ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერების დეპარტამენტი

გიორგი შველიძე

სამაგისტრო ნაშრომი: წითელ-შავი ხის დაბალანსების ალგორითმები

ნაშრომი შესრულებულია კომპიუტერული მეცნიერებათა მაგისტრის
აკადემიური ხარისხის მოსაპოვებლად.

ხელმძღვანელი: პროფესორი კობა გელაშვილი
თანახელმძღვანელი: ნიკოლოზ გრძელიძე

თბილისი
2015

Ivane Javakhishvili Tbilisi State University
Faculty of Exact and Natural Sciences

Department of Computer Science

Giorgi Shvelidze

Master's Thesis: Red-black tree balancing algorithms

Thesis proposed to achieve the degree of master in computer science

Supervisors: Prof. Koba Gelashvili
Nikoloz Grdzelidze

Tbilisi
2015

ანოტაცია

ძებნის ორობითი ხეები მნიშვნელოვან როლს ასრულებენ კომპიუტერულ სისტემებში. მასთან დაკავშირებული ალგორითმების მცირედმა გაუმჯობესებამ შესაძლოა დადებითი გავლენა მოახდინოს სისტემის სწრაფქმედებაზე. წითელ-შავი ხეები გამოყენებულია ოპერაციული სისტემების ბირთვის რეალიზაციაში. ასევე ფართოდ გამოიყენება მაღალი დონის პროგრამირების ენებში, მათ სტანდარტულ ბიბლიოთეკებში მონაცემთა სტრუქტურები რეალიზებულია წითელ-შავი ხის გამოყენებით.

სამაგისტრო ნაშრომის კვლევის საგანს წარმოადგენს წითელ-შავი ხის სწრაფი დაბალანსება, რათა ავაჩქაროთ მასში მონაცემების ძებნა. ნაშრომში განხილულია დაბალანსების ორი ალგორითმი და შემუშავებულია მიდგომა, რომლის გამოყენების უპირატესობა დასაბუთებული იქნება ექსპერიმენტულად. სამაგისტრო ნაშრომი წარმოადგენს ნიკოლოზ გრძელიძის ნაშრომის გაგრძელებას. ჩვენი მიზანია მონაცემთა სტრუქტურის შექმნა, რომელიც სწრაფად აიგება და მონაცემთა საკმაოდ დიდი რაოდენობისთვის ეფექტური ძებნის საშუალებას მოგვცემს.

Annotation

Binary search trees are important data structures used in computer systems. Any improvement of algorithms or approaches related to binary search trees can result advantages in performance. Many data structures used in high-level programming languages or in computational geometry can be based on red–black trees, and the Completely Fair Scheduler used in current Linux kernels uses red–black trees.
The master's thesis is about red-black tree balancing algorithms. Tests results will show advantages of our solution. We continue Nikoloz Grdzelidze's project. Goal of the project is to create data structure. For sufficiently big data it will be able to provide insertion and search fast.

სარჩევი

შესავალი

ნაშრომის შინაარსი. სამაგისტრო ნაშრომი წარმოადგენს ნიკოლოზ გრძელიძის სამაგისტრო ნაშრომის (თსუ, 2013 წ.) გაგრძელებას და განვითარებას. პროექტის მიზანია მონაცემთა სტრუქტურის შექმნა, რომელიც დიდი მოცულობის მონაცემებისთვის მოგვცემს ძებნის ორობითი ხის სწრაფად აგების და შემდეგ მასში ეფექტური ძებნის განხორციელების საშუალებას. ამისთვის გამოვიყენებთ წითელ-შავი ხისა (Red-black tree) და დალაგებული ძებნის ორობითი ხის (Order statistic tree) კომბინაციას. საწყისი მონაცემებისგან აიგება წითელ-შავი ხე, რომელსაც დავაბალანსებთ, რათა ავაჩქაროთ ძებნა. დაბალანსებისთვის შემუშავებულია სპეციალური ალგორითმი.

ნაშრომის მოტივაცია შემდეგია: თანამედროვე ენების მრავალი კონტეინერი (იხ. ქვემოთ) და ფაილური სისტემის მართვის უტილიტა იყენებს წითელ-შავი ხის საფუძველზე შექმნილ მონაცემთა სტრუქტურებს. სწრაფი გადაბალანსების ალგორითმის არსებობის შემთხვევაში, კომპიუტერის ან პროგრამის პასიურ მდგომარეობაში ყოფნის პირობებში შესაძლებელია მონაცემთა სტრუქტურების სწრაფი გადახალისება, რაც გააქტიურების შემდეგ გაზრდის კომპიუტერის სწრაფქმედებას.

მოკლედ იმის შესახებ, თუ რატომ წითელ-შავი ხე. იგი წარმოადგენს თვით-ბალანსირებად ძებნის ორობით ხეს, რომლისთვისაც რეალიზებულია კვანძის ჩასმა, წაშლა და ძებნა გარანტირებულ O(logN) დროში. არსებობს უფრო მკაცრად დაბალანსებული ცნობილი მონაცემთა სტრუქტურაც, AVL ხე, რომელსაც უფრო სწრაფი ძებნა აქვს რეალიზებული, მაგრამ ეს ხდება შედარებით ნელი ჩასმის ხარჯზე (უფრო მეტ დროს ხარჯავს ხის დასაბალანსებლად, შედეგად მისი სიმაღლე ნაკლებია, ვიდრე RBT-ის და ძებნა სწრაფია). ამიტომ ჩასმის ინტენსიური ოპერაციებისთვის უმჯობესია RBT-ის გამოყენება.

თავდაპირველი იდეა მდგომარეობდა ჰიბრიდული სტრუქტურის შექმნაში, რომელიც, ჩამატება-წაშლის გრძელი სერიის წინ გარდაიქმნებოდა წითელ-შავ ხედ, ხოლო შედარებით სტაბილურ, ძებნების პერიოდში - AVL ხედ.

ეს იდეა ეფექტურად იქნა რეალიზებული ნიკოლოზ გრძელიძის სამაგისტრო ნაშრომში, რომელშიც ნაჩვენები იქნა, რომ ერთი სახის ხის ტრანსფორმაცია მეორედ ხდება წრფივ დროში, ორივე ხე იმპლემენტირდება ერთი კლასის ფარგლებში, და ამ კლასის კოდის მოცულობა უმნიშვნელოდ განსხვავდება წითელ-შავი ხის კლასისგან. ამავე დროს, ამ ნაშრომის შედეგად გაირკვა, რომ

არსებული წითელ-შავი ხის დაბალანსება, თუ ეს ასევე სწრაფად განხორციელდება, უფრო ეფექტური გზაა იგივე მიზნის მისაღწევად.

წითელ-შავ და AVL ხე-ს შორის ჩვენც პირველს მივანიჭეთ უპირატესობა, რადგან,

როგორც წესი, წითელ-შავი ხე უფრო ფართოდ გამოიყენება, რადგან სხვებთან შედარებით თანაბრად კარგი მეთოდები აქვს ჩასმა/წაშლა/ძებნა, წინა და მომდევნო კვანძი და ზოგიერთი სხვა ალგორითმი. იგი გამოყენებულია თანამედროვე პროგრამული ენების იმპლემენტაციებში და ოპერაციულ სისტემებში:

- Java: java.util.TreeMap , java.util.TreeSet .
- C++ STL: set, map, multimap, multiset.
- Linux kernel: completely fair scheduler, linux/rbtree.h

აღსანიშნავია, რომ ძებნის ორობითი ხის დაბალანსების პირველი ალგორითმი [იხ. 2] წარმოადგენდა ორობითი ხის AVL ხედ გარდაქმნის ალგორითმის განზოგადებას.

სამაგისტრო ნაშრომის მომზადების პროცესში დამუშავებულ იქნა ბალანსირების ცნობილი ალგორითმები: **DSW** და სეჯვიკის. პირველი მათგანის გამოყენება ჩვენს შემთხვევაში უპრობლემოდ შეიძლება, რადგან იგი არ ითხოვს ხის კვანძის რაიმე გასაკუთრებულ სტრუქტურას. სეჯვიკის ალგორითმისთვის არსებითია, რომ ორობითი ხის კვანძის სტრუქტურა გაძლიერდეს ველით, რომელიც შეინახავს ქვეხეების ელემენტების რაოდენობებს. შესაბამისად, ზოგად ორობით ხეებზე სეჯვიკის ალგორითმის გამოყენება ვერ მოხერხდება. ე.წ. დალაგებულ ხეებზე, რომლებსაც ეს ველი აქვს, სეჯვიკის ალგორითმი მუშაობს n log *n* დროში (უარესი შემთხვევა), ხოლო **DSW** - წრფივ დროში.

წითელ-შავი ხისთვის სეჯვიკის ალგორითმის გამოყენების სქემა ჩვენს მიერ შემუშავდა. იგი მდგომარეობს ფერის ველის გამოყენებაში (დროებით) size ველად, რასაც წრფივი დრო სჭირდება და რაც საშუალებას გვაძლევს გამოვიყენოთ დაბალანსების ალგორითმი. შემდეგ, ნაშრომში შემუშავებული მეთოდით, ყოველი კვანძის size ველს შევცვლით გარკვეული ფერით, ისე რომ წითელ-შავი ხის თვისება დაცული დარჩეს. ამ პროცედურას ვუწოდებთ გადაღებვას. სამივე, size ველზე, გადასვლა, დაბალანსება და გადაღებვა, სრულდება წრფივ დროში და ძალიან სწრაფად. ნაშრომში მოყვანილია ამ მიდგომის უპირატესობის დამადასტურებელი მონაცემები.

დამატებით, ჩვენს მიერ შემუშავებულია სეჯვიკის ალგორითმის მოდიფიკაცია, რომელიც სრულად დაბალანსებულ ხეს იძლევა. ეს ალგორითმი ისე განსხვავდება სეჯვიკის ალგორითმისგან, როგორც **DSW** ალგორითმი დეის ალგორითმისგან.

ნაშრომის სტრუქტურა. ნაშრომი შედგება შესავალის, ხუთი თავის, დასკვნის, დანართისა და ლიტერატურის ნუსხისგან. სხვადასხვა თავების შინაარსი, მათი სიმარტივის გამო, კარგად გამოხატავს მის შინაარსს და ცალკე განხილვას არ საჭიროებს.

ძებნის ორობითი ხე

ძებნის ორობითი ხე (binary search tree) არის მონაცემთა სტრუქტურა, რომელიც საშუალებას იძლევა ხის სიმაღლის ლოგარითმის პროპორციულ დროში მოხდეს ახალი წვეროს ჩამატება, გასაღების მიხედვით წვეროს მოძებნა, წვეროს წაშლა, მინიმალური და მაქსიმალური გასაღების მქონე წვეროს მოძებნა. წვეროების რაოდენობის პროპორციულ დროში ხდება ხის წვეროების შემოვლა რამდენიმე გავრცელებული მეთოდით. ძებნის ორობითი ხე აკმაყოფილებს შემდეგ თვისებებს:

- კვანძის მარცხენა ქვეხე შეიცავს მხოლოდ იმ კვანძებს, რომელთა გასაღების მნიშვნილობა ნაკლებია აღნიშნული კვანძის გასაღების მნიშვნელობაზე.
- კვანძის მარჯვენა ქვეხე შეიცავს მხოლოდ იმ კვანძებს, რომელთა გასაღების მნიშვნილობა მეტია აღნიშნული კვანძის გასაღების მნიშვნელობაზე.
- კვანძის მარცხენა და მარჯვენა ქვეხეები წარმოადგენენ ძებნის ორობით ხეებს.
- კვანძები არ უნდა მეორდებოდნენ.

ხეში x მისამართის მქონე წვეროს, ჩვეულებრივ, მოეთხოვება რომ განსაზღვრულია შემდეგი ფუნქციები:

- left(x)- x-ის მარცხენა შვილის მისამართი
- right(x) - x-ის მარჯვენა შვილის მისამართი
- parent(x) - x-ის მშობლის მისამართი, რაც არის NULL ფესვისთვის
- key(x)- გასაღების მნიშვნელობა

ჩვეულებრივი, არაბალანსირებული ხის შემთხვევაში, დალაგებულად შემოსული მონაცემების ჩასმის სისწრაფე მცირდება O(lg n)-დან O(n)-მდე. ერთ-ერთ გადაწყვეტას წარმოადგენს დაბალანსებული ხეები, რომლებიც იყენებენ ბალანსირების წესებს, რათა ხის სიმაღლე შემოსაზღვრონ O(lg n) ფუნქციით. ყველაზე პოპულარული დაბალანსებული ხეები არის AVL და წითელ-შავი ხე.

წითელ-შავი ხე

წითელ-შავი ხე წარმოადგენს ძებნის ორობით ხეს, რომლის თითოეული წვერო შეიცავს დამატებით ინფორმაციას - ფერს. გარდა ძებნის ორობითი ხის ძირითადი თვისებებისა, იგი აკმაყოფილებს შემდეგ თვისებებს:

- ყოველი კვანძი ან შავია ან წითელი.
- ფესვი არის შავი.
- ყოველი ფოთოლი შავია;
- თუ კვანძი წითელია, მისი ორივე შვილი შავია;
- ფესვიდან მებისმიერ ფოთლამდე გზაში შავი კვანძების რაოდენობა ტოლია.

იმისთვის რომ წითელ-შავმა ხემ ძირითადი თვისებები შეინარჩუნოს, არსებობს მეთოდები, რომლებიც აღადგენენ ამ თვისებებს ხეში ცვლილებების შემთხვევაში.

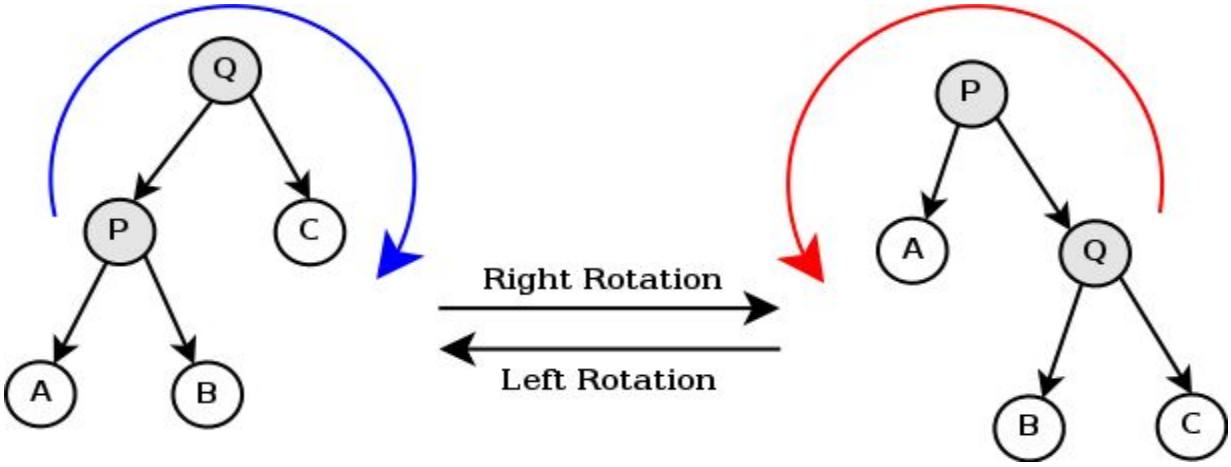
მობრუნება

გვაქვს ორი სახის მობრუნება, მობრუნება მარცხნივ და მობრუნება მარჯვნივ, ორივე მეთოდის მუშაობის დროითი შეფასება არის O(1). მარცხნივ მობრუნების ფსევდო-კოდი:

```
LEFT_ROTATE (T, x)
        y = x.right
        x.right = y.left
        if (y.left! = T.nil)
                y.left.p = x
        y.p = x.p
        if (x.p =  = T.nil)
                T.root = y
        elseif (x =  = x.p.left)
                x.p.left = y
        else x.p.right = y

        y.left = x
        x.p = y
```

მარჯვნივ მობრუნების კოდი სიმეტრიულია მარცხნივ მობრუნებისა.



კვანძის ჩასმა

ფუნქცია პარამეტრებად იღებს კვანძის სტრუქტურას და ჩასასმელი კვანძის მისამართს. კვანძის ჩასმა იწყება ხის ფესვიდან, რათა განისაზღვროს შემოსული კვანძის ადგილი ხეში, გამომდინარე BST-ის ძირითადი თვისებიდან. შემოსული კვანძის ფერი ჩამატების შემდეგ წითელია. ჩასმის ფუნქციის ფსევდო-კოდი:

```
RB-INSERT (T, z)
        y = T.nil
        x = T.root
        while (x != T.nil) {
                y = x
                if (z.key < x.key)
                        x = x.left
                else x = x.right
        }
        z.p = y
        if (y == T.nil)
                T.root = z
        elseif (z.key < y.key)
                y.left = z
        else y.right = z
        z.left = T.nil
        z.right = T.nil
        z.color = RED
        RB-INSERT-FIXUP(T,z)
```

ჩასმის შემდეგ მოსალოდნელია, რომ დაირღვეს წითელ-შავი ხის თვისებები, ამიტომ ყოველი ჩამატების, ან წაშლის შემდეგ შემდეგ ვიძახებთ Fixup მეთოდებს, ხის წითელ-შავ თვისების აღსადგენად. საინტერესოა, რომ წითელ-შავი ხის და AVL ხის ჩამატება-წაშლის ალგორითმები მხოლოდ ამ დამატებითი Fixup მეთოდებით განსხვავდება, რომლებსაც არ განვიხილავთ, რადგან უშუალოდ არ უკავშირდება ჩვენს თემას.

DSW ალგორითმი

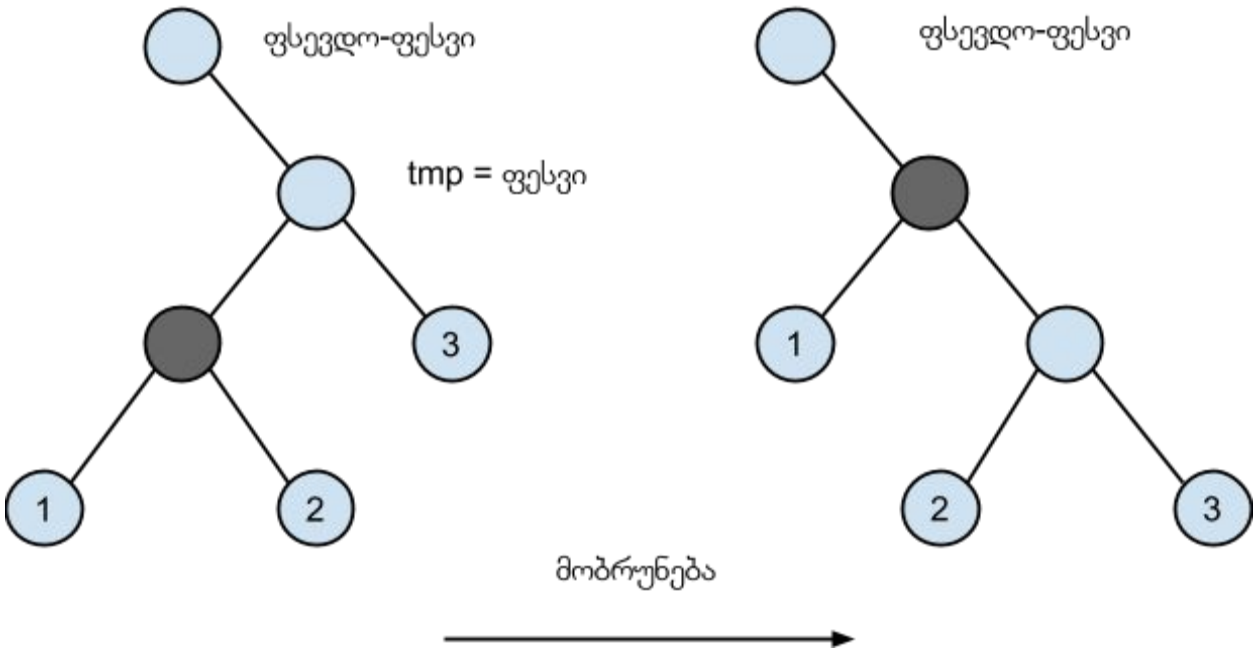
1976 წელს Colin Day-მ შეიუმუშავა ორობითი ხის დინამიკური დაბალანსების ალგორითმი, რომელიც თავიდან ირიდებს ოპერაციების გარკვეულ რაოდენობას, რასაც AVL გვთავაზობს. (თუმცა ის მაინც რჩება ორობითი ხის ბალანსირების დინამიკურ ალგორითმად). Day-ის ალგორითმის პირველი რეალიზაცია შესრულებული იყო fortran-ზე. რეკურსიის და კლასების უქონლობამ თავისებურად გაართულა იგი. ალგორითმი შედგება ორი ფაზისგან:

1. ორობითი ხისაგან გადაგვარებული ხის აგება - ნებისმიერ კვანძს ჰყავს მხოლოდ და მხოლოდ მარჯვენა შვილი;
2. გადაგვარებული ხისაგან დაბალანსებული ხის აგება - compression მეთოდის გამოყენებით.

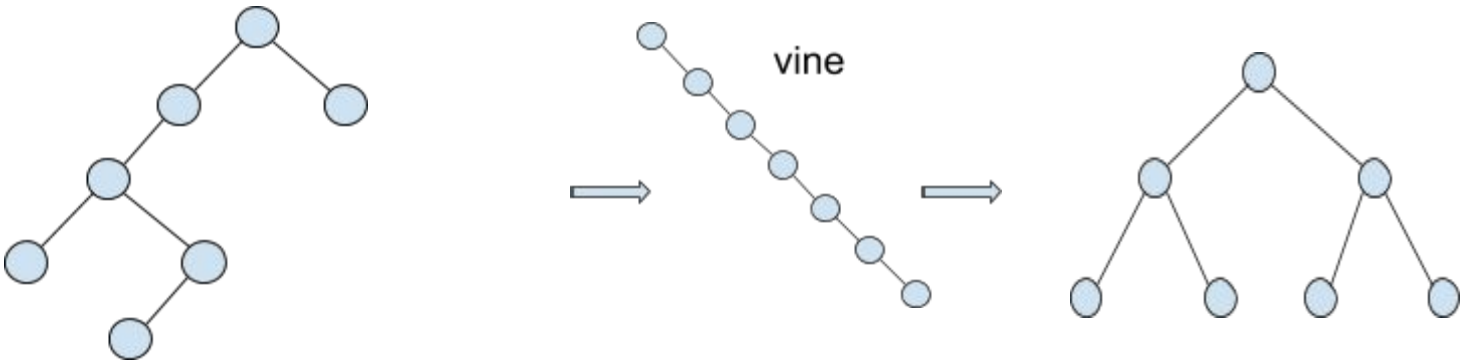
გადაგვარებული ხის მისაღებად Day იყენებდა ბმულ სიას, რომელიც მოითხოვდა დამატებით მეხსიერებას. პროგრამირების ენების განვითარებასთან ერთად გაჩნდა ახალი შესაძლებლობები და ალგორითმმაც ცვლილება განიცადა.

10 წლის შემდეგ Quentin F. Stout-მა და Bette L. Warren-მა ცვლილებები შიტანეს Colin Day-ის ალგორითმში. ძირეული ცვლილება განიცადა ალგორითმის პირველმა ფაზამ და მცირედი - მეორემ. Stout-მა და Warren-მა შენიშნეს, რომ პირველ ფაზაში, გადაგვარებული ხის აგება შესაძლებელია ტრიალების გამოყენებით, კერძოდ, მარჯვნივ მობრუნებების ოპერაციით. მათ იმპლემენტაციაში შემოიღეს ფსევდო-ფესვი, რაც ალგორითმს ხდის უფრო მარტივს და თავიდან გვარიდებს განშტოების ზედმეტ ოპერაციებს. მარჯვნივ ტრიალი ხორციელდება მანამ, სანამ კვანძის მარცხენა შვილი არ გახდება null. ამის შემდეგ ალოგირთმი ჩადის ერთი დონით დაბლა და იმეორებს იგივე ოპერაციას. მიღებულ სტრუქტურას ავტორებმა უწოდეს vine tree აღნიშნულ მეთოდს კი - tree_to_vine.

ალგორითმის პირველი ფაზის მუშაობის სქემა გამოსახულია შემდეგ სურათზე:



ალოგირითმის მეორე ფაზაში ხდება მობრუნებების (მარცხნივ მობრუნება) ოპტიმალური K რაოდენობების დათვლა. K არგუმენტად გადაეცემა ფუნქციას, რომელიც K ჯერ მოახდენს vine tree-ს მობრუნებას მარცხნივ, რის შედეგადაც მიიღება დაბალანსებული ორობითი ხე. ალგორითმის მუშაობის სრულ სურათს აქვს შემდეგი სახე:



ალგორითმის პირველი ფაზა, tree_to_vine ფსევდო-კოდი:

1. tmp = root // დროებითი ცვლადი. თავიდან დგას ფესვზე
2. while (tmp != NULL)
   - ➔ თუ tmp ჰყავს მარცხენა შვილი
     მარჯვნივ მობრუნება ამ წვეროსთვის
     tmp-ს მიანიჭე მარცხენა შვილი, რომელიც ახლა მშობელი გახდა
   - ➔ else: tmp-ს მიანიჭე მარჯვენა შვილი

როცა ხეში n ცალი წვეროა: საუკეთესო შემთხვევაში, როცა ხე უკვე vine-ია, ციკლი სრულდება n-ჯერ და არცერთი მობრუნება არ სრულდება. უარეს შემთხვევაში კი, როცა ფესვს მარჯვენა შვილი არ ჰყავს - ციკლი სრულდება 2n - 1 ჯერ და აკეთებს n - 1 მობრუნებას. პირველი ფაზის დროითი შეფასება გამოდის O(n).

ალგორითმის მეორე ფაზა, vine_to_tree ფსევდოკოდი:

```
m = 2^|lg(n+1)| - 1;
გააკეთე m - n ცალი მობრუნება vine-ის თავიდან
while (m > 1)
    m = m / 2;
    გააკეთე m ცალი მობრუნება vine-ის თავიდან
```

მეორე ფაზის სირთულე რომ გამოვთვალოთ, დავაკვირდეთ, რომ ციკლის მიერ შესრულებული მობრუნებები ტოლია:

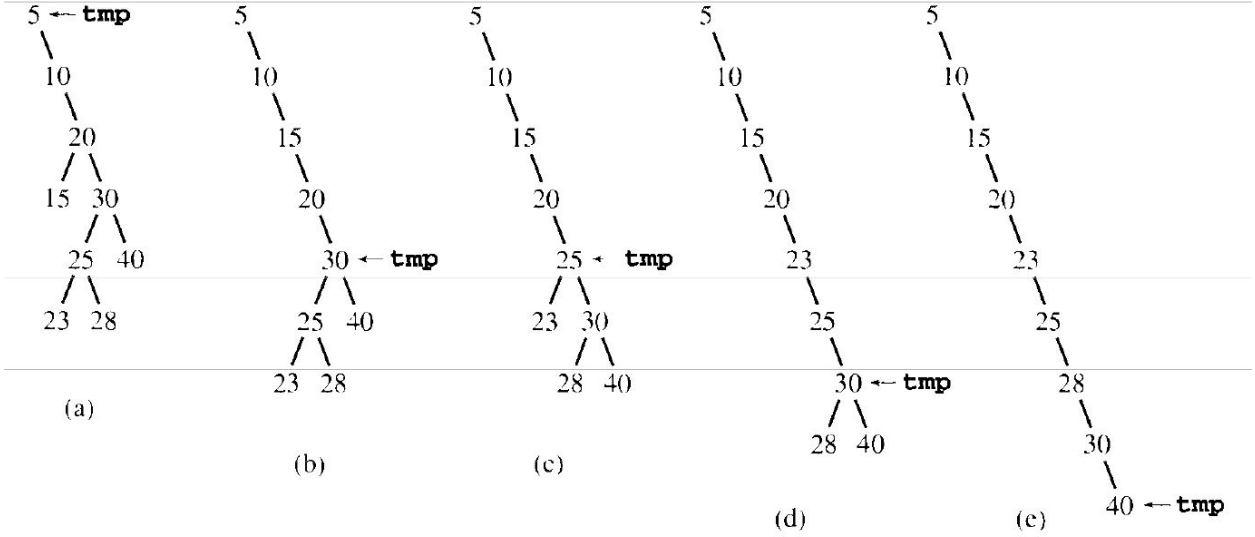
$$(2^{\lg(m+1)-1} - 1) + \ldots + 15 + 7 + 3 + 1 = \sum_{i=1}^{\lg(m+1)-1} (2^i - 1) = m - \lg(m+1)$$

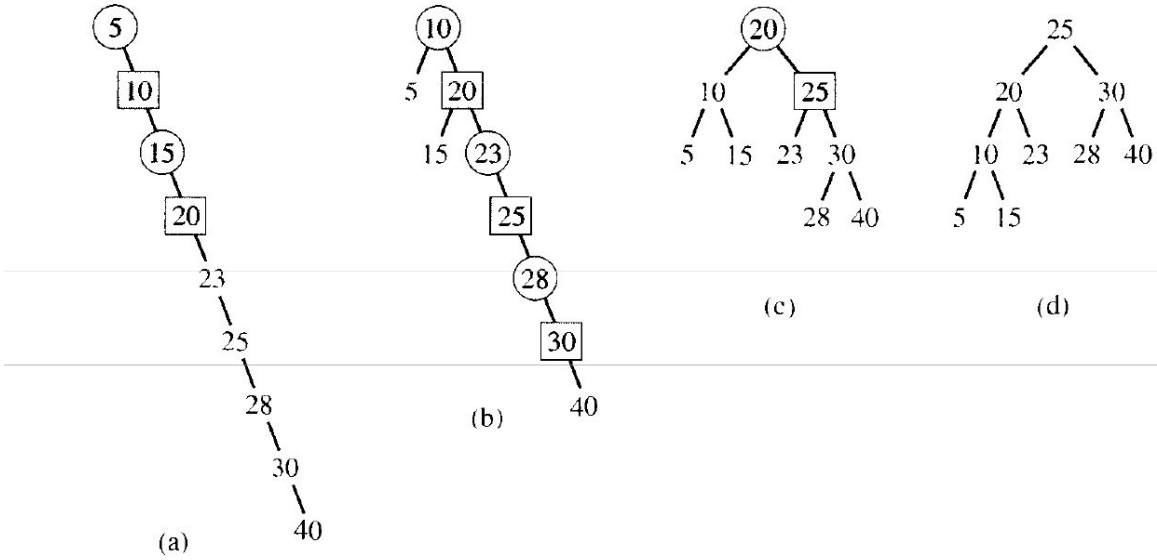
მობრუნებების რაოდენობა შეგვიძლია წარმოვადგინოთ ფორმულით:

$$n - m + (m - \lg(m+1)) = n - \lg(m+1) = n - |\lg(n+1)|$$

DSW ალგორითმის დროითი შეფასება გამოდის O(n).

შემდეგ ნაბაბზე მოცემულია ალგორითმის მუშაობის კონკრეტული შემთხვევა:


5 ← tmp
10
20
15 30
25 40
23 28
(a)
5
10
15
20
30 ← tmp
25 40
23 28
(b)
5
10
15
20
25 ← tmp
23 30
28 40
(c)
5
10
15
20
23
25
30 ← tmp
28 40
(d)
5
10
15
20
23
25
28
30
40 ← tmp
(e)


5
10
15
20
23
25
28
30
40
(a)
10
5 20
15 23
25
28
30
40
(b)
20
10 25
5 15 23 30
28 40
(c)
25
20 30
10 23 28 40
5 15
(d)

სეჯვიკის ალგორითმი

სეჯვიკის დინამიური დაბალანსების ალგორითმი მოითხოვს ძებნის ორობითი ხე იყოს დალაგებული (Order-statistic tree, შემოკლებით OST). ასეთ ხეში, გარდა სტანდარტული ძებნის ორობითი ხის ფუნქციებისა, შესაძლებელია მარტივად გადაიჭრას შემდეგი ორი ამოცანა:

- N ელემენტისაგან შედგენილ სიმრავლეში რიგით i- ური კვანძის (გასაღების სიდიდის მიხედვით) განსაზღვრა.
- მოცემული კვანძის რიგითი ნომრის განსაზღვრა (წინას შებრუნებული ამოცანა).

მის კვანძს დამატებული აქვს ველი size(x), რომელშიც ინახება x ფესვის მქონე ქვეხის ზომა, ანუ წვეროების რაოდენობა თავად x წვეროს ჩათვლით. ცხადია, სტრუქტურის ცვლილებასთან ერთად ზემოთაღწერილი ფუნქციები შესაბამის ცვლილებას განიცდიან. ჩავთვალოთ, რომ size(NULL) = 0. მაშინ შეგვიძლია ასეთი ტოლობა დავწეროთ:

size(x)= size(left(x)) + size(right(x)) + 1.

size(x) ველში ინფორმაციის განახლება ხეში ელემენტის ჩამატების ან წაშლის შემთხვევაში სირთულეს არ წარმოადგენს. ჩამატების დროს ყველა იმ ელემენტის size(x) ველი, რომელსაც ახალი ელემენტი შეედარება ხეში საკუთარი ადგილის პოვნამდე, 1-ით უნდა გავზარდოთ. წაშლის შემთხვევაში პირიქით, 1-ით უნდა შემცირდეს იმ ელემენტების size(x) ველი, რომლებიც წასაშლელი ელემენტის წინაპარს წარმოადგენენ. ჩვენ არ გვჭირდება წაშლა ost-ში. მხოლოდ size ველის დათვლა გვჭირდება არსებული წითელ-შავი ხისთვის, რომელსაც შევნახავთ იქ სადაც ფერს ვინახავდით, ამით დავზოგავთ მეხსიერებას.

სეჯვიკის ალგორითმი დაფუძნებულია დახარისხების ცნობილი ალგორითმის Quicksort-ის პრინციპზე. რეკურსიული მობრუნებების საშუალებით მედიანა-კვანძი ანუ სიდიდით საშუალო გასაღების მქონე კვანძი აჰყავს ფესვის ადგილზე (ფუნქციის რეკურსიული გამოძახებებით, ქვეხეების მედიანები ხდებიან ლოკალური ფესვები, გლობალური მედიანა კი - ხის ფესვი), შედეგად ვიღებთ დაბალანსებულ ხეს, რადგან თუ სიდიდით საშუალო კვანძი ფესვია, მასზე ნაკლები და მასზე მეტი გასაღების მქონე კვანძების რაოდენობა ტოლია და გადანაწილებულია მის მარცხენა და მარჯვენა ქვეხეებში.

ფუნქცია partR პარამეტრებად იღებს კვანძის მისამართს და რიგით ნომერს - მერამდენე რიგითი ელემენტიც უნდა ამოვიდეს ფესვად სტანდარტული მობრუნების ფუნქციების დახმარებით, იმ განსხვავებით, რომ მობრუნების გასათვალისწინებელია size ველის მნიშვნელობის დათვლაც.

```
Node* partR (Node* h, int k) {
        int t = size(h->left);
        if (t > k) {
                h->left = partR(h->left, k);
                h = rotR(h);
```

```
        }
        if (t < k) {
                h->right = partR(h->right, k-t-1);
                h = rotL(h);
        }
        return h;
}
```

ამ შემთხვევაში ინდექსის ათვლა იწყება 0-დან. ე.ი. მოხდება რიგით k+1 ელემენტის ფესვის ადგილზე ამოტანა. rotR და rotL ურთიერთსიმეტრიული სტანდარტული მობრუნების ფუნქციებია, მობრუნების შემდეგ დამატებით ითვლიან size-საც იმ კვანძებისთვის, რომლებმაც ცვლილება განიცადეს.

```
Node* rotR (Node* h) {
      Node* x = h->left;
        h->left = x->right;
        x->right = h;
        h->meta = size(h->left) + size(h->right) + 1;
        x->meta = size(x->left) + size(x->right) + 1;
        return x;
}
```

თავდაპირველად partR-ის გამოძახება ხდება ფესვის size/2-ისთვის.

```
Node* balanceR (Node* h) {
        if (h == NULL || h->size < 2) return h;
        h = partR (h, h->size / 2);
        h->left  = balanceR (h->left);
        h->right = balanceR (h->right);
        return h;
}
```

ჩვენი მოდიფიკაცია

ორობითი ხის წვეროების რაოდენობასა და ხის სიმაღლეს შორის არსებული დამოკიდებულების საფუძველზე შეგვიძლია წინასწარ დავხატოთ სრულად დაბალანსებული ხის სურათი. თუ ხე არ არის სრულად დაბალანსებული (შემდგომში სრული), მაშინ ის ყოველთვის შედგება ერთი სრული და მეორე არასრული ქვეხისგან.

ჩვენ შევადგენთ უტოლობას და N-ის მიხედვით მივხვდებით რომელი ქვეხე იქნება სრული და

რომელი არა.

h = 0	n = 1
h = 1	1 < n <= 3
. . .	. . .

$2^h \leq n < 2^{h+1}$

აქედან:

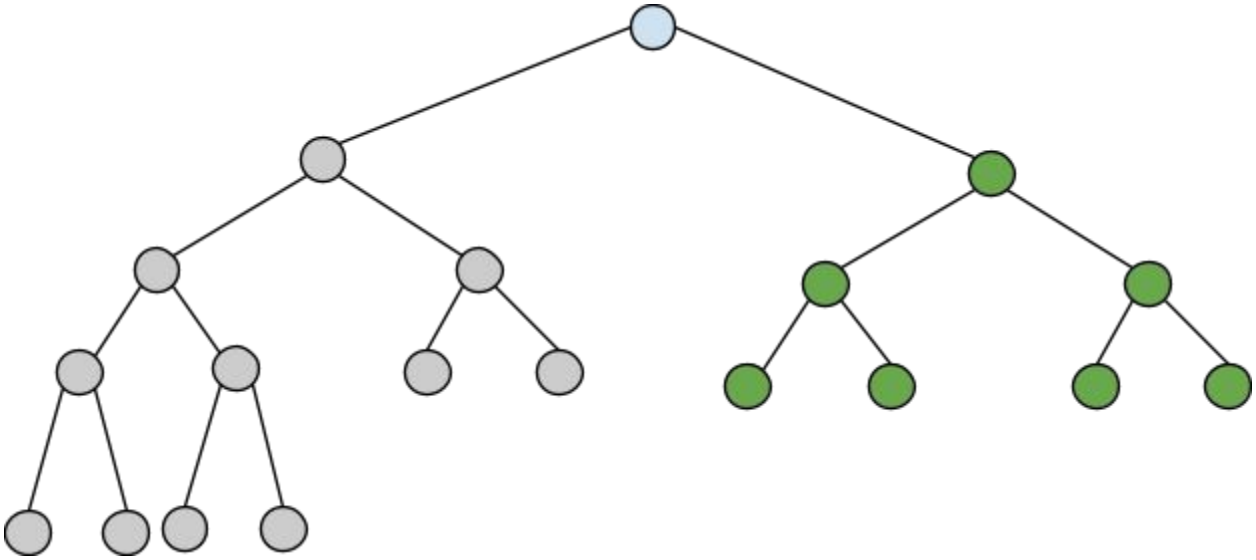
1ფესვი + $2^{h-1} - 1 + 2^h - 1 \geq n$

$n \leq 2^h + 2^{h-1} - 1$

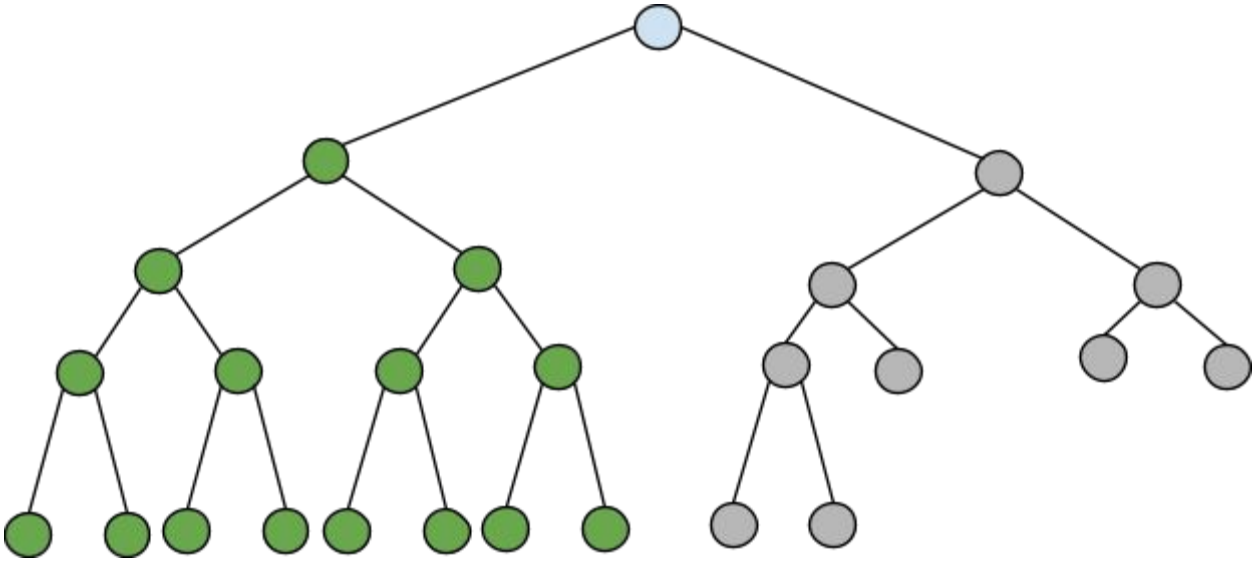
$n \leq 3 \cdot 2^{h-1} - 1$

ნახაზზე ნაჩვენებია შესაძლო ვარიანტები:

I შემთხვევა. მარჯვენა ქვეხე სრულია, მარცხენა - არასრული ($n <= 3 \cdot 2^{h-1} - 1$)



II შემთხვევა. მარცხენა ქვეხე სრულია, მარჯვენა - არასრული ($n > 3 \cdot 2^{h-1} - 1$)

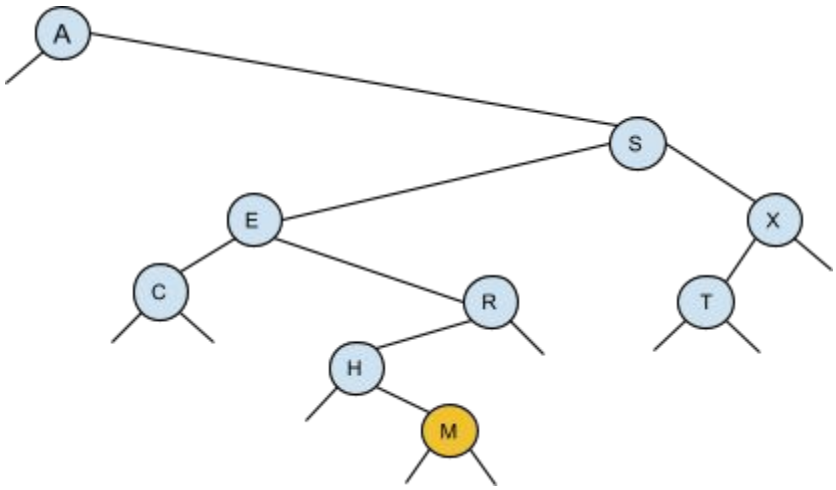


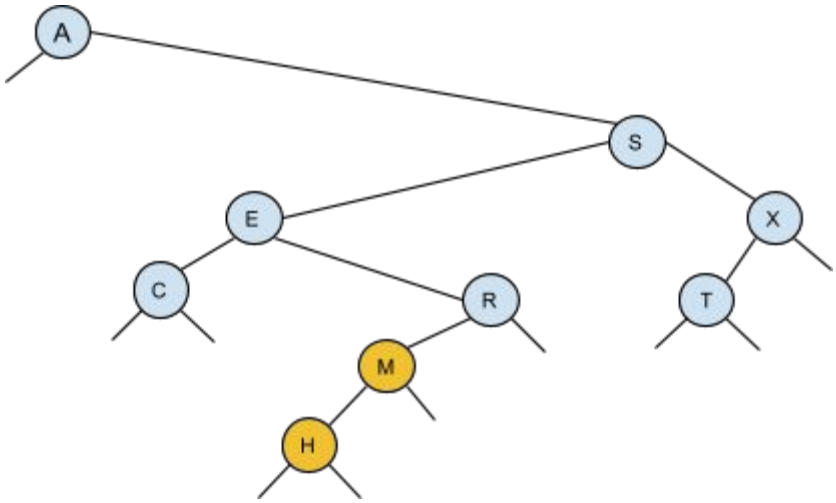
როგორც ვხედავთ, კვანძების რაოდენობის მიხედვით უნდა გამოვიცნოთ - დაბალანსების შემდეგ რომელი ქვეხე აღმოჩნდება სრული. შესაბამისად, სეჯვიკის ალგორითმისგან განსხვავებით, ჩვენ ბოლო უტოლობის საფუძველზე განვსაზღვრავთ თუ რომელი კვანძი ავიდეს ფესვად, ზუსტად შუა გასაღბის მქონე კვანძის ნაცვლად. უფრო დაწვრილებით შეგვიძლია ვიხილოთ პროგრამული ტკოდის ფრაგმენტში.

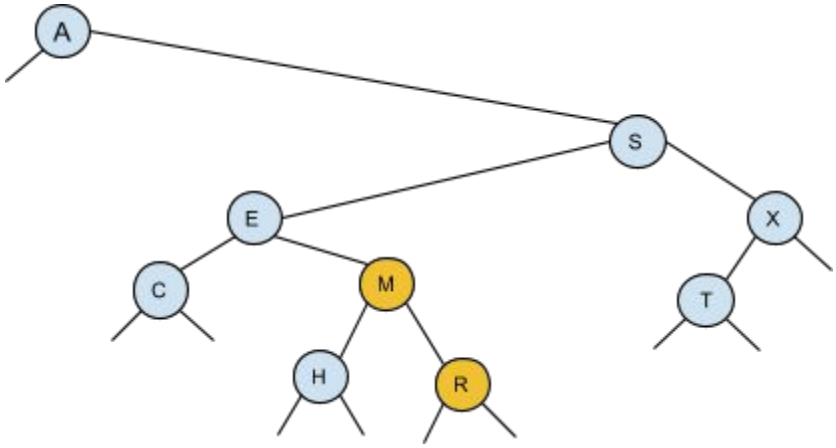
```
Node* balanceM (Node* x) {
  if (x == NULL || x->meta < 2) return x;
  int h = (int)log2(x->meta);
  if (x->meta <= 3 * (int)pow(2, h-1) - 1 ) {
        x = partR(x, x->meta - (int)pow(2, h-1) + 1 - 1);
        x-> left = balanceR(x->left);
        x->right = balanceM(x->right);
  } else {
        x = partR(x, (int)pow(2, h) - 1);
        x-> left = balanceM(x->left);
        x->right = balanceR(x->right);
  }
```

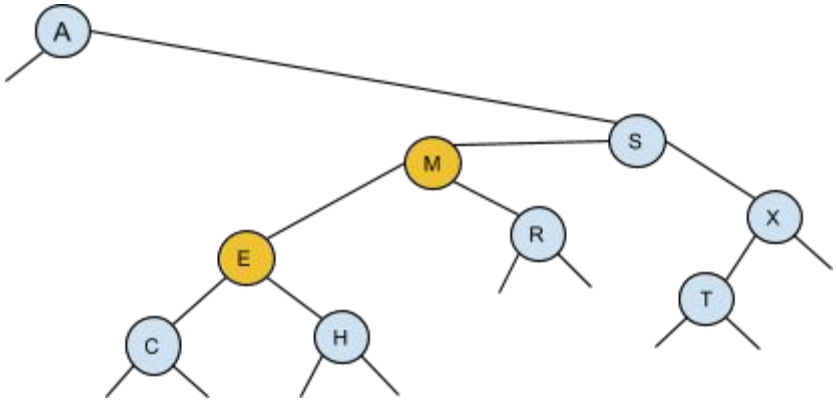
```
    return x;
}
```

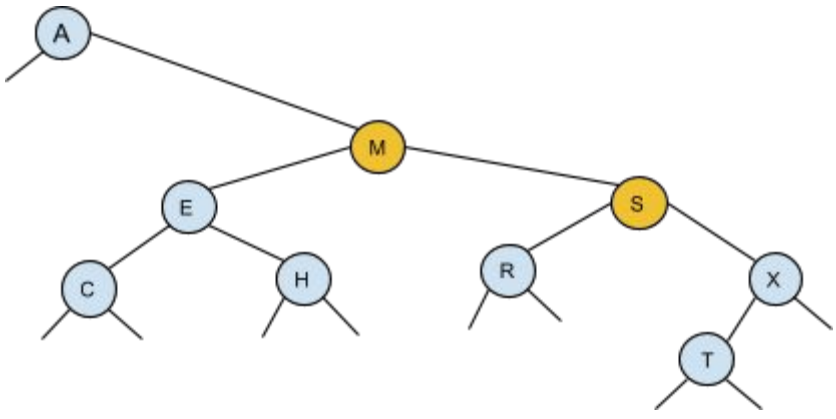
ნახაზზე ნაჩვენებია M (მედიანა) წვეროს ფესვის ადგილზე ამოტანა რეკურსიული ტრიალების საშუალებით:

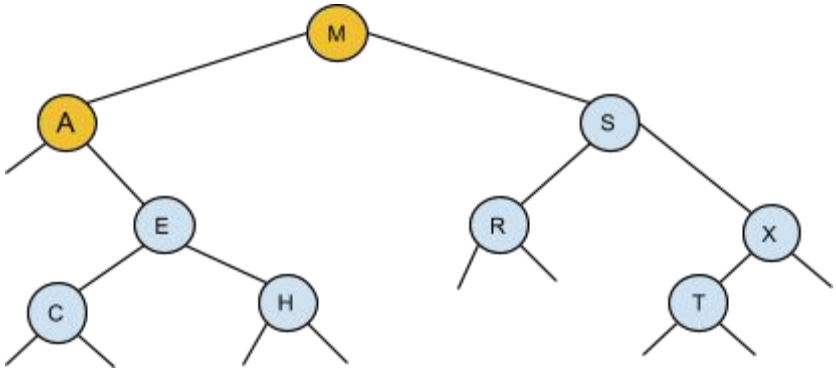





A
S
E
X
C
M
T
H
R


A
S
M
X
R
E
T
C
H


A
M
E
S
C
H
R
X
T



M ფესვი გახდა.

წითელ-შავი ხის დაბალანსების ალგორითმი

ალგორითმის ფსევდო-კოდი:

1. size ველის დათვლა. (RBT გადაიქცევა OST-დ)
2. დაბალანსება
3. ხის გადაღებვა. (OST გადაიქცევა RBT-დ)

კოდის ფრაგმენტი:

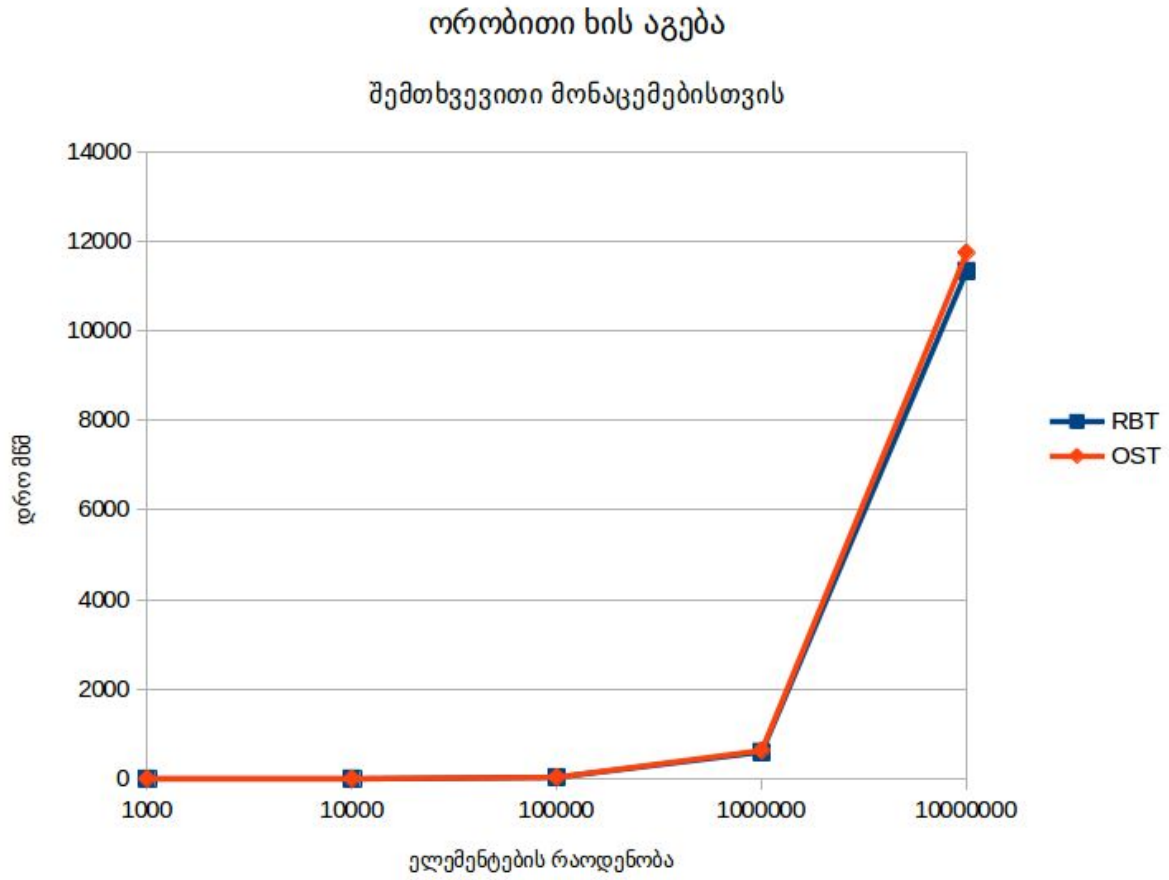
```
int size = updateSizes(root);// size ველის დათვლა.
int maxHeight = (int)log2(size); // სიმაღლის დათვლა ხის ზომის მიხედვით
root = balanceR(root); // დაბალანსება
colorTree(root, maxHeight, NULL); // color და parent ველების დათვლა
```

იდეალურ შემთხვევაში დაბალანსებული ხის ფესვის მარცხენა და მარჯვენა ქვეხის სიმაღლე და ელემენტების რაოდენობა ტოლია. განსხვავება სიმაღლეებს შორის შეიძლება იყოს მაქსიმუმ 1. ეს გარანტიას გვაძლევს რომ დაბალანსებული ხე დააკმაყოფილებს წითელ-შავი ხის თვისებებს. გადაღებვის დროს წვეროებს ვღებავთ შავად, ხის ბოლო დონეზე კი - წითლად. color ველის განახლებასთან ერთად ვანახლებთ parent ველს, რომლის მნიშვნელობაც მობრუნებების დროს “ფუჭდება”. შესაძლებელია ხის გადაღებვის უკეთესი მეთოდის მოფიქრება, რომელიც შემდგომი ჩასმებისთვის იქნება ოპტიმალური.

ტესტების შედეგები

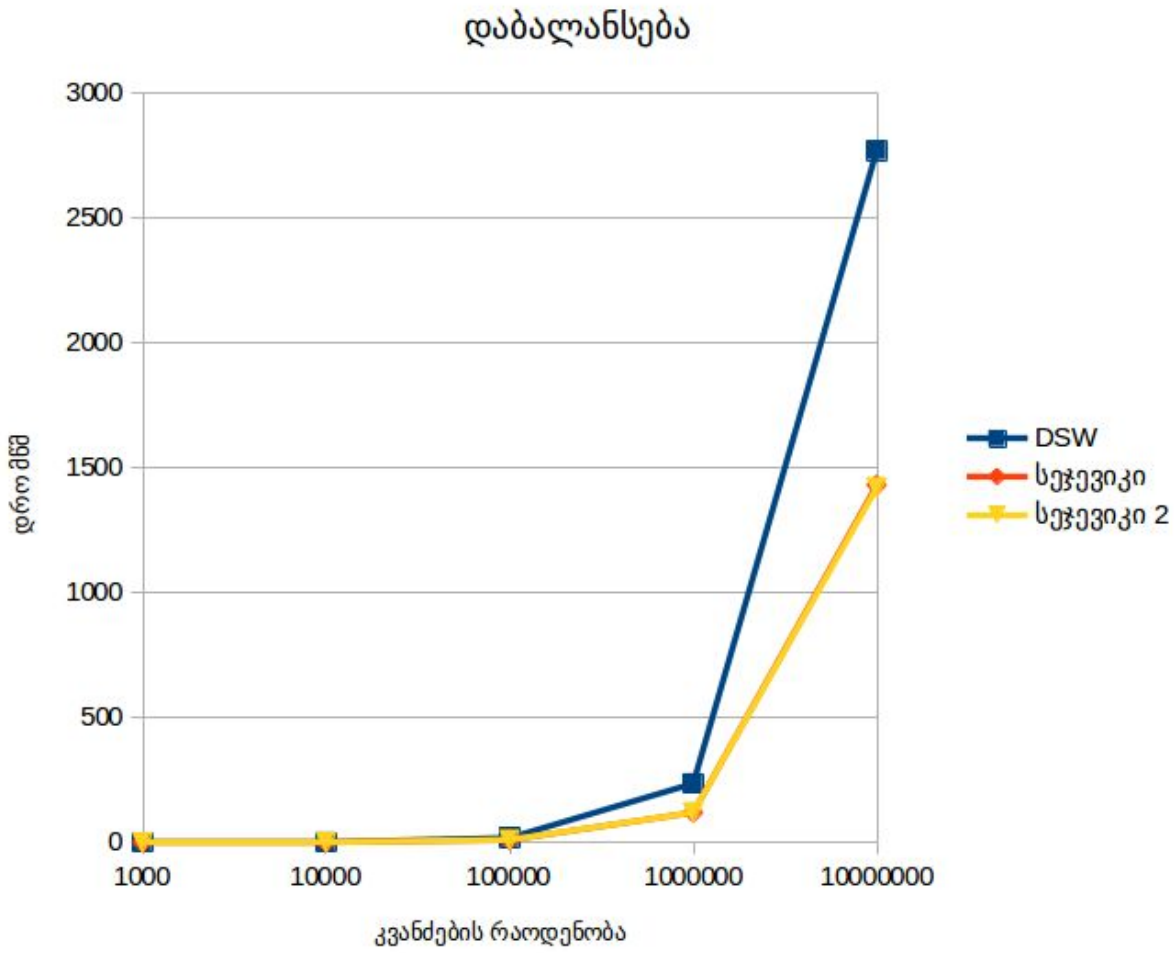
ქვემოთ ნაჩვენებია თუ რა დრო სჭირდება წითელ-შავი ხისა და დალაგებული ხის აგებას შემთხვევით შერჩეული მონაცემებისთვის (ორობითი ხის აგების დროები):

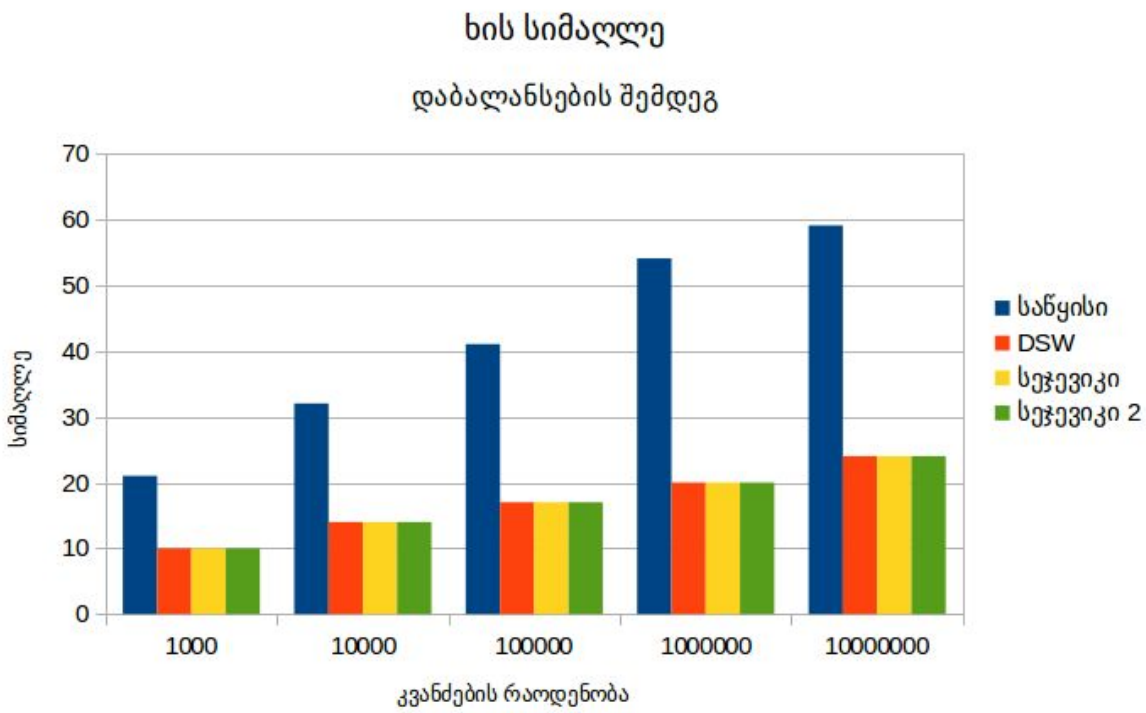
size	1000	10000	100000	1000000	10000000
RBT-time	0.217	2.658	34.728	599.934	11334
OST-time	0.171	2.512	38.035	632.868	11745.3
RBT-height	12	16	20	24	29
OST-height	20	30	45	52	64



ნაჩვენებია რა დრო სჭირდება შემთხვევით შერჩეული მონაცემებით უკვე აგებული დალაგებული ხის დაბალანსებას DSW ალგორითმით და სეჯვიკის ალგორითმით. ასევე სეჯვიკის ალგორითმის ჩვენი მოდიფიკაციით (მხოლოდ დაბალანსების დროები):

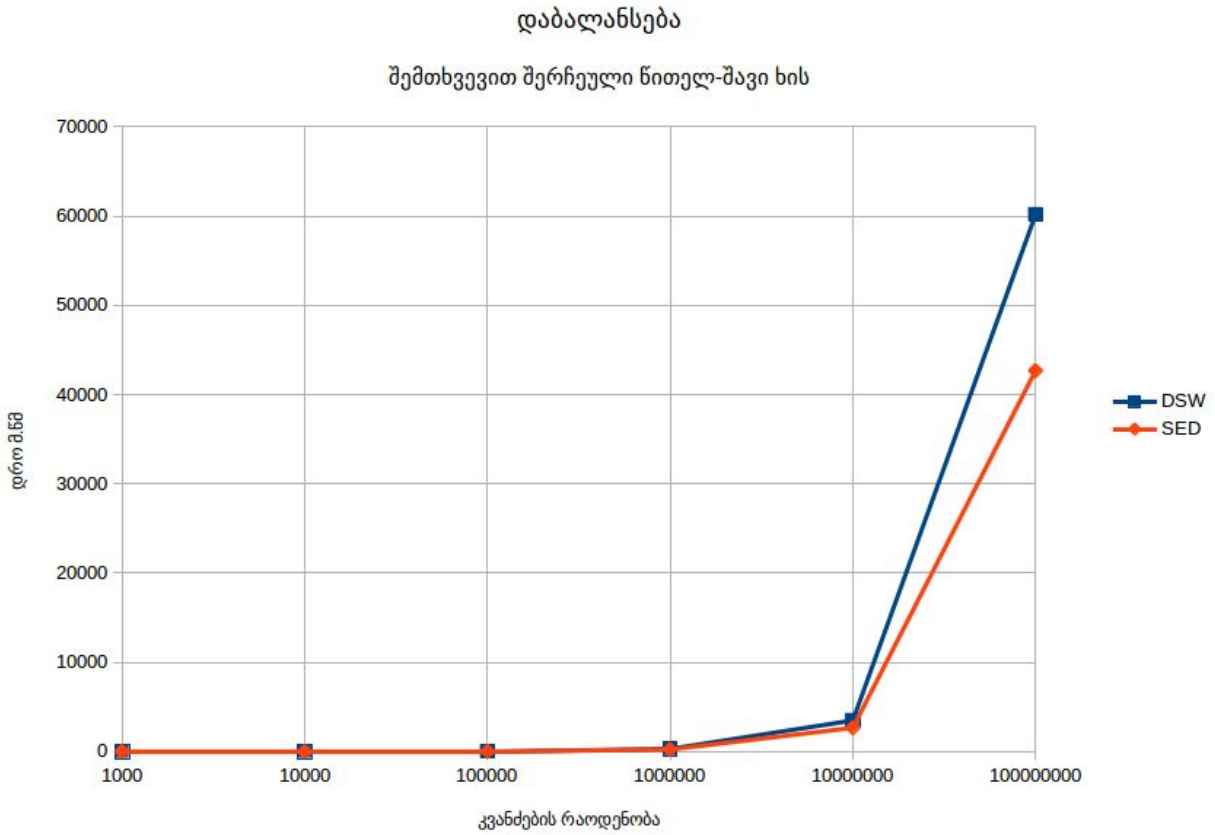
კვანძი	1000	10000	100000	1000000	10000000
DSW	0.043	1.957	19.522	237.064	2772.24
სეჯვიკი	0.069	1.274	10.842	120.862	1431.96
სეჯვიკი 2	0.085	1.237	11.52	120.602	1422.7





ნაჩვენებია რა დრო სჭირდება შემთხვევით შერჩეული მონაცემებით უკვე აგებული წითელ-შავი ხის დაბალანსებას DSW ალგორითმით და ჩვენი მიდგომით (სეჯვიკის ალგორითმით):

size	1000	10000	100000	1000000	10000000	10000000
DSW	0.043	0.735	19.25	296.419	3493.76	60169.3
SED	0.102	0.85	19.537	236.553	2663.49	42673.3



შედეგები მიღებულია 8 core cpu @ 1.6ghz პროცესორიან კომპიუტერზე.

დასკვნა

განვიხილეთ ორობითი ხის დაბალანსების ალგორითმები. ტესტებმა გვიჩვენა, რომ წითელ-შავი ხე გაცილებით სწრაფად აიგება და მისი სიმაღლე ხის ზომის გარკვეული მნიშვნელობის მიღწევის შემდეგ თითქმის ნახევრდება ჩვეულებრივ ორობით ხესთან შედარებით. ამასთან. თუ n > 1 000 000 პირობითად დავარქმევთ ამ ზღვარს გადაცილებულ მონაცემს დიდ მონაცემს, მაშინ პატარა მონაცემებისთვის წითელ-შავი ხის დაბალანსება უმჯობესია DSW ალგორითმის გამოყენებით. მაგრამ დიდი ხის შემთხვევაში ჩვენმა მიდგომამ მოგება მოგვცა დროის თვალსაზრისით. სხვაობა პატარაა, მაგრამ მონაცემების რაოდენობის ზრდასთან ერთად იზრდება.

დანართი: პროგრამული რეალიზაცია

პროექტი განთავსებულია მისამართზე https://github.com/offroader/bst და თითოეულ დაინტერესებულ პირს შეუძლია მისი გამოყენება.

კვანძის კლასი Node.cpp

```
#include <cstdlib>

using namespace std;

class Node {
public:
	int key;
	Node* parent;
	Node* left;
	Node* right;
	int meta;

	Node (int key) {
		this->key = key;
		parent = NULL;
		left = NULL;
		right = NULL;
		meta = 0;
	}

	virtual ~Node() {};
};
```

ხის კლასი Tree.cpp

```
#include <iostream>
#include <cstdlib>
#include <fstream>
#include <cmath>
#include "Node.cpp"
```

```
using namespace std;

class Tree {
public:
   Node* root;
   string mode;

        Tree () {
                root = NULL;
                mode = "RBT";
        }

        // insertion methods

        int insertNode(Node* z) {
                Node* y = NULL;
                Node* x = root;

                while (x != NULL) {
                        y = x;
                        if (z->key < x->key) {
                                x = x->left;
                        } else {
                                x = x->right;
                        }
                }
                z->parent = y;
                if (y == NULL) {
                        root = z;
                } else if (z->key < y->key) {
                        y->left = z;
                } else if (z->key > y->key) {
                        y->right = z;
                } else {
                        return 0;
                }
                z->meta = 1;

                insertFixup(z);
```

```
        return 1;
}

void insertFixup(Node* z) {
        while (z->parent != NULL && z->parent->meta == 1) {
                if (z->parent == z->parent->parent->left) {
                        Node* y = z->parent->parent->right;
                        if (y != NULL && y->meta == 1) {
                                z->parent->meta = 0;
                                y->meta = 0;
                                z->parent->parent->meta = 1;
                                z = z->parent->parent;
                        } else {
                                if (z == z->parent->right) {
                                        z = z->parent;
                                        rotateLeft(z);
                                }

                                if (z->parent != NULL) {
                                        z->parent->meta = 0;
                                        if (z->parent->parent != NULL) {
                                                z->parent->parent->meta = 1;
                                                rotateRight(z->parent->parent);
                                        }
                                }
                        }
                } else {
                        Node* y = z->parent->parent->left;
                        if (y != NULL && y->meta == 1) {
                                z->parent->meta = 0;
                                y->meta = 0;
                                z->parent->parent->meta = 1;
                                z = z->parent->parent;
                        } else {
                                if (z == z->parent->left) {
                                        z = z->parent;
                                        rotateRight(z);
                                }
```

```
                                if (z->parent != NULL) {
                                        z->parent->meta = 0;
                                        if (z->parent->parent != NULL) {
                                                z->parent->parent->meta = 1;
                                                rotateLeft(z->parent->parent);
                                        }
                                }
                        }
                }
        }
        root->meta = 0;
}

void rotateLeft(Node* x) {
        Node* y = x->right;
        x->right = y->left;

        if (y->left != NULL) {
                y->left->parent = x;
        }

        y->parent = x->parent;

        if (x->parent == NULL) {
                root = y;
        } else if (x == x->parent->left) {
                x->parent->left = y;
        } else {
                x->parent->right = y;
        }

        y->left = (x);
        x->parent = y;
}

void rotateRight(Node* x) {
        Node* y = x->left;
        x->left = (y->right);
```

```
        if (y->right != NULL) {
                y->right->parent = x;
        }

        y->parent = x->parent;

        if (x->parent == NULL) {
                root = y;
        } else if (x == x->parent->left) {
                x->parent->left = y;
        } else {
                x->parent->right = y;
        }

        y->right = x;
        x->parent = y;
}

// DSW balancing

int tree_to_vine (Node* r) {
  Node* vineTail = r;
  Node* remainder = vineTail->right;

  int size = 0;
  Node* tempPtr;
  while (remainder != NULL) {
          // if no leftward subtree, move rightward
          if (remainder->left == NULL) {
                  vineTail = remainder;
                  remainder = remainder->right;
                  size++;
          }
          // else eliminate the leftward subtree by rotations
          else {
                  // rightward rotation
                  tempPtr = remainder->left;
                  remainder->left = tempPtr->right;
                  tempPtr->right = remainder;
```

```
                remainder = tempPtr;
                vineTail->right = tempPtr;
            }
        }

        return size;
}

int fullSize(int size) {
        int n = 1;
        while (n <= size) {
                n = n + n + 1;
        }
        return n / 2;
}

void compression(Node* root, int count) {
        Node* scanner = root;
        // leftward rotation
        for (int i = 0; i < count; i++) {
                Node* child = scanner->right;
                scanner->right = child->right;
                scanner = scanner->right;
                child->right = scanner->left;
                scanner->left = child;
        }
}

void vine_to_tree(Node* root, int size) {
        int fullCount = fullSize(size);
        compression(root, size - fullCount);
        for (size = fullCount; size > 1; size /= 2) {
                compression(root, size / 2);
        }
}

void balanceDSW () {
        Node* pseudo_root = new Node(-1);
        pseudo_root->right = root;
```

```
        int size = tree_to_vine(pseudo_root);
        vine_to_tree(pseudo_root, size);

        root = pseudo_root->right;

        updateParents(root, NULL);
  }

  // balancing methods

  inline int size (Node* x) {
        if (x == NULL) return 0;
        return x->meta;
  }

  Node* rotR (Node* h) {
Node* x = h->left;
        h->left = x->right;
        x->right = h;

        h->meta = size(h->left) + size(h->right) + 1;
        x->meta = size(x->left) + size(x->right) + 1;

        return x;
  }

  Node* rotL (Node* h) {
Node* x = h->right;
h->right = x->left;
x->left = h;

h->meta = size(h->left) + size(h->right) + 1;
        x->meta = size(x->left) + size(x->right) + 1;

return x;
  }

  Node* partR (Node* h, int k) {
```

```
  int t = size(h->left);

  if (t > k) {
          h->left = partR(h->left, k);
          h = rotR(h);
  }

  if (t < k) {
          h->right = partR(h->right, k-t-1);
          h = rotL(h);
  }

  return h;
}


Node* balanceR (Node* h) {
  if (h == NULL || h->meta < 2) return h;

  h = partR (h, h->meta / 2);

  h->left  = balanceR (h->left);
  h->right = balanceR (h->right);

  return h;
}

Node* balanceM (Node* x) {
  if (x == NULL || x->meta < 2) return x;

  int h = (int)log2(x->meta);

  if (x->meta <= 3 * (int)pow(2, h-1) - 1 ) {

          x = partR(x, x->meta - (int)pow(2, h-1) + 1 - 1);

          x-> left = balanceR(x->left);
          x->right = balanceM(x->right);
```

```
        } else {

                x = partR(x, (int)pow(2, h) - 1);

                x-> left = balanceM(x->left);
                x->right = balanceR(x->right);
        }

        return x;
    }

    void balance () {
            int size = updateSizes(root); // converts to ost
            int maxHeight = (int)log2(size);
//          mode = "OST";
            root = balanceR(root);
            colorTree(root, maxHeight, NULL); // converts to rbt
//          mode = "RBT";
    }


    // destructor

    virtual ~Tree() {};

    // public methods to show performance

    int insert (int k) {
            return insertNode(new Node(k));
    }

    void draw () {
            drawTree(root);
    }

    void printInOrder () {
            printInOrder(root);
    }
```

```
        void printHeight () {
                cout << "Tree height: " << getHeight(root) << endl;
        }

        void printSize () {
                cout << "Tree size: " << count(root) << endl;
        }

        void destroy () {
                destroyTree(root);
        }

        // private methods

        void drawTree (Node* x) {
                static int node_level = -1;
                if (x != NULL) {
                        node_level += 1;
                        drawTree(x->right);
                        for (int i = 0; i < 7 * node_level; i++) {
                                printf(" ");
                        }
                        if (mode == "RBT") {
                                printf("%d - %d (%s)\n" , x->key, node_level, x->meta == 1 ? "R" :
"B");
                        } else {
                                printf("%d (%d)\n" , x->key, x->meta);
                        }
                        drawTree(x->left);
                        node_level -= 1;
          }
        }

        void destroyTree(Node* node) {
                if (node != NULL) {
                        destroyTree(node->left);
                        destroyTree(node->right);
                        delete node;
                }
```

```
}

inline int Max( int l, int r) {
        return l > r ? l : r;
}

void printInOrder (Node* node) {
        if (node != NULL) {
                printInOrder(node->left);
                cout << node->key << endl;
                printInOrder(node->right);
        }
}

void printRoot () {
        if (root != NULL) {
                cout << "Root is: " << root->key << endl;
        }
}

int updateSizes (Node* h) {
  if (h == NULL) return 0;

  int leftSize = updateSizes(h->left);
  int rightSize = updateSizes(h->right);

  h->meta = 1 + leftSize + rightSize;

  return h->meta;
}

// correct tree after balanceR
void updateParents (Node* node, Node* parent) {
        if (node != NULL) {
                updateParents (node->left, node);
                updateParents (node->right, node);
                node->parent = parent;
        }
}
```

```
        int getHeight (Node* node) {
                if (node == NULL) {
                        return 0;
                } else {
                        return 1 + Max (getHeight(node->left), getHeight(node->right));
                }
        }

        int count (Node* node) {
                if (node == NULL) {
                        return 0;
                } else {
                        return 1 + count(node->left) + count(node->right);
                }
        }

        void colorTree (Node* node, int maxHeight, Node* parent) {
                static int level = -1;
                if (node != NULL) {
                        level++;
                        node->meta = level < maxHeight ? 0 : 1; // set color BLACK(0) or RED(1)
                        node->parent = parent; // update parent field
                        colorTree(node->right, maxHeight, node);
                        colorTree(node->left, maxHeight, node);
                        level--;
          }
        }
};
```

სადემონსტრაციო პროგრამა:

```
#include <cstdlib>
#include <iostream>
#include <fstream>
#include <string>
#include <ctime>
#include <map>
#include <cmath>
```

```
#include "Tree.cpp"
#include "OST.cpp"
#include "RBT.cpp"

using namespace std;

void test_ostDSW_ostSED_ostMod(int);
void test_build_rbt_ost(int);
void test_rbtDSW_rbtSED(int);
void test_rbtDSW(int);
void test_rbtSED(int);

static int MAX_KEY_VALUE = 1000000000;
static int MAX_TREE_SIZE = 100000000;

int main (int argc, char** argv) {
	for (int n = 1000; n <= MAX_TREE_SIZE; n *= 10) {
		//test_rbtDSW_rbtSED(n);
	}
//	test_rbtDSW(MAX_TREE_SIZE);
	test_rbtSED(MAX_TREE_SIZE);
}

void test_ostDSW_ostSED_ostMod (int N) {
	if (!N || N > MAX_TREE_SIZE) {
		cout << "Invalid tree size" << endl;
		return;
	}

	cout << "Test OST balancing using DSW and Sedgewick's algorithm (also our
modification) for " << N << " elements" << endl;

	srand (time(NULL));

	cout << endl<< "N:" << N << endl;

	map<int, int> mymap;
```

```
OST* ost1 = new OST();
OST* ost2 = new OST();
OST* ost3 = new OST();

for (int i = 0; i < N; i++) {
	int k = rand() % MAX_KEY_VALUE;

	if (mymap.find(k) == mymap.end()) {
		mymap.insert(pair<int,int>(k, 1));
		ost1->insert(k);
		ost2->insert(k);
		ost3->insert(k);
	} else {
		i--;
		continue;
	}
}

clock_t start, finish;

cout << "Initial: " << endl;
ost1->printHeight();
ost2->printHeight();
ost3->printHeight();

cout << endl;

start = clock();
ost1->balanceDSW();
finish = clock();
cout << "DSW balancing time: " << ((double) (finish - start)) / 1000 << " ms" << endl;
ost1->printHeight();

start = clock();
ost2->balance();
finish = clock();
cout << "Sed. balancing time: " << ((double) (finish - start)) / 1000 << " ms" << endl;
ost2->printHeight();
```

```
	start = clock();
	ost3->balanceM();
	finish = clock();
	cout << "Modified Sed. balancing time: " << ((double) (finish - start)) / 1000 << " ms" <<
endl;
	ost3->printHeight();

	ost1->destroy();
	ost2->destroy();
	ost3->destroy();
}

void test_build_rbt_ost (int tree_size) {
	if (!tree_size || tree_size > MAX_TREE_SIZE) {
		cout << "Invalid tree size" << endl;
		return;
	}

	cout << "Test RBT and OST building times for " << tree_size << " elements" << endl;

	map<int, int> mymap;
	int* arr = new int[tree_size];

	for (int i = 0; i < tree_size; i++) {
		int k = rand() % MAX_KEY_VALUE;

		if (mymap.find(k) == mymap.end()) {
			mymap.insert(pair<int,int>(k, 1));
			arr[i] = k;
		} else {
			i--;
			continue;
		}
	}
	//for (map<int, int>::iterator it = mymap.begin(); it != mymap.end(); ++it) {}

	// check
	cout << "map size: " << mymap.size() << endl;
```

```
	RBT* rbt = new RBT();
	OST* ost = new OST();
	clock_t start, finish;

	start = clock();
	for (int i = 0; i < tree_size; i++) {
		rbt->insert(arr[i]);
	}
	finish = clock();
	cout << "RBT building time: " << ((double) (finish - start)) / 1000 << " ms" << endl;


	start = clock();
	for (int i = 0; i < tree_size; i++) {
		ost->insert(arr[i]);
	}
	finish = clock();
	cout << "OST building time: " << ((double) (finish - start)) / 1000 << " ms" << endl;

	rbt->printSize();
	rbt->printRoot();
	rbt->printHeight();

	ost->printSize();
	ost->printRoot();
	ost->printHeight();

	rbt->destroy();
	ost->destroy();

	free(arr);

	cout << "Test finished." << endl;
}

void test_rbtDSW_rbtSED1 (int tree_size) {
	if (!tree_size || tree_size > MAX_TREE_SIZE) {
		cout << "Invalid tree size" << endl;
		return;
```

```
}
cout << "Test red-black tree balancing times for " << tree_size << " elements" << endl;

int* arr = new int[MAX_TREE_SIZE];

Tree* t1 = new Tree();
Tree* t2 = new Tree();
clock_t start, finish;

cout << "building...." << endl;

for (int i = 0; i < tree_size; i++) {
        int k = rand() % MAX_KEY_VALUE;
        if (arr[k] != 1) {
                arr[k] = 1;
                t1->insert(k);
                t2->insert(k);
                if (i % 1000000 == 0) cout << i << endl;
        } else {
                i--;
                continue;
        }
}

free(arr);

cout << "balancing...." << endl;

start = clock();
t1->balanceDSW();
finish = clock();
cout << "DSW balancing time: " << ((double) (finish - start)) / 1000 << " ms" << endl;

start = clock();
t2->balance();
finish = clock();
cout << "Sedge balancing time: " << ((double) (finish - start)) / 1000 << " ms" << endl;

t1->printHeight();
```

```
	t2->printHeight();

	t1->destroy();
	t2->destroy();

	cout << "Test finished." << endl;
}

void test_rbtDSW (int tree_size) {
	if (!tree_size || tree_size > MAX_TREE_SIZE) {
		cout << "Invalid tree size" << endl;
		return;
	}
	cout << "Test red-black tree balancing times for " << tree_size << " elements" << endl;

	Tree* tree = new Tree();
	clock_t start, finish;

	start = clock();
	cout << "building...." << endl;
	for (int i = 0; i < tree_size; i++) {
		tree->insert(rand() % MAX_KEY_VALUE);
	}
	finish = clock();
	cout << "building time: " << ((double) (finish - start)) / 1000 << " ms" << endl;


	cout << "balancing...." << endl;
	start = clock();
	tree->balanceDSW();
	finish = clock();
	cout << "DSW balancing time: " << ((double) (finish - start)) / 1000 << " ms" << endl;

	tree->printSize();
	tree->printHeight();

	tree->destroy();

	cout << "Test finished." << endl;
```

```
}


void test_rbtSED (int tree_size) {
	if (!tree_size || tree_size > MAX_TREE_SIZE) {
		cout << "Invalid tree size" << endl;
		return;
	}
	cout << "Test red-black tree balancing times for " << tree_size << " elements" << endl;

	Tree* tree = new Tree();
	clock_t start, finish;

	start = clock();
	cout << "building...." << endl;
	for (int i = 0; i < tree_size; i++) {
		tree->insert(rand() % MAX_KEY_VALUE);
	}
	finish = clock();
	cout << "building time: " << ((double) (finish - start)) / 1000 << " ms" << endl;


	cout << "balancing...." << endl;
	start = clock();
	tree->balance();
	finish = clock();
	cout << "Sedgewick balancing time: " << ((double) (finish - start)) / 1000 << " ms" <<
endl;

	tree->printSize();
	tree->printHeight();

	tree->destroy();

	cout << "Test finished." << endl;
}
```

გამოყენებული ლიტერატურა

1. **Introduction To Algorithms, 3rd Edition, Thomas H. Cormen. The MIT Press, 2009**
2. **One-time binary search tree balancing: the Day/Stout/Warren (DSW) algorithm, Timothy J. Rolfe - Eastern Washington University, Cheney, Washington**
   Published in: Newsletter ACM SIGCSE Bulletin Volume 34 Issue 4, December 2002, Pages 85-88
3. **Algorithms in C, Vol. I (3rd edition; Addilson-Wesley, 1998), Robert Sedgewick**
4. **Data structures and algorithms in c++, 2nd edition, Adam Drozdek**
5. **ალგორითმები და მონაცემთა სტრუქტურები 2015, კობა გელაშვილი**
6. **მონაცემთა სტრუქტურები 2012 (მაგისტრებისთვის), კობა გელაშვილი**
7. სასარგებლო ბმულები:
   - http://en.wikipedia.org/wiki/Red%E2%80%93black_tree
   - http://en.wikipedia.org/wiki/AVL_tree
   - http://en.wikipedia.org/wiki/Order_statistic_tree
   - http://en.wikipedia.org/wiki/Day%E2%80%93Stout%E2%80%93Warren_algorithm
   - http://penguin.ewu.edu.s3.gvirabi.com/~trolfe/DSWpaper/