ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი

გოდერძი ლომინაშვილი

B-ხის იმპლემენტირება შიგა-მეხსიერებისთვის AVX ინსტრუქციების და გოგიერთი სხვა სპეციფიკის გამოყენებით

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერება

ნაშრომი შესრულებულია კომპიუტერული მეცნიერების მაგისტრის ნაშრომის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი, 2018

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, რაც საშუალებას მოგვცემს ოპტიმალურად გამოვიყენოთ მეხსიერება და ავასწრაფოთ მონაცემთა დამუშავების სიჩქარე.

ამ ნაშრომში შედარებისთვის განხილულია B-ხის რამდენიმე იმპლემენტაცია, რომლებიც იყენებენ შიგა მეხსიერების სპეციფიკას. წარმადობის ცვლილება შესწავლილია ექსპერიმეტულად 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 use memory and process huge number of data more quickly.

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

სარჩევი

შესავალი

მონაცემთა სტრუქტურებში ხეების მნიშვნელოვან ნაირსახეობას წარმოადგენს თვითბალანსირებადი ხეები. ბალანსირებულ ხეში ელემენტზე წვდომა უფრო სწრაფად ხდება ვიდრე არაბალანსირებულში, რადგან ხის კომპაქტურობა დამოკიდებული არ არის ელემენტების მიმდევრობაზე.

თვითბალანსირებადი ძებნის 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.

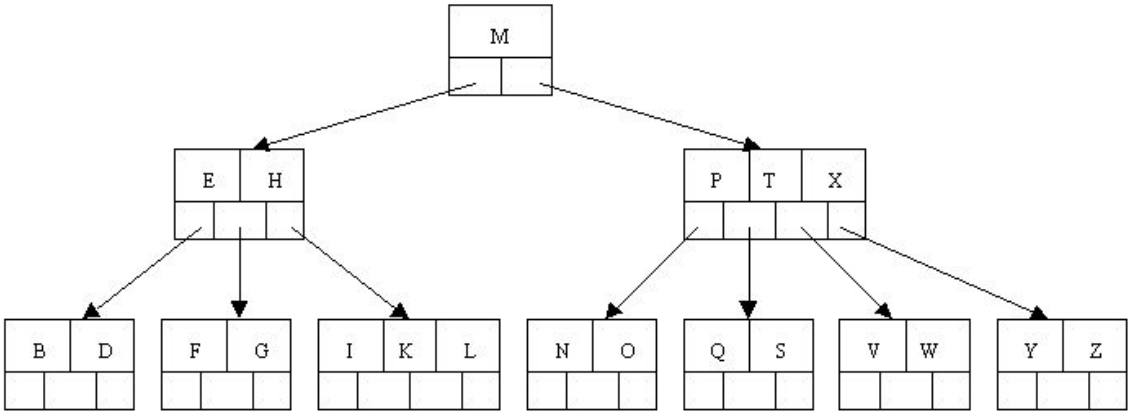
წარმოდგენილი ნაშრომი ამ საკითხს სწავლობს. იგი შედგება შესავალისა და ხუთი პარაგრაფისგან. აგრეთვე მოყვანილია მოკლე შეფასება და ლიტერატურის ნუსხა.

პირველ პარაგრაფში ...

...

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.

ხეში ელემენტის ჩამატება

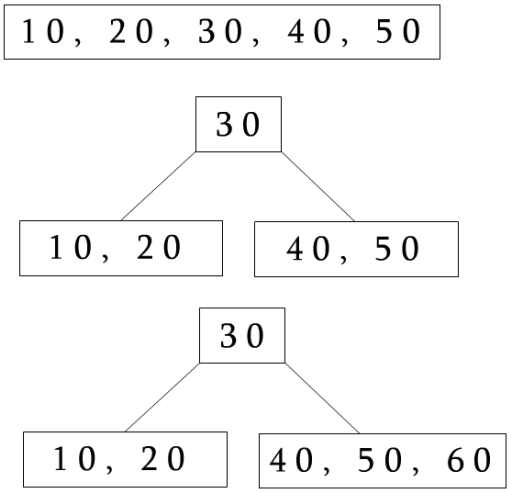
განვიხილოთ რაიმე k გასაღების ჩამატების ფსევდო-კოდი (შევნიშნოთ, რომ ეს ალგორითმი მიყვება [2] კორმენის წიგნში განხილულ ალგორითმს):

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 მიველით 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 გასაღები 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.
2. 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 გასაღებები 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 გასაღების 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 მეთოლს 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.

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-ხის მოდიფიკაცია AVX ინსტრუქციებით

ზემოთ აღწერილი B-ხის იმპლემენტაცია სავსებით სწორი და გამართულია, თუმცა იგი არ არის ოპტიმიზებული და მოუქნელია. მოცემულ იმპლემენტაციაში ხშირად გვიწევს მეხსიერებაზე ოპერაციების ჩატარება გადაკოპირება ან გადაწევა. არსებობს სპეციალური AVX-512 ინსტრუქციები რომლებიც ინტელის მიერ იქნა შემუშავებული კონკრეტული ამოცანების აჩქარების მიზნით, მაგალითად: ნეირონული ქსელების, ვექტორებსა და მეხსიერებაზე მუშაობის ოპტიმიზაციისთვის. ეს ინსტრუქციები ჩაშენებულია პოპულარულ კომპილატორებში როგორებიცაა: gcc 4.9+, clang 3.9+ და მაიკროსოფტის C++ კომპილატორი 2017 წლიდან.

განვიხილოთ როგორ შეგვიძლია მოცემული ინსტრუქციებით მეხსიერებაზე ოპერაციების ასწრაფება. B-ხე ჩასმისა და შვილის გაყოფის დროს გადაურბენს თითოეულ ელემენტს. ჩასმის დროს საჭირო ხდება ყველა გასაღები ჩასმის ადგილის მარჯვნივ გადავწიოთ 1 ინდექსით, რათა შევინარჩუნოთ დალაგებული სტრუქტურა. ამ დროს გავდივართ მასივის ბოლო ინდექსიდან ყველა ელემენტს ჩასმის ინდექსამდე. ვერ ვიყენებთ ორობით ძებნას მიუხედავად იმისა, რომ მასივი დალაგებულია. ინდექსის პოვნის შემდეგ მაინც მოგვიწევს სათითაოდ ყველა ელემენტის გადაწევა მარჯვნივ, რომ ჩავსვათ ახალი გასაღები. ამ შემთხვევაში შეგვიძლია გამოვიყენოთ std::memmove ფუნქცია, რადგან დანიშნულების მისამართი და წყარო მისამართი ერთმანეთს კვეთს, std::memmove გამოიყენებს შუალედურ ბაფერს მეხსიერების გადასატანად. ასევე სტანდარტული ფუნქციები როგორიცაა std::memmove და std::memcpy ოპტიმიზირებული არიან ზემოთ აღნიშნული ინსტრუქციებისთვის. ასევე შეგვიძლია კვანძის გახლეჩვის კოდი შევცვალოთ std::memcpy-ს გამოყენებით, კვანძის გახლეჩვის დროს ერთი კვანძის t-1 გასაღებისა და t+1 შვილის ახალ კვანძის ობიექტში გადასაკოპირებლად. რადგან მეხსიერების მისამართები სხვადასხვა ადგილებშია და ერთმანეთს არ კვეთს უნდა გამოვიყენოთ std::memcpy.

ამ ოპტიმიზაციის შედეგი ცხადი ხდება როცა t-ს ხარისხი არის მაღალი, მაგალითად t=1000. ამ დროს B-ხეს აკლდება სიმაღლე, რაც თავის მხრივ მოქმედებს ძებნის აჩქარებაზე და ასევე სწრაფდება შევსებაც ზემოთ აღნიშნული ოპტიმიზაციის

ხარჯზე. წარმადობის შედარებებში ჩანს რომ B-ხე ძებნით სწრობნის წითელ-შავ ხეს და მაღალი გასახებების ხარისხების დროს ჯობნის კიდევაც შევსებაში.

პირველ სურათზე ჩანს ციკლი ახალი გასაღების ჩასმის დროს,

```
// The following loop does two things
// a) Finds the location of new key to be inserted
// b) Moves all greater keys to one place ahead
/*while (i >= 0 && cmp(k, keys[i])) // keys[i] > k
{
    keys[i + 1] = keys[i];
    i--;
}*/

/*for (int i = 0; i < n; i++)
    std::cout << "existing key " << keys[i] << std::endl;*/

if (cmp(k, keys[0])) {
    memmove((void*)&keys[1], (const void*)&keys[0], n * sizeof(T));
    i = 0;
}
```

```
// Copy the last (t-1) keys of y to z
//for (int j = 0; j < t - 1; j++)
    //z->keys[j] = y->keys[j + t];

memcpy((void*)&z->keys[0], (const void*)&y->keys[t], (t - 1) * sizeof(T));

// Copy the last t children of y to z
if (y->leaf == false)
{
    //for (int j = 0; j < t; j++)
        //z->C[j] = y->C[j + t];
    memcpy((void*)&z->C[0], (const void*)&y->C[t], t * sizeof(BTreeNode*));
}

// Reduce the number of keys in y
y->n = t - 1;
```

მეორე სურათზე კვანძის გახლეჩვის დროს შეცვლილი ციკლები, ახალ კვანძში გასაღებების და შვილი კვანძების გადატანისთვის.

წარმადობის შედარებისთვის შეგვიძლია განვიხილოთ შედეგების სურათები.

პირველი წყვილი არის არაოპტიმიზირებული(მარცხნივ) და ოპტიმიზირებული(მარჯვნივ) B-ხეების შედარებები როცა t=10 000. მეორე წყვილი ასახავს t=1000 შემთხვევას.

```
Testing insertions:
Insert BTree 100000, Milliseconds: 364
Insert RedBlackTree 100000, Milliseconds: 22
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 20
Delete BTree 100000, Milliseconds: 461
Delete RedBlackTree 100000, Milliseconds: 10
Insert BTree 1000000, Milliseconds: 3872
Insert RedBlackTree 1000000, Milliseconds: 534
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 471
Delete BTree 1000000, Milliseconds: 5461
Delete RedBlackTree 1000000, Milliseconds: 287
Insert BTree 10000000, Milliseconds: 61066
Insert RedBlackTree 10000000, Milliseconds: 11088
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10008
Delete BTree 10000000, Milliseconds: 92149
Delete RedBlackTree 10000000, Milliseconds: 3865
All the test have finished.
```

```
$ ./BTreeMain.exe
Testing insertions:
Insert BTree 100000, Milliseconds: 91
Insert RedBlackTree 100000, Milliseconds: 23
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 19
Delete BTree 100000, Milliseconds: 537
Delete RedBlackTree 100000, Milliseconds: 16
Insert BTree 1000000, Milliseconds: 1747
Insert RedBlackTree 1000000, Milliseconds: 661
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 578
Delete BTree 1000000, Milliseconds: 7429
Delete RedBlackTree 1000000, Milliseconds: 286
Insert BTree 10000000, Milliseconds: 33362
Insert RedBlackTree 10000000, Milliseconds: 11194
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10127
Delete BTree 10000000, Milliseconds: 92405
Delete RedBlackTree 10000000, Milliseconds: 3907
All the test have finished.
```

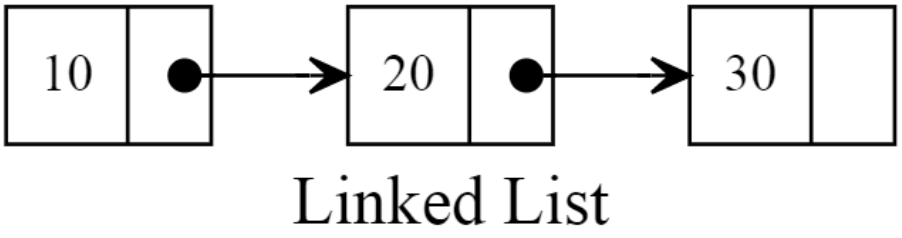
```
Testing insertions:
Insert BTree 100000, Milliseconds: 43
Insert RedBlackTree 100000, Milliseconds: 26
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 22
Delete BTree 100000, Milliseconds: 66
Delete RedBlackTree 100000, Milliseconds: 11
Insert BTree 1000000, Milliseconds: 532
Insert RedBlackTree 1000000, Milliseconds: 568
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 536
Delete BTree 1000000, Milliseconds: 1022
Delete RedBlackTree 1000000, Milliseconds: 279
Insert BTree 10000000, Milliseconds: 11627
Insert RedBlackTree 10000000, Milliseconds: 11603
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10506
Delete BTree 10000000, Milliseconds: 15034
Delete RedBlackTree 10000000, Milliseconds: 3873
All the test have finished.
```

```
Testing insertions:
Insert BTree 100000, Milliseconds: 19
Insert RedBlackTree 100000, Milliseconds: 24
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 20
Delete BTree 100000, Milliseconds: 66
Delete RedBlackTree 100000, Milliseconds: 11
Insert BTree 1000000, Milliseconds: 311
Insert RedBlackTree 1000000, Milliseconds: 544
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 477
Delete BTree 1000000, Milliseconds: 758
Delete RedBlackTree 1000000, Milliseconds: 294
Insert BTree 10000000, Milliseconds: 9508
Insert RedBlackTree 10000000, Milliseconds: 11112
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10121
Delete BTree 10000000, Milliseconds: 14854
Delete RedBlackTree 10000000, Milliseconds: 3905
All the test have finished.
```

B-ხის მოდიფიკაცია ბმული სიით

არ არსებობს მონაცემთა სტრუქტურა, რომელიც ყველა მოთხოვნას აკმაყოფილებს და სუსტი მხარე არ გააჩნია. B-ხის ზემოთ აღნიშნულ იმპლემენტაციას აქვს თავისი სუსტი მხარეები, მაგალითად მეხსიერების მოხმარება. როგორც კოდიდან ჩანს ყოველი კვანძის შექმნისას ხდება 2t და 2t - 1 ელემენტიანი მასივის ალოკაცია შვილებისთვის და გასაღებებისთვის. ამის გამო შეიძლება გვქონდეს შემთხვევა როდესაც ხეში დიდი ადგილი გამოუყენებელია და შესაბამისად მეხსიერებას დაკავებულია. ასევე ეს მეთოდი თავიდან აგვარიდებს მასივში ელემენტების გადაწევას წაშლის და ჩასმის დროს. რადგად ბმულ სიაში პირადაპირ გადავცვლით პოინტერს.

ერთ მხრივ ბმული სიის მაგალითია აღწერილი სურათზე, ერთ სტრუქტურაში გვაქვს ელემენტი და შემდეგ ელემენტზე მიმთითებელი, როგორც სურათზე ჩანს სიას აქვს ტრივიალური სტრუქტურა და მისი იმპლემენტაციაც მარტივია.



Linked List

C++ სტანდარტულ ბიბლიოთეკაში უკვე გვაქვს ორმხრივ და ცალმხრივ ბმული სიის იმპლემენტაციები. მოცემული ამოცანისთვის გამოყენებულია std::forward_list, რომელსაც საჭირო მეთოდები იმპლემენტირებული აქვს და გამოსაყენებლად მარტივია. ასევე აღსანიშნავია რომ მოცემული კლასი არ ინახავს ბმულ სიაში ელემენტების რაოდენობას, რაც ახსნილია წარმადობით, ამის გამო, როგორც მასივებით იმპლემენტირებულ B-ხეში, მოცემულ ვარიანტშიც მოგვიწევს კვანძში გასაღებების და შვილების რაოდენობის შენახვა და განახლებები.

სიებს ექნებათ შემდეგი სახე კოდში:

```
template <typename T>
class BTreeNode
{
    unsigned int childCount = 0;
    unsigned int keyCount = 0;
```

```
    unsigned int t = 0;
    std::forward_list<T> keys;
    std::forward_list<BTreeNode<T>*> children;
}
```

ჩასმის ალგორითმის იმპლემენტაცია ძალიან გავს მასივებით იმპლემენტაციაში გამოყენებულ ალგორითმს. თუ კვანძი ფოთოლი არ არის, გადავუყვებით ბმულ სიას მანამ სანამ არ ვიპოვით პირველ ელემენტს, რომელიც მეტია ჩასასმელ გასაღებზე, თუ ესეთი ასებობს, თუ არა მაშინ აღმოვჩნდებით ბოლო გასაღებზე. ვთქათ ბმულ სიაში ვიპოვეთ ელემენტი, რომლის შემდეგაც გასაღები უფრო მეტია ვიდრე ჩასასმელი ან რომლის შემდეგაც გვაქვს სიის ბოლო, i-ურ ადგილას, ამის შემდეგ 2t ზომის შვილების სიაში ვიღებთ შვილს, ეს შვილი უნდა იდგეს i+1 პოზიციაზე. გადავუვლით შვილების ბმულ სიას, გადავალთ შვილ კვანძში, თუ ეს კვანძი სავსეა მაშინ გავხლეჩთ და გავიმეორებთ პროცედურას სანამ არ მივალთ ფოთლამდე. ფოთოლში ვიპოვით ჩასასმელ ადგილს ზემოთ აღნიშნული წესით და სწორედ ამ ადგილზე ჩავსვავთ სიაში ჩვენს გასაღებს.

ჩასმა:

```
    void insertKey(T key) {
    if (this->isLeaf()) {
        if (key == this->keys.front()) return;
        if (key < this->keys.front()) {
            this->keys.push_front(key);
        }
        else {
            auto it = this->keys.begin();
            while (std::next(it) != this->keys.end() && key > *std::next(it)) ++it;
            auto nit = std::next(it);
            if (nit != this->keys.end() && key == *nit) return;
            this->keys.insert_after(it, key);
        }
        this->keyCount++;
    }
    else {
```

```
            auto it = this->keys.begin();
            unsigned int i = 0;
            while (it != this->keys.end() && key > *(it++)) i++;
            auto cit = this->children.begin();
            unsigned int childIndex = i;
            while (i-- != 0) cit++;
            auto childNode = (*cit);
            if (childNode->isFull()) {
                this->splitChild(childIndex, cit);
                if ((key < (*std::next(cit))->keys.front()))
                    childNode->insertKey(key);
                else
                    (*(++cit))->insertKey(key);
            }
            else
                childNode->insertKey(key);
        }
}
```

კვანძის გახლეჩა მოხდება იგივე პრინციპით, როგორც მასივში, მაგრამ მეხსიერების გადატანის მაგივრად სიის შუა ელემენტიდან მოვწყვეტთ სიას და ჩავაკერებთ ახალი კვანძის სიაში. შევცვლით ელემენტების რაოდენობებს და მშობელი კვანძის შვილებისა და გასაღებების სიაში ჩავსვავთ გახლეჩისას ამოსულ შუა ელემენტს და ახალ კვანძს.

გახლეჩა:

```
        void splitChild(
         unsigned int childIndex,
         typename std::forward_list<BTreeNode<T>*>::iterator childNodeIterator) {
         BTreeNode<T>* childNode = *childNodeIterator;
         BTreeNode<T>* newNode = new BTreeNode<T>(this->t);
         newNode->keyCount = this->t - 1;
```

```
        auto it = childNode->keys.begin();
        unsigned int i = this->t;
        while (--i != 1) ++it;
        newNode->keys.splice_after(newNode->keys.before_begin(), childNode->keys,
                std::next(it), childNode->keys.end());
        if (!childNode->isLeaf()) {
                i = this->t;
                auto cit = childNode->children.begin();
                while (--i != 0) ++cit;
                newNode->children.splice_after(newNode->children.before_begin(),
                        childNode->children, cit,
                        childNode->children.end());
                childNode->childCount = this->t;
                newNode->childCount = this->t;
        }
        auto pcit = this->keys.before_begin();
        i = childIndex;
        while (i-- != 0) ++pcit;
        this->keys.splice_after(pcit, childNode->keys, it, childNode->keys.end());
        this->keyCount++;
        childNode->keyCount = this->t - 1;
        this->children.insert_after(childNodeIterator, newNode);
        this->childCount++;
}
```

წაშლის ალგორითმი იქნება იგივე რაც მასივის შემთხვევაში, ამოშლის და ჩასმისთვის გამოვიყენებთ იგივე ტექნიკას რაც ზემოთ მოცემულ კოდშია. აქვე უნდა აღინიშნოს, რომ ბმულ სიაში O(1) დროში წვდომა ელემენტზე არ გვაქვს, შესაბამისად საჭირო ელემენტების საპოვნელად უნდა გადავუაროთ სიას. ამ დროს არ ვიყენებთ თვისებას, რომ ელემენტები დალაგებულია.

ამ შემთხვევაში ბმული სიის უპირატესობა მასივთან არის მეხსიერების ოპტიმალურად გამოყენებაში. ამ შემთხვევაში ჩვენ არ გვექნება ისეთი მეხსიერების ნაწილი, რომელიც გამოყოფილია ოპერაციული სისტემის მიერ ჩვენი ხისთვის და არის შეუვსებელი.

```
Random Shuffled List Insert!
 T = 3
Fill BTree 10000000, Milliseconds: 19528
 T = 4
Fill BTree 10000000, Milliseconds: 18568
 T = 5
Fill BTree 10000000, Milliseconds: 18900
 T = 6
Fill BTree 10000000, Milliseconds: 19164
Sorted List Insert!
 T = 3
Fill BTree 10000000, Milliseconds: 2697
 T = 4
Fill BTree 10000000, Milliseconds: 2530
 T = 5
Fill BTree 10000000, Milliseconds: 2617
 T = 6
Fill BTree 10000000, Milliseconds: 2706
Reverse sorted list insert!
 T = 3
Fill BTree 10000000, Milliseconds: 2114
 T = 4
Fill BTree 10000000, Milliseconds: 2018
 T = 5
Fill BTree 10000000, Milliseconds: 1554
 T = 6
Fill BTree 10000000, Milliseconds: 1474
```

წარმადობის ანალიზისთვის შეგვიძლია შევხედოთ სურათს, რომელზეც მოცემულია წარმადობა 100 000, 1000 000 და 10 000 000 ელემენტებზე. ასევე სურათზე ვნახავთ რომ პატარა t-სთვის წარმადობა უფრო მაღალია ვიდრე დიდი t-ებისთვის.

როგორც ვხედავთ 10 მილიონ ჩანაწერზე ყველაზე მაღალი წარმადობა მივიღეთ როცა t=3 ან t=4, თუმცა როდესაც ჩასასმელი გასაღებები ზრდადობით ან კლებადობითაა დალაგებული წარმადობა უფრო მაღალია როცა t იზრდება. ძიება მოცემულ ხეში, ისევე როგორც მასივით იმპლემენტირეულ B-ხეში, სწრაფია.

B-ხის მოდიფიკაცია წყვილების ბმული სიით

როდესაც ბმულ სიაში მოვძებნით სასურველ გასაღებს და შემდეგ გვიწევს მისი შესაბამისი შვილის მოძიება, გვიწევს იტერაცია შვილების ბმულ სიაში, ასევე შესამჩნევია, რომ მასივების ბმული სიით შეცვლამ კოდი შედარებით გაართულა. ამ პრობლემების გადასაჭრელად ჩვენ შეგვიძლია ორი ბმული სია შევცვალოთ ერთი სიით სადაც გასაღებების და შვილების წყვილები იქნებიან განთავსებული. რადგან შვილები შეიძლება 1-ით მეტი იყოს გასაღებებზე, მოგვიწევს ერთი შვილის ცალკე შენახვა. მნიშვნელობა არ აქვს იქნება ეს ბოლო შვილი თუ პირველი. მოცემულ მაგალითში შევინახავთ პირველ შვილს. აქედან გამომდინარე ყოველ გასაღებს წყვილში ექნებათ მასზე მეტი გასაღებებიან კვანძზე მიმთითებელი.

შეცვლილ კოდს ექნება შემდეგი სახე:

```
unsigned int childCount = 0;
unsigned int keyCount = 0;
unsigned int t = 0;
std::forward_list<std::pair<T, BTreeNode<T>*> > keyChildPairs;
BTreeNode<T>* firstChild;
```

ჩასმა:

```
void insertKey(T key) {
        if (this->isLeaf()) {
        if (key == this->keyChildPairs.front().first) return;
        if (key < this->keyChildPairs.front().first) {
        this->keyChildPairs.push_front(std::pair<T, BTreeNode<T>*>(key, nullptr));
        } else {
        auto it = this->keyChildPairs.begin();
        while (std::next(it) != this->keyChildPairs.end() && key > (*std::next(it)).first)
++it;
        auto nit = std::next(it);
        if (nit != this->keyChildPairs.end() && key == (*nit).first) return;
        this->keyChildPairs.insert_after(it, std::pair<T, BTreeNode<T>*>(key, nullptr));
```

```
        }
        this->keyCount++;
        } else {
        auto it = this->keyChildPairs.before_begin();
        BTreeNode<T> *childNode;
        if (key < this->keyChildPairs.front().first) {
        childNode = this->firstChild;
        }
        else {
        it = this->keyChildPairs.begin();
        while (std::next(it) != this->keyChildPairs.end() && key > (*std::next(it)).first)
it++;
        childNode = (*it).second;
        }

        if (childNode->isFull()) {
        this->splitChild(it);

        if(it == this->keyChildPairs.before_begin()) {
        if(key < this->keyChildPairs.front().first) {
        this->firstChild->insertKey(key);
        } else {
        this->keyChildPairs.front().second->insertKey(key);
        }
        } else {
        if((key < (*std::next(it)).first)){
        childNode->insertKey(key);
                }
        else{
```

```
        (*std::next(it)).second->insertKey(key);
                }
        }
        } else
        childNode->insertKey(key);
        }
 }
```

გახლეჩა:

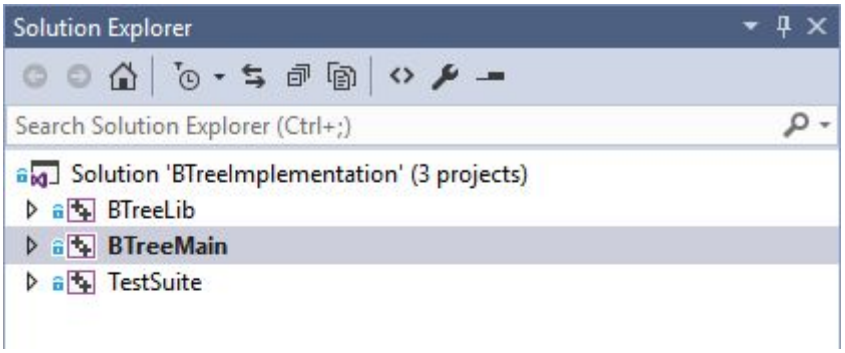
```
 void splitChild(typename std::forward_list<std::pair<T, BTreeNode<T>* > >::iterator
leftNodeIterator) {
        BTreeNode<T>* leftNode = leftNodeIterator ==
this->keyChildPairs.before_begin() ? firstChild : (*leftNodeIterator).second;
        BTreeNode<T>* rightNode = new BTreeNode<T>(this->t);

        auto it = leftNode->keyChildPairs.begin();
        unsigned int i = this->t;
        while(--i != 1) ++it;
        rightNode->firstChild = (*std::next(it)).second;
        rightNode->keyChildPairs.splice_after(rightNode->keyChildPairs.before_begin(),
leftNode->keyChildPairs,
                                                std::next(it),
leftNode->keyChildPairs.end());
        rightNode->keyCount = this->t - 1;
        leftNode->keyCount = this->t - 1;

        if(!leftNode->isLeaf()){
        rightNode->childCount = this->t;
        leftNode->childCount = this->t;
        }
```

```
        if(leftNodeIterator == this->keyChildPairs.before_begin()) {

        this->keyChildPairs.push_front(std::pair<T,
BTreeNode<T>*>((*std::next(it)).first, rightNode));

        } else {

        this->keyChildPairs.insert_after(leftNodeIterator, std::pair<T,
BTreeNode<T>*>((*std::next(it)).first, rightNode));

        }

  leftNode->keyChildPairs.erase_after(it);

        this->keyCount++;

        this->childCount++;

  return;

 }
```

ასევე აღსანიშნავია, რომ მოცემული კოდი არის კონცეფციის დამადასტურებელი და მომავალში შესაძლებელია მომავალში მისი მრავალმხრივი ოპტიმიზაცია და მოდიფიკაცია სხვადასხვა დანიშნულებისთვის.

პროექტის სტრუქტურა

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



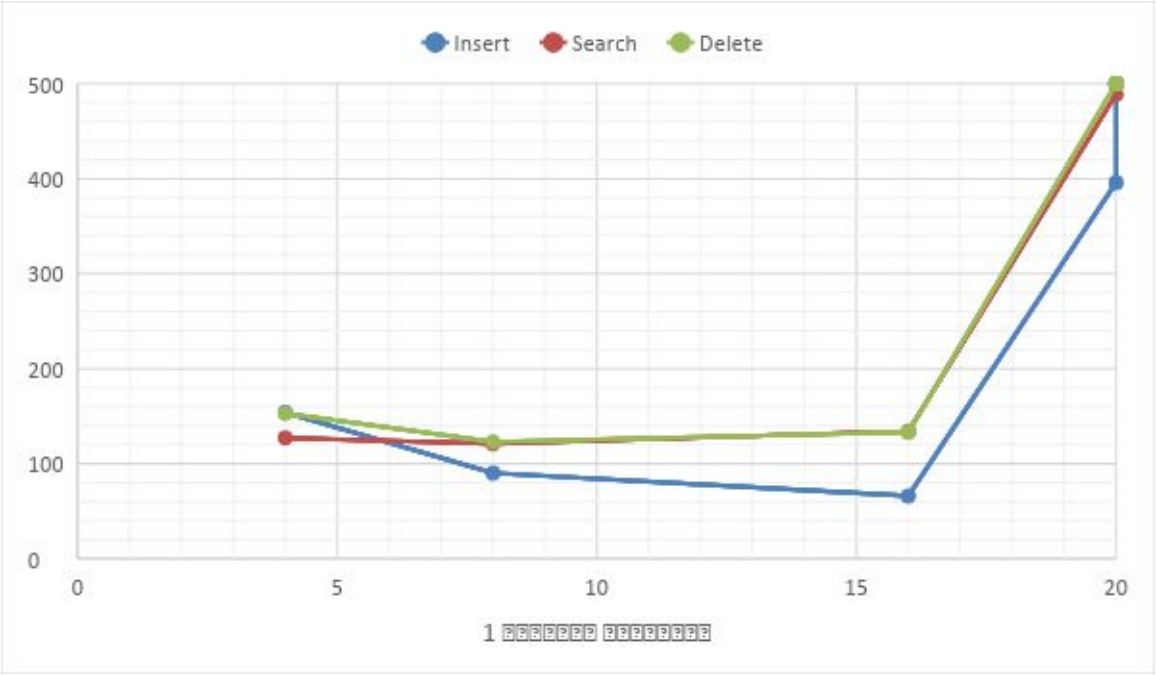
პირველი ნაწილი არის ბიბლიოთეკური პროექტი, რომელიც შეიცავს ჩვენი B-ხის იმპლემენტაციას, ძველი B-ხის იმპლემენტაციასა და წითელ-შავი ხის იმპლემენტაციას (თუ რატომ ვინახავთ ძველი ხის იმპლემენტაციასა და წითელ-შავი ხის იმპლემეტაციას ამას მიხვდებით პროექტის შემდეგი ნაწილების აღწერისას).

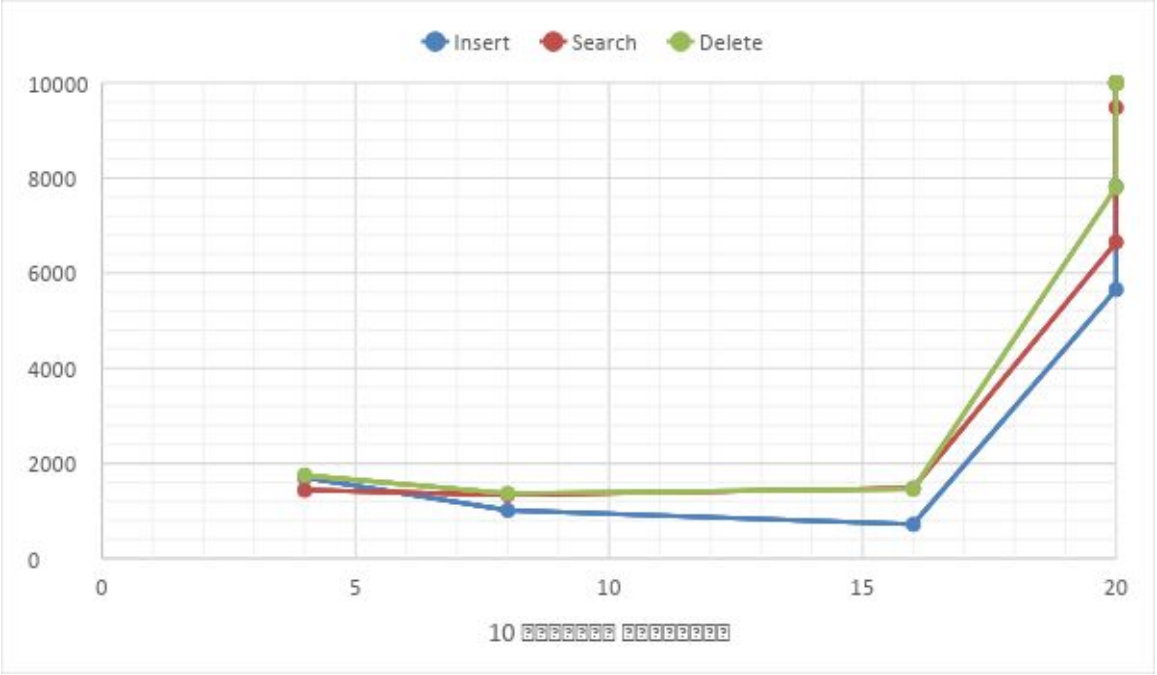
მეორე ნაწილი არის ტესტების პროექტი, რომელიც შეიცავს ერთეულოვან ტესტებს. ეს ტესტები ამოწმებს კოდის ცვლილების დროს იმას, რომ ის ოპერაციები რაც ჩვენ B-ხეში გვაქვს განსაზღვრული სწორად მუშაობს და იგივე შედეგს გვიბრუნებს. კერძოდ, ახალი იმპლემეტაციის ჩაწერის, ძებნისა და წაშლის კოდის შედეგებს ვადარებთ თავდაპირველი 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

სურათებზე ნაჩვენებია მასივებით იმპლემენტირებული არაოპტიმიზებული B-ხის წარმადობის დამოკიდებულება t პარამეტრზე.





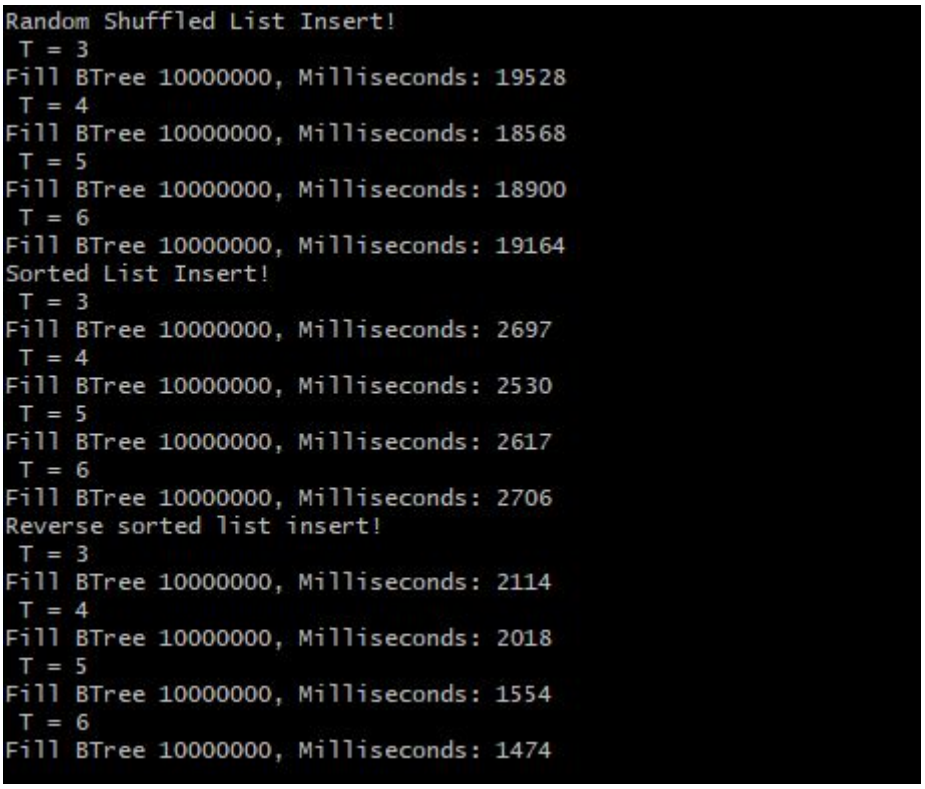
მოცემული შედეგებიდან ჩანს რომ ხისთვის ოპტიმალური t არის 16, თუმცა მოცემული პარამეტრი სულ სხვაა AVX ინსტრუქციებით ოპტიმიზირებული და ბმული სიით იმპლემენტირებული B-ხისთვის.

B-ხის ახალი იმპლემენტაციის ეფექტიანობის შესაფასებლად ჩვენ მთვარი პროექტი გავუშვით საშუალო სიმძლავრის მქონე კომპიუტერზე 1 მ. და 10 მილიონ მონაცემზე. გამოყენებული კომპიუტერის მონაცემებია: Intel i7 4700HQ 2.40 GHz.

```
Testing insertions:
Insert BTree 100000, Milliseconds: 19
Insert RedBlackTree 100000, Milliseconds: 24
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 20
Delete BTree 100000, Milliseconds: 66
Delete RedBlackTree 100000, Milliseconds: 11
Insert BTree 1000000, Milliseconds: 311
Insert RedBlackTree 1000000, Milliseconds: 544
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 477
Delete BTree 1000000, Milliseconds: 758
Delete RedBlackTree 1000000, Milliseconds: 294
Insert BTree 10000000, Milliseconds: 9508
Insert RedBlackTree 10000000, Milliseconds: 11112
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10121
Delete BTree 10000000, Milliseconds: 14854
Delete RedBlackTree 10000000, Milliseconds: 3905
All the test have finished.
```

```
$ ./BTreeMain.exe
Testing insertions:
Insert BTree 100000, Milliseconds: 91
Insert RedBlackTree 100000, Milliseconds: 23
Search BTree 100000, Milliseconds: 0
Search RedBlackTree 100000, Milliseconds: 19
Delete BTree 100000, Milliseconds: 537
Delete RedBlackTree 100000, Milliseconds: 16
Insert BTree 1000000, Milliseconds: 1747
Insert RedBlackTree 1000000, Milliseconds: 661
Search BTree 1000000, Milliseconds: 0
Search RedBlackTree 1000000, Milliseconds: 578
Delete BTree 1000000, Milliseconds: 7429
Delete RedBlackTree 1000000, Milliseconds: 286
Insert BTree 10000000, Milliseconds: 33362
Insert RedBlackTree 10000000, Milliseconds: 11194
Search BTree 10000000, Milliseconds: 0
Search RedBlackTree 10000000, Milliseconds: 10127
Delete BTree 10000000, Milliseconds: 92405
Delete RedBlackTree 10000000, Milliseconds: 3907
All the test have finished.
```

სურათებზე მოცემულია AVX B-ხე, მარცხნივ t=1000, მარჯვნივ t=10 000



სურათზე მოცემულია B-ხე ბმული სიით სხვადასხვა t-ებისთვის.

მოცემული სურათებიდან შეგვიძლია ვთქვათ რომ t=1000 AVX B-ხე წარმადობით სჯობნის t=10000 ხეს, თუმცა ეს ცვლილება გასაღებების რაოდენობის ზრდასთან ერთად მცირდება. შეგვიძლია დავაკვირდეთ რომ 10 მილიონ გასაღებზე ცვლილება შემცირდა. რაც შეეხება B-ხეს ბმული სიით, 10 მილიონ გასაღებზე შემთხვევითი ჩასმებისთვის t=4-ისთვის 7000 მილიწამით ჩამორჩება AVX B-ხეს, თუმცა აღსანიშნავია ორი რამ: როცა ჩასასმელი გასაღებები დალაგებულია, ბრდადობით ან კლებადობით, მაშინ B-ხე ბმული სიით რამოდენიმეჯერ სწრაფია ვიდრე სხვა ხვები, ეს გამომდინაორებს იქედან, რომ ჩვენ არ გვიწევს მასივში მეხსიერების გადაწევა ჩასმის დროს და პირდაპირ ვაბავთ სიას ელემენტს. მეორე, B-ხე ბმული სიით უფრო ეფექტურად მუშაობს მეხსიერებასთან ვიდრე სხვები, რადგან არ ხდება ცარიელი მეხსიერების ალოკაცია. მოცემული ტესტების დროს AVX B-ხე მოიხმარდა 700 მეგაბაიტ ოპერატიულ მეხსიერებას, ხოლო B-ხე ბმული სიით 400-ს.

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

[5] AVX Instructions - https://en.wikipedia.org/wiki/Advanced_Vector_Extensions