

Proiect laborator structuri de date

Am ales să implementăm structura de date: **Red-Black Tree (Arbore Roșu-Negru)**. Acesta este un arbore binar de căutare care are un bit suplimentar pentru memorarea fiecărui nod: culoarea acestuia, care poate fi *roșu*(1) sau *negru*(0). Prin restrângerea modului în care se colorează nodurile pe orice drum de la rădăcină la o frunză, arborii roșu-negru garantează că nici un astfel de drum nu este mai lung decât dublul lungimii oricărui alt drum, deci că arborele este aproximativ echilibrat.

Proprietățile arborelui roșu-negru sunt:

1. Fiecare nod este fie roșu, fie negru.
2. Fiecare frunză este neagră.
3. Dacă un nod este roșu, atunci ambii fii ai săi sunt negri.
4. Fiecare drum simplu de la un nod la un descendent care este frunză conține același număr de noduri negre.

- **Motivația structurii de date folosite**

Am ales această structură de date, deoarece este aproximativ “echilibrată”, pentru a garanta că înălțimea este proporțională cu $O(\log n)$ chiar și în cazul cel mai nefavorabil. Fiind o structură de actualitate, este folosită în multe librării din industrie ca și fundație pentru mulțimi și dicționare. De asemenea, sunt utilizate pentru a implementa clasele `TreeSet` și `TreeMap` din Java Core API, la fel și pentru Standardul C++.

- **Analiza timp a programului**

Pentru cazul cel mai defavorabil al execuției operațiilor de bază pe mulțimi dinamice este corespunzătoare o complexitate în timp de $O(\log n)$.

	Media	Cel mai defavorabil caz
Spațiu	$O(n)$	$O(n)$
Căutare	$O(\log n)$	$O(\log n)$
Inserare	$O(\log n)$	$O(\log n)$
Ștergere	$O(\log n)$	$O(\log n)$
Predecesor	$O(\log n)$	$O(\log n)$
Succesor	$O(\log n)$	$O(\log n)$
Minim	$O(\log n)$	$O(\log n)$
Maxim	$O(\log n)$	$O(\log n)$
Cardinal	$O(1)$	$O(1)$
K-element	$O(n)$	$O(n)$

Demonstrație căutare:

Care este timpul de execuție al algoritmului de căutare?

Pentru a căuta un nod în arbore, se pleacă din rădăcină, iar la fiecare pas se decide direcția pentru nivelul următor din arbore. În cel mai rău caz, se va ajunge într-un nod-frunză, deci se va parcurge întreaga înălțime a arborelui.

Deoarece înălțimea unui arbore roșu-negru având n noduri este $O(\lg n)$, apelul algoritmului de căutare va consuma un timp $O(\lg n)$.

Demonstrație inserare:

Care este timpul de execuție al algoritmului de inserare?

Algoritmul de inserare se folosește de cel de căutare și de metoda fixRedRed, care la rândul ei se apelează recursiv și folosește rotații (atât rotateLeft, cât și rotateRight sunt realizate în $O(1)$). Deoarece metoda fixRedRed se apelează recursiv pentru bunic, în cel mai rău caz se parcurge întreaga înălțime din două în două nivele, astfel încât complexitatea maximă va fi $O(\lg n)$.

$T(n) = O(\text{alg. căutare}) + O(\text{metoda fixRedRed})$

$T(n) = O(\lg n) + O(\lg n) + O(1) \in O(\lg n) \Rightarrow T(n) = O(\lg n)$

Demonstrație ștergere:

Care este timpul de execuție al algoritmului de ștergere?

Algoritmul de ștergere se folosește de cel de căutare și de metoda fixDoubleBlack, care la rândul ei se apelează recursiv și folosește rotații (atât rotateLeft, cât și rotateRight sunt realizate în $O(1)$). Deoarece metoda

fixDoubleBlack se apelează recursiv pentru părinte, în cel mai rău caz se parcurge întreaga înălțime, astfel încât complexitatea maximă va fi $O(\lg n)$.

$T(n) = O(\text{alg. căutare}) + O(\text{metoda fixDoubleBlack})$

$T(n) = O(\lg n) + O(\lg n) + O(1) \in O(\lg n) \Rightarrow T(n) = O(\lg n)$

Demonstrație predecesor/succesor:

Care este timpul de execuție al algoritmului de aflare a predecesorului/succesorului?

Pentru a afla predecesorul/succesorul unui element, întâlnim două cazuri:

1. cazul în care elementul nu este frunză: pornim de la rădăcină cu căutarea până găsim elementul; apoi mergem pe fiul stang, respectiv fiul drept pentru succesor, apoi mergem pe fiul drept, respectiv stâng, cât de mult posibil, astfel încât la fiecare pas ne găsim la un nivel inferior parcurgând întreaga înălțime în cel mai defavorabil caz $\Rightarrow O(\lg n)$
2. cazul în care elementul este frunză: pornim de la rădăcină până găsim poziția frunzei, apoi ne mutăm în sus până găsim un strămoș mai mic, respectiv mai mare, astfel încât parcurgem înălțimea de maxim două ori $\Rightarrow O(\lg n)$

Demonstrație minim/maxim:

Care este timpul de execuție al algoritmului de găsire a minimului/maximului?

Pentru a afla minimul/maximul unei mulțimi: pornim din rădăcină și coborâm în stânga, respectiv în dreapta, până ajungem la frunză, astfel încât parcurgem în cel mai rău caz întreaga înălțime $\Rightarrow O(\lg n)$

Demonstrație cardinal:

Care este timpul de execuție al algoritmului de aflare a cardinalului mulțimii?

Întrucât la inserare incrementăm numărul de elemente al mulțimii și la ștergere decrementăm, metoda de returnare a cardinalului se execută în timp constant $\Rightarrow O(1)$

Demonstrație K-element:

Care este timpul de execuție al algoritmului de aflare al celui de-al k-lea element?

Pentru aflarea celui de-al k-lea element pornim din rădăcină, verificăm subarborele stâng nod cu nod, apoi subarborele drept până îl găsim $\Rightarrow O(n)$

- **Avantaje/dezavantaje ale structurii de date folosite**

Avantaje	Dezavantaje
<ul style="list-style-type: none"> • arborele roșu-negru este folositor când vrem să inserăm sau să ștergem frecvent • se echilibrează singur, deci aceste operații garantează un timp de $O(\log n)$ • are toate avantajele arborilor binari de căutare • spre deosebire de un arbore binar de căutare simplu care poate deveni neechilibrat și care poate rezulta într-o performanță de $O(n)$ pentru operații, arborele roșu-negru se echilibrează singur. 	<ul style="list-style-type: none"> • este relativ complicat de implementat din cauza operațiilor de inserare și ștergere care conferă numeroase cazuri posibile de încălcare a proprietăților acestui arbore care trebuie rezolvate • deoarece arborii B pot avea un număr variabil de copii, se preferă aceștia în locul arborelui roșu-negru pentru memorarea și indexarea unei cantități mai mare de informații

- **Descrierea modului de testare**

Pentru a descoperi limitele programului, am încercat să introducem cât mai multe valori posibile. Pentru început, am introdus 2^{25} numere în arbore, într-un timp de 13.145 secunde.

The screenshot shows a C++ IDE with a code editor on the left and a console window on the right. The code in the editor is as follows:

```

486         return root;
487
488         // altfel cautam in subarborile drept
489         return k_Node(root->right, k);
490     }
491
492     //returneaza valoarea celui de-al k-lea element
493     int Red_Black_Tree::k_element(int k)
494     {
495         return k_Node(root, k)->info;
496     }
497
498     int main() {
499         Red_Black_Tree tree;
500         for(long long int i = 1; i<=pow(2, 25)-1;i++)
501             tree.insertNode(i);
502         cout<<tree.cardinal()<<endl;
503         /*
504         Red_Black_Tree tree;

```

The console window on the right shows the output of the program:

```

C:\Users\User\Desktop\ffj.exe
33554431
Process returned 0 (0x0)   execution time : 13.145 s
Press any key to continue.

```

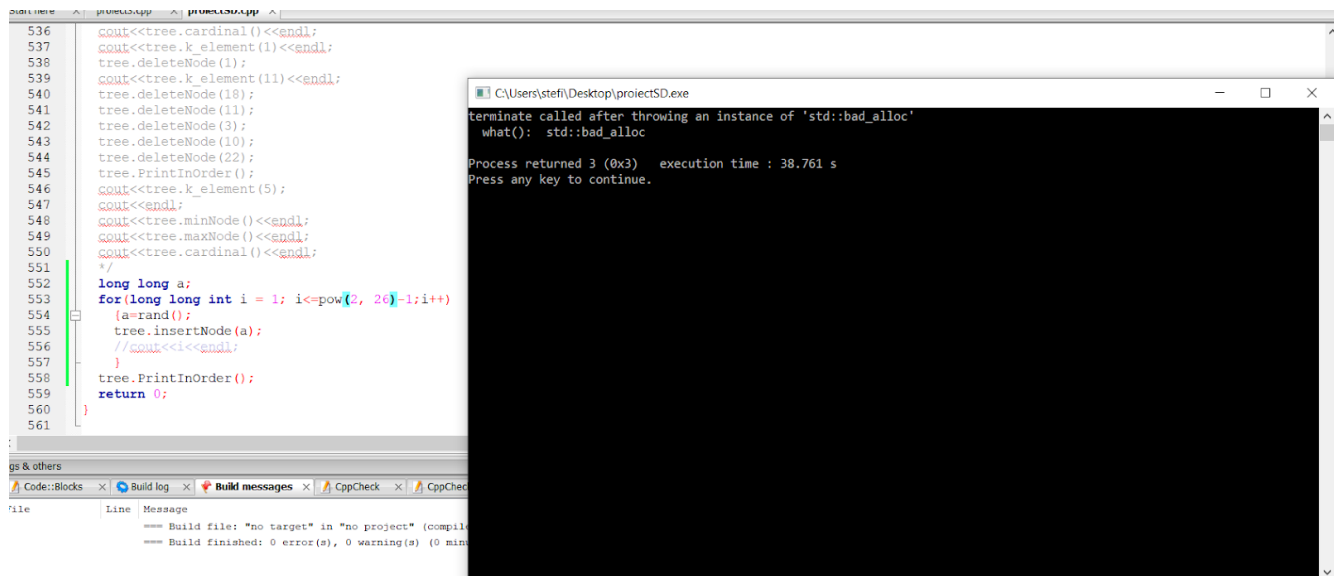
At the bottom of the IDE, there is a status bar showing the linking process and execution details:

```

Linking console executable: C:\Users\User\Desktop\ffj.exe
Process terminated with status 0 (0 minutes, 1 seconds)
0 errors, 0 warnings
Checking for existence: C:\Users\User\Desktop\ffj.exe
Executing: C:\work\CodeBlocks\cb_console_runner.exe "C:\Users\User\Desktop\ffj.exe" (in C:\Users\User\Desktop)

```

Apoi, am încercat să introducem 2^{26} numere:
-pe două din laptopuri, programul a dat următoarea eroare de memorie:

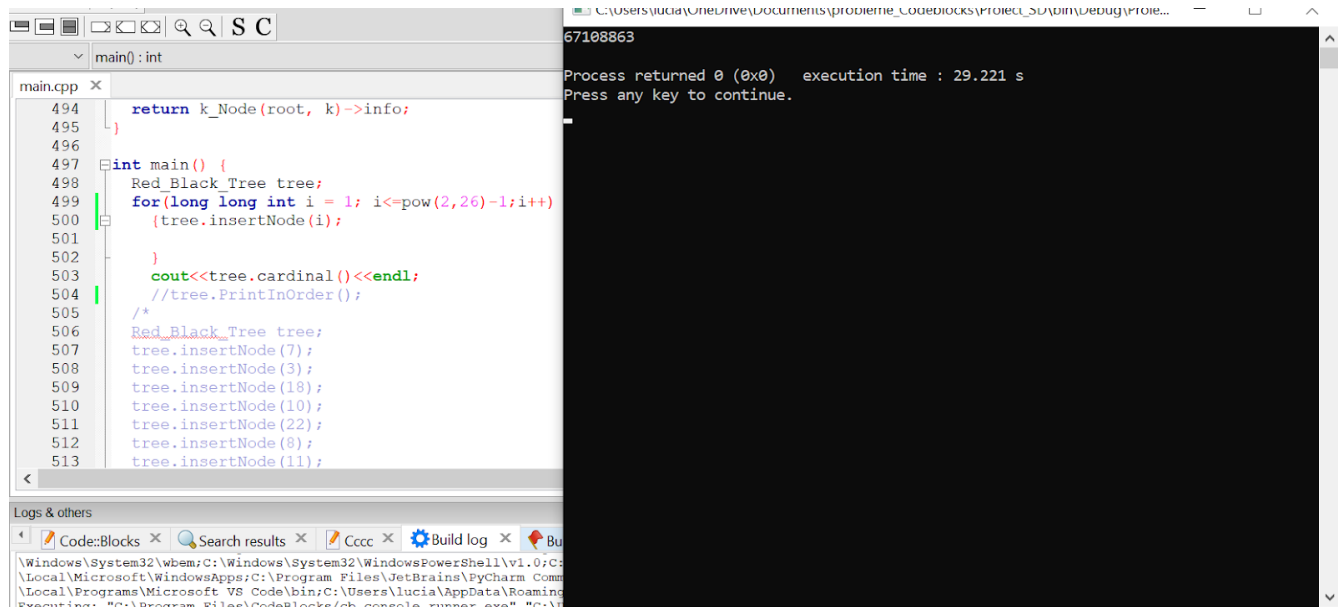


```
536 cout<<tree.cardinal()<<endl;
537 cout<<tree.k_element(1)<<endl;
538 tree.deleteNode(1);
539 cout<<tree.k_element(11)<<endl;
540 tree.deleteNode(18);
541 tree.deleteNode(11);
542 tree.deleteNode(3);
543 tree.deleteNode(10);
544 tree.deleteNode(22);
545 tree.PrintInOrder();
546 cout<<tree.k_element(5);
547 cout<<endl;
548 cout<<tree.minNode()<<endl;
549 cout<<tree.maxNode()<<endl;
550 cout<<tree.cardinal()<<endl;
551 */
552 long long a;
553 for(long long int i = 1; i<=pow(2, 26)-1;i++)
554 {a=rand();
555 tree.insertNode(a);
556 //cout<<i<<endl;
557 }
558 tree.PrintInOrder();
559 return 0;
560 }
561
```

```
terminate called after throwing an instance of 'std::bad_alloc'
what(): std::bad_alloc

Process returned 3 (0x3)   execution time : 38.761 s
Press any key to continue.
```

-pe un alt laptop, am reușit să introducem 2^{26} numere, într-un timp de 29.221 secunde.



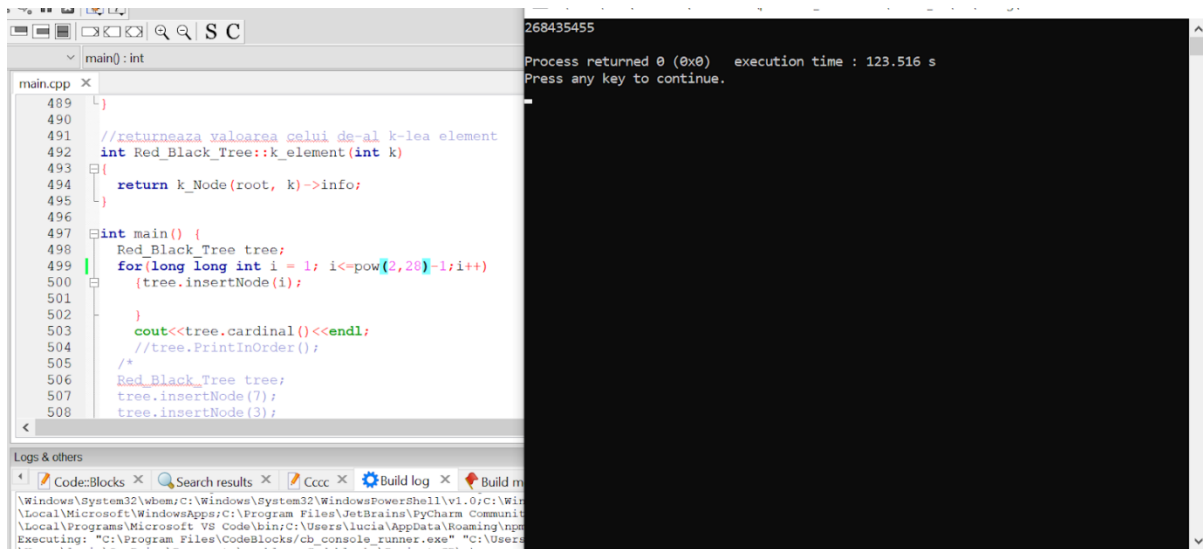
```
494 return k_Node(root, k)->info;
495 }
496
497 int main() {
498 Red_Black_Tree tree;
499 for(long long int i = 1; i<=pow(2,26)-1;i++)
500 {tree.insertNode(i);
501 }
502 cout<<tree.cardinal()<<endl;
503 //tree.PrintInOrder();
504
505 /*
506 Red_Black_Tree tree;
507 tree.insertNode(7);
508 tree.insertNode(3);
509 tree.insertNode(18);
510 tree.insertNode(10);
511 tree.insertNode(22);
512 tree.insertNode(8);
513 tree.insertNode(11);

```

```
67108863

Process returned 0 (0x0)   execution time : 29.221 s
Press any key to continue.
```

Pe acest laptop am reușit să introducem chiar și 2^{28} de numere, într-un timp de 123.516 secunde:



The screenshot shows a C++ program in a code editor and its execution output in a terminal window. The code defines a Red-Black Tree structure and inserts $2^{28} - 1$ elements. The execution output shows the process returned 0 and took 123.516 seconds to complete.

```
main.cpp x
489 }
490
491 //returneaza valoarea celui de-al k-lea element
492 int Red_Black_Tree::k_element(int k)
493 {
494     return k_Node(root, k)->info;
495 }
496
497 int main() {
498     Red_Black_Tree tree;
499     for(long long int i = 1; i<=pow(2,28)-1;i++)
500     {tree.insertNode(i);
501     }
502     cout<<tree.cardinal()<<endl;
503     //tree.PrintInOrder();
504     /*
505     Red_Black_Tree tree;
506     tree.insertNode(7);
507     tree.insertNode(3);
508     */
509 }

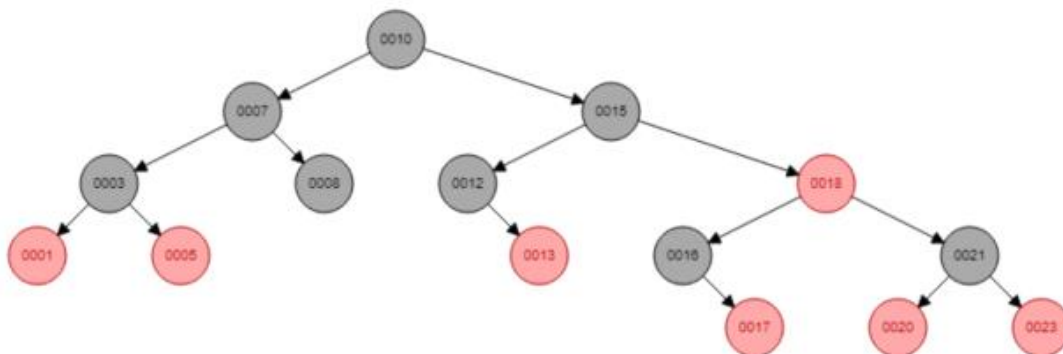
268435455
Process returned 0 (0x0)   execution time : 123.516 s
Press any key to continue.
```

După aceste teste, am ajuns la concluzia că programul nostru suportă cel mult 2^{28} numere pe device-urile pe care le deținem. O altă observație ar fi faptul că nu am reușit să citim cu ajutorul fișierelor de intrare o cantitate atât de mare de numere.

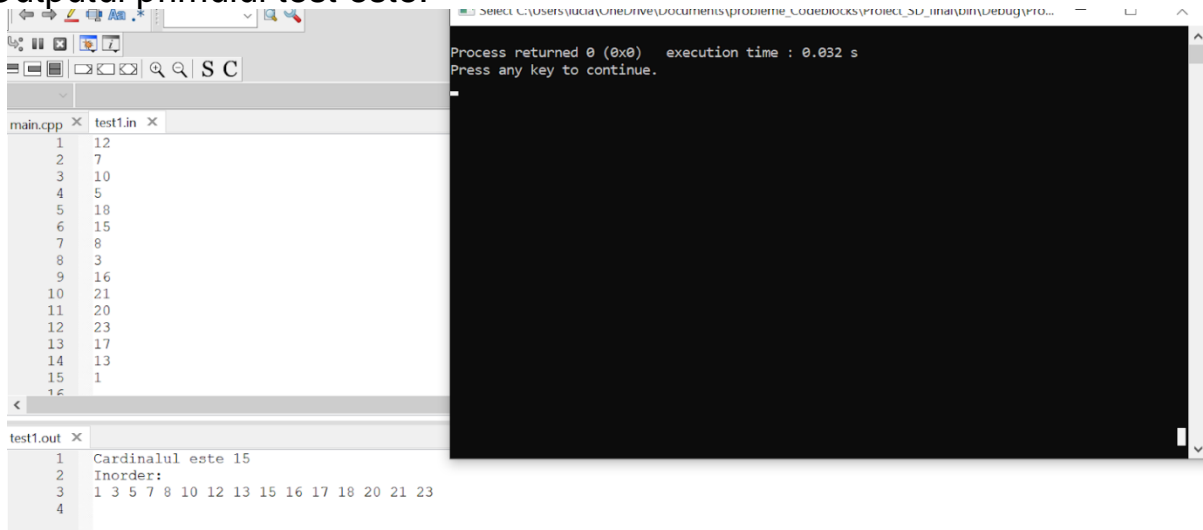
- Testul 1:

Am introdus 15 numere pentru a realiza operația de inserare și de aflare a cardinalului mulțimii. Toate aceste operații au fost realizate în 0.032 secunde.

Arborele introdus este:



Outputul primului test este:



The screenshot shows a code editor with two files: `main.cpp` and `test1.in`. `main.cpp` contains a list of 16 numbers. `test1.in` contains the input for the first test. The terminal output shows the execution results.

```
main.cpp x test1.in x
1 12
2 7
3 10
4 5
5 18
6 15
7 8
8 3
9 16
10 21
11 20
12 23
13 17
14 13
15 1
16

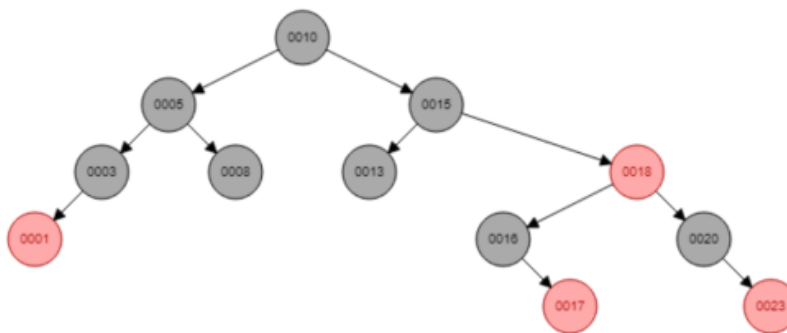
test1.out x
1 Cardinalul este 15
2 Inorder:
3 1 3 5 7 8 10 12 13 15 16 17 18 20 21 23
4

Process returned 0 (0x0) execution time : 0.032 s
Press any key to continue.
```

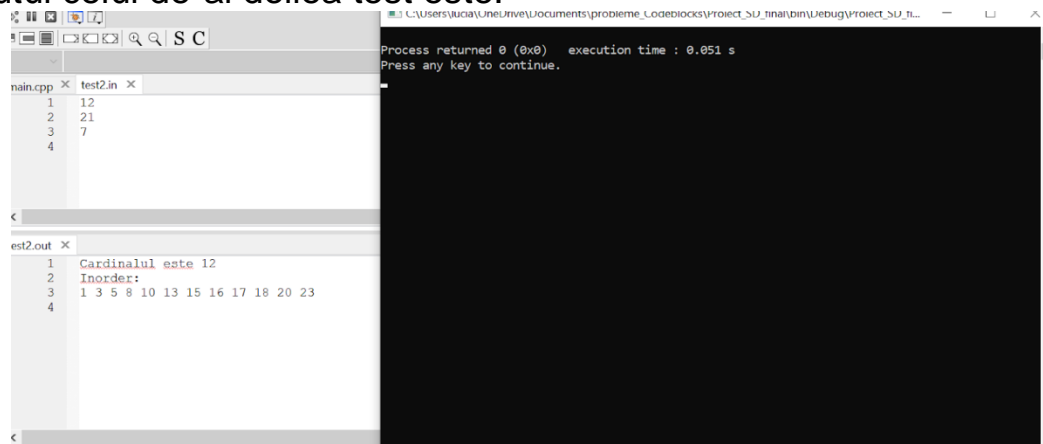
- Testul 2

Folosindu-ne de arborele introdus la primul test, am realizat operația de ștergere a 3 elemente și de aflare a cardinalului mulțimii (după ștergere), operații efectuate în 0.051 secunde.

Arborele după ștergerea celor trei elemente este:



Outputul celui de-al doilea test este:



The screenshot shows a code editor with two files: `main.cpp` and `test2.in`. `main.cpp` contains a list of 4 numbers. `test2.in` contains the input for the second test. The terminal output shows the execution results.

```
main.cpp x test2.in x
1 12
2 21
3 7
4

est2.out x
1 Cardinalul este 12
2 Inorder:
3 1 3 5 8 10 13 15 16 17 18 20 23
4

Process returned 0 (0x0) execution time : 0.051 s
Press any key to continue.
```

- Testul 3

Am introdus un nou arbore pentru a realiza operația de aflare a cardinalului, a minimului și a maximului din mulțime, toate acestea fiind efectuate în 0.049 secunde.

Cel de-al doilea arbore este:



Outputul celui de-al treilea test este:

```

Process returned 0 (0x0)   execution time : 0.049 s
Press any key to continue.

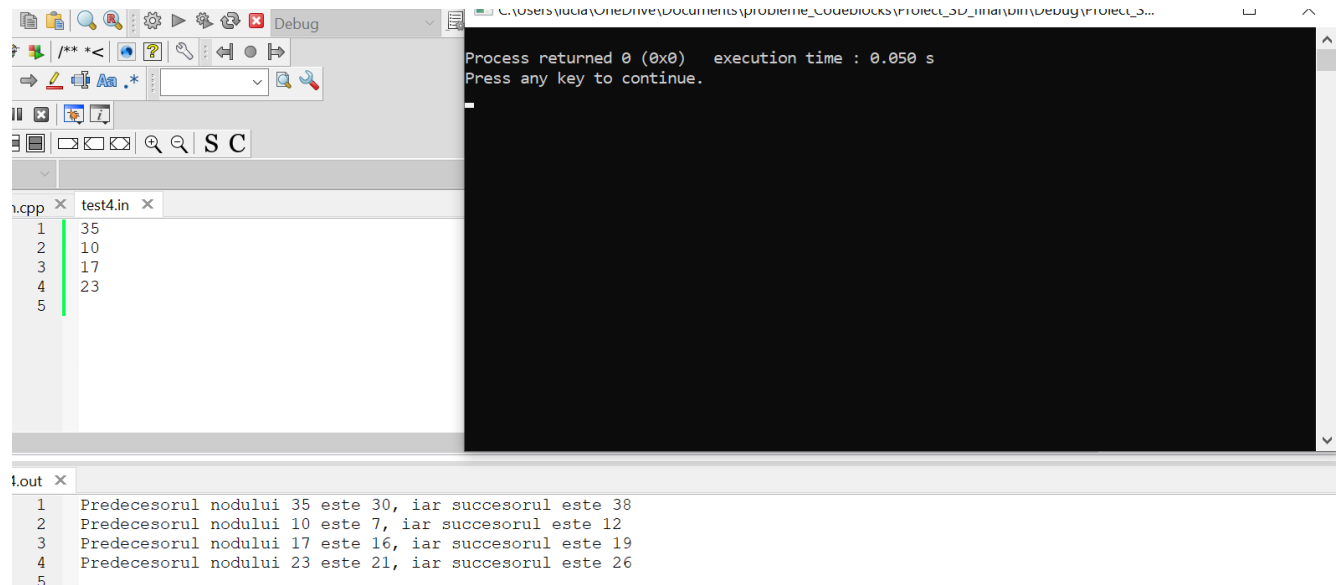
n.cpp x test3.in x
7 35
8 20
9 39
10 7
11 15
12 12
13 17
14 3
15 14
16 41
17 21
18 26
19 30

3.out x
1 Cardinalul este 19
2 Inorder:
3 3 7 10 12 14 15 16 17 19 20 21 23 26 28 30 35 38 39 41
4 Minimul este 3
5 Maximul este 41
  
```

- Testul 4

Folosindu-ne de arborele introdus la testul 3, am realizat operația de aflare a predecesorului și succesorului a 4 elemente din mulțime care s-au efectuat în 0.050 secunde.

Outputul celui de-al patrulea test este:



```
Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.
```

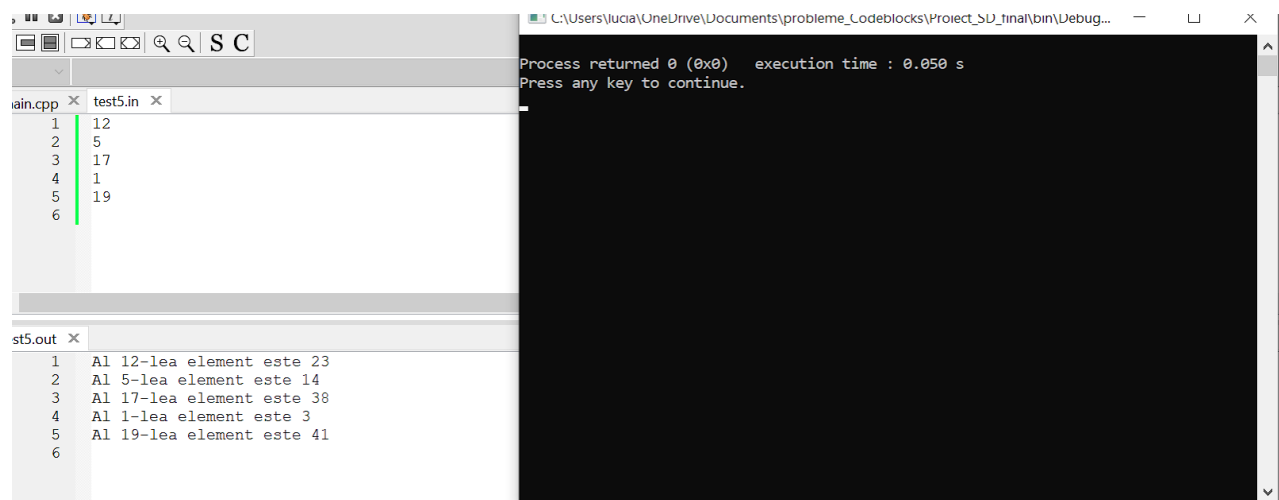
```
1 35
2 10
3 17
4 23
5
```

```
t.out
1 Predecesorul nodului 35 este 30, iar succesorul este 38
2 Predecesorul nodului 10 este 7, iar succesorul este 12
3 Predecesorul nodului 17 este 16, iar succesorul este 19
4 Predecesorul nodului 23 este 21, iar succesorul este 26
5
```

- Testul 5

Folosindu-ne de arborele introdus la testul 3, am realizat operația de aflare a k-lea element din mulțime în ordine crescătoare. Am introdus patru valori diferite pentru k, iar pentru fiecare valoare s-a afișat al k-lea element, operații care s-au efectuat în 0.050 secunde.

Outputul celui de-al cincilea test este:



```
Process returned 0 (0x0)   execution time : 0.050 s
Press any key to continue.
```

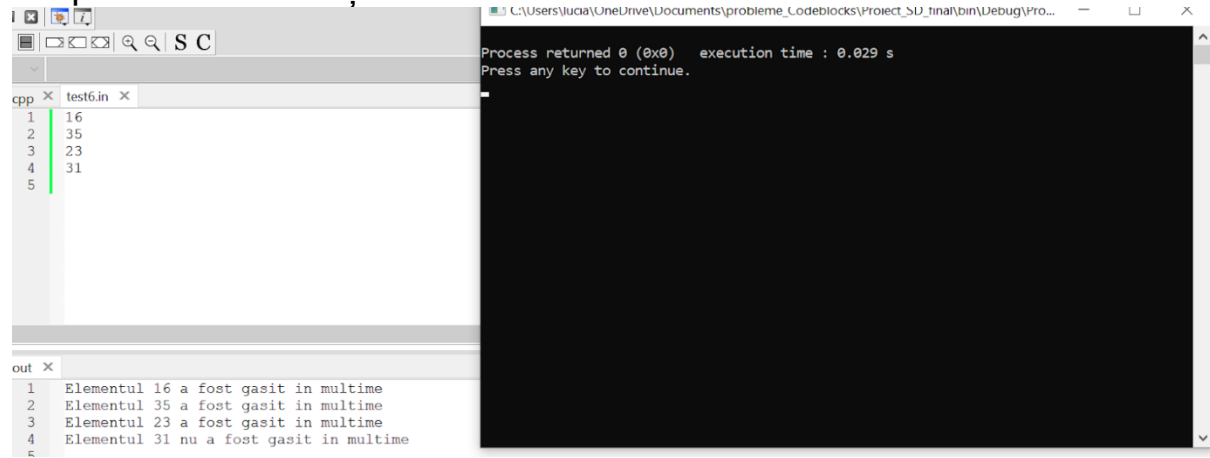
```
1 12
2 5
3 17
4 1
5 19
6
```

```
st5.out
1 Al 12-lea element este 23
2 Al 5-lea element este 14
3 Al 17-lea element este 38
4 Al 1-lea element este 3
5 Al 19-lea element este 41
6
```

- Testul 6

Folosindu-ne de arborele introdus la testul 3, am realizat operația de găsim a patru elemente introduse care s-a efectuat în 0.029 secunde.

Outputul celui de-al șaselea test este:



The screenshot shows the Code::Blocks IDE with a C++ file named 'test6.in' open. The code in the file is as follows:

```
1 16
2 35
3 23
4 31
5
```

The output window shows the results of the program's execution:

```
1 Elementul 16 a fost gasit in multime
2 Elementul 35 a fost gasit in multime
3 Elementul 23 a fost gasit in multime
4 Elementul 31 nu a fost gasit in multime
5
```

The console window shows the process return status and execution time:

```
Process returned 0 (0x0)   execution time : 0.029 s
Press any key to continue.
```

- **Sales Pitch**

Dorești să realizezi diferite operații pe o mulțime de numere într-un timp cât mai eficient?

Te-ai săturat să folosești numai stive și cozi și vrei să fii în pas cu tehnologia? Avem soluția perfectă pentru tine! Oferim un program care se bazează pe utilizarea arborelui roșu-negru pentru a realiza operațiile uzuale pe mulțimi : inserarea, ștergerea, căutarea unui element, aflarea minimului, maximumului, succesorul și predecesorul unui element , cardinalul mulțimii și al k-lea element în ordine crescătoare. Aproximativ toate aceste operații se efectuează într-un timp logaritmic, un timp mult mai eficient decât dimensiunea mulțimii (excepția se aplică pentru operația de cardinal, făcută în timp constant și pentru operația de aflare al celui de-al k-lea element în ordine crescătoare din mulțime, realizată în timp $O(n)$). Astfel, nu numai că economisești timp și reduci efortul de muncă folosind aceste operații, dar îmbunătățești și productivitatea afacerii tale. Dacă vrei să ne oferi o șansă, pentru mai multe informații contactează-ne la următoarea adresă de mail : RedBlackTree@proiect-sd.com.

Bibliografie:

- Introduction to Algorithms, Third Edition. Autori: Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
- <https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/>