



# Data Structures Study Notes

## What is the main purpose of a data structure?

The main purpose of a data structure is to organize and store data efficiently, allowing for optimal access, manipulation, and management of that data.

Data structures provide a way to represent and arrange data in computer memory in several ways, enabling algorithms to perform operations on that data with as little cost to runtime and memory space as possible. This efficiency is important for optimizing the performance of applications, especially when dealing with large datasets or resource-constrained environments.

By minimizing the computational and storage overhead, well-designed data structures enable faster processing times and more efficient use of system resources. By choosing the appropriate data structure for a given problem, developers can improve the performance and efficiency of their programs, as different data structures excel at different types of operations and use cases.

## Here are some common examples of data structures:

- **Arrays:** Ordered collections of elements stored in contiguous memory locations. They provide fast random access to elements by their index. Arrays are commonly used for implementing other data structures such as stacks and queues.
- **Trees:** Hierarchical structures with a root node and child nodes, used for representing hierarchical relationships and efficient searching.
- **Hash Tables:** Data structures that use a hash function to map keys to array indices, allowing for fast insertion, deletion, and lookup operations.
- **Linked Lists:** Sequences of nodes where each node contains data and a reference to the next node (called a pointer), without actually being in the same contiguous memory locations. This allows for efficient insertion and

deletion of elements at any point in the list, as well as dynamic memory allocation (growing and shrinking without issue). Linked lists are particularly useful when the size of the data structure needs to change frequently or when memory allocation needs to be more flexible than what arrays can provide.

- **Stacks:** Last-In-First-Out (LIFO) structures for managing data. Stacks are commonly used in programming for function call management, expression evaluation, and undo mechanisms. They provide efficient access to the most recently added elements, making them ideal for scenarios where the order of operations is important (specifically the most recent operation).

### Measuring efficiency and performance:

In order to measure a data structure's efficiency and performance, we use a concept called Big O notation. This notation helps us analyze and compare the time and space complexity of different algorithms and data structures. By understanding Big O, we can make informed decisions about which data structure to use for specific tasks.

Big O is basically asking the question, "if the input size of the data increases, how many steps does it take the algorithm to do an operation?"

## Big O

Big O notation is a mathematical notation used to describe the upper bound or worst-case scenario of an algorithm's time or space complexity. It provides a standardized way to express how an algorithm's performance scales with input size (typically denoted as  $n$ ).

### Common Big O Notations - Time Complexity

#### O(1) - Constant Time

- **What it means:** The algorithm's runtime does not depend on the size of the input ( $n$ ). It always takes the same amount of time to execute, regardless of the input size.
- **Example:** Accessing an element in an array by index. This takes one step, regardless of how many elements the array may hold.

- **Visual:** The runtime is flat—unchanging as  $n$  grows.

## **$O(\log n)$ - Logarithmic Time**

- **What it means:** The runtime grows logarithmically as the input size increases. Often seen in algorithms that divide the input into smaller parts (e.g., halving the input).
- **Common Examples:** Binary search, efficient tree operations.
- **Visual:** Runtime grows very slowly as  $n$  increases. It halves the steps as input increases.

## **$O(n)$ - Linear Time**

- **What it means:** The runtime grows linearly with the size of the input. If the input size doubles, the runtime also doubles.
- **Common Examples:** Looping through an array, linear search.
- **Visual:** The runtime grows directly proportional to the input size.

## **$O(n \log n)$ - Linearithmic Time**

- **What it means:** The runtime grows proportional to  $n$ , but with an extra logarithmic factor. This is common in divide-and-conquer algorithms that operate on all input parts.
- **Common Examples:** Merge sort, quicksort (average case), heapsort.
- **Visual:** Faster than  $O(n^2)$  but slower than  $O(n)$  for large inputs.

## **$O(n^2)$ - Quadratic Time**

- **What it means:** The runtime grows quadratically with the size of the input. This happens in algorithms that involve nested loops, where each loop iterates  $n$  times.

- **Common Examples:** Bubble sort, insertion sort, selection sort (on unsorted arrays).
- **Visual:** Runtime becomes very slow as input size grows.

## $O(2^n)$ - Exponential Time

- **What it means:** The runtime doubles with each additional input. This is often seen in algorithms that solve problems through exhaustive search, such as certain recursive algorithms without optimization.
- **Common Examples:** Recursive solutions for the Fibonacci sequence, the subset sum problem (brute force).
- **Visual:** Runtime grows explosively and is infeasible for large inputs.

---

# Data Structures

## Array

The array is one of the most basic data structures.

```
[1, 2, 3, 4, 5, 6]
["This", "is", "an", "array", "example"]
```

Arrays have:

- Size (length) = number of elements
- Index = number that identifies the position of an element

For example: An array with 5 elements is of **size 5**.

Indexes always start a 0, so the last number of this 5-element array will be **index 4**.

["This", "is", "an", "array", "example"]

0, 1, 2, 3, 4

## Operations of an Array

- Reading

- A computer can read from an array in just **one step**.
- This is because the computer has the ability to jump to any particular index in the array and look inside. We provide it an index (position) in the array and it can jump to that instantly.
- For example: Go to index 3.
- Time complexity:  $O(1)$

- Searching

- Searching means providing the computer a *value* and asking it to return the index number of that value's location in the array.
- Searching is tedious since the computer has no way of knowing where that value is at the start. It must search each index (each element) to find the value.
- Time complexity:  $O(n)$

- Insertion

- Insertion is adding a new data element to the array.
- The efficiency of inserting a new piece of data into an array depends on *where* within the array you're inserting it.
- If you're adding a new data element to the end of an array, insertion takes just one step. The computer knows where the end of the array is instantly because when allocating an array in memory, the computer always keeps track of the array's size. In other words, the computer knows where it ends.
- If the array begins at memory address 1010 and is of size 5, that means its final memory address is 1014. So to insert an item beyond that would mean adding it to the *next* memory address, which is 1015.

- Once the computer realizes this, it can jump to the last index and add the next value in one step. And it doesn't matter how long the array is, because we always know its size.
- However, inserting new data at the *beginning* of an array (or the middle) is a different story.
- In these cases, we need to *shift* pieces of data to make room for what we're inserting, and this leads to additional steps.
- If the array is at the beginning of a 5-element array, then 5 elements will need to shift to the right to make room for the new element at the start.
- In other words, it will require a step for each element in the array. That's  $n$  steps if the  $n$  is representing the number of elements in the array.
- Time complexity:  $O(n)$
- Deletion
  - Deletion from an array is the process of eliminating a value at a particular index.
  - So for example, deleting the value at index 2.
  - While the actual deletion of a value takes just one step, we may face a problem:
    - If the value deleted was somewhere in the middle of the array (between two values) or at the beginning, it causes an empty cell to sit where that value once was.
    - An array isn't effective when there are gaps between its indexes (it must be contiguous), so to resolve this, the computer shifts all values to the left until it fills the gap.
    - This means our deletion process requires additional steps.
  - Like insertion, the worst-case scenario of deleting an element is deleting the very first element of the array.
  - This is because index 0 would become empty, and we'd have to shift all the remaining elements to the left to fill the gap.
  - Time complexity:  $O(n)$

## Summary: Array

- **Description:** A contiguous block of memory where elements are indexed.
- **Time Complexity:**

Operation	Time Complexity	Explanation
<b>Reading</b>	$O(1)$	Accessing an element by index is instantaneous always.
<b>Searching</b>	$O(n)$	In the worst case, we might have to scan the entire array (n elements) to find the target element.
<b>Insertion</b>	$O(n)$	Inserting into the middle or beginning requires shifting elements (n elements).
<b>Deletion</b>	$O(n)$	Removing from the middle or beginning requires shifting elements (n elements).

## Stack

A stack is an abstract data type (ADT) with a linear arrangement resembling a pile. It is similar to an array in that items can be arranged linearly. However, the difference is that it is a last-in, first-out (LIFO) data structure. This means that stacks only allow access to the most recently added data first (top of stack) and you can only remove data from the top of stack.

So basically, the most recently added data gets removed first from the pile.

### Operations of a Stack

- Reading
  - A computer can read from a stack in just one step, similar to an array.
  - This is because the computer always has access to the top element of the stack, which is where reading operations occur.
  - Time complexity:  $O(1)$
- Searching

- Searching means providing the computer a *value* and asking it to return the index of that value's location in the stack.
- Searching a stack takes linear time,  $O(n)$ , in the worst case. This is because in a stack, elements are only accessible from the top, and to search for a specific value that exists in the stack, you might need to pop (remove) elements one by one until you find the desired value or reach the bottom of the stack. So for a stack of  $n$  elements, it might take  $n$  steps.
- Time complexity:  $O(n)$
- Insertion
  - Insertion is adding a new data element to the stack.
  - In a stack, insertion (also called "push") always occurs at the top of the stack. This operation is very efficient because the new element is simply placed on top of the existing elements.
  - The computer always knows where the top of the stack is, so it can add a new element in a single step, regardless of how many elements are already in the stack.
  - Time complexity:  $O(1)$
- Deletion
  - Deletion from a stack (also called "pop") always occurs at the top of the stack. This operation is very efficient because the computer only needs to remove the topmost element. Like insertion, deletion in a stack can be performed in a single step, regardless of the stack's size.
  - Time complexity:  $O(1)$

## Summary: Stack

- **Description:** A last-in, first-out (LIFO) data structure. Operations occur at the top of the stack.
- **Time Complexity:**



Operation	Time Complexity	Explanation
<b>Reading</b>	$O(1)$	Reading is instantaneous as it can only read the item at the top of the stack and can easily locate it
<b>Searching</b>	$O(n)$	Must scan the entire stack to find the target element.
<b>Insertion</b>	$O(1)$	Adding (pushing) to the top is instantaneous.
<b>Deletion</b>	$O(1)$	Removing (popping) from the top is instantaneous.

## Queue

A queue is a linear data structure, similar to a stack, but has a different ordering rule. Queues follow the First In, First Out (FIFO) principle. Unlike a stack, elements are added at one end (called the rear or tail) and removed from the other end (called the front or head). Think of it like people waiting in line - the first person to join the queue is the first one to be served. The last person in line has to wait.

Queues are typically implemented using a **linked list**

### Operations of a Queue

- Reading
  - A computer can read from a queue in constant time  $O(1)$  when accessing elements at the front of the queue. This is because the front pointer always points to the first element, making it immediately accessible. However, accessing elements at other positions requires traversing through the queue sequentially.
  - Time complexity:  $O(1)$
- Searching
  - Searching means providing the computer a *value* and asking it to return the index of that value's location in the queue.
  - Searching a queue takes linear time  $O(n)$  in the worst case scenario. This is because you might need to examine each element in the queue sequentially from front to back to find a specific value. Since elements can

only be accessed from the front of the queue, searching through the entire queue requires checking each element one by one until the desired value is found or the end is reached.

- Time complexity:  $O(n)$
- Insertion
  - Insertion is adding a new data element to the queue.
  - In a queue, insertion (also called "enqueue") always occurs at the rear/tail of the queue. This operation is very efficient because we maintain a pointer to the rear of the queue, allowing us to add new elements in a single step regardless of the queue's size.
  - Time complexity:  $O(1)$
- Deletion
  - Deletion from a queue (also called "dequeue") always occurs at the front/head of the queue. Just like insertion, deletion is very efficient because we maintain a pointer to the front of the queue, allowing us to remove elements in a single step regardless of the queue's size. When an element is dequeued, the front pointer is simply moved to the next element in line.
  - Time complexity:  $O(1)$

## Summary: Queue

- **Description:** A First In, First Out (FIFO) data structure where elements are added at the rear and removed from the front, similar to a real-world queue or line. Elements are processed in the order they arrive, making it ideal for managing sequential operations or waiting lists.
- **Time Complexity:**

Operation	Time Complexity	Explanation
Reading	$O(1)$	Reading from front of queue is instantaneous
Searching	$O(n)$	Must traverse through all elements to find target value

<b>Insertion</b>	O(1)	Adding to rear is constant time with tail pointer
<b>Deletion</b>	O(1)	Removing from front is constant time with head pointer

## Set

A set is just like an array, but the only difference is that it does not allow duplicate elements. Each element in a set must be unique. By unique, we mean that no two values can be the same. This property of "distinction" between elements makes sets particularly useful for storing collections of unique items or for quickly checking whether an item is already present in a collection.

### Operations of a Set

- Reading
  - A computer can read from a set in constant time, similar to an array. However, the implementation of a set can affect this operation.
  - If it is an array-based set, reading would have a time complexity of  $O(n)$  in the worst case, as it may need to search through all the elements to find the desired one.
  - In most modern implementations, sets use hash tables internally, which allows for  $O(1)$  average-case time complexity for reading (checking if an element exists). This data structure is particularly efficient for set operations. By using a hash function to map elements to specific locations in the table, sets can achieve constant-time performance for most operations on average.
  - However, even in a hash table set, in the worst-case scenario (due to hash collisions), the time complexity could degrade to  $O(n)$ .
  - Time complexity:  $O(1)$  average case,  $O(n)$  worst case
- Searching
  - Searching means providing the computer a *value* and asking it to return the index of that value's location.

- Searching a set is typically very efficient due to the set's internal structure. In most implementations, sets use hash tables, which allow for constant-time average-case complexity. If it is an array-based set, searching would have a time complexity of  $O(n)$  in the worst case, as it may need to search through all elements to find the desired one. This is because array-based sets don't have the same efficient lookup mechanisms as hash table-based sets. The choice between array-based and hash table-based implementations can significantly impact the performance of set operations, especially for large datasets.
- However, in the worst-case scenario (due to hash collisions), the search operation could still degrade to linear time  $O(n)$ .
- Time complexity:  $O(1)$  average case,  $O(n)$  worst case
- Insertion
  - Insertion is where sets and arrays diverge.
  - Inserting a value into a set means that the computer first needs to determine that this value doesn't already exist in the set, because that's what sets do: they prevent duplicate data from being inserted into them.
  - Because of this, the computer will first need to search the set to see whether the value we want to insert is already in there.
  - So every insertion into a set first requires a search.
  - Inserting a value at the end of a set is the best-case scenario, but we still have to search each of the previous elements to know if there are any duplicates before performing the final insertion.
  - Said another way: insertion into the end of a set will take up to  $N + 1$  steps for  $N$  elements. This is because there are  $N$  steps to ensure that the value doesn't already exist within the set, and then one step for the actual insertion.
  - Time complexity:  $O(n)$
- Deletion
  - Deletion from a set is typically very efficient, especially in hash table-based implementations. The process involves searching for the element to

be deleted and then removing it from the set. In most cases, this operation can be performed in constant time.

- However, like insertion and searching, the worst-case scenario for deletion in a set is  $O(n)$  due to potential hash collisions or if using an array-based implementation.
- In an array-based set, deletion takes  $O(n)$  time in the worst case. This is because after removing an element, we may need to shift all the subsequent elements to fill the gap left by the deleted item. For example, if we delete the first element of the set, we'd have to move all the remaining elements one position to the left to maintain the contiguous nature of the array.
- Time complexity:  $O(n)$

## Summary: Set

- **Description:** A collection of unique elements where each item appears only once. Sets are useful for storing distinct values.
- **Time Complexity:**

Operation	Time Complexity	Explanation
<b>Reading</b>	$O(1)$ avg, $O(n)$ worst	Constant time on average due to hash table implementation, but can degrade in worst case
<b>Searching</b>	$O(1)$ avg, $O(n)$ worst	Efficient lookup in average case, but can be linear in worst case due to hash collisions
<b>Insertion</b>	$O(n)$	Requires searching to ensure uniqueness before inserting
<b>Deletion</b>	$O(n)$	May require shifting elements in array-based implementation

## Linked List

A linked list is a data structure where elements are stored in a sequence of nodes, with each node containing both data and a reference (or pointer) to the next node

in the sequence. Unlike arrays where elements are stored in contiguous memory locations, linked lists can be stored in scattered memory locations while maintaining their sequential nature through these node connections.

Each node holds a piece of data, as well as the pointer referencing the next node in memory.

The last node in the list points to null, indicating the end of the list. This structure allows for efficient insertion and deletion of elements at any point in the list, though it requires traversing from the beginning to reach specific positions.

Linked lists come in two varieties: singly linked lists and doubly linked lists. In a singly linked list, each node only contains a reference to the next node in the sequence. A doubly linked list, on the other hand, contains references to both the next and previous nodes, allowing for bidirectional traversal. A doubly linked list can go either back or forward.

## Operations of a Linked List

- [Reading](#)
  - A computer can read from a linked list by traversing through the nodes sequentially. Unlike arrays where elements are stored in contiguous memory locations, accessing elements in a linked list requires following the node references from the beginning until reaching the desired position. This means that to read the  $n$ th element, we need to traverse through  $n-1$  nodes first. That means that reading a linked list takes linear time, meaning the time increases proportionally with the size of the list. This is because we must traverse through each node sequentially, following the chain of references, to reach any desired element. We cannot just jump to an index like we can with an array. We must traverse the nodes always.
  - Time complexity:  $O(n)$
- [Searching](#)
  - Searching means providing the computer a *value* and asking it to return the index of that value's location.
  - Searching for a value in a linked list requires traversing through the nodes sequentially from the beginning until the target value is found. Since we

need to examine each node one by one, this process has a linear time complexity. In the worst case, we might need to traverse the entire list if the element is at the end or not present at all.

- Time complexity:  $O(n)$
- Insertion
  - Insertion is adding a value into a linked list.
  - Inserting a value into a linked list takes different amounts of time depending on where in the list we're inserting. If we're inserting at the beginning (head) of the list, it takes  $O(1)$  time since we only need to update a few pointers for inserting a new node. However, if we're inserting at any other position, we first need to traverse to that position by following each node sequentially, making the time complexity  $O(n)$  in the worst case.
  - Time complexity:  $O(n)$
- Deletion
  - Deletion from a linked list involves removing a node from the list by adjusting the pointers of surrounding nodes. Like insertion, deletion from the beginning of the list takes  $O(1)$  time, but deleting from any other position requires traversing to that position first, making it  $O(n)$  in the worst case. The process involves updating the pointer of the previous node to skip over the deleted node and point to the next node in the sequence instead.
  - Since we're not technically deleting the target node and just "unreferencing" it by choosing not to point to it anymore, that means that it still exists in memory. It will be garbage collected by the program later when the computer realizes there are no more references pointing to it in memory. This automatic memory management helps prevent memory leaks and ensures efficient use of system resources.
  - Time complexity:  $O(n)$

## Summary: Linked List

- **Description:** A linear collection of nodes where each node points to the next (singly linked list) or both next and previous (doubly linked list).
- **Time Complexity:**

Operation	Time Complexity	Explanation
<b>Reading</b>	$O(n)$	Must traverse the list to access an element.
<b>Searching</b>	$O(n)$	Must traverse the list to find the target element.
<b>Insertion</b>	$O(1)$ best case, $O(n)$ worst case	Adding to the head or tail (if tail is tracked) is instantaneous. Anywhere else requires $n$ steps
<b>Deletion</b>	$O(1)$ best case, $O(n)$ worst case	Removing a node is instantaneous if it's at the head. However, if it's at the end, we need to traverse all nodes to reach it.

## Hash Table

A hash table is a list of paired values. The first item in each pair is called the *key*, and the second item is called the *value*. In a hash table, the key and value have some significant association with one another. For example: menu = { "french fries": 0.75, "hamburger": 2.5 }

In the example provided, the string "french fries" is the key, and the 0.75 is the value.

Hash tables are an efficient data structure that use a hash function to map keys to locations in memory. The hash function will use some sort of mathematical formula to convert a value to a particular location in memory.

For example, if we have a key "apple", the hash function might perform mathematical operations on the characters in "apple" to generate a specific memory address like 1052. This hash value then determines where in the hash table the key-value pair will be stored. This process of taking characters and converting them to numbers is known as *hashing*.

This mapping allows for extremely fast lookups and insertions, making hash tables ideal for scenarios where quick access to data is crucial. The hash function converts each key into a unique memory address, though collisions can occur when different keys hash to the same location.



## Operations of a Hash Table

- Reading

- Reading from a hash table involves using the hash function to convert the key into a memory address and then accessing that location directly. This makes reading operations extremely efficient in the average case, taking constant time  $O(1)$ . However, in cases where there are hash collisions (multiple keys mapping to the same location), the operation might require additional steps to resolve the collision, leading to a worst-case time complexity of  $O(n)$ .
- Time complexity:  $O(1)$  average case,  $O(n)$  worst case

- Searching

- Searching means providing the computer a *value* and asking it to return the index of that value's location.
- Searching for a value in a hash table involves a more complex process compared to reading. Since hash tables store key-value pairs, searching for a specific value requires checking each key-value pair until the desired value is found. This means we might need to examine every element in the hash table, making the worst-case time complexity  $O(n)$ . Unlike searching by key, which can leverage the hash function for direct access, searching by value requires a linear scan of the table.
- If we're searching by key, however, the search is instantaneous, because of hashing.
- Time complexity:  $O(1)$  average case,  $O(n)$  worst case

- Insertion

- Insertion is adding a value into the data structure.
- Inserting a value into a hash table involves first using the hash function to compute the memory location for the new key-value pair. In the average case, this is a constant-time operation  $O(1)$ , as the hash function directly maps to an available slot. However, when hash collisions occur (multiple

keys mapping to the same location), the operation might require additional steps like chaining or probing to find an available slot, leading to a worst-case time complexity of  $O(n)$ .

- Time complexity:  $O(1)$  average case,  $O(n)$  worst case
- Deletion
  - Deletion from a hash table involves removing an element from the data structure.
  - Deleting an item from a hash table follows a similar process to insertion. The hash function is used to locate the key-value pair, and in the average case, deletion takes constant time  $O(1)$ . However, when dealing with collision resolution methods like chaining or when the hash table needs to be resized, the operation might require  $O(n)$  time in the worst case.
  - Time complexity:  $O(1)$  average case,  $O(n)$  worst case

## Summary: Hash Table

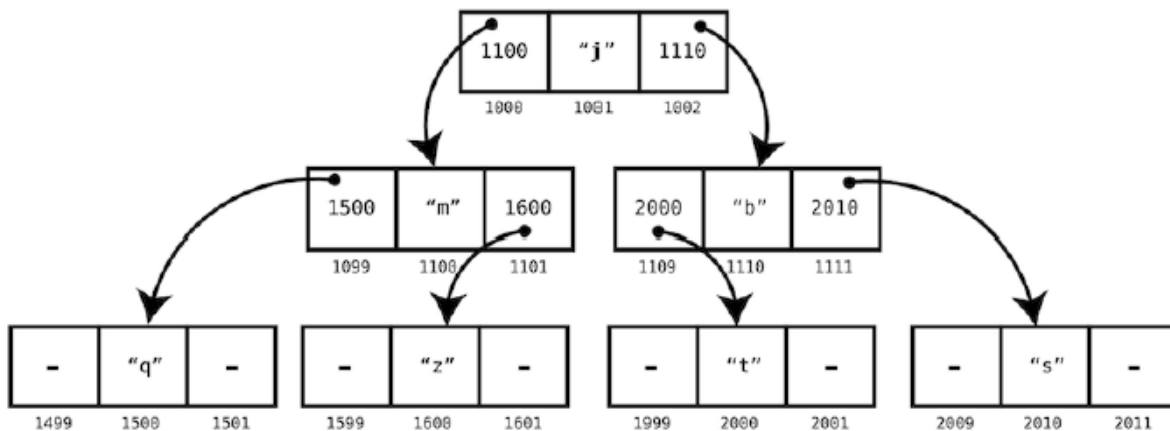
- **Description:** A data structure that maps keys to values using a hash function.
- **Time Complexity:**

Operation	Time Complexity	Explanation
Reading	$O(1)$	Accessing a value by its key is instantaneous.
Searching	$O(1)$	Searching for a key is fast due to hashing (ignoring hash collisions).
Insertion	$O(1)$	Adding a key-value pair is fast (ignoring hash collisions).
Deletion	$O(1)$	Deleting a key-value pair is fast (ignoring hash collisions).

## Binary Tree

A tree is a node-based data structure, just like a linked list, except that a tree is setup differently. In a tree, each node connects to two other nodes in a hierarchy, creating what's known as a binary tree structure. Each node in a binary tree can have at most two child nodes - typically referred to as the left child and right child. This hierarchical arrangement allows for efficient organization and searching of data, particularly when the tree is "balanced".

A tree is balanced when its nodes' subtrees have the same number of nodes in it.



Trees come with their own unique nomenclature.

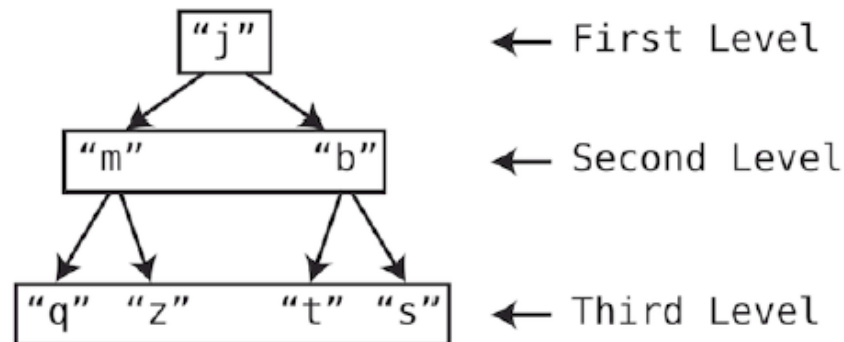
The uppermost node (in our example, "j") is called the *root*

The root is at the top of the tree; it's how trees are typically depicted.

In our example, we'd say that "j" is a parent to "m" and "b" (which are also nodes)

As in a family tree, a node can have descendants and ancestors. A node's descendants are all the nodes that stem from a node, while a node's ancestors are all the nodes that it stems from.

- Trees are said to have *levels*. Each level is a row within the tree. Our example tree has three levels:



The height of a tree is the longest path from the root node to any leaf node. A leaf node is a node that has no children (no nodes below it).

The height represents the maximum number of edges (links or lines) that must be traversed to reach the furthest leaf, which helps determine the tree's efficiency for various operations.

Knowing how to find the height of a tree is important as this will be on the assessment.

A **binary tree** is a type of tree where each node can have at most two children, typically referred to as the left child and right child. This structure allows for efficient organization of data in a tree-like format, with nodes branching off from a single root node at the top. The hierarchical nature of binary trees makes them particularly useful for representing relationships between data elements in a way that enables efficient searching and traversal operations.

- [Reading](#)
  - Reading from a binary tree involves accessing nodes in a specific order. The most common traversal methods are in-order (left subtree, root, right subtree), pre-order (root, left subtree, right subtree), and post-order (left subtree, right subtree, root). The time complexity varies depending on the traversal method and tree structure.

- Time complexity:  $O(n)$  - Since we need to visit each node once to perform a complete traversal of the tree, regardless of the traversal method chosen (in-order, pre-order, or post-order).
- Searching
  - Searching means providing the computer a *value* and asking it to return the index of that value's location.
  - Searching for a value in a binary tree involves traversing through the tree structure to find a specific value. In a standard binary tree, we may need to explore all nodes in the worst case since there's no guaranteed ordering of elements. This process typically involves checking the root node, then recursively searching both left and right subtrees until the value is found or all nodes have been checked.
  - Time complexity:  $O(n)$  - In a standard binary tree, we need to potentially examine all  $n$  nodes in the worst case to find a specific value, since there's no ordering guarantee. The time complexity would be better ( $O(\log n)$ ) in a balanced binary search tree, but for a regular binary tree it's  $O(n)$ .
- Insertion
  - Insertion is adding a value into the data structure.
  - Inserting a value into a binary tree typically involves finding the appropriate position in the tree structure where the new node should be placed. In a standard binary tree without specific ordering requirements, we can insert at any available position that maintains the binary tree property (maximum of two children per node). The process involves traversing the tree to find an empty spot (a null child pointer) and creating a new node at that location.
  - Time complexity:  $O(n)$  - In the worst case, we might need to traverse through all nodes to find an appropriate insertion point, particularly if we're trying to maintain any specific tree properties or balance.
- Deletion
  - Deletion from a binary tree involves removing an element from the data structure.

- Deleting an item from a binary tree involves first locating the node to be deleted, then handling the restructuring of the tree. The process becomes more complex when the node to be deleted has children - we may need to find a suitable replacement node and reorganize the surrounding nodes to maintain the tree structure. If the node has two children, we typically replace it with either its in-order predecessor or successor.
- Time complexity:  $O(n)$  - Similar to insertion, deletion requires first searching for the node ( $O(n)$  in worst case) and then potentially restructuring the tree, which could involve traversing and adjusting multiple nodes. The overall process remains linear in the worst case.

## Summary: Binary Tree

- **Description:** A hierarchical data structure where each node has at most two children (left and right).
- **Time Complexity):**

Operation	Time Complexity	Explanation
Reading	$O(n)$	Traversing all nodes is required to read all elements.
Searching	$O(n)$	Searching is efficient in balanced trees (BST) but not regular trees
Insertion	$O(n)$	Insertion requires traversal of tree to find the right node to insert into
Deletion	$O(n)$	Deletion requires traversal of tree to find the right node to delete

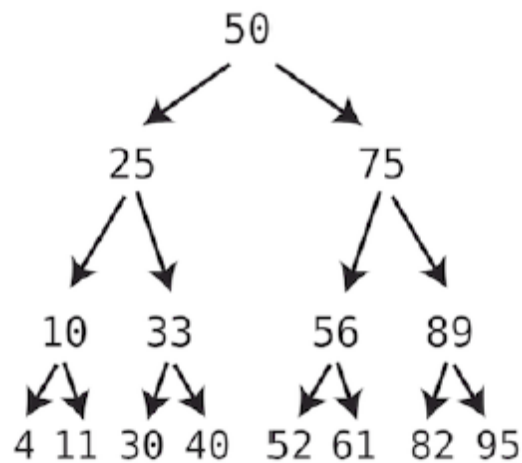
## Binary Search Tree

A binary search tree is a binary tree that follows a specific ordering rule: for any given node, all values in its left subtree are less than the node's value, and all values in its right subtree are greater than the node's value. This ordering property

makes binary search trees particularly efficient for searching, as we can quickly eliminate half of the remaining nodes at each step of our search.

The rules that make a binary tree a BST (binary search tree) are as follows:

- Each node can have at most one left child and one right child
- A node's left descendants can only contain values that are less than the node itself.
- A node's right descendants can only contain values that are greater than the node itself.



Note that each node has one child with a lesser value than itself, which is depicted using a left arrow, and one child with a greater value than itself, which is depicted using a right arrow.

Additionally, notice that all of the 50's left descendants are less than it. At the same time, all of the 50's right descendants are greater than it. The same pattern goes for each and every node.

While the following example is a binary tree, it's not a binary search tree:



It's a binary tree because each node has zero, one, or two children. But it's not a binary search tree, because the root node has two left children; that is, it has two children that are less than it. For a binary search tree to be valid, it can have at most one left (lesser) child and one right (greater) child.

- Reading

- Reading from a binary search tree involves traversing the tree structure in a systematic way. The most common traversal methods are in-order (left subtree, root, right subtree), pre-order (root, left subtree, right subtree), and post-order (left subtree, right subtree, root). In a binary search tree, an in-order traversal will visit nodes in ascending order of their values.
- Time complexity:  $O(n)$  - A complete traversal of a binary search tree requires visiting each node exactly once, regardless of the traversal method used. Even though BSTs have an ordered structure, reading all elements still requires examining every node in the tree.

- Searching

- Searching means providing the computer a *value* and asking it to return the index of that value's location.
- Searching for a value in a binary search tree is efficient due to its ordered structure. At each node, we can determine whether to search the left or right subtree based on comparing the target value with the current node's value. If the target value is higher than the current node, then we continue searching on the right. If the target value is lower than the current node, then we continue searching on the left. This binary decision process allows us to eliminate half of the remaining nodes at each step, making the search operation much faster than in a regular binary tree.



- Time complexity:  $O(\log n)$  - In a balanced BST, each comparison eliminates half of the remaining nodes from consideration. This logarithmic time complexity makes searching in a BST much more efficient than in a regular binary tree, though it can degrade to  $O(n)$  in an unbalanced tree.
- Insertion
  - Insertion is adding a value into the data structure.
  - Inserting a value into a binary search tree follows a similar process to searching. We start at the root and compare the new value with each node, moving left if the value is smaller or right if it's larger, until we find an empty spot where we can insert the new node while maintaining the BST property (all left descendants less than node, all right descendants greater).
  - Time complexity:  $O(\log n)$  - In a balanced BST, insertion follows the same path as searching, comparing the new value with nodes and moving left or right until finding the insertion point. Like searching, this takes logarithmic time in a balanced tree, though it can degrade to  $O(n)$  in an unbalanced tree.
- Deletion
  - Deletion from a binary search tree involves removing an element from the data structure.
  - Deleting an item from a binary search tree involves first locating the node to be deleted, then handling the restructuring based on the node's position and number of children. When deleting a node with no children, we simply remove it. For a node with one child, we connect its parent directly to its child. The most complex case is deleting a node with two children, where we typically replace it with its in-order successor (the smallest value in its right subtree) to maintain the BST property.
  - Time complexity:  $O(\log n)$  - Like insertion and searching, deletion in a balanced BST takes logarithmic time since we first need to find the node ( $O(\log n)$ ) and then perform restructuring operations that also take  $O(\log n)$ . However, in an unbalanced tree, this can degrade to  $O(n)$ .

## Summary: Binary Search Tree

- **Description:** A hierarchical data structure that follows specific ordering rules where all left child nodes must be less than their parent node and all right child nodes must be greater than their parent node. This organization enables efficient searching, insertion, and deletion operations through binary comparisons at each level.
- **Time Complexity):** (Balanced Binary Search Tree):

Operation	Time Complexity	Explanation
Reading	$O(n)$	Traversing all nodes is required to read all elements.
Searching	$O(\log n)$	Searching is efficient in balanced trees (e.g., BST).
Insertion	$O(\log n)$	Insertion maintains balance, keeping operations efficient.
Deletion	$O(\log n)$	Deletion requires rebalancing, which takes logarithmic time.

## Priority Queue

A priority queue is a specialized data structure that operates like a regular queue (FIFO) but with an important distinction: when inserting an element, we always make sure the data remains sorted based on their priority level rather than their order of entry.


Expressed another way:

A priority queue is a list whose deletions and access are just like a classic queue but whose insertions are like an ordered array. So we only delete and access data from the front of the priority queue, but when we insert data, we always make sure the data remains sorted in a specific order.

One classic example of where a priority queue is helpful is an application that manages the triage system for a hospital emergency room. In the ER, we don't treat people strictly in the order in which they arrived. Instead, we treat people in the order of the severity of their symptoms. If someone suddenly arrives with a life-threatening injury, that patient will be placed at the front of the queue, even if the person with the flu had arrived hours earlier.

Let's say our triage system ranked the severity of a patient's condition on a scale of 1 to 10, with 10 being the most critical. Our priority queue may look like this:


Patient C - Severity: 10
Patient A - Severity: 6
Patient B - Severity: 4
Patient D - Severity: 2



In determining the next patient to treat, we'll always select the patient at the front of the priority queue, since that person's need is the most urgent. In this case, the next patient we'd treat is Patient C.

If a new patient now arrives with a condition severity of 3, we'll initially place this patient at the appropriate spot within the priority queue. We'll call this person Patient E:

Patient C - Severity: 10
Patient A - Severity: 6
Patient B - Severity: 4
Patient E - Severity: 3
Patient D - Severity: 2



The priority queue is an example of an abstract data type— one that can be implemented using other, more fundamental, data structures. One straightforward way we can implement a priority queue is by using an ordered array; that is, we use an array and apply the following constraints:

When we insert data, we ensure we always maintain proper order. Data can only be removed from the end of the array. (This will represent the front of the priority queue.)

## Deque

A deque (double-ended queue) is a data structure that allows insertion and deletion of elements from both ends. It combines the functionality of both stacks and queues, making it a versatile data structure for situations where elements need to be processed from either end. Unlike a regular queue which only allows insertion at one end and deletion at the other, a deque provides flexibility by supporting operations at both ends efficiently.

Deques are usually implemented using doubly linked lists, as they provide efficient operations at both ends and maintain links between adjacent elements. This structure allows for  $O(1)$  time complexity for both insertion and deletion operations at either end. The double linking ensures that traversal can happen in both directions, making it ideal for implementing deque operations.

---

## Heaps

Heaps are of several different types, but we're going to focus on the binary heap.

The binary heap is a specific kind of binary tree.

As a reminder, a binary tree is a tree where each node has a maximum of two child nodes.

This is the same for heaps.

Now, even binary heaps come in two flavors: the max-heap and the min-heap. We're going to work with the max-heap for now, but as you'll see later, the difference between the two is trivial.

Going forward, I'm going to refer to this data structure simply as a heap, even though we're specifically working with a binary max-heap.

The **max-heap** is a binary tree that maintains the following conditions:

- The value of each node must be greater than each of its descendant nodes. This rule is known as the heap condition.

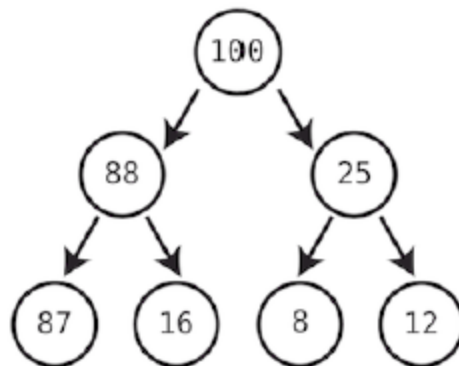
- The tree must be complete. (I'll explain the meaning of this shortly.)

Let's break down both of these conditions, starting with the heap condition.

## The Heap Condition

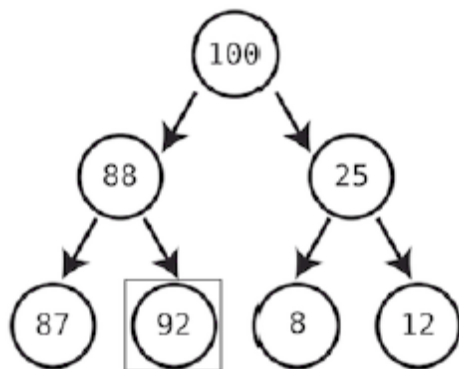
The heap condition says that each node's value must be greater than each and every one of its descendants.

For example, the following tree meets the heap condition since each node is greater than any of its descendants, as show in the following diagram:



In this example, the root node of 100 has no descendant that is greater than it. Similarly, the 88 is greater than both of its children, and so is the 25.

The following tree isn't a valid heap, because it doesn't mea the heap condition:



As you can see, the 92 is greater than its parent of 88. This violates the heap condition.

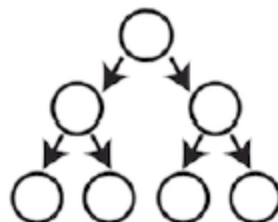
Note how the heap is structured very differently than the binary search tree. In a binary search tree, each node's right child is greater than it. In a heap, however, a node never has any descendant that is greater than it. As they say, "A binary search tree doth not a heap make." (Or something like that.)

## Complete Trees

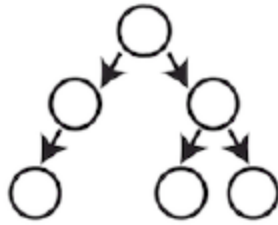
Now, let's get to the second rule of heaps—that the tree needs to be complete.

A complete tree is a tree that is completely filled with nodes; no nodes are missing. So if you read each level of the tree from left to right, all of the nodes are there.

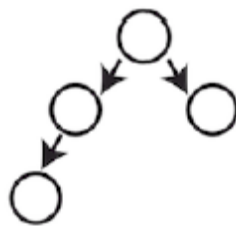
However, the bottom row can have empty positions, as long as there aren't any nodes to the right of these empty positions. This is best demonstrated with examples. The following tree is complete, since each level (meaning, each row) of the tree is completely filled in with nodes:



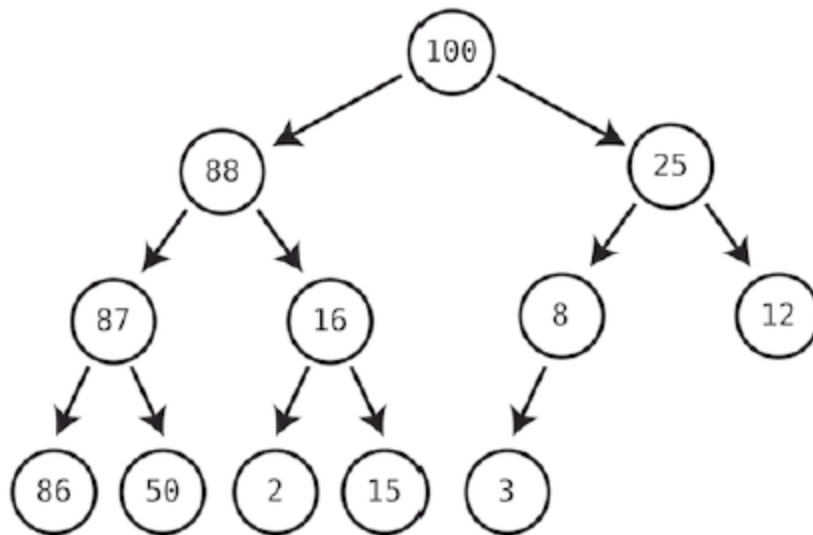
The following tree is not complete, since there's an empty position on the third level (and other nodes exist on the third level to the right of that empty position):



Now, the next tree is actually considered complete, since its empty positions are limited to the bottom row, and there aren't any nodes found to the right of the empty positions:



A heap, then, is a tree that meets the heap condition and is also complete. Here's one more example of a heap:



This is a valid heap since each node is greater than any of its descendants, and the tree is also complete. While it does have some gaps in the bottom row, these gaps are limited to the very right of the tree.

## Heap Properties

While the heap condition dictates that the heap be ordered a certain way, this ordering is still useless when it comes to searching for a value within the heap.

For example, let's say that in the above heap, we want to search for the value 3.

If we start at the root node of 100, should we search among its left or right descendants? in a binary search tree, we'd know that the 3 must be among the 100's left descendants.

In a heap, however, all we know is that the 3 has to be a descendant of the 100 and can't be its ancestor. But we'd have no idea as to which child node to search next.

Because of this, heaps are said to be *weakly ordered* as compared to binary search trees.

While heaps have *some* order, as descendants cannot be greater than their ancestors, this isn't enough order to make heaps worth searching through.

Another property of heaps, that may be obvious by now but is worth calling attention to, is in a heap, the root node will always have the greatest value.

This will be the key as to why the heap is a great tool for implementing priority queues.

In the priority queue, we always want to access the value with the greatest priority.

With a heap, we always know that we can find this in the root node. Thus, the root node will represent the item with the highest priority.

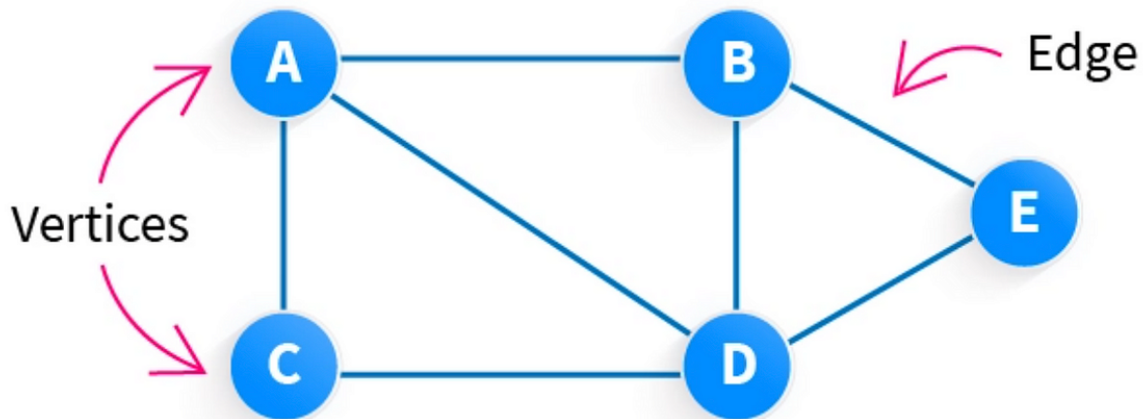
---

## Graphs

A graph is a data structure that consists of nodes (which are called **vertices**) connected by lines or links (which are known as **edges**). Unlike trees which have a hierarchical structure, graphs can represent more complex relationships where



connections can exist between any nodes. This versatile structure makes graphs ideal for modeling real-world networks, such as social connections, road systems, or computer networks.



Graphs can be directed (edges have a specific direction) or undirected (edges flow both ways), and they can be weighted (edges have assigned values) or unweighted. These properties make graphs incredibly flexible for solving complex problems like finding the shortest path between two points or analyzing network connectivity.

Leaf nodes are nodes in a graph that have no outgoing edges, meaning they are endpoints in the graph structure. So, in essence, leaf nodes are nodes that have no children. Unlike in trees where leaf nodes are always at the bottom level, graph leaf nodes can appear anywhere in the structure as long as they have no further connections leading from them.

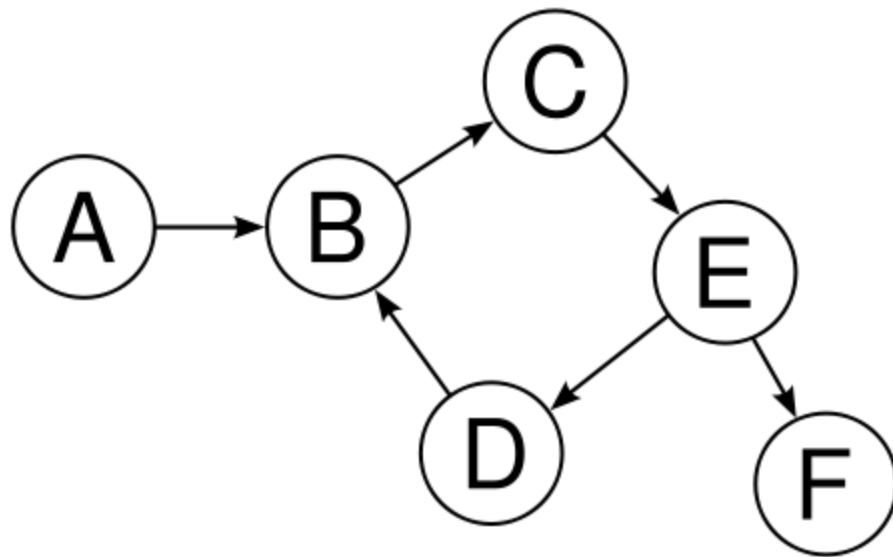
I call these nodes, but their official name are 'vertices' (as shown on the screenshot) when speaking about graphs.

Graph terminology:

- Vertex (Node): A fundamental unit in a graph that represents a point or entity
- Edge: A connection or link between two vertices that shows their relationship
- Path: A sequence of vertices connected by edges

- Degree: The number of edges connected to a vertex
- Adjacent: Two vertices that are directly connected by an edge

A **Circular** or **Cyclic Graph** means that a vertex can connect back to itself through a self-loop. This creates a circular reference where a node has an edge pointing back to itself. Self-loops are important in representing certain types of relationships in graph structures, such as recursive processes or self-referential data.



Time complexity for basic graph operations varies depending on the representation and operation type. For adjacency lists, adding a vertex is  $O(1)$ , while adding an edge is  $O(1)$  on average. Searching for an edge between two vertices takes  $O(V)$  time in the worst case, where  $V$  is the number of vertices.