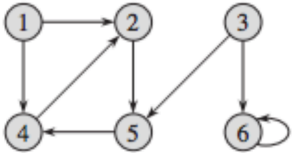
2521 Graph Theory

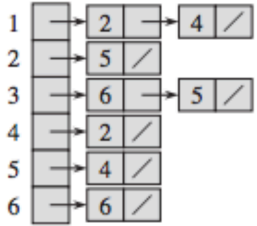
CPMSoc x CSESoc

Attendance form :D

Representation of graphs



(a)



(b)

	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

(c)

Breadth-First Search (BFS)

- **Concept:**
  - Start from a node.
  - Visit all its neighbors.
  - Move to the next level neighbors.

Breadth-First Search (BFS)

- **Concept:**
  - Start from a node.
  - Visit all its neighbors.
  - Move to the next level neighbors.

- **Algorithm Steps:**
  1. Enqueue the starting node.
  2. Dequeue a node, process it, and enqueue its unvisited neighbors.
  3. Repeat until the queue is empty.

Breadth-First Search (BFS)

- **Concept:**
  - Start from a node.
  - Visit all its neighbors.
  - Move to the next level neighbors.

- **Algorithm Steps:**
  1. Enqueue the starting node.
  2. Dequeue a node, process it, and enqueue its unvisited neighbors.
  3. Repeat until the queue is empty.
- **Data Structures:**
  - **Queue:** To keep track of nodes to be explored.
  - **Visited Array:** To mark nodes as visited.
  - **Predecessor Array:** To store the path (optional, but useful for shortest path reconstruction).

For more details, visit: GIF

Depth-First Search (DFS)

- **Concept:**
  - Start from a node.
  - Explore as deep as possible before backtracking.

Depth-First Search (DFS)

- **Concept:**
  - Start from a node.
  - Explore as deep as possible before backtracking.
- **Algorithm Steps:**
  1. Push the starting node onto the stack (or call recursively).
  2. Pop a node, process it, and push its unvisited neighbors.
  3. Repeat until the stack is empty.

Depth-First Search (DFS)

- **Concept:**
  - Start from a node.
  - Explore as deep as possible before backtracking.
- **Algorithm Steps:**
  1. Push the starting node onto the stack (or call recursively).
  2. Pop a node, process it, and push its unvisited neighbors.
  3. Repeat until the stack is empty.
- **Data Structures:**
  - **Stack/Recursion:** To keep track of nodes being explored.
  - **Visited Array:** To mark nodes as visited.
  - **Predecessor Array:** To store the path (optional, but useful for path reconstruction).

For more details, visit: GIF

Time Complexities

Time Complexities

- **BFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges

Time Complexities

- **BFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges
  - **Reasoning:**
    - Each vertex is enqueued and dequeued once, resulting in O(V) operations.
    - Each edge is considered once for each of its endpoints, resulting in O(E) operations.

Time Complexities

- **BFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges
  - **Reasoning:**
    - Each vertex is enqueued and dequeued once, resulting in O(V) operations.
    - Each edge is considered once for each of its endpoints, resulting in O(E) operations.
- **DFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges

Time Complexities

- **BFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges
  - **Reasoning:**
    - Each vertex is enqueued and dequeued once, resulting in O(V) operations.
    - Each edge is considered once for each of its endpoints, resulting in O(E) operations.
- **DFS:**
  - Time Complexity: O(V + E)
  - V = Number of vertices
  - E = Number of edges
  - **Reasoning:**
    - Each vertex is pushed and popped from the stack (or recursively called) once, resulting in O(V) operations.
    - Each edge is considered once for each of its endpoints, resulting in O(E) operations.

Strongly Connected Components

- **Problem:** Given a directed graph, determine whether it is strongly connected.
- **Think About It:**
  - How would you approach this problem using BFS or DFS?

Strongly Connected Components

- **Problem:** Given a directed graph, determine whether it is strongly connected.
- **Think About It:**
  - How would you approach this problem using BFS or DFS?
- **Solution:**
  - **Step 1:** Perform a DFS from any node. If any node is not reachable, the graph is not strongly connected.
  - **Step 2:** Reverse the direction of all edges.
  - **Step 3:** Perform a DFS from the same starting node on the reversed graph. If any node is not reachable, the graph is not strongly connected.
  - If all nodes are reachable in both the original and reversed graphs, the graph is strongly connected.

Dijkstra's

1. Dijkstra's algorithm is a SSSP (Single Source Shortest Path) algorithm that finds the shortest path from a source node to **all** other nodes.

Dijkstra's

1. Dijkstra's algorithm is a SSSP (Single Source Shortest Path) algorithm that finds the shortest path from a source node to **all** other nodes.
2. Dijkstra's algorithm can be used in both a directed and undirected graph.

Dijkstra's

1. Dijkstra's algorithm is a SSSP (Single Source Shortest Path) algorithm that finds the shortest path from a source node to **all** other nodes.
2. Dijkstra's algorithm can be used in both a directed and undirected graph.
3. Dijkstra's algorithm will not work if the edge weight is negative.

Dijkstra's

Some information we will need to keep when Dijkstra's algorithm is being ran

1. Distance array - the distance array represents the current best-known path from the source node to the current node. The array is initially all initialized to infinity (since we have no information on the nodes until we explore them) except the source node which has a distance of 0.

Dijkstra's

Some information we will need to keep when Dijkstra's algorithm is being ran

1. Distance array - the distance array represents the current best-known path from the source node to the current node. The array is initially all initialized to infinity (since we have no information on the nodes until we explore them) except the source node which has a distance of 0.
2. Predecessor Array - an array that keeps track of the node where we came from to reach the current node.

Dijkstra's

1. Maintain a set S of vertices for which the shortest path weight has been found, initially empty. S is often represented by a boolean array.
2. For every vertex v, maintain a value d_v which is the weight of the shortest known path from s to v, i.e. the shortest path using only intermediate vertices in S. Initially $d_s = 0$ and $d_v = \infty$ for all other vertices.
3. At each stage, we add to S the vertex $v \in V \setminus S$ which has the smallest d_v value. Record this value as the length of the shortest path from s to v, and update other d_z values as necessary.

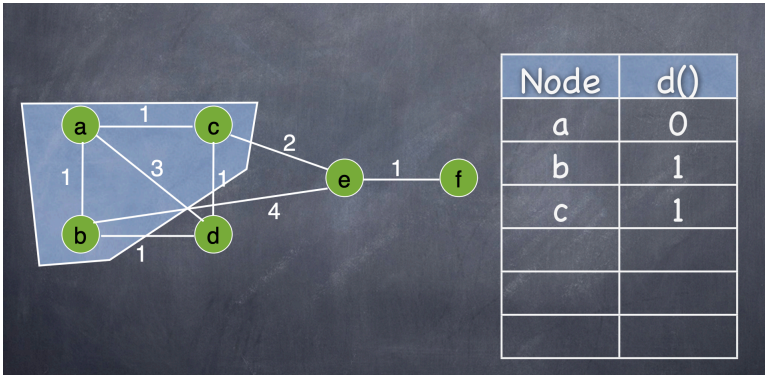
Dijkstra's

```
dijkstraSSSP(G, src):
    Input: graph G, source vertex src

    create dist array, initialised to ∞
    create pred array, initialised to -1
    create vSet containing all vertices of G

    dist[src] = 0
    while vSet is not empty:
        find vertex v in vSet such that dist[v] is minimal
        remove v from vSet
        for each edge (v, w, weight) in G:
            relax along (v, w weight)
```

Dijkstra's



Dijkstra's Complexity - Min Heap

- For Dijkstra's algorithm, in the worst-case scenario, we have to visit every single node and traverse every single edge to find the shortest path.

Dijkstra's Complexity - Min Heap

- For Dijkstra's algorithm, in the worst-case scenario, we have to visit every single node and traverse every single edge to find the shortest path.
- For each edge traversal, we may also have to add the new into the min heap, taking $O(\log V)$ time. Thus, the complexity of traversing all the edges is simply $O(E \log V)$.

Dijkstra's Complexity - Min Heap

- For Dijkstra's algorithm, in the worst-case scenario, we have to visit every single node and traverse every single edge to find the shortest path.
- For each edge traversal, we may also have to add the new into the min heap, taking $O(\log V)$ time. Thus, the complexity of traversing all the edges is simply $O(E \log V)$.
- For each node access from the min heap, this also takes $O(\log V)$ time, so the total time complexity to access all nodes is $O(V \log V)$.

Dijkstra's Complexity - Min Heap

- For Dijkstra's algorithm, in the worst-case scenario, we have to visit every single node and traverse every single edge to find the shortest path.
- For each edge traversal, we may also have to add the new into the min heap, taking $O(\log V)$ time. Thus, the complexity of traversing all the edges is simply $O(E \log V)$.
- For each node access from the min heap, this also takes $O(\log V)$ time, so the total time complexity to access all nodes is $O(V \log V)$.
- So the combined time complexity of Dijkstra's using a min heap $O((V + E) \log V)$.

Dijkstra's Complexity - Min Heap

- For Dijkstra's algorithm, in the worst-case scenario, we have to visit every single node and traverse every single edge to find the shortest path.
- For each edge traversal, we may also have to add the new into the min heap, taking $O(\log V)$ time. Thus, the complexity of traversing all the edges is simply $O(E \log V)$.
- For each node access from the min heap, this also takes $O(\log V)$ time, so the total time complexity to access all nodes is $O(V \log V)$.
- So the combined time complexity of Dijkstra's using a min heap $O((V + E) \log V)$.
- However, the time complexity of Dijkstra's introduced in the course is $O(E + V \log V)$, this is achieved through the use of the Fibonacci-heap.

Kruskal's Algorithm

Definition

- A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it minimises the total length of all edges in T.

Kruskal's Algorithm

Definition

- A minimum spanning tree T of a connected graph G is a subgraph of G (with the same set of vertices) which is a tree, and among all such trees it minimises the total length of all edges in T.

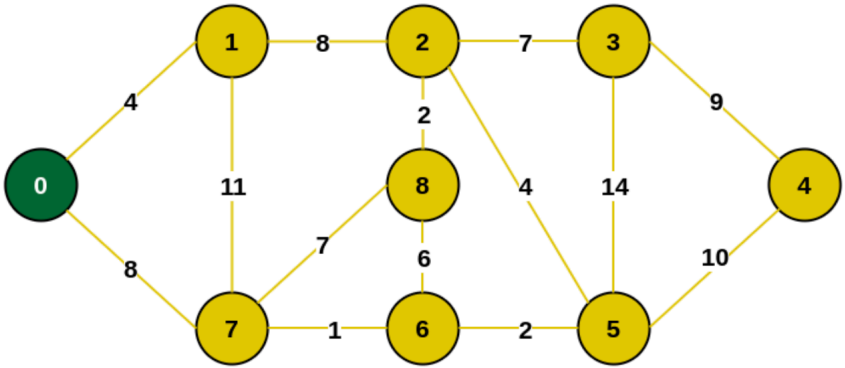
Kruskal's Algorithm

- We sort the edges E in increasing order by weight. An edge e is added if its inclusion does not introduce a cycle in the graph constructed thus far, or discarded otherwise. The process terminates when the forest is connected, i.e. when $n - 1$ edges have been added.

Kruskal's Algorithm

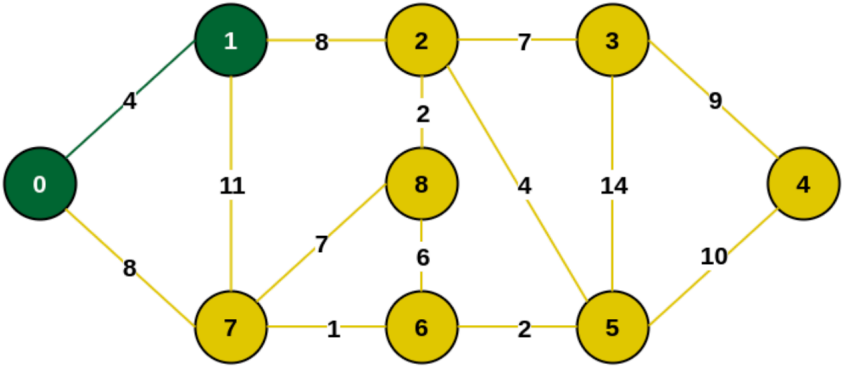
- To check whether an added edge belongs to a cycle, we can either naively check by performing either a DFS traversal throughout the graph, which would yield a time complexity of $O(E \log E + EV)$.
- If we use the union-find data structure, we can detect whether an edge would create a cycle in basically $O(1)$, which means the final time complexity of Kruskal's algorithm is $O(E \log E)$.

Prim's Algorithm



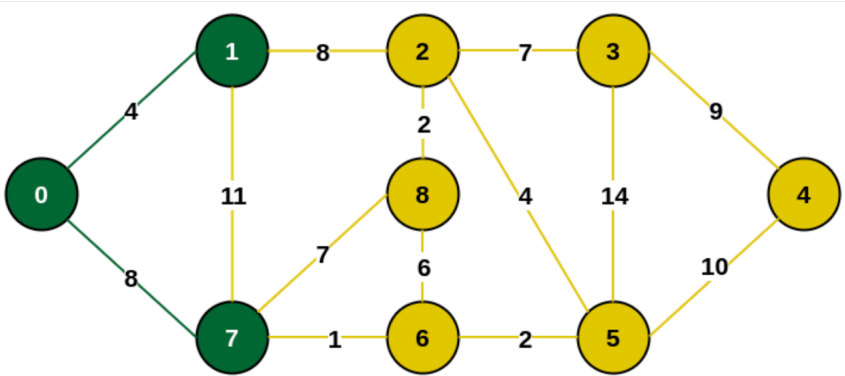
Select an arbitrary starting vertex. Here we have selected 0

Prim’s Algorithm



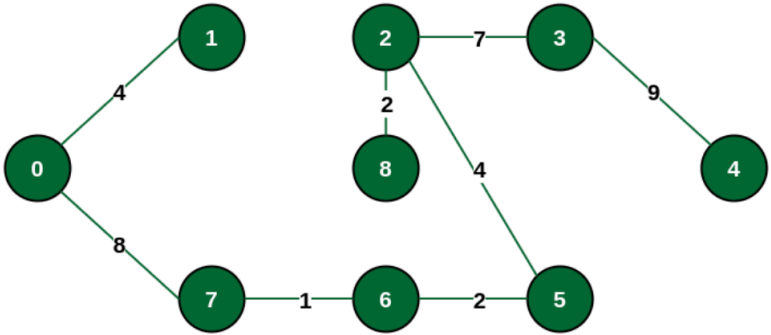
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

Prim's Algorithm



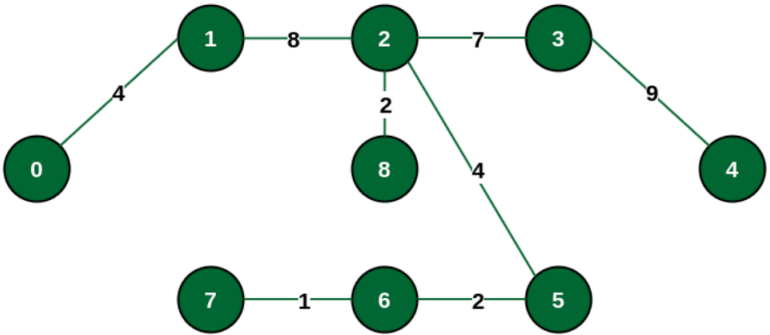
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

Prim’s Algorithm



The final structure of MST

Prim's Algorithm



Alternative MST structure

Question Time!

Suppose you've been tasked with powering the electricity network of Allabam. Given the locations of the towns within Allabam and the sole power plant, you are to construct powerlines to ensure that:

1. All towns are connected to the powerplant.
2. You incur the minimum cost possible, assuming that the cost of constructing a powerline is proportional to its length.

Question Time!

Suppose you've been tasked with powering the electricity network of Allabam. Given the locations of the towns within Allabam and the sole power plant, you are to construct powerlines to ensure that:

1. All towns are connected to the powerplant.
2. You incur the minimum cost possible, assuming that the cost of constructing a powerline is proportional to its length.

Power Planning

Specifically, you are tasking with implementing the following function:

```
int planGrid(struct place cities[], int numCities, struct place powerPlant,
          struct powerLine powerLines[])

struct place {
    char name[MAX_PLACE_NAME + 1];
    int  x;
    int  y;
};

struct powerLine {
    struct place p1;
    struct place p2;
};
```

Sample Solution

```
int planGrid(struct place cities[], int numCities, struct place powerPlant,
struct powerLine powerLines[]) {
// numCities + 1 vertices (to account for power plant)
Graph powerGrid = GraphNew(numCities + 1);
// add all potential powerlines to Pq
Pq powerlines = addPowerlinesToPq(cities, numCities, powerPlant);

bool *visited = calloc(numCities + 1, sizeof(bool));
int totalEdges = 0;
while (totalEdges < numCities) {
    struct edge currEdge = PqExtract(powerlines);
    if (!pathExistsAlready(powerGrid, currEdge.v, currEdge.w, visited)) {
        GraphInsertEdge(powerGrid, currEdge);

        struct place place1 = getPlace(currEdge.v, cities, powerPlant, numCities);
        struct place place2 = getPlace(currEdge.w, cities, powerPlant, numCities);
```

Sample Solution

```
                struct powerLine installedLine = {
                    .p1 = place1,
                    .p2 = place2
                };

                powerLines[totalEdges++] = installedLine;
            }

                // reset visited array for next iteration
            for (int i = 0; i < numCities + 1; i++) visited[i] = false;
        }

        free(visited);
        PqFree(powerlines);
        GraphFree(powerGrid);

        return numCities;
    }
```

Further events



Please join us for:

- Inter-uni programming competition next term!

Competitive
Programming and
Mathematics
Society

Hashmaps

Kyle, Freddie, and Andrew

Why HashMaps, what's the problem

- Consider the average case time complexities of array operations

Why HashMaps, what's the problem

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)

Why HashMaps, what's the problem

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?

Why HashMaps, what's the problem

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?
  - Bob the Builder, Yes We Can!
- Idea: If we give each element a 'fixed' position in an array, then when we insert that element, we don't need to move other elements around $\implies O(1)$ insertion.

Why HashMaps, what's the problem

UNSW CPMSoc

Arc CLUBS SUPPORTED BY Arc INDEPENDENTLY RUN

- Consider the average case time complexities of array operations
  - Search: $O(n)$
  - Insert / Delete: $O(n)$ (shifting other elements to open / close gaps respectively)
- Can we do better - insert and delete in constant time (on average)?
  - Bob the Builder, Yes We Can!
- Idea: If we give each element a 'fixed' position in an array, then when we insert that element, we don't need to move other elements around $\implies O(1)$ insertion.
  - Similarly for deletion.

Enter - The Hash Function

- A hashing function give us this 'fixed' position within an array.

Enter - The Hash Function

- A hashing function give us this 'fixed' position within an array.

```
int hash(int N /* (size of our array we're hashing into) */, int elem) {
    int hash = elem * elem;
    return hash % N;
}
```

- Let's insert into an array of size 8 using this hash function. Now we can call this 'array' a hash table. Consider 10, 12, 13:

Enter - The Hash Function

- A hashing function give us this 'fixed' position within an array.

```c
int hash(int N /* (size of our array we're hashing into) */, int elem) {
    int hash = elem * elem;
    return hash % N;
}
```

- Let's insert into an array of size 8 using this hash function. Now we can call this 'array' a hash table. Consider 10, 12, 13:

12	13			10			
0	1	2	3	4	5	6	7

- Now it's your turn - give us some integers to insert!

Hash Collisions

- As we saw from your examples, we run into issues when distinct elements have the same hash! This occurrence is called a 'hash collision'. If we do not do anything to deal with this we will overwrite previous data.
- So What can we do?

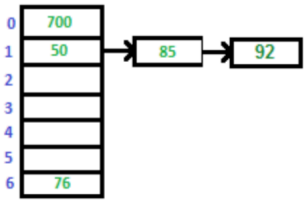
Hash Collisions

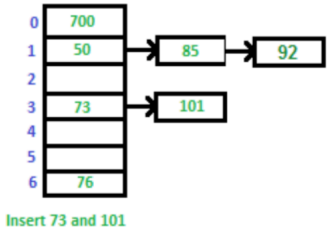
- As we saw from your examples, we run into issues when distinct elements have the same hash! This occurrence is called a 'hash collision'. If we do not do anything to deal with this we will overwrite previous data.
- So What can we do?
- Separate Chaining
- Linear Probing
- Double Hashing

Separate Chaining

- A natural idea is to chain these elements with identical hashes in a linked list:

We can chain these together very much like a simple linked list!

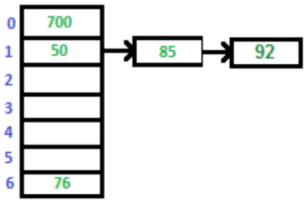


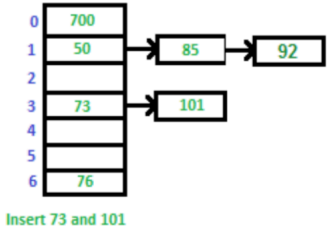


Separate Chaining

- A natural idea is to chain these elements with identical hashes in a linked list:

We can chain these together very much like a simple linked list!



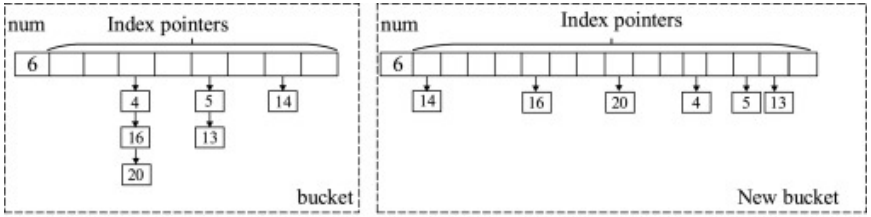


Can we do **better**?

- If all elements hash to a single slot (in the worst case), we'll have a $O(n)$ search, insert *and* delete ... so what was the point?!

Separate Chaining

- We can dynamically resize the underlying array when the total number of elements inserted is greater than the size of the array.
- This yields an amortized (averaged-out) complexity of $O(1)$ for all operations on a HashMap.
- Note: This assumes that we have a hashing function that distributes evenly.



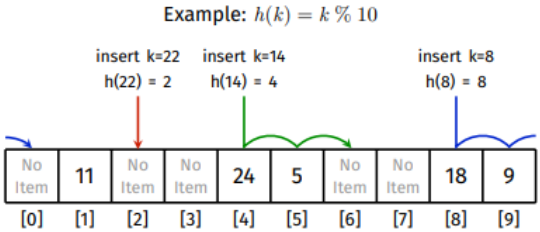
(a) original state

(b) state after rehash operation

Linear Probing

Insertion:

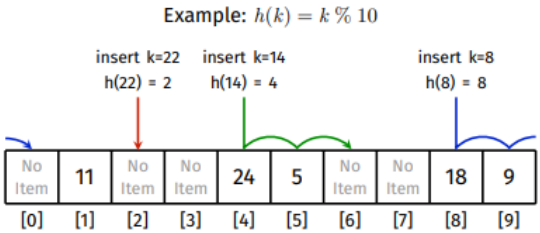
- If load factor exceeds threshold, dynamically resize
- Hash key and if a collision occurs find the next empty slot



Linear Probing

Lookup:

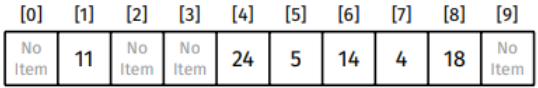
- Hash key and find the first slot that either contains the key or is empty
- If the key is found we return the value, otherwise the key does not exist in the hashmap



Linear Probing

Deletion:

- The tombstone Method
- The Backshift Method

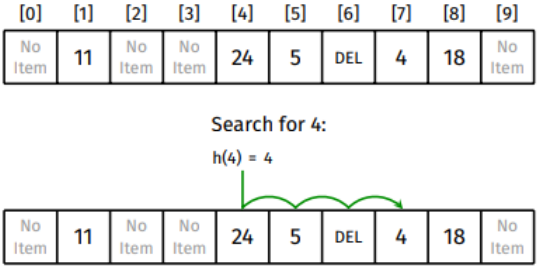


[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	24	5	14	4	18	No Item

Linear Probing

Deletion:

- The tombstone Method



Linear Probing

Deletion:

- The Backshift Method

Step 1: Remove 24

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	No Item	5	14	4	18	No Item

Step 2: Re-insert 5

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	No Item	5	14	4	18	No Item

Step 3: Re-insert 14

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	No Item	4	18	No Item

Linear Probing

Deletion:

- The Backshift Method

Step 4: Re-insert 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

Step 5: Re-insert 18

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

Linear Probing

Issues:

- Clustering
- Long filled sections which slow down insertion and lookup

Step 4: Re-insert 4

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

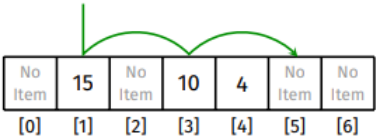
Step 5: Re-insert 18

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
No Item	11	No Item	No Item	14	5	4	No Item	18	No Item

Double Hashing

Insertion:

- If we run into a collision with our initial hash, we apply a second hash function to determine an increment
- Suppose $h_1(k) = k\%7$ and $h_2(k) = k\%3 + 1$
- Let us try to insert 22



Double Hashing

Lookup:

- Resize if necessary
- Apply initial hash
- whilst key/empty slot is not found keep jumping by second hash increment

Double Hashing

Deletion:

- Backshift method is more difficult due to the varying increments
- Tombstone method is still the same

Summary

Deletion:

- All method have $O(1)$ amortised and $O(n)$ worst assuming a good hash function and appropriate resizing
- Usually we can resize by doubling the number of slots

SORTING ALGORITHMS

CONTENTS

- Big O Time Complexities
- Practical understanding of how the sort is performed
- Pros / cons
- NO IMPLEMENTATION

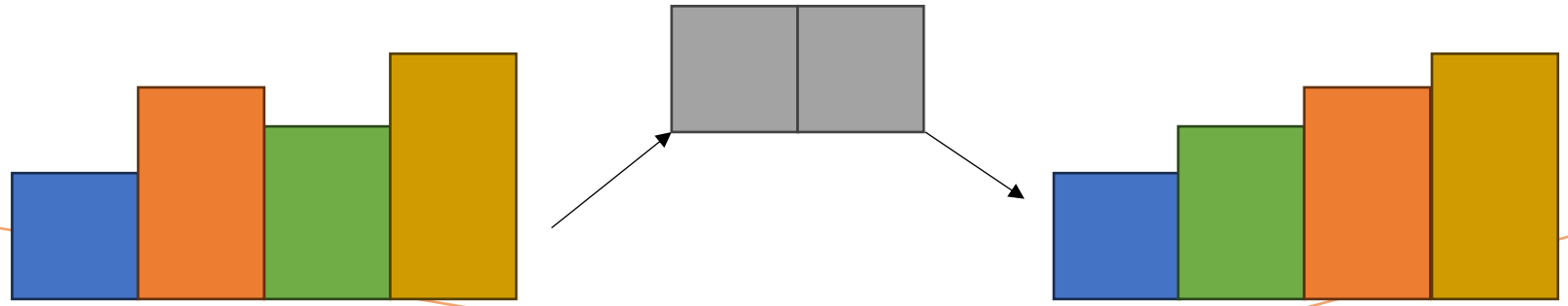
PROPERTIES OF SORTS

- Stability

- Adaptability

¯_(ツ)_/¯

- In-Place

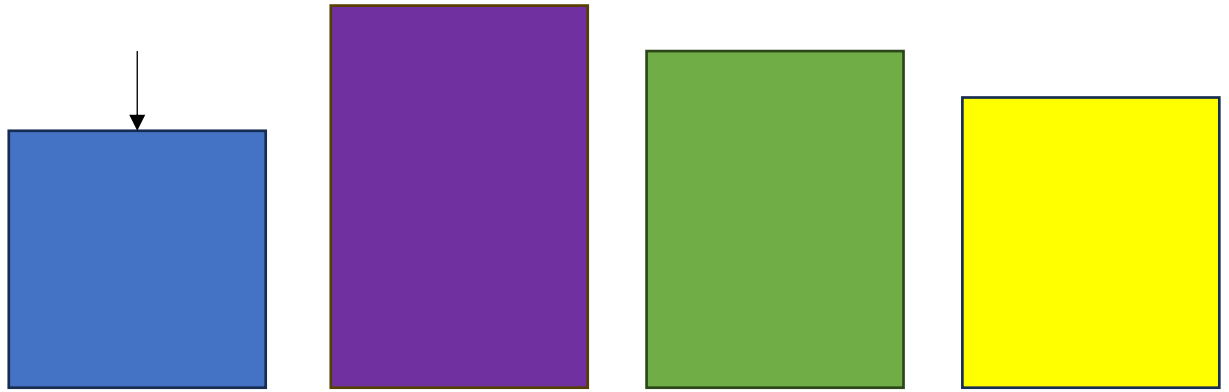


SELECTION SORT

- Properties:
  - Unstable
  - Non-Adaptive
  - In-Place
- Time Complexities
  - Worst Case: O(n^2)
  - Average Case: O(n^2)
  - Best Case: O(n^2)

CLICK ME TO SEE ANIMATIONS

SELECTION SORT

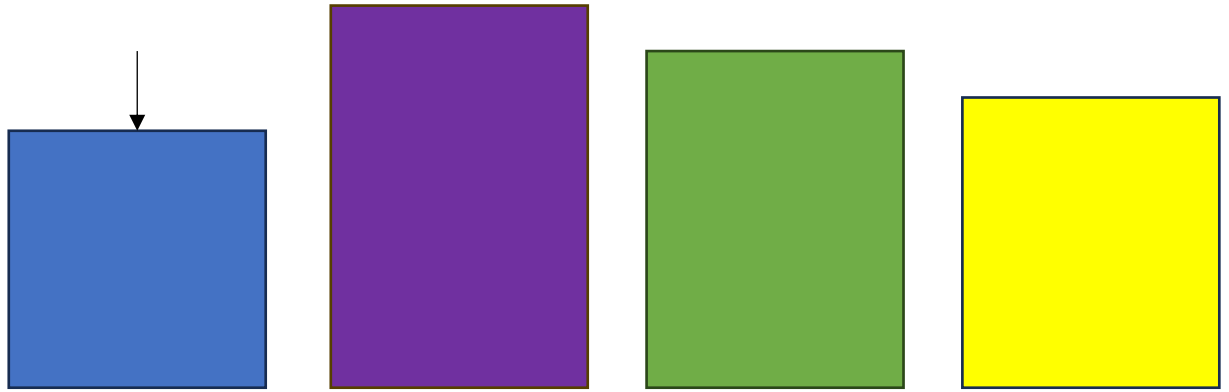


BUBBLE SORT

- Properties
  - Stable
  - Adaptive
  - In-Place
- Time Complexities
  - Worst Case: O(n^2)
  - Average Case: O(n^2)
  - Best Case: O(n) – An already sorted array

CLICK ME TO SEE ANIMATIONS

BUBBLE SORT

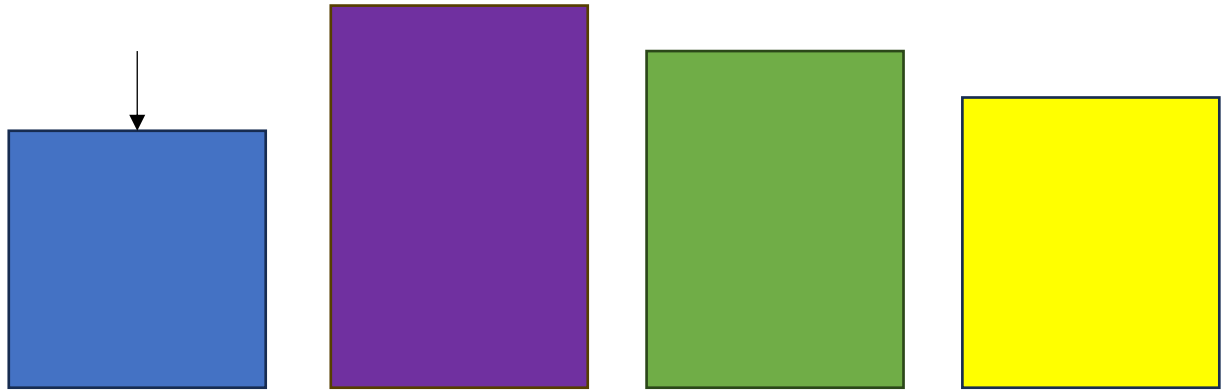


INSERTION SORT

- Properties
  - Stable
  - Adaptive
  - In-Place
- Time Complexities
  - Worst Case: O(n^2)
  - Average Case: O(n^2)
  - Best Case: O(n) – An already sorted array

CLICK ME TO SEE ANIMATIONS

INSERTION SORT

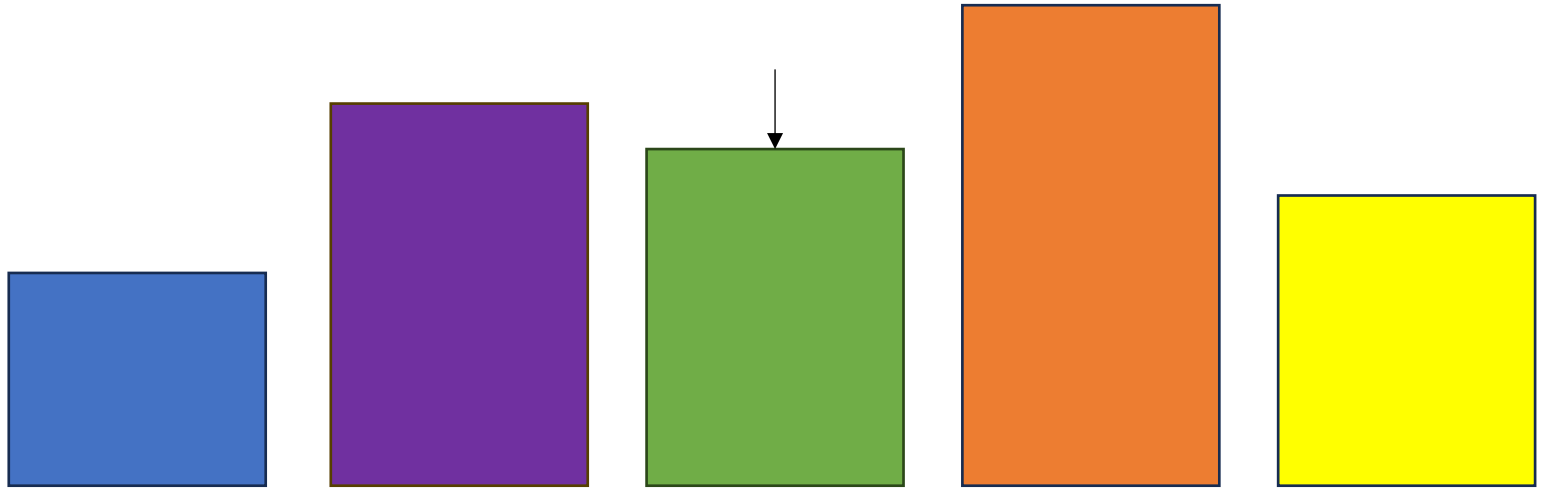


SHELL SORT

- Properties
  - Non-Stable
  - Adaptive
  - In-Place
- Time Complexities
  - Worst Case: Depends | O(n^2)
  - Average Case: Depends
  - Best Case: Depends | O(nlogn)

CLICK ME TO SEE ANIMATIONS

SHELL SORT

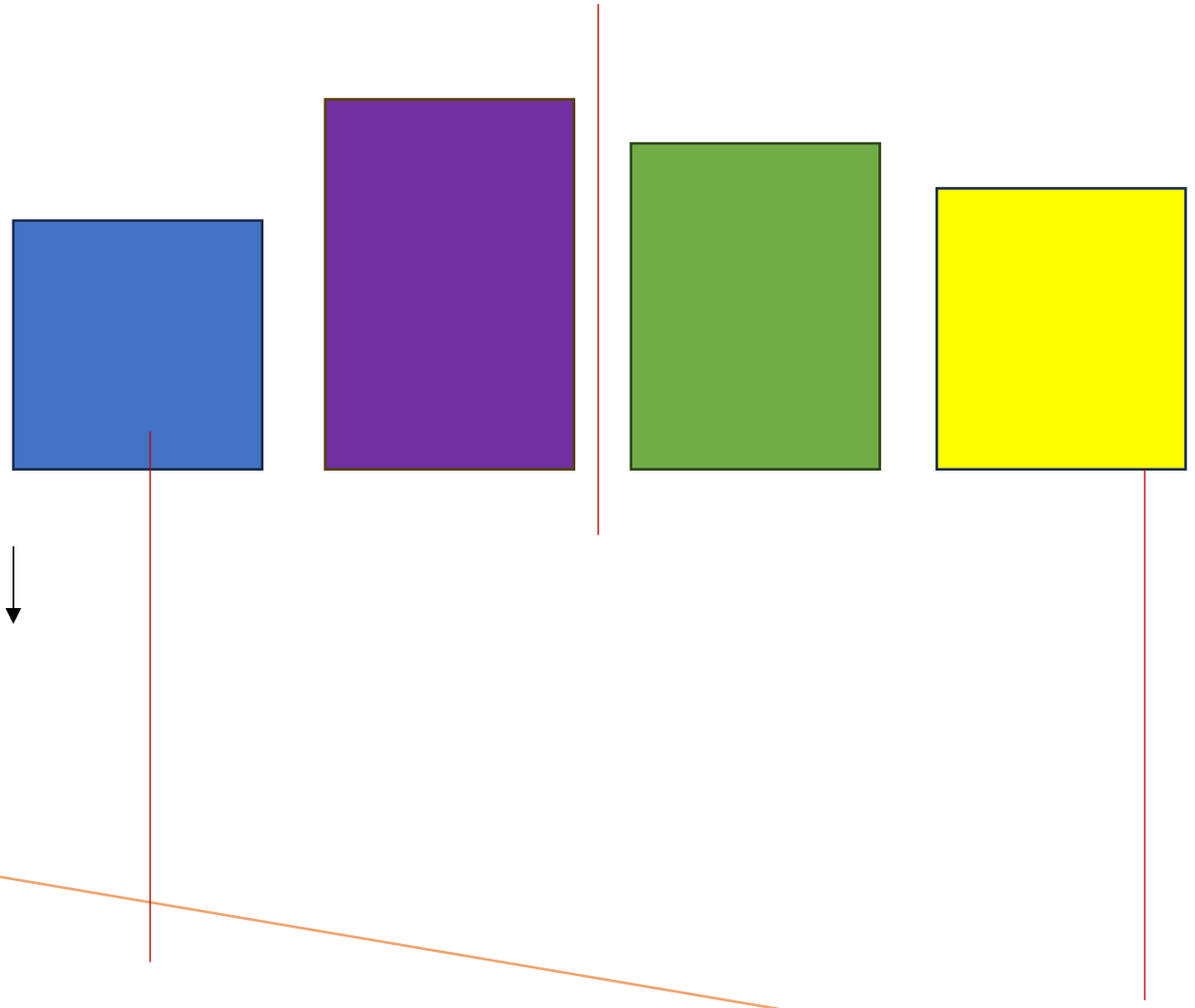


MERGE SORT

- Properties
  - Stable
  - Non-Adaptive
  - Not In-Place
- Time Complexities
  - Worst Case: O(nlogn)
  - Average Case: O(nlogn)
  - Best Case: O(nlogn)

CLICK ME TO SEE ANIMATIONS

MERGE SORT

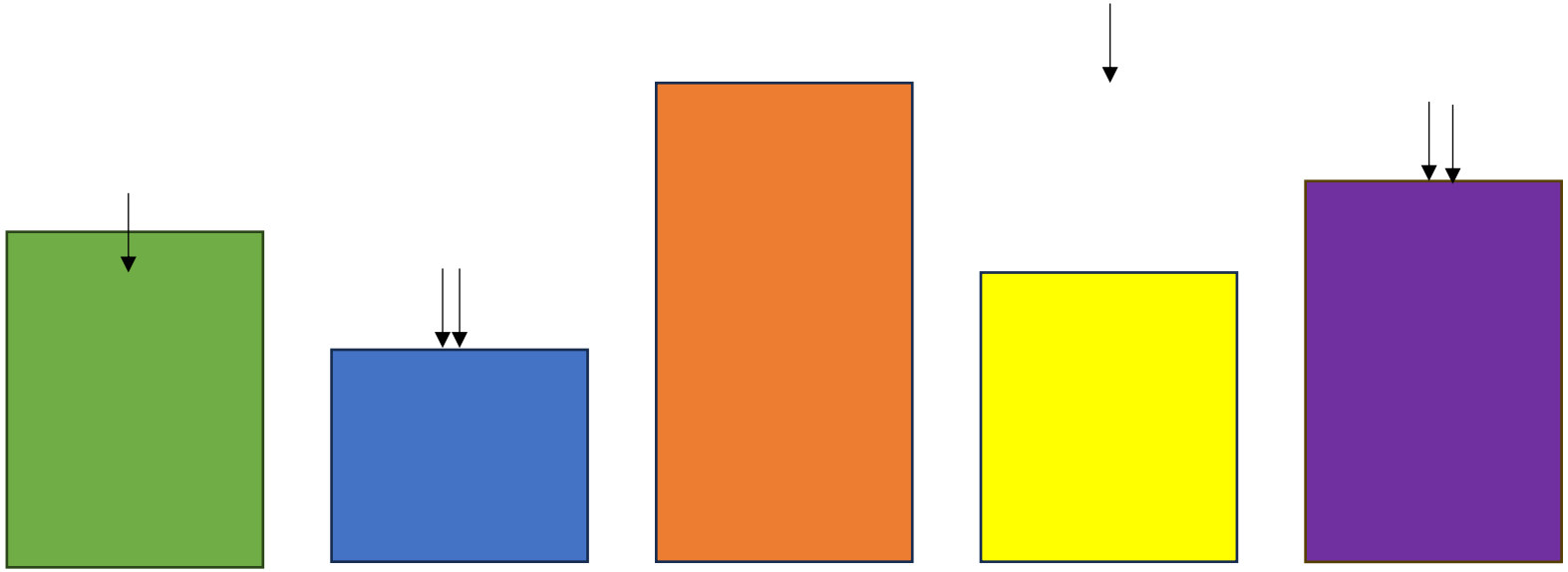


QUICK SORT

- Properties
  - Non-Stable
  - ??? | Non-Adaptive
  - In-Place
- Time Complexities
  - Worst Case: O(n^2)
  - Average Case: O(nlogn)
  - Best Case: O(nlogn)

CLICK ME TO SEE ANIMATIONS

QUICK SORT



QUICK SORT (MEDIAN OF 3)