Ministerul Educaţiei și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică şi Microelectronică

**RAPORT**

Lucrare de laborator Nr.8

*la Structuri de Date și Algoritmi*

Tema: Analiza empirică a algoritmilor de sortare și de căutare

A efectuat: st. gr. SI-212 Șeremet Alexandru

A verificat: lect. asist. Mititelu Vitalii

Chişinău 2022

**Scopul:**

Studierea posibilităților și mijloacelor limbajului C de programare a

algoritmilor de sortare și de căutare pentru tablouri unidimensionale și obținerea

deprinderilor de analiză empirică a algoritmilor

**Sarcina:**

Să se scrie un program în limbajul C pentru analiza empirică a algoritmului

propus (după variantă) cu crearea funcției de căutare sau sortare pentru tabloul

unidimensional de n elemente cu afișarea la ecran a următorului meniu de opțiuni:

1. Tabloul demonstrativ de n elemente (5<= n <=20).

2. Tabloul cu valori aleatorii (n=10000, n=100000, n=1000000).

3. Tabloul sortat crescător (n=10000, n=100000, n=1000000).

4. Tabloul sortat descrescător (n=10000, n=100000, n=1000000).

5. Ieșire din program.

Analiza empirică a algoritmului constă în: a) determinarea timpului de rulare,

numărului de comparații, numărului de schimbări (sau deplasări) de elemente pentru

tablouri cu 3 diferite seturi de valori și cu 3 diferite volume de elemente; b)

compararea și analiza rezultatelor obținute utilizând funcția creată și funcția din

biblioteca standard a limbajului C; c) tragerea concluziilor.

**Varianta 15:**

Sortare heap în ordine descendentă.

**Rezumat:**

**Testarea empirică**

Erorile software apar în timpul compilării, editării de legături, execuţiei şi interpretării rezultatelor.

Testarea empirică are caracter parţial şi se efectuează în etapele de analiză, proiectare, programare, integrare module.

Testarea empirică este atât proces de autoverificare cât şi proces global. Ea repezintă o serie de dezavantaje legate de volumul de muncă şi de imposibilitatea ameliorării peste o limită a software.

**1. Programul - o cutie neagră**

Se consideră un program P cu datele de intrare I şi rezultatele E, figura 4.1.



**Fig. 1. Intrări - Prelucrări - Ieşiri**

Testarea empirică se realizează în principal de către elaboratorii de programe şi mai apoi de către utilizatorii programelor.

Programul trebuie privit ca o cutie neagră. Din documentaţie, din modul de proiectare a interfeţelor rezultă structura datelor de intrare.

Cum se efectuează prelucrările, care sunt acestea, ce efecte secundare sunt generate, nu reprezintă un element esenţial din punctul de vedere al testării empirice.

Obiectivul testării empirice este acela de a pune în evidenţă că programul e bun, "merge" sau nu este bun şi în acest caz se vor identifica situaţiile în care programul nu oferă rezultatele cerute.

Testarea empirică este focalizată în trei puncte şi anume:

· la nivelul datelor de intrare, pentru a vedea dacă programul acceptă ca intrări datele care definesc problema; se identifică situaţiile de cerere de date mai mare decât oferta, cerere de date mai mică decât oferta şi în cazul cel mai bun egalitate între ceea ce solicită ca intrări programul şi ceea ce se oferă;

· la nivelul prelucrărilor, în sensul traversării paşilor algoritmului în mod complet, respectiv cu întrerupere într-un anumit punct al execuţiei sau în puncte diferite, cu găsirea unei legături între datele oferite şi punctul în care are loc întreruperea;

· la nivelul rezultatelor când se identifică rezultate incomplete structural, rezultate complete structural dar incorecte şi, respectiv, în situaţia în care rezultatele corespund calitativ fără a putea fi făcute menţiuni asupra corectitudinii efective a lor.

Testarea empirică are ca punct de plecare datele de intrare şi rezultatele aşteptate.

În cazul în care există corelaţii între rezultate, elemente invariante, testarea empirică va trebui să evidenţieze măsura în care sunt realizate aceste corelaţii.

În cazul privirii programului ca pe o cutie neagră, cu atât mai mult trebuie efectuat un studiu asupra laturilor calitative ale procesului de prelucrare, nefiind posibil accesul la componentele programului, la algoritmii de prelucrare.



**Fig. 2. Matrice rezultat a programului P**

De exemplu, se consideră programul P care efectuează o serie de prelucrări, conducând în final la obţinerea unei matrice cu o structură particulară, figura 4.2.

Pentru un set de date de intrare cu a=1, b=2, c=3, x=10, y=11, u=100 şi w=110, în cazul în care programul P afişează un tablou se verifică mai întâi:

· dacă există submatricea unitate de ordin 4 în colţul stânga sus al tabloului;

· dacă pe coloana a 5-a se află exclusiv valoarea lui a;

· dacă există submatricea nulă de ordin 3 în colţul din dreapta jos;

· dacă valoarea -1 este pe coloana a 2-a, liniile 5...8;

· dacă pe linia 1, coloanele 6 şi 7 se află valorile lui x, respectiv y, iar în coloana 8 este suma acestora;

· dacă pe liniile 2 şi 3, coloanele 6 şi 7 se găsesc valorile lui u, respectiv w, iar în coloana 8, linia 2 suma acestora, respectiv diferenţa lor pe coloana 8, linia 3;

· existenţa pe colona 4, linia 5 a valorii 1, iar pe liniile 6, 7 şi 8 a valorii 0;

· dacă pe coloana 1, liniile 5, 6, 7 şi 8 se află valorea lui a multiplicată de 4, 3, 2 respectiv o dată;

· existenţa pe coloana 3, liniile 5, 6, 7 şi 8 a rapoartelor 1/a, 2/a, 3/a, respectiv, 4/a.

Numeroase programe care efctuează prelucrări economice (evidenţe contabile, prognoze), pentru variabile precum venit naţional (VN), preţ (Pr), există numai valori strict pozitive. De aceea la apriţia într-un model de prognoză a estimărilor pentru VN sa Pr a unor termeni negativi sau nuli indică deja existenţa unor erori de prelucrare sau de concepţie la elaborarea modelului.

Caracterul empiric al testării este dat de parţialitatea, de lipsa de sistematicitate a procesului şi mai ales de ideea de încercare la întâmplare a comportamentului programului.

**Sortarea Heap**

În dorinţa de a îmbunătăţi algoritmul sortării pin selecţie (elementul minim/maxim din vector se va aşeza in locul primului/ultimului element al vectorului, algoritmul reluându-se pentru cele n-1 elemente rămase), se pot face următoarele observaţii:

- fiecare algoritm de căutare a minimului/maximului dintre n elemente, bazat pe compararea perechilor de elemente, trebuie să facă cel puţin n-1 comparaţii (altfel, ar rămâne elemente necomparate);

- observaţia anterioară se poate aplica doar primei etape de căutare, ulterior putând folosi informaţii deja dobândite (exemplu: pentru căutarea maximului din şirul 503, 87, 512, 61, 908, 170, 897, 275, putem împărţi şirul în patru subşiruri şi să determinăm maximul din fiecare subşir: 503, 87 →503, apoi din 512, 61→521, din 908, 170→908, şi 897, 275→897, apoi comparând doar elementele 503, 521, 908, 897 vom determina maximul total, 908. Pentru a determina următorul maxim este suficient să îl căutăm printre elementele rămase 503, 521, 897, alături de 170 din grupul maximului anterior, reducând o serie de comparaţii. Deci va fi 897, apoi alegem dintre 503, 521, 170, 275, etc. ).

Şirul valorilor anterioare poate fi organizat, după cum am observat şi în exemplu, într-un arbore binar complet, pe nivelurile superioare aşezând maximul a câte 2 valori ale nivelului anterior, rădăcină ajungând maximul şirului. În etapa următoare, se înlocuieşte maximul găsit anterior cu o valoare foarte mică, astfel încât printr-un algoritm asemănător să obţinem maximul valorilor rămase, etc.

Pentru orice n se poate construi arborele binar complet cu n noduri terminale.

Un vector x cu n componente poate fi interpretat ca un arbore binar. Mecanismul este următorul:

rădăcina este etichetată cu x[1];

nodul etichetat cu x[i] subordonează:

la stânga nodul etichetat cu x[2\*i], dacă 2\*i≤n;

la dreapta nodul etichetat cu x[2\*i+1], dacă 2\*i+1≤n.

Exemplu: Arborele binar complet asociat vectorului x: 5, 8, 2, 1, 9, 4, 3, 7, 6.



Definiţie: Un MinHeap este un arbore binar complet cu proprietăţile:

- informaţiile din noduri sunt valori dintr-o mulţime total ordonată, numite chei;

- pentru orice nod x[i], cheia memorată în x[i] este mai mică sau egală cu cheia oricăruia dintre fii.

-oricare ar fi i din mulţimea de indici{1, ... , [n/2]}, atunci x[i]<=x[2\*i] şi x[i]<=x[2\*i+1], dacă 2\*i+1<=n

Definiţie: Un MaxHeap este un arbore binar complet cu proprietăţile:

- informaţiile din noduri sunt valori dintr-o mulţime total ordonată, numite chei;

- pentru orice nod x[i], cheia memorată în x[i] este mai mare sau egală cu cheia oricăruia dintre fii.

-oricare ar fi i din mulţimea de indici{1, ... , [n/2]}, atunci x[i]>=x[2\*i] şi x[i]>=x[2\*i+1], dacă 2\*i+1<=n

Observaţie: În cazul arborelui reprezentat pentru un vector de tip MinHeap, orice nod subordonează noduri cu etichete mai mari, iar în cazul vectorului de tip MaxHeap, orice nod subordonează noduri cu etichete mai mici.

**Codul deplin al programului:**

**Fișierul algorithm.c:**

#include <stdio.h>

void swap(int \*p, int \*q)

{

int tmp;

tmp = \*p;

\*p = \*q;

\*q = tmp;

}

void heapify(int arr[], int n, int i, int \*swaps, int \*comps)

{

int smallest = i;

int l = 2 \* i + 1;

int r = 2 \* i + 2;

\*comps = \*comps + 1;

if (l < n && arr[l] < arr[smallest])

smallest = l;

\*comps = \*comps + 1;

if (r < n && arr[r] < arr[smallest])

smallest = r;

\*comps = \*comps + 1;

if (smallest != i)

{

\*swaps = \*swaps + 1;

swap(&arr[i], &arr[smallest]);

heapify(arr, n, smallest, swaps, comps);

}

}

void heapSort(int arr[], int n, int \*swaps, int \*comps)

{

for (int i = n / 2 - 1; i >= 0; i--)

heapify(arr, n, i, swaps, comps);

for (int i = n - 1; i >= 0; i--)

{

\*swaps = \*swaps + 1;

swap(&arr[0], &arr[i]);

heapify(arr, i, 0, swaps, comps);

}

}

**Fișierul algorithm.h:**

#pragma once

void heapSort(int arr[], int n, int \*swaps, int \*comps);

**Fișierul tests.c:**

#include <stdio.h>

#include <stdlib.h>

#include "algorithm.h"

void printArr(int arr[], int n)

{

for (int i = 0; i < n; ++i)

printf("%d ", arr[i]);

printf("\n");

}

int getRand(int lower, int upper)

{

return ((rand() % (upper - lower + 1)) + lower);

}

void getRandArr(int \*arr, int n)

{

for (int i = 0; i < n; i++)

arr[i] = rand();

}

void getAscArr(int \*arr, int n)

{

arr[0] = rand();

for (int i = 1; i < n; i++)

arr[i] = getRand(arr[i - 1], INT\_MAX);

}

void getDescArr(int \*arr, int n)

{

arr[0] = rand();

for (int i = 1; i < n; i++)

arr[i] = getRand(0, arr[i - 1]);

}

void example(int \*swaps, int \*comps)

{

int i = 5;

while (i <= 20)

{

int \*arr = (int \*)malloc(i \* sizeof(int));

printf("size %d:\n", i);

getRandArr(arr, i);

printf("original: ");

printArr(arr, i);

heapSort(arr, i, swaps, comps);

printf("sorted: ");

printArr(arr, i);

printf("\n");

i++;

}

}

void randArr(int n, int \*swaps, int \*comps)

{

int \*arr = (int \*)malloc(n \* sizeof(int));

getRandArr(arr, n);

heapSort(arr, n, swaps, comps);

}

void ascArr(int n, int \*swaps, int \*comps)

{

int \*arr = (int \*)malloc(n \* sizeof(int));

getAscArr(arr, n);

heapSort(arr, n, swaps, comps);

}

void descArr(int n, int \*swaps, int \*comps)

{

int \*arr = (int \*)malloc(n \* sizeof(int));

getDescArr(arr, n);

heapSort(arr, n, swaps, comps);

}

**Fișierul tests.h:**

#pragma once

void example(int \*swaps, int \*comps);

void randArr(int n, int \*swaps, int \*comps);

void ascArr(int n, int \*swaps, int \*comps);

void descArr(int n, int \*swaps, int \*comps);

**Fișierul user.c:**

#include <stdio.h>

#include <time.h>

#include "tests.h"

void show\_options()

{

printf("\nMeniu\n");

printf("1. Tabloul demonstrativ de n elemente (5 <= n <= 20)\n");

printf("2. Tabloul cu valori aleatorii (n = 10000, n = 100000, n = 1000000)\n");

printf("3. Tabloul sortat crescator (n = 10000, n = 100000, n = 1000000)\n");

printf("4. Tabloul sortat descrescator (n = 10000, n = 100000, n = 1000000)\n");

printf("0. Iesire din program.\n");

}

int handle\_options(int in)

{

int n = 10000;

int next = 0;

switch (in)

{

case 1:

int t = clock();

int swaps = 0;

int comps = 0;

example(&swaps, &comps);

t = clock() - t;

printf("time: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n", comps);

break;

case 2:

printf("continue to n = %d test? (1/0) ", n);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

randArr(n, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 10);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

randArr(n \* 10, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 100);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

randArr(n \* 100, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n", comps);

}

break;

case 3:

printf("continue to n = %d test? (1/0) ", n);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

ascArr(n, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 10);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

ascArr(n \* 10, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 100);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

ascArr(n \* 100, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n", comps);

}

break;

case 4:

printf("continue to n = %d test? (1/0) ", n);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

descArr(n, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 10);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

descArr(n \* 10, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n\n", comps);

}

printf("continue to n = %d test? (1/0) ", n \* 100);

if (scanf("%d", &next) && next == 1)

{

int t = clock();

int swaps = 0;

int comps = 0;

descArr(n \* 100, &swaps, &comps);

t = clock() - t;

printf("\ntime: %f", ((double)t) / CLOCKS\_PER\_SEC);

printf("\nswaps: %d", swaps);

printf("\ncomparisons: %d\n", comps);

}

break;

case 0:

return 1;

default:

printf("Valoarea %d nu exista in meniul de optiuni, introduceti o alta valoare.\n", in);

}

return (1);

}

int handle\_input()

{

int in;

do

{

show\_options();

printf("\nCe operatiune doriti sa efectuati?\n");

if (!scanf("%d", &in))

{

printf("\nvaloare invalida.\n");

return (0);

}

else if (!handle\_options(in))

{

return (0);

}

} while (in != 0);

return (1);

}

int main()

{

if (!handle\_input())

{

return (1);

}

return (0);

}

**Execuția programului:**







Procesul de aranjare a unui tablou cu valori aleatorii a avut cel mai mult timp, comparatii și schimbări, deoarece programul a trebuit să verifice și să schimbe cu locul mult mai multe valori decât într-unul sortat.



Tabloul sortat crescător a luat mai puține resurse decât cel aleatoriu, deoarece algoritmul dat de sortare transformă tabloul într-un heap, iar el fiind ordonat, toate elementele trec într-o ramura continuă a arborelui, fapt care face ordonarea mult mai ușoară.



Tabloul sortat descrescător a luat cele mai puține resurse, deoarece el nici nu avea nevoie de a fi sortat, așa că pe lângă eficiența de care a profitat tabloul sortat crescător, numărul de comparații și schimbări a fost redus considerabil și datorită faptului că ele toate au fost parte a transformării tabloului în heap, ci nu și a ordonării lui.

**Concluzii:**

* Întrucât algoritmul heapsort apelează de n−1 ori algoritmul combinare care are ordinul de complexitate O(log2n), iar cum algoritmul descris de funcţia heap are ordinul O(nlog2n) rezultă că heapsort are de asemenea ordinul O(nlog2n) .Pentru şiruri mici însă heapsort este ineficient, dar foarte eficient pentru cele mari.
* Testarea empirică este necesară în cazul produselor software vândute la cheie, fără a fi însoţite de o documentaţie clară sau sunt de clasă ce nu include o testare riguroasă.
* De asemenea, testarea empirică este specifică situaţiei în care utilizatorul rezolvă tot timpul probleme descrise cu aceleaşi structuri de date şi nu apar varibilităţi la nivelul dimensiunii, preciziei coeficienţilor grupării datelor. Testarea empirică nu e specifică dezvoltării de software, integrării şi reutilizării.

**Surse:**

* sortare.freewb.ro, Cursul de programare in C, Tema: HeapSort: <http://sortare.freewb.ro/HeapSort>
* testare-software.ase.ro, Cursul de programare, Tema: Testarea empirică

<http://www.testare-software.ase.ro/toc/testare_empirica.html>