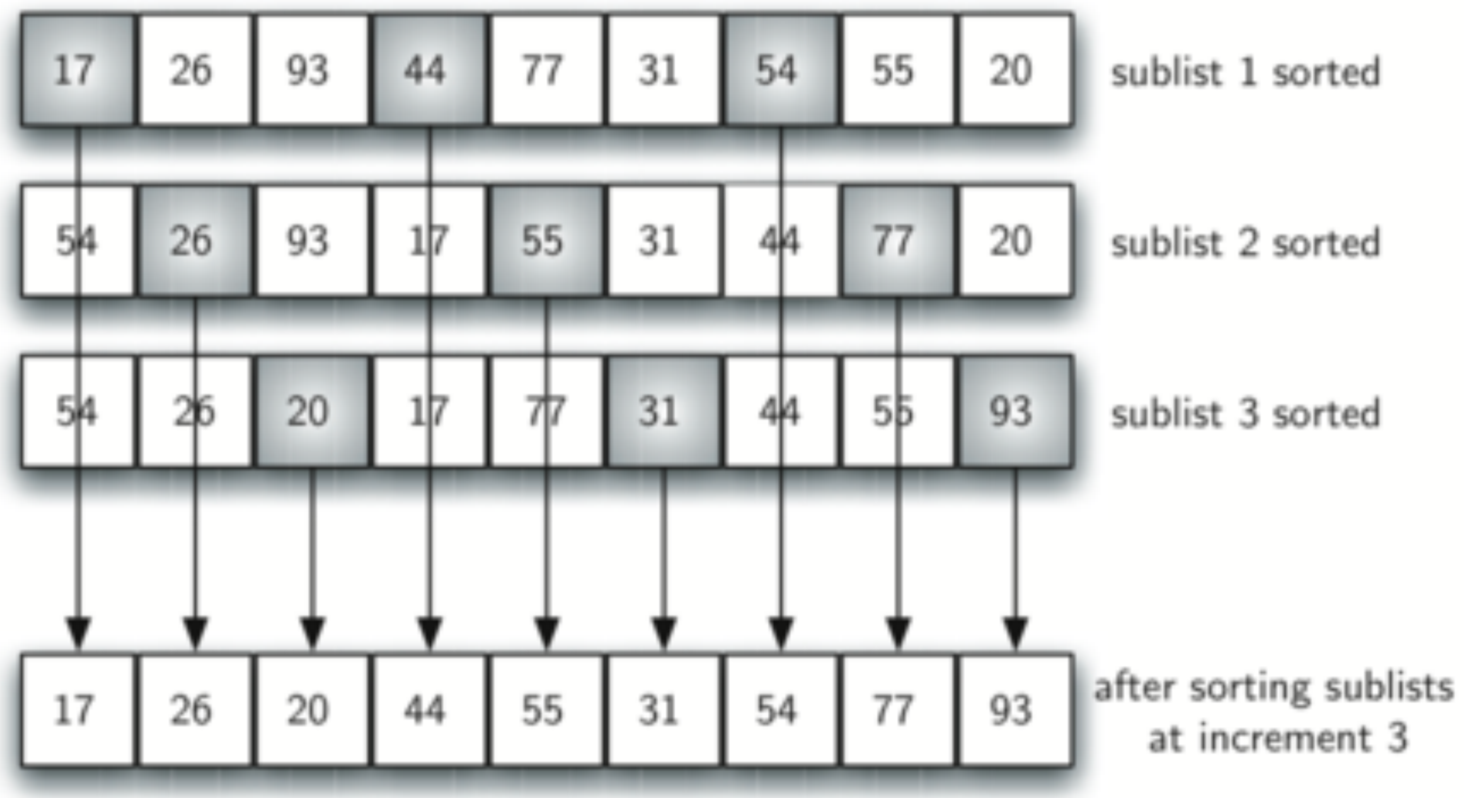


The Shell Sort

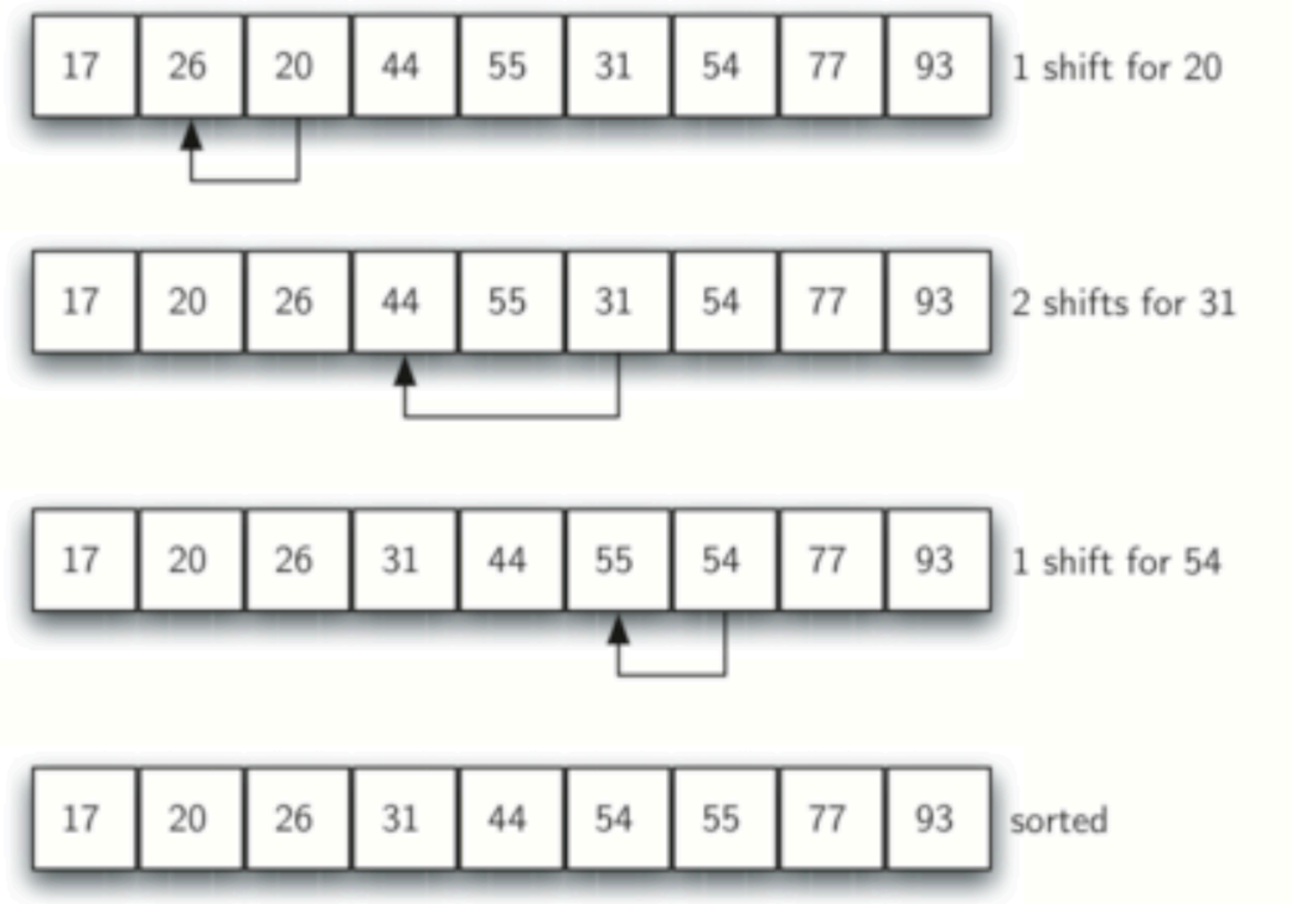
- The **Shell sort**, sometimes called the *diminishing increment sort*, improves on **the insertion sort** by breaking the original list into a number of smaller sublists, each of which is sorted using an insertion sort
- the Shell sort uses an increment, termed the **gap I**, to create a sublist by choosing all items that are **i** items apart
- The final step uses the insertion sort with gap 1
- O(n) to O(n^2)



A Shell Sort with Increments of Three



A Shell Sort after Sorting Each Sublist

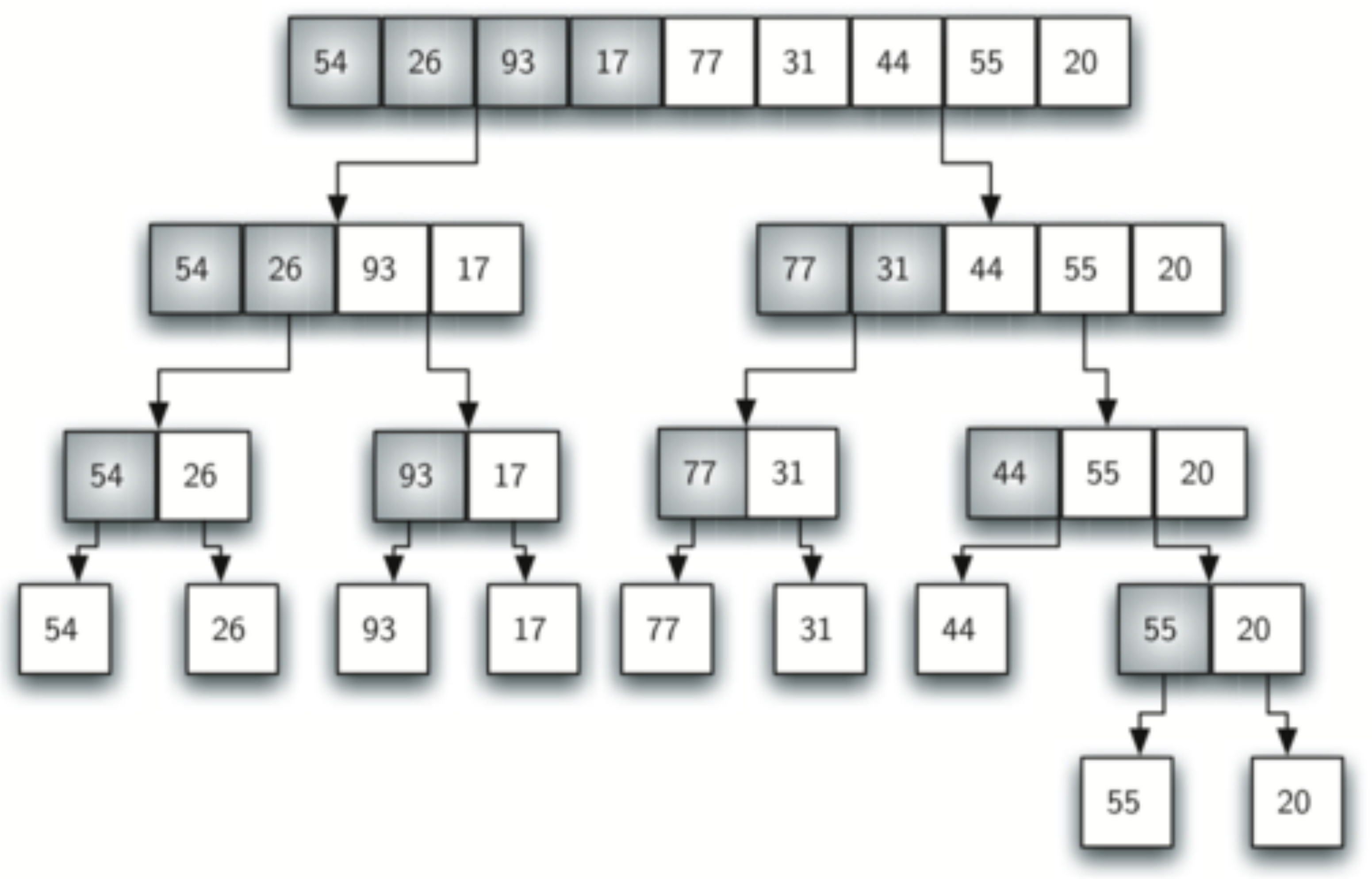


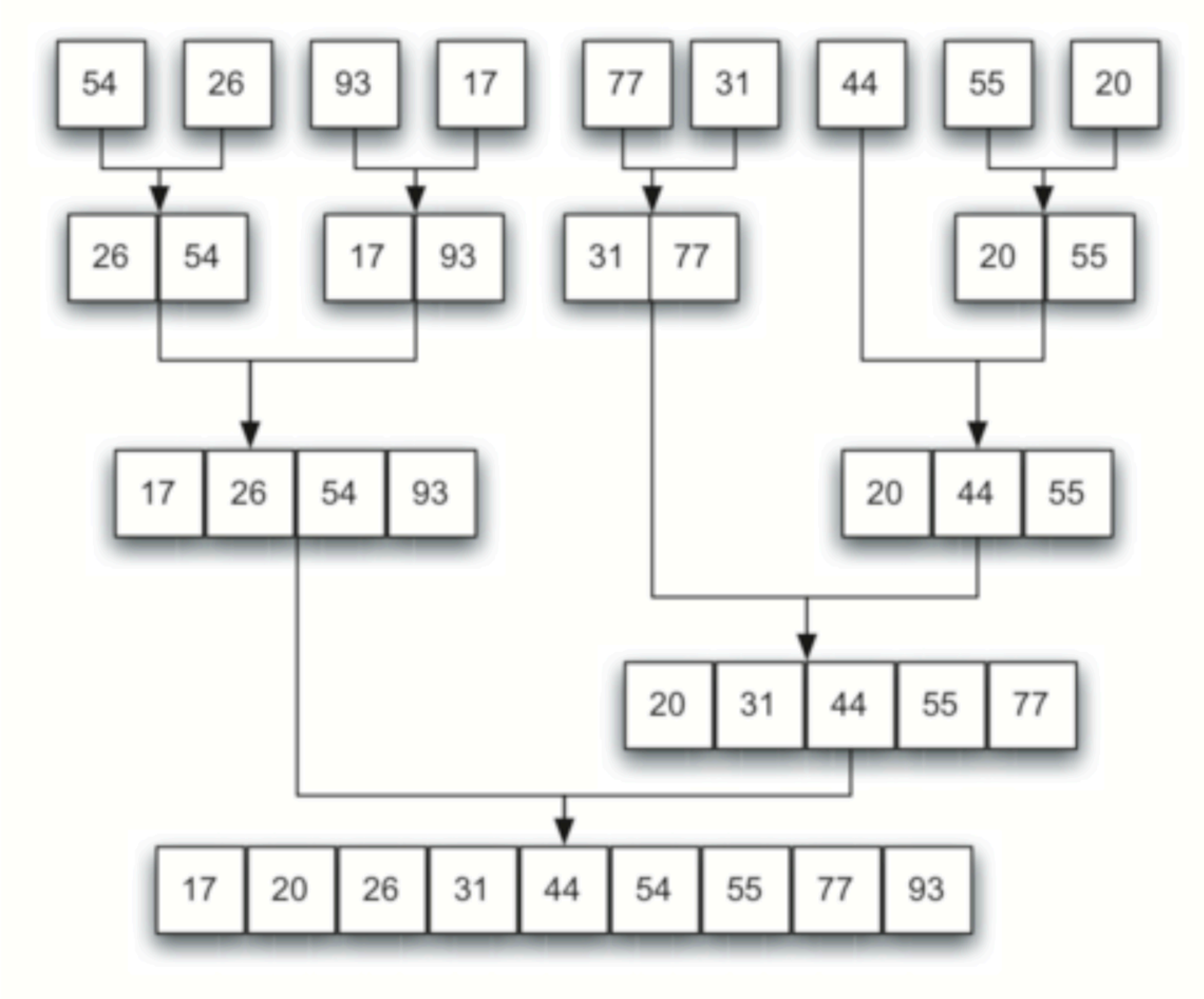
A Final Insertion Sort with Increment of 1

The Merge Sort
O(n*log n)

- **divide and conquer strategy** to improve the performance of sorting algorithms
- a recursive algorithm that continually splits a list in half:
  - the base case: If the list is empty or has one item, it is sorted by definition
  - If the list has more than one item, we split the list and recursively invoke a merge sort on both halves.
- Once the two halves are sorted, the fundamental operation —> **merge**
- Merging is the process of taking two smaller sorted lists and combining them together into a single sorted new list

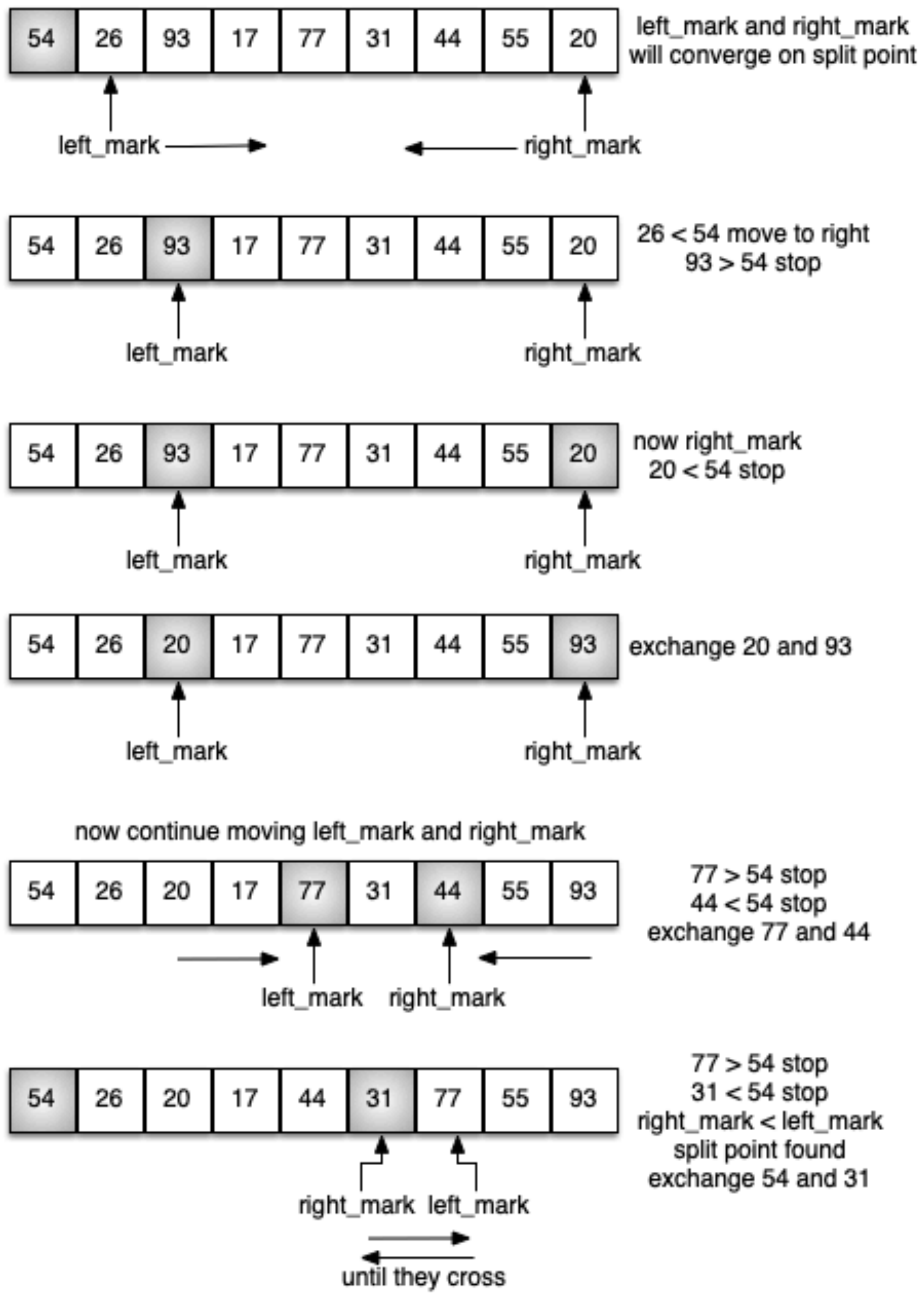
Splitting the List in a Merge Sort





Merging the Lists in a Merge Sort

The Quicksort



Finding the Split Point for 54

- A quicksort first selects a value, which is called the **pivot value e.g.** the first item in the list

- The role of the pivot value is to assist with splitting the list

- The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the **quicksort**.

- O(n*log n)