7 Algorithm II

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

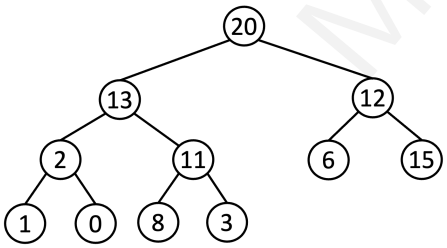
- Understand how min-heaps and max-heaps enable efficient dynamic data management.
- Understand binary search trees and the role of balancing in maintaining efficiency.
- Learn fundamental graph algorithms for shortest paths and minimum spanning trees.

7.1 Max/Min Heap

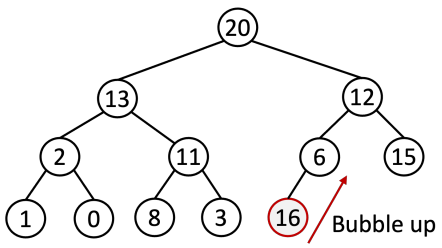
Problem: Given a stream of numbers arriving over time, how can we efficiently maintain the data so that the top-k largest elements can be retrieved at any time, where k is small?

An intuitive approach is to keep all received data in sorted order. Whenever a new element arrives, we insert it into its appropriate position in the sorted sequence. In this way, the top-k elements can be retrieved in $O(k)$ time. However, inserting a new element requires, on average, approximately $n/2$ comparisons if we use linear search, where n is the number of existing elements.

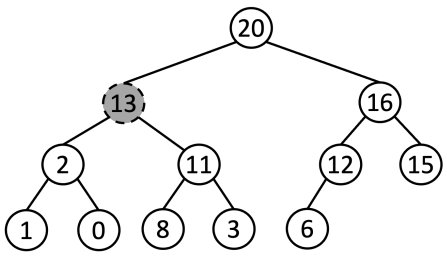
To overcome this inefficiency, we can use a heap. Unlike a sorted sequence, a heap maintains only a partial order among its elements. A heap is a complete binary tree, meaning that all levels are fully filled except possibly the last, which is filled from left to right. In a max-heap, each parent node has a value greater than or equal to those of its children. In a min-heap, each parent node has a value less than or equal to those of its children.



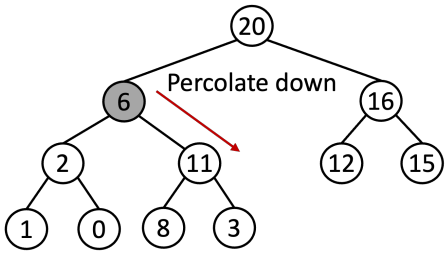
(a) Initial max-heap.



(b) Insert element 16.



(c) Delete element 13.



(d) Replace with the last node and percolate down.

Figure 7.1: Illustration of insertion and deletion operations on a max-heap.

Figure 7.1a illustrates an example of a max-heap. Next, we discuss three key operations on a heap:

- *Insert an element*: Place the new element at the next available position in the complete binary tree, then repeatedly swap it with its parent if necessary to restore the heap property (bubble up). In Figure 7.1b, inserting 16 causes it to bubble up until its parent (20) is greater than it.
- *Delete an element*: Remove the target element and replace it with the last element in the heap. Then restore the heap property by moving the replacement element up or down as needed. In Figure 7.1c, deleting 13 causes the last element (6) to take its place. The replacement element is then repeatedly swapped with its larger child until the heap property is restored (Figure 7.1d).
- *Find the top-k elements*: Repeatedly extract the root (the maximum in a max-heap or the minimum in a min-heap) k times. If the original heap must remain unchanged, a copy or an auxiliary heap can be used.

Implementation In practice, a heap is typically represented using an array or list, with elements stored in *level order* (from left to right at each level). For example, the max-heap in Figure 7.1a can be represented as:

$$[20, 15, 13, 2, 11, 6, 12, 1, 0, 8, 3]$$

Since a heap is a complete binary tree, its nodes can be stored consecutively in an array without any gaps. This allows us to determine the positions of parent and child nodes directly from their indices, eliminating the need for explicit pointers. Using zero-based indexing, the indices of the left and right children of a node at index i are:

$$\text{left} = 2i + 1, \quad \text{right} = 2i + 2$$

Similarly, for any non-root node at index i, the index of its parent is:

$$\text{parent} = \lfloor (i - 1)/2 \rfloor$$

Based on this representation, Code 7.1 presents a Python implementation of a max-heap with two main methods: `push()` and `pop()`. The `push()` method inserts a new element into the heap, while `pop()` removes and returns the maximum element (the root).

```python
@dataclass
class MaxHeap:
    data: List[int] = field(default_factory=list)

    def push(self, val: int):
        self.data.append(val)
        i = len(self.data) - 1
        while i > 0:
            parent = (i - 1) // 2
            if self.data[i] > self.data[parent]:
                self.data[i], self.data[parent] = self.data[parent], self.data[i]
                i = parent
            else:
                break

    def pop(self) -> int:
        if not self.data:
            raise IndexError("pop from empty heap")
        n = len(self.data)
        self.data[0], self.data[-1] = self.data[-1], self.data[0]
        max_val = self.data.pop() # pop self.data[-1]
        i = 0
        while True:
            left, right = 2 * i + 1, 2 * i + 2
            largest = i
            if left < len(self.data) and self.data[left] > self.data[largest]:
                largest = left
            if right < len(self.data) and self.data[right] > self.data[largest]:
                largest = right
            if largest == i:
                break
            self.data[i], self.data[largest] = self.data[largest], self.data[i]
            i = largest
        return max_val
```

Code 7.1: Max heap implemented with Python

7.2 Binary Search Tree

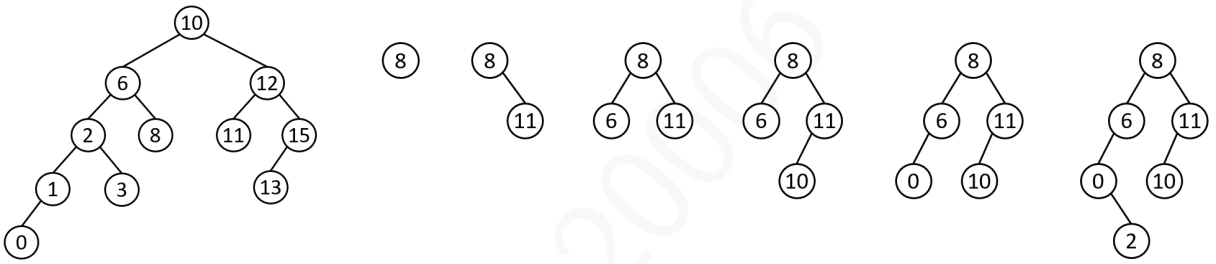
In another application scenario, we may need to search for a particular value in a collection of data, rather than simply retrieving the largest or smallest element. In such cases, a max-heap or min-heap is not suitable. A straightforward approach is to maintain a sorted list. Although binary search allows us to find an element in $O(\log n)$ time, inserting a new element still requires $O(n)$ time due to element shifting.

In this section, we introduce binary search trees (BSTs), which support both insertion and searching in $O(\log n)$ time when the tree is balanced.

7.2.1 Naive BST

A binary search tree organizes data according to the following rules:

- Each node has at most two children: a left child and a right child.
- All nodes in the left subtree of a node have values smaller than the node's value.
- All nodes in the right subtree of a node have values greater than the node's value.



(a) Binary search tree.

(b) BST construction with sequence: 8,11,6,10,0,2.

Figure 7.2: Demonstration of BST.

Figure 7.2a illustrates an example of a binary search tree. Unlike a heap, a BST is not necessarily a complete binary tree. Therefore, the array-based representation used for heaps is generally not suitable for BSTs, as it may leave many unused positions. Instead, a BST is typically implemented using nodes with explicit references to their left and right children.

Code 7.2 presents a Python implementation of a BST. The `Node` class stores a key and references to its left and right children, while the `BinarySearchTree` class maintains a reference to the root node. It provides two main methods: `insert()` and `search()`.

The `insert()` method starts from the root and compares the new key with the current node's key. If the new key is smaller, it moves to the left child; otherwise, if it is larger, it moves to the right child. This process continues until an empty position is found, where the new node is inserted. Duplicate keys are ignored. Similarly, the `search()` method starts from the root and follows the left or right child based on comparisons with the target key. The search terminates when the key is found or an empty position is reached.

The structure of a BST depends on the order in which elements are inserted. Figure 7.2b illustrates the construction of a BST from a given sequence of elements. Different insertion orders may result in different tree structures, even when the same set of elements is inserted.

NF 2006

```python
@dataclass
class Node:
    key: int
    left: Optional[Node] = None
    right: Optional[Node] = None

@dataclass
class BinarySearchTree:
    root: Optional[Node] = None
def insert(self, key):
    new_node = Node(key)
    if self.root is None:
        self.root = new_node
        return
    current = self.root
    while True:
        if key < current.key:
            if current.left is None:
                current.left = new_node
                return
            current = current.left
        elif key > current.key:
            if current.right is None:
                current.right = new_node
                return
            current = current.right
        else:
            return

def search(self, key):
    current = self.root
    while current is not None:
        if key == current.key:
            return True
        elif key < current.key:
            current = current.left
        else:
            current = current.right
    return False
```

Code 7.2: Binary search tree implemented with Python

The efficiency of insertion and searching depends on the height of the tree. Ideally, the tree should remain balanced, with a height of $O(\log n)$, allowing both operations to run in $O(\log n)$ time. However, in the worst case, the tree may degenerate into a linked list, with a height of $O(n)$, causing both operations to take $O(n)$ time.

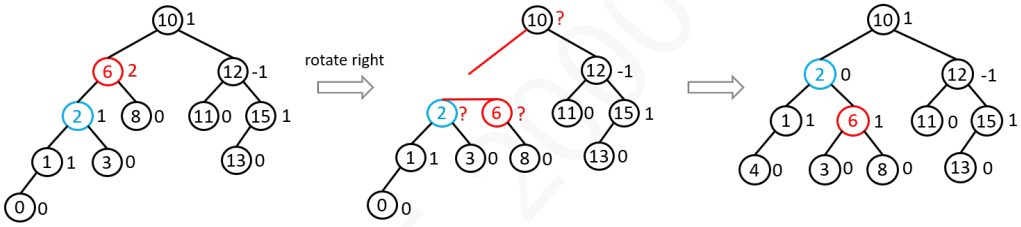
7.2.2 AVL Tree

An AVL tree is a type of binary search tree that maintains a *balance factor* for every node, defined as the height difference between its left and right subtrees. After every insertion or deletion, the tree performs *rotations* (single or double) to ensure that the balance factor of each node remains -1, 0, or 1. This guarantees that the height of the tree remains $O(\log n)$, ensuring logarithmic time complexity for search, insertion, and deletion operations.

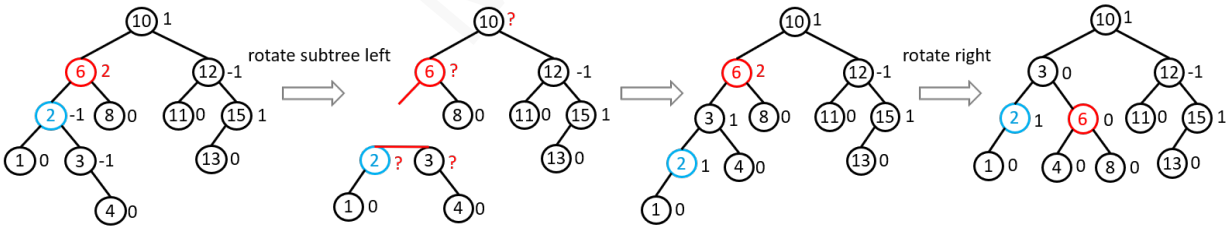
When inserting or deleting a node in an AVL tree, the tree may become imbalanced, *i.e.,* the difference in heights between the left and right subtrees becomes two. Through local rotations, the tree can be restored to balance. There are four cases that determine the rotation strategy for maintaining the AVL property, as summarized in Table 7.1.

Table 7.1: AVL tree balance adjustment policy

Imbalance Type	**Height Relation**	**Rotation Type**
Left-Left (LL)	Left subtree higher	Right Rotation
Right-Right (RR)	Right subtree higher	Left Rotation
Left-Right (LR)	Left subtree higher, right child higher	Left Rotation (left child) - Right Rotation (self)
Right-Left (RL)	Right subtree higher, left child higher	Right (right child) - Left Rotation (self)



(a) Node 6 is LL imbalanced: its left subtree is higher, and the left subtree of node 2 is also higher.



(b) Node 6 is LR imbalanced: its left subtree is higher, and the right subtree of node 2 is higher.

Figure 7.3: Examples of imbalanced nodes in an AVL tree and rotation methods.

Figure 7.3 illustrates two examples: one showing the LL imbalance type and the other showing the LR imbalance type. Note that during insertion and tree adjustment, the corresponding balance factors must be updated accordingly.

Besides AVL trees, red-black trees provide another approach to maintaining balanced binary search trees. They enforce less strict balancing requirements than AVL trees, reducing the need for rebalancing while still guaranteeing a tree height of $O(\log n)$. The details of red-black trees are beyond the scope of this course.

7.3 Problems on Graphs

This section introduces two fundamental graph problems: the shortest path problem and the minimum spanning tree problem.

7.3.1 Shortest Path Problem

The shortest path problem aims to find a path between two vertices that minimizes the sum of the edge weights along the path. Figure 7.4 illustrates an example, where the shortest path from A to E is A-B-D-E.

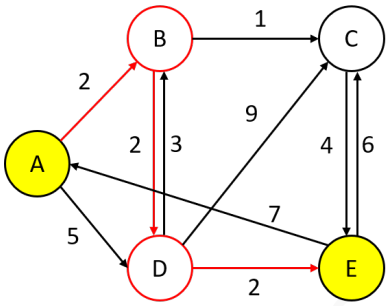


Figure 7.4: The shortest path from A to E is A-B-D-E.

When all edge weights are non-negative, Dijkstra's algorithm can be used to find the shortest paths from a source vertex to all other reachable vertices. The algorithm maintains a tentative distance for each vertex, representing the shortest path discovered so far from the source. Initially, the source has distance 0, while all other vertices have distance ∞. At each step, the algorithm selects the unvisited vertex with the smallest tentative distance and marks its distance as finalized. It then examines the outgoing edges of this vertex and updates the tentative distances of its neighbors whenever a shorter path is found. This process is called *edge relaxation*.

Table 7.2: Dijkstra' s algorithm progress for the shortest path problem of Figure 7.4.

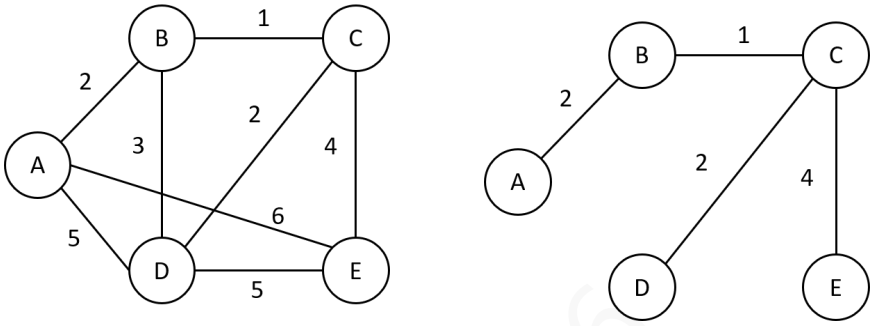
Step		**A**	**B**	**C**	**D**	**E**
1	Visited?	Y	N	N	N	N
	Distance	0	2	∞	5	∞
2	Visited?	Y	Y	N	N	N
	Distance	0	2	3	4	∞
	Previous	–	A	B	B	–
3	Visited?	Y	Y	Y	N	N
	Distance	0	2	3	4	7
	Previous	–	A	B	B	C
4	Visited?	Y	Y	Y	Y	N
	Distance	0	2	3	4	6
	Previous	–	A	B	B	D

Table 7.2 illustrates the execution of Dijkstra's algorithm on the graph in Figure 7.4. Starting from vertex A, the algorithm sets its distance to 0 and updates the distances of B and D to 2 and 5, respectively. Next, vertex B is selected, updating the distances of C and D to 3 and 4. Vertex C is then selected, updating

the distance of E to 7. Finally, selecting vertex D reduces the tentative distance of E from 7 to 6. Once vertex E is selected, its shortest distance from A is finalized as 6. By following the predecessor of each vertex, we can reconstruct the shortest path A-B-D-E.

7.3.2 Minimum Spanning Tree

The minimum spanning tree (MST) problem aims to find a subset of edges in a connected, undirected, weighted graph that connects all vertices without forming any cycles, while minimizing the total edge weight. Figure 7.5 illustrates a weighted graph and one of its minimum spanning trees. Two classic algorithms for finding an MST are Kruskal's algorithm and Prim's algorithm.



(a) A weighted graph. (b) A minimum spanning tree.

Figure 7.5: Illustration of the minimum spanning tree problem.

Kruskal's algorithm constructs an MST by repeatedly selecting the smallest-weight edge that connects two different components. Initially, each vertex forms a separate component. The edges are sorted in ascending order of weight and processed one by one. An edge is added if it connects two different components, thereby avoiding cycles. The process continues until all vertices are connected.

Prim's algorithm constructs an MST by starting from an arbitrary vertex and gradually expanding the tree. At each step, it selects the smallest-weight edge connecting a vertex already in the tree to a vertex outside the tree. The selected edge and the new vertex are then added to the tree. This process continues until all vertices are included. A min-heap can be used to efficiently select the smallest-weight edge at each step.

Exercise

1) Consider the following arrays, each representing a full binary tree in level order. Determine whether each array represents a valid min-heap.

$$A = [1, 3, 2, 7, 5, 4, 6],$$
$$B = [1, 2, 3, 4, 5, 6, 0],$$
$$C = [1, 2, 2, 3, 3, 4, 4].$$

2) Can we require a binary search tree to always be a complete binary tree? Explain how this requirement would affect its operations.

3) Explore the edge selection process in Prim's algorithm for finding a minimum spanning tree, and discuss how a min-heap can be used to improve its efficiency.

MF 20006