Treap

Treap, binary search tree ve binary heap veri yapılarının birleştiği bir veri yapısıdır bu yüzden adı treap'tir (tree + heap).

Net bir tanımla; treap her node'unda (X,Y) ikililerini taşıyan X değerleri için binary search tree özelliği gösteren ve Y değerleri için de heap özelliğini gösteren bir veri yapısıdır. Aynı X değerine sahip iki node'un ve aynı Y değerine sahip iki node'un olmadığını kabul edebilirsiniz. Örneğin belirli bir node'un değerleri (X_0,Y_0) olsun, bu durumda sol alt ağaçta bulunan node'ların X değerleri için $(X<X_0)$ ve sağ alt ağaçta bulunan node'ların X değerleri için $(X>X_0)$ şartları sağlanmaktadır. Bunun yanında her iki alt ağacında bulunan node'ların Y değerleri için de $(Y<Y_0)$ şartı sağlanmalıdır.

Treap 1989 yılında Siedel ve Aragon tarafından keşfedilmiştir.

Treap veri yapısının avantajları

Treap yapısında X değerleri anahtar değerleri ve Y değerleri de öncelikleri temsil etmektedir. Bunun yanında her bir node içinde farklı değerler de saklanabilir. Eğer öncelikler olmasaydı veri yapısı standart binar search tree halini alacaktı. Ancak bazı binary search tree'ler kendi şartlarını sağlayacak şekilde oluşup ancak dengesiz halde olabilir(örneğin zincir). Böyle durumlarda ekleme ve sorgulama zamanı O(N) olabilmektedir.

Ancak eklenen elemanların öncelikleri eklenme sırası olmak zorunda değildir. Örneğin onlara birer Y değerleri verirsek ve bunları ekleme sırasından random değerler olarak seçersek ağacımızın rasgeleleşmesini sağlamış oluruz. Ve ortlama durumda ağacımızdaki arama ve ekleme işlemlerinin çalışma zamanın O(log N)'e inmesini sağlamış oluruz. Bu şekilde oluşan yapının bir diğer adı da randomized binary search tree'dir.

Treap ile yapılabilecek işlemler

Insert(X,Y) : Yaklaşık olarak O(log N) zamanda veri yapısında yeni bir elemanın eklenmesi işlemi gerçekleştirecektir. Buradaki Y değeri random bir şekilde oluşturulmalıdır. Ancak önceki eklenen elemanlardan farklı bir Y değerine sahip olan elemanın eklenmesi tavsiye edilmektedir(kodlama kolaylığı için).

Search(X) : Yaklaşık olarak O(log N) zamanda X anahtar değerli node'u bulmaya yarayan bir işlemdir ve tıpkı binary search tree'deki search işlemi gibi kodlanmaktadır.

Erase(X) : Yaklaşık olarak O(log N) zamanda X anahtar değerli elemanın ağaçtan çıkarılmasını sağlayan bir işlemdir.

Build(X_1….X_N) : O(N) zamanda tree X listesindeki anahtar değerlerden oluşturulur ancak X dizisinin sıralı olduğu kabul edilmektedir. Öte yandan bütün elemanların tek tek insert işlemiyle eklenmesi durumunda dahi çalışma zamanı O(N log N) olacaktır.

Union(T_1,T_2) : Yaklaşık olarak O(N log (N/M)) zamanda verilen iki treap yapısının birleştirilmesini sağlar. Burada basit olarak treap'ler içerisindeki elemanların tamamının birbirinden farklı olduğunu kabul ediyoruz ancak aynı elemanlar olduğu durumda da tekrar edenleri silecek şekilde aynı zaman karmaşıklığıyla çalışan bir işlem de geliştirilebilir.

Intersect(T_1,T_2) : Yaklaşık olarak O(N log (N/M)) zamanda verilen iki treap yapısındaki ortak elemanları bulan işlemdir. Burada bu işlemin kodlanmasına değinmeyeceğiz. Ancak ufuk açması açısından verilmesini faydalı gördük.

Bunların yanında arama işlemleri daha karmaşık hale de getirilebilir örneğin K. en küçük elemanı bulmak vs gibi.

Kodlama için açıklama

Yapılacak işlemleri gerçekleştirmek için öncesinde kodlamamız gereken iki adet işlem vardır: Split ve Merge. Bu işlemler sırası ile, verilen iki treap yapısını birleştirmektedir ancak bu merge işleminde verilecek olan iki treap yapısının anahtar değerlerin sol taraftakinde daima sağ taraftakinden büyük olması beklenmektedir.

Split(T,X) : Treap yapısını verilen anahtar değerinden(X) küçük değerliler ve büyük değerliler şeklinde iki parçaya ayırmaktadır. Yaklaşık olarak O(log N) zamanda çalışmaktadır.

Merge(T_1,T_2) : Verilen iki alt ağacı birleştirmektedir ve yaklaşık olarak O(log N) zamanda çalışmaktadır. Burada önemli olan ayrıntı şudur T_1 altında bulunan bütün anahtar değerleri T_2 altında bulunan bütün anahtar değerlerinden küçük olmalıdır. Bu işlemi gerçekleştirirken Y değeri daha büyük olan root'u seçip recursive olarak yeni alt ağaçlar seçilip birleştirilecektir.

Insert(X,Y) : Öncelikle binary search tree'deki arama işlemi gibi X anahtar değerine göre arama işlemi gerçekleştirip alt node'lara inmeliyiz ancak öncelik değeri Y den küçük olan ilk yerde durmalıyız. Şimdi ekleyeceğimiz elemanın nerede duracağını bulmuş olduk. Ağacı buradan iki parçaya ayırmalıyız Split işlemi ile. Ardından eklememiz gereken elemanı bu alt ağaçlardan ekleyeceğimiz yere ekleme işlemlerini gerçekleştirip tekrar bu iki alt ağacı Merge işlemi ile birleştireceğiz.

Erase(X) : İstenilen X elemanını tıpkı binary search tree'deki arama işlemi gibi tree üzerinde bulacağız ve treap yapımızı buradan Split işlemi ile iki parçaya ayıracağız. Ayırdığımız bu yapılar içerisinden X elemanını silmek çok kolaydır artık. Silme işlemi gerçekleştirdikten sonra Merge işlemi ile ayrılan iki ağacı birleştireceğiz.

Build işlemini N defa Insert işlemini çağırarak yapabiliriz.

Union(T_1,T_2) : Birleştirme işlemi zaman karmaşıklığı olarak yaklaşık O(N log(N/M)) olmasına rağmen bu işlem gayet hızlı bir şekilde çalışmaktadır. İşlem kabaca şöyle çalışmaktadır. Öncelikle birleştirmemiz gereken her iki ağacın da root'larının öncelik değerlerinden maksimum olanı bulmalıyız bu yeni oluşacak olan ağacın root'unu oluşturacaktır bu ağacın T1 olduğunu varsayalım. Daha sonra diğer T2 ağacını T1 ağacının root'unun anahtar değerinden itibaren iki parçaya ayıracağız bu ayırdığımız parçalar sırasıyla L ve R olsun. Şimdi yapmamız gereken T1 ağacının sol çocuğu ile L ağacını ve T1 ağacının sağ çocuğu ile R ağacını birleştirmeliyiz bu birleşimlerden oluşan ağaçlar da sırasıyla bize T1 ağacının yeni sol alt ağacını ve T1 ağacının yeni sağ alt ağacını verecektir.

```
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;

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;
}

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)
		merge (t, t-> l, t-> r);
	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;
}
```

Bunların yanı sıra alt ağacın büyüklüğünü de tutabiliriz ve bu değeri tutarak ağaçta bulunan K. en küçük sayıyı bulabiliriz bunun için ekstradan iki kısa fonksiyon yeterli olacaktır.

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

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

Implicit Treap

Implicit treap çok güçlü bir veri yapısıdır. Temel olarak N elemanlı bir dizinin var olduğunu düşünelim elimizde bu dizi üzerinde yapılacak olan bir çok işlemin online olarak yapıldığı sırada zaman karmaşıklığını yaklaşık olarak O(logN)'e düşürmektedir. Örneğin:

- Bir pozisyona eleman eklemek.
- Bir elemanı diziden silmek.
- Rastgele bir aralığın maksimumunu/minimumunu/toplamını/EKOK'unu/EBOB'unu .. bulmak.
- Belli bir aralığı artırmak ya da belli bir sayıyla eşitlemek.
- Belli bir aralığı çıkarmak, ters çevirmek, dairesel şekilde döndürmek…

Bu veri yapısının altında yatan anahtar değer olarak dizideki indislerini tutmasıdır ancak bu değerleri hiç bir zaman ağaç içinde saklamayız sadece var olduklarını hayal ederek işlemlerimizi yaparız.

Bu anahtar değerlerin nasıl olduğunu düşünecek olursak root node için anahtar değer sol çocukta bulunan node'ların sayısıdır. Aynı şekilde sağ çocukta bulunan node'ların anahtar değerleri de sol çocuğun altında bulunan node'ların sayısı + 1 den başlamaktadır(indislerin 0'dan itibaren sıralandığını düşünmekteyiz).

Şimdi yeni Split ve Merge işlemlerinin nasıl kodlanacağını görelim.

```
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); // Calculate the 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);
}
```

Şimdi diğer işlemlerin nasıl kodlanacağını biraz açalım:

Insert: Öncelikle v değerli eleman p pozisyonuna eklenecek olsun bu durumda diziyi (0,p-1) ve (p,n-1) şeklinde iki parçaya ayırdığımızı düşünelim yani ağacı split ile iki parçaya ayıracağız ardından ilk parça ile yeni gelen elemanı birleştirip ardından da 2. parçayı bu birleşip oluşmuş olan grupla birleştirerek. Ağaca bu yeni elemanı eklemiş oluruz.

Remove: Bir elemanı silme işlemi yine ekleme işlemine benzerdir. Split işlemleri ile silinecek elemanı tek bir node haline gelecek şekilde ağacı parçalayalım. Ardından o node hariç diğer ağaçların birleşmesiyle elimize o node'un bulunmadığı ağaç gelmiş olacaktır.

Sum/Max/Min : Bu işlem için ekstradan bir değişken daha tutacağız her node'da bunu daha önce yaptığımız ağacın büyüklüğünü bulma işlemiyle neredeyse aynı şekilde hesaplayacağız. Bir node için altında bulunan sağ ve sol çocukların aranan değerlerine bakarak bunların max/min/sum değerlerini bulunan node'a aktararak ilerleyeceğiz. Bu değerlerin hesaplanmasından sonra ağaçta yine split

işlemini uygulayarak aradığımız aralığı tek bir ağaç haline getireceğiz ve bu ağacın root'unda bulunan bu hesapladığımız değer alt ağacın max/min/sum değerini bize verecektir.

Ekleme/Boyama : Yine üstte ekstradan sakladığımız değer gibi bir değer saklayacağız ve bu değer alt ağacın ne kadar artırıldığını yahut hangi sayıyla eşitlendiğini bize gösterecektir. Ve bu değerleri gerekli olduğu zaman daha önce segment tree dokümanında da bahsettiğimiz "push" fonksiyonuyla alt ağaçlara aktaracağız. Ayrıntılı bilgi için kodu incelemek faydalı olacaktır.

Reverse : Bu işlem bize bir aralığın ters çevrilmesinde yardımcı olacaktır. Yine önceki işlemlerde yaptığımız gibi ekstradan bir değer daha saklayacağız ve bu değer alt ağacın ters çevrilip çevrilmediğini tutacak. Yine gerekli olduğu durumda "push" işlemi ile bu değeri alt ağaçlara ileteceğiz ve bulunulan node için gerekli işlemleri gerçekleştireceğiz. Diğer bir deyişle bulunulan node'un sağ ve sol çocuklarını swap yapıp aynı zamanda onların da kendi içinde ters çevrileceğini belirtiyoruz. Bu şekilde bütün bir aralığın ters çevrilmesini sağlıyoruz.

Şimdi kodlanmasını inceleyelim:

```
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);
}
```