Декартово дърво (Treap, Cartesian tree)

Съдържание:

Декартовото дърво е структура от данни, която комбинира двоично дърво и двоична пирамида (откъдето идва и едно от имената му $tree + heap \Rightarrow Treap$).

По конкретно декартовото дърво е структура от данни, която съхранява двойки (X, Y) в двоично дърво по такъв начин, че е двоично дърво за търсене по X и двоична пирамида по Y. Ако приемем, че всички X и всички Y са различни, може да видим, че ако някой връх на дървото съдържа стойности (X_0, Y_0), то всички възли в лявото поддърво имат $X < X_0$, всички върхове в дясното дърво имат $X > X_0$, и всички върхове както в лявото, така и в дясното поддърво имат $Y < Y_0$.

Декартовите дървета са предложени от Siedel и Aragon през 1989 г.

Предимства на такава организация на данните

При такава организация, множеството от стойности X играе ролята на **ключове** (и в същото време стойностите, съхранени в декартовото дърво), а множеството от стойности Y играе ролята на **приоритети**. Без приоритети, декартовото дърво ще бъде обикновено бинарно дърво за търсене по X и едно множество от X стойности може да съответства на много различни дървета, някои от тях дегенерират (например под формата на свързан списък) и следователно са изключително бавни (основните операции биха имали сложност $O(n)$).

В същото време **приоритетите** позволяват **еднозначно** да се определи дървото, което ще бъде конструирано (разбира се, това не зависи от реда на добавяне на стойности), което може да бъде доказано с помощта на съответната теорема. Очевидно е, че ако **изберем приоритетите на случаен принцип**, ще получим средно недегенерирани дървета, което ще осигури сложност от $O(\log n)$ за основните операции. Оттук и друго име на тази структура от данни - **рандомизирано двоично дърво за търсене**.

Операции

Декартовото дърво осигурява следните операции:

- $\mathbf{Insert}(\mathbf{X}, \mathbf{Y})$ за $O(\log n)$.

  Добавя нов връх към дървото. Един възможен вариант е да се предаде само X и да се генерира Y на случаен принцип в рамките на операцията (като същевременно се гарантира, че тя се различава от всички други приоритети в дървото).
- $\mathbf{Search}(\mathbf{X})$ за $O(\log n)$.

  Търси връх с посочената ключова стойност X. Имплементацията е същата като за обикновено бинарно дърво за търсене.

- **Erase**$(\mathbf{X})$ за $O(\log n)$.
  Търси връх с посочената ключова стойност X и го премахва от дървото.
- **Build**$(\mathbf{X_1}, \ldots, \mathbf{X_n})$ за $O(n)$.
  Построява дърво от списък със стойности. Това може да стане за линейно време (ако приемем, че $X_1, \ldots, X_n$ са сортирани), но няма да обсъждаме тази имплементация тук. Ние просто ще използваме n последователни операции по вмъкване, които имат сложност от $O(\log n)$ всяка (т.е. общо $O(n \log n)$ време за построяване).
- **Union**$(\mathbf{T_1}, \mathbf{T_2})$ за $O(m \log(n/m))$.
  Обединява две дървета, приемайки че всички елементи са различни. Възможно е да се постигне същата сложност, ако дублиращите се елементи трябва да бъдат премахнати по време на сливането.
- **Intersect**$(\mathbf{T_1}, \mathbf{T_2})$ за $O(m \log(n/m))$.
  Намира сечението на две дървета (т.е. техните общи елементи). Тук няма да разглеждаме имплементацията на тази операция.

В допълнение, поради факта, че декартовото дърво е двоично дърво за търсене, то може да реаличира други операции, като например намиране на k-тия най-голям елемент или намиране на индекса на елемент.

Описание на построяването

Що се отнася до имплементацията, всеки връх има следните атрибути: стойности X и Y, и поинтъри към лявото (L) и дясното (R) си дете.

Ще имплементираме всички необходими операции, като използваме само две спомагателни операции: $Split$ и $Merge$.

Split$(\mathbf{T}, \mathbf{X})$ разделя дървото T на две поддървета L и R (които са стойностите връщани от функцията), така че L съдържа всички елементи с ключ $X_L < X$, а R съдържа всички елементи с ключ $X_R > X$. Тази операция има $O(\log n)$ времева сложност и се изпълнява с помощта на очевидна рекурсия.

Merge$(\mathbf{T_1}, \mathbf{T_2})$ комбинира две поддървета T_1 и T_2 и връща новото дърво. Тази операция също има времева сложност $O(\log n)$. Тя работи базирайки се на допускането, че T_1 и T_2 са подредени (всики ключове X в T_1 са по-маки от ключовете в T_2). По този начин трябва да комбинираме тези дървета, без да нарушаваме реда на приоритетите Y. За да направим това, ние избираме като корен дървото, което има по висок приоритет Y в кореновия връх и рекурсивно извикваме $Merge$ за другото дърво и съответното поддърво на избрания коренов връх.

Сега имплементацията на $Insert(X, Y)$ става очевидна. Първо се спускаме в дървото (както в обикновеното бинарно дърво за търсене по X) и спираме на първия връх, в който приоритетната стойност е по малка от Y. Намерили сме мястото, където ще вмъкнем новия елемент. След това извикваме $Split(T, X)$ на поддървото, започвайки от намерения възел и използваме корените на върнатите поддървета L и R за ляво и дясно дете на новия връх.

Имплементацията на $Erase(X)$ също е ясна. Първо се спускаме в дървото (както при обикновеното двоично дърво за търсене по X), търсейки елемента, който искаме да изтрием. След като върхът бъде намерен, извикваме $Merge$ върху децата му и слагаме върнатата стойност на мястото на елемента, който искаме да изтрием.

Имплементираме $Build$ операцията за $O(n \log n)$ времева сложност използвайки n операции за вмъкване.

$\mathbf{Union(T_1, T_2)}$ има теоритична времева сложност от $O(m \log(n/m))$, но на практика работи много добре, вероятно с много малка скрита константа. Нека приемем, без ограничение на общността, че $T_1 \rightarrow Y > T_2 \rightarrow Y$, т.е. корена на T_1 ще бъде резултатния корен. За да получим реултата трябва да обединим дърветата $T_1 \rightarrow L$, $T_1 \rightarrow R$ и T_2 в две дървета, които могат да бъдат деца на корена T_1. За целта извикваме функцията $Split(T_2, T_1 \rightarrow X)$, като по този начин разделяме T_2 на две части L и R, които след това рекурсивно комбинираме с децата на T_1: $Union(T_1 \rightarrow R, R)$, като по този начин получаваме лявото и дясното поддърво на резултата.

Имплементация

```
struct item {
    int key, prior;
    item * l, * r;
    item() { }
    item (int key, int prior) : key(key), prior(prior), l(NULL), r(NULL)
{ }
};
typedef item * pitem;
```

Това е структурата на един връх в Декартовото дърво по дефиницията описана по-горе. Обърнете внимание, че има два указателя към ляво и дясно дете, целочислен ключ (за BST) и цялостен приоритет (за Binary Heap). Приоритетът се определя с помощта на генератор на случайни числа.

```
void split (pitem t, int key, pitem & l, pitem & r) {
    if (!t)
        l = r = NULL;
    else if (key < t->key)
        split (t->l, key, l, t->l),  r = t;
    else
        split (t->r, key, t->r, r),  l = t;
}
```

t е декартовото дърво за разделяне, а ключът (**key**) е стойността на BST, по която да се разделя. Имайте предвид, че никъде не връщаме стойностите на резултата, а просто ги използваме по следния начин:

```
pitem l = nullptr, r = nullptr;
split(t, 5, l, r);
if (l) cout << "Left subtree size: " << (l->size) << endl; if (r) cout <<
"Right subtree size: " << (r->size) << endl;
```

Функция за разделяне (**split**) може да бъде трудна за разбиране, тъй като има както указатели (**pitem**), така и референции към тези указатели (**pitem & l**). Нека разберем с думи какво прави извикването на функцията **split (t, k, l, r)**: „раздели декартовото дърво **t** спрямо стойността **k** на две декартови дървета и съхрани лявата част в **l** и дясната в **r**“. Чудесно! Нека сега приложим тази дефиниция към двете рекурсивни извиквания, като използваме работата по случая, която анализирахме в предишния раздел: (Първото условие if е тривиален основен случай за празно декартово дърво)

1. Когато стойността на кореновия връх е **не** по-голяма (≤) от **key**, извикваме **split (t-> r, key, t-> r, r)**, което означава: „раздели декартовото дърво **t->r** (дясното поддърво на **t**) по стойността, която има **key** и съхрани лявото поддърво в **t->r** и дясно поддърво в **r**". След това актуализираме **l = t**. Обърнете внимание, че резултатът **l** съдържа **t->l**, **t**, както и **t->r** (което е резултат от рекурсивното извикване, което направихме), всички вече обединени в правилния ред! Трябва да направите пауза, за да сте сигурни, че този резултат на **l** и **r** отговаря точно на това, което обсъдихме по-рано в описанието на построяването.
2. Когато стойността на кореновия възел е по-голяма (>) от **key**, извикваме **split (t-> l, key, l, t-> l)**, което означава: „раздели декартовото дърво **t->l** (ляво поддърво на **t**) по стойността, която има **key** и съхрани лявото поддърво в **l** и дясното поддърво в **t->l**". След това актуализираме **r = t**. Обърнете внимание, че резултатът **r** съдържа **t->l** (което е резултат от рекурсивното извикване, което направихме), **t**, както и **t->r**, всички вече обединени в правилния ред! Трябва да направите пауза, за да сте сигурни, че този резултат на **l** и **r** отговаря точно на това, което обсъдихме по-рано в описанието на построяването.

Ако все още имате проблеми с разбирането на построяването, трябва да го разгледате индуктивно, тоест: не се опитвайте да разбивате рекурсивните извиквания отново и отново. Да приемем, че функцията за разбиване (split) работи правилно на празен Treap, след това се опитайте да я стартирате за Treap с един връх, след това за Treap с два върха и т.н.

```
void insert (pitem & t, pitem it) {
    if (!t)
        t = it;
    else if (it->prior > t->prior)
        split (t, it->key, it->l, it->r),  t = it;
    else
        insert (it->key < t->key ? t->l : t->r, it);
}

void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r),  t = l;
    else
        merge (r->l, l, r->l),  t = r;
}

void erase (pitem & t, int key) {
    if (t->key == key) {
        pitem th = t;
        merge (t, t->l, t->r);
        delete th;
    }
    else
        erase (key < t->key ? t->l : t->r, key);
}

pitem unite (pitem l, pitem r) {
    if (!l || !r)  return l ? l : r;
    if (l->prior < r->prior)  swap (l, r);
    pitem lt, rt;
    split (r, l->key, lt, rt);
```

```
    l->l = unite (l->l, lt);
    l->r = unite (l->r, rt);
    return l;
}
```

Поддържане на размерите на поддърветата

За да се разшири функционалността на Treap, често е необходимо да се съхранява броят на върховете в поддървото на всеки връх - атрибут **int cnt** в структурата на **item**. Например, може да се използва за намиране на K-тия най-голям елемент в дървото за $O(\log N)$ или за намиране на индекса на елемента в сортирания списък със същата сложност. Изпълнението на тези операции ще бъде същото като за обикновеното двоично дърво за търсене.

Когато дърво се промени (върхове се добавят или премахват и т.н.), **cnt** на някои върхове трябва да се актуализира. Ще създадем две функции: **cnt ( )** ще връща текущата стойност на **cnt** или **0**, ако върхът не съществува, и **upd_cnt ( )** ще актуализира стойността на **cnt** за този връх, ако приемем, че за неговите деца **L** и **R** стойностите на **cnt** вече са актуализирани. Очевидно е достатъчно да добавите извикванията на **upd_cnt ( )** в края на **insert**, **erase**, **split** и **merge**, за да се поддържат актуални стойностите на **cnt**.

Построяване на Treap за $O(N)$ време в offline режим

Като се има предвид **сортиран списък с ключове**, е възможно да се построи Treap по-бързо, отколкото чрез вмъкване на ключовете един по един, което струва $O(N \log N)$ времева сложност. Тъй като ключовете са сортирани, балансирано двоично дърво за търсене може лесно да бъде конструирано за линейно време. Стойностите Y за Heap-а (приоритетите) се инициализират на случаен принцип и след това могат да бъдат пренаредени независимо от ключовете X за изграждане на Heap-а за $O(N)$.

```
void heapify (pitem t) {
     if (!t) return;
     pitem max = t;
     if (t->l != NULL && t->l->prior > max->prior)
          max = t->l;
     if (t->r != NULL && t->r->prior > max->prior)
          max = t->r;
     if (max != t) {
          swap (t->prior, max->prior);
          heapify (max);
     }
}

pitem build (int * a, int n) {
     // Construct a treap on values {a[0], a[1], ..., a[n - 1]}
     if (n == 0) return NULL;
     int mid = n / 2;
     pitem t = new item (a[mid], rand ());
     t->l = build (a, mid);
     t->r = build (a + mid + 1, n - mid - 1);
     heapify (t);
     upd_cnt(t);
```

```
    return t;
}
```

Забележка: извикването на **upd_cnt (t)** е необходимо само ако имате нужда от размерите на поддърветата.

Имплицитен Treap

Имплицитнят Treap е проста модификация на оникновения Treap, която е много мощна структура от данни. Всъщност, имплицитният Treap може да се разглежда като масив със следните процедури (всички в $O(\log N)$ време в online режим):

- Вмъкване на елемент в масива на всяко място
- Премахване на произволен елемент
- Намиране на сума, минимален / максимален елемент и т.н. на произволен интервал
- Добавяне на стойност, оцветяване на произволен интервал
- Обръщане на елементи на произволен интервал

Идеята е, че ключовете трябва да бъдат **индекси** на елементите в масива. Но няма да съхраняваме тези стойности изрично (в противен случай, например, вмъкването на елемент би предизвикало промени на ключа в $O(N)$ върха на дървото).

Обърнете внимание, че ключът на връх е броят на върховете по-малки от него (такива върхове могат да присъстват не само в лявото му поддърво, но и в левите поддървета на неговите предци). По-конкретно, неявният (**имплицитен**) ключ за някой връх T е броят на върховете $cnt(T \rightarrow L)$ в лявото поддърво на този връх плюс подобни стойности $cnt(P \rightarrow L)$ за всеки предшественик (прародител) P на върха T, ако T е в дясното поддърво на P.

Сега е ясно как бързо да се изчисли неявният ключ на текущия връх. Тъй като при всички операции стигаме до всеки връх, като слизаме в дървото, можем просто да натрупаме тази сума и да я предадем на функцията. Ако отидем в лявото поддърво, натрупаната сума не се променя, ако отидем в дясното поддърво, то се увеличава с $cnt(T \rightarrow L) + 1$.

Ето новите реализации на **Split** и **Merge**:

```
void merge (pitem & t, pitem l, pitem r) {
    if (!l || !r)
        t = l ? l : r;
    else if (l->prior > r->prior)
        merge (l->r, l->r, r), t = l;
    else
        merge (r->l, l, r->l), t = r;
    upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
    if (!t)
        return void( l = r = 0 );
    int cur_key = add + cnt(t->l); //implicit key
    if (key <= cur_key)
        split (t->l, l, t->l, key, add), r = t;
    else
        split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
    upd_cnt (t);
}
```

Сега нека разгледаме имплементацията на различни операции върху имплицитни Treap-ове:

- Вмъкване на елемент. **Insert element**.
  Да предположим, че трябва да вмъкнем елемент на позиция **pos**. Разделяме Treap-а на две части, които съответстват на масиви **[0..pos-1]** и **[pos..sz]**; за да направим това, извикваме **split** (T, T_1, T_2, pos). След това можем да комбинираме дърво T_1 с новия връх чрез извикване на **merge** (T_1, T_2, **new_item**) (лесно е да се види, че всички предпоставки са изпълнени). Накрая комбинираме дървета T_1 и T_2 обратно в T, като извикаме **merge** (T, T_1, T_2).
- Изтриване на елемент. **Delete element**.
  Тази операция е още по -лесна: намерете елемента за изтриване T, извършете сливане на неговите потомци L и R и заменете елемента T с резултата от сливането. Всъщност изтриването на елемент в неявния Treap е точно същото като в обикновения Treap.
- Намиране на сума / минимум и т.н. на интервал. **Find sum / minimum, etc. of interval**.
  Първо, създайте допълнителен атрибут F в структурата (item) на елементите репрезентиращи върхове в дървото, за да съхраните стойността на целевата функция за поддървото на този връх. Този атрибут е лесен за поддръжка подобно на поддържането на размерите на поддървета: създайте функция, която изчислява тази стойност за връх въз основа на стойностите за неговите деца и добавете извиквания на тази функция в края на всички функции, които променят дървото.
  Второто нещо, което трябва да знаем как да обработим заявка за произволен интервал $[A, B]$.
  За да получите част от дърво, която съответства на интервала $[A, B]$, трябва да извикаме **split** (T, T_1, T_2, A), а след това **split** $(T_2, T_2, T_3, B - A + 1)$: след това T_2 ще се състои от всички елементи в интервала $[A, B]$, и само от тях. Следователно отговорът на заявката ще се съхранява в полето F на корена на T_2. След като отговорът на заявката е генериран, дървото трябва да бъде възстановено чрез извикване **merge** (T, T_1, T_2) и **merge** (T, T, T_3).
- Добавяне на сума / оцветяване върху интервал. **Addition / painting on the interval**.
  Действваме подобно на предишната операция, но вместо атрибут F, ще съхраняваме атрибут **add** за добавяне, който ще съдържа добавената стойност за поддървото (или стойността, индикираща за типа оцветяване на поддървото). Преди да извършим каквато и да е операция, трябва да „разнесем“ (push) тази стойност правилно – т.е. да променим $T \to L \to add$ и $T \to R \to add$ и да изчистим **add** в родителския връх. По този начин след промени в дървото информацията няма да бъде загубена.
- Обръщане на интервал. **Reverse on the interval**.
  Това отново е подобно на предишната операция: трябва да добавим булев флаг „rev“ и да го зададем на true, когато поддървото на текущия връх трябва да бъде обърнато. „разнасянето“ (pushing) на тази стойност е малко сложно – ние разменяме деца на този връх и задаваме този флаг на true за тях.

Ето една примерна реализация на имплицитен Treap с обръщане на интервал. За всеки връх съхраняваме атрибут, наречен **value**, който е действителната стойност на елемента от масива в текущата позиция. Също така предоставяме имплементация на функцията **output ( )**, която извежда масив, който съответства на текущото състояние на имплицитния Treap.

```
typedef struct item * pitem;
struct item {
      int prior, value, cnt;
      bool rev;
      pitem l, r;
};
```

```
int cnt (pitem it) {
	return it ? it->cnt : 0;
}

void upd_cnt (pitem it) {
	if (it)
		it->cnt = cnt(it->l) + cnt(it->r) + 1;
}

void push (pitem it) {
	if (it && it->rev) {
		it->rev = false;
		swap (it->l, it->r);
		if (it->l) it->l->rev ^= true;
		if (it->r) it->r->rev ^= true;
	}
}

void merge (pitem & t, pitem l, pitem r) {
	push (l);
	push (r);
	if (!l || !r)
		t = l ? l : r;
	else if (l->prior > r->prior)
		merge (l->r, l->r, r), t = l;
	else
		merge (r->l, l, r->l), t = r;
	upd_cnt (t);
}

void split (pitem t, pitem & l, pitem & r, int key, int add = 0) {
	if (!t)
		return void( l = r = 0 );
	push (t);
	int cur_key = add + cnt(t->l);
	if (key <= cur_key)
		split (t->l, l, t->l, key, add), r = t;
	else
		split (t->r, t->r, r, key, add + 1 + cnt(t->l)), l = t;
	upd_cnt (t);
}

void reverse (pitem t, int l, int r) { pitem t1, t2, t3;
	split (t, t1, t2, l);
	split (t2, t2, t3, r-l+1);
	t2->rev ^= true;
	merge (t, t1, t2);
	merge (t, t, t3);
}

void output (pitem t) {
	if (!t) return;
	push (t);
	output (t->l);
	printf ("%d ", t->value); output (t->r);
}
```

Източници:

[1] **https://e-maxx.ru/algo/treap**

[2] **https://cp-algorithms.com/data_structures/treap.html**