Δομές Δεδομένων

Μεταγλωττιστές,

Χειμερινό εξάμηνο 2018-2019

Διαδικαστικά

- Το εργαστήριο καλύπτει την εργασία εξαμήνου, η οποία περιλαμβάνει τη σταδιακή κατασκευή ενός απλού μεταγλωττιστή και γίνεται από ομάδες μέχρι τριών ατόμων
- Περιλαμβάνει χειρόγραφο και προγραμματιστικό κομμάτι
- Η παρουσία στο εργαστήριο μετράει θετικά **μόνο** εάν παραδοθεί το αντίστοιχο μέρος
- Στοιχεία επικοινωνίας
  - email : gefloros@uth.gr

Πρόγραμμα Εργαστηρίου

- 16/10 Δομές Δεδομένων
- 23/10 Εισαγωγή στο flex (jflap)
- 30/10 Κατασκευή λεκτικού αναλυτή με flex
- 6/11 Εισαγωγή στο bison
- 13/11 Bison, Symbol Table και συγκρούσεις
- 20/11 Εξέταση flex (λεκτικού αναλύτη)
- 27/11 Σημασιολογική ανάλυση
- 4/12 Ενδιάμεσος κώδικας
- 11/12 Εξέταση bison (συντακτικού αναλυτή)
- 18/12 Ενδιάμεσος κώδικας και ΑΣΔ
- 8/1 Τελικός κώδικας

Δομές Δεδομένων

- Μια δομή δεδομένων είναι μια συλλογή δεδομένων με κάποιες ιδιότητες η οποία προσφέρει εύκολη πρόσβαση σε αυτά τα δεδομένα

- Αυτοαναφορικές δομές
  - δομές που δείχνουν στον εαυτό τους λέγονται και δυναμικές δομές επειδή η μνήμη που καταλαμβάνουν τα αντίτυπά τους δεν είναι στατικά ορισμένη αλλά μπορεί να μεταβάλλεται “δυναμικά” στη διάρκεια εκτέλεσης του προγράμματος
  - δηλαδή υφίσταται δυναμική κατανομή (dynamic allocation) ανάλογα με τις ανάγκες της στιγμής

Δυναμικές Δομές Δεδομένων

- Το μέγεθος τους αυξομειώνεται ανάλογα με τις ανάγκες, έτσι ώστε να χρησιμοποιεί πάντα ακριβώς το χώρο που τους είναι απαραίτητος
  - (π.χ. ένας πίνακας για φοιτητές που το μέγεθος του να αυξάνεται κάθε φορά που ένας νέος φοιτητής εισάγεται στη βάση και να μειώνεται κάθε φορά που κάποιος διαγράφεται από αυτή)
- Η C ως γλώσσα προγραμματισμού δεν παρέχει τέτοιες δομές.
  - Παρέχει όμως τα απαραίτητα εργαλεία για να φτιάξουμε τέτοιες δομές δεδομένων
- Με χρήση Δεικτών, επιτρέπουν την έμμεση αναφορά στη μνήμη του υπολογιστή

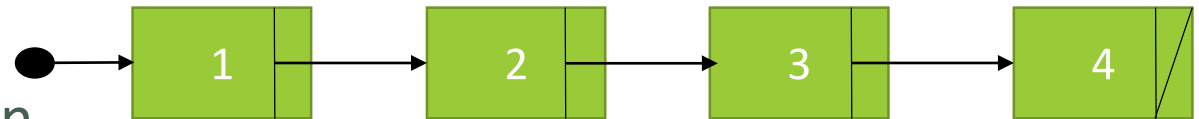
Δυναμικές Δομές Δεδομένων II

- Εφαρμογές
  - Λίστες
  - Πίνακες Κατακερματισμού
  - Γραφήματα – Δέντρα

- **Δυναμική Δέσμευση Μνήμης**
  Δέσμευση/Αποδέσμευση Μνήμης κατά τη εκτέλεση
- **malloc**
  Χρησιμοποιεί "**sizeof**" «Όρισμα» για να καθορίσει το μέγεθος ενός αντικειμένου (δομή)
  Παράδειγμα:
  newPtr = malloc(sizeof(struct node));
- **free**
  Αποδεσμεύει τη μνήμη που δεσμεύθηκε από malloc
  Χρησιμοποιεί Δείκτη για όρισμα
  Παράδειγμα:
  free (newPtr);

Απλά Συνδεδεμένες Δυναμικές Λίστες

- Μια συνεχόμενη αναπαράσταση στοιχείων(κόμβοι της λίστας)
  - Κάθε κόμβος συνδέεται με τον επόμενο μέσω μιας μεταβλητή δείκτη
- Παράδειγμα μιας απλής λίστας:



- Κάθε κόμβος αποτελείται από 2 μέρη
  - στο πρώτο αποθηκεύεται η πληροφορία που μας ενδιαφέρει
  - το δεύτερο μέρος είναι ένας δείκτης στον επόμενο κόμβο της λίστα
- Υλοποίηση Λίστας:
  - η υλοποίηση ενός κόμβου λίστας στη C θα γίνεται με μια δομή (struct)
  - αρχή της λίστας ορίζεται από μια μεταβλητή δείκτη που «δείχνει» στον πρώτο κόμβο
  - ο δείκτης στον τελευταίο κόμβο της λίστας έχει μια ειδική τιμή (null) που ορίζει ότι ο δείκτης αυτός δε δείχνει πουθενά

Δημιουργία Λίστας

- Αρχική Δήλωση Λίστας
  - Ορίζουμε τον τύπο των κόμβων της
  - Ορίζουμε μια μεταβλητή δείκτη σε έναν τέτοιο κόμβο

Κώδικας	Επεξήγηση
struct Komvos { int Plhroforia; struct Komvos *next; }; struct Komvos *Lista;	Κάθε κόμβος αποτελείται από ένα ακέραιο αριθμό Plhroforia και ένα δείκτη ο οποίος ονομάζεται next και δείχνει σε μια δομή τύπου Komvos

Lista ●⟶ ?

Δημιουργία Λίστας

- Δημιουργία Πρώτου κόμβου
  - χρήση συνάρτησης **malloc**
  - επιτρέπει τη δυναμική δέσμευση χώρου μνήμης για χρήση από κάποια μεταβλητή

ΜεταβλητήΔείκτη = (ΤύποςΚόμβου*) malloc(sizeof(ΤύποςΚόμβου));	Όρισμα το μέγεθος της δομής στην οποία θα δημιουργήσει ένα δείκτη [sizeof(ΤύποςΚόμβου)] Επιστρέφει ένα δείκτη σε μια τέτοια δομή, (ΤύποςΚόμβου *) της παραπάνω δήλωσης. Ο δείκτης αυτός εκχωρείται σε κάποια ΜεταβλητήΔείκτη, η οποία έτσι «δείχνει» σε μια «νέα» περιοχή της μνήμης που έχει δεσμευτεί για αυτό το σκοπό.

Δημιουργία Λίστας

- Δημιουργία Κόμβου στον οποίο θα δείχνει η Lista

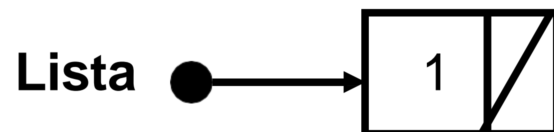
Lista = (struct Komvos *) malloc(sizeof(struct Komvos));
Δεσμεύεται στη μνήμη ελεύθερος χώρος ίσο με το χώρο που απαιτείται για μια μεταβλητή δομής τύπου Komvos Δίνεται τιμή στη μεταβλητή Lista, η διεύθυνση του πρώτου byte αυτού του νέου χώρου μνήμης

Δημιουργία Λίστας

- Απόδοση τιμών στα περιεχόμενα του νέου κόμβου

Χρήση τελεστή **->**	ΜεταβλητήΔείκτη -> ΌνομαΠεδίου

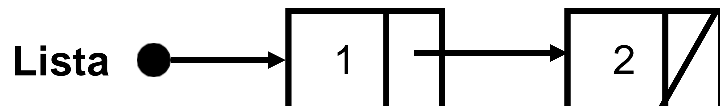
Κώδικας	Περιγραφή
Lista –> Plhroforia = 1; Lista –> next = NULL;	Απόδοση τιμών στα πεδία Plhroforia και next του κόμβου που δημιουργήσαμε



Δημιουργία Λίστας

- Δημιουργία επόμενου κόμβου

Code	Explanation
Komvos *temp;	1) Ορίζει μια μεταβλητή δείκτη σε κόμβο με το όνομα temp
temp =(struct Komvos *) malloc(sizeof(struct Komvos));	2)Δημιουργεί ένα νέο κόμβο στον οποίο θα δείχνει η temp
temp -> Plhroforia =2;	3)Αποδίδει περιεχόμενα κόμβο στον οποίο δείχνει η temp
temp -> next =NULL;	
Lista -> next =temp;	4) Συνδέει τους δυο κόμβους μέσω απόδοσης τιμής στη μεταβλητή Lista -> next έτσι ώστε αυτή να δείχνει στον ίδιο κόμβο που δείχνει και η temp



Διάσχιση Λίστας

- Αλγόριθμος Διάσχισης Λίστας
  - Ορίζουμε μια βοηθητική μεταβλητή δείκτη, έστω temp(απαραίτητη για να διασχίσουμε τη λίστα χωρίς να αλλάξουμε τα περιεχόμενα της μεταβλητής Lista)
  - Μετακινούμαστε από κόμβο σε κόμβο μέσω της μεταβλητής temp->next, η οποία δείχνει πάντα στον επόμενο κόμβο της λίστας
  - Η διάσχιση τελειώνει όταν η μεταβλητή temp->next έχει τιμή NULL(τέλος λίστας) ή όταν βρεθεί το ζητούμενο στοιχείο

```
temp =Lista;                              /* αρχικοποίηση μεταβλητής temp */
if (temp ==NULL)                          /* αν η λίστα είναι κενή */
    printf("Κενή Λίστα!")
else
 do
  printf("%d", temp -> Plhroforia); /*  εκτύπωση τιμής κάθε κόμβου */
  temp = temp -> next;                    /* μετακίνηση στον επόμενο κόμβο */
 while (temp!= NULL);                     /* όσο υπάρχουν ακόμα επόμενοι κόμβοι */
```

Αναζήτηση Στοιχείου σε Ταξινομημένη Λίστα

- Στις Λίστες μπορούμε να εφαρμόσουμε ΜΟΝΟ Γραμμική Αναζήτηση
- Αλγόριθμος παρόμοιος με αλγόριθμο διάσχισης

Αναζήτηση στοιχείου με τιμή Stoxos
Ο Αλγόριθμος τερματίζει αν βρεθεί στοιχείο με μεγαλύτερη τιμή από το ζητούμενο

```
temp = Lista;                              /* αρχικοποίηση μεταβλητής temp */
Vrethike = 0;                              /* αρχικοποίηση μεταβλητής: το στοιχείο δεν έχει βρεθεί */

while ((temp!=NULL) && (!Vrethike)) /* όσο δεν τέλειωσε η λίστα και δεν έχει βρεθεί */
 if (temp -> Plhroforia == Stoxos)    /* αν το στοιχείο βρεθεί */
     Vrethike = 1
 else if (temp -> Plhroforia > Stoxos)   /* αν ξεπεράσαμε την τιμή-στόχο */
     temp = NULL;                          /* "αναγκάζουμε" την επανάληψη να τελειώσει */
if (Vrethike)                              /* αν η επανάληψη τελειώσει επειδή το στοιχείο βρέθηκε */
     printf("Το στοιχείο υπάρχει στη λίστα")
else                                       /* αν η επανάληψη τελειώσει επειδή η λίστα εξαντλήθηκε */
     printf("Το στοιχείο δεν υπάρχει στη λίστα");
```

Εισαγωγή Στοιχείου στην Αρχή μη Ταξινομημένης Λίστας

- Δημιουργία κόμβου με τη συνάρτηση malloc
- Ο δείκτης next του κόμβου αυτού να δείχνει εκεί που δείχνει αρχικά ο δείκτης Lista
- Ο δείκτης Lista δείχνει στο νέο κόμβο
- Ονομάζουμε το νέο κόμβο Neos και έστω x το στοιχείο (πληροφορία) που θέλουμε να εισάγουμε

```
Neos = (struct Komvos *) malloc(sizeof(struct Komvos)); /* δημιουργία κόμβου */
Neos -> Plhroforia = x;                                  /* απόδοση τιμής στο νέο κόμβο */
Neos -> next = Lista;                     /* ο νέος κόμβος δείχνει στην προηγούμενη αρχή της λίστας */
Lista = Neos;                             /* η λίστα ξεκινάει πια από το νέο κόμβο */
```

Εισαγωγή Στοιχείου στο Τέλος μη Ταξινομημένης Λίστας

- Δημιουργία κόμβου με τη συνάρτηση malloc
- Διάσχιση της υπάρχουσας λίστας
- Ο δείκτης next του τελευταίου κόμβου να δείχνει στο νέο κόμβο

```
Neos = (struct Komvos *) malloc(sizeof(struct Komvos)); /* δημιουργία κόμβου */
Neos -> Plhroforia = x;                                 /* απόδοση τιμής στο νέο κόμβο */
Neos -> next = NULL;                                    /* ο νέος κόμβος δε δείχνει πουθενά */

if (Lista ==NULL)                                       /* αν η λίστα ήταν αρχικά άδεια */
 Lista = Neos;                                          /* η λίστα θα δείχνει στο νέο κόμβο */
else
 {temp =Lista;                                          /* βοηθητική μεταβλητή διάσχισης της λίστας */
  while (temp -> next != NULL)                          /* μέχρι να τελειώσει η λίστα */
   temp = temp -> next;                                 /* προχωράμε στον επόμενο κόμβο */
  temp -> next = Neos;                                  /* ο τελευταίος κόμβος θα δείχνει στο νέο */
 }
```

Εισαγωγή Στοιχείου σε Ταξινομημένη Λίστα

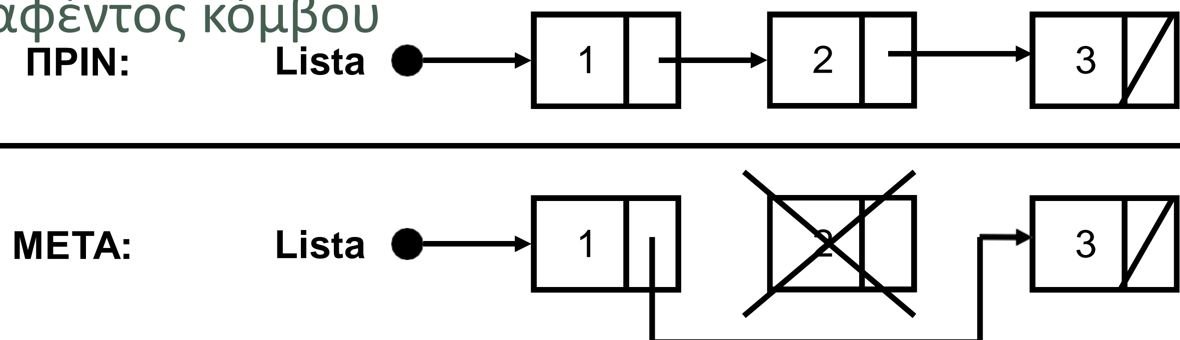
```
Neos = (struct Komvos *) malloc(sizeof(struct Komvos)); /* δημιουργία κόμβου */
Neos -> Plhroforia = x;                                 /* απόδοση τιμής στο νέο κόμβο */

if (Lista ==NULL)                                        /* αν η λίστα ήταν αρχικά άδεια */
 {Lista = Neos;                                         /* η λίστα θα δείχνει στο νέο κόμβο */
  Neos -> next = NULL;                                  /* ο νέος κόμβος δε θα δείχνει πουθενά */
else
 {temp = Lista;                                  /* βοηθητική μεταβλητή διάσχισης της λίστας */
  previous =NULL;                        /* βοηθητική μεταβλητή που δείχνει τον προηγούμενο κόμβο */

  while ((temp!=NULL) &&(temp -> Plhroforia < x))
                                   /* μέχρι να τελειώσει η λίστα ή να βρεθεί η σωστή θέση */

   {previous = temp;                        /* προχώρησε τον προηγούμενο κόμβο μια θέση */
    temp = temp -> next;                     /* προχώρησε τον τρέχοντα κόμβο μια θέση */
   }
  previous -> next = Neos;        /* ο δείκτης next του προηγούμενου κόμβου δείχνει το νέο */
  Neos -> next = temp;             /* ο δείκτης next του νέου κόμβου δείχνει τον επόμενο */
 }
```

Διαγραφή στοιχείου από λίστα

- Διαγραφή = Απελευθέρωση μνήμης
- Γίνεται με χρήση της συνάρτησης free (ουσιαστικά αντίθετη της malloc)

```
free(ΜεταβλητήΔείκτη);
```

- Αλγόριθμος Διαγραφής
  - διασχίζει πρώτα τη λίστα μέχρι να βρει τον κόμβο προς διαγραφή
  - ενημερώνει το δείκτη next του προηγούμενου κόμβου ώστε να δείχνει προς τον επόμενο
  - απελευθερώνει το χώρο μνήμης του διαγραφέντος κόμβου



Αλγόριθμος Διαγραφής - Παράδειγμα

```
if (Lista ==NULL)                                /* αν η λίστα είναι άδεια */
 printf("Δεν μπορεί να γίνει διαγραφή");
else if (Lista -> Plhroforia == x)               /* αν το ζητούμενο στοιχείο είναι το πρώτο */
 {temp =Lista;                                   /* ο τρέχων κόμβος είναι ο πρώτος */
  Lista = Lista ->next;                          /* η λίστα ξεκινάει από τον επόμενο κόμβο */
  free(temp);                                    /* απελευθερώνεται η μνήμη του τρέχοντος κόμβου */
 }
else
 {temp = Lista;                                  /* βοηθητική μεταβλητή διάσχισης της λίστας */
  previous =NULL;                                /* βοηθητική μεταβλητή που δείχνει τον προηγούμενο κόμβο */
  while ((temp!=NULL) &&(temp -> Plhroforia != x))
                                                 /* μέχρι να τελειώσει η λίστα ή να βρεθεί ο κόμβος */
   {previous = temp;                             /* προχώρησε τον προηγούμενο κόμβο μια θέση */
    temp = temp -> next;                         /* προχώρησε τον τρέχοντα κόμβο μια θέση */
   }
  if (temp!=NULL)                                /* αν το στοιχείο βρέθηκε */
   {previous->next =temp->next;                  /* ο δείκτης του προηγούμενου δείχνει το επόμενο */
    free(temp);                                  /* απελευθερώνεται η μνήμη του τρέχοντος κόμβου */
   }
  else                                           /* αν το στοιχείο προς διαγραφή δεν υπάρχει */
   printf("Το στοιχείο δεν υπάρχει στη λίστα");
}
```

Δυαδικά Δένδρα

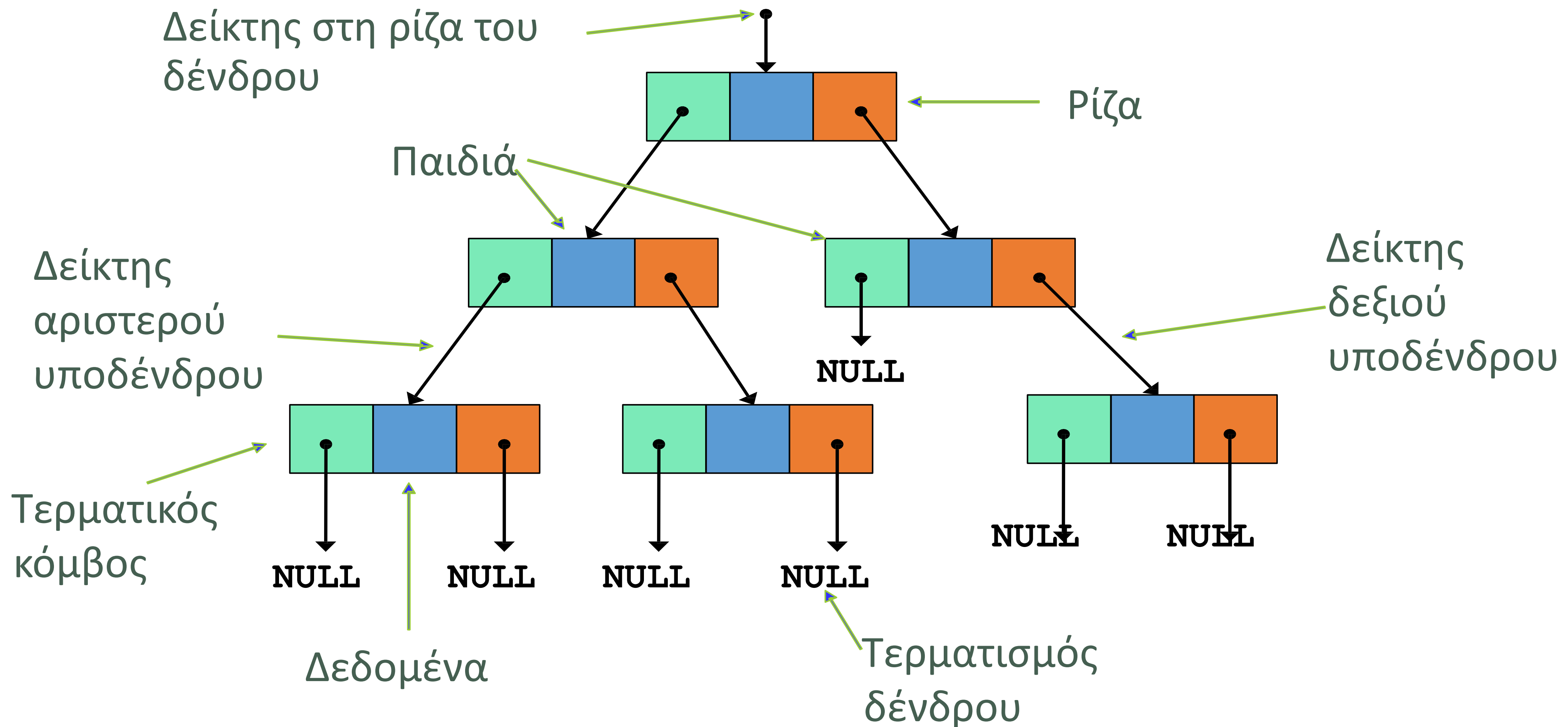
- Ένα δένδρο αποτελείται από στοιχεία που ονομάζονται κόμβοι (node)
- Οι κόμβοι είναι οργανωμένοι ιεραρχικά
- Ο κόμβος στην κορυφή της ιεραρχίας ονομάζεται ρίζα (root)
- Οι κόμβοι που βρίσκονται κάτω από έναν άλλο κόμβο είναι τα παιδιά του (children)
- Συνήθως τα παιδιά έχουν δικά τους παιδιά, κ.ο.κ.
- Κάθε κόμβος έχει ακριβώς έναν γονέα, εκτός από την ρίζα

Δυαδικά Δένδρα

- Ο αριθμός των παιδιών που μπορεί να έχει ένας κόμβος εξαρτάται από το είδος του δένδρου
  - Παράγοντας διακλάδωσης (branching factor)
- Δυαδικά δένδρα (binary tree)
  - Παράγοντας διακλάδωσης = 2
- Κάθε κόμβος του δυαδικού δένδρου είναι μία δομή με:
  - Ένα ή περισσότερα μέλη για τα δεδομένα του κόμβου
  - 1 δείκτη για το αριστερό παιδί (υποδένδρο)
  - 1 δείκτη για το δεξί παιδί (υποδένδρο)

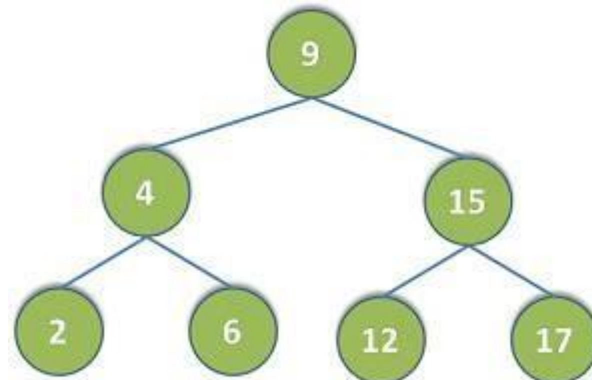
```
struct tnode
{ int data; ...
struct tnode *left;
struct tnode *right;
};
```

Δυαδικά Δένδρα



Δημιουργία Δυαδικού Δέντρου ακέραιων τιμών

- Τα παιδιά (κόμβοι) που έχουν μικρότερη τιμή από τον πατέρα τοποθετούνται αριστερά
- Τα παιδιά (κόμβοι) που έχουν μεγαλύτερη τιμή από τον πατέρα τοποθετούνται δεξιά
- Χρήση αναδρομικών συναρτήσεων για δημιουργία, αναζήτηση και διαγραφή



Δημιουργία Δυαδικού Δέντρου

```
11 void insert(node ** tree, int val) {
12 node *temp = NULL;
13 if(!(*tree)) {
14  temp = (node *)malloc(sizeof(node));
15  temp->left = temp->right = NULL;
16  temp->data = val;
17  *tree = temp;
18  return;
19 }
20
21 if(val < (*tree)->data) {
22    insert(&(*tree)->left, val);
23  } else if(val > (*tree)->data) {
24 insert(&(*tree)->right, val);
25  }
26 }
```

- Ελέγχει αν το δέντρο είναι άδειο και εισάγει κόμβο ως ρίζα (13-19)
- Ελέγχει αν η τιμή του κόμβου είναι μικρότερη της ρίζας (21) και
  - Καλεί την insert() αναδρομικά τόσο όσο δεν είναι τελευταίο αριστερό υπόδεντρο (not-NULL) (22)
  - Όταν φτάσει στο τελευταίο αριστερό NULL, εισάγει νέο κόμβο (13-19)
- Ελέγχει αν η τιμή του κόμβου είναι μεγαλύτερη της ρίζας (23) και
  - Καλεί αναδρομικά την insert() τόσο όσο δεν είναι τελευταίο δεξιό υπόδεντρο (not-NULL) (24)
  - Όταν φτάσει στο τελευταίο δεξιό NULL, εισάγει νέο κόμβο (13- 19)

Αναζήτηση σε Δυαδικό Δέντρου

```
46 node* search(node * *  tree, int val) {
47 if(!(*tree)) {
48    return NULL;
49  }
50  if(val == (*tree)->data) {
51    return *tree;
52   } else if(val < (*tree)->data){
53     search(&((*tree)->left), val);
54   } else if(val > (*tree)->data){
55     search(&((*tree)->right), val);
56  }
57 }
```

- Ελέγχει αν το δέντρο είναι άδειο επιστρέφοντας NULL (47-49)
- Ελέγχει αν η ζητούμενη τιμή του κόμβου είναι ίδια με της ρίζας (50-51)
- Ελέγχει αν η ζητούμενη τιμή του κόμβου είναι μικρότερη της ρίζας και τότε καλεί αναδρομικά την search() κοιτώντας το αριστερό κόμβο (52-53)
- Ελέγχει αν η ζητούμενη τιμή του κόμβου είναι μεγαλύτερη της ρίζας και τότε καλεί αναδρομικά την search() κοιτώντας το δεξιό κόμβο (54-55)
- Επαναλαμβάνει τα βήματα 2,3 και 4 μέχρι να βρει τον ζητούμενο κόμβο

Διαγραφή του Δυαδικού Δέντρου

```
38 void deltree(node * tree) {
39   if (tree) {
40     deltree(tree->left);
41     deltree(tree->right);
42   free(tree);
43   }
44 }
```

- Ελέγχει πρώτα αν ο κόμβος ρίζας είναι non-NULL (39), και
  - Καλεί αναδρομικά την deltree() όσο υπάρχει αριστερό κόμβο (40)
  - Καλεί αναδρομικά την deltree() όσο υπάρχει δεξιό κόμβο (41)
  - Διαγράφει τον κάθε κόμβο που συναντάει (42)

Διάσχιση - Εκτύπωση Δυαδικού Δέντρου

```
28 void print_preorder(node * tree) {
29 if (tree) {
30  printf("%d\n",tree->data);
31  print_preorder(tree->left);
32  print_preorder(tree->right);
33 }
34 }
35 void print_inorder(node * tree) {
36 if (tree) {
37  print_inorder(tree->left);
38  printf("%d\n",tree->data);
39  print_inorder(tree->right);
40 }
41 }
42  void print_postorder(node * tree) {
43  if (tree) {
44  print_postorder(tree->left);
45  print_postorder(tree->right);
46  printf("%d\n",tree->data);
47 }
48 }
```

- Η διάσχιση ενός δένδρου μπορεί να γίνει με τους παρακάτω τρόπους ανάλογα με τη σειρά επίσκεψης των κόμβων:
- Προδιαταγμένη διάσχιση (preorder traversal)
  - επίσκεψη της ρίζας
  - επίσκεψη του αριστερού υποδένδρου
  - επίσκεψη του δεξιού υποδένδρου
- Ενδοδιαταγμένη διάσχιση (inorder traversal)
  - επίσκεψη του αριστερού υποδένδρου
  - επίσκεψη της ρίζας
  - επίσκεψη του δεξιού υποδένδρου
- Μεταδιαταγμένη διάσχιση (postorder traversal)
  - επίσκεψη του αριστερού υποδένδρου
  - επίσκεψη του δεξιού υποδένδρου
  - επίσκεψη της ρίζας

Κατακερματισμός

- Ο κατακερματισμός (hashing) είναι μια μέθοδος για τη φύλαξη στοιχείων με βάση ένα κλειδί σε γραμμικές δομές δεδομένων (πίνακες, αρχεία) με στόχο τη γρήγορη ανεύρεσή τους

- Βασίζεται στη χρήση μιας συνάρτησης απεικόνισης με πεδίο ορισμού το κλειδί των στοιχείων και πεδίο τιμών τους δείκτες της αντίστοιχης δομής δεδομένων (λ.χ. ακέραιοι δείκτες σε πίνακα ή δείκτες θέσης σε αρχείο)

- Σε κάθε σύστημα κατακερματισμού πρέπει να λαμβάνουμε μέριμνα για την περίπτωση όπου δύο κλειδιά θα συμπίπτουν στην ίδια θέση της δομής μας

Κατακερματισμός -Ορισμοί

- Σε ένα σύστημα κατακερματισμού μπορούμε να ορίσουμε τα παρακάτω: Ονομάζουμε πυκνότητα κλειδιών (key density) το λόγο του πληθαρίθμου του συνόλου του πεδίου τιμών προς τον πληθάριθμο του συνόλου του πεδίου ορισμού της συνάρτησης κατακερματισμού
- Η διεύθυνση στην οποία απεικονίζει η συνάρτηση κατακερματισμού το κλειδί ονομάζεται βασική διεύθυνση (hash address)
- Ονομάζουμε συντελεστή φόρτισης (loading factor) το λόγο των στοιχείων της δομής που έχουν καταληφθεί προς τα στοιχεία της δομής που παραμένουν ελεύθερα
- Η συνάρτηση κατακερματισμού καλείται ομοιόμορφη (uniform) αν είναι ίσες οι πιθανότητες απεικόνισης κάθε πιθανής τιμής του κλειδιού

Πίνακας Κατακερματισμού

- Πίνακας κατακερματισμού (hash table) είναι μια δομή δεδομένων που υποστηρίζει τις διαδικασίες insert και find σε (σχεδόν) σταθερό χρόνο O(1).

- Ένας πίνακας κατακερματισμού χαρακτηρίζεται από
  - το μέγεθος του, **hsize,** και
  - κάποια **συνάρτηση κατακερματισμού h** η οποία αντιστοιχεί κλειδιά στο σύνολο των ακεραίων [0,…, hsize –1].

- Τα δεδομένα αποθηκεύονται στον πίνακα H[0,…, hsize –1]: ***το κλειδί k αποθηκεύεται στον H στη θέση H[h(k)].***

Συνάρτηση Κατακερματισμού

- Η συνάρτηση κατακερματισμού πρέπει να εκτελείται γρήγορα και να είναι κατά το δυνατό ομοιόμορφη. Οι παρακάτω είναι μερικές ενδεικτικές μέθοδοι υλοποίησης:
  - Λογικές μέθοδοι βασισμένες σε συναρτήσεις όπως η αποκλειστική διάζευξη
  - Πολλαπλασιαστικές μέθοδοι βασισμένες στον πολλαπλασιασμό
  - Χρήση του υπολοίπου
  - Μετατροπή βάσης
  - Διάσπαση του κλειδιού και πρόσθεση των τμημάτων

- Οι παραπάνω μέθοδοι μπορούν να χρησιμοποιηθούν και σε συνδυασμό

Βασική Ιδέα Κατακερματισμού

- Έχουμε την λίστα αριθμων S, S= {4, 10, 21, 32, 44, 58, 81, 302, 570}
- Θα φτιάξουμε ένα Πίνακα Κατακερματισμού (Hash Table), ο οποίος έχει μέγεθος hsize=7
- Χρησιμοποιώντας κάποια συνάρτηση κατακερματισμού (hashing function) h(key), θα εισάγουμε τα στοιχεία του S στο hashtable (hsize: 7)
  - Hash Function h(key): key mod hsize
- Αν ψάχνουμε το key=21, τότε ξέρω ότι πρέπει να ψάξω στην θέση 0 (21mod 7=0)

0	21		
1	302		
2	58	44	
3	570	10	
4	81	32	4
5			
6			

Δημιουργία Hash Table

```
struct _list_t_ {
    char *string;
    struct _list_t_ *next;
} list_t;
```

```
struct _hash_table_t_ {
    int size;
    list_t **table;
} hash_table_t;
```

```
hash_table_t *my_hash_table;
int size_of_table = 12;
my_hash_table =
 create_hash_table(size_of_table);
```

```
hash_table_t *create_hash_table(int size)
{
   hash_table_t *new_table;

   if (size<1) return NULL; /* invalid size for table */

if ((new_table = malloc(sizeof(hash_value_t))) == NULL) {
      return NULL;
   }


   if ((new_table->table = malloc(sizeof(list_t *) * size)) == NULL) {
      return NULL;
   }

for(i=0; i<size; i++) new_table->table[i] = NULL;

   /* Set the table's size */
   new_table->size = size;

   return new_table;
}
```

Συνάρτηση Κατακερματισμού – hash function

```
unsigned int hash(hash_table_t *hashtable, char*str)
{
    unsigned int hashval;
    hashval = 0;
    for(; *str != '\0'; str++) hashval = *str + (hashval << 5) - hashval;
 return hashval %  hashtable->size;
}
```

String lookup

```
list_t *lookup_string(hash_table_t *hashtable, char*str)
{
      list_t *list;
      unsigned int hashval = hash(hashtable, str);

for(list = hashtable->table[hashval]; list != NULL; list = list->next)
     {

            if (strcmp(str, list->str) == 0) return list;
      }
      return NULL;
}
```

Εισαγωγή string στο Hash Table

```
int add_string(hash_table_t *hashtable, char *str){
    list_t *new_list;
    list_t *current_list;
    unsigned int hashval = hash(hashtable, str);

    if ((new_list = malloc(sizeof(list_t))) == NULL) return 1;


    current_list = lookup_string(hashtable, str);
    if (current_list != NULL) return 2;
    /* Insert into list */
    new_list->str = strdup(str);
    new_list->next = hashtable->table[hashval];
    hashtable->table[hashval] = new_list;
    return 0;
}
```

Συνάρτηση διαγραφής string από hash table

```
int delete_string(hash_table_t *hashtable, char *str){
        int i;
        list_t *list, *prev;
        unsigned int hashval = hash(str);
                for(prev=NULL, list=hashtable->table[hashval];
                list != NULL && strcmp(str, list->str);
                prev = list;
                list = list->next;
        if (list==NULL) return 1; /* string does not exist in table */
        if (prev==NULL) hashtable[hashval] = list->next;
        else prev->next = list->next;
        free(list->str);
        free(list);
        return 0;
}
```

find table index

Πόσα string υπάρχουν στο hash table

```
int count_strings(hash_table_t *hashtable)
{
        int i, count = 0;
        list_t *list;


        if (hashtable==NULL) return -1;


        for(i=0; i<hashtable->size; i) {
                for(list=hashtable[i]; list != NULL; list = list->next) count++;
        }

        return count;
}
```

Διαγραφή Hash Table

```
void free_table(hash_table_t *hashtable){
    int i;
    list_t *list, *temp;
    if (hashtable==NULL) return;
    for(i=0; i<hashtable->size; i++) {
        list = hashtable->table[i];
        while(list!=NULL) {
            temp = list;
            list = list->next;
            free(temp->str);
            free(temp);
        }
    }
free(hashtable->table);
    free(hashtable);
}
```