Algorithm Analysis / Design

Practical Report

Kushagra Lakhwani
2021UCI8036

CSE (Internet of Things)
Semester 3

Contents

1 Counting Sort

1.1 Description

This sorting technique doesn't perform sorting by comparing elements. It performs sorting by counting objects having distinct key values like hashing. After that, it performs some arithmetic operations to calculate each object's index position in the output sequence. Counting sort is not used as a general-purpose sorting algorithm.

1.2 Algorithm

Algorithm 1 Counting Sort

```
procedure CountingSort(array, size)
    max = findMax(array, size)
    initialize countArray(max + 1) with 0
    for i = 0 to size do
        countArray[array[i]] + +
    end for
    for i = 1 to max do
        countArray[i] = countArray[i] + countArray[i − 1]
    end for
    for i = size to 1 do
        output[countArray[array[i]] − 1] = array[i]
        countArray[array[i]] = countArray[array[i]] - 1
    end for
end procedure
```

1.3 Code

```cpp
// Using counting sort to sort the elements in the basis of significant places
void countSort(vector<int> &arr, int n, int exp) {
    vector<int> output(n); // output array
    int i, count[10] = {0};

    // Store count of occurrences in count[]
    for (i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;

    // Change count[i] so that count[i] now contains actual
    //  position of this digit in output[]
    for (i = 1; i < 10; i++)
        count[i] += count[i - 1];

    // Build the output array
    for (i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] = arr[i];
        count[(arr[i] / exp) % 10]--;
```

```
    }

    // Copy the output array to arr[], so that arr[] now
    // contains sorted numbers according to current digit
    for (i = 0; i < n; i++)
        arr[i] = output[i];
}
```

1.4 Output

```
Sorted array: 1 2 3 4 5 6 7 8 9 10
```

2 Radix Sort

2.1 Description

Radix sort is the linear sorting algorithm that is used for integers. In Radix sort, there is digit by digit sorting is performed that is started from the least significant digit to the most significant digit.

2.2 Algorithm

Algorithm 2 Radix Sort

1: **procedure** RADIXSORT($array, size$)
2: max = findMax($array, size$)
3: **for** $exp = 1$ to max **do**
4: countSort($array, size, exp$)
5: **end for**
6: **end procedure**

2.3 Code

```
/** An implementation of Radix sort. */
#include <iostream>
#include <vector>
using namespace std;

extern void countSort(vector<int> &arr, int n, int exp);

// * Function to get the largest element from an array
int getMax(vector<int> &arr, int n) {
    int mx = arr[0];
    for (int i = 1; i < n; i++)
        if (arr[i] > mx)
```

```
            mx = arr[i];
    return mx;
}

void radixsort(vector<int> &arr, int n) {
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);

    // Do counting sort for every digit. Note that instead
    // of passing digit number, exp is passed. exp is 10^i
    // where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}

// * Driver program to test above functions
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    vector<int> arr(n);
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    radixsort(arr, n);
    cout << "Sorted array: ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

2.4 Output

```
Enter the number of elements: 5
Enter the elements: 170 45 75 90 802
Sorted array: 45 75 90 170 802
```

3 Bucket Sort

3.1 Description

Bucket sort is a sorting algorithm that separate the elements into multiple groups said to be buckets. Elements in bucket sort are first uniformly divided into groups called buckets, and then they are sorted by any other sorting algorithm. After that, elements are gathered in a sorted manner.

3.2 Algorithm

Algorithm 3 Bucket Sort

procedure BUCKETSORT(A)
  Let $B[0....n-1]$ be a new array
  n = length of A
  for $i \leftarrow 0$ to $n-1$ **do**
    make $B[i]$ an empty list
  end for
  for $i \leftarrow 1$ to n **do**
    insert $A[i]$ into list $B[na[i]]$
  end for
  for $i \leftarrow 0$ to $n-1$ **do**
    sort list $B[i]$ with insertion-sort
  end for
  Concatenate lists $B[0], B[1], ..., B[n-1]$ together in order
end procedure

3.3 Code

```cpp
#include <iostream>
#include <vector>
using namespace std;

void bucketSort(float arr[], int n) {
    // Create n empty buckets
    vector<float> b[n];

    // Put array elements in different buckets
    for (int i = 0; i < n; i++) {
        int bi = n * arr[i]; // Index in bucket
        b[bi].push_back(arr[i]);
    }

    // Sort individual buckets
    for (int i = 0; i < n; i++)
        sort(b[i].begin(), b[i].end());

    // Concatenate all buckets into arr[]
    int index = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < b[i].size(); j++)
            arr[index++] = b[i][j];
}

// * Driver program to test above functions
int main() {
    int n;
```

```cpp
    cout << "Enter the number of elements: ";
    cin >> n;
    float arr[n];
    cout << "Enter the elements: ";
    for (int i = 0; i < n; i++)
        cin >> arr[i];
    bucketSort(arr, n);

    cout << "Sorted array is ";
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

3.4 Output

```
Enter the number of elements: 5
Enter the elements: 0.897 0.565 0.656 0.1234 0.665
Sorted array is 0.1234 0.565 0.656 0.665 0.897
```

4 Quick Sort

4.1 Description

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot. After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.

4.2 Algorithm

Algorithm 4 Quick Sort

1: **procedure** QuickSort($array, left, right$)
2:     **if** $left < right$ **then**
3:         $pivot =$ partition$(array, left, right)$
4:         QuickSort$(array, left, pivot - 1)$
5:         QuickSort$(array, pivot + 1, right)$
6:     **end if**
7: **end procedure**

4.3 Code

```cpp
#include <iostream>
using namespace std;

/* This function takes last element as pivot, places
the pivot element at its correct position in sorted
array, and places all smaller (smaller than pivot)
to left of pivot and all greater elements to right
of pivot */

int partition (int arr[], int low, int high)
{
    int pivot = arr[high]; // pivot
    int i = (low - 1); // Index of smaller element

    for (int j = low; j <= high - 1; j++) {
        // If current element is smaller than or
        // equal to pivot
        if (arr[j] <= pivot) {
            i++; // increment index of smaller element
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}

// The main function that implements QuickSort
// arr[] --> Array to be sorted,
// low --> Starting index,
// high --> Ending index

void quickSort(int arr[], int low, int high)
{
    if (low < high) {
        /* pi is partitioning index, arr[p] is now
        at right place */
        int pi = partition(arr, low, high);

        // Separately sort elements before
        // partition and after partition
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}

// Function to print an array
void printArray(int arr[], int size)
{
    for (int i = 0; i < size; i++)
        cout << arr[i] << " ";
    cout << endl;
}
```

```
int main()
{
    int arr[] = { 10, 7, 8, 9, 1, 5 };
    int n = sizeof(arr) / sizeof(arr[0]);
    quickSort(arr, 0, n - 1);
    cout << "Sorted array: n";
    printArray(arr, n);
    return 0;
}
```

4.4 Output

```
Enter the number of elements: 5
Enter the elements: 170 45 75 90 802
Sorted array is 45 75 90 170 802
```

5 Merge Sort

5.1 Description

Merge sort is similar to the quick sort algorithm as it uses the divide and conquer approach to sort the elements. It is one of the most popular and efficient sorting algorithm. It divides the given list into two equal halves, calls itself for the two halves and then merges the two sorted halves. We have to define the merge() function to perform the merging.

The sub-lists are divided again and again into halves until the list cannot be divided further. Then we combine the pair of one element lists into two-element lists, sorting them in the process. The sorted two-element pairs is merged into the four-element lists, and so on until we get the sorted list.

5.2 Algorithm

Algorithm 5 Merge Sort

1: **procedure** MergeSort($array, left, right$)
2: **if** $left < right$ **then**
3: $middle = (left + right)/2$
4: MergeSort($array, left, middle$)
5: MergeSort($array, middle + 1, right$)
6: merge($array, left, middle, right$)
7: **end if**
8: **end procedure**

5.3 Code

```cpp
#include <iostream>
using namespace std;

// merge two sorted subarrays arr[low..mid] and arr[mid+1..high]
void merge(int arr[], int low, int mid, int high)
{
    // create a temp array
    int temp[high - low + 1];

    // crawlers for both intervals and for temp
    int i = low, j = mid + 1, k = 0;

    // traverse both arrays and in each iteration add smaller of both elements in temp
    while (i <= mid && j <= high)
    {
        if (arr[i] <= arr[j])
            temp[k++] = arr[i++];
        else
            temp[k++] = arr[j++];
    }

    // add elements left in the first interval
    while (i <= mid)
        temp[k++] = arr[i++];

    // add elements left in the second interval
    while (j <= high)
        temp[k++] = arr[j++];

    // copy temp to original interval
    for (int i = low; i <= high; i++)
        arr[i] = temp[i - low];
}

// Recursive implementation of merge sort
void mergeSort(int arr[], int low, int high)
{
    // base condition
    if (low == high)
        return;

    // find the mid value
    int mid = (low + (high - low) / 2);

    // recursively split the array into two halves until it can no more be split
    mergeSort(arr, low, mid);
    mergeSort(arr, mid + 1, high);

    merge(arr, low, mid, high);
}

int main()
```

```
{
    int arr[] = { 3, 5, 8, 4, 1, 9, -2 };
    int n = sizeof(arr) / sizeof(arr[0]);

    mergeSort(arr, 0, n - 1);
    for (int i = 0; i < n; i++)
        cout << arr[i] << " ";
    return 0;
}
```

5.4 Output

```
Enter the number of elements: 5
Enter the elements: 170 45 75 90 802
Sorted array is 45 75 90 170 802
```

6 Heap Sort

6.1 Description

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

6.2 Algorithm

Algorithm 6 Heap Sort

1: **procedure** HeapSort($array, n$)
2: buildMaxHeap($array, n$)
3: **for** $i \leftarrow n$ to 1 **do**
4: swap $array[0]$ and $array[i]$
5: heapify($array, i, 0$)
6: **end for**
7: **end procedure**

6.3 Code

```
#include <iostream>
using namespace std;

// Function to heapify the tree
void heapify(int arr[], int n, int i)
{
```

```
    int largest = i; // Initialize largest as root
    int l = 2 * i + 1; // left = 2*i + 1
    int r = 2 * i + 2; // right = 2*i + 2

    // If left child is larger than root
    if (l < n && arr[l] > arr[largest])
        largest = l;

    // If right child is larger than largest so far
    if (r < n && arr[r] > arr[largest])
        largest = r;

    // If largest is not root
    if (largest != i) {
        swap(&arr[i], &arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Function to build a Max-Heap from the given array
void buildHeap(int arr[], int n)
{
    // Index of last non-leaf node
    int startIdx = (n / 2) - 1;

    // Perform reverse level order traversal from last non-leaf node and heapify each node
    for (int i = startIdx; i >= 0; i--) {
        heapify(arr, n, i);
    }
}

// Function to sort an array using Heap Sort
void heapSort(int arr[], int n)
{
    // Build a max heap
    buildHeap(arr, n);

    // Heap sort
    for (int i = n - 1; i > 0; i--) {
        // Swap
        swap(&arr[0], &arr[i]);
        // Heapify root element to get highest element at root again
        heapify(arr, i, 0);
    }
}

// Function to print an array
void printArray(int arr[], int n)
{
    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";
    cout << endl;
```

```
}

// Driver code
int main()
{
    int arr[] = { 1, 12, 9, 5, 6, 10 };
    int n = sizeof(arr) / sizeof(arr[0]);

    heapSort(arr, n);

    cout << "Sorted array is " << endl;
    printArray(arr, n);
}
```

6.4 Output

```
Enter the number of elements: 5
Enter the elements: 170 45 75 90 802
Sorted array is 45 75 90 170 802
```

7 Depth First Search

7.1 Description

The depth-first search (DFS) algorithm starts with the initial node of graph G and goes deeper until we find the goal node or the node with no children.

7.2 Algorithm

Algorithm 7 Depth First Search

1: **procedure** DFS(G, v)
2: mark v as visited
3: **for** each unvisited neighbor w of v **do**
4: DFS(G, w)
5: **end for**
6: **end procedure**

7.3 Code

```cpp
#include <iostream>
#include <vector>
#include <stack>
using namespace std;

struct Edge {
    int src, dest;
};

class Graph
{
public:
    vector<vector<int>> adjList;

    Graph(vector<Edge> const &edges, int N)
    {
        // resize the vector to N elements of type vector<int>
        adjList.resize(N);

        // add edges to the directed graph
        for (auto &edge: edges)
        {
            // insert at the end
            adjList[edge.src].push_back(edge.dest);
            // Uncomment below line for undirected graph
            // adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform DFS on graph starting from vertex v
bool DFS(Graph const &graph, int v, vector<bool> &discovered)
{
    // create a stack used to do DFS
    stack<int> stack;

    // push the source node into stack
    stack.push(v);

    // loop till stack is empty
    while (!stack.empty())
    {
        // we pop a vertex from stack and print it
        v = stack.top();
        stack.pop();

        // if the vertex is already discovered yet, ignore it
        if (discovered[v])
            continue;

        // we print the vertex and mark it as discovered
        discovered[v] = true;
```

```
        cout << v << " ";

        // do for every edge (v -> u)
        for (int u : graph.adjList[v])
        {
            // if vertex u is not discovered, push it into stack
            if (!discovered[u])
                stack.push(u);
        }
    }

    return true;
}

// Depth First Search Algorithm
int main()
{
    // vector of graph edges as per above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
        // vertex 0, 13 and 14 are single nodes
    };

    // Number of nodes in the graph
    int N = 15;

    // create a graph from edges
    Graph graph(edges, N);

    // stores vertex is discovered or not
    vector<bool> discovered(N);

    // Do DFS traversal from all undiscovered nodes to
    // cover all unconnected components of graph
    for (int i = 0; i < N; i++)
        if (discovered[i] == false)
            // start DFS traversal from vertex i
            DFS(graph, i, discovered);

    return 0;
}
```

7.4 Output

```
1 4 8 12 7 11 3 2 6 10 9 5
```

8 Breadth First Search

8.1 Description

Breadth-first search is a graph traversal algorithm that starts traversing the graph from the root node and explores all the neighbouring nodes. Then, it selects the nearest node and explores all the unexplored nodes.

8.2 Algorithm

Algorithm 8 Breadth First Search

```
procedure BFS(G, s)
    for each vertex u ∈ V[G] do
        d[u] ← ∞
        π[u] ← NIL
    end for
    d[s] ← 0
    Q ← ∅
    ENQUEUE(Q, s)
    while Q ≠ ∅ do
        u ← DEQUEUE(Q)
        for each vertex v ∈ Adj[u] do
            if d[v] = ∞ then
                d[v] ← d[u] + 1
                π[v] ← u
                ENQUEUE(Q, v)
            end if
        end for
    end while
end procedure
```

8.3 Code

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

struct Edge {
    int src, dest;
};

class Graph
{
public:
    vector<vector<int>> adjList;

    Graph(vector<Edge> const &edges, int N)
```

```
    {
        // resize the vector to N elements of type vector<int>
        adjList.resize(N);

        // add edges to the directed graph
        for (auto &edge: edges)
        {
            // insert at the end
            adjList[edge.src].push_back(edge.dest);
            // Uncomment below line for undirected graph
            // adjList[edge.dest].push_back(edge.src);
        }
    }
};

// Perform BFS on graph starting from vertex v
bool BFS(Graph const &graph, int v, vector<bool> &discovered)
{
    queue<int> q;
    // mark source vertex as discovered
    discovered[v] = true;
    // push source vertex into the queue
    q.push(v);

    while (!q.empty())
    {
        // dequeue front node and print it
        v = q.front();
        q.pop();
        cout << v << " ";

        // do for every edge (v -> u)
        for (int u : graph.adjList[v])
        {
            // if vertex u is not discovered yet
            if (!discovered[u])
            {
                discovered[u] = true;
                q.push(u);
            }
        }
    }
}

int main()
{
    // vector of graph edges as per above diagram
    vector<Edge> edges = {
        {1, 2}, {1, 3}, {1, 4}, {2, 5}, {2, 6}, {5, 9},
        {5, 10}, {4, 7}, {4, 8}, {7, 11}, {7, 12}
        // vertex 0, 13 and 14 are single nodes
    };

    // total number of nodes in the graph
```

```
    int N = 15;

    // build a graph from the given edges
    Graph graph(edges, N);

    // to keep track of whether a vertex is discovered or not
    vector<bool> discovered(N);

    // Do BFS traversal from all undiscovered nodes to
    // cover all unconnected components of graph
    for (int i = 0; i < N; i++)
        if (discovered[i] == false)
            // start BFS traversal from vertex i
            BFS(graph, i, discovered);

    return 0;
}
```

8.4 Output

```
1 4 8 12 7 11 3 2 6 10 9 5
```

9 Prim's Algorithm

9.1 Description

Prim's algorithm is a greedy algorithm that finds a minimum spanning tree for a weighted undirected graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized. The algorithm operates by building this tree one vertex at a time, from an arbitrary starting vertex, at each step adding the cheapest possible connection from the tree to another vertex.

9.2 Algorithm

Algorithm 9 Prim's Algorithm

procedure PRIM(G)
  for each vertex $u \in V[G]$ **do**
    $key[u] \leftarrow \infty$
    $\pi[u] \leftarrow NIL$
  end for
  $key[s] \leftarrow 0$
  $Q \leftarrow V[G]$
  while $Q \neq \emptyset$ **do**
    $u \leftarrow EXTRACT - MIN(Q)$
    for each vertex $v \in Adj[u]$ **do**
      if $v \in Q$ and $w(u, v) < key[v]$ **then**
        $\pi[v] \leftarrow u$
        $key[v] \leftarrow w(u, v)$
      end if
    end for
  end while
end procedure

9.3 Code

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

class Graph
{
public:
    // construct a vector of vectors of Edge to represent an adjacency list
    vector<vector<Edge>> adjList;

    // Graph Constructor
    Graph(vector<Edge> const &edges, int N)
    {
        // resize the vector to N elements of type vector<Edge>
        adjList.resize(N);

        // add edges to the directed graph
        for (auto &edge: edges)
        {
            // insert at the end
            adjList[edge.src].push_back(edge);
        }
    }
};

```

```cpp
// Function to print adjacency list representation of graph
void printGraph(Graph const &graph, int N)
{
    for (int i = 0; i < N; i++)
    {
        // print current vertex number
        cout << i << " --> ";

        // print all neighboring vertices of vertex i
        for (Edge e : graph.adjList[i])
            cout << "(" << e.dest << ", " << e.weight << ") ";

        cout << endl;
    }
}

// Function to perform Prim's algorithm on a graph
int prim(Graph const &graph, int N)
{
    // create a min-heap using std::set in STL
    // we use `std::pair` as the data type of the element stored in the heap
    // the first element of the pair stores the key and the second element stores the vertex number
    set<pair<int, int>> pq;

    // create a vector to store key of the vertices which have been found by the algorithm
    // initialize all keys to infinite (INT_MAX)
    vector<int> key(N, INT_MAX);

    // create a vector to store parent node of a vertex
    // it keeps track of the minimum spanning tree
    vector<int> parent(N, -1);

    // to keep track of vertices included in MST
    vector<bool> inMST(N, false);

    // insert source vertex in the set and make its key 0
    pq.insert(make_pair(0, 0));
    key[0] = 0;

    // run till `pq` is not empty
    while (!pq.empty())
    {
        // find the vertex with the minimum key
        // extract it from the set
        int u = pq.begin()->second;
        pq.erase(pq.begin());

        // include vertex in MST
        inMST[u] = true;

        // do for each adjacent vertex `v` of `u`
        for (Edge e : graph.adjList[u])
```

```cpp
        {
            int v = e.dest;
            int weight = e.weight;

            // if `v` is not in MST and weight of (u, v) is smaller than current key of `v`
            if (!inMST[v] && key[v] > weight)
            {
                // update the key of `v` in the set
                pq.erase(make_pair(key[v], v));
                key[v] = weight;
                pq.insert(make_pair(key[v], v));

                // update the parent of `v`
                parent[v] = u;
            }
        }
    }

    // print edges of the MST
    for (int i = 1; i < N; i++)
        cout << parent[i] << " - " << i << endl;

    // return weight of the MST
    return accumulate(key.begin(), key.end(), 0);
}

int main()
{
    // vector of graph edges as per above diagram
    vector<Edge> edges =
    {
        // (x, y, w) -> edge from x to y having weight w
        { 0, 1, 6 }, { 1, 2, 7 }, { 2, 0, 5 }, { 2, 1, 4 },
        { 3, 2, 10 }, { 5, 4, 1 }, { 4, 5, 3 }, { 4, 0, 1 }
    };

    // Number of nodes in the graph
    int N = 6;

    // construct graph
    Graph graph(edges, N);

    // print adjacency list representation of graph
    printGraph(graph, N);

    // run Prim's algorithm on graph
    cout << "Weight of the MST is " << prim(graph, N);

    return 0;
}
```

9.4 Output

```
0 --> (1, 6) (2, 5) (4, 1)
1 --> (2, 7) (0, 6)
2 --> (1, 4) (0, 5) (3, 10)
3 --> (2, 10)
4 --> (5, 3) (0, 1)
5 --> (4, 1)
0 - 4
4 - 5
2 - 1
1 - 2
3 - 2
Weight of the MST is 16
```

10 Kruskal's Algorithm

10.1 Description

Kruskal's algorithm is a minimum spanning tree algorithm that finds an edge of the least possible weight that connects any two trees in the forest. It is a greedy algorithm in graph theory as it finds a minimum spanning tree for a connected weighted graph adding increasing cost arcs at each step.

10.2 Algorithm

Algorithm 10 Kruskal's Algorithm

procedure KRUSKAL(G)
    $A \leftarrow \emptyset$
    for each vertex $v \in V[G]$ **do**
        $MAKE - SET(v)$
    end for
    Sort the edges of$E[G]$into nondecreasing order by weight w
    for each edge $(u, v) \in E[G],$ taken in nondecreasing order by weight **do**
        if $FIND - SET(u) \neq FIND - SET(v)$ **then**
            $A \leftarrow A \cup \{(u, v)\}$
            $UNION(u, v)$
        end if
    end for
    $returnA$
end procedure

10.3 Code

```
#include <iostream>
#include <vector>

// data structure to store graph edges
struct Edge {
    int src, dest, weight;
};

// class to represent a graph object
extern class Graph;

// Function to print adjacency list representation of graph
extern void printGraph(Graph const &graph, int N);

int kruskal(Graph const &graph, int N)
{
    // stores edge list of the MST
    vector<Edge> MST;

    // initialize a Union-Find data structure
    DisjointSet ds(N);

    // get all edges of the graph in a vector
    vector<Edge> edges;
    for (int i = 0; i < N; i++)
    {
        for (Edge e : graph.adjList[i])
            edges.push_back(e);
    }

    // sort edges by increasing weight
    sort(edges.begin(), edges.end(), [](Edge const &e1, Edge const &e2) {
        return e1.weight < e2.weight;
    });

    // process edges in increasing weight order
    for (Edge const &edge: edges)
    {
        // check if the selected edge creates a cycle or not
        if (ds.find(edge.src) != ds.find(edge.dest))
        {
            // include the current edge in MST
            MST.push_back(edge);

            // merge two components
            ds.merge(edge.src, edge.dest);
        }
    }

    // print MST
    for (Edge const &e: MST)
        cout << "(" << e.src << ", " << e.dest << ", " << e.weight << ") ";
```

```cpp

    return 0;
}

// main function
int main()
{
    // vector of graph edges as per above diagram
    vector<Edge> edges = {
        {0, 1, 7}, {1, 2, 8}, {0, 3, 5}, {1, 3, 9}, {1, 4, 7}, {2, 4, 5},
        {3, 4, 15}, {3, 5, 6}, {4, 5, 8}, {4, 6, 9}, {5, 6, 11}
    };

    // total number of nodes in the graph
    int N = 7;

    // build a graph from the given edges
    Graph graph(edges, N);

    // print adjacency list representation of graph
    printGraph(graph, N);

    // run Kruskal's algorithm on the graph
    kruskal(graph, N);

    return 0;
}
```

10.4 Output

```
0 --> (1, 7) (3, 5)
1 --> (0, 7) (2, 8) (3, 9) (4, 7)
2 --> (1, 8) (4, 5)
3 --> (0, 5) (1, 9) (4, 15) (5, 6)
4 --> (1, 7) (2, 5) (3, 15) (5, 8) (6, 9)
5 --> (3, 6) (4, 8) (6, 11)
6 --> (4, 9) (5, 11)
(0, 3, 5) (2, 4, 5) (3, 5, 6) (0, 1, 7) (1, 4, 7) (4, 6, 9)
```