Κατ'οίκον Εργασία 4 – Σκελετοί Λύσεων

Άσκηση 1

Αναδρομική διαδικασία

Η αναδρομική διαδικασία `RecIsheap` παίρνει ως παραμέτρους τον πίνακα, το μέγεθός του καθώς και το στοιχείο το οποίο θα τύχει επεξεργασίας. Σε μια αναδρομική κλήση εξετάζουμε τα παιδιά ενός συγκεκριμένου κόμβου και καλούμε αναδρομικά για τον επόμενο κόμβο. Για την εξεύρεση της λύσης η συνάρτηση καλείται ως `RecIsheap(array, size, 1)` για να ξεκινήσουμε την επεξεργασία από το πρώτο στοιχείο.

```
int RecIsheap (int array, int size, int element){

 if (element > float(size/2))    //Μόνο οι size/2 κόμβοι έχουν παιδιά
      return 1;                  // ⇒ Ο κόμβος δεν έχει παιδιά

 first_child = 2*element;        // Το πρώτο παιδί του κόμβου
 second_child = 2*element+1;     // Το δεύτερο παιδί του κόμβου

 if (second_child <= size) // Έχει δύο παιδιά
      if (array[element] > array[first_child] || array[element]
                                          > array[second_child])
            return 0
      else
            return 1 && RecIsheap(array, size, element++)
 else
      if (array[element] > array[first_child])
            return 0
      else
            return 1
}
```

Μη αναδρομική διαδικασία

Για κάθε κόμβο του σωρού εξετάζουμε τα παιδιά του. Αν κάποιο από αυτά είναι μικρότερο από αυτό, τότε παραβιάζεται η συνθήκη και επιστρέφουμε 0. Αν επεξεργαστούμε τον πίνακα χωρίς παραβίαση της συνθήκης, τότε συμπεραίνουμε ότι ο πίνακας είναι σωρός και επιστρέφουμε 1.

```
int   IsHeap (int array, int size){

 for(i=1; i<float(size/2); i++)
   {
      if (2*i+1 <= A->size) // Υπάρχουν δύο παιδιά
            if (A->array[i] > A->array[2*i] || A->array[i]
                                       > A->array[2*i+1])
                  return 0
      else
            if (A->array[i] > A->array[2*i])
                  return 0
   }
 return 1
}
```

Άσκηση 2

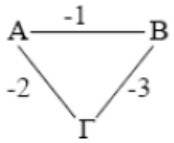
(α) Έστω γράφος G=(V,E) και E' υποσύνολο του E που συνδέει όλες τις κορυφές του V και έχει το ελάχιστο βάρος. Θέλουμε να δείξουμε ότι, αν τα βάρη των ακμών του E είναι θετικά, οι ακμές E' σχηματίζουν ένα δένδρο. Ας υποθέσουμε, για να φθάσουμε σε αντίφαση ότι το E' δεν είναι δένδρο, δηλαδή περιέχει κάποιο κύκλο.

Έστω (u,v) κάποια ακμή του κύκλου. Παρατηρούμε ότι εκτός από την ακμή (u,v) υπάρχει και άλλο μονοπάτι που ενώνει τις δύο κορυφές u και v, δηλαδή το υπόλοιπο του κύκλου που περιέχει την ακμή, έστω path.

Θεωρούμε το σύνολο E' – {(u,v)}. Το σύνολο αυτό έχει προφανώς μικρότερο βάρος από το βάρος του E. Συγκεκριμένα, αν γράψουμε W(E) και W(E') για το άθροισμα των βαρών των συνόλων E και E' αντίστοιχα, παρατηρούμε ότι W(E') = W(E) – w(u,v) και αφού το βάρος όλων των ακμών είναι θετικό w(u,v) > 0 και W(E') < W(E').

Επίσης το σύνολο E' – {(u,v)} συνδέει όλους τους κόμβους του γράφου: Για οποιοδήποτε ζεύγος κορυφών a και b αφού συνδέονται από τις ακμές E' υπάρχει μονοπάτι που τις συνδέει. Αν το μονοπάτι αυτό δεν περιέχει την ακμή οποιοδήποτε (u,v) τότε προφανώς οι δύο κορυφές συνδέονται και μέσω τον ακμών E' – {(u,v)}, διαφορετικά, αν περιέχει τη συγκεκριμένη ακμή, τότε η ακμή αυτή μπορεί να αντικατασταθεί από το μονοπάτι path. Το καινούριο μονοπάτι εξακολουθεί να συνδέει τις δύο κορυφές και επιπλέον περιέχει ακμές μόνο από το σύνολο E' – {(u,v)}. Κατά συνέπεια το E' – {(u,v)} είναι ένα σύνολο ακμών που συνδέει όλους τους κόμβους και έχει μικρότερο βάρος από το E' Το πιο πάνω μας οδηγεί σε αντίφαση στην υπόθεση ότι το σύνολο E' είναι σύνολο ακμών που συνδέει όλες τις κορυφές και έχει το ελάχιστο βάρος.

(β) Το σύνολο ακμών που συνδέει όλες τις κορυφές με το ελάχιστο βάρος είναι το {(A,B), (B,Γ), (Γ,A)} και προφανώς δεν είναι δένδρο.



(γ) Η ιδέα αυτή μας οδηγεί στον πιο κάτω αλγόριθμο. Ξεκίνα με το κενό σύνολο ακμών T και κατασκεύασε ένα γεννητορικό δένδρο ως εξής.

1. Επέλεξε την ακμή με τον μικρότερο βάρος από αυτές που δεν έχεις επεξεργαστεί.
2. Εφόσον η ακμή αυτή δεν δημιουργεί κύκλο στο σύνολο T τότε πρόσθεσέ την σε αυτό.

Διαφορετικά επανέλαβε από το βήμα 1 και συνέχισε μέχρι να επιλέξεις |V| - 1 ακμές.

```
Spanning_Tree(graph G){
        buildHeap(H); (που περιέχει όλες τις ακμές του G)
        i = 0;
        T = ∅ ;
        while (i < |V|-1){
                (u,v) = DeleteMin(H);
                if T ∪ {(u,v)} does not contain a cycle
                        T = T ∪ {(u,v)};
                i++;
}
```

Άσκηση 3

(1) Το πρόβλημα μπορεί να μοντελοποιηθεί ως πρόβλημα γράφων ως εξής: Θεωρήστε τον γράφο με βάρη G=(V,E) όπου $V = V_1 \cup V_2$ και
V_1 είναι το σύνολο των εργασιών τύπου OR
V_2 είναι το σύνολο των εργασιών τύπου AND
E={(*i, j*) | αν η εργασία i αποτελεί προαπαιτούμενο της εργασίας *j* }

Η δομή που θα χρησιμοποιήσουμε για την υλοποίηση του γράφου είναι:

```
struct node {
        int vertex;
        struct node *next ;
}
```

```
struct graph {
        struct node head[max];
        int type[max];
        int size;
}
```

Στον πίνακα type γράφουμε τον τύπο των διεργασιών σημειώνοντας type[i] = 0 αν η εργασία είναι τύπου AND και type[i] = 1 αν η εργασία είναι τύπου OR.

(2) Η διαδικασία είναι μια παραλλαγή του αλγόριθμου τοπολογικής ταξινόμησης η οποία διακρίνει ανάμεσα στις διεργασίες τύπου OR και AND, και, σε περίπτωση που εκτελεστεί κάποιο προαπαιτούμενο μιας διεργασία τύπου OR, τότε την καθιστά έτοιμη για εκτέλεση τοποθετώντας την στην ουρά.

```
TopSort(graph G){
  MakeEmpty(Q);

  int I[G->size];

  for (u = 0; u < G->size; u++)
        I[u] = 0;

  for (u = 0; u < G->size; u++)
        p = G->head[u];
        while (p != NULL)
              I[p->vertex]++;

  for (u = 0; u < G->size; u++)
        if (I[u]==0)
              Enqueue(u, Q);

  i = 0;
  while (! IsEmpty(Q)) {
        u = Dequeue(Q);
        p = G->head[u];
        while (p != NULL)
              I[p->vertex]--;
              if (G->type[p->vertex] == 1 OR I[p->vertex] == 0)
                    Enqueue(p->vertex, Q)
  }

  if (i < |V| -1)
        report “The tasks contain a dependency cycle”;
}
```

(γ) Μετά από εκτύπωση των τεσσάρων πρώτων διεργασιών, παρατηρούμε ότι η εκτέλεση της e_5 είναι αδύνατη λόγω ύπαρξης κύκλου ανάμεσα στις 3 τελευταίες διεργασίες.

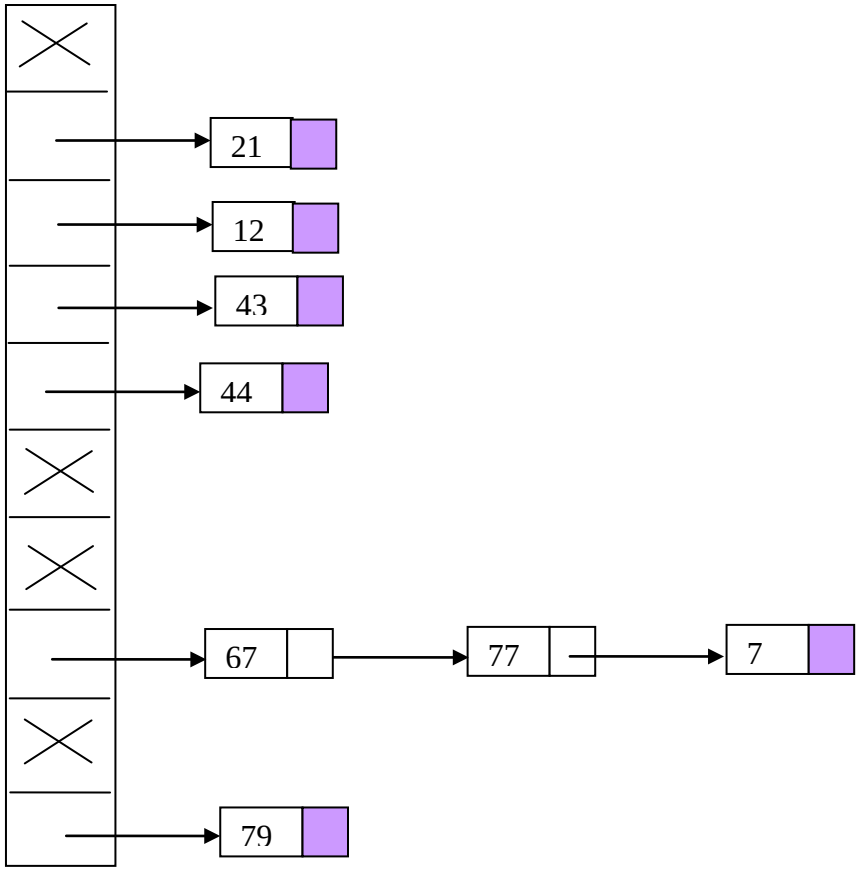
Άσκηση 4

Το πρόβλημα μπορεί να λυθεί με παραλλαγή του αλγόριθμου του Dijkstra. Αρχικά παραθέτουμε τις σχετικές δομές.

```
struct AVLNode{                          struct route{
     string name;                             string street;
     int pointnum;                            int distance;
     struct node *leftchild;                  char direction;
     struct node *rightchild;            }
}

string IdToLocation[N]        // Τα ονόματα των σημείων
struct AVLNode *LocationtoId // δένδρο με ζεύγη σημείο-αριθμός
struct route Connect[N][N];  // Οι συνδέσεις ανάμεσα στα σημεία

FindShortestPath(string A, string B){
  heap Q;
  for all v∈V
      d[v]=∞; P[v]= '-';
      Insert(Q, (v,d[v]));

  x = FindNum(A, LocationtoId);  //Διάσχιση του AVL δέντρου και
  y = FindNum(B, LocationtoId);  //επιστροφή του αριθμού της
                                 // τοποθεσίας βάση του ονόματος
  d[x]=0; DecreaseKey(Q,x,0);

  while (!IsEmpty(Q)){
      u = DeleteMin(Q);
      if (u == y)
           break;

      for(i = 1; i <= N; i++)
           if (Connect[u,i].direction != '#')
                if d[i] > d[u] + Connect[u,i].direction
                     d[i] = d[u] + Connect[u,i].direction;
                     DecreaseKey(Q, i, d[i]);
                     P[i] = u;
  }

MakeEmptyStack(S);
while (y != x)          // Αποθήκευση του βραχύτερου μονοπατιού
     Push(S, y)         // ανάποδα σε μια στοίβα
     y = P[y];
  while(!IsEmpty(S))  // εκτύπωση του μονοπατιού
     p = Pop(S);
     a = Connect[x,p];
     print "Από το σημείο IdtoLocation[x] οδήγησε προς την
κατεύθυνση a.direction
                πάνω στον δρόμο a.street";
     x = p;
}
```

Άσκηση 5

(α)



(β)

0	1	2	3	4	5	6	7	8	9
	21	**12**	**43**		**79**		**7**		**77**

Η εισαγωγή του 67 δεν είναι εφικτή.

(γ)

0	1	2	3	4	5	6	7	8	9
	21	**12**	**43**		**79**		**7**		**77**

Η εισαγωγή του 67 δεν είναι εφικτή.

(δ)

0	1	2	3	4	5	6	7	8	9
67	**21**	**12**	**43**	**77**			**7**		**79**

Η λύση που αναφέρεται στον επανακατακερματισμό παραλείπεται.