

## ۱- پیچیدگی الگوریتم

### ۱-۱ مقدمه

این قسمت شما را با قالب آنچه که در طول این کتاب در مورد طراحی و تحلیل الگوریتم استفاده خواهیم کرد، آشنا می‌کند.

ما با الگوریتم *insertion - sort* (مرتبسازی به روش درج) شروع می‌کنیم. برای نشان دادن چگونگی الگوریتممان از شبه کد استفاده می‌کنیم. بعد از اینکه الگوریتم را مشخص کردیم استدلال می‌کنیم که الگوریتم به درستی مرتب می‌کند و ما زمان اجرای آن را تحلیل می‌کنیم.

این تحلیل نشان می‌دهد که چگونه زمان اجرا متناسب با تعداد عناوینی که باید مرتب شوند، افزایش پیدا می‌کند. ما به دنبال توضیح *insertion-sort* رویکرد تقسیم و حل را برای طراحی الگوریتم‌ها معرفی می‌کنیم و از آن برای گسترش دادن یک الگوریتم که *Mergesort* است استفاده می‌کنیم و این قسمت را به پایان می‌رسانیم.

#### *Insertion-Sort*

ورودی: یک رشته از  $n$  عدد  $a_1, a_2, \dots, a_n$

خروجی: یک نظمی از  $\{a'_1, a'_2, \dots, a'_n\}$  از رشته ورودی به طوری که  $a'_1 < a'_2 < \dots < a'_n$ . اعدادی که ما می‌خواهیم مرتب کنیم به عنوان کلید (*key*) شناخته می‌شوند.

ما با مرتبسازی به روش درج شروع می‌کنیم که یک الگوریتم مفید برای مرتب کردن عناصر با تعداد کم است.

شبه کد ما مرتبسازی به روش درج، *Insertion-sort* نامیده می‌شود که یک آرایه  $A[1 \dots n]$  به عنوان پارامتر می‌گیرد.  $A$  یک رشته به طول  $n$  است که باید مرتب شود. در کد، تعداد  $n$  عنصر در  $A$  با  $Length[A]$  مشخص می‌شود. عدهای ورودی در محل مرتب می‌شوند. (Sorted in place) در واقع عدها درون آرایه  $A$  مجدداً چیده می‌شوند، با حداقل یک عدد از آنها که بیرون آرایه ذخیره می‌شود. (حداقل یک حافظه اضافی) وقتی که *insertion-sort* تمام می‌شود آرایه ورودی  $A$  دارای رشته خروجی مرتب شده است.

<b>INSERTION-SORT(<math>A</math>)</b>	<i>cost</i>	<i>times</i>
1 <b>for</b> $j \leftarrow 2$ <b>to</b> $\text{length}[A]$	$c_1$	$n$
2 <b>do</b> $\text{key} \leftarrow A[j]$	$c_2$	$n - 1$
3         ▷ Insert $A[j]$ into the sorted		
▷ sequence $A[1..j - 1]$ .	$0$	$n - 1$
4 $i \leftarrow j - 1$	$c_4$	$n - 1$
5 <b>while</b> $i > 0$ and $A[i] > \text{key}$	$c_5$	$\sum_{j=2}^n t_j$
6 <b>do</b> $A[i + 1] \leftarrow A[i]$	$c_6$	$\sum_{j=2}^n (t_j - 1)$
7 $i \leftarrow i - 1$	$c_7$	$\sum_{j=2}^n (t_j - 1)$
8 $A[i + 1] \leftarrow \text{key}$	$c_8$	$n - 1$

شکل ۱-۱

زمان اجرای الگوریتم :

: تابع هزینه

$$T(n) = \Sigma \text{زمان هر اجرا خط} * \text{تعداد تکرار هر خط}$$

$$\begin{aligned} T(n) = & c_1 n + c_2 (n - 1) + c_4 (n - 1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + \\ & + c_8 (n - 1) \sum_{j=2}^n (t_j - 1) c_7 \end{aligned}$$



شکل ۱-۲

= تعداد دفعاتی که حلقه  $i$  خط پنجم برای مقدار  $K$  اجرا می شود

تابع هزینه می تواند در ۳ حالت بررسی شود:

۱- بهترین حالت (احتمال کم) (آرایه از قبل مرتب باشد)

۲- بدترین حالت (احتمال کم)

### ۳-حالت متوسط

در بهترین حالت هزینه اجرای تابع *Insertion-sort* خطی است زیرا در بهترین حالت  $t_k = 1$  می باشد

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5(n-1) + C_8(n-1)$$

$$= (C_1 + C_2 + C_4 + C_5 + C_8)n - (C_2 + C_4 + C_5 + C_8)$$

عبارت فوق را می توان به فرم  $an+b$  نوشت . که در آن  $a, b$  ثابت هستند و به هزینه های ثابت  $C_i$

بستگی دارند . بنابراین  $T(n)$  یک تابع خطی از  $n$  می باشد .

در بدترین حالت یعنی آرایه به شکل برعکس مرتب شده باشد  $=kt_k$  است . پس تابع هزینه درجه ۲ می

شود.

$$\sum_{k=2}^n k = \frac{n(n+1)}{2} - 1$$

۹

$$\sum_{k=2}^n (k-1) = \frac{n(n-1)}{2}$$

$$T(n) = C_1 n + C_2(n-1) + C_4(n-1) + C_5\left(\frac{n(n+1)}{2} - 1\right) + C_6\left(\frac{n(n-1)}{2}\right) + C_7\left(\frac{n(n-1)}{2}\right) + C_8(n-1)$$

$$= \left(\frac{C_5}{2} + \frac{C_6}{2} + \frac{C_7}{2}\right)n^2 + (C_1 + C_2 + C_4 + \frac{C_5}{2} - \frac{C_6}{2} - \frac{C_7}{2} + C_8)n - (C_2 + C_4 + C_5 + C_8).$$

همان طور که پیداست زمانی که برای بدترین حالت مصرف می شود به فرم  $An^2 + Bn + C$  می باشد که

در آن ضرایب  $A, B, C$  ثابت هستند و به هزینه های ثابت  $C_i$  بستگی دارند .

احتمال در اینجا یکنواخت است یعنی عدد *Random* تولید شده ممکن است در هر جا قرار بگیرد و در

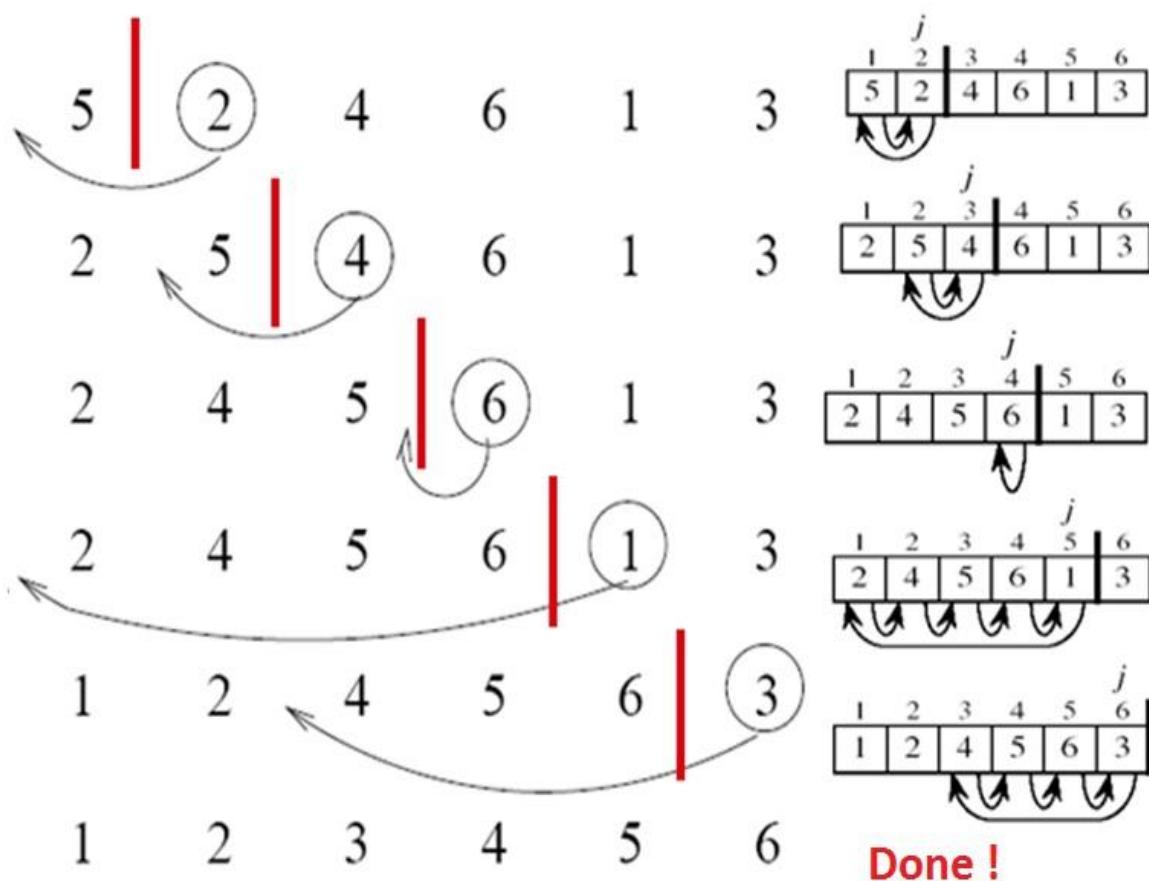
حال متوسط دقیقاً ممکن است در وسط قرار بگیرد یعنی فقط نصف اعداد قبلی را پیماش می‌کند.

$$p_i = \frac{k+1}{2} t_k = \frac{k+2}{2}$$

$$E(\textcolor{brown}{x}) = \sum_{x \in D} X.P(x)$$

$$E(t_k) = E_k = \sum_{t_{k-1}}^{k+1} t_k * (p_k)$$

در حالت متوسط هم هزینه اجرای این تابع درجه ۲ می شود.



شکل ۱-۳

## *Insertion – Sort*

1 – Best Case :  $An + B$

2 – Worst Case :  $A'n^2 + B'n + C$

3 – Average Case :  $A''n^2 + B''n + C$   $A'' < A'$

## *:Merge-sort*

### *:Merge*

ورودی: یک رشته از  $n$  عدد  $a_1, a_2, \dots, a_n$ . ابتدا و انتهای آرایه  $(p, r)$  و وسط آرایه  $q$  - به شکلی که اعداد

قبل و بعد  $q$  در رشته اعداد اولیه مرتب شده هستند.

خروجی: یک نظمی از  $\{a'_1, a'_2, \dots, a'_n\}$  از رشته ورودی به طوری که

MERGE( $A, p, q, r$ )

```

1   $n_1 = q - p + 1$ 
2   $n_2 = r - q$ 
3  let  $L[1..n_1 + 1]$  and  $R[1..n_2 + 1]$  be new arrays
4  for  $i = 1$  to  $n_1$ 
5     $L[i] = A[p + i - 1]$ 
6  for  $j = 1$  to  $n_2$ 
7     $R[j] = A[q + j]$ 
8   $L[n_1 + 1] = \infty$ 
9   $R[n_2 + 1] = \infty$ 
10  $i = 1$ 
11  $j = 1$ 
12 for  $k = p$  to  $r$ 
13   if  $L[i] \leq R[j]$ 
14      $A[k] = L[i]$ 
15      $i = i + 1$ 
16   else  $A[k] = R[j]$ 
17      $j = j + 1$ 

```

شکل ۱-۴

برای دریافتن آنچه در این الگوریتم رخ می دهد به شکل زیر توجه کنید. همان طور که پیداست اینتابع یک آرایه که از دو بخش مرتب شده تشکیل شده است را دریافت می کند و دو قسمت را با هم ادغام کرده و یک آرایه ی مرتب شده را بازمی گرداند. نحوه ی عملکرد به این شکل است که از ابتدای دو قسمت مرتب شده شروع میکند دو سر را با هم مقایسه میکند هر کدام که کوچک تر بود آن را در آرایه ی اصلی قرار داده و در آن قسمت یک خانه به جلو می رود. همین روند را تکرار می کند تا کل آرایه مرتب شود.

$A$	$\begin{array}{cccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ \dots &   & 2 & 4 & 5 & 7 & 1 & 2 & 3 & 6 & \dots \\ \hline k \end{array}$
$L$	$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\   & 2 & 4 & 5 & 7 & \infty \\ i & & & & & \end{array}$

(a)

$A$	$\begin{array}{cccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ \dots &   & 1 & 4 & 5 & 7 & 1 & 2 & 3 & 6 & \dots \\ \hline k \end{array}$
$R$	$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\   & 1 & 2 & 3 & 6 & \infty \\ j & & & & & \end{array}$

(b)

$A$	$\begin{array}{cccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ \dots &   & 1 & 2 & 5 & 7 & 1 & 2 & 3 & 6 & \dots \\ \hline k \end{array}$
$L$	$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\   & 2 & 4 & 5 & 7 & \infty \\ i & & & & & \end{array}$

(c)

$A$	$\begin{array}{cccccccccc} 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 17 \\ \dots &   & 1 & 2 & 2 & 7 & 1 & 2 & 3 & 6 & \dots \\ \hline k \end{array}$
$R$	$\begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \\   & 1 & 2 & 3 & 6 & \infty \\ j & & & & & \end{array}$

(d)

شکل ۱-۵

از تابع *merge* استفاده نموده و تابع *merge-sort* را ارایه می دهیم :

ورودی: یک رشته از  $n$  عدد  $a_1, a_2, \dots, a_n$  و ابتدا و انتهای آرایه  $(p, r)$

خروجی: یک نظمی از  $\{a'_1, a'_2, \dots, a'_n\}$  از رشته ورودی به طوری که  $a'_1 < a'_2 < \dots < a'_n$

MERGE-SORT( $A, p, r$ )

```

1  if  $p < r$ 
2     $q = \lfloor (p + r)/2 \rfloor$ 
3    MERGE-SORT( $A, p, q$ )
4    MERGE-SORT( $A, q + 1, r$ )
5    MERGE( $A, p, q, r$ )

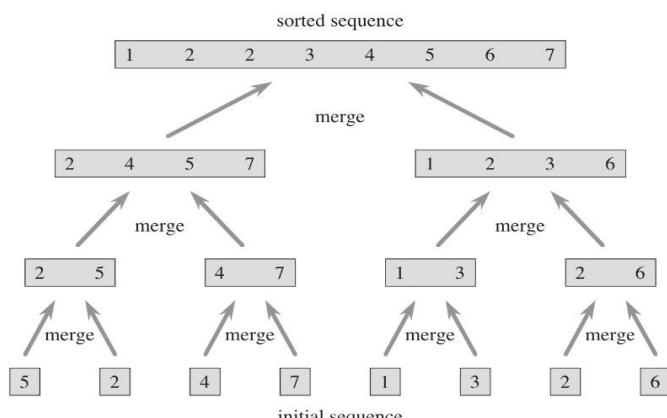
```

شکل ۱-۶

آن چه در این الگوریتم رخ می دهد را در شکل زیر مشاهده میکنیم (همان طور که پیداست نحوه ای عملکرد این تابع به شکل بازگشتی میباشد.)

رویکرد تقسیم و حل : در یک الگوریتم تقسیم و حل مساله به زیرمساله های کوچک تر تقسیم می شود. هر زیر مساله را به صورت بازگشتی حل می کنیم و سپس حل زیرمساله ها را برای حل مساله ای اصلی ترکیب می کنیم.

الگوریتم *merge-sort* نیز رویکرد تقسیم و حل را به کار گرفته است. در هر مرحله رشته ای اعداد را به دو زیررشته تقسیم کرده و سعی نموده تا مساله را برای دو زیررشته ای جدید حل کند.



شکل ۱-۷

هزینه‌ی تابع  $merge$  برابر با  $O(n)$  می‌باشد. اگر تابع هزینه‌ی  $T(n)$  را در نظر بگیریم داریم:

$$T(n) = 2T(n/2) + O(n) + O(1)$$

رابطه‌ی فوق برخاسته از رویکرد حل و تقسیم می‌باشد. به این معنا که هزینه‌ی مرتب کردن رشته‌ای از اعداد با طول  $n$  با این الگوریتم برابر با هزینه‌ی مرتب کردن دو رشته از اعداد با طول  $n/2$  به علاوه‌ی هزینه‌ی ادغام کردن این دو رشته است.

که جلوتر نشان خواهیم داد که حاصل فوق برابر با  $O(n \log n)$  است

## -۲- تحلیل الگوریتم

در این قسمت الگوریتم های متفاوت را بررسی می کنیم. برای هر الگوریتم  $S$  زمان اجرا در نظر می گیریم. در این زمان تعداد عناصری که می شود *Sort* کردها محاسبه می کنیم.

در بخش دیگر سرعت سخت افزار را  $10$  برابر می کنیم و نتایج متفاوتی را برای  $n$  بدست خواهیم آورد.

$O$	$T(n)$	$"n"$ that can be solved in $1000s$	For $10$ times faster machine	Ratio	<i>Ratio if was use a machine <math>1000</math> times faster <math>1000,00</math></i>
$O(n)$	$100n$	$10$	$100$	$10$	$1000,00$
$O(n^2)$	$5n^2$	$14$	$45$	$3,2$	$31,94$
$O(n^3)$	$\frac{n^3}{2}$	$12$	$27$	$2,3$	$10,50$
$O(2^n)$	$2^n$	$10$	$13$	$1,3$	$2,00$

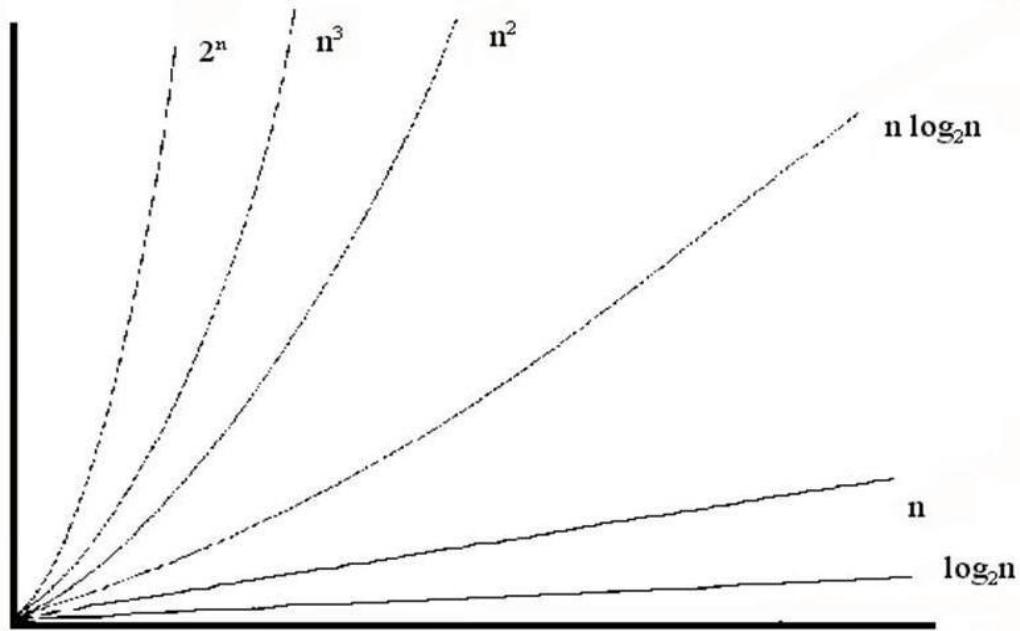
جدول ۱-۱

در بعضی از مواقع محاسبه  $T(n)$  عملی نیست.

در حالات کلی نیازی نیست که معادله  $T(n)$  تشکیل دهیم و ضرایب آن را بدانیم. این بدین معنی است که فقط دانستن درجه آن کافی است. هدف تنها تخمین و مقایسه الگوریتم ها در زمان اجرایشان است.

(همان طور که از جدول پیداست در الگوریتم هایی بالا  $O(n^2)$  و بالاتر با حتی با چند ده برابر کردن سرعت سخت افزار نسبت  $n$  در مقایسه با تغییر سرعت سخت افزار تغییر زیادی نمی کند).

### توابع هزینه متدال - ۳-۱



شکل ۱-۸

هر چه نمودار تابع هزینه یک الگوریتم به محور افقی نزدیک تر باشد یعنی سرعت رشد پایین تری داشته باشد الگوریتم مربوطه بهینه تر است.

<i>Function</i>	<i>Growth Rate Name</i>
$c$	<i>Constant</i>
$\log n$	<i>Logarithmic</i>
$\log_2 n$	<i>Log-Squared</i>
$N$	<i>Linear</i>
$n \log n$	
$n^2$	<i>Quadratic</i>
$n^3$	<i>Cubic</i>
$2^n$	<i>Exponential</i>

١-٢ جدول

## ۱.۴ توابع هزینه

گاهی اوقات لازم است که زمان نسبی انجام یک الگوریتم را بدانیم تا میزان سریع ( یا کند ) بودن آن را بدست بیاوریم . از انجا که ارتقا سخت افزار یک کامپیوتر به مرتبه پر هزینه تر از ارتقا نرم افزار آن است ما همواره تلاش میکنیم تا الگوریتم هایی با هزینه کمتر بسازیم تا با امکانات سخت افزاری سابق سریع تر به نتیجه برسیم.

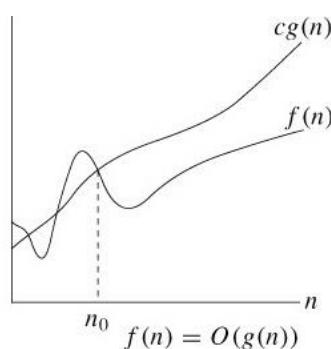
اما سوالی که مطرح میشود این است که زمان نسبی یک الگوریتم را چطور بدست بیاوریم ؟ برای این منظور تعاریف زیر را بیان میکنیم :

(order) O: تخمین بالای واقعیت ( سقف )

تعريف O : مجموع تمامی توابع از یک نقطه مشخص به بالا

$$f(n) = O(g(n)) \Leftrightarrow \exists c, n_0 > 0, \forall n > n_0, f(n) \leq cg(n)$$

g را سقف f میگوییم اگر بتوان c ای پیدا کرد تا تابع f به ازای همه n های بزرگ تر از n<sub>0</sub> کمتر مساوی cg(n) باشد.

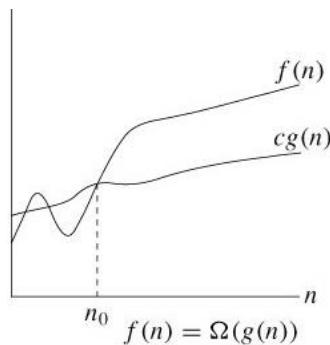


(omega) Ω: تخمین پایین واقعیت (کف)

تعريف Ω : مجموع تمامی توابع از یک نقطه مشخص به پایین

$$f(n) = \Omega(g(n)) \Leftrightarrow \exists c, n_0 > 0, \forall n > n_0, f(n) \geq cg(n)$$

g را کف f میگوییم اگر بتوان c ای پیدا کرد تا تابع f به ازای همه n های بزرگ تر از n<sub>0</sub> بزرگتر مساوی cg(n) باشد.

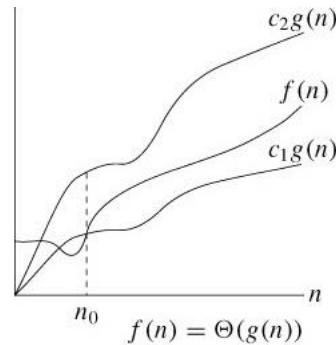


( $\Theta$ ) : تخمین دقیق (ضرایب را تعیین نکرده ایم ولی درجه معلوم است.)

تعریف  $\Theta$ : دقیق ترین تخمین است.

$$f(n) = \Theta(g(n)) \Leftrightarrow \exists c_1, c_2, n_0 > 0 \quad \forall n > n_0, c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$f(n) = O(g(n)) \quad \& \quad f(n) = \Omega(g(n))$$



o: اکیدا بالای واقعیت

$$f(n) = o(g(n)) \Leftrightarrow \exists c, n_0 > 0 \quad \forall n > n_0, f(n) < cg(n)$$

همان O است که حالت تساوی را ندارد

w: اکیدا پایین واقعیت

$$f(n) = \omega(g(n)) \Leftrightarrow \exists c, n_0 > 0 \quad \forall n > n_0, f(n) > cg(n)$$

همان  $\Omega$  است که حالت تساوی را ندارد

خواص تخمین ها:

Reflexivity : (بازتابی) O,  $\Omega$ ,  $\Theta$

Summetiy: ( $\Theta$  تقارنی)

Transitivity: (تعدی) همه ی تخمین ها

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

$$O(f(n)) + O(g(n)) = O(\max(g(n), f(n)))$$

If  $f(n) = O(kg(n))$  for Constant  $k > 0 \Rightarrow f(n) = O(g(n))$

## ۱.۵ آنالیز زمان اجرا

برای آنالیز هر خط و زمان اجرای آن را در نظر می‌گیریم.

Simple Sentences (read,write,...) :  $O(1)$

Simple Rules(+,=,\*,,==,...) :  $O(1)$

Loops(for,while,...) :  $O(1)$

Conditional Sentences :  $O(\text{Max}(T(\text{body}^1), T(\text{body}^2)))$

معمولًا آنالیز زمان اجرای Conditional Sentences پیچیده است و ما قادر به محاسبه آن نمی‌باشیم.

گلوگاه کد قسمتی از کد است که بیشترین زمان اجرای را در بر می‌گیرد.

چند مثال:

۱)

```
for i = 1 to n
```

```
    for j = i to n
```

```
        sum++;
```

$$T(n) = \sum_{i=1}^n \sum_{j=i}^n O(1) = \sum_{i=1}^n (n - i + 1) = \sum_{i=1}^n n - \sum_{i=1}^n i + \sum_{i=1}^n 1 = n^2 - \frac{n(n+1)}{2} + n$$

$$T(n) = \Theta(n^2)$$

۲)

```
for (i = 1 ; i < n ; i *= 2)
```

```
    S++;
```

$$T(n) = \Theta(\log n)$$

۳)

```
for (i = 1 ; i < n ; i++)
```

```
    for(j = 1 ; j < n ; j+=j)
```

```

Sum++;

T(n) = Σi=1n-1 log n = n log n

T(n) = Θ(n log n)

ε)

for (i = 1 ; i < n ; i+=i)

    for(j = 1 ; j < i ; j ++)

        s++;

T(n) = Σk=1log n Θ(2k) = 2log n - 1 = n - 1

T(n) = Θ(n)

```

```

o)

Int i=n;

While(i>1){

    i/=2;

    j=n;

    while(j > 1)

        j/=2;

}

T(n) = Σk=1log n log 2k = log 21 * 21 * 22 * .... * 2log n = log 21 log(log n + 1)

T(n) = Θ(log n)

```

## یک مثال ۱,۵,۱

فرض کنید ارایه ای از اعداد صحیح داشته باشیم می خواهیم بزرگ ترین زیر دنباله از این ارایه را به دست بیاوریم که مجموع اعضای آن بیشینه باشد.

زیردنباله : برای ارایه  $a$  به ازای تمام  $i$  هایی که  $i \leq k$  اعضای  $[i] \leq k$  عضو زیردنباله  $s$  از  $k$  تا  $n$  می باشد.

۴ الگوریتم متفاوت را بررسی می کنیم.

الگوریتم اول:

بزرگ ترین زیردنباله ای که از ان شروع می شود را (a) اولین راهی که به ذهن می رسد این است که به ازای هر عضو ارایه بدست اوریم و بیشینه آنها جواب مساله است

```
int Maxsum=0;  
  
for (int i=0 ; i< a.size();i++) O(Σj=in-1 (j - i))  
  
    for(int j=i; j < a.size() ; j++) O(Σj=in-1 (j - i))  
  
        int ThisSum=0; O(1)  
  
        for(int k=i; k<= j ; k++) O(j-i)  
  
            ThisSum+=a[k]; O(1)  
  
        if(ThisSum > MaxSum) O(1)  
  
            MaxSum=ThisSum; O(1)  
  
    Return MaxSum; O(1)  
  
T(n) = O(n2)
```

الگوریتم دوم: در این الگوریتم ۱ حلقه کم شده است.

در واقع همان الگوریتم اول است با این تفاوت که به ازای هر  $[i]$  دیگر از اول تا آخر بازه را محاسبه نمی کنیم بلکه با مقدار قبلی مقایسه و جمع می کنیم.

```
int MaxSum = 0 ;  
  
for (int i=0 ; i< a.size() ; i++){  
  
    int ThisSum=0;  
  
    for(int j=i; j <= a.size() ; j++){  
  
        ThisSum += a[j];
```

```

If(ThisSum > MAsSum)

    MaxSum = ThisSum;

}

}

Return MaxSum;

T(n) = O(n^r)

```

### الگوریتم سوم: Divide & Conquer ( تقسیم و غلبه )

در این قسمت مساله را به چند زیر مساله تقسیم می کنیم و به صورت بازگشتی زیرمساله ها را حل می کنیم تا به شرط خاتمه یا حالتی بدیهی بررسیم . این روش یکی از کاربردی ترین راه حل های حل مساله است ( یاد گیری ایده ان به شدت توصیه میشود )

برای این منظور یکتابع تعریف می کنیم که یک ارایه میگیرد و بزرگ ترین مجموع را برمی گرداند حال وسط ارایه را به دست می اوریم . جواب نیمه اول را  $S_1$  و جواب نیمه دوم را  $S_2$  در نظر میگیریم . حال تنها حالاتی باقی می ماند که ابتدا ان در نیمه اول و انتهای ان در نیمه دوم است که این کار با  $O(n)$  امکان پذیر است.

```

int MaxSubSeq(int A[], int f , int t){

    if(f==t){
        if(A[f] < 0 )
            return 0;
        return A[f];
    }

    M = (f+t)/2;

    S1=MaxSubSeq(A,f,m)           T(n/2)
    S2=MaxSubSeq(A,m+1,t)         T(n/2)
    S3=temp=0;

    for (i=m;i>=f;i--)
        temp += A[i];
}

```

```

if(temp > Sr)
Sr=temp;
}
Sl = temp=·;
for (i=m+1;i<=t;i++){
    temp += A[i];
    if(temp>Sl)
        Sl = temp;
}

```

۳ و ۴ قسمت غلبه این الگوریتم را نشان می دهد

$$T(n) = T(n/2) + T(n/2) + O(n)$$

$$T(n) = n \log n \log n$$

الگوریتم چهارم:

در این الگوریتم چند نکته حائز اهمیت است.

۱- جواب با عدد منفی شروع نمی شود.

۲- جواب با اعداد منفی پایان نمی یابد.

۳- جواب با پیشوند منفی شروع نمی شود

پس به این صورت عمل می کنیم که از سر ارایه شروع می کنیم و مجموع اعضا را بدست می اوریم هر جا مجموع منفی شد اعداد دیده شده را دور ریخته از نو شروع می کنیم

```

int MaxSum=·;
int ThisSum=·;
for(int j = · ; j < a.size() ; j++){
    ThisSum += a[j];
}

```

```

If(ThisSum>MaxSum)

    MaxSum=ThisSum;

else if(ThisSum< · )

    ThisSum=·;

}

Return MAxSum;

```

چون در کل هر عضو ارایه را یک بار دیدیم پس  $O(n)$  است

در الگوریتم اول زمان پاسخگویی به اندازه طول عمر ما ادامه پیدا خواهد کرد اما الگوریتم اول در کمتر از ۱ ثانیه  $n = 10^7$  برای جواب خواهد داد.

## ۱- الگوریتم های بازگشتی

### ۱-۱- مقدمه:

تابع بازگشتی توابعی هستند که داخل تعریف شان دوباره از خود تابع استفاده می شود. این توابع به علت اینکه خوانایی بیشتری نسبت به توابع دیگر دارند و کد کمتری برای نوشتن آن ها لازم است در برنامه نویسی طرفداران زیادی دارند.

تابع بازگشتی مشهور عبارت اند از: تابع فاکتوریل، تابع فیبونانچی، توان، ضرب و...

### ۲-۱- مفهوم بازگشتی:

حل کردن مسائل از طریق بازگشتی معمولاً از چند بخش تشکیل شده است. وقتی که یک تابع بازگشتی برای حل یک مسئله خوانده می شود یا برای حل بخش پایه فراخوانده شده است یا توانایی حل قسمتی از مسئله را دارد و برای حل قسمت دیگر باید خودش را صدا کند. مثلا برای محاسبه  $i!$  به روش بازگشتی از تعریف می دانیم  $i! = i \times (i-1)!$ . آن را به دو بخش  $i \times (i-1)!$  تقسیم می کنیم. برای  $i=1$  داریم  $1! = 1$  و دوباره آن را به دو بخش  $(1 \times 2)$  و  $2!$  تقسیم می کنیم.  $1!$  حالت پایه است که آن را می شناسیم. پس نتیجه این شد که اگر تابع با بخش پایه صدا شد معمولاً نتیجه ای را باز می گرداند و اگر با بخش پیچیده تری صداشد، معمولاً جواب به طور نظری به دو بخش تقسیم می شود. یک بخش که تابع می داند چه طور آن را حل کند ( $i \times (i-1)!$ ) و یک بخش که نمی داند ( $i \times (i-1)!$  در مثال بالا) برای این که بتوانیم تابع را به صورت بازگشتی تعریف کنیم بخواهی که حل آن را نمی دانیم باید قابل تبدیل به مسئله ای ساده تر یا کوچکتر از آن باشد. به خاطر این که این مسئله ای جدید شبیه به مسئله ای اولیه است تابع بازگشتی، تابعی تازه از خودش را صدا می زند تا مسئله ای کوچک تر را حل کند. به این مرحله ((فراخوانی بازگشتی)) یا ((گام بازگشتی)) می گوییم.

این طرز فکر که مسئله را به دو بخش کوچک تر تبدیل و آن ها را حل کنیم روش بازگشتی را جزو روش های تقسیم و غلبه قرار می دهد.

### أنواع بازگشت:

مفهوم بازگشتی در ریاضیات کاربرد دارد که نمونه های زیر مثال هایی از این نوع اند:

### ۱- دنباله های بازگشتی:

$$\begin{cases} S(n) = 2S(n-1) \\ S(0) = 1 \end{cases}$$

که این دنباله توان های ۲ را به ما می دهد:

اعداد دنباله ی توانهای ۲

$$n=0 \quad n=1 \quad n=2 \quad n=3 \quad n=4 \dots \dots \dots \quad n=m$$

$$S=1 \quad S=2 \quad S=4 \quad S=8 \quad S=16 \dots \dots \dots \quad S=2^m$$

$$\begin{cases} S(n) = 3S(n-1) + 2 \\ S(0) = 2 \end{cases}$$

که این دنباله اعدادی را به ما می دهد که در تقسیم به ۳ با قیمانده ی یکسانی دارند.(به پیمانه ی ۳ همنهشتند).

اعداد همنهشت به پیمانه ی ۳

$$n=0 \quad n=1 \quad n=2 \quad n=3 \quad n=4 \dots \dots \dots \quad n=m$$

$$S=2 \quad S=8 \quad S=26 \quad S=80 \quad S=242 \dots \dots \dots \quad S=3^{m+1}-1$$

## ۲- مجموعه های بازگشتی:

$$\left\{ \begin{array}{l} \text{پدر و مادر قلی جزو اجداد او هستند} \\ \text{هر پدر و مادر جد مجموعه‌ی اجداد قلی هستند} \\ \text{این مجموعه شامل تمام اجداد قلی است} \end{array} \right.$$

## ۳- عملگرهای بازگشتی:

$$\left\{ \begin{array}{l} M(1) = m \\ M(n) = M(n-1) + m \end{array} \right.$$

این عملگر ضرب را با استفاده از تعریف ضرب ( $n \times m = (n-1) \times m + m$ ) و عملگر جمع می‌سازد.

$$\left\{ \begin{array}{l} M(1) = m \\ M(n) = M(n-1) * m \end{array} \right.$$

این عملگر توان را با استفاده از تعریف توان ( $m^n = m * m^{n-1}$ ) و عملگر ضرب می‌سازد.

## ۴- الگوریتم های بازگشتی:

در این فصل به الگوریتم‌های بازگشتی و تحلیل آنها می‌پردازیم. بسیاری از مسائل را می‌توان به کمک الگوریتم‌های بازگشتی حل کرد. هر الگوریتم بازگشتی معمولاً از ۳ مرحله تشکیل شده است:

۱. تقسیم کردن مسئله اصلی به زیر مسئله‌های کوچک‌تر از همان مسئله.

۲. حل کردن زیر مسئله‌های کوچک‌تر به صورت بازگشتی.

۳. ترکیب کردن جواب‌های به دست آمده از حل زیر مسئله‌های کوچک‌تر برای به دست آوردن جواب مسئله اصلی.

به ۳ مرحله بالا که در کنار هم به حل مسئله‌های بازگشتی منجر می‌شوند، تقسیم و غلبه یا تقسیم و حل (Divide and conquer) می‌گویند.

به عنوان مثال فرض کنید داریم:

$$\begin{cases} F(0) = F(1) = 1 \\ F(n) = F(n-1) + F(n-2) \end{cases}$$

در اینجا حالت پایه می‌شود ( $F(0)$  و  $F(1)$ ) که جواب را می‌دانیم.

$F(n-2)$  و  $F(n-1)$  بخش‌هایی از الگوریتم هستند که حل آن‌ها را نمی‌دانیم.

بخشی که حل آن را می‌دانیم این است که با استفاده از جمع ( $F(n-2)$  و  $F(n-1)$  می‌توانیم  $F(n)$  را بسازیم.

در کل می‌توان ۲ مرحله برای این الگوریتم‌ها در نظر گرفت:

۱) هنگامی که زیر مسئله‌ها به اندازه‌ای کوچک شده باشند که جواب آنها بدیهی است و نیازی به بازگشت نیست به این مرحله، مرحله پایانی یا حالت بدیهی می‌گوییم. که معمولاً در ابتدای یک الگوریتم بازگشتهایی به عنوان شرط خاتمه چک می‌شوند.

۲) هنگامی که زیر مسئله‌ها به اندازه کافی بزرگ باشند که به صورت بازگشتهای حل شوند، به این مرحله، مرحله بازگشت (Recursion Step) می‌گوییم. این مرحله باید به گونه‌ای باشد که زیر مسئله را به سمت شرط خاتمه پیش ببرد.

```
int Fact (int n) {  
    if (n == 0)  
        return 1; // شرط خاتمه  
    else  
        return n*Fact (n-1); // مرحله بازگشتی  
}
```

تحلیل الگوریتم‌های بازگشتهای برای به دست آوردن زمان اجرا یا حافظه مصرفی آنها معمولاً به یک رابطه بازگشتهای منجر می‌شود. رابطه بازگشتهایی، یک معادله یا نا معادله است که یک تابع را بر حسب مقدارهای خود آن تابع به ازای ورودی‌های کوچکتر بیان می‌کند. مثلاً بدترین زمان اجرای الگوریتم مرتب سازی ادغامی را به کمک رابطه بازگشتهای زیر بیان می‌کنیم:

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 1 \\ 2T\left(\frac{n}{2}\right) + \theta(n), & \text{if } n > 1 \end{cases}$$

که در نهایت به  $T(n) = \theta(n \lg n)$  منجر خواهد شد.

**نکته** : مسائلی که به صورت روابط ریاضی و محا سباتی هستند یا میتوان آنها را به زیر مسئله هایی کوچکتر از همان مسئله تبدیل کرد را میتوان با استفاده از روش بازگشتی حل کرد.

(۱) مثال

```
int sum (int n) {
    if (n == 0)
        return 0;
    return sum(n-1) + n;
}
```

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 0 \\ T(n-1) + \theta(1), & \text{if } n > 0 \end{cases}$$

(۲) مثال

```
void print_rev() {
    char ch;
    if( (ch = getchar()) != '\n' ) {
        print_rev();
        print(ch);
    }
}
```

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 0 \\ T(n-1) + \theta(1), & \text{if } n > 0 \end{cases}$$

این شبه کد یک رشته را بعنوان ورودی میگیرد و آن را برعکس چاپ میکند

در این فصل ۵ روش را برای حل رابطه‌های بازگشتی معرفی خواهیم کرد که معمولاً برای به دست آوردن مرتبه تابع‌های مورد نظر کافی هستند:

۱. حدس جواب و اثبات با استقراء

۲. تکرار و جایگزینی

۳. درخت بازگشت

۴. قضیه اصلی

۵. روش‌های خلاقانه‌ی دیگر

در عمل، برای حل روابط بازگشتی از بدخی جزئیات چشم پوشی می‌کنیم:

۱. معمولاً از سقف و کف در روابط صرف نظر می‌کنیم. این‌ها در اکثر موارد در جواب تاثیری ندارند، پس ابتدا رابطه را بدون در نظر گرفتن آنها حل می‌کنیم و بعداً تعیین می‌کنیم که آیا در جواب تغییری ایجاد کرده‌اند یا خیر. برای این کار تجربه و تعدادی قضیه به ما کمک می‌کنند. مثلاً هنگام تحلیل زمان اجرای مرتب سازی ادغامی در واقع به رابطه  $T(n) = T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \theta(n)$  میرسیم اما ما با صرف نظر کردن از سقف و کف به رابطه  $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$  میرسیم.

۲. نمونه دیگری از جزئیاتی که از آنها چشم پوشی می‌کنیم، شرایط مرزی هستند. از آنجا که زمان اجرای یک الگوریتم برای ورودی‌های ثابت، ثابت است میتوانیم فرض کنیم که برای  $n$ ‌های به اندازه کافی  $T(n) = \theta(1)$ . مثلاً رابطه الگوریتم مرتب سازی ادغامی را به صورت  $T(n) = 2T\left(\frac{n}{2}\right) + \theta(n)$  بیان می‌کنیم بدون این که مقدار دقیق  $T(n)$  را برای  $n$ ‌های کوچک به صراحت مشخص کنیم. این  $\theta(n)$

کار به آن دلیل است که اگر چه تغییر در مقدار  $T(1)$  حل دقیق مسئله را تغییر میدهد، اما جواب نهایی حداکثر به اندازه یک عامل ثابت تغییر می‌کند و در نتیجه مرتبه رشد آن بدون تغییر خواهد ماند.

۳. گاهی هم برای راحتی کار از نماد گذاری‌های مجانبی همچون  $O$  و  $\Theta$  در رابطه‌ها صرف نظر می‌کنیم. مثلاً به جای  $(n\Theta)$  می‌گذاریم و به حل رابطه بازگشتی می‌پردازیم. در عمل این چشم پوشی هم مرتبه رشد تابع را تغییر نخواهد داد و حل رابطه را آسان‌تر خواهد کرد.

## ۱-۴ حدس و استقراء

این روش از ۲ مرحله تشکیل شده:

۱. حدس جواب (به صورت دقیق یا با نماد گذاری مجانبی)

۲. استفاده از استقرای ریاضی برای اثبات این که جواب درست است.

این روش روشی قدرتمند است، اما به وضوح باید بتوانیم شکل جواب را حدس بزنیم تا از این روش استفاده کنیم. از روش حدس و استقراء میتوان برای پیدا کردن حدود بالا و پایین برای جواب بازگشتی‌ها استفاده کرد. معمولاً برای اثبات حدس خود از تعاریف نماد گذاری‌های مجانبی  $(O, \Theta, \Omega)$  استفاده می‌کنیم. برای آشنا شدن با این روش، مثال زیر را دنبال کنید.

مثال ۱) رابطه بازگشتی زیر را حل کنید. (تحلیل مرتب سازی ادغامی)

$$T(n) = \begin{cases} c_1, & n = 2 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c_2 n, & n > 2 \end{cases}$$

حل: با دقت در رابطه متوجه می‌شویم که مسئله در هر مرحله به ۲ زیر مسئله به اندازه نصف مسئله تقسیم می‌شود. پس بعد از حدود  $\log(n)$  بار به زیرمسئله‌های با اندازه ۱ خواهیم رسید. با توجه به این که زیرمسئله‌ها را با ضریبی از  $n$  با هم ترکیب می‌کنیم، می‌توان حدس زد که جواب  $(n \lg n)\Theta$  است. حال با استفاده از استقراء به اثبات ادعای خود می‌پردازیم. با توجه به تعریف  $\Theta$  کافیست ثابت کنیم برای مقادیر بزرگ  $n$  عددی

.  $T(n) \geq a * n \lg n$  و همچنین عددی مانند  $b$  وجود دارد که  $T(n) \leq a * n \lg n + b * n \lg n$  مانند  $0 > a$  وجود دارد

برای قسمت اول داریم:

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + c_2 n \leq 2T\left(\frac{n}{2}\right) + c_2 n$$

$$n = 2 : T(2) \leq 2a \Rightarrow a \geq \frac{c_1}{2} > 0 : \text{پایه استقرای}$$

$$\forall k < n, k \geq 2 : T(k) \leq ck * \lg k : \text{فرض استقرای}$$

گام استقرای: باید اثبات کنیم  $T(n) \leq a * n \lg n$

$$T(n) \leq 2a\left(\frac{n}{2}\right)\lg\left(\frac{n}{2}\right) + c_2 n = an * \lg\left(\frac{n}{2}\right) + c_2 n = an * \lg n - an + c_2 n$$

که کافیست  $a$  را بزرگ‌تر از  $c_2$  انتخاب کنیم تا حکم ثابت شود.

اثبات قسمت دوم نیز مشابه قسمت اول است و از آنجا  $b$  به دست می‌آید. در واقع ثابت کردیم که :

$$T(n) = O(n \lg n) \& T(n) = \Omega(n \lg n) \Rightarrow T(n) = \Theta(n \lg n)$$

مثال ۲) مرتبه رابطه بازگشتی زیر را به دست آورید.

$$T(n) = T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + 1$$

حل: به نظر می‌رسد جواب  $O(n)$  است. پس باید اثبات کنیم  $T(n) \leq cn$ . اگر همانند روش مثال قبل از استقرای استفاده کنیم، خواهیم داشت:

$$T(n) \leq c\lceil n/2 \rceil + c\lfloor n/2 \rfloor + 1 = cn + 1 > cn$$

که اشتباه است. اما مقداری که در سمت چپ اضافه آمده، یک مقدار ثابت یعنی عدد ۱ است و به  $n$  بستگی ندارد. در این موارد یک راهکار وجود دارد، این که فرض استقرای را تغییر دهیم و یک مقدار ثابت به آن اضافه کنیم. یعنی فرض کنیم:

$$T(n) \leq cn + b$$

در این صورت با استقرار خواهیم داشت:

$$T(n) \leq 2c\left(\frac{n}{2}\right) + 2b + 1 = cn + 2b + 1 \leq cn + b$$

که برای این که نامساوی آخر برقرار باشد کافیست  $b$  را کوچک تر یا مساوی ۱- انتخاب کنیم. پس:

$$T(n) = O(n)$$

#### ۱-۱-۴ تغییر متغیر

گاهی اوقات رابطه بازگشته پیچیده تر است و حدس جواب سخت به نظر می‌رسد. در این موارد گاهی تغییر متغیر به کمک ما می‌اید. مثال زیر موضوع را روشن تر می‌کند.

$$\text{مثال ۱)} T(n) = 2T(\sqrt{n}) + lgn$$

حل: با تغییر متغیر زیر رابطه را حل می‌کنیم:

$$m = lgn \Rightarrow T(2^m) = 2T(2^{m/2}) + m$$

با فرض  $S(m) = T(2^m)$  خواهیم داشت:

$$S(m) = 2S\left(\frac{m}{2}\right) + m$$

که با توجه به مثال‌های قبل داریم

$$S(m) = O(mlgm) \Rightarrow T(n) = O(lgnlglgn)$$

$$\text{مثال ۲)} T(n) = \sqrt{n}T(\sqrt{n}) + n lgn$$

حل: ابتدا طرفین رابطه را بر  $n$  تقسیم می‌کنیم:

$$\frac{T(n)}{n} = \frac{T(\sqrt{n})}{\sqrt{n}} + lgn$$

با تغییر متغیر زیر رابطه را حل می‌کنیم:

$$S(n) = \frac{T(n)}{n} \quad \Rightarrow \quad S(n) = S(\sqrt{n}) + lgn$$

با فرض  $H(m) = S(2^m)$  و  $n = 2^m$  خواهیم داشت:

$$H(m) = H\left(\frac{m}{2}\right) + m$$

با توجه به حدس واستقرا  $H(m) = O(m)$

از آنجا که:

$$m = \log n \quad \text{و} \quad \frac{T(n)}{n} = S(n) = S(2^m) = H(m) = O(m)$$

داریم:

$$T(n) = O(nm) \quad \Rightarrow \quad T(n) = O(n \lg n)$$

## ۲-۴ تکرار و جایگزینی

در این روش در رابطه بازگشتی برای  $n$  که بر حسب  $n$  بیان شده،  $n_1, n_2, n_3, \dots, n_k$  با استفاده از خود رابطه بازگشتی جایگزین می‌کنیم و بسط میدهیم. این روند را آنقدر تکرار می‌کنیم تا به جواب نهایی برسیم. از این روش می‌توان برای به دست آوردن جواب دقیق رابطه‌های بازگشتی استفاده کرد. برای این که با این روش آشنا شویم، مثال زیر را دنبال کنید:

مثال ۱) رابطه بازگشتی  $T(0) = 1$ .  $T(n) = 2T(n-1) + 1$  را حل کنید.

حل :

$$\begin{aligned} T(n) &= 2T(n-1) + 1 = 2[2T(n-2) + 1] + 1 = 2^2T(n-2) + 2 + 1 \\ &= 2^2[2T(n-3) + 1] + 2 + 1 = 2^3T(n-3) + 2^2 + 2^1 + 2^0 = \dots \\ &= 2^nT(0) + 2^{n-1} + 2^{n-2} + \dots + 2^1 + 2^0 = 2^{n+1} - 1 \quad (T(0) = 1) \end{aligned}$$

در نتیجه  $T(n) = \theta(2^n)$

مثال ۲) رابطه بازگشتی  $T(1) = 1$ .  $T(n) = 2T\left(\frac{n}{2}\right) + n$  را حل کنید.

: حل

$$\begin{aligned}
 T(n) &= 2T\left(\frac{n}{2}\right) + n = 2 \left(2T\left(\frac{n}{4}\right) + \frac{n}{2}\right) + n = 2^2T\left(\frac{n}{2^2}\right) + 2n = \dots \\
 &= 2^i T\left(\frac{n}{2^i}\right) + in = \dots = 2^{\lg n} T(1) + n \lg n = n + n \lg n = \theta(n \lg n)
 \end{aligned}$$

### ۳-۴ درخت بازگشت

درخت بازگشت روشی است که اکثرا برای حل و مخصوصاً حدس جواب روابط بازگشتی، بسیار به کار می آید. در این درخت هر گره نشان دهنده هزینه یک زیر مسئله در جایی از مجموعه فراخوانی‌های تابع است. مجموع اعداد موجود در هر سطح از درخت برابر هزینه کلی آن سطح است و برای به دست آوردن هزینه کل، هزینه سطرها را با هم جمع می‌کنیم. جوابی که از این جمع به دست می‌آید یا دقیق است، یا با تقریب خوبی می‌توان از آن برای حدس در روش حدس و استقرا استفاده کرد. از آنجایی که معمولاً جوابی را که توسط درخت بازگشت به دست می‌آوریم (که یک حدس است) بعداً توسط روش‌های دیگر تایید می‌کنیم، هنگام پر کردن اعداد هر سطح و هنگام ساختن درخت میتوانیم کمی بی دقتی را هم تحمل کنیم.

مثال ۱) فرض کنید نوع جدیدی از الگوریتم مرتب سازی ادغامی را برای مرتب کردن یک آرایه  $n$  تایی از اعداد به کار می‌بریم. تنها تفاوت روش جدید با روش قبلی این است که هنگام ۲ قسمت کردن آرایه و تقسیم آن به ۲ زیر مسئله کوچکتر، به جای آن که آرایه را به ۲ آرایه مساوی با اندازه  $n/2$  تقسیم کنیم به ۲ آرایه با اندازه‌های  $n/3$  و  $2n/3$  تقسیم می‌کنیم، پس از حل مسئله برای هر یک از آرایه‌های جدید، آنها را با هم ادغام می‌کنیم. مرتبه الگوریتم جدید را به دست آورید.

حل : رابطه بازگشتی که این الگوریتم به دست می‌دهد، به صورت زیر است:

$$T(n) = \begin{cases} 1, & n = 1 \\ T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n, & n > 1 \end{cases}$$

دقت کنید که برای به دست آوردن بازگشتی بالا، از نشان گذاری‌های مجانبی و سقف و کف صرف نظر کردیم.

همانطور که در شکل زیر می‌بینید، در اولین سطح، محاسبه  $T(n)$  منجر به محاسبه  $T\left(\frac{2n}{3}\right)$  و  $T\left(\frac{n}{3}\right)$  و تولید مقدار ثابت  $n$  می‌شود. در سطح دوم نیز محاسبه  $T\left(\frac{2n}{3}\right)$  و  $T\left(\frac{n}{3}\right)$  به ترتیب منجر به محاسبه  $T\left(\frac{n}{3^2}\right)$  و  $T\left(\frac{2^2n}{3^2}\right)$  و  $T\left(\frac{2n}{3^2}\right)$  و همچنین تولید ثابت‌های  $n/3$  و  $2n/3$  می‌شوند که مجموع آنها برابر با  $n$  است.

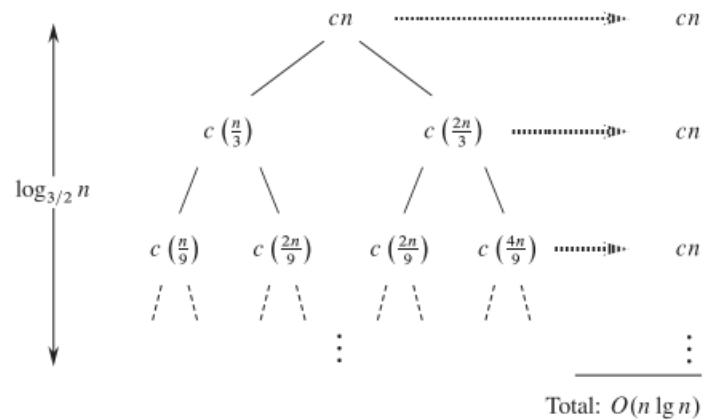
است. اگر همین روند را ادامه دهیم، درخت بازگشت کامل خواهد شد. البته در این مسئله درخت حاصل یک درخت نامتوازن است که عمق سمت چپ آن کمتر است و در عمق  $\log_3 n$  به برگ (**T(1)** می‌رسد. در حالی که عمق سمت راست  $\log_{3/2} n$  است. با کمی دقت در ساختار مسئله در می‌یابیم که مقدار ثابتی که از سطح آن بدست می‌آید اگر  $i \leq \log_3 n$  برابر با  $n$  و در غیر این صورت کوچک‌تر از  $n$  خواهد بود. چون می‌خواهیم مرتبه جواب را بدست آوریم، میتوانیم فرض کنیم مجموع همه سطوحها برابر  $n$  است. بنابراین:

$$T(n) \leq n \log_3 \frac{n}{2} = (n \log_2 n) \log_{3/2} 2 = c_1 n \lg n$$

$$T(n) \geq n \log_3 n = (n \log_2 n) \log_3 2 = c_2 n \lg n$$

و در نتیجه  $T(n) = \theta(n \lg n)$ . حال با استفاده از روش حدس و استقرا میتوانیم از جواب خود مطمئن شویم.

دقت کنید که در اینجا کمکی که درخت بازگشت به ما کرد حدس زدن شکل جواب بود. البته با توجه به قسمت‌های قبل شما این جواب را دیده بودید، اما هنگام رویارویی با یک مسئله جدید، شاید فقط به کمک درخت بازگشت بتوان یک حدس خوب زد.



شکل ۱ - درخت بازگشت برای عبارت  $T(n) = T(n/3) + T(2n/3) + cn$

#### ۴-۴ قضیه اصلی

قضیه اصلی یک راه حل مخصوص برای رابطه‌های بازگشتی به صورت  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  (که در آنها  $a$ ،  $b$  و  $f$  هر کدام شرایط خاصی دارند که در ادامه می‌آید) ارائه میدهد. این رابطه میتواند زمان اجرای

الگوریتمی را بیان کند که مسئله‌ای با اندازه  $n$  را به  $a$  زیر مسئله با اندازه‌های  $n/b$  تقسیم می‌کند. پس از حل شدن زیر مسئله‌ها با همین الگوریتم برای ترکیب آنها نیاز به زمانی برابر با  $f(n)$  داریم.

قضیه اصلی به صورت زیر بیان می‌شود:

فرض کنید  $1 \leq b < a$  و  $\epsilon > 0$  تابعی است که به صورت مجانبی مثبت است. رابطه بازگشتی

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

برای مقادیر مثبت  $n$  تعریف شده است. در این صورت:

الف) اگر  $f(n) = O(n^{\log_b a - \epsilon})$  (یعنی آهنگ رشد تابع  $f(n)$  از تابع  $n^{\log_b a}$  به صورت چند جمله‌ای کمتر باشد) در این صورت:  $T(n) = \Theta(n^{\log_b a})$

ب) اگر  $T(n) = \Theta(n^{\log_b a} \lg n)$ ، در این صورت:  $f(n) = \Theta(n^{\log_b a})$

ج) اگر  $T(n) = \Theta(f(n))$ ، در این صورت:  $f(n) = \Omega(n^{\log_b a + \epsilon})$

در واقع آهنگ رشد دو تابع  $f(n)$  و  $g(n) = n^{\log_b a}$  را با هم مقایسه می‌کنیم و آن تابعی که درجه رشد بیشتری دارد، به نوعی جواب را تعیین می‌کند. در مثال‌های زیر کاربرد این قضیه نشان داده شده است.

مثال ۱  $T(n) = 9T\left(\frac{n}{3}\right) + n^2$

حل :

$$\begin{aligned} a = 9, b = 3; \quad g(n) &= n^{\log_3 9} = n^2; \quad f(n) = n = O(n^{2-0.5}) \Rightarrow T(n) \\ &= \Theta(n^2) \end{aligned}$$

مثال ۲  $T(n) = T\left(\frac{2n}{3}\right) + 1$

حل :

$$a = 1, b = 3/2; \quad g(n) = n^{\frac{\log_3 1}{2}} = 1; \quad f(n) = 1 = O(n^0) \Rightarrow T(n) = \theta(lgn)$$

مثال ۳  $T(n) = 3T\left(\frac{n}{4}\right) + nlgn$

: حل

$$a = 3, b = 4; \quad g(n) = n^{\log_4 3}; \quad f(n) = nlgn = \Omega(n^{\log_4 3+0.1}) \Rightarrow T(n) = \theta(nlgn)$$

البته ذکر این نکته ضروری است که ۳ حالت فوق، همه‌ی حالت‌ها را در بر نمی‌گیرند و توابعی پیدا می‌شوند که هیچ یک از این حالت‌ها برای آنها صادق نیست. در این موارد نمی‌توان از قضیه اصلی استفاده کرد و باید از روش‌های دیگر حل رابطه بازگشتی به جواب رسید.

به عنوان نمونه قضیه اصلی در رابطه  $T(n) = 2T\left(\frac{n}{2}\right) + nlgn$  قابل استفاده نیست. در این عبارت داریم:

$$a = 2, b = 2; \quad g(n) = n^{\log_2 2} = n; \quad f(n) = nlgn$$

شاید در ابتدا این طور به نظر برسد که این رابطه حالت سوم قضیه اصلی است اما چون باید  $f(n) / g(n)$  یک عبارت چندجمله‌ای باشد تا رابطه در حالت سوم قرار گیرد و در اینجا حاصل  $f(n) / g(n) = lgn$  برابر با است که این عبارت کوچکتر از هر عبارت به صورت  $n^{\epsilon} - 0 < \epsilon$  است (به صورت حدی)، در نتیجه رابطه بازگشتی در حالتی بین حالت دوم و سوم قضیه اصلی قرار می‌گیرد و نمی‌توان از قضیه اصلی برای حل این رابطه استفاده کرد.

مثال ۴  $T(n) = 2T\left(\frac{n}{2} + 17\right) + n$

حل : اضافه کردن مقدار ثابت ۱۷ به رابطه  $T(n) = 2T\left(\frac{n}{2}\right) + n$  تاثیری در مرتبه پاسخ ندارد زیرا به ازای  $n$  های بسیار بزرگ تقریباً  $\frac{n}{2} + 17 = \frac{n}{2}$  است لذا داریم :

پاسخ رابطه فوق طبق قضیه اصلی برابر است با :  $O(nlgn)$

## ۵-۴ روش‌های خلاقانه

تنها تعداد کمی از مسائلی که ما با آنها مواجه می‌شویم با استفاده از روشهای بالا قابل حل است و بسیار دیگر برای حلشان باید از روشهای خلاقانه و ایده‌های نو استفاده کرد.

(۱) مثال

$$\binom{n}{m} = \frac{n!}{m!(n-m)!} \begin{cases} 1, & \text{if } m = 0 \text{ or } m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1}, & \text{otherwise} \end{cases}$$

باتوجه به رابطه فوق پیچیدگی زمانی شبه کد زیر را بدست آورید.

```

Int comb( int n , int m ){
    If( m == n || m == 0 )
        Return 1;
    Return comb(m , n-1) + comb(m-1 , n-1 );
}

```

حاصل تابع فوق برابر است با مجموع تعداد یک‌هایی که در طول اجرا کد return می‌شوند. بنابراین پیچیدگی زمانی این شبه کد بصورت زیر است :

$$T(n, m) = \begin{cases} 0, & \text{if } m = 0 \text{ or } m = n \\ 1 + T(n-1, m) + T(n-1, m-1), & \text{otherwise} \end{cases}$$

تغییر تابع زیر را درنظر می‌گیریم:

$$T(n, m) = H(m, n) - 1$$

پس داریم:

$$H(n, m) - 1 = \begin{cases} 0, & \text{if } m = 0 \text{ or } m = n \\ H(n-1, m) + T(n-1, m-1) - 1, & \text{otherwise} \end{cases}$$

$$H(n, m) = \begin{cases} 1, & \text{if } m = 0 \text{ or } m = n \\ H(n-1, m) + T(n-1, m-1), & \text{otherwise} \end{cases}$$

که این رابطه همان رابطه‌ی ذکر شده در صورت سوال است پس :

$$H(n, m) = \binom{n}{m} = T(n, m) + 1$$

در نتیجه :

$$T(n, m) = \binom{n}{m} - 1 = \begin{cases} 0, & \text{if } m = 0 \text{ or } m = n \\ \binom{n-1}{m} + \binom{n-1}{m-1} - 1, & \text{otherwise} \end{cases}$$

مثال ۲) پیچیدگی زمانی شبه کد زیر را بدست آورید.

فرض کنید  $f$  ارایه یک بعدی است که مقادیر آن در ابتدا ۱ است.

$$\text{Fib}(n)\{$$

if  $f[n] \geq 0$  then return  $f[n]$

if  $n = 0$  or  $n = 1$

Then  $f[n] = n$ ; return  $n$ ;

$$f[n] = \text{Fib}(n-1) + \text{Fib}(n-2)$$

```

    return ftab[n];
}

}

```

حل : این یک الگوریتم پویاست که در آن هر درایه یه آرایه  $ftab$  فقط یک بار وزمانی که مقدار فعلی آن منفی باشد محا سبه می شود. بنابراین هیچ زیر مسئله ای بیش از یک بار فراخوانی نمی شود پس مرتبه الگوریتم خطی است.  $O(n)$

چند رابطه سودمند

$$\sum_{i=1}^n a_i = \frac{a_1}{1-c}$$

$$\sum_{k=1}^n i^k = \frac{n(n+1)(2n+1)}{6}$$

$$H_n = \sum_{n=1}^{\infty} \frac{1}{n} = \ln n + \gamma + O\left(\frac{1}{n}\right)$$

$$e^x = 1 + x + x^2/2! + x^3/3! + \dots \quad 1+x < e^x < 1 + x + x^2/2$$

تقریب استرلینگ:

$$n!=\sqrt{2\pi n}\left[\frac{n}{e}\right]^n(1+o(\frac{1}{n}))$$

$$\text{نتیجه:}$$

$$N! = O \left( r^nn \right) \qquad \qquad \log{(n!)} = \Theta \left( nlg n \right)$$

$$f^j(n)=f(f(f(\ldots(n))))$$

$$\beta_r j$$

$$\lg\ ^{*}n=\min(\ j>=\cdot\ ;\log^jn<=j\ )$$

## ۱- داده گونه های مجرد و اولیه

### ۱-۱- ویژگی های یک برنامه خوب

۱. درست کار کند

۲. فهمیدن برنامه آسان باشد و به راحتی قابل تغییر باشد

۳. پس ضمینه ی استدلالی درست داشته باشد

یکی از راه ها برای رسیدن به هدف دوم استفاده از گونه داده ای مجرد (۱) است. ایده ی اصلی گونه های مجرد (ADT) جدا کردن روش های تشخیص است. یعنی اینکه بدانیم با چه نوع داده ای سر و کار داریم و چه نوع عملگر هایی بر روی این داده تعریف شده اند.

فاید استفاده از گونه های مجرد را میتوان در سه مورد خلاصه کرد:

۱. فهم آسان کد

۲. برای هدف های مختلف پیاده سازی برای گونه های داده ای مجرد (۲) ساده است، یعنی بدون اینکه نیاز باشد کل برنامه را تغییر دهیم با تغییرات اندک میتوان برنامه را برای گونه داده های مختلف استفاده کرد.

۳. ویژگی دوم سبب میشود تا بتوان کدی که بدین شکل نوشته شده را در آینده و برای پروژه های دیگر نیز استفاده کرد.

### ۲-۱- انواع گونه های مجرد

در کل دو نوع گونه ی مجرد داریم:

۱. public یا external که شامل قسمت های زیر است:

- دید ادراکی از تصویر داده (DID کاربر از اینکه داده ی مورد نظر چه شکلی است)

- دید ادراکی از عملکرد داده (DID کاربر نسبت به اینکه داده چه کارهایی میتواند بکند)

۲. private یا internal که شامل قسمت های زیر است:

- نمایش ساختاری داده (اینکه داده در حقیقت چگونه ذخیره میشود)

- اجرای عملگرهای داده (قسمت واقعی کد)

قسمت آخر یک گونه داده ی مجرد (اجرای عملگرهای داده) در کل میتواند شامل دسته های زیر باشد:

### ۱. تعریف داده

۲. افزودن داده ی جدید به داده های درون داده ی اصلی

۳. دسترسی به داده های درون داده ی اصلی

۴. پاک کردن داده های درون داده ی اصلی

از جمله ی این داده گونه ها می توان به Stack، Container، Queue، Graph، List و Set اشاره کرد.

### ۳-۱ - مراحل تبدیل یک کد به برنامه

۱. تجزیه (۳): تجزیه ی کد یعنی شکستن کد به قسمت های ریزتر، قبل از اینکه کد به یک اطلاعات معنی دار تبدیل شود باید تجزیه شود. بعد از تجزیه کد اماده ی مرحله ی بعدی است.

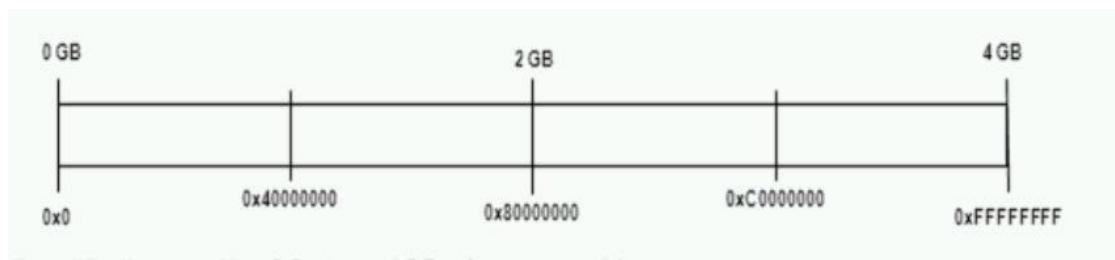
۲. کامپایل: کامپایل یک کد تجزیه شده در حقیقت تبدیل آن به یک برنامه است. البته این مرحله شامل ۲ گام است:

- تبدیل سورس کد تجزیه شده به یک object code -

- توسط یک لینکر لینک میشود

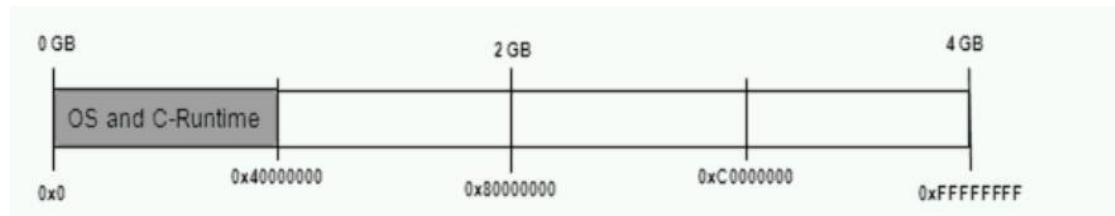
### ۴-۱ - مدیریت جافظه در حال اجرا در جاوا

جاوا یک پردازه در سطح سیستم عامل (۴) است. سیستم عامل و طراحی کامپیوتر یک سری محدودیت ها را برای اجرای برنامه به همراه می آورد. فرض میکنیم کامپیوتر دارای یک حافظه ی ۳۲ بیتی است، یعنی پهنانی هر یک از خانه های حافظه ۳۲ بیت است. اگر شماره ی خانه ها از ۰ تا FFFFFFFF باشد یعنی حجم حافظه ۴ GB خواهد بود.



انمایی از خانه های حافظه در یک کامپیوتر ۳۲ بیتی Figure ۱

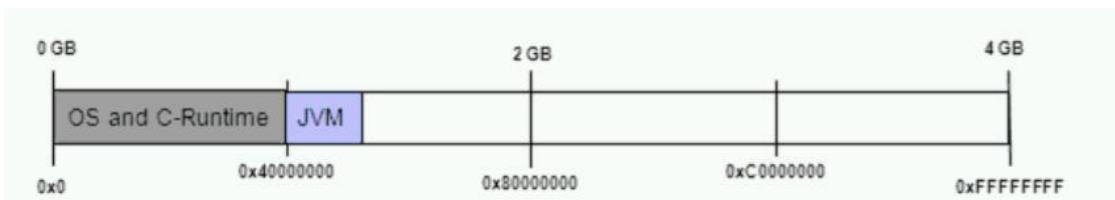
مقداری از حافظه به OS (سیستم عامل) و C-Language Runtime اختصاص میابد.



نحوه‌ی اختصاص خانه‌های ابتدایی حافظه Figure ۲

در سیستم عامل ویندوز این حجم ۲ گیگابایت است اما در لینوکس به ۱ گیگابایت کاهش میابد. به حجم باقیمانده فضای کاربری (user space) گفته میشود.

قسمتی از فضای حافظه هنگام اجرای جاوا به JVM اختصاص میابد. این حافظه شامل موتور اجرا و JIT Compiler وغیره میباشد.



JVM حافظه‌ی اختصاص داده شده به Figure ۳

بقيه ها اختصاص ميابند:

۱. java heap: هيپ های مربوط به اجرای برنامه که مقادیر ماکزيم و مينيم برايش تعیین ميشود

۲. native(System) heap: اين هيپ برای اجرای ريسمان های threads مختلف است

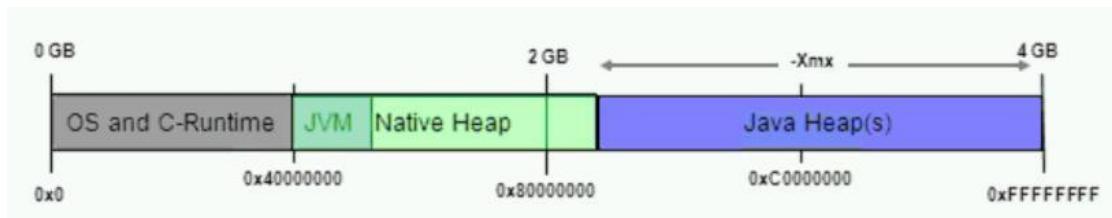


Figure ۴ حافظه های مربوط به هيپ ها

جاوا یک زبان statically-typed است . یعنی همه های متغیرها باید قبل از استفاده شدن تعریف شوند.

حال که داده گونه های مجرد را بررسی کردیم به یک نوع داده گونه ی دیگر میپردازیم یعنی داده گونه های اولیه (Primitive Datatype). این داده گونه ها ویژگی مشترکی با هم دارند که آنها را از non Primitive Data Type (method) و اينکه حافظه های ثابتی اشغال میکنند. داده گونه های اولیه به چند دسته تقسیم میشوند:

Type	Contains	Range	Storage Requirement(bit)	Default
boolean	true or false	NA	1	false
char	Unicode character unsigned	\u..... to \uFFFF	16	\u.....
byte	Signed integer	-128 to 127	8	.

short	Signed integer	-32768 to 32767	16	.
int	Signed integer	-2147483648 to 2147483647	32	.
long	Signed integer	-9223372036854775808 to 9223372036854775807	64	.
float	IEEE 754 floating point single-precision	1.4E-45 to 1.4028235E+38	32	.,.
double	IEEE 754 floating point double-precision	1.49E-324 to 1.7976931348623157E+308	64	.,.

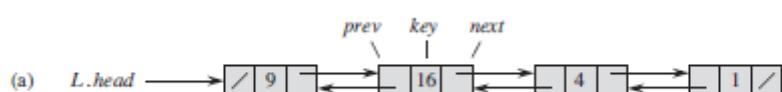
۱- انواع داده گونه های اولیه Table

## ۲- آرایه

چگونه می توان اشاره گر و اشیاء را در زبان هایی، مثل فرترن، که آن ها را پشتیبانی نمی کند پیاده سازی کرد؟ برای این کار می توان از جایگزینی آنها با آرایه و اندیس آرایه استفاده کرد. این تنها یک نمونه از کاربرد های فراوان آرایه میباشد که در این بخش به آن پرداخته میشود.

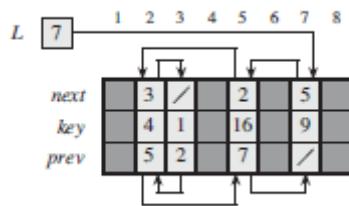
## ۱-۲- پیاده سازی یک لیست پیوندی با آرایه ی چند بعدی

شكل زیر پیاده سازی یک لیست پیوندی با آرایه را نشان می دهد.



## پیاده سازی لیست پیوندی با آرایه

برای این پیاده سازی ما میتوان از یک آرایه چند بعدی استفاده کرد که طول ردیف های آن سه است.



## پیاده سازی لیست پیوندی با آرایه

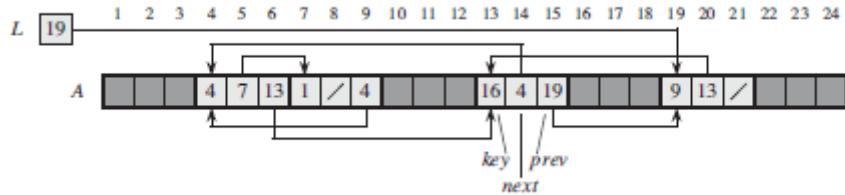
همانطور که در شکل بالادیده می شود، یک آرایه چند بعدی با سه ردیف داریم. ردیف اول اندیس خانه بعدی (next)، ردیف دوم کلید (key)، و ردیف سوم اندیس خانه قبلی (prev) هر جزء از لیست پیوندی را نگه میدارد. "L" نیز سر لیست پیوندی را مشخص میکند. هر قسمت عمودی از آرایه نمایش یک شی تنها است. خانه های روشن حاوی عناصر لیست هستند و خانه های تیره نیز بدون استفاده میباشند. برای مقداردهی خانه (( / )) از یک مقدار که در اندیس های حقیقی ظاهر نمیشود (مانند - ۱ ) استفاده میکنیم.

برای نمونه عدد ۴ در ستون دوم از ردیف key قرار دارد و مقدار next آن ۳ میباشد. بنابراین عنصر بعدی آن در لیست پیوندی در ستون سوم از آرایه ما قرار دارد که key آن عنصر ۱ است، پس عنصر بعد از عدد ۴ در لیست پیوندی، ۱ است.

از طرفی مقدار prev کلید ۴ ، ۵ بوده و بنابراین عنصر قبلی آن در لیست پیوندی در ستون پنجم از آرایه ما قرار دارد که key آن ۱۶ است، پس عنصر قبلی عدد ۴ در لیست پیوندی، ۱۶ است.

## -۲-۲ پیاده سازی لیست پیوندی و آرایه ی چند بعدی با آرایه یک بعدی

حال میخواهیم لیست پیوندی شکل ----- و آرایه ی چند بعدی شکل ----- را به وسیله یک آرایه یک بعدی نشان دهیم.



## پیاده سازی لیست پیوندی با آرایه یک بعدی

شکل بالا نحوه پیاده سازی لیست پیوندی شکل ---- و آرایه چند بعدی شکل ----- در یک آرایه یک بعدی را نشان میدهد.

به این ترتیب که key هر عنصر لیست پیوندی ما در یک خانه از آرایه با اندیس  $j$  (که باقیمانده آن بر  $3$ ،  $1$  است) قرار میگیرد. سپس  $prev$  آن عنصر در خانه های  $j+1$  (که باقیمانده آن بر  $3$ ،  $2$  است) و  $j+2$  (که مضرب  $3$  است) گذاشته میشود. همانند قبل  $L$  سر لیست پیوندی را مشخص میکند.

برای نمونه در خانه  $19$  ام عنصر با "key"  $9$  قرار دارد. در خانه  $20$  ام ( $1+19$ ) اندیس عنصر بعدی عدد  $19$  قرار دارد که  $13$  است و چون در خانه  $13$  ام عنصر  $16$  قرار دارد پس عنصر بعدی عدد  $9$  در لیست پیوندی  $16$  است. از طرفی در خانه  $21$  ام ( $2+20$ ) اندیس عنصر قبلی عدد  $19$  قرار دارد که " / " است. پس عنصر قبلی برای عدد  $9$  وجود ندارد و  $9$  در ابتدای لیست پیوندی قرار دارد.

## -۳-۲ - پیدا کردن آدرس خانه $i$ ام در یک آرایه

ما میخواهیم با داشتن آدرس اولین خانه از یک آرایه، آدرس هر کدام از خانه های آرایه را که میخواهیم بدست آوریم. بدست آوردن این آدرس با توجه به جنس دز نظر گرفته شده برای آرایه متفاوت است. در اینجا مبنای جنس آرایه را `int` در نظر گرفته می شود که هر خانه  $4$  بایت فضا را در حافظه اشغال میکند.

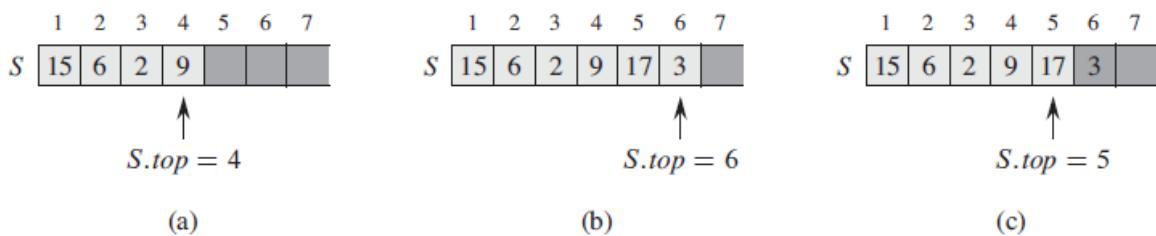
پس برای پیدا کردن خانه مورد نظر کافی است تا  $4$  برابر "تعداد خانه های قبل آن خانه" را به آدرس خانه‌هاول اضافه کنیم.

به عنوان مثال در یک آرایه دو بعدی مانند  $A[3][5]$  و با فرض اینکه آدرس خانه‌ی اول آرایه ۱۰۰۰ است، آدرس خانه‌ی  $A[2]$  به این صورت بدست می‌آید.

$$\text{Address of } A[2] = 1000 + (2 * 3 * 4) = 1024$$

### ۳- پشته :

پشته یا "استک" ساختمان داده‌ای "LIFO" یا "Last In First Out" است، یعنی اولین خروجی از پشته، آخرین ورودی پشته است. همچنین پشته دارای دو عمل PUSH و POP است، push یک متغیر را داخل پشته و روی متغیرهایی که قبل از داخل پشته بوده قرار میدهد و pop یک مغایر را از سر پشته برداشت و به عنوان خروجی بیرون میدهد.



### مثالی از پشته

در قسمت a در شکل ----- یک ساختمان داده پشته با ۴ عنصر دیده می‌شود. در این لحظه عددی که تابع  $\text{top}()$  میدهد ۴ است که یعنی بالاترین عنصر پشته در خانه چهارم قرار دارد.

در قسمت b در این شکل، دو عدد ۱۷ و ۳ به ترتیب در پشته push شده‌اند. اینبار عدد برگردانده شده توسط تابع  $\text{top}()$ ، ۶ است، چون دو عنصر به پشته ما اضافه شده است.

در قسمت c، ابتدا تابع POP فراخوانی شده و بنابراین عنصر سر پشته که ۳ بود از آن خارج شده است. حالا اینبار عدد برگردانده شده توسط تابع  $\text{top}()$ ، می‌شود.

شبه کد تابع PUSH در پشته : ( مقدار اولیه ۱ ( top=-۱

```
Int push(Datatype x) {  
    If ( is.full() )  
        Return -1;  
    S[++top] = x;  
    Return top;  
}
```

شبه کد تابع POP در پشته : ( مقدار اولیه ۱ ( top=-۱

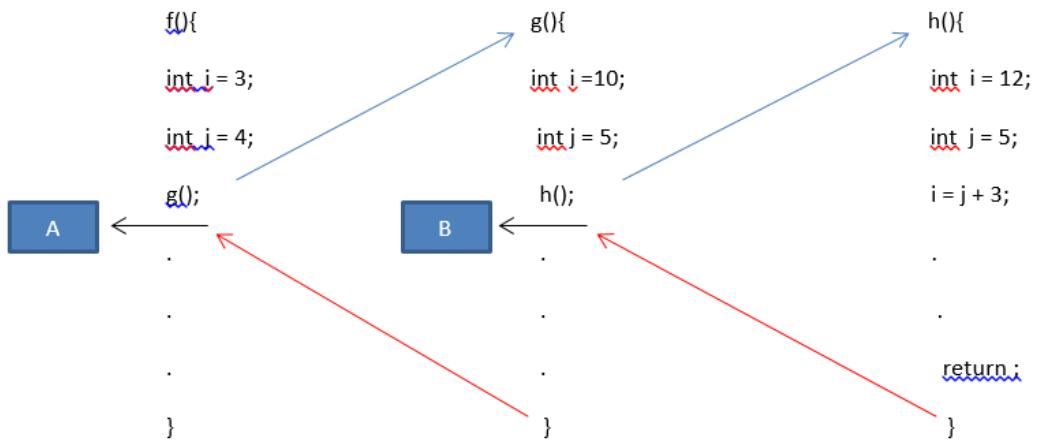
```
Datatype pop() {  
    If ( is.empty() )  
        Return -1;  
    Return S[top--];  
}
```

### کاربردهای پشته -۱-۳

از کاربردهای متعدد پشته میتوان به مدیریت توابع و عبارت های محاسباتی اشاره کرد.

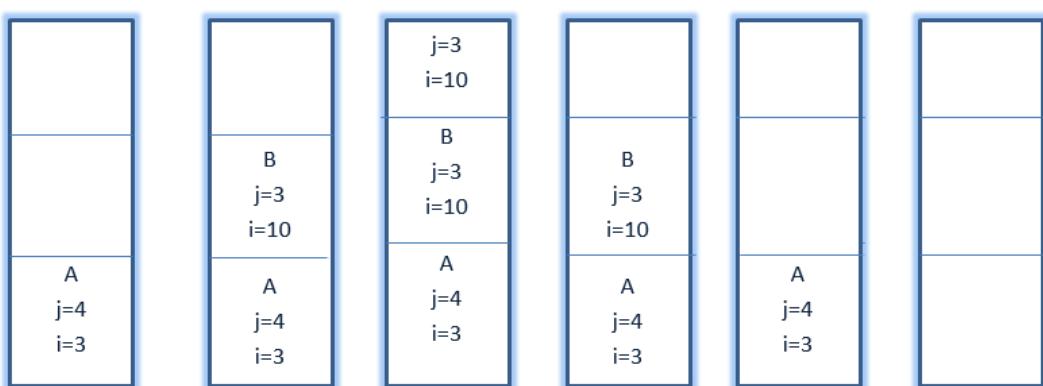
### مدیریت توابع -۱-۱-۳

در فرآخوانی توابع و پیاده سازی آنها ما از پشته ها و توابع آنها استفاده میکنیم. مثال زیر طرز پیاده سازی و استفاده از این توابع را برای شما روشن میکند :



## مدیریت و فرآخوانی توابع

برای انجام تابع `f()` ما از یک پشته استفاده می کنیم. در ابتدا متغیر های خود تابع `f()` در قسمتی از پشته PUSH میشوند. سپس آدرس مکان A (آدرس بازگشت) و سپس متغیر های تابع `g()` و آدرس مکان B و در آخر متغیر های تابع `h()` PUSH میشوند. پس از پایان کار تابع `h()` و خارج شدن از آن، متغیر های آن نیز از پشته POP میشوند و فرآیند کار به آدرس B بازمیگردد و ادامه تابع `g()` تا پایان انجام شده و پس از اتمام آن متغیر های این تابع نیز از پشته POP شده و فرآیند کار به آدرس A میرود و ادامه تابع `f()` انجام شده و کار به پایان میرسد؛ و بدین صورت فرآیند تابع `f()` با استفاده از یک پشته و دو تابع PUSH و POP آن پیاده سازی می شود.



## مراحل فرآخوانی و بازگشت از تابع

شکل بالا ۶ مرحله انجام این تابع را به ما نشان می دهد.

### ۲-۱-۳ عبارت های محاسباتی

اولویت عملگرهای محاسباتی: ۱ - (داخل پرانتز) ۲ - (یکتایی ها) ۳ - ( $^{^{\wedge}}$ ) ۴ - ( $\% / *$ ) ۵ - (-) (+ -)

انواع عبارت های محاسباتی:

الف) پیشوند(prefix): عملگر قبل از عملوند می آید.

ب) میانوند infix: عملگر وسط از عملوندها می آید.

پ) پسوند(postfix): عملگر بعد از عملوند می آید.

انواع عملگرهای محاسباتی:

الف) یکتایی unary: مثل sin, cos, ++, --, ~

الف) دوتایی binary: مثل +, -, \*, /, %

الف) سه تایی tinary: مثل a?b:c

### تبديل عبارت میانوندی به پسوندی با استفاده از پشته

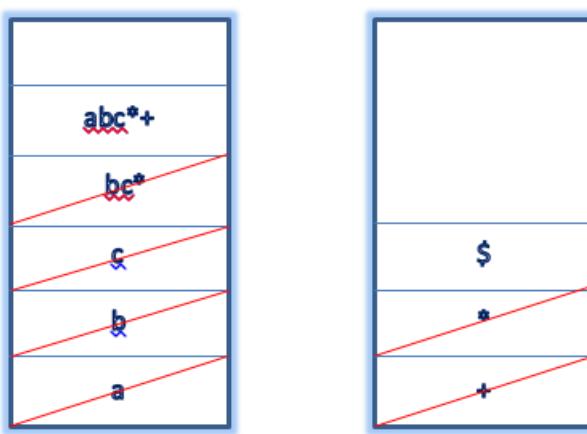
-۳-۱-۳

ما میتوانیم این کار را با استفاده از دو پشته انجام دهیم . یک پشته برای " عملوند ها " و یک پشته برای " عملگرها ". از ابتدای عبارت میانوندی شروع میکنیم و عملوند ها را در پشته خود و عملگر ها را نیز در پشته خود PUSH میکنیم. در کردن عملگر ها در پشته باید به این نکته توجه کنیم که یک عملگر فقط روی عملگری با اولویت کمتر از خود PUSH میشود ، یعنی ما نمیتوانیم یک عملگر را روی عملگری با اولویت بیشتر از خود PUSH کنیم. اگر این اتفاق در حال افتادن بود ما باید عملگر قبلی را POP کرده و با توجه به نوع عملگر(یکتایی ، دوتایی یا سه تایی ) یک یا دو یا ۳ عنصر از پشته عملوند ها را POP کرده و عملگر را روی آنها اعمال و عبارت پسوندی حاصل را در پشته عملوند ها PUSH میکنیم و این کار را تا جایی ادامه میدهم که در پشته عملگرها فقط عملگر \\$ که نشان دهنده پایان کار است باقی بماند.

نکات:

- ۱- قبل از انجام عملیات در پایان هر عبارت میانوندی عملگر \\$ را میگذاریم. این عملگر دارای کمترین اولویت بوده و نشان دهنده پایان عملیات میباشد.
- ۲- یک عملگر نمیتواند روی همان عملگر در پشته PUSH شود به جز عملگر ^ (توان).

برای مثال عبارت  $a+b*c\$$  را به شکل پسوندی تبدیل میکنیم.



پشته عملوند

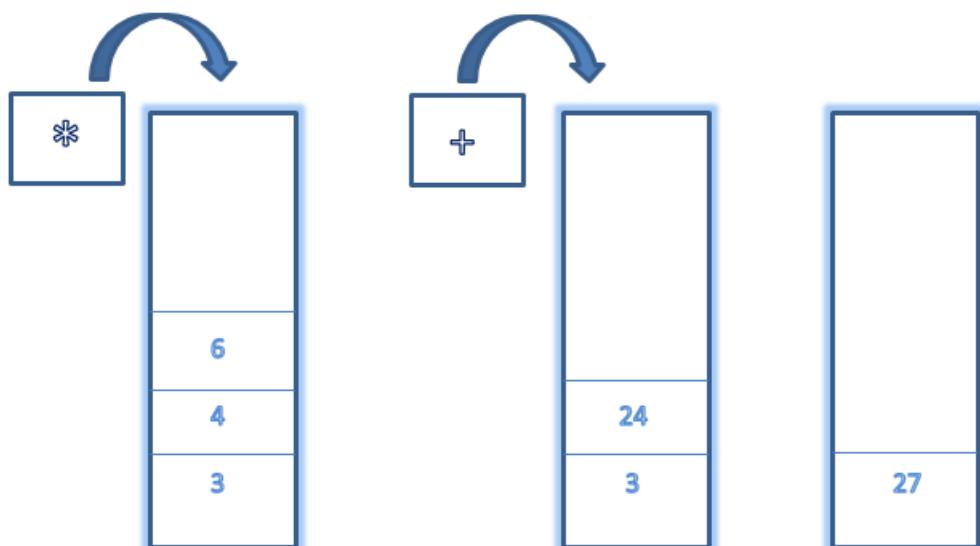
پشته عملگر

مراحل انجام عملیات :

- ۱- ابتدا به ترتیب  $a$  و  $b$  و  $c$  در پشته عملوند ها، و  $+$  و  $*$  در پشته عملگر ها PUSH میشوند.
- ۲- بعد نوبت به PUSH شدن \$ میرسد ولی چون اولیتش از  $*$  کمتر است نمیتواند وارد شود پس  $*$  باید POP شود.
- ۳- عملگر  $*$  و دو عملوند سر پشته عملوندها یعنی  $b$  و  $c$  POP شده و حاصل پسوندی آنها یعنی  $bc^*$  در پشته عملوند ها PUSH میشود.
- ۴- حالا دوباره \$ میخواهد وارد شود ولی چون اولیتش کمتر از  $+$  است نمیتواند و برای همین  $+$  POP شده و مانند مرحله قبل اینبار  $c$  و  $bc^*$  از پشته عملوندها POP شده و  $abc^*$  به جای آنها PUSH میشوند.
- ۵- حالا \$ در پشته عملگر ها PUSH میشود بنابراین عملیات تمام است و عبارت پسوندی ما بدست آمده است.

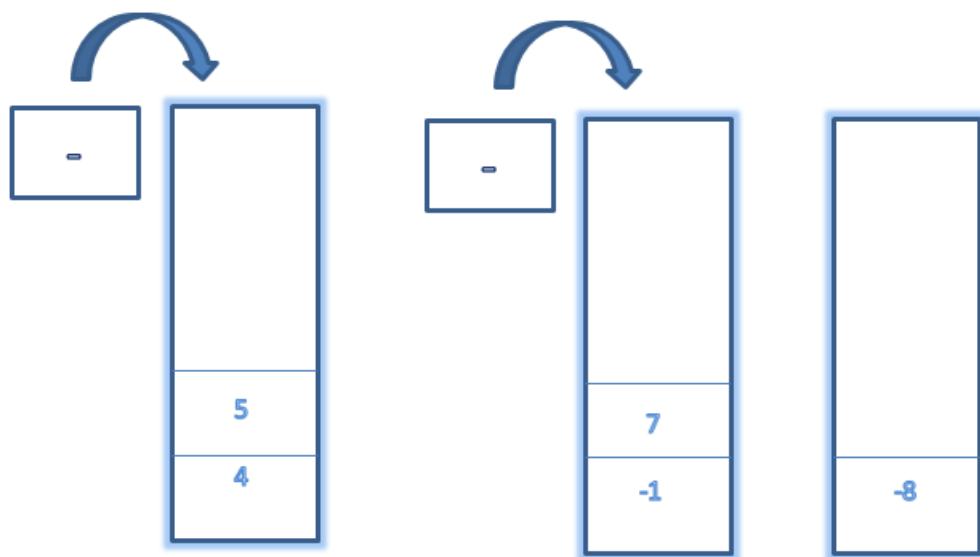
#### -۴-۱-۳ نحوه محاسبه مقدار عددی یک عبارت پسوندی با پشته

به عنوان مثال مقدار عددی عبارت  $+346*$  را محاسبه می کنیم.



از ابتدای عبارت پسوندی خود شروع کرده و عملوندها را در پشته PUSH میکنیم تا جایی که به یک عملگر برسیم. وقتی به عملگر رسیدیم بسته به نوع آن، اگر یکتایی بود عنصر سر پشته، اگر دوتایی بود دو عنصر سر پشته و اگر سه تایی بود ۳ عنصر سر پشته را POP کرده و عملگر را روی آنها اعمال میکنیم و حاصل را در پشته PUSH میکنیم، و این کار را ادامه میدهیم تا جایی که حاصل عبارت ما بدست آید.

شکل زیر محاسبه‌ی عبارت ۴۵-۷ را نشان می‌دهد.



#### ۴- ساختمان داده‌های ساده

در این فصل نمایشی از مجموعه‌های پویا که به وسیله ساختمان داده‌های ساده‌ای که از اشاره‌گرها استفاده می‌کنند را مورد مشاهده قرار می‌دهیم. با وجود آنکه بسیاری از ساختمان داده‌های پیچیده را می‌توان با اشاره‌گرها پیاده‌سازی کنیم، اما در اینجا فقط ساختمان داده‌های اصلی مانند پشته، صف، لیست پیوندی و درخت‌های ریشه‌دار را مورد بررسی قرار می‌دهیم. ما همچنین روش‌هایی را که به وسیله آن اشیاء و اشاره‌گرها به وسیله آرایه ترکیب می‌کنند را مورد بررسی قرار می‌دهیم.

#### ۱-۴- پشته‌ها و صف‌ها

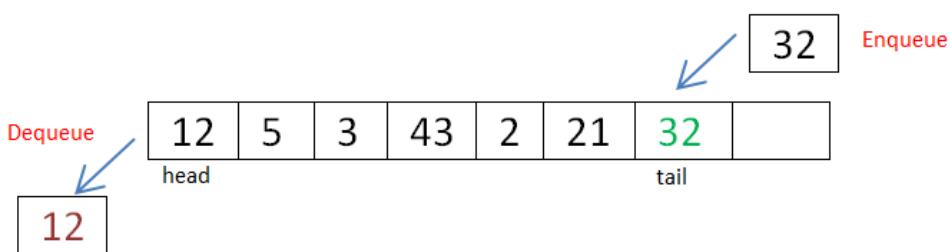
پشته (Stack) و صف (Queue) مجموعه‌های پویا هستند که در آن، عناصر به وسیله عمل پاک کردن (Delete) از مجموعه خارج می‌شوند. در پشته عنصر پاک شده از مجموعه همان آخرین عنصری که وارد آن شده است. به عبارت دیگر در پشته قانون «آخرین ورودی، اولین

خروجی»(Last-in, First-out) برقرار است. به طور مشابه در صف همیشه عنصری پاک می شود که مدت طولانی تری در صف بوده است. بنابراین صف با روش «اولین ورودی، اولین خروجی»(First-in, First-out) پیاده سازی می شود. چندین راه برای پیاده سازی صف و پشتنه در کامپیوتر وجود دارد. در این قسمت نشان می دهیم که چگونه از آرایه برای پیاده سازی هر دو ساختمان داده استفاده کنیم.

## صفها -۲-۴

### اضافه کردن (Enqueue) و خارج کردن (Dequeue) از صف -۱-۲-۴

عملیات وارد کردن یک عنصر به صف Enqueue و عملیات خارج کردن عنصر Dequeue نامیده می شود و مانند عمل Pop در پشتنه Dequeue هیچ آرگومانی را به عنوان ورودی دریافت نمی کند. ویژگی "اولین ورودی، اولین خروجی" در صف سبب می شود که عملکردی مشابه صف دانشجویان در صف دریافت غذا داشته باشد. هر صف یکابتدا (Head) و یک انتهای (Tail) دارد. هر زمانکه یک عنصر وارد صفحه شود، همیشه در انتهای صف قرار می گیرد. درست مانند صف سلف دانشجویان که دانشجوی تازه وارد در آخر صف قرار می گیرد. همچنین همیشه عنصری حذف می شود که در سر صف قرار دارد، مانند دانشجویی که بعد از مدتی طولانی به سر صف سلف رسیده و غذا دریافت می کند. (شکل ۱)



شکل ۱. ورود و خروج از ساختمان داده‌ی صف

صف دو ویژگی ویژگی دارد. Q.head که سر صف را مشخص می کند و Q.tail که موقعیت مکانی را که عنصر جدید قرار است وارد آن شود را نشان می دهد. عناصر در صف در مکان های Q.head و ... Q.tail-۱ قرار دارند، به طوری که «دور می زند» یعنی بعد از مکان ۱ مکان ۲ ... ۱ مکان به صورت دایره وار قرار دارد. بنابراین هرگاه  $Q.head = Q.tail$  صف خالی است. در حالت اولیه داریم  $Q.head = Q.tail = 1$ . وقتی صف خالی است سعی برای حذف کردن یک عنصر از صف با

«زیرریز» همراه است. هرگاه صف پر است و اگر بخواهیم یک عنصر به صف اضافه کنیم آنگاه صف «سر ریز» می‌کند.

در شبه کدهای زیر کنترل خطای مربوط به «سرریزی» و «زیرریزی» را حذف کرده‌ایم.  
 $n = Q.length$

## ENQUEUE (Q, X)

۱.  $Q[Q.TAIL] = X$
۲. IF  $Q.TAIL == Q.LENGTH$
۳.  $Q.TAIL = 1$
۴. ELSE
۵.  $Q.TAIL = Q.TAIL + 1$

## DEQUEUE (Q)

۱.  $X = Q[Q.HEAD]$
۲. IF  $Q.TAIL == Q.LENGTH$
۳.  $Q.HEAD = 1$
۴. ELSE
۵.  $Q.HEAD = Q.HEAD + 1$
۶. RETURN X

## تمرين‌ها -۲-۲-۴

(تمرين ۱) نشان دهيد که چگونه می‌توان دو پشته بر روی آرایه  $A[1..n]$  پياده‌سازی کرد به طوری که هیچ‌کدام از پشته‌ها overflow نکنند مگر آنکه تعداد کل اعضا در هر دو پشته  $n$  شود. هزينه عمل push و pop باید  $O(1)$  باشد.

(تمرين ۲) اگر شکل ۱۰.۲ را به عنوان مدل بگيريم نتيجه هر يك از عمل‌های منظم و پشت سر هم Enqueue(Q,  $.Dnqueue(Q)$ ,  $.Enqueue(Q, ۳)$ ,  $.Enqueue(Q, ۱)$ ,  $.Enqueue(Q, ۴)$ ,  $.Dequeue(Q)$ ) مشخص کنيد. صف در حالت اوليه خالي است.

(تمرين ۳) Dequeue و Enqueue را دوباره بنويسيد تا underflow و overflow را در يك صف مشخص کنيد.

(تمرين ۴) پشته اجازه اضافه کردن و خارج کردن عناصر را از يك طرف می‌دهد حال صف اجازه وارد کردن از يك طرف و خارج کردن از طرف ديگر را می‌دهد. يك صف دوسرطراحي کنيد که اجازه اضافه کردن و خارج کردن عناصر را از دو طرف بدهد.  $O(1)$  تابع با هزينه بنيسيد که عمل اضافه کردن و خارج کردن عناصر يك صف دردو سر صف که با استفاده از آرایه پياده‌سازی می‌شود، را انجام دهد.

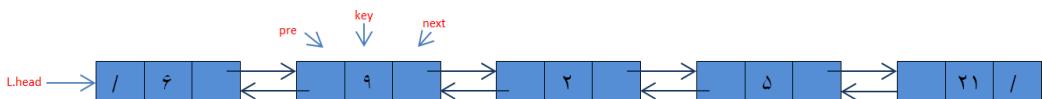
(تمرين ۵) نشان دهيد که چگونه می‌توان يك صف را با استفاده از دو پشته پياده‌سازی کرد. هزينه عمليات‌های صف را به دست آوريد.

(تمرين ۶) نشان دهيد که چگونه يك پشته را می‌توان با استفاده از دو صف پياده‌سازی کرد. هزينه عمليات‌های پشته را به دست آوريد.

### -۳-۴ لیست‌های پیوندی

لیست‌های پیوندی نوعی از ساختمان داده است که در آن اشیاء در یک ترتیب خطی قرار گرفته‌اند. بر خلاف آرایه که در آن ترتیب‌ها به وسیله اندیس‌های آرایه مشخص می‌شود، ترتیب در لیست‌های پیوندی به وسیله اشاره‌گر داخل هر شی مشخص می‌شود. لیست‌های پیوندی یک نمایش ساده و قابل انعطافی برای مجموعه‌های پویا فراهم می‌کند.

با توجه به شکل زیر هر یک از عناصر یک لیست پیوندی دو طرفه  $L$ ، یک شی است که یک متغیر کلید key و دو متغیر اشاره‌گر: بعدی next و قبلی prev دارد. یک شی ممکن است متغیرهای جانبی دیگری هم داشته باشد. با در نظر گرفتن عنصر  $X$  در لیست  $x.next$  به عنصر بعدی و  $x.prev$  به عنصر قبلی اشاره می‌کند. اگر  $x.next = NIL$  باشد یعنی عنصر قبل از آن وجود ندارد و این بدان معنا است که  $X$  سر لیست است. اگر  $x.next = NIL$  باشد، یعنی عنصری بعد از آن وجود ندارد این بدان معنا است که  $X$  عنصر آخر لیست است. صفت  $L.head$  به اولین عنصر لیست اشاره می‌کند. اگر  $L.head = NIL$  لیست خالی است.



شکل ۲ ساختمان داده‌ی لیست پیوندی

یک لیست می‌تواند به فرم‌های گوناگون باشد. ممکن است لیست دوطرفه یا یک‌طرفه باشد. ممکن است مرتب شده یا غیرمرتب باشد و همچنین ممکن است حلقوی یا غیرحلقوی باشد. اگر لیست پیوندی یک طرفه باشد، اشاره‌گر  $prev$  را از عناصر حذف خواهیم کرد، اگر لیست مرتب باشد ترتیب عناصر کلیدها در لیست ترتیب خطی دارند و عنصر مینیمم در سر لیست و عنصر ماکزیمم در ته لیست قرار دارد. اگر لیست نامرتب باشد عناصر به هر صورتی قرار خواهند داشت. در لیست پیوندی حلقوی اشاره‌گر  $prev$  سر لیست، به آخر لیست اشاره می‌کند و اشاره‌گر  $next$  از ته لیست به سر لیست اشاره می‌کند. ممکن است که لیست به صورت حلقه نمایش داده شود. توجه داشته باشید در این قسمت فرض می‌کنیم لیست‌ها دو طرفه و نامنظم هستند.

### -۱-۳-۴ جستجو در لیست پیوندی

تابع  $(k)$  اولین عنصری که کلید  $k$  را دارد در لیست  $L$  پیدا می‌کند که با یک جستجوی خطی ساده انجام می‌شود و اشاره‌گری به آن عنصر باز می‌گرداند. اگر هیچ شی‌ای با کلید  $k$  List\_Search( $L, k$ ) نشود  $NIL$  باز خواهد گرداند. برای لیست پیوندی در شکل بالا تابع  $(2)$

اشاره‌گری به عنصر سوم باز خواهد گرداند و تابع  $\text{List\_Search}(L, k)$  مقدار بازگشته  $\text{NIL}$  خواهد داشت.

$\text{List\_Search}(L, k)$

۱.  $x = L.\text{head}$
۲. While  $x \neq \text{NIL}$  and  $\text{key}[x] \neq k$
۳. do  $x = x.\text{next}$
۴. return  $x$

برای جستجو در یک لیست پیوندی با  $n$  عنصر در بدترین حالت با  $\Theta(n)$  طول می‌کشد چون باید کل لیست را جستجو کند.

#### -۴-۳-۲- اضافه کردن به لیست پیوندی

تابع  $\text{List\_Insert}$  عنصری را که کلید آن  $x$  است را به اول لیست اضافه می‌کند.

$\text{List\_Insert}(L, x)$

۱.  $x.\text{next} = L.\text{head}$
۲. If  $L.\text{head} \neq \text{NIL}$
۳.  $L.\text{head}.\text{prev} = x$
۴.  $L.\text{head} = x$
۵.  $x.\text{prev} = \text{NIL}$

زمان اجرایی تابع  $\text{List\_Insert}$  برای اضافه کردن یک عنصر به لیست پیوندی  $n$  عنصری از  $O(1)$  است.

## -۳-۳-۴ حذف کردن از لیست پیوندی

تابع List\_Delete عنصر  $x$  را از لیست  $L$  حذف خواهد کرد. باید اشاره‌گری به عنصر  $x$  داشته باشیم و بعد آن را از لیست حذف خواهیم کرد. اگر بخواهیم عنصری با یک کلید از قبل مشخص شده را حذف کنیم، ما باید اول List\_Search را صدا بزنیم تا اشاره‌گر به آن عنصر را داشته باشیم.

List\_Delete( $L, x$ )

۱. If  $x.prev \neq NIL$
۲.  $x.prev.next = x.next$
۳. Else
۴.  $L.head = x.next$
۵. If  $x.next \neq NIL$
۶.  $x.next.prev = x.prev$

تابع List\_Delete با هزینه  $O(1)$  اجرا می‌شود. اما اگر بخواهیم عنصری را با کلید مشخص خذف کنیم هزینه اجرای آن در بدترین حالت از  $O(n)$  است. برای اینکه ما باید اول List\_Search را صدا بزنیم.

## -۴-۳-۴ تمرین

(تمرین ۱) اضافه کردن یک عنصر بر روی یک لیست پیوندی یک طرفه با هزینه  $O(1)$  اجرا می‌شود؟ در مورد حذف کردن چه طور؟

(تمرین ۲) پشته را با استفاده از یک لیست پیوندی  $L$  پیاده سازی کنید عملیات‌های push, pop باید باز هم با  $O(1)$  اجرا شود.

(تمرین ۳) صف را با استفاده از یک لیست پیوندی  $L$  پیاده سازی کنید. عملیات‌های Dequeu, Enqueue باید باز هم با  $O(1)$  اجرا شود.

(تمرین ۴) با توجه به تابع 'List\_Search' که حلقه آن برای هر دفعه باید دو مورد را چک کند. یکی  $x!=NIL$  و دیگری  $key[x]!=k$  نشان دهید که چگونه چک کردن  $x!=NIL$  را حذف کنیم.

(تمرین ۵) عملیات اضافه و حذف و جستجو را در مورد فرهنگ لغت با یک لیست پیوندی یک طرفه دایروی پیاده سازی کنید. هزینه اجرایی آن را محاسبه کنید.

(تمرین ۶) عملگر مجموعه پویا Union دو مجموعه مجزای  $S_1, S_2$  از ورودی می‌گیرد و  $S = S_1 \cup S_2$  را باز خواهد گرداند که محتویات تمام اعضای  $S_1, S_2$  است. مجموعه‌های  $S_1, S_2$  در این عمل از بین خواهند رفت. نشان دهید که چگونه عملیات Union را با هزینه  $O(1)$  با توجه به ساختمان داده مناسب تامین کنیم.

(تمرین ۷) یک تابع معرفی کنید که غیر بازگشتی باشد و یک لیست پیوندی یک طرفه با  $n$  عنصر را معکوس سازد. حافظه‌ای بیشتر از حافظه که برای لیست پیوندی بکار می‌رود نگیرید.

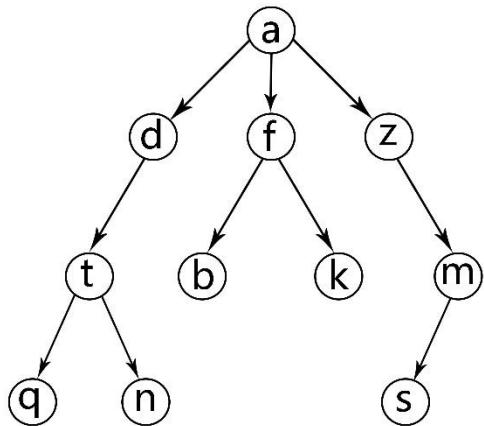
(تمرین ۸) توضیح دهید که چگونه یک لیست پیوندی دو طرفه را با یک اشاره‌گر  $x.np$  که جایگزینه‌ر دو اشاره‌گر  $x.next$  و  $x.prev$  می‌شود پیاده سازی کنیم. فرض کنید هر اشاره‌گر را بتوان با یک عدد صحیح  $k$  بیتی نمایش دهیم و  $np$  به صورت زیر تعریف شده است:  $x.np = np^k$ . اطلاعات لازم برای دسترسی به سر لیست را مشخص کنید. نشان دهید که چگونه عملیات Insert, Delete, Search را روی لیست پیاده سازی کنید. نشان دهید که چگونه لیست را معکوس کنیم که هزینه اجرایی آن  $O(1)$  باشد.

# فصل چهارم

## ساختمان داده‌ی درخت

### مقدمه

درخت (Tree) یک ساختمان داده‌ی پرمصرف در رایانه است که از تعدادی گره تشکیل شده که با یال‌هایی به هم وصل شده‌اند.



هر گره ممکن است تعدادی فرزند داشته باشد. گره‌هایی که فرزند دارند برای فرزندانشان پدر محسوب می‌شوند.

درخت یک گراف همبند بدون دور است و بین هر دو گره یک درخت، یک مسیر یکتا وجود دارد. مانند:

### تعریف بازگشتی درخت

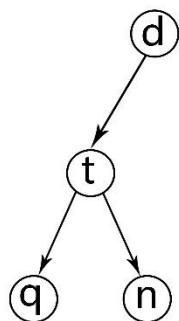
- ۱- یک گره به تنها یکی درخت است (Root)
- ۲- اگر  $T_1, T_2, \dots, T_k$  درخت باشند، از به هم چسباندن آن‌ها به Root، یک درخت به صورت بازگشتی تشکیل می‌شود.

## زیر درخت

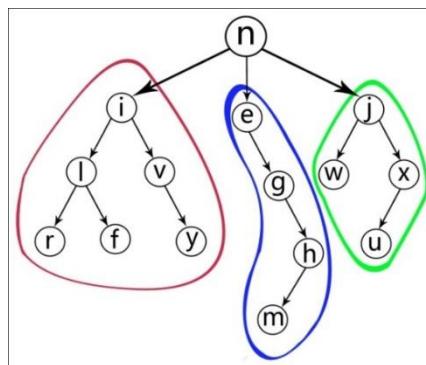
زیر درخت بخشی از درخت است که خود یه درخت کامل را تشکیل می دهد.

هر گره درخت را می توان به عنوان ریشه‌ی یک زیردرخت در نظر گرفت.

مانند شکل رو برو که در آن گره  $d$  نقش ریشه‌ی زیردرخت را دارد.



از چسپاندن چند درخت (زیر درخت) به یک گره ( $n$ )، درخت جدیدی ایجاد می شود.(در درخت جدید  $n$  ریشه است).



## اجزای مهم درخت

۱-ریشه (Root) : در درخت به گره‌ای که بالاتر از همه قرار دارد و دارای پدر نیست ریشه می گویند.(مانند گره  $a$  در درخت بالا)

۲-برگ (Leaf) : به گره‌ای که هیچ فرزندی ندارند و درنتیجه درجه اش  $0$  است ، برگ می گویند. (مثلاً گره  $a$  دارای سه فرزند  $d$  و  $f$  و  $z$  است اما گره  $s$  هیچ فرزندی ندارد.)

۳-گره داخلی (Internal Node) : به گره‌هایی که برگ نباشند - یعنی گره‌هایی که حداقل یک فرزند دارند - گره داخلی می گویند.(مانند گره‌های  $a$  و  $f$  در درخت بالا)

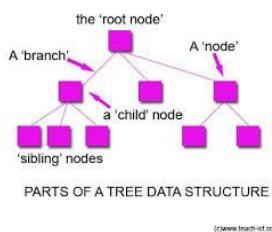
درخت یک ساختمان داده‌ی غیر خطی (non-linear) است. یعنی عناصر آن را نمی‌توان ترتیب خاصی بخشد. همچنین درخت یک گونه داده‌ای مجرد (ADT) است.

## تعاریف اولیه

**درجه‌ی گره (Degree)** : به تعداد زیردرخت‌های هر گره (یا تعداد فرزندان هر گره)، درجه‌ی آن گره می‌گویند.

درجه‌ی درخت : به بزرگترین درجه‌ی گره‌های درخت، درجه‌ی درخت می‌گویند.

**همزاد (Siblings)** : فرزندان یک گره همزاد هستند.

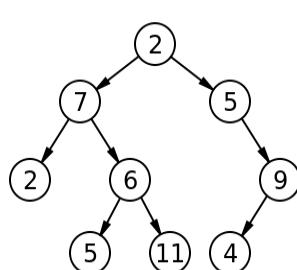


**سطح گره (Level)** : تعداد یال‌های مسیر آن گره تا ریشه را سطح یا ارتفاع گره می‌گویند.

**ارتفاع درخت (Height of Tree)** : به بیشترین سطح گره‌های یک درخت، ارتفاع یا عمق درخت می‌گویند.

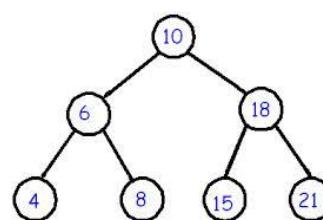
**اجداد یک گره (Ancestor)** : به گره‌های مسیر گره تا ریشه اجداد آن، گره گفته می‌شود.

**درخت مرتب (Ordered Tree)** : درختی است که در آن ترتیب خاصی بین فرزندان اعمال می‌شود. (درخت ناممرتب درختی است که در آن فرزندان هر رأس ترتیب خاصی ندارند)



مثالی از یک درخت ناممرتب

۲ گره با مقدار ۲ داریم که یکی پدرش ۷ است و دیگری پدری ندارد (ریشه است)



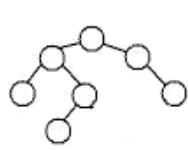
مثالی از یک درخت مرتب

این درخت یک درخت جستجوی دودوبی نیز هست

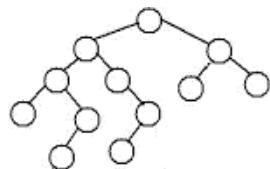
درخت  $K$  تایی : درختی که حداقل درجه هر گره  $K$  باشد.

درخت  $K$  کامل : درختی که درجه همه گره ها به جز برگ ها دقیقاً  $K$  باشد.

درخت متوازن (Balanced Tree) : درختی که اختلاف ارتفاع برگ ها در آن حداقل یک است.



درخت متوازن



درخت نامتوازن

درخت کاملاً متوازن : درختی که در آن اختلاف ارتفاع برگ ها صفر است.

جنگل : به بیشتر از یک درخت کنار هم جنگل می گویند.

## قضیه اولیه

در درخت تعداد یال ها کمتر از تعداد گره هاست. به بیان ریاضی :

$$1 - \text{تعداد گره ها} = \text{تعداد یال ها}$$

$$E = V - 1$$

اثبات با استقرا روی تعداد گره های یک درخت :

$$|V| = 1 \Rightarrow E = 0$$

حکم برای  $V = 1$  برقرار است.

فرض استقرا : هر درخت با  $V-1$  گره دارای  $E = V - 2$  یال است.

اگر از درخت با  $V$  گره یک گره کم کنیم درنتیجه  $1 - V$  گره داریم که با توجه به فرض استقرا  
 $2 - V$  یال داریم.

حال با اضافه کردن همان گره سر جای خودش یک یال اضافه می شود در نتیجه در درخت با  $V$   
 گره  $1 - V$  یال داریم. به این ترتیب قضیه اثبات می شود.

مثال : تعداد برگ های یک درخت  $K$  تایی کامل با  $n$  گره چقدر است ؟

اگر تعداد برگ ها را  $B$  در نظر بگیریم و با توجه به قضیه ای اویلر برای تعداد یال ها داریم :

$$E = n - 1$$

$$(n - B)K = E \quad \Rightarrow \quad B = n - \frac{n - 1}{K}$$

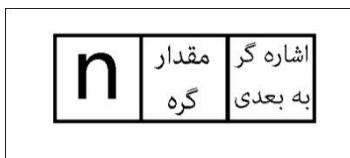
## ۴ - ۱ : روش های پیاده سازی درخت

درخت در کامپیوتر به روش های مختلفی پیاده سازی می شود که عبارت اند از :

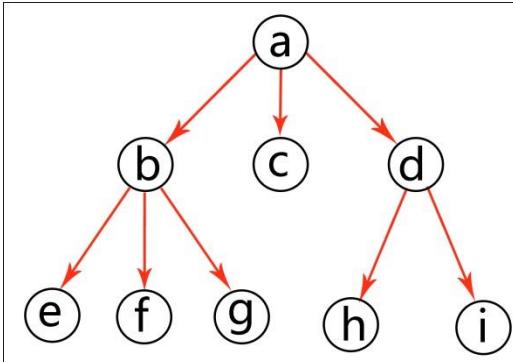
### ۱- استفاده از لیست پیوندی ( Linked List )

در این روش برای پیاده سازی از دو نوع گره استفاده می کنیم.

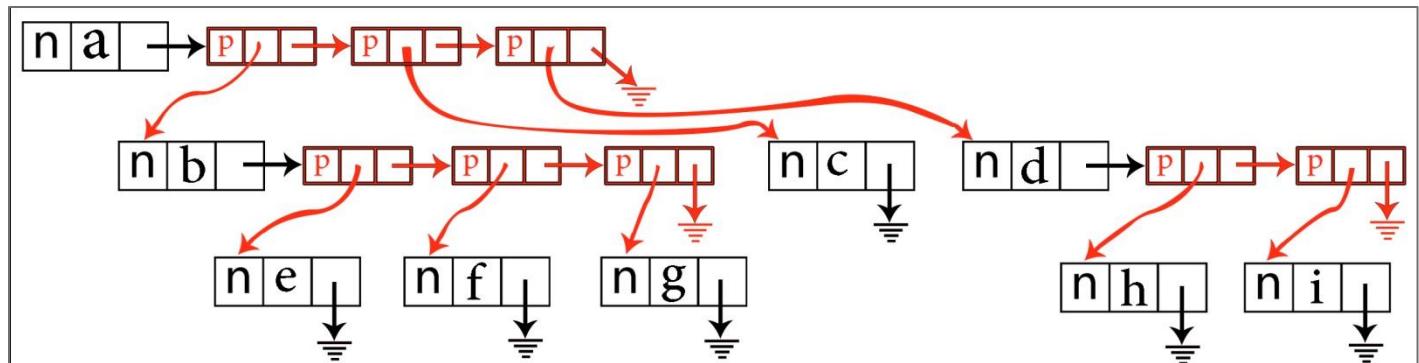
یک نوع متناظر با گره های درخت ( $n$ )



و یک نوع متناظر با یال ها ( $p$ )



مثلاً پیاده سازی درخت روبرو با این روش به شکل زیر است:

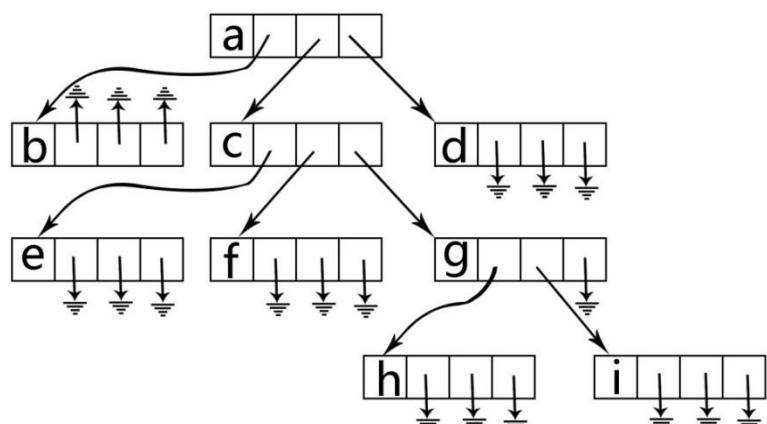
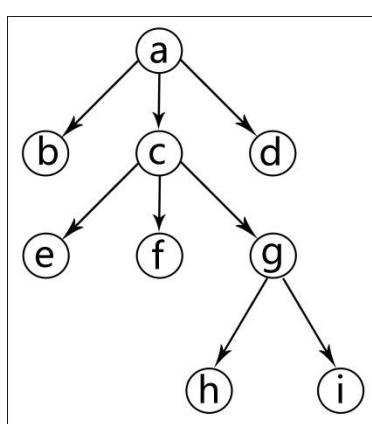


به تعداد  $E+V$  نود در لینک لیست داریم.

## ۲- استفاده از K اشاره گر برای درخت K-تایی:

در این روش هر کدام از گره‌ها دارای K اشاره گر برای اشاره کردن به فرزندان خود هستند.

مثلاً پیاده سازی درخت روبرو با این روش به شکل زیر است:



این روش دو مشکل اساسی دارد که عبارت اند از :

۱ - تلفات تعداد اشاره گرهای  $\text{NULL}$  زیاد است. زیرا:

تعداد یال ها - تعداد کل اشاره گرها = تعداد اشاره گرهای  $\text{NULL}$

$$\text{NULL} \times K - e = Kn - (n - 1) = Kn - n + 1$$

۲ - برای  $K$  محدودیت داریم یعنی اگر تعداد اشاره گرهای یک گرۀ بیشتر از  $K$  باشد نمی‌توان همه را ذخیره کرد.

مثال : فرض کنید درختی که میخواهیم ذخیره کنیم ، فولدرها در یک رایانه باشد. اگر از روش فوق استفاده کنیم، باید از قبل پیش بینی کنیم که قرار است هر فolder چند subfolder داشته باشد. که این خوب نیست و حداقل از لحاظ تئوری  $k$  بی نهایت است.

توجه: البته اگر  $k$  کوچیک باشد و حداکثر آن را بدانیم، این روش روش خوبی است.

(Leftmost Child-Right Sibling (LMC-RS))

### ۳- استفاده از درخت دودویی معادل

هر درختی قابلیت تبدیل به درخت دودویی معادل را دارد.

برای تبدیل درخت به درخت دودویی معادل به تعداد گره‌های درخت اصلی در درخت دودویی معادل گرۀ داریم، فرزند چپ هر گرۀ در درخت دودویی چپ ترین فرزند درخت اصلی است و فرزند راست در درخت دودویی همزاد راست گرۀ در درخت اصلی است.

مثال: الگوریتمی برای تبدیل یک درخت معمولی به درخت دودویی ارائه دهید.

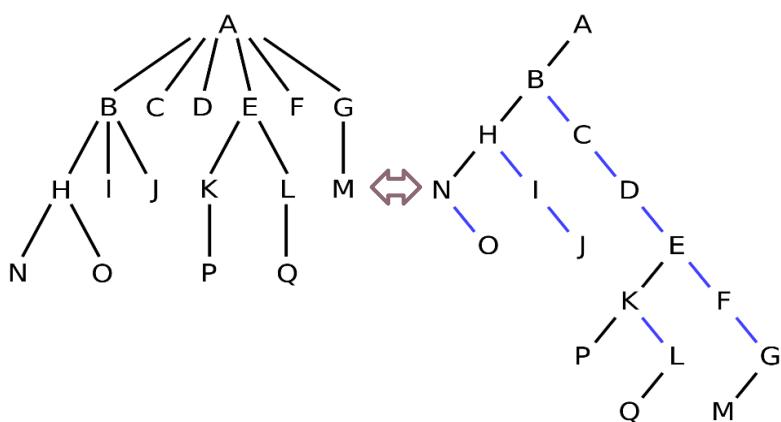
حل:

۱- از ریشه درخت شروع میکنیم(ریشه درخت معمولی ، ریشه درخت دودویی نیز هست).

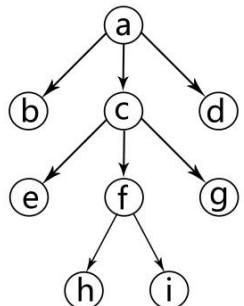
۲- اولین فرزند سمت چپ ریشه یعنی  $C_1$  را فرزند چپی ریشه در درخت دودویی قرار میدهیم و همزاد  $C_1$  را فرزند سمت راست  $C_1$  در درخت دودویی قرار میدهیم(اگر همزاد نداشت null قرار میدهیم) و ....

۳- گام دوم را برای هر گره جدید تکرار میکنیم

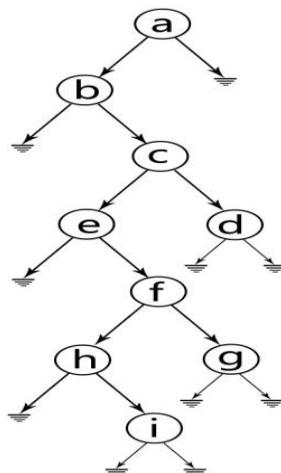
مثال: در شکل زیر مثالی از تبدیل درخت معمولی به درخت دودویی مشاهده میکنید.



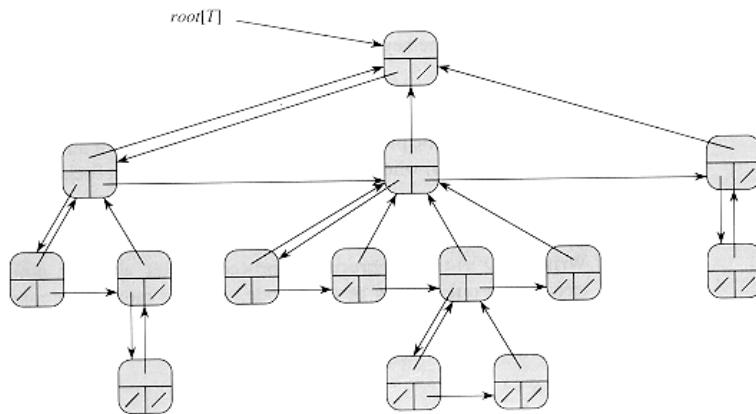
مثال : درخت رو برو را به درخت دودویی تبدیل کنید.



حل: پیاده سازی درخت دودویی معادل آن به صورت زیر است:



شکل زیر نشان میدهد چگونه میتوان یک درخت درجا به درخت باینری تبدیل کرد



توجه: با درخت دودویی میتوان جنگلی از درخت ها را نیز پیاده سازی کرد.

مثال : الگوریتمی با مرتبه  $O(n)$  پیشنهاد دهید که مقادیر تمام گره های یک درخت ریشه دار دلخواه را که

به صورت **LMC-RS** ذخیره شده است را چاپ کند.

حل: کد الگوریتم به زبان C به صورت زیر می باشد

```
#define MAX_SIZE 10

struct tree_t {
    struct tree_t *child;
    struct tree_t *sibling;
    struct tree_t *parent;
    int key;
};

typedef struct tree_t tree_t;

void store(int);

void print_tree(tree_t *tree) {
    store(tree->key);
```

```

if (tree->child)

    print_tree(tree->child);

if (tree->sibling)

    print_tree(tree->sibling);

}

int keys[MAX_SIZE];

int count = 0;

void reset_storage() {

    count = 0;

}

void store(int key) {

    keys[count++] = key;

}

```

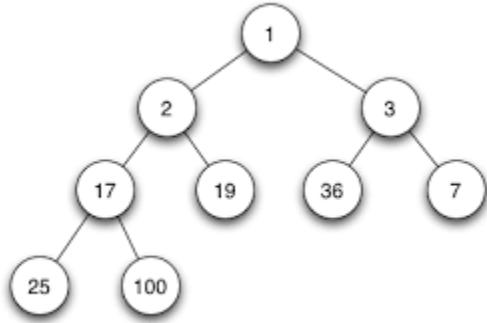
## ۴ – پیاده سازی درخت با استفاده از آرایه

برای این کار دو روش وجود دارد:

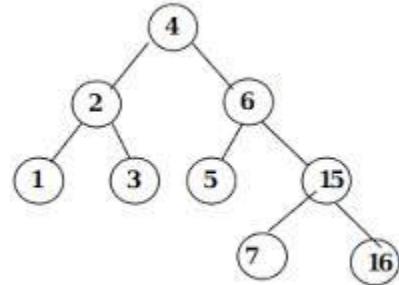
روش اول : در این روش از پیمایش سطحی(Level Order) استفاده میکنیم.

معمولا درخت های کاملا پُر را با این روش پیاده سازی می کنند.(در غیر این صورت اتلاف زیادی دارد)

درخت پر (Heap Tree) درخت متوازنی است که برگ های سمت چپ عمق بیشتری دارند.

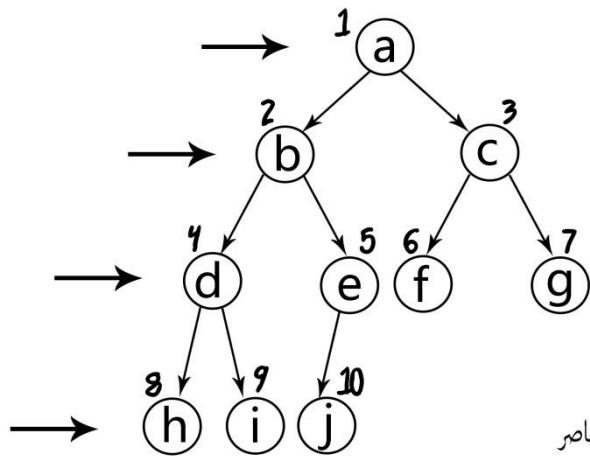


درخت پر



درخت پر نیست

درخت رو به رو یک درخت پر دوتایی است :



شماره های روی هر گره متناظر با ترتیب پیمایش سطحی (از چپ به راست) درخت است.

پیاده سازی این درخت با آرایه به شکل زیر است :

1	2	3	4	5	6	7	8	9	10
a	b	c	d	e	f	g	h	i	j

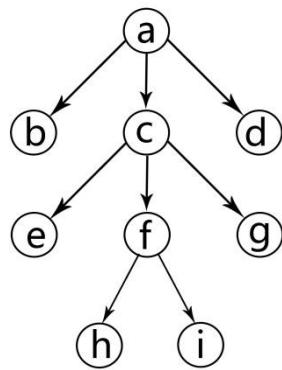
آرایه عناصر

در پیاده سازی درخت های پر با آرایه فقط به یک آرایه برای ذخیره ی عناصر نیاز داریم، زیرا با توجه به متوازن بودن درخت می توان با استفاده از اندیس فرزند اندیس پدر را محاسبه کرد.

برای مثال بالا اگر اندیس فرزند را  $A$  در نظر بگیریم اندیس پدر برابر است با :  $\left\lfloor \frac{A}{2} \right\rfloor$

روش دوم:

در این روش دو آرایه داریم، یکی برای ذخیره سازی داده‌ی داخل گره‌ها و یکی دیگر برای ذخیره سازی اندیس پدر هر گره.



مثلاً پیاده سازی درخت روبرو با این روش به صورت زیر است :

1	2	3	4	5	6	7	8	9
a	b	c	d	e	f	g	h	i
0	1	1	1	3	3	3	6	6

آرایه‌ی عناصر  
آرایه‌ی ذخیره‌ی  
اندیس پدر هر گره

در این روش ریشه در اندیس ۱ آرایه‌ی عناصر ذخیره می‌شود.

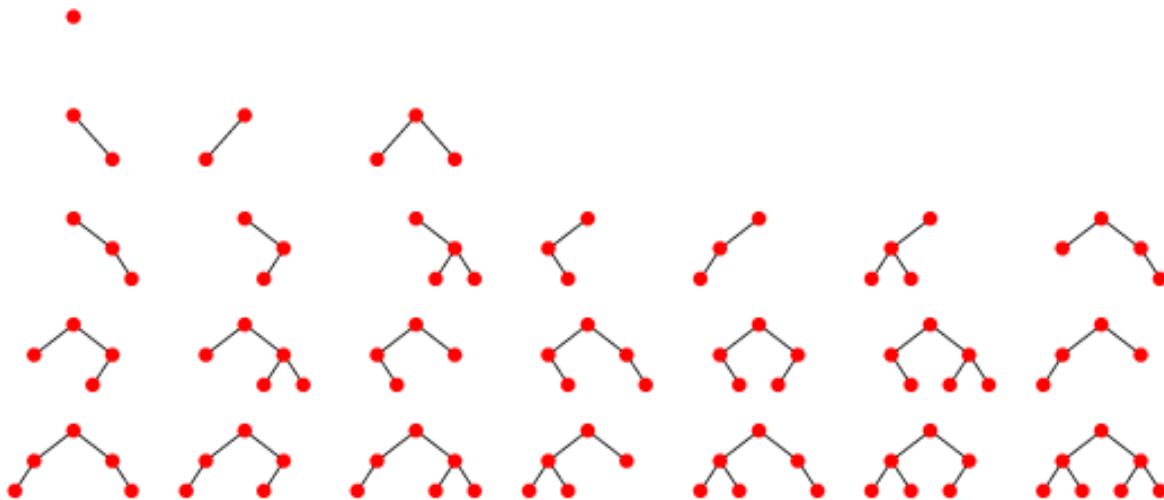
هر کدام از بیضی‌های قرمز معادل یک یال است.

این روش بیشتر برای ذخیره سازی درخت در فایل مناسب است و نه در حافظه‌ی اصلی.

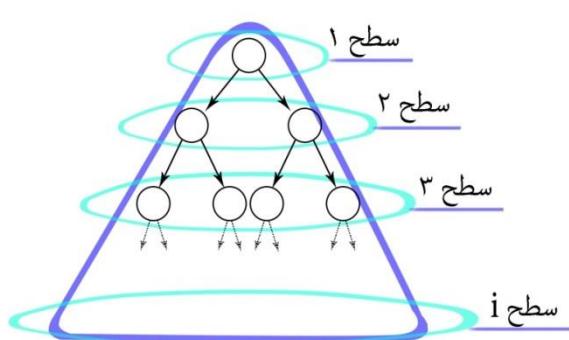
## ۴ - ۲ : درخت دودویی

تعریف: درخت دودویی، درختی است که در آن تعداد فرزندان هرگره حداقل ۲ باشد.

مثال: چند درخت دودویی با اندازه های مختلف



روابط ریاضی مربوط به درخت های دودویی:



۱- در درخت دودویی حداقل  $2^{i-1}$  سطح ام

است.

۲- اگر تعداد برگ ها را با  $n.$  و تعداد گره های با یک فرزند را با  $n_1$  و تعداد گره های با دو فرزند را با  $n_2$  نشان دهیم داریم:

اثبات:

بر اساس رابطه ای اویلر داریم :

$$\begin{aligned} e &= n - 1 \\ n &= n_+ + n_1 + n_- \\ e &= 2n_2 + n_1 \end{aligned} \quad \left. \begin{array}{c} \\ \\ \end{array} \right\} \Rightarrow n_2 + 1 = n.$$

### پیاده سازی درخت های دودویی:

برای پیاده سازی درخت دودویی از linked-list استفاده می کنیم، برای این کار هر گره را به صورت زیر تعریف می کنیم :

```
struct Tree{
    DataType data;
    Tree* left;
    Tree* right;
}
```

هر گره یک فیلد `data` دارد که اطلاعات مربوط به آن گره را نگه داری می کند و دو اشاره گر به فرزندان چپ و راست دارد که به ترتیب ریشه های زیر درخت چپ و راست هستند. میتوان برای تکمیل اشاره گری برای پدر هر گره در نظر گرفت.

برای مثال اگر بخواهیم درخت دودویی را به صورت پیش ترتیب پیمایش کنیم به صورت زیر خواهد بود.

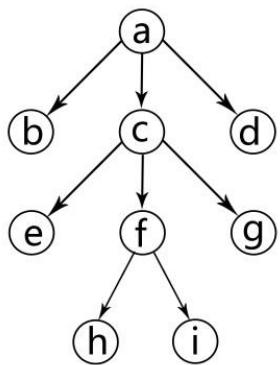
```
Preorder(T,r){
    If(r==NULL)
        return
    else
        Preorder(T,r.left)
        Preorder(T,r.right)
}
```

## تبدیل درخت های دلخواه به درخت های دودویی:

قضیه) هر درخت دلخواه را می توان به درخت دودویی تبدیل کرد.

برای تبدیل درخت به درخت دودویی معادل به تعداد گره های درخت اصلی در درخت دودویی معادل گره داریم، فرزند چپ هر گره در درخت دودویی چپ ترین فرزند درخت اصلی است و فرزند راست در درخت دودویی همزاد راست گره در درخت اصلی است.

مثال :

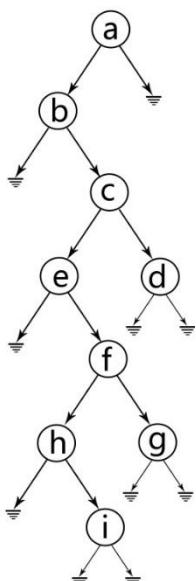


درخت روبه رو را در نظر بگیرید :

Pre order: a b c e f h i g d

In order : b c e f h f i g d a

Post order : b e h i f g c d a



پیاده سازی درخت دودویی معادل آن به این صورت است :

Pre order : a b c e f h i g d

In order: b c e f h i g d a

Post order: i h g f e d c b a

اگر پیمایش های مختلف دو درخت را بنویسیم به نتایج زیر میرسیم:

. پیمایش های پیش ترتیب و میان ترتیب درخت اصلی و درخت دودویی معادل است.

. پیمایش پس ترتیب درخت اصلی با درخت دودویی معادل یکسان نیست.

## درخت عبارت:

درخت عبارت، یکی از کاربردهای درخت برای محاسبه عبارت‌های مختلف می‌باشد. دو نوع معمول از عبارت‌هایی که این درخت محاسبه می‌کند عبارت‌های جبری و بولی است.

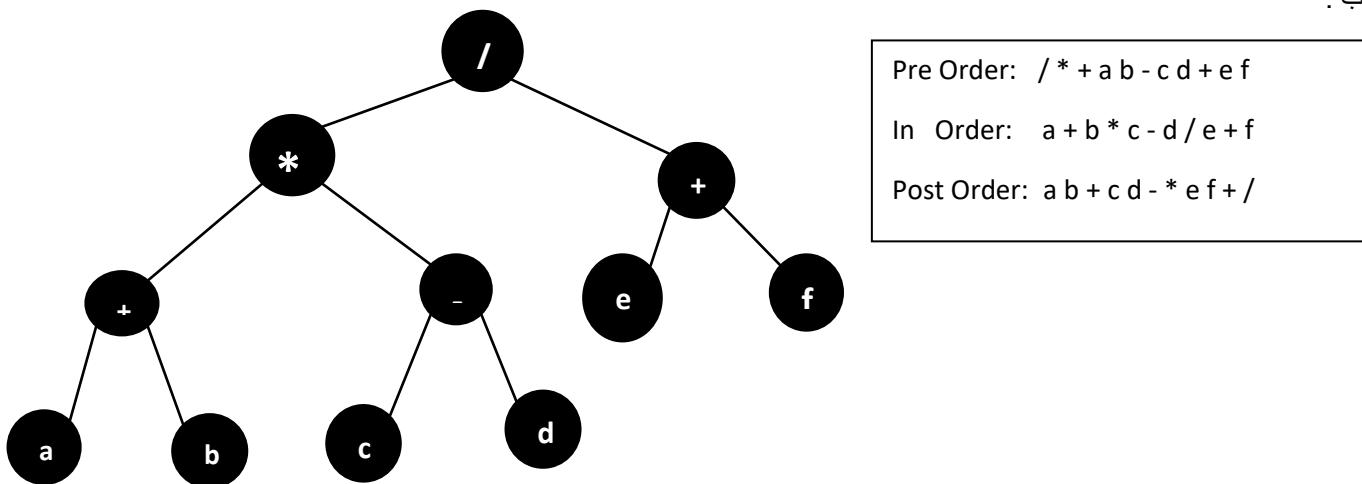
برگ‌های درخت عبارت عملوند‌ها (operands) هستند مثلاً اعداد ثابت یا نام متغیرها، اما سایر گره‌های درخت عملگرها (operators) می‌باشند. اما نکته‌ی مهم دلیل این است که درخت‌های عبارت اکثراً دودویی هستند. علت آن این است که اکثر عملگرها یا دودویی (binary) هستند یا یگانی (unary) و اگر عملگر دودویی باشد (مثلاً جمع، تفریق و ...) برای آن گره ۲ فرزند لازم است و اگر عملگر یگانی باشد (مثلاً نقیض، لگاریتم و ...)، برای آن گره یک فرزند لازم است.

## پیمایش درخت عبارت:

از آنجایی که درخت عبارت هم نوعی درخت است برای آن روش‌های پیمایش درخت (پیش ترتیب - میان ترتیب - پس ترتیب - عمق ترتیب و ...) عیناً اجرا می‌شود.

مثال : پیمایش‌های پیش، پس و میان ترتیب درخت مقابله را بنویسید.

جواب :



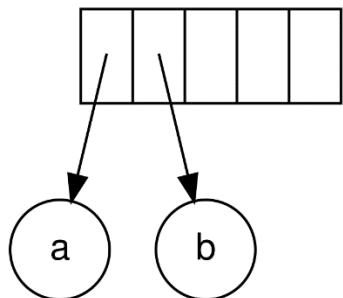
## ساخت درخت عبارت:

ساخت درخت عبارت با استفاده از پیمایش پس ترتیب و استفاده از پشته انجام میگیرد. به این ترتیب که عبارت پس ترتیب را میخوانیم (اگر پیش یا میان ترتیب بود به پس ترتیب تبدیل میکنیم). سپس اگر هر عنصر عملوند بود یک گره با آن نام ساخته و به پشته اضافه میکنیم ولی اگر عملگر بود، بسته به نوع آن (دو دویی، یگانی یا...) تعدادی عنصر از پشته برداشته و به عنوان فرزندان آن گره انتخاب میکنیم و این کار را تکرار میکنیم.

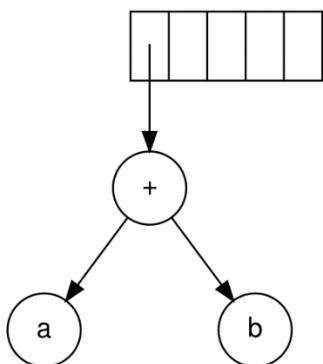
مثال: درخت عبارت  $a b + c d e + *$  را بسازید

(جواب)

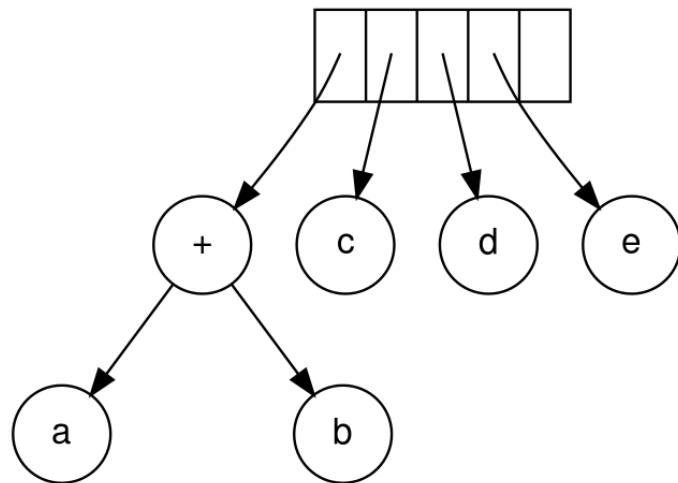
۱- دو حرف اول عملوند هستند پس  $a$  و  $b$  را به پشته اضافه میکنیم.



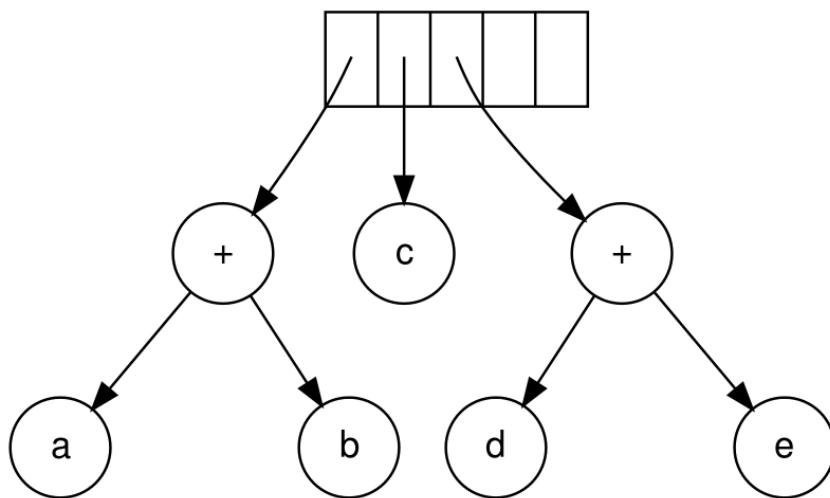
۲- حرف بعدی  $+$  است. گره ای با این نام ایجاد کرده و چون عملگر دو دویی است دو عنصر از پشته که در حال حاضر  $a$  و  $b$  است را برداشته و به عنوان فرزندان چپ و راست این گره در نظر میگیریم و این گره را به پشته اضافه میکنیم.



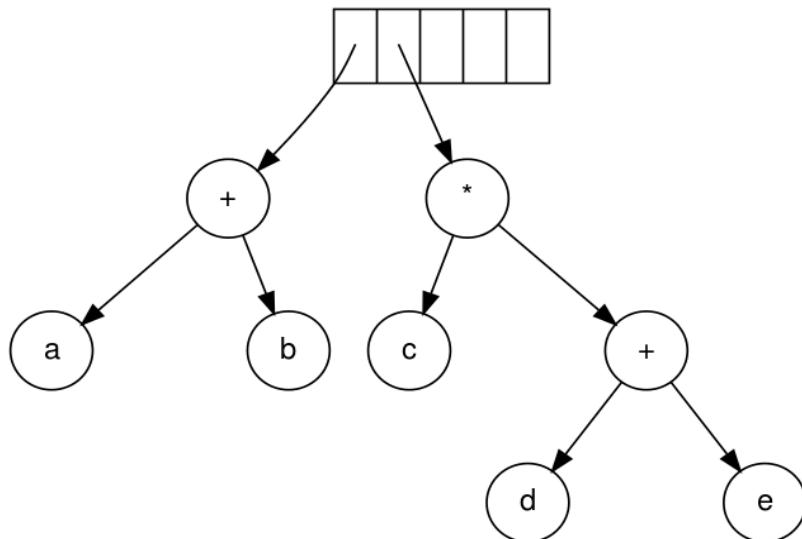
۳- سه حرف بعدی، سه عملوند هستند و به ترتیب به پشته اضافه میشوند.



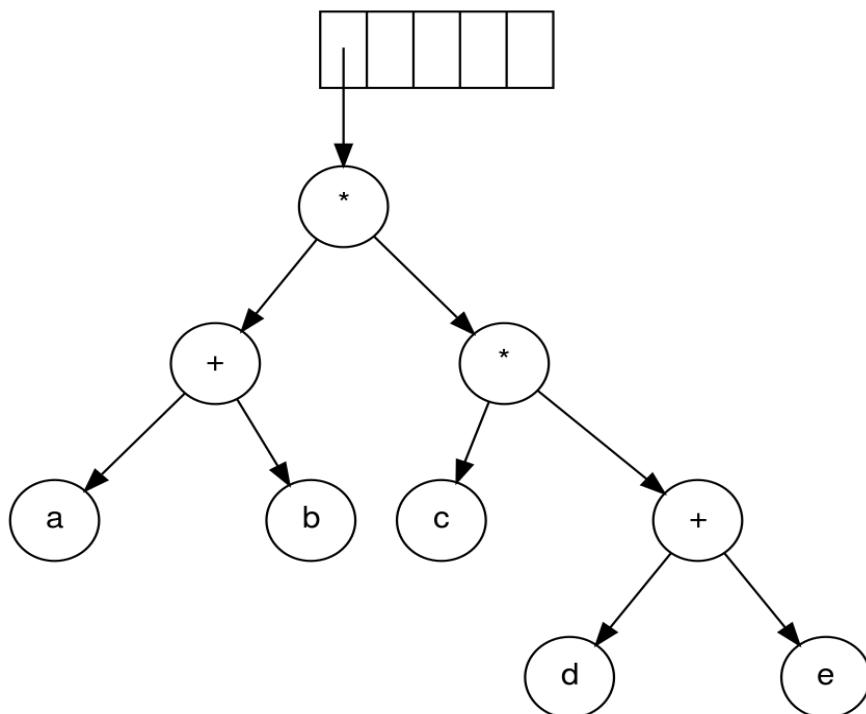
۴- حرف بعدی عملگر + است و چون دودویی است پس دو عنصر آخر پشته که e و d است را برداشته و مانند مرحله ۲ عمل میکنیم.



۶- حرف بعدی عملگر دودویی '\*' است پس دو عنصر را از پشته برداشته و به جای آن عنصر جدیدی اضافه میکنیم.



۷- در نهایت عنصر آخر که باز هم عملگر دودویی '\*' است را خوانده و گره ای با این اسم ایجاد میکنیم و دو عنصر آخر پشته که '\*' و '+' است را استخراج کرده و به عنوان فرزندان چپ و راست این عنصر اضافه میکنیم و در نهایت شکل اینگونه میشود:



# اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

## ۱- درخت هافمن

### ۱-۱- فشرده سازی

در علم کامپیوتر و فناوری اطلاعات، مفهومی با عنوان فشرده سازی اطلاعات (data compression) وجود دارد که به مفهوم کدگذاری اطلاعات با استفاده از تعداد بیت‌های کمتر نسبت به نسخه اصلی است. متدهای فشرده سازی به دو دسته تقسیم می‌شوند:

- فشرده سازی با اتلاف (Lossy)
- فشرده سازی بدون اتلاف (Loss Less)

#### ۱-۱-۱- کد گذاری با اتلاف :

کدگذاری با اتلاف حجم بیت‌ها را با تشخیص اطلاعات غیر ضروری و حذف آنها، کاهش می‌دهد؛ به طور کلی پروسه کاهش سایز یک داده به عنوان عمل فشرده سازی شناخته می‌شود.

حذف جزئیات غیر ضروری می‌تواند حجم اشغال شده به وسیله داده را کاهش دهد. این نوع فشرده سازی از نحوه دریافت اطلاعات توسط افراد الهام می‌گیرد. برای مثال یکی از چیزهایی که هرروزه با آنها در ارتباطیم، عکس‌ها با فورمات JPEG هستند. در این نوع فشرده سازی، جزئیاتی از عکس که برای چشم ما نامحسوس ترند حذف می‌شود. در نتیجه از وضوح عکس کاسته می‌شود؛ از طرفی این موضوع کمک بزرگی به کاهش سایز عکس می‌کند.

بسیاری از Data Type‌های مطرح و پر کاربرد این روزها، از نوع Lossy هستند. از فورمات‌های ویدیوئی و عکس‌ها گرفته تا موسیقی (نظریه ام پی تری).

#### ۱-۱-۲- کد گذاری بدون اتلاف :

در نظریه اطلاعات، همواره حجمی از بیت‌های بسته آمده حاوی Redundancy است. عبارت Redundancy است از تفاضل تعداد کل بیت‌های استفاده شده باری انتقال یک پیام (Data) و تعداد بیت‌های پیغام که از ابتدا مطلوب کاربر بوده است.

فشرده سازی، در عمل Redundancy را که در اصل اطلاعات ناخواسته محسوب می شود، کاهش داده یا از بین می برد. نکته قابل توجه اینکه برخلاف حالت قبل، این روش فشرده سازی قابل بازگشت بوده به طوری که اطلاعات اولیه قابل بازیابی به صورت کامل هستند.

در دنیای واقعی، مثال های زیادی از این روش در دسترس است؛ برای مثال در فشرده سازی یک عکس، اگر تمام نواحی تصویر که رنگ یکسانی دارند، یعنی پیکسل های تصویر در یک ناحیه، بدون تغییر بماند، بجای عبارت هایی نظری:

"Red Pixel, RedPixel, Red..."

خواهیم نوشت: "۲۷۸ Red Pixels".

روش ها و مثال های دیگری از این دست وجود دارند. یکی از آنها روش کدگذاری هافمن است.

## ۱-۲ - ایده هافمن

در کامپیوترها، ما تنها با اعداد در مبنای ۲ سروکار داریم؛ مشخص است که برای استفاده از یک عدد، کافی است آن را به مبنای ۲ ببریم. اما در مورد حروف چطور؟ کلمات و یا دستورات چگونه شکل می گیرند؟ برای حل این مشکل، از کدگذاری استفاده می کنیم (Encoding). به این شکل که به هر حرف، یک عدد نسبت می دهیم. برای اینکار، ۲ روش وجود دارد.

- کدگذاری با اندازه ثابت (fix length encoding) : در روش اول که کدگذاری با طول ثابت نام دارد، تعداد بیت‌های مشخصی را در نظر گرفته و هر کاراکتری را به یک عدد نسبت میدهیم. یعنی طول هر ۲ کاراکتر دلخواه همان مقدار ثابت (مثلاً ۸ بیت) است. واضح است که تعداد کاراکترهایی که این روش پوشش میدهد،  $2^8 = 256$  است. در کدگذاری ASCII<sup>۱</sup>، طول بیت‌های استفاده شده برای هر کاراکتر مشخص و ثابت است.

نکته خیلی مهمی در اینجا وجود دارد که باید به آن توجه کرد: یک متن را در نظر بگیرید. در یک فایل که حاوی تعداد زیادی کلمه است، احتمال وجود حرف های صدادار و حرف های خاص پر کاربرد، خیلی بیشتر از تعداد حرف های دیگر است. مثلاً در زبان فارسی، حروفی مانند 'ا'، 'ی'، 'ه' و... پرکاربردتر از حرفهایی مانند 'ظ' است.<sup>۲</sup> با در نظر گرفتن این موضوع، به روش دوم و ایده کدگذاری هافمن می رسیم. روش دوم، کدگذاری با طول متغیر است.

تعريف ۱. فرکانس : عبارت است از تعداد تکرار یک کاراکتر مشخص در یک رشته از اعداد (پاراگراف، متن و...).

ایده فشرده سازی هافمن، با توجه به قانون پارت، کدگذاری متغیر است. یعنی، به کاراکترهایی که بیشترین فرکانس را دارند، کدهایی با کمترین طول ممکن نسبت دهیم و به کاراکترهای کم تکرار تر، طول بیشتر.

<sup>۱</sup> American Standard Code for Information Interchange

Pareto Principle.<sup>۲</sup> که همچین به نام های قانون ۸۰-۲۰، قانون افراد اندک اساسی و اصل تُنکی فاکتور شناخته می شود، بیان می کند که ۸۰ درصد رخداد ها از ۲۰ درصد دلایل بوجود می آید. مثالهای زیادی از این قانون در اطراف ما وجود دارد. مثلاً "۸۰ درصد خرید مربوط به ۲۰ درصد مشتریان است." و یا "۸۰ درصد ثروت جهان در انحصار ۲۰ درصد جمعیت آن است." در مثالی که زده شد، میتوان گفت حدود ۸۰ درصد متن از ۲۰ درصد کاراکترهای موجود در زبان تشکیل شده است.

## اخطار : محتویات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

تعریف ۲. خاصیت پیشوندی : در کدهایی که برای هر کاراکتر انتخاب می‌کنیم، باید دقت کنیم یک کد، پیشوند کد دیگر نباشد. مثلاً نمی‌توانیم برای کاراکتری، کد ۱۰۰ را انتخاب کنیم و برای کاراکتر دیگری کد ۱۰۰۰.

- اگر کدهایی که انتخاب می‌کنیم خاصیت پیوندی نداشته باشند در این صورت اگر کد یک کاراکتر ۱۰۰۰ و کد یک کاراکتر دیگر ۱۰۰۰ باشد وقتی به عدد ۱۰۰۰۱ برخورد می‌کنیم نمی‌توانیم تشخیص دهیم که این عدد حاصل ۱۰۰۰+۱ است یا که کل آن یک کاراکتر است .

به جدول زیر دقت کنید:

Char	Fixed-length	Variable-length	Frequency
A	...	.	۵۰
B	..۱	۱۱۰	۱۰
C	.۱۰	۱۰	۲۰
D	.۱۱	۱۱۱۰	۱۰
E	۱۰۰	۱۱۱۱۰	۴
F	۱۰۱	۱۱۱۱۱۰	۴
G	۱۱۰	۱۱۱۱۱۱۰	۱
H	۱۱۱	۱۱۱۱۱۱۱	۱

جدول ۱

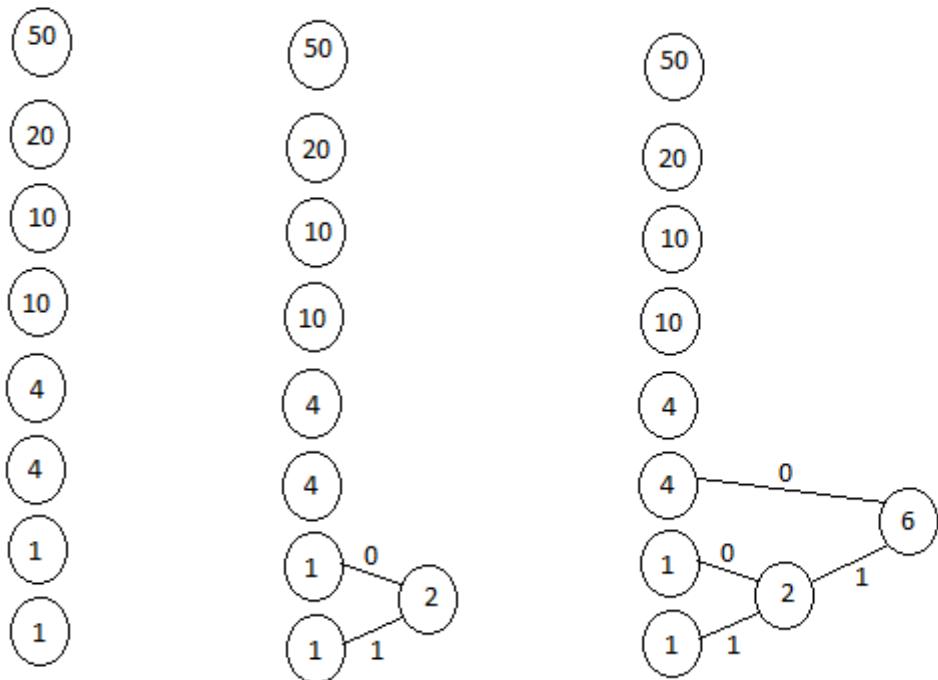
کدهای انتخاب شده برای کاراکترهای جدول ۱، از نوع متغیر، دارای خاصیت پیشوندی هستند. حال، با دانستن ایده‌ی کدگذاری، نیاز به روشی برای بدست آوردن کدها داریم.

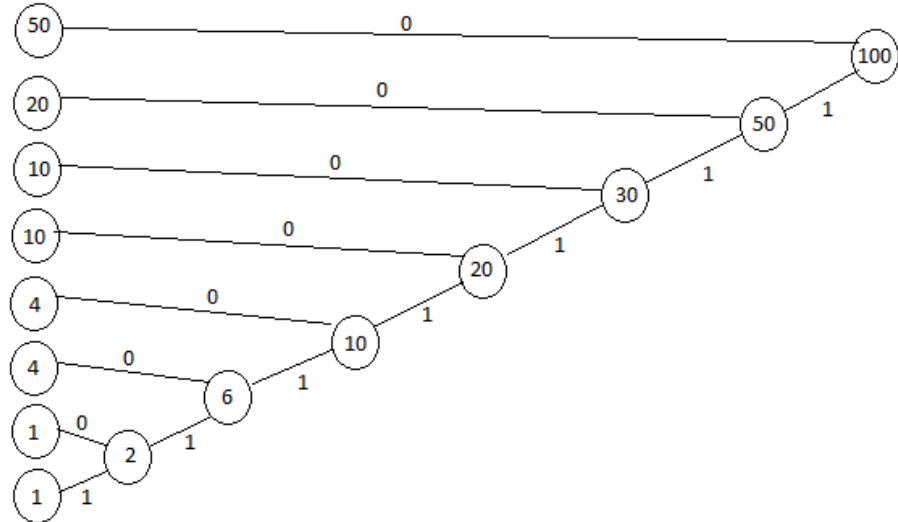
## -۳-۱ درخت هافمن

تعريف : درخت هافمن یک درخت دودویی است که برای کد گذاری با طول متغیر در فشرده سازی، به کار می رود.  
برخلاف معمول که برای ساخت درخت از ریشه شروع می کردیم، در اینجا ابتدا برگها را قرار می دهیم. برگهای درخت هافمن، در واقع همان کاراکترها و فرکانس تکرارشان است. بهتر است کاراکترها را به ترتیب نزولی یا صعودی فرکانسشان قرار دهیم. سپس، هر بار، فرکانس دو کاراکتری را که جمعشان از جمع فرکانس هر دو تای دیگری کمتر بود، به یک گرهی جدید تبدیل میکنیم به صورتی که عدد گرهی مورد نظر، برابر مجموع دو فرزندش باشد. چون درخت دودویی است (یعنی هر گره حداقل دو فرزند دارد)، به هر فرزند هر گره یک عدد ۰ یا ۱ نسبت می دهیم. این کار را تا زمانی که به آخرین گره برسیم بهتر، فرکانس ریشه درخت، برابر تعداد کل کنید فایلی داریم که تکرار حروف در آن به شکل

Char	Freq	(ریشه درخت) متوقف نمیکنیم. به عبارت کاراکترهایی است که در کل داریم. فرض زیر است :
A	۵۰	
B	۲۰	
C	۱۰	
D	۱۰	
E	۴	
F	۴	
G	۱	
H	۱	

**اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند**





Char	Code
A	.
B	10
C	110
D	1110
E	11110
F	111110
G	1111110
H	1111111

شکل 1

کدهایی که بر روی یال های درخت  
هستند. برای خواندن کدها، کافی است از  
راسی که حاوی کاراکتر مورد نظر است  
راست همان کد هافمن کاراکتر است. جواب

بدست آمده نوشته شده اند، همان کد های هافمن  
ریشه شروع کرده، مسیری را انتخاب کنیم که به  
بررسی بیت های خوانده شده از سمت چپ به  
نهایی به شکل زیر است :

## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

### ۱-۳-۱-نکات :

۱.ENCODING به روش هافمن برای هر فایل با فایل‌های دیگر متفاوت است و به فرکانس هر کاراکتر در کل فایل بستگی دارد.

۲.الگوریتم هافمن در جایی که توضیع فرکانس نسبتاً یکنواخت باشد، خوب عمل نمی‌کند. (درخت به درخت دودویی متقارن نزدیک شده و در نتیجه طول کاراکترها به یک مقدار ثابت میل می‌کند). - شکل ۵  
در درخت هافمن، کاراکترها با تعداد تکرار بیشتر، در عمق کمتری قرار دارند.

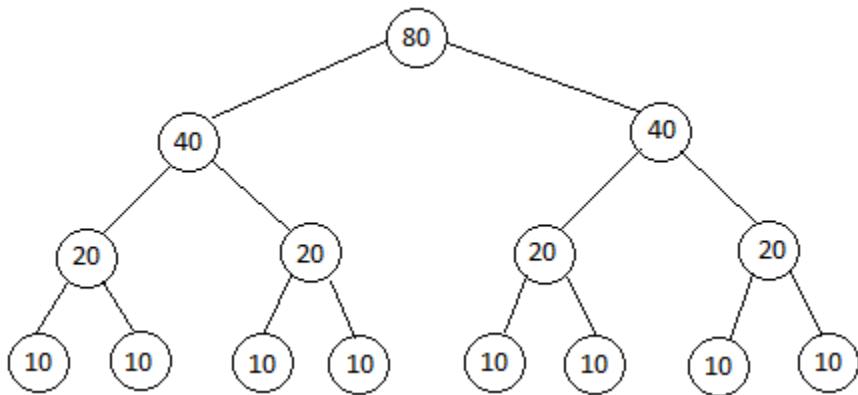
۳. برای محاسبهٔ حجم یک فایل، کافیست فرکانس کاراکتر‌های آن را در طول هر کاراکتر ضرب کنیم و نتایج را بهم جمع؛ مقدار به دست آمده، حجم فایل بر حسب BIT است. برای مثال، در جدول بالا به ازای کد‌های با طول ثابت، حجم فایل  $= 300 \times 3 = 900$  است. در حالی که اگر کد هافمن را استفاده کنیم، حجم فایل از راه زیر به دست می‌آید:

$$50 * 1 + 20 * 2 + 10 * 3 + 10 * 4 + 4 * 5 + 4 * 6 + 1 * 7 + 1 * 7 = 218 \text{ bit}$$

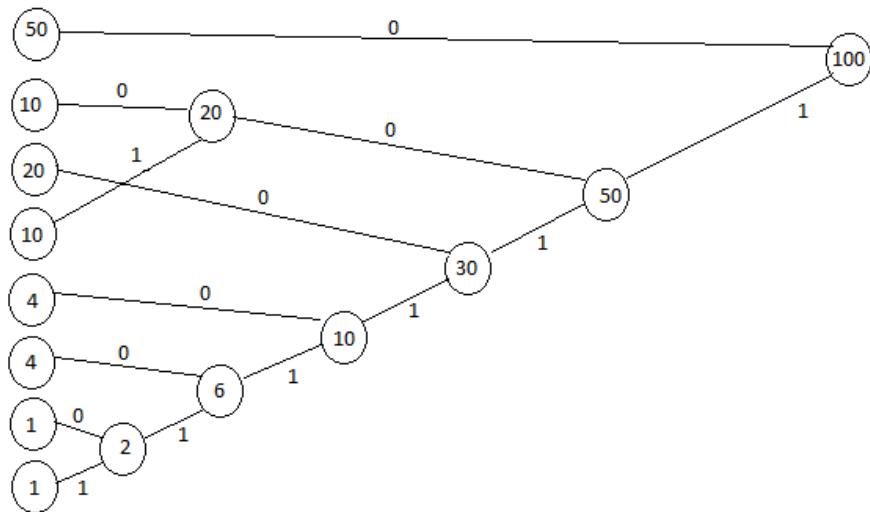
۴. پیشتر گفته شد ترتیب چیدمان برگ‌ها بهتر است صعودی یا نزولی باشد. علت این امر فقط یک دست شدن درخت است. و گرنه از چیدمان‌های مختلف برگ‌ها، ممکن است کد‌های مختلفی بدست آید اما در نهایت، حجم فایل هیچ‌گاه از ۲۱۸ کمتر و یا بیشتر نمی‌شود. به عبارت دیگر، با اینکه درخت هافمن برای یک مسئلهٔ خاص لزوماً یکتا نیست، اما در مسائل فشرده سازی، مقدار حجم نهایی همواره یکتاست. - شکل ۶

۵. یکی از ایرادات درخت هافمن اینست که فایل‌هایی که با این روش فشرده شده‌اند باید یک جدول که در آن کد هر کاراکتر وجود دارد در کنار خود ذخیره کنند که این کار خود به حجم فایل می‌افزاید.

۶. یک ایراد دیگر هافمن این است که باید متن دو بار خوانده شود یعنی یک بار باید فایل خوانده شود و از یک جدول که کد هر کاراکتر را ذخیره می‌کند فایل را بازنویسی کرد بعد می‌توان از آن فایل استفاده کرد.



شکل ۲



شکل ۳

## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

### -۱-۱ درخت Heap

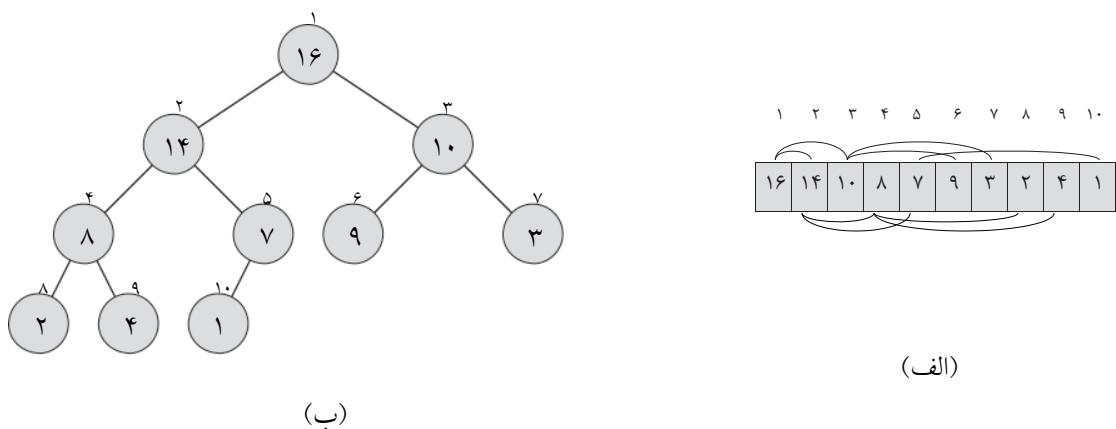
Heap نوعی درخت است که در آن فقط ارتباط میان پدران و فرزندان با هم اهمیت دارد و همزاد‌ها هیچگونه ارتباط خاصی با هم ندارند در ادامه با ویژگی‌های Heap آشنا می‌شویم.

ساختمان داده Heap (دوتایی) یک آرایه است (در پیاده سازی درخت Heap توسط آرایه هیچ خانه‌ای از آرایه خالی نمی‌ماند) که میتواند بصورت یک درخت دودویی تقریباً کامل دیده شود، همانطور که در شکل ۲ نشان داده شده است.

هرگره درخت به یک عنصر از آرایه ای که مقادیر گره‌ها را نگهداری می‌کند، اشاره می‌کند. یک آرایه مرتب همیشه میتواند یک آرایه Heap باشد اما برعکس آن لزوماً برقرار نیست مانند شکل ۲

درخت به طور کامل در همه سطح‌ها پر می‌شود (متوازن است) مگر در آخرین سطح که ممکن است به صورت کامل پر نشود و در آن از سمت چپ پر می‌شود در نتیجه درخت‌های Heap با تعداد مشخص گره دارای شکل یکسان هستند.

درخت Heap دودویی به دو نوع Min-Heap و Max-Heap تقسیم می‌شود. در heap ارزش هرگره از فرزندانش کمتر / بیشتر و ریشه maximum / minimum است.



شکل ۱ - یک max-heap که بصور تاریخی ای که ریشه در اندیس ۱ قرار دارد (شکل الف) و درخت دودویی با ارتفاع سه (شکل ب) نشان داده شده است. عددی که در هر دایره گره درخت است مقدار ذخیره شده در آن گره است و عددی که بالای هر دایره است نمایانگر اندیس مربوط به مقدار آن گره در آرایه است. خط های نشان داده شده بالا و پایین آرایه شکل (الف) رابطه پدر فرزندی را نشان می دهند. پدر ها همیشه سمت چپ فرزندانشان هستند.

رابطه پدر فرزندی در آرایه ای که بیانگر درخت heap است و مقادیر گره های درخت در آن ذخیره شده اند با در نظر گرفتن " $i$ " به عنوان اندیس آرایه به صورت زیر است:

PARENT( $i$ )

return  $[ i/2 ]$

LEFT( $i$ )

return  $2i$

RIGHT( $i$ )

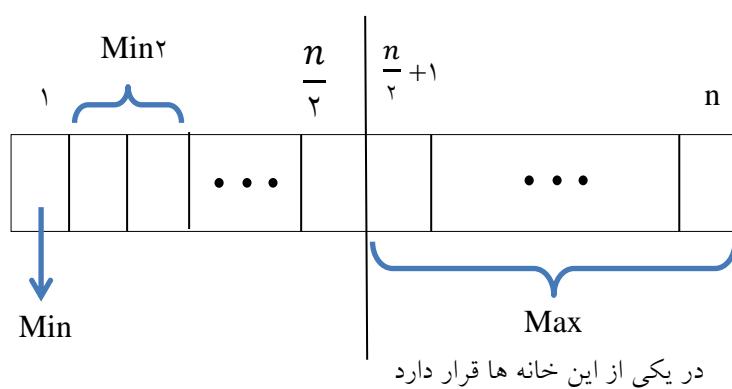
return  $2i+1$

در درخت Heap ، ارتفاع هر گره برابر با تعداد یال ها در بلندترین مسیر رو به پایین از آن گره به برگ و ارتفاع درخت برابر با ارتفاع ریشه است.

از آنجا که  $n$  عضو یک درخت دودویی کامل است ارتفاع آن  $\theta(\log n)$  است.

شكل Heap برای  $n$  عدد همیشه یکسان است.

**نکته:** پیاده سازی درخت توسط آرایه (ریشه در اندیس ۱ آرایه قرار دارد) :



## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

شکل ۲ - درخت Min-Heap ، پیدا کردن مینیمم ها و ماکزیمم درخت در آرایه  $\text{Min}^3$  (سومین کوچکترین مقدار در درخت) در یکی از اندیس های ۲ تا ۷ وجود دارد، یعنی شش حالت دارد. اگر  $\text{Min}^2$  مشخص باشد برای  $\text{Min}^3$  سه حالت وجود دارد.

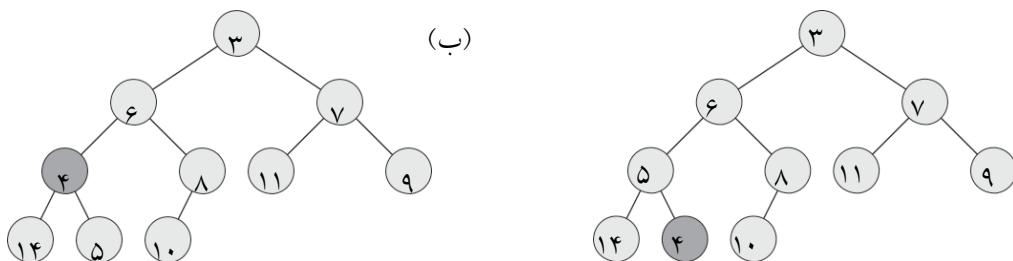
### ۱-۱-۱- تعريف :Heapify

يعنى در درخت Max-Heapify / Min-Heapify کاري کييم که ارزش هر گره از فرزندانش کمتر / بيشتر باشد. (خواص Binary Heap حفظ شود و درخت همواره باقی بماند) اين عمل به دو نوع زير تقسيم مى شود:

با پدرس مقاييسه ميكنيم اگر کوچکتر / بزرگتر بود آن ها را با هم Swap (جابجا)ميكنيم. شکل ۴

به منظور اين کار در درخت Max-Heap / Min-Heap با شروع از برگ ها، هر گره را با فرزندانش مقاييسه ميكنيم اگر از يكی از فرزندانش کوچکتر / بزرگتر بود آن ها را با هم Swap (جابجا)ميكنيم و اگر از هر دو فرزندش کوچکتر / بزرگتر بود آن را با کوچکترین / بزرگترین فرزندش جابجا ميكنيم.

(الف)



۳

۴

۷

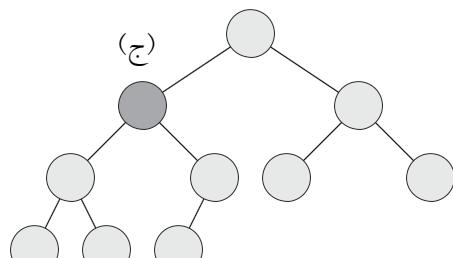
۶

۱۱

۹

۵

۱۰



شکل ۳ - BubbleUp ← Min-Heapify

پدرش (گره با مقدار ۵) کوچکتر است که این بر خلاف ویژگی های درخت Min-Heap (ارزش هر گره از فرزندانش کمتر است) می باشد. در نتیجه آن را با پدرش جابجا میکنیم و درخت قسمت (ب) حاصل میشود. حال در این شکل دوباره مشاهده میکنید که گره با مقدار ۴ از پدرش کوچکتر است پس آن ها را جابجا کرده و شکل قسمت (ج) حاصل میشود. شکل قسمت (ج) یک درخت Min-Heap است.

کد Max Heap برای Heapify به شکل زیر است:

همان طور که میبینید این کار به صورت بازگشتی ادامه پیدا میکند.

**MAX-HEAPIFY(A, i, n)**

```

 $l \leftarrow \text{LEFT}(i)$ 
 $r \leftarrow \text{RIGHT}(i)$ 
if  $l \leq n$  and  $A[l] > A[i]$ 
then largest  $\leftarrow l$ 
else largest  $\leftarrow i$ 
if  $r \leq n$  and  $A[r] > A[\text{largest}]$ 
then largest  $\leftarrow r$ 
if largest  $= i$ 
then exchange  $A[i] \leftrightarrow A[\text{largest}]$ 

```

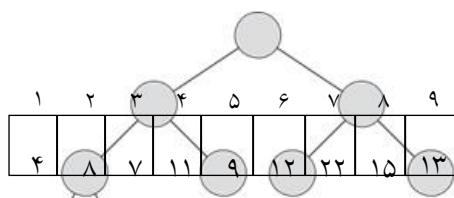
**MAX-HEAPIFY(A, largest, n)**

## ۱-۲-۱- اضافه کردن یک گره به درخت Heap : (Insert Heap)

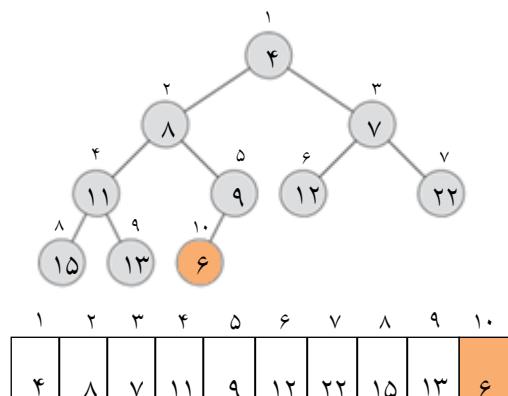
در اضافه کردن یک گره به درخت دودویی Heap همیشه این خاصیت درخت Heap که همیشه برگ ها از چپ به راست پر میشوند باید حفظ شود. پس درنتیجه برای این کار همیشه تنها یک جا برای اضافه کردن گره وجود دارد. پس از اضافه کردن گره به درخت آن را Heapify میکنیم تا درخت Heap کماکان درخت باقی بماند. هزینه آن به علت انجام عمل Heapify حداقل به اندازه ارتفاع درخت، برابر با ارتفاع آن یعنی  $O(\log n)$  است. به مثال زیر توجه کنید:



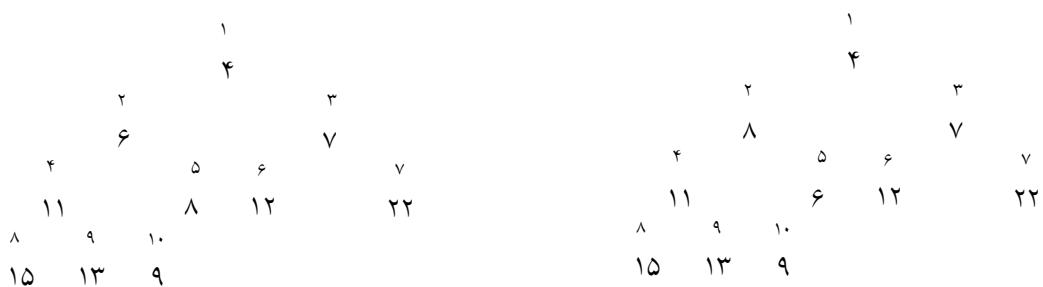
## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

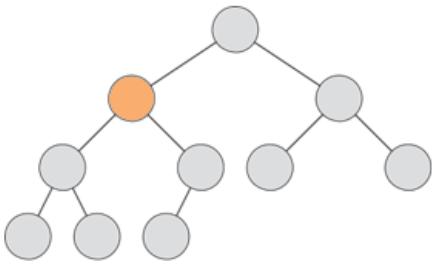


شکل ۴ - در درخت Min-Heap بالا میخواهیم یک گره با مقدار ۶ را به آن اضافه کنیم. با توجه به بودن این درخت چون باید برگ‌ها از چپ به راست پر شوند تنها جای ممکن برای این گره فرزند چپ گره با ارزش ۹ (اندیس ۵) است. پس آن را به درخت اضافه میکنیم. همچنین در آرایه متناظر با درخت همواره گره اضافه شده به درخت به آخر آرایه اضافه میشود. یعنی در اینجا گره اضافه شده در آرایه ۱۰ قرار خواهد گرفت.



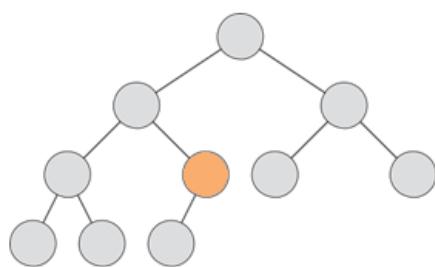
شکل ۵ - حال در این درخت که گره با مقدار ۶ اضافه شده است (در درخت به عنوان آخرین برگ و در آرایه بعد از آخرین خانه آرایه در اندیس ۱۰). مشاهده می‌شود که گره اضافه شده از پدرس کوچکتر است که این برخلاف Min-Heap بودن درخت است. در نتیجه لازم است که با Heapify کردن گره اضافه شده درخت را دوباره به Min-Heap تبدیل کنیم.





۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
۴		۶	۷	۱۱		۸	۱۲	۲۲	۱۵

(ب)



۱	۲	۳	۴	۵	۶	۷	۸	۹	۱۰
۴		۸	۷	۱۱		۶	۱۲	۲۲	۱۵

(الف)

شکل ۶ در شکل قسمت (ب) مشاهده می شود که پس از انجام Heapify(Bubble Up) روی درخت شکل ۶ و سپس روی درخت قسمت (الف)، درخت اولیه با اضافه شدن گره جدید (با مقدار ۶) باز هم یک Min-Heap باقی مانده است.

### ۱-۳-۱-۱- حذف کردن یک گره از درخت Heap : (Delete Heap)

برای حذف کردن گره  $X$  از درخت Heap با توجه به مکان قرار گیری گره دو حالت زیر وجود دارد:

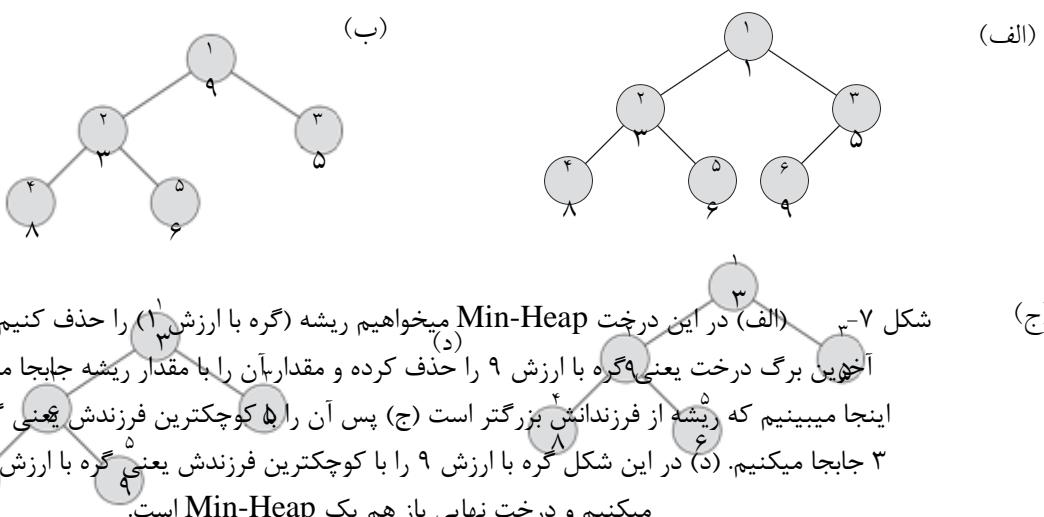
۱) گره  $X$  ریشه باشد (Delete Max در Min-Heap یا Delete Min در Max-Heap) : در این حالت مثلا در درخت Min-Heap آخرین برگ را حذف کرده و مقدار آن را بجای مقدار ریشه میگذاریم، سپس تا زمانی که همه گره ها از هر دو فرزندشان کوچکتر شوند با شروع از ریشه هر گره را با کوچکترین فرزندش جابجا میکنیم. (Heapify -> Bubble Down) شکل ۸

۲) گره  $X$  ریشه نباشد : در این حالت هم دوباره آخرین برگ را حذف کرده و مقدار آن را به جای عنصر  $X$  قرار می دهیم و بعد Heapify میکنیم.

**نکته:** در حالت ۲ معلوم نیست که باید Bubble Up انجام دهیم یا Bubble Down. این انتخاب بستگی به داده های درخت دارد. مثلا در حذف یک گره میانی در درخت Min-Heap اگر عنصر  $X$  که مقدار آن با آخرین برگ جایجا شده از پدرش کوچکتر بود Bubble Up و اگر از فرزندش بزرگتر بود Bubble Down انجام میدهیم تا درخت Min-Heap باقی بماند.

**نکته:** هزینه حذف یک گره از درخت دودویی Heapify به علت انجام عمل Heapify حداقل به اندازه ارتفاع درخت، برابر با ارتفاع آن یعنی  $O(\log n)$  است.

## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند



### Increase/Decrease Key

در Heap ها دو عمل رایج دیگر با نام های Decrease / Increase Key وجود دارد که عنصری از درخت حذف یا به آن اضافه نمی کند بلکه یک مقدار که در درخت وجود دارد را کم / زیاد میکند که بعد از انجام این کار باید یک دور Heapify هم انجام دهیم تا مطمئن شویم درخت ما همچنان Heap است در زیر نمونه کدی که در آن مقدار خانه  $A[i]$  ام آرایه را به مقدار key افزایش میدهد آورده شده برای کاهش هم به همین ترتیب عمل میکنیم :

**HEAP-INCREASE-KEY( $A, i, key$ )**

```

if key <  $A[i]$ 
    then error .new key is smaller than current key.
 $A[i] \leftarrow key$ 
while  $i > 1$  and  $A[PARENT(i)] < A[i]$ 
    do exchange  $A[i] \leftrightarrow A[PARENT(i)]$ 

 $i \leftarrow PARENT(i)$ 

```

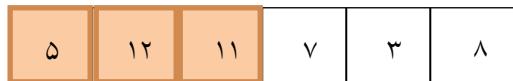
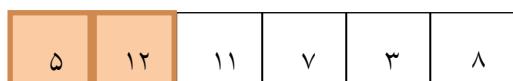
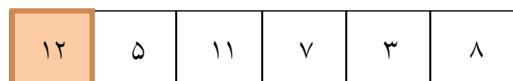
اگر ندانیم عنصری که میخواهیم مقدار آن را تغییر دهیم در چه خانه ای قرار دارد باید کل خانه های heap را چک کنیم به عبارت دیگر پیچیدگی زمانی جستجو روی درخت  $O(n)$  است.

#### ۱-۴-ساختن درخت دودویی (Make Heap) Heap

برای ساختن درخت Heap از روی آرایه ای به طول  $n$  بدون حافظه کمکی ( $O(1)$  حافظه) سه روش وجود دارد:

۱) میتوان با هزینه  $O(n \log n)$  آرایه را مرتب کرد. آرایه مرتب یک Heap است.

۲) با  $n$  بار  $O(n \log n)$  میتوانبا هزینه Insert Heap یک Heap ساخت. مانند شکل ۹.

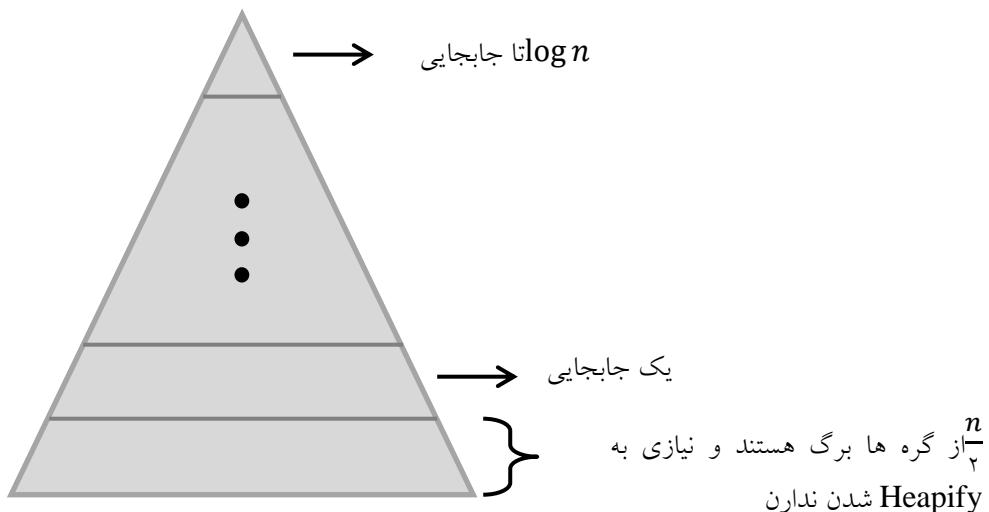


شکل ۸ - ساختن درخت برای ساخت درخت از حافظه کمکی استفاده نکرده ایم بلکه در هر مرحله از خود آرایه استفاده میکنیم و فقط چند سری خانه های آرایه را باهم جابجا (swap) کرده ایم. در واقع بخش های رنگی هر آرایه نمایانگر درخت Min-Heap ایجاد شده پس از هر بار اضافه Min-Heapify یک گره است. وقتی گره ای را اضافه میکنیم باید آن را Min-Heapify کنیم. تا درخت کماکان-

## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

Heap باقی بماند. مثلاً وقتی گره با ارزش ۳ را به درخت اضافه می‌کنیم مشاهده می‌شود که از پدرش یعنی گره با ارزش ۷ کوچکتر است که این بر خلاف Min-Heap بودن است پس آن را با پدرش جابجا می‌کنیم یعنی در آرایه خانه با مقدار ۳ را با خانه با مقدار ۷ که پدرش در درخت است جابجا (swap) می‌کنیم. و سپس به دلیل ذکر شده خانه با مقدار ۳ را با خانه با مقدار ۵ جابجا کرده و خواهیم دید که درخت با ریشه ۳ یک Min-Heap می‌باشد.

$O(n)$  با هزینه  $\frac{n}{2}$  گره آخر را Heapify می‌کنیم.



$$1 \times \log n + 2 \times (\log n - 1) + \dots + \frac{n}{4} \times 1 = O(n)$$

شکل ۹ - تعداد جابجایی هر سطح که از بالا به پایین کم می‌شود در شکل نشان داده شده است. پس برای محاسبه هزینه کل آن داریم : هزینه هر سطح (تعداد گره های آن سطح ضرب در تعداد جابجایی ها) را حساب کرده و سپس با هم جمع می‌کنیم. هزینه این کار  $O(n)$  می‌باشد.

مثال : فرض کنید دو Max Heap با نام های A,B داریم و میخواهیم این دو را در قالب یک Heap پیاده سازی کنیم بهترین روش چیست ؟ (Merge Heap)

مسئله را به دو حالت تقسیم میکنیم :

A,B برابر باشند : در این صورت سمت راست ترین برگ B را به عنوان ریشه قرار میدهیم و A را به عنوان فرزند چپ آن و B را به عنوان فرزند سمت راست قرار میدهیم و روی ریشه .Heapify را انجام میدهیم .

A,B برابر نباشند : فرض میکنیم A بزرگتر از B باشد در این صورت به اندازه B از پایین سمت چپ A جدا میکنیم و آن را C مینامیم . چون اندازه C و B برابر است طبق روش گفته شده میتوانیم آن هارا Merge کنیم و درخت حاصل را D مینامیم و آن را در جای C در درخت A قرار میدهیم و ریشه درخت D را مینامیم باید از E رو به بالا Heapify کنیم و هر کدام از رئوسی که با E جابجا میشوند را تا پایین heapify کرد چون ممکن است از راس های درخت B کوچک تر باشند.

پیچیدگی زمانی :  $O(\log(n))$

### ۱-۱-۵- کاربرد ها :

(۱) مرتب سازی Heap Sort با هزینه  $O(n \log n)$

۱. ابتدا Make-Heap را انجام میدهیم که هزینه کل آن  $O(n)$  میشود.
۲. سپس n بار Delete-Heap(Min/Max) را روی درخت اجرا میکنیم. خروجی ها مرتب هستند و هزینه آن  $O(n \log n)$  میشود.

(۲) صف اولویت:

$O(\log n)$  : Max-Heap .۱

\* بهترین روش از نظر هزینه

خروج از صف: Delete-Max با هزینه  $O(\log n)$

ورود به صف: Insert-Heap با هزینه  $O(\log n)$

## اخطار : محتويات فایل‌ها تایید شده نیستند و مفاهیم و روابط ممکن است اشتباه باشند

۲. آرایه مرتب:  $O(n)$

خروج از صف:  $O(1)$

ورود به صف:  $O(n)$

۳. آرایه نامرتب:  $O(n)$

خروج از صف:  $\theta(n)$

ورود به صف:  $O(1)$

۴. لیست پیوندی مرتب:  $O(n)$

خروج از صف:  $O(1)$

ورود به صف:  $O(n)$

۵. لیست پیوندی نامرتب:  $O(n)$

خروج از صف:  $O(n)$

ورود به صف:  $O(1)$

## فیبوناچی heap

یک نوع از درخت Heap، فیبوناچی نام دارد. به این نوع از ساختمان داده ها را تبل (lazy) می نامند که نسبت به درخت Heap معمولی دارای پیچیدگی زمانی بهتری میباشد . پیچیدگی زمانی هریک از دستور های این ساختمان داده به صورت زیر میباشد.

Delete min/max:  $O(\log(n))$

Insert :  $O(1)$

همان طور که مشاهده میکنید هزینه اضافه کردن به این نوع Heap بسیار کم است در اصل این نوع Heap در موقع اضافه کردن heapify نمیکند شیوه ای پیاده سازی این نوع Heap جزو مطالب درسی نمی باشد ولی برای اطلاع از شیوه ای پیاده سازی این درخت به آدرس زیر مراجعه کنید :

[http://en.wikipedia.org/wiki/Fibonacci\\_heap](http://en.wikipedia.org/wiki/Fibonacci_heap)

## ۱- مفهوم دیکشنری

در علوم کامپیوتر، هر ساختمان داده‌ای که دارای هر ۳ خاصیت زیر باشد، (در مورد آن عملیات‌های زیر پیاده‌سازی شده باشد) دیکشنری نام دارد:

- درج گره<sup>۱</sup>
- جستجو<sup>۲</sup>
- حذف<sup>۳</sup>

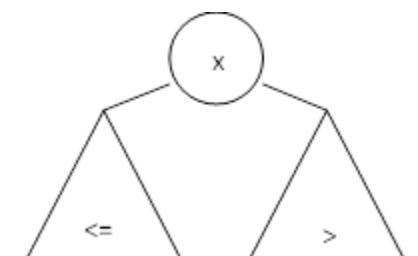
یکی از ساختمان داده‌های مهم و پرکاربرد از این دست، درخت جستجوی دودویی نام دارد.

## ۲- درخت جستجوی دودویی<sup>۴</sup>

درخت جستجوی دودویی، درختی با ویژگی‌های زیر است:

۱. درخت دودویی است. (هر گره حداقل دو فرزند دارد.)
۲. در این درخت، ارزش هر گره از تمام فرزندان سمت چپ خود بیشتر و از فرزندان سمت راست خود کمتر است. (برای حالت تساوی به دلخواه یکی از فرزندان چپ یا راست را به صورت قراردادی انتخاب می‌کنیم.)

شکل ۲,۱



شکل ۲,۱

---

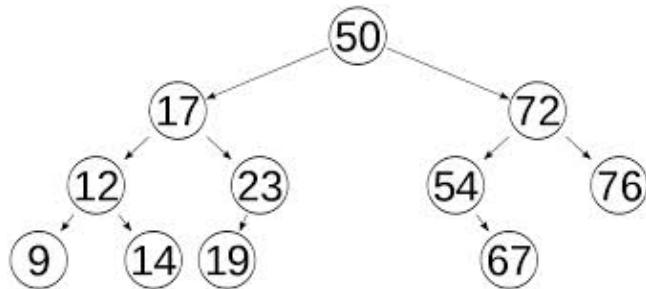
<sup>۱</sup> Insert

<sup>۲</sup> Search

<sup>۳</sup> Delete

<sup>۴</sup> Binary Search Tree (BST)

شکل‌های زیر، نمونه‌هایی از درخت جستجوی دودویی را نشان می‌دهند.



شکل ۲,۲



شکل ۲,۳

۳. اگر روی درخت جستجوی دودویی، الگوریتم میان‌ترتیب<sup>۱</sup> را اجرا کنیم، داده‌های ما مرتب شده<sup>۲</sup> هستند.

برای مثال، در شکل ۲,۲ پیمایش میان‌ترتیب که اعداد را مرتب می‌کند برابر خواهد بود با:

$$9, 12, 14, 17, 19, 23 \leq 50 < 54, 67, 72, 76$$

---

<sup>۱</sup> In-Order

<sup>۲</sup> Sorted

## ۱-۲- عملیات‌های درخت جستجوی دودویی

### ۱-۱-۲- جستجو:

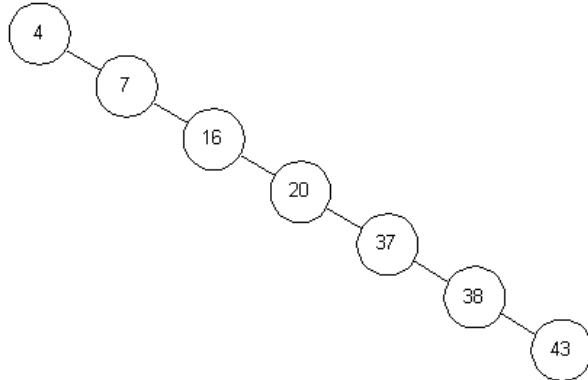
جستجو در BST به صورت بازگشتی انجام می‌شود. ابتدا داده مورد نظر را با ریشه‌ی درخت مقایسه می‌کنیم، اگر مقادیر برابر داشتند، عملیات جستجو به اتمام رسیده و گره مورد نظر یافت شده است. اگر داده مورد جستجو با ریشه برابر نبود، بررسی می‌کنیم که مقدار آن از ریشه کمتر است یا بیشتر. اگر کمتر بود، به زیردرخت سمت چپ رفته و عملیات جستجو را برای فرزند سمت چپ ریشه به عنوان ریشه‌ی جدید ادامه می‌دهیم. به صورت مشابه اگر مقدار داده از ریشه بزرگ‌تر بود، عملیات را در زیر درخت سمت راست ادامه می‌دهیم. اگر گره مورد نظر موجود باشد، با این روش پیدا می‌شود، در غیر این صورت تابع مقدار NULL را بازمی‌گرداند.

هزینه الگوریتم جستجو در BST، این‌گونه است که در بدترین حالت گرهی مطلوب ما پایین‌ترین ارتفاع ممکن از ریشه‌ی درخت است. بنابراین هزینه آن  $O(h)$  است که در آن  $h$  ارتفاع درخت می‌باشد.

- نکته: اگر درخت کامل باشد، هزینه جستجو به  $O(\log N)$  تبدیل می‌شود، چون ارتفاع درخت برای درخت کامل  $\log N$  است. در بدترین حالت ارتفاع درخت  $N$  خواهد بود. (هنگامی که ریشه اصلی درخت کوچک ترین کلید را داشته باشد و به همین ترتیب، تمام فرزندان آن به صورت مرتب قرار گیرند. شکل ۲.۴)
- بنابراین هزینه جستجو بین  $O(N)$  و  $O(\log N)$  تغییر می‌کند.

شبه کد زیر نحوه پیاده‌سازی جستجو را نشان می‌دهد:

```
FUNCTION SEARCH (KEY, NODE): // CALL INITIALLY WITH NODE = ROOT
    IF NODE = NULL OR NODE.KEY = KEY THEN
        RETURN NODE
    ELSE IF KEY < NODE.KEY THEN
        RETURN FIND-RECURSIVE (KEY, NODE.LEFT)
    ELSE
        RETURN FIND-RECURSIVE (KEY, NODE.RIGHT)
```



شکل ۲.۴

## ۲-۱-۲ - درج گره:

برای اضافه کردن یک گره، جایی که عنصر باید اضافه شود را با الگوریتم جستجو جو پیدا کرده و آن را اضافه می کنیم. نکته بسیار مهم در درخت جستجوی دودویی این است که گره های اضافه شده، همواره برگ هستند.

هزینه درج  $O(1)$  است. اما چون یکبار جستجو انجام می شود، هزینه‌ی آن همان هزینه‌ی جستجو یعنی  $O(\log N)$  می باشد.

برای به دست آوردن شبه کد درج کردن، کافی است تغییرات اندکی در شبه کد جستجو اعمال شود:

```

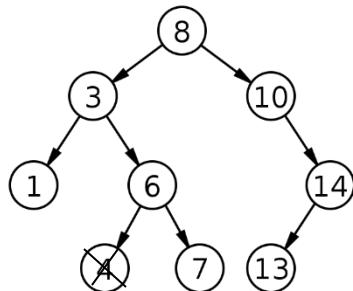
FUNCTION INSERT (KEY, NODE): // CALL INITIALLY WITH NODE = ROOT
  IF NODE = NULL
    NODE = NEW NODE (KEY)
  ELSE IF KEY <= NODE.KEY THEN
    FIND-RECURSIVE (KEY, NODE.LEFT)
  ELSE
    FIND-RECURSIVE (KEY, NODE.RIGHT)
  RETURN
  
```

## ۳-۱-۲- حذف گره:

برای حذف یک گره در درخت جستجوی دودویی، سه حالت مختلف ممکن است رخ دهد که در ادامه هر یک به صورت جداگانه شرح داده می‌شود:

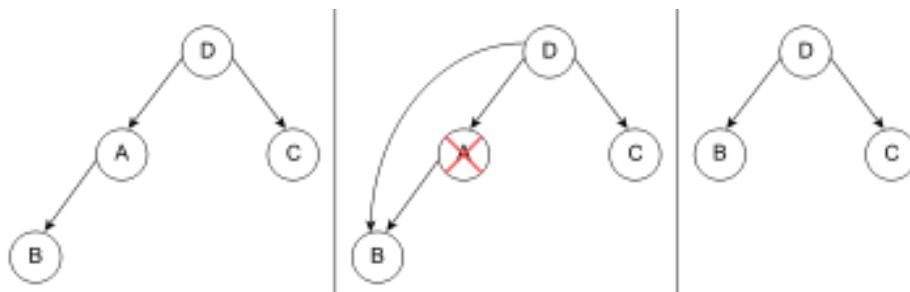
i. حذف یک برگ: در این حالت به راحتی برگ مورد نظر را حذف می‌کنیم(شکل ۲,۵) و هزینه

این کار  $O(1)$  خواهد بود.



شکل ۲,۵

ii. حذف یک گره با یک فرزند: در این حالت گره مورد نظر را حذف می‌کنیم و در ادامه پدر گره حذف شده را به فرزند آن وصل می‌کنیم.(شکل ۲,۶) هزینه انجام این کار نیز همانند حالت قبل از  $O(1)$  است.



شکل ۲,۶

iii. حذف یک گره با دو فرزند: این حالت پیچیده تر از دو حالت قبل است. در این حالت باید یکی از فرزندان گره مورد نظر جایگزین آن شود. اما این جایگزین می‌تواند از بین نوادگان آن نیز انتخاب گردد. دقت نمایید که در الگوریتم‌های حذف کردن و افروden، ثابت نگهداشتن وضعیت و ویژگی‌های درخت به عنوان یک BST بعد از عملیات حذف و اضافه نکته‌ای مهم به شمار می‌رود.

(یکی از این ویژگی‌های درخت جستجوی دودویی، خاصیت مرتب بودن پیمایش میان ترتیب است).

می‌دانیم گرهی که قرار است حذف شود ممکن است پدر داشته باشد. (مگر اینکه ریشه باشد). تمام زیر درخت‌هایی که به واسطه‌ی گره مورد نظر ما (مثلاً گره  $X$ ) به وجود آمده‌اند، نسبت به پدر  $X$  همان رابطه را دارند که  $X$  داشته است. اگر  $X$  از پدرش (مثلاً گره  $Y$ ) بزرگ‌تر باشد، فرزندانش نیز از آن بزرگ‌ترند و بالعکس. بنابراین از نظر  $Y$  تفاوتی ندارد که کدام‌یک از فرزندان  $X$ ، جای او را می‌گیرند. (یعنی بعد از حذف  $X$ ، به فرزندان  $Y$  تبدیل می‌شوند). بنابراین می‌توانیم  $X$  و فرزندانش را مستقلاب بررسی نماییم. مشکلی که وجود دارد، رابطه‌ایست که بین فرزندان  $X$  وجود دارد. باید گره‌ای انتخاب شود که اگر جایگاه  $X$  را به ارث ببرد، تعادل زیر درخت حفظ شده و همچنان فرزندان باقی‌مانده در سمت چپ از جانشین  $X$  کوچک‌تر یا مساوی و فرزندان سمت راست از آن بزرگ‌تر بمانند.

دو جانشین مناسب برای  $X$  پیدا می‌کنیم: یکی از فرزندان زیر درخت چپ و دیگری از فرزندان زیر درخت سمت راست.

به کوچک‌ترین گره زیردرخت راست گره، گره بعدی<sup>۱</sup> گفته می‌شود. این نام‌گذاری به این خاطر است که اگر عناصر درخت با توجه به مقدار آنها مرتب شوند. (مثلاً با پیمایش میان-ترتیب) این گره بلافاصله بعد از آن قرار می‌گیرد. به همین ترتیب، گره قبلی<sup>۲</sup> یک گره نیز گرهی با بزرگ‌ترین مقدار در زیردرخت سمت چپ آن است. این گره نیز پس از مرتب‌سازی در کنار گره مفروض و قبل از آن قرار می‌گیرد. در حالت ۳ برای حذف گره به جای عضو مابعد می‌توان از این گره نیز استفاده کرد.

کوچک‌ترین فرزند زیر درخت سمت راست همچنان از  $X$  و سایر راس‌ها در زیردرخت سمت چپ بزرگ‌تر و از تمام گره‌های سمت راست درخت کوچک‌تر است. از طرفی، بزرگ‌ترین فرزند زیر درخت سمت چپ از  $X$  و تمام راس‌های زیر درخت سمت راست کوچک‌تر و همچنین از تمام فرزندان زیر درخت سمت چپ بزرگ‌تر است. این دو گره به خصوص، جانشین‌های مناسبی برای  $X$  خواهند بود.

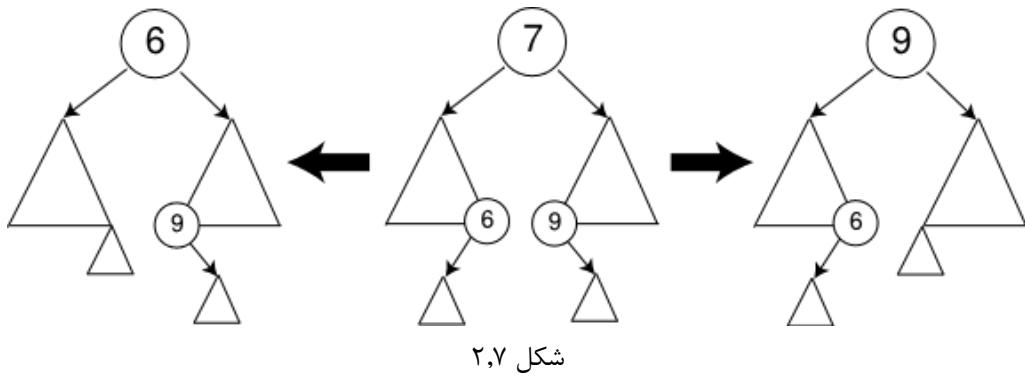
---

<sup>1</sup> Successor

<sup>2</sup> Predecessor

برای اینکه بهتر متوجه این موضوع شویم، درخت شکل ۲،۲ و پیمایش میان ترتیب آن را در نظر بگیرید. فرض کنید می خواهیم ریشه درخت که مقدارش ۵۰ است را از درخت حذف کنیم. برای جایگزین، یا بزرگ‌ترین گره زیردرخت سمت چپ (یعنی ۲۳) و یا کوچک‌ترین گره زیردرخت سمت راست (یعنی ۵۴) را جایگزین آن می کنیم. این دو عدد، دو مقدار مجاور ۵۰ در پیمایش میان ترتیب از درخت هستند و چون پیمایش میان ترتیب درخت مرتب است، پس از حذف ۵۰ و جایگزین کردن یکی از این دو عدد به عنوان ریشه، دنباله حاصل از پیمایش همچنان مرتب خواهد ماند.

برای پیدا کردن بزرگ‌ترین راس زیر درخت سمت چپ، کافی است یکبار از X به زیردرخت وارد شویم (یکبار به چپ برویم) و سپس تا جای ممکن، به سمت راست حرکت کنیم. خلاف این حرکت برای زیر درخت سمت راست انجام می‌شود. گره به دست آمده حداکثر یک فرزند خواهد داشت به دلیل این که اگر بیش از یک فرزند داشته باشد، یکی از فرزندان آن از آن بزرگتر/کوچکتر خواهد بود و می‌توانیم راسی بزرگتر/کوچکتر از راس مورد نظر در شکل پیدا کنیم. بنابراین این راس حداکثر یک فرزند دارد که به یکی از دو روش حذف قبل، آن راس را حذف کرده و جایگزین X می‌کنیم. (شکل ۲،۷)



برای اینکه درخت همواره مرتب بماند (پیمایش میان ترتیب آن مرتب بماند) در کل زمان استفاده از این ساختمان داده برای حذف یک گره با دو فرزند فقط از یکی از روش‌های اخیر استفاده می‌کنیم. برای یافتن راس جایگزین راس مورد نظر باید حداکثر به ارتفاع درخت دنبال آن بگردیم و این کار از  $O(h)$  است. هزینه حذف برگ نیز همواره  $O(h)$  است. (با در نظر گرفتن الگوریتم یافتن برگ مورد نظر)

## ۱-۴-۲- پیدا کردن گره قبلی<sup>۱</sup> و بعدی<sup>۲</sup>:

در یک درخت جستجوی دودویی، گاهی یافتن جانشین برای گره درخت یک در ترتیب منظم که با پیمایش میان ترتیب مشخص شده است، مسئله‌ای مهم به شمار می‌رود.

منظور ما از بعدی گره، گرهی است که در ترتیب مرتب شده کلید گره‌ها بلافاصله بعد از آن قرار گرفته باشد.

تعریف: در یک درخت جستجوی دودویی، بعدی گره  $x$ ، گرهی است که دارای حداقل مقدار key باشد که بزرگتر از  $\text{key}[x]$  است.

برنامه زیر بدون انجام عمل مقایسه فیلد key، اشاره‌گری به گره بعدی گره  $x$  برمی‌گرداند.  
(به شرطی که وجود داشته باشد!)

TREE-SUCCESSOR ( $x$ )

- ۱ if right [ $x$ ]  $\neq \text{Nil}$
- ۲ then return TREE-MINIMUM (right[ $x$ ])
- ۳  $y \leftarrow p[x]$
- ۴ while  $y \neq \text{Nil}$  and  $x = \text{right}[y]$
- ۵ do  $x \leftarrow y$
- ۶  $y \leftarrow p[y]$
- ۷ return  $y$

در خط ۳، مساوی پدر  $x$  می‌شود.

در خط ۴، در ریشه  $y = \text{Nil}$  می‌شود چون ریشه، پدر ندارد.

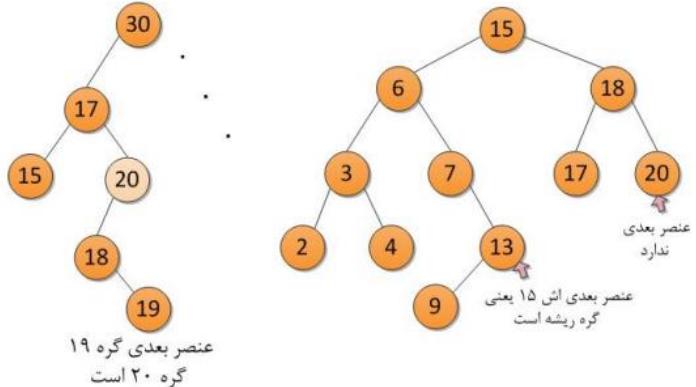
در خط ۶، مساوی پدر خودش می‌شود و  $p[y]$  اشاره‌گری به پدر  $y$  است.

الف- اگر زیر درخت راست  $X$  غیرتھی باشد، آنگاه Successor  $X$ ، چپ‌ترین گره در زیردرخت راست است که در سطر ۲ برنامه پیدا می‌شود. به عنوان مثال:

---

<sup>۱</sup> Successor

<sup>۲</sup> Predecessor



Successor of 15 is 17 Successor of 6 is 7

ب- اگر زیر درخت راست  $x$  غیرتھی باشد، آنگاه بعدی عنصر  $x$ ، عنصر  $y$  است بطوریکه  $y$  کوچکترین جدّ عنصر  $x$  است که فرزند چپ آن نیز جدّی برای  $x$  باشد.

$Y$  is the lowest ancestor of  $x$  whose left child is also an ancestor of  $x$ .

( $y$  : پدران یا اجداد)

اگر  $x=13$  باشد، آنگاه  $y=15$  پاسخ است چون 15 از پدران  $x$  است و 15 یک فرزند چپ 6 دارد از پدران 13 است.

پ- حالتی که زیر درخت راست  $x$  تھی است.

برای اینکه  $y$  یعنی بعدی گره  $x$  را پیدا کنیم، از گره  $x$  آنقدر به سمت بالا به سمت ریشه حرکت می‌کنیم تا به گرهی برسیم که فرزند چپ پدرش است، و در این مرحله از حلقه بیرون آمده و گره پدر این گره فرزند چپ، گره مورد نظر، یعنی  $y$  است. در شکل بالا اگر  $x=13$  باشد، آنگاه  $y=15$  خواهد بود. در مسیر حرکت از  $x=13$  به ریشه به گره های 7 (فرزندهای چپ پدرش یعنی عنیست) و سپس به گره 6 (فرزندهای چپ پدرش یعنی 15 است) می‌رسیم. بنابراین پدر گره 6 یعنی گره 15 گره مورد نظر ما است.

```

 $\exists y \leftarrow p[x]$ 
 $\forall \text{while } y <> \text{Nil} \text{ and } x = \text{right}[y]$ 
 $\quad \text{do } x \leftarrow y$ 
 $\quad \exists y \leftarrow p[y]$ 
 $\quad \forall \text{return } y$ 

```

اگر تابع را با  $x=17$  فراخوانی کنیم در سطر ۳،  $y=18$  می‌شود و سپس در سطر ۴،  $y \neq \text{Nil}$  است ولی  $x \neq \text{right}[y]$  است بنابراین از حلقه خارج شد و در سطر ۷ مقدار  $y=18$  که به درستی بعدی گره  $x=17$  است برگشت داده می‌شود.

**یادآوری مهم :** این تابع را نباید برای گره  $X$  که بزرگترین گره درخت است و بعدی ندارد صدا بزنیم.

زمان اجرای الگوریتم TREE-SUCCESSOR بر روی درختی با ارتفاع  $h$  مساوی  $O(h)$  است. چون یا یک مسیر به طرف ریشه درخت و یا مسیری به طرف پایین (یک برگ) را طی می‌کنیم و حداکثر طول مسیر همان ارتفاع درخت است.

#### -۴-۱-۲- پیدا کردن گره حداقل:

به علت خاصیتی که یک درخت جستجوی دودویی دارد گرهی که دارای کلید حداقل است را از طریق دنبال کردن مسیری که اشاره‌گرهای فرزند چپ نشان می‌دهند تا به  $NiL$  برسیم، می‌توانیم پیدا کنیم.

الگوریتم زیر این کار را انجام می‌دهد:

TREE-MINIMUM ( $x$ )

۱ while left [ $x$ ]  $\neq NiL$

۲ then  $x \leftarrow \text{left } [x]$

۳ return  $x$

الف - گره  $X$  که با آن فراخوانی انجام شده هیچ فرزند چپی ندارد، در این صورت در سطر ۳ خود  $X$  برمی‌گردد.

ب - گره  $X$  که فرزند چپ دارد، در این صورت سطر ۲ آن آنقدر تکرار می‌شود تا گره  $X$  یک برگ شود و یا اینکه گرهی بشود که فرزند چپ ندارد.

در هر دو صورت این گره  $X$  گرهی خواهد بود که فیلد key آن حداقل در کل درخت جستجوی دودویی خواهد بود.

## پیدا کردن گره حداقل:

TREE-MAXIMUM (x)

۴ while right [x] <> NIL

۵ then x <- right [x]

۶ return x

عملکرد این الگوریتم مانند الگوریتم TREE-MINIMUM است، با این تفاوت که مسیر فرزند راست را طی می کند تا به گرهی برسد که فرزند راست نداشته باشد.  
فیلد key این گره حاوی حداقل key این درخت خواهد بود.  
زمان اجرای هر یک از دو الگوریتم فوق  $O(h)$  است که ارتفاع درخت دودویی است.

## ۲-۲- نکات مهم:

i. بسیاری داده‌ها، قابل مرتب‌سازی نیستند و یا گاهی داده‌ها بسته‌هایی هستند از متغیرها (مانند کلاس یا Struct) در این موارد به هر داده یک کلید نسبت می‌دهیم که قابل مرتب‌سازی باشند، سپس درختی را با استفاده از این کلیدها ایجاد کرده و عملکرها را روی آن انجام می‌دهیم.

ii. در درخت جستجوی دودویی، بسته به درج کردن و حذف کردن‌هایی که روی آن انجام می‌شود، تقارن و توازن بعد از مدتی از بین رفته و ارتفاع درخت از  $n$  به  $\log n$  تغییر می‌کند و هزینه‌ی زمانی الگوریتم‌ها افزایش می‌یابد. (در BST تمامی الگوریتم‌ها  $O(h)$  هستند). برای این‌که همواره تقارن و توازن برقرار بماند، درخت‌های دودویی دیگری به وجود آمده‌اند. B-Tree و Red-Black Tree، AVL اعمال عملکرها، هزینه الگوریتم‌های دیکشنری در آن‌ها  $(\log n)$   $O$  می‌باشد.

iii. شبیه کد برای پیمایش میان‌ترتیب داده‌ها:

Function Inorder-Tree-Walk (x)

If  $x \neq \text{NIL}$

Then Inorder-Tree-Walk (left[x])

Print key[x]

Inorder-Tree-Walk (right[x])

این الگوریتم، هزینه‌ای معادل  $\Theta(n)$  خواهد داشت. (قضیه ۱۲.۱ کتاب CLRS).

iv. روش دیگر برای متوازن ساختن درخت جستجوی دودویی، این است که ابتدا روی درخت الگوریتم میان‌ترتیب را اجرا نماییم تا دنباله مرتب داده‌ها به دست آید. سپس مجدداً به صورت بازگشتی، درخت را می‌سازیم. هزینه این الگوریتم برابر هزینه الگوریتم میان‌ترتیب یعنی  $\Theta(n)$  خواهد بود.

v. تعداد درخت‌های BST با ارتفاع  $n-1$  برابر است با  $2^{n-1}$   
همچنین تعداد درخت‌های BST با ارتفاع  $n-2$  برابر است با  $2^{n-2}(n-2)$  - ۱

## ۵. الگوریتم‌های مرتب سازی

### ۵. مرتب سازی

الگوریتم‌های مرتب سازی را می‌توان به چند روش دسته‌بندی کرد.

- مقایسه‌ای و غیر مقایسه‌ای

مبتنی بر مقایسه : در مرتب سازی مقایسه‌ای عناصر موجود مستقیماً با یکدیگر مقایسه می‌شوند و این مقایسه تعیین کننده مکان آنهاست و با مقایسه دو عنصر جابجایی عناصر صورت می‌گیرد.

غیر مبتنی بر مقایسه: هیچ مقایسه‌ای میان عناصر صورت نمی‌گیرد و روش‌های دیگری مکان نهایی عناصر را تعیین می‌کنند.

پیچیدگی زمانی الگوریتم‌های مرتب سازی مقایسه‌ای در بدترین حالت از  $O(n \log n)$  بهتر نمی‌شود در حالی که الگوریم مرتب سازی غیر مقایسه‌ای می‌تواند پیچیدگی بدترین حالت در  $O(n)$  نیز داشته باشد.

- پایدار (stable) و ناپایدار (unstable)

الگوریتم مرتب سازی پایدار است اگر هنگام مرتب سازی موقعیت نسبی عناصر با کلید برابر را نسبت به هم تغییر ندهد و ترتیب نسبی عناصر مساوی قبل و بعد از مرتب سازی یکسان باشد.

- خارجی (external) و داخلی (internal)

در مرتب سازی داخلی تنها از حافظه اصلی استفاده می‌شود و در صورتی قابل استفاده است که تعداد و حجم عناصر بیشتر از ظرفیت حافظه اصلی نباشد در صورتی که نتوان مرتب سازی را به صورت داخلی انجام داد از مرتب سازی خارجی استفاده می‌شود که از حافظه جانبی برهه می‌برد.

## • درجا (in-place) و بروز جا (out-place)

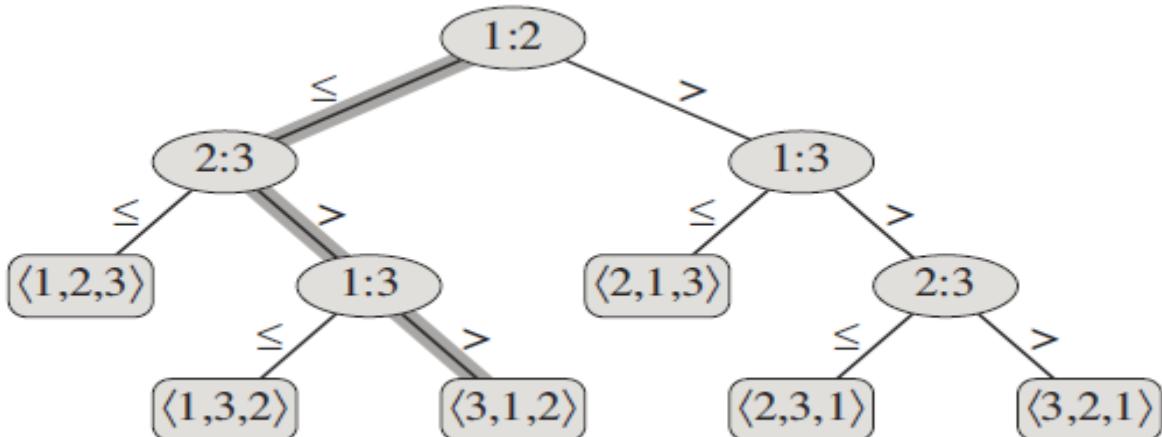
الگوریتم مرتب سازی را درجا میگویند، زمانی که فضای موقت و مورد استفاده ان با اندازه ورودی ارتباط نداشته باشد. مرتب سازی درجا عناصر را با پیچیدگی حافظه  $O(1)$  مرتب می کند.

از سوی دیگر اندازه فضای اضافی موقتی که الگوریتم های مرتب سازی بروز جا نیاز دارند با تعداد داده های ورودی متناسب است.

## درخت تصمیم Decision Tree

درخت تصمیم یک روش برای معادل سازی تصمیم گیری است. مساله‌ی تصمیم گیری به هر مساله گفته میشود که از بین  $k$  انتخاب یکی از تصمیم‌ها را بگیرد. در درخت تصمیم برگ‌ها انتخاب‌های ممکن ما هستند و گره‌های داخلی شرط‌های ممکن را پوشش دهیم، باید حداقل به آن تعداد برگ داشته باشیم).

به عنوان مثال شکل زیر را در نظر بگیرید:



در این شکل میخواهیم اعداد 1، 2، 3 را مرتب کنیم. در کل  $3! = 6$  حالت برای چینش این سه عدد در کنار یکدیگر داریم. بنابراین تعداد برگ‌ها حداقل 6 تا باید باشد. هر گره‌ی داخلی بیانگر یک شرط است برای مثال در گره‌ی ریشه، عناصر 1 و 2 با هم مقایسه شده‌اند و اگر 1 از 2 زودتر آمده باشد، به زیر درخت سمت چپ رفته و اگر 2 زودتر از 1 آمده باشد به زیر درخت سمت راست میرویم.

در این رابطه مساله‌ی مرتب سازی مبتنی بر مقایسه یک مساله‌ی تصمیم‌گیری می‌باشد چرا که از بین تمام جایگشت‌های قرارگیری  $n$  عدد در کنار هم (کل  $n!$  حالت ممکن)، میخواهیم یک جایگشت را انتخاب کنیم.

در این درخت تصمیم، تعداد برگ‌ها بزرگتر مساوی  $n!$  است. از طرفی میدانیم ارتفاع هر درخت دودویی بزرگتر مساوی  $\log(n)$  می‌باشد و هزینه‌ی زمانی پیدا کردن جواب مطلوب در درخت تصمیم برابر ارتفاع درخت می‌باشد (چرا که در هر سطر یک مقایسه لازم است) بنابرین:

$$h \geq \log(Tedad\ Bargha) \geq \log(n!) = n\log n$$

$$\rightarrow h \geq n\log n$$

## ۱.۵. مرتب سازی شمارشی

در مرتب‌سازی شمارشی فرض بر این است که هر یک از  $n$  عنصر ورودی، یک مقدار صحیح بین  $0$  تا  $k$  به ازای یک مقدار صحیح  $k$  می‌باشد. هنگامی که  $O(n)$  باشد، مرتب‌سازی در زمان  $\theta(n)$  اجرا می‌شود.

ایده اساسی مرتب‌سازی شمارشی این است که به ازای هر عنصر ورودی  $X$ ، تعداد عناصر کوچکتر از  $X$  تعیین شود. با این اطلاعات می‌توان مکان عنصر  $X$  در آرایه خروجی را به طور مستقیم تعیین کرد. برای مثال اگر ۱۷ عنصر کوچکتر از  $X$  داریم،  $X$  متعلق به خانه شماره ۱۸ از رشته خروجی می‌باشد. این طرح، باید اندکی تغییر کند تا در شرایطی که عناصر مختلف دارای مقادیر یکسان وجود دارند نیز قابل اعمال باشد. چرا که نمی‌خواهیم عناصر دارای مقدار مساوی را در یک مکان قرار دهیم.

در کد مرتب‌سازی شمارشی، فرض می‌کنیم که ورودی، آرایه  $A[1...n]$  می‌باشد که  $\text{length}[A]=n$  است. به دو آرایه دیگر نیز نیاز داریم. یکی آرایه  $B[1...N]$  که خروجی مرتب شده را در خود نگه دارد و یکی آرایه  $C[0...k]$  که فضای کاری موقتی را برای ما فراهم می‌کند.

Counting-Sort ( A, B, K )

۱        for  $i \leftarrow 0$  to  $k$

۲              do  $C[i] = 0$

```

۱   for j ← 1 to length[A]
۲     do C[A[j]] = C[A[j]] + 1
۳   C[i] now contains the number of elements equal to i
۴   for i ← 1 to k
۵     do C[i] ← C[i] + C[i-1]
۶   C[i] now contains the number of elements less than or equal to i.
۷   for j ← length [A] downto 1
۸     B [C[A[j]]] ← A[j]
۹     C[A[j]] ← C[A[j]] - 1

```

A	1	2	3	8	5	6	7	8
	2	5	3	0	2	3	0	3
C	0	1	2	3	4	5		
	2	0	2	3	0	1		

(a)

C	0	1	2	3	4	5
	2	2	4	7	7	8

(b)

B	1	2	3	4	5	6	7	8
	0	1	2	3	4	5		
C	2	2	4	6	7	8		

(c)

A	1	2	3	4	5	6	7	8
	2	5	3	0	2	3	0	3
C	0	1	2	3	4	5		

(d)

B	1	2	3	4	5	6	7	8
	0	1	2	3	4	5		
C	1	2	4	6	7	8		

(e)

B	1	2	3	4	5	6	7	8
	0	0	2	2	3	3	3	5
C	1	2	4	5	7	8		

(f)

شکل ۲.۸ : عملیات مرتب‌سازی شمارشی بر روی آرایه ورودی  $[1\dots 8]$  که هر عنصر آن دارای مقداری صحیح و نامنفی و کوچکتر یا مساوی ۵ است.

(a) آرایه  $A$  و آرایه  $C$  کمکی بعد از خط ۴

(b) آرایه  $C$  بعد از خط ۷

c-e) آرایه خروجی  $B$  و آرایه کمکی  $C$  پس از یک، دو و سه بار تکرار حلقه خطوط ۹ تا ۱۱ تنها خانه‌های روشن در آرایه  $B$  پر شده‌اند.

f) آرایه مرتب‌شده خروجی  $B$

شکل ۸. روند کار مرتب‌سازی شمارشی را نمایش می‌دهد. پس از مقدار دهی اولیه در حلقه `for` در خطهای ۱ و ۲، در حلقه `for` در خطهای ۳ و ۴ به تک تک عناصر ورودی رسیدگی می‌کنیم. اگر مقدار یک عنصر ورودی  $i$  بود،  $C[i]$  را یک واحد افزایش می‌دهیم. لذا پس از خط ۴،  $C[i]$  تعداد عناصر ورودی مساوی با ۱ در ورودی  $i$  را به ازای  $k$  و  $\dots, 2, 1, 0 = i$  نگهداری می‌کند. در خطوط ۶ و ۷، به ازای  $k, \dots, 1, 0 = i$ ، تعداد عناصر کوچکتر مساوی ۱ در آرایه ورودی را در  $C[i]$  ذخیره می‌کنیم.

در انتهای، در حلقه `for` خطوط ۹ تا ۱۱، هر عنصر  $A[j]$  را در مکان صحیح خود در آرایه خروجی  $B$  قرار می‌دهیم.

اگر هر  $n$  عنصر متمایز باشند، هنگامی که وارد خط ۹ می‌شویم، برای هر  $A[j]$ ، مقدار  $C[A[j]]$  تعداد عناصر کوچکتر یا مساوی  $A[j]$  است. از آنجا که ممکن است عناصر متمایز نباشند، هر بار که  $A[j]$  را در آرایه  $B$  قرار می‌دهیم، از  $C[A[j]]$  یکی کم می‌کنیم که سبب می‌شود عنصر ورودی بعدی، با مقداری برابر  $A[j]$ ، اگر موجود است، به خانه درست قبل از  $A[j]$  در آرایه خروجی برود.

مرتب‌سازی شمارشی به چه زمانی احتیاج دارد؟ حلقه ۱ و ۲، به اندازه  $\theta(k)$  زمان می‌گیرند. حلقه `for` خطوط ۳ و ۴ به اندازه  $\theta(n)$  و حلقه `for` در خطوط ۶ و ۷، به اندازه  $\theta(k)$  و حلقه `for` در خطوط ۹ تا ۱۱ به اندازه  $\theta(n)$  زمان می‌گیرند. لذا روی هم رفته به اندازه  $\theta(n+k)$  زمان صرف می‌شود. در عمل، هنگامی از مرتب‌سازی شمارشی استفاده می‌کنیم که  $O(n)=k$  باشد که در این شرایط هزینه زمانی از  $\theta(n)$  خواهد شد.

مرتب‌سازی شمارشی، بر کران پائین  $\Omega(n \log n)$  غلبه می‌کند، چرا که این یک روش مرتب‌سازی مقایسه‌ای نیست. در واقع هیچ مقایسه‌ای بین عناصر ورودی صورت نمی‌گیرد. در عوض، مرتب‌سازی شمارشی از مقادیر واقعی عناصر برای آدرس‌دهی در آرایه استفاده می‌کند.

تمرین. ۸ عدد داریم که می‌دانیم همگی اعداد صحیحی بین ۱ تا ۶ می‌باشند، اعداد را با روش مراتب کنید

آرایه اعداد

3	6	4	1	3	4	1	4
---	---	---	---	---	---	---	---

آرایه تعداد تکرار

2	0	2	3	0	1
---	---	---	---	---	---

آرایه تجمعی تعداد تکرار

2	2	4	7	7	8
---	---	---	---	---	---

مراحل سرت

2	2	4	6	7	8
---	---	---	---	---	---

4
---

1	2	4	6	7	8
---	---	---	---	---	---

1	4
---	---

2	2	4	5	7	8
---	---	---	---	---	---

1	4	4
---	---	---

.....

1	1	3	3	4	4	4	6
---	---	---	---	---	---	---	---

ویژگی های مرتب سازی شمارشی :

غیر مقایسه ای : این الگوریتم غیر مقایسه ای است چرا که هیچ مقایسه ای بین مقدار عناصر صورت نمیگیرد.

برون جا : برای مرتب کردن  $n$  عنصر نیاز به آرایه ای کمکی به اندازه  $O(k)$  داریم و برای نگه داشتن پاسخ نیز به آرایه ای به اندازه  $O(n)$  نیاز است.

پایدار : در خط ۹ در الگوریتم اگر عناصر را از اول به آخر بررسی میکردیم الگوریتم درست کار میکرد ولی دیگر پایدار نمی ماند. پس چون از آخر به اول بررسی کردیم، الگوریتم پایدار است.

هزینه زمانی الگوریتم فوق همانطور که محاسبه شد  $\theta(n+k)$  است.

یک ویژگی مهم مرتب‌سازی شمارشی، پایدار بودن آن است: اعداد با مقادیر یکسان، در آرایه خروجی با همان ترتیب موجود در آرایه ورودی ظاهر می‌شوند و این به معنای این قانون است که بین عناصر یکسان هر عنصری که در آرایه ورودی، زودتر ظاهر می‌شود، در آرایه خروجی نیز زودتر ظاهر خواهد شد. معمولاً ویژگی پایدار بودن، تنها زمانی اهمیت دارد که اطلاعات دیگری نیز در ارتباط با هر عنصر مورد نظر در مرتب‌سازی وجود دارد. پایداری مرتب‌سازی شمارشی، از یک لحاظ دیگر نیز دارای اهمیت است و آن اینکه روش مرتب‌سازی شمارشی اغلب به عنوان یک زیرروال در مرتب‌سازی مبنایی بکار می‌رود.

همانطور که در بخش بعد خواهیم دید، پایداری مرتب‌سازی شمارشی در صحت مرتب‌سازی مبنایی، بسیار حائز اهمیت است.

## ۲.۵. مرتب‌سازی مبنایی

مبنای مرتب‌سازی مبنایی، مرتب‌سازی شارشی است.

مرتب‌سازی مبنایی، الگوریتمی است که مورد استفاده ماشین‌های مرتب‌سازی کارتی بوده که هم اکنون تنها در موزه‌ها پیدا می‌شوند. در این روش، کارت‌ها در هشتاد ستون قرار می‌گیرند که در هر ستون یک سوراخ می‌تواند در ۱۲ حالت ایجاد شود. دستگاه مرتب‌ساز می‌تواند به صورت مکانیکی برنامه ریزی شود تا یک ستون داده شده از هر کارت در یک دسته را بررسی کند و کارت‌ها را در یکی از ۱۲ سطل بسته به اینکه در کجا سوراخ شده توزیع کند. یک اپراتور می‌تواند کارت‌ها را سطل به سطل جمع آوری کند، به صورتی که کارت‌هایی که در اولین جای ممکن سوراخ شده اند بر روی آن‌هایی قرار بگیرد که در محل دوم سوراخ شده اند، الى آخر.

برای ارقام ددهی، تنها ۱۰ قسمت در هر ستون استفاده می‌شود. (دو محل دیگر برای کد کردن کاراکتر‌های غیر عددی به کار می‌روند). یک عدد  $d$ -رقمی می‌تواند بخش  $d$  ستونی ای را پر کند. از آنجایی که مرتب‌ساز کارت‌ها در هر زمان می‌تواند تنها یک ستون را بررسی کند، مسئله مرتب‌سازی  $n$  کارت در یک عدد  $d$ -رقمی نیازمند یک الگوریتم مرتب‌سازی است.

329	720	720	329
457	355	329	355
657	436	436	436
839	.....	839	.....
436	657	355	657
720	329	457	720
355	839	657	839

شکل ۸,۳ عملکرد مرتب سازی مبنایی بر روی هفت عدد سه رقمی . سمت چپ ترین ستون نشان دهنده ورودی است بقیه ستون ها نشان دهنده اعداد بعد از انجام یک حلقه هستند. سایه ها نشان دهنده ستون هایی هستند که تفاوت هر لیست با لیست قبل از مرتب کردن آنها ناشی می شود.

بنابراین، یک شخص ممکن است بخواهد تا اعداد را بر حسب با ارزش ترین رقمشان مرتب بکند، و سپس دسته ها را به ترتیب با هم ترکیب کند. متأسفانه، از آنجایی که کارت ها در ۹ سطل از ۱۰ سطل برای مرتب سازی هر سطل باید کنار گذاشته شوند، این رویه تعداد زیادی پشته های واسط از کارت ها را ایجاد می کند که باید جای هر کدام مشخص باشد.

مرتب سازی مبنایی مشکل مرتب سازی کارت ها را به طور غیر ذاتی با مرتب کردن کم ارزش ترین رقم در ابتدا حل می کند. سپس کارت ها به ترتیب، به یک دسته ترکیب می شوند. به صورتی که پشت سر کارت های سطل صفرم کارت های سطل یکم قرار بگیرند که پس از آن ها کارت های سطل دو ... قرار بگیرند. سپس تمام دسته دوباره بر اساس دومین رقم بی ارزش ترشان مرتب سازی می شوند و به روش مشابهی با هم ترکیب می شوند. این روند تا زمانی که کارت ها بر اساس همه ۴ رقمشان مرتب شوند ادامه پیدا می کند. جالب توجه است که، در این زمان کارت ها بر اساس همه ۴ رقمشان مرتب خواهند بود. بنابراین، تنها ۴ حرکت روی دسته لازم خواهد بود تا مرتب سازی کامل شود. شکل ۸,۳ نشان می دهد که مرتب سازی مبنایی روی یک دسته هفت عدد سه رقمی صورت می گیرد.

این نکته مهم است که رقم های مرتب شده در این الگوریتم پایدار باشند. مرتب سازی ای که با یک مرتب ساز کارت صورت می گیرد پایدار است، ولی اپراتور باید درباره تغییر ندادن ترتیب کارت ها هنگامی که از یک سطل

بیرون می آیند احتیاط کند، حتی اگر همه کارت های درون یک سطل رقم مشابهی در یک ستون انتخابی داشته باشند.

در یک کامپیوتر معمولی، که یک ماشین با دسترسی تصادفی متوالی است، مرتب سازی مبنایی گاهی برای مرتب سازی رکوردهایی از اطلاعات که با فیلد های متعددی کلید گذاری شده اند به کار می رود. به عنوان مثال، ما ممکن است بخواهیم روزها را با سه کلید مرتب کنیم : سال، ماه، و روز. ما می توانیم یک الگوریتم مرتب سازی با یکتابع مقایسه اجرا کنیم که دو روز را بگیرد و سال ها را مقایسه کند اگر شباهتی بود ، ماه ها را مقایسه کند و اگر شباهتی دیگر پیدا شد، روزها را مقایسه کند. متناظراً ما می توانیم اطلاعات را سه بار با مرتب سازی پایدار، مرتب کنیم : اول بر اساس روز، دوم بر اساس ماه، و در آخر بر اساس سال.

- کد مربوط به مرتب سازی مبنایی بسیار سر راست است. رویه زیر فرض می کند که هر عنصر از یک آرایه  $n$ - عنصره  $A$ ، رقمه  $d$  است، که در آن رقم اول کوچکترین مرتبه است و رقم  $d$  ام بزرگ ترین مرتبه.

RADIX\_SORT ( $A, d$ )

**for**  $i \backslash$  to  $d$

**do** use a stable sort to sort array  $A$  on digit  $i$

اعداد شامل  $d$  رقم که هر رقم در آن می تواند  $k$  مقدار متفاوت بگیرد داده شده است، RADIX\_SORT این اعداد را به درستی در زمان  $\theta(d(n+k))$  مرتب می کند.

## اثبات

درستی مرتب سازی مبنایی از مقدمه استقرا برای ستونی که مرتب شده می آید. تحلیل زمان اجرا بستگی به مرتب سازی پایداری دارد که توسط الگوریتم مرتب سازی واسط استفاده می شود. هنگامی که هر رقم در بازه  $0 - 1 - k$  است (بنابراین می تواند  $k$  حالت ممکن داشته باشد)، و در آن  $k$  عدد زیاد بزرگی نیست، مرتب سازی شمارشی انتخاب مسلم است. بنابراین، هر حرکت بر روی  $n$  عدد  $d$ -رقمه زمان  $\theta(n+k)$  خواهد گرفت. و برای  $d$  حرکت زمان کل مرتب سازی مبنایی  $(d(n+k))$  خواهد بود.

زمانی که  $d$  ثابت است و ، مرتب سازی مبنایی در زمان خطی اجرا می شود. به طور کلی تر، ما مقداری انعطاف در شکستن هر کلید به رقم ها داریم.

$n$  عدد  $b$ -بیتی و هر عدد صحیح مثبت داده شده اند، RADIX\_SORT این اعداد را در  $\theta((b/r)(n + 2^r))$  مرتب می کند.

تمرین : به کمک مرتب سازی مبنایی نشان دهید که چگونه میتوان  $n$  عدد صحیح بین  $0$  تا  $n^{3-1}$  را با هزینه  $O(n)$  زمانی مرتب کرد؟

پاسخ : عدد صحیح را عددی سه رقمی در مبنای  $n$  در نظر بگیرید. در این صورت این عدد سه رقمی ارقامش از  $n-1$  تغییر میکنند. حال میتوانید این عدد سه رقمی را به روش مرتب سازی مبنایی، مرتب کنید. پس باید ۳ بار تابع مرتب سازی شمارشی را صدا بزنید که در آنصورت هزینه ای زمانی الگوریتم فوق خواهد بود  $O(3n)$  که در کل هزینه ای زمانی  $O(n)$  خواهد شد.

### ویژگی های مرتب سازی مبنایی

(۱) غیر مقایسه ای: در این الگوریتم نیز درست مشابه مرتب سازی شمارشی، هیچ مقایسه ای میان عناصر صورت نمی گیرد.

(۲) برون جا: در این الگوریتم در هر مرحله پخش و جمع کردن عناصر، به حافظه ای از مرتبه تعداد عناصر نیاز است.

(۳) پایدار: تمامی مراحل پخش و جمع کردن عناصر از ابتدا به انتهای لیست و خوشها صورت میگیرد، بنابراین اگر دو عنصر یکسان در لیست موجود باشند، مکان انها تغییر نمی کند.

(۴) پیچیدگی زمانی در همه حالت ها  $\theta(kn + kr)$  است. که  $k$  طول رشته ها و  $R$  مبنای عناصر و  $n$  تعداد عناصر است.

## ۲.۵. مرتب سازی سطلی

هنگامی که داده های ورودی از توزیع یکنواختی برخوردار باشند، مرتب سازی سطلی در زمان خطی اجرا می شود. مشابه مرتب سازی شمارشی، مرتب سازی سطلی نیز سریع است، زیرا در مورد ورودی چیزی را فرض می کند. مرتب سازی شمارشی تصور می کند که ورودی ها شامل اعداد صحیح در بازه کوچکی هستند، در حالی که مرتب سازی سطلی چنین تصور می کند که ورودی توسط یک روند تصادفی ایجاد شده که در آن عناصر به طور یکسان در بازه  $[0, n]$  قرار گرفته اند.

ایدها مرتب سازی سطلی تقسیم بازه های  $[0, n]$  به  $n$  زیربازه های هم اندازه، یا  $n$  سطل است. پس از آن ورودی در سطل ها قرار داده می شوند. از آنجایی که ورودی ها به صورت یکنواخت روی بازه  $[0, n]$  توزیع شده اند، ما انتظار نداریم که تعداد زیادی شماره در هر سطل قرار بگیرد. برای ایجاد خروجی، ما به سادگی، شماره های هر سطل را مرتب می کنیم و سپس به ترتیب از سطلی به سطل دیگر می رویم و عناصر موجود در هر یک را لیست می کنیم.

کد ما برای مرتب سازی سطلی، فرض می کند که ورودی یک آرایه  $n$ -عنصره است و هر عنصر آرایه به صورت  $A[1] \rightarrow A[n]$  است. این کد به یک آرایه کمکی  $B[0..n]$  از لیست های پیوندی (سطل ها) احتیاج دارد و فرض می کند که مکانیسمی برای نگهداری چنین لیستی موجود است.

BUCKET\_SORT (A)

- ۱  $n \leftarrow \text{length}[A]$
- ۲ **for**  $i \leftarrow 1$  **to**  $n$   
۳     **do**  $\text{insert } A[i] \text{ into list } B[[n, A[i]]]$
- ۴ **for**  $I \leftarrow 1$  **to**  $n - 1$   
۵     **do**  $\text{sort list } B[i] \text{ with insertion sort}$
- ۶  $\text{concatenate the lists } B[0], B[1], \dots, B[n - 1]$

شکل ۸.۴ عملکرد مرتب سازی سطلی را بر روی آرایه ورودی از ۱۰ عدد نشان می دهد.

برای اینکه ببینیم این الگوریتم چگونه کار می کند، دو عنصر  $A[i]$  و  $A[j]$  را در نظر بگیرید. فرض کنید بدون از دست دادن کلیت موضوع داریم  $A[i] \geq A[j]$ . از آنجایی که (کف)  $[n A[i]] <= [n A[j]]$  عنصر  $A[i]$  یا در سطل مشابهی با  $A[j]$  قرار گرفته است و یا در سطلی با اندیس پایین تر. اگر  $A[i] \geq A[j]$  در سطل مشابهی باشند، حلقه **for** در خط های ۴ و ۵ آن ها را در ترتیب صحیح قرار می دهد. اگر این دو عنصر در سطل های متفاوتی باشند، خط ۶ ام آن ها را در ترتیب صحیح قرار می دهد. بنابراین، مرتب سازی سطلی درست کار می کند.

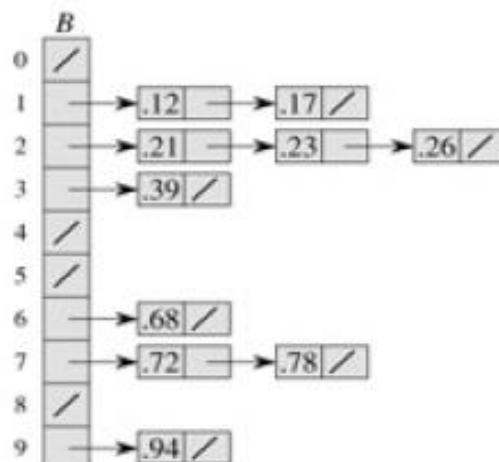
برای تحلیل زمان اجرای این الگوریتم، توجه کنید که همه خطوط به جز خط ۵ در بدترین حالت زمان اجرای  $O(n)$  خواهند داشت. این موضوع در توازن با زمان کل  $n$  فراخوانی در مرتب سازی درجی در خط ۵ است.

برای تحلیل هزینه همه فراخوانی ها در مرتب سازی درجی، تصور کنید  $ni$  متغیر تصادفی باشد که بیانگر تعداد عناصر قرار گرفته در سطل  $B[i]$  باشد. از آنجایی که مرتب سازی درجی در زمان درجه دویی اجرا می شود (بخش ۲،۲ را نگاه کنید)، زمان اجرای مرتب سازی سطلی عبارتست از

$$T(n) = \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

	<i>A</i>
1	.78
2	.17
3	.39
4	.26
5	.72
6	.94
7	.21
8	.12
9	.23
10	.68

(a)



(b)

## عملکرد مرتب سازی سطلی

شکل بالا عملکرد مرتب سازی سطلی را نشان می دهد. (a) آرایه ورودی  $A[1..n]$ . (b) آرایه  $B[1..n]$  از لیست های مرتب شده (سطل ها) بعد از خط ۵ ام الگوریتم. سطل  $i$  ام مقادیر موجود در نیم بازه  $(i/2, i+1)$  را نگه می دارد. خروجی مرتب شده شامل به هم چسباندن به ترتیب لیست های  $B[1..n]$  است.

با گرفتن امید ریاضی از دو طرف و با توجه به خاصیت خطی بودن امید ریاضی و با توجه به تساوی C.۲۱ داریم

$$\begin{aligned} E[T(n)] &= E \left[ \Theta(n) + \sum_{i=0}^{n-1} O(n_i^2) \right] \\ &= \Theta(n) + \sum_{i=0}^{n-1} E[O(n_i^2)] \\ &= \Theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) \end{aligned}$$

ما ادعا می کنیم که برای  $i = 1, 2, \dots, n-1$  داریم

$$E[n_i^2] = 2 - 1/n$$

تعجبی در این نیست که هر سطل  $i$  ام شامل مقدار  $E[ni]$  است، زیرا هر مقدار در آرایه ورودی  $A$  به طور مشابه وارد هر سطل می شود. برای اثبات تساوی (۸.۲)، ما متغیرهای تصادفی نشانگر را تعریف می کنیم

$$X_{ij} = \begin{cases} 1 & \text{در سطل } i \text{ قرار بگیرد} \\ 0 & \text{در سطل } i \text{ قرار ندارد} \end{cases}$$

برای  $i = 1, 2, \dots, n-1$  و  $j = 1, 2, \dots, n-i$ ، بنابراین،

$$n_i = \sum_{j=1}^n X_{ij} .$$

برای محاسبه  $E[ni]$ ، ما عبارت  $\sum_{j=1}^n X_{ij}$  را باز می کنیم و عبارات را دوباره دسته بندی می کنیم :

$$\begin{aligned}
E[n_i^2] &= E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] \\
&= E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij}X_{ik}\right] \\
&= E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} X_{ij}X_{ik}\right] \\
&= \sum_{j=1}^n E[X_{ij}^2] + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} E[X_{ij}X_{ik}] ,
\end{aligned}$$

به طوری که خط آخر از خاصیت خطی بودن امید ریاضی به دست آمد. ما دو جمع را جدا حساب می کنیم.  
نشانگر متغیر تصادفی  $X_{ij}$  یک است با احتمال  $1/n$  و برای حالت های دیگر و بنابراین

$$\begin{aligned}
E[X_{ij}^2] &= 1 \cdot \frac{1}{n} + 0 \cdot \left(1 - \frac{1}{n}\right) \\
&= \frac{1}{n} .
\end{aligned}$$

هنگامی که  $j \neq k$  متغیر های  $X_{ij}$  و  $X_{ik}$  مستقل از هم هستند و بنابراین

$$\begin{aligned}
E[X_{ij}X_{ik}] &= E[X_{ij}]E[X_{ik}] \\
&= \frac{1}{n} \cdot \frac{1}{n} \\
&= \frac{1}{n^2} .
\end{aligned}$$

با جایگزینی این مقادیر امید ریاضی در تساوی (۳,۸) به دست می آوریم

$$\begin{aligned}
E[n_i^2] &= \sum_{j=1}^n \frac{1}{n} + \sum_{1 \leq j \leq n} \sum_{\substack{1 \leq k \leq n \\ k \neq j}} \frac{1}{n^2} \\
&= n \cdot \frac{1}{n} + n(n-1) \cdot \frac{1}{n^2} \\
&= 1 + \frac{n-1}{n} \\
&= 2 - \frac{1}{n},
\end{aligned}$$

که تساوی (۸,۲) را ثابت می کند.

با استفاده این امید های ریاضی در تساوی (۸,۱)، ما نتیجه می گیریم که زمان منتظره مرتب سازی سطلی  $Q(n + n \cdot O(1/n))$  است. بنابراین، کل الگوریتم مرتب سازی سطلی در زمان خطی منتظره اجرا می شود.

حتی اگر ورودی از یک توزیع یکنواخت به دست نیامده باشد، مرتب سازی سطلی ممکن است در زمان خطی اجرا شود. تا زمانی که ورودی این خصوصیت که مجموع مربعات سایز سطل ها نسبت به تعداد کل عناصر خطی است، تساوی (۸,۱) به ما می گوید که مرتب سازی سطلی در زمان خطی اجرا می شود.

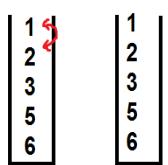
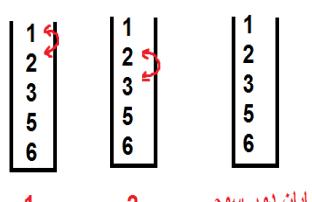
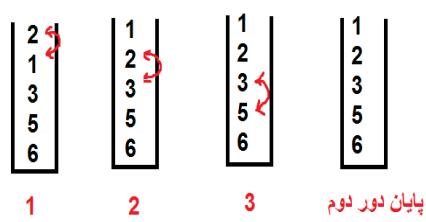
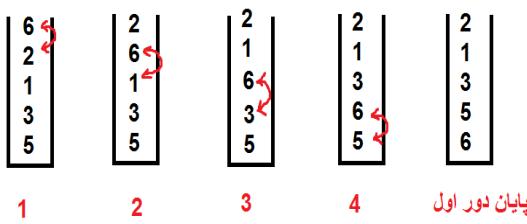
تمرین : توضیح دهید که چرا هزینه ای زمانی مرتب سازی سطلی در بدترین حالت  $n^{8/2}$  است؟ سپس پیشنهاد دهید که با چه تغییر ساده در الگوریتم میتوان هزینه ای زمانی حالت متوسط را همان مقدار قبلی نگه داشت ولی هزینه ای زمانی بدترین حالت بجای تتابی  $n^{8/2}$  تتابی  $n \log n$  شود؟

پاسخ : بدترین هزینه ای زمانی در مرتب سازی سطلی هنگامی اتفاق میفتد که برخلاف فرض ما، داده ها بصورت منظم بین سطل ها پخش نشده باشند و برای مثال کل داده ها فقط در یک سطل بیفتند که در آن صورت در مرحله ای که مرتب سازی درجی اتفاق می افتد داده ها باید با  $O(n^{8/2})$  مرتب شوند.

تغییر ساده ای در الگوریتم که میتواند هزینه ای زمانی حالت متوسط را نگه دارد و در عین حال هزینه ای بدترین حالت را کاهش دهد، این است که بجای مرتب سازی درجی که در بدترین حالت  $O(n^{8/2})$  است از مرتب سازی ادغامی برای مرتب کردن سطل ها کمک بگیریم که در بدترین حالت  $O(n \log n)$  است.

## (Bubble Sort)- مرتب سازی حبابی

مرتب سازی حبابی یک الگوریتم مرتب سازی است که لیست را پیمایش میکند و اگر عنصری از عنصر بعد از خود بزرگتر بود جای آن را با هم عوض میکند و در پیمایش بعد یکی از تعداد عناصر ارایه که بررسی میشوند کم میکند. به ازای هر عنصر بجز عنصر اخر، یک پیمایش صورت میگیرد (زیرا آخرین عنصر خود به خود مرتب



میشود) که در آن ممکن است جابجایی انجام شود یا

نشود. در این الگوریتم آرایه را به صورت عمودی تصور

میکنیم و این الگوریتم از آن رو حبابی نامیده میشود که

هر عنصر با عنصر زیری خود سنجیده شده و در صورتی که از آن بزرگ‌تر باشد جای خود را به آن می‌دهد و این کار

ادامه می‌یابد تا کوچک‌ترین عنصر به بالای آرایه برسد و

سایر عناصر نیز به ترتیب در جای خود قرار گیرند (این عمل همانند حرکت حباب به بالای مایع است. به این

الگوریتم، مرتب سازی سنگی نیز گفته میشود. چون

عناصر بزرگ‌تر به سمت پایین آرایه حرکت میکند)

این الگوریتم از آن رو که برای کار با عناصر آن‌ها را با یکدیگر مقایسه میکند، یک مرتب سازی بر مبنای مقایسه است.

همچنین چون میزان استفاده از حافظه‌ی اضافی در آن ۰ است پس مرتب سازی حبابی در جا نیز است.

با فرض داشتن  $n$  عضو، در بدترین حالت  $\frac{n(n-1)}{2}$  عمل لازم خواهد بود.

یک آرایه با مقادیر "۶، ۵, ۴, ۳, ۲, ۱" را در نظر بگیرید. آن را با استفاده از مرتب سازی حبابی مرتب می‌کنیم.

$$(6, 5, 4, 3, 2, 1) \rightarrow (2, 6, 5, 4, 3, 1)$$

$$(2, 6, 5, 4, 3, 1) \rightarrow (1, 2, 6, 5, 4, 3)$$

$$(1, 2, 6, 5, 4, 3) \rightarrow (1, 2, 3, 4, 5, 6)$$

(۲, ۱, ۳, ۶, ۵)  $\rightarrow$  (۲, ۱, ۳, ۵, ۶)

(۲, ۱, ۳, ۵, ۶)  $\rightarrow$  (۱, ۲, ۳, ۵, ۶)

حالا آرایه مرتب شده است. همانطور که در شکل مشاهده میشود بعد از دومین بررسی در دور دوم جابجایی صورت نگرفته است. پس عملاً عملیات مرتب سازی در آن مرحله پایان یافته است ولی طبق الگوریتم مقایسه کردن عناصر ادامه پیدا کرده است که بیهوده است. پس الگوریتم قابل بهینه سازی است.

```
void BubbleSort(int temp[], int len)
{
    int i, j, item;
    for(i=0;i<len-1;i++)
    {
        for(j=0;j<len-i-1;j++)
        {
            if(temp[j]>temp[j+1])
            {
                item=temp[j];
                temp[j]=temp[j+1];
                temp[j+1]=item;
            }
        }
    }
}
```

با حذف مساوی مشخص شده از کد، الگوریتم stable میشود. پس مرتب سازی حبابی stable است.

همانطور که مشاهده میشود هزینه‌ی کد در بدترین حالت بهترین حالت و حالت متوسط برابر  $O(n^2)$  است. با تغییراتی جزیی در کد الگوریتم میتوان هزینه‌ی آن را در بهترین حالت به  $O(n)$  کاهش داد. اگر از مرحله‌ای به بعد هیچ جابجایی صورت نگرفت میتوان نتیجه گرفت که تمام عناصر مرتب شدند پس میتوان الگوریتم را متوقف کرد.

```

void BubbleSort(int temp[], int len)
{
    int i, j, item;
    bool stop=false;
    while (!stop && i<len-1)
    {
        stop=true;
        for(j=0;j<len-i-1;j++)
        {
            if(temp[j]>temp[j+1])
            {
                item=temp[j];
                temp[j]=temp[j+1];
                temp[j+1]=item;
                stop=false;
            }
        }
        i++;
    }
}

```

## ۶-مرتب سازی درجی (Insertion Sort)

یک الگوریتم مفید برای مرتب کردن تعداد عناصرهای کم است که مبنای آن مقایسه است. ایده‌ی این مرتب سازی به این صورت است که در هر مرحله عنصر  $k$ ام به  $k-1$  عنصر مرتب شده‌ی قبل از خود insert می‌شود. Insert شدن به این معناست که عنصر را با هر کدام از عناصر مقایسه می‌کند و زمانی که جای مناسب آن را پیدا کرد آن را به عناصر وارد می‌کند. مزیت مرتب‌سازی درجی این است که برای تشخیص مکان درست عنصر  $(k+1)$ ام، فقط عناصر مورد نیاز را بررسی می‌کند.



شبیه کد ما یک آرایه  $A[1...n]$  به عنوان پارامتر می‌گیرد.  $A$  یک رشته به طول  $n$  است که باید مرتب شود. در کد، تعداد  $n$  عنصر در  $A$  با  $\text{Length}[A]$  مشخص می‌شود. عدهای ورودی در محل مرتب می‌شوند. Sorted در واقع عدها درون آرایه  $A$  مجدداً چیده می‌شوند، با حداکثر یک عدد از آنها که بیرون آرایه ذخیره می‌شود. (حداکثر یک حافظه اضافی) پس این مرتب سازی یک مرتب سازی درجا است. وقتی که اتمام می‌شود آرایه  $A$  دارای رشته خروجی مرتب شده است.

```

Ins_sort(A[·,...,n-1])
{
    for (k=· ;k<Length[A]-1 ;k++)
    {
        key=A[k];
        j=k;
        while (j>· & A[j] >=key)
        {
            A[j+1] = A[j];
            j--;
        }
        A[j+1]=key;
    }
}

```

اگر علامت مساوی مسخّص شده را از کد حذف کنیم ، الگوریتم **stable** می‌شود. پس مرتب سازی درجی است.

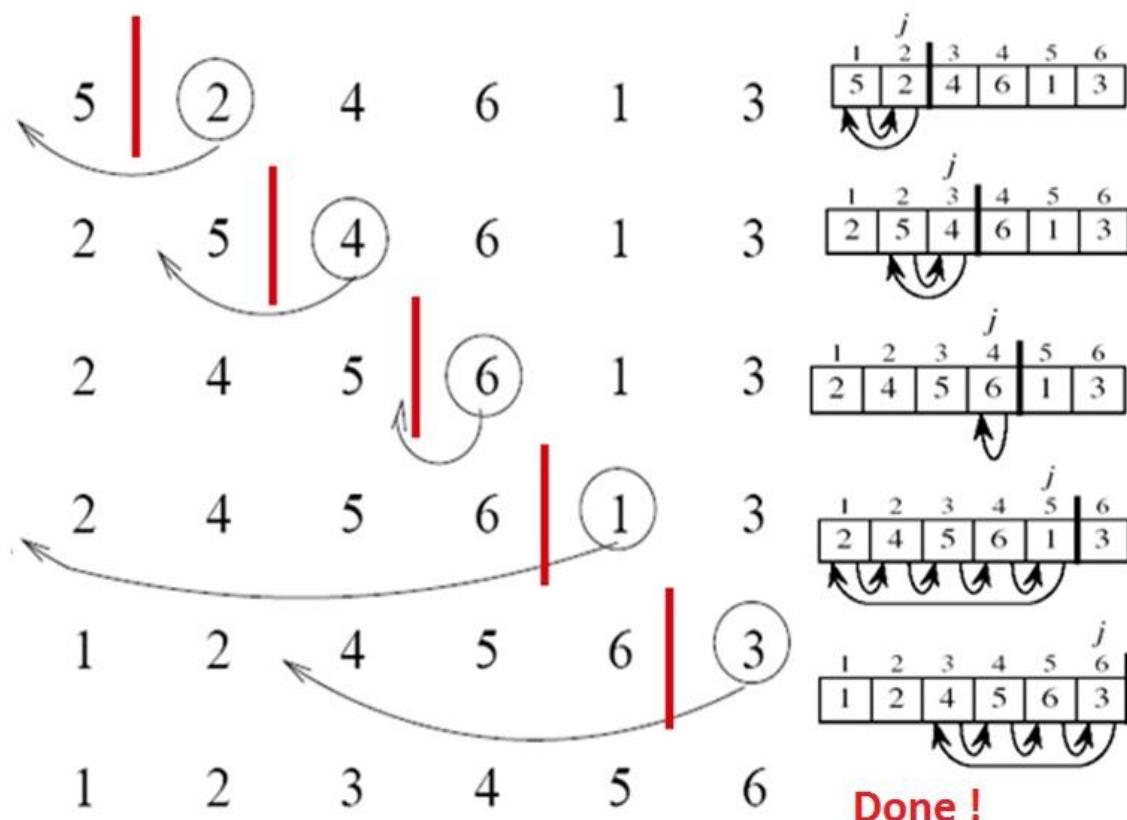
بهترین حالت زمانی است که آرایه از قبل مرتب شده باشد. در این حالت زمان اجرای الگوریتم از  $O(n)$  است: در هر مرحله اولین عنصر باقیمانده از لیست اولیه فقط با آخرین عنصر لیست مرتب شده مقایسه می‌شود. این

حالت بدترین حالت برای مرتب‌ساز سریع (غیرتصادفی و با پیاده‌سازی ضعیف) است که زمان  $O(n^2)$  صرف می‌کند.

بدترین حالت این الگوریتم، زمانی است که آرایه به صورت معکوس مرتب شده باشد. در این حالت هر اجرای حلقه داخلی، مجبور است که تمام بخش مرتب شده را بررسی کرده و انتقال دهد. در این زمان اجرای الگوریتم، مثل حالت متوسط، دارای زمان اجرای  $O(n^2)$  است که باعث می‌شود استفاده از این الگوریتم برای مرتب‌سازی تعداد داده‌های زیاد، غیرعملی شود. گرچه حلقه داخلی مرتب‌ساز درجی، خیلی سریع است و این الگوریتم را به یکی از سریع‌ترین الگوریتم‌های مرتب‌سازی، برای تعداد داده‌های کم (عموماً کمتر از ۱۰)، تبدیل می‌کند. در بدترین حالت و در حالت متوسط هزینه اجرای این تابع درجه ۲ می‌شود و برابر با  $O(n^2)$  است.

## ۵-۷- مرتب‌سازی انتخابی (Selection Sort)

مرتب‌سازی انتخابی یکی از انواع الگوریتم مرتب‌سازی می‌باشد که جزو دسته‌ی الگوریتم‌های مرتب‌سازی مبتنی بر مقایسه است. این الگوریتم دارای پیچیدگی زمانی از درجه  $O(n^2)$  است که به همین دلیل اعمال آن روی مجموعه‌ی بزرگی از اعداد کارا به نظرنمی‌رسد و به طور عمومی ضعیفتر از نوع مشابهش



که مرتب‌ساز درجی است عمل می‌کند. این مرتب‌سازی به دلیل سادگی اش قابل توجه است. این الگوریتم اینگونه عمل می‌کند: ابتدا کوچکترین عنصر مجموعه اعداد را یافته با اولین عدد جایجا می‌کنیم. سپس دومین عنصر کوچکتر را یافته با دومین عدد جایجا می‌کنیم و این روند را برای  $n-1$  عدد اول تکرار می‌کنیم. پس در کل  $n-1$  بار عملیات *minimum* گیری انجام می‌شود. (عنصر  $n$  ام خود به خود در جای مناسب قرار می‌گیرد). در حقیقت در هر مرحله ما لیست خود را به دو بخش تقسیم می‌کنیم. زیرلیست اول که قبلًا مرتب کردہ‌ایم و سایر اعضای لیست که هنوز مرتب نشده‌است.

برای مثال آرایه‌ی رو برو را با این الگوریتم مرتب کنیم.

۲	۸	۴	۱	۷
---	---	---	---	---

در مرحله‌ی اول کل لیست بررسی شده و کوچکترین عنصر با عنصر اول لیست جایجا می‌شود.

۱	۸	۴	۲	۷
---	---	---	---	---

در مرحله‌ی بعد پیمایش از عنصر دوم لیست شروع شده و کوچکترین عنصر با عنصر دوم لیست جایجا می‌شود. پیمایش از عنصر دوم شروع می‌شود زیرا در مرحله‌ی قبل عنصر اول به عنوان کوچکترین عنصر انتخاب شده بود.

۱	۲	۴	۸	۷
---	---	---	---	---

همین روند ادامه پیدا می‌کند تا تمامی عناصر در جای درستی قرار گیرند.

۱	۲	۴	۷	۸
---	---	---	---	---

به این ترتیب آرایه مرتب می‌شود.

شبه کد زیر کارکرد این الگوریتم را نشان میدهد.

Void selection\_sort (A[·,...,n-1])

{

for (k=·; k<n-2; k++)

{

```

min=A[k];
minidx=k;
for (j=k+1;j<n-1;j++)
{
    if (A[j]<=min)
    {
        minidx=j;
        min=A[j];
    }
}
Swap(A[minidx],A[k]);
}
}

```

هزینه‌ی این الگوریتم از رابطه‌ی روبرو محاسبه می‌شود:

$$T(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 2 + 1 = n(n - 1) / 2$$

که از مرتبه  $\Theta(n^2)$  است.

پیچیدگی زمانی اجرای این الگوریتم بر اساس محاسبات فوق در بدترین حالت  $\Theta(n^2)$  است. با توجه به قطعه کد نوشته شده، ترتیب عناصر تغییری در عملکرد آن اینجا نمی‌کند. یعنی این الگوریتم برای داده‌های کاملاً مرتب، نامرتب تصادفی و مرتب معکوس به یک ترتیب عمل کرده و تمام مقایسه‌های محاسبه شده در رابطه فوق را انجام می‌دهد. بنابراین پیچیدگی این الگوریتم در بهترین حالت و حالت منوسط نیز  $\Theta(n^2)$  است.

مرتبسازی انتخابی یک روش مرتبسازی درجا است. یعنی عملیات مرتبسازی به در داخل خود لیست و بدون نیاز به حافظه کمکی بزرگ انجام می‌گیرد.

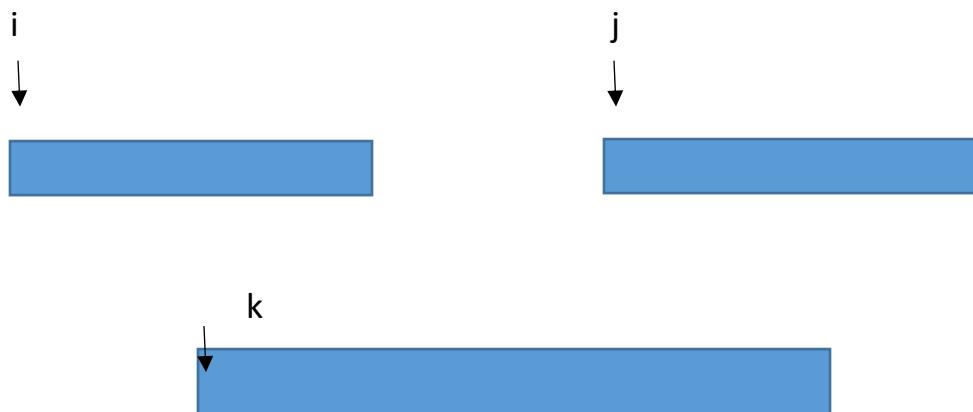
در پیاده‌سازی مرتب‌سازی انتخابی به روش فوق، اگر دو عنصر با مقدار کمینه داشته باشیم، دومی انتخاب شده و به ابتدای لیست منتقل می‌شود. در نتیجه ترتیب آنها به هم می‌خورد. بنابراین این پیاده‌سازی روش پایدار نیست. در روش پایدار ترتیب عناصر با مقدار یکسان تغییر نمی‌کند. اما اگر در مقایسه عناصر آرایه به جای  $<$  از  $>$  استفاده کنید، مرتب‌سازی پایدار خواهد شد.

## ۵-۸- مرتب‌سازی ادغامی (Merge Sort)

روش مرتب‌سازی ادغامی (Merge Sort) یک روش مرتب‌سازی مبتنی بر مقایسه است که از الگوریتمی بازگشتی پیروی می‌کند.

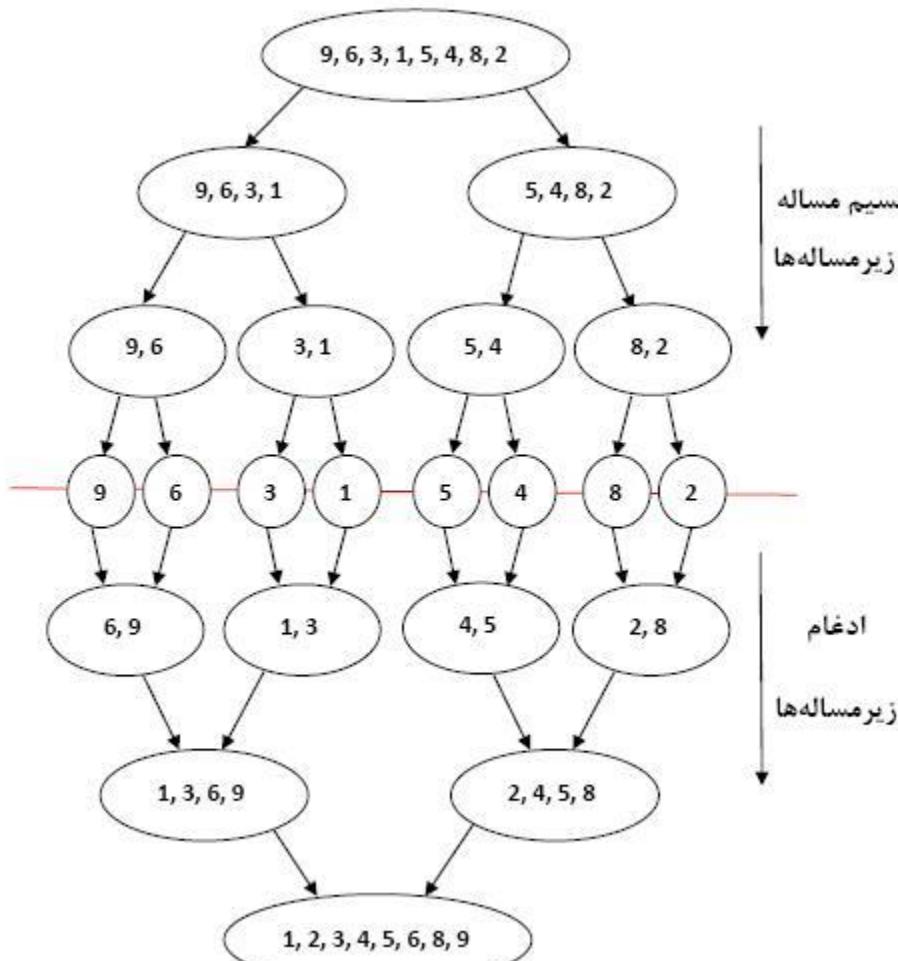
این مرتب‌سازی به این صورت عمل می‌کند که آرایه‌ی اولیه را به زیر‌آرایه‌هایی با اندازه‌ی برابر تقسیم می‌کند و هر کدام از این زیر‌آرایه‌ها را به صورت جداگانه مرتب می‌کند و در آخر دو زیر‌آرایه را با هم ادغام می‌کند و همین عملیات را تکرار می‌کند تا تمامی عناصر در جای خود قرار گیرند.

ادغام دو زیر‌آرایه به این صورت است که آرایه‌ای کمکی به اندازه‌ی مجموع تعداد اعضای ۲ زیر‌آرایه می‌گیریم (پس الگوریتم درجا نیست). اشاره گر  $A$  به ابتدای زیر‌آرایه‌ی اول و اشاره گر  $J$  به ابتدای زیر‌آرایه‌ی دوم و اشاره گر  $K$  به ابتدای ارایه‌ی کمکی اشاره می‌کند.

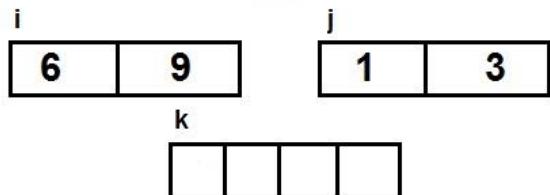


در هر مرحله عددی که اشاره گر  $A$  و اشاره گر  $Z$  به آن ها اشاره میکنند با یکدیگر مقایسه میشوند و هر کدام کوچکتر بود (برای چینش صعودی) را در خانه ای از آرایه  $i$  کمکی که  $k$  به آن اشاره میکند قرار میدهیم و اشاره گر  $k$  و همچنین اشاره گری که خانه  $i$  آن به آرایه  $i$  کمکی اضافه شده است را یک خانه به جلو میبریم. این کار را تا زمانی ادامه میدهیم یکی از اشاره گرهای  $A$  یا  $Z$  به پایان آرایه  $i$  خود برسند.

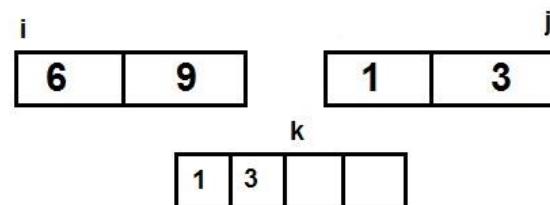
در آن زمان کل عناصر باقی مانده از آرایه  $i$  دیگر را در آرایه  $i$  کمکی، به همان ترتیب قرار میدهیم. به مثال روپرتو دقت کنید:



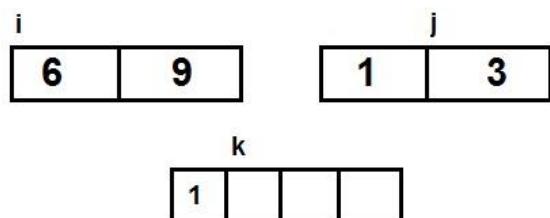
ادغام کردن دو زیرآرایه  $i$  (۶ و ۹) و (۳ و ۱) به صورت زیر انجام میگیرد:



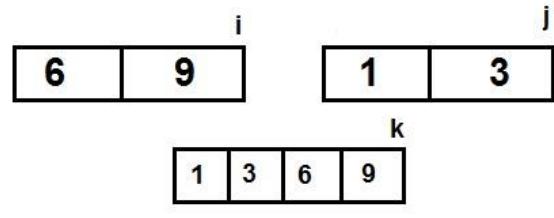
شکل 1



شکل 3



شکل 2



شکل 4

ادغام بقیه ی زیر آرایه ها نیز به همین صورت است.

شبه کد این مرتب سازی به صورت زیر است:

```
merge_sort(int A[], int l , int r )
```

```
{
```

```
if (l ≥ r )
```

```
    return;
```

```
m=  $\frac{(l+r)}{2}$ ;
```

```
merge_sort(A,l,m);
```

```
merge_sort(A,m+1,r);
```

```
merge(A,l,m,r);
```

```
}
```

```
merge(int A[] , int l , int m , int r )
```

```
{
```

```

n=r-l+1;
allocate B with size n;
i=l;
j=m+1;
for(k= ; k<n-1 ; k++)
{
    if (i>m)
        copy right part;
    else if (j> r)
        copy left part;
    else if (A[i] <= A[j])
        B[k++]=A[i++];
    else if (A[i] > A[j] )
        B[k++]=A[j++];
}
}

```

اگر در این الگوریتم در حالت تساوی، همواره اولویت به اندیس آ داده شود، مرتب سازی پایدار میشود.

هزینه ای از الگوریتم مرتب سازی ادغامی را به کمک رابطه بازگشتی زیر بیان می کنیم:

$$T(n) = \begin{cases} \theta(1), & \text{if } n = 1 \\ 2 T\left(\frac{n}{2}\right) + \theta(n), & \text{if } n > 1 \end{cases}$$

که در نهایت به  $T(n) = \theta(n \lg n)$  منجر خواهد شد.

پیچیدگی زمانی اجرای الگوریتم در تمامی حالات  $\Theta(n \log n)$  است. چرا که این الگوریتم تحت هر شرایطی آرایه را به دو قسمت کرده و مرتبسازی را انجام می‌دهد.

پیچیدگی حافظه مصرفی بستگی به روش پیاده‌سازی مرحله ادغام دارد، که تا  $O(n^2)$  افزایش می‌یابد. پیاده‌سازی درجای این الگوریتم حافظه مصرفی مرتبه  $\Theta(1)$  دارد. اما اجرای آن در بدترین حالت زمانبر است.

الگوریتم مرتبسازی ادغامی با پیاده‌سازی فوق یک روش پایدار است. چنان‌الگوریتمی ترتیب عناصر با مقدار یکسان را پس از مرتبسازی حفظ می‌کند.

## ۱- مرتبسازی سریع<sup>۱</sup>

مرتبسازی سریع یک الگوریتم مرتبسازی است که در بدترین حالت  $\Theta(n^2)$  است. با وجود کندی بدترین حالت مرتبسازی سریع، مرتبسازی سریع اغلب بهترین انتخاب عملی است، چون در حالت میانگین به طور قابل توجهی کارا است؛ زمان مورد انتظار برای اجرا  $\Theta(n \lg n)$  است. مرتبسازی سریع همچنین مزیت مرتب کردن درجا را هم دارد. ضریب ثابت  $n \lg n$  کاملاً کوچک است و این الگوریتم برای حافظه‌های خارجی و موازی نیز کارا می‌باشد.

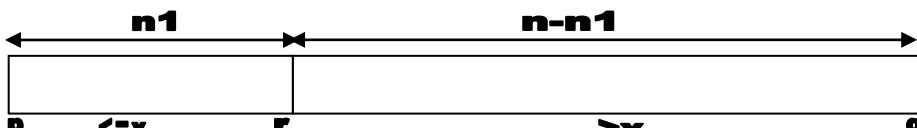
### ۱-۱- توصیف مرتبسازی سریع

مرتبسازی سریع، مانند مرتبسازی ادغامی<sup>۲</sup>، بر اساس مدل Divide & Conquer بنا شده است. در ادامه سه مرحله لازم برای مرتبسازی یک آرایه‌ی  $A[p \dots q]$  آورده شده است.

گونه‌ای که تمامی عناصر زیرآرایه‌ی اول کوچکتر یا مساوی  $X$  و تمامی عناصر زیرآرایه دوم بزرگتر از  $X$  هستند.

Conquer : مرتب کردن زیرآرایه‌های  $A[r+1 \dots q]$  و  $A[p \dots r]$  با فراخوانی بازگشته مرتبسازی سریع.

Combine : چون زیرآرایها درجا مرتب شده‌اند نیازی به ادغام آن‌ها نیست و کل آرایه‌ی  $A[p \dots q]$  اکنون مرتب است.



شکل ۱ - دو ناحیه‌ی به وجود آمده توسط  $A[p \dots q]$  روی آرایه‌ی  $A[p \dots q]$  partition کوچکتر یا مساوی  $x$  و مقادیر  $A[r+1 \dots q]$  بزرگتر از  $x$  هستند.

<sup>۱</sup> -Quick Sort

<sup>۲</sup> -Merge Sort

به طور کلی می‌توان چهار حالت را برای انتخاب محور<sup>۱</sup> X نام برد:

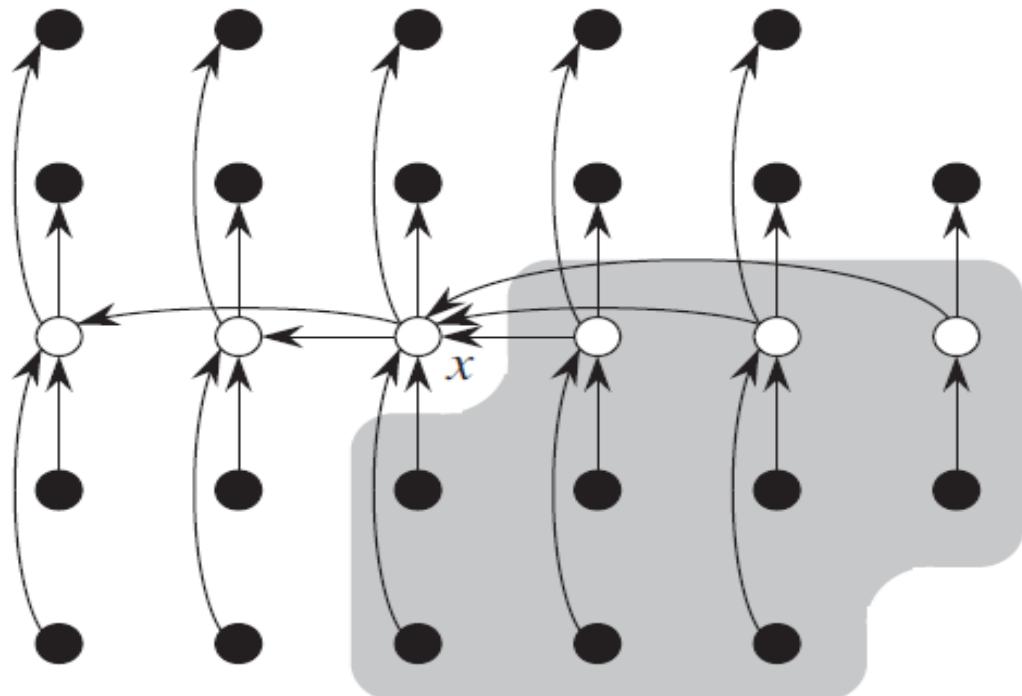
۱. مرتب‌سازی سریع ساده: در این روش محور X اولین عنصر آرایه است.
۲. مرتب‌سازی سریع تصادفی: در این روش محور X یکی از عناصر آرایه به صورت تصادفی است.
۳. مرتب‌سازی سریع میانگین: در این روش محور X میانگین عناصر آرایه است.
۴. مرتب‌سازی سریع خطی زمانی.

می‌خواهیم با  $O(n)$  یک محور انتخاب کنیم که در بدترین حالت نیز مرتب‌سازی سریع هزینه زمانی  $O(nlgn)$  داشته باشد.

۱. آرایه را به  $\lceil n/5 \rceil$  دسته ۵ تایی تقسیم می‌کنیم.

۲. هرگروه را با insertion sort مرتب می‌کنیم و میانه ۵ عنصر انتخاب شود،  $\lceil n/5 \rceil$  میانه داریم.

۳. میانه این  $\lceil n/5 \rceil$  عدد محاسبه شود. عدد انتخاب شده محور است.



شکل ۲- انتخاب محور X

<sup>۱</sup> -Pivot

نتیجه می‌گیریم که:

$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3n}{10} - 6.$$

ملاحظه می‌شود که یک نسبت از  $n$  به دست آمد. (۳/۱۰)

ملاحظه می‌شود که هزینه زمانی آن برابر است با:

$$T(n) = (n/5) * O(1) + O(n) = O(n)$$

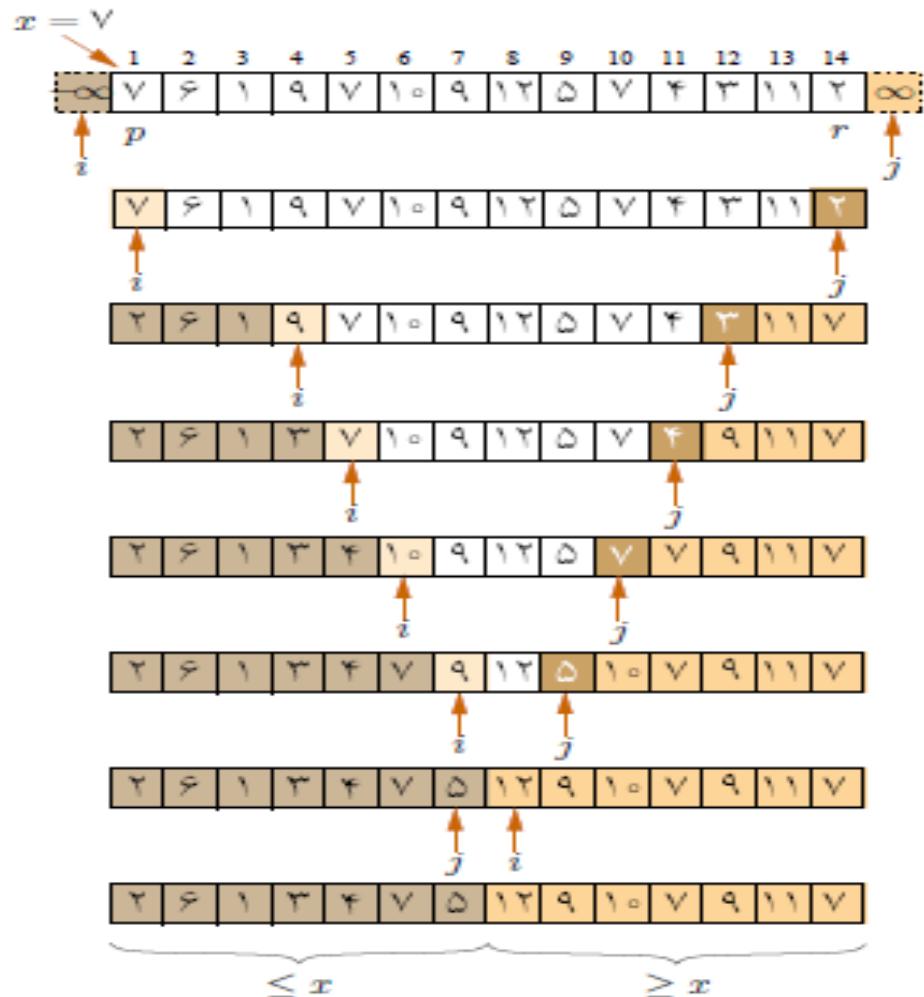
پروسه‌ی زیر نحوه‌ی پیاده‌سازی مرتب‌سازی سریع را برای آرایه‌ی  $A[p...r]$  نشان می‌دهد:  
اندیس اخرين عنصر سمت چپ را بر می‌گرداند.

```
QuickSort(A, p, r)
{
    if (p >= r)    return;
    q = Partition(A, p, r);
    QuickSort(A, p, q);
    QuickSort(A, q+1, r);
}
```

پروسه‌ی زیر نحوه‌ی پیاده‌سازی تابع Partition را برای آرایه‌ی  $A[p...r]$  و محور  $x$  نشان می‌دهد.

ساده‌ترین روش استفاده از یک ارایه کمکی به طول  $n$  می‌باشد (درجا نیست). روش بهتر استفاده از دو پوینتر است.

```
Partition (A, p, r) {
    x = get_pivot(A, p, r);
    i=p;
    j=r;
    while (j >= i) {
        while (A[i] <= x && i <= j)
            i++;
        while (A[j] > x && i <= j)
            j--;
        if (i < j)
            swap (A[i], A[j]);
    }
    return j;
}
```



شکل ۳- مثالی از partition

مثال: ارایه زیر را با انتخاب  $x=8$  به عنوان محور partition کنید.

۱۵	۱۰	۷	۳	۸	۳	۱۰	۴
----	----	---	---	---	---	----	---

حل:

۴	۱۰	۷	۳	۸	۳	۱۰	۱۵
۴	۳	۷	۳	۸	۱۰	۱۰	۱۵

## ۱-۲- کارایی مرتبسازی سریع

زمان اجرای مرتبسازی سریع به نحوه partition کردن بستگی دارد که این به نوبه‌ی خود به این که کدام عنصر به عنوان محور برای partition کردن انتخاب شده بستگی دارد.

### ۱-۲-۱- حالت کلی

$$T(n) = T(n_1) + T(n-n_1) + \Theta(n)$$

### ۱-۲-۲- بدترین حالت

بدترین حالت زمانی اتفاق می‌افتد که  $x$  بزرگترین یا کوچکترین عنصر باشد آنگاه  $n_1=1$  یا  $n_1=n-1$  است.

$$\begin{aligned} T(n) &= T(n-1) + T(1) + \Theta(n) \\ &= T(n-1) + \Theta(n) \\ T(n) &= \Theta(n^r) \end{aligned}$$

### ۱-۲-۳- بهترین حالت

بهترین حالت زمانی اتفاق می‌افتد که با partition اندازه‌ی هیچ‌کدام از زیرآرایه‌ها از  $n/2$  بیشتر نشود؛ یعنی بهتر است محور  $x$  میانه‌ی اعداد داخل آرایه انتخاب شود. در این حالت مرتبسازی سریع، بسیار سریع‌تر عمل می‌کند.

$$T(n) \leq 2T(n/2) + \Theta(n)$$

$$T(n) = O(n \lg n)$$

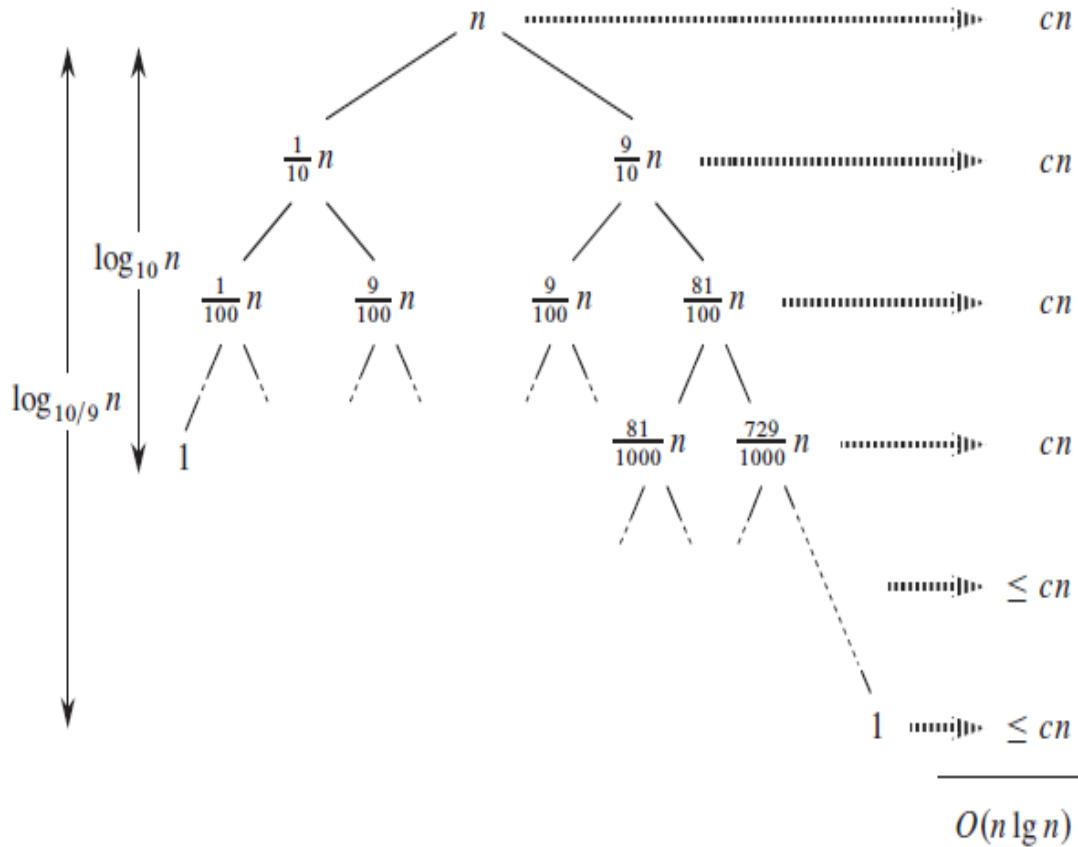
### ۱-۲-۴- حالت متوازن

حالت متوسط زمان اجرای مرتبسازی سریع به بهترین حالت بسیار نزدیک‌تر است تا به بدترین حالت.

فرض کنید، به عنوان مثال، عمل partition همیشه زیرآرایه‌هایی به نسبت ۱ به ۹ تولید می‌کند که در نگاه اول به نظر نامتوازن می‌آید.

$$T(n) \leq T(9n/10) + T(n/10) + cn$$

$$T(n) = O(n \lg n)$$



شکل ۴- درخت بازگشتی برای مرتبسازی سریع که در آن partition دو قسمت به نسبت ۹ به ۱ تقسیم می‌کند که زمان اجرا  $O(n \lg n)$  را نتیجه می‌دهد.

در واقع حتی تقسیم به نسبت ۱ به ۹۹ هم منجر به زمان اجرای  $O(n \lg n)$  خواهد شد. علت آن است که تقسیم با نسبت ثابت منجر به یک درخت بازگشت با ارتفاع  $\Theta(\lg n)$  می‌شود که هزینه‌ی هر سطح  $O(n)$  است. بنابراین زمان اجرا  $O(n \lg n)$  خواهد بود.

مرتبسازی سریع در جایی که تقسیم به نسبت باشد خیلی بهینه است، ولی اگر به دو بخش با اختلاف عددی تقسیم شود بهینه نیست.

### ۱-۳- درجا و پایدار بودن مرتب سازی سریع

پایدار	دربا
بلی	خیر
خیر	بلی

مثال:

اعداد زیر را به روش مرتبسازی سریع به صورت صعودی مرتب کنید، اولین عنصر هر آرایه را به عنوان محور در نظر بگیرید.

۸۳۰-۲۶۴-۵۳۸-۱۲۲-۲۹۳-۸۹۲-۱۲۳-۲۳۷-۹۱۰

حل:

(محور)



مثال:

کدام یک از الگوریتم های مرتبسازی زیر در شرایطی که آرایه از همان ابتدا به صورت صعودی مرتب شده باشد، بدترین زمان اجرا را دارد؟

Insertion sort	Heap sort	Quick sort	Merge sort
----------------	-----------	------------	------------

. ۱-

هزینه‌ی الگوریتم‌های Merge Sort و Heap Sort در بدترین و بهترین حالت، هر دو،  $O(n \log n)$  می‌باشد.

هزینه‌ی الگوریتم Insertion Sort هنگامی که آرایه‌ی ورودی از همان ابتدا Sort شده باشد، در بهترین حالت و  $O(n)$  است.

هزینه‌ی Quick Sort نیز در این حالت به بدترین حالت خود، یعنی  $O(n^2)$  می‌رسد. زیرا در حالتی که آرایه‌ی ورودی از همان ابتدا مرتب شده باشد، عمل Partitioning آرایه را به دو قسمت  $n-1$  تایی و ۰ تایی تقسیم می‌کند و در نتیجه هزینه‌ی زمانی از عبارت  $T(n) = T(n-1) + \theta(n)$  مربوط به هزینه‌ی عمل Partitioning به دست می‌آید.

(است).

پس در بین الگوریتم‌های داده شده پاسخ صحیح Quick Sort می‌باشد.

مثال:

در یک آرایه به طول  $n$  عددی وجود دارد که بیشتر از  $\frac{n}{2}$  بار در این آرایه تکرار شده است. الگوریتمی ارائه دهید که این عدد را پیدا کند. الگوریتم شما باید از مرتبه زمانی  $O(n)$  و مرتبه حافظه  $O(1)$  باشد.

حل:

روی اعداد الگوریتم مرتب‌سازی سریع را انجام می‌دهیم. با این تفاوت که هر دفعه دسته‌ی کوچکتر را دور می‌ریزیم و الگوریتم را روی دسته‌ی بزرگتر انجام می‌دهیم، تا جایی که تمام عناصر موجود در آرایه یکی شوند. این عنصر، عنصر مورد نظر است.

مثال:

پایداری الگوریتم مرتب‌سازی سریع را برای پیاده‌سازی استاندارد آن بررسی کنید.

حل:

Quick Sort است. مثلاً حالتی را در نظر بگیرید که جابجایی نهایی در مرحله‌ی `pivot, partition` را از انتهای راست به وسط آرایه می‌آورد و عنصری را جابجا می‌کند که ممکن است در اثر این جابجایی، ترتیب این عنصر و عناصر برابر با آن به هم بروزد.

مثال:

نشان دهید چگونه می‌توان هر الگوریتم مرتبسازی را به صورت بهینه تغییر داد به گونه‌ای که پایدار شود و این تغییرات چه مقدار زمان و فضای اضافه نیاز دارد.

حل:

همه‌ی این الگوریتم‌ها می‌توانند با شمای زیر Stable شوند، اگر به جای مرتب کردن آرایه‌ی  $a_1, a_2, \dots, a_n$  آرایه‌ی  $a_j, a_i = a_j$  در خروجی پیشی می‌گیرد، اگر و تنها اگر  $j > i$  باشد.  
این روش یک مقدار فضای اضافی ثابت به ازای هر عنصر و همچنین یک مقدار زمان ثابت هم به ازای هر مقایسه می‌گیرد.  
( $a_{1,1}, a_{2,2}, \dots, a_{n,n}$ )

و حالا که هر عنصر دو کلید دارد، هنگامی که  $a_j = a_i$ ، از  $a_j$  در خروجی پیشی می‌گیرد، اگر و تنها اگر  $j > i$  باشد.  
این روش یک مقدار فضای اضافی ثابت به ازای هر عنصر و همچنین یک مقدار زمان ثابت هم به ازای هر مقایسه می‌گیرد.  
پس زمان یا فضای asymptotic را تغییر نمی‌دهد. (در بدترین حالت زمان و فضای لازم را دو برابر می‌کند.)

تمرین:

داده‌های زیر را به کمک الگوریتم Quick sort مرتب کنید.

۵-۳-۱-۹-۸-۲-۴-۷

تمرین:

در مورد زمان اجرای partition که روی زیرآرایه‌ای به اندازه  $\Theta(n)$  است مختصرًا بحث کنید.

تمرین:

همانطور که ادعا شد با استفاده از روش جایگذاری ثابت کنید که  $T(n) = \Theta(n^2)$  جواب رابطه بازگشتی  $T(n) = T(n-1) + \Theta(n)$  است.

تمرین:

Quick sort را طوری تغییر دهید که آرایه را ب ترتیب غیرصعودی مرتب کند.

تمرین:

زمان اجرای Quick sort هنگامی که همه عناصر آرایه یک مقدار دارند چیست؟

تمرین:

نشان دهید که زمان اجرای Quick sort هنگامی که همه عناصر آرایه  $A$  متفاوت بوده و با ترتیبی نزولی ذخیره شده‌اند،  $\Theta(n^2)$  است.

تمرین:

یک ورودی مثال بزنید که مرتبسازی سریع روی آن نیازمند  $\Omega(n^2)$  مقایسه باشد. محور این مرتبسازی را میانه‌ی عنصرهای نخستو میانی و پایانی دنباله بگیرید.

## ۴- الگوریتم k-selection

الگوریتم k-selection کامین کوچکترین عنصر از یک آرایه است. از این الگوریتم برای پیدا کردن مینیمم ( $k=0$ )، ماکزیمم ( $k=n$ ) و میانه<sup>۱</sup> ( $k=n/2$ ) استفاده می‌شود.

### ۱-۲- یافتن k امین عنصر در $O(n)$

محور را طوری انتخاب می‌کنیم تا تضمین کنیم که اندازه دو بخش در بدترین حالت  $O(n)$  هستند. برای انتخاب عنصر مورد نظر می‌توان یکی از کارهای زیر را انجام داد:

۱. ساده ترین راه ابتدا آرایه را مرتب می‌کنیم و سپس اندیس  $k$ ام را برمی‌گردانیم.

۲.  $k$  بار مینیمم گرفت.

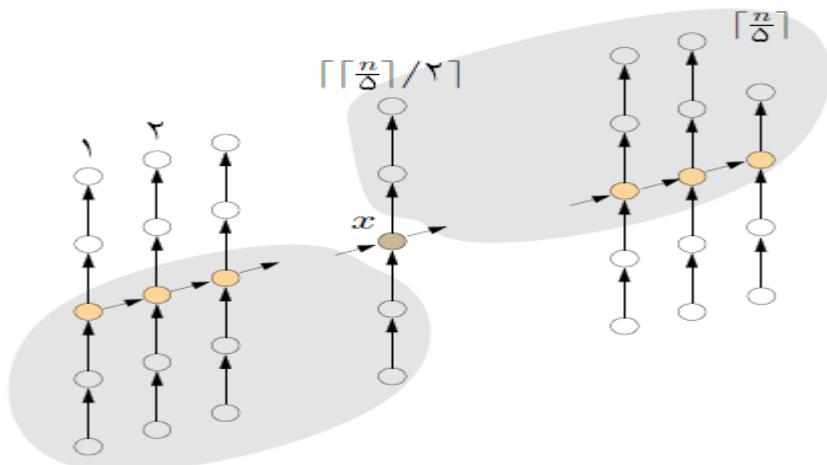
۳. ساخت  $[O(n+k\lg n)]$  و  $k$  بار  $O(n)$  .deleteMin ([در مجموع  $O(k\lg n)$ ]) .minHeap

۴. استفاده از  $O(n)$ ; selection tree با  $O(n)$  مینیمم انتخاب می‌شود، حال  $k$  بار مینیمم می‌گیریم.

۵. استفاده از  $i$ . partition (مقدار بازگشتی تابع  $i$ ). اگر  $i=k$  عنصر  $k$ ام عنصر مورد نظر است.

اگر  $i < k$  بود به صورت بازگشتی،  $k$ امین عنصر سمت چپ را برمی‌گردانیم.

اگر  $i > k$  بود به صورت بازگشتی،  $i-k$ امین عنصر سمت راست را برمی‌گردانیم.



شکل ۶- انتخاب محور  $x$

<sup>۱</sup> -Median

حداقل  $\lceil \frac{n}{5} \rceil$  گروه دارای ۳ عنصر کوچک‌تر از  $x$  هستند.

تعداد عناصر کوچک‌تر از  $x$  حداقل

$$3 \left( \left\lfloor \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rfloor - 2 \right) \geq \frac{3n}{10} - 6$$

است.

به صورت مشابه، تعداد عناصر بزرگ‌تر از  $x$  نیز حداقل  $6 - \frac{3n}{10}$  خواهد بود.

پس، در بد ترین حالت، الگوریتم به صورت بازگشته‌ی بمروری حداً کمتر  $6 + \frac{3n}{10}$  عنصر اعمال خواهد شد.

$$T(n) \leq \begin{cases} \Theta(1) & n \leq \Delta \\ T(\lceil n/5 \rceil) + T(\frac{3n}{10} + \varepsilon) + O(n) & n > \Delta \end{cases}$$

فرض: برای یک  $c$  مفروض و هر  $n \leq \Delta$  داشته باشیم

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(\frac{3n}{10} + \varepsilon) + O(n) \\ &\leq cn/5 + c + \frac{3cn}{10} + \varepsilon c + O(n) \\ &\leq \frac{9}{10}cn + \varepsilon c + O(n) \\ &\leq cn \end{aligned}$$

پروسه‌ی زیر نحوه‌ی پیاده‌سازی الگوریتم k-select را نشان می‌دهد:

kSelect (A, p, r, k)

{

```
if(p==r)      return A[p];
q = Partition (A, p, r);
if(k==q-p+1)  return A[q];
if (q-p+1 > k)
    return kSelect(A, p, q-1, k);
return kSelect (A, q+1, r, k-(q-p+1));
```

}

## **آنالیز الگوریتم k-select**

آنالیز الگوریتم k-select مستقل از  $k$  است.

### **۱-۲-۲ بهترین حالت**

$$n_1 = n/2$$

$$T(n) = O(n) + T(n/2)$$

$$T(n) = O(n)$$

### **۲-۲-۲ بدترین حالت**

$$n_1 = n-1 \text{ یا } n_1 = 1$$

$$T(n) = O(n) + T(n-1)$$

$$T(n) = O(n^2)$$

### **۳-۲-۲ حالت متوسط**

$$T(n) = O(n)$$

مثال:

الگوریتمی ارائه دهید که در کمترین زمان ممکن  $\min$  و  $\max$  آرایه‌ای به طول  $n$  را حساب کند، هزینه‌ی الگوریتم خود را محاسبه کنید و تعداد کمترین مقایسه لازم را به دست آورید.

حل:

آرایه را به دو قسمت تقسیم می‌کنیم و مینیمم و ماکزیمم را در هر دو زیرآرایه حساب کرده و مقایسه می‌کنیم.

$$T(n) = 2T(n/2) + 2, \quad T(2)=1$$

$$T(n) = \frac{3n}{2} - 2$$

$$\left[ \frac{3n}{2} - 2 \right] \text{ مقایسه لازم است.}$$

مثال:

الگوریتمی ارائه دهید که در کمترین زمان ممکن عنصر میانه را در یک لینک لیست به دست آورد. هزینه‌ی الگوریتم خود را محاسبه کنید.(توجه داشته باشید که میانه ممکن است میانگین دو عنصر وسط باشد.)

حل:

اگر تعداد اعضا فرد باشد با استفاده از الگوریتم select-آ می‌توان عنصر وسط را پیدا کرد و اگر زوج بود با دو بار استفاده از این الگوریتم می‌توان دو عنصر وسط را یافت و میانگین آن‌ها را بدست آورد. که این الگوریتم  $O(n)$  طول می‌کشد. استفاده کردن از لینک لیست تاثیری بر سرعت الگوریتم ندارد و تنها به جای جابجایی عناصر باید اشاره‌گرهای آن‌ها را عوض کرد.

تمرین:

نشان دهید که دومین عنصر کوچک از بین  $n$  عضو می‌تواند با  $\lceil lgn \rceil$  مقایسه در بدترین حالت پیدا شود.(راهنمایی: کوچکترین عضو را نیز پیدا کنید.)

تمرین:

نشان دهید  $2 - \lceil \frac{3n}{2} \rceil$  مقایسه در بدترین حالت برای پیدا کردن مینمم و ماکزیمم  $n$  عدد لازم است.(راهنمایی: در نظر بگیرید که چند عدد به صورت بالقوه مینمم یا ماکزیمم هستند و بررسی کنید که چطور یک مقایسه بر این تعداد تاثیر می‌گذارد.)

تمرین:

در الگوریتم  $k$ -select عناصر ورودی به گروههای ۵ عنصری تقسیم می‌شوند. آیا اگر عناصر ورودی به گروههای ۷ عنصری تقسیم شوند، الگوریتم در زمان خطی کار خواهد کرد؟ ثابت کنید که اگر از گروههای ۳ عنصری استفاده شود، الگوریتم در زمان خطی اجرا نمی‌شود.

تمرین:

نشان دهید چگونه مرتبسازی سریع می‌تواند در بدترین حالت در زمان  $O(nlgn)$  اجرا شود.

تمرین:

الگوریتمی با زمان  $O(n)$  شرح دهید که ، با دریافت مجموعه  $S$  با عدد متفاوت و یک عدد صحیح مثبت  $k$ ,  $k \leq n$  که به میانه نزدیکترین هستند را مشخص کند.

تمرین:

فرض کنید  $X[1..n]$  و  $Y[1..n]$  دو آرایه هستند ، که هر یک شامل  $n$  عدد مرتب شده است. الگوریتمی با زمان  $O(lgn)$  ارائه دهید تا میانه تمام  $2n$  عنصر در آرایه  $X$  و  $Y$  را پیدا کند.