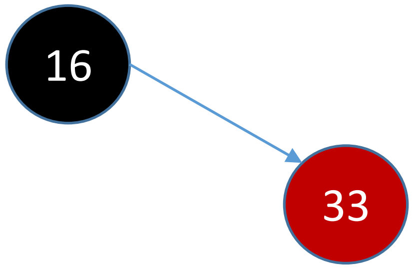
سوال 1 :

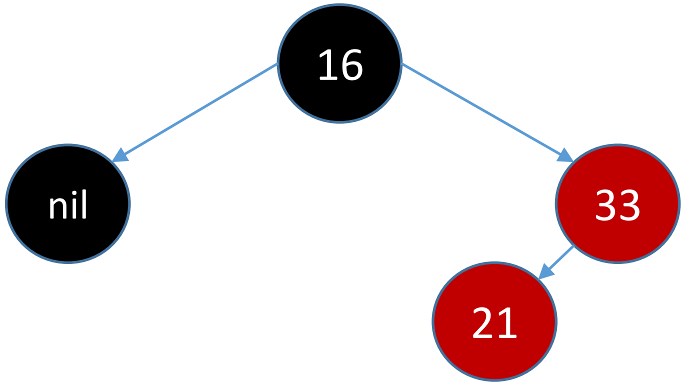
الف)

Insert : 16

Insert : 33

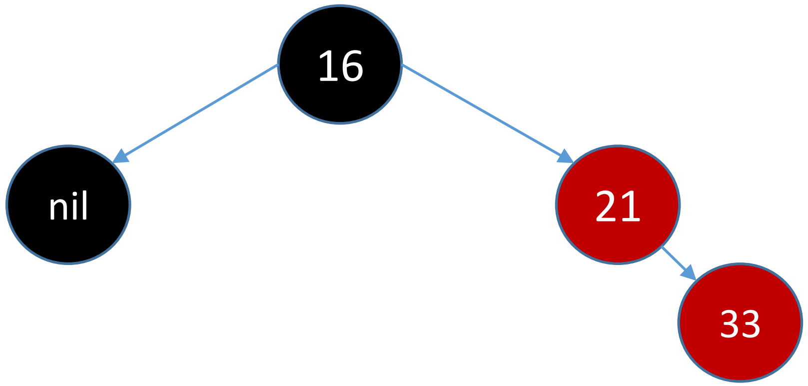


Insert : 21



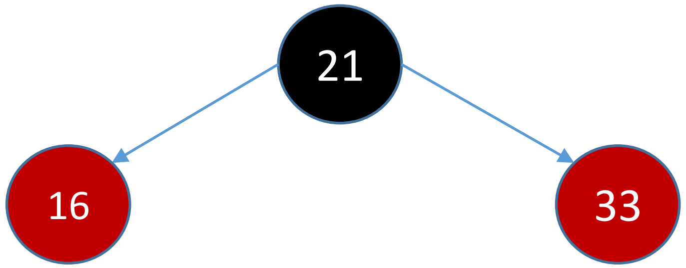
Case 2

Insert : 21



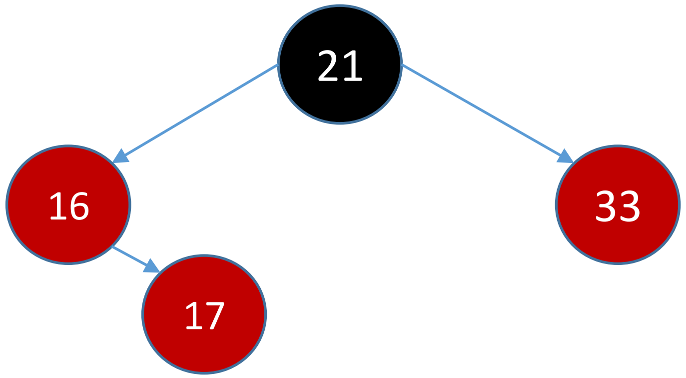
Case 3

Insert : 21



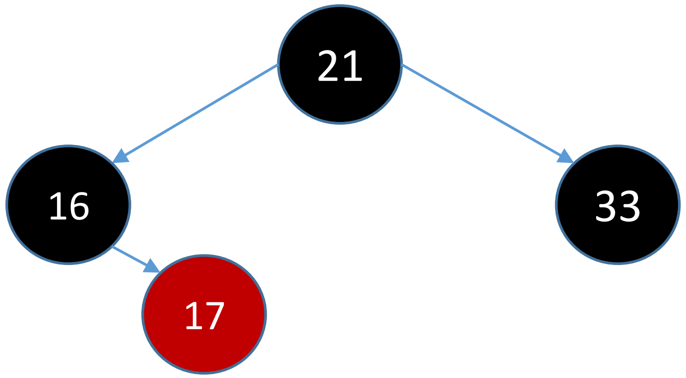
Fixed!!

Insert : 17



Case1

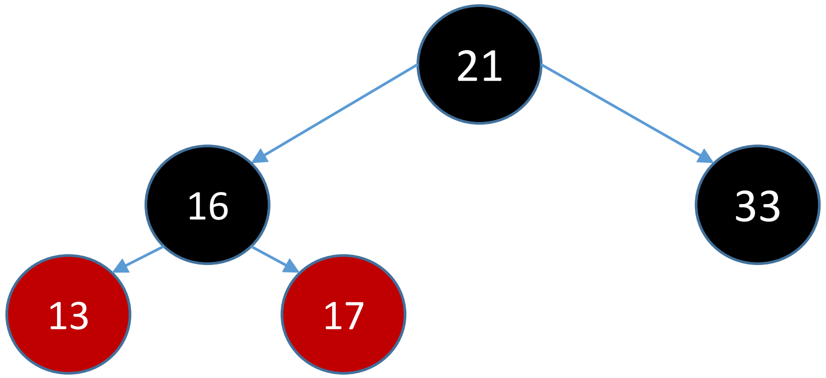
Insert : 17



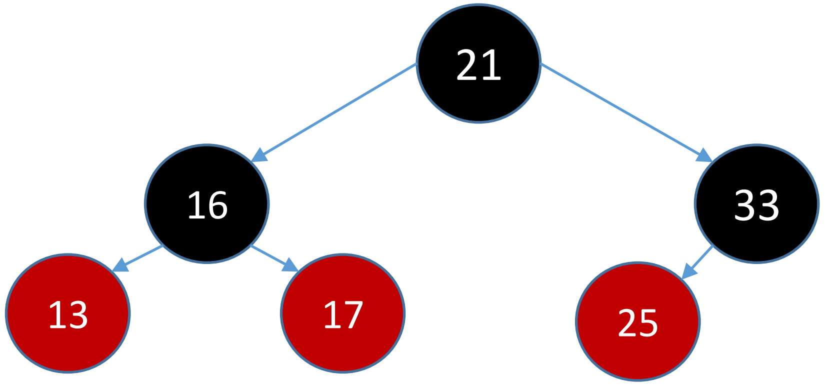
Fixed!!

سوال اول:

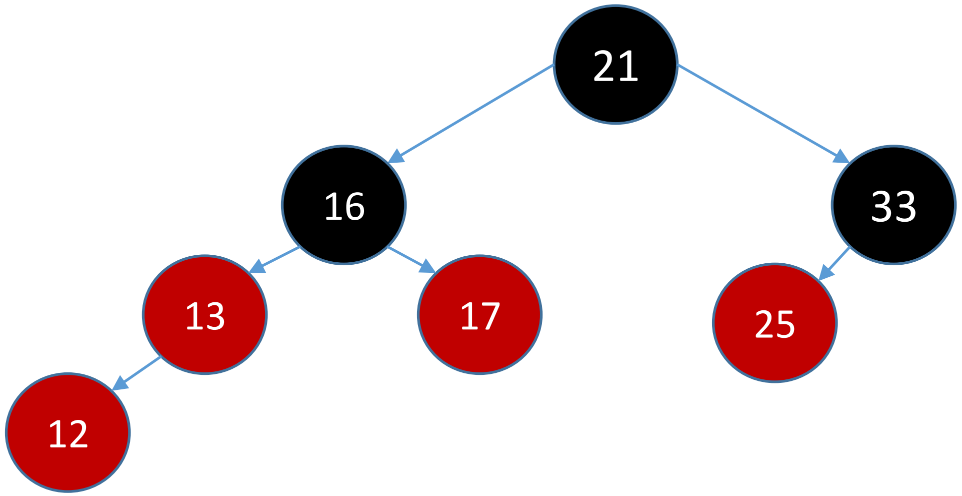
Insert : 13



Insert : 25

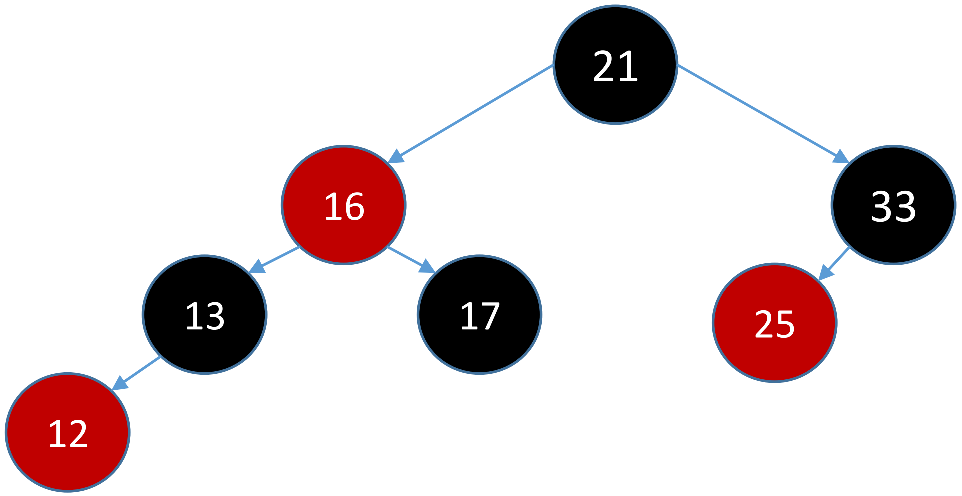


Insert : 12



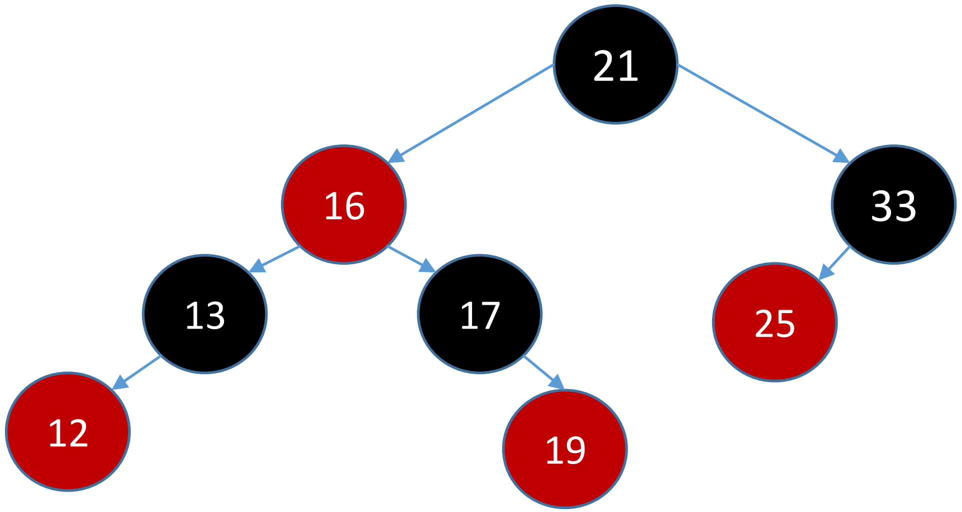
Case 1

Insert : 12

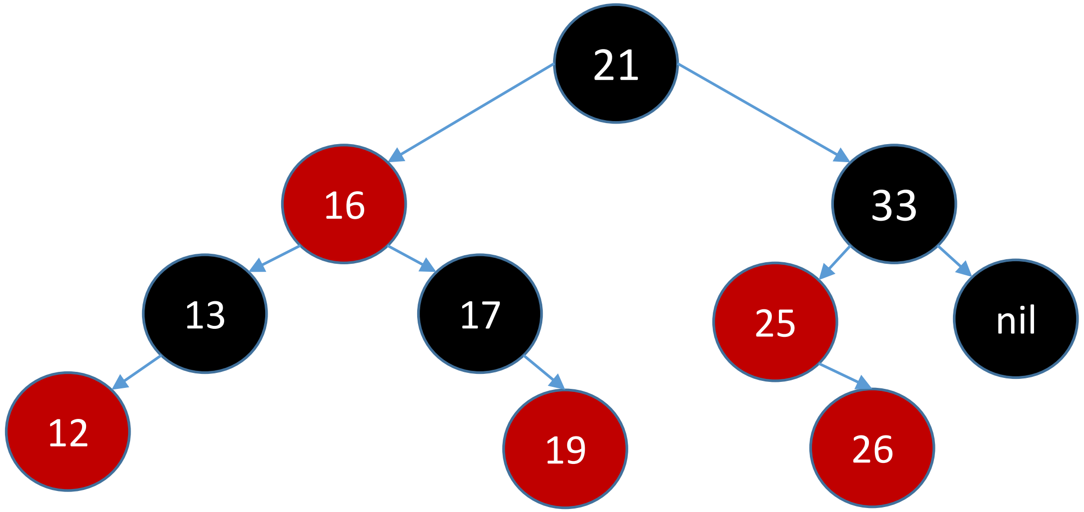


Fixed!!

Insert : 19

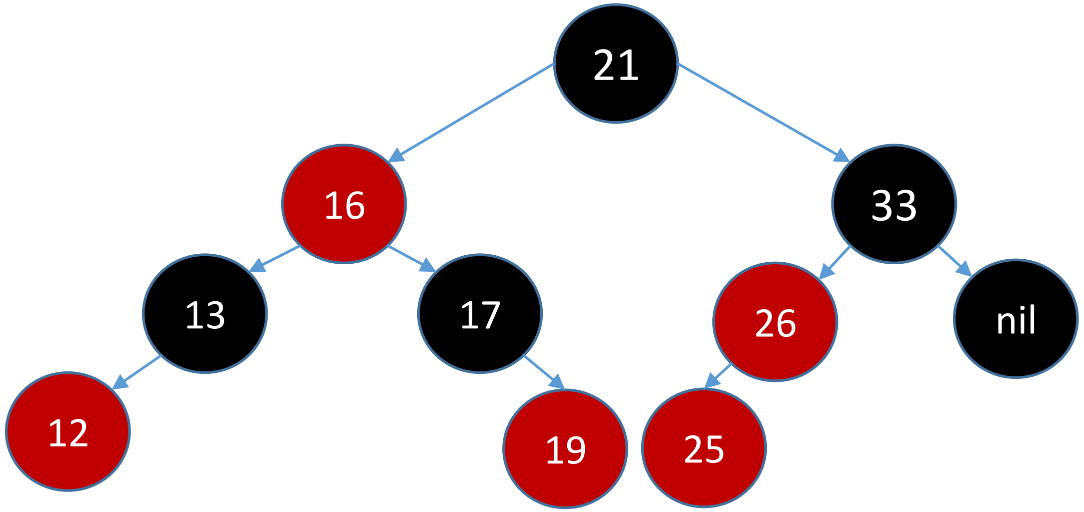


Insert : 26



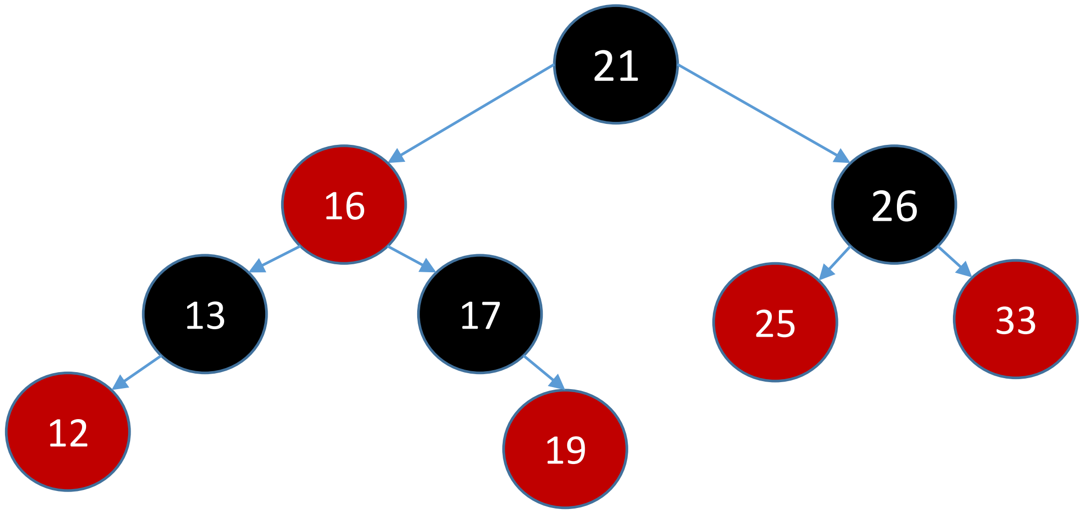
Case 3

Insert : 26



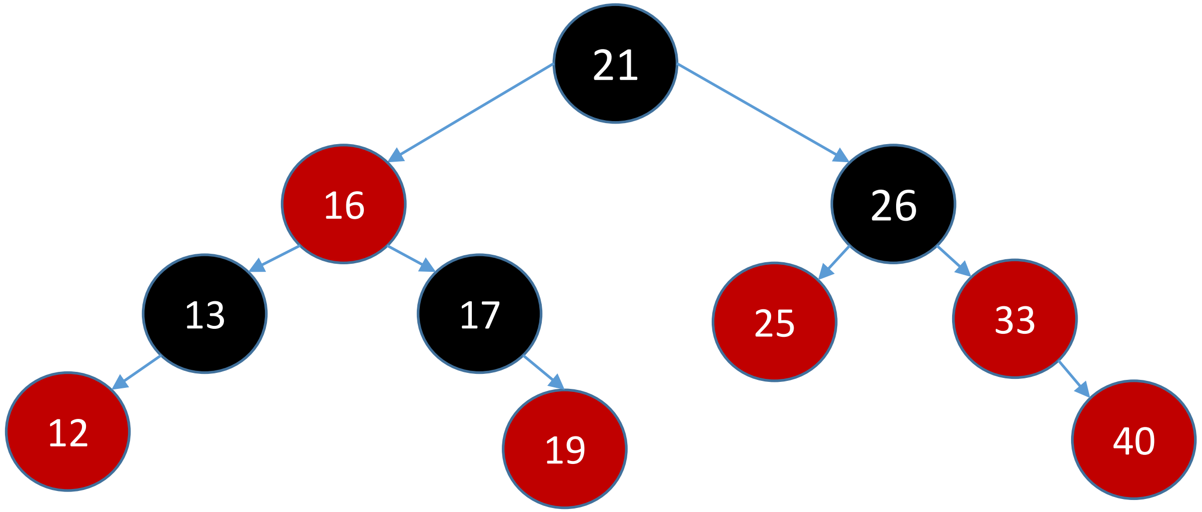
Case 2

Insert : 26



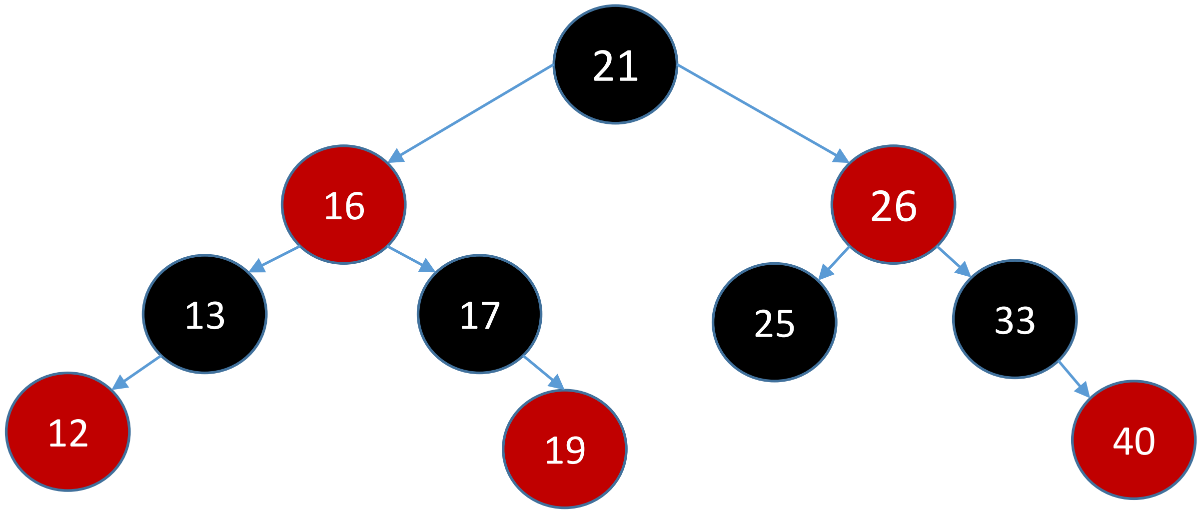
fixed

Insert : 40



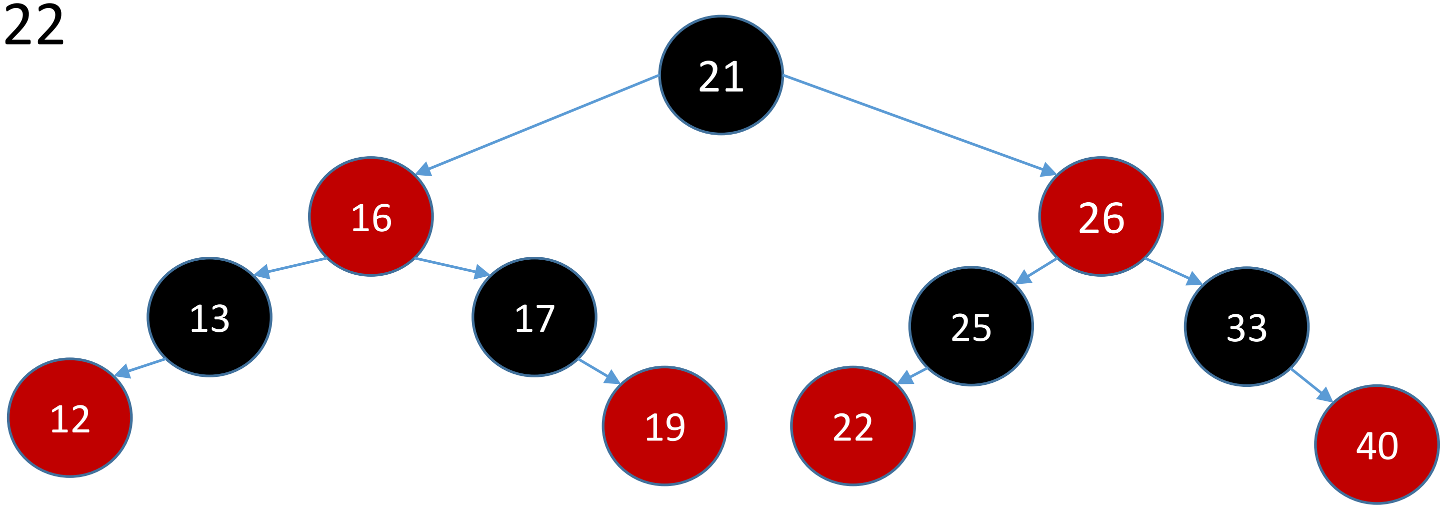
Case 1

Insert : 40



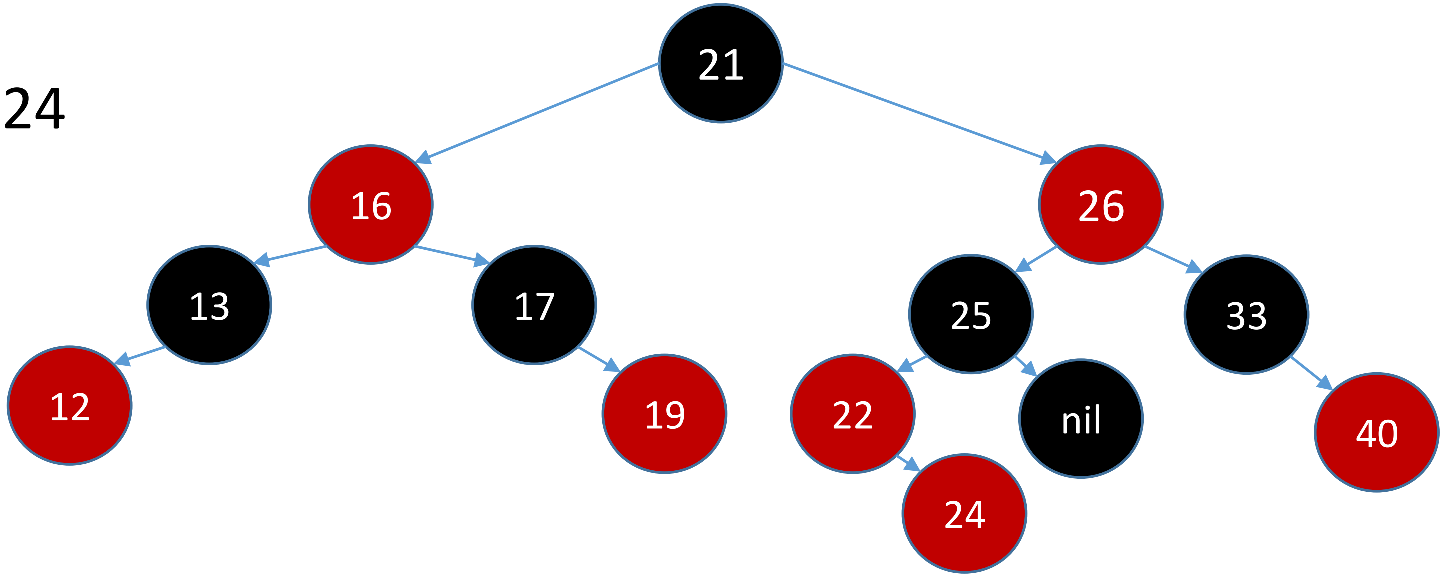
Fixed!!

Insert : 22



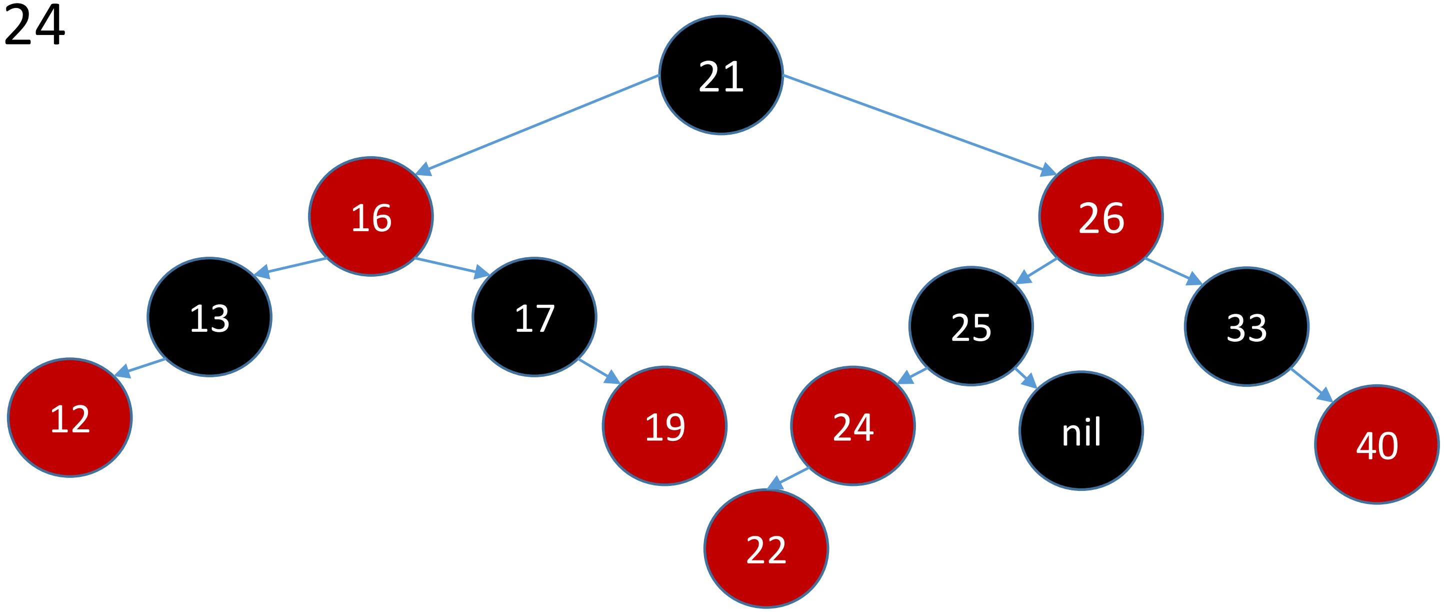
سوال اول:

Insert : 24



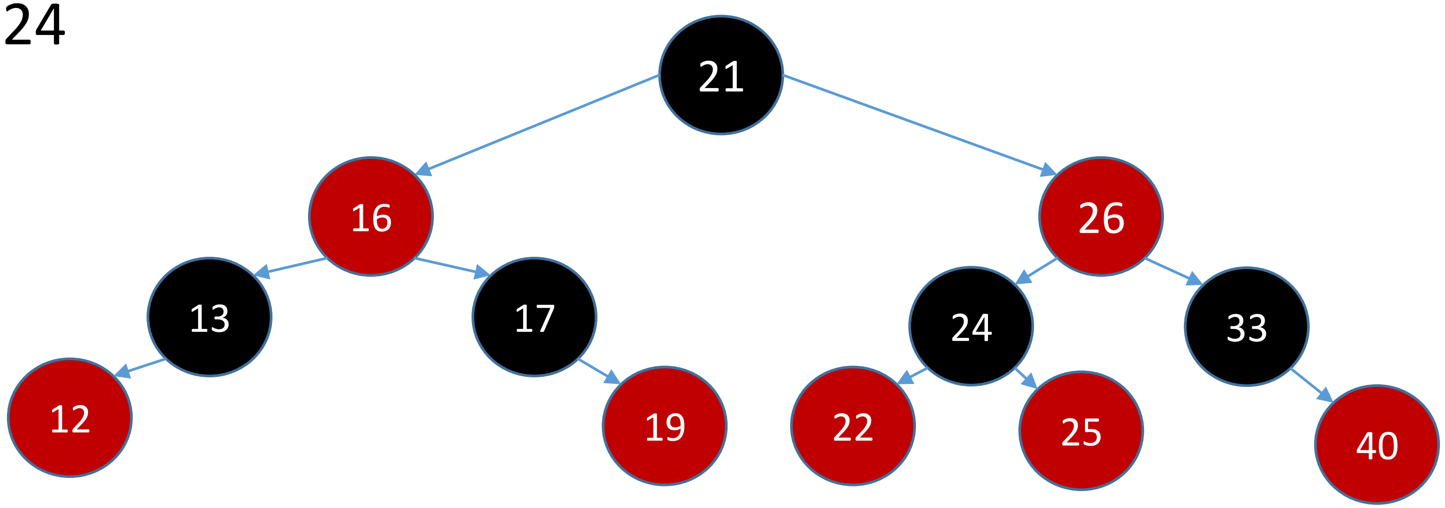
Case 2

Insert : 24



Case 3

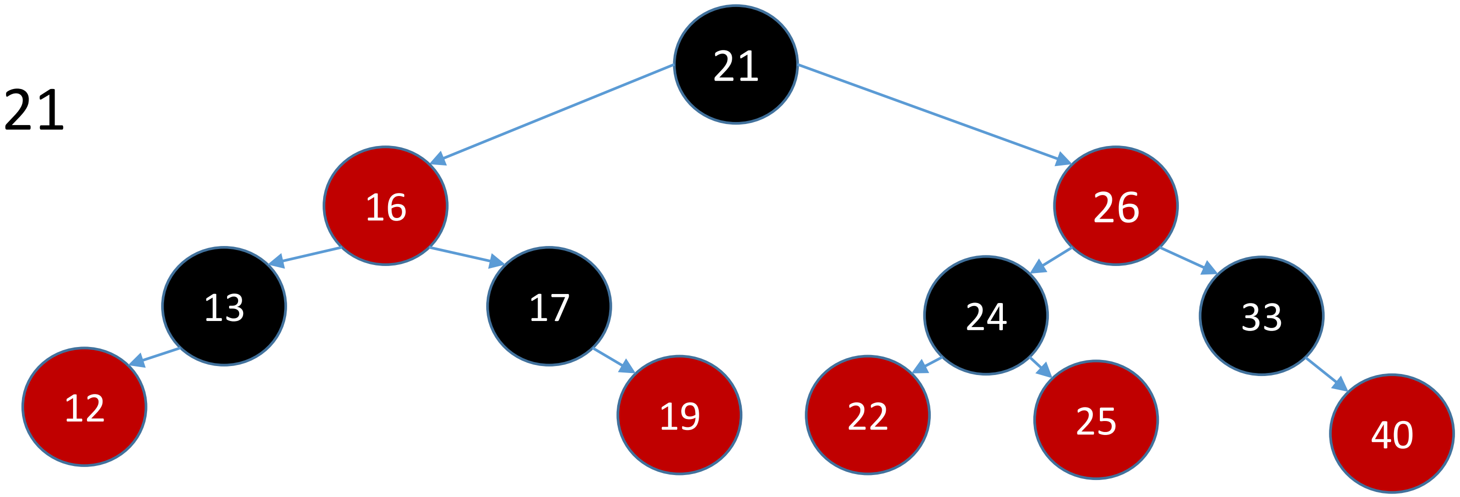
Insert : 24



Fixed!!

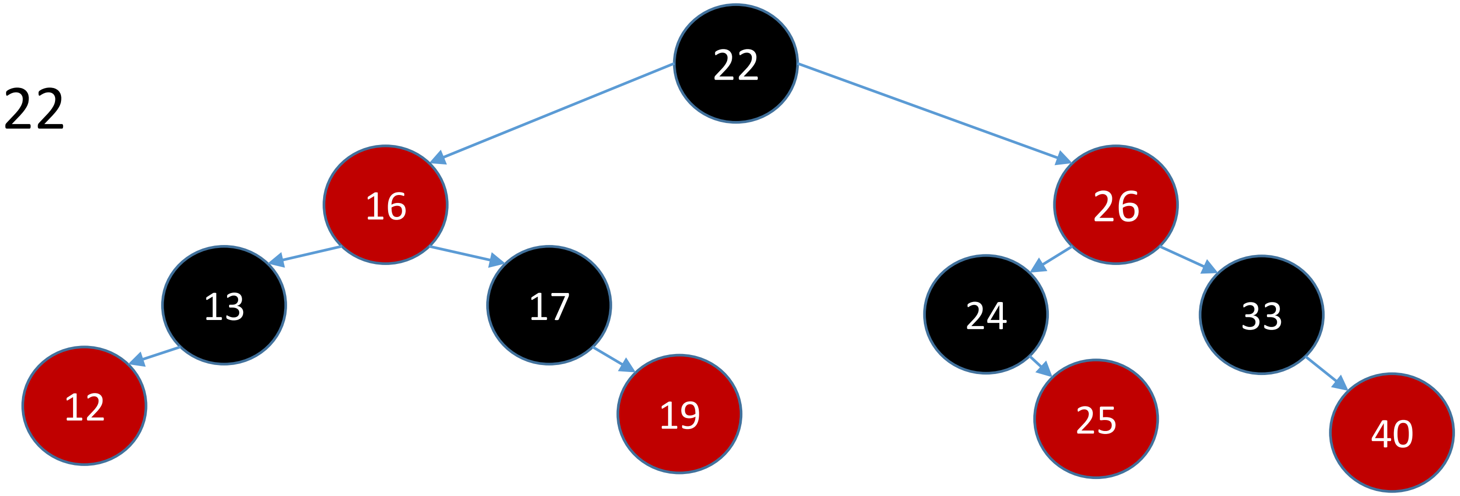
ب)

delete: 21



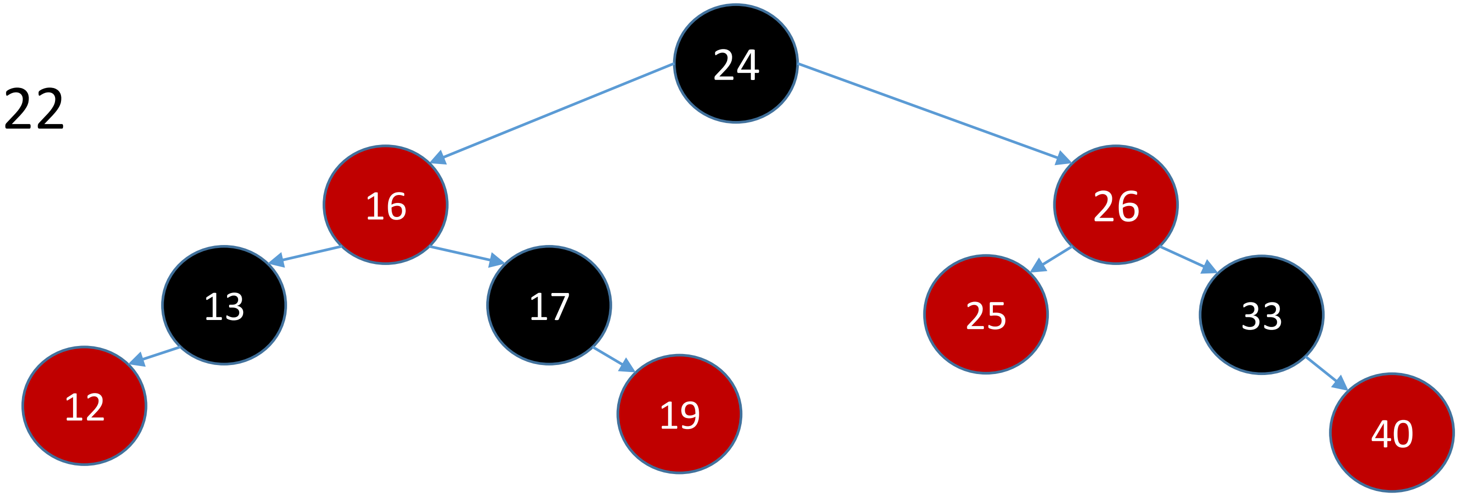
21 را با successor گره جابجا می کنیم و
گره جدید رنگ گره قبلی رو می گیره.

delete: 22



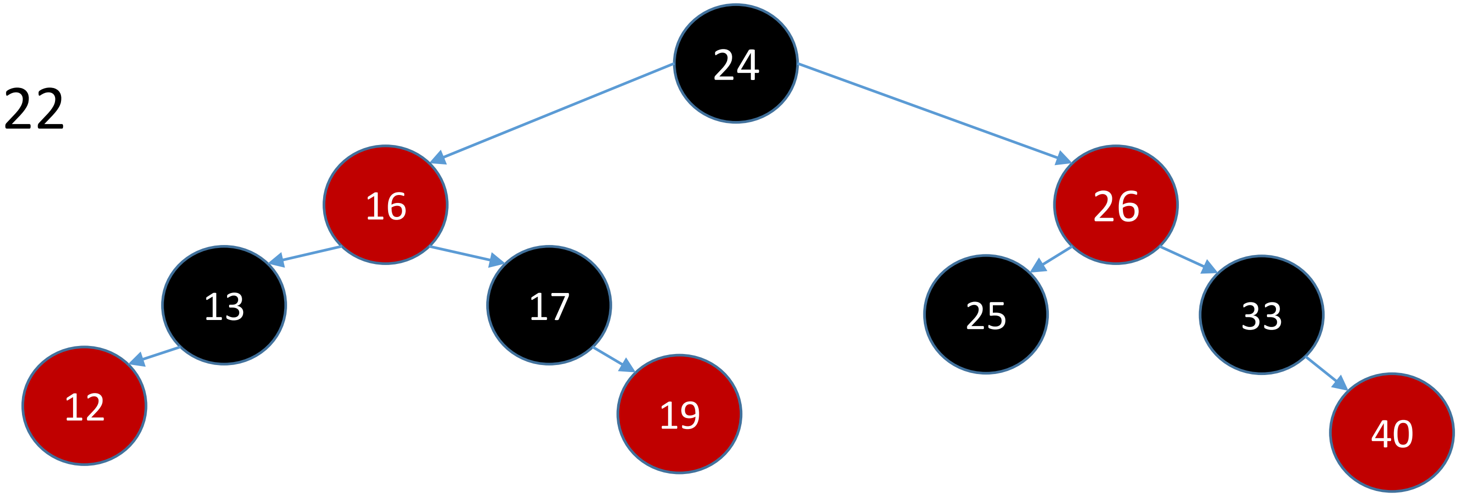
عنصر بعدی 22 ، 24 است و باید جای آن را با 22 عوض کنیم
و 24 رنگ 22 را می گیرد و ریشه زیر درخت سمت راست 24،
جای 24 رو می گیرد.

delete: 22

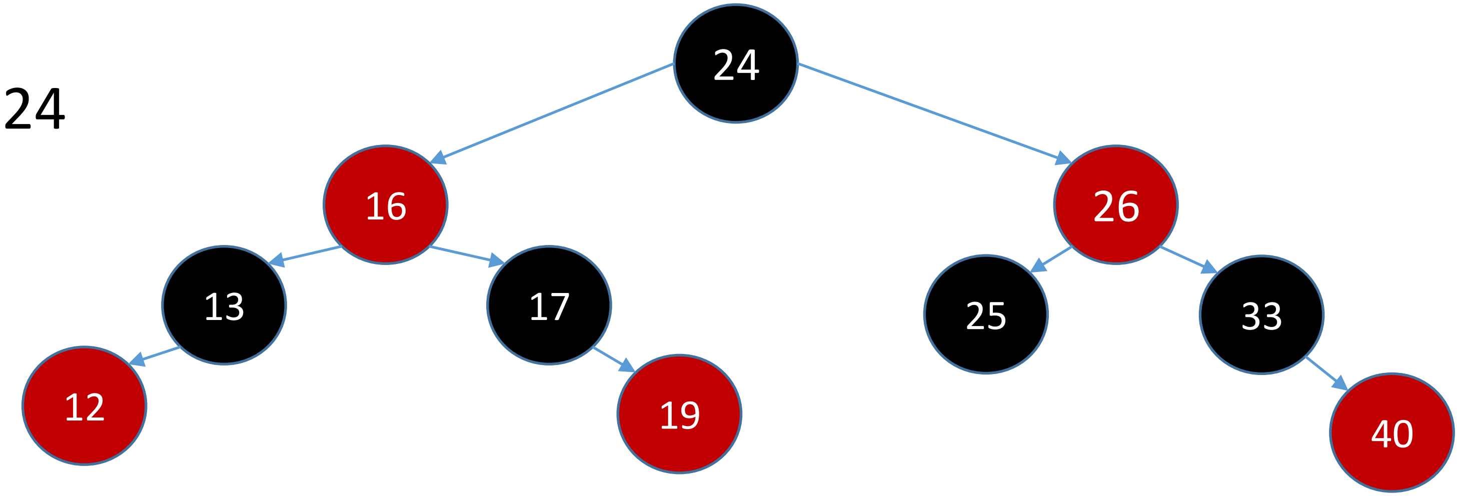


حال از اون جایی که رنگ گره 24 سیاه بوده، شرط پنجم RB نقض شده است.
از آنجایی که رنگ 25 قرمز بوده به هیچ کدام از case ها برنمی خوریم و کافی است رنگ آن گره را مشکی کنیم.

delete: 22

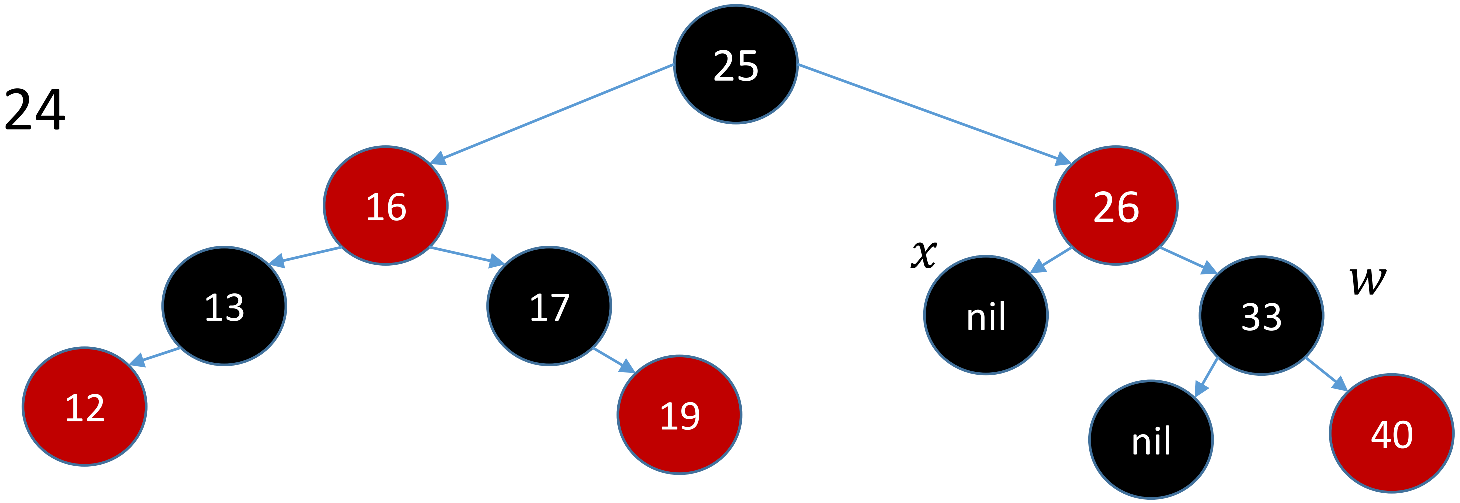


delete: 24



جای 25 با 24 عوض می‌شود و 25 رنگ 24 می‌گیرد.

delete: 24

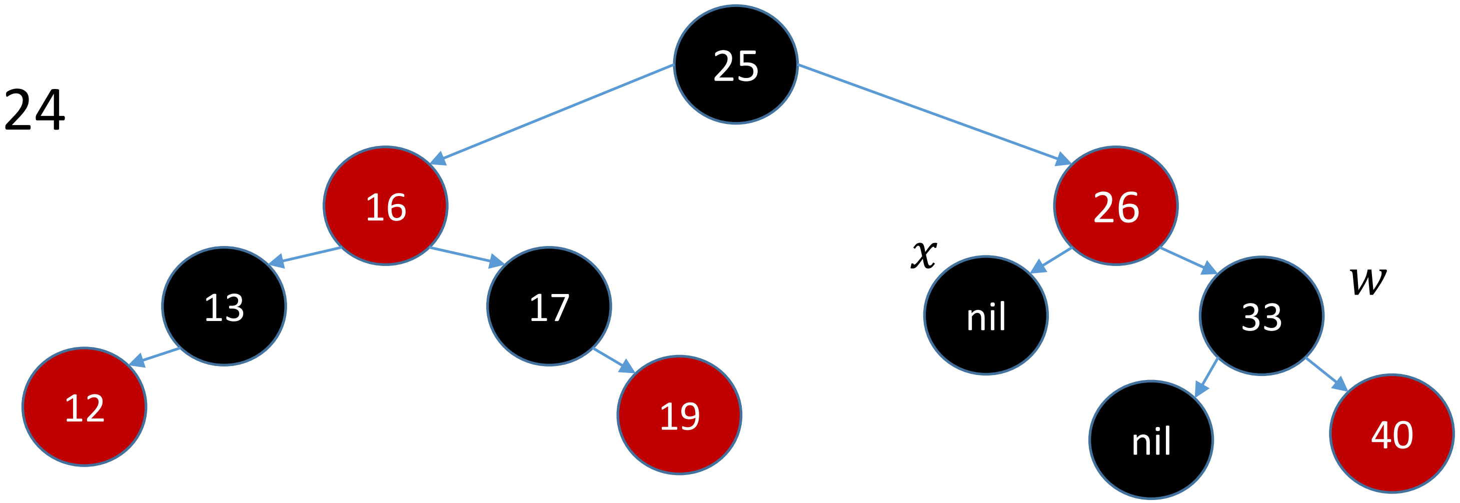


حال از آنجایی که رنگ 25 سیاه بوده است ، شرط 5 نقض شده است.

از آنجایی که گره ای که جای 25 را گرفته مشکی است (nil) ، یکی از case 4 پیش آمده است:

Case 4

delete: 24

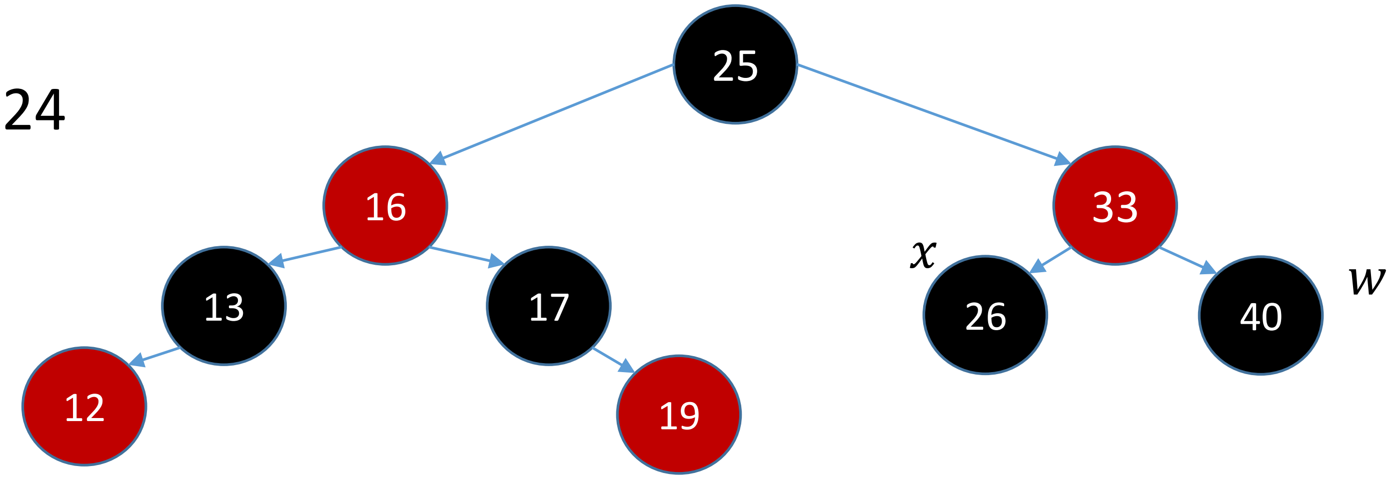


w رنگ parent خود را می‌گیرد

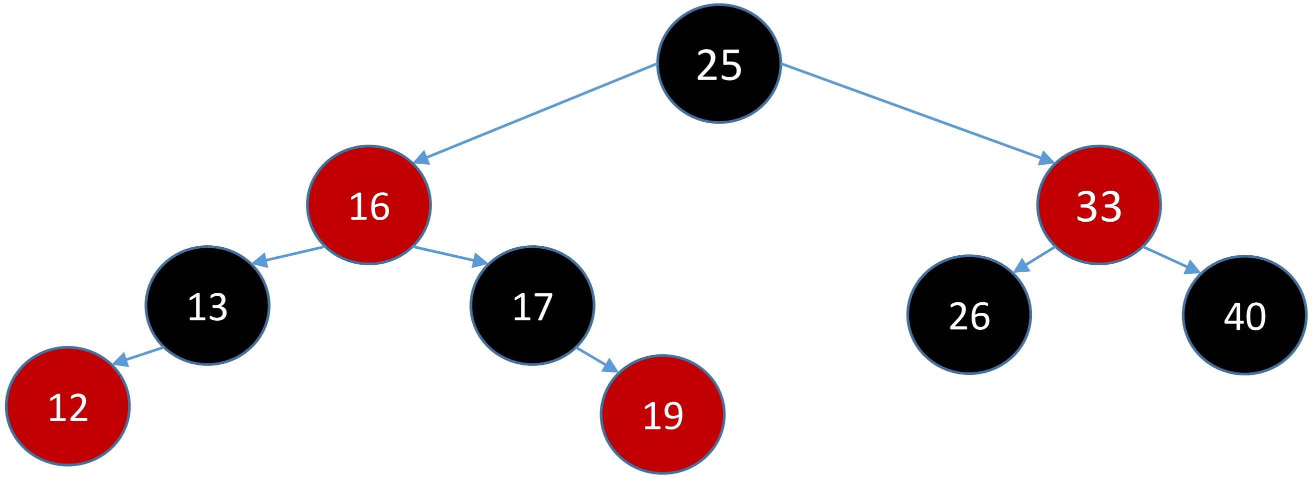
Parentوفرزند راست w مشکی می‌شود

بر روی 26، leftRotate را فراخوانی می کنیم.

delete: 24



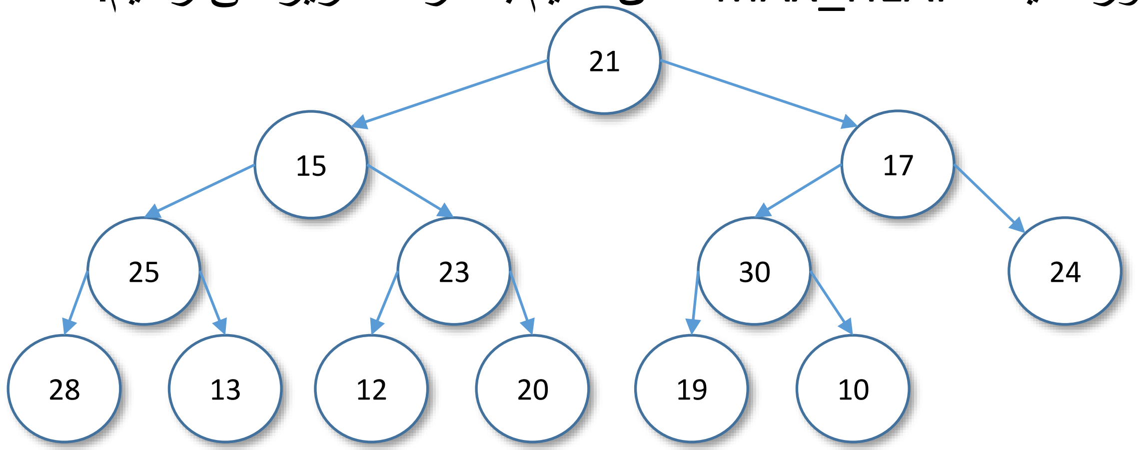
ج)



با توجه به درخت نهایی ارتفاع سیاه 2 و ارتفاع اصلی 4 است.

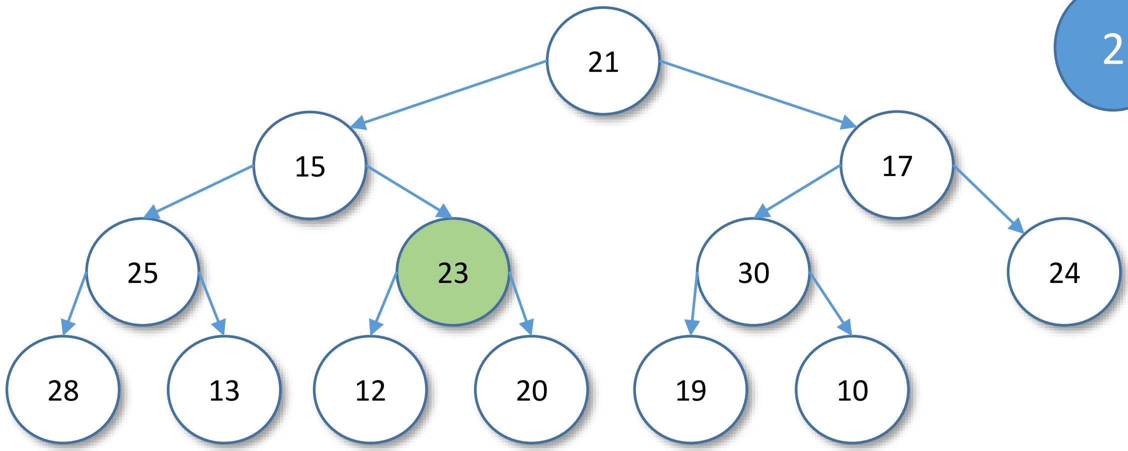
سوال 2 :

الف) اگر عناصر آرایه را به صورت یک MAX_HEAP نشان دهیم به درخت زیر می رسیم.

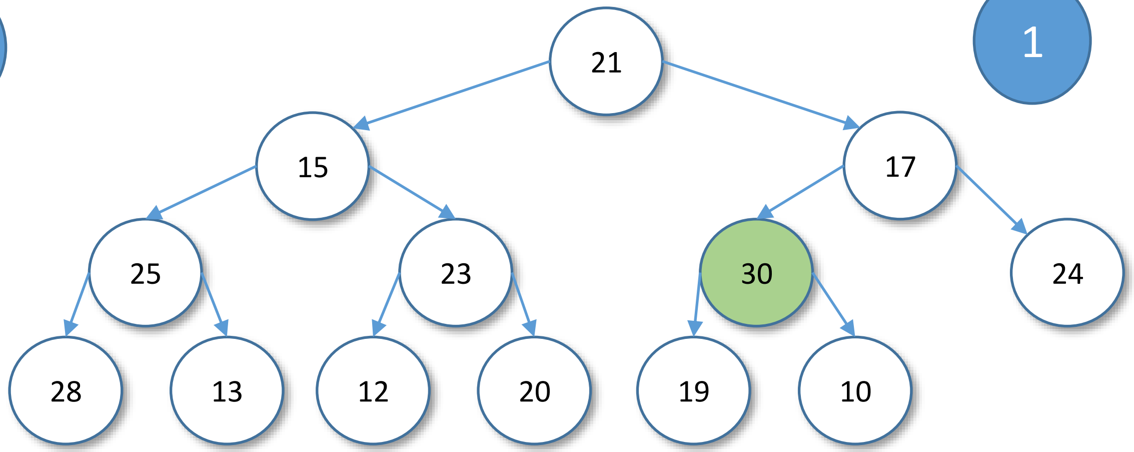


همانطور که می دانیم شرط هیپ بودن یک آرایه این است که هر گره از فرزند های خود بیشتر باشد در حالی که در اینجا گره 15 از فرزندان خود کمتر است.حال برای هیپ کردن این آرایه رویهٔ MAX_HEAPIFY را از پایین درخت برروی گره های غیر برگ فراخوانی می کنیم.

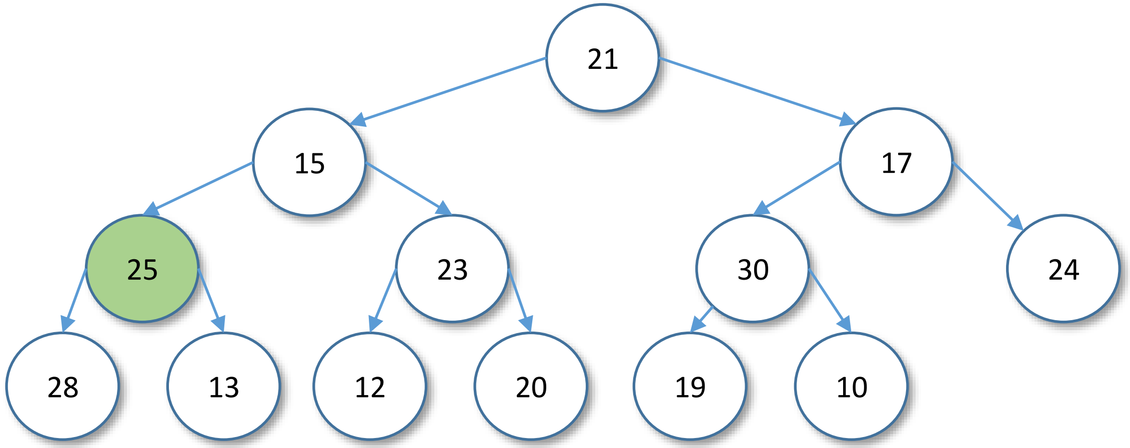
2

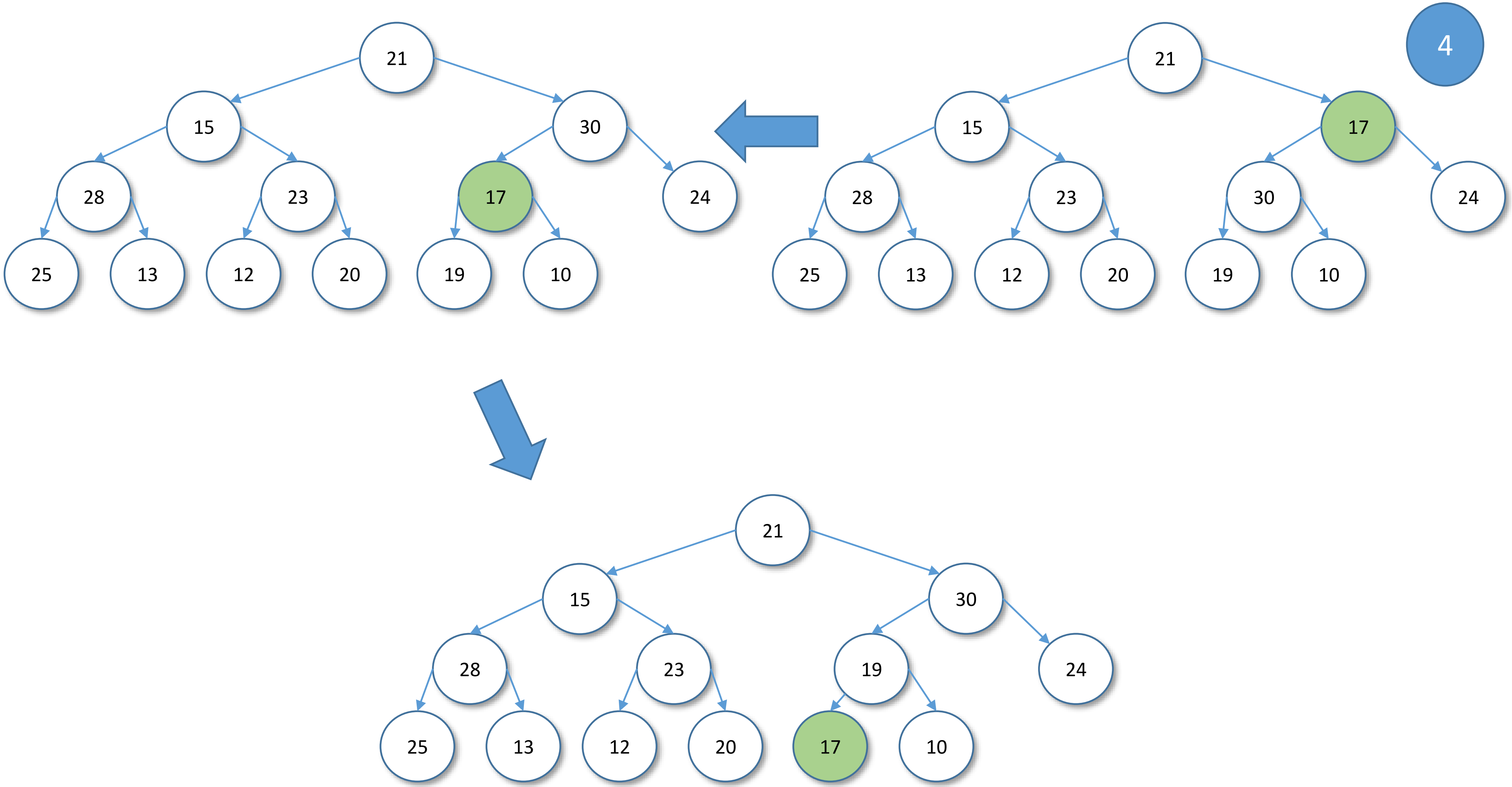

21
15
17
25
23
30
24
28
13
12
20
19
10

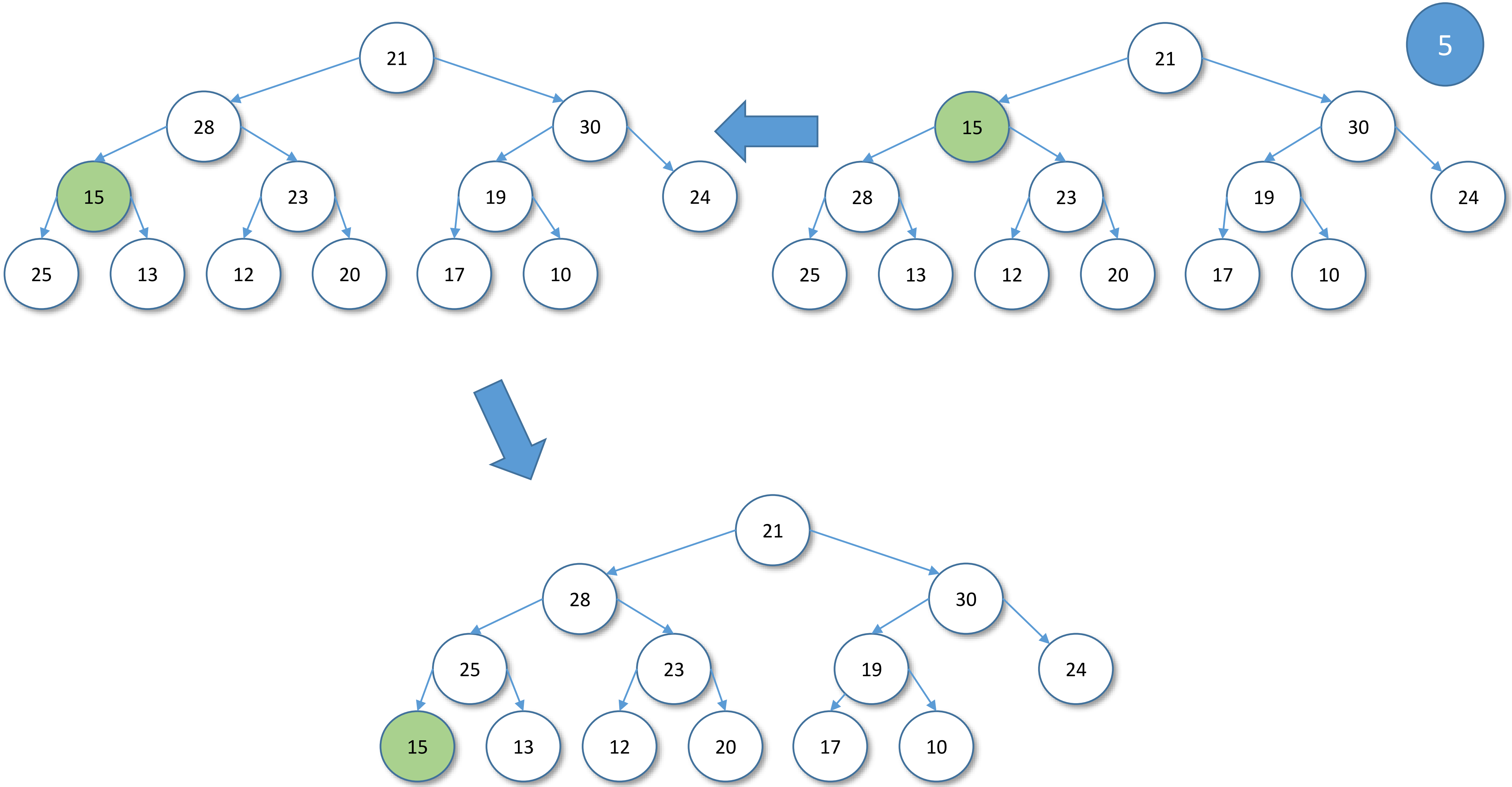
1


21
15
17
25
23
30
24
28
13
12
20
19
10

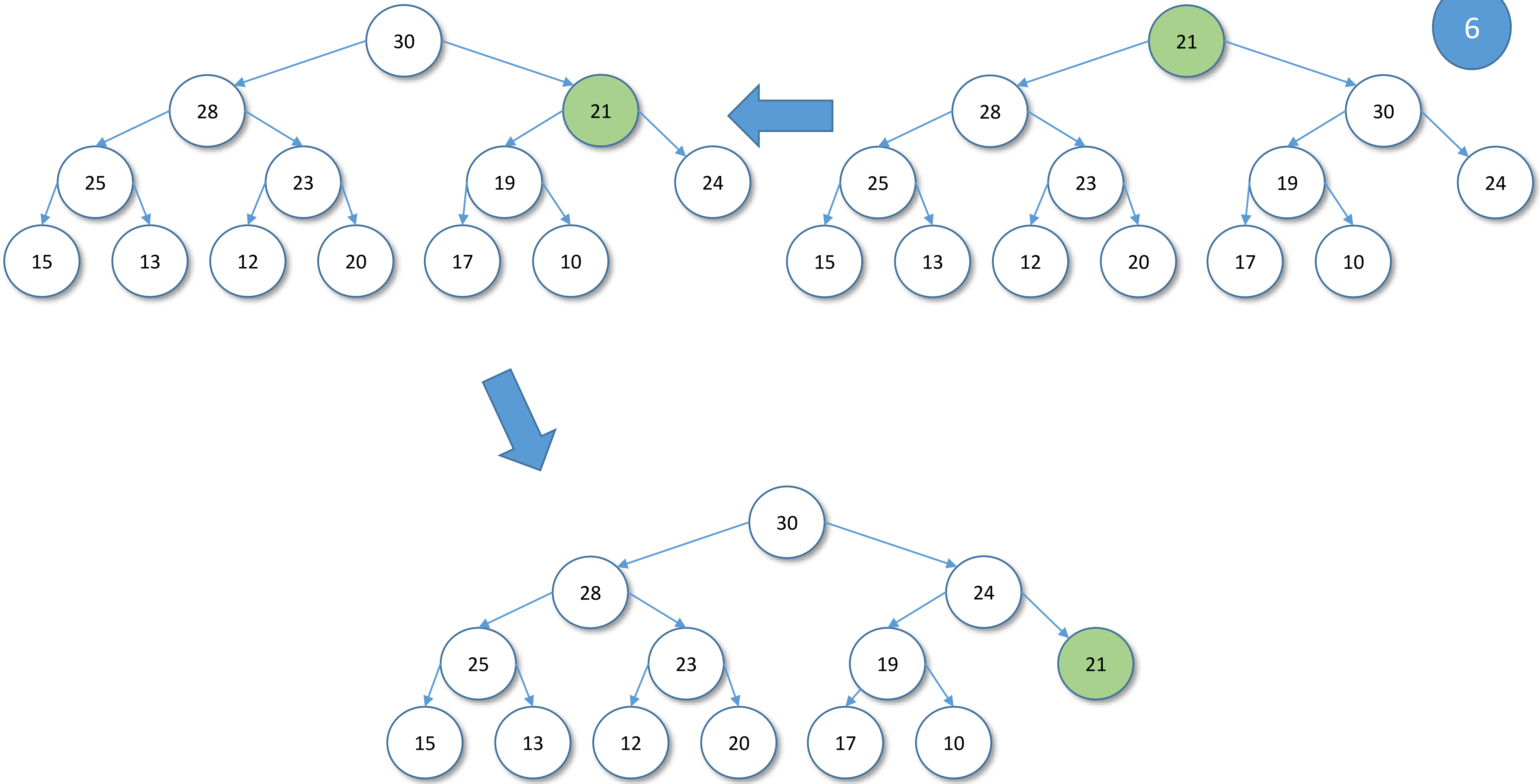
3


21
15
17
25
23
30
24
28
13
12
20
19
10

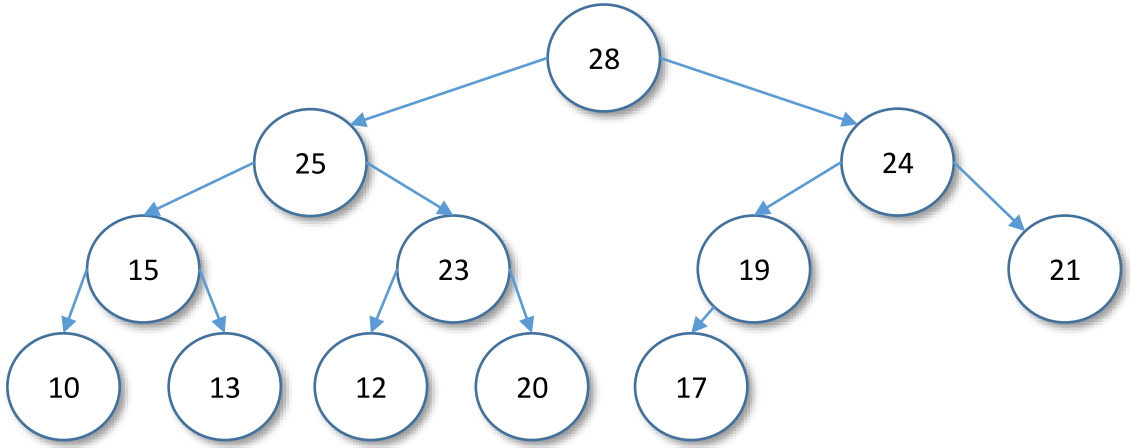

4
21
15 30
28 23 17 24
25 13 12 20 19 10
21
15 17
28 23 30 24
25 13 12 20 19 10
21
15 30
28 23 19 24
25 13 12 20 17 10


5
21
28
30
15
23
19
24
25
13
12
20
17
10
21
15
30
28
23
19
24
25
13
12
20
17
10
21
28
30
25
23
19
24
15
13
12
20
17
10

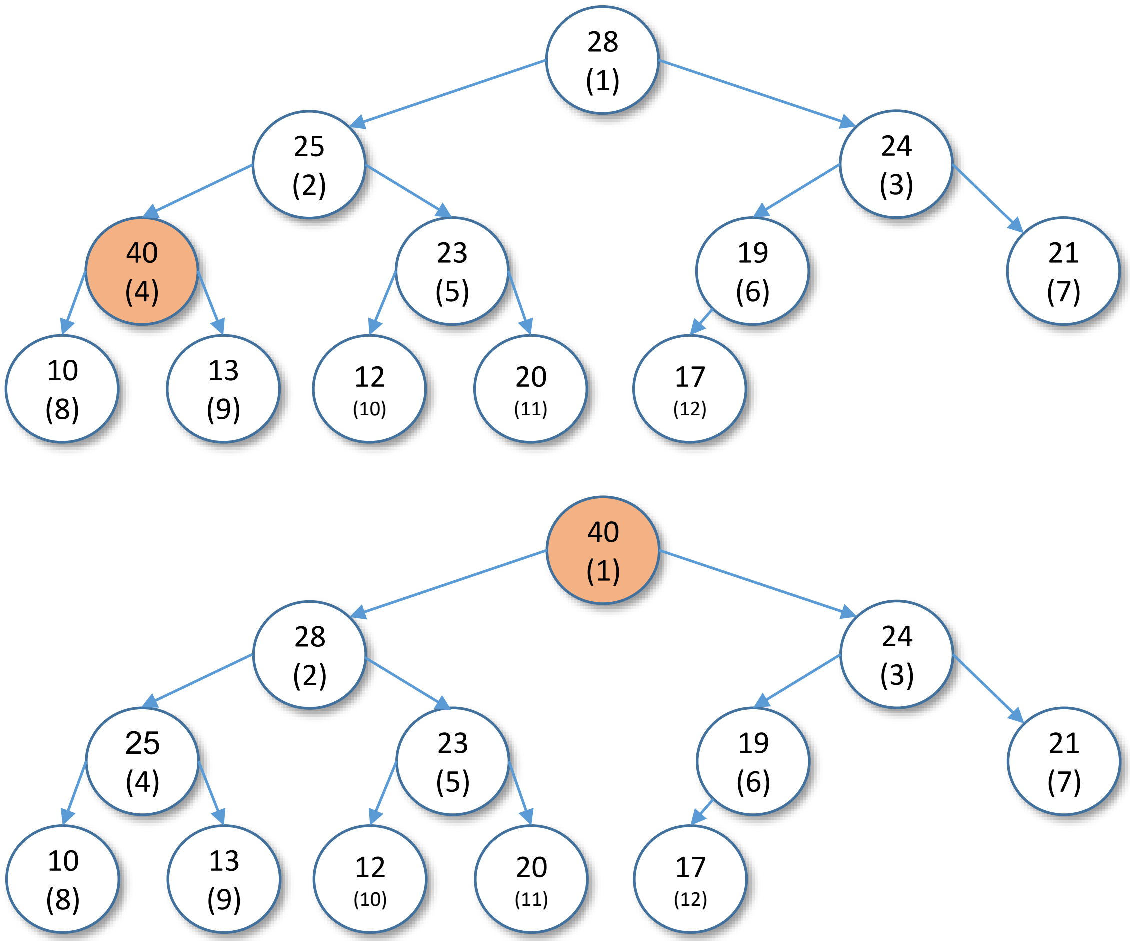
6


30
28
21
25
23
19
24
15
13
12
20
17
10
21
28
30
25
23
19
24
15
13
12
20
17
10
30
28
24
25
23
19
21
15
13
12
20
17
10

ب) 1.دراین فراخوانی ریشه هیپ pop شده و عنصر آخر را جای ریشه گذاشته و MAX-HEAPIFY را بر روی آن فراخوانی می‌کنیم.پس از فراخوانی درخت نهایی به شکل زیر است.



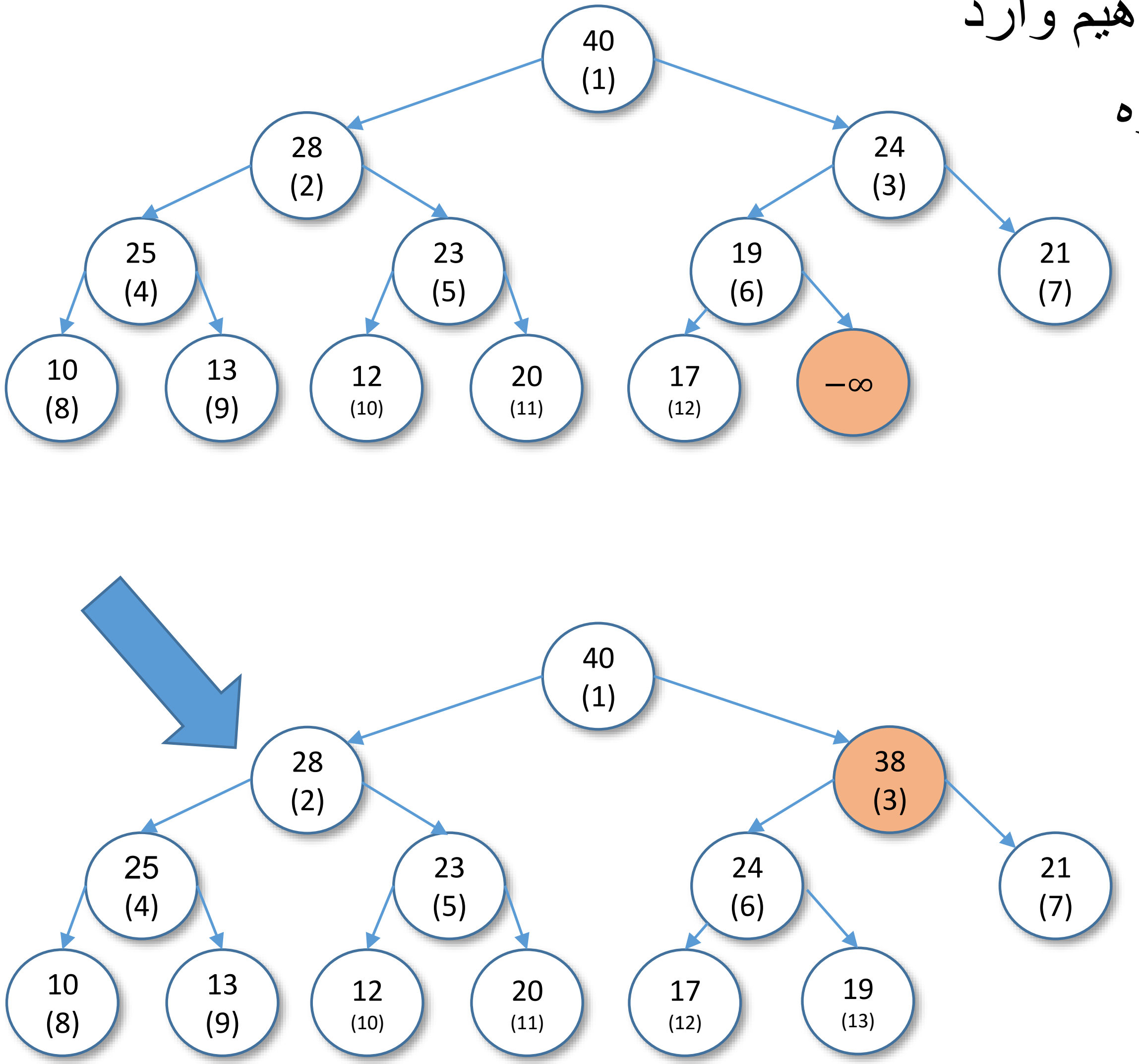
2.در این فراخونی خانه چهارم هیپ را به 40 افزایش می‌دهیم و تا جایی که گره از پدر خود بیشتر است بالا می‌رویم.



3.برای Insert، یک خانه با مقدار $-\infty$ در خانه آخر آرایه قرار می‌دهیم و HEAP_INCREAS_KEY را با مقداری که می خواهیم وارد

هیپ کنیم روی آن گره

فراخوانی می کنیم.



سوال 3 :

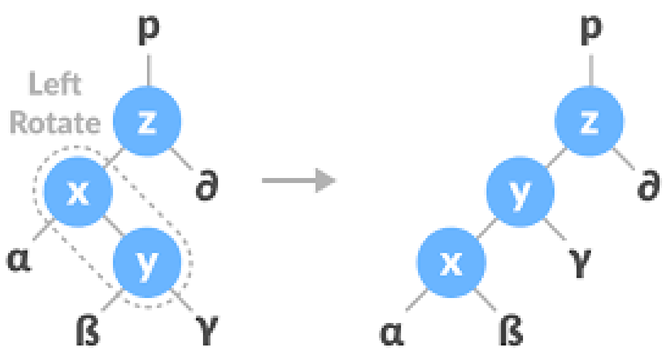
توابع Insert و Delete خود درخت قرمز- سیاه را می دانیم با هزینه زمانی O(log n) انجام می شوند پس مشکلی ندارند.
برای تابع Find نیز مانند Search درخت جستجو دوتایی عمل می کنیم و با توجه به اینکه در بدترین حالت باید ارتفاع درخت را طی کنیم و با در نظر گرفتن اینکه ارتفاع درخت قرمز – سیاه حتما از اردر O(log n) خواهد بود در نتیجه تابع Find نیز هزینه زمانی ای مشابه توابع Insert و Delete خواهد داشت.
تابع Count ما باید تعداد عناصر کوچکتر از X را بشمارد. توجه داشته باشید که ممکن ست X مقدار key هیچ کدام از نود های درخت ما نباشد.
برای اینکه این تابع هزینه زمانی O(log n) داشته باشد ما به صورت زیر عمل می کنیم.
ابتدا باید یک ویژگی به همه نود ها اضافه کنیم. ویژگی ای به نام Under Nodes که تعداد تمامی نودهایی که در درختی که ریشه (root) آن همان نود ماست وجود دارد. مثلا ریشه درخت قرمز – سیاه ما Under Nodes برابر با n می شود.

برای اینکه این ویژگی را مقدار دهی کنیم به صورت زیر عمل می کنیم :

- نکته : برای نود های nil مقدار Under Nodes را صفر در نظر می‌گیریم.

1. به هنگام Insert کردن، Under Nodes را برای نود جدید برابر 1 قرار می دهیم . بعد از نود جدید تا root درخت اصلی پیمایش می‌کنیم (به صورت x = parent[x]) و مقدار Nodes Under هر یک از نود ها را در مسیر+1 می‌کنیم.
2. هنگام Delete کردن، از نود پدر نودی که می خواهیم Delete کنیم شروع می کنیم و تا root درخت اصلی می‌رویم و مقدار Under Nodes را -1 می کنیم.
3. حال در هنگام Delete ممکن ست ما از عملیات های Rotate استفاده کنیم. دراسلاید بعد ما نشان می دهیم برای Left Rotate باید چیکار کنیم و برای Right Rotate مشابه Left Rotate باید عمل کنیم ولی به صورت قرینه

فرض می‌کنیم درخت ما به صورت زیر باشد:



1. از مقدار Under Nodes(X) مقدار Under Nodes(Y) را کم می کنیم و به ان مقدار Under Nodes(ß) را اضافه می‌کنیم
2. از مقدار Under Nodes(Y) مقدار Under Nodes(ß) را کم می‌کنیم و مقدار Under Nodes(X) (در مرحله قبل تغییر می‌کند) را به آن اضافه می‌کنیم.

با توجه با اینکه تمامی مراحل بالا در هزینه زمانی O (log n) انجام می شود در نتیجه به هزینه زمانی توابع Insert و Delete آسیبی نمی‌زند

حال به سراغ خود تابع Count می‌رویم :

1. ازمقدار Current_Node را برابر ریشه (root) و مقدار countLess را برابر 0 قرار می‌دهیم.
2. در این هر مرحله بررسی می‌کنیم که مقدار key نودی که روی آن هستیم از X بزرگتر ست یا نه

اگر مقدار key از X کمتر بود مقدار Under Nodes(left(Current_Node)+1 را به countLess اضافه می‌کنیم

و بعد Current_Node = right(Current_Node) می‌شود.

اگر مقدار key بزرگتر باشد فقط Current_Node = left(Current_Node) می‌شود.

3. درصورتی الگوریتم ما تمام می شود که یا key == X شود یا Current_Node == nil شود چون ممکن ست X اصلا در درخت ما وجود نداشته باشد.
4. در نهایت نیز اگر Find (X) مقداری به غیر از nil برگرداند (X برابر با key یکی از نود ها باشد) باید مقدار Under Nodes(left(Find(X))) را نیز به countLess اضافه کنیم.

با توجه به اینکه در این الگوریتم ما در بدترین حالت ارتفاع درخت را طی می‌کنیم در نتیجه هزینه زمانی تابع Count برابر با O (log n) می شود.

- شبه کد تابع Count :

```
Current_Node = root[rbt]
countLess = 0

while (key[Current_Node] != X or Current_Node != nil)
{
    if (key[Current_Node] < X)
    {
        countLess += under_nodes[left[Current_Node]] + 1
        Current_Node = right[Current_Node]
    }
    else
    {
        Current_Node = left[Current_Node]
    }
}
if (Find(X) != nil)
    countLess += under_nodes[left[Find(X)]

return countLess
```