به نام خدا

مرحله علیداری - تکلیف اول الگوریتم پیشرفته

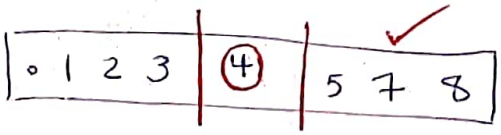
① آرایه دارای n عدد است. عدد $\left\lceil \frac{n+0}{2} \right\rceil$ را در نظر می‌گیریم و با یکی یکی بررسی می‌کنیم که آیا این عدد در آرایه وجود دارد یا خیر. اگر وجود نداشته باشد، این همان عدد گم شده است. (برنامه حل شده است. (شرط خاتمه)

اما اگر وجود داشته باشد، با یک پیمایش، آرایه را به این شکل به دو آرایه تقسیم می‌کنیم:

+ اگر n عددی زوج باشد، یک آرایه شامل اعداد بزرگتر از $\frac{n+0}{2}$ و یک آرایه شامل اعداد کوچکتر از $\frac{n+0}{2}$ تشکیل می‌دهیم. از بین این دو آرایه، آن آرایه‌ای که عناصر کمتری دارد را برای ادامه‌ی الگوریتم انتخاب می‌کنیم. و همین تکنیک‌ها را دوباره روی آن انجام می‌دهیم. به این شکل که:

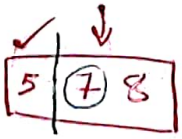
↳ اگر نمونه‌ی ما برای اعداد بزرگتر، تعدادش کمتر باشد، آن را به عنوان آرایه‌ی مورد نظر انتخاب می‌کنیم و آرایه‌ی دیگر را دور می‌ریزیم. برای این آرایه‌ی جدید بررسی می‌کنیم که آیا تعداد عناصرش فرد است یا زوج. ⊕ اگر زوج باشد، $\frac{n+\frac{n+0}{2}+1}{2}$ را محاسبه می‌کنیم، اگر وجود نداشت، همان عدد گم شده است. و برنامه حل شده است. و اگر وجود داشت، آن را به دو قسمت بزرگتر از این عدد و کوچکتر از این عدد، تقسیم می‌کنیم. ⊕ اگر فرد باشد، $\left\lceil \frac{n+\frac{n+0}{2}+1}{2} \right\rceil$ را محاسبه می‌کنیم. اگر وجود نداشت، همان عدد گم شده است. ولی اگر وجود داشت، آن را به دو آرایه یکی شامل خود این عدد و اعداد بزرگتر، و دیگری شامل اعداد کوچکتر، تقسیم می‌کنیم.

و این روند را ادامه می‌دهیم ...



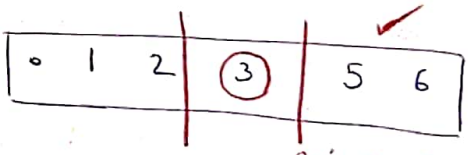
↳ n: زوج

$\frac{8+0}{2} = 4$



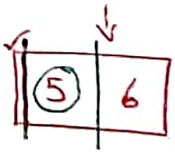
↳ تعداد عناصر: فرد

$\left\lceil \frac{8+\frac{8+0}{2}+1}{2} \right\rceil = \left\lceil \frac{13}{2} \right\rceil = 7$



↳ n: زوج

$\frac{6+0}{2} = 3$

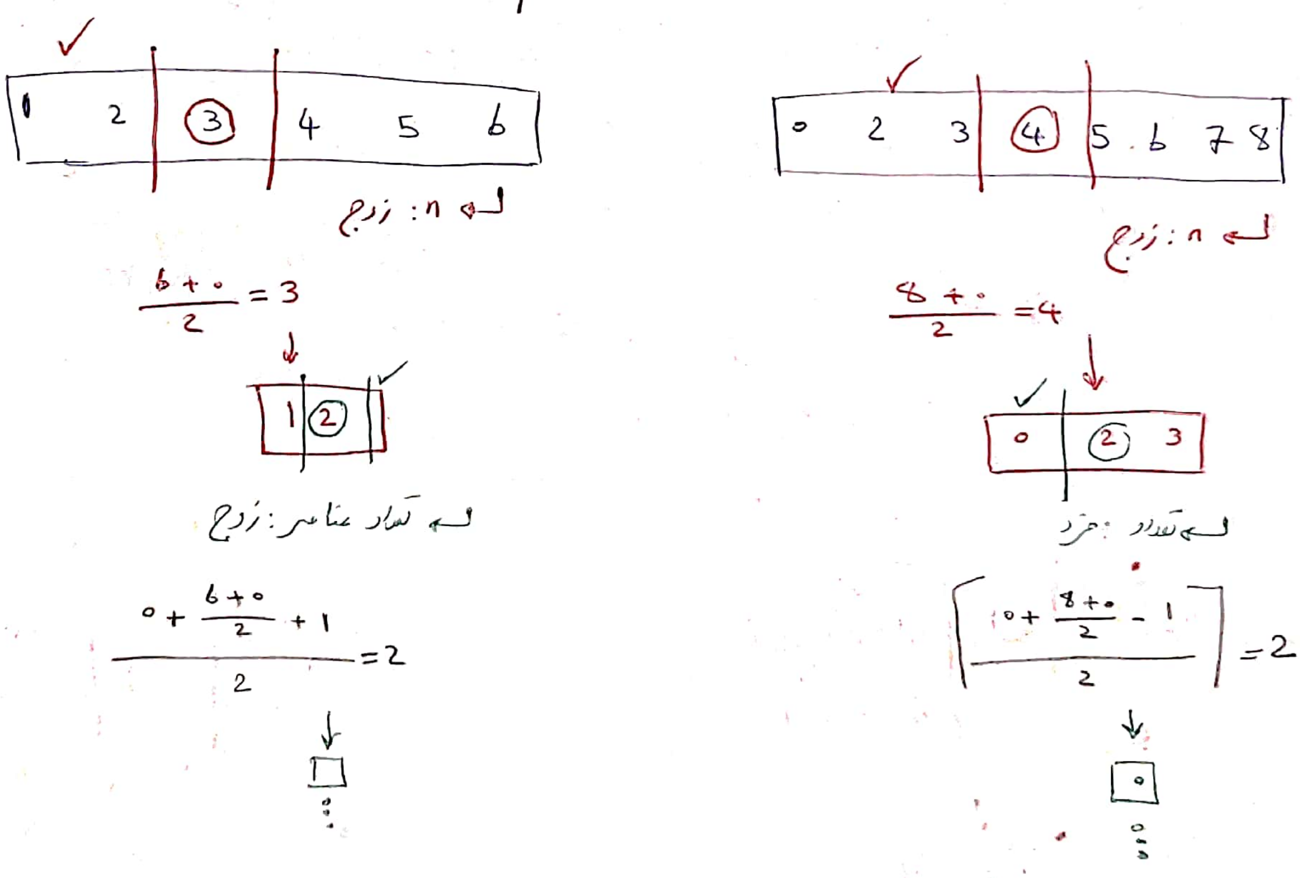


↳ تعداد عناصر: زوج

$\frac{6+\frac{6+0}{2}+1}{2} = 5$

page1

← یا اگر نیمه ی دارای اعداد کوچکتر، تعدادش کمتر باشد، آن را به عنوان آرایه ی مورد نظر انتخاب می‌کنیم و آرایه ی دیگر را دور می‌اندازیم. برای این آرایه ی جدید، زوج یا فرد بودن تعداد عناصرش را بررسی می‌کنیم. ⊕ اگر زوج باشد، $\frac{0 + \frac{n+0}{2} + 1}{2}$ را محاسبه می‌کنیم. اگر در آرایه وجود نداشت، این همان عدد گمشده است. اما اگر وجود داشت، آرایه را به دو بخش اعداد بزرگتر از این عدد و اعداد کوچکتر از این عدد، تقسیم می‌کنیم. ⊕ اگر فرد باشد، $\left[\frac{0 + \frac{n+0}{2} - 1}{2} \right]$ را محاسبه می‌کنیم، اگر در آرایه وجود نداشت، همان عدد گمشده است. ولی اگر وجود داشت، خود عدد و اعداد بزرگتر از آن را در یک آرایه، و اعداد کوچکتر از آن را در یک آرایه ی دیگر قرار می‌دهیم. و این روند را ادامه می‌دهیم...



+ اگر n عددی فرد باشد، یک آرایه شامل عدد $\left[\frac{n+0}{2} \right]$ و اعداد بزرگتر از آن و آرایه ای دیگر شامل اعداد کوچکتر از آن تشکیل می‌دهیم. از بین این دو آرایه، آرایه ای که تعداد عناصر کمتری دارد را برای ادامه ی الگوریتم انتخاب می‌کنیم. و همین عملیات را دوباره روی آن انجام می‌دهیم. بدین شکل که:

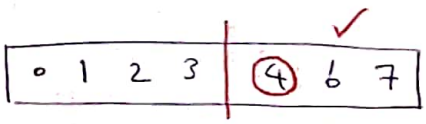
ادامه ①

← اگر نیمه ی داری اعداد بزرگتر، تعدادش کمتر باشد، آن را به عنوان آرایه ی مورد نظر انتخاب می کنیم و آرایه ی دیگر را دور می اندازیم. برای این آرایه ی جدید بررسی می کنیم که آیا تعداد عناصرش فرد است یا زوج.

⊕ اگر زوج باشد، $\frac{n+\left\lceil\frac{n+0}{2}\right\rceil}{2}$ را محاسبه می کنیم. اگر وجود نداشت، همان عدد کوچکتر جواب مسئله است. و برنامه حل شده است. اگر وجود داشت، اعداد بزرگتر از آن را در یک آرایه، و اعداد کوچکتر از آن را در آرایه ای دیگر قرار می دهیم.

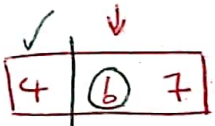
⊕ اگر فرد باشد، $\left\lceil\frac{n+\left\lceil\frac{n+0}{2}\right\rceil}{2}\right\rceil$ را محاسبه می کنیم. اگر وجود نداشت، همان عدد کوچکتر جواب مسئله است. و برنامه حل شده است. اما اگر وجود داشت، این عدد و اعداد بزرگتر از آن را در یک آرایه، و اعداد کوچکتر از آن را در یک آرایه ی دیگر قرار می دهیم.

و این روند را ادامه می دهیم ...



← n: فرد

$\left\lceil\frac{7+0}{2}\right\rceil = 4$



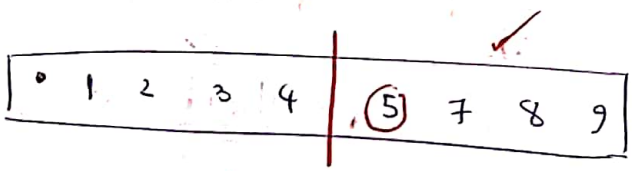
← تعداد عناصر: فرد

$\left\lceil\frac{7+\left\lceil\frac{7+0}{2}\right\rceil}{2}\right\rceil = 6$

↓

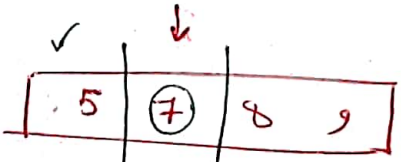
4

⋮



← n: فرد

$\left\lceil\frac{9+0}{2}\right\rceil = 5$



← تعداد عناصر: زوج

$\frac{9+\left\lceil\frac{9+0}{2}\right\rceil}{2} = 7$

↓

5

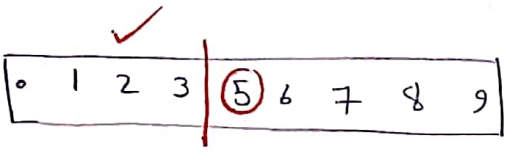
⋮

← یا اگر نیمه ی داری اعداد کوچکتر، تعدادش کمتر باشد، آن را به عنوان آرایه ی مورد نظر انتخاب می کنیم و آرایه ی دیگر را دور می اندازیم. برای این آرایه ی جدید، زوج یا فرد بودن تعداد عناصرش را بررسی می کنیم. ⊕ اگر

ادامه ①

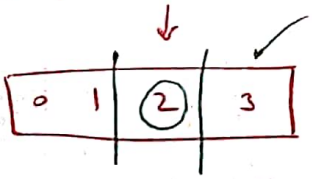
زوج باشد، $\frac{0+\lceil \frac{n+0}{2} \rceil - 1}{2} + 0$ را محاسبه می‌کنیم. اگر در آرایه وجود نداشت، همان عدد گم شده است. اما اگر وجود داشت، اعداد بزرگتر از این عدد را در یک آرایه و اعداد کوچکتر از آن را در یک آرایه دیگر قرار می‌دهیم.

⊕ اگر فرد باشد، $\left\lceil \frac{0+\lceil \frac{n+0}{2} \rceil - 1}{2} \right\rceil$ را محاسبه می‌کنیم. اگر وجود نداشت، همان عدد گم شده است. و بر پایه حال شده است. اما اگر وجود داشت، خود عدد و اعداد بزرگتر از آن را در یک آرایه، و اعداد کوچکتر از آن را در یک آرایه دیگر قرار می‌دهیم. و این روند را ادامه می‌دهیم ...



n : فرد

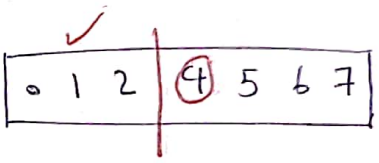
$\left\lceil \frac{9+0}{2} \right\rceil = 5$



تعداد عناصر : زوج

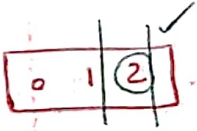
$\frac{0+\lceil \frac{9+0}{2} \rceil - 1}{2} = 2$

3



n : فرد

$\left\lceil \frac{7+0}{2} \right\rceil = 4$



تعداد عناصر : فرد

$\left\lceil \frac{0+\lceil \frac{7+0}{2} \rceil - 1}{2} \right\rceil = 2$

حل این آموزشی:

⊕ $T(n) = T(\frac{n}{2}) + \underbrace{O(n)}_{\approx n}$ $\rightsquigarrow T(\frac{n}{2}) = T(\frac{n}{4}) + \frac{n}{2}$ $\rightsquigarrow T(\frac{n}{4}) = T(\frac{n}{8}) + \frac{n}{4}$

$\rightsquigarrow \dots \rightsquigarrow T(2) = T(1) + 2 \rightsquigarrow T(1) = 0$

$\Rightarrow T(n) = T(\frac{n}{2}) + n = T(\frac{n}{4}) + \frac{n}{2} + n = \dots = T(\frac{n}{n}) + \frac{n}{\frac{n}{2}} + \frac{n}{\frac{n}{4}} + \dots + n =$

$= \cancel{T(1)}^{0} + 2 + 4 + \dots + \frac{n}{2} + n = n(\frac{2}{n} + \frac{4}{n} + \frac{8}{n} + \dots + \frac{1}{2} + 1) =$

$= n(\frac{1}{\frac{n}{2}} + \frac{1}{\frac{n}{4}} + \frac{1}{\frac{n}{8}} + \dots + \frac{1}{2} + 1) = \underline{O(n)}$ ✓

+ اگر آرایه a را داشته باشیم، a[i] یک local minimum باشد، مقدارش در مقایسه با همسایه هایش به این صورت است:

$$a[i-1] > a[i] < a[i+1]$$

که البته اگر a[i] در گوشه باشد، فقط با یک همسایه ای که دارد، مقایسه می شود:

a[0] a[n]

$$a[0] < a[1] \quad \text{or} \quad a[n-1] > a[n]$$

← با توجه به اینکه در سوال قید نشده که درایه های این آرایه به چه صورت است، نمی توان فرضی درباره ی مقادیر element ها داشت.

- اگر عناصر اجازه ی تکراری بودن داشته باشند، و برخی عناصر بیش از یکبار تکرار شده باشند، حتماً در بدترین حالت باید کل آن ها را بیمایش کنیم، تا بتوانیم minimum محلی را پیدا کنیم. که در این صورت الگوریتم از O(n) خواهد بود.

- اگر مقادیر تمامی عناصر با هم یکسان باشند، اصلاً minimum محلی در آرایه وجود ندارد. چرا که وقتی همه با هم برابرند، هیچ عنصری از همسایه هایش کوچکتر نیست. و برای اینکه این یکسان بودن را بتوان عناصر را هم بخواهیم بررسی کنیم، باید یکبار کل آرایه را بیمایش کنیم. که در این حالت هم الگوریتم از O(n) خواهد بود.

← اما اگر فرض کنیم که هیچ دو عنصر از این آرایه با هم یکسان نیستند، می توان آن را به صورت زیر با استفاده از یک روش D&C در O(log n) حل کرد:

⊕ عنصر وسط آرایه را در نظر می گیریم: $a[\lfloor \frac{n}{2} \rfloor]$. آن را با همسایه هایش یعنی $a[\lfloor \frac{n}{2} \rfloor + 1]$ و $a[\lfloor \frac{n}{2} \rfloor - 1]$ مقایسه می کنیم. 4 حالت ممکن است برای آن رخ بدهد:

1. $a[i-1] > a[i] < a[i+1]$ ⇐ در این صورت، a[i] همان minimum محلی است و مسئله حل شده است.

ادامه ②

2 $a[i-1] > a[i] < a[i+1]$ ⟸ در این صورت، می‌توان گفت که حتماً یک minimum محلی در نیمه‌ی سمت چپ آرایه وجود دارد.

چرا که اگر $a[i-1] > a[i-2]$ باشد، با توجه به اینکه $a[i-1] > a[i]$ هم برقرار است، عنصر $a[i-1]$ minimum محلی می‌شود. اگر این اتفاق نیفتد و $a[i-1] < a[i-2]$ باشد، باز هم به همین صورت بالا، در سمت چپ تر آرایه، احتمال minimum محلی شدن برای $a[i-2]$ وجود دارد. که باز هم اگر این اتفاق نیفتد، تا رسیدن به ابتدای آرایه، همین موارد برقرار است؛

اگر $a[1] > a[2]$ باشد، با توجه به اینکه $a[2] < a[3]$ هم برقرار است، $a[2]$، minimum محلی می‌شود. اگر این اتفاق نیفتد و $a[1] < a[2]$ باشد، حال این بار با توجه به اینکه $a[1] < a[2]$ است، اگر $a[1] < a[0]$ هم برقرار باشد، $a[1]$، minimum محلی است. اگر این اتفاق نیفتد و $a[0] > a[1]$ باشد، آنگاه $a[0]$ با توجه به اینکه دو گوشه کار دارد، با همین یک شرط، minimum محلی می‌شود. ⟸ پس اثبات شد که با شرایط گفته شده‌ی $a[i-1] > a[i] < a[i+1]$، حتماً در نیمه‌ی سمت چپ آرایه، حداقل یک minimum محلی وجود دارد.

3 $a[i-1] < a[i] > a[i+1]$ ⟸ در این صورت، مشابه شرط (2)، می‌توان گفت که حتماً یک minimum محلی در نیمه‌ی سمت راست آرایه وجود دارد. (اثبات هم دقیقاً مشابه حالت قبل انجام می‌شود. و همان دلایل، اینجا هم صدق می‌کند. منتها اینجا باید اثبات را به طرف نیمه‌ی راست انجام بدهیم نه به جای چپ)

4 $a[i-1] > a[i] < a[i+1]$ ⟸ در این صورت، همان مسئله‌ای که درباره‌ی دو شرط (2) و (3) بود، اینجا هم برای دو طرف راست و چپ برقرار است. و به طور مشابه اثبات می‌شود که در هر طرف حداقل یک minimum محلی وجود دارد. فرقی ندارد کدام سمت را انتخاب کنیم. یکی از دو نیمه‌ی راست یا چپ را انتخاب می‌کنیم و راه‌حل را در آن جا به همین شکل ادامه می‌دهیم.

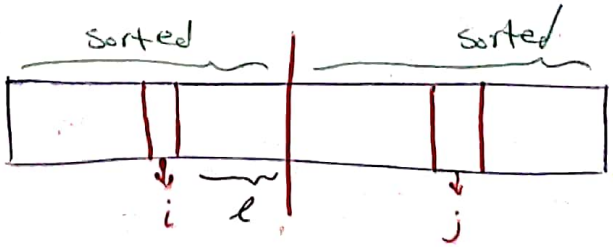
⟸ همین روند تکرار می‌شود ... هر بار آرایه نصف می‌شود ⟸ مرتبه $= O(\log n)$ ✓

برای حل این مسئله، مانند الگوریتم inversion، می‌توان از الگوریتم merge sort بهره گرفت. به این شکل که هر بار آرایه را از وسط به دو بخش تقسیم می‌کنیم. در مرحله‌ی آخر تقسیم کردن آرایه‌مان به آرایه‌های کوچکتر، هر آرایه فقط شامل یک عنصر است. در این مرحله شروع به conquer می‌کنیم و دوتا دوتا شروع به ساختن دوباره‌ی آرایه می‌کنیم. به این صورت که ابتدا دوتا را با هم مقایسه می‌کنیم و اگر لازم بود (اگر عنصر سمت راست، کوچکتر از عنصر سمت چپ بود)، به counter مربوط به عنصر سمت چپ، یک واحد اضافه می‌کنیم. و سپس آن‌ها را به صورت sort شده با هم merge می‌کنیم.

در مراحل جلوتر که آرایه‌ها بزرگتر شده‌اند، مقایسه‌ها به این شکل خواهد بود:

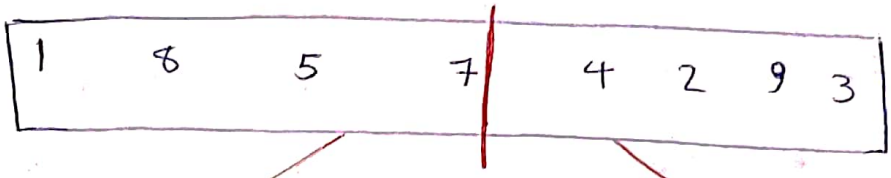
عناصر آرایه‌ی نیمه‌ی سمت راست را در نظر می‌گیریم. از چپ به راست، عنصرش را یکی یکی با تمام عناصر آرایه‌ی نیمه‌ی سمت چپ، از چپ به راست، مقایسه می‌کنیم. فرضاً عنصر x از آرایه‌ی سمت راست را داریم. همانگاه که داریم آن را با عناصر آرایه‌ی سمت چپ، از چپ به راست مقایسه می‌کنیم، در اولین جایی که یکی از عناصر از x بزرگتر است، به counter مربوط به آن عنصر و تمام عناصر سمت راست آن عنصر در همان آرایه‌ی سمت چپ، یک واحد اضافه می‌کنیم. چون آرایه‌ها به صورت صعودی sort شده هستند و عناصر سمت راست نیز از x بزرگتر خواهند بود.

عناصر موجود در یک آرایه یکسان، دیگر لازم نیست با هم مقایسه شوند؛ چرا که قبلاً قبل از اینکه به صورت sort شده با هم merge بشوند، ابتدا در جایگاه‌های واقعی خودشان با هم مقایسه شده‌اند و counter هایشان نسبت به هم update شده و سپس در یک آرایه sort شده‌اند.

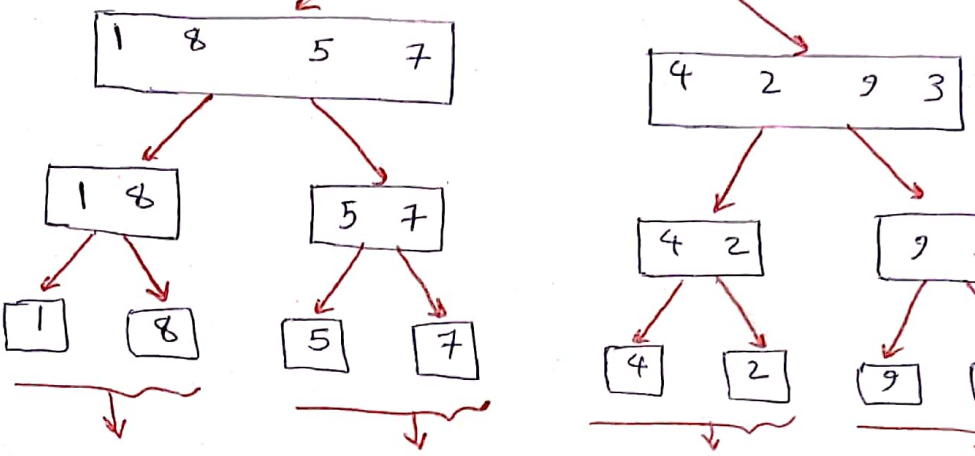


اگر $a[i] > a[j]$ باشد $\Rightarrow$ counter مربوط به عنصر i و تمام عناصر موجود در فاصله‌ی l، یک واحد اضافه می‌کنیم.

به مثال:



divide



array	1	8
Counter	0	0

array	5	7
Counter	0	0

array	2	4
Counter	0	1

array	3	9
Counter	0	1

conquer

5 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

5 با 8 مقایسه می‌شود ⇐ $Counter_8$++

7 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

7 با 8 مقایسه می‌شود ⇐ $Counter_8$++

array	1	5	7	8
Counter	0	0	0	2

3 با 2 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

3 با 4 مقایسه می‌شود ⇐ $Counter_9$++

9 با 2 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

9 با 4 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

array	2	3	4	9
Counter	0	0	2	1

4 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

4 با 5 مقایسه می‌شود ⇐ $Counter_5$++, $Counter_7$++, $Counter_8$++

9 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

9 با 5 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

9 با 7 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

9 با 8 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

2 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

2 با 5 مقایسه می‌شود ⇐ $Counter_5$++, $Counter_7$++, $Counter_8$++

3 با 1 مقایسه می‌شود ⇐ اتفاقی نمی‌افتد X

3 با 5 مقایسه می‌شود ⇐ $Counter_5$++, $Counter_7$++, $Counter_8$++

array	1	2	3	4	5	7	8	9
counter	0	0	0	2	3	3	5	1

✓

ادامه ③

در هر مرحله داریم از merge sort استفاده می‌کنیم. هزینه‌ی merge sort از $O(n\log n)$ است.

در این الگوریتم، علاوه بر merge sort، پیمایش هم برای n عنصر داریم. در این صورت، در هر مرحله، $O(n)$ داریم. و تعداد مراحل با توجه به اینکه آرایه هر بار به دو بخش تقسیم می‌شود، $\log n$ است.

پس هزینه‌ی پیمایش‌ها نیز $O(n\log n)$ است.

که در نتیجه داریم:

$$O(n\log n) + O(n\log n) = \boxed{O(n\log n)}$$ ✓

---

④ [a] بدترین دنباله باید به گونه‌ای باشد که بیشترین تعداد oddDelete را داشته باشد. پس باید تعداد insert ها به گونه‌ای باشد که در انتهای دنباله، تعداد عناصری که insert شده‌اند delete شده باشند.

پس هزینه‌ی بدترین دنباله با n operation به این شکل است:

$$\text{هزینه‌ی operation } n = \underbrace{i}_{\text{تعداد insert ها}} + \underbrace{i + \frac{i}{2} + \frac{i}{4} + \frac{i}{8} + \cdots + \left(\frac{i}{2^k} = \frac{1}{i} = 1\right)}_{\text{هزینه‌ی oddDelete ها}} =$$

$$= i + i\left(1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \cdots + \frac{1}{2^k}\right) = i + 2i = 3i$$

$$\frac{1}{1-\frac{1}{2}} = 2$$

$$\Longrightarrow \text{amortized cost} = \frac{3i}{i + \sum_{j=0}^{k} 1} \overset{\star}{\simeq} \frac{3i}{i} = 3 = \boxed{O(1)}$$ ✓

تعداد insert ها

تعداد oddDelete ها

که همواره از تعداد insert ها کمتر مساوی است.

پس $\sum_{j=0}^{k} 1$ مربوط به تعداد insert ها را می‌توان upper bound برای این هم استفاده کرد $\star$

$i + \sum_{j=0}^{k} 1$ $\star$

ادامه (4)

b

+ actual costs:

$c_{insert} = 1$

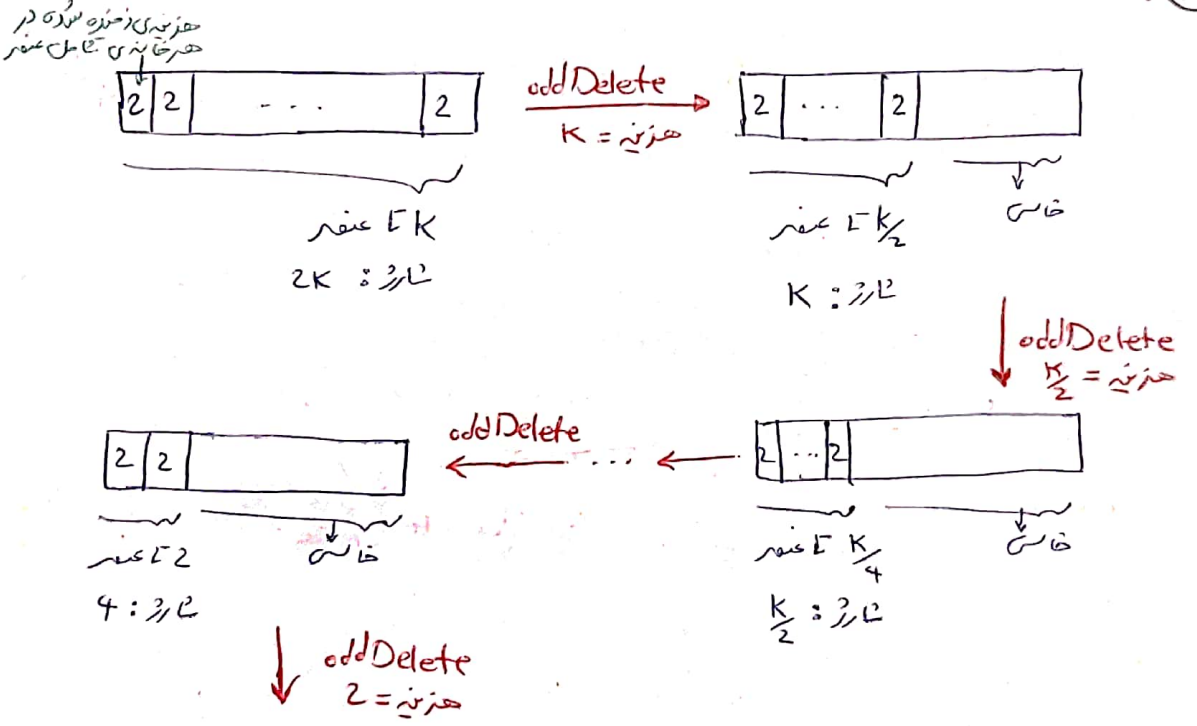
$c_{oddDelete} = k$ → به تعداد عناصر موجود در آرایه

+ amortized costs:

$c'_{insert} = 3\$$

$c'_{oddDelete} = 0\$$

← به ازای هر عنصری که insert می‌کنیم، 3$ شارژ می‌کنیم. یکی برای insert خودش هزینه می‌شود، دو تا باقی می‌ماند. اثبات می‌شود که همواره به ازای هر دنباله‌ای از عملیات، به اندازه‌ی کافی شارژ خواهیم داشت برای هزینه کردن:



لذا بدین ترتیب اثبات می‌شود همواره حداقل به اندازه‌ی کافی credit برای انجام operation های دلخواه داریم. ∎

ادامه‌ی ④

c

$\Phi_i = ($تعداد عناصر پس از انجام عملیات i ام$) \times 2$

$+ c'_{insert} = c_{insert} + \Delta\Phi = c_{insert} + \Phi_i - \Phi_{i-1} = 1 + 2(K) - 2(K-1) =$

(K: تعداد عناصر پس از انجام عمل i ام)

$= 1 + 2K - 2K + 2 = 1 + 2 = \boxed{3} \longrightarrow \boxed{O(1)}$ ✓

$+ c'_{oddDelete} = c_{oddDelete} + \Delta\Phi = c_{oddDelete} + \Phi_i - \Phi_{i-1} =$

$= \underbrace{K}_{\text{تعداد عناصر قبل از انجام عمل oddDelete}} + 2\underbrace{(\frac{K}{2})}_{\text{تعداد عناصر پس از انجام عمل oddDelete}} - 2(K) = K + K - 2K = \boxed{0} \longrightarrow \boxed{O(1)}$ ✓

---

⑤ a) اگر تضمین کنیم که پشتیبان گیری زمانی اتفاق می‌افتد، که تعداد مقادیر insert شده، تقریباً برابر n باشد، هزینه‌ی سرشکن شده، $O(1)$ خواهد بود.

⇐ فرضاً یک دنباله‌ی n تایی از operation ها در نظر می‌گیریم. $n-1$ عملیات اول، insert است. و عمل n ام، عمل پشتیبان گیری است. هزینه به این شکل خواهد بود:

$$\text{amortized cost} = \frac{(n-1) + n}{n-1+1} = \frac{2n-1}{n} \cong 2 = \boxed{O(1)}$$ ✓

(⇐ اگر این فرض را در نظر نمی‌گرفتیم، و مثلاً به این شکل پشتیبان گیری می‌کردیم، که پس از هر insert، یک پشتیبان می‌گرفتیم، هزینه به این شکل می‌شد، که مطلوب ما نیست:

$$\frac{n + n \times n}{n+n} = \frac{n^2+n}{2n} = O(n)$$ ✗

ادامه (5)

b

Φ = تعداد عناصر backup گرفته نشده در آرایه +

$$c'_{backup} = c_{backup} + \Delta\Phi = c'_{backup} + \Phi_i - \Phi_{i-1} =$$

$$= n + 0 - n = 0 = O(1)$$ ✓

6

+ amortized costs:

$c'_{0\to1} = 2\$$

$c'_{0\to-1} = 2\$$

$c'_{1\to0} = 0\$$

$c'_{-1\to0} = 0\$$

⊕ Φ_i = تعداد 1ها + تعداد 1- های عدد

← مطابق این محاسبات و فرضیات، در ابتدا

برای اینکه $\Phi_0 = 0$ باشد، باید $\underbrace{0\,0\ldots0}_{k\text{ بیت}}$

شروع کرد. و دلیل دیگر برای شروع از این عدد،

این است که اولین عملیاتی که انجام می‌شود، باید

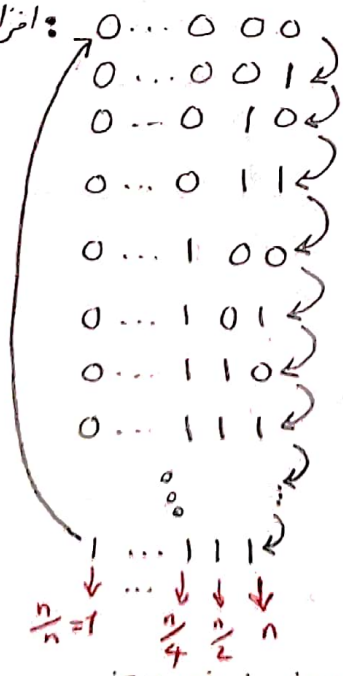
توانایی پرداخت هزینه‌ی خود را با استفاده از credit ای که

به آن می‌دهیم، داشته باشد. و در این حالت، با increment، یک بیت 0، 1 می‌شود.

و یا با decrement، یک بیت 0، 1- می‌شود که در هر دو حالت، هزینه‌ی این switch

ها با بار اولیه خودشان پرداخت می‌شود.

⊕ افزایش:

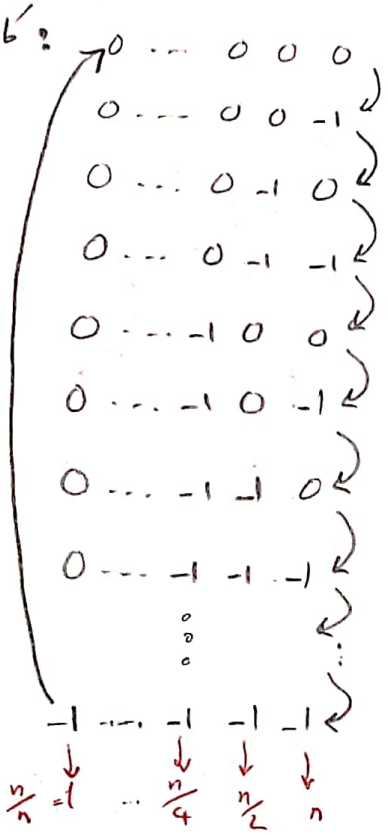


بیشترین تغییر ← دنباله‌ی increment

$$c'_{inc} = c_{inc} + \Delta\Phi = c_{inc} + \Phi_i - \Phi_{i-1} =$$

$$= \underbrace{t_i + 1}_{\text{تعداد بیت‌هایی که Flip می‌کنیم}} + \underbrace{1 - t_i}_{\substack{\text{تعداد 1ها} \\ \text{اضافه شده} \\ (t_i \text{ تا 1، 0 شد} \\ \text{و یک 0، 1 شد})}} = 2 = O(1)$$ ✓

⊕ کاهش:

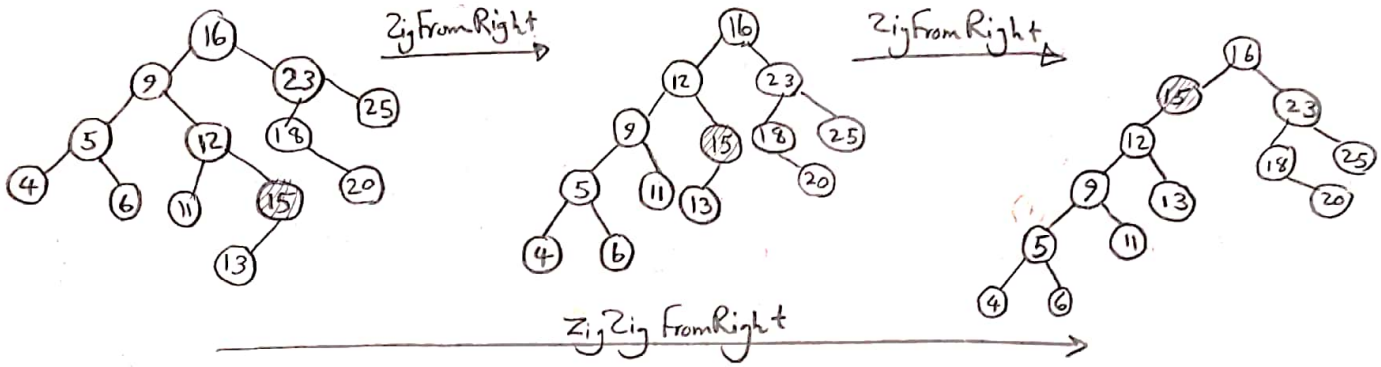


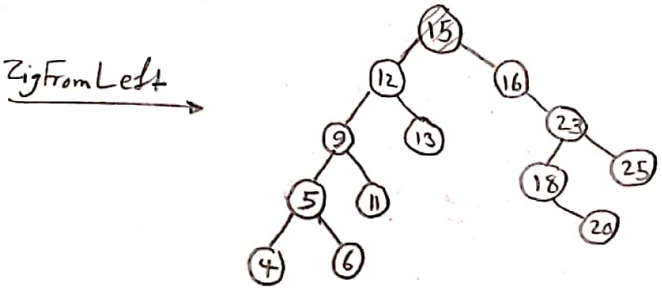
$$c'_{dec} = c_{dec} + \Delta\phi = c_{dec} + \phi_i - \phi_{i-1} =$$

$$= \underbrace{t_i + 1}_{\text{تعداد بیت‌هایی که flip می‌کنیم}} + \underbrace{1 - t_i}_{\text{تعداد 1-های اضافه شده (}t_i\text{ تا 1- و صفر شد و یک 0 و 1- شد)}} = 2 = \boxed{O(1)}$$

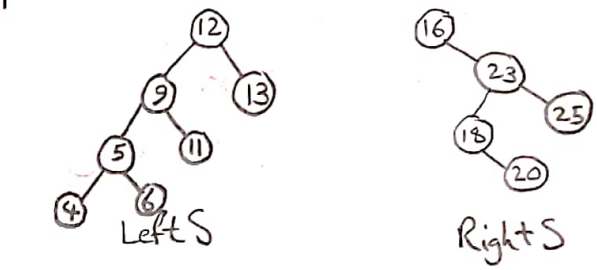
a

مرحله اول: splay (15, S):

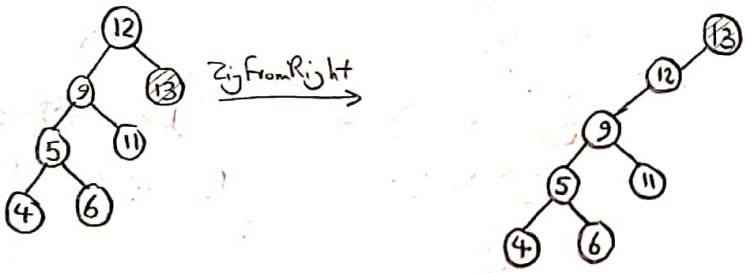




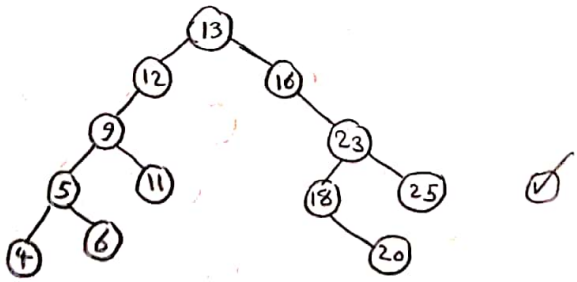
مرحله دوم: delete (15, S):



مرحله سوم: Splay (13, LeftS):

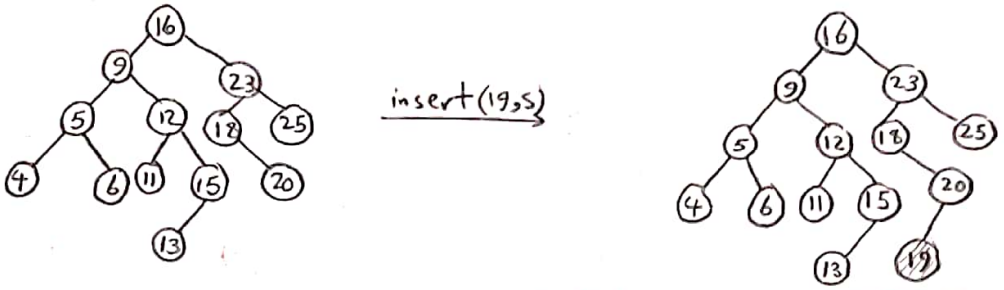


مرحله چهارم: Merge (LeftS, RightS):

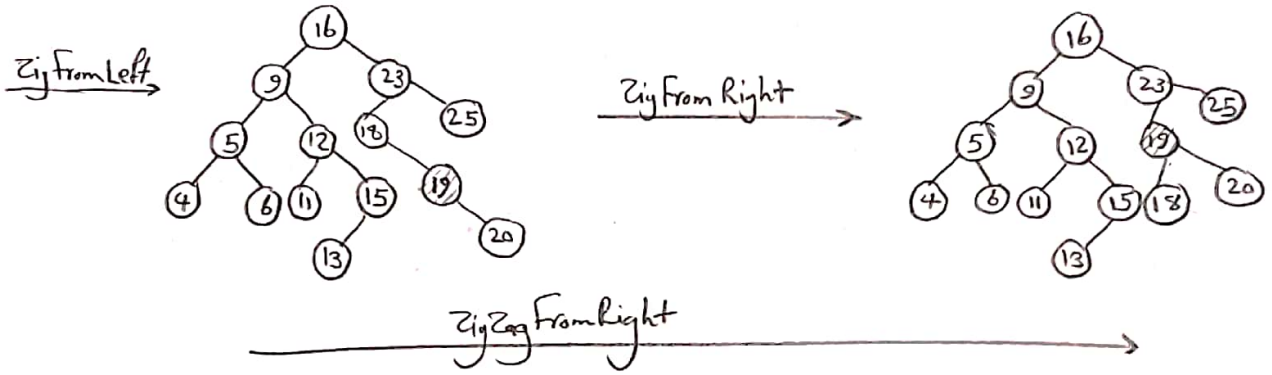


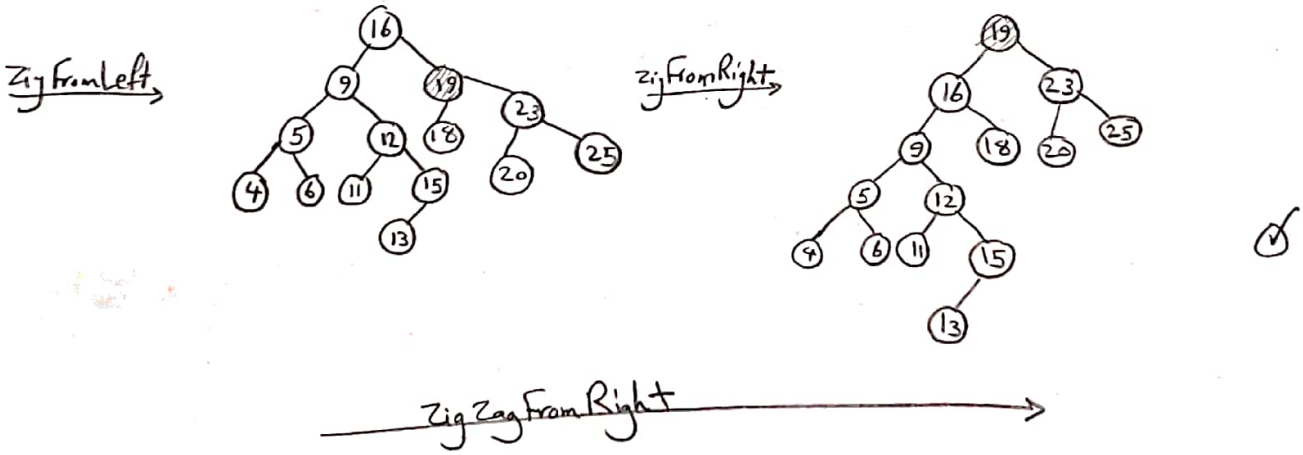
b

أولاً: insert (19,5) Like in BST:



ثانياً: Splay (19,5):

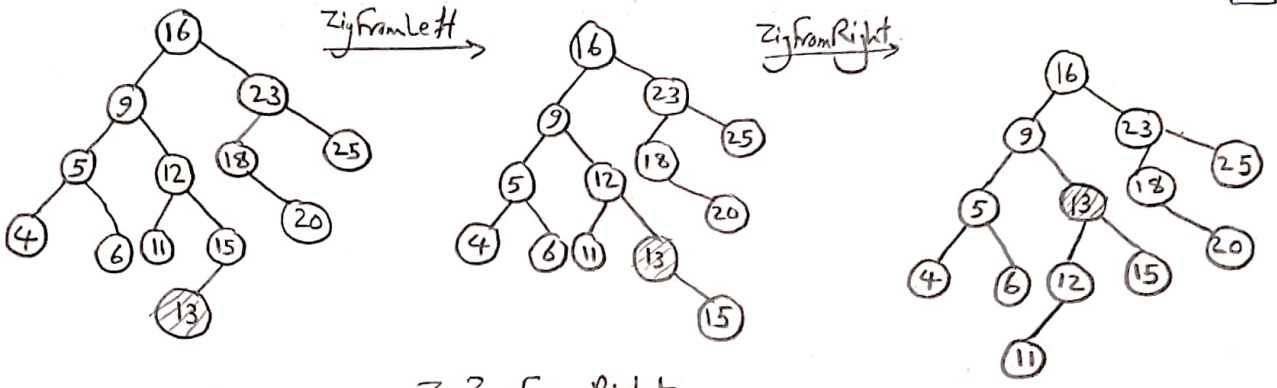


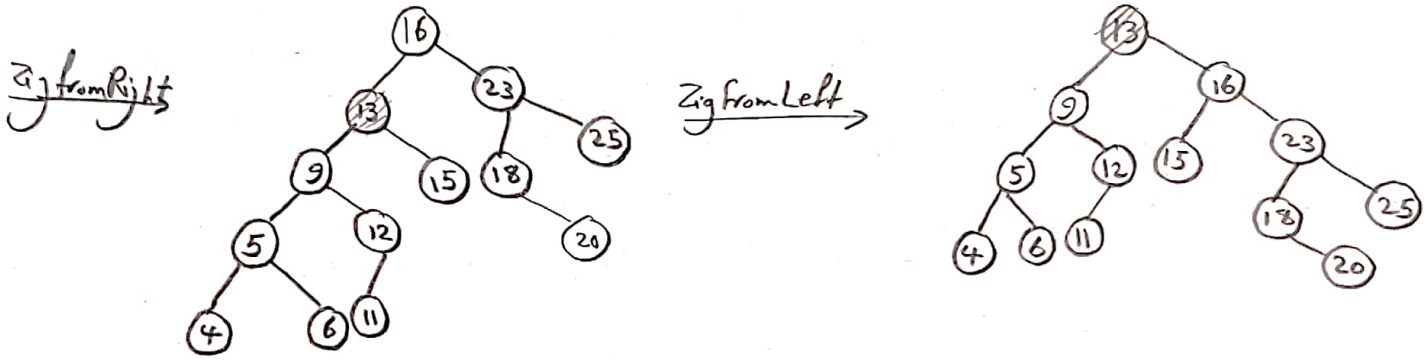


(7) مثال

C

الحل: Splay (13, S):





الحل: delete (13, S):