Euler Tour Tree

Euler Tour Tree е структура от данни, която ни позволя да пазим информация за гора от дървета. Подобно на Link-Cut Tree то може да поддържа операции като откачане на връх от родителя му и закачане на връх към някой друг. Разликата между двете дървета е, че Link-Cut Tree е по-добро за поддържане на информация за пътища, а Euler Tour Tree – за поддървета (повече информация в уикипедия).

Структурата от данни се базира на Ойлерово Обхождане на Дърво и идеята е, че може то може да се коригира динамично спрямо промените като се пази в някаква структура от данни. В тази имплементация ще използваме Imlicit Treap, но има и други варианти.

Целта ни ще бъде да направим структура от данни, която може да добавя единичен връх без родители, да закача и откача върхове и да може да изолира някое поддърво. Причината да се използва ойлерово обхождане е, че в него едно поддърво става интервал, който е ограничен от две появявания на един връх (коренът на поддървото). Единственото нещо, което трябва да поддържаме през цялото време е, че трябва да имаме правилно ойлерово обхождане за всяко дърво от гората.

За момент без да се фокусираме върху детайлите по имплементацията ще обясним как ще се извършват тези операции.

[Добавяне на връх]

За тази цел трябва да си представим, че имаме дърво, което е само връх u. Неговото ойлерово обхождане ще бъде просто [u, u].

[Откачане на връх]

Тук трябва да знаем кой е коренът на дървото, в което се намира нашият връх и така да намерим интервалът, който отговаря на неговото поддърво. По-конкретно – в ойлеровото обхождане трябва да намерим интервалът [u, …, u] и да го премахнем от там. По този начин ще запазим ойлеровото обхождане на старото дърво, а за новото отделило се такова ще имаме готово обхождане, което просто трябва да запазим някъде. Също за удобство може да отбележим, че вече върхът u няма родител.

[Закачане на връх]

Ако искаме да закачим връх u към връх v, трябва да намерим къде се намира поддървото на v. То ще бъде от вида [v, …, v]. За да закачим дървото с корен u, би било валидно просто да сложим неговото ойлерово обхождане точно след първото срещане на v. Тоест поддървото на v ще стане [v, u, …, u, …, v].

[Изолиране на поддърво]

Тъй като вече знаем, че всяко поддърво е просто интервал, който започва и свършва с един и същи връх то за нас е абсолютно достатъчно да знаем кой връх от кое дърво е част и какви са индексите на първото и последното негово срещане. Тъй като всички операции до сега включват само местене на части от редици то ние съвсем спокойно можем да поддържаме тази информация. Това ще стане много лесно по-нататък, когато представяме нещата като Treap-ове.

Имайки тази основна информация можем да си изведем и няколко други малки функции, които ще бъдат полезни по нататък.

[Намиране на корен на дърво]

Ако искаме да знаем кой е коренът на дървото, в което се намира връх u, трябва да знаем в коя редица е намира в момента неговото ойлерово обхождане и след това да вземем първия елемент на тази редица, тъй като той винаги ще се явява корен.

[Намиране на информация за поддърво]

Тук има много начини това да се направи. Единият е да пазим информацията за редиците, която да ни казва колко са реалните върхове. Точно това е и силата на Euler Tour Tree, понеже можем да пазим всякаква такава информация, за която е нужно просто да се измисли функция, която бързо слива две редици. Пример за това е сума на върхове, брой на върхове, произведение на върхове, брой на върхове с четни номера и т.н.

[Намиране на LCA]

Има един метод за намиране на LCA, който използва Oйлеровото обхождане и който гласи, че LCA-то на два върха е върхът от интервалът между двете им първи появявания, който има най-нисък level, тоест се намира най-високо в дървото. Тук виждаме, че на теория това е изпълнимо, понеже можем да поддържаме информация за дадени интервали и тази за най-високият връх е такава, която лесно се merge-ва. Също така информацията за нивата можем да поддържаме като всеки път ъпдейтваме нивата при промяна на дървото, което при Treap може да се направи с lazy propagation (при всяка промяна ще сметнем разликата в нивата на всички върхове от поддървото и ще я добавим като промяна по стандартният начин с lazy propagation). Проблемът при това нещо е, че Treap-ът може да стане потенциално , понеже, за да видим дълбочината на някой връх трябва всеки път да пушваме лейзито до него и така всяко recalc-ване ще стане твърде тромаво. Дори и след някакви оптимизации пак би било твърде тромаво и досадно за писане, но все пак трябва да се отбележи, че го имаме като вариант. Друг важен извод от това е, че въпреки тромавостта, която идва от работата с двоични дървета, все пак тази техника се доказва като доста „стабилна“, тоест че може да поддържа странни функции на сравнително приемлива цена.

**Имплементация**

Преди да започнем с каквато и техническа конкретика, трябва да отбележим, че чисто като код цялото нещо е сравнително голямо (около 200+ реда код само за цялата структура от данни в доста чист вид). Това означава, че не се очаква да се използва на състезания тип спринт заради времето, което би било изгубено в имплементиране. Това не пречи да се е случвало на българските състезания, разбира се (2019 контролно 2 за А група – задача restructuring, Ямбол). Но все пак такъв тип нещо би се очаквало да се даде на състезание, по-близо до маратон (например квалификацията за московската олимпиада).

Като за начало трябва да имплементираме Implicit Treap, под някаква форма, който е желателно да съдържа в себе си освен стандартните ляво, дясно дете и приоритет, неща като – конкретен връх от редицата и също така големина на поддървото, ако се намираме в първия връх от двата при някое поддърво. Това ни принуждава и да пазим дали сме в първи връх или не. Под първи връх се разбира следното: както знаем всяко поддърво с корен **u** се представя като сегмент **[u, …, u]**, та първото **u** наричаме първи връх. В тази имплементация ще напишем само структура за връх на Treap, а не за цялата структура, понеже това само ще утежни работата ни и няма да помогне особено много.

Също така ще използваме само raw pointers (обикновените pointer-и в С++), понеже ако се използват smart pointers нещата ще се усложнят и плюс това най-често за състезателните задачи няма нужда от толкова стриктен memory management, което в крайна сметка е и основната идея на smart pointers.

    struct TreapNode

    {

        //treap info

        int length;

        TreapNode \*L, \*R, \*parent;

        int priority;

        //abstract info

        int node, treeSize;

        bool isFirst;

За да можем все пак да поддържаме тази информация актуална за всеки връх на Treap-а е много важно да напишем хубава функция, която всеки път да пресмята правилно информацията наново. Примерна имплементация би била следната.

        void recalc()

        {

            length = 1;

            treeSize = isFirst;

            if(L!=nullptr)

            {

                length += L->length;

                treeSize += L->treeSize;

                L->parent = this;

            }

            if(R!=nullptr)

            {

                length += R->length;

                treeSize += R->treeSize;

                R->parent = this;

            }

        }

Както се вижда тук наново изчисляваме големината на Treap-а и също така поддържаме променливата treeSize като я пресмятаме наново чрез децата на конкретния връх, а както и чрез информацията в isFirst променливата. Също така и винаги сетваме родителите на децата на конкретния връх да бъдат правилни. Може би било по-правилно това да се отдели във функция за lazy propagation, но тъй като няма да правим нищо лейзи, оставяме нещата така. Като цяло функцията спазва основната философия на трийпа и като цяло всички подобни дървета – стойността на връх е винаги функция от неговата информация и тази на децата му.

Освен това ще ни трябват и още три функции, които ще ни позволят да извършим основните операции в Treap-a. Ще ни трябва една, която намира корена на някой връх, една, която намира най-левия връх в дърво и такава, която пресмята индекса на връх.

        int getInd(bool toAdd = true)

        {

            recalc();

            if(toAdd==true)

            {

                if(parent==nullptr) return ((L==nullptr)?0:L->length);

                return 1 + ((L==nullptr)?0:L->length) + parent->getInd((parent->R==this));

            }

            else

            {

                if(parent==nullptr) return -1;

                return parent->getInd((parent->R==this));

            }

        }

        TreapNode \*getRoot()

        {

            if(parent==nullptr) return this;

            return parent->getRoot();

        }

        TreapNode \*getLeftmost()

        {

            if(L==nullptr) return this;

            return L->getLeftmost();

        }

При конкретната имплементация на функцията getInd е извършено качване нагоре по дървото и всеки път ние знаем дали сме дошли като ляво дете и спрямо тази информация добавяме лявото поддърво като „предишни“ върхове. Когато сме в корена на дървото връщаме -1, понеже нашата функция винаги ще преброи с едно повече, тъй като тя не прави разлика между това дали е в началния връх или в някой междинен. Като цяло основната идея е, че индексът по дефиниция е броя на нещата, които са преди нас и така ние като се качваме нагоре се опитваме да броим винаги върховете, които са преди нас, тоест са в лявото поддърво. Досадната част идва от това, че трябва да се следи от какъв връх идваме и също така да внимаваме да не преброим нещо в повече. Всеки път пускаме recalc(), което може да се счете за излишно в слуая, но е хубаво все пак да се спазва философията, че винаги трябва да се пресмятат върховете наново, за да сме сигурни, че всичко е наред. Естествено тази реализация не е единствена или перфектна и може да се измисли нова.

Също за Treap-а ще ни трябват и стандартните функции за сливане и за разделяне (в нашия случай по-размер). За радост не ни трябва да пишем делене на Treap по големина на елементите. Тези две функции общо взето спазват основната идея за Treap като вземат предвид и факта, че винаги пазим родителите.

    TreapNode\* Merge(TreapNode \*small, TreapNode \*big)

    {

        if(small==nullptr) return big;

        if(big==nullptr) return small;

        if(small->priority > big->priority)

        {

            small->R = Merge(small->R, big);

            small->recalc();

            return small;

        }

        else

        {

            big->L = Merge(small, big->L);

            big->recalc();

            return big;

        }

    }

    pair <TreapNode\*, TreapNode\*> SplitSz(TreapNode \*T, int sz)

    {

        if(T==nullptr) return {nullptr, nullptr};

        if(((T->L==nullptr)?0:T->L->length)+1<=sz)

        {

            auto splitRes = SplitSz(T->R, sz-((T->L==nullptr)?0:T->L->length)-1);

            T->R = splitRes.first;

            T->recalc();

            if(splitRes.second!=nullptr)

                splitRes.second->parent = nullptr;

            return {T, splitRes.second};

        }

        else

        {

            auto splitRes = SplitSz(T->L, sz);

            T->L = splitRes.second;

            T->recalc();

            if(splitRes.first!=nullptr)

                splitRes.first->parent = nullptr;

            return {splitRes.first, T};

        }

    }

След като вече имаме работещ Treap можем да започнем да описваме основните функции на Euler Tour Tree. Тук отново ще използваме доста обектно ориентиран подход и ще имплементираме дървото като отделна структура. Това е много удобно, понеже така можем да имаме много Euler Tour Tree-та, които са активни по едно и също време. За да сме честни обаче, точно тази имплементация няма да бъде много подходяща за това, понеже ще използва големи статични масиви предназначени за много елементи и също така ще искаме доста начална информация за това колко може да бъде голямо дървото. Всички тези неща правят работата после леко неудобна, понеже ще трябва да знаем доста неща предварително, но за сметка на това печелим някаква минимална производителност в повече (масивите са stack allocated и освен това няма нужда да if-им всеки път за върховете къде и дали съществуват). Истинската причина имплементацията да остане така е, за да бъде максимално изчистена от досадни детайли свързани с реалността.

За нашето Euler Tour Tree ще пазим за всеки връх неговият пряк баща, а също така и pointer-и към неговите появявания в euler tour-овете. Примерна имплементация и инициализация би била следната:

    int n;

    int parent[MAXN];

    pair <TreapNode\*, TreapNode\*> ptr[MAXN];

    EulerTourTree(){}

    EulerTourTree(int n)

    {

        this->n = n;

        for(int x = 0;x<=n;x++)

        {

            this->parent[x] = -1;

            this->ptr[x].first = new TreapNode(x, true);

            this->ptr[x].second = new TreapNode(x, false);

            Merge(this->ptr[x].first, this->ptr[x].second);

        }

    }

Като цяло, знаейки колко върха ще ни трябват, правим за всеки от тях по два малки трийпа с правилната информация и след това ги сливаме, понеже реално ойлеровото обхождане на самотен връх **u** е просто **[u, u]**. Също така и инициализираме родителите на някоя стойност, която сме приели, че ще означава, че върха няма родител, в нашия случай - **-1**.

**Основни функции на Euler Tour Tree**

[Разкачване на връх]

За да можем успешно да разкачим един връх, първо трябва да намерим кой е корена на Treap-а, в който се намира неговото ойлерово обхождане. Това става, чрез вече готова функция. След това намираме границите на интервала на нашия връх, чрез две извикване на функцията **getInd().** След това разрязваме трийпа два пъти, за да можем да отделим точно тази част, която отговаря за обхождането на поддървото, чийто корен е нашия връх. След това сливаме остатъците от Treap-a, които нямат общо с нашия връх. На практика се получава нещо такова:

[нещо си 1, x, …., x, нещо си 2] -> [нещо си 1, нещо си 2].

Примерна имплементация за тази функция е следната.

    void disconnectNode(int x)

    {

        if(parent[x]==-1) return;

        parent[x] = -1;

        TreapNode \*T = ptr[x].first->getRoot();

        int l = ptr[x].first->getInd();

        int r =  ptr[x].second->getInd();

        auto help1 = SplitSz(T, r+1);

        auto help2 = SplitSz(help1.first, l);

        T = Merge(help2.first, help1.second);

    }

[Закачване на връх]

Тук отново намираме кой е коренът на Treap-a, в който се намира Ойлеровото обхождане на новия баща на нашия връх. След това разрязваме този трийп, точно през първото срещане на **par**, така че едното парче да бъде завърша с **par**, а в другото да стои цялата негова остатъчна структура. При сливането между тези две парчета вкарваме нашето Ойлерово обхождане и така получаваме съвсем валидно обхождане на новото дърво. Със самите редици се получава нещо такова

[нещо си 1, par, нещо си 2, par, нещо си 3] -> [нещо си 1, par] [нещо си 2, par, нещо си 3] -> [нещо си 1, par, u, …, u, нещо си 2, par, нещо си 3].

Примерна имплементация би била следната:

    void connectNode(int u, int par)

    {

        TreapNode \*T = ptr[par].first->getRoot();

        parent[u] = par;

        auto help = SplitSz(T, ptr[par].first->getInd()+1);

        T = Merge(help.first, ptr[u].first->getRoot());

        T = Merge(T, help.second);

    }