ივანე ჯავახიშვილის სახელობის თბილისის სახელმწიფო უნივერსიტეტი
ზუსტ და საბუნებისმეტყველო მეცნიერებათა ფაკულტეტი

კომპიუტერული მეცნიერების დეპარტამენტი

ნიკოლოზ გრძელიძე

მეხუთე საკონფერენციო მოხსენება

BST ხის დაბალანსების პარალელური ალბორითმი

თბილისი 2017

სარჩევი

ნაშრომის დასახელება

ორობითი ხის დაბალანსების, სწრაფი ალგორითმის საჭიროება წარმოიშვა, სადოქტორო თემის - მონაცემთა ჰიბრიდული სტრუქტურების შემუშავებისას. სტანდარტული ალგორითმების მოდიფიცირებით და ახალი ალგორითმების გადასინჯვით, დამაკმაყოფილებლად მივიჩნიეთ QBalance ალგორითმი. დაბალანსების უფრო სწრაფი მეთოდის მისაღებად მოვახდინეთ მისი გაპარალელება, რაც, ცხადია აუმჯობესებს ალგორითმის დროით შეფასებას. შესაბამისად, გვაქვს ორობითი ხის დაბალანსების პარალელური ალგორითმი QBalance Parallel.

QBalance

ალგორითმი, ორობითი ხის დასაბალანსებლად იყენებს მის წარმოდგენას "სიის" სახით, სიაში იგულისხმება სტრუქტურა, რომელზე იტარაცია ხორციელდება წრფივ დროში.
ორობითი ხის სიად წარმოდგენისათვის, შემუშავებული იქნა რამდენიმე მეთოდი, ყველაზე კარგი შედეგი აჩვენა გადაგვარებული ხის აგებამ(vine tree) -ორობითი ხე სადაც თითოეულ კვანძს მხოლოდ მარჯვენა შვილი ყავს. ალგორითმის მუშაობის შემდეგ ეტაპზე ხდება გადაგვარებული ხის დაბალანსება. დაბალანსების მეთოდი რეკურსიულია, რეკურსიის ბაზას წარმოადგენს ორზე ნაკლები ან ტოლი სიგრძის ელემენტების სია, რომელიც, ცხადია, დაბალანსებულია. ალგორითმის წინა ვერსიაში რეკურსიის ბაზას წარმოადგენდა ერთ ლემენტიანი სია. ასეთ დროს, რეკურსიული გამოძახებების რიცხვი აღემატება მიმდინარე რეალიზაციაში წარმოდგენილ გაჩერების პირობას. ალგორითმის იდეა შემდეგია, ბმული სია იყოფა ორ ნაწილად, ინდექსით შუა ელემენტის მიმართ, რომელიც უნდა გახდეს ფესვი, მის მარცხნივ და მარჯვნივ მყოფი, დაუბალანსებელი სიისათვის. გაყოფის შემდეგ ალგორითმი რეკურსიულად ეშვება გაყოფის შემდეგ მიღებული სეგმენტებისათვის. ალგორითმი რეკურსიის ბაზიდან, დაბრუნებისას იყენებს წინასწარ დამახსოვრებულ კვანძს(პარამეტრად გადაცემულს), რომლის შვილის უნდა გახდეს მის მერ დაბალანსებული ქვეხის ფესვი.
ალგორითმის პროგრამული რეალიზაცია შემდეგია:

```
Node<T>* Tree<T, Compare>::qBalance(Node<T>*& node, int m){
        if (m < 2) {
                Node<T>* tmp = node;
                node = node->right;
                tmp->left = tmp->right = nullptr;
                return tmp;
        }
        if (m == 2){
                Node<T>* tmp = node;
                node = node->right->right;
                tmp->left = nullptr;
                tmp->right->right = nullptr;
                return tmp;
        }
        int q = m / 2;
        Node<T>* a = qBalance(node, q);
        Node<T>* b = node;
        b->left = a;
        node = node->right;
        a = qBalance(node, m - q - 1);
```

```
        b->right = a;
        return b;}
```

კვანძის სტრუქტურა

კვანძის სტრუქტურა მინიმალისტურია, აქვს განზოგადებული ტიპის მხარდაჭერა, ამოღებულია მშობელ კვანძზე მიმთითებელი. კვანძის შვილების წარმოსადგენად, გამოყენებულია left და right მიმთითებლები. მართალია, შვილი კვანძების წარმოდგენა მასივის სახით, ამარტივებს პროგრამულ კოდს, თუმცა სტანდარტული იმპლემენტაცია უფრო ინტუიტიური და მარტივად წასაკითხია.

```
template<typename T>
struct Node
{
        T key;
        Node* left;
        Node* right;
        Node();
        Node(T keyValue);
};

template<typename T>
Node<T>::Node(T keyValue)
{
        key = keyValue;
        left = nullptr;
        right = nullptr;
}

template<typename T>
Node<T>::Node()
{
        //TODO initial
}
#endif
```

ორობითი ხის სტრუქტურა

ძებნის ორობითი ხის სტრუქტურა, დაბალნსების QBalance ალგორითმის მოთხოვნებიდან გამომდინარე, შეიცავ ინფორმაციას კვანძების რაოდენობაზე, რადგან აღნიშნული თვისების დადგენა არ მოხდეს დინამიურად, წრფივ დროში. ორობითი ხე პარამეტრიზირებულია, პირველი პარამეტრი მიუთითებს კვანძის ტიპს, ხოლო მეორე პარამეტრი შემდარებელ მეთოდს, რომელიც განსაზღვრულია კვანძის ტიპისათვის. გულისხმობის პრინციპით, შემდარებელი მეთოდი არის std::greater_equal, თუმცა შეგვიძლია ცხადად გადავცეთ კონსტრუქტორიში. შესაბამისად, იმპლემენტაციაში ტვაქვს ორი კონტრუქტორი:

1)
```
template<typename T, typename Compare>
Tree<T, Compare>::Tree(Compare& comp = Compare())
{
        cmp = comp;
        root = nullptr;
}
```

2)
```
template<typename T, typename Compare>
Tree<T, Compare>::~Tree() {
        destroyTree(root);
}
```

ორობითი ხის დესტრუქტორის იმპლემენტაციაში, გამოყენებულია სტეკი, რადგან კვანძის წაშლის რეკურსიული მეთოდი, დიდი რაოდენობის მონაცემზე იწვევდა გამოძახებების სტეკის გადავსებას.

```
template<typename T, typename Compare>
void Tree<T, Compare>::destroyTree(Node<T>* node) {
        Stack<Node<T> *> s;
        Node<T>* current = node;
        while (!s.empty() || current) {
                if (current) {
                        s.push(current);
                        current = current->right;;
                }
                else {
                        current = s.top();
                        s.pop();
                        current->right = root;
                        root = current;
                        Node<T> * for_delete = current;
                        current = current->left;
                        delete for_delete;

                }
        }
}
```

QBalance Successor

როგორც უკვე აღვნიშნე ალგორითმი ორ ეტაპიანია, პიველ ეტაპზე ხდება გარემოს მომზადება, ხოლო მეორე ეტაპზე დაბალანსება. პირველ ეტაპზე უარის თქმა შესაძლებელია, თუ განისაზღვრება მომდევნო კვანძის განსაზღვრის დინამიური მეთოდი -next. ამისათვის საჭიროა კვანძის სტრუქტურის მოდიფიცირება, მასში მშობელი კვანძის მიმთითებელით და სიდიდით მომდევნო კვანძის პოვნის მეთოდის რეალიზებით.

```
template<typename T>
struct Node
{
	T key;
	Node* left;
	Node* right;
	Node* parent;
	Node();
	Node(T keyValue);
};
```

სიდიდით მომდევნო კვანძის პოვნის ალგორითმი მარტივია -თუ კვანძი მისი მშობლის მარცხენა შვილია და არ ყავს მარჯვენა ქვეხე, მაშინ სიდიდით მომდევნო კვანძი არის ამ კვანძის მშობელი. თუ კვანძს მარჯვენა ქვეხე ყავს მაშინ სიდიდით მომდევნო კვანძი არის მინიმალური კვანძი მის მარჯვენა ქვეხეში.
თუ კვანძი მისი მშობლის მარჯვენა შვილია, იტერაციით გადავდივართ მის მშობელზე ვიდრე მშობელი არ გახდება მისი მშობლის მარცხენა შვილი.

```
template<typename T, typename Compare>
Node<T>* Tree<T, Compare>::findMin(Node<T>* node) {
	while (node->left != nullptr) {
		node = node->left;
	}
	return node;
}

template<typename T, typename Compare>
Node<T>* Tree<T, Compare>::next(Node<T>* node) {
	if (node ->right != nullptr) {
		return findMin(node->right);
	}
	else {
		Node<T>* tmp = node;
		while (tmp->parent != nullptr && tmp != tmp->parent->left) {
			tmp = tmp->parent;
		}
		return tmp->parent;
	}
}
```

პირველი ეტაპის ამოღების შემდეგ, QBalance მეთოდმა განიცადა მცირედი ცვლილება, ალგორითმს პარამეტრად გადაეცემა სიდიდით მინიმალური კვანძი, შემდეგ კვანძზე გადასასვლელად იყენებს დინამიურ next მეთოდს.

```
template<typename T, typename Compare>
Node<T>* Tree<T, Compare>::qBalance(Node<T>*& node, int m)
{
	if (m < 1)  return nullptr;
	if (m == 1)
	{
		Node<T>* tmp = node;
		node = next(node);
		tmp->left = tmp->right = nullptr;
		return tmp;
	}
	int q = m / 2;
	Node<T>* a = qBalance(node, q);
	Node<T>* b = node;
	b->left = a;

	node = next(node);

	a = qBalance(node, m - q - 1);
	b->right = a;
	return b;
}
```

აღნიშნული მეთოდზე გატარდა ტესტები და აჩვენა გაუმჯობესებული შედეგი, შემთხვევით შერჩეული რიცხვებისათვის, პროგრამული კოდი მარტივი და ინტუიტიურია, ალგორითმის მუშაობა კი ერთ ეტაპიანი.

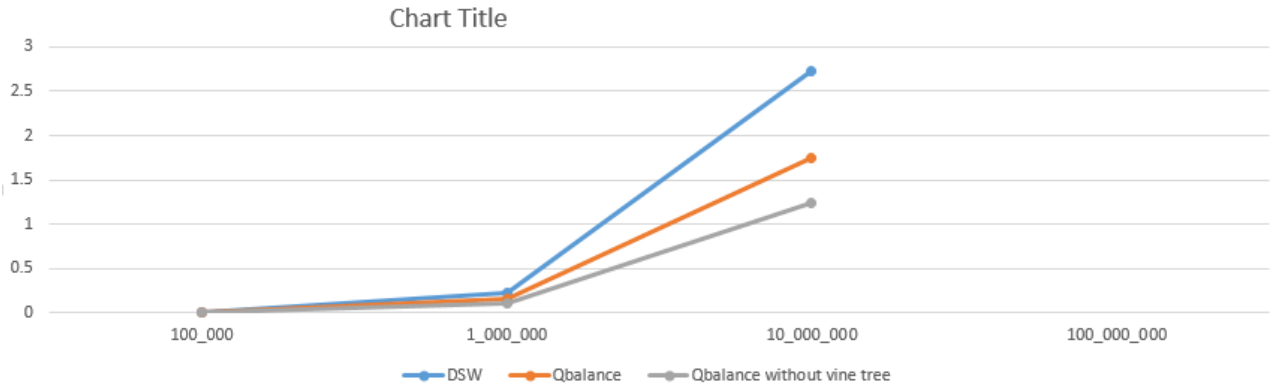
Qbalance Parallel

ბუნებრივია, ალგორითმის დაპარალელება, უკეთეს შედეგს მოგვცემს ვიდრე სერიული, ერთ ნაკადში გაშვებული. QBalance ალგორითმის დაპარალელებისათვის საჭიროა განისაზღვროს თითოეული ნაკადისათვის გამოყოფილი მონაცემთა სია, რომელსაც ნაკადი დააპარალელებს. ამისათვის სიად გარდაქმნის ეტაპზე, ხდება იმ სიების გამოყოფა რომელზეც უნდა იმუშაოს ნაკადებმა, ხლო გამომძახებელ მეთოდში ხდება მიღებული შედეგების გაერთიანება ერთი კვანძის, ფესვის, ქვეშ. აჟამად, სიმარტივისათვის, რეალიზებულია ორ ნაკადში გაშვება. სიად გარდაქმნის მეთოდმა, განიცადა ცვლილება და რეალიზებულია ჩადგმული ციკლების მეშვეობით, გარე ციკლი ორჯერ სრულდება ხოლო შიდა ციკლი N/2 ჯერ.

```
template<typename T, typename Compare>
int Tree<T, Compare>::tree_to_vine(Node<T>* r)
{
	Node<T>* vineTail = r;
	Node<T>* remainder = vineTail->right;

	int mid_point = size / 2;
	int size = 0;

	Node<T>* tempPtr;
	for (int i = mid_point; i <= this->size ; i += mid_point) {

		while (size < i)
		{
			if (remainder->left == nullptr)
			{
				vineTail = remainder;
				remainder = remainder->right;

				size++;
			}
			else
			{
				tempPtr = remainder->left;
				remainder->left = tempPtr->right;
				tempPtr->right = remainder;
				remainder = tempPtr;
				vineTail->right = tempPtr;
			}
		}
		if (mid_node == nullptr) {
			mid_node = vineTail;
		}
	}

	return size;
}
```

ტესტირების შედეგები

QBalance და QBalance Successor ალგორითმების შედარება, ერთი და იმავე, შემთვევით შემოსულ მონაცემებზე.



QBlance და QBalance Parallel ალგორითმების შედარება.