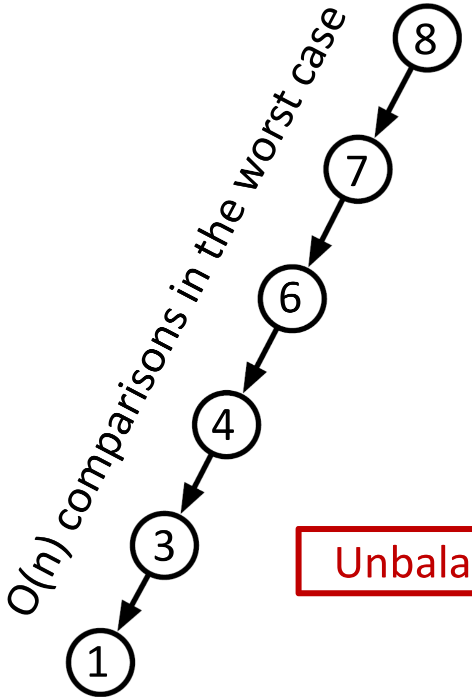
Review:
Algorithms & Data Structures

OUTLINE

1. Balanced Binary Search Trees
2. Heap
3. Hash Table
4. Stack & Queue
5. Sorting Algorithms

Why Balanced Binary Search Tree?



Unbalanced

Binary Search Tree

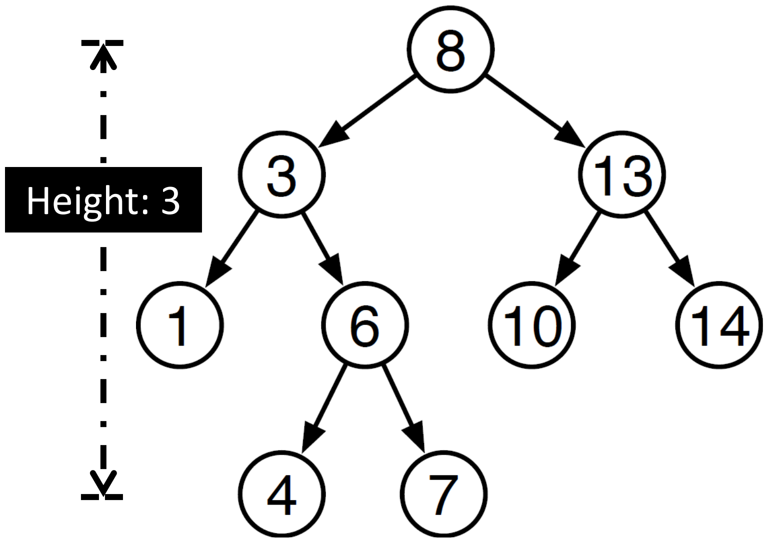
Balanced Binary Search Tree

To achieve **O($\log n$)** search efficiency, we need a **Balanced BST**

Balanced Binary Search Tree

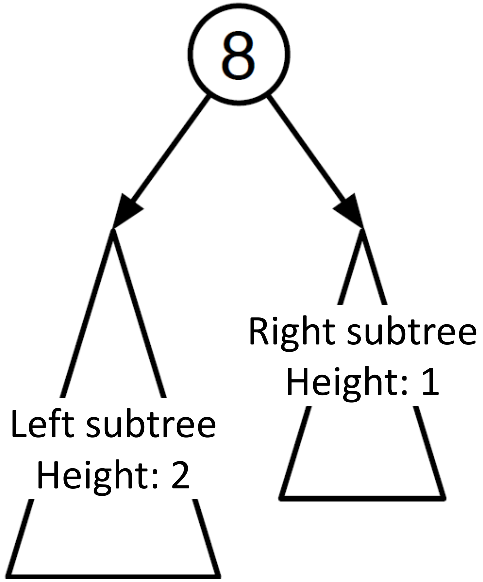
A **Binary Search Tree** in which each node is **Balanced**.

Balanced: The left and right subtrees differ in **height** by no more than **1**.



A balanced binary tree.

Height: length of the longest root-leaf path



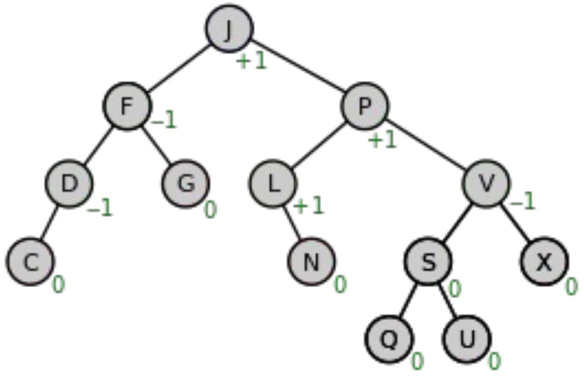
Balanced Binary Search Trees

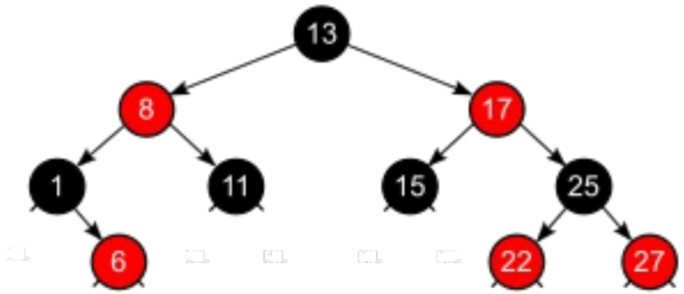
Motivation:

- Efficiently store and retrieve data while maintaining a balanced structure.
- Balanced trees ensure optimal performance for operations like insertion, deletion, and searching.

Balanced Binary Search Trees:

- AVL trees
- Red-black trees





Balanced Binary Search Trees

Operation: Both AVL trees and red-black trees are not included in the standard library of Python.

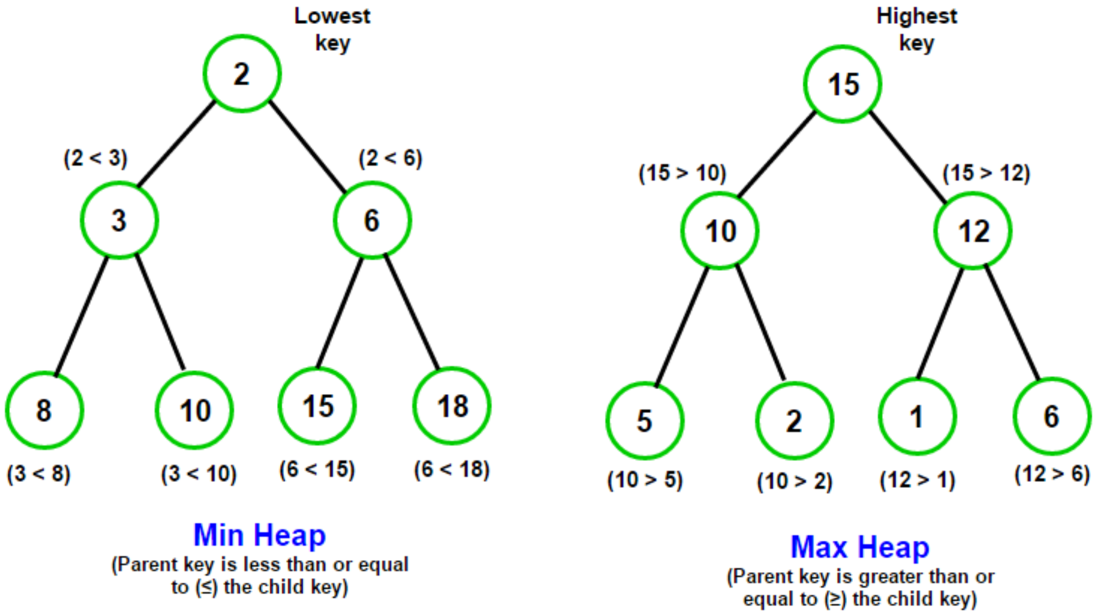
	AVL Tree	Red-Black Tree
Balancing Criteria	Height-Balanced (strictly)	Height-Balanced (relaxed)
Balancing Factor	Balance factor (-1, 0, +1)	Color (Red or Black)
Rotations	More rotations due to strict balancing	Fewer rotations due to relaxed balancing
Insertion and Deletion	Slower due to frequent rotations	Faster due to fewer rotations
Lookup/Searching	Slightly faster due to better height balance	Slightly slower due to relaxed height balance
Applications	When frequent searching is expected	When frequent insertion/deletion is expected

Heap (Priority Queue)

Motivation:

- Heaps provide efficient operations for insertion, deletion, and retrieval of the highest (or lowest) priority element.

Operation: See heap.py



Time Complexity	Average/Worst-case
Insertion	$O(\log n)$
Deletion	$O(\log n)$
Peek (max/min)	$O(1)$

Hash Table

Motivation:

- Hash tables provide fast insertion, deletion, and lookup operations.
- They have constant time complexity on average.

Operation: See hash_table.py

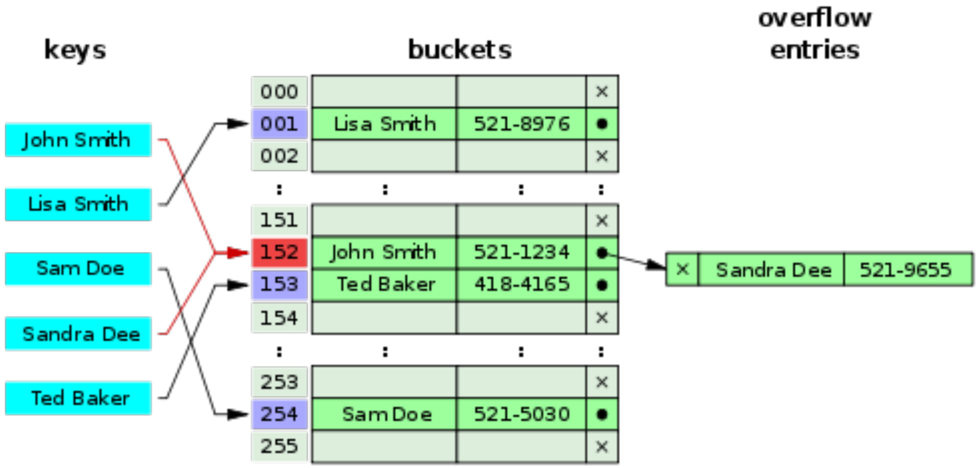
Time Complexity	Average	Worst (with collisions)
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Lookup	O(1)	O(n)

Hash Table

Limitation:

- Space Cost
- Hash Collisions
- Non-constant Time (worst-case)

Time Complexity	Average	Worst (with collisions)
Insertion	O(1)	O(n)
Deletion	O(1)	O(n)
Lookup	O(1)	O(n)



Space vs Efficiency

Stack

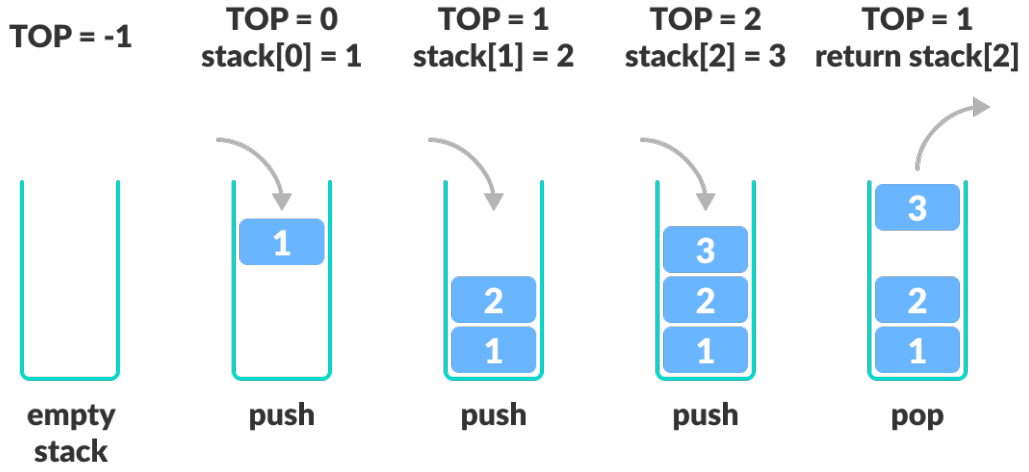
Motivation:

- Last-In-First-Out (LIFO)

Operation: See stack.py

Time Complexity:

- Constant O(1)



Queue

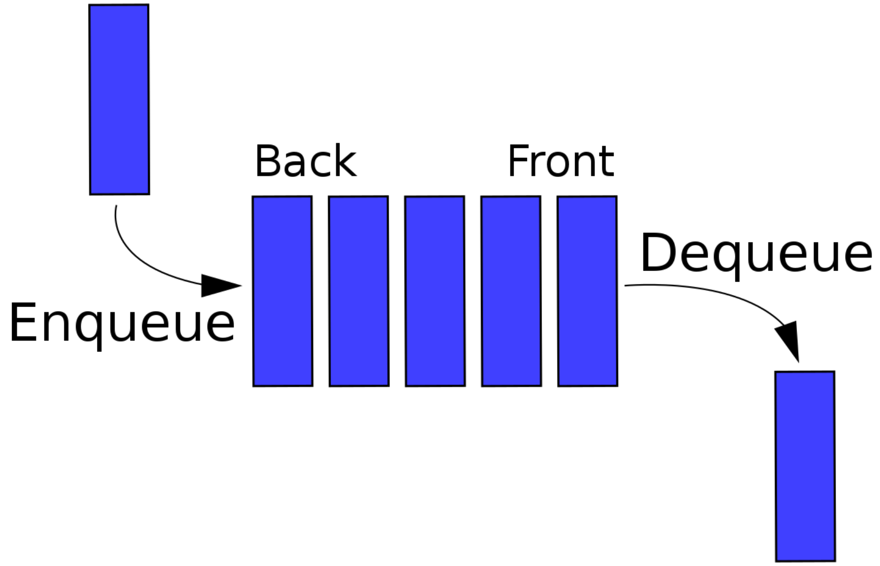
Motivation:

- First-In-First-Out (FIFO)

Operation: See queue.py

Time Complexity:

- Constant O(1)



Sorting Algorithms

Motivation:

- Sorting algorithms allow us to arrange elements in a particular order, making it easier to search, analyze, and manipulate data.

Operation: The built-in 'sorted()' function and the 'list.sort()' method uses an implementation of the Timsort algorithm.

Complexity	Best (Time)	Average (Time)	Worst (Time)	Worst (Space)
Bubble Sort	O(n)	O(n^2)	O(n^2)	O(1)
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n)	O(n^2)	O(n^2)	O(1)
Merge Sort	O($n \log n$)	O($n \log n$)	O($n \log n$)	O(n)
Quick Sort	O($n \log n$)	O($n \log n$)	O(n^2)	O($\log n$)
Timsort	O(n)	O($n \log n$)	O($n \log n$)	O(n)

THANK YOU