ივანე ჯავახიშვილის სახელობის თბილისის

სახელმწიფო უნივერსიტეტი

ვახტანგი ლალუაშვილი

InMemory B-ხის იმპლემენტაცია

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერება

ნაშრომი შესრულებულია კომპიუტერული მეცნიერების მაგისტრის
ნაშრომის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი, 2017

ანოტაცია

B-ხე ცნობილი მონაცემთა სტრუქტურაა, რომელიც ფართოდ გამოიყენება თანამედროვე სისტემებში, სადაც ოპერაციები სრულდება დიდ მონაცემებთა ბლოკებზე. აქედან გამომდინარე, B-ხის გაუმჯობესება მნიშვნელოვან ამოცანას წარმოადგენს, რომელიც დიდი მონაცემების სწრაფად დამუშავების შესაძლებლობას მოგვცემს.

ამ ნაშრომში განხილულია B-ხის რამდენიმე იმპლემენტაცია, რომლებიც იყენებენ შიგა მეხსიერების სპეციფიკას. წარმადობის ცვლილება შესწავლილია ექსპერიმენტულად შესაბამისი ტესტების საფუძველზე.

Abstract

B-tree is a well known data structure, which is widely used in modern systems where the operations are executed on big blocks of data. Hence, improvement of B-tree structure is an important task, which will give us possibility to process huge number of data more quickly.

Bellow, in this work, we discuss B-tree implementations, which use in-memory features. Efficiency difference is analyzed experimentally by relying on the corresponding tests.

სარჩევი

შესავალი 4

B-ხეები 5

B-ხე კომპარატორით 18

B-ხე წრიული დალაგებული მასივით 20

B-ხე წყვილების წრიული დალაგებული მასივით 23

პროექტის სტრუქტურა 25

შეფასება 27

ლიტერატურა 29

შესავალი

მონაცემთა სტრუქტურებში ხეების მნიშვნელოვან ნაირსახეობას წარმოადგენს თვითბალანსირებადი ხეები. ბალანსირებულ ხეში ელემენტზე წვდომა უფრო სწრაფად ხდება ვიდრე არაბალანსირებულში.

თვითბალანსირებადი ორობითი ხეების ერთ-ერთი ცნობილი ტიპია B-ხე. მის კვანძს აქვს შესაძლებლობა შეინახოს ერთზე მეტი გასაღები. ასევე, ხის ფესვიდან ფოთლამდე მანძილი ყველა ფოთლისთვის ერთი და იგივეა. ხის სიმაღლეში გაზრდის სიხშირე დამოკიდებულია ხეში განსაზღვრულ t პარამეტრზე.

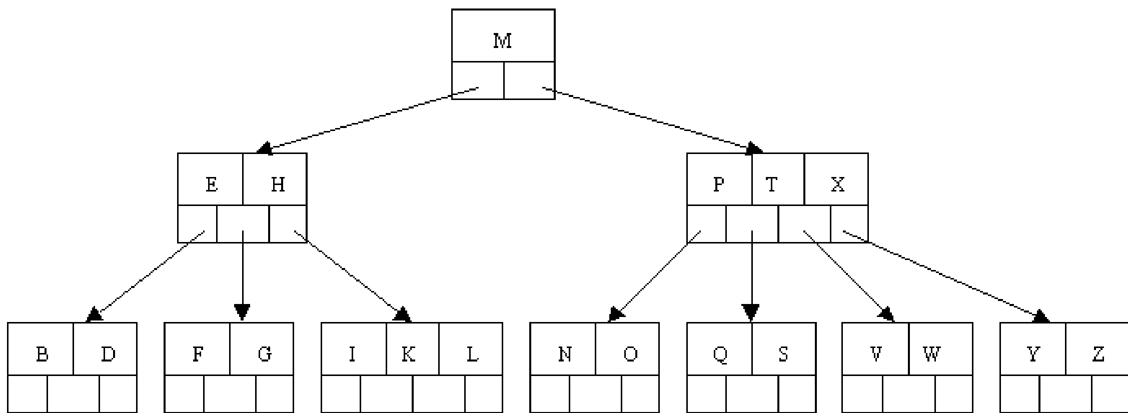
B-ხე ფართოდ გამოიყენება ფაილურ სისტემებში, მონაცემთა ბაზებში და სხვა სისტემებში. აქედან გამომდინარე, B-ხის იმპლემენტაციის დახვეწა და გაუმჯობესება აქტუალურ პრობლემას წარმოადგენს.

რადგან ამჟამად in-memory ხისებრი სტრუქტურებიდან B-ხე ყველაზე უკეთ ექვემდებარება გაპარალელებას, ამიტომ მისი უკეთ შესწავლა და რამდენიმე ძირითადი ვარიაციის იმპლემენტირება აგრეთვე აქტუალურია.

წარმოდგენილი ნაშრომი ამ საკითხს სწავლობს. იგი შედგება შესავალისა და ხუთი პარაგრაფისგან. აგრეთვე მოყვანილია მოკლე შეფასება და ლიტერატურის ნუსხა.

B-ხეები

B-ხე არის თვით-ბალანსირებადი ხისებრი მონაცემთა სტრუქტურა, რომელიც მონაცემებს დახარისხებულად ინახავს. მისი უპირატესობა იმაშია, რომ იგი ასრულებს ძებნის, ჩასმისა და წაშლის ოპერაციებს ლოგარითმულ დროში. B-ხე ძებნის ორობითი ხის განზოგადებაა, რაც გულისხმობს იმას რომ, ძებნის ორობითი ხისაგან განსხვავებით, B-ხის წვეროს შეიძლება ჰყავდეს ორზე მეტი შვილი. B-ხე ოპტიმალურია ისეთი სისტემებისთვის რომლებიც წაკითხვა-ჩაწერის ოპერაციებს არსულებენ დიდ მონაცემებზე. ამიტომაც, იგი ხშირად გვხვდება მონაცემთა ბაზებში და ფაილურ სისტემებში.



სურათი 1.

სურათ 1-ზე ნაჩვენებია B-ხე, რომლის გასაღებებსაც ინგლისური ენის სიმბოლოები წარმოადგენს. სხვა მონახემთა სტრუქტურების მსგავსად B-ხესაც გააჩნია თავისი წესები. თუ B-ხის შიდა კვანძი x შეიცავს x.n გასაღებს, მაშინ მას ჰყავს x.n + 1 შვილი. ყოველი ორი მეზობელი გასაღბი x კვანძში, წარმოადგენს მინიმუმს და მაქსიმუმს მათ შორის მოთვსებული შვილის ყველა გასაღებისთვის. გასაღბის ძებნისას ხეში ჩვენ ვაკეთბთ არჩევას x.n + 1 გზიდან ერთის ამოსარჩევად, x კვანძში მოთავსებულ გასაღებებთან საძიებელი გასაღების შედარების საფუძველზე. B-ხეში კვანძებს აქვს გასაღებების რაოდენობის ქვედა და ზედა ზღვარი. ეს ზღვარი გამოიხატება ერთი მუდმივი რიცხვის საშუალებით: t >= 2, რომელსაც ვუწოდებთ B-ხის მინიმალურ ხარისხს. თითოეული კვანძი, გარდა

ფესვისა, უნდა შეიცავდეს მინიმუმ t-1 გასაღებს. თითოეულ შიდა კვანძს, ფოთლების გარდა, უნდა ჰყავდეს t შვილი. თუ ხე ცარიელი არაა, მაშინ ფესვის უნდა შეიცავდეს მინიმუმ 1 გასაღებს. ხოლო რაც შეეხება ზედა ზღვარს, თითოეული კვანი შეიძლება შეიცავდეს მაქსიმუმ 2t-1 გასაღებს. აქედან გამომდინარე, შიდა კვანძს შეიძლება ჰყავდეს მაქსიმუმ 2t შვილი. ჩვენ ვამბობ რომ კვანძი სავსეა თუ იგი შეიცავს ზუსტად 2t-1 გასაღებს. ყველა ფოთოლს კი აქვს ერთი და იგივე სიღრმე, რომელიც არის ხის სიმაღლე h.

ჩანს, რომ B-ხე ხდება უმარტივესი თუ t=2. თითოეული შიდა კვანძს ამ შემთხვევაში ჰყავს 2, 3 ან 4 შვილი და ჩვენ გვაქვს ე.წ. 2-3-4 ხე. პრაქტიკულად, რაც უფრო დიდია t მით უფრო პატარაა B-ის სიმაღლე.

განვიხილოთ B-ხის მარტივი იმპლემენტაცია (რომელიც სიმარტივისთვის კვანძებში მთელ რიცხვებს ინახავს). იმპლემენტაციის კოდი შეგიძლიათ იხილოთ [1] ბმულზე. B-ხე წარმოდგენილია ერთ კლასში, ხოლო მისი კვანძები კი მეორე. ოპერაციები სრულდება ხის კლასზე. სანამ ამ ოპერაციებს განვიხილავთ, ვთქვათ თუ რა პარამეტრებს შეიცავს B-ხის კლასი: გვაქვს გასაღებების მასივი, შვილების მასივი, კვანძში გასაღებების რაოდენობის აღმნიშვნელი ცვლადი, t პარამეტრი და ბულის ცვლადი, რომელიც გვეუბნება კვანძი ფოთოლია თუ არა. B-ხის წესების თანახმად, შვილების მასივის ზომა ერთით მეტია გასაღებების მასივის ზომაზე. კვანძის შექმნისას გასაღებების და შვილების მასივის ზომა მაქსიმალურია, რათა თავიდან ავიცილოთ მასივის წაშლისა და შექმნის განმეორებადი ოპერაციები. შვილების მასივში i-ური ინდექსზე არის გასაღებების მასივში არსებული i-ური გასაღების მარცხენა შვილი (შესაბამისად (i + 1) იქნება i-ურის მარჯვენა და ამასთანავე (i + 1)-ის მარცხენა შვილი). ხოლო, სულ ბოლო (N + 1)-ე იქნება მე-N-ეს მარჯვენა შვილი.

B-ხისა და კვანძის კლასები:

```
class BTree
{
    BTreeNode *root;     // პოინტერი ფესვის კვანძზე
```

```
    int t;              // t პარამეტრი
    ...
}
class BTreeNode
{
    int *keys;          // გასაღებების მასივი
    int t;              // t პარამეტრი
    BTreeNode **C;      // შვილების პოინტერების მასივი
    int n;              // მიმდინარე კვანძში გასაღებების რაოდენობა
    bool leaf;          // გვეუბნება ფოთოლია თუ არა მიმდინარე კვანძი
    ...
}
```

ის ოპერაციები, რომლებიც გვაინტერესებს და რომლებსაც განვიხილავთ არის: ჩამატება, წაშლა და ძებნა. როგორც უკვე აღვნიშნეთ, ეს ოპერაციები სრულდება ლოგარითმულ დროში. სიმარტივისათვის, ჩვენ განვიხილავთ ფსევდო-კოდს.

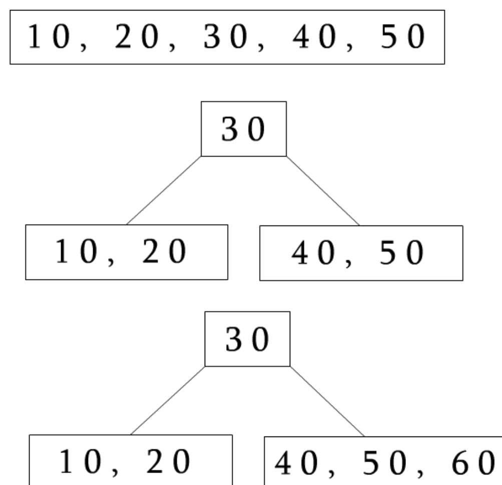
განვიხილოთ რაიმე k გასაღების ჩამატების ფსევდო-კოდი (შევნიშნოთ, რომ ეს ალგორითმი მიყვება [2] კორმენის წიგნში განხილულ ალგორითმს):

1. x გასაღები გახადე ფესვი.
2. სანამ x არ არის ფოთოლი, შეასრულე შემდეგი:
    a. იპოვე x-ის შვილი, რომელსაც შემდეგ განვიხილავთ. იყოს ეს შვილი y-ი.
    b. თუ y არ არის სავსე, მაშინ x მიუთითებდეს y-ს.
    c. თუ y სავსეა, მაშინ „გაყე“ y კვანძი და x მიუთითებდეს გაყოფილებიდან ერთ-ერთს: თუ k არის y-ის შუა გასაღებზე ნაკლები, მაშინ x მიუთითებდეს გაყოფილ მარცხენა ნაწილს, თუ არადა მარჯვენას. y-ის გაყოფის შემდეგ მისი შუა გასაღები ადის მშობელ კვანძში.

3. მეორე ციკლი ჩერდება მაშინ როდესაც x არის ფოთოლი. x-ს აუცილებლად ექნება ერთი ცარიელი ადგილი ახალი კვანძის ჩასამატებლად, რადგანაც ჩვენ აუცილებლობის შემთხვევაში კვანძებს წინასწარ ვყოფდით. ამიტომაც, ჩავსვამთ k-ს გასაღებს x-ში.

ასეთ ჩასმას ეწოდება პროაქტიული ჩასმის ალგორითმი, სადაც ქვედა კვანძში ჩასვლის წინ ჩვენ ვასრულებთ გაყოფას თუ იგი სავსეა. უპირადესობა წინასწარ გაყოფაში არის ის რომ ჩვენ არასოდეს გავივლით კვანძს ორჯერ. თუ ჩვენ არ გავყოფთ კვანძს ჩასვლამდე და გავყოფთ მას მხოლოდ მაშინ თუ ახალი გასაღები დაემატება (ამას ეწოდება რეაქტიული ალგორითმი), ჩვენ შეგვიძლია მოგვიწიოს ფოთლიდან ფესვამდე გზის კიდევ ერთხელ გავლა. ეს ხდება იმ შემთხვევაში, თუ ფესვიდან ფოთლამდე გზაზე ყველა კვანძი სავსეა. თუ ფოთლამდე მივედით და ის სავსეა, მაშინ გაყოფა მოგვიწევს, რომელიც მშობელ კვანძში აიტანს შუა ელემენტს, მაგრამ მშობელიც სავსეა და მისი გაყოფაც მოგვიწევს და ა. შ. ფესვის ჩათვლით. ეს კასკადური ეფექტი პროაქტიულ ალგორითმში არ გვხვდება, თუმცა ამ ბოლოსაც თავისი მინუსი გააჩნია. შეიძლება არასაჭირო გაყოფები შესრულდეს.

განვიხილოთ ჩასმის სცენარი:



სურათზე გამოსახულია სავსე კვანძი, რომელშიც ხდება 60 გასაღების ჩასმა, რაც მოითხოვს კვანძის გაყოფას ჩამატებამდე (იგულისხმება რომ t არის 3-ის ტოლი).

კოდში ჩასმისას ჩვენ ვიყენებთ სამ მთავარ ფუნქციას:

1. **void** insert(**int** k)
2. **void** insertNonFull(**int** k)
3. **void** splitChild(**int** i, **BTreeNode** *y)

ჩასმისას ვიძახებთ პირველ ფუნქციას, რომელიც შემდგომ იყენებს დანარჩენ ორს. insertNonFull მოიაზრებს, რომ მისი გამოძახების დროს კვანძი არ არის სავსე. თუ ფოთოლში ხდება ჩამატება, მაშინ ეძებს პოზიციას სადაც უნდა ჩაამატოს ახალი გასაღები და მას ამატებს (ასრულებს წაძვრებს გასაღებების მასივში თუ ეს საჭიროა). ხოლო, თუ ჩამატება ხდება არა ფოთოლში, მაშინ იგი ეძებს შვილს რომელშიც უნდა მოხდეს გასაღების ჩასმა. ჩასვლამდე იგი ამოწმებს თუ ეს შვილი სავსეა და თუ იგი არის სავსე, მაშინ ასრულებს მესამე ფუნქციას, splitChild-ს.

აქ პოზიციის ძებნა და წაძვრა სრულდება ერთდროულად while ციკლში:

```
int i = n - 1;
...
while (i >= 0 && keys[i] > k) // ეძებს პოზიციას და აგრეთვე წევს დიდ ელემენტებს
{
        keys[i + 1] = keys[i];
        i--;
}
```

გაყოფის ოპერაციის შემდეგ, იმ კვანძის შუა ელემენტი რომელიც გაიყო ამოდის მიმდინარე კვანძის i-ურ პოზიციაზე. ვიღებთ ორ ახალ შვილ კვანძებს და შესაბამისად ხელახლა ვამოწმებთ თუ რომელ კვანძში უნდა ჩაიწეროს ახალი გასაღები:

```
if (C[i + 1]->n == 2 * t - 1) // სისავსის შემოწმება
{
        splitChild(i + 1, C[i + 1]); // შვილის გაყოფა
        if (keys[i + 1] < k) // ვარკვევთ, სად უნდა ჩავსვათ გასაღები
```

```
            i++;
        C[i + 1]->insertNonFull(k); // გასაღების ჩასმა შესაბამის შვილში
}
```

აუცილებელია, რომ splitChild ოპერაციის გამოძახებისას კვანძი სავსე იყოს. ფუნქციას გადაეცემა ინდექსი სადაც შუა გასაღბი უნდა მოთავსდეს. რაც შეეხება შვილებს, მათი ადგილებიც ცნობილია: i-ურზე იქნება მარცხენა ნაწილი, ხოლო მის მარჯვნივ მარჯვენა. არსებული i-ურ შვილის მარჯვენა ნაწილს ვაკოპირებთ ახალ კვანძში, რომელიც ოპერაციის ბოლოს გახდება მიმდინარე კვანძის (i + 1)-ე შვილი. i-ურ შვილს კი ვამცირებთ. ორივე კვანძის ზომა, რადგან ვიცით რომ კვანძი სავსე იყო, იქნება (t – 1) -ის ტოლი.

splitChild ფუნქციის ზოგიერთი საკვანძო ფრაგმენტი:

```
BTreeNode *z = new OldBTreeNode(y->t, y->leaf); // მომავალი მარჯვენა ნაწილი
z->n = t - 1;
for (int j = 0; j < t - 1; j++) // ბოლო (t - 1) გასაღების კოპირება
        z->keys[j] = y->keys[j + t];
...
y->n = t - 1; // არსებულის ზომის შეცვლა
...
C[i + 1] = z; // მარჯვენა შვილის მინიჭება
...
keys[i] = y->keys[t - 1]; // შუა გასაღების გადატანა i -ურ პოზიციაზე
...
```

ახლა, განვიხილოთ K გასაღების წაშლის ალგორითმის ფსევდო-კოდი:

1. თუ გასაღები k არის კვანძ x-ში და x არის ფოთოლი, მაშინ წაშალე k გასაღები x-დან.
2. თუ გასაღები k არის კვანძ x-ში და x არის შუალედური კვანძი, შეასრულე შემდეგი:

a. თუ k-მდე მყოფ y შვილს, რომელიც x კვანძშია აქვს t გასაღები მაინც, მაშინ იპოვე k-ს წინამორბედი k0 y-ის ქვეხეში. რეკურსიულად წაშლე k0 გასაღები და ჩაანაცვლე k გასაღები k0-ით x კვანძში.
b. თუ y-ს აქვს t-ზე ნაკლები გასაღები, მაშინ სიმეტრიულად ზედა შემთხვევისა, შეამოწმე შვილი z რომელიც მოდის k-ს შემდეგ x კვანძში. თუ z-ს აქვს t გასაღები მაინც, მაშინ იპოვე "შემდეგი" k0 გასაღები k-ს გამოყენებით z ფესვის მქონე ქვეხეში. რეკურსიულად წაშალე k0 და x კვანძში ჩაანაცვლე k გასაღბი k0-ით.
c. სხვა შემთხვევაში, თუ ორივეს, y-სა და z-ს აქვს მხოლოდ t-1 გასაღები, მაშინ შეაერთე k და z-ის ყველა გასაღები y-თან, ისე რომ x-მა დაკარგოს ორივე: k და მიმთითებელი z-ზე, და y-ს საბოლოოდ ექნება 2t-1 გასაღები. შემდეგ, წაშლე z კვანძი და რეკურსიულად წაშალე k გასაღბი y-დან.

3. თუ გასაღები k არ არის შუალედურ კვანძ x-ში, მაშინ განსაზღვრე ფესვი x.c(i) შესაბამისი ქვეხის, რომელიც უნდა შეიცავდეს k გასაღებს, თუ, რათქმაუნდა, k არის ხეში. თუ x.c(i)-ს აქვს მხოლოდ t-1 გასაღები შეასრულე 3a ან 3b იმისათვის რომ გარანტირებულად შევძლოთ გადასვლა ისეთ კვანძში რომელსაც t გასაღები მაინც აქვს. დავასრულოთ ოპერაცია x-ზე რეკურსიულად გადასვლით.
   a. თუ x.c(i)-ს აქვს მხოლოდ t-1 გასაღები, მაგრამ ჰყავს უშუალო მეზობელი, რომელსაც აქვს t გასაღები მაინც, მაშინ x.c(i)-ს მიეცი ერთი ზედმეტი გასაღები შემდეგი ოპერაციების შესრულებით: x-დან გასაღები გადაგვაქვს x.c(i)-ში, x.c(i)-ს ახლო მარცხენა ან მარჯვენა მეზობლიდან აგვაქვს გასაღბი x-ში, და საბოლოოდ შესაბამისი შვილის მიმთითებელი მეზობლისგან გადაგვაქვს x.c(i)-ში.
   b. თუ ორივეს: x.c(i)-სა და x.c(i)-ს ახლო მეზობლებს t-1 რაოდენობის გასაღბები აქვთ, მაშინ შეაერთე x.c(i) ერთ-ერთ მეზობელთან, რაც გულისხმობს x გასაღების ქვემოთ ჩატანას, ახალ შერწყმულ კვანძში. ეს ჩანატა x გასაღებს ჩასვამს ზუსტად კვანძის შუაში.

კოდში გასაღბის წაშლისას ჩვენ ვიყენებთ შემდეგ ფუნქციებს:

- **void** remove(**int** k)
- **void** removeFromLeaf(**int** idx);
- **void** removeFromNonLeaf(**int** idx);
- **int** getPred(**int** idx);
- **int** getSucc(**int** idx);
- **void** fill(**int** idx);
- **void** borrowFromPrev(**int** idx);
- **void** borrowFromNext(**int** idx);
- **void** merge(**int** idx);

როგორც ხედავთ ჩასმის ოპერაციისგან განსხვავებით, წაშლის ოპერაცია ბევრ მეთოდს იყენებს. ეს განაპორიობა იმან, რომ, როგორც უკვე ვიცით, თავად წაშლის ოპერაცია რთულია.

removeFromLeaf მეთოდი გამოიძახება ფოთოლზე და ახდენს idx პოზიციაზე არსებული ელემენტის წაშლას. გვაქვს ერთი შეზღუდვა: ამ მეთოდის გამოყენება შეგვიძლია ისეთ ფოთოლზე, რომელსაც აქვს მინიმალურ გასაღებებზე მეტი გასაღები, ანუ (t - 1)-ზე მეტი გასაღები.

getPred და getSucc დამხმარე მეთოდებია, რომლებიც აბრუნებენ idx პოზიციაზე მყოფი გასაღების შესაბამისი წინამორბედისა და შემდეგი გასაღების მნიშვნელობას B-ხიდან. შეგვიძლია მოვიყვანოთ ერთ-ერთის კოდი, რათა ყველაფერი გასაგები იყოს. კოდი რეკურსიულია და ჩადის ფოთლამდე (getSucc მეთოდის კოდიც ანალოგიურად მუშაობს):

```
int getPred(int idx)
{
	// გადავიდეთ ყველაზე მარჯვენა ელემენტზე, სანამ არ მივალთ ფოთლამდე
	OldBTreeNode *cur = C[idx];
```

```
	while (!cur->leaf)
		cur = cur->C[cur->n];

	// დააბრუნე ფოთლის ბოლო გასაღები
	return cur->keys[cur->n - 1];
}
```

borrowFromPrev მეთოდი თხოულობს გასაღებს C[idx - 1] შვილისგან და სვამს მას C[idx]-ში. ეს ოპერაცია სრულდება მაშინ, როდესაც C[idx]-ის გასაღებების რაოდენობა მინიმალურია და წაშლის ოპერაციას სჭირდება ამ კვანძში ჩასვლა გასაღების წასაშლელად. უნდა შევნიშნოთ, რომ ეს მეთოდი გამოიძახება იმ შემთხვევაში თუ C[idx - 1] შვილი არსებობს. არ არსებობის შემთხვევაში წაშლის ალგორითმი ამ მეთოდს უბრალოდ არ გამოიძახებს.

ელემენტის გაცვლა მარტივად ხდება: C[idx - 1] შვილის ბოლო გასაღბი გადადის მშობლის idx პოზიციაზე ხოლო idx პოზიციაზე არსებული კი გადადის C[idx]-ის თავში:

```
void borrowFromPrev(int idx)
{
	BTreeNode *child = C[idx];
	BTreeNode *sibling = C[idx - 1];
	...
	child->keys[0] = keys[idx - 1];
	...
	keys[idx - 1] = sibling->keys[sibling->n - 1];
	...
}
```

კოდში გამოტოვებულია წაძვრის ოპერაციები და ასევე შვილის მიმთითებლის გადატანა. გასაღების გადატანასთან ერთად, თუ ჩვენ არ ვიმყოფებით ფოთოლში, მაშინ C[idx - 1]-ის ბოლო შვილი ხდება C[idx]-ის პირველი შვილი. ეს ოპერაცია აუცილებელია, რათა არ დავკარგოთ ხის ერთ-ერთი კვანძი. რაც შეეხება

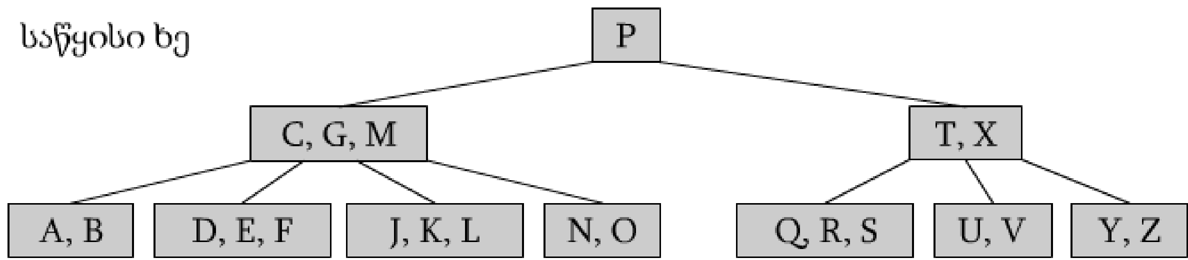
borrowFromNext-ს, ისიც მსგავს ოპერაციებს ასრულებს უბრალოდ შებრუნებულად.

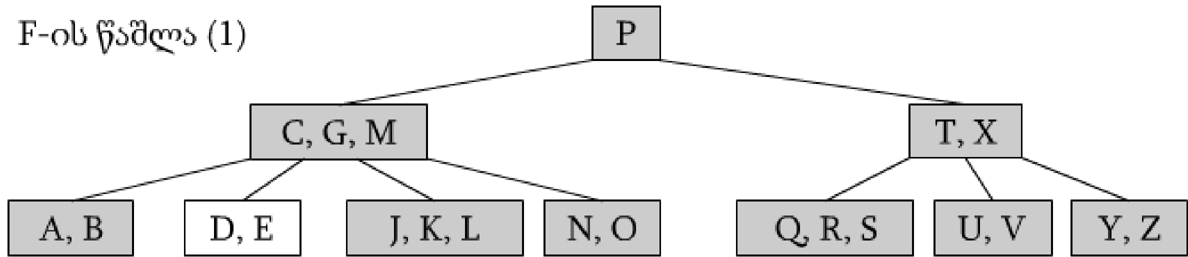
merge მეთოდი აერთიანებს C[idx]-სა და C[idx+1]-ს. ეს ხდება მაშინ, როდესაც ორივეში გასაღებების რაოდენობა მინიმალურია. გაერთიანების დროს, მიმდინარე კვანძის idx-ური გასაღები ჩამოდის გაერთიანებული შვილი კვანძის შუაში. აქაც, აუცილებელი პირობაა, რომ ეს ფუნქცია შესრულდეს ისეთ C[idx] და C[idx+1] შვილებზე, რომლებსაც მინიმალური რაოდენობა გასაღებები აქვთ, რადგანაც გაერთიანების შემდეგ მათი რაოდენობა მაქსიმალური გახდება და არ უნდა დაირღვეს მაქსიმალური გასაღებების რაოდენობის წესი B-ხის კვანძისთვის.

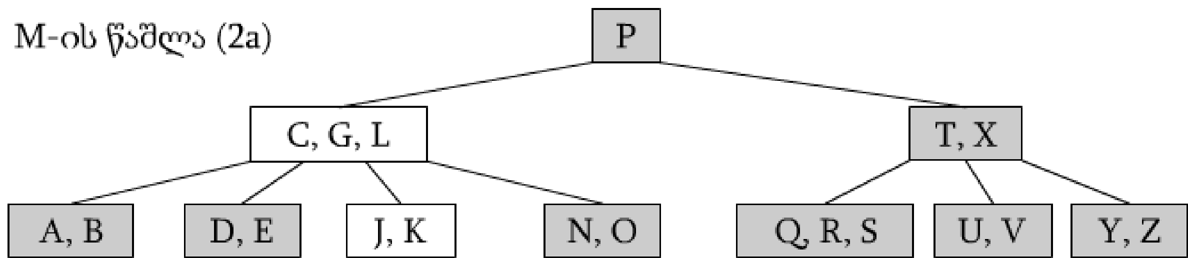
```
void merge(int idx)
{
        BTreeNode *child = C[idx];
        BTreeNode *sibling = C[idx + 1];

        child->keys[t - 1] = keys[idx]; // შუაში ჩამოტანა

        // გასაღებების კოპირება C[idx+1]-დან C[idx]-ის ბოლოში
        for (int i = 0; i < sibling->n; ++i)
                child->keys[i + t] = sibling->keys[i];
        ...
}
```

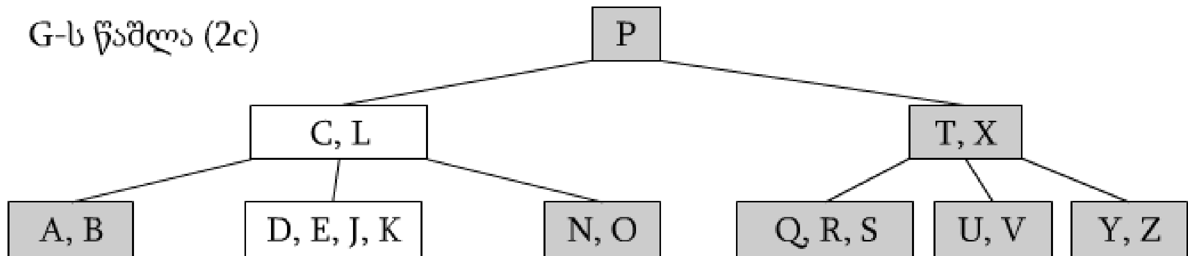
რაც შეეხება fill მეთოდს, იგი აერთიანებს ზემოთ აღწერილ მეთოდებს, რათა აუცილებლობის შემთხვევაში შეავსოს შვილი კვანძი მასზე გადასვლის წინ.

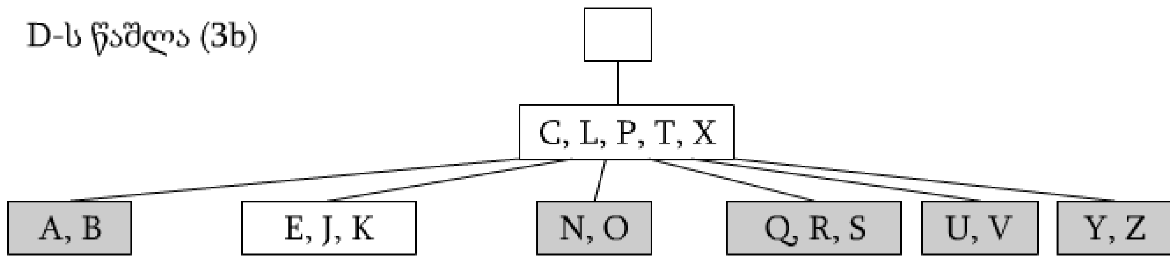
წაშლის ალგორითმის მუშაობის მაგალითი:

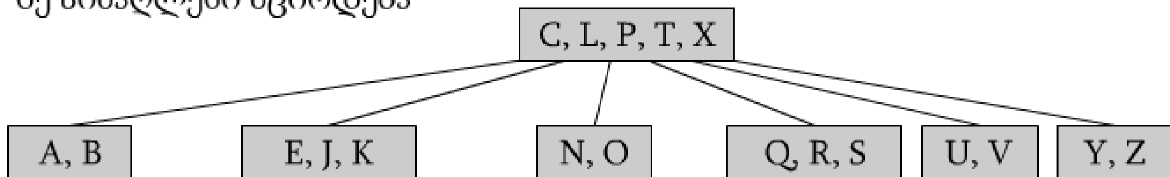

საწყისი ხე
P
C, G, M
T, X
A, B
D, E, F
J, K, L
N, O
Q, R, S
U, V
Y, Z

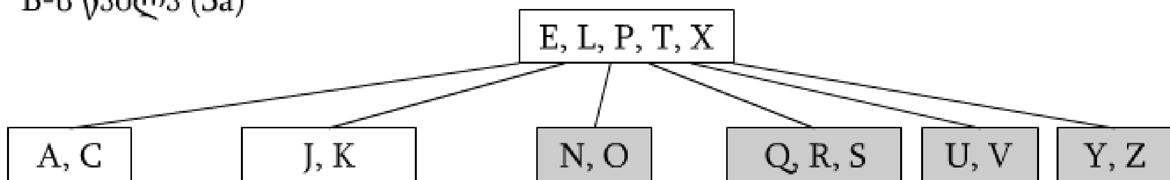

F-ის წაშლა (1)
P
C, G, M
T, X
A, B
D, E
J, K, L
N, O
Q, R, S
U, V
Y, Z


M-ის წაშლა (2a)
P
C, G, L
T, X
A, B
D, E
J, K
N, O
Q, R, S
U, V
Y, Z


G-ს წაშლა (2c)
P
C, L
T, X
A, B
D, E, J, K
N, O
Q, R, S
U, V
Y, Z







K გასაღების ძებნის ფსევდო-კოდი შედარებით მარტივია:

1. ძებნას ვიწყებთ ფესვიდან (გასაღებების დასაწყისიდან ბოლომდე გარბენით).
2. თუ გასაღები ვიპოვეთ, მაშინ ვაბრუნებთ მიმდინარე კვანძს, სადაც ეს გასაღები მოიძებნა.
3. თუ გასაღები არ მოიძებნა, მაშინ ვიღებთ შვილ კვანძს, რომელიც შეიძლება შეიცავდეს საძიებო გასაღებს და ვიმეორებთ პირველ (1) შემთხვევას ამ კვანძისთვის.

ძებნის კოდი გამოიყურება შემდეგნაირად:

```
BTreeNode* search(int k)

{
```

```
        int i = 0;

        while (i < n && k > keys[i])

                i++;

        if (keys[i] == k)

                return this;

        if (leaf == true)

                return NULL;

        return C[i]->search(k);

}
```

B-ხე კომპარატორით

ზემოთ აღწერილი B-ხის იმპლემენტაცია სავსებით სწორი და გამართულია, თუმცა იგი არ არის ოპტიმიზებული და მოუქნელია. მოუქნელია იმ მხრივ, რომ იგი გასაღებებში ინახავს მხოლოდ მთელ რიცხვებს და ისიც არის მხოლოდ int ტიპის. ეს არ არის კარგი სტილი, თუ ჩვენ გავითვალისწინებთ იმ გარემოებას, რომ დღვანდელი C++-ის შიდა ან გარე ბიბლიოთეკებში, სადაც კი გვხვდება რაიმე სტრუქტურა, არცერთი მათგანი არ არის შეზღუდული გასაღების ტიპით. თავდაპირველად, ჩვენ გადავწყვიტეთ მოგვეგვარებინა ზუსტად ეს პრობლემა, რომელიც დაგვეთნხმებით, რომ შეზღუდვა უფროა ვიდრე პრობლემა. ჩვენ B-ხე განვაზოგადეთ. კერძოდ, იგი გასაღების ტიპისგან დამოუკიდებელი გავხადეთ. ამ ყველაფრის შესაძლებლობა ჩვენ თანამედროვე C++-ის template-მა მოგვცა. ტიპის განზოგადებასთან ერთად აუცილებელი გახდა შედარების ოპერაციის განსაზღვრა, რომელიც სხვადასხვა ტიპებისთვის სხვადასხვა პრინციპით სრულდება. მაგალითად, რიცხვების ტიპებისთვის ნაკლებობის ოპერაცია შეიძლება იგივე დარჩეს რაც აქამდე იყო, მაგრამ, ვთქვათ, სტრიქონის ტიპისთვის ნაკლებობის შედარების ოპერაცია შეიძლება სხვანაირი ლოგიკით სრულდებოდეს.

ამ პრობლემის გადაჭრა ბევრი გზით და ხერხით შეიძლება, როგორც მარტივით ასევე რთულითაც. თუმცა, ჩვენ ჩავთვალეთ რომ საუკეთესო პრაქტიკული გამოსავალი იქნებოდა გვეხელმძღვანელა არსებული C++-ის მონაცემთა სტრუქტურის ბიბლიოთეკების სტილით. შეგვეძლო აგვეღო და გვენახა ნებისმიერი სტრუქტურის რეალიზაცია (შეზღუდულები არ ვიყავით), თუმცა სიმარტივისათვის ავირჩიეთ [3] set-ი. set-ის კლასის პროტოტიპს აქვს შემდეგი სახე:

```
template<class Key, class Compare = std::less<Key>, class Allocator = std::allocator<Key>>
class set;
```

აქ ჩვენ გვაინტერესებს შედარების ოპერატორი (რასაც ვუწოდებთ კომპარატორს), რომელიც template-ში არის განსაზღვრული. გაჩუმებით ვხედავთ, რომ

განსაზღვრული არის ნაკლებობის კომპარატორი. ჩვენი ხის კლასის განსაზღვრებაც შესაბამისად შეიცვალა და გახდა set-ის განსაზღვრების მსგავსი:

```
template<typename T, typename Compare = std::less<T>> class BTree;
```

less კომპარატორი არის განსაზღვრული C++-ის <functional> ბიბლიოთეკაში და არა მხოლოდ less. გვაქვს ასევე მეტობის, მეტობის-ან-ტოლობის, ნაკლებობის-ან-ტოლობისა და სხვა მრავალი კომპარატორი. კომპარატორი ჩანს ტემპლიტის განსაზღვრებაში, თუმცა მისი შექმნა და ინიციალიზება (სამომავლოდ კოდში გამოყენებისთვის) ხდება კლასის კონსტრუქტორში:

```
BTree(..., Compare& _cmp = Compare()) { ... cmp = _cmp; ... }
```

შედარების ოპერაციის შესასრულებლად ვიძახებთ კომპარატორს, როგორც ფუნქციას და გადავცემთ ორ შესადარებელ წყვილს:

```
if (cmp(a, b)) {… }
```

ამ ეტაპზე, გარდა ზემოთ თქმულისა, შევცვალეთ გასაღებებისა და შვილების მასივები პოინტერით. ამგვარად, კლასის განადგურების დროს, დესტრუქტორის გამოყენებით, ვათავისუფლებთ დაკავებულ მეხსიერებას. აგრეთვე დავაჩქრეთ ძებნის ალგორითმი. რადგან ვიცით რომ გასაღებები დალაგებულია, შეგვიძლია გამოვიყენოთ ბინარული ძებნის ალგორითმი, რომელიც მუშაობს ლოგარითმულ დროში განსხვავებით არსებული ალგორითმისა, რომელიც წრფივ დროს ანდომებს გასაღებში პოზიციის მოძებნას.

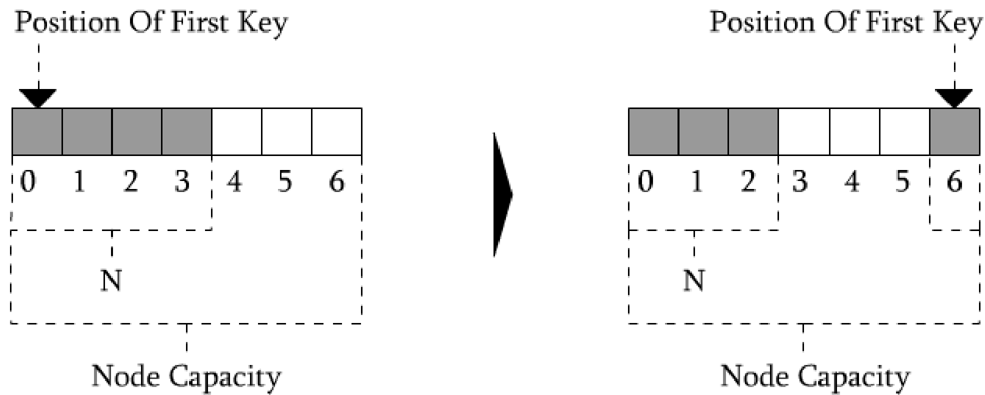
B-ხე წრიული დალაგებული მასივით

როგორც ჩვენ ზემოთ აღვნიშნეთ, B-ხის კვანძი (რაც ჩვენ შემთხვევაში ცალკე ობიექტს წარმოადგენს) შეიცავს ამ კვანძში არსებული გასაღებების სიმრავლესა და შვილების კვანძების (უფრო სწორად, მიმთითებლების) სიმრავლეს. ზემოთ წარმოდგენილ B-ხეში ეს გასაღებებისა და შვილების სიმრავლე უბრალო მასივია. მასივს გააჩნია საკუთარი მინუსები, რაც ზღუდავს ოპერაციების მოქნილად და ოპტიმალურად შესრულების შესაძლებლობას. აუცილებელია ავხსნათ თუ რას ვგულისხმობთ წინა წინადადებაში. მაგალითად, ავიღოთ მასივში ელემენტების ჩამატების ოპერაცია. ეს ჩამატების ოპერაცია გულისხმობს, თავდაპირველად, ჩასამატებელი ელემენტისთვის ადგილის გათვისუფლებას და მხოლოდ ამის შემდგომ მნიშვნელობის ჩაწერას. ხოლო რაც შეეხება მასივის ზომას, ჩვენ მასზე არ ვღელავთ, რადგანაც კვანძის შექმნისას ვიღებთ მაქსიმალური ზომის მასივს (t პარამეტრის მიხედვით: (2 * t – 1) გასაღებებისთვის და 2t შვილებისთვის). წაძვრა ადგილის გასათვისუფლებლად უნდა მოხდეს ელემენტების ბოლოში გადაწევით, რაც წრფივ დროს გამოიყენებს უარეს შემთხვევაში, როდესაც მასივის თავში მოგვინდება ჩამატება. წაშლის ოპერაციაც მოითხოვს ელემენტების წაძვრას, რათა წაშლილი ცარიელი ადგილი შეივსოს. აქედან ჩანს, რომ B-ხე ცუდად იმუშავებს თუ ჩვენ ჩასასმელ ელემენტებს გადავაწვდით კლებადობით და წაშლის დროს კი ყოველ იტერაციაზე წავშლით ხეში არსებულ მინიმალურ ელემენტს.

ენაში ჩაშენებული მასივის ცვლილებაზე ჩვენ არ გვიფიქრია (რაც შეუძლებელიც კია). ჩვენ შევცვალეთ მასივში ჩვენთვის საჭირო ინფორმაციის წარმოდგენის ლოგიკა, რათა გამოგვესწორებინა, ან მცირედიც მაინც დაგვეჩქარებინა თავდაპირველი მასივის ოპერაციები. კერძოდ, მასივის ორივე ბოლო გავხადეთ მოძრავი. ამისათვის, შემოვიღეთ ახალი ცვლადი: positionOfFirstKey, რომელიც არის მასივში პირველი გასაღების ინდექსი. ბოლოს ინდექსის გამოთვლა შეგვიძლია positionOfFirstKey-ს, ელემენტების რაოდენობისა და მასივის მაქსიმალური ზომის მნიშვნელობების გამოყენებით შემდეგნაირად:

(positionOfFirstKey + N - 1) % ndCapacity

აქ N არის ელემენტების რაოდენობა მასივში არსებულ მომენტში. ხოლო, ndCapacity არის კვანძში გასაღებების რაოდენობის მაქსიმუმი (ეს იგივეა რაც 2*t-1).



სურათზე გამოსახულია საწყისი პოზიციის გაწევა მარცხნივ.

ასეთ მასივს ვუწოდოთ და ამიერიდან მოვიხსენიებთ, როგორც წრიულ მასივს (B-ხის შემთხვევაში, როგორც წრიულ დალაგებულ მასივს). წრიულ მასივში ჩაწერისა და წაშლის დროს ჩვენ ვიყენებთ შემდეგ ალგორითმებს:

ჩამატების დროს:

- თუ ადგილი, სადაც ჩასმა ხდება ბოლოსთან უფრო ახლოსაა ვიდრე თავთან, მაშინ ადგილის გათვისუფლება ხდება ელემენტების წაძვრით ბოლოკენ (მარჯვნივ).
- თუ არადა, ადგილის გასათავისუფლებლად ელემენტებს ვწევთ თავისკენ (მარცხნივ).

წაშლის დროს:

- თუ წაშლა ხდება ბოლოსთან უფრო ახლოს, მაშინ წაშლის შემდეგ ბოლო ელემენტები გადმოდიან ცარიელი ადგილის შესავსებად.
- თუ არადა, თავითან არსებული ელემენტები გადმოდიან.

თეორიულად თუ შევაფასებთ, მაშინ ეს ორი ალგორითმი უარეს შემთხვევაში ორჯერ აჩქარებს წინა ჩამატებისა და წაშლის ალგორითმებს. რათქმაუნდა, უნდა

შევნიშნოთ რომ უბრალო მასივის გამოყენებისას არ გვიწევდა დამატებითი ცვლადების გამოყენება და მართვა.

შვილების რაოდენობა B-ხეში, როგორც ჩვენ აღვნიშნეთ, ერთით მეტია გასაღებების რაოდენობაზე. ის რაც ერთით ზრდის მასივის ზომას არის ბოლო შვილის მიმთითებელი. გასაღებების წაძვრის დროს i-ურ გასაღებთან ერთად მოძრაობს მისი i-ური შვილიც. რაც შეეხება ბოლო შვილს, ის შეიძლება ჩავთვალოთ, რომ ზედმეტია ასეთი გადაწევა-გადმოწევის ოპერაციების დროს. ყოველ შემთხვევაში ამ ოპერაციების უმეტესობა შემთხვევაში ჩვენ ბოლო შვილის გადაადგილებით ვასრულებთ ზედმეტ ოპერაციას. ამიტომაც, ჩვენ გადავწყვიტეთ და ბოლო შვილი ამოვიღეთ შვილების მასივიდან. იგი შევინახეთ ცალკე პარამეტრად, რამაც შეამცირა შვილების მასივის ზომა და გახადა გასაღებების მასივის ზომის ტოლი. ბოლო შვილს წაძვრების დროს ნაკლებად ვეხებით, ამიტომაც ეს მოდიფიკაცია კოდის ოპტიმიზაციაც არის:

```
BTreeNode<T, Compare>* c_last = NULL;
```

B-ხე წყვილების წრიული დალაგებული მასივით

წაშლის ოპერაციის დროს ხდება გასაღებისა და მისი მარცხენა შვილის გადაადგილება. ზემოთ აღწერილი B-ხის იმპლემენტაციის შემთხვევაში, წაშლის დროს ეს ორი ოპერაცია ხდება ცალ-ცალკე, რაც არაოპტიმალურია. ასეთი ზედმეტი ოპერაციის თავიდან ასაცილებლად, ჩვენ დავაწყვილეთ i-ური გასაღები თავის i-ურ (მარცხენა) შვილთან ერთად. ამისათვის ჩვენ შემოვიღეთ ახალი კლასი ასეთი წყვილებისთვის, რომელსაც დავარქვით NodePair და რომელიც გამოიყურება შემდეგნაირად:

```
template<typename T, typename Compare = std::less<T>>

class NodePair

{

        private:

                T key;

                BTreeNode<T, Compare> *child;

        public:

                ...

                T getKey() { return key; }

                BTreeNode<T, Compare>* getChild() { return child; }

                void setKey(T _key) { key = _key; }

                void setChild(BTreeNode<T, Compare>* _child) { child = _child; }

};
```

ეს კლასი ინახავს ერთ გასაღებს და მის მარხცენა შვილს. შესაბამისად, მთავარ B-ხის კლასში ჩვენ გასაღებებისა და შვილების მასივების მაგივრად გვექნება ერთი, წყვილების მასივი:

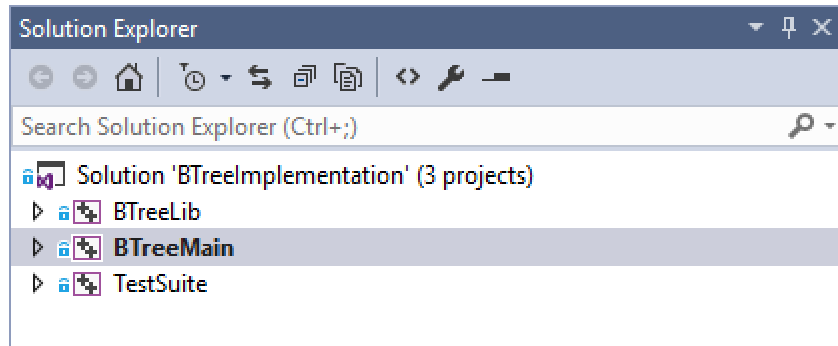
```
NodePair<T, Compare>** pairs;
```

კოდის იმ ფრაგმენტებში, სადაც ჩვენ ვახდენდით წაძვრას (მაგალითად, ჩასმისას ადგილის გასათავისუფლებლად ანდა წაშლისას ადგილის შესავსებად) ორი ოპერაციის მაგივრად შევასრულებთ ერთ ოპერაციას, რომელიც იქნება წყვილების მასივში i-ური ელემენტის j-ში გადატანა. მაგალითად:

```
...

while (i < fin)

{

	pairs[i % ndCapacity] = pairs[(i + 1) % ndCapacity];

	++i;

}

...
```

პროექტის სტრუქტურა

პროექტი დაწერილია Visual Studio-ში. იგი დაყოფილია სამ ნაწილად, პროექტად. თითოეულ ნაწილს თავისი დანიშნულება აქვს, თუმცა ერთი პროექტი შეიძლება მეორეს იყენებდეს.



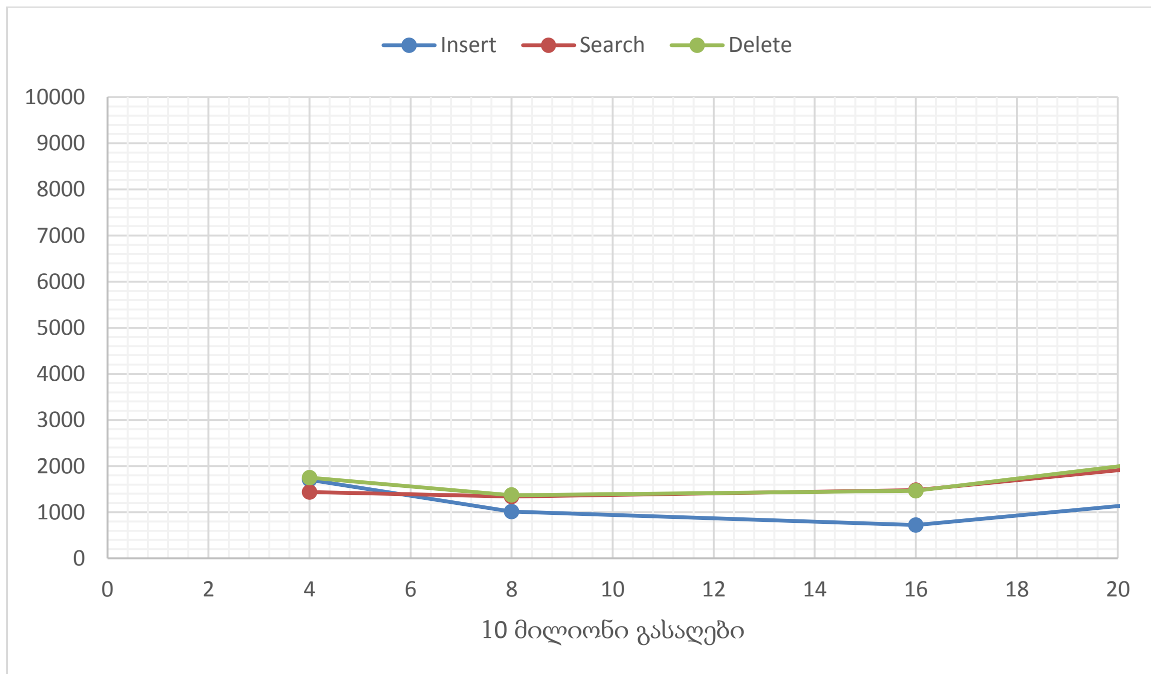
პირველი ნაწილი არის ბიბლიოთეკური პროექტი, რომელიც შეიცავს ჩვენი B-ხის იმპლემენტაციას, ძველი B-ხის იმპლემენტაციასა და წითელ-შავი ხის იმპლემენტაციას (თუ რატომ ვინახავთ ძველი ხის იმპლემენტაციასა და წითელ-შავი ხის იმპლემეტაციას ამას მიხვდებით პროექტის შემდეგი ნაწილების აღწერისას).

მეორე ნაწილი არის ტესტების პროექტი, რომელიც შეიცავს ერთეულოვან ტესტებს. ეს ტესტები ამოწმებს კოდის ცვლილების დროს იმას, რომ ის ოპერაციები რაც ჩვენ B-ხეში გვაქვს განსაზღვრული სწორად მუშაობს და იგივე შედეგს გვიბრუნებს. კერძოდ, ახალი იმპლემეტაციის ჩაწერის, ძებნისა და წაშლის კოდის შედეგებს ვადარებთ თავდაპირველი B-ხის იმპლემეტაციის შედეგებს სისწორის დასადგენად. შედარებისას ვიყენებთ B-ხეში განსაზღვრულ traverse() ფუნქციას, რომელიც აბრუნებს სტრიქონს, რომელიც აჩვენებს ხის მიმდინარე მდგომარეობას (გასაღებებს ფესვისას და ყველა კვანძისას) ფორმატირებულად. აგმვარი ერთეულოვანი ტესტების გამოყენების იდეა წარმოადგენს კარგ პრაქტიკას და მას სხვადასხვა ცნობილი ბიბლიოთეკა იყენებს. ამგვარად, მომხმარებელს შეუძლია გადაამოწმოს ბიბლიოთეკის მუშაობის სისწორე ფუნქციონალის გამოყენებამდე.

მესამე ნაწილი არის მთავარი პროექტი, რომელიც შეიცავს main() ფუნქციას და რომელიც ახდენს B-ხის იმპლემენტაციის სიჩქარის შედარებას ძველ B-ხესთან და აგრეთვე წითელ-შავ ხესთან. შედეგები იბეჭდება კონსოლურ ფანჯარაში. დროს ვითვლით მილიწამებში [4] chrono ბიბლიოთეკის გამოყენებით. კოდის გაშვებამდე, ვქმნით დიდ მასივს და ვავსებთ რიცხვებით (დაახლოებით მილიონი, 10 მ. და 100მ. რიცხვით), რომლებსაც შემდეგ ჩავსვამთ, მოვძებნით და წავშლით სამივე მონაცემთა სტრუქტურაში. ჩვენ წინასწარ განსაღვრული გვაქვს ოპტიმალური t პარამეტრი B-ხისთვის, რომელსაც ორივე ხის იმპლემენტაციას კონსტრუქტორში გადავცემთ შექმნისას.

შეფასება

თავდაპირველად შევაფასეთ t პარამეტრი საწყისი B-ხისთვის და ვნახეთ თუ რა სახის ცვლილებას გვაძლევდა სხვადასხვა მნიშვნელობისთვის:



შედეგებიდან გამოჩნდა რომ ოპტიმალური t პარამეტრი არის 16. შემდეგ ტესტებში ჩვენ t-ს მნიშვნელობად ავიღეთ 16.

B-ხის ახალი იმპლემენტაციის ეფექტიანობის შესაფასებლად ჩვენ მთვარი პროექტი გავუშვით საშუალო სიმძლავრის მქონე კომპიუტერზე 1 მ. და 10 მილიონ მონაცემზე. სიჩქარე შევადარეთ set-ის, საწყისი ხისა და წითელ-შავი ხის იმპლემენტაციებს:

1 მილიონი გასაღები

	B-tree (paired)	B-tree (not paired)	Old B-tree	Red-Black Tree	Set
Insert	397	280	68	361	1806
Search	463	394	146	218	717
Remove	491	369	141	86	1556

10 მილიონი გასაღები

	B-tree (paired)	B-tree (not paired)	Old B-tree	Red-Black Tree	Set
Insert	4620	3246	778	4152	20204
Search	5282	4100	1606	2382	8221
Remove	5783	4320	1543	867	17214

ჩანს რომ ჩვენი იმპლემენტაცია საგრძნობლად ნელია საწყის იმპლემენტაციასთან შედარებით. შენელების ერთ-ერთი მიზეზი არის კომპარატორი, რომელიც უფრო ნელა მუშაობს ვიდრე უბრალო შედარების ოპერატორი. რაც შეეხება მეხსიერების გამოყოფას, იგი ნანოწამებში სრულდება (იგულისხმება malloc). თუ ჩვენ ავიღებთ უფრო დიდ t პარამეტრს, მაშინ შევამჩნევთ დაჩქარებას საწყისი ხის იმპლემენტაციასთან შედარებით წაშლისა და ძებნის ოპერაციებში, რომლებშიც ბინარულ ძებნას ვიყენებთ:

1 მილიონი გასაღები (t = 1024)

	B-tree	Old B-tree	Red-Black Tree
Insert	3247	2395	691
Search	1188	3946	636
Remove	3235	4319	297

10 მილიონი გასაღები (t = 1024)

	B-tree	Old B-tree	Red-Black Tree
Insert	37932	34657	10673
Search	17110	42680	10020
Remove	38082	53948	3649

1 მილიონი გასაღები (t = 2048)

	B-tree	Old B-tree	Red-Black Tree
Insert	5472	4183	700
Search	1217	7398	629
Remove	5464	7772	298

10 მილიონი გასაღები (t = 2048)

	B-tree	Old B-tree	Red-Black Tree
Insert	60580	58783	10630
Search	17118	68778	10038
Remove	60252	97726	3658

ლიტერატურა

[1] Basic B-Tree implementation for C++ - http://www.geeksforgeeks.org/b-tree-set-1-introduction-2/

[2] Introduction to Algorithms - https://www.amazon.com/Introduction-Algorithms-3rd-MIT-Press/dp/0262033844

[3] Set - http://en.cppreference.com/w/cpp/container/set

[4] Chrono - http://en.cppreference.com/w/cpp/chrono