كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE
LAB 03 – SEARCHING PROBLEMS (1)

Eng. Sammer Kamal

Eng. Yousef Elbaroudy

2024/2025

GUIDELINES

You don’t need to memorize what will be mentioned. Instead, try to understand

Each PowerPoint Presentation will be available as PDF file on Google Drive Folder of the Course

REMEMBER !

In our course, we are targeting

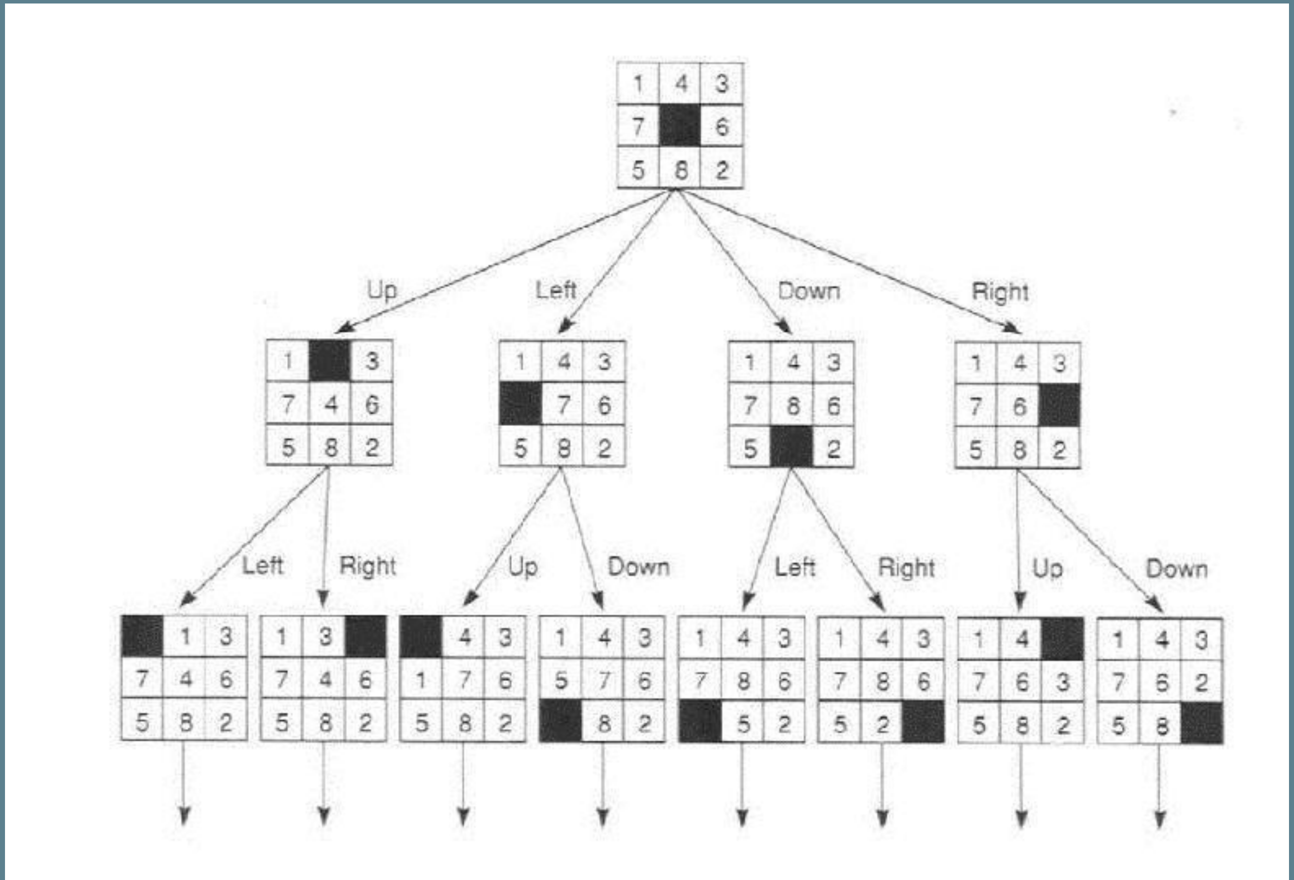
Goal-based agent in fully observable, deterministic and discrete environments

Which means, all of our problems will be in the form of states that shows all possible forms of the environment

STATE SEARCH SPACE

STATE SEARCH SPACE

state search space is the collection of all possible states a system can be in while searching for a solution to a given problem.

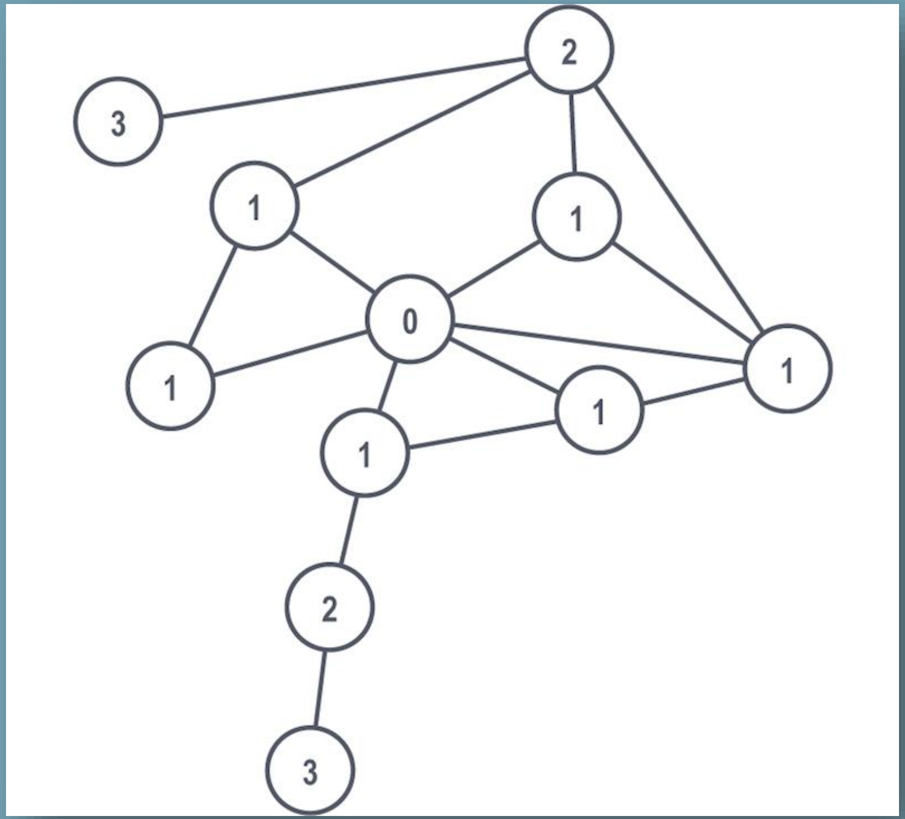


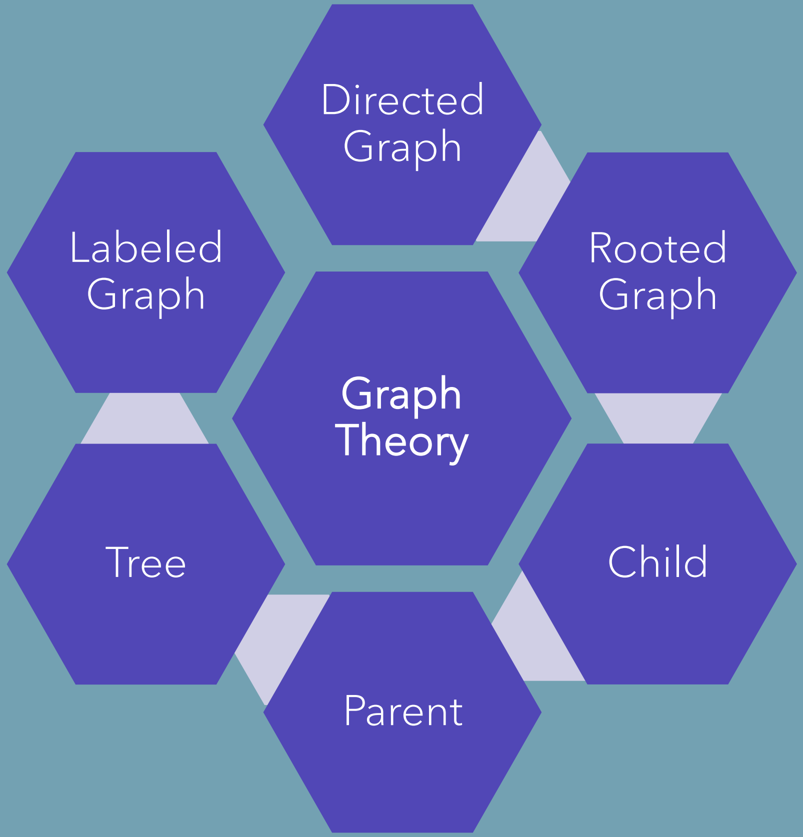
KEY ASPECTS OF "STATE SEARCH SPACE"

- States
- Initial State
- Goal State
- Operators
- Space Graph
- Search Path
- Optimal Solution
- Strategy

(1) SPACE GRAPH

- A representation of all possible states and transitions between them.





(1) SEARCH GRAPH

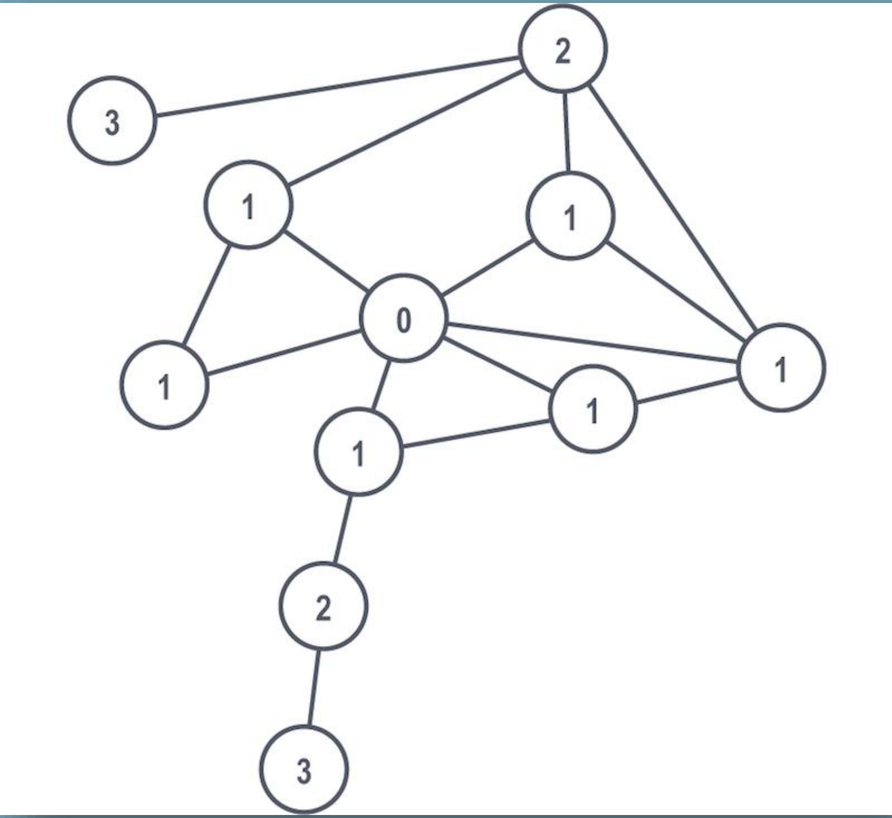
What is "Graph" ?

- A **graph** is a data structure used to represent relationships between objects. It consists of:
  1. Nodes (Vertices, V) → Represent objects or states.
  2. Edges (E) → Represent connections or relationships between nodes.

Directed Graph	Undirected Graph	Weighted Graph	Unweighted Graph	Cyclic Graph
	Acyclic Graph	Connected Graph	Disconnected Graph	

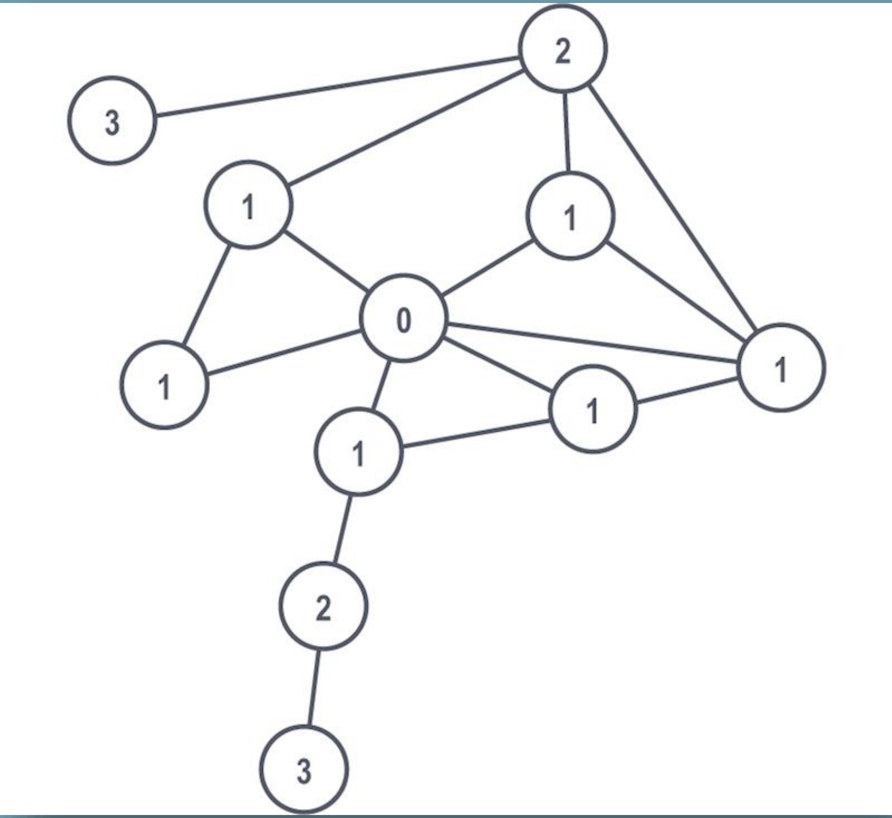
(2) STATES

- Each unique configuration of the system is called a state.



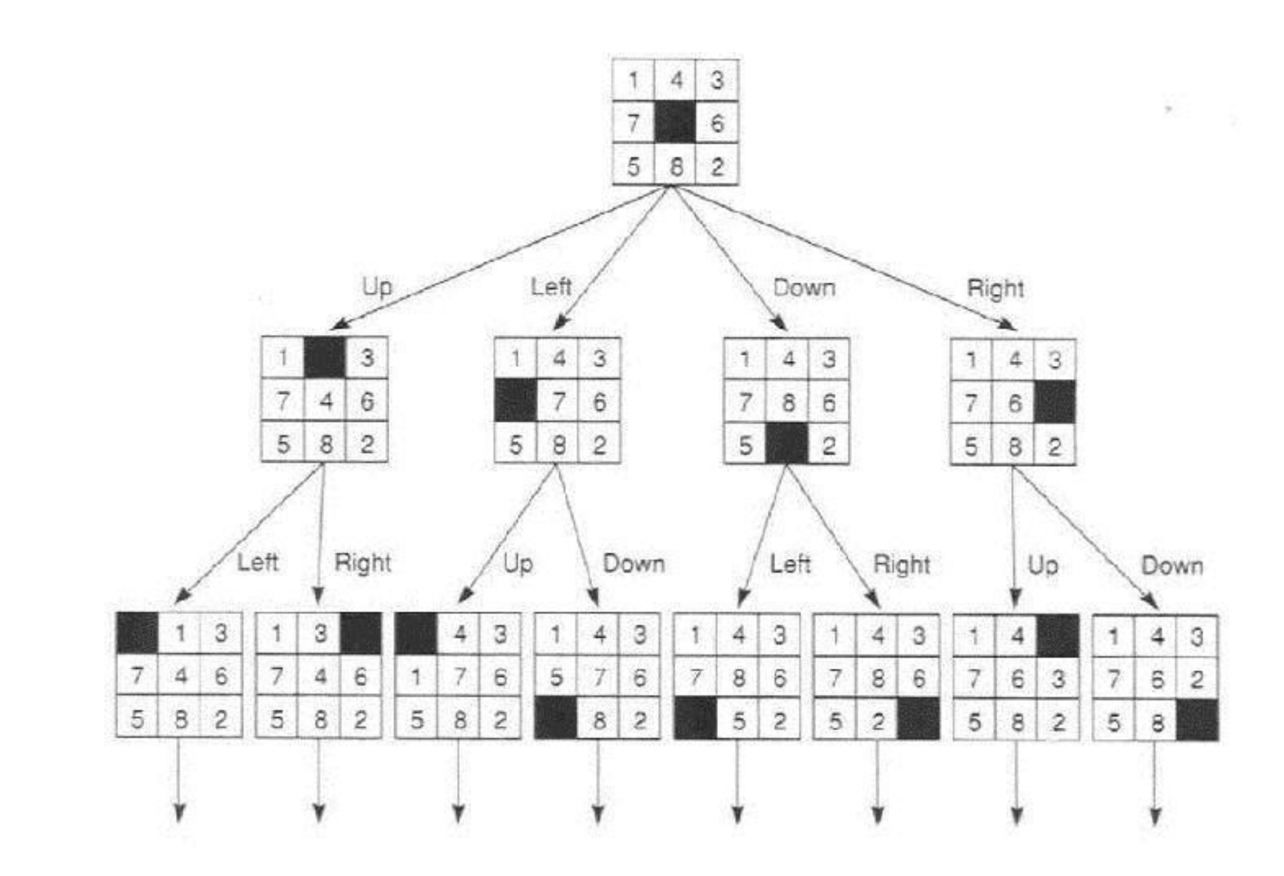
(3) INITIAL STATE

- The starting point in the search process.



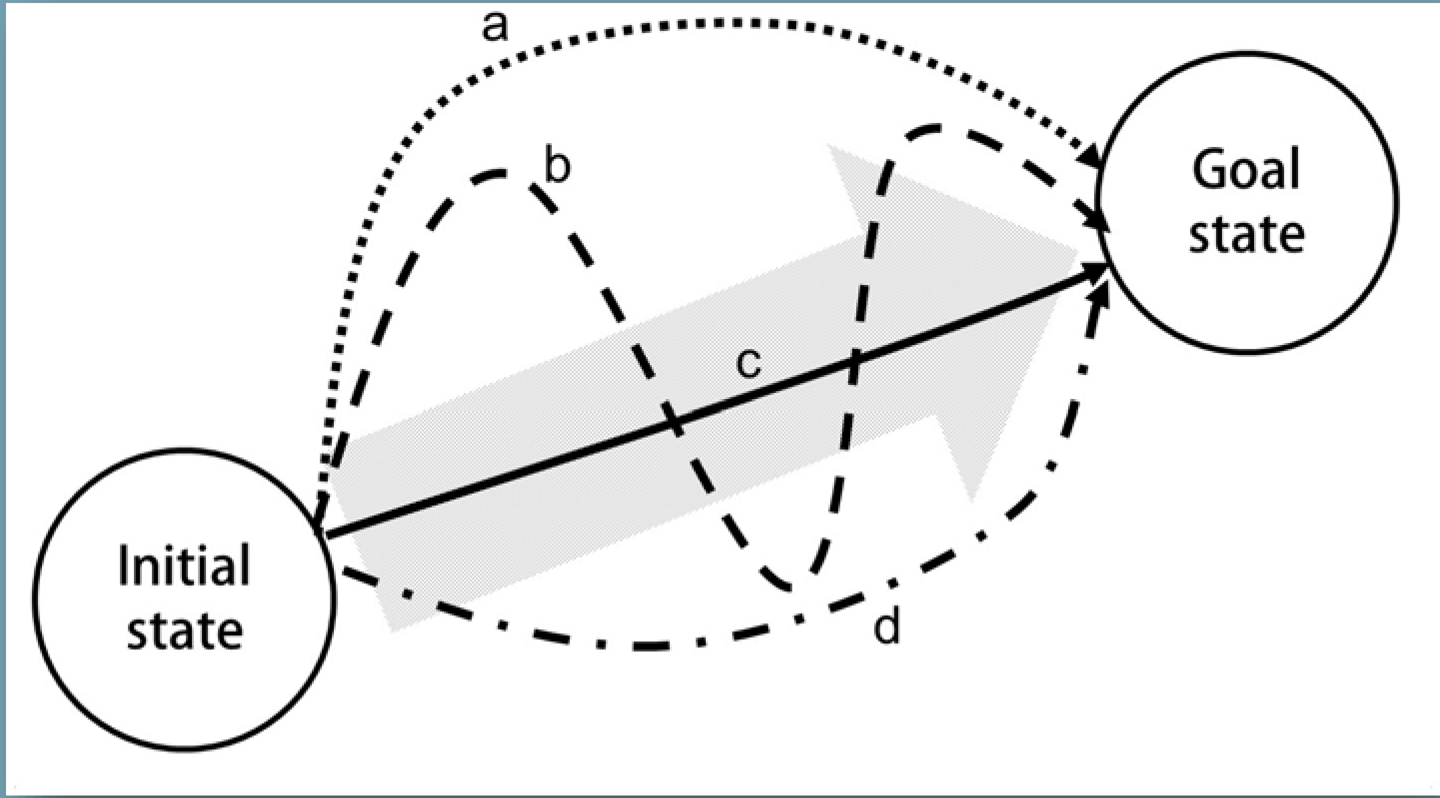
(4) OPERATORS (STATE TRANSITIONS)

- Actions that move from one state to another.



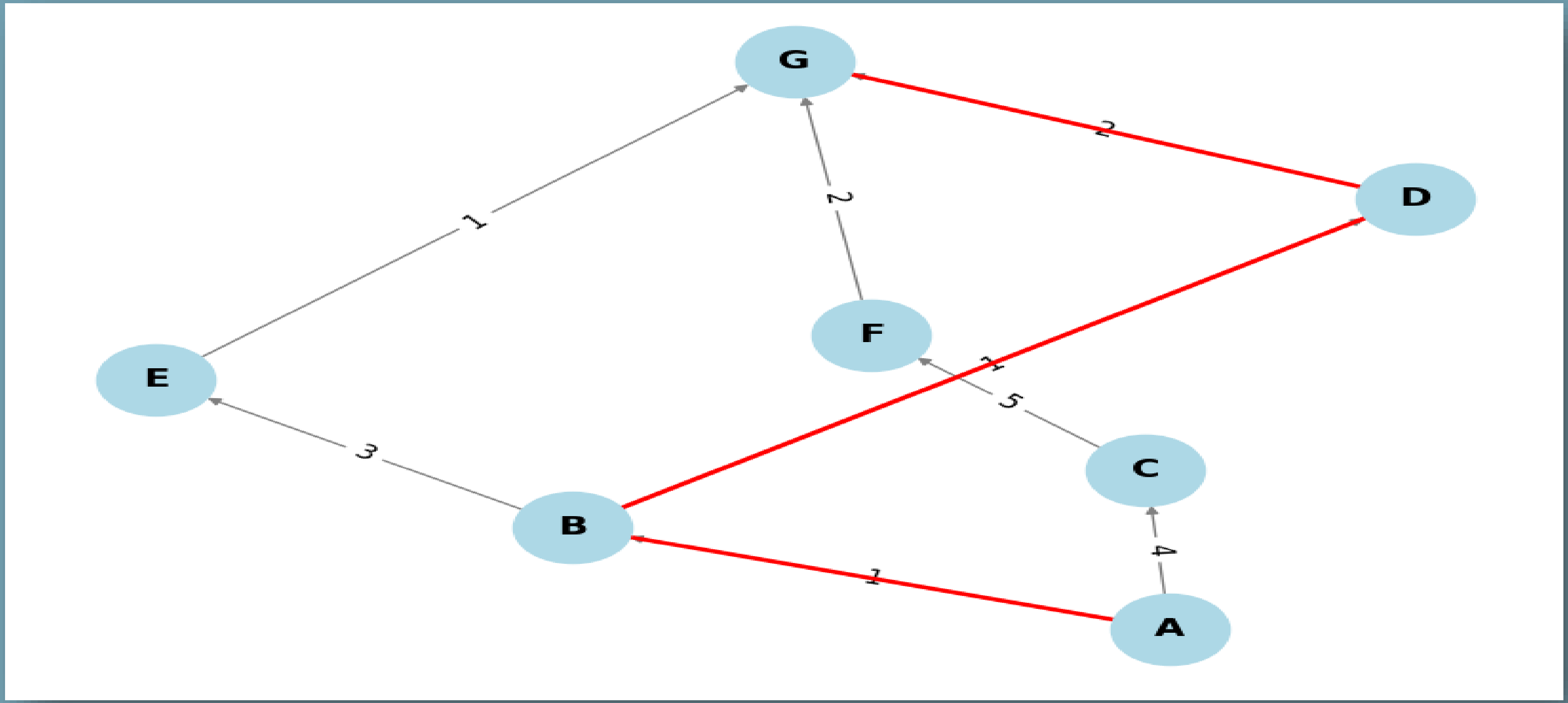
(5) GOAL STATE

- The state that meets the problem's success conditions.



(6) SEARCH PATH

- The sequence of states from the initial state to the goal state.



(7) SEARCH STRATEGY

- Depth First Search
- Breadth First Search
- A* Search
- Uniform cost search
- Greedy search
- Bidirectional search
- Iterative deepening search

(8) OPTIMAL SOLUTION PATH

- The best sequence of actions that leads to the goal state efficiently.

Shortest Path

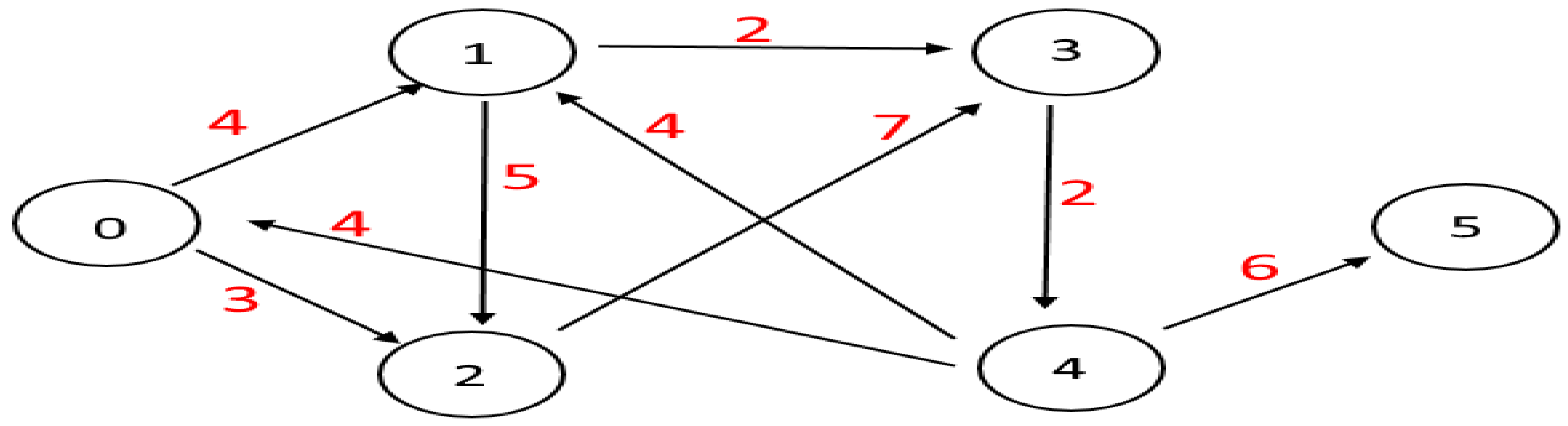
Lowest Cost

Fastest Path

How to measure the performance **“Optimality”** ?
1- Reaching the goal
2- How **“expensive”** that path to the goal is

(9) ADDITIONAL: PATH COST

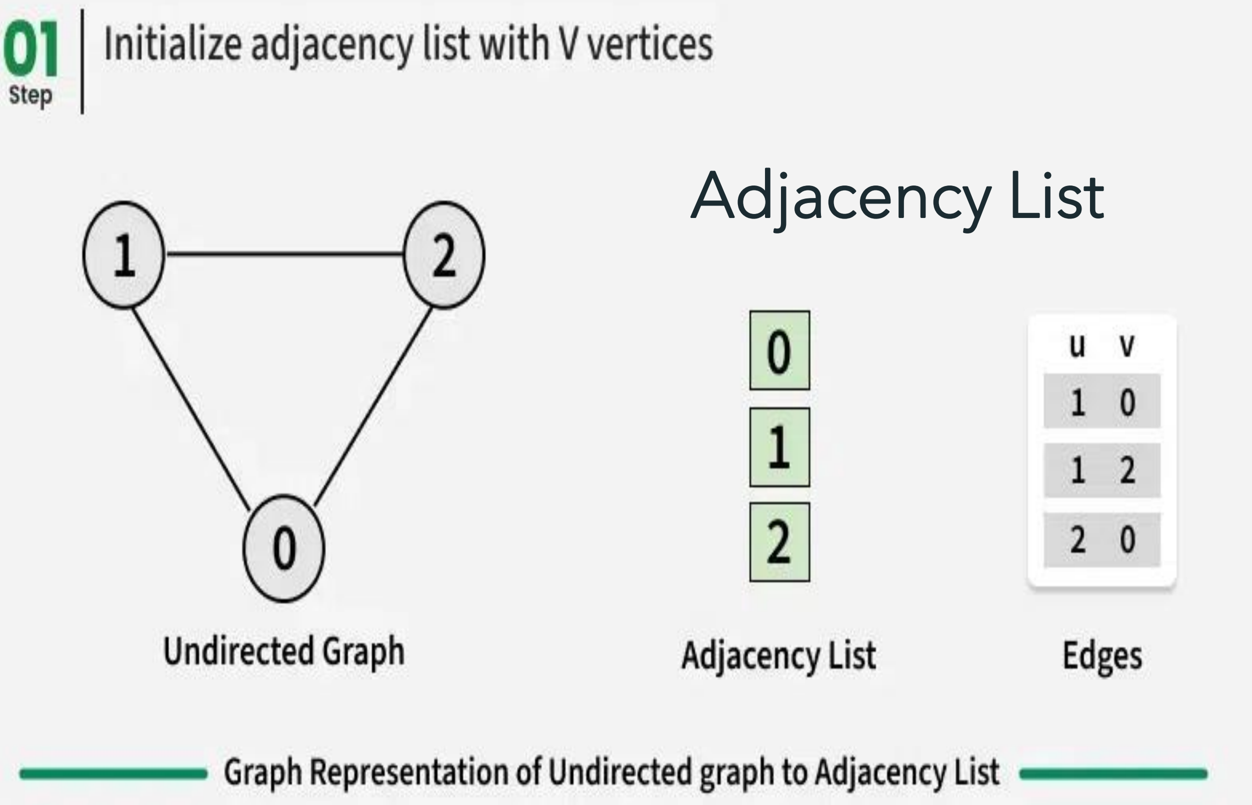
- **Path Cost** refers to the total weight or cost of a path between two nodes in a graph.
- **For the solution path**, the total cost is the **sum of edge weights** along a specific path.



Weighted Graph

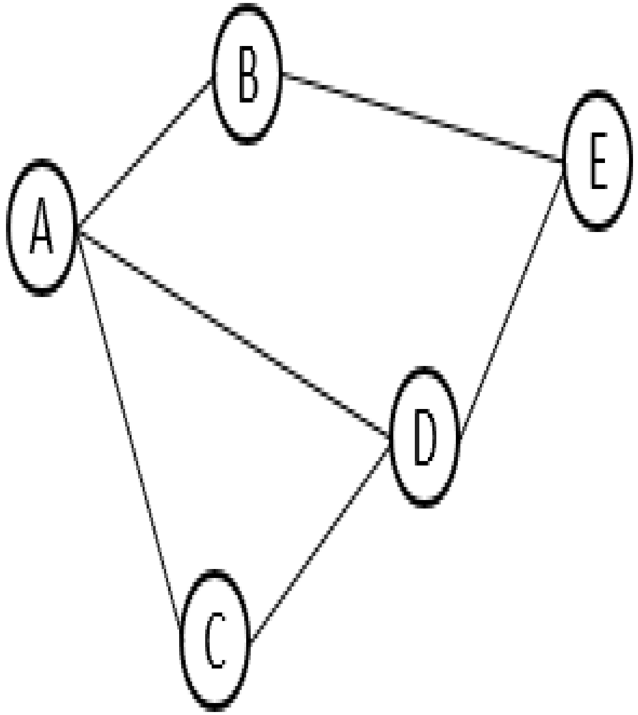
GRAPH REPRESENTATIONS

UNDIRECTED GRAPH



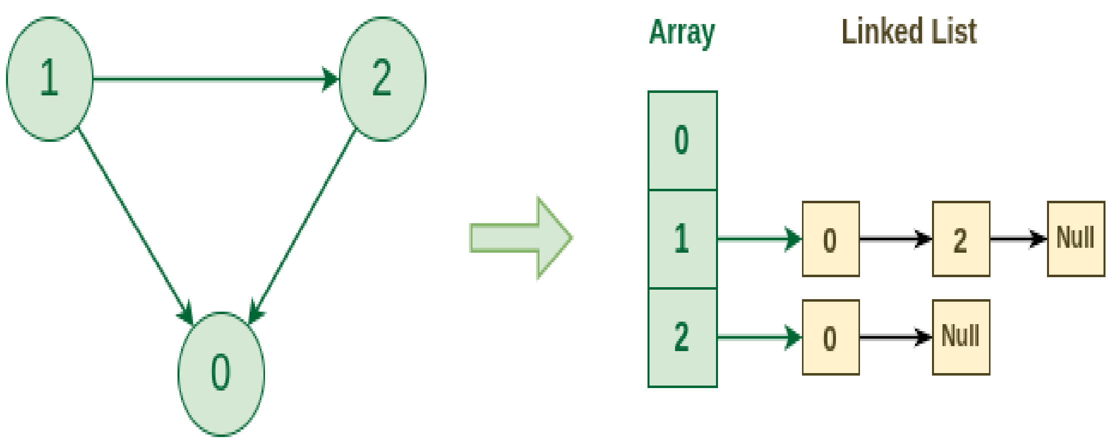
Adjacency List

Adjacency Matrix



$$\begin{array}{c|ccccc} & A & B & C & E & D \\ \hline A & 0 & 1 & 2 & 2 & 0 \\ B & 1 & 0 & 0 & 0 & 2 \\ C & 2 & 0 & 0 & 1 & 0 \\ D & 2 & 0 & 1 & 0 & 1 \\ E & 0 & 2 & 0 & 1 & 0 \end{array}$$

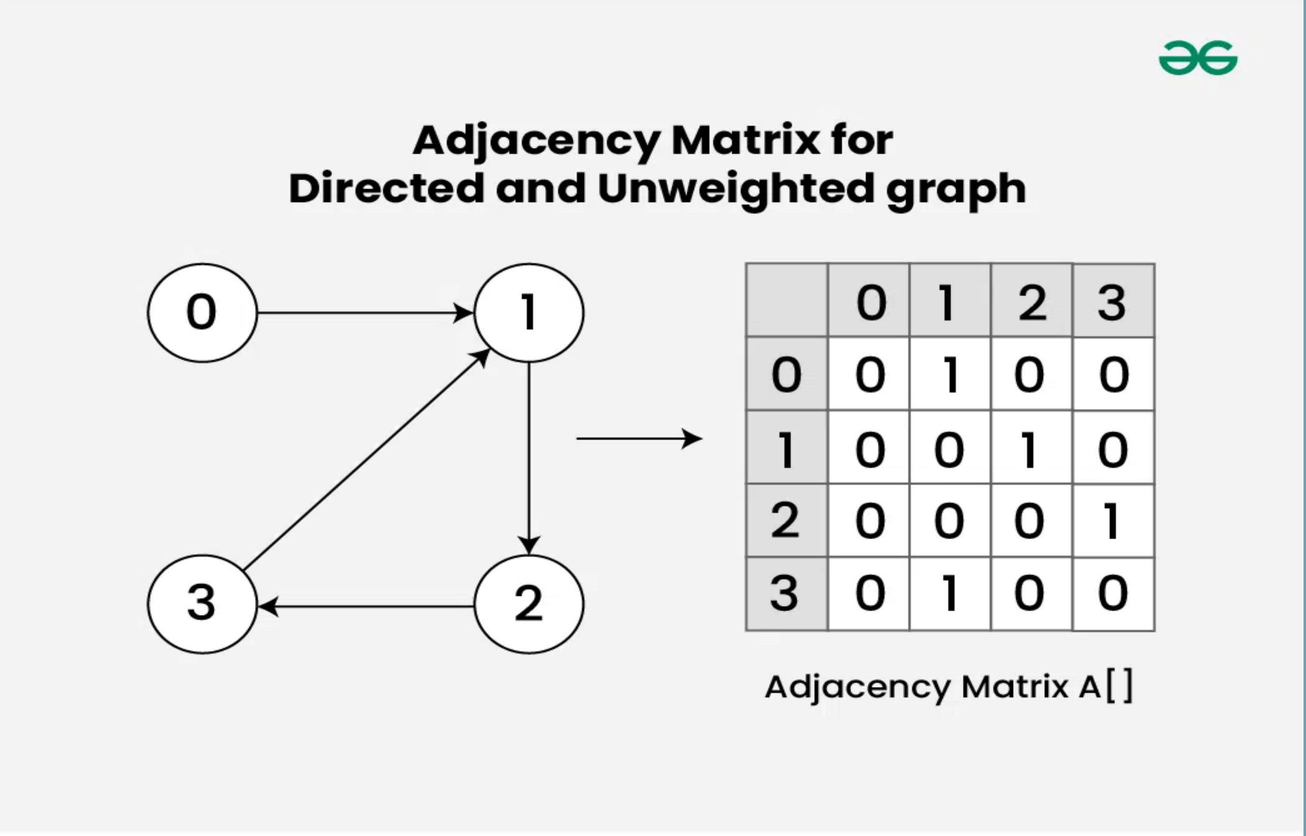
DIRECTED GRAPH



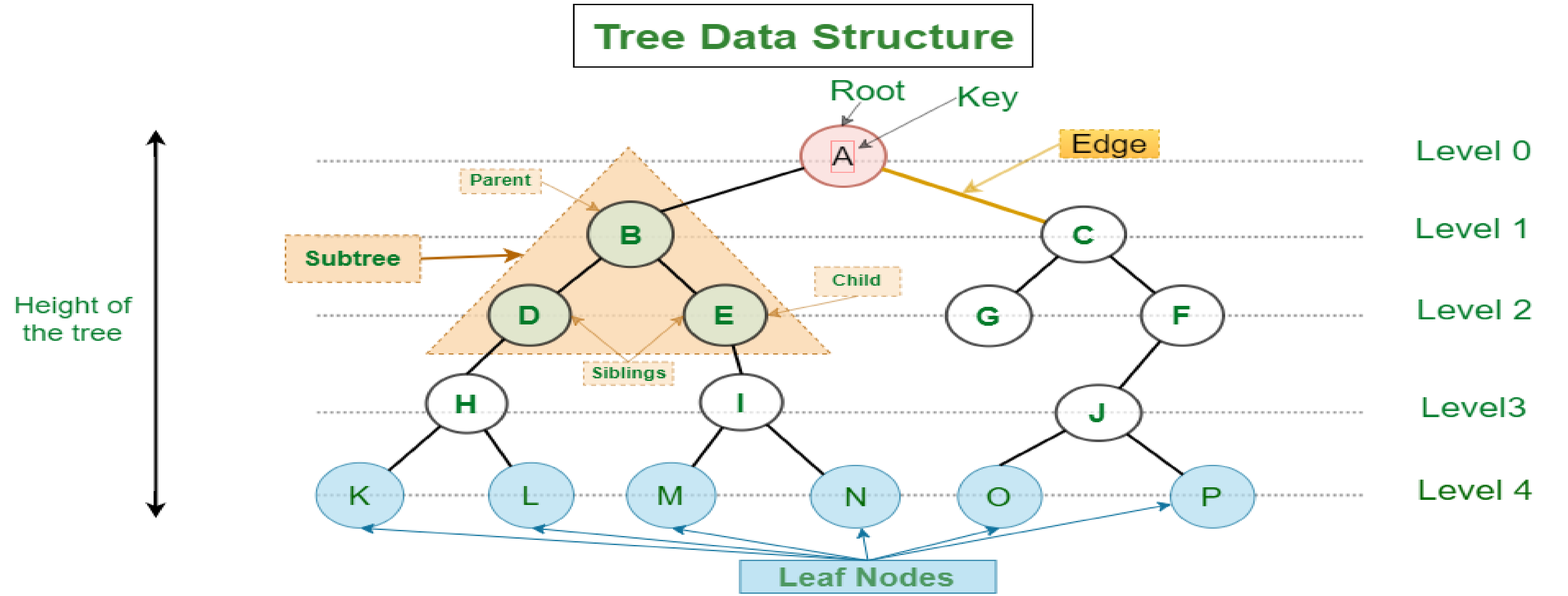
Directed Graph

Adjacency List

Graph Representation of Directed graph to Adjacency List



TREES



TREES

Special Type of graphs !

TREE TRAVERSING

- **Tree traversal** refers to the process of visiting each node in a tree **systematically**. It is essential for searching, sorting, and processing data in tree-based structures.

Pre-order	In-order	Post-order
Root-L-R	L-Root-R	L-R-Root

TREE TRAVERSING

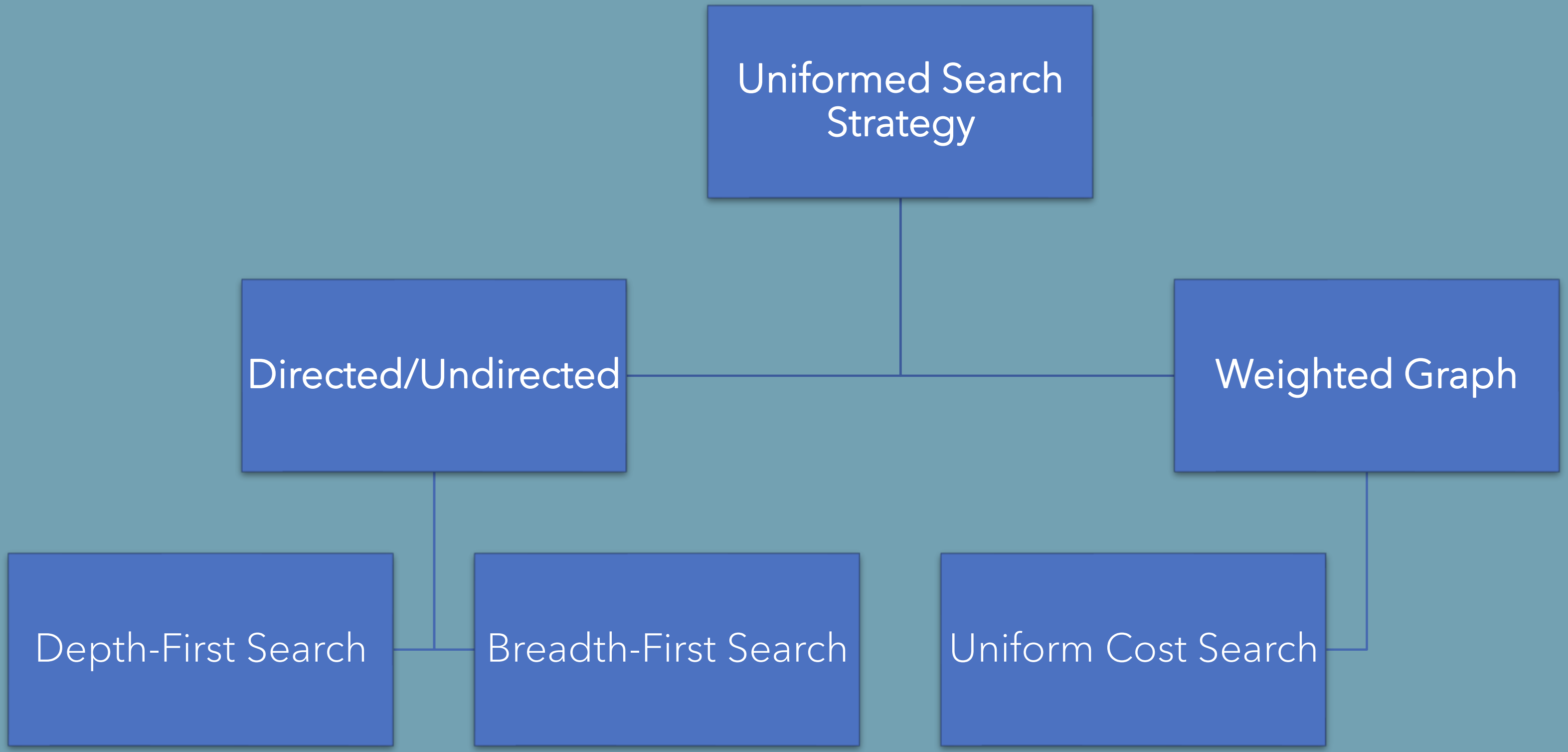
- **Tree traversal** refers to the process of visiting each node in a tree **systematically**. It is essential for searching, sorting, and processing data in tree-based structures.

Pre-order	In-order	Post-order
Root-L-R	L-Root-R	L-R-Root

HOW TO SEARCH ?
(SEARCH STRATEGY)

UNIFORMED SEARCH

- **Uniformed Search (Blind Search):** a type of search algorithm used in artificial intelligence to explore a problem space without any additional information about the state to reach it.



BASIC SEARCH

1. Initialize an **empty list** and put the **initial state** in it
2. Take the first state in the **list as the current if** **<u>it is not visited yet</u>** **otherwise skip it**, and remove it from the list
3. **Check if the current is the goal state**, if it is, then terminate the search and **return the solution path**
4. Otherwise, expand the current of its successors, **and add them into the list using a queuing function**
5. Repeat from (2) to (4)
6. If the current becomes empty, then there is no solution and **return "fail"**

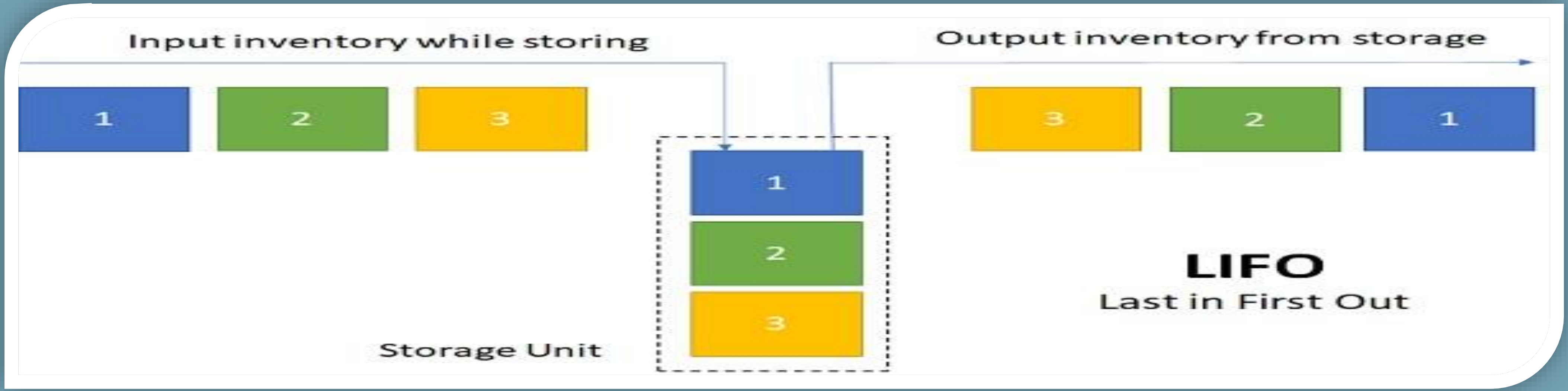
Don’t Worry !

The only difference between all strategies is the <u>queuing function</u>

DEPTH-FIRST SEARCH (العمق أولاً)

LIFO Function

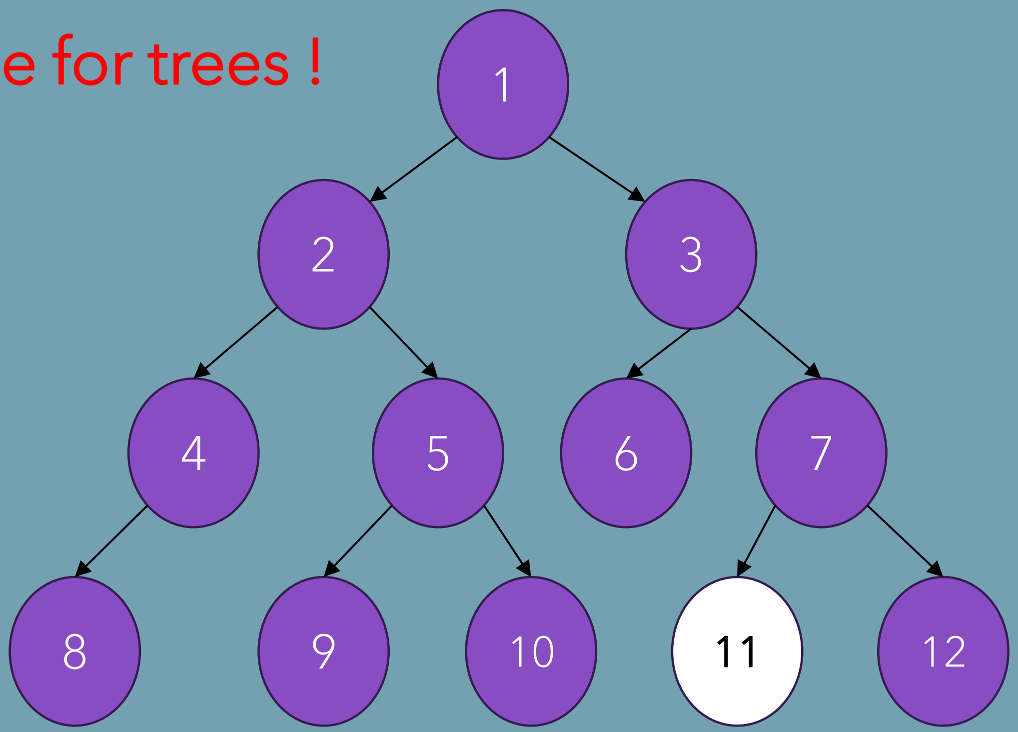
LIFO (Last In, First Out) is a data access principle where the last element added is the first one to be removed.



DEPTH-FIRST SEARCH

Goal: 11

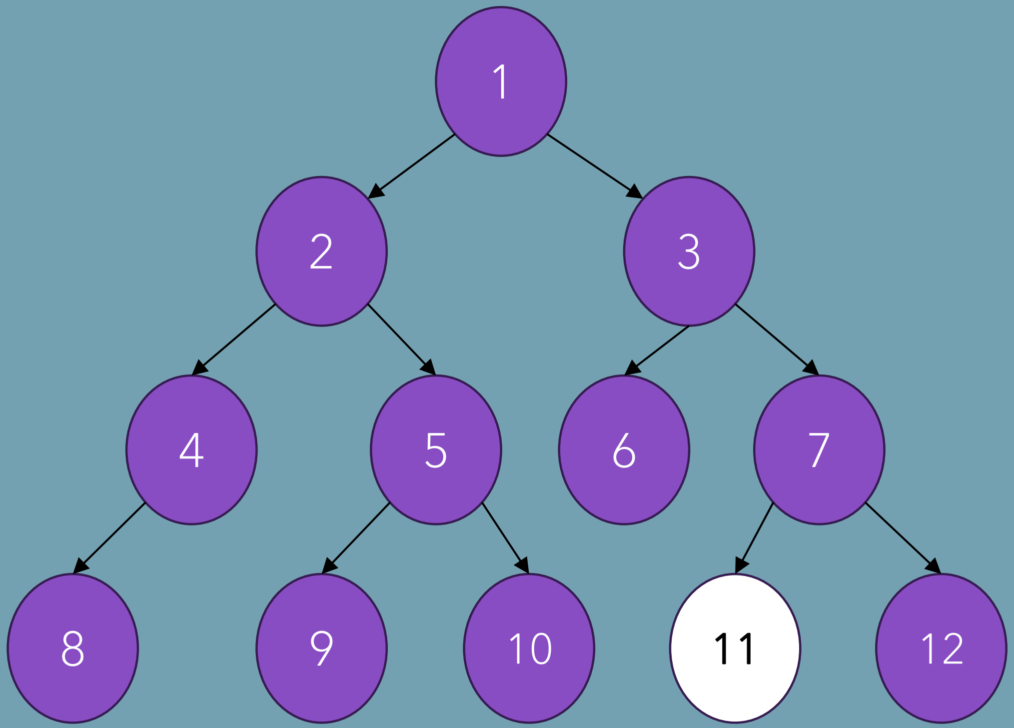
Remember:
Only Pre-order traverse for trees !



List

DEPTH-FIRST SEARCH

Goal: 11



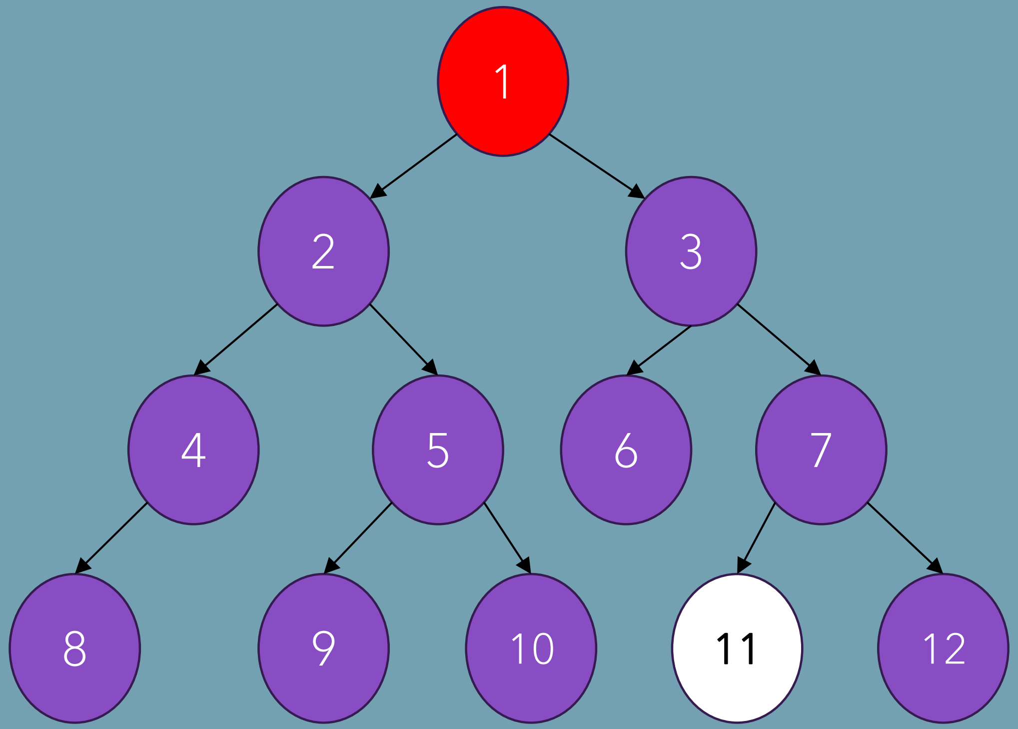
List

1

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11

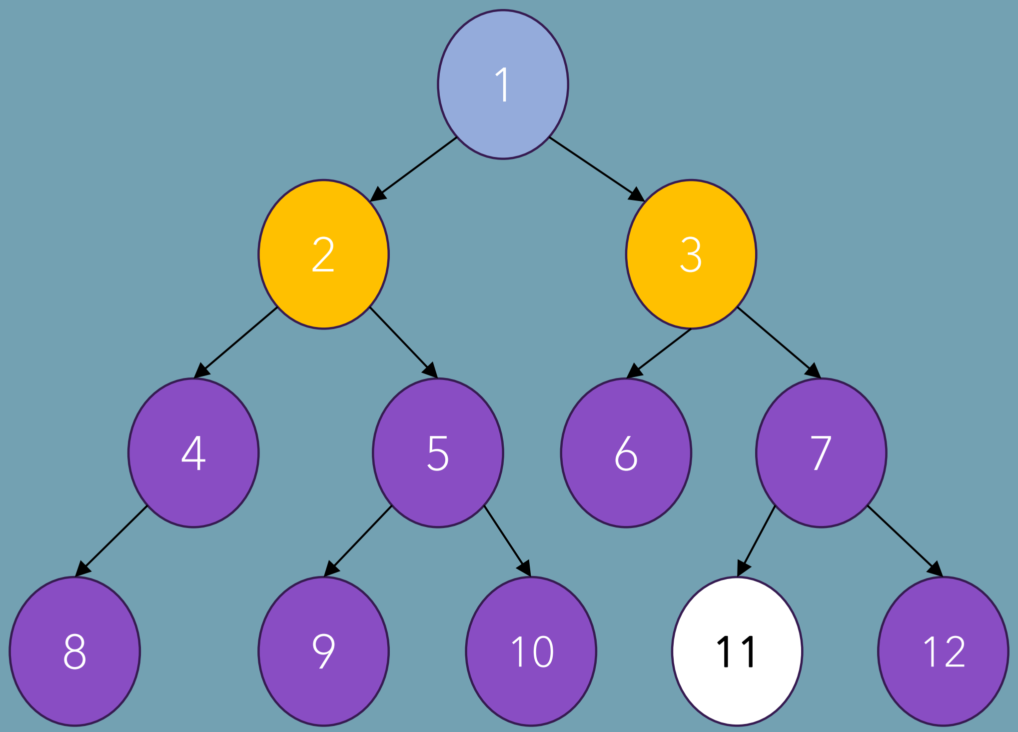


List

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



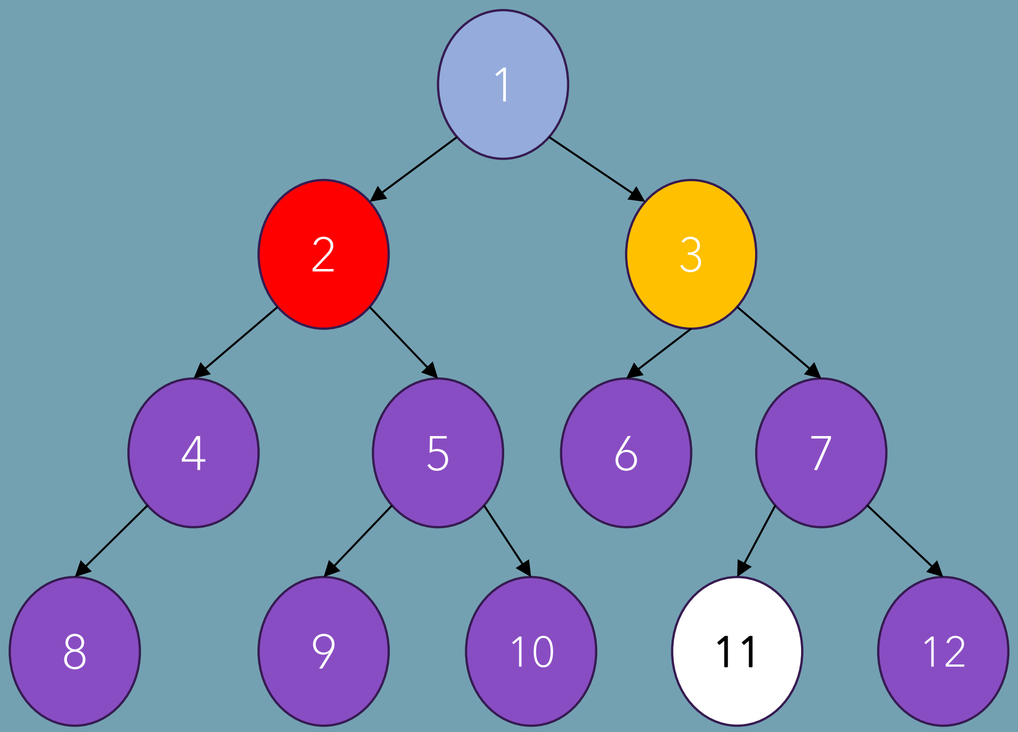
List

3,2

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



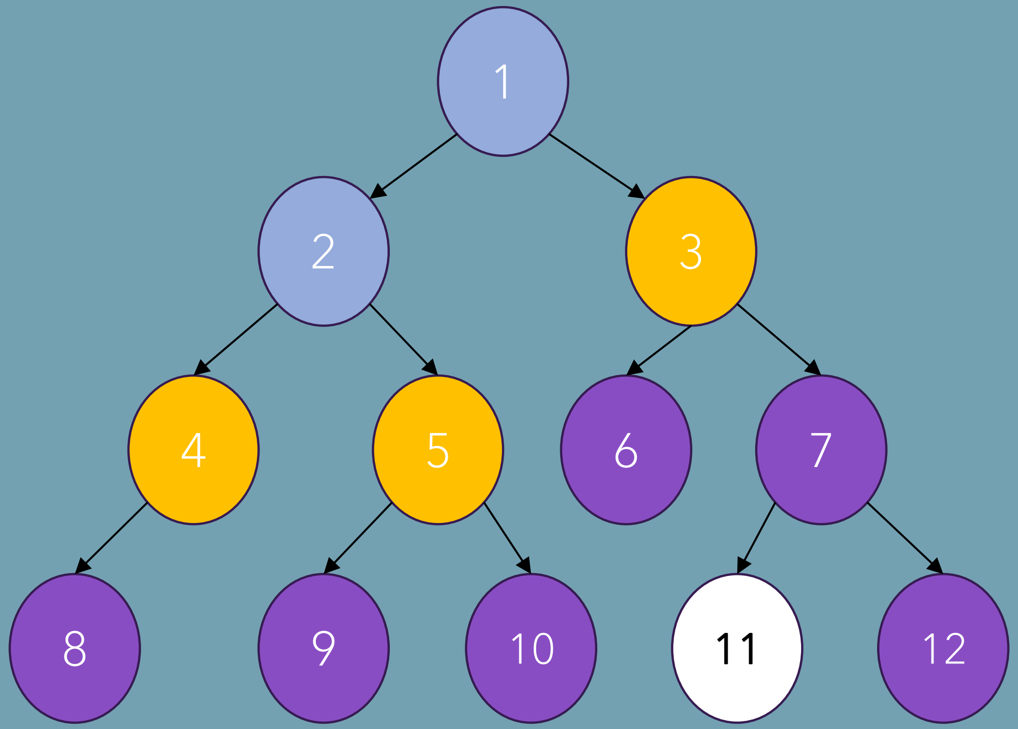
List

3

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



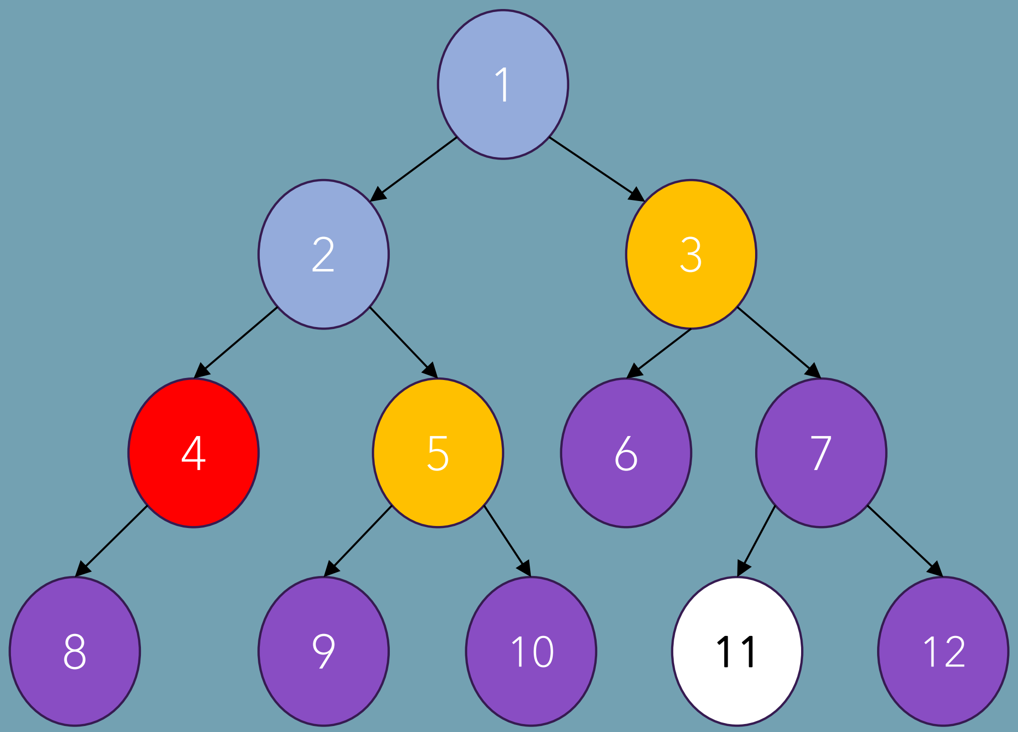
List

3,5,4

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



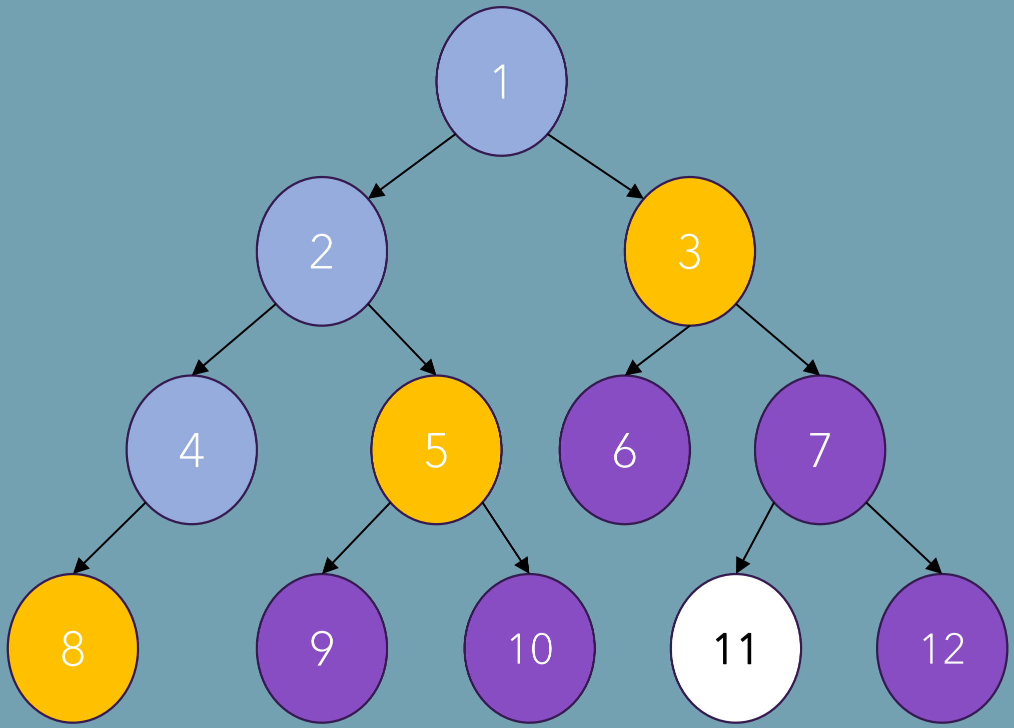
List

3,5

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



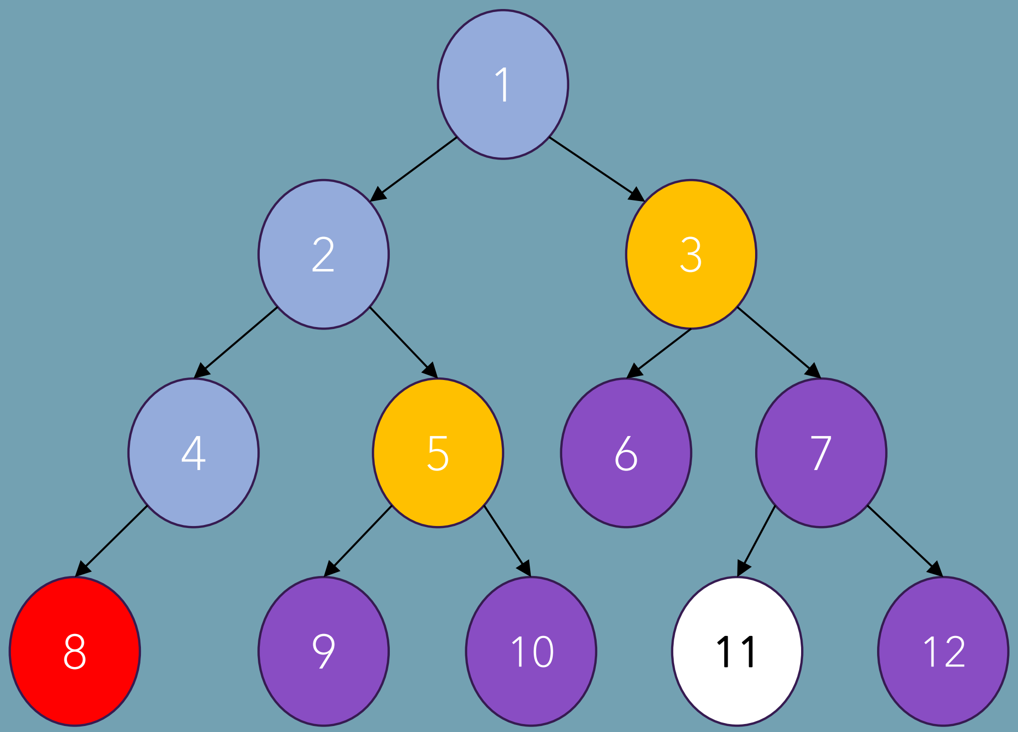
List

3,5,8

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



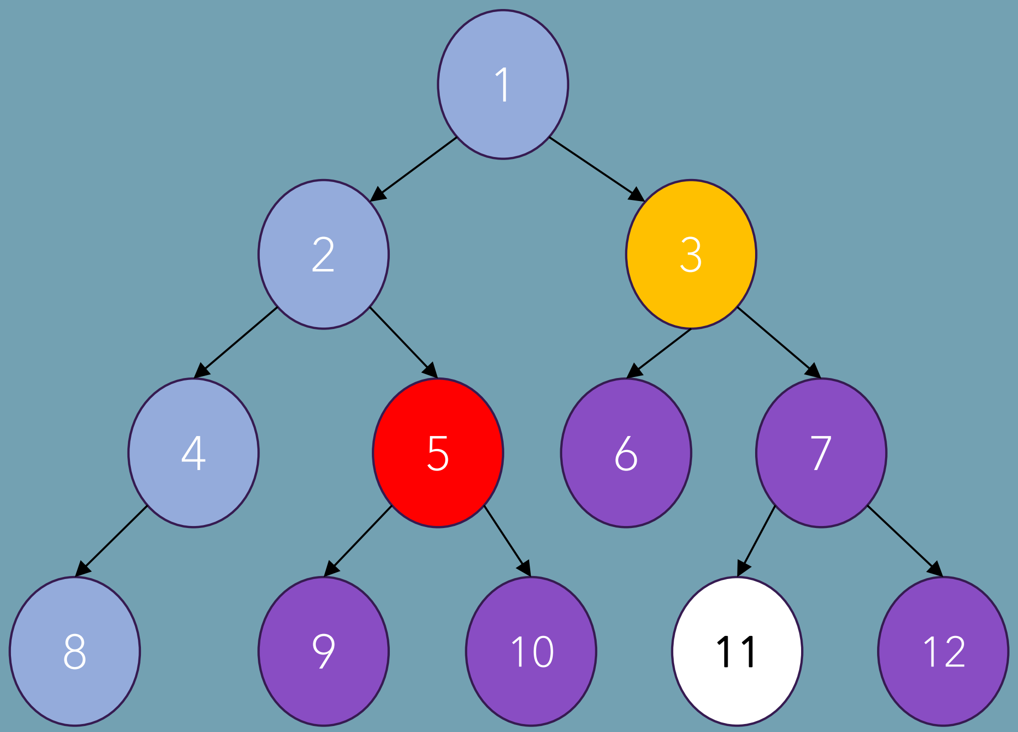
List

3,5

Direction of insert/remove

DEPTH-FIRST SEARCH

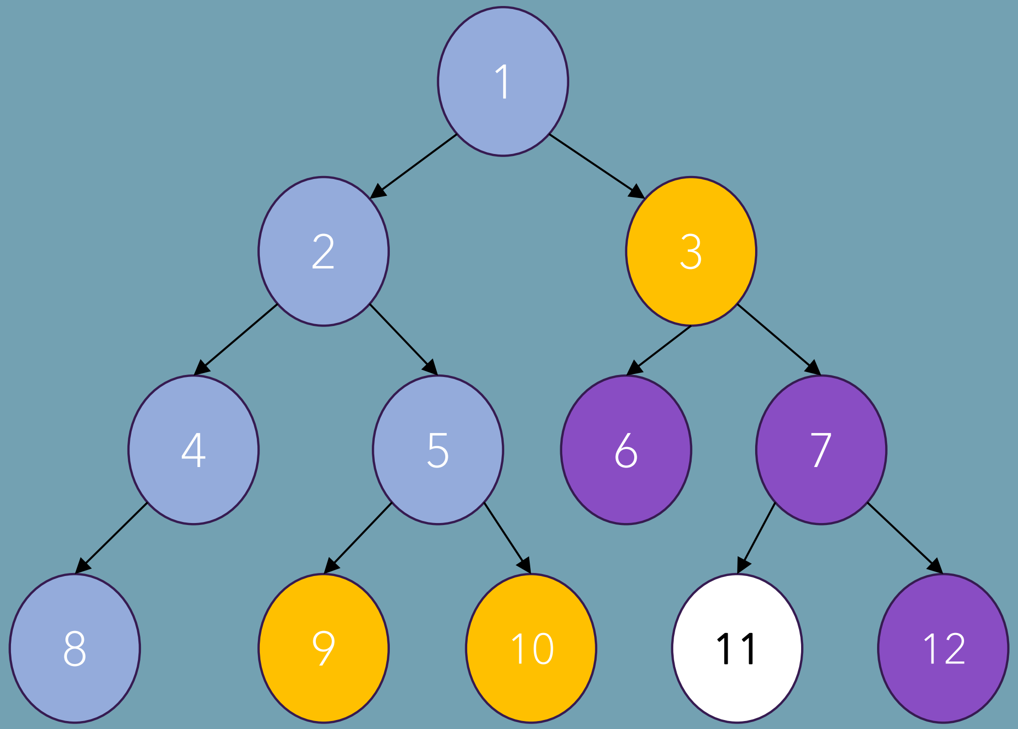
Goal: 11



List

DEPTH-FIRST SEARCH

Goal: 11



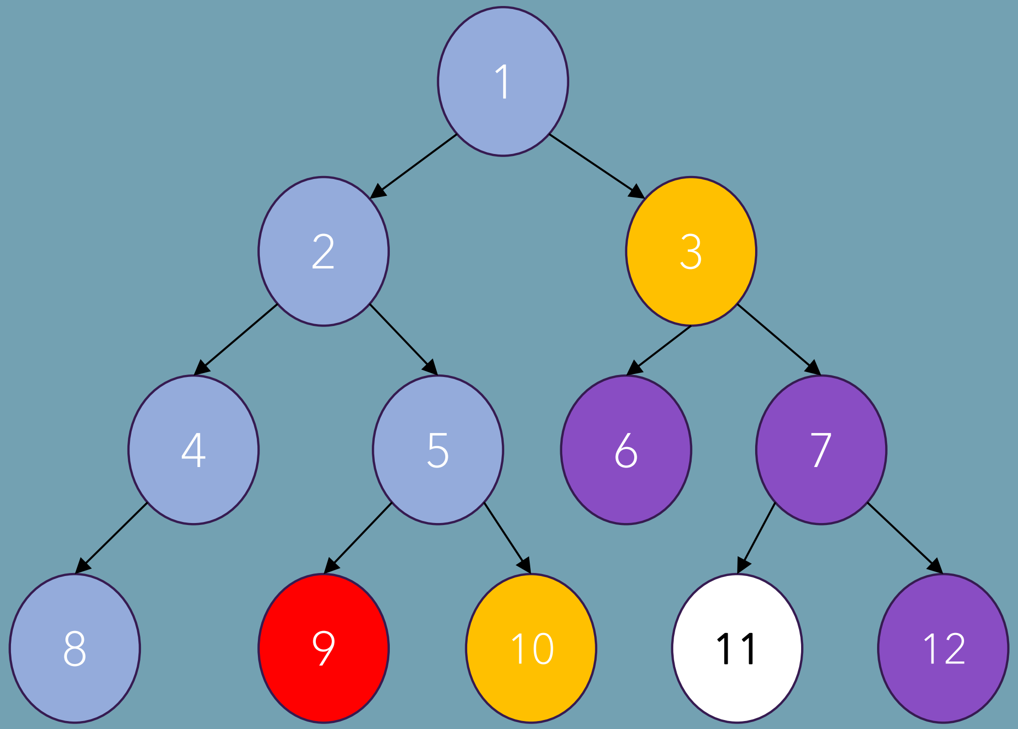
List

3,10,9

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



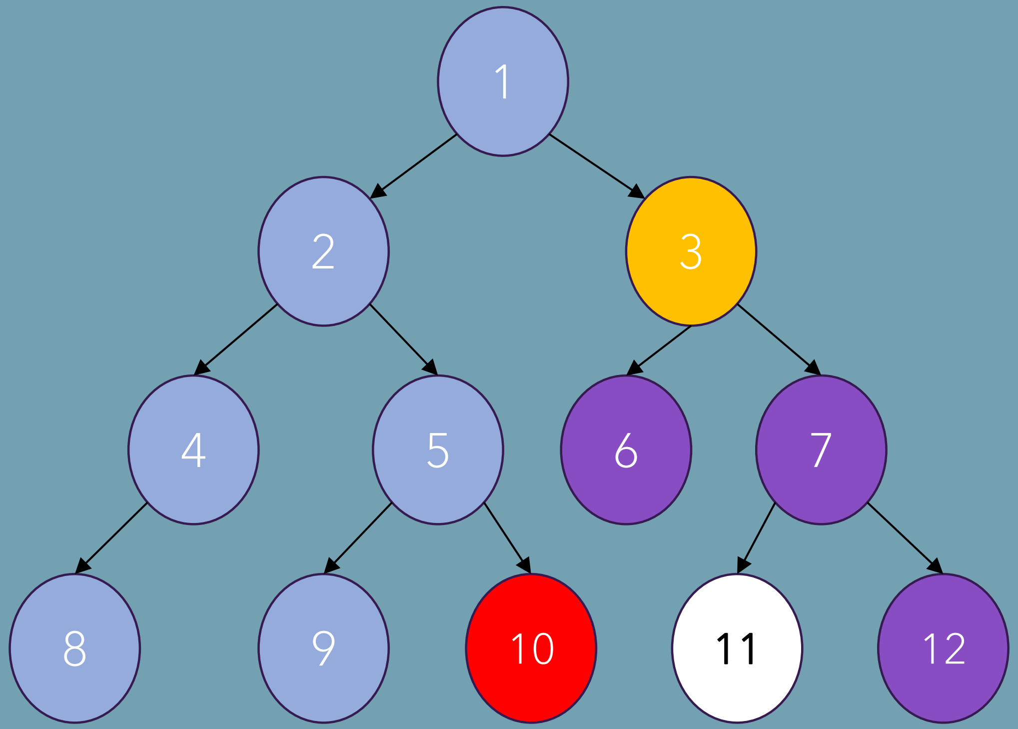
List

3,10

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



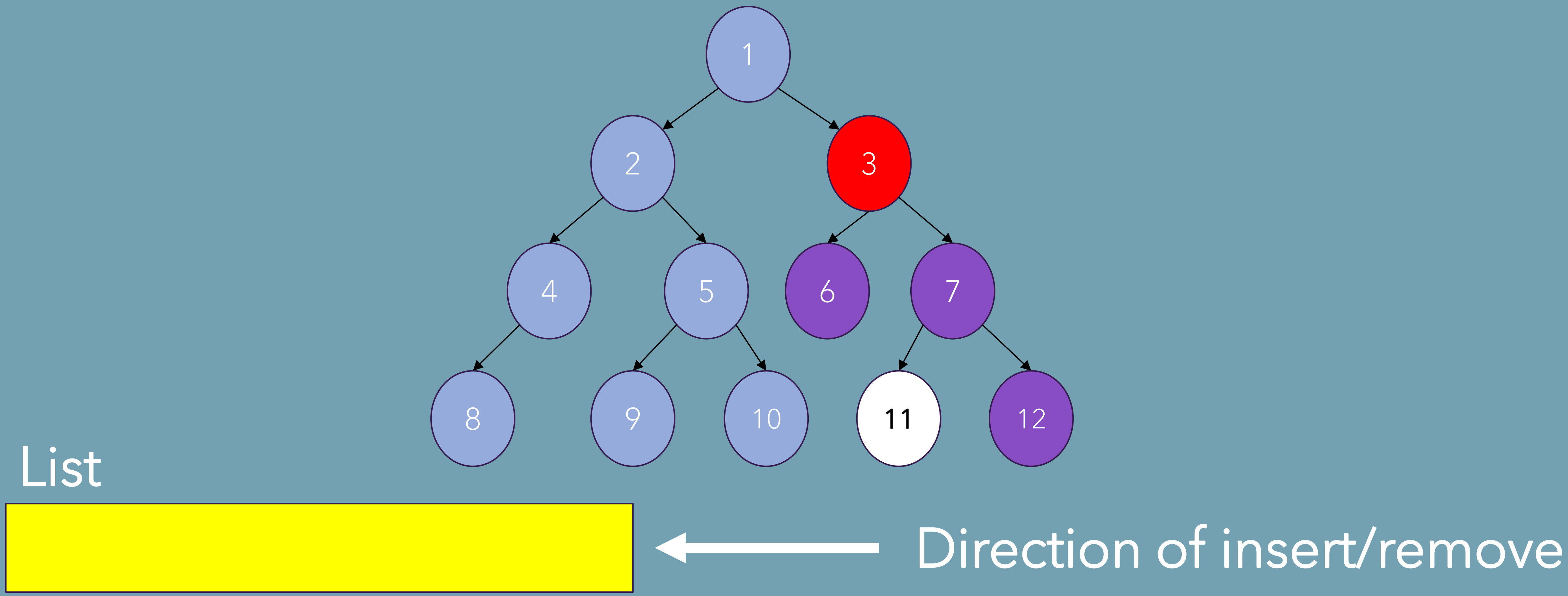
List

3

Direction of insert/remove

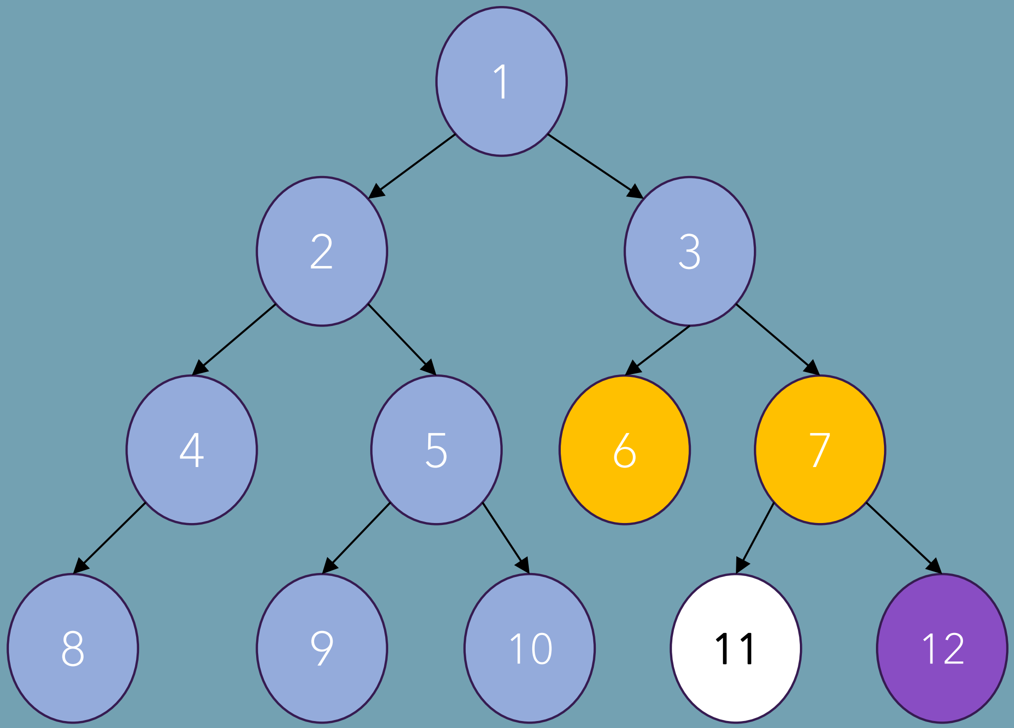
DEPTH-FIRST SEARCH

Goal: 11



DEPTH-FIRST SEARCH

Goal: 11



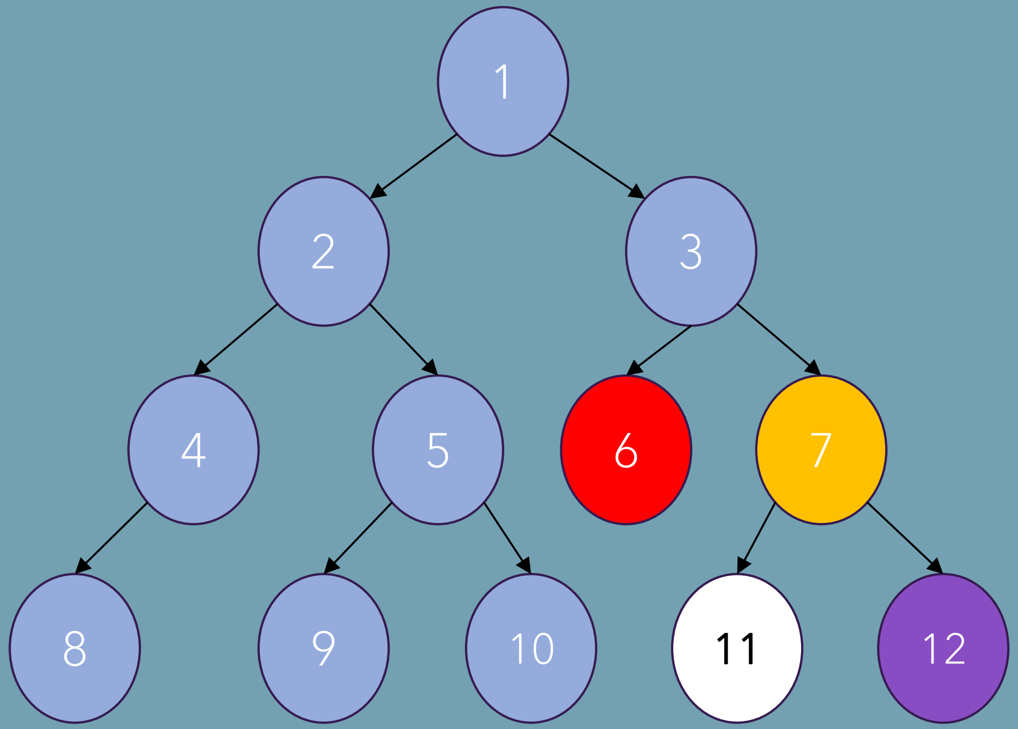
List

7,6

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



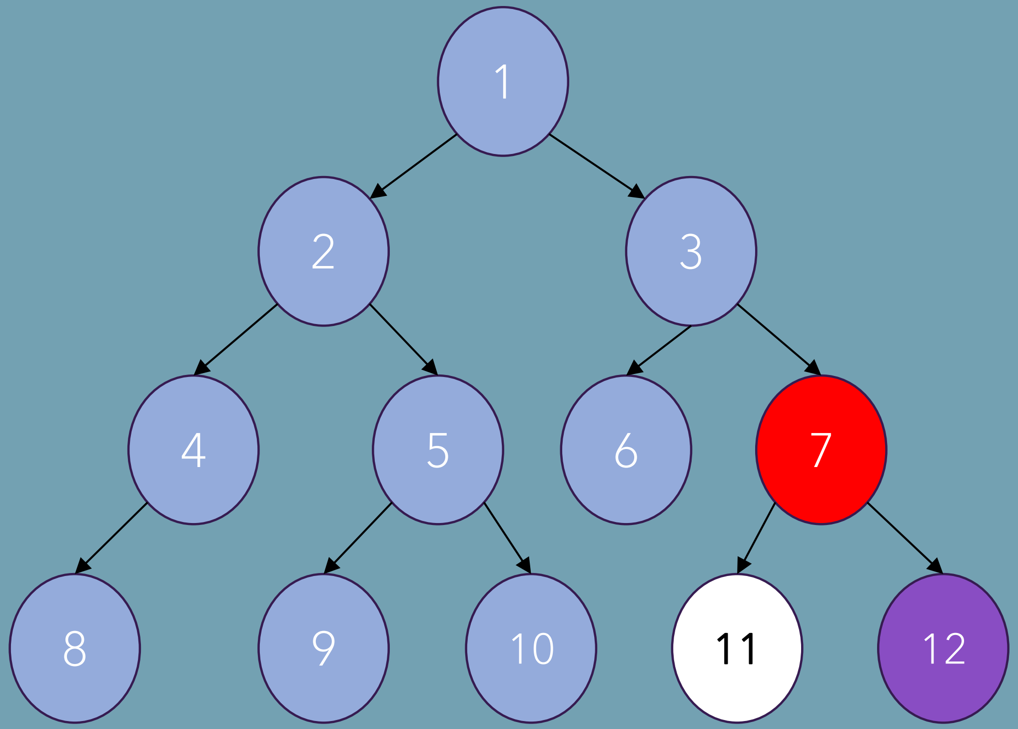
List

7

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11

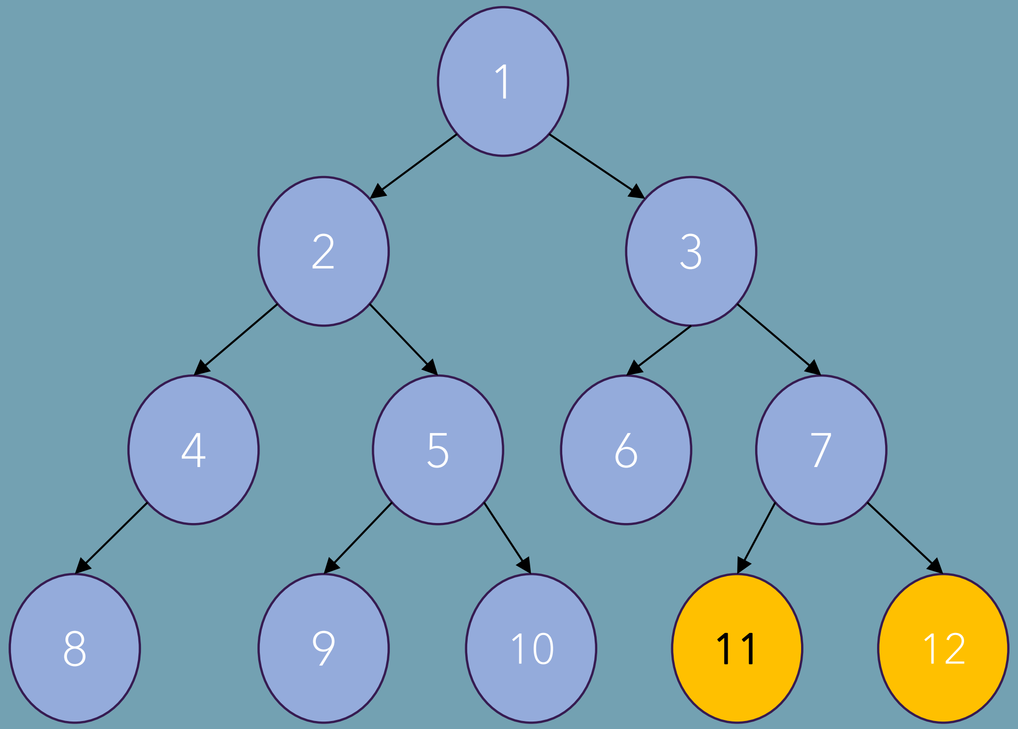


List

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11



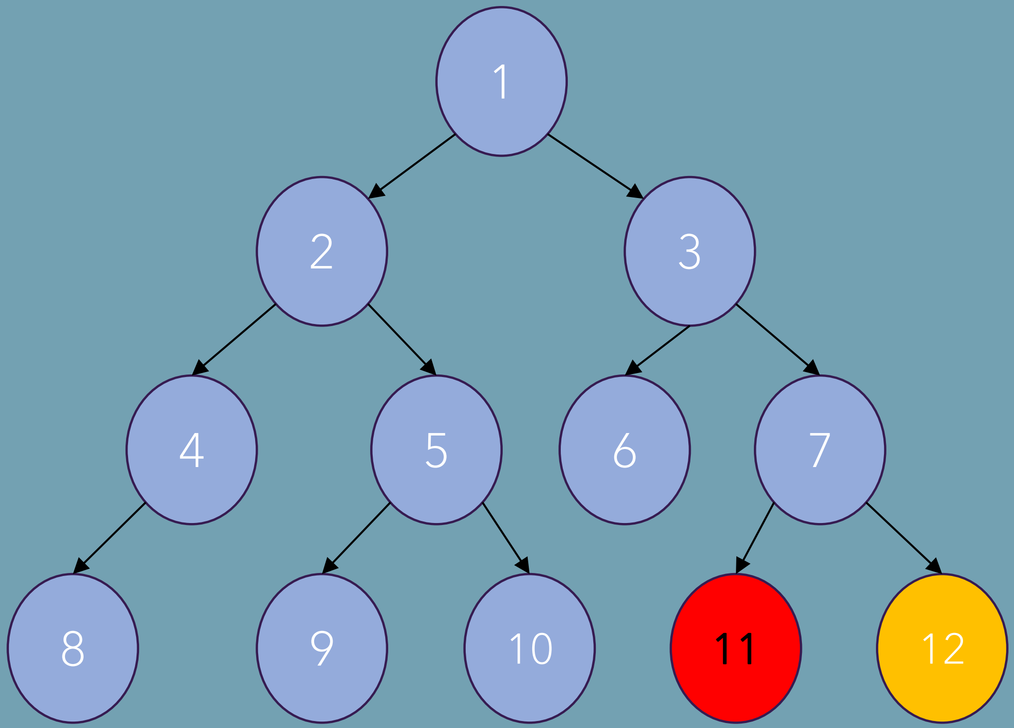
List

12,11

Direction of insert/remove

DEPTH-FIRST SEARCH

Goal: 11

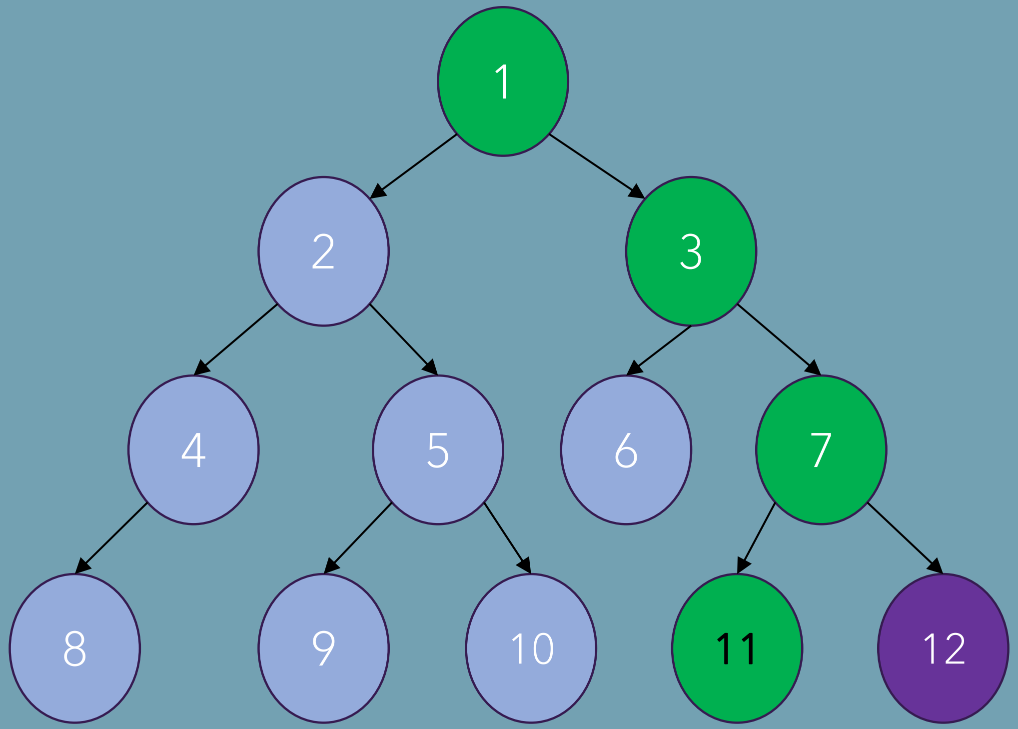


List

12

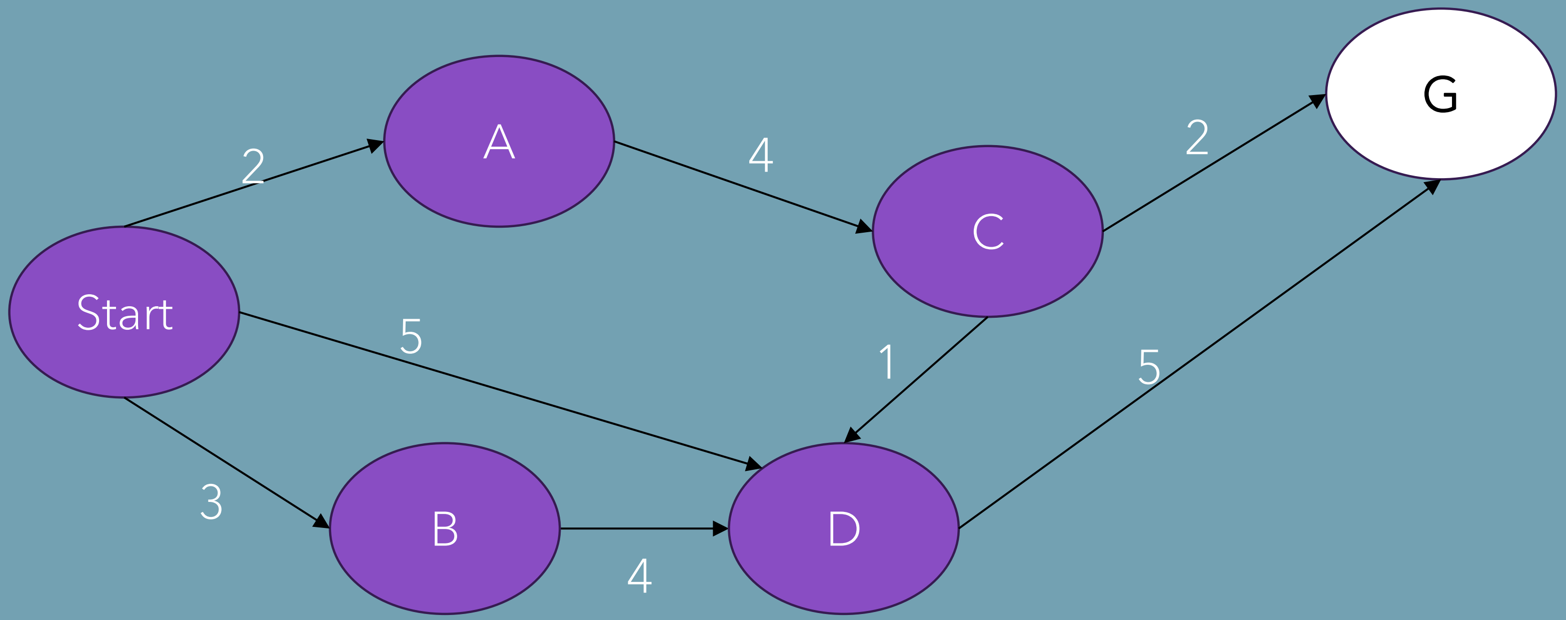
Direction of insert/remove

DEPTH-FIRST SEARCH



Solution: [1, 3, 7, 11] – Visited: [1,2,4,8,5,9,10,3,6,7,11]

DEPTH-FIRST SEARCH (GRAPH)



Current	LIFO - - - > TOP
	[S]
[S]	[S, A] , [S , B] , [S, D]
[S, D]	[S, A], [S , B] , [S, D, G]
[S, D, G]	[S, A] , [S , B]

In graphs, Sort alphabetically each expansion only for unified answer for all students !

BREADTH-FIRST SEARCH (الاتساع بالعرض أولاً)

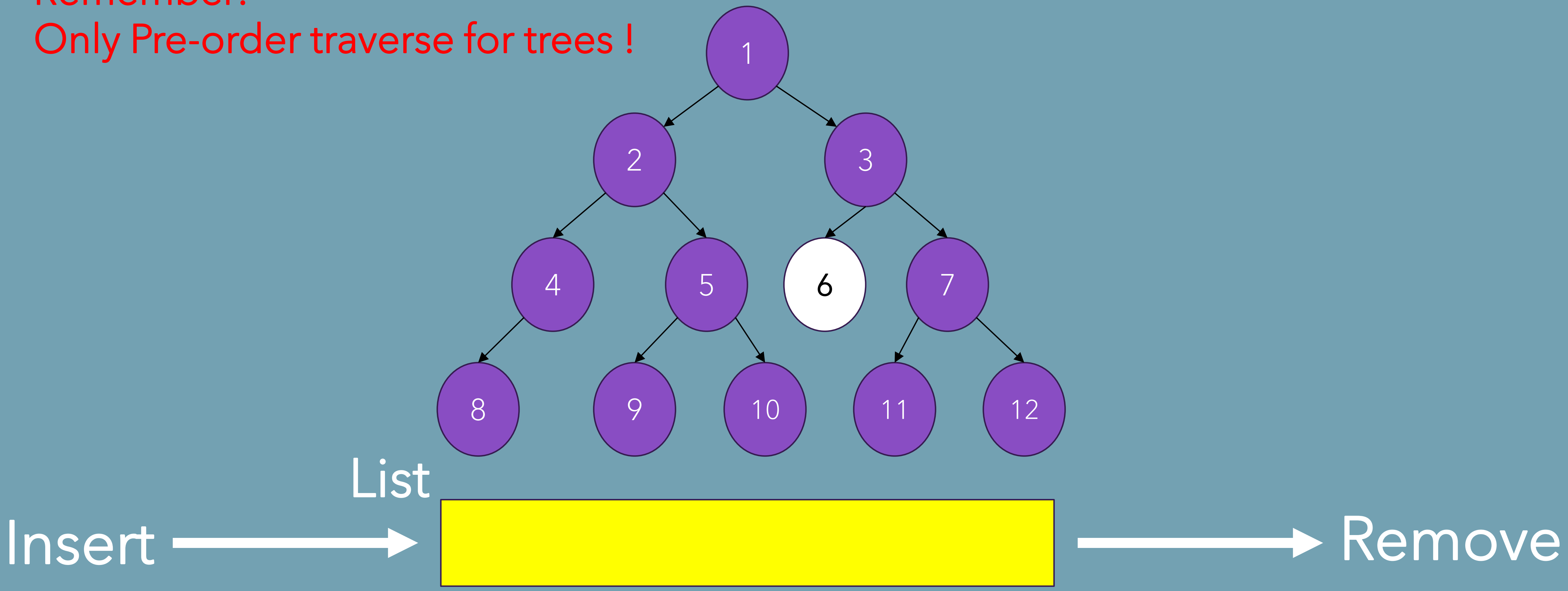
FIFO Function

FIFO (First In, First Out) is a data access principle where the first element added is the first one to be removed.

BREADTH-FIRST SEARCH

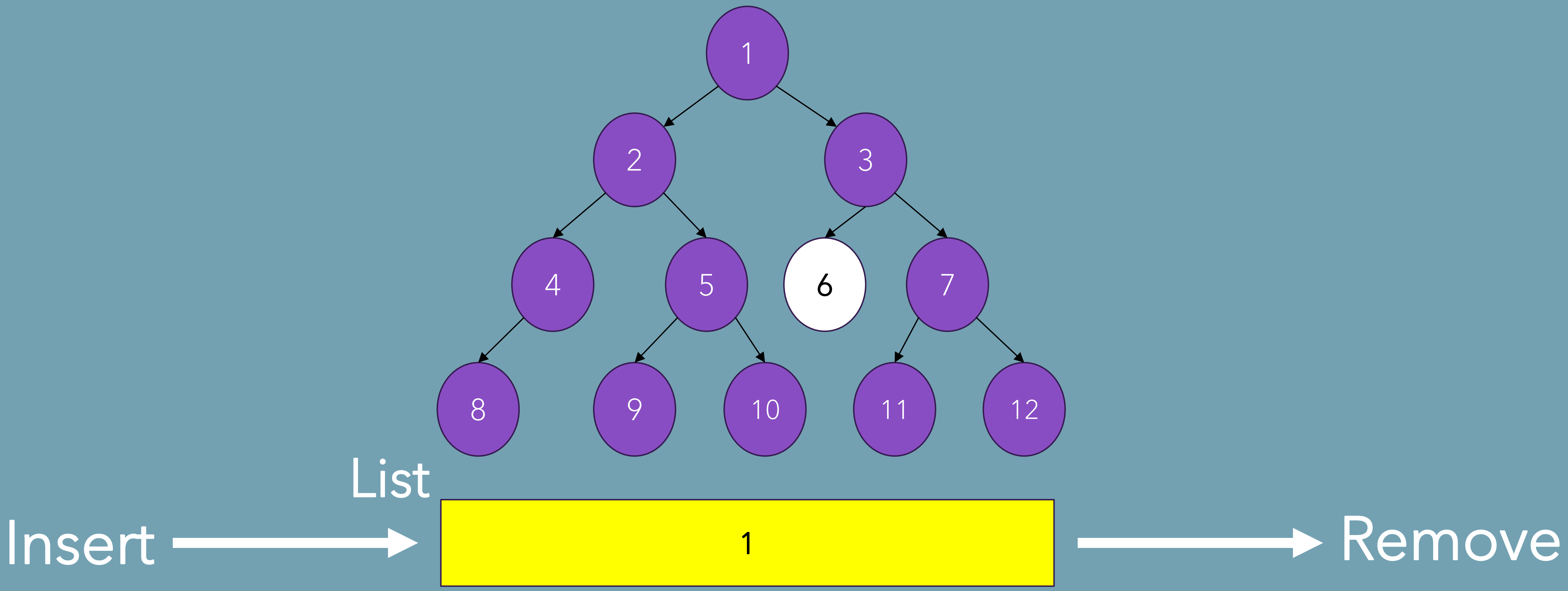
Goal: 6

Remember:
Only Pre-order traverse for trees !



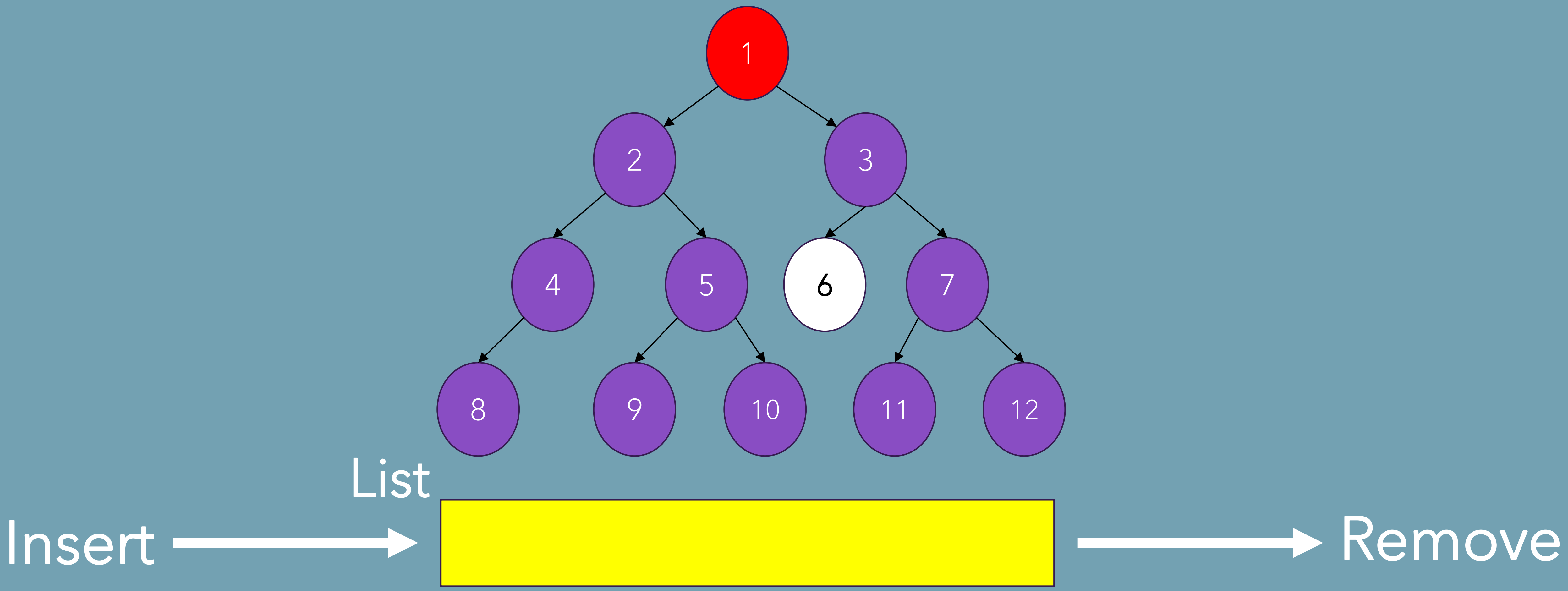
BREADTH-FIRST SEARCH

Goal: 6



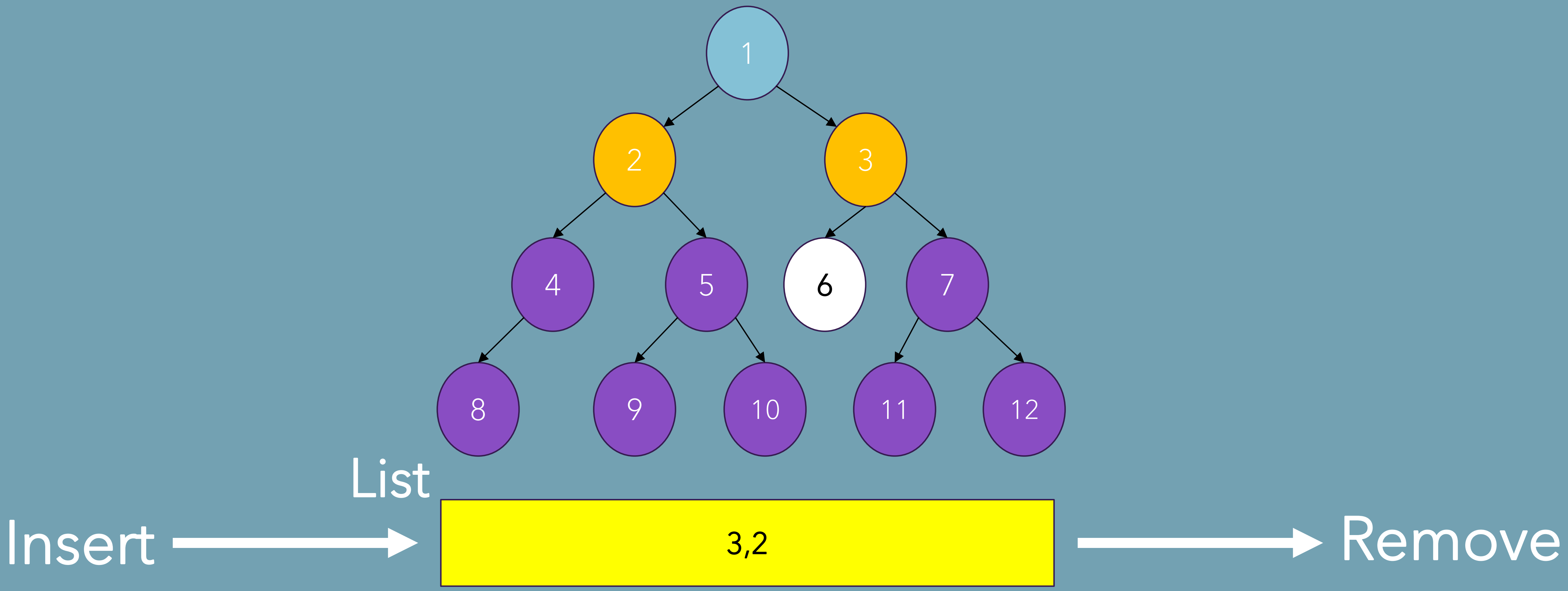
BREADTH-FIRST SEARCH

Goal: 6



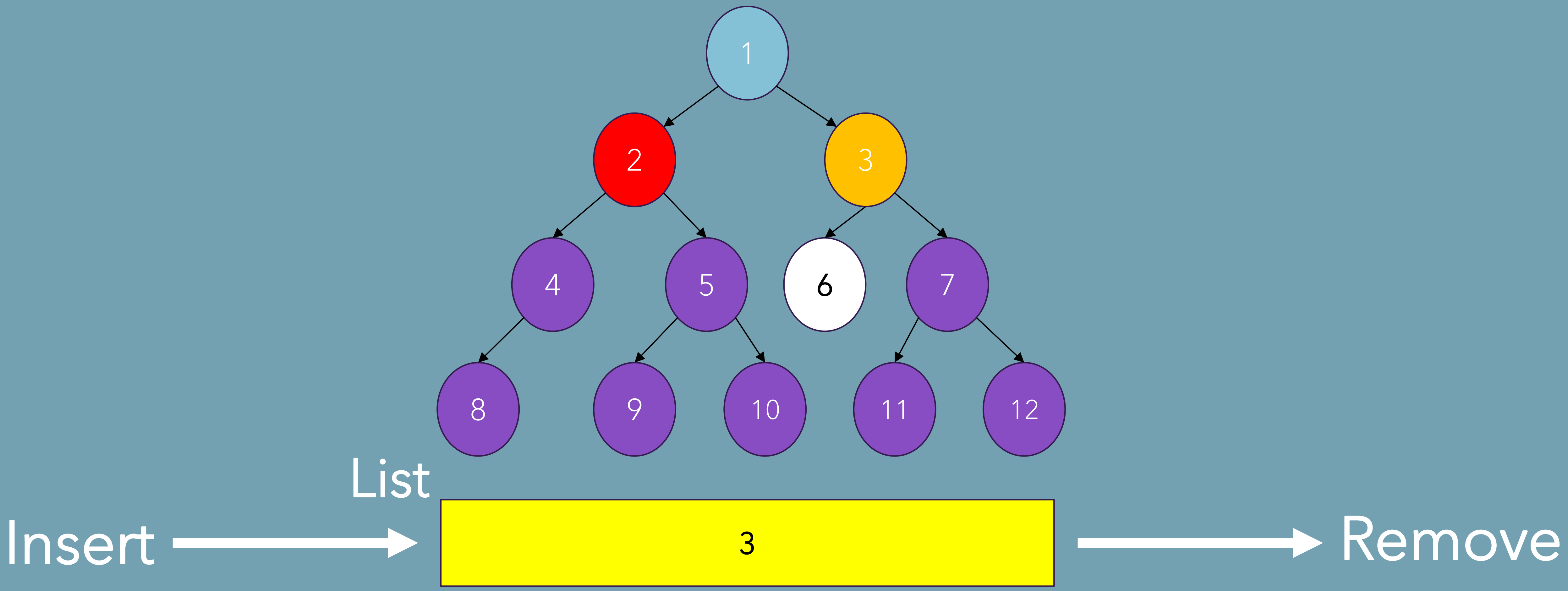
BREADTH-FIRST SEARCH

Goal: 6



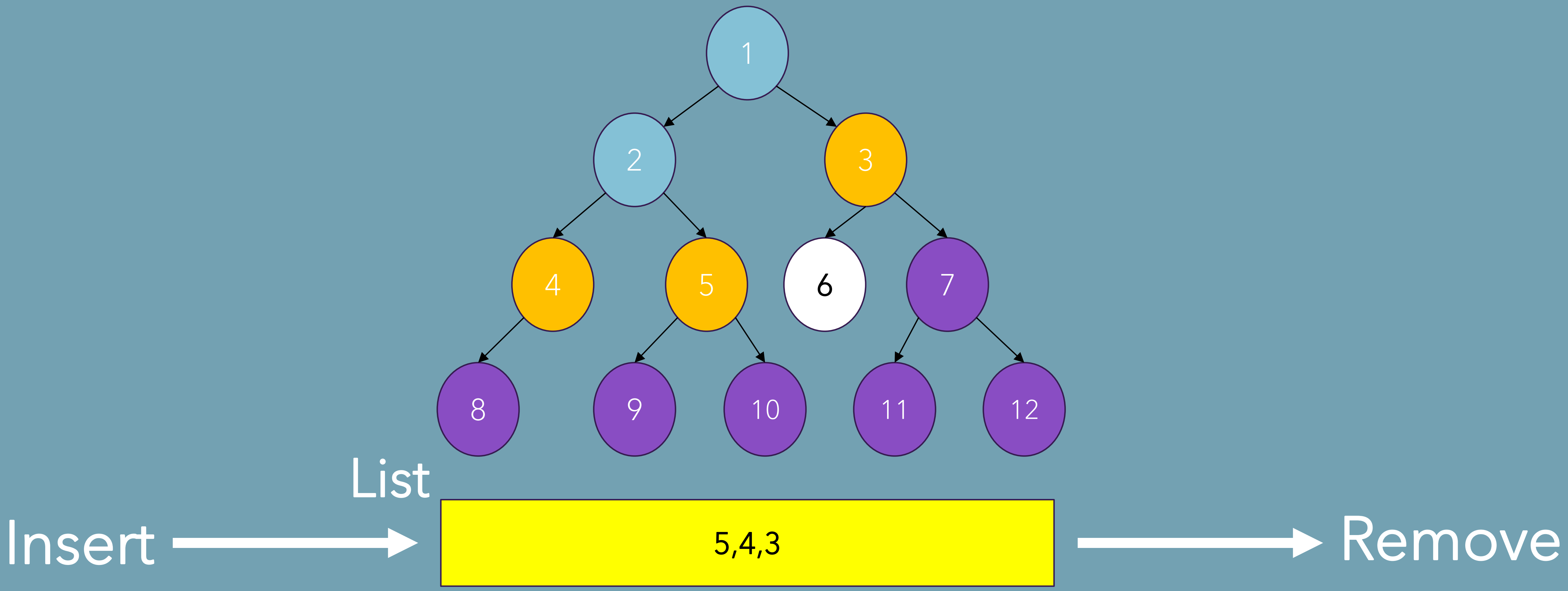
BREADTH-FIRST SEARCH

Goal: 6



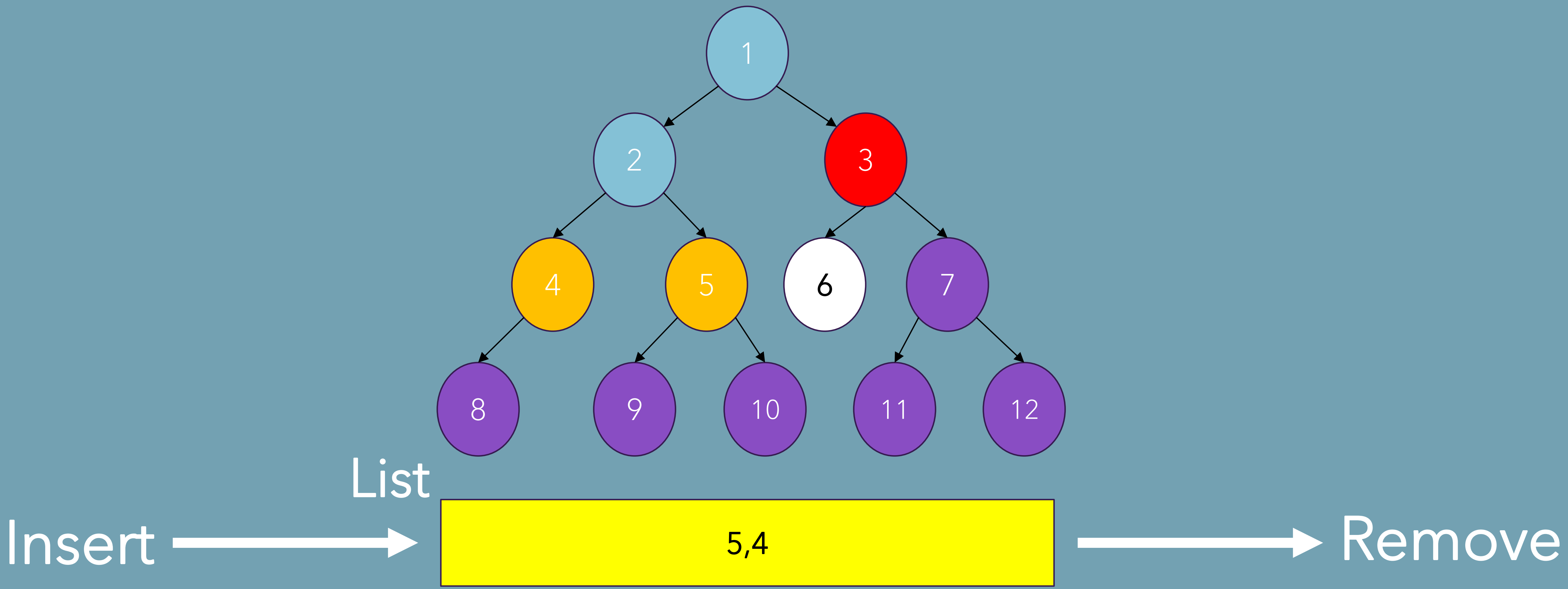
BREADTH-FIRST SEARCH

Goal: 6



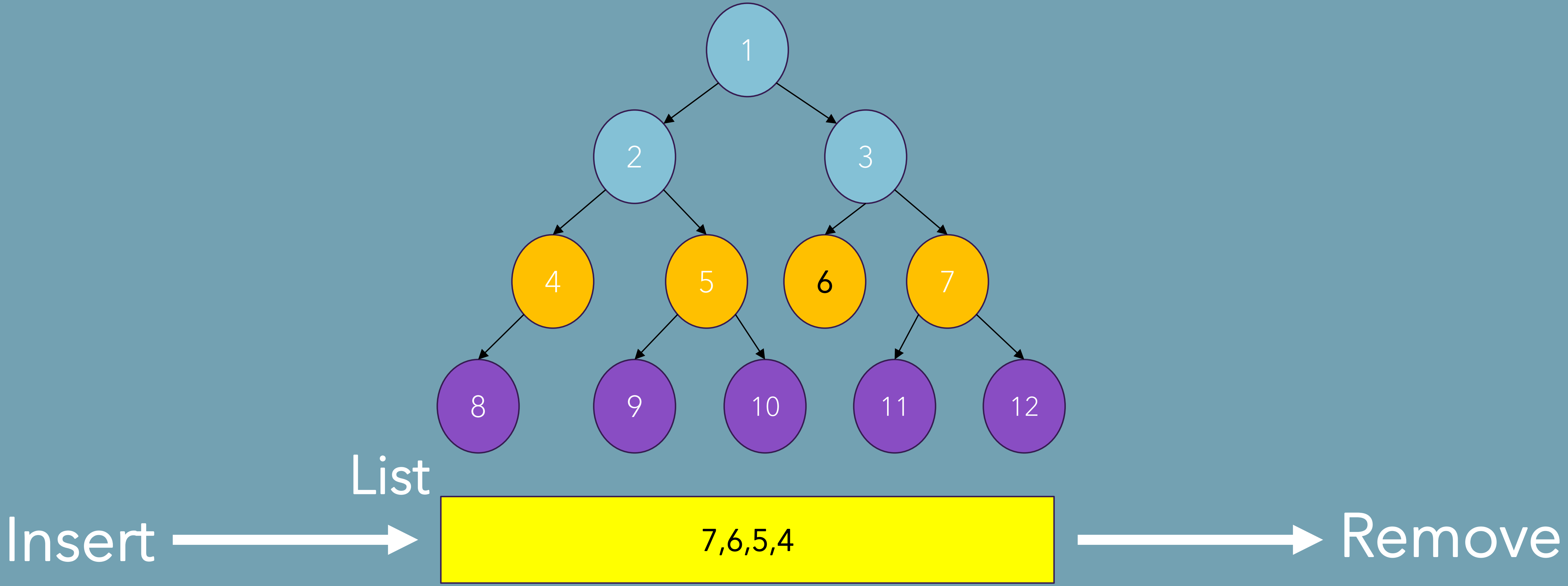
BREADTH-FIRST SEARCH

Goal: 6



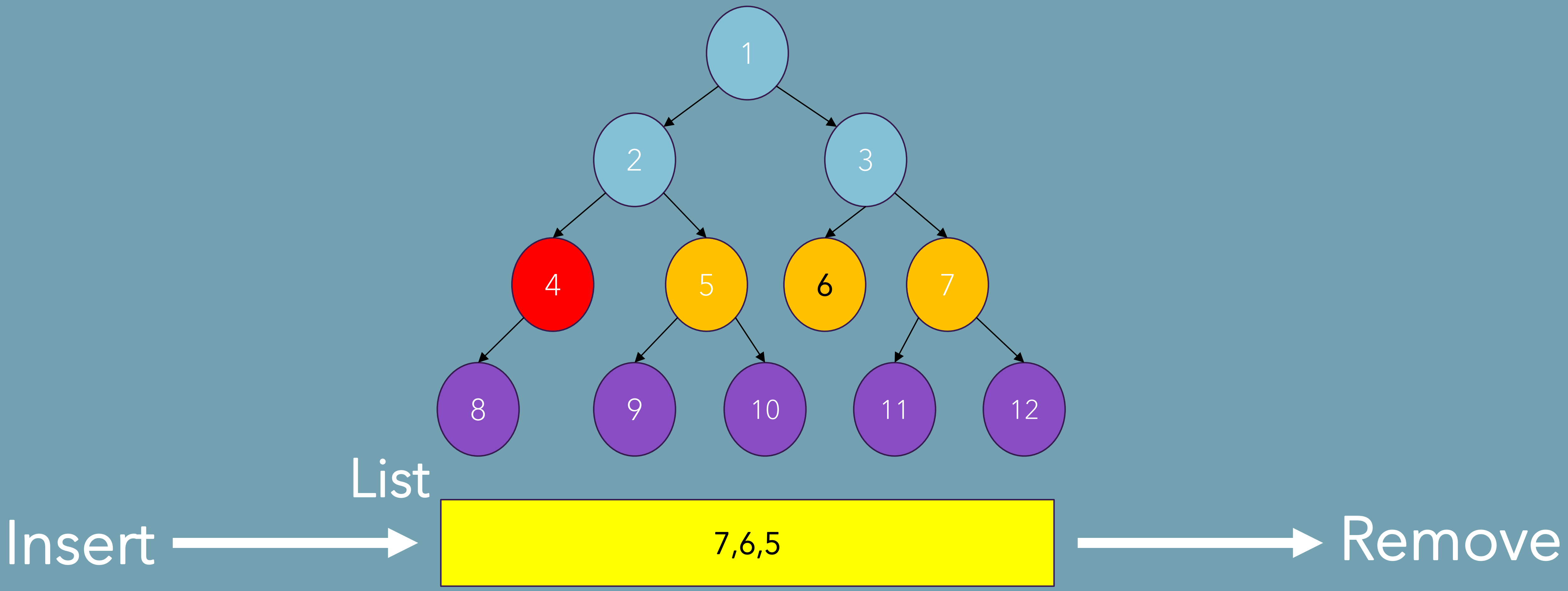
BREADTH-FIRST SEARCH

Goal: 6



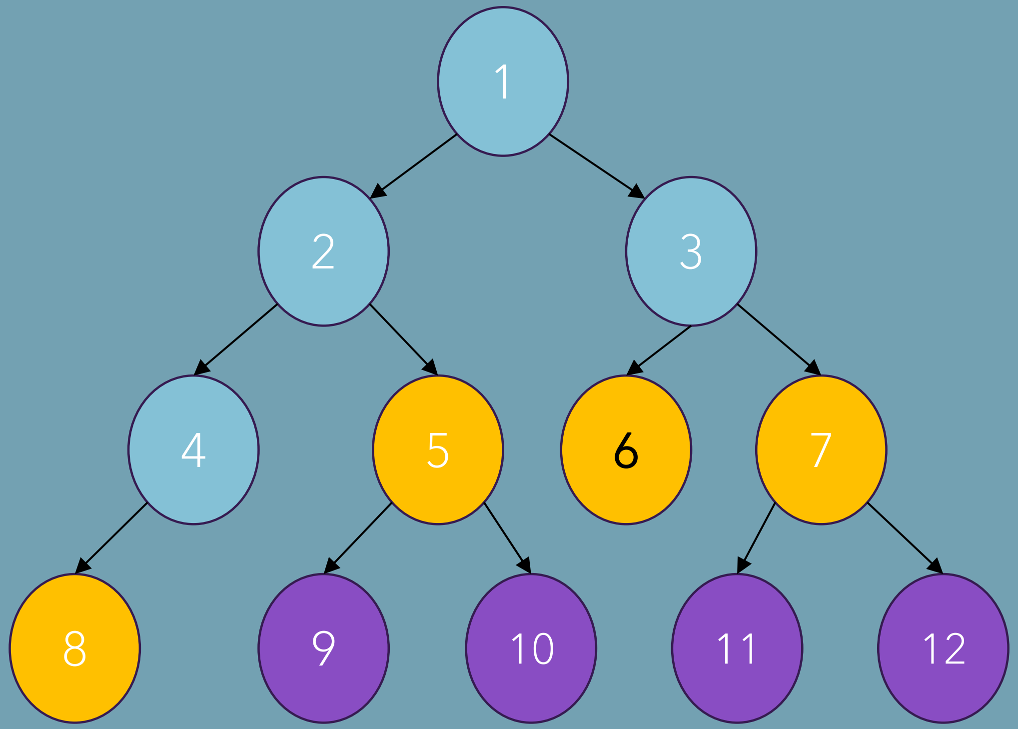
BREADTH-FIRST SEARCH

Goal: 6



BREADTH-FIRST SEARCH

Goal: 6

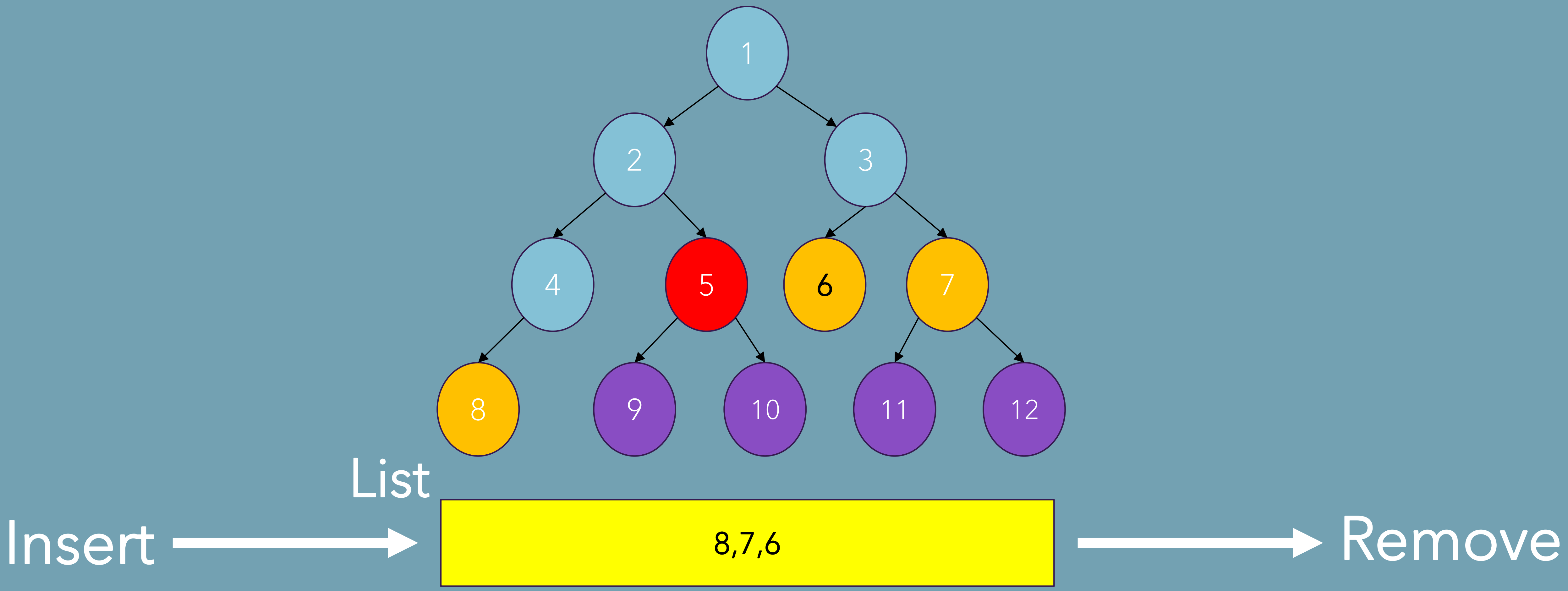


List

Insert → 8,7,6,5 → Remove

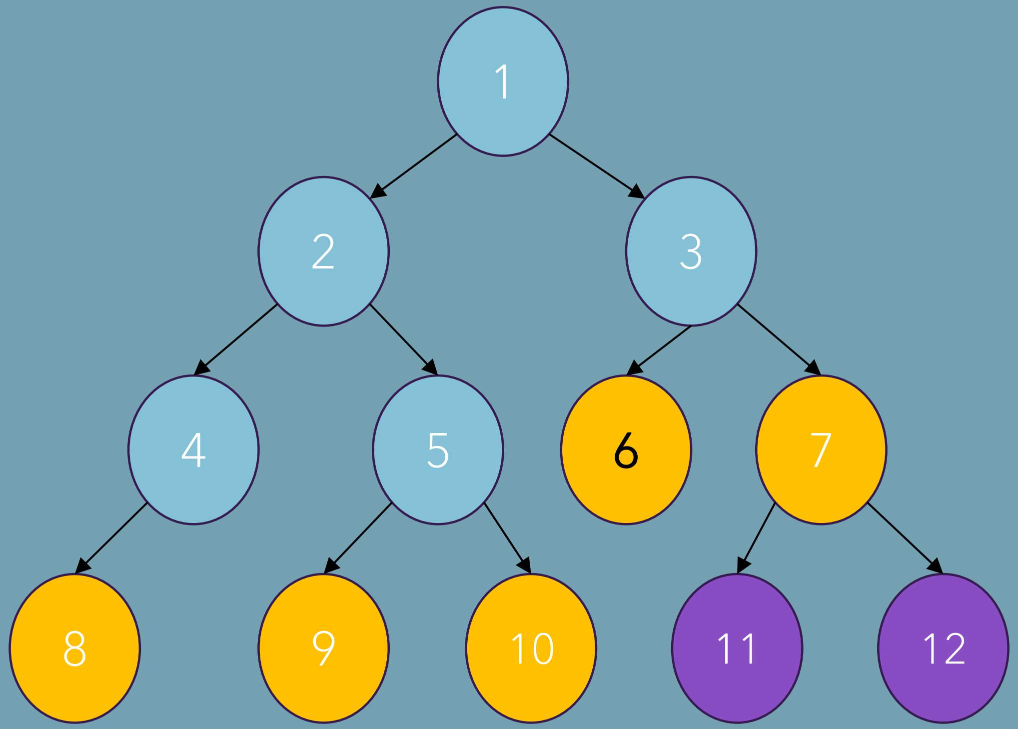
BREADTH-FIRST SEARCH

Goal: 6



BREADTH-FIRST SEARCH

Goal: 6

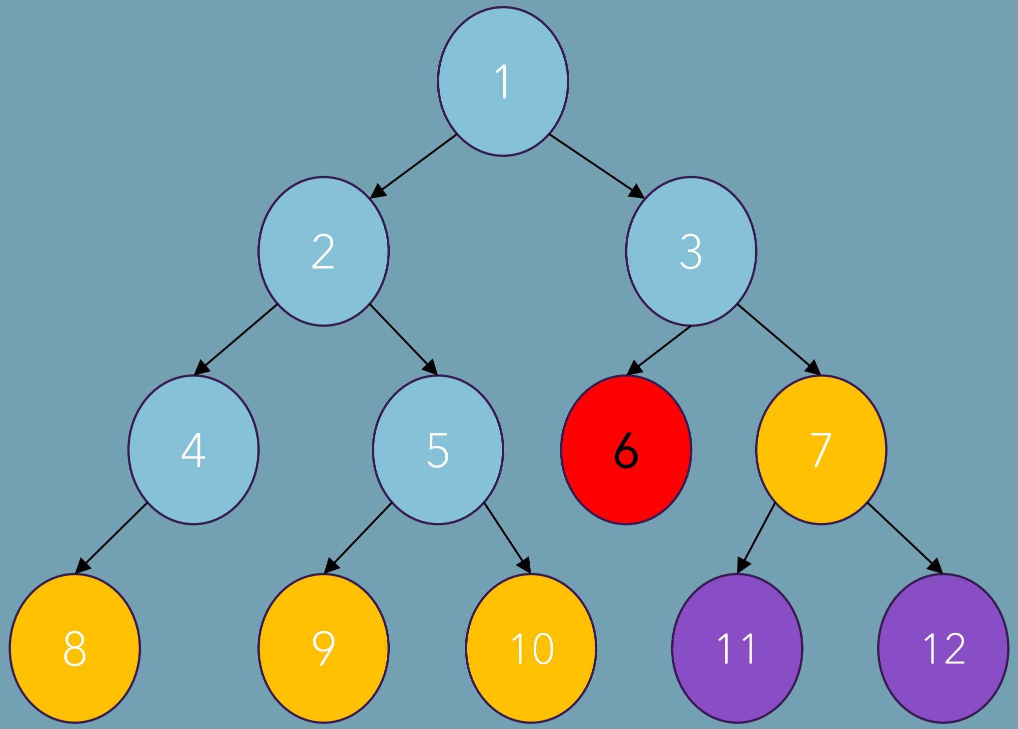


List

Insert → 10,9,8,7,6 → Remove

BREADTH-FIRST SEARCH

Goal: 6

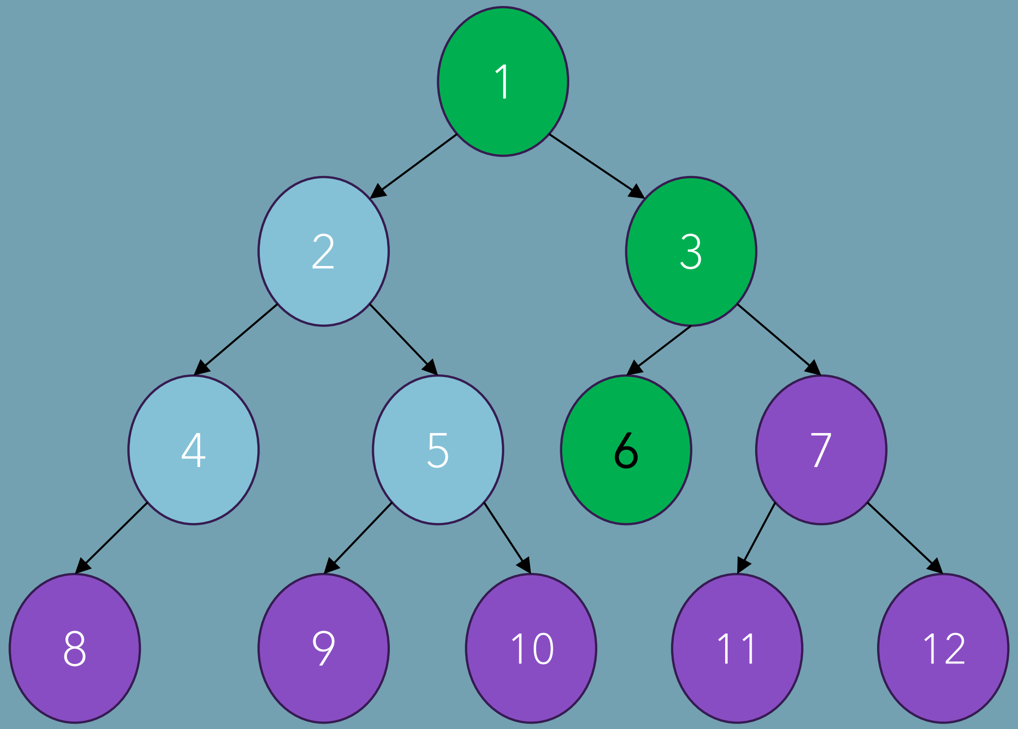


List

Insert → 10,9,8,7 → Remove

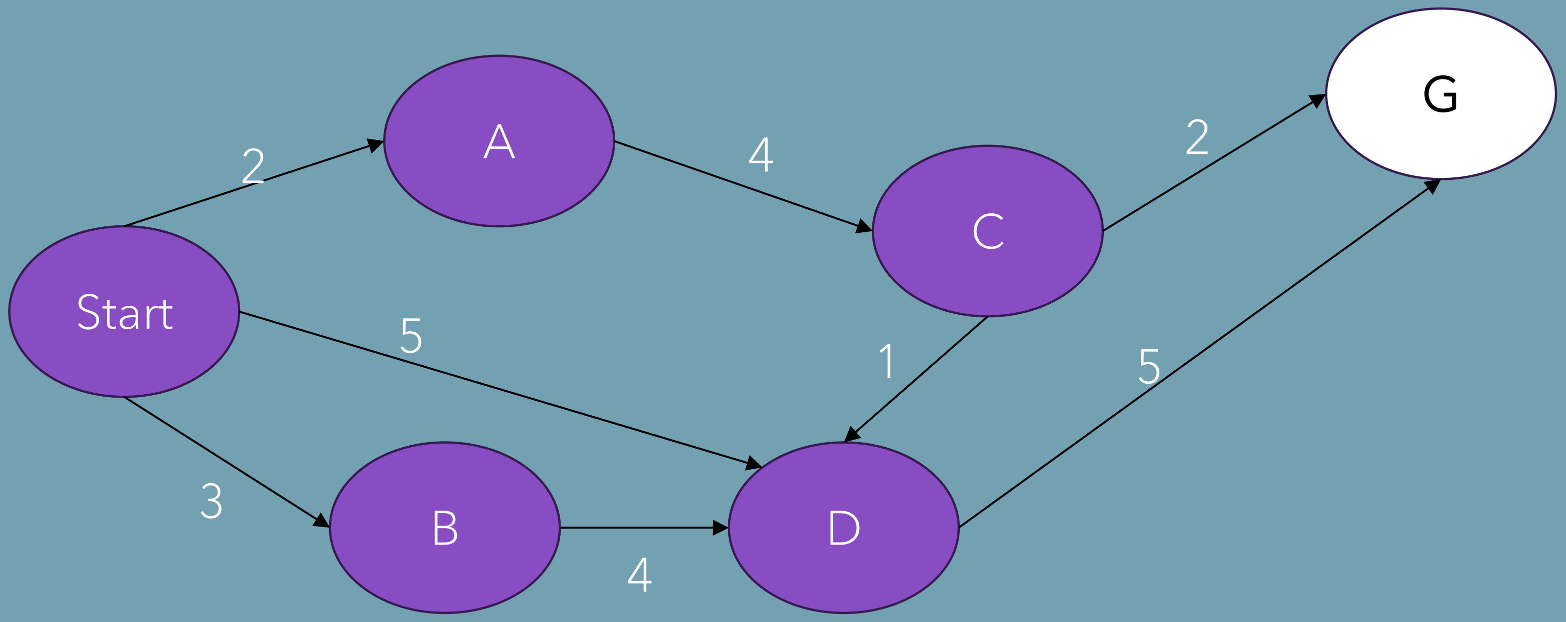
BREADTH-FIRST SEARCH

Goal: 6



Solution: [1, 3, 6] – Visited [1, 2, 3, 4, 5, 6]

BREADTH-FIRST SEARCH (GRAPH)



Current	FIFO FRONT < - - - - - - - > REAR
	[S]
[S]	[S,A] , [S,B] , [S,D]
[S, A]	[S,B] , [S,D], [S,A,C]
[S, B]	[S,D], [S,A,C], [S,B,D]
[S,D]	[S,A,C], [S,B,D], [S,D,G]
[S,A,C]	[S,B,D], [S,D,G], [S,A,C,D], [S,A,C,G]
[S,D,G]	

In graphs, Sort alphabetically each expansion only for unified answer for all students !

DFS VS. BFS

DFS

Current	LIFO - - - > TOP
	[S]
[S]	[S,A] , [S,B] , [S,D]
[S, D]	[S,A] , [S,B] , [S,D, G]
[S, D, G]	Actual Cost: 10 Sol. Steps: 3

BFS

Current	FIFO FRONT < - - - - - - - > REAR
	[S]
[S]	[S,A] , [S,B] , [S,D]
[S, A]	[S,B] , [S,D], [S,A,C]
[S, B]	[S,D], [S,A,C], [S,B,D]
[S,D]	[S,A,C], [S,B,D], [S,D,G]
[S,A,C]	[S,B,D], [S,D,G], [S,A,C,D], [S,A,C,G]
[S,D,G]	Actual Cost: 10 Sol. Steps: 6

IMPLEMENTATION

GRAPH CLASS (ADJACENCY LIST REP.)

```
# Make a Graph class
class Graph:
    def __init__(self, root=None, weighted=None, directed=None):
        # Adjacency list representation
        if root:
            self.__graph = {root:[]}
        else:
            self.__graph = {"root":[]}

        self.__weighted = weighted
        if weighted == True:
            self.__weighted = weighted
        elif self.__weighted != None:
            raise Exception("weighted parameter should be True or None")

        self.__directed = False
        if directed == True:
            self.__directed = directed
        elif self.__directed == None:
            raise Exception("directed parameter should be True or None")

        print(f"Graph is created with root name: {list(self.__graph.keys())[0]}")
```

GRAPH CLASS (ADD VERTICES METHOD)

```
def add_vertex(self, vertex):
    if vertex not in self.__graph.keys():
        self.__graph[vertex] = []
        print("Vertex is added successfully !")
    else:
        print("This vertex does exist in the graph already !")
```

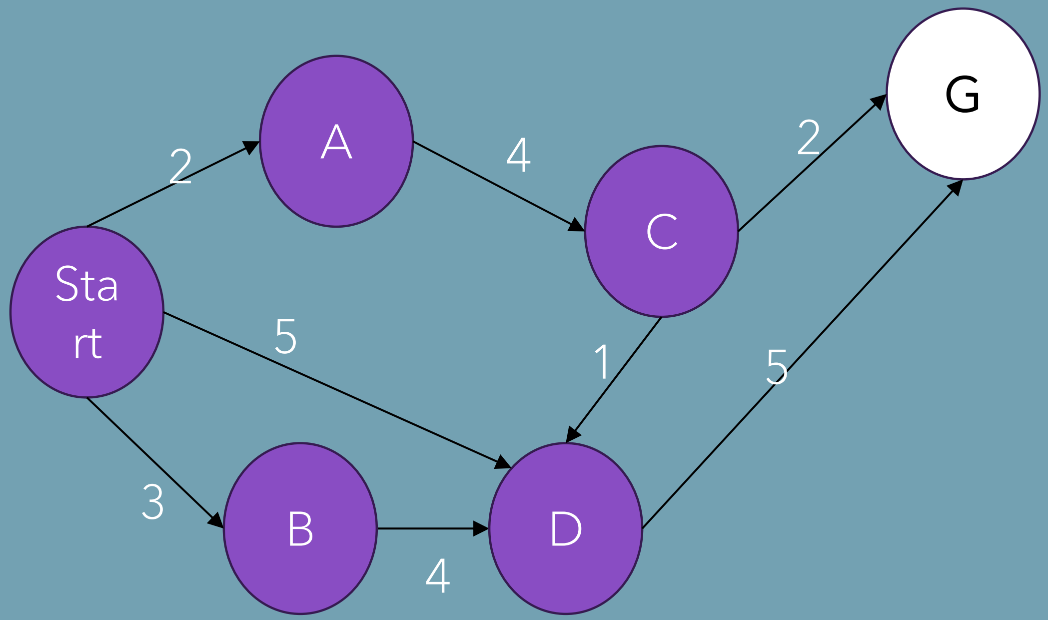
GRAPH CLASS (ADD EDGES METHOD)

```
def add_edge(self, vertex_1, vertex_2, weight=None):
    if vertex_1 in self.__graph.keys() and vertex_2 in self.__graph.keys():
        if not self.__directed:
            if not self.__weighted:
                self.__graph[vertex_1].append(vertex_2)
                self.__graph[vertex_2].append(vertex_1)
            else:
                if weight != None:
                    self.__graph[vertex_1].append((vertex_2, weight))
                    self.__graph[vertex_2].append((vertex_1, weight))
                else:
                    raise Exception("It should be a weight for the edge")
        else:
            if not self.__weighted:
                self.__graph[vertex_1].append(vertex_2)
            else:
                if weight != None:
                    self.__graph[vertex_1].append((vertex_2, weight))
                else:
                    raise Exception("It should be a weight for the edge")
    else:
        raise Exception("It should be both vertices exist in the graph !")

    print(f"Edge added between {vertex_1} and {vertex_2}")
```

GRAPH CLASS (GET GRAPH METHOD)

```
def get_graph(self):
    return self.__graph
```

FORMULATE THE PROBLEM



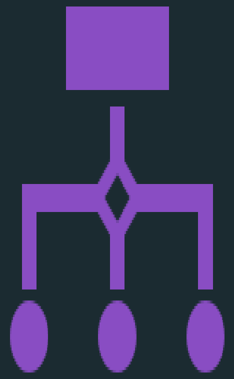

```python
my_graph = Graph("S", directed=True, weighted=True)
my_graph.add_vertex("A")
my_graph.add_vertex("B")
my_graph.add_vertex("C")
my_graph.add_vertex("D")
my_graph.add_vertex("G")
my_graph.add_edge("S","B", 3)
my_graph.add_edge("S","A", 2)
my_graph.add_edge("S","D", 5)
my_graph.add_edge("A","C", 4)
my_graph.add_edge("C","G", 2)
my_graph.add_edge("C","D", 1)
my_graph.add_edge("B","D", 4)
my_graph.add_edge("D","G", 5)
print(my_graph.get_graph())
```

```
Graph is created with root name: S
Vertex is added successfully !
Vertex is added successfully !
Vertex is added successfully !
Vertex is added successfully !
Vertex is added successfully !
Edge added between S and B
Edge added between S and A
Edge added between S and D
Edge added between A and C
Edge added between C and G
Edge added between C and D
Edge added between B and D
Edge added between D and G
{'S': [('B', 3), ('A', 2), ('D', 5)], 'A': [('C', 4)], 'B': [('D', 4)], 'C': [('G', 2), ('D', 1)], 'D': [('G', 5)], 'G': []}
```

GRAPH CLASS (DFS METHOD)

```python
def DFS(self, goal, start=None):
    if start != None:
        lst = [[start]]
    else:
        lst = [["root"]]

    visited = []
    while lst:
        # basic search
        current_path = lst.pop() # LIFO
        current_node = current_path[-1]

        if current_node in visited:
            continue
        visited.append(current_node)

        if current_node == goal:
            return current_path, visited
        else:
            # Sort all elements alphabetically to get a unified answer !
            # Sorting in ascending order to follow "LIFO" principle !
            adjacent_nodes = self.__graph[current_node]
            adjacent_nodes.sort()
            if self.__weighted:
                for node, _ in adjacent_nodes:
                    new_path = current_path.copy()
                    new_path.append(node)
                    lst.append(new_path)
            else:
                for node in adjacent_nodes:
                    new_path = current_path.copy()
                    new_path.append(node)
                    lst.append(new_path)

    raise Exception("Search Failed !")
```

BASIC SEARCH

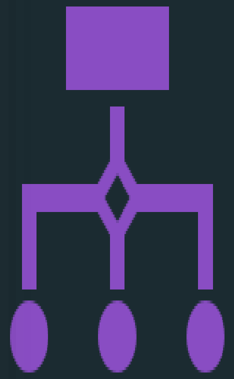


1. Initialize an **empty list** and put the **initial state** in it
2. Take the first state in the **list as the current if it is not visited yet otherwise skip it**, and remove it from the list
3. **Check if the current is the goal state**, if it is, then terminate the search and **return the solution path**
4. Otherwise, expand the current of its successors, **and add them into the list using a queuing function**
5. Repeat from (2) to (4)
6. If the current becomes empty, then there is no solution and **return "fail"**

GRAPH CLASS (BFS METHOD)

```python
def BFS(self, goal, start=None):
    if start != None:
        lst = [[start]]
    else:
        lst = [["root"]]

    visited = []
    while lst:
        # basic search
        current_path = lst.pop(0) # FIFO
        current_node = current_path[-1]

        if current_node in visited:
            continue
        visited.append(current_node)

        if current_node == goal:
            return current_path, visited
        else:
            # Sort all elements alphabetically to get a unified answer !
            # Sorting in ascending order to follow "FIFO" principle !
            adjacent_nodes = self.__graph[current_node]
            adjacent_nodes.sort()
            if self.__weighted:
                for node, _ in adjacent_nodes:
                    new_path = current_path.copy()
                    new_path.append(node)
                    lst.append(new_path)
            else:
                for node in adjacent_nodes:
                    new_path = current_path.copy()
                    new_path.append(node)
                    lst.append(new_path)

    raise Exception("Search Failed !")
```

BASIC SEARCH



1. Initialize an **empty list** and put the **initial state** in it
2. Take the first state in the **list as the current if <u>it is not visited yet</u> otherwise skip it**, and remove it from the list
3. **Check if the current is the goal state**, if it is, then terminate the search and **return the solution path**
4. Otherwise, expand the current of its successors, **and add them into the list using a queuing function**
5. Repeat from (2) to (4)
6. If the current becomes empty, then there is no solution and **return "fail"**

TESTING

```
[3]: print(my_graph.DFS('G', start='S'))
     print(my_graph.BFS('G', start='S'))

     (['S', 'D', 'G'], ['S', 'D', 'G'])
     (['S', 'D', 'G'], ['S', 'A', 'B', 'D', 'C', 'G'])
```

DFS

Current	LIFO ---> TOP
	[S]
[S]	[S,A] , [S,B] , [S,D]
[S, D]	[S,A] , [S,B] , [S,D, G]
[S, D, G]	Actual Cost: 10 Sol. Steps: 3

BFS

Current	FIFO FRONT <------> REAR
	[S]
[S]	[S,A] , [S,B] , [S,D]
[S, A]	[S,B] , [S,D], [S,A,C]
[S, B]	[S,D], [S,A,C], [S,B,D]
[S,D]	[S,A,C], [S,B,D], [S,D,G]
[S,A,C]	[S,B,D], [S,D,G], [S,A,C,D], [S,A,C,G]
[S,D,G]	Actual Cost: 10 Sol. Steps: 6

NEXT LAB: SEARCHING PROBLEMS (2)

Thank you