ДЪРВЕТА

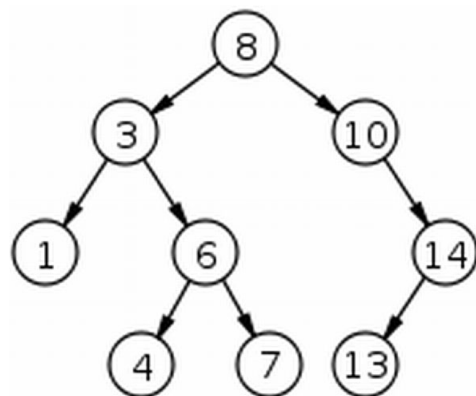
В информатиката често се използва структура от данни, наричана ***дърво*** (в математиката се предпочита наименованието ***кореново дърво***). Един връх има специална роля — нарича се ***корен*** и служи за представител на дървото като цяло. Всеки ***връх*** (***възел***), вкл. коренът, съдържа някаква информация: данни на потребителя плюс служебни данни. Служебните данни се състоят от списък на поддърветата, тоест списък на преките ***наследници*** (може празен), и / или указател към (единствения) ***родител*** (като само коренът няма родител). Броят на възможните преки наследници може да не е ограничен предварително, но най-често е ограничен от някое малко число. Дърво, чиито върхове имат най-много по два преки наследника, се нарича ***двоично дърво***; двата наследника се наричат ляв и десен (дори когато някой връх има само един пряк наследник, се уточнява дали наследникът е ляв, или десен).

Върховете, които нямат наследници, се наричат ***листа***. Останалите върхове се наричат ***вътрешни върхове*** на дървото.

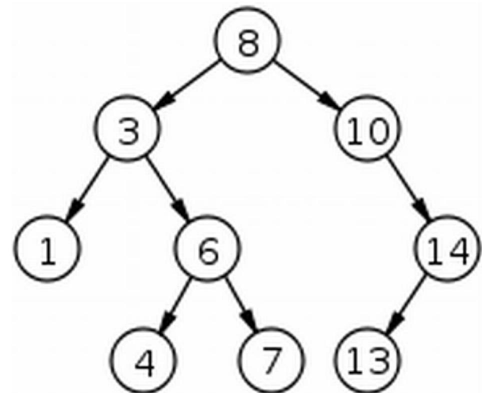
Всяко дърво се представя чрез своя корен (или чрез указател към корена). За удобство се допуска дървото да е празно, тоест да няма върхове, вкл. корен. ***Празно дърво*** се представя с нулев указател — такъв, който не сочи наникъде. Листата на двоично дърво имат празни поддървета за ляв и десен наследник.

Освен това всеки връх на дървото съдържа някакви данни на потребителя, нужни за решаване на конкретна задача. Тези данни могат да са организирани по произволен начин, но най-често се оформят като ***записи***, тоест съвкупности от именувани ***полета***. Полетата могат да бъдат разнотипни. Едно от полетата на записа обикновено играе по-особена роля и се нарича ***ключ***. Ключът служи за идентифициране на записа и се използва при търсене и изтриване на връх. Останалите полета обикновено се обединяват в подзапис, наречен ***стойност***, тоест потребителската информация във всеки връх на дървото представлява наредена двойка <***ключ***; ***стойност***>. В текста по-нататък ще бъдат показвани само ключовете (други стойности може да няма поначало).

Всички дървета са графи, затова могат да се обхождат в дълбочина (клон по клон) или в ширина (слой по слой, в зависимост от разстоянията си от корена на дървото). Преките наследници на един и същи връх се обхождат в реда, в който са подредени в списъка на преките наследници на върха (за двоично дърво е прието по традиция, че левият наследник се обхожда преди десния). Например върховете на показаното дърво при ***обхождане в ширина*** ще се отпечатат в следния ред: 8; 3; 10; 1; 6; 14; 4; 7; 13.



При ***обхождане в дълбочина*** всеки връх може да се обработва (напр. да се отпечатва) преди или след обхождането на поддървото, чийто корен се явява, или в междинен миг. Обхождането в дълбочина се дели съответно на три вида: ***право***, ***обратно*** и ***центрирано*** (центрирано обхождане се прилага най-вече за двоични дървета, иначе е нужно уточнение кога именно се обработва коренът).



Право обхождане: всеки от върховете се обработва в момента на отваряне, тоест преди наследниците си. Например дървото, изобразено вдясно, отпечатва върховете си в следния ред: 8; 3; 1; 6; 4; 7; 10; 14; 13.

Обратно обхождане: всеки връх се обработва в момента на затваряне, тоест след наследниците си. Например върховете на дървото по-горе ще бъдат отпечатани в следния ред: 1; 4; 7; 6; 3; 13; 14; 10; 8.

Центрирано обхождане на двоично дърво: всеки от върховете се обработва след обхождане на лявото поддърво, но преди обхождане на дясното поддърво. Например върховете на дървото от картинката ще се отпечатат в следния ред: 1; 3; 4; 6; 7; 8; 10; 13; 14.

Всички видове обхождане на дърво притежават линейна времева сложност, т.е. $\Theta(n)$, при всякакви входни данни, където n е броят на върховете на дървото.

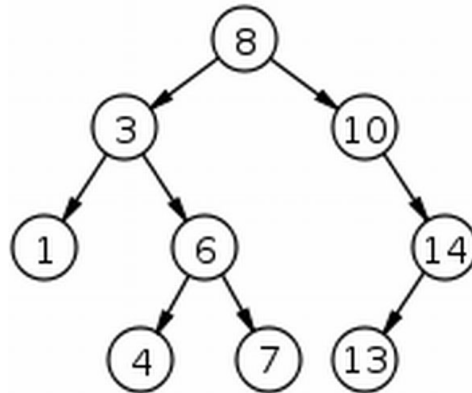
Височина h на дърво се нарича дължината (мерена с броя на ребрата) на най-дълъг път от корена до листо. Ако дърво съдържа само корен, то $h = 0$. За празното дърво приемаме по определение, че $h = -1$. За непразно дърво важи двойното неравенство $0 \leq h \leq n - 1$.

Едно дърво се нарича ***балансирано***, ако разстоянията от всичките му листа до корена се различават най-много с единица. Някои операции върху дървета имат времева сложност $\Theta(h)$, затова те са по-бързи, когато се извършват върху балансирани дървета, но самото балансиране отнема много време, затова изискването може да се отслаби по подходящ начин. За балансирани дървета (дори при по-слабото изискване да бъде ограничено отгоре от константа частното от дължините на пътищата от корен до листо), ако върховете, които не са листа, имат еднакъв брой преки наследници, то $h = \Theta(\log n)$, затова операциите с времева сложност $\Theta(h)$ са доста бързи. Обратно, ако едно дърво е твърде небалансирано, тогава h нараства и тези операции стават по-бавни. Крайният случай е, когато дървото се изражда в списък; тогава $h = n - 1 = \Theta(n)$; такова дърво няма никакво предимство пред свързания списък, затова е важно да се поддържа балансът.

Където е необходима наредба, ще предполагаме, че типът на ключовете е нареден (в смисъл на пълна, тоест линейна, наредба).

Двоично дърво за търсене

Двоично дърво за търсене или ***наредено двоично дърво*** се нарича такова двоично дърво, в което ключът на всеки връх не е по-малък от никой ключ в лявото му поддърво и не е по-голям от никой ключ в дясното му поддърво.



Центрираното обхождане на двоично дърво за търсене отпечатва ключовете в ненамаляващ ред. Например ключовете на това двоично дърво за търсене се отпечатват при центрираното му обхождане, образувайки следната редица: $1 \leq 3 \leq 4 \leq 6 \leq 7 \leq 8 \leq 10 \leq 13 \leq 14$.

Двете най-използвани структури от данни — масивът и свързаният списък, имат този недостатък, че от двете операции търсене и вмъкване на елемент едната винаги се извършва бавно. По-конкретно, търсенето в сортиран масив с n елемента (двоичното търсене) изисква време $\Theta(\log n)$, обаче операцията вмъкване на елемент се извършва за време $\Theta(n)$ при най-лоши входни данни: когато новият елемент е малък, трябва да бъде вмъкнат в началото на масива и тогава се налага да бъдат избутани с една позиция всички останали елементи. Обратно, вмъкването на елемент в свързан списък става за време $\Theta(1)$, обаче двоичното търсене е неприложимо, а последователното търсене изразходва време $\Theta(n)$ в най-лошия случай — когато търсеният елемент липсва в списъка или се намира на последното място. Ако тези две операции се изпълняват приблизително равен брой пъти, то времето на по-бавната от тях определя времето на алгоритъма, който ги използва. Следователно би било полезно, ако по-бавната операция се ускори, дори това да забави по-бързата операция.

Двоичното дърво за търсене предлага решение на този проблем, постигайки време $\Theta(h)$ при най-лоши входни данни; тази оценка важи за всички операции. Когато височината на дървото е много по-малка от броя на неговите елементи, тоест когато дървото е поне приблизително балансирано, използването му значително ускорява алгоритмите.

Операции върху двоично дърво за търсене:

— ***Търсене по ключ:***

1) Ако дървото е празно,
то не съдържа търсения ключ (край на търсенето).
2) В противен случай търсеният ключ се сравнява с ключа на корена.
   а) Ако ключовете са равни, то търсенето завършва успешно:
   търсеният връх е коренът на дървото (край на търсенето).
   б) Ако търсеният ключ е по-малък от ключа на корена,
   то търсенето продължава в лявото поддърво.
   в) Ако търсеният ключ е по-голям от ключа на корена,
   то търсенето продължава в дясното поддърво.

— ***Търсене на най-голям (най-малък) елемент:***

Търсенето започва от корена и се спуска само надясно (само наляво),
докато стигне до листо. Това листо е търсеният екстремален елемент.

— ***Търсене на следващ по-голям (по-малък) елемент:***

1) Ако даденият елемент има непразно дясно (ляво) поддърво,
търси се най-малкият (най-големият) елемент в това поддърво
и алгоритъмът приключва с този елемент като резултат.
2) В противен случай търсенето се изкачва от родител към родител,
докато стигне до корена или докато мине по ляво (дясно) ребро.
3) Ако търсенето се изкачи към връх по ребро от търсения вид:
   а) Ако другото поддърво на достигнатия връх е празно,
   то търсенето приключва с резултат достигнатия връх.
   б) В противен случай търсенето се спуска по дясно (ляво) ребро,
   а после — само по леви (десни) ребра, докато стигне до листо.
   Търсенето приключва с резултат достигнатото листо.
4) Ако при изкачването не се намери ребро от търсения вид,
то търсенето приключва с отрицателен резултат,
тоест няма по-голям (по-малък) елемент.

— ***Вмъкване:***

1) Ако дървото е празно,
то се създава дърво с един връх (корен без наследници)
и дадената потребителска информация се копира в неговата,
с което алгоритъмът приключва работа.
2) В противен случай новият ключ се сравнява с ключа на корена.
   а) Ако ключовете са равни, то дадената потребителска информация
   се копира на мястото на тази от корена и алгоритъмът приключва.
   б) Ако новият ключ е по-малък от ключа на корена,
   то търсенето на място за вмъкване продължава в лявото поддърво.
   в) Ако новият ключ е по-голям от ключа на корена,
   то търсенето на място за вмъкване продължава в дясното поддърво.

— ***Изтриване:***

Най-удобно е да бъде подаден на входа указател към указател към връх. (Това може да бъде също указател към указателя към корена на дървото.)

1) Ако върхът, подлежащ на изтриване, е листо, то паметта за него се освобождава, а указателят, сочен от указател, се заменя с празен (нулев) указател и алгоритъмът приключва. (Това включва случая на единствен връх; тогава дървото става празно.)

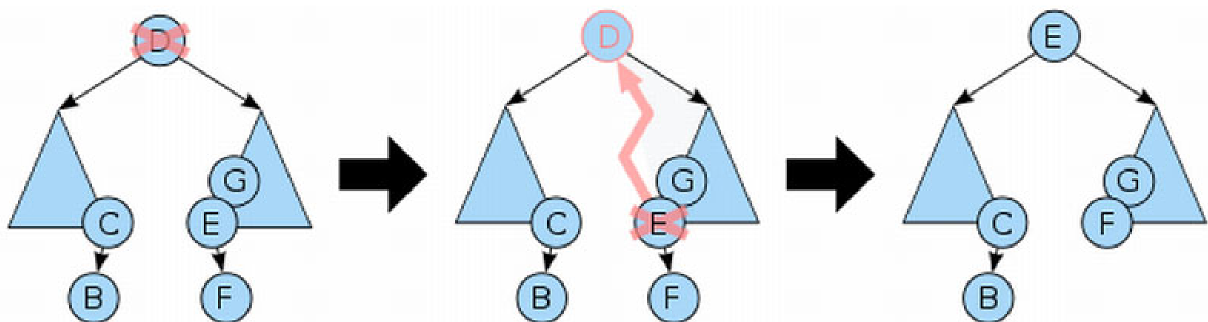
2) Ако върхът, подлежащ на изтриване, има само един пряк наследник, то паметта за върха, подлежащ на изтриване, се освобождава, а указателят към изтрития връх (от неговия родител) се пренасочва към споменатия единствен пряк наследник и алгоритъмът приключва.

3) Ако върхът D, подлежащ на изтриване, има два преки наследника, то първо се намира следващият по-голям елемент E. После:

а) Ако E е пряк (десен) наследник на D, то E се издига на мястото на D (заедно с цялото си дясно поддърво, което може да бъде и празно), а левият указател на E (който в този миг е нулев) се пренасочва към лявото поддърво на D (което може да бъде и празно). Паметта за излишния елемент се освобождава (край на алгоритъма).

б) Ако E е непряк наследник на D (от дясното поддърво на D), то потребителската информация от E заменя тази в D, а десният пряк наследник на E (ако съществува такъв) се издига на предишното място на елемента E. Излишната памет се освобождава.

Илюстрация на последния (най-трудния) случай от операцията изтриване:



Някои от изброените операции изискват изкачване от елемент към корена, тоест имат нужда от указатели към родителите. Такива указатели биха могли да се пазят допълнително в дървото или да се намерят еднократно чрез търсене на посочения елемент (по неговия ключ) и временно запомняне на пътя до него.

Ако е дадено двоично дърво за търсене с n върха и височина h, а се променя само вторият параметър на някоя от операциите (търсеният ключ или върхът, подлежащ на изтриване, и т.н.), то всички операции се изпълняват за време $\Theta(h)$ при най-лош втори параметър. Но ако дървото не е фиксирано, то за оценката трябва да се вземе най-лошият случай за h и времевата сложност става $\Theta(n)$, когато дървото се изражда в свързан списък.

В общия случай получените оценки не могат да се подобрят. Но докато програмистът няма значително влияние върху втория параметър на операциите (тъй като много често някоя сложна обработка на друга структура от данни започва с празно двоично дърво за търсене, попълва го с елементи, а после ги търси и изтрива в определен ред), то първият параметър (самото дърво) обикновено е фиксиран (един и същ) за множество поредни операции. Тогава може да се предприеме някаква оптимизация на структурата на дървото с цел ускоряване на следващите операции. Те са най-бързи при височина $h = \Theta(\log n)$, тоест при балансирано дърво.

Средната времева сложност на всички разгледани операции съвпада по порядък с минималната сложност, тоест $\Theta(\log n)$. Това следва от факта, че средната височина на двоично дърво за търсене с n върха е $\Theta(\log n)$. За целта се фиксират n произволни елемента и се разглеждат всички двоични дървета за търсене, които могат да се съставят от тях. Нека $T(n)$ е математическото очакване на височината на случайно двоично дърво за търсене, получено по следния начин.

Най-напред се избира коренът на дървото като един от дадените n елемента, изтеглен случайно и равновероятно. Другите елементи се разделят на два вида: по-малки и по-големи от корена (за простота се приема, че няма повторения). Чрез описаната процедура (избиране на нови и нови елементи) се построяват лявото и дясното поддърво, съответно от малките и големите елементи.

Поради равномерното разпределение на корена броят на малките елементи е равномерно разпределен в множеството на целите числа от 0 до $n - 1$ вкл. Затова функцията $T(n)$ удовлетворява следното рекурентно уравнение:

$$T(n) = 1 + \frac{1}{n} \sum_{m=0}^{n-1} \max \Big(T(m)\, ;\, T(n-m-1) \Big),$$

където m е броят на малките елементи. От смисъла на T е ясно, че тя е ненамаляваща функция, затова уравнението се преобразува така:

$$T(n) = 1 + \frac{2}{n} \Big(T(n-1) + T(n-2) + T(n-3) + \ldots + T((n-1)/2) \Big),$$

като аргументът на T в последното събираемо се закръгля нагоре при четно n, а при нечетно n това събираемо участва с коефициент $1/n$ вместо $2/n$. Оттук по индукция следва, че $T(n) \leq c \ln n$ за всяко естествено число $n > 1$, където c е константа, избрана подходящо (достатъчно голямо положително число).

Рекурентното уравнение важи за всички достатъчно големи цели числа n. Базата на индукцията се състои от останалите цели $n > 1$. Те са краен брой, затова неравенството $T(n) \leq c \ln n$ ще важи за всички тях, стига константата c да бъде избрано достатъчно голяма — по-голяма от всички частни $T(n) / \ln n$ (които също са краен брой).

Индуктивната стъпка се провежда за достатъчно големи цели числа n (тези, за които важи рекурентното уравнение). Използва се силна индукция.

Действително, от рекурентното уравнение и индуктивното предположение следва веригата от равенства и неравенства (първото неравенство замества равенството по-горе, обхващайки едновременно случаите на четно и нечетно n):

$$T(n) \le 1 + \frac{2}{n}\Big(T(n-1) + T(n-2) + T(n-3) + \ldots + T((n-1)/2)\Big) \le$$

$$\le 1 + \frac{2}{n}\Big(c\ln(n-1) + c\ln(n-2) + c\ln(n-3) + \ldots + c\ln((n-1)/2)\Big) \le$$

$$\le 1 + \frac{2c}{n}\int_{(n-1)/2+1}^{n} \ln x \, dx.$$ Последното неравенство следва от това, че $\ln x$ е растяща функция. След намаляване на долната граница интегралът нараства:

$$T(n) \le 1 + \frac{2c}{n}\int_{n/2}^{n} \ln x \, dx = 1 + \frac{2c}{n}\Big\{ [n\ln n - n] - [(n/2)\ln(n/2) - (n/2)] \Big\} =$$

$$= 1 + \frac{2c}{n}\Big[(n/2)\ln n + (n/2)\ln 2 - (n/2)\Big] = 1 + c\,\Big[\ln n + \ln 2 - 1\Big].$$ Дясната страна след подреждане на събираемите по асимптотичен порядък приема следния вид:

$T(n) \le 1 + c\,\Big[\ln n + \ln 2 - 1\Big] = c\ln n + \Big[1 - c\,(1 - \ln 2)\Big] \le c\ln n$, при условие че свободният член е отрицателен, тоест $c > 1/(1 - \ln 2)$, а това неравенство важи за всички достатъчно големи числа c.

Доказаната горна граница означава, че $T(n) = \mathrm{O}(\log n)$.

От друга страна, за произволно двоично дърво с n върха и височина h важи неравенството $n \le 1 + 2 + 4 + 8 + 16 + \ldots + 2^{h} = 2^{h+1} - 1 \le 2^{h+1}$. Ето защо $h + 1 \ge (\ln n) / (\ln 2)$, тоест $h \ge (\ln n) / (\ln 2) - 1$. Това неравенство важи за всяка височина h, вкл. минималната височина $T(n)$. Ето защо $T(n) = \Omega(\log n)$.

Долната граница в съчетание с горната води до извода, че $T(n) = \Theta(\log n)$. Това е средната височина на двоично дърво с n върха и средната сложност на разгледаните по-горе операции върху наредено двоично дърво.

Приложения на двоичното дърво за търсене:

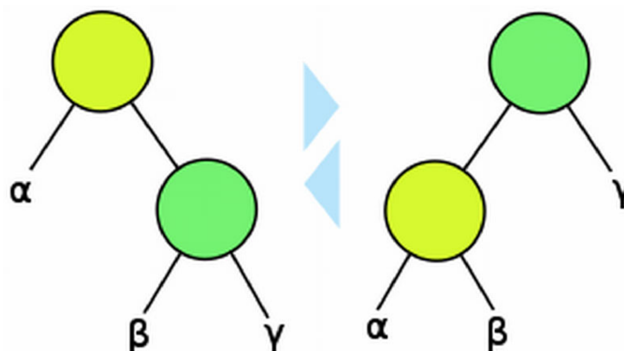
— ***Сортиране:*** Елементите, които трябва да се подредят в нарастващ ред, общо n на брой, се вмъкват в двоично дърво за търсене (отначало празно), което после се обхожда центрирано. Средната времева сложност е $\Theta(n \log n)$, максималната е $\Theta(n^2)$. По същество този алгоритъм е т.нар. бързо сортиране.

— ***Крайни автомати:*** Преходите, излизащи от едно състояние на автомат, могат да бъдат пазени в масив от d указателя, където d е размерът на азбуката. Ако от всеки връх излизат k прехода, където $k \ll d$, много указатели са нулеви. Употребата на наредено двоично дърво вместо масив пести памет (пазят се k вместо d указателя), но забавя търсенето (времето нараства от $\Theta(1)$ на $\Omega(\log k)$).

Самобалансиращо се двоично дърво за търсене

Самобалансиращо се двоично дърво за търсене (наричано по друг начин ***самобалансиращо се наредено двоично дърво***) е двоично дърво за търсене, автоматично поддържащо малка височина въпреки добавянето и изтриването на елементи. Повечето (но не всички) самобалансиращи се двоични дървета поддържат височина $O(\log n)$, където n е броят на върховете им; по този начин се осигурява бързодействието на разгледаните по-горе разнообразни операции.

Най-често използваната операция е завъртането на дърво. Обикновено тя се прилага към поддървета и води до промяна на техните височини с 1. Завъртането, независимо от посоката, има константна времева сложност и запазва реда на върховете на дървото при центрирано обхождане. Посоката на завъртането трябва да се избере правилно, та завъртането да намали разликата на височините.



Обикновено не е нужно идеално балансиране (тоест не е нужно разликата на дължините на различните пътища от корена до листата да не надхвърля 1); за гарантиране на оценката $h = \Theta(\log n)$ и оттам — на бързината на операциите е достатъчно частното от дължините на пътищата от корена до листата да бъде ограничено отгоре от някаква константа $c \geq 1$ и върховете, които не са листа, да имат един и същи брой преки наследници.

Действително, ако най-дълъг и най-къс път от корена до листата притежават съответно дължини h и h / c, то всички слоеве на дървото до № h / c вкл. са запълнени, затова важи двойното неравенство

$$1 + d + d^2 + \ldots + d^{h/c} \leq n \leq 1 + d + d^2 + \ldots + d^h,$$

тоест

$$(d^{(h/c)+1} - 1) / (d - 1) \leq n \leq (d^{h+1} - 1) / (d - 1),$$

където n е броят на всички върхове, а $d \geq 2$ е броят на преките наследници на всеки връх, който не е листо.

След като се реши относно h, двойното неравенство приема следния вид:

$$-1 + \log_d\big((d-1)\,n + 1\big) \leq h \leq c\big(-1 + \log_d\big((d-1)\,n + 1\big)\big).$$

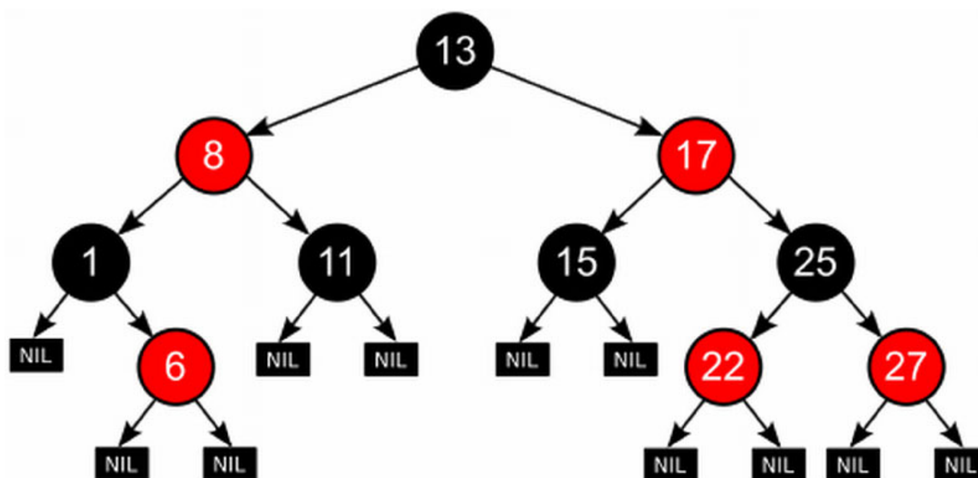
Тъй като лявата и дясната страна са от порядък $\log n$, то за височината h важи асимптотичната оценка $h = \Theta(\log n)$, а тя гарантира бързодействие.

Въпреки че могат да се балансират всякакви дървета (не само двоични), все пак балансирането е най-интересно именно за наредените двоични дървета. Самобалансират се разнообразни типове дървета, които според свойствата си намират различни приложения.

Червено-черно дърво

Червено-черно дърво е вид самобалансиращо се двоично дърво за търсене, което притежава едновременно следните свойства:

1) Всеки връх е оцветен в един от два възможни цвята — червено или черно.
2) Всички листа са черни. Листата се представят като празни върхове (`NIL`): те не съдържат данни и указателите към тях са нулеви.
3) Никой червен връх не може да има червен пряк наследник.
4) Всеки път от даден връх до кое да е листо в неговото поддърво съдържа еднакъв брой черни върхове (зависещ само от избрания начален връх).



Понякога се добавя изискването коренът да бъде черен. Тъй като коренът винаги може да се пребоядиса в черно (без да се нарушат другите изисквания), това правило не влияе съществено на структурата от данни.

Според свойство 4 всички пътища от корена до кое да е листо съдържат един и същи брой черни върхове. Нека този брой бъде означен с b. Тогава всеки път от корена до листо съдържа поне b черни върха и (поради свойство 3) не повече от b червени върха (един червен връх в началото — корена; и още $b - 1$ червени върха — по един между всеки два черни). Така дължината на път от корена до листо (мерена с броя на ребрата) е между $b - 1$ вкл. и $2b - 1$ вкл. Частното $(2b - 1) / (b - 1)$ е между 1 и 4 (за големи b — между 1 и $2 + \varepsilon$), затова в тези граници лежи и частното от дължините на най-дълъг и най-къс път. Следователно височината $h = \Theta(\log n)$, където n е броят на всички върхове. Това равенство гарантира бързодействието.

Горните разсъждения важат само при $b \geq 2$. Но при $b = 1$ дървото се състои само от три върха: червен корен с два преки наследника — черни листа. Тогава всички пътища от корен до листо имат равни дължини (единици).

За съхраняването на информацията за цвета на всеки връх е нужен един бит, който за икономия на памет може да се пази в указателя към върха. Указателите са цели числа без знак (адреси на клетки). Тези числа рядко са твърде големи, старшите битове обикновено са нули; цветовете могат да заместят тези нули.

Търсенето на най-малък или най-голям елемент изисква да бъде изминат път от корена до някое листо, поради което изразходва време от порядъка на височината на дървото (намалена не повече от два пъти), тоест $\Theta(\log n)$, при всякакви входни данни.

Търсенето по ключ може да бъде по-бързо или по-бавно в зависимост от близостта на възела с ключа до корена на дървото. В най-лошия случай ключът се намира в листо или липсва изобщо; тогава времето е пак $\Theta(\log n)$. В най-добрия случай (когато ключът е в корена) времевата сложност е $\Theta(1)$, но това се случва рядко. Тъй като листата и близките до тях върхове са много, а върховете, близо до корена, са малко, то средната времева сложност е $\Theta(h)$, тоест $\Theta(\log n)$; строгото доказателство се свежда до изчисляването на израза

$$\frac{1}{n}\sum_{k=1}^{\log_2 n} k\,.\,2^{k-1} = \frac{1}{n}\left(\sum_{k=1}^{\log_2 n} k\,.\,x^{k-1}\right)\Bigg|_{x=2} = \frac{1}{n}\left(\sum_{k=1}^{\log_2 n}\left(x^k\right)'\right)\Bigg|_{x=2} = \frac{1}{n}\left(\sum_{k=1}^{\log_2 n} x^k\right)'\Bigg|_{x=2} =$$

$$\frac{1}{n}\left(\frac{x^{\log_2 n}-1}{x-1}\right)'\Bigg|_{x=2} = \frac{\left(\log_2 n\right)x^{-1+\log_2 n}.(x-1)-x^{\log_2 n}+1}{n(x-1)^2}\Bigg|_{x=2} = \frac{\frac{1}{2}n\left(\log_2 n\right)-n+1}{n} =$$

$$\frac{1}{2}\left(\log_2 n\right)-1+\frac{1}{n} = \Theta(\log n).$$

Вмъкване: Най-напред новият елемент се вмъква както при което и да е наредено двоично дърво, тоест по неговия ключ се търси мястото му в дървото и на това място се добавя ново листо с потребителската информация. Тъй като обаче червено-черните дървета не пазят информация в листата си, то при тях новият връх се добавя на мястото на съответното листо като вътрешен връх **N** с два преки наследника (черни листа) и се боядисва в червено. По-нататък има няколко случая, към чийто анализ може да се пристъпи и от други места. *

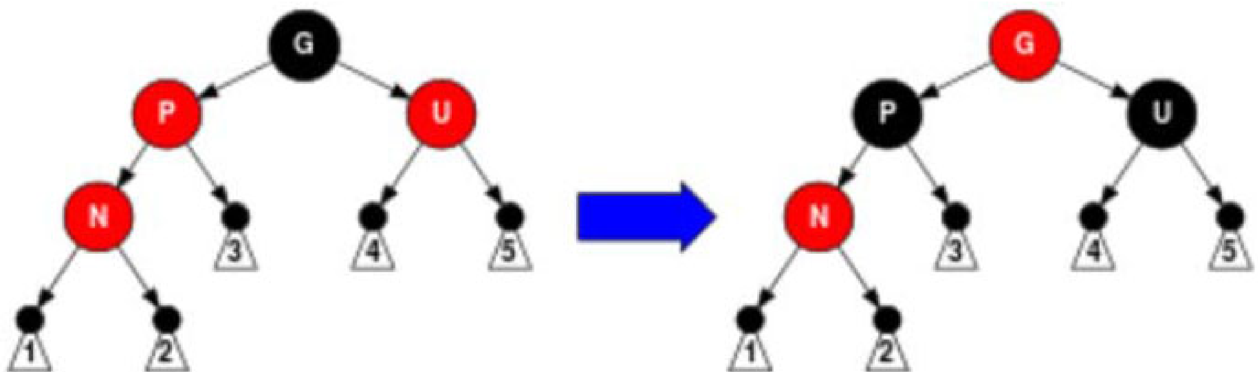
Първи случай: **N** няма родител, тоест **N** е корен на дървото. Следователно няма какво повече да се прави. Ако се изисква коренът да е черен, то върхът **N** се преоцветява в черно (това увеличава с единица броя на черните върхове по всички пътища от корен до листо, така че свойство 4 запазва силата си).

В останалите случаи **N** има някакъв родител **P**.

Втори случай: **P** е черен връх. Тогава няма какво повече да се прави, защото всички свойства се запазват: новият връх **N** е червен между черни (децата му и родителя **P**), а броят на черните върхове по пътищата, завършвали досега с отстраненото листо, не се променя, защото вместо отстраненото (черно) листо сега стои червен връх **N** с две черни листа (по едно на всеки нов път до листо).

В останалите случаи родителят **P** е червен връх. Това нарушава свойство 3, тъй като и прекият му наследник **N** е червен. Тъй като може без ограничение да се предполага, че коренът е черен, то **P** не е корен, тоест **P** притежава някакъв родител **G**. Върхът **G** има два преки наследника, единият от които е **P**; нека другият пряк наследник на **G** бъде означен с **U**. От свойство 3 (изпълнено преди вмъкването на новия връх **N**) следва, че върхът **G** е черен, понеже има червен пряк наследник (върха **P**). За цвета на върха **U** има две възможности.

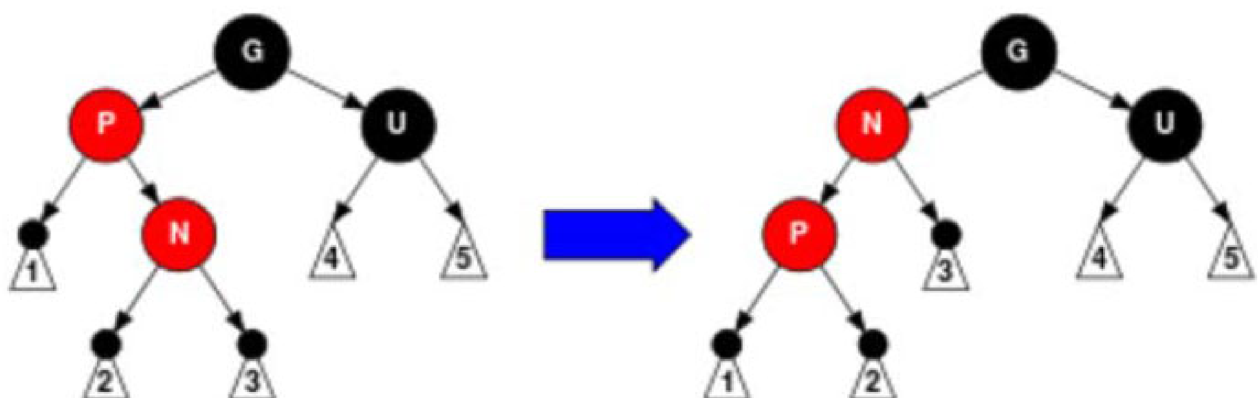
Трети случай: Върхът **U** е червен. Тогава пребоядисваме **U** и **P** в черно, а **G** — в червено. Така се отстранява конфликтът между цветовете на **N** и **P**, като се запазва броят на черните върхове по пътищата, минаващи през върха **G**.



Сега обаче може да възникне проблем с цвета на върха **G**: той може да е корен или да има червен родител. От мястото по-горе, обозначено с единична звезда на син фон, се провежда анализ на случаи за върха **G**, като **G** играе ролята на **N**. Така дефектът се премества към корена, докато бъде отстранен.

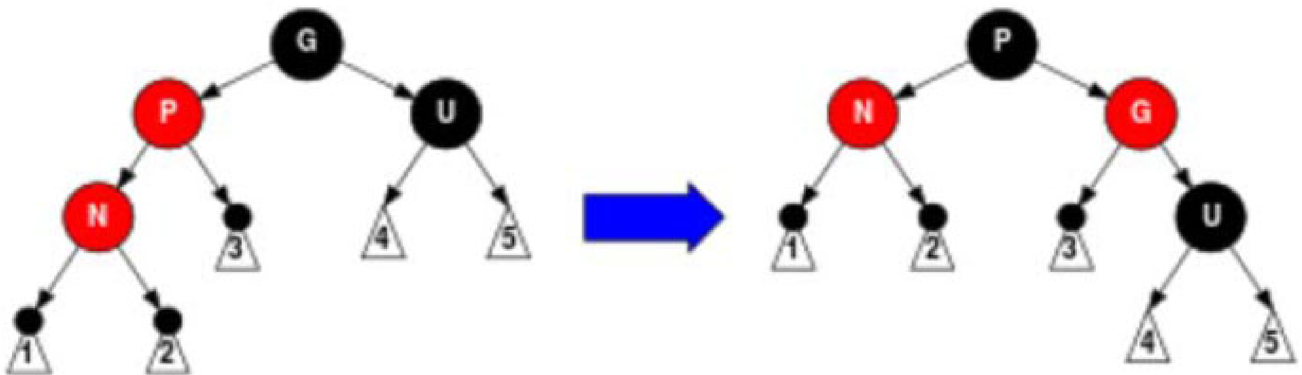
В останалите случаи върхът **U** е черен. Нужни са завъртания на поддървета. Нататък се предполага за определеност, че **P** е ляв, а **U** е десен пряк наследник на върха **G** (ако не е така, трябва само да се сменят посоките на завъртанията).

Четвърти случай: **N** е десен пряк наследник на **P**. Тогава в **P** се прави ляво завъртане на поддървото.



Това завъртане не нарушава свойствата на червено-черното дърво. Но тъй като конфликтът между **N** и **P** остава, то са нужни още действия. Ролите на **N** и **P** се разменят и алгоритъмът продължава към последния (петия) случай.

Пети случай: **N** е ляв пряк наследник на **P**. Тогава във върха **G** се прави дясно завъртане на поддървото, **P** се преоцветява в черно, а **G** — в червено.



Това завъртане не нарушава свойствата на червено-черното дърво. Освен това то решава конфликта между **N** и **P**, така че алгоритъмът приключва работа.

Направеното описание на алгоритъма е рекурсивно, но може да се забележи, че всяко препращане от един случай към друг представлява опашкова рекурсия, а тя лесно се заменя с цикъл. Следователно вмъкването се изпълнява на място, тоест почти не използва допълнителна памет.

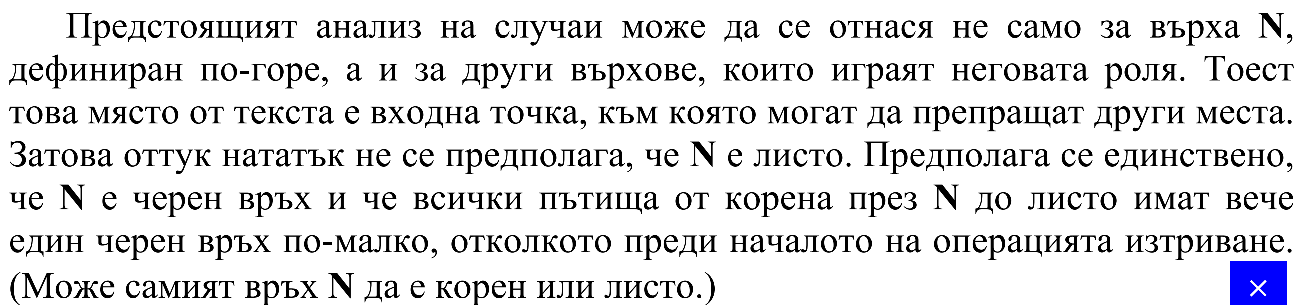
Изтриване: Операцията протича, както при всяко наредено двоично дърво, но с допълнителна обработка. При червено-черните дървета листо не може да бъде изтрито чрез пряка команда, защото това би нарушило изискванията към структурата (листата се трият само косвено, в резултат на изтриването на друг връх). Връх, който не е листо и чиито преки наследници не са листа, не се изтрива никога, а се изтрива само потребителската информация в него, като тя се заменя с потребителската информация в следващия по-голям връх, тоест най-малкия в дясното поддърво, който именно подлежи на изтриване. (Обаче цветът не се копира при пренасянето на потребителската информация.) В крайна сметка истински се изтрива (с освобождаване на паметта) само връх, който не е листо, но поне един от преките му наследници е листо.

Нека **M** е връх, който не е листо, но някой негов пряк наследник е листо. Нека **N** е другият пряк наследник на **M** (върхът **N** може също да е листо). Върхът **M** може да се изтрие по следния начин.

Ако **M** е червен, то **M** просто се заменя с **N** (указателят от родителя на **M**, сочещ към **M**, се пренасочва към **N**). Паметта за **M** се освобождава и прекият наследник на **M**, който не е **N**, се губи (изгубеният наследник е листо, така че не съдържа потребителска информация; на практика това е само указател `NIL`). Това изтриване не нарушава изискванията за цветовете на върховете: щом като **M** е червен връх, то **N** и родителят на **M** са черни, следователно свързването им с ребро е позволено; не се променя и броят на черните върхове по кой да е път от корена до листо (само се губи един от тези пътища — пътят, който завършва с листото, изтрито заедно с **M** като негов пряк наследник). Не е необходимо да се прави нищо повече в този случай, тоест алгоритъмът завършва.

Ако обаче **M** е черен връх, то може **N** да е червен. Отново **N** замества **M** (както в предишния случай), но сега **N** се пребоядисва в черно и алгоритъмът приключва работа, тъй като всички изисквания към структурата са спазени.

Единственият труден случай е този, в който върховете **M** и **N** са черни. Тъй като другият пряк наследник на **M** е листо и всички листа са черни, то **N** също е листо (иначе различните пътища от **M** до листо в поддървото на **M** биха съдържали различен брой черни върхове). Отново **N** замества **M** (указателят към **M**, идващ от родителя **P** на **M**, се пренасочва към **N**, тоест става нулев, а **P** става родител на **N**), но това не е достатъчно: пътищата от корена до листо, които досега са минавали през **M**, вече съдържат един черен връх по-малко от другите пътища, което е нарушение на свойство 4. Предстои да опишем как се отстранява това нарушение. За целта ще разгледаме няколко случая.

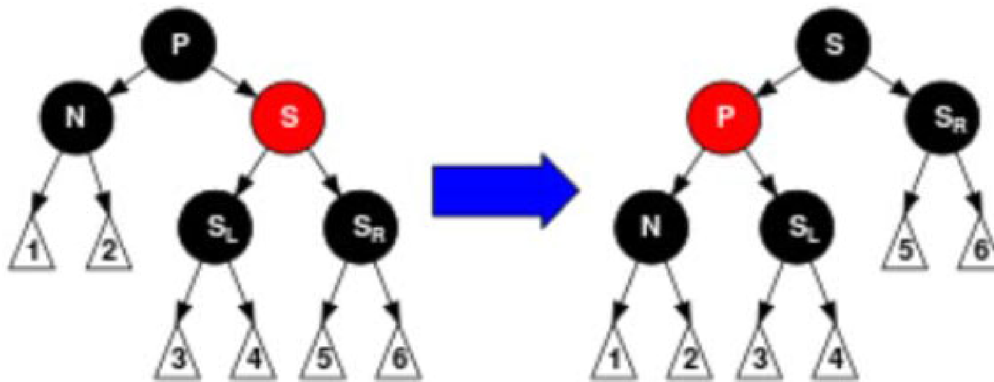


Предстоящият анализ на случаи може да се отнася не само за върха **N**, дефиниран по-горе, а и за други върхове, които играят неговата роля. Тоест това място от текста е входна точка, към която могат да препращат други места. Затова оттук нататък не се предполага, че **N** е листо. Предполага се единствено, че **N** е черен връх и че всички пътища от корена през **N** до листо имат вече един черен връх по-малко, отколкото преди началото на операцията изтриване. (Може самият връх **N** да е корен или листо.)

Първи случай: **N** е коренът на дървото. Тогава няма никакви други пътища (уж оставащи с повече черни върхове), тоест няма нарушение на свойство 4. Алгоритъмът завършва. (Където в двата абзаца от началото на страницата се говори за указател от родителя на **M** към **M**, трябва да се разбира просто указателят към корена на дървото. Удобно е върхът, подлежащ на изтриване, да не бъде посочван направо, а да се посочва указател към него, т.е. входният параметър на процедурата по изтриване да бъде указател към указател. Ако алгоритъмът е попаднал в първия случай направо от двата абзаца в началото на страницата, то указателят към корена е станал нулев (`NIL`), т.е. дървото се е опразнило.)

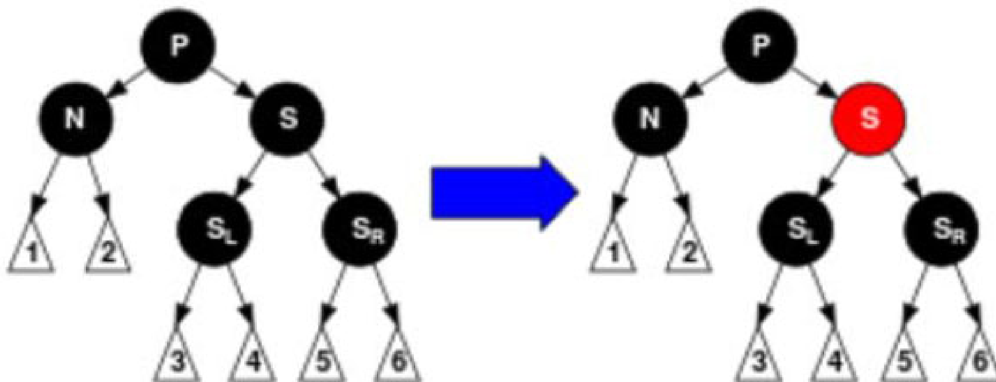
Остава да разгледаме другата възможност: **N** да не е коренът на дървото. Нека **P** е родителят на **N**. Нека **S** е другият пряк наследник на **P**. Върхът **S** не може да е листо, тъй като всички пътища от **P** през **N** до листо съдържат (в частта си след **P**) поне един черен връх (**N**), затуй всички пътища от **P** през **S** до листо имат (в частта си след **P**) един черен връх повече, тоест общо поне два черни върха, а ако **S** е листо, единственият такъв път е **P** — **S** и той съдържа (в частта си след **P**) един-единствен връх изобщо (**S**), което е противоречие. И така, **S** не е листо, значи **S** има точно два преки наследника; нека $\mathbf{S_L}$ е левият, а $\mathbf{S_R}$ е десният пряк наследник на **S**. Без ограничение можем да предположим, че **N** е левият, а **S** е десният пряк наследник на **P** (ако не е така, просто сменяме посоките на завъртане в случаите, разгледани по-нататък).

Втори случай: **S** е червен връх. От свойство 3 следва, че **P**, $\mathbf{S_L}$ и $\mathbf{S_R}$ са черни. Поддървото на **P** се завърта наляво, а върховете **P** и **S** си разменят цветовете.



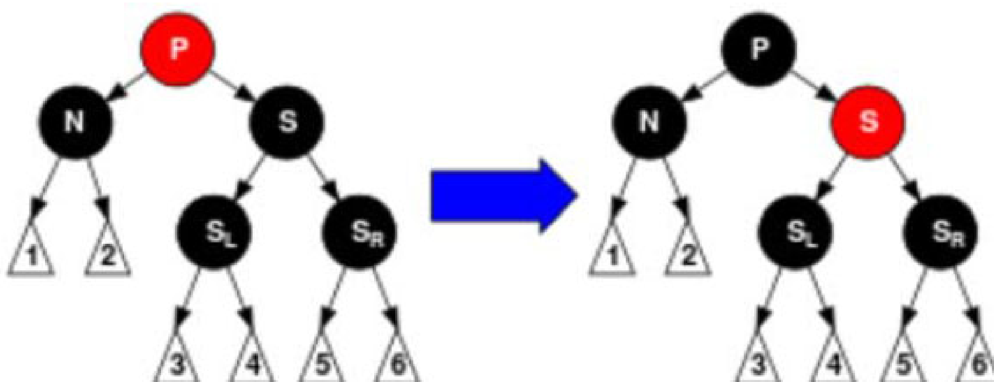
Това действие не променя броя на черните върхове по пътищата, т.е. дефектът остава засега, но може да бъде отстранен в някой от следващите случаи. Поради промяната в обкръжението на **N** стават приложими действията от четвъртия, петия или шестия случай.

Трети случай: **P**, **S**, $\mathbf{S_L}$ и $\mathbf{S_R}$ са черни върхове. **S** се преоцветява в червено.

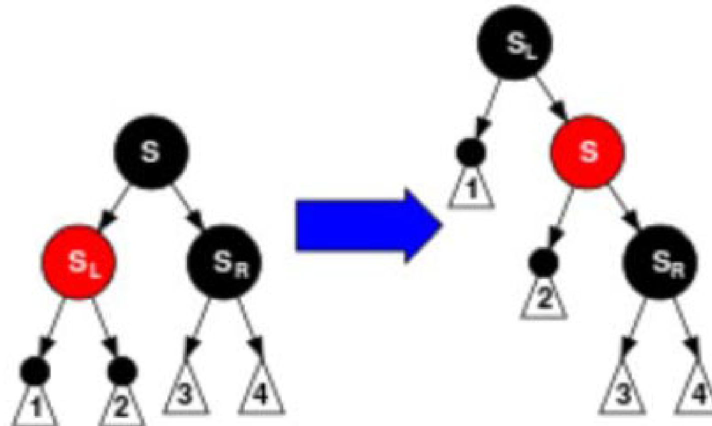


Това действие намалява с единица броя на черните върхове върху пътищата през **S**. Така дефектът вече е общ за цялото поддърво на върха **P**, а не само на **N**. Оттук нататък **P** започва да играе ролята на **N**, като анализът на случаи започва от мястото преди първия случай, обозначено с кръстче на син фон.

Четвърти случай: **S**, $\mathbf{S_L}$ и $\mathbf{S_R}$ са черни върхове, а **P** е червен връх. Тогава алгоритъмът разменя цветовете на **P** и **S**; така отстранява дефекта и завършва.

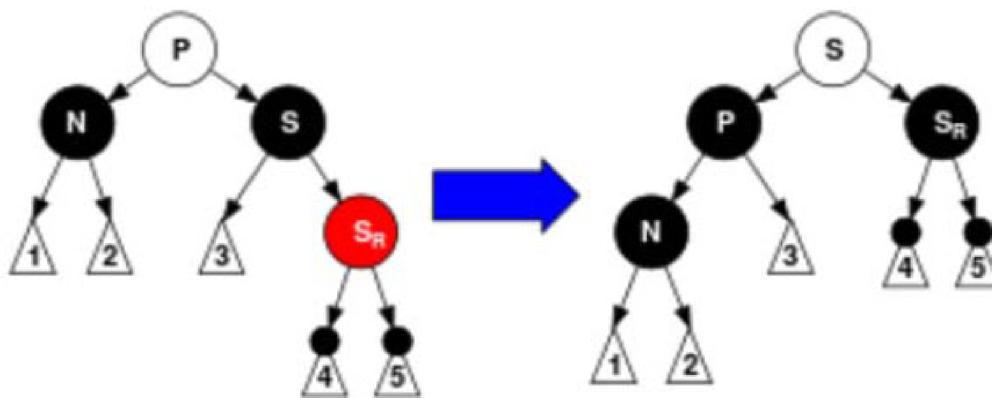


Пети случай: **S** и $\mathbf{S_R}$ са черни върхове, а $\mathbf{S_L}$ е червен. Тогава алгоритъмът завърта надясно поддървото на **S** и разменя цветовете на върховете $\mathbf{S_L}$ и **S**.



Така броят на черните върхове по пътищата от корена до листо не се променя, но алгоритъмът може да продължи към шестия случай (с други обозначения).

Шести случай: Върхът **S** е черен, $\mathbf{S_R}$ е червен, а **P** е оцветен произволно. Завърта се наляво поддървото на **P**, разменят се боите на **S** и **P**, а $\mathbf{S_R}$ става черен.



По този начин пътищата от корена през **N** до листо получават допълнителен черен връх, с което дефектът е отстранен и алгоритъмът приключва работа.

Изтриването се изпълнява на място, тоест почти не изразходва памет.

Операциите спазват свойствата на червено-черното дърво плюс изискването за черен корен. Времето за вмъкване / изтриване е $\Theta(\log n)$ в най-лошия случай (когато дефект се мести от най-далечно листо до корена). Средното време за вмъкване / изтриване е $\Theta(\log n)$, ако търсенето се смята за техен първи етап. Средното време за същинското вмъкване и изтриване, т.е. без търсенето, е $\Theta(1)$, защото, ако оцветяването се счита случайно, на всяка стъпка има вероятност p дефектът да не се премести по-нататък (стойността на p не е важна; тя е голяма, защото местенето спира при струпване на черни върхове, на които свойство 3 дава предимство пред червените). Геометричното вероятностно разпределение влече очакван брой стъпки $1 / p$, тоест константа (и то малка).

Забележка: В изложението са използвани материали от *Уикипедия*.