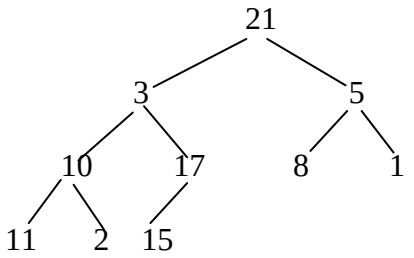
Κατ'οίκον Εργασία 4 – Σκελετοί Λύσεων

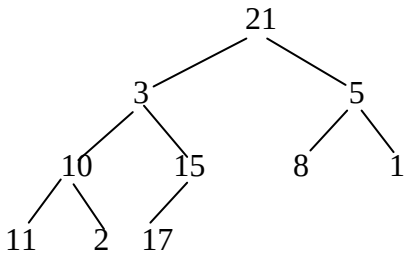
Άσκηση 1

α) Εφαρμογή της BuildHeap στον πίνακα [-, 21, 3, 5, 10, 17, 8, 1, 11, 2, 15] έχει τις εξής ενδιάμεσες καταστάσεις.

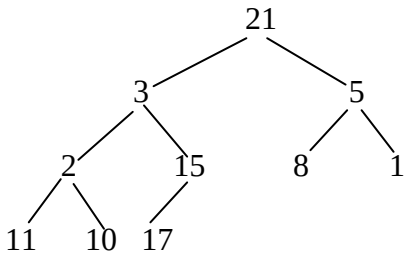
Αρχική Κατάσταση:



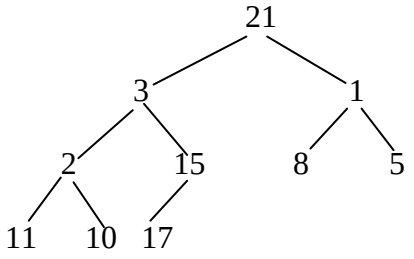
Μετά από εφαρμογή της PercoladeDown του στοιχείου στη θέση 5:



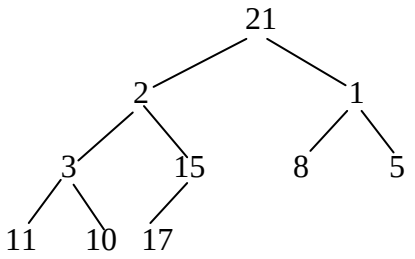
Μετά από εφαρμογή της PercoladeDown του στοιχείου στη θέση 4:



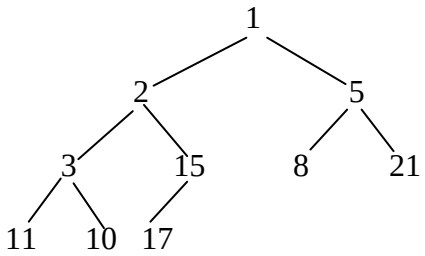
Μετά από εφαρμογή της PercoladeDown του στοιχείου στη θέση 3:



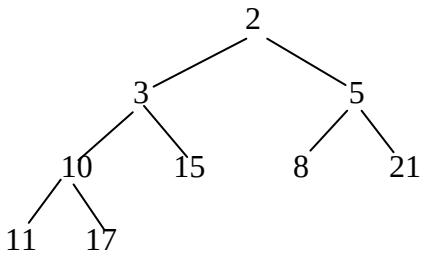
Μετά από εφαρμογή της PercoladeDown του στοιχείου στη θέση 2:



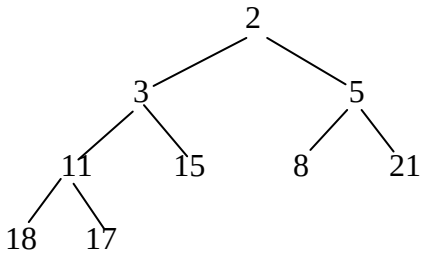
Μετά από εφαρμογή της PercoladeDown του στοιχείου στη θέση 1:



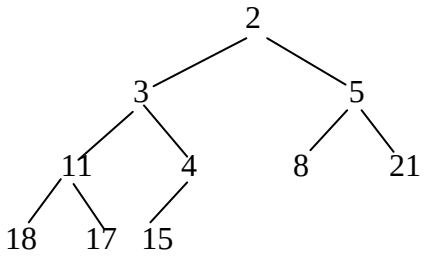
β) Μετά από εφαρμογή της DeleteMin:



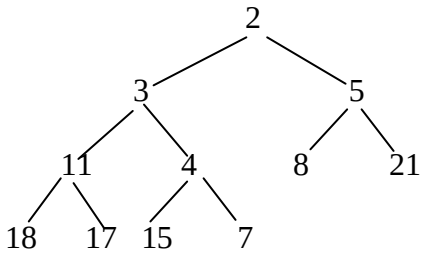
γ) Μετά από εφαρμογή της IncreaseKey (4, 8):



δ) Μετά από εφαρμογή της Insert (4):



ε) Μετά από εφαρμογή της Insert (7):



Άσκηση 2

Χρησιμοποιούμε τις πιο κάτω δομές

```
typedef struct fourheap{
     int size;
     quadruple data[maxsize];
} 4heap;
```

```
typedef struct Quadruple {
     int key1;
     int key2;
     int key3;
     int key4;
} quadruple ;
```

και υποθέτουμε πως μία σωρός έχει τύπο fourheap και περιέχει τετράδες τύπου quadruple.

Οι πράξεις υλοποιούνται παρόμοια με τις πράξεις σε σωρό. Η βασική διαφορά αφορά στο ότι, σε αδελφούς κόμβους, τα κλειδιά του ενός πρέπει να είναι μικρότερα από τα κλειδιά του άλλου.

(α) Η διαδικασία εισαγωγής ξεκινά τοποθετώντας τα 4 νέα στοιχεία στην πρώτη διαθέσιμη θέση του πίνακα, έστω h. Συνδυάζουμε τα στοιχεία της θέσης h με τον πατέρα της, έστω f, έτσι ώστε η θέση h να πάρει τα μεγαλύτερα στοιχεία και ο πατέρας τα μικρότερα. Συνδυάζουμε τα στοιχεία της θέσης h με τον αδελφό της, αν υπάρχει αδελφός, έτσι ώστε ο αδελφός που είχε τα μεγαλύτερα στοιχεία να πάρει τα 4 μεγαλύτερα στοιχεία των δύο κόμβων και ο άλλος τα μικρότερα. Συνεχίσουμε την διαδικασία για h = f μέχρις ότου είτε φθάσουμε στη ρίζα είτε φθάσουμε σε κάποιο σημείο όπου ο κόμβος h έχει μεγαλύτερα στοιχεία από τον πατέρα του.

Σε ψευδοκώδικα έχει ως εξής:

```
Insert (4heap E, quadruple x){
  check E->size < maxsize;
  int h = E->size + 1;
  E->size++;
  E.data[h] = x;
  while(h>1 && υπάρχει κλειδί στον πατέρα, (contents[⌊h/2⌋]),
                μεγαλύτερο από κάποιο κλειδί του contents[h]) {
     data[h] = τα 4 μεγαλύτερα κλειδιά των θέσεων
                    contents[⌊h/2⌋] και contents[h];
     data[⌊h/2⌋] τα 4 μικρότερα κλειδιά
                    των θέσεων contents[⌊h/2⌋] και contents[h];
     if (h = 2 · ⌊h/2⌋)
          brother = h+1;
     else
          brother = h-1;
```

```
      if (brother <= E->size)
            if (maximum key of data[brother] >
                                          maximum key of data[h]) {
                  data[brother] = τα 4 μεγαλύτερα κλειδιά των
                              θέσεων data[brother] και data[h];
                  data[h]= τα 4 μικρότερα κλειδιά των θέσεων
                              data[brother] και data[h];
            }
            else
                  data[h] = τα 4 μεγαλύτερα κλειδιά των θέσεων
                              data [brother] και data[h];
                  data[brother]= τα 4 μικρότερα κλειδιά των θέσεων
                              data[brother] και data[h];
            }
  }
}
```

(β) Η διαδικασία εξαγωγής ελαχίστου ξεκινά εξάγοντας το στοιχείο της πρώτης θέσης και τοποθετώντας εκεί τα κλειδιά της τελευταίας θέσης του πίνακα. Ονομάζουμε την πρώτη θέση h. Ελέγχουμε τα παιδιά της θέσης h και συνδυάζουμε τα στοιχεία της με αυτά του παιδιού της με τα μικρότερα κλειδιά, έστω s, έτσι ώστε η θέση h να αποκτήσει τα 4 μικρότερα κλειδιά και η θέση s τα μεγαλύτερα. Στη συνέχεια συνδυάζουμε τα κλειδιά της θέσης s με αυτά του αδελφού της (αν υπάρχει), έστω, q, έτσι ώστε η θέση s να αποκτήσει τα 4 μεγαλύτερα στοιχεία και η θέση q τα 4 μικρότερα. Συνεχίσουμε την διαδικασία για h = s μέχρις ότου είτε φθάσουμε σε φύλλο είτε φθάσουμε σε κάποιο σημείο όπου ο κόμβος h έχει μικρότερα στοιχεία από το μικρότερο παιδί του. Σε ψευδοκώδικα έχει ως εξής:

```
quadrable DeleteMin (4heap E){
  if IsEmpty(E)
     report error; return;

  min = E->data[1];
  E->data[1] = E->data[E->size];
  E->size--;
  h=1;

  while(h*2 <= size){
     minchild = h*2;
     if (minchild == size) {
           maxchild = 0;
     else
           if (max E->data[minchild+1]< min E->[minchild]) {
                 minchild++;
                 maxchild = minchild– 1;
           }
           else
                 maxchild = minchild+1;

     if (min E->data[minchild] < max E->[h]) {
           data[h] = τα 4 μικρότερα κλειδιά των θέσεων
                       contents[minchild] και contents[h];
           data[minchild] = τα 4 μεγαλύτερα κλειδιά των θέσεων
                       contents[minchild] και contents[h];
```

```
            if (maxchild != 0) {
                  data[minchild] = τα 4 μικρότερα κλειδιά των
                        θέσεων data[minchild] και data[maxchild];
                  data[maxchild] = τα 4 μεγαλύτερα κλειδιά των
                        θέσεων data[minchild] και data[maxchild];
            }
      }
      else
            break;
      h = minchild;
  }
  return min;
}
```

Άσκηση 3

Το πρόβλημα μπορεί να μεταφραστεί σε πρόβλημα εύρεσης του μονοπατιού με το ελάχιστο κόστος σε γράφο όπου κάθε ακμή έχει βάρος 1.

(α) Θεωρήστε τον γράφο με βάρη $G = (V, E)$ όπου

$V = \{ P_1, P_2, ..., P_n \}$, δηλαδή υπάρχει μία κορυφή για κάθε ιστοσελίδα

$E = \{(P_i, P_j) \mid$ αν υπάρχει σύνδεσμος μεταξύ των ιστοσελίδων P_i και $P_j \}$

Υποθέτουμε ότι οι γράφοι υλοποιούνται με βάση την πιο κάτω δομή:

```
struct graph{
   int matrix[max][max];
   int size;
}
```

(β) Το ελάχιστο μονοπάτι μπορεί να υπολογιστεί με μια κατά πλάτος διερεύνηση στο γράφο. Επομένως, εφαρμόζουμε κατά πλάτος διερεύνηση από τον κόμβο εκκίνησης φυλάγοντας για κάθε κόμβο τους κόμβους που έχουμε περάσει στο μονοπάτι από τον κόμβο εκκίνησης και μόλις συναντήσουμε τον κόμβο προ φισμο ύ για πρώτη φορά επιστρέφουμε το μονοπάτι που έχουμε υπολογίσει:

```
Reachable(vertex A, vertex B, graph G){
  int Visited[n];
  int Previous[n];
  Q=MakeEmptyQueue();
  for ( i=0; i<= G->size-1; i++ ) // for each w in G
     Visited[i]=False;
  Visited[A]= True;
  Enqueue(A,Q);

  while (!IsEmpty(Q)){
     w = Dequeue(Q);
     if (w == B)
           print w;
           while(w != A)
                 w = Previous[w];
                 print w;
```

```
            return;
      for ( u = 0; u <= G->size-1; u++ )
            if (G->matrix[w][u] > 0) // for each u adjacent to w
                  if (Visited[u]=False) {
                        Previous[u] = w;
                        Visited[u]=True;
                        Enqueue(u,Q);
                  }
  }
}
```

Χρόνος Εκτέλεσης: $O(|V|^2)$, όπου V είναι ο αριθμός κορυφών του γράφου.

Για να έχουμε αποδοτική συνάρτηση MasterPage, πρέπει να τροποποιήσουμε τη συνάρτηση BFS, έτσι ώστε να υπολογίζει τον αριθμό των διαφορετικών κόμβων που είναι προσβάσιμοι από την σελίδα εκκίνησης. Αν ο αριθμός των προσβάσιμων κόμβων είναι ίσος με τον αριθμό των κόμβων του γράφου, τότε η σελίδα εκκίνησης είναι MasterPage.

```
int Reachable2(vertex A, graph G){
  Q=MakeEmptyQueue();
  int reachable = 1; // Για τον κόμβο Α
  for ( i=0; i<= G->size-1; i++ ) // for each w in G
     Visited[i]=False;
  Visited[A]= True;
  Enqueue(A,Q);

  while (!IsEmpty(Q)){
     w = Dequeue(Q);
     for ( u = 0; u <= G->size-1; u++ )
           if (G->matrix[w][u] > 0) // for each u adjacent to w
                 if (Visited[u]=False) {
                       Visited[u]=True;
                       Enqueue(u,Q);
                       reachable++;
                 }
  }
  if (G->size == reachable)
     return 1;
  else
     return 0;
}
```

Χρόνος Εκτέλεσης: $O(|V|^2)$, όπου V είναι ο αριθμός κορυφών του γράφου.

```
MasterPage(graph G) {
  for ( w=0; w<= G->size-1; i++ ) { // for each w in G
     if (Reachable2(w, G)) { // αν όλοι είναι προσβάσιμοι
           print w;
}
```

Χρόνος Εκτέλεσης: $O(|V|^3)$, όπου V είναι ο αριθμός κορυφών του γράφου.