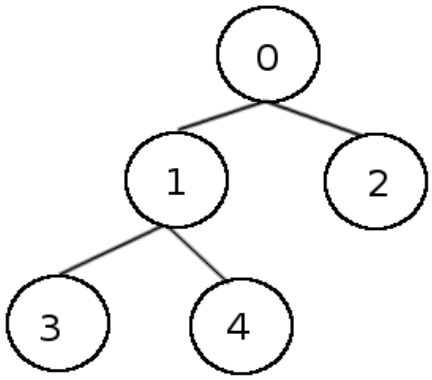
АНАЛИЗ НА РЕШЕНИЕТО НА ЗАДАЧА ФИРМА

Подзадача 1:

Ограничението на първата подзадача ни подсказва, че очакваното решение извършва брой операции, който е линеен по броя върхове. Това е лесно: поддържаме масив par[x], който съдържа прекия предшественик (родител) на върха x. Добавянето на нов връх променя стойностите на par[] за най-много два върха. Отговарянето на заявка is_par(x, y) се извършва като вземем върха y и вървим „нагоре“ по дървото, използвайки масива par. Ако стигнем върха x, отговаряме **true**, а ако достигнем корена преди това, отговаряме **false.**

Подзадача 2:

Да разгледаме примерно дърво на йерархия:



Ако направим обхождане в дълбочина на това дърво ще посетим върховете в ред 0, 1, 3, 4, 2.

Нека направим обхождане в дълбочина на дървото като записваме всеки връх по два пъти: когато го видим за първи и за последен път.

Списъка ни с върхове ще изглежда така:

0IN, 1IN, 3IN, 3OUT, 4IN, 4OUT, 1OUT, 2IN, 2OUT, 0OUT
Всеки връх фигурира в списъка два пъти. Нещо повече, всички върхове в поддървото на някой връх X се срещат само между XIN и XOUT в списъка. Например 3IN, 4IN, 3OUT, 4OUT се срещат между 1IN и 1OUT, тъй като 3 и 4 са в поддървото на връх 1.

Тоест, за да отговорим на заявка **is_par(x, y)** е достатъчно да проверим дали **yIN** е между **xIN** и **xOUT** в списъка. Добавяне на листо с номер **n** при операция **add_leaf(p)** е лесно: достатъчно е да добавим два върха **nIN** и **nOUT** непосредствено след **pIN**.

Например, ако извикаме add_leaf(2) на по-горното дърво ще добавим връх номер 5 и списъка с върхове ще изглежда:

0IN, 1IN, 3IN, 3OUT, 4IN, 4OUT, 1OUT, 2IN, 5IN, 5OUT, 2OUT, 0OUT

Ако бяхме извикали **add_above(d)**, която добавя връх с номер **n** като баща на **d**, е достатъчно да добавим **nIN** точно преди **dIN**, и **nOUT** точно след **dOUT.** Отново, това са две операции върху списъка с върхове.

Например ако извикаме add_above(2):

0IN, 1IN, 3IN, 3OUT, 4IN, 4OUT, 1OUT, 5IN, 2IN, 2OUT, 5OUT, 0OUT

И двете операции запазват исканите свойства на списъка с върхове. В каква структура можем да държим списъка с IN- OUT- върхове?

Тривиална имплементация със свързан списък, би довела до константни заявки add_above, add_leaf, но линейно търсене за всяка заявка is_par. Ако използвахме масив за това, заявката is_par би била с константно време, но add_node и add_leaf – линейно, защото би трябвало да пренареждат списъка.

Структурата, в която ще държим списъка на върхове, трябва да поддържа добавяне на елемент на конкретна позиция и запитвания: дали елемент x е преди елемент y в списъка. Например може да използваме подобрен свързан списък, който е разбит на блокове. Ако имаме N елемента в свързания списък, големината на всички блокове ще е приблизително

sqrt(N). Когато добавяме елемент в списъка, го добавяме в блока, който съответства на съседа му отляво. Блоковете ще са строго подредени: всички елементи в блок номер A са преди всички елементи в блок номер B, тогава и само тогава, когато A < B. Заявките ни ще бъдат със сложност O(sqrt(N)): ако два елемента са в различни блокове, отговаряме константни, ако са в един и същи блок, сложността на заявката е sqrt(N): пак трябва да обходим всички елементи. Добавянето на елемент е с константна сложност. Като добавяме елементи, блоковете ни ще станат с неравномерна дължина. За да балансираме това, е достатъчно да преизчисляваме кой елемент на кой блок принадлежи през няколко операции. Например, на всеки sqrt(N) операции.

Това решение би било достатъчно да решим подзадача 2.

Подзадача 3:

Всичко, което е необходимо да решим подзадача 3 е по-добра структура от данни, която да поддържа двете ни заявки: добавяне на елемент на позиция, и отговор на въпрос дали един елемент е преди друг. Това е възможно с използване на наша имплементация на балансирано двоично дърво за търсене, в което да пазим колко елемента има в поддървото на всеки връх. Като имаме тази информация лесно можем да видим кой е номера на всеки елемент в списъка ни за log(N) време, и така да отговорим на is_par(x, y) заявките за логаритмично време.

Ако искаме да добавим елемент в структурата на определена позиция в списъка, правим търсене подобно на стандартния алгоритъм, за да установим позицията в дървото на новодобавения връх. Ако искаме да добавим елемент на позиция P в списъка, тръгваме да търсим позиция в дървото от корена. Ако в лявото поддърво на корена има повече от P върха, тогава ще добавим елемента в дясното поддърво на корена, и трябва да намалим P с броят на върховете в лявото поддърво на корена. В противен случай, добавяме елемента в лявото поддърво на корена. Рекурсивно следвайки тази процедура, намираме къде трябва да добавим новия елемент.

Не навлизаме в детайли относно имплементацията на подзадача 3, защото съществуват много имплементации на двоични дървета за търсене и структури, които биха свършили работа за задачата. Например:

http://en.wikipedia.org/wiki/Treap
http://en.wikipedia.org/wiki/AVL_tree
http://en.wikipedia.org/wiki/Red%E2%80%93black_tree

2-3-4 tree http://en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree

Автори: Георги Георгиев, Йордан Чапъров