

Dynamic Arrays

Definition:

A dynamic array is a variable-sized list data structure that allows elements to be added or deleted. It provides efficient random access to elements and can resize itself based on the number of elements it contains.

Comparison with the static array:

Dynamic Arrays	Static Arrays
Memory allocation: Allocates memory during runtime and can resize themselves as needed.	Memory allocation: Static arrays allocate a fixed amount of memory during compile-time. Once the size is set, it cannot be changed during runtime
Memory management: Dynamic arrays require sophisticated memory management especially during resizing.	Memory management: Static arrays have a simpler memory management since they allocate a fixed amount of memory upfront.
Usage: These are preferred when the number of elements is unknown in advance or can vary significantly.	Usage: Static arrays are suitable when the number of elements is known in advance and remains constant throughout the program’s execution.
<div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div><div></div><div></div><div></div></div>	<div><div>7</div><div>-4</div><div>11</div><div>0</div></div>

Some key advantages:

- **Resizable:**
 - Dynamic arrays adjust their size as elements are added or removed, providing flexibility in managing varying amounts of data.
- **Memory allocation:**
 - They allocate memory dynamically, expanding or shrinking as needed.

Operations:

Operation	Example	Time Complexity
Insertion: Insertion involves adding a new element to the array.	<div><div><div>0</div><div>1</div><div>2</div><div>3</div></div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div></div></div><div>← Insert “19”</div></div> <div><div><div>0</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div></div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div><div></div><div></div><div></div></div></div>	O(1)
Deletion: Deletion involves removing an element from the array.	<div><div><div>0</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div><div>8</div></div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div><div>0</div><div>0</div><div>0</div><div>0</div></div></div>	O(n)
Resizing: Resizing ensures that a dynamic array can accommodate new elements when it reaches capacity.	<div><div><div>0</div><div>1</div><div>2</div><div>3</div><div>4</div></div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div></div></div> <div><div><div>0</div><div>1</div><div>2</div><div>3</div><div>4</div><div>5</div><div>6</div><div>7</div><div>8</div><div>9</div></div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div><div>0</div><div>0</div><div>0</div><div>0</div><div>0</div></div></div>	O(n)

Implementation:

- **Resizable:** Dynamic arrays rely on a contiguous block of memory allocated for an array and use pointers to manage this memory.
- **Mechanism of resizing:**
 - **Allocation:** Allocate a new array with a larger capacity.
 - **Copying:** Copy all the elements to the new array.
 - **Deallocation:** Deallocate the old array.
 - **Pointer:** Update the pointer to reference the new array

Key concepts:

- Size vs. capacity

Size is the number of elements currently stored in the array.	Capacity is the total number of elements the array can hold before resizing.
<div><div><div>7</div><div>-4</div><div>11</div><div>0</div><div>19</div><div>34</div></div><div>size = 6</div></div> <div><div><div>?</div><div>?</div><div>?</div><div>?</div><div>?</div><div>?</div></div><div>free space</div></div> <div>capacity = 12</div>	

Applications:

- Manages lines of text dynamically in text editors.
- Implements resizable lists, stacks, and queues.
- Tracks dynamic entities like players and objects in game development.

Linked list (circular and circular doubly)

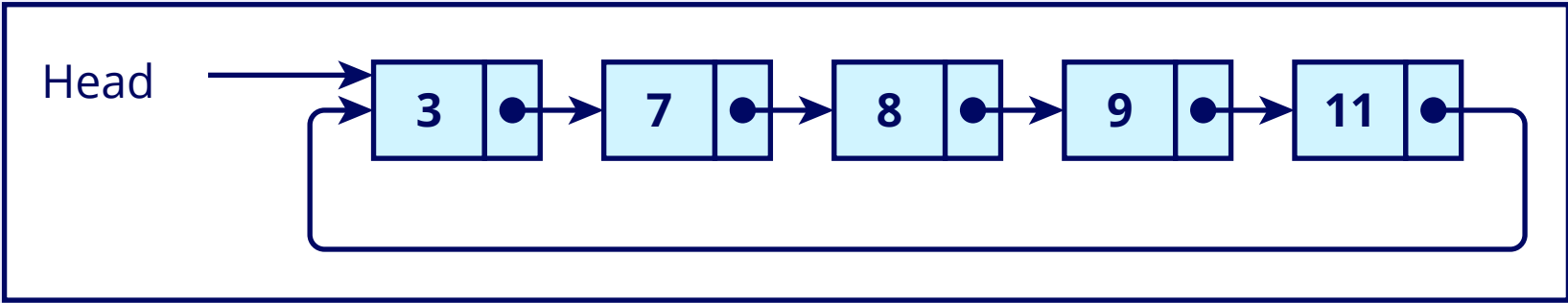
Circular linked list

Definition:

A linked list where the last node points back to the first node, forming a circular structure. Unlike a singly linked list, it has no NULL reference at the end.

Traversal:

Forward traversal similar to singly linked lists. Also, it can start from any node and continue until it reaches the same node again.



Operation	Example	Time Complexity
Insertion	<div>Head → <div><div>3</div><div>•</div></div> → <div><div>7</div><div>•</div></div> → <div><div>9</div><div>•</div></div> → <div><div>11</div><div>•</div></div> → Head</div> <p>After inserting 8, the above CLL should be changed to the following</p> <div>Head → <div><div>3</div><div>•</div></div> → <div><div>7</div><div>•</div></div> → <div><div>8</div><div>•</div></div> → <div><div>9</div><div>•</div></div> → <div><div>11</div><div>•</div></div> → Head</div>	O(n)
Deletion	<div>Head → <div><div>3</div><div>•</div></div> → <div><div>7</div><div>•</div></div> → <div><div>8</div><div>•</div></div> → <div><div>9</div><div>•</div></div> → <div><div>11</div><div>•</div></div> → Head</div> <p>(Node 9 is highlighted for deletion)</p>	O(n)

Applications:

- Implementation of queue data structure, used in round-robin scheduling and creating a playlist for a music or media player.

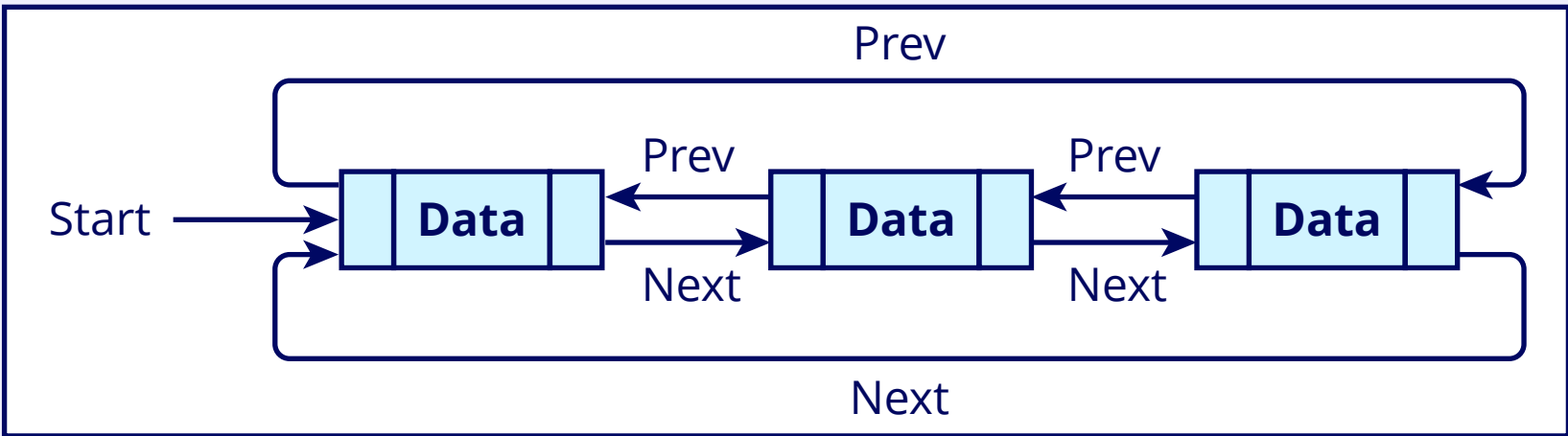
Circular doubly linked list

Definition:

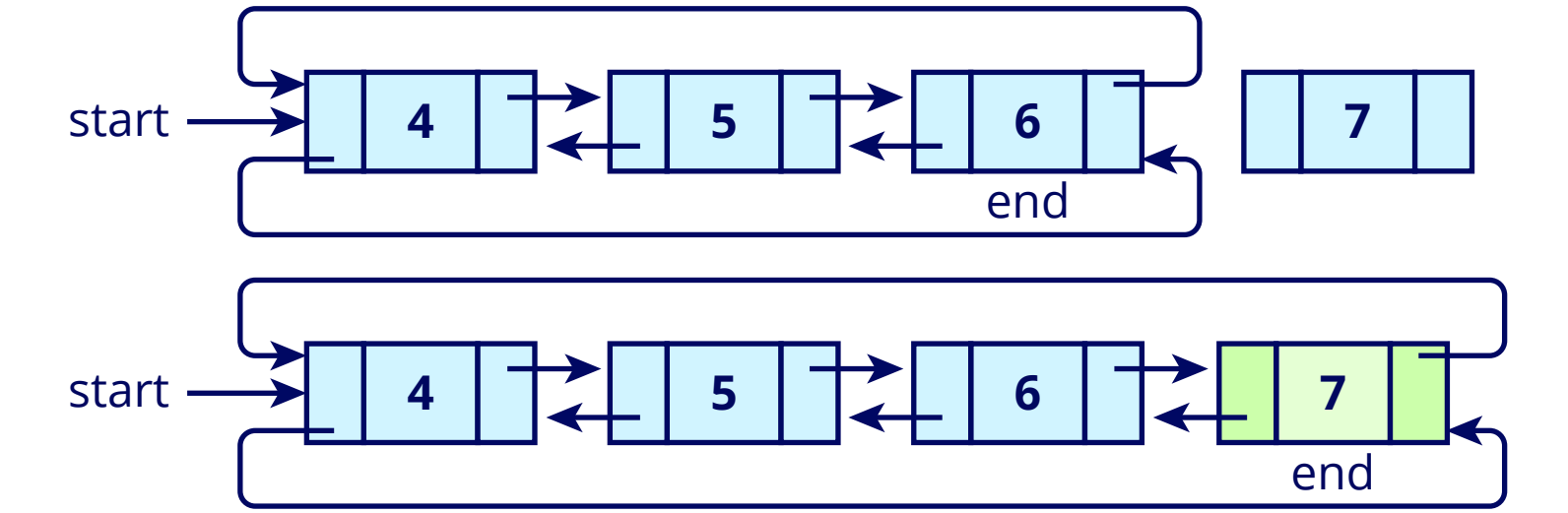
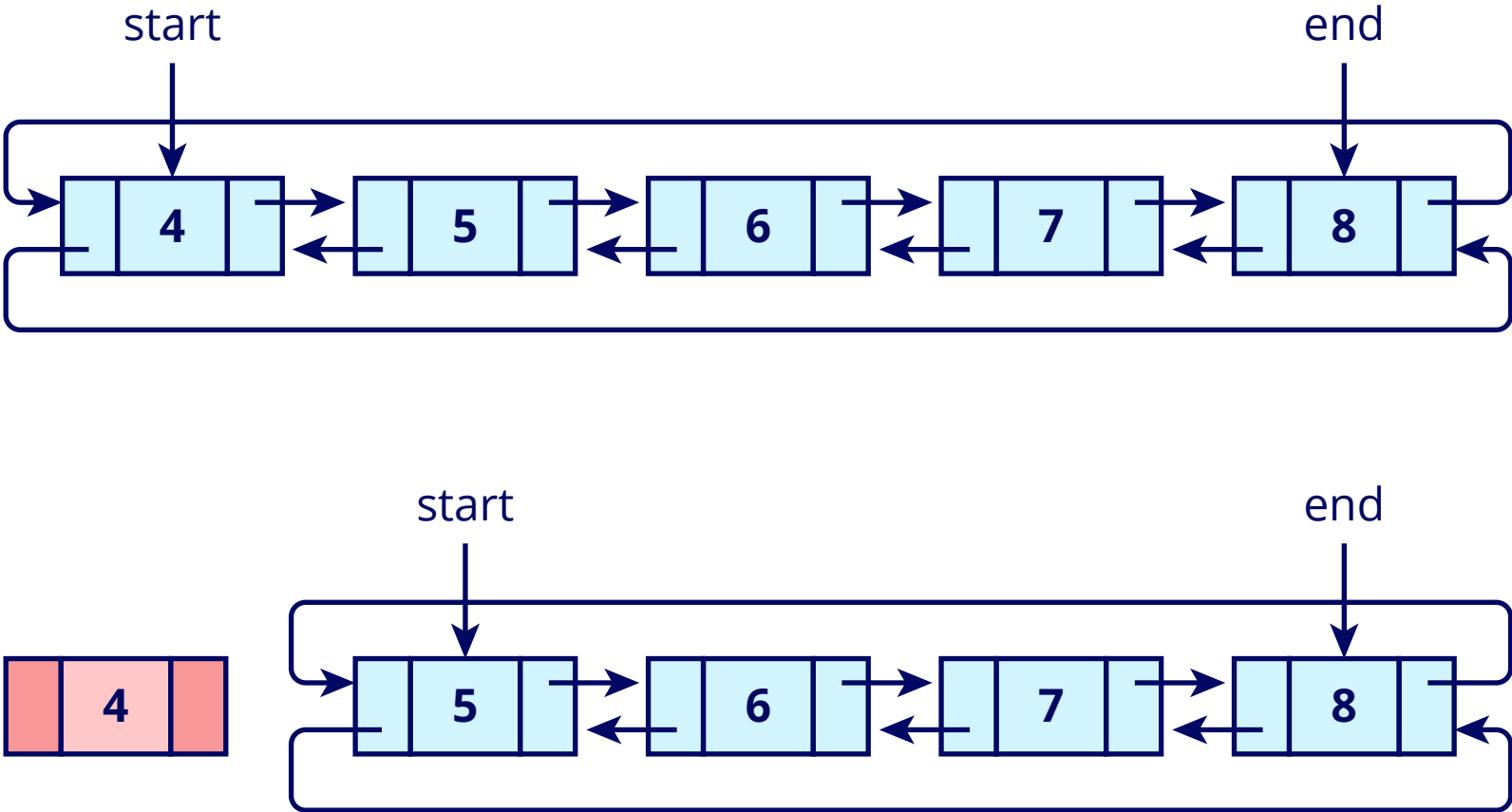
A linked list where each node points to its next and previous nodes, and the last node points back to the first node and vice versa, forming a bidirectional circular structure.

Traversal:

Bidirectional: Allows both forward and backward traversal.



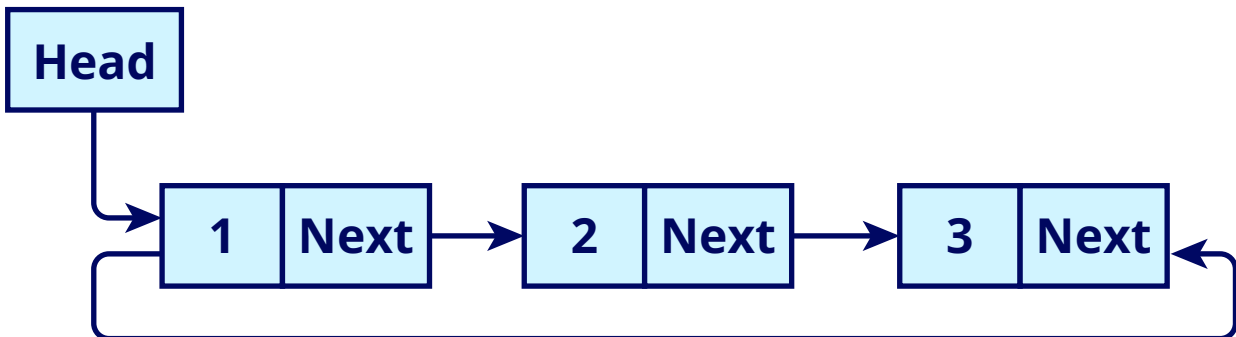
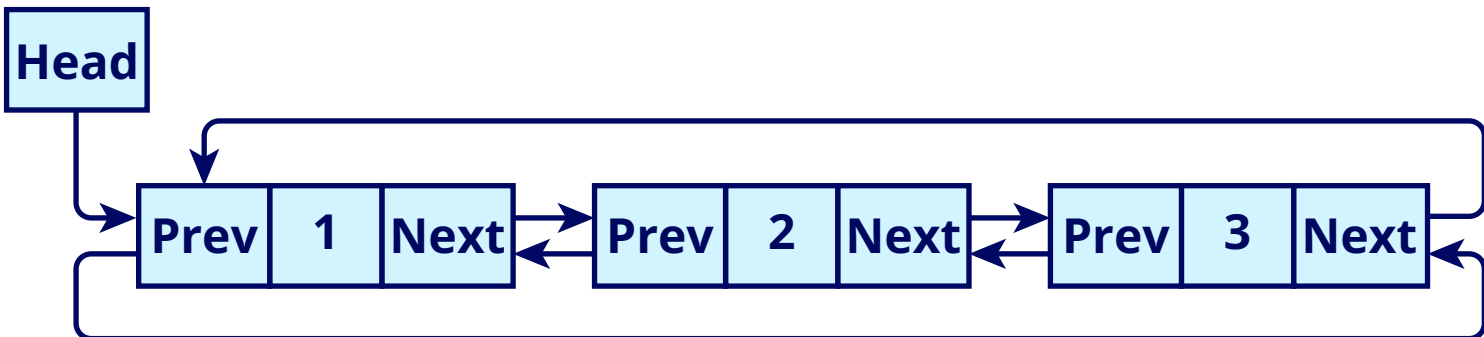
Insertion and deletion operations:

Operation	Example	Time Complexity
Insertion		O(n)
Deletion		O(n)

Applications:

- Implementation of double-ended queues, advanced data structures like Fibonacci heaps, browser caches, and shopping carts on online websites.

Circular linked lists vs. circular doubly linked lists

Circular Linked Lists	Circular Doubly Linked Lists
Advantages: <ul style="list-style-type: none">Simpler implementationLess memory overhead	Advantages: <ul style="list-style-type: none">Supports bidirectional traversalEasier insertion and deletion
Disadvantages: <ul style="list-style-type: none">Allows unidirectional traversal onlyComplex insertion and deletion operations at positions other than the beginning	Disadvantages: <ul style="list-style-type: none">Complex implementationMemory overhead because of storing more pointers
Use cases: Useful in applications requiring simple circular navigation, such as a round-robin scheduler.	Use cases: Ideal for applications where efficient <i>bidirectional</i> traversal and manipulation
	

Best practices:

- **Common pitfalls to avoid:**
 - Ensure the last node points back to the first node in circular linked lists.
 - Ensure that all references are updated after node deletion to avoid memory leaks.
- **Error handling:**
Check for any NULL references during traversal to prevent infinite loops.
- **Memory management:**
Implement efficient memory allocation techniques to minimize overhead.

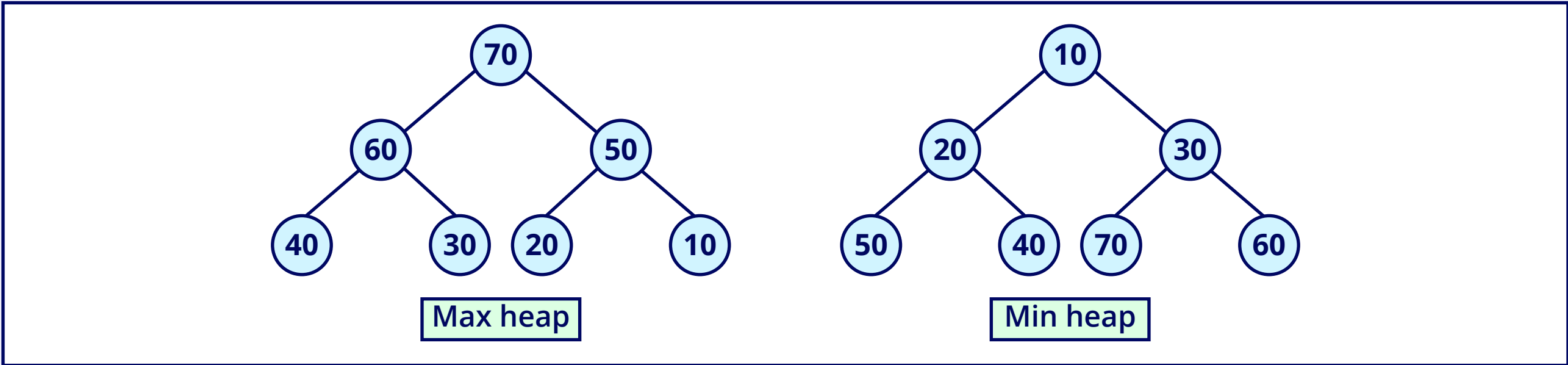
Heaps

Definition:

A heap is a binary tree where each node has a specific relationship with its children, defined by the heap property.

Types of heaps:

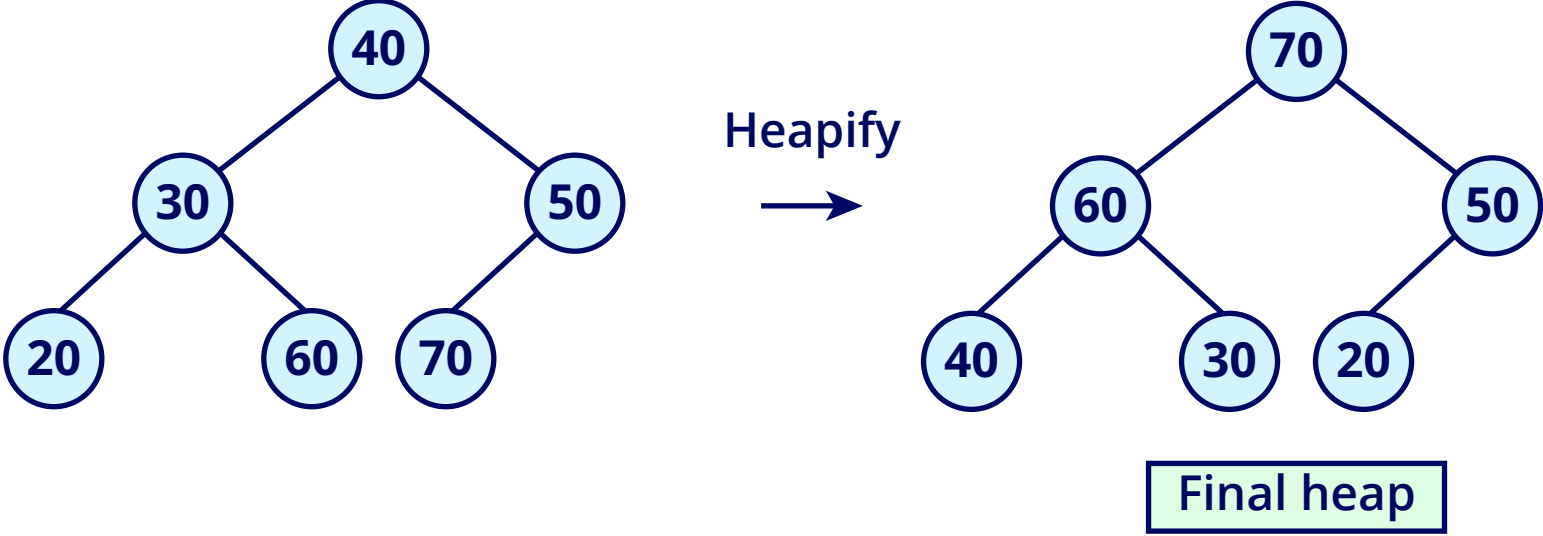
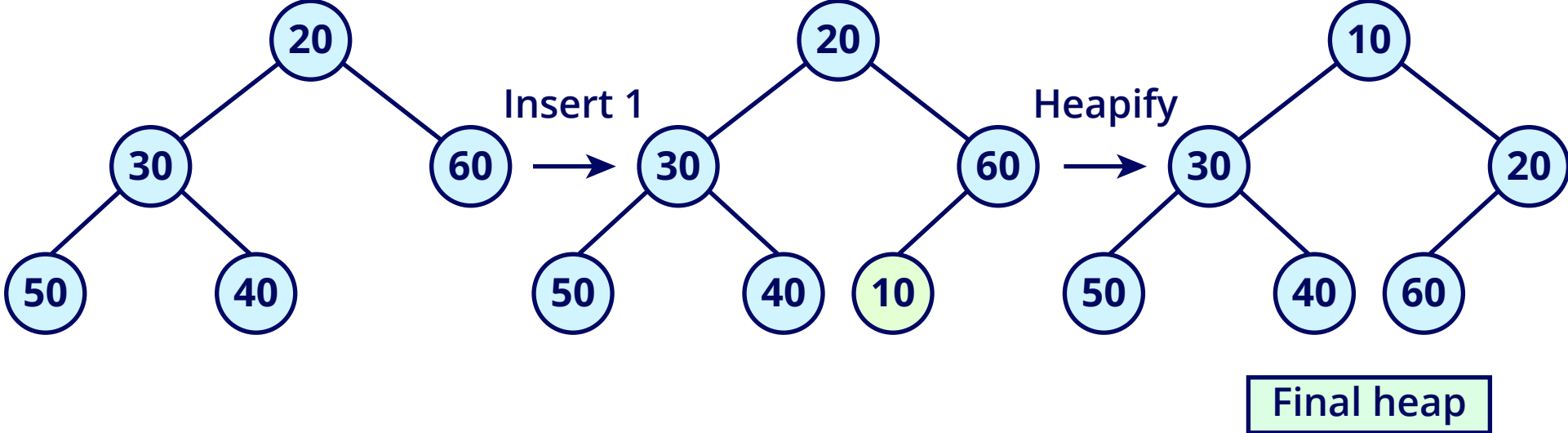
- Max heap
- Min heap

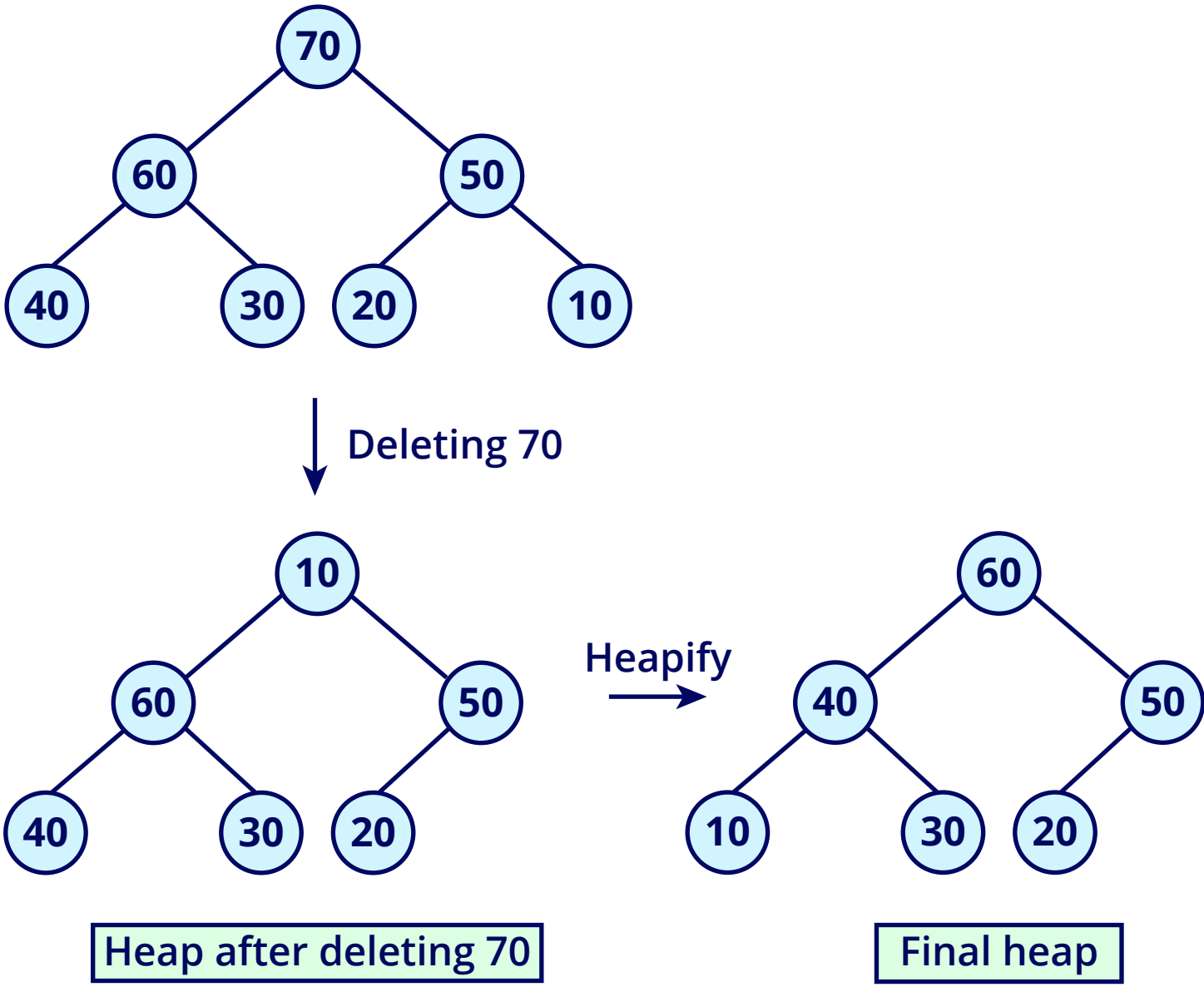
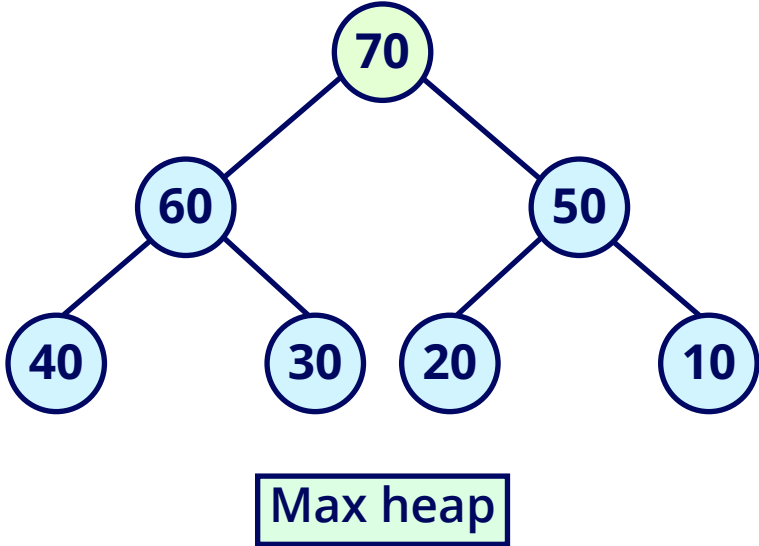


Heap property:

For a max heap, each parent node is greater than or equal to its children, while for a min heap, each parent node is less than or equal to its children.

Heap operations

Operation	Example	Time Complexity (worst-case)
Heapify: Transforms a binary tree into a heap by ensuring that the heap property is maintained at every node		O(log n)
Insertion: A new element is added to the heap, and the heap property is maintained.		O(log n)

Operation	Example	Time Complexity (worst-case)
Deletion: Removes the root element (the maximum in a max heap or the minimum in a min heap) and heapify the remaining elements.	 <div>Heap after deleting 70</div> <div>Final heap</div>	O(log n)
Extract Max/Min: Removes and returns the root element of the heap and heapify the remaining elements.	 <div>Max heap</div>	O(log n)

Applications:

- Implementation of the priority queues where elements are retrieved based on their priority (maximum or minimum value).
- It is used in sorting algorithms like Heapsort and graph algorithms like Dijkstra’s and Prim’s.

Hash Tables

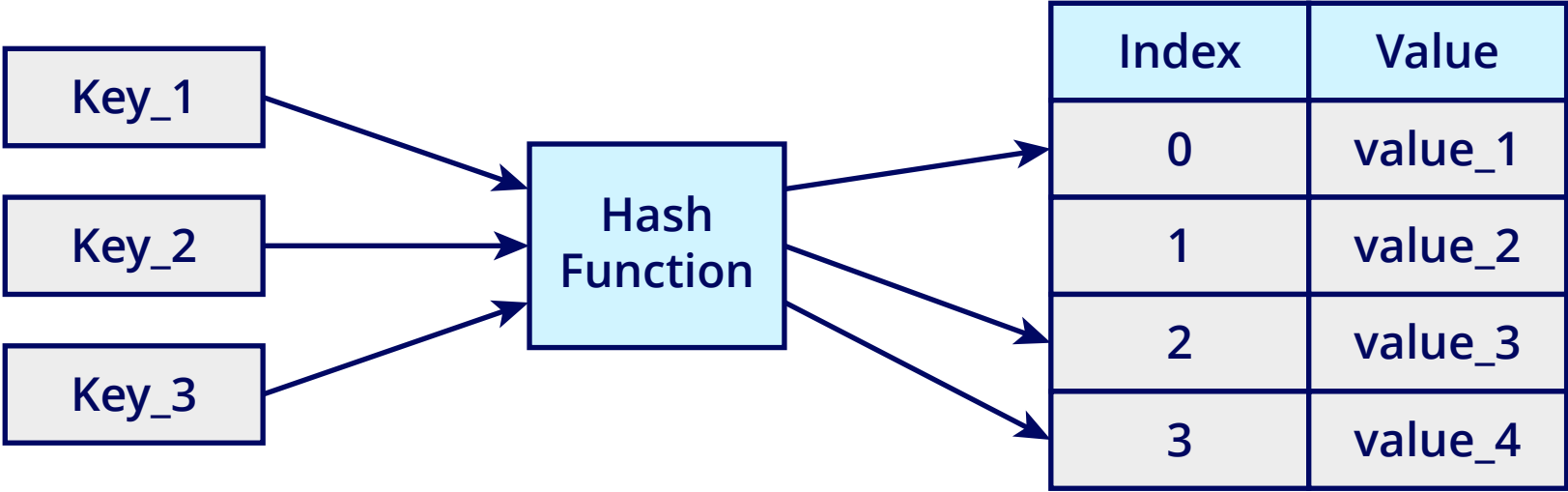
Definition:

Stores key-value pairs using a hash function.

Key concepts:

- **Hashing Function:** Maps keys to indices.
- **Collision Resolution:** Chaining, open addressing.
- **Load Factor:** $\alpha = n/k$ (n = number of entries, k = table size)

Operation	Example	Time Complexity										
Insertion	<div>Adding: 1 and 2</div> <div><div>1 →</div><div>2 →</div><div>Hash Function</div><div>100 →</div><div>105 →</div><table><tr><th>Index</th><th>Value</th></tr><tr><td>100</td><td>value_1</td></tr><tr><td>602</td><td>value_2</td></tr><tr><td>304</td><td>value_3</td></tr><tr><td>105</td><td>value_4</td></tr></table></div>	Index	Value	100	value_1	602	value_2	304	value_3	105	value_4	O(1)
Index	Value											
100	value_1											
602	value_2											
304	value_3											
105	value_4											
Deletion	<div>Deleting: 2</div> <div><div>2 →</div><div>Hash Function</div><div>105 →</div><table><tr><th>Index</th><th>Value</th></tr><tr><td>100</td><td>value_1</td></tr><tr><td>602</td><td>value_2</td></tr><tr><td>304</td><td>value_3</td></tr><tr><td>105</td><td>value_4</td></tr></table></div>	Index	Value	100	value_1	602	value_2	304	value_3	105	value_4	O(1)
Index	Value											
100	value_1											
602	value_2											
304	value_3											
105	value_4											

Operation	Example	Time Complexity
Searching		O(1)

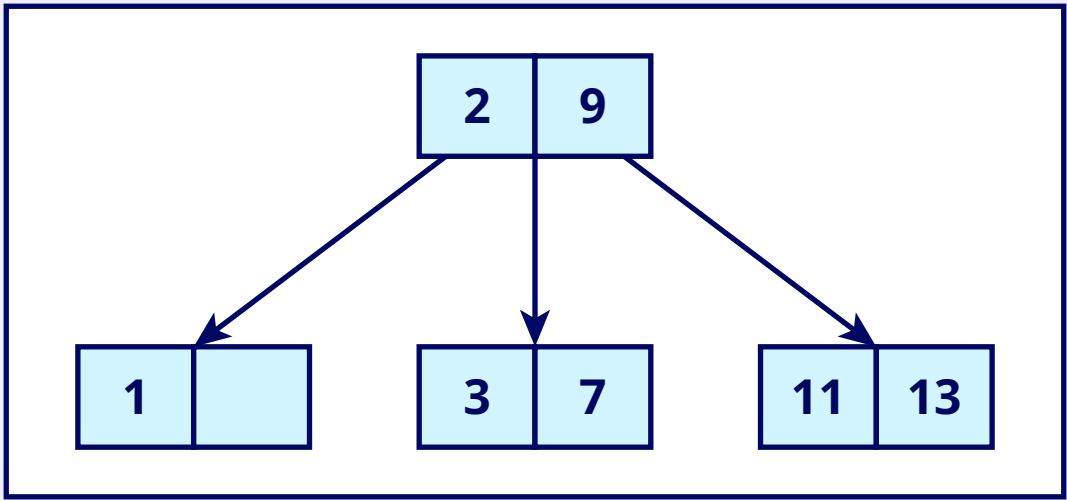
- Applications:
- Fast data retrieval.
 - Implementing associative arrays and database indexing.

Trees (2-3 Trees, AVL, Red black)

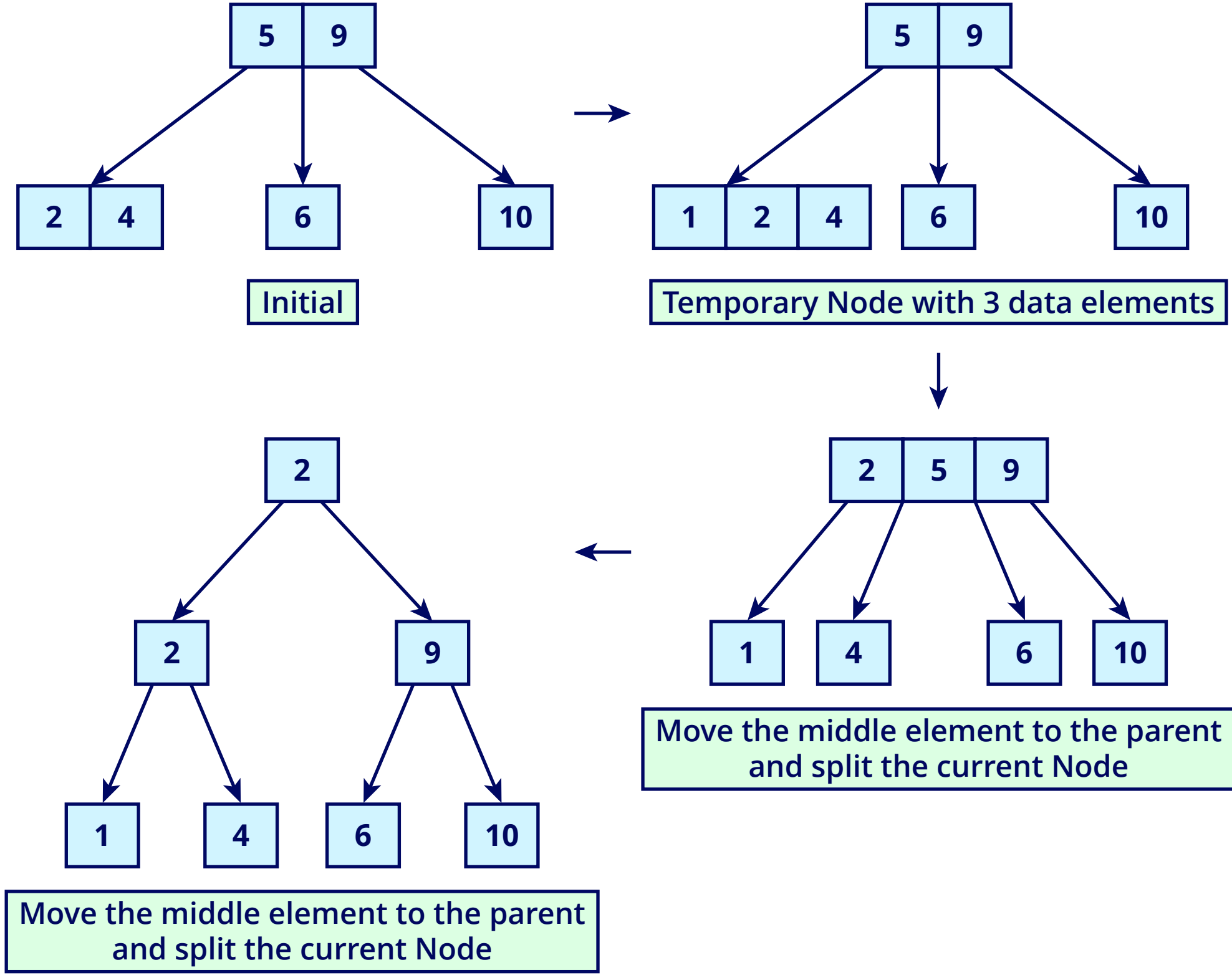
2-3 trees

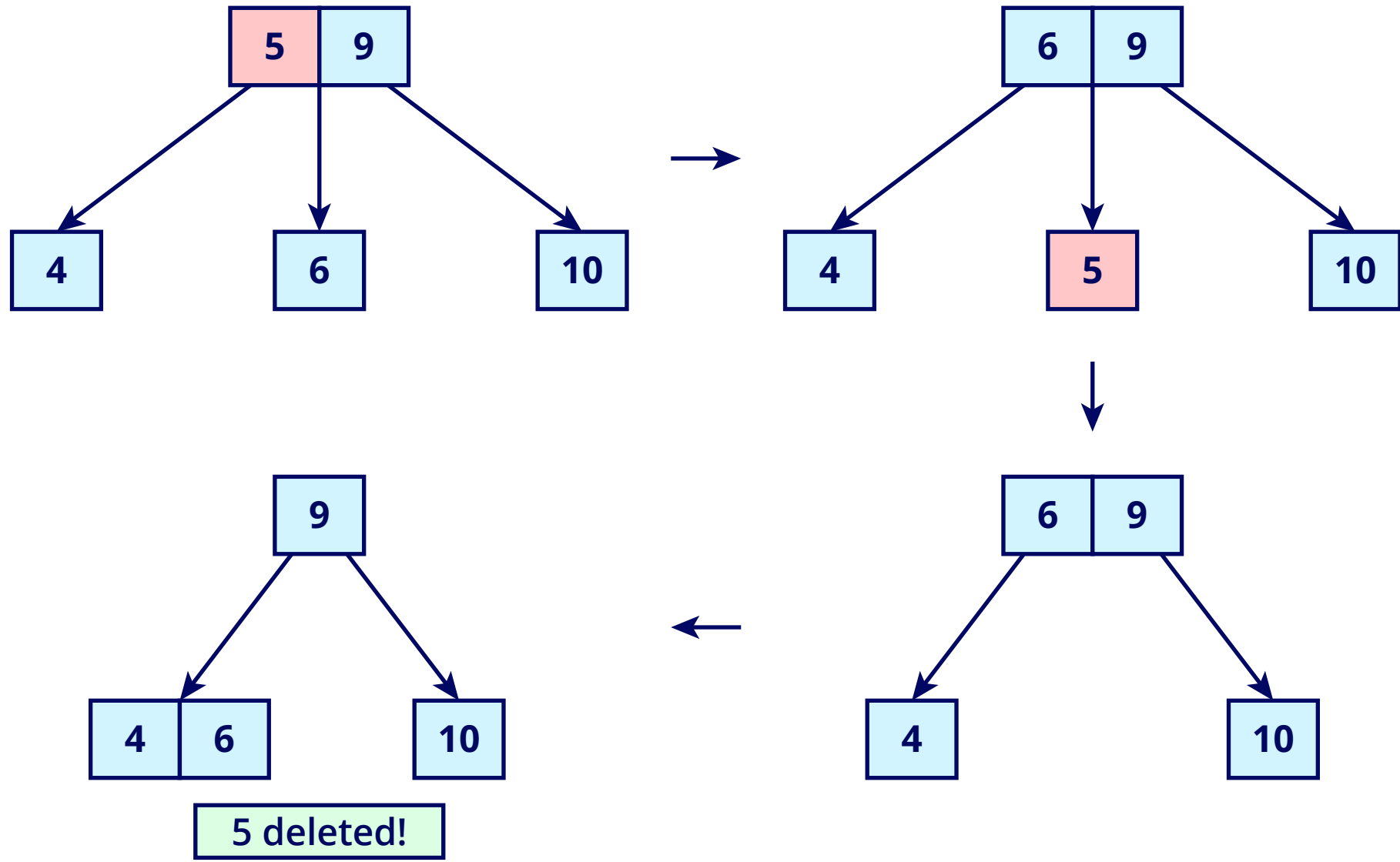
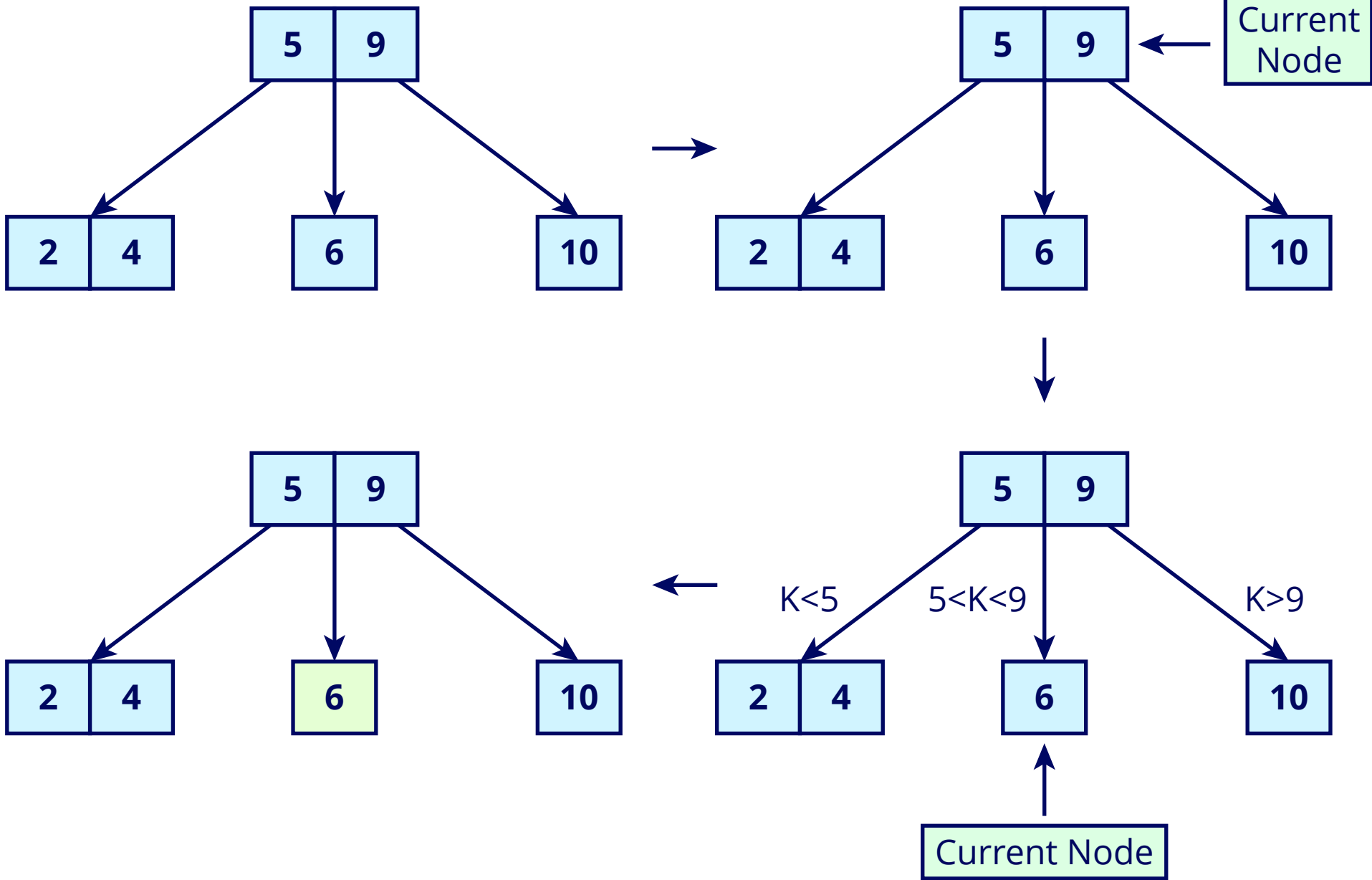
Definition:

A 2-3 tree is a balanced search tree where each node can have two or three children and one or two keys.



Operations:

Operation	Example	Time Complexity
<div>Insertion: Involves finding the correct leaf position and splitting nodes if necessary to maintain balance.</div>	<p>Insert 1 in the following 2-3 Tree:</p>  <p>Initial</p> <p>Temporary Node with 3 data elements</p> <p>Move the middle element to the parent and split the current Node</p> <p>Move the middle element to the parent and split the current Node</p>	O(log n)

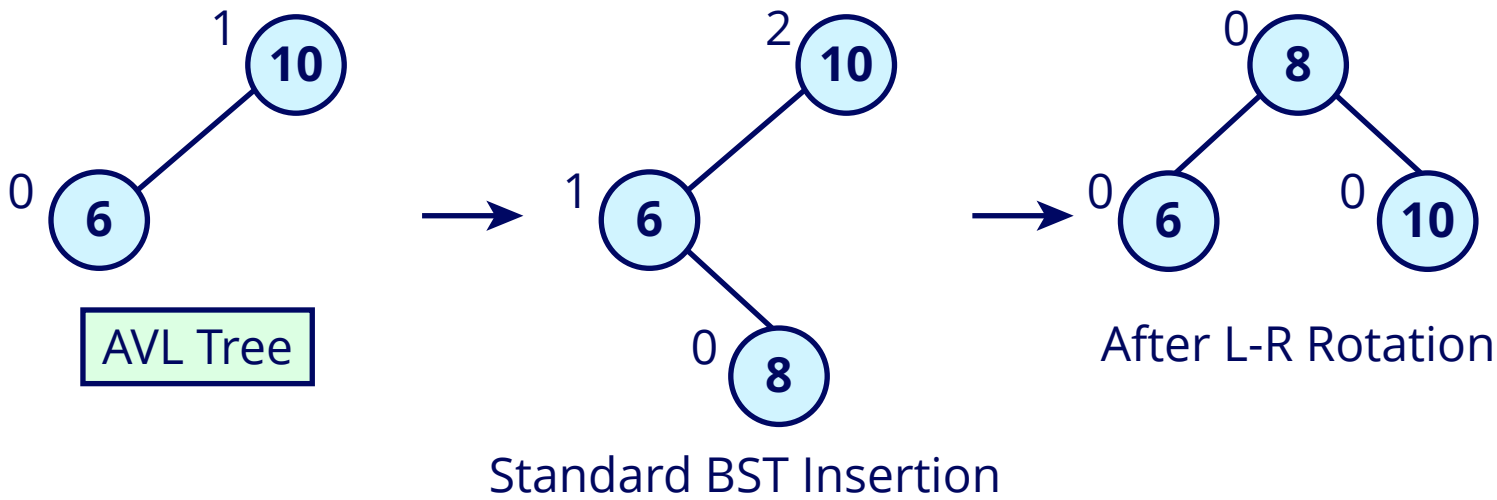
Operation	Example	Time Complexity
Deletion: It Involves removing the key and adjusting the tree to maintain balance.	<div><div>Delete: 5</div><div></div></div>	$O(\log n)$
Searching: Searching is navigating through the tree based on the keys, similar to binary search.	<div><div>Search 6 in the following 2-3 Tree:</div><div></div></div>	$O(\log n)$

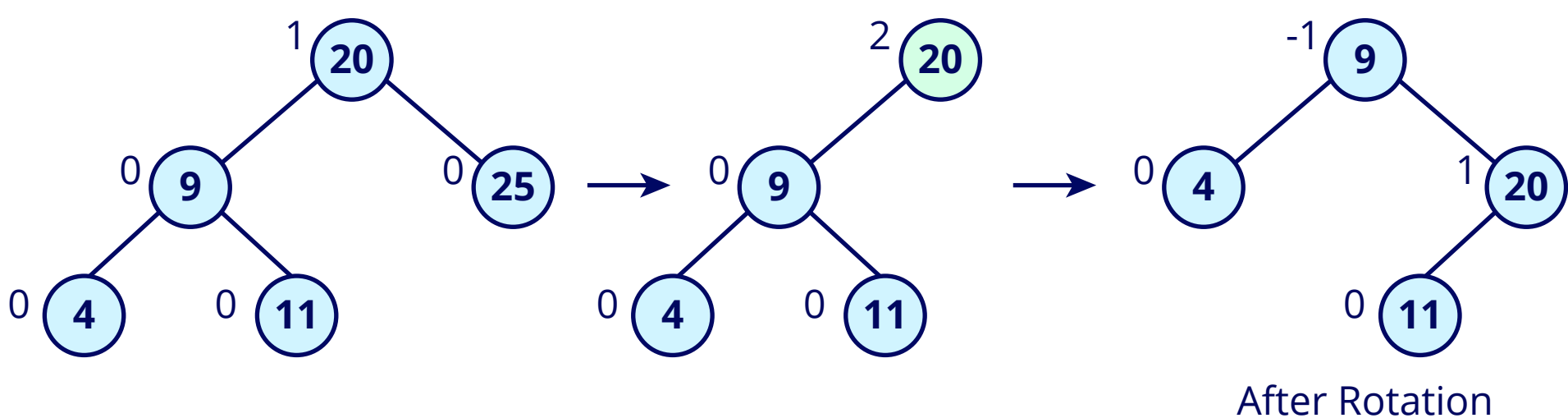
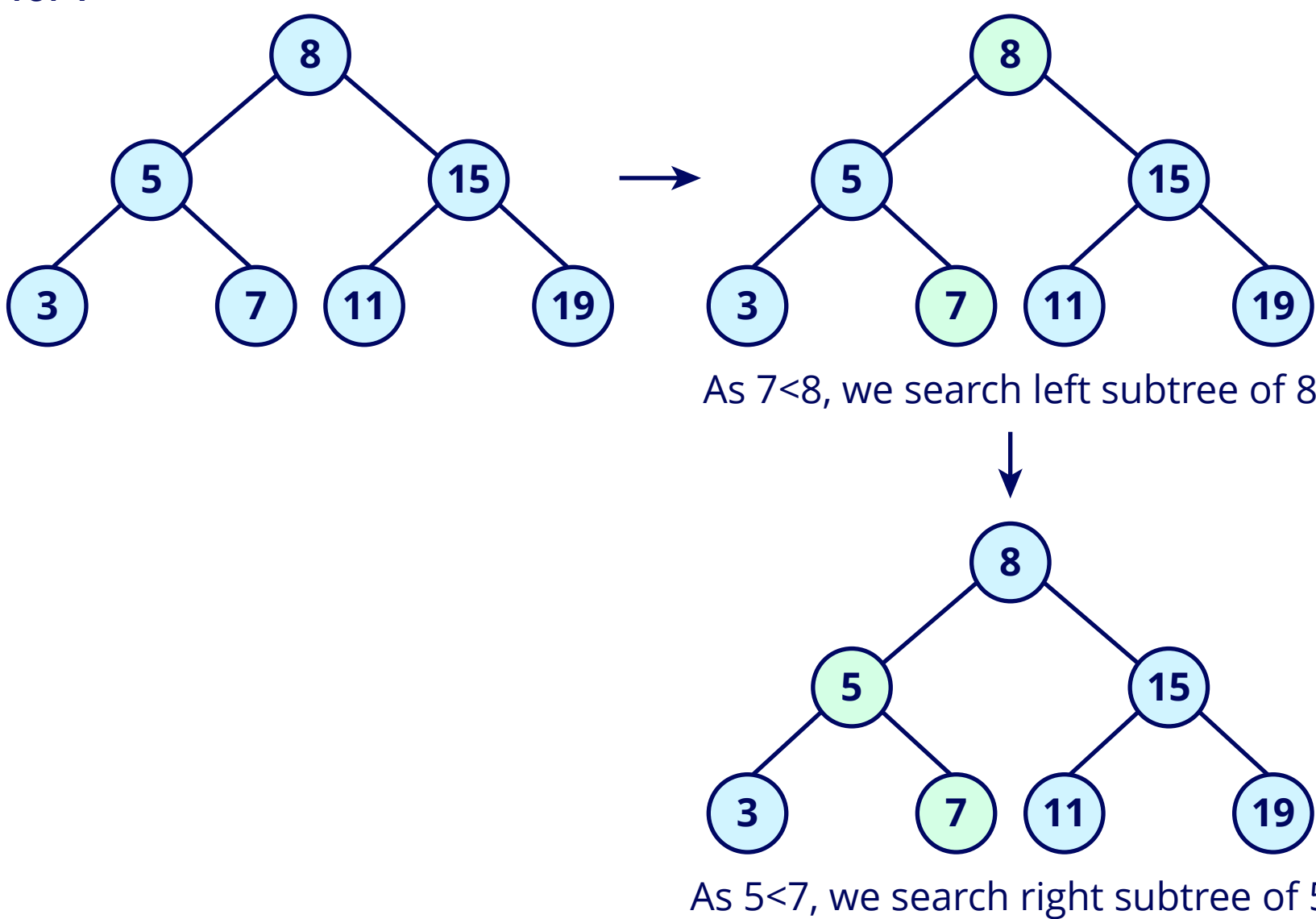
Balance maintenance:
They maintain balance by splitting nodes during insertion and merging or redistributing nodes during deletion.

AVL trees

Definition:
AVL (Adelson-Velsky and Landis) trees are self-balancing binary search trees where the difference in heights between any node’s left and right subtrees is at most one.

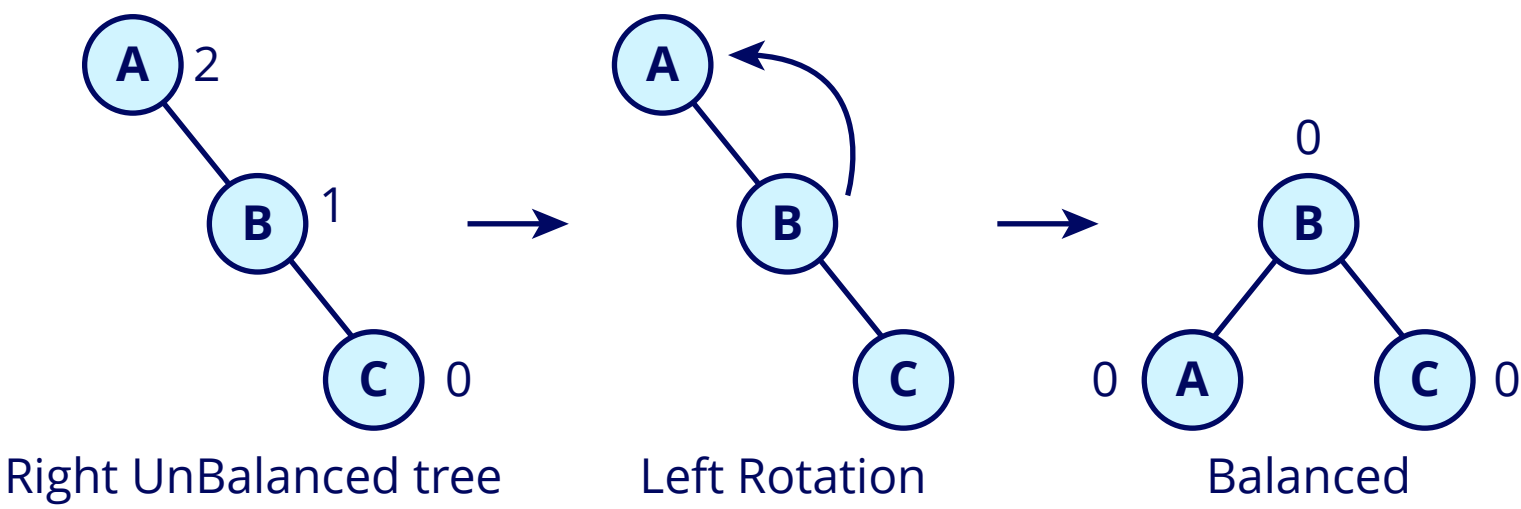
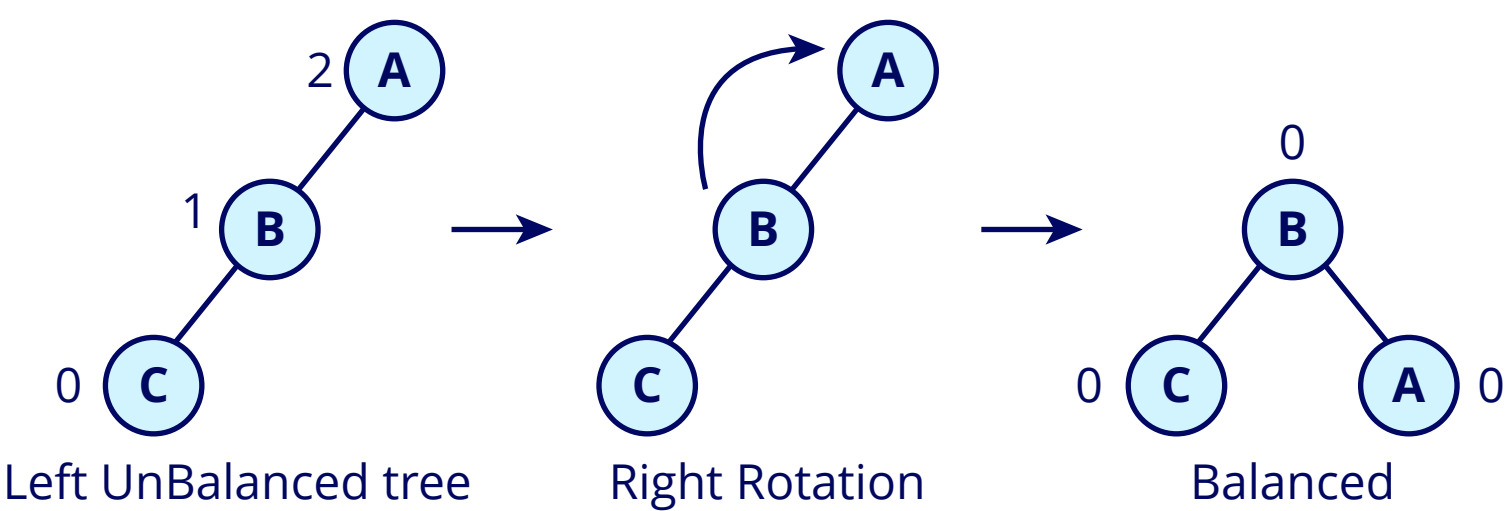
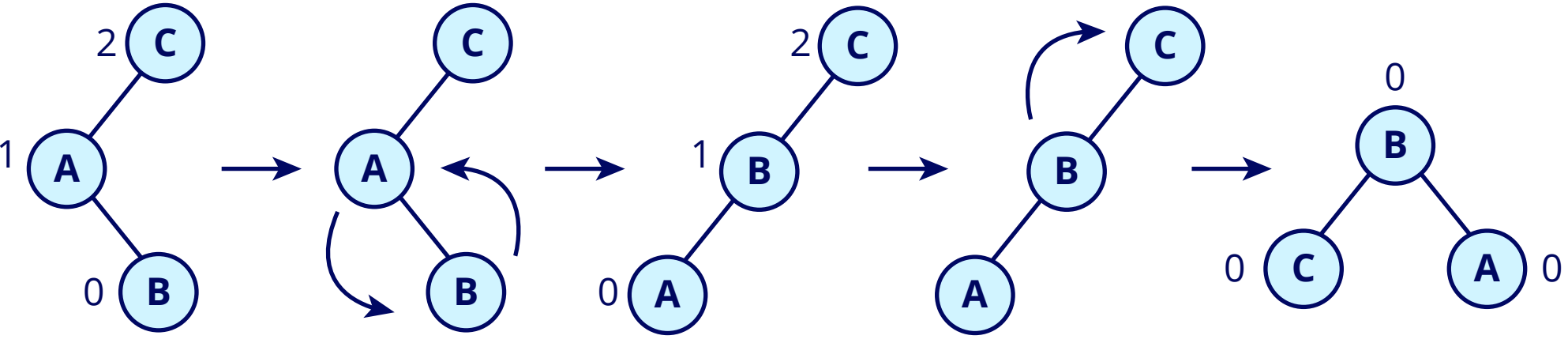
Balance factor:
This is the difference between any node’s left and right subtree heights.

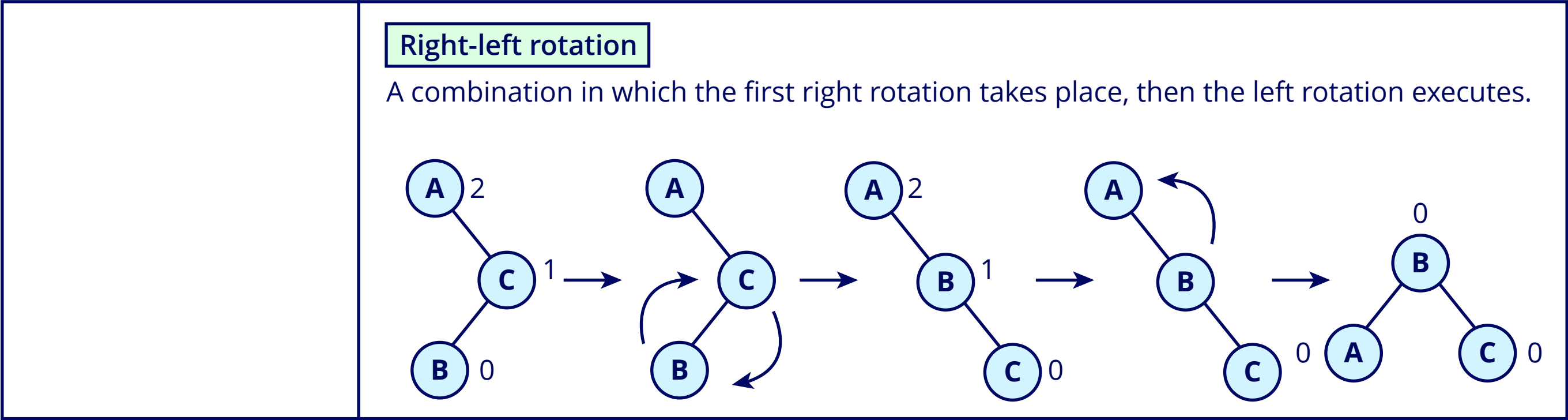
Operation	Example	Time Complexity (worst-case)
Insertion: A standard BST insertion followed by rebalancing using rotations if the balance factor is violated.	<div><div>Insert 8 in the tree</div><div></div></div>	$O(\log n)$

Operation	Example	Time Complexity (worst-case)
Deletion: It is a standard BST deletion without violating the balance factor.	<p>Delete Node 25 from the tree</p>  <p>After Rotation</p>	$O(\log n)$
Searching: Standard BST search operation.	<p>Search for 7</p> 	$O(\log n)$

Balance maintenance:

AVL trees use rotations (single and double) to maintain balance after insertions and deletions:

Single rotation	<div><div>Left Rotation</div><p>Rotates a right-heavy subtree to the left.</p><p>Right UnBalanced tree Left Rotation Balanced</p></div>
	<div><div>Right Rotation</div><p>Rotates a left-heavy subtree to the right.</p><p>Left UnBalanced tree Right Rotation Balanced</p></div>
Double rotations	<div><div>Left-right rotation</div><p>A combination in which the first left rotation takes place, then the right rotation executes.</p></div>



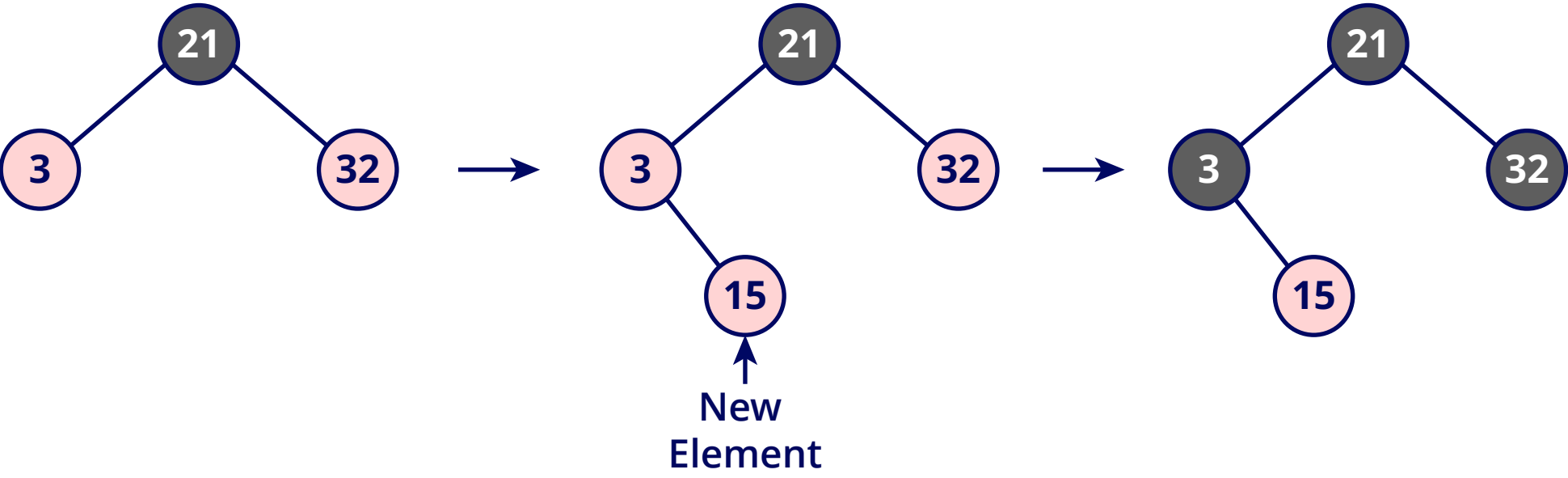
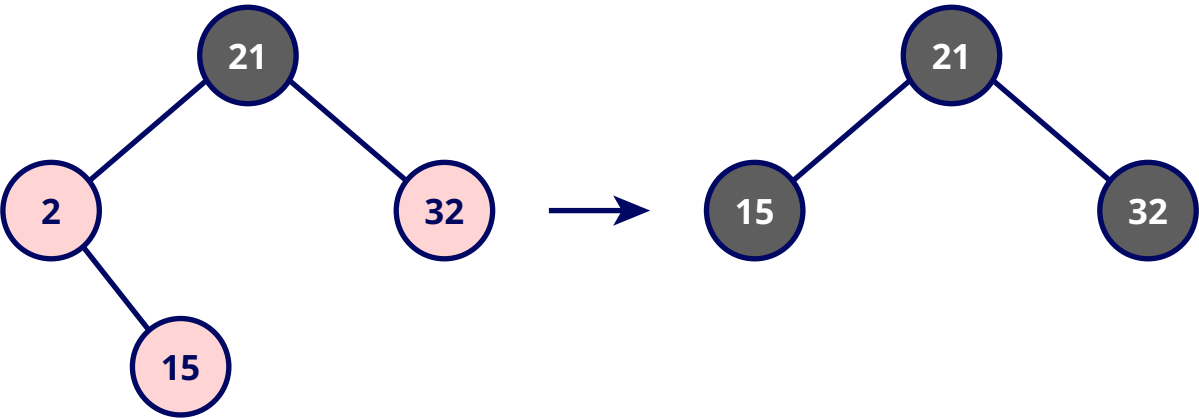
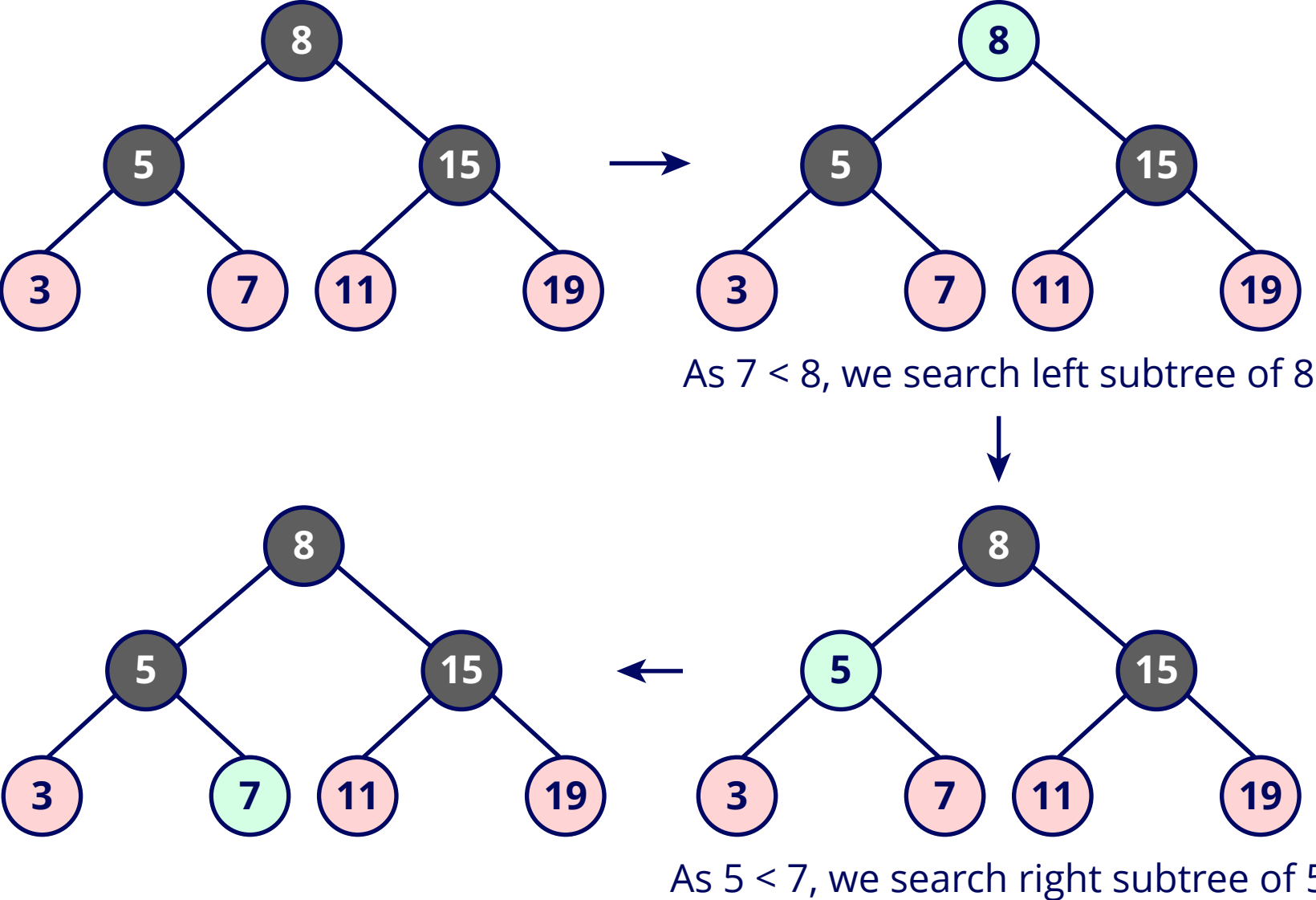
Red-black trees

Definition:

Red-black trees are balanced binary search trees where each node has an additional color property (red or black) to maintain balance.

Properties:

- **Red property:** Red nodes cannot have red children, ensuring no two consecutive red nodes appear on any path.
- **Black property:** Every path from a node to its descendant leaf nodes (null nodes) contains the same number of black nodes.

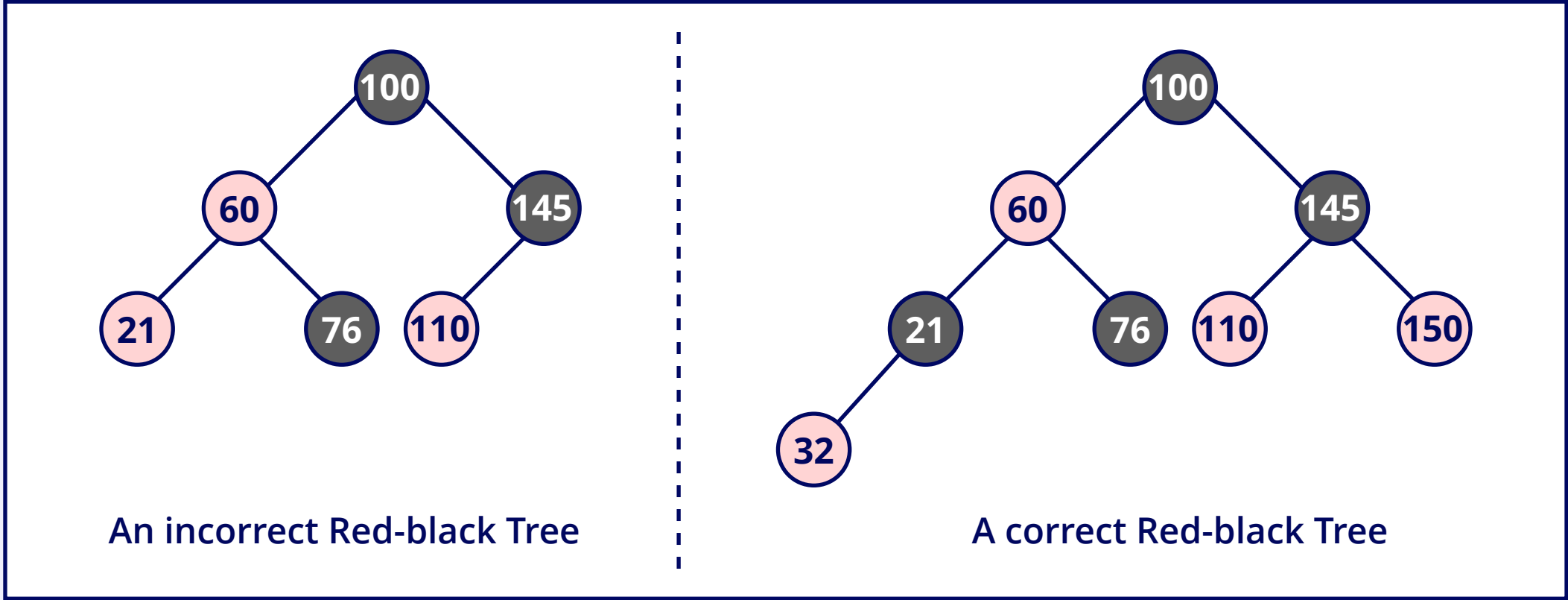
Operation	Example	Time Complexity (worst-case)
Insertion: It is standard BST insertion followed by recoloring and rotations to maintain balance.		$O(\log n)$
Deletion: Involves standard BST deletion; then, to maintain the balance, required recoloring and rotations are made.		$O(\log n)$
Searching: Standard BST search operation.	<p>Search for 7</p> 	$O(\log n)$

Balance maintenance:

The rules of a Red-black tree for balance maintenance:

- Each node is either red or black.
- The root is always black.
- Red nodes cannot have red children (no two consecutive red nodes on a path).
- Every path from a node to its descendant leaf nodes (null nodes) has the same number of black nodes.

Example of Red-black Tree:



Comparison:

2-3 Trees	AVL trees	Red-Black trees
Advantages: <ul style="list-style-type: none">• Strict balance ensures consistent performance. Also, it has simple insertion and deletion mechanisms.	Advantages: <ul style="list-style-type: none">• Provides strict balancing, ensuring logarithmic height.• Efficient for search-intensive applications.	Advantages: <ul style="list-style-type: none">• Balances more flexibly than AVL trees as they require fewer rotations.• Suitable for inserting/deleting heavy workloads.
Disadvantages: <ul style="list-style-type: none">• It has a more complex node structure than binary trees.	Disadvantages: <ul style="list-style-type: none">• Rotations can be complex and frequent, impacting performance.	Disadvantages: <ul style="list-style-type: none">• Less strictly balanced than AVL trees, potentially less efficient for search-heavy operations.
Use case: <ul style="list-style-type: none">• Database indexing (consistent search, insert, and delete operations)	Use case: <ul style="list-style-type: none">• Applications with a high frequency of search operations.	Use case: <ul style="list-style-type: none">• Real-time event processing (frequent insertions and deletions).

Applications:

The tree data structure is widely used to implement database indexing, compiler symbol tables, file systems, etc.

