Φροντιστήριο 4 – Σκελετοί Λύσεων

Άσκηση 1

Υποθέτουμε πως οι λίστες είναι υλοποιημένες χρησιμοποιώντας τις πιο κάτω δομές.

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

typedef struct List{
   node *top;
   int size;
} list;
```

Υλοποιούμε διαδικασίες οι οποίες ελέγχουν κατά πόσο μια λίστα είναι ταξινομημένη σε αύξουσα σειρά.

Μη-αναδρομική εκδοχή

```
int IsSorted(list *L){

   p = L->top;
   if (p == NULL)
      report “The list is empty”;
      return 1;
   else
        q = p-> next;
        while (q != NULL)
          if (p->data <= q-> data)
             p = q;
             q = q-> next;
          else
              report “List not sorted”
              return 0;

   report “List is sorted”
   return 1;
}
```

Αναδρομική εκδοχή

```
int RecIsSorted(list *L){
  return RecIsIncreasing(L->top);
}

int RecIsIncreasing(node *p){
   if (p == NULL OR p->next == NULL)
      return 1;
   if (p->data <= (p->next)->data)
        return RecIsIncreasing(p->next);
   else
        return 0;
   }
```

Προφανώς και οι δύο αλγόριθμοι είναι της τάξης O(n), όπου n είναι ο αριθμός κόμβων της λίστας. Ο αναδρομικός αλγόριθμος είναι όμως λιγότερο αποδοτικός από άποψη χώρου (και χρόνου). Αυτό οφείλεται στο ότι για κάθε αναδρομική κλήση της διαδικασίας

δεσμεύεται επιπλέον χώρος για τις παραμέτρους και τις τοπικές μεταβλητές των κλήσεων. Κατά συνέπεια, κατά την επεξεργασία του τελευταίου κόμβου της λίστας θα υπάρχουν σε κατάσταση αναμονής n – 1 κλήσεις (όπου n είναι το μέγεθος της λίστας) που αφορούν όλους τους προηγούμενους κόμβους και για κάθε κλήση θα υπάρχει δεσμευμένη μνήμη για όλες τις παραμέτρους, τοπικές μεταβλητές, κλπ.

Άσκηση 2

Υποθέτουμε ότι η στοίβα θα χρησιμοποιηθεί για την αποθήκευση ακέραιων. Σύμφωνα με την προτεινόμενη τεχνική μια στοίβα είναι υλοποιημένη ως μία συνδεδεμένη λίστα, κάθε κόμβος της οποίας έχει τρία πεδία,

- ένα πίνακα όπου αποθηκεύονται οι πληροφορίες, μήκους maxsize,
- ένα ακέραιο που αντιπροσωπεύει την θέση κορυφής στον πίνακα που βρίσκεται στον κόμβο κορυφής (δηλαδή την θέση όπου θα γίνει η επόμενη εισαγωγή στοιχείου, και
- ένα δείκτη ο οποίος δείχνει στον επόμενο κόμβο της λίστας.

Η πιο κάτω εγγραφή ορίζει τον τύπο κάθε κόμβου:

```
struct hnode{
  int block[maxsize];
  int top;
  *hnode next;
}
```

Μία στοίβα ορίζεται ως εγγραφή με ένα πεδίο, το πεδίο start τύπου *hnode, που δείχνει στον πρώτο κόμβο της στοίβας

```
struct hstack{
  hnode    *start;
}
```

Ακολουθούν οι πράξεις σε ψευδοκώδικα:

```
MakeEmpty(hstack s){
   (*s).start = null;
}

bool IsEmpty(hstack s){
   return ((*s).start == null);
}

int TopNode(hstack s){
   if (!IsEmpty(s))
     hnode    *n = (*s).start;
     return ((*n).block[(*n).top –1]);
   else
     report ERROR
}

Push(hstack s,int x){
   hnode *n=(*s).start;
   if (n == null \/ (*n).top == maxsize )  {
     hnode    *m = *New(hnode);
     (*m).top = 1;
     (*m).next = n;
     (*s).start = m;
```

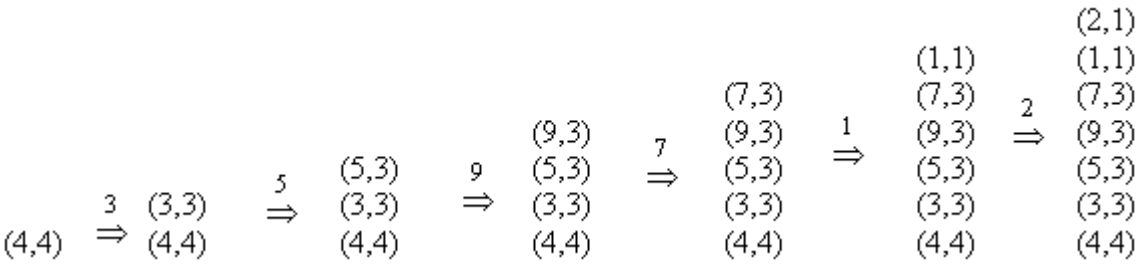
```
    }
   n=(*s).start;
   (*n).block[(*n).top-1] = x;
   (*n).top = (*n).top +1;
   }

Pop(hstack s){
   n=(*s).start;
   x= (*n).block[(*n).top-1];
   (*n).top = (*n).top -1;
   if ((*n).top==0)
        (*s).start= (*n).next;
        free(n);
        return x;
   }
   else ERROR;
}
```

Όλες οι διαδικασίες έχουν χρόνο εκτέλεσης O(1).

Άσκηση 3

Σε κάθε θέση της στοίβας αποθηκεύουμε ζεύγη στοιχείων όπου το πρώτο μέλος κάθε ζεύγους αντιστοιχεί στο στοιχείο που εισήχθηκε, και το δεύτερο μέλος του ζεύγους στο μέχρι στιγμής ελάχιστο στοιχείο της στοίβας.

Για παράδειγμα οι διαδοχικές εισαγωγές 4,3,5,9,7,1,2 έχουν ως αποτέλεσμα τις πιο κάτω καταστάσεις



Κατά συνέπεια, ανά πάσα στιγμή μπορούμε να βρούμε και να επιστρέψουμε το ελάχιστο στοιχείο της στοίβας σε χρόνο σταθερό (επιστρέφοντας το δεύτερο μέλος του ζεύγους που βρίσκεται στον κόμβο κορυφής της στοίβας).

Άσκηση 4

Ο ζητούμενος ΑΤΔ μπορεί να υλοποιηθεί ως εξής:

(i) Διαδοχική χορήγηση μνήμης

Υποθέτουμε ότι οι λίστες μας έχουν μέγιστο μέγεθος max και χρησιμοποιούμε τη δομή

```
typedef struct List{
    type elements[max];
    int size;
} list;
```

Οι διαδικασίες υλοποιούνται ως εξής:

```
type Access (list *L, int i){
    if (i< L->size)
        return L[i-1];
}
```

Χρόνος Εκτέλεσης: O(1)

```
InsertAfter(list *L, int i, type x){
  int k;

  if ( L->size < max AND i < L->size ){
     for ( k = L->size; k > i; k--)
        L->elements[k] = L->elements[k-1];
     L->elements[i] = x;
     L->size++;
  }
}
```

Χρόνος Εκτέλεσης: $n - i \in O(n)$, όπου n είναι το μέγεθος της λίστας.

```
Delete(list *L, int i){
  int k;

  if (i < L->size ){
     for ( k = i-1; k < L->size; k++)
           L->elements[k] = L->elements[k+1];
     L->size--;
  }
}
```

Χρόνος Εκτέλεσης: $n - i \in O(n)$, όπου n είναι το μέγεθος της λίστας.

(ii) Συνδετική χορήγηση μνήμης

Χρησιμοποιούμε τις πιο κάτω δομές

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

typedef struct List{
    node *top;
    int size;
} list;
```

Οι διαδικασίες υλοποιούνται ως εξής:

```
type Access (list *L, int i){
  node *p;
  int j;

  if i> L->size return;
```

```
  p = L->top;
  for (j=0; j < i; j++)
      p= p->next;
  return p->data;
}
```

Χρόνος Εκτέλεσης: i, O(n), όπου n είναι το μέγεθος της λίστας.

```
InsertAfter(list *L, int i, type x){
  node *p;
  int j;

  if (i > L->size) return;

  p = L->top;
  for (j=0; j < i; j++)
      p= p->next;
  q = (node *) malloc (sizeof(node));
  q->data = x;
  q->next = p->next;
  p->next = q;
}
```

Χρόνος Εκτέλεσης: i $\in$O(n), όπου n είναι το μέγεθος της λίστας.

```
Delete(list *L, int i){
  node * ptr, *p;
  int j;

  if (i > L->size) return;

  p = L->top;
  for (j=0; j < i; j++)
      p= p->next;
  ptr = p->next;
  p->next = p->next->next;
  free(ptr);
}
```

Χρόνος Εκτέλεσης: i $\in$O(n), όπου n είναι το μέγεθος της λίστας.

Η υλοποίηση του ΑΤΔ με διαδοχική χορήγηση μνήμης αποτελείται από απλούστερες προγραμματιστικά διαδικασίες και από αυτές η Access είναι αποδοτικότερη από την αντίστοιχη διαδικασία της υλοποίησης με δυναμική δέσμευση μνήμης. Το βασικό πλεονέκτημα της υλοποίησης με δυναμική δέσμευση μνήμης είναι ότι δεν επιβάλλει περιορισμούς σχετικά με το πλήθος των στοιχείων της λίστας, και ανά πάσα στιγμή χρησιμοποιεί μνήμη ανάλογη με το πλήθος των στοιχείων της λίστας. Αντίθετα η υλοποίηση με πίνακα δεσμεύει μνήμη για τις max θέσεις του πίνακα καθ'όλη τη διάρκεια της εκτέλεσης του προγράμματος, άσχετα με το πλήθος των στοιχείων της λίστας. Παρατηρούμε όμως, πως στην υλοποίηση με δυναμική δέσμευση μνήμης χρειάζεται να αποθηκεύουμε και μνήμη για του δείκτες next κάθε κόμβου της λίστας.