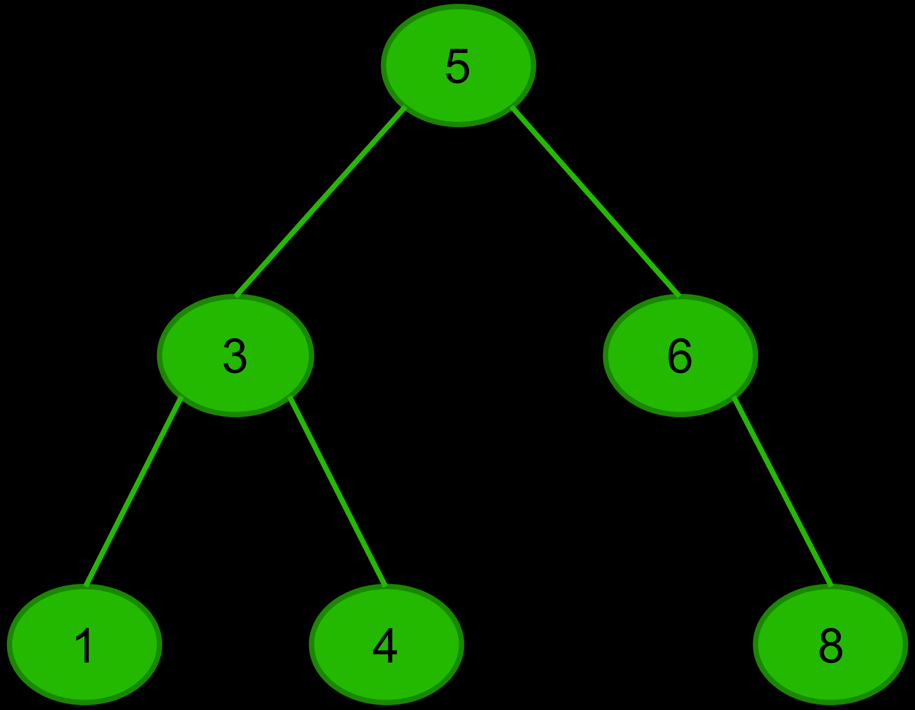
BM40A1500 DATA STRUCTURES AND ALGORITHMS

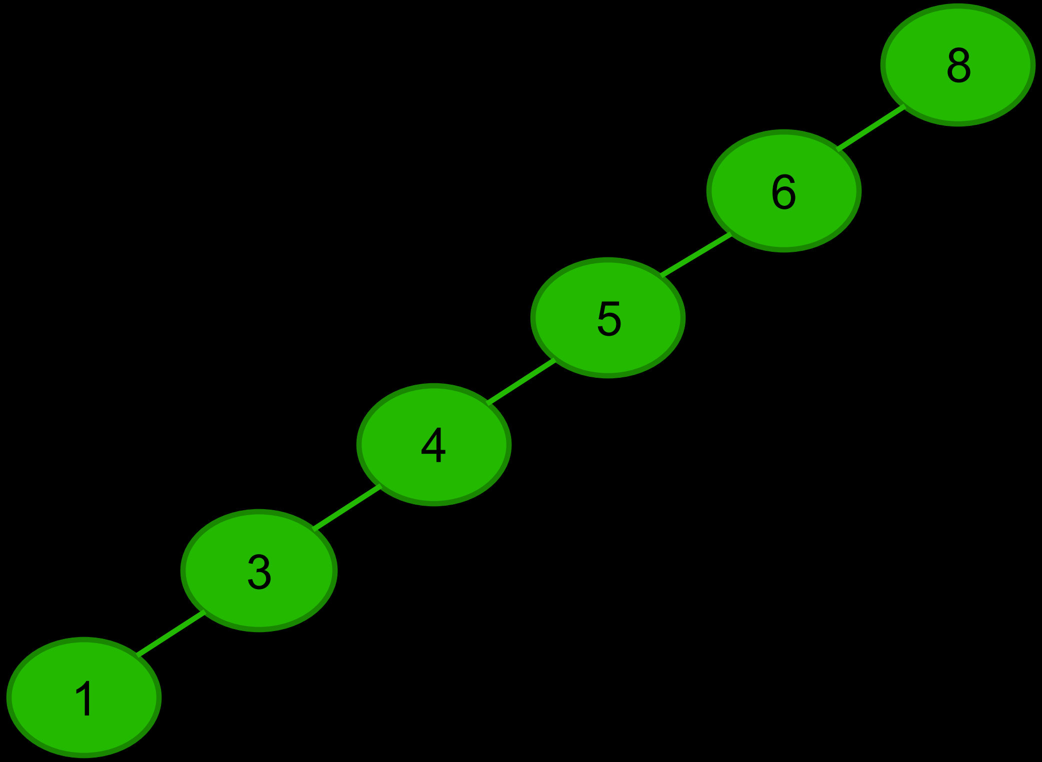
BALANCED TREES AND HEAPS

2024

THE PROBLEM WITH BINARY SEARCH TREES

- The search operation in BST is efficient only if the tree is balanced.
- Depending on the order in which the keys are added, the tree might end up unbalanced.

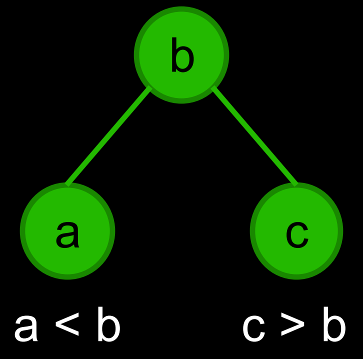


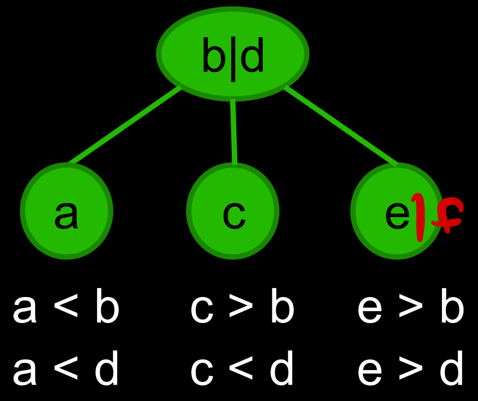


2-3 TREE

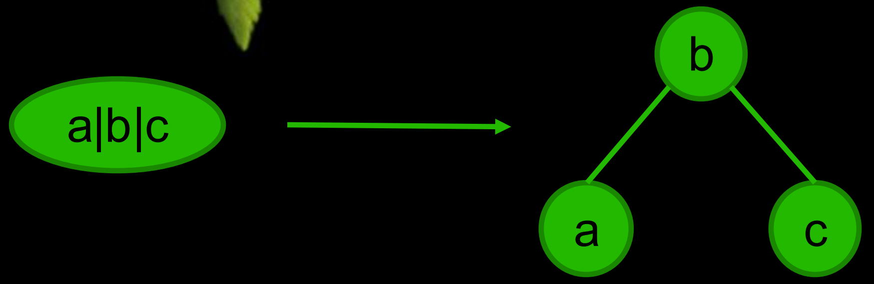
- A node contains one or two keys.
- Every internal node has either two children (if it contains one key) or three children (if it contains two keys).
  - Left subtree contains keys that are smaller than both keys in the root node.
  - Possible middle tree contains keys that are between the the two key values in the root node.
  - Right subtree contains keys that are larger than both keys in the root node.
- All leaves are at the same level in the tree

  → the tree is always height balanced.

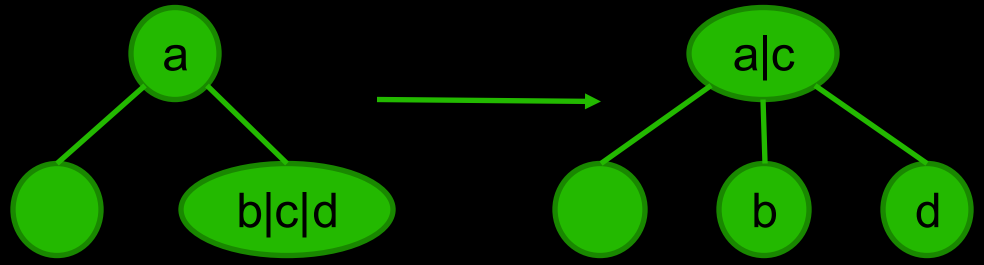


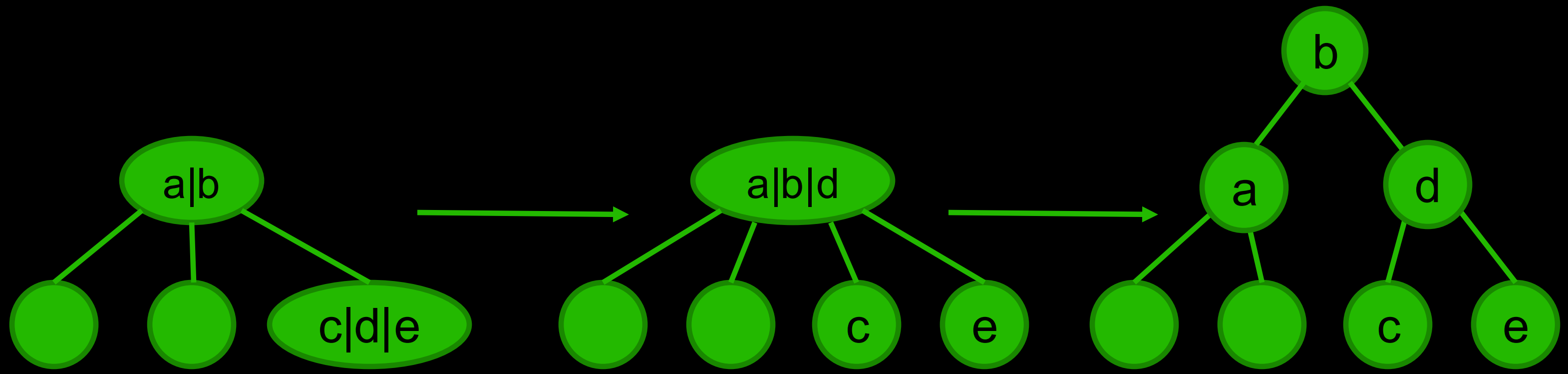


2-3 TREE



- If after insertion, a node contains three keys, transformations are done to satisfy the criteria.
- Split-and-promote process:
  - The middle key is “promoted” to the parent node and
  - The original node is split into two.
  - Repeated recursively if needed.

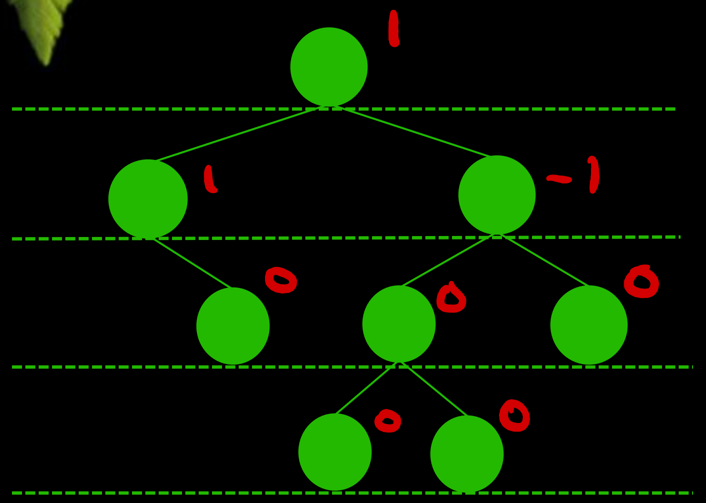




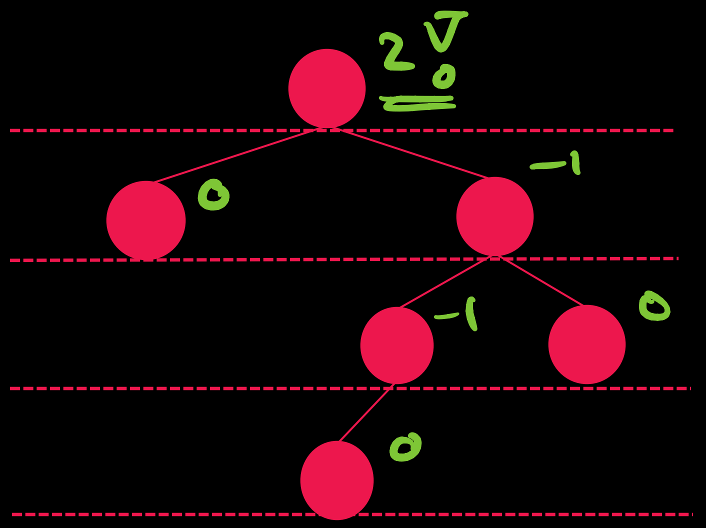
AVL TREE

- A self-balancing BST.
- Balance property: for every node, the heights of its left and right subtrees differ by at most 1.
- To maintain the balance property during insertion (and delete), rotations are done when needed:
  - Single rotation
  - Double rotation

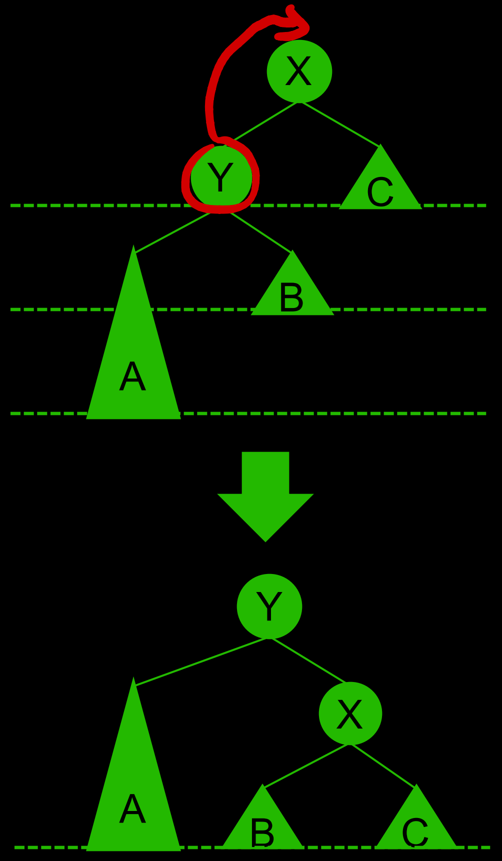
Balanced



Imbalanced

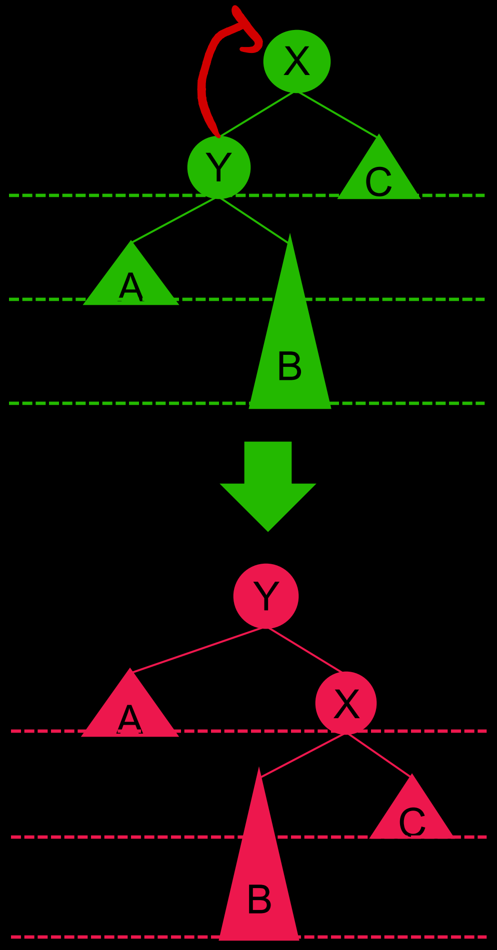


Case 1: a single rotation

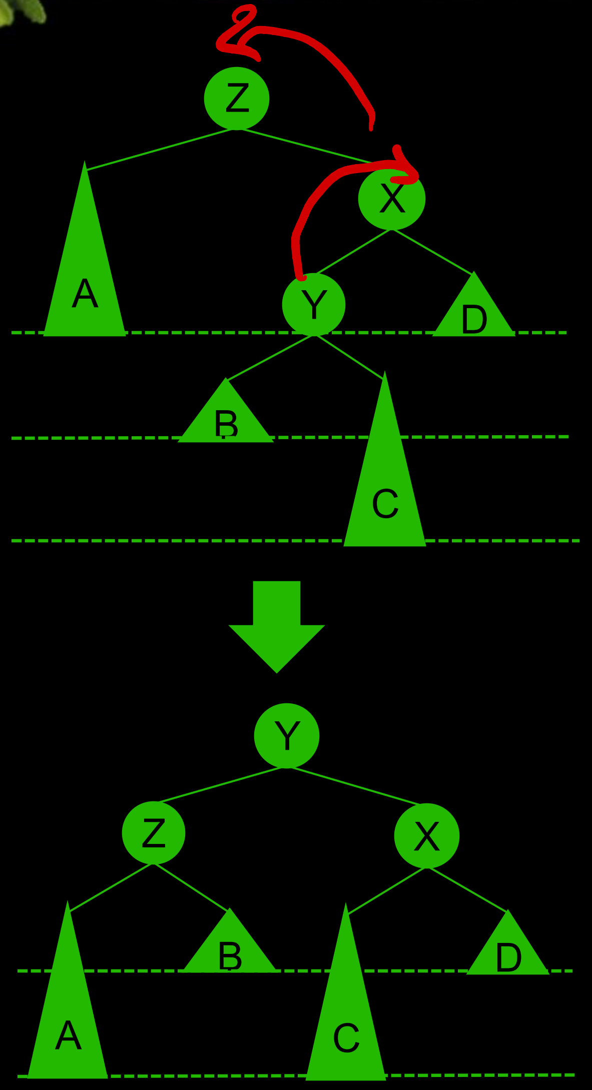


Symmetrical cases similarly

Case 2: a double rotation needed



Still imbalanced



IMPLEMENTING AVL TREE IN PYTHON

```python
class AVLNode:
    # Initialize new node
    def __init__(self, key):
        self.key = key
        self.left = self.right = None
        self.balance = 0


class AVL:
    # Initialize new tree
    def __init__(self) -> None:
        self.root = None
        self.is_balanced = True

    # Inserts a new key to the search tree
    def insert(self, key: int):
        self.root = self.insert_help(self.root, key)
```

```python
def insert_help(self, root, key):
    if not root:
        root = AVLNode(key)
        self.is_balanced = False
    elif key < root.key:
        root.left = self.insert_help(root.left, key)
        if not self.is_balanced:    # Check for possible rotations
            if root.balance >= 0:   # No Rotations needed
                self.is_balanced = root.balance == 1
                root.balance -= 1
            else:                   # Rotation(s) needed
                if root.left.balance == -1:
                    root = self.right_rotation(root)      # Single
                else:
                    root = self.left_right_rotation(root) # Double
                self.is_balanced = True
    elif key > root.key:
        root.right = self.insert_help(root.right, key)
        # ...

    return root
```

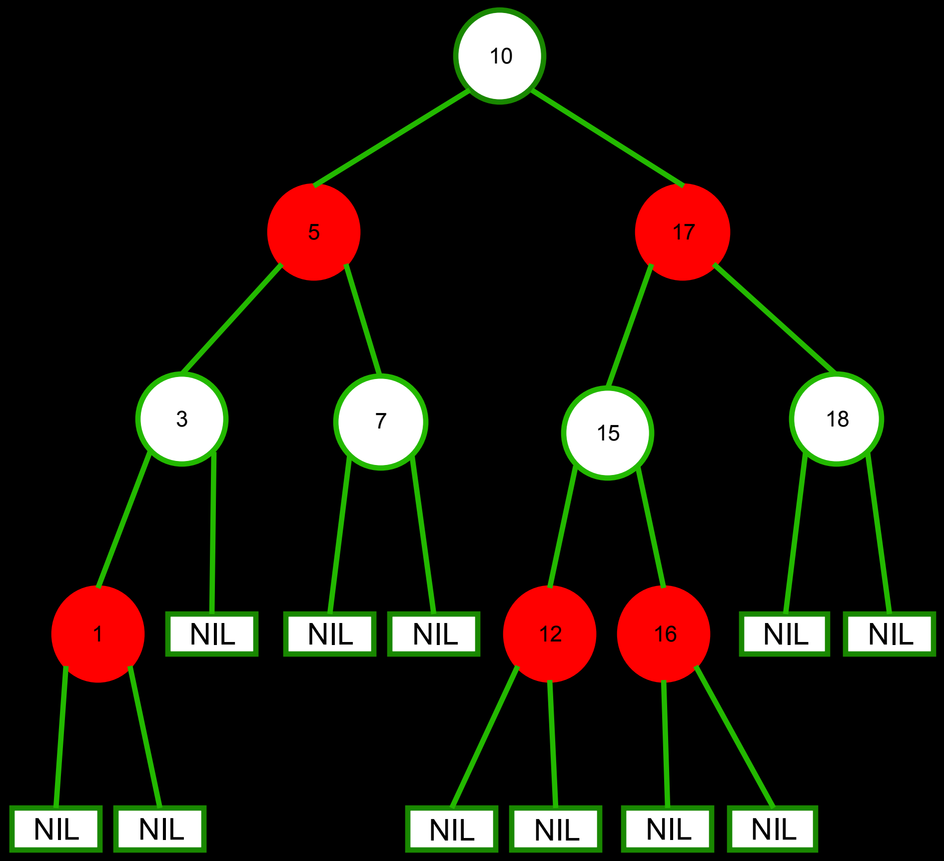
IMPLEMENTING AVL TREE IN PYTHON

```
# Single rotation: right rotation around root
def right_rotation(self, root):
    child = root.left                   # Set variable for child node
    root.left = child.right             # Rotate
    child.right = root
    child.balance = root.balance = 0    # Fix balance variables
    return child

# Double rotation: left rotation around child node followed by right rotation around root
def left_right_rotation(self, root: AVLNode):
    child = root.left
    grandchild = child.right            # Set variables for child node and grandchild node
    child.right = grandchild.left       # Rotate
    grandchild.left = child
    root.left = grandchild.right
    grandchild.right = root
    root.balance = child.balance = 0    # Fix balance variables
    if grandchild.balance == -1:
        root.balance = 1
    elif grandchild.balance == 1:
        child.balance = -1
    grandchild.balance = 0
    return grandchild
```

RED-BLACK TREE

- Each node stores the color: red or black
  - used to ensure that the tree stays balanced
- Requirements:
  - Each node is either red or black
  - Empty sub trees (NIL) are black.
  - Red nodes do not have red child nodes.
  - Each path from a root to a leaf node contains the same number of black nodes.
  - The root node is black.
- To maintain the requirements during the insert and remove operations, color changes and rotations are done if necessary.

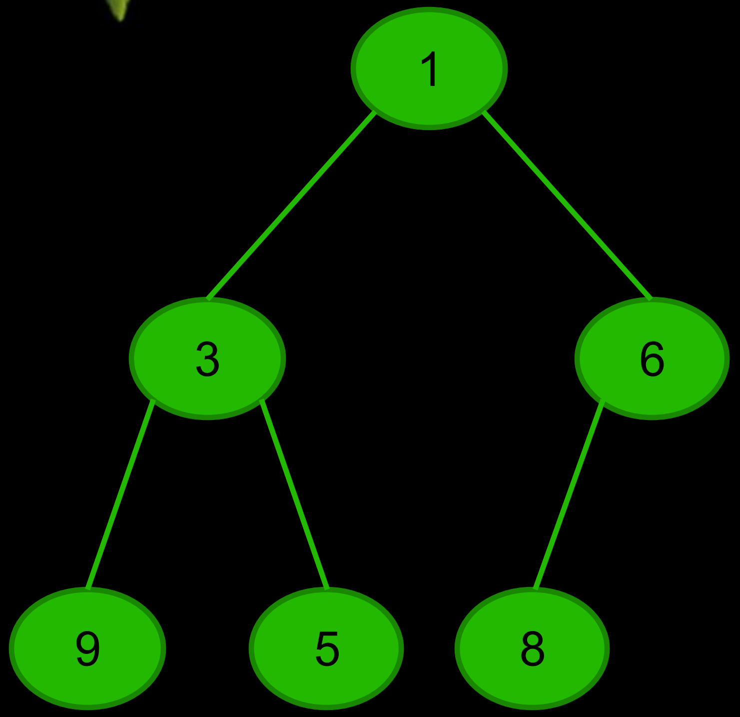


COMPARISON TO HASH TABLES

- Hash tables:
  - The basic operations (insert, remove, search) are very fast when the hash table is properly implemented.
    - Average case: Θ(1)
    - Worst case: Θ(n)
  - Easier to implement
  - Suitable only for exact-match queries (e.g., searching based on a key value)
- Balanced trees
  - The basic operations are slower than in hash tables
    - Average case: Θ(log n)
    - Worst case: Θ(log n)
  - Keep data sorted → allow more complex queries
    - Range queries (finding all records with key value between A and B)
    - Sequential access
  - Dynamic tree structures are more memory efficient (a hash table needs to be large to be efficient)

HEAPS

- Simplified version of binary tree.
- Heap condition:
  - Max heap: every node stores a value that is greater than or equal to the value of either of its children.
  - Min heap: every node stores a value that is less than or equal to that of its children.
- Efficient to access and remove the smallest (or largest) key and, to add keys.
  - Commonly used as an implementation for priority queue.
- Can be implemented as an array:
  - Tree structure is always complete.



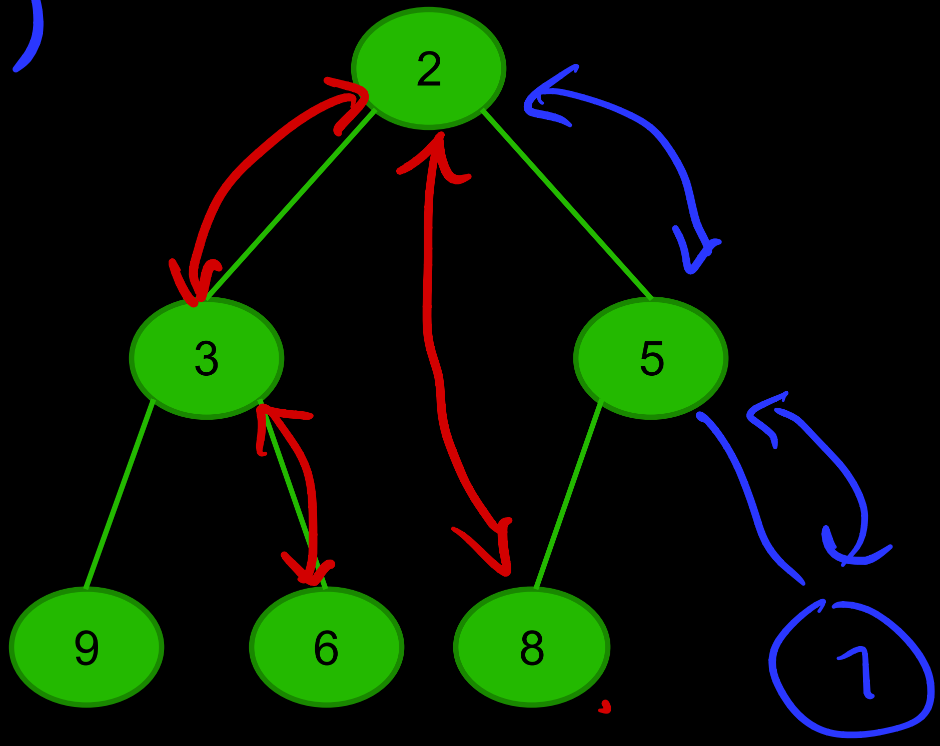
1	3	6	9	5	8

HEAPS

$\Theta(\log n)$

- Insert:
  - Insert the new value as a leaf by keeping the tree structure complete.
    - Put the new value at the end of the array.
  - Move the value upwards on tree until the heap condition is satisfied.
- Remove
  - Swap the root and last leaf.
    - Swap the first and last position in the array.
  - Remove the last leaf.
  - Push the top value down the tree until the heap condition is satisfied.

$\Theta(\log n)$



HEAPSORT

- A heap can be used for sorting a list:
  - Building a min heap.
  - Pop/remove all the elements one-by-one.
- Instead of inserting values one-by-one, a heap can be built from an arbitrary array very efficiently:
  - Working from bottom to top, from right to left, do a shiftdown on each internal node.
  - $\Theta(n)$ (see the proof in OpenDSA material)
- Note: when using the array implementation, the max heap is more convenient as no additional array is needed.


LUT
University