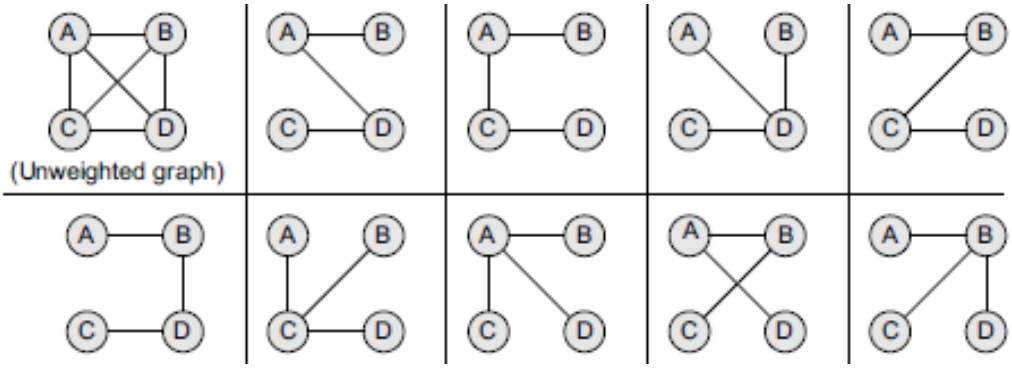
ΠΑΝΕΠΙΣΤΗΜΙΟ ΘΕΣΣΑΛΙΑΣ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ
ΧΕΙΜΕΡΙΝΟ ΕΞΑΜΗΝΟ
ΜΑΘΗΜΑ: ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ

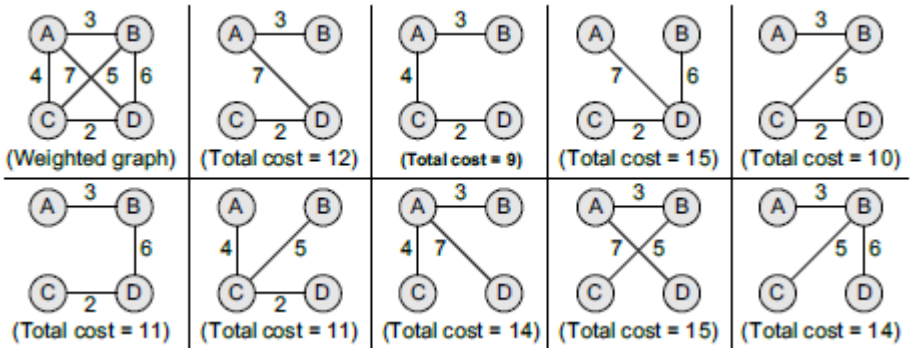
ΕΡΓΑΣΤΗΡΙΟ 10 – ΣΗΜΕΙΩΣΕΙΣ

Δένδρα Επικάλυψης Ελάχιστου Κόστους (Minimum Spanning Trees)

Τα **δένδρα επικάλυψης (spanning trees)** είναι δένδρα τα οποία καλύπτουν όλους τους κόμβους ενός γράφου. Ένας γράφος μπορεί να έχει περισσότερα από ένα δένδρα επικάλυψης. Αν έχουν αποδοθεί βάρη στις ακμές του γράφου, τότε το **δένδρο επικάλυψης ελαχίστου κόστους (minimum spanning tree - MST)** είναι το δένδρο επικάλυψης που έχει το μικρότερο άθροισμα βαρών από όλα τα δένδρα επικάλυψης.

Παραδείγματα:





Ο Αλγόριθμος του Prim

Ο αλγόριθμος υιοθετεί την άπληστη μέθοδο και διατηρεί τρία σύνολα κόμβων:

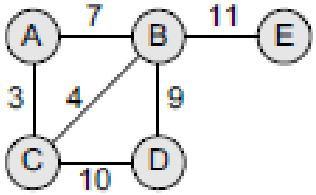
- **Tree vertices**. Οι κόμβοι αποτελούν τμήμα του MST.
- **Fringe vertices**. Οι κόμβοι αυτοί δεν είναι τμήμα του MST αλλά είναι γείτονες κόμβων που ανήκουν στο MST.
- **Unseen vertices**. Όλοι οι υπόλοιποι κόμβοι.

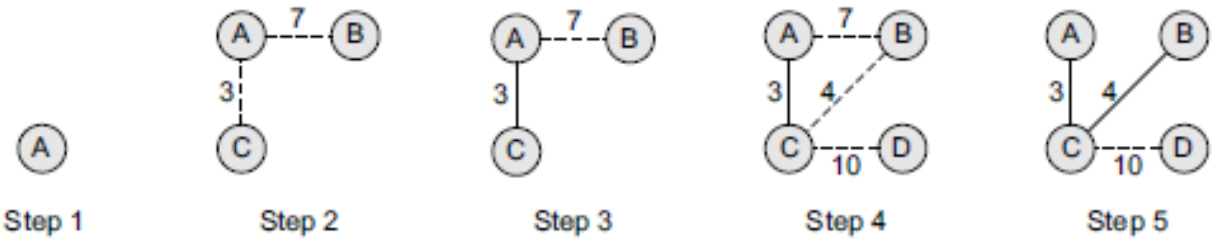
Ο αλγόριθμος έχει ως εξής:

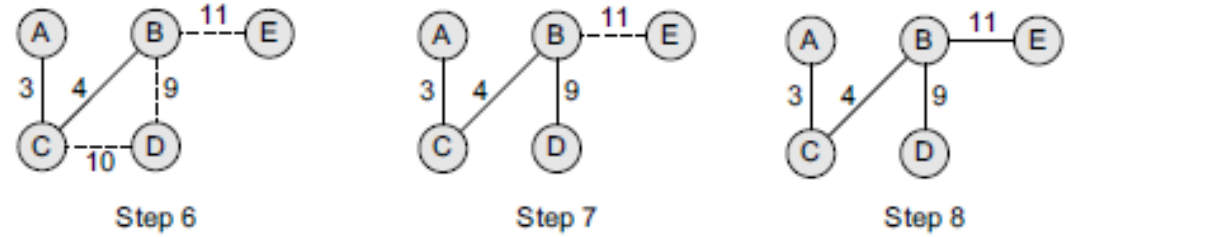
```
Step 1: Select a starting vertex
Step 2: Repeat Steps 3 and 4 until there are fringe vertices
Step 3:     Select an edge e connecting the tree vertex and
            fringe vertex that has minimum weight
Step 4:     Add the selected edge and the vertex to the
            minimum spanning tree T
        [END OF LOOP]
Step 5: EXIT
```

Παραδείγματα εκτέλεσης.

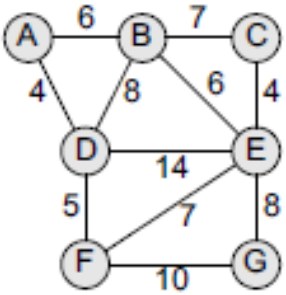
Α. Εκκίνηση από τον κόμβο Α

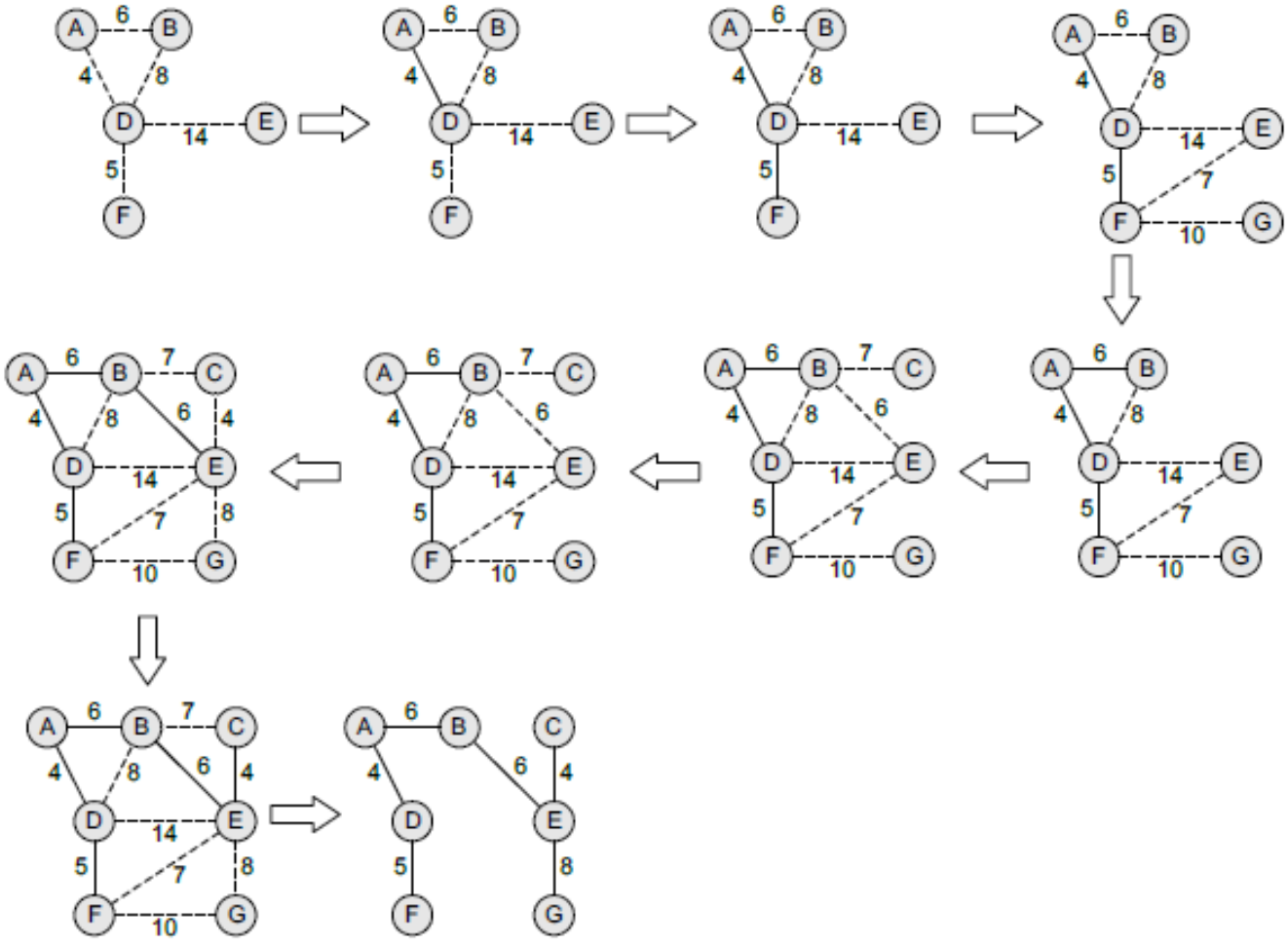






Β. Εκκίνηση από τον κόμβο D





Code 1

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

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with minimum key value, from
// the set of vertices not yet included in MST
int minKey(int key[], bool mstSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (mstSet[v] == false && key[v] < min)
```

```
        min = key[v], min_index = v;

  return min_index;
}

// A utility function to print the constructed MST stored in parent[]
int printMST(int parent[], int n, int graph[V][V])
{
  printf("Edge   Weight\n");
  for (int i = 1; i < V; i++)
    printf("%d - %d    %d \n", parent[i], i, graph[i][parent[i]]);
}

// Function to construct and print MST for a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    int parent[V]; // Array to store constructed MST
    int key[V];   // Key values used to pick minimum weight edge in cut
    bool mstSet[V];  // To represent set of vertices not yet included in MST

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
      key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    key[0] = 0;     // Make key 0 so that this vertex is picked as first vertex
    parent[0] = -1; // First node is always root of MST

    // The MST will have V vertices
    for (int count = 0; count < V-1; count++)
    {
      // Pick the minimum key vertex from the set of vertices
      // not yet included in MST
      int u = minKey(key, mstSet);

      // Add the picked vertex to the MST Set
      mstSet[u] = true;

      // Update key value and parent index of the adjacent vertices of
      // the picked vertex. Consider only those vertices which are not yet
      // included in MST
      for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent vertices of m
        // mstSet[v] is false for vertices not yet included in MST
        // Update the key only if graph[u][v] is smaller than key[v]
       if (graph[u][v] && mstSet[v] == false && graph[u][v] <  key[v])
          parent[v]  = u, key[v] = graph[u][v];
    }
```

```
    // print the constructed MST
    printMST(parent, V, graph);
}


// driver program to test above function
int main()
{
  /* Let us create the following graph
        2   3
     (0)--(1)--(2)
      |   / \   |
     6| 8/   \5 |7
      | /     \ |
     (3)-------(4)
           9         */
  int graph[V][V] = {{0, 2, 0, 6, 0},
                     {2, 0, 3, 8, 5},
                     {0, 3, 0, 0, 7},
                     {6, 8, 0, 0, 9},
                     {0, 5, 7, 9, 0},
                    };

   // Print the solution
   primMST(graph);

   return 0;
}
```

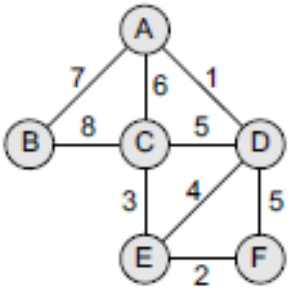
Execution: https://visualgo.net/mst

Ο Αλγόριθμος του Kruskal

Ο αλγόριθμος αναζητά το υποσύνολο ακμών που σχηματίζουν ένα δένδρο που περιλαμβάνει όλους τους κόμβους. Αν ο γράφος δεν είναι συνδεδεμένος, ο αλγόριθμος επιστρέφει ένα δάσος από MSTs. Ο αλγόριθμος υιοθετεί μια ουρά με προτεραιότητα στην οποία οι ακμές με το μικρότερο βάρος έχουν μεγαλύτερη προτεραιότητα έναντι των υπολοίπων. Ο αλγόριθμος έχει ως εξής:

```
Step 1: Create a forest in such a way that each graph is a separate
        tree.
Step 2: Create a priority queue Q that contains all the edges of the
        graph.
Step 3: Repeat Steps 4 and 5 while Q is NOT EMPTY
Step 4:      Remove an edge from Q
Step 5: IF the edge obtained in Step 4 connects two different trees,
        then Add it to the forest (for combining two trees into one
        tree).
        ELSE
             Discard the edge
Step 6: END
```

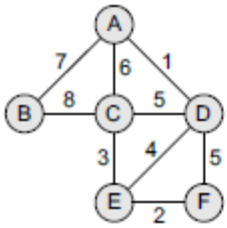
Παράδειγμα



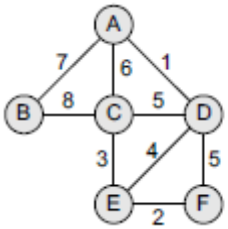
Αρχικά έχουμε:

$$F = \{\{A\}, \{B\}, \{C\}, \{D\}, \{E\}, \{F\}\}$$
$$MST = \{\}$$
$$Q = \{(A, D), (E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)\}$$

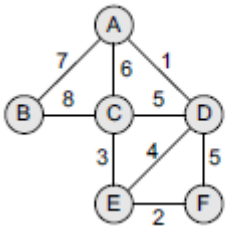
Οπότε



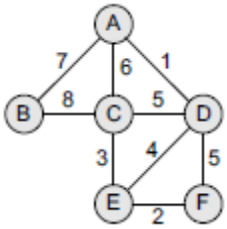
F = {{A, D}, {B}, {C}, {E}, {F}}
MST = {A, D}
Q = {(E, F), (C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}



F = {{A, D}, {B}, {C}, {E, F}}
MST = {(A, D), (E, F)}
Q = {(C, E), (E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}



F = {{A, D}, {B}, {C, E, F}}
MST = {(A, D), (C, E), (E, F)}
Q = {(E, D), (C, D), (D, F), (A, C), (A, B), (B, C)}

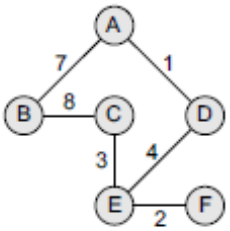


F = {{A, C, D, E, F}, {B}}
MST = {(A, D), (C, E), (E, F), (E, D)}
Q = {(C, D), (D, F), (A, C), (A, B), (B, C)}

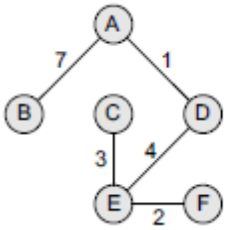
$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(D, F), (A, C), (A, B), (B, C)\}$$

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, C), (A, B), (B, C)\}$$

$$F = \{\{A, C, D, E, F\}, \{B\}\}$$
$$MST = \{(A, D), (C, E), (E, F), (E, D)\}$$
$$Q = \{(A, B), (B, C)\}$$



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {(B, C)}



F = {A, B, C, D, E, F}
MST = {(A, D), (C, E), (E, F), (E, D), (A, B)}
Q = {}

Code 2

```
// C++ program for Kruskal's algorithm to find Minimum Spanning Tree
// of a given connected, undirected and weighted graph
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// a structure to represent a weighted edge in graph
struct Edge
{
   int src, dest, weight;
};

// a structure to represent a connected, undirected and weighted graph
struct Graph
{
   // V-> Number of vertices, E-> Number of edges
   int V, E;

   // graph is represented as an array of edges. Since the graph is
   // undirected, the edge from src to dest is also edge from dest
```

```
    // to src. Both are counted as 1 edge here.
    struct Edge* edge;
};

// Creates a graph with V vertices and E edges
struct Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;

    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );

    return graph;
}

// A structure to represent a subset for union-find
struct subset
{
    int parent;
    int rank;
};

// A utility function to find set of an element i
// (uses path compression technique)
int find(struct subset subsets[], int i)
{
    // find root and make root as parent of i (path compression)
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);

    return subsets[i].parent;
}

// A function that does union of two sets of x and y
// (uses union by rank)
void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);

    // Attach smaller rank tree under root of high rank tree
    // (Union by Rank)
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;

    // If ranks are same, then make one as root and increment
    // its rank by one
```

```
    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

// Compare two edges according to their weights.
// Used in qsort() for sorting an array of edges
int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

// The main function to construct MST using Kruskal's algorithm
void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];  // Tnis will store the resultant MST
    int e = 0;  // An index variable, used for result[]
    int i = 0;  // An index variable, used for sorted edges

    // Step 1:  Sort all the edges in non-decreasing order of their weight
    // If we are not allowed to change the given graph, we can create a copy of
    // array of edges
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);

    // Allocate memory for creating V ssubsets
    struct subset *subsets =
        (struct subset*) malloc( V * sizeof(struct subset) );

    // Create V subsets with single elements
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }

    // Number of edges to be taken is equal to V-1
    while (e < V - 1)
    {
        // Step 2: Pick the smallest edge. And increment the index
        // for next iteration
        struct Edge next_edge = graph->edge[i++];

        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
```

```
        // If including this edge does't cause cycle, include it
        // in result and increment the index of result for next edge
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
        // Else discard the next_edge
    }

    // print the contents of result[] to display the built MST
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
        printf("%d -- %d == %d\n", result[i].src, result[i].dest,
                                        result[i].weight);
    return;
}

// Driver program to test above functions
int main()
{
    /* Let us create following weighted graph
             10
        0--------1
        | \     |
       6|   5\   |15
        |      \ |
        2--------3
            4       */
    int V = 4;  // Number of vertices in graph
    int E = 5;  // Number of edges in graph
    struct Graph* graph = createGraph(V, E);


    // add edge 0-1
    graph->edge[0].src = 0;
    graph->edge[0].dest = 1;
    graph->edge[0].weight = 10;

    // add edge 0-2
    graph->edge[1].src = 0;
    graph->edge[1].dest = 2;
    graph->edge[1].weight = 6;

    // add edge 0-3
    graph->edge[2].src = 0;
    graph->edge[2].dest = 3;
    graph->edge[2].weight = 5;

    // add edge 1-3
```

```
    graph->edge[3].src = 1;
    graph->edge[3].dest = 3;
    graph->edge[3].weight = 15;

    // add edge 2-3
    graph->edge[4].src = 2;
    graph->edge[4].dest = 3;
    graph->edge[4].weight = 4;

    KruskalMST(graph);

    return 0;
}
```

Execution: https://visualgo.net/mst

Συντομότερα Μονοπάτια (Shortest Paths)

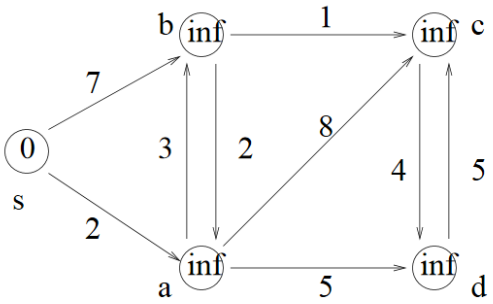
Πέρα από την υιοθέτηση των MSTs για την εύρεση συντομότερων μονοπατιών πάνω σε ένα γράφο, στη βιβλιογραφία έχουν προταθεί και επιπλέον αλγόριθμοι για το σκοπό αυτό. Οι αλγόριθμοι συντομότερου μονοπατιού (shortest path algorithms) επιτελούν το σκοπό αυτό. Ένας από αυτούς είναι ο αλγόριθμος του Dijkstra. Ο αλγόριθμος εξάγει το συντομότερο μονοπάτι δοσμένων ενός γράφου και ενός κόμβου από όπου θα ξεκινήσει η αναζήτηση του μονοπατιού. Ο αλγόριθμος αναθέτει μια ετικέτα σε κάθε κόμβο που δείχνει την απόστασή του από τον κόμβο – πηγή. Χρησιμοποιούνται δύο ειδών ετικέτες: οι **προσωρινές** και οι **μόνιμες**. Οι προσωρινές ετικέτες **ανατίθενται σε κόμβους στους οποίους δεν έχουμε φτάσει ακόμα** ενώ οι μόνιμες μπαίνουν **σε κόμβους στους οποίους έχει φτάσει ο αλγόριθμος και είναι γνωστή η απόστασή τους από τον κόμβο – πηγή**. Κάθε κόμβος έχει μόνο μια ετικέτα και ποτέ και τις δύο μαζί. Η εκτέλεση του αλγορίθμου θα επιφέρει ένα από τα ακόλουθα αποτελέσματα:

1. Αν ο κόμβος προορισμού έχει ετικέτα, τότε η ετικέτα θα αναπαριστά την απόσταση του προορισμού από τον κόμβο – πηγή.
2. Αν ο κόμβος προορισμού δεν έχει ετικέτα, τότε δεν υπάρχει μονοπάτι από τον κόμβο – πηγή προς τον προορισμό.

```
Dijkstra(G,w,s)
{                                          % Initialize
    for (each u ∈ V)
    {
        d[u] = ∞;
        color[u] =white;
    }
    d[s] = 0;
    pred[s] = NIL;
    Q = (queue with all vertices);

    while (Non-Empty(Q))                   % Process all vertices
    {
        u = Extract-Min(Q);                % Find new vertex
        for (each v ∈ Adj[u])
            if (d[u] + w(u,v) < d[v])      % If estimate improves
            {
                d[v] = d[u] + w(u,v);          relax
                Decrease-Key(Q,v,d[v]);
                pred[v] = u;
            }
        color[u] =black;
    }
}
```

Παράδειγμα εκτέλεσης:

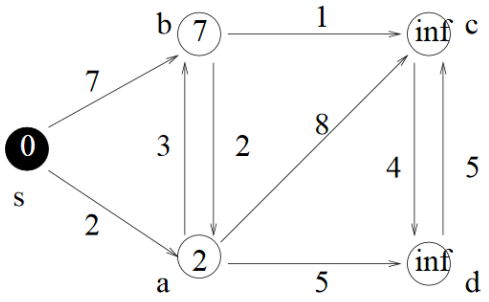


Step 0: Initialization.

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞
$pred[v]$	nil	nil	nil	nil	nil
$color[v]$	W	W	W	W	W

Priority Queue:

v	s	a	b	c	d
$d[v]$	0	∞	∞	∞	∞

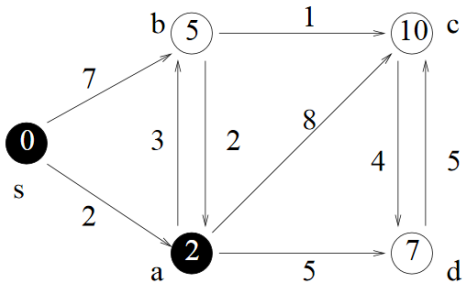


Step 1: As $Adj[s] = \{a, b\}$, work on a and b and update information.

v	s	a	b	c	d
$d[v]$	0	2	7	∞	∞
$pred[v]$	nil	s	s	nil	nil
$color[v]$	B	W	W	W	W

Priority Queue:

v	a	b	c	d
$d[v]$	2	7	∞	∞

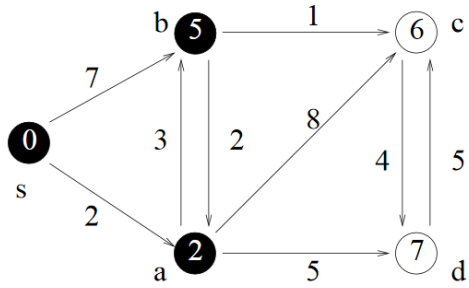


Step 2: After Step 1, a has the minimum key in the priority queue. As $Adj[a] = \{b, c, d\}$, work on b, c, d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	10	7
$pred[v]$	nil	s	a	a	a
$color[v]$	B	B	W	W	W

Priority Queue:

v	b	c	d
$d[v]$	5	10	7

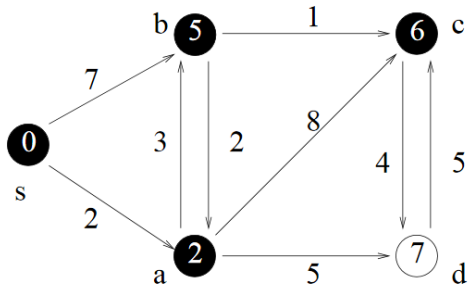


Step 3: After Step 2, b has the minimum key in the priority queue. As $Adj[b] = \{a, c\}$, work on a, c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	W	W

Priority Queue:

v	c	d
$d[v]$	6	7

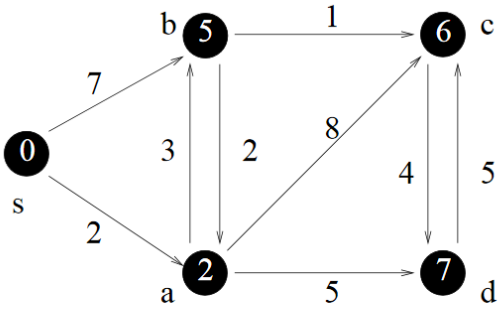


Step 4: After Step 3, c has the minimum key in the priority queue. As $Adj[c] = \{d\}$, work on d and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	W

Priority Queue:

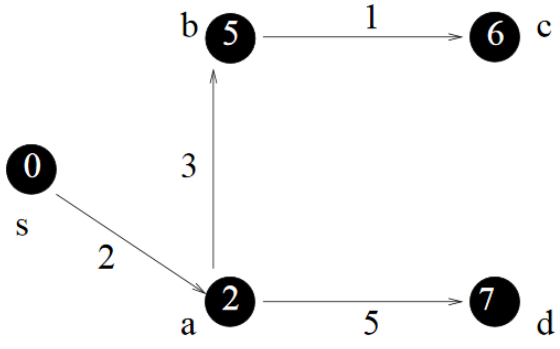
v	d
$d[v]$	7



Step 5: After Step 4, d has the minimum key in the priority queue. As $Adj[d] = \{c\}$, work on c and update information.

v	s	a	b	c	d
$d[v]$	0	2	5	6	7
$pred[v]$	nil	s	a	b	a
$color[v]$	B	B	B	B	B

Priority Queue: $Q = \emptyset$.



Visual example: https://visualgo.net/sssp

Code 3

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

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum distance value, from
// the set of vertices not yet included in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
  // Initialize min value
  int min = INT_MAX, min_index;

  for (int v = 0; v < V; v++)
    if (sptSet[v] == false && dist[v] <= min)
       min = dist[v], min_index = v;

  return min_index;
}

// A utility function to print the constructed distance array
int printSolution(int dist[], int n)
{
  printf("Vertex   Distance from Source\n");
  for (int i = 0; i < V; i++)
     printf("%d \t\t %d\n", i, dist[i]);
}

// Funtion that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V];     // The output array.  dist[i] will hold the shortest
                     // distance from src to i

    bool sptSet[V]; // sptSet[i] will true if vertex i is included in shortest
                    // path tree or shortest distance from src to i is finalized

    // Initialize all distances as INFINITE and stpSet[] as false
    for (int i = 0; i < V; i++)
       dist[i] = INT_MAX, sptSet[i] = false;

    // Distance of source vertex from itself is always 0
    dist[src] = 0;
```

```
    // Find shortest path for all vertices
    for (int count = 0; count < V-1; count++)
    {
      // Pick the minimum distance vertex from the set of vertices not
      // yet processed. u is always equal to src in first iteration.
      int u = minDistance(dist, sptSet);

      // Mark the picked vertex as processed
      sptSet[u] = true;

      // Update dist value of the adjacent vertices of the picked vertex.
      for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet, there is an edge from
        // u to v, and total weight of path from src to  v through u is
        // smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX
                               && dist[u]+graph[u][v] < dist[v])
          dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist, V);
}

// driver program to test above function
int main()
{
   /* Let us create the example graph discussed above */
   int graph[V][V] = {{0, 4, 0, 0, 0, 0, 0, 8, 0},
                      {4, 0, 8, 0, 0, 0, 0, 11, 0},
                      {0, 8, 0, 7, 0, 4, 0, 0, 2},
                      {0, 0, 7, 0, 9, 14, 0, 0, 0},
                      {0, 0, 0, 9, 0, 10, 0, 0, 0},
                      {0, 0, 4, 14, 10, 0, 2, 0, 0},
                      {0, 0, 0, 0, 0, 2, 0, 1, 6},
                      {8, 11, 0, 0, 0, 0, 1, 0, 7},
                      {0, 0, 2, 0, 0, 0, 6, 7, 0}
                     };

    dijkstra(graph, 0);

    return 0;
}
```

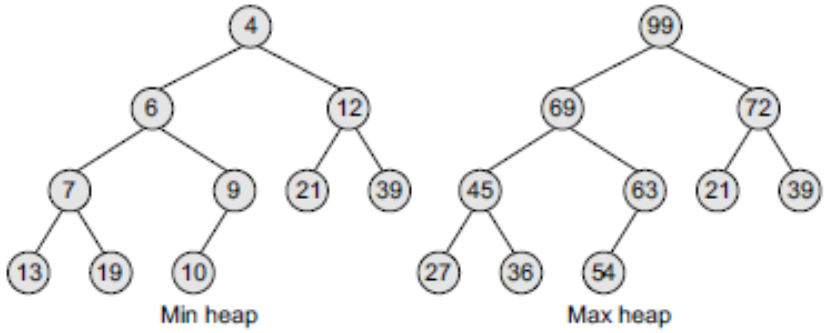
Σωροί (Heaps)

Ένας **δυαδικός σωρός (binary heap)** είναι ένα δυαδικό δένδρο στο οποίο κάθε κόμβος ικανοποιεί την ιδιότητα του σωρού που είναι:

Αν ο Β είναι παιδί του Α, τότε το κλειδί του Α είναι μεγαλύτερο ή ίσο από το κλειδί του Β

Αυτό σημαίνει πως τα στοιχεία σε κάθε κόμβο θα είναι μεγαλύτερα ή ίσα από τα στοιχεία που βρίσκονται στο αριστερό και στο δεξιό παιδί. Έτσι, ο ριζικός κόμβος έχει το μεγαλύτερο κλειδί σε ολόκληρο το σωρό. Αυτού του είδους οι σωροί είναι γνωστοί ως **max-heaps**. Αν αντιστρέψουμε την ανίσωση της ιδιότητας του σωρού, τότε θα έχουμε ένα **min-heap** στον οποίο κάθε κόμβο έχει μικρότερο κλειδί από τα στοιχεία που βρίσκονται στο αριστερό και στο δεξιό παιδί. Φυσικά, σε τέτοιου είδους σωρούς, ο ριζικός κόμβος έχει το μικρότερο κλειδί από όλους τους υπόλοιπους.

Παραδείγματα:



Τα στοιχεία ενός σωρού μπορούν να αποθηκευτούν ακολουθιακά σε ένα πίνακα. Για την αποθήκευση διατηρούμε τους κανόνες που ισχύουν για ένα δυαδικό δένδρο. Αν λοιπόν ένα στοιχείο είναι στη θέση **i** στον πίνακα, τότε το αριστερό παιδί είναι αποθηκευμένο στη θέση **2i** και το δεξιό παιδί είναι στη θέση **2i+1**. Αντίστοιχα, ο κόμβος που είναι αποθηκευμένος στη θέση **i**, έχει τον πατρικό του στη θέση **i/2**.

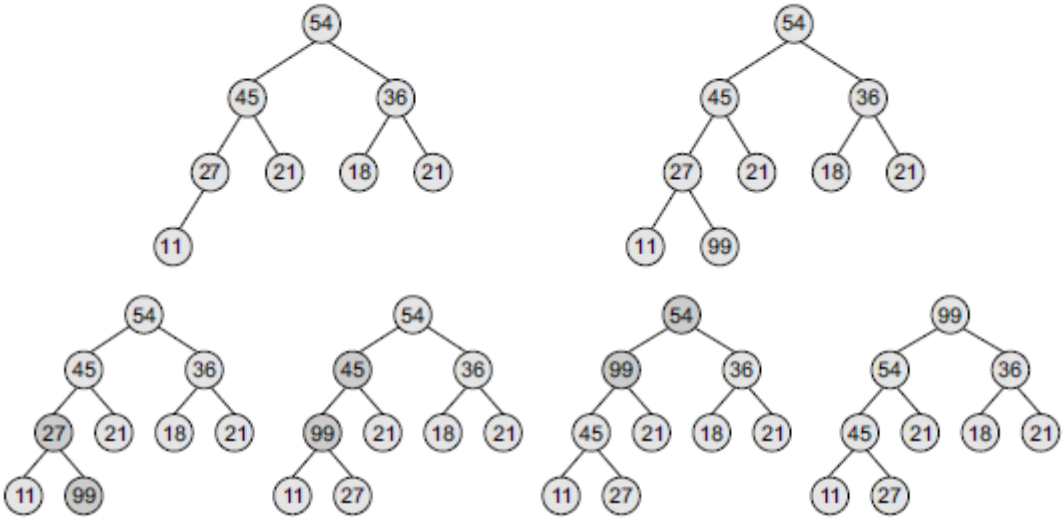
Οι σωροί αποτελούν μια πολύ καλή δομή για την διατήρηση ουρών με προτεραιότητα.

Εισαγωγή στοιχείου. Η διαδικασία αποτελείται από δύο βήματα. Στο πρώτο βήμα, ***προσθέτουμε τη νέα τιμή στο τέλος του σωρού*** έτσι ώστε ο σωρός να αποτελεί ένα έγκυρο δυαδικό δένδρο. Στο δεύτερο βήμα, ***'ανεβάζουμε' τη νέα τιμή στην κατάλληλη θέση*** εφόσον αυτό είναι απαραίτητο. Με αυτό τον τρόπο, το δένδρο συνεχίζει να αποτελεί ένα έγκυρο σωρό.

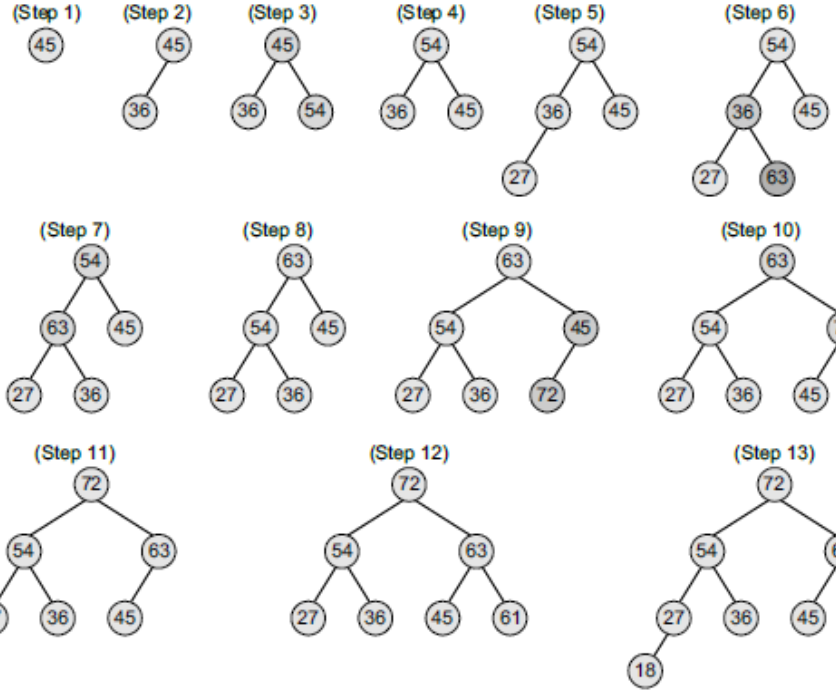
Αλγόριθμος:

```
Step 1: [Add the new value and set its POS]
        SET N = N + 1, POS = N
Step 2: SET HEAP[N] = VAL
Step 3: [Find appropriate location of VAL]
        Repeat Steps 4 and 5 while POS > 1
Step 4:     SET PAR = POS/2
Step 5:     IF HEAP[POS] <= HEAP[PAR],
            then Goto Step 6.
            ELSE
                  SWAP HEAP[POS], HEAP[PAR]
                  POS = PAR
            [END OF IF]
      [END OF LOOP]
Step 6: RETURN
```

Παράδειγμα – Εισαγωγή του 99 σε ένα σωρό.



Παράδειγμα – Δημιουργία και διατήρηση σωρού με την εισαγωγή των: 45, 36, 54, 27, 63, 72, 61, 18



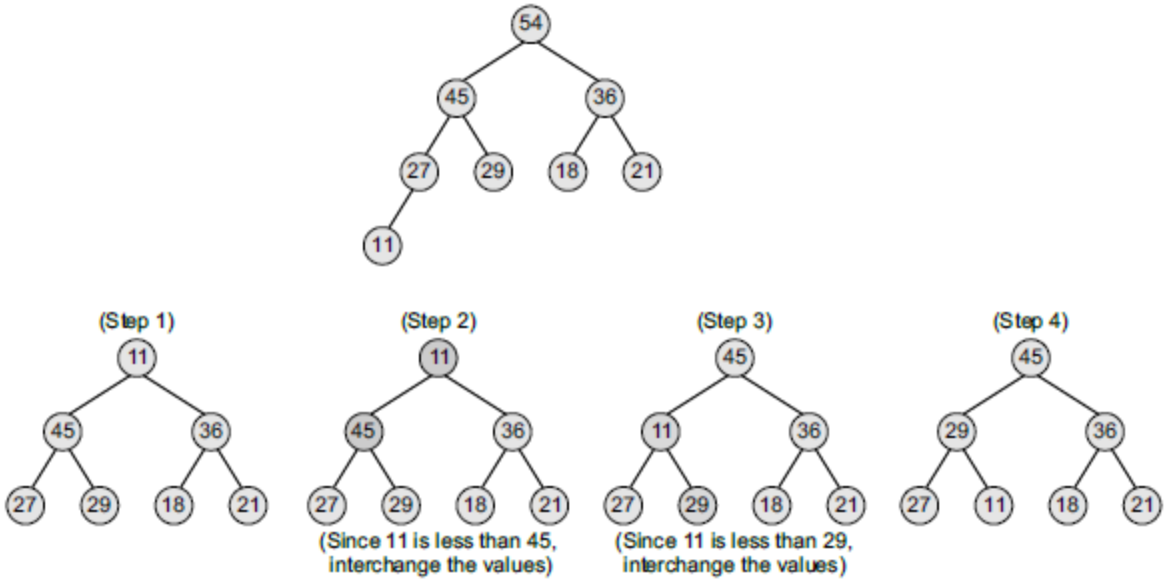
HEAP[1]	HEAP[2]	HEAP[3]	HEAP[4]	HEAP[5]	HEAP[6]	HEAP[7]	HEAP[8]	HEAP[9]	HEAP[10]
72	54	63	27	36	45	61	18		

Διαγραφή στοιχείου. Ένα στοιχείο διαγράφεται πάντα από τη **ρίζα του σωρού** μέσω τριών βημάτων. Αρχικά, ***αντικαθιστούμε τη ρίζα με το τελευταίο στοιχείο του σωρού***. Έπειτα, ***διαγράφουμε τον τελευταίο κόμβο*** του σωρού. Τέλος, '***κατεβάζουμε' το ριζικό κόμβο ώστε να βρεθεί στη σωστή θέση***. Κάθε φορά, αντικαθιστούμε ένα πατρικό κόμβο με ένα παιδί.

Αλγόριθμος

```
Step 1: [Remove the last node from the heap]
        SET LAST = HEAP[N], SET N = N - 1
Step 2: [Initialization]
        SET PTR = 1, LEFT = 2, RIGHT = 3
Step 3: SET HEAP[PTR] = LAST
Step 4: Repeat Steps 5 to 7 while LEFT <= N
Step 5: IF HEAP[PTR] >= HEAP[LEFT] AND
        HEAP[PTR] >= HEAP[RIGHT]
              Go to Step 8
        [END OF IF]
Step 6: IF HEAP[RIGHT] <= HEAP[LEFT]
              SWAP HEAP[PTR], HEAP[LEFT]
              SET PTR = LEFT
        ELSE
              SWAP HEAP[PTR], HEAP[RIGHT]
              SET PTR = RIGHT
        [END OF IF]
Step 7: SET LEFT = 2 * PTR and RIGHT = LEFT + 1
        [END OF LOOP]
Step 8: RETURN
```

Παράδειγμα – Διαγραφή του 54



Code 4

```
#include <stdio.h>

int array[100], n;

display()
{
   int i;
   if (n == 0)
   {
      printf("Heap is empty \n");
      return 0;
   }
   for (i = 0; i < n; i++)
      printf("%d ", array[i]);
```

```
   printf("\n");
}/*End of display()*/


insert(int num, int locatio
{
   int parentnode;
   while (location > 0)
   {
      parentnode =(location - 1)/2;
      if (num <= array[parentnode])
      {
        array[location] = num;
         return 0;
      }
      array[location] = array[parentnode];
      location = parentnode;
   }/*End of while*/
   array[0] = num; /*assign number to the root node */
}/*End of insert()*/


deleted(int num)
{
   int left, right, i, temp, parentnode;
   for (i = 0; i < num; i++) {
      if (num == array[i])
         break;
   }
   if (num != array[i])
   {
      printf("%d not found in heap list\n", num);
      return 0;
   }
   array[i] = array[n - 1];
   n = n - 1;
   parentnode =(i - 1) / 2; /*find parentnode of node i */
   if (array[i] > array[parentnode])
   {
      insert(array[i], i);
      return 0;
   }
   left = 2 * i + 1; /*left child of i*/
   right = 2 * i + 2; /* right child of i*/
   while (right < n)
   {
      if (array[i] >= array[left] && array[i] >= array[right])
        return 0;
      if (array[right] <= array[left])
      {
```

```
            temp = array[i];
            array[i] = array[left];
            array[left] = temp;
            i = left;
        }
        else
        {
            temp = array[i];
            array[i] = array[right];
            array[right] = temp;
            i = right;
        }
        left = 2 * i + 1;
        right = 2 * i + 2;
    }/*End of while*/
    if (left == n - 1 && array[i])    {
        temp = array[i];
        array[i] = array[left];
        array[left] = temp;
    }
}

main()
{
    int choice, num;
    n = 0;/*Represents number of nodes in the heap*/
    while(1)
    {
        printf("1.Insert the element \n");
        printf("2.Delete the element \n");
        printf("3.Display all elements \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);
        switch(choice)
        {
        case 1:
            printf("Enter the element to be inserted to the heap : ");
            scanf("%d", &num);
            insert(num, n);
            n = n + 1;
            break;
        case 2:
            printf("Enter the elements to be deleted from the heap: ");
            scanf("%d", &num);
            deleted(num);
            break;
        case 3:
            display();
            break;
```

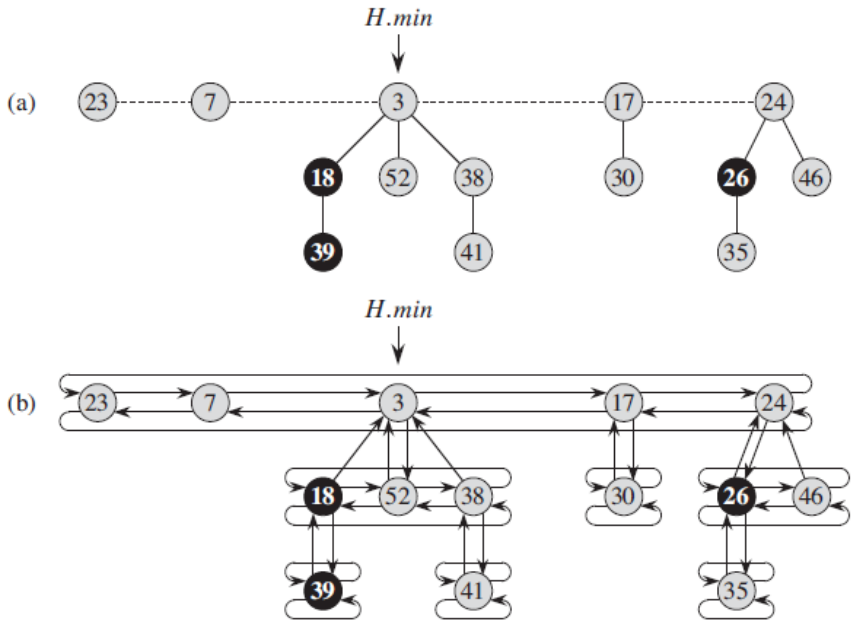
```
        case 4:
           return 0;
        default:
           printf("Invalid choice \n");
                    }/*End  of switch */

          }/*End of while */

}/*End of main()*/
```

Σωροί Fibonacci (Fibonacci Heaps)

Ένας σωρός Fibonacci είναι ένα σύνολο από δένδρα. Το κάθε δένδρο είναι ένας σωρός. Κάθε κόμβος περιλαμβάνει δείκτες προς: (α) **τον πατρικό κόμβο**, (β) **προς κάθε παιδί**. Επίσης, το κάθε δένδρο υπακούει στην ιδιότητα του **min-heap**: το κλειδί ενός κόμβου είναι μεγαλύτερο ή ίσο από το κλειδί του πατρικού κόμβου. Τα παιδιά ενός κόμβου συνδέονται με μια κυκλική διπλά συνδεδεμένη λίστα που αποκαλείται η λίστα παιδιών του κόμβου. Κάθε παιδί στη λίστα περιλαμβάνει δείκτες προς τους κόμβους-αδέρφια.



Operation	Description	Time complexity	
		Binary	Fibonacci
Create Heap	Creates an empty heap	O(n)	O(n)
Find Min	Finds the node with minimum value	O(1)	O(n)
Delete Min	Deletes the node with minimum value	O(log n)	O(log n)
Insert	Inserts a new node in the heap	O(log n)	O(1)
Decrease Value	Decreases the value of a node	O(log n)	O(1)
Union	Unites two heaps into one	O(n)	O(1)

Code 5

Ανατρέξτε στο http://www.sanfoundry.com/cpp-program-implement-fibonacci-heap/

Code 4 (Dijkstra with Binary Heap)

```
/* dijkstra.c57.  C57 code to run Dijkstra's algorithm on a small
   directed graph.  Uses a min-heap as the priority queue. */

#include <stdio.h>

/* Return the index of the parent of node i. */
int parent(int i) {
  return i / 2;
}

/* Return the index of the left child of node i. */
int left(int i) {
  return 2 * i;
}

/* Return the index of the right child of node i. */
int right(int i) {
  return 2 * i + 1;
}

/* Exchange nodes i and j, updating their keys, handles, and
   heap_index values. */
void exchange(double key[], int handle[], int heap_index[], int i, int j) {
  double key_temp;
  int handle_temp;

  /* Exchange the keys in nodes i and j. */
  key_temp = key[i];
  key[i] = key[j];
  key[j] = key_temp;

  /* Exchange the handles in nodes i and j. */
  handle_temp = handle[i];
  handle[i] = handle[j];
  handle[j] = handle_temp;

  /* Update the heap_index values. */
  heap_index[handle[i]] = i;
  heap_index[handle[j]] = j;
}

/* Make the min-heap rooted at node i obey the min-heap property.
   Assumes that the subtrees rooted at i's left and right children
   already obey the min-heap property. */
void heapify(double key[], int handle[], int heap_index[], int i, int size) {
  int l = left(i);
  int r = right(i);
  int smallest;
```

```
  /* Is the left child smaller than node i? */
  if (l <= size && key[l] < key[i])
    smallest = l;
  else
    smallest = i;

  /* Is the right child smaller than node i and i's left child? */
  if (r <= size && key[r] < key[smallest])
    smallest = r;

  /* If the min-heap property is violated between node i and one of
     its children, fix it. */
  if (smallest != i) {
    exchange(key, handle, heap_index, i, smallest);
    heapify(key, handle, heap_index, smallest, size);
  }
}

/* Take an array that does not necessarily obey the min-heap property,
   and rearrange it so that it does. */
void build_heap(double key[], int handle[], int heap_index[], int size) {
  int i;
  for (i = size/2; i >= 1; --i)
    heapify(key, handle, heap_index, i, size);
}

/* Extract the node with the minimum key, at index 1, from the
   min-heap. */
void extract_min(double key[], int handle[], int heap_index[], int size) {
  exchange(key, handle, heap_index, 1, size);
  heapify(key, handle, heap_index, 1, size-1);
}

/* Bubble the key in node i up toward the root until the min-heap
   property is restored. */
void decrease_key(double key[], int handle[], int heap_index[],
                  int i, int size, double new_key) {
  key[i] = new_key;
  while (i > 1 && key[parent(i)] > key[i]) {
    exchange(key, handle, heap_index, i, parent(i));
    i = parent(i);
  }
}

/* Insert a new node into the min-heap.  Assumes that an array element
   has already been allocated. */
void insert(double key[], int handle[], int heap_index[],
            int vertex, int size, double new_key) {
  key[++size] = 1000000000.0;    /* will be fixed later */
```

```
  handle[size] = vertex;
  heap_index[vertex] = size;
  decrease_key(key, handle, heap_index, size, size, new_key); /* here's later */
}

/* Relax edge (u, v) with weight w. */
void relax(int u, int v, double w,
           double key[], int handle[], int heap_index[], int size, int pi[]) {
  if (key[heap_index[v]] > key[heap_index[u]] + w) {
    decrease_key(key, handle, heap_index, heap_index[v], size, key[heap_index[u]] + w);
    pi[v] = u;
  }
}

/* Initialize a single-source shortest-paths computation. */
void initialize_single_source(double key[], int handle[], int heap_index[],
                              int pi[], int s, int n) {
  int i;
  for (i = 1; i <= n; ++i) {
    key[i] = 1000000000.0;
    handle[i] = i;
    heap_index[i] = i;
    pi[i] = 0;
  }

  key[s] = 0.0;
  build_heap(key, handle, heap_index, n);
}

/* Run Dijkstra's algorithm from vertex s.  Fills in arrays d and pi. */
void dijkstra(int first[], int node[], int next[], double w[], double d[],
              int pi[], int s, int n, int handle[], int heap_index[]) {
  int size = n;
  int u, v, i;

  initialize_single_source(d, handle, heap_index, pi, s, n);
  while (size > 0) {
    u = handle[1];
    extract_min(d, handle, heap_index, size);
    --size;
    i = first[u];
    while (i > 0) {
      v = node[i];
      relax(u, v, w[i], d, handle, heap_index, size, pi);
      i = next[i];
    }
  }
}

/* Set up a directed graph and run Dijkstra's algorithm on it.  Print
```

```
 out the result. */
int main(void) {
 int first[6], node[11], next[11], pi[6], handle[6], heap_index[6];
 double w[11], d[6];
 int s;
 int i;

 /* The graph contains the following directed edges with weights:
   (1, 2), weight 10
   (1, 4), weight 5
   (2, 3), weight 1
   (2, 4), weight 2
   (3, 5), weight 4
   (4, 2), weight 3
   (4, 3), weight 9
   (4, 5), weight 2
   (5, 1), weight 7
   (5, 3), weight 6
 */

 first[1] = 1;
 first[2] = 3;
 first[3] = 5;
 first[4] = 6;
 first[5] = 9;

 node[1] = 2;
 node[2] = 4;
 node[3] = 3;
 node[4] = 4;
 node[5] = 5;
 node[6] = 2;
 node[7] = 3;
 node[8] = 5;
 node[9] = 1;
 node[10] = 3;

 w[1] = 10.0;
 w[2] = 5.0;
 w[3] = 1.0;
 w[4] = 2.0;
 w[5] = 4.0;
 w[6] = 3.0;
 w[7] = 9.0;
 w[8] = 2.0;
 w[9] = 7.0;
 w[10] = 6.0;

 next[1] = 2;
 next[2] = 0;
```

```
  next[3] = 4;
  next[4] = 0;
  next[5] = 0;
  next[6] = 7;
  next[7] = 8;
  next[8] = 0;
  next[9] = 10;
  next[10] = 0;

  printf("Enter source node: ");
  scanf("%d", &s);

  dijkstra(first, node, next, w, d, pi, s, 5, handle, heap_index);

  for (i = 1; i <= 5; ++i) {
    printf("%d: %f  %d\n", i, d[heap_index[i]], pi[i]);
  }

  return 0;
}

/* Correct outputs:

   Enter source node: 1
   1: 0.000000  0
   2: 8.000000  4
   3: 9.000000  2
   4: 5.000000  1
   5: 7.000000  4

   Enter source node: 2
   1: 11.000000  5
   2: 0.000000  0
   3: 1.000000  2
   4: 2.000000  2
   5: 4.000000  4

   Enter source node: 3
   1: 11.000000  5
   2: 19.000000  4
   3: 0.000000  0
   4: 16.000000  1
   5: 4.000000  3

   Enter source node: 4
   1: 9.000000  5
   2: 3.000000  4
   3: 4.000000  2
   4: 0.000000  0
   5: 2.000000  4
```

```
  Enter source node: 5
  1: 7.000000  5
  2: 15.000000  4
  3: 6.000000  5
  4: 12.000000  1
  5: 0.000000  0
*/
```