პრაქტიკული მეცადინეობა 1

სააუდიტორიო სამუშაო:

ამოცანა 1:

1) დაწერეთ ძებნის ალგორითმი, რომელიც დაადგენს არის თუ არა მოცემულ v ვექტორში მოცემული b-ს ტოლი ელემენტი? დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი -1;
2) მოიყვანეთ შესაბამისი ფუნქციის კოდი C++ ენაზე;
3) აჩვენეთ, თუ რა სახით მოხდება ამ ფუნქციის გამოძახება ძირითად პროგრამაში რამდენიმე განსხვავებული ტიპის ვექტორისთვის.

ამოხსნა:

1) ძებნის ალგორითმის ფსევდოკოდი:

```
search(ვექტორი v, საძიებელი b)
    for(i=0; i< v.size(); i++)
        if (v[i] == b) return i;
    return -1;
```

2) ფსევდოკოდში არაა აუცილებელი დავაკონკრეტოთ თუ რა ტიპის ელემენტებისგან შედგება ვექტორი, რადგან იმპლემენტაცია დამოუკიდებელია ტიპისგან, საკმარისია მხოლოდ, რომ გადატვირთული იყოს"==" ოპერატორი . C++ ენა საშუალებას იძლევა, რომ ფუნქციას ერთ-ერთ არგუმენტად გადაეცეს ვექტორის ტიპი. ეს კეთდება **template**-ების გამოყენებით, რაც ქართულად შეგვიძლია ვთარგმნოთ როგორც ყალიბი, ან თარგი. ანალოგია სრულიად ბუნებრივია, რადგან როგორც ერთ ყალიბში შეგვიძლია ჩავასხათ სხვადასხვა მასალა, ან ერთი თარგით შეიძლება გამოიჭრას სხვადასხვა სახის ქსოვილისგან ერთი ფორმის ნაჭერი, ასევე, როდესაც სხვადასხვა ტიპისთვის ფუნქციის კოდი იდენტურია, იგი ცხადდება ყალიბად სპეციალური სინტაქსის საშუალებით. **template**-ების მოქმედების მექანიზმი საფუძვლიანად იქნება ახსნილი ობიექტზე ორიენტირებული პროგრამირების კურსში.

ზემოთ მოყვანილი ალგორითმის შესაბამისი კოდი C++ ენაზე არის:

```
template<typename T>
int search(vector<T>& v, T b)
{
    int i(0);
    for( ; i< v.size(); i++)
       if (v[i] == b) return i;
    return -1;
}
```

3) თუ x ვექტორი შედგება მთელი რიცხვებისგან და მთელ ვექტორში ვეძებთ პირველივე 100-ის ტოლი ელემენტის ინდექსს, მაშინ გამოძახებას აქვს სახე:

```
int ind = Search(x, 100);
```

თუ v ვექტორი შედგება სტრინგებისგან,

```
string s("Uuh!");
```

და გვინდა დავბეჭდოთ პირველივე s -ის ტოლი ელემენტის ინდექსი, მაშინ გამოძახებას აქვს სახე:

```
cout << "index: " << search(str,s) << endl;
```

ამოცანა 2:

მოიყვანეთ შესაბამისი ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა მოცემულ v ვექტორში მოცემული p თვისების მქონე ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი -1. მოიყვანეთ ფუნქციის გამოძახების რამდენიმე ვარიანტი სხვადასხვა ტიპის ვექტორებისთვის.

შენიშვნა: ჩვეულებრივ, ტერმინი **თვისება** ნიშნავს ერთი ცვლადის ფუნქციას, რომლის მნიშვნელობათა სიმრავლე არის **bool**.

ამოხსნა:

```
template<typename T>
int search_if(vector<T>& v, bool p(T b))
{
    int i(0);
    for( ; i< v.size(); i++)
       if (p(v[i])) return i;
    return -1;
}
```

მაგალითად, თუ

```
bool f(int n)
{
    return (n>100);
}
```

ძირითად ფუნქციაში შეტყობინება

```
cout << "index: " << search_if(v,f) << endl;
```

დაბეჭდავს პირველივე 100-ზე მეტი რიცხვის ინდექსს, ხოლო თუ ასეთი არაა, -1-ს. ხოლო თუ

```
bool g(int n)
{
    return (n%2 == 1);
}
```

შეტყობინება

```
cout << "index: " << search_if(v,g) << endl;
```

დაბეჭდავს პირველივე კენტი რიცხვის ინდექსს, ხოლო თუ ასეთი არაა, -1-ს.

ფუნქციის არგუმენტად თვისების გამოყენება საშუალებას იძლევა შევქმნათ უფრო ზოგადი ალგორითმები. ჩვენ ვიცით რამდენიმე მაგალითი, როდესაც სტანდარტული ბიბლიოთეკის ალგორითმს ვაწვდიდით ჩვენს მიერ შექმნილ თვისებებს. მაგრამ ეს იმდენად მნიშვნელოვანი საკითხია, რომ სტანდარტული ბიბლიოთეკა თვითონ გვაძლევს საშუალებას ადვილად შევქმნათ შედარებით ხშირად გამოყენებადი თვისებები. გასათვალისწინებელია, რომ ეს გზა იძლევა საიმედო და ძალიან ეფექტურ კოდს. ტექმნიკურად ეს კეთდება ე.წ. binder-ების (ამკინძავების) საშუალებით. განვიხილოთ მაგალითები.

ამოცანა 3:

count_if ალგორითმის გამოყენებით დათვალეთ მთელ v ვექტორში, რომელშიც წერია რიცხვები

34, 100, 783, 22, 33, 54, 92, 100, 21,

ა) 100-ზე მეტი ელემენტების რაოდენობა;
ბ) 100-ზე მეტი ან ტოლი ელემენტების რაოდენობა;
გ) 100-ზე ნაკლები ელემენტების რაოდენობა;
დ) 100-ზე ნაკლები ან ტოლი ელემენტების რაოდენობა;

შენიშვნა: ვისარგებლოთ სტანდარტული ბიბლიოთეკის საშუალებებით, და ჩვენთვის ცნობილი ორადგილიანი დამოკიდებულებიდან (`less, less_equal, greater, greater_equal`) შევქმნათ ერთადგილიანი დამოკიდებულებები, ანუ თვისებები.

ამოხსნა: ნებისმიერი ორი მთელი `a, b` რიცხვებისთვის, დამოკიდებულება `greater<int>()` ნიშნავს, რომ `a>b.` თუ ჩვენ დავაფიქსირებთ მეორეს მნიშვნელობას, ვთქვათ `b`=100, მაშინ `greater<int>()` დაფიქსირებული მეორე არგუმენტით დაგაიქვეცა ერთადგილიან დამოკიდებულებად, რომლის მნიშვნელობა ყოველი `a`-სთვის ტოლია (`a`>100)-ის. ე.ი. ან ჭეშმარიტია ან მცდარი. ამიტომ, შეტყობინებების

```
int k = count_if(v.begin(),v.end(),bind2nd(greater<int>(),100));
cout <<"100-ze metia: " << k <<endl;
```

შესრულების შემდეგ `k` გახდება 1-ის ტოლი. აქ `bind2nd` ნიშნავს მეორე არგუმენტის დაფიქსირებას.

ერთადგილიანი დამოკიდებულებისთვის შეგვიძლია გამოვიყენოთ ნეგატორი `not1()`, რომელიც შეაბრუნებს მის მნიშვნელობას. ამ შემთხვევაში, 100-ზე მეტის შებრუნებული არის ნაკლები ან ტოლი 100-ზე. ამიტომ, შემდეგი კოდის

```
int k = count_if(v.begin(),v.end(),  not1(bind2nd(greater<int>(),100)));
cout <<"100-ze naklebia an toli: " << k <<endl;
```

შესრულების შემდეგ `k` გახდება 8-ის ტოლი.

იგივე ეფექტის მიღწევა შეგვიძლია უფრო იოლად, თუ პირდაპირ გამოვიყენებთ `less_equal`-ს. ანალოგიურად, 100-ზე მეტი ან ტოლის რაოდენობის გასაგებად შეგვიძლია გამოვიყენოთ ან უშუალოდ `greater_equal`, ან უფრო რთული გზა.

შენიშვნა: სტანდარტული ალგორითმები მუშაობენ ხელით შექმნილ თვისებებთან, მაგრამ პრიმიტიული მეთოდით შექმნილი ფუნქციები, ისეთები როგორც ჩვენს მიერ შექმნილი `search_if`, არ მუშაობენ სტანდარტულ ორადგილიან დამოკიდებულებებთან, ადაპტორებთან და ნეგატორებთან. ასეთ თავსებადობას რომ მივაღწიოთ, ამისთვის საჭიროა ჩვენს მიერ შექმნილი ფუნქცია იყოს განზოგადებული (generic) იმ აზრით, როგორც ეს STL ბიბლიოთეკაში არის მიღებული.

ამოცანები დამოუკიდებელი მუშაობისთვის:

1. პირველ და მეორე ამოცანაში, ძებნის ფუნქციები გადააკეთეთ ისე, რომ მათ მოძებნონ საძიებელი ელემენტი არა მთელ ვექტორში, არამედ მის ფრაგმენტში, რომლის მარცხენა ინდექსი არის `l` (ელ), ხოლო მარჯვენა ინდექსი არის r(ერ).
2. გამოიძახეთ დავალება 1-ის ფუნქცია რამდენიმე სხვადასხვა ტიპის ვექტორისთვის და სხვადასხვა ფრაგმენტისთვის (მაგალითად, ვექტორის პირველი და მეორე ნახევრებისთვის ცალ-ცალკე).
3. შექმენით ფუნქციები, რომლებიც გაარკვევს არის თუ არა მთელი რიცხვი სამის, ხუთის და 11-ის ჯერადი. გამოიყენეთ ისინი ძებნის ფუნქციებში.
4. დაწერეთ ფუნქცია, რომელიც დაითვლის ამ ვექტორში `l`–დან `r`–მდე (ჩათვლით) ინდექსების მქონე ელემენტებს შორის მაქსიმალურის ინდექსს.
5. მოცემულია ნამდვილი ან მთელი რიცხვების ვექტორი. დაწერეთ ფუნქცია, რომელიც დაითვლის ამ ვექტორში `l`–დან `r`–მდე (ჩათვლით) ინდექსების მქონე ელემენტების
   - ჯამს;
   - კვადრატების ჯამს;
   - შებრუნებული ელემენტების ჯამს (ვიგულისხმოთ რომ არანულოვანებია);
   - საშუალო არითმეტიკულს.

პრაქტიკული მეცადინეობა 2

სააუდიტორიო სამუშაო:

ამოცანა 1:

მოიყვანეთ ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა მოცემულ v ვექტორში მოცემული p თვისების მქონე ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის ინდექსი, წინააღმდეგ შემთხვევაში კი -1. მოიყვანეთ ფუნქციის გამოძახების რამდენიმე ვარიანტი სხვადასხვა ტიპის ვექტორებისთვის. p თვისებას შესაძლოა წარმოადგენდეს როგორც მომხმარებლის მიერ შექმნილი ბულის ტიპის უნარული (ანუ ერთადგილიანი) ფუნქცია, ასევე ფუნქციის ადაბტორი.

ამოხსნა: ფუნქციას შესაძლოა ჰქონდეს სახე:

```
template<typename T, typename Predicate>
int my_search_if(const vector<T>& v, Predicate p)
{
    int i = 0;
    while(i != v.size())
    {
        if (p(v[i]) )
              return i;
        ++i;
    }
    return -1;
}
```

შეგვეძლო ეს ფუნქცია დაგვეწერა როგორც წინა მეცადინეობაზე, `for` შეტყობინებით. მაგრამ მომდევნო ამოცანაში განსხვავება ამ ორ შეტყობინებას შორის არსებითია. ამიტომ ცნობილ ალგორითმზე გადავიმეოროთ ასეთი მიდგომა.

მაგალითად, თუ ძირითად ფუნქციაში გვაქვს შექმნილი ორი ვექტორი:

```
vector<int> v;
```

და

```
vector<double> vec;
```

და ორივე შევსილია ელემენტებით, მაშინ კოდი:

```
int i(0);
    i = my_search_if(v, bind2nd(greater<int>(),100));
    if(i != -1)
        cout << v[i] << endl;
    else
        cout << "not found!"<<endl;
```

დაბეჭდავს v ვექტორში მარცნიდან პირველივე ასზე მეტ რიცხვს, თუ ასეთი არის, ან დაბეჭდავს გზავნილს რომ ასეთი რიცხვები ვერ მოიძებნა.

ანალოგიურად, კოდი

```
i = my_search_if(vec, not1(bind2nd(less<double>(),11.9)));
    if(i != -1)
        cout << v[i] << endl;
    else
        cout << "not found!"<<endl;
```

დაბეჭდავს `vec` ვექტორში მარცნიდან პირველივე `11.9`-ზე ნაკლებ რიცხვს, თუ ასეთი არის, ან დაბეჭდავს გზავნილს რომ ასეთი რიცხვები ვერ მოიძებნა.

ამავე დროს თუ ჩვენ უკვე შექმნილი გვაქვს პრედიკატი

```
bool g(int n)
{
```

```
    return (n%3 == 0);
}
```
მაშინ კოდი
```
i = my_search_if(v, g);
    if(i != -1)
        cout << v[i] << endl;
    else
        cout << "not found!"<<endl;
```
დაბეჭდავს v ვექტორში მარცნიდან პირველივე სამის ჯერად რიცხვს, თუ ასეთი არის, ან დაბეჭდავს გზავნილს რომ ასეთი რიცხვები ვერ მოიძებნა.

ამოცანა 2:

შექმენით იტერაციული ფუნქცია ვექტორში l–დან r–მდე (ჩათვლით) ინდექსების მქონე ელემენტების ჯამის განსაზღვრისთვის (0≤l≤r). შემდეგ ეს ფუნქცია გარდაქმენით ეკვივალენტურ რეკურსიულ ფუნქციად.

ამოხსნა: დავწეროთ ისეთი იტერაციული კოდი, რომ ფუნქციის ტანი მხოლოდ ერთი **while**(true) შეტყობინებისგან შედგებოდეს:
```
template<typename T>
T mySumIter(const vector<T>& v, int l, int r)
{
    T s(0);
    while(true)
    {
        if (l>r)
            return s;
        s += v[l];
        ++l;
    }
}
```
იტერაციული ფუნქციის გარდაქმნა რეკურსიულად ხდება შემდეგი მიმდევრობით: **while** შეტყობინება შევცვალოთ ფუნქციის გამოძახებით, ოღონდ თუ **while** იყო პირველი შესრულებადი შეტყობინება, ფუნქციის გამოძახება გახდება ბოლო შესრულებადი. ოღონდ ახლა T s(0); შეტყობინება რეკურსიულ ფუნქციაში არ იქნება საუკეთესო არჩევანი, ყოველ გაშვებაზე რეკურსიული ფუნქცია შექმნის ახლ ლოკალურ ცვლადს ერთი და იგივე სახელით. გამოსავალი მდგომარეობს სტატიკური ცვლადის შექმნაში, რომლის ინიციალიზაცია (საწყისი მნიშვნელობის მინიჭება) მოხდება მხოლოდ ერთხელ, პირველი გაშვების დროს:
```
template<typename T>
T mySumRec(const vector<T>& v, int l, int r)
{
     static T s(0);
     if(l>r)
       return s;
     s+=v[l];
     l++;
     return mySumRec(v, l, r);
}
```

ამოცანები დამოუკიდებელი მუშაობისთვის:

1. განხილულ ამოცანაში, ძებნის ფუნქცია გადააკეთეთ ისე, რომ მათ მოძებნოს საძიებელი თვისების მქონე ელემენტი არა მთელ ვექტორში, არამედ მის ფრაგმენტში, რომლის მარცხენა ინდექსი არის l (ელ), ხოლო მარჯვენა ინდექსი არის r(ერ).

2. შექმენით ფუნქცია, რომელიც დაბეჭდავს სხვადასხვა ტიპის ვექტორის ელემენტებს `m` სვეტად. ვექტორის სახელი და სვეტების რაოდენობა უნდა იყოს ფუნქციის პარამეტრები.
3. შექმენით ფუნქცია, რომელიც `m` სვეტად დაბეჭდავს სხვადასხვა ტიპის ვექტორის ისეთ ელემენტებს, რომელიბიც აკმაყოფილებენ გარკვეულ თვისებას. ვექტორის სახელი, სვეტების რაოდენობა და თვისება უნდა იყოს ფუნქციის პარამეტრები.
4. შექმენით ფუნქცია, რომელიც შეავსებს სხვადასხვა ტიპის ვექტორებს კლავიატურიდან ან ფაილიდან.
5. მოცემულია ნამდვილი ან მთელი რიცხვების ვექტორი. დაწერეთ ფუნქცია, რომელიც რეკურსიულად დაითვლის ამ ვექტორში `l`–დან `r`–მდე (ჩათვლით) ინდექსების მქონე ელემენტების
    - კვადრატების ჯამს;
    - შებრუნებული ელემენტების ჯამს (ვიგულისხმოთ რომ არანულოვანებია).

პრაქტიკული მეცადინეობა 3

სააუდიტორიო სამუშაო:

ამოცანა 1:

მოიყვანეთ ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა კონტეინერის [first, last) ფრაგმენტში (დიაპაზონში, in range) მოცემული სიდიდე. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის იტერატორი, წინააღმდეგ შემთხვევაში კი last. მოიყვანეთ ფუნქციის გამოძახების რამდენიმე ვარიანტი სხვადასხვა ტიპის ობიექტების შესანახი სხვადასხვა კონტეინერისთვის.

ამოხსნა: იტერატორის მნიშვნელობას, ისევე როგორც პოინტერის მნიშვნელობას, წარმოადგენს მისამართი. ამიტომ, პოინტერის მსგავსად იტერატორსაც აქვს ირიბი მნიშვნელობა, რომლისთვისაც იგივე ოპერატორი - ″*″ გამოიყენება. განსხვავება არის ++ ოპერატორის მოქმედების პრინციპში, ნებისმიერი it იტერატორისთვის ამ მისამართზე არის მოთვსებული რაღაც ობიექტი, ხოლო it++ არის მისამართი, რომელზეც მოთავსებულია კონტეინერში ამ ობიექტის მომდევნო ობიექტი.

ჩვენს ფუნქციას შესაძლოა ჰქონდეს სახე:

```
template<typename ForwardIterator, typename T>
ForwardIterator my_search(ForwardIterator first, ForwardIterator last, T
b)
{
    while(first != last)
    {
        if (*first == b)
           return first;
        ++first;
    }
    return last;
}
```

მაგალითად, მთავარ ფუნქციაში შესაძლოა ასე გამოვიყენოთ ეს ფუნქცია:

```
    vector<int> v;
    v.push_back(11);
    v.push_back(21);
    v.push_back(541);

    vector<int>:: iterator it;
    it = my_search(v.begin(), v.end(), 5421);
    if(it != v.end())
        cout << "vector - number found!  " <<*it<< endl;
    else
        cout << "vector - number not found!"<<endl;

    list<double> lst;
    lst.push_back(11.88);
    lst.push_back(21.4);
    lst.push_back(541.9);

    list<double>:: iterator itLst;

    itLst = my_search(lst.begin(), lst.end(), 541.9);
    if(itLst != lst.end())
        cout << "list -  number found!  " << *itLst<< endl;
    else
        cout << "list - not found!"<<endl;
```

ამოცანა 2:

მოიყვანეთ ფუნქციის კოდი, რომელიც დაადგენს არის თუ არა კონტეინერის `[first, last)` ფრაგმენტში ანუ დიაპაზონში (range) მოცემული თვისების მქონე ელემენტი. დადებითი პასუხის შემთხვევაში ალგორითმმა უნდა დააბრუნოს პირველივე (დასაწყისიდან) ასეთი ელემენტის იტერატორი, წინააღმდეგ შემთხვევაში კი `last`. მოიყვანეთ ფუნქციის გამოძახების რამდენიმე ვარიანტი სხვადასხვა ტიპის ობიექტების შესანახი სხვადასხვა კონტეინერისთვის. p თვისებას შესაძლოა წარმოადგენდეს როგორც მომხმარებლის მიერ შექმნილი ბულის ტიპის უნარული (ანუ ერთადგილიანი) ფუნქცია, ასევე ფუნქციის ადაბტორი.

ამოხსნა: ფუნქციის ერთი შესაძლო ვარიანტი ასეთია:

```
template<typename ForwardIterator, typename Predicate>
ForwardIterator my_search_if(ForwardIterator first, ForwardIterator last,
Predicate pred)
{
    while(first != last)
    {
        if (pred(*first) )
           return first;
        ++first;
    }
    return last;
}
```

წინა ამოცანის მთავარი ფუნქციის აღნიშვნებში, გამოძახებას შესაძლოა ჰქონდეს სახე:

```
it = my_search_if(v.begin(), v.end(), not1(bind2nd(greater<int>(),11)));
if(it != v.end())
     cout << "vector - needed number found!  " <<*it<< endl;
else
     cout << "not found!"<<endl;
```

ამოცანები დამოუკიდებელი მუშაობისთვის:

1. შექმენით ფუნქცია, რომელიც დაბეჭდავს კონტეინერის `[first, last)` ფრაგმენტის ელემენტებს m სვეტად. იტერატორის ტიპის შერჩევისთვის ისარგებლეთ ლექციაში მოყვანილი მასალით.
2. შექმენით ფუნქცია, რომელიც m სვეტად დაბეჭდავს კონტეინერის `[first, last)` ფრაგმენტიდან ისეთ ელემენტებს, რომლებიც აკმაყოფილებენ გარკვეულ თვისებას.
3. დაწერეთ ფუნქცია, რომელიც კონტეინერის `[first, last)` ფრაგმენტის გარკვეული თვისების მქონე ელემენტებს შორის მაქსიმალურს და დაარუნებს მის იტერატორს. თუ ასეთი არ არის, ფუნქციამ უნდა დააბრუნოს იტერატორი `last`.
4. (*) დაწერეთ ფუნქცია, რომელიც კონტეინერის `[first, last)` ფრაგმენტის ელემენტში მოძებნის მაქსიმალურ ელემენტს და დაარუნებს მის დაწერეთ ფუნქცია, რომელიც კონტეინერის `[first, last)` ფრაგმენტის ელემენტში მოძებნის მაქსიმალურ ელემენტს და დაარუნებს მის ინდექსს.
5. დაწერეთ ფუნქცია, რომელიც დაითვლის კონტეინერის `[first, last)` ფრაგმენტის
    - ელემენტების ჯამს;
    - კვადრატების ჯამს;
    - შებრუნებული ელემენტების ჯამს (ვიგულისხმოთ რომ არანულოვანებია).

პრაქტიკული მეცადინეობა 4

სააუდიტორიო სამუშაო:

ამოცანა 1:

მოიყვანეთ იტერაციული ალგორითმის კოდი, რომელიც შექმნის კონტეინერის `[first, last)` ფრაგმენტის ასლს ისე რომ ახალი ფრაგმენტის მისამართი დაიწყება პარამეტრად მიღებული `result` იტერატორის მნიშვნელობიდან, გამოვიყენოთ ეს ფუნქცია `main()` -ში რამდენიმე განსხვავებული კონტეინერისთვის. კონტეინერის ბეჭვდისთვის გამოვიყენოთ იგივე ფუნქციის გადატვირთული ვერსია, რომელიც მუშაობს `ostream_iterator` -თან.

გარდაქმენით ალგორითმი რეკურსიულად და მოიყვანეთ შესაბამისი კოდი.

ამოხსნა: ფუნქციის ერთი შესაძლო ვარიანტი ასეთია:

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy0(InputIterator first, InputIterator last,
OutputIterator result)
{
    while(first != last)
    {
        *result = *first;
        ++first;
        ++result;
    }
    return result;
}
```

რადგან ეს ალგორითმი `copy` სახელით უკვე არსებობს შესაბამის ბიბლიოთეკაში, ამიტომ მას ვარქმევთ ოდნავ შეცვლილ სახელს, რომ არ გამოვიწვიოთ სახელების კონფლიქტი.

გარდავქმნათ ეს ალგორითმი ისე, რომ მისი ტანი შედგებოდეს ერთი **`while`** შეტყობინებისგან:

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copy0(InputIterator first, InputIterator last,
OutputIterator result)
{
    while(true)
    {
        if(first == last)
            return result;
        *result = *first;
        ++first;
        ++result;
    }
}
```

ახლა, საკმარისია მოვაშოროთ განმეორების შეტყობინება და დავუმატოთ რეკურსიული გამოძახება:

```
template<typename InputIterator, typename OutputIterator>
OutputIterator copyRec(InputIterator first, InputIterator last,
OutputIterator result)
{
     if(first == last)
         return result;
     *result = *first;
     ++first;
     ++result;
     copyRec(first, last, result);
}
```

ფუნქცია დრაივერს, რომელიც გამოიყენებს ამ კოდს, შესაძლოა ჰქონდეს სახე:

```
int main()
{
    vector<int> v;
    v.push_back(11);
    v.push_back(21);
    v.push_back(541);

    vector<int> vec(7);
    copyRec(v.begin(), v.end(),vec.begin()+2);

    ostream_iterator<int> kout(cout, "\t");
    copy(vec.begin(), vec.end(),kout);
    cout << endl;


    list<double> lst;
    lst.push_back(11.88);
    lst.push_back(21.4);
    lst.push_back(541.9);

    ostream_iterator<double> out(cout, "\t");
    copy(lst.begin(), lst.end(), out);
    cout << endl;

    system("PAUSE");
    return 0;
}
```

როგორც ვხედავთ, ძალიან მოხერხებულია, რომ კონტეინერის ბეჭდისთვის გამოვიყენოთ იგივე ალგორითმის გადატვირთული ვერსია, რომელიც მუშაობს `ostream_iterator`-თან.

ამოცანა 2. მოიყვანეთ ჩანაცვლების იტერაციული ალგორითმის კოდი, რომელიც `[first, last)` ფრაგმენტში `x`-ის ტოლ ობიექტებს შეცვლის `y`-ის ტოლი ობიექტებით.

გარდაქმენით ალგორითმი რეკურსიულად და მოიყვანეთ შესაბამისი კოდი.

ამოხსნა: ამ ფუნქციის ერთი ვარიანტი, რომელიც იმუშვებს მუდმივ მესამე და მეოთხე პარამეტრებთან, არის:

```
template<typename ForwardIterator, typename T>
void replace(ForwardIterator first, ForwardIterator last, T x,  T y)
{
    while(first != last)
    {
        if(*first == x)
          *first = y;
        ++first;
    }
}
```

მისი გარდაქმნის შედეგია:

```
template<typename ForwardIterator, typename T>
void replace0(ForwardIterator first, ForwardIterator last, T x,  T y)
{
    while(true)
    {
        if(first == last) return;
```

```
        if(*first == x)
          *first = y;
        ++first;
    }
}
```

საბოლოოდ:

```
template<typename ForwardIterator, typename T>
void replaceRec(ForwardIterator first, ForwardIterator last, T x,  T y)
{

     if(first == last) return;
     if(*first == x)
        *first = y;
     ++first;
     replaceRec(first,last, x, y);
}
```

ამოცანა 3: შექმენით ფუნქცია, რომელიც მთელი რიცხვების ვექტორში მთელ არაუარყოფით რიცხვს ჩაწერს იმ ელემენტად რისი ტოლიც არის ამ რიცხვის მნიშვნელობა (მაგალითად,რიცხვი k უნდა შევინახოთ იმ ელემეტად, რომლის ინდექსი არის k).

ვიგულისხმოთ, რომ თავდაპირველად შექმნილილა 100 ელემენტიანი ვექტორი და იგი ინიციალიზებულია ნულებით.

ვთქვათ, ვექტორში ამ წესით ჩავწერეთ რამდენიმე მთელი რიცხვი. როგორ შეგვიძლია მოვძებნოთ მაქსიმალური ელემენტი ჩაწერის სპეციფიკის გათვალისწინებით? როგორ შეგვიძლია გავარკვიოთ, უკვე ჩავწერეთ თუ არა მოცემული რიცხვი ამ ვექტორში? რა იქნება ამ ოპერაციების სისწარაფე?

რა ნაკლი ექნება ასე ორგანიზებულ შენახვას? როგორი გამოსავალი შეგვიძლია მოვიფიქროთ?

ამოხსნა: რიცხვი k უნდა შევინახოთ იმ ელემეტად, რომლის ინდექსი არის k.

მაქსიმალური ელემენტის ძებნას ვიწყებთ მარჯვნიდან. ამ ალგორითმის სისწრაფის შეფასება უარეს შემთხვევაში იგივეა, რაც მაქსიმალის განსაზღვრის ჩვეულებრივი ალგორითმისთვის. მაგრამ ამ შემთხვევაში გვაქვს საუკეთესო შემთხვევა, როცა 100 იყო ჩაწერილი. ძებნის ალგორითმი არის $O(1)$ რიგის.

ამოცანები დამოუკიდებელი მუშაობისთვის:

1: შექმენით ასლის შექმნის ალგორითმის გადატვირთული იტერაციული ვერსია, რომელიც მოცემული დიაპაზონიდან შექმნის მხოლოდ გარკვეული თვისებების მქონე ობიექტების ასლს. გარდაქმენით იგი რეკურსიულად.

2: შექმენით ჩანაცვლების ალგორითმის გადატვირთული იტერაციული ვერსია, რომელიც მოცემული დიაპაზონიდან გარკვეული თვისებების მქონე ობიექტებს ჩაანაცვლებს რომელიმე ფიქსირებული ობიექტით. შემდეგ გარდაქმენით ეს ალგოტიღი რეკურსიულ ალგორითმად.

3: მოიფიქრეთ პრაქტიკული სიტუაცია, როდესაც ასეთი ძებნა, ამ ან მსგავს ამოცანაში შესაძლოა ეფექტური აღმოჩნდეს.

4: ვთქვათ, ფაილში “numbers.txt” წერია რამდენიმე ნამდვილი რიცხვი სამკუთხედის სახით, ანუ პირველ სტრიქონში ერთი, მეორეში ორი, მესამეში სამი და ა.შ. მოიფიქრეთ რამდენიმე მონაცემთა სტრუქტურა, რომელშიც შეიძლება შევინახოთ ეს რიცხვები. რომელი სტრუქტურები უნდა გამოვიყენოთ, თუ გვინდა ელემენტებთან ვიმუშავოთ აუცილებლად ინდექსების საშუალებით? რა დამატებით საშუალებებს გვიქნის იტერატორების გამოყენება?

პრაქტიკული მეცადინება 7

სააუდიტორიო სამუშაო:

ამოცანა 1. მთელი არაუარყოფით რიცხვების a ვექტორი

i	0	1	2	3	4	5	6	7	8
a[i]	3	5	0	3	3	1	5	3	2

დაახარისხეთ გადათვლის (CountingSort) ალგორითმით. აჩვენეთ როგორ იცვლება დამხმარე c ვექტორი, რომლის ელემენტების რაოდენობა არის k+1

(k=max{a[0],a[1],...,a[a.size()-1]}),

და საბოლოო b ვექტორები.

ამოხსნა: გადათვლით დახარისხების ალგორითმის იდეა შემდეგია: თუ კონტეინერის ნებისმიერი *x* ელემენტისათვის წინასწარ დავთვლით ამ კონტეინერის რამდენი ელემეტია *x* -ზე ნაკლები (ვთქვათ *m*), მაშინ ის შეგვიძლია საბოლოო კონტეინერში პირდაპირ ჩავწეროთ (*m* +1) – ე ადგილზე, ანუ ინდექსით *m*. თუ შემავალ კონტეინერში გვხვდება ერთმანეთის ტოლი რიცხვები, მაშინ დამატებით უნდა ვიზრუნოთ, რომ ტოლი რიცხვები ერთმანეთის მეზობლად და ძველი რიგის შენარჩუნებით განლაგდეს.

მოვიყვანოთ გადათვლით დახარისხების ალგორითმის ფსევდოკოდი და ამოცანის მონაცემიდან გამომდინარე თვალი გავადევნოთ ალგორითმის სტრიქონების მიმდევრობას.

ალგორითმის კოდს აქვს სახე:

```
 void countingSort(vector<int>& a, vector<int>& b)
 {
1    int k = *max_element(a.begin(), a.end());
2    k++;

3    vector<int> c(k);

4    for (int i=0; i< a.size(); i++)
5         c[a[i]]++;
6    for (int i=1; i<k; i++)
7         c[i] += c[i-1];

8    for (int i=a.size()-1; i>=0; i--)
     {
9        b[c[a[i]]-1] = a[i];
10       c[a[i]]--;
     }
 }
```

დამხმარე c ვექტორის ბოლო ინდექსია k=max{3, 5, 0, 3, 3,1, 5, 3, 2}=5, ამიტომ დამხმარე ვექტორი არის ექვსელემენტიანი. მასზე განაცხადი კეთდება სტრიქონში 3; ყურადღება მივაქციოთ, რომ განაცხადის გაკეთების მომენტშივე ხდება ვექტორის ინიციალიზაცია ნულებით. მაქსიმალური ელემენტის იტერატორის განსაზღვრისთვის ვიყენებთ STL-ის ალგორითმს max_element, რომლის რამდენიმე გადატვირთული ვერსია არსებობს.

i	0	1	2	3	4	5
c[i], განაცხადის გაკეთების შემდეგ	0	0	0	0	0	0
c[i], I for-ის შემდეგ	1	1	1	4	0	2
c[i], II for-ის შემდეგ	1	2	3	7	7	9

I for შეტყობინება საშუალებას გვაძლევს განვსაზღვროთ, თუ საწყისი ვექტორის თითოეული ელემენტი რამდენ ეკზემპლიარად გვხდება ამ ვექტორში, II for შეტყობინება საშუალებას გვაძლევს განვსაზღვროთ, თუ თითოეული ელემენტი რამდენ ელემენტზე არის მეტი ან ტოლი.

III for-ის მუშაობის შედეგი ბიჯების მიხედვით:

```
1) i=8,  a[8]=2,  c[a[8]]=c[2]=3 Ю b[3-1]=b[2]=2, c[2]=2;
2) i=7,  a[7]=3,  c[a[7]]=c[3]=7 Ю b[7-1]=b[6]=3, c[3]=6;
3) i=6,  a[6]=5,  c[a[6]]=c[5]=9 Ю b[9-1]=b[8]=5, c[5]=8;
4) i=5,  a[5]=1,  c[a[5]]=c[1]=2 Ю b[2-1]=b[1]=1, c[1]=1;
5) i=4,  a[4]=3,  c[a[4]]=c[3]=6 Ю b[6-1]=b[5]=3, c[3]=5;
6) i=3,  a[3]=3,  c[a[3]]=c[3]=5 Ю b[5-1]=b[4]=3, c[3]=4;
7) i=2,  a[2]=0,  c[a[2]]=c[0]=1 Ю b[1-1]=b[0]=0, c[0]=0;
8) i=1,  a[1]=5,  c[a[1]]=c[5]=8 Ю b[8-1]=b[7]=5, c[5]=7;
9) i=0,  a[0]=3,  c[a[0]]=c[3]=4 Ю b[4-1]=b[3]=3, c[3]=3;
```

b ვექტორი, 9-ელემენტიანი

i	0	1	2	3	4	5	6	7	8
i=8			**2**						
i=7			2				**3**		
i=6			2				3		**5**
i=5		**1**	2				3		5
i=4		1	2			**3**	3		5
i=3		1	2		**3**	3	3		5
i=2	**0**	1	2		3	3	3		5
i=1	0	1	2		3	3	3	**5**	5
i=0	0	1	2	**3**	3	3	3	5	5

c ვექტორი, 9-ელემენტიანი

1	2	3	7	7	9
1	2	**2**	7	7	9
1	2	2	**6**	7	9
1	2	2	**6**	7	8
1	**1**	2	6	7	**8**
1	1	2	**5**	7	8
1	1	2	**4**	7	8
0	1	2	4	7	8
0	1	2	**4**	7	**7**
0	1	2	**3**	7	7

ამოცანები დამოუკიდებელი მუშაობისთვის:

1. მოცემულია **მთელი არაუარყოფით** რიცხვების n =10 ელემენტიანი ვექტორი a[n]={2, 3, 1, 2, 3, 0, 5, 7, 1, 2 }. დაახარისხეთ იგი გადათვლის (CountingSort) ალგორითმით და აჩვენეთ ბიჯების მიხედვით როგორ იცვლება დამხმარე c ვექტორი,

(k=max{a[0],a[1],...,a[a.size()-1]}),

და საბოლოო b[n] ვექტორი.

2*. მოცემულია **მთელი** რიცხვების n ელემენტიანი a[n] ვექტორი,, რომელშიც

k1=min{a[0],a[1],..., a[a.size()-1]}<0, k2=max{a[0],a[1],..., a[a.size()-1]}.

დაწერეთ გადათვლით დახარისხების (CountingSort) ალგორითმის მოდიფიცირებული ვარიანტი, რომელიც გამოდგება k1<0 შემთხვევისთვის.

3. ალგორითმის რომელი ვერსიის გამოყენებაა მიზანშეწონილი, თუ გვინდა დავახარისხოთ 1000 მთელი რიცხვი, მოთავსებული [3002001, 3004002] შუალედში?

პრაქტიკული მეცადინება 11

ამოცანა 1. სწრაფი ვდომის *A* კონტეინრში წერია შემდეგი მთელი რიცხვები:

i	1	2	3	4	5	6	7	8	9	10	11
A[i]	12	40	15	67	3	24	31	10	16	77	13

კონტეინერის `[1,10)` ფრაგმენტი გარდაქმენით გროვად. გროვის აგება აჩვენეთ ფსევდოკოდის ძირითადი შეტყობინებების (ანუ სტრიქონების) მიმდევრობის მითითებით.

ამოხსნა. გროვის აგების ალგორითმის ფსევდოკოდია:

```
MAKE_HEAP(1,n+1)
1 | for (i = n/2; i>=1; i--)
2 |       | HEAPIFY(i,n+1);
```

რადგან `[1,10)` ფრაგმენტს გარავქმნით გროვად, ამიტომ `n = 9` და რეალურად გვაქვს ფსევდოკოდი

```
MAKE_HEAP(1,10)
1 | for (i = 4 ; i>=1; i--)
2 |       | HEAPIFY(i,10);
```

ხოლო სტრიქონების მიმდევრობა (ტრასირება) ასეთია:

i=4:

HEAPIFY(4,10);

l=8 ; r=9;

if (8<10 && 10>67) ☹ ⇒ largest=4; //შესრულდა else შეტყობინება

if (9<10 && 16>67) ☹ ⇒ largest არ შეცვლილა;

if (4 != 4) ☹ + // largest=i

i=3:

HEAPIFY(3,10)

l=6 ; r=7;

if (6 <10 && 24>15) ☺⇒ largest=6; //შესრულდა if შეტყობინება

if (7 <10 && 31>24); ☺⇒ largest=**7**; //შესრულდა if შეტყობინება

if (3!=7) ☺ // largest != i

15 Ъ 31 // a[3] Ъ a[7]; //ცვლილებები უნდა აისახოს კონტეინერში

HEAPIFY(**7,10**)

l=14 , r=15

if (14 <10 ...); ☹ ⇒ largest=7; //შესრულდა else შეტყობინება

if (15 <10 ...); ☹ ⇒ largest არ შეცვლილა;

if (7!=7) ☹ + //a[7] ფოთოლია

i=2:

HEAPIFY(2,10)

l=4 ; r=5;

if (4 <10 && 67>40); ☺ ⇒ largest=**4**; //შესრულდა if შეტყობინება

if (5 <10 && 3>67); ☹ ⇒ largest არ შეცვლილა;

if (2!=4) ☺ //largest != i

40 Ъ 67 //a[2] Ъ a[4]; //ცვლილებები უნდა აისახოს კონტეინერში

HEAPIFY(**4**)

l=8; ,r=9;

if (8 <10 && 10>40); ☹ ⇒ largest=4; //შესრულდა else შეტყობინება

if (9 <10 && 16>40); ☹⇒ largest არ შეცვლილა;

if (4!=4) ☹ + //largest=i

i=1:

HEAPIFY(1,10)

```
l=2; , r=3;
if (2 <10  && 67>12); ☺ ⇒ largest=2;
if (3 <10  && 31>67); ☹⇒ largest არ შეცვლილა;
if (1 != 2) ☺                                   //largest != i
    12 Ъ 67                                     //a[1] Ъ a[2];
    HEAPIFY(2,10)
        l=4 ,r=5
        if (4 <10  && 40>12); ☺⇒ largest=4;
        if (5 <10  && 3>40); ☹ ⇒ largest არ შეცვლილა;
        if (2!=4) ☺                             //largest != i
            12 Ъ 40                             //ცვლილებები უნდა აისახოს კონტეინერში
            HEAPIFY(4)
                l=8; r=9;
                if (8 <10  && 10>12); ☹⇒ largest=4;     //შესრულდა else შეტყობინება
                if (9 <10  && 16>12); ☺ ⇒ largest=9;
                if (4!=9) ☺                     //largest ≠ i
                        12 Ъ 16                 //ცვლილებები უნდა აისახოს კონტეინერში
                        HEAPIFY(9,10)
                            l=18;   r=19;
                            if (18 <10  ...); ☹ ⇒ largest=7;    //შესრულდა else შეტყობინება
                            if (19 <10  ...); ☹⇒ largest არ შეცვლილა;
                            if (9!=9)  ☹  +     //a[9] ფოთოლია
```

ამოცანა 2. მოცემული

i	1	2	3	4	5	6	7	8	9	10	11
A[i]	1	8	15	7	3	24	31	10	16	7	13

კონტეინერის `[1,9)` ფრაგმენტისგან ააგეთ გროვა ფსევდოკოდის ძირითადი შეტყობინებების (ანუ სტრიქონების) მიმდევრობის მითითებით.

ამოცანა 3. მოცემული

i	1	2	3	4	5	6	7	8	9	10	11
A[i]	12	40	15	17	83	24	31	10	16	7	93

კონტეინერის `[1,10)` ფრაგმენტისგან ააგეთ გროვა ფსევდოკოდის ძირითადი შეტყობინებების (ანუ სტრიქონების) მიმდევრობის მითითებით.

ამოცანა42. მოცემული

i	1	2	3	4	5	6	7	8	9	10	11
A[i]	12	40	15	67	3	4	31	10	16	71	13

კონტეინერის `[1,10)` ფრაგმენტისგან ააგეთ გროვა ფსევდოკოდის ძირითადი შეტყობინებების (ანუ სტრიქონების) მიმდევრობის მითითებით.

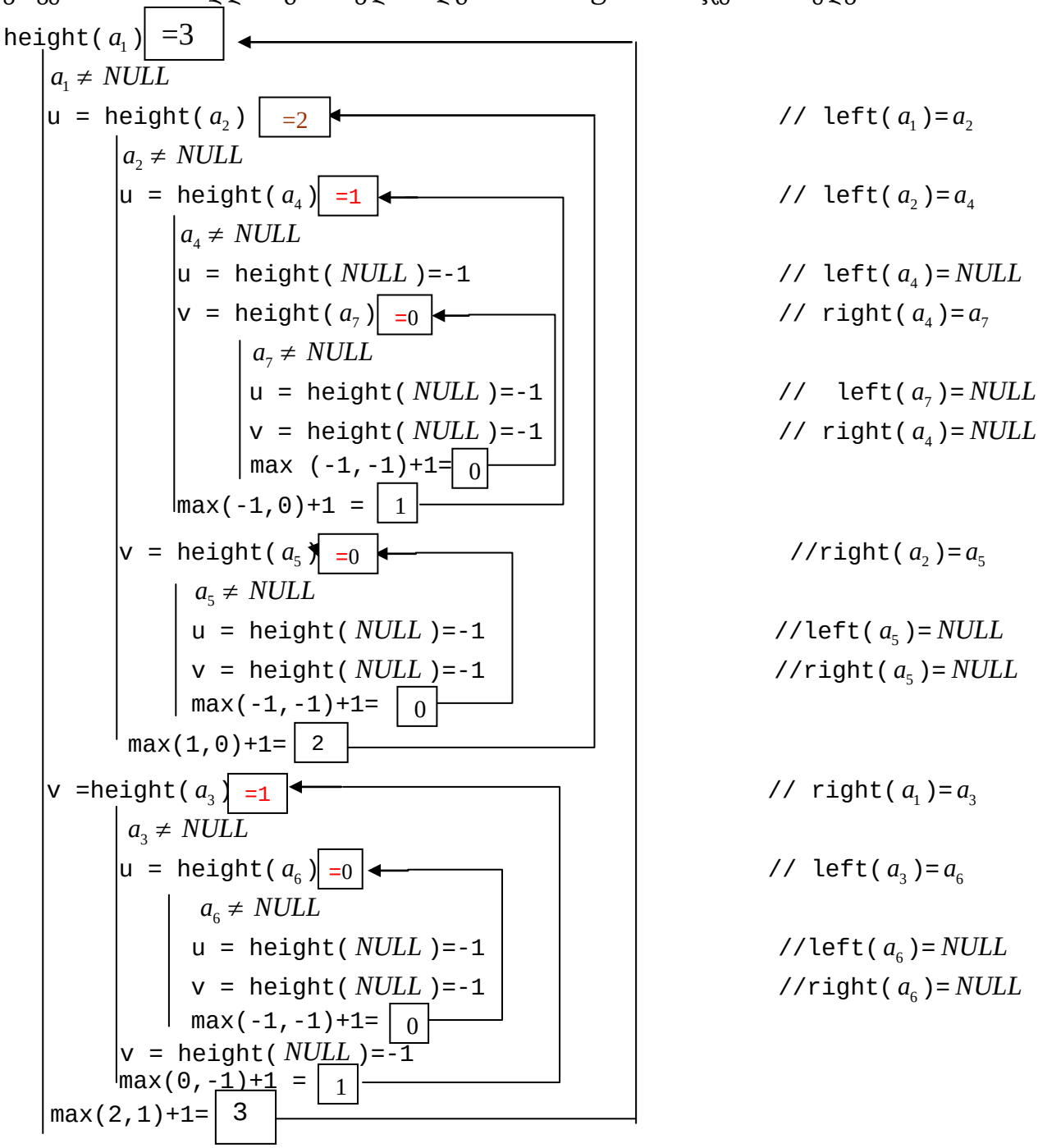
პრაქტიკული მეცადინება 12

ამოცანა 1. შემდეგი მონაცემებისგან ავაგოთ ძებნის ორობითი ხე: 47 51 23 15 40 49 21. ჩავთვალოთ, რომ პირველი მონაცემი მოთავსებულია a_1 მისამართზე, მეორე a_2-ზე, და ა.შ. შემდეგ, მოვიყვანოთ ხის სიმაღლის გამოთვლის ალგორითმის ფსევდოკოდი და ვაჩვენოთ მისი მუშაობა (ტრასირება) მოცემულ ხეზე შესრულებული სტრიქონების მიმდევრობის მითითებით.

ამოხსნა: ვისარგებლოთ ფსევდოკოდით:

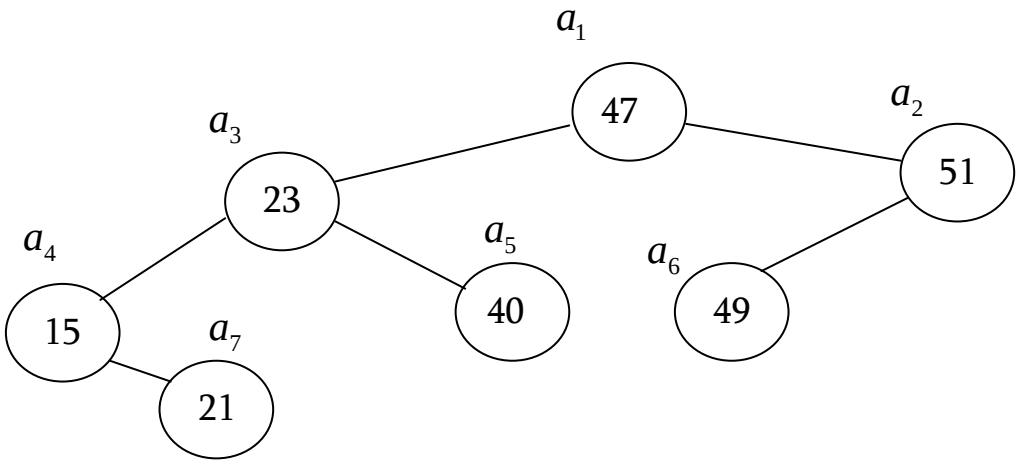
```
int  height (link x){
     if (x == NULL) return -1;
     int u = height (left(x));
     int v = height (right(x));
     return ( u < v )?( v +1):(u +1); }
```

ვაჩვენოთ ხის სიმაღლის გამოთვლის ალგორითმის მუშაობა ბიჯების მიხედვით:



ამოცანა 2. შემდეგი მონაცემებისგან ავაგოთ ძებნის ორობითი ხე: 47 51 23 15 40 49 21. ჩავთვალოთ, რომ პირველი მონაცემი მოთავსებულია a_1 მისამართზე, მეორე a_2-ზე, და ა.შ. შემდეგ, ვაჩვენოთ ST_search(a_1, 21) ალგორითმის მუშაობა ბიჯების მიხედვით. ვისარგებლოთ ფსევდოკოდით, რომელშიც ტიპი link აღნიშნავს წვეროს მისამართს:

```
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) ; }
```




```
ST_search(x = a1, k = 21)= a7
| if (NULL == a1 || 21 == 47) ☹
| if (21 < 47 ) ☺
|     return ST_search (x = a2, k = 21)= a7                    //left(a1)=a2
|            | if (NULL == a2 || 21 == 23) ☹
|            | if (21 < 23 ) ☺
|            |     return ST_search (x = a4, k = 21)= a7       //left(a2)=a4
|            |            | if( NULL ==a4 || 21 == 15) ☹
|            |            | if (21 < 15 ) ☹
|            |            | return ST_search (x = a7, k = 21)= a7
|            |            |        | if (NULL==a1 || 21= 21)  return   a7
```

სავარჯიშო: შემდეგი მონაცემებისგან:

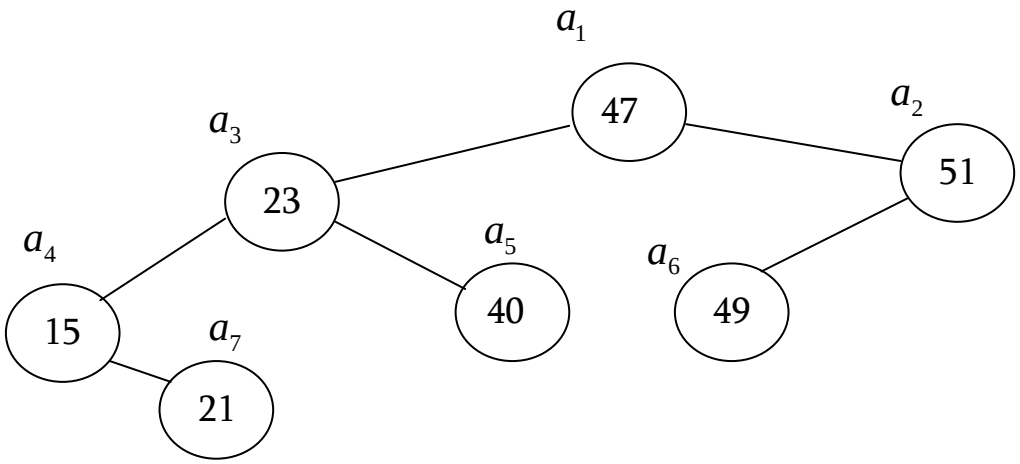
1. 7 51 23 115 4 1
2. 55 23 51 53 20

ააგეთ ძებნის ორობითი ხე. ჩათვალეთ, რომ პირველი მონაცემი მოთავსებულია a_1 მისამართზე, მეორე a_2-ზე, და ა.შ. მოიყვანეთ ხის სიმაღლის გამოთვლის ალგორითმის ფსევდოკოდი და აჩვენოთ მისი მუშაობა (ტრასირება) მოცემულ ხეზე შესრულებული სტრიქონების მიმდევრობის მითითებით. შემდეგ, აჩვენეთ ST_search(a_1,115) ალგორითმის მუშაობა (ტრასირება) ბიჯების სტრიქონების მიმდევრობის მითითებით.

პრაქტიკული მეცადინება 13

ამოცანა 2. დავხატოთ შემდეგი მონაცემებისგან აგებული ძებნის ორობითი ხე: 47 51 23 15 40 49 21. ჩავთვალოთ, რომ პირველი მონაცემი მოთავსებულია a_1 მისამართზე, მეორე a_2 -ზე, და ა.შ. შემდეგ, ჩავამატოთ ახალი z კვანძი გასაღებით 25. ვაჩვენოთ ST_insert(a_1 ,z) ალგორითმის მუშაობა ბიჯების მიხედვით.

ამოხსნა: შესაბამის ხეს აქვს სახე:



ვისარგებლოთ ფსევდოკოდით, რომელშიც ტიპი link აღნიშნავს წვეროს 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;
    }
```

ვაჩვენოთ ალგოირთმის მუშაობა სტრიქონების მიმდევრობის მითითებით:

ST_insert(a_1 ,z)

```
y = NULL;
x = a1;
while ( a1 ≠ NULL )
    y = a1;
    if(25<47)   x=a3;
while ( a3 ≠ NULL )
    y = a3;
    if(25<23)  ☹
```

```
    else x=a5;
while ( a5 ≠ NULL )
    y = a5;
    if(25<40)  x=NULL;
while ( NULL != NULL ) ☹
p(z) = a5;
if (NULL == y) ☹
else
if   (25 <  40)  left(y) = z;   //a5 მისამართზე left გახდა z
    return root;
```

ამოცანა 2. წინა ამოცანის ხისთვის, რისი ტოლია a_1-ის და a_5 -ის მომდევნო (გასაღებების სიდიდის აზრით) კვანძების მისამართები?

ამოხსნა: ვისარგებლოთ ფსევდოკოდით,

```
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);
}
```

ვაჩვენოთ ალგორითმის მუშაობა სტრიქონების მიმდევრობის მითითებით:

```
ST_ successor(x=a1)
    if ( NULL != a2 ) return ST_minimum( a2 )=a6;

ST_ successor(x=a5)
    if ( NULL != NULL) ☹
    while ( NULL != p( a5 )=a3 && a5 == right( a3 )) {
        x = a3;   // x და არა a5, რადგან a5 არის ე.წ. right-value
    while ( NULL != p( a3 )=a1  && a3 == right( a1 ))☹
    return  a1(= p( a3 ));
```

სავარჯიშო: შემდეგი მონაცემებისგან:

1. 7 51 23 115 4 1
2. 55 23 51 53 20

ააგეთ ძებნის ორობითი ხე. ჩათვალეთ, რომ პირველი მონაცემი მოთავსებულია a_1 მისამართზე, მეორე a_2-ზე, და ა.შ. მოიყვანეთ ახალი z კვანძის ჩამატების ალგორითმის ფსევდოკოდი და აჩვენეთ მისი მუშაობა (ტრასირება) მოცემულ ხეზე შესრულებული სტრიქონების მიმდევრობის მითითებით, თუ key(z) = 22 და key(z) = 48. შემდეგ, ახლად ჩასმული კვანძების მაგალითზე აჩვენეთ წინა და მომდევნო კვანძების განსაზღვრის ალგირითმის მუშაობა შესრულებული სტრიქონების მიმდევრობის მითითებით.