Δομές Δεδομένων και Αλγόριθμοι - Εργαστήριο 7
Κατακερματισμός, δομές κατακερματισμού στην STL

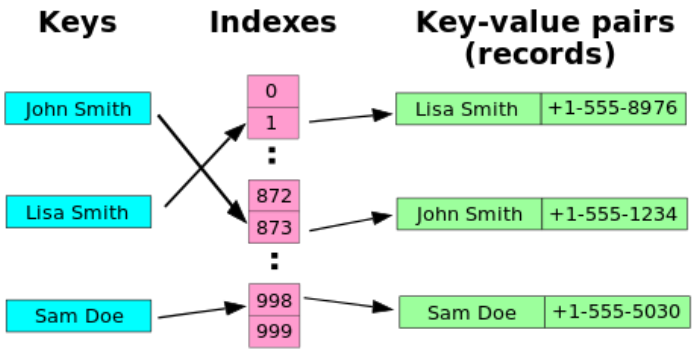
Τ.Ε.Ι. Ηπείρου, Τμήμα Μηχανικών Πληροφορικής Τ.Ε.
Χρήστος Γκόγκος - Αναπληρωτής Καθηγητής

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.

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.



Σχήμα 1: Κατακερματισμός εγγραφών σε πίνακα κατακερματισμού [1]

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).

```
#include <string>

using namespace std;

size_t hash0(string &key) {
   size_t h = 0;
   for (char c : key)
      h += c;
   return h;
}

size_t hash1(string &key) {
   size_t h = 0;
   for (char c : key)
      h = 37 * h + c;
   return h;
}

// Jenkins One−at−a−time hash
size_t hash2(string &key) {
   size_t h = 0;
   for (char c : key) {
      h += c;
      h += (h << 10);
      h ^= (h >> 6);
   }
   h += (h << 3);
   h ^= (h >> 11);
   h += (h << 15);
   return h;
}

// FNV (—FowlerNollVo) hash
size_t hash3(string &key) {
   size_t h = 0x811c9dc5;
   for (char c : key)
      h = (h ^ c) * 0x01000193;
   return h;
}
```

Κώδικας 1: Διάφορες συναρτήσεις κατακερματισμού (hashes.cpp)

```
#include ”hashes.cpp”
#include <iostream>

using namespace std;

int main() {
   constexpr int HT_SIZE = 101;
   string keys[] = {”nikos”, ”maria”, ”petros”, ”kostas”};
   for (string key : keys) {
      size_t h0 = hash0(key) % HT_SIZE;
      size_t h1 = hash1(key) % HT_SIZE;
      size_t h2 = hash2(key) % HT_SIZE;
      size_t h3 = hash3(key) % HT_SIZE;
      cout << ”string ” << key << ” hash0=” << h0 << ” hash1=” << h1
```

```
15            << ", hash2=" << h2 << ", hash3=" << h3 << endl;
16    }
17 }
```

Κώδικας 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 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 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.

2.1 Ανοικτή διευθυνσιοδότηση

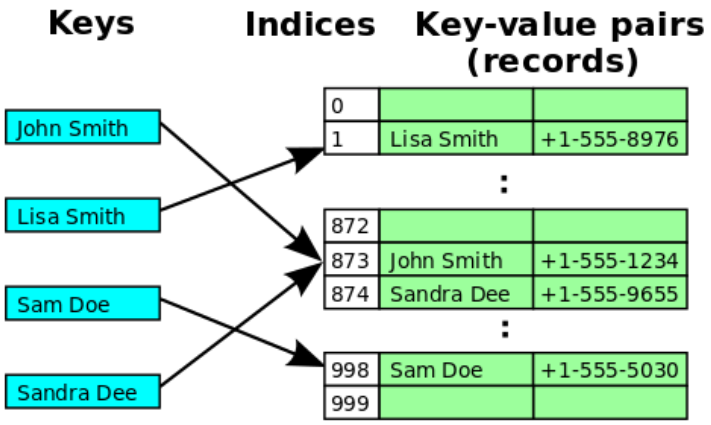
Στην ανοικτή διευθυνσιοδότηση (open addressing, closed hashing) όλα τα δεδομένα αποθηκεύονται απευθείας στον πίνακα κατακερματισμού. Αν συμβεί σύγκρουση τότε ελέγχεται αν κάποιο από τα υπόλοιπα κελιά είναι διαθέσιμο και η εγγραφή τοποθετείται εκεί. Συνεπώς, θα πρέπει το μέγεθος του hashtable να είναι 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;
4
5  struct record {
6    string key;
7    string value;
8  };
```



Σχήμα 2: Κατακερματισμός εγγραφών με ανοικτή διευθυνσιοδότηση και γραμμική ανίχνευση [1]

```
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
```

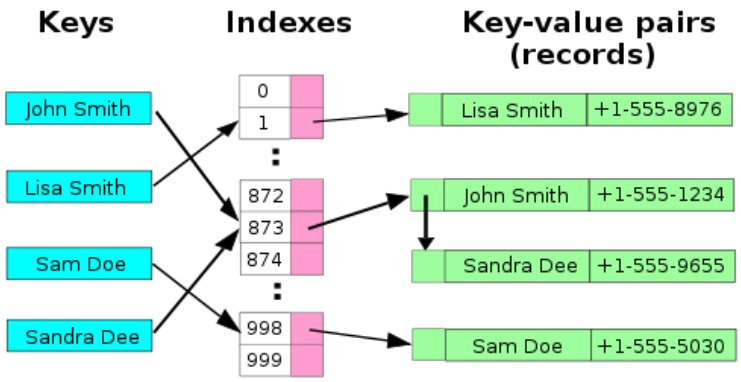
```
49    void put(record *arecord) {
50      if (size == capacity) {
51        cerr << "The hashtable is full" << endl;
52        return;
53      }
54      size_t hash_code = hash(arecord->key);
55      while (data[hash_code] != nullptr && data[hash_code]->key != "ERASED") {
56        if (data[hash_code]->key == 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 5: Ανοικτή διευθυνσιοδότηση (open_addressing.cpp)

```
#(1) Sam Doe +1−555−5030
#(46) John Smith +1−555−1234
#(57) Lisa Smith +1−555−8976
Load factor: 0.029703
Key found: Sam Doe +1−555−5030
#(46) John Smith +1−555−1234
#(57) Lisa Smith +1−555−8976
Load factor: 0.019802
```

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 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.

```
#include <iostream>
#include <list>

using namespace std;

struct record {
   string key;
   string value;
};

class sc_hashtable {
private:
   int size;
   list<record *> *buckets;

   size_t hash(string &key) {
      size_t value = 0;
      for (size_t i = 0; i < key.length(); i++)
         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 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].

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: 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 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 αναγνωρίζει 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 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
```

4 Παραδείγματα

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 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;
```

```
};

void insert(employee hash_table[], employee &ypa) {
  int pos = hash1(ypa.name) % N;
  while (hash_table[pos].name != "") {
    pos++;
    pos %= N;
  }
  hash_table[pos] = ypa;
}

bool search(employee hash_table[], string &name, employee &ypa) {
  int pos = hash1(name) % N;
  int c = 0;
  while (hash_table[pos].name != name) {
    if (hash_table[pos].name == "")
      return false;
    pos++;
    pos %= N;
    c++;
    if (c > N)
      return false;
  }
  ypa = hash_table[pos];
  return true;
}

int main() {
  vector<int> SIZES{20000, 30000, 80000};
  for (int x : SIZES) {
    struct employee *hash_table = new struct employee[N];
    // generate x random employees, insert them at the hashtable
    vector<string> names;
    for (int i = 0; i < x; i++) {
      employee ypa;
      ypa.name = generate_random_string(3);
      ypa.address = generate_random_string(20);
      insert(hash_table, ypa);
      names.push_back(ypa.name);
    }
    // generate x more names
    for (int i = 0; i < x; i++)
      names.push_back(generate_random_string(3));
    // time execution of 2*x searches in the HashTable
    auto t1 = high_resolution_clock::now();
    employee ypa;
    int c = 0;
    for (string name : names)
      if (search(hash_table, name, ypa)) {
        // cout << "Employee " << ypa.name << " " << ypa.address << endl;
        c++;
      }
    auto t2 = high_resolution_clock::now();
    std::chrono::duration<double, std::micro> duration = t2 - t1;
    cout << "Load factor: " << setprecision(2) << (double)x / (double)N
         << " employees found: " << c << ", employees not found: " << 2 * x - c
         << " time elapsed: " << std::fixed << duration.count() / 1E6
         << " seconds" << endl;
    delete[] hash_table;
  }
}
```

Κώδικας 10: Υλοποίηση πίνακα κατακερματισμού για γρήγορη αποθήκευση και αναζήτηση εγγραφών (lab07_ex1.cpp)

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

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.

```
#include "random_strings.cpp"
#include <chrono>
#include <iomanip>
#include <iostream>
#include <string>
#include <unordered_map>
#include <vector>

using namespace std::chrono;

struct employee {
  string name;
  string address;
};

int main() {
  vector<int> SIZES{20000, 30000, 80000};
  for (int x : SIZES) {
    unordered_map<string, employee> umap;
    // generate x random employees, insert them at the hashtable
    vector<string> names;
    for (int i = 0; i < x; i++) {
      employee ypa;
      ypa.name = generate_random_string(3);
      ypa.address = generate_random_string(20);
      umap[ypa.name] = ypa;
      names.push_back(ypa.name);
    }
    // generate x more names
    for (int i = 0; i < x; i++)
      names.push_back(generate_random_string(3));

    // time execution of 2*x searches in the HashTable
    auto t1 = high_resolution_clock::now();
    int c = 0;
    for (string name : names)
      if (umap.find(name) != umap.end()) {
        // cout << "Employee " << name << " " << umap[name].address << endl;
        c++;
      }
    auto t2 = high_resolution_clock::now();
    std::chrono::duration<double, std::micro> duration = t2 - t1;
    cout << "Load factor: " << setprecision(2) << umap.load_factor()
         << " employees found: " << c << ", employees not found: " << 2 * x - c
         << " time elapsed: " << std::fixed << duration.count() / 1E6
         << " seconds" << endl;
```

```
47   }
48 }
```

Κώδικας 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
```

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 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
```

5 Ασκήσεις

1. Γράψτε μια συνάρτηση που να δέχεται έναν πίνακα ακεραίων Α και έναν ακέραιο αριθμό sum και να βρίσκει το πλήθος από όλα τα ζεύγη τιμών του Α 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 Α και έναν ακέραιο αριθμό Κ και να βρίσκει τη μεγαλύτερη σε μήκος υποακολουθία στοιχείων του Α που έχει άθροισμα ίσο με Κ.

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.

Αναφορές

[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