Binary Search

- List must be sorted before
- Look through by cutting list in half every Tim

Linear Search

- List does not need to be sorted
- Look through like looking for word in dictionary, one by one

Arrays (Sorted by specialVal)

- **Search (Fast):** You can use **binary search** (like searching a word in a dictionary) to quickly find a value. This takes about **O(log n)** time.
- **Insert (Slow):** Adding new records is slow because the array needs to stay sorted, requiring you to shift other records. This takes about **O(n)** time
- **Real-World Example:** Think of an alphabetically sorted class roster. Finding a student's name is fast, but adding a new name in the right spot takes more time
- Ex. [2, 4, 5, 7, 10]

Linked Lists (Sorted by specialVal)

- **Search (Slow):** You have to go one record at a time to find what you're looking for (**O(n)** time).
- **Insert (Fast):** Adding a new record is easy because you just adjust the pointers (**O(1)** or **O(n)** time).
- **Real-World Example:** Think of a sign-up sheet where names are added as people arrive. Adding is easy, but finding someone's name takes time
- Ex. 2 → 4 → 7 → 10

Arrays vs linked lists

- Use arrays for **fast searching** and linked lists for **fast inserting**.

Binary Search Tree (BST) combines the best parts of arrays and linked lists

- **Search (Fast):** You don't have to scan everything; you follow branches like a decision tree (**O(log n)** time).
- **Insert (Balanced):** Adding new records is also efficient because you just add them in the right spot in the tree (**O(log n)** time
- **How It Works:**
  - Start at the top of the tree.
  - If the value you're looking for is smaller, go left.
  - If it's larger, go right

BST Property

- For every node x in a Binary Search Tree:
  - All nodes in the **left subtree** of x have **keys smaller** than the key at x.
  - All nodes in the **right subtree** of x have **keys greater** than the key at x

- Ex.
  - Imagine you're organizing books on a shelf:
    - The middle book is your current node (x).
      - All books **to the left** are alphabetically earlier (smaller keys).
      - All books **to the right** are alphabetically later (greater keys)
- Each **node** in a BST has:
  - **p** – A pointer to its **parent node**.
  - **left** – A pointer to its **left child** node.
  - **right** – A pointer to its **right child** node.
  - **key** – The **value** or **data** stored in the node

Traversal in nodes of BST

- Traversal strategies differ by the ordering of the objects to visit
- Goal of traversal is to visit every node (search is to find a specific node)
- Inorder
  - Left, current, right
- Preorder
  - Current, left, right
- Postorder
  - Left, right, current
- Inorder:
  - 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 15, 19, 20
- Preorder
  - 7, 4, 2, 3, 6, 5, 12, 9, 8, 11, 19, 15, 20
- Postorder
  - 3, 2, 5, 6, 4, 8, 11, 9, 15, 20, 19, 12, 7

Searching

- **If the key matches:** 🎯 You're done! The search is successful.
- **If the key is smaller:** 🔍 Look in the **left subtree** because all smaller keys are on the left.
- **If the key is larger:** 🔍 Look in the **right subtree** because all larger keys are on the right

Deletion
3 possible scenarios:

1. **Node has no children (Leaf Node):** Simply remove the node.
2. **Node has one child:** Remove the node and replace it with its child.
3. **Node has two children:** Replace the node with either:
   - The **inorder successor** (smallest node in the right subtree), or
   - The **inorder predecessor** (largest node in the left subtree)

Balancing

- AVL Trees ensure that the **balance factor** (difference between heights of the left and right subtrees of any node) is always **-1, 0, or 1**.
- Left left and right right: single rotation
- Left right and right left: double rotation

- A rotates clockwise around B
- Everything else rearranges based off of B being new parent…
    - T2.L gets assigned to A, not B because its smaller than B and therefore must be on the left side of it (its on left side originally)

B+ Trees
It ensures that all leaf nodes are at the **same level** and maintains a **high branching factor**, meaning each node can have many children
Properties

- **Root Node:** Has at least 2 children (unless it's a leaf).
- **Internal Nodes:** Have between ⌈m/2⌉ and *m* children (*m* is the tree order).
- **Leaf Nodes:** All leaf nodes are at the same leve

How B-Tree Works:

- **Search:** Perform a **binary search** within a node, and follow the appropriate child pointer.
- **Insertion:** Insert data into a node. If it overflows, **split the node** and promote the middle key to the parent.
- **Deletion:** Remove data, potentially merging nodes if they fall below the utilization threshold

B+ Tree

- R**efined version of the B-Tree**, designed to improve **range queries** and make disk operations even more efficient.

B- VS B+ Tree
B-Tree: Think of it like an **index in a book**, where data (content) is sprinkled throughout the pages (nodes).
B+ Tree: Think of it as **an index at the start** (internal nodes) that points directly to content chapters (leaf nodes), and the chapters are all **linked together for quick scanning**

- Internal nodes in a B+ Tree act like **a directory for data in leaf nodes**, improving **lookup and traversal speeds**.

Inserting into B+ Tree
1. Find the Correct Leaf Node

- Start at the **root node** and follow the correct child pointers **based on the key** being inserted.
- Repeat until you reach the **leaf node** where the new key belongs.

2. Insert the Key into the Leaf Node

- If there's **space** in the leaf node:
    - Add the key in **sorted order**.
- If the leaf node is **full**:
    - **Split the leaf node** into two nodes.
    - The **middle key** is promoted to the **parent node** as a separator.

3. Handle Parent Node Overflow (If Any)

- If the **parent node becomes full** after promoting the middle key:
    - Split the parent node and promote its middle key further **up the tree**.
- Repeat this process **recursively** until no node overflows or until the **root node is split**.

4. Update Links Between Leaf Nodes

- Ensure the **leaf nodes remain doubly linked** after the split for efficient **range traversal**