Assignment-5

Name – Ujjwal Lehri
Roll No. – 60216403224
Course – B.Tech CSE 4th sem

1. **WAP to show the implementation of Kruskal's algorithm for computing MST, calculate its time complexity, and show the working of the algorithm.**
2. **WAP for disjoint sets and show their working.**

Sol.

1) Kruskal's algorithm:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX 100

typedef struct{
   int src, dest, weight;
} Edge;

typedef struct{
   int V, E;
   Edge* edges;
}Graph;

typedef struct Node{
   int parent;
   int rank;
} Node;

void printMST(Edge *result, int e);
Graph* createGraph(int V, int E);
int find(Node *set, int i);
void Union(Node *set, int x, int y);
int compareEdges(const void* a, const void* b);
void KruskalMST(Graph* graph);
```

```
int main() {
    int V = 6;
    int E = 8;
    Graph* graph = createGraph(V, E);

    graph->edges[0].src = 0;
    graph->edges[0].dest = 1;
    graph->edges[0].weight = 5;

    graph->edges[1].src = 0;
    graph->edges[1].dest = 3;
    graph->edges[1].weight = 11;

    graph->edges[2].src = 1;
    graph->edges[2].dest = 4;
    graph->edges[2].weight = 3;

    graph->edges[3].src = 1;
    graph->edges[3].dest = 2;
    graph->edges[3].weight = 7;

    graph->edges[4].src = 2;
    graph->edges[4].dest = 4;
    graph->edges[4].weight = 1;

    graph->edges[5].src = 2;
    graph->edges[5].dest = 5;
    graph->edges[5].weight =-3;

    graph->edges[6].src = 3;
    graph->edges[6].dest = 4;
    graph->edges[6].weight = 0;

    graph->edges[7].src = 5;
    graph->edges[7].dest = 4;
    graph->edges[7].weight = 2;

    KruskalMST(graph);

    free(graph->edges);
    free(graph);

    return 0;
}
```

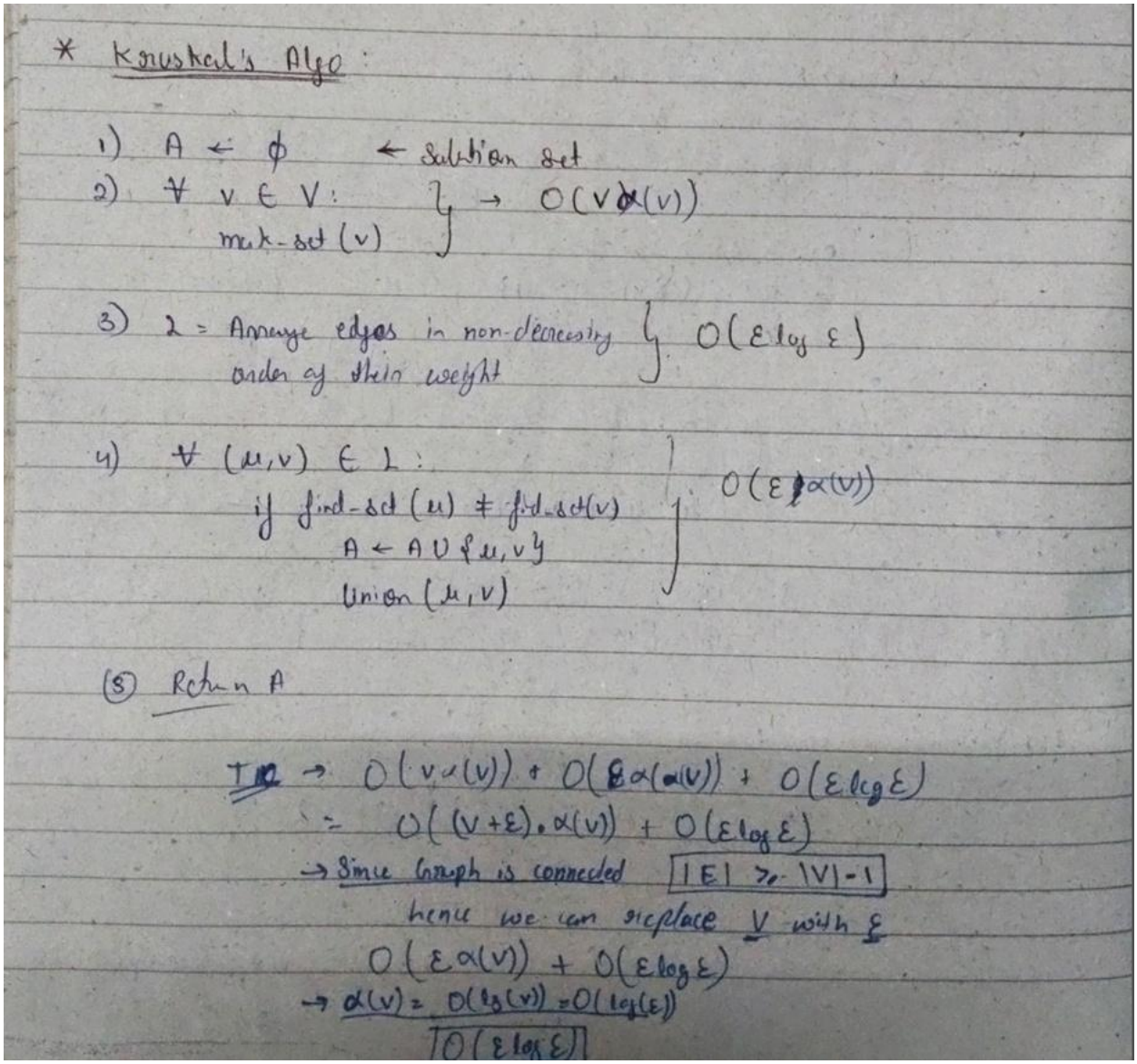
```
//-----------------------------------------------------------------------------

void printMST(Edge *result, int e){
   printf("Constructed MST:\n");
   int totalWeight = 0;
   for (int i = 0; i < e; ++i) {
      printf("%d--> %d, weight = %d\n", result[i].src, result[i].dest, result[i].weight);
      totalWeight += result[i].weight;
   }
   printf("MST weight: %d\n", totalWeight);
}

//-----------------------------------------------------------------------------

Graph* createGraph(int V, int E) {
   Graph* graph = (Graph*)malloc(sizeof(Graph));
   graph->V = V;
   graph->E = E;
   graph->edges = (Edge*)malloc(graph->E * sizeof(Edge));
   return graph;
}

//-----------------------------------------------------------------------------

Node *InitSet(int V){
   Node *set = (Node*)calloc(V, sizeof(Node));
   for(int i = 0; i < V; i++){
      set[i].parent = i;
      set[i].rank = 0;
   }
   return set;
}

//-----------------------------------------------------------------------------

int find(Node *set, int v){
   if(set[v].parent != v) set[v].parent = find(set, set[v].parent);

   return set[v].parent;
}

//-----------------------------------------------------------------------------
```

```
void Union(Node*set, int x, int y){
   int rootX = find(set, x);
   int rootY = find(set, y);

   if (rootX == rootY) return;

   if(set[rootX].rank > set[rootY].rank){
      set[rootY].parent = rootX;
   }else if(set[rootX].rank < set[rootY].rank){
      set[rootX].parent = rootY;
   }else{
      set[rootY].parent = rootX;
      set[rootX].rank++;
   }
}

//-------------------------------------------------------------------------------------

int compareEdges(const void* a, const void* b) {
   Edge* edgeA = (Edge*)a;
   Edge* edgeB = (Edge*)b;
   return edgeA->weight > edgeB->weight;
}

//-------------------------------------------------------------------------------------

void KruskalMST(Graph* graph) {
   int V = graph->V;
   Edge result[MAX];
   int e = 0; // indexof result
   int i = 0; // index of sorted edges
   //Sort edges in non-decreasing order
   qsort(graph->edges, graph->E, sizeof(graph->edges[0]), compareEdges);
   Node* set = InitSet(V);
   while (e < V- 1 && i < graph->E) {
      Edge nextEdge = graph->edges[i++];
      int x = find(set, nextEdge.src);
      int y = find(set, nextEdge.dest);
      if (x != y) {
         result[e++] = nextEdge;
         Union(set, x, y);
      }
   }
   printMST(result, e);
   free(set);
}
```

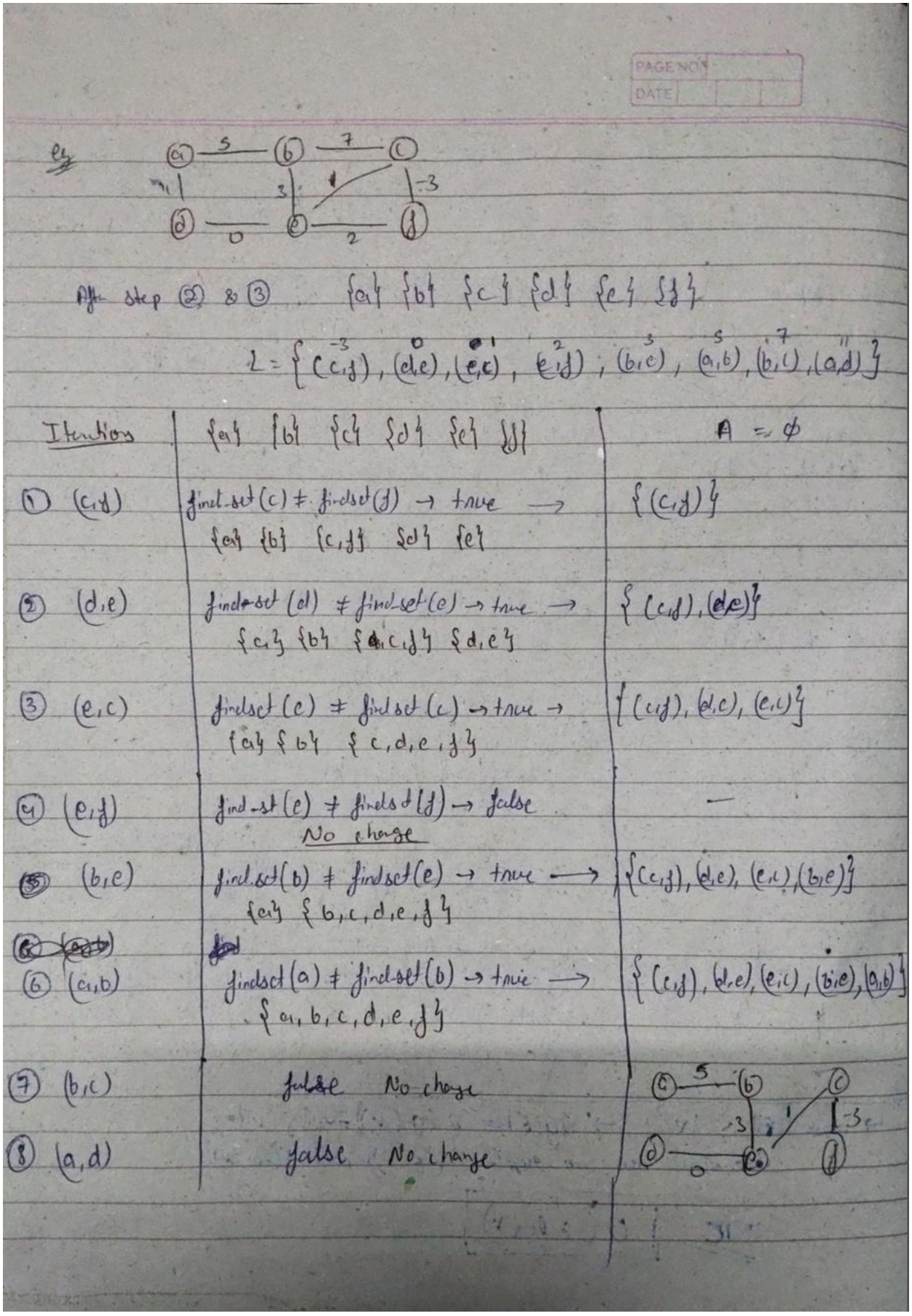
Output:

```
PS C:\Users\Ujjwal\Desktop\C\Graph_algo> cd "c
Constructed MST:
2 --> 5, weight = -3
3 --> 4, weight = 0
2 --> 4, weight = 1
1 --> 4, weight = 3
0 --> 1, weight = 5
MST weight: 6
```

- **Time complexity:**



- **Working:**


PAGE NO.
DATE
Eg
a b c d e f
5 7 3 1 -3 0 2
After step ② & ③ {a} {b} {c} {d} {e} {f}
L = {(c,f), (d,e), (e,c), (e,f), (b,c), (a,b), (b,f), (a,d)}
Iterations
A = ϕ
① (c,f) find_set(c) ≠ find_set(f) → true → {c,f} {a} {b} {c,f} {d} {e} {(c,f)}
② (d,e) find_set(d) ≠ find_set(e) → true → {c,f} {b} {d,c,f} {d,e} {(c,f), (d,e)}
③ (e,c) find_set(e) ≠ find_set(c) → true → {a} {b} {c,d,e,f} {(c,f), (d,e), (e,c)}
④ (e,f) find_set(e) ≠ find_set(f) → false No change —
⑤ (b,c) find_set(b) ≠ find_set(c) → true → {a} {b,c,d,e,f} {(c,f), (d,e), (e,c), (b,c)}
⑥ (a,b) find_set(a) ≠ find_set(b) → true → {a,b,c,d,e,f} {(c,f), (d,e), (e,c), (b,c), (a,b)}
⑦ (b,c) false No change
⑧ (a,d) false No change

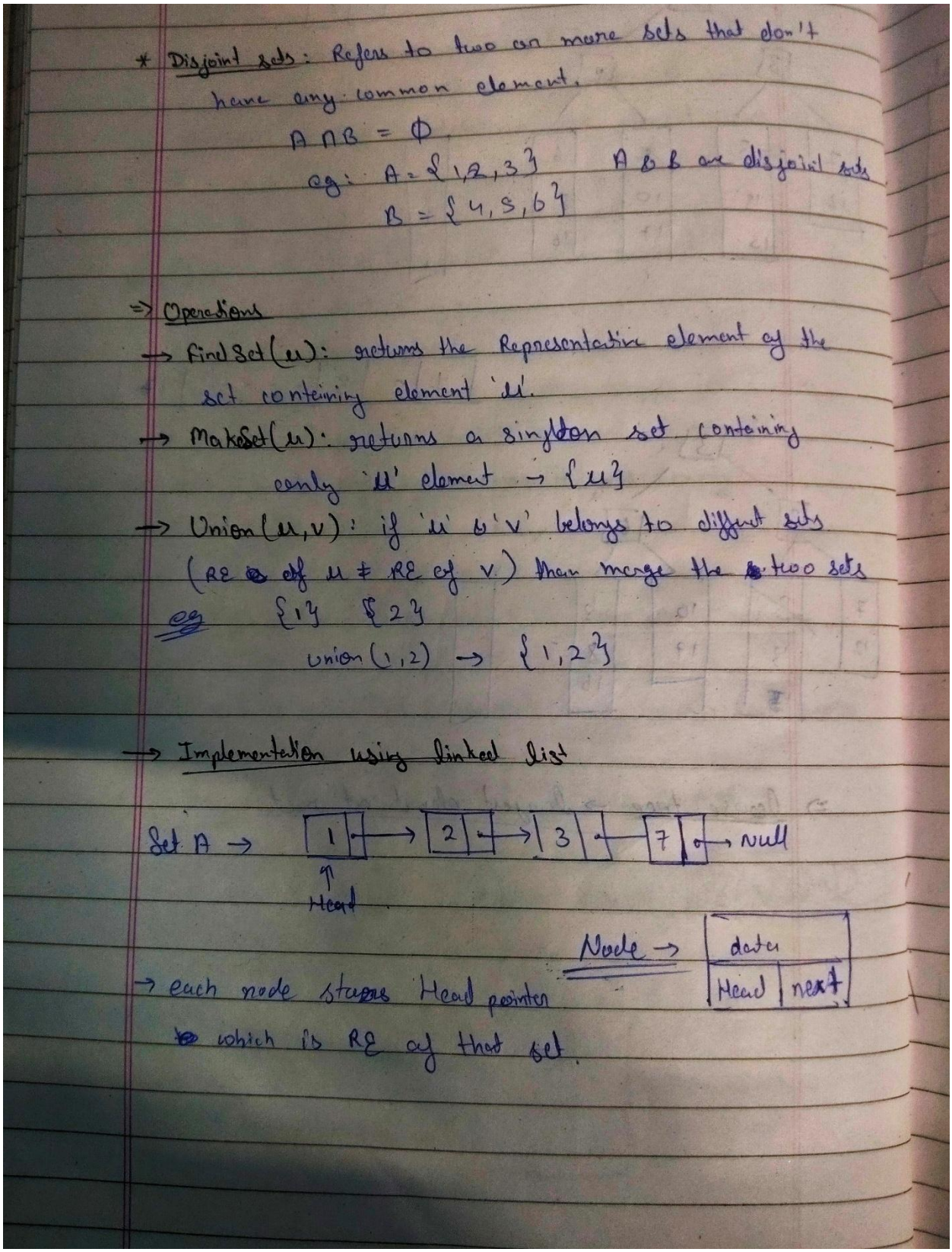
2) WAP for disjoint sets and show their working.

```
#include<stdio.h>
#include<stdlib.h>

typedef struct Node{
   int parent;
   int rank;
} Node;
Node *InitSet(int V){
   Node *set = (Node*)calloc(V, sizeof(Node));
   for(int i = 0; i < V; i++){
      set[i].parent = i;
      set[i].rank = 0;
   }
   return set;
}
int find(Node *set, int v){
   if(set[v].parent != v) set[v].parent = find(set, set[v].parent);
   return set[v].parent;
}
void Union(Node*set, int x, int y){
   int rootX = find(set, x);
   int rootY = find(set, y);
   if (rootX == rootY) return;
   if(set[rootX].rank > set[rootY].rank){
      set[rootY].parent = rootX;
   }else if(set[rootX].rank < set[rootY].rank){
      set[rootX].parent = rootY;
   }else{
      set[rootY].parent = rootX;
      set[rootX].rank++;
   }
}
int main() {
   int V = 5;
   Node *set = InitSet(V);
   Union(set, 0, 1);
   Union(set, 1, 2);
   Union(set, 3, 4);
   printf("Find(2): %d\n", find(set, 2));
   printf("Find(4): %d\n", find(set, 4));
   Union(set, 2, 3);
   printf("Find(4) after merging: %d\n", find(set, 4));
   free(set);
   return 0;
}
```

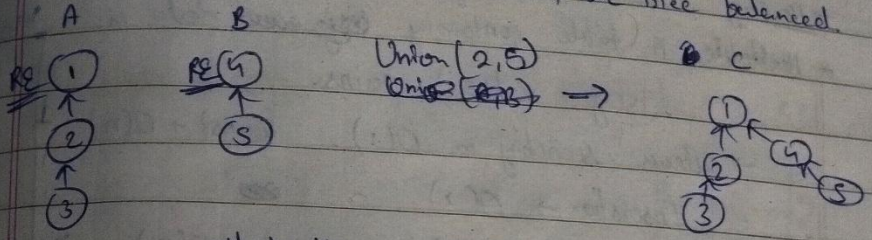
Output:

```
PS C:\Users\Ujjwal\Desktop\C\Disjo
Find(2): 0
Find(4): 3
Find(4) after merging: 0
PS C:\Users\Ujjwal\Desktop\C\Disjo
```

- **Time complexity and Working:**

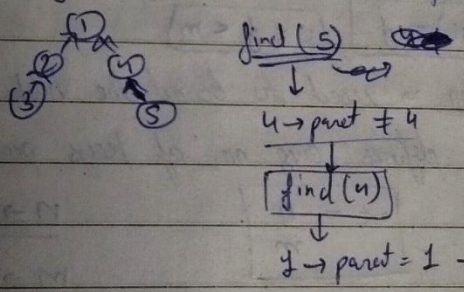


⇒ Rooted tree → set is represented as tree
→ each elemt points to the parent
→ Rank is used to keep the tree balanced.



attach the smaller tree to root of larger tree

⇒ Finset → uses path compression
recursively call finset(u) until u→parent = u



⇒ Time Complexity of Rooted tree method → $O(\alpha(v))$

α → Inverse ackermann function
(grows very slow) ≈ $O(1)$