

به نام خدا

ساختمان داده

آرش شفيعی

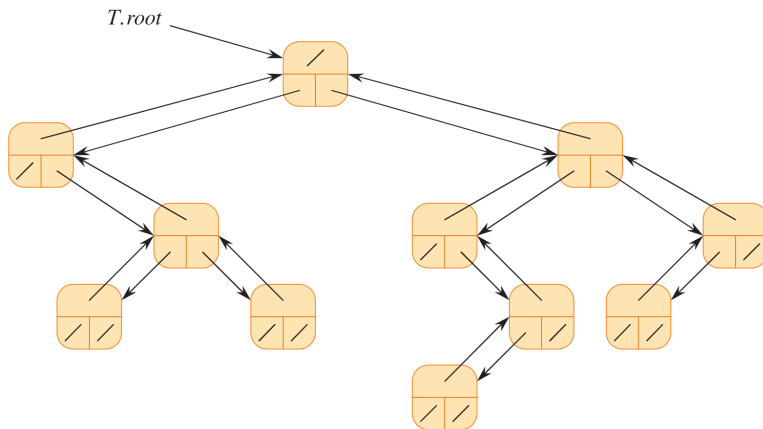


## درخت‌ها

- 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.

– 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.



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

---

## Data Structure Node

---

```
struct NODE
1: int key
2: Node * p  ▷ pointer to the parent node
3: Node * left  ▷ pointer to the left child
4: Node * right  ▷ pointer to the right child
```

---



---

## Data Structure Binary Tree

---

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

---

- اگر  $x.p = \text{NIL}$  باشد، آنگاه  $x$  ریشه درخت است. اگر  $x$  فرزند سمت چپ نداشته باشد، آنگاه  $x.\text{left} = \text{NIL}$  است و اگر فرزند سمت راست نداشته باشد،  $x.\text{right} = \text{NIL}$  است.
- ریشه درخت را با اشاره گر  $T.\text{root}$  مشخص می‌کنیم و اگر  $T.\text{root} = \text{NIL}$  باشد، آنگاه درخت خالی است.

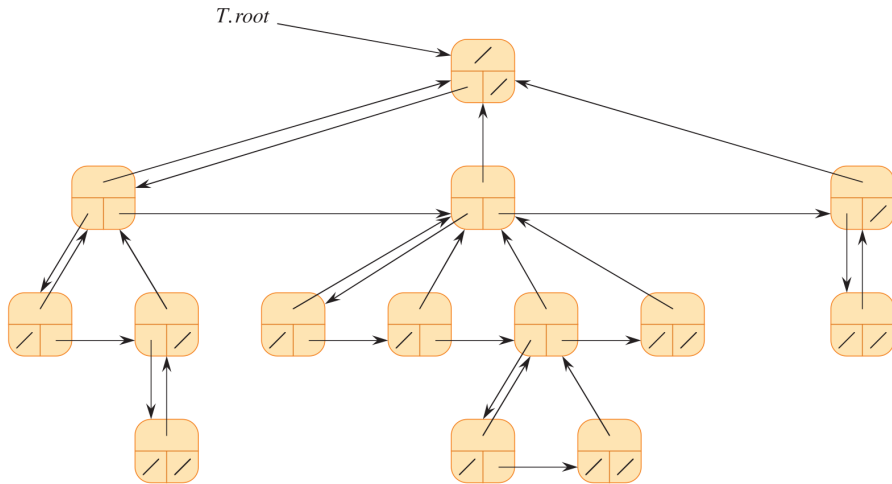
- می‌توانیم درخت دودویی را تعمیم دهیم به طوری که یک رأس درخت بتواند به هر تعداد دلخواه فرزند داشته باشد. می‌توانیم درختی تعریف کنیم که تعداد فرزند هر رأس در آن حداکثر  $k$  باشد. بنابراین ویژگی‌های  $left$  و  $right$  را با ویژگی‌های  $child_1$ ،  $child_2$ ، ... و  $child_k$  جایگزین می‌کنیم.
- حال 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 ریشه 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.



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

---

## Data Structure Node

---

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

---

---

## Data Structure Tree

---

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

---

– اگر رأس  $x$  فرزندی نداشته باشد، آنگاه  $x.\text{left-child}=\text{NIL}$  است و اگر رأس  $x$  خود راست‌ترین فرزند یک پدر باشد، آنگاه  $x.\text{right-sibling}=\text{NIL}$  است.

- 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.

- درخت‌ها را می‌توانیم به دو روش عمق‌اول<sup>1</sup> و سطح‌اول<sup>2</sup> پیمایش کنیم. در پیمایش عمق‌اول ابتدا رئوسی که در عمق بیشتر نسبت به ریشه هستند پیمایش می‌شوند و در پیمایش سطح‌اول رئوس سطح به سطح، با شروع از نزدیکترین سطح به ریشه پیمایش می‌شوند.

---

<sup>1</sup> depth-first

<sup>2</sup> breath-first

## پیمایش درخت‌ها

- در پیمایش عمق‌اول، درخت‌ها را می‌توانیم به سه روش پیمایش کنیم: پیمایش میان‌ترتیب<sup>1</sup>، پیمایش پیش‌ترتیب<sup>2</sup>، و پیمایش پس‌ترتیب<sup>3</sup>.
- پیمایش میان‌ترتیب 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 پیش‌ترتیب مقدار کلید ریشه قبل از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.
- در پیمایش پس‌ترتیب مقدار کلید ریشه بعد از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.

---

<sup>1</sup> inorder tree walk

<sup>2</sup> preorder tree walk

<sup>3</sup> postorder tree walk

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

---

## Algorithm Inorder Tree Walk

---

```
function INORDER-TREE-WALK(x)
1: if  $x \neq \text{NIL}$  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  $\text{Inorder-Tree-Walk}(T.\text{root})$  را فراخوانی کنیم.



- الگوریتم پیمایش پیش‌ترتیب در زیر نشان داده شده است.

---

## Algorithm Preorder Tree Walk

---

```
function PREORDER-TREE-WALK(x)
1: if  $x \neq \text{NIL}$  then
2:   print x.key
3:   Preorder-Tree-Walk (x.left)
4:   Preorder-Tree-Walk (x.right)
```

---

- الگوریتم پیمایش پس‌ترتیب در زیر نشان داده شده است.

---

## Algorithm Postorder Tree Walk

---

```
function POSTORDER-TREE-WALK(x)
1: if  $x \neq \text{NIL}$  then
2:   Postorder-Tree-Walk (x.left)
3:   Postorder-Tree-Walk (x.right)
4:   print x.key
```

---

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

- 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.

- 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.

- در پیمایش سطح‌اول، از یک صف استفاده می‌کنیم. ابتدا ریشه درخت را در صف قرار می‌دهیم. سپس رئوس را به ترتیب از صف خارج می‌کنیم و به ازای هر رأس خارج شده، فرزندان آن را وارد صف می‌کنیم. این کار را ادامه می‌دهیم تا صف خالی شود.

---

### Algorithm Breath-First (Level-Order) Tree Walk

---

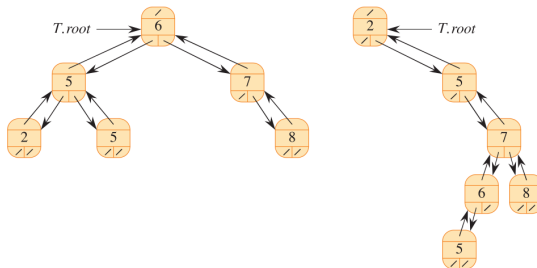
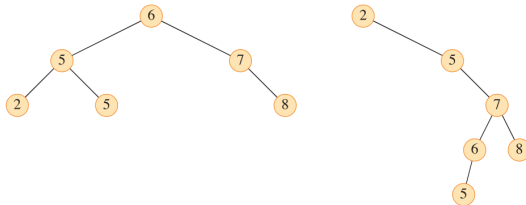
```
function BREATH-FIRST-TREE-WALK(s)
1: Q =  $\emptyset$ 
2: Enqueue(Q, s)
3: while !Empty(Q) do
4:   x = Dequeue(Q)
5:   print x.key
6:   if x.left  $\neq$  NIL then Enqueue(x.left)
7:   if x.right  $\neq$  NIL then Enqueue(x.right)
```

---

- 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  $y.key \leq x.key$  و اگر  $y$  یک رأس در زیر درخت سمت راست  $x$  باشد، آنگاه داریم  $y.key \geq x.key$ .

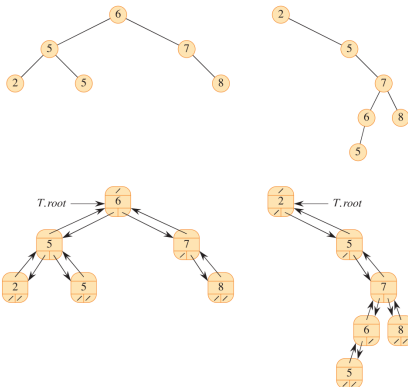
# درخت جستجوی دودویی

- 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.





- به دلیل ویژگی خاص درخت جستجوی دودویی، با پیمایش میان ترتیب<sup>1</sup> 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<sup>2</sup> مقدار کلید ریشه قبل از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود و در پیمایش پس ترتیب<sup>3</sup> مقدار کلید ریشه بعد از کلیدهای رئوس زیر درخت‌های سمت چپ و راست چاپ می‌شود.

---

<sup>1</sup> inorder tree walk

<sup>2</sup> preorder tree walk

<sup>3</sup> postorder tree walk

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

---

### Algorithm Inorder Tree Walk

---

```
function INORDER-TREE-WALK(x)
1: if  $x \neq \text{NIL}$  then
2:   Inorder-Tree-Walk (x.left)
3:   print x.key
4:   Inorder-Tree-Walk (x.right)
```

---

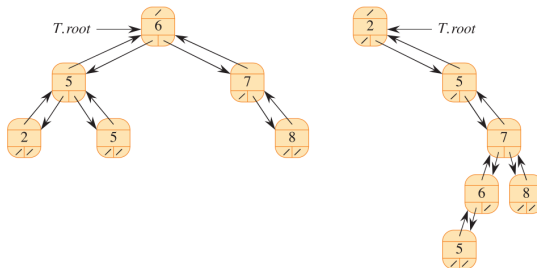
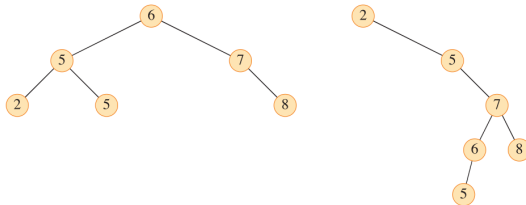
- برای چاپ همه عناصر درخت جستجوی دودویی باید تابع  $\text{Inorder-Tree-Walk}(T.\text{root})$  را فراخوانی کنیم.
- این الگوریتم در زمان  $\Theta(n)$  به ازای درخت جستجوی دودویی با  $n$  رأس اجرا می‌شود.
- درستی این الگوریتم مستقیماً با استفاده از استقرا بر روی ویژگی درخت جستجوی دودویی اثبات می‌شود.

## درخت جستجوی دودویی

- قضیه : اگر  $x$  ریشه یک زیر درخت با  $n$  رأس باشد، فراخوانی  $\text{Inorder-Tree-Walk}(x)$  در زمان  $\Theta(n)$  انجام می‌شود.
- اثبات : فرض کنید  $T(n)$  زمان مورد نیاز برای اجرای الگوریتم بر روی زیر درختی با  $n$  رأس باشد.
- همه رئوس زیر درخت در نهایت بررسی می‌شوند پس لزوماً  $T(n) = \Omega(n)$  . 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  $T(n) \leq T(k) + T(n - k - 1) + d$  .
- با حل این رابطه به روش جایگذاری به دست می‌آوریم  $T(n) = O(n)$  .

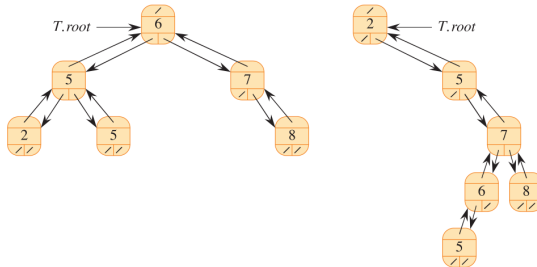
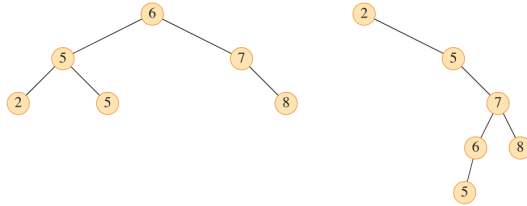
## درخت جستجوی دودویی

- 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.



- 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)
1: if x == NIL or k == x.key then
2:   return x
3: if k < x.key then
4:   return Tree-Search (x.left, k)
5: else return Tree-Search (x.right, k)
```

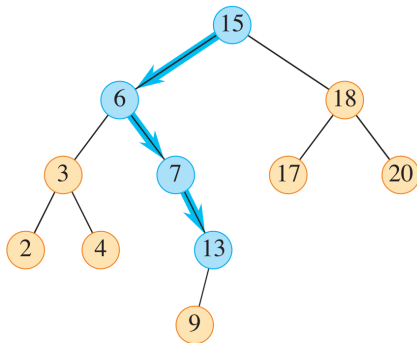
---

- به ازای اشاره‌گر  $x$  به ریشه یک زیر درخت و مقدار کلید  $k$  تابع  $\text{Tree-search}(x,k)$  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  $\text{Tree-search}(T.\text{root},k)$  فراخوانی می‌شود.

- 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 درخت 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.



- 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.

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

---

### Algorithm Tree Minimum

---

```
function TREE-MINIMUM(x)
1: while x.left  $\neq$  NIL do
2:   x = x.left
3: 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 همه 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

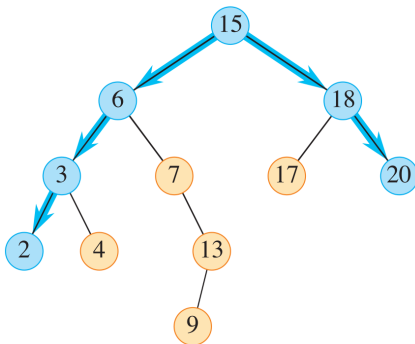
---

```
function TREE-MAXIMUM(x)
1: while x.right  $\neq$  NIL do
2:   x = x.right
3: 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.

- رئوس بعدی و قبلی : اگر همه کلیدها در درخت جستجوی دودویی یکتا باشند، رأس بعدی  $x^1$  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.

---

<sup>1</sup> successor

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

---

### Algorithm Tree Successor

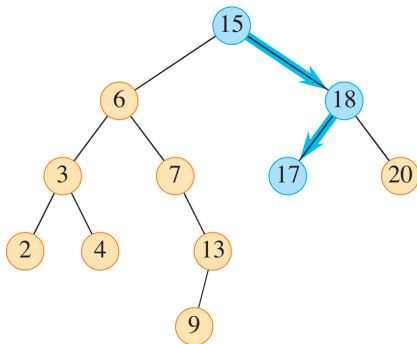
---

```
function TREE-SUCCESSOR(x)
1: if x.right  $\neq$  NIL then
2:   return Tree-Minimum (x.right)      ▷ leftmost node in right subtree
3: else ▷ find the lowest ancestor of x whose left child is an ancestor of
   x
4:   y = x.p
5:   while y  $\neq$  NIL and x == y.right do
6:     x = y
7:     y = y.p
8:   return y
```

---

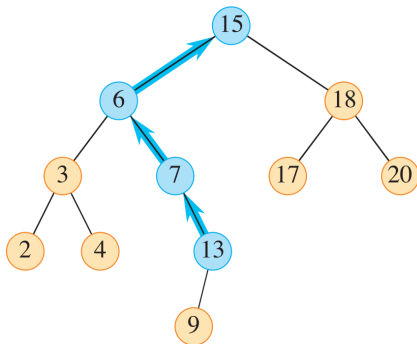
## درخت جستجوی دودویی

- 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 جایی که یا به ریشه برسیم و یا به رأسی برسیم که فرزند چپ پدر خود باشد.
- در شکل زیر رأس بعدی ۱۳ مقدار ۱۵ است.





- 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.

- 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.

---

## Algorithm Tree Insert

---

```

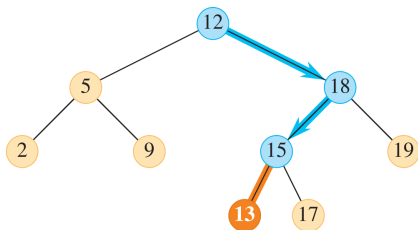
-   function TREE-INSERT(T,z)
1:  x = T.root  ▷ node being compared with z
2:  y = NIL    ▷ y will be parent of z
3:  while x ≠ NIL do  ▷ descend until reaching a leaf
4:      y = x
5:      if z.key < x.key then
6:          x = x.left
7:      else x = x.right
8:  z.p = y  ▷ found the location-insert z with parent y
9:  if y == NIL then
10:     T.root = z    ▷ tree T was empty
11:  else if z.key < y.key then
12:     y.left = z
13:  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.

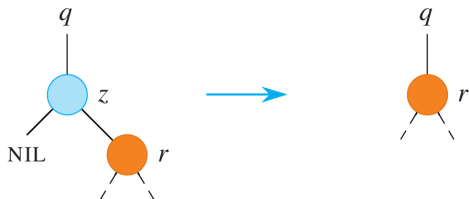


- پیچیدگی زمانی درج در درخت جستجو برای درخت با ارتفاع h برابر است با  $O(h)$ .

- 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.



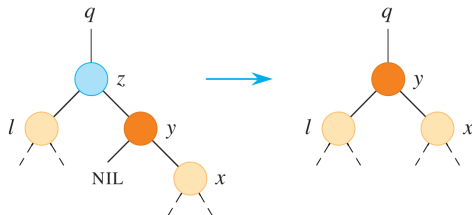
- اگر  $z$  تنها فرزند سمت چپ داشته باشد، طبق شکل زیر، رأس  $z$  با فرزند سمت چپ آن جایگزین می‌شود.



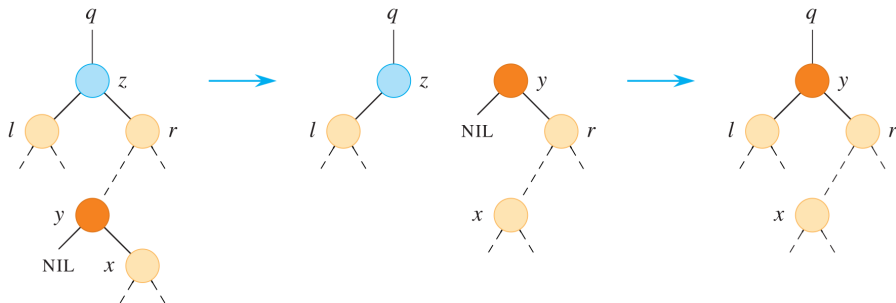
## درخت جستجوی دودویی

- 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.



(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)
1: if u.p == NIL then
2:   T.root = v
3: else if u == u.p.left then
4:   u.p.left = v
5: else u.p.right = v
6: if v  $\neq$  NIL then
7:   v.p = u.p
```

---

- خطوط ۱ و ۲ حالتی را بررسی می کنند 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 روزرسانی می کند. اگر  $u$  یک فرزند سمت راست باشد، خط ۵ مقدار  $u.p.right$  را به روزرسانی می کند.
- از آنجایی که  $v$  می تواند تهی باشد، خطوط ۶ و ۷ مقدار  $v.p$  را به روزرسانی می کند تنها اگر  $v$  تهی نباشد.
- تابع Transplant مقادیر  $v.left$  و  $v.right$  را تغییر نمی دهد.

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

---

### Algorithm Tree Delete

---

```

function TREE-DELETE(T,z)
1: if z.left == NIL then
2:   Transplant (T,z,z.right)      ▷ replace z by its right child
3: else if z.right == NIL then
4:   Transplant (T,z,z.left)       ▷ replace z by its left child
5: else y = Tree-Minimum(z.right)  ▷ y is z's successor
6:   if y ≠ z.right then           ▷ is y farther down the tree?
7:     Transplant (T,y,y.right)    ▷ replace y by its right child
8:     y.right = z.right           ▷ z's right child becomes
9:     y.right.p = y              ▷ y's right child
10:  Transplant (T,z,y)            ▷ replace z by its successor y
11:  y.left = z.left               ▷ and give z's left child to y,
12:  y.left.p = y                 ▷ which had no left child

```

---

- 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  $\text{Tree-Minimum}(z.\text{right})$  استفاده می‌شود. قبلاً ذکر کردیم که الزاماً  $y$  فرزند سمت چپ ندارد. تابع حذف، باید  $y$  را از مکان فعلی خود خارج و  $z$  را با  $y$  جایگزین کند.

- 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 فرزند سمت راست  $y$  را با فرزند سمت راست  $z$  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.

- 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.

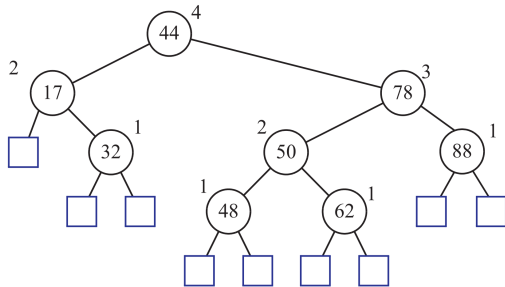
- درخت جستجوی دودویی ممکن است عناصر را به گونه ای در درخت درج کند که پیچیدگی زمانی جستجو در مجموعه ای از  $n$  عنصر در بدترین حالت  $O(n)$  باشد.
- اگر درخت جستجوی دودویی ارتفاع متوازن داشته باشد یا به عبارت دیگر دارای ویژگی ارتفاع متوازن<sup>1</sup> باشد می توانیم پیچیدگی زمانی بدترین حالت را کاهش دهیم.
- ویژگی ارتفاع متوازن : به ازای هر رأس میانی  $v$  در درخت  $T$  ، اختلاف ارتفاع فرزندان آن حداکثر برابر با یک است.

---

<sup>1</sup> height-balance property

# درخت ای وی ال

- هر درخت جستجوی دودویی T که دارای ویژگی ارتفاع متوازن باشد، درخت ای وی ال<sup>1</sup> نامیده می شود که نام خود را از ابتدای نام ابداع کنندگان آن یعنی آدلسون ولسکی<sup>2</sup> و لندیس<sup>3</sup> گرفته است.
- یک مثال از درخت ای وی ال در شکل زیر نشان داده شده است.



---

<sup>1</sup> AVL tree

<sup>2</sup> Adelson-Velskii

<sup>3</sup> Landis



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

## درخت ای وی ال

- به ازای اعداد کوچک داریم  $n(1) = 1$  و  $n(2) = 2$  زیرا یک درخت ای وی ال با ارتفاع یک باید حداقل یک رأس میانی داشته باشد و یک درخت ای وی ال با ارتفاع ۲ باید حداقل ۲ رأس میانی داشته باشد.
  - حال به ازای  $h \geq 3$  یک درخت ای وی ال با ارتفاع  $h$  و کمترین تعداد رئوس به گونه ای است که هر دو زیر درخت آن درخت های ای وی ال 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 می آوریم :
- $$n(h) = 1 + n(h - 1) + n(h - 2)$$
- تابع  $n(h)$  یک تابع اکیدا صعودی است. بنابراین  $n(h - 1) \geq n(h - 2)$  .
  - پس می توانیم بنویسیم  $n(h) > 2n(h - 2)$  .

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

- با جایگذاری مقدار  $i$  به دست می آوریم :

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

- از دو طرف رابطه لگاریتم می گیریم و به دست می آوریم  $\lg n(h) > \frac{h}{2} - 1$ .

- بنابراین داریم  $h < 2 \lg n(h) + 2$

- نتیجه می گیریم که ارتفاع یک درخت ای وی ال با  $n$  رأس حداکثر  $2 \lg n + 2$  است که برابر با  $O(\lg n)$  است.

# درج در درخت ای وی ال

- درج در درخت ای وی ال 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 ای وی ال 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 <sup>1</sup> متوازن است، اگر مقدار قدر مطلق تفاضل ارتفاع فرزندان  $v$  حداکثر ۱ باشد و در غیر این صورت درخت نامتوازن <sup>2</sup> است. بنابراین هر یک از رئوس میانی درخت طبق ویژگی توازن ارتفاع باید متوازن باشند.

---

<sup>1</sup> balanced

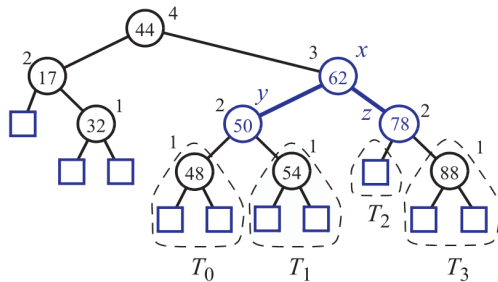
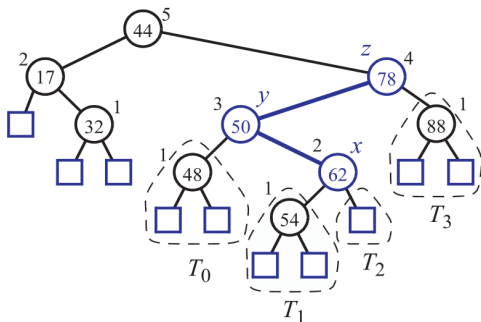
<sup>2</sup> unbalanced

## درج در درخت ای وی ال

- فرض کنیم  $T$  یک درخت ای وی ال باشد. پس از عملیات درج در درخت  $T$ ، ارتفاع برخی از رئوس درخت  $T$  ممکن است افزایش پیدا کند. رئوسی که ارتفاع آنها تغییر می کند 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 ای وی ال نیست و باید آن را مجددا متوازن کنیم.

# درج در درخت ای وی ال

- 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 درآمده است.



# درج در درخت ای وی ال

- توازن رئوس در درخت  $T$  را با یک استراتژی جستجو و ترمیم<sup>1</sup> بازیابی می کنیم.
- فرض کنید  $z$  اولین رأس نامتوازن باشد که با حرکت از رأس  $w$  به سمت ریشه به آن برخورد می کنیم.
- 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  $y$  نامتوازن شده است و بنابراین ارتفاع  $y$  دو واحد بزرگ تر از همزادش است.

---

<sup>1</sup> search and repair

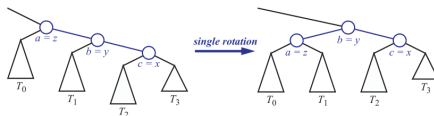


## درج در درخت ای وی ال

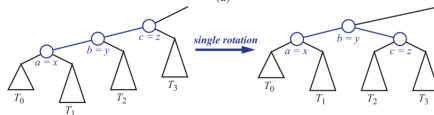
- حال با استفاده از الگوریتمی که شرح داده خواهد شد، زیر درخت با ریشه  $z$  را متوازن می کنیم.
- این تابع به طور موقت رئوس  $x$  و  $y$  و  $z$  را به  $a$  و  $b$  و  $c$  تغییر نام می دهد، به طوری که در یک پیمایش میان ترتیب از درخت  $T$  رأس  $a$  قبل از  $b$  و رأس  $b$  قبل از  $c$  قرار بگیرد.

# درج درخت ای وی ال

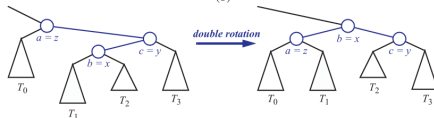
- رئوس  $x$  و  $y$  و  $z$  به چهار حالت زیر ممکن است به  $a$  و  $b$  و  $c$  نگاشت شوند.



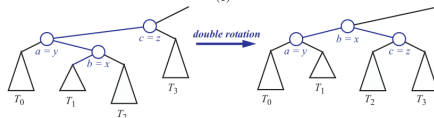
(a)



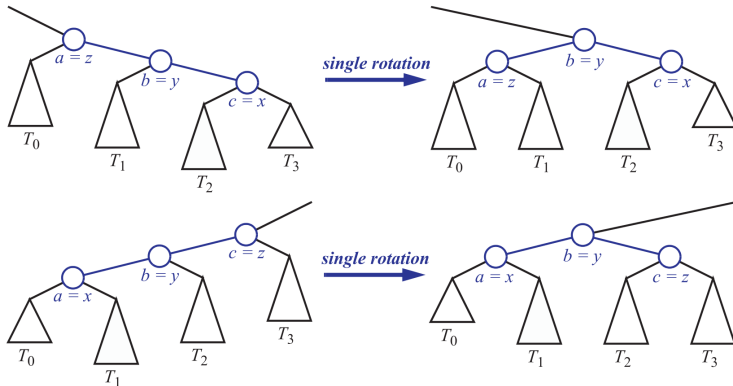
(b)



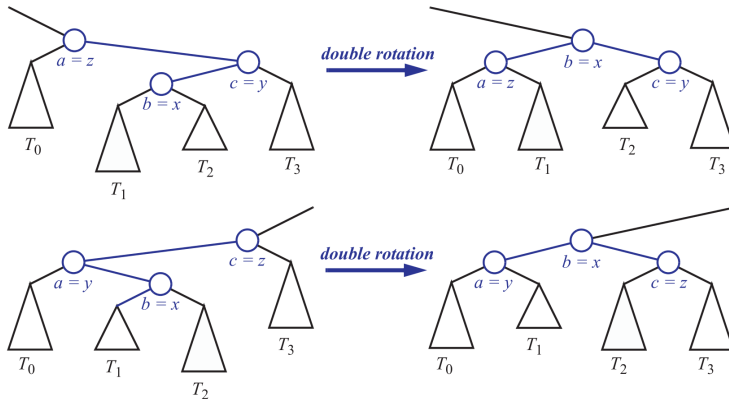
(c)



# درج درخت ای وی ال



# درج در درخت ای وی ال



# درج در درخت ای وی ال

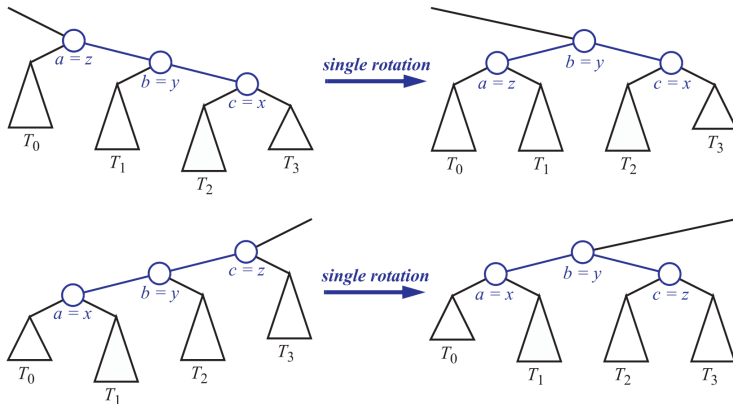
- برای بازگرداندن توازن درخت، الگوریتم تغییر ساختار (restructure) درخت را به نحوی تغییر می دهد که  $z$  با  $b$  جایگزین شود به طوری که فرزندان آن  $a$  و  $c$  و فرزندان  $a$  و  $c$  فرزندان قبلی  $x$  و  $y$  و  $z$  باشند و ترتیب رئوس در پیمایش میان ترتیب حفظ شود.
- عملیات تغییر درخت  $T$  با استفاده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

---

<sup>1</sup> rotation

# درج درخت ای وی ال

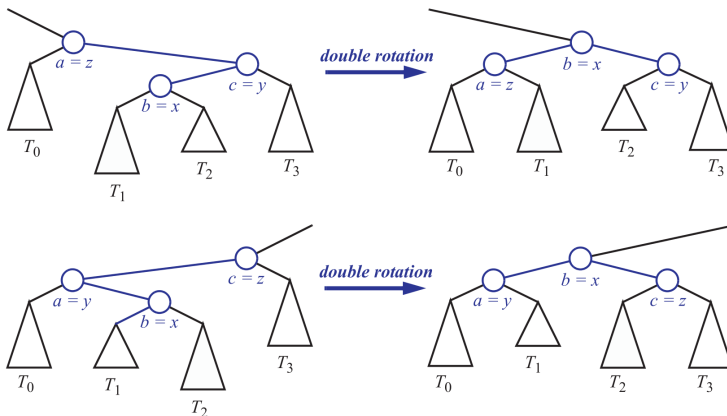
- اگر  $b = y$  باشد، این تغییر ساختار دوران تکی<sup>1</sup> نامیده می شود، به طوری که به نظر می رسد  $y$  بر روی  $z$  چرخیده است.



<sup>1</sup> single rotation

# درج در درخت ای وی ال

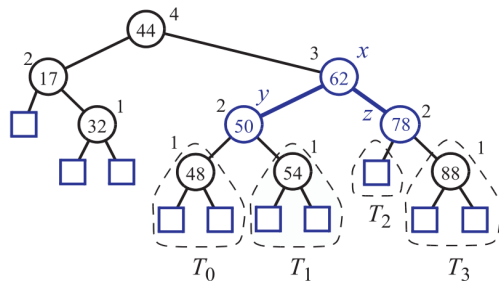
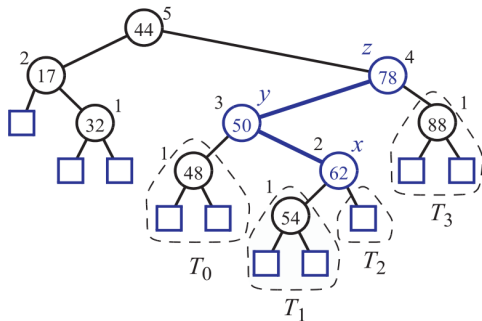
- اگر  $b = x$  باشد، عملیات تغییر ساختار دوران دوتایی<sup>1</sup> نامیده می شود، به طوری که به نظر می رسد  $x$  بر روی  $y$  و سپس بر روی  $z$  چرخیده است.



<sup>1</sup> double rotation

# درج در درخت ای وی ال

- در مثال زیر یک دوران دوتایی انجام شده است.





---

## 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 (T<sub>0</sub> , T<sub>1</sub> , T<sub>2</sub> , T<sub>3</sub> ) 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 T<sub>0</sub> and T<sub>1</sub> be the left and right subtrees of a, respectively.
  - 4: Let c be the right child of b and let T<sub>2</sub> and T<sub>3</sub> be the left and right subtrees of c, respectively.
-

# درج در درخت ای وی ال

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

## درج در درخت ای وی ال

- توجه کنید تابع  $\text{restructure}(x)$  فراخوانی می شود زیرا  $z$  پدر بزرگ  $x$  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 می شود، بنابراین بعد از تغییر ساختار همه رئوس در زیر درخت با ریشه  $b$  متوازن می شود.
- بنابراین ویژگی توازن ارتفاع در رئوس  $x$  و  $y$  و  $z$  به صورت محلی<sup>1</sup> برقرار می شود.
- از آنجایی که بعد از انجام عملیات درج، زیر درخت با ریشه  $b$  جایگزین زیر درختی می شود که قبلاً دارای ریشه  $z$  بود، همه اجداد  $z$  که نامتوازن بودند مجدداً متوازن می شوند. در نتیجه درخت به صورت عمومی<sup>2</sup> متوازن می شود.

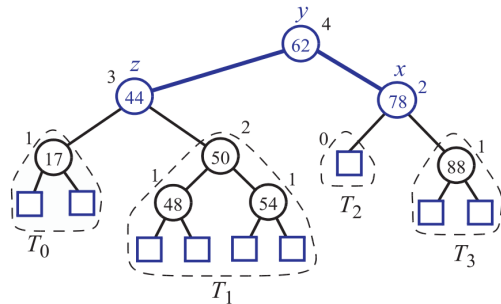
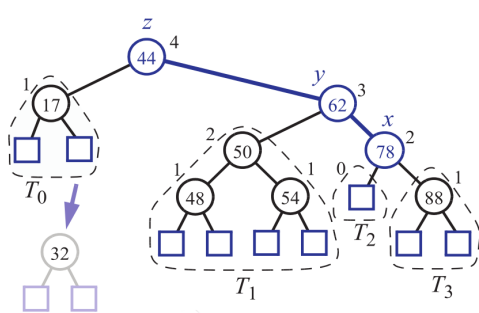
---

<sup>1</sup> locally

<sup>2</sup> globally

# حذف از درخت ای وی ال

- برای حذف یک رأس از درخت ای وی ال ابتدا عملیات حذف معمولی بر روی درخت جستجوی دودویی را انجام می دهیم. 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 ای وی ال

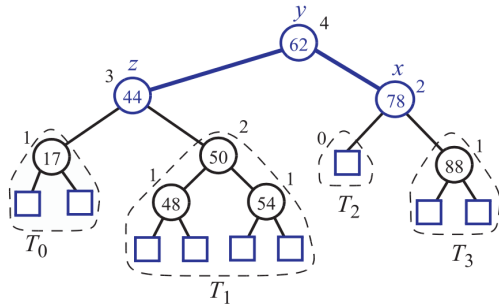
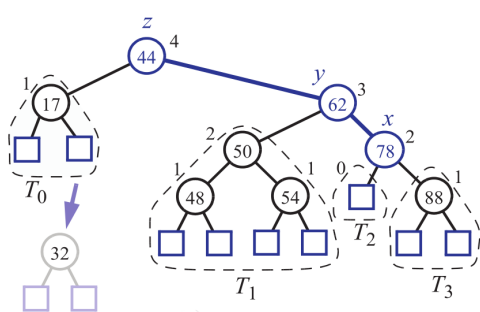
- فرض کنید  $z$  اولین رأس نامتوازن باشد که با بالا رفتن از رأس حذف شده  $w$  به سمت ریشه درخت  $T$  با آن برخورد می کنیم.
- فرض کنید  $y$  فرزند  $z$  با ارتفاع بیشتر باشد. رأس  $y$  فرزندی از  $z$  است که جد  $w$  نیست.
- فرض کنید  $x$  فرزند  $y$  باشد که به صورت زیر تعریف شده است : اگر یکی از فرزندان  $y$  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 ای وی ال

- سپس عملیات  $\text{restructure}(x)$  را انجام می دهیم که ویژگی توازن ارتفاع را به صورت محلی در زیر درخت با ریشه  $z$  که اکنون دارای ریشه  $b$  است بازیابی می کند.

# حذف از درخت ای وی ال

- شکل زیر یک مثال از بازیابی توازن را نشان می دهد.



## حذف از درخت ای وی ال

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



## حذف از درخت ای وی ال

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

- 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<sup>1</sup> یکی از انواع درخت‌های جستجو هستند که متوازن<sup>2</sup> 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  $c$  کوچکتري دارد یا به عبارت دیگر توازن بیشتری دارد و در نتیجه در جستجو سریع‌تر عمل می‌کند.

---

<sup>1</sup> red-black trees

<sup>2</sup> balanced

- 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 تقریباً 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.

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

۱. ویژگی ریشه : ریشه درخت سیاه است.

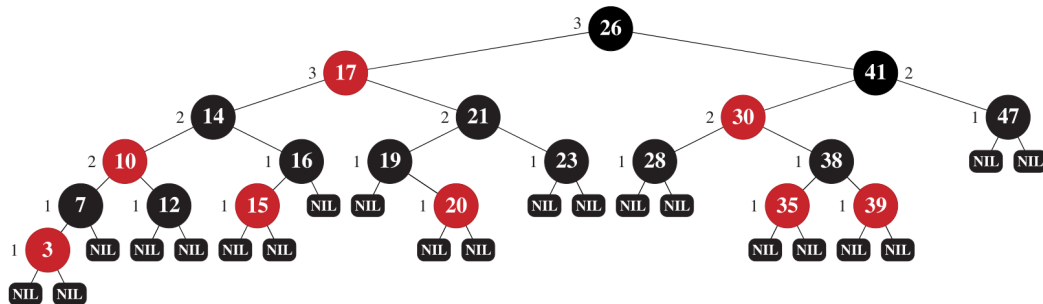
۲. ویژگی برگ‌ها : هر برگ سیاه است.

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.



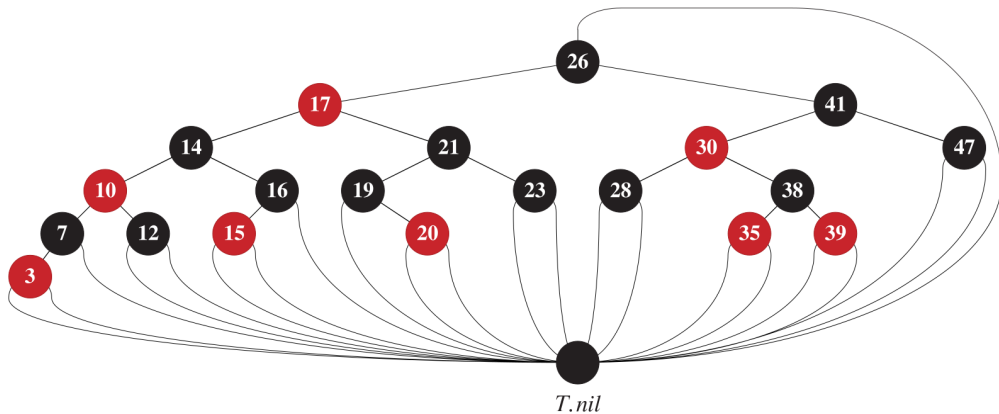
- برای مدیریت شرایط مرزی و صرفه‌جویی در حافظه در درخت قرمز-سیاه از یک نگهبان<sup>1</sup> برای ذخیره‌سازی رئوس تهی NIL استفاده می‌کنیم. برای درخت T ، نگهبان T.nil شیئی است که یک رأس معمولی از درخت است ، رنگ آن سیاه است، و مقدار p ، left ، right و key در آن تهی است.

---

<sup>1</sup> sentinel

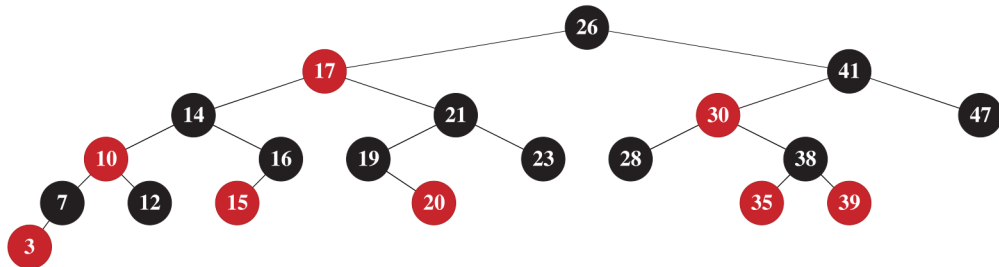
## درخت‌های قرمز-سیاه

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



## درخت‌های قرمز-سیاه

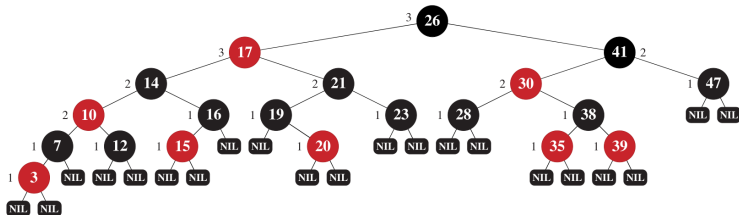
– جهت سهولت نمایش درخت، برگ‌ها را حذف می‌کنیم و درخت قرمز-سیاه را تنها با رئوس میانی نشان می‌دهیم. شکل زیر نمایش درخت قرمز-سیاه بدون برگ‌های تهی است.





## درخت‌های قرمز-سیاه

- تعداد رئوس سیاه بر روی هر مسیر از رأس  $x$  بدون احتساب خود رأس تا یکی از برگ‌ها را ارتفاع سیاه<sup>1</sup> 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 در کنار آن درج شده است.



<sup>1</sup> black-height

- 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.

- 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.

- 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  $1 + (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1)$  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 بر روی 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  $n \geq 2^{h/2} - 1$ .
- بنابراین به دست می‌آوریم  $h \leq 2 \lg(n + 1)$ .

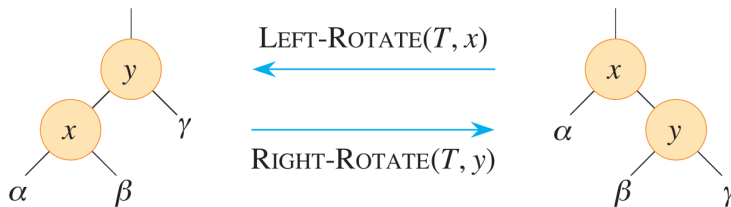
- از این قضیه نتیجه می‌گیریم عملیات مجموعه‌های پویای جستجو (Search) ، کمینه (Minimum) ، بیشینه (Maximum) ، محاسبه رأس بعدی (Successor) ، و رأس قبلی (Predecessor) در زمان  $O(\lg n)$  در درخت قرمز-سیاه انجام می‌شوند، زیرا هر کدام در زمان  $O(h)$  در درخت جستجوی دودویی انجام می‌شوند و درخت قرمز-سیاه با  $n$  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.

- عملیات درج و حذف بر روی 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<sup>1</sup> انجام می‌شود که یک عملیات محلی در یک درخت جستجو است.

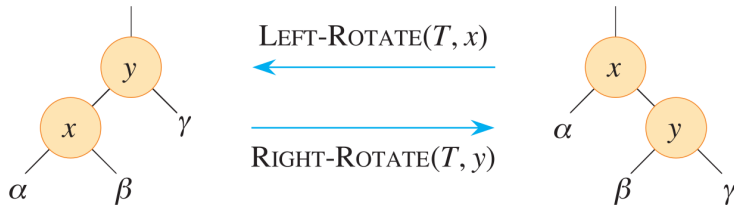
---

<sup>1</sup> 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<sup>1</sup> 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  $\beta$  است) فرزند راست  $x$  شود.



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

<sup>1</sup> left rotation



---

### Algorithm Left Rotate

---

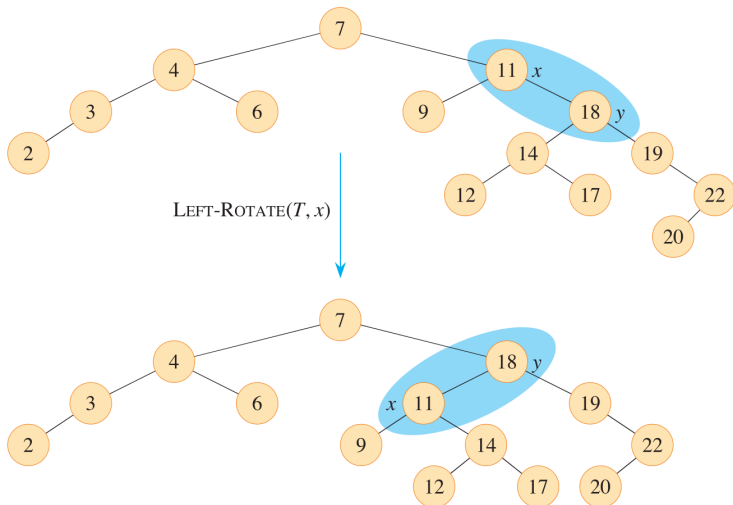
```

-   function LEFT-ROTATE(T,x)
1:  y = x.right
2:  x.right = y.left  ▷ turn y's left subtree into x's right subtree
3:  if y.left ≠ T.nil then  ▷ if y's left subtree is not empty ...
4:    y.left.p = x      ▷ ... then x becomes the parent of the subtree's
    root
5:  y.p = x.p  ▷ x's parent becomes y's parent
6:  if x.p == T.nil then  ▷ if x was the root ...
7:    T.root = y      ▷ ... then y becomes the root
8:  else if x == x.p.left then  ▷ otherwise, if x was a left child
9:    x.p.left = y      ▷ ... then y becomes a left child
10: else  x.p.right = y  ▷ otherwise, x a right child, and now y is
11: y.left = x  ▷ make x become y's left child
12: x.p = y

```

---

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



- 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.

---

## Algorithm RB Insert

---

```
function RB-INSERT(T,z)
1: x = T.root  ▷ node being compared with z
2: y = T.nil  ▷ y will be parent of z
3: while x ≠ T.nil do  ▷ descend until reaching the sentinel
4:   y = x
5:   if z.key < x.key then
6:     x = x.left
7:   else x = x.right
8: z.p = y  ▷ found the location - insert z with parent y
9: if y == T.nil then
10:  T.root = z  ▷ tree T was empty
11: else if z.key < y.key then
12:  y.left = z
13: else y.right = z
14: z.left = T.nil  ▷ both of z's children are the sentinel
15: z.right = T.nil
16: z.color = RED  ▷ the new node starts out red
17: 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 نقص شود، در خط ۱۷ تابع RB-Insert-Fixup فراخوانی می‌شود تا ویژگی درخت قرمز-سیاه حفظ شود.

## 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 این که بفهمیم تابع RB-Insert-Fixup چگونه عمل می کند، تابع را در سه گام بررسی می کنیم.
- ابتدا تعیین می کنیم با درج کردن رأس  $z$  و رنگ کردن آن با قرمز، چگونه ویژگی های درخت قرمز-سیاه نقض می شوند.
- سپس بررسی می کنیم هدف از حلقه خطوط ۱ تا ۲۹ چیست.
- در پایان سه حالت حلقه اصلی تابع را بررسی می کنیم.
- در توصیف درخت قرمز-سیاه نیاز داریم درمورد همزاد رأس پدر صحبت کنیم. از واژه عمو<sup>۱</sup> برای نام بردن از رأس همزاد پدر استفاده می کنیم.

---

<sup>۱</sup> uncle

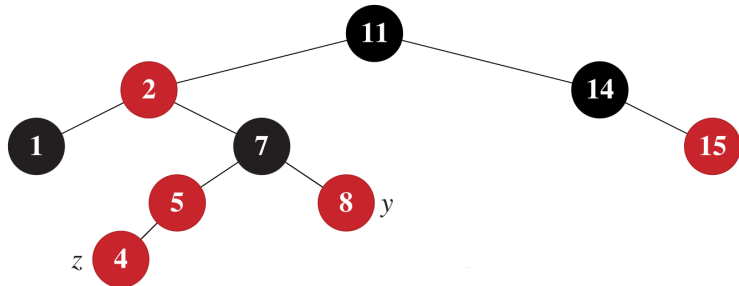


## درج در درخت قرمز-سیاه

- 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 پدر بزرگ  $z$  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)
```

---

- 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 قرمز-سیاه

- 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 عمومی رأس  $z$  یعنی  $y$  متفاوت است. در خط ۳ اشاره گر  $y$  به عمومی 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، پدر بزرگ  $z$  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

---

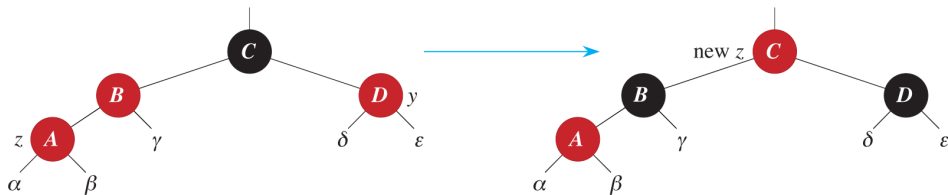
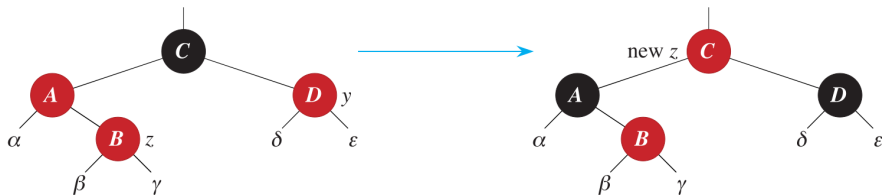
```
function RB-INSERT-FIXUP(T,z)
1: while z.p.color == RED do
2:   if z.p == z.p.p.left then      ▷ is z's parent a left child?
3:     y = z.p.p.right              ▷ y is z's uncle
4:     if y.color == RED then        ▷ are z's parent and uncle both red?
5:       z.p.color = BLACK           ▷ Case 1
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 : عموی  $z$  قرمز است.

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



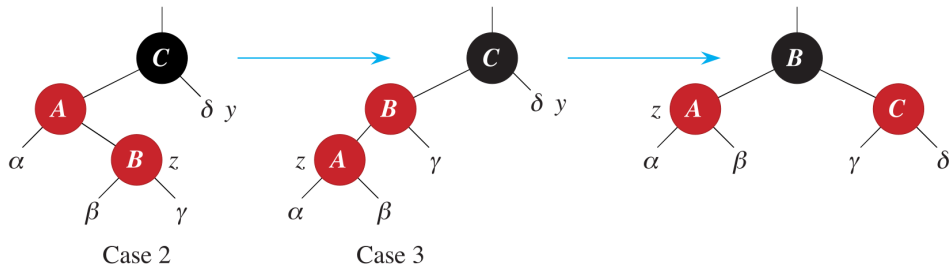


## درج در درخت قرمز-سیاه

– از آنجایی که پدر بزرگ  $z$  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 این صورت پدر بزرگ  $z$  قرمز می‌شود. حلقه با  $z.p.p$  به عنوان رأس جدید  $z$  تکرار می‌شود، و اشاره گر  $z$  دو سطح به بالا حرکت می‌کند.

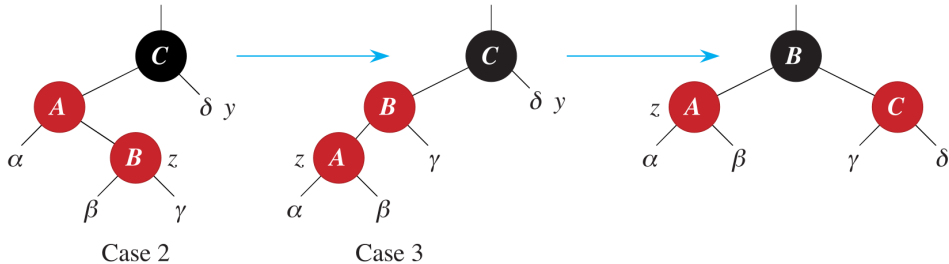
## درج در درخت قرمز-سیاه

- 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.



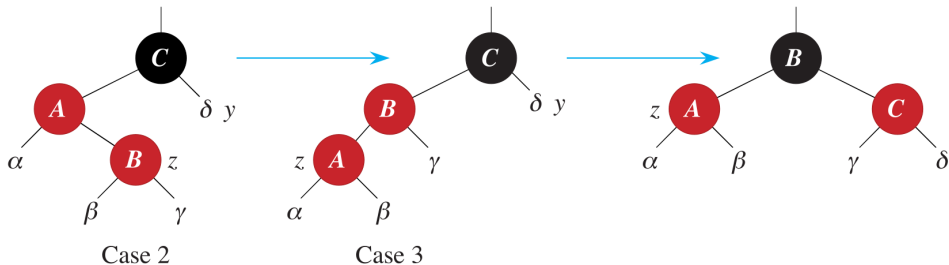
## درج در درخت قرمز-سیاه

- 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 مستقیماً و یا از طریق حالت ۲ اجرا شود، رأس  $y$  عموی  $z$  سیاه است، زیرا در غیراینصورت حالت ۱ اجرا شده بود.



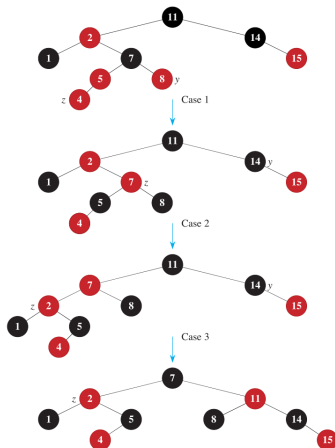
## درج در درخت قرمز-سیاه

- 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 بر روی یک درخت قرمز-سیاه عمل می‌کند بسته به این که رأس پدر و عموی رأس  $z$  چه رنگی 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 این صورت 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 زیاده‌تری 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.