Δομές Δεδομένων και Αλγόριθμοι
ΜΕΡΟΣ Γ’
Εργαστήριο (C++)
Τ.Ε.Ι. Ηπείρου - Τμήμα Μηχανικών Πληροφορικής Τ.Ε.

Χρήστος Γκόγκος

Άρτα - 2017

Εργαστήριο 7

Κατακερματισμός, δομές κατακερματισμού στην STL

7.1 Εισαγωγή

Ο κατακερματισμός (hashing) αποτελεί μια από τις βασικές τεχνικές στη επιστήμη των υπολογιστών. Χρησιμοποιείται στις δομές δεδομένων αλλά και σε άλλα πεδία της πληροφορικής όπως η κρυπτογραφία. Στο εργαστήριο αυτό θα παρουσιαστεί η δομή δεδομένων πίνακας κατακερματισμού χρησιμοποιώντας δύο διαδεδομένες υλοποιήσεις: την ανοικτή διευθυνσιοδότηση και την υλοποίηση με αλυσίδες. Επιπλέον, θα παρουσιαστούν δομές της STL όπως η unordered_set και η unordered_map οι οποίες στηρίζονται στην τεχνική του κατακερματισμού. Ο κώδικας όλων των παραδειγμάτων, όπως και στα προηγούμενα εργαστήρια, βρίσκεται στο https://github.com/chgogos/ceteiep_dsa.

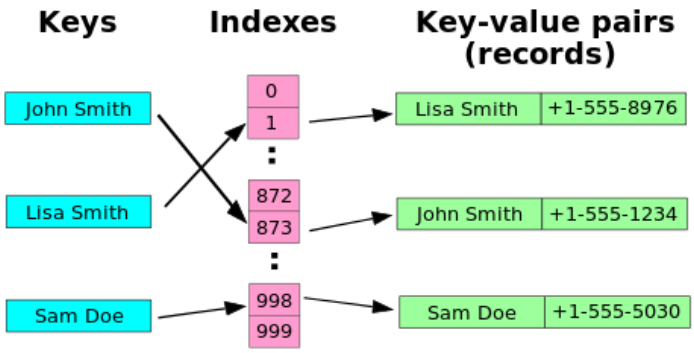
7.2 Τι είναι ο κατακερματισμός;

Ο κατακερματισμός είναι μια μέθοδος που επιτυγχάνει ταχύτατη αποθήκευση, αναζήτηση και διαγραφή δεδομένων. Σε ένα σύστημα κατακερματισμού τα δεδομένα αποθηκεύονται σε έναν πίνακα που ονομάζεται πίνακας κατακερματισμού (hash table). Θεωρώντας ότι τα δεδομένα είναι εγγραφές που αποτελούνται από ζεύγη τιμών της μορφής κλειδί-τιμή, η βασική ιδέα είναι, ότι εφαρμόζοντας στο κλειδί κάθε εγγραφής που πρόκειται να αποθηκευτεί ή να αναζητηθεί τη λεγόμενη συνάρτηση κατακερματισμού (hash function), προσδιορίζεται μονοσήμαντα η θέση του πίνακα στην οποία τοποθετούνται τα δεδομένα της εγγραφής. Η συνάρτηση κατακερματισμού αναλαμβάνει να αντιστοιχήσει έναν μεγάλο αριθμό ή ένα λεκτικό σε ένα μικρό ακέραιο που χρησιμοποιείται ως δείκτης στον πίνακα κατακερματισμού.

Μια καλή συνάρτηση κατακερματισμού θα πρέπει να κατανέμει τα κλειδιά στα κελιά του πίνακα κατακερματισμού όσο πιο ομοιόμορφα γίνεται και να είναι εύκολο να υπολογιστεί. Επίσης, είναι επιθυμητό το παραγόμενο αποτέλεσμα από τη συνάρτηση κατακερματισμού να εξαρτάται από το κλειδί στο σύνολό του.

Στον κώδικα που ακολουθεί παρουσιάζονται τέσσερις συναρτήσεις κατακερματισμού κάθε μία από τις οποίες δέχεται ένα λεκτικό και επιστρέφει έναν ακέραιο αριθμό. Στις συναρτήσεις hash2 και hash3 γίνεται χρήση τελεστών που εφαρμόζονται σε δυαδικές τιμές (bitwise operators). Ειδικότερα χρησιμοποιούνται οι τελεστές << (αριστερή ολίσθηση), >> (δεξιά ολίσθηση) και ^(xor - αποκλειστικό ή).

```
1 #include <string>
2
3 using namespace std;
4
5 size_t hash0(string &key) {
6   size_t h = 0;
7   for (char c : key)
8     h += c;
```



Σχήμα 7.1: Κατακερματισμός εγγραφών σε πίνακα κατακερματισμού [1]

```
9    return h;
10 }
11
12 size_t hash1(string &key) {
13    size_t h = 0;
14    for (char c : key)
15       h = 37 * h + c;
16    return h;
17 }
18
19 // Jenkins One-at-a-time hash
20 size_t hash2(string &key) {
21    size_t h = 0;
22    for (char c : key) {
23       h += c;
24       h += (h << 10);
25       h ^= (h >> 6);
26    }
27    h += (h << 3);
28    h ^= (h >> 11);
29    h += (h << 15);
30    return h;
31 }
32
33 // FNV (—FowlerNollVo) hash
34 size_t hash3(string &key) {
35    size_t h = 0x811c9dc5;
36    for (char c : key)
37       h = (h ^ c) * 0x01000193;
38    return h;
39 }
```

Κώδικας 7.1: Διάφορες συναρτήσεις κατακερματισμού (hashes.cpp)

```
1 #include "hashes.cpp"
2 #include <iostream>
3
4 using namespace std;
5
6 int main() {
7    constexpr int HT_SIZE = 101;
8    string keys[] = {"nikos", "maria", "petros", "kostas"};
9    for (string key : keys) {
```

```
10      size_t h0 = hash0(key) % HT_SIZE;
11      size_t h1 = hash1(key) % HT_SIZE;
12      size_t h2 = hash2(key) % HT_SIZE;
13      size_t h3 = hash3(key) % HT_SIZE;
14      cout << "string " << key << " hash0=" << h0 << " hash1=" << h1
15           << ", hash2=" << h2 << ", hash3=" << h3 << endl;
16    }
17  }
```

Κώδικας 7.2: Παραδείγματα κλήσεων συναρτήσεων κατακερματισμού (hashes_ex1.cpp)

```
1  string nikos hash0=43 hash1=64, hash2=40, hash3=27
2  string maria hash0=17 hash1=98, hash2=71, hash3=33
3  string petros hash0=63 hash1=89, hash2=85, hash3=82
4  string kostas hash0=55 hash1=69, hash2=17, hash3=47
```

Οι πίνακες κατακερματισμού είναι ιδιαίτερα κατάλληλοι για εφαρμογές στις οποίες πραγματοποιούνται συχνές αναζητήσεις εγγραφών με δεδομένες τιμές κλειδιών. Οι βασικές λειτουργίες που υποστηρίζονται σε έναν πίνακα κατακερματισμού είναι η εισαγωγή (insert), η αναζήτηση (get) και η διαγραφή (erase). Και οι τρεις αυτές λειτουργίες παρέχονται σε χρόνο $O(1)$ κατά μέσο όρο προσφέροντας ταχύτερη υλοποίηση σε σχέση με άλλες υλοποιήσεις όπως για παράδειγμα τα ισοζυγισμένα δυαδικά δένδρα αναζήτησης που παρέχουν τις ίδιες λειτουργίες σε χρόνο $O(logn)$.

Ωστόσο, οι πίνακες κατακερματισμού έχουν και μειονεκτήματα καθώς είναι δύσκολο να επεκταθούν από τη στιγμή που έχουν δημιουργηθεί και μετά. Επίσης, η απόδοση των πινάκων κατακερματισμού υποβαθμίζεται καθώς οι θέσεις τους γεμίζουν με στοιχεία. Συνεπώς, εφόσον ο προγραμματιστής προχωρήσει στη δική του υλοποίηση ενός πίνακα κατακερματισμού είτε θα πρέπει να γνωρίζει εκ των προτέρων το πλήθος των στοιχείων που πρόκειται να αποθηκευτούν είτε όταν αυτό απαιτηθεί να υπάρχει πρόβλεψη έτσι ώστε τα δεδομένα να μεταφέρονται σε μεγαλύτερο πίνακα κατακερματισμού.

Στις περισσότερες εφαρμογές υπάρχουν πολύ περισσότερα πιθανά κλειδιά εγγραφών από ότι θέσεις στο πίνακα κατακερματισμού. Αν για δύο ή περισσότερα κλειδιά η εφαρμογή της συνάρτησης κατακερματισμού επιστρέφει το ίδιο αποτέλεσμα τότε λέμε ότι συμβαίνει σύγκρουση (collision) η οποία θα πρέπει να διευθετηθεί με κάποιο τρόπο. Ο ακόλουθος κώδικας μετρά το πλήθος των συγκρούσεων που συμβαίνουν καθώς δημιουργούνται hashes για ένα σύνολο 2.000 κλειδιών αλφαριθμητικού τύπου.

```
1  #include <random>
2  using namespace std;
3
4  mt19937 mt(1821);
5  uniform_int_distribution<int> uni(0, 25);
6
7  string generate_random_string(int k) {
8    string s{};
9    const string letters_en = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
10   for (int i = 0; i < k; i++)
11     s += letters_en[uni(mt)];
12   return s;
13 }
```

Κώδικας 7.3: Δημιουργία τυχαίων λεκτικών (random_strings.cpp)

```
1  #include "hashes.cpp"
2  #include "random_strings.cpp"
3  #include <iostream>
4  #include <set>
5
6  using namespace std;
7  constexpr int HT_SIZE = 10001;
8
```

```
9  int main() {
10   set<int> aset;
11   int collisions = 0;
12   for (int i = 0; i < 2000; i++) {
13     string key = generate_random_string(10);
14     size_t h = hash0(key) % HT_SIZE; // 1863 collisions
15     // size_t h = hash1(key) % HT_SIZE; // 172 collisions
16     // size_t h = hash2(key) % HT_SIZE; // 188 collisions
17     // size_t h = hash3(key) % HT_SIZE; // 196 collisions
18     if (aset.find(h) != aset.end())
19       collisions++;
20     else
21       aset.insert(h);
22   }
23   cout << "number of collisions " << collisions << endl;
24 }
```

Κώδικας 7.4: Συγκρούσεις (hashes_ex2.cpp)

```
1 number of collisions 1863
```

Γενικότερα, σε έναν πίνακα κατακερματισμού, η εύρεση μιας εγγραφής με κλειδί key είναι μια διαδικασία δύο βημάτων:

- Εφαρμογή της συνάρτησης κατακερματισμού στο κλειδί της εγγραφής.
- Ξεκινώντας από την θέση που υποδεικνύει η συνάρτηση κατακερματισμού στον πίνακα κατακερματισμού, εντοπισμός της εγγραφής που περιέχει το ζητούμενο κλειδί (ενδεχόμενα θα χρειαστεί να εφαρμοστεί κάποιος μηχανισμός διευθέτησης συγκρούσεων).

Οι βασικοί μηχανισμοί επίλυσης των συγκρούσεων είναι η ανοικτή διευθυνσιοδότηση και ο κατακερματισμός με αλυσίδες.

7.2.1 Ανοικτή διευθυνσιοδότηση

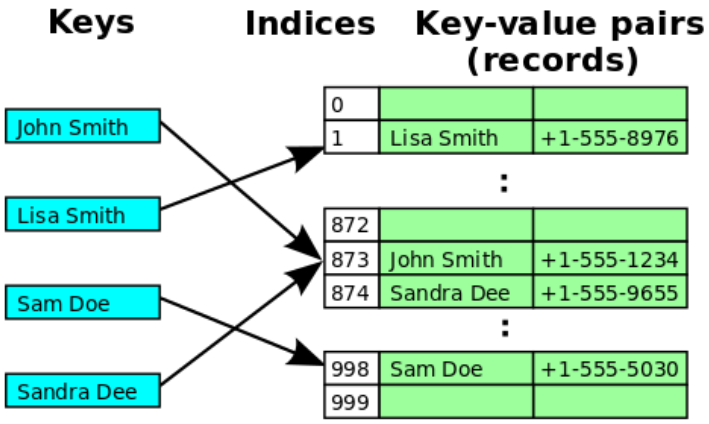
Στην ανοικτή διευθυνσιοδότηση (open addressing, closed hashing) όλα τα δεδομένα αποθηκεύονται απευθείας στον πίνακα κατακερματισμού. Αν συμβεί σύγκρουση τότε ελέγχεται αν κάποιο από τα υπόλοιπα κελιά είναι διαθέσιμο και η εγγραφή τοποθετείται εκεί. Συνεπώς, θα πρέπει το μέγεθος του hashtable να είναι μεγαλύτερο ή ίσο από το πλήθος των στοιχείων που πρόκειται να αποθηκευτούν σε αυτό. Θα πρέπει να σημειωθεί ότι η απόδοση της ανοικτής διευθυνσιοδότησης μειώνεται κατακόρυφα σε περίπτωση που το hashtable είναι σχεδόν γεμάτο.

Αν το πλήθος των κελιών είναι m και το πλήθος των εγγραφών είναι n τότε το πηλίκο $a = \frac{n}{m}$ που ονομάζεται παράγοντας φόρτωσης (load factor) καθορίζει σημαντικά την απόδοση του hashtable. Ο παράγοντας φόρτωσης στην περίπτωση της ανοικτής διευθυνσιοδότησης δεν μπορεί να είναι μεγαλύτερος της μονάδας.

Υπάρχουν πολλές παραλλαγές της ανοικτής διευθυνσιοδότησης που σχετίζονται με τον τρόπο που σε περίπτωση σύγκρουσης επιλέγεται το επόμενο κελί που εξετάζεται αν είναι ελεύθερο προκειμένου να τοποθετηθούν εκεί τα δεδομένα της εγγραφής. Αν εξετάζεται το αμέσως επόμενο στη σειρά κελί και μέχρι να βρεθεί το πρώτο διαθέσιμο, ξεκινώντας από την αρχή του πίνακα αν βρεθεί στο τέλος, τότε η μέθοδος ονομάζεται γραμμική ανίχνευση (linear probing). Άλλες διαδεδομένες μέθοδοι είναι η τετραγωνική ανίχνευση (quadratic probing) και ο διπλός κατακερματισμός (double hashing) [4].

Στη συνέχεια ακολουθεί μια υλοποίηση ενός πίνακα κατακερματισμού με ανοικτή διευθυνσιοδότηση και γραμμική ανίχνευση. Στον πίνακα κατακερματισμού τοποθετούνται εγγραφές με κλειδιά και τιμές αλφαριθμητικού τύπου.

```
1 #include <iostream>
2
3 using namespace std;
```



Σχήμα 7.2: Κατακερματισμός εγγραφών με ανοικτή διευθυνσιοδότηση και γραμμική ανίχνευση [1]

```
4
5  struct record {
6     string key;
7     string value;
8  };
9
10 class oa_hashtable {
11 private:
12    int capacity;
13    int size;
14    record **data; // array of pointers to records
15
16    size_t hash(string &key) {
17       size_t value = 0;
18       for (size_t i = 0; i < key.length(); i++)
19          value = 37 * value + key[i];
20       return value % capacity;
21    }
22
23 public:
24    oa_hashtable(int capacity) {
25       this->capacity = capacity;
26       size = 0;
27       data = new record *[capacity];
28       for (int i = 0; i < capacity; i++)
29          data[i] = nullptr;
30    }
31
32    ~oa_hashtable() {
33       for (size_t i = 0; i < capacity; i++)
34          if (data[i] != nullptr)
35             delete data[i];
36       delete[] data;
37    }
38
39    record *get(string &key) {
40       size_t hash_code = hash(key);
41       while (data[hash_code] != nullptr) {
42          if (data[hash_code]->key == key)
43             return data[hash_code];
```

```
44          hash_code = (hash_code + 1) % capacity;
45      }
46      return nullptr;
47    }
48
49    void put(record *arecord) {
50      if (size == capacity) {
51        cerr << "The hashtable is full" << endl;
52        return;
53      }
54      size_t hash_code = hash(arecord->key);
55      while (data[hash_code] != nullptr && data[hash_code]->key != "ERASED") {
56        if (data[hash_code]->key == arecord->key) {
57          delete data[hash_code];
58          data[hash_code] = arecord; // update existing key
59          return;
60        }
61        hash_code = (hash_code + 1) % capacity;
62      }
63      data[hash_code] = arecord;
64      size++;
65    }
66
67    void erase(string &key) {
68      size_t hash_code = hash(key);
69      while (data[hash_code] != nullptr) {
70        if (data[hash_code]->key == key) {
71          delete data[hash_code];
72          data[hash_code] = new record{"ERASED", "ERASED"}; // insert dummy record
73          size--;
74          return;
75        }
76        hash_code = (hash_code + 1) % capacity;
77      }
78    }
79
80    void print_all() {
81      for (int i = 0; i < capacity; i++)
82        if (data[i] != nullptr && data[i]->key != "ERASED")
83          cout << "#(" << i << ") " << data[i]->key << " " << data[i]->value
84               << endl;
85      cout << "Load factor: " << (double)size / (double)capacity << endl;
86    }
87  };
88
89  int main() {
90    oa_hashtable hashtable(101); // hashtable with maximum capacity 101 items
91    record *precord1 = new record{"John Smith", "+1-555-1234"};
92    record *precord2 = new record{"Lisa Smith", "+1-555-8976"};
93    record *precord3 = new record{"Sam Doe", "+1-555-5030"};
94    hashtable.put(precord1);
95    hashtable.put(precord2);
96    hashtable.put(precord3);
97    hashtable.print_all();
98    string key = "Sam Doe";
99    record *precord = hashtable.get(key);
100   if (precord == nullptr)
101     cout << "Key not found" << endl;
102   else {
103     cout << "Key found: " << precord->key << " " << precord->value << endl;
104     hashtable.erase(key);
```

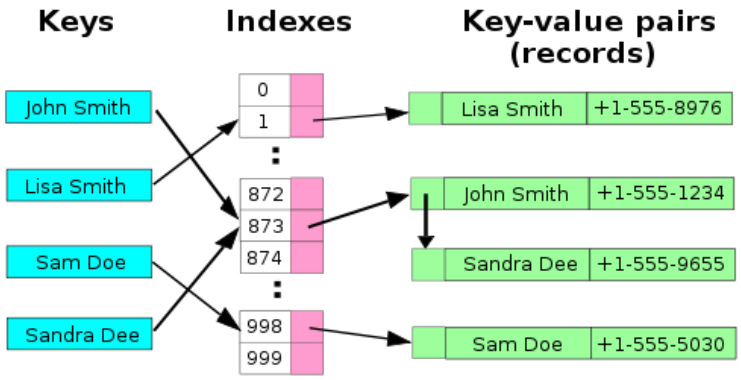
```
105   }
106   hashtable.print_all();
107 }
```

Κώδικας 7.5: Ανοικτή διευθυνσιοδότηση (open_addressing.cpp)

```
1 #(1) Sam Doe +1−555−5030
2 #(46) John Smith +1−555−1234
3 #(57) Lisa Smith +1−555−8976
4 Load factor: 0.029703
5 Key found: Sam Doe +1−555−5030
6 #(46) John Smith +1−555−1234
7 #(57) Lisa Smith +1−555−8976
8 Load factor: 0.019802
```

7.2.2 Κατακερματισμός με αλυσίδες

Στον κατακερματισμό με αλυσίδες (separate chaining) οι εγγραφές αποθηκεύονται σε συνδεδεμένες λίστες κάθε μια από τις οποίες είναι προσαρτημένες στα κελιά ενός hashtable. Συνεπώς, η απόδοση των αναζητήσεων εξαρτάται από τα μήκη των συνδεδεμένων λιστών. Αν η συνάρτηση κατακερματισμού κατανέμει τα n κλειδιά ανάμεσα στα m κελιά ομοιόμορφα τότε κάθε λίστα θα έχει μήκος $\frac{n}{m}$. Ο παράγοντας φόρτωσης, $a = \frac{n}{m}$, στον κατακερματισμό με αλυσίδες δεν θα πρέπει να απέχει πολύ από την μονάδα. Πολύ μικρό load factor σημαίνει ότι υπάρχουν πολλές κενές λίστες και συνεπώς δεν γίνεται αποδοτική χρήση του χώρου ενώ μεγάλο load factor σημαίνει μακριές συνδεδεμένες λίστες και μεγαλύτεροι χρόνοι αναζήτησης.



Σχήμα 7.3: Κατακερματισμός εγγραφών με αλυσίδες [1]

Στη συνέχεια ακολουθεί μια υλοποίηση ενός πίνακα κατακερματισμού με κατακερματισμό με αλυσίδες. Για τις συνδεδεμένες λίστες χρησιμοποιείται η λίστα std::list.

```
1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  struct record {
7    string key;
8    string value;
9  };
10
11 class sc_hashtable {
12 private:
13   int size;
14   list<record *> *buckets;
```

```
15
16    size_t hash(string &key) {
17      size_t value = 0;
18      for (size_t i = 0; i < key.length(); i++)
19        value = 37 * value + key[i];
20      return value % size;
21    }
22
23  public:
24    sc_hashtable(int size) {
25      this->size = size;
26      buckets = new list<record *>[size];
27    }
28
29    ~sc_hashtable() {
30      for (size_t i = 0; i < size; i++)
31        for (record *rec : buckets[i])
32          delete rec;
33      delete[] buckets;
34    }
35
36    record *get(string &key) {
37      size_t hash_code = hash(key);
38      if (buckets[hash_code].empty())
39        return nullptr;
40      else
41        for (record *rec : buckets[hash_code])
42          if (rec->key == key)
43            return rec;
44      return nullptr;
45    }
46
47    void put(record *arecord) {
48      size_t hash_code = hash(arecord->key);
49      buckets[hash_code].push_back(arecord);
50    }
51
52    void erase(string &key) {
53      size_t hash_code = hash(key);
54      list<record *>::iterator itr = buckets[hash_code].begin();
55      while (itr != buckets[hash_code].end())
56        if ((*itr)->key == key)
57          itr = buckets[hash_code].erase(itr);
58        else
59          ++itr;
60    }
61
62    void print_all() {
63      int m = 0;
64      for (size_t i = 0; i < size; i++)
65        if (!buckets[i].empty())
66          for (record *rec : buckets[i]) {
67            cout << "#(" << i << ") " << rec->key << " " << rec->value << endl;
68            m++;
69          }
70      cout << "Load factor: " << (double)m / (double)size << endl;
71    }
72  };
73
74  int main() {
75    sc_hashtable hashtable(101);
```

```
76    record *precord1 = new record{"John Smith", "+1-555-1234"};
77    record *precord2 = new record{"Lisa Smith", "+1-555-8976"};
78    record *precord3 = new record{"Sam Doe", "+1-555-5030"};
79    hashtable.put(precord1);
80    hashtable.put(precord2);
81    hashtable.put(precord3);
82    hashtable.print_all();
83    string key = "Sam Doe";
84    record *precord = hashtable.get(key);
85    if (precord == nullptr)
86       cout << "Key not found" << endl;
87    else {
88       cout << "Key found: " << precord->key << " " << precord->value << endl;
89       hashtable.erase(key);
90    }
91    hashtable.print_all();
92 }
```

Κώδικας 7.6: Κατακερματισμός με αλυσίδες (separate_chaining.cpp)

```
1 #(1) Sam Doe +1-555-5030
2 #(46) John Smith +1-555-1234
3 #(57) Lisa Smith +1-555-8976
4 Load factor: 0.029703
5 Key found: Sam Doe +1-555-5030
6 #(46) John Smith +1-555-1234
7 #(57) Lisa Smith +1-555-8976
8 Load factor: 0.019802
```

Περισσότερες πληροφορίες σχετικά με τον κατακερματισμό και την υλοποίηση πινάκων κατακερματισμού μπορούν να βρεθούν στις αναφορές [2], [3].

7.3 Κατακερματισμός με την STL

Η STL διαθέτει την κλάση std::hash που μπορεί να χρησιμοποιηθεί για την επιστροφή hash τιμών για διάφορους τύπους δεδομένων. Στον ακόλουθο κώδικα παρουσιάζεται η χρήση της std::hash.

```
1  #include <iostream>
2  #include <string>
3  using namespace std;
4
5  int main() {
6     constexpr int HT_SIZE = 101; // hypothetical hashtable size
7     double d1 = 1000.1;
8     double d2 = 1000.2;
9     hash<double> d_hash;
10    cout << "The hash value for: " << d1 << " is " << d_hash(d1) << " -> #"
11         << d_hash(d1) % HT_SIZE << endl;
12    cout << "The hash value for: " << d2 << " is " << d_hash(d2) << " -> #"
13         << d_hash(d2) % HT_SIZE << endl;
14
15    char c1[15] = "This is a test";
16    char c2[16] = "This is a test.";
17    hash<char *> c_strhash;
18    cout << "The hash value for: " << c1 << " is " << c_strhash(c1) << " -> #"
19         << c_strhash(c1) % HT_SIZE << endl;
20    cout << "The hash value for: " << c2 << " is " << c_strhash(c2) << " -> #"
21         << c_strhash(c2) % HT_SIZE << endl;
22
23    string s1 = "This is a test";
```

```
24    string s2 = "This is a test.";
25    hash<string> strhash;
26    cout << "The hash value for: " << s1 << " is " << strhash(s1) << " --> #"
27         << strhash(s1) % HT_SIZE << endl;
28    cout << "The hash value for: " << s2 << " is " << strhash(s2) << " --> #"
29         << strhash(s2) % HT_SIZE << endl;
30 }
```

Κώδικας 7.7: Παράδειγμα χρήσης της std::hash (stl_hash.cpp)

```
1 The hash value for: 1000.1 is 18248755989755706217 --> #44
2 The hash value for: 1000.2 is 2007414553616229599 --> #30
3 The hash value for: This is a test is 2293264 --> #59
4 The hash value for: This is a test. is 2293248 --> #43
5 The hash value for: This is a test is 5122661464562453635 --> #23
6 The hash value for: This is a test. is 10912006877877170250 --> #46
```

Επιπλέον, η STL υποστηρίζει δύο βασικές δομές κατακερματισμού το std::unordered_set και το std::unordered_map. Το std::unordered_set υλοποιείται ως ένας πίνακας κατακερματισμού και μπορεί να περιέχει τιμές (κλειδιά) οποιουδήποτε τύπου οι οποίες γίνονται hash σε διάφορες θέσεις του πίνακα κατακερματισμού. Κατά μέσο όρο, οι λειτουργίες σε ένα std::unordered_set (εύρεση, εισαγωγή και διαγραφή κλειδιού) πραγματοποιούνται σε σταθερό χρόνο O(1). Ένα std::unordered_set δεν περιέχει διπλότυπα, ενώ αν υπάρχει αυτή η ανάγκη τότε μπορεί να χρησιμοποιηθεί το std::unordered_multiset.

Στον κώδικα που ακολουθεί οι χαρακτήρες ενός λεκτικού εισάγονται ένας προς ένας σε ένα std::unordered_set έτσι ώστε να υπολογιστεί το πλήθος των διακριτών χαρακτήρων ενός λεκτικού.

```
1  #include <cctype> // tolower
2  #include <iostream>
3  #include <unordered_set>
4
5  using namespace std;
6
7  int main() {
8     string text = "You can do anything but not everything";
9     unordered_set<char> uset;
10    for (char c : text)
11       if (c != ' ')
12          uset.insert(tolower(c));
13    cout << "Number of discrete characters=" << uset.size() << endl;
14    for (unordered_set<char>::iterator itr = uset.begin(); itr != uset.end();
15          itr++)
16       cout << *itr << " ";
17    cout << endl;
18 }
```

Κώδικας 7.8: Παράδειγμα χρήσης του std::unordered_set (stl_unordered_set.cpp)

```
1 Number of discrete characters=15
2 r v e g c b y n u o d a t i h
```

Το std::unordered_map αποθηκεύει ζεύγη (κλειδί-τιμή). Το κλειδί αναγνωριζει με μοναδικό τρόπο το κάθε ζεύγος και γίνεται hash σε συγκεκριμένη θέση του πίνακα κατακερματισμού. Όπως και στο std::unordered_set. κατά μέσο όρο, οι λειτουργίες σε ένα std::unordered_map πραγματοποιούνται σε σταθερό χρόνο O(1). Η ανάθεση τιμής σε κλειδί μπορεί να γίνει με τους τελεστές = και [], ενώ το πέρασμα από τις τιμές ενός std::unordered_map μπορεί να γίνει με iterator ή με range for.

```
1 #include <iostream>
2 #include <unordered_map>
3
4 using namespace std;
```

```
5
6  int main() {
7    unordered_map<string, double> atomic_mass{{"H", 1.008}, // Hydrogen
8                                               {"C", 12.011}}; // Carbon
9    atomic_mass["O"] = 15.999; // Oxygen
10   atomic_mass["Fe"] = 55.845; // Iron
11   atomic_mass.insert(make_pair("Al", 26.982)); // Aluminium
12
13   for (unordered_map<string, double>::iterator itr = atomic_mass.begin();
14        itr != atomic_mass.end(); itr++)
15     cout << itr->first << ":" << itr->second << " ";
16   cout << endl;
17
18   for (const std::pair<string, double> &kv : atomic_mass)
19     cout << kv.first << ":" << kv.second << " ";
20   cout << endl;
21
22   string element = "Fe";
23   // string element = "Ti"; // Titanium
24   if (atomic_mass.find(element) == atomic_mass.end())
25     cout << "Element " << element << " is not in the map" << endl;
26   else
27     cout << "Element " << element << " has atomic mass " << atomic_mass[element]
28          << "" << endl;
29 }
```

Κώδικας 7.9: Παράδειγμα χρήσης του std::unordered_map (stl_unordered_map.cpp)

```
1 Al:26.982 H:1.008 C:12.011 O:15.999 Fe:55.845
2 Al:26.982 H:1.008 C:12.011 O:15.999 Fe:55.845
3 Element Fe has atomic mass 55.845
```

7.4 Παραδείγματα

7.4.1 Παράδειγμα 1

Έστω μια επιχείρηση η οποία επιθυμεί να αποθηκεύσει τα στοιχεία των υπαλλήλων της (όνομα, διεύθυνση) σε μια δομή έτσι ώστε με βάση το όνομα του υπαλλήλου να επιτυγχάνει τη γρήγορη ανάκληση των υπόλοιπων στοιχείων των υπαλλήλων. Στη συνέχεια παρουσιάζεται η υλοποίηση ενός πίνακα κατακερματισμού στον οποίο κλειδί θεωρείται το όνομα του υπαλλήλου και η επίλυση των συγκρούσεων πραγματοποιείται με ανοικτή διευθυνσιοδότηση (open addressing) και γραμμική ανίχνευση (linear probing). Καθώς δεν υπάρχει η ανάγκη διαγραφής τιμών από τον πίνακα κατακερματισμού παρουσιάζεται μια απλούστερη υλοποίηση σε σχέση με αυτή που παρουσιάστηκε στον κώδικα 7.5. Ο πίνακας κατακερματισμού μπορεί να δεχθεί το πολύ 100.000 εγγραφές υπαλλήλων. Στο παράδειγμα χρονομετρείται η εκτέλεση για 20.000, 30.000 και 80.000 υπαλλήλους. Παρατηρείται ότι λόγω των συγκρούσεων καθώς ο συντελεστής φόρτωσης του πίνακα κατακερματισμού αυξάνεται η απόδοση της δομής υποβαθμίζεται.

```
1 #include "hashes.cpp"
2 #include "random_strings.cpp"
3 #include <chrono>
4 #include <iomanip>
5 #include <iostream>
6 #include <string>
7 #include <vector>
8
9 using namespace std::chrono;
10
```

```
11 const int N = 100000; // HashTable size
12
13 struct employee {
14   string name;
15   string address;
16 };
17
18 void insert(employee hash_table[], employee &ypa) {
19   int pos = hash1(ypa.name) % N;
20   while (hash_table[pos].name != "") {
21     pos++;
22     pos %= N;
23   }
24   hash_table[pos] = ypa;
25 }
26
27 bool search(employee hash_table[], string &name, employee &ypa) {
28   int pos = hash1(name) % N;
29   int c = 0;
30   while (hash_table[pos].name != name) {
31     if (hash_table[pos].name == "")
32       return false;
33     pos++;
34     pos %= N;
35     c++;
36     if (c > N)
37       return false;
38   }
39   ypa = hash_table[pos];
40   return true;
41 }
42
43 int main() {
44   vector<int> SIZES{20000, 30000, 80000};
45   for (int x : SIZES) {
46     struct employee *hash_table = new struct employee[N];
47     // generate x random employees, insert them at the hashtable
48     vector<string> names;
49     for (int i = 0; i < x; i++) {
50       employee ypa;
51       ypa.name = generate_random_string(3);
52       ypa.address = generate_random_string(20);
53       insert(hash_table, ypa);
54       names.push_back(ypa.name);
55     }
56     // generate x more names
57     for (int i = 0; i < x; i++)
58       names.push_back(generate_random_string(3));
59     // time execution of 2*x searches in the HashTable
60     auto t1 = high_resolution_clock::now();
61     employee ypa;
62     int c = 0;
63     for (string name : names)
64       if (search(hash_table, name, ypa)) {
65         // cout << "Employee " << ypa.name << " " << ypa.address << endl;
66         c++;
67       }
68     auto t2 = high_resolution_clock::now();
69     std::chrono::duration<double, std::micro> duration = t2 - t1;
70     cout << "Load factor: " << setprecision(2) << (double)x / (double)N
71          << " employees found: " << c << ", employees not found: " << 2 * x - c
```

```
72            << " time elapsed: " << std::fixed << duration.count() / 1E6
73            << " seconds" << endl;
74      delete[] hash_table;
75    }
76  }
```

Κώδικας 7.10: Υλοποίηση πίνακα κατακερματισμού για γρήγορη αποθήκευση και αναζήτηση εγγραφών (lab07_ex1.cpp)

```
1  Load factor: 0.2 employees found: 33565, employees not found: 6435 time elapsed: 0.01 seconds
2  Load factor: 0.30 employees found: 54478, employees not found: 5522 time elapsed: 0.13 seconds
3  Load factor: 0.80 employees found: 159172, employees not found: 828 time elapsed: 12.50 seconds
```

7.4.2 Παράδειγμα 2

Στο παράδειγμα αυτό παρουσιάζεται η λύση του ίδιου προβλήματος με το παράδειγμα 1 με τη διαφορά ότι πλέον χρησιμοποιείται η δομή std::unordered_map της STL.

```
1  #include "random_strings.cpp"
2  #include <chrono>
3  #include <iomanip>
4  #include <iostream>
5  #include <string>
6  #include <unordered_map>
7  #include <vector>
8
9  using namespace std::chrono;
10
11 struct employee {
12   string name;
13   string address;
14 };
15
16 int main() {
17   vector<int> SIZES{20000, 30000, 80000};
18   for (int x : SIZES) {
19     unordered_map<string, employee> umap;
20     // generate x random employees, insert them at the hashtable
21     vector<string> names;
22     for (int i = 0; i < x; i++) {
23       employee ypa;
24       ypa.name = generate_random_string(3);
25       ypa.address = generate_random_string(20);
26       umap[ypa.name] = ypa;
27       names.push_back(ypa.name);
28     }
29     // generate x more names
30     for (int i = 0; i < x; i++)
31       names.push_back(generate_random_string(3));
32
33     // time execution of 2*x searches in the HashTable
34     auto t1 = high_resolution_clock::now();
35     int c = 0;
36     for (string name : names)
37       if (umap.find(name) != umap.end()) {
38         // cout << "Employee " << name << " " << umap[name].address << endl;
39         c++;
40       }
41     auto t2 = high_resolution_clock::now();
42     std::chrono::duration<double, std::micro> duration = t2 - t1;
```

```
43      cout << "Load factor: " << setprecision(2) << umap.load_factor()
44           << " employees found: " << c << ", employees not found: " << 2 * x - c
45           << " time elapsed: " << std::fixed << duration.count() / 1E6
46           << " seconds" << endl;
47    }
48  }
```

Κώδικας 7.11: Γρήγορη αποθήκευση και αναζήτηση εγγραφών με τη χρήση της std::unordered_map (lab07_ex2.cpp)

```
1  Load factor: 0.79 employees found: 33565, employees not found: 6435 time elapsed: 0.01 seconds
2  Load factor: 0.95 employees found: 54478, employees not found: 5522 time elapsed: 0.01 seconds
3  Load factor: 0.57 employees found: 159172, employees not found: 828 time elapsed: 0.02 seconds
```

7.4.3 Παράδειγμα 3

Στο παράδειγμα αυτό εξετάζονται τέσσερις διαφορετικοί τρόποι με τους οποίους ελέγχεται για ένα μεγάλο πλήθος τιμών (5.000.000) πόσες από αυτές περιέχονται σε ένα δεδομένο σύνολο 1.000 τιμών. Οι τιμές είναι ακέραιες και επιλέγονται με τυχαίο τρόπο στο διάστημα [0,100.000]. Ο χρόνος που απαιτεί η κάθε προσέγγιση χρονομετρείται.

- Η πρώτη προσέγγιση (scenario1) χρησιμοποιεί ένα vector για να αποθηκεύσει το σύνολο των 1.000 τυχαίων ακεραίων τιμών και αναζητά σειριακά κάθε τιμή στο vector.
- Η δεύτερη προσέγγιση (scenario2) χρησιμοποιεί επίσης ένα vector για να αποθηκεύσει το σύνολο των 1.000 τυχαίων ακεραίων τιμών, τις ταξινομεί και αναζητά κάθε τιμή στο ταξινομημένο vector.
- Η τρίτη προσέγγιση (scenario3) αποθηκεύει τις 1.000 τυχαίες ακεραίες τιμές σε ένα std::set (υλοποιείται στην STL ως δυαδικό δένδρο αναζήτησης) και αναζητά κάθε τιμή σε αυτό.
- Η τέταρτη προσέγγιση (scenario4) αποθηκεύει τις 1.000 τυχαίες ακεραίες τιμές σε ένα std::unordered_set (υλοποιείται στην STL ως πίνακας κατακερματισμού) και αναζητά κάθε τιμή σε αυτό.

```
1  #include <algorithm>
2  #include <bitset>
3  #include <chrono>
4  #include <iostream>
5  #include <random>
6  #include <set>
7  #include <unordered_set>
8  #include <vector>
9
10 using namespace std;
11 using namespace std::chrono;
12
13 // number of items in the set
14 constexpr int N = 1000;
15 // number of values checked whether they exist in the set
16 constexpr int M = 5E6;
17
18 uniform_int_distribution<uint32_t> dist(0, 1E5);
19
20 void scenario1(vector<uint32_t> &avector) {
21   long seed = 1940;
22   mt19937 mt(seed);
23   int c = 0;
24   for (int i = 0; i < M; i++)
25     if (find(avector.begin(), avector.end(), dist(mt)) != avector.end())
26       c++;
27   cout << "Values in the set (using unsorted vector): " << c << " ";
```

```
28 }
29
30 void scenario2(vector<uint32_t> &avector) {
31   sort(avector.begin(), avector.end());
32   long seed = 1940;
33   mt19937 mt(seed);
34   int c = 0;
35   for (int i = 0; i < M; i++)
36     if (binary_search(avector.begin(), avector.end(), dist(mt)))
37       c++;
38   cout << "Values in the set (using sorted vector): " << c << " ";
39 }
40
41 void scenario3(set<uint32_t> &aset) {
42   long seed = 1940;
43   mt19937 mt(seed);
44   int c = 0;
45   for (int i = 0; i < M; i++)
46     if (aset.find(dist(mt)) != aset.end())
47       c++;
48   cout << "Values in the set (using std::set): " << c << " ";
49 }
50
51 void scenario4(unordered_set<uint32_t> &auset) {
52   long seed = 1940;
53   mt19937 mt(seed);
54   int c = 0;
55   for (int i = 0; i < M; i++)
56     if (auset.find(dist(mt)) != auset.end())
57       c++;
58   cout << "Values in the set (using std::unordered_set): " << c << " ";
59 }
60
61 int main() {
62   long seed = 1821;
63   mt19937 mt(seed);
64   high_resolution_clock::time_point t1, t2;
65   duration<double, std::micro> duration_micro;
66   vector<uint32_t> avector(N);
67   // fill vector with random values using std::generate and lambda function
68   std::generate(avector.begin(), avector.end(), [&mt]() { return dist(mt); });
69
70   t1 = high_resolution_clock::now();
71   scenario1(avector);
72   t2 = high_resolution_clock::now();
73   duration_micro = t2 - t1;
74   cout << "elapsed time: " << duration_micro.count() / 1E6 << " seconds"
75        << endl;
76
77   t1 = high_resolution_clock::now();
78   scenario2(avector);
79   t2 = high_resolution_clock::now();
80   duration_micro = t2 - t1;
81   cout << "elapsed time: " << duration_micro.count() / 1E6 << " seconds"
82        << endl;
83
84   set<uint32_t> aset(avector.begin(), avector.end());
85   t1 = high_resolution_clock::now();
86   scenario3(aset);
87   t2 = high_resolution_clock::now();
88   duration_micro = t2 - t1;
```

```
89    cout << "elapsed time: " << duration_micro.count() / 1E6 << " seconds"
90         << endl;
91
92    unordered_set<uint32_t> auset(avector.begin(), avector.end());
93    t1 = high_resolution_clock::now();
94    scenario4(auset);
95    t2 = high_resolution_clock::now();
96    duration_micro = t2 - t1;
97    cout << "elapsed time: " << duration_micro.count() / 1E6 << " seconds"
98         << endl;
99 }
```

Κώδικας 7.12: Έλεγχος ύπαρξης τιμών σε ένα σύνολο τιμών (lab07_ex3.cpp)

```
1 Values in the set (using unsorted vector): 49807 elapsed time: 34.8646 seconds
2 Values in the set (using sorted vector): 49807 elapsed time: 1.7819 seconds
3 Values in the set (using std::set): 49807 elapsed time: 1.7591 seconds
4 Values in the set (using std::unordered_set): 49807 elapsed time: 0.921053 seconds
```

7.5 Ασκήσεις

1. Γράψτε μια συνάρτηση που να δέχεται έναν πίνακα ακεραίων A και έναν ακέραιο αριθμό sum και να βρίσκει το πλήθος από όλα τα ζεύγη τιμών του A που το άθροισμά τους είναι ίσο με sum.

2. Γράψτε ένα πρόγραμμα που για ένα λεκτικό που θα δέχεται ως είσοδο, να επιστρέφει το χαρακτήρα (γράμματα κεφαλαία, γράμματα πεζά, ψηφία, σύμβολα) που εμφανίζεται περισσότερες φορές καθώς και πόσες φορές εμφανίζεται στο λεκτικό.

3. Γράψτε μια συνάρτηση που να δέχεται έναν πίνακα ακεραίων A και έναν ακέραιο αριθμό K και να βρίσκει τη μεγαλύτερη σε μήκος υποακολουθία στοιχείων του A που έχει άθροισμα ίσο με K.

4. Γράψτε ένα πρόγραμμα που να δέχεται μια λέξη και να βρίσκει γρήγορα όλες τις άλλες έγκυρες λέξεις που είναι αναγραμματισμοί της λέξης που δόθηκε. Θεωρείστε ότι έχετε δεδομένο ένα αρχείο κειμένου με όλες τις έγκυρες λέξεις (words.txt), μια ανά γραμμή.

Βιβλιογραφία

[1] Wikibooks, Data Structures - Hash Tables, https://en.wikibooks.org/wiki/Data_Structures/Hash_Tables

[2] C++ tutorial: Intro to Hash Tables, https://pumpkinprogrammerdotcom4.wordpress.com/2014/06/21/c-tutorial-intro-to-hash-tables/

[3] HackerEarth, Basics of Hash Tables, https://www.hackerearth.com/practice/data-structures/hash-tables/basics-of-hash-tables/tutorial/

[4] VisualAlgo.net Open Addressing (LP, QP, DH) and Separate Chaining Visualization, https://visualgo.net/en/hashtable

Εργαστήριο 8

Γραφήματα

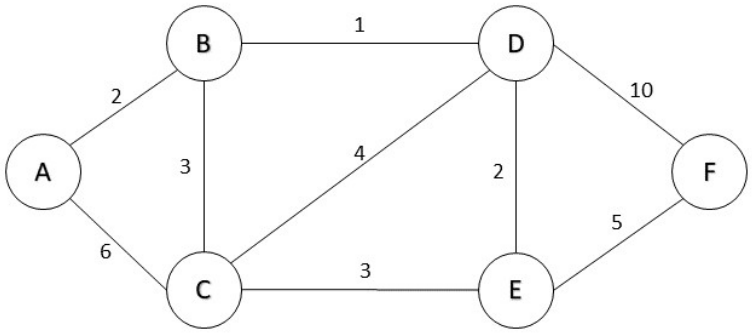
8.1 Εισαγωγή

Τα γραφήματα είναι δομές δεδομένων που συναντώνται συχνά κατά την επίλυση προβλημάτων. Η ευχέρεια προγραμματισμού αλγορίθμων που εφαρμόζονται πάνω σε γραφήματα είναι ουσιώδης. Καθώς μάλιστα συχνά ανακύπτουν προβλήματα για τα οποία έχουν διατυπωθεί αλγόριθμοι αποδοτικής επίλυσής τους η γνώση των αλγορίθμων αυτών αποδεικνύεται ισχυρός σύμμαχος στην επίλυση δύσκολων προβλημάτων.

8.2 Γραφήματα

Ένα γράφημα ή γράφος (graph) είναι ένα σύνολο από σημεία που ονομάζονται κορυφές (vertices) ή κόμβοι (nodes) για τα οποία ισχύει ότι κάποια από αυτά είναι συνδεδεμένα απευθείας μεταξύ τους με τμήματα γραμμών που ονομάζονται ακμές (edges ή arcs). Συνήθως ένα γράφημα συμβολίζεται ως $G = (V, E)$ όπου V είναι το σύνολο των κορυφών και E είναι το σύνολο των ακμών.

Αν οι ακμές δεν έχουν κατεύθυνση τότε το γράφημα ονομάζεται μη κατευθυνόμενο (undirected) ενώ σε άλλη περίπτωση ονομάζεται κατευθυνόμενο (directed). Ένα πλήρες γράφημα (που όλες οι κορυφές συνδέονται απευθείας με όλες τις άλλες κορυφές) έχει $\frac{|V||V-1|}{2}$ ακμές ($|V|$ είναι το πλήθος των κορυφών του γραφήματος). Αν σε κάθε ακμή αντιστοιχεί μια τιμή τότε το γράφημα λέγεται γράφημα με βάρη. Το γράφημα του σχήματος 8.1 είναι ένα μη κατευθυνόμενο γράφημα με βάρη.



Σχήμα 8.1: Ένα μη κατευθυνόμενο γράφημα 6 κορυφών και 9 ακμών με βάρη στις ακμές του

Ένα γράφημα λέγεται συνεκτικό αν για δύο οποιεσδήποτε κορυφές του υπάρχει μονοπάτι που τις συνδέει. Αν ένα γράφημα δεν είναι συνεκτικό τότε αποτελείται από επιμέρους συνεκτικά γραφήματα τα οποία λέγονται συνιστώσες. Είναι προφανές ότι ένα συνεκτικό γράφημα έχει μόνο μια συνιστώσα.

8.2.1 Αναπαράσταση γραφημάτων

Δύο διαδεδομένοι τρόποι αναπαράστασης γραφημάτων είναι οι πίνακες γειτνίασης (adjacency matrices) και οι λίστες γειτνίασης (adjacency lists).

Στους πίνακες γειτνίασης διατηρείται ένας δισδιάστατος πίνακας $n \times n$ όπου n είναι το πλήθος των κορυφών του γραφήματος. Για κάθε ακμή του γραφήματος που συνενώνει την κορυφή i με την κορυφή j εισάγεται στη θέση i, j του πίνακα το βάρος της ακμής αν το γράφημα είναι με βάρη ενώ αν δεν υπάρχουν βάρη τότε εισάγεται η τιμή 1. Όλα τα υπόλοιπα στοιχεία του πίνακα λαμβάνουν την τιμή 0. Για παράδειγμα η πληροφορία του γραφήματος για το σχήμα 8.1 διατηρείται όπως φαίνεται στον πίνακα 8.1.

	A	B	C	D	E	F
A	0	2	6	0	0	0
B	2	0	3	1	0	0
C	6	3	0	4	3	0
D	0	1	4	0	2	10
E	0	0	3	2	0	5
F	0	0	0	10	5	0

Πίνακας 8.1: Πίνακας γειτνίασης για το σχήμα 8.1

Στις λίστες γειτνίασης διατηρούνται λίστες που περιέχουν για κάθε κορυφή όλη την πληροφορία των συνδέσεών της με τους γειτονικούς της κόμβους. Για παράδειγμα το γράφημα του σχήματος 8.1 μπορεί να αναπαρασταθεί με τις ακόλουθες 6 λίστες (μια ανά κορυφή). Κάθε στοιχείο της λίστας για την κορυφή v είναι ένα ζεύγος τιμών (w, u) και αναπαριστά μια ακμή από την κορυφή v στην κορυφή u με βάρος w, όπως φαίνεται στο πίνακα 8.2.

A	(2,B), (6,C)
B	(2,A), (3,C), (1,D)
C	(6,A), (3,B), (4,D), (3,E)
D	(1,B), (4,C), (2,E), (10,F)
E	(3,C), (2,D), (5,F)
F	(10,D), (5,E)

Πίνακας 8.2: Λίστα γειτνίασης για το σχήμα 8.1

Περισσότερα για τις αναπαραστάσεις γραφημάτων μπορούν να βρεθούν στις αναφορές [1] και [2].

8.2.2 Ανάγνωση δεδομένων γραφήματος από αρχείο

Υπάρχουν πολλοί τρόποι με τους οποίους μπορούν να βρίσκονται καταγεγραμμένα τα δεδομένα ενός γραφήματος σε ένα αρχείο. Το αρχείο αυτό θα πρέπει να διαβαστεί έτσι ώστε να αναπαρασταθεί το γράφημα στη μνήμη του υπολογιστή. Στη συνέχεια παρουσιάζεται μια απλή μορφή αποτύπωσης κατευθυνόμενων με βάρη γραφημάτων χρησιμοποιώντας αρχεία απλού κειμένου. Σύμφωνα με αυτή τη μορφή για κάθε κορυφή του γραφήματος καταγράφεται σε ξεχωριστή γραμμή του αρχείου κειμένου το όνομά της ακολουθούμενο από ζεύγη τιμών, χωρισμένων με κόμματα, που αντιστοιχούν στις ακμές που ξεκινούν από τη συγκεκριμένη κορυφή. Στο κείμενο που ακολουθεί (graph1.txt) και το οποίο αφορά το γράφημα του σχήματος 8.1 η πρώτη γραμμή σημαίνει ότι η κορυφή A συνδέεται με μια ακμή με βάρος 2 με την κορυφή B καθώς και με μια ακμή με βάρος 6 με την κορυφή C. Ανάλογα καταγράφεται η πληροφορία ακμών και για τις άλλες κορυφές.

```
1 A 2,B 6,C
2 B 2,A 3,C 1,D
3 C 6,A 3,B 4,D 3,E
4 D 1,B 4,C 2,E 10,F
```

```
E 3,C 2,D 5,F
F 10,D 5,E
```

Η ανάγνωση του αρχείου και η αναπαράσταση του γραφήματος ως λίστα γειτνίασης γίνεται με τη συνάρτηση read_data που δίνεται στη συνέχεια όπου fn είναι το όνομα του αρχείου. Η συνάρτηση αυτή δημιουργεί ένα λεξικό (map) που αποτελείται από εγγραφές τύπου key-value. Σε κάθε εγγραφή το key είναι ένα λεκτικό με το όνομα μιας κορυφής ενώ το value είναι ένα διάνυσμα (vector) που περιέχει ζεύγη (pair<int,string>) στα οποία το πρώτο στοιχείο είναι ένας ακέραιος αριθμός που αναπαριστά το βάρος μιας ακμής ενώ το δεύτερο ένα λεκτικό με το όνομα της κορυφής στην οποία καταλήγει η ακμή από την κορυφή key. Ο κώδικας έχει "σπάσει" σε 3 αρχεία (graph.hpp, graph.cpp και graph_ex1.cpp) έτσι ώστε να είναι ευκολότερη η επαναχρησιμοποίηση του. Η συνάρτηση print_data εμφανίζει τα δεδομένα του γραφήματος.

```
#include <fstream>
#include <iostream>
#include <map>
#include <sstream>
#include <utility>
#include <vector>

using namespace std;

map<string, vector<pair<int, string>>> read_data(string fn);
void print_graph(map<string, vector<pair<int, string>>> &g);
```

Κώδικας 8.1: header file με τις συναρτήσεις για ανάγνωση και εμφάνιση γραφημάτων (graph.hpp)

```
#include "graph.hpp"

using namespace std;

map<string, vector<pair<int, string>>> read_data(string fn) {
  map<string, vector<pair<int, string>>> graph;
  fstream filestr;
  string buffer;
  filestr.open(fn.c_str());
  if (filestr.is_open())
    while (getline(filestr, buffer)) {
      string buffer2;
      stringstream ss;
      ss.str(buffer);
      vector<string> tokens;
      while (ss >> buffer2)
        tokens.push_back(buffer2);
      string vertex1 = tokens[0].c_str();
      for (size_t i = 1; i < tokens.size(); i++) {
        int pos = tokens[i].find(",");
        int weight = atoi(tokens[i].substr(0, pos).c_str());
        string vertex2 =
            tokens[i].substr(pos + 1, tokens[i].length() - 1).c_str();
        graph[vertex1].push_back(make_pair(weight, vertex2));
      }
    }
  else {
    cout << "Error opening file: " << fn << endl;
    exit(-1);
  }
  return graph;
}

void print_graph(map<string, vector<pair<int, string>>> &g) {
```

```
35    for (const auto &p1 : g) {
36      for (const auto &p2 : p1.second)
37        cout << p1.first << "<--" << p2.first << "-->" << p2.second << " ";
38      cout << endl;
39    }
40  }
```

Κώδικας 8.2: source file με τις συναρτήσεις για ανάγνωση και εμφάνιση γραφημάτων (graph.cpp)

```
1  #include "graph.hpp"
2
3  using namespace std;
4
5  int main() {
6    map<string, vector<pair<int, string>>> graph = read_data("graph1.txt");
7    print_graph(graph);
8    return 0;
9  }
```

Κώδικας 8.3: Ανάγνωση και εκτύπωση των δεδομένων του γραφήματος του σχήματος 8.1 (graph_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

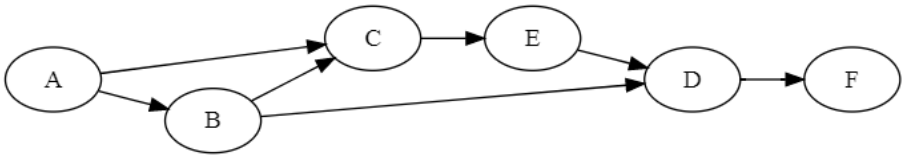
```
1  $ g++ -Wall -std=c++11 graph.cpp graph_ex1.cpp -o graph_ex1
2  $ ./graph_ex1
```

Η δε έξοδος που παράγεται είναι η ακόλουθη:

```
1  A<--2-->B A<--6-->C
2  B<--2-->A B<--3-->C B<--1-->D
3  C<--6-->A C<--3-->B C<--4-->D C<--3-->E
4  D<--1-->B D<--4-->C D<--2-->E D<--10-->F
5  E<--3-->C E<--2-->D E<--5-->F
6  F<--10-->D F<--5-->E
```

8.2.3 Κατευθυνόμενα ακυκλικά γραφήματα

Τα κατευθυνόμενα ακυκλικά γραφήματα (Directed Acyclic Graphs=DAGs) είναι γραφήματα για τα οποία δεν μπορεί να εντοπιστεί διαδρομή από μια κορυφή προς την ίδια. Στο σχήμα 8.2 παρουσιάζεται ένα γράφημα το οποίο δεν παρουσιάζει κύκλους. Αν για παράδειγμα υπήρχε μια ακόμα ακμή από την κορυφή E προς την κορυφή A τότε πλέον το γράφημα δεν θα ήταν DAG καθώς θα υπήρχε ο κύκλος A-C-E-A.



Σχήμα 8.2: Ένα κατευθυνόμενο ακυκλικό γράφημα (DAG)

Τα DAGs χρησιμοποιούνται στη μοντελοποίηση πολλών καταστάσεων. Μπορούν για παράδειγμα να αναπαραστήσουν εργασίες που πρέπει να εκτελεστούν και για τις οποίες υπάρχουν εξαρτήσεις όπως για παράδειγμα ότι για να ξεκινήσει η εκτέλεση της εργασίας D θα πρέπει πρώτα να έχουν ολοκληρωθεί οι εργασίες B και E.

8.2.4 Σημαντικοί αλγόριθμοι γραφημάτων

Υπάρχουν πολλοί αλγόριθμοι που εφαρμόζονται σε γραφήματα προκειμένου να επιλύσουν ενδιαφέροντα προβλήματα που ανακύπτουν σε πρακτικές εφαρμογές. Οι ακόλουθοι αλγόριθμοι είναι μερικοί από αυτούς:

- Αναζήτηση συντομότερων διαδρομών από μια κορυφή προς όλες τις άλλες κορυφές (Dijkstra). Ο αλγόριθμος αυτός θα αναλυθεί στη συνέχεια.
- Εύρεση μήκους συντομότερων διαδρομών για όλα τα ζεύγη κορυφών (Floyd Warshall) [3].
- Αναζήτηση κατά βάθος (Depth First Search). Είναι αλγόριθμος διάσχισης γραφήματος ο οποίος ξεκινά από έναν κόμβο αφετηρία και επισκέπτεται όλους τους άλλους κόμβους που είναι προσβάσιμοι χρησιμοποιώντας της ακμές του γραφήματος. Λειτουργεί επεκτείνοντας μια διαδρομή όσο βρίσκει νέους κόμβους τους οποίους μπορεί να επισκεφθεί. Αν δεν βρίσκει νέους κόμβους οπισθοδρομεί και διερευνά άλλα τμήματα του γραφήματος.
- Αναζήτηση κατά πλάτος (Breadth First Search). Αλγόριθμος διάσχισης γραφήματος που ξεκινώντας από έναν κόμβο αφετηρία επισκέπτεται τους υπόλοιπους κόμβους σε αύξουσα σειρά βημάτων από την αφετηρία. Βήματα θεωρούνται οι μεταβάσεις από κορυφή σε κορυφή.
- Εντοπισμός ελάχιστου συνεκτικού (ή γεννητικού) δένδρου (Prim [4], Kruskal [5]). Δεδομένου ενός γραφήματος, το πρόβλημα αφορά την εύρεση ενός δένδρου στο οποίο θα περιέχονται όλες οι κορυφές του γραφήματος ενώ οι ακμές του δένδρου θα είναι ένα υποσύνολο των ακμών του γραφήματος τέτοιο ώστε το άθροισμα των βαρών τους να είναι το ελάχιστο δυνατό.
- Τοπολογική ταξινόμηση (Topological Sort) [6]. Ο αλγόριθμος τοπολογικής ταξινόμησης εφαρμόζεται σε DAGs και παράγει μια σειρά κορυφών του γραφήματος για την οποία ισχύει ότι για κάθε κατευθυνόμενη ακμή από την κορυφή u στην κορυφή v στη σειρά των κορυφών η κορυφή u προηγείται της κορυφής v. Για παράδειγμα, για το DAG του σχήματος 8.2 αποτέλεσμα του αλγορίθμου είναι το A,B,C,E,D,F. Σε συνθετότερα γραφήματα μπορεί να υπάρχουν περισσότερες από μια τοπολογικές σειρές κορυφών για το γράφημα.
- Εντοπισμός κυκλωμάτων Euler (Eulerian circuit) [7]. Σε ένα γράφημα, διαδρομή Euler (Eulerian path) είναι μια διαδρομή που περνά από όλες τις ακμές του γραφήματος. Αν η διαδρομή αυτή ξεκινά και τερματίζει στην ίδια κορυφή τότε λέγεται κύκλωμα Euler.
- Εντοπισμός ισχυρά συνδεδεμένων συνιστωσών (Strongly Connected Components) [8]. Ισχυρά συνδεδεμένες συνιστώσες υφίστανται μόνο σε κατευθυνόμενα γραφήματα. Ένα κατευθυνόμενο γράφημα είναι ισχυρά συνδεδεμένο όταν υπάρχει διαδρομή από κάθε κορυφή προς κάθε άλλη κορυφή. Ένα κατευθυνόμενο γράφημα μπορεί να σπάσει σε ισχυρά συνδεδεμένα υπογραφήματα. Τα υπογραφήματα αυτά αποτελούν τις ισχυρά συνδεδεμένες συνιστώσες του γραφήματος.

8.3 Αλγόριθμος του Dijkstra για εύρεση συντομότερων διαδρομών

Ο αλγόριθμος δέχεται ως είσοδο ένα γράφημα $G = (V, E)$ και μια κορυφή του γραφήματος s η οποία αποτελεί την αφετηρία. Υπολογίζει για όλες τις κορυφές $v \in V$ το μήκος του συντομότερου μονοπατιού από την κορυφή s στην κορυφή v. Για να λειτουργήσει σωστά θα πρέπει κάθε ακμή να έχει μη αρνητικό βάρος. Αν το γράφημα περιέχει ακμές με αρνητικό βάρος τότε μπορεί να χρησιμοποιηθεί ο αλγόριθμος των Bellman-Ford [9].

8.3.1 Περιγραφή του αλγορίθμου

Ο αλγόριθμος εντοπίζει τις συντομότερες διαδρομές προς τις κορυφές του γραφήματος σε σειρά απόστασης από την κορυφή αφετηρία. Σε κάθε βήμα του αλγορίθμου η αφετηρία και οι ακμές προς τις κορυφές για τις οποίες έχει ήδη βρεθεί συντομότερο μονοπάτι σχηματίζουν το υποδένδρο S του γραφήματος. Οι κορυφές που είναι προσπελάσιμες με 1 ακμή από το υποδένδρο S είναι υποψήφιες να αποτελέσουν την επόμενη κορυφή που θα εισέλθει στο υποδένδρο. Επιλέγεται μεταξύ τους η κορυφή που βρίσκεται στη μικρότερη απόσταση από την αφετηρία. Για κάθε υποψήφια κορυφή u υπολογίζεται το άθροισμα της απόστασής της από την πλησιέστερη κορυφή v του δένδρου συν το μήκος της συντομότερης διαδρομής από την αφετηρία s προς την κορυφή v. Στη συνέχεια επιλέγεται η κορυφή με το μικρότερο άθροισμα και προσαρτάται στο σύνολο των κορυφών που απαρτίζουν το υποδένδρο S. Για κάθε μία από τις υποψήφιες κορυφές που συνδέονται με μια ακμή με την

κορυφή που επιλέχθηκε ενημερώνεται η απόστασή της από το υποδένδρο εφόσον προκύψει μικρότερη τιμή.

Ψευδοκώδικας Το σύνολο S περιέχει τις κορυφές για τις οποίες έχει προσδιοριστεί η συντομότερη διαδρομή από την κορυφή s ενώ το διάνυσμα d περιέχει τις αποστάσεις από την κορυφή s
1. Αρχικά $S = s$, $d_s = 0$ και για όλες τις κορυφές $i \neq s, d_i = \infty$
2. Μέχρι να γίνει $S = V$
3. Εντοπισμός του στοιχείου $v \notin S$ με τη μικρότερη τιμή d_v και προσθήκη του στο S
4. Για κάθε ακμή από την κορυφή v στην κορυφή u με βάρος w ενημερώνεται η τιμή d_u έτσι ώστε:
$$d_u = min(d_u, d_v + w)$$
5. Επιστροφή στο βήμα 2.

Εκτέλεση του αλγορίθμου Στη συνέχεια ακολουθεί παράδειγμα εκτέλεσης του αλγορίθμου για το γράφημα του σχήματος 8.1.

$S = \{A\}, d_A = 0, d_B = 2, d_C = 6, d_D = \infty, d_E = \infty, d_F = \infty$	Από το S μπορούμε να φτάσουμε στις κορυφές B και C με μήκος διαδρομής 2 και 6 αντίστοιχα. Επιλέγεται η κορυφή B.
$S = \{A, B\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = \infty, d_F = \infty$	Από το S μπορούμε να φτάσουμε στις κορυφές C και D με μήκος διαδρομής 5 και 3 αντίστοιχα. Επιλέγεται η κορυφή D.
$S = \{A, B, D\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 13$	Από το S μπορούμε να φτάσουμε στις κορυφές C, E και F με μήκος διαδρομής 5, 5 και 13 αντίστοιχα. Επιλέγεται (με τυχαίο τρόπο) ανάμεσα στις κορυφές C και E η κορυφή C.
$S = \{A, B, D, C\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 13$	Από το S μπορούμε να φτάσουμε στις κορυφές E και F με μήκος διαδρομής 5 και 13 αντίστοιχα. Επιλέγεται η κορυφή E.
$S = \{A, B, D, C, E\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 10$	Η μοναδική κορυφή στην οποία μένει να φτάσουμε από το S είναι η κορυφή F και το μήκος της συντομότερης διαδρομής από την A στην F είναι 10.
$S = \{A, B, D, C, E, F\}, d_A = 0, d_B = 2, d_C = 5, d_D = 3, d_E = 5, d_F = 10$	

Πίνακας 8.3: Αναλυτική εκτέλεση του αλγορίθμου

Συνεπώς ισχύει ότι:

- Για την κορυφή A η διαδρομή αποτελείται μόνο από τον κόμβο A και έχει μήκος 0.
- Για την κορυφή B η διαδρομή είναι η A-B με μήκος 2.
- Για την κορυφή C η διαδρομή είναι η A-B-C με μήκος 5.
- Για την κορυφή D η διαδρομή είναι η A-B-D με μήκος 3.
- Για την κορυφή E η διαδρομή είναι η A-B-D-E με μήκος 5.
- Για την κορυφή F η διαδρομή είναι η A-B-D-E-F με μήκος 10.

Σύνολο S	A	B	C	D	E	F
$\{\}$	0	∞	∞	∞	∞	∞
$\{A\}$	0	2_A	6_A	∞	∞	∞
$\{A, B\}$	0	2_A	5_B	3_B	∞	∞
$\{A, B, D\}$	0	2_A	5_B	3_B	5_D	13_D
$\{A, B, D, C\}$	0	2_A	5_B	3_B	5_D	13_D
$\{A, B, D, C, E\}$	0	2_A	5_B	3_B	5_D	10_E
$\{A, B, D, C, E, F\}$	0	2_A	5_B	3_B	5_D	10_E

Πίνακας 8.4: Συνοπτική εκτέλεση του αλγορίθμου

Στο σύνδεσμο της αναφοράς [10] μπορεί κανείς να παρακολουθήσει την εκτέλεση του αλγορίθμου για διάφορα γραφήματα.

Απόδοση του αλγορίθμου Η ταχύτητα εκτέλεσης του αλγορίθμου εξαρτάται από τις δομές δεδομένων που χρησιμοποιούνται για να αναπαρασταθεί το γράφημα. Γενικά, πρόκειται για έναν εξαιρετικά γρήγορο αλγόριθμο με πολυπλοκότητα χειρότερης περίπτωσης $O(|E|log|V|)$, όπου $|E|$ είναι ο αριθμός των ακμών και $|V|$ ο αριθμός των κορυφών του γραφήματος.

8.3.2 Κωδικοποίηση του αλγορίθμου

```
#include <climits>
#include <map>
#include <set>
#include <string>
#include <vector>

using namespace std;

struct path_info {
   string path;
   int cost;
};

void compute_shortest_paths_to_all_vertices(
      map<string, vector<pair<int, string>>> &graph, string source,
      map<string, path_info> &shortest_path_distances);
```

Κώδικας 8.4: header file για τον αλγόριθμο του Dijkstra (dijkstra.hpp)

```
#include "dijkstra.hpp"

using namespace std;

void compute_shortest_paths_to_all_vertices(
      map<string, vector<pair<int, string>>> &graph, string source,
      map<string, path_info> &shortest_path_distances) {
   vector<string> S{source};
   set<string> NS;
   for (auto &kv : graph) {
      string path = "";
      if (kv.first == source) {
         path += source;
         shortest_path_distances[kv.first] = {path, 0};
      } else {
```

```
16        NS.insert(kv.first);
17        shortest_path_distances[kv.first] = {path, INT_MAX};
18      }
19    }
20
21    while (!NS.empty()) {
22      string v1 = S.back();
23      for (pair<int, string> w_v : graph[v1]) {
24        int weight = w_v.first;
25        string v2 = w_v.second;
26        if (NS.find(v2) != NS.end())
27          if (shortest_path_distances[v1].cost + weight <
28              shortest_path_distances[v2].cost) {
29            shortest_path_distances[v2].path =
30                shortest_path_distances[v1].path + " " + v2;
31            shortest_path_distances[v2].cost =
32                shortest_path_distances[v1].cost + weight;
33          }
34      }
35      int min = INT_MAX;
36      string pmin = "None";
37      for (string v2 : NS) {
38        if (shortest_path_distances[v2].cost < min) {
39          min = shortest_path_distances[v2].cost;
40          pmin = v2;
41        }
42      }
43      // in case the graph is not connected
44      if (pmin == "None")
45        break;
46      S.push_back(pmin);
47      NS.erase(pmin);
48    }
49  }
```

Κώδικας 8.5: source file για τον αλγόριθμο του Dijkstra (dijkstra.cpp)

```
1  #include "dijkstra.hpp"
2  #include "graph.hpp"
3
4  using namespace std;
5
6  int main() {
7    map<string, vector<pair<int, string>>> graph = read_data("graph1.txt");
8    map<string, path_info> shortest_path_distances;
9    string source = "A";
10   compute_shortest_paths_to_all_vertices(graph, source,
11                                          shortest_path_distances);
12   for (auto p : shortest_path_distances) {
13     cout << "Shortest path from vertex " << source << " to vertex " << p.first
14          << " is {" << p.second.path << "} having length " << p.second.cost
15          << endl;
16   }
17 }
```

Κώδικας 8.6: source file προγράμματος που καλεί τον αλγόριθμο του Dijkstra (dijkstra_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

```
1  $ g++ -std=c++11 graph.cpp dijkstra.cpp dijkstra_ex1.cpp -o dijkstra_ex1
2  $ ./dijkstra_ex1
```

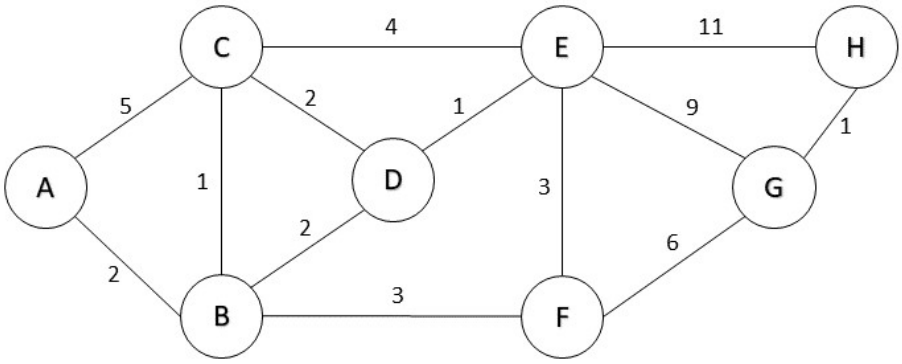
Η δε έξοδος που παράγεται είναι η ακόλουθη:

```
Shortest path from vertex A to vertex A is {A} having length 0
Shortest path from vertex A to vertex B is {A B} having length 2
Shortest path from vertex A to vertex C is {A B C} having length 5
Shortest path from vertex A to vertex D is {A B D} having length 3
Shortest path from vertex A to vertex E is {A B D E} having length 5
Shortest path from vertex A to vertex F is {A B D E F} having length 10
```

8.4 Παραδείγματα

8.4.1 Παράδειγμα 1

Για το σχήμα 8.3 και με αφετηρία την κορυφή A συμπληρώστε τον πίνακα εκτέλεσης του αλγορίθμου για την εύρεση των συντομότερων διαδρομών του Dijkstra και καταγράψτε τις διαδρομές που εντοπίζονται από την αφετηρία προς όλες τις άλλες κορυφές.



Σχήμα 8.3: Ένα μη κατευθυνόμενο γράφημα 8 κορυφών με βάρη στις ακμές του

Ο ακόλουθος πίνακας δείχνει την εκτέλεση του αλγορίθμου

Σύνολο S	A	B	C	D	E	F	G	H
$\{\}$	0	∞	∞	∞	∞	∞	∞	∞
$\{A\}$	0	2_A	5_A	∞	∞	∞	∞	∞
$\{A,B\}$	0	2_A	3_B	∞	∞	5_B	∞	∞
$\{A,B,C\}$	0	2_A	3_B	4_B	7_C	5_B	∞	∞
$\{A,B,C,D\}$	0	2_A	3_B	4_B	5_D	5_B	∞	∞
$\{A,B,C,D,E\}$	0	2_A	3_B	4_B	5_D	5_B	14_E	16_E
$\{A,B,C,D,E,F\}$	0	2_A	3_B	4_B	5_D	5_B	11_F	16_E
$\{A,B,C,D,E,F,G\}$	0	2_A	3_B	4_B	5_D	5_B	11_F	12_E
$\{A,B,C,D,E,F,G,H\}$	0	2_A	3_B	4_B	5_D	5_B	11_F	12_E

Πίνακας 8.5: Συνοπτική εκτέλεση του αλγορίθμου

Οι συντομότερες διαδρομές είναι:

- Για την κορυφή A η διαδρομή είναι η A με μήκος 0
- Για την κορυφή B η διαδρομή είναι η A-B με μήκος 2
- Για την κορυφή C η διαδρομή είναι η A-B-C με μήκος 3
- Για την κορυφή D η διαδρομή είναι η A-B-D με μήκος 4
- Για την κορυφή E η διαδρομή είναι η A-B-D-E με μήκος 5

- Για την κορυφή F η διαδρομή είναι η A-B-F με μήκος 5
- Για την κορυφή G η διαδρομή είναι η A-B-F-G με μήκος 11
- Για την κορυφή H η διαδρομή είναι η A-B-F-G-H με μήκος 12

8.4.2 Παράδειγμα 2

Γράψτε πρόγραμμα που να διαβάζει ένα γράφημα και να εμφανίζει για κάθε κορυφή το βαθμό της, δηλαδή το πλήθος των κορυφών με τις οποίες συνδέεται απευθείας καθώς και το μέσο όρο βαρών για αυτές τις ακμές. Επιπλέον για κάθε κορυφή να εμφανίζει τις υπόλοιπες κορυφές οι οποίες μπορούν να προσεγγιστούν με διαδρομές μήκους 1,2,3 κοκ.

```
#include "dijkstra.hpp"
#include "graph.hpp"
#include <algorithm> // max_element
#include <sstream>

using namespace std;

int main() {
  map<string, vector<pair<int, string>>> graph = read_data("graph2.txt");
  for (auto &kv : graph) {
    double sum = 0.0;
    for (auto &p : kv.second) {
      sum += p.first;
    }
    cout << "Vertex " << kv.first << " has degree " << kv.second.size()
         << " and average weighted degree " << sum / kv.second.size() << endl;
  }

  for (auto &kv : graph) {
    string source_vertex = kv.first;
    cout << "Source " << source_vertex << ": ";
    map<string, path_info> shortest_path_distances;
    compute_shortest_paths_to_all_vertices(graph, source_vertex,
                                           shortest_path_distances);

    vector<int> distances;
    for (auto &p : shortest_path_distances)
      distances.push_back(p.second.cost);
    int max = *(max_element(distances.begin(), distances.end()));

    for (int i = 1; i <= max; i++) {
      stringstream ss;
      ss << "dist=" << i << "->{";
      for (auto &p : shortest_path_distances)
        if (p.second.cost == i)
          ss << p.first << " ";
      ss << "} ";
      string sss = ss.str();
      if (sss.substr(sss.length() - 3) != "{} ") // check for empty list
        cout << sss;
    }
    cout << endl;
  }
}
```

Κώδικας 8.7: (lab08_ex2.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

```
$ g++ -std=c++11 lab08_ex2.cpp graph.cpp dijkstra.cpp -o lab08_ex2
$ ./lab08_ex2
```

Η δε έξοδος που παράγεται είναι η ακόλουθη:

```
Vertex A has degree 2 and average weighted degree 3.5
Vertex B has degree 4 and average weighted degree 2
Vertex C has degree 4 and average weighted degree 3
Vertex D has degree 3 and average weighted degree 1.66667
Vertex E has degree 5 and average weighted degree 5.6
Vertex F has degree 3 and average weighted degree 4
Vertex G has degree 3 and average weighted degree 5.33333
Vertex H has degree 2 and average weighted degree 6
Source A: dist=2->{B } dist=3->{C } dist=4->{D } dist=5->{E F } dist=11->{G } dist=12->{H }
Source B: dist=1->{C } dist=2->{A D } dist=3->{E F } dist=9->{G } dist=10->{H }
Source C: dist=1->{B } dist=2->{D } dist=3->{A E } dist=4->{F } dist=10->{G } dist=11->{H }
Source D: dist=1->{E } dist=2->{B C } dist=4->{A F } dist=10->{G } dist=11->{H }
Source E: dist=1->{D } dist=3->{B C F } dist=5->{A } dist=9->{G } dist=10->{H }
Source F: dist=3->{B E } dist=4->{C D } dist=5->{A } dist=6->{G } dist=7->{H }
Source G: dist=1->{H } dist=6->{F } dist=9->{B E } dist=10->{C D } dist=11->{A }
Source H: dist=1->{G } dist=7->{F } dist=10->{B E } dist=11->{C D } dist=12->{A }
```

8.5 Ασκήσεις

1. Υλοποιήστε τον αλγόριθμο των Bellman-Ford [9] για την εύρεση της συντομότερης διαδρομής από μια κορυφή προς όλες τις άλλες κορυφές.

2. Υλοποιήστε έναν αλγόριθμο τοπολογικής ταξινόμησης για DAGs [6].

Βιβλιογραφία

[1] Geeks for Geeks, graphs and its representations, https://www.geeksforgeeks.org/graph-and-its-representations/

[2] HackerEarth, graph representation, https://www.hackerearth.com/practice/algorithms/graphs/graph-representation/tutorial/

[3] Programming-Algorithms.net, Floyd-Warshall algorithm, http://www.programming-algorithms.net/article/45708/Floyd-Warshall-algorithm

[4] PROGRAMIZ, Prim's algorithm, https://www.programiz.com/dsa/prim-algorithm

[5] PROGRAMIZ, Kruskal's algorithm, https://www.programiz.com/dsa/kruskal-algorithm

[6] Geeks for Geeks, topological sorting, https://www.geeksforgeeks.org/topological-sorting/

[7] Discrete Mathematics: An open introduction by Oscar Levin, Euler Paths and Circuits, http://discretetext.oscarlevin.com/dmoi/sec_paths.html

[8] HackerEarth, Strongly Connected Components, https://www.hackerearth.com/practice/algorithms/graphs/strongly-connected-components/tutorial/

[9] Brilliant.org, Bellman-Ford Algorithm, https://brilliant.org/wiki/bellman-ford-algorithm/

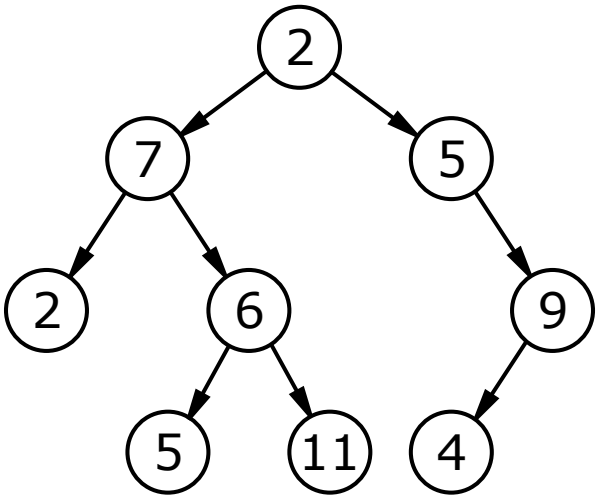
[10] Algorithm visualization, Dijkstra's shortest path, https://www.cs.usfca.edu/ galles/visualization/Dijkstra.html

Εργαστήριο 9

Δένδρα

9.1 Εισαγωγή

Τα δένδρα όπως και τα γραφήματα είναι μη γραμμικές δομές δεδομένων που αποτελούν συλλογές κόμβων. Τα δένδρα επιτρέπουν ιεραρχική οργάνωση των δεδομένων όπως φαίνεται στο Σχήμα 9.1. Αυτό το στοιχείο τους επιτρέπει να έχουν καλύτερες επιδόσεις προσπέλασης των επιμέρους στοιχείων σε σχέση με τις γραμμικές λίστες. Με κατάλληλη διευθέτηση των στοιχείων ενός δένδρου καθώς και με εφαρμογή προχωρημένων μηχανισμών εισαγωγής και διαγραφής στοιχείων ο χρόνος εκτέλεσης των περισσότερων λειτουργιών σε ένα δένδρο (ισοζυγισμένο δυαδικό δένδρο αναζήτησης) γίνεται $O(\log n)$. Στην STL τα δένδρα χρησιμοποιούνται στην υλοποίηση των containers std::map και std::set.



Σχήμα 9.1: Ένα απλό δένδρο [1]

9.2 Δένδρα

Ένα δένδρο (tree) αποτελείται από κόμβους (nodes) που συνδέονται μεταξύ τους με κατευθυνόμενες ακμές (edges). Ο πρώτος (υψηλότερος) κόμβος του δένδρου ονομάζεται ρίζα (root) ενώ οι κόμβοι που βρίσκονται στα άκρα του δένδρου λέγονται φύλλα (leaves). Οι κόμβοι με τους οποίους συνδέεται απευθείας ένας κόμβος ονομάζονται παιδιά (children) του κόμβου. Αντίστοιχα, ένας κόμβος που έχει παιδιά ονομάζεται γονέας (parent) των αντίστοιχων παιδιών-κόμβων. Απόγονοι (descendants) ενός κόμβου είναι οι κόμβοι για τους οποίους υπάρχει διαδρομή-μονοπάτι (path) πραγματοποιώντας διαδοχικές μεταβάσεις από γονείς σε παιδιά. Αντίστοιχα ορίζε-

ται και η έννοια των προγόνων (ancestors) ενός κόμβου με τη ρίζα να είναι ο μοναδικός κόμβος που δεν έχει προγόνους.

Τα δένδρα είναι αναδρομικές δομές από τη φύση τους. Κάθε κόμβος ενός δένδρου ορίζει έναν αριθμό από μικρότερα δένδρα, ένα για κάθε παιδί του. Σε ένα δένδρο με N κόμβους υπάρχουν πάντα $N-1$ ακμές καθώς όλοι οι κόμβοι εκτός από τον κόμβο ρίζα έχουν μια ακμή η οποία τους συνδέει με τον γονέα τους.

Το μήκος ενός μονοπατιού ανάμεσα σε δύο κόμβους είναι ίσο με το πλήθος των ακμών του μονοπατιού. Εφόσον υπάρχει μονοπάτι μέσω του οποίου συνδέονται δύο κόμβοι το μονοπάτι αυτό είναι μοναδικό. Για κάθε κόμβο ορίζεται ως **βάθος του κόμβου** (depth) το μήκος του μονοπατιού από τη ρίζα του δένδρου μέχρι τον ίδιο τον κόμβο. Αντίστοιχα, **ύψος ενός κόμβου** (height) είναι το μήκος του μακρύτερου μονοπατιού από τον κόμβο προς ένα από τα φύλλα του δένδρου για τα οποία υφίσταται μονοπάτι με αφετηρία τον κόμβο.

9.3 Δυαδικά δένδρα

Δυαδικό δένδρο είναι ένα δένδρο για το οποίο ισχύει ότι κάθε κόμβος έχει το πολύ δύο παιδιά [2]. Ένα δένδρο μπορεί να διανυθεί με διαφορετικούς τρόπους. Ορισμένοι βασικοί τρόποι διάσχισης (traversal) του δένδρου παρουσιάζονται στη συνέχεια [5].

9.3.1 Αναζήτηση κατά βάθος

Η αναζήτηση κατά βάθος (DFS = Depth First Search) διανύει το δένδρο αναζήτησης εξαντλώντας μονοπάτια από τη ρίζα προς τα φύλλα του δένδρου. Ένας τρόπος για να επιτευχθεί αυτό είναι η χρήση αναδρομής.

Προ-διατακτική αναζήτηση κατά βάθος

Στη διάσχιση του δένδρου προ-διατακτικά (pre-order) πρώτα πραγματοποιείται η επίσκεψη στη ρίζα και μετά καλείται αναδρομικά η ίδια συνάρτηση πρώτα για το αριστερό υποδένδρο και μετά για το δεξιό υποδένδρο. Συνηθισμένες χρήσεις της pre-order διάσχισης είναι η δημιουργία αντιγράφων ενός δένδρου καθώς και η λήψη της prefix μορφής ενός expression tree [3].

Ένδο-διατακτική αναζήτηση κατά βάθος

Στη διάσχιση του δένδρου ένδο-διατακτικά (in-order) καλείται αναδρομικά η συνάρτηση για το αριστερό υποδένδρο, μετά πραγματοποιείται επίσκεψη στη ρίζα και μετά καλείται αναδρομικά η συνάρτηση για το δεξιό υποδένδρο. Εφόσον το δένδρο είναι δυαδικό δένδρο αναζήτησης, η in-order διάσχιση επιστρέφει τους κόμβους σε μη φθίνουσα σειρά. Σχετικά με το τι είναι τα δυαδικά δένδρα αναζήτησης δείτε την παράγραφο 9.4).

Μέτα-διατακτική αναζήτηση κατά βάθος

Στη διάσχιση του δένδρου μέτα-διατακτικά (post-order) πρώτα καλείται αναδρομικά η συνάρτηση για το αριστερό υποδένδρο, μετά καλείται αναδρομικά για το δεξιό υποδένδρο και τέλος πραγματοποιείται η επίσκεψη στη ρίζα. Συνηθισμένες χρήσεις της post-order διάσχισης είναι η διαγραφή ενός δένδρου καθώς και η λήψη της postfix μορφής ενός expression tree [4].

9.3.2 Αναζήτηση κατά πλάτος

Στην αναζήτηση κατά πλάτος (BFS=Breadth First Search) οι κόμβοι του δένδρου διανύονται κατά επίπεδα ξεκινώντας από τη ρίζα και μεταβαίνοντας από πάνω προς τα κάτω. Σε κάθε επίπεδο η προσπέλαση στους κόμβους γίνεται από αριστερά προς τα δεξιά. Για να επιτευχθεί αυτό το είδος διάσχισης του δένδρου χρησιμοποιείται μια ουρά (queue) στην οποία μόλις εξετάζεται ένα στοιχείο προστίθενται στο πίσω άκρο της ουράς τα παιδιά του.

```
#include <cstddef>
#include <string>

struct node {
    std::string key;
    node *left;
    node *right;
};

node* insert(node *root_node, std::string key);
void print_level_order(node *root_node);
void print_pre_order(node *root_node);
void destroy(node *root_node);
void print_in_order(node *root_node);
void print_post_order(node *root_node);
```

Κώδικας 9.1: header file για το δυαδικό δένδρο (binary_tree.hpp)

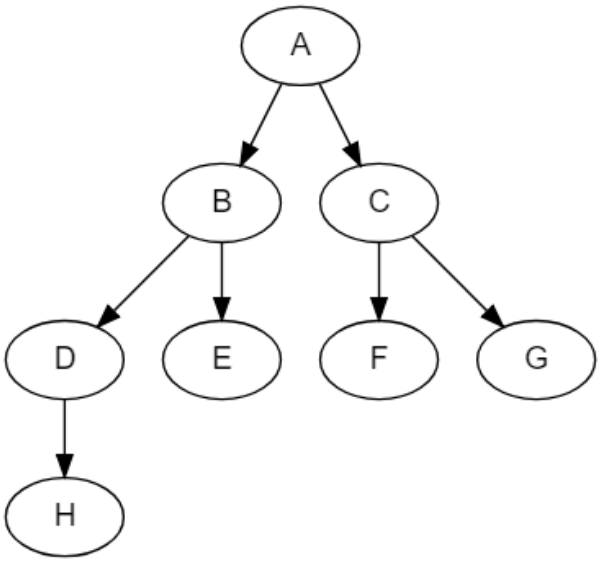
```
#include "binary_tree.hpp"
#include <queue>
#include <iostream>
using namespace std;

node* insert(node *root_node, string key)
{
    if (root_node == NULL)
    {
        cout << "key " << key << " inserted (root of the tree)" << endl;
        return new node{key, NULL, NULL};
    }
    else
    {
        queue<node *> q;
        q.push(root_node);
        while (!q.empty())
        {
            node *anode = q.front();
            q.pop();
            if (anode->left != NULL && anode->right != NULL)
            {
                q.push(anode->left);
                q.push(anode->right);
            }
            else
            {
                if (anode->left == NULL)
                {
                    anode->left = new node{key, NULL, NULL};
                }
                else
                {
                    anode->right = new node{key, NULL, NULL};
                }
                cout << "key " << key << " inserted" << endl;
                return anode;
            }
        }
    }
    return NULL;
}
```

```
43
44 void print_level_order(node *root_node)
45 {
46     if (root_node == NULL)
47     {
48         return;
49     }
50     queue<node *> q;
51     q.push(root_node);
52     while (!q.empty())
53     {
54         node *node = q.front();
55         q.pop();
56         cout << node->key << " ";
57         if (node->left != NULL)
58         {
59             q.push(node->left);
60         }
61         if (node->right != NULL)
62         {
63             q.push(node->right);
64         }
65     }
66 }
67
68 void print_pre_order(node *root_node)
69 {
70     if (root_node != NULL)
71     {
72         cout << root_node->key << " ";
73         if (root_node->left != NULL)
74         {
75             print_pre_order(root_node->left);
76         }
77         if (root_node->right != NULL)
78         {
79             print_pre_order(root_node->right);
80         }
81     }
82     else
83     {
84         cout << "EMPTY";
85     }
86 }
87
88 void print_in_order(node *root_node)
89 {
90     if (root_node != NULL)
91     {
92         if (root_node->left != NULL)
93         {
94             print_in_order(root_node->left);
95         }
96         cout << root_node->key << " ";
97         if (root_node->right != NULL)
98         {
99             print_in_order(root_node->right);
100        }
101    }
102    else
103    {
```

```
104             cout << "EMPTY";
105         }
106 }
107
108 void print_post_order(node *root_node)
109 {
110     if (root_node != NULL)
111     {
112         if (root_node->left != NULL)
113         {
114             print_post_order(root_node->left);
115         }
116         if (root_node->right != NULL)
117         {
118             print_post_order(root_node->right);
119         }
120         cout << root_node->key << " ";
121     }
122     else
123     {
124         cout << "EMPTY";
125     }
126 }
127
128 void destroy(node *root_node)
129 {
130     if (root_node != NULL)
131     {
132         destroy(root_node->left);
133         destroy(root_node->right);
134         delete root_node;
135     }
136 }
```

Κώδικας 9.2: source file για το δυαδικό δένδρο αναζήτησης (binary_tree.cpp)

Ο ακόλουθος κώδικας δημιουργεί το δυαδικό δένδρο του Σχήματος 9.2.



Σχήμα 9.2: Δυαδικό δένδρο με λεκτικά ως τιμές κλειδιών στους κόμβους

```
#include "binary_tree.hpp"
#include <iostream>
#include <vector>
using namespace std;

int main()
{
    node *root_node = NULL;
    vector<string> v = {"A", "B", "C", "D", "E", "F", "G", "H"};
    for (string x : v)
    {
        if (root_node == NULL)
            root_node = insert(root_node, x);
        else
            insert(root_node, x);
    }

    cout << "Level-order Traversal ";
    print_level_order(root_node);
    cout << endl;
    cout << "Pre-order Traversal ";
    print_pre_order(root_node);
    cout << endl;
    cout << "In-order Traversal ";
    print_in_order(root_node);
    cout << endl;
    cout << "Post-order Traversal ";
    print_post_order(root_node);
    cout << endl;
}
```

Κώδικας 9.3: Δοκιμή των συναρτήσεων του δυαδικού δένδρου (binary_tree_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

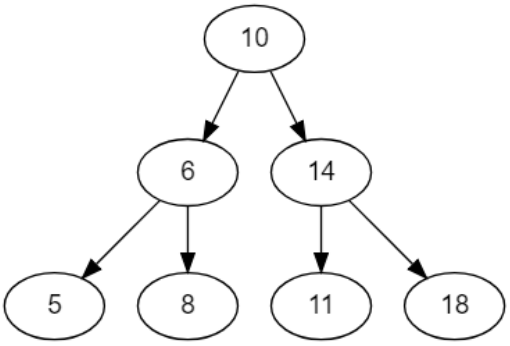
```
$ g++ -Wall -std=c++11 binary_tree.cpp binary_tree_ex1.cpp -o binary_tree_ex1
$ ./binary_tree_ex1
```

Η δε έξοδος που παράγεται και για τους 4 τρόπους διάσχισης του δένδρου είναι η ακόλουθη:

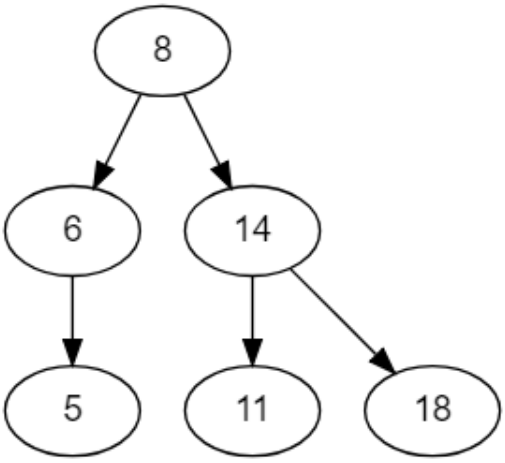
```
key A inserted (root of the tree)
key B inserted
key C inserted
key D inserted
key E inserted
key F inserted
key G inserted
key H inserted
Level-order Traversal A B C D E F G H
Pre-order Traversal A B D H E C F G
In-order Traversal H D B E A F C G
Post-order Traversal H D E B F G C A
```

9.4 Δυαδικά δένδρα αναζήτησης

Σε ένα δυαδικό δένδρο αναζήτησης θα πρέπει να ισχύει ότι για κάθε κόμβο όλες οι τιμές κλειδιών στο δένδρο αριστερά του κόμβου θα πρέπει να είναι μικρότερες από την τιμή κλειδιού του κόμβου. Αντίστοιχα, όλες οι τιμές κλειδιών στο δένδρο δεξιά του κάθε κόμβου θα πρέπει να είναι μεγαλύτερες από την τιμή κλειδιού του κόμβου.



Σχήμα 9.3: Δυαδικό δένδρο αναζήτησης



Σχήμα 9.4: Το δυαδικό δένδρο αναζήτησης μετά τη διαγραφή της ρίζας

9.4.1 Υλοποίηση δυαδικού δένδρου αναζήτησης

Ιδιαίτερη προσοχή θα πρέπει να δοθεί στην υλοποίηση της διαγραφής ενός κόμβου από το δένδρο έτσι ώστε το δένδρο και μετά τη διαγραφή να εξακολουθεί να είναι δυαδικό δένδρο αναζήτησης [6].

```
1  #include <cstddef>
2  #include <iostream>
3
4  struct node
5  {
6      int key;
7      node *left;
8      node *right;
9  };
10
11 node *insert(node *tree, int key);
12 node *search(node *tree, int key);
13 node *remove(node *tree, int key);
14 void destroy(node *tree);
15 node *max(node *tree);
16 node *remove_max_node(node *tree, node *max_node);
17 node *min(node *tree);
18 void print_in_order(node *tree);
```

Κώδικας 9.4: header file για το δυαδικό δένδρο αναζήτησης (bst.hpp)

```
1  #include "bst.hpp"
2
3  using namespace std;
4
5  node *insert(node *tree, int key)
6  {
7      if (tree == NULL)
8      {
9          node *new_tree = new node;
10         new_tree->left = NULL;
11         new_tree->right = NULL;
12         new_tree->key = key;
13         return new_tree;
```

```
14      }
15      if (key < tree->key)
16      {
17          tree->left = insert(tree->left, key);
18      }
19      else
20      {
21          tree->right = insert(tree->right, key);
22      }
23      return tree;
24  }
25
26  node *search(node *tree, int key)
27  {
28      if (tree == NULL)
29      {
30          return NULL;
31      }
32      else if (tree->key == key)
33      {
34          return tree;
35      }
36      else if (key < tree->key)
37      {
38          return search(tree->left, key);
39      }
40      else
41      {
42          return search(tree->right, key);
43      }
44  }
45
46  node *remove(node *tree, int key)
47  {
48      if (tree == NULL)
49      {
50          return NULL;
51      }
52      if (tree->key == key)
53      {
54          if (tree->left == NULL)
55          {
56              node *right_subtree = tree->right;
57              delete tree;
58              return right_subtree;
59          }
60          if (tree->right == NULL)
61          {
62              node *left_subtree = tree->left;
63              delete tree;
64              return left_subtree;
65          }
66          node *max_node = max(tree->left);
67          max_node->left = remove_max_node(tree->left, max_node);
68          max_node->right = tree->right;
69          delete tree;
70          return max_node;
71      }
72      else if (tree->key > key)
73      {
74          tree->left = remove(tree->left, key);
```

```
75      }
76      else
77      {
78          tree->right = remove(tree->right, key);
79      }
80      return tree;
81  }
82
83  void destroy(node *tree)
84  {
85      if (tree != NULL)
86      {
87          destroy(tree->left);
88          destroy(tree->right);
89          delete tree;
90      }
91  }
92
93  node *max(node *tree)
94  {
95      if (tree == NULL)
96      {
97          return NULL;
98      }
99      else if (tree->right == NULL)
100     {
101         return tree;
102     }
103     return max(tree->right);
104 }
105
106 node *remove_max_node(node *tree, node *max_node_tree)
107 {
108     if (tree == NULL)
109     {
110         return NULL;
111     }
112     if (tree == max_node_tree)
113     {
114         return max_node_tree->left;
115     }
116     tree->right = remove_max_node(tree->right, max_node_tree);
117     return tree;
118 }
119
120 node *min(node *tree)
121 {
122     if (tree == NULL)
123     {
124         return NULL;
125     }
126     else if (tree->left == NULL)
127     {
128         return tree;
129     }
130     return min(tree->left);
131 }
132
133
134 void print_in_order(node *tree)
135 {
```

```
136     if (tree != NULL)
137     {
138         if (tree->left != NULL)
139         {
140             print_in_order(tree->left);
141         }
142         cout << tree->key << " ";
143         if (tree->right != NULL)
144         {
145             print_in_order(tree->right);
146         }
147     }
148     else
149     {
150         cout << "EMPTY";
151     }
152 }
```

Κώδικας 9.5: source file για το δυαδικό δένδρο αναζήτησης (bst.cpp)

```
1  #include "bst.hpp"
2  #include <vector>
3  using namespace std;
4
5  void test_search(node *root_node, int key)
6  {
7      cout << "Searching for key " << key << ": ";
8      node *p = search(root_node, key);
9      if (p != NULL)
10     {
11         cout << "found (" << p->key << ")" << endl;
12     }
13     else
14     {
15         cout << "not found " << endl;
16     }
17 }
18
19 void test_min_max(node *root_node)
20 {
21     cout << "Minimum " << min(root_node)->key << " Maximum " << max(root_node)->key << endl;
22 }
23
24 int main()
25 {
26     node *root_node = NULL;
27     vector<int> v = {10, 6, 5, 8, 14, 11, 18};
28     for (int x : v)
29     {
30         if (root_node == NULL)
31         {
32             root_node = insert(root_node, x);
33         }
34         else
35         {
36             insert(root_node, x);
37         }
38     }
39     cout << "In-order Traversal ";
40     print_in_order(root_node);
41     cout << endl;
```

```
42      test_search(root_node, 11);
43      test_search(root_node, 13);
44      test_min_max(root_node);
45      cout << "Remove node 10 ";
46      root_node = remove(root_node, 10);
47      cout << endl << "In-order Traversal ";
48      print_in_order(root_node);
49      cout << endl;
50      destroy(root_node);
51  }
```

Κώδικας 9.6: Δοκιμή των συναρτήσεων του δυαδικού δένδρου αναζήτησης (bst_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

```
1 $ g++ -Wall -std=c++11 bst.cpp bst_ex1.cpp -o bst_ex1
2 $ ./bst_ex1
```

Η δε έξοδος που παράγεται είναι η ακόλουθη:

```
1 In-order Traversal 5 6 8 10 11 14 18
2 Searching for key 11: found (11)
3 Searching for key 13: not found
4 Minimum 5 Maximum 18
5 Remove node 10
6 In-order Traversal 5 6 8 11 14 18
```

Για να πραγματοποιηθεί η διαγραφή του κόμβου 10, εντοπίζεται ο κόμβος με τη μεγαλύτερη τιμή στο αριστερό υποδένδρο του κόμβου 10, που είναι ο 8 και ο κόμβος αυτός αφαιρείται από το δένδρο αντικαθιστώντας τον κόμβο 10.

9.5 Ισοζυγισμένα δυαδικά δένδρα αναζήτησης

Οι καλές επιδόσεις ενός δυαδικού δένδρου αναζήτησης χάνονται όταν το δένδρο δεν είναι ισοζυγισμένο (balanced), δηλαδή όταν υπάρχουν μονοπάτια από τη ρίζα προς τα φύλλα με μεγάλα βάθη ενώ άλλα μονοπάτια έχουν μικρά βάθη. Υπάρχουν διάφορες μορφές ισοζυγισμένων δένδρων με πλέον δημοφιλή τα κόκκινα-μαύρα δένδρα (red black trees) και τα AVL (Adelson, Velskii και Landis) δένδρα. Σε αυτά τα δένδρα πραγματοποιούνται ειδικές λειτουργίες (περιστροφές) έτσι ώστε κατά την εισαγωγή νέων τιμών στο δένδρο και τη διαγραφή τιμών από το δένδρο, τα βάθη των φύλλων του δένδρου εγγυημένα να διατηρούνται σε κοντινές τιμές μεταξύ τους. Ισχύει ότι τα AVL δένδρα είναι καλύτερα ισοζυγισμένα από τα κόκκινα-μαύρα δένδρα αλλά έχουν το μειονέκτημα της υψηλότερης υπολογιστικής επιβάρυνσης κατά την εισαγωγή και τη διαγραφή κόμβων.

9.6 Παραδείγματα

9.6.1 Παράδειγμα 1

Δεδομένου ενός δυαδικού δένδρου ζητείται η εκτύπωση όλων των διαδρομών από τη ρίζα του δένδρου μέχρι κάθε φύλλο. Για παράδειγμα, για το δένδρο του Σχήματος 9.2 το πρόγραμμα θα πρέπει να επιστρέψει ABDH, ABE, ACF και ACG.

```
1 #include "binary_tree.hpp"
2 #include <iostream>
3 #include <vector>
4 using namespace std;
5
6 void print_vector(vector<string> previous_nodes)
7 {
8     for (string s : previous_nodes)
```

```
9       {
10          cout << s << " ";
11      }
12      cout << endl;
13  }
14
15  void print_tree(node *root_node, vector<string> previous_nodes)
16  {
17      if (root_node == NULL)
18      {
19          print_vector(previous_nodes);
20      }
21      else
22      {
23          // cout << "call root node=" << root_node->key << " path=";
24          // print_vector(previous_nodes);
25          previous_nodes.push_back(root_node->key);
26          if (root_node->left == NULL && root_node->right == NULL)
27          {
28              print_vector(previous_nodes);
29          }
30          else
31          {
32              if (root_node->left != NULL)
33                  print_tree(root_node->left, previous_nodes);
34              if (root_node->right != NULL)
35                  print_tree(root_node->right, previous_nodes);
36          }
37      }
38  }
39
40  int main()
41  {
42      node *root_node = NULL;
43      vector<string> v = {"A", "B", "C", "D", "E", "F", "G", "H"};
44      for (string x : v)
45      {
46          if (root_node == NULL)
47              root_node = insert(root_node, x);
48          else
49              insert(root_node, x);
50      }
51
52      cout << "Level-order Traversal ";
53      print_level_order(root_node);
54      cout << endl;
55
56      vector<string> path;
57      print_tree(root_node, path);
58  }
```

Κώδικας 9.7: Λύση παραδείγματος 1 (lab09_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

```
1  $ g++ -Wall -std=c++11 binary_search.cpp lab09_ex1.cpp -o lab09_ex1
2  $ ./lab09_ex1
```

Η έξοδος που παράγεται είναι η ακόλουθη:

```
1  key A inserted (root of the tree)
2  key B inserted
3  key C inserted
```

```
key D inserted
key E inserted
key F inserted
key G inserted
key H inserted
Level−order Traversal A B C D E F G H
A B D H
A B E
A C F
A C G
```

9.6.2 Παράδειγμα 2

Δεδομένου ενός δυαδικού δένδρου ζητείται να πραγματοποιείται έλεγχος σχετικά με το εάν το δένδρο είναι δυαδικό δένδρο αναζήτησης ή όχι.

```
#include ”bst.hpp”
#include <vector>
using namespace std;

int is_bst(node *node)
{
    if (node == NULL)
    {
        return true;
    }
    if (node−>left != NULL && min(node−>left)−>key > node−>key)
    {
        return false;
    }
    if (node−>right != NULL && max(node−>right)−>key <= node−>key)
    {
        return false;
    }
    if (!is_bst(node−>left) || !is_bst(node−>right))
    {
        return false;
    }
    return true;
}

int main()
{
    node *root_node = NULL;
    vector<int> v = {10, 6, 5, 8, 14, 11, 18};
    for (int x : v)
    {
        if (root_node == NULL)
        {
            root_node = insert(root_node, x);
        }
        else
        {
            insert(root_node, x);
        }
    }
    cout << (is_bst(root_node) ? ”The tree is a BST” : ”The tree is not a BST”) << endl;
    // replacing root node with zero
    root_node−>key = 0;
    cout << (is_bst(root_node) ? ”The tree is a BST” : ”The tree is not a BST”) << endl;
}
```

Κώδικας 9.8: Λύση παραδείγματος 2 (lab09_ex2.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:

```
$ g++ -Wall -std=c++11 bst.cpp lab09_ex2.cpp -o lab09_ex2
$ ./lab09_ex2
```

Η έξοδος που παράγεται είναι η ακόλουθη:

```
The tree is a BST
The tree is not a BST
```

9.7 Ασκήσεις

1. Να γράψετε πρόγραμμα που να εμφανίζει τους κόμβους ενός δυαδικού δένδρου κατά επίπεδα από κάτω προς τα πάνω και από αριστερά προς τα δεξιά. Δηλαδή στο δένδρο του Σχήματος 9.2 θα πρέπει οι κόμβοι να εμφανιστούν ως D,E,F,G,B,C,A.

2. Να γράψετε πρόγραμμα που να δημιουργεί από έναν ταξινομημένο πίνακα ακεραίων ένα δυαδικό δένδρο αναζήτησης. Να χρησιμοποιηθεί ο ακόλουθος αλγόριθμος:

   (α΄) Εύρεση του μεσαίου στοιχείου του πίνακα και ορισμός του ως ρίζα του δένδρου

   (β΄) Αναδρομική εκτέλεση για το αριστερό και το δεξιό μισό

      i. Εύρεση του μεσαίου στοιχείου του αριστερού μέρους και ορισμός του ως αριστερό παιδί της ρίζας του βήματος α'

      ii. Εύρεση του μεσαίου στοιχείου του δεξιού μέρους και ορισμός του ως δεξί παιδί της ρίζας του βήματος α'

Βιβλιογραφία

[1] Wikipedia, Tree (data structure), https://en.wikipedia.org/wiki/Tree_(data_structure)

[2] Binary Trees by Nick Parlante, http://cslibrary.stanford.edu/110/BinaryTrees.html

[3] Wikipedia, Polish Notation, https://en.wikipedia.org/wiki/Polish_notation

[4] Wikipedia, Reverse Polish Notation, https://en.wikipedia.org/wiki/Reverse_Polish_notation

[5] Tree Traversals (Inorder, Preorder and Postorder), https://www.geeksforgeeks.org/tree-traversals-inorder-preorder-and-postorder/

[6] Alex Allain, Jumping into C++, cprogramming.com, Chapter 17 - Binary Trees, 2013