Algorithms & Data Structures Analysis

Cheatsheet

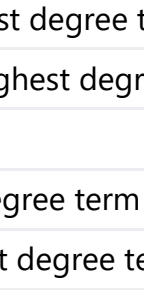
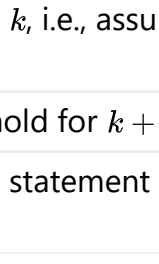
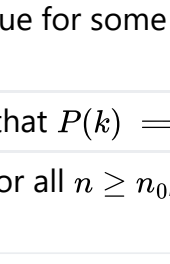
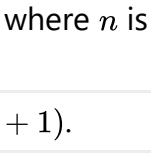
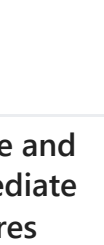
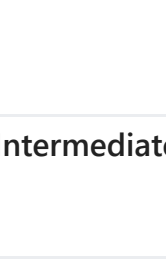
Asymptotic Analysis

Notation	Meaning
$f(n) = O(g(n))$	Highest degree term of $f(n)$ is less than or equal to highest degree term of $g(n)$
$f(n) = \Omega(g(n))$	Highest degree term of $f(n)$ is greater than or equal to highest degree term of $g(n)$
$f(n) = \Theta(g(n))$	Highest degree terms of $f(n)$ and $g(n)$ are equal
$f(n) = o(g(n))$	Highest degree term of $f(n)$ is strictly less than highest degree term of $g(n)$
$f(n) = \omega(g(n))$	Highest degree term of $f(n)$ is strictly greater than highest degree term of $g(n)$

Induction

Step	Description
1. Base Case	Prove that the statement holds for the smallest value of n, usually $n = 0$ or $n = 1$, depending on the problem.
2. Inductive Hypothesis	Assume that the statement holds for some arbitrary value k, i.e., assume $P(k)$ is true for some $k \geq n$, where n is the base case value.
3. Inductive Step	Prove that if the statement holds for k, then it must also hold for $k + 1$, i.e., show that $P(k) \implies P(k + 1)$.
4. Conclusion	Therefore, by the principle of mathematical induction, the statement $P(n)$ holds for all $n \geq n_0$, where n_0 is the base case value. □

AVL Rotations

Imbalance Type	Condition	Rotations Needed	Example and Intermediate Structures	Intermediate	Final Structure
LL (Left-Left)	Left subtree of left child	Single Right Rotation at Node			
RR (Right-Right)	Right subtree of right child	Single Left Rotation at Node			
LR (Left-Right)	Right subtree of left child	Left Rotation at Left Child, then Right Rotation at Node			
RL (Right-Left)	Left subtree of right child	Right Rotation at Right Child, then Left Rotation at Node			

Imbalance Type	Condition	Balance Factor of Node	Balance Factor of Node's Child	Rotations Needed
LL (Left-Left)	Left subtree of left child is deeper	Balance Factor: +2	Balance Factor of Left Child: +1	Single Right Rotation at Node
RR (Right-Right)	Right subtree of right child is deeper	Balance Factor: -2	Balance Factor of Right Child: -1	Single Left Rotation at Node
LR (Left-Right)	Right subtree of left child is deeper	Balance Factor: +2	Balance Factor of Left Child: -1	Left Rotation at Left Child, then Right Rotation at Node

Imbalance Type	Condition	Balance Factor of Node	Balance Factor of Node's Child	Rotations Needed
RL (Right-Left)	Left subtree of right child is deeper	Balance Factor: -2	Balance Factor of Right Child: +1	Right Rotation at Right Child, then Left Rotation at Node

NOTE: When performing a x rotation, if a child node has two children, then the x child becomes the child of the root, where x can be left or right.

Algorithms

Algorithm	Description	Complexity	Key	General Process	Special Considerations
Recursive Multiplication	Recursive multiplication algorithm that breaks the problem into smaller subproblems	$O(n^2)$	n: number of digits	Split numbers into halves, recursively calculate four partial products, add the partial products	Requires combining four partial products, less efficient than Karatsuba's method
Karatsuba's Algorithm	Efficient algorithm for multiplying two n-digit numbers	$O(n^{\log_2(3)})$	n: number of digits	Split numbers into halves, find partial 3 products, add partial products	Uses 3 partial products instead of 4
Counting Sort	Sorting algorithm that counts the number of objects having distinct key values	$O(n+k)$	n: number of elements, k: maximum value of the input range (e.g., for input range [0, 10], k = 10)	Count the occurrences of each distinct element, then use the counts to determine the positions of each element in the sorted array	Only works for integer keys with a small range
Radix Sort	Sorting algorithm that sorts data with integer keys by grouping keys by individual digits	$O(d(n+k))$	n: number of elements, k: range of each digit (e.g., for decimal digits, k = 10), d: maximum number of digits	Sort elements by each digit, starting from the least significant digit to the most significant	Only works for integer keys
Randomised Selection Algorithm	Selects the k-th smallest element from an unsorted list	$O(n)$ average case, $O(n^2)$ worst case	n: number of elements	Partition the list around a randomly selected pivot, recursively search the appropriate sublist	Worst case is extremely unlikely
Deterministic Selection Algorithm	Selects the k-th smallest element from an unsorted list	$O(n)$	n: number of elements	Divide the list into groups of 5, find the median of medians, partition the list around the median of medians, recursively search	Guaranteed $O(n)$ worst case, but more complex to implement and mostly theoretical
Kosaraju's Algorithm	Finds strongly connected components in a directed graph	$O(V+E)$	V: number of vertices, E: number of edges	Perform DFS on the graph, store an array of visited nodes. Reverse the edges on the graph, then DFS again from the last node visited from the initial DFS	-
Dijkstra's Algorithm	Finds the shortest path from a single source to all other nodes	$O((V+E)\log V)$	V: number of vertices, E: number of edges	Maintain a priority queue of vertices, greedily select the vertex with the smallest distance	Cannot handle negative edge weights
Bellman-Ford Algorithm	Finds the shortest path from a single source to all other nodes (handles negative edges)	$O(VE)$	V: number of vertices, E: number of edges	Relax all edges $V-1$ times	Can detect negative cycles
Floyd-Warshall Algorithm	Finds the shortest path between all pairs of nodes	$O(V^3)$	V: number of vertices	Dynamic programming approach, update the	Finds shortest paths between all pairs

Algorithm	Description	Complexity	Key	General Process	Special Considerations
				shortest path matrix in phases	
Kruskal's Algorithm	Finds a minimum spanning tree in a weighted, connected graph	$O(E \log E)$	E: number of edges	Sort edges by weight, greedily add edges to the MST that don't create cycles using UF Data structure	Better for sparse graphs with fewer edges
Prim's Algorithm	Finds a minimum spanning tree in a weighted, connected graph	$O(E \log V)$	V: number of vertices, E: number of edges	Maintain a priority queue of vertices, greedily add the vertex with the smallest edge weight to the MST	Better for dense graphs with more edges

Data Structures

Data Structure	Operation	Average Case	Worst Case	Best Case	Special Considerations
Binary Search Trees	Search	$O(\log n)$	$O(n)$	$O(\log n)$	Worst case occurs when the tree is unbalanced, resembling a linked list. Balancing techniques like AVL trees and Red-Black trees can maintain $O(\log n)$ height, ensuring optimal search, insertion, and deletion times in the worst case.
	Insertion	$O(\log n)$	$O(n)$	$O(\log n)$	
	Deletion	$O(\log n)$	$O(n)$	$O(\log n)$	
AVL Trees	Search	$O(\log n)$	$O(\log n)$	$O(\log n)$	AVL trees are self-balancing binary search trees that maintain a height difference of at most 1 between the left and right subtrees of any node. This self-balancing property ensures a height of $O(\log n)$ in the worst case, providing optimal search, insertion, and deletion times.
	Insertion	$O(\log n)$	$O(\log n)$	$O(\log n)$	
	Deletion	$O(\log n)$	$O(\log n)$	$O(\log n)$	
Skip Lists	Search	$O(\log n)$	$O(n)$	$O(\log n)$	Skip lists are probabilistic data structures that maintain a hierarchy of linked lists, with each higher level being a "skip" of the lower levels. The number of levels for each element is determined randomly, with a probability of 1/2 for each level. This random balancing ensures an expected $O(\log n)$ height, providing efficient search, insertion, and deletion operations. However, the worst-case height can be $O(n)$ with a low probability. Skip lists are useful in scenarios where the simplicity and ease of implementation are prioritized over strict worst-case guarantees.
	Insertion	$O(\log n)$	$O(n)$	$O(\log n)$	
	Deletion	$O(\log n)$	$O(n)$	$O(\log n)$	
Hash Tables	Search	$O(1)$	$O(n)$	$O(1)$	Hash tables provide constant-time average-case operations by using a hash function to map keys to array indices. Collisions occur when multiple keys hash to the same index. Hash tables resolve collisions using either: 1. Separate chaining: Colliding elements are stored in linked lists at each index. 2. Linear probing: Probes the next sequential element until an empty slot is found. Worst case: All keys hash to the same index, resulting in $O(n)$ time complexity.
	Insertion	$O(1)$	$O(n)$	$O(1)$	
	Deletion	$O(1)$	$O(n)$	$O(1)$	

Graphs

Type	Description	Space Complexity	Time Complexity	Advantages	Disadvantages	Examples of Use
Adjacency List	Represents a graph using a list or array of lists. Each vertex has a list of its adjacent vertices.	$O(\|\|V\|\| + \|\|E\|\|)$	- Add Edge: $O(1)$ - Remove Edge: $O(\|\|V\|\|)$ - Check Edge: $O(\|\|V\|\|)$	- Space-efficient for sparse graphs - Faster for iterating over edges	- Slower for checking edge existence - Slower for dense graphs	- Social networks (e.g., friends list) - Web crawlers (links between web pages)
Adjacency Matrix	Represents a graph using a matrix. The element at index (i, j) represents an edge from vertex i to vertex j.	$O(\|\|V\|\|^2)$	- Add Edge: $O(1)$ - Remove Edge: $O(1)$ - Check Edge: $O(1)$	- Faster for checking edge existence - Easier to implement	- Consumes more space - Slower for iterating over edges	- Storing distances between cities - Representing a chessboard (edges between squares)
Adjacency Array (Unweighted)	Represents a graph using an array of integers. Each vertex is assigned an index, and the array stores the indices of its adjacent vertices.	$O(\|\|V\|\| + \|\|E\|\|)$	- Add Edge: $O(1)$ - Remove Edge: $O(\|\|E\|\|)$ - Check Edge: $O(\|\|E\|\|)$	- Space-efficient - Fast for iterating over edges	- Slower for checking edge existence - Requires contiguous memory	- Game maps (e.g., adjacent rooms) - Character relationships in a story
Adjacency Array (Weighted)	Similar to unweighted adjacency array, but each vertex has an additional array to store edge weights.	$O(\|\|V\|\| + \|\|E\|\|)$	- Add Edge: $O(1)$ - Remove Edge: $O(\|\|E\|\|)$ - Check Edge: $O(\|\|E\|\|)$	- Space-efficient - Fast for iterating over edges - Supports weighted edges	- Slower for checking edge existence - Requires contiguous memory	- Road networks with distances - Flight routes with costs
Linked List	Represents a graph using a linked list of vertices. Each vertex contains a pointer to a linked list of its adjacent vertices.	$O(\|\|V\|\| + \|\|E\|\|)$	- Add Edge: $O(1)$ - Remove Edge: $O(\|\|V\|\|)$ - Check Edge: $O(\|\|V\|\|)$	- Dynamic memory allocation - Efficient for adding/removing vertices	- Slower for checking edge existence - More complex implementation	- Family trees (adding/removing family members) - Version control systems (branching and merging)

P & NP

Concept	Description	Examples	Implications
P (Polynomial Time)	Problems solvable in polynomial time, i.e., time complexity is $O(n^k)$, where k is a constant.	- Linear search - Euler path - Shortest path (Dijkstra's algorithm) - Maximum flow (Ford-Fulkerson algorithm)	- Efficiently solvable - Deterministic algorithms exist
NP (Non-deterministic Polynomial Time)	Problems verifiable in polynomial time, but not necessarily solvable in polynomial time.	- Traveling Salesman Problem - Subset Sum - Graph Coloring - Boolean Satisfiability (SAT)	- No known polynomial-time solution - Brute-force may be required
NP-Hard	Problems at least as hard as the hardest problems in NP. A problem is NP-Hard if all other NP problems can be reduced to it in polynomial time, but it may not be in NP itself.	- Halting Problem (undecidable) - Integer Factorization - Traveling Salesman Problem (optimization version) - Graph Coloring (optimization version)	- Solving an NP-Hard problem efficiently would mean $P = NP$ - NP-Hard problems may not have polynomial-time verification
NP-Complete	Hardest problems in NP. A problem is NP-Complete if it is in NP and all other NP problems can be reduced to it in polynomial time.	- Boolean Satisfiability (SAT) - Knapsack Problem - Hamiltonian Cycle - Vertex Cover	- Solving an NP-Complete problem efficiently would mean $P = NP$ - No polynomial-time

Concept	Description	Examples	Implications
			algorithm known for any NP-Complete problem

- $P \subseteq NP$: All problems in P are also in NP, but it is unknown whether $P = NP$.
- If $P = NP$, all problems in NP would be solvable in polynomial time, which would have significant implications for cryptography, optimization, and algorithm design.
- Boolean Satisfiability (SAT) is the first problem proven to be NP-Complete. It asks whether a Boolean expression can be satisfied by some assignment of Boolean values to its variables.

Notes

1.1 Asymptotic Notation

The Idea

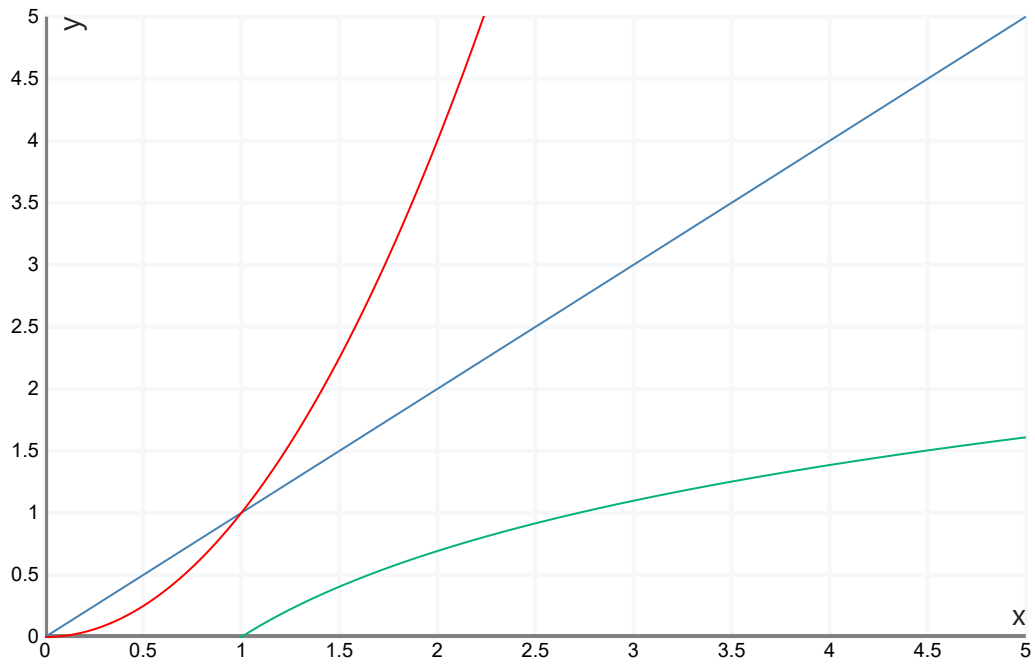
When we discuss an algorithm's time complexity, we often represent it as a function $T(n)$, where n is the size of the input. As N increases, the execution time of the algorithm typically increases as well. *The goal of asymptotic notation is to provide ways to describe the behaviour of $T(n)$ as n approaches infinity.* This helps in comparing algorithms by providing a high-level understanding of their efficiency in terms of time and space.

Bound	Symbol	Mathematical Description	Logical Description
Upper Bound	O (big oh)	$T(n)\ is\ O(f(n)) \iff T(n) \leq c \cdot f(n)\ \forall\ n > n_0$	An algorithm $T(n)$ is upper bounded by a function $f(n)$ if and only if there exists a constant c whereby $T(n)$ is smaller than $c \cdot f(n)$ for every single n bigger than the initial n
Lower Bound	(Ω) (big omega)	$T(n)\ is\ \Omega(f(n)) \iff c \cdot f(n) \leq T(n)\ \forall\ n > n_0$	An algorithm $T(n)$ is lower bounded by a function $f(n)$ if and only if there exists a constant c whereby $T(n)$ is greater than or equal to $c \cdot f(n)$ for every n larger than some initial n
Tight Bound	(Θ) (theta)	$T(n)\ is\ \Theta(f(n)) \iff c_1 \cdot f(n) \leq T(n) \leq c_2 \cdot f(n)\ \forall\ n > n_0$	An algorithm $T(n)$ is tightly bounded by a function $f(n)$ if and only if there exist constants c_1 and c_2 whereby $T(n)$ is sandwiched between $c_1 \cdot f(n)$ and $c_2 \cdot f(n)$ for every n larger than some initial n
Stricter Upper Bound	o (little oh)	$T(n)\ is\ o(f(n)) \iff \lim_{n\to\infty} \frac{T(n)}{f(n)} = 0$	An algorithm $T(n)$ grows strictly slower than $f(n)$ if and only if the ratio of $T(n)$ to $f(n)$ approaches 0 as n becomes infinitely large.
Stricter Lower Bound	ω (little omega)	$T(n)\ is\ \omega(f(n)) \iff \lim_{n\to\infty} \frac{T(n)}{f(n)} = \infty$	An algorithm $T(n)$ grows strictly faster than $f(n)$ if and only if the ratio of $T(n)$ to $f(n)$ becomes infinitely large as n grows without bound.

Graphically

Upper Bound - If we have two functions $T(n) = n$ and $f(n) = n^2$, we can say that $T(n)$ is upper bounded by $f(n)$.

Lower Bound - What happens when we can't find a constant c such that $c \cdot f(n)$ is greater than $f(n)$? If we have two functions $T(n) = n$ and $f(n) = log(n)$, we can say that $T(n)$ is lower bounded by $f(n)$.



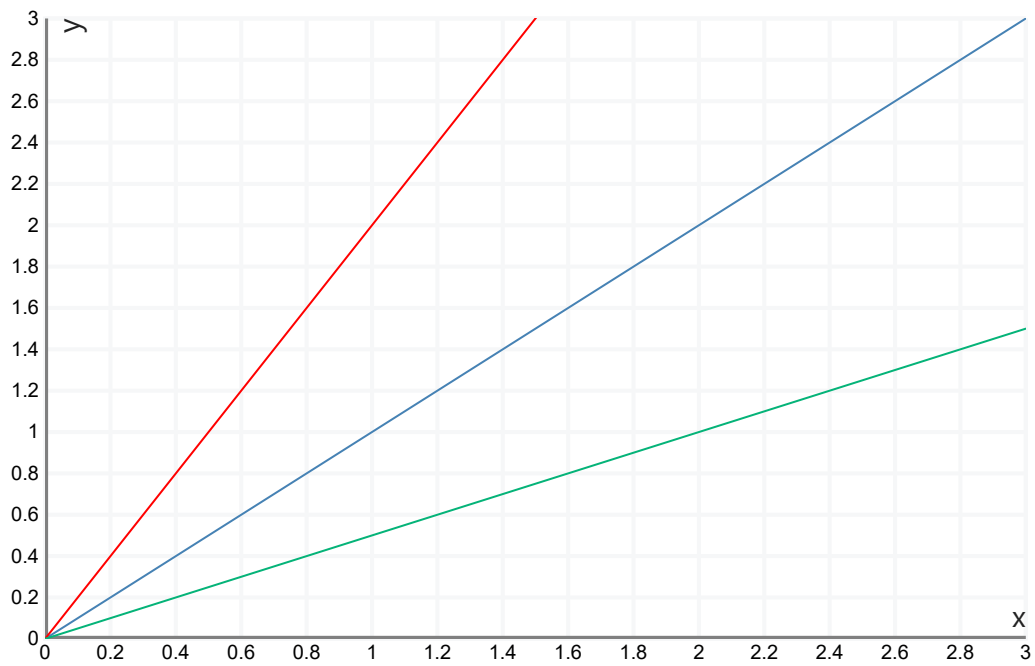
Consider:

When evaluating the efficiency of an algorithm, it's crucial to consider how **tight** our bound is. Although multiple functions can serve as upper and lower bounds for $T(n)$, not all of them provide an accurate representation of the algorithm's efficiency. A *tighter* bound, which closely matches the actual growth rate of $T(n)$, offers a more precise understanding of the algorithm's performance. It is *both* an upper AND lower bound at the same time.

Graphically

Tight Bound - If we have two functions $T(n) = n$ and $f(n) = n$, we can say that $T(n)$ has a tight bound $f(n)$, because there exists two constants c_1 and c_2 that sandwich $T(n)$

There also exists two other bounds, which describe upper and lower bounds that are **NOT** tight.



1.1.1

Exercise 1 - Determine the correctness of the following statements:

Statement	In English	Answer	Explanation
`5n log n ∈ O(n log n)`	`n · log(n)` is an upper bound for `5n · log(n)`	True	Dropping constants, it's the same function. Therefore, it's a tight bound and also an upper bound.
`5n log n ∈ O(n^2)`	`n^2` is an upper bound for `5n · log(n)`	True	As `n` grows, `n^2` increases faster than `n log n`, making `n^2` an upper bound.
`5n log n ∈ Ω(n^2)`	`n^2` is a lower bound for `5n · log(n)`	False	`n log n` grows slower than `n^2`, so `n^2` cannot be a lower bound for `5n log n`.
`5n log n ∈ o(n^2)`	`5n · log(n)` grows strictly slower than `n^2`	True	For very large `n`, `n^2` grows much faster than `n log n`, so `5n log n` is in `o(n^2)`.
`5n log n + n^2 ∈ O(n log n)`	`n · log(n)` is an upper bound for `5n · log(n) + n^2`	False	`n^2` term dominates as `n` grows, so `n log n` cannot be an upper bound for the sum.

Statement	In English	Answer	Explanation
`5n log n + n^2 ∈ O(n^2)`	`n^2` is an upper bound for `5n · log(n) + n^2`	True	`n^2` is the highest order term, making it the tight bound and thus also an upper bound.

1.1.2

Exercise 2 - Find the bound for each function

Function	Upper Bound	Lower Bound	Tight Bound	Strict Upper Bound	Strict Lower Bound
$5n$	$O(n)$	$\Omega(n)$	$\Theta(n)$	$o(n \log n)$	$\omega(\sqrt{n})$
$n^2 - n \log n$	$O(n^2)$	$\Omega(n^2)$	$\Theta(n^2)$	$o(n^3)$	$\omega(n \log n)$
$100n$	$O(n^2)$	$\Omega(\sqrt{n})$	$\Theta(n)$	$o(n \log n)$	$\omega(\sqrt{n})$

1.1.3

Exercise 3

The expression for the worst case runtime of the school method of multiplication is given by f(n) = 3n^2 + 2n. Which of the following notations can best describe the worst-case runtime complexity of this algorithm?

- A: $O(n^2)$ - This notation represents an upper bound on the time complexity of the algorithm. Since the highest order term in the given expression is n^2, this notation correctly describes the growth rate of the algorithm's running time.
- B: $\Omega(n^2)$ - Omega notation represents a lower bound on the time complexity. Given that the n^2 term will dominate the runtime for large n, this also accurately represents the algorithm's complexity.
- C: $\Theta(n^2)$ - Theta notation represents a tight bound, meaning it's both an upper and lower bound. Since the leading term is n^2 and the coefficients do not change the order of growth, this is the correct tight bound for the complexity.
- D: All of the above - Given that A, B, and C are all correct, the best description for the worst-case runtime complexity of the algorithm is indeed all of the above.

1.2 Asymptotic Analysis

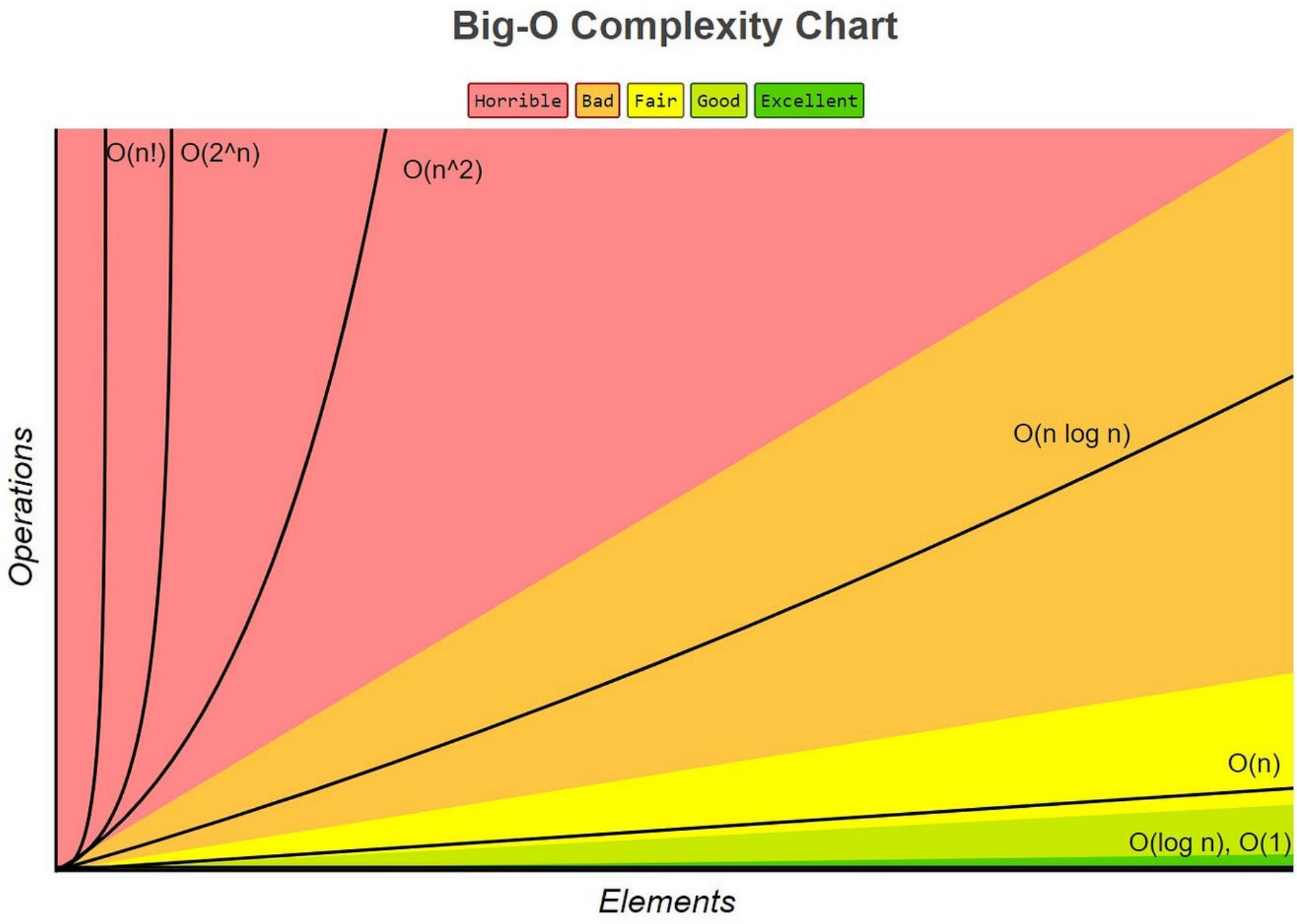
The Idea

1. Complexity of algorithm is a function of it's input size n.
2. Complexity is machine independent
3. Complexity measures basic computer steps
4. Complexity encompasses both time & space
5. Complexity measures best, average and worst cases

When figuring out the complexity of a given algorithm, we must follow some rules:

Rule	Example	Explanation
Ignore Constants	$5n \rightarrow O(n)$	Ignore the 5
Only worry about highest order term	$3n^3 + 5n^2 + 10n + 20 \rightarrow O(n^3)$	n^3 is the highest order, so we ignore everything else

The following picture describes effectiveness of the most common complexities.



1.2.1

Exercise 1 - Determine the complexity of the following code segments:

Type	Example	Complexity	Explanation
Constant	`x = 5 + (15 * 20)`	$O(1)$	Independent of input size N

	`x = 5 + (15 * 20)`
	`y = 15-2`
	`print x+y`
		$O(1) + O(1) + O(1)$
		$= 3 \cdot O(1)$
		$= O(1)$
			Add each of their times, ignore the constants

Linear	`for x in range (0, n):` `  print x  // O(1)`	$N \cdot O(1)$ $= O(N)$	The print statement is $O(1)$ and we do it N times

	`y = 5 + (15 * 20)`
	`for x in range (0, n):`
	`  print x  // O(1)`
		$O(1) + O(n)$
		$= O(N)$
			The summation of each step and we ignore the lower order terms

Quadratic	`for x in range (0, n):`
	`  for y in range (0, n):`
	`  print x*y  \\ O(1)`
		$O(N^2)$	The print statement is executed $N \cdot N$ times

1.2.2

Exercise 2 - Determine the complexity of the following code segment:

```
x = 5 + (15 * 20)                                   # O(1)
n=10
for x in range (0,n):
        print(x)                                        # O(N)
for x in range (0,n):
        for y in range (0,n):
                print(x*y)                                  # O(N^2)
```

#	Step	Working Out
1	Take the summation of each of the run times	$O(1) + O(N) + O(N^2)$
2	Find the highest order term	$O(N^2)$

1.2.3

Exercise 3 - Determine the complexity of the following code segment:

```
if x > 0:
        # O(1)
else if x < 0:
        # O(logn)
else:
        # O(n^2)
```

#	Step	Working Out
1	Take the largest run time (since we want worst case)	$O(N^2)$

1.2.4

Exercise 4

Imagine you have developed an efficient algorithm for a Mars rover to calculate the shortest path between its current location and a designated target point. Your algorithm runs very fast in the Lab tests. However, to be considered for deployment on Mars, NASA requires a comprehensive evaluation of the algorithm's performance. Why type of guarantee needs to be provided?

1. Provide the guarantee with actual run-times that the algorithm will always run fast as it has shown excellent run time in the Lab, ensuring energy conservation.
2. Provide the worst-case complexity analysis of the algorithm can handle the most complex navigation tasks without exceeding time and energy constraints.
3. Providing simulation results showing that the algorithm runs faster in Martian conditions.
4. Provide the best-case complexity, because NASA only uses algorithms that perform well in lab tests, to ensure a margin of safety.

Answer: Option 2, Provide the worst case complexity analysis of the algorithm. This is a guarantee, that no matter the circumstances, the algorithm will always at least run this fast.

1.3 School Method Arithmetic

Rule

When adding two integers of different lengths, the number of primitive operations required is determined by the length of the longer integer. This is because each digit of the shorter integer is added to the corresponding digit of the longer integer, and any additional digits in the longer integer are also processed to account for carries. Therefore, the total number of operations equals the number of digits in the longer integer, which can be described as `max(a,b)` primitive operations, where `a` and `b` are the lengths of the two integers.

As for the number of digits in the output, the result of adding two integers can have at most `max(a,b) + 1` digits. This maximum occurs when a carry is generated from the addition of the most significant digits of the integers. In summary, the length of the result is determined by the longer of the two integers, with the possibility of an additional digit if a carry is generated in the final addition operation.

Theorem - School Method Addition

The addition of two n-digit integers requires exactly n primitive operations. The result is at most, an n+1 digit integer.

	1	7	0	9
$+$	2	5	3	0
carries	1	0	0	0
sum	4	2	3	9

In code, we implement this as follows:

```
a=number1
b=number2
n=numberofdigits

carry=0
for i=0 to n-1 do
        s[i] = a[i] + b[i]
        carry = s[i] / base
        s[i] = s[i] % base
end for
s[n] = carry
```

Theorem - School Method Multiplication

The school method multiplication of two n-digit integers requires $3n^2 + 2n = n^2$ primitive operations. It results in at most, a $2n$ digit integer.

$$\begin{array}{r} 1709 \\ \times\ 25 \\ \hline 8545\ (\text{partial product } a \cdot b_0) \\ 34180\ (\text{partial product } a \cdot b_1) \\ \hline 42725\ (\text{add aligned partial products}) \end{array}$$

In pseudo code, this can be represented as follows:

```
a=number1
b=number2
n=numberofdigits

product=0

for i=0 to n-1 do
        product = product + a * b[i] * base^[i]
end for
```

We already have a pretty fast algorithm to compute addition, but can we make a faster multiplication algorithm than $O(n^2)$?

Number Representation

Number Representation

$$\text{Any digit } d \text{ of any base } B, \text{ can be represented in decimal by:} \quad d = \sum_{i=0}^{n-1} d_i B^i$$

Base (B)	Expression	Decimal Value
2 (binary)	$1 \cdot 2^3 + 0 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0$	11
8 (octal)	$5 \cdot 8^3 + 3 \cdot 8^2 + 4 \cdot 8^1 + 7 \cdot 8^0$	2791
10 (decimal)	$1 \cdot 10^3 + 7 \cdot 10^2 + 0 \cdot 10^1 + 9 \cdot 10^0$	1709
16 (hexadecimal)	$A \cdot 16^3 + D \cdot 16^2 + 5 \cdot 16^1 + A \cdot 16^0$	44378

This formula is essentially saying, for every digit in a number, multiply each of them by the base to the power of the position of the digit and sum each result.

2.1 Standard Multiplication Algorithm

The Idea

Reduce complexity of school method multiplication through Divide & Conquer! Let a and b be the numbers to be multiplied.

Step 1: Split the integers

$$\begin{aligned} a &= \overbrace{12}^{a_1}\ \overbrace{34}^{a_0} \\ b &= \underbrace{56}_{b_1}\ \underbrace{78}_{b_0} \end{aligned}$$

$$\begin{aligned} n &= digits \\ k &= \frac{n}{2} \end{aligned}$$

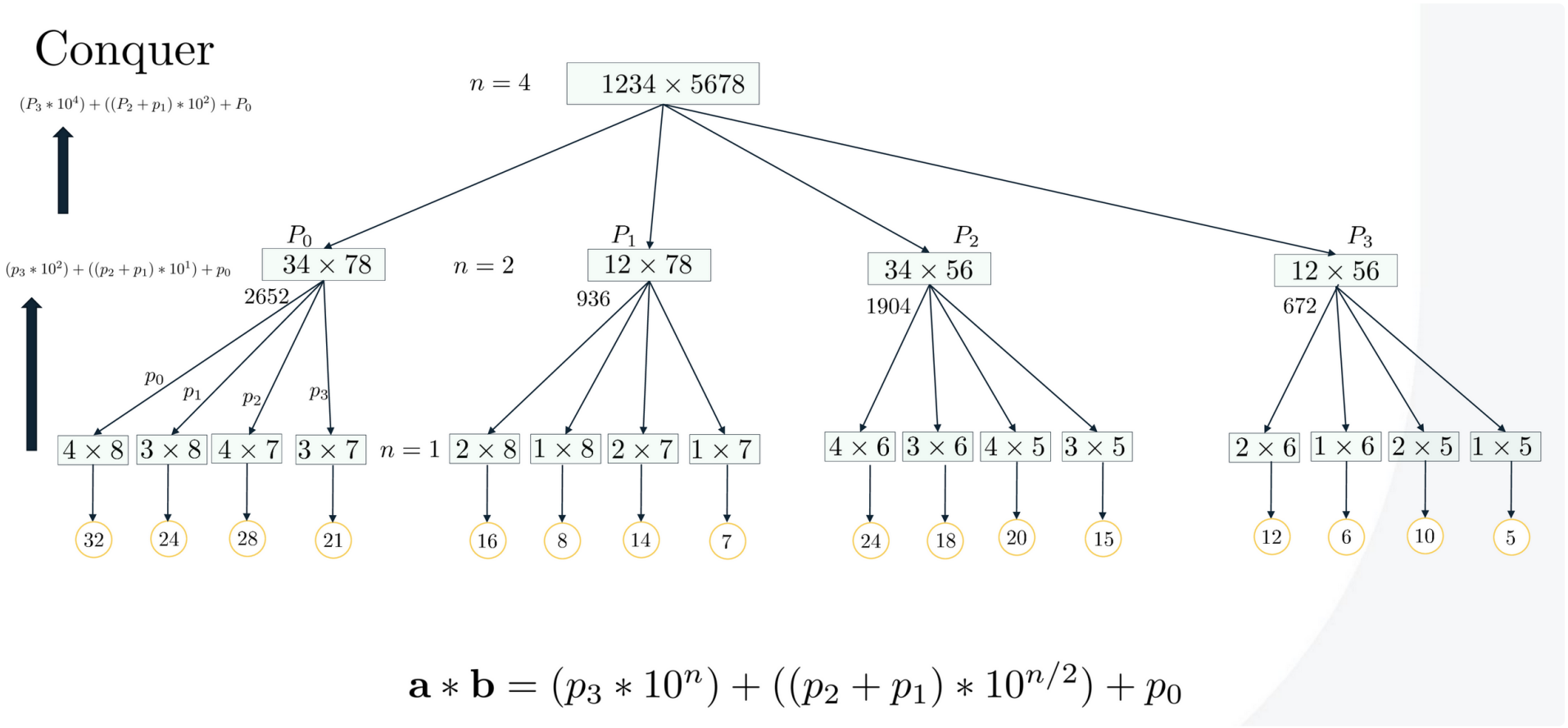
Step 2: Calculate Partial Products

$$\begin{aligned} p_0 &= a_0 \times b_0 = 34 \times 78 = 2652 \\ p_1 &= a_1 \times b_0 = 12 \times 78 = 936 \\ p_2 &= a_0 \times b_1 = 34 \times 56 = 1904 \\ p_3 &= a_1 \times b_1 = 12 \times 56 = 672 \end{aligned}$$

Step 3: Add the Aligned products

$$\begin{aligned} a \times b &= p_3 \times B^{2k} + B^k \times (p_1 + p_2) + p_0 \\ &= 6720000 + 284000 + 2652 \\ &= 7006652 \end{aligned}$$

Notice, that we can split the integers even further and recursively find those multiplications



$$\mathbf{a} * \mathbf{b} = (p_3 * 10^n) + ((p_2 + p_1) * 10^{n/2}) + p_0$$

Pseudocode

```
function multiply(a, b, n) {
        if n == 1:
                return a * b
        else:
                // Split the numbers
                a1, a0 = split(a)
                b1, b0 = split(b)

                // Solve four subproblems using the ceiling of n/2
                p0 = multiply(a0, b0, ceil(n/2))
                p1 = multiply(a1, b0, ceil(n/2))
                p2 = multiply(a0, b1, ceil(n/2))
                p3 = multiply(a1, b1, ceil(n/2))

                // Combine the results
                return p3 *B^2k + B^k(p1 + p2) + p0
```

```
}
```

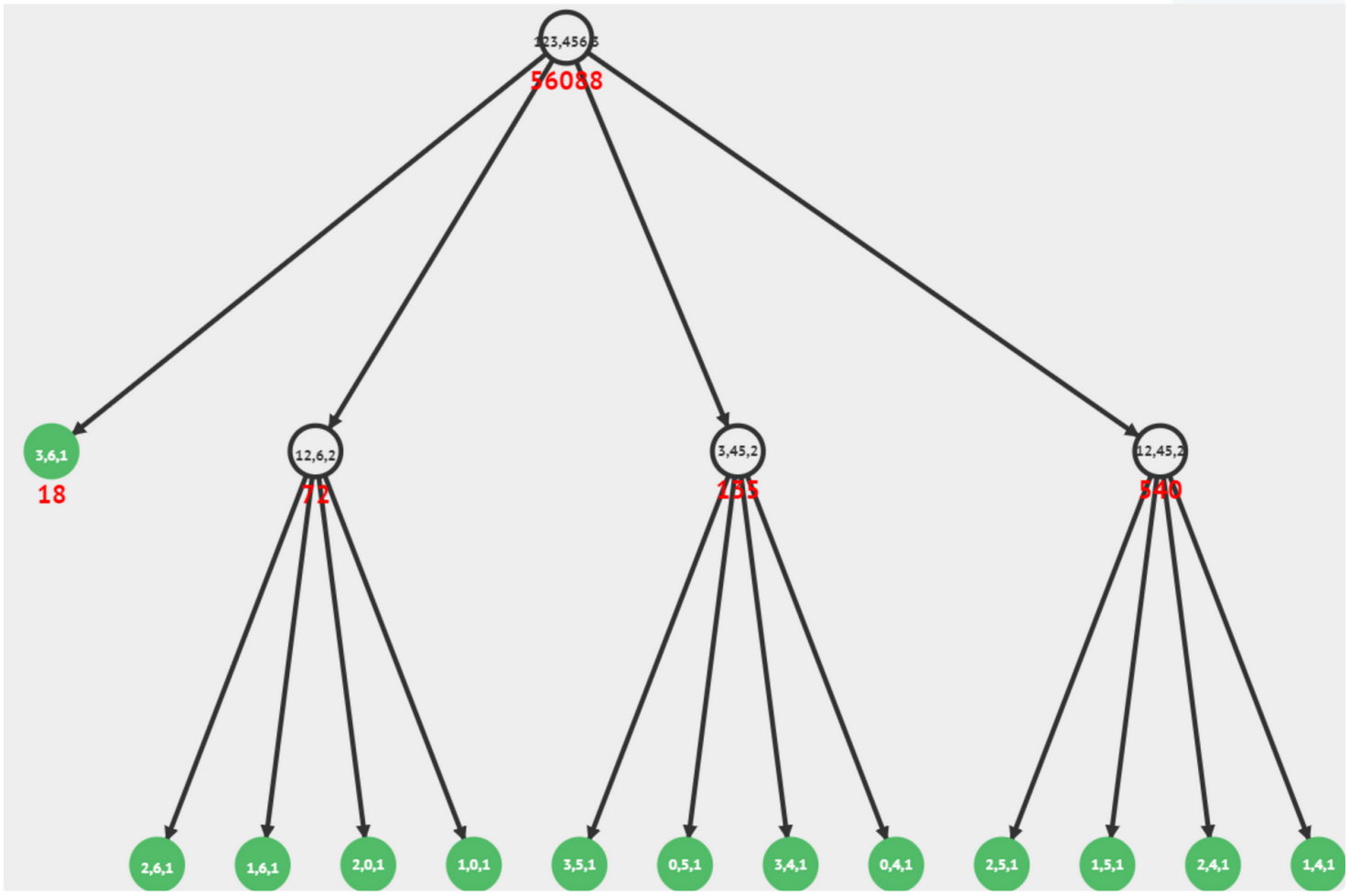
⚠ **Consider the value of n**

If $n \in n^2$, then each of the sub levels will have 4 subproblems. Otherwise, one of the problems will end prematurely.

- Essentially, this is saying when $n \in n^2$, we have the worst case run time of this algorithm.

$$\mathbf{a} = 123 \Rightarrow \mathbf{a} = \overbrace{12}^{a_1}\overbrace{3}^{a_0}$$

$$\mathbf{b} = 456 \Rightarrow \mathbf{b} = \underbrace{45}_{b_1}\underbrace{6}_{b_0}$$



Recurrence Relation of Recursive Multiplication

We can use a 2.3 Recurrence Relations to describe the complexity of this recursive function. Let $T(n)$ be the maximal number of primitive operations to multiply two n-digit numbers recursively.

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 4 \cdot T\left(\lfloor \frac{n}{2} \rfloor\right) + 3 \cdot 2^n & \text{if } n \geq 2 \end{cases}$$

$$\text{For } n \text{ power of } 2 : T(n) \leq 7n^2 - 6n$$

$$\text{For general } n : T(n) \leq 28n^2$$

- It should also be noted that this complexity is actually worse than that seen in the School Method Multiplication

2.1.1

Proving Recursive Multiplication with School Method

Recall the Master Theorem:

$$T(n) \leq \begin{cases} 1 & \text{if } n = 1 \\ 4 \cdot T\left(\lfloor \frac{n}{2} \rfloor\right) + 3 \cdot 2 \cdot n & \text{if } n \geq 2 \end{cases}$$

We find that:

- $a = 4$
- $b = 2$
- $d = 1$

Applying the master theorem, we get:

- $a > b^d \rightarrow 4 > 2^1$

This means our complexity is:

- $O(n^{\log_b a}) \to O(n^{\log_2 4}) \to O(n^2)$

2.2 Karatsuba's Algorithm

The Idea

Recall in the Recursive Multiplication Method, our time complexity is actually not reduced below $O(n^2)$. Enter the Karatsuba Algorithm, a faster Divide and Conquer multiplication algorithm $\approx O(n^{1.59})$.

The method is quite similar to its predecessor, but instead of calculating 4 partial products, we only need 3. On the surface, this single multiplication decrease seems trivial, but we must understand that there is an entire operation removed PER sub level. This greatly reduces the worst-case complexity of the algorithm.

$$
\begin{aligned}
a \times b &= (a_1 B^k + a_0) \times (b_1 B^k + b_0) \\
&= (a_1 b_1) B^{2k} + (a_1 b_0 + a_0 b_1) B^k + a_0 b_0 && \text{(Recursive Method)} \\
&= (a_1 \times b_1) B^{2k} + ((a_1 + a_0) \times (b_1 + b_0) - (a_1 \times b_1 + a_0 \times b_0)) B^k + a_0 \times b_0 && \text{(Karatsuba)}
\end{aligned}
$$

$$
= (a_1 b_1) 10^{2k} \oplus [(a_0 \oplus a_1)(b_0 \oplus b_1) \ominus a_1 b_1 \ominus a_0 b_0)] 10^k \oplus a_0 b_0
$$

Three multiplications Six additions

Karatsuba Pseudocode

```
function karatsuba(a, b, n) {
if n < 4
        return a * b
else
        // Split the numbers
        a1, a0 = split(a)
        b1, b0 = split(b)

        // Solve three sub problems, using the Karatsuba method
        int k = ceil(n / 2);
        p0 = karatsuba(a0, b0, k)
        p1 = karatsuba(a1, b1, k)
        p2 = karatsuba((a0+a1),(b0+b1), k) - p0 - p1

        // Combine the results
        b^2k*p2 + b^k*p1 + p0
}
```

Karatsuba Recurrence Relation

$$
T_K(n) \leq \begin{cases} 3n^2 + 2n & \text{if } n < 4 \\ 3 \cdot T_K\left(\lceil \frac{n}{2} \rceil + 1\right) + 6 \cdot 2 \cdot n & \text{if } n \geq 4 \end{cases}
$$

$$
T_K(n) \leq 207 \cdot n^{\log_3 3}
$$

Exercise: Solve the Karatsuba Recurrence Relation using Substitution

Prove by induction that the solution to the Karatsuba recurrence relation is:

$$
T_K(n) \leq 207 \cdot n^{\log_2 3}
$$

Step 1: Base Case

- Prove that the solution holds for the base case, when $n < 4$.

Step 2: Inductive Hypothesis

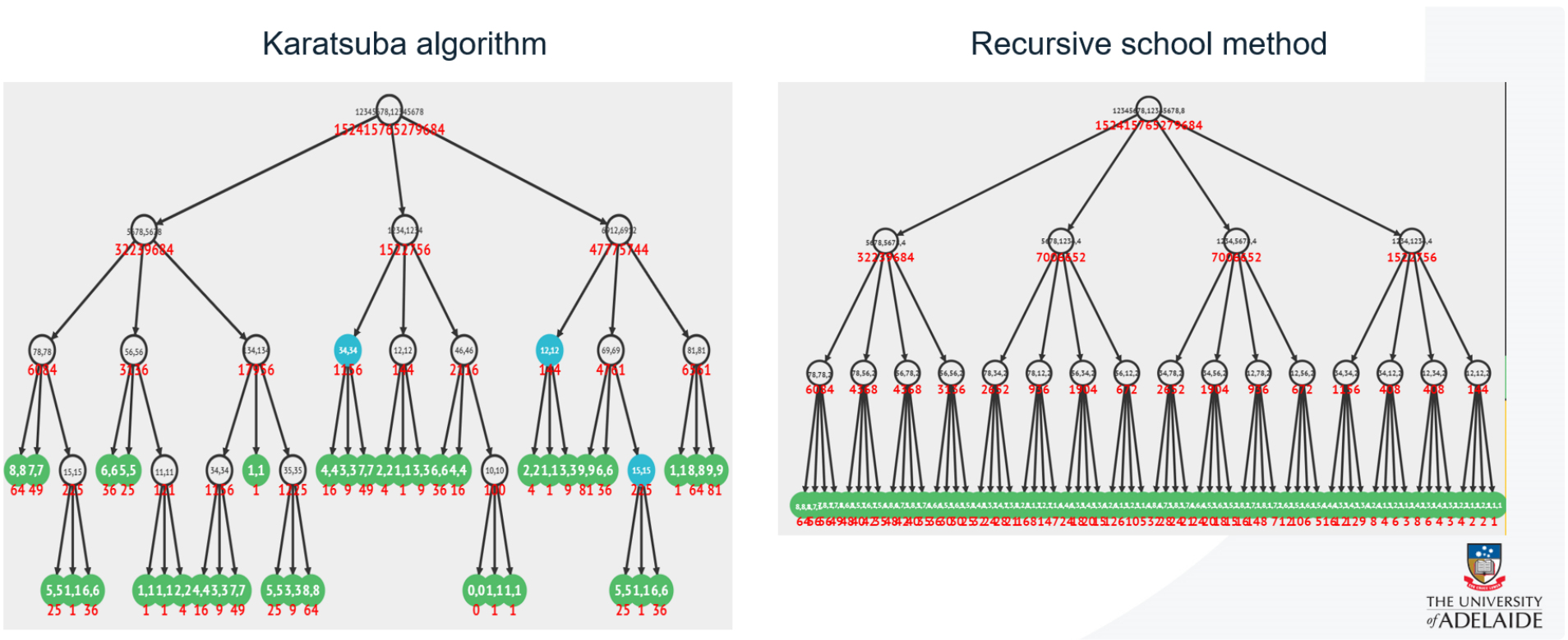
- Assume that the solution holds for all values of n less than some arbitrary value k, where $k \geq 4$.

Step 3: Inductive Step

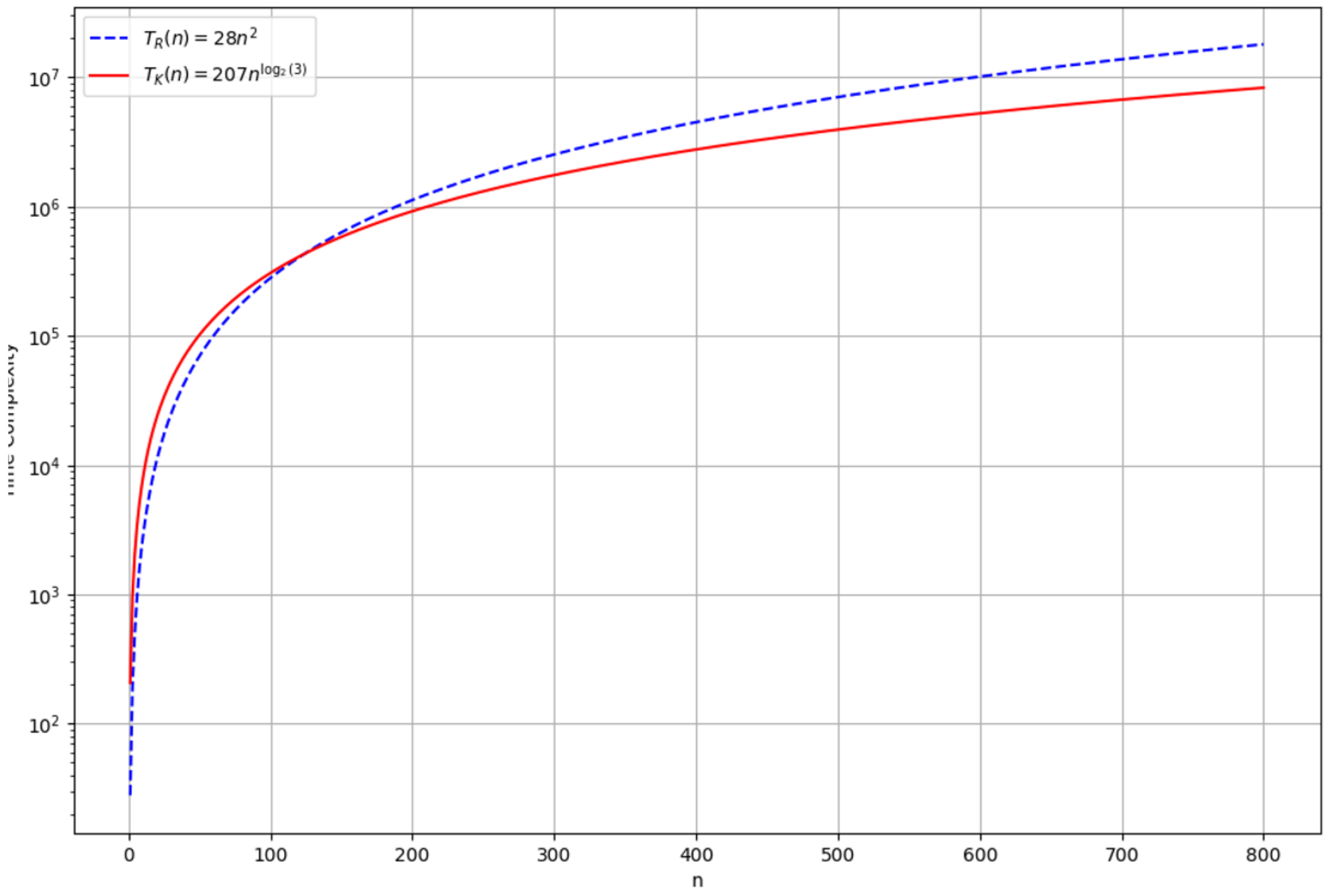
- Prove that if the solution holds for $n < k$, then it also holds for $n = k$.
- Substitute the inductive hypothesis into the recurrence relation for $n \geq 4$ and simplify the expression to show that it holds for $n = k$.

If you can prove the base case and the inductive step, then you have successfully proven that the solution holds for all values of n using the substitution method.

⚠ **Consider - Karatsuba vs Recursive School**



In the Karatsuba method, leaf nodes are reached more quickly compared to the recursive school method. The Karatsuba algorithm has a shallower tree structure with fewer levels, allowing it to arrive at the base cases (leaf nodes) faster.



2.3 Recurrence Relations

Recurrence Relations

Recurrence relations are mathematical equations that describe the running time or space complexity of recursive algorithms by expressing the performance of the algorithm in terms of its input size. They offer a structured way to analyze and predict the efficiency of recursive algorithms as the size of the input data changes.

Recurrence relations are useful for understanding and analyzing various types of recursive algorithms, including those that break down a problem into smaller subproblems, solve these subproblems recursively, and then combine the solutions to solve the original problem (e.g., divide and conquer algorithms). They are also applicable to other recursive paradigms such as decrease and conquer, dynamic programming, and backtracking.

Example - Fibonacci Sequence

Let's find the recurrence relation for the Fibonacci sequence using the code provided:

```
int fibonacci(int n)
{
    if (n == 0)
    {
        return 0;
    }
    else if (n == 1)
    {
        return 1;
    }
    else
    {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

The recurrence relation for the Fibonacci sequence can be written as:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2) + 1, & \text{if } n > 1 \end{cases}$$

This recurrence relation states that the nth Fibonacci number is equal to the sum of the (n-1)th and (n-2)th Fibonacci numbers, plus some work (+1) to combine them, with the base cases of $F(0) = 0$ and $F(1) = 1$.

Example - Factorial

Now, let's find the recurrence relation for the factorial function using the following code:

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }
    else
    {
        return n * factorial(n - 1);
    }
}
```

The recurrence relation for the factorial function can be written as:

$$F(n) = \begin{cases} 1, & \text{if } n = 0 \\ n \cdot F(n-1) + 1, & \text{if } n > 0 \end{cases}$$

This recurrence relation states that the factorial of n (denoted as $n!$) is equal to n multiplied by the factorial of $(n-1)$ plus some work to combine them (+1), with the base case of $F(0) = 1$.

Solving Recurrence Relations

There are many ways to solve recurrence relations, including:

1. Plugging and chugging (iteration)
   - This method involves iteratively expanding the recurrence relation by substituting the formula back into itself until a pattern emerges.
2. Substitution and proof by induction
   - In this approach, you guess a solution for the recurrence relation and then prove its correctness using mathematical induction.
3. Recursive tree method
   - This technique involves drawing a tree that represents the recursive calls and their associated costs. By summing up the costs at each level of the tree, you can determine the overall time complexity.
4. Master theorem
   - The master theorem provides a formula for solving certain types of divide-and-conquer recurrences. It allows you to quickly determine the time complexity of a recursive algorithm based on its structure.
5. Akra-Bazzi method
   - The Akra-Bazzi method is a generalization of the master theorem and can be used to solve a wider range of divide-and-conquer recurrences. It provides a formula for solving recurrences of the form:

$$T(n) = \sum_{i=1}^{k} a_i T(\frac{n}{b_i}) + f(n)$$

where $a_i > 0$, $b_i > 1$, and $f(n)$ is a positive function.

2.4 Master Theorem

The Idea

The Master Theorem provides a way to determine the asymptotic behaviour of $T(n)$, which is the time complexity of a Divide & Conquer algorithm.

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

$$T(n) \in \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Solution	Explanation
$a < b^d$	If $a < b^d$: The work done at the root of the recursion tree is the most significant. The running time is dominated by the work done outside the recursive calls, $O(n^d)$.
$a = b^d$	If $a = b^d$: The work is evenly distributed across the levels of the recursion tree. The running time is a combination of the work done at each level of the recursion and the depth of the recursion, $O(n^d \log n)$.
$a > b^d$	If $a > b^d$: The work done at the leaves of the recursion tree is the most significant. The running time is dominated by the work done at the deepest level of the recursive calls, $O(n^{\log_b a})$.

Term	Meaning
$T(n)$	This is the function we are trying to solve for. It represents the total running time of the algorithm.
a	This is the number of subproblems in the recursion. If an algorithm splits its input data into smaller pieces to solve the problem, a is the count of those pieces.
b	This is the factor by which the subproblem size is reduced. If an algorithm splits the problem into subproblems that are half the size of the original, then $b = 2$.
$f(n)$	This function represents the work done at each level of non-recursive part. It's typically the amount of time it takes to split the problem into subproblems and combine the results of the subproblems.
d	This represents the exponent in the running time of the non-recursive part (often called the "combine step" or "work done outside the recursive calls") of an algorithm. If the non-recursive work takes $O(n^d)$ time, then d is that exponent.

3.1 Comparison Sorts

Sorting So Far

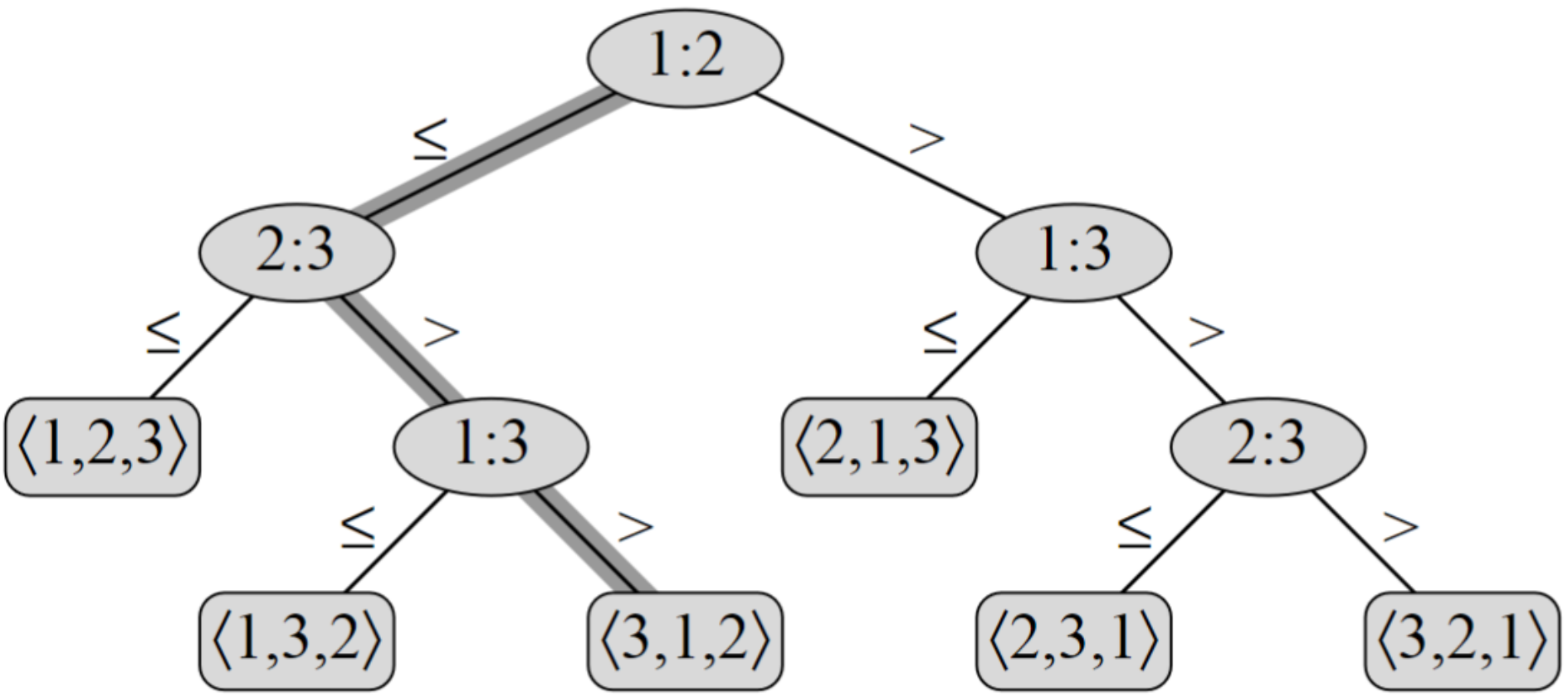
So far, we've seen sorting algorithms that compare each element inside a list to sort them, i.e. The only operation used to gain information about two elements is the pairwise comparison of those elements.

- **Theorem:** All comparison sorts are $\Omega(n \cdot \log n)$ in the worst case.
- It should also be noted that Heapsort and Merge Sort are the most asymptotically optimal comparison sorts.

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Space Complexity	Stable Sort	Notes
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Simple, but inefficient for large lists
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Simple, but not suitable for large lists
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Efficient for small or nearly sorted lists
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Efficient and stable, good for large lists
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$ to $O(n)$	No	Fast on average, but worst-case can be bad
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	In-place, but not stable

Decision Trees

Decision trees provide a visual representation of the process of comparison sorting. The height of this tree is the worst case running time - $\Omega(n \cdot \log n)$



Consider

These sorting algorithms sort in $\Omega(n \cdot \log n)$ in the worst case. Can we sort faster than this? Yes we can!

- 3.2 Counting Sort
- 3.3 Radix Sort

3.2 Counting Sort

Linear Sorting

Counting sort is a stable sorting algorithm that runs in linear time with respect to two variables, k and n. It **ONLY** works with numbers that span from 1 to k, i.e. positive numbers only.

- k is the largest number in the list
- n is the length of the list

Counting sort runs in $O(n+k)$ in the worst case. It works by:

- Counting the occurrence of each number in the list
- Based on these occurrences, determining the position of each number in the sorted list
- Iterating from end to start when placing elements in the sorted array to keep stability

3.2.1

Exercise 1 - Sort a list whose numbers span from 1 to 9 using counting sort

1. Create an array of size k, and increment each position by 1 for every occurrence of that position in the array.

Unsorted Array:

1	4	1	2	7	5	2

Element Counts:

0	2	2	0	1	1	0	1	0	0

2. Starting from the second position, we add the value of the previous position to the current position.

Element Counts:

0	2	4	4	5	6	6	7	7	7

3. Create a sorted array of length max of previous array (7)

Sorted Array:

4. Iterate through the unsorted array. For each element i in the unsorted array starting from the end of the array, find the value at position i in the element counts array. This value is the position that i will be in the sorted array. Decrement the value in the counts array by 1.

- For example, the first element in the unsorted array is 1. We go to $\text{counts}[1]$ and find that the value is 2. We place 1 at position 2 in the sorted array, and decrement $\text{counts}[1]$ by 1.

Element Counts:

0	1	4	4	5	6	6	7	7	7

Sorted Array:

	1					

5. Repeat this for every element in the unsorted array

Element Counts:

0	0	2	4	4	5	6	6	7	7

Sorted Array:

1	1	2	2	4	5	7

Pseudocode - Input `A[1..n]`, Output `B[1..n]`, Storage `C[1..k]`

```
def CountingSort(A, B, k):
    # Initialize the count array C[1..k]. O(k)
        C = []
        for i in range(k + 1):
           C.append(0)

    # Count the occurrences of each number in array A. O(n)
    for j in range(1, len(A) + 1):
```

```
        C[A[j]] += 1

    # Update the count array to contain the position of the number. O(k)
    for i in range(2, k + 1):
        C[i] = C[i] + C[i - 1]

    # Build the output array B. O(n)
    for j in range(len(A), 0, -1):
        B[C[A[j]]] = A[j]
        C[A[j]] -= 1
```

⚠ **Consider the space complexity of counting sort**

We don't always want to use counting sort because it has quite a large space complexity. We need an array of size K for storage. Consider sorting an array of 32 bit integers.

3.3 Radix Sort

Radix Sort

Radix Sort is an efficient, non-comparative integer sorting algorithm that sorts numbers digit by digit, starting from the least significant digit to the most significant digit. It uses Counting Sort as a stable subroutine to sort the numbers in each pass.

The key advantages of Radix Sort are:

1. It has a linear time complexity of $O(d * (n + k))$, where d is the number of digits, n is the number of elements, and k is the range of input.
2. It is a stable sort, preserving the relative order of elements with equal keys.
3. It can be optimized by using a larger radix, reducing the number of passes required to sort the numbers.

However, Radix Sort has some limitations:

1. It is only suitable for integers or elements that can be represented as integers.
2. The space complexity is $O(n + k)$, which can be significant for large datasets or a large range of input.
3. The performance depends on the number of digits and the radix chosen. Choosing an optimal radix is crucial for efficiency.

⚠ **Consider the use of different radixes**

Given:

- You have an array of $n = 1,000,000$ numbers, each represented using $b = 64$ bits.
- You want to sort these numbers using Radix Sort.

Question:

What will be the time complexity of Radix Sort if you treat the numbers as:

1. Four-digit numbers with a radix of 2^{16}?
2. Sixteen-digit numbers with a radix of 2^4?

Step 1: Determine the number of passes required for each radix.

a) With a radix of 2^{16}, we group 16 bits together. The number of passes required is:

$64 \text{ bits} \div 16 \text{ bits per digit} = 4 \text{ passes}$

b) With a radix of 2^4, we group 4 bits together. The number of passes required is:

$64 \text{ bits} \div 4 \text{ bits per digit} = 16 \text{ passes}$

Step 2: Calculate the time complexity using the formula $\Theta(b/r * (n + 2^r))$.

a) For a radix of 2^{16}:

$\Theta(64/16 * (1,000,000 + 2^{16}))$

$= \Theta(4 * 1,065,536)$

$= \Theta(4,262,144)$

b) For a radix of 2^4:
$\Theta(64/4 * (1,000,000 + 2^4))$
$= \Theta(16 * 1,000,016)$
$= \Theta(16,000,256)$

Therefore, the Radix Sort time complexity will be:
a) $\Theta(4,262,144)$ with a radix of 2^{16}
b) $\Theta(16,000,256)$ with a radix of 2^4

As you can see, using a larger radix (2^{16}) results in fewer passes and a lower time complexity compared to using a smaller radix (2^4). In this example, treating the 64-bit numbers as four-digit numbers with a radix of 2^{16} is more efficient than treating them as sixteen-digit numbers with a radix of 2^4.

The choice of radix is an important factor in optimizing the performance of Radix Sort. A larger radix leads to fewer passes, but it also increases the space complexity and the time required for each pass. The optimal radix depends on the specific problem, the hardware, and the implementation details.

3.2.2

Exercise - Sort a list using Radix Sort

1. **Prepare the array**:

Unsorted Array:

170	45	75	90	802	24	2	66

2. **Sort by the least significant digit (1s place)**: Use counting sort to sort the array based on the 1s digit.

1s Sorted Array:

170	90	802	2	24	45	75	66

3. **Sort by the next significant digit (10s place)**: Again, use counting sort to sort the array, this time based on the 10s digit.

10s Sorted Array:

802	2	24	45	66	170	75	90

4. **Sort by the next significant digit (100s place)**: Again, use counting sort to sort the array, this time based on the 100s digit.

100s Sorted Array:

2	24	45	66	75	90	179	802

5. **Array is now sorted!**

Psuedocode

```
def RadixSort(A, d)
        for i=1 to d
                CountingSort(A) on digit i
```

4.1 Randomised Selection Algorithm

The Idea

Consider a scenario of finding the $i'th$ order statstic in a set. We know we can find this element in $O(n \cdot \log(n))$ time using a divide and conquer comparison sort. Can we do this faster?

Using a selection algorithm rather than a sorting algorithm, we can do this in linear time - $O(n)$. We do this by modifying the Quick Sort algorithm, and worrying only about one of partitions on either side of the pivot. We use a randomised method to choose the pivot.

RSelect

```
RSelect(array A, length n, order_statistic i) {
        if n == 1
                return A[0]

        pivot = A[random]
        partition(A, pivot)           //  A = [   < pivot   | pivot |      > pivot      ]
        j = index of pivot

        if i == j
                return pivot        // since the pivot is in it's final position, it is the j'th smallest
element
        else if j > i
                return RSelect(A[partition 1], j-1, i)
        else if j > i
                return RSelect(A[partition 2], n-j, i-j)
}
```

⚠ **Consider the Running Time of `RSelect`**

Depends on how 'well' we pick the pivot. A poorly chosen pivot can result in an at worst $O(n^2)$ running time, but this incredibly unlikely. Your computer is more likely to be struck by a meteor than for this to happen.

The best pivot is the median element. But this is circular. We prove that the recurrence $T(n) \leq T(\frac{n}{2} + O(n)) = O(n)$ using the first case of the Master Theorem.

A random pivot is *pretty good, often enough*

Scenario	Running Time	Note
Best Case	$O(n)$	The pivot always splits the array in half
Average Case	$O(n)$	On average, splits are reasonably good
Worst Case	$O(n^2)$	The pivot is the smallest or largest element
Expected Time	$O(n)$	Average case, assuming random pivot selection

4.1.1

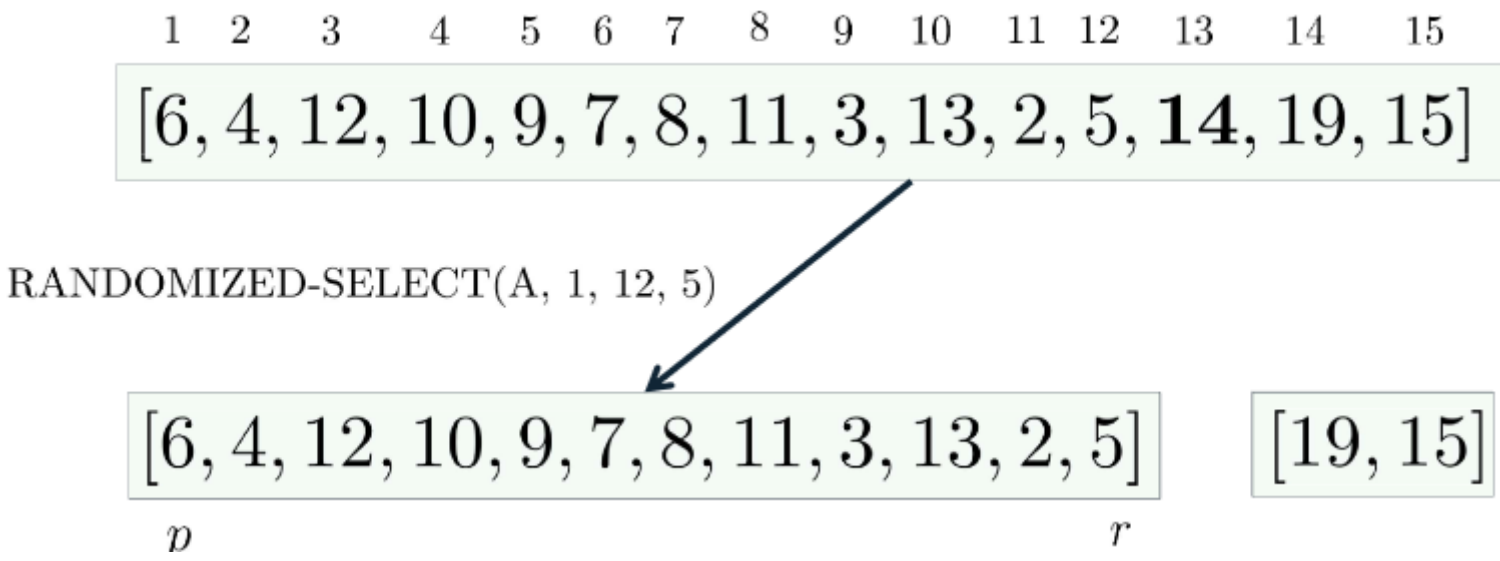
✎ **Exercise - Find the fifth order statistic of the following array:**

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	19	4	12	**14**	9	15	7	8	11	3	13	2	5	10
p														r

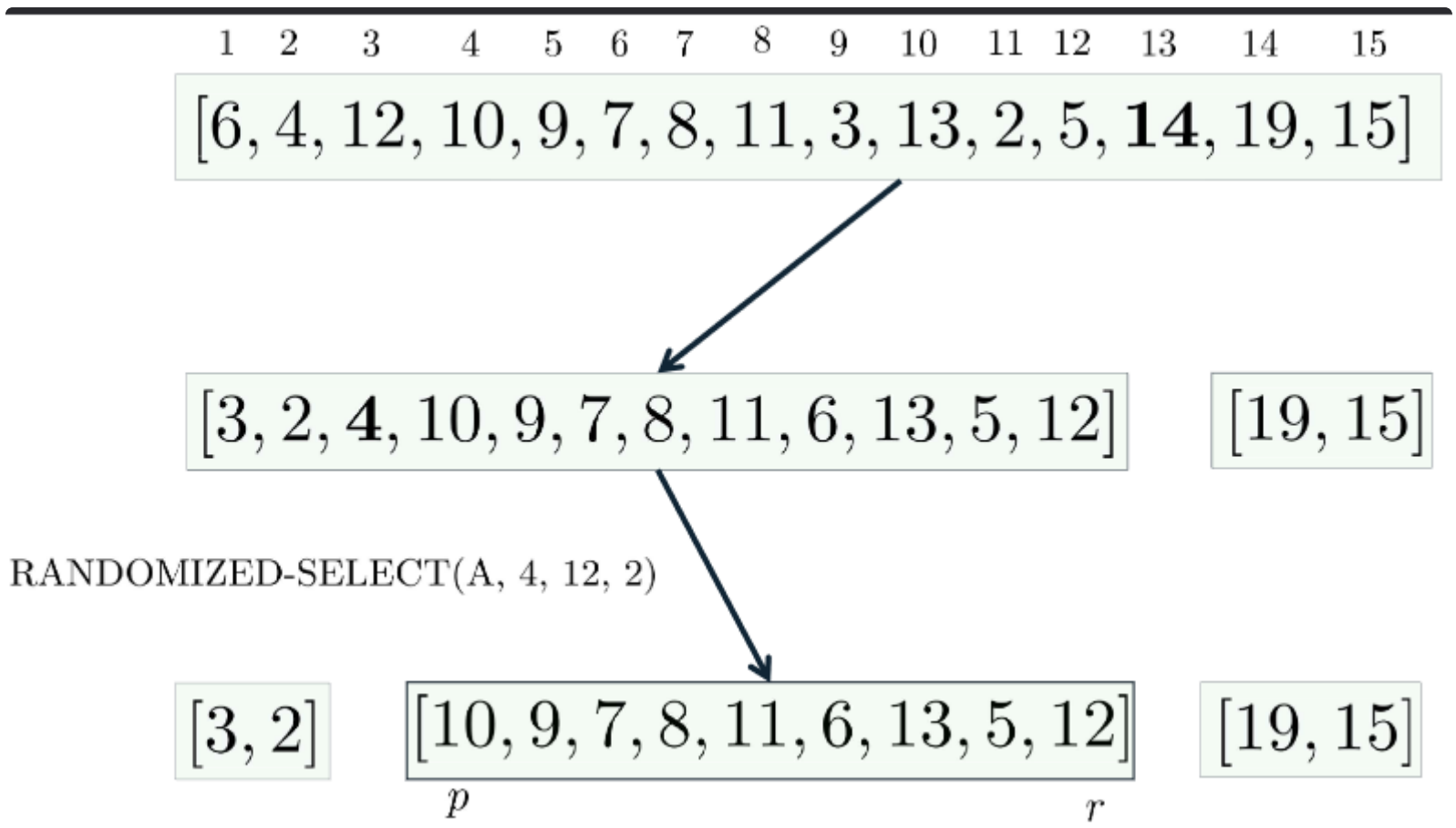
1. Choose a random pivot - 14 is chosen.
2. Partition the array around 14, smallest elements before and larger elements after.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
6	4	12	10	9	7	8	11	3	13	2	5	**14**	19	15
p														r

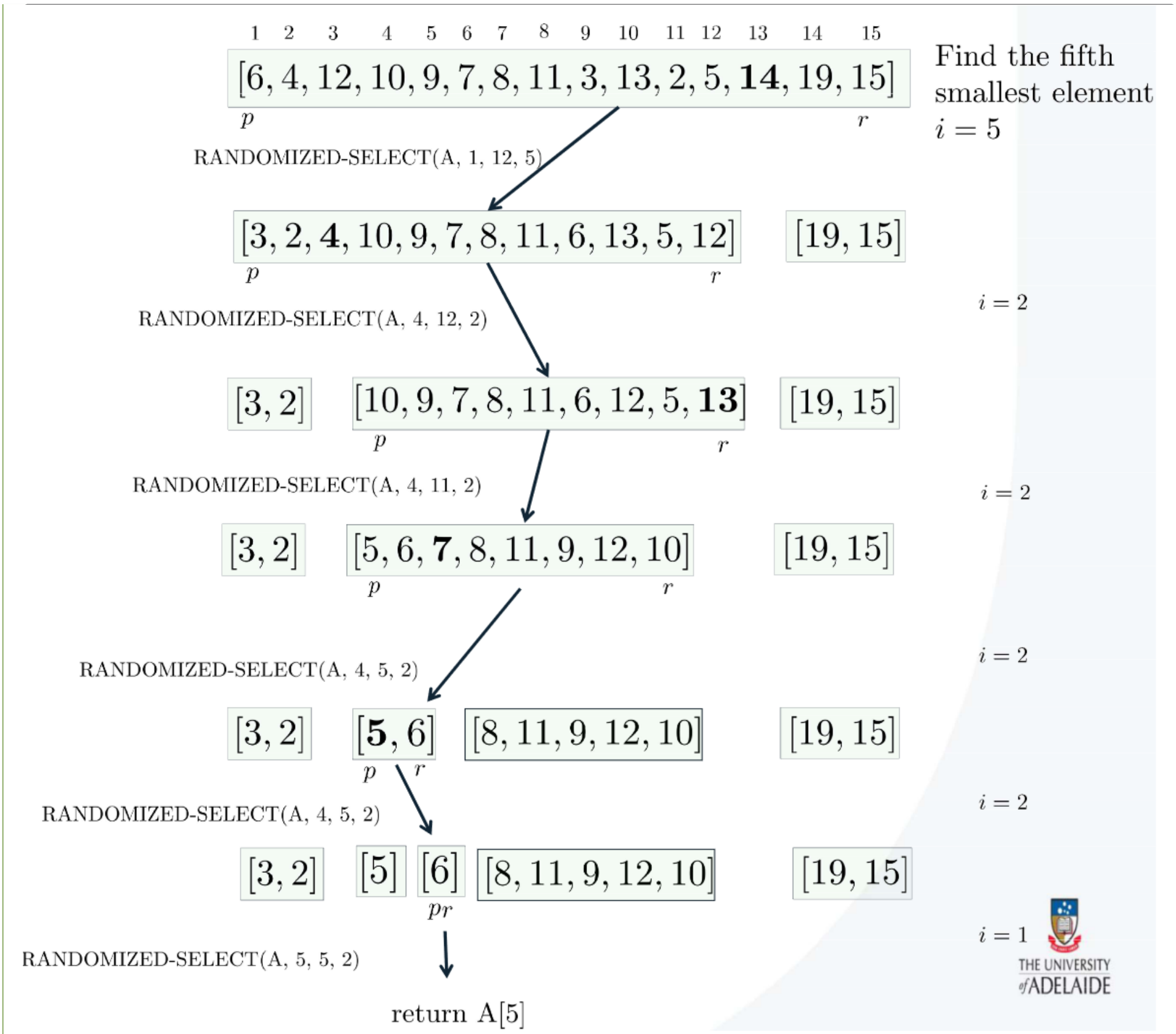
3. Determine if the chosen element is smaller or bigger than the pivot. Because its smaller, we call the function again on the left partition



4. Repeat the process. Pick a random pivot - 4 is chosen.



5. Repeat until i is 1.



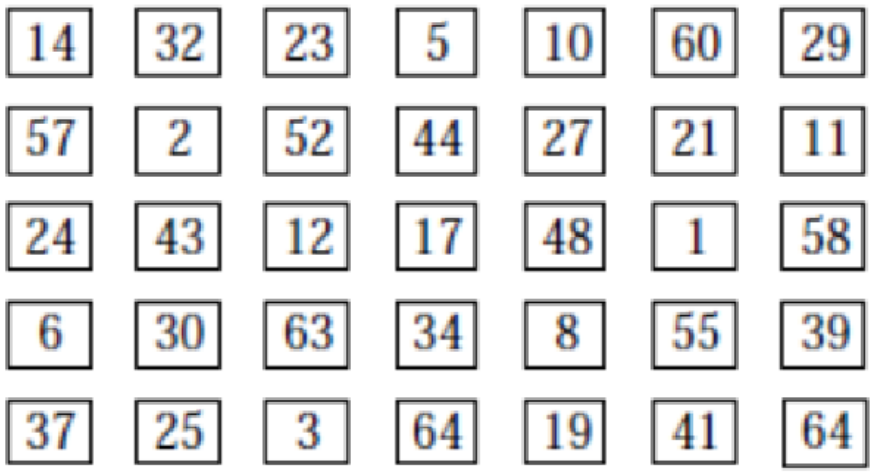
4.2 Deterministic Selection Algorithm

Idea

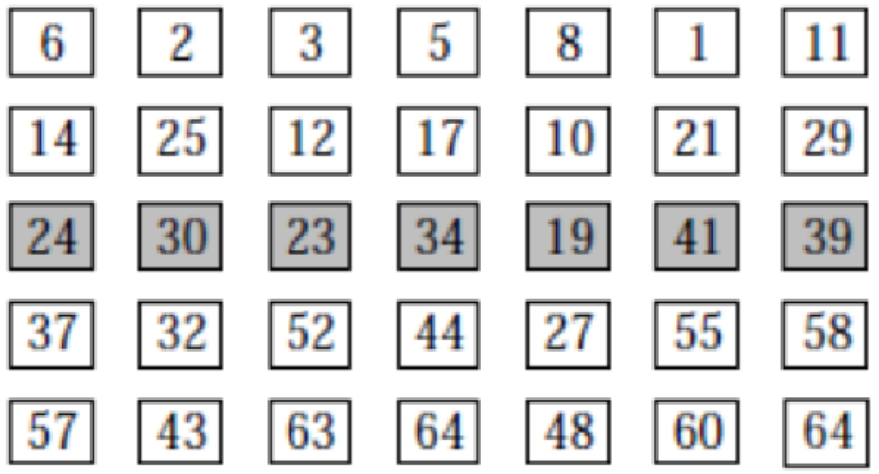
We already found a selection algorithm in RSelect that can sort on average in $O(n)$ time, why bother with anything else? Because DSelect provides an at worst $O(n)$ running time by carefully selecting the pivot, rather than at random (and it's also really cool!)

⚠ Guaranteeing a Good Pivot

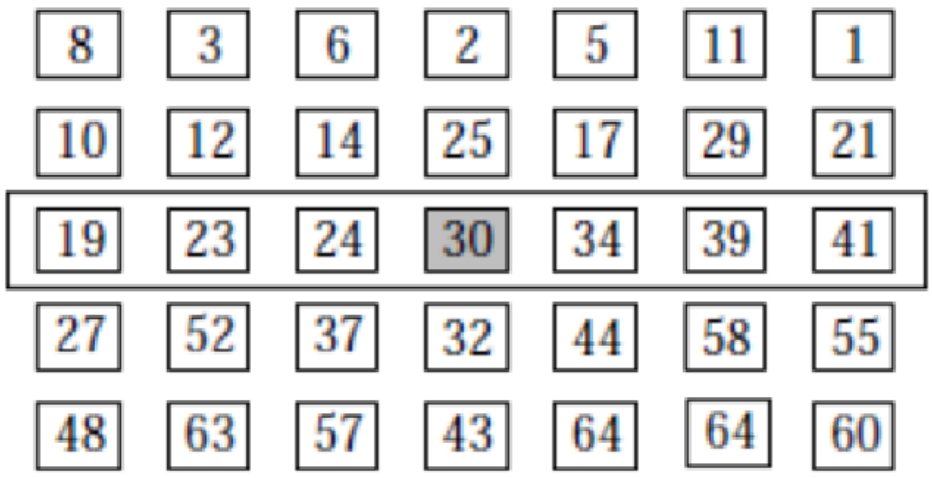
Use the **Median of Medians** strategy as it ensures a good pivot by selecting a pivot that is not too extreme. This strategy divides the array into smaller groups, sorts these groups, and takes the median of each.



Group



Get group medians



By recursively finding the median of these medians, we guarantee a pivot that is reasonably central. This method significantly improves the efficiency of quicksort and selection algorithms by ensuring that the partitioning is relatively balanced, avoiding the worst-case performance associated with choosing poor pivots.

DSelect

```
ChoosePivot(array A, length n) {
    array C[n / 5]; // Create an array to store medians
    for (int i = 0; i < n / 5; i++) {
        sort(A + i * 5, 5); // Sort each group of 5 elements
        C[i] = A[i * 5 + 2]; // Pick the median and store it in C
    }

    // Recursively find the median of the medians
    return ChoosePivot(C, n / 5);
}
```

```
RSelect(array A, length n, order_statistic i) {
    if n == 1
        return A[0]

    pivot = ChoosePivot(A, n)
    partition(A, pivot)           // A = [   < pivot   | pivot |      > pivot      ]
    j = index of pivot

    if i == j
        return pivot        // since the pivot is in its final position, it is the j'th smallest element
    else if j > i
        return RSelect(A[partition 1], j-1, i)
    else if j < i
        return RSelect(A[partition 2], n-j, i-j)
}
```

⚠ The Practicality of DSelect Compared to RSelect

While **DSelect** (Deterministic Select) offers theoretical guarantees on performance, it's worth considering its practicality, especially in comparison to **RSelect** (Randomized Select).

- **Complex Implementation**: DSelect's process, which involves dividing the array into smaller groups, sorting these groups, finding medians, and recursively selecting the median of medians, introduces complexity that can be cumbersome in real-world applications.
- **Overhead Costs**: The additional steps in DSelect, although they ensure a good pivot, add significant overhead. This can make DSelect less efficient for smaller datasets where simpler algorithms would suffice.
- **Theoretical vs. Practical Benefits**: The O(n) time guarantee of DSelect is a theoretical advantage that might not always translate into practical benefits, especially given that the worst-case scenarios it guards against are rare in practice.

Given these considerations, while DSelect is an important algorithm to study for its theoretical insights, RSelect might often be the preferable choice for practical applications due to its simplicity and effectiveness in typical use cases.

5.1 Binary Search Trees

Binary Search Trees

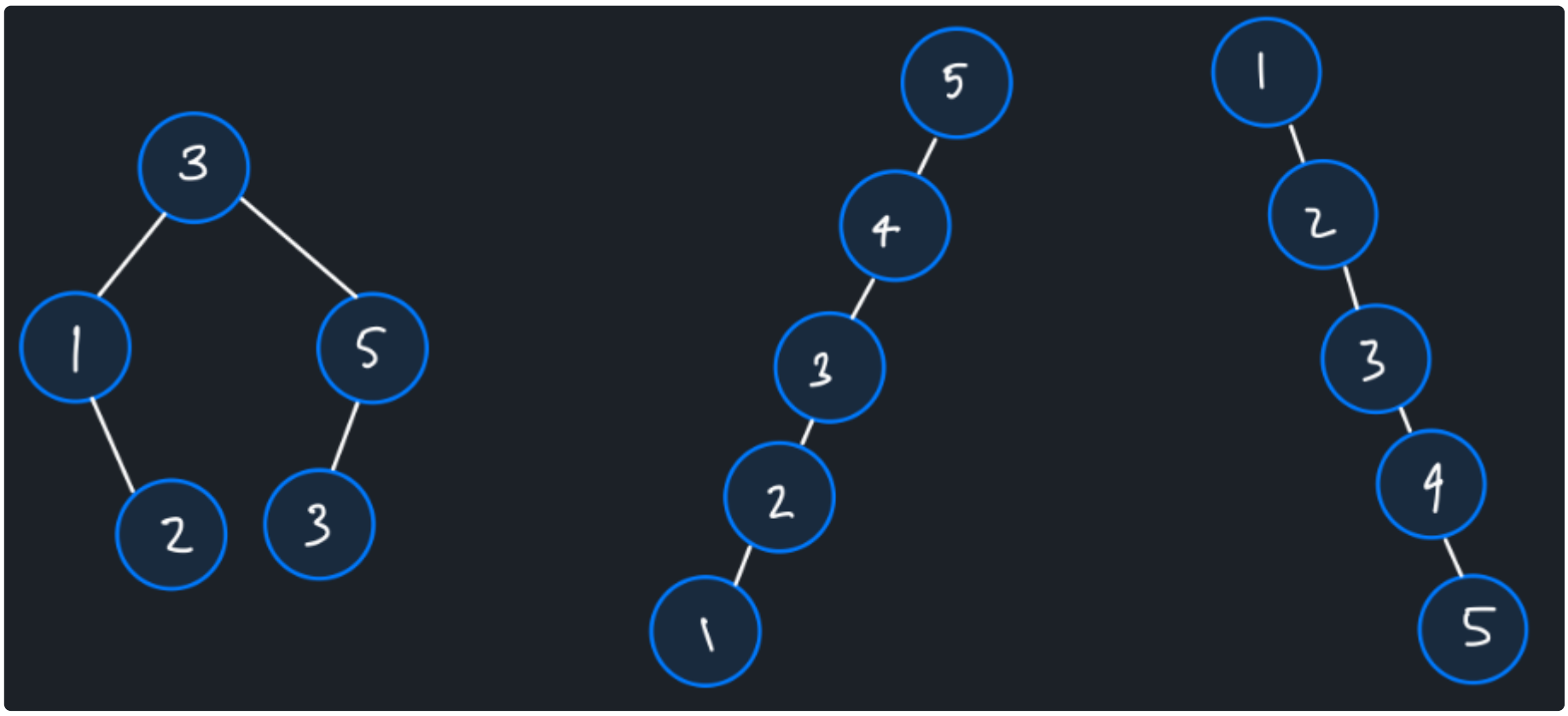
Balanced Binary Search Trees (BSTs) offer a dynamic, highly efficient way to store data that is always sorted, allowing for fast retrieval similar to that of a sorted array. However, unlike arrays, balanced BSTs are designed to maintain optimal balance through various algorithms, such as AVL trees or Red-Black trees, ensuring that operations like **search, insertion and deletion can be performed in logarithmic time complexity**.

A BST has exactly one node per key, which forms the most basic version of the tree.

```
class Node:
  def constuctor(self, key):
      self.key = key
      self.left = None
      self.right = None
      self.height = 1


class BST:
  def __init__(self):
      self.root = None
```

The **Search Tree Property** is a fundamental characteristic of a Binary Search Tree. It states that for any node x in the BST, **all keys in the left subtree of x must be less than x** , and **all keys in the right subtree of x must be greater than x**. This property should hold at every node of the tree, ensuring that the keys are stored in a sorted manner. By maintaining the Search Tree Property, BSTs enable efficient searching, insertion, and deletion operations.

In-Order Traversal is a way to **visit all the nodes of a BST in ascending order of their keys**. It follows a specific sequence: first, it recursively traverses the left subtree, then visits the current node, and finally, it recursively traverses the right subtree. The in-order traversal of a BST produces a sorted sequence of keys, which is particularly useful when you need to process the nodes in sorted order. Some common applications of in-order traversal include printing the keys in ascending order, performing range queries efficiently by leveraging the sorted order of keys, and validating the correctness of a BST by checking if the in-order traversal produces a sorted sequence. The time complexity of in-order traversal is O(n), where n is the number of nodes in the tree, as it visits each node exactly once.



⚠ **Consider the height of a BST**

It's important to note that there are multiple possible binary search trees for the same set of keys. The structure of a BST can vary widely, depending on the order of insertion and deletion of keys. The image above shows valid BSTs for a set of elements.

The height of a Binary Search Tree (BST) is a crucial factor in determining the running time of search, insertion, and deletion operations. In a **perfectly balanced BST** of height $\log n$, the height is logarithmic in relation to the number of nodes, resulting in efficient $O(\log n)$ time complexity for these operations.

However, in the worst-case scenario, when the **BST is skewed or unbalanced,** the height can become linear, leading to a time complexity of $O(n)$ for search, insertion, and deletion. This occurs when the BST resembles a linked list, with each node having only one child.

Operation	Best Case	Average Case	Worst Case
Search	O(1)	O(log n)	O(n)
Insertion	O(1)	O(log n)	O(n)
Deletion	O(1)	O(log n)	O(n)

BST Search

Search for a key k in a tree t.

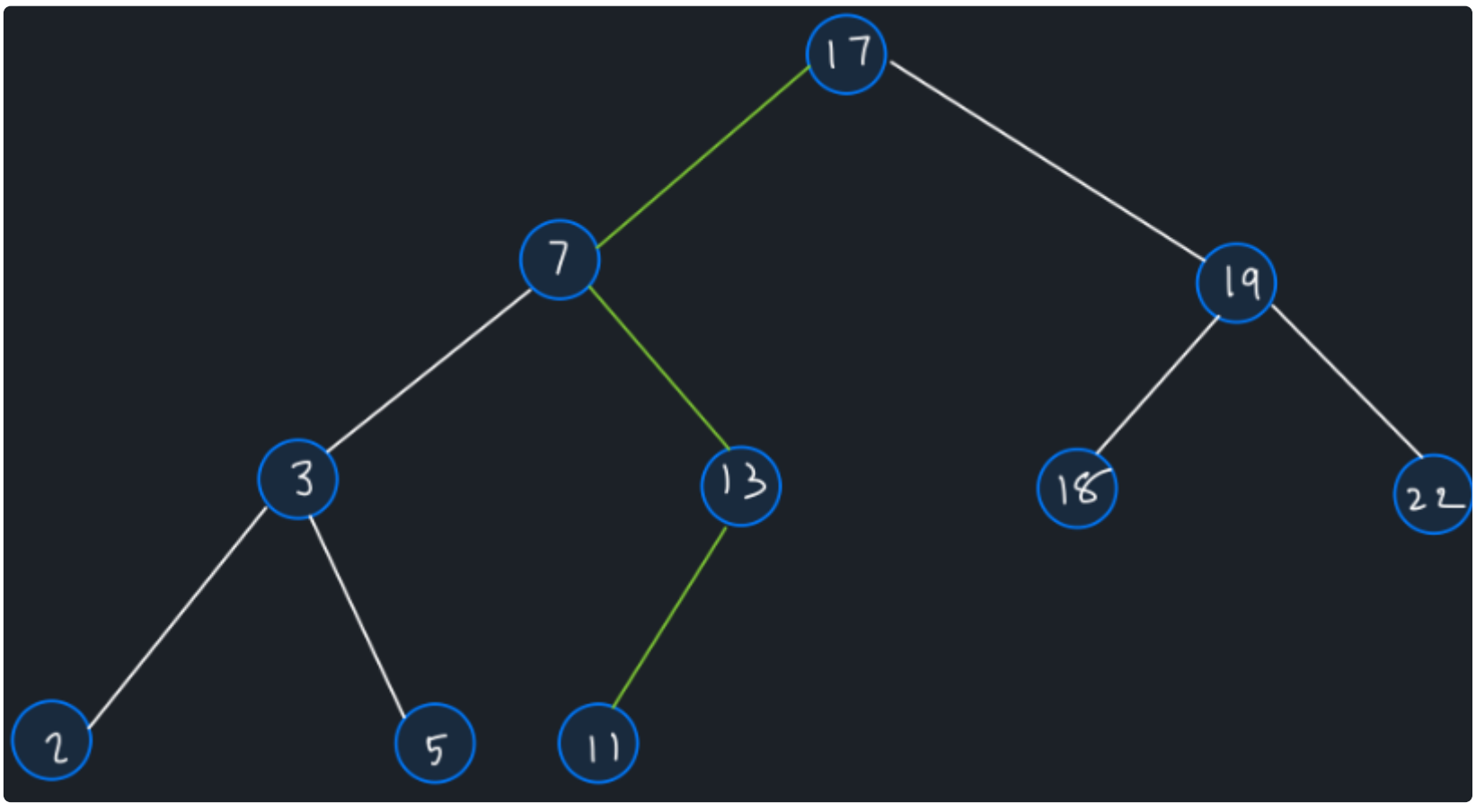
- Start at the root
- At node x, compare x and k

1. If `k = x`, then found
2. If `k < x`, search in the left subtree of x. If subtree does not exist, return `not found`
3. If `k > x`, search in the right subtree of x. If subtree does not exist, return `not found`

```
def search(self, key):
   x = self.root
   while x is not None and key != x.key:
           if key < x.key:
                   x = x.left
           else:
                   x = x.right
   return x
```

5.1.1

Exercise 1 - Traverse the following BST to find the value of 11



BST Insertion

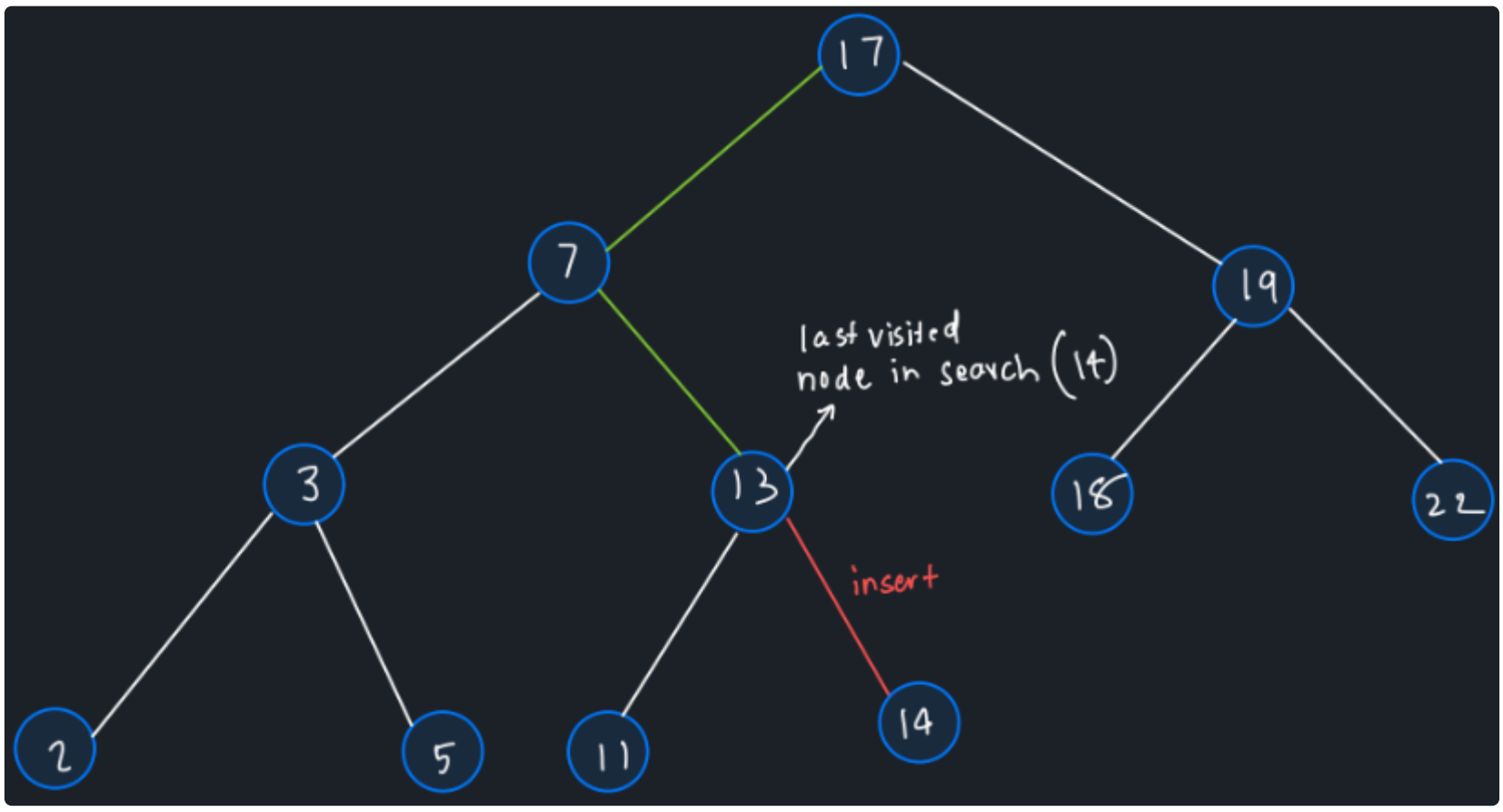
The `insert` function inserts a new node with the given `key` into the AVL tree. It follows these steps:

1. If the `root` is `None`, create a new node with the given `key` and return it as the new root.
2. If the `key` is less than the `root`'s key, recursively insert the key into the left subtree.
3. If the `key` is greater than or equal to the `root`'s key, recursively insert the key into the right subtree.
4. Return the updated `root` of the subtree.

```
def insert(self, root, key):
    if not root:
        return Node(key)
    elif key < root.key:
        root.left = self.insert(root.left, key)
    else:
        root.right = self.insert(root.right, key)

    return root
```

5.1.2

Exercise 1 - Insert a node of 14 into the following BST



BST Deletion

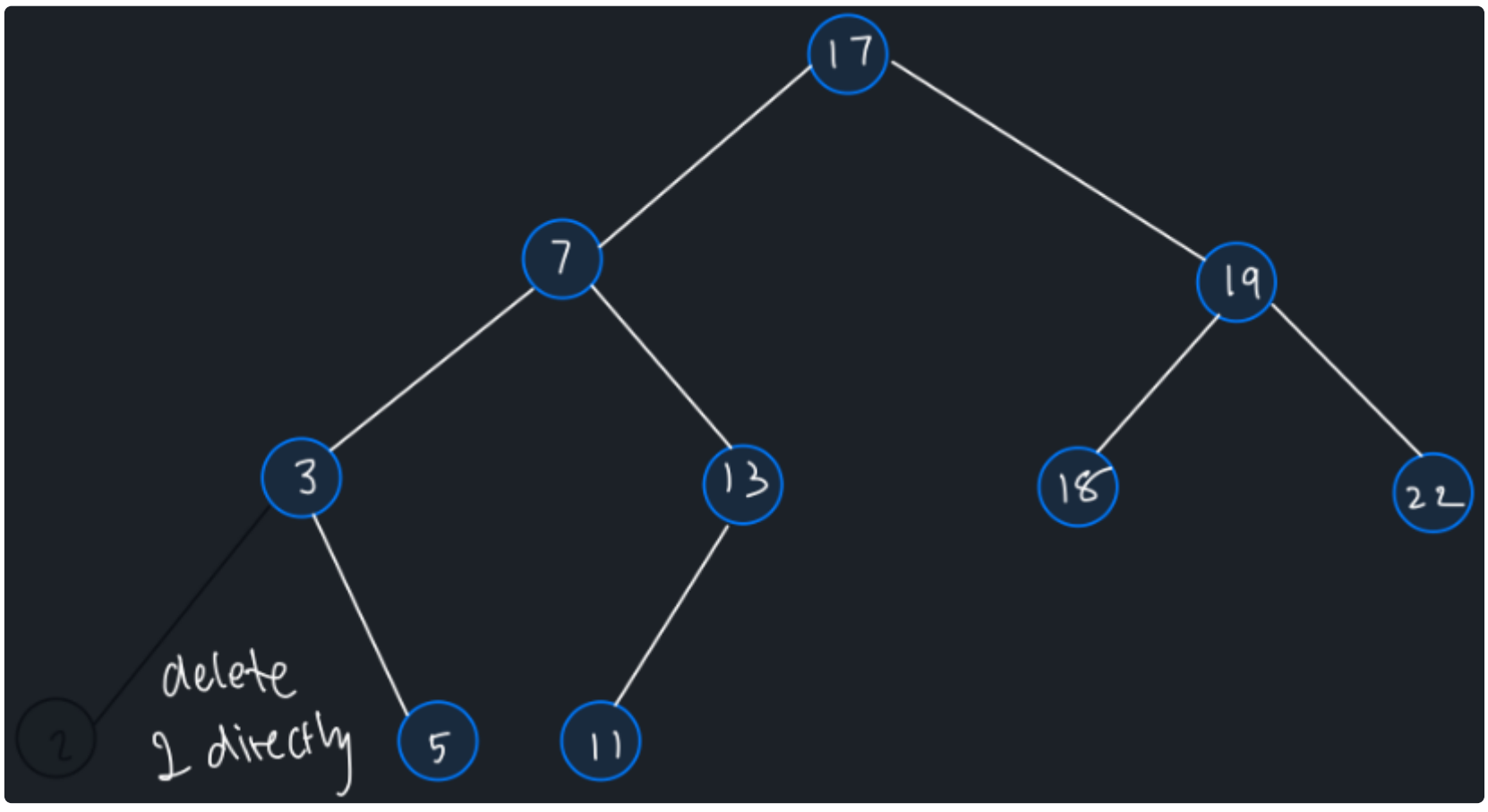
The `delete` function removes a node with the given `key` from the AVL tree. It follows these steps:

1. If the `root` is `None`, return `root` as there is nothing to delete.
2. If the `key` is less than the `root`'s key, recursively delete the key from the left subtree.
3. If the `key` is greater than the `root`'s key, recursively delete the key from the right subtree.
4. If the `key` is equal to the `root`'s key, we have found the node to delete:
    - If `key` is stored at a leaf, delete this leaf and the incoming edge.
    - If `key` has 1 child, redirect the pointer pointing to `k` to `k`'s child and delete `k`.
    - If `key` has 2 children:
        - Search in the tree for the largest element `x` smaller than `k`.
        - In the left subtree of `k`, follow the right path as long as possible to find `x`.
        - Swap `x` and `k` and recursively delete `k`.
5. Return the updated `root` of the subtree.

```
def delete(self, root, key):
    if not root:
        return root
    elif key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        if not root.left:
            temp = root.right
            root = None
            return temp
        elif not root.right:
            temp = root.left
            root = None
            return temp
        # find inorder predecessor (biggest element smaller than key)
        temp = self.get_max_node(root.left)
        root.key = temp.key
        root.left = self.delete(root.left, temp.key)

    return root
```

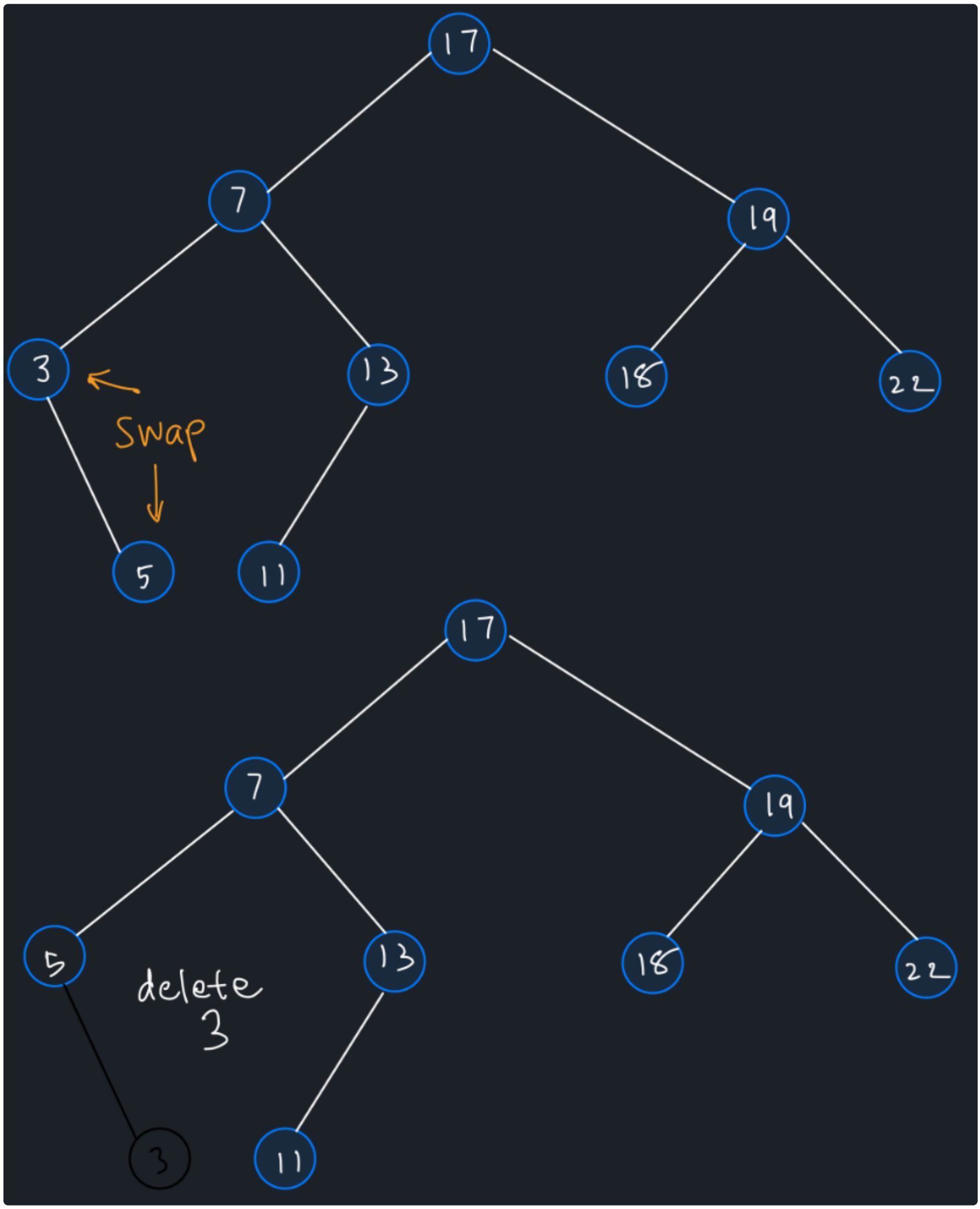
5.1.3

Delete the node with value 2 with 0 children



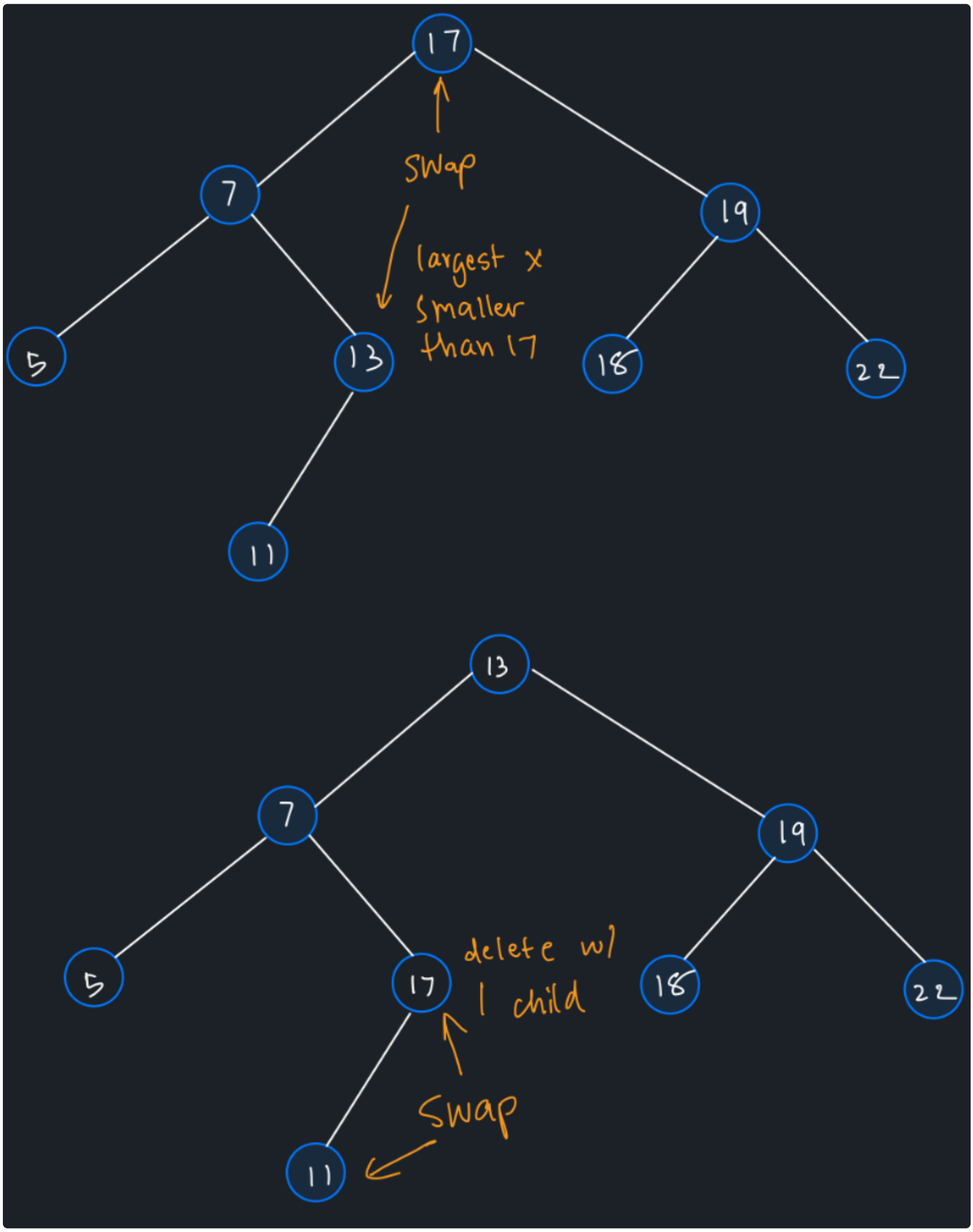
5.1.4

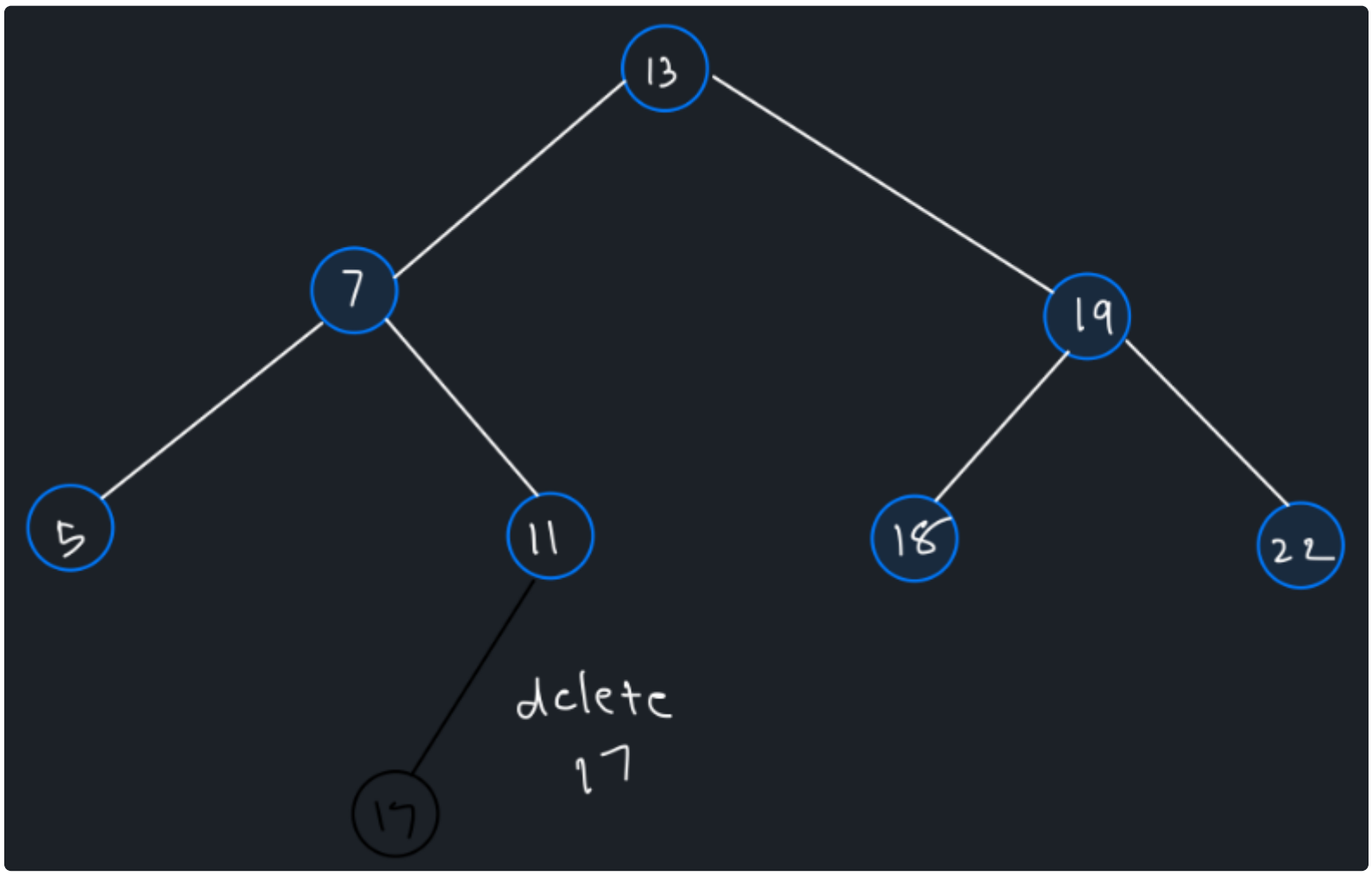
Delete the node with value 3 with 1 child



5.1.5

Delete the node with value 17 with 2 children


17
7
19
5
13
18
22
11
swap
largest x
smaller
than 17
13
7
19
5
17
18
22
11
delete w/
1 child
swap

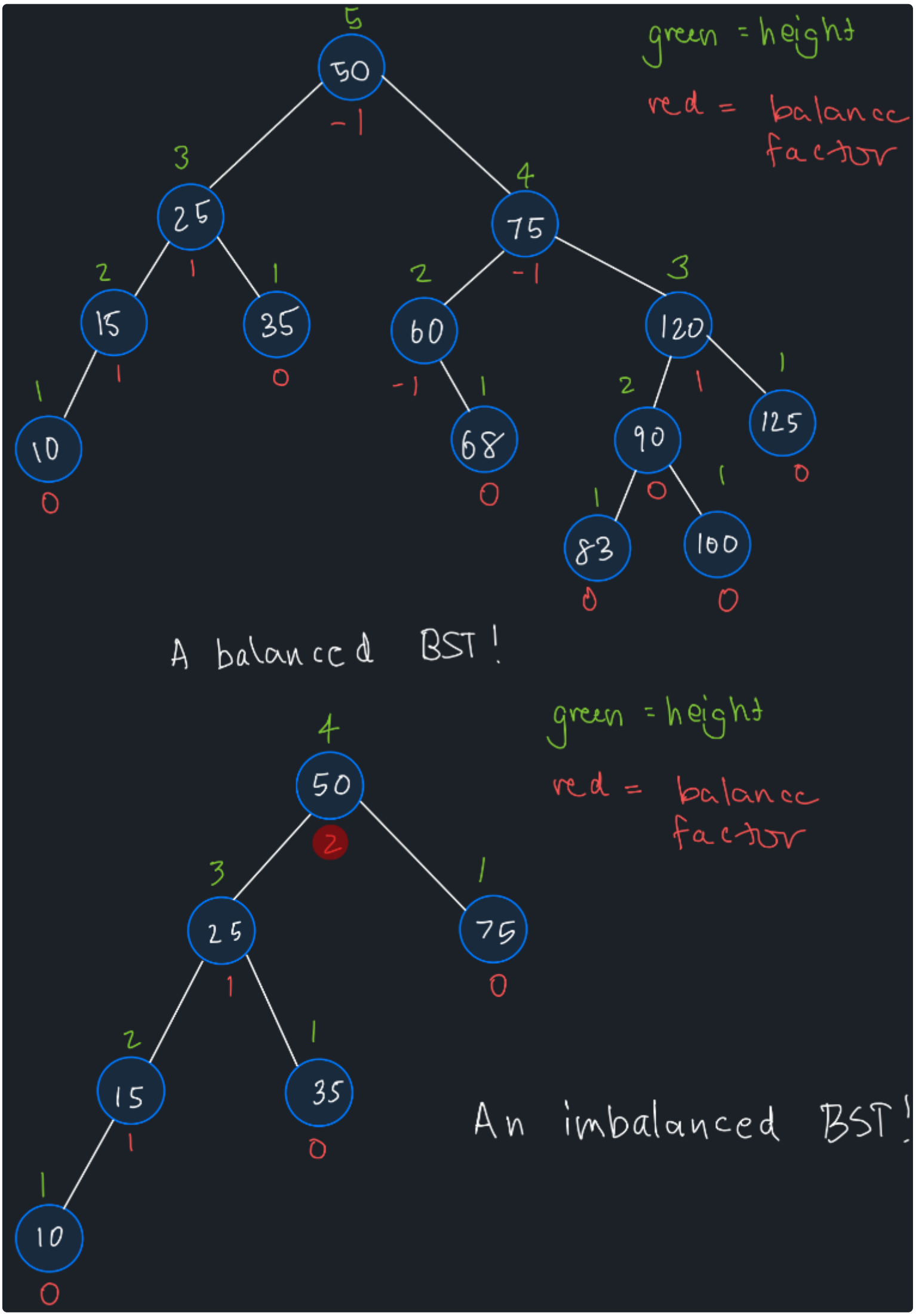


5.2 AVL Trees

Imbalanced BSTs

Recall than an imbalanced Binary Search Tree can degrade into a structure similar to a linked list, resulting in a worst-case time complexity of $O(n)$ for common operations. To address this issue and ensure optimal performance, we can proactively balance the BST during its construction and modification. By **maintaining a balanced structure with a height of $\log n$, the time complexity of key operations becomes $O(\log n)$** in the worst case, significantly improving upon the linear time complexity of an imbalanced tree. One type of these self balancing trees is an **AVL Tree.**

An AVL tree says that for any node, the height of it's two subtrees differs by at most, 1 node, i.e. $Balance\ Factor \in -1, 0, 1$



⚠ **How do we balance BSTs?**

In an AVL Tree, rotations are used to fix imbalance.

There are four types of rotations:

1. Left Rotation (LL): Used when the tree is right-heavy and the right subtree is right-heavy.
2. Right Rotation (RR): Used when the tree is left-heavy and the left subtree is left-heavy.
3. Left-Right Rotation (LR): Used when the tree is left-heavy and the left subtree is right-heavy.
4. Right-Left Rotation (RL): Used when the tree is right-heavy and the right subtree is left-heavy.

To determine which rotation to use, we look at the balance factor.

```
def balance(self, root, key, balance_factor):
    if balance_factor > 1 and key < root.left.key:
        return self.right_rotate(root)
```

```
    if balance_factor < -1 and key > root.right.key:
        return self.left_rotate(root)

    if balance_factor > 1 and key > root.left.key:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if balance_factor < -1 and key < root.right.key:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root
```

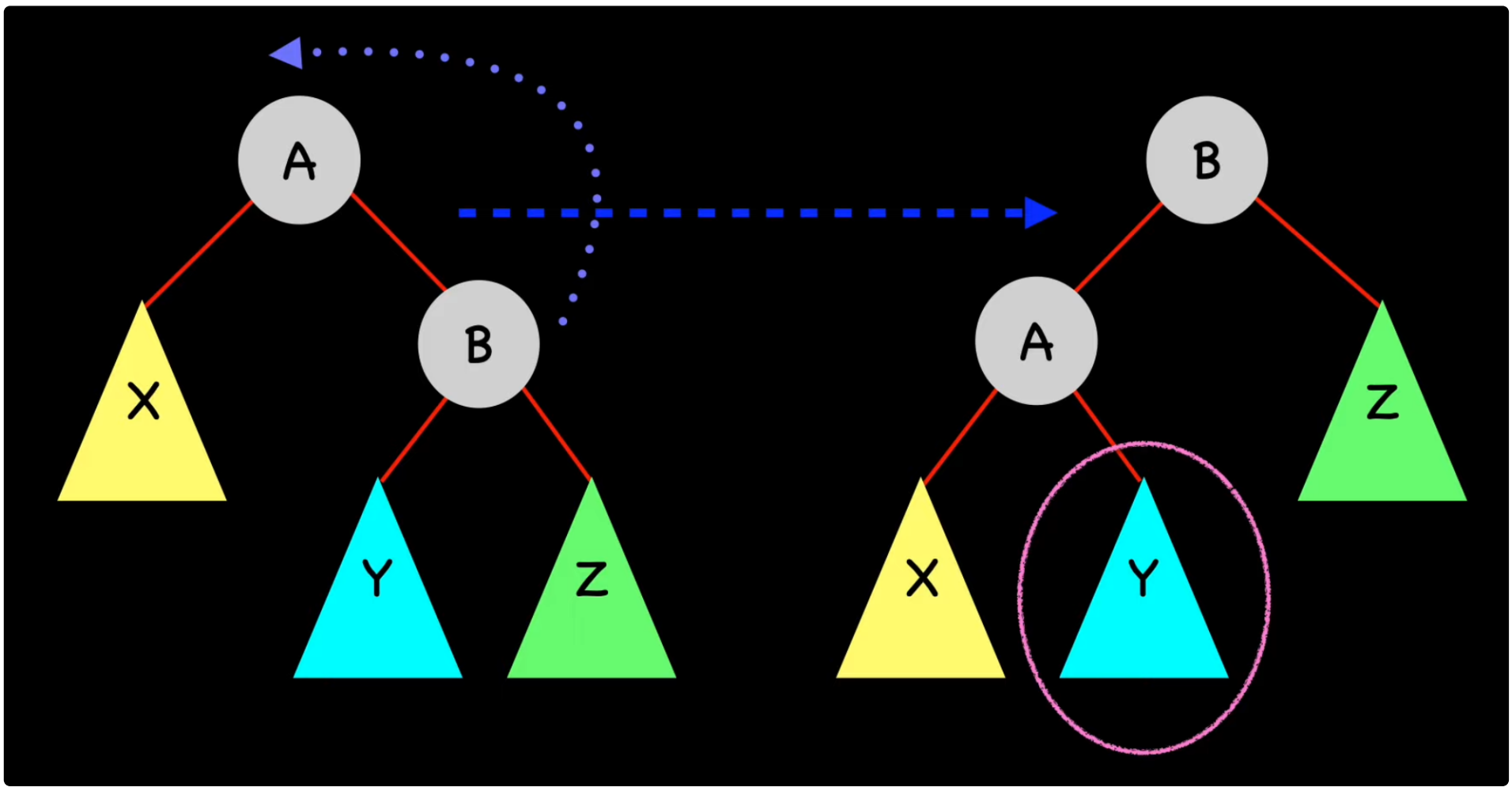
The `balance` function is a method of the AVL tree class and takes four parameters:

- `self`: The instance of the AVL tree class.
- `root`: The root node of the subtree to be balanced.
- `key`: The key of the newly inserted or deleted node.
- `balance_factor`: The balance factor of the root node.

The function returns the new root of the balanced subtree after performing the necessary rotations based on the balance factor and the key.

Left Rotation

A left rotation is performed when a node's right subtree is causing an imbalance. The node is rotated to the left, making its right child the new root of the subtree, and the old node becomes the left child of the new root. The new root's left subtree becomes the right subtree of the old node.

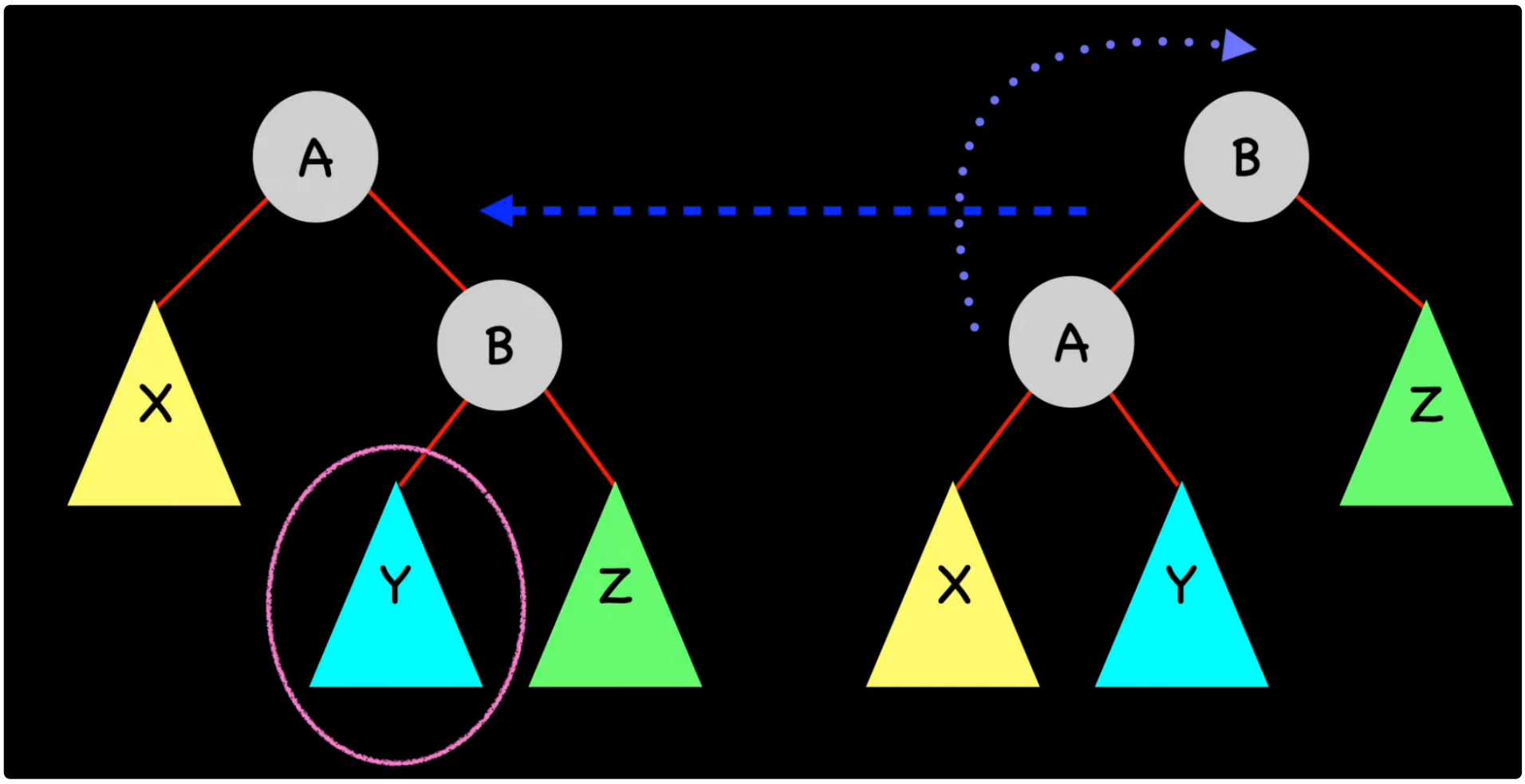



```
def left_rotate(self, node):
    B = node.right
    Y = B.left

    B.left = node
    node.right = Y

    node.height = 1 + max(self.get_height(node.left),
                          self.get_height(node.right))
    B.height = 1 + max(self.get_height(B.left),
                       self.get_height(B.right))

    return B
```

Right Rotation

A right rotation is performed when a node's left subtree is causing an imbalance. The node is rotated to the right, making its left child the new root of the subtree, and the old node becomes the right child of the new root. The new root's right subtree becomes the left subtree of the old node.

```
def right_rotate(self, node):
    A = node.left
    Y = A.right

    A.right = node
    node.left = Y

    node.height = 1 + max(self.get_height(node.left),
                          self.get_height(node.right))
    B.height = 1 + max(self.get_height(B.left),
                       self.get_height(B.right))

    return B
```



AVL Search

Search for a key k in a tree t. The same implementation as a BST Search.

- Start at the root
- At node x, compare x and k

1. If `k = x`, then found
2. If `k < x`, search in the left subtree of x. If subtree does not exist, return `not found`
3. If `k > x`, search in the right subtree of x. If subtree does not exist, return `not found`

```
def search(self, key):
   x = self.root
   while x is not None and key != x.key:
           if key < x.key:
                   x = x.left
           else:
                   x = x.right
   return x
```

AVL Insertion

The `insert` function inserts a new node with the given `key` into the AVL tree. It follows these steps:

1. If the `root` is `None`, create a new node with the given `key` and return it as the new root.
2. If the `key` is less than the `root`'s key, recursively insert the key into the left subtree.
3. If the `key` is greater than or equal to the `root`'s key, recursively insert the key into the right subtree.
4. Update the height of the current `root` node.
5. Calculate the balance factor of the current `root` node.
6. If the balance factor is greater than 1 and the `key` is less than the left child's key, perform a right rotation.
7. If the balance factor is less than -1 and the `key` is greater than the right child's key, perform a left rotation.
8. If the balance factor is greater than 1 and the `key` is greater than the left child's key, perform a left-right rotation.
9. If the balance factor is less than -1 and the `key` is less than the right child's key, perform a right-left rotation.
10. Return the updated `root` of the subtree.

```python
def insert(self, root, key):
    if not root:
        return Node(key)
    elif key < root.key:
        root.left = self.insert(root.left, key)
    else:
        root.right = self.insert(root.right, key)

    root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

    # Update the balance factor and balance the tree
    bf = self.get_balance_factor(root)

    if bf > 1 and key < root.left.key:
        return self.right_rotate(root)

    if bf < -1 and key > root.right.key:
        return self.left_rotate(root)

    if bf > 1 and key > root.left.key:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if bf < -1 and key < root.right.key:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root
```

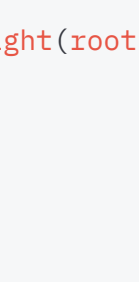
AVL Deletion

The `delete` function removes a node with the given `key` from the AVL tree. It follows these steps:

1. If the `root` is `None`, return `root` as there is nothing to delete.
2. If the `key` is less than the `root`'s key, recursively delete the key from the left subtree.
3. If the `key` is greater than the `root`'s key, recursively delete the key from the right subtree.
4. If the `key` is equal to the `root`'s key, we have found the node to delete:
    - If `key` is stored at a leaf, delete this leaf and the incoming edge.
    - If `key` has 1 child, redirect the pointer pointing to `k` to `k`'s child and delete `k`.
    - If `key` has 2 children:
        - Search in the tree for the largest element `x` smaller than `k`.
        - In the left subtree of `k`, follow the right path as long as possible to find `x`.
        - Swap `x` and `k` and recursively delete `k`.
5. Update the height of the current `root` node.
6. Calculate the balance factor of the current `root` node.
7. If the balance factor is greater than 1 and the left subtree's balance factor is greater than or equal to 0, perform a right rotation.
8. If the balance factor is less than -1 and the right subtree's balance factor is less than or equal to 0, perform a left rotation.
9. If the balance factor is greater than 1 and the left subtree's balance factor is less than 0, perform a left-right rotation.
10. If the balance factor is less than -1 and the right subtree's balance factor is greater than 0, perform a right-left rotation.

11. Return the updated `root` of the subtree.

```
def delete(self, root, key):
    if not root:
        return root
    elif key < root.key:
        root.left = self.delete(root.left, key)
    elif key > root.key:
        root.right = self.delete(root.right, key)
    else:
        if not root.left:
            temp = root.right
            root = None
            return temp
        elif not root.right:
            temp = root.left
            root = None
            return temp
        # find inorder predecessor (biggest element smaller than key)
        temp = self.get_max_node(root.left)
        root.key = temp.key
        root.left = self.delete(root.left, temp.key)

    root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))

    # Update the balance factor and balance the tree
    bf = self.get_balance_factor(root)

    if bf > 1 and self.get_balance_factor(root.left) >= 0:
        return self.right_rotate(root)

    if bf < -1 and self.get_balance_factor(root.right) <= 0:
        return self.left_rotate(root)

    if bf > 1 and self.get_balance_factor(root.left) < 0:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    if bf < -1 and self.get_balance_factor(root.right) > 0:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root
```

Imbalance Type	Condition	Rotations Needed	Example and Intermediate Structures	Intermediate	Final Structure
LL (Left-Left)	Left subtree of left child	Single Right Rotation at Node			
RR (Right-Right)	Right subtree of right child	Single Left Rotation at Node			
LR (Left-Right)	Right subtree of left child	Left Rotation at Left Child, then Right Rotation at Node			

Imbalance Type	Condition	Rotations Needed	Example and Intermediate Structures	Intermediate	Final Structure
RL (Right-Left)	Left subtree of right child	Right Rotation at Right Child, then Left Rotation at Node			

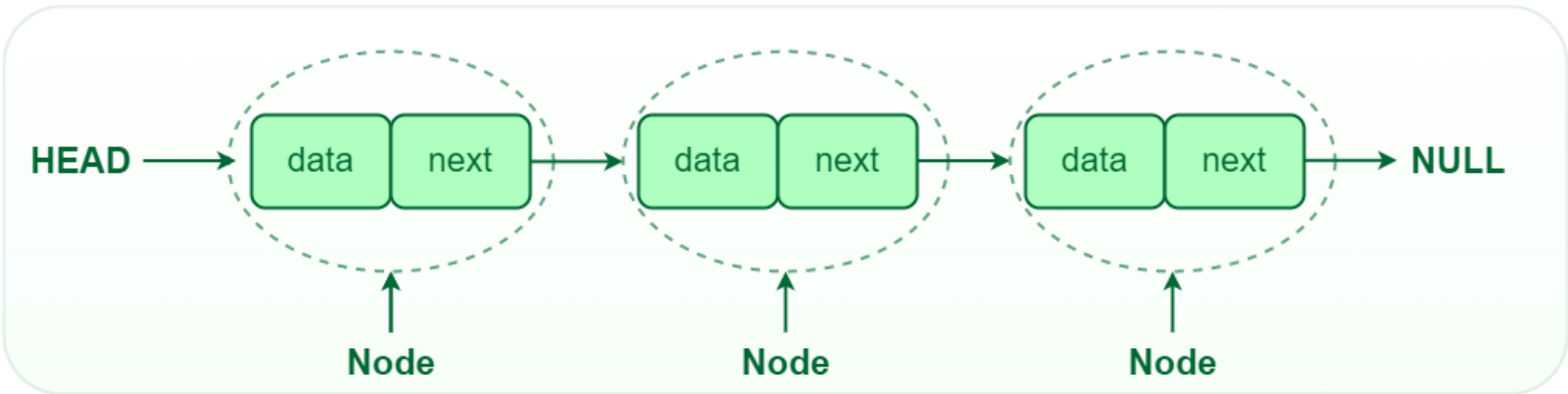
Imbalance Type	Condition	Balance Factor of Node	Balance Factor of Node's Child	Rotations Needed
LL (Left-Left)	Left subtree of left child is deeper	Balance Factor: +2	Balance Factor of Left Child: +1	Single Right Rotation at Node
RR (Right-Right)	Right subtree of right child is deeper	Balance Factor: -2	Balance Factor of Right Child: -1	Single Left Rotation at Node
LR (Left-Right)	Right subtree of left child is deeper	Balance Factor: +2	Balance Factor of Left Child: -1	Left Rotation at Left Child, then Right Rotation at Node
RL (Right-Left)	Left subtree of right child is deeper	Balance Factor: -2	Balance Factor of Right Child: +1	Right Rotation at Right Child, then Left Rotation at Node

NOTE: When performing a x rotation, if a child node has two children, then the x child becomes the child of the root, where x can be left or right.

6.1 Linked Lists

Linked Lists

Linked lists are a fundamental data structure that consists of a sequence of nodes, where each node contains a value and a reference (or link) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, allowing for dynamic memory allocation and efficient insertion and deletion operations.




```python
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedList:
    def __init__(self):
        self.head = None
```

Linked lists can be classified into three main types:

1. Singly Linked List: Each node contains a single reference to the next node in the sequence.
2. Doubly Linked List: Each node contains references to both the next and previous nodes, allowing for traversal in both directions.
3. Circular Linked List: The last node in the sequence points back to the first node, forming a circular structure.

Pros of Linked Lists:

- Dynamic Size: Unlike arrays, linked lists can grow or shrink in size dynamically, which makes them more flexible when dealing with data whose size changes over time.
- Efficient Insertions/Deletions: Adding or removing elements from a linked list is generally faster than an array, especially for operations at the beginning of the list or in the middle, assuming you have direct access to the point of insertion/deletion.

- **No Memory Wastage**: Linked lists allocate memory as needed, so there's no need to allocate excess memory upfront, as is often the case with arrays.

Cons of Linked Lists:

- **Sequential Access**: Accessing an element in a linked list requires traversing from the beginning of the list, which can be slow compared to the direct access provided by arrays.
- **Memory Overhead**: Each element in a linked list requires extra memory for storing the reference (pointer) to the next (and possibly previous) node, which can be significant, especially with a large number of elements.
- **Complexity**: Implementing and managing linked lists can be more complex than arrays, especially when it comes to more sophisticated operations or types of linked lists (e.g., doubly linked or circular linked lists).
- **Poor Cache Performance**: The non-contiguous storage of linked list elements can lead to poor cache performance, making linked list operations slower than those on arrays for certain tasks.

⚠ **Consider the time complexity of linked list operations**

Operation	Time Complexity
Access	O(n)
Search	O(n)
Insertion	O(1) at Start or End (with tail pointer in singly-linked list or in doubly-linked lists) O(n) between two nodes or at End without tail pointer in singly-linked list
Deletion	O(1) at Start O(n) at End without tail pointer in singly-linked list or between two nodes O(1) at End in doubly-linked lists or singly-linked lists with tail pointer

Accessing or searching for an element in a linked list requires traversing the nodes sequentially, resulting in a linear time complexity of O(n). However, insertion and deletion operations can be performed efficiently in constant time O(1) by adjusting the references between nodes.

6.2 Skip Lists

Skip Lists

Skip lists are a probabilistic data structure that provides efficient **search, insertion, and deletion operations with complexity of** $O(\log n)$. They are a variation of linked lists that offer improved performance by maintaining multiple levels of linked lists, allowing for faster traversal. They have $O(n)$ **space complexity** and $O(\log n)$ **height.**

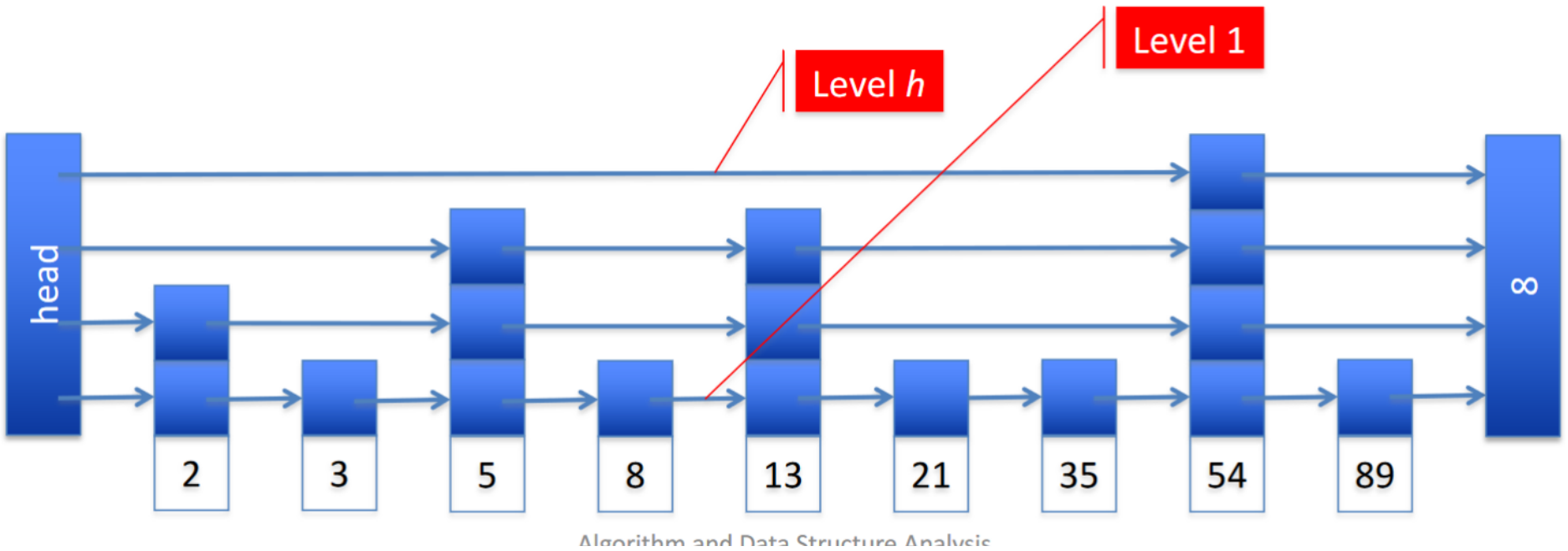
Key characteristics of skip lists:

1. **Balanced**: Skip lists are designed to maintain a balanced structure, ensuring efficient search and insertion operations.
2. **Ordered**: Elements in a skip list are stored in sorted order, enabling efficient search and retrieval.
3. **Dynamic**: Skip lists support dynamic insertion and deletion of elements, allowing the data structure to adapt to changes in the dataset.
4. **Randomized**: The levels of each node in a skip list are determined randomly, providing a probabilistic guarantee of efficiency.
5. **Probabilistic Structure**: Skip lists are built based on probabilistic principles. Each element in the skip list has a probability p (usually 1/2) of being promoted to the next higher level. This probabilistic promotion of elements creates a hierarchical structure with multiple levels, where the expected number of nodes at each level decreases geometrically.

Skip lists consist of **multiple levels of linked lists**, with **each level being a subset of the level below it**. The bottom level (level 0) contains all the elements, while higher levels contain fewer elements, acting as express lanes for faster traversal. The topmost level typically consists of a single node. Each node contains pointers to every node on it's side (left, right, above and below)

⚠ **Subway Express Lines**

Skip Lists are like express lines in a subway. Some trains are able to bypass specific stops and therefore find stations faster!

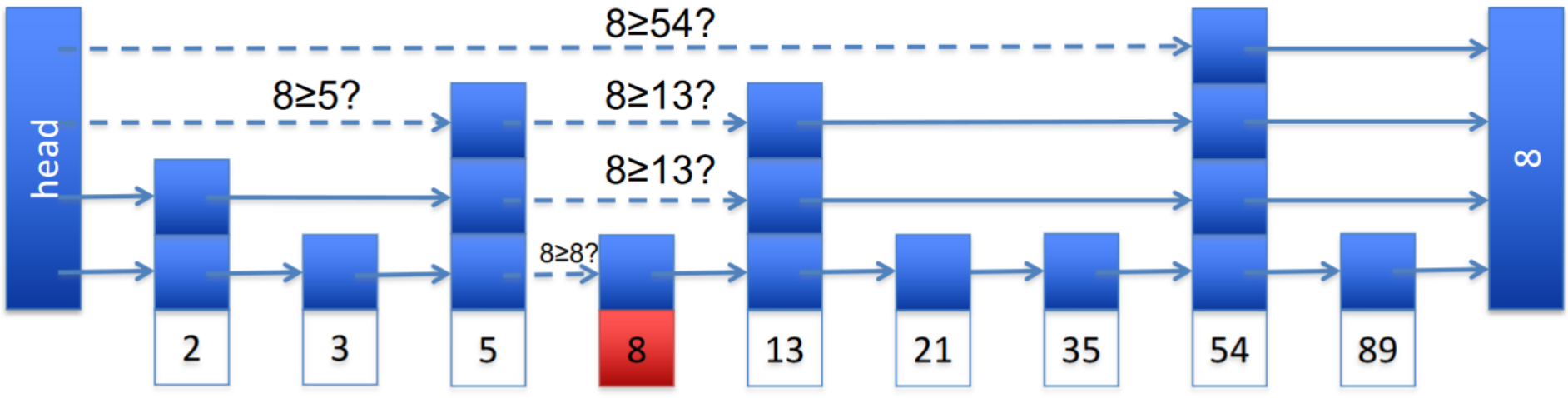


Pseudocode implementation of Node and SkipList classes:

```
class Node:
    value
    left
    right
    above
    below

class SkipList:
    head
    tail
    maxLevel

    constructor():
        this.head = new Node(null)
        this.tail = new Node(null)
        this.maxLevel = 0
        this.head.right = this.tail
        this.tail.left = this.head
```

Searching

To search a skip list, start with the top most level. If the next element in the current level is less than the target element, go to that element. Otherwise, go down a level.



Pseudocode implementation of the search operation:

```
search(value):
    current = this.head

    for level from maxLevel downto 0:
        while current.right != null and current.right.value <= value:
            current = current.right

        if current.value == value:
            return current
```

```
        if level > 0:
            current = current.below

    return null
```

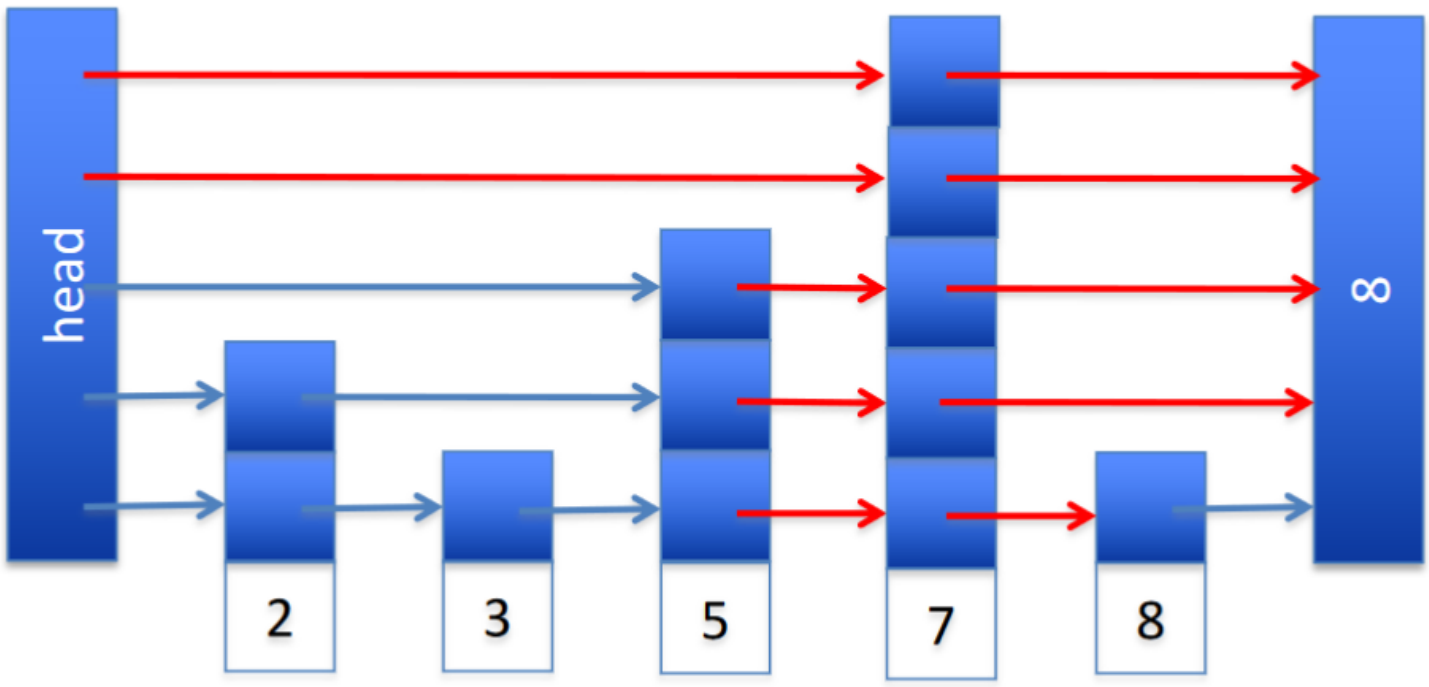
Insertion in Skip List

To insert a new element into a skip list, the steps are as follows:

1. Create a new node with the value to be inserted.
2. Flip a coin repeatedly until tails come up. The number of heads before the tails determines the level of the new node.
3. If the new node's level is higher than the skip list's current height, increase the height of the list.
4. Traverse the skip list from the highest level down to the determined level, moving right to find the appropriate insertion point.
5. At each level down to the lowest, insert the new node after the appropriate node and update the forward pointers.

Example Insert(7)

$h=5$



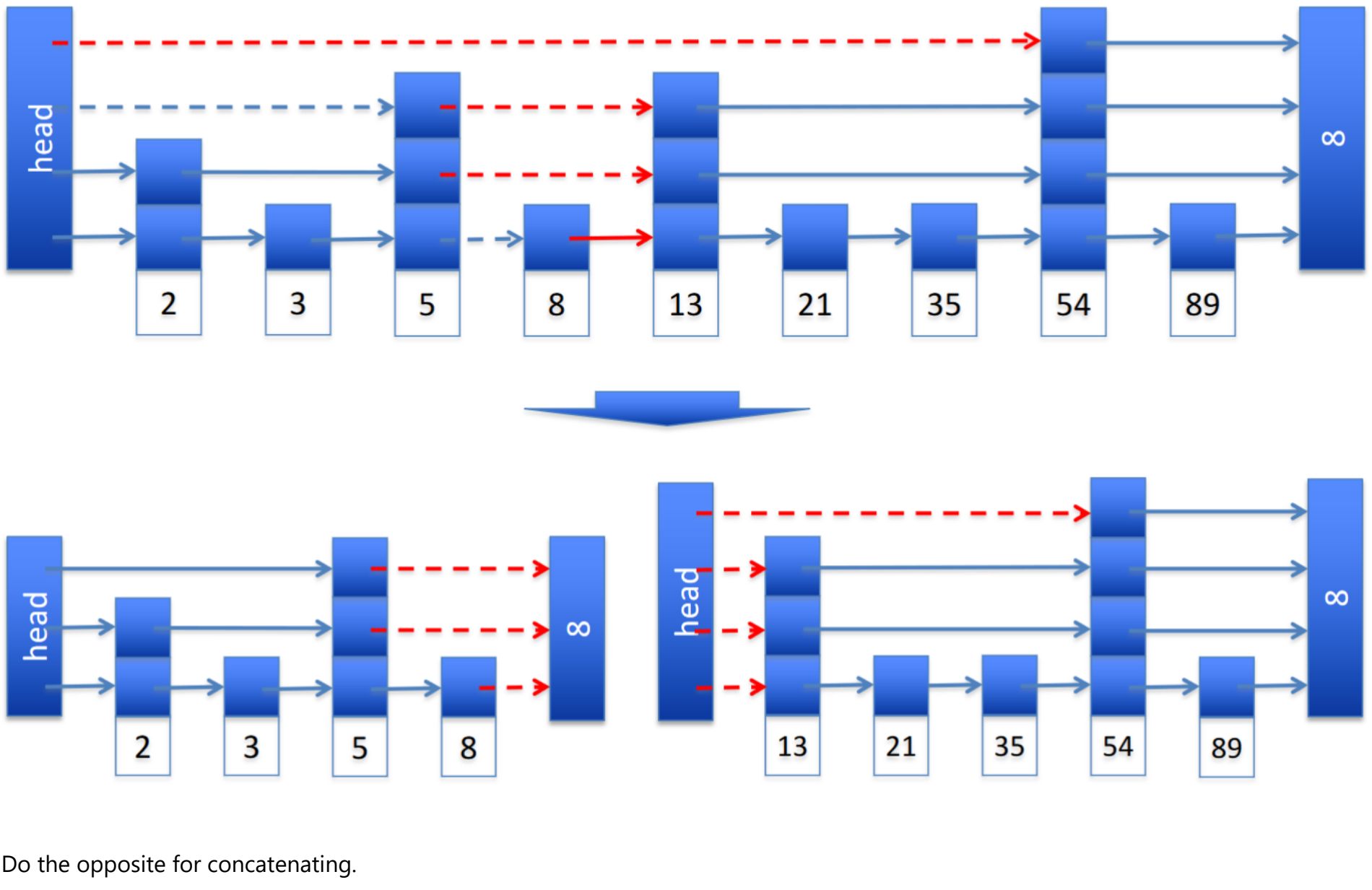
Pseudocode implementation of the insert operation:

```
insert(value):
    newNode = new Node(value)
    stack = empty stack
    current = this.head

    for level from maxLevel downto 0:
        while current.right != null and current.right.value < value:
            current = current.right

        if level < newNode.level:
            stack.push(current)

        if level > 0:
            current = current.below

    down = null

    while !stack.isEmpty():
        node = stack.pop()
        newNode.left = node
```

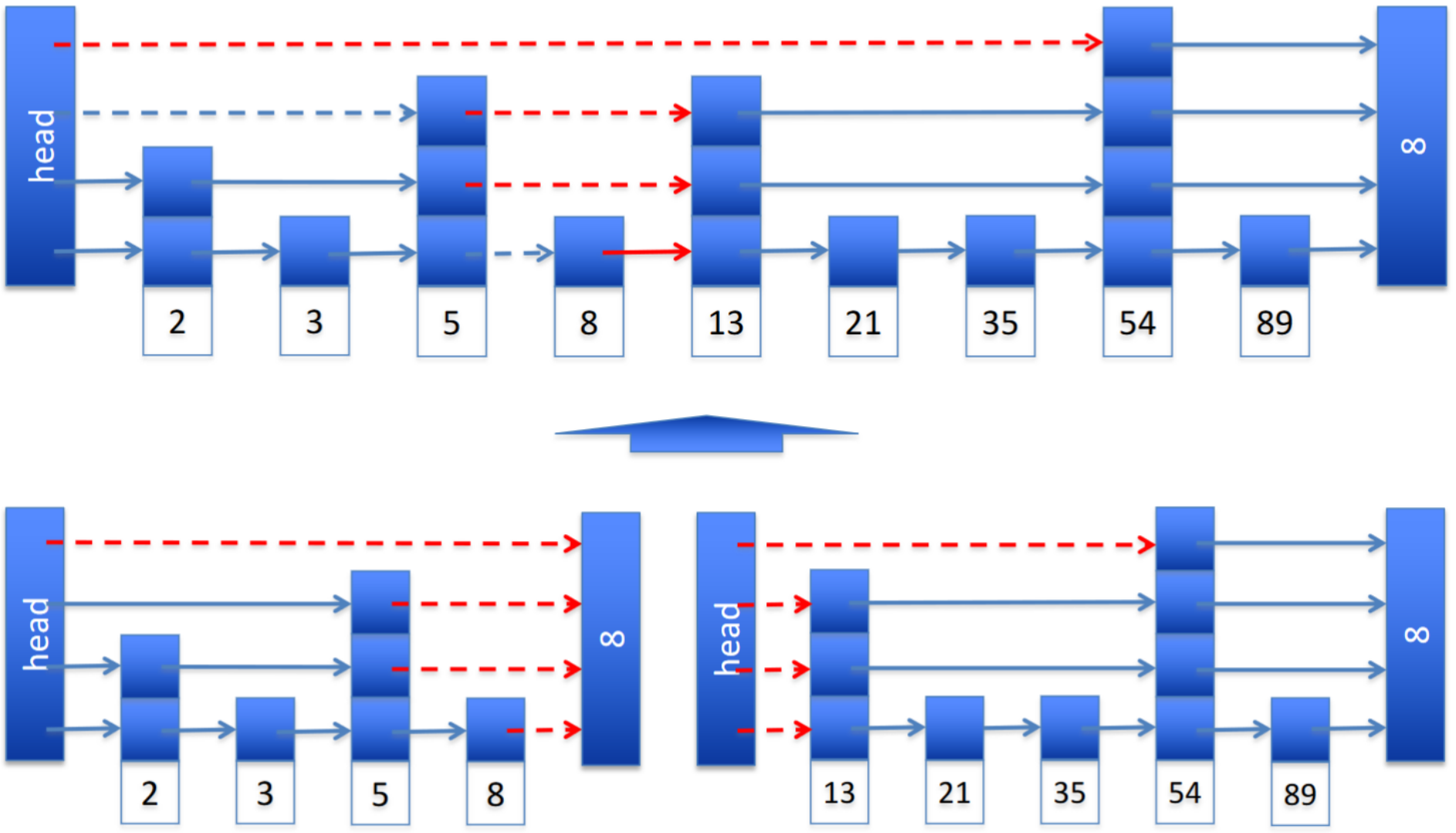
```
        newNode.right = node.right
        node.right.left = newNode
        node.right = newNode

        if down != null:
            newNode.below = down
            down.above = newNode

        down = newNode
        newNode = new Node(value)

    if this.maxLevel < newNode.level:
        this.maxLevel = newNode.level
```

Splitting / Concatenating

Search through list, and update pointers. A 2nd head points to where the end of the first list used to point to. End of the first list points to null. Splitting is really useful for large structures. Being able to store single data sets in multiple places.

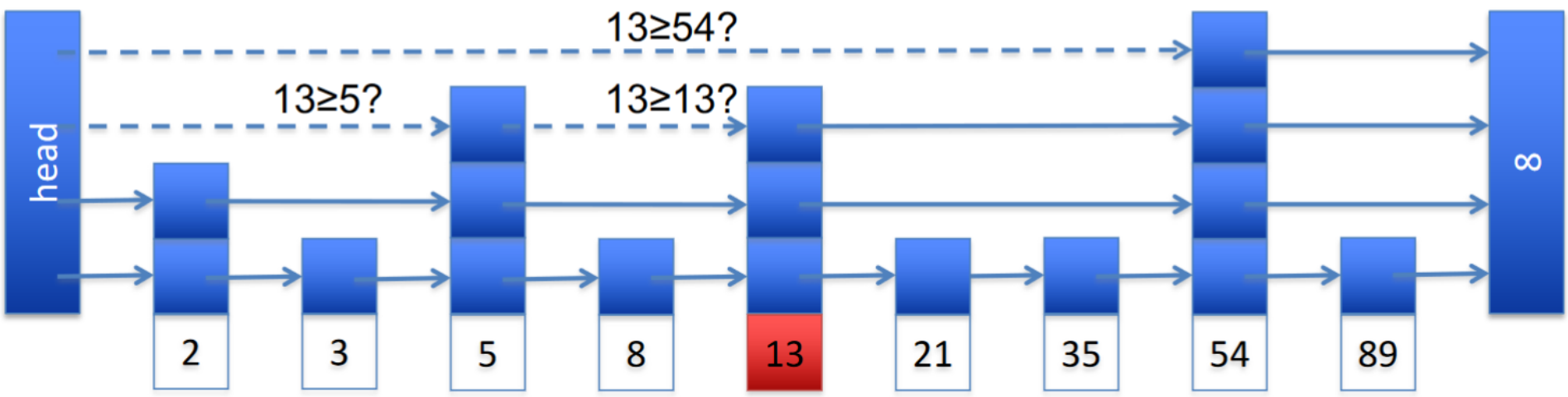


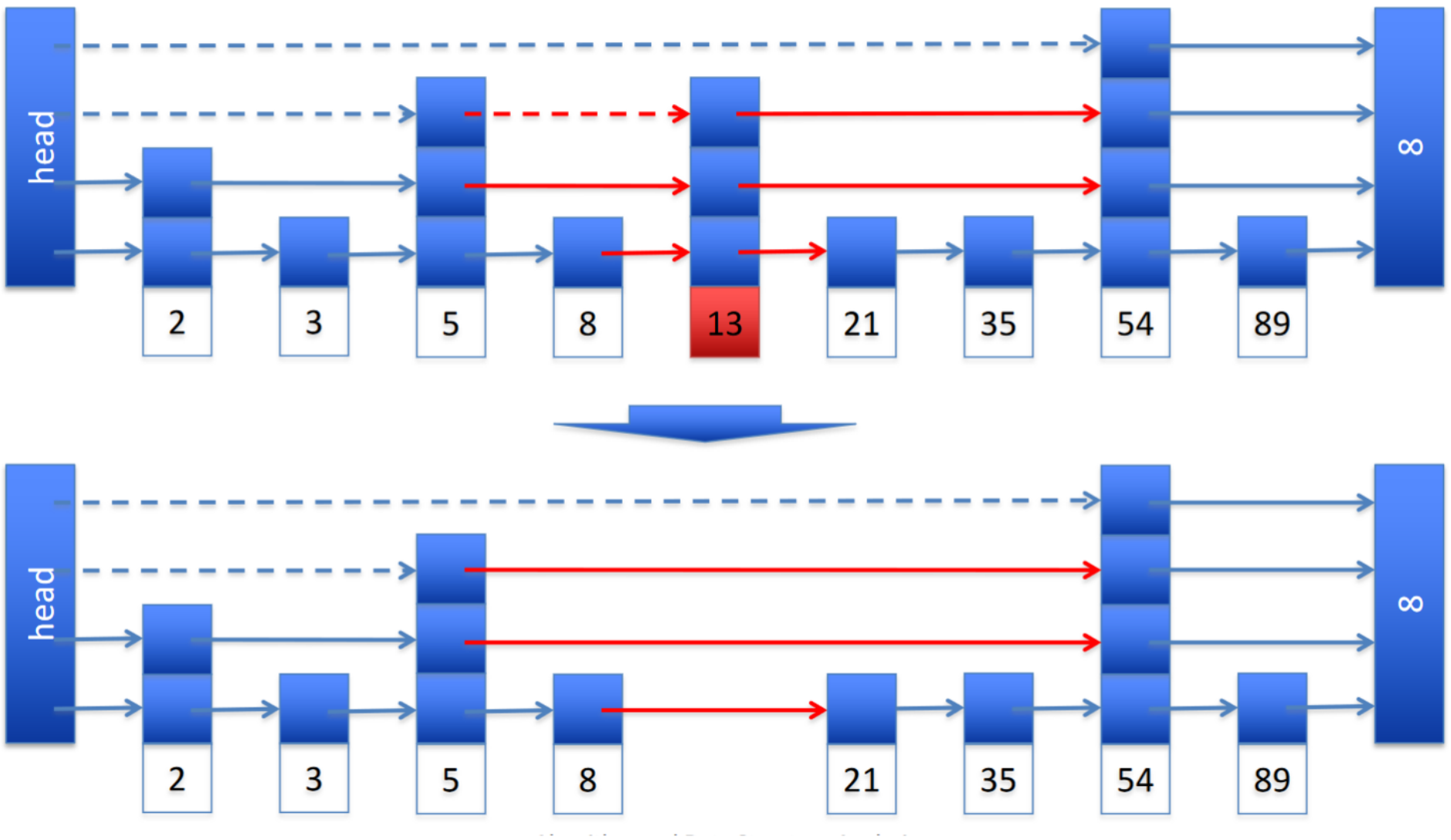
Do the opposite for concatenating.



Deletion

Search for the predecessor of the node to be deleted. Update pointers for this node to be what the deleted node points to.





7.1 Hash Tables

Hash Tables

A hash table is a dynamic data structure that allows for **efficient insertion, deletion, and searching** of elements using a **unique key**. It achieves **constant average time $O(1)$** for these basic operations by using a **hash function** to map keys to indices in an underlying array. This makes hash tables **highly useful for maintaining and manipulating sets of data** where quick access is essential.

⚠ Consider a Phone Book

Imagine storing phone numbers in an array, where the index corresponds to the person's name (an integer). While this allows for immediate access $O(1)$, it becomes impractical for large ranges of names. For example, if the highest name is "1,000,000", you would need an array of size 1,000,001, even if only a few people are listed, resulting in wasted space.

Hash tables solve this issue by using a hash function to map keys (e.g., strings) to indices in a smaller array, providing space-efficient storage and retrieval. This makes hash tables versatile and powerful, handling large key ranges without excessive memory overhead.

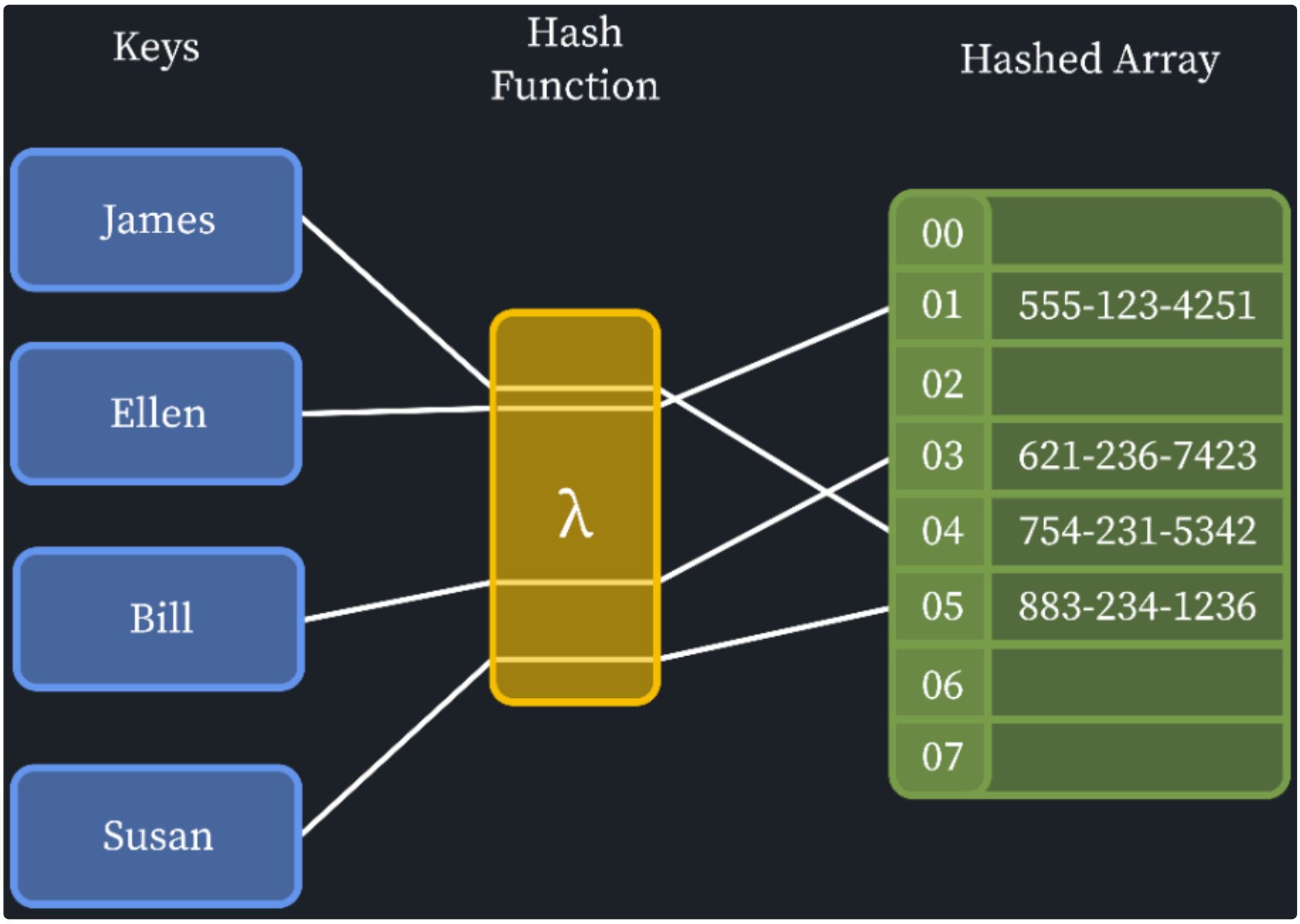
Hash Functions

A hash function **maps a given key to an index in the underlying array**. The goal is to distribute the keys evenly across the array. A well-designed hash function should be efficient to compute and provide a uniform distribution of keys.

When a key is passed through the hash function, it generates a hash code, which is then mapped to an index within the bounds of the array. This allows for **direct access to the desired element without the need to search through the entire array**.

Example: Modulo Function

- The modulo function is a simple hash function that takes the remainder of the key divided by the size of the array.
- Formula: `hash(key) = key % array_size`
- Example: If the key is 42 and the array size is 10, the hash function would map the key to index 2 (42 % 10 = 2).



⚠ **Collisions and Hashing Techniques**

Hash collisions occur when multiple keys map to the same array index. There are two main approaches to handle collisions: open hashing (chaining) and closed hashing (open addressing).

1. **Open Hashing / Closed Addressing / Hashing with Chaining**
   - Elements with the same hash are stored as a linked list in the same array slot.
   - **Advantages**:
     - Ensures referential integrity by maintaining a correct and navigable link for each key-value pair, despite collisions.
     - The average length of the list at a slot is n/m, where n is the number of elements and m is the array size.
   - **Disadvantages**:
     - Can lead to wasted space if the array size is much larger than the number of elements.
     - Requires additional memory for the linked list pointers.
2. **Closed Hashing / Open Addressing**
   - Elements with the same hash are stored in different array slots, determined by a probing sequence.
   - **Advantages**:
     - Uses contiguous memory, which can be more cache-friendly.
     - Does not require additional memory for pointers.
   - **Disadvantages**:
     - Can lead to clustering and longer probe sequences as the table fills up.
     - Deletion is more complicated, as it requires marking slots as "deleted" to maintain the probing sequence.

The choice between open hashing and closed hashing depends on the specific requirements of the application, such as memory constraints, expected load factor, and the cost of collisions.

⚠ **Hash Table Operation Complexities**

Case	Complexity	Description
Best Case	$O(1)$	Direct access as no collisions occur.
Average Case	$O(1)$	Efficient handling with minimal collisions, assuming a good hash function and low load factor.

Case	Complexity	Description
Worst Case	$O(n)$	All keys hash to the same index, leading to a linked list (if chaining is used) or long probing sequences (in open addressing).

Load Factor and Rehashing

The load factor of a hash table is a measure that indicates how full the hash table is. It's calculated as the ratio of the number of entries to the total number of slots available in the table. A high load factor means the table is getting full, leading to an increased chance of collisions and thus, a decrease in performance.

- Formula: Load Factor = (Number of Entries) / (Size of Hash Table)

Rehashing is the process of adjusting the size of the hash table and re-distributing the entries. This typically happens in two scenarios:

1. When the load factor exceeds a certain threshold, indicating the table is too full. Expanding the table size and rehashing helps spread out the entries more thinly, reducing the chance of collisions.
2. When the table becomes too sparse, usually after many deletions. This scenario is less common but can lead to unnecessary memory usage. Shrinking the table and rehashing makes it more space-efficient.

Rehashing involves creating a new hash table (usually of a larger size to accommodate more entries) and then re-inserting each existing entry. This process ensures that the hash table continues to operate efficiently, but it can be computationally expensive, affecting the time complexity of operations that trigger rehashing.

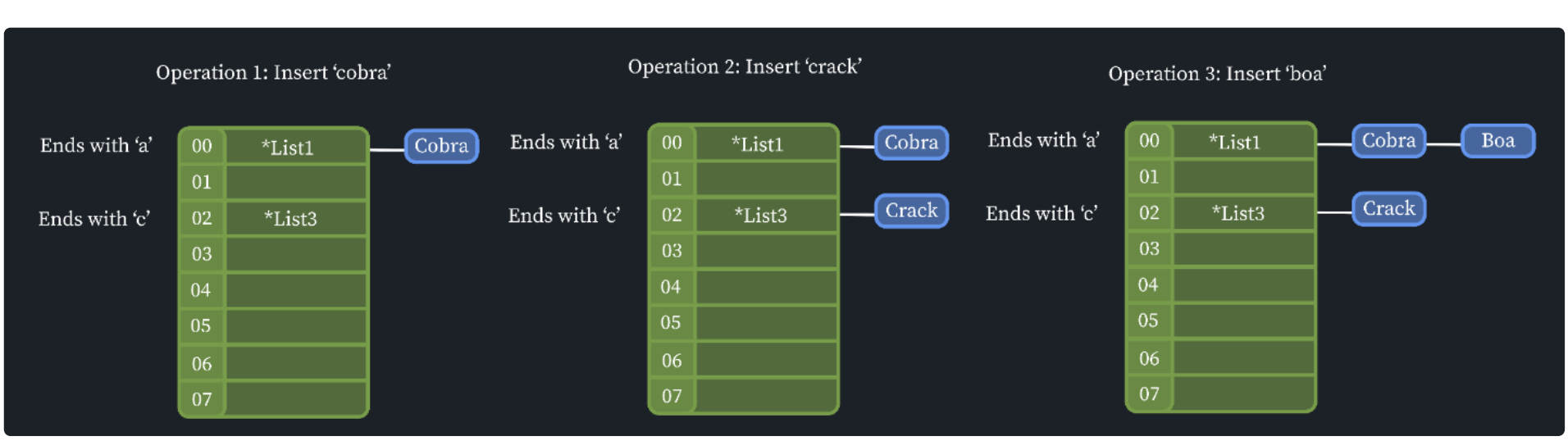
Operation	Open Hashing (Chaining)	Closed Hashing (Open Addressing)
Find	O(1 + α)	O(1 + α)
Insert	O(1 + α)	O(1 + α)
Delete	O(1 + α)	O(1 + α)

Note: α (alpha) represents the load factor, which is the ratio of the number of elements to the array size (n/m). In practice, the load factor is kept below a certain threshold (e.g., 0.75) to maintain good performance.

Open Hashing - Chaining

Each array index points to a linked list.

- When a collision happens, the key-value pair is appended to the linked list at that index, allowing multiple pairs to be stored.
- Searching for a key involves traversing the linked list at the corresponding index to find the desired value.
- Deleting a value requires adjusting the pointers in the linkedlist



Closed Hashing - Probing

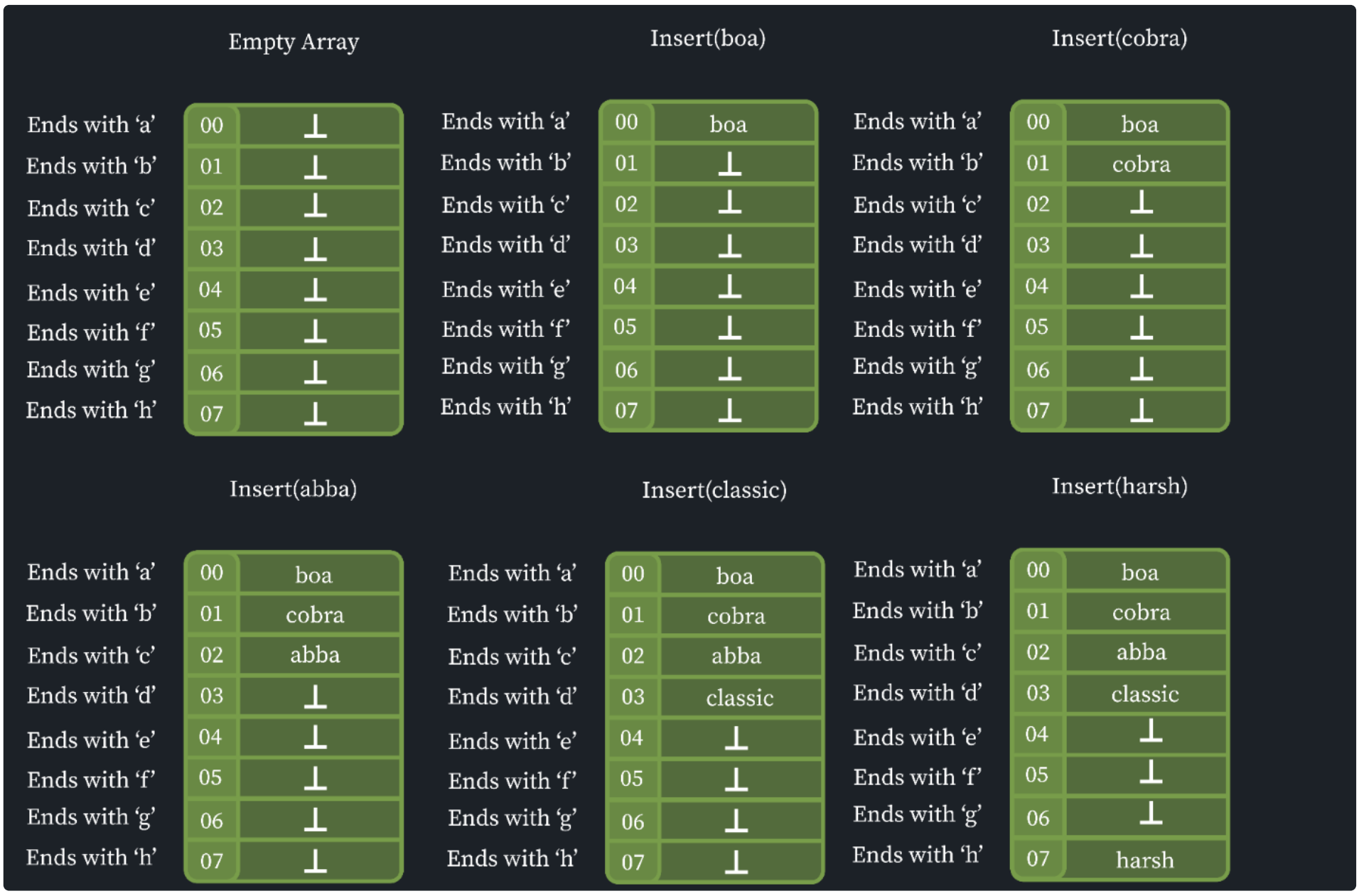
In closed hashing, when a collision occurs, probing is used to find an alternative empty slot in the array. ⊥ is used to indicate empty slots that values can be placed in.

- Common probing techniques include:
- Linear Probing: Incrementally search for the next empty slot.
- Quadratic Probing: Use quadratic function to determine the next slot.
- Double Hashing: Use a secondary hash function to calculate the step size for probing.

Insertion in Linear Probing:

- Start at the initial index obtained by applying the hash function.
- If the slot is occupied, increment to the next available slot and place the key there.

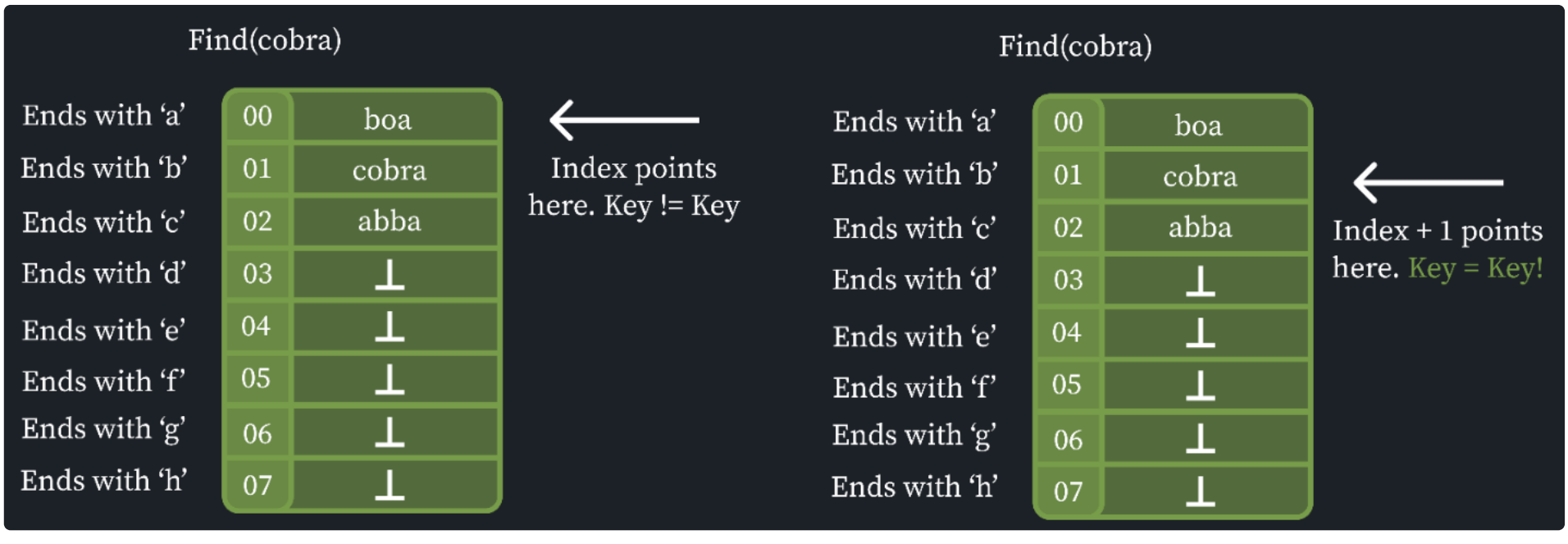
```
insert(e: Element)
   1. Get index i = h(key(e))
   2. If t[i] == ⊥, store e at t[i]
   3. If t[i] is not empty, increase i by 1 and go to step 2.
```



Search in Linear Probing:

- To search for a key in a hash table using linear probing, we start at the initial index obtained by applying the hash function to the key.
- We compare the key at the current index with the key we are searching for.
- If the keys match, we have found the desired key-value pair.
- If the keys don't match, we linearly probe to the next index.

```
search(k: Key)
   1. Get index i = h(k)
   2. If t[i] == ⊥, return null
   3. If key(t[i]) == k, return t[i]
   4. Increase i by 1 and go to step 2.
```

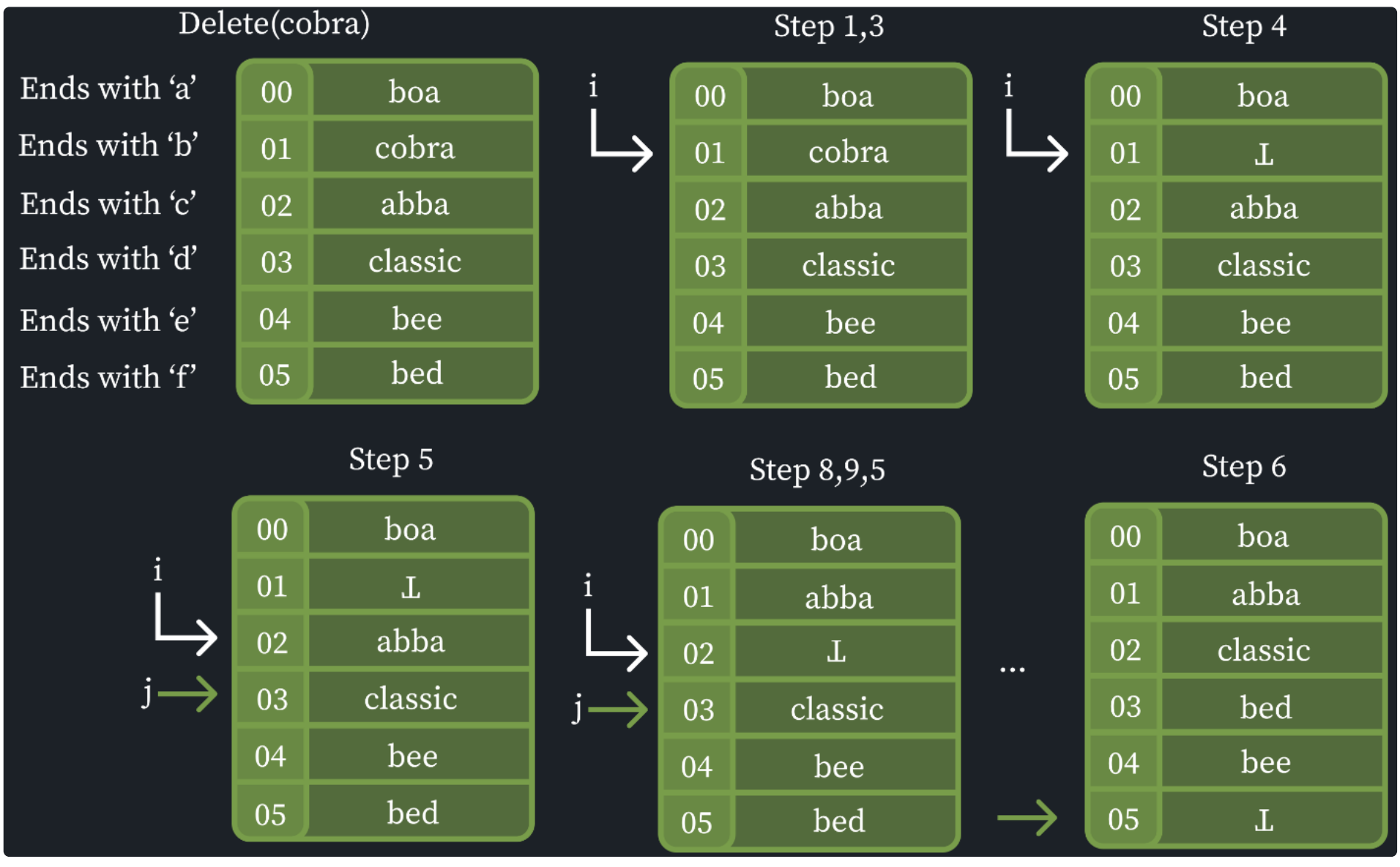


Deletion in Linear Probing:

When deleting a key-value pair in a hash table using linear probing, the process involves shifting elements to maintain the integrity of the probing sequence. Here's how the deletion process works:

1. Get the index `i` by applying the hash function `h` to the key `k`.
2. If the slot at index `i` is empty (denoted by `⊥`), the key is not present in the hash table, so we return.
3. If the element `e` at index `i` has a key different from `k`, we linearly probe to the next slot by incrementing `i` by 1 and go back to step 2.
4. If the key at index `i` matches `k`, we have found the key-value pair to be deleted. We set the slot at index `i` to `⊥` to mark it as empty.
5. We set the index `j` to `i+1` to start shifting elements.
6. If the slot at index `j` is empty (`⊥`), we have finished shifting and can return.
7. If the hash value of the element at index `j` is greater than `i`, it means the element at `j` was not probed to its current position and should not be shifted. We increment `j` by 1 and go back to step 6.
8. If the hash value of the element at index `j` is less than or equal to `i`, it means the element at `j` was probed to its current position and needs to be shifted. We move the element from index `j` to index `i` and set the slot at index `j` to `⊥`.
9. We update the index `i` to `j` and go back to step 5 to continue shifting elements.

```
delete(k: Key)
   1. Get index i = h(k)
   2. If t[i] == ⊥, return
   3. If key(t[i]) != k, increase i by 1 and go to step 2
   4. Set t[i] = ⊥
   5. Set j = i + 1
   6. If t[j] == ⊥, return
   7. If h(key(t[j])) > i, increase j by 1 and go to step 6
   8. Move t[j] to t[i], set t[j] = ⊥
   9. Set i = j and go to step 5
```



Application: 2-SUM Problem

- Input: An unsorted array `A` of `n` integers and a target sum `T`.
- Goal: Determine whether or not there are two numbers `x` and `y` in `A` such that `x + y = T`.
- Naive solution: Exhaustive search, which has a time complexity of $O(n^2)$.
- Better solution: Sort the array `A` $O(n \cdot logn)$ and use binary search $O(\log n)$ to find the complement of each number.

- Best solution: Use a hash table. For each element in the array, search for the difference of the target minus the current number. This operation takes constant time. By iterating through each number in the array, the overall time complexity becomes $O(n)$.

Dynamic Resizing

Dynamic resizing is a critical feature of modern hash table implementations, ensuring they can adapt to varying amounts of stored data while maintaining high performance and efficient space usage. This process is closely related to the concept of load factor and rehashing.

The main goals of dynamic resizing are:

- Maintain optimal performance: By adjusting the size of the hash table in response to changes in the number of stored elements, dynamic resizing helps maintain a low load factor, which is crucial for minimizing collisions and maintaining fast access times.
- Efficient use of memory: Dynamic resizing helps avoid excessive memory consumption when the number of elements is small, and ensures there's enough space to add more elements without significantly increasing the chance of collisions.

Implementations vary, but a common approach is to double the size of the hash table once the load factor exceeds a certain threshold (e.g., 0.75). Conversely, if a large number of elements are removed, reducing the size of the hash table can reclaim unused space.

While dynamic resizing is a powerful mechanism for maintaining efficiency, it's important to handle it carefully to minimize the computational cost of rehashing all keys during resize operations.

8.1 Graphs

Graphs

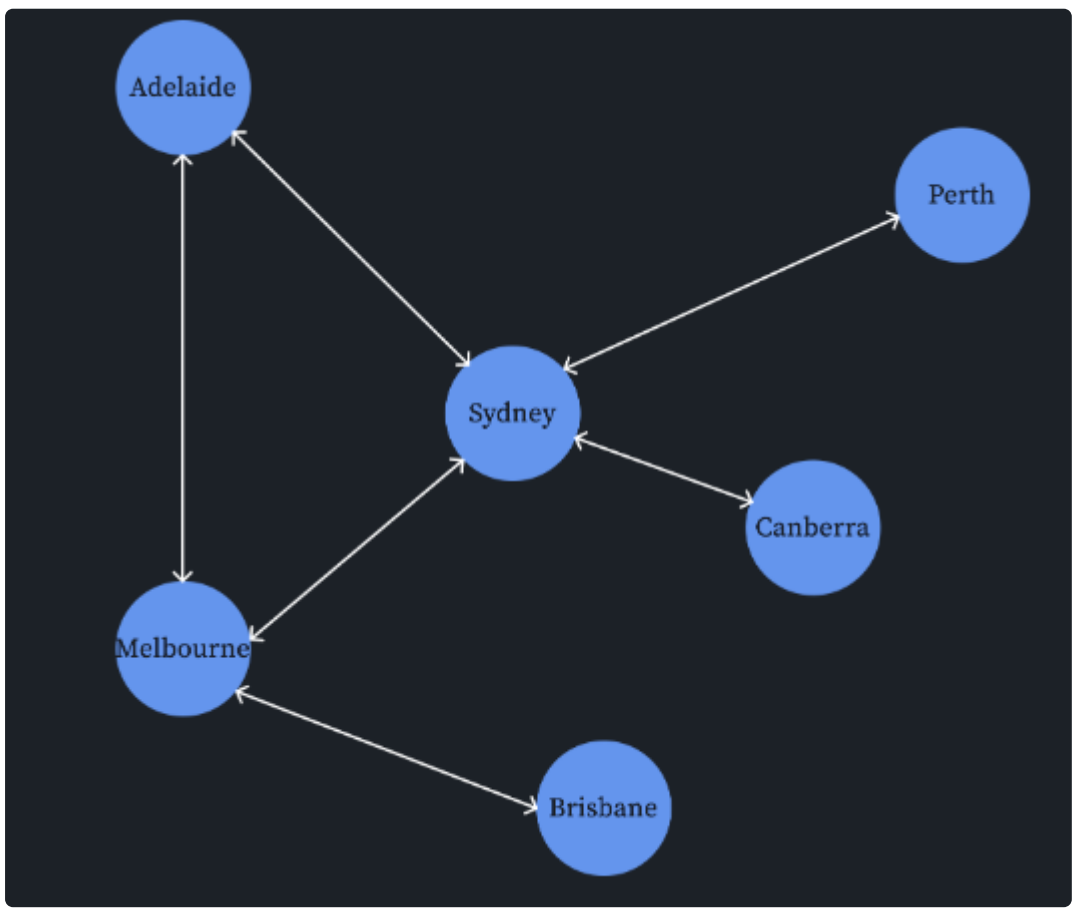
A graph is a way to represent connections between objects.

In a graph, each object is represented by a vertex (or node) , and the connections between objects are represented by edges. These edges can represent various types of connections, such as communication links, roads, or even abstract relationships like friendships or dependencies.

Real-life analogy

Imagine a map of cities (nodes) connected by flight routes (edges). This is an example of a real-world graph.

In this analogy, the cities are the nodes, and the flight routes are the edges that connect them. Each flight route has a direction (from one city to another), and there can be multiple ways to travel between cities, just like there can be multiple paths between nodes in a graph.



Graphs can represent many other real-life scenarios, such as:

- Social networks, where people are nodes and friendships are edges

- Computer networks, where computers are nodes and communication links are edges
- Road networks, where intersections are nodes and roads are edges

Graph Terminology

- Graph: A graph G is defined as an ordered pair (V, E), where V is the vertex set and E is the edge set.
- Vertex set: The set of all nodes in a graph G is called the vertex set, denoted as V.
  - $n = |V|$ represents the number of vertices in the graph.
- Edge set: The set of all edges in a graph G is called the edge set, denoted as E.
  - $m = |E|$ represents the number of edges in the graph.
  - For an edge $e = (u, v)$, we say that e is an outgoing edge of u and an incoming edge of v.
- Digraph: A pair consisting of a vertex set V and an edge set E, where each edge is an ordered pair of vertices, is called a directed graph or digraph.
- Incident edges: An edge $e = (u, v)$ is said to be incident to nodes u and v.
- Adjacent nodes: If an edge $e = (u, v)$ exists in the graph, nodes u and v are said to be adjacent.
- Self-loop: An edge that connects a node to itself, i.e., $e = (v, v)$, is called a self-loop.
- Outdegree: The outdegree of a node v, denoted as $\text{outdegree}(v)$, is the number of outgoing edges from v.
  - $\text{outdegree}(v) = |\{(v, u) \in E\}|$, where $|\cdot|$ denotes the cardinality (size) of the set.
- Indegree: The indegree of a node v, denoted as $\text{indegree}(v)$, is the number of incoming edges to v.
  - $\text{indegree}(v) = |\{(u, v) \in E\}|$, where $|\cdot|$ denotes the cardinality (size) of the set.
- Edge weights/costs: Graphs can have associated values or weights assigned to each edge, representing costs, distances, or other relevant quantities. The weight of an edge e is often denoted as c_e or w_e.
  - For example, in a weighted graph, the edge set can be represented as $E = \{(u, v, c_{uv}) \mid u, v \in V\}$, where c_{uv} is the weight of the edge (u, v).
- Strongly Connected Components: Two nodes, u and v, belong to the same strongly connected component if there is a path from u to v and from v to u.

Graph Example

Consider a directed graph $G = (V, E)$ with the following vertex and edge sets:

- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (2, 3), (3, 1), (3, 4), (4, 4)\}$

In this graph:

- The number of vertices is $n = |V| = 4$.
- The number of edges is $m = |E| = 5$.
- The edge $(1, 2)$ is an outgoing edge of vertex 1 and an incoming edge of vertex 2.
- The edge $(4, 4)$ is a self-loop.
- $\text{outdegree}(3) = |\{(3, 1), (3, 4)\}| = 2$.
- $\text{indegree}(4) = |\{(3, 4), (4, 4)\}| = 2$.
- The graph has two strongly connected components: $\{1, 2, 3\}$ and $\{4\}$.

Types of Graphs

- A directed graph, also known as a digraph, is a graph where edges have a specific direction associated with them.
  - In a directed graph $G = (V, E)$, each edge $e \in E$ is an ordered pair $e = (u, v)$, representing a connection from node $u \in V$ to node $v \in V$.
  - The edge $e = (u, v)$ is said to be directed from u to v, where u is the source (or initial) node and v is the target (or terminal) node.
  - In a directed graph, the edges are represented using arrows to indicate the direction of the connection.
    - For example, an edge (u, v) is represented as $u \to v$, showing the direction from u to v.
  - The presence of an edge (u, v) in a directed graph does not imply the existence of an edge (v, u) in the opposite direction, unless explicitly specified.
- A bidirected graph $G = (V, E)$ is a directed graph (digraph) where for every edge (u, v), there is also an edge (v, u) in the opposite direction.

- In other words, if there is a connection from node u to node v, there must also be a connection from node v to node u.
- We can write an edge in a bidirected graph as an unordered pair $\{u, v\}$ since the direction of the edge is not important.

- An undirected graph can be seen as a simplified representation of a bidirected graph.
  - In an undirected graph, edges have no specific direction, and the connection between two nodes is represented by a single edge.
  - An edge in an undirected graph is also written as an unordered pair (u, v) and $(v, u) = \{u, v\}$.

8.2 Subgraphs

Subgraphs

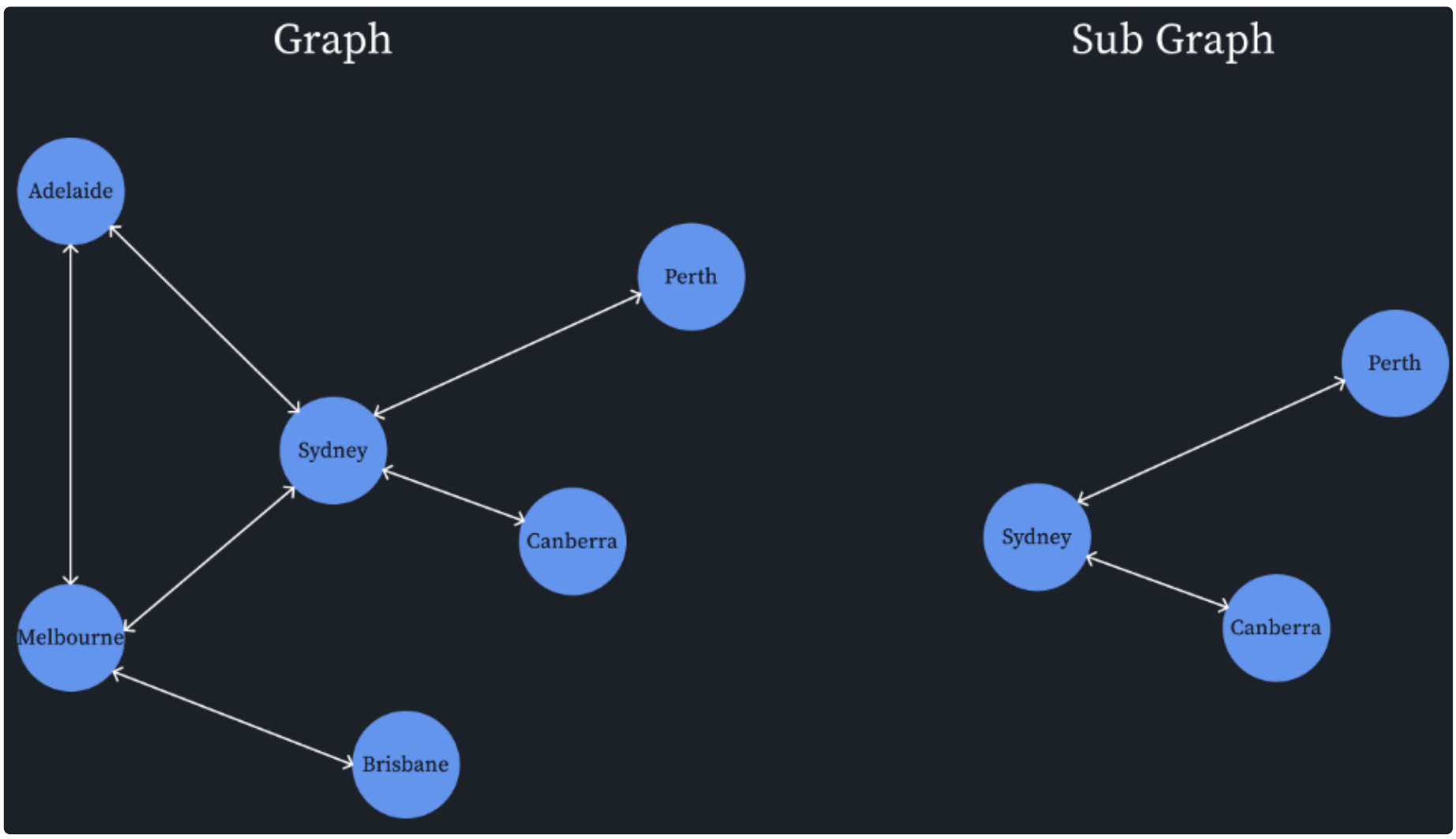
A subgraph is a graph that is contained within another graph. More formally:

- A graph $G' = (V', E')$ is a subgraph of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E$.
  - In other words, the nodes and edges of G' are subsets of the nodes and edges of G.
- Given a graph $G = (V, E)$ and a subset of nodes $V' \subseteq V$, the induced subgraph $G' = (V', E')$ is defined as:
  - $V' \subseteq V$
  - $E' = \{(u, v) \in E \mid u, v \in V'\}$
  - The induced subgraph contains all the nodes in V' and all the edges from E that connect nodes in V'.

Subgraphs are useful for focusing on specific parts or regions of a larger graph. By extracting a subgraph, we can analyze or process a smaller portion of the graph that is relevant to a particular problem or question.

⚠ **Real-life examples**

- In a social network graph, a subgraph could represent a specific community or group of people with strong connections among themselves.
- In a transportation network graph, a subgraph could represent a specific region or a particular mode of transportation, such as a subway system within a larger transportation network.

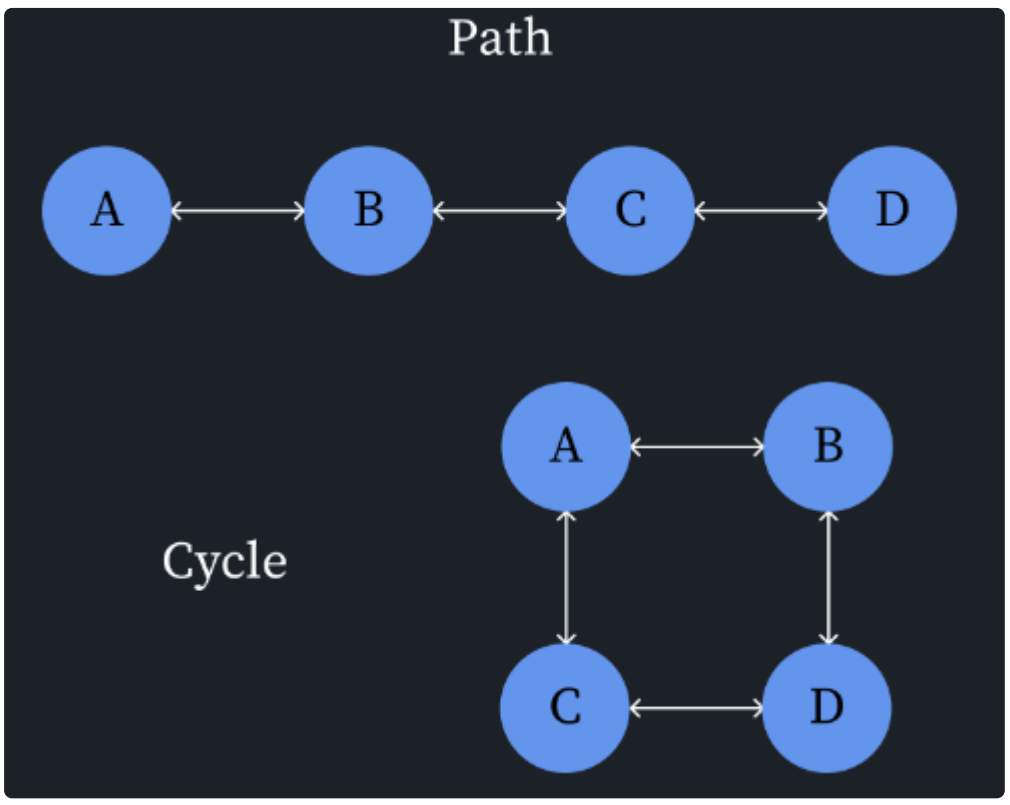


In the image above, the graph on the right is a subgraph of the larger graph on the left. The subgraph focuses on a specific region of interest within the larger graph.

8.3 Paths & Cycles

Paths and Cycles

- A path in a graph $G = (V, E)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ such that there exists an edge $(v_i, v_{i+1}) \in E$ for each consecutive pair of vertices in the sequence.
  - In other words, a path is a sequence of vertices where each consecutive pair of vertices is connected by an edge in the graph.
  - The length of a path is the number of edges traversed in the path.
- A cycle is a path where the starting and ending nodes are the same, i.e., $v_1 = v_k$.
  - In a cycle, the first and last vertices are the same, and there are no repeated edges or vertices in between.
  - The length of a cycle is the number of edges (or vertices) in the cycle.



⚠ **Simple Graph Algorithm: Detecting Acyclic Graphs**

A simple algorithm to check if a directed graph $G = (V, E)$ is acyclic (contains no cycles) is based on the following observation:

- If a graph contains a cycle, then every vertex in the cycle has at least one outgoing edge.
- Conversely, if a graph is acyclic, then there exists at least one vertex with no outgoing edges (a sink vertex).

The algorithm works as follows:

1. Initialize an empty graph G'.
2. If G is not empty, check through all nodes in G:
   - If there exists a node v in G with outdegree zero (no outgoing edges):
     - Remove v and all its incoming edges from G.
     - Add v to G'.
     - Repeat step 2.
   - Otherwise, G contains a cycle, and the algorithm terminates.
3. If G becomes empty, then the original graph G is acyclic.

The algorithm iteratively removes vertices with no outgoing edges and their incoming edges from the graph. If the graph becomes empty at the end, it means that the original graph was acyclic. If the algorithm encounters a situation where no vertex has an outdegree of zero, it implies the presence of a cycle in the graph.

8.4 Trees & Forests

Trees and Forests

- A tree is a connected undirected graph with no cycles.
  - In a tree, there exists a unique path between any pair of nodes.
  - A tree with (n) vertices has exactly (n-1) edges.
  - A tree is acyclic (it has no cycles)
- A forest is a collection of zero or more disjoint trees.
  - A forest is an undirected graph where each subgraph is a tree. Each tree in the forest is independent and not connected to any other trees in the forest.

- In a forest, a unique path exists between nodes only if they belong to the same tree. There is no path connecting nodes in different trees.
- Like individual trees, a forest has no cycles and may consist of one or several disconnected trees.

8.5 Graph Representation

Graph Operations and Storage

Working with graphs computationally involves a variety of operations that require efficient execution for practical use. It is essential to select an appropriate data structure that allows for the following operations:

- Accessing associated information: Retrieve data associated with specific nodes and edges.
- Navigation: Explore the graph by following edges from one node to another.
- Edge queries: Determine if a connection exists between two nodes.
- Construction, conversion, and output: Build new graphs, transform one graph representation into another, and output graph data.
- Update: Add or remove nodes and edges to reflect changes in the graph.

Different storage structures offer various trade-offs in terms of memory usage and the efficiency of operations. Choosing the right one depends on the specific needs of the application and the nature of the graph.

⚠ Linked List Representation

This structure involves storing edges in an unordered sequence, such as a linked list. While it's suitable for simply outputting the edges of a graph, most operations, like checking the existence of an edge, require $O(m)$ time, where m is the number of edges.

Example:

Consider the directed graph $G = (V, E)$ from the previous example:

- $V = \{1, 2, 3, 4\}$
- $E = \{(1, 2), (2, 3), (3, 1), (3, 4), (4, 4)\}$

Here's how the graph might be represented using a linked list:

$$[\text{Head}] \quad \rightarrow \quad (1, 2) \quad \rightarrow \quad (2, 3) \quad \rightarrow \quad (3, 1) \quad \rightarrow \quad (3, 4) \quad \rightarrow \quad (4, 4) \quad \rightarrow \quad [\text{Tail}]$$

In this representation:

- Each edge is stored as a node in the linked list.
- The nodes are connected in an arbitrary order.
- To output the edges of the graph, we can traverse the linked list and print each edge.
- However, operations like checking if an edge exists between two vertices would require traversing the entire list, resulting in $O(m)$ time complexity.

While the linked list representation is simple to implement, it may not be the most efficient for certain graph operations. Alternative representations, such as an adjacency matrix or adjacency list, can provide better performance for specific tasks.

⚠ Adjacency Arrays (Static Graphs Only)

The adjacency array organizes the edges of a static graph using two main components, which we can represent as follows:

- The Vertex Array (V) serves as a guide, indicating the starting point for each vertex's edges in the Edge Array.

  $$V = [1 \quad 3 \quad \ldots]$$

  In this array, the number at each position tells us where the edges for that vertex start in the Edge Array. For instance, if the first vertex's edges start at index 1 in the Edge Array, that's where we'll find it in $E[1]$.
- The Edge Array (E) lists all the edges in the graph sequentially:

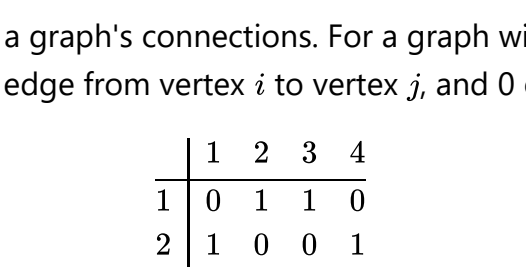
  $$E = [2 \quad 3 \quad 3 \quad 4 \quad \ldots]$$

  Here, the edges are arranged one after another for each vertex. If vertex 1 is connected to vertices 2 and 3, and vertex 2 to vertices 3 and 4, these edges are laid out in E just as listed.

Together, these arrays allow for efficient traversal and querying of the graph. However, keep in mind that inserting or deleting edges would necessitate adjustments throughout the Edge Array, which is why this structure is best suited for graphs that do not change often.

Adjacency Lists

Adjacency lists store nodes inside a node array. Each element inside this array points to a linked list which contains the edges for that node.



Advantages:

- Edge insertion and deletion can be done in $O(1)$ time.
- Memory usage is proportional to the number of edges, making it efficient for sparse graphs.

Disadvantages:

- Edge queries (checking if an edge exists) require traversing the linked lists, taking $O(n)$ time in the worst case, where n is the number of vertices.
- Not as cache-friendly as adjacency matrices due to the non-contiguous memory layout of linked lists.

Adjacency lists strike a balance between space efficiency and the ability to modify the graph structure dynamically, making them a popular choice for many graph algorithms.

Adjacency Matrices

An adjacency matrix is a 2D array that represents a graph's connections. For a graph with n vertices, the matrix has dimensions $n \times n$. The entry at row i and column j is 1 if there is an edge from vertex i to vertex j, and 0 otherwise.

	1	2	3	4
1	0	1	1	0
2	1	0	0	1
3	1	0	0	1
4	0	1	1	0

Advantages:

- Edge queries can be done in $O(1)$ time by checking the corresponding matrix entry.

Disadvantages:

- Space complexity is $O(n^2)$, even for sparse graphs with few edges.
- Adding or removing vertices requires resizing the matrix, which can be costly.

Adjacency matrices are preferred when the graph is dense (many edges) and fast edge queries are crucial. However, for large, sparse graphs, the space overhead can be significant.

8.6 Graph Traversal

⚠ Graph Traversal

Graph traversal is the process of **visiting each node in a graph**. It is a fundamental operation in graph algorithms and is used for tasks such as searching for a specific vertex, determining connectivity, or exploring the structure of the graph.

Efficient graph traversal algorithms should be able to visit all vertices in a graph in **linear time, i.e., $O(V + E)$**, where V is the number of vertices and E is the number of edges.

There are two primary approaches to graph traversal:

1. Breadth-First Search (BFS)
2. Depth-First Search (DFS)

Breadth-First Search (BFS)

Breadth-First Search is a graph traversal algorithm that **explores the graph level by level**, ensuring it visits all vertices at the present "breadth" or layer before moving on to nodes at the next layer.

The key idea behind BFS is to use a queue data structure to keep track of the vertices to be visited. The algorithm works as follows:

1. Choose a starting node and add it to the queue.
2. While the queue is not empty:
   - Dequeue a vertex from the queue.
   - Mark the dequeued vertex as visited.
   - Enqueue all the unvisited neighbours of the dequeued vertex.
3. Repeat step 2 until the queue is empty.

Implementation Details:

- We can use an Adjacency Array representation of the graph to efficiently access the neighbors of each node.
- We introduce each node into the Priority Queue only once, which takes $O(n)$ time, where n is the number of nodes.
- We only consider a node in the Priority Queue together with its edges once, which takes $O(n+m)$ time, where m is the number of edges.
- Updating the distance vector and the parent vector is done once for every node, taking O(n) time.
- The total runtime of BFS is $O(n+m)$.

Pseudocode:

```
function BFS(graph, startNode):
    queue = new Queue()
    visited = new Array(graph.numNodes).fill(false)
    distance = new Array(graph.numNodes).fill(infinity)
    parent = new Array(graph.numNodes).fill(null)

    queue.enqueue(startNode)
    visited[startNode] = true
    distance[startNode] = 0

    while queue is not empty:
        currentNode = queue.dequeue()
```

```
        for each neighbor of currentNode:
            if not visited[neighbor]:
                queue.enqueue(neighbor)
                visited[neighbor] = true
                distance[neighbor] = distance[currentNode] + 1
                parent[neighbor] = currentNode

    return (visited, distance, parent)
```

The `BFS` function takes the graph and the starting node as input. It initializes a queue, a visited array, a distance array (to store the shortest distance from the start node to each node), and a parent array (to store the parent of each node in the BFS tree).

The starting node is enqueued, marked as visited, and its distance is set to 0. Then, the algorithm enters a loop that continues until the queue is empty. In each iteration, it dequeues a node, marks it as visited, and enqueues all its unvisited neighbors. It also updates the distance and parent information for each newly visited node.

Finally, the function returns the visited array, distance array, and parent array, which can be used for further analysis or to reconstruct the shortest paths from the start node to any other node in the graph.

Depth-First Search (DFS)

Depth-First Search is a graph traversal algorithm that **explores the graph as far as possible along each branch before backtracking**. It visits the deepest nodes in the graph first before gradually visiting nodes at shallower depths.

The key idea behind DFS is to use a stack data structure (often implemented recursively) to keep track of the nodes to be visited. The algorithm works as follows:

1. Choose a starting node and push it onto the stack.
2. While the stack is not empty:
    - Pop a node from the stack.
    - If the popped node is not visited:
        - Mark the node as visited.
        - Push all the unvisited neighbors of the node onto the stack.
3. Repeat step 2 until the stack is empty.

Implementation Details:

- We can use an Adjacency Array representation of the graph to efficiently access the neighbors of each node.
- We introduce each node into the stack only once, which takes $O(n)$ time, where n is the number of nodes.

- We only consider a node in the stack together with its edges once, which takes $O(n+m)$ time, where m is the number of edges.
- Marking each node as visited takes $O(n)$ time.
- The total runtime of DFS is $O(n+m)$.

Pseudocode:

```
function DFS(graph, startNode):
    stack = new Stack()
    visited = new Array(graph.numNodes).fill(false)

    stack.push(startNode)

    while stack is not empty:
        currentNode = stack.pop()

        if not visited[currentNode]:
            visited[currentNode] = true

            for each neighbor of currentNode:
                if not visited[neighbor]:
                    stack.push(neighbor)

    return visited
```

The `DFS` function takes the graph and the starting node as input. It initializes a stack and a visited array to keep track of the visited nodes.

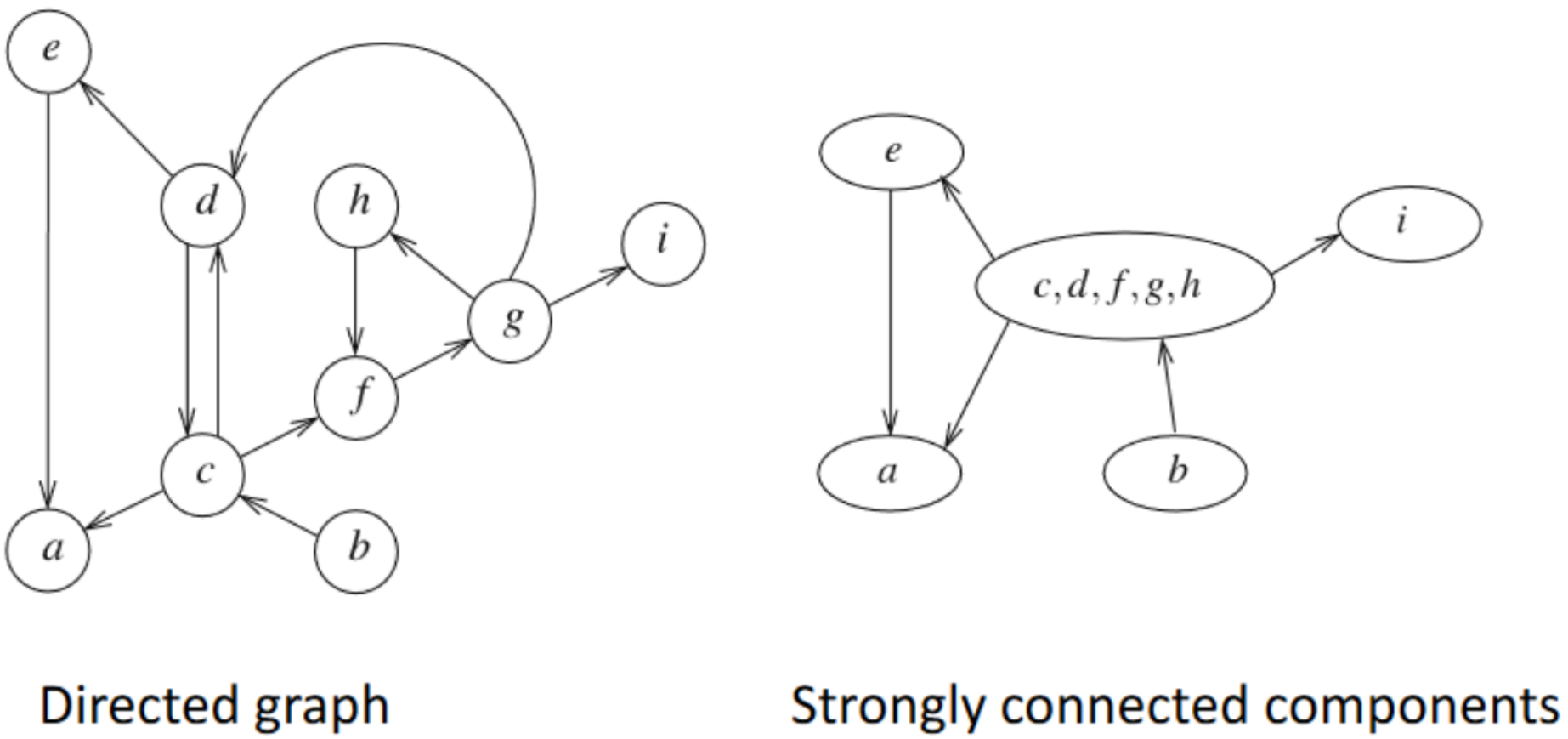
The starting node is pushed onto the stack. Then, the algorithm enters a loop that continues until the stack is empty. In each iteration, it pops a node from the stack. If the popped node is not visited, it marks it as visited and pushes all its unvisited neighbors onto the stack.

Finally, the function returns the visited array, which indicates the nodes that were visited during the DFS traversal.

8.7 Kosaraju's Algorithm

Strongly Connected Components

Strongly Connected Components (SCCs) refer to subgraphs in a directed graph where each vertex is reachable from every other vertex.



Finding Connected Components in an Undirected Graph

Breadth First Search (BFS) is an effective method for identifying all connected components in an undirected graph:

- **Process**:
  1. Initialize all vertices as unvisited.
  2. For each unvisited vertex, perform a BFS to mark all reachable vertices.
  3. Each BFS from an unvisited vertex identifies all the vertices in one connected component.
  4. Repeat until all vertices have been visited and associated with a component.
- **Example**: In an undirected graph with vertices A, B, C, D, and E where edges connect A-B, B-C, and D-E:
  - Starting BFS from vertex A will visit A, B, and C, marking them as one connected component.
  - Starting BFS from D, the next unvisited vertex, will identify D and E as another component.

⚠ Challenges of Finding SCCs in Directed Graphs

In **directed graphs**, identifying strongly **connected components (SCCs) is complex** due to the unidirectional nature of edges. Methods like Breadth First Search (BFS), suitable for undirected graphs, fail here because they do not consider edge directionality, leading to potential misidentification of SCCs. BFS assumes mutual connectivity, which is not guaranteed in **directed scenarios where a path from A to B doesn't imply a path from B to A**.

Kosaraju's Algorithm

Kosaraju's algorithm utilises a two-pass DFS process to find SCCs in directed graphs:

1. **First DFS Pass (DFS-1)**:
   - **Purpose**: To determine the traversal finish order of nodes based on their exploration depth. This ordering helps in prioritizing nodes for the next DFS pass.
   - **Process**: Start a DFS from any unvisited node and explore as deep as possible. Record when you finish exploring each node (when it's added to the visited list and all possible explorations are complete).
2. **Second DFS Pass (DFS-2)**:
   - **Preparation**: Before this pass, reverse the direction of every edge in the graph. For example, if there is a directed edge from A to B in the original graph, it would be reversed to B to A. This reversal is key to identifying the SCCs.
   - **Purpose**: To discover groups of nodes that are mutually reachable under the reversed edge conditions.
   - **Process**: Using the finish order from DFS-1 (starting with the node that finished last), conduct another DFS. The set of nodes you visit before you can no longer traverse further forms one SCC. Once an SCC is fully traversed, move to the next highest node in the finish order that has not yet been visited, and repeat the process.

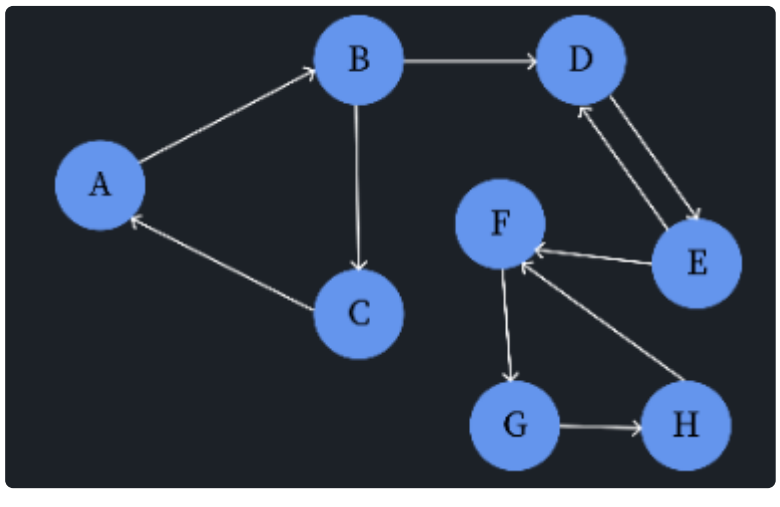
This method ensures mutual reachability within SCCs, both in the original and the reversed graph orientations, by leveraging the finish order to guide efficient exploration.

Using Kosaraju's Algorithm to Identify SCCs in Directed Graphs

Depth First Search (DFS) is crucial for identifying strongly connected components (SCCs) in directed graphs. The process involves two passes:

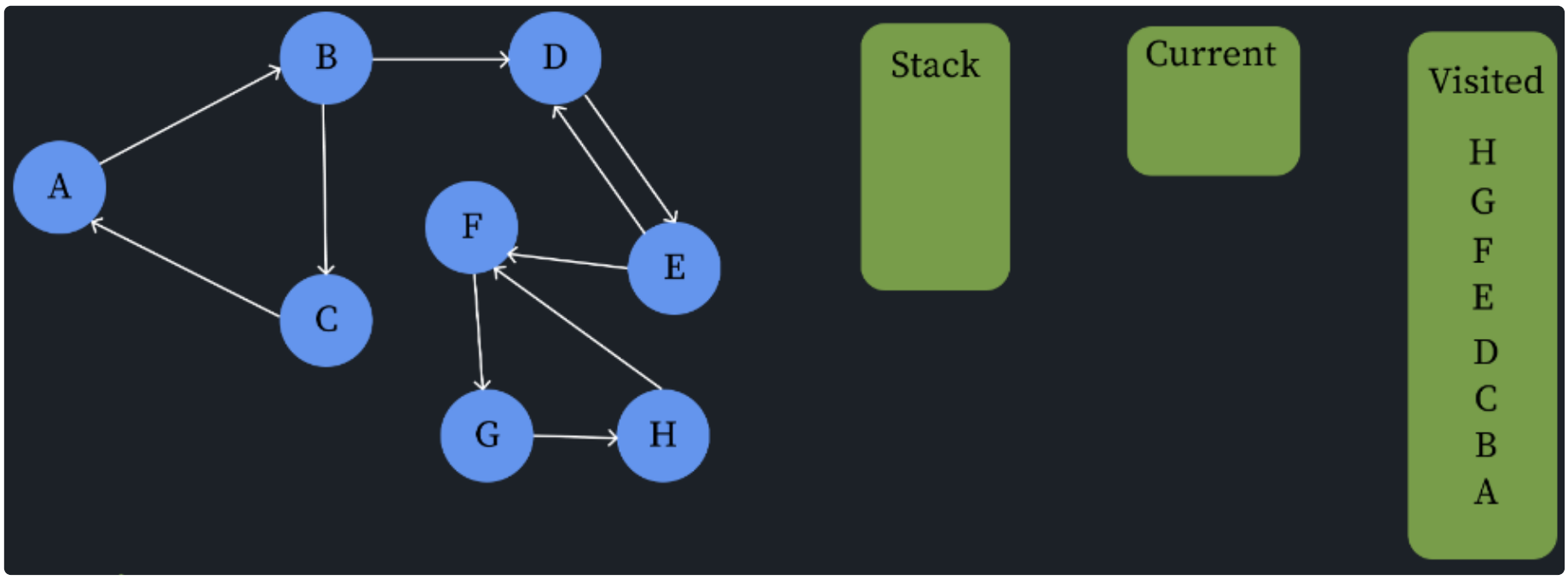
- **First DFS Pass (DFS-1)**: Begins from any unvisited node, exploring as deeply as possible and recording the order of completion.
- **Second DFS Pass (DFS-2)**: After reversing the graph's edges, performs DFS from the node with the highest finish order, identifying SCCs with each traversal.

Example: Consider a directed graph with nodes A, B, C, D, E, F, G, and H with the following edges:

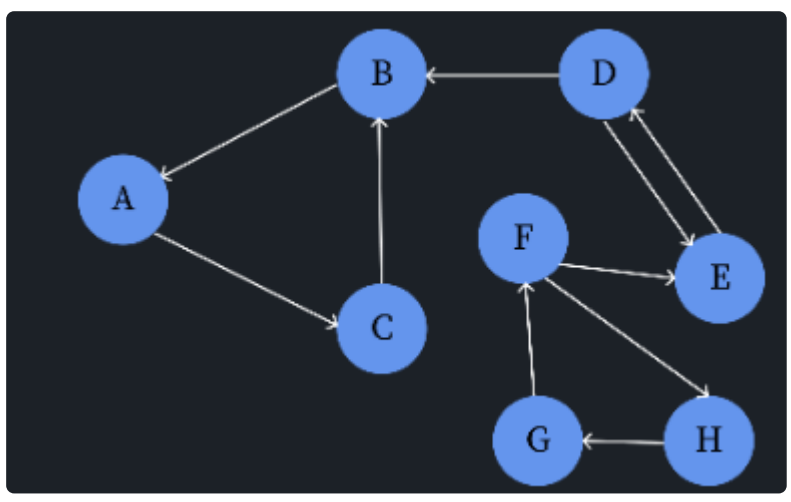


- **First Pass**:

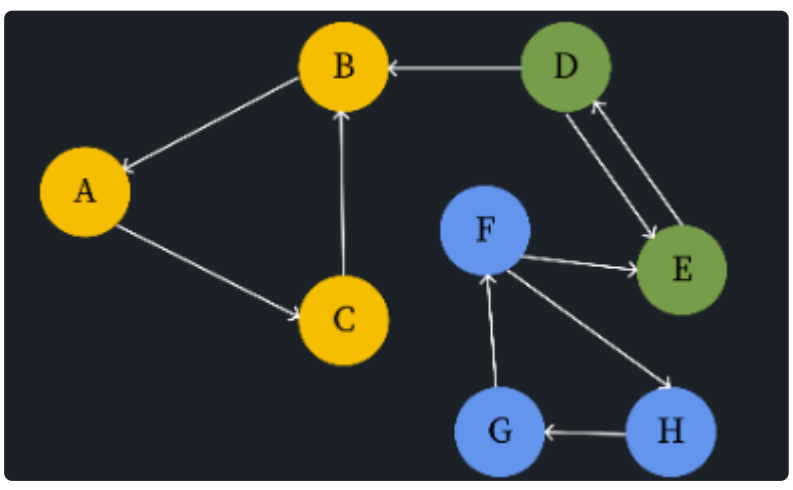
- Start DFS at A, exploring A → B → C, then back to A, marking this entire loop as visited.
- Move to the next unvisited node, D, exploring D → E and back to D.
- Finally, start at F, exploring F → G → H and back to F.



- Reversal:
  - Reverse all edges: A → C becomes C → A, B → A becomes A → B, and so on.



- Second Pass:
  - Begin DFS at the node that finished last in the first pass - H. The traversal covers H, G and F, identifying them as part of one SCC.
  - Next, begin at E, covering E and D, marking these as another SCC.
  - Finally, start at A (or any of the last remaining highest finished nodes), covering A, B, and C, confirming these as the third SCC.



This example shows how Kosaraju's algorithm identifies each SCC through systematic exploration and re-exploration after reversing the graph's edges, ensuring each set of mutually reachable nodes is grouped together.

Complexity of Kosaraju's Algorithm

Kosaraju's Algorithm for identifying strongly connected components (SCCs) in directed graphs operates with a time complexity of $O(V + E)$, where V represents the number of vertices and E represents the number of edges. This linear complexity is due to its two-pass DFS process:

- First DFS Pass (DFS-1): Visits each vertex and edge once to determine the nodes' finish order.
- Second DFS Pass (DFS-2): After reversing all edges, another DFS is conducted from the node that finished last, effectively identifying SCCs.

Each edge and vertex are processed only once per pass, ensuring the algorithm's operations stay within $O(V + E)$, making it suitable for large graphs.

Impact of SCCs in Google's PageRank Algorithm

One of the most impactful applications of strongly connected components has been in the development of Google's PageRank algorithm. Initially, PageRank helped revolutionize web searches by using the link structure of the web as an indicator of an individual page's quality. Here's how SCCs contributed:

- **Understanding Web Structure**: Google's crawler uses concepts similar to SCCs to understand the clustering of web pages. By identifying SCCs within the vast network of web pages, PageRank could better assess the importance of a page based on the cluster's characteristics, such as size and connectivity.
- **Enhancing Search Relevance**: By analyzing the SCCs, PageRank could differentiate between sets of highly interconnected pages (like link farms) and genuinely authoritative pages, improving the relevance of search results.

The ability to map and evaluate the web's network structure through SCCs allowed Google to deliver more accurate, contextually relevant search results, fundamentally shifting the dynamics of search engine technology.

9.1 Shortest Paths

Shortest Paths

In a graph, the **route that minimises the total weight of the edges between any two nodes** is the shortest path between those two nodes.

In mathematical terms:

- We have a graph G with nodes V and edges E.
- Each edge has a weight associated with it.
- Weight can be distance, cost, etc.
- The shortest path **minimises the sum of the edge weights along the path**.

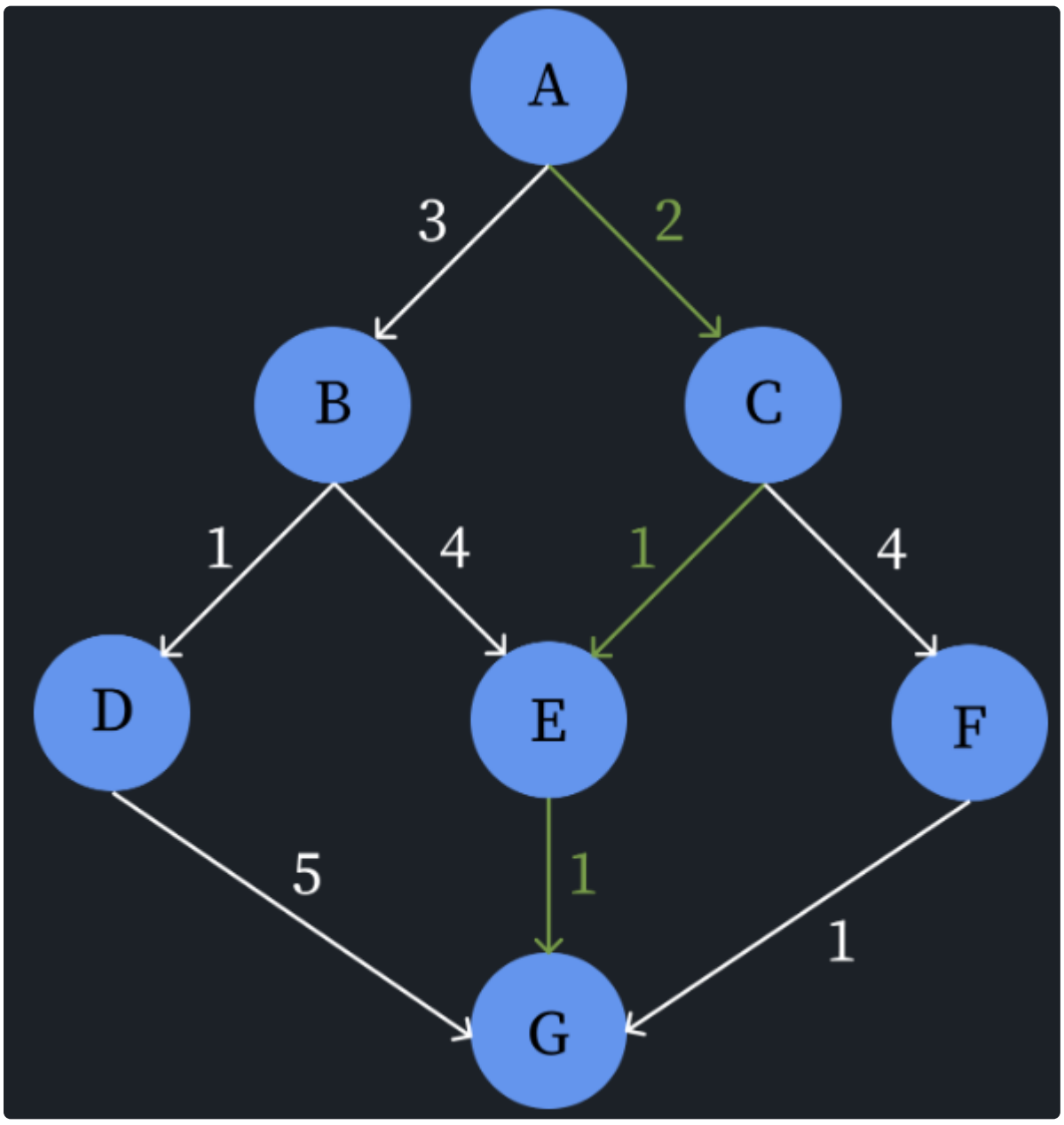
Real-World Applications

Shortest path algorithms have numerous practical applications:

- **Navigation systems**: Finding the quickest route between locations.
- **Network routing**: Determining efficient paths for data packets in computer networks.
- **Robot motion planning**: Calculating optimal paths for robots to navigate environments.

In these scenarios, the graph represents the network or environment, vertices represent locations or nodes, and edge weights represent distances or costs. Finding the shortest path helps optimize routes, minimize travel time, or reduce resource consumption.

Shortest Path Example



- A -> B -> E -> G = 3 + 4 + 1 = 8
- A -> C -> E -> G = 2 + 1 + 1 = 4
- A -> C -> F -> G = 2 + 4 + 1 = 7

Properties of Shortest Paths

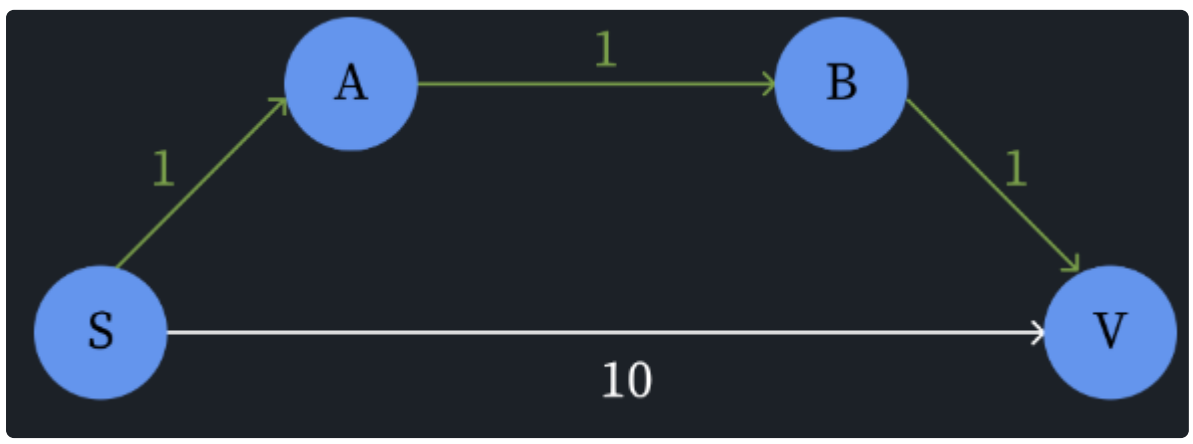
A segment of the shortest path between two nodes is itself the shortest path between any two nodes located within that segment.

Mathematically, if the shortest path from node s to node v passes through an intermediate node u, then:

- The portion of the path from s to u is the shortest path from s to u.
- The portion of the path from u to v is the shortest path from u to v.

This property is crucial for understanding and developing shortest path algorithms. It allows algorithms to build shortest paths incrementally by extending known shortest paths.

Subpath Property Illustration



In the image, if the shortest path from s to v is s → a → b → v, then:

- The subpath s → a → b is the shortest path from s to b.
- The subpath a → b → v is the shortest path from a to v.

The subpath property ensures that the shortest path from s to v can be constructed by combining the shortest paths from s to b and from a to v.

Comparison of Shortest Path Algorithms

Algorithm	Time Complexity	Space Complexity	Negative Edges	Negative Edge Weight Cycles	All Pairs Shortest Path
Dijkstra (Binary Heap)	$O((V+E)\log V)$	$O(V)$	No	Not Applicable	No
Dijkstra (Fibonacci Heap)	$O(V\log V+E)$	$O(V)$	No	Not Applicable	No
Bellman-Ford	$O(VE)$	$O(V)$	Yes	Can Detect	No
Floyd-Warshall	$O(V^3)$	$O(V^2)$	Yes	Can Detect	Yes

- Dijkstra's Algorithm:
  - Efficient for graphs with non-negative edge weights.
  - Time complexity depends on the priority queue implementation (binary heap or Fibonacci heap).
  - Cannot handle negative edge weights.
  - Finds shortest paths from a single source vertex to all other vertices.
  - Not applicable for detecting negative edge weight cycles.
- Bellman-Ford Algorithm:
  - Can handle graphs with negative edge weights.
  - Slower than Dijkstra's algorithm.
  - Finds shortest paths from a single source vertex to all other vertices.
  - Can detect negative edge weight cycles.
- Floyd-Warshall Algorithm:
  - Solves the all-pairs shortest path problem.
  - Can handle graphs with negative edge weights.
  - Has a higher time complexity compared to Dijkstra's and Bellman-Ford algorithms.
  - Uses dynamic programming to find shortest paths between all pairs of vertices.
  - Can detect negative edge weight cycles.

The choice of algorithm depends on the specific requirements of the problem, such as the presence of negative edge weights, the need for all-pairs shortest paths, the ability to detect negative edge weight cycles, and the size and density of the graph.

⚠ Non-Negative Edge Costs

Many shortest path algorithms, such as Dijkstra's algorithm, assume that all edge weights in the graph are non-negative. This means that the weight of each edge is greater than or equal to zero.

In mathematical terms:

- For every edge (u, v) in the graph, the weight w(u, v) ≥ 0.

The assumption of non-negative edge weights simplifies the problem and allows for more efficient algorithms. It ensures that as we explore the graph, the shortest path distances cannot decrease.

⚠ Negative Cycles and Shortest Path Algorithms

Shortest path algorithms, such as Dijkstra's algorithm and Bellman-Ford algorithm, assume that there are no negative cycles in the graph. A negative cycle is a cycle in the graph where the sum of the edge weights along the cycle is negative.

The presence of negative cycles can cause these algorithms to fail or produce incorrect results. Here's why:

1. **Dijkstra's Algorithm:**
   - Dijkstra's algorithm relies on the property that the shortest path from the source to any vertex always passes through vertices with non-decreasing shortest path estimates.
   - If there is a negative cycle, the algorithm may keep updating the shortest path estimates of vertices in the cycle indefinitely, as each traversal of the cycle would decrease the estimate further.

- This leads to the algorithm getting stuck in an infinite loop and failing to terminate.

2. **Bellman-Ford Algorithm:**
   - The Bellman-Ford algorithm can detect negative cycles by checking if the shortest path estimates continue to change after `|V| - 1` iterations, where `|V|` is the number of vertices.
   - However, if there is a negative cycle reachable from the source vertex, the algorithm will report that there is no shortest path because the shortest path distance to vertices in the cycle can be arbitrarily small by repeatedly traversing the negative cycle.

In the presence of negative cycles, the concept of the shortest path becomes ill-defined, as the path length can be made arbitrarily small by repeatedly traversing the negative cycle. Therefore, these algorithms are not applicable in such scenarios.

To handle graphs with negative cycles, specialized algorithms like the Floyd-Warshall algorithm can be used to detect the presence of negative cycles and report them explicitly. Alternatively, the graph can be preprocessed to remove negative cycles before applying shortest path algorithms.

9.2 Dijkstra's Algorithm

Dijkstra's Algorithm

Dijkstra's algorithm is a way to **find the shortest path from a starting node to all other nodes in a graph**, where each edge has a non-negative weight. The algorithm **keeps track of the distances from the starting node to each node in the graph** and updates these distances as it progresses.

Here's how the algorithm works:

1. Maintain two sets - visited nodes and unvisited nodes.
2. Choose a starting node and mark it as visited. Consider the distance from the starting node to itself as 0 and the distances to all other nodes as infinity.
3. Look at the edges coming out of the starting node. Update the distances to the nodes connected by these edges. If there's no edge between the starting node and a particular node, the distance remains infinity.
4. For each node updated in step 3, record the starting node as its predecessor.
5. Pick the unvisited node with the smallest distance from the starting node and mark it as visited.
6. Update the distances to the other nodes by considering the edges coming out of the newly visited node. If the distance to a node through the newly visited node is smaller than the current distance, update the distance and set the newly visited node as the predecessor for these nodes.
7. Repeat steps 3 through 6 until all nodes are visited or the destination node is reached.
8. To reconstruct the shortest path to any node, trace back from the node to the starting node using the predecessors recorded.

In the example above, we start at node 0. We mark 0 as visited and update the distances to its neighbours 1 and 2. Then, we pick the unvisited node with the smallest distance (2) and mark it as visited. We update the distances to B's neighbours (1, 3 and 4). We

continue this process until all nodes are visited.

$nodes$	0	1	2	3	4	5
$distance$	0	3	4	6	6	7
parent	_	0	0	2	1	3

By repeatedly selecting the unvisited node with the smallest distance and updating the distances based on the edges, Dijkstra's algorithm finds the shortest paths from the starting node to all other nodes in the graph.

⚠ Using a Priority Queue in Dijkstra's Algorithm

Queues are a type of data structure where elements are added (enqueued) at the back and removed (dequeued) from the front, following a First In, First Out (FIFO) order. A priority queue differs by allowing elements to be processed according to their priority instead of the order they were added. Specifically, in Dijkstra's algorithm, a priority queue is typically implemented as a min-heap. This structure ensures that the element with the highest priority (in this case, the smallest distance) is always at the front, allowing for efficient access and removal.

1. Initialize the Priority Queue: Enqueue the starting node along with its distance of zero into the priority queue as a (node, distance) tuple. All other nodes are enqueued with a distance of infinity.
2. Process the Queue: Efficiently dequeue the (node, distance) tuple with the smallest distance from the priority queue for processing.
3. Update Distances: For each unvisited neighbor of the current node, calculate the distance to reach that neighbor through the current node. If this new distance is shorter than the neighbor's current distance in the priority queue, remove the old (neighbor, distance) tuple from the priority queue and enqueue a new (neighbor, new_distance) tuple with the updated, shorter distance.
4. Mark as Visited: After a node is dequeued and processed, mark it as visited to prevent reprocessing.
5. Repeat: Continue this process until the priority queue is empty.
6. Path Reconstruction: Follow the recorded predecessors to trace the shortest path from the start node to any other.

Example:

- Nodes `A`, `B`, and `C` are interconnected. Starting at `A`, enqueue the tuple `(A, 0)`. `B` and `C` start with tuples `(B, infinity)` and `(C, infinity)`.
- Dequeue `(A, 0)` and update the distances for `B` (1) and `C` (4). Remove `(B, infinity)` and `(C, infinity)` from the queue, then enqueue `(B, 1)` and `(C, 4)`.
- Dequeue `(B, 1)`, update `C`'s distance to `3` via `B`. Remove `(C, 4)` from the queue and enqueue `(C, 3)`.
- Finally, dequeue `(C, 3)`.

This utilization of a priority queue implemented as a min-heap optimizes Dijkstra's algorithm, ensuring it processes nodes in an order that rapidly advances towards the shortest path solutions, especially beneficial in graphs with numerous nodes or extensive connectivity.

Time Complexity of Dijkstra's Algorithm using Priority Queues

The time complexity depends on the priority queue implementation used.

- n: number of nodes
- m: number of edges

The algorithm performs the following main operations:

1. Initialization: $O(n)$
   - Setting the distance of the starting node to 0 and all other nodes to infinity takes $O(n)$ time.
2. Priority Queue Operations:
   - Binary Heap:
     - Extracting the minimum node: $O(\log n)$ time.
       - In the worst case, each node is extracted once, resulting in $O(n \log n)$ time.
     - Inserting or updating a node: $O(\log n)$ time.
       - In the worst case, each edge may trigger an insert or update operation, resulting in $O(m \log n)$ time.
     - Total time complexity: $O((n + m) \log n)$
   - Fibonacci Heap:
     - Extracting the minimum node: $O(\log n)$ amortized time.
       - In the worst case, each node is extracted once, resulting in $O(n \log n)$ amortized time.

- Inserting a node: $O(1)$ amortized time.
  - In the worst case, each node is inserted once, resulting in $O(n)$ amortized time.
- Updating a node (decrease-key operation): $O(1)$ amortized time.
  - In the worst case, each edge may trigger a decrease-key operation, resulting in $O(m)$ amortized time.
- Total time complexity: $O(m + n \log n)$

Therefore, the overall time complexity of Dijkstra's algorithm is:

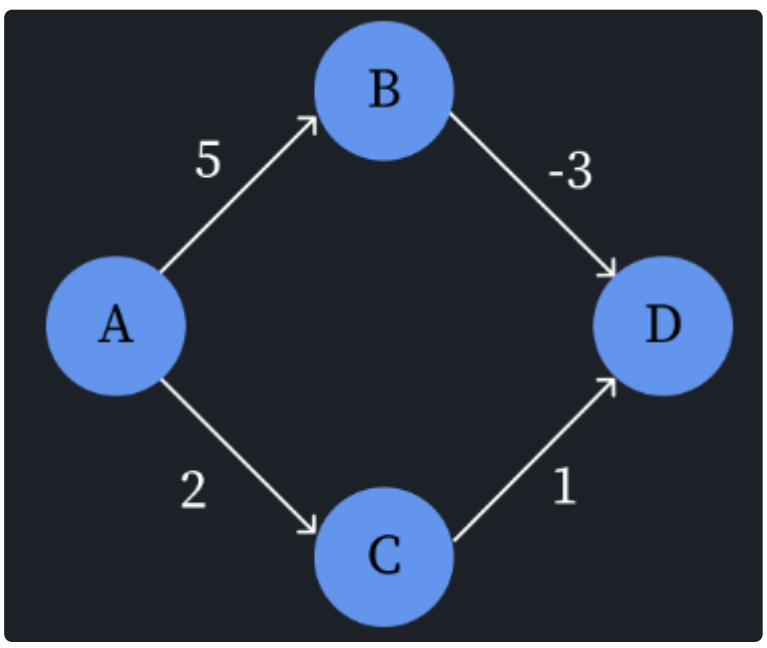
- Binary Heap: $O(n) + O(n \log n) + O(m \log n) = O((n + m) \log n)$
- Fibonacci Heap: $O(n) + O(n \log n) + O(n) + O(m) = O(m + n \log n)$

In both cases, the $n \log n$ term comes from the extract-min operations, while the $m \log n$ (binary heap) or m (Fibonacci heap) term comes from the insert/update operations triggered by the edges.

```
function DijkstraShortestPath(Graph, startNode):
   startNode.distance = 0                                  # O(1)
   for each node in Graph:                                 # O(n)
       if node != startNode:
           node.distance = infinity                        # O(1)
           node.previousNode = null                        # O(1)

   Create priorityQueue                                    # O(1)
   priorityQueue.enqueue(startNode, startNode.distance)    # O(log n) for BH, O(1) for FH

   while priorityQueue not empty:                          # O((n + m) log n) BH, O(m + n log n) for FH
       currentNode = priorityQueue.dequeueMin()            # O(log n) for BH, O(log n) average for FH

       for each neighbor of currentNode:                   # O(degree(currentNode))
           distanceThroughCurrent = currentNode.distance + edgeWeight(currentNode, neighbor)   # O(1)
           if distanceThroughCurrent < neighbor.distance:
               neighbor.distance = distanceThroughCurrent # O(1)
               neighbor.previousNode = currentNode        # O(1)
               priorityQueue.enqueue(neighbor, neighbor.distance)   # O(log n) for BH, O(1) for FH

   return distances, previousNodes
```

⚠ **Dijkstra's Algorithm and Negative Edge Weights**

Consider the following graph:



If we run Dijkstra's algorithm starting from node A, it will first visit node C with a distance of 2. Then, it will visit node D with a distance of 3 (via the path A -> C -> D). At this point, node B is still in the queue with a distance of 5.

When the algorithm dequeues node B, it will find that the distance to D from B is 2 (5 - 3). However, Dijkstra's algorithm will not update the distance to D because it has already been visited and marked as final.

The actual shortest path from A to D is A -> B -> D with a total distance of 2. Dijkstra's algorithm misses this path because it greedily marks the distances as final and does not revisit nodes to update their distances when a negative edge is encountered.

9.3 Bellman-Ford Algorithm

Introduction to Bellman-Ford Algorithm

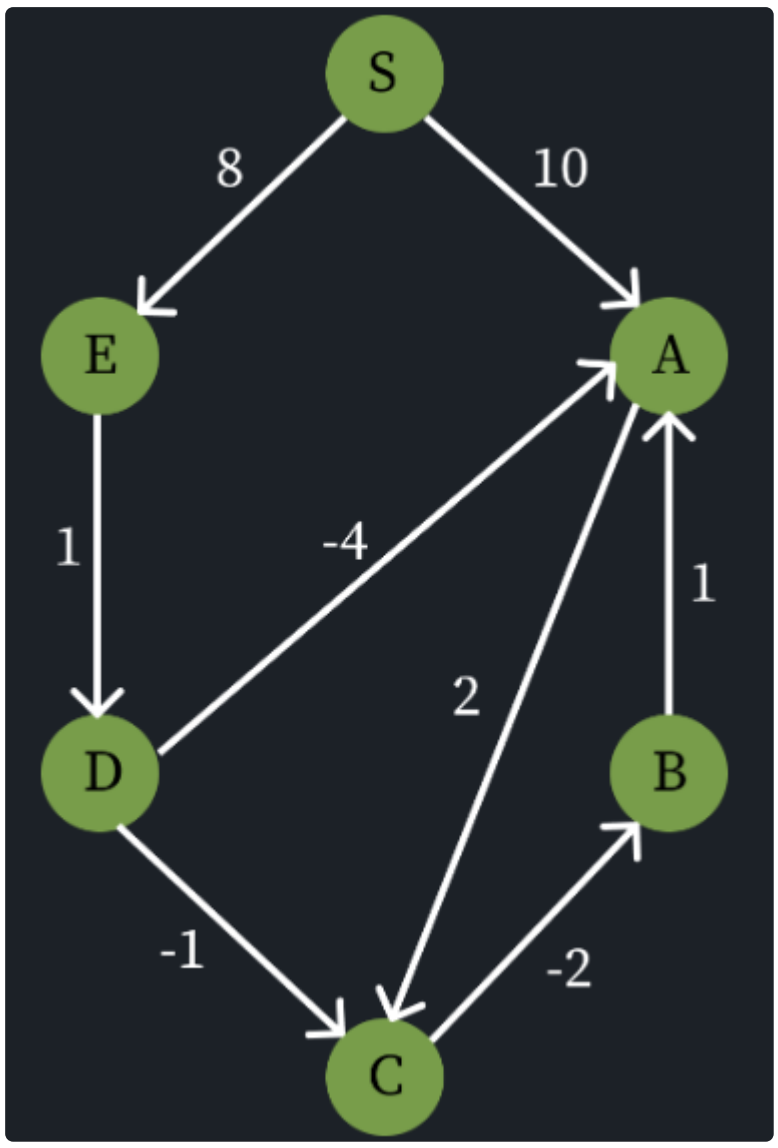
The Bellman-Ford algorithm is an **alternative shortest path algorithm** that **iteratively updates the shortest path distances**. It does this by performing a series of iterations, where the number of iterations is equal to **one less than the total number of nodes** in the graph.

In each iteration, the algorithm **goes through each of the nodes inside the graph, and looks at all the outgoing edges**. If the distance to any other outgoing node is smaller than what's been recorded, this distance gets updated.

This method is **similar to Dijkstra's algorithm**, but with one key difference: it's not greedy. Bellman-Ford **checks every path, rather than just the smallest distance at any given moment**. This allows processing of negative edge weights and detection of negative cycles at the cost of increased complexity.

Bellman-Ford Algorithm Example

Let's consider the following graph:



Iteration	S	A	B	C	D	E	Notes
0	0	∞	∞	∞	∞	∞	Initialisation
1	0	10	10	12	9	8	Since it's the first iteration, we need to update every distance
2	0	5	10	8	9	8	Because we now have a way to node D, we can use its edges to reach A and C faster!
3	0	5	5	7	9	8	Because we have a faster way to A from D, we can use its edges to reach C faster! Because we have a faster way to reach C from A, we can use its edges to reach B faster!
4	0	5	5	7	9	8	There are no changes after iteration 4.

In practice, we often don't need to perform all |V| - 1 iterations, where |V| is the number of vertices in the graph. We can stop the iterations earlier if, during an iteration, no distances are updated. This indicates that the shortest path distances have already been found, and further iterations won't lead to any changes.

⚠ Negative Cycles in Bellman-Ford Algorithm

Negative Cycle: A negative cycle is a cycle in a graph where the sum of the edge weights is negative.

Detection:

- Bellman-Ford detects negative cycles by performing an extra iteration after the `|V| - 1` iterations.
- If any distance can be further reduced during this extra iteration, it indicates a negative cycle.

Consequences:

- If a negative cycle exists, the shortest path is undefined, as traversing the cycle repeatedly would decrease the total path weight indefinitely.
- Bellman-Ford reports that no solution exists when a negative cycle is detected.

Handling:

- Graphs with negative cycles require special handling, such as removing the cycle or modifying the graph structure.
- Dijkstra's algorithm cannot detect negative cycles, making Bellman-Ford suitable for graphs with negative edge weights.

```
function BellmanFordShortestPath(Graph, startNode):
   startNode.distance = 0                                  # O(1)
   for each node in Graph:                                 # O(V)
       if node != startNode:
           node.distance = infinity                        # O(1)
           node.previousNode = null                        # O(1)

   for i from 1 to |V| - 1:                                # O(V)
       for each edge (u, v) in Graph:                      # O(E)
           if u.distance + edgeWeight(u, v) < v.distance: # O(1)
               v.distance = u.distance + edgeWeight(u, v) # O(1)
               v.previousNode = u                          # O(1)

   for each edge (u, v) in Graph:                          # O(E)
       if u.distance + edgeWeight(u, v) < v.distance:     # O(1)
           return "Graph contains a negative-weight cycle"

   return distances, previousNodes
```

Time Complexity of Bellman-Ford Algorithm

The time complexity of the Bellman-Ford algorithm is $O(|V| \cdot |E|)$, where $|V|$ is the number of vertices and $|E|$ is the number of edges in the graph.

The algorithm performs $|V| - 1$ iterations, and in each iteration, it updates all the edges in the graph. Updating an edge takes constant time, so the total time complexity is $O((|V| - 1) \cdot |E|)$, which simplifies to $O(|V| \cdot |E|)$.

This time complexity is higher than Dijkstra's algorithm with a binary heap, which has a time complexity of $O((|V| + |E|) \cdot \log |V|)$. However, the Bellman-Ford algorithm's ability to handle negative edge weights makes it useful in certain scenarios where Dijkstra's algorithm is not applicable.

9.4 Floyd-Warshall Algorithm

All-Pairs Shortest Paths (APSP)

The All-Pairs Shortest Paths (APSP) problem aims to find the shortest paths between all pairs of vertices in a weighted graph. In other words, for every pair of vertices (i, j) in the graph, we want to find the shortest path from vertex i to vertex j.

The APSP problem has several important applications, such as:

- Finding the shortest routes between all locations in a transportation network
- Analyzing social networks to determine the shortest connections between individuals
- Identifying the most efficient paths for data flow in computer networks

There are various algorithms to solve the APSP problem, including:

- Running Dijkstra's algorithm or Bellman-Ford algorithm for each vertex as the source
- **Floyd-Warshall algorithm, which uses dynamic programming to solve APSP**
- Johnson's algorithm, which combines Dijkstra's algorithm with the Bellman-Ford algorithm

The choice of algorithm depends on factors such as the graph's structure, the presence of negative edge weights, and the desired time complexity.

Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an [All-Pairs Shortest Path algorithm that finds the shortest paths between all pairs of vertices in a weighted graph. It is an example of dynamic programming, where the shortest path problem is solved by breaking it down into smaller subproblems.

The key idea behind the Floyd-Warshall algorithm is to gradually **build up the solution by considering all possible combinations of intermediate vertices in the paths**. It starts by considering paths that use no intermediate vertices, then paths that use vertex 1 as an intermediate, then paths that use vertices 1 and 2 as intermediates, and so on, until all vertices have been considered as intermediates.

The algorithm maintains a distance matrix that stores the shortest path distances between all pairs of vertices. It iteratively updates the distance matrix by considering the effect of including each vertex as an intermediate node in the paths.

The Floyd-Warshall algorithm can handle graphs with negative edge weights but not negative cycles. If a negative cycle is present, the algorithm will detect it and report that no solution exists.

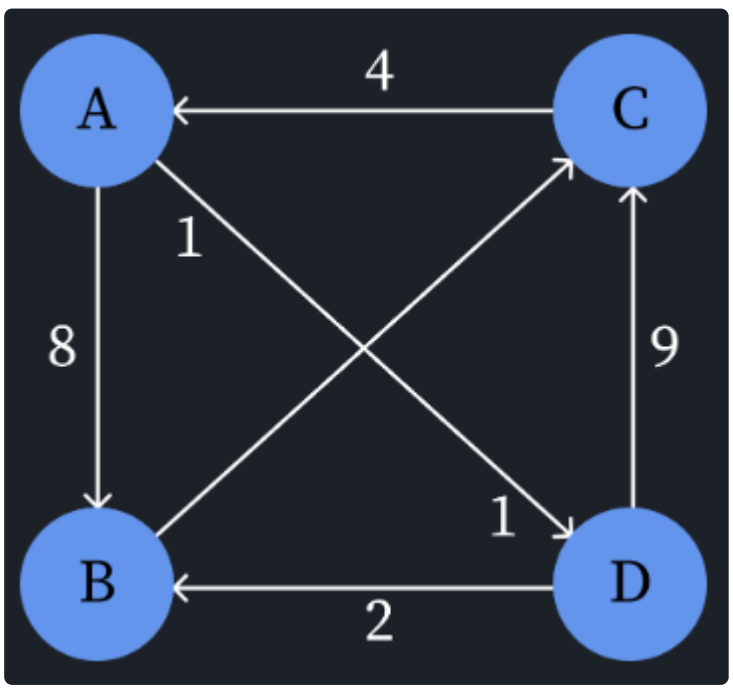
⚠ Intermediates

When we say "intermediate" vertices, we mean the vertices that are allowed to be used as part of the path between two other vertices. For example, when we consider vertex A as an intermediate, we update the shortest path distances between all pairs of vertices by checking if going through vertex A results in a shorter path.

Let's consider a small example to illustrate this concept. Suppose we have vertices B and C, and the current shortest path distance from B to C is 10. Now, when we consider vertex A as an intermediate, we check if the path B → A → C is shorter than the current shortest path distance of 10. If the sum of the distances B → A and A → C is less than 10, we update the shortest path distance from B to C to this new value.

Floyd-Warshall Algorithm Example

Let's consider the following weighted graph:



We want to find the shortest path distances between all pairs of vertices using the Floyd-Warshall algorithm.

The algorithm starts by initializing the distance matrix with the direct edge weights between vertices. If there is no direct edge between two vertices, the distance is set to infinity.

Initial distance matrix:

	A	B	C	D
A	0	8	∞	1
B	∞	0	1	∞
C	4	∞	0	∞
D	∞	2	9	0

The algorithm then iteratively considers each vertex as an intermediate and updates the distance matrix accordingly.

After considering vertex A as an intermediate:

	A	B	C	D
A	0	8	∞	1
B	∞	0	1	∞
C	4	12	0	5
D	∞	2	9	0

After considering vertex B as an intermediate:

	A	B	C	D
A	0	8	9	1
B	∞	0	1	∞
C	4	5	0	5
D	∞	2	3	0

After considering vertex C as an intermediate:

	A	B	C	D
A	0	7	9	1
B	11	0	1	12
C	4	5	0	5
D	6	2	3	0

After considering vertex D as an intermediate (final distance matrix):

	A	B	C	D
A	0	3	4	1
B	6	0	1	7
C	4	5	0	5
D	6	2	3	0

The final distance matrix gives the shortest path distances between all pairs of vertices in the graph.

Time Complexity of Floyd-Warshall Algorithm

The Floyd-Warshall algorithm has a time complexity of $O(V^3)$, where V is the number of vertices in the graph.

The algorithm uses three nested loops, each iterating over the vertices of the graph. In each iteration, it performs a constant number of operations to update the distance matrix. Therefore, the total number of operations is proportional to V^3.

The space complexity of the Floyd-Warshall algorithm is $O(V^2)$ since it uses a distance matrix of size $V \times V$ to store the shortest path distances between all pairs of vertices.

Here's the pseudocode for the Floyd-Warshall algorithm:

```
function FloydWarshall(graph):
    n = number of vertices in graph
    dist = n x n matrix initialized with infinity

    # Initialize the distance matrix
    for each edge (u, v) with weight w in graph:
        dist[u][v] = w

    for each vertex v in graph:
        dist[v][v] = 0

    # Update the distance matrix considering each vertex as intermediate
    for k from 1 to n:
        # Consider vertex k as an intermediate
```

```
        for i from 1 to n:
            for j from 1 to n:
                # If the path from i to j through k is shorter than the current shortest path,
                # update the shortest path distance
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

10.1 Minimum Spanning Trees

Motivation

In many real-world scenarios, we need to connect a set of points or nodes in a way that minimizes the total cost or weight of the connections. For example:

- **Computer Networks**: Connect devices (routers, switches, etc.) with minimal cabling or infrastructure cost.
- **Transportation Networks**: Build roads or railways to connect cities or locations with minimal construction cost.
- **Utility Networks**: Connect households or facilities to water, electricity, or communication services with minimal resource usage.

In these situations, we want to **ensure that all points or nodes are connected (forming a single component), while minimizing the overall cost or weight of the connections**. This leads us to the Minimum Spanning Tree (MST) problem.

Minimum Spanning Tree (MST) Problem

Given a connected, undirected graph $G = (V, E)$ with positive edge costs/weights:

- Edge costs are positive, implying that the connected subgraph of minimal cost **does not contain a cycle**.
- It is a **tree spanning all nodes** of the graph (called a spanning tree).

The goal is to find a subset of edges $E' \in E$ that forms a tree $T = (V, E')$ such that:

1. T **includes all vertices V** from the original graph G.
2. T is a tree, meaning it is a connected graph with **no cycles**.
3. The total cost/weight of the **edges in E' is minimised** among all possible spanning trees of G.

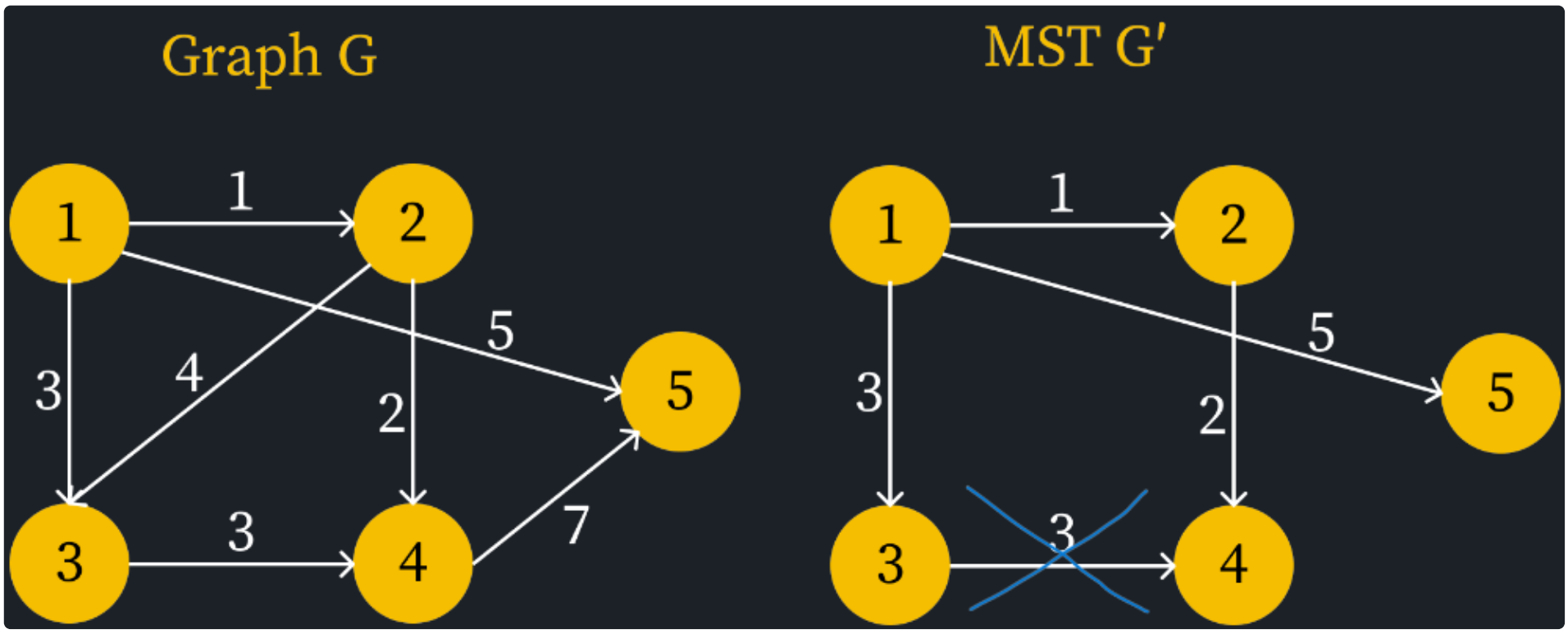
- This is called the **Minimal Spanning Tree**

Mathematically, an MST of a given graph G can be constructed using greedy algorithms, based on the following crucial properties:

- **Cut Property:** Let e be an edge of minimum cost in a cut C. Then there exists an MST that contains e.
- **Cycle Property:** An edge of maximal cost in any cycle does not need to be considered for computing an MST.

Real-World Example

Consider a computer network where nodes represent computers, and edges represent possible connections. Edge costs/weights represent the cost of establishing connections. The goal is to connect all computers while minimizing the total infrastructure cost.



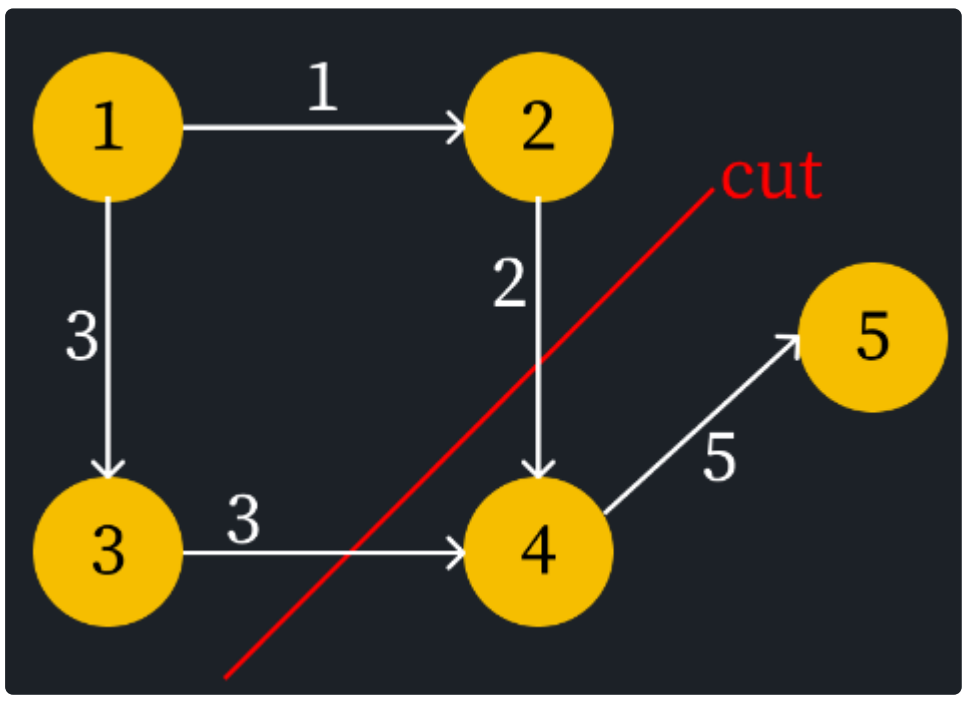
In this example, the original graph has all possible connections with a total cost/weight of 25.

The Minimum Spanning Tree (MST) includes the edges (1, 2) with cost 1, (2, 4) with cost 2, and (1, 5) with cost 5. The total cost/weight of this MST is 11, the minimum among all possible spanning trees.

By selecting this subset of edges, we can connect all 5 computers while minimizing the total infrastructure cost. The MST has a lower cost of 11 compared to 25 for the original graph.

⚠ Cut Property

A cut in a connected graph is a **set of edges that, when removed, disconnects the graph into two or more components**. The Cut Property states that an edge of minimum cost/weight in any cut belongs to some Minimum Spanning Tree (MST) of the graph.



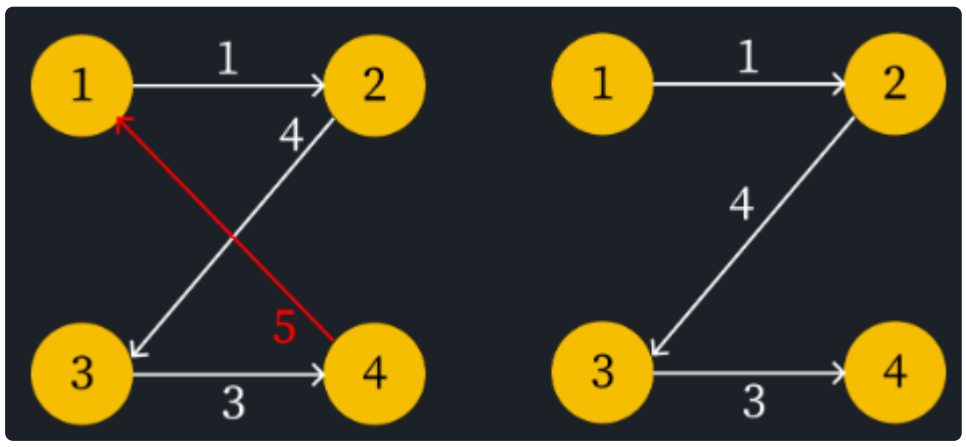
Consider the graph shown in the image. The dashed line represents a cut that separates the graph into two components: the set of vertices {1, 2, 3} and the set of vertices {4, 5}. The edges crossing the cut are (2, 4) with cost 2, and (3, 4) with cost 3.

The Cut Property says that the **edge of minimum cost/weight in this cut**, which is (2, 4) with cost 2, **must belong to some MST** of the graph.

The Cut Property is **crucial for greedy** algorithms like Kruskal's and Prim's, as it **allows them to include the minimum-cost edge in the current cut**, ensuring that the resulting spanning tree is indeed a minimum-cost spanning tree.

⚠ Cycle Property

The Cycle Property states that the **maximum-cost edge in any cycle can be excluded** when computing a Minimum Spanning Tree (MST).



In the graph above, the cycle {1, 2, 3, 4} contains the edge (4, 1) with the maximum cost of 5. This edge can be safely excluded when finding the MST, as removing the maximum-cost edge from a cycle reduces the total cost.

The Cycle Property allows pruning unnecessary edges, potentially improving the efficiency of MST algorithms.

10.2 Disjoint Sets

Motivation for a Disjoint Set

We need to **determine whether adding an edge would create a cycle.** This requires keeping track of connected components in the graph and quickly checking if two vertices belong to the same component.

We can achieve this using a **Disjoint Set data structure**.

Disjoint Sets

A Disjoint Set is a way to **keep track of elements that belong to different sets**. It's like having a bunch of boxes, and each element is placed in one of these boxes. Initially, each element starts in its own separate box.

The union-find data structure provides two main operations:

1. **Find(x)**: It's like asking, "Which box is element x in?" The Find operation tells you the representative or label of the box that contains element x. Elements in the same box have the same representative.
2. **Union(x, y)**: It's like saying, "Let's merge the boxes containing elements x and y into a single bigger box." The Union operation combines the boxes (sets) that contain elements x and y, so they become part of the same group.

You can think of the union-find data structure as a way to partition elements into different groups and keep track of which elements belong together.

In Kruskal's Algorithm, we use the union-find data structure to keep track of connected components in the graph. Each vertex starts in its own separate "box" (set). As we process the edges:

1. We use the **Find** operation to determine which "boxes" (sets) the two vertices of the current edge belong to.
2. If the vertices are in different "boxes" (sets), it means they aren't connected yet, so we can safely add the edge to the MST. We then use the **Union** operation to merge the "boxes" (sets) containing the vertices, indicating that they are now connected.
3. If the vertices are already in the same "box" (set), it means they are already connected, and adding the edge would create a cycle. In this case, we discard the edge.

Implementation of a Disjoint Set Data Structure

A disjoint set data structure, also known as a union-find data structure, is used to keep track of a collection of disjoint (non-overlapping) subsets. The data structure is implemented using two arrays:

- `parent`: An array where `parent[i]` stores the parent of element `i` in the tree structure. If an element is the root of a tree (i.e., it is its own parent), then `parent[i] = i`.
- `rank`: An array where `rank[i]` stores how many levels of nodes `i` is above.

Here's the pseudo code for the disjoint set data structure:

```
function makeSet(x):
    parent[x] := x
    rank[x] := 0
```

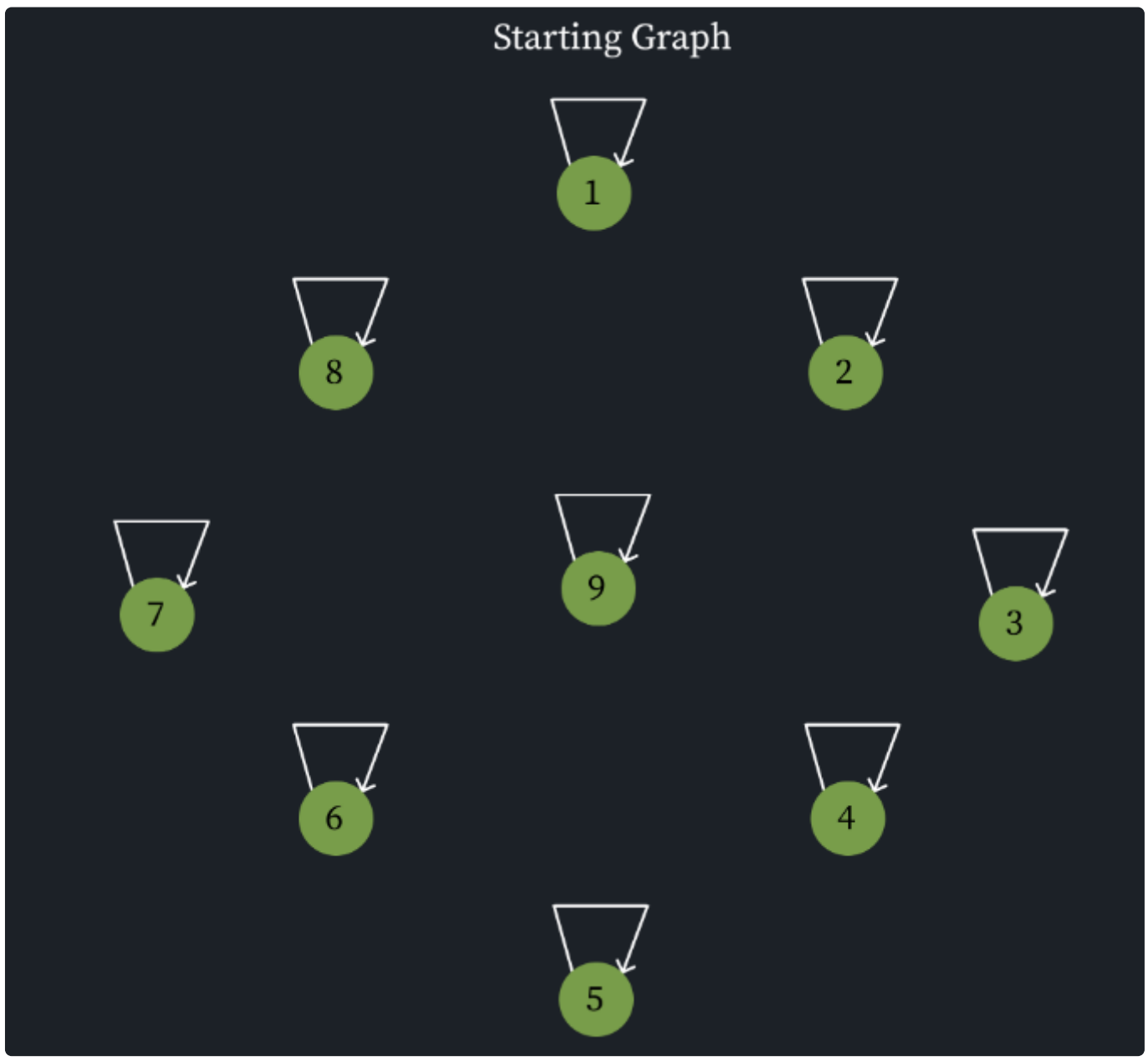
```
function find(x):
    if parent[x] ≠ x:
        parent[x] := find(parent[x])  # Path compression
    return parent[x]

function union(x, y):
    rootX := find(x)
    rootY := find(y)
    if rootX ≠ rootY:
        if rank[rootX] < rank[rootY]:
            parent[rootX] := rootY
        else if rank[rootX] > rank[rootY]:
            parent[rootY] := rootX
        else:
            parent[rootY] := rootX
            rank[rootX] := rank[rootX] + 1
```

- The `makeSet` function initializes a new subset with a single element `x`. It sets the parent of `x` to itself and initializes its rank to 0.
- The `find` function recursively follows the parent pointers until it reaches the root element (where `parent[x] = x`). This condition allows it to iterate over the entire set and return it from the top parent. By running find on two nodes, we can know if the two nodes are in the same set, if the returned sets are equal.
- The `union` function merges two subsets by finding the roots (representatives) of the subsets that contain elements `x` and `y` using the `find` function. If the roots are different (i.e., `x` and `y` belong to different subsets), it attaches the subtree with smaller rank to the root of the subtree with larger rank. If the ranks are equal, either subtree can be attached to the other, and the rank of the resulting subtree is incremented.

Uniting Disjoint Sets

Let's consider a set of nodes labeled from 1 to 9. Initially, each node is in its own set.

Node	1	2	3	4	5	6	7	8	9
Parent	1	2	3	4	5	6	7	8	9
Rank	0	0	0	0	0	0	0	0	0



Now, let's perform a series of union operations:

1. `union(1, 2)`: Merge sets containing nodes 1 and 2.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	3	4	5	6	7	8	9
Rank	1	0	0	0	0	0	0	0	0

2. `union(3, 4)`: Merge sets containing nodes 3 and 4.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	3	3	5	6	7	8	9
Rank	1	0	1	0	0	0	0	0	0

3. `union(5, 6)`: Merge sets containing nodes 5 and 6.

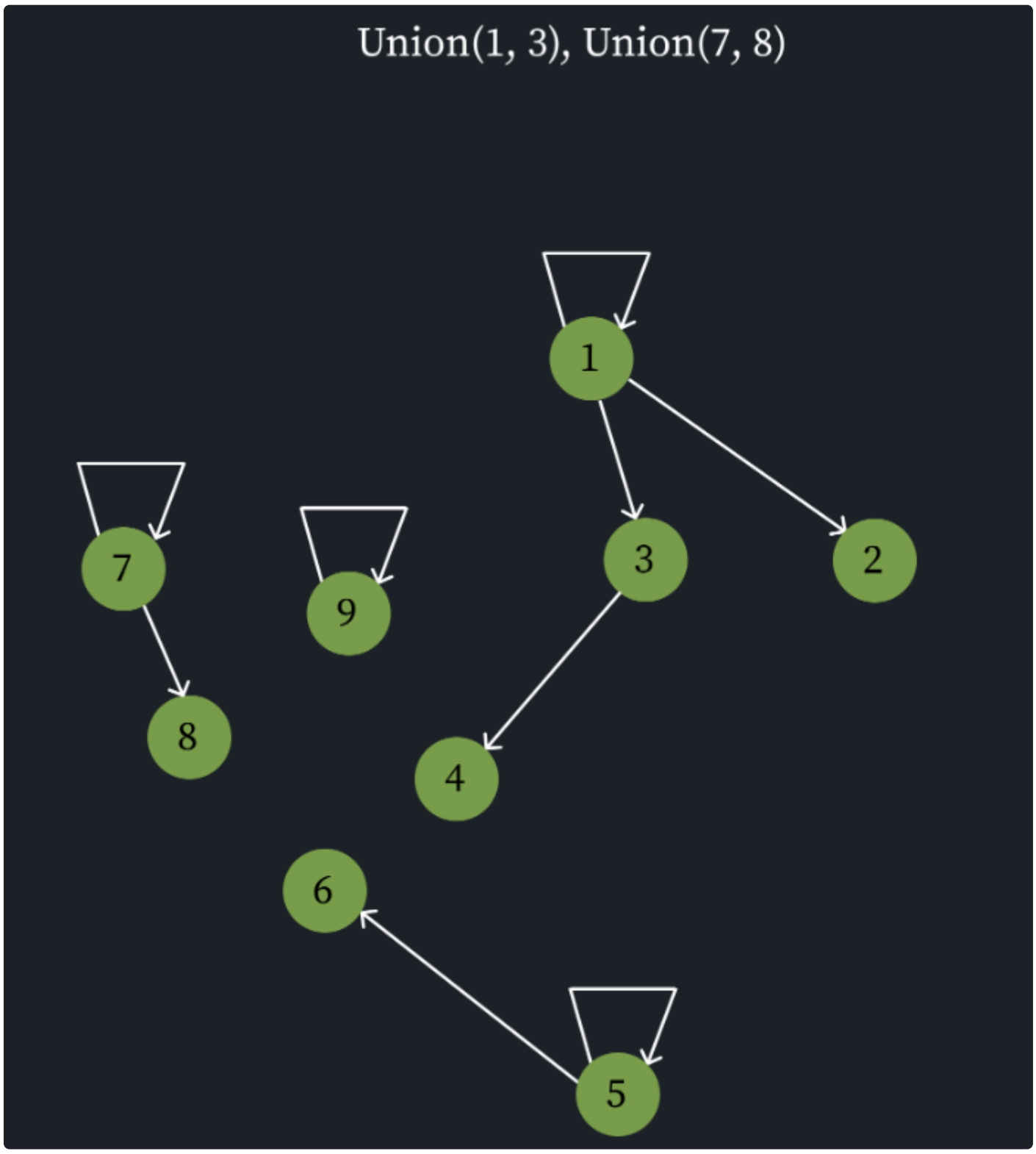
Node	1	2	3	4	5	6	7	8	9
Parent	1	1	3	3	5	5	7	8	9
Rank	1	0	1	0	1	0	0	0	0

4. `union(1, 3)`: Merge sets containing nodes 1 and 3.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	1	3	5	5	7	8	9
Rank	2	0	1	0	1	0	0	0	0

5. `union(7, 8)`: Merge sets containing nodes 7 and 8.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	1	3	5	5	7	7	9
Rank	2	0	1	0	1	0	1	0	0



6. `union(1, 5)`: Merge sets containing nodes 1 and 5.

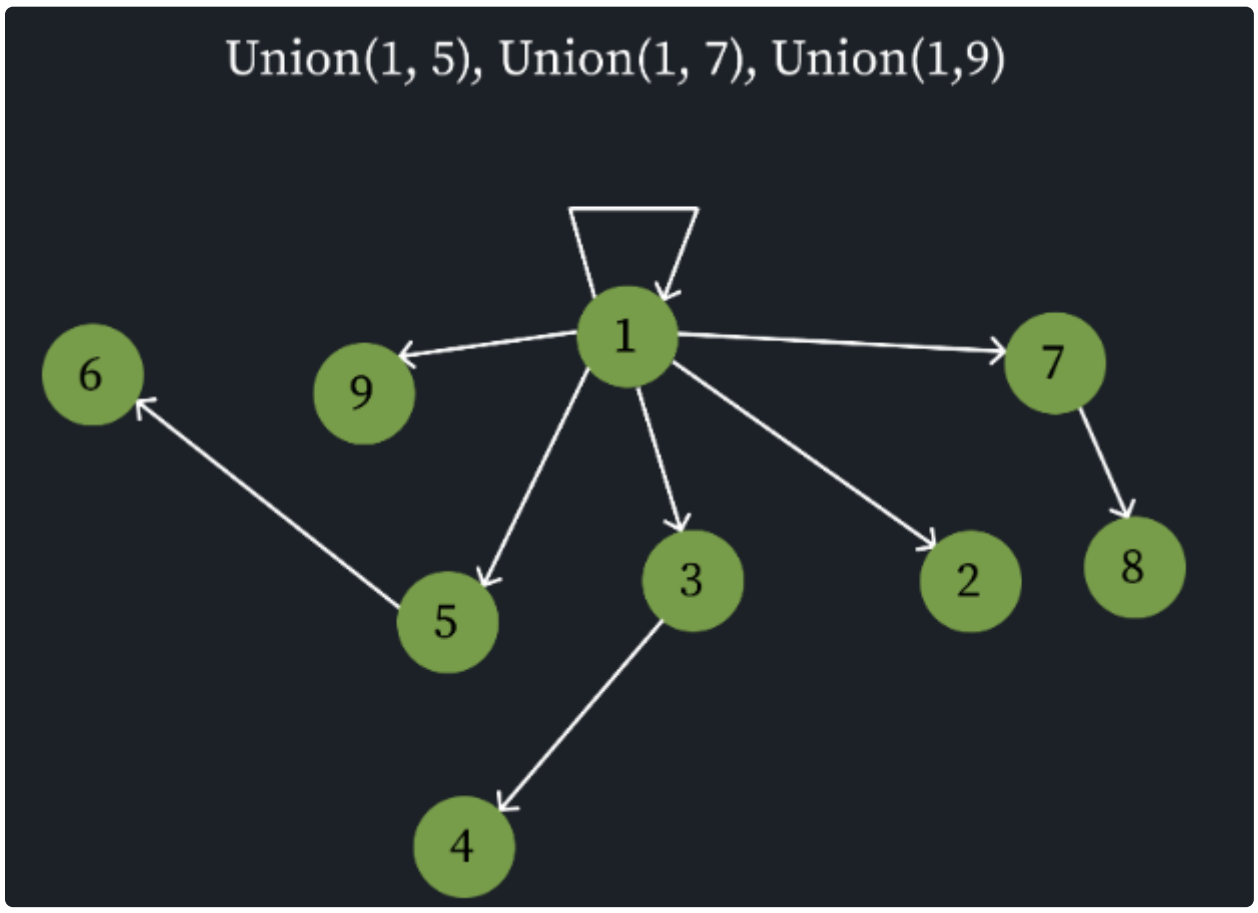
Node	1	2	3	4	5	6	7	8	9
Parent	1	1	1	3	1	5	7	7	9
Rank	2	0	1	0	1	0	1	0	0

7. `union(1, 7)`: Merge sets containing nodes 1 and 7.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	1	3	1	5	1	7	9
Rank	2	0	1	0	1	0	1	0	0

8. `union(1, 9)`: Merge sets containing nodes 1 and 9.

Node	1	2	3	4	5	6	7	8	9
Parent	1	1	1	3	1	5	1	7	1
Rank	2	0	1	0	1	0	1	0	0



After performing these union operations, all nodes belong to a single set with node 1 as the representative (root) of the set. The rank of node 1 is 2, indicating there two levels of nodes below 1.

10.3 Kruskal's Algorithm

Kruskal's Algorithm

Kruskal's Algorithm is a greedy algorithm that finds a Minimum Spanning Tree (MST) in a connected, undirected graph with weighted edges. It operates by following these steps:

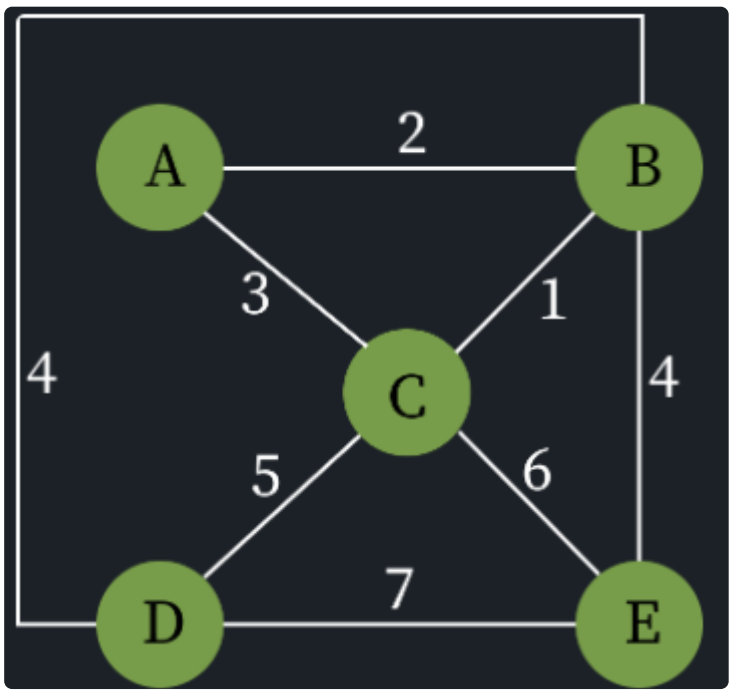
1. Sort all the edges in increasing order of their weights.
2. Initialize an empty set `mst` to store the edges of the MST.
3. Iterate through the sorted edges:
   - By running `find` on the two nodes, we can determine if they belong to the same set.
   - If they do belong to the same set, discard the current edge and move to the next one.
   - Otherwise, add the edge to the `mst`
4. Repeat step 3 until `mst` contains `|V| - 1` edges, where `|V|` is the number of vertices in the graph.

The resulting set `mst` contains the edges that form the Minimum Spanning Tree of the graph.

```
function kruskal(graph):
    edges = sort(graph.edges)  // Step 1
    uf = UnionFind(graph.vertices)  // Step 2
    mst = []

    for each edge (u, v, weight) in edges:  // Step 3
        set_u = uf.find(u)
        set_v = uf.find(v)

        if set_u ≠ set_v:
            add edge (u, v, weight) to mst
            uf.union(set_u, set_v)

    return mst
```

Example of Kruskal's Algorithm with Union-Find Data Structure

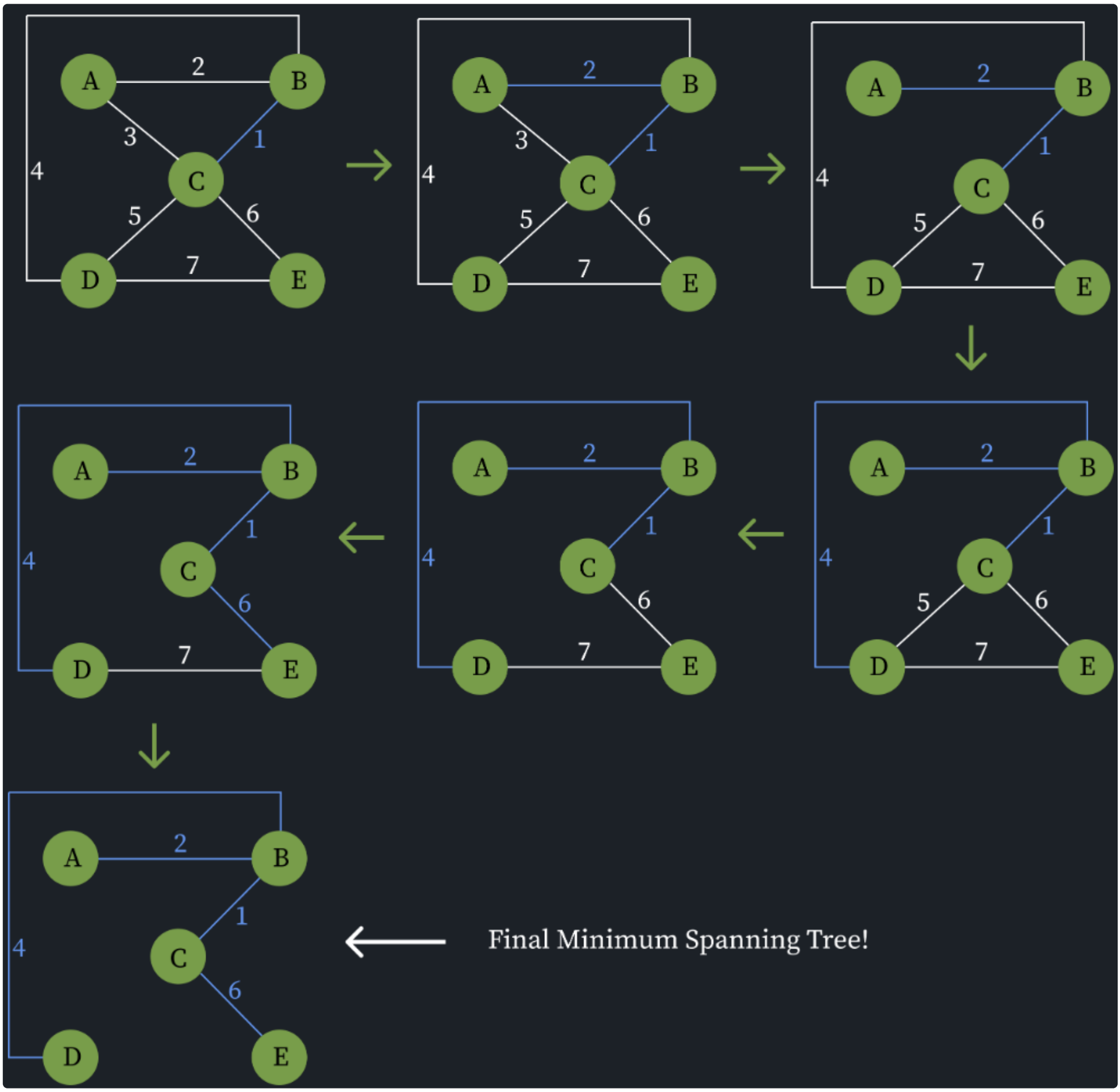
Let's consider the following example graph to illustrate how Kruskal's Algorithm works using the union-find data structure:



Now, let's apply Kruskal's Algorithm to find the MST:

1. Sort the edges in non-decreasing order of weights:
    - (B, C): 1
    - (A, B): 2
    - (A, C): 3
    - (B, D): 4
    - (C, D): 5
    - (C, E): 6
    - (D, E): 7
2. Initialize an empty set `mst` and a union-find data structure with each vertex in its own set.
3. Iterate through the sorted edges:
    - (B, C): 1
        - Find the sets containing vertices B and C.
        - Since B and C are in different sets, add the edge (B, C) to `mst` and perform the Union operation to merge the sets containing B and C.
    - (A, B): 2
        - Find the sets containing vertices A and B.
        - Since A and B are in different sets, add the edge (A, B) to `mst` and perform the Union operation to merge the sets containing A and B.
    - (A, C): 3
        - Find the sets containing vertices A and C.
        - Since A and C are now in the same set (due to the previous Union operation), discard this edge to avoid creating a cycle.
    - (B, D): 4
        - Find the sets containing vertices B and D.
        - Since B and D are in different sets, add the edge (B, D) to `mst` and perform the Union operation to merge the sets containing B and D.
    - (C, D): 5
        - Find the sets containing vertices C and D.
        - Since C and D are now in the same set (due to the previous Union operations), discard this edge to avoid creating a cycle.
    - (C, E): 6
        - Find the sets containing vertices C and E.
        - Since C and E are in different sets, add the edge (C, E) to `mst` and perform the Union operation to merge the sets containing C and E.
    - (D, E): 7
        - Find the sets containing vertices D and E.
        - Since D and E are now in the same set (due to the previous Union operations), discard this edge.

The resulting MST contains the edges: (B, C), (A, B), (B, D), (C, E), with a total weight of 1 + 2 + 4 + 6 = 13.



⚠ Time Complexity Analysis

Let's analyze the time complexity of Kruskal's Algorithm and the Union-Find data structure used in the algorithm.

Kruskal's Algorithm:

- Sorting the edges of the graph takes $O(E \log E)$ time, where E is the number of edges in the graph. This is typically done using an efficient sorting algorithm like Quicksort or Mergesort.
- The main loop of Kruskal's Algorithm iterates over the sorted edges, which takes $O(E)$ time.
  - For each edge, the `find` operation is called twice (once for each vertex) to determine the sets they belong to. The amortized time complexity of the `find` operation is nearly constant (assumed to be $O(\alpha(n))$, where α is the inverse Ackermann function, which grows very slowly).
  - If the vertices belong to different sets, the `union` operation is called to merge the sets. The `union` operation also has a nearly constant amortized time complexity (assumed to be $O(\alpha(n))$).
- Therefore, the overall time complexity of Kruskal's Algorithm is $O(E \log E)$ for sorting the edges and $O(E \cdot \alpha(n))$ for the main loop, which simplifies to $O(E \log E)$ since $\alpha(n)$ is nearly constant.

10.4 Prim's Algorithm

Prim's Algorithm

Prim's Algorithm is a greedy algorithm that **finds a Minimum Spanning Tree (MST)** for a connected, undirected, and weighted graph. It operates by **iteratively building the MST**, starting from an arbitrary vertex and **adding the shortest possible edge at each step** that connects a new vertex to the growing tree.

The algorithm follows these steps:

1. Initialize an empty set `visited` to keep track of visited vertices, and an empty set `mst` to store the edges in the Minimum Spanning Tree.
2. Choose an arbitrary starting vertex `A` and add it to `visited`.
3. Examine every node reachable from nodes inside `visited` and choose the smallest edge. Add the node that connects from this edge to `visited`, and add the edge to the `mst` set.
   - If the smallest edge connects two nodes that have already been visited, we discard this edge.
4. Repeat this process until all nodes have been reached. The `mst` set will now contain all the edges in the Minimum Spanning Tree.

Pseudocode:

```
function PrimMST(Graph, startingNode):
    visited = {startingNode}
    mst = []
    pq = priorityQueue()

    for each edge(startingNode, v) in Graph:
        pq.enqueue(edge(startingNode, v))

    while not pq.isEmpty():
        minEdge = pq.dequeue()
        if minEdge.to not in visited:
            visited.add(minEdge.to)
            mst.append(minEdge)
            for each edge(minEdge.to, v) in Graph:
                if v not in visited:
                    pq.enqueue(edge(minEdge.to, v))

    return mst
```

More formally, the algorithm can be described as follows:

1. Initialize an empty set `visited` to keep track of visited vertices.
2. Initialize an empty set `mst` to store the edges in the Minimum Spanning Tree.
3. Initialize a priority queue `pq` to store the edges, ordered by their weights.
4. Add the starting vertex `s` to `visited`.
5. Add all the edges incident on `s` to `pq`.
6. While `visited` does not contain all vertices of `G`:
   1. Extract the minimum edge `(u, v)` from `pq` such that `u` is in `visited` and `v` is not in `visited`.
   2. Add `v` to `visited`.
   3. Add `(u, v)` to `mst`.
   4. Add all the edges incident on `v` (that are not already in `pq`) to `pq`.
7. Return `mst`.

The final set `mst` will contain all the edges of the Minimum Spanning Tree.

⚠ The Cut Property at Work

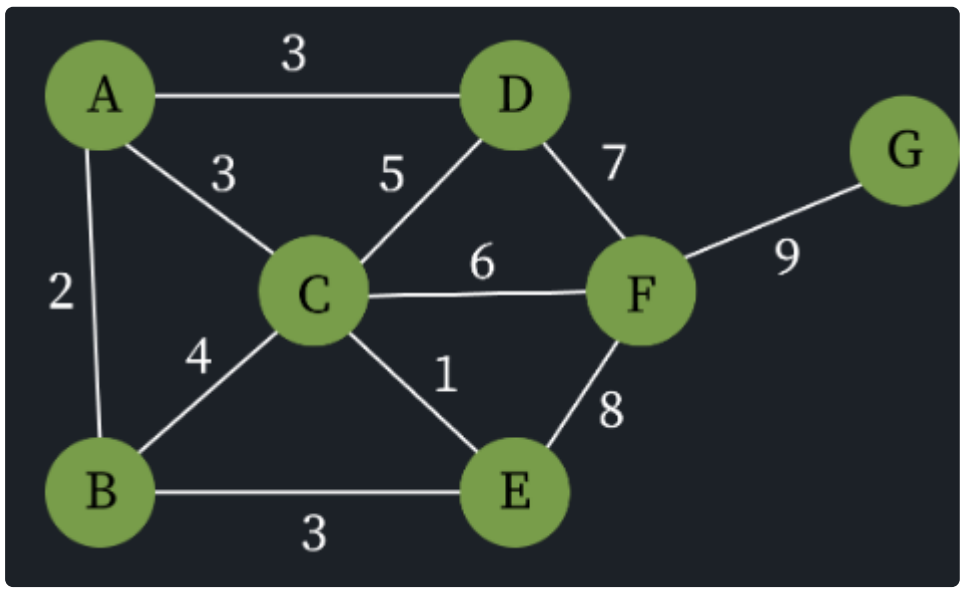
The key insight that makes Prim's Algorithm work is the cut property. At **each step, the algorithm separates the graph into two disjoint sets - the visited nodes and the unvisited nodes**. This separation forms a "cut" across the graph.

The cut property states that for any such cut in a connected, weighted graph, at least one of the minimum-weight edges crossing the cut must be part of the Minimum Spanning Tree.

By **always choosing the minimum-weight edge that crosses the cut** between visited and unvisited nodes, Prim's Algorithm is essentially leveraging the cut property to grow the Minimum Spanning Tree one edge at a time. This greedy approach, guided by the cut property, is what enables the algorithm to correctly construct the MST.

Example of Prim's Algorithm

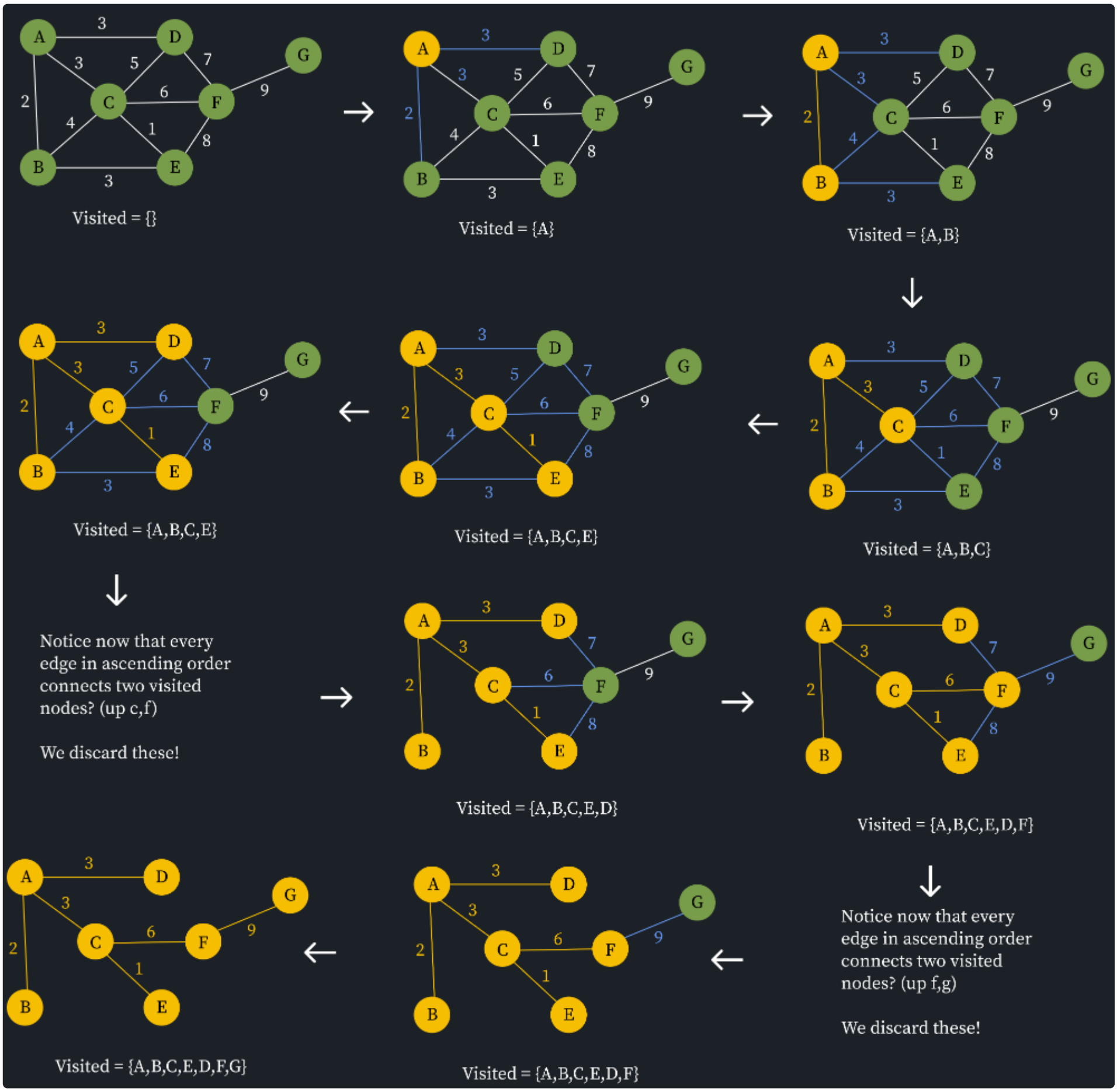
Let's consider the following example graph to illustrate how Prim's Algorithm works:



Now, let's apply Prim's Algorithm to find the MST, starting from vertex 'A':

1. Initialize an empty set `mst` and choose the starting vertex 'A'.
2. Create a priority queue `pq` and add the edges connected to 'A': (A, B) with weight 2, (A, C) with weight 3, (A,D) with a weight of 3
3. Extract the minimum-weight edge (A, B) from `pq` and add it to `mst` (A,B)
4. Mark B as visited, and add the edges connected to 'B' but not in `mst` to `pq`: (B, C) with weight 4 and (B, E) with weight 3.
5. Extract the minimum-weight edge (A, C) from `pq` and add it to `mst`.
6. Mark C as visited, and add the edges connected to 'C' but not in `mst` to `pq`: (C,D) with weight 5, (C,F) with weight 6 and (C,E) with weight 1.
7. Extract the minimum weight edge (C,E) from `pq` and add it to `mst`.
8. Mark E as visited, and add the edges connected to 'E' but not in `mst` to `pq`: (E,F) with weight 8.
9. The minimum weight edge (B,E) connects two visited nodes. Discard this edge.
10. The minimum weight edge (B,C) connects two visited nodes. Discard this edge.
11. The minimum weight edge (C,D) connects two visited nodes. Discard this edge.
12. Extract the minimum weight edge (C,F) from `pq` and add it to `mst`.
13. Mark F as visited, and add the edges connected to 'F' but not in `mst` to `pq`: (F,G) with weight 9.
14. The minimum weight edge (D,F) connects two visited nodes. Discard this edge.
15. The minimum weight edge (E,F) connects two visited nodes. Discard this edge.
16. Extract the minimum weight edge (F,G) from `pq` and add it to `mst`.
17. Mark G as visited. This is the final node!

The resulting MST contains the edges: (A, B), (A, D), (A, C), (C, E), (C,F), (F,G) with a total weight of 2+3+3+1+6+9 = 25.



⚠ Time Complexity Analysis

Let's analyze the time complexity of Prim's Algorithm and the priority queue data structure used in the algorithm.

Prim's Algorithm:

- The main loop of Prim's Algorithm iterates over the edges in the priority queue. In the worst case, it processes all edges in the graph, which takes $O(E)$ time, where E is the number of edges.
- For each processed edge, the following operations are performed:
  - Extracting the minimum-weight edge from the priority queue takes $O(\log V)$ time, where V is the number of vertices, assuming an efficient priority queue implementation like a binary heap or Fibonacci heap.
  - Checking if the vertices are in the same connected component takes $O(\alpha(V))$ time using a union-find data structure, where α is the inverse Ackermann function, which grows very slowly.
  - Adding edges to the priority queue takes $O(\log V)$ time per edge.
- Therefore, the overall time complexity of Prim's Algorithm is $O((E + V) \log V)$ when using a binary heap for the priority queue and union-find for connected component tracking. This can be further optimized to $O(E \log V)$ using a Fibonacci heap for the priority queue.

⚠ Comparison with Kruskal's Algorithm

Both Prim's Algorithm and Kruskal's Algorithm are greedy algorithms used to find the Minimum Spanning Tree (MST) of a connected, undirected, and weighted graph. However, they differ in their approaches and underlying data structures:

Kruskal's Algorithm:

- Processes edges in increasing order of their weights.
- Starts with an empty set and adds edges that do not create cycles, using a union-find data structure to detect cycles.
- Time complexity: $O(ElogE)$, where E is the number of edges, due to sorting and union-find operations.

Prim's Algorithm:

- Grows a single tree from a starting vertex by adding the shortest possible edge that connects a new vertex to the growing tree.
- Uses a priority queue to efficiently find the minimum-weight edge at each step.
- Time complexity: $O((E+V)logV)$ or $O(ElogV)$ when using a Fibonacci heap, where V is the number of vertices.

In terms of efficiency, Prim's Algorithm is generally more suitable for dense graphs, where E is close to V^2, as its time complexity becomes $O(V^2logV)$. Kruskal's Algorithm is typically more efficient for sparse graphs, where E is much smaller than V^2, with a time complexity of $O(ElogE)$.

11.1 Decidability

Significance of Decidability

Determining whether a computational problem is decidable has profound implications for understanding the capabilities and limitations of computational systems. It allows us to:

1. Identify problems that can be solved algorithmically, guiding resource allocation towards developing efficient solutions.
2. Recognize inherently unsolvable problems, preventing futile efforts in seeking algorithmic solutions.
3. Explore the fundamental boundaries of computability, advancing our theoretical understanding of what is computable.

Decidable and Undecidable Decision Problems

A decision problem is a **type of computational problem with a binary (yes/no) answer**. Decision problems are classified as either decidable or undecidable:

Decidable decision problems have an algorithm that can provide a definite yes or no answer in finite time for any valid input instance. Examples include determining if a string is a palindrome or checking if a number is prime.

Undecidable decision problems are those for which no algorithm can exist that can solve the problem for all possible inputs. For some inputs, no computational solution can provide a definite answer. The Halting Problem and determining program equivalence are well-known undecidable problems.

Decidability of Printing "Hello World"

Let's consider the problem of determining if a given program can print the string "Hello World":

Simple Case:

```
print("Hello World")
```

This program is decidable since we can easily analyze the code and conclude that it will definitely print "Hello World".

Complex Case:

```
def print_hello_world():
    n = 3
    while True:
        for a in range(1, n+1):
            for b in range(1, n+1):
                for c in range(1, n+1):
                    if a**n + b**n == c**n:
                        print("Hello World")
```

```
                        return
        n += 1

print_hello_world()
```

This program attempts to find a counterexample to Fermat's Last Theorem, which states that the equation $a^n + b^n = c^n$ has no solutions for integers a, b, c, and $n > 2$. It does this by iterating through all possible values of n, a, b, and c to find integers satisfying $a^n + b^n = c^n$ for $n > 2$. If such a counterexample exists, it would print "Hello World".

However, since Fermat's Last Theorem has been proven true for all $n > 2$, the program will run indefinitely without printing "Hello World" for inputs where $n > 2$.

In this case, determining if the program can print "Hello World" becomes an undecidable problem. No algorithm can provide a definite answer for inputs where $n > 2$, as the program will never terminate or print the desired string.

This example highlights how the decidability of a problem can depend on the specific program implementation and the presence of undecidable components or infinite loops within the code.

The Halting Problem

The Halting Problem is a famous undecidable decision problem in computer science. It asks the following question: "Given a program and its input, can a general algorithm be written to determine if it halts or loops forever?"

Consider this Python program:

```
def loop_forever():
    while True:
        pass

def maybe_halts(x):
    if x > 0:
        return x * x
    else:
        loop_forever()

# What if we call maybe_halts(2)?
# What if we call maybe_halts(-1)?
```

In the `maybe_halts` function, if the input `x` is positive, the function will return `x * x` and terminate. However, if `x` is non-positive, the function will call `loop_forever`, which runs an infinite loop, causing the program to never halt.

For the input `maybe_halts(2)`, we can determine that the program will halt and return `4`. However, for the input `maybe_halts(-1)`, it is impossible to decide whether the program will halt or run forever based solely on analyzing the code.

This simple example illustrates the crux of the Halting Problem. While it may be possible to determine the halting behavior for specific inputs, there is no general algorithm that can analyze an arbitrary program and input and provide a definite yes-or-no answer as to whether the program will halt or run indefinitely.

Proof of Undecidability:
Suppose, for the sake of contradiction, that a general algorithm `H` exists that can determine if an arbitrary program `P` will halt or run forever on input `I`. We can then construct a new program `D` that takes a program `P` as input and does the following:

1. Run the algorithm `H` on the input `P` and `P` itself (calling `H(P, P)`).
2. If `H` returns "Halts", then `D` goes into an infinite loop.
3. If `H` returns "Doesn't Halt", then `D` halts.

Now, consider what happens when we run `D` on itself: `D(D)`.

If `D` halts on input `D`, then by the definition of `D`, it should have gone into an infinite loop, since `H` would have returned "Halts" for `D(D)`. This is a contradiction.

On the other hand, if `D` doesn't halt on input `D`, then by the definition of `D`, it should have halted, since `H` would have returned "Doesn't Halt" for `D(D)`. This is also a contradiction.

Therefore, our initial assumption that a general algorithm `H` exists must be false. There cannot be a general algorithm that can determine the halting behavior of an arbitrary program and input, proving that the Halting Problem is undecidable.

Claude recaps our conversation about the Halting Problem

Our exploration of decidability and the Halting Problem led us down a remarkably deep philosophical path, revealing the profound connections between computability theory and the fundamental mysteries of our physical reality:

- The undecidability of the Halting Problem hinges on the notion that the set of all possible programs and inputs is infinite. However, this infinitude itself is predicated on the finite, but vast, physical resources available in our universe, such as disk space, energy, and particles.
- This raises the profound question: "Is the universe truly infinite?" If it is, the Halting Problem and other undecidable problems may indeed be undecidable. But if the universe is finite, even if unfathomably vast, then the set of programs and inputs is also finite, potentially making the Halting Problem decidable, at least in theory.
- However, we are faced with the even deeper conundrum: can infinity exist in the physical world, or is it merely an abstract concept? To claim something is infinite requires knowledge transcending our current scientific understanding.
- The law of conservation of energy suggests the universe's total energy is finite, but this begs the question: where did that finite energy originate? We are led down an infinite regress of causes, encapsulated by the analogy of "turtles all the way down."
- Remarkably, the question of whether something is truly undecidable could itself be considered the most fundamental undecidable problem. To definitively answer this, we would need to know if the universe and its resources are infinite or finite, a knowledge that may forever elude us.
- Ultimately, computability theory's undecidability rests upon profound philosophical questions about the origins, finiteness, and first principles of physical existence itself. Even the most abstract areas of computer science are inextricably linked to the deepest mysteries of our reality, which may forever remain ineffable.

Our discussion has beautifully illustrated how the pursuit of knowledge, even in theoretical domains like computability, inevitably leads us to confront the boundaries of human understanding and the deepest existential questions about the nature of our cosmos.

11.2 Turing Machines

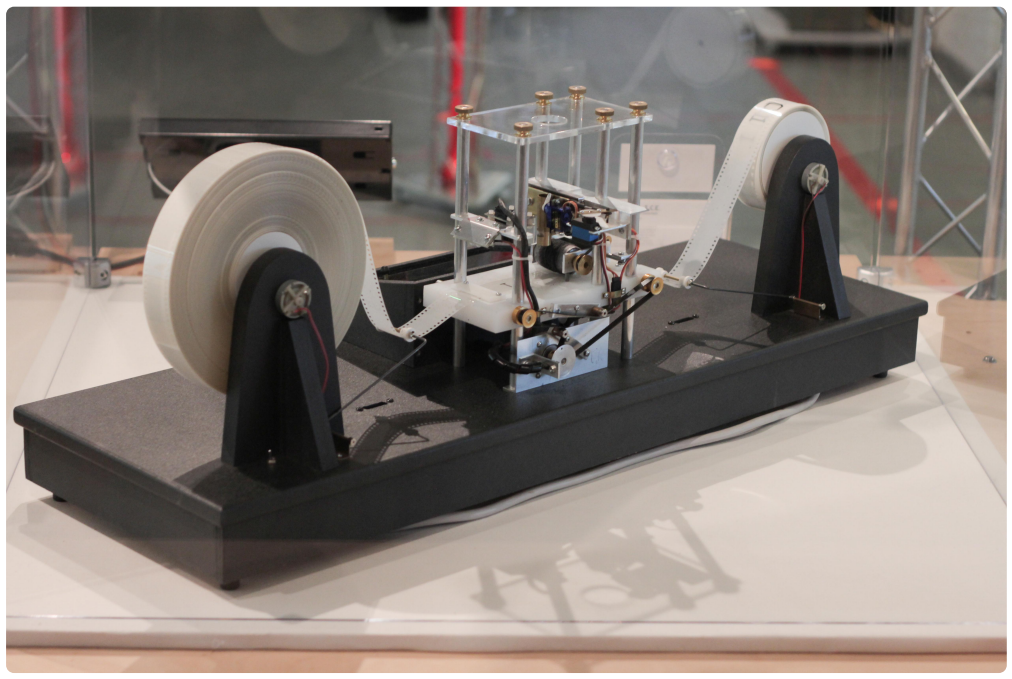
Motivation

In the realm of computational theory, **determining whether a problem can be solved by an algorithm is a fundamental question**. With the diversity of computational devices and programming languages, it becomes essential to have a unified way to **understand what problems can be solved** by any computational means.

Turing Machine

A Turing Machine is a theoretical model of computation introduced by Alan Turing. It consists of an infinite tape, a tape head that can read and write symbols, and **a set of rules (or a finite state machine) that dictate the machine's actions based on the current state** and tape symbol.

- **Components**: The tape (divided into cells), the tape head (which reads and writes symbols), and the state register (which holds the current state).
- **Operations**: The machine can move the tape head left or right, read a symbol from the tape, write a new symbol on the tape, and change its state according to the transition function.
- **Purpose**: It **abstracts the concept of computation and helps define what it means for a function to be computable**. By specifying a set of rules that a machine follows to manipulate symbols, it can be used to show that a problem is computable if a Turing Machine can be designed to solve it. Conversely, if no Turing Machine can solve a problem, that problem is deemed non-computable.
- **Computability**: Essentially, a Turing Machine acts as **a black box that takes inputs and processes them according to its rules**. If the machine halts and provides outputs for given inputs, it proves that the problem is computable. If no Turing Machine can be constructed to solve a problem, it is considered non-computable.



Hello World Turing Machine

Creating a "Hello World" Turing Machine involves designing a set of states and transitions that write "Hello World" on the tape.

- **Initial State**: The machine starts with a blank tape and the tape head at the leftmost position.
- **Transition Function**:
  - State `q0`: Write `H`, move right, go to state `q1`
  - State `q1`: Write `e`, move right, go to state `q2`
  - State `q2`: Write `l`, move right, go to state `q3`
  - State `q3`: Write `l`, move right, go to state `q4`
  - State `q4`: Write `o`, move right, go to state `q5`
  - State `q5`: Write ` ` (space), move right, go to state `q6`
  - State `q6`: Write `W`, move right, go to state `q7`
  - State `q7`: Write `o`, move right, go to state `q8`
  - State `q8`: Write `r`, move right, go to state `q9`
  - State `q9`: Write `l`, move right, go to state `q10`
  - State `q10`: Write `d`, move right, go to state `q11`
  - State `q11`: Halt
- **Result**: The tape will have "Hello World" written on it, demonstrating the machine's ability to perform specific, predefined tasks.

Turing Completeness

A system is Turing complete if it can simulate any Turing Machine. This means it can solve any problem that is computationally solvable, given enough time and resources.

- **Definition**: A programming language or computational system is Turing complete if it can simulate the computation of any Turing Machine.
- **Criteria**: The system must have conditional branching (like `if` statements), a form of repetition or looping (like `while` or `for` loops), and the ability to read and write an arbitrary amount of data.

Examples of Turing Complete Systems

- **Minecraft Redstone**: Believe it or not, you can build a fully functional computer inside Minecraft using Redstone circuits. Players have created everything from calculators to full-fledged CPUs, making Minecraft a playground for digital logic and Turing completeness.
- **Magic: The Gathering**: In 2019, researchers proved that a sufficiently complex game of Magic: The Gathering can simulate a Turing Machine. This means you could, in theory, decide computational problems while battling with your favorite cards.
- **Conway's Game of Life**: This cellular automaton, consisting of simple rules for cell survival and reproduction, can simulate a universal constructor. You could say it's like having a tiny, living computer on your screen!
- **PowerPoint**: Yes, even PowerPoint can be Turing complete. By using slide transitions and hyperlinks in creative ways, you can simulate computational processes. Who knew your next presentation could double as a computing engine?
- **Baking Recipes**: Imagine a cookbook that is Turing complete. By meticulously following a set of rules (steps), where each ingredient and instruction acts like a state transition, you can bake a cake while theoretically solving any computable problem. It might not be the most efficient way to compute, but it sure is delicious!

- **Esoteric Programming Languages (Esolang)**: Languages like Brainfuck and Befunge are intentionally designed to be as confusing and impractical as possible, yet they are Turing complete. Writing code in these languages feels like solving a riddle wrapped in an enigma, but it's computationally powerful!

12.1 P & NP

Importance of Efficient Algorithms

As computational problems grow in size and complexity, the **need for efficient algorithms becomes increasingly crucial**. Inefficient algorithms can lead to impractical running times and resource usage, making certain problems infeasible to solve. By understanding the efficiency of algorithms, we can make **informed decisions about which algorithms to use for specific problems, ensuring that we can obtain solutions in a reasonable amount of time** and with acceptable resource consumption.

P and NP Classes

- P (Polynomial) is the class of decision problems that can be **solved in polynomial time**.
  - Shortest path
  - Maximum flow
  - Linear programming.
- NP (Non-deterministic Polynomial) is the class of decision problems for which a solution can be **verified in polynomial time**.
  - Traveling salesman
  - Boolean satisfiability
  - Graph coloring.
- All problems in P are also in NP, but it is unknown whether P = NP or P ≠ NP.

Algorithm with O(n!) Time Complexity

One example of an algorithm with a time complexity of O(n!) is the brute-force approach to solving the traveling salesman problem (TSP). The TSP asks to find the shortest possible route that visits each city exactly once and returns to the starting city, given a list of cities and the distances between each pair of cities.

```
function tsp_brute_force(cities):
    best_distance = infinity
    best_path = None

    for each permutation p of cities:
        distance = calculate_distance(p)
        if distance < best_distance:
            best_distance = distance
            best_path = p

    return best_path
```

In this brute-force approach, we generate all possible permutations of the cities and calculate the total distance for each permutation. The algorithm then returns the permutation with the shortest distance as the optimal solution.

Verifying a Solution to the TSP

Suppose we have a proposed solution to an instance of the TSP, which is a permutation of the cities representing the order in which they are visited. We can verify whether this proposed solution is valid and calculate its total distance in polynomial time.

```
function verify_tsp_solution(cities, proposed_path):
    if not is_valid_permutation(proposed_path):
        return False

    total_distance = 0
    prev_city = proposed_path[-1]

    for city in proposed_path:
        distance = get_distance(prev_city, city)
```

```
        total_distance += distance
        prev_city = city

    distance_to_start = get_distance(proposed_path[-1], proposed_path[0])
    total_distance += distance_to_start

    return total_distance
```

In this verification algorithm, we first check if the proposed path is a valid permutation of the cities. If not, we return False. If it is a valid permutation, we calculate the total distance by summing the distances between consecutive cities in the proposed path and adding the distance from the last city back to the starting city.

The time complexity of this verification algorithm is O(n), where n is the number of cities, as we iterate through the permutation once to calculate the total distance. This is a polynomial-time operation.

⚠ **Implications of P vs. NP**

- If P = NP, many difficult problems could be solved efficiently, revolutionizing fields such as cryptography, optimization, and artificial intelligence.
- If P ≠ NP, it would mean that there are problems in NP that cannot be solved efficiently, and approximation algorithms or heuristics would be necessary for such problems.
- The P vs. NP problem is one of the most important open questions in computer science and has significant implications for the field and beyond.

Category	Description
P	Problems that can be solved by a deterministic Turing machine in polynomial time.
NP	Problems whose solutions can be verified by a deterministic Turing machine in polynomial time.
NP-hard	Problems that are at least as hard as the hardest problems in NP.
NP-complete	Problems that are both NP and NP-hard.

12.2 Reduction

🔥 **Reduction**

In computational complexity theory, a reduction is a way to **convert one problem into another problem in such a way that a solution to the second problem can be used to solve the first problem**. Reductions are used to show the relationship between the difficulty of different problems.

Specifically, a polynomial-time reduction from problem A to problem B means that any instance of problem A can be transformed into an instance of problem B in polynomial time, such that a solution to the instance of problem B gives a solution to the original instance of problem A.

Reductions are crucial for understanding NP-hardness and NP-completeness:

1. To prove that a problem X is NP-hard, we show that every problem in NP can be reduced to X in polynomial time. This means that X is at least as hard as any problem in NP.
2. To prove that a problem Y is NP-complete, we first show that Y is in NP, and then we find a known NP-complete problem that can be reduced to Y in polynomial time. This reduction proves that Y is NP-hard, and since Y is also in NP, it is NP-complete.

Reductions help us understand the relative difficulty of problems and are essential for identifying NP-hard and NP-complete problems. By reducing a known NP-hard or NP-complete problem to a new problem, we can prove the hardness or completeness of the new problem.

⚠ **ELI5: Reductions**

Imagine you have two different types of puzzles: Puzzle A and Puzzle B. Reductions are like a way to convert Puzzle A into Puzzle B, so that if you know how to solve Puzzle B, you can use that knowledge to solve Puzzle A as well.

Think of it like this:

1. You have a bunch of Puzzle A pieces, and you want to solve Puzzle A.
2. You find a way to change the Puzzle A pieces into Puzzle B pieces.

3. You solve Puzzle B using the pieces you got from changing Puzzle A.
4. Because of the way you changed Puzzle A into Puzzle B, the solution to Puzzle B also gives you the solution to Puzzle A!

Reductions help us understand how hard different puzzles are compared to each other. If we can change a really hard puzzle into an easier puzzle, it means that the easier puzzle is at least as hard as the really hard puzzle.

However, in real life, we don't actually solve puzzles by changing them into other puzzles. Reductions are more like a "theoretical magic trick" that computer scientists use to understand how puzzles relate to each other in terms of their difficulty. It's not something we do to solve puzzles in practice, but it helps us understand which puzzles are harder than others.

Reduction Example: Independent Set to Clique

Consider the Independent Set problem (finding the largest set of nodes in a graph with no edges between them) and the Clique problem (finding the largest set of nodes in a graph where every node is connected to every other node).

We can reduce the Independent Set problem to the Clique problem in polynomial time by taking the complement of the graph (i.e., replacing every edge with a non-edge and every non-edge with an edge). In the complement graph, an independent set becomes a clique and vice versa.

This reduction shows that if we had a polynomial-time algorithm to solve the Clique problem, we could use it to solve the Independent Set problem in polynomial time as well. Since the Independent Set problem is known to be NP-hard, this reduction proves that the Clique problem is also NP-hard.

12.3 NP Hardness

NP-Hardness

A problem X is said to be NP-hard if every problem in NP can be reduced to X in polynomial time. In other words, if we have a polynomial-time algorithm that solves X, we can use it to solve any problem in NP in polynomial time by first reducing the NP problem to X and then solving X.

Formally, a problem X is NP-hard if, for every problem Y in NP, there exists a polynomial-time reduction from Y to X. This means that we can transform any instance of Y into an instance of X in polynomial time, such that a solution to the X instance gives us a solution to the original Y instance.

It's important to note that NP-hard problems don't necessarily have to be in NP themselves. In fact, some NP-hard problems, like the Halting Problem, are undecidable, meaning there is no algorithm that can always give a correct answer for every instance of the problem.

If a certain puzzle was NP hard, then we can say that every other puzzle can be transformed into it to find a solution

NP-Hardness: The Party Problem

Imagine you're planning a party and want to invite a group of friends. However, some of your friends don't get along with each other. You want to invite the maximum number of friends possible while ensuring that no two friends who don't get along are invited together.

This problem is similar to the Maximum Independent Set problem, which is NP-hard. In this context, each friend represents a node in a graph, and an edge between two nodes means that those two friends don't get along. Finding the largest group of friends who can attend the party without any conflicts is equivalent to finding the maximum independent set in the graph, which is an NP-hard problem.

The party problem illustrates the concept of NP-hardness because if you had a polynomial-time algorithm to solve this problem, you could use it to solve many other hard problems in NP by reducing them to the party problem.

12.4 NP Completeness

NP-Completeness

A problem is NP-complete if it is both in NP and NP-hard. In other words, an NP-complete problem is a problem in NP that is at least as hard as any other problem in NP.

To prove that a problem X is NP-complete, we need to show two things:

1. X is in NP, meaning that given a solution, we can verify its correctness in polynomial time.
2. X is NP-hard, meaning that every problem in NP can be reduced to X in polynomial time.

NP-complete problems are the hardest problems in NP because if we find a polynomial-time algorithm for any NP-complete problem, we would have a polynomial-time algorithm for all problems in NP, proving that P = NP.

NP-Completeness: The Jigsaw Puzzle Problem

Consider a jigsaw puzzle with a large number of pieces. The goal is to determine whether it's possible to assemble the puzzle in a specific way, given a picture of the completed puzzle.

This problem is NP-complete because:

1. Given a proposed solution (an arrangement of the puzzle pieces), you can easily verify whether it matches the completed picture in polynomial time, so the problem is in NP.
2. The problem is also NP-hard, as many other problems in NP, such as the Hamiltonian Cycle problem, can be reduced to the jigsaw puzzle problem. For example, you could encode a Hamiltonian Cycle problem as a jigsaw puzzle, where each piece represents a node in the graph, and the connections between pieces represent the edges. Solving the puzzle would then be equivalent to finding a Hamiltonian cycle in the original graph.

The jigsaw puzzle problem demonstrates the concept of NP-completeness because it is both in NP (verifiable in polynomial time) and NP-hard (other NP problems can be reduced to it).