УЧЕНИЧЕСКИ ИНСТИТУТ ПО МАТЕМАТИКА И ИНФОРМАТИКА

ПЕТНАДЕСЕТА УЧЕНИЧЕСКА СЕКЦИЯ
УК'15

ТЕМА НА ПРОЕКТА

Персистентни структури от данни

Автори:

Димитър Николаев Карев

Петър Петров Велков

10 клас

ПМГ „Акад. Боян Петканчин" – гр. Хасково

Научен ръководител:

Христо Цветанов Стоянов

Ученик в СМГ „П. Хилендарски", София

Резюме

В компютърното програмирането преобладават ефимерните структури от данни, които не притежават механизъм, с който да могат да се преглеждат предишни състояния, нито пък да бъдат променяни. Персистентните структури от данни предлагат подобен механизъм и функции. Пълната персистентност предлага достъп до всички състояния, както и възможността да се променят стойностите не само на текущата структура, но и на предходните.

Тривиалният начин за постигане на персистентност е чрез копирането на стойностите на текущата версия, преди да се обнови, като по този начин се запазват всички състояния на структурата. Това обаче изисква допълнително време за изпълнение и значително количество памет. Чрез този проект представяме имплементации на различни персистентни структури от данни като списък, стек, масив, индексни дървета, Disjoint-Set и интервални дървета.

Persistent Data Structures

Summary

The most frequently used data structures in computer programming are ephemeral data structures, which do not possess a mechanism to access previous states, nor to alter them. Persistent data structures offer such functions and mechanisms. Full persistence offers access to all states, as well as the ability to alter not only the present values, but also the previous ones.

The trivial manner of reaching persistency is implemented by copying the values of the current version before it is updated, in order to preserve all states of the structure. This however requires additional execution time and significant amount of memory. By creating this project we aim to present some implementations of various persistent data structures such as List, Stack, Array, Segment Trees and Disjoint-Set.

1. Увод

С този проект авторите си поставят за задача да популяризират предимствата от персистентните структури от данни, като ги съпоставят с обикновените ефимерни структури. Те също така целят да разкрият, както значението им в теоретичната информатика, така и приложенията им в практиката и индустрията.

Ограничените персистентни структури и версиите им образуват обикновен линеен път, докато при напълно персистентните структури този път има вид на дърво. В проекта се разглеждат само пълните персистентни структури.

Приложенията на персистентните структури от данни са многобройни. Въпреки това и до ден-днешен те са малко известни. При многоядрените процесори се развива съвместно изчисление, като така се ускорява работата на компютъра и се намаляват негативните черти откъм време и памет на персистентните структури. По този начин значението им за технологичната индустрия се засилва и тази тенденция ще продължи в бъдеще. Те са изключително безопасни, тъй като не променяме старите версии, а всеки път създаваме нови. Така лесно винаги може да се възстанови старо и безопасно състояние.

Персистентните структури от данни имат приложения в изчислителната геометрия, обработката на текст, файлове и други. Използват се в реализацията на функции подобни на undo и redo. Също така са полезни за имплементацията на езици от типа VHLL – Very High Level Language като Ruby, Python и Perl.

Персистентните структури могат да заменят напълно ефимерните заради безопасността, която дават, както и заради това, че често сложността им по време и памет не отстъпва от тази на неперсистентните. Ефективно имплементирани те са изключително полезно средство в света на новите технологии.

2. Дефиниции

Дефиниция 1. Персистентна структура: Персистентна структура от данни наричаме такава структура от данни, която винаги запазва предходните си версии след промяна.

Дефиниция 2. Ефимерна структура: Структура от данни, която не е персистентна.

Дефиниция 3. Базисна / Основна структура и версия на структура: При прилагането на промени върху една структура се образуват нови производни. Новополучените наричаме версии, а първоначалната структура базисна или основна.

Дефиниция 4. Полуперсистентна структура: Структура, която позволява достъп до стойностите на всички състояния на структурата, но промяна може да бъде извършена единствено върху последната версия.

Дефиниция 5. Confluent persistence – Персистентна структура, която позволява сливането на две версии

.**Дефиниция 6. Functional persistence** – Персистентна структура, която не позволява промяната на елемент, но позволява добавянето на нов.

Дефиниция 7. Пълно двоично дърво: Това е двоично дърво, което има N нива и се състои от 2^N-1 върха.

3. Свързан списък

3.1. Ефимерен свързан списък

Списъкът е линейна структура от данни, която подобно на масив, съдържа в себе си поредица от елементи. За разлика от масива, където елементите се съхраняват последователно в паметта – статично, при списъка елементите са разпръснати. За да могат да се обхождат в оригиналната си последователност, всеки елемент „сочи" мястото в паметта, където се намира следващия елемент и също така пазим адреса на началото на списъка. Използвайки съхраняваната информация за съседните му елементи и „движейки" се по тях, можем да стигнем до всеки един елемент от списъка. Макар динамичната същност на списъка да усложнява достъпа до него, тя позволява друг вид операции като премахване и добавяне на елементи.

3.1.1. Структура:

Всеки елемент съдържа три параметъра:

- Стойност на елемент - Value
- Адрес на следващ елемент - *Next

3.1.2.Функции:

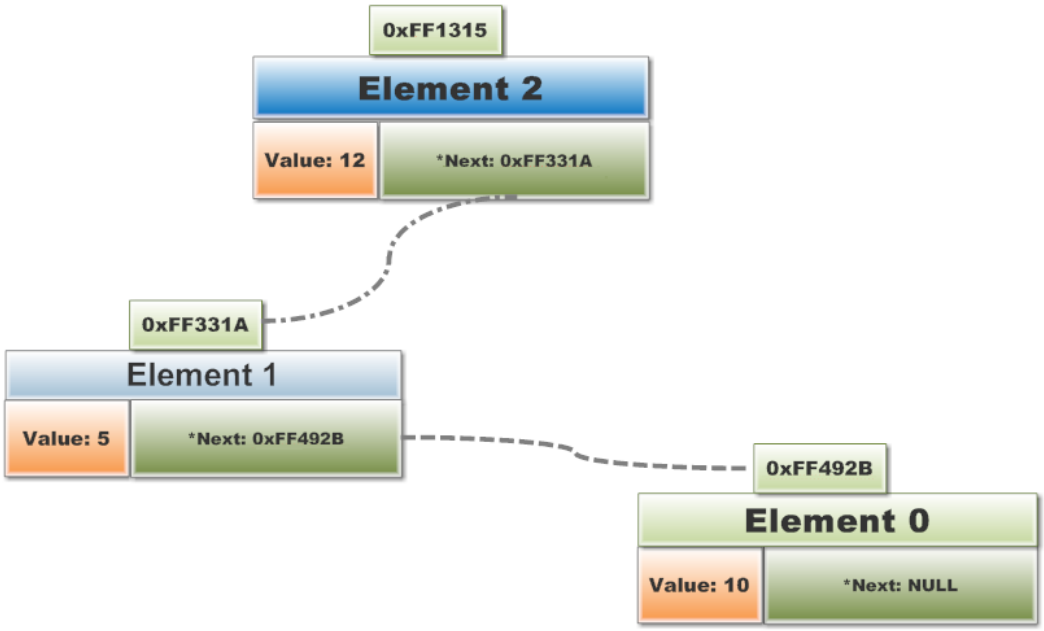
- Отпечатване на стойност на елемент - $FindElement$ (...)
- Промяна на стойност на елемент – $Update$ (...)
- Добавяне на елемент – $Insert$ (...)
- Отпечатване на списък – $Print$ (...)

Отпечатване на стойността на $i^{-тия}$ по ред елемент – $FindElement(\ i\ )$

За да бъде отпечатана стойността на елемент, то първо той трябва да бъде достъпен. Започвайки от началния елемент, обхождаме всеки следващ, докато не достигнем до желания и изведем неговата стойност.

Използвайки адресите на съседните елементи, се придвижваме $\boldsymbol{i-1}$ пъти по списъка и отпечатваме стойността **Value.**

Сложността на функцията е $O(i)$.



Промяна на стойност – *Update*(*i, NewValue*)

Задава стойност NewValue на $i^{-тия}$ по ред елемент.

Аналогично на функцията ***FindElement*** (…)**,** първо е необходимо да обходим списъка, докато не достигнем до $i^{-тия}$ по ред елемент, но с разликата, че тук няма да отпечатваме стойността, а ще му зададем нова – **NewValue**.

Сложността на функцията е $O(i)$.

Добавяне на елемент – *Insert* (*i, NewValue*)

Поставя нов елемент с стойност NewValue между $i-1^{-ия}$ и $i^{-тия}$;

При масивите добавянето на елементи е лесно и бързо, но само когато се поставят в края. Ако се поставят в междинна част, то голям брой от елементите ще трябва да бъдат изместени. Чрез списъците могат да се добавят нови части към всеки отрязък от структурата. За да се постави нов елемент между два вече съществуващи е необходимо първо да се прекъсне връзката между тях и да се установи нова, минаваща през добавената част, така че списъкът да бъде свързан правилно и да може да бъде обхождан.

Новия елемент се свързва със следващия ($\boldsymbol{i}^{-тия}$), посредством указател Next, и по аналогичен начин предходния **(** $\boldsymbol{i-1}^{-ия}$), свързваме с новия.

Сложността на функцията е $O(i)$.

Отпечатване на списък – $Print$ ( )

Отпечатва стойността на всеки елемент от списъка в съответната последователност, започвайки от началото на структурата и се придвижваме към следващите елементи, като извеждаме стойностите им.

Сложността на функцията е $O(N)$.

След като имаме обикновен списък ще можем да направим и персистентен такъв. Нека го разгледаме.

3.2. Персистентен свързан списък

Персистентният списък ще изпълнява същите функции, но те ще бъдат възможни за всички версии на списъка. При ефимерен списък всяка версия, освен началната произлиза от предишната, т.е. е получена чрез прилагане на функциите $Insert$ (...) или $Update$ (...) върху предходната версия. Всяко състояние освен последното води към ново. Поради това всички версии на списъка са свързани линейно.

При персистентния списък всяка версия, създадена в хода на работа се съхранява, като по този начин запазваме достъпа си до нея. В такъв случай можем да продължим да прилагаме функции върху нея и оттам да създаваме многобройни версии, използвайки я за основа. При наличието на персистентност, версиите са свързани подобно на дърво. Започвайки от първоначалния списък и разклонявайки се към производните му версии.

3.2.1. Функции

За да се постигне персистентност, при всяко добавяне на елемент или промяна на стойност е необходимо да запазваме, както новополучената версия, така и основната. Следователно никоя от версиите, които сме получили не се премахва, всички се запазват, за да могат да бъдат използвани отново. Една неефективна реализация би била да копираме целия списък и да прилагаме желаната функция. В такъв случай бихме създавали N нови елемента за всяка промяна, която извършим. Ако разгледаме двата списъка - основния, този върху който сме направили промяна, и производния, получения от него, ще забележим, че се различават с един елемент. Затова можем да направим оптимизация по следния начин:

1. При всяка промяна ($Update( ) / Insert ( )$) добавяме към масив Beginnings[] нов елемент, началния за всяка версия. Тоест Beginnings [k] е първият елемент от версия k.

2. След като запазим началния елемент, трябва да построим останалата част от списъка. Тъй като всички елементи от двете версии освен $i^{-тия}$ са идентични, можем да свържем $i^{-тия}$ елемент от новополучената версия с $i + 1^{-ия}$ елемент от базисната. Така построяваме част от списъка надясно от $i^{-тия}$ елемент като създаваме само един нов елемент.

3. Подобна техника обаче не може да се използва за първата част от списъка, тъй като не е двойно свързан, а оптимизацията би изисквала повече от една връзка в дадена посока.

4. Затова построяването на първата част се състои в копиране на първите $i-1$ елементи и свързването им към началото на версията Beginnings [k].

Отпечатване на стойността на i–тия елемент от версия с номер State – $FindElement(State, i)$

Началото на версия **State** се съдържа в **Beginnings [State],** затова чрез него се обхожда списъка до достигането на i–тия елемент и се извежда стойността му Value.

Сложността на функцията е $O(i)$.

Добавяне на елемент – $Insert(i, State, NewValue)$

Във версия с номер State поставя нов елемент с стойност NewValue между $i-1$-ия и i-тия;

Добавяме нов елемент към Beginnings, след което последователно към него свързваме копираните елементи, които се намират между началото на версия с номер State и $i-1$-ия елемент.

Сложността на функцията е $O(i)$.

Промяна на стойност – $Update(i, State, NewValue)$

Задава стойност NewValue на i–тия по ред елемент във версия State.

Добавяме нов елемент към масива **Beginnings** (върха на новата версия) и към него свързваме копия на елементите разположени между началото на версия **State** и $i-1$-ия му по ред елемент. След това добавяме i-тия елемент с обновената стойност **NewValue** и го свързваме посредством указател към $i+1$-ия елемент от списъка **State**.

Сложността на функцията е $O(i)$.

Отпечатване на списък – $Print(State)$

Отпечатва стойността на всеки елемент от версия **State** в съответната последователност. Започва от елемент Begginings [State] и отпечатва всички елементи до края на версията. Последният елемент от версията е дефиниран със следващ елемент Next равен на NULL.

Сложността на функцията е $O(N)$.

Персистентния вид на структурата има същата сложност като ефимерната. Свързаният списък може да не е от структурите, които имат изключителна използваемост, но е добър пример за представянето на персистентния вид на структура.

Забележете, че ако правихме промени само в края на списъка, сложността ни щеше да се намали многократно. Нека разгледаме такъв случай.

4. Стек

4.1. Ефимерен стек

Стекът е линейна структура от данни, в която обработката на информация става само от едната страна, наречена връх. Стековете са базирани на принципа **LIFO – "Last**

In First Out” – последен влязъл, пръв излязъл. Стекът се имплементира подобно на описания по-горе списък, всеки елемент пази адреса на елемента под него. Първият елемент (връх), чийто адрес не се пази от никой друг елемент от стека, се съхранява в отделен елемент, който се обновява всеки път щом променим върха. Последният (най-долният елемент от стека / дъното) се инициализира със стойност ***NULL***, за адреса на елемента под него. Тъй като подобен не съществува.

4.1.1. Структура:

- Стойност на елемент - **Value**
- Адрес на долен елемент – ***Next**

4.1.1. Основни функции:

- $Push$ (…)
- Pop (…)
- Top (…)

Поставяне на нов елемент на върха – $Push$ ($NewValue$);

За да добавим нов елемент и за да запазим функциите и механизма на структурата, е необходимо да създадем нов елемент от съответната структура и да го свържем към вече съществуващия връх, посредством указателя – **Next**, който „сочи“ адреса на следващия елемент. По този начин структурата остава свързана и може да бъде обхождана. Също така обновяваме елемента, който пази върха на стека, за да имаме актуален достъп до него.

Сложността на функцията е $O(1)$.

Премахване на връх - Pop( );

Върха се премахва, като обновим указателя пазещ адреса на върха на стека, задавайки му за адрес, стойността на указателя – **Next** на стария връх, сочещ към елемента под него. Това е така, тъй като, след премахването на върха, елемента под него ще стане новият връх.

Сложността на функцията е $O(1)$.

Отпечатване на връх – Top ( )**;**

Използваме елемента, чрез който пазим върха на текущия стек и отпечатваме неговата стойност - **Value**.

Сложността на функцията е $O(1)$.

4.2. Персистентен стек

Тук достигането на пълна персистентност би изисквало масив, в който да запазваме копие на върха на съответната версия на стека. В приложената реализация тази роля се изпълнява от масив **Beginnings**. В клетка **Beginnings[*i*]** се съхранява върха на ***i***-тата версия.

4.2.1 Структура:

- Стойност на елемент - **Value**
- Адрес на следващ / долен елемент - ***Next**

4.2.2. Основни функции:

Добавяне на нов елемент на върха на версия State – $Push\ (State, Value)$

Добавя към версията на стек с номер **State** нов елемент със стойност **Value**. Ако новата версия е **k**-та по ред то върхът и се запазва в **Beginnings [k]**. Функцията присвоява стойността Value на **Beginnings [k].** Също така за да стане част от списъка, указателя $Beginning\ [\ k\ ] -> Next$ приема за стойност адреса на **Beginnings [State].** При тази стъпка новия елемент **Beginnings [k]** посочва върха на версия S**tate - Beginnings [State]** за долен / следващ и по този начин се превръща във връх за версия k.

Сложността на функцията е $O(1)$.

Премахване на върха на версия State – $Pop(State)$

Премахва върха на версия с номер State. Тук отново се добавя нов елемент към масива **Beginnings[ ],** но в този случай върха се измества с една позиция надолу. Следователно за нов корен ще се ползва указателя $Beginnings\ [State\ ] -> Next$ (адреса на елемента, който е след върха), тъй като след премахването на върха, в новополучената версия връх е следващият елемент.

Сложността на функцията е $O(1)$.

Отпечатване стойността на върха на версия State – $Top\ (State)$

Върхът на версия State се пази в **Beginnings [State]** , затова отпечатваме стойността му – $Beginnings\ [\ State\ ] \rightarrow Value$.

Сложността на функцията е $O(1)$.

Проверка дали версия State не е празна – $Empty\ (State)$

Проверяваме дали $Beginnings\ [\ State\ ]$ сочи към $NULL$, ако това е така, то версията не съдържа елементи и функцията връща стойност *True*, в противен случай съдържа елементи и функцията връща стойност *False.*

Сложността на функцията е $O(1)$.

Успяхме да създадем оптимален персистентен стек, който работи със сложност $O(1)$ за операция. Следователно, ако имаме N заявки, сложността ни ще бъде $O(N)$. Тази сложност е оптимална. Нека сега разгледаме по-сложните персистентни структури.

5. Индексно дърво[1]

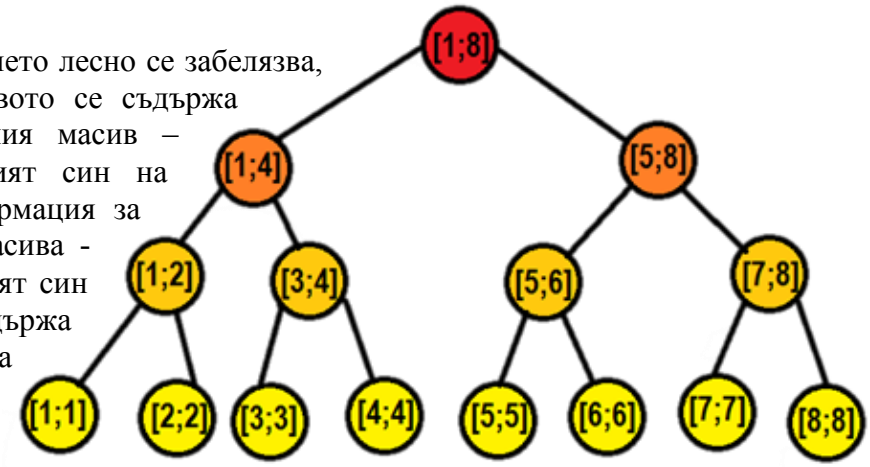
5.1. Ефимерно индексно дърво

Нека разгледаме следната задача[2]:

Дадени са N на брой банкови сметки и техните стойности. Една банкова сметка може да бъде положителна, ако клиента има пари в банката или отрицателна, ако клиента дължи пари на банката. Банката изисква следните заявки:

1. Промени сумата в банкова сметка k на NewVal.
2. Да се намери сумата от всички банкови сметки в интервала [L;R].

За да решим тази задача ще използваме **индексно дърво**. Индексното дърво е пълно двоично дърво, съдържащо информация за отделните интервали в масив с числа. Дървото се изгражда, като листата се запълват с дадените ни числа, а всеки родител съдържа сумата от децата си.



От изображението лесно се забелязва, че в коренът на дървото се съдържа информацията за целия масив – елементи $[1;8]$. Левият син на корена съдържа информация за първата половина от масива - елементи $[1;4]$. Десният син на корена съдържа информация за втората половина на масива – елементи $[5;8]$. Ако продължим със същите разсъждения, ще открием, че ако даден връх съдържа информацията за интервала $[X;Y]$, то неговият ляв син ще съдържа информацията за интервала $[X;(X+Y)/2]$, а десният му син за интервала $[(X+Y)/2+1;Y]$.

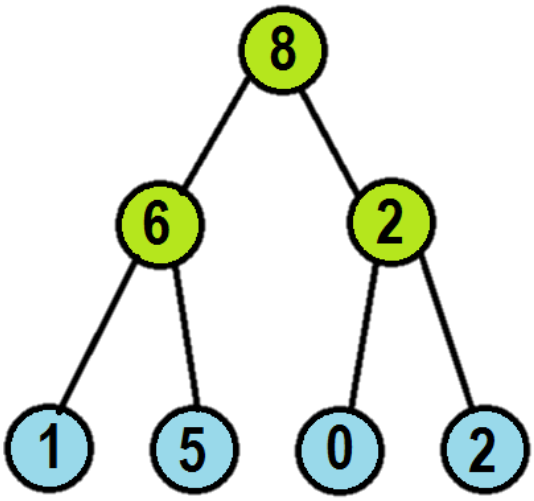
В повечето случаи ще ни се налага да заделим памет за такъв брой елементи, който не е степен на двойката. Тогава просто можем да допълним с нужния ни брой елементи до първата степен на двойката. Всеки допълнително добавен елемент ще трябва да инициализираме със стойност, която няма да ни пречи в следващите изчисления. Такава стойност може да бъде 0, ∞, -∞ или някоя друга в зависимост от задачата.

[1] Оттук нататък реализациите, които ще правим ще бъдат за намиране сума. Възможни са преобразования на структурите, така че задачата да намира минимален или максимален елемент, но ние няма да се спираме на тях.

[2] Задачите, които даваме са само примерни. Те не целят нищо друго освен да създадат представа за проблема, който трябва да реши персистентната структура.

След като имаме 2^k елементи в листата, можем да пресметнем, че общата ни нужна памет ще бъде 2^{k+1}-1, което в най-лошия случай ще е 4N. Броят на нивата в дървото ни ще бъде k+1, следователно най-дългият път от корена, до който и да е връх ще отнеме най-много k+1 или $\log_2 N$ стъпки.

Информацията за това дърво можем да пазим в статична или в динамична памет. За целите на персистентните структури ще използваме динамичното заделяне на памет. По този начин дървото може да се построи рекурсивно. Рекурсията приема за параметри „корена“ на поддървото и нивото, на което се намира дадения връх. Алгоритъма работи по следния начин:



1. Ако в дадения момент се намираме в последното ниво, то даденият ни връх е листо. Задаваме му стойност и приключваме текущата рекурсия.
2. Ако текущият връх не е листо, то тогава пускаме рекурсии, които построяват поддърветата на левия и на десния си син. След което, даденият ни връх има стойност равна на сумата (специално за нашата задача) на левия и на десния му син.

След като сме построили дървото, може да започнем да отговаряме на заявките в задачата ни. И двете заявки ще изпълняваме рекурсивно.

За първата ни заявка ще тръгнем от корена и ще се движим по дървото, като целта ни ще бъде да стигнем до листото, което отговаря за клетката, която трябва да променим. Алгоритъмът ни ще изглежда по следният начин:

1. Ако интервалът, който разглеждаме в момента, не съдържа в границите си търсения от нас елемент – приключваме рекурсията.
2. Ако сме в листото, до което искаме да отидем, променяме върха на исканата стойност и приключваме рекурсията.
3. В противен случай, пускаме рекурсии за левия и за десния син на върха, в който сме. След изпълнението на тези рекурсии обновяваме върха ни, като му зададем стойност, равна на сумата от левия и десния му син.

Така винаги поддържаме стойностите на структурата актуални.

За втората заявка ще направим подобно обхождане, както при първата, но този път ще търсим интервали, а не само една клетка. Както може да превърнем едно число в двоична бройна система, така може и да представим един интервал като множество от няколко подинтервала всеки с дължина степен на двойката. Алгоритъмът ни ще е следният:

1. Ако търсеният интервал не „споделя“ **дори една част** с текущия – приключваме рекурсията.
2. Ако търсеният интервал изцяло попада с текущия, то прибавяме стойността на дадения връх към отговора ни.

3. В противен случай, пускаме рекурсии за левия и за десния син на върха, в който се намираме.

Сложността и на двете заявки е $\mathbf{O(\log_2 N)}$. Това лесно се вижда при първата, където пътя ни от корена до листото ни отнема точно $\log_2 N$ стъпки, както доказахме по-горе. Следователно ако имаме Q заявки, то сложността ни ще бъде O(Q*$\log_2 N$).

5.2. Персистентно индексно дърво

Сега нека разгледаме следната задача:

Дадени са N на брой банкови сметки и техните стойности. Една банкова сметка може да бъде положителна, ако клиента има пари в банката или отрицателна, ако клиента дължи пари на банката. Тъй като много често стават грешки по превода на пари, банката изисква следните заявки:

1. В състояние p, промени сумата в банкова сметка k на NewVal.
2. В състояние p, да се намери сумата от всички банкови сметки в интервала [L;R].

Номерът на началното състояние на банковите сметки се счита за първи. Всяка следваща версия взима първото неизползвано естествено число за номер на състоянието.

Входните данни имат следния вид:

На първия ред се въвежда N. На следващия ред се въвеждат N на брой цели числа. На третия ред се въвежда Q. Следват Q на брой реда, всеки с по 4 числа - Type, State, a, b.

Type – показва номера на заявката;

State – показва номера на състоянието;

a и b – това са стойностите на k и NewVal или на L и R, в зависимост от стойността на Type.

След като имаме ефективно индексно дърво, ни остава само да го направим персистентно. За да направим структурата персистентна, трябва да можем да извършваме заявки в старите версии на дървото, като всяка заявка за промяна в дървото трябва да се отрази чрез нова версия.

5.2.1. Неефективна реализация

Ще започнем с неефективната реализация на персистентното индексно дърво.

За всяка заявка за промяна на дървото можем да създаваме ново дърво като копираме цялата информация от миналото състояние и променим само клетките, които се намират по пътя от корена до клетката, която променяме. След като имаме всички състояния на дървото, лесно можем да изпълним втората заявка за търсене на сумата в даден интервал, като използваме реализацията на ефимерното индексно дърво.

В реализацията си използваме статично заделяне на памет. Използваме свойството, че дървото е пълно двоично, за да се получи лесното преминаване от баща към син. Записваме корена в клетка с индекс 1. Ако имаме връх, който сме записали в клетка с индекс k, то за него е характерно, че:

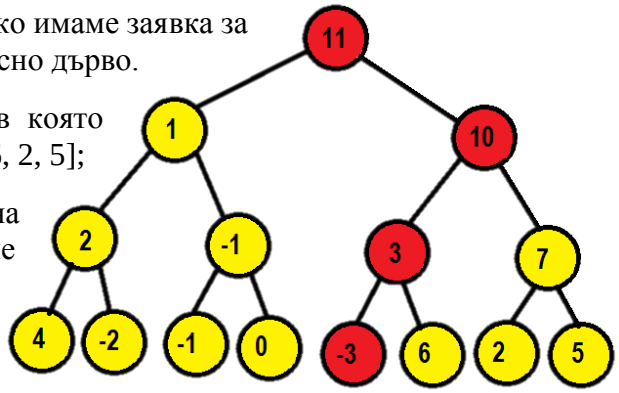
1. Бащата на върха се намира в клетка с индекс k/2;
2. Левият син на върха се намира в клетка с индекс k*2, а десният в клетка с индекс k*2+1;

Тази реализация има сложност O(Q*N) по памет и време. Ако имаме Q заявки от първи тип, то за всяка заявки ние ще заемаме по N клетки. Тъй като копираме стойността на клетките, сложността ни за време се увеличава.

5.2.2. Ефективна реализация

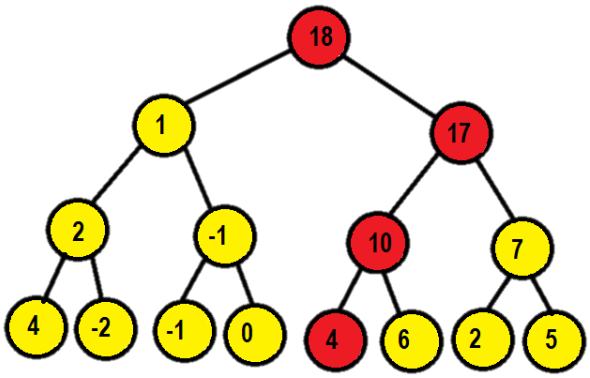
Сега ще разгледаме ефективната реализация. Тя се базира на наблюдението, че променяме само $\log_2 N$ клетки.

Нека разгледаме какво се случва ако имаме заявка за промяна на елемент в обикновеното индексно дърво.



Ако имаме версия на дървото, в която стойностите на листата са: [4, -2, -1, 0, -3, 6, 2, 5];

Искаме да променим стойността на 5-та клетка на 4. Алгоритъмът ни ще стигне до 5-та клетка, ще я промени и ще обнови всички клетки по пътя от корена до самата клетка. Ще получим следната версия:



Забелязваме, че само клетките по пътя от корена до крайния елемент се променят. Следователно ако можем да пазим само тези клетки, решението ни ще бъде оптимално.

В реализацията се ползва динамично заделяне на памет. Имаме структура Vertex, която пази стойността на дадения връх и указатели към децата на върха.

Построяването на дървото отново става рекурсивно, както описахме по-горе. Заявките имат почти еднакъв вид с тези от ефимерното индексно дърво:

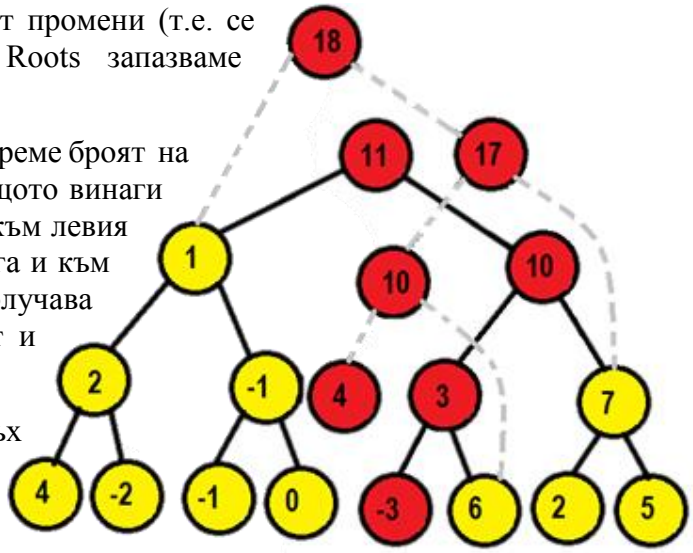
Първата заявка има за параметри Vertex *NewNode, Vertex *OldNode, int x, int y. Съответно това е указател към върха, в който ще се пази нова стойност; указател към върха от старата версия; начало и край на текущия интервал. През цялото време дърветата се движат успоредно. Алгоритъмът е следният:

1. Ако се намираме във върха, който искаме да променим. Запазваме в NewNode->val новата стойност и приключваме текущата рекурсия.
2. Ако върха, който търсим е в лявата част на интервала, тогава насочваме NewNode->Right към десния наследник на OldNode (OldNode->Right). Пускаме рекурсия за левите съседи на NewNode и OldNode и съответно за

лявата част на интервала. По този начин, нищо не се променя в дясното поддърво, а единствено в лявото.

3. Ако върха, който търсим е в дясната част на интервала, тогава насочваме NewNode->Left към левия наследник на OldNode (OldNode->Left). Пускаме рекурсия за десните съседи на NewNode и OldNode и съответно за дясната част на интервала.
4. След приключване на рекурсиите, обновяваме текущия връх[3].

Всеки път, когато настъпват промени (т.е. се създаде ново дърво), в масива Roots запазваме указател към корена на това дърво.



Забележете, че през цялото време броят на върховете намалява наполовина, защото винаги избираме дали да пуснем рекурсия към левия или към десния наследник, но никога и към двамата. По този начин се получава логаритмичната сложност по памет и по време.

Лесно се вижда, че един връх може да има повече от един родител, но това не е проблем за нас, защото обхождането на дървото винаги започва от корена.

Втората заявка е идентична с тази от ефимерното индексно дърво.

Сложността на двете заявки продължава да бъде $O(\log_2 N)$, следователно алгоритъмът е оптимален по време (тъй като персистентната форма на структурата има същата сложност като обикновено индексно дърво). Сложността на цялата структура за Q заявки е $O(Q*\log_2 N)$ и по време, и по памет.

Персистентните индексни дървета имат голямо приложение. Те са бързи откъм време за изпълнение и заемат малко памет. Те са основа за много други структури. Изчистената им идея, помага за по-доброто разбиране на персистентните структури, като цяло. Лесни са за реализиране и изключително ефективни.

[3] Тъй като структурата е персистентна, никога няма да променяме OldNode, защото по този начин бихме променили стара версия.

6. Персистентен масив

Масива е базисна структура от данни. Операциите с масив са лесни. Може да променяме клетка от масива или да намираме стойността на такава.

Оказва се, че константните решения на персистентните масиви са сложни за реализация и много често не постигат осезаема промяна. Нашата реализация на персистентния масив използва вече изучен материал - индексно дърво.

Можем да работим с персистентен масив по същия начин, както работим с персистентно индексно дърво. Поради бързината на дървото, то се използва в много случаи. При обикновена реализация на масив, логаритмичната сложност не се различава от константната.

В реализацията превръщаме масива в дърво. Това става лесно, ако стойностите на масива се представят като листа на дървото. Всъщност няма значение какво се записва във върховете, които не са листа, защото алгоритъмът ги ползва единствено за поддържане целостта на структурата.

Заявките са идентични с тези на индексното дърво, единствената разлика е, че при втората заявка търсим един елемент, а не интервал от елементи. По този начин запазваме сложността на алгоритъма и отново получаваме сложност по време и по памет - O(Q*$\log_2 N$) за Q заявки и N елемента.

7. Union-Find

7.1. Ефимерен Union-Find

Ако имаме следната задача:

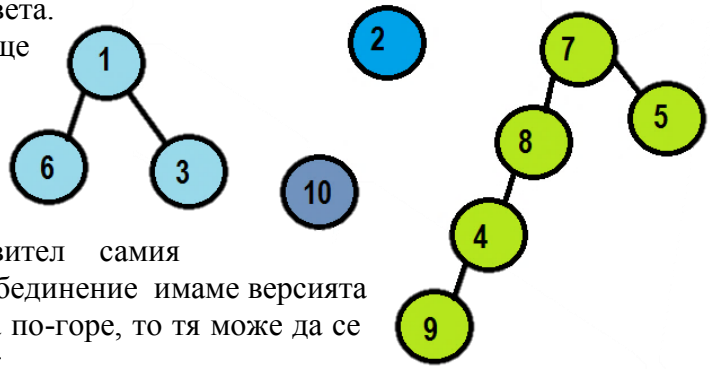
Дадени са N елемента и Q заявки. Първоначално всеки елемент образува множество, състоящо се от самия него. Заявките са от два типа:

1. Обединение множествата на елементите p и q.
2. Проверка дали елементите p и q са в едно множество.

За да решим задачата ще използваме Union-Find.

Union-Find или Disjoint Set е структура от данни, която извършва две действия – обединение на множества и търсене на представител (корен) на даденото множество.

Union-Find се представя най-често с масив, в който елементът a[k] е "баща" на елемента k. По време на изпълнението на алгоритъма във всяко едно състояние, за дадено множество може да кажем, че има само един представител и никога не може да съществува ситуация, в която един елемент да е едновременно „родител" и „наследник" на друг връх. Затова лесно може целия Disjoint Set да се представи като гора, а самите множества в него като дървета. Представителят за всяко множество ще е коренът на всяко дърво.



Най-често началното състояние на структурата се състои от N елемента, като всеки от тях образува множество с представител самия елемент. Ако след шест заявки за обединение имаме версията на множествата показана на схемата по-горе, то тя може да се представи с масив по следния начин:

1	2	3	4	5	6	7	8	9	10
1	2	1	7	7	1	7	7	7	10

Първото и най-важно наблюдение, което трябва да направим е, че ще трябва да работим единствено с представителите на отделните множества. Ако имаме горната ситуация и трябва да обединим елементите 3 и 8 в едно множество, то тогава ще бъде много по-лесно да използваме 1 и 7, защото след като обединим множествата ще сме сигурни, че ще си запазят свойството да имат само един корен. След като намерим, че корените на дърветата, в които принадлежат са 1 и 7, то тогава ни остава единствено да „прикрепим" **множеството с по-малко елементи към това с повече елементи.**

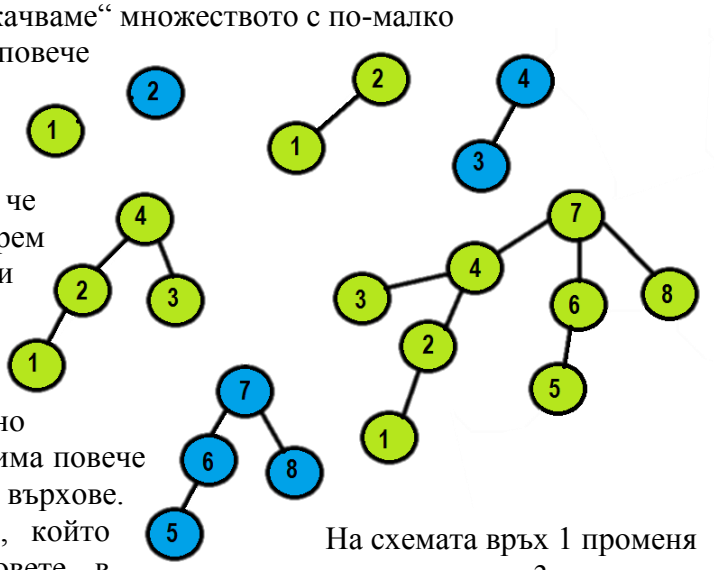
Реализацията на Union-Find е лесна за разбиране и много бърза за писане. За да я направим възможно най-ефективна ще трябва да направим още две наблюдения. Първото от тях бе показано по-рано. Когато свързваме две дървета, ще запазим корена

на дървото с повече върхове като корен на новото дърво. Или с други думи казано, „прикрепяме“ дървото с по-малко елементи към това с повече елементи. Второто необходимо наблюдение е, че веднъж узнали корена на дървото, то можем да свържем всеки връх по пътя от даден елемент към корена, със самия корен. След като извършим тези наблюдения, задачата вече става елементарна.

Целият алгоритъм се състои от две функции – за намиране на представител в множество и за обединение на две множества. Първата функция за намиране на представител в множество лесно може да се реализира рекурсивно. Функцията приема за параметър елемента, чийто корен трябва да търсим. След това функцията има следния вид:

1. Ако бащата на върха, който разглеждаме, е самият връх, то тогава ние се намираме в корена на дървото. Приключваме текущата рекурсия, като връщаме разглеждания връх.
2. В противен случай пускаме рекурсия за да намерим корена на текущия връх. Записваме резултата от изпълнението като баща на върха, който разглеждаме. Приключваме рекурсията, като връщаме същия резултат, който получихме от извикването на функцията по-рано.

Втората функция за обединение на два върха в едно множество работи по следния начин. Функцията приема като параметри двата върха, които искаме да обединим. Намира представителя от всяко множество по описания вече алгоритъм. Ако представителите са различни, „прикачваме“ множеството с по-малко елементи към множеството с повече елементи. Сложността на тези функции е логаритмична. За да докажем това ще използваме следното: Нека предположим, че имаме елемент x и искаме да разберем колко най-много пъти ще се обнови стойността му (т.е. ще се промени родителя му). Както казахме горе, ние ще променяме корена на текущото дърво само и единствено ако дървото, с което се обединява има повече (или поне същото количество) върхове. Следователно най-лошият случай, който можем да имаме, е ако върховете в множеството, което разглеждаме се увеличават двойно. Т.е. можем да променим задачата на „колко пъти трябва да умножим по две, за да получим число, което не надминава N”. Отговорът на тази задача е $\log_2 N$. Следователно всеки елемент ще се променя максимално $\log_2 N$ пъти. Това дава сложност при Q заявки – O(Q*$\log_2 N$).



На схемата връх 1 променя корена си 3 пъти.

7.2. Персистентен Union-Find

Вече имаме обикновен Union-Find. Въпросът е как да го направим персистенен? За целта просто ще сведем задачата до някоя от познатите ни.

Както казахме по-горе, Disjoint Set може удобно да се представи визуално като гора, но когато го реализираме на компютърен език ние не извършваме типичните действия, които бихме извършвали върху едно дърво. Всъщност през цялото време ние използваме само един масив. Това ни навежда на мисълта, че може да използваме персистентен масив. Но тъй като в масива, с родителите на върховете, променяме различен брой клетки всеки път, сложността може много да се повиши. За да избегнем подобно увеличение на сложността може да използваме само оптимизацията, при която винаги се „прикачва“ дървото с по-малко върхове към това с повече. По този начин за всяко обединение на множества се извършват най-много $\log_2 N$*$\log_2 N$ операции. По-късно ще обясним по-подробно изчисляването на тази сложност. След като изяснихме основната част, нека минем към подробностите от алгоритъма.

Задачата приема като входни данни N – броя на елементите, Q – броя на заявките и Q реда, всеки съдържащ 4 числа – Type, State, Ind, Ind1; Type показва коя заявка трябва да се изпълни; State показва версията, от която трябва да се вземе информацията (началната версия е 1); Ind и Ind1 са елементите, който съответно трябва да се обединят в едно множество или, за които трябва да се провери дали принадлежат в едно и също множество.

В решението използваме структурата Array, която се състои от целите числа val и Nodes, които са съответно номерът на бащата на дадения елемент и броя елементи, които съдържа поддървото на текущия елемент. Освен тях, структурата съдържа и указатели към левия и десния наследник на върха. В масива от указатели Roots, запазваме корена на всяка нова версия на персистентния ни масив.

Първото нещо, което трябва да направим, след като си въведем броя елементи е да си построим дървото. Функцията, която построява дървото ни, е същата рекурсия, която ползвахме за индексните дървета. Различното в имплементацията е, че запазваме първата версия на персистентната ни структура на втори индекс. Това го правим с цел улеснение. Тъй като ще ни се наложи да обновяваме два елемента в масива, то последната версия на всяко обновяване ще се намира на четен индекс.

Нека преди да продължим с описанието на главната ни функция, разгледаме функциите Find и Update. Решението се базира на тези две подпрограми. Първо ще разгледаме функцията Update. Тя е много подобна на функцията Update, която написахме при индексните дървета. Естествено, разликата е при обновяването на листата. Тъй като от main се вика функцията Update с две различни предназначения, ние различаваме какво се иска от нас чрез булевата променлива l. Ако променливата има стойност 0, то тогава ние обновяваме корена на новото дърво (тоест променяме броя на върховете в поддървото на корена). Ако променливата има стойност 1, то тогава ние обновяваме върха, който е бил корен преди да го обединим. За да го обновим присвояваме стойност на val - индекса на корена му. Функцията има логаритмична сложност. Както доказахме при индексните дървета, пътя от корена, до което и да е листо отнема точно $\log_2 N$ операции.

Целта на функция Find е да намира представителя на множеството на даден връх. Тя използва следния алгоритъм:

1. Ако се намираме в листото, което търсим, проверяваме дали не сме в представителя на множеството. Ако сме в него, приключваме рекурсията, като връщаме индекса на листото, в което се намираме. В противен случай, извикваме функция Find за родителя на върха ни (т.е. за Node->val).

2. Ако не се намираме в листо, то тогава избираме дали да тръгнем в лявото или дясното поддърво. Връщаме резултат - отговора от извикването на функция Find за избраното от нас поддърво.

Сега ще докажем, че за най-много $\log_2 N$ операции може да стигнем, от който и да е връх до корена (т.е., че няма да съществува дърво с N върха, което да има дълбочина повече от $\log_2 N$).

Ако имаме дървото K, то нека върховете в него да бъдат V(K), а дълбочината му да бъде D(K); Нека докажем с индукция следното твърдение:

За всяко дърво K е в сила, че $V(K) >= 2^{D(K)-1}$;

Нека допуснем, че това е вярно в даден и момент и нека обединяваме дърветата A и B, образувайки ново дърво C;

$$V(C) = V(A) + V(B);$$

Нека V(A)>=V(B); Тъй като винаги „закачаме“ дървото с по-малко върхове към това с повече, то следва, че ще образуваме дървото C като към дървото A ще „прикачим“ дървото B. Имаме два случая за дълбочината на C:

1) $\boldsymbol{D(C) = D(A)}$;

$V(A) + V(B) >= 2^{D(A)-1} + 2^{D(B)-1;}$
$V(C) >= 2^{D(A)-1} + 2^{D(B)-1}$;
но D(C) = D(A), следователно неравенството е изпълнено, защото D(B)<=D(A).
$V(C) >= 2^{D(C)-1;}$

2) $\boldsymbol{D(C) = D(B) + 1}$;

Искаме да докажем следното:
$V(C) >= 2^{D(C)-1}$;
$V(A) + V(B) >= 2^{D(B)}$;
Понеже V(A) >= V(B) е достатъчно да докажем по-силното твърдение:
$2*V(B) >= 2^{D(B)}$;
Но ние знаем, че:
$V(B) >= 2^{D(B)-1}$;
$2*V(B) >= 2^{D(B)-1} *2$;
$2*V(B) >= 2^{D(B)}$;

Следователно неравенството се запазва в дървото C;

Началните стойности на всички дървета са такива, че V()=D()=1, следователно неравенството има вида:

$V(K) >= 2^{D(K)-1}$; $1 >= 2^0$; $1 >= 1$;

По индукция следва, че е изпълнено във всеки един момент от алгоритъма. Следователно всички дървета имат $\log_2 N$ дълбочина, от където идва и логаритмичната сложност на самия алгоритъм, защото пътя от произволен връх до корена отнема най-много $\log_2 N$ операции.

Сложността на цялата функция е $O(log_2 N * log_2 N)$, защото търсенето на листото, което ни трябва ни отнема $\log_2 N$ операции. За всеки връх извикваме функцията най-много $\log_2 N$ пъти, както доказахме по-горе.

След като изяснихме всичко по работата на функциите, остава единствено да продължим обясненията си по main. След като сме построили дървото, започваме да отговаряме на заявки.

Ако заявката е от първи тип, то трябва първо да намерим представителите на дадените множества. Следователно извикваме функция Find за двата елемента, които търсим и запазваме отговорите съответно в Ans и Ans1. Ако елементите вече принадлежат в едно множество, тогава не трябва да правим нищо, освен да кажем, че съществува дърво, чийто корен е еднакъв с този на $State * 2$ и приключваме заявката. В противен случай разменяме Ans и Ans1, така че Ans да е родителя с повече върхове в поддървото си. Извикваме функция Update, като казваме, че ще променим представителя на цялото множество, следователно върховете в множеството му ще станат Ans->Nodes + Ans1->Nodes. Коренът, по който извършваме търсенето е $State * 2$ а не State, защото за всяко обединение на две множества ние променяме два елемента в масива ни, следователно създаваме две нови дървета. Следващото дърво, което създаваме използва предишното, като за него пускаме функцията Update, която търси представителя на множеството с по-малко върхове. Въпреки че във val съхраняваме индекса на родителя на даден връх, то в случая с Ans1->val, там ще се съдържа индекса на Ans1, тъй като той е представителят на множеството.

Ако заявката е от втори тип, то тогава трябва единствено да пуснем функция Find за дадените ни елементи и да сравним родителите на множествата им. Ако принадлежат в едно множество, тогава извеждаме "Yes". В противен случай извеждаме „No".

Сложността и на двете заявки е $O(\ log_2 N * log_2 N)$, което прави сложност на цялата задача за Q заявки $O(\ Q * log_2 N * log_2 N)$ по време и по памет. Оказва се, че решението е по-бързо от решение със сложност $O(N^2)$ откъм време за изпълнение, но най-голямото му предимство е спестената памет. Решенията, които имат $O(N^2)$ сложност по памет работят до към 5000-10000 ограничение за N, докато нашето решение може да работи за N=1000000.

Union-Find е структура, която има голямо приложение в компютърните науки. Ние успяхме ефективно да реализираме персистентната и форма, като запазихме свойството на персистентните структури да имат ниска сложност по памет и по време.

8. Интервално дърво[4]

8.1. Обикновено интервално дърво

Нека отново разгледаме задача:

Дадени са N на брой водни басейна. Всеки воден басейн се характеризира с нивото на водата в него. Нивото на водата може да е положително число – ако е над нормата, може да е отрицателно – ако е под нормата и може да е 0 ако е точно, колкото е и нормата за дадения воден басейн. Искаме да изпълняваме заявки от типа:

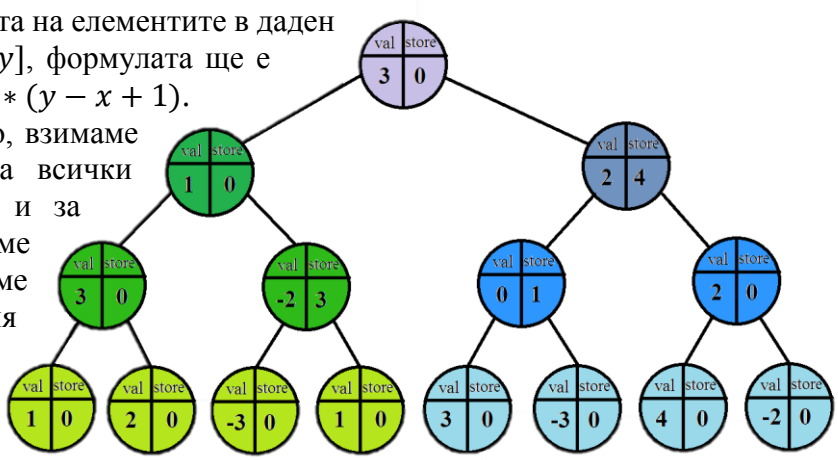
1. В интервала [L;R] нивото на водата се е покачило с k. Числото k може да е отрицателно число, ако поради засушаване, нивото на водата е спаднало.
2. Да се намери колко е сбора от нивата на водата в интервала [L;R].

[4] Тук трябва да отбележим, че наименованието „интервално дърво" е условно, тъй като няма точно определение за различните типове дървета в българските източници. Под интервално дърво ще разбираме дървото, върху което се използва техниката Lazy Propagation.

Задачата звучи и се решава подобно на задачата, която решихме с индексното дърво. Разликата в този случай е, че не обновяваме само едно число, а цял интервал от числа. За да решим задачата ще използваме техниката Lazy Propagation.

Техниката Lazy Propagation използва решението на индексните дървета като основа за следващите си разсъждения. Наблюдението, което трябва да направим в този случай е, че няма нужда целият интервал да се обновява веднага след пристигането на заявката. По такъв начин можем да натрупваме заявки за обновяване на интервал, докато стойността му не е необходима.

За всеки връх ще пазим променливите val и store, които ще са съответно стойността на интервала в дървото (т.е. сумата на всички числа, за които отговаря интервала) и с колко е била увеличена/намалена стойността на всяко число от даденият интервал. Ако сме увеличили стойностите в един интервал няколко пъти, промените в store ще се натрупват. Следователно ако сме имали заявки за увеличаване на стойностите в интервала $[1; 4]$ три пъти, със съответно 5, -3 и 2, то тогава store-а на върха, който отговаря за този интервал ще бъде 5-3+2=4. Ако искаме да изчислим сумата на елементите в даден интервал от дървото $[x; y]$, формулата ще е следната: $val + store * (y - x + 1)$. Или с други думи казано, взимаме първоначалната сума на всички елементи от интервала и за всеки елемент добавяме толкова, колкото сме увеличили целия интервал. Тук трябва да отбележим, че след като сме задали нова стойност на даден интервал, обновяваме само върховете, които са по пътя между корена и върха, който отговаря за интервала. Следователно, няма да променяме върховете, които са в поддървото на върха ни. Тъй като ще пазим дървото чрез статична памет, няма да са нужни други променливи в структурата. Съхранението на дървото е идентично с това, което правихме в наивното решение за индексните дървета.



Нека разгледаме по-детайлно изпълнението на отделните заявки. Ще започнем със заявката за обновяване на стойностите в даден интервал. Функцията изпълнява следния алгоритъм:

1. Ако сме изцяло извън интервала, който търсим, прекратяваме изпълнението на текущата функция.
2. Ако сме изцяло в интервала, който търсим, увеличаваме store-а на върха, в който сме с NewVal, а val с $NewVal * (y - x + 1)$ и прекратяваме рекурсията.
3. В противен случай пускаме рекурсии за левия и за десния син на върха ни и след тяхното приключване обновяваме val. Стойността на текущия връх ще е равна на сумата от стойностите на левия и на десния му син $+ \ store * (y - x + 1)$.

Сложността на функцията е $O(log_2 N)$.

Втората функция трябва да намира сумата на всички числа в интервала $[L; R]$. Тя подържа през цялото време променлива Add, която показва колко трябва да се добави към сумата на текущия интервал (т.е. Add ще бъде сума на 0, 1 или няколко store-а на върхове, които се намират по пътя от корена до текущия връх). Подпрограмата използва подобно решение като при индексните дървета. Алгоритъмът е следният:

1. Ако сме изцяло извън интервала, който търсим, прекратяваме изпълнението на текущата рекурсия.
2. Ако сме изцяло в интервала, който търсим, прибавяме към отговора следния израз: $Add * (y - x + 1) + val$ и приключваме рекурсията.
3. В противен случай увеличаваме стойността на Add със store-а на текущия връх. Пускаме рекурсии за левия и за десния син на върха ни. След приключването им, намаляваме[5] Add със същата стойност, с която го увеличихме по-рано.

Сложността на тази функция е $O(log_2 N)$.

Сложността и на двете функции е логаритмична, следователно запазваме сложността от индексното дърво. Лесно може да се докаже, че със същата сложност на строене на дървото не може да съществува константно отговаряне на заявки. Ако подобно дърво съществуваше, щеше да е възможно сортирането с линейна сложност[6]. Следователно, интервалното ни дърво е оптимално.

Интервалните дървета са много приложими. Те се използват в торент сайтовете, както и в сайтове като youtube. Изключителната им бързина и способност да се пригаждат към различни типове задачи ги прави много използвани. Много от състезателните задачи използват интервалните дървета, а на последния есенен турнир имаше задача, която се решаваше с персистентно такова. Нека разгледаме подобна такава.

8.2. Персистентно интервално дърво

Дадени са N на брой водни басейна. Всеки воден басейн се характеризира с нивото на водата в него. Нивото на водата може да е положително число – ако е над нормата, може да е отрицателно – ако е под нормата и може да е 0 ако е точно, колкото е и нормата за дадения воден басейн. Искаме да изпълняваме заявки от типа:

1. Какво би станало, ако във версия p в интервала [L;R] нивото на водата се покачи с v. Числото v може да е отрицателно число, ако поради засушаване, нивото на водата е спаднало.
2. Да се намери колко е сбора от нивата на водата в интервала [L;R] във версия p.

Тъй като разглеждаме няколко версии на водните басейни и може да създаваме версия от състояние, което не е последното, то става ясно, че се изисква от нас да имплементираме пълна персистентна структура.

След като имаме ефимерното интервално дърво можем да направим и персистентната версия. Ще използваме същата логика като при индексните дървета,

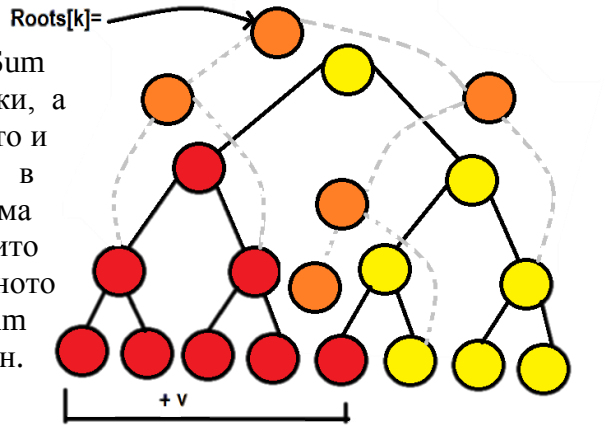
[5] Правим това, защото Add е глобална променлива. Ако се подаваше по параметър нямаше да е нужно.

[6] Все още не е открит алгоритъм за сортиране, който да сортира елементи с линейна сложност.

където имахме допълнителна памет в най-лошия случай $O(log_2 N)$. Следователно, ще запазваме само елементите, които променяме в дървото. Както доказахме по-горе, можем да представим интервал като множество от интервали за $log_2 N$ стъпки. Т.е. променяме най-много $log_2 N$ елемента, което показва, че може да запазим и сложността по памет от индексните дървета. Сега нека разгледаме входът, който се задава.

От първия ред на стандартния вход се задават две числа – N и Q, съответно брой върхове в дървото и брой заявки. Следват Q реда, всеки от които съдържа по 4 или 5 числа - Type, State, L, R. Това са съответно: тип на заявката; версия, от която строим новото дърво; ляв и десен край на интервал; Ако Type=1, се въвежда и v, което показва с колко трябва да се увеличи всяко число от интервала. След като имаме входните данни, нека опишем алгоритъма.

Започваме отново с построяването на дървото. Дървото се състои от val, store, Left, Right. Val и store имат същото предназначение като това, което описахме при обикновеното интервално дърво. Left и Right са съответно указатели към левия и към десния наследник на върха. За листата тези указатели имат стойност $NULL$. Използваме динамично заделяне на памет, като докато създаваме дървото, задаваме начална стойност за всеки негов връх. Отново имаме масива Roots, който пази корените на различните версии.



Функциите Update и FindSum изпълняват исканите от условието заявки, а имено обновяване на елементите в дървото и намиране сумата на всички елементи в интервал. Тези функции приличат до голяма степен на функциите, които имплементирахме при ефимерното интервално дърво, като функцията FindSum работи по абсолютно аналогичен начин. Сега ще опишем функцията Update:

1. Ако сме изцяло в интервала, който търсим, увеличаваме store-а на върха, в който сме с v, а val с $v * (y - x + 1)$. Свързваме левия и десния наследник на текущия връх със съответно левия и десния наследник в старата версия и прекратяваме рекурсията.
2. Ако търсеният интервал не споделя дори и една обща част с интервала на левия наследник на текущия връх, то тогава насочваме указателя Left към левия наследник на базисната ни структура. В противен случай пускаме функцията Update за левите наследници на текущия връх и базисната структура.
3. Ако търсеният интервал не споделя дори и една обща част с интервала на десния наследник на текущия връх, то тогава насочваме указателя Right към десния наследник на базисната ни структура. В противен случай пускаме функцията Update за десните наследници на текущия връх и базисната структура.
4. Стойността на текущия връх ще е равна на сумата от стойностите на левия и на десния му син $+ store * (y - x + 1)$.

Сложността и на двете заявки е $O(log_2 N)$. Следователно сложността по памет и време за Q заявки ще бъде – $O(Q * log_2 N)$.

Успяхме да реализираме ефективно персистентно интервално дърво. Сложността ни е оптимална.

9. Заключение

Ние разгледахме свойството на пълна персистентност при няколко структури. Установихме голямата приложност на персистентните структури и плануваме да допълним текущата си разработка, като я разширим чрез реализации на пълна персистентност при по-сложни структури от данни, подобни на Heap, Trie, AVL-Tree, Red-Black Tree и други. Плануваме да разработим и материали отнасящи се към Partial persistence, Confluent persistence и Functional persistence. По темата за персистентност, като цяло, няма създадени български разработки. Всички източници, които намерихме бяха на английски език. Целта ни е да разпространим персистентните структури от данни в българските информатични среди. За целта се надяваме разработките ни да станат достъпни до по-голям кръг от хора и се надяваме да бъдем полезни както на учениците, така и на техните преподаватели.

10. Благодарности

Искаме да изкажем специални благодарности към:

Енчо Мишинев - за неговата неотлъчна подкрепа – и тялом и духом.

Коля Петрова – за насоките, с които ни помогна.

Паолина Гаджулова – за съветите, които получихме.

Приложения:

В CD прилагаме:

1. Реализация на персистентно индексно дърво – Persistent_IndexTree;
2. Реализация на неефективно персистентно индексно дърво – Naïve_PersistentIndex;
3. Реализация на генератор за тестове за индексно дърво – TestGenIndex;
4. Реализация на персистентен масив– Persistent_Array;
5. Реализация на персистентен Union-Find – Persistent_UnionFind;
6. Реализация на неефективен Union-Find – Naïve_PersistentUnionFind;
7. Реализация на генератор за тестове за Union-Find – TestGenUnionFind;
8. Реализация на персистентен свързан списък – Persistent_List;
9. Реализация на персистентен стек – Persistent_Stack;
10. Реализация на персистентно интервално дърво – Persistent_LazyPropagation

Използвана литература:

http://blog.anudeep2011.com/persistent-segment-trees-explained-with-spoj-problems/

http://www.codeproject.com/Articles/9680/Persistent-Data-Structures#_comments

http://bartoszmilewski.com/2013/11/13/functional-data-structures-in-c-lists/

https://www.lri.fr/~filliatr/ftp/publis/puf-wml07.pdf

http://www.math.bas.bg/infos/files/2014-11-23-A1-sol.pdf