Implementarea unui set in C++ folosind un arbore roșu-negru mărit (augmented red-black tree)

George-Mihai Radu

Analiza timp și spațiu a programului

Introducere

Pentru implementarea mulțimii considerând operațiile cerute s-a decis folosirea unui arbore roșu-negru mărit(augmented red-black tree).

Un arbore roșu-negru îndeplinește 5 proprietăți: [1]

1. Fiecare nod din arbore are o culoare, această culoare poate fi ori roșu ori negru, după cum este și denumirea arborelui.
2. Rădăcina arborelui are mereu culoarea negru.
3. Fiecare frunză NULL are culoarea negru.
4. Dacă un nod are culoarea roșu, atunci copii săi au culoarea negru.
5. Pentru orice nod ales arbitrar, toate căile simple de la el la frunzele descendente conțin același număr de noduri negre.

Arborele roșu-negru este un caz special de arbore binar de căutare. În tipul de arbore ales pentru fiecare nod valorile din subarborele stâng sunt mai mici strict decât valoarea nodului ales, iar valorile din subarborele drept sunt mai mari strict decât valoarea nodului ales. (Prop. 6)

Fiecare nod din arbore va conține culoarea sa(un enum class, poate fi RED sau BLACK), o variabilă pentru a reține o valoare(în cazul nostru, acest lucru poate fi modificat ulterior în funcție de schimbare cerințelor), 3 pointeri către alte noduri ale unui arbore roșu-negru(unul pentru părinte, unul pentru fiul stâng și unul pentru fiul drept), și în plus s-a adăugat încă o variabilă de tip ll int pentru a reține numărul de noduri pe care le conține subarborele cu rădăcina nodul respectiv(daca subarborele este format doar din noul respectiv, atunci valoarea size va fi 1). Este imediat evident faptul că un nod roșu-negru are complexitatea spațiu $\Theta(1)$ și reiese imediat că un arbore cu n elemente va avea complexitatea spațiu $\Theta(n)$, exact ce se dorea, o strcutură de date care ocupă mai mult spațiu nu ar fi benefică.

Înălțimea arborelui

Pentru a obține mai multe informații pentru următoarele operații pe mulțime ar fi convenabil să aflăm mai multe proprietăți ale arborelui ales. Una dintre ele ar fi cât de repede crește înălțimea arborelui la adăugarea mai multor elemente.

Ne este cunoscută următoarea lemă din cartea Introduction to Algorithms, 3rd Edition:

Lema 1 Un arbore roșu-negru cu n noduri interne are înălțimea cel mult $2\log(n+1)$. [1]

Din această lemă putem deduce imediat că putem implementa majoritatea operațiilor(min, max, search, predecesor, succesor) să ruleze în timp $O(\log n)$, întrucât arborele roșu-negru este un caz particular de arbore binar de căutare, structură pentru care aceste operații rulează în timp $O(h)$ unde h este înălțimea arborelui.

Metoda de inserare

Dacă nu avem noduri în arbore(cazul de bază) setăm noul nod ca fiind rădăcina. Altfel căutăm poziția nodului în care ar trebui să se afle x, complexitatea timp pentru acest pas este $O(\log n)$.

Nodul găsit anterior reprezintă tatăl noului nod cu valoarea x. Setăm tatăl lui x să fie nodul găsit anterior și vechii fii ai nodului găsit anterior vor deveni fii noului nod. Aceste operații au complexitatea timp $O(1)$.

După apelăm metoda $fixRedRedViolation$ care ar trebui să rezolve cazurile în care un nod roșu are un fiu roșu, metodă care doar face schimbări de pointeri și rotații $\Rightarrow$ complexitatea timp $O(1)$, metodele de rotații au aceeași complexitate $O(1)$. La orice inserție se fac cel mult 2 rotații, deci complexitatea timp pentru rotații și actualizarea mărimii nodurilor este $O(1)$.

În metoda de inserare trebuie să actualizăm și mărimea nodurilor, de la nodul găsit parcurgem arborele până la rădăcină incrementând mărimea fiecărui nod, ca să marcăm faptul că un nou nod a fost adăugat.

Complexitatea timp pentru această operație este $O(\log n)$ și spațiu $O(1)$.

Pentru a insera n elemente(i.e. a construi arborele) complexitatea timp este $O(n \log n)$.

Metoda de ștergere a unei valori din arbore

Căutăm poziția nodului care trebuie să fie șters. Dacă nu există niciun nod cu valoarea x, atunci nu avem ce să stergem din arbore și se încheie metoda. Căutarea are complexitatea timp $O(\log n)$.

De la nodul care trebuie șters până la rădăcină actualizăm mărimile nodurilor complexitatea timp fiind $O(\log n)$.

Următoarele operații constau în ștergerea legăturilor cu nodul care urmează să fie șters și rezolvarea problemei când un nod este dublu negru prin apelarea

metodei *fixDoubleBlack* care realizează doar rotații, schimbări de culori și actualizează mărimea nodului curent, al fratelui său și părintelui. Complexitatea timp este $O(1)$.

Complexitatea timp pentru ștergerea unei valori din arbore este $O(\log n)$ și spațiu $O(1)$.

Metoda de găsirea a minimului

Folosind prop. 6 este imediat evident că pentru a găsi minimul din arbore vom pleca din rădăcină și vom merge mereu în fiul stâng până ajungem la o frunză. Ultimul nod găsit va fi minimul din set conform proprietății 6.

Algoritmul constă în trecerea dintr-un nod în altul de la rădăcină până la o frunză(în cazul acesta cea mai din stânga), astfel complexitatea timp va fi $O(h)$ și folosind lema 1 rezultă imediat că metoda de găsire a minimului are complexitatea timp $O(\log n)$.

Întrucât metoda nu folosește apeluri recursive astfel încât să avem memorie folosită pe stack, ci s-a ales metoda iterativă, complexitatea spațiu este $O(1)$.

Metoda de găsirea a maximului

Folosind prop. 6 este imediat evident că pentru a găsi maximul din arbore vom pleca din rădăcină și vom merge mereu în fiul drept până ajungem la o frunză. Ultimul nod găsit va fi maximul din set conform proprietății 6.

Algoritmul constă în trecerea dintr-un nod în altul de la rădăcină până la o frunză(în cazul acesta cea mai din dreapta), astfel complexitatea timp va fi $O(h)$ și folosind lema 1 rezultă imediat că metoda de găsire a maximului are complexitatea timp $O(\log n)$.

Întrucât metoda nu folosește apeluri recursive astfel încât să avem memorie folosită pe stack, ci s-a ales metoda iterativă, complexitatea spațiu este $O(1)$.

Metoda de aflare a succesorului unei valori

Metoda de găsire a succesorului unei valori din mulțime primește o valoare x nu un nod, motiv pentru care mai întâi trebuie să căutăm nodul cu valoarea x, pentru a vedea dacă x există măcar în mulțime. Acest pas are complexitatea timp $O(\log n)$. Dacă x nu se află în mulțime atunci întoarcem -1.

Odată găsit nodul cu valoarea x, succesorul său se află în subarborele drept, subarborele drept conținând numai valori mai mari decât x, noi avem nevoie de cea mai mică valoare din subarbore, adică cea mai din stânga frunză din subarbore. Această căutare reprezintă metoda de găsire a minimului din arbore care știm că are complexitatea timp $O(\log n)$ în cel mai rău caz.

Dacă nodul cu valoarea x nu are un subarbore drept, atunci începem să căutăm prin părinții săi, de la părinte până la rădăcina arborelui până găsim un nod cu valoarea mai mare decât x. Motivul pentru care încercăm să cautăm

până la rădăcina arborelui este că părintele nodului ar putea fi și el fiul stâng al bunicului lui x și tot așa, caz în care trebuie să înaintăm spre x. Întrucât facem o căutare pe o cale simplă de la nodul x la rădăcină, această căutare este aproximativ identică cu cea de căutare a lui x în arbore, motiv pentru care complexitatea timp pentru acest pas este $O(\log n)$ în cel mai rău caz.

Dacă nici după acest pas nu am găsit succesorul atunci, x reprezintă maximul din arbore și evident nu are un succesor; întoarcem -1.

Complexitatea timp totală pentru această metodă este $O(\log n)+O(\log n) = O(\log n)$ și spațiu $O(1)$.

Metoda de aflare a predecesorului unei valori

Metoda de găsire a predecesorului unei valori din mulțime primește o valoare x nu un nod, motiv pentru care mai întâi trebuie să căutăm nodul cu valoarea x, pentru a vedea dacă x există măcar în mulțime. Acest pas are complexitatea timp $O(\log n)$. Dacă x nu se află în mulțime atunci întoarcem -1.

Odată găsit nodul cu valoarea x, predecesorul său se află în subarborele stâng, subarborele stâng conținând numai valori mai mici decât x, noi avem nevoie de cea mai mare valoare din subarbore, adică cea mai din dreapta frunză din subarbore. Această căutare reprezintă metodă de găsire a maximului din arbore care știm că are complexitatea timp $O(\log n)$ în cel mai rău caz.

Dacă nodul cu valoarea x nu are un subarbore stâng, atunci începem să căutăm prin părinții săi, de la părinte până la rădăcina arborelui până găsim un nod cu valoarea mai mică decât x. Motivul pentru care încercăm să cautăm până la rădăcina arborelui este că părintele nodului ar putea fi și el fiul drept al bunicului lui x și tot așa, caz în care trebuie să înaintăm spre x. Întrucât facem o căutare pe o cale simplă de la nodul x la rădăcină, această căutare este aproximativ identică cu cea de căutare a lui x în arbore, motiv pentru care complexitatea timp pentru acest pas este $O(\log n)$ în cel mai rău caz.

Dacă nici după acest pas nu am găsit predecesorul atunci, x reprezintă minimul din arbore și evident nu are un predecesor; întoarcem -1.

Complexitatea timp totală pentru această metodă este $O(\log n)+O(\log n) = O(\log n)$ și spațiu $O(1)$.

Metoda de găsire a celui de-al k-element din set în ordine crescătoare

Pentru găsirea celui de-al k-lea element în ordine crescătoare ne folosim de proprietatea arborelui mărit: fiecare nod x conține numărul de noduri din subarborele cu rădăcina x. Dacă subarborele este format numai din nodul x atunci numărul noduri este 1.

Algoritmul plecă cu un pointer din rădăcină și cât timp $k \neq curr\_rank$, unde *curr_rank* reprezintă indexul nodului curent în mulțimea sortată. Rankul curent

se calculează astfel $curr\_rank = rank\_fiu\_stâng + 1$. Dacă nodul curent nu are un fiu stâng atunci $rank\_fiu\_stâng = 0$.

Dacă $k < curr\_rank$ atunci al k-lea element se află în arborele stâng al nodului curent. În caz contrar, din k scădem rankul curent și mergem cu pointerul în subarborele drept.

La fiecare iterație actualizăm rankul curent și verificăm ca pointerul curent să nu fie *null* caz în care întoarcem -1.

Algoritmul de mai sus este similar cu o căutare în arbore, motiv pentru care va avea aceeași complexitate. Complexitatea timp este $O(h)$ folosind lema 1 obținem $O(\log n)$ și complexitatea spațiu este $O(1)$(nu avem apeluri recursive să ocupe spațiu pe stivă, metoda este iterativă).

Metoda de calcul a cardinalului mulțimii

În structura arborelui reținem un pointer către rădăcina sa și o variabilă de tip ll int, care reprezintă cardinalul mulțimii. În metodele de inserare și de ștergere ne ocupăm de incrementarea, respectiv decrementarea acesteii variabile, operație care are complexitate $O(1)$ neafectând complexitățile celor 2 operații. Astfel vom avea mereu calculat cardinalul mulțimii și când acesta este cerut nu mai trebuie să calculăm nimic, motiv pentru care complexitatea timp și spațiu a acestei metode este $O(1)$.

Metoda de verificare dacă o valoare se află în mulțime

Metoda primește o variabilă de tip ll int, care reprezintă valoarea x pentru care vrem să verificăm daca se află în mulțime.

Cum funcționează algoritmul:

- începem căutarea valorii din rădăcina arborelui și îl parcurgem până când ajungem la o frunză.

- cât timp valoarea nodului curent în care ne aflăm nu este x mergem mai departe într-unul din fii, în funcție de valoarea x, dacă x ¡ valoarea nodului curent atunci mergem în subarborele stâng, altfel mergem în subarborele drept.

- dacă am ajuns într-un nod a cărei valore este x atunci algoritmul se oprește și întoarce true(indicație că valoarea x se află în mulțime).

- dacă am ajuns într-o frunză a cărei valore nu este x atunci deducem că x nu se află în mulțime și întoarcem false.

După descrierea algoritmului de mai sus reiese imediat că are complexitatea $O(h)$, i.e. $O(\log n)$. Întrucât am ales metoda de implementare iterativă și nu avem apeluri recursive pe stivă, avem complexitatea spațiu $O(1)$.

Metoda de printare a arborelui

Metoda va printa elementele arborelui în ordine crescătoare. Metoda folosește traversarea în ordine (i.e. SRD). Indiferent de tipul de traversare a unui arbore, se va parcurge fiecare nod din arbore, motiv pentru care complexitatea timp este $O(n)$. Complexitatea spațiu este $O(\log n)$ dată de apelurile recursive de pe stivă.

Metoda de ștergere a arborelui

Pentru a șterge arborele în destrcutor vom apela o metodă privată recursivă, care va șterge arborele de jos în sus. Metoda de ștergere a arborelui face aceeasi parcurgere cu metoda de afișare SDR (fiu stâng după fiu drept și la final root). Parcurgerea va avea compexitatea spațiu $O(n) = O(\log n)$ (datorită apelurilor recursive de pe stivă pentru a ajunge la frunze). Complexitatea timp va fi aceeași cu cea de la printare, întrucât cele 2 metode sunt aproximativ similare în parcurgerea arborelui. Complexitate timp $O(n)$.

Motivația structurii de date folosite

O mulțime de cele mai multe ori este afișată în ordine crescătoare. De asemenea ar fi convenabil să avem o structură de date care să rețină elementele în ordine pentru a ne fi mai ușor să găsim minimul/maximul din mulțime, dar și succesorul/predecesorul unei valori din mulțime. O altă proprietate a unei mulțimi este accea că orice 2 elemente sunt distincte 2 câte 2.

O structură de date care să păstreze elementele în ordine este un arbore binar de cătuare(BST). Fiecare nod din BST are maxim 2 fii și relația dintre nod și cei doi fii este următoarea: valoare fiu stâng $<$ valoare nod $<$ valoare fiu drept. Arborele binar de căutare are complexitatea în medie pentru operațiile de inserare, ștergere și căutare de $O(\log n)$, cu toate acestea complexitatea depinde de cum au fost inserate elementele în arbore, putând ajunge la complexitate $O(n)$ (Cel mai rău caz ar fi acela în care elementele sunt introduse în ordine crescătoare sau descrescătoare).

Dorim să ne asigurăm că o să avem mereu complexitate $O(\log n)$, pentru această cerință ar trebui să avem mereu înălțimea arborelui $O(\log n)$ indiferent de input. Motiv pentru care încercăm să găsim un caz special de arbore binar de căutare; putem alege între un AVL și un RBT, ambele având complexitatea cea mai rea $O(\log n)$ pentru operațiile de inserare, ștergere și căutare. Dintre acești doi arbori, AVL-ul oferă căutări mai rapide întrucât sunt mai strict balansați (înălțimea unui RBT fiind $2\log(n+1)$). Un RBT oferă inserări și ștergeri mai rapide decât un AVL, din moment ce sunt realizate mai puține rotații dat fiind că are o balansare relativ mai relaxată.

Dat fiind că folosim o mulțime pe care dorim să adăugăm/ștergem elemente mai des vom alege să folosim un red-black tree. Astfel pentru toate operațiile cerute vom avea complexitatea timp $O(\log n)$. Singura excepție fiind metoda de a gasi al k-lea cel mai mic element.

Am trecut după de la un red-black tree la un augmented red-black tree, schimbare care nu afectează complexitatea timp și spațiu a operațiilor prezentate anterior. Avantajul pe care îl obținem este că putem implementa după metoda de a găsi al k-lea cel mai mic element în timp $O(\log n)$ și spațiu $O(\log h)$(apelurile de pe stivă) comparativ cu alternativa $O(n)$ timp și spațiu $O(\log h)$.

Avantaje/dezavantaje ale structurii de date folosite

Avantaje

- Elementele din structură sunt reținute în ordine crescătoare.
- Toate operațiile prezentate au complexitatea timp $O(\log n)$ și spațiu ori $O(1)$ ori $O(\log n)$. Complexitatea logaritmică este una din cele mai bune complexități pe care le puteam avea. O să arătăm în teste și ce diferența mare este de la $O(n)$ (timp liniar) la $O(\log n)$ (timp logaritmic).
- Întrucât ajustăm valoarea cardinalului la fiecare inserare și ștergere din arbore, metoda de a găsi câte elemente se află în structură va avea complexitate $O(1)$.
- Fiind o specializare a unui arbore binar de căutare are toate avantajele sale, ce are în plus este faptul că atunci când se inserează un element se auto-echilibrează, motiv pentru care avem cel mai rău caz la operațiile de căutare $O(\log n)$, evitând cazul în care am putea obține o listă dublu înlănțuită.
- Nu trebuie să verificăm toată structura de date (complexitate timp $O(n)$) pentru a păstra unicitatea elementelor din arbore, verificarea se va face în $O(\log n)$.

Dezavantaje

- Față de un AVL, RBT are înălțimea mai mare; deși complexitatea timp pentru operații este în ambele cazuri $O(h) = O(\log n)$, aceasta depinde de înălțimea arborelui, motiv pentru care operațiile dintr-un AVL sunt o idee mai rapide.
- Ar putea exista structuri de date care să aibă pentru unele operații complexități timp mai bune($O(\log \log n)$ sau chiar $O(1)$).

Modelul de testare

Pentru a testa strcutura de date pe diferite mărimi vom face 5 teste cu mărimea inputului n(=câte elemente va trebui să conțină RBT) + $10\%n$(=elemente adiționale pentru a demonstra că strcutura de date acceptă numai valori unice). n va lua valorile (15, 100, 10 000, 1 000 000, 10 000 000); am ales aceste valori pentru a obține teste atât pentru cei care vor dori să folosească strcutura de date pentru mulțimi relativ mici(15, 100, 10 000), dar și pentru mulțimi mai mari(1 000 000, 10 000 000).

Pentru fiecare n vom avea un cod sursă *testi.cpp*, un executabil și câte un fișier de intrare *testi.in* și unul de ieșire *testi.out* cu $i = \overline{1,5}$.

La începutul execuției fiecărui test se va crea un vector v de lungime n de valori distincte generate random(complexitatea va fi $O(n^2)$, ne așteptăm ca acest pas să îngreuneze cel mai mult execuția testelor). După acest pas mai alegem random alte $10\%n$ elemente din vectorul v pe care să le scriem în fișierul de intrare. Vom folosi acest vector v și pentru a testa dacă metodele din structura noastră RBT funcționează corect.

Formatul fișierului de intrare:

- pe prima linie se va specifica numărul de valori care urmează să fie citite ($n + 10\%n$).
- pe următoarea linie vor fi scrise valorile generate anterior separate de câte un spațiu, mai întai vom scrie toate cele n valori, după duplicatele $10\%n$.

După scrierea în fișierul de intrare, vom sorta vectorul v pentru a-l transforma într-un set sortat, astfel când îl vom folosi pentru a verifica metodele din RBT, orice operație va fi de acces a unui element $O(1)$, astfel amprenta pe care o va avea la timpul procesării unei metode nu va fi afectat atât de mult.

În fișierul de ieșire vom realiza teste pentru fiecare metodă, vom scrie rezultatul fiecărui apel al metodei, timpul de procesare și dacă răspunsul este corect. Metodele testate:

1. Citire datelor - inserarea a $n + 10\%n$ elemente în arbore
2. Scrierea mulțimii în fișierul de ieșire
3. Scrierea cardinalului mulțimii

4. Scrierea minimului

5. Scrierea maximului

6. Pentru $10\%n$ de elemente scriem succesorii, individual scriem timpul de procesare pentru fiecare căutare a succesorului unei valori și la final timpul total pentru cele $10\%n$ elemente.

7. Pentru $10\%n$ de elemente scriem predecesorii, individual scriem timpul de procesare pentru fiecare căutare a predecesorului unei valori și la final timpul total pentru cele $10\%n$ elemente.

8. Pentru $10\%n$ de indicii $k = \overline{1,n}$ scriem al k-lea cel mai mic element din mulțime, individual scriem timpul de procesare pentru fiecare apel al metodei și la final timpul total pentru cele $10\%n$ apeluri ale metodei.

9. Pentru $10\%n$ de indicii $k = \overline{1,n}$ verificăm dacă $v[k]$ se află în mulțime, individual scriem timpul de procesare pentru fiecare apel al metodei și la final timpul total pentru cele $10\%n$ apeluri ale metodei. Toate apelurile ar trebui să întoarcă *true*.

10. Pentru $5\%n$ de valori $x = \overline{min - 1000, min - 1}$ verificăm dacă se află în mulțime, individual scriem timpul de procesare pentru fiecare apel al metodei și la final timpul total pentru cele $5\%n$ apeluri ale metodei. Toate apelurile ar trebui să întoarcă *false*.

11. Pentru $5\%n$ de valori $x = \overline{max + 1, max + 1000}$ verificăm dacă se află în mulțime, individual scriem timpul de procesare pentru fiecare apel al metodei și la final timpul total pentru cele $5\%n$ apeluri ale metodei. Toate apelurile ar trebui să întoarcă *false*.

12. După alegem iar k indici, analog cu pasurile anterioare și ștergem din arbore valorile de pe pozițiile k din vectorul v și le salvăm într-un alt vector *to_delete*, după ștergere verificăm dacă valoarile din *to_delete* se mai găsesc în arbore.

13. La final scriem timpul total în care au fost executate operațiile de mai sus, de la print la delete.

Pentru a verifica faptul că toate testele au fost executate corect, în fișierul de ieșire căutăm string-ul *not*, acesta marchează pentru orice operație testată dacă a fost executată corect sau nu. Căutarea s-a făcut cu 2 metode: am dat search in fișier la cuvântul *not*, a doua am folosit comanda UNIX *grep* pentru o dublă verificare. Din testele realizate de noi toate operațiile funcționează corect și trimit răspunsul așteptat.

Cardinalul îl vom compara cu valoarea n stabilită inițial în program, n reprezentând numărul de elemente pe care dorim să facem testele, pentru a testa că structura de date păstrează numai numere unice, o să avem adăugate în fișierul de intrare și duplicate la final.

Restul de verificări se fac cu ajutorul vectorului sortat v, vector din care putem accesa valori în timp $O(1)$. Vectorul va fi folosit pentru a compara rezultatele dintr-o metodă naivă care știm sigur că merge(exemple: accesarea celui de-al k-lea element, succesorul unei valori aflate pe poziția k este evident într-un vector sortat elementul aflat pe poziția k+1, analog predecesorul va fi pe poziția k-1). Minimul în vectorul sortat va fi elementul de pe prima poziție, iar maximul va fi ultimul element din vector.

Pentru a testa metoda *isIn* verificăm mai întâi dacă $10\%n$ din elementele din v se află și în arbore. Toate aceste verificări trebuie să întoarcă *true*. După verificăm dacă alte $10\%n$ numere din afara intervalului $[min, max]$ se află în arbore. Toate aceste verificări trebuie să întoarcă *false*.

Pentru metoda *delete* vom construi un vector *to_delete* în care vom salva ce numere vom șterge pentru verificare. După ce ștergem un element verificăm cu metoda de găsire *isIn* dacă elementul se află sau nu în mulțime(folosim metoda *isIn* întrucât știm deja de la pasul anterior că funcționează corect).

Pentru fiecare test vom calcula timpul total pentru testele pe care le facem pe strcutura de date creată, iar pentru testele 4 și 5 la care avem un n foarte mare vom testa și timpul de generare a fișierului de intrare, întrucât din acestea obținem date foarte importante. Pentru măsurarea timpului vom folosi și comanda UNIX *time* pentru o dublă verificare.

Primele teste fiind pe mulțimi mici le-am grupat într-o singură secțiune.

```
$ time ./test1
./test1 0.00s user 0.00s system 89% cpu 0.002 total

$ time ./test2
./test2 0.00s user 0.00s system 86% cpu 0.003 total

$ time ./test3
./test3 0.13s user 0.00s system 99% cpu 0.139 total
```

Testul 4:

```
$ time ./test4
finished generating the input file
1385.3
finished sorting the array for testing
1385.61
./test4 1404.89s user 0.38s system 99% cpu 23:25.91 total
```

Testul 5:

```
$ time ./test5
finished generating the input file
241526
finished sorting the array for testing
4.00661
./test5 140461.70s user 5.56s system 57% cpu 67:32:50.99 total
```

Se poate observa deja că generarea setului pentru teste într-un vector este foarte costisitoare cu cât eșalonul de testare este mai mare, în cazul nostru deja de la un eșalon de 1 000 000 de numere testele durează foarte mult, iar de la 10 000 000 nu mai este viabilă metoda(67h doar pentru construirea setului în timp ce citirea cu arborele nostru a durat 29.3621s).

Pentru citirea mulțimii avem următoarele rezultate (complexitatea timp este $O(n \log n)$):

Test1: 3.3554e−05s

Test2: 8.2154e−05s

Test3: 0.00476954s

Test4: 2.22973s

Test5: 29.3621s

Timpii de citire a mulțumilor este rezonabil considerând eșaloanele alese.

Pentru afișarea mulțimii avem următoarele rezultate (complexitatea timp este $O(n)$, mai bine de atât nu se poate):

Test1: 4.001e−06s

Test2: 1.1633e−05s

Test3: 0.000747131s

Test4: 0.236384s

Test5: 2.30707s

Timpii de afișare a mulțumilor este rezonabil considerând eșaloanele alese.

Operațiile de complexitate timp $O(\log n)$ au următorii timpi(întrucât toate metodele noastre au aceeași complexitate vom alege o medie a acestora pentru a nu obține același rezultat pentru toate testele):

Test1: 8.67e−07s −− 1.11e−06s

Test2: 3.81e−07s −− 6.34e−07s

Test3: 3.39e−07s −− 5.518e−06s

Test4: 9.94e−07s −− 3.198e−06s

Test5: 3.677e−06s −− 1.393e−05s

Timpii de rulare pentru restul operațiilor sunt foarte buni maximul fiind de aproximativ 0.01393ms.

De asemenea în timpul testării am luat o decizie de îmbunătățire a metodei de găsirea a celui de-al k-lea element, întrucât prima implementare avea complexitatea timp $O(n)$ (fișierele i/o test4_n.in test4_n.out). Am schimbat metoda prin mărirea informațiilor din nodurile arborelui, ajungând astfel la complexitatea timp $O(\log n)$. Astfel în teste am ajuns de la timpii de procesare:

0.0044208s −− 0.0601068s per apel metoda
6896.45s total

la

1.259e−06s −− 2.117e−06s per apel metoda
0.276759s total

îmbunătățire semnificativă dacă luăm în calcul faptul că aceste operații vor fi apelate de mai multe ori.

Timpii totali pentru realizarea tuturor testelor pe arbore(luăm în calcul de la citirea arborelui până la final):

Test1: 0.000113469s

Test2: 0.000222856s

Test3: 0.0136167s

Test4: 20.163s

Test5: 1638.96s

Probleme cu metoda de testare

Întrucât am decis să fim riguroși și să testăm fiecare metodă dacă funcționează corect, evident trebuie să salvăm datele în altă structură de date mai simplă de care vom fi siguri că va returna răspunsurile corecte(în cazul nostru un simplu vector alocat dinamic pe care urmează să îl sortăm folosind merge-sort pentru a obține un boost de eficiență) întrucât nu avem cum să testăm manual structura de date pentru numere foarte mari (câteva mii). Când construim vectorul inițial costul timp este foarte mare, lucru care îngreunează programul înainte să apucăm să testăm structura noastră de date.

Am vrut să testăm și pe mulțimi de numere mai mari(100 000 000), dar generarea mulțimii are complexitatea timp $O(n^2)$, lucru care în practică pentru numere mari durează foarte mult, motiv pentru care după 4 zile de rulat am oprit programul forțat:

$ ./test5 214039.27s user 8.62s system 61% cpu 96:27:10.64 total

Sales pitch

Soluția oferită de noi este atât performantă, bine structurată, și documentată încât oferă o accesibilitate cât mai ușoară pentru cei care nu au lucrat la acest proiect pentru a înțelege cu explicații în comentarii la fiecare pas important pentru a trece de dificultățile pe care le poate provoca alocarea dinamică de memorie în C++ și alte detalii tehnice.

Soluția noastră este viabilă atât pentru cei care vor să o folosească pentru mulțimi mici cât și pentru mulțimi extrem de mari, testele pentru o mulțime de 10 000 000 finalizându-se în timp 1638.96s, datorită structurii de arbore roșu-negru la care toate operațiile au complexitatea logaritmică și în plus pentru metoda de a găsi cel de-al k-lea element în ordine crescătoare am adus modifcări pentru a oferi o performanță cât mai mare și un timp de procesare cât mai mic, renunțând la implementarea în timp liniar mai ușoară de realizat care pentru un test cu $n = 1\ 000\ 000$ numere, în care am apelat metoda și am primit un răspuns în timp de 6896.45s, alegând să trecem la un arbore roșu-negru mărit pentru a trece această metodă la timp logaritmic oferind un timp de răspuns pentru același test de 0.289568s. Astfel vă oferim numai performanță fără compromisuri.

Bibliography

[1] Thomas H. Cormen Charles E. Leiserson Ronald L. Rivest Clifford Stein. *Introduction to Algorithms*. 3rd ed. MIT Press, 2009. ISBN: 0-262-03384-4.

Tot din același carte am preluat toate ideile pentru implementare și detaliile despre complexitățile timp și spațiu a arborelui roșu-negru și a arborelui mărit.