RB-ხის დაბალანსების ალგორითმი

შესავალი

ფორმალურად, ძებნის ორობითი ხის დაბალანსების თემა კომპიუტერული მეცნიერებისთვის საკმაოდ ძველია, თუმცა შედეგების მრავალფეროვნებით არ გამოიჩევა, - ამჟამად ძირითადად ორი ალგორითმი არის გავრცელებული ([2], [3]. 1976 წელს გამოქვეყნდა დეის (A. Colin Day) ნაშრომი (იხ. [1]), აქ შემოთავაზებული ალგორითმი დამატებითი მეხსიერების გამოყენების გარეშე, წრფივ დროში აბალანსებს ძებნის ორობით ხეს. თეორიულად ეს შეფასება გაუმჯობესებას არ ექვემდებარება. იმპლემენტაციის თვალსაზრისით უფრო დახვეწილი არის ამ ალგორითმის ვარიანტი, რომელიც ქმნის სრულ ხეს (იხ. [2]). რობერტ სეჯევიკის (Robert Sedgewick) წიგნში (იხ. [3]) მოყვანილია რეკურსიული ალგორითმი, ხის კვანძების n რაოდენობის მიმართ უარეს შემთხვევაში $n \log n$ შეფასებით. პრაქტიკაში გამოყენებული ხეებისთვის უმეტეს შემთხვევაში ის მუშაობს უფრო სწრაფად ვიდრე დეის ალგორითმი, თუმცა მისი გამოყენებისთვის საჭიროა ძებნის ორობითი ხის კვანძის სტრუქტურის გაძლიერება ქვეხის ზომის დამატებით.

რადგან პრაქტიკაში გასაგები მიზეზების გამო უპირატესობა ენიჭება დაბალანსებული ხეების, RB (იხ. [4]) და AVL ხეების (იხ. [5]) გამოყენებას, ამიტომ ძებნის ორობითი ხის დაბალანსების ალგორითმები არ მიეკუთვნებოდა უაღრესად აქტუალურ თემებს. თუმცა, ბოლო წლებში მდგომარეობა იცვლება (თუ უკვე არ შეიცვალა) შემდეგი მიზეზების გამო:

- გამოყენებებში სულ უფრო დიდი მოცულობის მონაცემების გამოყენება ხდება აუცილებელი;
- კომპიუტერები პასიურ მდგომარეობაში (მაგ. hibernate) უცებ არ გადადიან, და არსებობს ამ პერიოდში გარკვეული სტრუქტურების ხელახალი დაბალანსების შესაძლებლობა;
- თანამედროვე კომპიუტერებს შეუძლიათ პროცესების დაპარალელება, 4 და 8 და მეტი thread -ის უზრუნველყოფა შეუძლიათ ეკონომ-კლასის ლეპტოპებს.

რეალურ ამოცანებში RB-ხის სიმაღლეს აქვს მიდრეკილება რომ დასაშვები თეორიული შეფასების მაქსიმუმისკენ წავიდეს. შესაბამისად, საგრძნობი ხდება RB-ხის სიმაღლის $2\log(n + 1)$ მნიშვნელობას და სრული ხის $\log n$ მნიშვნელობას შორის სხვაობა. მაგალითად, როდესაც $n = 1000000$, ეს განსხვავება დაახლოებით 20-ის ტოლია.

ისევ იმ ფაქტის გათვალისწინებით, რომ თანამედროვე გამოყენები, მათ შორის ლინუქსის ბირთვი, ძირითადად RB-ხეებს იყენებენ, ხოლო მათთვის ორივე აღნიშნული ალგორითმი წრფივ დროში მუშაობს (RB-ხისთვის ეს მხოლოდ ექსპერიმენტულად დასტურდება), მათი ხელახალი დაბალანსება შესაძლოა მოხდეს ძალიან სწრაფად, განსაკუთრებით პარალელური ალგორითმების გამოყენების შემთხვევაში.

წარმოდგენილ ნაშრომში ჩვენ შევეცდებით უშუალოდ RB-ხის ხელახალი დაბალანსების საუკეთესო სერიული ალგორითმის შერჩევას. კერძოდ, მოვიყვანთ სეჯევიკის ალგორითმის ვარიანტს, რომელიც ისეთივე კავშირშია საბაზო ალგორითმთან, როგორც DSW დეის ალგორითმთან: შედეგად მოგვცემს სრულ ხეს.

ამ მიზნით, ჩვენ ოპტიმალურად გამოვიყენებთ RB-ხის სტრუქტურას: აუცილებლობის შემთხვევაში ფერისთვის გამოყოფილ ველს დავტვირთავთ ქვეხეთა ზომებით (გადავერთვებით დალაგებულ ხეზე -OST), დაბალანსების შემდეგ ხელახლა გადავღებავთ ხეს და აღვადგენთ RB-ხის თვისებებს. სხვა სიტყვებით, სეჯევიკის ალგორითმის დასახვეწად ჩვენ RB-ხეს განვიხილავთ ჰიბრიდულ მონაცემთა სტრუქტურად.

შედეგების თვალსაჩინოებისთვის ჩვენ მოგვყავს C++ ენაზე შესრულებულ კოდის ფრაგმენტები და ვხატავთ ხეებს github -ზე მოთავსებული კოდის მიხედვით:

https://github.com/mooseman/pdlinkedlist/blob/master/draw-tree.c

რომლის ავტორადაც მითითებულია დანიელ სლეიტორი (Daniel Sleator) (იხ. http://www.cs.cmu.edu/~sleator).

ჰიბრიდული RB-ხე

ჰიბრიდული მონაცემთა სტრუქტურა ძალიან მჭიდრო კავშირშია დაბალანსების არსებულ ალგორითმებთან. ჰიბრიდული RB-ხე ნიშნავს, რომ გარკვეულ შემთხვევებში, დროებით, ფერის ველს ვანიჭებთ სხვა დატვირთვას, რაც მოგვცემს იმ უნარებს, რაც არ გააჩნია RB-ხეს. თუმცა, კვანძის სტრუქტურა უცვლელი რჩება.

წარმოდგენილი ნაშრომის წინანდელ მოხსენებაში განხილული იყო ჰიბრიდული RB-ხე, რომელსაც ჰქონდა AVL ხედ გარდაქმნის (და პირიქით) უნარი. ასეთი სტურქტურის მოტივაცია მარტივია. RB-ხის ოპერაციები სრულდება გარანტირებულ ლოგარითმულ დროში, თუმცა ძებნის ოპერაციები შედარებით ნელია AVL ხესთან შედარებით. თუ პრაქტიკულ გამოყენებებსი იქნება სიტუაცია, როდესაც სრულდება კვანძების ჩამატება-წაშლების გრძელი სერია, შემდეგ ძებნის ოპერაციების გრძელი სერია, და ა.შ., მაშინ მიზანშეწონლილია ძებნის ოპერაციების სერიის წინ RB-ხე გარდავქმნათ AVL ხედ და აუცილებლობის შემთხვევაში განვახორციელოთ შებრუნებლი გარდაქმნა. AVL ხედ გარდაქმნა მოხდება საწყისი ხის inorder სტილში შემოვლით და კვანძების ზრდადი რიგით ამოღება RB-ხიდან და იგივე რიგით ჩამატება AVL -ხეში. ფაქტიურად, RB-ხის ხელახალი დაბალანსებისთვის დეის ალგორითმის გამოყენება თითქმის იგივეა რაც AVL ხედ გარდაქმნა (იმპლემენტაციის დეტალების სიზუსტით). ტექნიკურად არავითარი დაბრკოლება არ ჩნდება, რადგან პროგრამულად ყველაფერი კეთდება ერთი კლასის ფარგლებში, რომელსაც ემატება ერთი - რეჟიმის (mode) ველი. კვანძის სტრუქტურა უცვლელი რჩება. გვაქვს ორი fixup ჩამატებისთვის და ორიც წაშლისთვის- რეჟიმის მიხედვით. RB-ხე და AVL ხე მხოლოდ ამ fixup -ებიტ განსხვავდება ერთმანეთისგან. გვაქვს აგრეთვე მატი ერთმანეთში გადაყვანის ალგორითმები, რომელთაგან ერთი ნიშნავს RB-ხის ხელახალ დაბალანსებას დეის ალგორითმით.

პრქტიკული თვალსაზრისით უფრო საინტერესოა ჰიბრიდული RB-ხე, რომელსაც სწრაფად, წრფივ დროში შეუძლია OST-დ გარდაქმნა და და პირიქით, RB-ხის სტრუქტურის აღდგენა. ამ ჰიბრიდს აქვს OST-ს სპეციფიკური ალგორითმები, სტრუქტურების გარდაქმნის და ხელახალი დაბალანსების სწრაფი ალგორითმი რომლის საშუალებითად შეგვიძლია მომზადებული შევხვდეთ ძებნის ოპერაციების გრძელ სერიას. ხელახალი დაბალანსების სწრაფი ალგორითმი წარმოადგენს სეჯევიკის ალგორითმის დახვეწილ ვარიანტს და ჩვენს ტესტებზე სრულდება საგრძნობლად სწრაფად, ვიდრე DSW -ალგორითმი. აღსანიშნავია, რომ ჰიბრიდული RB-ხის იმპლემენტაცია მოცულობით უმნიშვნელოდ განსხვავდება ჩვეულებრივი შემთხვევისგან.

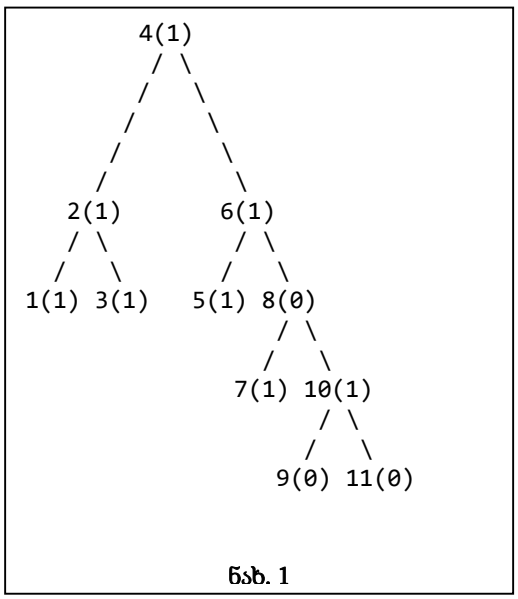
ჰიბრიდული RB-ხის ხელახალი დაბალანსება სეჯევიკის ალგორითმით

ვიგულისხმოთ, რომ კვანძის სტრუქტურა შემდეგია:

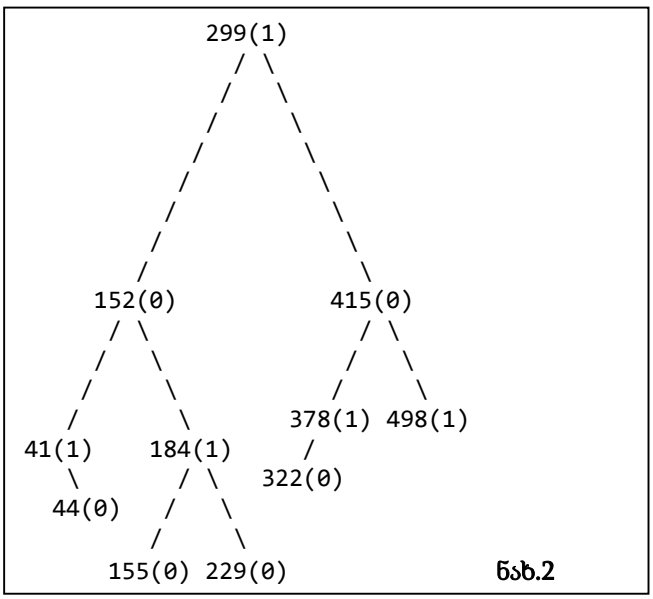
```
template<typename T>
struct Node
{
    T key;
    Node* p;
    Node* child[2];
    int bf;
    Node();
    Node(T keyValue);
};
```

RB-ხისთვის `int bf;` ველი ასრულებს ფერის ფუნქციას, თუ ხეს გარდავქმნით AVL-ად, იგი იქნება ე.წ. ბალანს-ფაქტორი, OST-ს შემთხვევაში იგი შეინახავს კვანძის ქვეხის ზომას. კონსტრუქტორები მას ანიჭებენ 0-ს (წითელი).

მაგალითად განვიხილოთ ორი 14-ელემენტიანი ახლადშექმნილი RB-ხე. მარცხენაში ელემენტები შეტანილია ზრდადი რიგით, ხოლო მეორეში ჩასმულია შემთხვევითი გასაღებებით (გარკვეულ დიაპაზონში) შექმნილი კვანძები:



ნახ. 1



ნახ.2

წითელ კვანძებს ფრჩხილებში უწერია 0, შავებს კი 1.

მარჯვენა და მარცხენა შვილებისთვის

```
Node* child[2];
```

მასივის გამოყენება საშუალებას გვაძლევს შევქმნათ სწრაფი კოდი და ამარტივებს რამდენიმე ალგორითმს. ჩვენს მიერ შემუშავებული დაბალანსების ახალი ალგორითმი მუშაობს როდესაც ხე იმყოფება OST რეჟიმში. იგი რეკურსიულია და გარემოების მიხედვით იძახებს ან საკუთარ თავს, ან სეჯევიკის ალგორითმს, ამიტომ ამ უკანასკნელს აგრეთვე აღვწერთ მოკლედ ჩვენს ტერმინებში. RB-ხის რეჟიმის შეცვლა ხორციელდება შემდეგი საზიარო (public) მეთოდით:

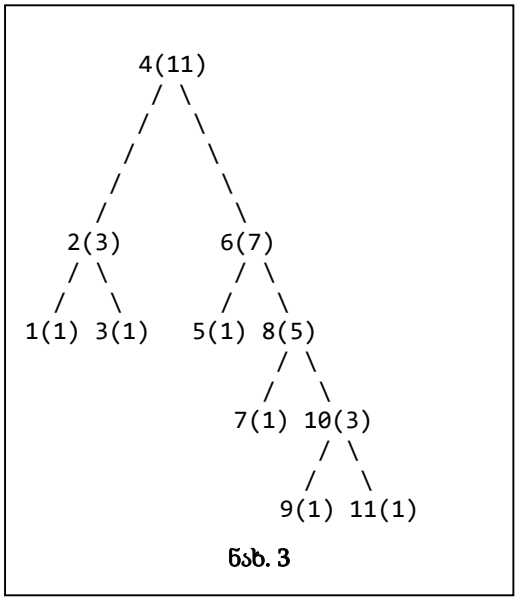
```
template<typename T>
void Tree<T>::setModeOST(void)
{
	mode = OST;
	updateSizes(root);
}
```

რომელიც თავის მხრივ იყენებს კერძო მეთოდს

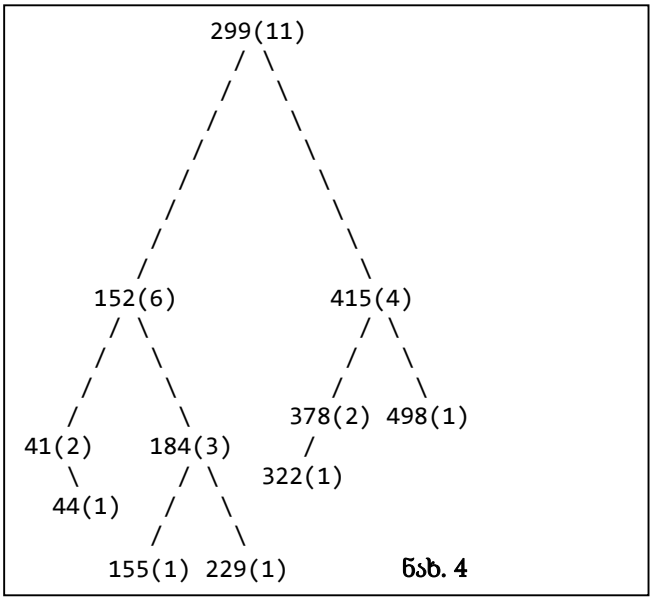
```
template<typename T>
int Tree<T>::updateSizes(Node<T>* h)
{
	if (h == NULL) return 0;

	int lN = updateSizes(h->child[0]);
	int rN = updateSizes(h->child[1]);
	h->bf = 1 + lN + rN;

	return h->bf;
}
```

ცხადია, იგი მუშაობს წრფივ დროში. OST რეჟიმიდან RB-ზე დაბრუნება აგრეთვე მარტივად ხორციელდება და ამიტომ მას არ აღარ განვიხილავთ.

შემდეგი ორი ნახაზი გვიჩვენებს იგივე ნახაზებზე 1 და 2 გამოსახულ ხეებს რეჟიმის შეცვლის შემდეგ. ფრჩხილებში ახლა წერია ქვეხის ზომა:



ნახ. 3



ნახ. 4

სეჯევიკის ალგორითმი იყენებს დამხმარე ალგორითმს, რომელსაც სიდიდით k-ური გასაღების მქონე კვანძი აჰყავს ფესვად (using an algorithm that partitions a binary search tree, moving the k^{th} entry to the root). ჩვენს აღნიშვნებში მას აქვს სახე

```
template<typename T>
Node<T>* Tree<T>::partR(Node<T>* h, int k)
{
	int t = N(h->child[0]);
	if (t != k)
	{
		int dir = (t < k)?1:0;
		h->child[dir] = partR(h->child[dir], k - dir*(t+1));
		h = rotate(h, !dir);
	}
	return h;
}
```

და წარმოადგენს კლასის კერძო მეთოდს. კიდევ ერთი კერძო მეთოდი

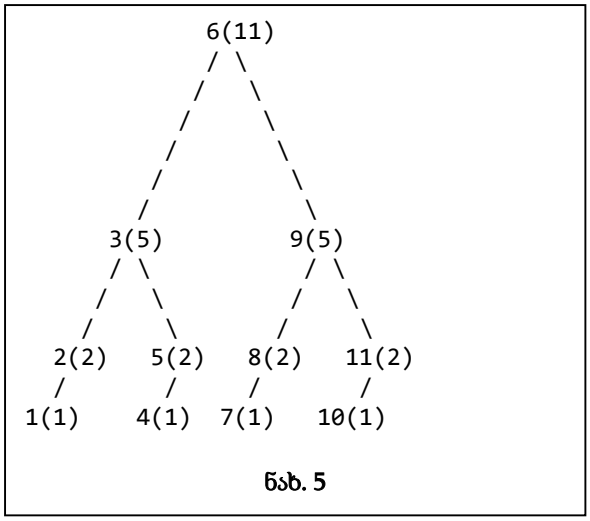
```
Node<T>* Tree<T>::balanceR(Node<T>* h)
{
	if (h == NULL || h->bf < 2) return h;
	h = partR(h, h->bf / 2);

	h->child[0] = balanceR(h->child[0]);
	if (NULL != h->child[0])   h->child[0]->p = h;
	h->child[1] = balanceR(h->child[1]);
	if (NULL != h->child[1])   h->child[1]->p = h;
	return h;
}
```

არის საჭირო იმისთვის, რომ კლასის საზიარო მეთოდმა

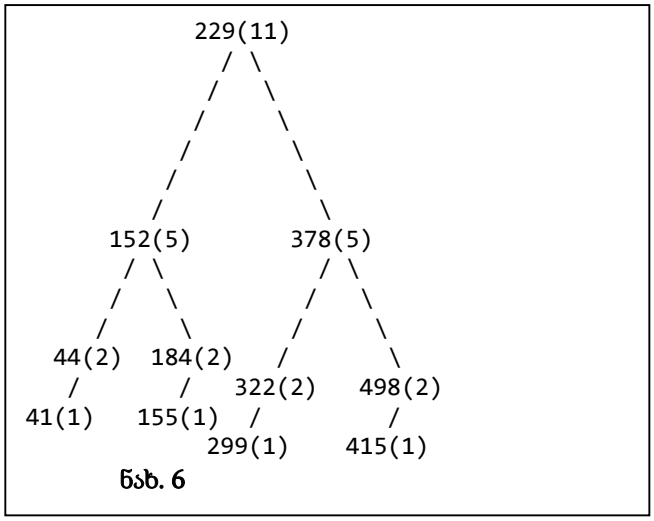
```
template<typename T>
void Tree<T>::balance()
{
	root = balanceR(root);
```

```
}
```
მოახდინოს ხის დაბალანსება. `rotate(h, 1)` არის მარჯვნივ მობრუნების ალგორითმი, ხოლო `rotate(h, 0)` - მარცხნივ მობრუნების.

შემდეგი ორი ნახაზი გვიჩვენებს თუ რა სახეს მიიღებს ნახ. 3 და 4-ზე მოყვანილი ხეები ხელახალი დაბალანსების შემდეგ (რეჟიმი ისევ OST არის):



ნახ. 5



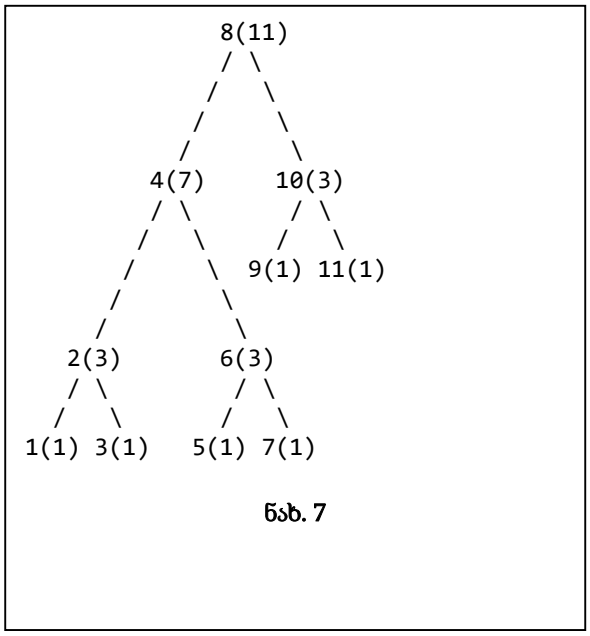
ნახ. 6

სიმაღლე ორივე შემთხვევაში ღებულობს მინიმალურ მნიშვნელობას (კვანძების მოცემული რაოდენობისთვის), მაგრამ არცერთი მათგანი არაა სრული ხე.

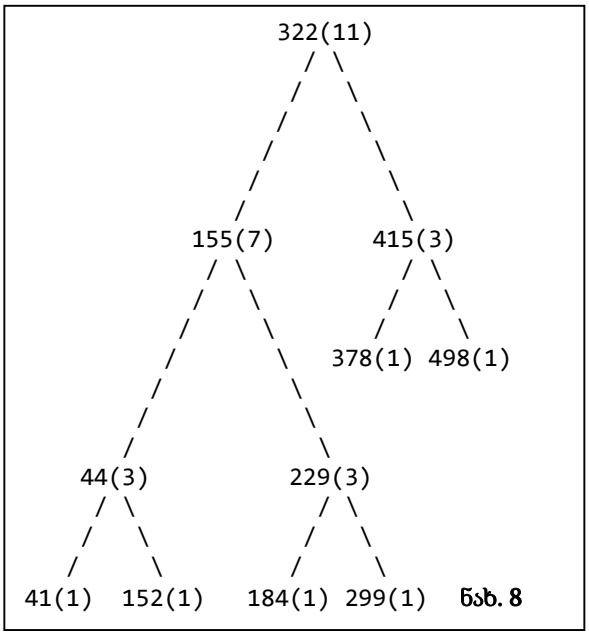
ამ მომენტამდე ჩვენ არსებითად ახალი არაფერი გვითქვამს, მხოლოდ გავავრცელეთ სეჯევიკის ალგორითმი RB-ხეზე კვანძის სტრუქტურის ცვლილების გარეშე.

RB-ხის ხელახალი დაბალანსების ახალი ალგორითმი

როგორც ზემოთ აღვნიშნეთ, ხელახალი დაბალანსების ახალ ალგორითმსა და სეჯევიკის ალგორითმს შორის იგივე დამოკიდებულებაა, რაც დეისა და DSW ალგორითმებს შორის.



ნახ. 7



ნახ. 8

მაგალითად, ნახ. 3 და 4-ზე მოყვანილი ხეებს ხელახლა დავაბალანსებთ მოდიფიცირებული ალგორითმით მივიღებთ ნახაზებს 7 და 8.

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), ფესვის ერთ-ერთი ქვეხე, ან მარცხენა, ან მარჯვენა აუცილებლად იქნება სრული. თუ რომელი მათგანია სრული, ამის გამოცნობა შესაძლებელია მარტივი გამოთვლებით.

ვთქვათ h აღნიშნავს სიმაღლეს, n-ხეში კვანძების რაოდენობას. ორობითი ხის შემთხვევაში მათ შორის მარტივი დამოკიდებულებაა: $2^h <= n < 2^{h+1}$. ასეთ ხეში კვანძების მაქსიმალური რაოდენობა არის $1 + 2 + \cdots + 2^h = 2^{h+1} - 1$.

ძებნის ორობითი ხეში ფესვის მარცხნივ განთავსებულია ნაკლები ან ტოლი, ხოლო მარჯვნივ მეტი ან ტოლი გასაღებები. ეს საშუალებას გვაძლევს ადვილად ვიანგარიშოთ სრულ ხედ დაბალანსების შემთხვევაში ფესვის რანგი (rank). თუ მარჯვენა ქვეხე არაა სრული, მაშინ მარცხენა ქვეხე იქნება სრული (მისი სიმაღლეა h-1, ელემენტების რაოდენობაა $\left(2^h - 1\right)$), ხეში ელემენტების რაოდენობა დააკმაყოფილებს უტოლობას:

$$1 + \left(2^h - 1\right) + \left(2^{h-1} - 1\right) \geq n \qquad \text{ანუ} \qquad n \leq 3 \cdot 2^{h-1} - 1.$$

ამავე შემთხვევაში სრულად დაბალანსებული ხის ფესვის რანგი იქნება $2^{h-1}+1$.

თუ $n \leq 3 \cdot 2^{h-1} - 1$ არ სრულდება, მაშინ მარჯვენ აქვეხე არის სრულად დაბალანსებული და ფესვის რანგის გამოსათვლელად ხის ელემენტების რაოდენობას უნდა დავაკლოთ მარჯვენა ქვეხის ელემენტების რაოდენობა. ამ მოსაზრებების გათვალისწინებით, სრულ ხედ დაბალანსების მოდიფიცირებულ სეჯევიკის ალგორითმს აქვს სახე:

```
template<typename T>
Node<T>* Tree<T>::balanceMod(Node<T>* h)
{
	if (h == NULL || h->bf < 2) return h;

	int height = (int)log2(h->bf);

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

		h = partR(h, h->bf - (int)pow(2, height - 1) + 1 - 1);

		h->child[0] = balanceR(h->child[0]);
		if (NULL != h->child[0])   h->child[0]->p = h;
		h->child[1] = balanceMod(h->child[1]);
		if (NULL != h->child[1])   h->child[1]->p = h;
	}
	else {

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

		h->child[0] = balanceMod(h->child[0]);
		if (NULL != h->child[0])   h->child[0]->p = h;
		h->child[1] = balanceR(h->child[1]);
		if (NULL != h->child[1])   h->child[1]->p = h;
	}

	return h;
}
```

როგორც ვხედავთ, სრული ქვეხის მხარეს ეშვება სეჯევიკის ალგორითმი, რომელიც ასეთ შემთხვევებში უნაკლოა, ხოლო მეორე მხარეს - მოდიფიცირებული ალგორითმი, რომელიც შემდეგ რეკურსიულად იგივეს გაიმეორებს.

დასასრულ, კლასის საზიარო მეთოდს აქვს სახე:

```
template<typename T>
void Tree<T>::balanceMod()
{
	root = balanceMod(root);
}
```

რიცხვითი ექსპერიმენტები

References

1. Colin Day. "Balancing a Binary Tree. **Computer Journal**, XIX (1976), pp. 360-361.
2. Robert Sedgewick. Algorithms in C, Parts 1-5. third edition; Addison-Wesley Professional, 2001.
3. Quentin F. Stout and Bette L. Warren. Tree Rebalancing in Optimal Time and Space. **Communications of the ACM**, Vol. 29, No. 9 (September 1986), p. 902-908.
4. T.Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009
5. G. M. Adel'son-Velskii and E. M. Landis. An Algorithm for the Organization of Information. *Soviet Math. Dockl.*, 1962.