ساختمان داده:

علیرضا سلطانی نشان

99 / 07 / 8

فهرست مطالب

توابع بازگشتی:

توابعی که خودشان را با ورودی متفاوت صدا میکنند.

فاکتوریل یک عدد

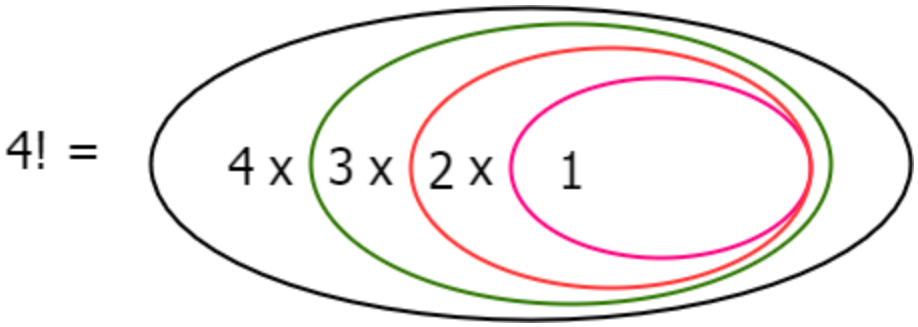
در حالت کلی برای داشتن یک فاکتوریل، به صورت زیر عمل میکنیم.

```
١. const getFact = (n) => {
٢.     var fact = 1
٣.     for(let i = 1; i <= n; i++){
٤.         fact *= i
٥.     }
٦.     return fact
٧. }
٨.
٩. console.log(getFact(4))
```

همانطور که می دانید، برای داشتن فاکتوریل عدد 4 می بایست این عدد را تا عدد یک باهم ضرب کنیم، که در نهایت مانند کد خط بالا عمل میکنیم و به چنین نتیجه ای خواهیم رسید:

```
1. 24
2. // In math 4 x 3 x 2 x 1 = 24
3. [Done] exited with code=0 in 0.129 seconds
```

اما توابعی به نام **توابع بازگشتی** یا (**Recursive Functions**) که همزمان با اجرای خود آن تابع، در کد خطی دیگر صدا میشوند، وجود دارند. توجه داشته باشید که اگه توابع بازگشتی را که نوشته ایم را دیباگ کنیم، متوجه میشیم که برنامه در تابع مربوطه به محض اینکه خودش را صدا میکند، اگه کد خطی بعد از آن باشد صورت نمیگیرد و آن خطی که تابع را صدا زده با ورودی متفاوت صورت میگیرد، تا زمانی که بالاخره، توسط یک شرطی، بقای این حلقه (باطنی) تمام شود، و به آن خطی که تابع خودش را صدا میزد مقداری برگردانده شود، که بوسیله ساختاری که در حافظه استک ساخته شده، نتیجه آن قسمت هایی که صورت نگرفته محاسبه شود.



برنامه نوشته با زبان جاوا اسکریپت:

```
const fact = (a) => {
    if (a <= 1) {
        return 1
    }else {
        var factorial = a * fact(a - 1)
        return factorial
    }
}
```

یا به نحوی بهتر:

function ➔ return n x fact(n – 1)

4! = 4 x fact(3)

fact(3) = 3 x fact(2)

fact(2) = 2 x fact(1)

با توجه به روندی که صورت گرفته، از پایین به بالا آنرا باهم بررسی میکنیم: (از راست به چپ)

همانطور که گفته شد تا زمانی که شرطی وجود نداشته باشد تا این حلقه را بشکند، این حلقه بی نهایت خواهد شد و به نتیجه ای نمی رسید، با توجه به شرط نوشته، اگر ورودی ما خود یک یا کوچک تر از 1 باشد، سریعا عدد یک ریترن خواهد شد، همین که یک Return برای تابع داریم کارمان را آسان میکند، پس از پایین، fact(1) بما یک بر میگرداند که با 2 میشه 1، در نهایت fact(2) برابر با 2 میشود، در مرحله بعد با وجود داشتن جواب، Fact(2) که میدانیم 2 به ما میدهد، 2 در 3 برابر 6 و در مرحله آخر هم همین صورت اتفاق میوفتد که در نهایت به عدد 24 خواهیم رسید.

تابع بازگشتی جمع دو عدد:

```
const p = (a, b) =>{
   if (b == 0) return a
   else return 1 + p(a, b-1)
}
console.log(pluser(3, 5))
```

تریس مسئله بالا: (از پایین به بالا بخوان)

p(3, 5)

p(3, 5) ➔ (3, 4) + 1 ➔ (3, 4) ➔ 7 + 1 =8

p(3, 4) ➔ (3, 3) + 1 ➔ (3, 3) ➔ 6 + 1 = 7

p(3, 3) ➔ (3, 2) + 1 ➔ (3, 2) ➔ 5 + 1 = 6

p(3, 2) ➔ (3, 1) + 1 ➔ (3, 1)➔ 4 + 1 = 5

p(3, 1) ➔ (3, 0) + 1 ➔ (3, 0)➔ 3 + 1 = 4

تابع بازگشتی فیبوناچی:

```
const fib = (a) => {
    if (a == 1 || a == 2)
        return 1
    else {
        const fibonacci = fib(a - 1) + fib(a - 2)
        return fibonacci
    }
}
```

تریس مسئله بالا اگر تعداد نمایش دنباله عدد 5 باشد.

fib(5) ➔ fib(4) + **fib(3)** ➔ fib(4) = 3 + fib(3) = 2 ➔ 5

fib(4) ➔ **fib(3)** + **fib(2)** ➔ fib(3) = 2 + fib(2) = 1 ➔ 3

fib(3) ➔ **fib(2)** + fib(1) ➔ fib(2) = 1 + fib(1) = 1 ➔ 2

ضرب دو عدد با استفاده از تابع بازگشتی

در نوشتن این گونه تابع بازگشتی باید توجه داشته باشیم که نیاز به یک پایان دهنده داریم که بر اساس شرطی منطقی انجام تکرار، متوقف شود، در این تابع در ضرب دو عدد نیاز به دو عدد داریم که برای مثال من از عدد دوم استفاده کرده ام که در هربار یکی از آن کم شود، اگر به صفر رسید، صفر را بر گرداند، که در آخر وقتی آن عدد صفر را با عدد اول خود جمع می‌کنیم و این مراحل را تا مرحله مناسب تکرار کنیم، به ضرب دو عدد می‌رسیم، یا به نوعی دیگر مثلا 3 x 2 در تابع بازگشتی مانند سه بسته دوتایی عمل می‌کند:

```
1.        # Q1
# a * b
def mul(a=8, b=9):
    if b == 0:
        return 0
    else:
        return a + mul(a, b - 1)

print(mul()) # 17
```

انجام عمل توان به وسیله توابع بازگشتی

در انجام این نوع تابع، من عدد توان را به عنوان عامل اصلی و شرط بقا انتخاب کردم که وقتی به عدد کوچکتر از 1 رسید بتواند عدد یک را برگرداند تا در مراحل بعدی به عنوان عامل ضرب استفاده شود:

```
1.        # Q2
# a ^ b
def pow(a=16, b=3):
    if b < 1:
        return 1
    else:
        return a * pow(a, b - 1)


print(pow()) # 4096
```

تابع بازگشتی بنویسید که بتواند حاصل مسئله مقابل را در تعداد 50 بار جمع **رادیکال 6**، بدست بیاورد.

$$\sqrt{6}\ \sqrt{6+\sqrt{6+\sqrt{6+\sqrt{6+\sqrt{6}}}}}$$

در این مسئله هر بار نسبت به عدد وارد شده، تابع خودش را صدا میکند تا زمانی که به مقدار صفر برسد که عدد صفر را برگرداند و بعد از آن صفر با رادیکال 6 آخرین مرحله (اولیه مرحله از پایین) جمع میشود و وارد مراحل بالاتر خود خواهد شد.

```
1.        # Q3
# SQRT recursion
def sqrt(n=6, r=50):
    if r == 0:
        return math.sqrt(n)
    else:
        return math.sqrt(n +sqrt(n, r - 1))


print(sqrt()) # 3.0
```

مسئله زیر را تریس کنید.

```
1.        # Q4
# Tst
def t(x=5, y=2):
    if x <= y or y == 0:
        return x
    elif y == 1:
        return t(x - 1, y) + 1
    else:
        return t(t(y, x), y - 1) + 2
print(t()) #4
```

شرح مسئله بالا:

1. T(5, 2) = T(T(y=2, x=5), y−1=1) + 2 ➔ T(T(2, 5) = 2, 1) ➔ T(2, 1) + 2
2. T(2, 1) = {T(1, 1)] = 1} + 1 ➔ 2
3. T(2, 1) =2 + 2 ➔ 4

جزء صحیح برای عدد 25

L(n) = {

0 n = 1

L([n/2])+1 n>1

} L(25)

```
1.        # Q5
# floor division for recursive def
def fd(n=25):
    if n == 1:
        return 0
    else:
        return fd(n // 2) + 1

print(fd()) #4
```

از پایین به بالا:

I(25) = I([25/2]) +1 ➔ [12.5] ➔ 12 ➔ 3 + 1 = **4**

I(12) = I([12/2]) + 1 ➔ [6] ➔6 ➔ 2 + 1 = 3

I(6) = I([6/2]) +1 ➔ [3] ➔ 1 + 1 = 2

I(3) = I([3/2]) + 1 ➔ [1.5] ➔ 1 ➔0 + 1 = 1

مسئله زیر را تریس کنید:

```
۱. function f (int a, int b){
۲.   if (b==0) return a;
۳.   else return f(b, a%b)
۴. }
```

راه سریعی برای فهمیدن این که این مسئله چه نتیجه ای را میدهد، وجود دارد، اول به قسمت else نگاهی کنیم، هیچ عملیاتی (جمع، تفریق، ضرب، تقسیم، جزء صحیح، رادیکال وغیره) انجام نمی شود، یعنی اگر B به صفر برسد خود عدد a را بر میگرداند، و این یعنی اگر ما عدد 3 و 4 را به ترتیب برای a و b در نظر بگیریم، دوباره سه 3 میرسیم، و این نتیجه یعنی: باقی مانده 3 بر 4 میشود خود 3.

مسئله زیر را بررسی کنید.

(اثبات از پایین به بالا)

$$A(m,n) = \begin{cases} n+1, & m = 0 \\ A(m-1,1), & n = 0 \\ A\big(m-1, A(m,n-1)\big), & outher\ points \end{cases}$$

A(1,3)

A(1, 3) = A(0, A(1, 2)) ➔ A(0, (A(1, 2)=4)) = m=0, n=4 ➔ n+1 = 5

A(1, 2) = A(0, A(1, 1)) ➔ A(0, (A(1, 1)=3)) = m=0, n=3 ➔ n+1 = 4

A(1, 1) = A(0, $A(1,0)$) ➔ A(0, (A(1, 0)=2)) = m=0, n=2 ➔ n+1 = 3

A(1, 0) = **A(0, 1) ➔ n + 1 ➔ 2**

مسئله زیر را تریس کنید. با فرض x = 2 و n = 7 :

```
int f(int x, int n){
   if (n == 1) return x;
   else if (n % 2 == 0) return x * f(x, n/2);
   else return 2 * f(x, n-1);
}
```

f(2, 7) = 2 * f(2, 6) ➔ 2 * 16 = 32

f(2, 6) = 2 * f(2, 3) ➔ 2 * 8 = 16

f(2, 3) = 2 * f(2 ,2) ➔ 2 * 4 = 8

f(2, 2) = 2 * f(2, 1) ➔ 2 * 2 = 4

f(2, 1) = **2**

جست و جو ها

دو نوع جست و جو وجود دارد:

جست و جوی خطی

```
data = [1, 5, 58 , 12, -9, 42, 33, 44, 87, 54]
def linearSearch(ls, sk):
    for i in range(len(ls)):
        if ls[i] == sk:
            return ls[i] , i
    return -1
linearSearch(data, -9) # (-9, 4)
```

جست و جوی دودویی

```
data = [1, 5, 58 , 12, -9, 42, 33, 44, 87, 54]
def binarySearch(ls, sk):
    ls.sort()
    low = 0
    high = len(ls) - 1
    while(low <= high):
        middle = (low + high) // 2
        if sk > ls[middle]:
            low = middle + 1
        elif sk < ls[middle]:
            high = middle - 1
        else: return ls[middle], middle
    return -1
binarySearch(data, 42) # (42, 5)
```

ماتریس اسپارس

$$Sparse\ matrix = \begin{bmatrix} 15 & 0 & 0 & 27 & 0 & -15 \\ 0 & 11 & 3 & 0 & 0 & 0 \\ 0 & 0 & 0 & -60 & 0 & \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 91 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 28 & 0 & 0 & 0 \end{bmatrix}$$

در این نوع ماتریکس در بیشتر ستون و سطر ها دارای مقدار صفر هستیم، در مثال بالا یک ماتریکس 6 در 6 داریم که 36 مقدار دارد که فقط 8 مقدار واقعی در آن غیر از صفر است، برای بر طرف کردن صفر های بیخودی میتوانیم به صورت زیر عمل کنیم:

	سطر	ستون	مقدار غیر صفر
1	6	6	8

2	0	0	15
3	0	3	27
4	0	5	−15
5	1	1	11
6	1	2	3
7	2	3	6
8	4	0	91
9	5	2	28

پس در نتیجه خواهیم داشت که $3x9 = 29 < 6x6 = 36$

مرتب سازی

مرتب سازی انتخابی [1]

```
١. def selectionSorting(arr):
٢.     for i in range(len(arr)-1, 0, -1):
٣.         max_i = 0
٤.         max_v = 0
٥.         for j in range(i):
٦.             if arr[j] > max_v:
٧.                 max_v = arr[j]
٨.                 max_i = j
٩.         if arr[i] < max_v:
١٠.             temp = arr[i]
١١.             arr[i], arr[max_i] = arr[max_i], temp
١٢.     return arr
١٣. selectionSorting([11, 33, 55, 22, 22, 92, 44])
```

11 33 55 22 22 **92** 44

11 33 **55** 22 22 **44** 92

11 33 **44** 22 **22** **55** 92

11 33 22 **22** **44** 55 92

11 **22** **22** **33** 44 55 92

11 22 22 33 44 55 92

در مرتب سازی، انتخابی ما میتوانیم به راحتی از آخرین عدد اقدام به عملیات زیر کنیم:

1) عدد اخر را نگه میداریم.
2) به دنبال بزرگ ترین عدد میرویم.
3) در نهایت عدد بزرگی که پیدا کردیم را با عدد اخر مقایسه میکنیم.
4) اگر عدد اخری که نگهداشتیم کوچک بود، با آن عدد جابه جا میکنیم، در غیر این صورت به سراغ عدد سمت چپی میرویم.

[1] Selection sort

مرتب سازی حبابی[2]

مرتب سازی حبابی در حالت کلی مقایسه هر عدد با همه است، یعنی چی؟

یعنی اینکه برای چند مرحله ما از اولین عدد تا آخرین عدد را با هم مقایسه میکنیم اگر عدد خانه 0 بزرگتر از عدد خانه 1 بود جای این دو خانه را تغییر میدهیم، این کار تا انتهای لیست اتفاق خواهد افتاد، بعد از مرحله اول مرحله دوم دوباره همین کار ادامه پیدا میکنید تا زمانی که حلقه اول به انتها لیست برسد که میتوانیم مطمئن شویم که لیست برای چند مرتبه به صورت حبابی برای مرتب سازی مورد بررسی قرار گرفته است.

اعداد امتحانی :

3 1 7 20 2 6

1 **3 7** 20 2 6

1 3 **7 20** 2 6

1 3 7 **20 2** 6

1 3 7 2 **20 6**

1 3 7 2 6 20

پایان مرحله اول

1 3 7 2 6 20

1 **3 7** 2 6 20

1 3 **7 2** 6 20

1 3 2 **7 6** 20

1 3 2 6 **7 20**

1 3 2 6 7 20

[2] Bubble sort

پایان مرحله دوم

1 **3** 2 6 7 20

1 **3** **2** 6 7 20

1 **2** **3** 6 7 20

پایان مرحله سوم

اما اینجا پایان عملیات نیست بلکه حلقه اول تا زمانی که به انتها برسد این کار را تکرار میکند.

کد الگوریتم مرتب سازی حبابی:

```
۱. def bubbleSort (ls):
۲.     for i in range (len(ls)):
۳.         for j in range (len(ls)-1):
۴.             if ls[j] > ls[j+1]:
۵.                 temp = ls[j]
۶.                 ls[j], ls[j+1] = ls[j+1], temp
۷.     return ls
۸. data_test_b_3 = [3, 1, 7, 20, 2, 6]
```

مرتب سازی درجی[3]

در این نوع مرتب سازی همانطور که از نامش معلوم است، اعداد را از اول با هم بررسی میکنیم، همین که متوجه شدیم عددی در خانه ای نسبت به عدد قبلی خود کوچکتر است با ان خانه جا به جا خواهیم کرد و دوباره این مقایسه را با خانه بغلی آن (از سمت راست به چپ) انجام میدهیم.

مثال، اعداد داده شده زیر را به روش درجی مرتب کنید.

35 51 27 85 66 23

35 **51** 27 85 66 23

35 51 **27** 85 66 23

27 35 51 **85** 66 23

27 35 51 85 **66** 23

27 35 51 66 85 **23**

23 27 35 51 66 85

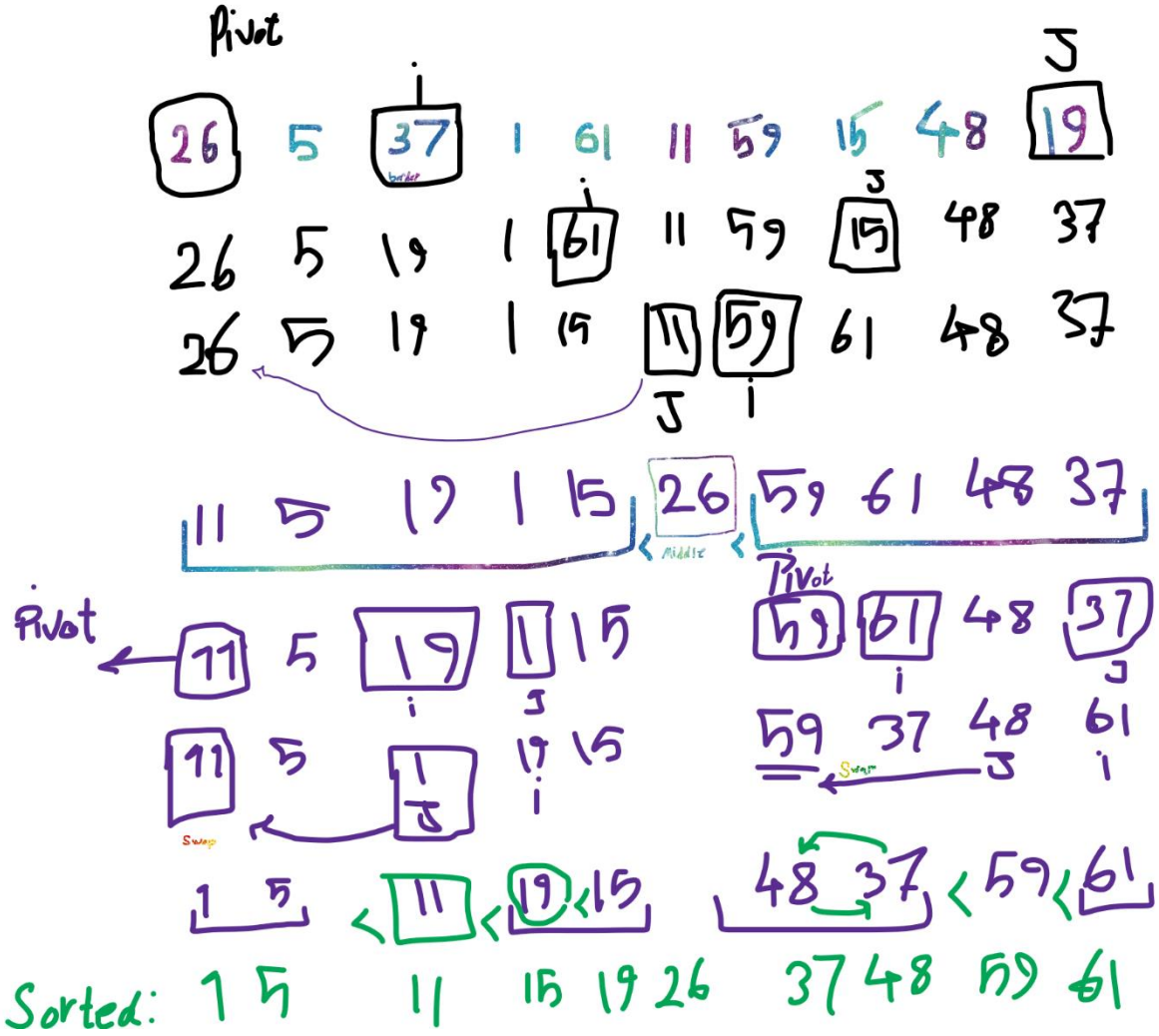
برای درک بهتر کد آن در زیر نوشته شده است:

```
١. def insertionSort(ls):
٢.     for i in range (len(ls)):
٣.         for j in range(i, 0, -1):
٤.             if ls[j] < ls[j-1]:
٥.                 temp = ls[j]
٦.                 ls[j], ls[j-1] = ls[j-1] , temp
٧.     return ls
٨. data_test_2 = [35, 51, 27, 85, 66, 23]
```

[3] Insertion sort

مرتب سازی سریع[4]

در این نوع مرتب سازی اولین عدد را **لولا**[5] می‌نامیم، از چپ به راست، اولین عدد بزرگ نسبت به لولا را انخاب می‌کنیم و به عنوان **i** علامت می‌گذاریم، از راست به چپ اولین عدد کوچک را نسبت به لولا انتخاب می‌کنیم و به عنوان **j** علامت گذاری میکنیم، و در نهایت جای آنها را با هم تغییر می‌دهیم. این کار را تا جایی انجام می‌دهیم که i از j بگذرد، درنهایت عدد **j** را با عدد لولا تغییر خواهیم داد، که متوجه می‌شویم، عددی که به عنوان لولا در قسمت **j** قرار گرفته است، اعداد قبل آن **کوچکتر** از آن و اعداد بعد از آن **بزرگتر** از آن می‌باشد.



[4] Quick sort
[5] Pivot

کد آن:

```
١. def quickSort(ls, low, high):
٢.     if len(ls) == 1:
٣.         return ls
٤.
٥.     if low < high:
٦.         pivot = partition(ls, low, high)
٧.         quickSort(ls, low, pivot - 1)
٨.         quickSort(ls, pivot + 1, high)
٩.  def partition(ls, low, high):
١٠.     i = low - 1
١١.     pivot = ls[high]
١٢.
١٣.     for j in range(low, high):
١٤.         if ls[j] <= pivot:
١٥.             i += 1
١٦.             ls[i], ls[j] = ls[j], ls[i]
١٧.     ls[i + 1], ls[high] = ls[high], ls[i + 1]
١٨.     return i + 1
١٩. data_for_quick_sort= [35, 51, 27, 85, 66, 23]
٢٠. quickSort(data_for_quick_sort, 0, len(data_for_quick_sort)-1)
٢١.
٢٢. data_for_quick_sort
```

پشته و صف:[6]

پشته:

پشته فضاهایی رو رم هستند که در آن ها یکسری از دستورعمل ها قرار میگیرند، قرارگیری این دستور عمل ها از پایین به بالا هست و سیستم خواندن آن بر اساس اخرین ورودی، اولین خروجی، یعنی **LIFO** یا **Last Input First Output**، مانند قرارگیری ظروف است، **آخرین** ظرف چیده شده روی هم به عنوان **اولین** ظرف استفاده میشود.

وقتی که بخواهیم در این فضا داده ای را اضافه یا بنویسیم، به این عمل **push** میگویند.

وقتی بخواهیم از این فضا داده ای را حذف یا بخوانیم، به این عمل **pop** میگویند.

شبه کد اضافه کردن داده به پشته:

```
١. function push_data(k: list){
٢.       int top = -1;
٣.      while (top < stack.length){
٤.            top += 1;
٥.            stack[top] = k[top];
٦.      }
٧. }
```

شبه کد حذف و خواندن از حافظه پشته:

```
١. function pop_data (){
٢.       int top = stack.length-1;
٣.      while (top <= 0){
٤.            k = stack[top];
٥.            top -= 1;
٦.      }
٧. }
```

[6] Stack and queue

صف:

وقتی در مورد صف ها صحبت می‌کنیم دقیقا منظور همان صف‌های واقعی هستند که می‌تواند متشکل از گروهی از انسان ها باشد که برای رسیدن به چیزی یا انجام‌کاری به صورت مرتب صف شده اند، برعکس **پشته** که براساس الگوریتم اخرین ورودی به عنوان اولین خروجی خواهد بود، مانند ظروف یک آشپزخانه که برای شسته شدن روی هم قرار گرفته اند و آخرین ظرف به عنوان اولین ظرف برای شست‌وشو استفاده می‌شود. در صف‌ها براساس الگوریتم اولین ورودی، اولین خروجی [7]عمل می‌کنیم، یعنی مانند صف نانوایی که هرکسی اول باشد به عنوان اولین نفر کارش انجام میشود و می‌رود.

دو مفهوم اصلی در صف وجود دارد:

Front: در حقیت به ابتدای صف اشاره می‌کند.

Rear: به پشت و انتهای صف اشاره می‌کند.

عمل نوشتن، (**خواندن** و **حذف**) کردن توسط این دو اصطلاح انجام میشود.

برای نوشتن در صف از rear استفاده می‌شود، یعنی rear پیشرونده به جلو خواهد و دیتا به همراه آن درج خواهد شد.

برای خواندن یا حذف‌کردن از first استفاده می‌شود، آن هم مانند rear به صورت پیشرونده است اما در هنگام حذف‌کردن از سمت چپ به جلو خواهد آمد.

[7] First in First output

شبه‌کد نوشتن در صف به صورت زیر است:

```
۱. def addq (k):
۲.         while rear < len(queue):
۳.             rear+=1
۴.             queue[rear] = k
```

مراحل اضافه کردن مقدار به صف

Index	-1	-1	0	1	2	3	4
Value/status	F	R	3				
			R				

Index	-1	-1	0	1	2	3	4
Value/status	F	R	3	5			
				R			

Index	-1	-1	0	1	2	3	4
Value/status	F	R	3	5	5		
					R		

Index	-1	-1	0	1	2	3	4
Value/status	F	R	3	5	5	10	
						R	

در نتیجه ممکن است از ما بخواهند که Rear در چه خانه در انتها قرار خواهد گرفت؟ با این وجود در خانه read[3] قرار دارد که مقدار آن را قبلا پر کرده است.

شبه کد خواندن و حذف کردن از صف:

```
۱. def delq (k):
۲.     while Front < rear :
۳.         Front +=1
۴.         k = queue[Front]
```

Index	-1	-1	0	1	2	3	4
Value/status	F	R	3	5	5	10	
			F			R	

Index	-1	-1	0	1	2	3	4
Value/status	F	R		5	5	10	
				F		R	

Index	-1	-1	0	1	2	3	4
Value/status	F	R			5	10	
				F		R	

که در نهایت میتوان نوشت که **Front** در خانه [1] **Front** مقدارش را خوانده و سپس حذف کرده است.

مثال:

addq(15), addq(7), addq(−12), delq(A), delq(B), addq(3)

−1	−1	0	1	2	3	4
F	R					

−1	−1	0	1	2	3	4
F		15				
		R				

−1	−1	0	1	2	3	4
F		15	7			
			R			

−1	−1	0	1	2	3	4
F		15	7	−12		
				R		

−1	−1	0	1	2	3	4
		15	7	−12		
		F		R		

−1	−1	0	1	2	3	4
			7	−12		
			F	R		

−1	−1	0	1	2	3	4
				−12		
			F	R		

-1	-1	0	1	2	3	4
				-12	3	
			F		R	

R [3] , F [1]

در همین لحظه نوع دیگری از صف وجود دارد به نام صف **حلقوی**، که مانند قبل عمل می‌کند متنها وقتی که فضای صف پر می‌شود اگر توسط F فضای قبل از R حذف شده باشد، در صورت نبود فضای مناسب در انتها، به ابتدای (سمت چپ به راست) آن صف اضافه می‌شود.

مثال:

addq(10), addq(20), addq(30), addq(40), delq(A), delq(B), addq(50), addq(60)

-1	-1	0	1	2	3	4
F	R					

-1	-1	0	1	2	3	4
F		10				
		R				

-1	-1	0	1	2	3	4
F		10	20			
			R			

-1	-1	0	1	2	3	4
F		10	20	30		
				R		

−1	−1	0	1	2	3	4
F		10	20	30	40	
					R	

−1	−1	0	1	2	3	4
		10	20	30	40	
		F			R	

−1	−1	0	1	2	3	4
			20	30	40	
			F		R	

−1	−1	0	1	2	3	4
				30	40	
			F		R	

−1	−1	0	1	2	3	4
				30	40	50
			F			R

−1	−1	0	1	2	3	4
		60		30	40	50
		R	F			

میان‌وند[8]، پیشوند[9]، پسوند[10]

معمولا در عبارت ریاضی مورد استفاده قرار میگیرند.

ما به صورت پیشفرض در مسائل ریاضی از میان‌وند استفاده میکنیم بطوری‌که مسائل خود را به این شکل می نویسیم:

A +B

در پیشوند همان طور که از نامش پیداست، به صورت زیر خواهیم نوشت:

+AB

در پسوند به صورت زیر:

AB+

به عبارت ریاضی زیر توجه، کنید، یک عبارت **میان‌وند** است:

اول باید توجه‌کنیم که حتما اولیت حساب را تعین‌کرده باشیم، و با استفاده از پرانتز آنرا مشخص‌کنیم:

(A*(B-D)/E)-(F*(G+(H/K)))

پسوند آن:

ABD-*E/FGHK/+*-

پیشوند آن:

-/*A-BDE*F+G/HK

```
INFIX: (((A/B)-C)+(((D*E)-A)*E))
```

[8] Infix
[9] Prefix
[10] Postfix

```
POSTFIX: AB/C- DE*A-E*+
PREFIX: +-/ABC *-*DEAE
```

تبدیل پسوند به میان‌وند توسط استک ها میتواند انجام گیرد:

در این تبدیل ما از سمت چپ به راست استک ها را کامل خواهیم کرد.

POSTFIX: ab/c−de*+

➔

1	2	3	4	5
		e		
b	c	d	(d*e)	
a	a/b	(a/b)−c	(a/b)−c	((a/b)−c)+(d*e)

INFIX: (((a/b)−c)+(d*e))

مسئله زیر را حل کنید (از پسوند به میان‌وند ببرید)

```
62/2-42*+
```

INFIX: (((6/2)-2)+(4*2)) = 9

عبارت زیر را به پسوند ببرید:

-3^4

34^-

نکته: این مسئله بالا بیانگر آن است که اول سه به توان چهار خواهد رسید و منفی در آن ضرب می‌شود اما اگر 3- در حالت کلی (3-) باشد، خب آن را به این صورت مینویسیم:

(-3)4^

تبدیل پیشوند به میان‌وند.

در این تبدیل، تبدیل از سمت راست به چپ در استک مورد بررسی قرار میگیرد:

```
PREFIX: +-/abc*de ←
INFIX: (((a/b)-c)+(d*e))
```

	/			
	a			
	b	a/b		
e	c	C	((a/b)-c)	
d	d*e	d*e	d*e	(((a/b)-c)+(d*e))

مسئله زیر را از پیشوند به پسوند ببرید:

```
PREFIX: ^-*+ABC-DE+FG
INFIX: (((A+B)*C)-(D-E))^(F+G))
```

		+			
		B	*		
	-	A	C	-	
+	E	C	(A+B)	((A+B)*C)	^
G	D	(D-E)	(D-E)	(D-E)	(((A+B)*C)-(D-E))
F	F+G	(F+G)	(F+G)	(F+G)	(F+G)

INFIX: (((A+B)*C)-(D-E))^(F+G))

```
POSTFIX: AB+C*DE--FG+^
```

مسئله زیر را از پسوند به پیشوند ببرید:

```
POSTFIX: AB^C*D-EF/GH+/+
```

حل:

```
INFIX: ((A^B)*C-D) + ((E/F) / (G+H)))
PREFIX: +-*^ABCD //EF +GH
```

مسئله:

```
PREFIX: ++A/B-CD/-AB  -+C*D5/A-BC
```

حل:

```
INFIX: ((A+(B/(C-D))+((A-B))/(C+(D*5))-(A/(B-C)))
POSTFIX: ABCD-/+AB-CD5*+ABC-/-/+
```

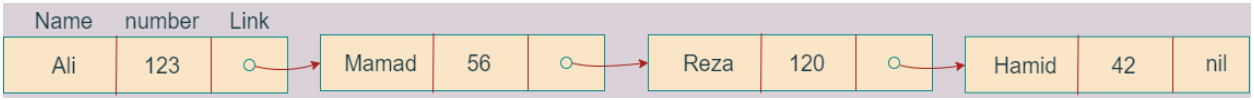
لیست پیوندی:

داده هایی که میخواهیم روی سیستم ذخیره کنیم به صورت آرایه ای این کار انجام می‌شود، آرایه به طور کلی خیلی پویا نیست چرا که داده ها به صورت پشت سر هم ذخیره میشود و حذف و اضافه از این آرایه کار سختی است. راهی که برای این کار وجود دارد آن است که داده ها را به صورت لیستی ذخیره کنیم.

برای مثال جدول زیر یک نوع ساختار آرایه ای در وارد کردن و خواندن اطلاعات است»

Name	number
Ali	123
Mamad	56
Reza	120
Hamid	42

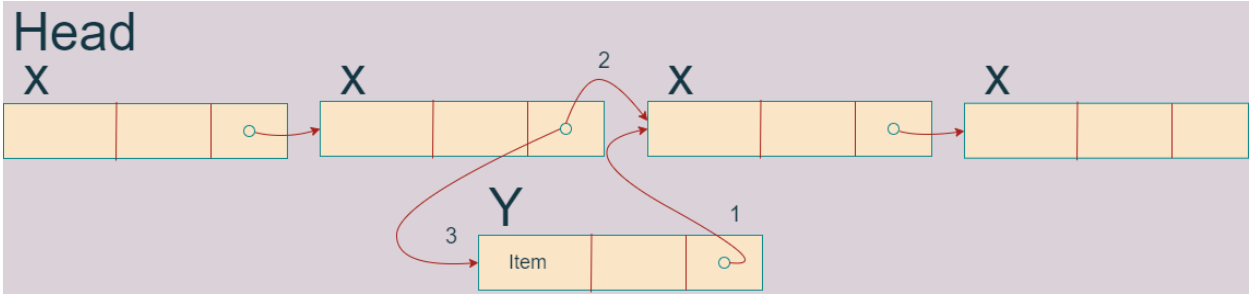
ساختار لیستی این آرایه به صورت زیر است:



الگوریتم این ساختار:

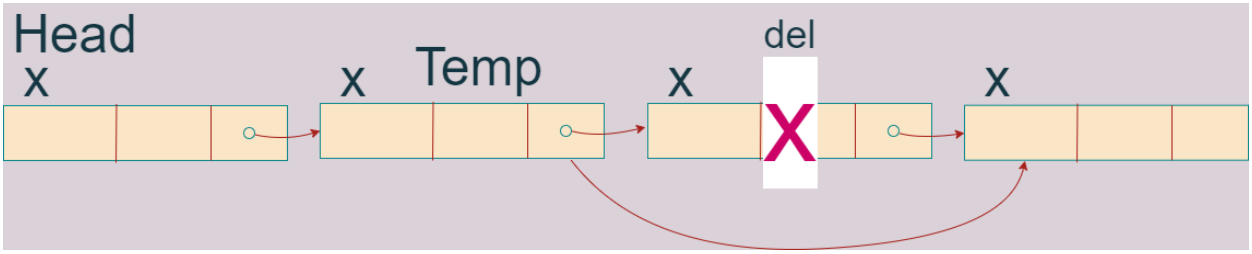
```
1. // define some var
2. type p_node = ^node;
3. node = record
4. String name [20];
5. int num ;
6. link = p_node;
7. end;
8. var x, y = p_node;
9. begin:
10. new(x):
11.    x^.name = "Ali"
12.    x^.num = 120;
13.    x^.link = nil;
14. new(y):
15.    y^.name = "mamad"
16.    y^.num = 56;
17. y^.link = nil;
18. x^.link = y;
19. end;
```

الگوریتم اضافه کردن به لیست یک طرفه:




```
1. new(Y):
2. Y^.data = item;
3. if head == nil {
4. print('list is empty');
5. }
6. begin
7. head = Y;
8. Y^.link = nil;
9. and else
10. begin
11. y^.link = X^link;
12. X^.link=Y;
13. end
```

الگوریتم حذف از لیست یکطرفه:




```
1. temp = head;
2. while(temp^.link != x){
3. temp = temp^.link;
4. temp^.link = x^.link;
5. dispose(x); // delete and destroy node!!
6. }
```

کامپیوتر عمل ریاضی زیر را چگونه انجام می‌دهید؟

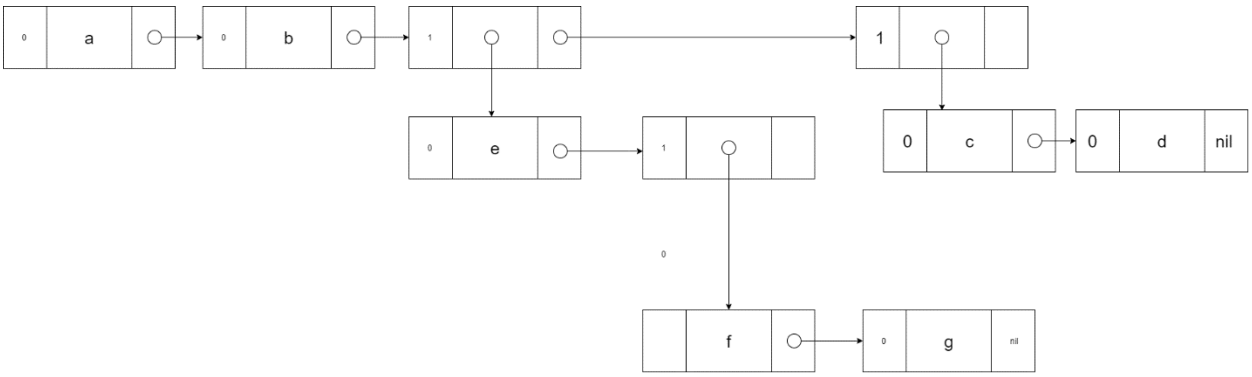
$$3x^{14} + 2x^{8} + 1$$

بصورت لیست پیوندی مانند زیر:

تبدیل فرم پرانتزی به لیست عمومی:

در این تبدیل در قسمت لیست عمومی یک قسمتی به نام تگ وجود دارد که اگر 0 باشد یعنی هیچ انشعابی در آنجا وجود ندارد و در نهایت در فرم پرانتزی نیازی به‌گذاشتن پرانتز برای اشنعاب جدید نیست اما اگر این تگ یک باشد باید انشعاب‌گرفته شود مانند زیر و در فرم پرانتزی از پرانتز استفاده شود.

تبدیل لیست عمومی به فرم پرانتزی:



(a, b, (e, (f, g)), (c, d))

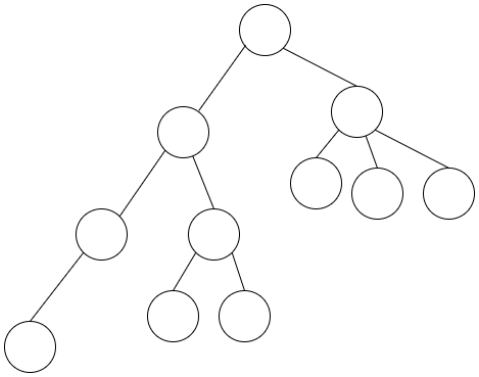
تبدیل فرم پرانتزی به لیست عمومی:

دقیقا مشابه روش بالاست، یعنی از چپ به راست شروع میکنیم به‌کشیدن لیست ها به محض این‌که به پرانتز رسیدیم یعنی در آنجا انشعاب رخ داده پس یک لیست با تگ یک میسازیم و سپس زیر مجموعه های آنرا مینویسم و اگر خاتمه داشت آنرا nil خواهیم کرد.

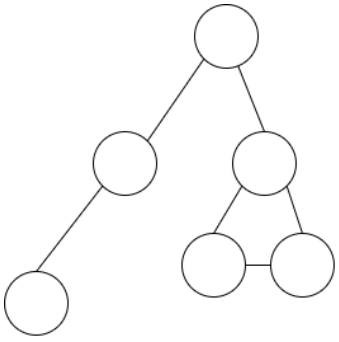
درخت:

درخت یک ساختار سلسله مراتبی دارد که بر اساس والد و فرزندی میتوان بین هر جز تفاوت قائل شد.

گراف زیر یک درخت را نمایش می دهد.



اما شکل زیر یک درخت نمیباشد.



تعریف گره:

به هر کدام از دایره ها گره یا Node گفته می‌شود.

درجه یک گره:

تعداد فرزندان یک گره را درجه گره میگویند. مانند شکل بالا که گره اول دارای دو درجه و گره دوم سمت راست دارای سه درجه است.

درجه یک درخت:

درجه یک درخت از طریق از بیشترین درجه یک گره که دارای آن است بدست می آید.

برگ:

به هر گره ای گفته می‌شود که هیچ شاخه شاخه ای ندارد یعنی به فرزندان و محصولات دیگری انشعاب ندارد، یا اینکه دیگر درجه آن صفر است. مانند شکل درست بالا در انتهای هر شاخه که روی هم 6 تا برگ هستند.

همزاد یا همنیا:

فرزندان یک گره را همزاد گویند. در شکل بالا 4 همزاد یا همنیا وجود دارد، و در انتها که در سمت چپ اخرین نود، تک فرزند است و همزادی ندارد.

اجداد یک گره:

گره هایی هستند در مسیر طی شده از ریشه تا آن گره وجود دارد.

ارتفاع یا عمق یک درخت:

به سطوح یک درخت گفته می‌شود.

یال:

یال به فاصله دو گره گفته میشود.

مسیر:

مسیر رسیدن از یک گره به گره‌ای دیگر گفته می‌شود.

شاخه:

مسیری که به برگ ختم شده باشد را شاخه می‌گویند.

درخت مرتب:

درختی که ترتیب زیر درخت در آن مهم است. یعنی تمام سطح هایش در یک وزن باشند، یا اینکه بهتر بگوییم یه طرف از طرف دیگر سنگین تر نباشد، درخت مرتب است.

درخت مشابه:

درختی که شبیه درخت جاری باشد (از نظر شکلی نه از نظر محتوایی).

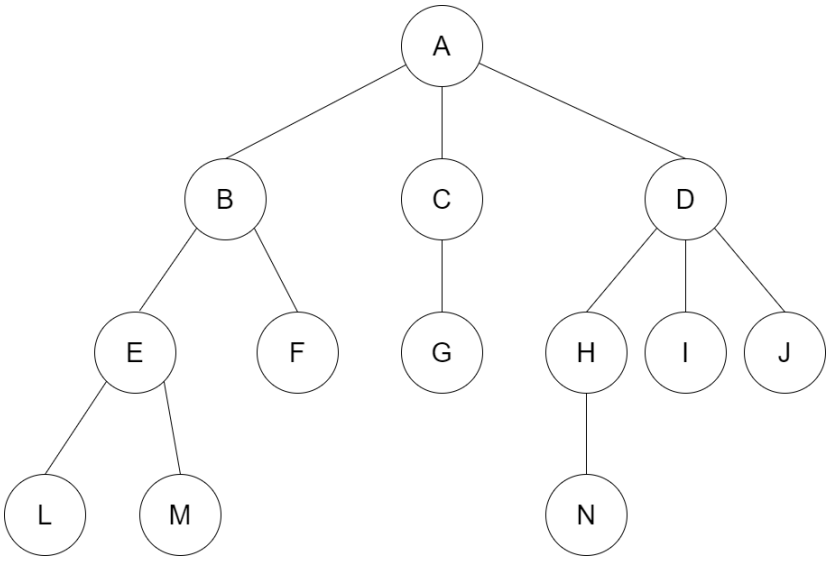
درخت کاپی:

نه تنها مشابه درخت بلکه محتوای آن هم مانند درخت ما باشد.

شکل پرانتزی درخت:

باتوجه والد هر فرزند به راحتی آن را به صورت فرم پرانتزی نمایش خواهیم داد.

با توجه به درخت زیر به سوالات زیر پاسخ دهید:



1. این درخت چند تا گره دارد؟
2. عمق این درخت چقدر است؟
3. همزاد های (A, B, C, D) را مورد بررسی قرار دهید:
4. این درخت چندتا برگ دارد؟
5. این درخت چند شاخه دارد؟
6. آیا این درخت مرتب است؟
7. درجه کل درخت چند است؟
8. درجه گره های ABCD و اگر کودکانی دارد را مورد بررسی قرار دهید.
9. مسیر بین A تا M و بعد از آن D تا N را بنویسید.
10. اجداد L را بررسی کنید.
11. اجداد N را بررسی کنید.
12. فرم پرانتزی این درخت را بنویسید.

1) این درخت 13 تا نود یا گره دارد.
2) با توجه به وزن آن، ارتفاع این درخت 4 است.
3) همزاد های A B C D:

A(n) = 3

B(n) = 2

C(n) = تک فرزند

D(n) = 3

4) این درخت 7 تا برگ دارد.
5) این درخت 7 تا شاخه نسبت به ریشه اصلی A دارد:

ABEL, ABEM, ABEF, ACG, ADHN, ADI, ADJ

این درخت میتواند در نقاطی نزدیک تر هم شاخه داشته باشد، مانند:

BEL, EM, EF, CG, DHNM, DI, DJ

6) بله درخت تا حدودی مرتب است چرا که وزن دو بازوی آن تعادل نسبی دارند.
7) با توجه به بیشترین درجه نود A و D این درخت 3 درجه دارد.
8) درجه های آن به صورت زیر است:

A(n) = 3

B(n) = 2

➔ E(n) = 2

C(n) = 1

D(n) = 3

➔ H(n) = 1

9) مسیر A تا M، ABEM و مسیر D تا H، DHN

10) اجداد L: EBA

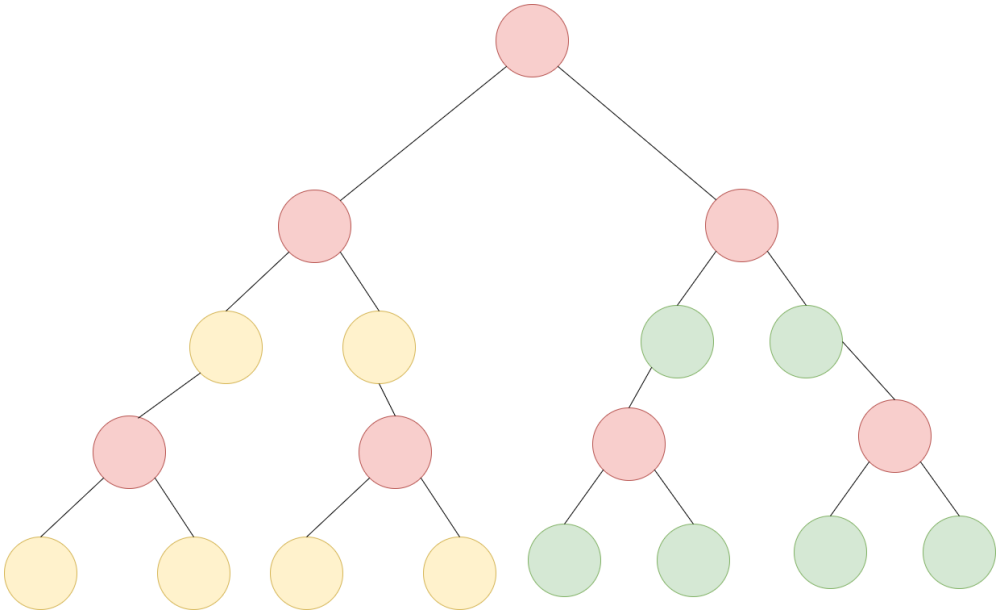
11) اجداد HDA: N

12) فرم پراتزی این درخت به صورت زیر است:

(A(B(E(L, M),F), C(G), D(H(N), I, J))

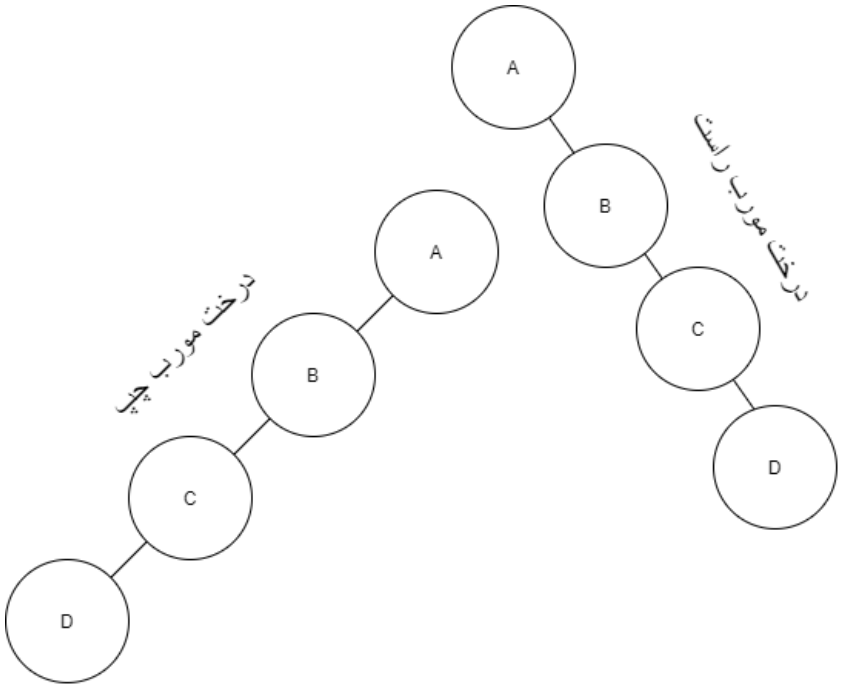
درخت دودویی

یا تهی است یا مجموعه ای از محدودی از گره ها که شامل یک ریشه و دو زیر درخت دودویی است. به این درخت، درخت پر، کامل هم گفته می‌شود.



درخت مورب:

درختی که برای به خاطر سپردنش باید به یاد دو دست خود بیوفتید، مانند شکل زیر:

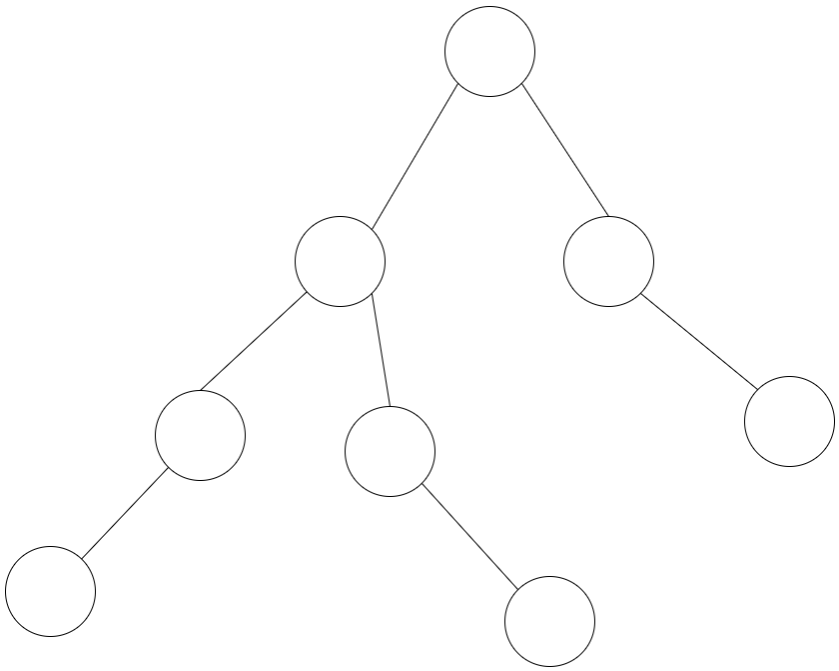


در درخت مورب چپ هر گره، فرزند چپ والد خودش است.

در درخت مورب راست هر گره، فرزند راست والد خودش است.

نمایش درخت در خانه های آرایه:

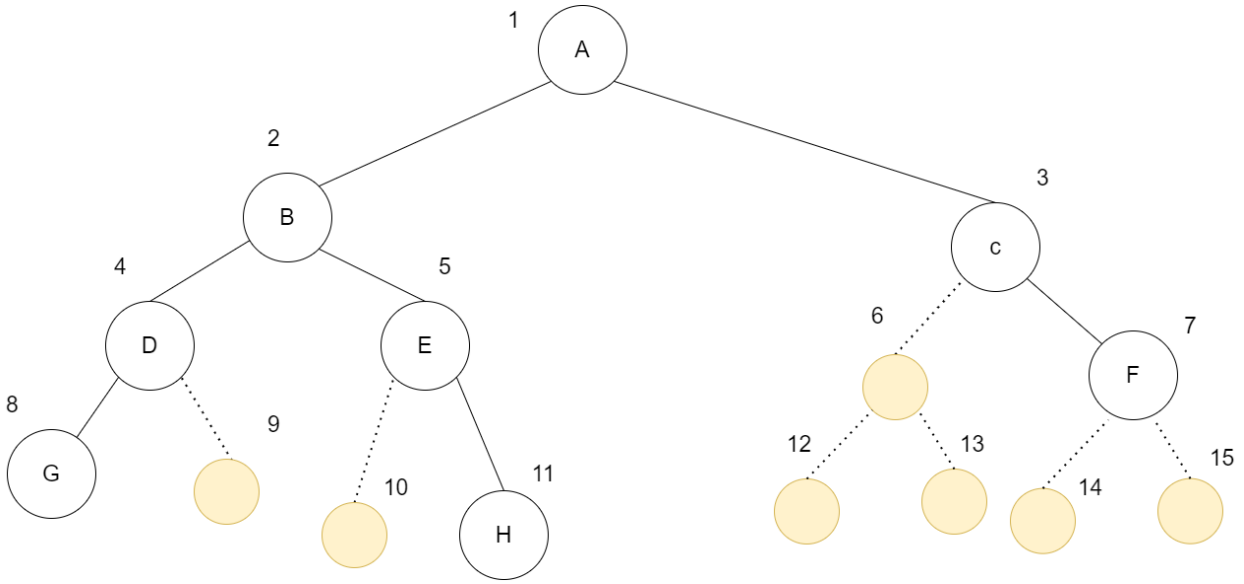
نکته، برای اینکار باید بررسی کنیم که هر ریشه به صورت دودویی فرزند داشته باشد، در غیر این صورت به صورت حاشور مانند باید فرزند آن را تداعی کنیم، اما فقط میکشیم در آرایه چیزی در مورد آن نمی-نویسیم.



در این درخت برای بدست آوردن تعداد خانه هایی که در آرایه مورد نیاز است، از عمق درخت استفاده میکنیم و آنرا در توانی از دو محاسبه خواهیم کرد.

در اینجا درخت ما چهار عمق دارد که میشود 2 به توان 4 برابر با 16 و منهای 1 = 15!

$$2^{tree\ deep} - 1$$

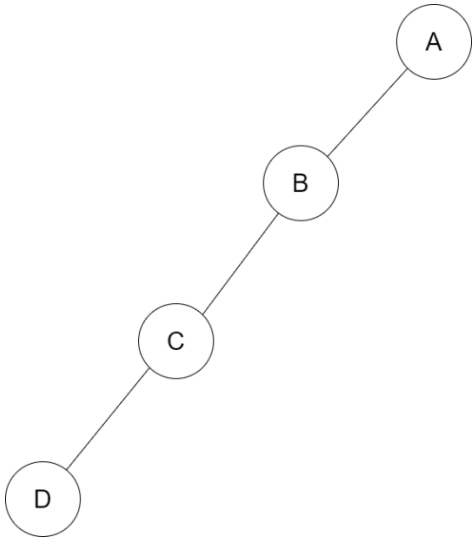


آرایه آن:

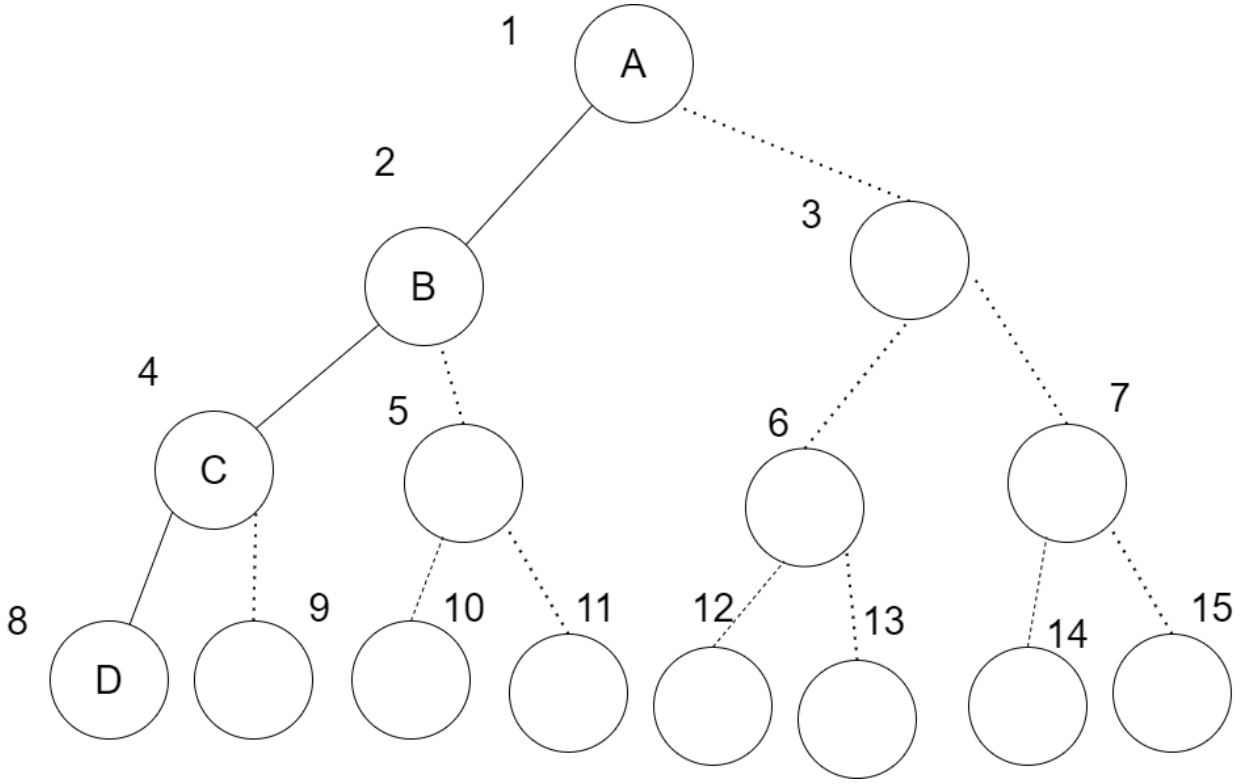
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B	C	D	E		F	G			H				

مثال دوم از این مسئله:

درخت زیر را در خانه های آرایه نمایش دهید.

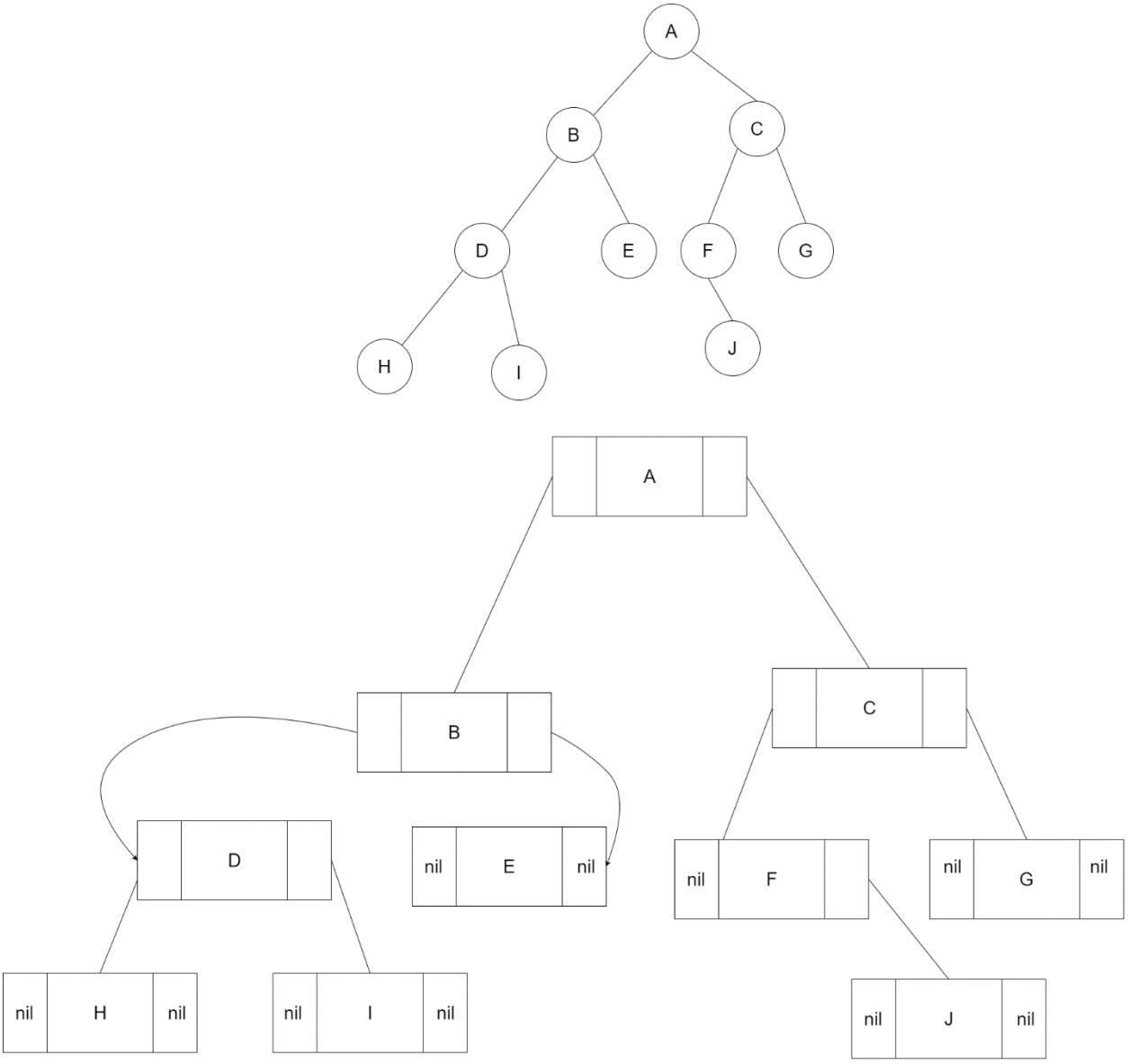


تکمیل درخت بالا:



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	B		C				D							

تبدیل درخت به لیست پیوندی:



پیمایش ها:

میانوندی **Inorder**

پیشوندی **Preorder**

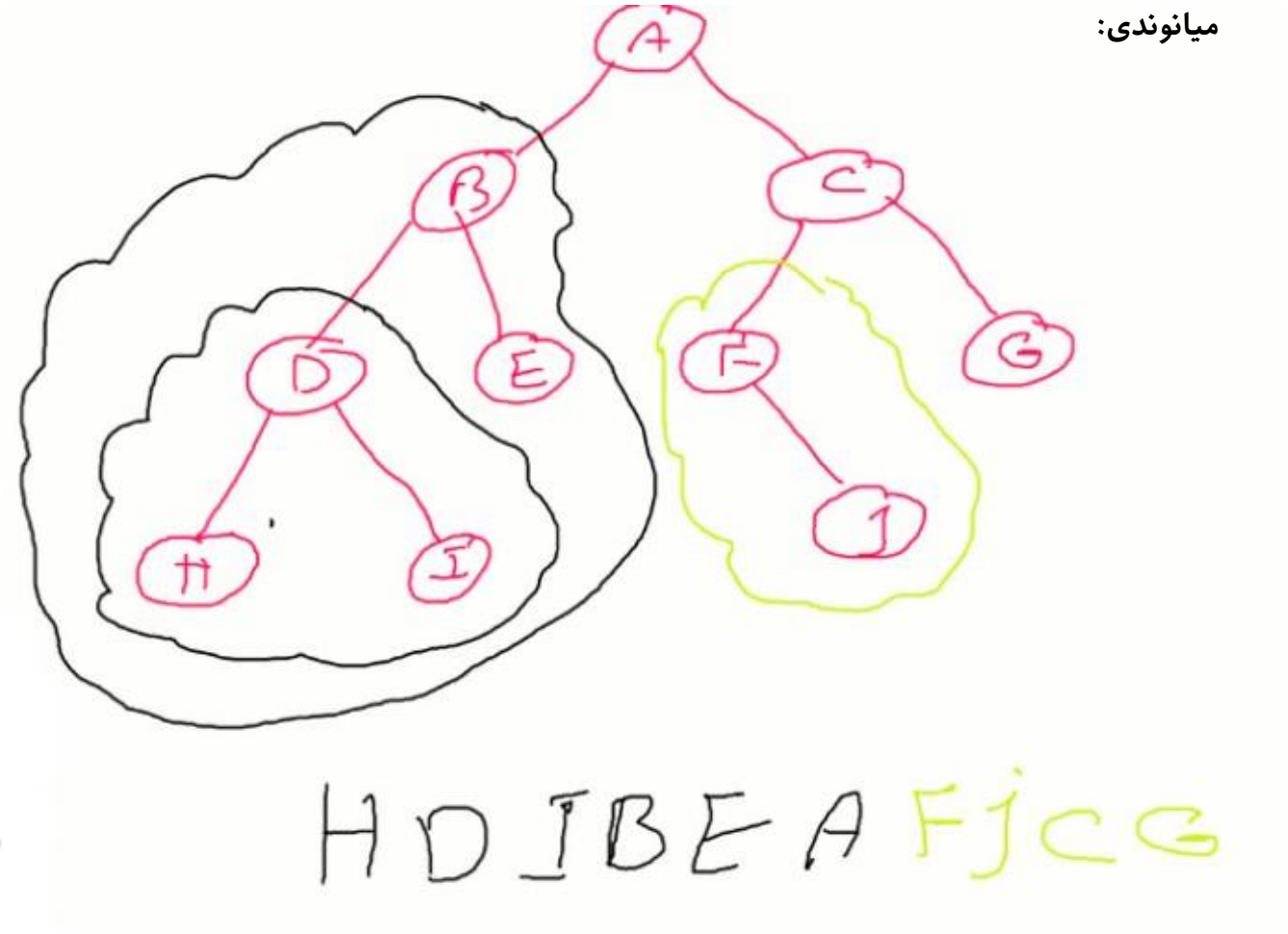
پسوندی **Postorder**

میانوندی: از پایین به بالا، اول چپ بعد گره بعد راست.

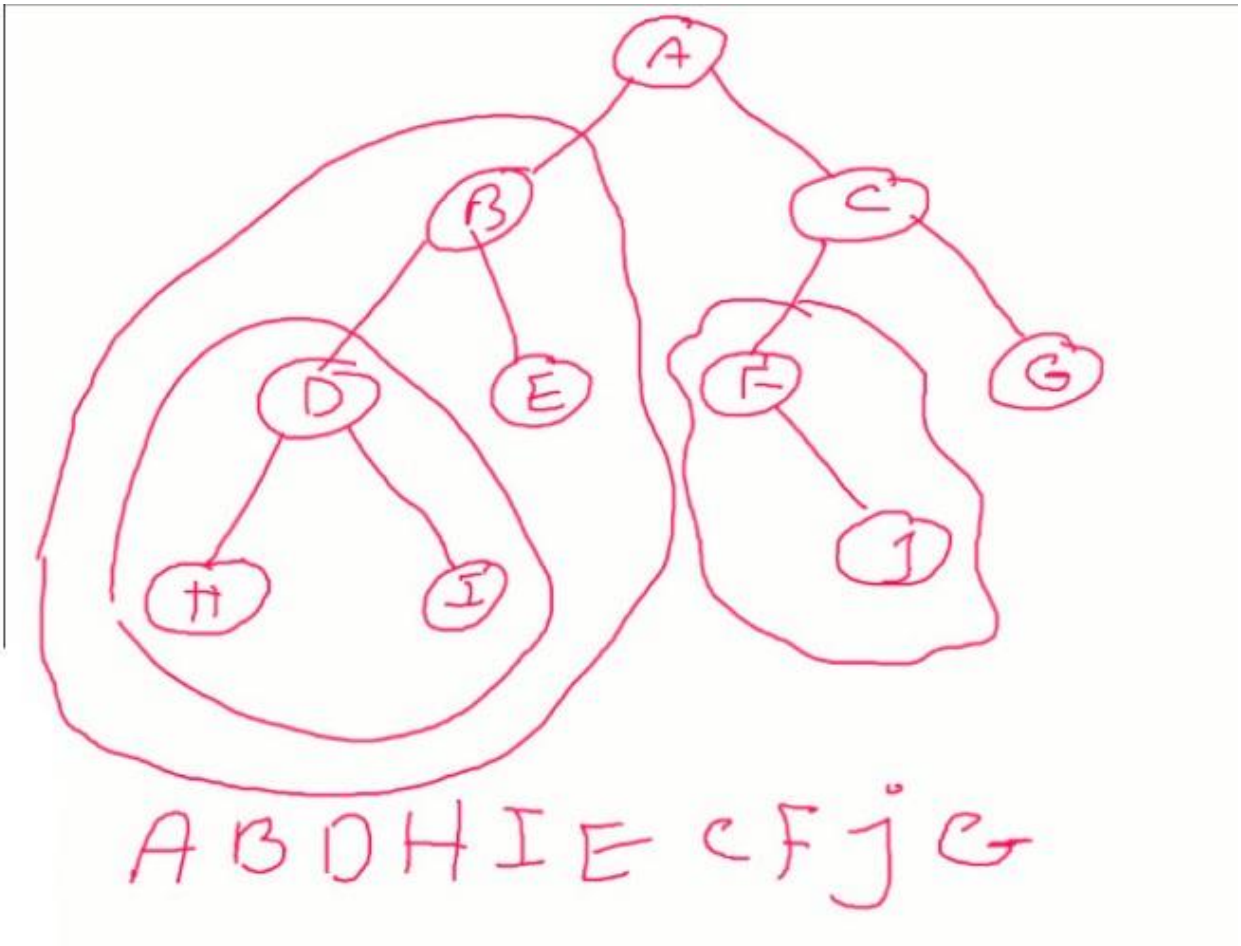
پیشوندی: از بالا به پایین، اول ریشه، بعد چپ بعد راست.

پسوندی: از پایین به بالا، اول چپ بعد راست بعد ریشه.

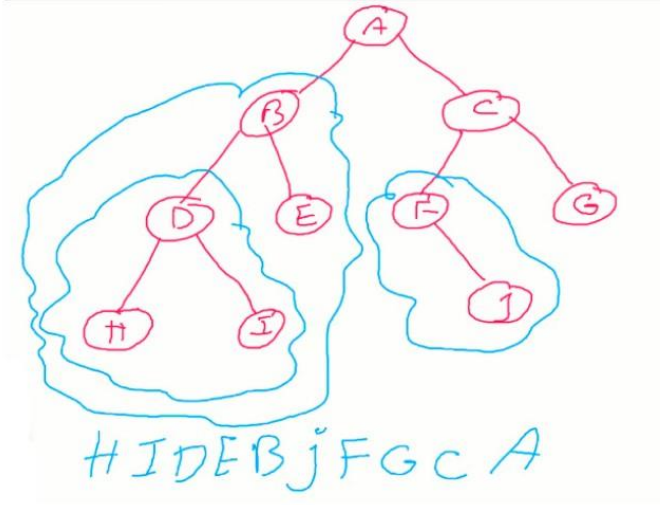
میانوندی:



پیشوندی:



پسوندی:


A
B
C
D
E
F
G
H
I
J
HIDEBjFGcA

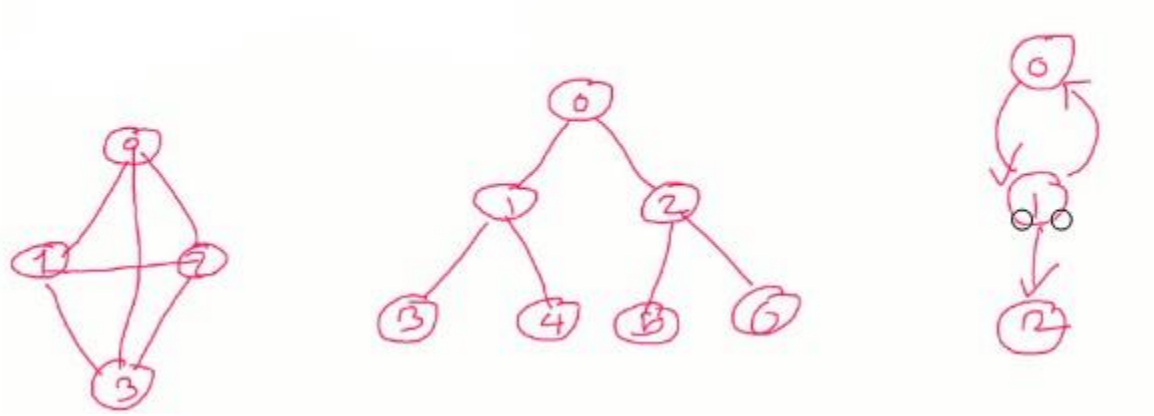
گراف ها

هر گراف شامل دو جموعه V و E است.

V: مجموعه محدود یا تهی از رئوس.

E: مجموعه محدود یا تهی از لبه ها یا یال ها.

G = (V, E)



تعداد رئوس:

v(G1) = 3

v(G2) =6

تعداد یال ها: v(G3) = 3

E(G1) = {(0, 1),(1, 3),(0, 2),(2, 3),(0, 3),(1, 2)}

E(G2) = {(0, 1), (1, 3), (1, 4), (0, 2), (2, 5), (2, 6)}

E(G3) = {<0, 1>, <1, 0>, <1, 2>}

گراف چندگانه:

گرافی است که چند مسیر، چند جهت و میتواند دارای طوقه یا حلقه هم باشد.

گراف ساده:

بر خلاف گراف چندگانه، نه حلقه دارد نه مسیر های چندگانه.

گراف کامل:

گرافي که حداکثر لبه ها را داراست. یعني از راسي یک مسیر دارد و به نوع خودش تعدي است یعني

A → B

B → C

A → C

مانند توپولوژي مش که در هر نودی ارتباطات برقرار است.

در گراف ساده و بدون جهت براحتی با استفاده از تعداد رئوس میتوان به تعداد لبه ها رسید.

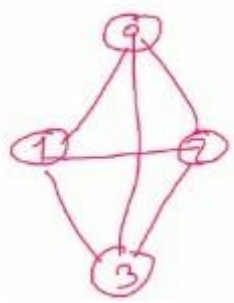
فرمول:

$$\frac{n(n-1)}{2}$$

برای مثال، تعداد یال های ممکن برای یک مربع بدون جهت را بدست آورید.

$$\frac{4(4-1)}{2}$$

که در نهایت جواب آن 6 است.

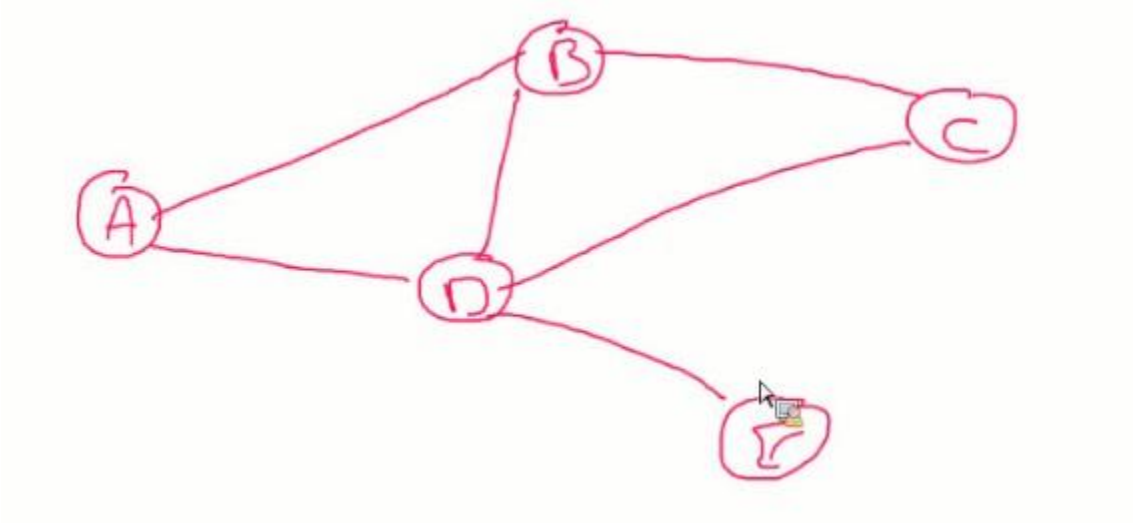


برای بدست آوردن حداکثر تعداد یال ها یا لبه های یک گراف جهت دار، از فرمول زیر استفاده میشود:

$$\underline{n(n-1)}$$

برای اینکه بتوانیم تعداد کل یال های یک گراف را بدست بیاوریم از فرمول زیر استفاده خواهیم کرد:

$$\frac{1}{2}\sum_{0}^{n-1} d$$



برای بدست آوردن مجموع تعداد یال های هر گره میتوان یکی یکی آنها را بررسی و باهم جمع کرد مانند فرمول بالا:

½ X (2, 3, 2, 4, 1) = 12/2 = 6