Practical -10

Name:shubham shivraj Suryawanshi
Reg. No:2020BIT004

Practical no:10 implement the following algo.

Dijekistra algo.

Huffmen coding

Write a Algorithm with complete Simulation

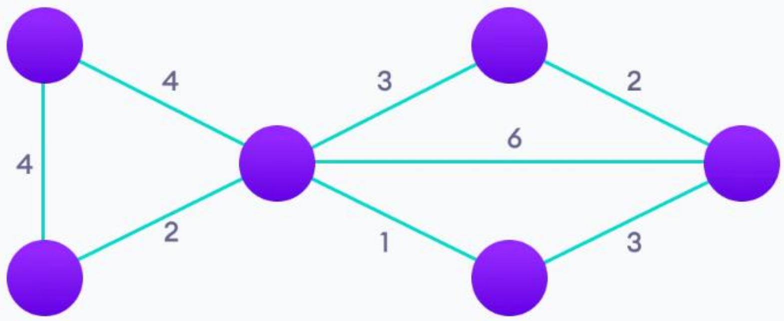
1) Dijekistra algo.

```
#include <stdio.h>
#define INFINITY 9999
#define MAX 10

void Dijkstra(int Graph[MAX][MAX], int n, int start);

void Dijkstra(int Graph[MAX][MAX], int n, int start) {
 int cost[MAX][MAX], distance[MAX], pred[MAX];
 int visited[MAX], count, mindistance, nextnode, i, j;

 for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
   if (Graph[i][j] == 0)
    cost[i][j] = INFINITY;
   else
    cost[i][j] = Graph[i][j];

 for (i = 0; i < n; i++) {
  distance[i] = cost[start][i];
  pred[i] = start;
  visited[i] = 0;
 }

 distance[start] = 0;
 visited[start] = 1;
 count = 1;

 while (count < n - 1) {
  mindistance = INFINITY;

  for (i = 0; i < n; i++)
   if (distance[i] < mindistance && !visited[i]) {
    mindistance = distance[i];
    nextnode = i;
   }

  visited[nextnode] = 1;
  for (i = 0; i < n; i++)
   if (!visited[i])
    if (mindistance + cost[nextnode][i] < distance[i]) {
     distance[i] = mindistance + cost[nextnode][i];
     pred[i] = nextnode;
    }
  count++;
 }

 for (i = 0; i < n; i++)
  if (i != start) {
   printf("\nDistance from source to %d: %d", i, distance[i]);
  }
}
int main() {
 int Graph[MAX][MAX], i, j, n, u;
 n = 7;
```

```
  Graph[0][0] = 0;
  Graph[0][1] = 0;
  Graph[0][2] = 1;
  Graph[0][3] = 2;
  Graph[0][4] = 0;
  Graph[0][5] = 0;
  Graph[0][6] = 0;

  Graph[1][0] = 0;
  Graph[1][1] = 0;
  Graph[1][2] = 2;
  Graph[1][3] = 0;
  Graph[1][4] = 0;
  Graph[1][5] = 3;
  Graph[1][6] = 0;

  Graph[2][0] = 1;
  Graph[2][1] = 2;
  Graph[2][2] = 0;
  Graph[2][3] = 1;
  Graph[2][4] = 3;
  Graph[2][5] = 0;
  Graph[2][6] = 0;

  Graph[3][0] = 2;
  Graph[3][1] = 0;
  Graph[3][2] = 1;
  Graph[3][3] = 0;
  Graph[3][4] = 0;
  Graph[3][5] = 0;
  Graph[3][6] = 1;

  Graph[4][0] = 0;
  Graph[4][1] = 0;
  Graph[4][2] = 3;
  Graph[4][3] = 0;
  Graph[4][4] = 0;
  Graph[4][5] = 2;
  Graph[4][6] = 0;

  Graph[5][0] = 0;
  Graph[5][1] = 3;
  Graph[5][2] = 0;
  Graph[5][3] = 0;
  Graph[5][4] = 2;
  Graph[5][5] = 0;
  Graph[5][6] = 1;

  Graph[6][0] = 0;
  Graph[6][1] = 0;
  Graph[6][2] = 0;
  Graph[6][3] = 1;
  Graph[6][4] = 0;
  Graph[6][5] = 1;
  Graph[6][6] = 0;

  u = 0;
  Dijkstra(Graph, n, u);

  return 0;
}
```
Output:

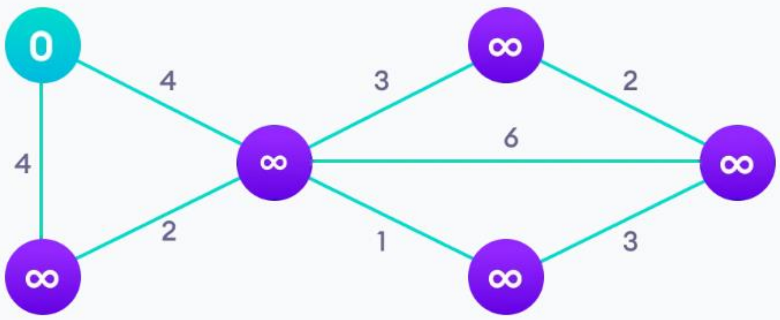
```
Distance from source to 1: 3
Distance from source to 2: 1
Distance from source to 3: 2
Distance from source to 4: 4
Distance from source to 5: 4
Distance from source to 6: 3
PS D:\Assignments TY\DAA\Codes\output>
```

Simulation:



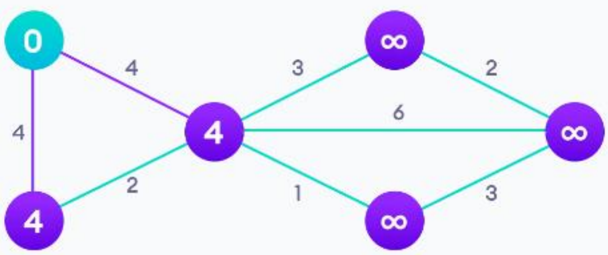
Step: 1

Start with a weighted graph



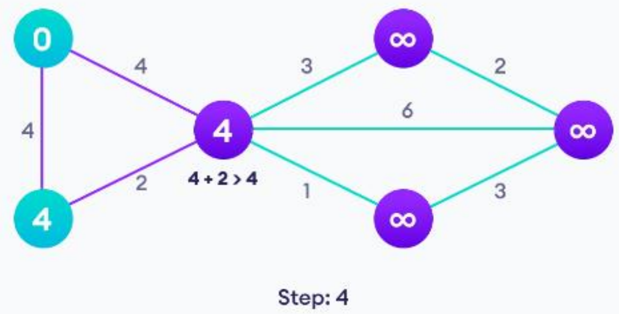
Step: 2

Choose a starting vertex and assign infinity path values to all other devices

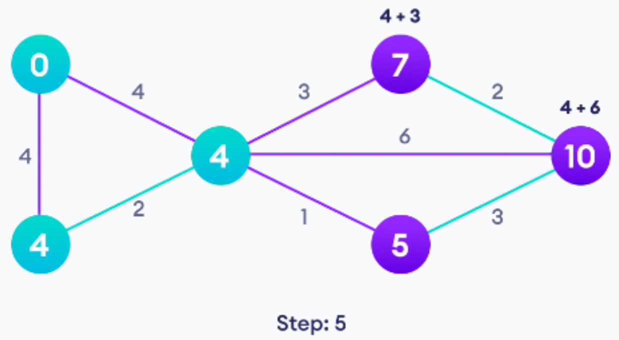


Step: 3

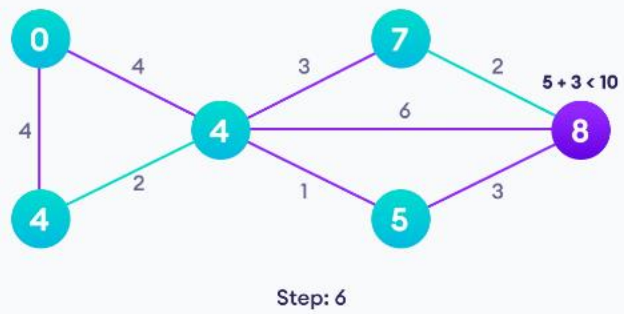
Go to each vertex and update its path length



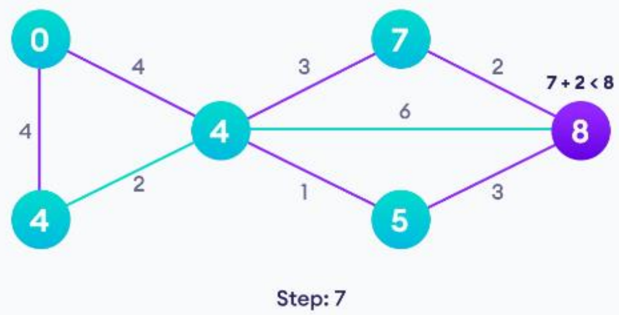
If the path length of the adjacent vertex is lesser than new path length, don't update it



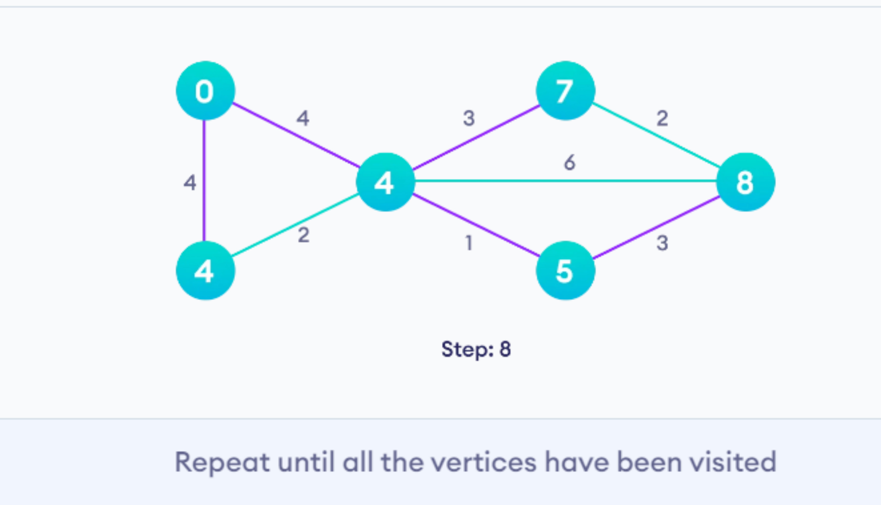
Avoid updating path lengths of already visited vertices



After each iteration, we pick the unvisited vertex with the least path length. So we choose 5 before 7



Notice how the rightmost vertex has its path length updated twice



2) Huffmen coding

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

#define MAX_TREE_HT 50

struct MinHNode {
  char item;
  unsigned freq;
  struct MinHNode *left, *right;
};

struct MinHeap {
  unsigned size;
  unsigned capacity;
  struct MinHNode **array;
};

struct MinHNode *newNode(char item, unsigned freq) {
  struct MinHNode *temp = (struct MinHNode *)malloc(sizeof(struct MinHNode));

  temp->left = temp->right = NULL;
  temp->item = item;
  temp->freq = freq;

  return temp;
}

struct MinHeap *createMinH(unsigned capacity) {
  struct MinHeap *minHeap = (struct MinHeap *)malloc(sizeof(struct MinHeap));

  minHeap->size = 0;

  minHeap->capacity = capacity;

  minHeap->array = (struct MinHNode **)malloc(minHeap->capacity * sizeof(struct MinHNode *));
  return minHeap;
}

void swapMinHNode(struct MinHNode **a, struct MinHNode **b) {
  struct MinHNode *t = *a;
```

```
  *a = *b;
  *b = t;
}

void minHeapify(struct MinHeap *minHeap, int idx) {
  int smallest = idx;
  int left = 2 * idx + 1;
  int right = 2 * idx + 2;

  if (left < minHeap->size && minHeap->array[left]->freq < minHeap->array[smallest]->freq)
    smallest = left;

  if (right < minHeap->size && minHeap->array[right]->freq < minHeap->array[smallest]->freq)
    smallest = right;

  if (smallest != idx) {
    swapMinHNode(&minHeap->array[smallest], &minHeap->array[idx]);
    minHeapify(minHeap, smallest);
  }
}

int checkSizeOne(struct MinHeap *minHeap) {
  return (minHeap->size == 1);
}

struct MinHNode *extractMin(struct MinHeap *minHeap) {
  struct MinHNode *temp = minHeap->array[0];
  minHeap->array[0] = minHeap->array[minHeap->size - 1];

  --minHeap->size;
  minHeapify(minHeap, 0);

  return temp;
}

void insertMinHeap(struct MinHeap *minHeap, struct MinHNode *minHeapNode) {
  ++minHeap->size;
  int i = minHeap->size - 1;

  while (i && minHeapNode->freq < minHeap->array[(i - 1) / 2]->freq) {
    minHeap->array[i] = minHeap->array[(i - 1) / 2];
    i = (i - 1) / 2;
  }
  minHeap->array[i] = minHeapNode;
}

void buildMinHeap(struct MinHeap *minHeap) {
  int n = minHeap->size - 1;
  int i;

  for (i = (n - 1) / 2; i >= 0; --i)
    minHeapify(minHeap, i);
}

int isLeaf(struct MinHNode *root) {
  return !(root->left) && !(root->right);
}
```

```
struct MinHeap *createAndBuildMinHeap(char item[], int freq[], int size) {
  struct MinHeap *minHeap = createMinH(size);

  for (int i = 0; i < size; ++i)
    minHeap->array[i] = newNode(item[i], freq[i]);

  minHeap->size = size;
  buildMinHeap(minHeap);

  return minHeap;
}

struct MinHNode *buildHuffmanTree(char item[], int freq[], int size) {
  struct MinHNode *left, *right, *top;
  struct MinHeap *minHeap = createAndBuildMinHeap(item, freq, size);

  while (!checkSizeOne(minHeap)) {
    left = extractMin(minHeap);
    right = extractMin(minHeap);

    top = newNode('$', left->freq + right->freq);

    top->left = left;
    top->right = right;

    insertMinHeap(minHeap, top);
  }
  return extractMin(minHeap);
}

void printHCodes(struct MinHNode *root, int arr[], int top) {
  if (root->left) {
    arr[top] = 0;
    printHCodes(root->left, arr, top + 1);
  }
  if (root->right) {
    arr[top] = 1;
    printHCodes(root->right, arr, top + 1);
  }
  if (isLeaf(root)) {
    printf("  %c   | ", root->item);
    printArray(arr, top);
  }
}

void HuffmanCodes(char item[], int freq[], int size) {
  struct MinHNode *root = buildHuffmanTree(item, freq, size);

  int arr[MAX_TREE_HT], top = 0;

  printHCodes(root, arr, top);
}

void printArray(int arr[], int n) {
  int i;
  for (i = 0; i < n; ++i)
```

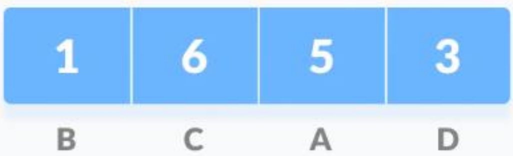
```
    printf("%d", arr[i]);

  printf("\n");
}

int main() {
  char arr[] = {'A', 'B', 'C', 'D'};
  int freq[] = {5, 1, 6, 3};

  int size = sizeof(arr) / sizeof(arr[0]);

  printf(" Char | Huffman code ");
  printf("\n--------------------\n");

  HuffmanCodes(arr, freq, size);
}
```

Output:

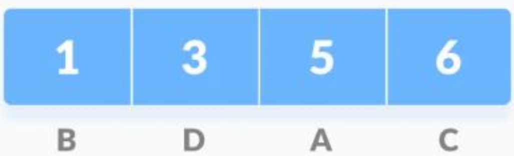
```
Char | Huffman code
--------------------
  C  | 0
  B  | 100
  D  | 101
  A  | 11
```

Simulation:

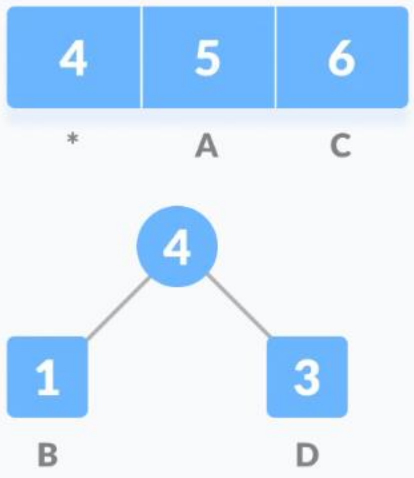
Initial string



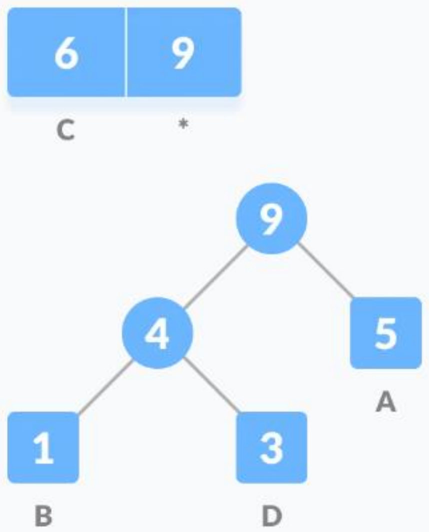
Frequency of string



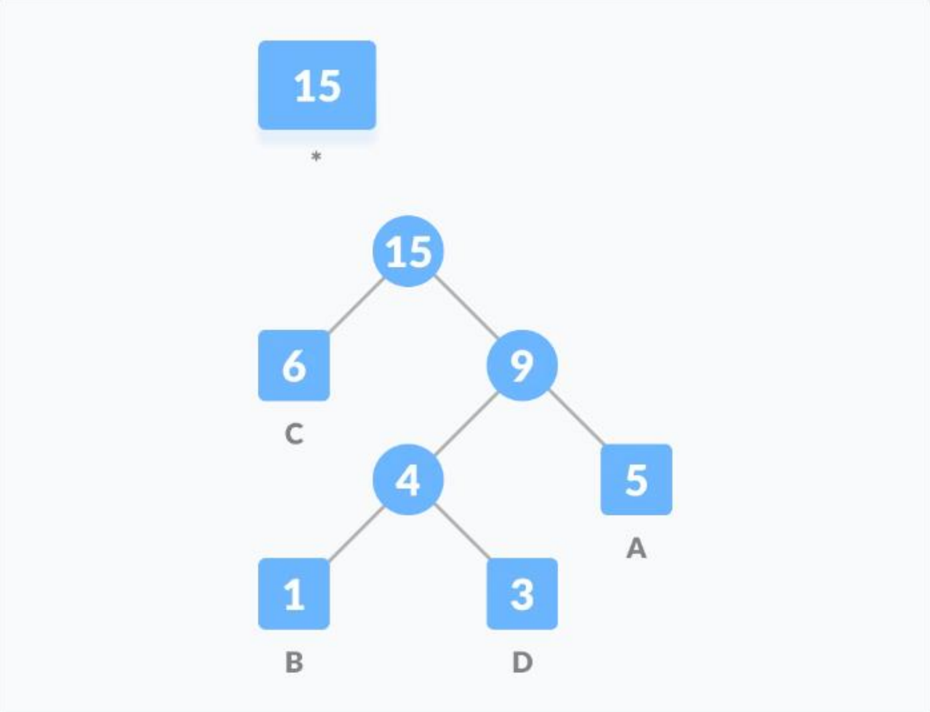
Characters sorted according to the frequency



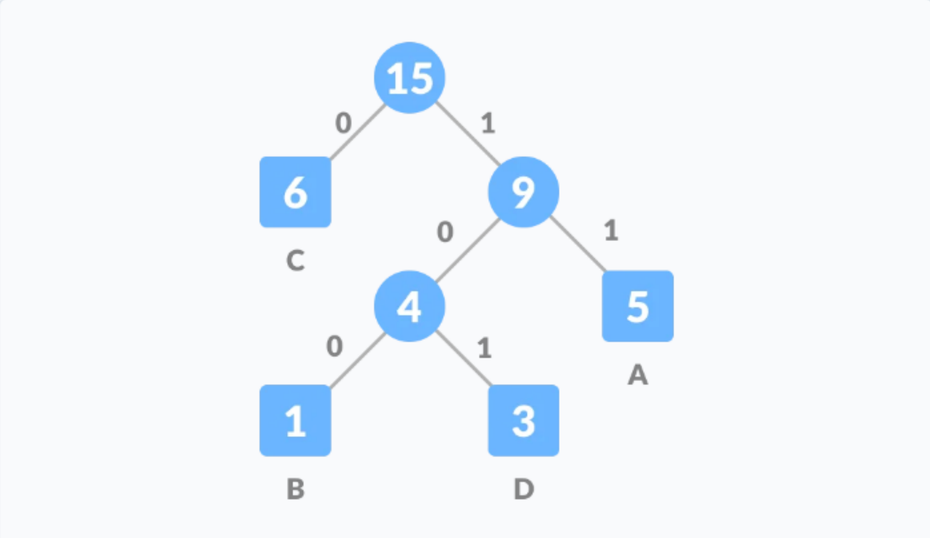
Getting the sum of the least numbers



Repeat steps 3 to 5 for all the characters.

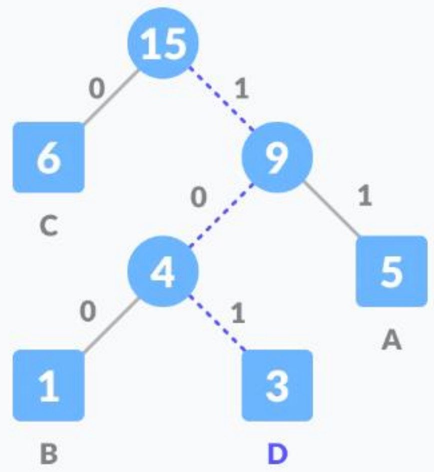


Repeat steps 3 to 5 for all the characters.



Assign 0 to the left edge and 1 to the right edge

Character	Frequency	Code	Size
A	5	11	5*2 = 10
B	1	100	1*3 = 3
C	6	0	6*1 = 6
D	3	101	3*3 = 9
4 * 8 = 32 bits	15 bits		28 bits



Decoding