Principles of Computer Science II

Data Structure

Marco Zecchini

Sapienza University of Rome

Lecture 6

Counting Frequencies Problem

Problem: Given a list of numbers, count how many times each number appears.

Example:

$$[2, 3, 2, 5, 3, 2, 7, 5, 3] \longrightarrow \begin{cases} 2 : 3 \\ 3 : 3 \\ 5 : 2 \\ 7 : 1 \end{cases}$$

We will solve this problem using two different data structures:

- A **list** of pairs (number, count)
- A **dictionary**

Solution 1: Using a List

Idea: Maintain a list of pairs `[number, count]`. For each number in the input list, search if it already exists; if yes, increase its count, otherwise append it.

```
counts = []
for n in numbers:
    found = False
    for pair in counts:
        if pair[0] == n:
            pair[1] += 1
            found = True
            break
    if not found:
        counts.append([n, 1])
```

Solution 2: Using a Dictionary

Idea: Use a dictionary that directly associates each number to its count (remember: in a dictionary we access the value associated with a key in constant time).

```
counts = {}
for n in numbers:
    if n in counts:
        counts[n] += 1
    else:
        counts[n] = 1
```

Question

Which is better of the two in terms of time complexity? And why?

Summary and Discussion

Same problem, different data structures:

Approach	Data Structure	Time Complexity
Naive counting	List of pairs	$O(n^2)$
Efficient counting	Dictionary	$O(n)$

Key takeaway: Choosing the right **data structure** can drastically improve algorithm performance, even when solving the *same* problem.

Outline of the lecture

- Until now, most of the times, we have always worked with primitive data types (integer, float, string, char)...

Outline of the lecture

- Until now, most of the times, we have always worked with primitive data types (integer, float, string, char)...
- ...and combined them into two simple data structure: which are these?

Outline of the lecture

- Until now, most of the times, we have always worked with primitive data types (integer, float, string, char)...
- ...and combined them into two simple data structure: which are these?
- List and Dictionaries

Outline of the lecture

- Until now, most of the times, we have always worked with primitive data types (integer, float, string, char)...
- ...and combined them into two simple data structure: which are these?
- List and Dictionaries
- In the rest of the lecture (and partially of the course), we will see more complex data structure

First: what is a Class in Python?

The Idea

A **class** is a **template** for creating objects. It defines the *properties* (data) and *behaviours* (functions) that those objects will have.

Quick Intuition

Think of a class as a **recipe**. An **object** is a dish prepared from that recipe.

Mini example

```
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        print("Woof!")

my_dog = Dog("Fido")
my_dog.bark()
```

Linked List

Idea: A sequence of nodes, each containing a value and a reference to the next node.

```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None
```

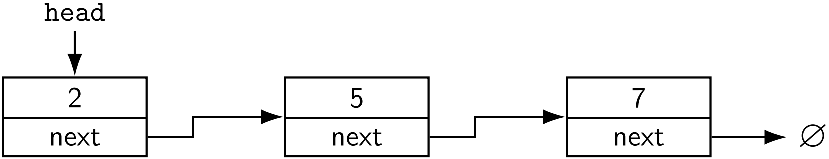
Advantages:

- Efficient insertions/deletions at both ends
- Dynamic memory usage

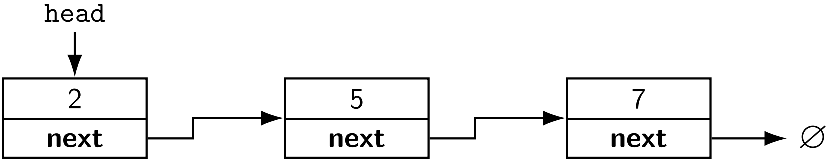
Drawbacks:

- Random access not possible ($O(n)$)
- Extra memory for pointers

Singly Linked List — Structure

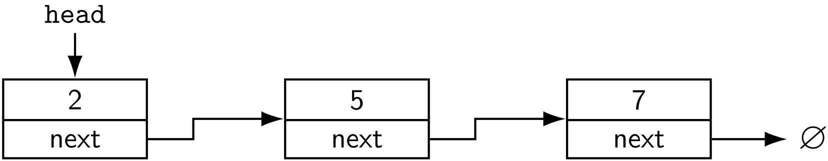


Singly Linked List — Structure



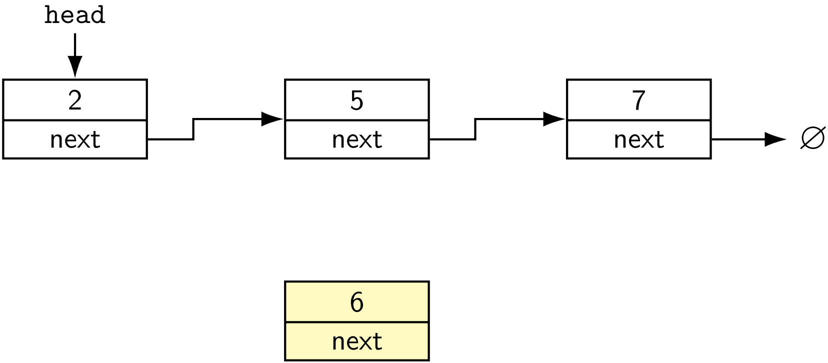
Each node stores **data** and a pointer to **next**

Linked List — Insertion (between two nodes)



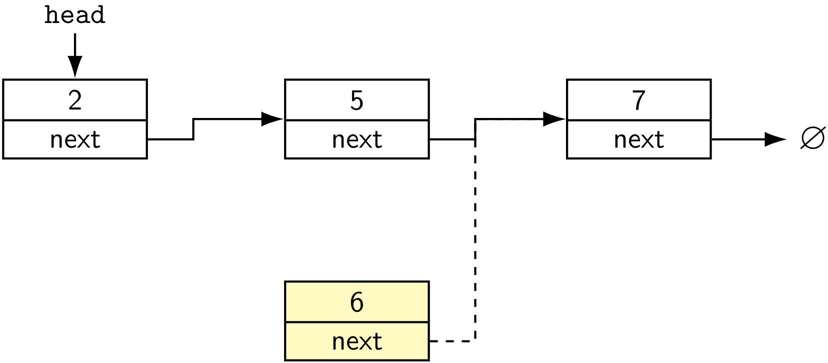
Goal: insert **6** after node **5**

Linked List — Insertion (between two nodes)



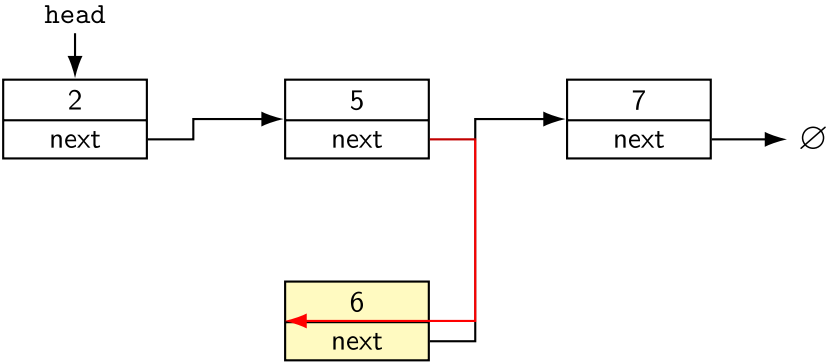
1) Allocate new node **X=6**

Linked List — Insertion (between two nodes)



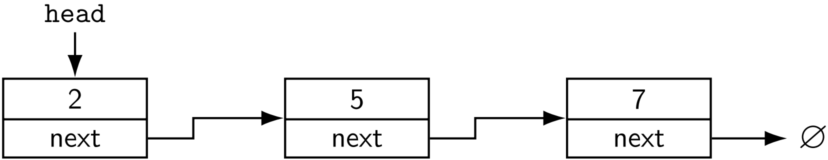
2) Set `X.next = 7`

Linked List — Insertion (between two nodes)



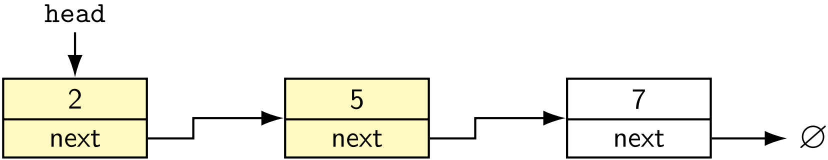
3) Set `5.next = X` → insertion complete

Linked List — Deletion (bypass the target node)



Goal: delete node **5** (have pointer to **prev**=2)

Linked List — Deletion (bypass the target node)

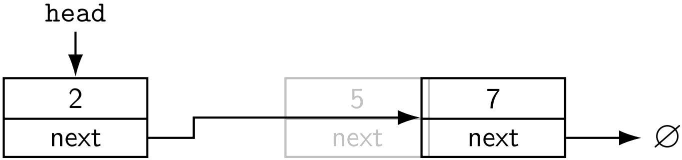


1) Identify **prev** (2) and **target** (5)

Linked List — Deletion (bypass the target node)

2) Set `prev.next = target.next`

Linked List — Deletion (bypass the target node)



3) (Optional) deallocate **target**

Do we need Linked List in Python?

Question

In a framework like Python, do we actually need them?

Do we need Linked List in Python?

Question

In a framework like Python, do we actually need them?

No, we actually don't! We already have Lists that have the same benefits: we essentially saw how Lists are implemented behind the hood (or a way to do that)

Stack and Queue

Two fundamental ways to organize and manage elements:

Stack — LIFO (Last In, First Out)

- The last element added is the first one to be removed.
- Think of a stack of plates: you remove the top one first.
- Useful when an algorithm needs to "go back": recursion, undo mechanisms, backtracking.

Queue — FIFO (First In, First Out)

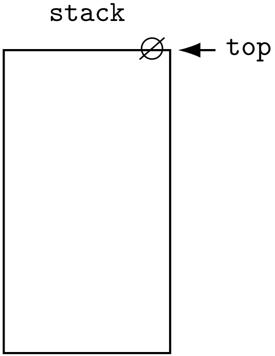
- The first element added is the first one to be removed.
- Like a line at the supermarket: first come, first served.
- Used when order must be respected: task scheduling, BFS, event handling.

Stack and Queue (cont.)

Why are they important?

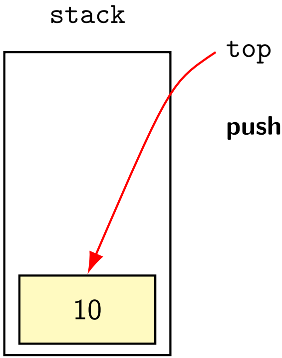
- They impose a simple but powerful order on how elements are processed.
- They help control the flow of algorithms clearly and predictably.
- They mirror natural behaviors (stacks, lines) $\rightarrow$ easy to understand, essential in computing.

Stack (LIFO) — Structure and Push



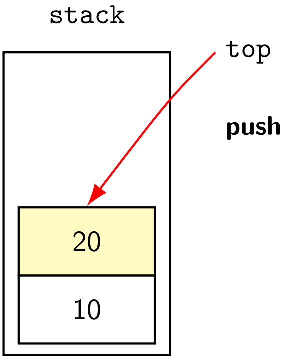
Empty stack: `top = null`

Stack (LIFO) — Structure and Push



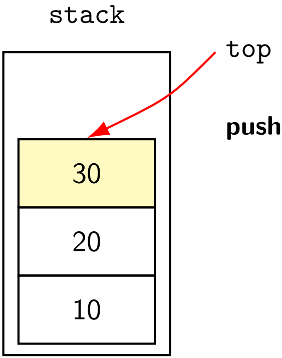
push(10): place at bottom, `top` → 10

Stack (LIFO) — Structure and Push



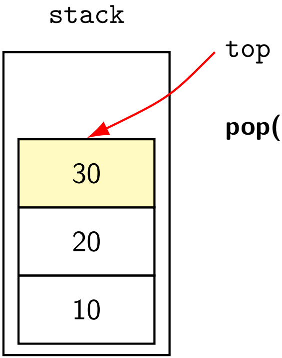
push(20): new item goes on top

Stack (LIFO) — Structure and Push



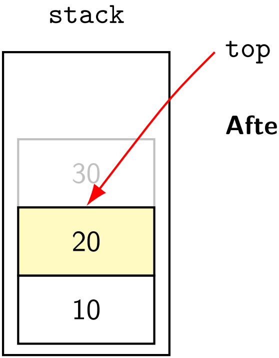
push(30): LIFO grows upwards

Stack (LIFO) — Pop (Deletion)



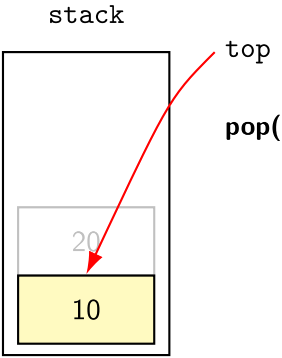
pop() removes the **top** element (30)

Stack (LIFO) — Pop (Deletion)



After pop: `top` $\rightarrow$ 20

Stack (LIFO) — Pop (Deletion)



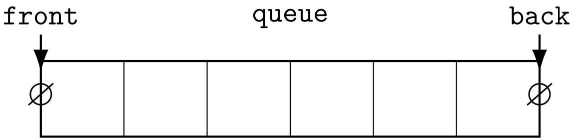
pop() again: top → 10

Stack in Python

Minimal Stack implementation

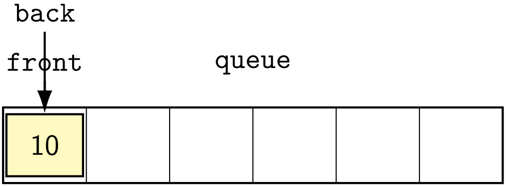
```
class Stack:
  def __init__(self):
    self.items = []
  def is_empty(self):
    return len(self.items) == 0
  def push(self, item):
    self.items.append(item)
  def pop(self):
    if not self.is_empty():
      return self.items.pop()
    raise IndexError("Stack is empty.")
  def peek(self):
    if not self.is_empty():
      return self.items[-1]
    raise IndexError("Stack is empty.")

my_stack = Stack()
my_stack.push(10)
my_stack.push(20)
my_stack.push(30)
print(my_stack.pop())  # 30
print(my_stack.peek()) # 20
```

Queue (FIFO) — Structure and Enqueue



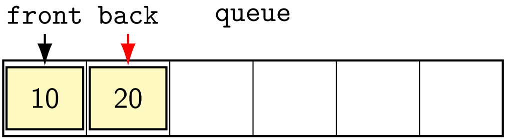
Empty queue: `front = back = null`

Queue (FIFO) — Structure and Enqueue



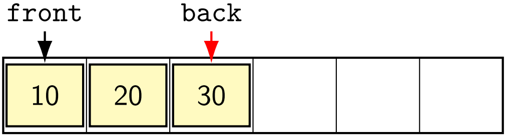
enqueue(10): first element sets both `front` and `back`

Queue (FIFO) — Structure and Enqueue



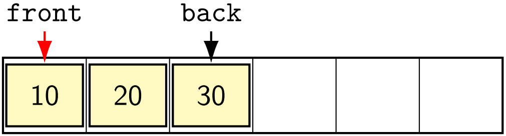
enqueue(20): insert at **back**

Queue (FIFO) — Structure and Enqueue



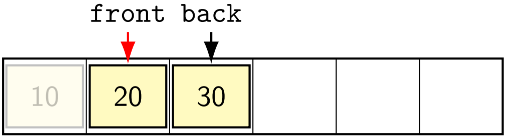
enqueue(30): FIFO grows to the right

Queue (FIFO) — Dequeue (Deletion)



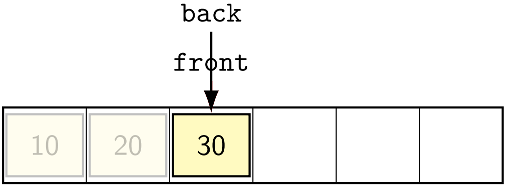
dequeue() removes the **front** element (10)

Queue (FIFO) — Dequeue (Deletion)



After dequeue: `front` → 20

Queue (FIFO) — Dequeue (Deletion)



dequeue() again: `front` → 30

Queue in Python

Minimal Queue implementation

```python
from collections import deque
class Queue:
  def __init__(self):
    self.items = deque()

  def is_empty(self):
    return len(self.items) == 0
  def enqueue(self, item):
    self.items.append(item)
  def dequeue(self):
    if not self.is_empty():
      return self.items.popleft()
    raise IndexError("Queue is empty.")
  def peek(self):
    if not self.is_empty():
      return self.items[0]
    raise IndexError("Queue is empty.")

# Using the queue
my_queue = Queue()
my_queue.enqueue("A")
my_queue.enqueue("B")
my_queue.enqueue("C")
print(my_queue.dequeue()) # Output: A
print(my_queue.peek()) # Output: B
```

Tree Data Structures

What is a Tree?

A **tree** is a hierarchical data structure composed of **nodes** connected by **edges**. It represents relationships like those found in family trees, organization charts, or file systems.

Tree Data Structures

What is a Tree?

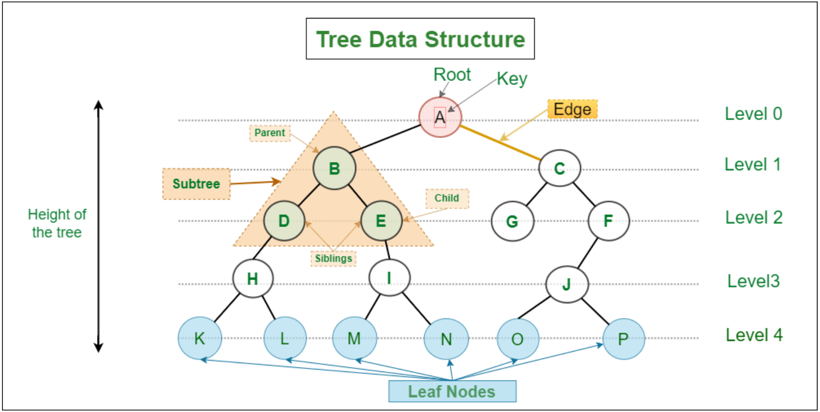
A **tree** is a hierarchical data structure composed of **nodes** connected by **edges**. It represents relationships like those found in family trees, organization charts, or file systems.

Why Trees?

They allow efficient representation of hierarchical relationships and form the basis of:

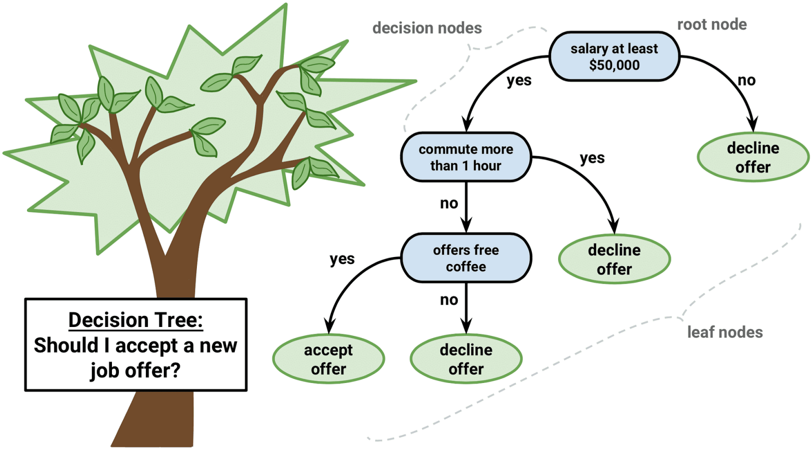
- Search and decision structures (e.g. Binary Search Trees)
- Hierarchical data models (e.g. XML, file systems)
- Optimization algorithms and parsing

Tree (cont.)

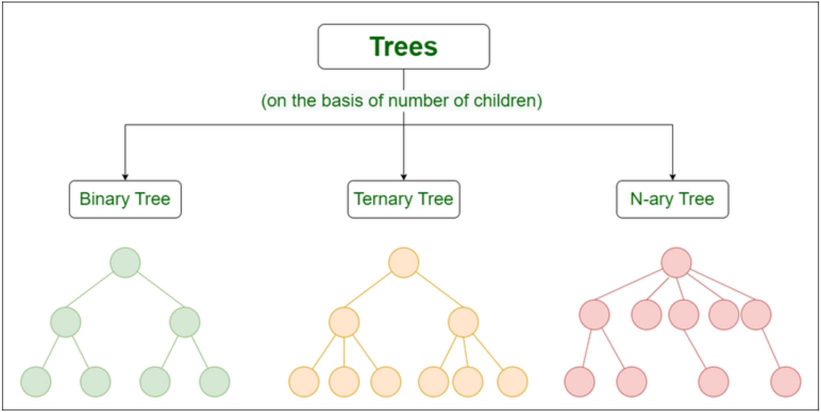


Tree Data Structure
Root
Key
Edge
A
B
C
D
E
G
F
H
I
J
K
L
M
N
O
P
Parent
Subtree
Child
Siblings
Height of the tree
Level 0
Level 1
Level 2
Level3
Level 4
Leaf Nodes

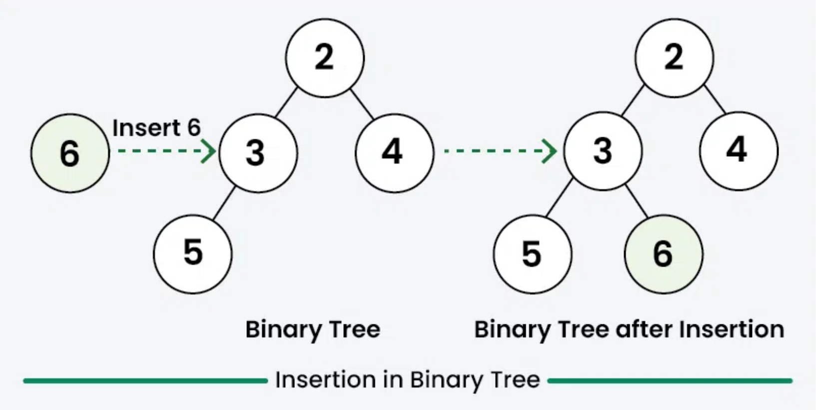
Tree application



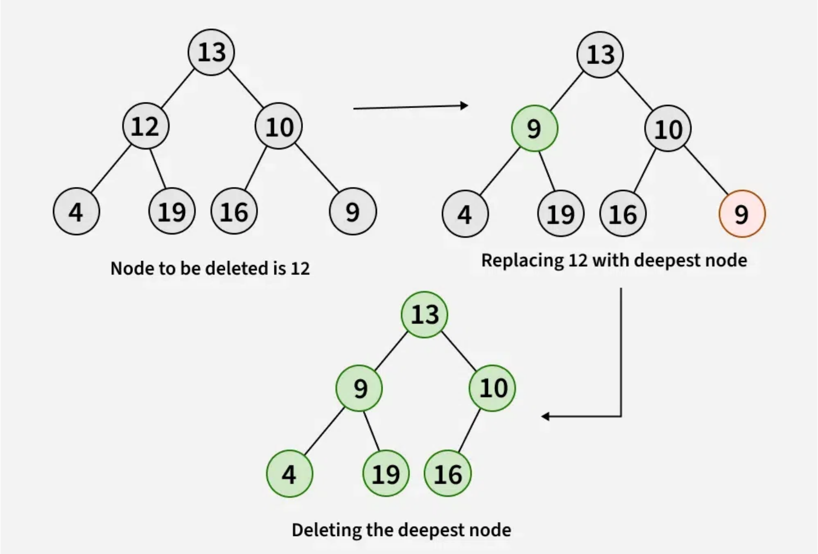
N-ary Tree, Ternary Tree, Binary Tree



Insertion in a binary tree



Deletion in a binary tree



Questions

Are the nodes sorted?

Questions

Are the nodes sorted? How can I look for a specific key in a tree?

Binary Search Trees (BST)

What is a Binary Search Tree?

A **Binary Search Tree (BST)** is a special type of **binary tree** where each node satisfies:

$$\text{left subtree values} < \text{node value} < \text{right subtree values}$$

- Each node has at most **two children**: a **left child** and a **right child**.
- The structure maintains a **sorted order**, enabling efficient search.
- Common operations: **insertion**, **search**, and **deletion**.

Binary Search Trees (BST)

What is a Binary Search Tree?

A **Binary Search Tree (BST)** is a special type of **binary tree** where each node satisfies:

$$\text{left subtree values} < \text{node value} < \text{right subtree values}$$

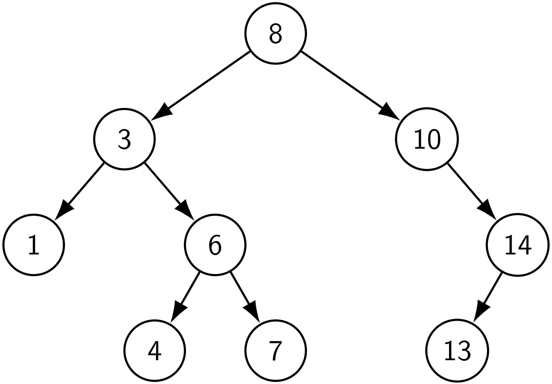
- Each node has at most **two children**: a **left child** and a **right child**.
- The structure maintains a **sorted order**, enabling efficient search.
- Common operations: **insertion**, **search**, and **deletion**.

Why are BSTs useful?

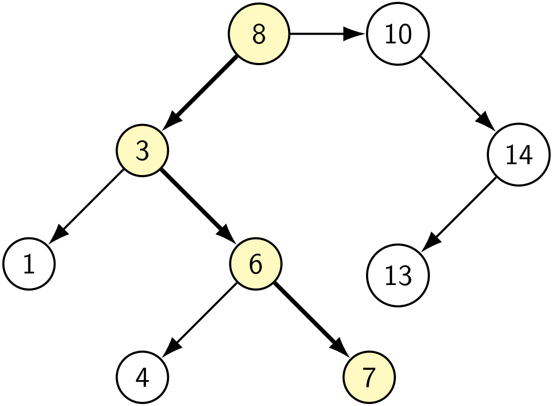
They allow:

- Searching in $O(\log n)$ time (on average)
- Maintaining dynamic, sorted data
- Forming the basis for balanced trees

BST — Example

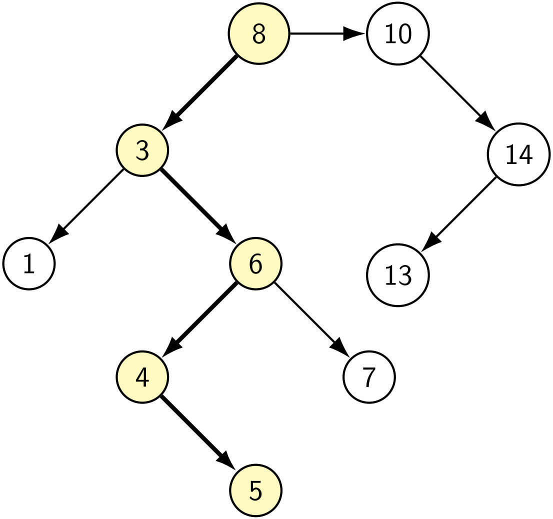


BST — Search for 7



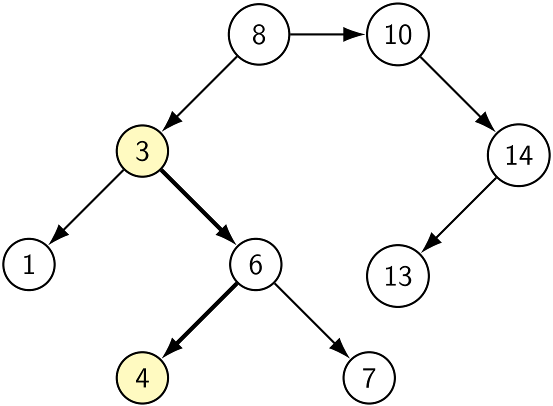
Path compared: $8 \to 3 \to 6 \to 7$ (found).

BST — Insertion of 5



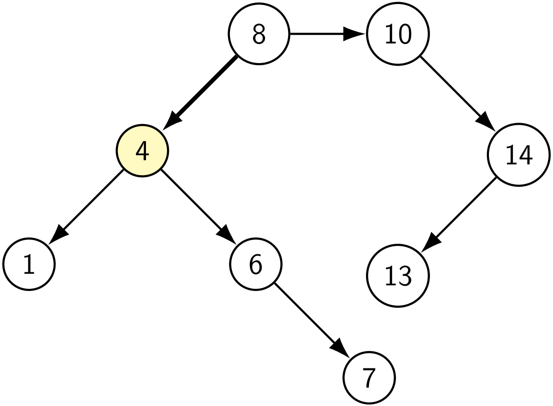
Insert path: $8 \to 3 \to 6 \to 4 \to$ _(right) $\Rightarrow 5$.

BST — Deletion of 3 (two children)



Step 1: Node 3 has two children. Inorder successor is 4.

BST — Deletion of 3 (two children)



Step 2: Copy 4 into node, then delete the original 4 (simple case).

BST in Python

BST in Python

```
class Node:
  def __init__(self, value):
    self.value = value
    self.left = None
    self.right = None

class BinarySearchTree:
  def __init__(self):
    self.root = None

  def insert(self, value):
    if not self.root: self.root = Node(value)
    else: self._insert_recursive(self.root, value)

  def _insert_recursive(self, node, value):
    if value < node.value:
      if not node.left: node.left = Node(value)
      else: self._insert_recursive(node.left, value)
    else:
      if not node.right: node.right = Node(value)
      else: self._insert_recursive(node.right, value)
```

BST Insertions Can Produce an Unbalanced Tree

Claim. A Binary Search Tree (BST) built using naive insertions can become *highly unbalanced*.

Idea of the proof. Consider inserting the following sequence of keys:

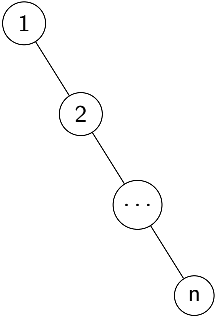
$$1, 2, 3, \ldots, n$$

- Insert 1: becomes the root.
- Insert 2: since $2 > 1$, it becomes the right child of 1.
- Insert 3: since $3 > 1$ and $3 > 2$, it goes even further right.
- The same happens for all successive elements.

Key observation: When keys arrive in sorted order, at every step the new node is placed as the *right child of the deepest node*, creating a chain.

Example of Degenerate BST

Resulting structure:



Height of the resulting tree:

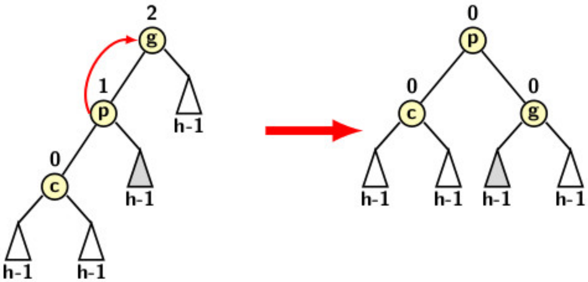
$$h = n - 1$$

Search time in this BST:

$$O(n) \quad \text{(same as a linked list)}$$

Conclusion. A naive BST provides no guarantee of balance: the insertion order alone can force the tree to degenerate.
Balanced trees (AVL, Red-Black, etc.) avoid this problem by performing **rotations**.

Rotation



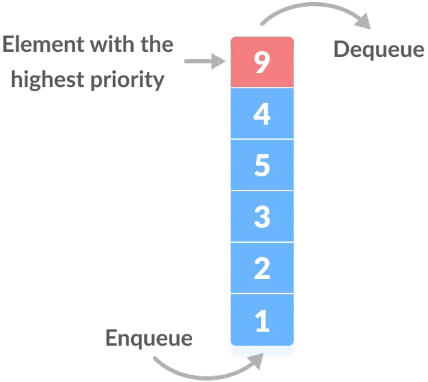
There are different type of **Balanced BST** that changes according to the way they implement this rotation in insertion and/or deletion (AVL, Red-Black, etc.).

Introduction to Heaps

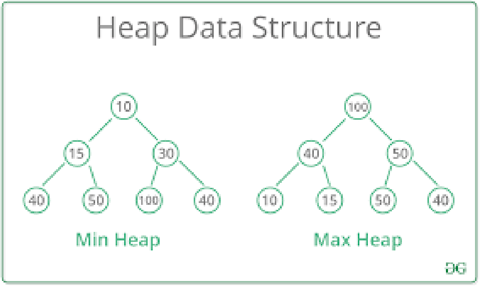
- A **heap** is a special kind of binary tree used to store elements with a quick access to the **minimum** or **maximum** value.
- It is a **complete binary tree**: all levels are full except possibly the last, which is filled left to right.
- It satisfies the **heap property**:
  - **Min-heap**: every node is $\geq$ its parent (root contains the minimum).
  - **Max-heap**: every node is $\leq$ its parent (root contains the maximum).
- Efficient operations:
  - `insert`: $O(\log n)$
  - `extract-min`/`extract-max`: $O(\log n)$
  - `peek`: $O(1)$
- Used in priority queues and in algorithms like **Heapsort**.

Heap application: Priority Queues

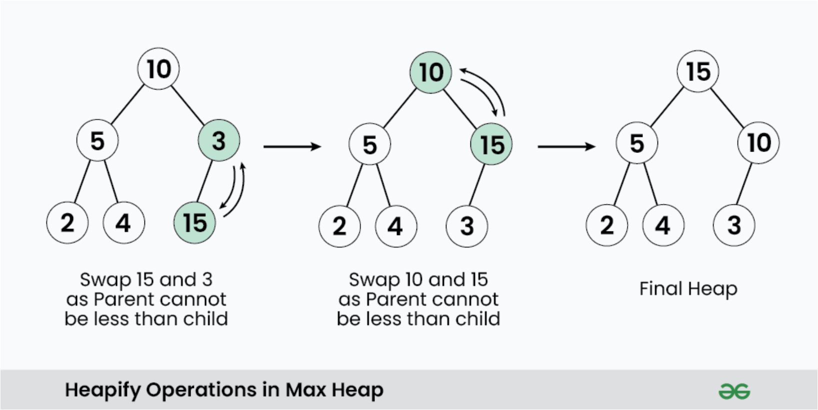
We are in a hospital and we have to visit patients according to the urgency of their situations.



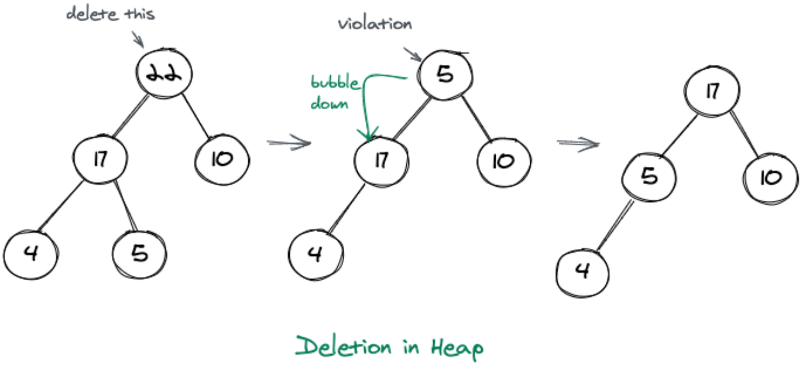
Heap representation



Insertion in Heaps



Deletion in Heaps



Heap in Python

Heap in Python

```
import heapq

# Creating a heap
my_heap = [3, 1, 4, 1, 5, 9, 2, 6, 5]
heapq.heapify(my_heap)

print(my_heap) # Output: [1, 1, 2, 5, 3, 9, 4, 6, 5]

# Inserting into a heap
heapq.heappush(my_heap, 0)
print(my_heap) # Output: [0, 1, 1, 5, 2, 9, 4, 6, 5, 3]

# Extracting the smallest element
min_element = heapq.heappop(my_heap)
print(min_element) # Output: 0
```

What is next?

- Graph (another data structure)
- Points in more than one dimension (points in 2 dimensions, in 3 dimensions)