ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი

თეიმურაზ თუთბერიძე

ფიბონაჩის გროვის იმპლემენტაცია

ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერება

ნაშრომი შესრულებულია კომპიუტერული მეცნიერების მაგისტრის აკადემიური ხარისხის მოსაპოვებლად

ხელმძღვანელი: პროფესორი კობა გელაშვილი

თბილისი, 2018

ანოტაცია

ფიბონაჩის გროვა ცნობილი მონაცემთა სტრუქტურაა, რომელიც ფართოდ გამოიყენება პრიორიტეტების რიგებსა და გრაფის ალგორითმებში, სადაც ოპერაციები სრულდება დიდი მოცულობის მონაცემებზე. მისი თეორიული მახასიათებლების გათვალისწინებით, ამ ამოცანებში მიზანშეწონილია ფიბონაჩის გროვების გამოყენება. შესაბამისად, მნიშვნელოვან ამოცანას წარმოადგენს ფიბონაჩის გროვების ეფექტური იმპლემენტაცია და მისი გამოყენების ჩვევების შეძენა.

აქედან გამომდინარე, წარმოდგენილ ნაშრომში განხილულია ფიბონაჩის გროვის ალგორითმები თეორიული და პრაქტიკული იმპლემენტაციის თვალსაზრისით. განხილულია იმპლემენტაციის გამოყენება დეიქსტრას ალგორითმში. წარმადობის ცვლილება შესწავლილია ექსპერიმენტულად შესაბამისი ტესტების საფუძველზე.

Abstract

Fibonacci heap a well-known data structure, which is widely used in priority queues and graph algorithms, where operations are executed on big blocks of data. Taking into account its theoretical characteristics, it is desirable to use Fibonacci heaps in this problems. Hence, effective implementation of Fibonacci heap and obtaining using habits is very important.

Hence, in this work the theoretical and practical implementations of Fibonacci heap algorithms are discussed. We've examined implementation on Dijkstra algorithm. Efficiency difference is analyzed experimentally by relying on the corresponding tests.

სარჩევი

ფიბონაჩის გროვა 6
განმარტებები 6
მინიმალური გასაღების მქონე კვანძის ამოჭრა 8
ფიბონაჩის გროვის გათანაბრება 8
მინიმუმის ამოღების ამორტიზებული ღირებულება 14
ნებისმიერი კვანძის ამოჭრა 19
კვანძის გასაღების შემცირება 19
კვანძის წაშლის ამორტიზებული ღირებულება 21
მაქიმალური ხარისხის განსაზღვრა 23
ფიბონაჩის გროვა დალაგების მიმართებით (კომპარატორით) 26
პრიორიტეტების რიგი 27
დეიქსტრას ალგორითმი 30
შეფასება 35
დასკვნა 36
ლიტერატურა 37

შესავალი

კომპიუტერულ მეცნიერებაში უმნიშვნელოვანესი როლი უჭირავს პრიორიტეტების რიგს. მათი მთავარი ხიბლი ისაა, რომ ყოველ ელემენტს გააჩნია საკუთარი პრიორიტეტი (ნომერი) და არ აქვს მნიშვნელობა ელემენტების რიგში მოხვედრის თანმიმდევრობას. მასში უმაღლესი პრიორიტეტის მქონე ელემენტი მუშავდება უპირველესად.

პრიორიტეტების რიგის კონტეინერად შეიძლება გამოვიყენოთ უამრავი მონაცემთა სტრუქტურა, თუმცა საუკეთესო შედეგებს გვაძლევენ გროვები. გროვები ასპარეზზე გამოჩნდა 1964 წლიდან, მაგრამ ფიბონაჩის გროვები, რომლების განხილვასაც მიეძღვნება ეს ნაშრომი, შეიქმნა 1984 წელს. მისი სახელი გამომდინარეობს იქედან, რომ ფიბონაჩის რიცხვები გამოიყენება გროვის მუშაობის დროის შესაფასებლად.

გროვები აქტიურად გამოიყენება გრაფებზე ალგორითმებში. გამონაკლისი არც დეიქსტრას ალგორითმია, რომლის მიზანია გრაფის საწყისი წვეროდან ყველა დანარჩენ წვერომდე უმოკლესი მანძილების პოვნა, რაზეც ქვემოთ ვრცლად ვისაუბრებთ.

ფიბონაჩის გროვის უპირატესობა სხვა გროვებთან ისაა, რომ მასში ელემენტის ჩასმის, მინიმალური ელემენტის ამოკითხვის, გასაღების შემცირების და გროვების გაერთიანების ოპერაციები სრულდება $O(1)$ დროში. ამიტომ, ფიბონაჩის გროვები ოპტიმალური მონაცემთა სტრუქტურაა პრიორიტეტების რიგებისთვის. შესაბამისად, მისი შესწავლა, გამოყენება და განვითარება ძალზედ აქტუალურია დიდ მონაცემებზე მუშაობისათვის.

წარმოდგენილი ნაშრომი ეძღვნება ამ საკითხის შესწავლას. იგი შედგება შესავალისა და ოთხი პარაგრაფისგან.

პირველ თავში განხილულია ფიბონაჩის გროვის სტრუქტურა და მასზე ოპერაციები. კერძოდ, მინიმალური გასაღების ამოჭრის, გასაღების შემცირების და ნებისმიერი კვანძის ამოჭრის ალგორითმები. ასევე, მოყვანილია თითოეული ალგორითმის ამორტიზებული ღირებულებები. ამავე თავში განხილულია ფიბონაჩის გროვა კომპარატორით.

მეორე თავში მოყვანილია პრიორიტეტების რიგის განსაზღვრება და მისი იმპლემენტაციის მაგალითი ფიბონაჩის გროვის გამოყენებით.

მესამე თავში მოყვანილია დეიქსტრას ალგორითმის როგორც ფსევდო, ასევე C++-ის კოდი პრიორიტეტების რიგის გამოყენებით.

მეოთხე თავში შედარებულია ჩვენს მიერ შემუშავებული პრიორიტეტების რიგის გამოყენებით იმპლემენტირებული დეიქსტრას ალგორითმისა და მისი სწორხაზოვანი ალგორითმის მუშაობის დროები.

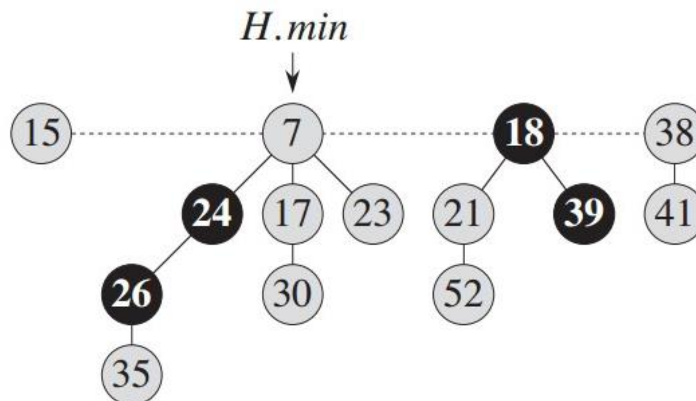
ბოლოს კი მოყვანილია მოკლე დასკვნა და ლიტერატურის ნუსხა.

ფიბონაჩის გროვა

ფიბონაჩის გროვა [1] არის მონაცემთა სტრუქტურა, რომელიც გამოიყენება პრიორიტეტების რიგის მონაცემთა სტრუქტურის აგებისთვის. მას აქვს ზევრად უკეთესი ამორტიზებული მუშაობის დრო სხვა მონაცემთა სტრუქტურებთან შედარებით, რომლებიც გამოიყენება ამ მიზნით. კერძოდ, უკეთესია ვიდრე ორობითი და ბინომიალური გროვები. იგი მინიმალური ელემენტის ძებნის, ჩასმის და გასაღების შემცირების ოპერაციებს ასრულებს $O(1)$ ამორტიზებულ დროში, ხოლო მინიმალური ელემენტის წაშლის ოპერაციას ასრულებს $O(\log n)$ ამორტიზებულ დროში, სადაც n არის გროვის ელემენტების რაოდენობა. იგი ასევე აერთიანებს ორ გროვას მუდმივ ამორტიზებულ დროში, რაც უფრო ეფექტურია, ვიდრე ბინომიალური და ორობითი ხეების გაერთიანების დრო. სწორედ ამ შეფასებების გამო გამოიყენება ფიბონაჩის გროვები პრიორიტეტების რიგის კონტეინერად. პრიორიტეტების რიგში ელემენტის ჩასმა, მინიმუმის ძებნა და გასაღების შემცირება სრულდება პოლინომიალურ ამორტიზებულ დროში.

განმარტებები

განვიხილოთ ფიბონაჩის გროვის ერთ-ერთი მაგალითი.



სურ1.

სურათ 1-ზე ნაჩვენებია ფიბონაჩის ხე, რომლის ზომა არის 14 და რომელსაც ფესვების სიაში აქვს ოთხი ელემენტი. ფესვების სია წარმოადგენს ორმხრივად ბმულ სიას, რომლის

საწყისი ელემენტი ყოველთვის არის მინიმალური. ფიბონაჩის გროვის მთავარი წესის თანახმად ფესვების სიაში მყოფი თითოეული ელემენტის გასაღები უნდა იყოს ნაკლები ან ტოლი მისი შვილების ყოველ გასაღებზე, რაც უზრუნველყოფს რომ მინიმალური ელემენტი ყოველთვის არის ფესვების სიაში. ფიბონაჩის გროვა ძალიან მოქნილია, რადგან მას არ აქვს წინასწარ განსაზღვრული ფორმა. ეს გვაძლევს საშუალებას, რომ ზოგიერთი ოპერაცია შესრულდეს ძალიან სწრაფად (მაგალითად ჩასმა, რომელიც გულისხმობს ფესვების სიაში ელემენტის ჩამატებას; გაერთიანება, რომელიც გულისხმობს ორი გროვის ფესვების სიის გადაბმას და ა.შ.). თუმცა გროვის დაბალანსება მაინც საჭიროა სასურველი შედეგის მისაღწევად, რაც წარმოადგენს ყველაზე დიდ პრობლემას ფიბონაჩის გროვებში.

ფიბონაჩის გროვისა და კვანძის კლასები:

```
class FibHeap {
    int n;  //გროვის ელემენტების რაოდენობა
    FibHeapNode *min; //მინიმალურ ელემენტზე მიმთითებელი
}
class FibHeapNode {
     int *key; //კვანძის გასაღები
     int degree; //კვანძის ხარისხი (შვილების რაოდენობა)
     FibHeapNode *p; //მიმთითებელი მშობელ კვანძზე
     FibHeapNode **child; //შვილების მასივი
     bool mark; //დაჭდევებულია თუ არა კვანძი (გამოიყენება ანალიზისთვის)
     FibHeapNode *left; //მარცხენა მეზობელზე მიმთითებელი
     FibHeapNode *right; //მარჯვენა მეზობელზე მიმთითებელი
}
```

როგორც უკვე აღვნიშნეთ, ელემენტის ჩასმა და მინიმალურის ძებნა ხდება მუდმივ ამორტიზებულ დროში. მთავარ სირთულეს წარმოადგენს მინიმალური ელემენტის ამოშლა (extract min) და ნებისმიერი ელემენტის გროვიდან ამოშლა, რომლებიც სრულდება ლოგარითმულ დროში.

მინიმალური გასაღების მქონე კვანძის ამოჭრა

განვიხილოთ მინიმალური ელემენტის ამოჭრის ფსევდოკოდი.

1. მინიმალური ელემენტის მისამართი შევინახოთ რაიმე ცვლადში, მაგალითად Z-ში;
2. Z-ის ყოველი შვილი ავიტანოთ ფესვების სიაში;
3. Z ამოვშალოთ ფესვების სიიდან;
4. თუ Z იყო ერთადერთი ელემენტი ამ გროვაში, მინიმალური ელემენტი გავხადოთ NULL;
5. თუ Z არ იყო ერთადერთი ელემენტი გროვაში, მინიმალური გავხადოთ Z-ს მარჯვენა მეზობელი და გამოვიძახოთ ფუნქცია CONSOLIDATE;
6. შევამციროთ გროვის ელემენტების რაოდენობა და დავაბრუნოთ ამოშლილი ელემენტი.

ამ ალგორითმში ყველაზე ფაქიზი საკითხი არის CONSOLIDATE ფუნქციის იმპლემენტაცია, რომელმაც უნდა მოაწესრიგოს გროვა ამოშლის შემდეგ. მოწესრიგება გულისხმობს, რომ ფესვების სიაში მყოფ ყოველ ორ განსხვავებულ ელემენტს უნდა ჰყავდეს შვილების განსხვავებული რაოდენობები.

ფიბონაჩის გროვის გათანაბრება

განვიხილოთ CONSOLIDATE ფუნქციის ფსევდო კოდი.

პირველ რიგში დაგვჭირდება კვანძების დამხმარე მასივი, რომელშიც კვანძის ხარისხის შესაბამის ინდექსზე დავიმახსოვრებთ ამ კვანძის მისამართს, რათა შემდეგში ტოლი

ხარისხის მქონე კვანძები ავკინძოთ. ასევე აქ გამოვიყენებთ $D(n)$ ელემენტიან მასივს, სადაც $D(n)$ არის ფიბონაჩის გროვის მაქსიმალური ხარისხი (შვილების მაქსიმალური რაოდენობა, რაც შეიძლება ჰყავდეს ფესვების სიაში მყოფ კვანძს). ამ სიდიდეს ჩვენ ბოლოს შევაფასებთ.

1. შევქმნათ $D(n)$ ელემენტიანი A მასივი და შევავსოთ ის $NULL$-ებით;
2. ყოველი ელემენტისთვის ფესვების სიიდან გავაკეთოთ შემდეგი:
    1. x-ში დავიმახსოვროთ მიმდინარე კვანძის მისამართი, ხოლო d-ში მიმდინარე ელემენტის ხარისხი;
    2. გადავუაროთ A მასივს d ინდექსიდან მანამ, სანამ არ მივაღწევთ ცარიელ ელემენტს A მასივში და გავაკეთოთ შემდეგი:
        1. შევადაროთ მიმდინარე კვანძის გასაღები მისი ტოლი ხარისხის მქონე კვანძის გასაღებს A მასივიდან.
        2. მეტი გასაღების მქონე კვანძი ავკინძოთ ნაკლები გასაღების მქონე კვანძთან (გავხადოთ მისი შვილი).
        3. A მასივის d ინდექსიდან ამოვშალოთ აკინძული კვანძი და გავაგრძელოთ ციკლის შესრულება
    3. როდესაც 2.2 ციკლი შეწყვეტს მუშაობას, ეს ნიშნავს რომ ფესვების სიიდან მიმდინარე კვანძის ტოლი ხარისხის კვანძი არ გვაქვს A მასივში. შესაბამისად, ამ კვანძის მისამართი ჩავწეროთ A მასივში მისი ხარისხის შესაბამის ინდექსზე და გადავიდეთ შემდეგ კვანძზე ფესვების სიიდან.
3. მე-2 ციკლის შესრულების შემდეგ A მასივში გვექნება ყველა უნიკალური ხარისხის მქონე კვანძები. დავამატოთ ისინი ფესვების სიაში და მათ შორის მინიმალური გასაღების მქონე კვანძი გავხადოთ მინიმალურად.

2.2 ციკლი გვჭირდება იმიტომ, რომ პირველადი აკინძვის შედეგად ფესვებში დარჩენილი კვანძის ხარისხი იზრდება ერთით, შესაბამისად გაზრდილი ხარისხის ტოლი ხარისხის მქონე კვანძი შეიძლება კიდევ არსებობდეს ჩვენს გროვაში და ისიც უნდა აიკინძოს შესაბამის კვანძთან.

კოდში მინიმალური ელემენტის წაშლისას ჩვენ ვიყენებთ შემდეგ ფუნქციებს:

```
Void AddToRootList ( FibHeapNode *x );

Void RemoveFromRootList (  );

Void Consolidate ( );

Void swap ( FibHeapNode **x, FibHeapNode **y );

Void FibHeapLink ( FibHeapNode *y, FibHeapNode *x );
```

AddToRootList და RemoveFromRootList არიან დამხმარე მეთოდები, რომლებიც შესაბამისად ამატებს ელემენტს ფესვები სიაში და შლის ელემენტს ფესვების სიიდან.

AddToRootList მეთოდი ამატებს x კვანძს ფესვების სიაში და სვამს მას მინიმალური ელემენტის მარცხენა მეზობლად.

RemoveFromRootList არის კვანძის მეთოდი. ის არ აკეთებს ცვლილებებს ამოშლილ კვანძზე. იგი უბრალოდ გადააბამს ერთმანეთს ამოსაშლელი კვანძის მარცხენა და მარჯვენა მეზობლებს.

```
void RemoveFromRootList() {

        this->GetLeft()->SetRight(this->GetRight());

        this->GetRight()->SetLeft(this->GetLeft());

  }
```

Swap მეთოდი ადგილებს უცვლის გადაცემულ კვანძებს და გამოიყენება Consolidate ფუნქციაში იმ შემთხვევაში, თუ *A* მასივში მოთავსებული კვანძის გასაღები ნაკლებია იგივე ხარისხის მქონე კვანძის გასაღებზე ფესვების სიიდან. იგი არგუმენტებად იღებს კვანძების მიმთითებლების მისამართებს. მაგალითად, თუ გვინდა გავუცვალოთ ადგილი x და y კვანძს, მას უნდა გადავცეთ x და y კვანძების მიმთითებლების მისამართები. ფუნქცია

გაუცვლის ადგილებს მიმთითებლებს და x-ს გახდის y კვანძის მიმთითებლად, ხოლო y – x კვანძის მიმთითებლად.

FibHeapLink მეთოდიც გამოიყენება Consolidate მეთოდში ერთნაირი ხარისხის მქონე კვანძების ასაკინძად. მას გადაეცემა y და x კვანძების მისამართები. მეთოდი y კვანძს ხდის x კვანძის შვილად.

```
void FibHeapLink(FibHeapNode *y, FibHeapNode *x) {
        y->GetRight()->SetLeft(y->GetLeft());
        y->GetLeft()->SetRight(y->GetRight());
        x->SetChild(y);
        y->SetParent(x);
        y->SetLeft(NULL);
        y->SetRight(NULL);
        x->SetDegree(x->GetDegree() + 1);
        y->SetMark(false);
}
```

მინიმალური ელემენტის ამოჭრის ალგორითმში საკვანძო საკითხს წარმოადგენს CONSOLIDATE() მეთოდი. მისი ძირითადი ამოცანაა გაათანაბროს გროვა ისე, რომ ყოველ კვანძს ფესვების სიიდან ჰყავდეს სხვადასხვა რაოდენობის შვილები. ამისათვის შემოგვაქვს დამხმარე *A* მასივი, სადაც დროებით ვიმახსოვრებთ ერთნაირი ხარისხის მქონე კვანძების მისამართებს. ყოველ ელემენტს ფესვების სიიდან ვადარებთ *A* მასივში - თუ გვაქვს დამახსოვრებული იგივე ხარისხის კვანძი. თუ არ გვაქვს, მაშინ ვიმახსოვრებთ მიმდინარე კვანძს ფესვების სიიდან *A* მასივში. წინააღმდეგ შემთხვევაში, ვკინძავთ მათ, *A* მასივიდან ვშლით დამახსოვრებულ კვანძს და ვეძებთ იგივე ხარისხის კვანძს *A* მასივში, რაც აქვს

აკინძული კვანძის მშობელს. თუ აღმოვაჩინეთ ასეთი ელემენტი *A* მასივში, მას კიდევ ერთხელ ავკინძავთ. წინააღმდეგ შემთხვევაში გადავდივართ შემდეგ კვანძზე ფესვების სიიდან. ამ ალგორითმის შესრულების შემდეგ *A* მასივში გვექნება ყველა კვანძი ფესვების სიიდან და ამასთანავე, მათი ხარისხები იქნება განსხვავებული. ამის შემდეგ, ვადგენთ ახალ ფესვების სიას *A* მასივის ელემენტებისგან და მათ შორის მინიმალური გასაღების მქონე კვანძს ვუწოდებთ გროვის მინიმალურ ელემენტს.

```
void Consolidate() {
	……..
	FibHeapNode **A;
	A = new FibHeapNode *[maxDegree];
	FibHeapNode *z = this->min;
	FibHeapNode *start = this->min->GetLeft();
	FibHeapNode *next;
	//do for all nodes in root list
	while (true) {
		FibHeapNode *x = z;
		//Get degree of current node
		int d = x->GetDegree();
		//Remember right neighbour
		next = z->GetRight();
		while (A[d] != NULL) {
			FibHeapNode *y = A[d];
```

```
				//If current node is greater than node in array with the same degree
				if (y->GetKey() < x->GetKey()) {
					//swap names
					this->swap(&x, &y);
				}
				//Make greater node child of smaller
				this->FibHeapLink(y, x);
				A[d] = NULL;
				d++;
			}
			//Add node with known unique degree to A
			A[d] = x;
			//If we looped through all root nodes, stop the loop
			if (z == start) {
				break;
			}
			//Go to next node in root list
			z = next;
		}
		……..
	}
```

მინიმუმის ამოღების ამორტიზებული ღირებულება

იმისათვის, რომ გავიგოთ მინიმუმის ამოღების ამორტიზებული ღირებულება, ჯერ დავთვალოთ რეალური ღირებულება.

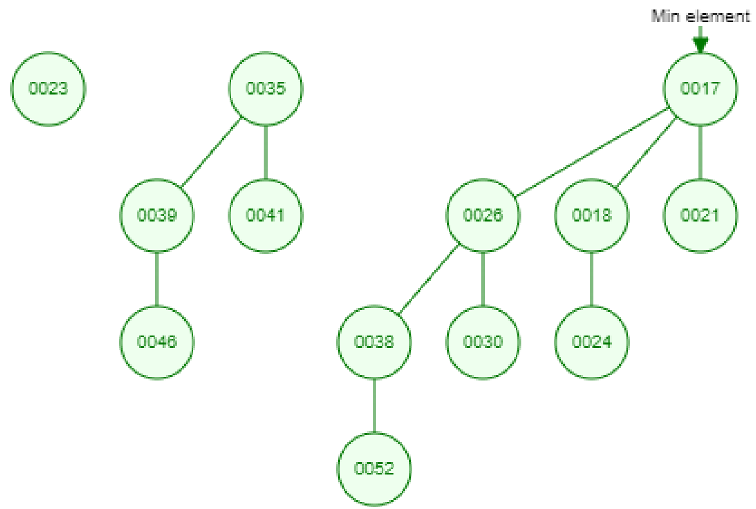
CONSOLIDATE პროცედურის გამოძახების წინ ფესვების სიის სიგრძე მაქსიმუმ შეიძლებოდა ყოფილიყო $D(n) + t(H) - 1$, რადგან $t(H)$ არის თავდაპირველი ფესვების რაოდენობას გამოკლებული ამოჭრილი მინიმალური ელემენტი და დამატებული მინიმალური ელემენტის მაქსიმალური ხარისხი, რომელიც არის $D(n)$. აქვე გვხვდება ორი ჩადგმული ციკლი. ერთი გადის ფესვების სიას, ხოლო მეორე - დროებით მასივს. აღსანიშნავია, რომ ჩადგმული ციკლის ერთი იტერაციისას მცირდება ფესვების სიაში ხეების რაოდენობა. შესაბამისად, გარე ციკლის ყოველი იტერაციისას შიდა ციკლის იტერაციების რაოდენობა არ აღემატება ფესვების სიაში კვანძების რაოდენობას. ე. ი. შესრულებული სამუშაოს რეალური ღირებულება არის $(D(n) + t(H))$-ის პროპორციული.

პოტენციალი მინიმუმის ამოღებამდე იყო $t(H) + 2 * m(H)$, რადგან ფესვების სიაში რჩება მაქსიმუმ $D(n) + 1$ კვანძი, ხოლო დაჭდევებული კვანძების რაოდენობა არ იცვლება. ე. ი. პროცედურის მუშაობის ამორტიზებული ღირებულება იქნება:

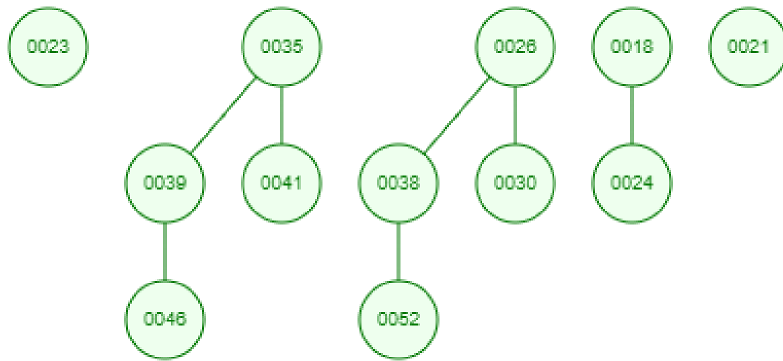
$$O\big(D(n) + t(H)\big) + (D(n) + 1) + 2m(H)) - (t(H) + 2m(H))$$

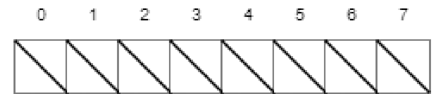
$$= O(D(n)) + O(t(H)) - t(H) = O(D(n))$$

მინიმალური ელემენტის ამოშლის მაგალითი:



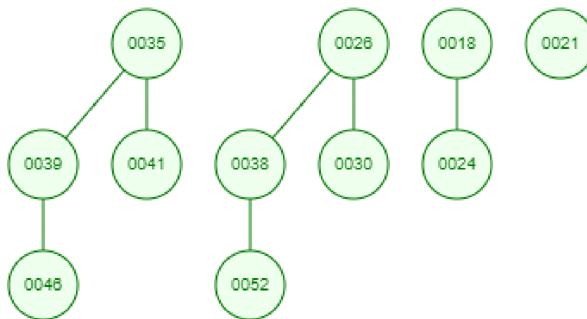
0017

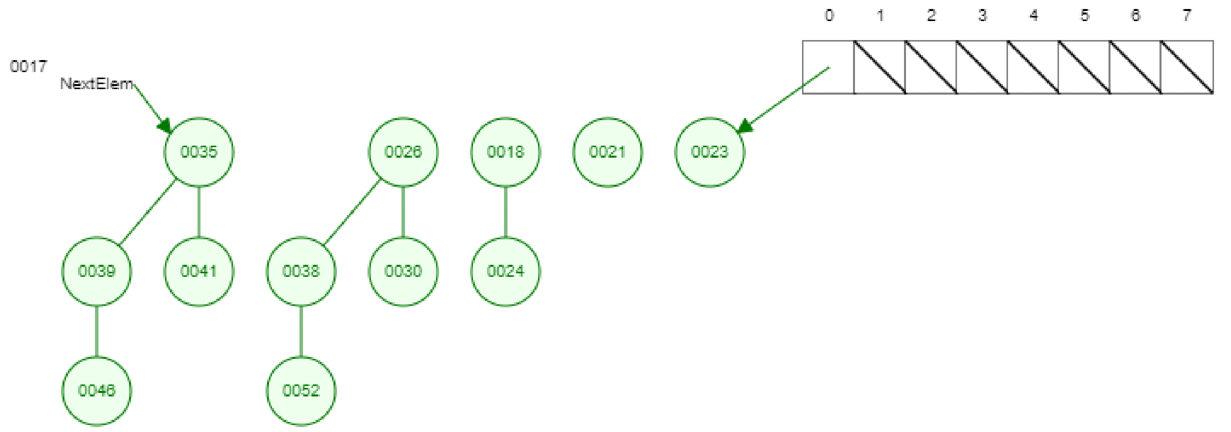


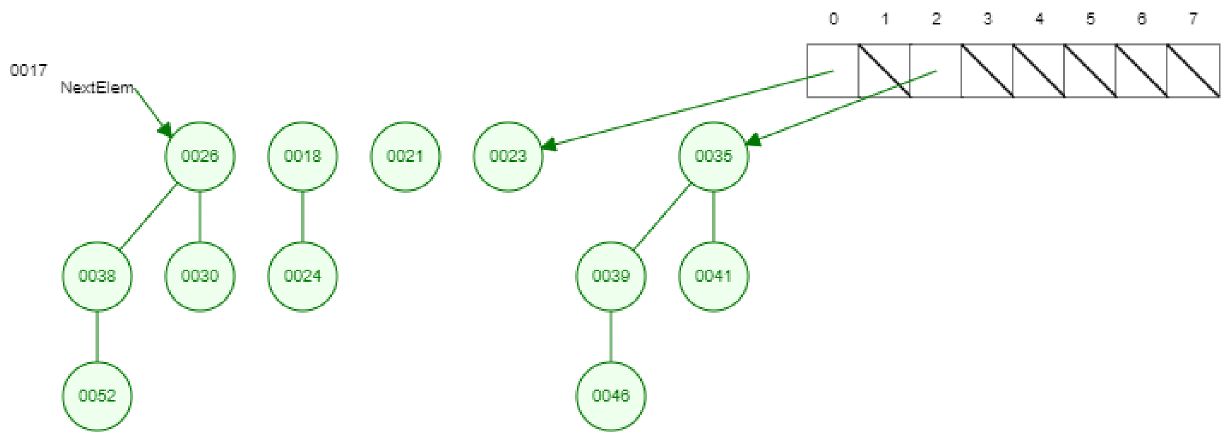


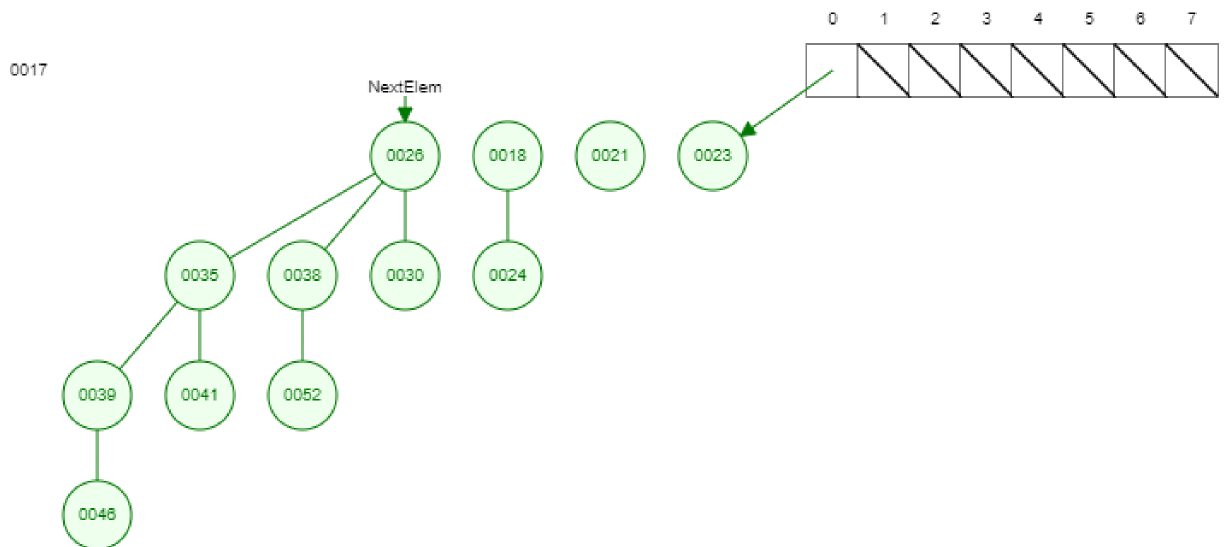
0017

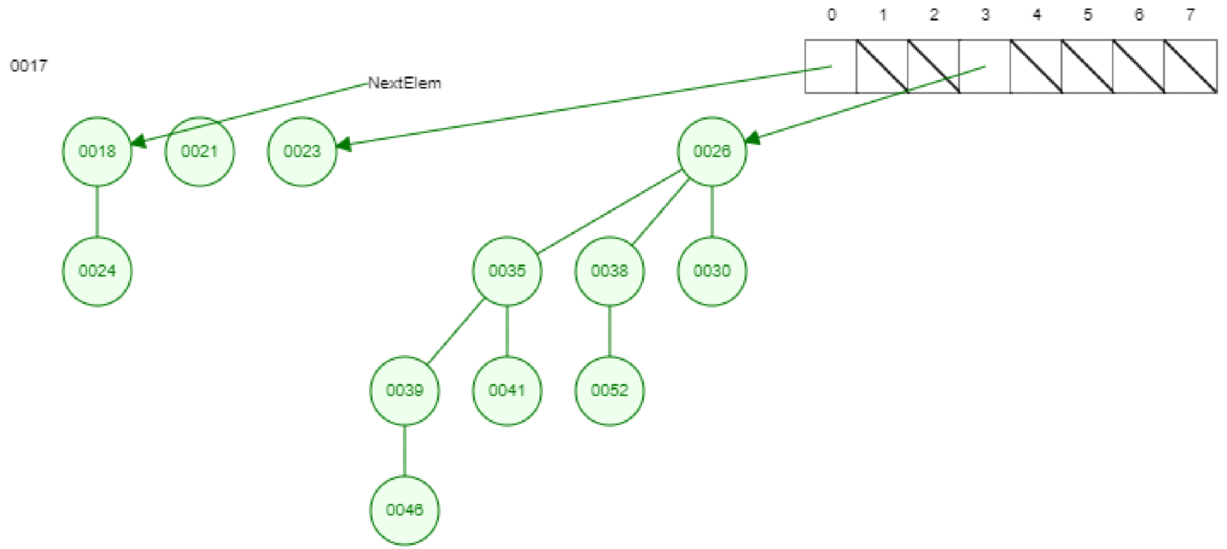
NextElem

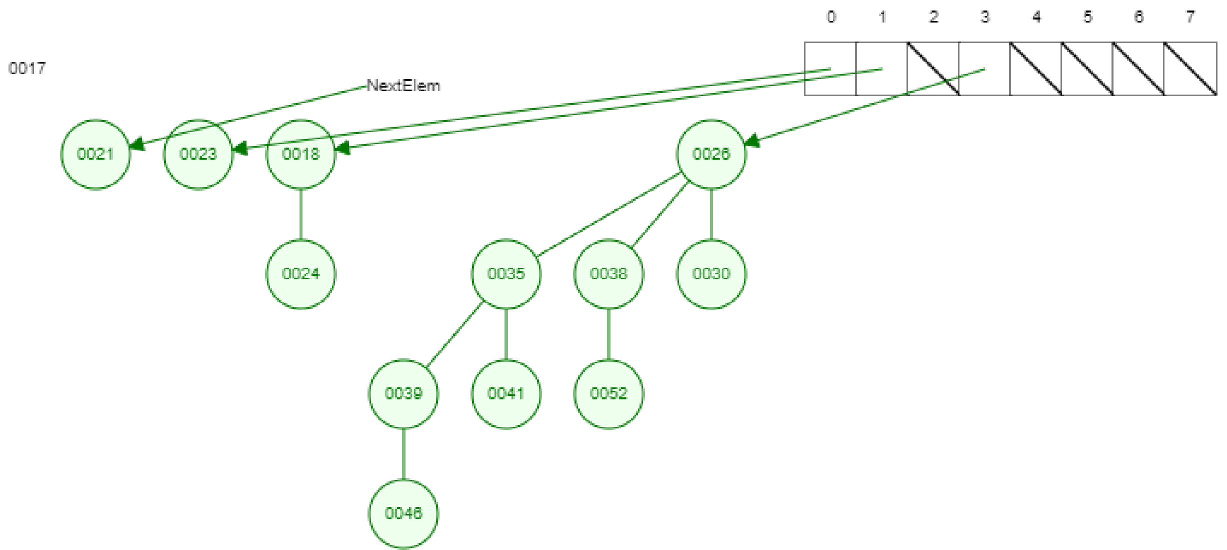


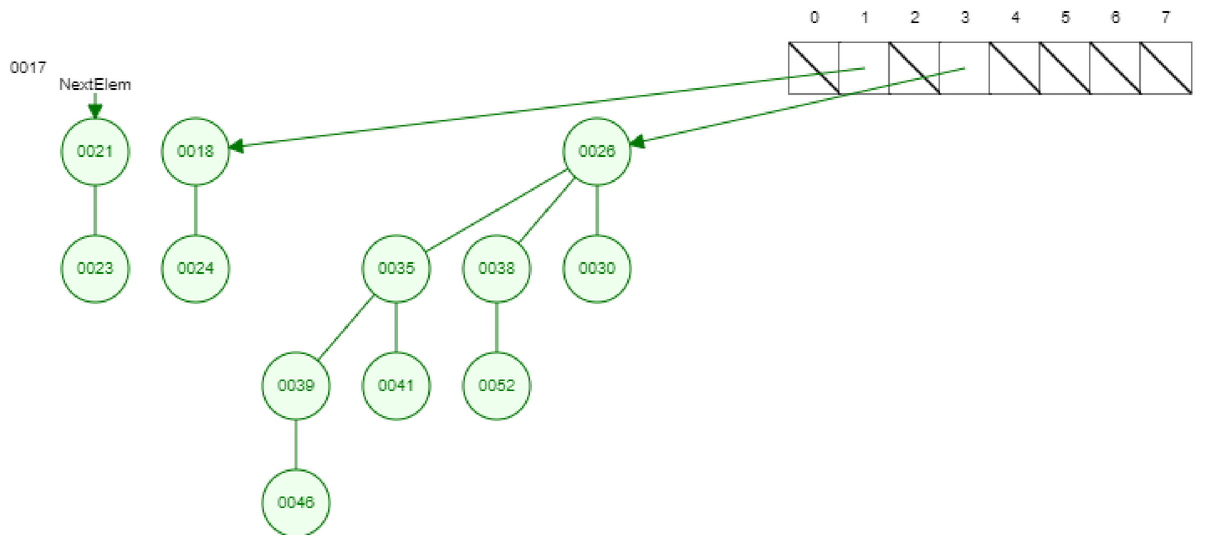

0 1 2 3 4 5 6 7
0017
NextElem
0035
0026
0018
0021
0023
0039
0041
0038
0030
0024
0046
0052

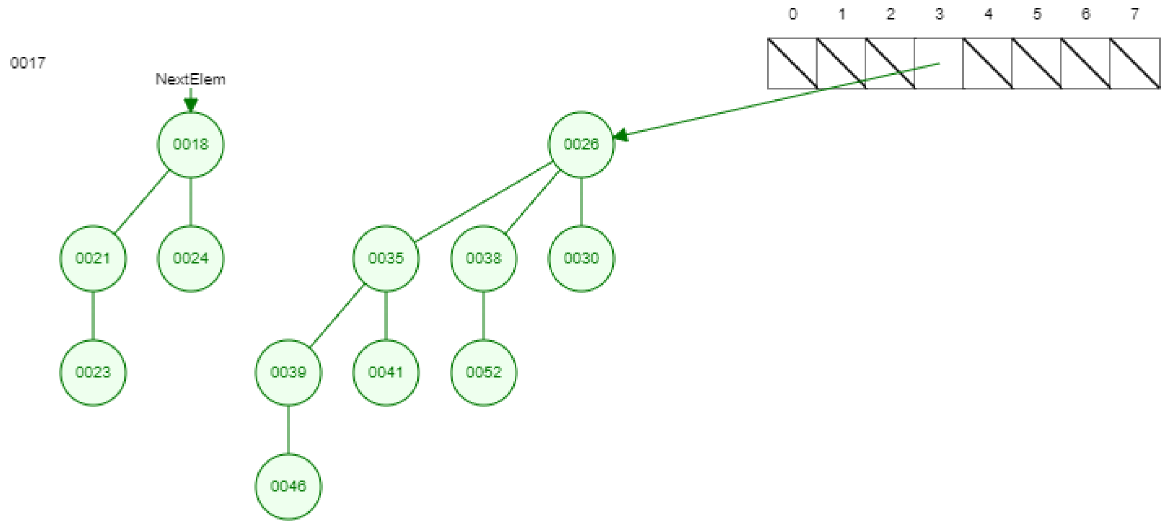

0 1 2 3 4 5 6 7
0017
NextElem
0026
0018
0021
0023
0035
0038
0030
0024
0039
0041
0052
0046

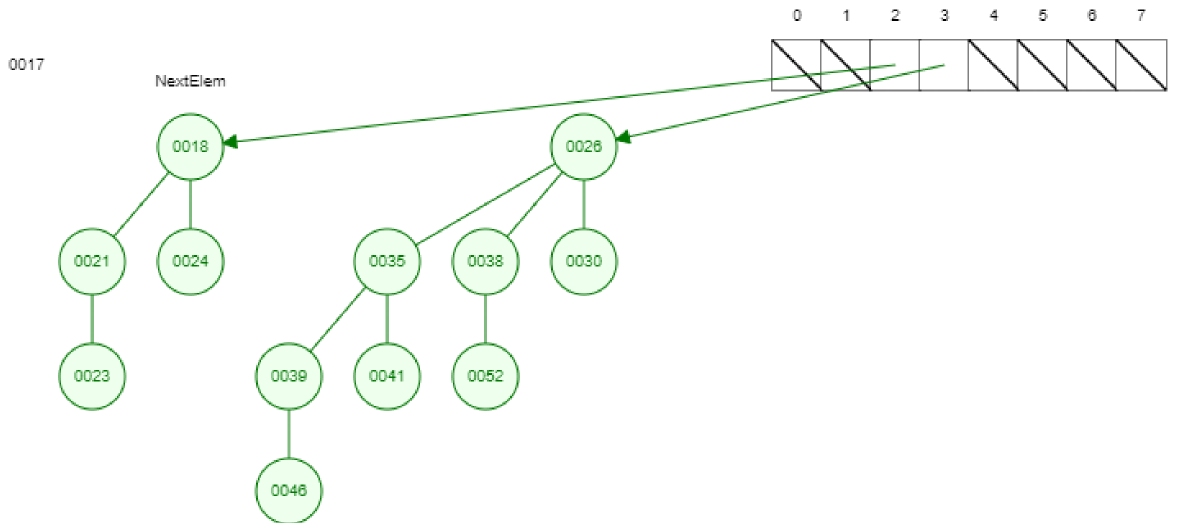

0 1 2 3 4 5 6 7
0017
NextElem
0026
0018
0021
0023
0035
0038
0030
0024
0039
0041
0052
0046

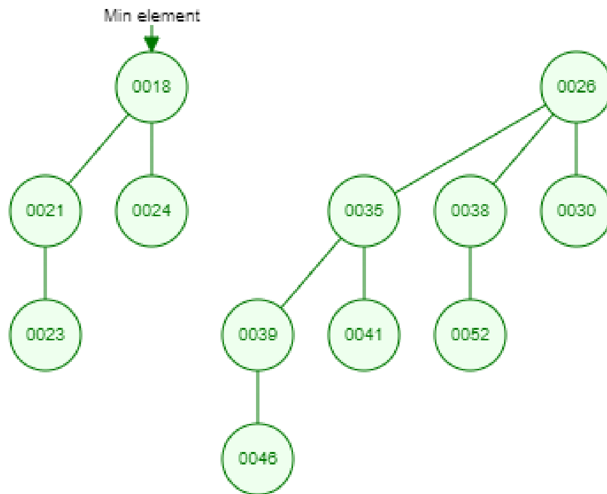

0
1
2
3
4
5
6
7
0017
NextElem
0018
0021
0023
0026
0024
0035
0038
0030
0039
0041
0052
0046


0
1
2
3
4
5
6
7
0017
NextElem
0021
0023
0018
0026
0024
0035
0038
0030
0039
0041
0052
0046


0
1
2
3
4
5
6
7
0017
NextElem
0021
0018
0026
0023
0024
0035
0038
0030
0039
0041
0052
0046


0
1
2
3
4
5
6
7
0017
NextElem
0018
0026
0021
0024
0035
0038
0030
0023
0039
0041
0052
0046


0
1
2
3
4
5
6
7
0017
NextElem
0018
0026
0021
0024
0035
0038
0030
0023
0039
0041
0052
0046


Min element
0018
0026
0021
0024
0035
0038
0030
0023
0039
0041
0052
0046

ნებისმიერი კვანძის ამოჭრა

მეორე ალგორითმი, რომელიც მუშაობს ლოგარითმულ დროში, არის კვანძის ამოჭრა (FIB-HEAP-DELETE), რომელიც იყენებს ორ მეთოდს:

1. კვანძის გასაღების შემცირებას (Decrease Key);
2. მინიმალური გასაღების მქონე კვანძის ამოჭრას (Extract Min).

ჩვენ უკვე განვიხილეთ მინიმალური კვანძის ამოჭრის ალგორითმი. ახლა განვიხილოთ კვანძის გასაღების შემცირების ალგორითმი.

კვანძის გასაღების შემცირება

განვიხილოთ x კვანძის გასაღების k მნიშვნელობით შეცვლის ალგორითმის ფსევდო-კოდი:

1. თუ k მეტია x კვანძის გასაღებზე ალგორითმი წყვეტს მუშაობას, რადგან ჩვენ გვინდა მხოლოდ გასაღების შემცირება და არა გაზრდა.
2. თუ k ნაკლებია ან ტოლი x კვანძის გასაღებზე, x-ის გასაღების მნიშვნელობა გავხადოთ k.
3. თუ x არ არის ფესვების სიაში და მისი გასაღები ნაკლებია მშობლის გასაღებზე, მოვიქცეთ შემდეგნაირად:
    1. ამოვჭრათ x კვანძი მისი მშობლის შვილების სიიდან და გადავიტანოთ ის ფესვების სიაში.
    2. თუ x-ის მშობელი არ არის ფესვების სიაში და მას x კვანძის ფესვებში გადატანამდე უკვე დაკარგული ჰყავდა სხვა შვილიც, მაშინ x-ის მშობელიც ამოიჭრება მისი მშობლის შვილების სიიდან, გადავა ფესვების სიაში და შესრულდება კვლავ 3.2 პუნქტი. ეს პროცედურა გაგრძელდება მანამ, სანამ მიმდინარე კვანძის მშობელი არ იქნება ფესვების სიაში ან მიმდინარე კვანძის მშობელს არ ყავდა დაკარგული სხვა შვილი მიმდინარეს გარდა.
4. თუ x-ის გასაღები ნაკლებია მინიმალური კვანძის გასაღებზე, x კვანძი გავხადოთ მინიმალურ ელემენტად.

კოდში კვანძის ამოჭრის ალგორითმში გამოყენებულია შემდეგი ორი მეთოდი:

- **void** Cut (**FibHeapNode** *x);
- **void** CascadingCut (**FibHeapNode** *y);

CascadingCut აღნიშნავს, რომ მიმდინარე კვანძის მშობელს დაკარგული აქვს ერთი შვილი მისი ამოჭრის შემდეგ, ან თუ მიმდინარე კვანძი მეორეა, რომელიც დაკარგა მისმა მშობელმა, მშობელსაც გადაიყვანს ფესვების სიაში და ა. შ. მანამ, სანამ მიმდინარეს მშობელი არ იქნება ფესვების სიაში, ან მისი მშობლისთვის მიმდინარე კვანძი პირველი დანაკარგი არ იქნება.

```
void CascadingCut(FibHeapNode *y) {
    FibHeapNode<Key, Compare> *z = y->GetParent();
    if (z != NULL) {
        if (!y->GetMark()) {
            y->SetMark(true);
        }
        else {
            this->Cut(y, z);
            this->CascadingCut(z);
        }
    }
}
```

Cut მეთოდი შლის x კვანძს მისი მშობლის შვილების სიიდან და გადააქვს ის ფესვების სიაში.

```
void Cut(FibHeapNode *x) {

	x->RemoveFromChildList();

	this->AddToRootList(x);

	x->SetMark(false);

}
```

SetMark მეთოდი ცვლის x კვანძის mark თვისებას false/true-თი, რაც იმას ნიშნავს, რომ შესაბამისად მას არ დაუკარგავს არცერთი შვილი ან მას დაკარგული ჰყავს 1 შვილი.

კვანძის წაშლის ამორტიზებული ღირებულება

დავთვალოთ რა არის კვანძის წაშლის ამორტიზებული ღირებულება. ამისათვის, უნდა დავთვალოთ კვანძის გასაღების შემცირების ალგორითმის ამორტიზებული ღირებულება. მას დასჭირდება $O(1)$ დრო პლუს Cut და CascadingCut-ის მუშაობის დროების ჯამი. Cut მეთოდსაც დასჭირდება $O(1)$ დრო. თუ დავუშვებთ, რომ Decrease Key პროცედურა იწვევს CascadingCut-ის c გამოძახებას, მაშინ ის გამოიწვევს CascadingCut-ის $(c-1)$ რეკურსიულ გამოძახებას. რადგან, ერთი CascadingCut-ის გამოძახება საჭიროებს $O(1)$ დროს, გამოდის, რომ Decrease Key პროცედურის რეალური ღირებულებაა O(c).

ახლა განვსაზღვროთ პოტენციალის ცვლილება. ვთქვათ H არის ფიბონაჩის ხე უშუალოდ FibHeap-Decrease-Key() -ის შესრულების წინ. ამ პროცედურაში Cut()-ის გამოძახება ქმნის ახალ ხეს, რომელშიც დამატებულია x კვანძი ფესვების სიაში (თუ მისი მნიშვნელობა გახდა მშობლის მნიშვნელობაზე ნაკლები) და მისი Mark ატრიბუტი ხდება False (თუმცა ეს ატრიბუტი შესაძლოა თავიდანვე იყო False). Cascading_Cut()-ის ყოველი გამოძახება ბოლოს გარდა, ამოჭრის დაჭდევებულ კვანძს და მის Mark ველში ჩაწერს False-ს. საბოლოოდ, ფიბონაჩის გროვა შეიცავს $t(H)+c$ კვანძს ფესვების სიაში ($t(H)$-რაც თავიდან იყო, $(c-1)$ ცალი CascadingCut()- ების შედეგად დამატებული და x კვანძი რომელიც გახდა ფესვი) და

მაქსიმუმ $m(H)-c+2$ დაჭდევებულ კვანძს ($c-1$ კვანძის Mark ველი გახდებოდა False, მაგრამ შესაძლოა ბოლო გამოძახება დამთავრდა მშობლის დაჭდევებით). აქედან გამომდინარე პოტენციალი შეიცვლებოდა შემდეგნაირად:

$$(t(H)+c)+2(m(H)-c+2))-\left(t(H)+2m(H)\right)=4-c$$

მივიღეთ, რომ Decrease Key-ს ამორტიზებული ღირებულებაა: $O(c)+4-c=O(1)$

კვანის წაშლის პროცედურის ამორტიზებული დრო არის გასაღების შემცირების და მინიმალური კვანძის ამოჭრის პროცედურების ჯამი, ანუ: $O\left(D(n)\right)+O(1)=O(D(n))$

მაქიმალური ხარისხის განსაზღვრა

ჩვენ მივიღეთ, რომ მინიმუმის ამოჭრის და ნებისმიერი კვანძის წაშლის ალგორითმების ამორტიზებული ღირებულება არის $O(D(n))$. ახლა ვაჩვენოთ, რომ n კვანძიან ფიბონაჩის გროვაში მაქსიმალური ხარისხის, ანუ $D(n)$-ის ზედა საზღვარია $O(\log(n))$. უფრო ზუსტად, ვაჩვენოთ, რომ $D(n) < [\log_{\phi} n]$, სადაც $\phi = \frac{1+\sqrt{5}}{2} \approx 1.619$

ყოველი x კვანძისთვის size(x)-ით აღვნიშნოთ იმ კვანძების რაოდენობა, რომლებიც შედიან x-ის ქვეხეში, x-ის ჩათვლით. ვაჩვენოთ , რომ size(x) დამოკიდებულია x.degree-ზე ექსპონენციალურად.

ლემა 1: ვთქვათ x არის ნებისმიერი კვანძი ფიბონაჩის გროვაში და x.degree=k. დავუშვათ $y_1, y_2, \dots, y_k$ არიან x-ს შვილები, დალაგებული იმ თანმიმდევრობით, რა თანმიმდევრობითაც გახდნენ x-ს შვილები. მაშინ $y_1.degree \geq 0$ და $y_i.degree \geq i-2$ ყოველი I = 2,3,…,k-სთვის.

დამტკიცება: ის ფაქტი, რომ $y_1.degree \geq 0$ ცხადია.

ყოველი i-სათვის დაწყებული 2-დან, როდესაც y_i აიკინძა x-თან, ამ დროს $y_1, y_2, \dots, y_{i-1}$ უკვე იყვნენ x-ს შვილები. ანუ იმ მომენტისათვის $x.degree \geq \mathrm{i} - 1$. ასევე $y_i.degree \geq i-1$, რადგან y_i მხოლოდ იმ შემთხვევაში აიკინძებოდა x-თან, თუ $x.degree = y_i.degree$. აკინძვის შემდეგ y_i-მა დაკარგა მაქსიმუმ 1 შვილი, რადგან 2 რომ დაეკარგა, ის ამოიჭრებოდა x-ს შვილებიდან. ამიტომ, $y_i.degree \geq i-2$ ■

ლემა 2: ყოველი მთელი $k > 0$-სთვის სამართლიანია ფორმულა: $F_{k+2} = 1 + \sum_{i=0}^{k} F_i$.

დამტკიცება: დავამტკიცოთ ინდუქციით.

როდესაც $k = 0$;

$$1 + \sum_{i=0}^{0} F_i = 1 + F_0 = 1 + 0 = F_2$$

ინდუქციის დაშვება:

$$F_{k+1} = 1 + \sum_{i=0}^{k-1} F_i$$

აქედან გამომდინარე:

$F_{k+2} = F_k + F_{k+1} = F_k + \left(1 + \sum_{i=0}^{k-1} F_i\right) = 1 + \sum_{i=0}^{k} F_i$ ■

ლემა 4: ვთქვათ x არის ნებისმიერი კვანძი ფიბონაჩის გროვიდან და $k = x.degree$. მაშინ,

$$size(x) \geq F_{k+2} \geq \phi^k$$

დამტკიცება: s_k-თი აღვნიშნოთ ნებისმიერ ფიბონაჩის გროვაში ნებისმიერ k ხარისხის მქონე კვანძებს შორის მინიმალური შესაძლო ზომა. ტრივიალურია, რომ $s_0 = 1$ და $s_1 = 2$.

s_k არის მაქსიმუმ size(x) ზომის. ზოგადად $size(x) \geq s_{x.degree}$, რადგან კვანძზე შვილის დამატება ვერ შეამცირებს კვანძის ზომას. s_k-ს ზომა მონოტონურად იზრდება k-სთან ერთად. განვიხილოთ, რაიმე z კვანძი ფიბონაჩის გროვიდან ისეთი, რომ z.degree=k და $size(z) = s_k$. რადგან $s_k \leq size(x)$, s_k-ს ქვედა ზღვრის გამოთვლით ჩვენ გამოვთვლით size(x)-ის ქვედა ზღვარს. ვთქვათ $y_1, y_2, \dots, y_k$ არიან z-ს შვილები დალაგებული იმ თანმიმდევრობით, რა თანმიმდევრობითაც გახდნენ z-ს შვილები. შემდეგ ფორმულაში s_k-ს ანგარიშში z-ს რაოდენობად ემატება ერთი. აგრეთვე, მისი პირველი შვილის, y_1-ის რაოდენობად ემატება აგრეთვე ერთი.

$$size(x) \geq s_k = 2 + \sum_{i=2}^{k} size(y_i) \geq 2 + \sum_{i=2}^{k} s_{y_i.degree} \geq 2 + \sum_{i=2}^{k} s_{i-2}$$

უტოლობის ბოლო ნაწილი გამომდინარეობს ლემა 1-დან. ($y_i.degree \geq i-2$) და s_k-ს მონოტონურობისგან ($s_{y_i}.degree \geq s_{i-2}$).

ინდუქციით ვაჩვენოთ, რომ ეს სამართლიანია ყოველი არაუარყოფითი მთელი k-სთვის. ინდუქციის ბაზა, როცა k=0 და k=1 ტრივიალურია.

ინდუქციის ბიჯი: ვთქვათ $k \geq 2$ და დავუშვათ $s_i \geq F_{i=2}$, ყოველი i=0, 1, 2,…,k-1-სთვის. საბოლოოდ გვექნება:

$$s_k \geq 2 + \sum_{i=2}^{k} s_{i-2} \geq 2 + \sum_{i=2}^{k} F_i = 1 + \sum_{i=0}^{k} F_i = F_{k+2} \geq \phi^k$$

ჩვენ ვაჩვენეთ, რომ $size(x) \geq s_k \geq F_{k+2} \geq \phi^k$. ■

შედეგი: n-კვანძიან ფიბონაჩის გროვაში მაქსიმალური ხარისხი $D(n)$ არის $O(logn)$ რიგის.

დამტკიცება: ვთქვათ x არის n-კვანძიანი ფიბონაჩის გროვის კვანძი და $x.degree = k$. ლემა 4-ს თანახმად გვაქვს: $n \geq size(x) \geq \phi^k$. ϕ-ფუძიანი ლოგარითმის ამოხსნა გვაძლევს, რომ $k \leq \log_\phi n$ (რადგან k მთელი რიცხვია, $k \leq [\log_\phi n]$). აქედან გამომდინარე, ნებისმიერი კვანძის მაქსიმალური ხარისხი $D(n)$ არის $O(logn)$ რიგის. ■

ფიბონაჩის გროვა დალაგების მიმართებით (კომპარატორით)

ფიბონაჩის გროვის ზემოთ აღწერილი იმპლემენტაცია სწორი და გამართულია, თუმცა ის არ არის მოქნილი, რადგან იგი გასაღებში ინახავს მხოლოდ მთელ რიცხვებს. ეს არ არის მისაღები სტილი იმ გარემოების გათვალისწინებით, რომ თანამედროვე C++-ის შიდა ან გარე ბიბლიოთეკებში არცერთი სტრუქტურის გასაღების ტიპი არ არის შეზღუდული. ჩვენი მიზანი იყო, რომ ეს შეზღუდვა აღმოგვეფხრა და გაგვეხადა ჩვენი სტრუქტურა გასაღების ტიპისგან დამოუკიდებელი. ამ ყველაფრის შესაძლებლობას გვაძლევს თანამედროვე C++-ის template-ის ფუნქციები. ტიპის განზოგადებასთან ერთად აუცილებელი გახდა შედარების ოპერაციის განსაზღვრა, რომელიც სხვადასხვა ტიპებისთვის სხვადასხვა პრინციპით სრულდება. მაგალითად, რიცხვების შემთხვევაში შედარების ოპერაცია არ იცვლება, მაგრამ თუ ჩვენი სტრუქტურის გასაღები გვინდა იყოს ობიექტი, მაშინ გაურკვეველია ობიექტების შედარება როგორ უნდა მოხდეს.

ჩვენ შევცვალეთ ფიბონაჩის გროვის კლასის განსაზღვრება, რომელიც შემდეგნაირად გამოიყურება:

template<typename Key, typename Compare = std::less<Key>>.

როგორც ხედავთ, გაჩუმების პრინციპით, დალაგების მიმართება, ანუ შედარების ოპერატორი (რასაც ხშირად ვუწოდებთ კომპარატორს) არის ნაკლებობის მიმართება. C++-ის კომპარატორები არის განსაზღვრული <functional> ბიბლიოთეკაში, მათ შორის less კომპარატორიც. Less-თან ერთად მასში გაერთიანებულია მეტობის, მეტობის-ან-ტოლობის, ნაკლებობის-ან-ტოლობისა და სხვა მრავალი კომპარატორი. კომპარატორი ჩანს ტემპლიტის განსაზღვრებაში, თუმცა მისი შეყვანა და ინიციალიზება ხდება კლასის კონსტრუქტორში. FibHeap(Compare comp = Compare()) { this->comp = comp; .... }. შედარების ოპერაციის შესასრულებლად ვიძახებთ კომპარატორს, როგორც ფუნქციას და გადავცემთ ორ შესადარებელ წყვილს: if (cmp(a, b)) {… }.

პრიორიტეტების რიგი

პრიორიტეტების რიგი [3] არის აბსტრაქტული მონაცემთა ტიპი, რომლის ყოველ ელემენტს აქვს პრიორიტეტი. მასში მაღალი (დაბალი) პრიორიტეტის ელემენტი მუშავდება სხვა ელემენტების წინ. თუ ორ ელემენტს გააჩნია ერთი და იგივე პრიორიტეტი, ისინი სრულდება მათი რიგში დალაგების მიხედვით.

პრიორიტეტების რიგს როგორც წესი უნდა ჰქონდეს მინიმუმ შემდეგი ოპერაციები:

- empty() - არის თუ არა პრიორიტეტების რიგში ელემენტები;
- insert() - პრიორიტეტების რიგში ელემენტის ჩასმა;
- pop() - პრიორიტეტული ელემენტის ამოჭრა რიგიდან;

ჩვენ გამოვიყენეთ ფიბონაჩის გროვა პრიორიტეტების რიგის კონტეინერად. ჩვენს რიგს ასევე დავუმატეთ შემდეგი მეთოდები:

- top() - მინიმალური ელემენტის ამოკითხვა რიგიდან;
- size() - პრიორიტეტების რიგში ელემენტების რაოდენობა;
- sort() - ალაგებს ობიექტის შექმნისას გადმოცემულ კონტეინერის ელემენტებს;

პრიორიტეტების რიგის კლასის კონსტრუქტორს აქვს შემდეგი სახე:

```
PriorityQueue(Iterator start, Iterator end, Compare comp = Compare()) {
    this->comp = comp;
    //Create Fibonacci Heap
    this->fibheap = new FibHeap<Key, Compare>();
    //Insert each element to Heap
    while (start != end) {
        this->fibheap->FibHeapInsert(&(*start));
```

```
                start++;
            }
        }
```

ფიბონაჩის გროვის შექმნა და შევსება ხდება სწორედ პრიორიტეტების რიგის ობიექტის შექმნისას. შემდეგ კი შეგვიძლია გამოვიძახოთ ზემოხსენებული მეთოდები სასურველი შედეგის მისაღებად.

მაგალითად, თუ გვინდა დავასორტიროთ ვექტორის ელემენტები, main ფუნქცია გამოიყურება შემდეგნაირად:

```
PriorityQueue<double> *queue = new PriorityQueue<double>(arr.begin(), arr.end());
queue->sort();
```

sort() მეთოდი თავისმხრივ ასრულებს შემდეგ ოპერაციებს:

```
void sort() {
    while (!this->empty()) {
        *start = this->pop();
        start++;
    }
}
```

Pop() მეთოდი კი იძახებს ფიბონაჩის გროვის ExtractMin მეთოდს.

ჩვენ შემთხვევაში, პრიორიტეტების რიგის empty(), insert(), top() და size() მეთოდების ამორტიზებული ღირებულებაა $O(1)$ დრო, pop მეთოდის ამორტიზებული ღირებულებაა $O(\log n)$, სადაც n არის პრიორიტეტების რიგში ელემენტების რაოდენობა.

რა თქმა უნდა პრიორიტეტების რიგსაც არ უნდა ჰქონდეს გასაღების ტიპზე შეზღუდვა. ამიტომ, ჩვენ შევცვალეთ პრიორიტეტების რიგის კლასის განსაზღვრება და ის ახლა შემდეგნაირად გამოიყურება:

template<typename Key, typename Compare = std::less<Key>, class Iterator = typename std::vector<Key>::iterator>.

ჩვენს პრიორიტეტების რიგსაც აქვს კომპარატორი, რომელიც ავტომატურად იმოქმედებს ფიბონაჩის გროვაზე. ასევე, შემოვიღეთ Iterator ტიპი, რომელიც გაჩუმებით არის std::vector-ის იტერატორი (თუმცა შეიძლება ნებისმიერი სხვაც იყოს).

დეიქსტრას ალგორითმი

დეიქსტრას ალგორითმი გვაძლევს შეწონილ გრაფში მინიმალური გზების პოვნის საშუალებას. ალგორითმის ერთ-ერთი გავრცელებული ვარიანტი არის ფიქსირებული კვანძიდან ყველა დანარჩენ კვანძებამდე უმოკლესი გზის პოვნა, რომელსაც ჩვენ განვიხილავთ. მაგალითად, თუ გრაფის კვანძები წარმოადგენენ ქალაქებს, მათი შემაერთებელი წიბოები არიან ქალაქების შემაერთებელი გზები, ხოლო წიბოების წონები - ქალაქებს შორის მანძილებს, დეიქსტრას ალგორითმი შეგვიძლია გამოვიყენოთ კონკრეტული ქალაქიდან ყველა სხვა ქალაქამდე უმოკლესი მანძილების გასაგებად.

ორიგინალი ალგორითმის [2] ფსევდო კოდი:

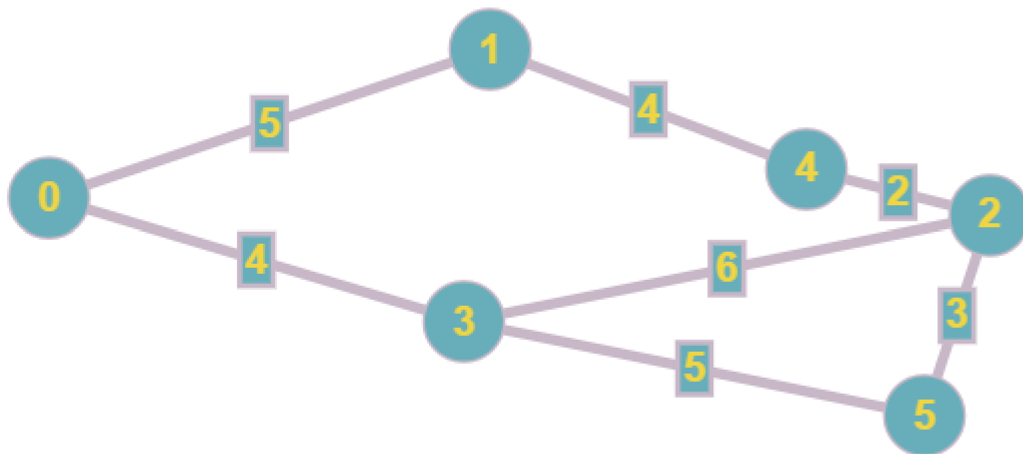
1. ყველა კვანძს გარდა საწყისი კვანძისა მივანიჭოთ მანძილებად უსასრულობა, ხოლო საწყის კვანძს - 0
2. თუ ყველა წვერო დამუშავებულია, ალგორითმი წყვეტს მუშაობას;
3. ვირჩევთ მინიმალური მანძილის მქონე წვეროს დაუმუშავებელი წვეროებიდან;
4. U წვეროს ყოველი უნახავი მეზობელი წვეროსათვის განვიხილოთ ახალი მანძილი, რომელიც იქნება u წვეროს მიმდინარე მანძილისა და u-ს თითოეული მეზობლის შემაერთებელი წიბოს წონის ჯამი. თუ ეს ჯამი ნაკლებია u-ს მეზობლის მანძილზე, მისი მანძილი ჩავანაცვლოთ ახალი მნიშვნელობით.
5. U-ს ყველა მეზობლის გავლის შემდეგ მოვნიშნოთ u როგორც დამუშავებული და დავუბრუნდეთ მე-2 ნაბიჯს.

რადგან მე-3 ნაბიჯზე გვჭირდება მინიმუმის ამორჩევა კვანძებს შორის, შეგვიძლია გამოვიყენოთ ჩვენს მიერ შემუშავებული პრიორიტეტების რიგი. კერძოდ, ყველა კვანძს თავისი მანძილებით ჩავალაგებთ პრიორიტეტების რიგში და შემდეგ, სათითაოდ ამოვიღებთ მინიმალურ ელემენტს და შევასრულებთ მე-4 და მე-5 ნაბიჯებს.

დეიქსტრას ალგორითმის ფსევდოკოდი პრიორიტეტების რიგის გამოყენებით გამოიყურება შემდეგნაირად:

1. ყველა კვანძს გარდა საწყისი კვანძისა მივანიჭოთ მანძილებად უსასრულობა, ხოლო საწყის კვანძს - 0 და დავამატოთ ისინი პრიორიტეტების რიგში;
2. სანამ პრიორიტეტების რიგში არის წვეროები შევასრულოთ შემდეგი ოპერაციები:
   - ამოვჭრათ რიგიდან მინიმალური ელემენტი u;
   - მინიმალური ელემენტის შესაბამისი წვეროს ყოველი v მეზობლისთვის შევასრულოთ შემდეგი:
     - თუ u-ს მანძილი და u და v წვეროებს შორის მანძილის (u და v წვეროების შემაერთებელი წიბოს წონა) ჯამი sum ნაკლებია v-ს მანძილზე, v-ს მანძილი გავხადოთ sum და პრიორიტეტების რიგში v გასაღების მქონე კვანძს შევუმციროთ გასაღები და გავხადოთ sum-ის ტოლი.

კოდში ჩვენ გრაფის წვეროებს შორის მანძილები გამოვსახეთ ორგანზომილებიანი მასივის მეშვეობით. მაგალითად, სურ. 2-ზ ე გამოსახულ გრაფს შეესაბამება A მატრიცა.



სურ.2

$$A = \begin{matrix} 0 & 5 & 0 & 4 & 0 & 0 \\ 5 & 0 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 6 & 2 & 3 \\ 4 & 0 & 6 & 0 & 0 & 5 \\ 0 & 4 & 2 & 0 & 0 & 0 \\ 0 & 0 & 3 & 5 & 0 & 0 \end{matrix}$$

მატრიცა ისეა მოწყობილი, რომ შესაბამისი წვეროების ინდექსების გადაკვეთაზე წერია ამ წვეროების შემაერთებელი წიბოების წონები. მაგალითად, 0 და 3 წიბოების წონა არის $A_{0,3} = A_{3,0} = 4$. თუ წვეროები არ არიან მეზობლები, მათი შესაბამისი ინდექსების ადგილზე A მატრიცაში წერია 0.

ჩვენს ალგორითმშიც ასეთი მიდგომა გამოვიყენეთ. კერძოდ, გრაფის კლასს აქვს შემდეგი ველები

- int size; // გრაფის წვეროების რაოდენობა
- int **length; //ორგანზომილებიანი მასივი (მატრიცა), რომელიც ან გადმოეცემა მომხმარებლის მიერ, ან იქმნება შემთხვევითი რიცხვებით
- Compare comp; // კომპარატორი
- int *distance; //კვანძებამდე უმოკლესი მანძილების შემნახველი მასივი

კლასის კონსტრუქტორში ხდება length ორგანზომილებიანი მასივის დაკომპლექტება გადმოცემული მასივით, ან შემთხვევითი რიცხვებით, ხოლო distance მასივში იწერება უსასრულობები (თავდაპირველად ყველა წვეროს აქვს მანძილი უსასრულობა) გარდა საწყისი წვეროსი. მისი საწყისი მანძილია 0.

იმისათვის, რომ პრიორიტეტების რიგში დავამატოთ წვერო და მისი მანძილი, ჩვენ ფიბონაჩის გროვის გასაღებად ვიყენებთ std::pair-ს, რომელშიც პირველი ელემენტი არის წვეროს ნომერი, ხოლო მეორე - მისი მანძილი.

გრაფის კლასს არ აქვს უნივერსალური გასაღები. ის მუშაობს მხოლოდ დანომრილ გრაფებთან, ხოლო პრიორიტეტების რიგის გასაღები არის std::pair. ასევე, ჩვენს კლასს არ აქვს კომპარატორი. თუმცა, პრიორიტეტების რიგს კომპარატორი უნდა გადავცეთ, რათა ფიბონაჩის გროვაში წყვილები ერთმანეთს შევადაროთ. ამ მიზნით ჩვენ შემოვიღეთ კომპარატორი, რომელიც ადარებს წყვილების მეორე მნიშვნელობებს (მანძილებს).

ჩვენს მიერ შემუშავებული დეიქსტრას ალგორითმის კოდი გამოიყურება შემდეგნაირად:

```
}

 int *Dijkstra() {
	PriorityQueue<std::pair<int, int>, Compare> *queue = new PriorityQueue<std::pair<int, int>, Compare>(this->size);
	for (int i = 0; i < this->size; i++) {
		std::pair<int, int> pair(i, this->distance[i]);
		queue->add(&pair);
	}
	while (!queue->empty()) {
		std::pair<int, int> u = queue->pop();
		for (int v = 0; v < this->size; v++)
			if (this->length[u.first][v] > 0) {
				int alt = this->distance[u.first] + this->length[u.first][v];
				if (alt < this->distance[v]) {
					this->distance[v] = alt;
					std::pair<int, int> pair(v, alt);
					queue->decreaseKey(&pair);
				}
			}
	}
	return this->distance;
}
```

კომპარატორს აქვს შემდეგი სახე:

```
struct comp {
	bool operator()(const std::pair<int, int> lhs,  const std::pair<int, int> rhs)
	{
		return lhs.second < rhs.second;
	}
};
```

შეფასება

დეიქსტრას ალგორითმის ფიბონაჩის გროვით იმპლემენტაციის ეფექტიანობის შესაფასებლად, ჩვენ მთავარი პროექტი გავუშვით საშუალო სიმძლავრის მქონე კომპიუტერზე 1000, 10 000 და 20 000 წვეროიან გრაფებზე და შევადარეთ სიჩქარე დეიქსტრას ორიგინალი ალგორითმის იმპლემენტაციას. შედეგები მოცემულია მილიწამებში [4] chrono ბიბლიოთეკის გამოყენებით.

1000 კვანძი:

ფიბონაჩის გროვა	ორიგინალი ალგორითმი
9	39

10 000 კვანძი:

ფიბონაჩის გროვა	ორიგინალი ალგორითმი
348	674

20 000 კვანძი:

ფიბონაჩის გროვა	ორიგინალი ალგორითმი
1445	2862

დასკვნა

ფიბონაჩის გროვის გამორჩეული თეორიული მახასიათებლების გათვალისწინებით, წარმოდგენილ ნაშრომში განხილულია ფიბონაჩის გროვის და პრიორიტეტების რიგის მონაცემთა სტრუქტურების ალგორითმების თეორიული და პრაქტიკული იმპლემენტაციის საკითხები. იმპლემენტაციის ეფექტიანობის საკითხი გასინჯული იქნა დეიქსტრას ალგორითმის მაგალითზე. ჩატარებული ტესტები ადასტურებს, რომ ფიბონაჩის გროვების გამოყენებით შექმნილი იმპლემენტაცია საგრძნობლად უკეთესია დეიქსტრას ალგორითმის სწორხაზოვან იმპლემენტაციასთან შედარებით.

ლიტერატურა

[1] Cormen T., Leiserson C., Rivest R., Stein C. Introduction to Algorithms, 3rd Edition. The MIT Press, 2009

[2] Dijkstra Algorithm - https://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/

[3] Priority Queue - https://en.cppreference.com/w/cpp/container/priority_queue

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