Κατ'οίκον Εργασία 2 – Σκελετοί Λύσεων

Άσκηση 1

Ξεκινούμε με τον αριθμό μας, *n*, και αρχίζουμε να τον διαιρούμε με ακέραιους ξεκινώντας με το 2 και προχωρώντας στο 3, 4, 5, …. Όταν εντοπίσουμε πως ένας αριθμός i είναι παράγοντας του *n* (δηλαδή *n* mod *i* == 0) τον τοποθετούμε σε μία στοίβα και θέτουμε *n* = *n*/*i*. Όταν ο *n* γίνει 1 τότε σταματούμε και τυπώνουμε τα στοιχεία της στοίβας. Υποθέτουμε την ύπαρξη υλοποίησης στοίβας με πράξεις MakeEmpty, IsEmpty, Push, Pop και Top.

```
Paragontopoihsh(int n) {
  i=2;
  MakeEmpty(S);

  while (n != 1){
      if (n mod i == 0) {
            Push (i, S);
            n = n div i
      }
      else
            i++;
   }

  while (!IsEmpty(S)) {
      print "Top(S)";
      Pop(S);
  }
}
```

Ο χρόνος εκτέλεσης του αλγορίθμου είναι της τάξης O(*n*), αφού στη χείριστη περίπτωση που το *n* είναι πρώτος αριθμός, θα χρειαστεί να κοιτάξουμε όλους τους υποψήφιους παράγοντες 2..*n*.

Άσκηση 2

Για την επίλυση της άσκησης αυτής θα χρησιμοποιήσουμε διπλά συνδεδεμένες λίστες υλοποιημένες με δυναμική χορήγηση μνήμης.

Χρησιμοποιούμε τις πιο κάτω δομές:

```
typedef struct Node{
      type data;
      int popularity;
      struct Node *next;
      struct Node *prev;
} node;
```

```
typedef struct DLlist {
      node *head;
} dllist;
```

Οι ζητούμενες διαδικασίες υλοποιούνται ως εξής:

(1) Η διαδικασία που ακολουθείται για την εισαγωγή είναι η εξής: Ξεκινούμε μία διάσχιση της λίστας μέχρι να φτάσουμε στο τέλος της. Εκεί δημιουργούμε ένα καινούριο κόμβο τον οποίο συνδέουμε με τον προηγούμενο κόμβο της λίστας, αν υπάρχει, ενώ, αν δεν υπάρχει τότε ενημερώνουμε τον κόμβο κορυφής της λίστας ως τον νεοδημιούργητο κόμβο.

```
void Insert(dllist *X, type e){
     node *previous = NULL;
     for (p = X->head ; p!= NULL && p->data != e; p=p->next)
               previous = p;
     if (p->data == e)
          report "Element is already in list";
     else {
          node *r = (node *) malloc (sizeof(node));
          r->next = NULL;
          r->prev = previous;
          r->data = e;
          r->popularity = 0;
          if previous == NULL
               X->head = r;
          else
               previous->next = p;
     }
  }
}
```

Χρόνος Εκτέλεσης: Ο(*n*), όπου *n* το μέγεθος της λίστας Χ.

(2) Η διαδικασία πρόσβασης στοιχείου εντοπίζει το στοιχείο, αν υπάρχει, αυξάνει τη δημοτικότητά του κατά 1 και στη συνέχεια αναπροσαρμόζει τη λίστα μέσω της βοηθητικής συνάρτησης Balance(dllist *X, node *p) η οποία μεταφέρει το στοιχείο του κόμβου p μέσα στη λίστα έτσι ώστε να βρεθεί στην κατάλληλη θέση σε σχέση με την αυξημένη του δημοτικότητα.

```
node *Access(dllist *X, type e){
  node *p;
  if (X->head == NULL)
     return NULL;
  else {
     for(p = X->head; p->data!=e && p->next!=NULL; p=p->next)
                    ;
     if (p->data == e)
          p->next->popularity++;
          Balance (&X, &p);
          return p;
     else
          return NULL:
}
```

```
Balance(dllist *X, node *p){
while(p->prev != NULL && p->popularity>p->prev->popularity) {
             //Ανταλλάσσουμε τα στοιχεία data και popularity
             // των δύο κόμβων και το προωθούμε πιο μπροστά
               SwapElements(p,p->prev);
```

```
                p = p->prev;
            }
if p->prev = NULL
     X->head = p;
}
```

Χρόνος Εκτέλεσης: Στην περίπτωση που το e ανήκει στην X: O(*i*), όπου *i* είναι η θέση του στοιχείου e στην ουρά, και O(*n*), όπου *n* είναι το μέγεθος της ουράς X, αν το στοιχείο e δεν ανήκει στην X.

(3) Η διαδικασία εξαγωγής εντοπίζει τον ζητούμενο κόμβο και τον διαγράφει από τη λίστα.

```
void Delete(dllist *X, type e){
  node *p;
  if (X->head == NULL) {
     printf("The list is empty");
  else {
     for (p = X->head; p->data != e && p->next!=NULL;
          p=p->next)
               ;
     if (p->data == e) {
          if (p->next != NULL)
               p->next->prev = p->prev;
          if (p->prev != NULL)
               p->prev->next = p->next;
          else
               X->head = p->next;
          free(p);
     }
  }
}
```

Χρόνος Εκτέλεσης: Στην περίπτωση που το e ανήκει στην X: O(*i*), όπου *i* είναι η θέση του στοιχείου e στην ουρά, και O(*n*), όπου *n* είναι το μέγεθος της ουράς X, αν το στοιχείο e δεν ανήκει στην X.

(4) Η διαδικασία Union ξεκινά αντιγράφοντας τα στοιχεία τη πρώτης λίστας X στην καινούρια λίστα Z και στη συνέχεια παίρνει ένα-ένα τα στοιχεία της λίστας Y και τα εισάγει στη λίστα μέσω της βοηθητικής διαδικασίας Insert2. Η διαδικασία Insert2 λειτουργεί παρόμοια με την διαδικασία Insert με τη διαφορά ότι αν το στοιχείο προς εισαγωγή υπάρχει ήδη στη λίστα τότε αυξάνει τη δημοτικότητά του προσθέτοντας σε αυτή τη δημοτικότητα του καινούριου στοιχείου.

```
void Union(dllist *X, dllist *Y, dllist *Z){
  node *p=X->head, *q=NULL, *r;
  Z->head = NULL;
  while(p != NULL) {
     r =(node *) malloc (sizeof(node));
```

```
     r->data = p->data;
     r->popularity = p->popularity;
     r->prev = q;
     r->next = NULL;
     if (q == NULL)
          Z->head = r;
     else
          q->next = r;
     q = r;
  }
   p = Y->head;
   while (p != NULL)
     Insert2(&Z, &p)
}
```

```
void Insert2(dllist *Z, node *q){
  node *previous = NULL;
  for (p = X->head ; p!= NULL && p->data!=q->data; p=p->next)
          previous = p;
  if (p->data == q->data)
          p->popularity = p->popularity + q->popularity;
          Balance(&Z, &p);
     else {
          node *r = (node *) malloc (sizeof(node));
          r->next = NULL;
          r->prev = previous;
          r->data = q->data;
          r->popularity = q->popularity;
          if previous == NULL
               X->head = r;
          else
               previous->next = p;
     }
  }
}
```

Χρόνος Εκτέλεσης: $O(n \cdot m)$, όπου n και m είναι τα μεγέθη των λιστών X και Y αντίστοιχα.

Άσκηση 3

(1) Ευθύγραμμη διπλά συνδεδεμένη λίστα

Χρησιμοποιούμε τις πιο κάτω δομές:

```
typedef struct Node{
     type data;
     struct Node *next;
     struct Node *prev;
} node;
```

```
typedef struct DLlist {
     node *top;
} dllist;
```

Μη-αναδρομική εκδοχή

```
EvenList1(dllist *L, dllist *new_list){
  node *p, *previous, *new_node;
  new_list->top = NULL;
  p = L->top;
  previous = NULL;
  while (p != NULL) {
     if (p->data mod 2 == 0) {
          new_node =(node *) malloc (sizeof(node));
          new_node->data = p->data;
          new_node->next = NULL;
          new_node->prev = previous;

          if (previous == NULL)
               new_list->top = new_node;
          else
               previous->next = new_node;
          previous = new_node;
     }
     p = p->next;
  }
}
```

Αναδρομική εκδοχή

```
RecEvenList1(dllist *L, dllist *new_list){
  new_list->top = RecEven1(L->top)
}

node *RecEven1(node *n){
node *new_node, p;
if (n == NULL)
     return NULL;

if (n->data mod 2 == 0) {
     new_node = (node *) malloc (sizeof(node));
     new_node->data = n->data;
     new_node->prev = NULL;
     p = RecEven1(n->next);
     new_node->next = p;
     if (p != NULL)
          p->prev = new_node;
     return new_node;
  }
  else
     return RecEven1(n->next);
}
```

(2) Κυκλική απλά συνδεδεμένη λίστα

Χρησιμοποιούμε τις πιο κάτω δομές:

```
typedef struct Node{
     type data;
     struct Node *next;
} node;
```

```
typedef struct  Clist {
     node *head;
} clist;
```

Μη-αναδρομική εκδοχή

```
EvenList2(Clist *L, Clist *new_list){
  node *p, *previous, *new_node;
  new_list->head = NULL;
  p = L->head;
  previous = NULL;
  if (p == NULL)
     return;
  do {
     if (p->data mod 2 == 0) {
          new_node =(node *) malloc (sizeof(node));
          new_node->data = p->data;
          if (previous == NULL){
               new_list->head = new_node;
               new_node->next = new_node;
          else
               previous->next = new_node;
               new_node->next = new_list->head;
          previous = new_node;
     }
     p = p->next;
  } while (p != L->head)
}
```

Αναδρομική εκδοχή

```
RecEvenList2(clist *L, clist *new_list){
   new_list->top = RecEven2(L, L->top, NULL, 0)
}
```

```
node *RecEven2(clist *L, node *n, node *first, int visited){
node *new_node;
  if (n == NULL)
     return NULL;

  if (n->data mod 2 == 0) {
     new_node = (node *) malloc (sizeof(node));
     new_node->data = n->data;
```

```
     (new_node)->next = RecEven2(L, n->next,first,1);
     return(new_node);
  }
  else
     if (visited == 1 and n == L->head)
          return first;
     else
          return RecEven2(L, n->next, first, 1);
}
```

(3) Κυκλική απλά συνδεδεμένη λίστα με κεφαλή

Χρησιμοποιούμε τις πιο κάτω δομές:

```
typedef struct Node{
     type data;
     struct Node *next;
} node;
```

```
typedef struct  Clist {
     node *head;
} clist;
```

Μη-αναδρομική εκδοχή

```
EvenList3(Clist *L, Clist *new_list){
  node *p, *previous, *new_node;
  new_node =(node *) malloc (sizeof(node));
  new_list->head = new_node;
  previous = new_node;
  p = L->head->next;
  while (p != L->head) {
     if (p->data mod 2 == 0) {
          new_node =(node *) malloc (sizeof(node));
          new_node->data = p->data;
          previous->next = new_node;
          new_node->next = new_list->head;
          previous = new_node;
     }
     p = p->next;
  }
}
```

Αναδρομική εκδοχή

```
RecEvenList3(clist *L, clist *newlist){
   head = (node *) malloc (sizeof(node));
   new_list-> head = head;
   head->next = RecEven(L, L->top->next, head)
}

node *RecEven3(clist *L, node *n, node *head_new){
```

```
node *new_node;

  if (n == L->head)
     return head_new;

  if (n->data mod 2 == 0) {
     new_node = (node *) malloc (sizeof(node));
     new_node->data = n->data;
     (new_node)->next=RecEven3(L, n->next, head_new);
     return(new_node);
  }
  else
     return RecEven3(L, n->next, head_new);
}
```