Още за двоичните индексни дървета

Част 1

Структурата двоично индексно дърво(Binary Index Tree, BIT) е сравнително нова в информатиката и в състезателното програмиране, като същевременно с това е изключително проста и удобна и може да се използва широко.

Предистория

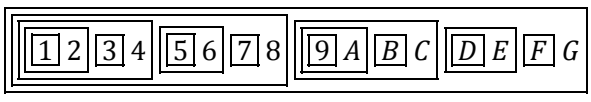
Нека имаме масив A от N числа. Оттук нататък ще приемаме, че масив от елементи започва от 1, т.е. първият елемент на A е $A[1]$, а последният n-ти елемент е $A[n]$. В информатиката често възниква нуждата да се намери сумата на елементите от масива на позиции, принадлежащи на даден интервал. Един вариант да се направи това е като предварително се преизчислят парциалните суми. Нека те са пресметнати в нов масив S. Така $S[i] = A[1] + A[2] + \cdots + A[i]$. Сега, сумата на елементите на позиции $[a; b]$, е $S[b] - S[a-1]$, като сме положили $S[0] = 0$.

Но този подход ни помага само ако стойностите на A са фиксирани и не се променят в хода на програмата. Ако освен намиране на сумата искаме във хода на програмата и да променим стойността на елемент на дадена позиция, се налага да приложим друг подход. Идеята е вместо парциалните суми да се пазят суми на елементите на позиции от „прости" интервали – такива интервали, които са сравнително „балансирани" така, че произволен интервал да се разбива на „малък" брой прости интервали и общия брой на простите интервали да е достатъчно „малък". Fenwick предлага определен набор от прости интервали, които отговарят на горните условия. Във структурата двоично индексно дърво разлагането на произволен интервал на прости интервали е сравнимо със разлагането на произволно число като сума от степените на двойката. Тук ще представим индуктивна дефиниция на двоичното индексно дърво, която е особено удобна за анализ.

Дефиниция:

Двоично индексно дърво:

1. *Двоично индексно дърво (BIT) на масив от 1 елемент е същият масив от 1 елемент.*
2. *Искаме да дефинираме двоичното индексно дърво B на масив от $2n$ елемента A. Нека X и Y са двоичните индексни дървета съответно на първите и вторите n елемента на A. Тогава B е масив от $2n$ елемента, първите n от които са n-те елемента на X, вторите $n-1$ от които са първите $n-1$ елемента на Y, а последният $2n$-ти елемент на B е сумата на всички елементи от A.*



Двоично индексно дърво на 16 елемента. Кутийките показват простите интервали.

Сега всяка позиция участва в логаритмичен брой прости интервали и всеки интервал, започващ от 1, може да се покрие с логаритмичен брой прости интервали. Алгоритмите за работа с двоично индексно дърво са базирани на основната операция *извличане на най-десния бит на число*, която се реализира ефективно чрез побитово изключващо или и използване свойствата на отрицателните числа, записани в допълнителен код.

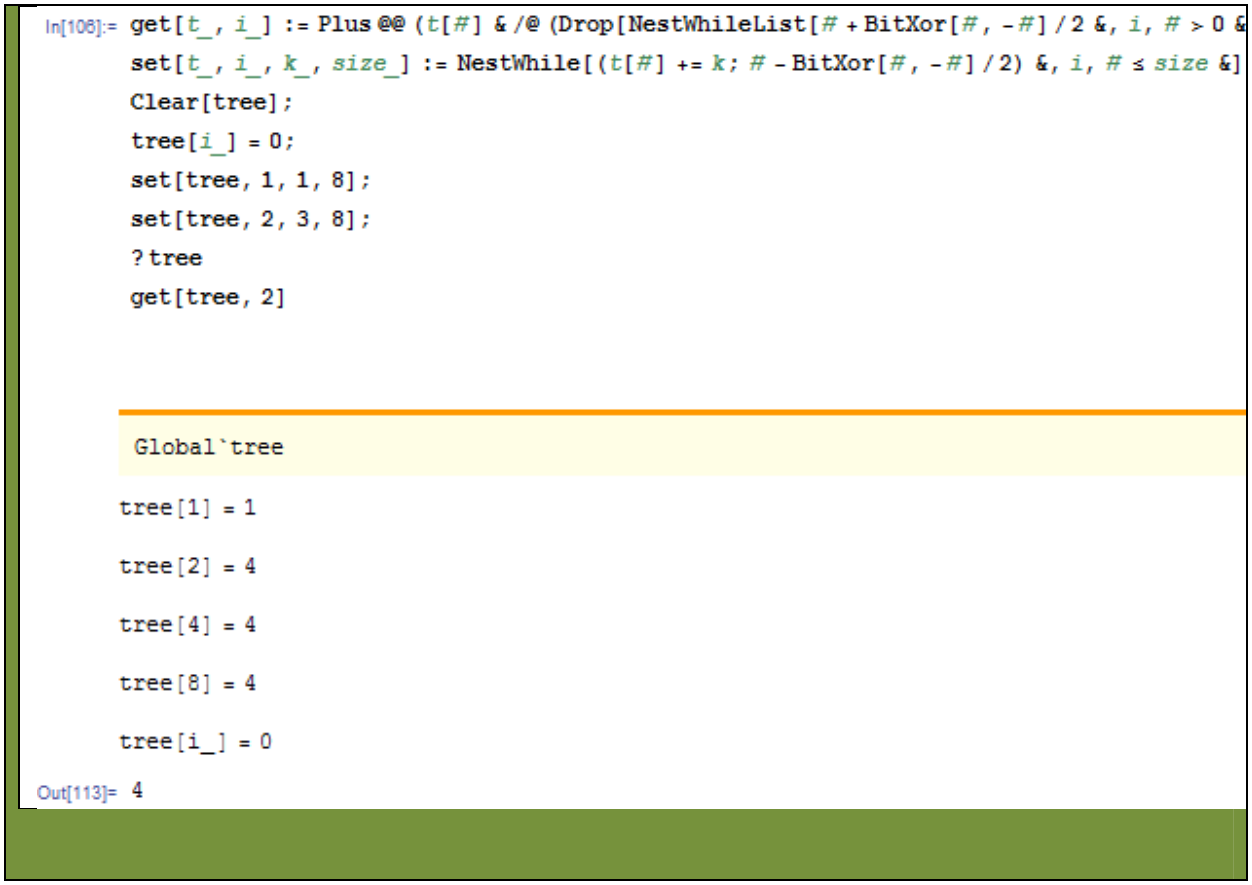
Примерен код 1 – C++

```
//index tree sample
const int max_n = (1<<10);
int t[max_n];

//Adds k to the i'th index in the index tree
void set(int i, int k){
	//t[0] is the actual size of the tree -
	//the number of the indexes
	while(i <= t[0]){
		t[i] += k;
		i += i&(-i);
	}
}

//Gets the cumulative frequency of the first i indexes
int get(int i){
	int r = 0;
	while(i > 0){
		r += t[i];
		i -= i&(-i);
	}
	return r;
}
```

Примерен код 2– Mathematica



Сега ще разгледаме някои по-неизвестни приложения на двоичното индексно дърво.

Индексно дърво на масив от произволен брой елементи

По-внимателен поглед върху двете основни операции на индексното дърво показва, че за действието на дървото не е необходимо броят на елементите да е степен на двойката. Всищност, операциите работят без промяна и със индексни дървета с произволен брой елементи.

Начално строене с линейна сложност

Когато трябва да си постоим индексно дърво от някакви първоначално зададени стойности, естественият начин да го направим, е като прибавим всеки елемент в първоначално празно индексно дърво. Този подход има сложност $O(n \log n)$, който е практически достатъчно добър вариант в почти всички случаи. Не толкова известно е обаче, че строенето на индексно дърво от първоначално зададен масив от стойности може да се извърши за линейно време, и дори *in-place*(със константа допълнителна памет). За практически приложния разликата във времената може да не е видима, но когато имаме ограничена памет, този начин е за предпочитане.

С използване на индуктивната дефиниция на двоичното индексно дърво аглоритъмът се формулира лесно:

Алгоритъм 1:

За стоене на двоично индексно дърво от масив A с $2n$ елемента, in-place:

1. *Построй двоичното индексно дърво на първите n елемента на A(in-place).*
2. *Построй двоичното индексно дърво на вторите n елемента на A(in-place).*
3. *Към последния елемент от масива (с индекс $2n$) прибави елемента с индекс n(последния на първото индексно дърво).*

Сложността на описания алгоритъм е линейна. Това може да се докаже чрез изследване на функцията на сложността:

$$F(n) = 2F\left(\frac{n}{2}\right) + c \Longrightarrow F \in O(n).$$

Примерен код 3 – C++

```
//Builds an index tree in-place in an array
void build(int* a){
	//in a[0] is the size of the index tree - must
	//be a power of 2. The indexes are the values
	//a[1..a[0]] inclusively.
	int i, j, k;
	for(k = 1, j = 2; j <= a[0]; k = j, j<<=1){
		i = j;
		while(i <= a[0]){
			a[i] += a[i-k];
			i += j;
		}
	}
}
```

Намиране на елементите с линейна сложност

Когато сме извършили някакви операции с индексното дърво и искаме да намерим елементите, които го пораждат, обикновено постъпваме така: Намираме всички парциални суми със сложност $O(n \log n)$ и от тях възстановяваме елементите. Новият алгоритъм е подобен на **Алгоритъм 1** и решава проблема с линейна сложност и отново in-place, което е основното му предимство.

Алгоритъм 2:

За намиране на елементите, пораждащи двоично индексно дърво в масив A с $2n$ елемента, in-place:

1. *Извади от последния елемент(с индекс $2n$) елемента с индекс n.*
2. *Разгледай първите и вторите n елемента като двоични индексни дървета и намери елементите им(in-place).*

Сложността на **Алгоритъм 2** е същата като тази на **Алгоритъм 1**, следователно е линейна.

Примерен код 4 – C++

```
//Finds the elements of the index tree, in-place
void take(int* a){
	//in a[0] is the size of the index tree - must
	//be a power of 2. The indexes are the values
	//a[1..a[0]] inclusively.
	int i, j, k;
	for(k = a[0]>>1, j = a[0]; j >= 2; j = k, k>>=1){
		i = j;
		while(i <= a[0]){
			a[i] -= a[i-k];
			i += j;
		}
	}
}
```

Произволни елементи и бинарна операция

Двоичното индексно дърво е добре приспособено за намиране на парциални суми и динамична промяна на елемента на даден индекс. Допълнителните операции, които могат да се реализират с него, като например намиране на сумата на елементите на позиции от интервал, не са вътрешни за структурата, а са базирани на груповите свойства на събирането спрямо числата(*целите или реалните числа, снабдени с операцията събиране, формират група*). По-внимателен поглед върху индексното дърво разкрива, че събирането може да се замени с произволна бинарна операция, стига тя да е асоциативна.

Дефиниция:

Бинарна операция е всяко изображение $\phi : A \times A \to A$, *където* A *е произволно множество.*

Бинарната операция се нарича асоциативна, когато:

$$\phi\big(a, \phi(b, c)\big) = \phi(\phi(a, b), c) = \phi(a, b, c),$$

за всеки три елемента $a, b, c \in A$.

Когато имаме асоциативна операция, ние можем да говорим за *композиция* на елементи:

1. *Композиция на един елемент е този елемент.*
2. *Композиция на* n *елемента е елементът, получен от прилагане на бинарната операция към композицията на първите* $n - 1$ *елемента и последният елемент.*

Сега, обобщеното двоично индексно дърво ще поддържа операцията намиране на композицията на първите i елемента. Допълнителната операция, типична за индексното дърво, за композиране на елемент на дадена позиция с друг, произволен елемент, е коректна само в случай, че

бинарната операция е и комутативна. Също така, за да работи коректно **Алгоритъм 2**, ни трябва аналог на операцията изваждане, т.е. бинарната операция трябва да е и *обратима*.

За акцентиране върху общността на новата задача, предлагаме код, пригоден за работа с произволни данни и бинарна операция.

Примерен код 5 – C++

```
//Classic binary index tree with generic binary operation
template<typename T, int N, T (*Op)(const T&, const T&)>
class index_tree{
public:
	T t[N+1];

	/*
	Empty Constructor.
	*/
	//O(1)
	index_tree(){};

	/*
	Fill constructor.
	*/
	//O(n)
	index_tree(const T& t_){
		for(int i = 1; i <= N; i++)
			t[i] = t_;
	};

	/*
	Array of elements constructor.
	*/
	//O(n)
	index_tree(const T* t_){
		build(t_);
	};

	/*
	Copy constructor.
	*/
	//O(n)
	index_tree(index_tree& tree){
		for(int i = 1; i <= N; i++){
			t[i] = tree.get_value(i);
		}
	};

	/*
	Copies the array to the BIT array.
	*/
	//O(n)
	void copyarr(const T* t_){
		memcpy(t, t_, sizeof(t));
	};
```

```
	/*
	Builds the index tree from an array.
	*/
	//O(n)
	void build(const T* t_){
		copyarr(t_);
		buildp();
	};

	/*
	Builds the index tree in-place from the values
	of the index tree as elements.
	*/
	//O(n)
	void buildp(){
		int i, j, k;
		for(k = 1, j = 2; j <= N; k = j, j<<=1){
			i = j;
			while(i <= N){
				t[i] = Op(t[i-k], t[i]);
				i += j;
			}
		}
	};

	/*
	Gets or sets the BIT value at index i.
	For efficiency, we've made the tree available.
	*/
	//O(1)
	inline T& get_value(int i){
		return t[i];
	};

	/*
	Composes the element on the i'th index with a
	value.

	-> WORKS ONLY WITH COMMUTATIVE OPERATIONS.
	*/
	//O(log n)
	void comp_index(int i, const T& v){
		while(i <= N){
			t[i] = Op(t[i], v);
			i += i&(-i);
		}
	};

	/*
	Extracts the elements in all indexes in an array.

	-> WORKS ONLY WITH INVERTABLE OPERATIONS.
	*/
	//O(n)
	void get_elems(T* t_, T (*iOp)(const T&, const T&)){
		int i, j, k;
		memcpy(t_, t, sizeof(t));
```

```
            for(k = N>>1, j = N; j >= 2; j = k, k>>=1){
                  i = j;
                  while(i <= N){
                        t_[i] = iOp(t_[i], t_[i-k]);
                        i += j;
                  }
            }
      };

      /*
      Gets the partial composition to index i.
      */
      //O(log i)
      T get_comp(int i) const{
            T r = t[i];
            i -= i&(-i);
            while(i > 0){
                  r = Op(t[i], r);
                  i -= i&(-i);
            }
            return r;
      };
.....................................
```

По-нататък ще са ни нужни още няколко нестандартни операции, които първо ще дефинираме тук. На пръв поглед, може да няма смисъл от тях, но те влизат в употреба при работа с двулицево индексно дърво.

При операцията за намиране на композицията на първите няколко елемента се извършва композиране на прости интервали, дъкато се получи търсения интервал. Първата нова операция е просто да се промени реда на композирането на простите интервали – вместо да се започне от първия към последния, да се започне от последния към първия:

Примерен код 5 – продължение – C++

```
      /*
      Gets the partial composition to index i,
      but composes the simple intervals in reversed order.

      -> SAME AS get_comp FOR COMMUTATIVE OPERATIONS.
      */
      //O(log i)
      T get_compr(int i) const{
            T r = t[i];
            i -= i&(-i);
            while(i > 0){
                  r = Op(r, t[i]);
                  i -= i&(-i);
            }
            return r;
```

```
        };
.....................................
```

Литература

Fenwick M. – A New Data Structure for Cumulative Frequency Tables
TopCoder Tutorials