თავი 7. ძებნის ორობითი ხე (BST)

განხილული საკითხები:

- განმარტებები
- ხის შემოვლის ალგორითმები >>>
- ძებნა ორობით ხეში >>>
- კვანძის დამატება და წაშლა >>>
- მომდევნო და წინა ელემენტები >>>
- დალაგებული ხეები >>>
- ლიტერატურა >>>

განმარტებები

ძებნის ორობითი ხის (binary search tree) შესწავლა განპირობებულია იმ გარემოებით, რომ ამ მონაცემთა სტრუქტურაში **ხის სიმაღლის პროპორციულ დროში** ხდება 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, მათ შორის გასაღებების ზრდის მიხედვით.

ხეში x მისამართის მქონე წვეროს, ჩვეულებრივ, მოეთხოვება რომ მასზე განსაზღვრული იყოს შემდეგი ფუნქციები:

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

როგორც ალტერნატივა, შესაძლოა ვიგულისხმოთ, რომ x მისამართის მქონე წვეროს აქვს ატრიბუტები:

`x->left` - x-ის მარცხენა შვილის მისამართი;
`x->right` - x-ის მარჯვენა შვილის მისამართი;
`x->p` - x-ის მშობლის მისამართი, რაც არის NULL ფესვისთვის;
`x->key` - გასაღების მნიშვნელობა.

ამ დროს ხის წვერო ანუ კვანძი წარმოადგენს სტრუქტურას, რომლის ობიექტებს აქვთ აღნიშნული ველები. კვანძის ასეთი აღწერა უფრო გავრცელებულია, ვიდრე ფუნქციების გამოყენებით. თუმცა, სიმარტივისთვის, ჩვენ გამოვიყენებთ ფუნქციებიან ვარიანტს.

თუ გავაძლიერებთ კვანძის სტრუქტურას (მაგალითად, დავამატებთ ზომისა ან/და "ფერის" ატრიბუტებს), მაშინ შესაძლებელი ხდება ზოგიერთი სხვა ფუნქციის ეფექტიანი განხორციელება.

ზოგადად, **ხე** (`tree`) არის წვეროებისა და წიბოების არაცარიელი სიმრავლე, რომელიც გარკვეული თვისებების მატარებელია. **წვერო** (`vertex`) (იგივე **კვანძი** -`node`) არის ობიექტი, რომელსაც აქვს სახელი, მისამართი და რომელიც თავის ველებში შეიცავდეს გარკვეულ ინფორმაციას. ამ ველებიდან გრაფიკული წარმოდგენისას ყურადღება მახვილდება მხოლოდ ერთზე, რომელსაც ეწოდება **გასაღები** (`key`) და რომლის მნიშვნელობაც გადამწყვეტია ძებნის ხის აგების დროს. **წიბო** (`edge`) არის კავშირი ორ კვანძს შორის, რომელიც გრაფიკულად მონაკვეთით გამოისახება, ხოლო პროგრამულად მას შეესაბამება მიმთითებლების ველის გარკვეული მნიშვნელობები. ხეში **გზა** (`path`) არის წვეროთა მიმდევრობა, რომელშიც ერთმანეთის მომდევნო წვეროები შეერთებულია წიბოებით. ხის დამახასიათებელი თვისება ისაა, რომ მისი ორი ნებისმიერი წვერო შეერთებულია ერთადერთი მარტივი გზით. თუ მოცემულია წიბოებისა და წვეროების რაიმე სიმრავლე, რომელშიც რომელიმე ორი წვერო შეერთებულია ერთზე მეტი მარტივი გზით, მაშინ ეს ერთობლიობა წარმოადგენს გრაფს და არა

ხეს. **ფესვიანი ხე** (rooted tree) ეწოდება ხეს, რომელშიც ერთი კვანძი გამოცხადებულია **ფესვად** (root). როგორც წესი, ფესვიან ხეს ჩვეულებრივ ხატავენ გადმობრუნებულს, ფესვით ზემოთ. გავრცელებული ტერმინოლოგიით, x კვანძი მოთავსებულია y კვანძის ზემოთ, თუ იგი ეკუთვნის ფესვისა და y კვანძის შემაერთებელ ერთადერთ გზას. ამ დროს ასევე ვამბობთ, რომ y კვანძი მოთავსებულია x კვანძის **ქვემოთ.**

ყოველი კვანძისთვის, გარდა ფესვისა, არსებობს ერთადერთი კვანძი, რომელიც მოთავსებულია მის ზემოთ და შეერთებულია მასთან წიბოთი; მას ეწოდება მოცემული კვანძის **მშობელი** (parent). კვანძებს, რომლებიც განლაგებულია მოცემული კვანძის ქვემოთ და შეერთებულია მასთან, ეწოდებათ **შვილები** (children). კვანძს, რომელსაც არა აქვს შვილები, ფოთოლს (leaf) უწოდებენ. თუ კვანძი ფოთოლი არაა, მას შიგა კვანძი ეწოდება.

ხეში კვანძები სხვადასხვა დონეზე (ანუ სიღრმეზე) არის განთავსებული. განმარტების თანახმად, ფესვის დონედ მიღებულია 0, ხოლო მისგან განსხვავებული ნებისმიერი კვანძის **დონე** (level) განიმარტება რეკურსიულად: ნებისმიერი კვანძის დონე ერთით მეტია მისი მშობელი კვანძის დონეზე. ხის **სიმაღლე** (height) არის კვანძების დონეთა შორის მაქსიმალურის ტოლი.

შემდეგი ფუნქციის გამოყენებით ვადგენთ ძებნის ხის სიმაღლეს. თუ ხე არ არსებობს, მისი სიმაღლე -1 –ის ტოლად მიიჩნევა:

```
int  ST_height(link x) {
       if (x == NULL) return -1;
       int u = ST_height(left(x));
       int v = ST_height(right(x));
       return (u < v) ? (v + 1) : (u + 1);
}
```

ამ ალგორითმის მუშაობის დრო კვანძების რაოდენობის პროპორციულია (სავარჯიშოს სახით შეამოწმეთ თუ რატომ). მისი კორექტულობის დამტკიცება ხდება მათემატიკური ინდუქციით სიმაღლის მიმართ.

მოყვანილ კოდში link აღნიშნავს წვეროს (კვანძის) მისამართის ტიპს. სიმარტივისთვის ვიგულისხმოთ, რომ link ტიპის ცვლადი წარმოადგენს პოინტერს წვეროს ობიექტზე.

ხეს ეწოდება **ორობითი ხე** (bynary tree, შემოკლებით BT), თუ მის ნებისმიერ კვანძს გააჩნია არაუმეტეს ორი შვილისა.

ძებნის **ორობითი ხე** (bynary search tree, შემოკლებით BST) ეწოდება ორობით ხეს, რომელიც აკმაყოფილებს შემდეგ ე. წ. BST –ის ძირითად თვისებას:

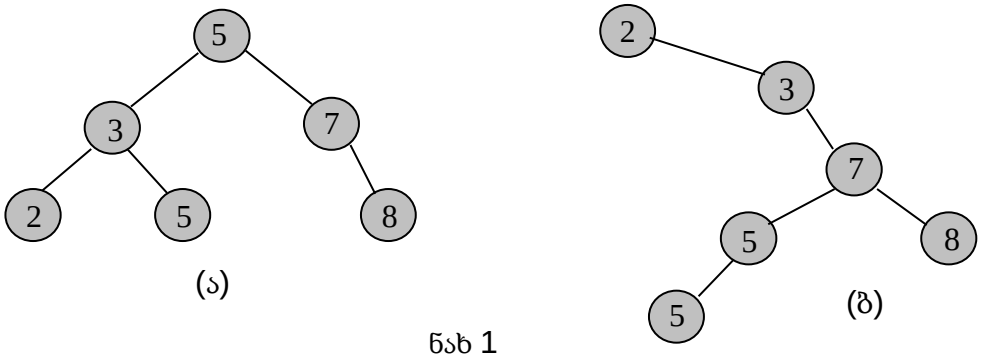
> *ვთქვათ x არის BST –ის ნებისმიერი კვანძი. თუ y მოთავსებულია x – ის მარცხენა ქვეხეში (ანუ x–ის მარცხნივ და ქვევით მოთავსებულ ხეში), მაშინ*
>
> $$\text{key}(y) \le \text{key}(x),$$
>
> *ხოლო თუ y კვანძი მოთავსებულია x– ის მარჯვენა ქვეხეში, მაშინ*
>
> $$\text{key}(y) \ge \text{key}(x),$$

სადაც key(x) – ით აღნიშნულია x კვანძის გასაღების მნიშვნელობა.

მონაცემთა ერთი და იგივე სიმრავლეს შესაძლოა შეესაბამებოდეს სხვადასხვა BST. მაგალითად, ნახ. 1 –ზე წარმოდგენილია BST–ები, რომელთა წვეროებში წერია გასაღებები. ორივე ხე შეესაბამება გასაღებების (უფრო ზუსტად კი მონაცემების) ერთი და იმავე სიმრავლეს: {2,3,5,7,8}. მაგალითად, ამ გასაღებების შემცველი ჩანაწერები შესაძლოა იყოს

გიორგი	1987 წ	10000$	189 სმ	2 სეზონი
ლაშა	1988 წ	15000$	195 სმ	3 სეზონი

და ა. შ., რომლებშიც გასაღებად მიჩნეულია ბოლო ველის მნიშვნელობა. ნახ.1 –ზე წარმოდგენილი ხეები განსხვავდებიან ფესვით, სიმაღლით და სხვა მახასიათებლებით, თუმცა მიიღებიან ერთი და იგივე მონაცემების სხვადასხვა მიმდევრობით დამუშავებისას:



ნახ 1

BT-ს მნიშვნელოვანი მათემატიკური თვისებები გააჩნია. მოვიყვანოთ ერთი მათგანი.

თვისება. N ცალი კვანძის მქონე ბინარული ხის სიმაღლე არაა ნაკლები $\log N$ -ზე და არაა მეტი $(N-1)$-ზე.

დამტკიცება. უარეს შემთხვევას წარმოადგენს წრფივ სიაში გადაგვარებული ხე, რომელსაც აქვს ერთადერთი ფოთოლი და $(N-1)$ კავშირი ფოთლიდან ფესვამდე. საუკეთესო შემთხვევაში გვაქვს (ბალანსირებული) ხე. თუ ხის სიმაღლე k - ს ტოლია, მაშინ ინდუქციის მეთოდით ადვილად, ჩანს, რომ

$$2^k \le N < 2^{k+1}, \quad k \in \{0,1,2,\ldots\}, \qquad \text{ანუ} \qquad k \le \log N < k+1, \quad k \in \{0,1,2,\ldots\},$$

რაც ნიშნავს, რომ: $\lfloor \log N \rfloor = k$. □

<<< ხის შემოვლის ალგორითმები

BST არის მონაცემთა დინამიკური სტრუქტურა. მისი ზომა ფიქსირებული არაა, კვანძები ემატება და წაიშლება საჭიროების მიხედვით (პროგრამულად ეს კეთდება მეხსიერების გამოყოფით და გათავისუფლებით). მეხსიერებაში მას არ უკავია ერთი მთლიანი ფრაგმენტი.

თუ პროგრამის გარკვეულ მომენტში მოცემული BST აღარაა საჭირო, იგი მთლიანად უნდა წავშალოთ. ამისათვის საჭიროა ხის ყოველი კვანძისთვის გამოყოფილი მეხსიერება გავათავისუფლოთ, რასაც თავის მხრივ სჭირდება ისეთი ფუნქცია (ე.წ. შემოვლის ფუნქცია), რომელიც ხის ყოველ კვანძში მხოლოდ ერთხელ მოხვედრის საშუალებას მოგვცემს. შემოვლის ფუნქციები სხვა მიზნითაც გამოიყენება, - მაგალითად გასაღებების დახარისხებისთვის ზრდადობის მიხედვით, ან ხის მარტივი ვიზუალიზაციისთვის.

განვიხილოთ სამი ასეთი ფუნქცია. მათ არგუმენტს წარმოადგენს BST – ის ფესვის მისამართი, პირობითად `root (ფესვი)`. მნიშვნელობას ეს ფუნქციები არ აბრუნებენ, ისინი მხოლოდ გაარკვეულ მოქმედებებს ასრულებენ. ერთ-ერთი მათგანი იძლევა გასაღებების დახარისხების საშუალებას.

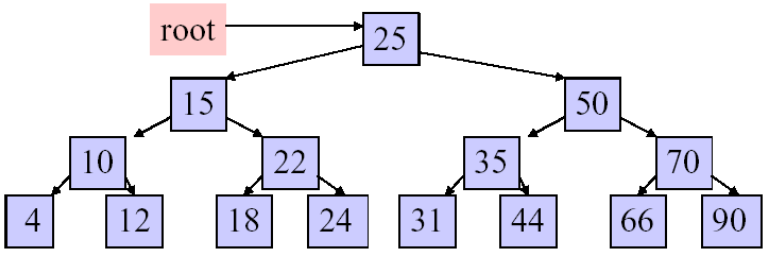
```
void ST_inorder_walk(link x)
{
  if (x != NULL) {
       ST_inorder_walk(left(x));
       x წვეროს დამუშავება;
       ST_inorder_walk(right(x));
}
```

```
void ST_preorder_walk(link x)
{
    if (x != NULL) {
        x წვეროს დამუშავება;
        ST_preorder_walk(left(x));
        ST_preorder_walk(right(x));
    }
}
```

სავარჯიშო: რა სახე ექნება ანალოგიურ **void** ST_postorder_walk (link x); ფუნქციას?

თუ დამუშავებაში ვიგულისხმებთ cout<< key(x) << endl; შეტყობინების შესრულებას, მაშინ ფუნქციები ST_inorder_walk(root), ST_preorder_walk(root), ST_postorder_walk (root), მითითებული მიმდევრობით ბეჭდავენ გასაღებებს root მისამართიანი ხისთვის (ნახ. 2):

ST inorder:	4	10	12	15	18	22	24	25	31	35	44	50	66	70	90
ST preorder:	25	15	10	4	12	22	18	24	50	35	31	44	70	66	90
ST postorder:	4	12	10	18	24	22	15	31	44	35	66	90	70	50	25



ნახ.2

მათემატიკური ინდუქციის გამოყენებით დავამტკიცოთ, რომ ფუნქცია ST_inorder გამობეჭდავს ზრდადობის მიხედვით დახარისხებულ გასაღებებს. მართლაც, თუ ხის სიმაღლე ნულია (ანუ მხოლოდ ერთი კვანძია), ფუნქცია ბეჭდავს გასაღებს. ვთქვათ ფუნქცია ზრდადობით ახარისხებს m სიმაღლის მქონე ნებისმიერ BST – ს და მოცემულია m+1 სიმაღლის ხე. ოპერაციების მიმდევრობა:

```
ST_inorder_walk(left(root));
cout << key(x) << endl;
ST_inorder_walk(right(root));
```

ინდუქციური დაშვების გათვალისწინებით გვაძლევს, რომ ჯერ დაიბეჭდება ფესვის გასაღების მარცხენა ქვეხის (სიმაღლე <= m) გასაღებები, დახარისხებული ზრდადობით, შემდეგ ფესვის გასაღები და შემდეგ მარჯვენა ქვეხის (სიმაღლე<= m) გასაღებები, კვლავ დახარისხებული ზრდადობით. რადგან ფესვის გასაღები მეტია ან ტოლი მარცხენა ქვეხის ყველა გასაღებზე და ნაკლებია ან ტოლი მარჯვენა ქვეხის ყველა გასაღებზე, ამიტომ m+1 სიმაღლის ხის გასაღებებიც დაიბეჭდება ზრდადობის მიხედვით დახარისხებული.

განხილული ფუნქციების მუშაობის დრო წრფივადაა დამოკიდებული კვანძების რაოდენობაზე (სხვა სიტყვებით, არის $O(n)$, სადაც n არის ხეში გასაღებების რაოდენობა), რადგან თითოეულ კვანძზე იხარჯება ფიქსირებული დრო (გარდა რეკურსიული გამოძახებებისა) და თითოეული კვანძი მუშავდება მხოლოდ ერთხელ.

<<< ძებნა ორობით ხეში

შემდეგი ფუნქციები: ST_search, ST_minimum, ST_maximum ახორციელებენ სხვადასხვა სახის ძებნის ოპერაციებს და მათ არგუმენტს წარმოადგენს ან საძიებელი გასაღები k, ან

BST – ის ფესვის მისამართი x, ან ორივე; ხოლო მნიშვნელობას– განსაზღვრული თვისებების მქონე კვანძის მისამართი. თითოეული მათგანის შესრულების დრო არის $O(h)$, სადაც h არის ხის სიმაღლე, რადგან მოძრაობა ხდება მხოლოდ ერთი გზის გასწვრივ და ამ გზაზე თითო კვანძი დამუშავდება ერთხელ.

ძებნა. ფუნქცია

```
link ST_search(link x, int k)
{
   if (NULL == x || k == key(x)) return x;
   if (k < key(x)) return ST_search(left(x), k);
   return  ST_search(right(x), k);
}
```

პარამეტრებად იღებს წვეროს მისამართს და საძებნ გასაღებს; ძებნა ხორციელდება იმ ხეში, რომლისთვისაც ფესვს წარმოადგენს არგუმენტად გადაცემული წვერო; ფუნქცია აბრუნებს მეორე არგუმენტად გადაცემული გასაღების მქონე კვანძის მისამართს ან NULL-ს (როცა ასეთი კვანძი არაა). ძებნის პროცესში ვმოძრაობთ ფესვიდან ქვემოთ და თან ვადარებთ საძიებ k გასაღებს მიმდინარე კვანძის გასაღებს. თუ ისინი ტოლია, ძებნა დასრულდება, თუ არა – მოძრაობა გრძელდება BST –ის ძირითადი თვისების გათვალისწინებით.

მინიმუმი და მაქსიმუმი. ცხადია, BST –ში მინიმალური გასაღების მოსაძებნად საკმარისია ფესვიდან სულ მარცხენა შვილების მხარეს მოძრაობა:

```
link ST_minimum(link x) {
     while (left(x) != NULL) {
       x = left(x);
     }
     return x;
 }
```

ხოლო მაქსიმუმის ძებნის ფუნქცია მისი სიმეტრიულია:

```
link  ST_maximum(link x) {
       while (right(x) != NULL) {
             x = right(x);
       }
       return x;
 }
```

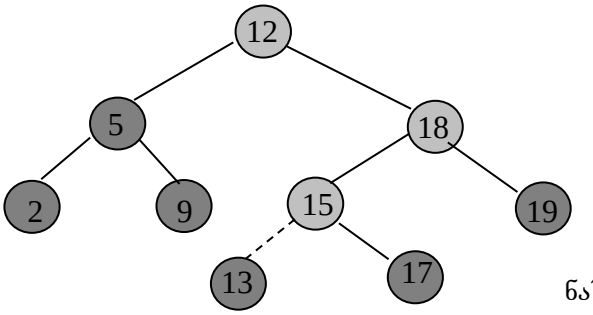
<<< კვანძის ჩამატება და წაშლა

ჩამატება. განვიხილოთ, თუ როგორ მუშაობს ჩამატების ფუნქცია, რომელიც პარამეტრებად იღებს ხის ფესვის მისამართსა და ჩასამატებელი კვანძის მისამართს:

```
link ST_insert(link root, link z){
1        link x,y;
2        y = NULL;
3        x = root;
4        while ( NULL != x ) {
5            y = x;
6            if( key(z) < key(x))  x = left(x);
7            else  x= right(x);
8        }
9        p(z) = y;
10       if (NULL == y) root = z;
11       else  {
12           if   (key(z) <  key(y))  left(y) = z;
13           else  right(y) = z;
14       }
15
```

```
        return root;
    }
```

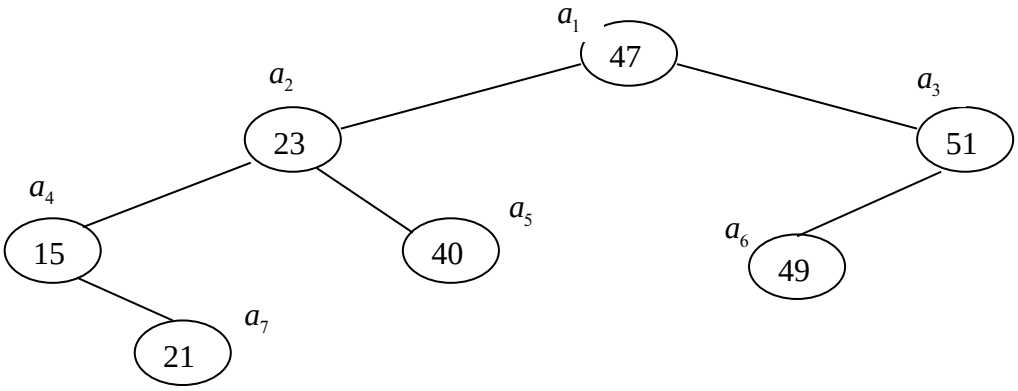
ეს ფუნქციაც მოძრაობს ზემოდან ქვემოთ, მისი ფესვიდან დაწყებული. ამასთან *y* ინახავს *x* კვანძის მშობლის მისამართს (4–7 სტრიქონებში). `key(z)` და `key(x)` –ის შედარების საფუძველზე ფუნქცია წყვეტს თუ რომელ მხარეს წავიდეს, მარცხნივ თუ მარჯვნივ. პროცესი მთავრდება, როდესაც *x* ხდება `NULL` მისამართი. ეს `NULL` ზუსტად იქ დგას, სადაც უნდა ჩაისვას *z* კვანძი, რაც კეთდება სტრიქონებში 9 – 13.



ნახ. 3

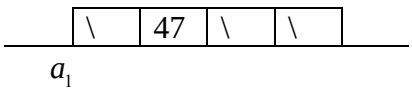
ნახ. 3 –ზე ნაჩვენებია 13–ის ტოლი გასაღების მქონე კვანძის დამატება. ღია ფერით მონიშნული კვანძები მოთავსებულია გზაზე ფესვიდან ჩასამატებელი კვანძის პოზიციამდე. პუნქტირი ძველ ელემენტს აკავშირებს ახალთან.

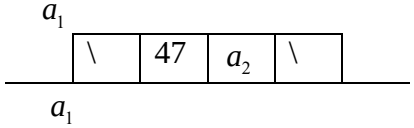
მაგრამ ასეთი გრაფიკული წარმოდგენა ძალიან გამარტივებულია და მხოლოდ ალგორითმის უკეთ წარმოდგენას ემსახურება. რეალურად მეხსიერებაში, ჩასასმელი კვანძის გასაღების მიმდინარე კვანძის გასაღებთან შედარების შემდეგ, არც უფრო მეტი გასაღების მქონე კვანძი მიდის მარჯვნივ და არც უფრო მცირესი მიდის მარცხნივ. ახალი კვანძისთვის მეხსიერებას თვითონ სისტემა გამოყოფს, რისთვისაც გამოიყენება მეხსიერების განაწილების ალგორითმები. მაგალითად, თუ ვიგულისხმოთ რომ ახალი `x` წვეროს შექმნისას მეხსიერება გამოიყოფა შემდეგი ატრიბუტებისთვის: `p(x)`- მშობლის მისამართი, `key(x)` -გასაღები, `left(x)`- `x`-ის მარცხენა შვილის მისამართი, `right(x)`- `x`-ის მარჯვენა შვილის მისამართი,- ამავე თანმიმდევრობით, და ვნახოთ როგორ აიგება ხე რიცხვების მიმდევრობისთვის 47, 23, 51, 15, 40, 49, 21. ვიგულისხმოთ, რომ წვეროებისთვის

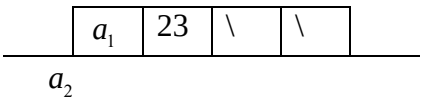


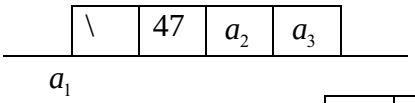
გამოყოფილი მეხსიერების მისამართები აღინიშნება a_1, a_2 და ა. შ.. ხის სახეა

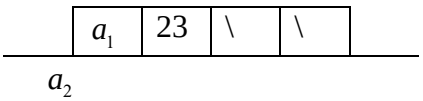
თუმცა რეალური სურათი სხვანაირია:

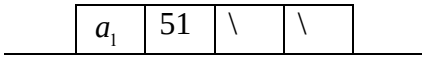
1)


2).




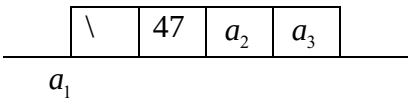
3)


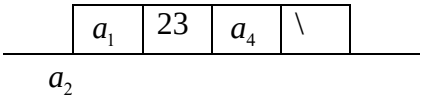


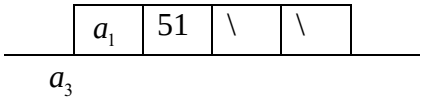


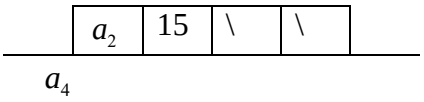

a_3

4)

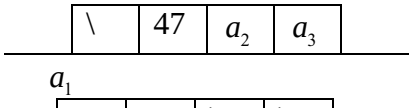

\
47
a_2
a_3
a_1

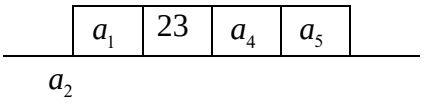

a_1
23
a_4
\
a_2

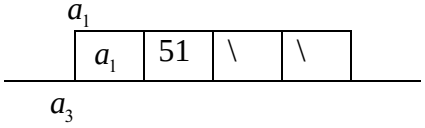

a_1
51
\
\
a_3

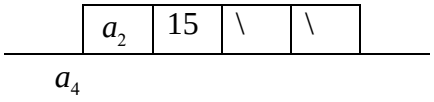

a_2
15
\
\
a_4

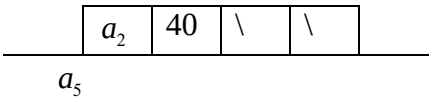
5)


\
47
a_2
a_3
a_1

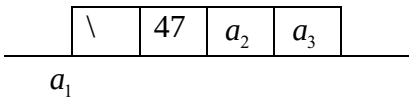

a_1
23
a_4
a_5
a_2

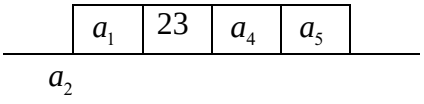

a_1
51
\
\
a_3

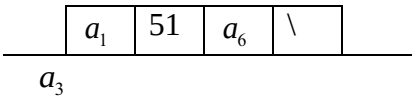

a_2
15
\
\
a_4

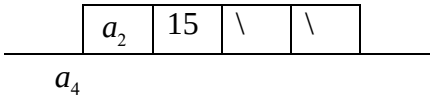

a_2
40
\
\
a_5

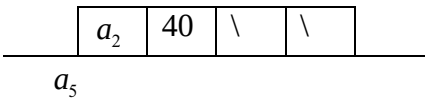
6)

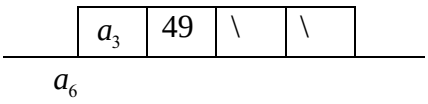

\
47
a_2
a_3
a_1


a_1
23
a_4
a_5
a_2

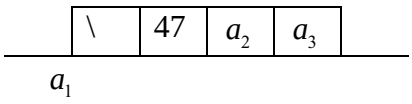

a_1
51
a_6
\
a_3

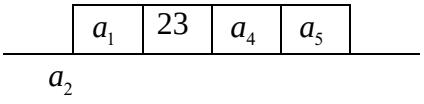

a_2
15
\
\
a_4

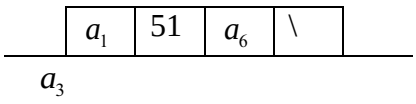

a_2
40
\
\
a_5

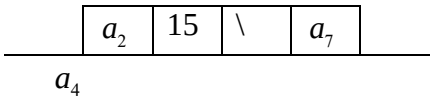

a_3
49
\
\
a_6

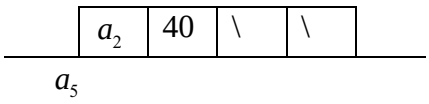
7)

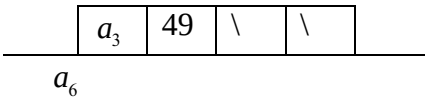

\
47
a_2
a_3
a_1

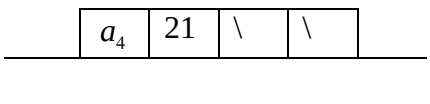

a_1
23
a_4
a_5
a_2


a_1
51
a_6
\
a_3


a_2
15
\
a_7
a_4


a_2
40
\
\
a_5


a_3
49
\
\
a_6


a_4
21
\
\

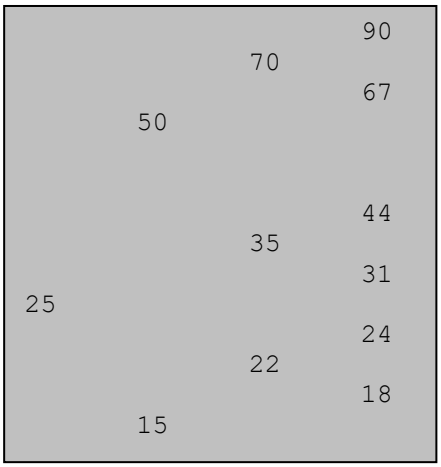
a_7

წაშლა. ფუნქცია რომელიც შლის კვანძს ხეში, პარამეტრად იღებს ხისა და წასაშლელი კვანძის მისამართებს. წაშლის ფუნქციის ალგორითმი საკმაოდ რთულია, თუ მოვინდომებთ მის იმპლემენტირებას ისე, რომ რომ წაშლის დროს მხოლოდ წასაშლელი კვანძის იტერატორი გაუქმდეს. ამიტომ ჩვენ ამ საკითხს არ ვეხებით საბაკალავრო პროგრამაში.

<<< მომდევნო და წინა ელემენტები

ხშირად (მაგალითად ხიდან ელემენტის წაშლისას) აუცილებელია BST –ში მოცემული კვანძის მომდევნო კვანძის (გასაღებების ზრდის თვალსაზრისით) მოძებნა. შემდეგი ფუნქცია აბრუნებს მოცემული x კვანძის მომდევნო კვანძის მისამართს:

```
link ST_successor(link x)
{
	if (NULL != right(x)) return ST_minimum(right(x));
	while (NULL != p(x) && x == right(p(x)))
	{
		x = p(x);
	}
	return p(x);
}
```



ფუნქცია განიხილავს ორ შემთხვევას. თუ x–ის მარჯვენა ქვეხე არაცარიელია, მაშინ ამ ქვეხის მინიმალური ელემენტი არის x–ის მომდევნო. მაგალითად, ამ ხეზე 50–ის ტოლი გასაღების მქონე კვანძის მომდევნო არის 67 –ის ტოლი გასაღების მქონე კვანძი.

თუ x–ის მარჯვენა ქვეხე ცარიელია. ახლა ჩვენ ვმოძრაობთ x –იდან ზემოთ, ვიდრე არ ვიპოვით კვანძს, რომელიც წარმოადგენს თავისი მშობლის მარცხენა შვილს. ეს მშობელი არის მომდევნო. მაგალითად, ამ ხეზე 24 -ის ტოლი გასაღების მქონე კვანძის მომდევნო არის 25 –ის ტოლი გასაღების მქონე კვანძი.

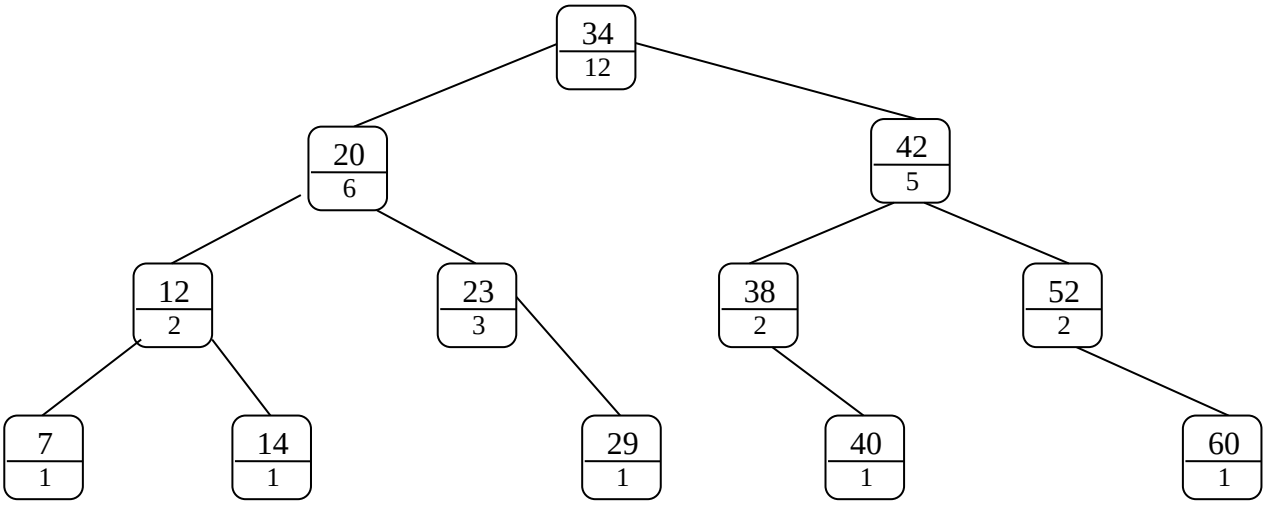
წინა კვანძის განსაზღვრა ანალოგიურად ხდება.

<<< დალაგებული ხეები

ძებნის ორობითი ხეების გამოყენების დიაპაზონი ძალზე ვრცელია. იგი ეფექტურია სრულიად განსხვავებული ხასიათის მქონე ამოცანების გადაწყვეტისას. ახლა განვიხილოთ BST- ის ერთი განზოგადება, რომელსაც ეწოდება დალაგებული ხე (შემოკლებით OST, ანუ order-statistic tree). ასეთ ხეში, გარდა ზემოთ განხილული ფუნქციებისა, შესაძლებელია მარტივად გადაიჭრას შემდეგი ორი ამოცანა:

1. n ელემენტისაგან შედგენილ სიმრავლეში რიგით i- ური კვანძის (გასაღების სიდიდის მიხედვით) განსაზღვრა;
2. მოცემული კვანძის რიგითი ნომრის განსაზღვრა (წინას შებრუნებული ამოცანა, რომლის იმპლემენტაციისთვის საჭიროა აგრეთვე წინაპარი კვანძის დასამახსოვრებელი ველი);

თუმცა მეორე ამოცანის გადაჭრისთვის საჭიროა ყოველი კვანძისთვის განსაზღვრული იყოს მშობლის ველი.



ნახაზზე ნაჩვენებია დალაგებული ხე. მის კვანძს გარდა ჩვეული `key(x)`, `left(x)`, `right(x)` ველებისა 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, რომლებსაც 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.

სიდიდით i-ური გასაღების განსაზღვრა. ალგორითმს ვაგებთ იგივე იდეის გამოყენებით, რის საფუძველზეც ვახორციელებდით ძებნას ორობით ხეში. შესაბამის ფუნქციას აქვს სახე:

```
  link OST_select(x,i){
1    int r = size(left(x)) + 1;
2    if (i == r)  return x;
3    if (i < r)  return OST_select(left(x), i);
4    return OST_select(right(x), i - r);
  }
```

ფუნქცია პარამეტრებად იღებს ხის ფესვის მისამართს `x` და `i` ნომერს. შედეგად აბრუნებს იმ კვანძის მისამართს, რომელშიც მოთავსებულია სიდიდით i-ური გასაღები. სწორად დასმულ ამოცანაში, როდესაც $1 \le i \le \text{size(x)}$, ალგორითმის სამართლიანობა ადვილად ჩანს მათემატიკური ინდუქციის მეთოდის გამოყენებით ქვეხეთა ზომის მიმართ. თუ ხე ერთადერთი კვანძისგან შედგება, მაშინ ცხადია, რომ `OST_select(x,1)` ტოლია `x` –ის. ვთქვათ, ფუნქცია სწორად მუშაობს როცა ხის ზომაა m და ვაჩვენოთ რომ სწორ შედეგს მოგვცემს როცა `size(x)=` $m+1$. რადგან ფესვის ორივე ქვეხის ზომა $\le m$, ამიტომ ფუნქცია დააბრუნებს სწორ პასუხს, იმის გათვალისწინებით, რომ თუ `i < r`, მაშინ საწყის ხეში სიდიდით i- ური გასაღები არის მარჯვენა ქვეხეში სიდიდით `(i-r)` გასაღები.

ყოველი რეკურსიული გამოძახების დროს ხდება დაშვება ხეზე ერთი დონით, ამიტომ ფუნქციის მუშაობის დრო არის $O(h)$, სადაც h არის ხის სიმაღლე.

მოცემული კვანძის რიგითი ნომრის განსაზღვრა. ფუნქცია `OST_rank(root,x)` პარამეტრებად ხის მისამართის გარდა იღებს ამ ხის ერთერთი კვანძის მისამართს და განსაზღვრავს მის (ანუ მისი გასაღების რიგით) ნომერს.

```
int OST_rank(root, x){
1    int r = size(left(x)) + 1;
2    link y = x;
3    while (y != root) {
4        if (y = right(p(y)))  r = r + size(left(p(y))) + 1;
5        y = p(y);
6    }
7    return r;
  }
```

ჯერ ხდება `x` –ის რიგითი ნომრის განსაზღვრა იმ ქვეხეში, რომლის ფესვს თვითონ წარმოადგენს. ეს ნომერი არის პირველ სტრიქონში განსაზღვრული რიცხვი. შემდეგ სტრიქონში ხდება ამ მისამართის დამახსოვრება `y` –ში, ხოლო ციკლი (სტრიქონები 3-5) ახდენს პირველად განხილული ქვეხის თანდათან ამაღლებას `y` –იდან მის მშობელზე თანდათანობითი გადასვლის გზით, ვიდრე არ მივაღწევთ `root` ფესვს. თითოეულ ამ ხეში ხდება `x` –ის რიგითი ნომრის ხელახალი გამოთვლა იმ პრინციპით, რომ როცა `y` მარცხენა შვილია, ფესვის მშობელში გადატანით `x` – ის რიგითი ნომერი არ იცვლება, ხოლო როცა `y` მარჯვენა შვილია, ფესვის მშობელში გადატანით `x` –ის რიგითი ნომერი იცვლება ჩვენთვის უკვე ნაცნობი და მეოთხე სტრიქონში გამოყენებული ფორმულით. ამ ფუნქციის მუშაობის დროც არის $O(h)$, სადაც h რის ხის სიმაღლე.

<<< ლიტერატურა

1. T.Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009
2. R. Sedgewick. Algorithms in C, Parts 1-5. 2001.