FIT5196 DATA WRANGLING

Week 6

Data Structuring

By Jackie Rong

Faculty of Information Technology

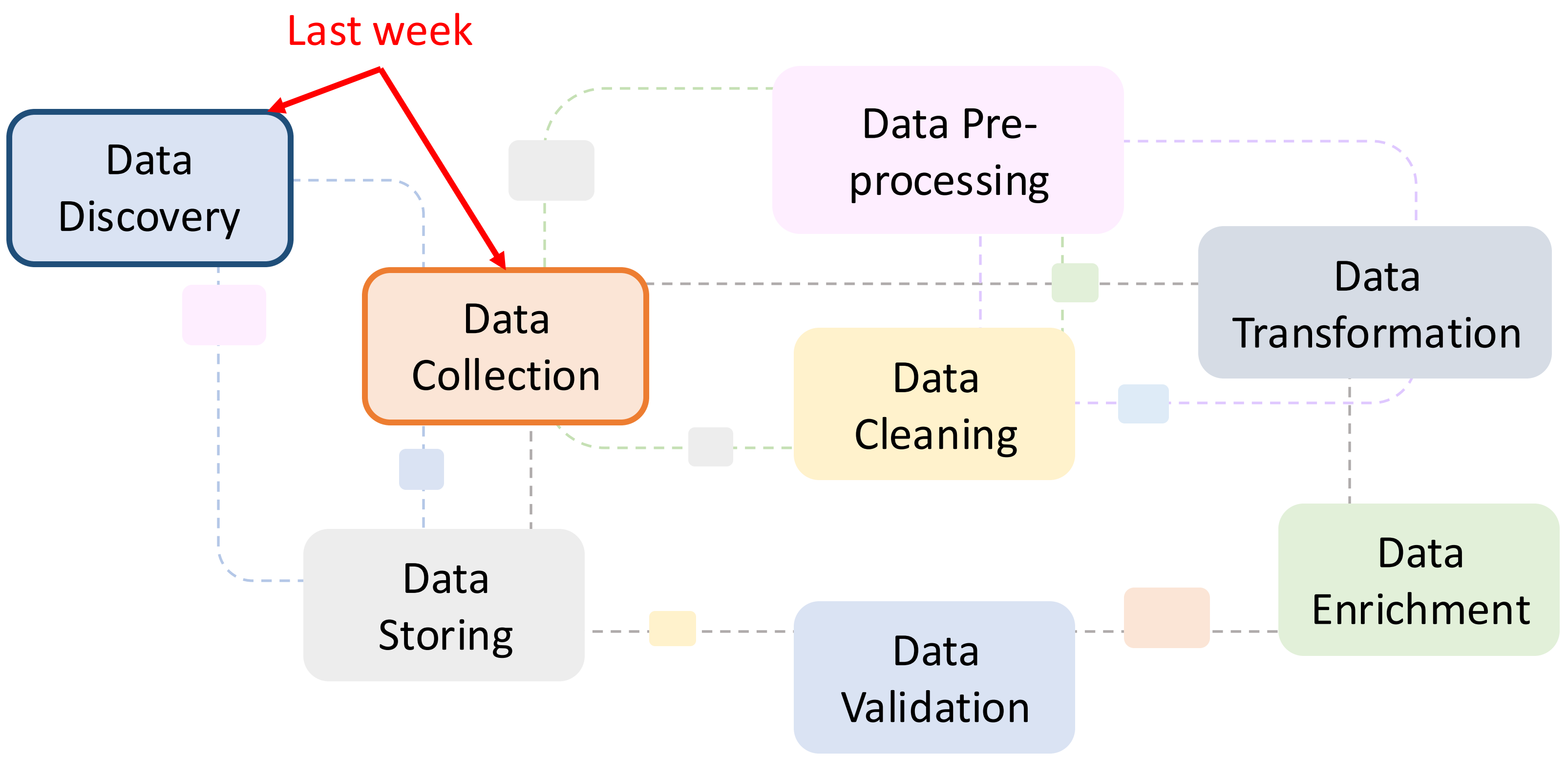
Monash University

Data Wrangling Tasks (Recap)

- **Data Wrangling** is the process of acquiring, cleaning, structuring, and enriching raw data into a format that is directly usable for analysis.

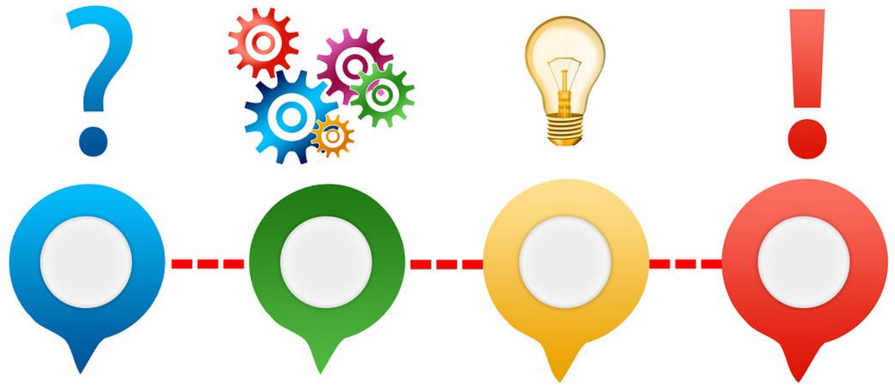
Data discovery is the process of identifying and understanding data sources that can be used for analytical purposes.

Data collection is the systematic process of gathering and measuring information on variables of interest, in an established systematic fashion that enables one to answer stated research questions, test hypotheses, and evaluate outcomes.



Data Structuring

- Overview of Data Structuring
- Primitive Data Types
- Non-Primitive Data Types
- Complex Data Structures
- Key Operations



MONASH University

Data Structuring

- **Data structuring** is a critical aspect of managing and organizing data in a way that it can be efficiently accessed and manipulated.
- This process involves setting up data in a systematic, organized format that supports various operations such as retrieval, update, and management.
- Data structuring involves the creation and use of data structures, which are formats that store and organize data.
- The **main goal** is to enable data to be processed in an efficient manner, enhancing both speed and accessibility while minimizing resource usage.

Importance in Data Wrangling

- In the context of data wrangling—which involves transforming and mapping raw data into a more useful format—data structuring is fundamental. It affects how easily data can be cleaned, transformed, and analysed.
  - **Efficiency**: Proper data structuring can greatly improve the efficiency of data wrangling processes, reducing the computational time required for data manipulation.
  - **Data Integrity**: Well-structured data helps maintain accuracy and consistency, which are vital for reliable analysis.
  - **Scalability**: Effective data structures can handle increases in data volume without a significant loss of performance, making them essential for big data applications.

Types of Data Structures

- Data structures are typically divided into two main types: **primitive** and **non-primitive**.
  - **Primitive Data Structures**
    - These include basic types like integers, floats, Booleans, and characters. They form the building blocks for more complex structures.
  - **Non-Primitive Data Structures**:
    - **Linear Data Structures**: Such as arrays, lists, stacks, and queues, where data elements are arranged in a sequential order.
    - **Non-Linear Data Structures**: Such as trees and graphs, where data elements maintain a hierarchical or networked relationship.

Primitive Data Types

	Integer	Floating Point
Description	Represents whole numbers, both positive and negative	Used to represent real numbers that include a fractional component, modelled in a format similar to scientific notation
Usage	Used in operations that require counting, indexing, or any form of integer arithmetic	Essential for representing and conducting calculations with decimal or large numbers, especially in scientific computations
Storage	Typically stored in 16, 32, or 64 bits, depending on the language and the system architecture	Usually available in formats like float and double, with float often being a 32-bit number and double a 64-bit number, providing greater precision

Primitive Data Types

	Character	Boolean
Description	Represents a single character from a character set such as ASCII or Unicode	Represents truth values, typically **true** and **false**
Usage	Used to store characters for text processing, such as letters, numbers, punctuation marks, and other symbols	Crucial in control structures (like if-statements and loops) and anywhere binary choices are mapped, such as toggling states, decision-making in algorithms, and more
Storage	Typically stored in 8 bits (1 byte) for ASCII characters, while Unicode characters might use 16 or 32 bits depending on the encoding (UTF-16, UTF-32)	Usually stored in a single bit, but often represented as 1 byte in practice due to the structure of memory in most computers

Primitive Data Types

- **Primitive types** are type-safe, meaning operations performed on them are checked by the compiler for type compatibility, reducing certain types of runtime errors.
- **Operations** using primitive types are generally faster than those involving complex data structures because they are directly supported by the underlying hardware.
- Since primitive types occupy minimal space and are fixed in size, they are highly efficient in terms of memory usage, making them suitable for high-performance computing tasks.

Non-Primitive Data Types

	Array	String
Description	An array is a collection of elements, typically of the same type, stored in contiguous memory locations.	Strings are sequences of characters used to store text.
Usage	Arrays are used for storing multiple values in a single variable and are useful for operations that require indexing, such as iterating through elements or quick access via an index.	Strings are essential for any program that takes textual input, displays text, or manipulates text. Functions to search, slice, replace, or modify text are commonly used in string handling.
Storage	The size of an array is often fixed at the time of declaration, although some languages support dynamic arrays or similar structures (e.g., ArrayList in Java).	Internally, strings may be represented as arrays of characters.

Non-Primitive Data Types

	List (or Linked List)	Queue
Description	Lists are collections of elements that are not necessarily stored contiguously; elements are linked using pointers.	A queue is a collection that follows the First In, First Out (FIFO) principle.
Usage	Lists are ideal for applications where the amount of data varies dynamically, and frequent insertion and deletion operations are required.	Queues are used in scenarios where order needs to be preserved, such as in print queues, task scheduling, and breadth-first search algorithms.
Storage	Each element (or node) in a list typically stores its data and a reference (or pointer) to the next item in the list.	Can be implemented using arrays or linked lists, though specialized types like circular queues are also common.

Non-Primitive Data Types

	Dictionary (or Map)
Description	Dictionaries store pairs of keys and values, with each key being unique within the dictionary.
Usage	Dictionaries are essential for cases where data needs to be retrieved quickly by a specific key. They are commonly used for lookups, caching data, and implementing associative arrays.
Storage	Often implemented using hash tables, allowing for efficient retrieval.

- These non-primitive data types provide the necessary tools for managing complex data in an organized, efficient, and scalable manner.
- They form the backbone of data structures and algorithms, essential for advanced programming and application development.

MONASH University

Complex Data Structures

- **Complex data structures** are sophisticated constructs that allow for the organization and management of large and complicated datasets in a way that facilitates efficient processing and retrieval.
- They are generally built using primitive and other non-primitive data types but are designed to address specific problems or optimize certain types of operations.
- These complex data structures are fundamental in computer science, enabling efficient handling of dynamic and large datasets, supporting various operations like search, insert, delete, and update with optimal performance.
- They are crucial in areas such as database systems, operating systems, and networking, where large amounts of data must be managed efficiently.

Complex Data Structures

- Common complex data structures involve
  - Graphs
  - Trees
  - Hash Tables
  - Heaps

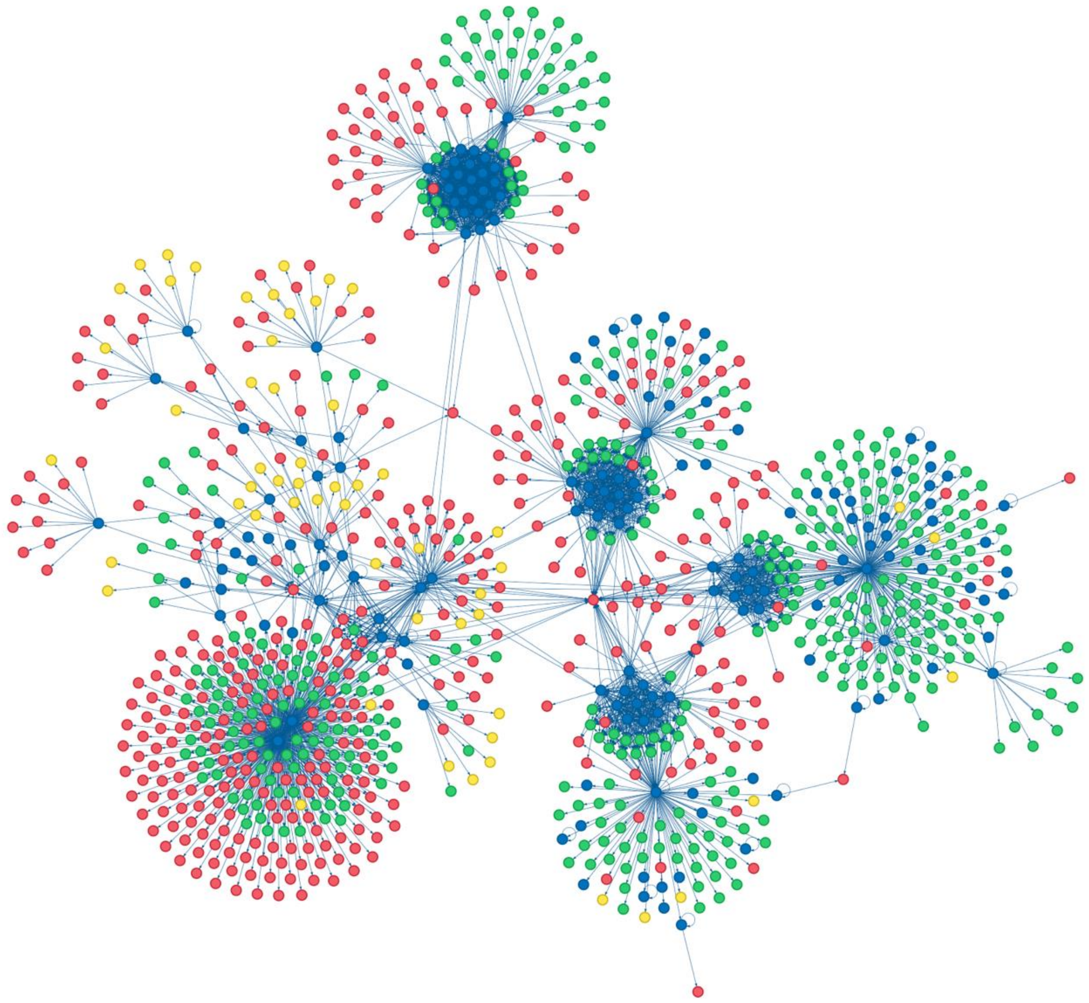
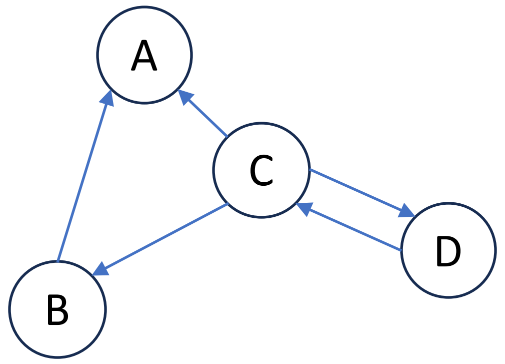


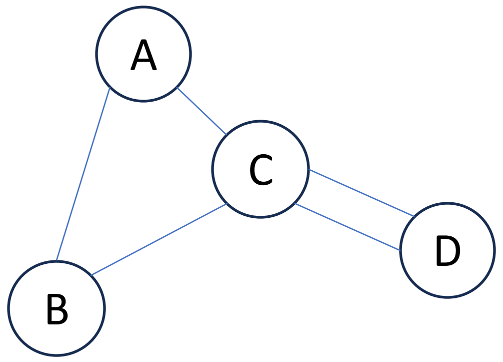
figure from: https://www.cs.cornell.edu/~kt/post/site-graph/

Graphs

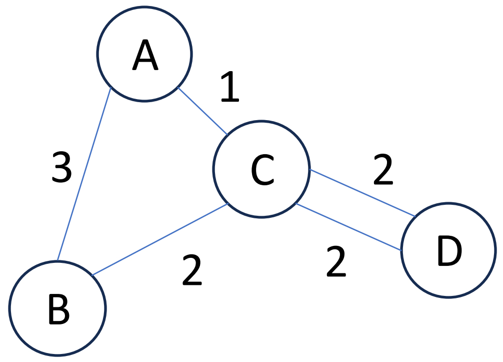
- A **graph** is a set of nodes (or vertices) connected by edges.
- It can be used to represent virtually any network where some items are connected to others, such as social networks, transportation networks, and communication networks.
  - **Directed Graphs**: Where edges have a direction.
  - **Undirected Graphs**: Where edges are bidirectional.
  - **Weighted Graphs**: Where edges carry weights representing cost or distance.



Directed Graph



Undirected Graph



Weighted Graph

Trees

- **Trees** are a type of graph with a hierarchical structure, consisting of a root node and child nodes, with no cycles.
  - **Binary Trees**
  - **Binary Search Trees (BST)**
  - **B-Trees**

Trees

- **AVL Trees**: An AVL tree is a self-balancing binary search tree where the heights of two child subtrees of any node differ by at most one.
- **Red-Black Tree**: Similar to AVL trees, Red-Black trees are self-balancing binary search trees, which are used in environments where the tree is modified frequently
- **Suffix Trees**: A suffix tree is a compressed tree containing all the suffixes of the given text as their keys and positions in the text as their values.
- **Segment Trees**: Segment trees are mainly used for answering range queries in logarithmic time and are very efficient for scenarios where array elements are not static.
- **Trie (Prefix Trees)**: A trie is used to store a dynamic set of strings where the keys are usually strings. It is particularly efficient for solving problems like word search, auto-complete, and prefix matching in texts.
- **Quad-Trees**: Each node represents a specific quadrant, and its four children represent sub-quadrants.
- **k-d Trees**: A space-partitioning tree used to organize points in a k-dimensional space.

Binary Trees

- **Binary Trees**: A binary tree where each node has up to two children, often used to model simple hierarchical data.
  - **Full Binary Tree** is a Binary Tree in which every node has 0 or 2 children.
  - **Complete Binary Tree** has all levels completely filled with nodes except the last level and in the last level, all the nodes are as left side as possible.
  - **Degenerate Binary Tree** is a Binary Tree where every parent node has only one child node.
  - **Perfect Binary Tree** is a Binary Tree in which all internal nodes have 2 children, and all the leaf nodes are at the same depth or same level.
  - **Balanced Binary Tree** is a Binary tree in which height of the left and the right sub-trees of every node may differ by at most 1.

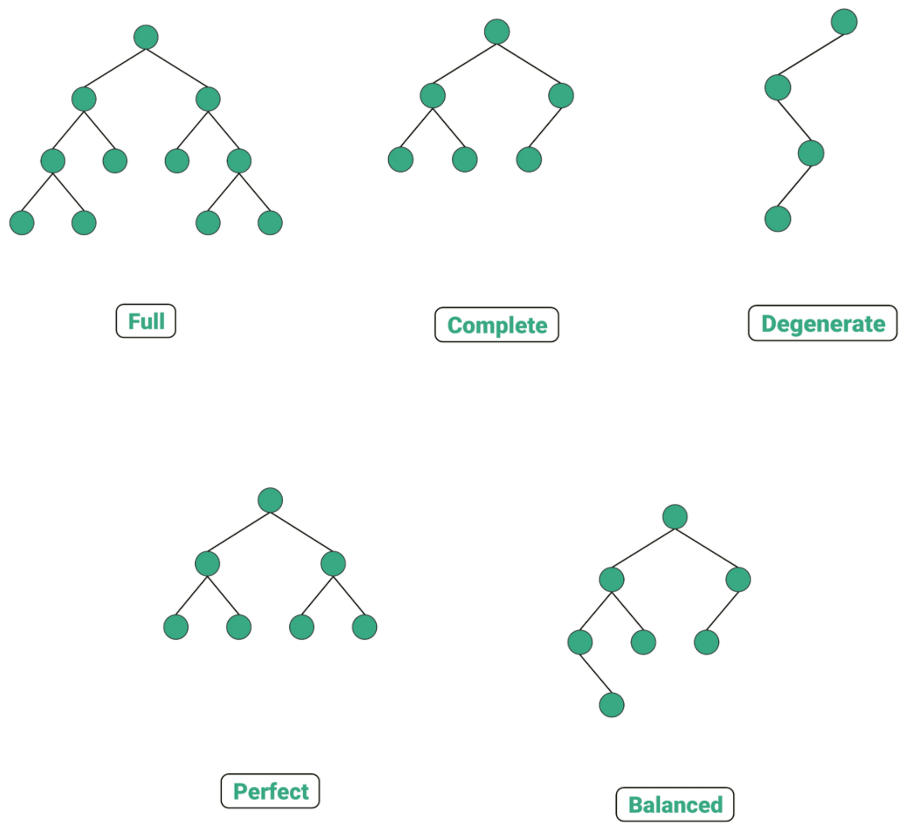
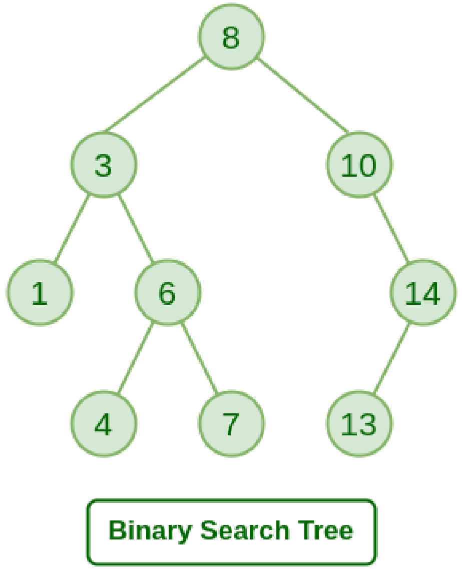


Figure from: https://towardsdatascience.com/5-types-of-binary-tree-with-cool-illustrations-9b335c430254

Binary Search Trees

- **Binary Search Trees (BST)**: A binary tree where each node has a key greater than all the keys in its left subtree and less than those in its right subtree.
- A binary search tree is used for efficient data searching. Each node stores a key greater than all the keys in its left subtree and less than or equal to all in its right subtree.



- Each node in a Binary Search Tree has at most two children, a left child and a right child
  - the left child containing values less than the parent node and
  - the right child containing values greater than the parent node.
- This hierarchical structure allows for efficient searching, insertion, and deletion operations on the data stored in the tree.

Figure from: https://www.geeksforgeeks.org/binary-search-tree-data-structure/

Binary Search Trees

- Applications of Binary Search Tree:
  - **Searching**: Finding a specific element in a sorted collection
  - **Sorting**: Sorting a collection of elements in ascending or descending order
  - **Range queries**: Finding elements within a specified range
  - **Data storage**: Storing and retrieving data in a hierarchical manner
  - **Databases**: Indexing data for efficient retrieval
  - **Computer graphics**: Representing spatial data in a tree structure
  - **Artificial intelligence**: Decision trees and rule-based systems

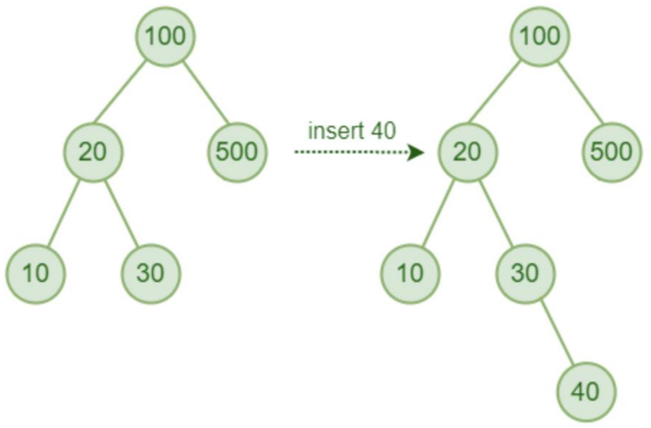
Binary Search Trees

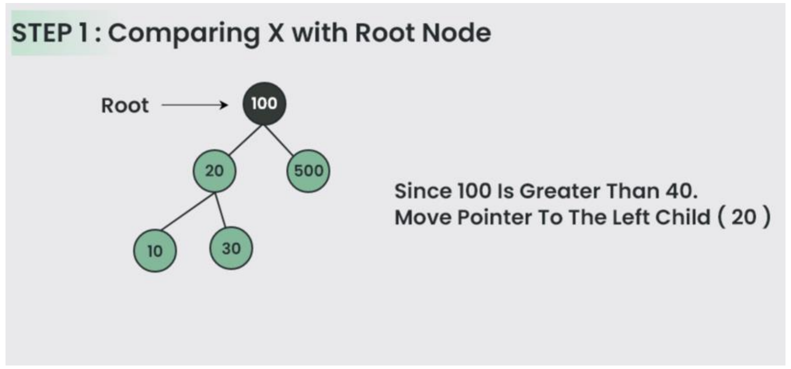
- **Advantages** of Binary Search Tree:
  - **Efficient searching**: O(log n) time complexity for searching
  - **Ordered structure**: Elements are stored in sorted order, making it easy to find the next or previous element
  - **Dynamic insertion and deletion**: Elements can be added or removed efficiently
  - **Balanced structure**: Balanced BSTs maintain a logarithmic height, ensuring efficient operations
  - **Space efficiency**: BSTs store only the key values, making them space-efficient
- **Disadvantages** of Binary Search Tree:
  - **Not self-balancing**: Unbalanced BSTs can lead to poor performance
  - **Worst-case time complexity**: In the worst case, BSTs can have a linear time complexity for searching and insertion
  - **Memory overhead**: BSTs require additional memory to store pointers to child nodes
  - **Not suitable for large datasets**: BSTs can become inefficient for very large datasets
  - **Limited functionality**: BSTs only support searching, insertion, and deletion operations

Source: https://www.geeksforgeeks.org/applications-advantages-and-disadvantages-of-binary-search-tree/

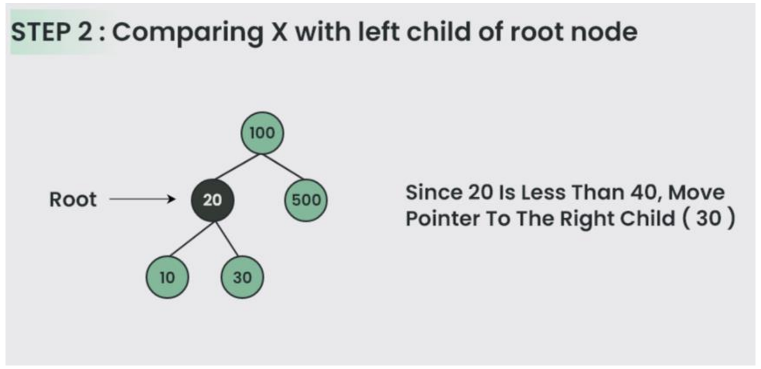
Binary Search Trees

- Basic operations on BST
  - **Insertion**

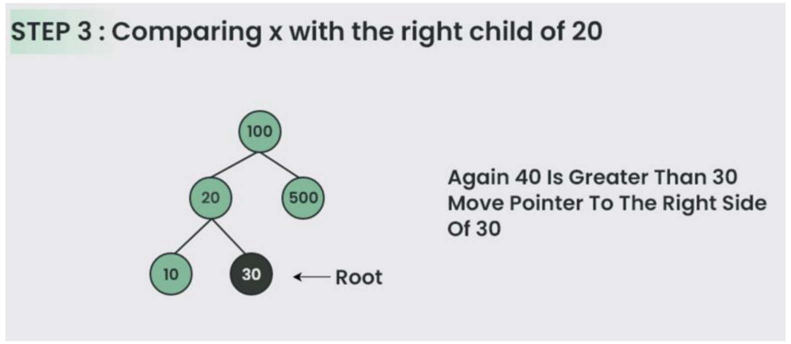




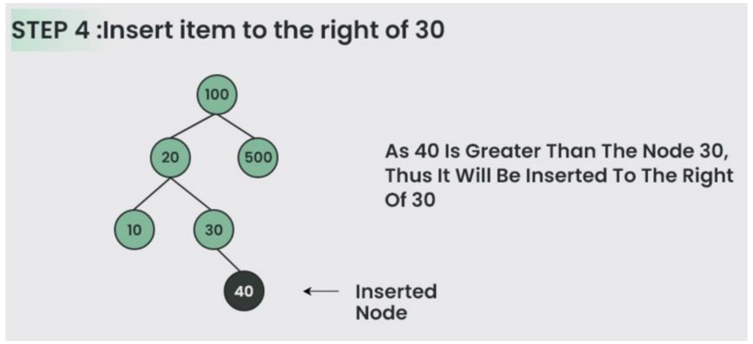
STEP 1 : Comparing X with Root Node
Root
100
20
500
10
30
Since 100 Is Greater Than 40.
Move Pointer To The Left Child (20)



STEP 2 : Comparing X with left child of root node
100
Root
20
500
10
30
Since 20 Is Less Than 40, Move
Pointer To The Right Child (30)



STEP 3 : Comparing x with the right child of 20
100
20
500
10
30
Root
Again 40 Is Greater Than 30
Move Pointer To The Right Side
Of 30

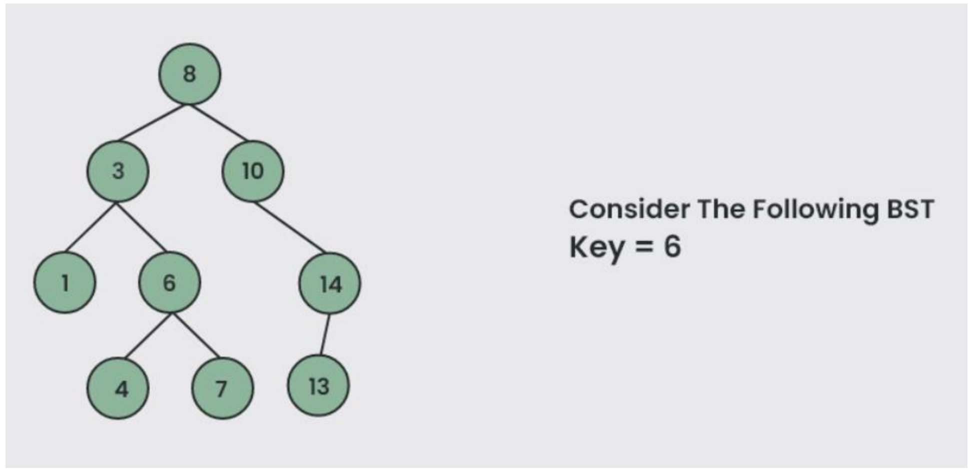


STEP 4 :Insert item to the right of 30
100
20
500
10
30
40
Inserted
Node
As 40 Is Greater Than The Node 30,
Thus It Will Be Inserted To The Right
Of 30

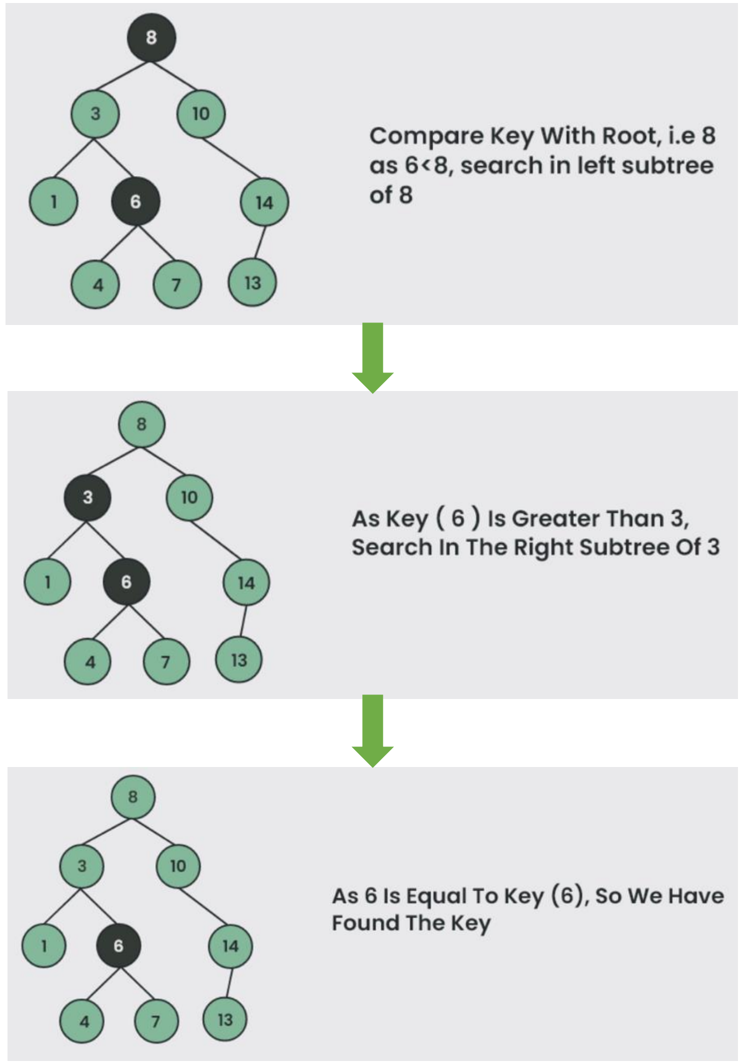
Figure from: https://www.geeksforgeeks.org/insertion-in-binary-search-tree/

Binary Search Trees

- Basic operations on BST
  - Insertion
  - **Searching**



Consider The Following BST
Key = 6



Compare Key With Root, i.e 8 as 6<8, search in left subtree of 8

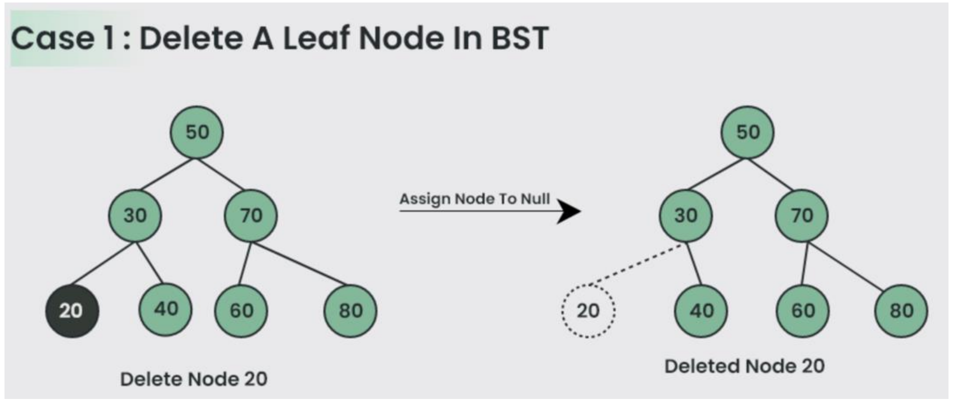
As Key (6) Is Greater Than 3, Search In The Right Subtree Of 3

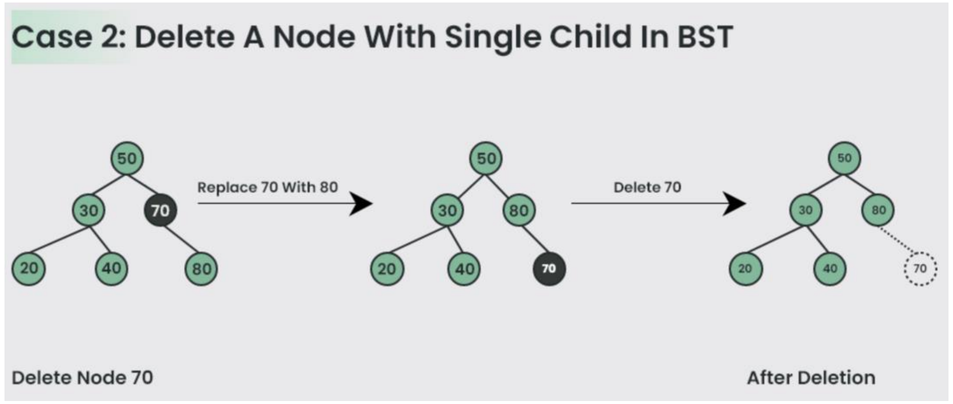
As 6 Is Equal To Key (6), So We Have Found The Key

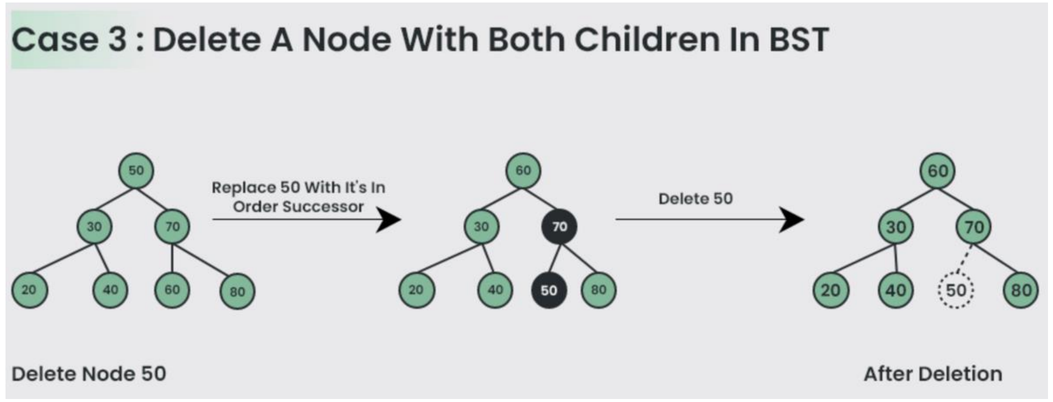
Figure from: https://www.geeksforgeeks.org/binary-search-tree-set-1-search-and-insertion/

Binary Search Trees

- Basic operations on BST
  - Insertion
  - Searching
  - **Deletion**





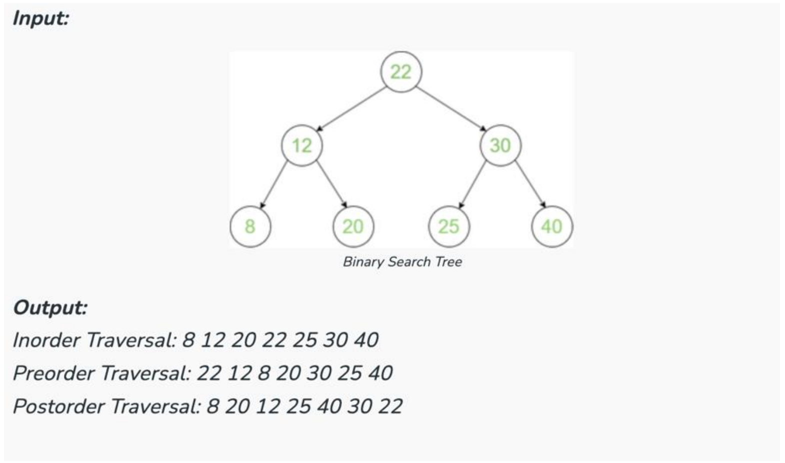


Case 3 : Delete A Node With Both Children In BST
Replace 50 With It's In Order Successor
Delete 50
Delete Node 50
After Deletion

Figure from: https://www.geeksforgeeks.org/deletion-in-binary-search-tree/

Binary Search Trees

- Basic operations on BST
  - Insertion
  - Searching
  - Deletion
  - **Traversal**
    - Inorder traversal
    - Preorder traversal
    - Postorder traversal



Input:

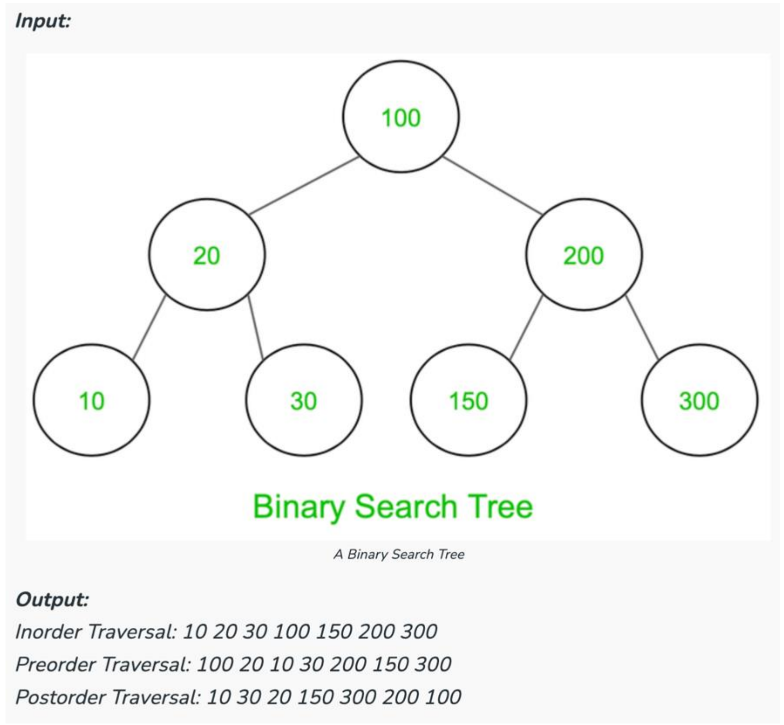
Binary Search Tree

Output:

Inorder Traversal: 8 12 20 22 25 30 40

Preorder Traversal: 22 12 8 20 30 25 40

Postorder Traversal: 8 20 12 25 40 30 22



Input:

A Binary Search Tree

Output:

Inorder Traversal: 10 20 30 100 150 200 300

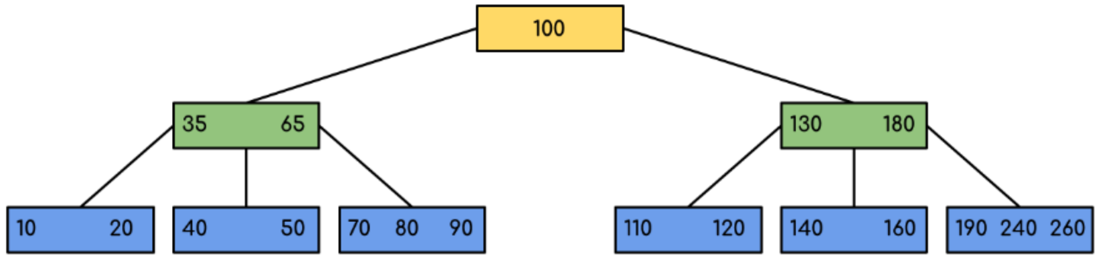
Preorder Traversal: 100 20 10 30 200 150 300

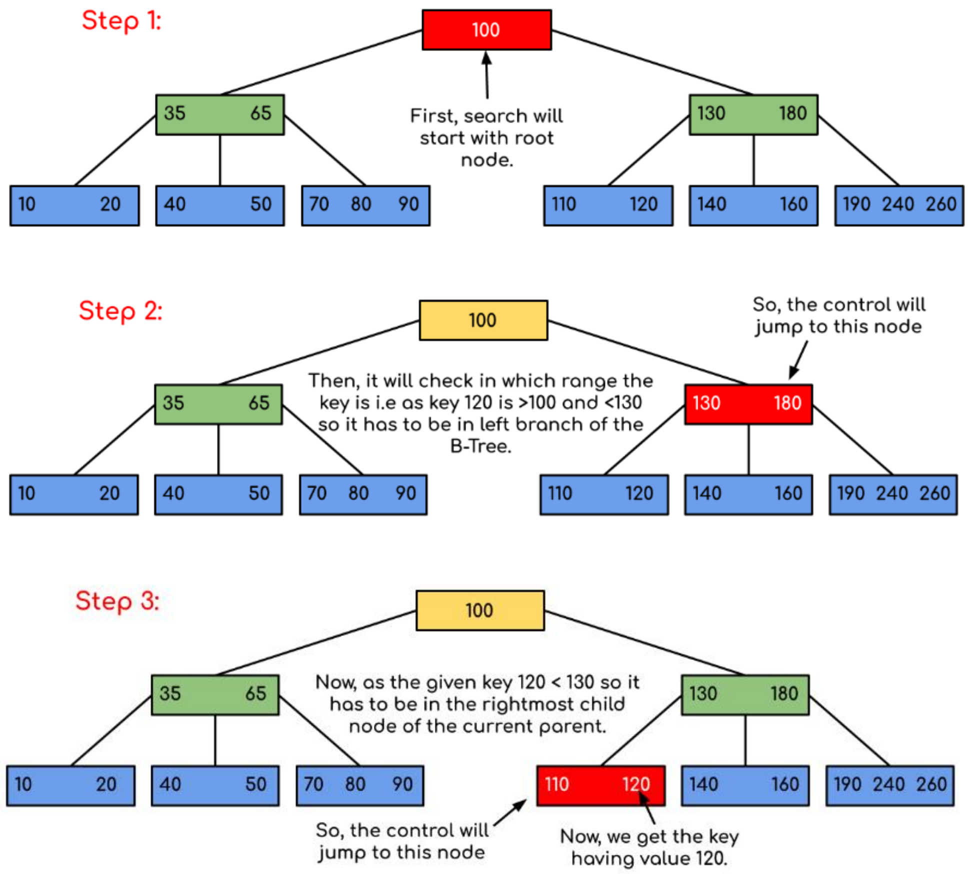
Postorder Traversal: 10 30 20 150 300 200 100

Figure from: https://www.geeksforgeeks.org/binary-search-tree-traversal-inorder-preorder-post-order/

B-Tree

- **B-Trees**: B-trees are generalizations of binary search trees in that a node can have more than two children.





Step 1:
100
First, search will start with root node.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
Step 2:
100
So, the control will jump to this node
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
Step 3:
100
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
35 65
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

Source from: https://www.geeksforgeeks.org/introduction-of-b-tree-2/

Hash Tables

- A **hash table**, also known as a hash map, is a data structure that implements an associative array abstract data type, a structure that can map keys to values.
- A hash table uses a hash function to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

Hash Tables

- **Hash Function**: Converts the key into an integer index, which determines where the value is stored in the table.
- **Buckets or Slots**: These are the array elements where data records (key-value pairs) are stored.
- **Collision Resolution**: Since a hash function can produce the same index for more than one key, mechanisms such as chaining (using additional data structures like linked lists) or open addressing (finding another open slot in the array) are used to resolve collisions.

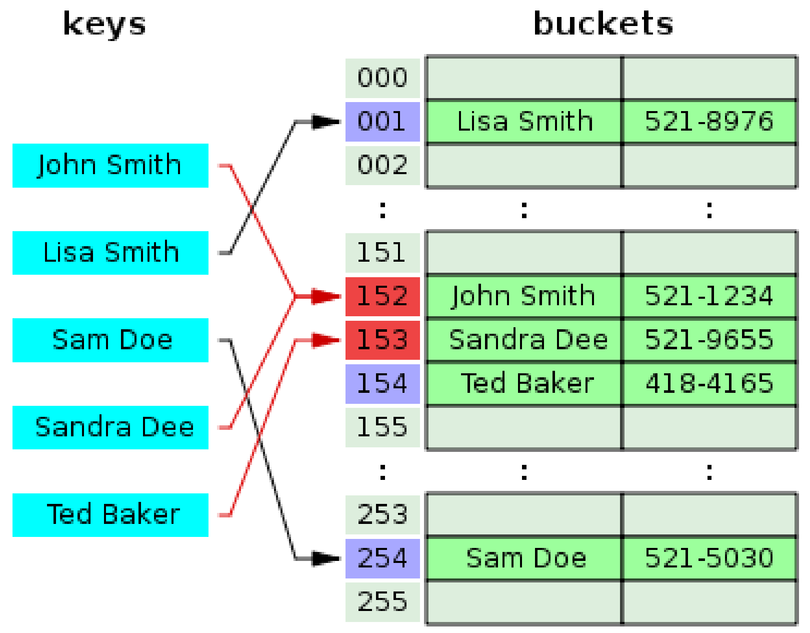


Figure from: https://hopegiometti.medium.com/my-first-data-structure-hash-tables-536ce099d73a

Applications of Hash Tables

- Hash tables are widely used due to their efficient average-case time complexity for lookups, insertions, and deletions, generally being $O(1)$.
  - **Database Indexing**
    - Hash tables are used to index large datasets in databases, allowing for rapid retrieval of information based on a key.
    - For instance, a database may use a hash table to quickly locate customer information using a customer ID as the key.
  - **Caching**
    - Hash tables are ideal for caching data from a database or a slow API.
    - For instance, a web application might store the results of a complex database query in a hash table (cache), where the query string itself could be the key, and the result set the value.

Applications of Hash Tables

- **Unique Data Representation**
  - Hash tables can be used to implement sets, where each key is unique, and the presence of the key in the hash table signifies membership in the set.
  - This is useful in scenarios requiring quick membership checks, like filtering unique items from a list of data.
- **Counting and Frequency Dictionaries**
  - Hash tables are excellent for counting occurrences of elements in a collection (like words in a document).
  - The keys would be the elements themselves (e.g., words), and the values are the counts of occurrences.

Applications of Hash Tables

- **Implementing Associative Arrays**
  - In programming languages that support it (like Python with dictionaries, JavaScript with objects, etc.), associative arrays are implemented using hash tables.
  - They allow developers to associate unique keys with specific values efficiently.
- **Lookup Tables**
  - Hash tables can store precomputed results of a function to save computational time.
  - For example, in graphics programming, a hash table might be used to store the results of complex lighting calculations for different input parameters to speed up rendering.
- **Tracking User Sessions**
  - In web development, hash tables can be used to track user sessions where each session ID (key) corresponds to user-specific data (value).
  - This facilitates efficient access and management of user session data across web requests.

Hash Tables

- **Advantages** of Hash Table:
  - **Efficient Data Access**: Provides nearly constant time complexity for search, insert, and delete operations on average.
  - **Flexibility of Keys**: Allows any item that can be hashed to be used as a key, unlike arrays which only use integer indices.

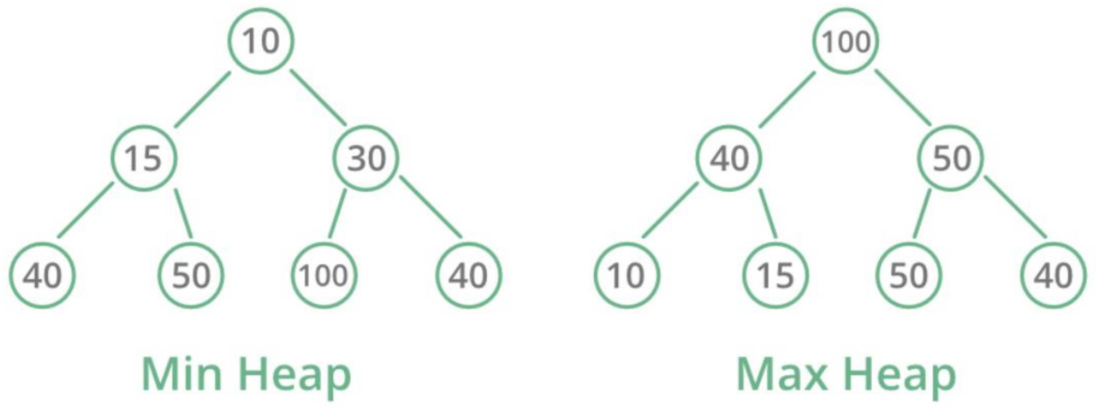
- **Limitations** of Hash Table:
  - **Performance Issues**: Poorly designed hash functions can lead to many collisions, significantly slowing down access times.
  - **Memory Overhead**: Handling collisions with chaining can require additional memory outside the initial table array.

Heaps

- A **heap** is a specialized tree-based data structure that satisfies the heap property:
  - the value of each node is greater than or equal to the value of its children
  - makes sure that the root node contains the **maximum** or **minimum** value (depending on the type of heap), and the values decrease or increase as you move down the tree.
- This structure is typically visualized as a binary tree but is usually implemented using arrays for efficiency.

Heaps

- The heap can be of two types, each serving different sorting needs:
  - **Max-Heap**: In a max-heap, the value of each node is greater than or equal to the values of its children. The largest element is found at the root.
  - **Min-Heap**: In a min-heap, the value of each node is less than or equal to the values of its children. The smallest element is found at the root.



Applications of Heaps

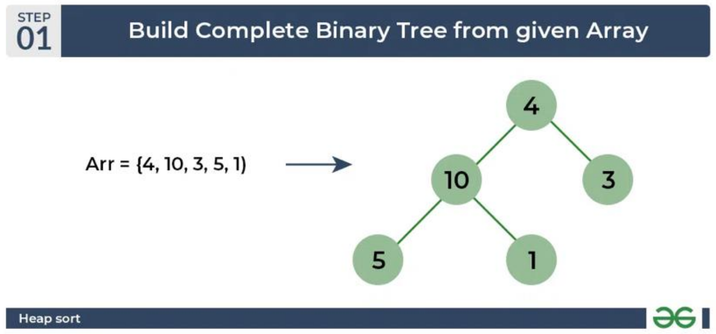
- **Heaps** are commonly used to implement priority queues, where elements are retrieved based on their priority (maximum or minimum value).
  - This is useful in scheduling tasks, handling interruptions, and processing events.
- **Heapsort** is a sorting algorithm that uses a heap to sort an array in ascending or descending order.
  - It has a time complexity of O(n log n), making it efficient for large datasets.
- Heaps are used in graph algorithms like Dijkstra’s algorithm and Prim’s algorithm for finding the shortest paths and minimum spanning trees.

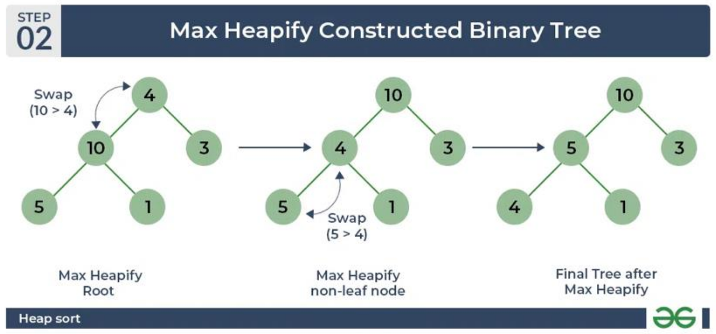
Heap Operations

- Common heap operations are:
  - **Insert**: Adds a new element to the heap while maintaining the heap property.
  - **Extract Max/Min**: Removes the maximum or minimum element from the heap and returns it.
  - **Heapify**: Converts an arbitrary binary tree into a heap.

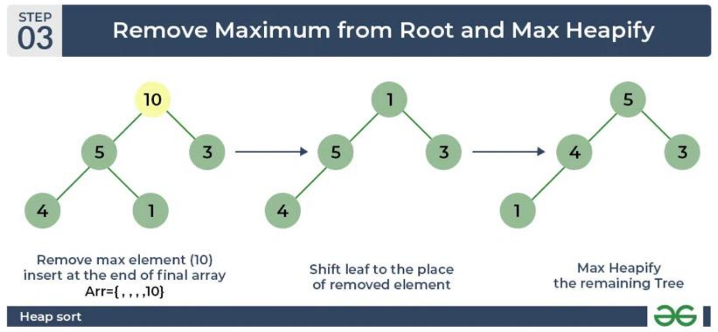
Heap Sort

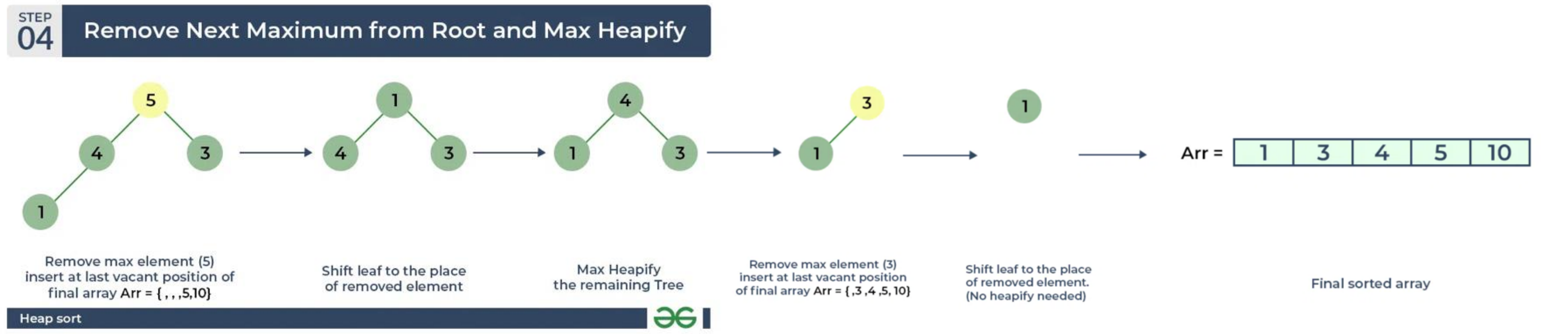
Transform array into max heap




STEP 02
Max Heapify Constructed Binary Tree
Swap (10 > 4)
Swap (5 > 4)
Max Heapify Root
Max Heapify non-leaf node
Final Tree after Max Heapify
Heap sort

Perform Heap Sort


STEP 03
Remove Maximum from Root and Max Heapify
Remove max element (10) insert at the end of final array Arr={ , , , ,10}
Shift leaf to the place of removed element
Max Heapify the remaining Tree
Heap sort


STEP 04
Remove Next Maximum from Root and Max Heapify
Remove max element (5) insert at last vacant position of final array Arr = { , , ,5,10}
Shift leaf to the place of removed element
Max Heapify the remaining Tree
Remove max element (3) insert at last vacant position of final array Arr = { ,3 ,4 ,5, 10}
Shift leaf to the place of removed element. (No heapify needed)
Arr = 1 3 4 5 10
Final sorted array
Heap sort

Summary & To-do List

- Please download and read the materials provided on Moodle.
- Review the content learnt from 6.

- Assessments
  - Continue to work on Group Assessment 1.
  - Attend the applied session for Quiz 1.

- Next week: Data Quality & Anomalies