თემა 9. ჰეშირების ელემენტები

განხილული საკითხები:

- ჰეშირება
- პირდაპირი მიმართვის ცხრილი >>>
- ჰეშ-ფუნქციები >>>
- ჰეშირება ღია მისამართებით (open addressing) >>>
- ჰეშირება სიებით, ანუ ჯაჭვებით >>>
- დაუხარისხებელი ასოცირებული კონტეინერები >>>
- ლიტერატურა >>>

ჰეშირება

ჰეშირების მეთოდები საშუალებას გვაძლევს შევქმნათ მონაცემთა სტრუქტურები, რომლებშიც მხოლოდ სამი ოპერაცია: ელემენტის ჩამატება, ელემენტის წაშლა და ძებნა ხორციელდება, მაგრამ ხორციელდება სხვა სტრუქტურებთან შედარებით გაცილებით სწრაფად.

ჰეშირების იდეა მარტივია, ელემენტების (ჩანაწერების) მისამართები ინახება ცხრილში. ცხრილში ჩანაწერის ინდექსი გამოითვლება ჩანაწერის გასაღების მიხედვით. ჩანაწერი შეიძლება იყოს სხვადასხვა ტიპის ობიექტი, მაგრამ გასაღები ყოველთვის არის ნატურალური რიცხვი. გასაღებები უნიკალურია, ანუ ორი ჩანაწერის გასაღები არ შეიძლება ერთმანეთის ტოლი იყოს. ამიტომ გასაღებების მთლიანი რაოდენობა და თვითონ გასაღების სიდიდე შესაძლოა ბევრად დიდი იყოს, ვიდრე ცხრილის შესაძლო ზომა, რისი უზრუნველყოფაც შეუძლია სისტემას (ცხრილის ელემენტებზე მიმართვა უნდა ხდებოდეს ინდექსის საშუალებით, რაც ნიშნავს რომ ცხრილის დასაპროგრამებლად უნდა გამოვიყენოთ ისეთი კონტეინერი, რომლის იტერატორს აქვს სწრაფი წვდომა, ანუ ვექტორი, დეკი ან უბრალოდ მასივი).

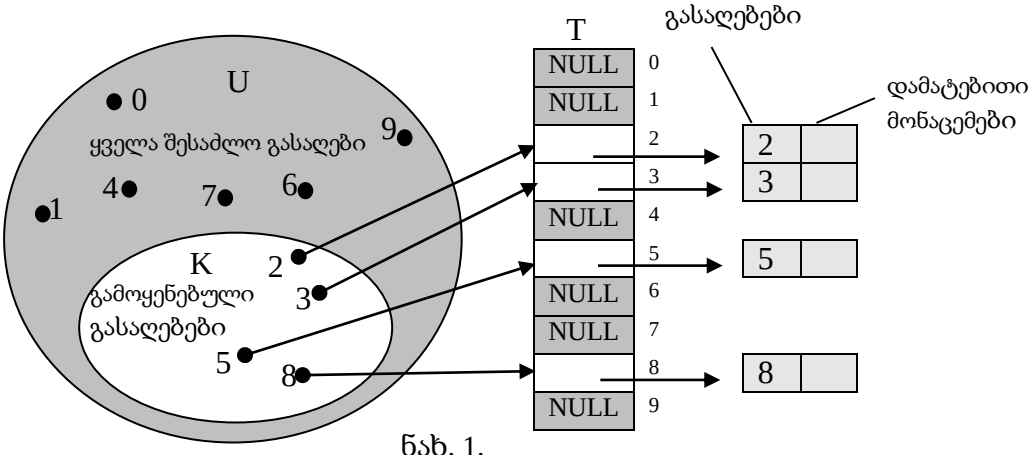
იდეალურ შემთხვევაში (კონკრეტულ ამოცანაში გამოსაყენებელი მეხსიერების მოცულობა შეზღუდული რომ არ იყოს) გასაღებებს შევინახავდით ამ გასაღების ტოლ მისამართზე (ინდექსით), მაშინ ნებისმიერი ელემენტის ძებნა განხორციელდებოდა მეხსიერებაზე ერთადერთი მიმართვით $O(1)$ დროში. მეხსიერებაში.

რადგან ასეთი იდეალური სიტუაცია პრაქტიკაში არ გვხვდება (გასაღებები, როგორც წესი ძალიან გრძელია), ამიტომ მივმართავთ ჰეშირებას და გასაღების მიხედვით ვითვლით ჩანაწერის (ანუ დინამიკური სიმრავლის ელემენტის) პოზიციას ჰეშ-ცხრილში. ამ დროს განირჩევა ორი შემთხვევა. ერთი, როდესაც აქტიური გასაღებების რაოდენობა ყოველ მომენტში ნაკლებია ცხრილის ზომაზე. ამ დროს ცხრილში ყოველთვის არსებობს თავისუფალი უჯრები ანუ ღია მისამართები (**ღია მისამართებიანი ჰეშ-ცხრილი, ჩაკეტილი ჰეშირება**), და მეორე, როდესაც აქტიური გასაღებების რაოდენობა აღემატება ცხრილის ზომას (**ჯაჭვებით ჰეშირება, ღია ჰეშირება**). საშუალედო შემთხვევები არ არსებობს, ისინი შეგვიძლია გამოვრიცხოთ ცხრილის ზომების შერჩევის ხარჯზე.

ბოლო ორ შემთხვევაში საჭიროა რაიმე ფორმულის საშუალებით $O(1)$ დროში განვსაზღვროთ ნებისმიერი ჩასამატებელი (ან საძებნი, ან წასაშლელი) გასაღების მისამართი ცხრილში. ამისათვის გამოიყენება ჰეშ-ფუნქციები, რომლებიც გასაღებებს გარდაქმნიან ცხრილის ინდექსებად. ტრადიციულად, ჰეშ ფუნქციის არგუმენტებს **გასაღებებს,** ხოლო კონკრეტულ გასაღებებზე ჰეშ ფუნქციის მნიშვნელობებს **ჰეშ-მნიშვნელობებს** ვუწოდებთ. რადგან გასაღებების მთლიანი რაოდენობა დიდია, ხოლო ცხრილის უჯრების რაოდენობა შედარებით მცირე, აუცილებლად იჩენს თავს ე.წ. კოლიზიები, ანუ სხვადასხვა გასაღებისთვის მათი შესაბამისი ჰეშ-მნიშვნელობის ტოლობა (დამთხვევა). კოლიზიების გადაჭრა ხდება აქტიური გასაღებების რაოდენობიდან გამომდინარე: ზოგად ჰეშ-ცხრილებში კოლიზიები გადაიჭრება **გადაბმის** (სიების, ჯაჭვების) საშუალებით, ხოლო ღია მისამართებიან ცხრილებში - მისამართების გადასინჯვის მეთოდებით.

<<< პირდაპირი მიმართვის ცხრილი

პირდაპირი მიმართვის ცხრილი ნიშნავს, რომ შესაძლო გასაღებთა რაოდენობა მცირეა და გასაღებების სიდიდე არ აღემატება ცხრილის ზომას, ანუ გასაღებებს წარმოადგენენ რიცხვები {0, 1, ..., m-1} სიმრავლიდან (m მცირე რიცხვია). სიმრავლის შესანახად გამოვიყენოთ m ელემენტიანი $T[m]$ მასივი, რომელსაც ეწოდება პირდაპირი მიმართვის ცხრილი (direct-address table). ყოველი პოზიცია, ან უჯრედი (position, slot) შეესაბამება გარკვეულ გასაღებს U სიმრავლიდან.



ნახ. 1.

ნახ. 1: დინამიკური სიმრავლის რეალიზაცია პირდაპირი მიმართვის T ცხრილის საშუალებით: U ={0,1,…,9} წარმოადგენს ყველა შესაძლო გასაღების სიმრავლეს. თითოეულ გასაღებს ამ სიმრავლიდან T ცხრილში შეესაბამება საკუთარი უჯრედი. ცხრილის 2, 3, 5 და 8 ნომრის შესაბამის პოზიციებში (ფაქტობრივად გამოყენებული გასაღებები) ჩაწერილია მიმთითებლები სიმრავლის ელემენტებზე, ხოლო ცხრილის გამოუყენებელ პოზიციებში (მუქი ფერითაა მოცემული) ჩაწერილია NULL.

აღვნიშნოთ `T[k]` – თი ცხრილის ის პოზიცია, რომელშიც იწერება *k* გასაღების მქონე ელემენტის (ჩანაწერის) `x` მისამართი. თუ k გასაღების მქონე ელემენტი ცხრილში არა გვაქვს, მაშინ `T[k]= NULL`. ლექსიკონური ოპერაციების რეალიზება ასე მოხდება:

```
DirectAddressSearch(T,k)
      return T[k];

DirectAddressInsert(T,x)
   T[key(x)]=x;

DirectAddressDelete(T,x)
    T[key(x)]= NULL;
```

თითოეული ამ ოპერაციიდან საჭიროებს O(1) დროს.

პრაქტიკაში, პირდაპირი მიმართვის ცხრილი იშვიათად გამოიყენება.

<<< ჰეშ-ფუნქციები

ჰეშ-ფუნქციის დანიშნულება არის, რომ გასაღებები (უნიკალური ნატურალური რიცხვები, ან მთლიანი ობიექტები) გარდაქმნას ერთგანზომილებიანი ცხრილის ინდექსებად, ანუ **მისამართებად.** ცხრილს რა კონტეინერით დავაპროგრამებთ ამ მომენტში მნიშვნელობა არ აქვს. ცხრილის დანომრვა იწყებება ნულიდან. მისი ზომა აღვნიშნოთ m-ით, თვითონ ცხრილი T-თი.

განვიხილოთ ჰეშ-ფუნქციის აგების ორი მეთოდი: ნაშთიანი გაყოფა და გამრავლება. ცხადია, არსებობს უფრო რთული და დახვეწილი მეთოდები, რომლებსაც ახლა არ განვიხილავთ. პრაქტიკაში მათი გამოყენება დამოკიდებულია ამოცანის სპეციფიკაზე. ზოგადად, კარგი ჰეშ-ფუნქციისაგან მოითხოვება, რომ მან უზრუნველყოს თანაბარი ჰეშირება. ჩვეულებისამებრ,

გულისხმობენ რომ ჰეშ-ფუნქციების განსაზღვრის არეში ნატურალური რიცხვებია. თუკი გასაღებები არ წარმოადგენენ ნატურალურ რიცხვებს, მათ გარდაქმნიან ასეთ სახემდე.

ნაშთიანი გაყოფის მეთოდი (division method) მდგომარეობს იმაში, რომ ჰეშ-ფუნქცია k სიდიდის მქონე გასაღებს შეუსაბამებს k-ს m- ზე გაყოფის ნაშთს:

`h(k)= k mod m,` ანუ `h(k)= k%m.`

მაგალითად, თუკი ჰეშ-ცხრილის ზომა m = 12 და გასაღები უდრის 100-ს, მაშინ შესაბამისი ჰეშ-მნიშვნელობა იქნება 4. m-ის გარკვეულ მნიშვნელობებს თავი უნდა ავარიდოთ. მაგალითად თუ $m = 2^p$, მაშინ `h(k)` წარმოადგენს k რიცხვის p უმცროს ბიტს. თუკი დარწმუნებული არა ვართ, რომ გასაღებთა უმცროსი ბიტები ერთნაირი სიხშირით შეგვხვდება, m რიცხვის როლში ორის ხარისხები არ უნდა ავიღოთ. არასასურველია m -ის მნიშვნელობად 10-ის ხარისხების არჩევა თუკი გასაღებები ათობით რიცხვებს წარმოადგენენ – ასეთ შემთხვევაში აღმოჩნდება, რომ გასაღებთა ციფრების ნაწილი მთლიანად განსაზღვრავს ჰეშ-მნიშვნელობებს.

კარგ შედეგებს იძლევა m -ის როლში ისეთი მარტივი რიცხვის აღება, რომელიც შორს დგას 2-ის ხარისხებისაგან. მაგალითად, თუ ჰეშ-ცხრილში შესატანია 2000-მდე ჩანაწერი და ჯაჭვში ანუ სიაში დაახლოებით სამი ვარიანტის გადარჩევა პრობლემას არ ჰქმნის ელემენტთა ძებნისას, m -ის მნიშვნელობად შეგვიძლია ავიღოთ 701. რიცხვი 701 მარტივია, $701 \approx 2000/3$ და ორის ხარისხებისგანაც შორს დგას. ამის გამო მიზანშეწონილია ავირჩიოთ ჰეშ-ფუნქცია

`h(k) = k mod 701.`

გამრავლების მეთოდი. (multiplication method) მდგომარეობს შემდეგში – ვთქვათ ჰეშ-მნიშვნელობების რაოდენობა m -ის ტოლია. დავაფიქსიროთ რაიმე A მუდმივი `0 < A < 1` და განვსაზღვროთ:

$$h(k) = \lfloor m \times ((kA) \bmod 1) \rfloor,$$

სადაც $(kA) \bmod 1$ წარმოადგენს kA -ს წილად ნაწილს.

გამრავლების მეთოდის ღირსება იმაში მდგომარეობს, რომ ჰეშ-ფუნქციის ეფექტურობა ნაკლებადაა დამოკიდებული m -ის არჩევაზე. ჩვეულებისამებრ m -ის როლში ირჩევენ ორის ხარისხს, რადგან კომპიუტერთა უმეტესობაში ასეთ m -ზე გამრავლება რეალიზდება როგორც სიტყვის წანაცვლება.

გამრავლების მეთოდი მუშაობს A მუდმივის ნებისმიერი მნიშვნელობისათვის, მაგრამ ზოგიერთმა მნიშვნელობამ შეიძლება უკეთესი შედეგი მოგვცეს. ითვლება, რომ

$$A \approx (\sqrt{5} - 1)/2 = 0.6180339887 \qquad (1)$$

მნიშვნელობა საკმაოდ ეფექტურია.

მაგალითად: k=123456, m=10000 და A განსაზღვრულია (1) ფორმულით. მაშინ:

`h(k) = ⌊10000·(123456·0.61803… mod 1⌋ = ⌊10000·(76300.0041151… mod 1⌋ = ⌊10000·0.0041151…⌋=⌊41.151…⌋=41.`

<<< ჰეშირება ღია მისამართებით (open addressing)

ამ დროს ყველა ჩანაწერი (ჩვენს შემთხვევაში სიმარტივისთვის ნატურალური რიცხვები) ინახება თავად ჰეშ-ცხრილში, რომელშიც ყოველთვის უნდა იყოს ღია, ანუ თავისუფალი უჯრები (მისამართები). ამის გამო ჰეშ-ცხრილს ეწოდება ღია მისამართებიანი ცხრილი, ხოლო ჰეშირების მეთოდს - ჰეშირება ღია მისამართებით.

ღია მისამართებიანი ჰეშ-ცხრილის ყოველი უჯრედი შეიცავს ან დინამიკური სიმრავლის ელემენტს (ჩვენს შემთხვევაში სიმარტივისთვის გასაღებს) ან NIL-ს (რაიმე საკონტროლო მნიშვნელობა, რომელიც მიგვანიშნებს რომ უჯრა ცარიელია, ჩვენს შემთხვევაში 0-ს).

ძებნის დროს, ვიწყებთ ცხრილის ერთი უჯრედიდან და, თუ იგი დაკავებულია, შემდეგ გარკვეული წესით გრძელდება ძებნის პროცესი ვიდრე არ ვიპოვით ღია მისამართს ცხრილში ან არ დავრწმუნდებით მის არარსებობაში. წესით, ღია მისამართების განსაზღვრის დროს შესანახ ელემენტთა რაოდენობა არ უნდა იყოს ცხრილის ზომაზე მეტი.

ღია მისამართების განსაზღვრის დროს გასასინჯ უჯრედთა მიმდევრობა გამოითვლება ფორმულით, გასაღების მიხედვით.

ახალი ელემენტის ჩამატებისას ჩვენ ვსინჯავთ ღია მისამართების ცხრილს, გადანომრილს 0-დან `(m-1)`-ის ჩათვლით, თავისუფალი ადგილის პოვნამდე. თუკი ყოველ ჯერზე მოგვიწევს თავიდან ბოლომდე მთელი ცხრილის გადასინჯვა, დაიხარჯება n–ის პროპორციული დრო, მაგრამ მეთოდის არსი და ღირსება ისაა, რომ ცხრილის განხილვის რიგი დამოკიდებულია გასაღებზე. კერძოდ, ძებნა შეიძლება დაიწყოს ცხრილის ნებისმიერი ადგილიდან გასაღების მნიშვნელობის მიხედვით და შეწყდეს რამდენიმე უჯრის გასინჯვის შემდეგ. სხვა სიტყვებით რომ ვთქვათ, ჰეშ-ფუნქციას ემატება მეორე არგუმენტი გასინჯული უჯრების რაოდენობა ასე რომ ჰეშ-ფუნქციას აქვს სახე:

$$H: U \times \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$$

სადაც U გასაღებთა სიმრავლეა. გამოყენებულ ადგილთა მიმდევრობას ანუ ცდათა მიმდევრობას (probe sequence) მოცემული k გასაღებისათვის აქვს სახე:

$$\langle H(k,0),\ H(k,1),\ldots,H(k,m-1)\rangle;$$

H ფუნქცია ისეთი უნდა იყოს, რომ ამ მიმდევრობაში 0-დან (m-1)- მდე ყველა რიცხვი ზუსტად ერთხელ შეგვხვდეს (ყოველი გასაღებისათვის ცხრილის ნებისმიერი პოზიცია მისაწვდომი უნდა იყოს). ქვემოთ მოყვანილია ღია მისამართების ცხრილში ელემენტის დამატების ალგორითმი. მასში იგულისხმება, რომ ჩანაწერებს გასაღების გარდა არა აქვთ დამატებითი ინფორმაცია. თუ ცხრილის უჯრედი ცარიელია, მასში ჩაწერილია 0 ან -1 (თუ ამ ადგილზე ეწერა რაღაც გასაღები, რომელიც შემდეგ რაღაც მომენტში იქნა წაშლილი).

```
int insert(int k)
{
   for(int i=0; i<m; i++)
   {
        int j = H(k,i);
        if( T[j]<=0)
        {
            T[j] = k;
            size++;
            return j;
        }
   }
   cout << "Hash Table Overflow!" << endl;
   return m;
}
```

თუ ელემენტი წარმატებით იქნა დამატებული, ალგორითმი დააბრუნებს იმ უჯრის ინდექსს, სადაც იგი ჩემატა. თუ ცხრილი გადავსებული იყო, ალგორითმი დააბრუნებს m-ს, ანუ დიაპაზონიდან გასულ ინდექსს.

ღია მისამართებიან ცხრილში k გასაღების მქონე ელემენტის ძებნისას ცხრილის უჯრედები განიხილება იმავე მიმდევრობით, როგორც იმავე გასაღების მქონე ელემენტის დამატებისას. თუკი ამ განხილვისას ჩვენ წავაწყდით უჯრედს, რომელშიც 0 წერია, შეგვიძლია დავასკვნათ,

რომ საძებნი ელემენტი ცხრილში არაა – წინააღმდეგ შემთხვევაში ის შეტანილი იქნებოდა ამ უჯრედში.

ქვემოთ მოყვანილია გაშლილი მიმართვის ცხრილში ელემენტის ძებნის პროცედურა. თუ k გასაღების მქონე ელემენტი ჩაწერილია T ცხრილის `j-` ურ პოზიციაში, პროცედურა პასუხად აბრუნებს `j-` ს, წინააღმდეგ შემთხვევაში m-ს, ანუ დიაპაზონიდან გასულ ინდექსს.

```
int search(int k)
{
    int i = 0;
    while( true)
    {
          int j = H(k,i);
          if( T[j] == k)
               return j;
          ifT[j] == 0 || i == m)
               return m;
          i++;
    }
}
```

ღია მისამართებიან ცხრილიდან ელემენტის წაშლა მარტივი საქმე არ არის. თუკი წასაშლელი ელემენტის ნაცვლად ცხრილში უბრალოდ 0 - ს ჩავწერთ, მაშინ ვეღარ ვიპოვით ელემენტებს, რომელთა ცხრილში დამატებისას ეს ადგილი შევსებული იყო. ამ სიტუაციიდან ასეთი გამოსავალია: წაშლილი ელემენტის ადგილას ჩავწეროთ არა 0, არამედ რაიმე სპეციალური მნიშვნელობა – DELETED რაც ნიშნავს "წაშლილია" (ჩვენს შემთხვევაში სიმარტივისთვის ესაა -1). ელემენტის დამატების დროს ეს უჯრა განვიხილოთ როგორც თავისუფალი, ხოლო ძებნის დროს – როგორც დაკავებული და გავაგრძელოთ ძებნა. ამ მიდგომის ნაკლი ისაა, რომ ძებნის დრო შეიძლება დიდი იყოს ნაკლებად შევსებულ ცხრილშიც კი. ამიტომ როცა ჰეშ-ცხრილიდან ჩანაწერების წაშლა ხშირად ხდება, უპირატესობას ჯაჭვებით ჰეშირებას ანიჭებენ. ალგორითმი ასეთია:

```
void erase(int k)
{
     int j =  search(k);
     if( j != m )
     {
          T[j] = -1;
          size--;
     }
}
```

ღია მისამართების ცხრილში დაკავებული ადგილების გამოთვლისათვის გამოიყენება სამი მეთოდი: წრფივი, კვადრატული და ორმაგი ჰეშირება. მათგან განვიხილოთ პირველი და მესამე.

ღია მისამართების განსაზღვრა წრფივი გადასინჯვის მეთოდით. ამ დროს,

$$H(k,i) = (h(k)+i)\%m,$$

და ღია მისამართის ძებნაში მისამართებს ვსინჯავდით მიმდევრობით, ანუ წრფივად. თავისი სიმარტივის გამო, ზოგჯერ იქმნება დაკავებლი უჯრების გრძელი სერიები, რასაც კლასტერები ეწოდება და რაც ანელებს ძებნას.

ღია მისამართების განსაზღვრა ორმაგი ჰეშირების მეთოდით. ამ მიდგომით, i-ურ ცდაზე ვცდილობთ k გასაღების ჩამატებას $\left(h_1(k)+ih_2(k)\right)\%m$ მისამართზე, სადაც $h_1(k), h_2(k)$ ორი ჰეშ-ფუნქციაა, ანუ

$$H(k,i) = \left(h_1(k)+ih_2(k)\right)\%m.$$

ამ დროს აუცილებელია, რომ მეორე ფუნქცია მხოლოდ დადებით მნიშვნელობებს იღებდეს. მართლაც, იმისათვის რომ სიმრავლეები

$$\langle H(k,0),\ H(k,1),\ldots,H(k,m-1)\rangle \quad \text{და} \quad \{0,1,\ldots,m-1\}$$

ერთმანეთს დაემთხვას, m და $h_2(k)$ უნდა იყოს ურთიერთმარტივი. მართლაც, თუ რომელიმე ორი $i_1, i_2 \in \{0,1,\ldots,m-1\}$ ცდისთვის $H(k,i_1) = H(k,i_2)$, ეს ნიშნავს რომ

$$\left(h_1(k) + i_1 h_2(k)\right)\%m = \left(h_1(k) + i_2 h_2(k)\right)\%m$$

ანუ რომელიღაც ნატურალური p რიცხვისთვის სრულდება:

$$\left(h_1(k) + i_1 h_2(k)\right) = \left(h_1(k) + i_2 h_2(k)\right) + p \cdot m,$$

ანუ

$$\left(i_1 - i_2\right) h_2(k) = p \cdot m,$$

თუ m და $h_2(k)$ ურთიერთმარტივია, მაშინ ბოლო ტოლობა ვერასოდეს შესრულდება, რადგან ამ ტოლობის მარცხენა მხარე არ იყოფა m-ზე: პირველი თანამამრავლი მასზე პატარაა, ხოლო მეორე თანამარტივია.

ზოგადად, რადგან k იცვლება, m და $h_2(k)$ რიცხვების თანამარტივობის შემოწმება საკმაოდ რთულია ყოველი k -სთვის. შესაძლოა მარტივ გამოსავალს წარმოადგენდეს m-ის მარტივ რიცხვად აღება.

შენიშვნა: ღია მისამართებით ჰეშირებისას, ზოგჯერ ჰეშ-ფუნქციას უწოდებენ უბრალოდ ჩანაწერის გასაღებს. ამ შემთხვევაშიც

$$H(k,i) = (h(k) + i)\%m,$$

$$H(k,i) = \left(h_1(k) + i h_2(k)\right)\%m.$$

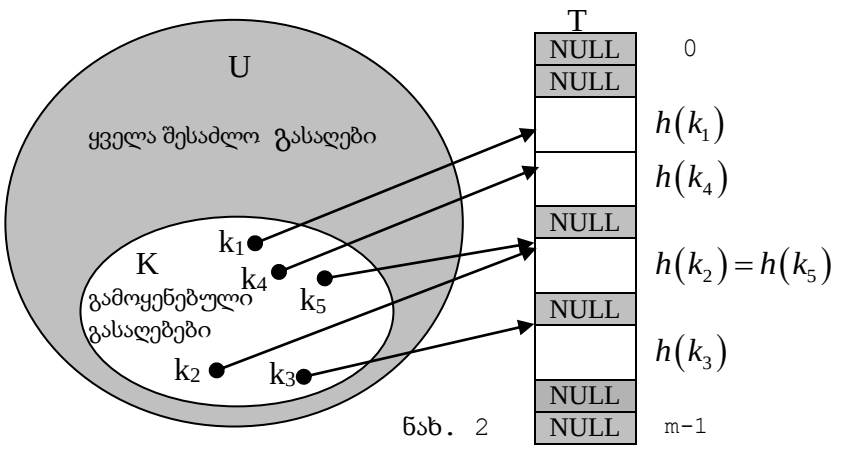
ფორმულების გამოყენება შესაძლებელია იგივე წარმატებით, მაგრამ დაპროგრამების დროს იქმნება პრობლემები გამრავლების მეთოდის გამოყენებასთან.

<<< ჰეშირება სიებით, ანუ ჯაჭვებით

k გასაღების მქონე ელემენტი ჩაიწერება h(k) ნომრის მქონე პოზიციაზე `T[m]` ჰეშ-ცხრილში (hash table), სადაც

$$h: U \to \{0,1,\ldots,m-1\}$$

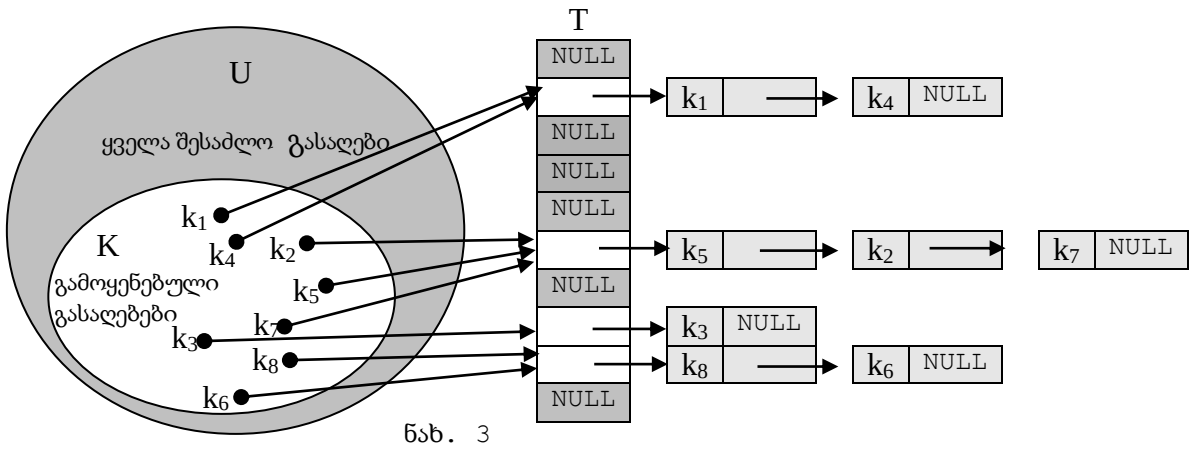
რომელიმე ჰეშ-ფუნქციაა. `h(k)` k გასაღების ჰეშ-მნიშვნელობას (hash value). ჰეშირების იდეა ნაჩვენებია ნახ. 2-ზე, სადაც ვიყენებთ არა $|U|$, არამედ m სიგრძის მასივს და ამით ვზოგავთ მეხსიერებას.



ნახ. 2

ახლა პრობლემა კვლავ კოლიზიაა,- ორი ან მეტი გასაღების ჰეშ-მნიშვნელობა შეიძლება დაემთხვეს. როდესაც $|U| > m$, იარსებებენ განსხვავებული გასაღებები, რომელთაც ერთნაირი ჰეშ-მნიშვნელობები ექნებათ. ჰეშ-ფუნქციის შერჩევისას ყოველთვის არ ვიცით, თუ რომელი გასაღებები იქნება შენახული. ჰეშ-ფუნქცია რაღაც აზრით შემთხვევითი უნდა იყოს, რათა კარგად "გააბნიოს" გასაღებები უჯრედებში. თუმცა, ასეთი შემთხვევითი ჰეშ-ფუნქციაა იმდენად მაინც უნდა იყოს დეტერმინირებული, რომ განმეორებითი გამოძახებების დროსაც ერთ და იგივე არგუმენტისათვის ერთნაირი ჰეშ-მნიშვნელობა მიიღოს.

ახლა გავარჩიოთ კოლიზიის გადაჭრა ბმული სიების გამოყენებით (ჰეშირება გადაბმით, ანუ



ნახ. 3

ჯაჭვებით: chaining). მისი არსი იმაში მდგომარეობს, ერთნაირი ჰეშ-მნიშვნელობის მქონე ელემენტებისაგან ყალიბდება ბმული სია. j -ურ პოზიციაში ინახება მიმთითებელი იმ ელემენტთა სიაზე, რომელთა გასაღებების ჰეშ-მნიშვნელობა j-ს ტოლია. თუკი ასეთი ელემენტები არ არსებობენ, j-ურ პოზიციაში იწერება NULL.

დამატების, ძებნისა და წაშლის ოპერაციები იოლად რეალიზდება:

`ChainedHashInsert(t,x)`
დავუმატოთ x ელემენტი `T[h(key[x])]` სიის დასაწყისში;

`ChainedHashSearch(t,k)`
მოვძებნოთ k გასაღების მქონე ელემენტი `T[h(k)]` სიაში;

`ChainedHashDelete(T,x)`
წავშალოთ x ელემენტი `T[h(key[x])]` სიიდან;

დამატების ოპერაცია უარეს შემთხვევაში მუშაობს O(1) დროში. ძებნის ოპერაციის მუშაობის დრო სიის სიგრძის პროპორციულია, თუმცა კარგად შერჩეული პარამეტრების შემთხვევაში მისი სისწრაფე, ისევე როგორც ელემენტის წაშლისა ორმხრივ ბმულ სიაში, შეიძლება განხორციელდეს O(1) დროში. თუკი სიები ცალმხრივადაა ბმული, მაშინ x ელემენტის წასაშლელად საჭიროა წინასწარ ვიპოვოთ მის წინ მდგომი ელემენტი. ზოგადად, საკმაოდ გავრცელებულია მიდგომა, რომ სიაში შევინახოთ არა თვითონ ობიექტები, არამედ მათი მისამართები.

<<< დაუხარისხებელი ასოცირებული კონტეინერები

დაუხარისხებელი ასოცირებული კონტეინერები იმპლემენტირებულია ჯაჭვებით ჰეშირების საფუძველზე. ასეთ კონტეინერებში, ჩასასმელი ელემენტის გასაღების სიდიდესთან ასოცირდება მისამართი, სადაც ელემეტნი განთავსდება.

C++ ენაში ამჟამად რეალიზებულია ოთხი კონტინერი, ანალოგიური დახარისხებული კონტეინერები. მათზე ჩასატარებელი მოქმედებების სიმრავლე ნაჩვენებია შემდეგ ცხრილში. ისევე როგორც ნებისმიერი სხვა კონტეინერი, ნებისმიერს აქვს მეთოდები:

`begin()`, `end()`, `size()`, `max_size()`, `empty()`, and `swap()`

	unordered_set (C++11)	unordered_map (C++11)	unordered_multiset (C++11)	unordered_multimap (C++11)	აღწერა
	(constructor)	(constructor)	(constructor)	(constructor)	აგებს კონტეინერს სხვადასხვა წყაროსგან
	(destructor)	(destructor)	(destructor)	(destructor)	შლის სიმრავლეს და მის ელემენტებს
	operator=	operator=	operator=	operator=	კონტეინერს მიანიჭებს მნიშვნელობებს
	get_allocator	get_allocator	get_allocator	get_allocator	აბრუნებს ალოკატორს, რომელიც გამოიყენება ამ კონტეინერში
წვდომა	N/A	at	N/A	N/A	წვდება მითითებულ ელემენტს საზღვრების შემოწმებით
	N/A	operator[]	N/A	N/A	წვდება მითითებულ ელემენტს საზღვრების შეუმოწმებლად
Iterators	begin	begin	begin	begin	აბრუნებს იტერატორს კონტეინერის დასაწყისზე
	end	end	end	end	აბრუნებს იტერატორს კონტეინერის დასასრულზე
Capacity	empty	empty	empty	empty	ამოწმება, ცარიელია კონტეინერი?
	size	size	size	size	ელემენტების რაოდენობა
	max_size	max_size	max_size	max_size	ელემენტების მაქსიმალური შესაძლო რაოდენობა
Modifiers	clear	clear	clear	clear	ასუფთავებს კონტეინერს
	insert	insert	insert	insert	ჩაამატებს ელემენტს
	emplace	emplace	emplace	emplace	ადგილზე აგებს ელემენტებს (C++11)
	emplace_hint	emplace_hint	emplace_hint	emplace_hint	ადგილზე აგებს ელემენტებს კარნახის დახმარებით (C++11)
	erase	erase	erase	erase	წაშლის ელემენტს
	swap	swap	swap	swap	სხვა კონტეინერს გაუცვლის შიგთავსს
Lookup	count	count	count	count	ითვლის მითითებული გასაღების მქონე ელემენტებს
	find	find	find	find	ეძებს ელემენტს გასაღებით
	equal_range	equal_range	equal_range	equal_range	მითითებული გასაღების შესაბამის დიაპაზონს აბრუნებს
Observers	hash_function	hash_function	hash_function	hash_function	აბრუნებს ჰეშ-ფუნქციას
	key_eq	key_eq	key_eq	key_eq	აბრუნებს ტოლობის ფუნქციას

<<< ლიტერატურა

1. By William H. Ford and William R. Topp. Data Structures with C++ using STL. Prentice Hall, 2001.
2. D. Musser, G. Derge, A. Saini. STL Tutorial and Reference Guide, Second Edition, Addison-Wesley, 2006.
3. T.Cormen, C. Leiserson, R. Rivest, C. Stein. Introduction to Algorithms, Third Edition. The MIT Press, 2009.