ΔΟΜΕΣ ΔΕΔΟΜΕΝΩΝ – PROJECT 2018
ΑΝΑΦΟΡΑ

Ονοματεμπώνυμο : ΤΟΥΡΛΙΔΑ ΒΑΓΙΑ
Αριθμός Μητρώου : 1041905 (6233)
E-mail : tourlida@ceid.upatras.gr
Γλώσσα Προγραμματισμού : C++

ΜΕΡΟΣ Α

Ο τρόπος με το οποίο διαλέξαμε να διαβάσουμε το αρχείο ήταν μέσω ενός vector ακεραίων για την υλοποίηση του file read υλοποιήθηκε η παρακάτω συνάρτηση

```
void LoadData(ifstream &file,vector<string> &v)
{
int j=0;
long int data;

while(file>>data){
string x=to_string (data);
v.push_back(x);
j++;
}
file.close();
int sz=v.size();
cout<<"\bfile has been loaded\n";
cout<<"\b\nInitial array :\n";

    for (int i=0; i < sz; i++)
        cout<<"\b"<<v[i]<<"\n";

}
```

• Η ταξινόμηση με συγχώνευση είναι διαδικασία διαίρει και βασίλευε (divide and conquer, δηλ. αναδρομική διαδικασία όπου το πρόβλημα μοιράζεται σε μέρη τα οποία λύνονται ξεχωριστά, και μετά οι λύσεις συνδυάζονται.).

• Χρόνος Εκτέλεσης: O(n log n) .

• Περιγραφή του Mergesort 1. Μοιράζουμε τον πίνακα στα δύο. 2. Αναδρομικά ταξινομούμε τα δύο μέρη. 3. Συγχωνεύουμε τα αποτελέσματα των πιο πάνω αναδρομικών ταξινομήσεων.

Screenshoots των αποτελεσμάτων για ένα μικρό δείγμα εισόδου παρατίθενται παρακάτω:

```
[vtourlida@192 data_structures_2018_final]$ g++ last_version.cpp
[vtourlida@192 data_structures_2018_final]$ ./a.out
1.Reading a file with integers
2.Reading a file with words
1
file has been loaded

Initial array :
35910
423586
130085
316736
415018
1.Enter a number for searching:
2.Create a red black tree
3.Testing Times of each search
1

Sorted Array
35910
130085
316736
415018
423586

Array sorted with Merge Sort Algorithm
Enter the number you want to find:
```

ΜΕΡΟΣ Β

1. Linear Search (Γραμμική Αναζήτηση)

Πρόκειται για έναν αλγόριθμο “ωμής βίας” και είναι ο πιο απλός σε σχέση με τους υπόλοιπους. Η λογική του είναι ότι ξεκινάμε από την αρχή της δομής και κάνουμε συνεχής ελέγχούς μέχρις ότου βρούμε το στοιχείο που αναζητούμε. Στην περίπτωση που το στοιχείο δεν υπάρχει στη δομή τότε ο αλγόριθμος σταματάει αφού ελένξει όλα τα στοιχεία της δομής. Αυτό έχει ως αποτέλσμα το άνω φράγμα του αλγορίθμου να είναι O (n) και στη χειρότερη περίπτωση να χρειάζεται n συγκρίσεις.

Για την υλοποίηση του Linear Search δημιουργήθηκε η παρακάτω συνάρτηση:

```
void linear_search(vector<int> &v,int searching_num)
{
  int pos,flag=0;
  for(int i=0;i<v.size();i++)
  {
    if(searching_num==v[i]){
    flag=1;
    pos=i;
    }
  }
  if(flag==1)
  {
  cout<<"the "<<searching_num<<" is found at the location "<<pos<<endl;
  }else
  {
  cout<<"The searching number "<<searching_num<<" does not contained in our file!\n"<<endl;
  }

}
```

2. Binary Search (Δυϊκή Αναζήτηση)

Το Binary Search ανήκει στο είδος των αλγορίθμων “Διαίρει και Βασίλευε”. Η λογική του είναι πως σε κάθε βήμα (while loop) ελέγχει το μέσο της δομής και μετά εξετάζει , αν χρειάζεται ένα από τα δύο ίσα διαστήματα (είναι ίσα αν το n είναι περιττός αριθμός, αλλιώς τα διαστήματα θα διαφέρουν κατά ένα). Προόδευτικα, καθώς το προς εξέταση διάστημα υποδιπλασιάζεται, μετά από καποίο αριθμό επαναλήψεων, έστω h, αυτό θα αποκτήσει μήκος ένα. Στο επόμενο βήμα η πράξη σίγουρα θα τερματίστει γιατί είτε το μοναδικό στοιχείο θα είναι ίσο με το x, αυτό δηλαδή που αναζητάμε, είτε το διάστημα θα γίνει μηδενικό και τότε δεν θα υπάρχει το x στα στοιχεία του πίνακα. Η χειρότερη περίπτωση χρειάζεται χρόνο O (n logn) και αριθμό συγκρίσεων ίσο με h. Στη συνέχεια δίνουμε και την απόδειξη.

Binary Search with O(n*h) = O(n logn):

$$n(\frac{1}{2})^h = 1 => (\frac{1}{2})^h = \frac{1}{h} => log_2(\frac{1}{2})^h = log_2\frac{1}{n} => hlog_2(\frac{1}{2}) = log_2\frac{1}{n} => h = log_2 n$$

Για την υλοποίηση του **Binary** Search δημιουργήθηκε η παρακάτω συνάρτηση:

```
void binary_search(vector<int> &v,int searching_num)
{
  int first=0;
  int last=v.size();
  int middle=(first+last)/2;

  while(first<=last)
  {
  if(v[middle]<searching_num)
  {
  first=middle+1;
  }else if(v[middle]==searching_num){

  cout<<"search is found at the location "<<middle+1<<"\n";
  break;
  }else
  {
  last=middle-1;
  }
  middle=(first+last)/2;
  }
  if(first>last)
  {
  cout<<"The searching number "<<searching_num<<" does not contained in our file!"<<endl;

  }

  }
```

3. Interpolation Search (Αναζήτηση Παρεμβολής)

Η αναζήτηση παρεμβολής είναι ανάλογη με τον τρόπο που ένας άνθρωπος ψάχνει σε ένα λεξικό ή σε έναν τηλεφωνικό κατάλαγο. Όταν ψάχνουμε για μία λέξη σε ένα λεξικο δεν συνηθίζουμε να κάνουμε δυϊκή αναζήτηση. Για παράδειγμα αν ψάχνουμε μια λέξη που αρχίζει από Α ανοίγουμε το λεξικό από την αρχή, ενώ αν αρχίζει με Ω το ανοίγουμε προς το τέλος του. Έτσι λοιπόν στην αναζήτηση παρεμβολής όταν γνωρίζουμε ότι το χ βρίσκεται μεταξύ του S[1] και του S[n] η γενικότερα του S[left] και του S[right] το στοιχείο που επιλέγεται από τη σχέση ((x – S[left])/S[right] – S[left]). Στην ουσία εξετάζετε πόσο μεγαλύτερο είναι το ζητούμενο στοιχείο από το αριστερό άκρο καθώς και το πόσο μεγαλύτερο είναι το δεξί άκρο από το αριστερό γίνεται εκτίμηση της θέσης του x παίρνοντας το λόγο των δύο διαφορών. Όταν κάθε στοιχείο διαφέρει από τα γειτονικά του κατά μία σταθερή ποσότητα και όχι κατά ένα και τότε η εκτίμηση είναι αρκετά κοντά. Ο χρόνος χειρότερης περίπτωσης για την αναζήτηση με παρεμβολή είναι O(n) ενώ ο μέσος χρόνος είναι O(log logn). Παρ' όλα αυτά , πειραματικά αποτελέσματα έδειξαν ότι η αναζήτηση παρεμβολής δεν εκτελεί πολύ λιγότερες συγκρίσεις έτσι ώστε να γλιτώνει από το αυξανόμενο υπολογιστικό κόστος εύρεσης του next και να έχει καλύτερους χρόνους, εκτός και αν οι πίνακες είναι πολύ μεγάλοι. Για αυτό πολλές φορές είναι προτιμότερο τα πρώτα βήματα να εκτελούν αναζήτηση παρεμβολής, έτσι ώστε το διαστήμα να μειώνεται δραματικά και στη συνέχεια να εκτελείται δυϊκή αναζήτηση.

Για την υλοποίηση του **Interpolation Search** δημιουργήθηκε η παρακάτω συνάρτηση:

```
void interpolation_search(vector<int> &v,int searching_num)
{
int low=0;
int high=v.size();
int flag=0;
while(low<=high && searching_num>=v[low] && searching_num<=v[high])
{
int pos=low+ (((double)(high-low) / (v[high]-v[low]))*(searching_num-v[low]));
if(v[pos]==searching_num) {cout<<"search is found at the location "<<pos<<endl;flag=1;}
if(v[pos]<searching_num) {
low=pos+1;
}else{
high=pos-1;
}
}
if(flag==0){cout<<"The searching number "<<searching_num<<" does not contained in our file!"<<endl;}

}
```

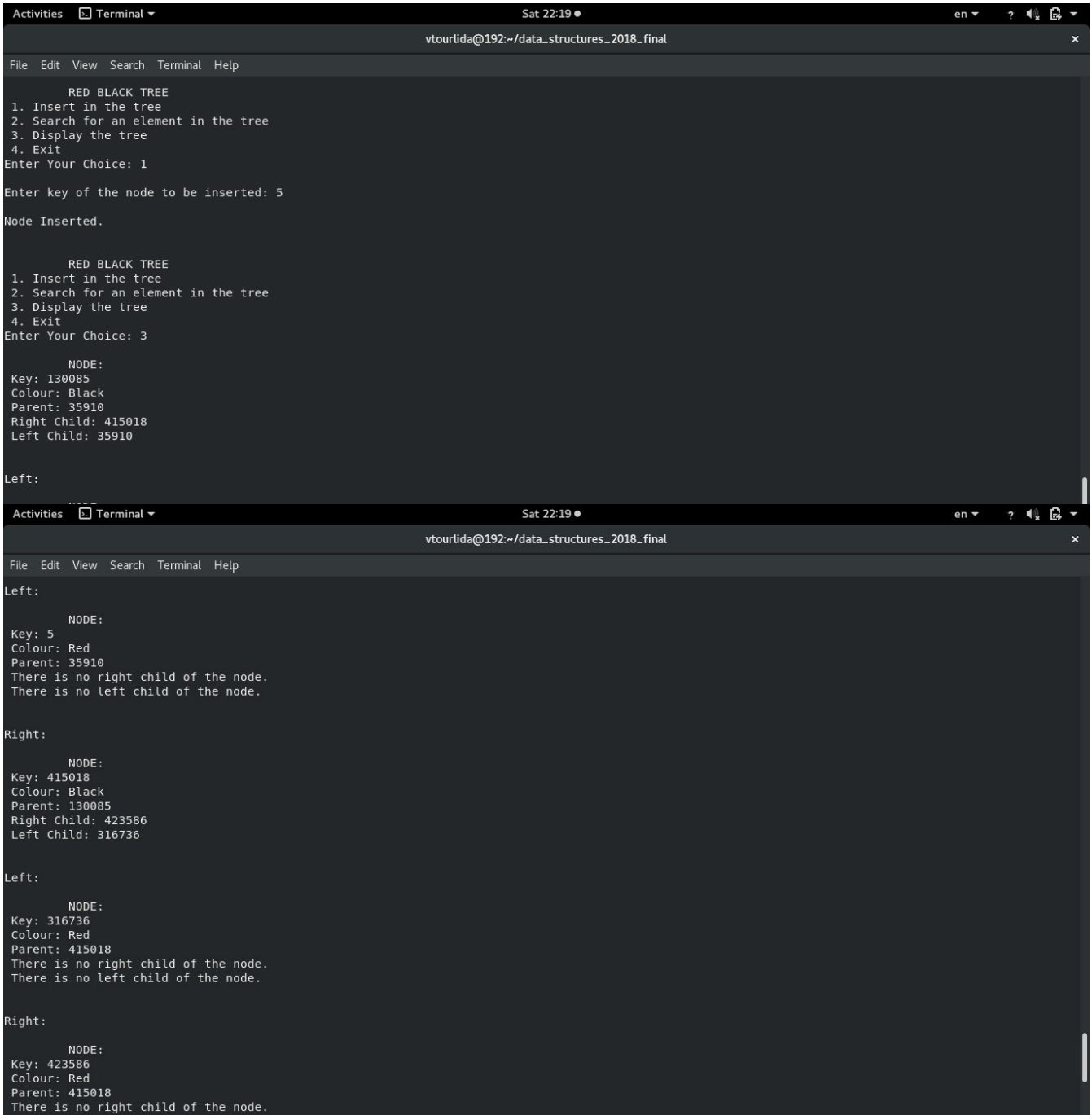
Μέρος Γ

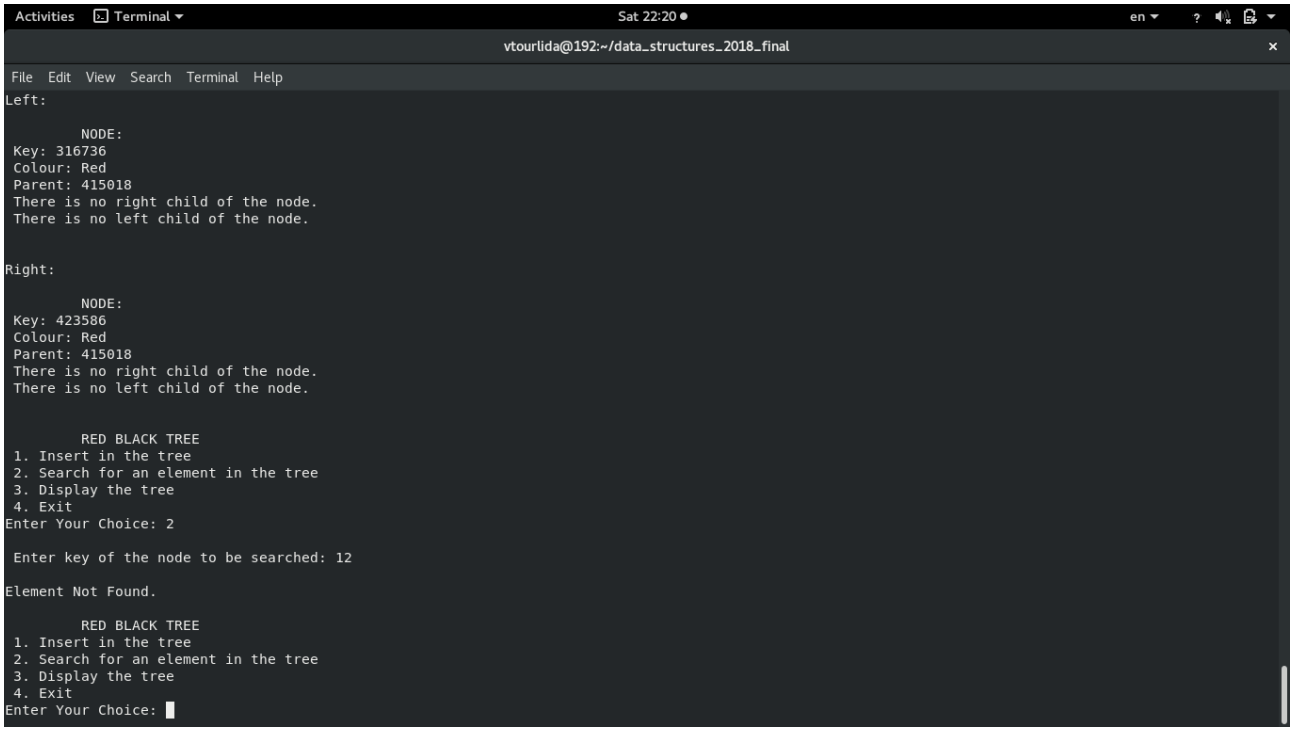
Red black

Το **Red black** δέντρο ανήκει στη κατηγορία των υψοζυγισμένων δυαδικών δέντρων (Height Balanced Tree). Αυτό σημαίνει ότι για κάθε κόμβο του ισχύει πως τα ύψη των υποδένδρων του διαφέρουν το πολύ κατά ένα.Ακοόλουθεί τις εξής ιδιότητες:

- Η ρίζα να είναι μαύρη .
- Τα φύλλα να είναι πάντα μαύρα.
- Κάθε μονοπάτι από τη ρίζα ως τα φύλλα έχει τον ίδιο αριθμό μαύρων κόμβων.
- Κάθε κόκκινος κόμβος έχει μαύρο πατέρα.

Screenshot αποτελεσμάτων

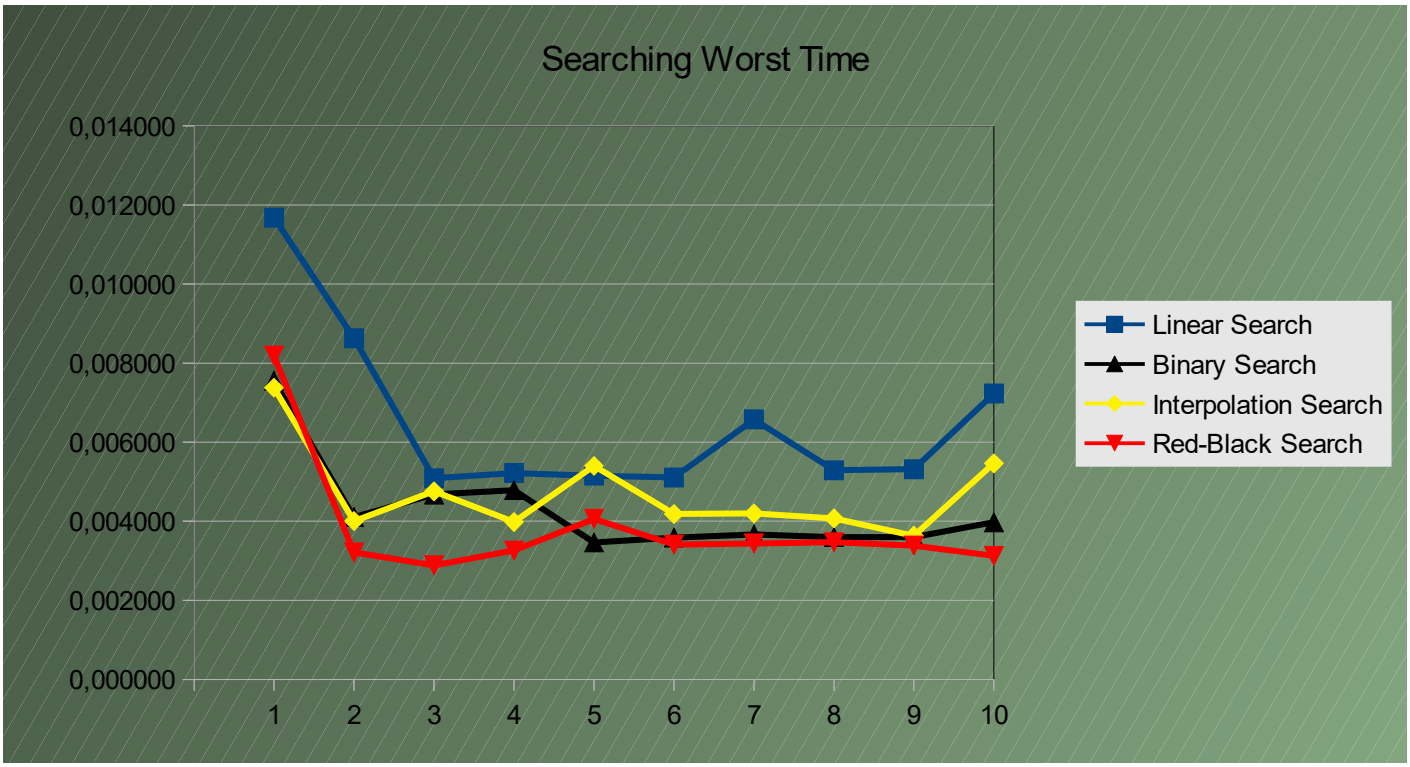




Μέρος Δ

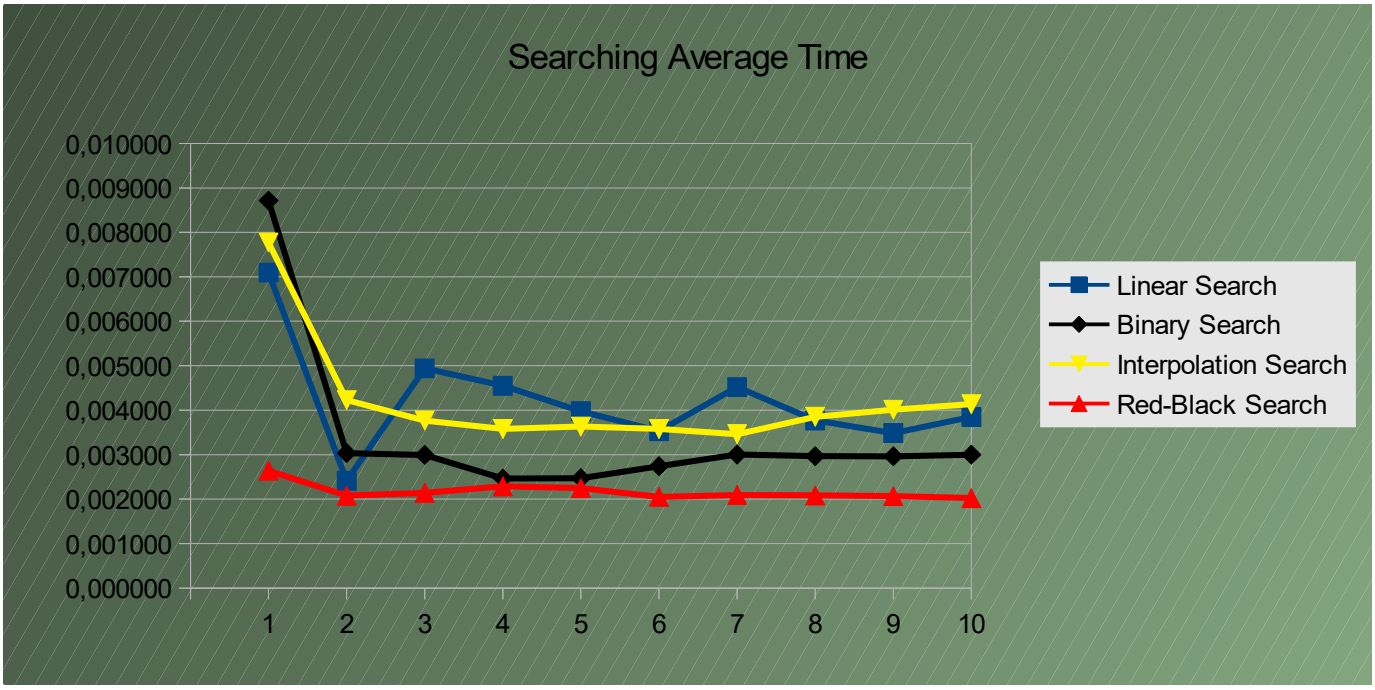
Χρησιμοποιώντας το αρχείο που μας δίνεται εκτελούμε ένα μεγάλο αριθμό από αναζητήσεις με βάση ένα όνομα. Κάναμε 10 πειράματα για τις τέσσεροις διαφορετικές αναζήτησης. Αυτά τα πειράματα εκτελούνται με τυχαία σύνολα ερωτημάτων του ίδιου μεγέθους(100). Στο παρακάτω πίνακα φαίνονται οι χειρότεροι χρόνοι των πειραμάτων :

Πειράματα	**Binary Search**	**Interpolation Search**	**Linear Search**	**Red Black Search**
1ο	0.007555	0.007376	0.011681	0.008188
2ο	0.004106	0.004007	0.008629	0.00321
3ο	0.004677	0.004757	0.005087	0.002886
4ο	0.004786	0.003982	0.005221	0.003259
5ο	0.003463	0.005399	0.005152	0.004059
6ο	0.003587	0.004183	0.005107	0.00341
7ο	0.003665	0.004197	0.006576	0.003439
8ο	0.003599	0.004073	0.005291	0.00347
9ο	0.003597	0.003636	0.005592	0.003384
10ο	0.003605	0.003785	0.005318	0.004559



Στο παρακάτω πίνακα φαίνονται οι μέσοι χρόνοι των πειραμάτων :

Linear Search	**Interpolation Search**	**Binary Search**	**Red Black Search**
0.007095	0.007774	0.008716	0.00637
0.002414	0.004228	0.003034	0.000757
0.004942	0.003767	0.002993	0.001384
0.004551	0.003578	0.002462	0.002879
0.003978	0.003631	0.002471	0.02461
0.00353	0.003578	0.003002	0.000468
0.004522	0.003461	0.002739	0.00938
0.00377	0.003851	0.002966	0.000837
0.003485	0.004012	0.002962	0.000675
0.004225	0.004125	0.003060	0.000535



Από τα στοιχεία του παραπάνω πίνακα καταλήγουμε στο εξής συμπέρασμα : Interpolation Search < Binary Search <= Red Search < Linear Search <==> O (log logn) < O (n logn) <= O (h logh) < O (n) .

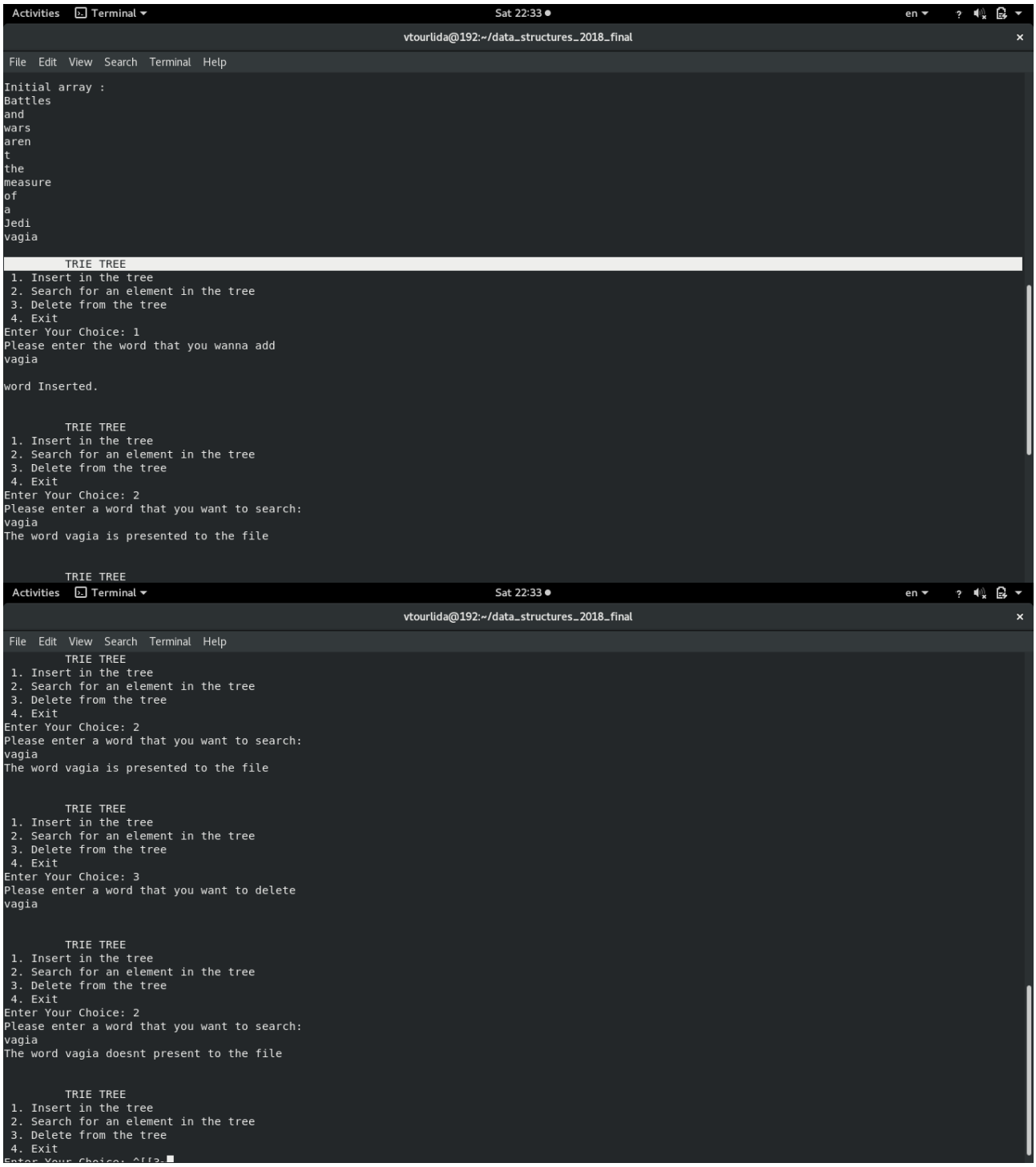
Μέρος Ε

Trie

Αποτελεί μια αποτελεσματική δομή δεδομένων ανάκτησης πληροφοριών. Χρησιμοποιώντας το trie, η πολυπλοκότητα αναζήτησης μπορεί να φτάσει στο βέλτιστο όριο δηλαδή το μήκος κλειδιού. Εάν αποθηκεύσουμε κλειδιά σε δυαδικό δένδρο αναζήτησης, ένα καλά ισορροπημένο BST θα χρειαστεί χρόνο ανάλογο με το M * log N, όπου το M είναι το μέγιστο μήκος της συμβολοσειράς και το N είναι ο αριθμός των κλειδιών στο δέντρο. Χρησιμοποιώντας το trie, μπορούμε να αναζητήσουμε το κλειδί σε χρόνο O (M). Ωστόσο, έχει ένα μειονέκτημα όσων αφορά τις απαιτήσεις αποθήκευσης.Κάθε κόμβος trie αποτελείται από πολλαπλούς κλάδους. Κάθε διακλάδωση οδηγεί σε νέο σύνολο κλειδιών που αντιπροσωπεύει έναν διαφορετικό αλφαριθμητικό . Πρέπει να σημειώσουμε τον τελευταίο κόμβο κάθε κλειδιού ως κόμβο φύλλων. Για την διάκριση του κόμβου ως κόμβου φύλλου θα χρησιμοποιηθεί μια τιμή πεδίου κόμβου trie (συνήθους τύπου boolean). Κάθε χαρακτήρας του αλφαριθμητικού εισόδου εισάγεται ως μεμονωμένος κόμβος στο trie. Εάν ο χαρακτήρας του αλφαριθμητικού εισόδου είναι καινούργιος ή μια επέκταση του υπάρχοντος κλειδιού, πρέπει να κατασκευάσουμε μη υπάρχοντες κόμβους του trie και να σημειώσουμε τον κόμβο των φύλλων. Αν ο χαρακτήρας του αλφαριθμητικού εισόδου είναι πρόθεμα του υπάρχοντος κλειδιού στο trie, απλά

σημειώνουμε τον τελευταίο κόμβο του κλειδιού ως φύλλο. Το μήκος του αλφαριθμητικού καθορίζει το βάθος του trie. Η αναζήτηση ενός χαρακτήρα ή ενός αλφαριθμητικού είναι παρόμοια με την λειτουργία εισαγωγής, αλλά συγκρίνουμε μόνο τους χαρακτήρες και προχωρούμε προς τα κάτω. Η αναζήτηση μπορεί να τερματιστεί λόγω του τέλους της συμβολοσειράς ή της έλλειψης κλειδιού στο trie. Στην πρώτη περίπτωση, εάν το πεδίο τιμών του τελευταίου κόμβου είναι μη-μηδένικο τότε το κλειδί υπάρχει στο trie. Στη δεύτερη περίπτωση, η αναζήτηση τερματίζεται χωρίς να εξεταστούν όλοι οι χαρακτήρες του κλειδιού, δεδομένου ότι το κλειδί δεν υπάρχει στο trie. Τέλος, η εισαγωγή και αναζητήση αλφαριθμητικού στο trie έχουν κόστος O (key_length), ωστόσο οι απαιτήσεις μνήμης του trie είναι αρκετά υψηλές αφού O (ALPHABET_SIZE * key_length * N) όπου N είναι ο αριθμός των κόμβων στο trie.

Screenshot αποτελεσμάτων


Activities
Terminal
Sat 22:33
en
vtourlida@192:~/data_structures_2018_final
File Edit View Search Terminal Help
Initial array :
Battles
and
wars
aren
t
the
measure
of
a
Jedi
vagia
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit
Enter Your Choice: 1
Please enter the word that you wanna add
vagia
word Inserted.
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit
Enter Your Choice: 2
Please enter a word that you want to search:
vagia
The word vagia is presented to the file
TRIE TREE
Activities
Terminal
Sat 22:33
en
vtourlida@192:~/data_structures_2018_final
File Edit View Search Terminal Help
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit
Enter Your Choice: 2
Please enter a word that you want to search:
vagia
The word vagia is presented to the file
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit
Enter Your Choice: 3
Please enter a word that you want to delete
vagia
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit
Enter Your Choice: 2
Please enter a word that you want to search:
vagia
The word vagia doesnt present to the file
TRIE TREE
1. Insert in the tree
2. Search for an element in the tree
3. Delete from the tree
4. Exit

```
int searchTrie(trie_t *pTrie, string key)
{
    int level;
    int index;
    trie_node_t *pCrawl;

    pCrawl = pTrie->root;

    for( level = 0; level < key.size(); level++ )
    {
        index = INDEX(key[level]);
 if( !pCrawl->children[index] )
        {
            return 0;
        }

        pCrawl = pCrawl->children[index];
    }

    return (0 != pCrawl && pCrawl->value);
}
```