به نام خدا

ساختمان داده

آرش شفیعی

درخت‌ها

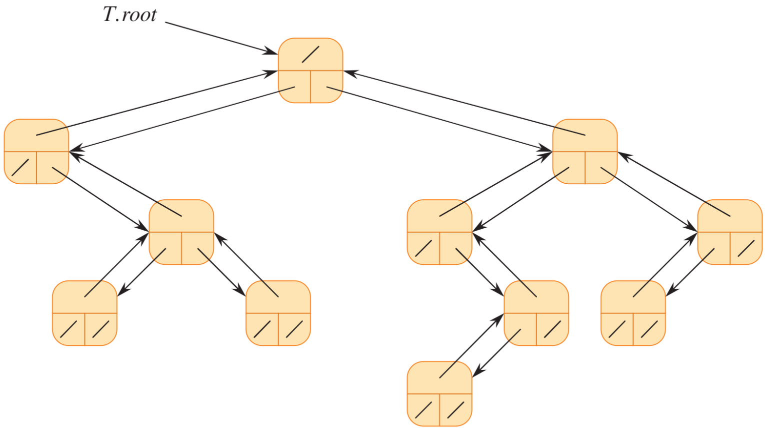
درخت‌ها

- لیست‌های پیوندی برای نمایش داده‌هایی به کار می‌روند که عناصر آن رابطه خطی دارند، اما همیشه روابط بین عناصر خطی نیست.
- داده ساختار درخت یکی از داده ساختارهایی است که برای نمایش و ذخیره‌سازی روابط غیر خطی استفاده می‌شود.

درخت‌ها

- در داده ساختار درخت هر عنصر می‌تواند صفر یا یک یا چند فرزند داشته باشد. یکی از حالات خاص درخت، درخت دودویی است که در آن هر عنصر حداکثر می‌تواند دو فرزند داشته باشد.

- در شکل زیر یک درخت دودویی نشان داده شده است. هر عنصر یک ویژگی p دارد که برای ذخیره‌سازی اشاره‌گر به پدر آن عنصر به کار می‌رود. همچنین هر عنصر دو ویژگی left و right دارد که اشاره‌گرهایی به فرزند سمت چپ و فرزند سمت راست آن عنصر در درخت دودویی T هستند.



درخت‌ها

- ساختمان داده‌های زیر، نحوه ذخیره سازی درخت دودویی را نشان می‌دهد.

Data Structure Node

```
struct NODE
int key
Node * p  ▷ pointer to the parent node
Node * left  ▷ pointer to the left child
Node * right  ▷ pointer to the right child
```

Data Structure Binary Tree

```
struct BINARYTREE
Node * root  ▷ pointer to the root of the tree
```

- اگر x.p=NIL باشد، آنگاه x ریشه درخت است. اگر x فرزند سمت چپ نداشته باشد، آنگاه x.left=NIL است و اگر فرزند سمت راست نداشته باشد، x.right=NIL است.
- ریشه درخت را با اشاره‌گر T.root مشخص می‌کنیم و اگر T.root=NIL باشد، آنگاه درخت خالی است.

درخت‌ها

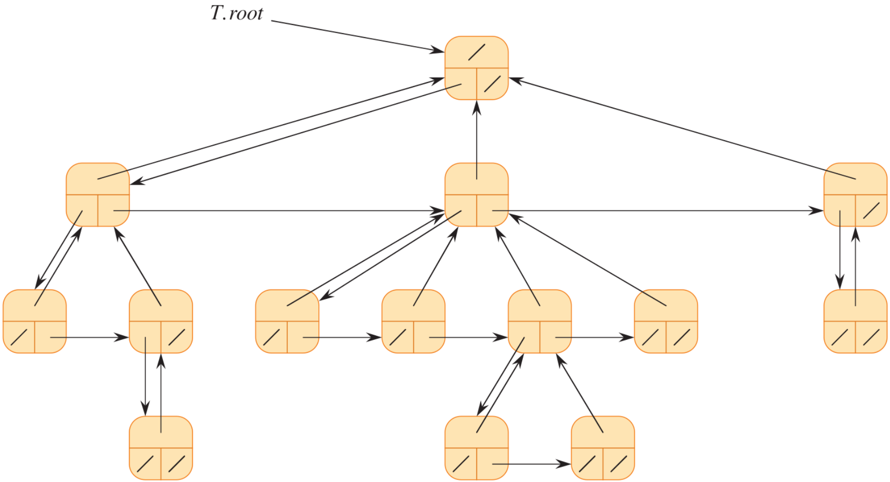
- می‌توانیم درخت دودویی را تعمیم دهیم به طوری که یک رأس درخت بتواند به هر تعداد دلخواه فرزند داشته باشد. می‌توانیم درختی تعریف کنیم که تعداد فرزند هر رأس در آن حداکثر k باشد. بنابراین ویژگی‌های left و right را با ویژگی‌های $child_1$ ، $child_2$ ، $\cdots$ و $child_k$ جایگزین می‌کنیم.
- حال فرض کنید تعداد فرزندان نامحدود باشد، بدین معنی که هیچ کران بالایی برای تعداد فرزندان وجود نداشته باشد. در این صورت نمی‌توانیم تعداد معینی ویژگی برای فرزندان یک رأس داشته باشیم. علاوه بر این، اگر k یک عدد بسیار بزرگ باشد و در عمل یک عنصر در اغلب مواقع تعداد کمی فرزند داشته باشد، برای ذخیره‌سازی اشاره‌گرها مقدار زیادی از حافظه را هدر داده‌ایم. در چنین مواقعی باید داده ساختار درخت را به گونه‌ای دیگر ذخیره کنیم.

- یک روش برای ذخیره‌سازی درخت وقتی تعداد فرزندان یک رأس نامحدود است بدین صورت است که از ویژگی left−child برای ذخیره‌سازی فرزند سمت چپ و از ویژگی right−sibling برای ذخیره‌سازی همزاد سمت راست استفاده کنیم.

- در این روش، هر رأس درخت یک اشاره‌گر p برای اشاره به پدر دارد و T.root به ریشهٔ درخت اشاره می‌کند. علاوه بر این دو، هر رأس x دو ویژگی دارد. ویژگی x.left−child به فرزند سمت چپ رأس x اشاره می‌کند و ویژگی x.right−sibling به همزاد سمت راست رأس x اشاره می‌کند.

درخت‌ها

- در شکل زیر یک درخت نشان داده شده است.



درخت‌ها

- ساختمان داده‌های زیر، نحوه ذخیره سازی درخت دودویی را نشان می‌دهد.

Data Structure Node

```
struct NODE
int key
Node * p  ▷ pointer to the parent node
Node * left_child  ▷ pointer to the left child
Node * right_sibling  ▷ pointer to the right sibling
```

Data Structure Tree

```
struct TREE
Node * root  ▷ pointer to the root of the tree
```

- اگر رأس x فرزندی نداشته باشد، آنگاه x.left−child=NIL است و اگر رأس x خود راست‌ترین فرزند یک پدر باشد، آنگاه x.right−sibling=NIL است.

درخت‌ها

- درخت‌ها را می‌توانیم به اشکال دیگری نمایش دهیم. در آینده خواهیم دید که درخت را می‌توان در آرایه نیز ذخیره کرد و یا در مواردی خاص که درخت تنها از فرزندان به سمت ریشه پیمایش می‌شود، می‌توانیم اشاره‌گری به فرزندان تعریف نکنیم.

پیمایش درخت‌ها

- درخت‌ها را می‌توانیم به سه روش پیمایش کنیم: پیمایش میان‌ترتیب [1]، پیمایش پیش‌ترتیب [2]، و پیمایش پس‌ترتیب [3].
- پیمایش میان ترتیب بدین دلیل اینگونه نامیده می‌شود که مقدار کلید یک رأس را بعد از چاپ کلیدهای رئوس زیر درخت سمت چپ و قبل از چاپ کلیدهای رئوس زیر درخت سمت راست چاپ می‌کند.
- در پیمایش پیش ترتیب مقدار کلید ریشه قبل از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.
- در پیمایش پس ترتیب مقدار کلید ریشه بعد از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.

[1] inorder tree walk
[2] preorder tree walk
[3] postorder tree walk

پیمایش درخت‌ها

- الگوریتم پیمایش میان ترتیب در زیر نشان داده شده است.

Algorithm Inorder Tree Walk

```
   function INORDER-TREE-WALK(x)
1: if x ≠ NIL then
2:    Inorder-Tree-Walk (x.left)
3:    print x.key
4:    Inorder-Tree-Walk (x.right)
```

- برای چاپ همه عناصر درخت جستجوی دودویی باید تابع Inorder-Tree-Walk(T.root) را فراخوانی کنیم.

پیمایش درخت‌ها

- الگوریتم پیمایش پیش‌ترتیب در زیر نشان داده شده است.

Algorithm Preorder Tree Walk

```
function PREORDER-TREE-WALK(x)
if x ≠ NIL then
   print x.key
   Preorder-Tree-Walk (x.left)
   Preorder-Tree-Walk (x.right)
```

پیمایش درخت‌ها

- الگوریتم پیمایش پس‌ترتیب در زیر نشان داده شده است.

Algorithm Postorder Tree Walk

```
function POSTORDER-TREE-WALK(x)
if x ≠ NIL then
    Postorder-Tree-Walk (x.left)
    Postorder-Tree-Walk (x.right)
    print x.key
```

- تمرین: یک درخت دودویی بسازید و آن را با پیمایش‌های میان‌ترتیب، پیش‌ترتیب و پس‌ترتیب پیمایش کنید.

پیمایش درخت‌ها

- یک عبارت ریاضی با عملگرهای دوگانی را می‌توان به صورت یک درخت دودویی ذخیره کرد، به طوری که یک رأس میانی در درخت همیشه یک عملگر است که بر روی مقدار زیردرخت سمت چپ به عنوان عملوند اول و زیردرخت سمت راست به عنوان عملوند دوم، عمل می‌کند.
- مسئله: می‌خواهیم الگوریتمی بنویسیم که یک عبارت پسوندی را دریافت کرده، آن را به یک درخت دودویی تبدیل کند.
- جواب: برای این کار از یک پشته استفاده می‌کنیم. یک مقدار از ورودی می‌خوانیم. اگر مقدار خوانده شده عملوند بود، آن را وارد پشته می‌کنیم و اگر عملگر p بود، دو مقدار x و y را از پشته می‌خوانیم و درختی می‌سازیم که ریشه آن عملگر p و زیر درخت سمت چپ آن x و زیردرخت سمت راست آن y باشد.

پیمایش درخت‌ها

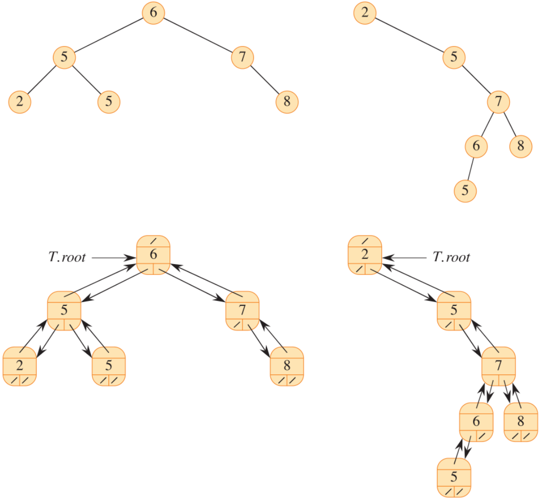
- مسئله: می‌خواهیم از یک درخت که یک عبارت ریاضی را نگهداری می‌کند، یک عبارت میانوندی بسازیم.
- جواب: یک پیمایش میان‌ترتیب انجام می‌دهیم به طوری که قبل از پیمایش زیردرخت سمت چپ یک پرانتز باز می‌کنیم و بعد از پیمایش زیردرخت سمت راست پرانتز را می‌بندیم.
- تمرین: برای عبارت پسوندی $2\ 3\ +\ 4\ 1\ -\ *$ یک درخت دودویی بسازید. پیمایش پس‌ترتیب چه عبارتی تولید می‌کند؟ با یک پیمایش میان‌ترتیب، یک عبارت میانوندی تولید کنید.

درخت جستجوی دودویی

- درخت جستجوی دودویی یک درخت دودویی است که برای جستجوی بهینه در مجموعه‌ای از عناصر استفاده می‌شود.
- عناصر در درخت جستجوی دودویی به گونه‌ای ذخیره می‌شوند که ویژگی درخت جستجوی دودویی همیشه حفظ شود.
- فرض کنید x یک رأس در درخت جستجوی دودویی باشد. اگر y یک رأس در زیر درخت سمت چپ x باشد آنگاه $y.key \leqslant x.key$ و اگر y یک رأس در زیر درخت سمت راست x باشد، آنگاه داریم $y.key \geqslant x.key$.

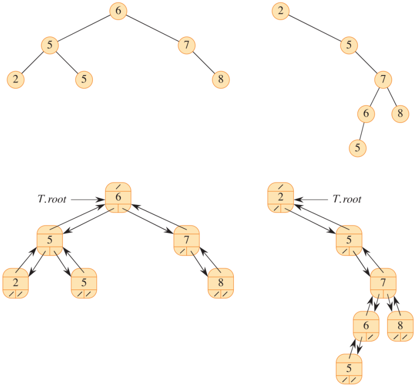
درخت جستجوی دودویی

- شکل زیر دو درخت جستجوی دودویی را نشان می‌دهد.



درخت جستجوی دودویی

- در درخت سمت چپ کلید ریشه درخت ۶ است. کلیدهای ۲ و ۵ و ۵ در زیر درخت سمت چپ قرار دارند و مقدار آنها از ۶ بیشتر نیست و کلیدهای ۷ و ۸ در زیر درخت سمت راست قرار دارند و مقدار آنها از ۶ کمتر نیست.



درخت جستجوی دودویی

- به دلیل ویژگی خاص درخت جستجوی دودویی، با پیمایش میان ترتیب [1] می‌توان کلیدها را به ترتیب چاپ کرد.
- پیمایش میان ترتیب بدین دلیل اینگونه نامیده می‌شود که مقدار کلید یک رأس را بعد از چاپ کلیدهای رئوس زیر درخت سمت چپ و قبل از چاپ کلیدهای رئوس زیر درخت سمت راست چاپ می‌کند.
- در پیمایش پیش ترتیب [2] مقدار کلید ریشه قبل از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود و در پیمایش پس ترتیب [3] مقدار کلید ریشه بعد از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.

[1] inorder tree walk
[2] preorder tree walk
[3] postorder tree walk

درخت جستجوی دودویی

- الگوریتم پیمایش میان ترتیب در زیر نشان داده شده است.

Algorithm Inorder Tree Walk

```
function INORDER-TREE-WALK(x)
if x ≠ NIL then
    Inorder-Tree-Walk (x.left)
    print x.key
    Inorder-Tree-Walk (x.right)
```

- برای چاپ همه عناصر درخت جستجوی دودویی باید تابع Inorder-Tree-Walk(T.root) را فراخوانی کنیم.

- این الگوریتم در زمان $\Theta(n)$ به ازای درخت جستجوی دودویی با n رأس اجرا می‌شود.

- درستی این الگوریتم مستقیما با استفاده از استقرا بر روی ویژگی درخت جستجوی دودویی اثبات می‌شود.

درخت جستجوی دودویی

- قضیه : اگر x ریشه یک زیر درخت با n رأس باشد، فراخوانی Inorder-Tree-Walk(x) در زمان $\Theta(n)$ انجام می‌شود.

- اثبات : فرض کنید $T(n)$ زمان مورد نیاز برای اجرای الگوریتم برروی زیر درختی با n رأس باشد.

- همه رئوس زیر درخت در نهایت بررسی می‌شوند پس لزوما $T(n) = \Omega(n)$. کافی است ثابت کنیم $T(n) = O(n)$.

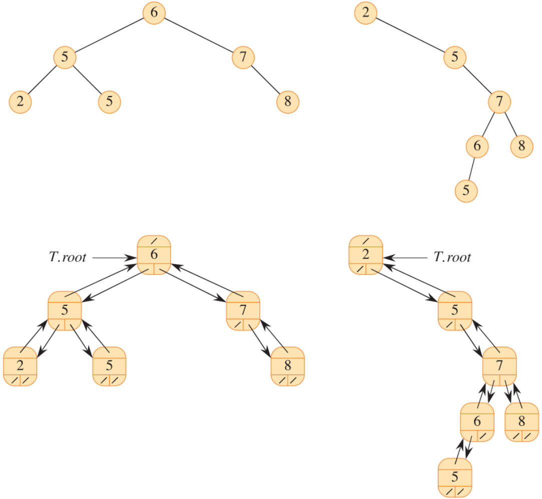
- الگوریتم به ازای درخت تهی مقدار ثابتی زمان صرف می‌کند، پس $T(0) = c$ به طوری که $c > 0$.

- به ازای $n > 0$ ، فرض کنید با فراخوانی تابع برای رأس x تعداد k رأس در زیر درخت سمت چپ وجود داشته باشند آنگاه در زیر درخت سمت راست $n - k - 1$ رأس وجود خواهد داشت. اگر اجرای بدنه تابع در زمان d انجام شود به طوری که $d > 0$ آنگاه خواهیم داشت $T(n) \leqslant T(k) + T(n - k - 1) + d$.

- با حل این رابطه به روش جایگذاری به دست می‌آوریم $T(n) = O(n)$.

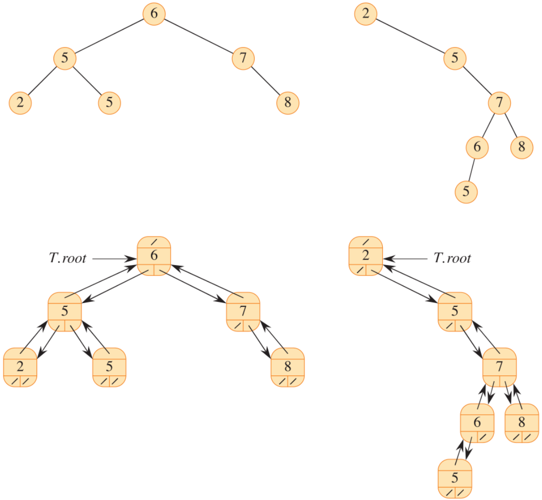
درخت جستجوی دودویی

- با پیمایش میان ترتیب در هر دو درخت زیر ترتیب ۸، ۷، ۶، ۵، ۵، ۲ به دست می‌آید.



درخت جستجوی دودویی

- تمرین: درخت‌ها را به صورت پیش‌ترتیب و پس‌ترتیب نیز پیمایش کنید.



درخت جستجوی دودویی

- جستجو : برای یک رأس با یک کلید دلخواه در درخت جستجوی دودویی، تابع Tree-Search به صورت زیر تعریف می‌شود.

Algorithm Tree Search

```
function TREE-SEARCH(x,k)
if x == NIL or k == x.key then
   return x
if k < x.key then
   return Tree-Search (x.left, k)
else  return Tree-Search (x.right, k)
```

- به ازای اشاره‌گر x به ریشه یک زیر درخت و مقدار کلید k تابع Tree-search(x،k) اشاره‌گری به رأسی از درخت برمی‌گرداند که مقدار کلید آن k است در صورتی که چنین رأسی وجود داشته باشد، در غیر اینصورت مقدار NIL را باز می‌گرداند.

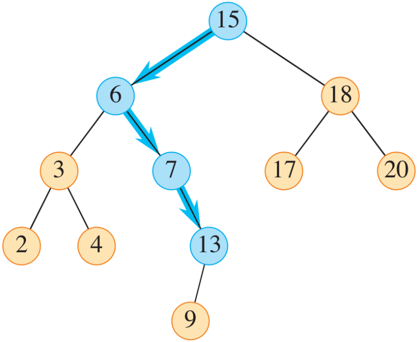
- برای جستجو در تمام درخت تابع Tree-search(T.root،k) فراخوانی می‌شود.

درخت جستجوی دودویی

- این تابع جستجو را با ریشه شروع می‌کند. به ازای هر رأس x مقدار کلید k و x.key را مقایسه می‌کند. اگر این دو مقدار برابر باشند، جستجو خاتمه پیدا می‌کند. اگر k کوچک‌تر از x.key باشد، جستجو با فراخوانی تابع برای زیر درخت سمت چپ x ادامه پیدا می‌کند، زیرا به علت ویژگی درخت جستجوی دودویی کلید k نمی‌تواند در زیر دخت سمت راست باشد. همچنین اگر k بزرگ‌تر از x.key باشد، جستجو با فراخوانی تابع برای زیر درخت سمت راست x ادامه می‌یابد.

درخت جستجوی دودویی

- در شکل زیر جستجوی کلید ۱۳ در یک درخت جستجوی دودویی نشان داده شده است.



- رئوسی که در فرایند جستجو بررسی می‌شوند مسیری از ریشه درخت به سمت برگ‌های درخت می‌سازند و بنابراین پیچیدگی زمانی الگوریتم $O(h)$ است. به طوری که h ارتفاع درخت است.

درخت جستجوی دودویی

- کمینه و بیشینه : برای یافتن عنصری در درخت جستجوی دودویی که کلید آن کمینه است، باید فرزند چپ هر رأس را با شروع از ریشه بررسی کنیم تا به NIL برسیم. آخرین رأس بررسی شده کلید با کمترین مقدار در درخت است.

درخت جستجوی دودویی

- تابع Tree−Minimum اشاره‌گری به عنصر کمینه در زیر درخت با ریشه x باز می‌گرداند.

Algorithm Tree Minimum

```
function TREE-MINIMUM(x)
while x.left ≠ NIL do
    x = x.left
return x
```

- ویژگی درخت جستجوی دودویی تضمین می‌کند که تابع Tree−Minimum درست است. اگر رأس x زیر درخت سمت چپ نداشته باشد، از آنجایی که همهٔ کلیدهای زیر درخت سمت راست x از x.key بزرگ‌تر یا مساوی هستند، بنابراین x.key کوچک‌ترین کلید در درخت است. اگر رأس x زیر درخت سمت چپ داشته باشد، همه کلیدهای زیر درخت سمت چپ از x.key کوچک‌تر هستند، بنابراین زیر درخت سمت چپ باید بررسی شود.

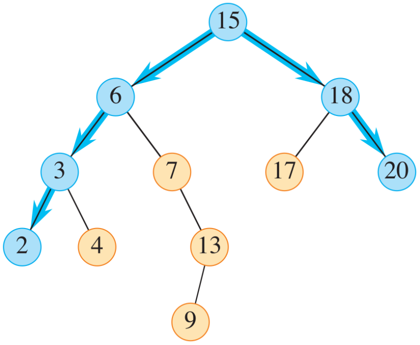
- به طور مشابه تابع Tree-Maximum اشاره‌گری به عنصر بیشینه در زیر درخت ریشه x باز می‌گرداند.

Algorithm Tree Maximum

```
function TREE-MAXIMUM(x)
while x.right ≠ NIL do
    x = x.right
return x
```

درخت جستجوی دودویی

- در شکل زیر مقدار کمینه و بیشینه در درخت پیدا شده‌اند.



- هر دو تابع Tree–Minimum و Tree–Maximum در زمان O(h) اجرا می‌شوند به طوری که h ارتفاع درخت است.

درخت جستجوی دودویی

- رئوس بعدی و قبلی : اگر همه کلیدها در درخت جستجوی دودویی یکتا باشند، رأس بعدی [1] x کوچک‌ترین رأسی است که مقدار آن از x.key بزرگ‌تر است.
- رأس بعدی رأس x ، رأسی است که در یک پیمایش میان ترتیب بعد از رأس x پیمایش می‌شود.

[1] successor

درخت جستجوی دودویی

- تابع Tree-Successor رأس بعدی رأس x را در یک درخت جستجوی دودویی باز می‌گرداند اگر چنین رأسی وجود داشته باشد و در غیر این‌صورت مقدار NIL را باز می‌گرداند.

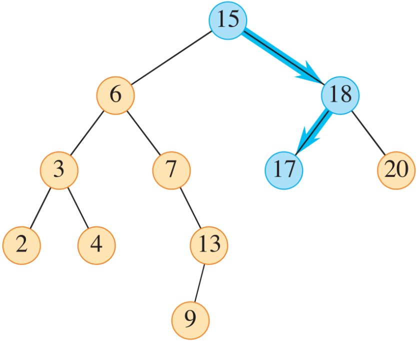
Algorithm Tree Successor

```
function TREE-SUCCESSOR(x)
if x.right ≠ NIL then
    return Tree-Minimum (x.right)      ▷ leftmost node in right subtree
else ▷ find the lowest ancestor of x whose left child is an ancestor of x
    y = x.p
    while y ≠ NIL and x == y.right do
        x = y
        y = y.p
    return y
```

درخت جستجوی دودویی

- اگر زیر درخت سمت راست رأس x تهی نباشد، رأس بعدی x کوچک‌ترین (چپ‌ترین) رأس در زیر درخت سمت راست رأس x است.

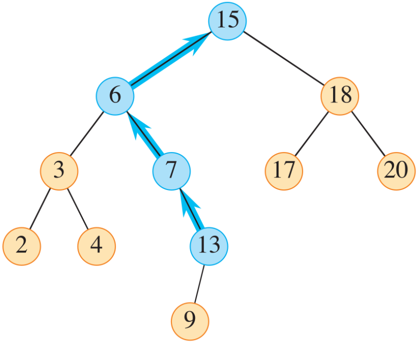
- در شکل زیر رأس بعدی ۱۵ کوچک‌ترین مقدار در زیر درخت سمت راست آن یعنی ۱۷ است.



درخت جستجوی دودویی

- اگر زیر درخت سمت راست رأس x تهی باشد و رأس بعدی x رأس y باشد، آنگاه برای یافتن y در درخت با شروع از رأس x بالا می‌رویم تا جایی‌که یا به ریشه برسیم و یا به رأسی برسیم که فرزند چپ پدر خود باشد.

- در شکل زیر رأس بعدی ۱۳ مقدار ۱۵ است.



درخت جستجوی دودویی

- زمان اجرای تابع Tree-Successor در درختی با ارتفاع h برابر است با O(h) زیرا یا باید مسیری از برگ به ریشه طی شود و یا مسیری به سمت برگ‌ها.
- تابع Tree-Predecessor قرینه تابع Tree-Successor است و پیچیدگی آن نیز O(h) است.

درخت جستجوی دودویی

- درج : عملیات درج در درخت جستجوی دودویی باعث تغییر ساختار درخت می‌شود. درج یک عنصر باید به گونه‌ای باشد که ویژگی درخت جستجوی دودویی حفظ شود.
- تابع Tree−Insert یک رأس جدید به درخت جستجوی دودویی اضافه می‌کند. این تابع درخت T و رأس z را که مقدار کلید آن در z.key ذخیره شده است دریافت می‌کند. همچنین z.left=NIL و z.right=NIL قرار داده شده است. تابع درخت T را به گونه‌ای تغییر می‌دهد که z در مکان مناسب در درخت قرار بگیرد.

درخت جستجوی دودویی

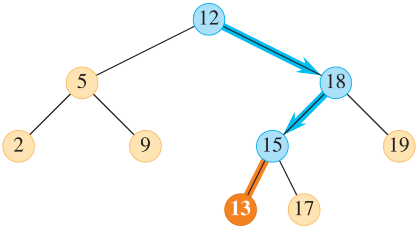
Algorithm Tree Insert

```
function TREE-INSERT(T,z)
x = T.root  ▷ node being compared with z
y = NIL  ▷ y will be parent of z
while x ≠ NIL do  ▷ descend until reaching a leaf
    y = x
    if z.key < x.key then
        x = x.left
    else  x = x.right
z.p = y  ▷ found the location-insert z with parent y
if y == NIL then
    T.root = z      ▷ tree T was empty
else if z.key < y.key then
    y.left = z
else  y.right = z
```

درخت جستجوی دودویی

- شکل زیر نشان می‌دهد Tree−Insert چگونه کار می‌کند. این تابع از ریشه آغاز می‌کند و در درخت جستجوی دودویی پایین می‌رود تا مکان مناسب z را پیدا کند.

- در بررسی درخت، تابع اشاره‌گر x و اشاره‌گر y را به عنوان پدر x نگهداری می‌کند. با توجه به مقدار z.key اشاره‌گر x در درخت حرکت می‌کند تا جایی که x برابر با NIL شود. رأس z در مکان به دست آمده توسط اشاره‌گر x قرار می‌گیرد. درواقع جایگاه به دست آمده برای z سمت چپ یا سمت راست پدر z است که اشاره‌گر y به آن اشاره می‌کند.



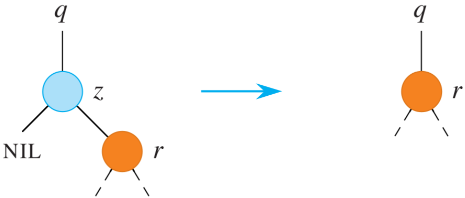
- پیچیدگی زمانی درج در درخت جستجو برای درخت با ارتفاع h برابر است با O(h) .

درخت جستجوی دودویی

- حذف : برای حذف رأس z از درخت جستجوی دودویی T سه حالت زیر را در نظر می‌گیریم.
- اگر z فرزندی نداشته باشد، با تغییر اشاره‌گری که به z اشاره می‌کند به NIL به سادگی رأس z حذف می‌شود.
- اگر z تنها یک فرزند داشته باشد، آنگاه فرزند z تبدیل به فرزند پدر z می‌شود. درواقع پدر z به جای اشاره به z به فرزند z اشاره خواهد کرد.
- اگر z دو فرزند داشته باشد، ابتدا باید رأس مابعد z را که در زیر درخت سمت راست z است پیدا کنیم و توسط اشاره‌گر y مکان آن را نگهداری کنیم. سپس باید y را در مکان z در درخت قرار دهیم. مابقی رئوس در زیر درخت سمت راست z زیر درخت سمت راست y می‌شوند و همه زیر درخت سمت چپ z زیر درخت سمت چپ y می‌شود. از آنجایی که y رأس مابعد z است، نمی‌تواند فرزند سمت چپ داشته باشد و فرزند سمت راست y در مکان اصلی y قرار خواهد گرفت.

درخت جستجوی دودویی

- اگر z فرزند سمت چپ نداشته باشد، طبق شکل زیر، رأس z با فرزند سمت راست آن (که ممکن است NIL باشد) جایگزین می‌شود.

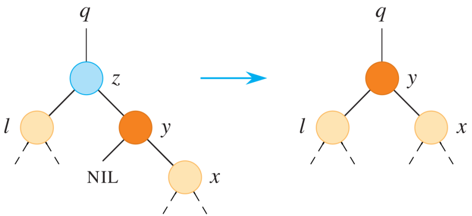


- اگر z تنها فرزند سمت چپ داشته باشد، طبق شکل زیر، رأس z با فرزند سمت چپ آن جایگزین می‌شود.

درخت جستجوی دودویی

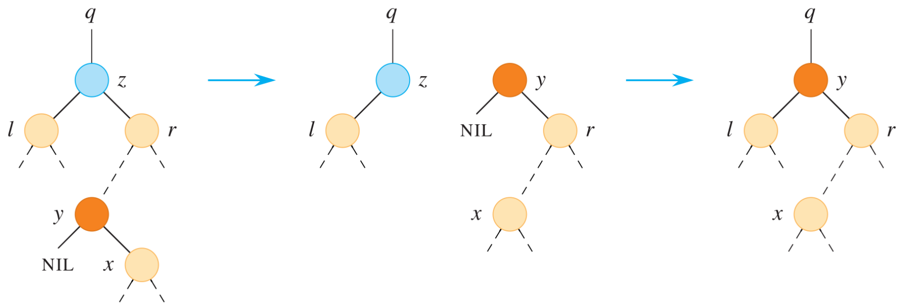
- اگر z هر دو فرزند چپ و راست را داشته باشد، ابتدا رأس مابعد z به نام y را که در زیر درخت سمت راست z قرار دارد پیدا می‌کنیم. رأس y الزاما زیر درخت سمت چپ ندارد. رأس y را از مکان خود خارج کرده و جایگزین رأس z می‌کنیم. دو حالت برای y وجود دارد.

(۱) اگر y فرزند سمت راست z باشد، رأس z را با y جایگزین می‌کنیم و فرزند سمت راست y را در جای خود باقی می‌گذاریم.



درخت جستجوی دودویی

(۲) اگر y در زیر درخت راست رأس z باشد اما فرزند سمت راست z نباشد، ابتدا y را با فرزند سمت راست آن جایگزین کرده، سپس z را با y جایگزین می‌کنیم.



درخت جستجوی دودویی

- در فرایند حذف یک رأس، نیاز داریم زیر درخت‌ها را در درخت جستجو دودویی جابجا کنیم.

- تابع Transplant یک زیر درخت را با یک زیر درخت دیگر جایگزین می‌کند. وقتی این تابع زیر درختی با ریشه u را با یک زیر درخت با ریشه v جایگزین می‌کند، پدر رأس u پدر رأس v می‌شود، در نتیجه پدر رأس u در نهایت v را به عنوان فرزند خود خواهد داشت. همچنین v می‌تواند NIL باشد.

Algorithm Transplant

```
function TRANSPLANT(T,u,v)
if u.p == NIL then
    T.root = v
else if u == u.p.left then
    u.p.left = v
else  u.p.right = v
if v ≠ NIL then
    v.p = u.p
```

درخت جستجوی دودویی

- خطوط ۱ و ۲ حالتی را بررسی می‌کنند که u ریشه درخت T باشد. در غیر اینصورت، u فرزند چپ یا فرزند راست پدر خود است. اگر u یک فرزند سمت چپ باشد، خطوط ۳ و ۴ مقدار u.p.left را به روز رسانی می‌کند. اگر u یک فرزند سمت راست باشد، خط ۵ مقدار u.p.right را به روزرسانی می‌کند.
- از آنجایی که v می‌تواند تهی باشد، خطوط ۶ و ۷ مقدار v.p را به روزرسانی می‌کند تنها اگر v تهی نباشد.
- تابع Transplant مقادیر v.left و v.right را تغییر نمی‌دهد و این کار را به عهده تابع فراخوانی کننده Transplant می‌گذارد.

درخت جستجوی دودویی

- تابع Tree-Delete از تابع Transplant برای حذف z از درخت جستجوی دودویی T استفاده می‌کند.

Algorithm Tree Delete

```
function TREE-DELETE(T,z)
if z.left == NIL then
    Transplant (T,z,z.right)        ▷ replace z by its right child
else if z.right == NIL then
    Transplant (T,z,z.left)         ▷ replace z by its left child
else  y = Tree-Minimum(z.right)     ▷ y is z's successor
    if y ≠ z.right then             ▷ is y farther down the tree?
        Transplant (T,y,y.right)    ▷ replace y by its right child
        y.right = z.right           ▷ z's right child becomes
        y.right.p = y               ▷ y's right child
    Transplant (T,z,y)              ▷ replace z by its successor y
    y.left = z.left                 ▷ and give z's left child to y,
    y.left.p = y                    ▷ which had no left child
```

درخت جستجوی دودویی

- خطوط ۱ و ۲ به حالتی رسیدگی می‌کنند که رأس z فرزند سمت چپ ندارد و خطوط ۳ و ۴ حالتی را بررسی می‌کنند که z فرزند چپ دارد ولی فرزند سمت راست ندارد.

- خطوط ۵ تا ۱۲ حالات دیگر که z دو فرزند دارد را بررسی می‌کنند. خط ۵ رأس y را که رأس مابعد z است پیدا می‌کند. از آنجایی که z یک زیردرخت راست غیر تهی دارد، رأس مابعد آن رأسی در زیر درخت سمت راست آن با کمترین مقدار کلید است، بنابراین از تابع Tree-Minimum(z.right) استفاده می‌شود. قبلا ذکر کردیم که الزاما y فرزند سمت چپ ندارد. تابع حذف، باید y را از مکان فعلی خود خارج و z را با y جایگزین کند.

درخت جستجوی دودویی

- اگر y فرزند سمت راست z باشد، آنگاه در خطوط ۱۰ تا ۱۲ رأس z با رأس y جایگزین می‌شود و فرزند سمت چپ y را با فرزند سمت چپ z جایگزین می‌کند. رأس y فرزند راست خود را حفظ می‌کند و بنابراین y.right تغییر نمی‌کند.

- اگر y فرزند سمت راست z نباشد، آنگاه دو رأس باید جابجا شوند. خطوط ۷ تا ۹ رأس y را با فرزند سمت راست y جایگزین می‌کند و رأس سمت راست z را با رأس y جایگزین می‌کند. در نهایت در خطوط ۱۰ تا ۱۲ رأس z با y جایگزین می‌شود و فرزند سمت چپ y با فرزند سمت چپ z جایگزین می‌شود.

درخت جستجوی دودویی

- هر یک از خطوط تابع Tree-Delete در زمان ثابت اجرا می‌شوند به جز فراخوانی تابع Tree-Minimum و بنابراین تابع در زمان O(h) اجرا می‌شود به طوری که h ارتفاع درخت است.

درخت ای‌وی‌ال

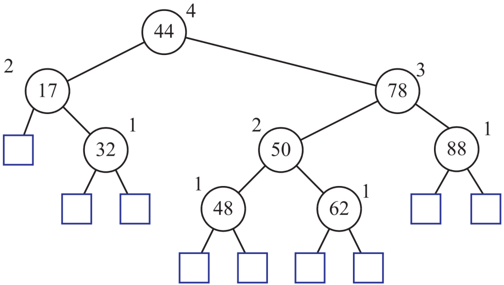
- درخت جستجوی دودویی ممکن است عناصر را به گونه‌ای در درخت درج کند که پیچیدگی زمانی جستجو در مجموعه‌ای از n عنصر در بدترین حالت $O(n)$ باشد.
- اگر درخت جستجوی دودویی ارتفاع متوازن داشته باشد یا به عبارت دیگر دارای ویژگی ارتفاع متوازن [1] باشد می‌توانیم پیچیدگی زمانی بدترین حالت را کاهش دهیم.
- ویژگی ارتفاع متوازن : به ازای هر رأس میانی v در درخت T ، اختلاف ارتفاع فرزندان آن حداکثر برابر با یک است.

[1] height-balance property

درخت ای‌وی‌ال

- هر درخت جستجوی دودویی T که دارای ویژگی ارتفاع متوازن باشد، درخت ای‌وی‌ال [1] نامیده می‌شود که نام خود را از ابتدای نام ابداع کنندگان آن یعنی آدلسون ولسکی [2] و لندیس [3] گرفته است.

- یک مثال از درخت ای‌وی‌ال در شکل زیر نشان داده شده است.



[1] AVL tree

[2] Adelson-Velskii

[3] Landis

درخت ای‌وی‌ال

- هر زیر درخت در یک درخت ای‌وی‌ال یک درخت ای‌وی‌ال است.
- قضیه : ارتفاع یک درخت ای‌وی‌ال که n رأس را ذخیره می‌کند $O(\lg n)$ است.
- اثبات : به جای این که کران بالای ارتفاع درخت را محاسبه کنیم، به جهت سهولت، کران پایین رئوس میانی یک درخت ای‌وی‌ال با ارتفاع h یعنی $n(h)$ را محاسبه می‌کنیم.

درخت ای‌وی‌ال

- به ازای اعداد کوچک داریم $n(1) = 1$ و $n(2) = 2$ زیرا یک درخت ای‌وی‌ال با ارتفاع یک باید حداقل یک رأس میانی داشته باشد و یک درخت ای‌وی‌ال با ارتفاع ۲ باید حداقل ۲ رأس میانی داشته باشد.
- حال به ازای $h \geqslant 3$ یک درخت ای‌وی‌ال با ارتفاع h و کمترین تعداد رئوس به گونه‌ای است که هر دو زیر درخت آن درخت‌های ای‌وی‌ال با کمترین تعداد رأس هستند : یکی با ارتفاع $h - 1$ و دیگری با ارتفاع $h - 2$. اگر ریشه را هم در نظر بگیریم، رابطه زیر را به دست می‌آوریم :

$$n(h) = 1 + n(h-1) + n(h-2)$$

- تابع $n(h)$ یک تابع اکیدا صعودی است. بنابراین $n(h-1) \geqslant n(h-2)$.
- پس می‌توانیم بنویسیم $n(h) > 2n(h-2)$.

- با بسط دادن این رابطه به دست می‌آوریم $n(h) > 2^i \cdot n(h-2i)$ به ازای $h - 2i \geqslant 1$.
- مقادیر پایه $n(1) = 1$ و $n(2) = 2$ را قبلا محاسبه کردیم، بنابراین i را به گونه‌ای انتخاب می‌کنیم که $h - 2i$ برابر با ۱ یا ۲ شود، پس $i = \lceil \frac{h}{2} \rceil - 1$ (در اینصورت اگر h زوج باشد $h - 2i = 2$ و اگر h فرد باشد $h - 2i = 1$).

درخت ای‌وی‌ال

- با جایگذاری مقدار i به دست می‌آوریم :

$$n(h) > 2^{\lceil \frac{h}{2} \rceil - 1} \cdot n(h - 2\lceil \frac{h}{2} \rceil + 2) > 2^{\lceil \frac{h}{2} \rceil - 1} n(1) > 2^{\frac{h}{2} - 1}$$

- از دو طرف رابطه لگاریتم می‌گیریم و به دست می‌آوریم $\lg n(h) > \frac{h}{2} - 1$.
- بنابراین داریم $h < 2 \lg n(h) + 2$
- نتیجه می‌گیریم که ارتفاع یک درخت ای‌وی‌ال با n رأس حداکثر $2 \lg n + 2$ است که برابر با $O(\lg n)$ است.

درج در درخت ای‌وی‌ال

- درج در درخت ای‌وی‌ال و حذف از آن همانند درج و حذف درخت جستجوی دودویی است با این تفاوت که عملیات بیشتری برای ایجاد توازن باید انجام شود.
- درج در یک درخت جستجوی دودویی ممکن است ویژگی توازن ارتفاع درخت ای‌وی‌ال را نقض کند، بنابراین باید پس از درج یک رأس درخت را متوازن کنیم.
- به ازای درخت جستجوی دودویی T ، می‌گوییم رأس v از درخت متوازن[1] است، اگر مقدار قدر مطلق تفاضل ارتفاع فرزندان v حداکثر ۱ باشد و در غیر اینصورت درخت نامتوازن[2] است. بنابراین هر یک از رئوس میانی درخت طبق ویژگی توازن ارتفاع باید متوازن باشند.

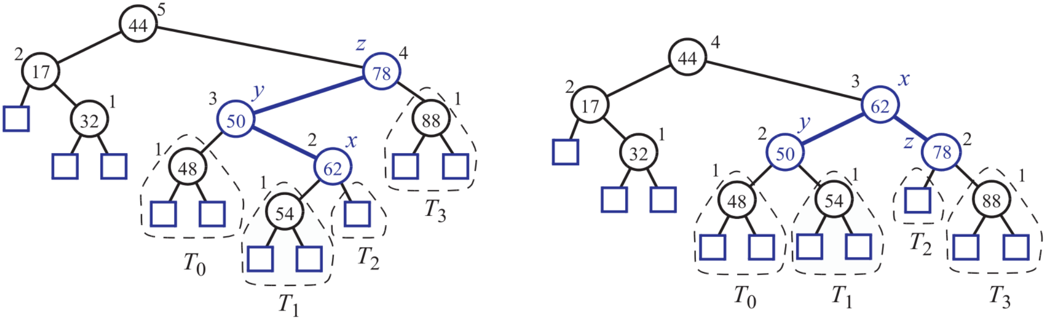
[1] balanced
[2] unbalanced

درج در درخت ای‌وی‌ال

- فرض کنیم T یک درخت ای‌وی‌ال باشد. پس از عملیات درج در درخت T ، ارتفاع برخی از رئوس درخت T ممکن است افزایش پیدا کند. رئوسی که ارتفاع آنها تغییر می‌کند بر روی مسیری از T از عنصر درج شده w تا ریشه درخت T هستند و این رئوس تنها رئوسی هستند که ممکن است نامتوازن شده باشند.
- اگر چنین اتفاقی بیافتد، درخت T دیگر ای‌وی‌ال نیست و باید آن را مجددا متوازن کنیم.

درج در درخت ای‌وی‌ال

- در شکل سمت چپ پس از درج رأسی با کلید ۵۴ ، رئوس حاوی کلید ۷۸ و ۴۴ نامتوازن شده‌اند. در شکل سمت راست درخت مجدداً به حالت متوازن در آمده است.



درج در درخت ای‌وی‌ال

- توازن رئوس در درخت T را با یک استراتژی جستجو و ترمیم [1] بازیابی می‌کنیم.
- فرض کنید z اولین رأس ناموتوازن باشد که با حرکت از رأس w به سمت ریشه به آن برخورد می‌کنیم.
- همچنین فرض کنید y فرزندی از z با ارتفاع بیشتر باشد. توجه کنید که y باید یکی از اجداد w باشد.
- همچنین فرض کنید x فرزندی از y با ارتفاع بیشتر باشد. در اینجا نیز x باید یکی از اجداد w باشد. بنابراین رأس x یکی از نوادگان z است و البته ممکن است همان w باشد.
- رأس z به علت درج در زیر درخت با ریشه y ناموتوازن شده است و بنابراین ارتفاع y دو واحد بزرگ‌تر از همزادش است.

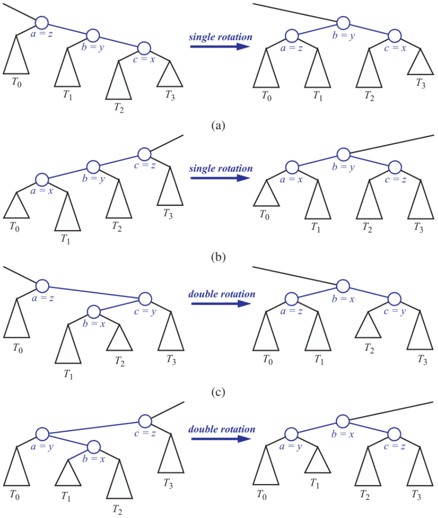
[1] search and repair

درج در درخت ای‌وی‌ال

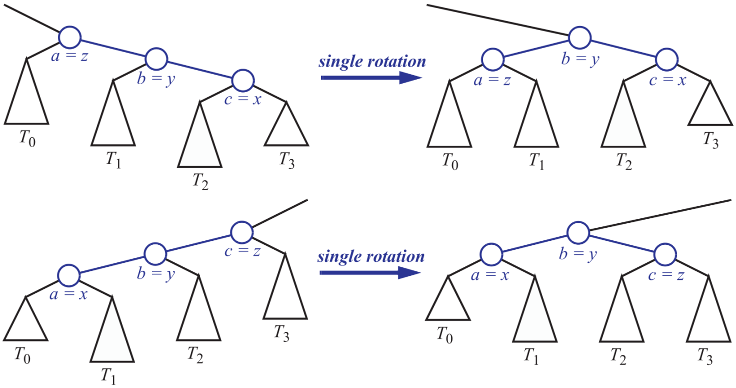
- حال با استفاده از الگوریتمی که شرح داده خواهد شد، زیر درخت با ریشه z را متوازن می‌کنیم.
- این تابع به طور موقت رئوس x و y و z را به a و b و c تغییر نام می‌دهد، به طوری که در یک پیمایش میان ترتیب از درخت T رأس a قبل از b و رأس b قبل از c قرار بگیرد.

درج در درخت ای‌وی‌ال

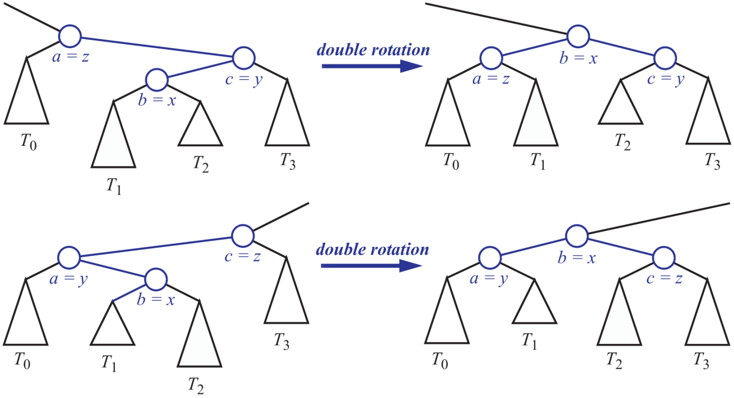
- رئوس x و y و z به چهار حالت زیر ممکن است به a و b و c نگاشت شوند.



درج در درخت ای‌وی‌ال



درج در درخت ای‌وی‌ال



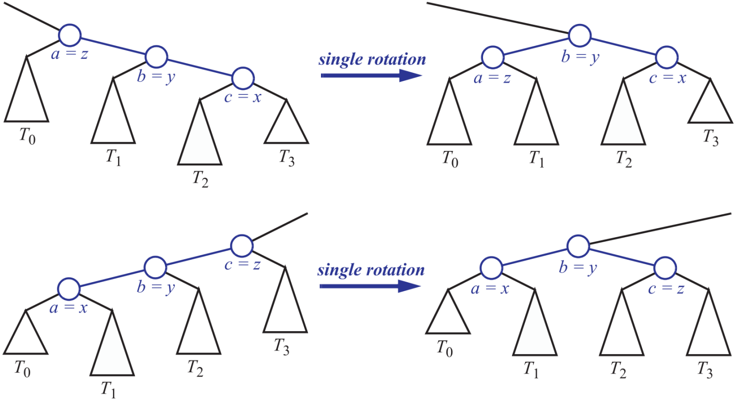
درج در درخت ای‌وی‌ال

- برای بازگرداندن توازن درخت، الگوریتم تغییر ساختار (`restructure`) درخت را به نحوی تغییر می‌دهد که z با b جایگزین شود به طوری که فرزندان آن a و c و فرزندان a و c فرزندان قبلی x و y و z باشند و ترتیب رئوس در پیمایش میان ترتیب حفظ شود.
- عملیات تغییر درخت T با استفاده از این تغییر ساختار معمولا دوران [1] نامیده می‌شود، زیرا از لحاظ بصری به نظر یک دوران (چرخش) در زیر درخت مربوطه اعمال شده است.

[1] rotation

درج در درخت ای‌وی‌ال

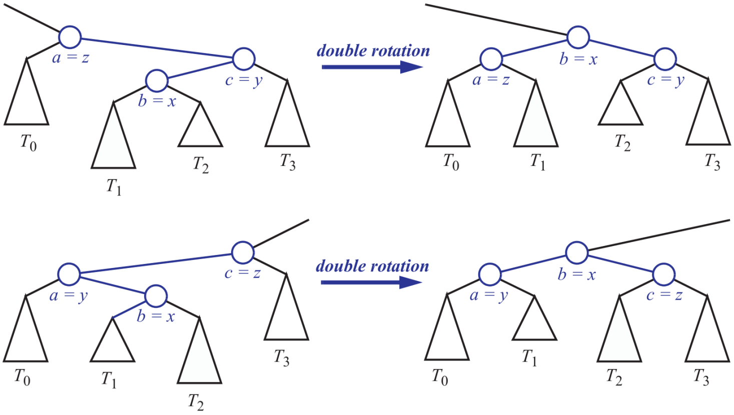
- اگر $y = b$ باشد، این تغییر ساختار دوران تکی [1] نامیده می‌شود، به طوری‌که به نظر می‌رسد y بر روی z چرخیده است.



[1] single rotation

درج در درخت ای‌وی‌ال

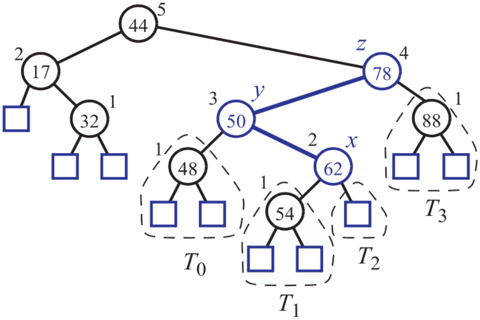
- اگر $b = x$ باشد، عملیات تغییر ساختار دوران دوتایی [1] نامیده می‌شود، به طوری‌که به نظر می‌رسد x بر روی y و سپس بر روی z چرخیده است.

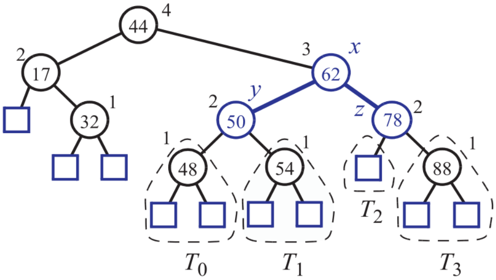


[1] double rotation

درج در درخت ای‌وی‌ال

- در مثال زیر یک دوران دوتایی انجام شده است.





درج در درخت ای‌وی‌ال

- در اینجا از یک تابع برای اعمال چهار نوع دوران استفاده کرده‌ام.

Algorithm restructure

```
function RESTRUCTURE(x)
 ▷ Input : A node x of a binary search tree T that has both a parent y
 and a grandparent z
 ▷ Output : Tree T after a trinode restructuring (which corresponds to a
 single or double rotation) involving nodes x, y, and z
1: Let (a, b, c) be a left-to-right (inorder) listing of the nodes x, y,
   and z, and let (T0 , T1 , T2 , T3 ) be a left-to-right (inorder) listing
   of the four subtrees of x, y, and z not rooted at x, y, or z.
2: Replace the subtree rooted at z with a new subtree rooted at b.
3: Let a be the left child of b and let T0 and T1 be the left and right
   subtrees of a, respectively.
4: Let c be the right child of b and let T2 and T3 be the left and right
   subtrees of c, respectively.
```

درج در درخت ای‌وی‌ال

- تابع تغییر ساختار رابطه فرزند–پدر را در تعداد ثابتی از رئوس تغییر می‌دهد به طوری که ترتیب آنها در پیمایش میان ترتیب حفظ می‌شود.
- علاوه بر حفظ ترتیب رئوس، تغییر ساختار به نحوی انجام می‌شود که ارتفاع تعدادی از رئوس تغییر کرده و درخت مجدداً متوازن می‌شود.

درج در درخت ای‌وی‌ال

- توجه کنید تابع `restructure(x)` فراخوانی می‌شود زیرا z پدربزرگ x نامتوازن شده است. این عدم توازن به علت این است که ارتفاع یکی از فرزندان x افزایش پیدا کرده که باعث شده یکی از فرزندان z نسبت به فرزند دیگر z ارتفاع بیشتری داشته باشد.

- با استفاده از دوران فرزند x با ارتفاع بیشتر به بالا و فرزند z با ارتفاع کمتر به پایین منتقل می‌شود، بنابراین بعد از تغییر ساختار همه رئوس در زیر درخت با ریشه b متوازن می‌شود.

- بنابراین ویژگی توازن ارتفاع در رئوس x و y و z به صورت محلی [1] برقرار می‌شود.

- از آنجایی که بعد از انجام عملیات درج، زیر درخت با ریشه b جایگزین زیر درختی می‌شود که قبلا دارای ریشه z بود، همه اجداد z که نامتوازن بودند مجدداً متوازن می‌شوند. در نتیجه درخت به صورت عمومی [2] متوازن می‌شود.

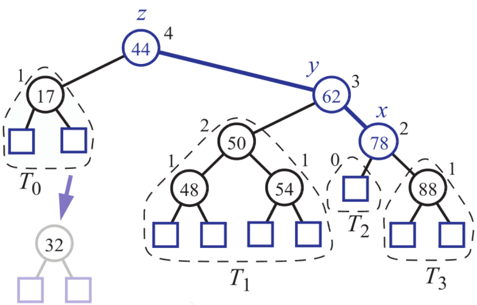
---

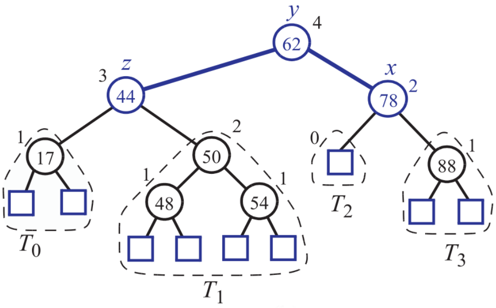
[1] locally
[2] globally

حذف از درخت ای‌وی‌ال

- برای حذف یک رأس از درخت ای‌وی‌ال ابتدا عملیات حذف معمولی برروی درخت جستجوی دودویی را انجام می‌دهیم. در هنگام حذف نیز ممکن است توازن درخت نقض شود.

- در شکل سمت چپ حذف رأس با کلید ۳۲ باعث عدم توازن شده که در شکل سمت راست این توازن مجدداً بازیابی شده است.





حذف از درخت ای‌وی‌ال

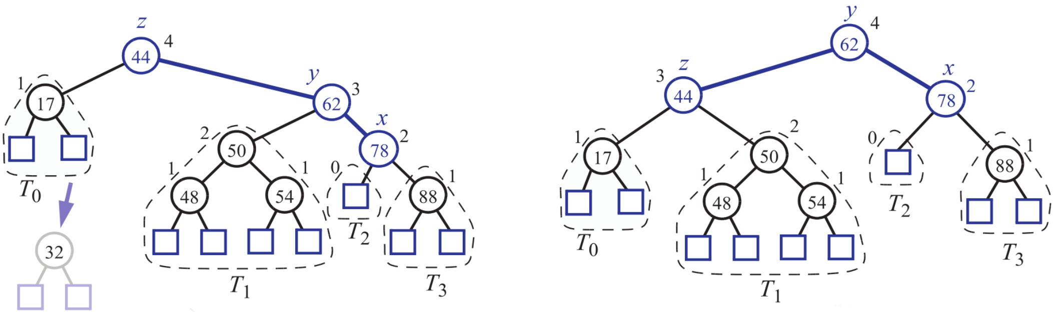
- فرض کنید z اولین رأس نامتوازن باشد که با بالا رفتن از رأس حذف شده w به سمت ریشه درخت T با آن برخورد می‌کنیم.
- فرض کنید y فرزند z با ارتفاع بیشتر باشد. رأس y فرزندی از z است که جد w نیست.
- فرض کنید x فرزند y باشد که به صورت زیر تعریف شده است : اگر یکی از فرزندان y ارتفاع بیشتری نسبت به دیگری داشته باشد، x فرزند y با ارتفاع بیشتر است. در غیراینصورت اگر هر دوی فرزندان y ارتفاع برابر داشته باشند، x فرزند y در طرف y است، بدین معنی که اگر y فرزند چپ باشد، x فرزند چپ y است و اگر y فرزند راست باشد، x فرزند راست y است.

حذف از درخت ای‌وی‌ال

- سپس عملیات `restructure(x)` را انجام می‌دهیم که ویژگی توازن ارتفاع را به صورت محلی در زیر درخت با ریشه z که اکنون دارای ریشه b است بازیابی می‌کند.

حذف از درخت ای‌وی‌ال

- شکل زیر یک مثال از بازیابی توازن را نشان می‌دهد.



حذف از درخت ای‌وی‌ال

- متاسفانه این تغییر ساختار ممکن است ارتفاع زیر درخت با ریشه b را به میزان یک واحد کاهش دهد، که باعث می‌شود اجداد b نامتوازن شوند. بنابراین پس از متوازن کردن z در درخت T به سمت بالا حرکت می‌کنیم و رئوس نامتوازن را پیدا می‌کنیم. اگر به یک رأس نامتوازن برخورد کردیم، عملیات تغییر ساختار را مجدداً انجام می‌دهیم و مجدداً در درخت به سمت بالا حرکت می‌کنیم.

- از آنجایی که ارتفاع درخت T برابر با $O(\lg n)$ است، تغییر ساختار درخت در هنگام عملیات حذف در زمان $O(\lg n)$ انجام می‌شود.

حذف از درخت ای‌وی‌ال

- پیچیدگی زمانی عملیات درج و حذف و جستجو در درخت ای‌وی‌ال برابر با $O(\lg n)$ است.

درخت‌های قرمز-سیاه

- قبلاً نشان دادیم که درخت جستجوی دودویی به ارتفاع h عملیات جستجو و درج و حذف را در زمان $O(h)$ انجام می‌دهد. بنابراین این عملیات می‌توانند سریع باشند، اگر ارتفاع درخت کم باشد. در بدترین حالت اگر ارتفاع درخت بسیار زیاد باشد زمان اجرای عملیات در درخت و لیست پیوندی یکسان خواهد بود.

- درخت‌های قرمز-سیاه [1] یکی از انواع درخت‌های جستجو هستند که متوازن [2] اند و عملیات روی مجموعه‌های پویا را در بدترین حالت در زمان $O(\lg n)$ انجام می‌دهند.

- درخت قرمز-سیاه به گونه‌ای طراحی شده است که در درج و حذف تعداد کمتری دوران انجام می‌دهد و در نتیجه در درج و حذف نسبت به درخت ای‌وی‌ال سریع‌تر است. با این که ارتفاع هر دو درخت ای‌وی‌ال و قرمز-سیاه با n رأس $O(\lg n) = c \lg n$ است، اما درخت ای‌وی‌ال ضریب c کوچکتری دارد یا به عبارت دیگر توازن بیشتری دارد و در نتیجه در جستجو سریع‌تر عمل می‌کند.

---

[1] red-black trees

[2] balanced

درخت‌های قرمز–سیاه

- یک درخت قرمز–سیاه یک درخت جستجوی دودویی است با یک بیت حافظه اضافی به ازای هر رأس. در این بیت رنگ رأس تعیین می‌شود که می‌تواند قرمز یا سیاه باشد.
- با محدود کردن رنگ رئوس بر روی هر مسیر ساده از ریشه به یک برگ، درخت‌های قرمز–سیاه اطمینان حاصل می‌کنند که هیچ مسیری بیشتر از دو برابر مسیر دیگر طول ندارد و بنابراین درخت تقریبا متوازن است.
- ارتفاع یک درخت قرمز–سیاه با n کلید حداکثر $2\lg(n+1)$ است که برابر است با $O(\lg n)$.
- هر رأس درخت دارای ویژگی‌های color ، key ، left ، right و p است. اگر یک رأس دارای پدر یا فرزند چپ یا راست نباشد، اشاره‌گرهای مربوطه تهی NIL خواهند بود.

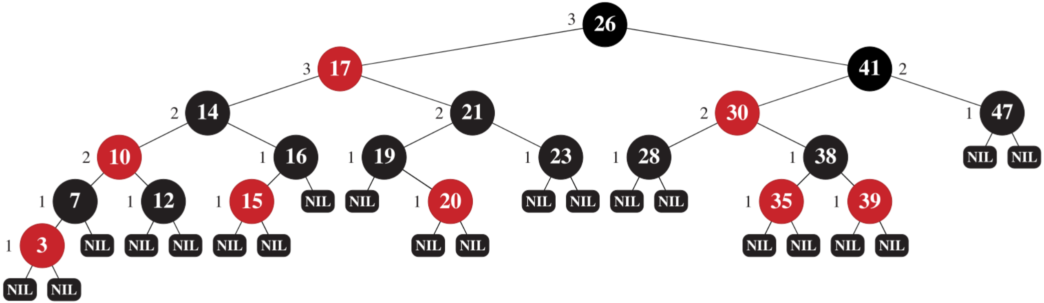
درخت‌های قرمز–سیاه

- یک درخت قرمز–سیاه یک درخت جستجوی دودویی است که ویژگی‌های زیر را داراست :

۱. ویژگی ریشه : ریشه درخت سیاه است.
۲. ویژگی برگ‌ها : هر برگ سیاه است.
۳. ویژگی رئوس قرمز : اگر یک رأس قرمز باشد، هر دو فرزند آن سیاه هستند.
۴. ویژگی ارتفاع سیاه : به ازای هر رأس، تعداد رئوس سیاه از همهٔ مسیرهای ساده از آن رأس به برگ‌های درخت برابر هستند.

درخت‌های قرمز–سیاه

- شکل زیر یک مثال از درخت قرمز–سیاه را نشان می‌دهد.



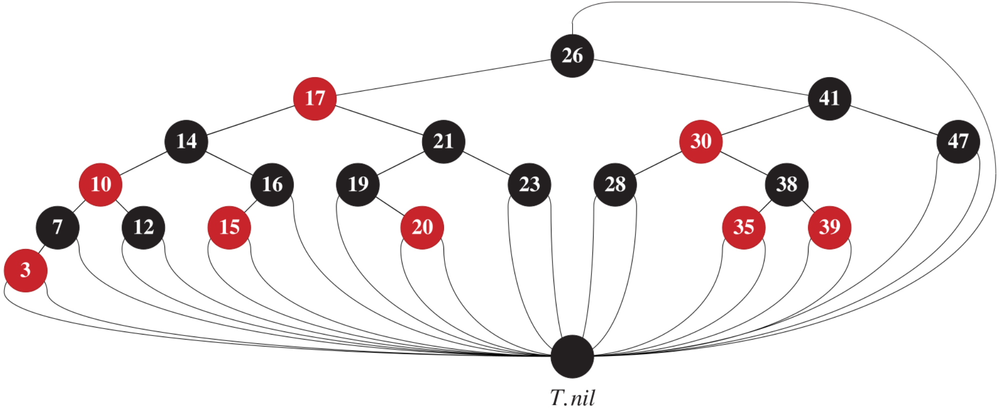
درخت‌های قرمز-سیاه

- برای مدیریت شرایط مرزی و صرفه‌جویی در حافظه در درخت قرمز-سیاه از یک نگهبان [1] برای ذخیره‌سازی رئوس تهی NIL استفاده می‌کنیم. برای درخت T ، نگهبان T.nil شیئی است که یک رأس معمولی از درخت است ، رنگ آن سیاه است، و مقدار p ، left ، right و key در آن تهی است.

[1] sentinel

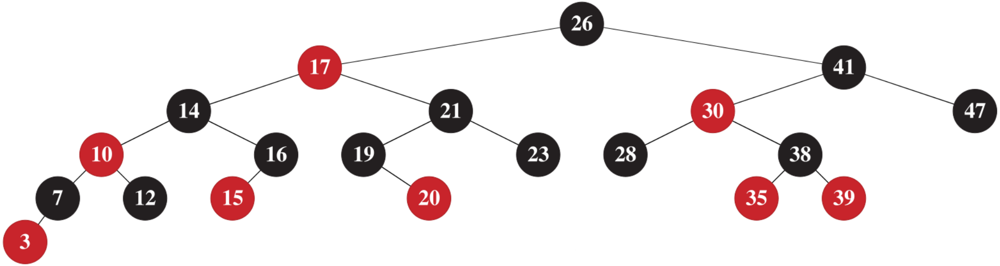
درخت‌های قرمز-سیاه

- در شکل زیر، همه اشاره‌گرها به NIL با اشاره‌گری به T.nil جایگزین شده‌اند. همهٔ برگ‌ها و پدر ریشه تهی هستند.



درخت‌های قرمز-سیاه

- جهت سهولت نمایش درخت، برگ‌ها را حذف می‌کنیم و درخت قرمز-سیاه را تنها با رئوس میانی نشان می‌دهیم. شکل زیر نمایش درخت قرمز-سیاه بدون برگ‌های تهی است.

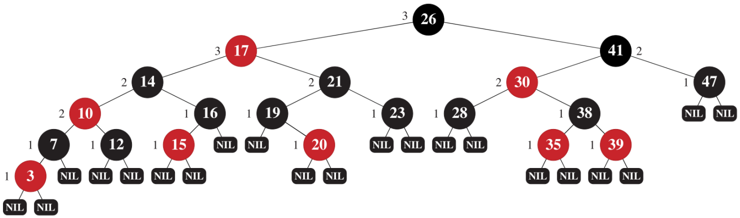


درخت‌های قرمز-سیاه

- تعداد رئوس سیاه برروی هر مسیر از رأس x بدون احتساب خود رأس تا یکی از برگ‌ها را ارتفاع سیاه[1] رأس می‌نامیم و با $bh(x)$ نمایش می‌دهیم.

- توجه کنید که همه مسیرهای ساده از یک رأس تا هریک از برگ‌ها تعداد رأس سیاه برابر دارند.

- ارتفاع سیاه یک درخت قرمز-سیاه ارتفاع سیاه ریشه آن است. ارتفاع سیاه هر رأس در شکل زیر درکنار آن درج شده است.



[1] black-height

درخت‌های قرمز-سیاه

- قضیه : ارتفاع درخت قرمز-سیاه با n رأس حداکثر $2\lg(n+1)$ است.
- اثبات : ابتدا نشان می‌دهیم که یک زیردرخت با ریشه x حداقل $2^{bh(x)} - 1$ رأس میانی دارد. این ادعا را توسط استقرا بر روی ارتفاع x اثبات می‌کنیم.
- اگر ارتفاع x برابر با صفر باشد، رأس x یک برگ است (T.nil) و زیردرخت با ریشه x دارای $2^{bh(x)} - 1 = 2^0 - 1 = 0$ رأس میانی است.

درخت‌های قرمز-سیاه

- برای اثبات گام استقرا، رأس x با ارتفاع مثبت را در نظر بگیرید. رأس x دارای دو فرزند است که هر دو یا یکی از آنها می‌تواند برگ باشد. ارتفاع سیاه یک فرزند سیاه، یک واحد کمتر از ارتفاع سیاه x است، اما ارتفاع سیاه یک فرزند قرمز، برابر با ارتفاع سیاه x است.
- بنابراین ارتفاع سیاه یک فرزند سیاه برابر با $bh(x) - 1$ است و ارتفاع سیاه یک فرزند قرمز برابر با $bh(x)$ است.

درخت‌های قرمز-سیاه

- از آنجایی که ارتفاع یک فرزند x کمتر از ارتفاع x است، می‌توانیم فرض استقرا را اعمال کنیم و نتیجه بگیریم که هر فرزند حداقل $2^{bh(x)-1} - 1$ رأس میانی دارد. بنابراین زیردرخت با ریشه x شامل حداقل $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1$ رأس میانی است که برابر با $2^{bh(x)} - 1$ است.
- حال فرض کنید h ارتفاع درخت باشد. بنابر ویژگی‌های درخت قرمز-سیاه حداقل نصف رئوس برروی هر مسیر ساده از ریشه به یک برگ بدون در نظر گرفتن خود ریشه باید سیاه باشند. بنابراین ارتفاع سیاه ریشه باید حداقل $h/2$ باشد و بنابراین $n \geqslant 2^{h/2} - 1$.
- بنابراین به دست می‌آوریم $h \leqslant 2\lg(n+1)$.

درخت‌های قرمز-سیاه

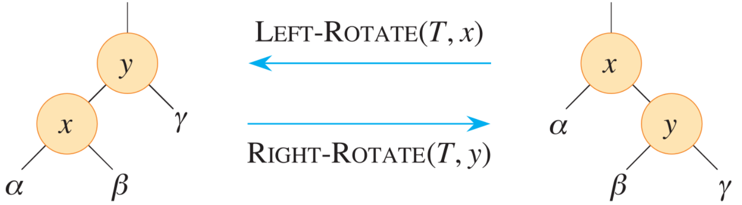
- از این قضیه نتیجه می‌گیریم عملیات مجموعه‌های پویای جستجو (Search) ، کمینه (Minimum) ، بیشینه (Maximum) ، محاسبه رأس بعدی (Successor) ، و رأس قبلی (Predecessor) در زمان $O(\lg n)$ در درخت قرمز-سیاه انجام می‌شوند، زیرا هر کدام در زمان $O(h)$ در درخت جستجوی دودویی انجام می‌شوند و درخت قرمز-سیاه با n رأس، یک درخت جستجوی دودویی با ارتفاع $O(\lg n)$ است.
- اگرچه عملیات درج (Tree-Insert) و حذف (Tree-Delete) در درخت جستجوی دودویی در زمان $O(\lg n)$ قابل انجام‌اند، اما در درخت قرمز-سیاه نمی‌توانیم از آنها استفاده کنیم زیرا ویژگی درخت قرمز-سیاه را حفظ نمی‌کنند. در ادامه نشان خواهیم داد که چگونه می‌توانیم عملیات درج و حذف را درخت قرمز-سیاه در زمان $O(\lg n)$ پیاده‌سازی کنیم.

درخت‌های قرمز-سیاه

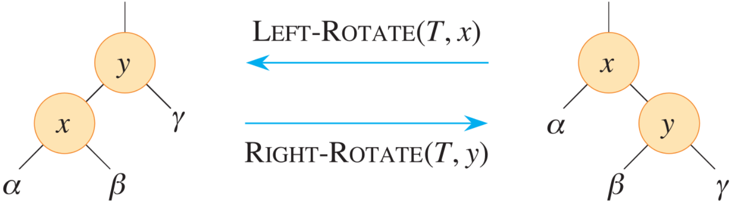
- عملیات درج و حذف برروی درخت قرمز-سیاه، در زمان $O(\lg n)$ اجرا می‌شوند، اما از آنجایی که این عملیات درخت را تغییر می‌دهند، ممکن است ویژگی درخت قرمز-سیاه را حفظ نکنند. برای حفظ ویژگی درخت قرمز-سیاه لازم است تغییراتی در این عملیات اعمال کنیم.
- حفظ ویژگی درخت قرمز-سیاه توسط دوران [1] انجام می‌شود که یک عملیات محلی در یک درخت جستجو است.

[1] rotation

- شکل زیر دو نوع دوران را نشان می‌دهد : دوران چپ و دوران راست.



- فرض کنید رأس x یک فرزند سمت راست به نام y دارد که برابر با T.nil نیست.
- دوران چپ[1] زیر درخت اصلی با ریشه x را به گونه‌ای تغییر می‌دهد که ریشه زیردرخت برابر با y شود و x فرزند چپ رأس y شود و فرزند چپ قبلی y (که در شکل β است) فرزند راست x شود.



- شبه کد Left–Rotate در زیر فرض می‌کند که $x.right \neq T.nil$ است و پدر ریشه برابر با T.nil است.

[1] left rotation

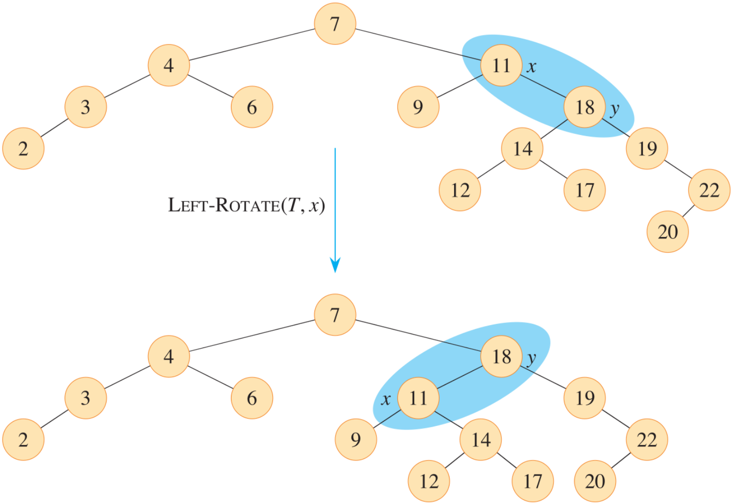
درخت‌های قرمز–سیاه

Algorithm Left Rotate

```
function LEFT-ROTATE(T,x)
y = x.right
x.right = y.left  ▷ turn y's left subtree into x's right subtree
if y.left ≠ T.nil then  ▷ if y's left subtree is not empty ...
    y.left.p = x     ▷ ... then x becomes the parent of the subtree's root
y.p = x.p  ▷ x's parent becomes y's parent
if x.p == T.nil then  ▷ if x was the root ...
    T.root = y     ▷ ... then y becomes the root
else if x == x.p.left then  ▷ otherwise, if x was a left child
    x.p.left = y     ▷ ... then y becomes a left child
else  x.p.right = y  ▷ otherwise, x a right child, and now y is
y.left = x  ▷ make x become y's left child
x.p = y
```

درخت‌های قرمز–سیاه

- شکل زیر یک مثال از اعمال Left-Rotate را برروی درخت جستجوی دودویی نشان می‌دهد.



درخت‌های قرمز-سیاه

- تابع Right-Rotate متقارن با تابع Left-Rotate است. هر دوی این توابع در زمان $O(1)$ اجرا می‌شود.

درج در درخت قرمز–سیاه

- تابع RB–Insert رأس z را در درخت T شبیه درخت جستجوی دودویی درج می‌کند و رنگ رأس z را قرمز می‌کند. سپس از تابع RB–Insert–Fixup استفاده می‌کند تا رنگ رئوس را تصحیح کند.

درج در درخت قرمز–سیاه

Algorithm RB Insert

```
function RB-INSERT(T,z)
x = T.root  ▷ node being compared with z
y = T.nil  ▷ y will be parent of z
while x ≠ T.nil do  ▷ descend until reaching the sentinel
   y = x
   if z.key < x.key then
      x = x.left
   else  x = x.right
z.p = y  ▷ found the location - insert z with parent y
if y == T.nil then
   T.root = z     ▷ tree T was empty
else if z.key < y.key then
   y.left = z
else  y.right = z
z.left = T.nil  ▷ both off z's children are the sentinel
z.right = T.nil
z.color = RED  ▷ the new node starts out red
RB-Insert-Fixup (T,z)  ▷ correct any violations of red-black properties
```

درج در درخت قرمز–سیاه

- تابع RB–Insert چند تفاوت با تابع Tree–Insert دارد :
- مقادیر NIL با T.nil جایگزین شده‌اند. همچنین در خطوط ۱۴ و ۱۵ از تابع RB–Insert مقادیر z.left و z.right برابر با T.nil قرار می‌گیرند، در صورتی که این مقادیر در تابع درج در درخت برابر با NIL بودند.
- در خط ۱۶ رنگ رأس جدید z برابر با قرمز قرار می‌گیرد.
- از آنجایی که قرمز کردن رأس z ممکن است باعث شود ویژگی درخت قرمز–سیاه نقض شود، در خط ۱۷ تابع RB–Insert–Fixup فراخوانی می‌شود تا ویژگی درخت قرمز–سیاه حفظ شود.

درج در درخت قرمز–سیاه

Algorithm RB Insert Fixup

```
   function RB-INSERT-FIXUP(T,z)
1:  while z.p.color == RED do
2:     if z.p == z.p.p.left then          ▷ is z's parent a left child?
3:        y = z.p.p.right           ▷ y is z's uncle
4:        if y.color = RED then          ▷ are z's parent and uncle both red?
5:           z.p.color = BLACK
6:           y.color = BLACK
7:           z.p.p.color = RED
8:           z = z.p.p
9:        else
10:          if z == z.p.right then
11:             z = z.p
12:             Left-Rotate (T,z)
13:          z.p.color = BLACK
14:          z.p.p.color = RED
15:          Right-Rotate (T, z.p.p)
16:    else    ▷ same as lines 3-15 , but with "right" and "left" exchanged
17:       y = z.p.p.left
18:       if y.color == RED then
19:          z.p.color = BLACK
20:          y.color = BLACK
21:          z.p.p.color = RED
22:          z = z.p.p
23:       else
24:          if z == z.p.left then
25:             z = z.p
26:             Right-Rotate (T,z)
27:          z.p.color = BLACK
28:          z.p.p.color = RED
29:          Left-Rotate (T,z.p.p)
30: T.root.color = BLACK
```

درج در درخت قرمز–سیاه

- برای این‌که بفهمیم تابع RB–Insert–Fixup چگونه عمل می‌کند، تابع را در سه‌گام بررسی می‌کنیم.
- ابتدا تعیین می‌کنیم با درج کردن رأس z و رنگ کردن آن با قرمز، چگونه ویژگی‌های درخت قرمز–سیاه نقض می‌شوند.
- سپس بررسی می‌کنیم هدف از حلقه خطوط ۱ تا ۲۹ چیست.
- در پایان سه حالت حلقه اصلی تابع را بررسی می‌کنیم.

- در توصیف درخت قرمز–سیاه نیاز داریم درمورد همزاد رأس پدر صحبت کنیم. از واژهٔ عمو [1] برای نام بردن از رأس همزاد پدر استفاده می‌کنیم.

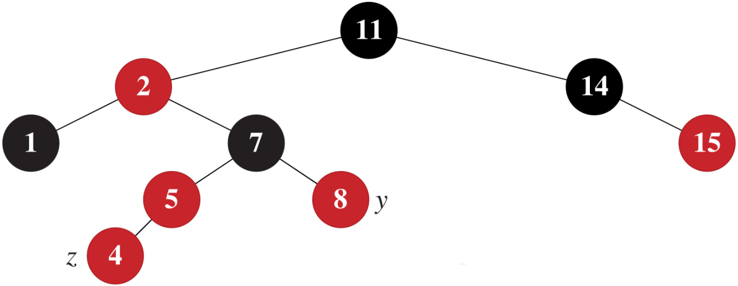
[1] uncle

درج در درخت قرمز-سیاه

- در اینجا بررسی می‌کنیم چگونه ویژگی‌های درخت قرمز-سیاه ممکن است نقض شوند.
- همچنین ویژگی برگ‌ها که ملزم می‌کند هر برگ سیاه باشد حفظ می‌شود، زیرا هر دو فرزند رأس جدید قرمز اضافه شده T.nil هستند که سیاه است.
- ویژگی ارتفاع سیاه ملزم می‌کند که تعداد رئوس سیاه در هر مسیر ساده از یک رأس برابرند. این ویژگی نیز حفظ می‌شود، زیرا رأس z که قرمز است جایگزین یک برگ سیاه می‌شود و فرزندان آن هر دو سیاه هستند.
- تنها ویژگی ریشه و ویژگی رئوس قرمز ممکن است نقض شوند. طبق ویژگی ریشه، ریشه درخت باید سیاه باشد و طبق ویژگی رئوس قرمز، یک رأس قرمز نمی‌تواند فرزند قرمز داشته باشد. ویژگی ریشه ممکن است نقض شود اگر z ریشه باشد و ویژگی رئوس قرمز ممکن است نقض شود اگر پدر رأس z قرمز باشد.

درج در درخت قرمز–سیاه

- شکل زیر نقض ویژگی رئوس قرمز را بعد از درج رأس z نشان می‌دهد.



درج در درخت قرمز–سیاه

- حلقه خطوط ۱ تا ۲۹ دو احتمال را بررسی می‌کند : خطوط ۳ تا ۱۵ وضعیتی را بررسی می‌کند که در آن پدر رأس z یعنی z.p فرزند چپ پدربزرگ z یعنی z.p.p است و خطوط ۱۷ تا ۲۹ وضعیتی که در آن z.p فرزند راست z.p.p است.

Algorithm RB Insert Fixup

```
    function RB-INSERT-FIXUP(T,z)
1:  while z.p.color == RED do
2:     if z.p == z.p.p.left then        ▷ is z's parent a left child?
3:        y = z.p.p.right           ▷ y is z's uncle
4:        if y.color = RED then         ▷ are z's parent and uncle both red?
5:           z.p.color = BLACK          ▷ Case 1
6:           y.color = BLACK
7:           z.p.p.color = RED
8:           z = z.p.p
9:        else
10:          if z == z.p.right then
11:             z = z.p                 ▷ Case 2
12:             Left-Rotate (T,z)
13:          z.p.color = BLACK          ▷ Case 3
14:          z.p.p.color = RED
15:          Right-Rotate (T, z.p.p)
```

درج در درخت قرمز-سیاه

- خطوط ۱۷ تا ۲۹ وضعیتی را بررسی می‌کنند که در آن z.p فرزند راست z.p.p است.

Algorithm RB Insert Fixup

```
    function RB-Insert-Fixup(T,z)
1:  while z.p.color == RED do
2:     if z.p == z.p.p.left then      ▷ is z's parent a left child?
3:        ...
15:       ...
16:    else    ▷ same as lines 3-15 , but with "right" and "left" exchanged
17:       y = z.p.p.left
18:       if y.color == RED then
19:          z.p.color = BLACK
20:          y.color = BLACK
21:          z.p.p.color = RED
22:          z = z.p.p
23:       else
24:          if z == z.p.left then
25:             z = z.p
26:             Right-Rotate (T,z)
27:          z.p.color = BLACK
28:          z.p.p.color = RED
29:          Left-Rotate (T,z.p.p)
30: T.root.color = BLACK
```

درج در درخت قرمز‌ـسیاه

- در اینجا سه حالت را بررسی می‌کنیم. حالت ۱ در خطوط ۵ تا ۸ ، حالت ۲ در خطوط ۱۱ تا ۱۲ و حالت ۳ در خطوط ۱۳ تا ۱۵ بررسی می‌شوند.

Algorithm RB Insert Fixup

```
    function RB-INSERT-FIXUP(T,z)
1:  while z.p.color == RED do
2:     if z.p == z.p.p.left then      ▷ is z's parent a left child?
3:        y = z.p.p.right             ▷ y is z's uncle
4:        if y.color = RED then       ▷ are z's parent and uncle both red?
5:           z.p.color = BLACK        ▷ Case 1
6:           y.color = BLACK
7:           z.p.p.color = RED
8:           z = z.p.p
9:        else
10:          if z == z.p.right then
11:             z = z.p               ▷ Case 2
12:             Left-Rotate (T,z)
13:          z.p.color = BLACK        ▷ Case 3
14:          z.p.p.color = RED
15:          Right-Rotate (T, z.p.p)
```

درج در درخت قرمز–سیاه

- حالت ۱ از حالت‌های ۲ و ۳ به علت رنگ عموی رأس z یعنی y متفاوت است. در خط ۳ اشاره‌گر y به عموی رأس z یعنی $z.p.p.right$ اشاره می‌کند و خط ۴ رنگ رأس y را بررسی می‌کند. اگر y قرمز باشد، حالت ۱ اجرا می‌شود. در غیراینصورت حالت ۲ و ۳ در نظر گرفته می‌شوند.

Algorithm RB Insert Fixup

```
    function RB-Insert-Fixup(T,z)
1:  while z.p.color == RED do
2:     if z.p == z.p.p.left then      ▷ is z's parent a left child?
3:        y = z.p.p.right             ▷ y is z's uncle
4:        if y.color = RED then       ▷ are z's parent and uncle both red?
5:           z.p.color = BLACK        ▷ Case 1
6:           y.color = BLACK
7:           z.p.p.color = RED
8:           z = z.p.p
9:        else
10:          if z == z.p.right then
11:             z = z.p               ▷ Case 2
12:             Left-Rotate (T,z)
13:          z.p.color = BLACK        ▷ Case 3
14:          z.p.p.color = RED
15:          Right-Rotate (T, z.p.p)
```

درج در درخت قرمز–سیاه

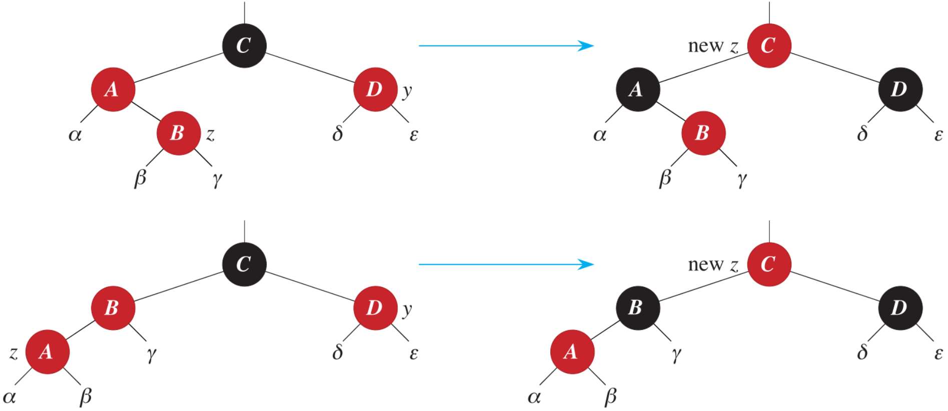
- در هر سه حالت، پدربزرگ z یعنی $z.p.p$ سیاه است، زیرا $z.p$ قرمز است و ویژگی رئوس قرمز درخت نقض می‌شود اگر z و $z.p$ قرمز باشند.

Algorithm RB Insert Fixup

```
function RB-INSERT-FIXUP(T,z)
while z.p.color == RED do
   if z.p == z.p.p.left then     ▷ is z’s parent a left child?
      y = z.p.p.right            ▷ y is z’s uncle
      if y.color = RED then      ▷ are z’s parent and uncle both red?
         z.p.color = BLACK       ▷ Case 1
         y.color = BLACK
         z.p.p.color = RED
         z = z.p.p
      else
         if z == z.p.right then
            z = z.p              ▷ Case 2
            Left-Rotate (T,z)
         z.p.color = BLACK       ▷ Case 3
         z.p.p.color = RED
         Right-Rotate (T, z.p.p)
```

درج در درخت قرمز–سیاه

- حالت ۱ : عموی z قرمز است.
- شکل زیر حالت ۱ (خطوط ۵ تا ۸) را نشان می‌دهد وقتی که $z.p$ و y هر دو قرمز هستند.

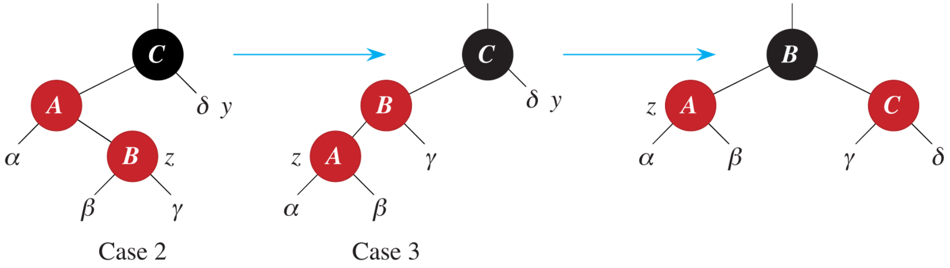


درج در درخت قرمز–سیاه

- از آنجایی که پدربزرگ z یعنی $z.p.p$ سیاه است، سیاهی آن می‌تواند یک سطح به پایین به $z.p$ و y منتقل شود که مشکل z و $z.p$ را که هر دو قرمز هستند حل می‌کند. در این‌صورت پدربزرگ z قرمز می‌شود. حلقه با $z.p.p$ به عنوان رأس جدید z تکرار می‌شود، و اشاره‌گر z دو سطح به بالا حرکت می‌کند.

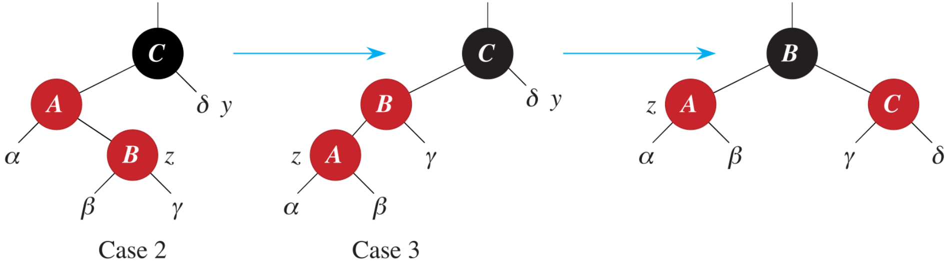
درج در درخت قرمز–سیاه

- حالت ۲ : رنگ y عموی z سیاه و z یک فرزند راست است.
- حالت ۳ : رنگ y عموی z سیاه و z یک فرزند چپ است.
- در حالت‌های ۲ و ۳، رنگ y عموی z سیاه است و رنگ پدر z قرمز است. دو حالت را در نظر می‌گیریم. رأس z یا فرزند راست یا فرزند چپ z.p است.
- خطوط ۱۱ و ۱۲ حالت ۲ را بررسی می‌کنند که همراه با حالت ۳ در شکل زیر نشان داده شده است.



درج در درخت قرمز-سیاه

- در حالت ۲، رأس z فرزند راست پدر خود است. یک دوران چپ این حالت را به حالت ۳ تبدیل می‌کند (خطوط ۱۳ تا ۱۵) که در آن رأس z یک فرزند چپ است.
- چون z و z.p هر دو قرمز هستند، دوران بر ارتفاع سیاه رئوس تأثیری نمی‌گذارد.
- چنانچه حالت ۳ مستقیما و یا از طریق حالت ۲ اجرا شود، رأس y عموی z سیاه است، زیرا در غیراینصورت حالت ۱ اجرا شده بود.

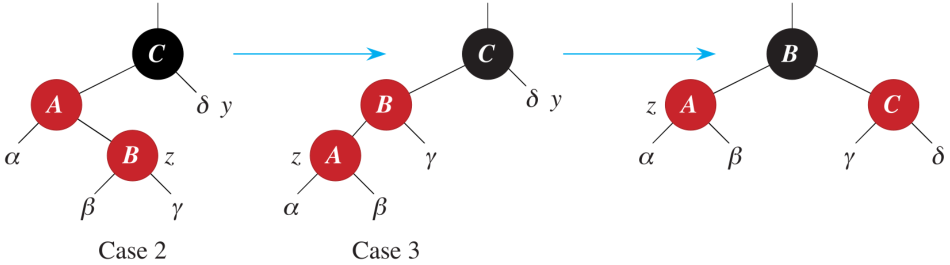


درج در درخت قرمز–سیاه

- علاوه بر این رأس z.p.p وجود دارد، زیرا این رأس در هنگامی که خطوط ۲ و ۳ اجرا شده‌اند، وجود داشته است و بعد از انتقال z یک سطح به سمت بالا در خط ۱۱ و سپس یک سطح به سمت پایین در خط ۱۲، رأس z.p.p تغییر نکرده است.

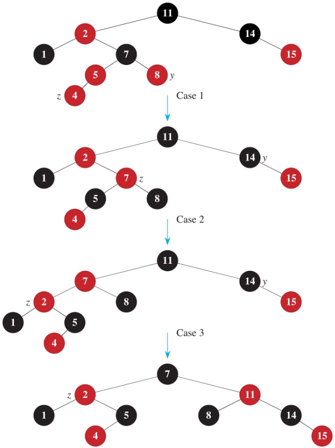
- حالت ۳ برخی رنگ‌ها را تغییر می‌دهد و یک دوران راست انجام می‌دهد، که ویژگی ارتفاع را حفظ می‌کند.

در این لحظه، هیچ دو رأس قرمزی پشت سرهم وجود ندارند. حلقه در تکرار بعدی در خط ۱ به پایان می‌رسد، زیرا z.p اکنون سیاه است.



درج در درخت قرمز–سیاه

- شکل زیر نشان می‌دهد تابع RB–Insert–Fixup چگونه برروی یک درخت قرمز–سیاه عمل می‌کند بسته به این که رأس پدر و عموی رأس z چه رنگی داشته باشند.



درج در درخت قرمز_سیاه

- تحلیل زمانی : چون ارتفاع یک درخت قرمز_سیاه با n رأس برابر با $O(\lg n)$ است، خطوط ۱ تا ۱۶ از تابع RB–Insert در زمان $O(\lg n)$ اجرا می‌شوند.
- در RB–Insert–Fixup حلقه تکرار می‌شود تنها اگر حالت ۱ اتفاق بیافتد، و در این‌صورت اشاره‌گر z دو سطح به سمت بالا حرکت می‌کند.
- بنابراین کل تعداد تکرار حلقه $O(\lg n)$ است و RB–Insert در کل در زمان $O(\lg n)$ اجرا می‌شود.
- علاوه بر این هیچگاه بیشتر از دو دوران انجام نمی‌شود، زیرا در صورتی که حالت ۲ یا ۳ اجرا شوند، حلقه خاتمه می‌یابد.

حذف از درخت قرمز–سیاه

- برای حذف یک رأس از درخت قرمز–سیاه باید حالت‌های زیادتری در نظر گرفته شود که در اینجا از آن صرف نظر می‌کنیم.
- حذف از درخت قرمز–سیاه نیز در زمان $O(\lg n)$ انجام می‌شود.