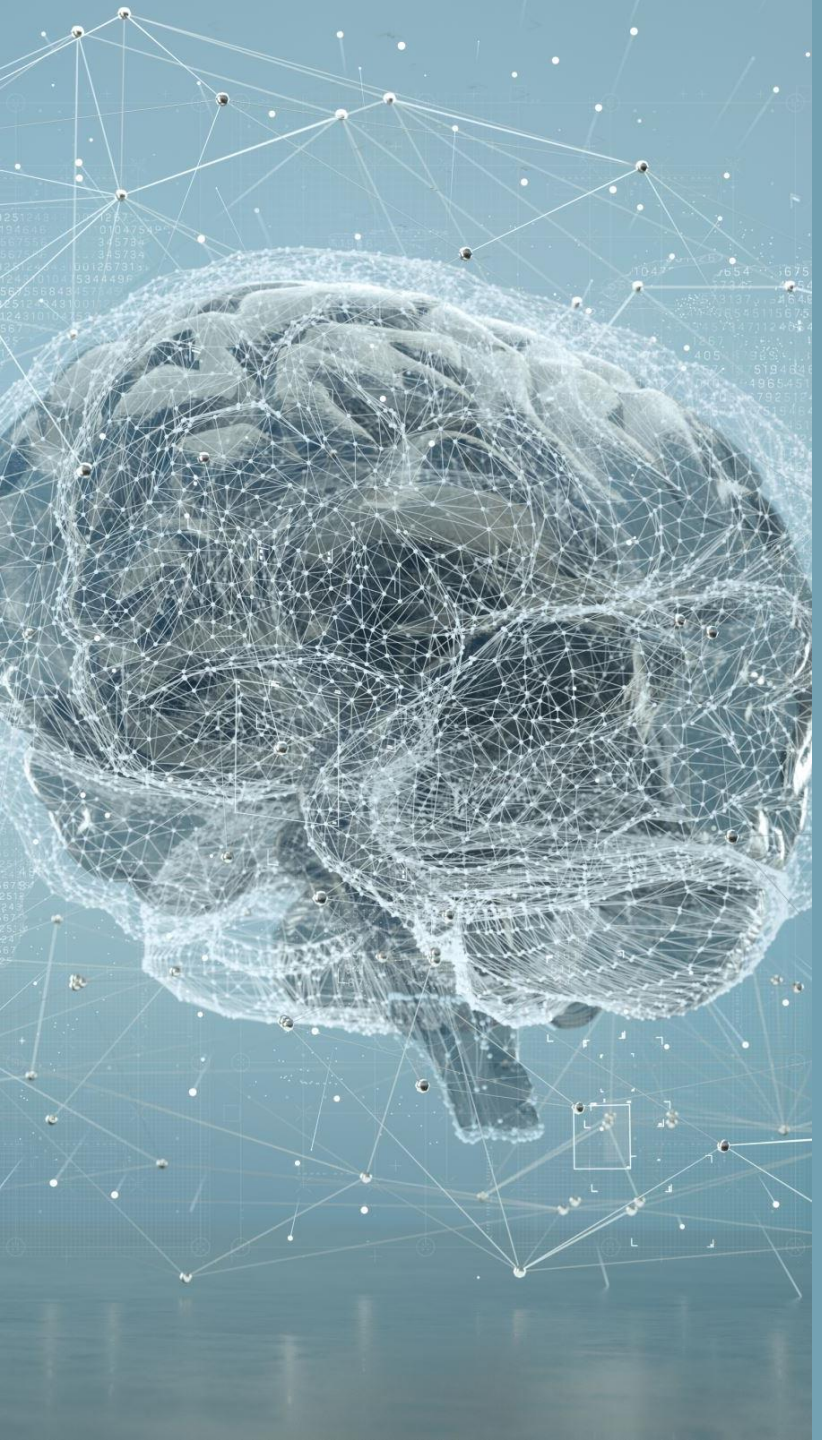


كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE
LAB 04 – SEARCHING PROBLEMS (2)

AI Course Team

Eng. Yousef Elbaroudy – Eng Sahar Mohammed

Eng. Reham Ibrahim – Eng. Mawada Ashraf

Eng. Eman Nasser

2025/2026

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 !

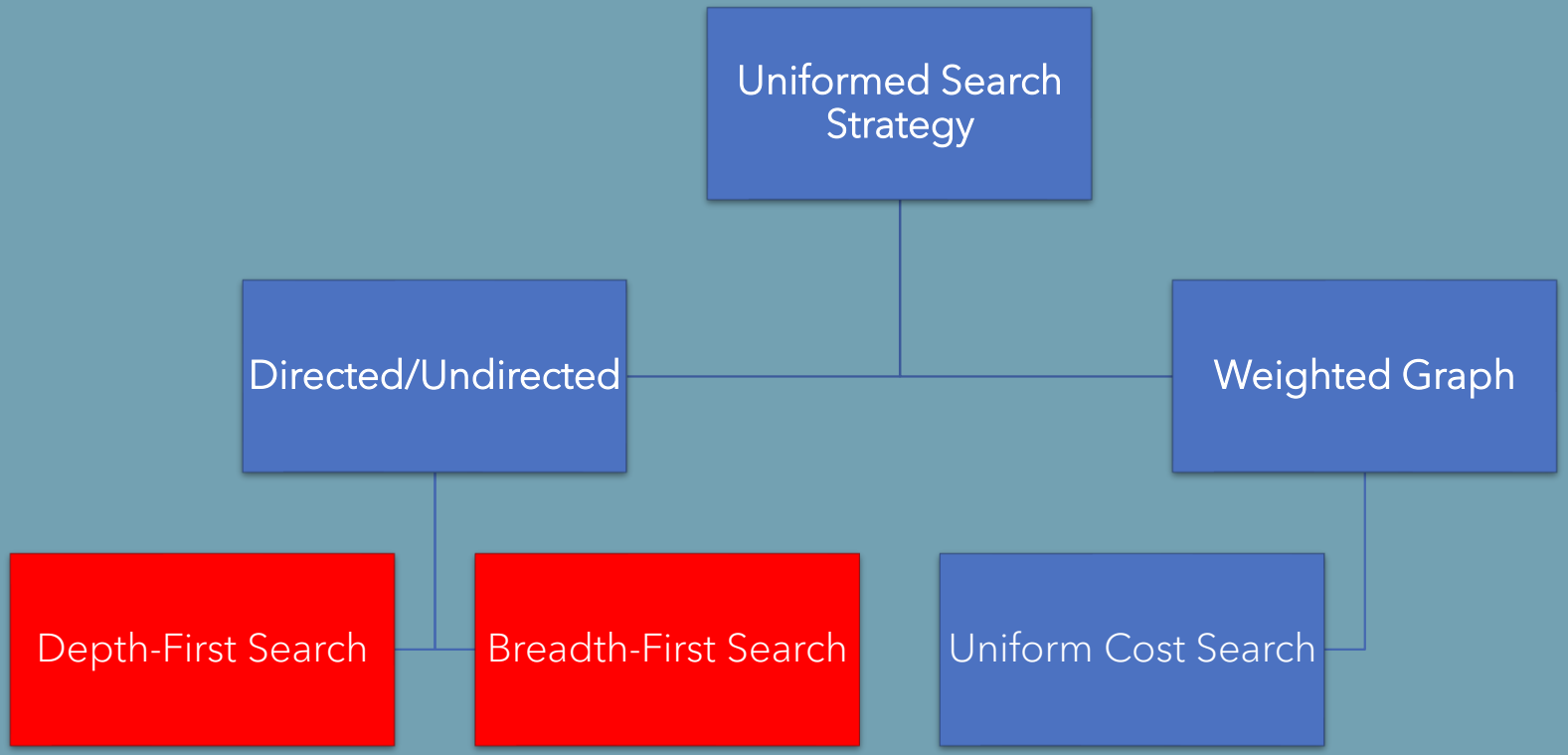
In our course, we are targeting

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

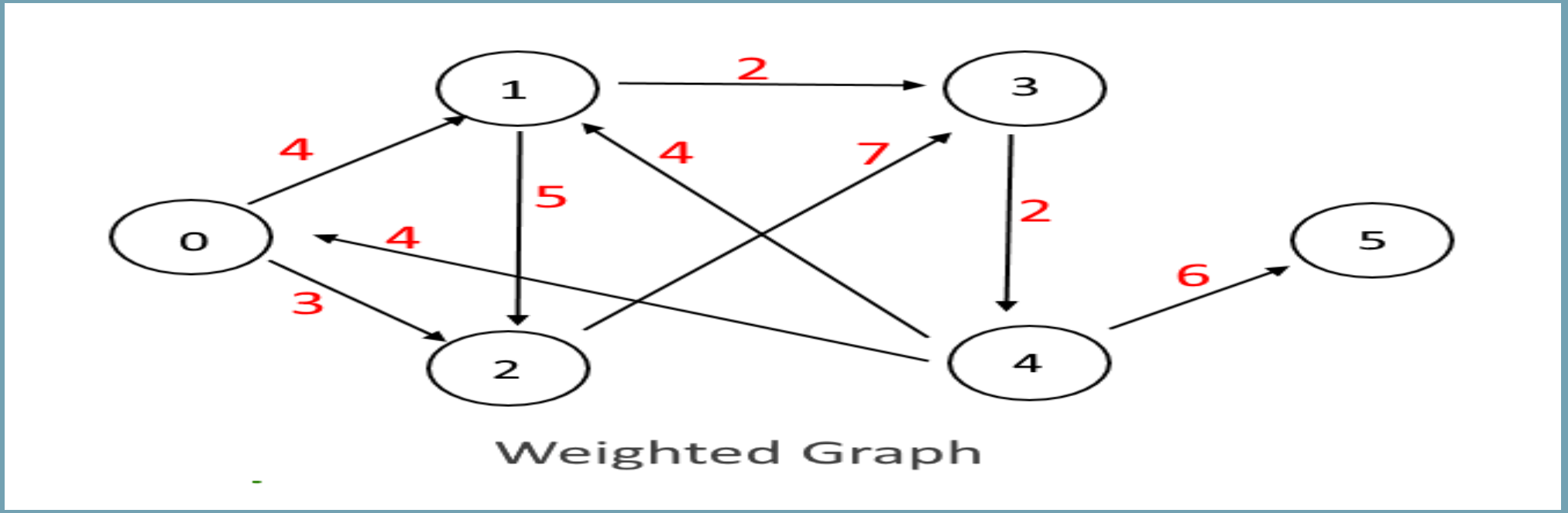
UNIFORMED SEARCH

- **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.



ADDITIONAL: PATH COST (REVISION)

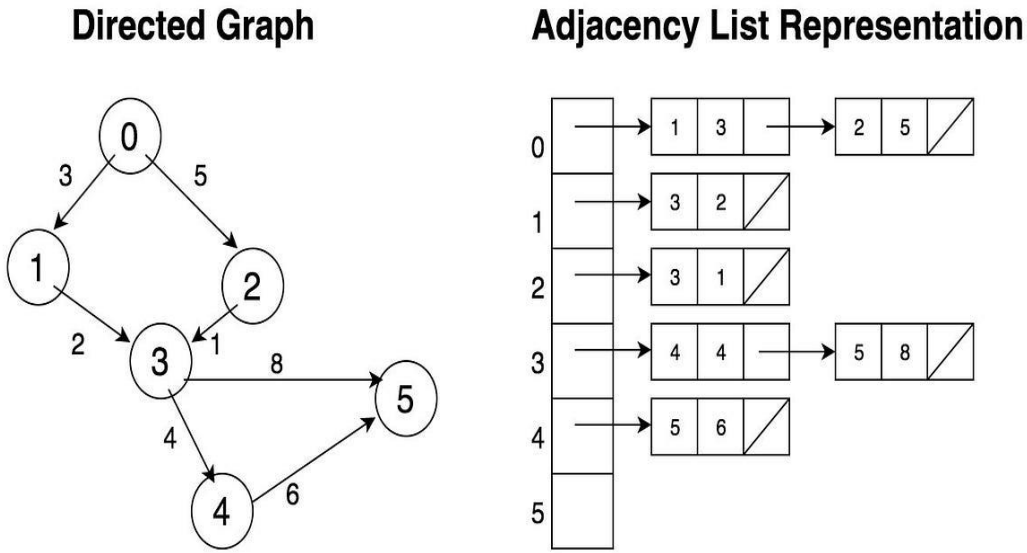
- **Path Cost** refers to the total weight or cost of a path between two nodes in a graph.
- **For the solution path**, the total cost is the **sum of edge weights** along a specific path.



GRAPH REPRESENTATIONS

WEIGHTED GRAPH DIR/UNDIR

The only difference between Directed/Undirected graphs is the Vice-Versa rule of representation

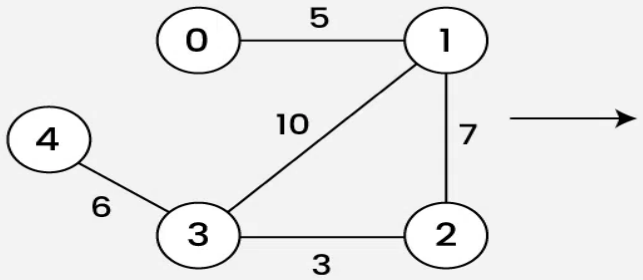


Adjacency List Method

www.kodefork.com

Adjacency Matrix Method

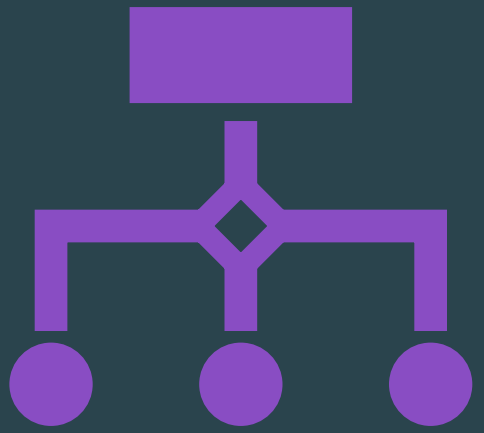
Adjacency Matrix for Undirected and Weighted graph



	0	1	2	3	4
0	INF	5	INF	INF	INF
1	5	INF	7	10	INF
2	INF	7	INF	3	INF
3	INF	10	3	INF	6
4	INF	INF	INF	6	INF

Adjacency Matrix A[ ]

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"**

UNIFORM-COST SEARCH

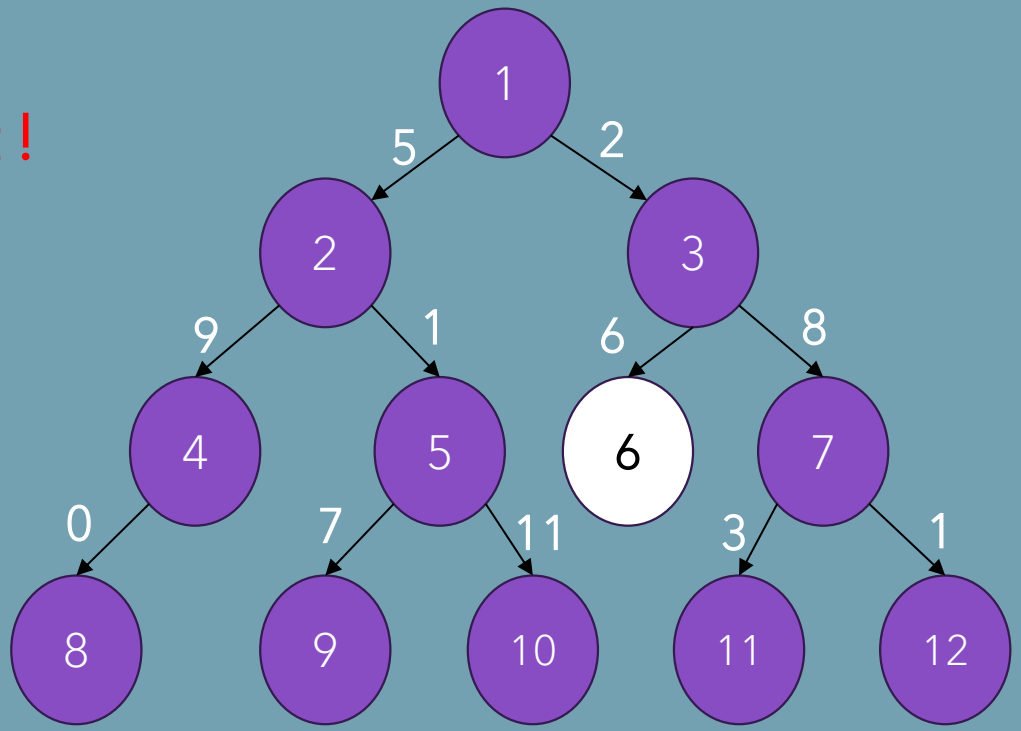
FIFO + Priority Function

Priority function is a data access principle that gives a priority for each element using FIFO, which the highest priority removed first. In UCS, the priority for the lowest cost **(Sort)**, and the cost of each path is <u>accumulative</u>

UNIFORM-COST SEARCH

Goal: 6

Remember:
Pre-order traverse !
Priority for lowest cost !

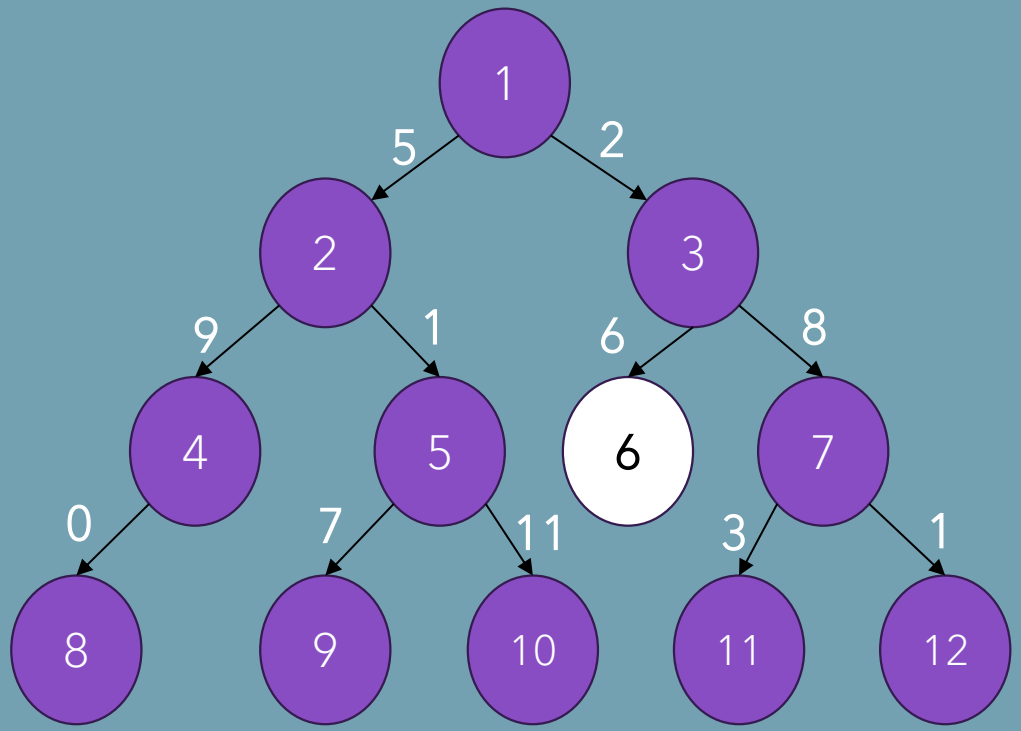


List

Insert → [ ] → Remove

UNIFORM-COST SEARCH

Goal: 6

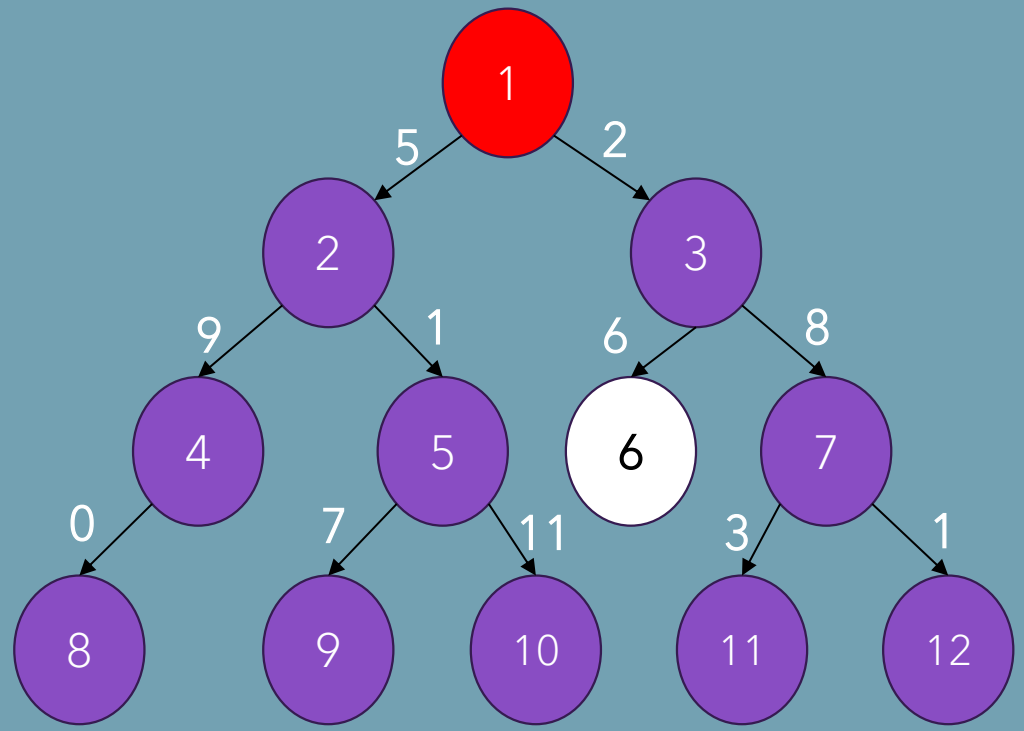


List

Insert → 1 → Remove

UNIFORM-COST SEARCH

Goal: 6

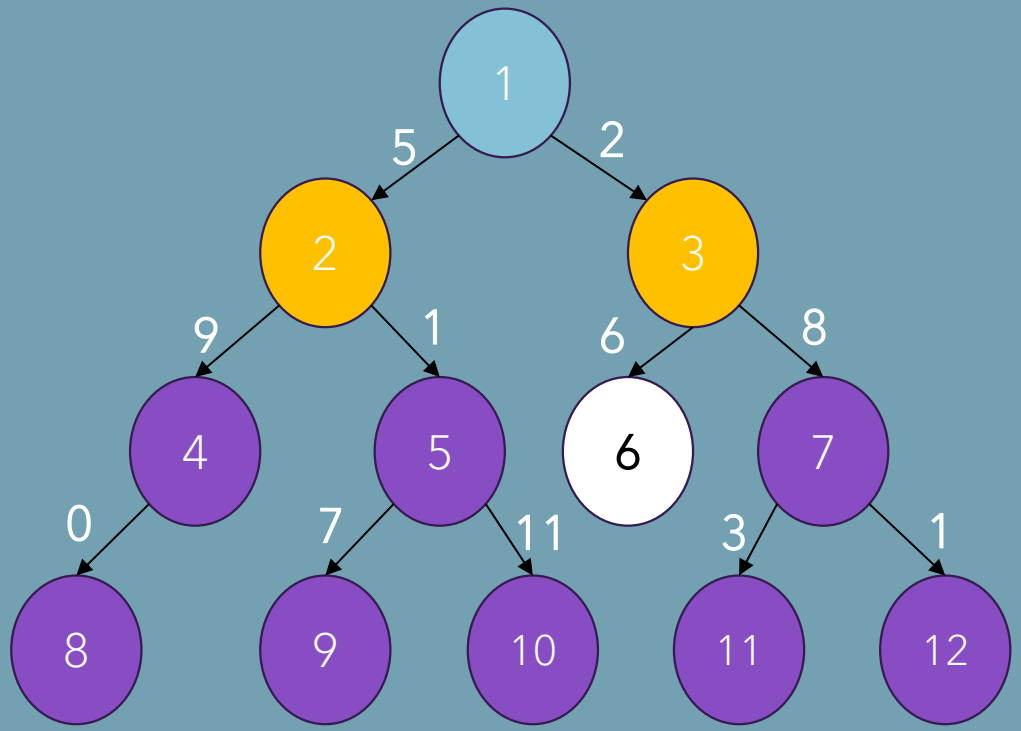


List

Insert → [ ] → Remove

UNIFORM-COST SEARCH

Goal: 6

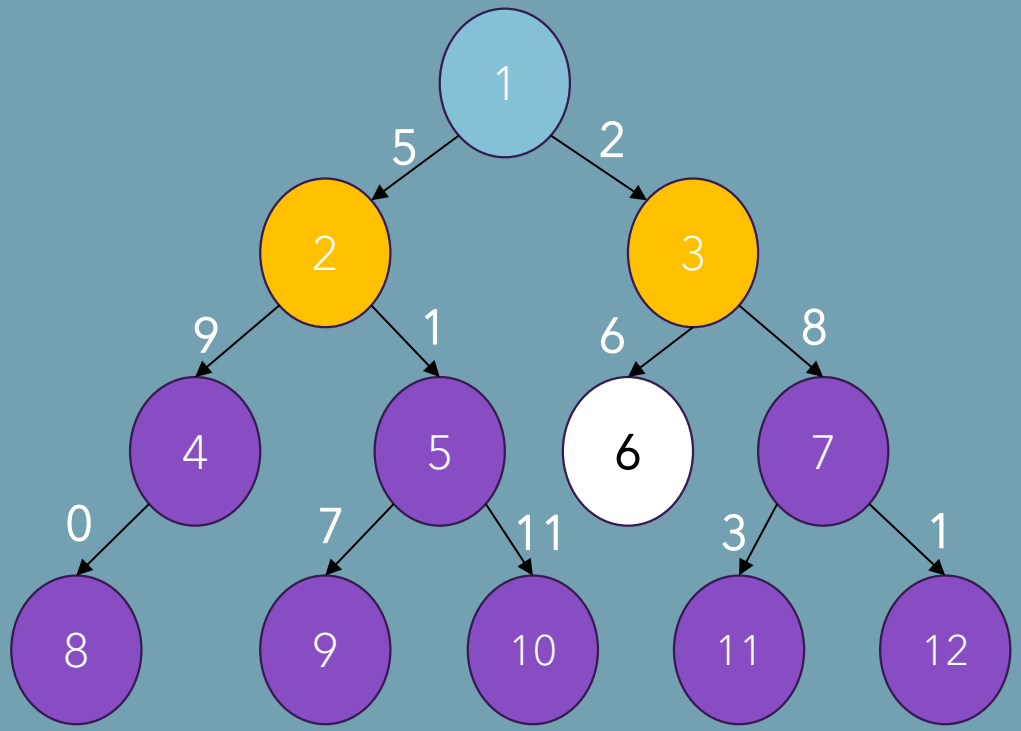


List

Insert → 3 (2), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6



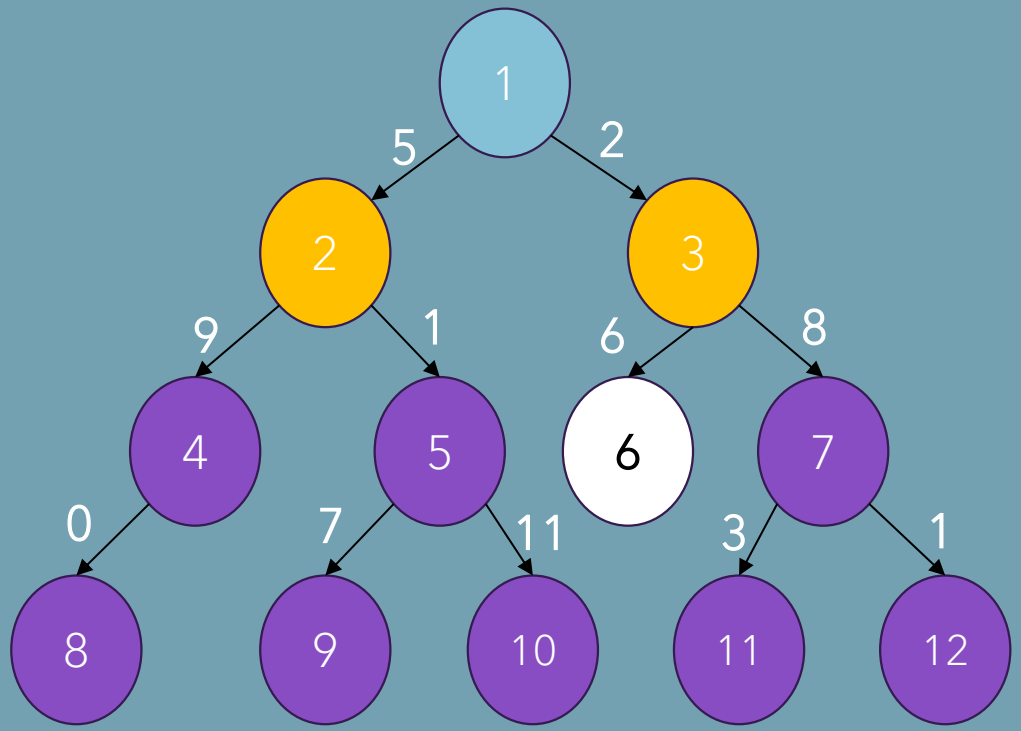
List

Sort based on cost

Insert → 3 (2), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6

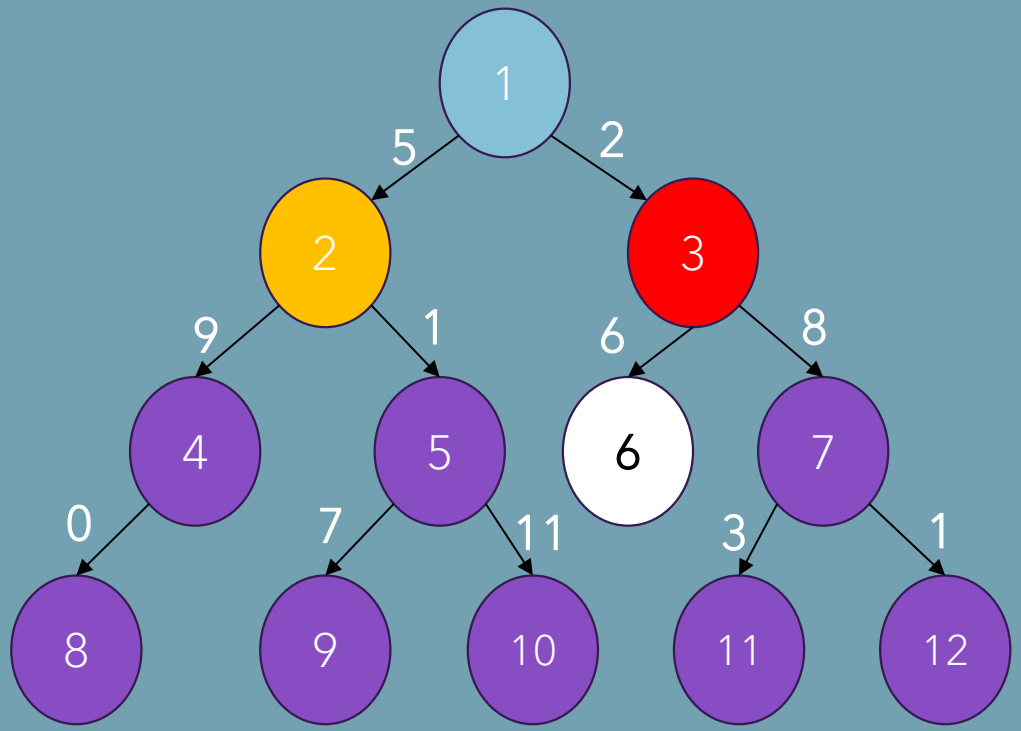


List

Insert → 2 (5), 3 (2) → Remove

UNIFORM-COST SEARCH

Goal: 6



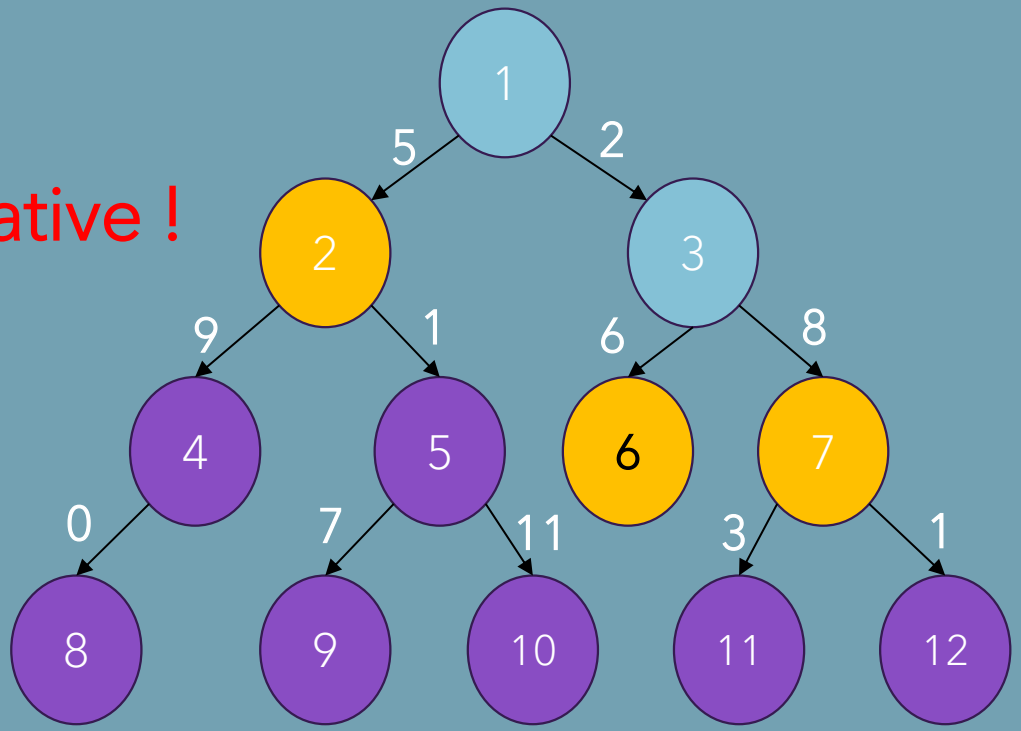
List

Insert → 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6

Costs are accumulative !

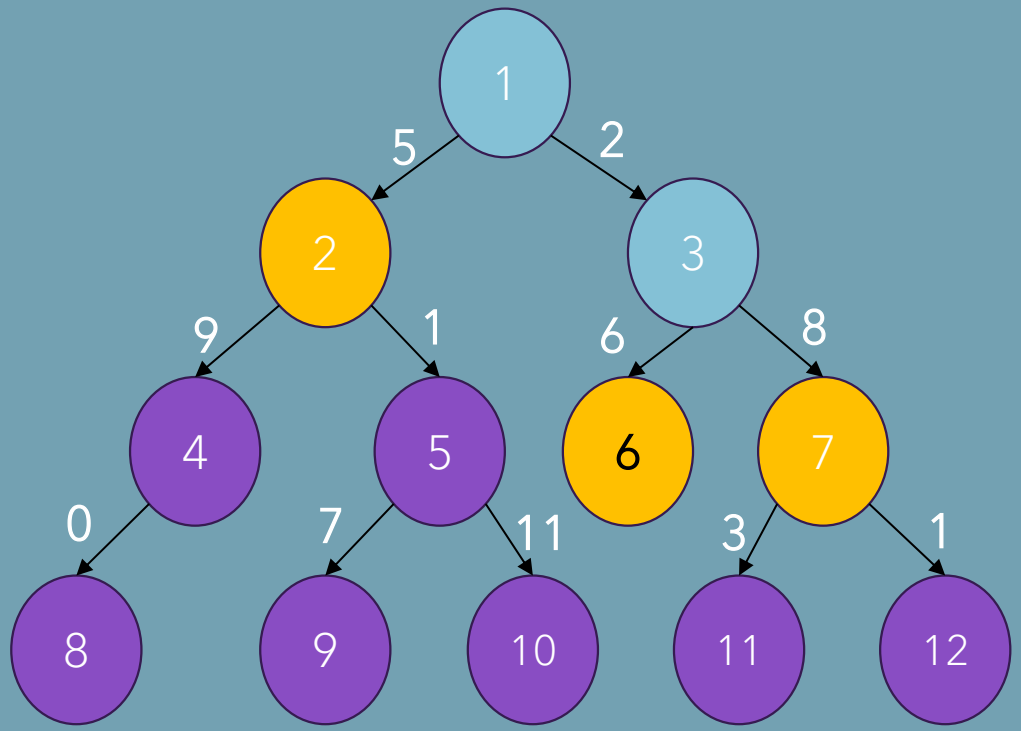


List

Insert → 7 (2 + 8), 6 (2 + 6), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6

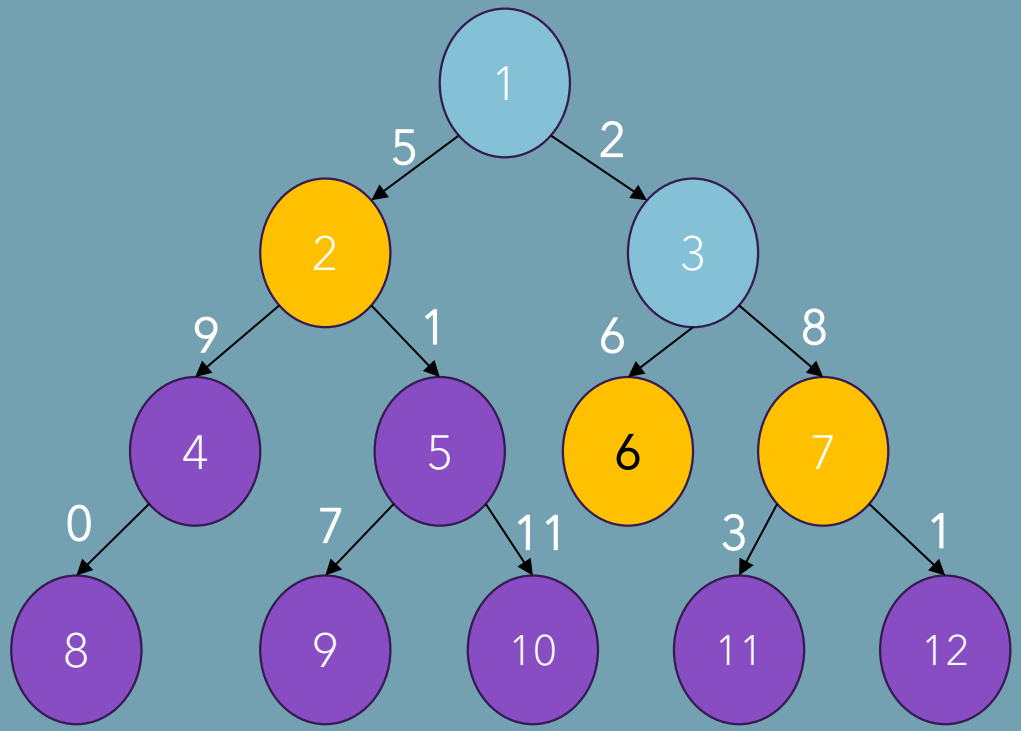


List

Insert → 7 (10), 6 (8), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6



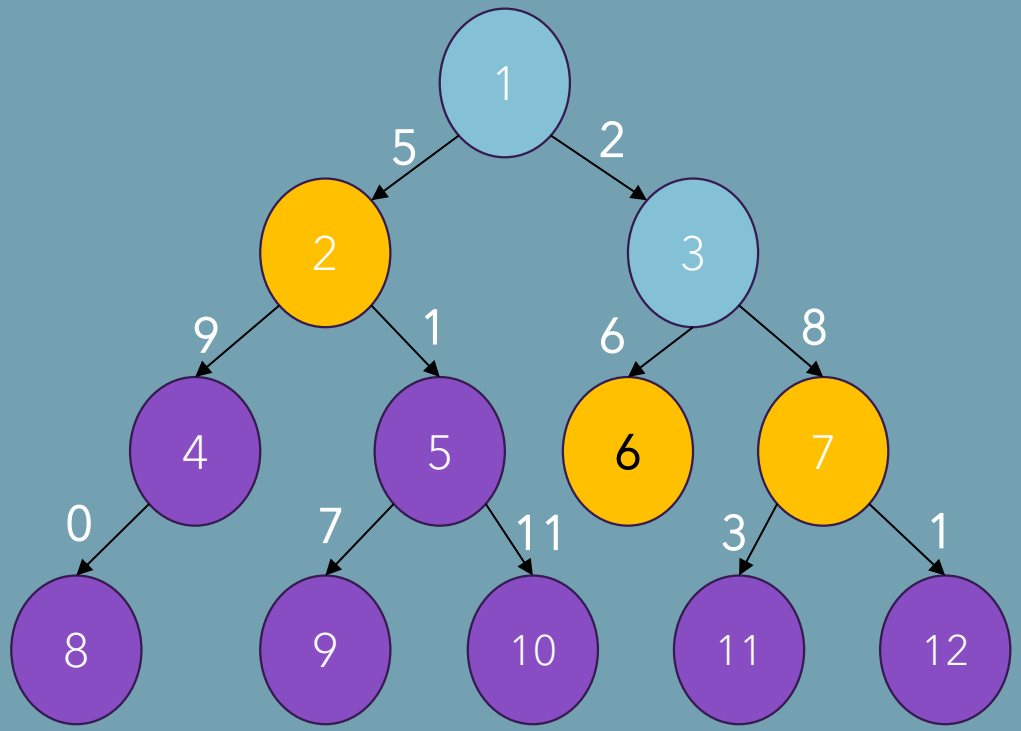
List

Sort based on cost

Insert → 7 (10), 6 (8), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6

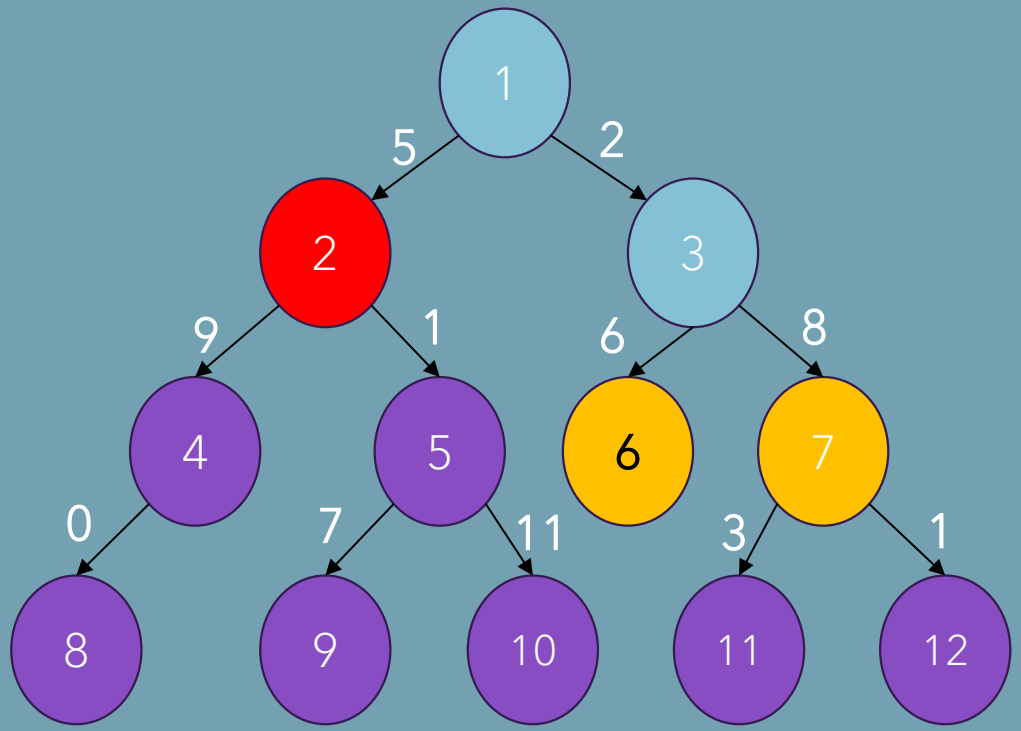


List

Insert → 7 (10), 6 (8), 2 (5) → Remove

UNIFORM-COST SEARCH

Goal: 6

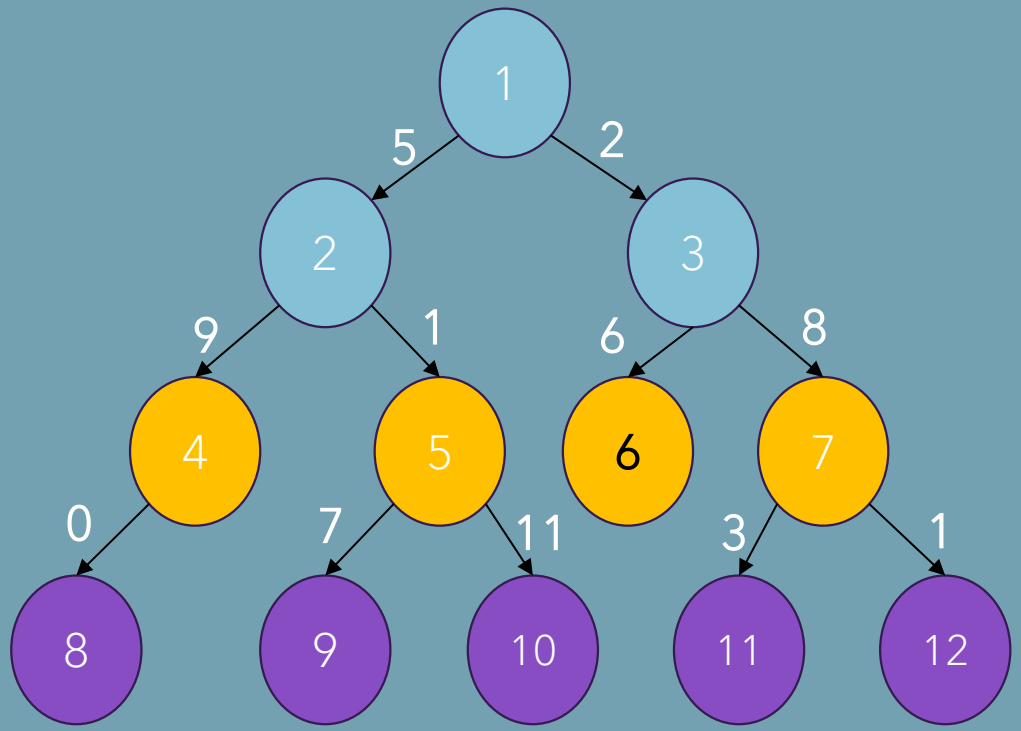


List

Insert → 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6

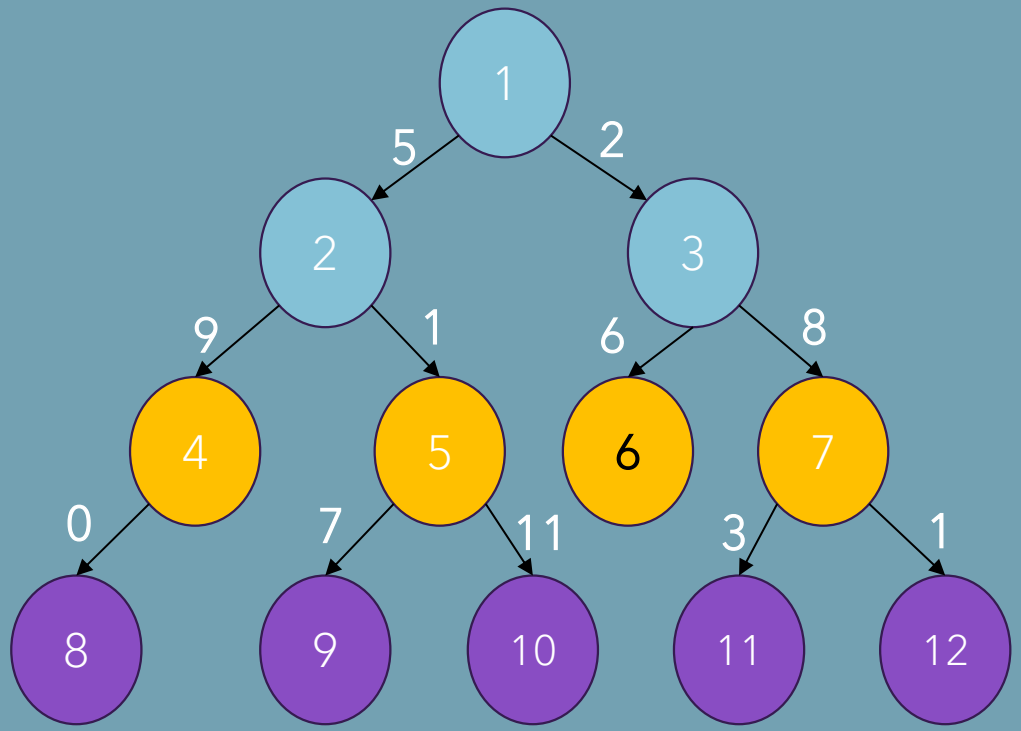


List

Insert → 5 (6), 4 (14), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6



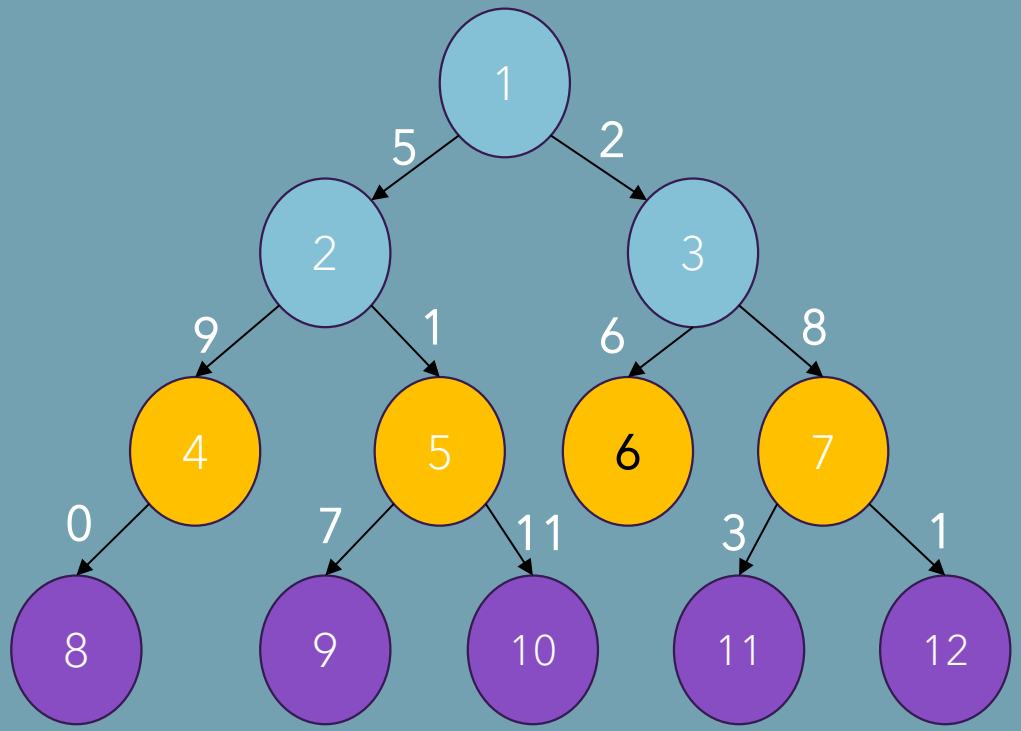
List

Sort based on cost

Insert → 5 (6), 4 (14), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6

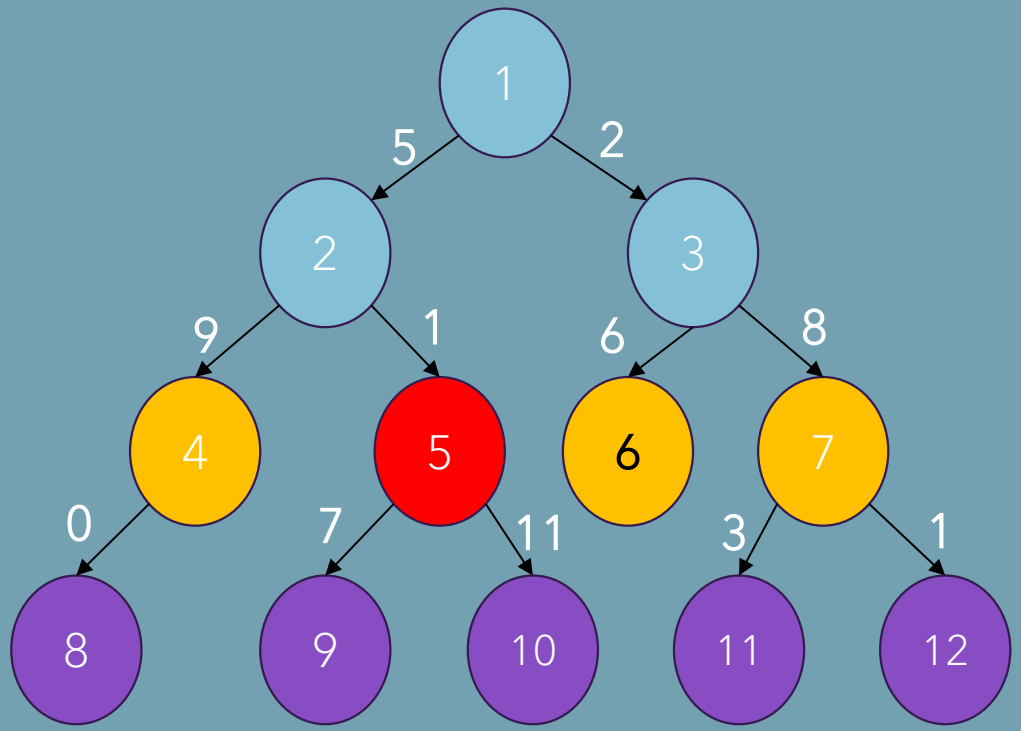


List

Insert → 4 (14), 7 (10), 6 (8), 5 (6) → Remove

UNIFORM-COST SEARCH

Goal: 6

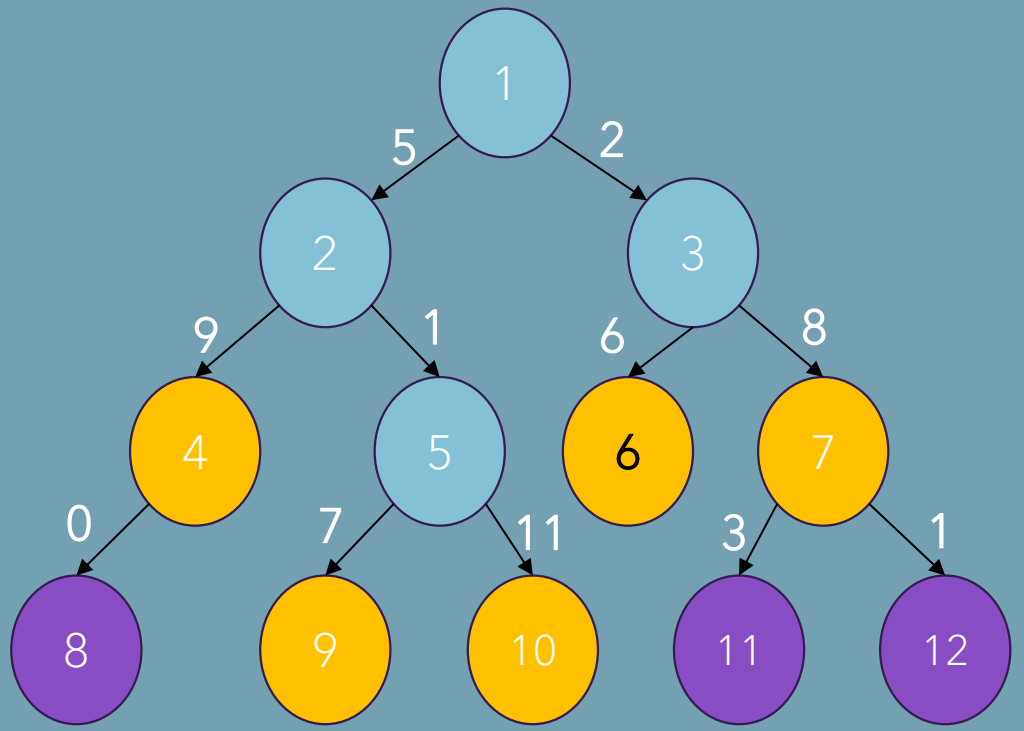


List

Insert → 4 (14), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6

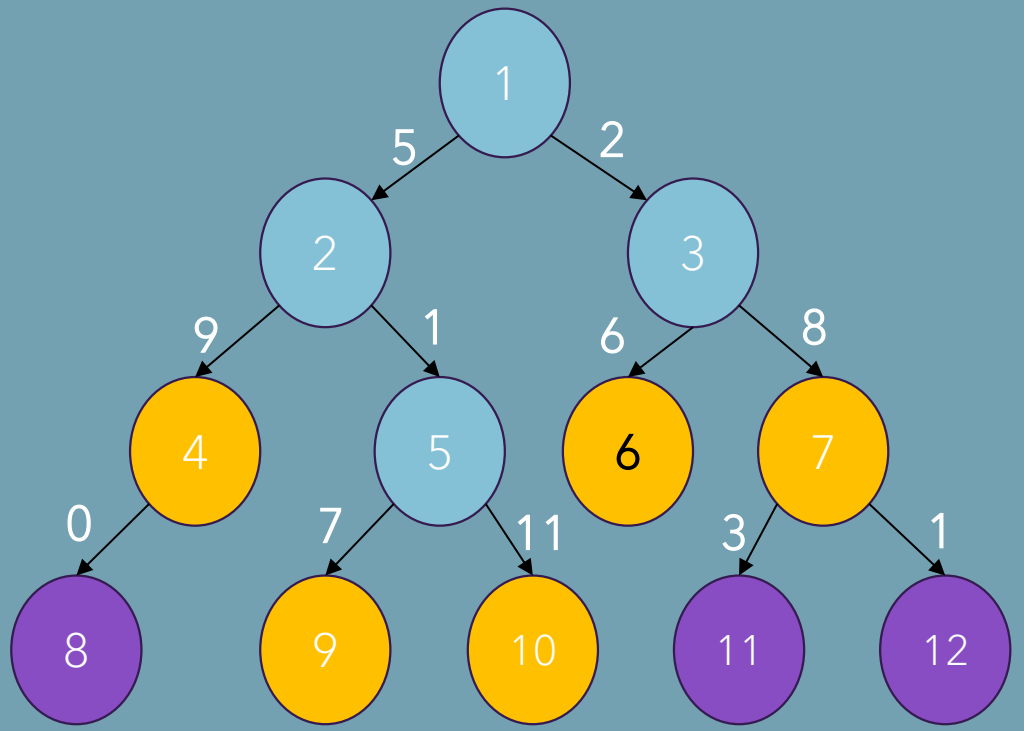


List

Insert → 10 (17), 9 (13), 4 (14), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6



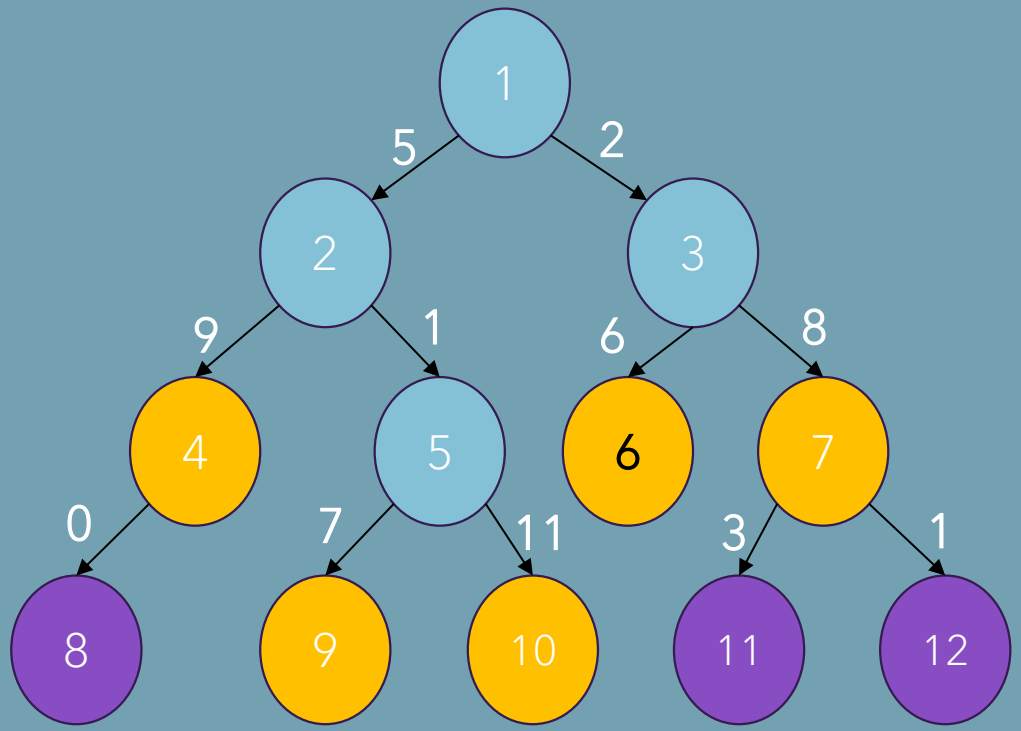
List

Sort based on cost

Insert → 10 (17), 9 (13), 4 (14), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6

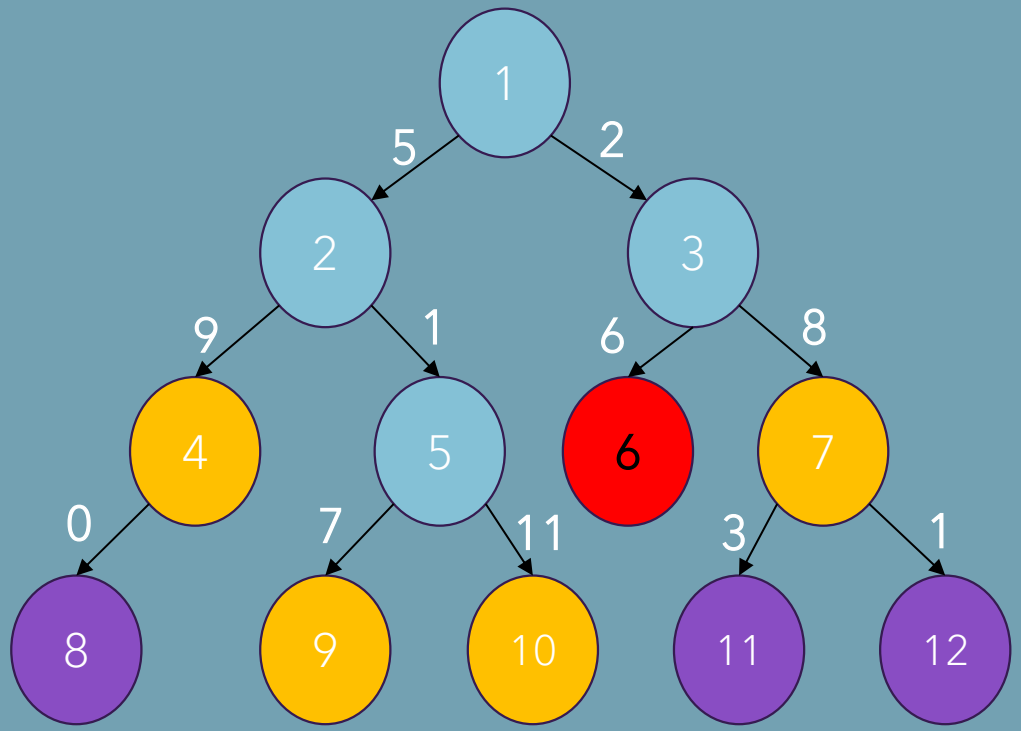


List

Insert → 10 (17), 4 (14), 9 (13), 7 (10), 6 (8) → Remove

UNIFORM-COST SEARCH

Goal: 6

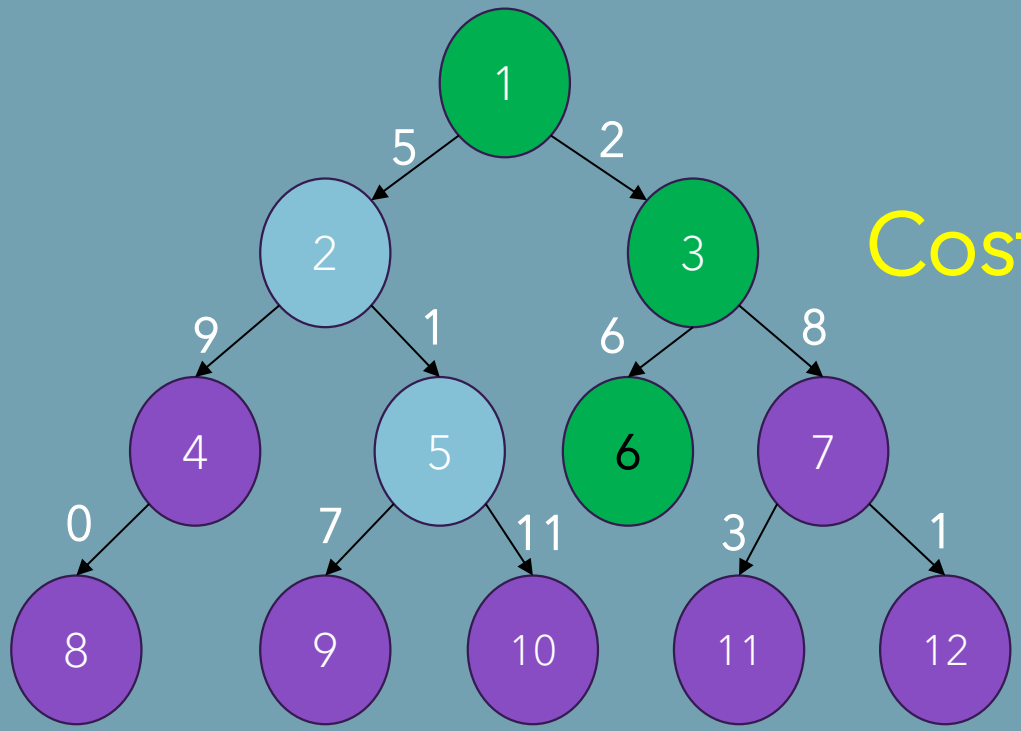


List

Insert → 10 (17), 4 (14), 9 (13), 7 (10) → Remove

UNIFORM-COST SEARCH

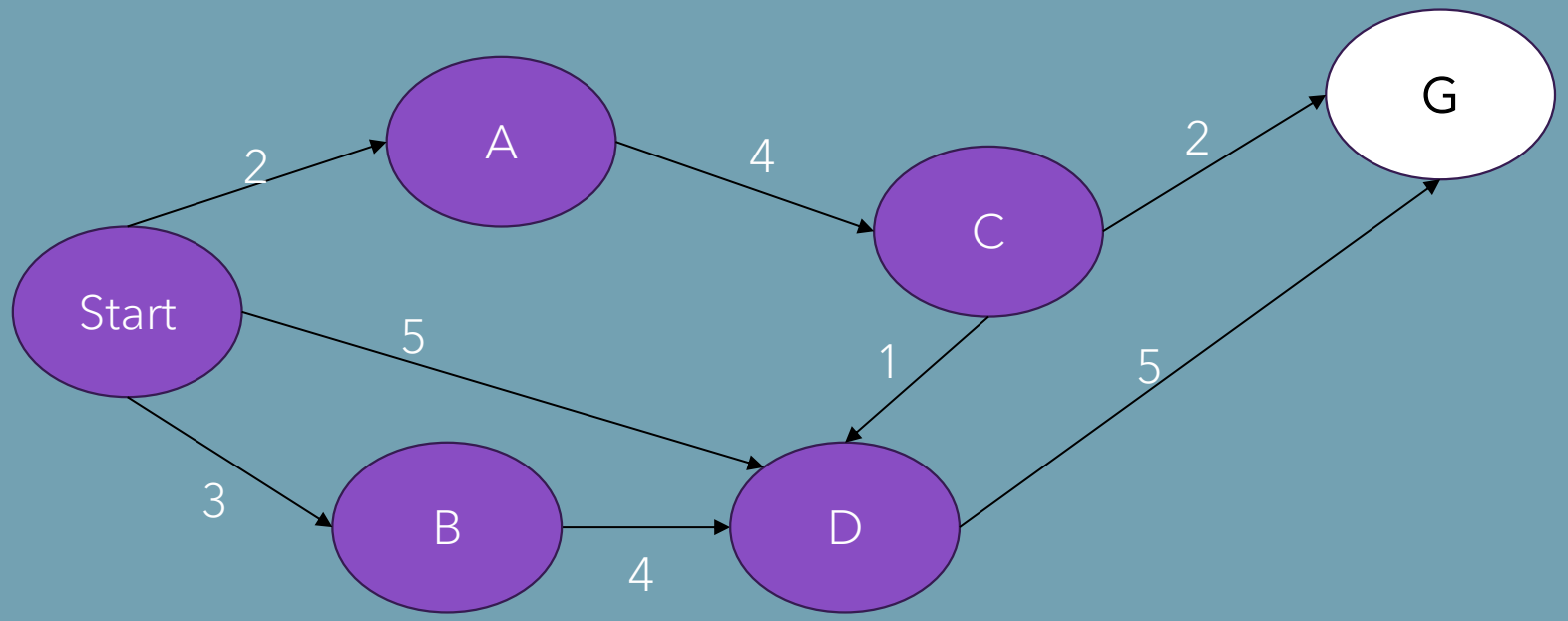
Goal: 6



Cost of the solution = 8

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

UNIFORM-COST SEARCH (GRAPH)



Current	FIFO FRONT < - - - - - - > REAR
	[S]
[S]	[S,A (2)], [S,B (3)], [S,D (5)]
[S, A (2)]	[S,B (3)], [S,D (5)], [S,A,C (6)]
[S, B (3)]	[S,D (5)], [S,A,C (6)], [S,B,D (7)]
[S,D (5)]	[S,A,C (6)], [S,B,D (7)], [S,D,G (10)]
[S,A,C (6)]	[S,A,C,D (7)], [S,B,D (7)], [S,A,C,G (8)], {S,D,G (10)]
[S,A, C, G (8)]	

In graphs, if two elements got the same cost, then sort alphabetically only for unified answer for all students !

DFS VS. BFS VS. UCS

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

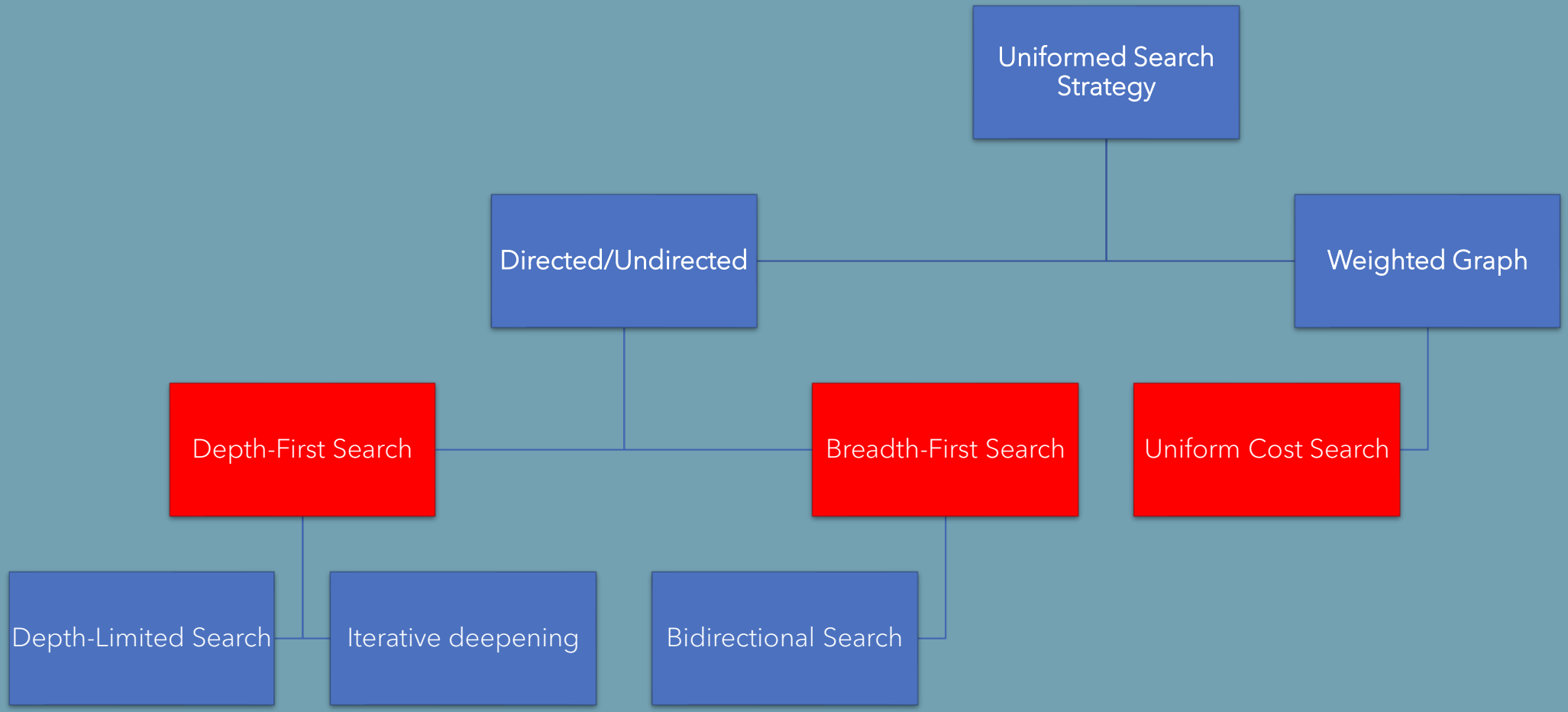
BFS

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

UCS

Current	FIFO FRONT < - - - - - - > REAR
	[S]
[S]	[S,A (2)], [S,B (3)], [S,D (5)]
[S, A (2)]	[S,B (3)], [S,D (5)], [S,A,C (6)]
[S, B (3)]	[S,D (5)], [S,A,C (6)], [S,B,D (7)]
[S,D (5)]	[S,A,C (6)], [S,B,D (7)], [S,D,G (10)]
[S,A,C (6)]	[S,A,C,D (7)], [S,B,D (7)], [S,A,C,G (8)], {S,D,G (10)]
[S,A, C, G (8)]	Actual Cost: 8 Sol. Steps: 6

SOME VARIANTS



IMPLEMENTATION

GRAPH CLASS (ADJACENCY LIST REP.)

```python
# 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)

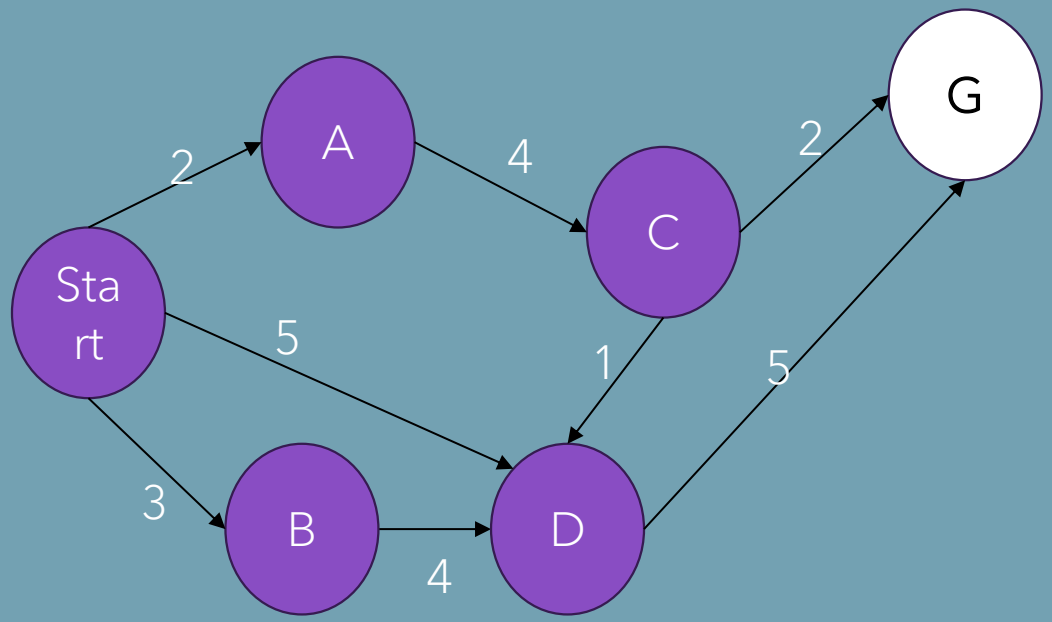
```
def add_edge(self, vertex_1, vertex_2, weight=None):
    if vertex_1 in self.__graph.keys() and vertex_2 in self.__graph.keys():
        if not self.__directed:
            if not self.__weighted:
                self.__graph[vertex_1].append(vertex_2)
                self.__graph[vertex_2].append(vertex_1)
            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




```
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 (UCS METHOD)

```python
def UCS(self, goal, start=None):
    if not self.__weighted:
        raise Exception("Uniform cost Search only works with Weighted Graphs !")

    if start != None:
        lst = [[(start, 0)]]
    else:
        lst = [[("root", 0)]]

    visited = []
    while lst:
        # Sort all elements alphabetically to get a unified answer !
        # Sorting based on total path cost and alphabetical order to follow "FIFO + Priority" principle
        lst.sort(key=self.__path_cost)
        # basic search
        current_path = lst.pop(0) # FIFO
        current_node = current_path[-1][0]

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

        if current_node == goal:
            return list(map(lambda x: x[0], current_path)), sum(map(lambda x: x[-1], current_path)), visited
        else:
            adjacent_nodes = self.__graph[current_node]
            for node, cost in adjacent_nodes:
                new_path = current_path.copy()
                new_path.append((node, cost))
                lst.append(new_path)

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

```python
def __path_cost(self, path):
    total_cost = 0
    for node,cost in path:
        total_cost += cost
    return total_cost, path[-1][0]
```

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"**

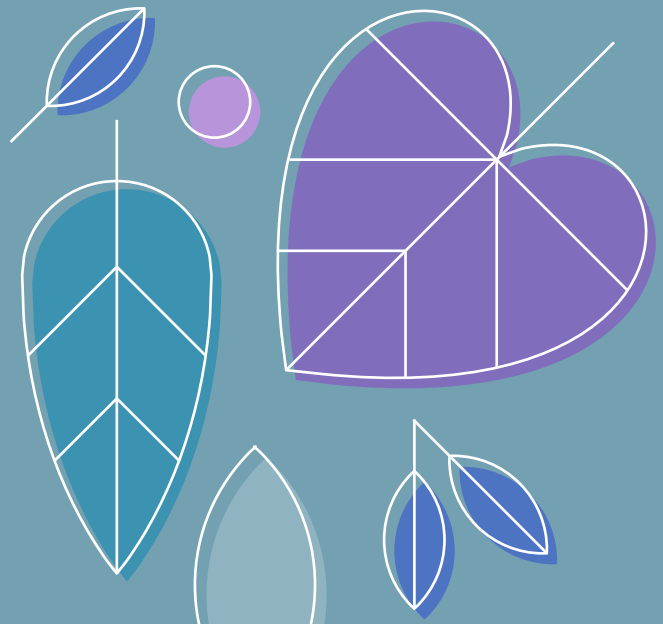
TESTING

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

(['S', 'A', 'C', 'G'], 8, ['S', 'A', 'B', 'D', 'C', 'G'])
```

UCS

Current	FIFO FRONT <------> REAR
	[S]
[S]	[S,A (2)], [S,B (3)], [S,D (5)]
[S, A (2)]	[S,B (3)], [S,D (5)], [S,A,C (6)]
[S, B (3)]	[S,D (5)], [S,A,C (6)], [S,B,D (7)]
[S,D (5)]	[S,A,C (6)], [S,B,D (7)], [S,D,G (10)]
[S,A,C (6)]	[S,A,C,D (7)], [S,B,D (7)], [S,A,C,G (8)], [S,D,G (10)]
[S,A, C, G (8)]	Actual Cost: 8 Sol. Steps: 6



NEXT LAB: SEARCHING PROBLEMS (3)

Thank you