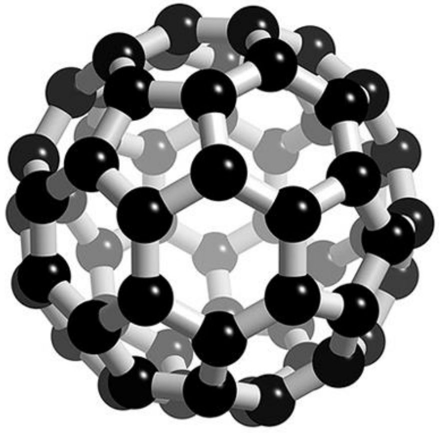
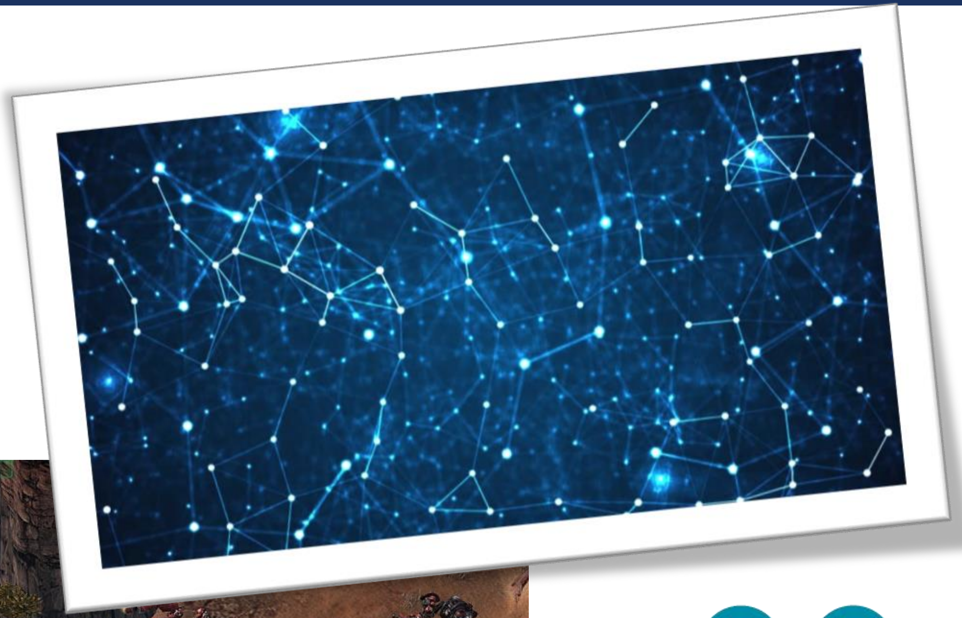
DISJOIN SETS

DATA STRUCTURES AND ALGORITHMS

DISJOIN SETS

Content





- Disjoin Sets (a.k.a. Union Find)
  - Work Principles
  - Implementation
- Disjoin Set based Tasks and algorithms
  - Minimum Spanning tree
    - Kruskal's algorithm
  - Cycle finding (in undirected graph)

DISJOIN SET

ALGORITHM, IMPLEMENTATIONS, USAGE

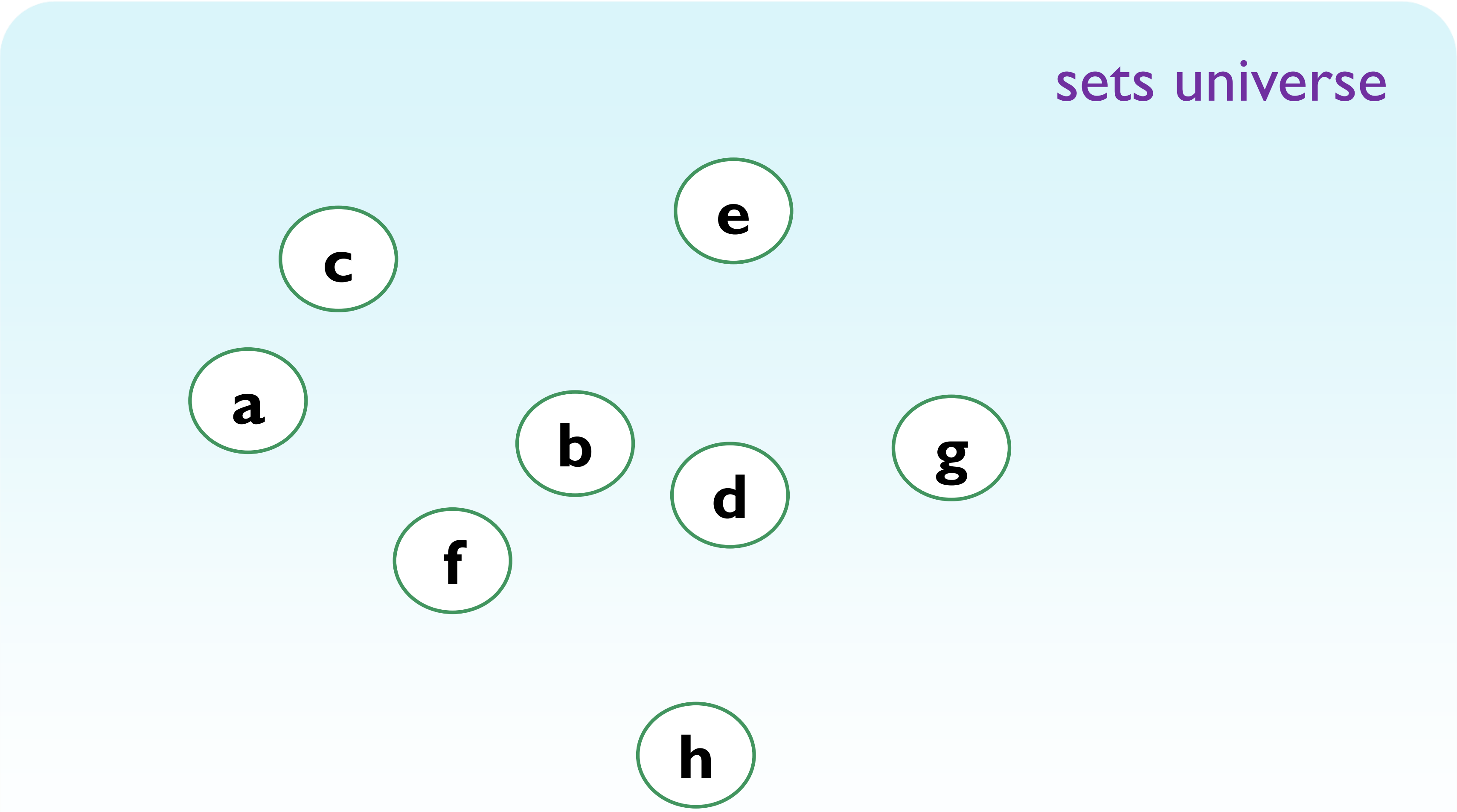
DISJOIN SETS

Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
init_set(a);
init_set(b);
init_set(c);
init_set(d);
init_set(e);
init_set(f);
init_set(g);
init_set(h);
```



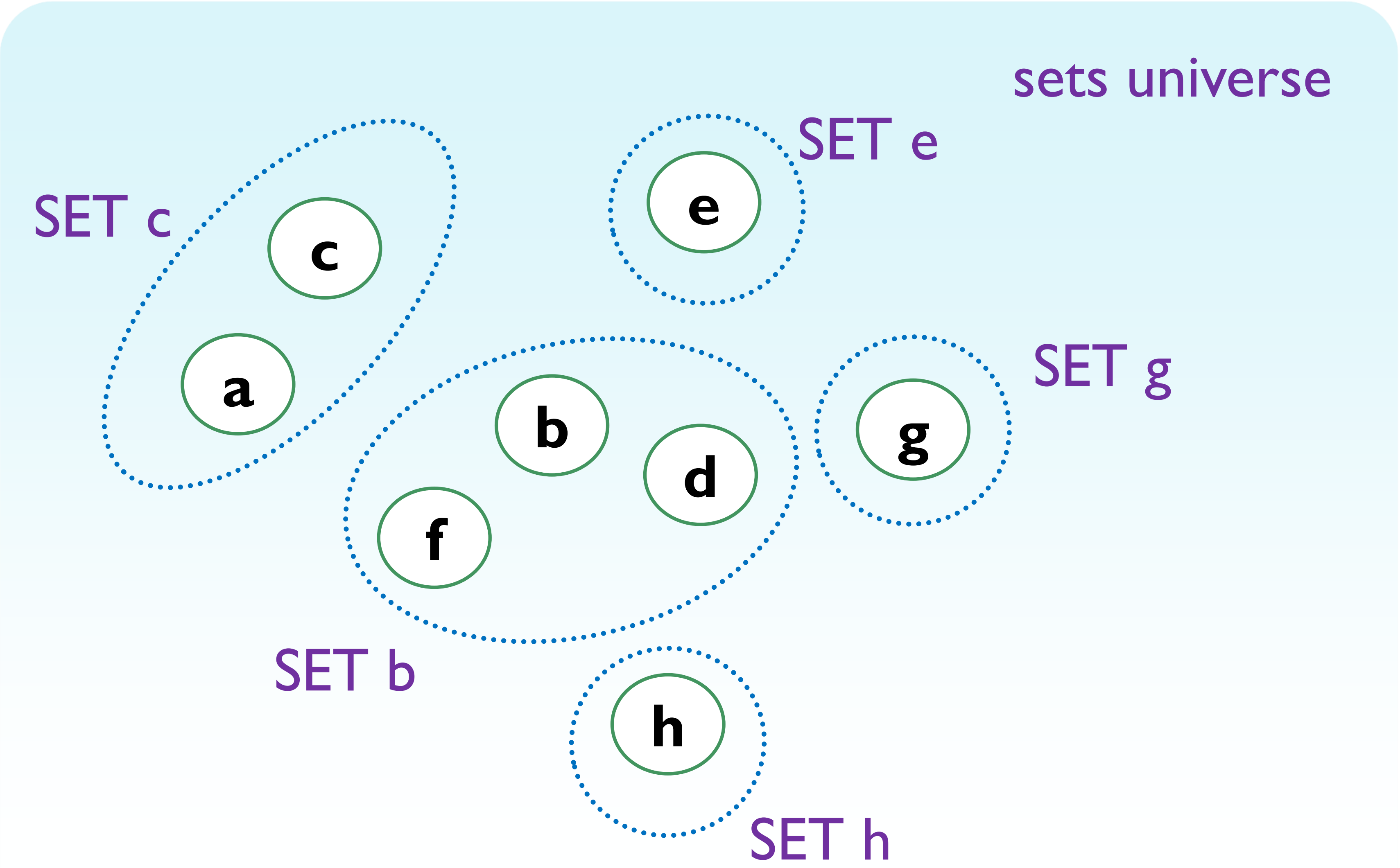
DISJOIN SETS

Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
find_set(b) // Set b
find_set(a) // Set c
find_set(g) // Set g
```



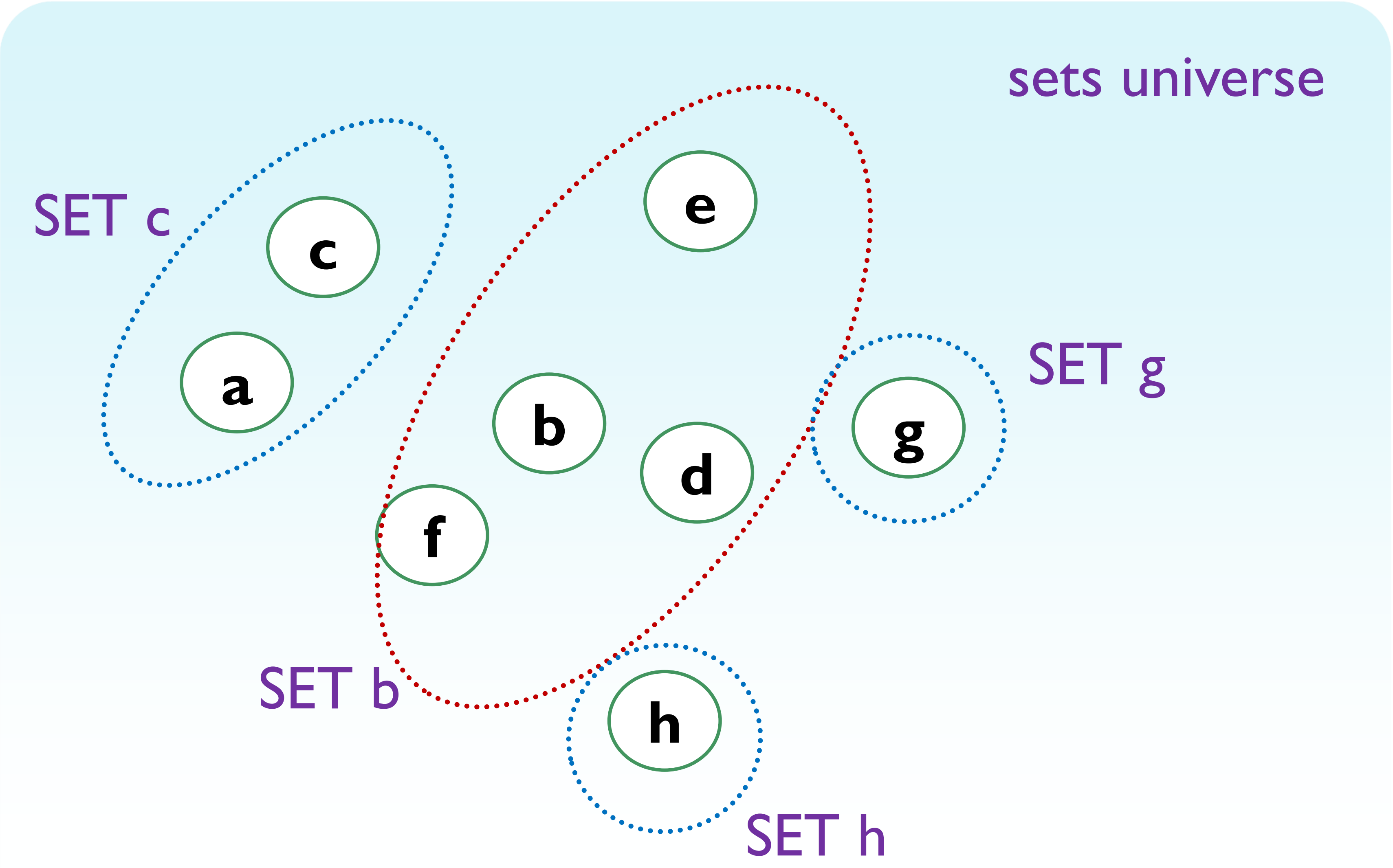
DISJOIN SETS

Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
find_set(a) // Set c
union_set(e, b)
find_set(a) // Set b
```



DISJOIN SETS

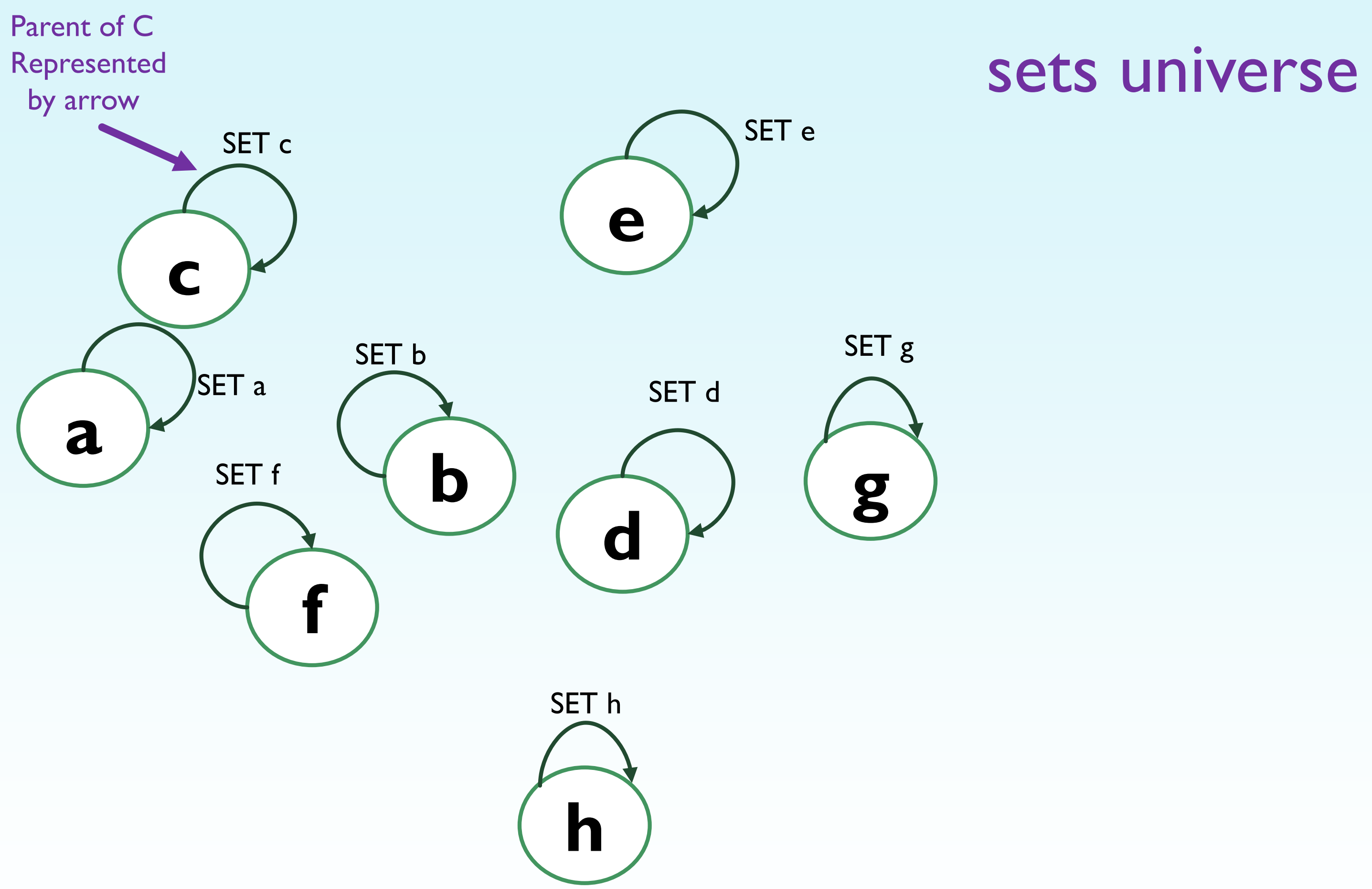
Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
init_set(a-h)
```

- Initialization creates the set universe where each element represents separated set.
- Representation of set done stores in parent vector (arrow in picture)



DISJOIN SETS

Work Principles

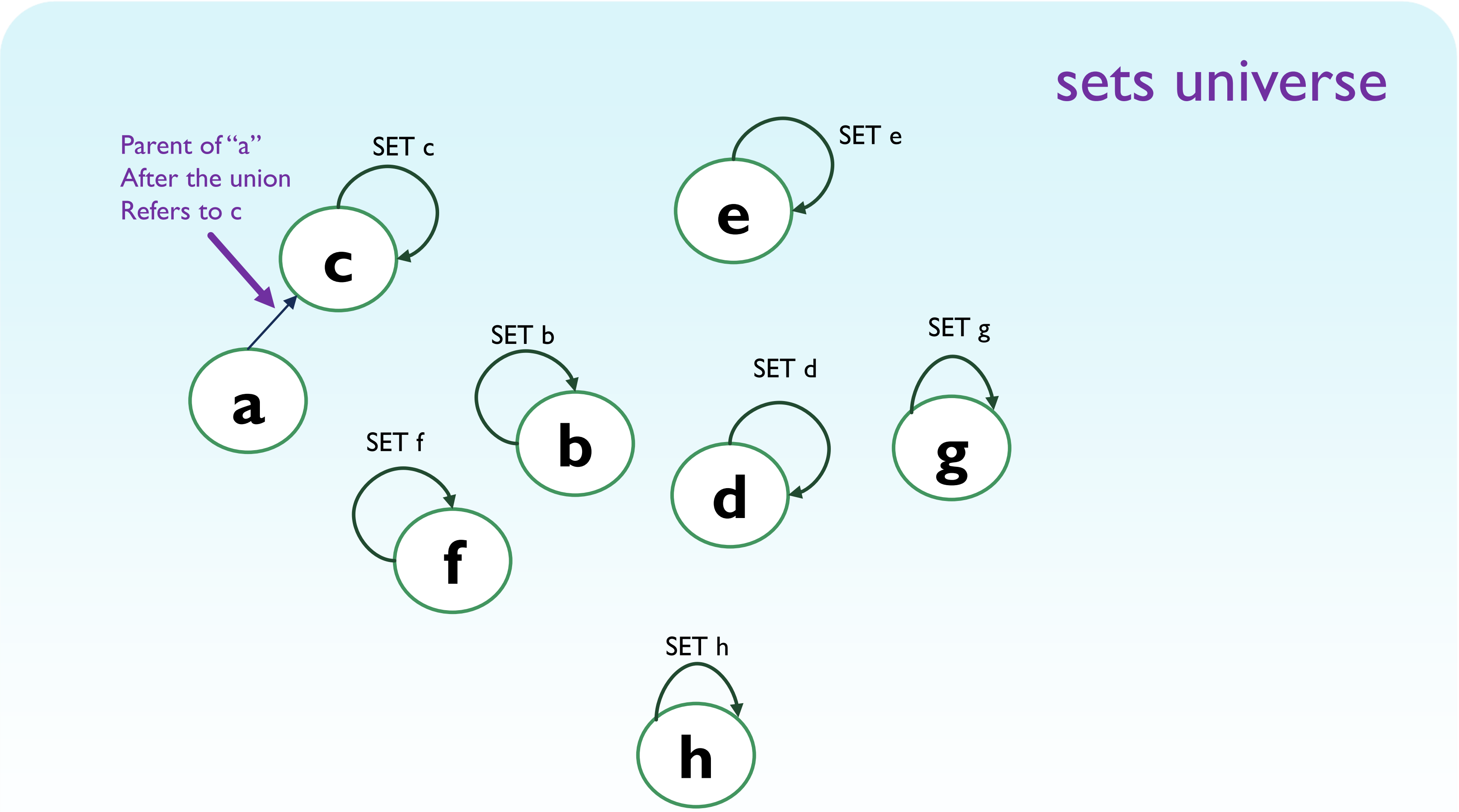
Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
union_set(a, c)
```

- Union is achieved by changing the parent of first element to the second parent

```
union_set(h, b)
union_set(f, b)
union_set(e, b);
```



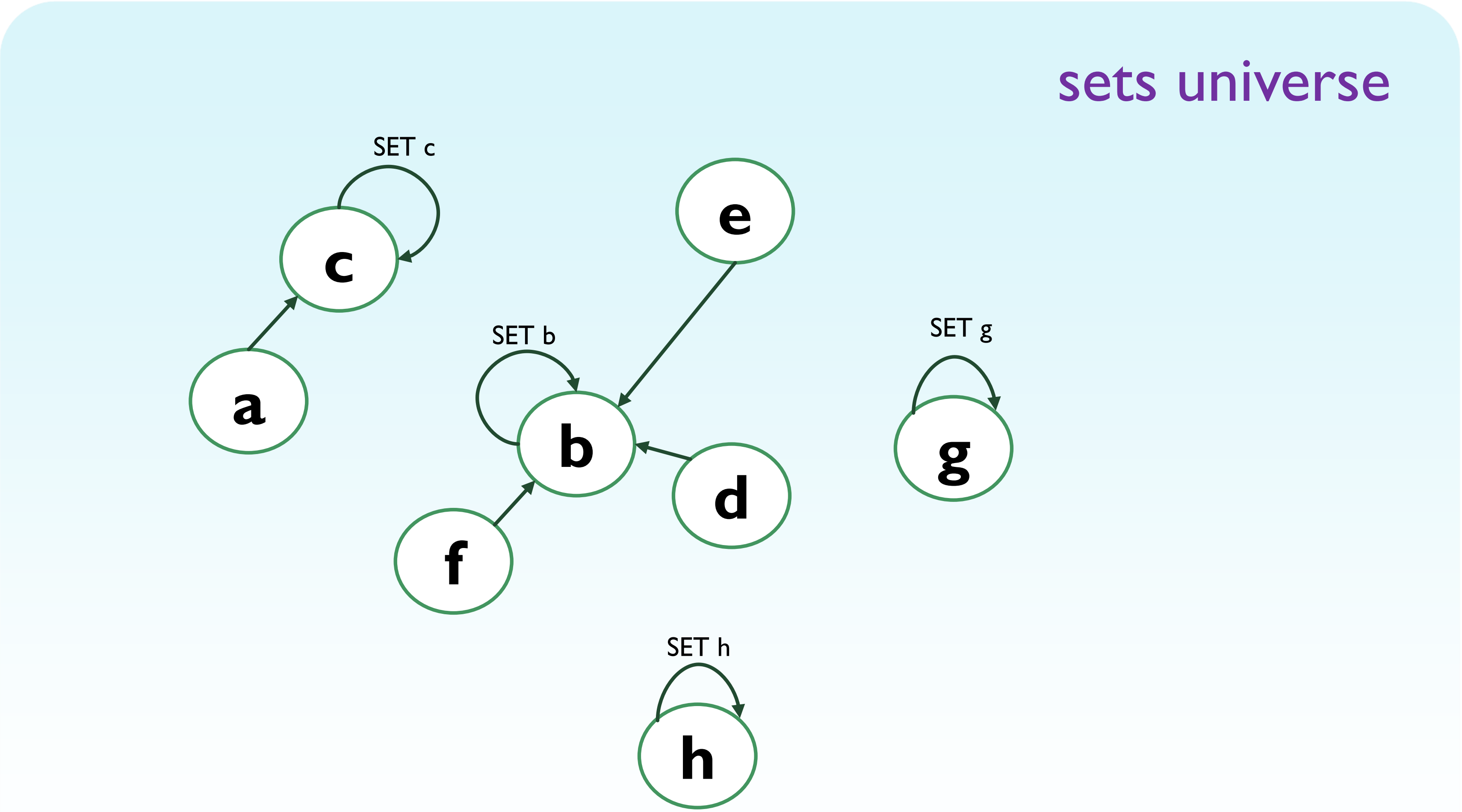
DISJOIN SETS

Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
find_set(a) // Set c
```



DISJOIN SETS

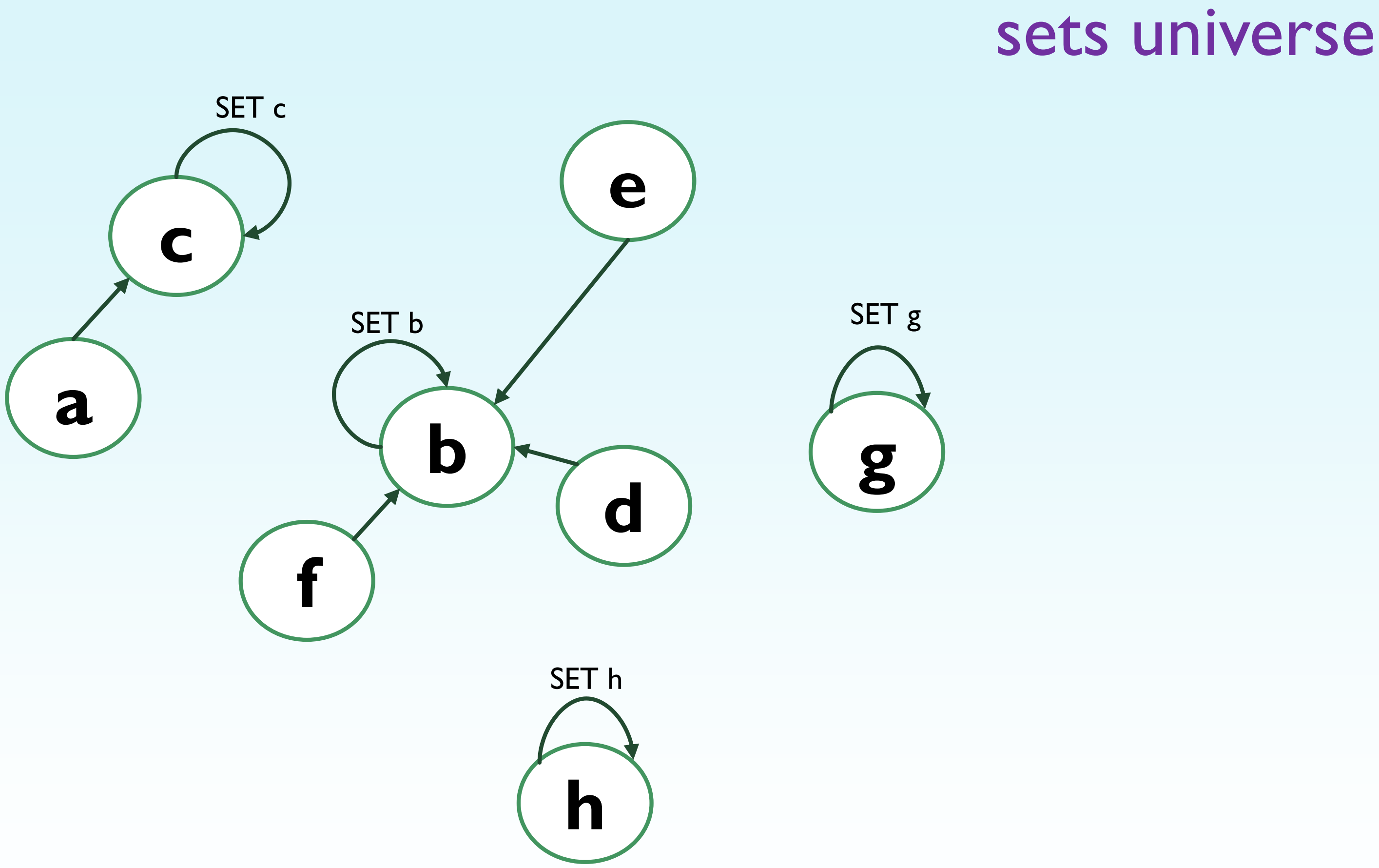
Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

```
find_set(a) // Set c
```

- find_set searches for parent
- When parent vector of an element returns an element itself, the parent is founded.



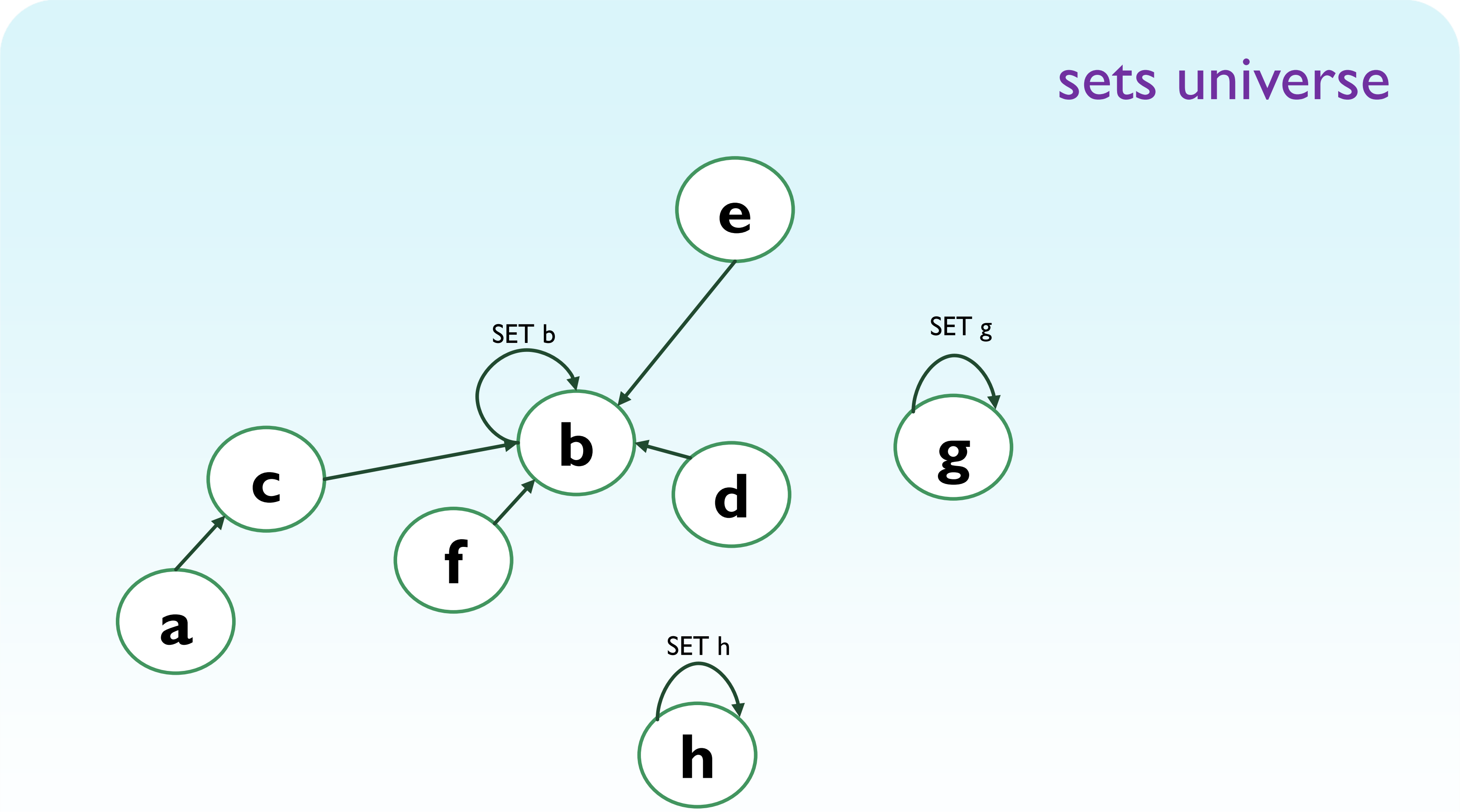
DISJOIN SETS

Work Principles

Functions:

```
init_set() // O(1)
find_set() //O(depth)
union_set()
```

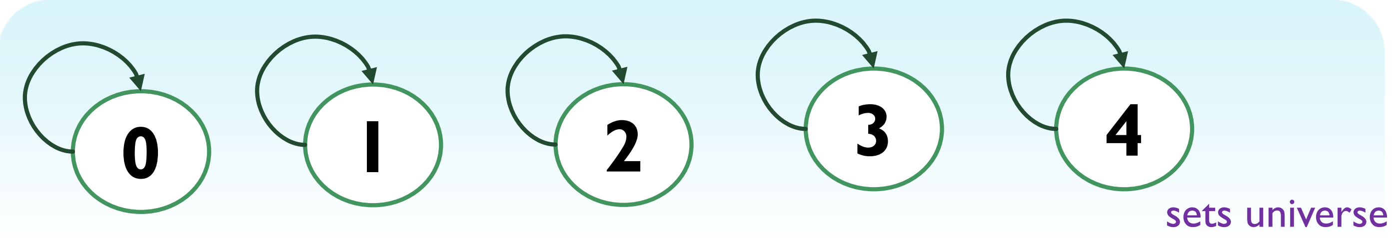
```
find_set(a) // Set c
union_set(setc, setf)

find_set(a) // set b
find_set(c) // set b
```



IMPLEMENTATION (INITIALIZATION)

```
class DisjoinSet{
private:
    vector<int> parent, rank;
public:
    DisjoinSet(int n);
    int find_set(int i);
    bool is_same_set(int i, int j);
    void union_set(int i, int j);
};

DisjoinSet::DisjoinSet(int n){
rank.assign(n, 0);
parent.assign(n, 0);
    for (int i = 0; i < n; i++){
        parent[i] = i;
    }
}
```

```
int main(){
    DisjoinSet *ds = new DisjoinSet(5);
    return 0;
}
```



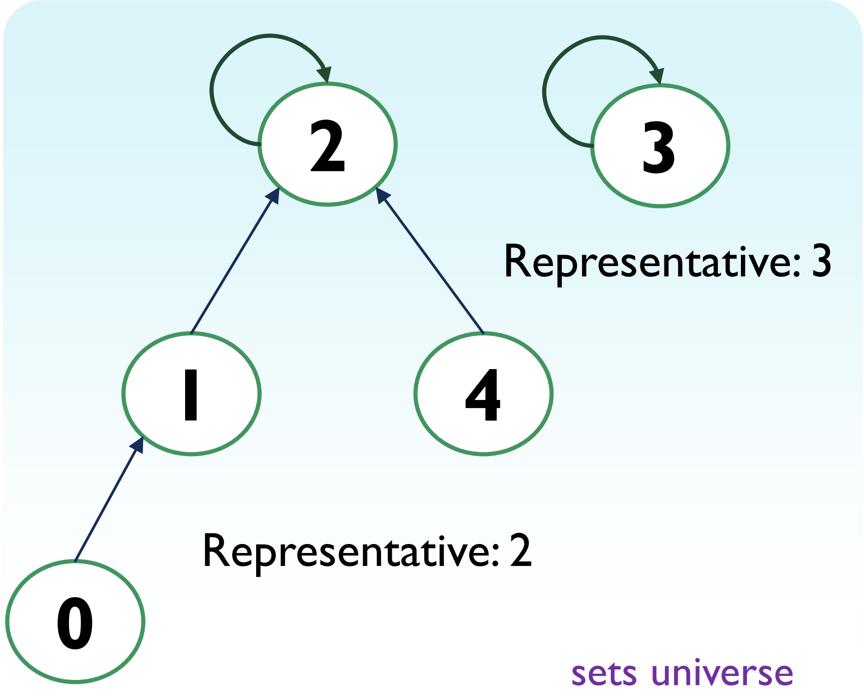
- Initially Disjoin set creates the set of an element where each element points to itself
- The parent vector points to the set of representatives

IMPLEMENTATION (FIND SET)

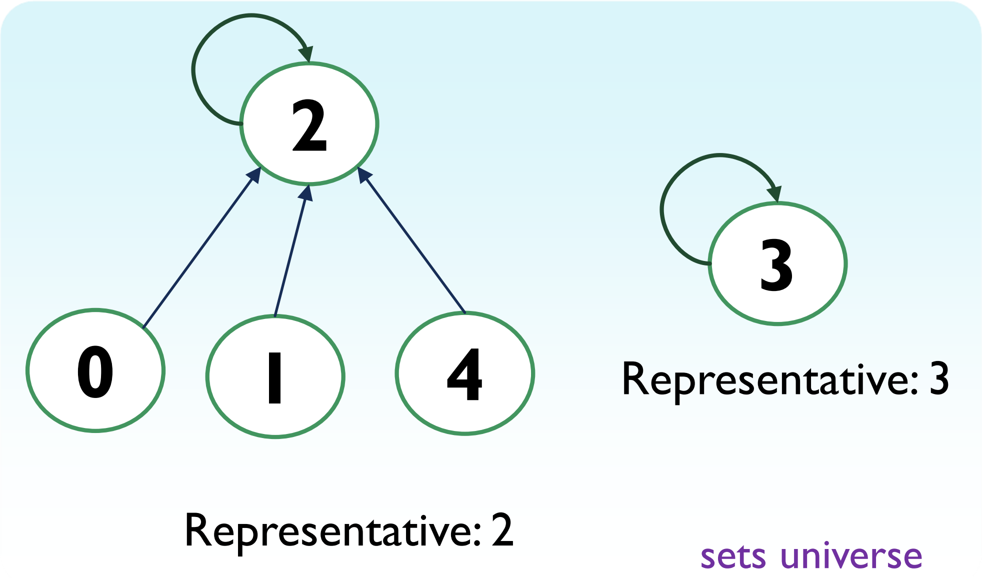
```
int DisjoinSet::find_set(int i){
 if(parent[i] == i)
    return i;
 else
   return parent[i] = find_set(parent[i]);
}
```

explanation

- Find set uses recursive approach to find the representative of the set
- Recursive approach not only finds, but also updates the chain of parents that minimizes parent paths.
- After calling the find_set(0) the set also updates it's parent directly to 2




```
ds->find_set(3); // 3
ds->find_set(2); // 2
ds->find_set(4); // 2
ds->find_set(0); // 2
```

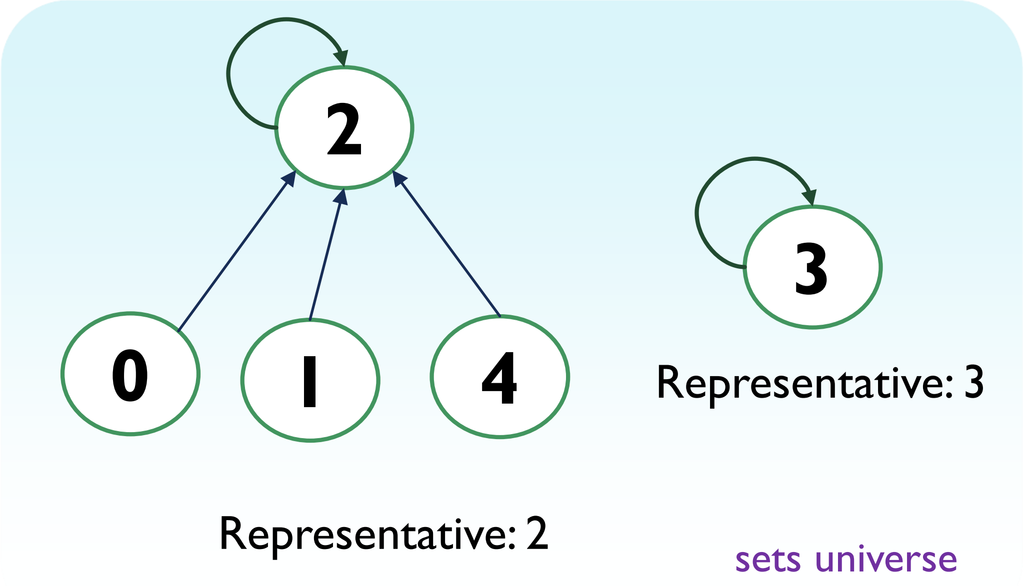


IMPLEMENTATION (UNION FIND)

```
bool DisjoinSet::is_same_set(int i, int j){

   return find_set(i) == find_set(j);

}
```

explanation

- Checking the parents of both items recursively.




```
ds->is_same_set(0, 1); // true
ds->is_same_set(1, 3); // false
ds->is_same_set(4, 2) // true;
```

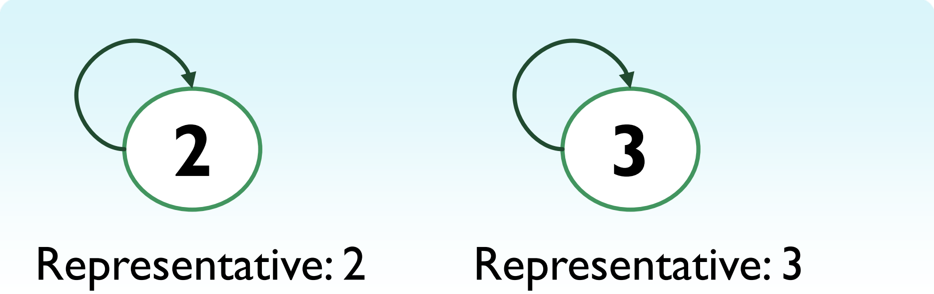
DISJOIN SET: UNION SET

```
void DisjoinSet::union_set(int i, int j){
   if(!is_same_set(i, j)){
      int x = find_set(i);
      int y = find_set(j);
      if(rank[x] > rank[y])
        parent[y] = x;
      else{
        parent[x] = y;
        if(rank[x] == rank[y])
           rank[y]++;
      }
   }
}
```

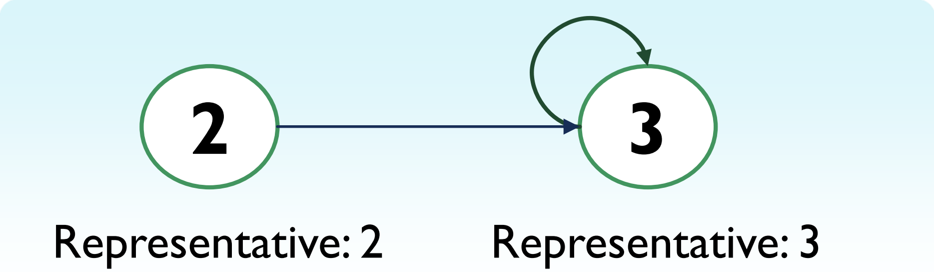
Heuristic approach

```
void DisjoinSet::union_set(int i, int j){
       find_set(parent[i]) = find_Set(j);
}
```

Non heuristic approach




```
ds->union_set(2,3);
```

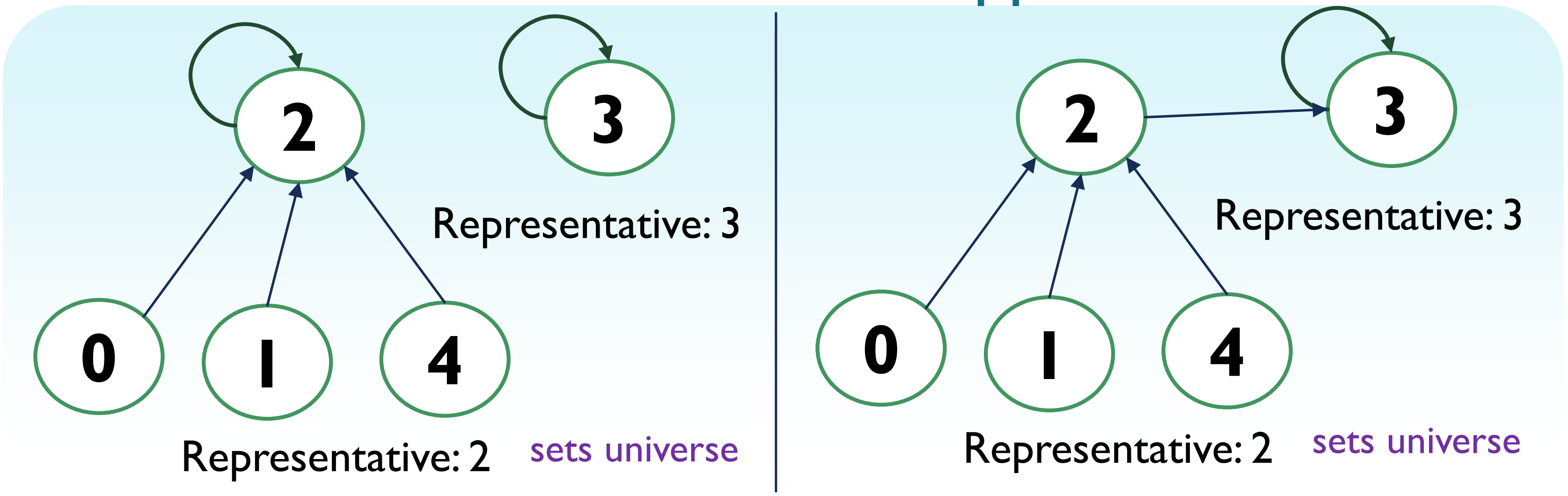


HEURISTIC VS NON HEURISTIC APPROACH

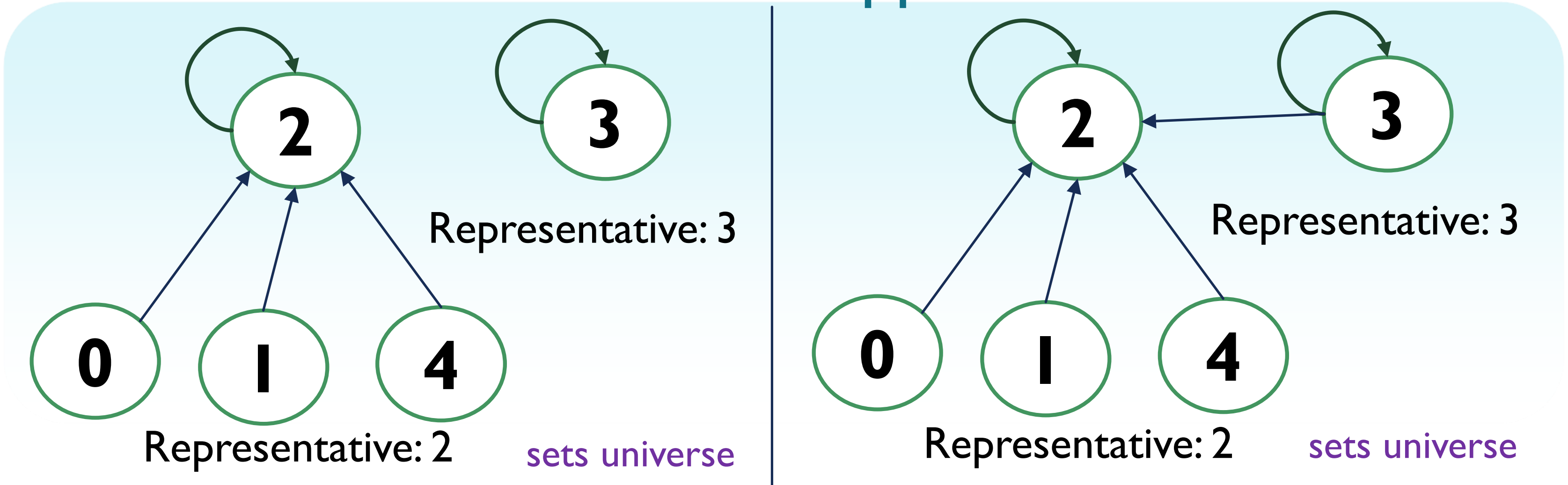
Difference

- In the non heuristic approach the union_set function causes to bigger chain of parents.
- Heuristic approach checks the rank of two merged elements then less parent rank element refers to more rank element
- On picture with non heuristic approach the chain consist of tree levels while heuristic approach created two leveled chain.

Non heuristic approach



Heuristic approach



DISJOIN SET DATA STRUCTURE

```
class DisjoinSet{

private:
  vector<int> p, rank;

public:
  DisjoinSet(int n){
    rank.assign(n, 0);
    p.assign(n, 0);
    for (int i = 0; i < n; i++){
      p[i] = i;
    }
  }

  int find_set(int i){
       return (p[i]==i) ? i : (p[i] = find_set(p[i]));
  }
  bool is_same_set(int i, int j){
         return find_set(i) == find_set(j);
  }
```

```
void union_set(int i, int j){
    if(!is_same_set(i, j)){
      int x = find_set(i), y = find_set(j);
      if(rank[x] > rank[y])
          p[y] = x;
      else{
         p[x] = y;
         if(rank[x] == rank[y])
               rank[y]++;
      }
    }
  }
};
```

MINIMUM SPANNING TREE. KRUSKAL'S ALGORITHM

ALGORITHM, IMPLEMENTATIONS, USAGE

MINIMUM SPANNING TREE

Definition

A minimum spanning tree (MST) or minimum weight spanning tree is a subset of the edges of a connected, edge-weighted (un)directed graph that connects all the vertices together, without any cycles and with the minimum possible total edge weigh

Usage

- Networking
- Telecommunication

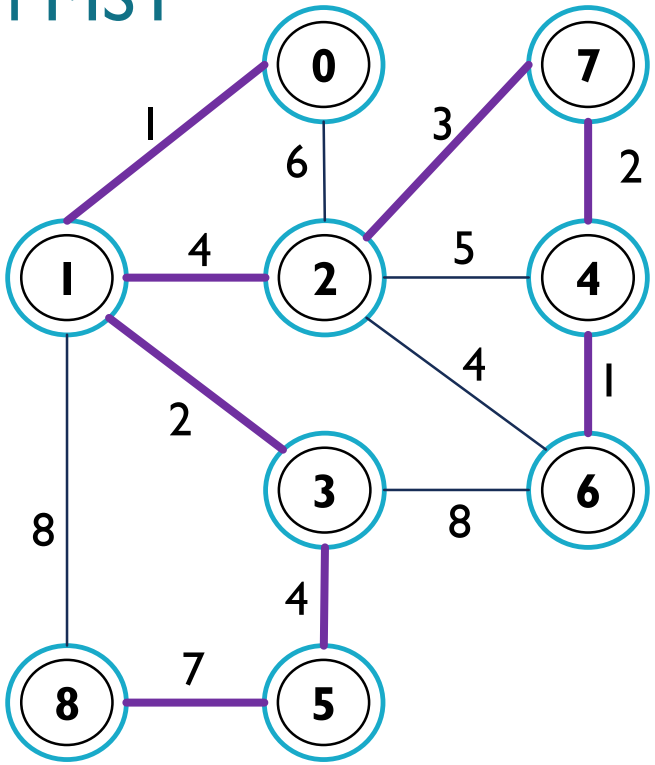
Algorithms

- Prim's algorithm
  - $O(n^2)$, $O(m \log n)$ – priority queue solution
- Kruskal's algorithm $O(m \log n)$ $- with\ Disjoin\ Sets$

$$G = (V, E)$$
$$E_G = \{\{0,1\}, \{1,2\}\{1,3\}, \{1,8\}, \{8,5\}, \{5,3\}, \{3,6\}, \{6,4\}, \{4,2\}, \{2,6\}, \{2,0\}. \{2,7\}, \{7,4\}\}$$

$$MST \in G$$
$$MST = (V, E)$$
$$E_{MST} = \{\{0,1\}, \{1,2\}, \{1,3\}, \{3,5\}, \{5,8\}, \{2,7\}, \{7,4\}, \{4,6\}\}$$

Graph with MST example



Purple edges: the MST of the Graph

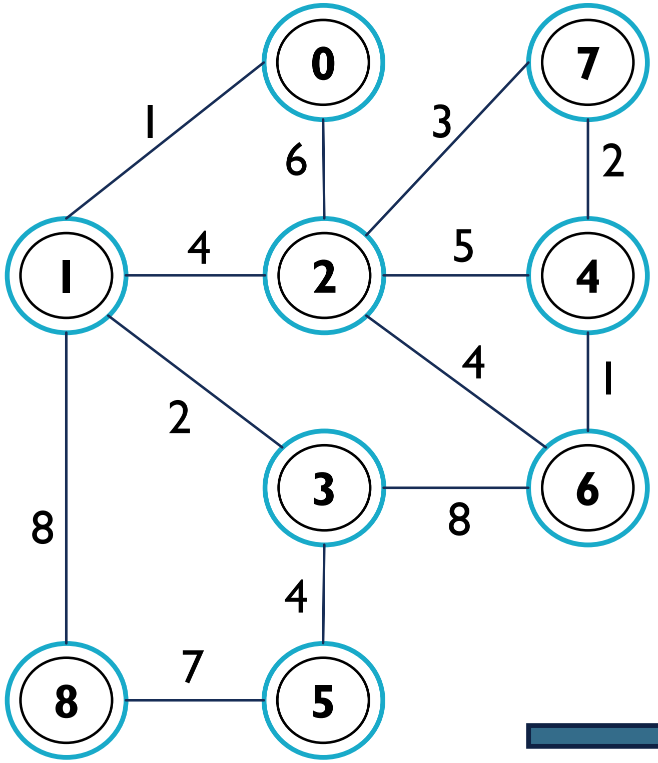
KRUSKAL'S ALGORITHM

Algorithm

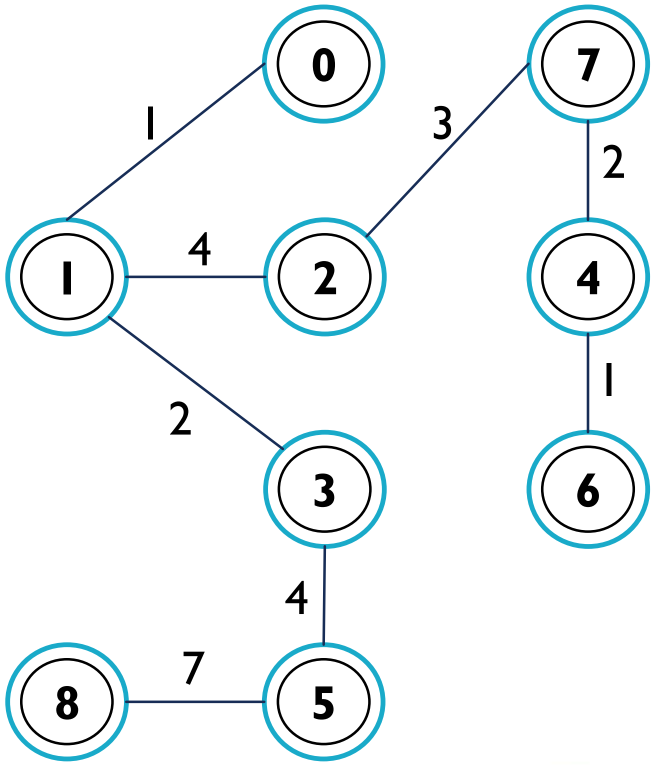
Just as in the simple version of the Kruskal algorithm, we sort the all the edges of the graph in non-decreasing order of weights. Then put each vertex in its own tree (i.e. its set) via DSU make_set() function call - it will take a total of $O(N)$. Iterate through all the edges (in sorted order) and for each edge determine whether the ends belong to different trees (with two find_set() calls in $O(1)$ each). Finally, we need to perform the union of the two trees(sets), for which the DSU union_sets() function will be called - also in $O(1)$. So we get the total asymptotic complexity $O(\mathrm{M}\log N + N + M) = O(MlogN)$.

```
MST-KRUSKAL(G, w)
A = ∅
for each vertex v ∈ G.V
    MAKE-SET(v)
sort the edges of G.E into nondecreasing order by weight w
for each edge (u, v) ∈ G.E, taken in nondecreasing order by weight
    if FIND-SET(u) ≠ FIND-SET(v)
        A = A ∪ {(u, v)}
        UNION(u, v)
return A
```

KRUSKAL'S ALGORITHM



Adj:		Adj:		Adj:		MST:
{1,{1,0}}		{1,{1,0}}		**{1,{1,0}}**		{1,{1,0}}
{8,{1,8}}		{1,{4,6}}		**{1,{4,6}}**		{1,{4,6}}
{2,{1,3}}		{2,{1,3}}		**{2,{1,3}}**		{2,{1,3}}
{7,{8,5}}		{2,{7,4}}		**{2,{7,4}}**		{2,{7,4}}
{4,{5,3}}	Sort by the weights	{3,{2,7}}	union(a,b) If a & b not in the same set	**{3,{2,7}}**		{3,{2,7}}
{4,{1,2}}		{4,{5,3}}		**{4,{5,3}}**		{4,{5,3}}
{6,{0,2}}		{4,{1,2}}		**{4,{1,2}}**		{4,{1,2}}
{5,{2,4}}		{4,{2,6}}		{4,{2,6}}		{7,{8,5}}
{4,{2,6}}		{5,{2,4}}		{5,{2,4}}		
{3,{2,7}}		{6,{0,2}}		{6,{0,2}}	add to MST selected	
{2,{7,4}}		{7,{8,5}}		**{7,{8,5}}**		
{1,{4,6}}		{8,{1,8}}		{8,{1,8}}		
{8,{3,6}}		{8,{3,6}}		{8,{3,6}}		



Kruskal's Algorithm

```
struct edge {
    int from, to, weight;
};
vector<edge> adj, MST;
```

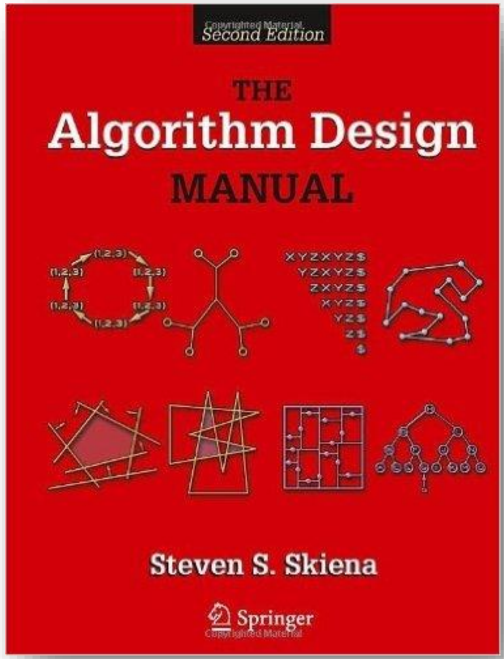
weight from to

```
vector<pair<int, pair<int,int>>> adj;
vector<pair<int, pair<int,int>>> MST;
```

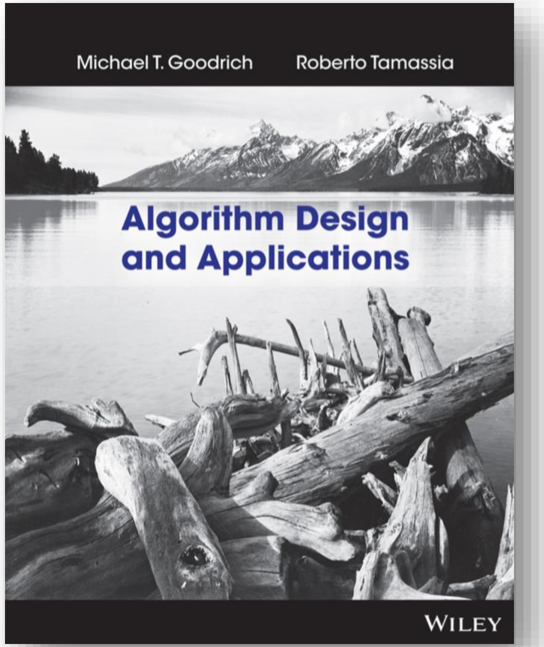
KRUSKAL ALGORITHM IMPLEMENTATION

```
vector<pair<int, pair<int,int>>> adj;

int kruskal(int n){

    int cost = 0;
    DisjoinSet* DS = new DisjoinSet(n);

    sort(adj.begin(), adj.end());

    for (int i = 0; i < adj.size(); i++){
        auto edge = adj[i];
        if(! DS->is_same_set( edge.second.first, edge.second.second)){

            cost += edge.first;
            DS->union_set(edge.second.first, edge.second.second);
        }
    }
    return cost;
}
```

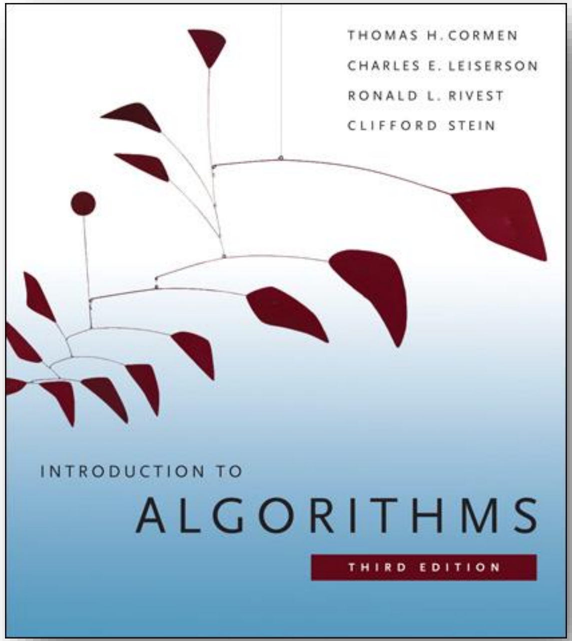
LITERATURE



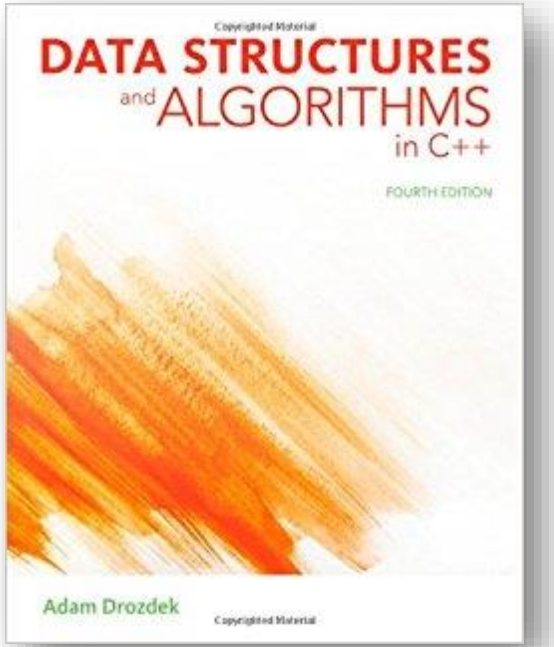
Stieven Skienna
Algorithms design manual
Chapter 15.3
Minimum Spanning Tree
Page 484



Michael T Goodrich
Roberto Tamassia
Algorithms design and Applications
Chapter 7
Union Find Structure
Page 219



Thomas H. Cormen
Introduction to Algorithms
Chapter V, 21 Data Structures for Disjoin Sets
Page 561.



Adam Drozdek
Data structures and Algorithms in C++
Spanning trees 411
Union Find Problem 409