7 Algorithm II

Hui Xu, xuh@fudan.edu.cn

Learning Objective:

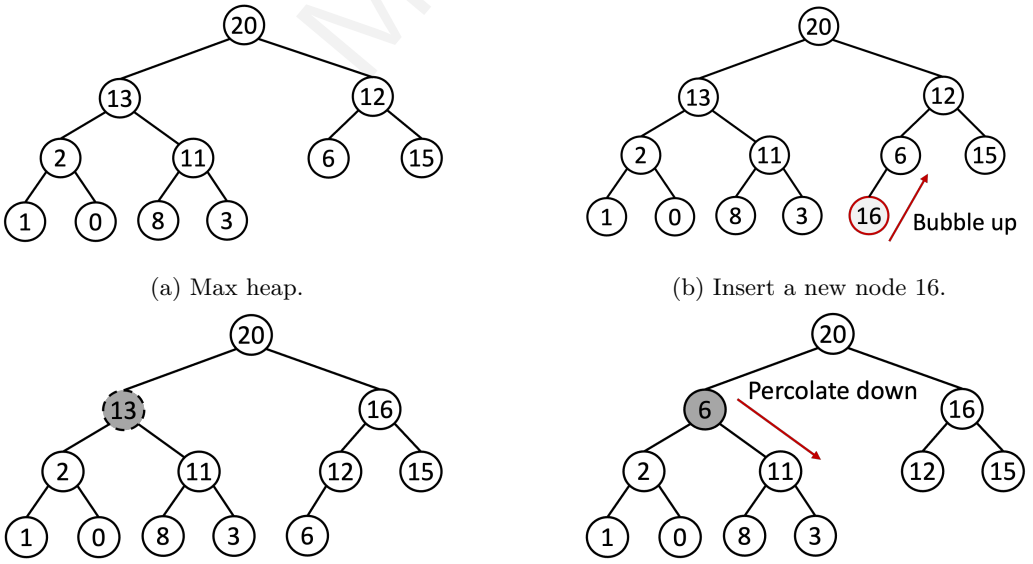
- Understand how to find the top-k elements using a max heap.
- Understand binary search trees and balance issues.
- Learn the algorithms for shortest paths and minimum spanning trees.

7.1 Max/Min Heap

Consider the following problem: given a sequence of numbers arriving at runtime, how can we store the incoming data so that retrieving the top-k (k is a small number) elements is efficient?

An intuitive idea is to keep all received data sorted. For example, each time a new element arrives, insert it into the appropriate position in the sorted sequence. In this way, retrieving the top-k elements takes constant time. However, finding the correct insertion position requires, on average, requires $n/2$ comparisons, where n is the number of existing elements.

To overcome this inefficiency, we can use a heap. A heap maintains a partial order rather than keeping all elements fully sorted. It is a special kind of complete binary tree, which means all levels are fully filled except possibly the last, and the last level' s nodes are filled from left to right. In a max heap, every parent node is greater than or equal to its children. In a min heap, every parent node is less than or equal to its children.



(a) Max heap.

(b) Insert a new node 16.

(c) Delete node 13.

(d) Replace the deleted node with the last.

Figure 7.1: Demonstration of a max heap with insertion and deletion operations.

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

- *Add an element*: Insert the new element at the end of the heap (the next available position in the complete binary tree), then bubble up to restore the heap property. In Figure 7.1b, adding 16 to the tree causes the element to bubble up along the path toward the root until it becomes smaller than its parent node, 20.
- *Delete an element*: Remove the element, move the last element to the current place, and percolate down to maintain the heap order. In Figure 7.1c, deleting 13 moves the last element (6) to the current place, which is then swapped with its larger child repeatedly until the heap property is restored.
- *Find the top-k element*: Repeatedly remove the root (the current maximum or minimum) k times, or traverse the heap iteratively if removal is not required.

In practice, a heap is usually stored as an array or list in *level order*. For example, the max heap in Figure 7.1a can be represented as the list:

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

This representation eliminates the need for explicit pointers, unlike a standard binary tree. Given a node at index i, the indices of its left and right children can be calculated as:

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

Similarly, given a child node at index i, the index of its parent can be computed as:

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

Code 7.1 presents an implementation of a max heap with two main functions: `push()` and `pop()`. `push()` corresponds to inserting a new element into the heap, while `pop()` corresponds to deleting the root node.

```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, users may need to search for a particular value in a list of data, rather than the largest or smallest element. In such cases, a max or min heap is not suitable. A naive approach based on sorting and linear search is also inefficient, as it requires $O(n)$ time for insertion and $O(n)$ time for searching.

In this section, we introduce an approach that allows both insertion and searching in $O(\log n)$ time.

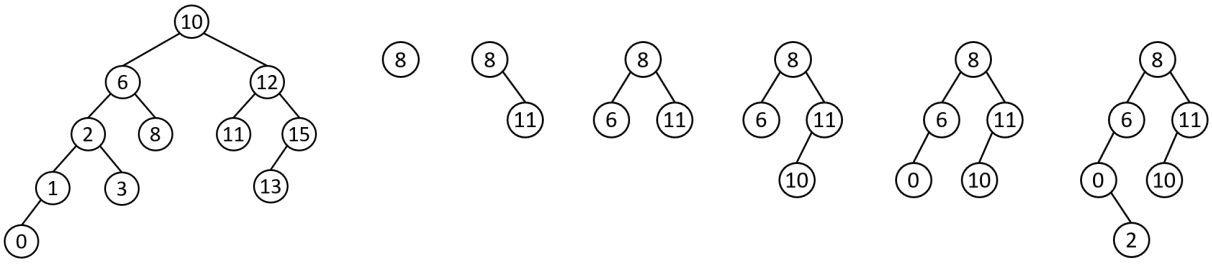
7.2.1 Naive BST

A binary search tree (BST) 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 contain values less than that node.
- All nodes in the right subtree of a node contain values greater than that node.

Figure 7.2a demonstrates a sample binary search tree (BST). Code 7.2 provides an implementation with `insertion()` and `search()` functions. Note that the structure of a binary search tree depends on the order in which elements arrive. Figure 7.2b illustrates the construction process of a BST for a given sequence of elements. If the insertion order changes, the resulting tree will also be different.

In an ideal case, the tree should be balanced to achieve $O(\log n)$ performance for insertion and search operations. In the worst case, the tree can degenerate into a list, and the performance degrades to $O(n)$.



(a) Binary search tree.

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

Figure 7.2: Demonstration of BST.

```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: Max heap implemented with Python

7.2.2 AVL Tree

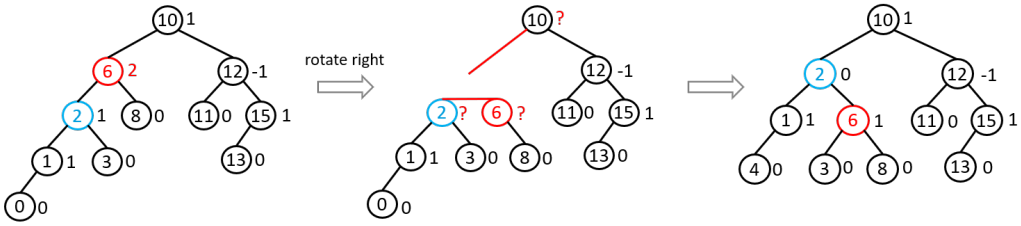
An AVL tree is a type of binary search tree that maintains a *balance factor* for every node, defined as the height difference between its left and right subtrees. After every insertion or deletion, the tree performs *rotations* (single or double) to ensure that the balance factor of each node remains -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

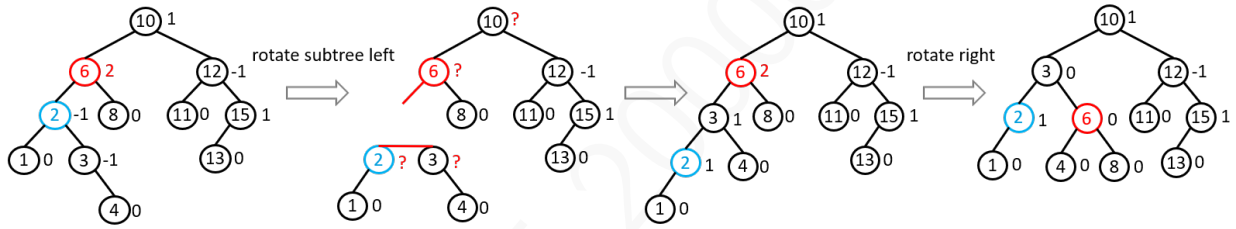
in heights between the left and right subtrees becomes two. Through local rotations, the tree can be restored to balance. There are four cases that determine the rotation strategy for maintaining the AVL property, as summarized in Table 7.1.

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 imbalanced: its left subtree is higher, and the left subtree of node 2 is also higher.



(b) Node 6 is 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 the AVL tree, the Red-Black tree is another approach for maintaining a balanced binary search tree. It employs a trade-off between strict balance and operational cost, avoiding the need to maintain precise balance factors while still guaranteeing logarithmic height. In particular, every node is either red or black, and it requires that the tree satisfies specific coloring properties (such as no two consecutive red nodes on any path and equal black-height for all paths from a node to its descendants). By checking for local red–red conflicts, it does not require tracking a global imbalance factor.

7.3 Problems on Graphs

This section discusses two classic problems on graphs.

7.3.1 Shortest Path Problem

The shortest path problem aims to find a path between two vertices such that the sum of the edge weights is minimized.

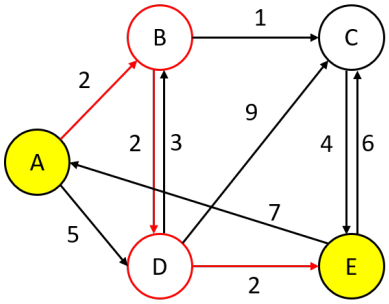


Figure 7.4: The shortest path from A to E is A-B-D-E.

If all edge weights are non-negative, a classic algorithm for solving this problem is Dijkstra' s algorithm. Dijkstra' s algorithm proceeds by repeatedly selecting the unvisited vertex with the smallest tentative distance, marking it as visited, and relaxing all of its outgoing edges. As shown in Table 7.2, the algorithm starts from vertex A, setting its distance to 0 and updating its neighbors B and D to 2 and 5, respectively. In the next step, vertex B (the closest unvisited vertex) is selected, updating the distances of C and D to 3 and 4. Then, vertex C is visited, leading to an update of vertex E with distance 7. Finally, visiting vertex D shortens the path to E from 7 to 6. When all vertices are visited, the shortest-path tree rooted at A is obtained. The previous node recorded during each relaxation step can be used to reconstruct the shortest path from the source to any vertex.

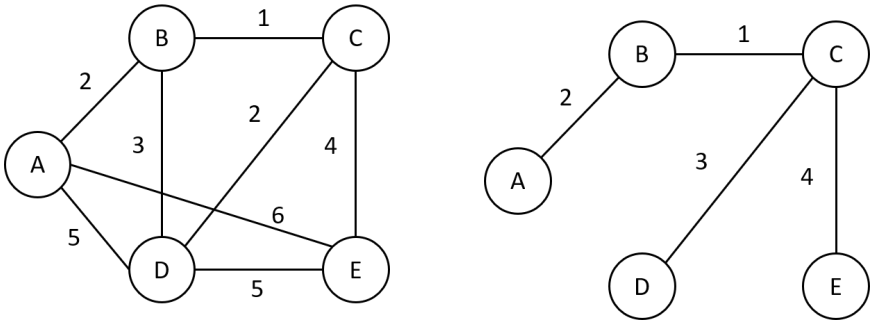
Table 7.2: Dijkstra' s algorithm progress for the shortest path problem of Figure 7.4.

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

7.3.2 Minimal Spanning Tree

The minimum spanning tree (MST) problem seeks a subset of edges that connects all vertices with the smallest possible total edge weight, without forming any cycles. Figure 7.5 demonstrates a weighted graph and corresponding MST. Typical algorithms to construct an MST include Kruskal' s algorithm and Prim' s algorithm.

Kruskal' s algorithm builds the MST by repeatedly adding the smallest-weight edge that connects two different components. Initially, each vertex is an isolated component. Edges are sorted in ascending order of weight, and one by one, edges are added as long as they do not form a cycle. The process continues until all vertices are connected.



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

Figure 7.5: Demonstration of minimal spanning tree.

Prim' s algorithm grows the MST from an arbitrary starting vertex. It maintains two sets of vertices: those already included in the MST and those not yet included. At each step, it selects the smallest-weight edge that connects a vertex inside the tree to one outside, and adds that edge and vertex to the tree. The process repeats until all vertices are included. To efficiently select the smallest-weight edge at each step, a min-heap is often used.