Резюме

Структурите от данни се използват за ефективно съхраняване на информация в компютъра. Разработката разглежда недостатъчно познати, но ефективни структури от данни като Treap и Implicit Treap, основни операции с тях и използването им при решаване на алгоритмични задачи. Сравнително лесната имплементация на тези структури от данни и бързото изпълнение на операциите с тях, ги превръщат в ефективен инструмент при решаването на определен вид задачи в състезателното програмиране. В текущата разработка авторите предлагат реализация на горе посочените структури от данни и операции с тях на езика C++.

Summary

Data structures are used for storage of information in the computer. This projects looks at relatively unpopular, but efficient structures such as Treap and Implicit Treap, their properties and applications in solving algorithmic problems. The relatively easy implementation and efficient operations make them an effective tool in competitive programming. In this paper the authors have given an implementation of the above mentioned data structures in C++.

I. Увод

За ефективно решаване на алгоритмични задачи от състезанията по информатика понякога е уместно използването на различни видове структури от данни. Проектът разглежда структурите Treap и Implicit Treap и техните приложения в състезателното програмиране. Treap и Implicit Treap са рандомизирани структури от данни, които могат да се използват за запазване на информация в удобен за обработка вид и решаване на редица проблеми.

Проектът е разделен на три части. В първата е разгледана структурата Treap и някои основни операции с тази структура. Във втората част от документацията е разгледана структурата Implicit Treap и някои стратегии при изплозването и. В третата част са разгледани задачи от състезания, изискващи използването на тези структури от данни и решенията на тези проблеми и имплементацията им на езика C++.

Допълнително са представени сорс кодове на всички основни операции с Treap и Implicit Treap, както и решени задачи, реализирани на езика C++.

II. Treap

1. Определение: Дървовидна структура от данни, всеки връх от която се характеризира от две величини – ключ(стойност) и приоритет. *Трипът* владее свойствата на BST(дърво за двоично търсене) и Heap(пирамида). Друго наименование на *трип* е Cartesian Tree(декартово дърво).

Нека означим ключа на всеки връх с x, а приоритета с y. За декартовото дърво е изпълнено следното:

- Дървото, образувано само от ключовете на всеки връх е BST.
- Дървото, образувано от приоритетите на всеки връх е Heap (пирамида).

Теорема: Давайки на y случайна стойност за всеки връх, то декартовото дърво ще има очаквана височина $\log_2 N$.

Доказателство:

Означаваме с $E[...]$ очакваната стойност на някаква променлива.

Нека A_{ij} е променлива, която показва *i* дали е родител на *j*, където *i* и *j* са позициите на ключовете на 2 върха. *i* е родител на *j*, само ако има най-висок приоритет от всички върхове *i* между и *j*, защото ако вземем връх *k* между *i* и *j* с приоритет по-висок от приоритета на *i* и *j*, то *k* ще е родител на *i* и *j* => единствения начин *i* да е родител на *j* е да има най-висок приоритет измежду всички върхове между *i* и *j*.

Тъй като всеки връх между и има еднаква вероятност да се окаже с най-висок приоритет => вероятността да *i* е родител на *j* е $\frac{1}{|j-i|+1}$ => $E[A_{ij}] = \frac{1}{|j-i|+1}$.

Намираме очакваната височина D_i на декартово дърво с корен връх *i*.

$$E[D_i] = \sum_{j=1\ j\neq i}^{n} E[A_{ij}] = \sum_{j=1\ j\neq i}^{n} \frac{1}{|j-i|+1} = \sum_{j=1}^{i-1} \frac{1}{i-j+1} + \sum_{j=i+1}^{n} \frac{1}{j-i+1}$$

$$= \sum_{p=1}^{i} \frac{1}{p} - 1 + \sum_{p=1}^{n-i+1} \frac{1}{p} - 1 = H_i - 1 + H_{n-i+1} - 1$$

Тъй като $H_N \in (\ln(N)\,;\ln(N)+1) => E[D_i] < 2\ln(n) => E[D_i] = O(\log_2(n)) =>$ Очакваната височина на декартово дърво е $\log_2 N$.

2. Сливане на 2 декартови дървета: Тази операция ще бъде означена като Merge. Изискванията за осъществяване на тази операция са ключовете(стойностите) на всички върхове от едното да са по-малки или равни на ключовете(стойностите) на всички върхове от второто.

Нека означим двете дървета с *L*(по-малки ключове) и *R*(по-големи ключове) и полученото след извършването на Merge дърво с *T*.

Построяването на корена на *T* ще стане много лесно – взимаме или корена на *L*, или корена на *R*, в зависимост от това, кой от тях има по-висок приоритет.

Нека приемем, че коренът на *L* има по-висок приоритет и стойността му е *x*. Лесно се забелязва, че лявото поддърво на *L* ще остане ляво поддърво на *T* - всички елементи от него са с по-малък ключ и с по-нисък приоритет и не съществуват елементи от *R*, които да са с по-нисък ключ. Следователно лявото поддърво на *L* ще е ляво поддърво на *T*.

Остава да построим само дясното поддърво на *T* от $L.Right$ и *R*, запазвайки свойствата на *treap*. Това се свежда до същата задача – имаме две декартови дървета – всички ключове на едното са по-малки от всички

ключове на другото. Следователно ще получим крайния резултат T изпълнявайки рекурсивно функцията Merge, докато лявото и дясното дърво не останат празни.

Симетричен е и случаят, когато коренът на R има по-висок приоритет – дясното поддърво остава дясно поддърво и прилагаме рекурсивно Merge за L и $R.Left$ – ключовете на всички елементи от $R.Left$ са по-големи от тези на L.

3. Разделяне на декартово дърво: Тази операция ще бъде означена като Split. Разделяме дадено декартово дърво T на 2 поддървета L и R, където в L са всички елементи с по-малка(или равна) стойност от дадено x_0, а в R – всички с по-голяма от x_0 стойност. Числото x_0 е предварително зададено – то е известно.

Първо проверяваме дали коренът на T е с по-голяма или по-малка от x_0 стойност(ключ) и съответно дали ще се намира в L или R. Нека предположим, че коренът е с по-малка стойност от x_0.

Следователно, всички елементи от лявото поддърво на T ще са в L (всички елементи от лявото поддърво на T са с по-малка стойност от корена на T, който е спо-малка стойност от x_0), а коренът на T ще бъде и корен на L => лявото поддърво на T ще бъде ляво поддърво на L. Остава само да добавим елементите от дясното поддърво на T със стойности по-малки от x_0 към дясното поддърво на L, а всички останали елементи ще формират R.

Стигаме до идентична задача - трябва да отделим всички елементи с ключове(стойности) по-малки от x_0 от дясното поддърво на T и да ги добавим към дървото L. Рекурсивно прилагаме операцията Split за дясното поддърво на T.

Ако коренът на текущото поддърво е по-голям от x_0, имаме симетричен случай – всички елементи от дясното поддърво стават част от дясното поддърво на R и прилагаме рекурсивно Split за лявото поддърво.

Време за работа на Merge и Split: Всяка от операциите прави максимално $2H$(височина на декартовото дърво) операции => работят със сложност $O(H)$, но $H \to 4\log_2 N$ => операциите ще работят със сложност $O(\log_2 N)$.

4. Основни операции върху декартови дървета:

- Добавяне на елемент със стойност(ключ) x:

1. Прилагаме Split върху текущото поддърво с $x_0 = x$. По този начин отделяме всички елементи с по-малки или равни на x ключове в L и всички с по-големи ключове в R.
2. Построяваме декартово дърво M, състоящо се само от един връх $(x; y)$, където y е случайно генериран приоритет.
3. Прилагаме Merge върху L и M ,след това отново прилагаме върху полученото дърво и R.

- Премахване(изтриване) на всички елементи от декартово дърво със стойност(ключ) x:
  1. Прилагаме Split върху даденото дърво при $x_0 = x - 1$, получавайки две поддървета – едното от елементи със стойности по-малки от x, а другото с елементи със стойности по-големи или равни на x.
  2. Прилагаме Split върху полученото дясно поддърво при $x_0 = x$, получавайки две дървета – едното само от елементи със стойност x, а другото от елементи със стойност по-голяма от x.
  3. Прилагаме Merge върху дървото със стойности, по-малки от x, и върху дървото със стойности, по-големи от x.
- И двете операции работят със сложност $O(\log_2 N)$, защото прилагаме Merge 1 път, а Split – 2 пъти.

5. Построяване на декартово дърво: Построяваме декартово дърво с корен първия елемент**.** За всеки следващ елемент прилагаме операцията добавяне. Този алгоритъм работи със сложност $O(N\log_2 N)$.

Ако добавяме ключовете в нарастващ ред, сложността може да се сведе до $O(N)$. Ще пазим мястото, където е добавен последният връх. Нека приемем, че текущият елемент за добавяне има стойност x и приоритет y.

Последно добавеният връх е най-десен в дървото(има най-голяма стойност). Ако приоритетът му е по-голям от приоритета на текущия връх, добавяме го като десен наследник на последно добавения връх. В противен случай тръгваме нагоре по дървото, докато не стигнем до връх с по голям приоритет $y_0 > y$ или до корена на дървото.

Ако стигнем до корена, добавяме текущия връх като нов корен и правим корена ляв наследник.

Ако не сме стигнали до корена, а до връх $(x_0; y_0) - y_0 > y$, запазваме текущия връх като дясно поддърво на $(x_0; y_0)$, а дясното поддърво на $(x_0; y_0)$ ще бъде ляво поддърво на текущия връх (всички стойности ще са по-малки от x).

Този алгоритъм прави максимално $2N$ операции => работи със сложност $O(N)$.

6. Търсене на елемент в декартово дърво: Във всеки връх запазваме размера на поддървото с корен този връх като допълнителен параметър - S.

За да намерим K–ия по големина елемент, проверяваме размера на поддървото с корен левия наследник на всеки връх - S_L:

1. Ако $S_L = K$, то сме намерили търсения елемент и това е текущо обхождания връх.
2. Ако $S_L > K$, то търсеният елемент се намира някъде в лявото поддърво. Спускаме се към левия наследник на текущия връх и повтаряме процеса.
3. Ако $S_L < K$, то търсеният връх се намира някъде в дясното поддърво на текущия връх. $K = K - S_L - 1$ и се спускаме надясно, рекурсивно повтаряйки процеса.

Този алгоритъм работи със сложност $O(\log_2 N)$, защото се извършват не повече от H операции.

Проблем възниква при добавянето или премахването на елемент от декартовото дърво, защото тогава трябва размерът на всяко поддърво да бъде преизчислен наново – със сложност $O(N)$.

Този проблем може да бъде решен с модифициране на Merge и Split за преизчисляване на големините на всяко поддърво, защото всички операции са базирани на тях.

При Merge трябва да се слеят две декартови дървета с някакви големини, за които ще предположим, че за всеки връх е запазен размерът на поддървото. То тогава големината на полученото при сливане дърво ще бъде $T.size = L.size + R.size + 1$. Така получаваме дърво с изчислени големини на поддърветата във всеки връх.

При модифицирането на Split правим аналогично предположение – големините на поддърветата във всеки връх от изходното декартово дърво са пресметнати. При всяко рекурсивно делене на $T.Left$ или $T.Right$, преизчисляваме големината на съответно R или L.

7. Допълнителни параметри

При деклариране на декартово дърво, не сме лимитирани само на два параметъра - *(x;y)*. Може да бъде построено декартово дърво с неограничн брой параметри за всеки връх. За целта те трябва да бъдат обявени при построяването на всеки връх и за целта Merge и Split трябва да бъдат модифицирани. Тези модификации стават по аналогичен начин на модификациите за намирана на K–ия по големина елемент. При всяко

извикване на Merge и Split преизчисляваме параметрите на обходените върхове от тези функции.

Например, за някои задачи можем да добавим допълнителен параметър, показващ големината на поддървото в даден връх (както при задачата за намиране на *K*-ия по големина елемент в дървото) или някакъв параметър Cost – стойност на връх, която можем да променяме.

8. Lazy Propagation

При промяна на параметрите на няколко елемента от декартовото дърво всички параметри в поддърветата на тези елементи ще трябва да бъдат преизчислени, което би отнело много време. Затова можем да отложим действията за някои върхове (през които не сме преминали) и да ги изпълним когато минем през дадените върхове.

Lazy Propagation/Отложени действия – операции върху връх или множество от върхове, които не са изпълнени веднага – отложени са. Тези операции биват изпълнени когато върховете с отложени действия са разгледани, чрез извикването на Merge и Split. За да извършваме отложените действия трябва да въведем допълнителен параметър или параметри във всеки връх, с които да обозначим наличието на отложени действия във върховете. Отложените действия са полезни когато искаме да приложим еднотипни операции върху група от върхове. Вместо да изпълняваме действията за всеки връх, ние отлагаме тези операции и ги прилагаме само върху върхове, от които имаме нужда. Тъй като всички операции в декартовото дърво се извършват чрез прилагането на Merge и Split, можем да модифицираме тези две функции така, че да поддържат отложени действия.

Нека имаме операцията увеличаване на стойността(някакъв параметър Cost) на група върхове. За целта ще въведем нов параметър *Add* във всеки връх, който обозначава, че във всички върхове от поддървото с корен текущия връх трябва да бъде добавена стойност равна на *Add*. При преминаване през даден връх, ако *Add* е различно от 0, променяме стойността на върха – увеличаваме параметъра Cost - $X.Cost = X.cost + X.Add$, и променяме стойността на параметъра *Add* на наследниците му - $X.Left.Add = X.Left.Add + X.Add$ и $X.Right.Add = X.Right.Add + X.Add$. След това нулираме стойността на параметъра *Add* в текущия връх, с което обозначаваме, че отложените действия в този връх са изпълнени. Прехвърляме отложените действия на наследниците на разгледания връх, защото наследниците му са останалите елементи от групата.

При изпълняване на операцията за промяната на стойността на група елементи, отделяме тези елементи от остатъка от дървото, чрез извикване

на Split два пъти, в ново декартово дърво. По този начин всички елементи от текущата група ще се намират в едно поддърво след сливане на отделеното декартово дърво с останалите две части. Затова променяме параметъра *Add* само на корена на това ново дърво, гарантирайки че стойностите на всички елементи от исканите ще бъдат променени, когато преминем през тях, защото са наследници на елемент, чийто параметър *Add* е различен от 0. След това изпълняваме Merge, получавайки декартово дърво с поддърво, стойностите на върховете на което трябва да бъдат променени – това поддърво всъщност е групата от елементи, чиито стойности трябва да бъдат променени.

Тъй като всички операции върху декартово дърво работят чрез Merge и Split, тези две основни операции трябва да бъдат модифициране, за да поддържат параметъра *Add*, тоест при всяко рекурсивно извикване на тези две основни функции преизчисляваме стойността (ако параметърът *Add* е различен от 0) на всеки връх, през който сме преминали и променяме параметрите *Add* на наследниците му, отлагайки действията надолу по дървото.

Нека разгледаме Split. Тъй като получаваме две нови дървета и губим изходното, то трябва да изпълним част от отложените действия в дървото. Започвайки от корена, променяме стойността в този връх $X.Cost = X.Cost + X.Add$, прибавяме *T.Add* към *Add* на левия и десния му наследник, след което да продължим с рекурсивното извикване на Split. Правим аналогични действия с всеки връх, през който преминаваме, като след изпълняването на действията в даден връх трябва да променим $Add = 0$.

Аналогично модифицираме и Merge – при всяко преминаване през връх при рекурсивното извикване на Merge, изпълняваме отложените действия в текущия връх и променяме параметрите *Add* на левия и десния му наследник, означавайки, че в тези върхове трябва да бъдат изпълнени действия.

По този начин не губим време за преизчисляване на стойностите на върхове, които не са ни нужни – ако не сме преминали през даден връх, отложените действия в него няма да бъдат изпълнени. Тъй като отложените операции са изпълнени при самото изпълняване на Merge и Split и изпълняването им е със сложност $O(1)$ – просто промяна на стойностите на няколко променливи, то тази модификация няма да промени бързината на работа на Merge и Split и те ще си останат с логаритмична сложност.

III. Implicit Treap

1. Определение – Нека разгледаме декартово дърво с неявни(без) ключове *x* във върховете, всеки връх на което се дефинира само от приоритет *y* и някакъв параметър или стойност *C*. Тъй като за декартовите дървета е изпълнено свойството на BST, то трябва да подберем ключовете *x*, че това свойство да бъде изпълнено. Числата от 0 до *N-1* поставени по подходящ начин удовлетворяват това условие. По този начин декартовото дърво може да се представи като масив, където ключовете *x* всъщност са индексите на елементите от масива, а параметърът *C* – стойността на тези елементи, приоритетът *y* е само, за да са изпълнени свойствата на декартовото дърво.

2. <u>Merge</u> – Условието, при което можем да слеем две декартови дървета е всички ключове от първото да са със по-голяма стойност от всички на второто. Затова, когато имаме декартови дървета без ключове, можем да използваме <u>Merge</u> без да променяме алгоритъма – функцията зависи само приоритета на съответните елементи от декартовите дървета, предполагайки, че условието за ключовете е изпълнено.

С помощта на тази операция можем да слеем два масива за логаритмично време – представяме двата масива като *имплицит трипове* и прилагаме операцията <u>Merge</u> върху тях, в последствие на което върховете, отговарящи на елементите от втория масив ще придобият ключове (подбрали сме числата от 0 до големината на масива да играят ролята на ключове) от N до $N + M - 1$, ако големините на изходните масиви са N и M.

3. <u>Split</u> – Операцията за разделяне на декартово дърво на две части изисква ключ, спрямо който да ги разделим. Но как ще разделим *имплицит трип* на две части, все пак нямаме ключове във върховете(всъщност имаме индексите на елементите на масива като ключове, но с тези ключове не може да бъде извършена операцията за разделяне *на имплицит трип*)? Това може да бъде извършено с добавянето на параметър *Size* – големината на поддървото с корен дадения връх – този параметър ще играе ролята на ключ при тази операция.

Операцията <u>Split</u> ще изглежда по следния начин – нека S_L е размерът на поддървото на левия наследник на корена. Има две възможности:

1) Ако $S_L \leq x_0$, то коренът се намира в левия *трип*, получен след разрязването => дясното поддърво трябва да бъде рекурсивно разрязано, но ключът x_0трябва да бъде намален - $x_0 = x_0 - S_L - 1$, защото $S_L + 1$ елементи са вече в левия резултат.
2) Ако $S_L > x_0$, то коренът се намира в десния *трип*, получен след разрязването => лявото поддърво трябва да бъде рекурсивно

разрязано. Този случай не е симетричен на предишния, защото не променяме ключа x_0, защото все още нямаме елементи в левия резултат.

Операции с масиви – *имплицит трип* може да се разгледа като масив, където неявните ключове *x* са всъщност индексите на елементите на масива. Това свойство позволява извършването на множество операции с масиви с логаритмична сложност.

1. Добавяне на елемент на определена позиция – добавянето на елемент става по аналогичен начин на добавянето на елемент в обикновено декартово дърво, само че вместо ключ имаме индекса на позицията, на която трябва да добавим елемента. Добавянето на елемент на позиция Pos ще стане по следния начин:

- Разделяме масива $T[0; N-1)$ на $L[0; Pos)$ и $R[Pos, N-1)$ с помощта на Split с ключ Pos.
- Построяваме *имплицит трип* от един връх – елемента, който искаме да добавим. Върхът ще има ключ *Pos* и случайно генериран приоритет.
- Сливаме *трипа* от един връх с L.
- Сливаме L с R.

2. Премахване на елемент от позиция – Операцията е извършена по аналогичен на операцията за добавяне на елемент начин:

- Разделяме масива $T[0; N-1)$ на $L[0; Pos)$ и $R[Pos, N-1)$ с помощта на Split с ключ Pos.
- Разделяме $R[Pos, N-1)$ по ключ 1 – премахваме само първия елемент.
- Сливаме R с L.

3. Операции върху част от масива – Операции върху част от масива се извършват с помощта на Lazy Propagation. Lazy Propagation върху *имплицит трип* не е по-различно от това при декартово дърво. Имайки работещи Split и Merge, можем да ги модифицираме, така че при всяко тяхно рекурсивно извикване, да изпълняваме отложените действия(ако има) за текущия връх и да ги прехвърляме на наследниците му.

За да извършваме операции върху част от масива, първо трябва да отделим тази част от *имплицит трипа* с две прилагания Split. Ще променим параметрите, отговарящи за отложените действия, на корена на отделеното декартово дърво(частта от масива). Нужно е да променим параметрите само в корена, защото параметрите, отговарящи за отложените действия, в наследниците ще се изчислят при прилагане на другите операции(може и да

не се преизчислят, ако не е нужно да разглеждаме някой връх). След това връщаме частта от масива обратно с две извиквания на Merge.

4. Циклично завъртане на част от масив - Циклично завъртане на един масив на K позиции е първите K елемента от масива да отидат на последните К позиции или на обратно – последните K елемента да минат на първите K позиции.

Тази операция може да бъде извършена с едно извикване на Split и едно извикване на Merge. Първо разделяме масива чрез Split по ключ K или ключ $N -$ К(ако цикличното завъртане е в обратната посока). След което сливаме масивите в обратен ред на този, по който сме ги разделили. Тъй като операцията Merge дописва елементите на втория масив след елементите на първия, то в нашия случай вторият масив(сливаме ги в обратен ред) са първите K елемента и те ще отидат на последните позиции.

5. Обръщане на подмасив – Тази операция може да бъде извършена чрез използване на отложени действия. За всеки връх добавяме булев параметър *Reversed* , показващ дали дадена част от масива трябва да е обърната – дадената част са всички върхове от поддървото с корен текущия връх.

Ще модифицираме Merge и Split. При преминаването през даден връх от тези функции, ще проверяваме дали дадена част от масива трябва да бъде обърната($Reversed = 1$). Ако това е изпълнено, разменяме местата на лявото и дясното поддърво на текусхтия връх – лявото става дясно, а дясното- ляво. Това е възможно, защото все пак нямаме ключове във върховете.

При модификациите на Merge и Split при изпълняването на отложените действия за текущия връх прехвърляме параметъра *Reversed* на наследниците на текущия връх посредством операцията ксор(^) – ако някой от наследниците участва в обръщане, двете обръщания се самоизключват – елементът не си променя мястото.

Време на работа на операциите върху масив: Всички горни операции работят със сложност $O(\log_2 N)$, защото всички те се основават на няколкократно прилагане на Merge и Split върху масива, които работят с логаритмична сложност.

Имплементацията на Implicit Treap може да намерите в **Приложение А** или файла *implicit_treap.cpp.*

IV. Съпоставка на Implicit Treap с други структури от данни

Тестовете са извършени на компютър с процесор *Intel core i7* и операционна система *Linux Mint* на задачата **RMQ(Range Minimum Query)** - намиране на минимален елемент в подмасив.

Изследвани са решенията:

- Segment Tree – Интервално дърво със сложност на заявка $O(\log_2 N)$.
- Naive – Наивно решение (Brute Force) със сложност на заявка $O(N)$.
- **Implicit Treap** – Декартово дърво със сложност на заявка $O(\log_2 N)$.
- **Implicit AVL** – Implicit AVL със сложност на заявка $O(\log_2^2 N)$.

	$N = 10^5$ $M = 10$	$N = 10^5$ $M = 10^2$	$N = 10^5$ $M = 10^3$	$N = 10^5$ $M = 10^4$	$N = 10^5$ $M = 10^5$	$N = 10^5$ $M = 10^6$	$N = 10^5$ $M = 10^7$	$N = 10^5$ $M = 10^8$
Segment tree	0.01s	0.02s	0.03s	0.05s	0.07s	0.23s	1.01s	9.30s
Naive	0.01s	0.15s	2.21s	4.57s	8.27s	16.73s	173.53s	> 10min
Implicit Treap	0.01s	0.02s	0.03s	0.10s	0.11s	0.34s	2.24s	13.15s
Implicit AVL	0.02s	0.02s	0.04s	0.12s	0.19s	0.46s	2.68s	15.23s

Bold – означени са структурите от данни, поддържащи добавяне и премахване на елементи и други операции.

Резултатите показват, че ако има възможност, то интервалното дърво е най-ефективно, но в задачи, изискващи добавяне или премахване на елементи от масив, декартовото дърво(*Implicit Treap*) е по-ефективно от Implicit AVL.

V. Implicit treap в състезателното програмиране

Имплицит трипът може да се използва като заместител на интервални или индексни дървета в повечето задачи.

Пример 1:Даден е масив с *N* елемента. Имаме *M* заявки. Всяка заявка е от вида:

1) Обръщане на подмасива между позиции *l* и *r* в масива .
2) Минимален елемент между позиции *l* и *r*.

Решение: Тази задача може да бъде решена с интервално дърво и Lazy Propagation. Но решението и с *имплицит трип* е по-лесно за писане – първия вид заявки са просто разгледаната операцията за обръщане на подмасив, а втория вид заявки изискват добавянето на допълнителен параметър във всеки връх и Lazy Propagation.

За изпълняването на втория вид заявки въвеждаме нов параметър *Min* във всеки връх, равен на върха с минимална стойност в поддървото на текущия връх. При отделянето на търсения подмасив, отговорът на заявката е *Min* в корена на полученото декартово дърво. Параметрите *Min* ще бъдат преизчислявани при всяко извикване на Merge и Split - $T.Min = Minimum(T.Left.Min,\ T.Right.Min)$, където T е връх, обходен при извикването на Merge и Split.

Това решение работи със сложност $O(Mlog_2(N))$, с такава сложност работи и решението с интервално дърво. Решението на тази задача на езика C++ може да намерите в файла *problem1.cpp*.

В някои задачи обаче използването на интервални и индексни дървета е невъзможно, защото те не могат да поддържат някои операции(които *имплицит трипа* поддържа). Например добавяне на елемент, премахване на подмасив, циклично завъртане. Тези задачи могат да бъдат решени с използването на много сложни структури от данни, но решението им с *имплицит трип* е много по-лесно и бързо за писане, не отстъпвайки по сложност и бързина на работа и използвана памет.

Пример 2: Даден е масив *A* с *N* на брой елемента. Имаме *M* заявки, всяка от които е от вида:

1) *Insert* - Добавяне на елемент в масива на позиция *k* със стойност *x*.
2) *Erase* - Изтриване на интервал от масива между позиции *l* и *r*.
3) *Reverse* - Преобръщане на подмасив между позиции *l* и *r*.
4) *Shift* - Циклично завъртане на подмасив между позиции *l* и *r* с *k*.
5) *RMQ* - Заявка за минимален елемент между позициите *l* и *r*.
6) *Update* - Промяна на стойността на всички елементи между *l* и *r* със *k* (увеличава стойността на всеки елемент от масива между позиции *l* и *r* с *k*).

Решение: Първо построяваме *имплицит трип* със следните допълнителни параметри:

- *Min* – връх с минимална стойност в поддървото на всеки връх.
- *Reversed* – дали даден елемент от масива е обърнат или не.

- *Add* – каква стойност трябва да се добави към стойността на текущия връх.

Ще разгледаме всяка заявка.

Insert е просто добавяне на елемент на дадена позиция. За изпълняването и прилагаме Split с $x_0 = k$ и получаваме 2 *имплицит трипа – L* и *R*. Построяваме *имплицит трип* с 1 връх *M* със стойност *x*. Прилагаме Merge върху *L* и *M*, след това прилагаме Merge върху получения *имплицит трип* и *R*.

Erase може да бъде изпълнена с няколко извиквания на Merge и Split. Прилагаме Split с $x_0 = l$ и получаваме два *трипа* - *L* и *R*, където в *L* се намират всички елементи преди позицията, а в *R* – всички останали елементи от масива. Затова прилагаме Split с $x_0 = r - l$ върху *R* и получаваме 2 *трипа – L'* и *R'*. *L'* съдържа елементите на подмасива, който искаме да изтрием, а в *R'* са всички елементи от масива , намиращи се след позиция. Прилагаме Merge върху *L* и *R'* и получаваме искания резултат.

Reverse е вече разгледаната операция за преобръщане на подмасив и е извършена чрез Lazy Propagation върху *имплицит трипа* ни – използваме параметъра *Reversed*.

Shift е разгледаната операция за циклично завъртане. Извършва се едно извикване на Split и едно извикване на Merge. Първо разделяме масива чрез Split по ключ $x_0 = k$. След което сливаме масивите в обратен ред на този, по който сме ги разделили.

RMQ е заявка, разгледана в предишната задача. Използва се параметър *Min* във всеки връх, равен на върха с минимална стойност в поддървото на текущия връх, преизчисляван при всяко извикване на Merge и Split - $T.Min = Minimum(T.Left.Min,\ T.Right.Min)$, където *T* е връх, обходен при извикването на Merge и Split.

Update може да бъде извършена чрез въвеждането на параметъра *Add* и използването на Lazy Propagation. При всяко извикване на *Update* отделяме масива между позициите *l* и *r* в отделен *трип M* по аналогичен начин на операцията *Erase*. Променяме параметъра *Add* на корена на получения *трип* $M.Root.Add = M.Root.Add + x$. Прилагаме Merge върху *L* и *M* и още веднъж прилагаме Merge върху получения *трип* и *R*. При всяко преминаване от Merge или Split през даден връх *T* добавяме параметъра *Add* към стойността на върха $T.Val = X.Val + X.Val$. И променяме параметъра *Add* на неговите наследници (ако има наследници) - $T.Left.Add = T.Left.Add + T.Add, T.Right.Add = T.Right.Add + T.Add$.

Решението на тази задача с Implicit Treap е със сложност $O(Mlog_2(N))$, защото всички операции със тази структура се извършват с няколко извиквания на Merge или Split, които работят за логаритмично време, тоест всяка заявка се изпълнява с логаритмична сложност. Тази задача може да бъде решена с използването на Implicit AVL Tree. Решението с тази структура ще работи със сложност $O\left(Mlog_2^2(N)\right)$, което е осезаемо по-бавно от решението с Implicit Treap при големи стойности за N и M. Двете структури от данни заемат приблизително еднаква памет, затова решението с Implicit Treap е по-добро.

Решението на тази задача на езика C++ може да намерите в файла *problem2.cpp*.

VI. Заключение

Документацията е една от малкото, обхващащи структурите от данни Treap и Implicit Treap, както и единствената на български език, известна на авторите. Разгледана е същността на тези структури от данни, както и най-важните операции с тях. Обяснени са основни стратегии и техники за използване на тези структури. Решени са няколко примерни задачи, в които Treap и Implicit Treap са съпоставени на други структури от данни по време и сложност на работа и заета памет.

Литература:

1) http://habrahabr.ru/post/102364/ - Декартово дерево по неявному ключу
2) www.eecs.berkley.edu/~jshun/a8-shun.pdf - A Simple Parallel Cartesian Tree Algorithm and its Application to Parallel Suffix Tree Construction
3) perso.ens-lyon.fr/eric.thierry/WCPS2015/data/uploads/lectures/20151027.pdf – ENS Lyon Camp. Day 2. Basic group. Cartesian Tree.
4) http://www.cs.cmu.edu/afs/cs/academic/class/15210-s12/www/lectures/lecture16.pdf - Lecture 16 - Treaps; Augmented BSTs
5) http://www.win.tue.nl/~mdberg/Onderwijs/AdvAlg_Material/Course%20Notes/lecture3.pdf - Randomized Searching: skip lists and treaps.