Δομές Δεδομένων και Αλγόριθμοι
Εργαστήριο (C++)
Τ.Ε.Ι. Ηπείρου - Τμήμα Μηχανικών Πληροφορικής Τ.Ε.
Έκδοση 1.1

Χρήστος Γκόγκος
Αναπληρωτής Καθηγητής

Χειμερινό εξάμηνο 2018-2019

Περιεχόμενα

Εργαστήριο 1

Βασικές έννοιες στη C και στη C++

1.1 Εισαγωγή

Στο πρώτο αυτό εργαστήριο θα επιχειρηθεί η παρουσίαση των βασικών γνώσεων που απαιτούνται έτσι ώστε να είναι δυνατή η κατανόηση της ύλης που ακολουθεί. Ειδικότερα, θα γίνει αναφορά σε δείκτες, στη δυναμική δέσμευση και αποδέσμευση μνήμης, στο πέρασμα παραμέτρων σε συναρτήσεις (με τιμή και με αναφορά), στην παραγωγή τυχαίων τιμών, στους πίνακες (μονοδιάστατους, δισδιάστατους, πολυδιάστατους, δυναμικούς), στις δομές (struct), στις κλάσεις και στα αντικείμενα και τέλος στην ανάγνωση από αρχεία και στην εγγραφή σε αυτά. Θα παρουσιαστούν λυμένα παραδείγματα καθώς και εκφωνήσεις ασκήσεων προς επίλυση. Ο κώδικας όλων των παραδειγμάτων βρίσκεται στο https://github.com/chgogos/ceteiep_dsa. Αναλυτικότερη παρουσίαση των ανωτέρω θεμάτων γίνεται στα ελεύθερα διαθέσιμα βιβλία [1], [2], [3], [4] που παρατίθενται ως αναφορές στο τέλος του κειμένου του εργαστηρίου.

1.2 Δείκτες

Κάθε θέση μνήμης στην οποία μπορούν να αποθηκευτούν δεδομένα βρίσκεται σε μια διεύθυνση μνήμης. Η δε μνήμη του υπολογιστή αποτελείται από ένα συνεχόμενο χώρο διευθύνσεων. Αν μια μεταβλητή δηλωθεί ως τύπου int * (δείκτης σε ακέραιο) τότε η τιμή που θα λάβει ερμηνεύεται ως μια διεύθυνση που δείχνει σε μια θέση μνήμης η οποία περιέχει έναν ακέραιο. Από την άλλη μεριά το σύμβολο & επιτρέπει τη λήψη της διεύθυνσης μιας μεταβλητής. Στον ακόλουθο κώδικα δηλώνονται 2 ακέραιες μεταβλητές (a και b) και ένας δείκτης σε ακέραια τιμή (p). Ο δείκτης p λαμβάνει ως τιμή τη διεύθυνση της μεταβλητής a. Στη συνέχεια, οι μεταβλητές a και b λαμβάνουν τιμές μέσω του δείκτη p. Για να συμβεί αυτό γίνεται έμμεση αναφορά ή αλλιώς αποαναφορά (dereference) του δείκτη με το *p. Συνεπώς, το *p αντιστοιχεί στο περιεχόμενο της διεύθυνσης μνήμης που έχει ο δείκτης p.

```
1  #include <cstdio>
2
3  int main(int argc, char **argv) {
4    int a, b;
5    int *p;
6    p = &a;
7    *p = 5;
8    b = *p + 1;
9    printf("variable a=%d address=%p\n", a, &a);
10   printf("variable b=%d address=%p\n", b, &b);
11   printf("pointer p=%p *p=%d\n", p, *p);
12   return 0;
13 }
```

Κώδικας 1.1: Παράδειγμα με δείκτες (pointers1.cpp)

Χρησιμοποιώντας τον compiler g++, η μεταγλώττιση του κώδικα 1.1 γίνεται με την ακόλουθη εντολή:

```
g++ pointers1.cpp -o pointers1
```

Δημιουργείται το εκτελέσιμο pointers1 το οποίο όταν εκτελεστεί παράγει ως έξοδο το:

```
variable a=5 address=000000000022fe44
variable b=6 address=000000000022fe40
pointer p=000000000022fe44 *p=5
```

Ένα συνηθισμένο λάθος με δείκτες παρουσιάζεται όταν γίνεται dereference ενός δείκτη (δηλαδή, δεδομένου ενός δείκτη p όταν χρησιμοποιείται το *p) χωρίς ο δείκτης να έχει αρχικοποιηθεί πρώτα δείχνοντας σε μια έγκυρη θέση μνήμης. Σε αυτή την περίπτωση το πρόγραμμα καταρρέει.

```
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int *p;
  *p = 2;
  cout << *p << endl;
  return 0;
}
```

Κώδικας 1.2: Λανθασμένη χρήση δείκτη (pointers2.cpp)

```
segmentation fault
```

Αν η μεταγλώττιση του κώδικα γίνει με το compiler flag **-Wall** τότε θα εμφανιστεί μήνυμα που θα προειδοποιεί για τη λάθος χρήση του δείκτη.

```
g++ -Wall pointers2.cpp -o pointers2
pointers2.cpp: In function 'int main(int, char**)':
pointers2.cpp:8:11: warning: 'p' is used uninitialized in this function [-Wuninitialized]
     *p = 2;
          ^
```

1.3 Κλήση με τιμή και κλήση με αναφορά

Οι δείκτες μπορούν να χρησιμοποιηθούν έτσι ώστε να επιτευχθεί, εφόσον απαιτείται, κλήση με αναφορά (call by reference ή pass by reference) στις παραμέτρους μιας συνάρτησης και όχι κλήση με τιμή (call by value ή pass by value) που είναι ο προκαθορισμένος τρόπος κλήσης συναρτήσεων. Όταν γίνεται κλήση με τιμή τα δεδομένα αντιγράφονται από τη συνάρτηση που καλεί προς τη συνάρτηση που καλείται. Συνεπώς. αν στη συνάρτηση που καλείται τα δεδομένα αλλάξουν η τιμή τους παρουσιάζεται αλλαγμένη μόνο μέσα σε αυτή τη συνάρτηση και όχι στη συνάρτηση που την κάλεσε. Στην περίπτωση της κλήσης με αναφορά ένας δείκτης προς τα δεδομένα αντιγράφεται αντί για τα ίδια τα δεδομένα. Οι αλλαγές που γίνονται στη συνάρτηση που καλείται εφόσον αφορούν τα δεδομένα στα οποία δείχνει ο δείκτης αφορούν τα δεδομένα και της συνάρτησης από την οποία έγινε η κλήση. Συνεπώς, πρόκειται για τα ίδια δεδομένα και οποιαδήποτε αλλαγή σε αυτά μέσα από τη συνάρτηση που καλείται αντικατοπτρίζεται και στη συνάρτηση που καλεί.

Στο παράδειγμα που ακολουθεί η συνάρτηση swap (σε αντίθεση με τη συνάρτηση swap_not_ok) επιτυγχάνει την αντιμετάθεση των δύο μεταβλητών που δέχεται ως ορίσματα καθώς χρησιμοποιεί δείκτες που αναφέρονται στις ίδιες τις μεταβλητές του κυρίου προγράμματος και όχι σε αντίγραφά τους.

```
#include <cstdio>

void swap_not_ok(int x, int y) {
  int temp = x;
```

```
5   x = y;
6   y = temp;
7 }
8
9 void swap(int *x, int *y) {
10  int temp = *x;
11  *x = *y;
12  *y = temp;
13 }
14
15 int main(int argc, char **argv) {
16  int a = 5, b = 7;
17  printf("BEFORE a=%d b=%d\n", a, b);
18  swap_not_ok(a, b);
19  printf("AFTER a=%d b=%d\n", a, b);
20  printf("BEFORE a=%d b=%d\n", a, b);
21  swap(&a, &b);
22  printf("AFTER a=%d b=%d\n", a, b);
23  return 0;
24 }
```

Κώδικας 1.3: Αντιμετάθεση μεταβλητών με δείκτες (swap1.cpp)

```
1 BEFORE a=5 b=7
2 AFTER a=5 b=7
3 BEFORE a=5 b=7
4 AFTER a=7 b=5
```

Η γλώσσα C++ προκειμένου να απλοποιήσει την κλήση με αναφορά εισήγαγε την έννοια των ψευδωνύμων (aliases). Τοποθετώντας στη δήλωση μιας παραμέτρου συνάρτησης το σύμβολο **&** η παράμετρος λειτουργεί ως ψευδώνυμο για τη μεταβλητή που περνά στην αντίστοιχη θέση. Η συγκεκριμένη συμπεριφορά παρουσιάζεται στον ακόλουθο κώδικα.

```
1 #include <cstdio>
2
3 void swap_cpp(int &x, int &y) {
4   int temp = x;
5   x = y;
6   y = temp;
7 }
8
9 int main(int argc, char **argv) {
10  int a = 5, b = 7;
11  printf("BEFORE a=%d b=%d\n", a, b);
12  swap_cpp(a, b);
13  printf("AFTER a=%d b=%d\n", a, b);
14  return 0;
15 }
```

Κώδικας 1.4: Αντιμετάθεση μεταβλητών με αναφορές (swap2.cpp)

```
1 BEFORE a=5 b=7
2 AFTER a=7 b=5
```

1.4 Πίνακες

Ένας πίνακας είναι μια συλλογή από στοιχεία του ίδιου τύπου καθένα από τα οποία μπορεί να αναγνωριστεί από την τιμή ενός ακεραίου δείκτη (index). Το γεγονός αυτό επιτρέπει την τυχαία προσπέλαση (random access) στα στοιχεία του πίνακα. Οι δείκτες των πινάκων ξεκινούν από το μηδέν.

1.4.1 Μονοδιάστατοι πίνακες

Οι μονοδιάστατοι πίνακες είναι η πλέον απλή δομή δεδομένων. Η αναφορά στα στοιχεία του πίνακα γίνεται συνήθως με μια δομή επανάληψης (π.χ. for). Στο ακόλουθο παράδειγμα δύο μονοδιάστατοι πίνακες αρχικοποιούνται κατά τη δήλωσή τους και εν συνεχεία υπολογίζεται το εσωτερικό γινόμενό τους δηλαδή το άθροισμα των γινομένων των στοιχείων των πινάκων που βρίσκονται στην ίδια θέση.

```
#include <cstdio>

int main(int argc, char **argv) {
   double a[] = {3.2, 4.1, 7.3};
   double b[] = {6.0, .1, −5.3};
   double sum = .0;
   for (int i = 0; i < 3; i++)
      sum += a[i] * b[i];
   printf("inner product %.2f\n", sum);
   return 0;
}
```

Κώδικας 1.5: Υπολογισμός εσωτερικού γινομένου δύο πινάκων (arrays1.cpp)

```
inner product −19.08
```

1.4.2 Δυναμικοί πίνακες

Δυναμικοί πίνακες χρησιμοποιούνται όταν το μέγεθος του πίνακα πρέπει να αλλάζει κατά τη διάρκεια εκτέλεσης του προγράμματος και συνεπώς δεν μπορεί να ορισθεί κατά τη μεταγλώττιση. Πριν χρησιμοποιηθεί ένας δυναμικός πίνακας θα πρέπει δεσμευτούν οι απαιτούμενες θέσεις μνήμης. Επίσης, θα πρέπει να απελευθερωθεί ο χώρος που καταλαμβάνει όταν πλέον δεν χρησιμοποιείται. Στο ακόλουθο παράδειγμα ο χρήστης εισάγει το μέγεθος ενός μονοδιάστατου πίνακα και ο απαιτούμενος χώρος δεσμεύεται κατά την εκτέλεση του κώδικα. Στη συνέχεια ο πίνακας γεμίζει με τυχαίες ακέραιες τιμές στο διάστημα [1,100]. Παρουσιάζονται δύο εκδόσεις του κώδικα, μια που χρησιμοποιεί τις συναρτήσεις malloc και free της γλώσσας C και μια που χρησιμοποιεί τις εντολές new και delete της C++ για τη δέσμευση και την αποδέσμευση μνήμης. Επιπλέον, χρησιμοποιείται διαφορετικός τρόπος για τη δημιουργία των τυχαίων τιμών στα δύο προγράμματα.

```
#include <cstdio>
#include <cstdlib>
#include <ctime>

int main(int argc, char **argv) {
   int n;
   printf("Enter the size of the vector: ");
   fflush(stdout);
   scanf("%d", &n);
   int *a = (int *)malloc(sizeof(int) * n);
   // srand(time(NULL));
   srand(1821);

   for (int i = 0; i < n; i++)
      a[i] = (rand() % 100) + 1;

   for (int i = 0; i < n; i++)
      printf("%d ", a[i]);

   free(a);
   return 0;
}
```

Κώδικας 1.6: Δημιουργία δυναμικού πίνακα με συναρτήσεις της C (arrays2.cpp)

```
Enter the size of the vector: 10
86 72 71 16 32 49 75 35 1 70
```

```
#include <iostream>
#include <random>
using namespace std;
int main(int argc, char **argv) {
   mt19937 mt(1821);
   uniform_int_distribution<int> dist(1, 100);
   int n;
   cout << "Enter the size of the vector: ";
   cin >> n;
   int *a = new int[n];
   for (int i = 0; i < n; i++)
      a[i] = dist(mt);
   for (int i = 0; i < n; i++)
      cout << a[i] << " ";
   delete[] a;
   return 0;
}
```

Κώδικας 1.7: Δημιουργία δυναμικού πίνακα με συναρτήσεις της C++ (arrays3.cpp)

```
Enter the size of the vector: 10
73 44 67 66 1 98 52 85 19 24
```

Για να γίνει η μεταγλώττιση του κώδικα 1.7 θα πρέπει να χρησιμοποιηθεί το flag **-std=c++11** όπως φαίνεται στην ακόλουθη εντολή.

```
g++ arrays3.cpp -o arrays3 -std=c++11
```

1.4.3 Πίνακας ως παράμετρος συνάρτησης και επιστροφή πολλών αποτελεσμάτων

Ένας πίνακας μπορεί να περάσει ως παράμετρος σε μια συνάρτηση. Συχνά χρειάζεται να περάσουν ως παράμετροι και οι διαστάσεις του πίνακα. Στον ακόλουθο κώδικα η συνάρτηση simple_stats δέχεται ως παράμετρο έναν μονοδιάστατο πίνακα ακεραίων και το πλήθος των στοιχείων του και επιστρέφει μέσω κλήσεων με αναφορά το μέσο όρο, το ελάχιστο και το μέγιστο από όλα τα στοιχεία του πίνακα.

```
#include <cstdio>

void simple_stats(int a[], int N, double *avg, int *min, int *max) {
   int sum;
   sum = *min = *max = a[0];
   for (int i = 1; i < N; i++) {
      sum += a[i];
      if (*max < a[i])
         *max = a[i];
      if (*min > a[i])
         *min = a[i];
   }
   *avg = (double)sum / (double)N;
}

int main(int argc, char **argv) {
   int a[] = {3, 2, 9, 5, 1};
   double avg;
   int min, max;
   simple_stats(a, 5, &avg, &min, &max);
   printf("Average= %.2f min=%d max=%d\n", avg, min, max);
```

```
22   return 0;
23 }
```

Κώδικας 1.8: Δυναμικός πίνακας ως παράμετρος συνάρτησης και χρήση δεικτών για επιστροφή τιμών (arrays4.cpp)

```
1 Average= 4.00 min=9 max=1
```

Το ίδιο αποτέλεσμα με τον παραπάνω κώδικα μπορεί να επιτευχθεί απλούστερα χρησιμοποιώντας το μηχανισμό των ψευδωνύμων της C++. Σε αυτή την περίπτωση ο κώδικας θα είναι ο ακόλουθος.

```
1  #include <cstdio>
2
3  void simple_stats(int a[], int N, double &avg, int &min, int &max) {
4    int sum;
5    sum = min = max = a[0];
6    for (int i = 1; i < N; i++) {
7      sum += a[i];
8      if (max < a[i])
9        max = a[i];
10     if (min > a[i])
11       min = a[i];
12   }
13   avg = (double)sum / (double)N;
14 }
15
16 int main(int argc, char **argv) {
17   int a[] = {3, 2, 9, 5, 1};
18   double avg;
19   int min, max;
20   simple_stats(a, 5, avg, min, max);
21   printf("Average= %.2f min=%d max=%d\n", avg, min, max);
22   return 0;
23 }
```

Κώδικας 1.9: Δυναμικός πίνακας ως παράμετρος συνάρτησης και χρήση ψευδωνύμων για επιστροφή τιμών (arrays5.cpp)

1.4.4 Δισδιάστατοι πίνακες

Ένας δισδιάστατος πίνακας αποτελείται από γραμμές και στήλες και η αναφορά στα στοιχεία του γίνεται με δύο δείκτες από τους οποίους ο πρώτος δείκτης υποδηλώνει τη γραμμή και ο δεύτερος υποδηλώνει τη στήλη του πίνακα. Οι πίνακες είναι ιδιαίτερα σημαντικοί για την εκτέλεση μαθηματικών υπολογισμών (π.χ. πολλαπλασιασμό πινάκων, επίλυση συστημάτων γραμμικών εξισώσεων κ.α.). Στον ακόλουθο κώδικα δίνεται ένα παράδειγμα δήλωσης ενός δισδιάστατου πίνακα 5 x 4 ο οποίος περνά ως παράμετρος στη συνάρτηση sums_row_wise. Η δε συνάρτηση επιστρέφει το άθροισμα κάθε γραμμής του πίνακα.

```
1  #include <cstdio>
2
3  void sums_row_wise(int a[][4], int M, int N, int *row) {
4    for (int i = 0; i < M; i++) {
5      row[i] = 0;
6      for (int j = 0; j < N; j++)
7        row[i] += a[i][j];
8    }
9  }
10
11 int main(int argc, char **argv) {
12   int a[5][4] = {{5, 4, 0, -1},
```

```
                    {1, 5, 42, 2},
                    {−3, 7, 8, 2},
                    {7, 312, −56, 6},
                    {19, 45, 6, 5}};
  int row[5];
  sums_row_wise(a, 5, 4, row);
  for (int i = 0; i < 5; i++)
    printf("sum of row %d is %d\n", i, row[i]);
  return 0;
}
```

Κώδικας 1.10: Δισδιάστατος πίνακας ως παράμετρος συνάρτησης (arrays6.cpp)

```
sum of row 0 is 8
sum of row 1 is 50
sum of row 2 is 14
sum of row 3 is 269
sum of row 4 is 75
```

1.4.5 Πολυδιάστατοι πίνακες

Αν και οι μονοδιάστατοι και οι δισδιάστατοι πίνακες χρησιμοποιούνται συχνότερα, υποστηρίζονται και πίνακες μεγαλύτερων διαστάσεων. Στη συνέχεια δίνεται ένα παράδειγμα δήλωσης και αρχικοποίησης ενός τρισδιάστατου πίνακα 3x3x2 και ενός τετραδιάστατου πίνακα 3x3x3x2.

```
int main() {
  int d[3][3][2];
  int e[3][3][2][2];
  for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
      for (int k = 0; k < 2; k++) {
        d[i][j][k] = 1;
        for (int l = 0; l < 2; l++)
          e[i][j][k][l] = 1;
      }
}
```

Κώδικας 1.11: Δήλωση και αρχικοποίηση τρισδιάστατου και τετραδιάστατου πίνακα (arrays7.cpp)

1.4.6 Πριονωτοί πίνακες

Εφόσον ένας πολυδιάστατος πίνακας δημιουργείται δυναμικά μπορεί να οριστεί με τέτοιο τρόπο έτσι ώστε η κάθε γραμμή του να μην έχει τον ίδιο αριθμό στοιχείων. Στον ακόλουθο κώδικα δημιουργείται ένας δισδιάστατος πίνακας 5 γραμμών με την πρώτη γραμμή να έχει 1 στοιχείο και κάθε επόμενη γραμμή ένα περισσότερο στοιχείο από την προηγούμενη της.

```
#include <iostream>
using namespace std;
int main(int argc, char **argv) {
  int *a[5];
  for (int i = 0; i < 5; i++) {
    a[i] = new int[i + 1];
    for (int j = 0; j < i + 1; j++)
      a[i][j] = i + j;
  }
  for (int i = 0; i < 5; i++) {
    for (int j = 0; j < i + 1; j++)
      cout << a[i][j] << " ";
    cout << endl;
```

```
14    }
15    for (int i = 0; i < 5; i++)
16       delete[] a[i];
17    return 0;
18 }
```

Κώδικας 1.12: Παράδειγμα πριονωτού πίνακα με 5 γραμμές (arrays8.cpp)

```
1 0
2 1 2
3 2 3 4
4 3 4 5 6
5 4 5 6 7 8
```

1.5 Δομές

Οι δομές χρησιμοποιούνται όταν απαιτούνται σύνθετοι τύποι δεδομένων οι οποίοι αποτελούνται από επιμέρους στοιχεία. Στο παράδειγμα που ακολουθεί ορίζεται η δομή Book με 3 πεδία. Στη συνέχεια δημιουργούνται 3 μεταβλητές που πρόκειται να αποθηκεύσουν πληροφορίες για ένα βιβλίο η κάθε μια. Η τρίτη μεταβλητή είναι δείκτης προς τη δομή Book και προκειμένου να χρησιμοποιηθεί θα πρέπει πρώτα να δεσμευθεί μνήμη (new) ενώ με τον τερματισμό του προγράμματος θα πρέπει η μνήμη αυτή να επιστραφεί στο σύστημα (delete).

```
1  #include <iostream>
2
3  using namespace std;
4
5  typedef struct {
6     string title;
7     int price;
8     bool isHardpack;
9  } Book;
10
11 // struct Book
12 // {
13 // string title;
14 // int price;
15 // bool isHardpack;
16 // };
17
18 void print_book(Book b) {
19    cout << "Title: " << b.title << " Price: " << b.price / 100.0
20         << " Hardcover: " << (b.isHardpack ? "YES" : "NO") << endl;
21 }
22
23 int main(int argc, char **argv) {
24    Book b1, b2, *pb;
25    b1.title = "The SIMPSONS and their mathematical secrets";
26    b1.price = 1899;
27    b1.isHardpack = false;
28    b2 = b1;
29    b2.isHardpack = true;
30    b2.price = 2199;
31    pb = new Book;
32    pb->title = "Bad Science";
33    pb->price = 999;
34    pb->isHardpack = false;
35    print_book(b1);
36    print_book(b2);
37    print_book(*pb);
```

```
38   delete pb;
39   return 0;
40 }
```

Κώδικας 1.13: Μεταβλητές τύπου δομής Book (structs1.cpp)

```
1 Title: The SIMPSONS and their mathematical secrets Price: 18.99 Hardcover: NO
2 Title: The SIMPSONS and their mathematical secrets Price: 21.99 Hardcover: YES
3 Title: Bad Science Price: 9.99 Hardcover: NO
```

1.6 Κλάσεις - Αντικείμενα

Ο αντικειμενοστρεφής προγραμματισμός εντοπίζει τα αντικείμενα που απαρτίζουν την εφαρμογή και τα συνδυάζει προκειμένου να επιτευχθεί η απαιτούμενη λειτουργικότητα. Για κάθε αντικείμενο γράφεται μια κλάση η οποία είναι υπεύθυνη για τη δημιουργία των επιμέρους στιγμιοτύπων (object instances). Κάθε αντικείμενο έχει μεταβλητές και συναρτήσεις οι οποίες μπορεί να είναι είτε ιδιωτικές (private) είτε δημόσιες (public) (είτε προστατευμένες-protected). Τα ιδιωτικά μέλη χρησιμοποιούνται εντός της κλάσης που ορίζει το αντικείμενο ενώ τα δημόσια μπορούν να χρησιμοποιηθούν και από κώδικα εκτός της κλάσης. Στο ακόλουθο παράδειγμα ορίζεται η κλάση Box η οποία έχει 3 ιδιωτικά μέλη (length, width, height) και 1 δημόσιο μέλος, τη συνάρτηση volume. Επιπλέον η κλάση Box διαθέτει έναν κατασκευαστή (constructor) που δέχεται τρεις παραμέτρους και μπορεί να χρησιμοποιηθεί για τη δημιουργία νέων αντικειμένων. Στη main δημιουργούνται με τη βοήθεια του κατασκευαστή δύο αντικείμενα (στιγμιότυπα) της κλάσης Box και καλείται για καθένα από αυτά η δημόσια συνάρτηση μέλος της Box, volume. Θα πρέπει να σημειωθεί ότι η πρόσβαση στα μέλη δεδομένων (length, width, height) δεν είναι εφικτή από συναρτήσεις εκτός της κλάσης καθώς τα μέλη αυτά είναι ιδιωτικά. Ωστόσο, αν η πρόσβαση στα ιδιωτικά μέλη ήταν επιθυμητή τότε θα έπρεπε να κατασκευαστούν δημόσιες συναρτήσεις μέλη (getters και setters) έτσι ώστε να δοθεί έμμεση πρόσβαση μέσω αυτών στα ιδιωτικά μέλη της κλάσης.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Box {
6  public:
7    // constructor declaration with default values
8    Box(double l = 1, double w = 1, double h = 1) {
9      length = l;
10     width = w;
11     height = h;
12   };
13   // member function
14   double volume() { return length * width * height; }
15
16 private:
17   // member data
18   double length;
19   double width;
20   double height;
21 };
22
23 int main(int argc, char **argv) {
24   Box b1(10, 5, 10);
25   cout << "The volume is " << b1.volume() << endl;
26   Box *pb = new Box();
27   cout << "The volume is " << pb->volume() << endl;
28   delete pb;
```

```
29   return 0;
30 }
```

Κώδικας 1.14: Παράδειγμα κλάσης Box (objects1.cpp)

```
1 The volume is 500
2 The volume is 1
```

Εναλλακτικά, ο κώδικας του προηγούμενου παραδείγματος μπορεί να γραφτεί όπως παρακάτω, πραγματοποιώντας τη συγγραφή του σώματος των συναρτήσεων της κλάσης Box μετά τη δήλωση της κλάσης και χρησιμοποιώντας λίστα αρχικοποίησης τιμών (initializer list) στον κατασκευαστή της κλάσης.

```
1  #include <iostream>
2
3  using namespace std;
4
5  class Box {
6  public:
7    // constructor declaration with default values
8    Box(double l = 1, double w = 1, double h = 1);
9    // member function
10   double volume();
11
12 private:
13   // member data
14   double length;
15   double width;
16   double height;
17 };
18
19 // constructor using initializer list
20 Box::Box(double l, double w, double h) : length(l), width(w), height(h) {}
21
22 double Box::volume() { return length * width * height; }
23
24 int main(int argc, char **argv) {
25   Box b1(10, 5, 10);
26   cout << "The volume is " << b1.volume() << endl;
27   Box *pb = new Box();
28   cout << "The volume is " << pb->volume() << endl;
29   delete pb;
30   return 0;
31 }
```

Κώδικας 1.15: Παράδειγμα κλάσης Box (objects2.cpp)

1.7 Αρχεία

Συχνά χρειάζεται να αποθηκεύσουμε δεδομένα σε αρχεία ή να επεξεργαστούμε δεδομένα τα οποία βρίσκονται σε αρχεία. Ο ακόλουθος κώδικας πρώτα δημιουργεί έναν αρχείο με 100 τυχαίους ακεραίους στον τρέχοντα κατάλογο και στη συνέχεια ανοίγει το αρχείο και εμφανίζει τα στοιχεία του.

1.7.1 Εγγραφή και ανάγνωση δεδομένων από αρχείο με συναρτήσεις της C

```
1 #include <cstdio>
2 #include <cstdlib>
3 #include <ctime>
4
```

```
int main(int argc, char **argv) {
  FILE *fp = fopen("data_int_100.txt", "w");
  srand(time(NULL));
  for (int i = 0; i < 100; i++)
    fprintf(fp, "%d ", rand() % 1000 + 1);
  fclose(fp);

  fp = fopen("data_int_100.txt", "r");
  if (fp == NULL) {
    printf("error opening file");
    exit(-1);
  }
  int x;
  while (!feof(fp)) {
    fscanf(fp, "%d", &x);
    printf("%d ", x);
  }
  fclose(fp);
  return 0;
}
```

Κώδικας 1.16: Εγγραφή 100 ακέραιων αριθμητικών δεδομένων σε αρχείο και ανάγνωση τους από το ίδιο αρχείο (files1.cpp)

```
811 718 632 412 529 957 359 735 498 302 855 265 749 756 336 625 489 870 120 177 ...
```

1.7.2 Εγγραφή και ανάγνωση δεδομένων από αρχείο με συναρτήσεις της C++

Η C++ έχει προσθέσει νέους τρόπους με τους οποίους μπορεί να γίνει η αλληλεπίδραση με τα αρχεία. Ακολουθεί ένα παράδειγμα εγγραφής και ανάγνωσης δεδομένων από αρχείο με τη χρήση των fstream και sstream.

```
#include <fstream>
#include <iomanip>
#include <iostream>
#include <sstream>

using namespace std;

int main(int argc, char **argv) {
  int constexpr N = 10;
  fstream filestr;
  string buffer;
  int i = 0;
  string names[N] = {"nikos", "maria", "petros", "sofia", "kostas",
                     "dimitra", "giorgos", "christos", "anna", "apostolis"};
  int grades[N] = {55, 30, 70, 80, 10, 25, 75, 90, 100, 30};
  filestr.open("data_student_struct10.txt", ios::out);
  if (!filestr.is_open()) {
    cerr << "file not found" << std::endl;
    exit(-1);
  }
  for (i = 0; i < N; i++)
    filestr << names[i] << "\t" << grades[i] << endl;
  filestr.close();

  filestr.open("data_student_struct10.txt");
  if (!filestr.is_open()) {
    cerr << "file not found" << std::endl;
    exit(-1);
  }
```

```
30    string name;
31    int grade;
32    while (getline(filestr, buffer)) {
33       stringstream ss(buffer);
34       ss >> name;
35       ss >> grade;
36       cout << name << " " << setprecision(1) << fixed
37             << static_cast<double>(grade) / 10.0 << endl;
38    }
39    filestr.close();
40    return 0;
41 }
```

Κώδικας 1.17: Εγγραφή και ανάγνωση αλφαριθμητικών και ακεραίων από αρχείο (files2.cpp)

```
1  nikos 5.5
2  maria 3.0
3  petros 7.0
4  sofia 8.0
5  kostas 1.0
6  dimitra 2.5
7  giorgos 7.5
8  christos 9.0
9  anna 10.0
10 apostolis 3.0
```

1.8 Παραδείγματα

1.8.1 Παράδειγμα 1

Γράψτε κώδικα που να δημιουργεί μια δομή με όνομα Point και να έχει ως πεδία 2 double αριθμούς (x και y) που υποδηλώνουν τις συντεταγμένες ενός σημείου στο καρτεσιανό επίπεδο. Δημιουργήστε έναν πίνακα με όνομα points με 5 σημεία με απευθείας εισαγωγή τιμών για τα ακόλουθα σημεία: (4, 17), (10, 21), (5, 32), (-1, 16), (-4, 7). Γράψτε τον κώδικα που εμφανίζει τα 2 πλησιέστερα σημεία. Ποια είναι τα πλησιέστερα σημεία και ποια η απόσταση μεταξύ τους;

```
1  #include <climits>
2  #include <cmath>
3  #include <cstdio>
4
5  struct Point {
6     double x;
7     double y;
8  };
9
10 int main(int argc, char **argv) {
11    struct Point points[5] = {{4, 17}, {10, 21}, {5, 32}, {−1, 16}, {−4, 7}};
12
13    double min = INT_MAX;
14    Point a, b;
15    for (int i = 0; i < 5; i++)
16       for (int j = i + 1; j < 5; j++) {
17          Point p1 = points[i];
18          Point p2 = points[j];
19          double distance = sqrt(pow(p2.x − p1.x, 2.0) + pow(p2.y − p1.y, 2.0));
20          if (distance < min) {
21             min = distance;
22             a = p1;
23             b = p2;
24          }
```

```
25      }
26    printf("Min %.4f\n", min);
27    printf("Point A: (%.4f, %.4f)\n", a.x, a.y);
28    printf("Point B: (%.4f, %.4f)\n", b.x, b.y);
29    return 0;
30  }
```

Κώδικας 1.18: Λύση παραδείγματος 1 (lab01_ex1.cpp)

```
1  Min 5.0990
2  Point A: (4.0000, 17.0000)
3  Point B: (−1.0000, 16.0000)
```

1.8.2 Παράδειγμα 2

Με τη γεννήτρια τυχαίων αριθμών mt19937 δημιουργήστε 10000 τυχαίες ακέραιες τιμές στο διάστημα 0 έως 10000 με seed την τιμή 1729. Τοποθετήστε τις τιμές σε ένα δισδιάστατο πίνακα 100 x 100 έτσι ώστε να συμπληρώνονται οι τιμές στον πίνακα κατά σειρές από πάνω προς τα κάτω και από αριστερά προς τα δεξιά. Να υπολογιστεί το άθροισμα της κάθε γραμμής του πίνακα. Ποιος είναι ο αριθμός της γραμμής με το μεγαλύτερο άθροισμα και ποιο είναι αυτό;

```
1  #include <iostream>
2  #include <random>
3
4  using namespace std;
5
6  int main(int argc, char **argv) {
7    int seed = 1729;
8    int a[100][100];
9    mt19937 mt(seed);
10   uniform_int_distribution<int> dist(0, 10000);
11   for (int i = 0; i < 100; i++)
12     for (int j = 0; j < 100; j++)
13       a[i][j] = dist(mt);
14   int b[100];
15   int max = 0;
16   for (int i = 0; i < 100; i++) {
17     int sum = 0;
18     for (int j = 0; j < 100; j++)
19       sum += a[i][j];
20     b[i] = sum;
21     if (sum > max)
22       max = sum;
23   }
24   cout << "Maximum row sum: " << max << endl;
25   for (int i = 0; i < 100; i++)
26     if (b[i] == max)
27       cout << "Occurs at row: " << i << endl;
28   return 0;
29 }
```

Κώδικας 1.19: Λύση παραδείγματος 2 (lab01_ex2.cpp)

```
1  Maximum row sum: 586609
2  Occurs at row: 40
```

1.8.3 Παράδειγμα 3

Γράψτε 10000 τυχαίες ακέραιες τιμές στο διάστημα [1,10000] στο αρχείο data_int_10000.txt χρησιμοποιώντας τις συναρτήσεις rand και srand και seed την τιμή 1729. Διαβάστε τις τιμές από το αρχείο. Εντοπίστε τη μεγαλύτερη τιμή στα δεδομένα. Ποιες είναι οι τιμές που εμφανίζονται τις περισσότερες φορές στα δεδομένα;

```
#include <cstdio>
#include <cstdlib>
#include <iostream>


using namespace std;

int main(int argc, char **argv) {
  const char *fn = "data_int_10000.txt";
  int N = 10000;
  srand(1729);

  FILE *fp = fopen(fn, "w");
  if (fp == NULL) {
    printf("error opening file");
    exit(-1);
  }
  for (int i = 0; i < N; i++)
    fprintf(fp, "%d ", rand() % 10000 + 1);
  fclose(fp);

  fp = fopen(fn, "r");
  if (fp == NULL) {
    printf("error opening file");
    exit(-1);
  }
  int *data = new int[N];
  int i = 0;
  while (!feof(fp)) {
    fscanf(fp, "%d", &data[i]);
    i++;
  }
  fclose(fp);
  int max = data[0];
  for (i = 1; i < N; i++)
    if (data[i] > max)
      max = data[i];
  printf("Maximum value %d\n", max);
  int *freq = new int[max + 1];
  for (i = 0; i < max + 1; i++)
    freq[i] = 0;
  for (i = 0; i < N; i++)
    freq[data[i]]++;
  int max2 = 0;
  for (i = 0; i < max + 1; i++)
    if (freq[i] > max2)
      max2 = freq[i];
  for (i = 0; i < max + 1; i++)
    if (freq[i] == max2)
      printf("Value %d exists %d times\n", i, max2);
  delete[] freq;
  return 0;
}
```

Κώδικας 1.20: Λύση παραδείγματος 3 (lab01_ex3.cpp)

```
Maximum value 9999
Value 885 exists 6 times
Value 1038 exists 6 times
Value 3393 exists 6 times
Value 4771 exists 6 times
Value 5482 exists 6 times
Value 8722 exists 6 times
Value 9501 exists 6 times
```

1.8.4 Παράδειγμα 4

Γράψτε κώδικα που να δημιουργεί μια δομή με όνομα student (σπουδαστής) και να έχει ως πεδία το name (όνομα) τύπου string και το grade (βαθμός) τύπου int. Διαβάστε τα περιεχόμενα του αρχείου που έχει δημιουργηθεί με τον κώδικα 1.17 (data_student_struct10.txt) και τοποθετήστε τα σε κατάλληλο πίνακα. Βρείτε τα ονόματα και το μέσο όρο βαθμολογίας των σπουδαστών με βαθμό άνω του μέσου όρου όλων των σπουδαστών. Θεωρείστε ότι οι βαθμοί έχουν αποθηκευτεί στο αρχείο data_student_struct10.txt ως ακέραιοι αριθμοί από το 0 μέχρι και το 100, αλλά η εμφάνισή τους θα πρέπει να γίνεται εφόσον πρώτα διαιρεθούν με το 10. Δηλαδή, ο βαθμός 55 αντιστοιχεί στο βαθμό 5.5.

```
#include <fstream>
#include <iostream>
#include <sstream>

using namespace std;

struct student {
   string name;
   int grade;
};

int main(int argc, char **argv) {
   constexpr int N = 10;
   int i = 0;
   student students[N];
   const char *fn = "data_student_struct10.txt";
   fstream filestr;
   string buffer;
   filestr.open(fn);
   if (!filestr.is_open()) {
      cerr << "File not found" << std::endl;
      exit(-1);
   }
   while (getline(filestr, buffer)) {
      stringstream ss(buffer);
      ss >> students[i].name;
      ss >> students[i].grade;
      i++;
   }
   filestr.close();
   double sum = 0.0;
   for (i = 0; i < N; i++)
      sum += students[i].grade;
   double avg = sum / N;
   cout << "Average grade =" << avg / 10.0 << endl;

   double sum2 = 0.0;
   int c = 0;
   for (i = 0; i < N; i++)
      if (students[i].grade > avg) {
```

```
41          cout << students[i].name << " grade = " << students[i].grade / 10.0
42                  << endl;
43          sum2 += students[i].grade;
44          c++;
45        }
46    double avg2 = sum2 / c;
47    cout << "Average grade for students having grade above the average grade = "
48           << avg2 / 10.0 << endl;
49    return 0;
50  }
```

Κώδικας 1.21: Λύση παραδείγματος 4 (lab01_ex4.cpp)

```
1  Average grade =5.65
2  petros grade = 7
3  sofia grade = 8
4  giorgos grade = 7.5
5  christos grade = 9
6  anna grade = 10
7  Average grade for students having grade above the average grade = 8.3
```

1.9 Ασκήσεις

1. Γράψτε μια συνάρτηση που να δέχεται έναν πίνακα ακεραίων και το μέγεθός του και να επιστρέφει το μέσο όρο των τιμών καθώς και το πλήθος των τιμών που απέχουν το πολύ 10% από το μέσο όρο. Δοκιμάστε την κλήση της συνάρτησης για έναν πίνακα 100 θέσεων με τυχαίες ακέραιες τιμές στο διάστημα [1,100] οι οποίες θα δημιουργηθούν με τη χρήση των συναρτήσεων srand() και rand() της C. Χρησιμοποιήστε ως seed για την αρχικοποίηση των τυχαίων τιμών την τιμή 12345.

2. Γράψτε πρόγραμμα που να διαβάζει τα στοιχεία υπαλλήλων (όνομα, μισθό και έτη προϋπηρεσίας) από το αρχείο data_ypallhlos_struct20.txt και να εμφανίζει τα στοιχεία του κάθε υπαλλήλου μέσω μιας συνάρτησης που θα δέχεται ως παράμετρο μια μεταβλητή τύπου δομής υπαλλήλου. Στη συνέχεια να υπολογίζει και να εμφανίζει το ποσό που θα συγκεντρωθεί αν για κάθε υπάλληλο με περισσότερα από 5 έτη προϋπηρεσίας παρακρατηθεί το 5% του μισθού του ενώ για τους υπόλοιπους υπαλλήλους παρακρατηθεί το 7% του μισθού τους.

3. Γράψτε το προηγούμενο πρόγραμμα ξανά χρησιμοποιώντας κλάση στη θέση της δομής. Επιπλέον ορίστε constructor και getters/setters για τα μέλη δεδομένων του αντικειμένου υπάλληλος.

4. Γράψτε ένα πρόγραμμα που να γεμίζει έναν πίνακα με όνομα a, 5 γραμμών και 5 στηλών, με τυχαίες ακέραιες τιμές στο διάστημα 1 έως και 1000 (χρησιμοποιήστε ως seed την τιμή 12345). Γράψτε μια συνάρτηση που να δέχεται ως παράμετρο τον πίνακα a και να επιστρέφει σε μονοδιάστατο πίνακα με όνομα col το άθροισμα των τιμών κάθε στήλης του πίνακα. Οι τιμές που επιστρέφονται να εμφανίζονται στο κύριο πρόγραμμα το οποίο να εμφανίζει επιπλέον και τον αριθμό στήλης με το μεγαλύτερο άθροισμα.

Βιβλιογραφία

[1] Σταμάτης Σταματιάδης. Εισαγωγή στη γλώσσα προγραμματισμού C++11. Τμήμα Επιστήμης και Τεχνολογίας Υλικών, Πανεπιστήμιο Κρήτης, 2017, https://www.materials.uoc.gr/el/undergrad/courses/ETY215/notes.pdf.

[2] Allen B. Downey. How to think like a computer scientist, C++ version, 2012, http://www.greenteapress.com/thinkcpp/.

[3] Juan Soulié. C++ Language Tutorial. cplusplus.com, 2007, http://www.cplusplus.com/files/tutorial.pdf.

[4] Brian Hall. Beej's Guide to C Programming, 2007, http://beej.us/guide/bgc/.

Εργαστήριο 2

Εισαγωγή στα templates, στην STL και στα lambdas - TDD

2.1 Εισαγωγή

Στο εργαστήριο αυτό παρουσιάζεται ο μηχανισμός των templates, τα lambdas και οι βασικές δυνατότητες της βιβλιοθήκης STL (Standard Template Library) της C++. Τα templates επιτρέπουν την κατασκευή γενικού κώδικα επιτρέποντας την αποτύπωση της λογικής μιας συνάρτησης ανεξάρτητα από τον τύπο των ορισμάτων που δέχεται. Από την άλλη μεριά, η βιβλιοθήκη STL, στην οποία γίνεται εκτεταμένη χρήση των templates παρέχει στον προγραμματιστή έτοιμη λειτουργικότητα για πολλές ενέργειες που συχνά συναντώνται κατά την ανάπτυξη εφαρμογών. Τέλος, γίνεται μια σύντομη αναφορά στην τεχνική ανάπτυξης προγραμμάτων TDD η οποία εξασφαλίζει σε κάποιο βαθμό την ανάπτυξη προγραμμάτων με ορθή λειτουργία εξαναγκάζοντας τους προγραμματιστές να ενσωματώσουν τη δημιουργία ελέγχων στον κώδικα που παράγουν καθημερινά. Επιπλέον υλικό για την STL βρίσκεται στις αναφορές [1], [2], [3], [4].

2.2 Templates

Τα templates (πρότυπα) είναι ένας μηχανισμός της C++ ο οποίος μπορεί να διευκολύνει τον προγραμματισμό. Η γλώσσα C++ είναι statically typed το οποίο σημαίνει ότι οι τύποι δεδομένων των μεταβλητών και σταθερών ελέγχονται κατά τη μεταγλώττιση. Το γεγονός αυτό μπορεί να οδηγήσει στην ανάγκη υλοποίησης διαφορετικών εκδόσεων μιας συνάρτησης έτσι ώστε να υποστηριχθεί η ίδια λογική για διαφορετικούς τύπους δεδομένων. Για παράδειγμα, η εύρεση της ελάχιστης τιμής ανάμεσα σε τρεις τιμές θα έπρεπε να υλοποιηθεί με δύο συναρτήσεις έτσι ώστε να υποστηρίζει τόσο τους ακεραίους όσο και τους πραγματικούς αριθμούς, όπως φαίνεται στον κώδικα που ακολουθεί.

```
1  #include <iostream>
2  using namespace std;
3
4  int min(int a, int b, int c) {
5     int m = a;
6     if (b < m)
7        m = b;
8     if (c < m)
9        m = c;
10    return m;
11 }
12
13 double min(double a, double b, double c) {
14    double m = a;
15    if (b < m)
```

```
16      m = b;
17    if (c < m)
18      m = c;
19    return m;
20  }
21
22  int main(int argc, char *argv[]) {
23    cout << "The minimum among 3 integer numbers is " << min(5, 10, 7) << endl;
24    cout << "The minimum among 3 real numbers is " << min(3.1, 0.7, 2.5) << endl;
25  }
```

Κώδικας 2.1: Επανάληψη λογικής κώδικα (minmax.cpp)

```
1  The minimum among 3 integer numbers is 5
2  The minimum among 3 real numbers is 0.7
```

Με τη χρήση των templates μπορεί να γραφεί κώδικας που να υποστηρίζει ταυτόχρονα πολλούς τύπους δεδομένων. Ειδικότερα, χρησιμοποιείται, η δεσμευμένη λέξη template και εντός των συμβόλων < και > τοποθετείται η λίστα των παραμέτρων του template. Ο μεταγλωττιστής αναλαμβάνει να δημιουργήσει όλες τις απαιτούμενες παραλλαγές των συναρτήσεων που θα χρειαστούν στον κώδικα που μεταγλωττίζει.

```
1  #include <iostream>
2  using namespace std;
3
4  template <typename T> T min(T a, T b, T c) {
5    T m = a;
6    if (b < m)
7      m = b;
8    if (c < m)
9      m = c;
10   return m;
11 }
12
13 int main(int argc, char *argv[]) {
14   cout << "The minimum among 3 integer numbers is " << min(5, 10, 7) << endl;
15   cout << "The minimum among 3 real numbers is " << min(3.1, 0.7, 2.5) << endl;
16 }
```

Κώδικας 2.2: Χρήση template για αποφυγή επανάληψης λογικής κώδικα (minmaxt.cpp)

```
1  The minimum among 3 integer numbers is 5
2  The minimum among 3 real numbers is 0.7
```

θα πρέπει να σημειωθεί ότι στη θέση της δεσμευμένης λέξης typename μπορεί εναλλακτικά να χρησιμοποιηθεί η δεσμευμένη λέξη class.

2.3 Η βιβλιοθήκη STL

Η βιβλιοθήκη STL (Standard Template Library) της C++ προσφέρει έτοιμη λειτουργικότητα για πολλά θέματα τα οποία ανακύπτουν συχνά στον προγραμματισμό εφαρμογών. Πρόκειται για μια generic βιβλιοθήκη, δηλαδή κάνει εκτεταμένη χρήση των templates. Βασικά τμήματα της STL είναι οι containers (υποδοχείς), οι iterators (επαναλήπτες) και οι αλγόριθμοι.

2.3.1 Containers

Η STL υποστηρίζει έναν αριθμό από containers στους οποίους μπορούν να αποθηκευτούν δεδομένα. Ένα από τα containers είναι το vector (διάνυσμα). Στον ακόλουθο κώδικα φαίνεται πως η χρήση του vector διευκολύνει τον προγραμματισμό καθώς δεν απαιτούνται εντολές διαχείρισης μνήμης ενώ η δομή είναι δυναμική, δηλαδή το μέγεθος της μπορεί να μεταβάλλεται κατά τη διάρκεια εκτέλεσης του προγράμματος.

```cpp
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[]) {
   int x;
   cout << "Enter the size of the vector: ";
   cin >> x;
   vector<int> v(x);
   for (int i = 0; i < x; i++)
      v[i] = i;
   v.push_back(99);
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
}
```

Κώδικας 2.3: Παράδειγμα με προσθήκη στοιχείων σε vector (container1.cpp)

```
Enter size of vector: 10
0 1 2 3 4 5 6 7 8 9 99
```

Ένα container τύπου vector μπορεί να λάβει τιμές με πολλούς τρόπους. Στον ακόλουθο κώδικα παρουσιάζονται έξι διαφορετικοί τρόποι με τους οποίους μπορεί να γίνει αυτό.

```cpp
#include <iostream>
#include <vector>

using namespace std;

void print_vector(const string &name, const vector<int> &v) {
   cout << name << ": ";
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << endl;
}

int main(int argc, char *argv[]) {
   vector<int> v1;
   v1.push_back(5);
   v1.push_back(16);
   print_vector("v1", v1);

   vector<int> v2 = {16, 3, 6, 1, 9, 10};
   print_vector("v2", v2);

   vector<int> v3{5, 2, 10, 1, 8};
   print_vector("v3", v3);

   vector<int> v4(5, 10);
   print_vector("v4", v4);

   vector<int> v5(v2);
   print_vector("v5", v5);

   vector<int> v6(v2.begin() + 1, v2.end() - 1);
   print_vector("v6", v6);
}
```

Κώδικας 2.4: Αρχικοποίηση vectors (container2.cpp)

```
v1: 5 16
v2: 16 3 6 1 9 10
v3: 5 2 10 1 8
v4: 10 10 10 10 10
v5: 16 3 6 1 9 10
v6: 3 6 1 9
```

Τα containers χωρίζονται σε σειριακά (sequence containers) και συσχετιστικά (associative containers). Τα σειριακά containers είναι συλλογές ομοειδών στοιχείων στις οποίες κάθε στοιχείο έχει συγκεκριμένη θέση μέσω της οποίας μπορούμε να αναφερθούμε σε αυτό. Τα σειριακά containers είναι τα εξής:

- array (πίνακας)
- deque (ουρά με δύο άκρα)
- forward_list (λίστα διανυόμενη προς τα εμπρός)
- list (λίστα)
- vector (διάνυσμα)

Τα συσχετιστικά containers παρουσιάζουν το πλεονέκτημα της γρήγορης προσπέλασης. Συσχετιστικά containers της STL είναι τα εξής:

- map (λεξικό)
- unordered_map (λεξικό χωρίς σειρά)
- multimap (πολλαπλό λεξικό)
- unordered_multimap. (πολλαπλό λεξικό χωρίς σειρά)
- set (σύνολο)
- unordered_set (σύνολο χωρίς σειρά)
- multiset (πολλαπλό σύνολο)
- unordered_multiset (πολλαπλό σύνολο χωρίς σειρά)

Στη συνέχεια παρουσιάζεται ένα παράδειγμα χρήσης του συσχετιστικού container map. Δημιουργείται ένας τηλεφωνικός κατάλογος που περιέχει πληροφορίες της μορφής όνομα - τηλέφωνο και ο οποίος δίνει τη δυνατότητα να αναζητηθεί ένα τηλέφωνο με βάση ένα όνομα.

```cpp
#include <iostream>
#include <map>
#include <string>
using namespace std;

int main(int argc, char *argv[]) {
  map<string, string> phone_book;
  phone_book.insert(make_pair("nikos", "1234567890"));
  phone_book.insert(make_pair("maria", "2345678901"));
  phone_book.insert(make_pair("petros", "3456789012"));
  phone_book.insert(make_pair("kostas", "4567890123"));

  string name;
  cout << "Enter name: ";
  cin >> name;
  if (phone_book.find(name) == phone_book.end())
    cout << "No such name found ";
  else
    cout << "The phone is " << phone_book[name] << endl;
}
```

Κώδικας 2.5: Παράδειγμα με map (container3.cpp)

```
Enter name: nikos
The phone is 1234567890
```

2.3.2 Iterators

Οι iterators αποτελούν γενικεύσεις των δεικτών και επιτρέπουν την πλοήγηση στα στοιχεία ενός container με τέτοιο τρόπο έτσι ώστε να μπορούν να χρησιμοποιηθούν οι ίδιοι αλγόριθμοι σε περισσότερα του ενός containers. Στον ακόλουθο κώδικα παρουσιάζεται το πέρασμα από τα στοιχεία ενός vector με τέσσερις διαφορετικούς τρόπους. Καθώς το container είναι τύπου vector παρουσιάζεται αρχικά το πέρασμα από τις τιμές του με τη χρήση δεικτοδότησης τύπου πίνακα. Στη συνέχεια χρησιμοποιείται η πρόσβαση στα στοιχεία του container μέσω του range for. Ακολούθως, χρησιμοποιείται ένας iterator για πέρασμα από την αρχή προς το τέλος και ένας reverse_iterator για πέρασμα από το τέλος προς την αρχή.

```
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char **argv)
{
   vector<int> v = {23, 13, 31, 17, 56};
   cout << "iteration using index: ";
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << " ";
   cout << endl;

   cout << "iteration using ranged based for: ";
   for (int x : v)
      cout << x << " ";
   cout << endl;

   cout << "forward iteration using iterator: ";
   vector<int>::iterator iter;
   for (iter = v.begin(); iter != v.end(); iter++)
      cout << *iter << " ";
   cout << endl;

   cout << "backward iteration using iterator: ";
   vector<int>::reverse_iterator riter;
   for (riter = v.rbegin(); riter != v.rend(); riter++)
      cout << *riter << " ";
   cout << endl;
}
```

Κώδικας 2.6: Τέσσερις διαφορετικοί τρόποι προσπέλασης των στοιχείων ενός vector (iterator1.cpp)

```
iteration using index: 23 13 31 17 56
iteration using ranged based for: 23 13 31 17 56
forward iteration using iterator: 23 13 31 17 56
backward iteration using iterator: 56 17 31 13 23
```

Ακολουθεί κώδικας στον οποίο παρουσιάζεται το πέρασμα από όλα τα στοιχεία ενός map με τρεις διαφορετικούς τρόπους. Ο πρώτος τρόπος χρησιμοποιεί range for. Ο δεύτερος έναν iterator και ο τρίτος έναν reverse iterator.

```
#include <iostream>
#include <map>

using namespace std;
int main(int argc, char *argv[]) {
   map<string, int> cities_population_2011 = {{"arta", 21895},
                                              {"ioannina", 65574},
                                              {"preveza", 19042},
                                              {"igoumenitsa", 9145}};
```

```
10
11    cout << "Cities of Epirus using range for:" << endl;
12    for (auto kv : cities_population_2011)
13        cout << kv.first << " " << kv.second << endl;
14
15    cout << "Cities of Epirus using iterator:" << endl;
16    for (map<string, int>::iterator iter = cities_population_2011.begin();
17            iter != cities_population_2011.end(); iter++)
18        cout << iter->first << " " << iter->second << endl;
19
20    cout << "Cities of Epirus using reverse iterator:" << endl;
21    for (map<string, int>::reverse_iterator iter = cities_population_2011.rbegin();
22            iter != cities_population_2011.rend(); iter++)
23        cout << iter->first << " " << iter->second << endl;
24 }
```

Κώδικας 2.7: Τρεις διαφορετικοί τρόποι προσπέλασης των στοιχείων ενός map (iterator2.cpp)

```
1  Cities of Epirus using range for:
2  arta 21895
3  igoumenitsa 9145
4  ioannina 65574
5  preveza 19042
6  Cities of Epirus using iterator:
7  arta 21895
8  igoumenitsa 9145
9  ioannina 65574
10 preveza 19042
11 Cities of Epirus using reverse iterator:
12 preveza 19042
13 ioannina 65574
14 igoumenitsa 9145
15 arta 21895
```

2.3.3 Αλγόριθμοι

Η STL διαθέτει πληθώρα αλγορίθμων που μπορούν να εφαρμοστούν σε διάφορα προβλήματα. Για παράδειγμα, προκειμένου να ταξινομηθούν δεδομένα μπορεί να χρησιμοποιηθεί η συνάρτηση sort της STL η οποία υλοποιεί τον αλγόριθμο Introspective Sort. Στον ακόλουθο κώδικα πραγματοποιείται η ταξινόμηση αρχικά ενός στατικού πίνακα και στη συνέχεια εφόσον πρώτα οι τιμές του πίνακα μεταφερθούν σε ένα vector και ανακατευτούν τυχαία, πρώτα ταξινομούνται σε αύξουσα και μετά σε φθίνουσα σειρά.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <random>
4  #include <vector>
5
6  using namespace std;
7
8  int main(int argc, char **argv) {
9      cout << "### STL Sort Example ###" << endl;
10     int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
11     cout << "BEFORE (static array example): ";
12     for (int i = 0; i < 10; i++)
13         cout << a[i] << " ";
14     cout << endl;
15     sort(a, a + 10);
16     cout << "AFTER: ";
17     for (int i = 0; i < 10; i++)
18         cout << a[i] << " ";
19     cout << endl;
```

```
20
21    cout << "BEFORE (vector example 1): ";
22    vector<int> va(a, a + 10);
23    auto rng = default_random_engine{};
24    shuffle(va.begin(), va.end(), rng);
25    for (auto it = va.begin(); it < va.end(); it++)
26       cout << *it << " ";
27    cout << endl;
28    sort(va.begin(), va.end());
29    cout << "AFTER: ";
30    for (auto it = va.begin(); it < va.end(); it++)
31       cout << *it << " ";
32    cout << endl;
33
34    cout << "BEFORE (vector example 2): ";
35    shuffle(va.begin(), va.end(), rng);
36    for (auto it = va.begin(); it < va.end(); it++)
37       cout << *it << " ";
38    cout << endl;
39    // descending
40    sort(va.begin(), va.end(), greater<int>());
41    cout << "AFTER: ";
42    for (auto it = va.begin(); it < va.end(); it++)
43       cout << *it << " ";
44    cout << endl;
45    return 0;
46 }
```

Κώδικας 2.8: Ταξινόμηση με τη συνάρτηση sort της STL (sort1.cpp)

```
1  ### STL Sort Example ###
2  BEFORE (static array example): 45 32 16 11 7 18 21 16 11 15
3  AFTER: 7 11 11 15 16 16 18 21 32 45
4  BEFORE (vector example 1): 21 18 16 16 11 15 45 11 7 32
5  AFTER: 7 11 11 15 16 16 18 21 32 45
6  BEFORE (vector example 2): 45 11 15 11 21 7 32 16 16 18
7  AFTER: 45 32 21 18 16 16 15 11 11 7
```

Στον παραπάνω κώδικά έγινε χρήση της δεσμευμένης λέξης auto στη δήλωση μεταβλητών. Η λέξη auto μπορεί να χρησιμοποιηθεί στη θέση ενός τύπου δεδομένων όταν γίνεται ταυτόχρονα δήλωση και ανάθεση τιμής σε μια μεταβλητή. Σε αυτή την περίπτωση ο μεταγλωττιστής της C++ είναι σε θέση να αναγνωρίσει τον πραγματικό τύπο της μεταβλητής από την τιμή που της εκχωρείται.

Η συνάρτηση sort() εφαρμόζεται σε sequence containers πλην των list και forward_list στα οποία δεν μπορεί να γίνει απευθείας πρόσβαση σε στοιχεία τους χρησιμοποιώντας ακεραίους αριθμούς για τον προσδιορισμό της θέσης τους. Ειδικά για αυτά τα containers υπάρχει η συνάρτηση μέλος sort που επιτρέπει την ταξινόμησή τους. Στον ακόλουθο κώδικα δημιουργείται μια λίστα με αντικείμενα ορθογωνίων παραλληλογράμμων τα οποία ταξινομούνται με βάση το εμβαδόν τους σε αύξουσα σειρά. Για την ταξινόμηση των αντικειμένων παρουσιάζονται τρεις διαφορετικοί τρόποι που παράγουν το ίδιο αποτέλεσμα.

```
1  #include <iostream>
2  #include <list>
3
4  using namespace std;
5
6  class Rectangle {
7  public:
8     Rectangle(double w, double h) : width(w), height(h){};
9     double area() const { return width * height; } // must be const
10    void print_info() {
11       cout << "Width:" << width << " Height:" << height << " Area "
```

```
12              << this->area() << endl;
13    }
14    bool operator<(const Rectangle &other) const {
15      return this->area() < other.area();
16    }
17
18  private:
19    double width;
20    double height;
21  };
22
23  int main(int argc, char *argv[]) {
24    list<Rectangle> rectangles;
25    rectangles.push_back(Rectangle(5, 6));
26    rectangles.push_back(Rectangle(3, 3));
27    rectangles.push_back(Rectangle(5, 2));
28    rectangles.push_back(Rectangle(6, 1));
29
30    rectangles.sort();
31    for (auto r : rectangles)
32      r.print_info();
33  }
```

Κώδικας 2.9: Ταξινόμηση λίστας με αντικείμενα - α' τρόπος (sort2.cpp)

Θα πρέπει να σημειωθεί ότι η δεσμευμένη λέξη this στον κώδικα μιας κλάσης αναφέρεται σε έναν δείκτη προς το ίδιο το αντικείμενο για το οποίο καλούνται οι συναρτήσεις μέλη.

```
1   #include <iostream>
2   #include <list>
3
4   using namespace std;
5
6   class Rectangle {
7   public:
8     Rectangle(double w, double h) : width(w), height(h){};
9     double area() const { return width * height; }
10    void print_info() {
11      cout << "Width:" << width << " Height:" << height << " Area "
12           << this->area() << endl;
13    }
14
15  private:
16    double width;
17    double height;
18  };
19
20  bool operator<(const Rectangle &r1, const Rectangle &r2) {
21    return r1.area() < r2.area();
22  }
23
24  int main(int argc, char *argv[]) {
25    list<Rectangle> rectangles = {{5,6},{3,3},{5,2},{6,1}};
26
27    rectangles.sort();
28    for (auto r : rectangles)
29      r.print_info();
30  }
```

Κώδικας 2.10: Ταξινόμηση λίστας με αντικείμενα - β' τρόπος (sort3.cpp)

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

using namespace std;

class Rectangle {
public:
  Rectangle(double w, double h) : width(w), height(h){};
  double area() const { return width * height; }
  void print_info() {
    cout << "Width:" << width << " Height:" << height << " Area "
         << this->area() << endl;
  }

private:
  double width;
  double height;
};

int main(int argc, char *argv[]) {
  list<Rectangle> rectangles = {{5,6},{3,3},{5,2},{6,1}};

  struct CompareRectangle {
    bool operator()(Rectangle lhs, Rectangle rhs) {
      return lhs.area() < rhs.area();
    }
  };
  rectangles.sort(CompareRectangle());

  for (auto r : rectangles)
    r.print_info();
}
```

Κώδικας 2.11: Ταξινόμηση λίστας με αντικείμενα - γ' τρόπος (sort4.cpp)

```
Width:6 Height:1 Area 6
Width:3 Height:3 Area 9
Width:5 Height:2 Area 10
Width:5 Height:6 Area 30
```

Αν αντί για αντικείμενα το container περιέχει εγγραφές τύπου struct Rectangle τότε ένας τρόπος με το οποίο μπορεί να επιτευχθεί η ταξινόμηση των εγγραφών ορθογωνίων σε αύξουσα σειρά εμβαδού είναι ο ακόλουθος.

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

using namespace std;

struct Rectangle {
  double width;
  double height;
  bool operator<(const Rectangle &other) const {
    return width * height < other.width * other.height;
  }
};

int main(int argc, char *argv[]) {
  list<Rectangle> rectangles = {{5, 6}, {3, 3}, {5, 2}, {6, 1}};

  rectangles.sort();
  for (auto r : rectangles)
    cout << "Width:" << r.width << " Height:" << r.height
```

```
20              << " Area: " << r.width * r.height << endl;
21 }
```

Κώδικας 2.12: Ταξινόμηση λίστας με εγγραφές (sort5.cpp)

```
1 Width:6 Height:1 Area: 6
2 Width:3 Height:3 Area: 9
3 Width:5 Height:2 Area: 10
4 Width:5 Height:6 Area: 30
```

Αντίστοιχα, για να γίνει αναζήτηση ενός στοιχείου σε έναν ταξινομημένο πίνακα μπορούν να χρησιμοποιηθούν συναρτήσεις της STL όπως η συνάρτηση binary_search, η συνάρτηση lower_bound και η συνάρτηση upper_bound. Η binary_search επιστρέφει true αν το στοιχείο υπάρχει στον πίνακα αλλιώς επιστρέφει false. Οι lower_bound και upper_bound εντοπίζουν την χαμηλότερη και την υψηλότερη θέση στην οποία μπορεί να εισαχθεί το στοιχείο χωρίς να διαταραχθεί η ταξινομημένη σειρά των υπόλοιπων στοιχείων. Ένα παράδειγμα χρήσης των συναρτήσεων αυτών δίνεται στον ακόλουθο κώδικα.

```
1  #include <algorithm>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char **argv) {
7    int a[] = {45, 11, 16, 11, 7, 18, 21, 16, 11, 15};
8    int N = sizeof(a) / sizeof(int);
9    cout << "The size of the array is " << N << endl;
10   int key;
11   sort(a, a + N);
12   for (int i = 0; i < N; i++)
13     cout << a[i] << " ";
14   cout << endl;
15   cout << "Enter a value to be searched for: ";
16   cin >> key;
17   if (binary_search(a, a + N, key))
18     cout << "Found using binary_search" << endl;
19   else
20     cout << "Not found (binary_search)" << endl;
21
22   auto it1 = lower_bound(a, a + N, key);
23   auto it2 = upper_bound(a, a + N, key);
24   if (*it1 == key) {
25     cout << "Found at positions " << it1 - a << " up to " << it2 - a - 1
26          << endl;
27   } else
28     cout << "Not found (lower_bound and upper_bound)" << endl;
29 }
```

Κώδικας 2.13: Αναζήτηση σε ταξινομημένο πίνακα (search1.cpp)

```
1 The size of the array is 10
2 7 11 11 11 15 16 16 18 21 45
3 Enter a value to be searched for: 11
4 Found using binary_search
5 Found at positions 1 up to 3
```

```
1 #include <algorithm>
2 #include <iostream>
3 #include <vector>
4
5 using namespace std;
6
```

```
int main(int argc, char **argv) {
  vector<int> a = {45, 11, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << "The size of the vector is " << a.size() << endl;
  int key;
  sort(a.begin(), a.end());
  for (int i = 0; i < a.size(); i++)
    cout << a[i] << " ";
  cout << endl;
  cout << "Enter a value to be searched for: ";
  cin >> key;
  if (binary_search(a.begin(), a.end(), key))
    cout << "Found using binary_search" << endl;
  else
    cout << "Not found (binary_search)" << endl;

  auto it1 = lower_bound(a.begin(), a.end(), key);
  auto it2 = upper_bound(a.begin(), a.end(), key);
  if (*it1 == key) {
    cout << "Found at positions " << it1 - a.begin() << " up to "
         << it2 - a.begin() - 1 << endl;
  } else
    cout << "Not found (lower_bound and upper_bound)" << endl;
}
```

Κώδικας 2.14: Αναζήτηση σε ταξινομημένο διάνυσμα (search2.cpp)

```
The size of the vector is 10
7 11 11 11 15 16 16 18 21 45
Enter a value to be searched for: 16
Found using binary_search
Found at positions 5 up to 6
```

2.4 Lambdas

Η δυνατότητα lambdas έχει ενσωματωθεί στη C++ από την έκδοση 11 και μετά και επιτρέπει τη συγγραφή ανώνυμων συναρτήσεων στο σημείο που χρειάζονται, διευκολύνοντας με αυτό τον τρόπο τη συγγραφή προγραμμάτων. Ο όρος lambda ιστορικά έχει προέλθει από τη συναρτησιακή γλώσσα προγραμματισμού LISP. Μια lambda έκφραση στη C++ έχει την ακόλουθη μορφή:

```
[capture list] (parameter list) -> return type
{
function body
}
```

Συνήθως το τμήμα -> return type παραλείπεται καθώς ο μεταγλωττιστής είναι σε θέση να εκτιμήσει ο ίδιος τον τύπο επιστροφής της συνάρτησης. Στον επόμενο κώδικα παρουσιάζεται μια απλή συνάρτηση lambda η οποία δέχεται δύο double παραμέτρους και επιστρέφει το γινόμενό τους.

```
cout << "Area = " << [](double x, double y){return x * y;}(3.0, 4.5) << endl;
```

Μια lambda συνάρτηση μπορεί να αποθηκευτεί σε μια μεταβλητή και στη συνέχεια να κληθεί μέσω της μεταβλητής αυτής όπως στο ακόλουθο παράδειγμα:

```
auto area = [](double x, double y)
{
    return x * y;
};
cout << "Area = " << area(3.0, 4.5) << endl;
```

Στη συνέχεια παρουσιάζονται διάφορα παραδείγματα lambda συναρτήσεων καθώς και παραδείγματα στα οποία χρησιμοποιούνται lambda συναρτήσεις σε συνδυασμό με τις συναρτήσεις της STL: find_if, count_if, sort (σε list και σε vector) και for_each.

```
#include <algorithm>
#include <iostream>
#include <list>
#include <vector>

using namespace std;

int main() {

  cout << "Example1: call lambda function" << endl;
  cout << "Area = " << [](double x, double y) { return x * y; }(3.0, 4.5)
       << endl;

  cout << "Example2: assign lambda function to a variable" << endl;
  auto area = [](double x, double y) { return x * y; };
  cout << "Area = " << area(3.0, 4.5) << endl;
  cout << "Area = " << area(7.0, 5.5) << endl;

  vector<int> v{5, 1, 3, 2, 8, 7, 4, 5};
  // find_if
  cout << "Example3: find the first even number in the vector" << endl;
  vector<int>::iterator iter =
      find_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
  cout << *iter << endl;

  // count_if
  cout << "Example4: count the number of even numbers in the vector" << endl;
  int c = count_if(v.begin(), v.end(), [](int x) { return x % 2 == 0; });
  cout << c << endl;

  // sort
  cout << "Example5: sort list of rectangles by area (ascending)" << endl;
  struct Rectangle {
    double width;
    double height;
  };
  list<Rectangle> rectangles_list = {{5, 6}, {3, 3}, {5, 2}, {6, 1}};
  rectangles_list.sort([](Rectangle &r1, Rectangle &r2) {
    return r1.width * r1.height < r2.width * r2.height;
  });
  for (Rectangle r : rectangles_list)
    cout << "Width:" << r.width << " Height:" << r.height
         << " Area: " << r.width * r.height << endl;

  cout << "Example6: sort vector of rectangles by area (descending)" << endl;
  vector<Rectangle> rectangles_vector = {{5, 6}, {3, 3}, {5, 2}, {6, 1}};
  sort(rectangles_vector.begin(), rectangles_vector.end(),
       [](Rectangle &r1, Rectangle &r2) {
         return r1.width * r1.height > r2.width * r2.height;
       });
  for (Rectangle r : rectangles_vector)
    cout << "Width:" << r.width << " Height:" << r.height
         << " Area: " << r.width * r.height << endl;

  // for_each
  cout << "Example7: for_each" << endl;
  for_each(v.begin(), v.end(), [](int i) { cout << i << " "; });
```

```
58    cout << endl;
59    for_each(v.begin(), v.end(), [](int i) { cout << i * i << " "; });
60    cout << endl;
61 }
```

Κώδικας 2.15: Παραδείγματα με lambdas (lambda1.cpp)

```
1  Example1: call lambda function
2  Area = 13.5
3  Example2: assign lambda function to variable
4  Area = 13.5
5  Area = 38.5
6  Example3: find the first even number in the vector
7  2
8  Example4: count the number of even numbers in the vector
9  3
10 Example5: sort list of rectangles by area (ascending)
11 Width:6 Height:1 Area: 6
12 Width:3 Height:3 Area: 9
13 Width:5 Height:2 Area: 10
14 Width:5 Height:6 Area: 30
15 Example6: sort vector of rectangles by area (descending)
16 Width:5 Height:6 Area: 30
17 Width:5 Height:2 Area: 10
18 Width:3 Height:3 Area: 9
19 Width:6 Height:1 Area: 6
20 Example7: for_each
21 5 1 3 2 8 7 4 5
22 25 1 9 4 64 49 16 25
```

Μια lambda έκφραση μπορεί να έχει πρόσβαση σε μεταβλητές που βρίσκονται στην εμβέλεια που περικλείει την ίδια τη lambda έκφραση. Ειδικότερα, η πρόσβαση (capture) στις εξωτερικές μεταβλητές μπορεί να γίνει είτε με αναφορά (capture by reference), είτε με τιμή (capture by value) είτε να γίνει μικτή πρόσβαση (mixed capture). Το δε συντακτικό που χρησιμοποιείται για να υποδηλώσει το είδος της πρόσβασης είναι:

- []: καμία πρόσβαση σε εξωτερικές της lambda συνάρτησης μεταβλητές
- [&]: πρόσβαση σε όλες τις εξωτερικές μεταβλητές με αναφορά
- [=]: πρόσβαση σε όλες τις εξωτερικές μεταβλητές με τιμή
- [a, &b]: πρόσβαση στην a με τιμή και πρόσβαση στη b με αναφορά

```
1  #include <iostream>
2  #include <vector>
3
4  using namespace std;
5
6  int main(int argc, char *argv[]) {
7     vector<int> v1{1, 2, 3, 4, 5, 6};
8     vector<int> v2(6, 1);
9
10    // capture by value
11    auto lambda1 = [=](int x) {
12       cout << "v1: ";
13       for (auto p = v1.begin(); p != v1.end(); p++)
14          if (*p != x)
15             cout << *p << " ";
16       cout << endl;
17    };
18
19    // capture by reference
20    auto lambda2 = [&](int x) {
21       for (auto p = v2.begin(); p != v2.end(); p++)
22          (*p) += x;
```

```
23  };
24
25  // mixed capture
26  auto lambda3 = [v1, &v2](int x) {
27      cout << "v1: ";
28      for (auto p = v1.begin(); p != v1.end(); p++)
29          if (*p != x)
30              cout << *p << " ";
31      cout << endl;
32      for (auto p = v2.begin(); p != v2.end(); p++)
33          (*p) += x;
34  };
35
36  cout << "Example1: capture by value (all external variables)" << endl;
37  lambda1(1);
38  cout << "Example2: capture by reference (all external variables)" << endl;
39  lambda2(2);
40  cout << "Example3: mixed capture (v1 by value, v2 by reference)" << endl;
41  lambda3(1);
42
43  cout << "v1: ";
44  for (int x : v1)
45      cout << x << " ";
46  cout << endl;
47
48  cout << "v2: ";
49  for (int x : v2)
50      cout << x << " ";
51  cout << endl;
52 }
```

Κώδικας 2.16: Παραδείγματα με πρόσβαση σε εξωτερικές μεταβλητές σε lambdas (lambda2.cpp)

```
1 Example1: capture by value (all external variables)
2 v1: 2 3 4 5 6
3 Example2: capture by reference (all external variables)
4 Example3: mixed capture (v1 by value, v2 by reference)
5 v1: 2 3 4 5 6
6 v1: 1 2 3 4 5 6
7 v2: 4 4 4 4 4 4
```

2.5 TDD (Test Driven Development)

Τα τελευταία χρόνια η οδηγούμενη από ελέγχους ανάπτυξη (Test Driven Development) έχει αναγνωριστεί ως μια αποδοτική τεχνική ανάπτυξης εφαρμογών. Αν και το θέμα είναι αρκετά σύνθετο, η βασική ιδέα είναι ότι ο προγραμματιστής πρώτα γράφει κώδικα που ελέγχει αν η ζητούμενη λειτουργικότητα ικανοποιείται και στη συνέχεια προσθέτει τον κώδικα που θα υλοποιεί αυτή τη λειτουργικότητα [5]. Ανά πάσα στιγμή υπάρχει ένα σύνολο από συσσωρευμένους ελέγχους οι οποίοι για κάθε αλλαγή που γίνεται στον κώδικα είναι σε θέση να εκτελεστούν άμεσα και να δώσουν εμπιστοσύνη μέχρι ένα βαθμό στο ότι το υπό κατασκευή ή υπό τροποποίηση λογισμικό λειτουργεί ορθά. Γλώσσες όπως η Java και η Python διαθέτουν εύχρηστες βιβλιοθήκες που υποστηρίζουν την ανάπτυξη TDD (junit και pytest αντίστοιχα). Στην περίπτωση της C++ επίσης υπάρχουν διάφορες βιβλιοθήκες που μπορούν να υποστηρίξουν την ανάπτυξη TDD. Στα πλαίσια του εργαστηρίου θα χρησιμοποιηθεί η βιβλιοθήκη Catch για το σκοπό της επίδειξης του TDD.

Στο παράδειγμα που ακολουθεί παρουσιάζεται η υλοποίηση της συνάρτησης παραγοντικό. Το παραγοντικό συμβολίζεται με το θαυμαστικό (!), ορίζεται μόνο για τους μη αρνητικούς ακεραίους αριθμούς και είναι το γινόμενο όλων των θετικών ακεραίων που είναι μικρότεροι ή ίσοι του αριθμού για τον οποίο ζητείται το πα-

ραγοντικό. Το δε παραγοντικό του μηδενός είναι εξ ορισμού η μονάδα. Η πρώτη υλοποίηση είναι λανθασμένη καθώς δεν υπολογίζει σωστά το παραγοντικό του μηδενός αποδίδοντάς του την τιμή μηδέν.

```
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do this
                          // in one cpp file
#include "catch.hpp"

unsigned int Factorial(unsigned int number) {
  return number <= 1 ? number : Factorial(number - 1) * number;
}

TEST_CASE("Factorials are computed", "[factorial]") {
  REQUIRE(Factorial(0) == 1);
  REQUIRE(Factorial(1) == 1);
  REQUIRE(Factorial(2) == 2);
  REQUIRE(Factorial(3) == 6);
  REQUIRE(Factorial(10) == 3628800);
}
```

Κώδικας 2.17: Πρώτη έκδοση της συνάρτησης παραγοντικού και έλεγχοι (tdd1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται ως εξής:

```
g++ tdd1.cpp -o tdd1
./tdd1
```

Τα δε αποτελέσματα της εκτέλεσης είναι τα ακόλουθα:

```
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
tdd1 is a Catch v1.10.0 host application.
Run with -? for options

-------------------------------------------------------------------------------
Factorials are computed
-------------------------------------------------------------------------------
tdd1.cpp:9
...............................................................................

tdd1.cpp:10: FAILED:
  REQUIRE( Factorial(0) == 1 )
with expansion:
  0 == 1

===============================================================================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

Η δεύτερη υλοποίηση είναι σωστή. Τα μηνύματα που εμφανίζονται σε κάθε περίπτωση υποδεικνύουν το σημείο στο οποίο βρίσκεται το πρόβλημα και ότι πλέον αυτό λύθηκε.

```
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do this
                          // in one cpp file
#include "catch.hpp"

unsigned int Factorial(unsigned int number) {
  return number > 1 ? Factorial(number - 1) * number : 1;
}

TEST_CASE("Factorials are computed", "[factorial]") {
  REQUIRE(Factorial(0) == 1);
  REQUIRE(Factorial(1) == 1);
  REQUIRE(Factorial(2) == 2);
  REQUIRE(Factorial(3) == 6);
```

```
14   REQUIRE(Factorial(10) == 3628800);
15 }
```

Κώδικας 2.18: Δεύτερη έκδοση της συνάρτησης παραγοντικού και έλεγχοι (tdd2.cpp)

```
1 ===============================================================================
2 All tests passed (5 assertions in 1 test case)
```

2.6 Παραδείγματα

2.6.1 Παράδειγμα 1

Να γράψετε πρόγραμμα που να δημιουργεί πίνακα A με 1.000 τυχαίες ακέραιες τιμές στο διάστημα [1, 10.000] και πίνακα B με 100.000 τυχαίες ακέραιες τιμές στο ίδιο διάστημα τιμών. Η παραγωγή των τυχαίων τιμών να γίνει με τη γεννήτρια τυχαίων αριθμών mt19937 και με seed την τιμή 1821. Χρησιμοποιώντας τη συνάρτηση binary_search της STL να βρεθεί πόσες από τις τιμές του B υπάρχουν στον πίνακα A.

```
1  #include <algorithm>
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[]) {
7    mt19937 mt(1821);
8    uniform_int_distribution<int> dist(1, 10000);
9    constexpr int N = 1000;
10   constexpr int M = 100000;
11   int a[N];
12   int b[M];
13   for (int i = 0; i < N; i++)
14     a[i] = dist(mt);
15   for (int i = 0; i < M; i++)
16     b[i] = dist(mt);
17   sort(a, a + N);
18   int c = 0;
19   for (int i = 0; i < M; i++)
20     if (binary_search(a, a + N, b[i]))
21       c++;
22   cout << "Result " << c << endl;
23   return 0;
24 }
```

Κώδικας 2.19: Λύση παραδείγματος 1 (lab02_ex1.cpp)

```
1 Result 9644
```

2.6.2 Παράδειγμα 2

Η συνάρτηση accumulate() της STL επιτρέπει τον υπολογισμό αθροισμάτων στα στοιχεία ενός container. Δημιουργήστε ένα vector με διάφορες ακέραιες τιμές της επιλογής σας και υπολογίστε το άθροισμα των τιμών με τη χρήση της συνάρτησης accumulate. Επαναλάβετε τη διαδικασία για ένα container τύπου array.

```
1 #include <array>
2 #include <iostream>
3 #include <numeric>
4 #include <vector>
5
```

```cpp
using namespace std;

int main(int argc, char *argv[]) {
   vector<int> v{5, 15, 20, 17, 11, 9};
   int sum = accumulate(v.begin(), v.end(), 0);
   cout << "Sum over vector using accumulate: " << sum << endl;

   array<int, 6> a{5, 15, 20, 17, 11, 9};
   sum = accumulate(a.begin(), a.end(), 0);
   cout << "Sum over array using accumulate: " << sum << endl;
}
```

Κώδικας 2.20: Λύση παραδείγματος 2 (lab02_ex2.cpp)

```
Sum over vector using accumulate: 77
Sum over array using accumulate: 77
```

2.6.3 Παράδειγμα 3

Δημιουργήστε ένα vector που να περιέχει ονόματα. Χρησιμοποιώντας τη συνάρτηση next_permutation() εμφανίστε όλες τις διαφορετικές διατάξεις των ονομάτων που περιέχει το vector.

```cpp
#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

int main(int argc, char *argv[]) {
   vector<string> v{"petros", "anna", "nikos"};
   sort(v.begin(), v.end());
   do {
      for (string x : v)
         cout << x << " ";
      cout << endl;
   } while (next_permutation(v.begin(), v.end()));
}
```

Κώδικας 2.21: Λύση παραδείγματος 3 (lab02_ex3.cpp)

```
anna nikos petros
anna petros nikos
nikos anna petros
nikos petros anna
petros anna nikos
petros nikos anna
```

2.6.4 Παράδειγμα 4

Κατασκευάστε μια συνάρτηση που να επιστρέφει την απόσταση Hamming ανάμεσα σε δύο σειρές χαρακτήρων (η απόσταση Hamming είναι το πλήθος των χαρακτήρων που είναι διαφορετικοί στις ίδιες θέσεις ανάμεσα στις δύο σειρές). Δημιουργήστε ένα διάνυσμα με 100 τυχαίες σειρές μήκους 20 χαρακτήρων η κάθε μια χρησιμοποιώντας μόνο τους χαρακτήρες G,A,T,C. Εμφανίστε το πλήθος από τις σειρές για τις οποίες υπάρχει τουλάχιστον μια άλλη σειρά χαρακτήρων με απόσταση Hamming μικρότερη ή ίση του 10.

```cpp
#include <algorithm>
#include <iostream>
#include <string>
#include <vector>
```

```
5
6  using namespace std;
7
8  int hamming(string x, string y) {
9    int c = 0;
10   int length = x.size() > y.size() ? x.size() : y.size();
11   for (int i = 0; i < length; i++)
12     if (x.at(i) != y.at(i))
13       c++;
14   return c;
15 }
16
17 int main(int argc, char *argv[]) {
18   constexpr int N = 100;
19   constexpr int L = 20;
20   mt19937 mt(1821);
21   uniform_int_distribution<int> dist(0, 3);
22   char gact[] = {'A', 'G', 'C', 'T'};
23   vector<string> sequences(N);
24   for (int i = 0; i < N; i++)
25     for (int j = 0; j < L; j++)
26       sequences[i] += gact[dist(mt)];
27
28   int c = 0;
29   for (int i = 0; i < N; i++) {
30     cout << "Checking sequence: " << sequences[i] << "..." << endl;
31     for (int j = 0; j < N; j++) {
32       if (i == j)
33         continue;
34       int hd = hamming(sequences[i], sequences[j]);
35       cout << sequences[i] << " " << sequences[j]
36            << " ==> hamming distance=" << hd << endl;
37       if (hd <= 10) {
38         c++;
39         break;
40       }
41     }
42     cout << endl;
43   }
44   cout << "Result=" << c << endl;
45 }
```

Κώδικας 2.22: Λύση παραδείγματος 4 (lab02_ex4.cpp)

```
1  Checking sequence: CGCCATCTAAGGACTCCCCA
2  CGCCATCTAAGGACTCCCCA CACATTCAAACTGTGGGCCA ==> hamming distance=11
3  ...
4  CGCCATCTAAGGACTCCCCA CCCCATCTCGGCCCACGCTG ==> hamming distance=9
5
6  Checking sequence: CACATTCAAACTGTGGGCCA
7  CACATTCAAACTGTGGGCCA CGCCATCTAAGGACTCCCCA ==> hamming distance=11
8  ...
9  CACATTCAAACTGTGGGCCA CATATAACAACAGCATGCGA ==> hamming distance=9
10
11 ...
12
13 Checking sequence: TAGGTGCCATAAGAATCACT
14 TAGGTGCCATAAGAATCACT CGCCATCTAAGGACTCCCCA ==> hamming distance=16
15 ...
16 TAGGTGCCATAAGAATCACT AGCTGATTACGACAGCCTTC ==> hamming distance=16
17
18 Result=71
```

2.7 Ασκήσεις

1. Γράψτε ένα πρόγραμμα που να δέχεται τιμές από το χρήστη και για κάθε τιμή που θα δίνει ο χρήστης να εμφανίζει όλες τις τιμές που έχουν εισαχθεί μέχρι εκείνο το σημείο ταξινομημένες σε φθίνουσα σειρά.

2. Γράψτε ένα πρόγραμμα που να γεμίζει ένα διάνυσμα 1.000 θέσεων με τυχαίες πραγματικές τιμές στο διάστημα -100 έως και 100 διασφαλίζοντας ότι γειτονικές τιμές απέχουν το πολύ 10% η μια από την άλλη. Στη συνέχεια υπολογίστε την επτάδα συνεχόμενων τιμών με το μεγαλύτερο άθροισμα σε όλο το διάνυσμα.

3. Γράψτε ένα πρόγραμμα που να δέχεται τιμές από το χρήστη. Οι θετικές τιμές να εισάγονται σε ένα διάνυσμα v ενώ για κάθε αρνητική τιμή που εισάγεται να αναζητείται η απόλυτη τιμή της στο διάνυσμα v. Καθώς εισάγονται οι τιμές να εμφανίζονται στατιστικά για το πλήθος των τιμών που περιέχει το διάνυσμα, πόσες επιτυχίες και πόσες αποτυχίες αναζήτησης υπήρξαν.

4. Γράψτε ένα πρόγραμμα που να διαβάζει όλες τις λέξεις ενός αρχείου κειμένου και να εμφανίζει πόσες φορές υπάρχει η κάθε λέξη στο κείμενο σε αύξουσα σειρά συχνότητας. Χρησιμοποιήστε ως είσοδο το κείμενο του βιβλίου 1984 του George Orwell (http://gutenberg.net.au/ebooks01/0100021.txt).

5. Υλοποιήστε μια κλάση με όνομα BankAccount (λογαριασμός τράπεζας). Για κάθε αντικείμενο της κλάσης να τηρούνται τα στοιχεία όνομα δικαιούχου και υπόλοιπο λογαριασμού. Οι δε λειτουργίες που θα υποστηρίζει να είναι κατ' ελάχιστον η κατάθεση και η ανάληψη χρηματικού ποσού. Η υλοποίηση της κλάσης να γίνει ακολουθώντας τις αρχές του TDD.

Βιβλιογραφία

[1] http://www.geeksforgeeks.org/cpp-stl-tutorial/.

[2] https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-with-the-standard-template-library-part-1/.

[3] https://www.topcoder.com/community/data-science/data-science-tutorials/power-up-c-with-the-standard-template-library-part-2/.

[4] https://www.hackerearth.com/practice/notes/standard-template-library/

[5] http://butunclebob.com/ArticleS.UncleBob.TheThreeRulesOfTdd

Εργαστήριο 3

Θεωρητική μελέτη αλγορίθμων, χρονομέτρηση κώδικα, αλγόριθμοι ταξινόμησης και αλγόριθμοι αναζήτησης

3.1 Εισαγωγή

Στο εργαστήριο αυτό παρουσιάζονται ορισμένοι αλγόριθμοι ταξινόμησης και ορισμένοι αλγόριθμοι αναζήτησης. Πρόκειται για μερικούς από τους σημαντικότερους αλγορίθμους στην επιστήμη των υπολογιστών. Σε πρακτικό επίπεδο ένα σημαντικό ποσοστό της επεξεργαστικής ισχύος των υπολογιστών δαπανάται στην ταξινόμηση δεδομένων η οποία διευκολύνει τις αναζητήσεις που ακολουθούν. Επιπλέον, γίνεται αναφορά στη θεωρητική εκτίμηση της απόδοσης ενός αλγορίθμου και στη μέτρηση χρόνου εκτέλεσης κώδικα.

3.2 Θεωρητική και εμπειρική εκτίμηση της απόδοσης αλγορίθμων

Συχνά χρειάζεται να εκτιμηθεί η καταλληλότητα ενός αλγορίθμου για την επίλυση ενός προβλήματος. Ποιοτικά χαρακτηριστικά όπως ο χρόνος εκτέλεσης και ο χώρος που απαιτεί στη μνήμη και στο δίσκο μπορεί να τον καθιστούν υποδεέστερο άλλων αλγορίθμων ή ακόμα και ακατάλληλο για επίλυση του προβλήματος. Γενικά, η απόδοση ενός αλγορίθμου μπορεί να εκτιμηθεί θεωρητικά και εμπειρικά.

3.2.1 Θεωρητική μελέτη αλγορίθμων

Η θεωρητική μελέτη προσδιορίζει την ασυμπτωτική συμπεριφορά του αλγορίθμου, δηλαδή πως θα συμπεριφέρεται ο αλγόριθμος καθώς τα δεδομένα εισόδου αυξάνονται σε μέγεθος προσεγγίζοντας μεγάλες τιμές. Μελετώντας θεωρητικά διαφορετικούς αλγορίθμους που επιτελούν το ίδιο έργο μπορεί να πραγματοποιηθεί σύγκριση μεταξύ τους ακόμα και χωρίς να γραφεί κώδικας που να τους υλοποιεί σε μια συγκεκριμένη γλώσσα προγραμματισμού. Η μέθοδος που έχει επικρατήσει για τη θεωρητική μελέτη αλγορίθμων είναι η ασυμπτωτική ανάλυση και ιδιαίτερα ο συμβολισμός του μεγάλου Ο (Big O notation). Ο συμβολισμός του μεγάλου Ο περιγράφει τη χειρότερη περίπτωση εκτέλεσης και συνήθως αφορά το χρόνο εκτέλεσης ή σπανιότερα το χώρο που απαιτείται από τον αλγόριθμο. Στη συνέχεια θα επιχειρηθεί μια πρακτική παρουσίαση του εν λόγω συμβολισμού μέσω παραδειγμάτων [1].

$O(1)$

Ο συμβολισμός $O(1)$ περιγράφει αλγορίθμους που πάντα εκτελούνται απαιτώντας τον ίδιο χρόνο (ή χώρο), άσχετα με το μέγεθος των δεδομένων με τα οποία τροφοδοτούνται. Ο ακόλουθος κώδικας που είναι ένα παρά-

δειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(1)$ επιστρέφει true αν το πρώτο στοιχείο ενός πίνακα ακεραίων είναι άρτιο, αλλιώς επιστρέφει false.

```
bool is_first_element_even(int a[]) {
    return a[0] % 2 == 0;
}
```

$O(n)$

Ο συμβολισμός $O(n)$ περιγράφει αλγορίθμους που ο χρόνος εκτέλεσής τους μεγαλώνει γραμμικά και σε ευθεία αναλογία με το μέγεθος της εισόδου. Ο κώδικας που ακολουθεί επιστρέφει true αν το στοιχείο key υπάρχει στον πίνακα a, N θέσεων. Η ασυμπτωτική του πολυπλοκότητα είναι $O(n)$.

```
bool exists(int a[], int N, int key) {
  for (int i = 0; i < N; i++)
    if (a[i] == key)
      return true;
  return false;
}
```

$O(n^2)$

Ο συμβολισμός $O(n^2)$ περιγράφει αλγορίθμους που ο χρόνος εκτέλεσής τους μεγαλώνει ανάλογα με το τετράγωνο του μεγέθους της εισόδου. Αυτό τυπικά συμβαίνει όταν ο κώδικας περιέχει δύο εντολές επανάληψης την μια μέσα στην άλλη. Ο ακόλουθος κώδικας εξετάζει αν ένας πίνακας έχει διπλότυπα και έχει ασυμπτωτική πολυπλοκότητα $O(n^2)$.

```
bool has_duplicates(int a[], int N) {
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++) {
      if (i == j)
        continue;
      if (a[i] == a[j])
        return true;
    }
  return false;
}
```

$O(2^n)$

Το $O(2^n)$ αφορά αλγορίθμους που ο χρόνος εκτέλεσής τους διπλασιάζεται για κάθε μονάδα αύξησης των δεδομένων εισόδου. Η αύξηση είναι εξαιρετικά απότομη καθιστώντας τον αλγόριθμο μη χρησιμοποιήσιμο παρά μόνο για μικρές τιμές του n. Ένα παράδειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(2^n)$ είναι ο αναδρομικός υπολογισμός των αριθμών Fibonacci. Η ακολουθία Fibonacci είναι η ακόλουθη: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... Εξ ορισμού οι δύο πρώτοι όροι είναι το μηδέν και το ένα, ενώ κάθε επόμενος όρος είναι το άθροισμα των δύο προηγούμενων του.

```
int fibo(int n) {
  if (n <= 1)
    return n;
  else
    return fibo(n − 2) + fibo(n − 1);
}
```

$O(log(n))$

Το $O(log(n))$ περιγράφει αλγορίθμους στους οποίους σε κάθε βήμα τους το μέγεθος των δεδομένων που μένει να εξετάσει ο αλγόριθμος μειώνεται στο μισό. Ένα παράδειγμα αλγορίθμου με ασυμπτωτική πολυπλοκότητα $O(log(n))$ είναι ο ακόλουθος κώδικας που επιστρέφει λογική τιμή σχετικά με το εάν υπάρχει το στοιχείο key στον ταξινομημένο πίνακα a, N θέσεων.

```
bool exists_in_sorted(int a[], int N, int key) {
  int left = 0, right = N − 1, m;
  while (left <= right) {
    m = (left + right) / 2;
    if (a[m] == key)
      return true;
    else if (a[m] > key)
      right = m − 1;
    else
      left = m + 1;
  }
  return false;
}
```

Στη συνέχεια παρατίθενται ασυμπτωτικές πολυπλοκότητες αλγορίθμων από τις ταχύτερες προς τις βραδύτερες: $O(1)$, $O(log(n))$, $O(n)$,$O(nlog(n))$, $O(n^2)$, $O(n^3)$, $O(n^4)$, $O(2^n)$, $O(n!)$. Με εξαιρέσεις, επιθυμητές πολυπλοκότητες αλγορίθμων είναι μέχρι και $O(n^2)$.

3.2.2 Εμπειρική μελέτη αλγορίθμων

Η εμπειρική εκτίμηση της απόδοσης ενός προγράμματος έχει να κάνει με τη χρονομέτρησή του για διάφορες περιπτώσεις δεδομένων εισόδου και τη σύγκρισή του με εναλλακτικές υλοποιήσεις προγραμμάτων. Στη συνέχεια θα παρουσιαστούν δύο τρόποι μέτρησης χρόνου εκτέλεσης κώδικα που μπορούν να εφαρμοστούν στη C++.

Μέτρηση χρόνου εκτέλεσης κώδικα με τη συνάρτηση clock()

Ο ακόλουθος κώδικας μετράει το χρόνο που απαιτεί ο υπολογισμός του αθροίσματος των τετραγωνικών ριζών 10.000.000 τυχαίων ακέραιων αριθμών με τιμές στο διάστημα από 0 έως 10.000. Η μέτρηση του χρόνου πραγματοποιείται με τη συνάρτηση clock() η οποία επιστρέφει τον αριθμό από clock ticks που έχουν περάσει από τη στιγμή που το πρόγραμμα ξεκίνησε την εκτέλεση του. Ο αριθμός των δευτερολέπτων που έχουν περάσει προκύπτει διαιρώντας τον αριθμό των clock ticks με τη σταθερά CLOCKS_PER_SEC. Αυτός ο τρόπος υπολογισμού του χρόνου εκτέλεσης έχει "κληρονομηθεί" στη C++ από τη C.

```
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  clock_t t1, t2;
  t1 = clock();
  srand(1821);
  double sum = 0.0;
  for (int i = 1; i <= 10000000; i++) {
    int x = rand() % 10000 + 1;
    sum += sqrt(x);
  }
```

```
17    cout << "The sum is: " << sum << endl;
18
19    t2 = clock();
20    double elapsed_time = (double)(t2 - t1) / CLOCKS_PER_SEC;
21    cout << "Elapsed time " << elapsed_time << " seconds" << endl;
22 }
```

Κώδικας 3.1: Μέτρηση χρόνου εκτέλεσης κώδικα(timing1.cpp)

```
1 The sum is: 6.39952e+008
2 Elapsed time 0.213
```

Μέτρηση χρόνου εκτέλεσης κώδικα με τη χρήση του high_resolution_clock::time_point

Η C++ έχει προσθέσει νέους τρόπους μέτρησης του χρόνου εκτέλεσης προγραμμάτων. Στον ακόλουθο κώδικα παρουσιάζεται ένα παράδειγμα με χρήση time_points.

```
1  #include <chrono>
2  #include <cmath>
3  #include <iostream>
4  #include <random>
5
6  using namespace std;
7  using namespace std::chrono;
8
9  int main() {
10   high_resolution_clock::time_point t1 = high_resolution_clock::now();
11   mt19937 mt(1821);
12   uniform_int_distribution<int> dist(0, 10000);
13   double sum = 0.0;
14   for (int i = 1; i <= 10000000; i++) {
15     int x = dist(mt);
16     sum += sqrt(x);
17   }
18   cout << "The sum is: " << sum << endl;
19   high_resolution_clock::time_point t2 = high_resolution_clock::now();
20   auto duration = duration_cast<microseconds>(t2 - t1).count();
21   cout << "Time elapsed: " << duration << " microseconds "
22        << duration / 1000000.0 << " seconds" << endl;
23 }
```

Κώδικας 3.2: Μέτρηση χρόνου εκτέλεσης κώδικα (timing2.cpp)

```
1 The sum is: 6.6666e+008
2 Time elapsed: 537030 microseconds 0.53703 seconds
```

Θα πρέπει να σημειωθεί ότι κατά τη μεταγλώττιση είναι δυνατό να δοθεί οδηγία προς το μεταγλωττιστή έτσι ώστε να προχωρήσει σε βελτιστοποιήσεις του κώδικα που παράγει που θα οδηγήσουν σε ταχύτερη εκτέλεση. Το flag που χρησιμοποιείται είναι το -O και οι πιθανές τιμές που μπορεί να λάβει είναι: -O0, -O1, -O2, -O3. Καθώς αυξάνεται ο αριθμός δεξιά του -O που χρησιμοποιείται στη μεταγλώττιση ενεργοποιούνται περισσότερες βελτιστοποιήσεις σε βάρος του χρόνου μεταγλώττισης. Στη συνέχεια παρουσιάζονται οι εντολές μεταγλώττισης και ο χρόνος εκτέλεσης για κάθε μια από τις 4 περιπτώσεις του κώδικα .

```
1 g++ timing2.cpp -o timing2a -O0 -std=c++11
2 ./timing2a
3 The sum is: 6.6666e+008
4 Time elapsed: 537030 microseconds 0.53703 seconds
5
6 g++ timing2.cpp -o timing2b -O1 -std=c++11
7 ./timing2b
```

```
The sum is: 6.6666e+008
Time elapsed: 125007 microseconds 0.125007 seconds

g++ timing2.cpp -o timing2c -O2 -std=c++11
./timing2c
The sum is: 6.6666e+008
Time elapsed: 127007 microseconds 0.127007 seconds

g++ timing2.cpp -o timing2d -O3 -std=c++11
./timing2d
The sum is: 6.6666e+008
Time elapsed: 114006 microseconds 0.114006 seconds
```

3.3 Αλγόριθμοι ταξινόμησης

3.3.1 Ταξινόμηση με εισαγωγή

Η ταξινόμηση με εισαγωγή (insertion-sort) λειτουργεί δημιουργώντας μια ταξινομημένη λίστα στο αριστερό άκρο των δεδομένων και επαναληπτικά τοποθετεί το στοιχείο το οποίο βρίσκεται δεξιά της ταξινομημένης λίστας στη σωστή θέση σε σχέση με τα ήδη ταξινομημένα στοιχεία. Ο αλγόριθμος ταξινόμησης με εισαγωγή καθώς και η κλήση του από κύριο πρόγραμμα για την αύξουσα ταξινόμηση ενός πίνακα 10 θέσεων παρουσιάζεται στον κώδικα που ακολουθεί.

```cpp
template <class T> void insertion_sort(T a[], int n) {
  for (int i = 1; i < n; i++) {
    T key = a[i];
    int j = i - 1;
    while ((j >= 0) && (key < a[j])) {
      a[j + 1] = a[j];
      j--;
    }
    a[j + 1] = key;
  }
}
```

Κώδικας 3.3: Ο αλγόριθμος ταξινόμησης με εισαγωγή (insertion_sort.cpp)

```cpp
#include "insertion_sort.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << "Sort using insertion sort" << endl;
  insertion_sort(a, 10);
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
}
```

Κώδικας 3.4: Κλήση της συνάρτησης insertion_sort (sort1.cpp)

```
Sort using insertion sort
7 11 11 15 16 16 18 21 32 45
```

3.3.2 Ταξινόμηση με συγχώνευση

Η ταξινόμηση με συγχώνευση (merge-sort) είναι αναδρομικός αλγόριθμος και στηρίζεται στη συγχώνευση ταξινομημένων υποακολουθιών έτσι ώστε να δημιουργούνται νέες ταξινομημένες υποακολουθίες. Μια υλοποί-

ηση του κώδικα ταξινόμησης με συγχώνευση παρουσιάζεται στη συνέχεια.

```
template <class T> void merge(T a[], int l, int m, int r) {
  T la[m − l + 1];
  T ra[r − m];
  for (int i = 0; i < m − l + 1; i++)
    la[i] = a[l + i];
  for (int i = 0; i < r − m; i++)
    ra[i] = a[m + 1 + i];
  int i = 0, j = 0, k = l;
  while ((i < m − l + 1) && (j < r − m)) {
    if (la[i] < ra[j]) {
      a[k] = la[i];
      i++;
    } else {
      a[k] = ra[j];
      j++;
    }
    k++;
  }
  if (i == m − l + 1) {
    while (j < r − m) {
      a[k] = ra[j];
      j++;
      k++;
    }
  } else {
    while (i < m − l + 1) {
      a[k] = la[i];
      i++;
      k++;
    }
  }
}

template <class T> void merge_sort(T a[], int l, int r) {
  if (l < r) {
    int m = (l + r) / 2;
    merge_sort(a, l, m);
    merge_sort(a, m + 1, r);
    merge(a, l, m, r);
  }
}

template <class T> void merge_sort(T a[], int N) { merge_sort(a, 0, N − 1); }
```

Κώδικας 3.5: Ο αλγόριθμος ταξινόμησης με συγχώνευση (merge_sort.cpp)

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

using namespace std;

int main(int argc, char **argv) {
  int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << ”Sort using merge sort” << endl;
  merge_sort(a, 10);
  for (int i = 0; i < 10; i++)
    cout << a[i] << ” ”;
}
```

Κώδικας 3.6: Κλήση της συνάρτησης merge_sort (sort2.cpp)

```
Sort using merge sort
7 11 11 15 16 16 18 21 32 45
```

3.3.3 Γρήγορη ταξινόμηση

Ο κώδικας της γρήγορης ταξινόμησης παρουσιάζεται στη συνέχεια. Πρόκειται για κώδικα ο οποίος καλείται αναδρομικά σε υποακολουθίες των δεδομένων και σε κάθε κλήση επιλέγει ένα στοιχείο (pivot) και διαχωρίζει τα υπόλοιπα στοιχεία έτσι ώστε αριστερά να είναι τα στοιχεία που είναι μικρότερα του pivot και δεξιά αυτά τα οποία είναι μεγαλύτερα.

```
#include <utility> // std::swap

template <class T> int partition(T a[], int l, int r) {
  int p = l;
  int i = l + 1;
  for (int j = l + 1; j <= r; j++) {
    if (a[j] < a[p]) {
      std::swap(a[j], a[i]);
      i++;
    }
  }
  std::swap(a[p], a[i − 1]);
  return i − 1;
}

template <class T> void quick_sort(T a[], int l, int r) {
  if (l >= r)
    return;
  else {
    int p = partition(a, l, r);
    quick_sort(a, l, p − 1);
    quick_sort(a, p + 1, r);
  }
}

template <class T> void quick_sort(T a[], int N) { quick_sort(a, 0, N − 1); }
```

Κώδικας 3.7: Ο αλγόριθμος γρήγορης ταξινόμησης (quick_sort.cpp)

```
#include "quick_sort.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
  int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
  cout << "Sort using quick sort" << endl;
  quick_sort(a, 10);
  for (int i = 0; i < 10; i++)
    cout << a[i] << " ";
}
```

Κώδικας 3.8: Κλήση της συνάρτησης quick_sort (sort3.cpp)

```
Sort using quick sort
7 11 11 15 16 16 18 21 32 45
```

3.3.4 Ταξινόμηση κατάταξης

ο αλγόριθμος ταξινόμησης κατάταξης (rank-sort) λειτουργεί ως εξής: Για κάθε στοιχείο του δεδομένου πίνακα a που επιθυμούμε να ταξινομήσουμε υπολογίζεται μια τιμή κατάταξης (rank). Η τιμή κατάταξης ενός στοιχείου του πίνακα είναι το πλήθος των μικρότερων από αυτό στοιχείων συν το πλήθος των ίσων με αυτό στοιχείων που έχουν μικρότερο δείκτη σε σχέση με αυτό το στοιχείο (δηλαδή βρίσκονται αριστερά του). Δηλαδή ισχύει ότι η τιμή κατάταξης ενός στοιχείου x του πίνακα είναι ίση με το άθροισμα 2 όρων: του πλήθους των μικρότερων στοιχείων του x από όλο τον πίνακα και του πλήθους των ίσων με το x στοιχείων που έχουν μικρότερο δείκτη σε σχέση με το x. Για παράδειγμα στην ακολουθία τιμών a=[44, 21, 78, 16, 56, 21] θα πρέπει να δημιουργηθεί ένας νέος πίνακας r = [3, 1, 5, 0, 4, 2]. Έχοντας υπολογίσει τον πίνακα r θα πρέπει τα στοιχεία του a να αντιγραφούν σε ένα νέο βοηθητικό πίνακα temp έτσι ώστε κάθε τιμή που υπάρχει στον πίνακα r να λειτουργεί ως δείκτης για το που πρέπει να τοποθετηθεί το αντίστοιχο στοιχείο του a στον πίνακα temp. Τέλος θα πρέπει να αντιγραφεί ο πίνακας temp στον πίνακα a. Στη συνέχεια παρουσιάζεται ο κώδικας του αλγορίθμου rank-sort. Παρουσιάζονται δύο υλοποιήσεις. Η πρώτη υλοποίηση (rank_sort) αφορά τον αλγόριθμο όπως έχει περιγραφεί παραπάνω ενώ η δεύτερη (rank_sort_in_place) είναι από το βιβλίο “Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++ του Sartaj Sahnii” [2] και δεν απαιτεί τη χρήση του βοηθητικού πίνακα temp, συνεπώς είναι αποδοτικότερος.

```
#include <iostream>
#include <utility> // swap

using namespace std;

template <class T> void rank_sort(T a[], int n) {
   int r[n] = {0};
   for (int i = 0; i < n; i++)
      for (int j = 0; j < n; j++)
         if (a[j] < a[i] || (a[j] == a[i] && j < i))
            r[i]++;
   int temp[n];
   for (int i = 0; i < n; i++)
      temp[r[i]] = a[i];
   for (int i = 0; i < n; i++)
      a[i] = temp[i];
}

template <class T> void rank_sort_in_place(T a[], int n) {
   int r[n] = {0};
   for (int i = 0; i < n; i++)
      for (int j = 0; j < i; j++)
         if (a[j] <= a[i])
            r[i]++;
         else
            r[j]++;
   for (int i = 0; i < n; i++)
      while (r[i] != i) {
         int t = r[i];
         swap(a[i], a[t]);
         swap(r[i], r[t]);
      }
}

int main(int argc, char **argv) {
   int a[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
   cout << "Sort using rank sort" << endl;
   rank_sort(a, 10);
   for (int i = 0; i < 10; i++)
      cout << a[i] << " ";
```

```
41    cout << endl << "Sort using rank sort (in place)" << endl;
42    int b[] = {45, 32, 16, 11, 7, 18, 21, 16, 11, 15};
43    rank_sort_in_place(b, 10);
44    for (int i = 0; i < 10; i++)
45       cout << b[i] << " ";
46 }
```

Κώδικας 3.9: Ο αλγόριθμος ταξινόμησης κατάταξης (sort4.cpp)

```
1  Sort using rank sort
2  7 11 11 15 16 16 18 21 32 45
3  Sort using rank sort (in place)
4  7 11 11 15 16 16 18 21 32 45
```

3.3.5 Σταθερή ταξινόμηση (stable sorting)

Ένας αλγόριθμος ταξινόμησης είναι stable αν τα στοιχεία με την ίδια τιμή εμφανίζονται με την ίδια σειρά με την οποία βρισκόταν στην αρχική λίστα και στην ταξινομημένη λίστα [3]. Για παράδειγμα, αν σε μια λίστα εγγραφών φοιτητών/φοιτητριών (όνομα - βαθμός) πραγματοποιηθεί σταθερή ταξινόμηση με βάση το βαθμό, θα πρέπει οι φοιτητές με τον ίδιο βαθμό να μην αλλάξουν σειρά μεταξύ τους σε σχέση με τη σειρά που είχαν στην αρχική λίστα. Ο ακόλουθος κώδικας διαβάζει τα στοιχεία υποθετικών ατόμων από το αρχείο students20.txt και εφαρμόζοντας αλγορίθμους ταξινόμησης που παρουσιάστηκαν νωρίτερα καθώς και αλγορίθμους ταξινόμησης της STL παρουσιάζει τα δεδομένα ταξινομημένα.

```
1  #include "insertion_sort.cpp"
2  #include "merge_sort.cpp"
3  #include "quick_sort.cpp"
4  #include <algorithm>
5  #include <fstream>
6  #include <iostream>
7  #include <sstream>
8  #include <vector>
9
10 using namespace std;
11
12 struct student {
13    string name;
14    int grade;
15    bool operator<(student other) const { return grade > other.grade; }
16 };
17
18 void print_students(student a[], int N) {
19    for (int i = 0; i < N; i++)
20       cout << a[i].name << "-" << a[i].grade << " ";
21    cout << endl;
22 }
23
24 int main(void) {
25    vector<student> v;
26    fstream filestr;
27    string buffer;
28
29    filestr.open("students20.txt");
30    if (!filestr.is_open()) {
31       cerr << "File not found" << endl;
32       exit(-1);
33    }
34    while (getline(filestr, buffer)) {
35       stringstream ss(buffer);
```

```
36      student st;
37      ss >> st.name;
38      ss >> st.grade;
39      v.push_back(st);
40    }
41    filestr.close();
42
43    int N = v.size();
44    cout << "ORIGINAL LIST:" << endl;
45    for (int i = 0; i < N; i++)
46      cout << v[i].name << "-" << v[i].grade << " ";
47    cout << endl;
48
49    cout << "INSERTION SORT: (STABLE)" << endl;
50    student *a = new student[N];
51    for (int i = 0; i < N; i++)
52      a[i] = v[i];
53    insertion_sort(a, N);
54    print_students(a, N);
55    delete[] a;
56
57    cout << "MERGE SORT: (NON STABLE)" << endl;
58    a = new student[N];
59    for (int i = 0; i < N; i++)
60      a[i] = v[i];
61    merge_sort(a, N);
62    print_students(a, N);
63    delete[] a;
64
65    cout << "QUICK SORT: (NON STABLE)" << endl;
66    a = new student[N];
67    for (int i = 0; i < N; i++)
68      a[i] = v[i];
69    quick_sort(a, N);
70    print_students(a, N);
71    delete[] a;
72
73    cout << "STL SORT: (NON STABLE)" << endl;
74    a = new student[N];
75    for (int i = 0; i < N; i++)
76      a[i] = v[i];
77    sort(a, a + N);
78    print_students(a, N);
79    delete[] a;
80
81    cout << "STL STABLESORT: (STABLE)" << endl;
82    a = new student[N];
83    for (int i = 0; i < N; i++)
84      a[i] = v[i];
85    stable_sort(a, a + N);
86    print_students(a, N);
87    delete[] a;
88  }
```

Κώδικας 3.10: Σταθερή ταξινόμηση (stable.cpp)

```
1  ORIGINAL LIST:
2  apostolis-5 nikos-5 petros-7 maria-2 kostas-5 giannis-5 sofia-1 dimitra-10 kiki-5 aristea-9 christos-7 niki-4 katerina-9 giorgos-8
       lefteris-4 efthymios-10 iordanis-9 alexis-5 anna-5 georgia-4
3  INSERTION SORT:
4  dimitra-10 efthymios-10 aristea-9 katerina-9 iordanis-9 giorgos-8 petros-7 christos-7 apostolis-5 nikos-5 kostas-5 giannis-5 kiki-5
       alexis-5 anna-5 niki-4 lefteris-4 georgia-4 maria-2 sofia-1
```

```
MERGE SORT:
efthymios-10 dimitra-10 iordanis-9 katerina-9 aristea-9 giorgos-8 christos-7 petros-7 anna-5 alexis-5 kiki-5 giannis-5 kostas-5 nikos-5 apostolis-5 georgia-4 lefteris-4 niki-4 maria-2 sofia-1
QUICK SORT:
efthymios-10 dimitra-10 iordanis-9 aristea-9 katerina-9 giorgos-8 christos-7 petros-7 apostolis-5 anna-5 kostas-5 giannis-5 nikos-5 kiki-5 alexis-5 georgia-4 niki-4 lefteris-4 maria-2 sofia-1
STL SORT:
efthymios-10 dimitra-10 iordanis-9 katerina-9 aristea-9 giorgos-8 petros-7 christos-7 nikos-5 anna-5 alexis-5 kiki-5 giannis-5 kostas-5 apostolis-5 niki-4 lefteris-4 georgia-4 maria-2 sofia-1
STL STABLESORT:
dimitra-10 efthymios-10 aristea-9 katerina-9 iordanis-9 giorgos-8 petros-7 christos-7 apostolis-5 nikos-5 kostas-5 giannis-5 kiki-5 alexis-5 anna-5 niki-4 lefteris-4 georgia-4 maria-2 sofia-1
```

3.4 Αλγόριθμοι αναζήτησης

3.4.1 Σειριακή αναζήτηση

Η σειριακή αναζήτηση είναι ο απλούστερος αλγόριθμος αναζήτησης. Εξετάζει τα στοιχεία ένα προς ένα στη σειρά μέχρι να βρει το στοιχείο που αναζητείται. Το πλεονέκτημα του αλγορίθμου είναι ότι μπορεί να εφαρμοστεί σε μη ταξινομημένους πίνακες.

```cpp
#include <iostream>

using namespace std;

template <class T> int sequential_search(T a[], int n, T key) {
  for (int i = 0; i < n; i++)
    if (a[i] == key)
      return i;
  return -1;
}

int main(int argc, char **argv) {
  int a[] = {5, 11, 45, 23, 10, 17, 32, 8, 9, 4};
  int key;
  cout << "Search for: ";
  cin >> key;
  int pos = sequential_search(a, 10, key);
  if (pos == -1)
    cout << "Not found" << endl;
  else
    cout << "Found at position " << pos << endl;
}
```

Κώδικας 3.11: Ο αλγόριθμος σειριακής αναζήτησης (search1.cpp)

```
Search for: 45
Found at position 2
```

3.4.2 Δυαδική αναζήτηση

Η δυαδική αναζήτηση μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα. Διαιρεί επαναληπτικά την ακολουθία σε 2 υποακολουθίες και απορρίπτει την ακολουθία στην οποία συμπεραίνει ότι δεν μπορεί να βρεθεί το στοιχείο.

```cpp
// non-recursive implementation
template <class T> int binary_search(T a[], int l, int r, T key) {
  while (l <= r) {
    int m = (l + r) / 2;
```

```
    if (a[m] == key)
      return m;
    else if (a[m] < key)
      l = m + 1;
    else
      r = m - 1;
  }
  return -1;
}

template <class T> int binary_search(T a[], int n, T key) {
  return binary_search(a, 0, n - 1, key);
}

// recursive implementation
template <class T> int binary_search_r(T a[], int l, int r, T key) {
  int m = (l + r) / 2;
  if (l > r)
    return -1;
  else if (a[m] == key)
    return m;
  else if (key < a[m])
    return binary_search_r(a, l, m - 1, key);
  else
    return binary_search_r(a, m + 1, r, key);
}

template <class T> int binary_search_r(T a[], int n, T key) {
  return binary_search_r(a, 0, n - 1, key);
}
```

Κώδικας 3.12: Ο αλγόριθμος δυαδικής αναζήτησης σε μη αναδρομική και σε αναδρομική έκδοση (binary_search.cpp)

```
#include "binary_search.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
    int key, a[] = {11, 45, 53, 60, 67, 72, 88, 91, 94, 98}, N = sizeof(a) / sizeof(int);
    for (int i = 0; i < N; i++)
        cout << "a[" << i << "]=" << a[i] << " ";
    cout << endl;
    cout << "Search for: ";
    cin >> key;
    cout << "Using non recursive algorithm (binary search)" << endl;
    int pos = binary_search(a, N, key);
    if (pos == -1)
        cout << "Not found" << endl;
    else
        cout << "Found at position " << pos << endl;

    cout << "Using recursive algorithm (binary search)" << endl;
    pos = binary_search_r(a, N, key);
    if (pos == -1)
        cout << "Not found" << endl;
    else
        cout << "Found at position " << pos << endl;
}
```

Κώδικας 3.13: Κλήση της συνάρτησης binary_search (search2.cpp)

```
a[0]=11 a[1]=45 a[2]=53 a[3]=60 a[4]=67 a[5]=72 a[6]=88 a[7]=91 a[8]=94 a[9]=98
Search for: 88
Using non recursive algorithm (binary search)
Found at position 6
Using recursive algorithm (binary search)
Found at position 6
```

3.4.3 Αναζήτηση με παρεμβολή

Η αναζήτηση με παρεμβολή (interpolation-search) είναι μια παραλλαγή της δυαδικής αναζήτησης και μπορεί να εφαρμοστεί μόνο σε ταξινομημένα δεδομένα. Αντί να χρησιμοποιηθεί η τιμή 50% για να διαχωριστούν τα δεδομένα σε 2 ισομεγέθεις λίστες (όπως συμβαίνει στη δυαδική αναζήτηση) υπολογίζεται μια τιμή η οποία εκτιμάται ότι θα οδηγήσει πλησιέστερα στο στοιχείο που αναζητείται. Αν l είναι ο δείκτης του αριστερότερου στοιχείου της ακολουθίας και r ο δείκτης του δεξιότερου στοιχείου της ακολουθίας τότε υπολογίζεται ο συντελεστής c = (key−a[l])/(a[r]−a[l]) όπου key είναι το στοιχείο προς αναζήτηση και a είναι η ακολουθία τιμών στην οποία αναζητείται το key. Η ακολουθία των δεδομένων διαχωρίζεται με βάση τον συντελεστή c σε δύο υποακολουθίες. Η διαδικασία επαναλαμβάνεται ανάλογα με τη δυαδική αναζήτηση. Στη συνέχεια παρουσιάζεται ο κώδικας της αναζήτησης με παρεμβολή.

```
template <class T> int interpolation_search(T a[], int l, int r, T key) {
  int m;
  if (l > r)
    return −1;
  else if (l == r)
    m = l;
  else {
    double c = (double)(key − a[l]) / (double)(a[r] − a[l]);
    if ((c < 0) || (c > 1))
      return −1;
    m = (int)(l + (r − l) ∗ c);
  }
  if (a[m] == key)
    return m;
  else if (key < a[m])
    return interpolation_search(a, l, m − 1, key);
  else
    return interpolation_search(a, m + 1, r, key);
}

template <class T> int interpolation_search(T a[], int n, T key) {
  return interpolation_search(a, 0, n − 1, key);
}
```

Κώδικας 3.14: Ο αλγόριθμος αναζήτησης με παρεμβολή (interpolation_search.cpp)

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

using namespace std;

int main(int argc, char ∗∗argv) {
  int key, a[] = {11, 45, 53, 60, 67, 72, 88, 91, 94, 98}, N = sizeof(a) / sizeof(int);
  for (int i = 0; i < N; i++)
      cout << ”a[” << i << ”]=” << a[i] << ” ”;
   cout << endl;
```

```
11    cout << "Search for: ";
12    cin >> key;
13    int pos = interpolation_search(a, 10, key);
14    if (pos == -1)
15       cout << "Not found" << endl;
16    else
17       cout << "Found at position " << pos << endl;
18 }
```

Κώδικας 3.15: Κλήση της συνάρτησης interpolation_search search3.cpp

```
1 a[0]=11 a[1]=45 a[2]=53 a[3]=60 a[4]=67 a[5]=72 a[6]=88 a[7]=91 a[8]=94 a[9]=98
2 Search for: 91
3 Found at position 7
```

3.5 Παραδείγματα

3.5.1 Παράδειγμα 1

Γράψτε πρόγραμμα που να συγκρίνει τους χρόνους εκτέλεσης των αλγορίθμων ταξινόμησης insertion-sort, merge-sort, quick-sort καθώς και του αλγορίθμου ταξινόμησης της βιβλιοθήκης STL (συνάρτηση sort). Η σύγκριση να αφορά τυχαία δεδομένα τύπου float με τιμές στο διάστημα από -1.000 έως 1.000. Τα μεγέθη των πινάκων που θα ταξινομηθούν να είναι 5.000, 10.000, 20.0000, 40.000, 80.000, 160.0000 και 320.000 αριθμών.

```
1  #include "insertion_sort.cpp"
2  #include "merge_sort.cpp"
3  #include "quick_sort.cpp"
4  #include <algorithm>
5  #include <chrono>
6  #include <iomanip>
7  #include <iostream>
8  #include <random>
9
10 using namespace std;
11 using namespace std::chrono;
12
13 void benchmark_sort_algorithm(string alg) {
14    mt19937 mt(1821);
15    uniform_real_distribution<float> dist(-1000, 1000);
16
17    int sizes[] = {5000, 10000, 20000, 40000, 80000, 160000, 320000};
18    for (int i = 0; i < 7; i++) {
19       int N = sizes[i];
20       float *a = new float[N];
21       for (int i = 0; i < N; i++)
22          a[i] = dist(mt);
23
24       auto t1 = high_resolution_clock::now();
25       if (alg.compare("merge-sort") == 0)
26          merge_sort(a, N);
27       else if (alg.compare("quick-sort") == 0)
28          quick_sort(a, N);
29       else if (alg.compare("STL-sort") == 0)
30          sort(a, a + N);
31       else if (alg.compare("insertion-sort") == 0)
32          insertion_sort(a, N);
33       auto t2 = high_resolution_clock::now();
34
```

```
35      auto elapsed_time = duration_cast<milliseconds>(t2 − t1).count();
36      cout << fixed << setprecision(3);
37      cout << "Elapsed time " << alg << "\t" << sizes[i] << "\t" << elapsed_time
38           << " milliseconds" << endl;
39      delete[] a;
40    }
41  }
42
43  int main(int argc, char **argv) {
44    benchmark_sort_algorithm("insertion−sort");
45    cout << "####################################" << endl;
46    benchmark_sort_algorithm("merge−sort");
47    cout << "####################################" << endl;
48    benchmark_sort_algorithm("quick−sort");
49    cout << "####################################" << endl;
50    benchmark_sort_algorithm("STL−sort");
51    cout << "####################################" << endl;
52  }
```

Κώδικας 3.16: Σύγκριση χρόνου εκτέλεσης αλγορίθμων ταξινόμησης (lab03_ex1.cpp)

```
1  Elapsed time insertion−sort 5000 46 milliseconds
2  Elapsed time insertion−sort 10000 167 milliseconds
3  Elapsed time insertion−sort 20000 658 milliseconds
4  Elapsed time insertion−sort 40000 2595 milliseconds
5  Elapsed time insertion−sort 80000 10377 milliseconds
6  Elapsed time insertion−sort 160000 41441 milliseconds
7  Elapsed time insertion−sort 320000 167593 milliseconds
8  ####################################
9  Elapsed time merge−sort 5000 1 milliseconds
10 Elapsed time merge−sort 10000 3 milliseconds
11 Elapsed time merge−sort 20000 7 milliseconds
12 Elapsed time merge−sort 40000 14 milliseconds
13 Elapsed time merge−sort 80000 32 milliseconds
14 Elapsed time merge−sort 160000 60 milliseconds
15 Elapsed time merge−sort 320000 125 milliseconds
16 ####################################
17 Elapsed time quick−sort 5000 1 milliseconds
18 Elapsed time quick−sort 10000 3 milliseconds
19 Elapsed time quick−sort 20000 5 milliseconds
20 Elapsed time quick−sort 40000 10 milliseconds
21 Elapsed time quick−sort 80000 24 milliseconds
22 Elapsed time quick−sort 160000 47 milliseconds
23 Elapsed time quick−sort 320000 105 milliseconds
24 ####################################
25 Elapsed time STL−sort 5000 1 milliseconds
26 Elapsed time STL−sort 10000 2 milliseconds
27 Elapsed time STL−sort 20000 4 milliseconds
28 Elapsed time STL−sort 40000 8 milliseconds
29 Elapsed time STL−sort 80000 17 milliseconds
30 Elapsed time STL−sort 160000 37 milliseconds
31 Elapsed time STL−sort 320000 79 milliseconds
32 ####################################
```

3.5.2 Παράδειγμα 2

Γράψτε πρόγραμμα που να συγκρίνει τους χρόνους εκτέλεσης των αλγορίθμων αναζήτησης binary-search, interpolation-search και του αλγορίθμου αναζήτησης της βιβλιοθήκης STL binary_search για ταξινομημένα ακέραια δεδομένα με τιμές στο διάστημα από 0 έως 10.000.000. Η σύγκριση να εξετάζει τα ακόλουθα μεγέθη πινάκων 5.000, 10.000, 20.000, 40.000, 80.000, 160.000 και 320.000 αριθμών. Οι χρόνοι εκτέλεσης να αφορούν τους συνολικούς χρόνους που απαιτούνται έτσι ώστε να αναζητηθούν 100.000 τυχαίες τιμές με καθένα από τους αλγορίθμους.

```cpp
#include "binary_search.cpp"
#include "interpolation_search.cpp"
#include <algorithm>
#include <ctime>
#include <iomanip>
#include <iostream>
#include <random>
#include <chrono>

using namespace std;
using namespace std::chrono;

void benchmark_search_algorithm(string alg) {
  clock_t t1, t2;
  mt19937 mt(1729);
  uniform_int_distribution<int> dist(0, 1000000);
  int M = 100000;
  int keys[M];
  for (int i = 0; i < M; i++) {
    keys[i] = dist(mt);
  }
  int sizes[] = {5000, 10000, 20000, 40000, 80000, 160000, 320000};
  for (int i = 0; i < 7; i++) {
    int N = sizes[i];
    int *a = new int[N];
    for (int i = 0; i < N; i++)
      a[i] = dist(mt);
    sort(a, a + N);

    auto t1 = high_resolution_clock::now();
    int c = 0;
    if (alg.compare("binary-search") == 0) {
      for (int j = 0; j < M; j++)
        if (binary_search(a, 0, N - 1, keys[j]) != -1)
          c++;
    } else if (alg.compare("interpolation-search") == 0) {
      for (int j = 0; j < M; j++)
        if (interpolation_search(a, 0, N - 1, keys[j]) != -1)
          c++;
    } else if (alg.compare("STL-binary-search") == 0)
      for (int j = 0; j < M; j++)
        if (binary_search(a, a + N, keys[j]))
          c++;
    auto t2 = high_resolution_clock::now();

    auto elapsed_time = duration_cast<milliseconds>(t2 - t1).count();
    cout << fixed << setprecision(3);
    cout << "Elapsed time " << alg << "\t" << sizes[i] << "\t" << elapsed_time
         << " milliseconds" << endl;
    delete[] a;
  }
}

int main(int argc, char **argv) {
  benchmark_search_algorithm("binary-search");
  cout << "#####################################" << endl;
  benchmark_search_algorithm("interpolation-search");
  cout << "#####################################" << endl;
  benchmark_search_algorithm("STL-binary-search");
  cout << "#####################################" << endl;
```

```
61 }
```

Κώδικας 3.17: Σύγκριση χρόνου εκτέλεσης αλγορίθμων ταξινόμησης (lab03_ex2.cpp)

```
1  Elapsed time binary−search 5000 20 milliseconds
2  Elapsed time binary−search 10000 20 milliseconds
3  Elapsed time binary−search 20000 22 milliseconds
4  Elapsed time binary−search 40000 24 milliseconds
5  Elapsed time binary−search 80000 31 milliseconds
6  Elapsed time binary−search 160000 30 milliseconds
7  Elapsed time binary−search 320000 35 milliseconds
8  ####################################
9  Elapsed time interpolation−search 5000 13 milliseconds
10 Elapsed time interpolation−search 10000 14 milliseconds
11 Elapsed time interpolation−search 20000 14 milliseconds
12 Elapsed time interpolation−search 40000 15 milliseconds
13 Elapsed time interpolation−search 80000 15 milliseconds
14 Elapsed time interpolation−search 160000 17 milliseconds
15 Elapsed time interpolation−search 320000 19 milliseconds
16 ####################################
17 Elapsed time STL−binary−search 5000 30 milliseconds
18 Elapsed time STL−binary−search 10000 33 milliseconds
19 Elapsed time STL−binary−search 20000 35 milliseconds
20 Elapsed time STL−binary−search 40000 39 milliseconds
21 Elapsed time STL−binary−search 80000 41 milliseconds
22 Elapsed time STL−binary−search 160000 43 milliseconds
23 Elapsed time STL−binary−search 320000 51 milliseconds
24 ####################################
```

3.6 Ασκήσεις

1. Ο αλγόριθμος bogosort αναδιατάσσει τυχαία τις τιμές ενός πίνακα μέχρι να προκύψει μια ταξινομημένη διάταξη. Γράψτε ένα πρόγραμμα που να υλοποιεί τον αλγόριθμο bogosort για την ταξινόμηση ενός πίνακα ακεραίων τιμών. Χρησιμοποιήστε τη συνάρτηση shuffle.

2. Να υλοποιηθεί ο αλγόριθμος ταξινόμησης με επιλογή (selection sort) και να εφαρμοστεί για τη ταξινόμηση ενός πίνακα πραγματικών τιμών, ενός πίνακα ακεραίων και ενός πίνακα με λεκτικά (δηλαδή να γίνουν τρεις κλήσεις του αλγορίθμου). Ο αλγόριθμος ταξινόμησης με επιλογή ξεκινά εντοπίζοντας το μικρότερο στοιχείο και το τοποθετεί στη πρώτη θέση. Συνεχίζει, ακολουθώντας την ίδια διαδικασία χρησιμοποιώντας το τμήμα του πίνακα που δεν έχει ταξινομηθεί ακόμα.

3. Γράψτε μια αναδρομική έκδοση του κώδικα για την ταξινόμηση με επιλογή (selection sort).

4. Υλοποιήστε τον αλγόριθμο radix sort (https://en.wikipedia.org/wiki/Radix_sort) σε C++ και χρησιμοποιήστε τον για την ταξινόμηση ενός μεγάλου πίνακα ακεραίων.

Βιβλιογραφία

[1] A beginners guide to Big O notation, https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation

[2] Sartaj Sahnii. Δομές Δεδομένων, Αλγόριθμοι και Εφαρμογές στη C++. Εκδόσεις Τζιόλα, 2004.

[3] Stable Sorting, https://hackernoon.com/stable-sorting-677453884792

Εργαστήριο 4

Γραμμικές λίστες, λίστες της STL

4.1 Εισαγωγή

Οι γραμμικές λίστες είναι δομές δεδομένων που επιτρέπουν την αποθήκευση και την προσπέλαση στοιχείων έτσι ώστε τα στοιχεία να βρίσκονται σε μια σειρά με σαφώς ορισμένη την έννοια της θέσης καθώς και το ποιο στοιχείο προηγείται και ποιο έπεται καθενός. Σε χαμηλού επιπέδου γλώσσες προγραμματισμού όπως η C η υλοποίηση γραμμικών λιστών είναι ευθύνη του προγραμματιστή. Από την άλλη μεριά, γλώσσες υψηλού επιπέδου όπως η C++, η Java, η Python κ.α. προσφέρουν έτοιμες υλοποιήσεις γραμμικών λιστών. Ωστόσο, η γνώση υλοποίησης των συγκεκριμένων δομών (όπως και άλλων) αποτελεί βασική ικανότητα η οποία αποκτά ιδιαίτερη χρησιμότητα όταν ζητούνται εξειδικευμένες υλοποιήσεις. Στο συγκεκριμένο εργαστήριο θα παρουσιαστούν δύο πιθανές υλοποιήσεις γραμμικών λιστών (στατικής λίστας και απλά συνδεδεμένης λίστας) καθώς και οι ενσωματωμένες δυνατότητες της C++ μέσω containers της STL όπως το vector, το list και άλλα. Ο κώδικας όλων των παραδειγμάτων βρίσκεται στο https://github.com/chgogos/ceteiep_dsa.

4.2 Γραμμικές λίστες

Υπάρχουν δύο βασικοί τρόποι αναπαράστασης γραμμικών λιστών, η στατική αναπαράσταση η οποία γίνεται με τη χρήση πινάκων και η αναπαράσταση με συνδεδεμένη λίστα η οποία γίνεται με τη χρήση δεικτών.

4.2.1 Στατικές γραμμικές λίστες

Στη στατική γραμμική λίστα τα δεδομένα αποθηκεύονται σε ένα πίνακα. Κάθε στοιχείο της στατικής λίστας μπορεί να προσπελαστεί με βάση τη θέση του στον ίδιο σταθερό χρόνο με όλα τα άλλα στοιχεία άσχετα με τη θέση στην οποία βρίσκεται (τυχαία προσπέλαση). Ο κώδικας υλοποίησης μιας στατικής λίστας με μέγιστη χωρητικότητα 50.000 στοιχείων παρουσιάζεται στη συνέχεια.

```
1  #include <iostream>
2  #include <stdexcept>
3
4  using namespace std;
5
6  const int MAX = 50000;
7  template <class T> struct static_list {
8    T elements[MAX];
9    int size = 0;
10 };
11
12 // get item at position i
13 template <class T> T access(static_list<T> &static_list, int i) {
14   if (i < 0 || i >= static_list.size)
```

```
15      throw out_of_range("the index is out of range");
16    else
17      return static_list.elements[i];
18  }
19
20  // get the position of item x
21  template <class T> int search(static_list<T> &static_list, T x) {
22    for (int i = 0; i < static_list.size; i++)
23      if (static_list.elements[i] == x)
24        return i;
25    return -1;
26  }
27
28  // append item x at the end of the list
29  template <class T> void push_back(static_list<T> &static_list, T x) {
30    if (static_list.size == MAX)
31      throw "full list exception";
32    static_list.elements[static_list.size] = x;
33    static_list.size++;
34  }
35
36  // append item x at position i, shift the rest to the right
37  template <class T> void insert(static_list<T> &static_list, int i, T x) {
38    if (static_list.size == MAX)
39      throw "full list exception";
40    if (i < 0 || i >= static_list.size)
41      throw out_of_range("the index is out of range");
42    for (int k = static_list.size; k > i; k--)
43      static_list.elements[k] = static_list.elements[k - 1];
44    static_list.elements[i] = x;
45    static_list.size++;
46  }
47
48  // delete item at position i, shift the rest to the left
49  template <class T> void delete_item(static_list<T> &static_list, int i) {
50    if (i < 0 || i >= static_list.size)
51      throw out_of_range("the index is out of range");
52    for (int k = i; k < static_list.size; k++)
53      static_list.elements[k] = static_list.elements[k + 1];
54    static_list.size--;
55  }
56
57  template <class T> void print_list(static_list<T> &static_list) {
58    cout << "List: ";
59    for (int i = 0; i < static_list.size; i++)
60      cout << static_list.elements[i] << " ";
61    cout << endl;
62  }
```

Κώδικας 4.1: Υλοποίηση στατικής γραμμικής λίστας (static_list.cpp)

```
1  #include "static_list.cpp"
2  #include <iostream>
3
4  using namespace std;
5
6  int main(void) {
7    static_list<int> alist;
8    cout << "#1. Add items 10, 20 and 30" << endl;
9    push_back(alist, 10);
10   push_back(alist, 20);
```

```
11   push_back(alist, 30);
12   print_list(alist);
13   cout << "#2. Insert at position 1 item 15" << endl;
14   insert(alist, 1, 15);
15   print_list(alist);
16   cout << "#3. Delete item at position 0" << endl;
17   delete_item(alist, 0);
18   print_list(alist);
19   cout << "#4. Item at position 2: " << access(alist, 2) << endl;
20   try {
21     cout << "#5. Item at position -1" << access(alist, -1) << endl;
22   } catch (out_of_range oor) {
23     cerr << "Exception: " << oor.what() << endl;
24   }
25   cout << "#6. Search for item 20: " << search(alist, 20) << endl;
26   cout << "#7. Search for item 21: " << search(alist, 21) << endl;
27   cout << "#8. Append item 99 until full list exception occurs" << endl;
28   try {
29     while (true)
30       push_back(alist, 99);
31   } catch (const char *msg) {
32     cerr << "Exception: " << msg << endl;
33   }
34 }
```

Κώδικας 4.2: Παράδειγμα με στατική γραμμική λίστα (list1.cpp)

```
1  #1. Add items 10, 20 and 30
2  List: 10 20 30
3  #2. Insert at position 1 item 15
4  List: 10 15 20 30
5  #3. Delete item at position 0
6  List: 15 20 30
7  #4. Item at position 2: 30
8  Exception: the index is out of range
9  #6. Search for item 20: 1
10 #7. Search for item 21: -1
11 #8. Append item 99 until full list exception occurs
12 Exception: full list exception
```

Εξαιρέσεις στη C++ Στους κώδικες που προηγήθηκαν καθώς και σε επόμενους γίνεται χρήση εξαιρέσεων (exceptions) για να σηματοδοτηθούν γεγονότα τα οποία αφορούν έκτακτες καταστάσεις που το πρόγραμμα θα πρέπει να διαχειρίζεται. Για παράδειγμα, όταν επιχειρηθεί η προσπέλαση ενός στοιχείου σε μια θέση εκτός των ορίων της λίστας (π.χ. ενέργεια 5 στον κώδικα 4.2) τότε γίνεται throw ένα exception out_of_range το οποίο θα πρέπει να συλληφθεί (να γίνει catch) από τον κώδικα που καλεί τη συνάρτηση που προκάλεσε το throw exception. Περισσότερες πληροφορίες για τα exceptions και τον χειρισμό τους μπορούν να αναζητηθούν στην αναφορά [1].

Σχετικά με τις στατικές γραμμικές λίστες ισχύει ότι έχουν τα ακόλουθα πλεονεκτήματα:

- Εύκολη υλοποίηση.
- Σταθερός χρόνος, $O(1)$, εντοπισμού στοιχείου με βάση τη θέση του.
- Γραμμικός χρόνος, $O(n)$, για αναζήτηση ενός στοιχείου ή λογαριθμικός χρόνος, $O(log(n))$, αν τα στοιχεία είναι ταξινομημένα.

Ωστόσο, οι στατικές γραμμικές λίστες έχουν και μειονεκτήματα τα οποία παρατίθενται στη συνέχεια:

- Δέσμευση μεγάλου τμήματος μνήμης ακόμη και όταν η λίστα είναι άδεια ή περιέχει λίγα στοιχεία.
- Επιβολή άνω ορίου στα δεδομένα τα οποία μπορεί να δεχθεί (ο περιορισμός αυτός μπορεί να ξεπεραστεί με συνθετότερη υλοποίηση που αυξομειώνει το μέγεθος του πίνακα υποδοχής όταν αυτό απαιτείται).

- Γραμμικός χρόνος $O(n)$ για εισαγωγή και διαγραφή στοιχείων του πίνακα.

4.2.2 Συνδεδεμένες γραμμικές λίστες

Η συνδεδεμένη γραμμική λίστα αποτελείται από μηδέν ή περισσότερους κόμβους. Κάθε κόμβος περιέχει δεδομένα και έναν ή περισσότερους δείκτες σε άλλους κόμβους της συνδεδεμένης λίστας. Συχνά χρησιμοποιείται ένας πρόσθετος κόμβος με όνομα head (κόμβος κεφαλής) που δείχνει στο πρώτο στοιχείο της λίστας και μπορεί να περιέχει επιπλέον πληροφορίες όπως το μήκος της. Στη συνέχεια παρουσιάζεται ο κώδικας που υλοποιεί μια απλά συνδεδεμένη λίστα.

```
#include <iostream>
#include <stdexcept>

using namespace std;

template <class T> struct node {
  T data;
  struct node<T> *next = NULL;
};

template <class T> struct linked_list {
  struct node<T> *head = NULL;
  int size = 0;
};

// get node item at position i
template <class T>
struct node<T> *access_node(linked_list<T> &linked_list, int i) {
  if (i < 0 || i >= linked_list.size)
    throw out_of_range("the index is out of range");
  struct node<T> *current = linked_list.head;
  for (int k = 0; k < i; k++)
    current = current->next;
  return current;
}

// get node item at position i
template <class T>
T access(linked_list<T> &linked_list, int i) {
  struct node<T> *item = access_node(linked_list, i);
  return item->data;
}

// get the position of item x
template <class T> int search(linked_list<T> &linked_list, T x) {
  struct node<T> *current = linked_list.head;
  int i = 0;
  while (current != NULL) {
    if (current->data == x)
      return i;
    i++;
    current = current->next;
  }
  return -1;
}

// append item x at the end of the list
template <class T> void push_back(linked_list<T> &l, T x) {
  struct node<T> *new_node, *current;
  new_node = new node<T>();
```

```
51    new_node->data = x;
52    new_node->next = NULL;
53    current = l.head;
54    if (current == NULL) {
55       l.head = new_node;
56       l.size++;
57    } else {
58       while (current->next != NULL)
59          current = current->next;
60       current->next = new_node;
61       l.size++;
62    }
63 }
64
65 // append item x after position i
66 template <class T> void insert_after(linked_list<T> &linked_list, int i, T x) {
67    if (i < 0 || i >= linked_list.size)
68       throw out_of_range("the index is out of range");
69    struct node<T> *ptr = access_node(linked_list, i);
70    struct node<T> *new_node = new node<T>();
71    new_node->data = x;
72    new_node->next = ptr->next;
73    ptr->next = new_node;
74    linked_list.size++;
75 }
76
77 // append item at the head
78 template <class T> void insert_head(linked_list<T> &linked_list, T x) {
79    struct node<T> *new_node = new node<T>();
80    new_node->data = x;
81    new_node->next = linked_list.head;
82    linked_list.head = new_node;
83    linked_list.size++;
84 }
85
86 // append item x at position i
87 template <class T> void insert(linked_list<T> &linked_list, int i, T x) {
88    if (i == 0)
89       insert_head(linked_list, x);
90    else
91       insert_after(linked_list, i - 1, x);
92 }
93
94 // delete item at position i
95 template <class T> void delete_item(linked_list<T> &l, int i) {
96    if (i < 0 || i >= l.size)
97       throw out_of_range("the index is out of range");
98    if (i == 0) {
99       struct node<T> *ptr = l.head;
100      l.head = ptr->next;
101      delete ptr;
102   } else {
103      struct node<T> *ptr = access_node(l, i - 1);
104      struct node<T> *to_be_deleted = ptr->next;
105      ptr->next = to_be_deleted->next;
106      delete to_be_deleted;
107   }
108   l.size--;
109 }
110
111 template <class T> void print_list(linked_list<T> &l) {
```

```
112   cout << "List: ";
113   struct node<T> *current = l.head;
114   while (current != NULL) {
115     cout << current->data << " ";
116     current = current->next;
117   }
118   cout << endl;
119 }
```

Κώδικας 4.3: Υλοποίηση συνδεδεμένης γραμμικής λίστας (linked_list.cpp)

```
1  #include "linked_list.cpp"
2  #include <iostream>
3
4  using namespace std;
5
6  int main(int argc, char *argv[]) {
7    linked_list<int> alist;
8    cout << "#1. Add items 10, 20 and 30" << endl;
9    push_back(alist, 10);
10   push_back(alist, 20);
11   push_back(alist, 30);
12   print_list(alist);
13   cout << "#2. Insert at position 1 item 15" << endl;
14   insert(alist, 1, 15);
15   print_list(alist);
16   cout << "#3. Delete item at position 0" << endl;
17   delete_item(alist, 0);
18   print_list(alist);
19   cout << "#4. Item at position 2: " << access(alist, 2) << endl;
20   try {
21     cout << "#5. Item at position -1" << access(alist, -1) << endl;
22   } catch (out_of_range oor) {
23     cerr << "Exception: " << oor.what() << endl;
24   }
25   cout << "#6. Search for item 20: " << search(alist, 20) << endl;
26   cout << "#7. Search for item 21: " << search(alist, 21) << endl;
27   cout << "#8. Delete allocated memory " << endl;
28   for (int i = 0; i < alist.size; i++)
29     delete_item(alist, i);
30 }
```

Κώδικας 4.4: Παράδειγμα με συνδεδεμένη γραμμική λίστα (list2.cpp)

```
1  #1. Add items 10, 20 and 30
2  List: 10 20 30
3  #2. Insert at position 1 item 15
4  List: 10 15 20 30
5  #3. Delete item at position 0
6  List: 15 20 30
7  #4. Item at position 2: 30
8  Exception: the index is out of range
9  #6. Search for item 20: 1
10 #7. Search for item 21: -1
11 #8. Delete allocated memory
```

Οι συνδεδεμένες γραμμικές λίστες έχουν τα ακόλουθα πλεονεκτήματα:

- Καλή χρήση του αποθηκευτικού χώρου (αν και απαιτείται περισσότερος χώρος για την αποθήκευση κάθε κόμβου λόγω των δεικτών).
- Σταθερός χρόνος, $O(1)$, για την εισαγωγή και διαγραφή στοιχείων.

Από την άλλη μεριά τα μειονεκτήματα των συνδεδεμένων λιστών είναι τα ακόλουθα:

- Συνθετότερη υλοποίηση.
- Δεν επιτρέπουν την απευθείας μετάβαση σε κάποιο στοιχείο με βάση τη θέση του.

Οι αναφορές [2] και [3] παρέχουν χρήσιμες πληροφορίες και ασκήσεις σχετικά με τις συνδεδεμένες λίστες και το ρόλο των δεικτών στην υλοποίησή τους.

4.2.3 Γραμμικές λίστες της STL

Τα containers της STL που μπορούν να λειτουργήσουν ως διατεταγμένες συλλογές (ordered collections) είναι τα ακόλουθα: vector, deque, arrays, list, forward_list και bitset.

Vectors

Τα vectors αλλάζουν αυτόματα μέγεθος καθώς προστίθενται ή αφαιρούνται στοιχεία σε αυτά. Τα δεδομένα τους τοποθετούνται σε συνεχόμενες θέσεις μνήμης. Περισσότερες πληροφορίες για τα vectors μπορούν να βρεθούν στις αναφορές [4] και [5]. Στο ακόλουθο παράδειγμα παρουσιάζονται 4 διαφορετικοί τρόποι με τους οποίους μπορεί να προσπελαστεί το πρώτο και το τελευταίο στοιχείο του διανύσματος καθώς και η δυνατότητα ελέγχου με τον τελεστή της ισότητας σχετικά με το αν δύο διανύσματα είναι ίσα.

```
#include <iostream>
#include <vector>

using namespace std;
int main() {
   vector<int> v1{10, 20, 30, 40};
   cout << "1. The first element is " << v1.front() << endl;
   cout << "2. The first element is " << v1[0] << endl;
   cout << "3. The first element is " << v1.at(0) << endl;
   cout << "4. The first element is " << *(v1.begin()) << endl;
   cout << "1. The last element is " << v1.back() << endl;
   cout << "2. The last element is " << v1[3] << endl;
   cout << "3. The last element is " << v1.at(3) << endl;
   cout << "4. The last element is " << *(v1.end() - 1) << endl;

   vector<int> v2{10, 20, 30, 40};
   if (v1 == v2)
      cout << "equal vectors" << endl;
}
```

Κώδικας 4.5: Παράδειγμα με vectors (vector.cpp)

```
1. The first element is 10
2. The first element is 10
3. The first element is 10
4. The first element is 10
1. The last element is 40
2. The last element is 40
3. The last element is 40
4. The last element is 40
equal vectors
```

Deques

Τα deques (double ended queues = ουρές με δύο άκρα) είναι παρόμοια με τα vectors αλλά μπορούν να προστεθούν ή να διαγραφούν στοιχεία τόσο από την αρχή όσο και από το τέλος τους. Περισσότερες πληροφορίες για τα deques μπορούν να βρεθούν στην αναφορά [6]. Στο παράδειγμα που ακολουθεί εισάγονται σε ένα deque εναλλάξ στο αριστερό και στο δεξί άκρο οι άρτιοι και οι περιττοί ακέραιοι αριθμοί στο διάστημα [1,20].

```cpp
#include <deque>
#include <iostream>

using namespace std;

int main() {
   deque<int> de;
   for (int i = 1; i <= 20; i++)
      if (i % 2 == 0)
         de.push_front(i);
      else
         de.push_back(i);

   for (int x : de)
      cout << x << " ";
   cout << endl;
}
```

Κώδικας 4.6: Παράδειγμα με deque (deque.cpp)

```
20 18 16 14 12 10 8 6 4 2 1 3 5 7 9 11 13 15 17 19
```

Arrays

Τα arrays εισήχθησαν στη C++11 με στόχο να αντικαταστήσουν τους απλούς πίνακες της C. Κατά τη δήλωση ενός array προσδιορίζεται και το μέγεθός του. Περισσότερες πληροφορίες για τα arrays μπορούν να βρεθούν στην αναφορά [7]. Στο ακόλουθο παράδειγμα δημιουργείται ένα array με 5 πραγματικές τιμές, ταξινομείται και εμφανίζεται.

```cpp
#include <algorithm>
#include <array>
#include <iostream>

using namespace std;

int main() {
   array<double, 5> a{6.5, 2.1, 7.2, 8.1, 1.9};
   sort(a.begin(), a.end());
   for (double x : a)
      cout << x << " ";
   cout << endl;
}
```

Κώδικας 4.7: Παράδειγμα με array (array.cpp)

```
1.9 2.1 6.5 7.2 8.1
```

Lists

Οι lists είναι διπλά συνδεδεμένες λίστες. Δηλαδή κάθε κόμβος της λίστας διαθέτει έναν δείκτη προς το επόμενο και έναν δείκτη προς το προηγούμενο στοιχείο στη λίστα. Περισσότερες πληροφορίες για τις lists μπορούν να βρεθούν στην αναφορά [8]. Στο παράδειγμα που ακολουθεί μια διπλά συνδεδεμένη λίστα διανύεται από δεξιά προς τα αριστερά και από αριστερά προς τα δεξιά στην ίδια επανάληψη.

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

```

```
using namespace std;

int main() {
   list<int> alist{10, 20, 30, 40};
   list<int>::iterator it = alist.begin();
   list<int>::reverse_iterator rit = alist.rbegin();

   while (it != alist.end()) {
      cout << "Forwards:" << *it << endl;
      cout << "Backwards:" << *rit << endl;
      it++;
      rit++;
   }
}
```

Κώδικας 4.8: Παράδειγμα με list (forward_list.cpp)

```
Forwards:10
Backwards:40
Forwards:20
Backwards:30
Forwards:30
Backwards:20
Forwards:40
Backwards:10
```

Forward Lists

Οι forward lists (λίστες προς τα εμπρός) είναι απλά συνδεδεμένες λίστες με κάθε κόμβο να διαθέτει έναν δείκτη προς το επόμενο στοιχείο της λίστας. Περισσότερες πληροφορίες για τις forward lists μπορούν να βρεθούν στις αναφορές [9] και [10]. Ακολουθεί ένα παράδειγμα που αντιστρέφει μια απλά συνδεδεμένη λίστα στην οποία έχουν πριν προστεθεί στοιχεία.

```
#include <forward_list>
#include <iostream>

using namespace std;
int main() {
   forward_list<int> fl{10, 20, 30, 40, 50};
   for (int x : fl)
      cout << x << " ";
   cout << endl;
   fl.reverse();
   for (int x : fl)
      cout << x << " ";
   cout << endl;
}
```

Κώδικας 4.9: Παράδειγμα με forward_list (forward_list.cpp)

```
10 20 30 40 50
50 40 30 20 10
```

Bitset

Τα bitsets είναι πίνακες με λογικές τιμές τις οποίες αποθηκεύουν με αποδοτικό τρόπο καθώς για κάθε λογική τιμή απαιτείται μόνο 1 bit. Το μέγεθος ενός bitset πρέπει να είναι γνωστό κατά τη μεταγλώττιση. Μια ιδιαιτερότητά του είναι ότι οι δείκτες θέσης που χρησιμοποιούνται για την αναφορά στα στοιχεία του ξεκινούν

την αρίθμησή τους με το μηδέν από δεξιά και αυξάνονται προς τα αριστερά. Για παράδειγμα ένα bitset με τιμές 101011 έχει την τιμή 1 στις θέσεις 0,1,3,5 και 0 στις θέσεις 2 και 4. Περισσότερες πληροφορίες για τα bitsets μπορούν να βρεθούν στις αναφορές [11] και [12]. Ακολουθεί ένα παράδειγμα που εμφανίζει χρησιμοποιώντας 5 δυαδικά ψηφία τους ακέραιους αριθμούς από το 0 μέχρι το 7.

```
#include <bitset>
#include <iostream>

using namespace std;

int main() {
   for (int x = 0; x < 8; x++) {
      bitset<5> b(x);
      cout << x << " ==> " << b << " bits set " << b.count() << endl;
   }
}
```

Κώδικας 4.10: Παράδειγμα με bitset (bitset.cpp)

```
0 ==> 00000 bits set 0
1 ==> 00001 bits set 1
2 ==> 00010 bits set 1
3 ==> 00011 bits set 2
4 ==> 00100 bits set 1
5 ==> 00101 bits set 2
6 ==> 00110 bits set 2
7 ==> 00111 bits set 3
```

4.3 Παραδείγματα

4.3.1 Παράδειγμα 1

Γράψτε ένα πρόγραμμα που να ελέγχεται από το ακόλουθο μενού και να πραγματοποιεί τις λειτουργίες που περιγράφονται σε μια απλά συνδεδεμένη λίστα με ακεραίους.

1. Εμφάνιση στοιχείων λίστας. (Show list)
2. Εισαγωγή στοιχείου στο πίσω άκρο της λίστας. (Insert item (back))
3. Εισαγωγή στοιχείου σε συγκεκριμένη θέση. (Insert item (at position))
4. Διαγραφή στοιχείου σε συγκεκριμένη θέση. (Delete item (from position))
5. Διαγραφή όλων των στοιχείων που έχουν την τιμή. (Delete all items having value)
6. Έξοδος. (Exit)

```
#include "linked_list.cpp"
#include <iostream>

using namespace std;

int main(int argc, char **argv) {
   linked_list<int> alist;
   int choice, position, value;
   do {
      cout << "1.Show list"
           << "\n";
      cout << "2.Insert item (back)"
           << "\n";
      cout << "3.Insert item (at position)"
           << "\n";
      cout << "4.Delete item (from position)"
           << "\n";
```

```
18      cout << "5.Delete all items having value"
19           << "-";
20      cout << "6.Exit" << endl;
21      cout << "Enter choice:";
22      cin >> choice;
23      if (choice < 1 || choice > 6) {
24        cerr << "Choice should be 1 to 6" << endl;
25        continue;
26      }
27      try {
28        switch (choice) {
29        case 1:
30          print_list(alist);
31          break;
32        case 2:
33          cout << "Enter value:";
34          cin >> value;
35          push_back(alist, value);
36          break;
37        case 3:
38          cout << "Enter position and value:";
39          cin >> position >> value;
40          insert(alist, position, value);
41          break;
42        case 4:
43          cout << "Enter position:";
44          cin >> position;
45          delete_item(alist, position);
46          break;
47        case 5:
48          cout << "Enter value:";
49          cin >> value;
50          int i = 0;
51          while (i < alist.size)
52            if (access(alist, i) == value)
53              delete_item(alist, i);
54            else
55              i++;
56        }
57      } catch (out_of_range oor) {
58        cerr << "Out of range, try again" << endl;
59      }
60    } while (choice != 6);
61  }
```

Κώδικας 4.11: Έλεγχος συνδεδεμένης λίστας ακεραίων μέσω μενού (lab04_ex1.cpp)

```
1  1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
2  Enter choice:2
3  Enter value:10
4  1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
5  Enter choice:2
6  Enter value:20
7  1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
8  Enter choice:2
9  Enter value:20
10 1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
11 Enter choice:3
12 Enter position and value:1 15
13 1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
14 Enter choice:1
15 List: 10 15 20 20
16 1.Show list-2.Insert item (back)-3.Insert item (at position)-4.Delete item (from position)-5.Delete all items having value-6.Exit
```

```
Enter choice:4
Enter position:0
1.Show list−2.Insert item (back)−3.Insert item (at position)−4.Delete item (from position)−5.Delete all items having value−6.Exit
Enter choice:1
List: 15 20 20
1.Show list−2.Insert item (back)−3.Insert item (at position)−4.Delete item (from position)−5.Delete all items having value−6.Exit
Enter choice:5
Enter value:20
1.Show list−2.Insert item (back)−3.Insert item (at position)−4.Delete item (from position)−5.Delete all items having value−6.Exit
Enter choice:1
List: 15
1.Show list−2.Insert item (back)−3.Insert item (at position)−4.Delete item (from position)−5.Delete all items having value−6.Exit
Enter choice:6
```

4.3.2 Παράδειγμα 2

Έστω μια τράπεζα που διατηρεί για κάθε πελάτη της τον κωδικό του και το υπόλοιπο του λογαριασμού του. Για τις ανάγκες του παραδείγματος θα δημιουργηθούν τυχαίοι πελάτες ως εξής: ο κωδικός κάθε πελάτη θα αποτελείται από 10 σύμβολα που θα επιλέγονται με τυχαίο τρόπο από τα γράμματα της αγγλικής αλφαβήτου και το υπόλοιπο κάθε πελάτη θα είναι ένας τυχαίος ακέραιος αριθμός από το 0 μέχρι το 5.000. Το πρόγραμμα θα πραγματοποιεί τις ακόλουθες λειτουργίες:

Α΄ Θα δημιουργεί μια λίστα με 40.000 τυχαίους πελάτες.
Β΄ Θα υπολογίζει το άθροισμα των υπολοίπων από όλους τους πελάτες που ο κωδικός τους ξεκινά με το χαρακτήρα Α.
Γ΄ Θα προσθέτει για κάθε πελάτη που ο κωδικός του ξεκινά με το χαρακτήρα G στην αμέσως επόμενη θέση έναν πελάτη με κωδικό το αντίστροφο κωδικό του πελάτη και το ίδιο υπόλοιπο λογαριασμού.
Δ΄ Θα διαγράφει όλους τους πελάτες που ο κωδικός τους ξεκινά με το χαρακτήρα Β.

Τα δεδομένα θα αποθηκεύονται σε μια συνδεδεμένη λίστα πραγματοποιώντας χρήση του κώδικα 4.3 καθώς και άλλων συναρτήσεων που επιτρέπουν την αποδοτικότερη υλοποίηση των παραπάνω ερωτημάτων.

```
#include ”linked_list.cpp”
#include <algorithm>
#include <chrono>
#include <iomanip>
#include <iostream>
#include <list>
#include <random>
#include <string>

using namespace std;
using namespace std::chrono;

mt19937 *mt;
uniform_int_distribution<int> uni1(0, 5000), uni2(0, 25);

struct customer {
   string code;
   int balance;
   bool operator<(customer other) { return code < other.code; }
};

string generate_random_code(int k) {
   string code{};
   string letters_en(”ABCDEFGHIJKLMNOPQRSTUVWXYZ”);
   for (int j = 0; j < k; j++) {
      char c{letters_en[uni2(*mt)]};
      code += c;
```

```
28    }
29    return code;
30  }
31
32  void generate_data_linked_list(linked_list<customer> &linked_list, int N) {
33    struct node<customer> *current, *next_customer;
34    current = new node<customer>();
35    current->data.code = generate_random_code(10);
36    current->data.balance = uni1(*mt);
37    current->next = NULL;
38    linked_list.head = current;
39    linked_list.size++;
40    for (int i = 1; i < N; i++) {
41      next_customer = new node<customer>();
42      next_customer->data.code = generate_random_code(10);
43      next_customer->data.balance = uni1(*mt);
44      next_customer->next = NULL;
45      current->next = next_customer;
46      current = next_customer;
47      linked_list.size++;
48    }
49  }
50
51  void print_customers_linked_list(linked_list<customer> &linked_list, int k) {
52    cout << "LIST SIZE=" << linked_list.size << ": ";
53    for (int i = 0; i < k; i++) {
54      customer cu = access(linked_list, i);
55      cout << cu.code << " - " << cu.balance << " ";
56    }
57    cout << endl;
58  }
59
60  void total_balance_linked_list(linked_list<customer> &linked_list, char c) {
61    struct node<customer> *ptr;
62    ptr = linked_list.head;
63    int i = 0;
64    int sum = 0;
65    while (ptr != NULL) {
66      customer cu = ptr->data;
67      if (cu.code.at(0) == c)
68        sum += cu.balance;
69      ptr = ptr->next;
70      i++;
71    }
72    cout << "Total balance for customers having code starting with character "
73         << c << " is " << sum << endl;
74  }
75
76  void add_extra_customers_linked_list(linked_list<customer> &linked_list,
77                                       char c) {
78    struct node<customer> *ptr = linked_list.head;
79    while (ptr != NULL) {
80      customer cu = ptr->data;
81      if (cu.code.at(0) == c) {
82        customer ncu;
83        ncu.code = cu.code;
84        reverse(ncu.code.begin(), ncu.code.end());
85        ncu.balance = cu.balance;
86        struct node<customer> *new_node = new node<customer>();
87        new_node->data = ncu;
88        new_node->next = ptr->next;
```

```
89          ptr->next = new_node;
90          linked_list.size++;
91          ptr = new_node->next;
92        } else
93          ptr = ptr->next;
94      }
95  }
96
97  void remove_customers_linked_list(linked_list<customer> &linked_list, char c) {
98      struct node<customer> *ptr1;
99      while (linked_list.size > 0) {
100       customer cu = linked_list.head->data;
101       if (cu.code.at(0) == c) {
102         ptr1 = linked_list.head;
103         linked_list.head = ptr1->next;
104         delete ptr1;
105         linked_list.size--;
106       } else
107         break;
108     }
109     if (linked_list.size == 0)
110       return;
111     ptr1 = linked_list.head;
112     struct node<customer> *ptr2 = ptr1->next;
113     while (ptr2 != NULL) {
114       customer cu = ptr2->data;
115       if (cu.code.at(0) == c) {
116         ptr1->next = ptr2->next;
117         delete (ptr2);
118         ptr2 = ptr1->next;
119         linked_list.size--;
120       } else {
121         ptr1 = ptr2;
122         ptr2 = ptr2->next;
123       }
124     }
125 }
126
127 int main(int argc, char **argv) {
128     long seed = 1940;
129     mt = new mt19937(seed);
130     cout << "Testing linked list" << endl;
131     struct linked_list<customer> linked_list;
132     string msgs[] = {"A(random customers generation)",
133                      "B(total balance for customers having code starting with A)",
134                      "C(insert new customers)", "D(remove customers)"};
135     for (int i = 0; i < 4; i++) {
136       cout << "############################################################" << endl;
137       auto t1 = high_resolution_clock::now();
138       if (i == 0) {
139         generate_data_linked_list(linked_list, 40000);
140       } else if (i == 1)
141         total_balance_linked_list(linked_list, 'A');
142       else if (i == 2) {
143         add_extra_customers_linked_list(linked_list, 'G');
144       } else if (i == 3) {
145         remove_customers_linked_list(linked_list, 'B');
146       }
147       auto t2 = high_resolution_clock::now();
148       auto duration = duration_cast<microseconds>(t2 - t1).count();
149       print_customers_linked_list(linked_list, 5);
```

```
150     cout << msgs[i] << ". Time elapsed: " << duration << " microseconds "
151          << setprecision(3) << duration / 1000000.0 << " seconds" << endl;
152   }
153   delete mt;
154 }
```

Κώδικας 4.12: Λίστα πελατών για το ίδιο πρόβλημα (lab04_ex2.cpp)

```
Testing linked list
##########################################################
LIST SIZE=40000: GGFSICRZWW – 2722 UBKZNBPWLH – 4019 UPIHSBIIBS – 3896 JRQVGHLTNM – 395 LUWYTFTNFJ – 784
A(random customers generation). Time elapsed: 39002 microseconds 0.039 seconds
##########################################################
Total balance for customers having code starting with character A is 3871562
LIST SIZE=40000: GGFSICRZWW – 2722 UBKZNBPWLH – 4019 UPIHSBIIBS – 3896 JRQVGHLTNM – 395 LUWYTFTNFJ – 784
B(total balance for customers having code starting with A). Time elapsed: 1000 microseconds 0.001 seconds
##########################################################
LIST SIZE=41548: GGFSICRZWW – 2722 WWZRCISFGG – 2722 UBKZNBPWLH – 4019 UPIHSBIIBS – 3896 JRQVGHLTNM – 395
C(insert new customers). Time elapsed: 2000 microseconds 0.002 seconds
##########################################################
LIST SIZE=39928: GGFSICRZWW – 2722 WWZRCISFGG – 2722 UBKZNBPWLH – 4019 UPIHSBIIBS – 3896 JRQVGHLTNM – 395
D(remove customers). Time elapsed: 1000 microseconds 0.001 seconds
```

Αν στη θέση της συνδεδεμένης λίστας του κώδικα 4.3 χρησιμοποιηθεί η στατική λίστα του κώδικα 4.1 ή ένα vector ή ένα list της STL τα αποτελέσματα θα είναι τα ίδια αλλά η απόδοση στα επιμέρους ερωτήματα του παραδείγματος θα διαφέρει όπως φαίνεται στον πίνακα 4.1. Ο κώδικας που παράγει τα αποτελέσματα βρίσκεται στο αρχείο lab04/lab04_ex2_x4.cpp.

	Ερώτημα Α	Ερώτημα Β	Ερώτημα Γ	Ερώτημα Δ
Συνδεδεμένη λίστα	0.030	0.001	0.002	0.001
Στατική λίστα	0.034	0.003	0.642	0.671
std::vector	0.033	0.002	0.543	0.519
std::list	0.033	0.002	0.002	0.001

Πίνακας 4.1: Χρόνοι εκτέλεσης σε δευτερόλεπτα των ερωτημάτων του παραδείγματος 2 ανάλογα με τον τρόπο υλοποίησης της λίστας

4.4 Ασκήσεις

1. Έστω η συνδεδεμένη λίστα που παρουσιάστηκε στον κώδικα 4.3. Προσθέστε μια συνάρτηση έτσι ώστε για μια λίστα ταξινομημένων στοιχείων από το μικρότερο προς το μεγαλύτερο, να προσθέτει ένα ακόμα στοιχείο στην κατάλληλη θέση έτσι ώστε η λίστα να παραμένει ταξινομημένη.

2. Έστω η συνδεδεμένη λίστα που παρουσιάστηκε στον κώδικα 4.3. Προσθέστε μια συνάρτηση που να αντιστρέφει τη λίστα.

3. Υλοποιήστε τη στατική λίστα (κώδικας 4.1) και τη συνδεδεμένη λίστα (κώδικας 4.3) με κλάσεις. Τροποποιήστε το παράδειγμα 1 έτσι ώστε να δίνεται επιλογή στο χρήστη να χρησιμοποιήσει είτε τη στατική είτε τη συνδεδεμένη λίστα προκειμένου να εκτελέσει τις ίδιες λειτουργίες πάνω σε μια λίστα.

4. Υλοποιήστε μια κυκλικά συνδεδεμένη λίστα. Η κυκλική λίστα είναι μια απλά συνδεδεμένη λίστα στην οποία το τελευταίο στοιχείο της λίστας δείχνει στο πρώτο στοιχείο της λίστας. Η υλοποίηση θα πρέπει να συμπεριλαμβάνει και δύο δείκτες, έναν που να δείχνει στο πρώτο στοιχείο της λίστας και έναν που να δείχνει στο τελευταίο στοιχείο της λίστας. Προσθέστε τις απαιτούμενες λειτουργίες έτσι ώστε η λίστα να παρέχει τις ακόλουθες λειτουργίες: εμφάνιση λίστας, εισαγωγή στοιχείου, διαγραφή στοιχείου, εμφάνιση πλήθους στοιχείων, εύρεση στοιχείου. Γράψτε πρόγραμμα που να δοκιμάζει τις λειτουργίες της λίστας.

Βιβλιογραφία

[1] C++ Tutorial-exceptions-2017 by K. Hong, http://www.bogotobogo.com/cplusplus/exceptions.php.

[2] Linked List Basics by N. Parlante, http://cslibrary.stanford.edu/103/.

[3] Linked List Problems by N. Parlante, http://cslibrary.stanford.edu/105/.

[4] Geeks for Geeks, Vector in C++ STL, http://www.geeksforgeeks.org/vector-in-cpp-stl/.

[5] Codecogs, Vector, a random access dynamic container, http://www.codecogs.com/library/computing.

[6] Geeks for Geeks, Deque in C++ STL, http://www.geeksforgeeks.org/deque-cpp-stl/.

[7] Geeks for Geeks, Array class in C++ STL http://www.geeksforgeeks.org/array-class-c/.

[8] Geeks for Geeks, List in C++ STL http://www.geeksforgeeks.org/list-cpp-stl/

[9] Geeks for Geeks, Forward List in C++ (Set 1) http://www.geeksforgeeks.org/forward-list-c-set-1-introduction-important-functions/

[10] Geeks for Geeks, Forward List in C++ (Set 2) http://www.geeksforgeeks.org/forward-list-c-set-2-manipulating-functions/

[11] Geeks for Geeks, C++ bitset and its application, http://www.geeksforgeeks.org/c-bitset-and-its-application/

[12] Geeks for Geeks, C++ bitset interesting facts, http://www.geeksforgeeks.org/c-bitset-interesting-facts/

Εργαστήριο 5

Στοίβες και ουρές, οι δομές στοίβα και ουρά στην STL

5.1 Εισαγωγή

Οι στοίβες και οι ουρές αποτελούν απλές δομές δεδομένων που είναι ιδιαίτερα χρήσιμες στην επίλυση αλγοριθμικών προβλημάτων. Η στοίβα είναι μια λίστα στοιχείων στην οποία τα νέα στοιχεία τοποθετούνται στην κορυφή και όταν πρόκειται να αφαιρεθεί ένα στοιχείο αυτό πάλι συμβαίνει από την κορυφή των στοιχείων της στοίβας. Από την άλλη μεριά, η ουρά είναι επίσης μια λίστα στοιχείων στην οποία όμως οι εισαγωγές γίνονται στο πίσω άκρο της ουράς ενώ οι εξαγωγές πραγματοποιούνται από το εμπρός άκρο της ουράς. Στο εργαστήριο αυτό θα παρουσιαστούν υλοποιήσεις της στοίβας και της ουράς. Επιπλέον, θα παρουσιαστούν οι δομές της STL std::stack και std::queue. Ο κώδικας όλων των παραδειγμάτων βρίσκεται στο https://github.com/chgogos/ceteiep_dsa.

5.2 Στοίβα

Η στοίβα (stack) είναι μια ειδική περίπτωση γραμμικής λίστας στην οποία οι εισαγωγές και οι διαγραφές επιτρέπονται μόνο από το ένα άκρο. Συνήθως αυτό το άκρο λέγεται κορυφή (top). Πρόκειται για μια δομή στην οποία οι εισαγωγές και οι εξαγωγές γίνονται σύμφωνα με τη μέθοδο τελευταίο μέσα πρώτο έξω (LIFO=Last In First Out).

Στον κώδικα 5.1 παρουσιάζεται μια υλοποίηση στοίβας που χρησιμοποιεί για την αποθήκευση των στοιχείων της έναν πίνακα. Εναλλακτικά, στη θέση του πίνακα μπορεί να χρησιμοποιηθεί συνδεδεμένη λίστα. Μια υλοποίηση στη γλώσσα C μπορεί να βρεθεί στην αναφορά [2], ενώ στην εργασία [1] παρουσιάζονται 16(!) διαφορετικοί τρόποι υλοποίησης της στοίβας στην C++.

Στο παράδειγμα που ακολουθεί ωθούνται σε μια στοίβα τα γράμματα της αγγλικής αλφαβήτου (A-Z) και στη συνέχεια απωθούνται ένα προς ένα και μέχρι η στοίβα να αδειάσει.

```
1  #include <iostream>
2
3  using namespace std;
4
5  template <typename T> class my_stack {
6  private:
7    T *data;
8    int top, capacity;
9
10 public:
11   // constructor
12   my_stack(int c) {
```

```
13      top = -1;
14      capacity = c;
15      data = new T[capacity];
16    }
17
18    // destructor
19    ~my_stack() { delete[] data; }
20
21    bool empty() { return (top == -1); }
22
23    void push(T elem) {
24      if (top == (capacity - 1))
25        throw "The stack is full";
26      else {
27        top++;
28        data[top] = elem;
29      }
30    }
31
32    T pop() {
33      if (top == -1)
34        throw "the stack is empty";
35      top--;
36      return data[top + 1];
37    }
38
39    void print() {
40      for (int i = 0; i <= top; i++)
41        cout << data[i] << " ";
42      cout << endl;
43    }
44  };
45
46  int main() {
47    cout << "Custom stack implementation" << endl;
48    my_stack<char> astack(100);
49    for (char c = 65; c < 65 + 26; c++)
50      astack.push(c);
51    astack.print();
52    while (!astack.empty())
53      cout << astack.pop() << " ";
54    cout << endl;
55  }
```

Κώδικας 5.1: Υλοποίηση στοίβας (stack_oo.cpp)

```
1  Custom stack impementation
2  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
3  Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

5.3 Ουρά

Η ουρά (queue) είναι μια ειδική περίπτωση γραμμικής λίστας στην οποία επιτρέπονται εισαγωγές στο πίσω άκρο της και εξαγωγές από το εμπρός άκρο της μόνο. Τα δύο αυτά άκρα συνήθως αναφέρονται ως πίσω (rear) και εμπρός (front) αντίστοιχα. Η ουρά είναι μια δομή στην οποία οι εισαγωγές και οι εξαγωγές γίνονται σύμφωνα με τη μέθοδο πρώτο μέσα πρώτο έξω (FIFO=First In First Out).

Στη συνέχεια παρουσιάζεται μια υλοποίηση ουράς στην οποία τα δεδομένα της τοποθετούνται σε έναν πίνακα (εναλλακτικά θα μπορούσε να είχε χρησιμοποιηθεί μια άλλη δομή όπως για παράδειγμα η συνδεδεμένη

λίστα). Ο πίνακας λειτουργεί κυκλικά, δηλαδή όταν συμπληρωθεί και εφόσον υπάρχουν διαθέσιμες κενές θέσεις στην αρχή του πίνακα, τα νέα στοιχεία που πρόκειται να εισαχθούν στην ουρά τοποθετούνται στις πρώτες διαθέσιμες, ξεκινώντας από την αρχή του πίνακα, θέσεις.

```
#include <iostream>

using namespace std;

template <typename T> class my_queue {
private:
  T *data;
  int front, rear, capacity, size;

public:
  // constructor
  my_queue(int c) {
    front = 0;
    rear = -1;
    size = 0;
    capacity = c;
    data = new T[capacity];
  }

  // destructor
  ~my_queue() { delete[] data; }

  bool empty() { return (size == 0); }

  void enqueue(T elem) {
    if (size == capacity)
      throw "The queue is full";
    else {
      rear++;
      rear %= capacity;
      data[rear] = elem;
      size++;
    }
  }

  T dequeue() {
    if (size == 0)
      throw "the queue is empty";
    T x = data[front];
    front++;
    front %= capacity;
    size--;
    return x;
  }


  void print(bool internal = true) {
    for (int i = front; i < front + size; i++)
      cout << data[i % capacity] << " ";
    cout << endl;
    if (internal){
      for (int i = 0; i < capacity; i++)
        if (front <= rear && i >= front && i <= rear)
          cout << "[" << i << "]->" << data[i] << " ";
        else if (front >= rear && (i >= front || i <= rear))
          cout << "[" << i << "]->" << data[i] << " ";
        else
```

```
58              cout << "[" << i << "]->X ";
59      }
60      cout << " (front:" << front << " rear:" << rear << ")" << endl;
61    }
62 };
63
64 int main() {
65    cout << "Custom queue implementation" << endl;
66    my_queue<int> aqueue(10);
67    cout << "1. Enqueue 10 items" << endl;
68    for (int i = 51; i <= 60; i++)
69       aqueue.enqueue(i);
70    aqueue.print();
71    cout << "2. Dequeue 5 items" << endl;
72    for (int i = 0; i < 5; i++)
73       aqueue.dequeue();
74    aqueue.print();
75    cout << "3. Enqueue 3 items" << endl;
76    for (int i = 61; i <= 63; i++)
77       aqueue.enqueue(i);
78    aqueue.print();
79 }
```

Κώδικας 5.2: Υλοποίηση ουράς (queue_oo.cpp)

```
1  Custom queue implementation
2  1. Enqueue 10 items
3  51 52 53 54 55 56 57 58 59 60
4  [0]->51 [1]->52 [2]->53 [3]->54 [4]->55 [5]->56 [6]->57 [7]->58 [8]->59 [9]->60 (front:0 rear:9)
5  2. Dequeue 5 items
6  56 57 58 59 60
7  [0]->X [1]->X [2]->X [3]->X [4]->X [5]->56 [6]->57 [7]->58 [8]->59 [9]->60 (front:5 rear:9)
8  3. Enqueue 3 items
9  56 57 58 59 60 61 62 63
10 [0]->61 [1]->62 [2]->63 [3]->X [4]->X [5]->56 [6]->57 [7]->58 [8]->59 [9]->60 (front:5 rear:2)
```

5.4 Οι δομές στοίβα και ουρά στην STL

Οι δομές std::stack και std::queue έχουν υλοποιηθεί στην STL ως container adaptors δηλαδή κλάσεις που χρησιμοποιούν εσωτερικά ένα άλλο container και παρέχουν ένα συγκεκριμένο σύνολο από λειτουργίες που επιτρέπουν την προσπέλαση και την τροποποίηση των στοιχείων τους.

5.4.1 std::stack

Το προκαθορισμένο εσωτερικό container που χρησιμοποιεί η std::stack είναι το std::deque. Ωστόσο, μπορούν να χρησιμοποιηθούν και τα std::vector και std::list καθώς και τα τρία αυτά containers παρέχουν τις λειτουργίες empty(), size(), push_back(), pop_back() και back() που απαιτούνται για να υλοποιηθεί το stack interface [3]. Τυπικές λειτουργίες που παρέχει η std::stack είναι οι ακόλουθες:

- empty(), ελέγχει αν η στοίβα είναι άδεια.
- size(), επιστρέφει το μέγεθος της στοίβας.
- top(), προσπελαύνει το στοιχείο που βρίσκεται στη κορυφή της στοίβας (χωρίς να το αφαιρεί).
- push(), ωθεί ένα στοιχείο στη κορυφή της στοίβας
- pop(), αφαιρεί το στοιχείο που βρίσκεται στη κορυφή της στοίβας.

Ένα παράδειγμα χρήσης της std::stack παρουσιάζεται στη συνέχεια.

```
#include <deque>
#include <iostream>
#include <list>
#include <stack>
#include <vector>

using namespace std;
int main(void) {
  cout << "std::stack example" << endl;
  stack<char> items; // adaptor over a deque container
  // stack<char, deque<char>> items; // adaptor over a deque container
  // stack<char, vector<char>> items; // adaptor over a vector container
  // stack<char, list<char>> items; // adaptor over a list container

  for (char c = 65; c < 65 + 26; c++) {
    cout << c << " ";
    items.push(c);
  }
  cout << endl;

  while (!items.empty()) {
    cout << items.top() << " ";
    items.pop();
  }
  cout << endl;
}
```

Κώδικας 5.3: Παράδειγμα χρήσης της std::stack (stl_stack_example.cpp)

```
std::stack example
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
Z Y X W V U T S R Q P O N M L K J I H G F E D C B A
```

5.4.2 std::queue

Στην περίπτωση του std::queue το εσωτερικό container μπορεί να είναι κάποιο από τα containers std::dequeue, std::list (προκαθορισμένη επιλογή) ή οποιοδήποτε container που υποστηρίζει τις λειτουργίες empty(), size(), front(), back(), push_back() και pop_front() [4]. Τυπικές λειτουργίες που παρέχει η std::queue είναι οι ακόλουθες:

- empty(), ελέγχει αν η ουρά είναι άδεια.
- size(), επιστρέφει το μέγεθος της ουράς.
- front(), προσπελαύνει το στοιχείο που βρίσκεται στο εμπρός άκρο της ουράς (χωρίς να το αφαιρεί).
- back(), προσπελαύνει το στοιχείο που βρίσκεται στο πίσω άκρο της ουράς (χωρίς να το αφαιρεί).
- push(), εισάγει ένα στοιχείο στο πίσω άκρο της ουράς
- pop(), εξάγει το στοιχείο που βρίσκεται στο εμπρός άκρο της ουράς.

Ένα παράδειγμα χρήσης της std::queue παρουσιάζεται στη συνέχεια.

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

using namespace std;

int main() {
  cout << "std::queue" << endl;
  queue<int> aqueue; // adaptor over a deque container
```

```
10   // queue<int, deque<int>> aqueue; // adaptor over a deque container
11   // queue<int, list<int>> aqueue; // adaptor over a list container
12
13   cout << "1. Enqueue 10 items" << endl;
14   for (int i = 51; i <= 60; i++) {
15      cout << i << " ";
16      aqueue.push(i);
17   }
18   cout << endl << "2. Dequeue 5 items" << endl;
19   for (int i = 0; i < 5; i++) {
20      cout << aqueue.front() << " ";
21      aqueue.pop();
22   }
23   cout << endl << "3. Enqueue 3 items" << endl;
24   for (int i = 61; i <= 63; i++) {
25      cout << i << " ";
26      aqueue.push(i);
27   }
28   while (!aqueue.empty()) {
29      cout << aqueue.front() << " ";
30      aqueue.pop();
31   }
32   cout << endl;
33 }
```

Κώδικας 5.4: Παράδειγμα χρήσης της std::queue (stl_queue_example.cpp)

```
1 std::queue
2 1. Enqueue 10 items
3 51 52 53 54 55 56 57 58 59 60
4 2. Dequeue 5 items
5 51 52 53 54 55
6 3. Enqueue 3 items
7 60 61 62 63 56 57 58 59 61 62 63
```

5.5 Παραδείγματα

5.5.1 Παράδειγμα 1

Να γραφεί πρόγραμμα που να δέχεται μια φράση ως παράμετρο γραμμής εντολών (command line argument) και να εμφανίζει το εάν είναι παλινδρομική ή όχι. Μια φράση είναι παλινδρομική όταν διαβάζεται η ίδια από αριστερά προς τα δεξιά και από δεξιά προς τα αριστερά.

```
1  #include <iostream>
2  #include <stack>
3
4  using namespace std;
5  // examples of palindromic sentences:
6  // SOFOS, A MAN A PLAN A CANAL PANAMA, AMORE ROMA, LIVE NOT ON EVIL
7  int main(int argc, char **argv) {
8     if (argc != 2) {
9        cerr << "Usage examples: " << endl;
10       cerr << "\t\t " << argv[0] << " SOFOS" << endl;
11       cerr << "\t\t " << argv[0] << " \"A MAN A PLAN A CANAL PANAMA\"" << endl;
12       exit(-1);
13    }
14    string str = argv[1];
15    stack<char> astack;
16    string str1;
17    for (char c : str)
```

```
18      if (c != ' ') {
19          str1 += c;
20          astack.push(c);
21      }
22    string str2;
23    while (!astack.empty()) {
24      str2 += astack.top();
25      astack.pop();
26    }
27    if (str1 == str2)
28      cout << "The sentence " << str << " is palindromic." << endl;
29    else
30      cout << "The string " << str << " is not palindromic." << endl;
31  }
```

Κώδικας 5.5: Έλεγχος παλινδρομικής φράσης (lab05_ex1.cpp)

```
1  $ ./lab05_ex1
2  Usage examples:
3                      ./lab05_ex1 SOFOS
4                      ./lab05_ex1 "A MAN A PLAN A CANAL PANAMA"
5
6  $ ./lab05_ex1 "A MAN A PLAN A A CANAL PANAMA"
7  The string A MAN A PLAN A CANAL PANAMA is palindromic.
8
9  $ ./lab05_ex1 "A MAN A PLAN A A CANAL PANAM"
10 The string A MAN A PLAN A A CANAL PANAM is not palindromic.
```

5.5.2 Παράδειγμα 2

Να γραφεί πρόγραμμα που να δέχεται ένα δυαδικό αριθμό ως λεκτικό και να εμφανίζει την ισοδύναμη δεκαδική του μορφή.

```
1  #include <iostream>
2  #include <stack>
3  #include <string> // stoi
4
5  using namespace std;
6  int main() {
7    string bs;
8    stack<int> astack;
9    cout << "Enter a binary number: ";
10   cin >> bs;
11   for (char c : bs) {
12     if (c != '0' && c != '1') {
13       cerr << "use only digits 0 and 1" << endl;
14       exit(-1);
15     }
16     astack.push(c - '0');
17   }
18
19   int sum = 0, x = 1;
20   while (!astack.empty()) {
21     sum += astack.top() * x;
22     astack.pop();
23     x *= 2;
24   }
25   cout << "Decimal: " << sum << endl;
26
27   cout << "Decimal: " << stoi(bs, nullptr, 2) << endl; // one line solution :)
28 }
```

Κώδικας 5.6: Μετατροπή δυαδικού σε δεκαδικό (lab05_ex2.cpp)

```
1 Enter a binary number: 10101010101010101111111100111
2 Decimal: 178958311
3 Decimal: 178958311
```

5.6 Ασκήσεις

1. Να υλοποιηθεί η δομή της ουράς χρησιμοποιώντας αντικείμενα στοίβας (std::stack) και τις λειτουργίες που επιτρέπονται σε αυτά. Υλοποιήστε τις λειτουργίες της ουράς empty(), size(), enqueue(), dequeue() και front().

2. Να υλοποιηθεί η δομή της στοίβας χρησιμοποιώντας αντικείμενα ουράς (std::queue) και τις λειτουργίες που επιτρέπονται σε αυτά. Υλοποιήστε τις λειτουργίες της στοίβας empty(), size(), push(), pop() και top().

Βιβλιογραφία

[1] Sixteen Ways To Stack a Cat, by Bjarne Stroustrup http://www.stroustrup.com/stack_cat.pdf

[2] Tech Crash Course, C Program to Implement a Stack using Singly Linked List, http://www.techcrashcourse.com/2016/06/c-program-implement-stack-using-linked-list.html

[3] C++ Reference Material by Porter Scobey, The STL stack Container Adaptor http://cs.stmarys.ca/ porter/csc/ref/stl/cont_stack.html

[4] C++ Reference Material by Porter Scobey, The STL queue Container Adaptor http://cs.stmarys.ca/ porter/csc/ref/stl/cont_queue.html

Εργαστήριο 6

Σωροί μεγίστων και σωροί ελαχίστων, η ταξινόμηση heapsort, ουρές προτεραιότητας στην STL

6.1 Εισαγωγή

Οι σωροί επιτρέπουν την οργάνωση των δεδομένων με τέτοιο τρόπο έτσι ώστε το μεγαλύτερο στοιχείο να είναι συνεχώς προσπελάσιμο σε σταθερό χρόνο. Η δε λειτουργίες της εισαγωγής νέων τιμών στη δομή και της διαγραφή της μεγαλύτερης τιμής πραγματοποιούνται ταχύτατα. Σε αυτό το εργαστήριο θα παρουσιαστεί η υλοποίηση ενός σωρού μεγίστων και ο σχετικός με τη δομή αυτή αλγόριθμος ταξινόμησης, heapsort. Επιπλέον, θα παρουσιαστεί η δομή std::priority_queue που υλοποιεί στην STL της C++ τους σωρούς μεγίστων και ελαχίστων. Ο κώδικας όλων των παραδειγμάτων βρίσκεται στο https://github.com/chgogos/ceteiep_dsa.

6.2 Σωροί

Ο σωρός είναι μια μερικά ταξινομημένη δομή δεδομένων. Υπάρχουν δύο βασικά είδη σωρών: ο σωρός μεγίστων (MAXHEAP) και ο σωρός ελαχίστων (MINHEAP). Οι ιδιότητες των σωρών που θα περιγραφούν στη συνέχεια αφορούν τους σωρούς μεγίστων αλλά αντίστοιχες ιδιότητες ισχύουν και για τους σωρούς ελαχίστων. Ειδικότερα, ένας σωρός μεγίστων υποστηρίζει ταχύτατα τις ακόλουθες λειτουργίες:

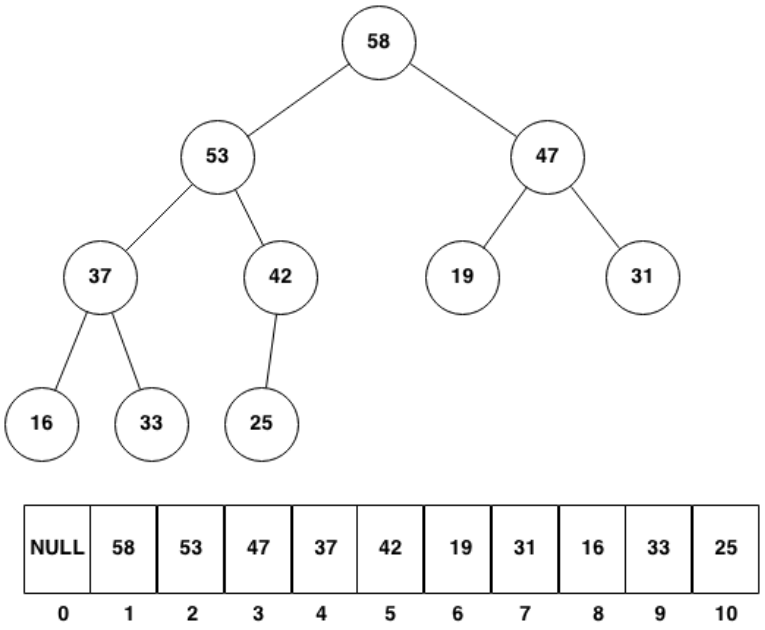
- Εύρεση του στοιχείου με τη μεγαλύτερη τιμή κλειδιού.
- Διαγραφή του στοιχείου με τη μεγαλύτερη τιμή κλειδιού.
- Εισαγωγή νέου κλειδιού στη δομή.

Ένας σωρός μπορεί να θεωρηθεί ως ένα δυαδικό δένδρο για το οποίο ισχύουν οι ακόλουθοι δύο περιορισμοί:

- *Πληρότητα*: το δυαδικό δένδρο είναι συμπληρωμένο, δηλαδή όλα τα επίπεδά του είναι πλήρως συμπληρωμένα εκτός πιθανά από το τελευταίο (χαμηλότερο) επίπεδο στο οποίο μπορούν να λείπουν μόνο κάποια από τα δεξιότερα φύλλα.
- *Κυριαρχία γονέα*: το κλειδί σε κάθε κορυφή είναι μεγαλύτερο ή ίσο από τα κλειδιά των παιδιών (σε MAXHEAP).

Ένας σωρός μπορεί να υλοποιηθεί με ένα πίνακα καταγράφοντας στον πίνακα στη σειρά τα στοιχεία του δυαδικού δένδρου από αριστερά προς τα δεξιά και από πάνω προς τα κάτω (σχήμα 6.1). Μερικές σημαντικές ιδιότητες οι οποίες προκύπτουν εφόσον τηρηθεί ο παραπάνω τρόπος αντιστοίχισης των στοιχείων του δένδρου στα στοιχεία του πίνακα είναι οι ακόλουθες:

- Στον πίνακα, τα κελιά γονείς βρίσκονται στις πρώτες $\lfloor \frac{n}{2} \rfloor$ θέσεις ενώ τα φύλλα καταλαμβάνουν τις υπόλοιπες θέσεις.
- Στον πίνακα, τα παιδιά για κάθε κλειδί στις θέσεις i από 1 μέχρι και $\lfloor \frac{n}{2} \rfloor$ βρίσκονται στις θέσεις $2 * i$ και $2 * i + 1$.
- Στον πίνακα, ο γονέας για κάθε κλειδί στις θέσεις i από 2 μέχρι και n βρίσκεται στη θέση $\lfloor \frac{i}{2} \rfloor$.



Σχήμα 6.1: Αναπαράσταση ενός σωρού μεγίστων ως πίνακα

Για το παράδειγμα του σχήματος ισχύουν τα ακόλουθα:

- Οι κόμβοι που είναι γονείς (έχουν τουλάχιστον ένα παιδί) βρίσκονται στις θέσεις από 1 μέχρι και 5.
- Οι κόμβοι που είναι φύλλα βρίσκονται στις θέσεις από 6 μέχρι και 10.
- Ο γονέας στη θέση 1 (η τιμή 58) έχει παιδιά στις θέσεις $2 * 1 = 2$ (τιμή 53) και $2 * 1 + 1 = 3$ (τιμή 47).
- Ο γονέας στη θέση 2 (η τιμή 53) έχει παιδιά στις θέσεις $2 * 2 = 4$ (τιμή 37) και $2 * 2 + 1 = 5$ (τιμή 42).
- Ο γονέας στη θέση 3 (η τιμή 47) έχει παιδιά στις θέσεις $2 * 3 = 6$ (τιμή 19) και $2 * 3 + 1 = 7$ (τιμή 31).
- Ο γονέας στη θέση 4 (η τιμή 37) έχει παιδιά στις θέσεις $2 * 4 = 8$ (τιμή 16) και $2 * 4 + 1 = 9$ (τιμή 33).
- Ο γονέας στη θέση 5 (η τιμή 42) έχει παιδιά στις θέσεις $2 * 5 = 10$ (τιμή 25).
- Ο κόμβος παιδί στη θέση 2 (η τιμή 53) έχει γονέα στη θέση $\lfloor \frac{2}{2} \rfloor = 1$ (τιμή 58).
- Ο κόμβος παιδί στη θέση 3 (η τιμή 47) έχει γονέα στη θέση $\lfloor \frac{3}{2} \rfloor = 1$ (τιμή 58).
- Ο κόμβος παιδί στη θέση 4 (η τιμή 37) έχει γονέα στη θέση $\lfloor \frac{4}{2} \rfloor = 2$ (τιμή 53).
- Ο κόμβος παιδί στη θέση 5 (η τιμή 42) έχει γονέα στη θέση $\lfloor \frac{5}{2} \rfloor = 2$ (τιμή 53).
- Ο κόμβος παιδί στη θέση 6 (η τιμή 19) έχει γονέα στη θέση $\lfloor \frac{6}{2} \rfloor = 3$ (τιμή 47).
- Ο κόμβος παιδί στη θέση 7 (η τιμή 31) έχει γονέα στη θέση $\lfloor \frac{7}{2} \rfloor = 3$ (τιμή 47).

- Ο κόμβος παιδί στη θέση 8 (η τιμή 16) έχει γονέα στη θέση $\lfloor \frac{8}{2} \rfloor = 4$ (τιμή 37).
- Ο κόμβος παιδί στη θέση 9 (η τιμή 33) έχει γονέα στη θέση $\lfloor \frac{9}{2} \rfloor = 4$ (τιμή 37).
- Ο κόμβος παιδί στη θέση 10 (η τιμή 25) έχει γονέα στη θέση $\lfloor \frac{10}{2} \rfloor = 5$ (τιμή 42).

6.3 Υλοποίηση ενός σωρού

Στη συνέχεια παρουσιάζεται η υλοποίηση ενός σωρού μεγίστων που περιέχει ακέραιες τιμές-κλειδιά.

```
#include <iostream>
using namespace std;

// MAXHEAP
const int static HEAP_SIZE_LIMIT = 100000;
int heap[HEAP_SIZE_LIMIT + 1];
int heap_size = 0;

void clear_heap() {
  for (int i = 0; i < HEAP_SIZE_LIMIT + 1; i++)
    heap[i] = 0;
  heap_size = 0;
}

void print_heap(bool newline = true) {
  cout << "HEAP(" << heap_size << ") [";
  for (int i = 1; i <= heap_size; i++)
    if (i == heap_size)
      cout << heap[i];
    else
      cout << heap[i] << " ";
  cout << "]";
  if (newline)
    cout << endl;
}

void heapify(int k) {
  int v = heap[k];
  bool flag = false;
  while (!flag && 2 * k <= heap_size) {
    int j = 2 * k;
    if (j < heap_size)
      if (heap[j] < heap[j + 1])
        j++;
    if (v >= heap[j])
      flag = true;
    else {
      heap[k] = heap[j];
      k = j;
    }
  }
  heap[k] = v;
}

void heap_bottom_up(int *a, int N, bool verbose = false) {
  heap_size = N;
  for (int i = 0; i < N; i++)
    heap[i + 1] = a[i];
  for (int i = heap_size / 2; i >= 1; i--) {
    if (verbose)
```

```
51          cout << "heapify " << heap[i] << " ";
52        heapify(i);
53        if (verbose)
54          print_heap();
55      }
56  }
57
58  bool empty() {
59    return (heap_size==0);
60  }
61
62  int top() { return heap[1]; }
63
64  void push(int key) {
65    heap_size++;
66    heap[heap_size] = key;
67    int pos = heap_size;
68    while (pos != 1 && heap[pos / 2] < heap[pos]) {
69      swap(heap[pos / 2], heap[pos]);
70      pos = pos / 2;
71    }
72  }
73
74  void pop() {
75    swap(heap[1], heap[heap_size]);
76    heap_size--;
77    heapify(1);
78  }
```

Κώδικας 6.1: Σωρός μεγίστων με κλειδιά ακέραιες τιμές (max_heap.cpp)

Οι συναρτήσεις δημιουργίας σωρού από πίνακα, heap_bottom_up() και heapify()

Ένας πίνακας μπορεί να μετασχηματιστεί ταχύτατα σε σωρό. Η διαδικασία ξεκινά από τον τελευταίο κόμβο γονέα του δένδρου (που βρίσκεται στη θέση $\lfloor \frac{n}{2} \rfloor$) και σταδιακά εφαρμόζεται μέχρι να φτάσει στον κόμβο στη θέση 1. Για καθένα από αυτούς τους κόμβους εξετάζεται από πάνω προς τα κάτω αν ισχύει η κυριαρχία γονέα και αν δεν ισχύει τότε γίνεται αντιμετάθεση με το μεγαλύτερο από τα παιδιά του επαναληπτικά.

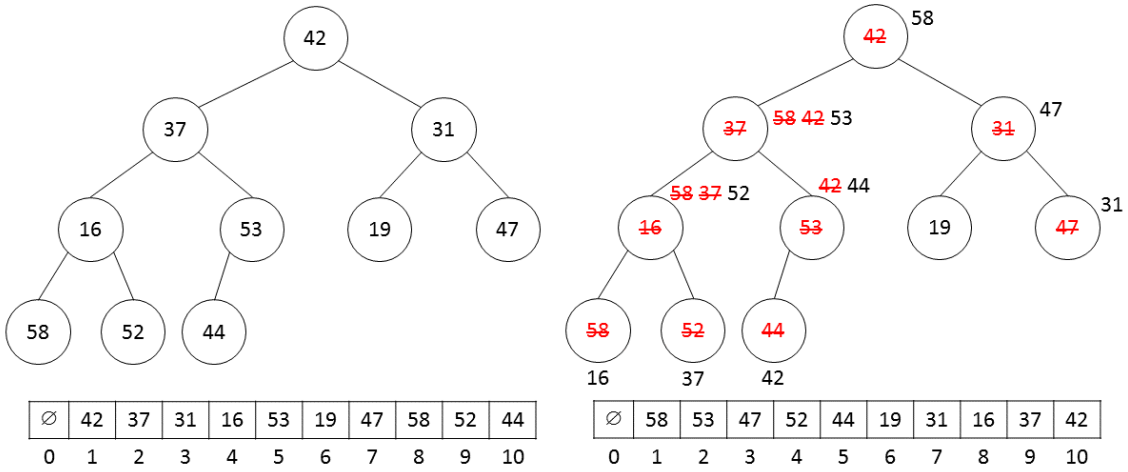
Ο ακόλουθος κώδικας χρησιμοποιεί τη συνάρτηση heap_bottom_up() και μέσω αυτής τη συνάρτηση heapify() προκειμένου να μετασχηματίσει έναν πίνακα ακεραίων σε σωρό μεγίστων.

```
1  #include "max_heap.cpp"
2
3  int main(void) {
4    cout << "#### Test heap construction with heapify ####" << endl;
5    int a[10] = {42, 37, 31, 16, 53, 19, 47, 58, 52, 44};
6    heap_bottom_up(a, 10, true);
7    print_heap();
8  }
```

Κώδικας 6.2: Δημιουργία σωρού από πίνακα με heapify (heap1.cpp)

```
1  #### Test heap construction with heapify ####
2  heapify 53 HEAP(10) [42 37 31 16 53 19 47 58 52 44]
3  heapify 16 HEAP(10) [42 37 31 58 53 19 47 16 52 44]
4  heapify 31 HEAP(10) [42 37 47 58 53 19 31 16 52 44]
5  heapify 37 HEAP(10) [42 58 47 52 53 19 31 16 37 44]
6  heapify 42 HEAP(10) [58 53 47 52 44 19 31 16 37 42]
7  HEAP(10) [58 53 47 52 44 19 31 16 37 42]
```

Στο σχήμα 6.2 παρουσιάζονται οι τιμές που έλαβε κάθε κόμβος του δένδρου προκειμένου να μετασχηματιστεί τελικά σε σωρό μεγίστων.



Σχήμα 6.2: Δημιουργία σωρού από πίνακα (heapify)

Η συνάρτηση ελέγχου του εάν ο σωρός είναι άδειος, empty()

Η συνάρτηση empty εξετάζει το μέγεθος του σωρού μέσω της μεταβλητής heap_size. Αν η μεταβλητή heap_size είναι μηδέν τότε επιστρέφει true, αλλιώς επιστρέφει false.

Η συνάρτηση λήψης της μεγαλύτερης τιμής από το σωρό, top()

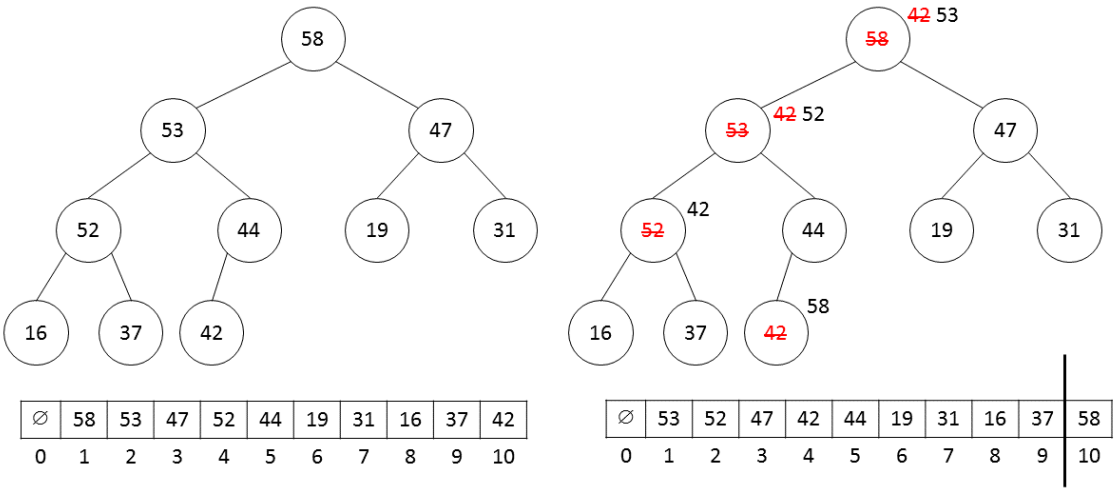
Καθώς η μεγαλύτερη τιμή βρίσκεται πάντα στη θέση 1 του πίνακα που διατηρεί τα δεδομένα του σωρού η συνάρτηση top απλά επιστρέφει την τιμή αυτή.

Η συνάρτηση εξαγωγής της μεγαλύτερης τιμής από το σωρό, pop()

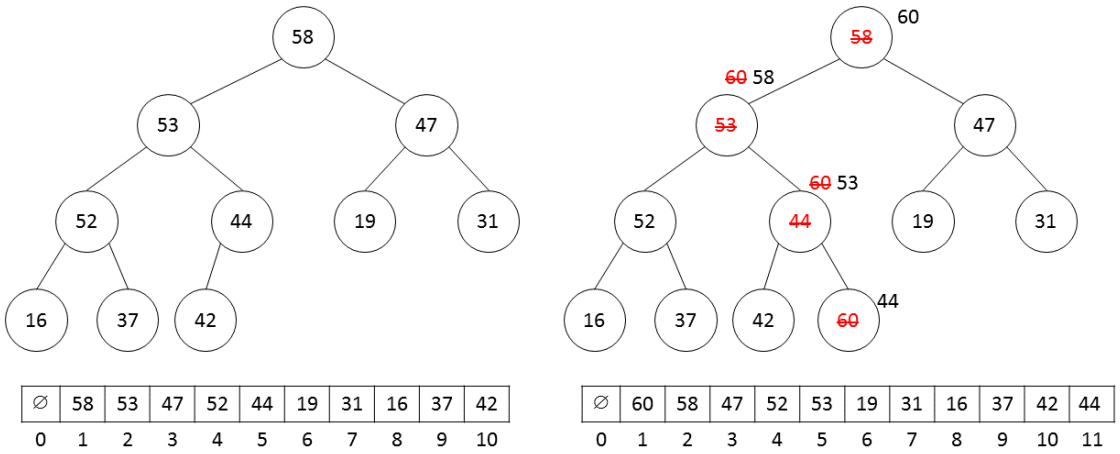
Η εξαγωγή της μεγαλύτερης τιμής γίνεται ως εξής. Το στοιχείο που βρίσκεται στην κορυφή του σωρού αντιμετατίθεται με το τελευταίο στοιχείο του σωρού. Στη συνέχεια το στοιχείο που έχει βρεθεί στην κορυφή του σωρού κατεβαίνει προς τα κάτω αν έχει παιδί που είναι μεγαλύτερό του πραγματοποιώντας αντιμετάθεση με το μεγαλύτερο στοιχείο από τα παιδιά του. Η διαδικασία επαναλαμβάνεται για τη νέα θέση του στοιχείου που αρχικά είχε μεταφερθεί στη κορυφή και μέχρι να ισχύσει ότι είναι μεγαλύτερο και από τα δύο παιδιά του. Στο σχήμα 6.3 παρουσιάζεται η εξαγωγή της κορυφαίας τιμής του σωρού.

Η συνάρτηση εισαγωγής νέας τιμής στο σωρό, push()

Η εισαγωγή ενός στοιχείου γίνεται ως φύλλο στη πρώτη διαθέσιμη θέση από πάνω προς τα κάτω και από δεξιά προς τα αριστερά. Το στοιχείο αυτό συγκρίνεται με το γονέα του και αν είναι μεγαλύτερο αντιμετατίθεται με αυτόν. Η διαδικασία συνεχίζεται μέχρι είτε να βρεθεί το νέο στοιχείο στην κορυφή είτε να ισχύει η κυριαρχία γονέα. Στο σχήμα 6.4 παρουσιάζεται η εισαγωγή της τιμής 60 σε έναν σωρό μεγίστων.



Σχήμα 6.3: Εξαγωγή της μεγαλύτερης τιμής του σωρού (pop)



Σχήμα 6.4: Εισαγωγή της τιμής 60 στο σωρό (push)

Παράδειγμα χρήσης των συναρτήσεων push() και pop()

Ο ακόλουθος κώδικας δημιουργεί σταδιακά έναν σωρό εισάγοντας δέκα τιμές με τη συνάρτηση push(). Στη συνέχεια πραγματοποιούνται εξαγωγές τιμών με τη συνάρτηση pop() μέχρι ο σωρός να αδειάσει.

```
1  #include ”max_heap.cpp”
2
3  int main(void) {
4    int a[10] = {42, 37, 31, 16, 53, 19, 47, 58, 33, 25};
5    for (int i = 0; i < 10; i++) {
6      print_heap(false);
7      cout << ” ==> push key ” << a[i] << ” ==> ”;
8      push(a[i]);
9      print_heap();
10   }
11   while (heap_size > 0) {
12     print_heap(false);
13     cout << ” ==> pop ==> key=” << heap[1] << ”, ”;
14     pop();
```

```
15      print_heap();
16    }
17  }
```

Κώδικας 6.3: Δημιουργία σωρού με εισαγωγές τιμών και εν συνεχεία άδειασμα του σωρού με διαδοχικές διαγραφές της μέγιστης τιμής (heap2.cpp)

```
1  HEAP(0) [] ==> push key 42 ==> HEAP(1) [42]
2  HEAP(1) [42] ==> push key 37 ==> HEAP(2) [42 37]
3  HEAP(2) [42 37] ==> push key 31 ==> HEAP(3) [42 37 31]
4  HEAP(3) [42 37 31] ==> push key 16 ==> HEAP(4) [42 37 31 16]
5  HEAP(4) [42 37 31 16] ==> push key 53 ==> HEAP(5) [53 42 31 16 37]
6  HEAP(5) [53 42 31 16 37] ==> push key 19 ==> HEAP(6) [53 42 31 16 37 19]
7  HEAP(6) [53 42 31 16 37 19] ==> push key 47 ==> HEAP(7) [53 42 47 16 37 19 31]
8  HEAP(7) [53 42 47 16 37 19 31] ==> push key 58 ==> HEAP(8) [58 53 47 42 37 19 31 16]
9  HEAP(8) [58 53 47 42 37 19 31 16] ==> push key 33 ==> HEAP(9) [58 53 47 42 37 19 31 16 33]
10 HEAP(9) [58 53 47 42 37 19 31 16 33] ==> push key 25 ==> HEAP(10) [58 53 47 42 37 19 31 16 33 25]
11 HEAP(10) [58 53 47 42 37 19 31 16 33 25] ==> pop ==> key=58, HEAP(9) [53 42 47 33 37 19 31 16 25]
12 HEAP(9) [53 42 47 33 37 19 31 16 25] ==> pop ==> key=53, HEAP(8) [47 42 31 33 37 19 25 16]
13 HEAP(8) [47 42 31 33 37 19 25 16] ==> pop ==> key=47, HEAP(7) [42 37 31 33 16 19 25]
14 HEAP(7) [42 37 31 33 16 19 25] ==> pop ==> key=42, HEAP(6) [37 33 31 25 16 19]
15 HEAP(6) [37 33 31 25 16 19] ==> pop ==> key=37, HEAP(5) [33 25 31 19 16]
16 HEAP(5) [33 25 31 19 16] ==> pop ==> key=33, HEAP(4) [31 25 16 19]
17 HEAP(4) [31 25 16 19] ==> pop ==> key=31, HEAP(3) [25 19 16]
18 HEAP(3) [25 19 16] ==> pop ==> key=25, HEAP(2) [19 16]
19 HEAP(2) [19 16] ==> pop ==> key=19, HEAP(1) [16]
20 HEAP(1) [16] ==> pop ==> key=16, HEAP(0) []
```

6.4 Ταξινόμηση Heapsort

Ο αλγόριθμος Heapsort προτάθηκε από τον J.W.J.Williams το 1964 [1] και αποτελείται από 2 στάδια:

- Δημιουργία σωρού με τα n στοιχεία ενός πίνακα που ζητείται να ταξινομηθούν.
- Εφαρμογή της διαγραφής της ρίζας n -1 φορές.

Το αποτέλεσμα είναι ότι τα στοιχεία αφαιρούνται από το σωρό σε φθίνουσα σειρά (για έναν σωρό μεγίστων). Καθώς κατά την αφαίρεσή του κάθε στοιχείου, αυτό τοποθετείται στο τέλος του σωρού, τελικά ο σωρός περιέχει τα αρχικά δεδομένα σε αύξουσα σειρά.

Στη συνέχεια παρουσιάζεται η υλοποίηση του αλγορίθμου Heapsort. Επιπλέον ο κώδικας ταξινομεί πίνακες μεγέθους 10.000, 20.000, 40.000 80.000 και 100.000 που περιέχουν τυχαίες ακέραιες τιμές και πραγματοποιείται σύγκριση με τους χρόνους εκτέλεσης που επιτυγχάνει η std::sort().

```
1  #include "max_heap.cpp"
2  #include <algorithm>
3  #include <chrono>
4  #include <random>
5
6  using namespace std::chrono;
7
8  void heapsort() {
9    while (!empty())
10     pop();
11 }
12
13 int main(void) {
14   high_resolution_clock::time_point t1, t2;
15   mt19937 mt(1940);
16   uniform_int_distribution<int> uni(0, 200000);
17   int problem_sizes[] = {10000, 20000, 40000, 80000, 100000};
18   for (int i = 0; i < 5; i++) {
```

```
19      clear_heap();
20      int N = problem_sizes[i];
21      int *a = new int[N];
22      for (int i = 0; i < N; i++)
23        a[i] = uni(mt);
24      heap_bottom_up(a, N);
25      t1 = high_resolution_clock::now();
26      heapsort();
27      t2 = high_resolution_clock::now();
28      duration<double, std::milli> duration1 = t2 - t1;
29      for (int i = 0; i < N; i++)
30        a[i] = uni(mt);
31      t1 = high_resolution_clock::now();
32      sort(a, a + N);
33      t2 = high_resolution_clock::now();
34      duration<double, std::milli> duration2 = t2 - t1;
35      cout << "SIZE " << N << " heap sort " << duration1.count()
36           << "ms std::sort " << duration2.count() << "ms" << endl;
37      delete[] a;
38    }
39  }
```

Κώδικας 6.4: Ο αλγόριθμος heapsort (heapsort.cpp)

```
1  SIZE 10000 heap sort 4.0003ms std::sort 4.0003ms
2  SIZE 20000 heap sort 5.0003ms std::sort 4.0002ms
3  SIZE 40000 heap sort 10.0006ms std::sort 10.0006ms
4  SIZE 80000 heap sort 19.0011ms std::sort 18.001ms
5  SIZE 100000 heap sort 24.0014ms std::sort 22.0013ms
```

Περισσότερες πληροφορίες για την ταξινόμηση heapsort μπορούν να βρεθούν στην αναφορά [2].

6.5 Η δομή priority_queue της STL

Η STL της C++ περιέχει υλοποίηση της δομής std::priority_queue (ουρά προτεραιότητας) η οποία είναι ένας σωρός μεγίστων. Κάθε στοιχείο που εισέρχεται στην ουρά προτεραιότητας έχει μια προτεραιότητα που συνδέεται με αυτό και το στοιχείο με τη μεγαλύτερη προτεραιότητα βρίσκεται πάντα στην αρχή της ουράς. Οι κυριότερες λειτουργίες που υποστηρίζονται από την std::priority_queue είναι οι ακόλουθες:

- push(), εισαγωγή ενός στοιχείου στη δομή.
- top(), επιστροφή χωρίς εξαγωγή του στοιχείου με τη μεγαλύτερη προτεραιότητα.
- pop(), απώθηση του στοιχείου με τη μεγαλύτερη προτεραιότητα.
- size(), πλήθος των στοιχείων που υπάρχουν στη δομή.
- empty(), επιστρέφει true αν η δομή είναι άδεια αλλιώς επιστρέφει false.

Η std::priority_queue είναι ένας container adaptor που χρησιμοποιεί ως εσωτερικό container ένα std::vector. Εναλλακτικά μπορεί να χρησιμοποιηθεί std::deque το οποίο όπως και το std::vector παρέχει τις λειτουργίες empty(), size(), push_back(), pop_back() και front() που απαιτούνται.

Ένα παράδειγμα χρήσης της std::priority_queue ως σωρού μεγίστων αλλά και ως σωρού ελαχίστων παρουσιάζεται στη συνέχεια.

```
1  #include <algorithm>
2  #include <iostream>
3  #include <queue>
4
5  using namespace std;
6
7  int main(void) {
```

```
8    int a[10] = {15, 16, 13, 23, 45, 67, 11, 22, 37, 10};
9    cout << "priority queue (MAXHEAP): ";
10   priority_queue<int> pq1(a, a + 10);
11   while (!pq1.empty()) {
12      int x = pq1.top();
13      pq1.pop();
14      cout << x << " ";
15   }
16   cout << endl;
17
18   cout << "priority queue (MINHEAP): ";
19   priority_queue<int, std::vector<int>, std::greater<int>> pq2(a, a + 10);
20   while (!pq2.empty()) {
21      int x = pq2.top();
22      pq2.pop();
23      cout << x << " ";
24   }
25   cout << endl;
26 }
```

Κώδικας 6.5: Παράδειγμα με priority_queue της STL (stl_priority_queue.cpp)

```
1 priority queue (MAXHEAP): 67 45 37 23 22 16 15 13 11 10
2 priority queue (MINHEAP): 10 11 13 15 16 22 23 37 45 67
```

Περισσότερες πληροφορίες για τη δομή std::priority_queue μπορούν να βρεθούν στις αναφορές [3] και [4].

6.6 Παραδείγματα

6.6.1 Παράδειγμα 1

Χρησιμοποιώντας τον κώδικα 1, να γραφεί πρόγραμμα που να εισάγει 100.000 τυχαίες ακέραιες τιμές (στο διάστημα [-1.000.000,1.000.000]) σε έναν σωρό μεγίστων με τη συνάρτηση heap_bottom_up() καθώς και με διαδοχικές κλήσεις της συνάρτησης push(). Χρονομετρείστε τον κώδικα και στις δύο περιπτώσεις δημιουργίας του σωρού και εμφανίστε το κορυφαίο στοιχείο του σωρού. Επαναλάβετε τη διαδικασία χρησιμοποιώντας την std::priority_queue.

```
1  #include "max_heap.cpp"
2  #include <chrono>
3  #include <queue>
4  #include <random>
5
6
7  using namespace std::chrono;
8
9  int main(void) {
10   constexpr int N = 100000;
11   mt19937 mt(1821);
12   int a[N];
13   uniform_int_distribution<int> dist(−1000000, 1000000);
14   for (int i = 0; i < N; i++)
15      a[i] = dist(mt);
16
17   auto t1 = high_resolution_clock::now();
18   heap_bottom_up(a, N, false);
19   auto t2 = high_resolution_clock::now();
20   std::chrono::duration<double, std::micro> duration_micro_sec = t2 − t1;
21   cout << "A. Top item: " << top() << endl;
22   cout << "Time elapsed (heap_bottom_up): " << duration_micro_sec.count()
```

```
        << " microseconds " << endl;

    clear_heap();

    t1 = high_resolution_clock::now();
    for (int i = 0; i < N; i++)
        push(a[i]);
    t2 = high_resolution_clock::now();
    duration_micro_sec = t2 - t1;
    cout << "B. Top item: " << top() << endl;
    cout << "Time elapsed (push): " << duration_micro_sec.count()
        << " microseconds " << endl;

    t1 = high_resolution_clock::now();
    priority_queue<int> pq(a, a + N);
    t2 = high_resolution_clock::now();
    duration_micro_sec = t2 - t1;
    cout << "C. Top item: " << pq.top() << endl;
    cout << "Time elapsed (push): " << duration_micro_sec.count()
        << " microseconds " << endl;
}
```

Κώδικας 6.6: Χρόνος δημιουργίας MAXHEAP: A) με την heap_bottom_up() B) με σταδιακές εισαγωγές (push) τιμών στο σωρό και C) με την std::priority_queue (lab06_ex1.cpp)

```
A. Top item: 999994
Time elapsed (heap_bottom_up): 3000.2 microseconds
B. Top item: 999994
Time elapsed (push): 6000.4 microseconds
C. Top item: 999994
Time elapsed (push): 11000.6 microseconds
```

6.6.2 Παράδειγμα 2

Έστω ένα παιχνίδι στο οποίο οι παίκτες έχουν όνομα (name) και επίδοση (score). Να γράψετε πρόγραμμα στο οποίο να εισέρχονται στο παιχνίδι 10 παίκτες στη σειρά (player1, player2, ...), πετυχαίνοντας κάποια επίδοση ο καθένας (τυχαίος ακέραιος από το 0 μέχρι το 50.000). Να εμφανίζεται μετά την εισαγωγή του κάθε παίκτη ο παίκτης που προηγείται και η επίδοση του. Τέλος, να εμφανίζονται τα ονόματα των παικτών με τις 3 υψηλότερες επιδόσεις.

```
#include <iostream>
#include <queue>
#include <random>
#include <string>
#define N 10
#define TOP 3

using namespace std;
struct player {
    string name;
    int score;
    bool operator<(const player &other) const { return score < other.score; }
};

int main() {
    mt19937 mt(1821);
    uniform_int_distribution<int> dist(0, 50000);
    priority_queue<player> pq;
    int best_score = -1;
    for (int i = 0; i < N; i++) {
```

```
21      player p;
22      p.name = ”player” + to_string(i + 1);
23      p.score = dist(mt);
24      pq.push(p);
25      player top_player = pq.top();
26      if (top_player.score != best_score)
27        best_score = top_player.score;
28      cout << ”New player: ” << p.name << ” with score ” << p.score << ” best[”
29           << top_player.name << ” score ” << top_player.score << ”]” << endl;
30    }
31    cout << ”Top ” << TOP << ” players:” << endl;
32    for (int i = 0; i < TOP; i++) {
33      player p = pq.top();
34      cout << i + 1 << ” ” << p.name << ” ” << p.score << endl;
35      pq.pop();
36    }
37  }
```

Κώδικας 6.7: Διατήρηση επιδόσεων σε σωρό (lab06_ex2.cpp)

```
1  New player: player1 with score 36323 best[player1 score 36323]
2  New player: player2 with score 21613 best[player1 score 36323]
3  New player: player3 with score 33218 best[player1 score 36323]
4  New player: player4 with score 32634 best[player1 score 36323]
5  New player: player5 with score 454 best[player1 score 36323]
6  New player: player6 with score 48987 best[player6 score 48987]
7  New player: player7 with score 25627 best[player6 score 48987]
8  New player: player8 with score 42239 best[player6 score 48987]
9  New player: player9 with score 9284 best[player6 score 48987]
10 New player: player10 with score 11639 best[player6 score 48987]
11 Top 3 players:
12 1 player6 48987
13 2 player8 42239
14 3 player1 36323
```

6.6.3 Παράδειγμα 3

Διάμεσος ενός δείγματος N παρατηρήσεων οι οποίες έχουν διαταχθεί σε αύξουσα σειρά ορίζεται ως η μεσαία παρατήρηση, όταν το N είναι περιττός αριθμός, ή ο μέσος όρος (ημιάθροισμα) των δύο μεσαίων παρατηρήσεων όταν το N είναι άρτιος αριθμός. Έστω ότι για διάφορες τιμές που παράγονται με κάποιον τρόπο ζητείται ο υπολογισμός της διάμεσης τιμής καθώς παράγεται κάθε νέα τιμή και για όλες τις τιμές που έχουν προηγηθεί μαζί με την τρέχουσα τιμή όπως φαίνεται στο επόμενο παράδειγμα:
5 ⇒ διάμεσος 5
5, 7 ⇒ διάμεσος 6
5, 7, 13 ⇒ διάμεσος 7
5, 7, 13, 12 ⇒ 5, 7, 12, 13 ⇒ διάμεσος 9.5
5, 7, 13, 12, 2 ⇒ 2, 5, 7, 12, 13 ⇒ διάμεσος 7

```
1  #include <chrono>
2  #include <iomanip>
3  #include <iostream>
4  #include <queue>
5  #include <random>
6
7  using namespace std;
8  using namespace std::chrono;
9
10 double medians(int a[], int N) {
11   priority_queue<int, std::vector<int>, std::less<int>> pq1;
12   priority_queue<int, std::vector<int>, std::greater<int>> pq2;
```

```
13    int first = a[0];
14    int second = a[1];
15    if (first < second) {
16      pq1.push(first);
17      pq2.push(second);
18    } else {
19      pq2.push(first);
20      pq1.push(second);
21    }
22    double sum = first + (first + second) / 2.0;
23    for (int i = 2; i < N; i++) {
24      int x = a[i];
25      if (x <= pq1.top())
26        pq1.push(x);
27      else
28        pq2.push(x);
29      if (pq1.size() < pq2.size()) {
30        pq1.push(pq2.top());
31        pq2.pop();
32      }
33      double median;
34      if (pq1.size() == pq2.size())
35        median = (pq1.top() + pq2.top()) / 2.0;
36      else
37        median = pq1.top();
38      sum += median;
39    }
40    return sum;
41  }
42
43  int main(int argc, char **argv) {
44    high_resolution_clock::time_point t1, t2;
45    t1 = high_resolution_clock::now();
46    mt19937 mt(1940);
47    uniform_int_distribution<int> uni(0, 200000);
48    int N = 500000;
49    int *a = new int[N];
50    for (int i = 0; i < N; i++)
51      a[i] = uni(mt);
52    double sum = medians(a, N);
53    delete[] a;
54    t2 = high_resolution_clock::now();
55    duration<double, std::milli> duration = t2 - t1;
56    cout.precision(2);
57    cout << "Moving medians sum = " << std::fixed << sum << " elapsed time "
58         << duration.count() << "ms" << endl;
59  }
```

Κώδικας 6.8: Υπολογισμός διαμέσου σε μια ροή τιμών (lab06_ex3.cpp)

```
1  Moving medians sum = 54441518145.50 elapsed time 132.52ms
```

6.7 Ασκήσεις

1. Να υλοποιηθεί ο σωρός μεγίστων που παρουσιάστηκε στον κώδικα 1 ως κλάση. Προσθέστε εξαιρέσεις έτσι ώστε να χειρίζονται περιπτώσεις όπως όταν ο σωρός είναι άδειος και ζητείται εξαγωγή της μεγαλύτερης τιμής ή όταν ο σωρός είναι γεμάτος και επιχειρείται εισαγωγή νέας τιμής.

2. Να γραφεί συνάρτηση που να δέχεται ως παράμετρο έναν πίνακα ακεραίων και έναν ακέραιο αριθμό κ και να επιστρέφει το κ-οστό μεγαλύτερο στοιχείο του πίνακα.

Βιβλιογραφία

[1] NIST, heapsort, https://xlinux.nist.gov/dads/HTML/heapSort.html

[2] PROGRAMIZ, Heap Sort Algorithm, https://www.programiz.com/dsa/heap-sort

[3] Geeks for Geeks, Priority Queue in C++ Standard Template Library (STL), http://www.geeksforgeeks.org/priority-queue-in-cpp-stl/

[4] Cppreference.com, std::priority_queue, http://en.cppreference.com/w/cpp/container/priority_queue

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.

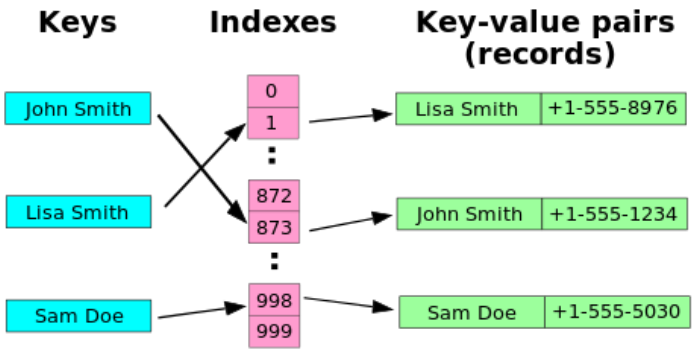
7.2 Τι είναι ο κατακερματισμός;

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]

```
  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 7.1: 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) {
```

```
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
```

```cpp
int main() {
  set<int> aset;
  int collisions = 0;
  for (int i = 0; i < 2000; i++) {
    string key = generate_random_string(10);
    size_t h = hash0(key) % HT_SIZE; // 1863 collisions
    // size_t h = hash1(key) % HT_SIZE; // 172 collisions
    // size_t h = hash2(key) % HT_SIZE; // 188 collisions
    // size_t h = hash3(key) % HT_SIZE; // 196 collisions
    if (aset.find(h) != aset.end())
      collisions++;
    else
      aset.insert(h);
  }
  cout << "number of collisions " << collisions << endl;
}
```

Κώδικας 7.4: Συγκρούσεις (hashes_ex2.cpp)

```
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.

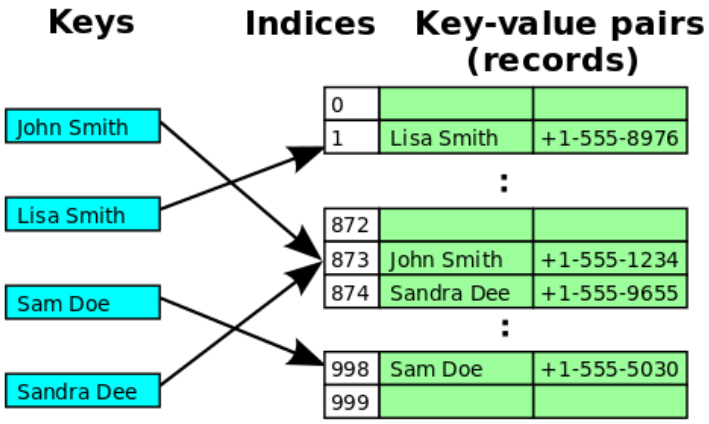
Αν το πλήθος των κελιών είναι m και το πλήθος των εγγραφών είναι n τότε το πηλίκο $a = \frac{n}{m}$ που ονομάζεται παράγοντας φόρτωσης (load factor) καθορίζει σημαντικά την απόδοση του hashtable. Ο παράγοντας φόρτωσης στην περίπτωση της ανοικτής διευθυνσιοδότησης δεν μπορεί να είναι μεγαλύτερος της μονάδας.

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.

```cpp
#include <iostream>

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);
```

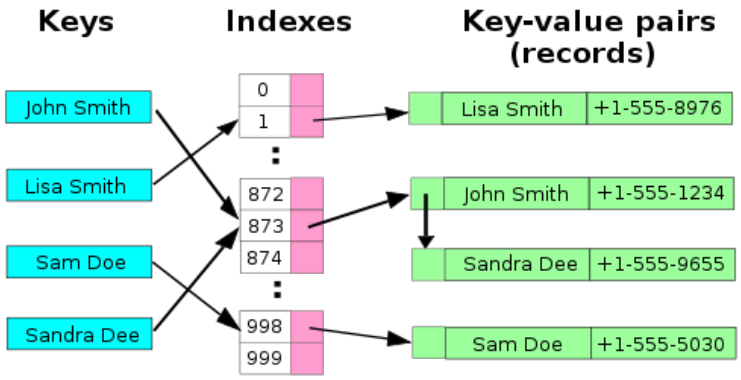
```
105   }
106   hashtable.print_all();
107 }
```

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
```

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

Κώδικας 7.10: 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
```

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.

```
#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;
```

```
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

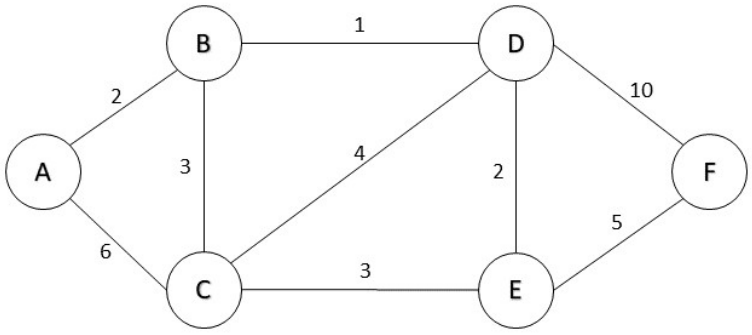
8.1 Εισαγωγή

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:

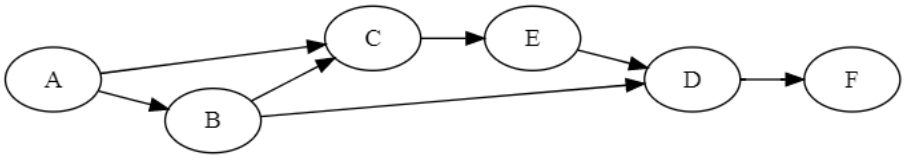
```
1 $ g++ -Wall -std=c++11 graph.cpp graph_ex1.cpp -o graph_ex1
2 $ ./graph_ex1
```

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 {
```

```
        NS.insert(kv.first);
        shortest_path_distances[kv.first] = {path, INT_MAX};
      }
    }

    while (!NS.empty()) {
      string v1 = S.back();
      for (pair<int, string> w_v : graph[v1]) {
        int weight = w_v.first;
        string v2 = w_v.second;
        if (NS.find(v2) != NS.end())
          if (shortest_path_distances[v1].cost + weight <
              shortest_path_distances[v2].cost) {
            shortest_path_distances[v2].path =
                shortest_path_distances[v1].path + " " + v2;
            shortest_path_distances[v2].cost =
                shortest_path_distances[v1].cost + weight;
          }
      }
      int min = INT_MAX;
      string pmin = "None";
      for (string v2 : NS) {
        if (shortest_path_distances[v2].cost < min) {
          min = shortest_path_distances[v2].cost;
          pmin = v2;
        }
      }
      // in case the graph is not connected
      if (pmin == "None")
        break;
      S.push_back(pmin);
      NS.erase(pmin);
    }
}
```

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

```
#include "dijkstra.hpp"
#include "graph.hpp"

using namespace std;

int main() {
  map<string, vector<pair<int, string>>> graph = read_data("graph1.txt");
  map<string, path_info> shortest_path_distances;
  string source = "A";
  compute_shortest_paths_to_all_vertices(graph, source,
                                         shortest_path_distances);
  for (auto p : shortest_path_distances) {
    cout << "Shortest path from vertex " << source << " to vertex " << p.first
         << " is {" << p.second.path << "} having length " << p.second.cost
         << endl;
  }
}
```

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:

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

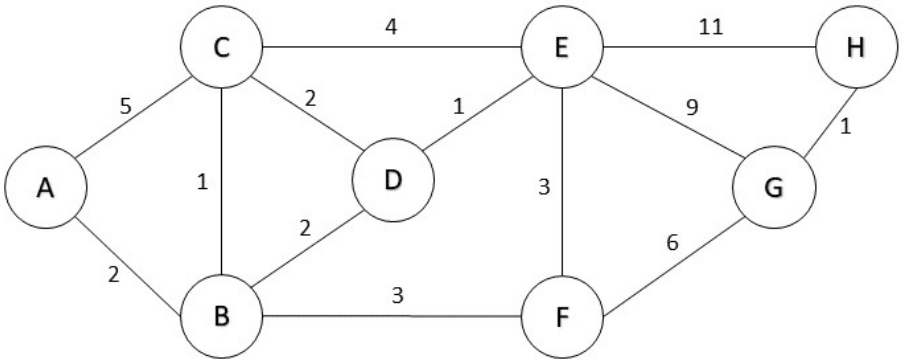
Η δε έξοδος που παράγεται είναι η ακόλουθη:

```
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/

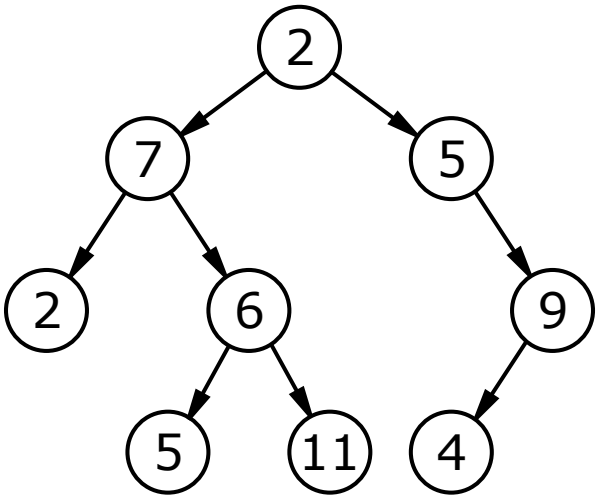
[10] Algorithm visualization, Dijkstra's shortest path, https://www.cs.usfca.edu/ galles/visualization/Dijkstra.html

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 (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.

Preorder DFS

Στη διάσχιση του δένδρου προ-διατακτικά (preorder) 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 preorder διάσχισης είναι η δημιουργία αντιγράφων ενός δένδρου καθώς και η λήψη της prefix μορφής ενός expression tree [3].

Inorder DFS

Στη διάσχιση του δένδρου ένδο-διατακτικά (inorder) 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 9.4, η inorder διάσχιση επιστρέφει τους κόμβους σε μη φθίνουσα σειρά.

Postorder DFS

Στη διάσχιση του δένδρου μέτα-διατακτικά (postorder) 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 postorder 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 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)

```
1  #include "binary_tree.hpp"
2  #include <iostream>
3  #include <vector>
4  using namespace std;
5
6  int main()
7  {
8      node *root_node = NULL;
9      vector<string> v = {"A", "B", "C", "D", "E", "F", "G", "H"};
10     for (string x : v)
11     {
12         if (root_node == NULL)
13             root_node = insert(root_node, x);
14         else
15             insert(root_node, x);
16     }
17
18     cout << "Level-order Traversal ";
19     print_level_order(root_node);
20     cout << endl;
21     cout << "Pre-order Traversal ";
22     print_pre_order(root_node);
23     cout << endl;
24     cout << "In-order Traversal ";
25     print_in_order(root_node);
```

```
26      cout << endl;
27      cout << "Post-order Traversal ";
28      print_post_order(root_node);
29      cout << endl;
30  }
```

Κώδικας 9.3: Δοκιμή των συναρτήσεων του δυαδικού δένδρου (lab09_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:
Η δε έξοδος που παράγεται είναι η ακόλουθη:

9.4 Δυαδικά δένδρα αναζήτησης

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

9.4.1 Υλοποίηση δυαδικού δένδρου αναζήτησης

```
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 *tree, int key)
6  {
7      cout << "Searching for key " << key << ": ";
8      node *p = search(tree, 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 *tree)
20 {
21     cout << "Minimum " << min(tree)->key << " Maximum " << max(tree)->key << endl;
22 }
23
24 int main()
25 {
26     node *tree = NULL;
27     vector<int> v = {10, 6, 5, 8, 14, 11, 18};
28     for (int x : v)
29     {
30         if (tree == NULL)
31         {
32             tree = insert(tree, x);
33         }
34         else
35         {
36             insert(tree, x);
37         }
38     }
39     cout << "In-order Traversal ";
40     print_in_order(tree);
41     cout << endl;
42     test_search(tree, 11);
43     test_search(tree, 13);
44     test_min_max(tree);
45     destroy(tree);
```

```
46 }
```

Κώδικας 9.6: Δοκιμή των συναρτήσεων του δυαδικού δένδρου αναζήτησης (lab09_ex1.cpp)

Η μεταγλώττιση και η εκτέλεση του κώδικα γίνεται με τις ακόλουθες εντολές:
Η δε έξοδος που παράγεται είναι η ακόλουθη:

9.5 Παραδείγματα

9.5.1 Παράδειγμα 1

Δεδομένου ενός δυαδικού δένδρου και μιας τιμής SUM ζητείται να βρεθεί το εαν υπάρχει διαδρομή από τη ρίζα του δένδρου μέχρι κάποιο κόμβο που να έχει άθροισμα κλειδιών ίσο με την τιμή SUM.

9.5.2 Παράδειγμα 2

9.6 Ασκήσεις

1.

2.

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/

Revision History

Revision	Date	Author(s)	Description
1.0	17.01.2018	Χρήστος Γκόγκος	Σημειώσεις για το εργαστήριο του μαθήματος Δομές Δεδομένων και Αλγόριθμοι
1.1	21.08.2018	Χρήστος Γκόγκος	Σημειώσεις για το εργαστήριο του μαθήματος Δομές Δεδομένων και Αλγόριθμοι - προσθήκη κεφαλαίου 9 (δένδρα)