

# ساختمان داده ها

# Data Structure

## فصل پنجم

## درخت tree

تهیه و تنظیم : محمد نعیمی

عضو هیات علمی دانشگاه آزاد اسلامی

# تعریف درخت

درخت مجموعه محدودی از یک یا چند گره به صورت زیر می باشد :  
دارای گره خاصی به نام ریشه می باشد که میتواند تعدادی فرزند داشته باشد.  
بقیه گره ها به  $n$  مجموعه مجزا تقسیم میشوند که هر کدام نیز یک درخت می باشند و در درخت دور وجود ندارد.

**گره :** به عنصر حاوی اطلاعات و لینک به دیگر عناصر اطلاق می شود.

**درجه گره :** تعداد زیر درختهای یک گره، درجه آن نامیده میشود.

**درجه درخت :** ماکزیمم درجه گره های یک درخت

**برگ :** گره ای که درجه آن صفر باشد، برگ یا گره پایانی نامیده میشود و در مقابل بقیه گره ها گره های غیر پایانی نامیده میشوند.

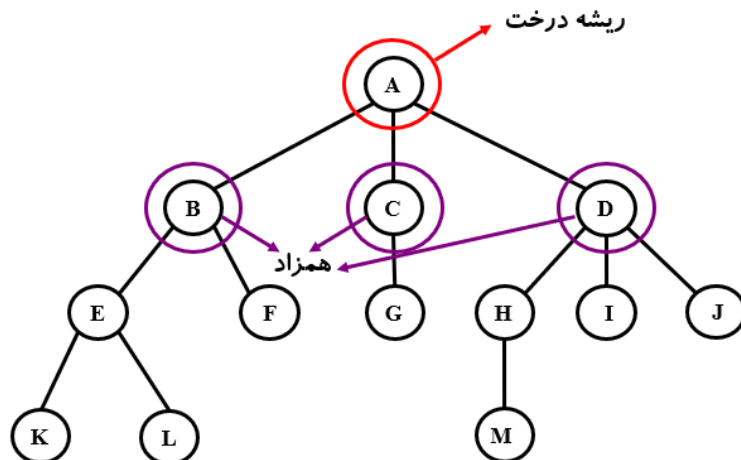
**همزاد (sibling) :** فرزندی که پدر یکسان دارند برادر (یا همزاد) نامیده میشوند.

**عمق یا ارتفاع یا سطح گره :** ریشه عمق 1 دارد و عمق هر گره میشود عمق پدر بعلاوه 1

**عمق یا ارتفاع درخت :** ماکزیمم عمق گره های درخت

**درخت M-tree :** درختی که در آن گره ای با درجه بیشتر از  $M$  وجود نداشته باشد.

**درخت دودویی (binary) :** درختی که در آن گره ای با درجه بیشتر از 2 وجود نداشته باشد. در این درخت دو فرزند را اصطلاحاً فرزند چپ و فرزند راست مینامیم.



درجه A ، 3 می باشد

عمق M ، 4 می باشد

درجه B ، 2 می باشد

عمق E ، 3 می باشد

ارتفاع درخت 4 می باشد

درجه درخت 3 می باشد

D پدر H ، 1 می باشد

K-L-F-G-M-I-J برگ می باشند

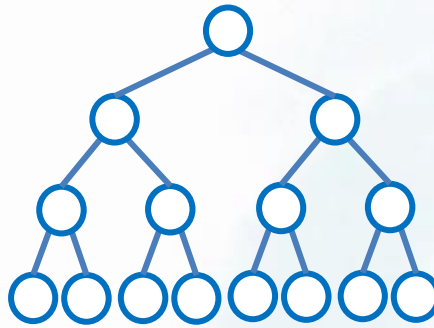
B-C-D با یکدیگر همزاد هستند

# برخی از مدل‌های درخت دودویی (binary)

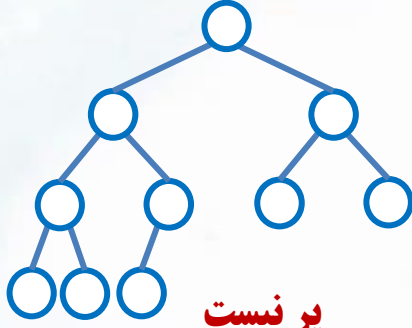
در درخت دودویی فرزند چپ با فرزند راست متفاوت است. این دو درخت متفاوت هستند



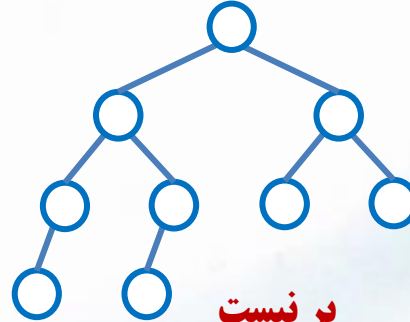
**درخت پر (perfect):** درختی با عمق  $d$  که  $2^d + 1$  گره داشته باشد را درخت پر می‌گوییم. (یعنی در تمام مکانهای ممکن تا عمق  $d$  گره وجود داشته باشد. به عبارت دیگر درجه تمام گره‌ها 2 یا 0 باشد و تمام برگ‌ها در یک سطح باشند)  
**درخت کامل (complete):** درختی با عمق  $d$  کامل است اگر تا عمق  $d-1$  پر باشد و در عمق  $d$  بین گره‌ها از چپ به راست جای خالی وجود نداشته باشد (تمام برگ‌ها در چپ‌ترین مکان ممکن باشند). هر درختی پری کامل نیز است.  
**درخت مورب:** درختی که تمام گره‌های فرزند دارش یا همگی فرزند چپ داشته باشند یا فرزند راست



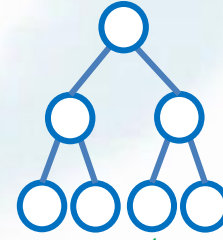
پر است  
کامل است



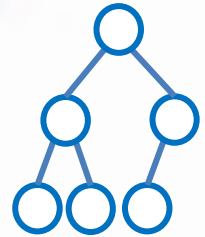
پر نیست  
کامل است



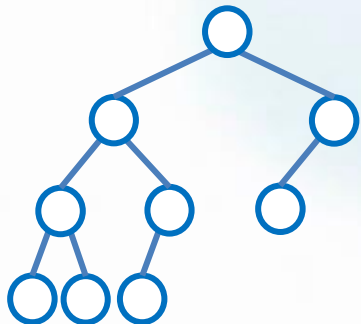
پر نیست  
کامل نیست



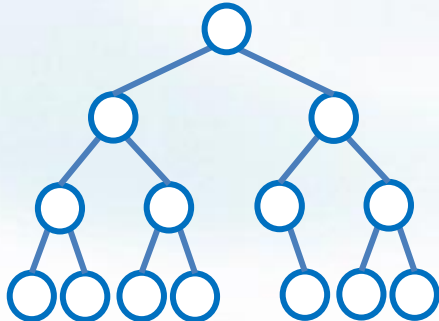
پر است  
کامل است



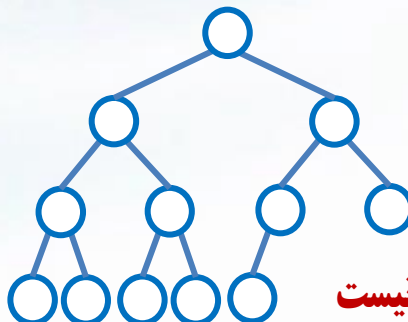
پر نیست  
کامل است



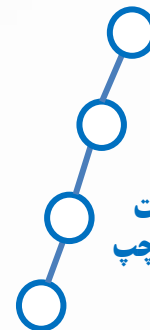
پر نیست  
کامل نیست



پر نیست  
کامل نیست



پر نیست  
کامل است



درخت  
مورب چپ



درخت  
مورب راست

# ویژگی های درخت دودویی (binary) (۲)

اگر  $n$  تعداد کل گره ها،  $n_0$  تعداد گره های بی فرزند (برگ)،  $n_1$  تعداد گره های تک فرزند و  $n_2$  تعداد گره های دو فرزندی باشد:

$$\begin{cases} n = n_0 + n_1 + n_2 \\ n = 1 + n_1 + 2n_2 \end{cases} \Rightarrow n_0 + \cancel{n_1} + \cancel{n_2} = 1 + \cancel{n_1} + 2\cancel{n_2} \Rightarrow \begin{cases} n_0 = n_2 + 1 \\ n_2 = n_0 - 1 \end{cases}$$

تعداد گره ها میشود تعداد کل فرزندان  
بعلاوه 1 (ریشه) که فرزند کسی نیست

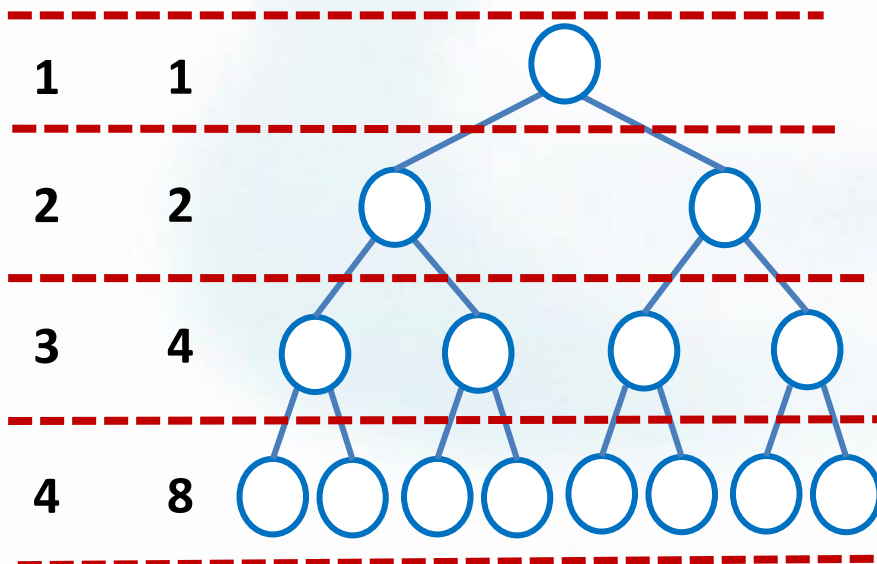
تعداد گره های موجود در عمق  $i$  می شود  $2^{i-1}$

حداکثر تعداد گره های یک درخت با عمق  $d$  می شود  $2^d - 1$

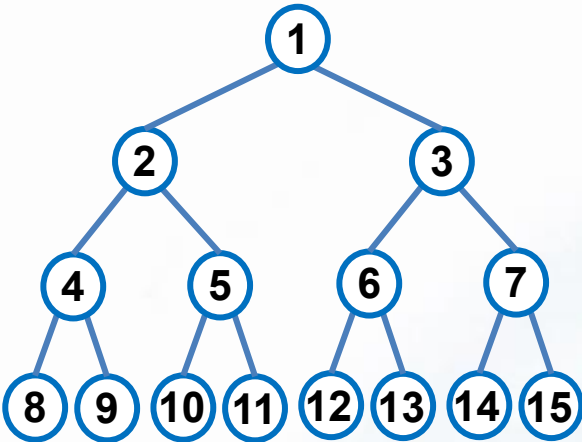
عمق درخت کامل با  $n$  گره می شود  $\lceil \log_2 n \rceil + 1$

تعداد کل گره ها  $2^4 - 1 = 15$

تعداد گره سطح



# نحوه ذخیره سازی درخت دودویی در برنامه (۱)

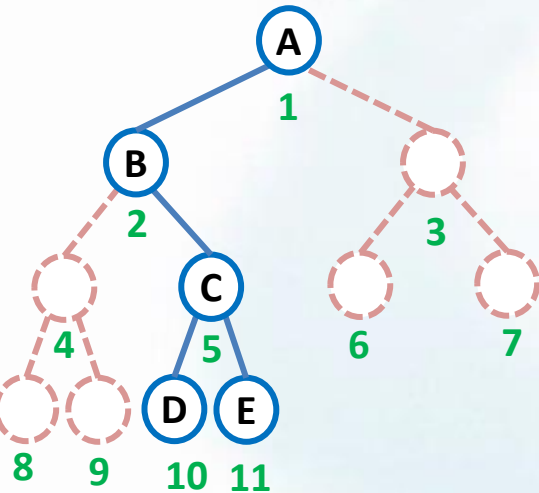


## درخت با استفاده از آرایه:

برای استفاده از آرایه لازم است خانه های درخت را به شکل رو برو شماره گذاری کنیم  
هر گره در خانه شماره مربوط به خودش در آرایه قرار میگیرد و در خانه 0 آرایه  
شماره آخرین عنصر (n) قرار میگیرد.

در خانه هایی که عنصر وجود ندارد یک مقدار نا معتبر میتوان گذاشت (مثلا -1)  
در نمایش درخت با آرایه نکات زیر برای گره با شماره i برقرار است:

- ☐ فرزند چپ آن در خانه  $2i$  قرار دارد
- ☐ فرزند راست آن در خانه  $2i+1$  قرار دارد
- ☐ پدر آن در صورتی که خودش ریشه نباشد در خانه  $[i/2]$  قرار دارد.



## مثال:

در این مثال، آخرین گره در خانه 11 قرار دارد لذا مقدار خانه 0 آرایه عدد 11 می باشد.  
گره های صورتی رنگ وجود ندارند و محتوای آن در آرایه عدد -1 می باشند.  
در این مثال ۶ خانه صورتی آرایه فارغ از طول آرایه همیشه هدر خواهد رفت.

11	A	B	-1	-1	C	-1	-1	-1	-1	D	E								
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17		

بهترین حالت برای درخت بوسیله آرایه، درختان کامل هستند که هدر رفت خانه آرایه در آنها 0 است  
بدترین حالت نیز درخت مورب از راست است. برای n عنصر تعداد  $2^n - 1$  خانه از آرایه نیاز است.

# نحوه ذخیره سازی درخت دودویی در برنامه (۲)

درخت با استفاده از لیست پیوندی:

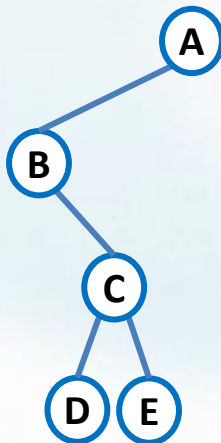
در این روش هر گره دارای سه قسمت است. یک بخش داده ای و یک اشاره گر به فرزند چپ و یک اشاره گر به فرزند راست

```
struct nodeT {  
    type item;  
    nodeT *left,*right;  
}
```

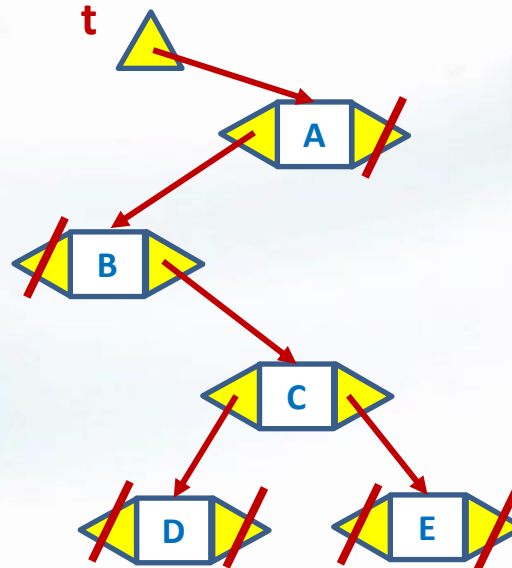


تعریف رکورد گره های درخت

```
nodeT * t;  
t=NULL;
```



درخت توسط یک متغیر (t) از نوع اشاره گر سازماندهی می شود زیرا باید به گره ریشه اشاره کند و آدرس آن را نشان دهد. تعریف اشاره گر درخت و مقدار دهی اولیه آن که ابتدا خالی (NULL) است.



# پیمایش درخت

پیمایش درخت در حقیقت نحوه و الگوی رسیدن به گره های درخت و چاپ آنها می باشد. روشهای مختلفی برای پیمایش درخت وجود دارد.

## ۱- پیمایش پیشوندی (preorder) :

این پیمایش به VLR نیز مشهور است. ابتدا خود گره رویت می شود و سپس کل درختچه چپ به صورت پیشوندی و پس از آن کل درختچه راست به صورت پیشوندی پیمایش می شود.

با توجه به این تعریف نوشتن این پیمایش و دو پیمایش بعد با توابع بازگشتی بسیار آسان است  
این روش به جستجوی عمقی (DFS) نیز مشهور است. با تعریف دیگر در این روش از چپ ترین گره شروع کرده تا آخرین عمق ممکن می رویم و هر جا به بن بست رسیدیم یک مرحله عقب می آییم و از فرزند دیگر در صورت امکان برای ادامه مسیر استفاده میکنیم

در سه پیمایش ابتدایی از پشته استفاده کرده یا با کمک توابع بازگشتی آن را مینویسیم (تابع بازگشتی توسط پشته مدیریت می شود)

## ۲- پیمایش میانوندی (inorder) :

این پیمایش به LVR نیز مشهور است. ابتدا کل درختچه چپ به صورت میانوندی پیمایش می شود سپس خود گره رویت می شود و پس از آن کل درختچه راست به صورت پیشوندی

## ۳- پیمایش پسوندی (postorder) :

این پیمایش به LRV نیز مشهور است. ابتدا کل درختچه چپ به صورت پسوندی و پس از آن کل درختچه راست به صورت پسوندی و پس از آن خود گره رویت می شود.

## ۴- پیمایش سطحی یا عرضی (BFS) :

در این روش گره ها به ترتیب از عمق یک تا d و در هر سطح از چپ به راست (شبه شماره گذاری گره ها در مدل آرایه ای) برای این پیمایش از صف استفاده می شود.

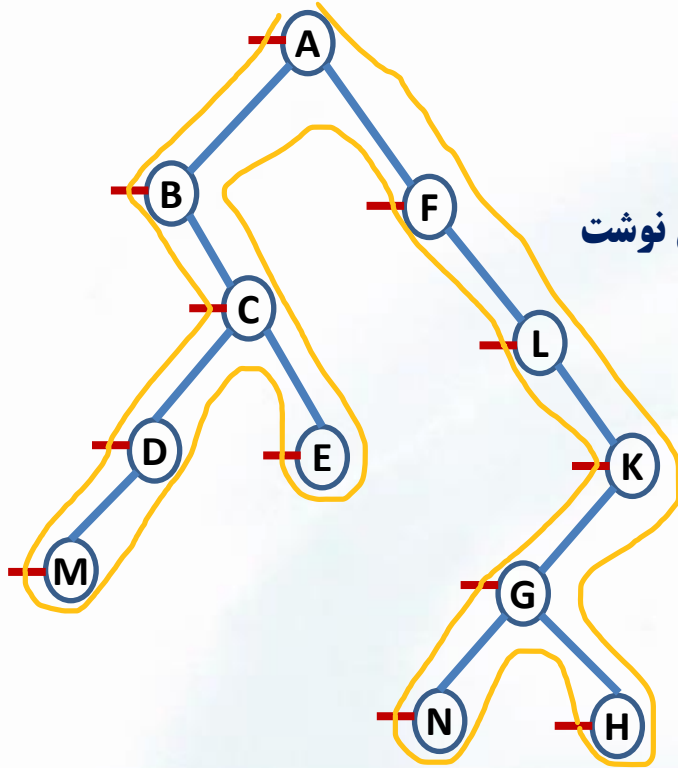


# پیمایش پیشوندی (preorder) VLR

- ☐ در این پیمایش ابتدا ریشه ملاقات می شود.
- ☐ سپس کل درخت سمت چپ preorder پیمایش می شود.
- ☐ سپس کل درخت سمت راست preorder پیمایش می شود.

تعریف فوق یک تعریف بازگشتی است لذا تابع بازگشتی آن را میتوان به سادگی نوشت

```
void preorder (nodeT *t)
{
    if (t!=NULL)
    {
        cout<<t->item;
        preorder(t->left);
        preorder(t->right);
    }
}
```



A B C D M E F L K G N H

روش سریع:  
کافی است در سمت چپ گره ها یک خط بگذاریم  
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت چپ  
گره رسیدیم آن را چاپ میکنیم.



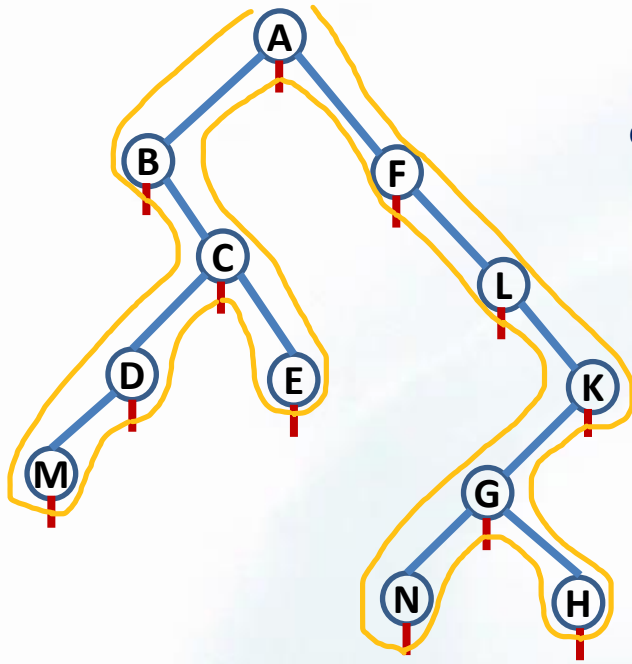
# پیمایش میانوندی (inorder) LVR

- ☐ در این پیمایش ابتدا کل درخت سمت چپ inorder پیمایش می شود.
- ☐ سپس ریشه ملاقات می شود.
- ☐ سپس کل درخت سمت راست inorder پیمایش می شود.

تعریف فوق یک تعریف بازگشتی است لذا تابع بازگشتی آن را میتوان به سادگی نوشت

```
void inorder (nodeT *t)
{
    if (t!=NULL)
    {
        inorder(t->left);
        cout<<t->item;
        inorder(t->right);
    }
}
```

```
void inorder (nodeT *t)
{
    مدل غیر بازگشتی
    nodeT *p=t;
    while(p!=NULL || !empty_stack(S))
    {
        while(p!=NULL)
        {
            push(S,p);
            p=p->left;
        }
        if (!empty_stack(S))
        {
            p=pop(S);
            cout<<p->item;
            p=p->right;
        }
    }
}
```



B M D C E A F L N G H K

روش سریع:  
کافی است در **پایین** گره ها یک خط بگذاریم  
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط پایین  
گره رسیدیم آن را چاپ میکنیم.

# پیمایش پسوندی (postorder) LRV

- ☐ در این پیمایش ابتدا کل درخت سمت چپ postorder پیمایش می شود.
- ☐ سپس کل درخت سمت راست postorder پیمایش می شود.
- ☐ سپس ریشه ملاقات می شود.

تعریف فوق یک تعریف بازگشتی است لذا تابع بازگشتی آن را میتوان به سادگی نوشت

```
void postorder (nodeT *t)
{
    if (t!=NULL)
    {
        postorder(t->left);
        postorder(t->right);
        cout<<t->item;
    }
}
```

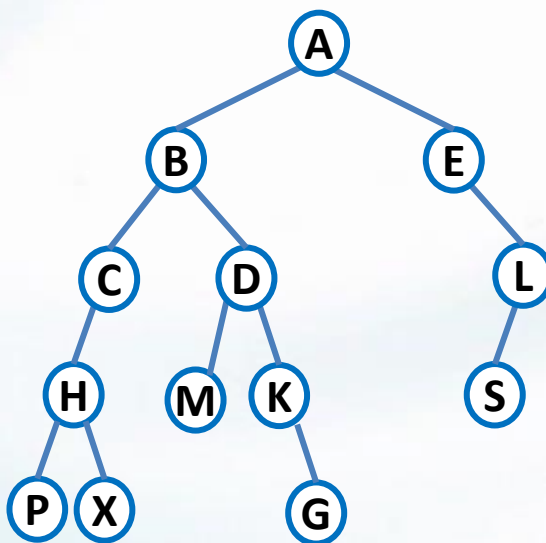


M D E C B N H G K L F A

روش سریع:  
کافی است در **سمت راست** گره ها یک خط بگذاریم  
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت راست  
گره رسیدیم آن را چاپ میکنیم.

# پیمایش سطحی BFS

☐ در این پیمایش ابتدا ریشه به صف منتقل می شود.  
☐ با خروج هر عنصر از صف عنصر رویت (چاپ) میشود  
☐ چون قرار است سطح بعد از آن به ترتیب از چپ به راست رویت شود ابتدا فرزند چپ گره خارج شده از صف به صف وارد می شود و پس از آن فرزند راست آن و چون صف ترتیب را تغییر نمیدهد به همان ترتیب عناصر خارج و رویت می شوند.  
 میتوان قبل از ورود عناصر به صف آنها را رویت (چاپ) نمود اما چون برای هر گره دو قسمت ورود فرزند چپ و راست داریم در کد نویسی راحت تریم که هنگام خروج از صف عناصر را چاپ نماییم



**BFS:** ABECDLHMKSPXG

```

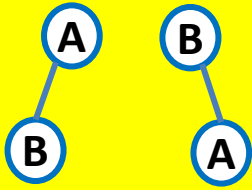
void bfs (nodeT *t)
{
    nodeT *p;
    if (t!=NULL)
        addq(Q,t);
    while(!empty_queue(Q))
    {
        p=delq(Q);
        cout<<p->item;
        if (p->left!=NULL)
            addq(Q,p->left);
        if(p->right!=NULL)
            addq(Q,p->right);
    }
}
    
```

خروجی	عملیات	صف
	addq(A)	A
A	delq()	A
BE	addq(B) addq(E)	
BE	delq()	B
ECD	addq(B) addq(E)	
ECD	delq()	E
CDL	addq(L)	
CDL	delq()	C
DLH	addq(H)	
DLH	delq()	D
LHMK	addq(M) addq(K)	
LHMK	delq()	L
HMKS	addq(S)	
HMKS	delq()	H
MKSPX	addq(P) addq(X)	
MKSPX	delq()	M
KSPX	delq()	K
SPXG	addq(G)	
SPXG	delq()	S
PXG	delq()	P
PX	delq()	
XG	delq()	X
G	delq()	G

# ساخت درخت بر اساس پیمایش های آن

این دو درخت متفاوت پیمایش  
pre و post یکسان دارند:

pre:AB      post:BA



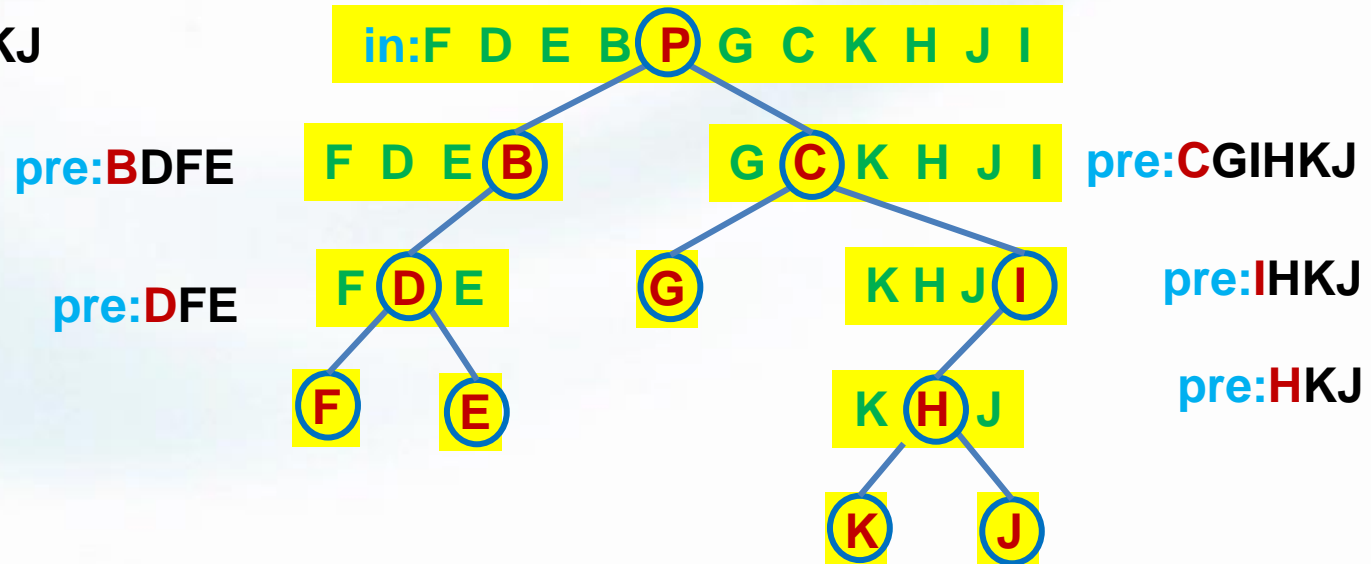
با داشتن پیمایش inorder و preorder می توان یک درخت منحصر به فرد ساخت.  
با داشتن پیمایش inorder و postorder می توان یک درخت منحصر به فرد ساخت.  
با داشتن پیمایش preorder و preorder نمی توان یک درخت منحصر به فرد ساخت.

**الگوریتم ساخت درخت از پیمایش in و pre:**

اولین عنصر در pre ریشه می باشد. پس آن را در پیمایش in پیدا کرده عناصر قبل از آن را در سمت چپ و بعد از آن را در سمت راست قرار می دهیم  
عملیات فوق را برای تمام قسمت های چپ و راست تا رسیدن به تک عنصر ادامه می دهیم.  
در حقیقت ما روی in کار میکنیم و از روی قسمتهای pre ریشه هر قسمت را پیدا میکنیم.  
□ در صورت داشتن post بجای pre، آخرین عنصر پیمایش post "ریشه" می باشد.

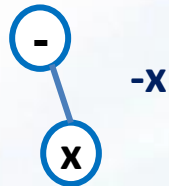
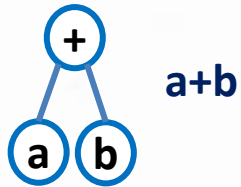
in:FDEBPGCKHJI

pre:PBDFECGIHKJ



# نمایش عبارت محاسباتی با درخت باینری (دودویی)

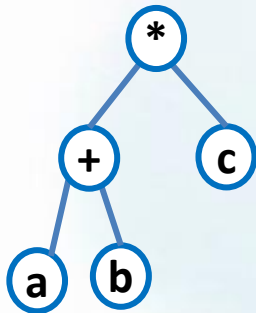
عبارت محاسباتی را میتوان با درخت دودویی نشان داد. عملگرهایی که دو عملوند دارد به صورت عملگر پدر، عملوند اول فرزند چپ و عملوند دوم فرزند راست نمایش داده می شود. عملگرهای تک عملوندي (– منفي علامت عدد) به صورت عملگر پدر و عملوند فرزند راست نمایش داده می شود.



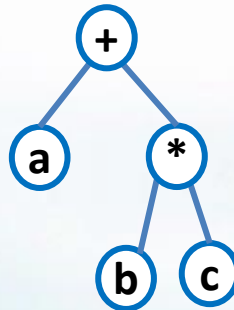
عملگرهایی که در سطح بالاتری قرار دارند دیرتر اعمال می شوند. برگ ها عملوند و گره های غیر برگ عملگرها هستند.

preorder، inorder و postorder درخت به ترتیب معادل prefixed، infix و postfix عبارت خواهد بود

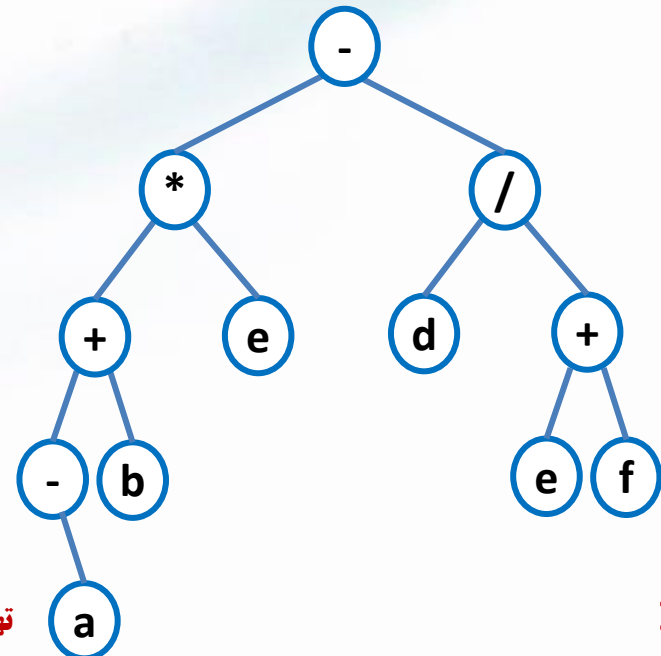
$(a+b)*c$



$a+(b*c)$



$((-a+b)*e)-(d/(e+f))$



# توابع بازگشتی برای درخت دودویی (۱)

تابع بازگشتی که دو درخت را گرفته مشخص کند  
عین هم هستند یا خیر؟

تابع بازگشتی که ریشه درخت را گرفته از آن  
درخت کپی گرفته، آدریس ریشه کپی را برگرداند

```
nodeT* copy (nodeT *t)
{
    nodeT *p;
    if (t==NULL)
        return(NULL);
    p=new();
    p->item=t->item;
    p->left=copy(t->left);
    p->right=copy(t->right);
    return(p);
}
```

```
int equal (nodeT *a, nodeT *b)
{
    if (a==NULL && b==NULL)
        return(1);
    if (a!=NULL && b!=NULL)
        if (a->item==b->item)
            if (equal(a->left,b->left))
                if(equal(a->right,b->right))
                    return (1);
    return(0);
}
```

# توابع بازگشتی برای درخت دودویی (۲)

تابع بازگشتی محاسبه عمق یک درخت

```
int deep (nodeT *t)
{
    if (t==NULL)
        return(0);
    return(1+MAX(deep(t->left), deep(t->right)));
}
```

عمق درخت خالی  
صفر است

عمق درخت میشود 1 (خود گره) بعلاوه ماکزیمم عمق چپ و عمق راست

تابع بازگشتی محاسبه تعداد گره های یک درخت

```
int elements (nodeT *t)
{
    if (t==NULL)
        return(0);
    return(1+ elements(t->left)+ elements(t->right));
}
```

عناصر درخت خالی  
صفر است

عناصر درخت میشود 1 (خود گره) بعلاوه عناصر چپ و عناصر راست

تابع بازگشتی محاسبه تعداد گره های دو فرزندی یک درخت

```
int parent2 (nodeT *t)
{
    if (t==NULL)
        return(0);
    if (t->left!=NULL && t->right!=NULL)
        return(1+ parent2(t->left) + parent2(t->right));
    return(parent2(t->left) + parent2(t->right));
}
```

تابع بازگشتی محاسبه تعداد برگهای یک درخت

```
int leaf (nodeT *t)
{
    if (t==NULL)
        return(0);
    if (t->left==NULL && t->right==NULL)
        return(1)
    return(leaf(t->left) + leaf(t->right));
}
```