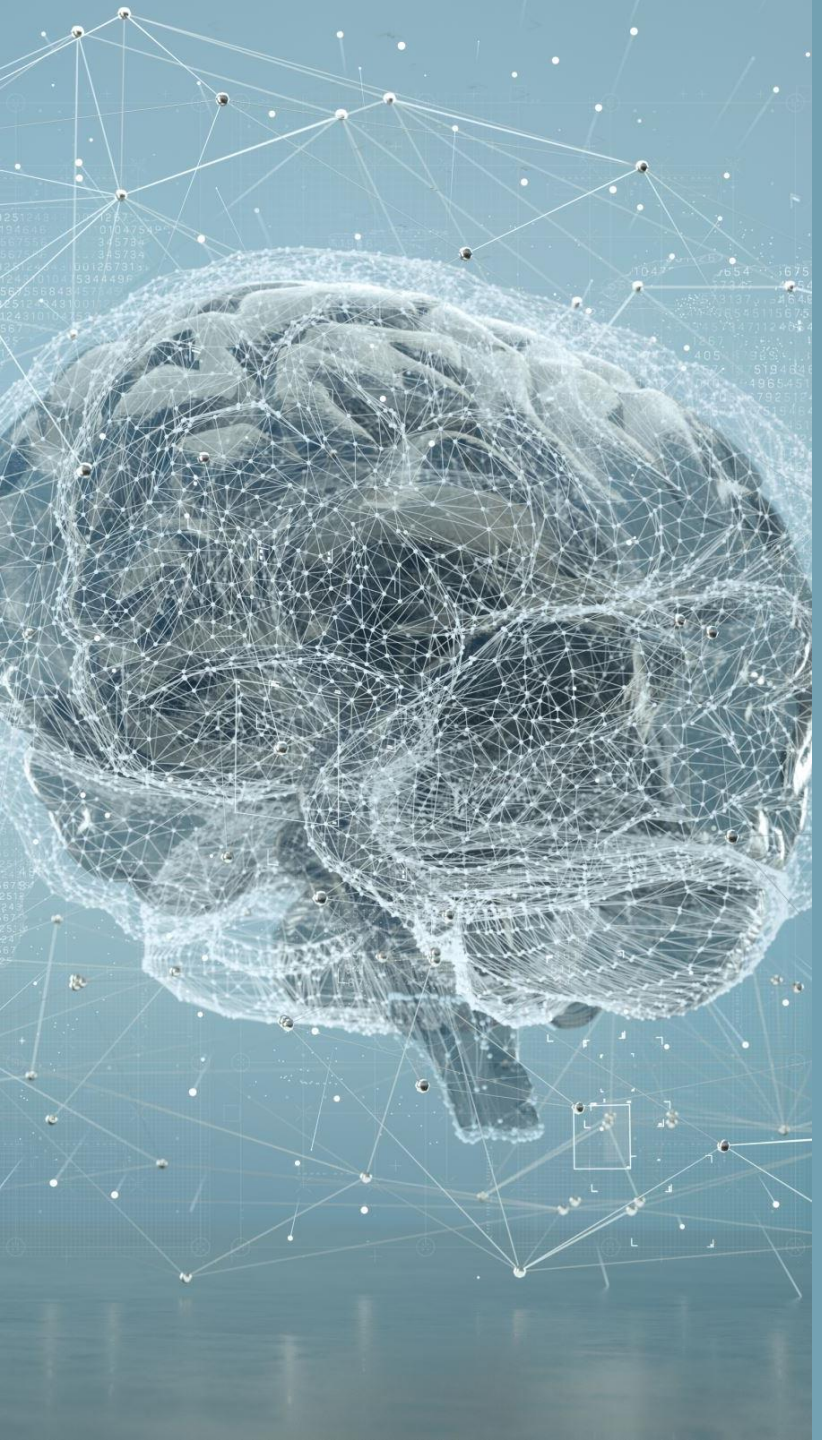


كلية الحاسبات والذكاء الاصطناعي
FACULTY OF COMPUTERS & ARTIFICIAL INTELLIGENCE

ARTIFICIAL INTELLIGENCE
LAB 05 – SEARCHING PROBLEMS (3)

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 !

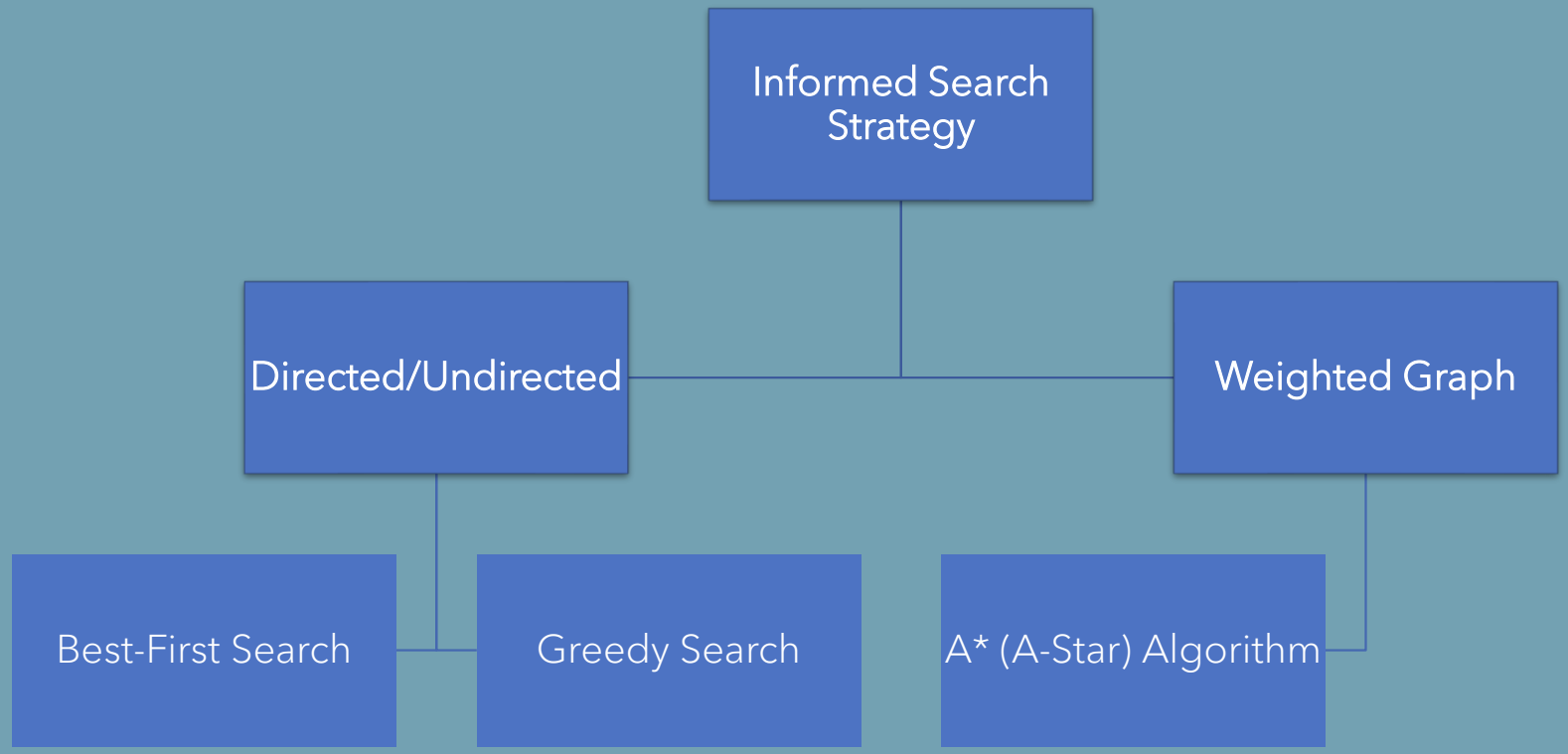
In our course, we are targeting

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

INFORMED SEARCH

- Informed Search (Heuristic Search): a type of search strategy in artificial intelligence that uses additional knowledge (heuristics) about the state to make better decisions about which path to follow toward the goal.



WHAT IS "HEURISTIC" ?

How close are we currently to the goal ?

Heuristic values are such an **indicators (hints)** that ESTIMATES how close a given state to the goal in a search problem

It is used in Informed search algorithms to help decide which path to explore next

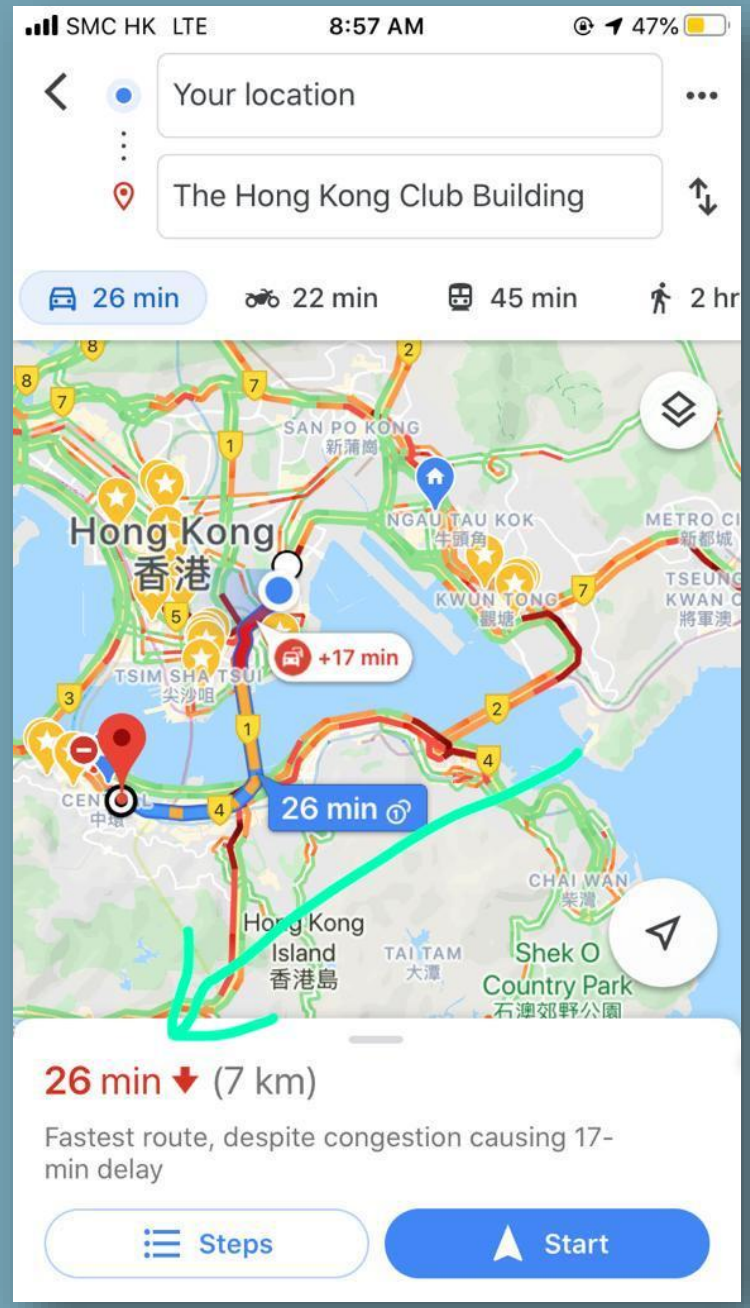
Remember !
Just an estimation, but not the actual !

Could be something similar ☺

REAL-LIFE ANALOGY (1)

You are using Google maps too much !

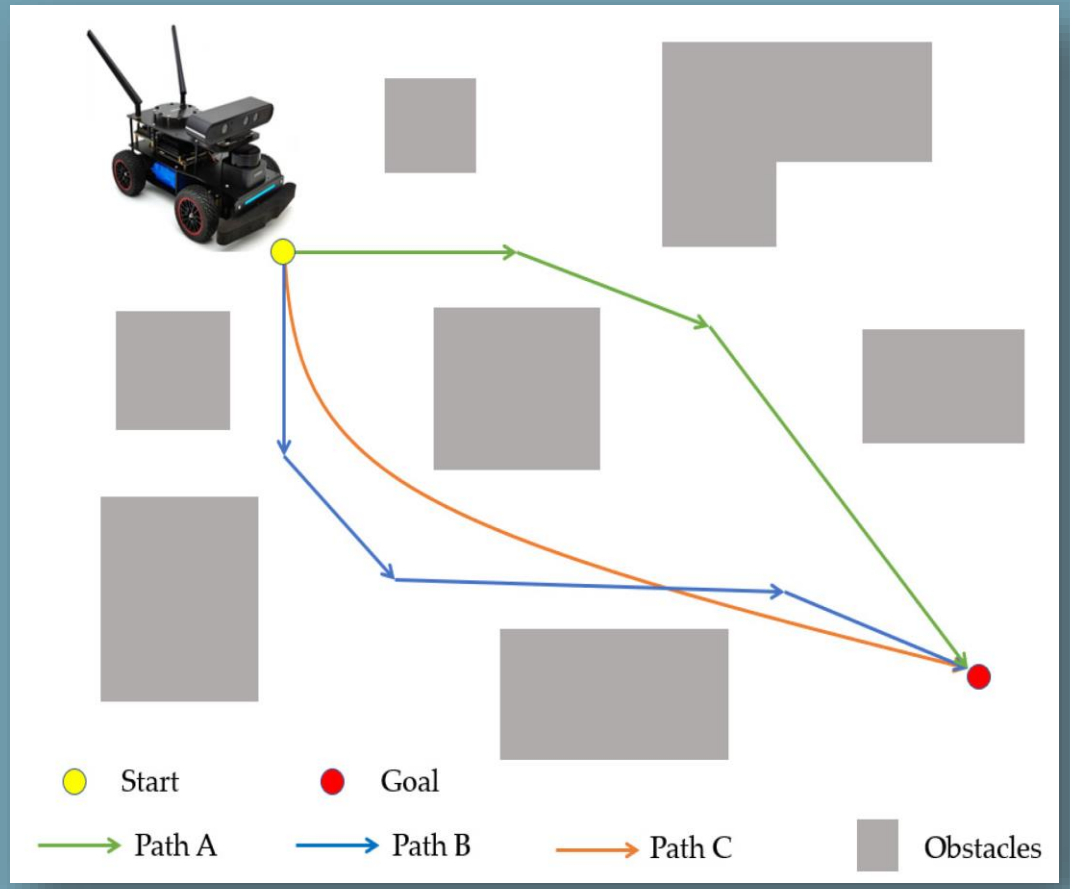
- The **estimated time arrival** is like **heuristic**
- It's based on distance, traffic, etc. but it's not always 100% accurate !



REAL-LIFE ANALOGY (2)

Robots planning too !

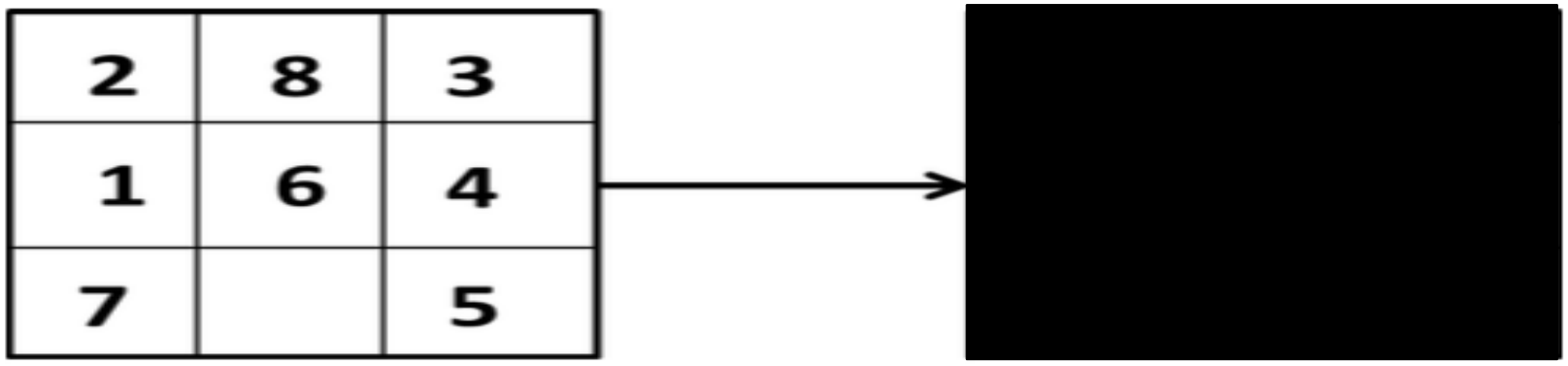
- In Robot path planning, the heuristics can be the Euclidean distance in route-finding problems
- We can estimate the distance from the current state to a goal by computing the straight-line distance on the map.



REAL-LIFE ANALOGY (3)

The most common 8-Puzzle problem

In 8-puzzle problem, the heuristic could be the number of misplaced tiles
In the example below, assume that you can't see the goal state, I'll give you a hint that the number of misplaced tiles = 5, as this considered as a hint, so it is a heuristic



REAL-LIFE ANALOGY (4)

Soda matching challenge on TikTok

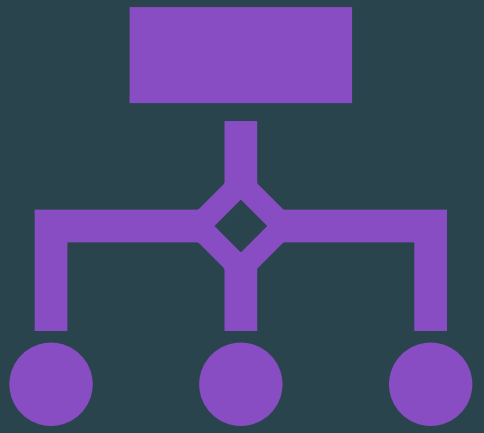
- The most popular challenge on TikTok is the Soda matching challenge
- The judge is telling the competitors how many soda are matching to the pattern, this can be considered as heuristic

WHAT IS THE NOTATION ?

We will refer to the actual path cost as g(n)

As we will refer to the heuristic function (estimated value) as h(n)

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"**

BEST-FIRST SEARCH

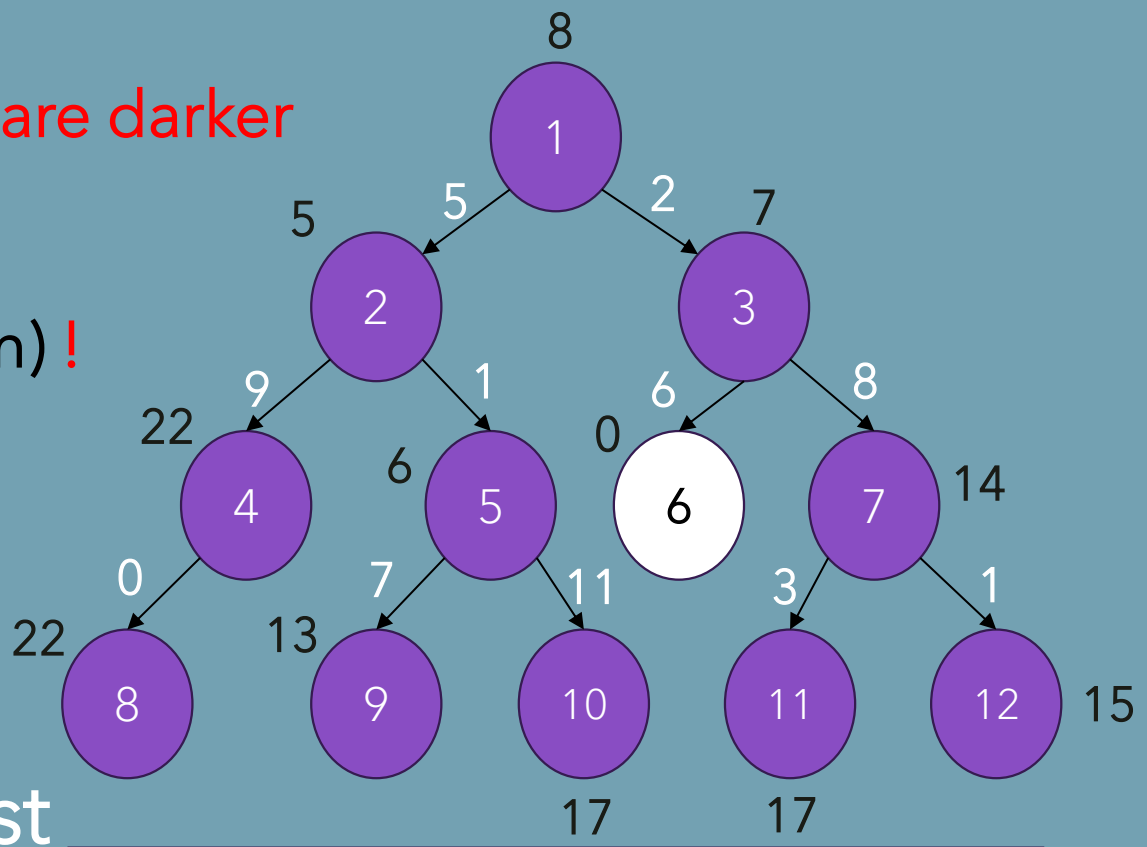
FIFO + Priority Function

Priority function is a data access principle that gives a priority for each element using FIFO, which the highest priority removed first. In Best-First Search, the priority for the lowest heuristic cost h(n) **(Sort)**, and the cost of each path is **not accumulative**

BEST-FIRST SEARCH

Goal: 6

Note:
The heuristic values are darker
Remember:
Pre-order traverse !
Priority for lowest h(n) !

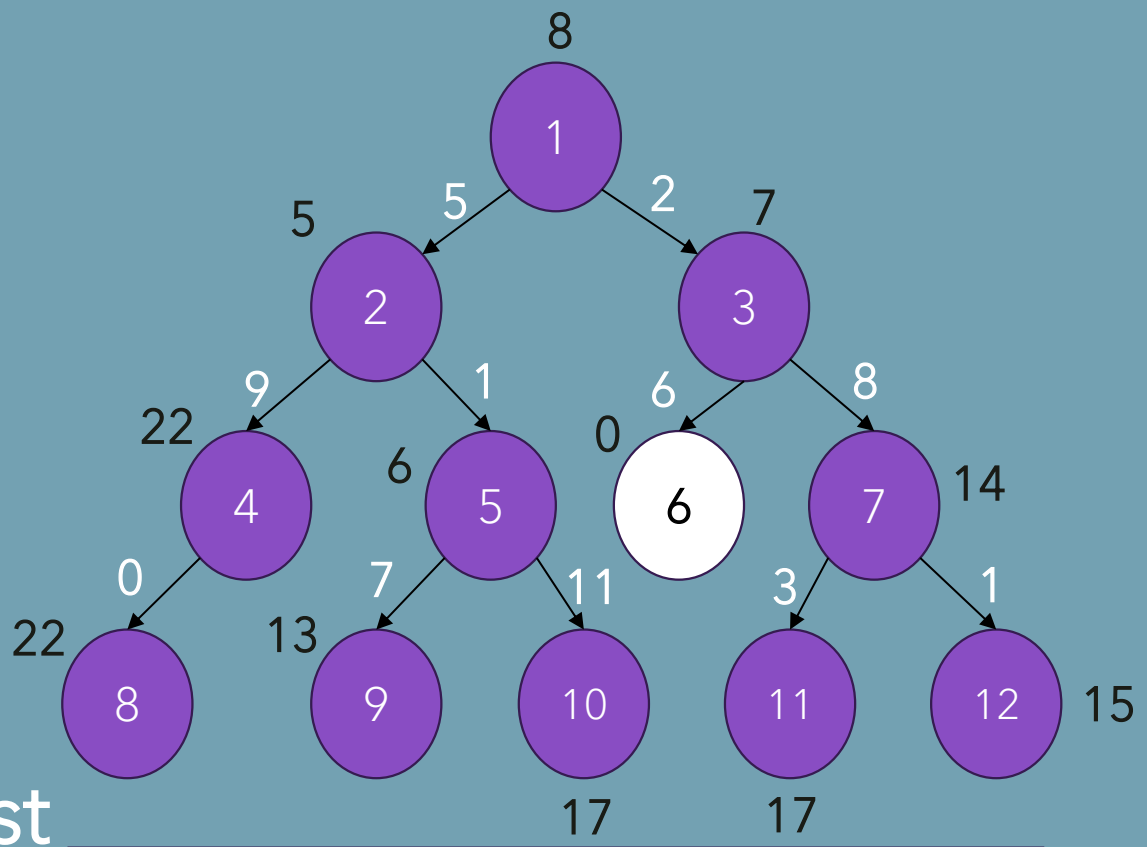


List

Insert → [ ] → Remove

BEST-FIRST SEARCH

Goal: 6

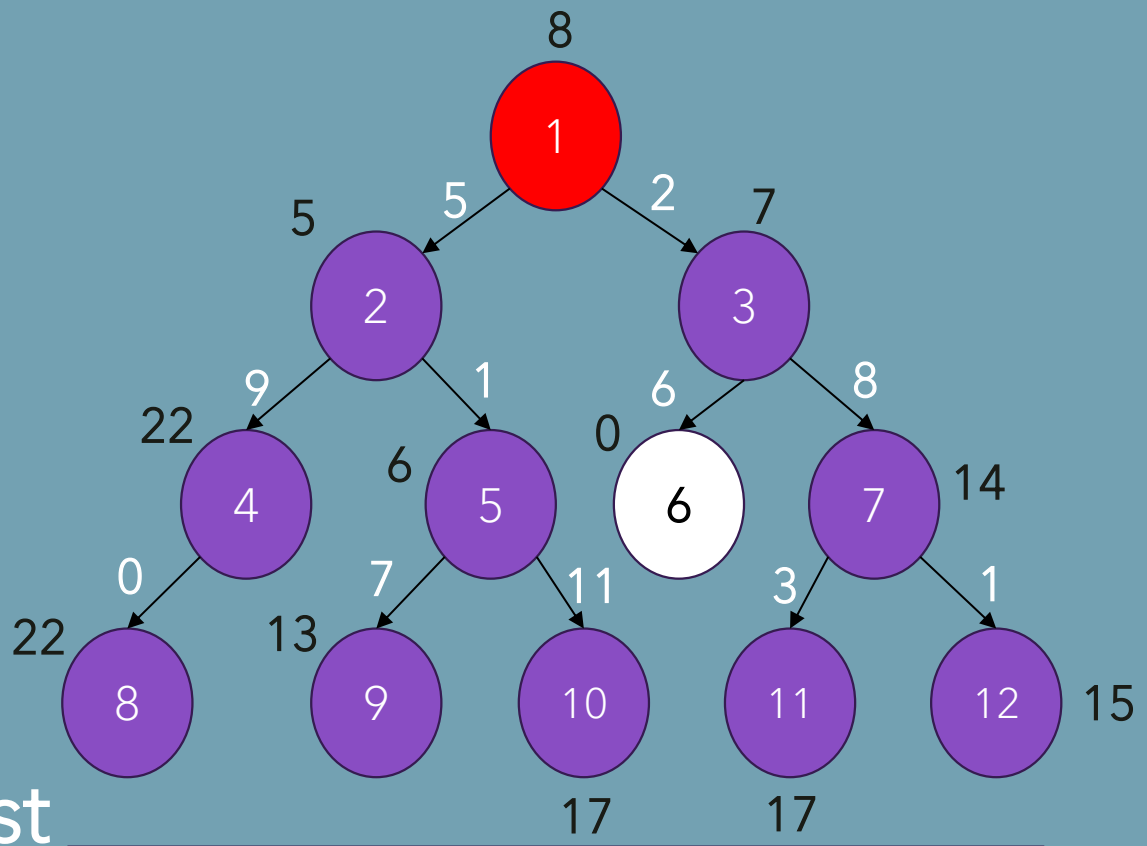


List

Insert → 1 (8) → Remove

BEST-FIRST SEARCH

Goal: 6

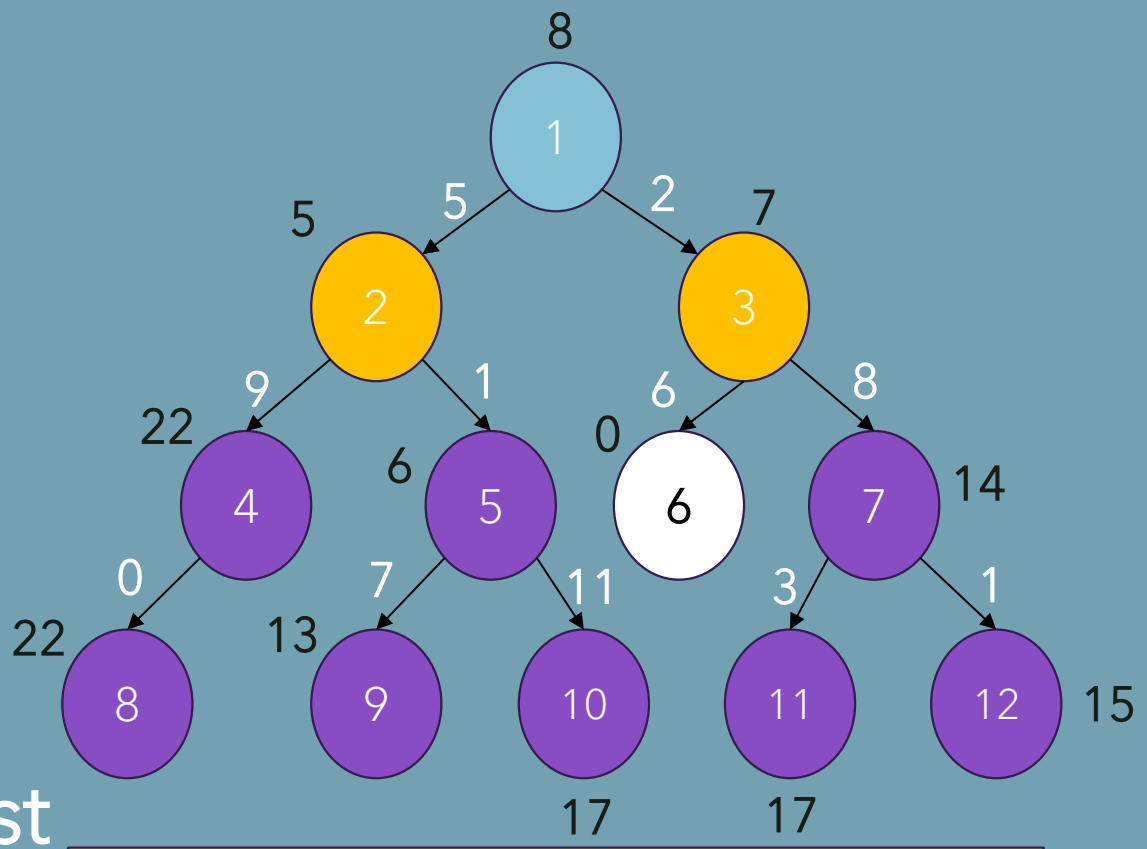


List

Insert → → Remove

BEST-FIRST SEARCH

Goal: 6

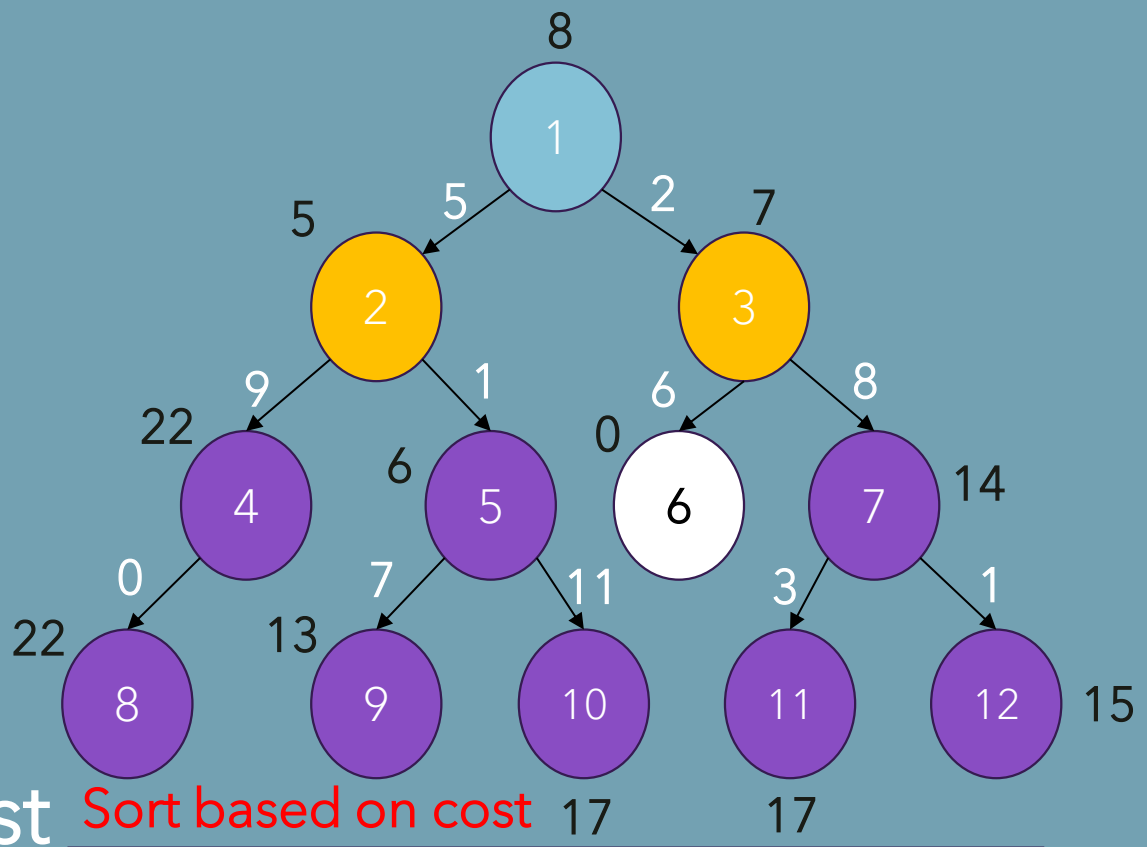


List

Insert → 3 (7), 2 (5) → Remove

BEST-FIRST SEARCH

Goal: 6

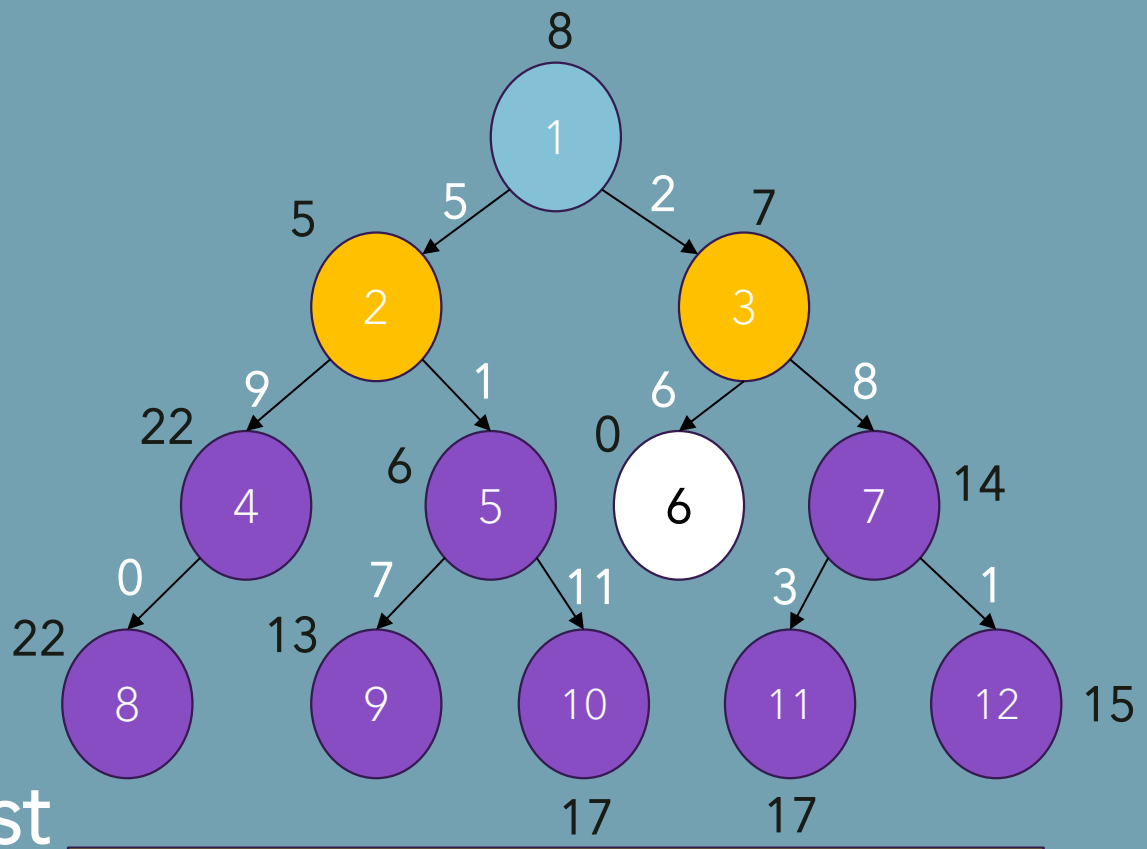


List Sort based on cost

Insert → 3 (7), 2 (5) → Remove

BEST-FIRST SEARCH

Goal: 6

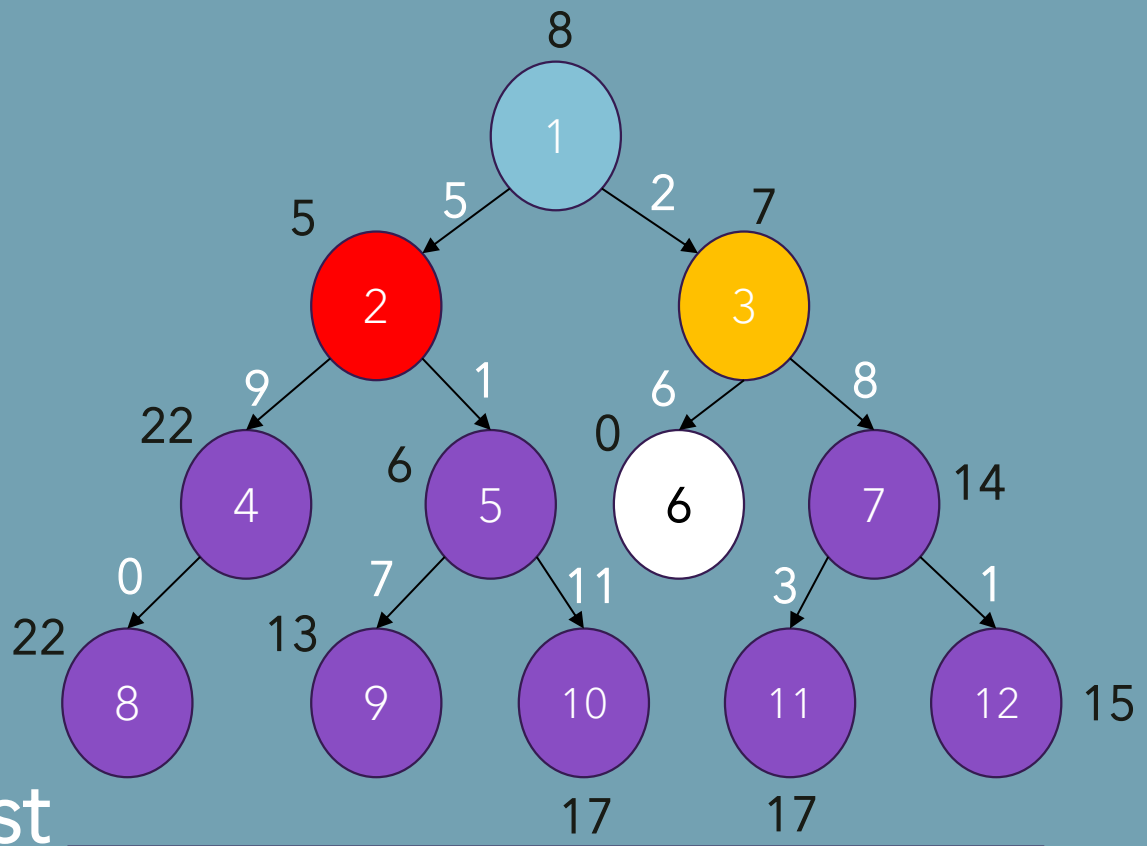


List

Insert → 3 (7), 2 (5) → Remove

BEST-FIRST SEARCH

Goal: 6



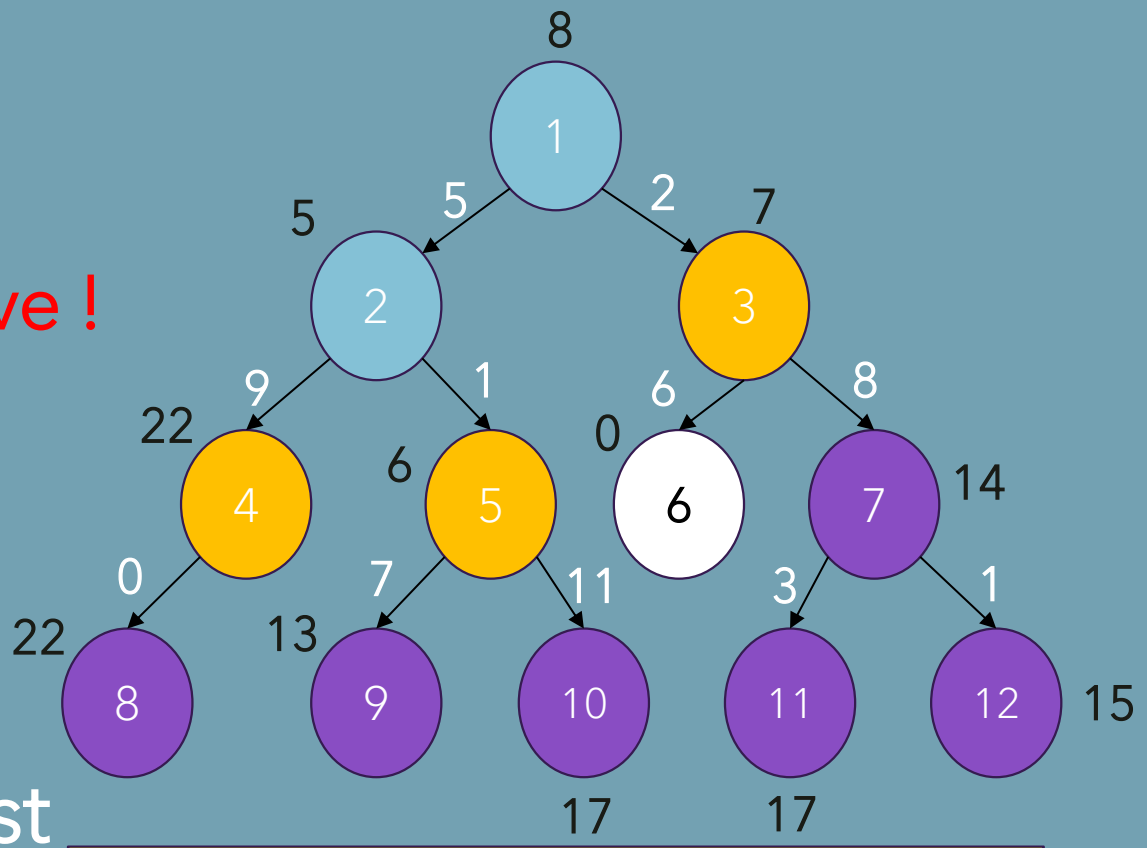
List

Insert → 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

h(n) costs are
NOT accumulative !

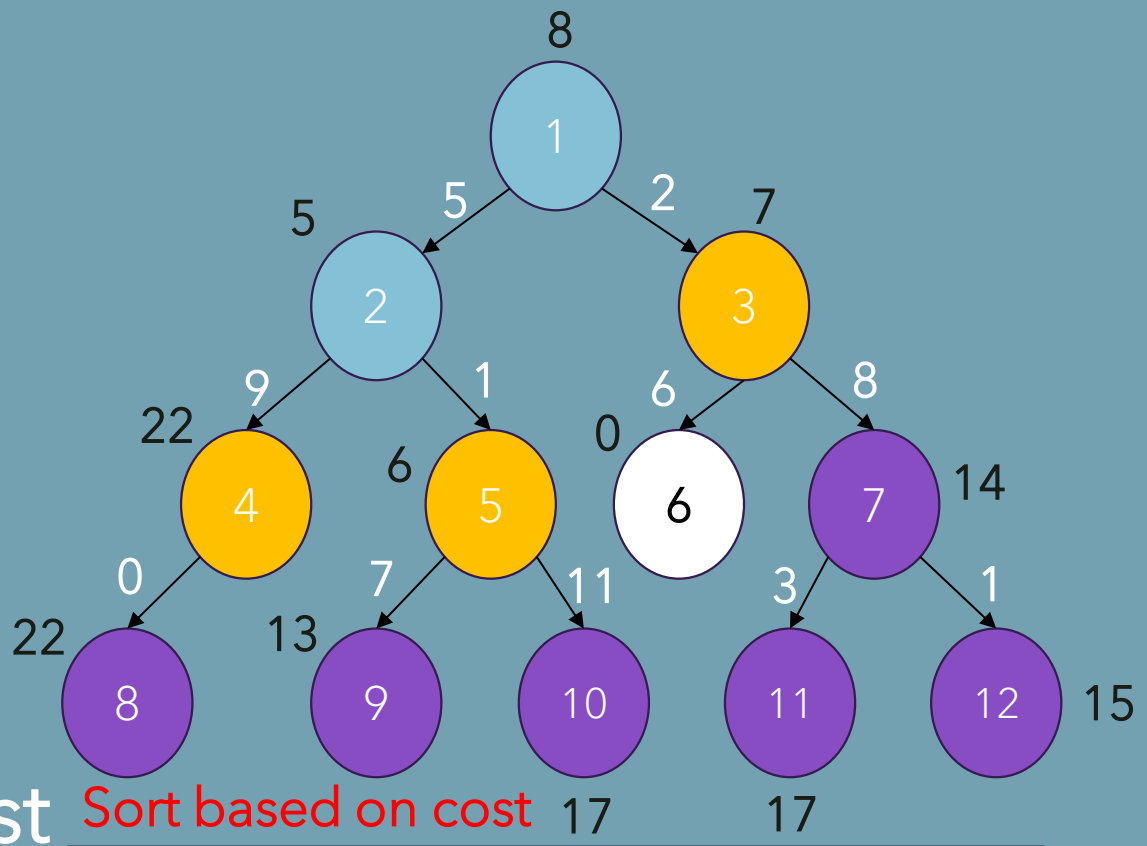


List

Insert → 5 (6), 4 (22), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

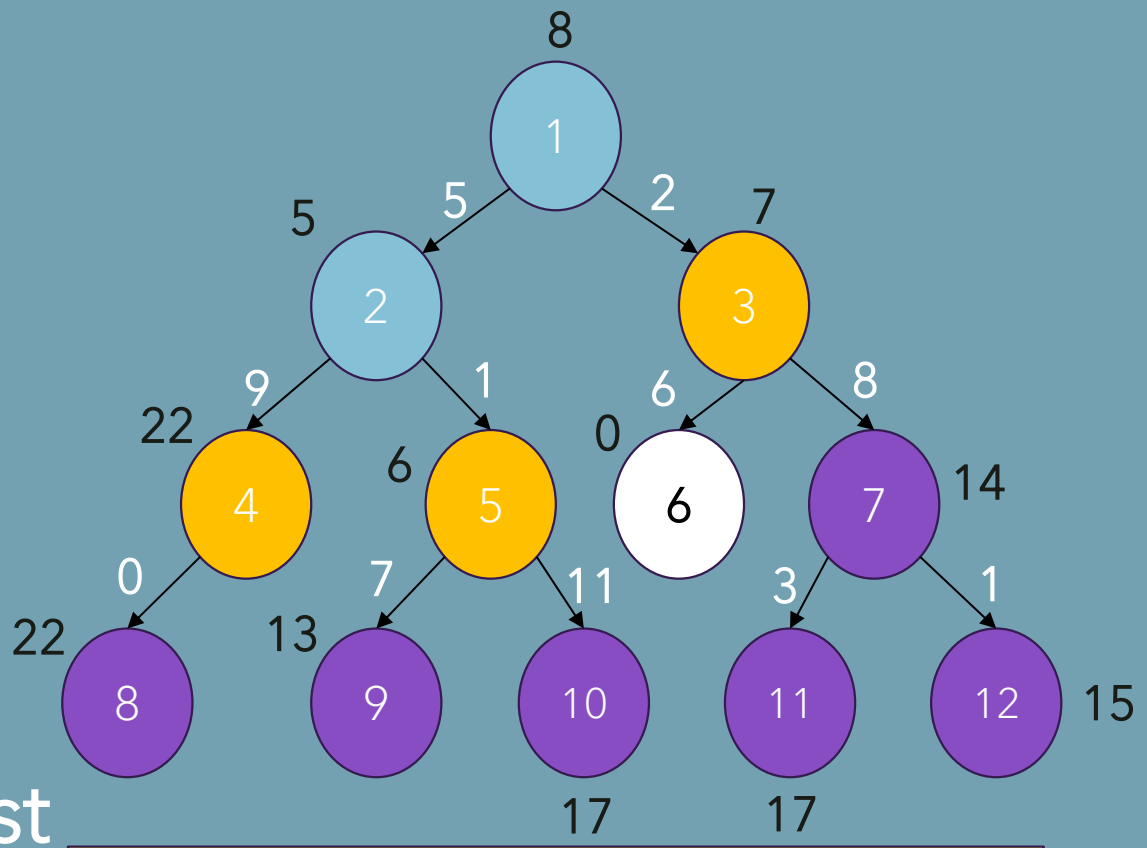


List Sort based on cost

Insert → 5 (6), 4 (22), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

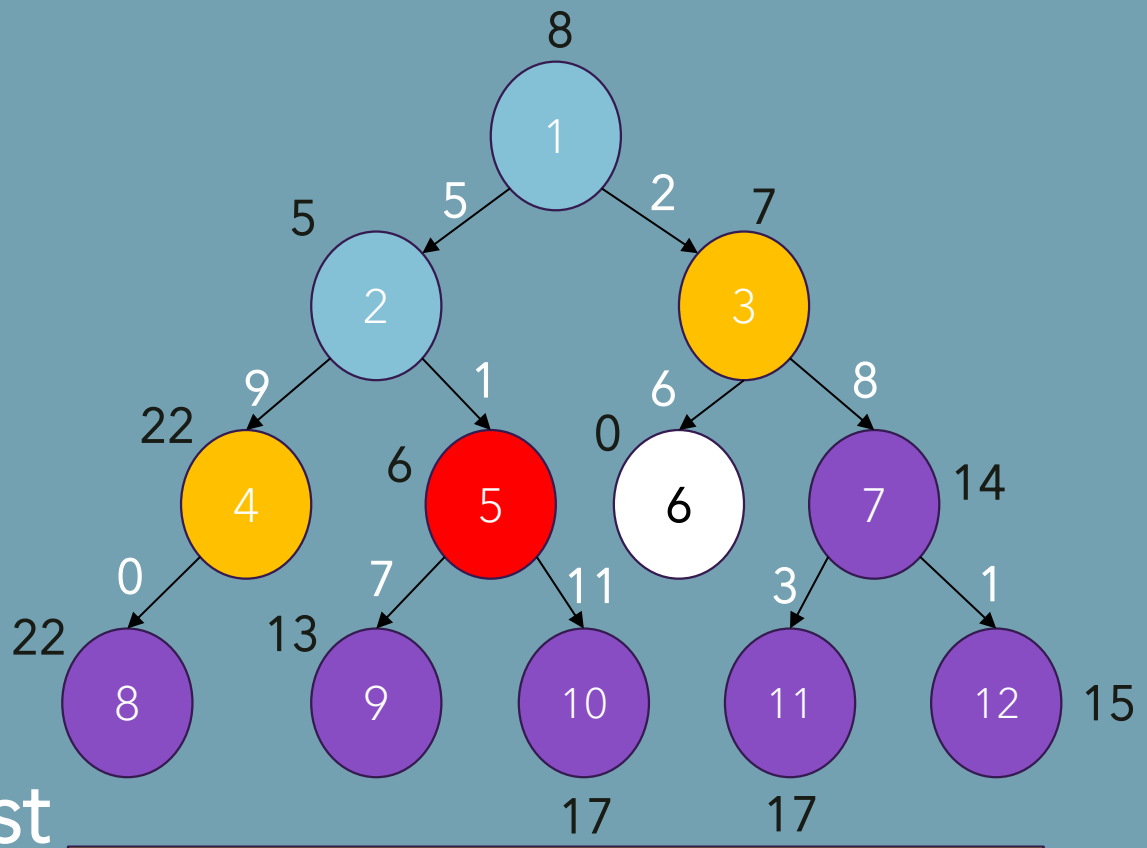


List

Insert → 4 (22), 3 (7), 5 (6) → Remove

BEST-FIRST SEARCH

Goal: 6

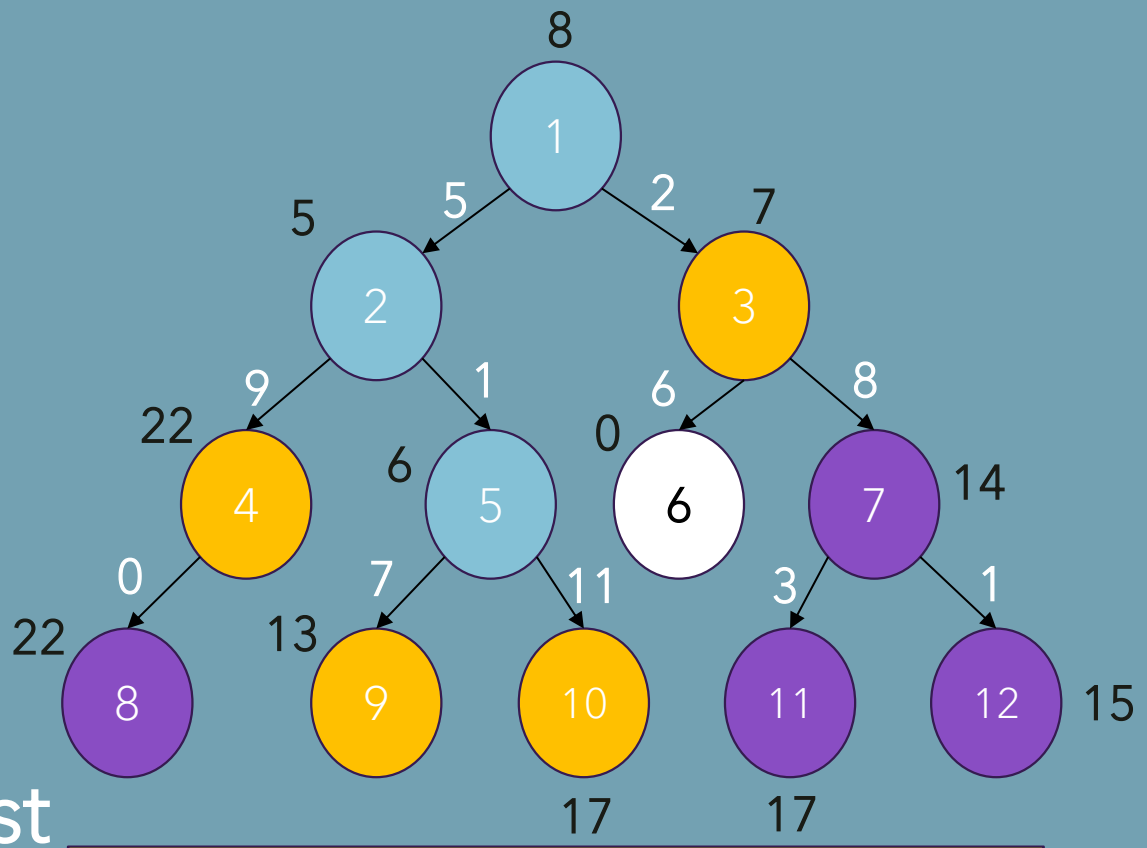


List

Insert → 4 (22), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

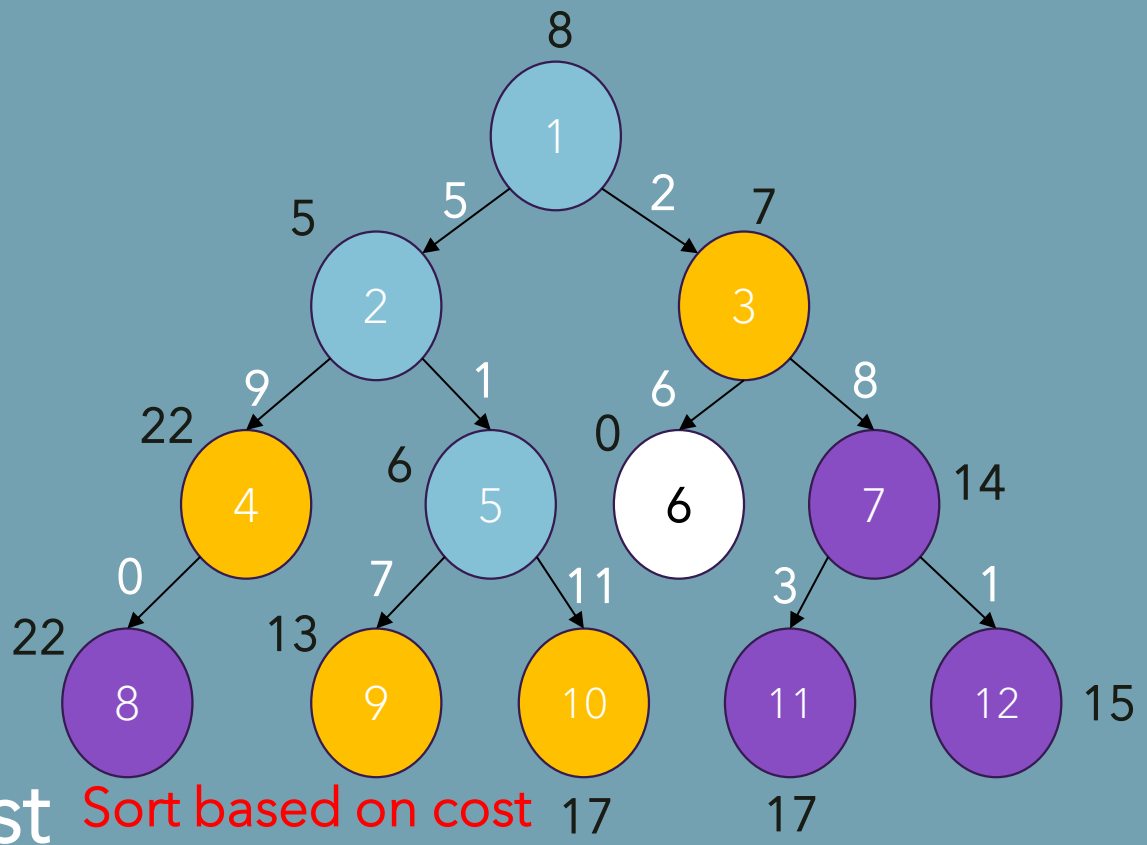


List

Insert → 10 (17), 9 (13), 4 (22), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

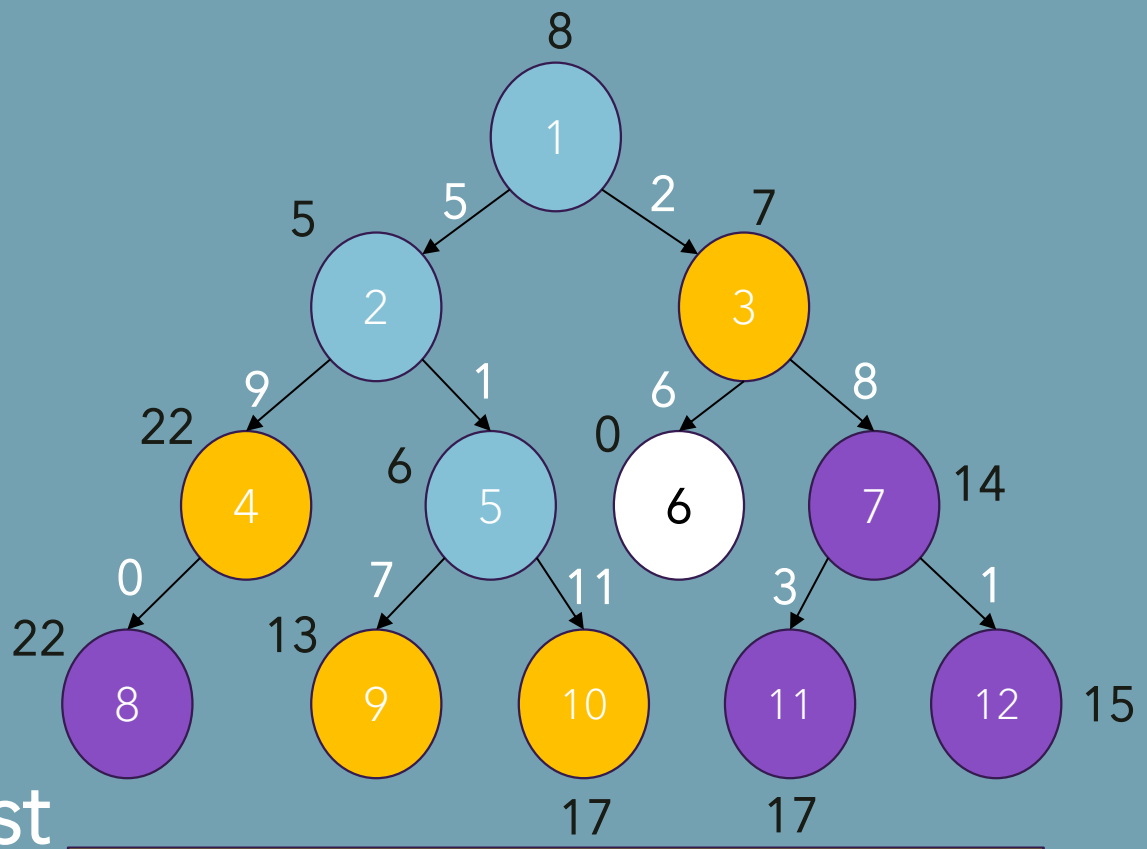


List Sort based on cost

Insert → 10 (17), 9 (13), 4 (22), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

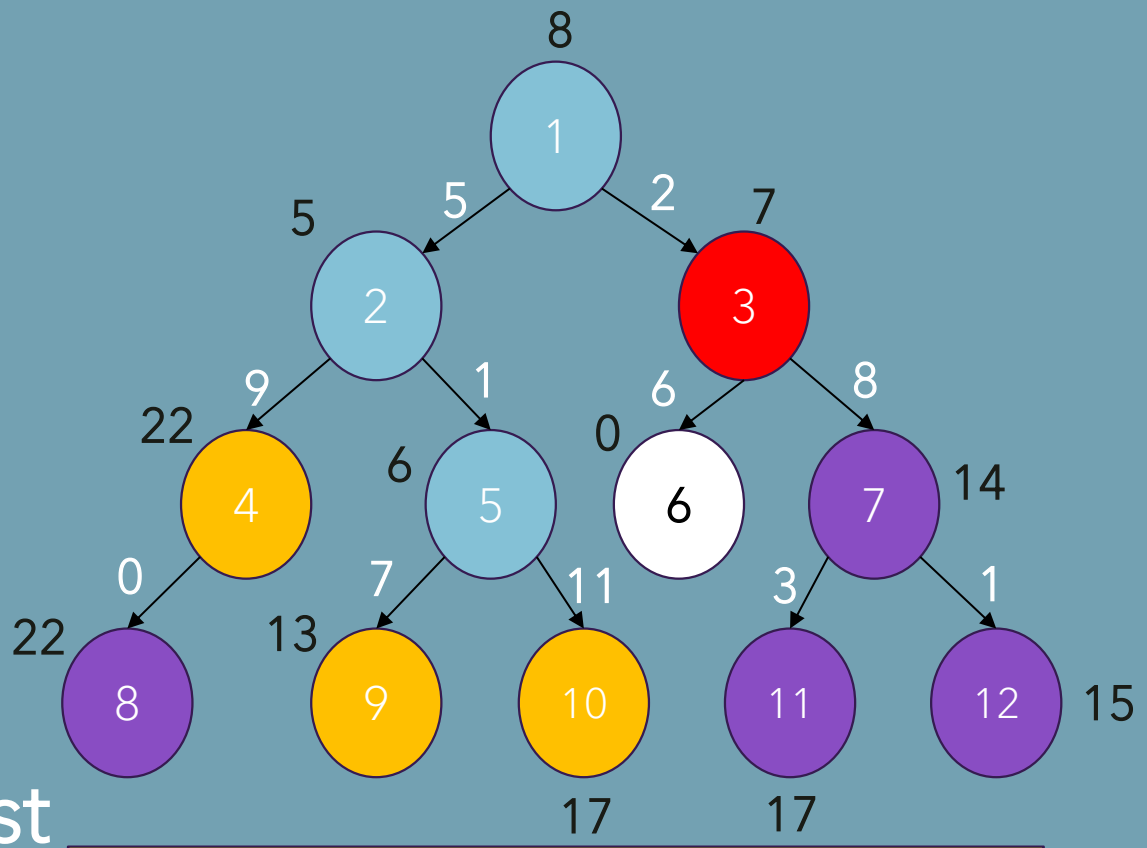


List

Insert → 4 (22), 10 (17), 9 (13), 3 (7) → Remove

BEST-FIRST SEARCH

Goal: 6

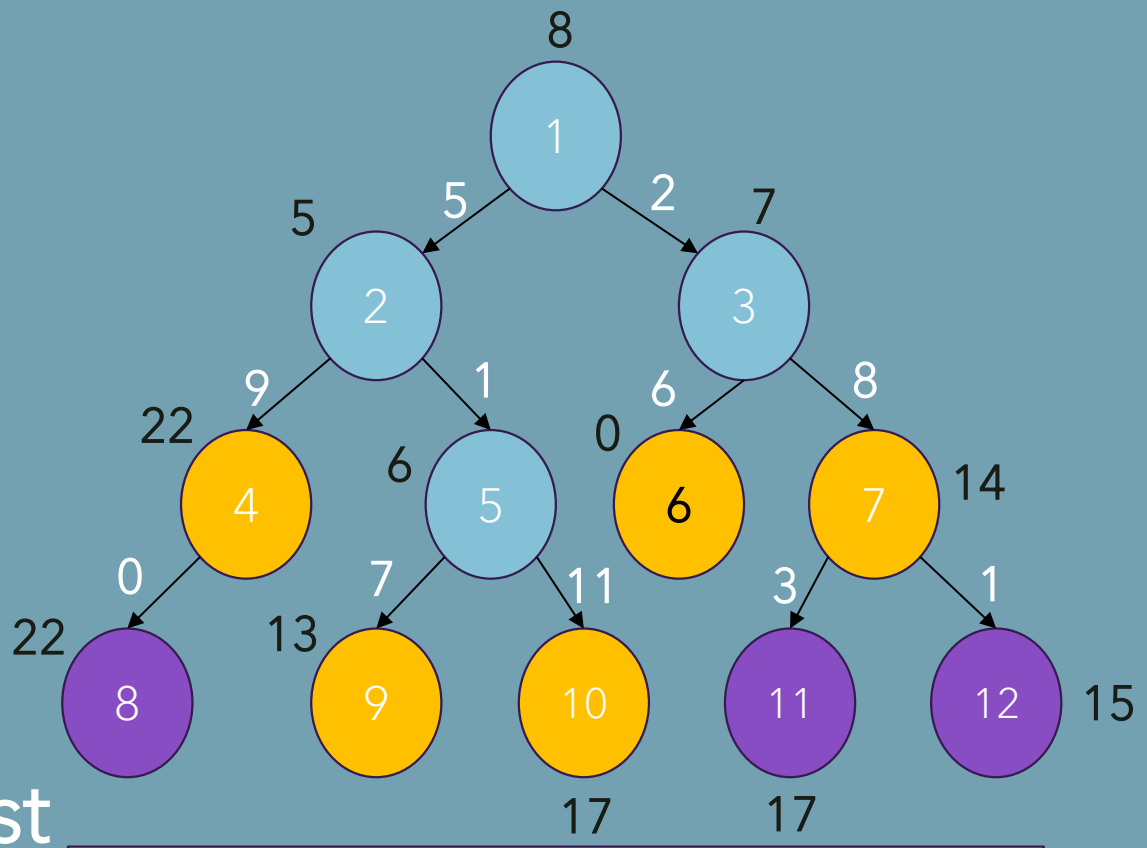


List

Insert → 4 (22), 10 (17), 9 (13) → Remove

BEST-FIRST SEARCH

Goal: 6

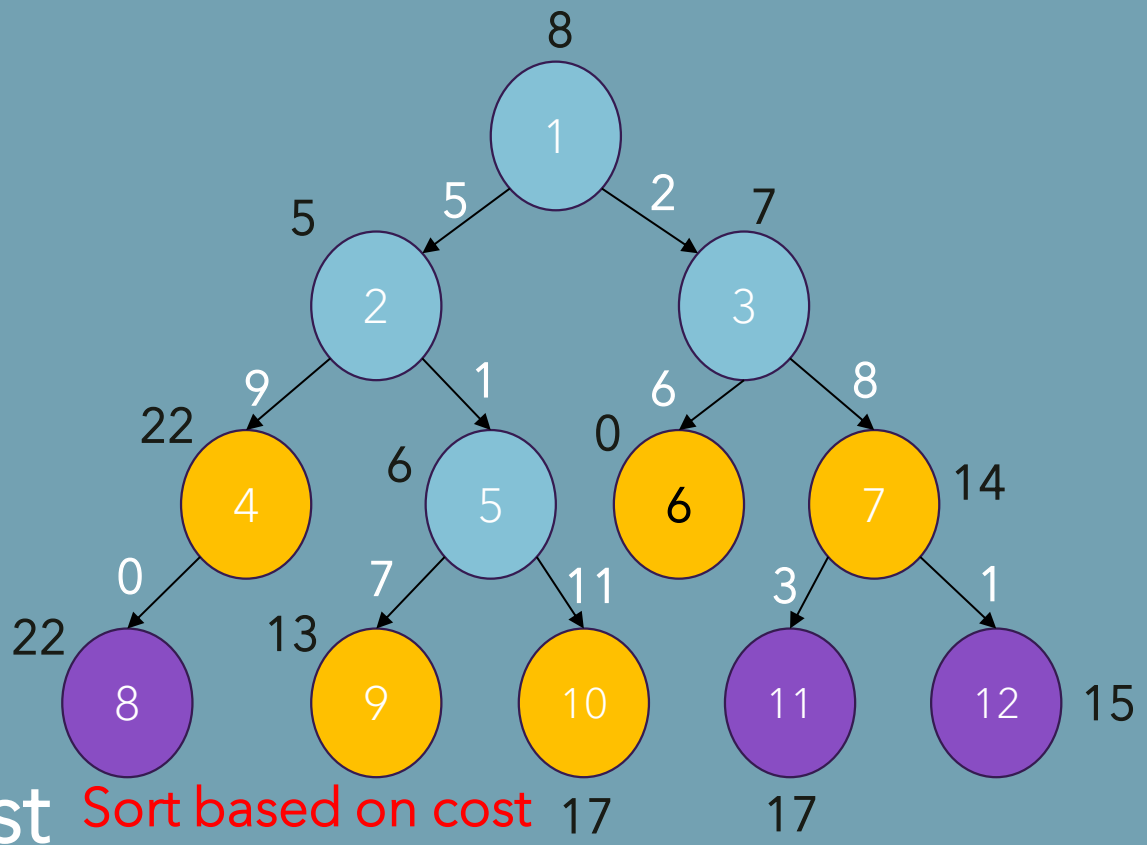


List

Insert → 7 (14), 6 (0), 4 (22), 10 (17), 9 (13) → Remove

BEST-FIRST SEARCH

Goal: 6

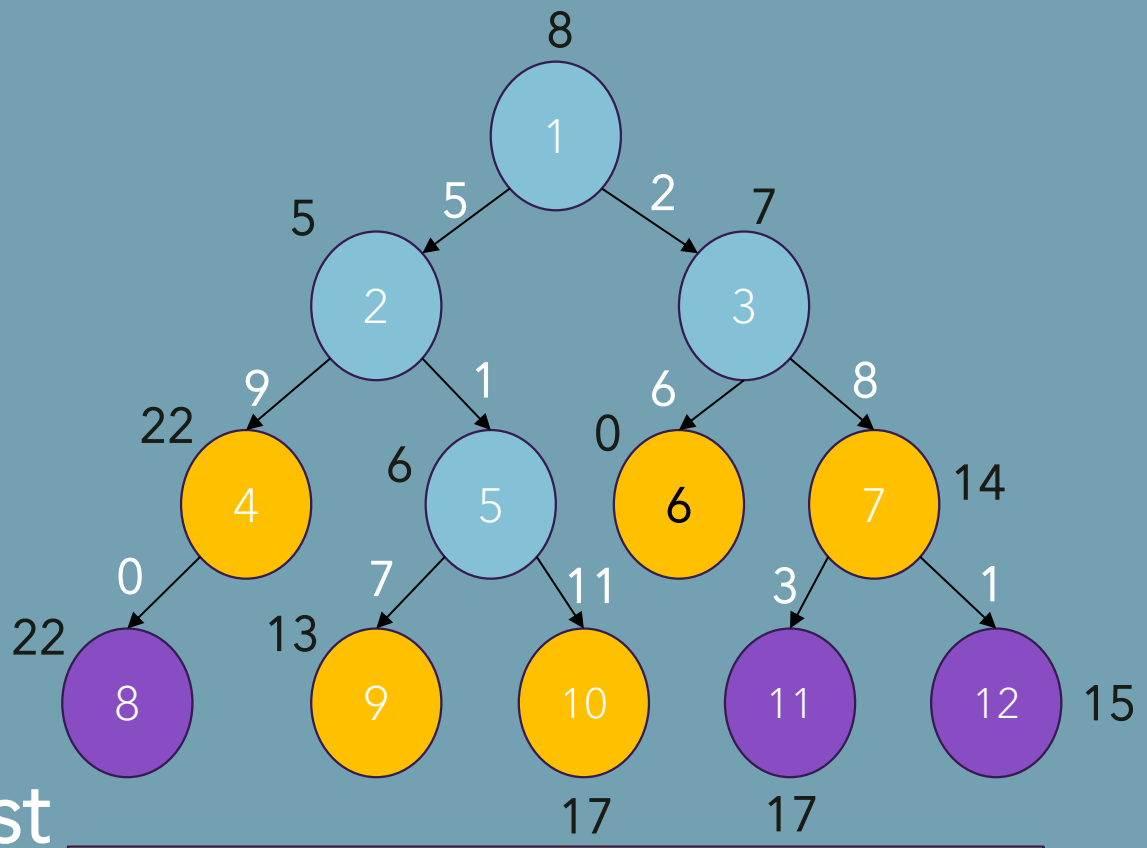


List Sort based on cost

Insert → 7 (14), 6 (0), 4 (22), 10 (17), 9 (13) → Remove

BEST-FIRST SEARCH

Goal: 6

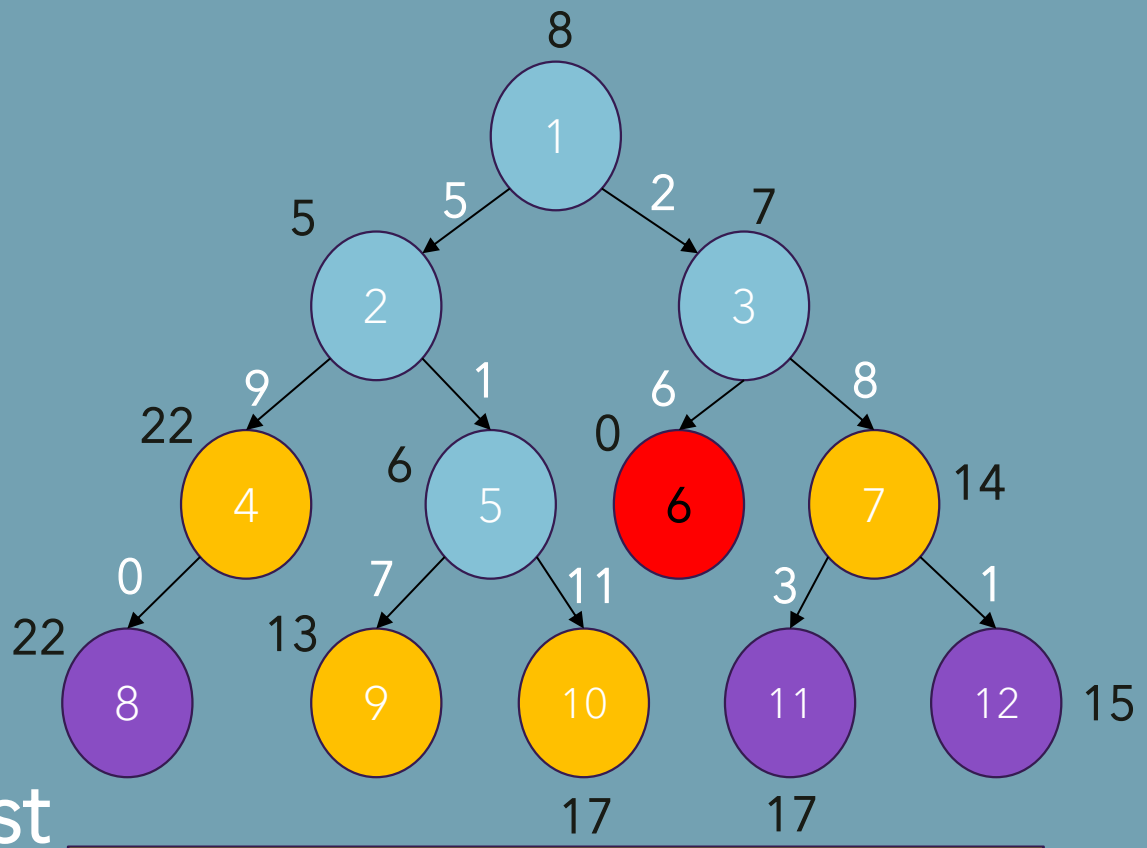


List

Insert → 4 (22), 10 (17), 7 (14), 9 (13), 6 (0) → Remove

BEST-FIRST SEARCH

Goal: 6

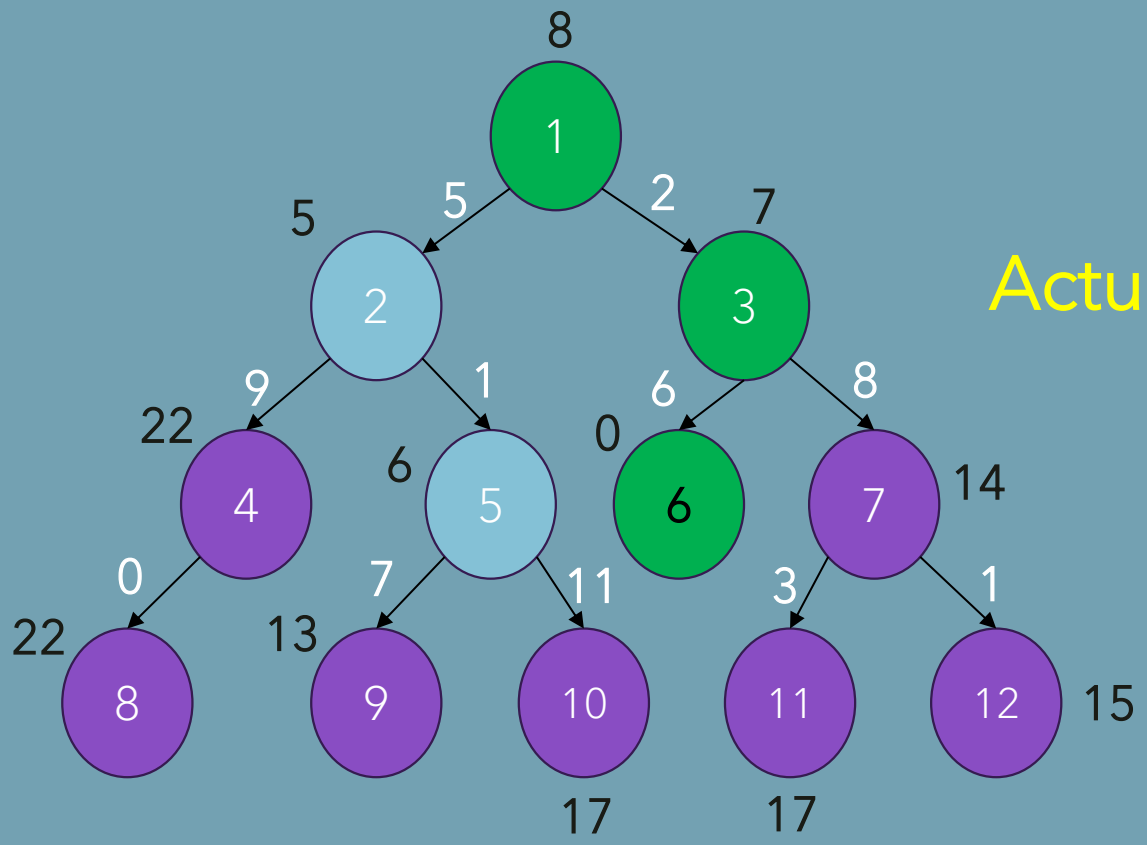


List

Insert → 4 (22), 10 (17), 7 (14), 9 (13) → Remove

BEST-FIRST SEARCH

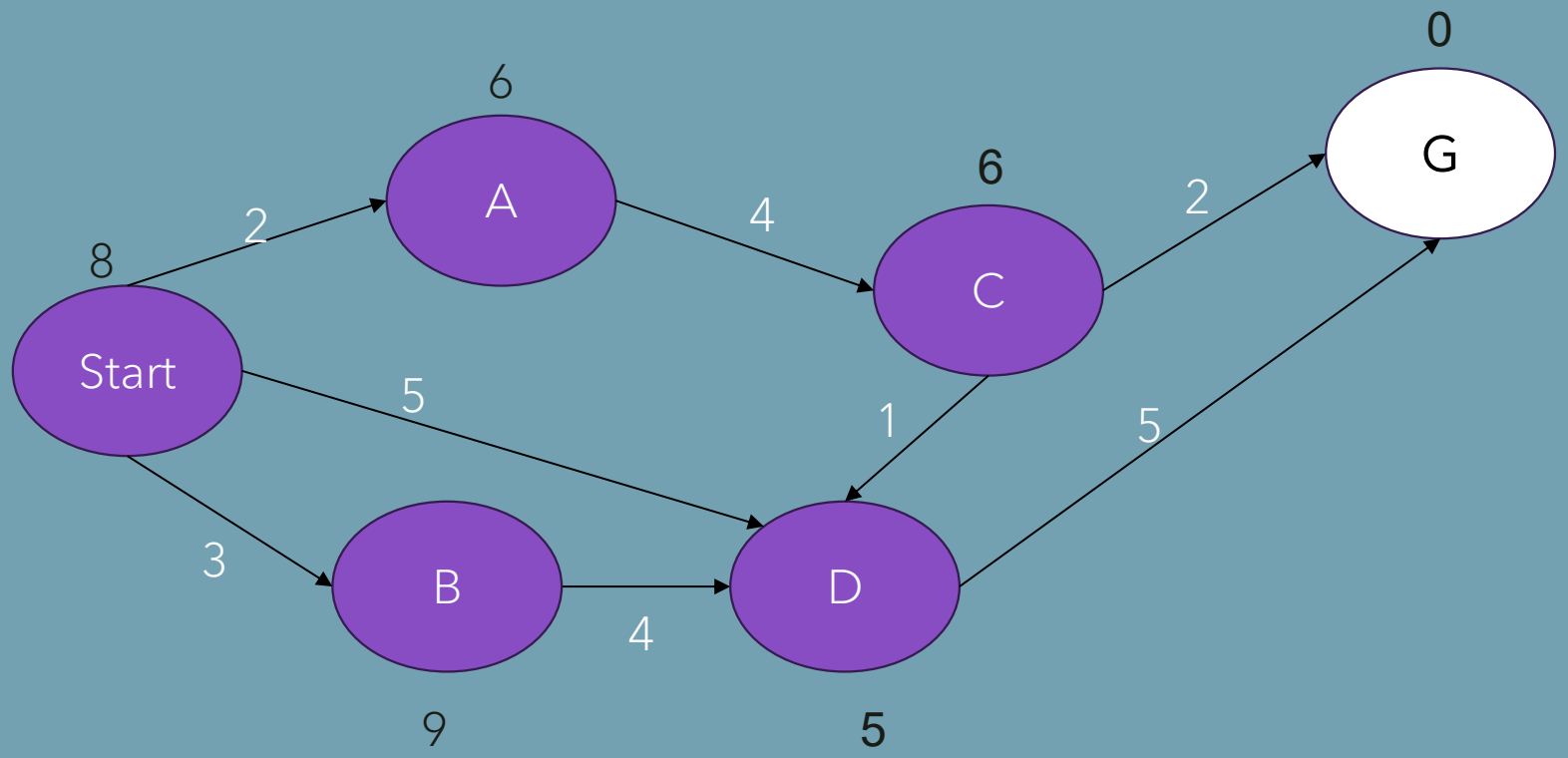
Goal: 6



Actual solution cost = 8

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

BEST-FIRST SEARCH (GRAPH)



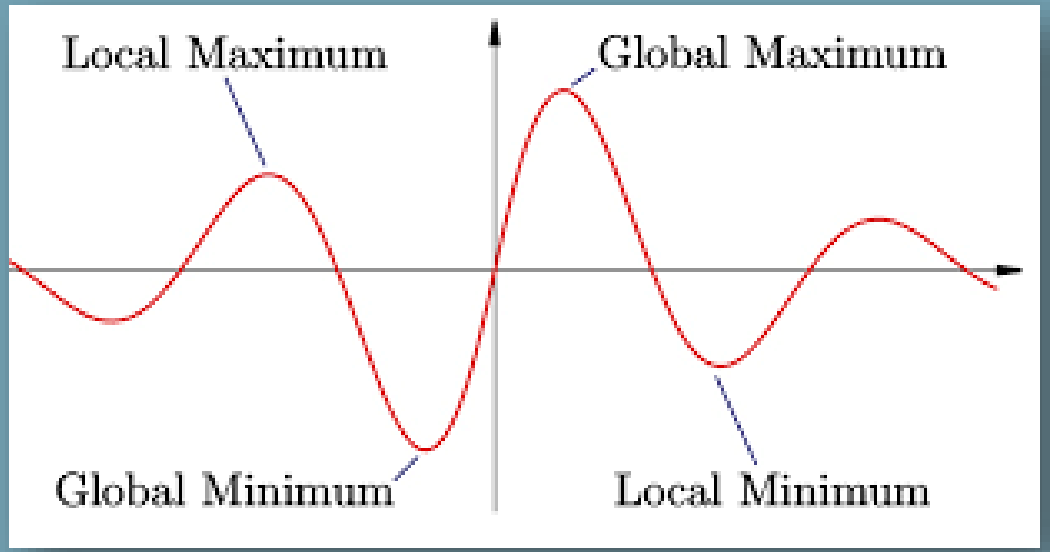
Current	FIFO FRONT < - - - - - - > REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S,D, G (0)], [S, A (6)], [S, B (9)]
[S, D, G (0)]	

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

GREEDY SEARCH (طمع/جشع)

- **Greedy search** is a form of Best-First Search that uses a **heuristic** to always expand the node that seems **closest to the goal**, based **ONLY ON THAT ESTIMATE without looking for the others.** It is like the running toward the goal without looking for anything else.

- In Greedy search, it always looking for the best/closet to the goal, which it seems you are reaching to it, but it may not actually be close to the goal at all (Trap). Which it is called **Local Minima**



In other words, Greedy search looking for the minimum but not the goal

GREEDY SEARCH (طمع/جشع)

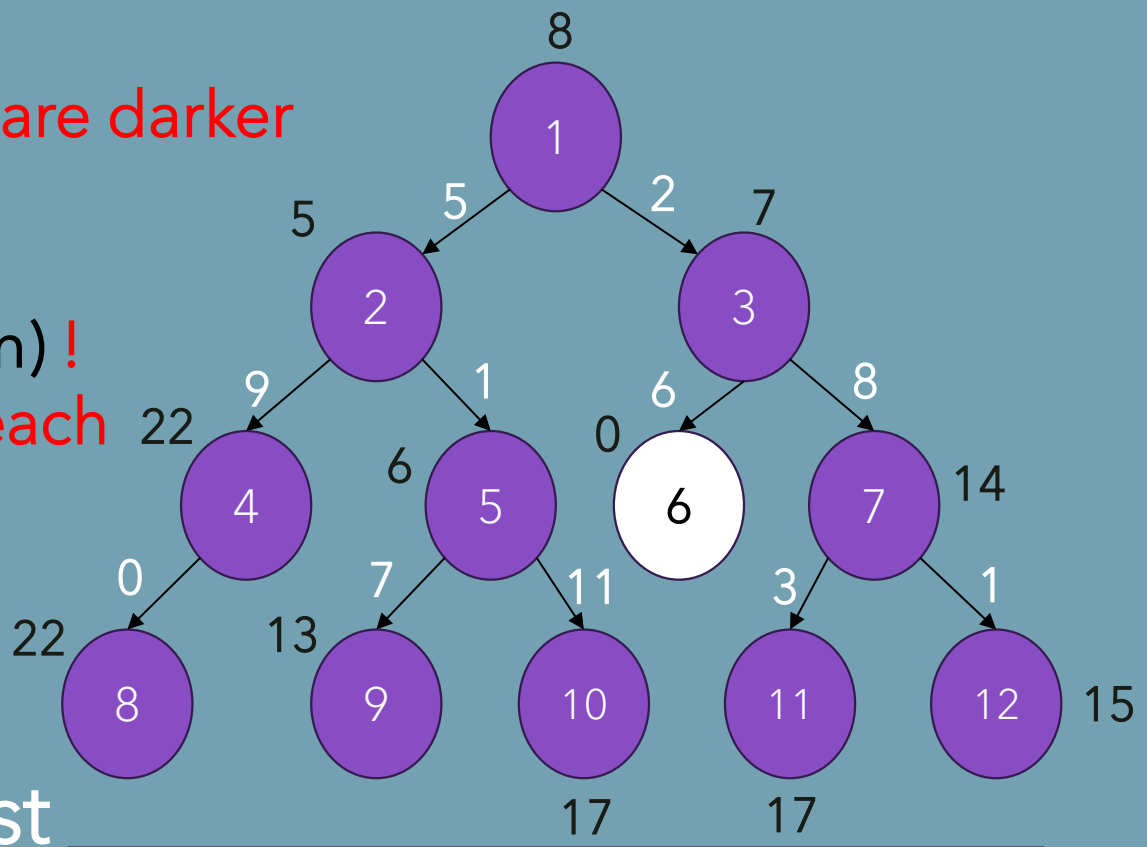
FIFO + Priority Function

Priority function is a data access principle that gives a priority for each element using FIFO, which the highest priority removed first. In Greedy Search, the priority for the lowest heuristic cost h(n) **(Sort)**, and the cost of each path is **not accumulative**. AND IN EACH EXPAND, CLEAR THE QUEUE.

GREEDY SEARCH

Goal: 6

Note:
The heuristic values are darker
Remember:
Pre-order traverse !
Priority for lowest h(n) !
Clear the queue in each expand !

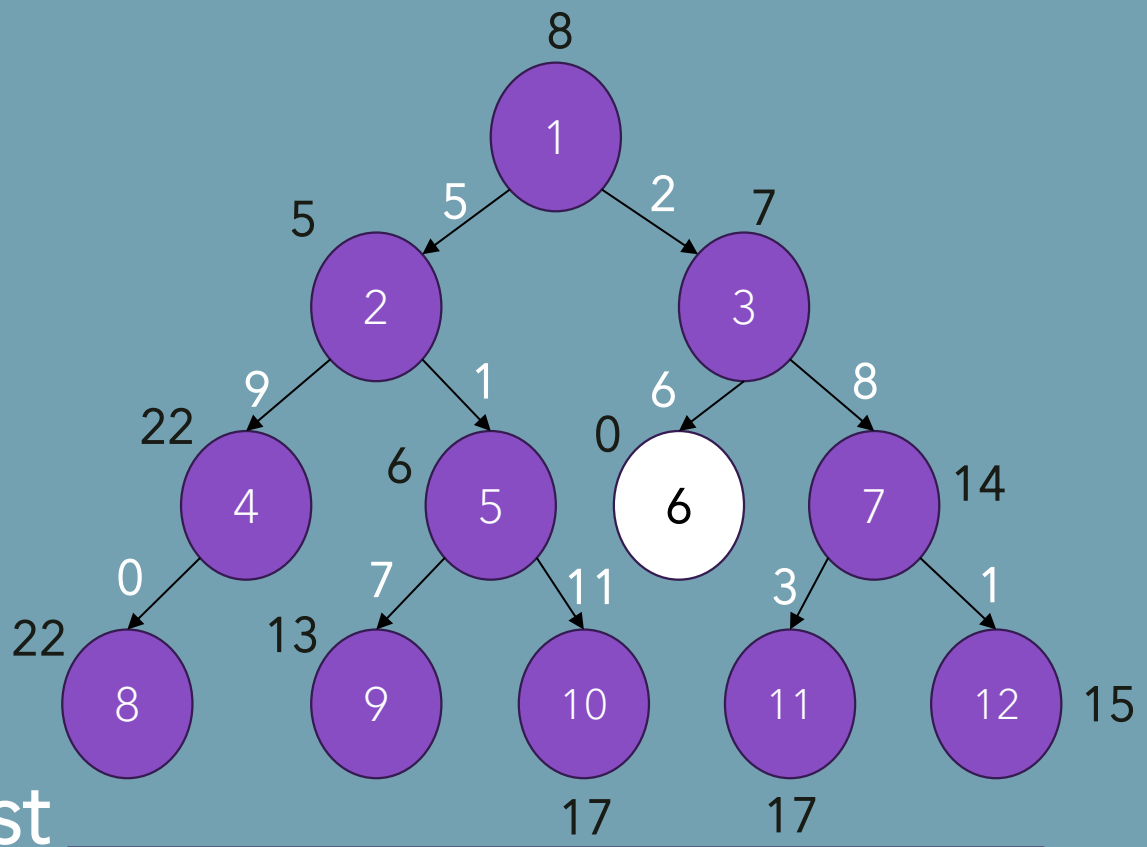


List

Insert → [ ] → Remove

GREEDY SEARCH

Goal: 6

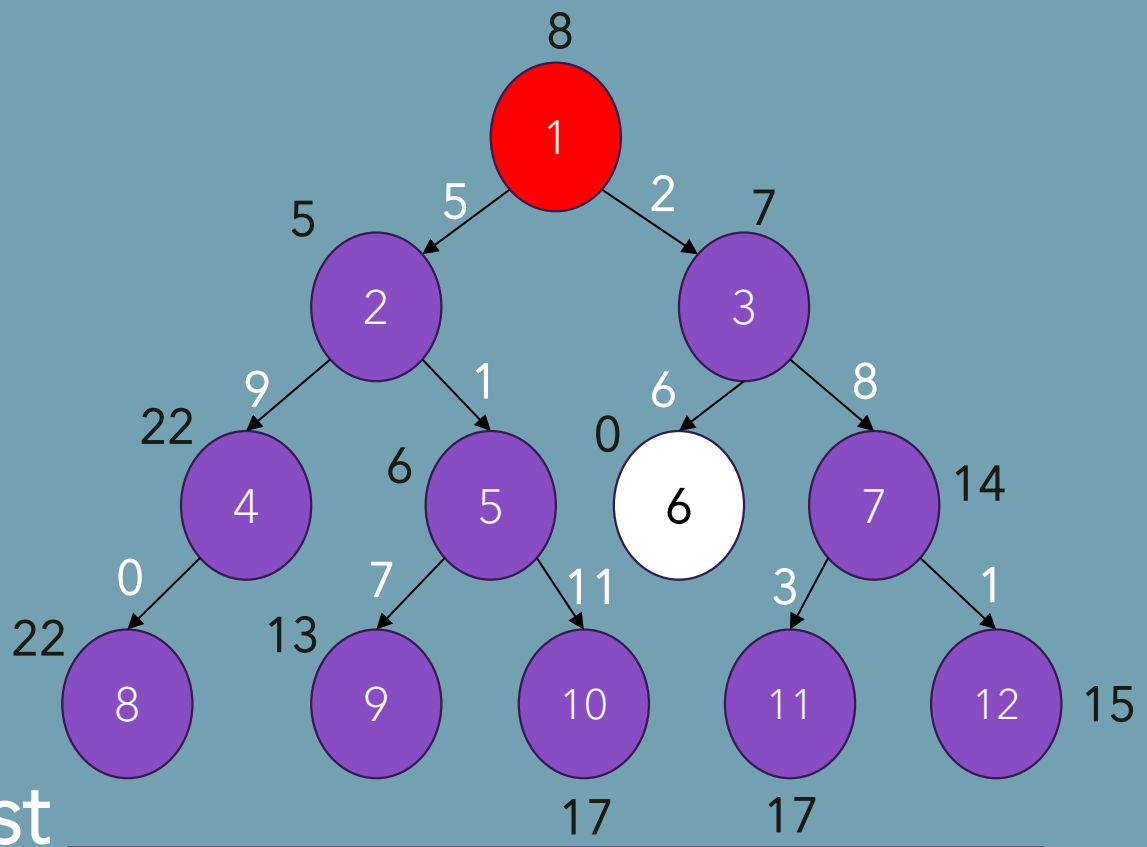


List

Insert → 1 (8) → Remove

GREEDY SEARCH

Goal: 6

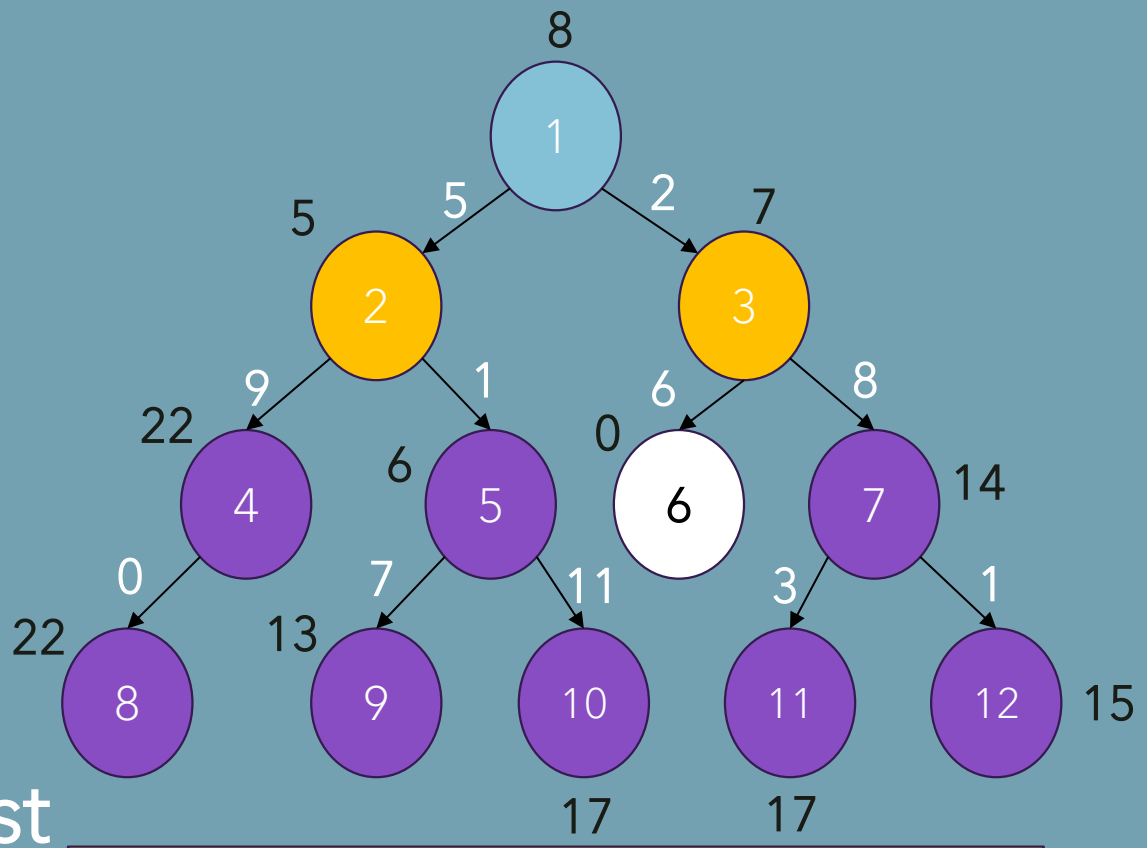


List

Insert → → Remove

GREEDY SEARCH

Goal: 6

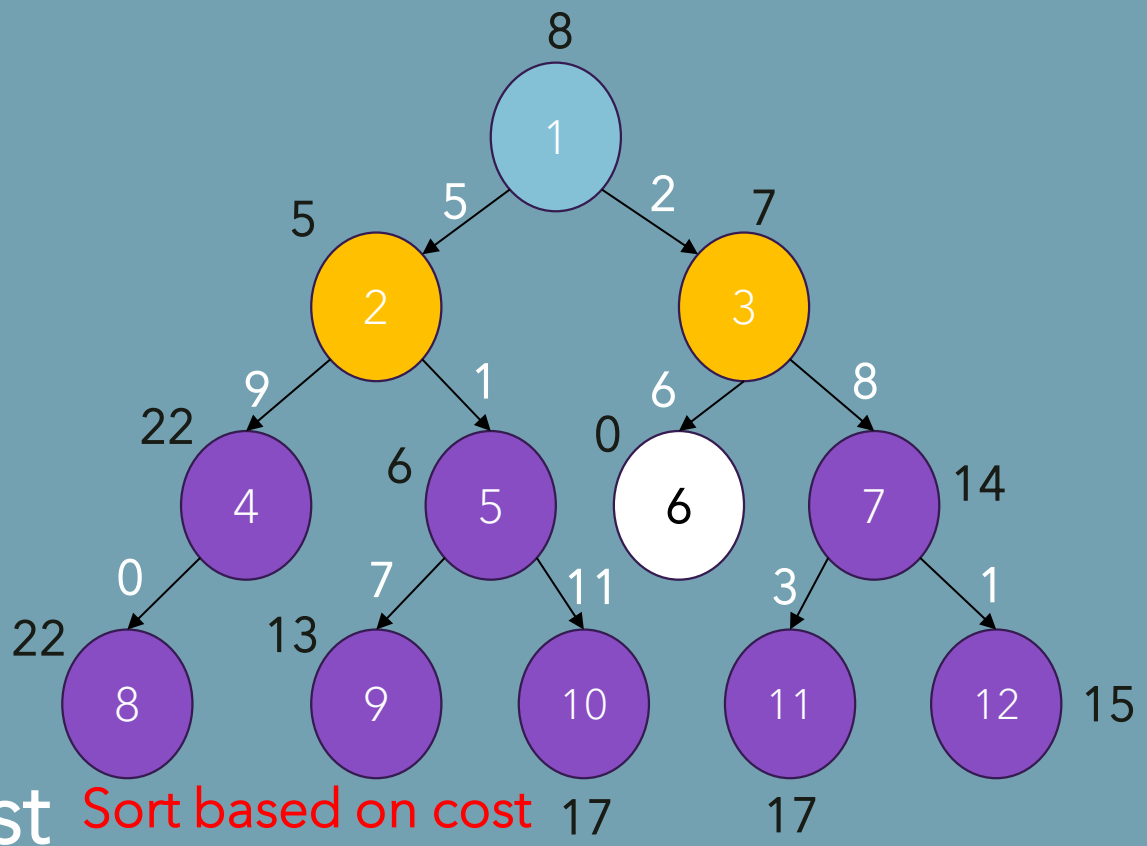


List

Insert → 3 (7), 2 (5) → Remove

GREEDY SEARCH

Goal: 6

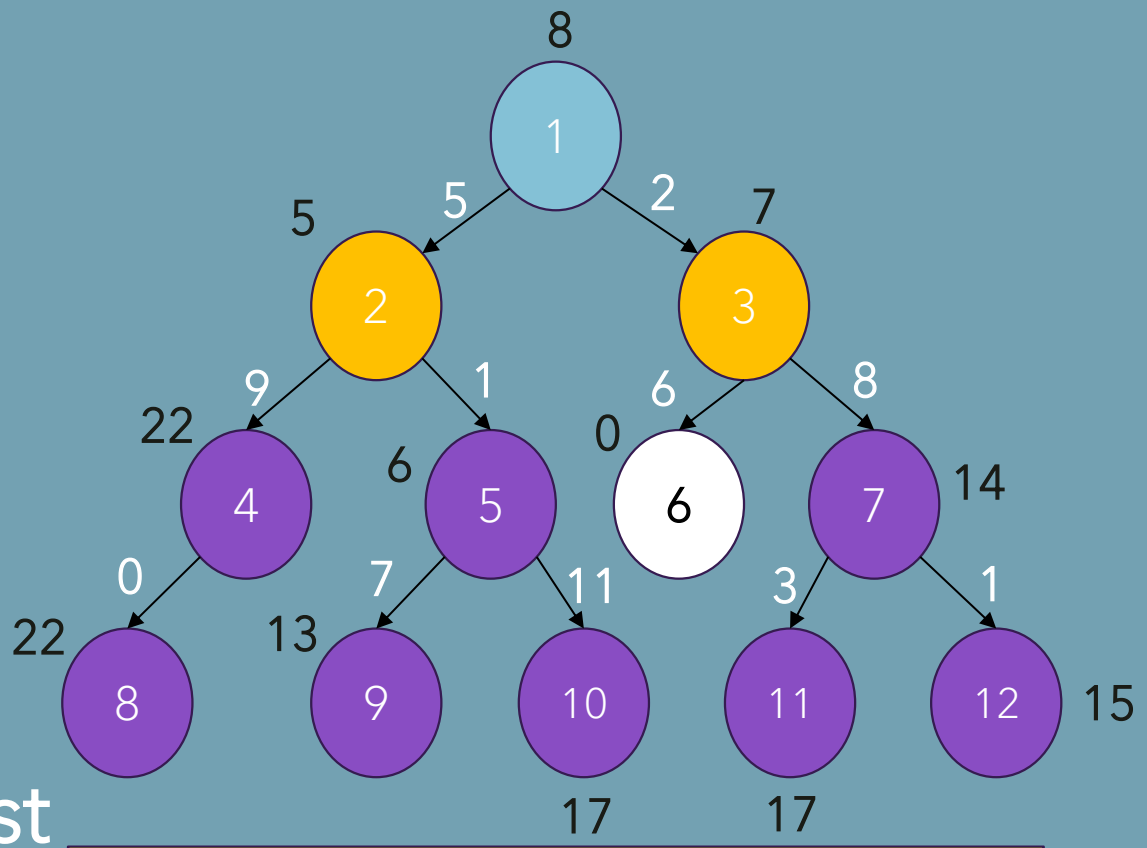


List Sort based on cost

Insert → 3 (7), 2 (5) → Remove

GREEDY SEARCH

Goal: 6



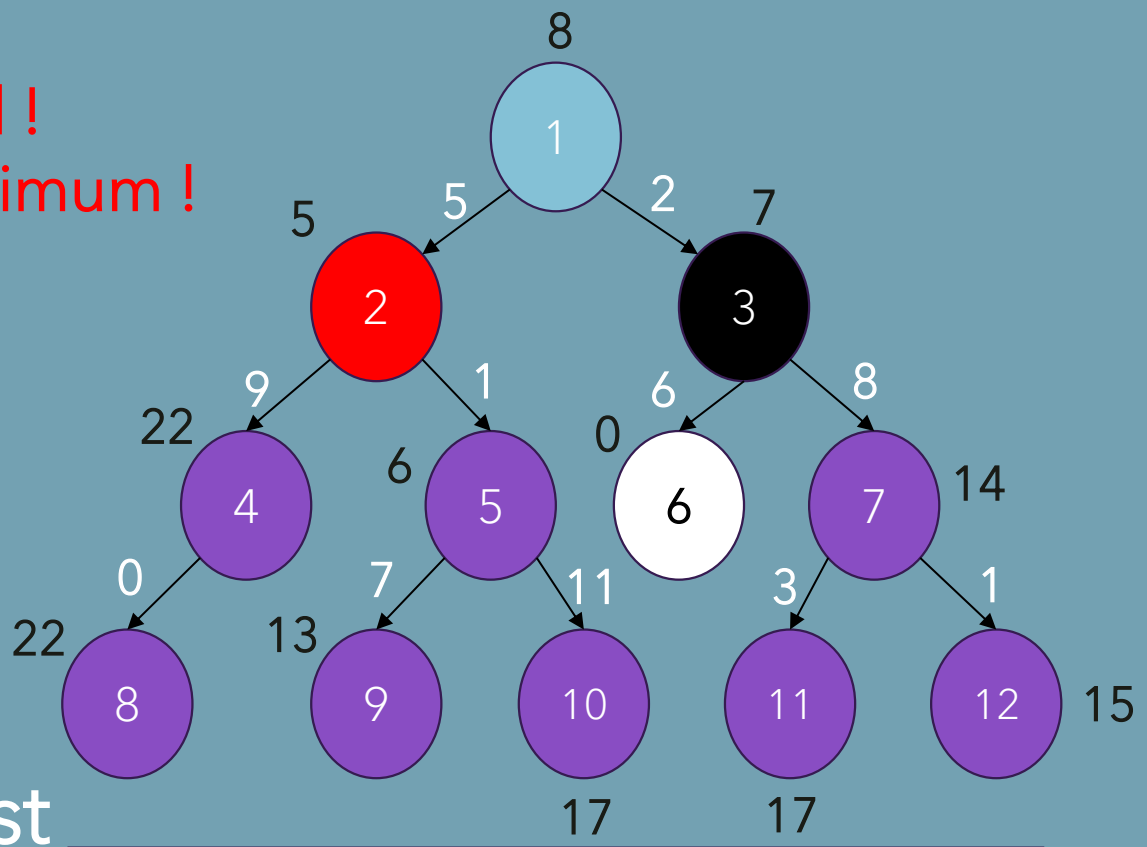
List

Insert → 3 (7), 2 (5) → Remove

GREEDY SEARCH

Goal: 6

The queue is cleared !
Run towards the minimum !

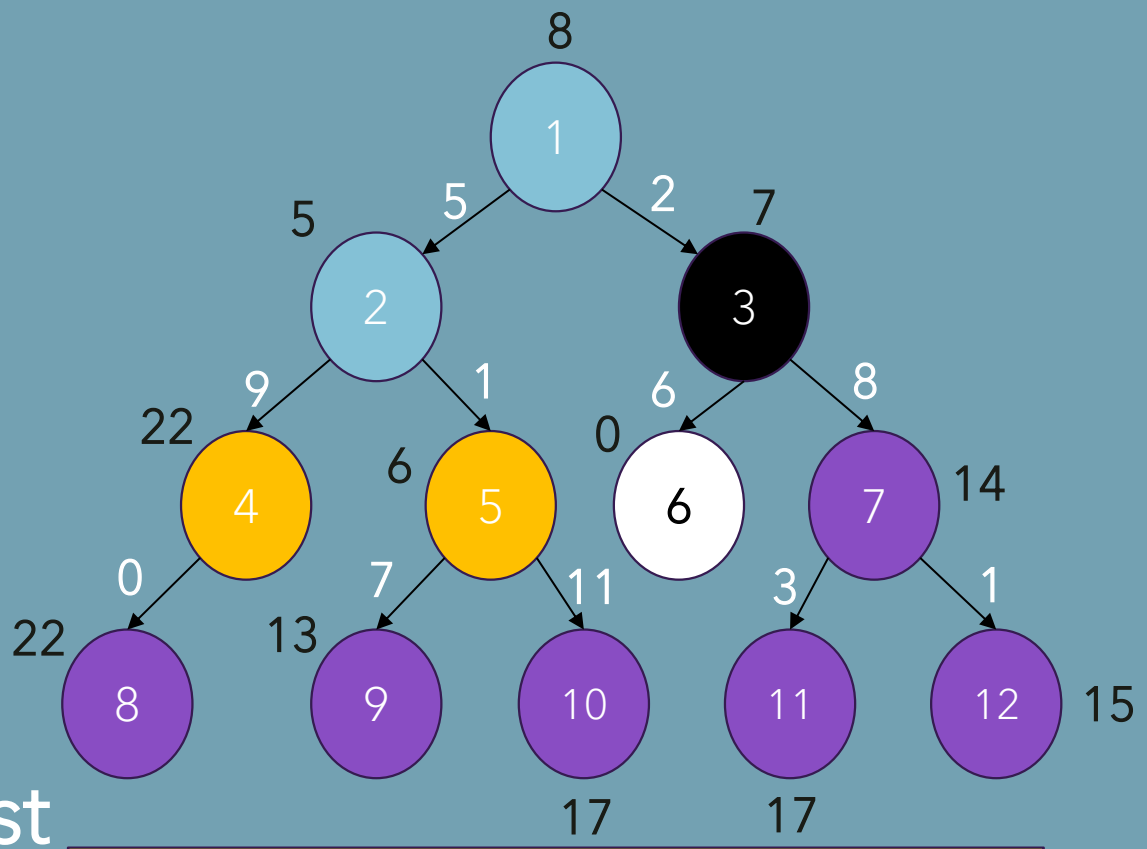


List

Insert → [ ] → Remove

GREEDY SEARCH

Goal: 6

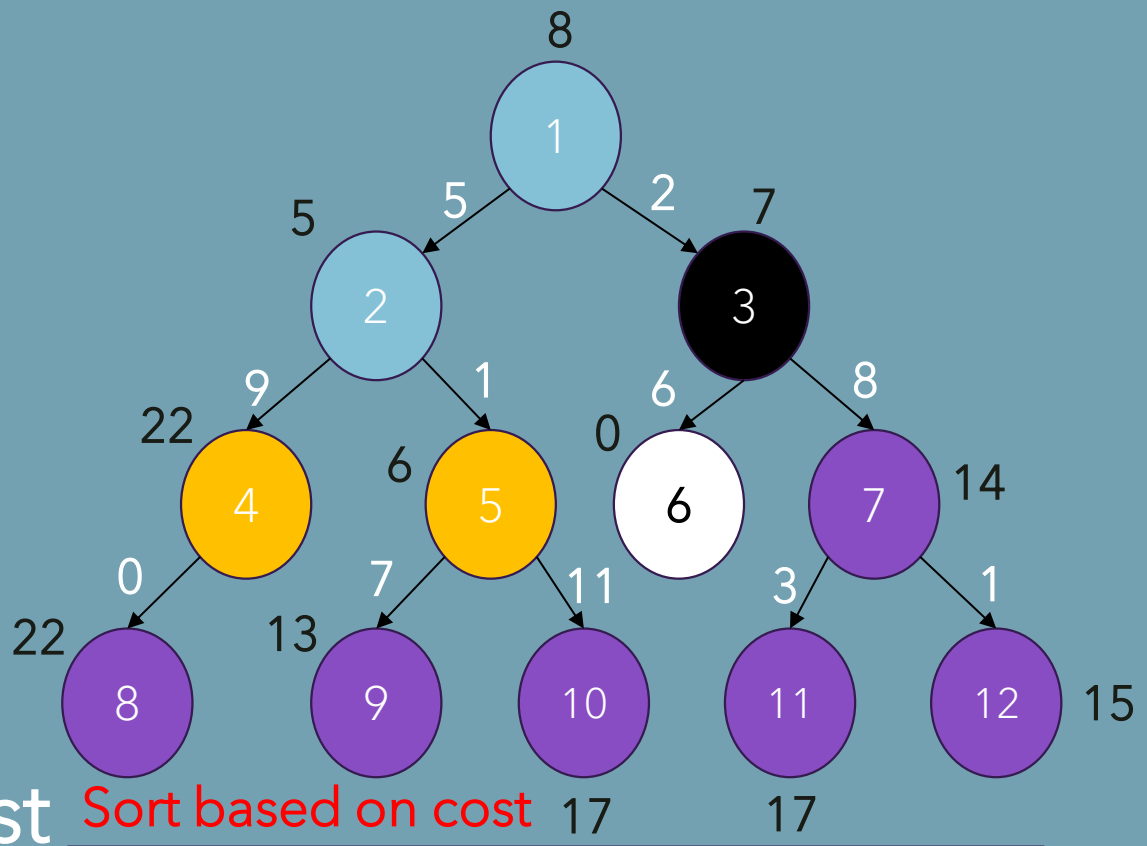


List

Insert → 5 (6), 4 (22) → Remove

GREEDY SEARCH

Goal: 6

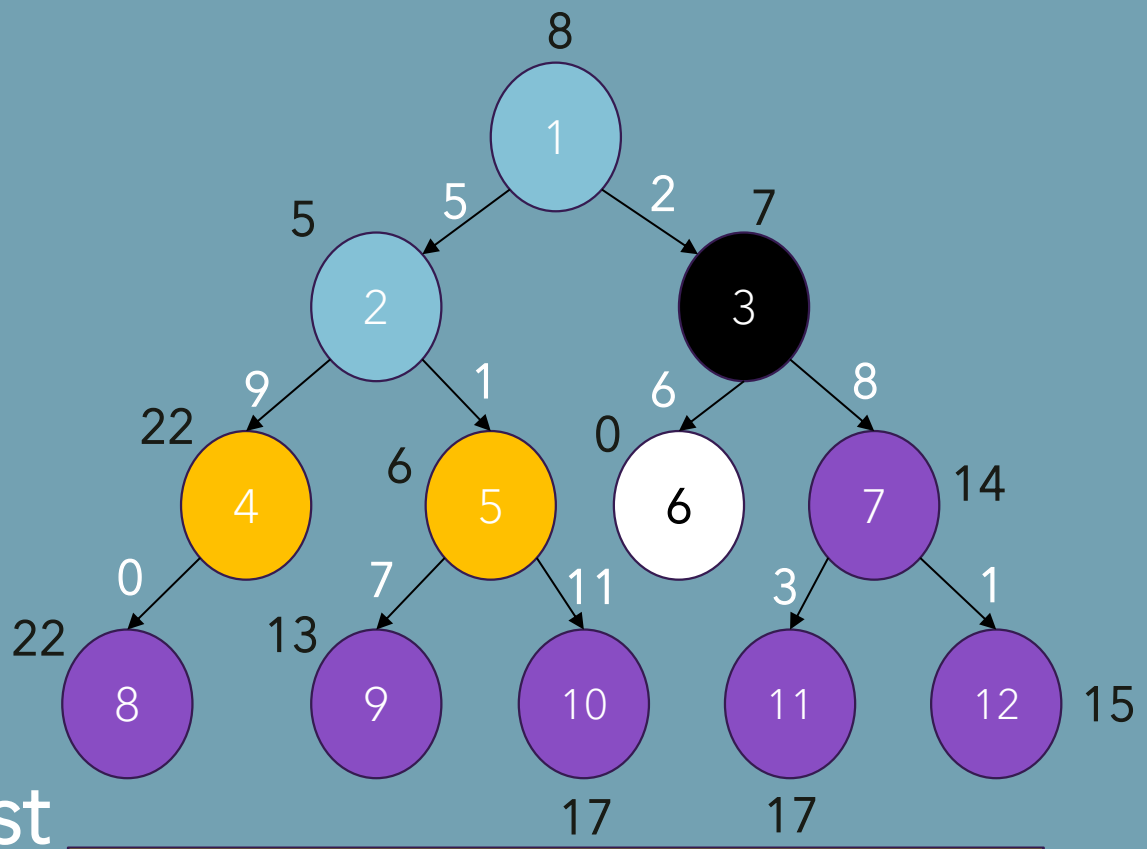


List Sort based on cost

Insert → 5 (6), 4 (22) → Remove

GREEDY SEARCH

Goal: 6

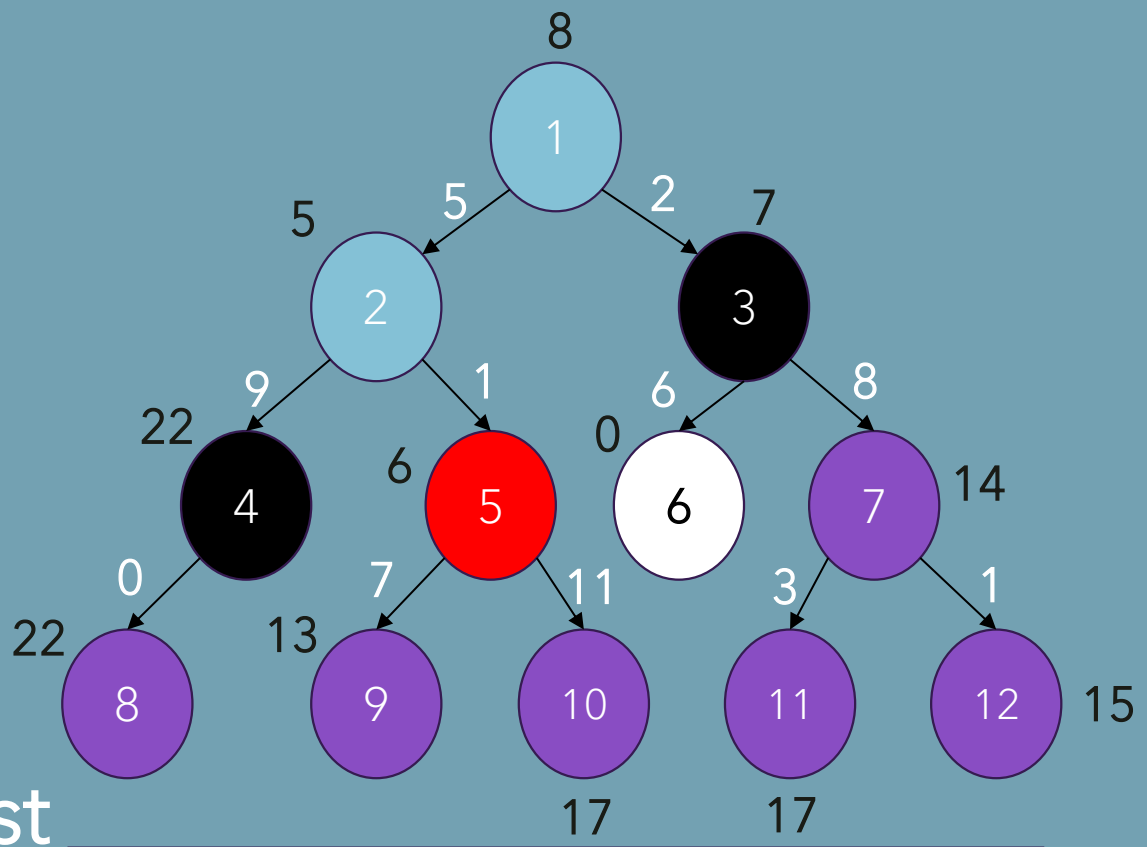


List

Insert → 4 (22), 5 (6) → Remove

GREEDY SEARCH

Goal: 6

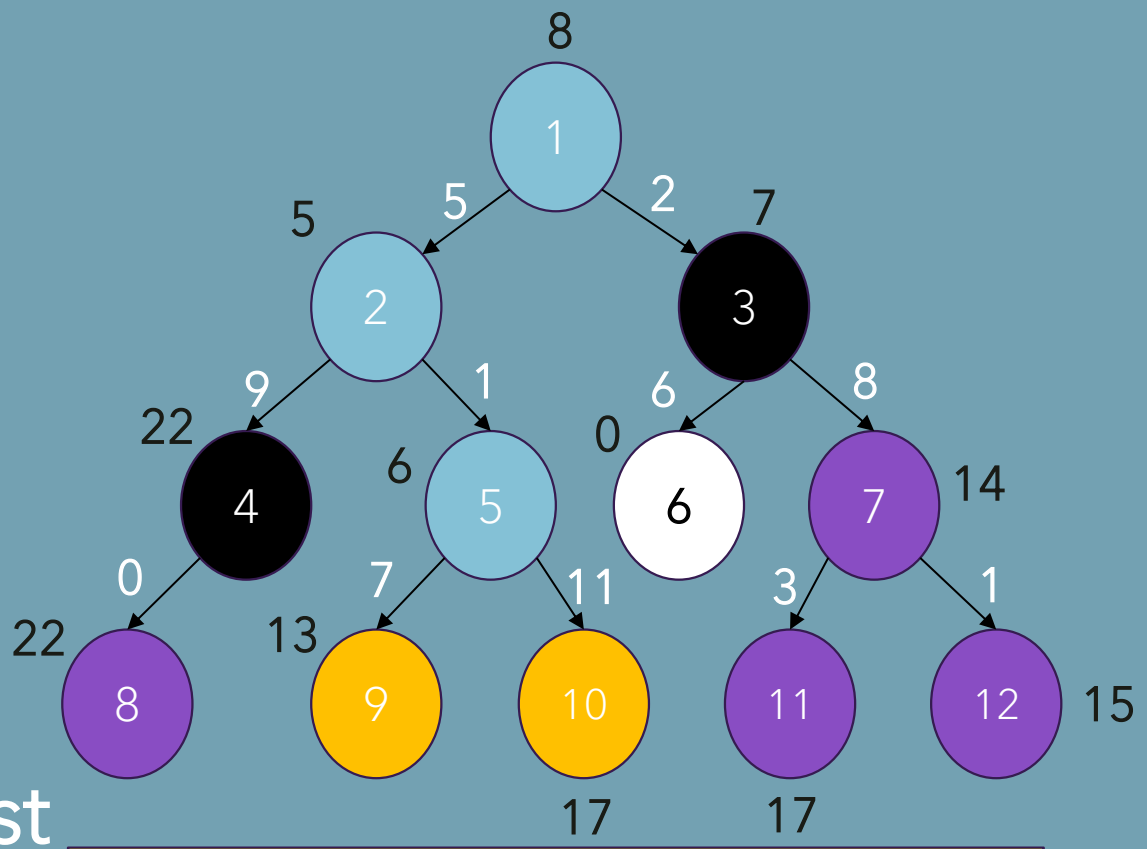


List

Insert → → Remove

GREEDY SEARCH

Goal: 6

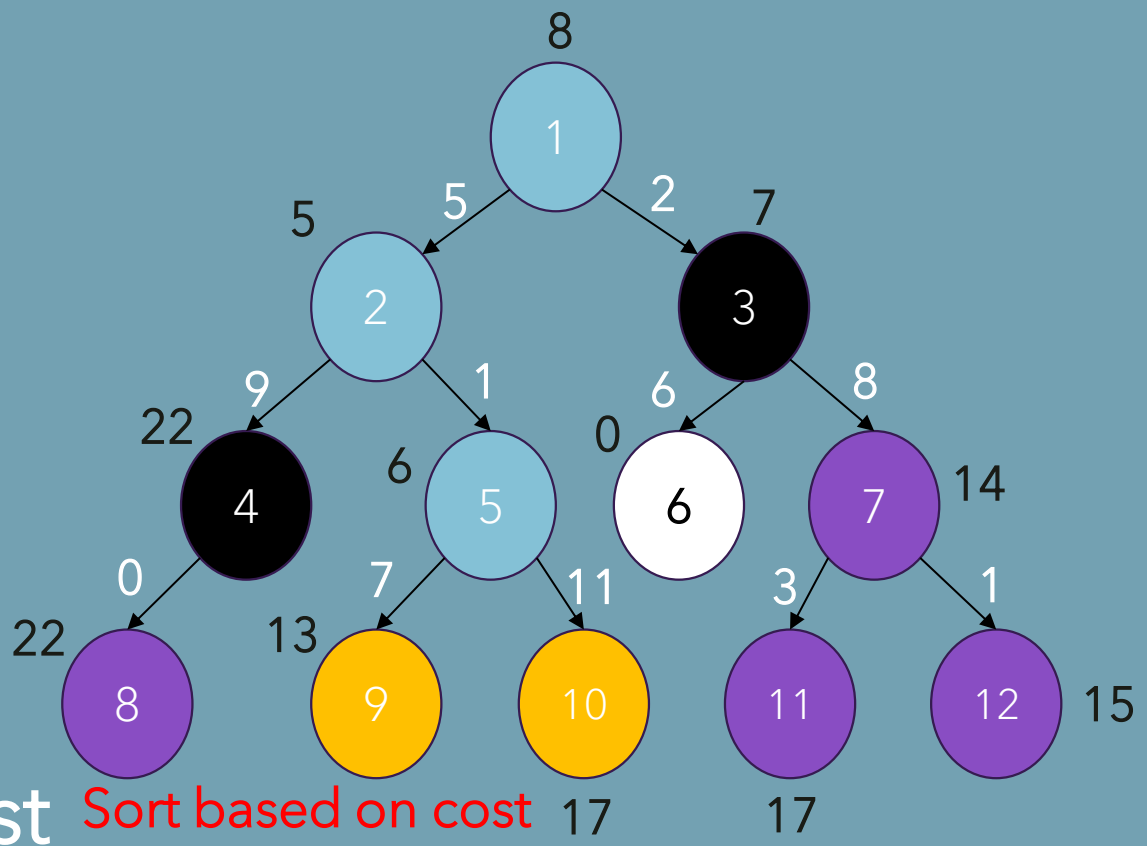


List

Insert → 10 (17), 9 (13) → Remove

GREEDY SEARCH

Goal: 6

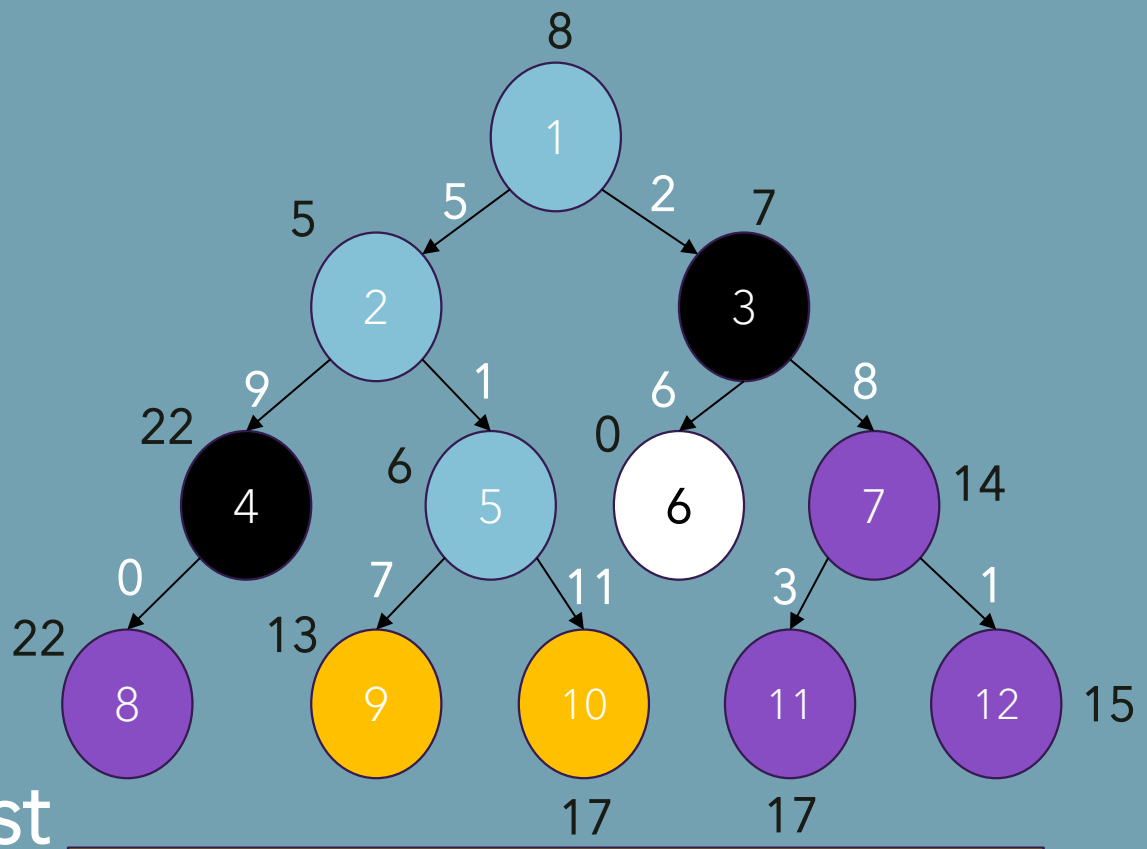


List Sort based on cost

Insert → 10 (17), 9 (13) → Remove

GREEDY SEARCH

Goal: 6

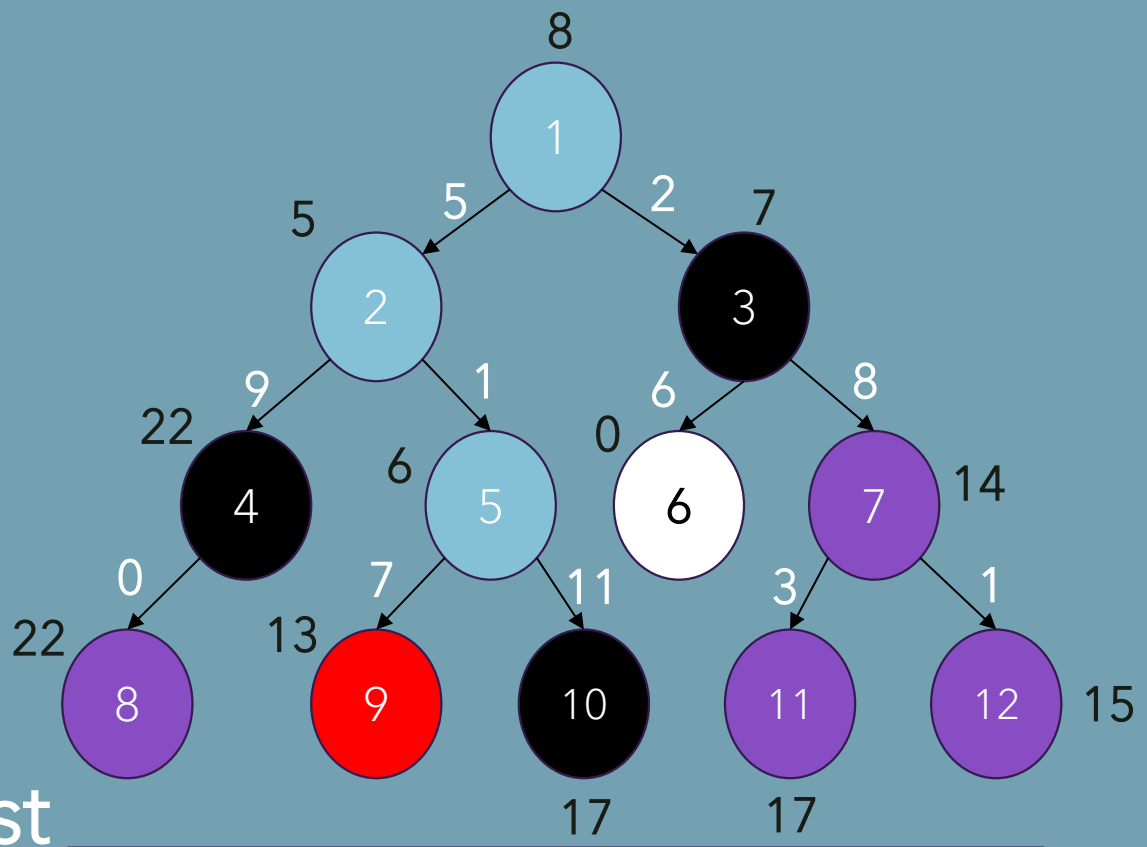


List

Insert → 10 (17), 9 (13) → Remove

GREEDY SEARCH

Goal: 6

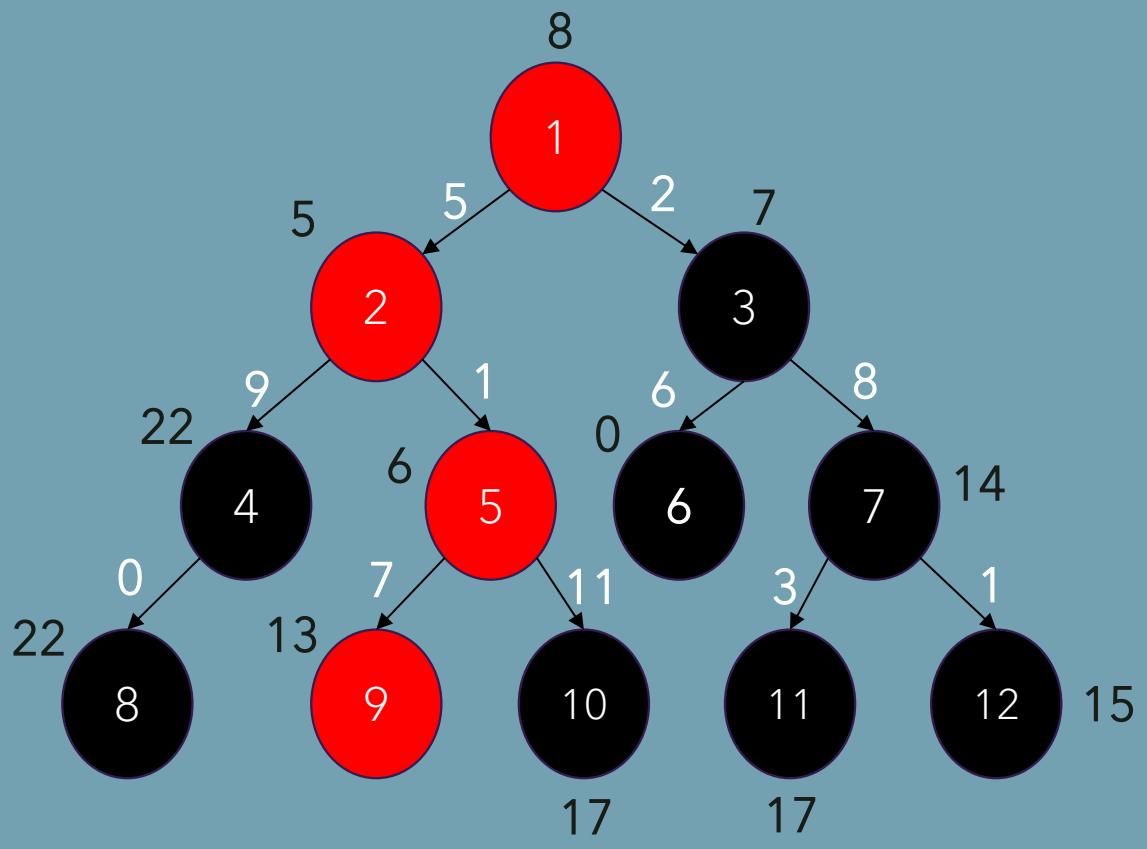


List

Insert → → Remove

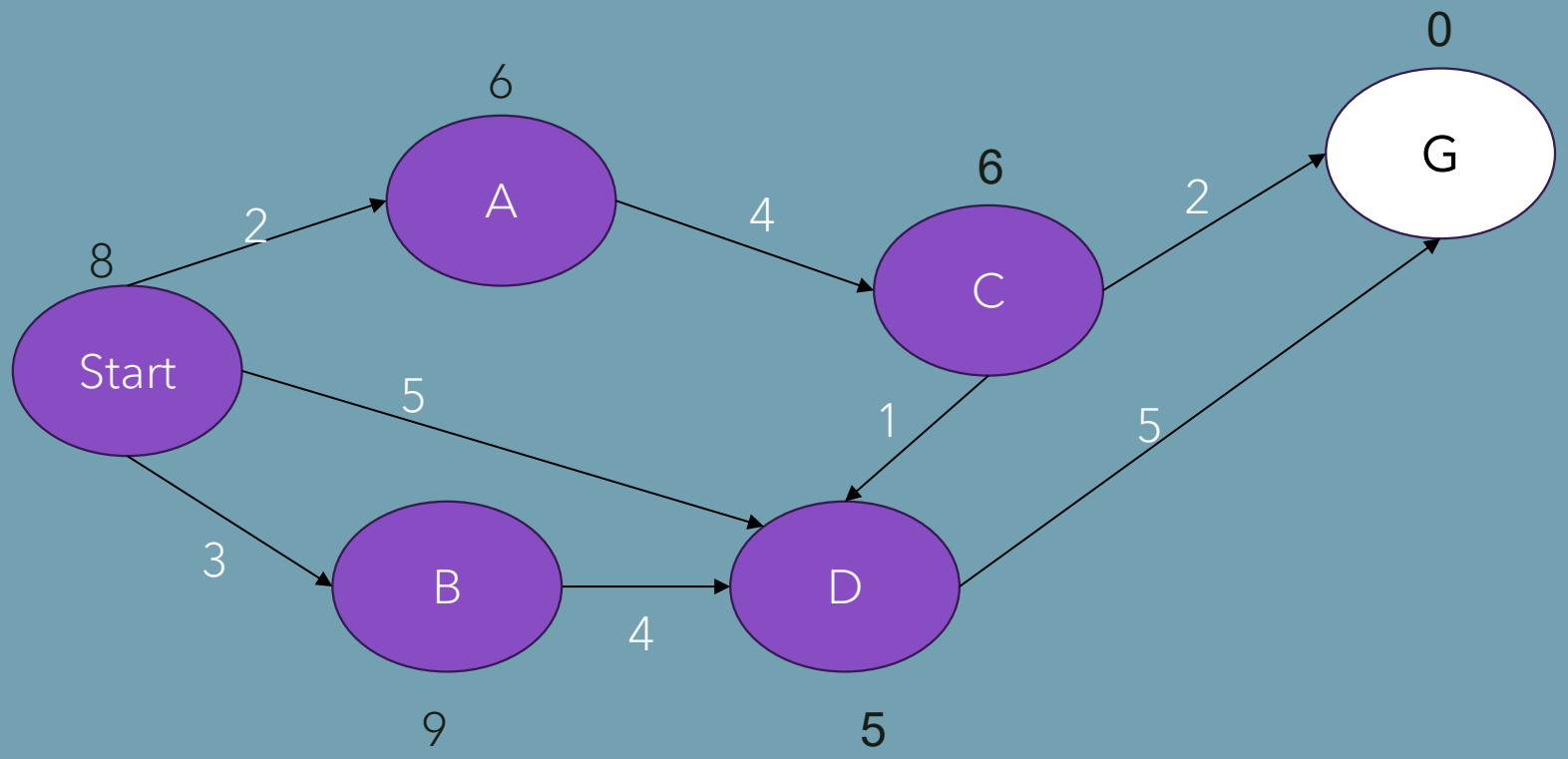
GREEDY SEARCH

Goal: 6



SEARCH FAILED !

GREEDY SEARCH (GRAPH)



Current	FIFO FRONT < - - - - - - > REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S, D, G (0)]
[S, D, G (0)]	

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

BEST-FIRST VS. GREEDY

Best-First

Current	FIFO FRONT < - - - - - - > REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S,D, G (0)], [S, A (6)], [S, B (9)]
[S, D, G (0)]	No. of steps = 3 Actual cost = 10

Greedy

Current	FIFO FRONT < - - - - - - > REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S, D, G (0)]
[S, D, G (0)]	No. of steps = 3 Actual cost = 10

Remember, Greedy search doesn't guarantee a solution (Not Complete)

IMPLEMENTATION

GRAPH CLASS (ADJACENCY LIST REP.)

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

        self.__h_flag = h_flag
        if h_flag == True:
            self.__h_flag = h_flag
            if root_h_val:
                self.__h_map = {list(self.__graph.keys())[0]: root_h_val}
            else:
                raise Exception("There should be an heuristic value for the root")
        elif self.__h_flag != None:
            raise Exception("h_flag parameter should be True or None")

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

        self.__directed = directed
        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, h_val=None):
    if vertex not in self.__graph.keys():
        self.__graph[vertex] = []
        print("Vertex is added successfully !")
        if self.__h_flag:
            if h_val is not None:
                self.__h_map[vertex] = h_val
            else:
                raise Exception("There should be an heuristic value for the vertex")
    else:
        print("This vertex does exist in the graph already !")
```

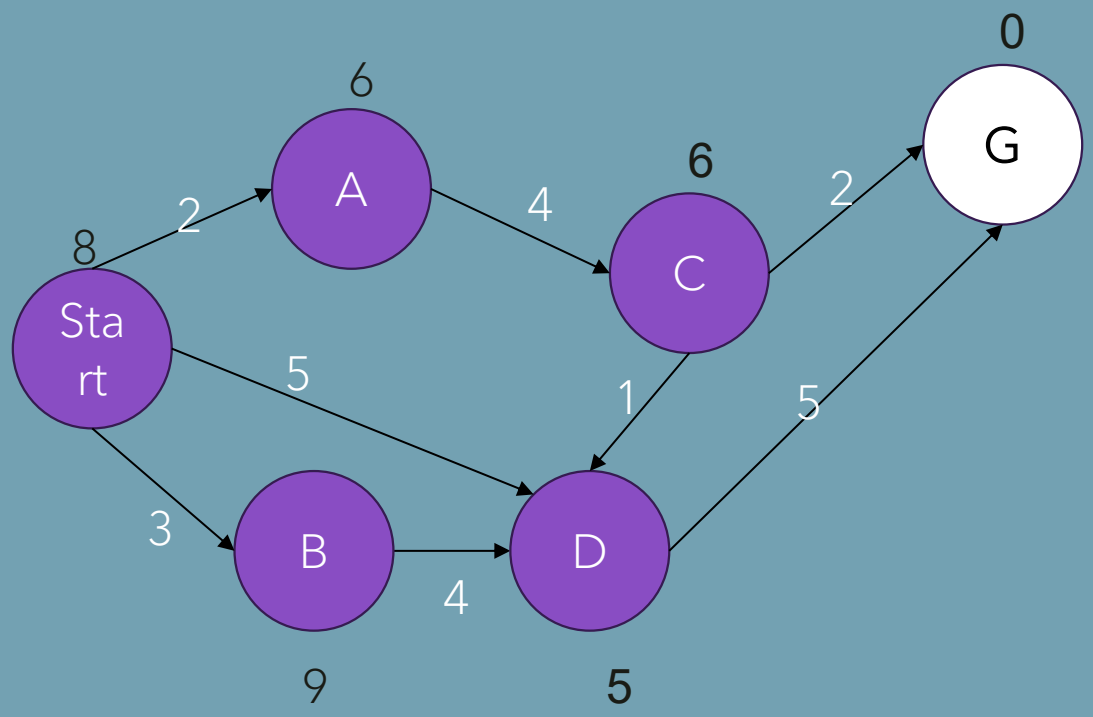
GRAPH CLASS (ADD EDGES METHOD)

```
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, self.__h_map
```

FORMULATE THE PROBLEM




```
my_graph = Graph("S" , directed=True, weighted=True, h_flag=True, root_h_val=8)
my_graph.add_vertex("A", 6)
my_graph.add_vertex("B", 9)
my_graph.add_vertex("C", 6)
my_graph.add_vertex("D", 5)
my_graph.add_vertex("G", 0)
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': []}, {'S': 8, 'A': 6, 'B': 9,
'C': 6, 'D': 5, 'G': 0})
```

GRAPH CLASS (BEST-FIRST METHOD)

```python
def Best_first(self, goal, start=None):
    if self.__h_flag is None:
        raise Exception("Best First Search only works with Graphs with Heuristic values !")

    if start != None:
        lst = [[start]]
    else:
        lst = [["root"]]

    visited = []
    while lst:
        # Sort all elements alphabetically to get a unified answer !
        # Sorting based on heuristic cost and alphabetical order to follow "FIFO + Priority" principle
        lst.sort(key=self.__heuristic_cost)
        # 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:
            adjacent_nodes = self.__graph[current_node]
            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 !")
```

```python
def __heuristic_cost(self, path):
    return self.__h_map[path[-1]], path[-1]
```

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 (GREEDY METHOD)

```python
def Greedy_search(self, goal, start=None):
    if self.__h_flag is None:
        raise Exception("Greedy Search only works with Graphs with Heuristic values !")

    if start != None:
        lst = [[start]]
    else:
        lst = [["root"]]

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

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

        if current_node == goal:
            return current_path, visited
        else:
            adjacent_nodes = self.__graph[current_node]
            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)

    return "Search Failed"
```

```python
def __heuristic_cost(self, path):
    return self.__h_map[path[-1]], path[-1]
```

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

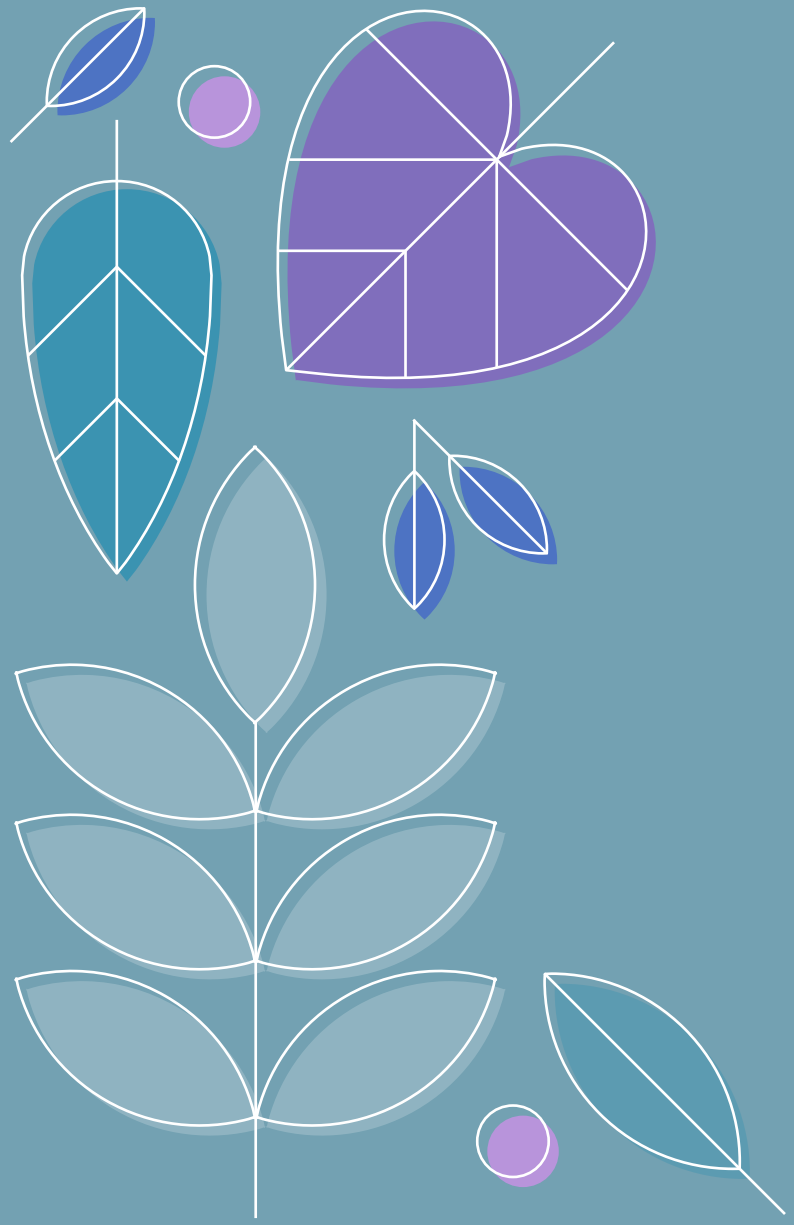
```
[3]: print(my_graph.Best_first("G", start="S"))
     print(my_graph.Greedy_search("G", start="S"))

     (['S', 'D', 'G'], ['S', 'D', 'G'])
     (['S', 'D', 'G'], ['S', 'D', 'G'])
```

Best-First

Current	FIFO FRONT <------> REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S,D, G (0)], [S, A (6)], [S, B (9)]
[S, D, G (0)]	No. of steps = 3 Actual cost = 10

Greedy

Current	FIFO FRONT <------> REAR
	[S (8)]
[S (8)]	[S,D (5)], [S,A (6)], [S,B (9)]
[S,D (5)]	[S, D, G (0)]
[S, D, G (0)]	No. of steps = 3 Actual cost = 10



NEXT LAB: SEARCHING PROBLEMS (4)

Thank you