# ساختمان داده ها

## **فصل ۴:** درخت

**عین الله پیرا**

**نیمسال اول سال تحصیلی ۹۹-۱۳۹۸**

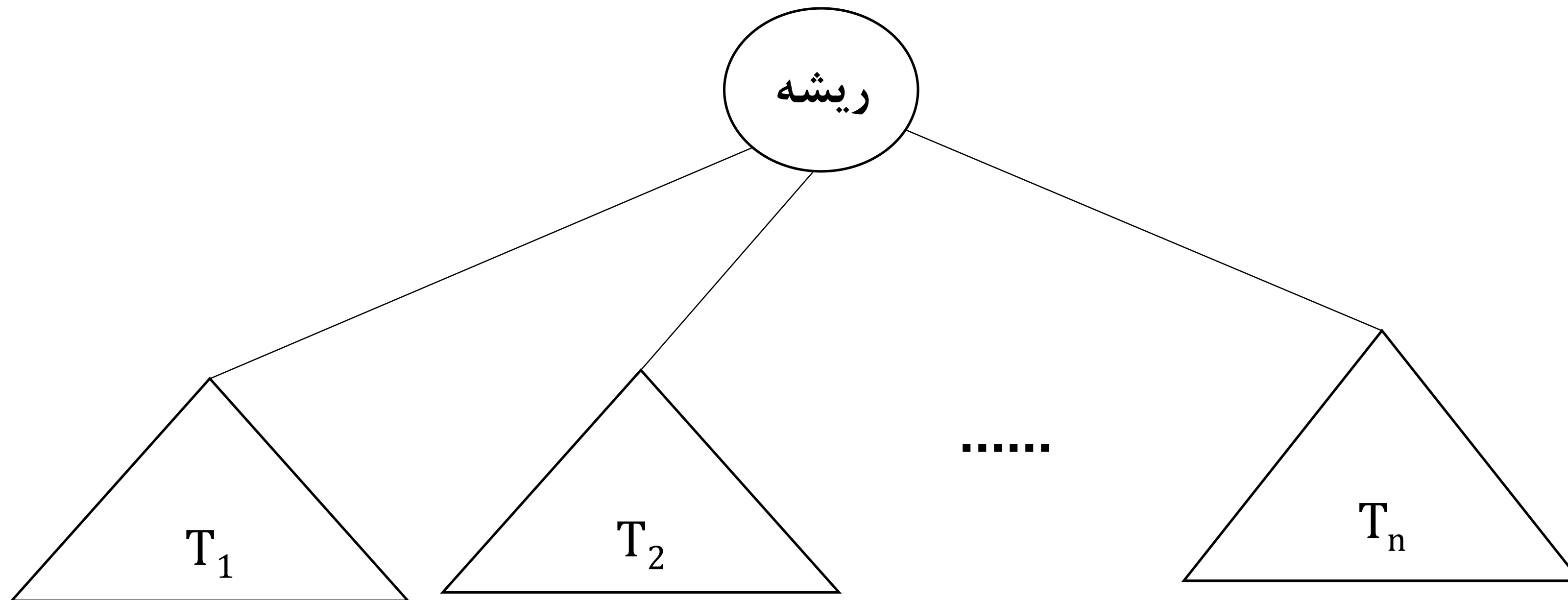
- اصطلاحات مربوط به مبحث درخت (Tree)
- درخت دودویی (Binary Tree)
- درخت جستجوی دودویی (Binary Search Tree: BST)
- درخت دودویی نخی (Threaded Binary Tree:TBT)
- کاربرد درخت های دودویی: الگوریتم رمزگذاری هافمن
- هرم (Heap)
- مرتب سازی هرمی (HeapSort)
- پیاده سازی صف اولویت با هرم
- تمرینات برنامه نویسی

# اصطلاحات مربوط به درخت (Tree)

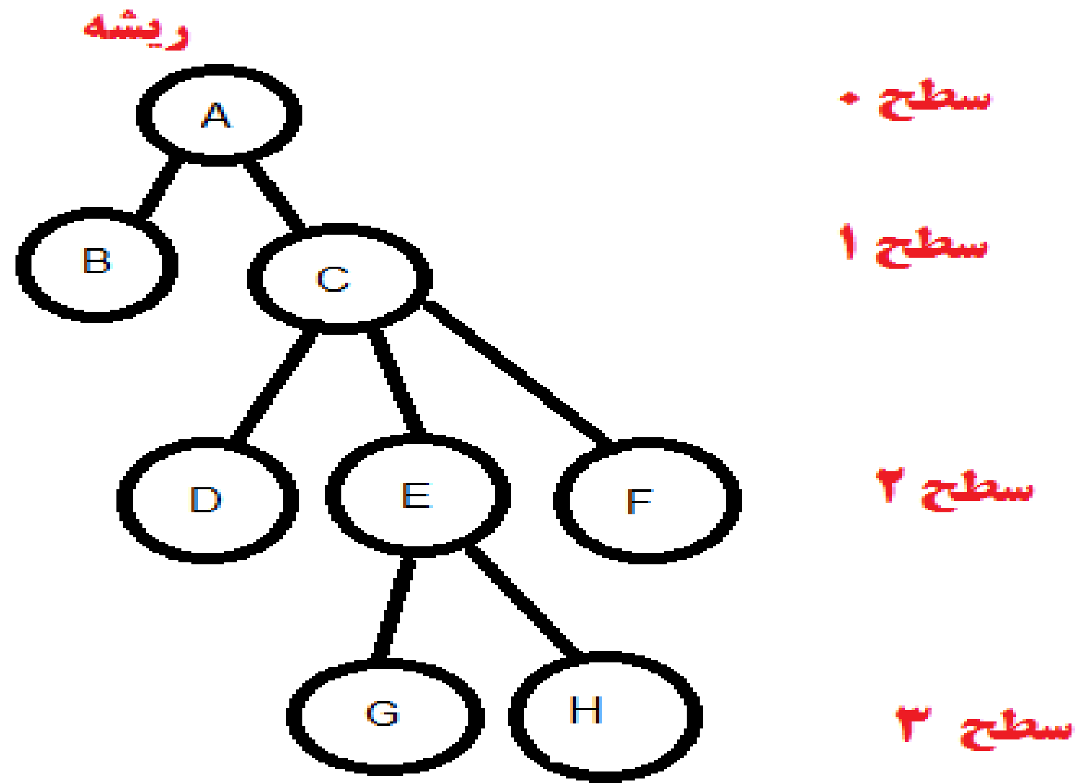
- تعریف درخت (Tree): درخت مجموعه محدودی از یک یا چند گره به صورت زیر می باشد:

1) دارای گره خاصی به نام ریشه (root) است.

2) بقیه گره ها به $n \geq 0$ مجموعه مجزا $T_1,T_2,\ldots,T_n$ تقسیم شده که هر یک از این مجموعه ها خود یک درخت هستند. $T_1,T_2,\ldots,T_n$ زیر درختان ریشه نامیده می شوند.



❑ مثال:



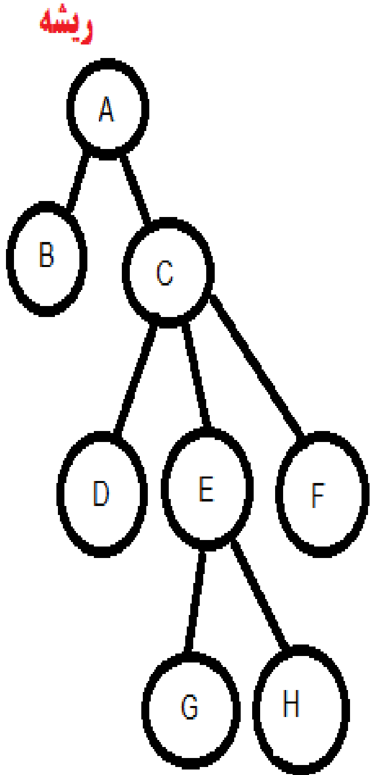

❑ گره های B و C فرزندان گره A هستند و A والد گره های B و C است.

❑ ریشه در سطح صفر قرار دارد. برای تمامی گره های بعدی، سطح گره برابر است با سطح والد به اضافه یک.

❑ گره های D، E و F همزاد هستند چون دارای والد مشترکی می باشند.

- به تعداد فرزندان یک گره، درجه آن گره گویند.

deg(A)=2 deg(C)=3 deg(H)=0




- گره هایی که درجه آنها برابر صفر هست را گره خارجی یا برگ (leaf) گویند.

بعنوان مثال: برگ های درخت عبارتند از: B,D,G,H,F

- گره هایی که درجه آنها صفر نباشد را گره داخلی گویند.

بعنوان مثال: گره های داخلی درخت عبارتند از: A, C, E

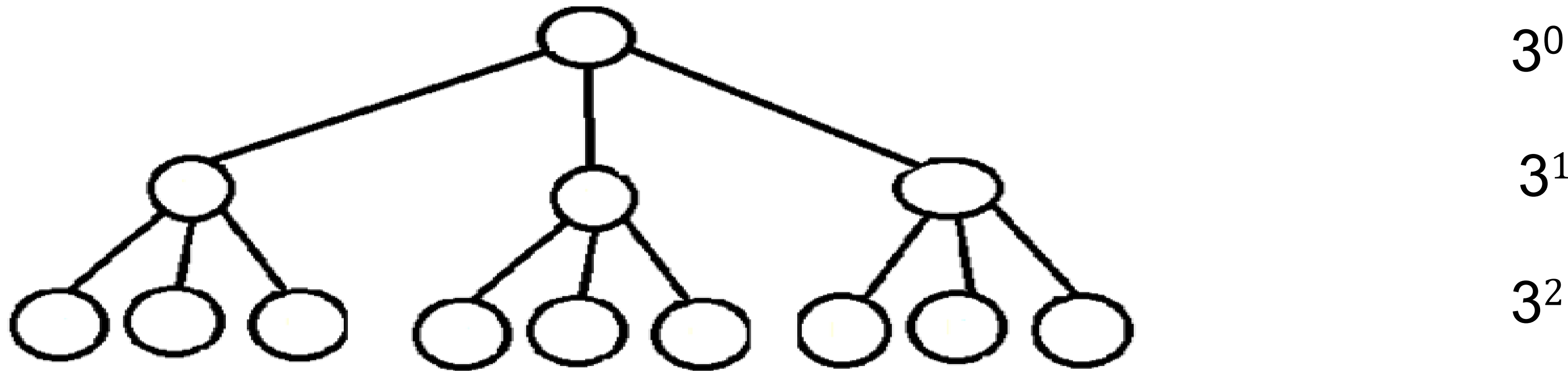
- عمق (ارتفاع) درخت به بیشترین سطح گره های آن درخت گفته می شود.

بعنوان مثال: عمق درخت برابر ۳ است.

- **درخت پر:** درختی است که درجه تمام گره های داخلی آن یکسان باشد و تمام برگ های آن در سطح آخر قرار داشته باشند.

- **بعنوان مثال:** درخت پر با ارتفاع ۲ و درجه ۳ (تمام گره های داخلی فقط ۳ فرزند داشته باشند)

تعداد گره ها در هر سطح



$3^0$

$3^1$

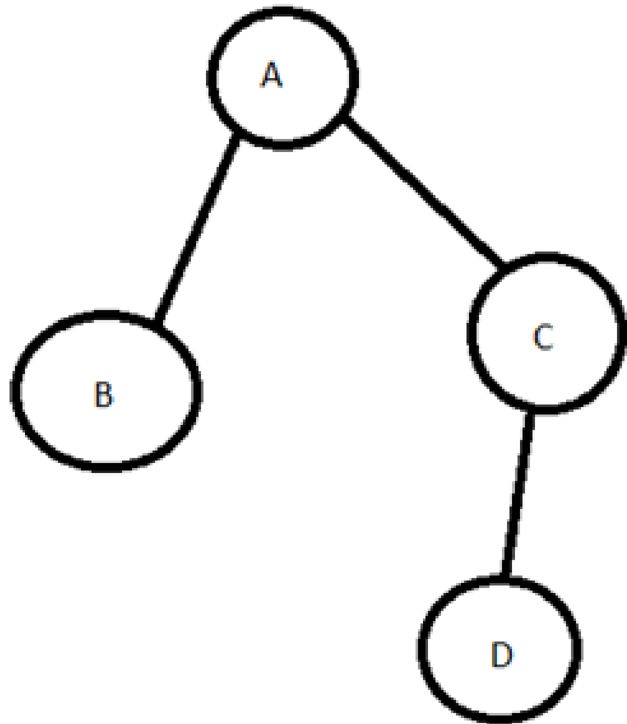
$3^2$

تعداد کل گره ها $n = 3^0 + 3^1 + 3^2 = \frac{3^{2+1} - 1}{3 - 1}$

- تعداد کل گره های یک درخت پر با درجه d و ارتفاع h برابر است با: $n = d^0 + \cdots + d^h = \frac{d^{h+1} - 1}{d - 1}$

# درخت دودویی (Binary Tree)

- یک درخت دودویی یا تهی است یا حاوی مجموعه ای محدود از گره ها شامل یک ریشه و دو زیر درخت دودویی است. این درخت ها زیر درخت های چپ و راست نامیده می شوند.
- در واقع، هر گره حداکثر دو فرزند دارد، فرزند چپ و فرزند راست.




- گره B فرزند چپ گره A است و گره C فرزند راست گره A است.
- گره D فرزند چپ گره C است.

- تفاوت درخت عادی با درخت دودویی:

۱- در هیچ درخت عادی صفر گره وجود ندارد ، اما درخت دودویی تهی وجود دارد.

۲- در یک درخت دودویی ترتیب فرزندان دارای اهمیت بوده در حالی که در درخت عادی به این صورت نیست.

- رابطه بین تعداد گره های برگ و تعداد گره های درجه ۲:

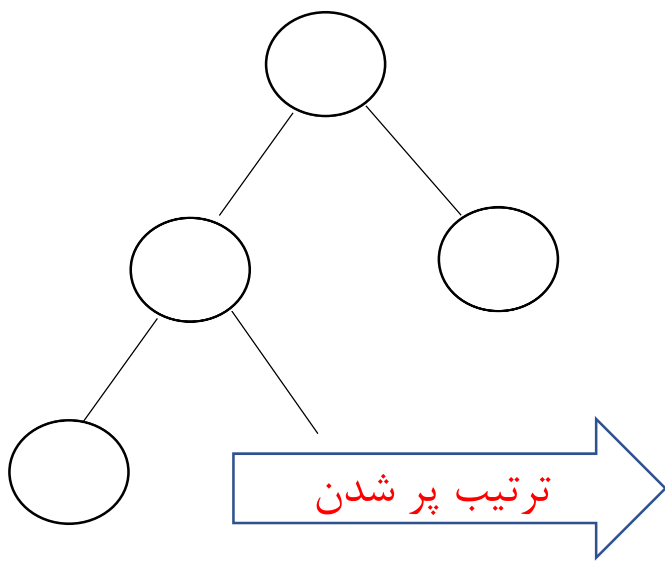
اگر $n_0$ تعداد گره های برگ و $n_2$ گره های درجه ۲ یک درخت دودویی باشد رابطه زیر را خواهیم داشت:

$$n_0=n_2+1$$

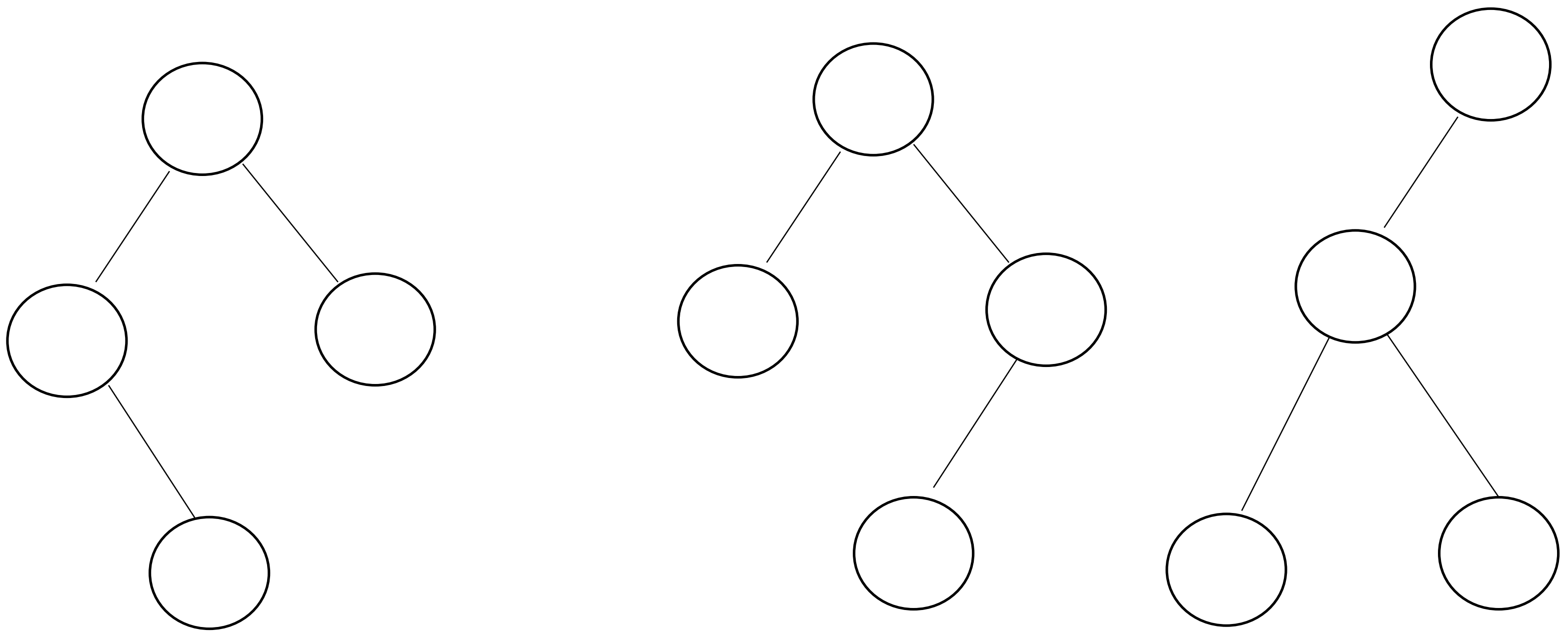
- درخت دودویی پر: درختی است که درجه تمام گره های داخلی آن یکسان و برابر ۲ باشد و تمام برگ های آن در سطح آخر قرار داشته باشند.

$$n = 2^0 + \cdots + 2^h = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$ تعداد کل گره ها

- درخت دودویی کامل: درختی است که یا پر است و یا می توان با افزودن گره هایی به سطح آخر و از سمت چپ به راست، آن را به درخت پر تبدیل کرد.




- بعنوان مثال: درختان زیر کامل نیستند.



❑ کار در کلاس: اگر h عمق درخت دودویی کامل باشد حداقل و حداکثر تعداد گره های آن را مشخص کنید.

جواب: حداقل، موقعی است که تمام سطوح h-1 کاملا پر باشد و فقط یک گره در سطح h باشد:

$$n=\left(2^{(h-1)+1}-1\right)+1=2^{h}$$

حداکثر، موقعی است که درخت پر باشد:

$$n=2^{h+1}-1$$

بنابراین داریم:

$$2^{h}\leq n\leq 2^{h+1}-1$$

# پیاده سازی درخت دودویی با آرایه

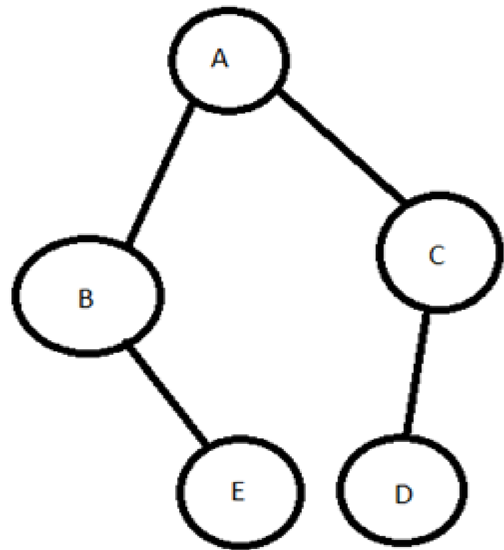
- برای پیاده سازی درخت دودویی با ارتفاع h به کمک آرایه، یک آرایه با نام a و بطول $2^{h+1}$ را در نظر می گیریم:

**خانه a[0] خالی می ماند**

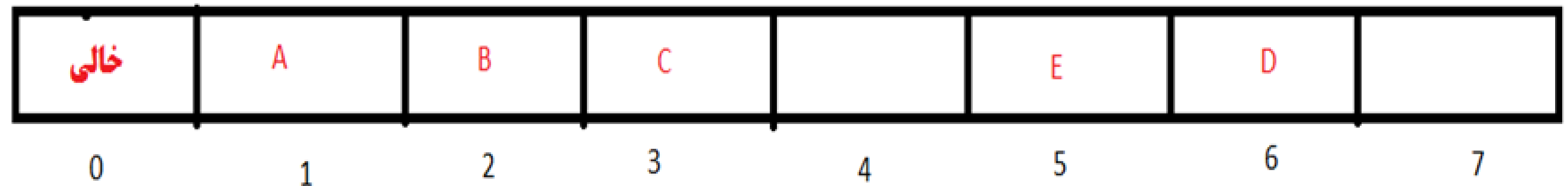
**ریشه ⟵ a[1]**

**فرزند چپ گره a[i] ⟵ a[2*i]**

**فرزند راست گره a[i] ⟵ a[2*i+1]**

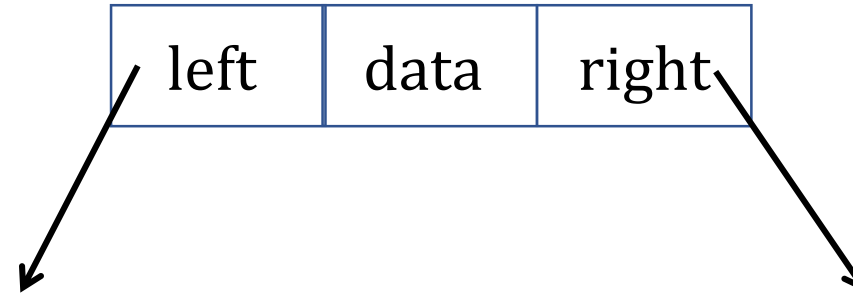



- مثال: در درخت زیر داریم h=2 پس طول آرایه باید برابر $2^{2+1}=8$ باشد.



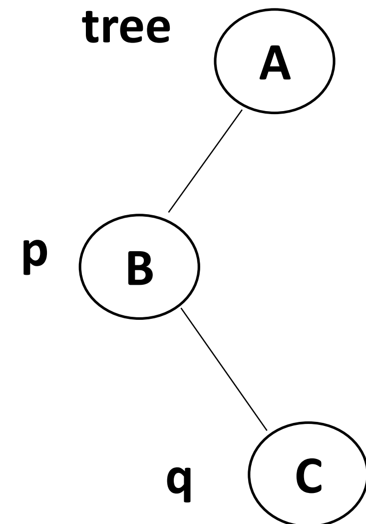

- نکته: والد گره a[j] ام در خانه a[[j/2]] قرار دارد (جزء صحیح).

```
struct  node {
        char     data;
        node   *left,*right;
};
```



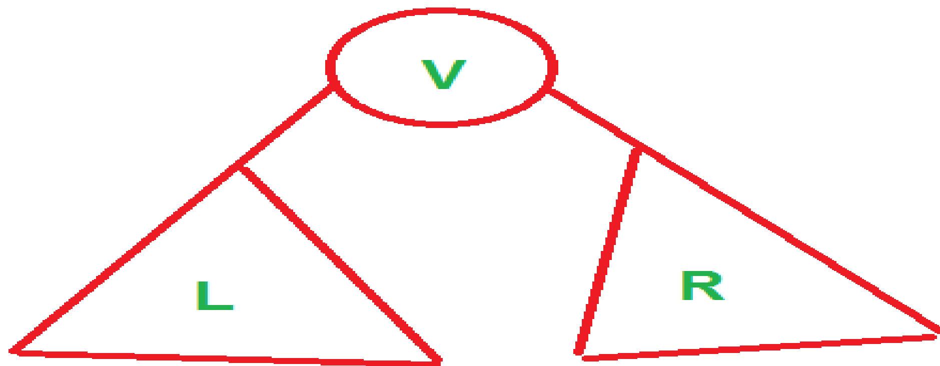

❑ مثال: درخت دودویی زیر را ایجاد کنید.

```
node  *tree,*p,*q;
tree=new  node(); p=new  node(); q=new  node();
tree ->data ='A' ;  p ->data='B';  q ->data='C';
tree->left=p;  tree->right=NULL;
p->left=NULL; p->right=q;
q->left=NULL; q->right=NULL;
```

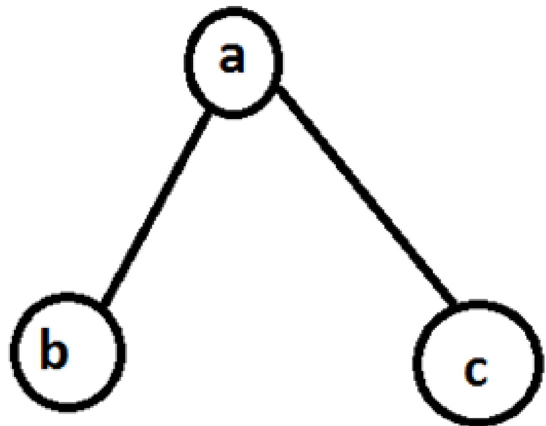


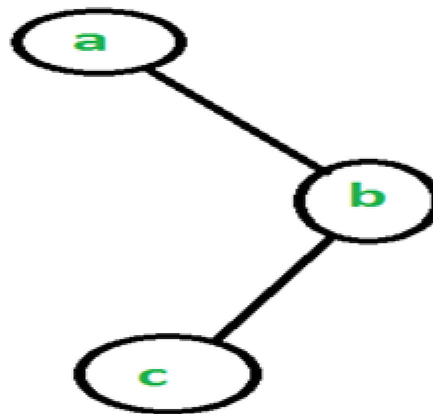
۱- روش پیش ترتیبی (Preorder):

- ملاقات ریشه
- پیمایش زیر درخت چپ به روش پیش ترتیبی
- پیمایش زیر درخت راست به روش پیش ترتیبی



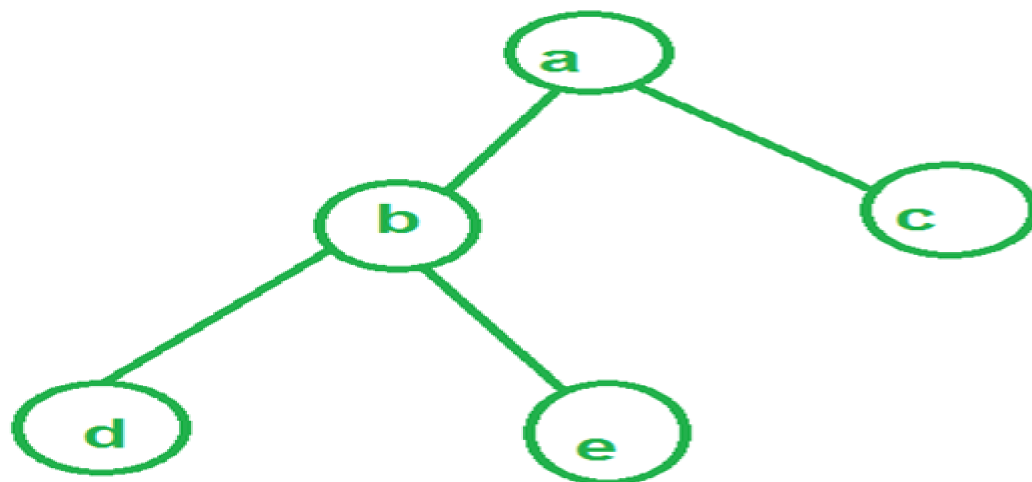

Preorder: vLR




Preorder: abc

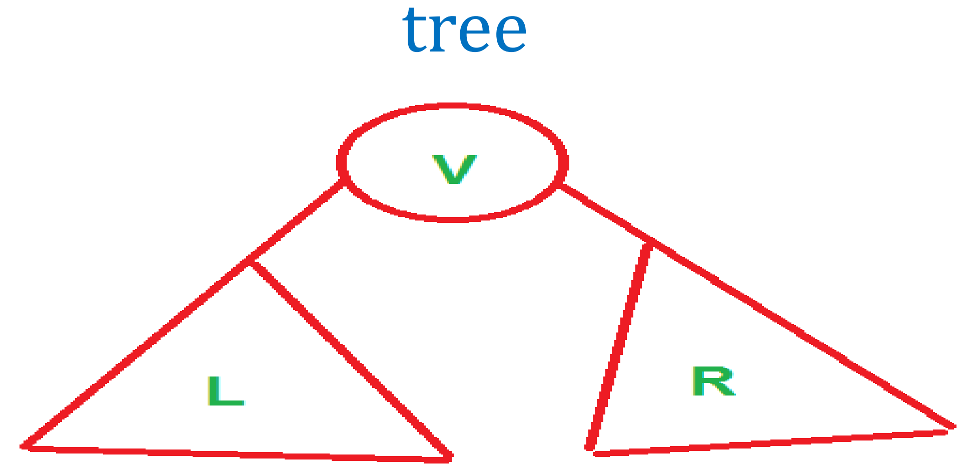
Preorder: abc




Preorder: abdec

## ❑ الگوریتم بازگشتی روش پیش ترتیبی

```
void preorder ( node *tree)
{
    if (tree) {
        cout<< tree->data;
        preorder(tree->left);
        preorder(tree->right);
    }
}
```

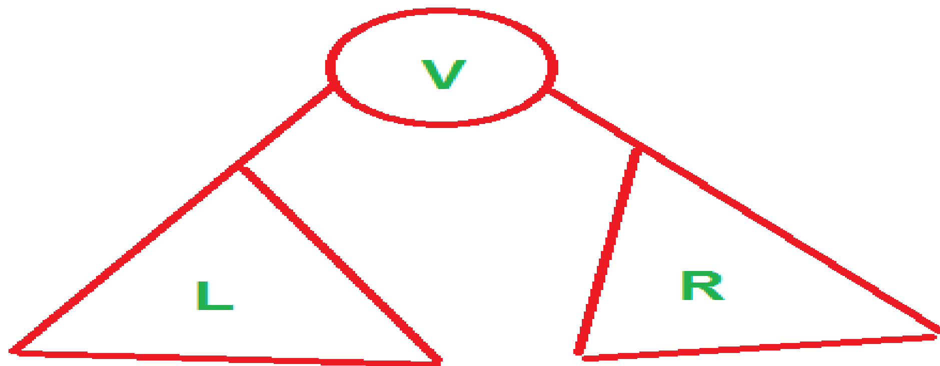



Preorder: vLR

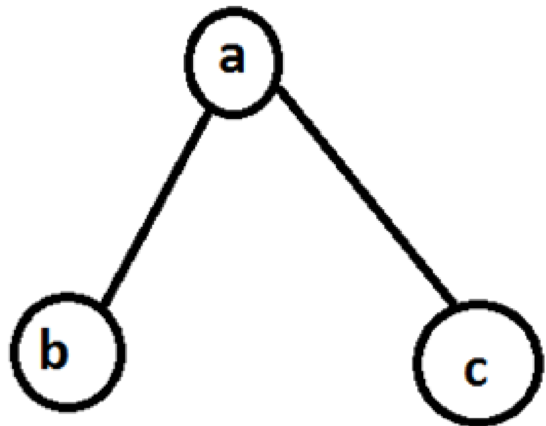
# پیمایش درختهای دودویی- ادامه

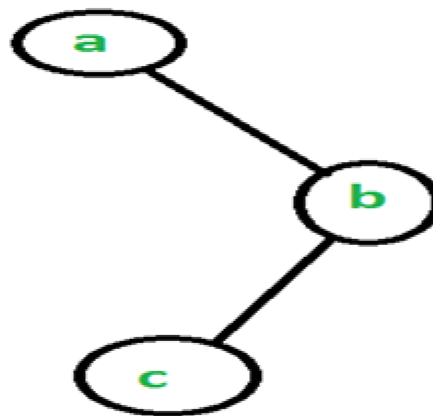
۲- روش میان ترتیبی (Inorder):

- پیمایش زیر درخت چپ به روش میان ترتیبی
- ملاقات ریشه
- پیمایش زیر درخت راست به روش میان ترتیبی



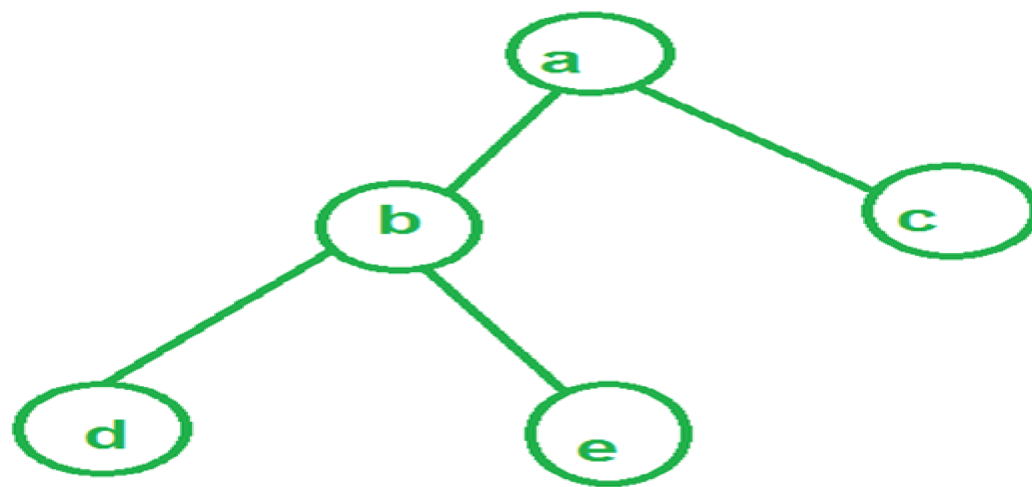

Inorder: LvR




Inorder: acb

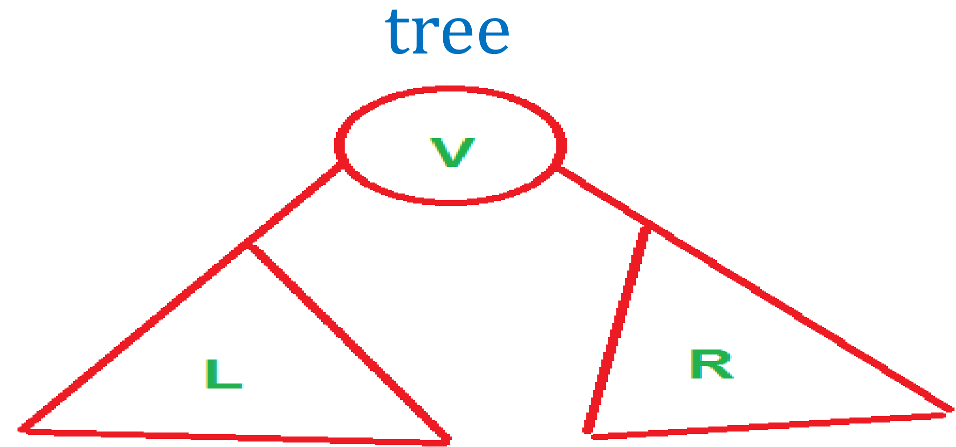
Inorder: bac




Inorder: dbeac

## ❑ الگوریتم بازگشتی روش میان ترتیبی

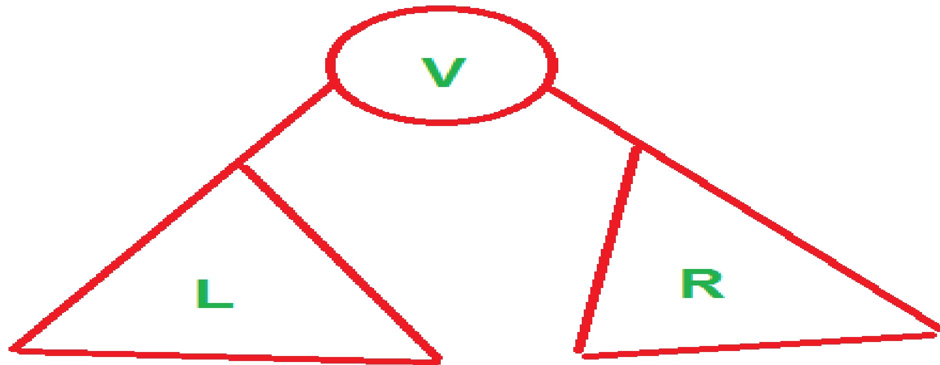
```
void inorder ( node *tree)
{
    if (tree) {
        inorder(tree->left);
        cout<< tree->data;
        inorder(tree->right);
    }
}
```



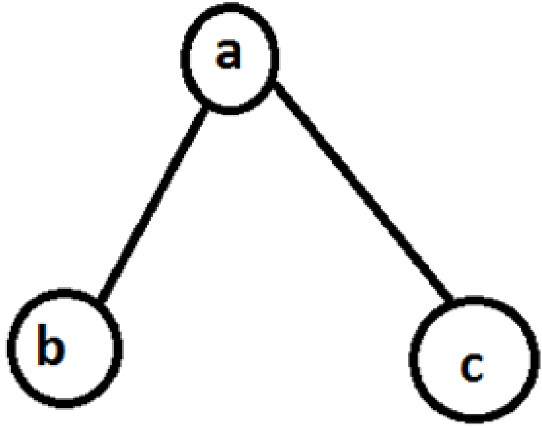

Inorder: LvR

## ۳- روش پس ترتیبی (Postorder):

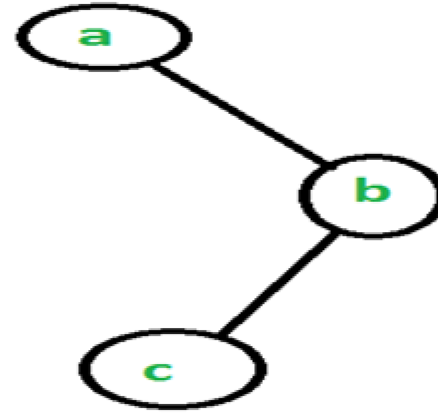
- پیمایش زیر درخت چپ به روش پس ترتیبی
- پیمایش زیر درخت راست به روش پس ترتیبی
- ملاقات ریشه



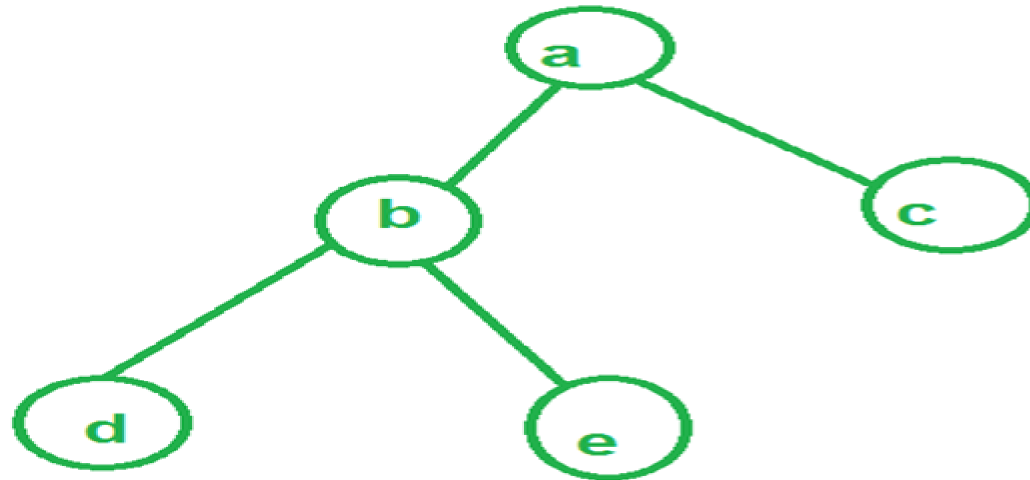

Postorder: LRv




Postorder: bca



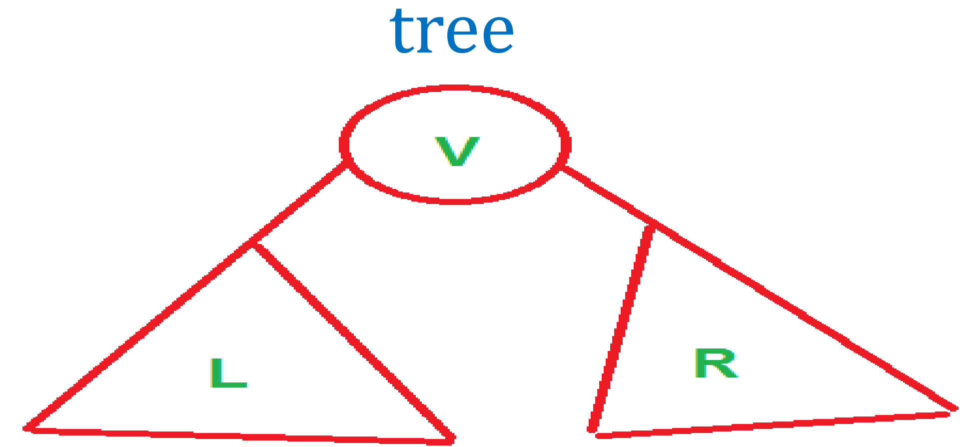

Postorder: cba




Postorder: debca

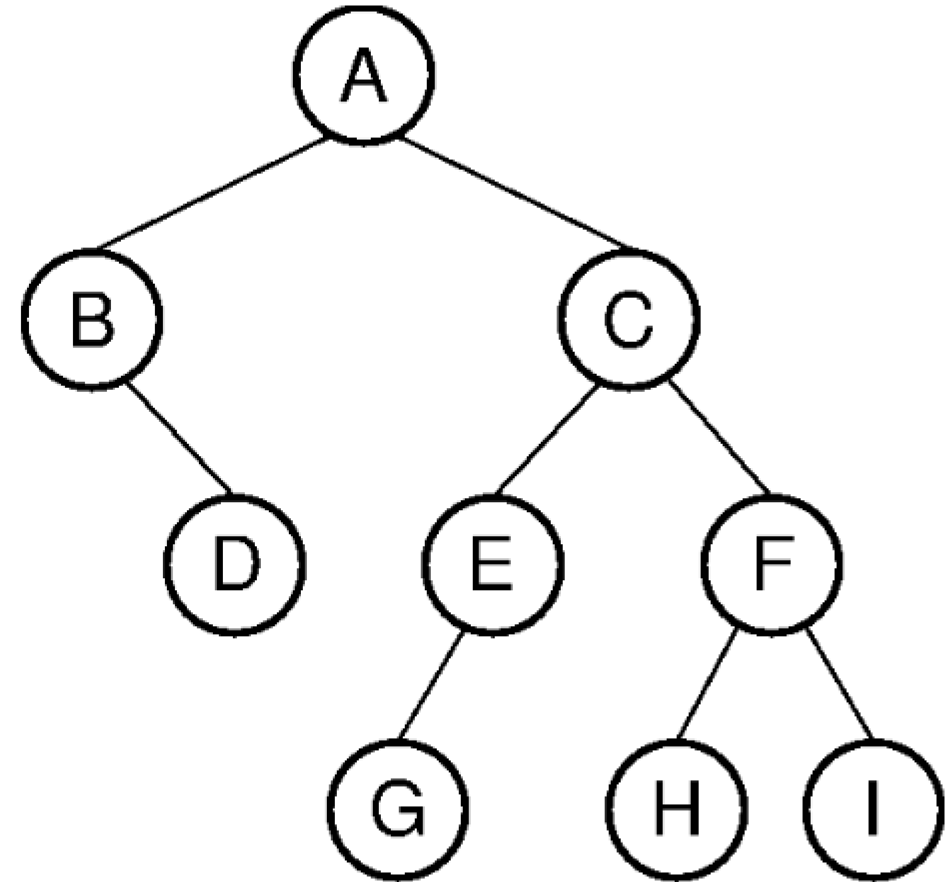
## ❑ الگوریتم بازگشتی روش پس ترتیبی

```
void postorder ( node *tree)
{
    if (tree) {
          postorder(tree->left);
          postorder(tree->right);
         cout<< tree->data;
     }
}
```




Postorder: LRv

❑ کار در کلاس: درخت دودویی زیر را به سه روش مختلف پیمایش کنید.




Preorder: ABDCEGFHI

Inorder: BDAGECHFI

Postorder: DBGEHIFCA

❑ الگوریتم غیر بازگشتی روش میان ترتیبی

```
#define MAX_STACK   100
node  *s[MAX_STACK];
void inorder2 (node  *tree)
{
   for(;;) {
      for( ; tree ; tree=tree->left)
         push(tree);
      tree= pop();
      if(!tree)
            break;
       cout<<tree->data;
      tree=tree->right;
    }
}
```

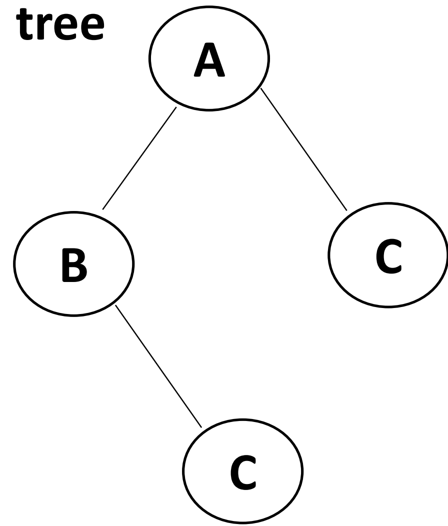
پشته ای را تعریف می کند که عناصر آن، اشاره گر به گره است.

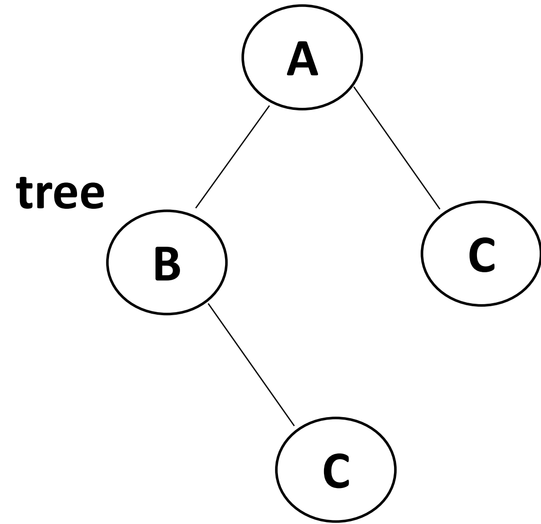
به زیردرخت های چپ رفته و اشاره گرهای tree را تا موقعی که به NULL نرسیده است به داخل پشته push می کند.

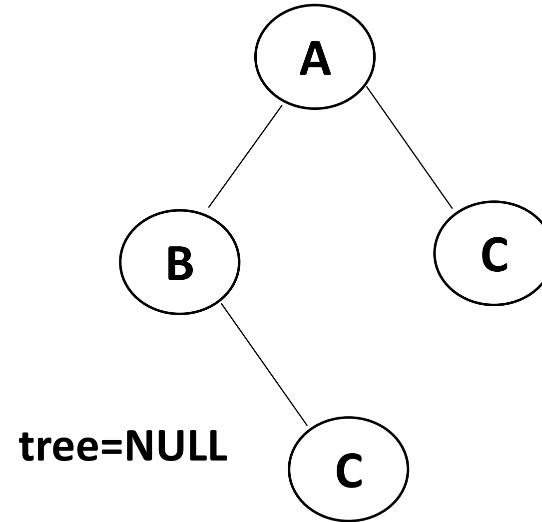
آخرین اشاره گر موجود در پشته را pop می کند.

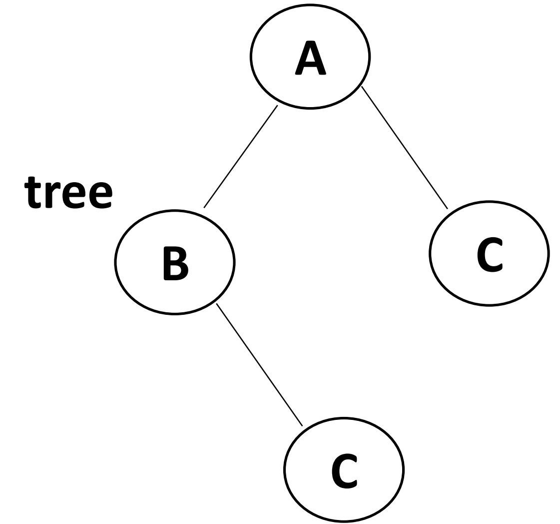
بعد از چاپ محتوای اشاره گر tree، اشاره گر سمت راست را در tree می گذارد.

❑ مثال: الگوریتم قبلی را روی درخت زیر trace کنید.

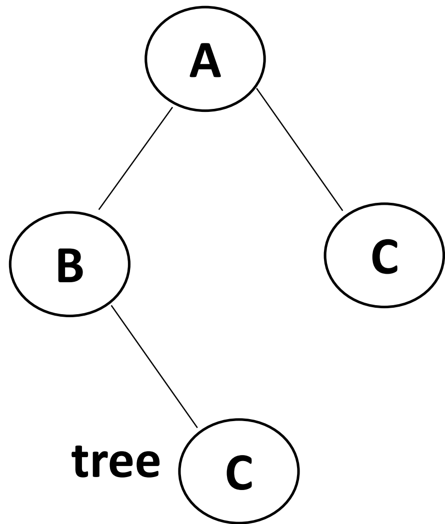


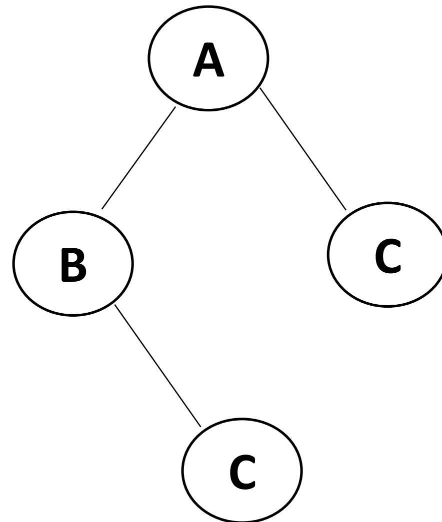



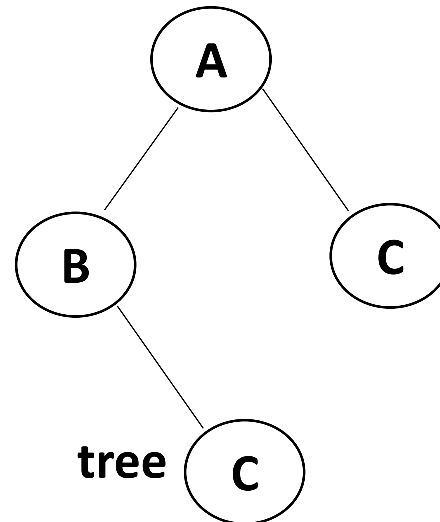




cout << 'B'

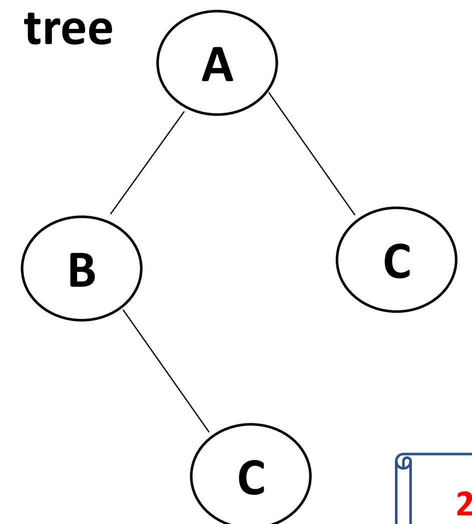


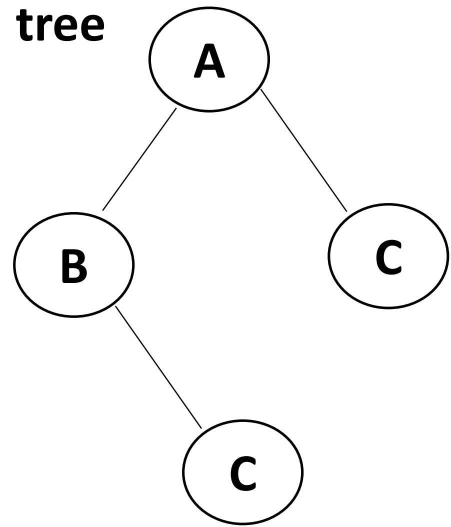




tree=NULL

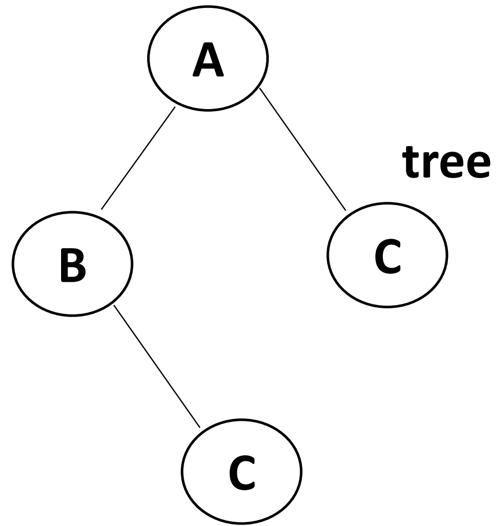


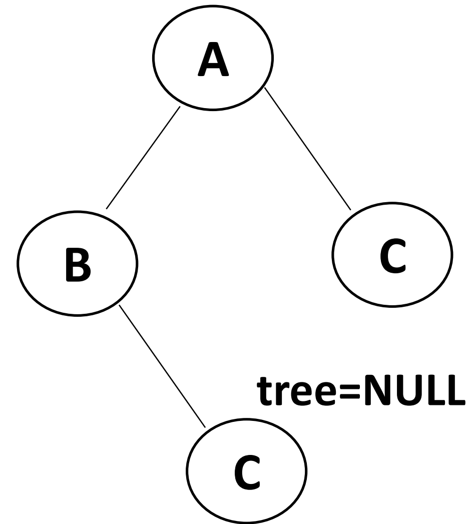

cout << 'C'

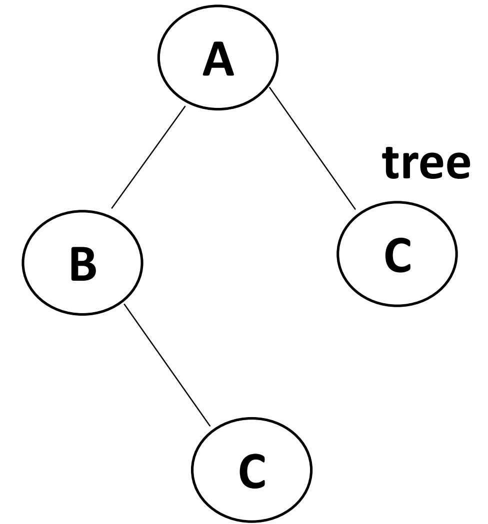




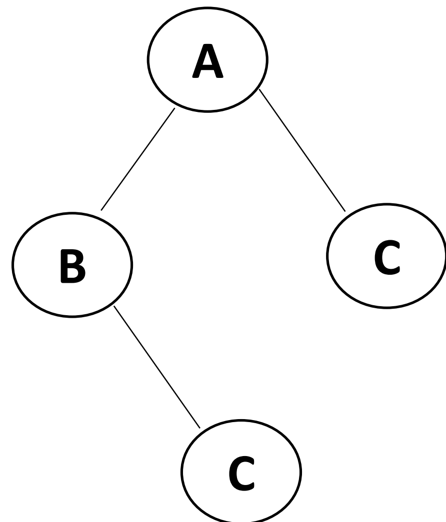

cout << 'A'




cout << 'C'




tree=NULL

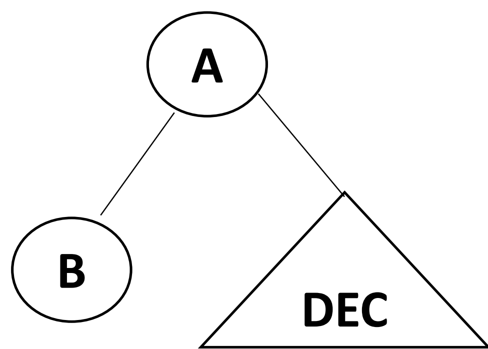
break

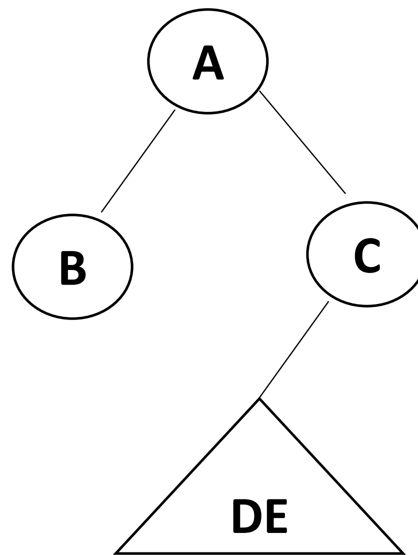
# ساختن درخت دودویی با استفاده از انواع پیمایش های ترتیبی

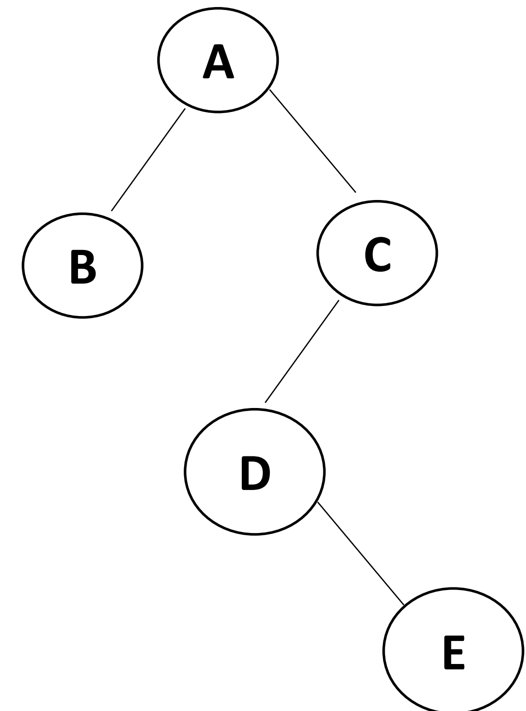
- اگر پیمایش های preorder و inorder یک درخت دودویی موجود باشند می توان یک درخت دودویی منحصر بفرد ساخت. ریشه های درخت اصلی و زیردرختان را از پیمایش پیش ترتیبی و زیردرختان چپ و راست را از پیمایش میان ترتیبی تشخیص می دهیم.

مثال: یک درخت دودویی بسازید که

$$\begin{cases} \text{preorder: } A\ B\ C\ D\ E \\ \text{inorder: } B\ A\ D\ E\ C \end{cases}$$

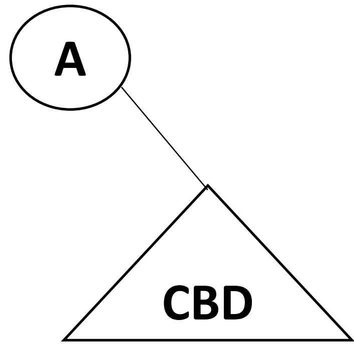


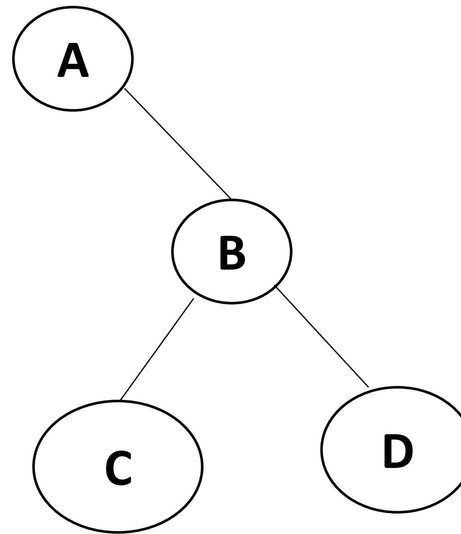




کار در کلاس: یک درخت دودویی بسازید که

$$\begin{cases} \text{preorder: } A\ B\ C\ D \\ \text{inorder: } A\ C\ B\ D \end{cases}$$

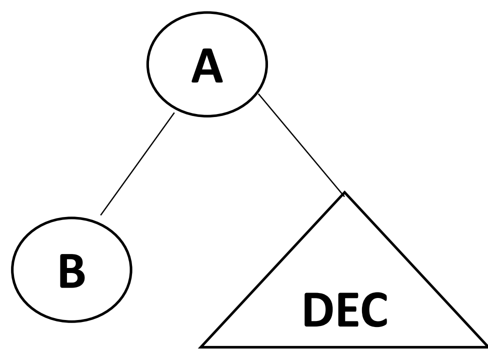


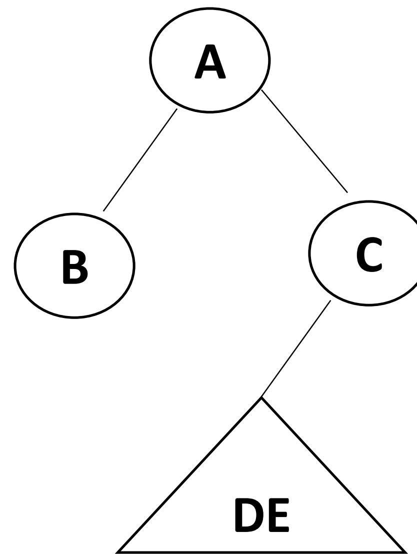


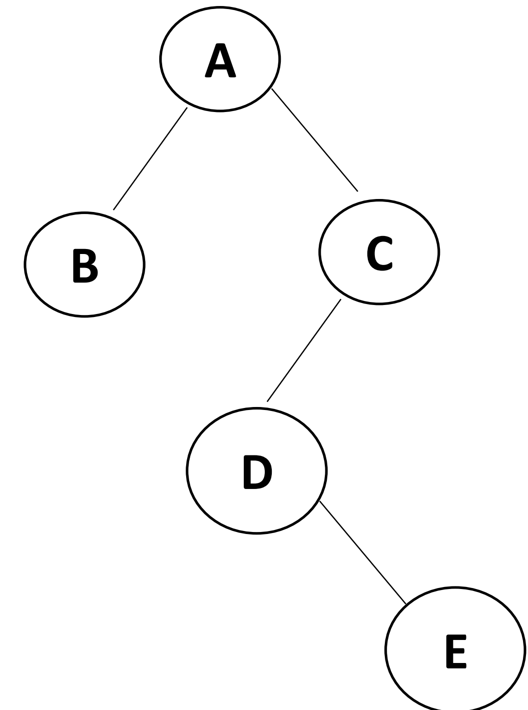
- اگر پیمایش های postorder و inorder یک درخت دودویی موجود باشند می توان یک درخت دودویی منحصر بفرد ساخت. ریشه های درخت اصلی و زیردرختان را از پیمایش پس ترتیبی و زیردرختان چپ و راست را از پیمایش میان ترتیبی تشخیص می دهیم.

مثال: یک درخت دودویی بسازید که

$$\begin{cases} \text{postorder: } B\ E\ D\ C\ A \\ \text{inorder: } B\ A\ D\ E\ C \end{cases}$$

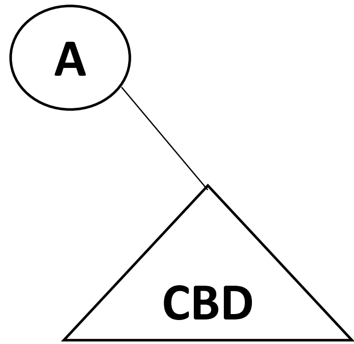


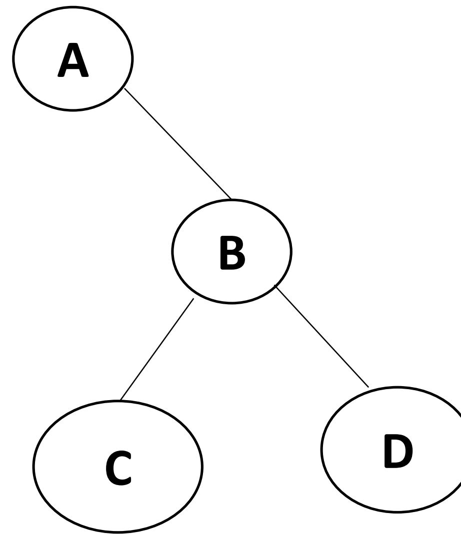




کار در کلاس: یک درخت دودویی بسازید که

$$\begin{cases} \text{postorder: } C\ D\ B\ A \\ \text{inorder: } A\ C\ B\ D \end{cases}$$



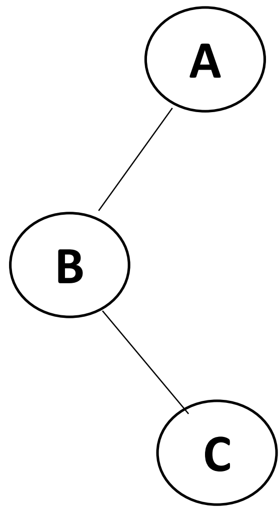


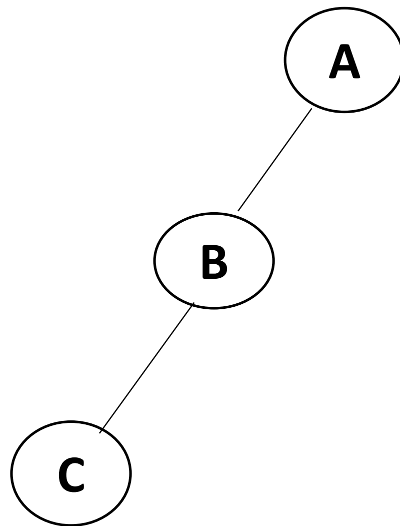
# ساختن درخت دودویی با استفاده از انواع پیمایش های ترتیبی- ادامه

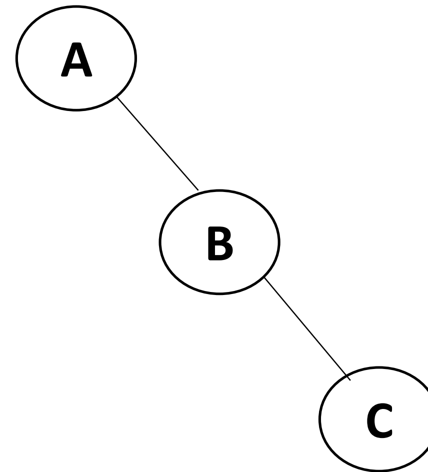
- اگر پیمایش های postorder و preorder یک درخت دودویی موجود باشند نمی توان یک درخت دودویی منحصر بفرد ساخت.

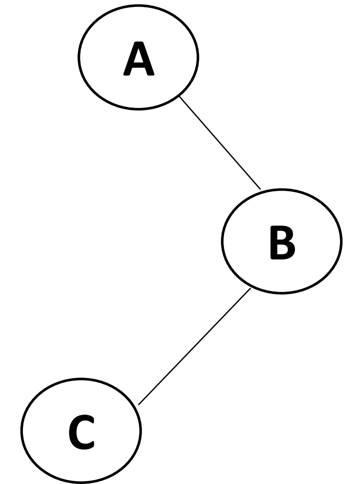
مثال: یک درخت دودویی بسازید که

$$\begin{cases} \text{postorder: } C\ B\ A \\ \text{preorder: } A\ B\ C \end{cases}$$



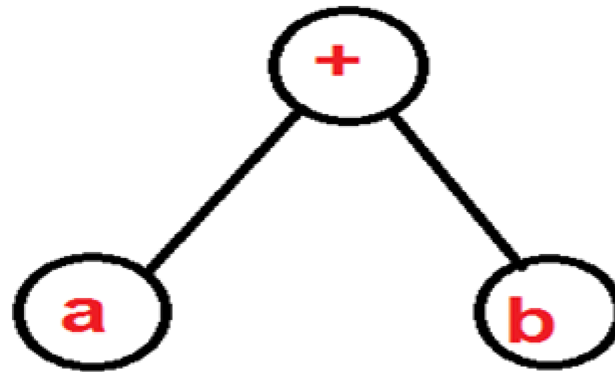



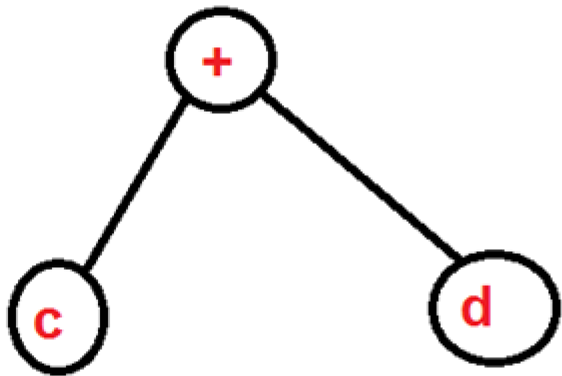
# درخت عبارت محاسباتی

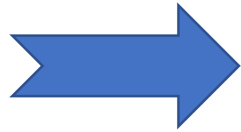
- نوعی درخت دودویی است که برای نمایش عبارات ریاضی بکار می رود.
- برای رسم چنین درختی، ابتدا عبارت ریاضی را بر اساس اولویت ها پرانتزگذاری می کنیم و سپس از داخلی ترین پرانتز شروع کرده و درخت معادل را رسم می کنیم.
- مثال: درخت عبارت محاسباتی a+b را رسم کنید و معادل prefix و postfix آنرا بدست آورید.

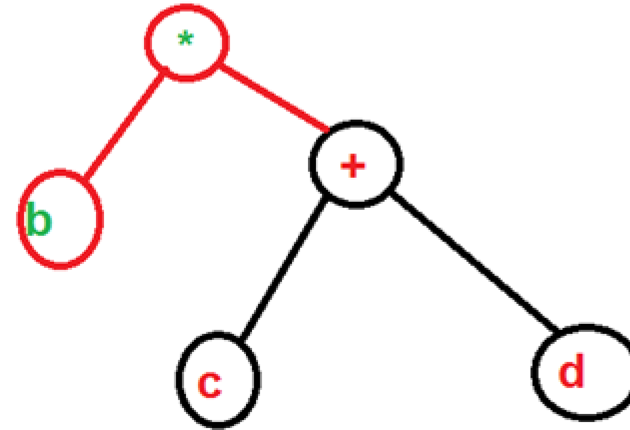


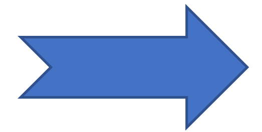

Preoreder=prefix = +ab
Postorder=postfix= ab+

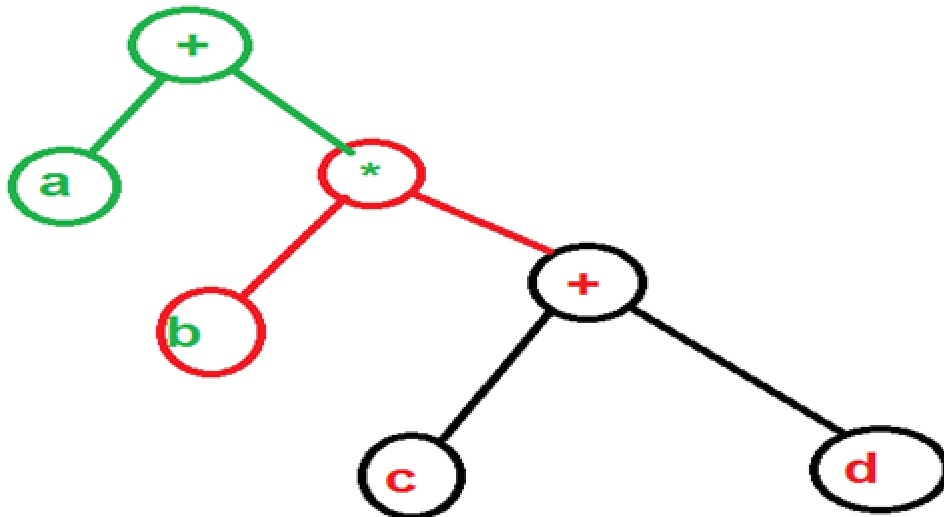
❑ مثال: درخت عبارت محاسباتی a+b*(c+d) را رسم کنید و معادل prefix و postfix آنرا بدست آورید. ابتدا پرانتزگذاری می کنیم. a+(b*(c+d))





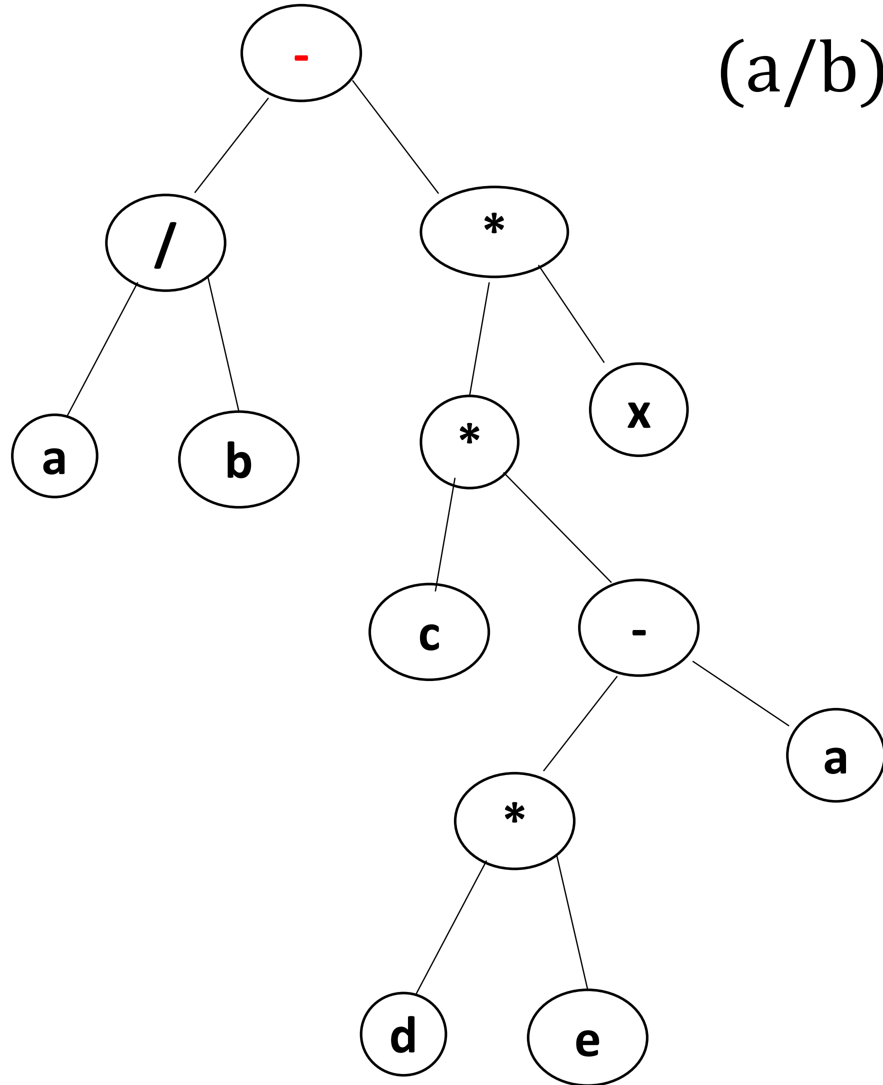



**Pre: + a * b + c d**

**Post: a b c d +* +**

❑ کار در کلاس: درخت عبارت محاسباتی a/b-c*(d*e-a)*x را رسم کنید و معادل prefix و postfix آنرا بدست آورید. ابتدا پرانتزگذاری می کنیم.

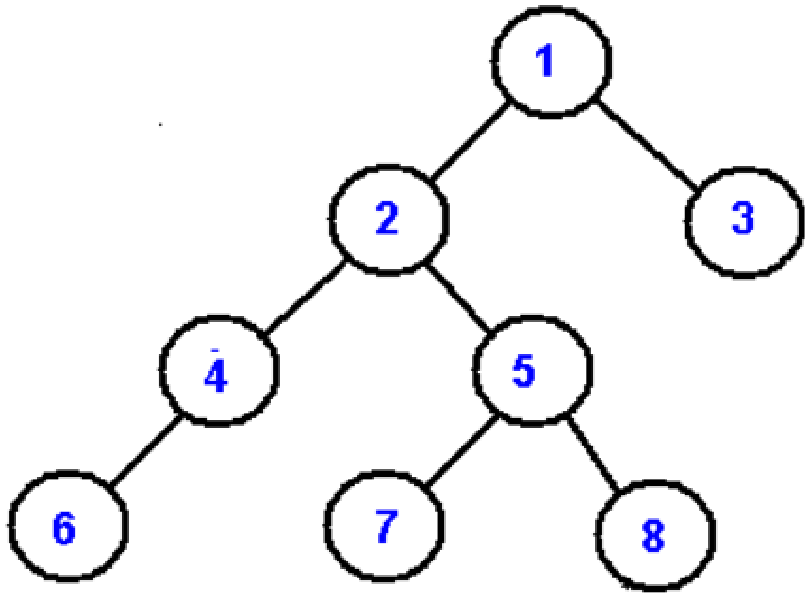
(a/b) - ((c * ((d*e) - a)) * x)




preorder= prefix: - /ab * * c- *deax

postorder=postfix: ab/cde*a-*x*-

۱- درخت دودویی زیر را به سه روش مختلف پیمایش کنید.




۲- یک درخت دودویی بسازید که

$$\begin{cases} \text{preorder: } A\ B\ C\ D\ \ E\ F\ G \\ \text{inorder: } \ B\ D\ C\ A\ F\ G\ E \end{cases}$$

۳- درخت عبارت محاسباتی (a+b)*(c/d-e)+f*g*h را رسم کنید و معادل prefix و postfix آنرا بدست آورید.

# درخت جستجوی دودویی (Binary Search Tree: BST)

❑ نوعی درخت دودویی است که ممکن است تهی باشد. در صورت غیرتهی بودن باید خصوصیات زیر را داشته باشد:

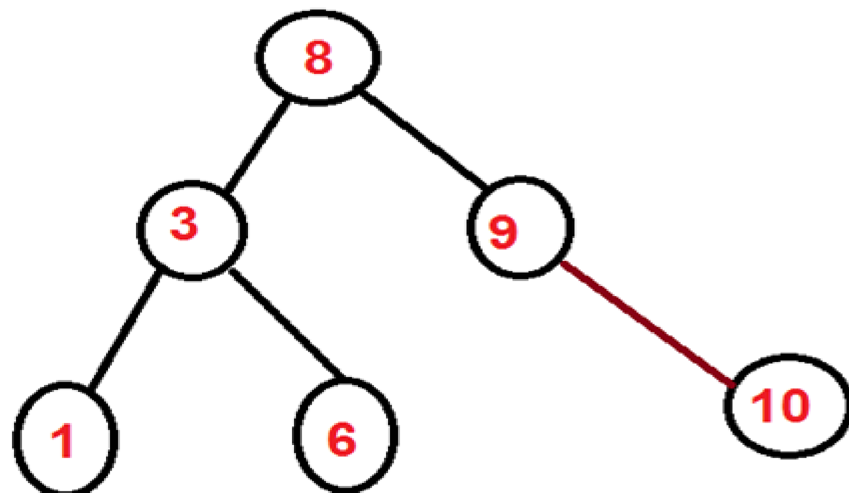
۱- گره ها نباید دارای مقادیر یکسان باشند.

۲- مقادیر گره های واقع در زیردرخت چپ باید کمتر از مقدار گره ریشه باشند.

۳- مقادیر گره های واقع در زیردرخت راست باید بیشتر از مقدار گره ریشه باشند.

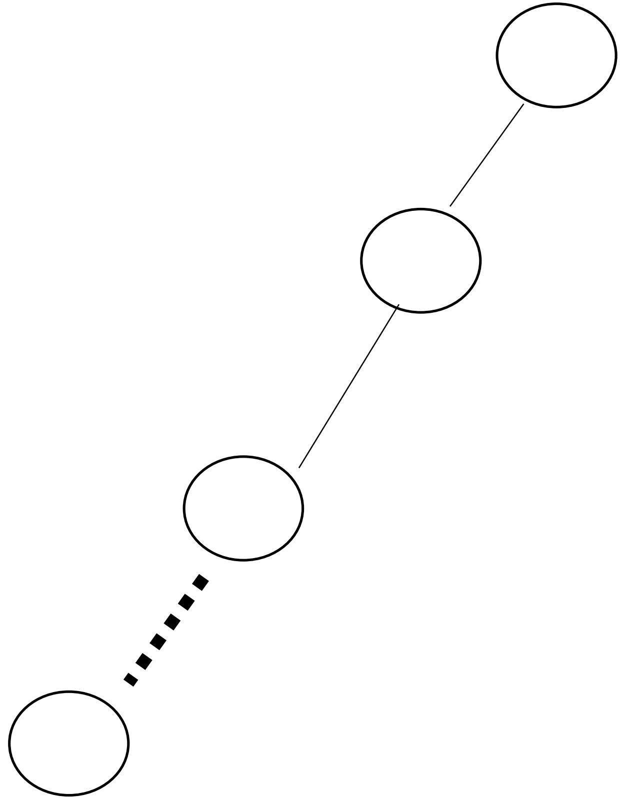
۴- زیردرختان چپ و راست نیز باید درخت جستجوی دودویی باشند.

بعنوان مثال: درخت زیر یک BST است.

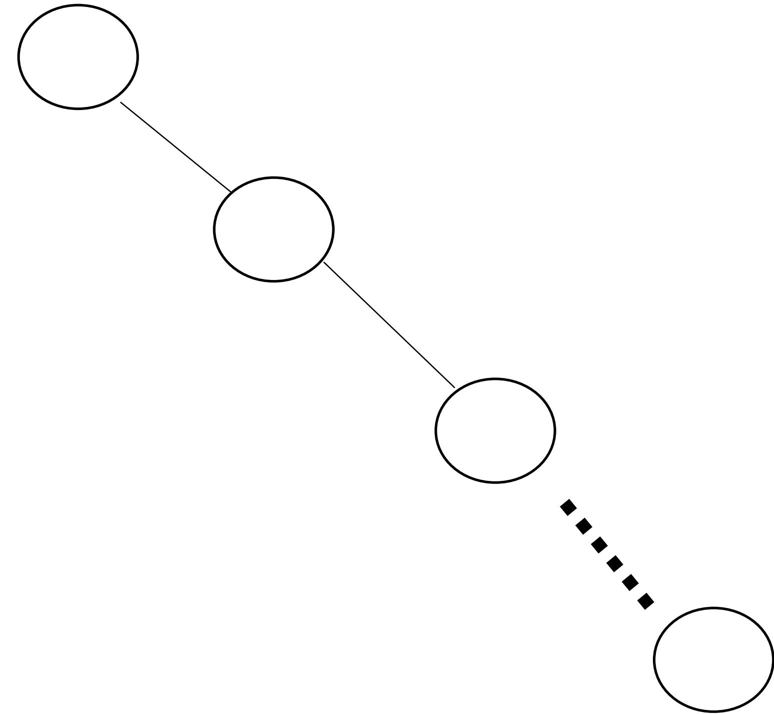


# بررسی ارتفاع درخت جستجوی دودویی

- فرضاً $h$ ارتفاع درخت جستجوی دودویی با $n$ گره باشد بدترین حالت برای $h$ موقعی است که درخت بصورت یک لیست مورب باشد.

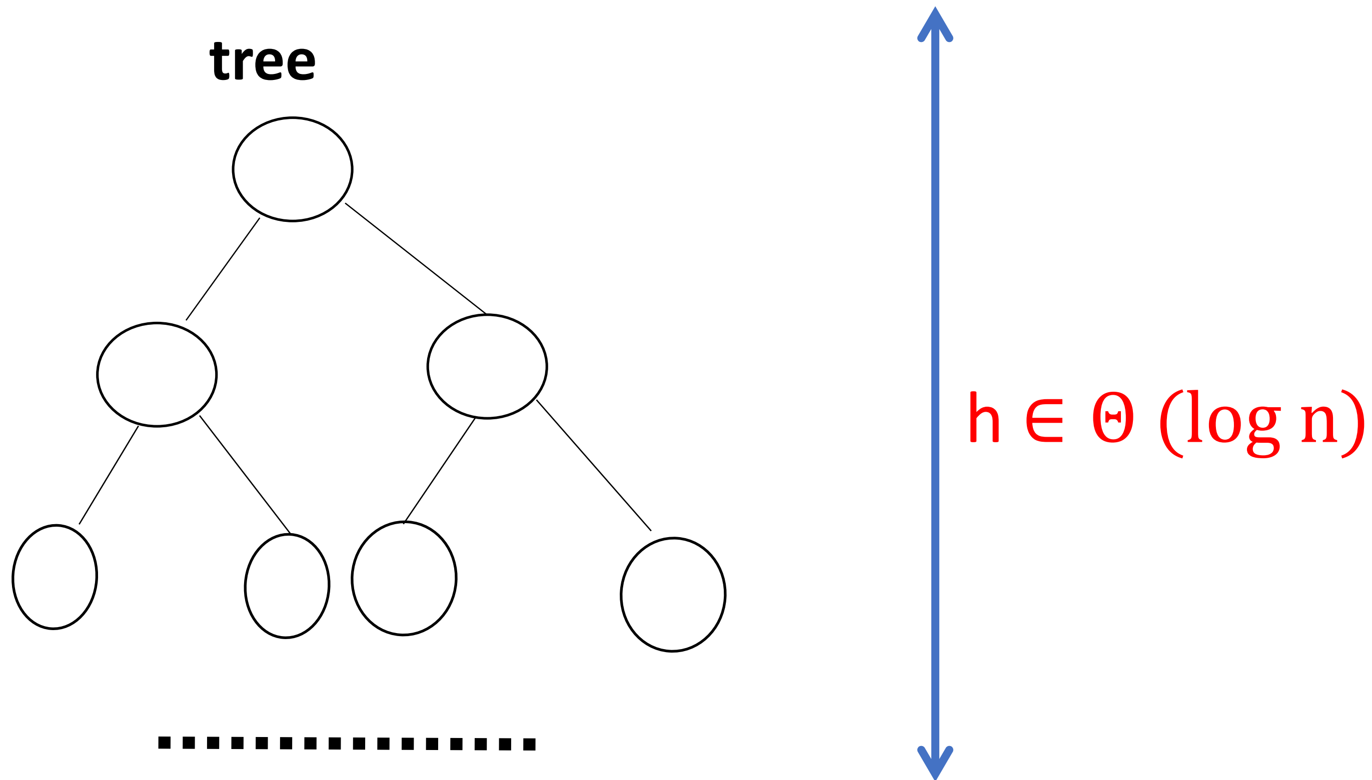



$h=n-1 \in O(n)$



# بررسی ارتفاع درخت جستجوی دودویی - ادامه

- بهترین حالت یا حالت متوسط برای h موقعی است که درخت بصورت زیر باشد.

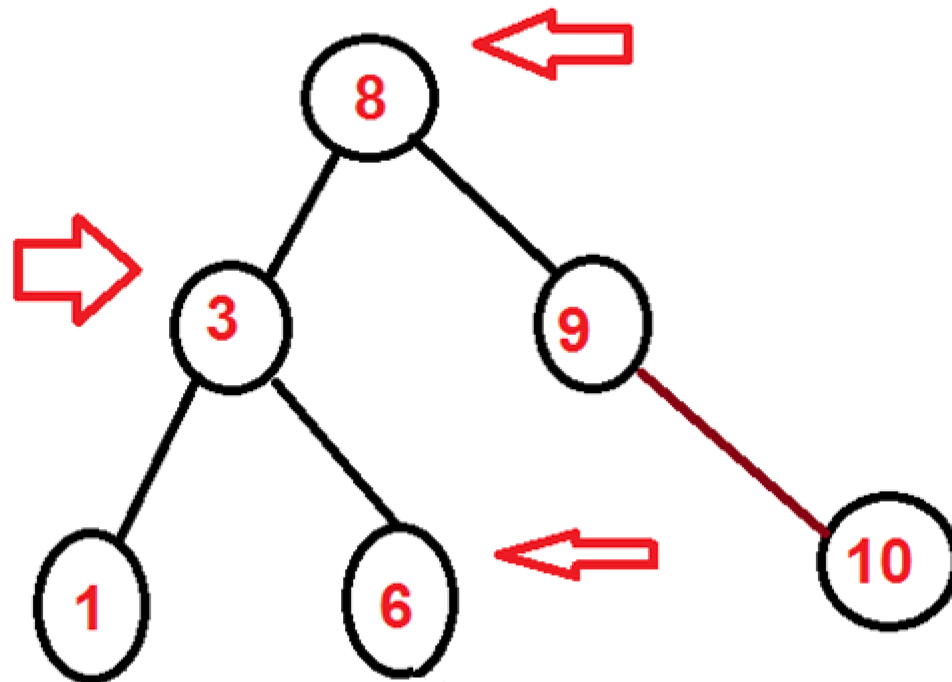



$h \in \Theta(\log n)$

# الگوریتم جستجوی یک BST

- برای جستجوی عدد x در یک درخت BST ابتدا آن را با ریشه درخت مقایسه می کنیم اگر برابر بود به جواب رسیدیم. اگر کوچکتر بود به سمت چپ درخت رفته و جستجو را ادامه می دهیم و اگر بزرگتر بود به سمت راست درخت رفته جستجو را ادامه می دهیم.

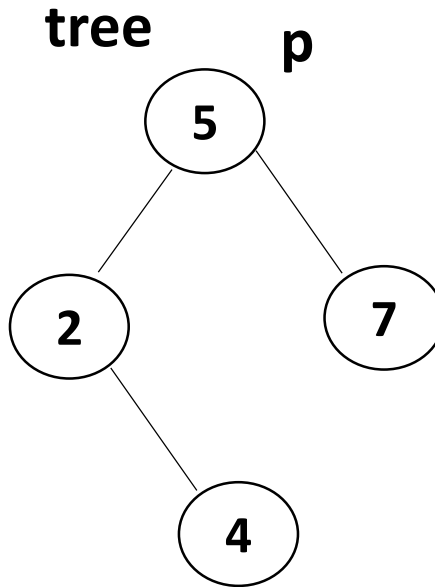
- بعنوان مثال: جستجوی عدد x=6 در درخت زیر

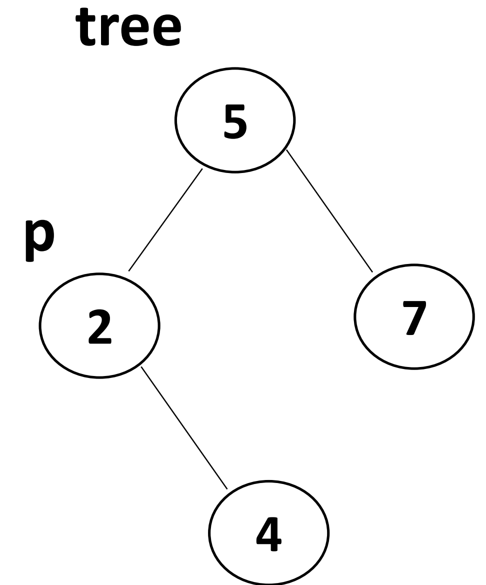


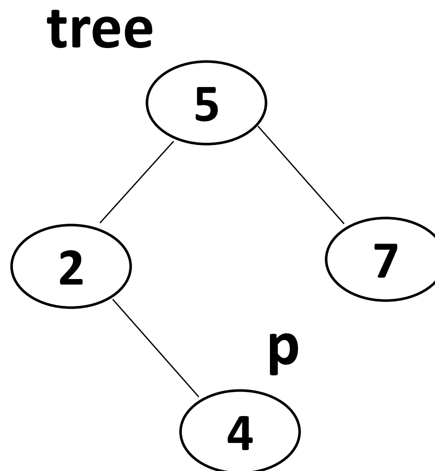
این تابع گره ای را پیدا کرده که مقدار آن برابر x است
سپس اشاره گر به آن گره را بر می گرداند.

**x=4**

```
node  *search(node  *tree,int x)
{
    node *p=tree;
    while(p!=NULL){
      if(x < p->data)
          p=p->left;
      else  if(x > p->data)
          p=p->right;
      else
          return p;
    }
    return NULL;
}
```





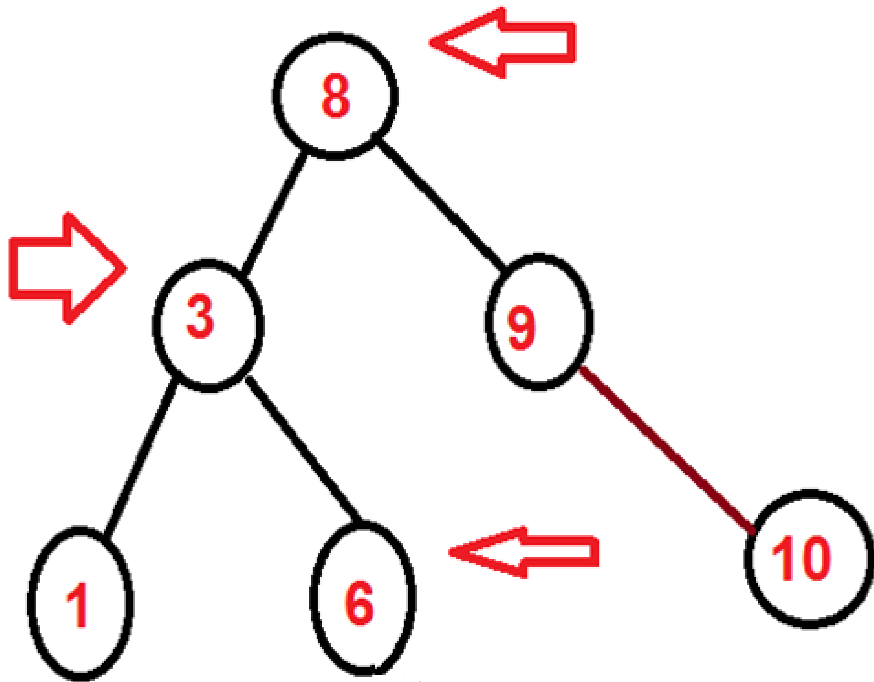



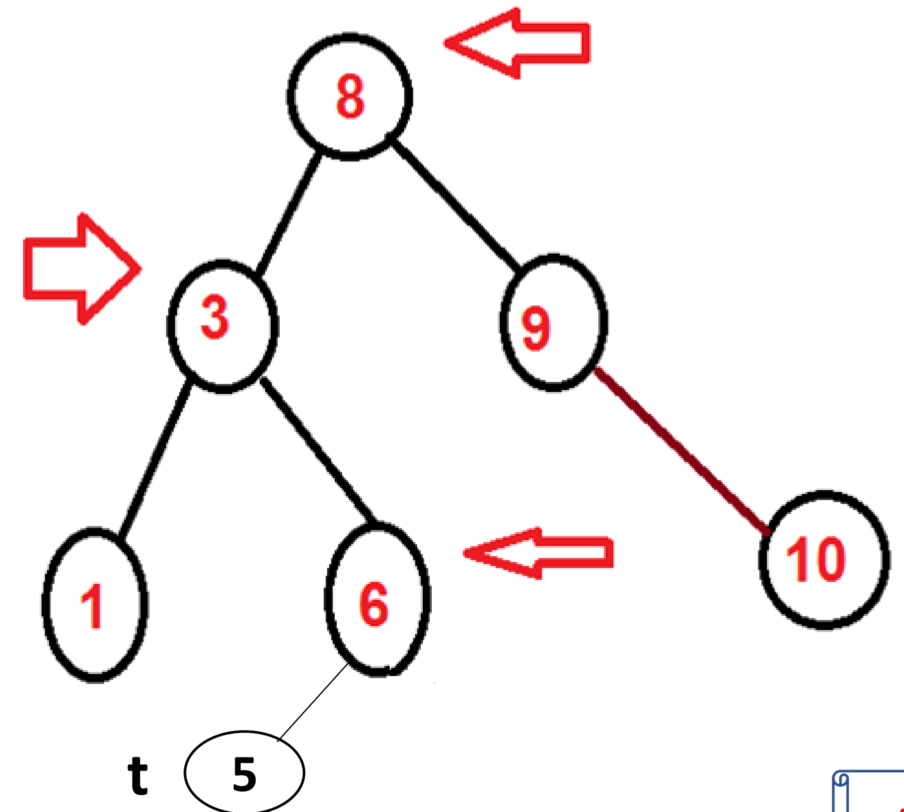
**retrun  p**

- زمان اجرای الگوریتم جستجو به ارتفاع درخت بستگی دارد.
- در بدترین حالت، ارتفاع درخت از مرتبه $O(n)$ است بنابراین زمان اجرای الگوریتم جستجو نیز از مرتبه $O(n)$ خواهد بود.
- در بهترین حالت یا حالت متوسط، ارتفاع درخت از مرتبه $\Theta(\log n)$ است بنابراین زمان اجرای الگوریتم جستجو نیز از مرتبه $\Theta(\log n)$ خواهد بود.

# الگوریتم درج به یک BST

- برای درج عدد x در یک درخت BST ابتدا باید جای آن را برای درج پیدا کنیم. بنابراین، مشابه روش جستجو، با ریشه درخت مقایسه می کنیم اگر کوچکتر بود به سمت چپ درخت رفته و الگوریتم را ادامه می دهیم و اگر بزرگتر بود به سمت راست درخت رفته و الگوریتم را ادامه می دهیم.
- بعنوان مثال: درج عدد x=5 در درخت زیر



```
void  insert(node  **tree, int x)
{
    node *p=*tree, *t=new node();
    t->data=x; t->left=NULL; t->right=NULL;
    if(*tree==NULL){  *tree=t;  return;  }
     while(p!=NULL){
         if(x < p->data)
              if (p->left !=NULL)
                  p=p->left;
              else
                  { p->left=t;  break ;}
         else
             if (p->right !=NULL)
                  p=p->right;
              else
                  { p->right=t;  break ;}
    }
}
```

حالت ۱: درخت تهی است.

حالت ۲: درخت تهی نیست.

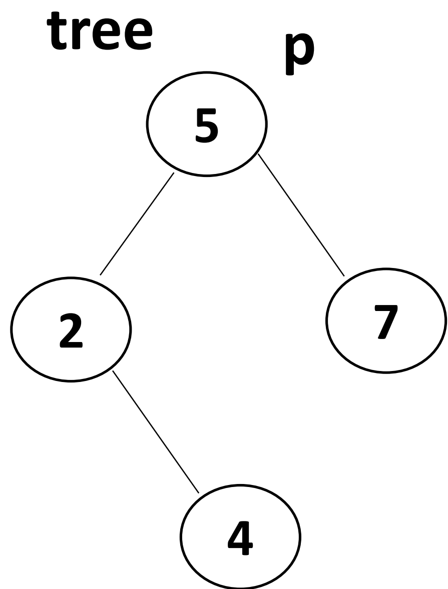
به سمت چپ حرکت می کند.

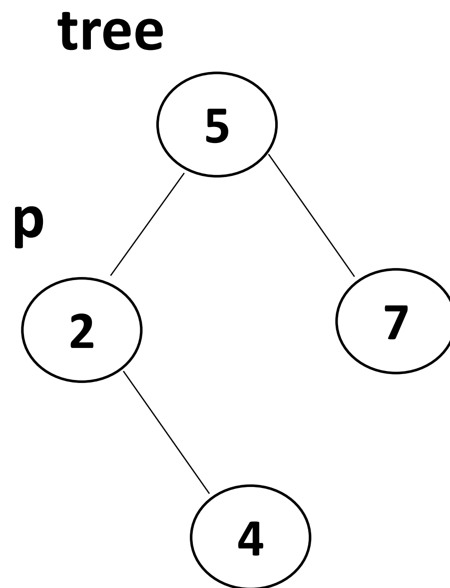
جای واقعی x در سمت چپ p است و در آنجا درج می کند.

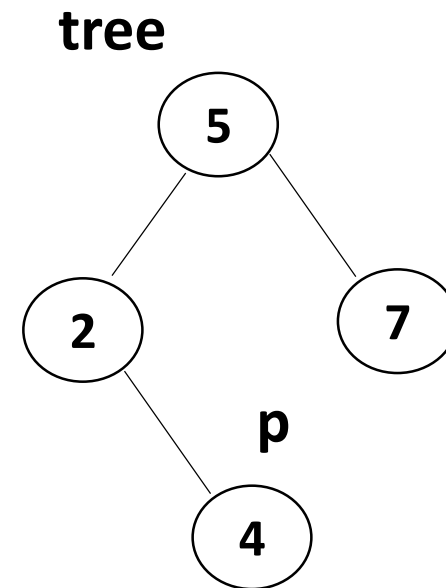
به سمت راست حرکت می کند.

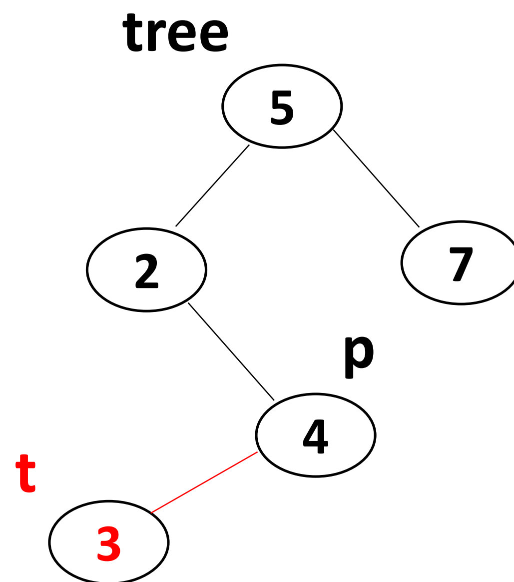
جای واقعی x در سمت راست p است و در آنجا درج می کند.

x=3





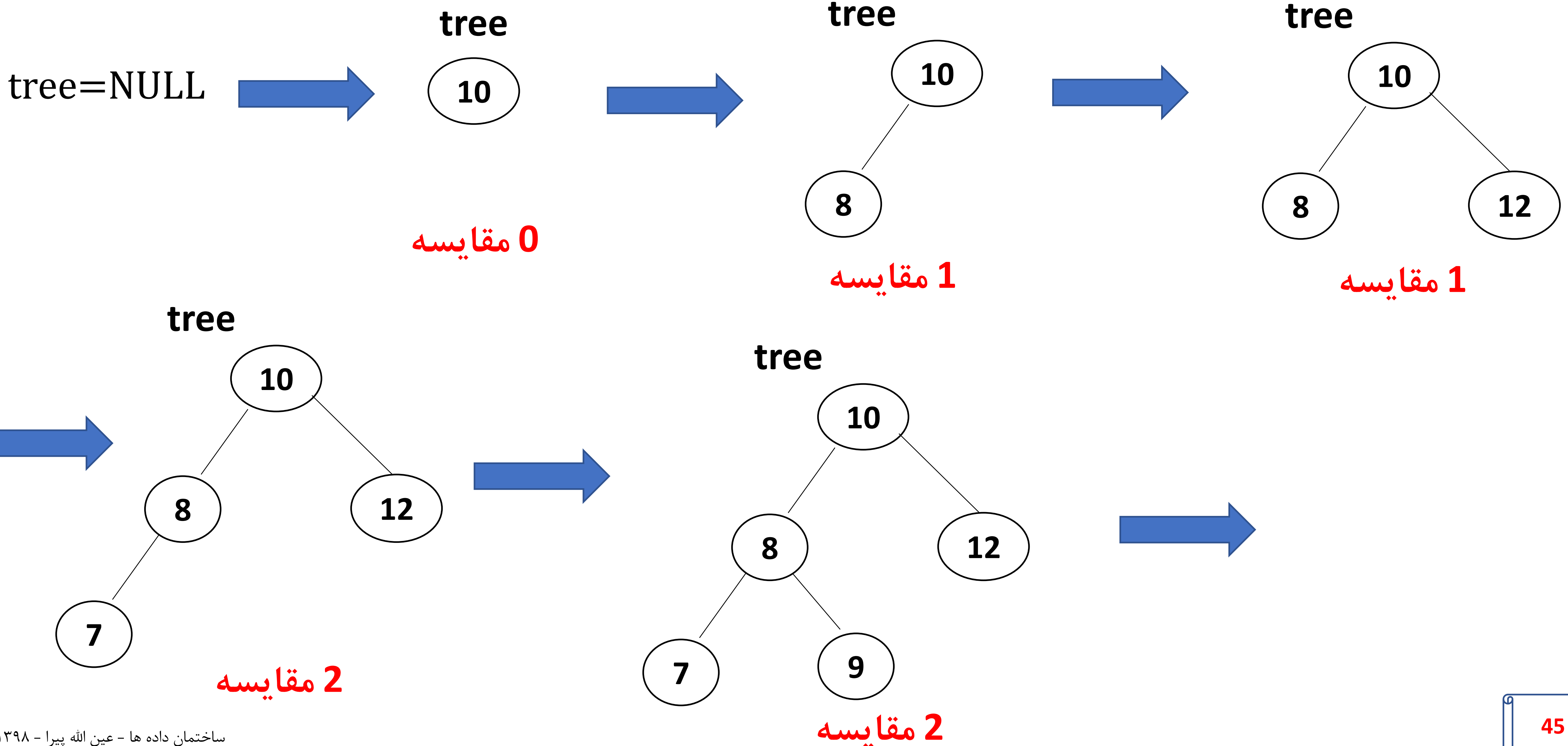


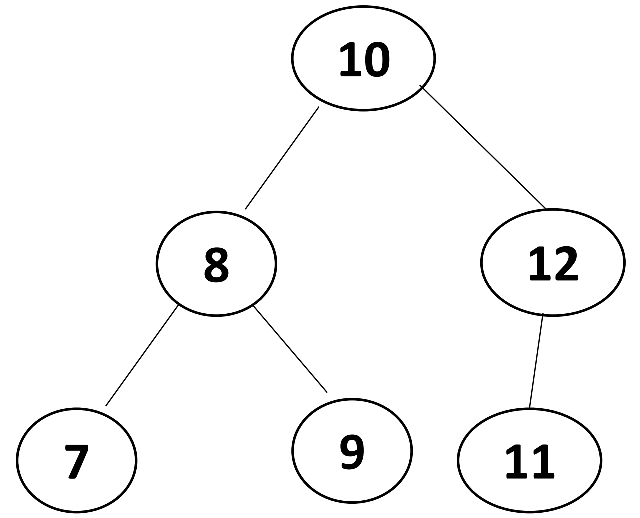


- زمان اجرای الگوریتم درج به ارتفاع درخت بستگی دارد.
- در بدترین حالت، ارتفاع درخت از مرتبه $O(n)$ است بنابراین زمان اجرای الگوریتم درج نیز از مرتبه $O(n)$ خواهد بود.
- در بهترین حالت یا حالت متوسط، ارتفاع درخت از مرتبه $\Theta(\log n)$ است بنابراین زمان اجرای الگوریتم درج نیز از مرتبه $\Theta(\log n)$ خواهد بود.

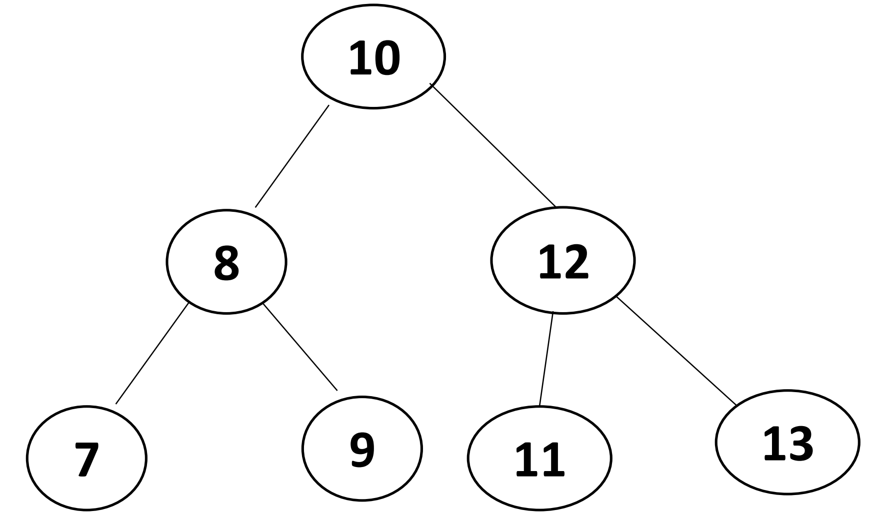
❑ کار در کلاس: با اعداد زیر یک BST بسازید. تعداد مقایسه ها  را حساب کنید.

10 , 8, 12, 7, 9, 11, 13






2 مقایسه




2 مقایسه

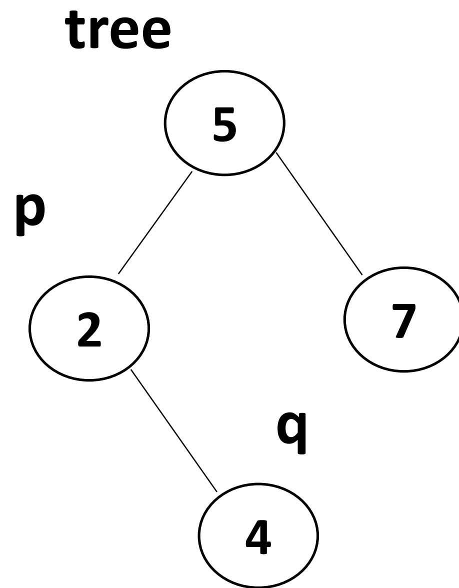
تعداد کل مقایسه ها = 1+1+2+2+2+2 =10

# الگوریتم حذف از یک BST

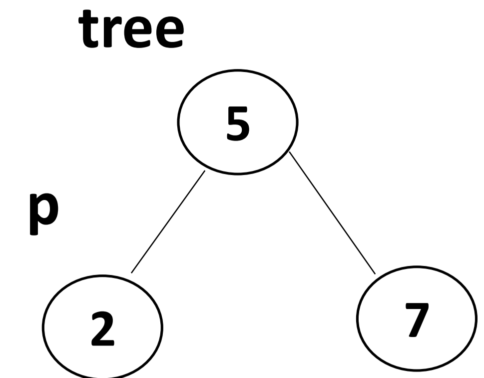
- برای حذف x از یک BST سه حالت زیر را خواهیم داشت:

حالت ۱: x برگ است: برای این حالت، کافی است گره حاوی x را پیدا کنیم. البته باید اشاره گر والدش را نیز داشته باشیم. سپس، اشاره گر left یا right والد را، بسته به اینکه x چپ آن است یا راست آن، برابر NULL قرار دهیم.

به عنوان مثال: در درخت زیر، گره با مقدار x=4 را حذف کنید.

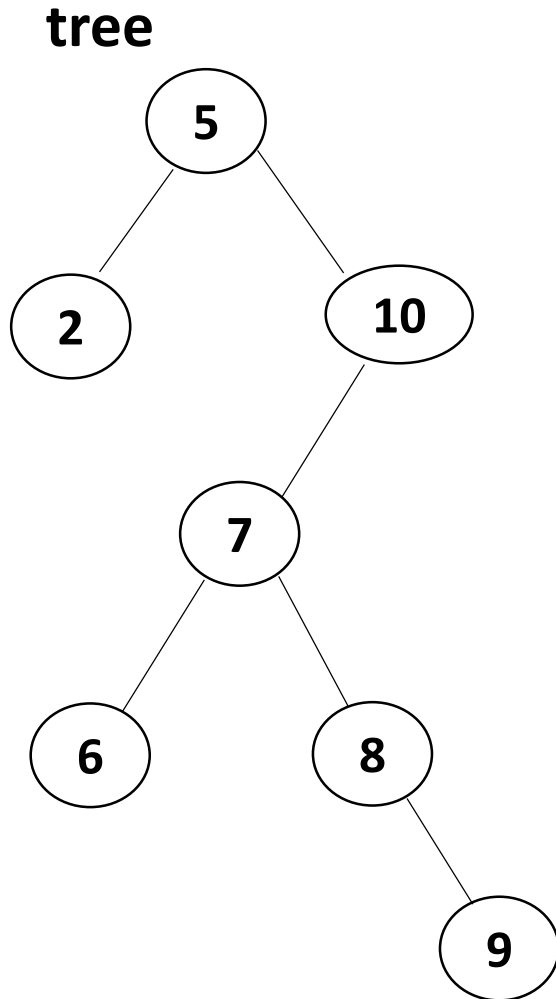


```
p->right=NULL;
free(q);
```

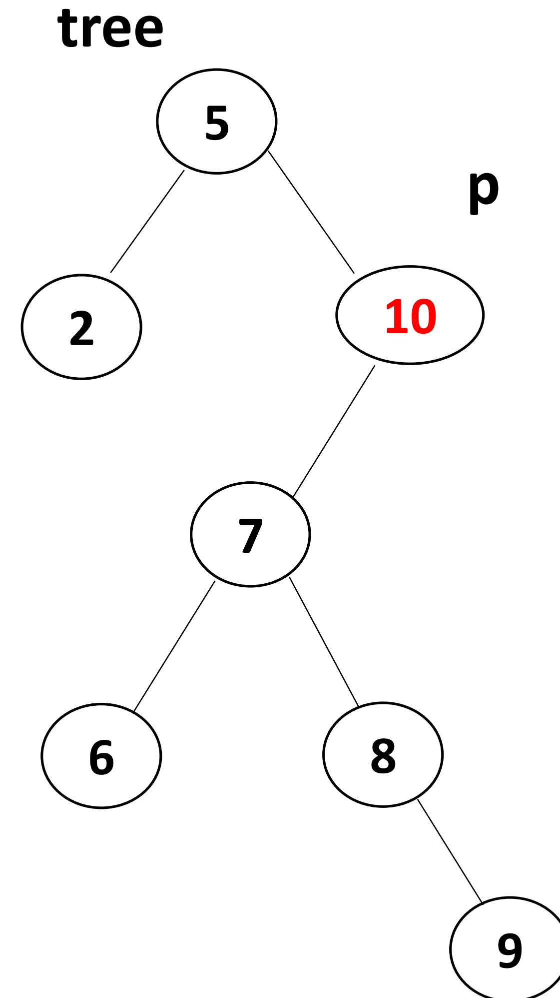


حالت ۲: X  فقط فرزند چپ دارد: برای این حالت، کافی است گره حاوی X را پیدا کنیم. سپس، بزرگترین بچه زیردرخت چپ را پیدا کرده و مقدار آن را بجای X نوشته و آن بچه بزرگتر را که اگر برگ باشد به روش حالت ۱ حذف اش می کنیم در غیر اینصورت بسته به اینکه چه حالتی است به روش بازگشتی آن را حذف می کنیم.

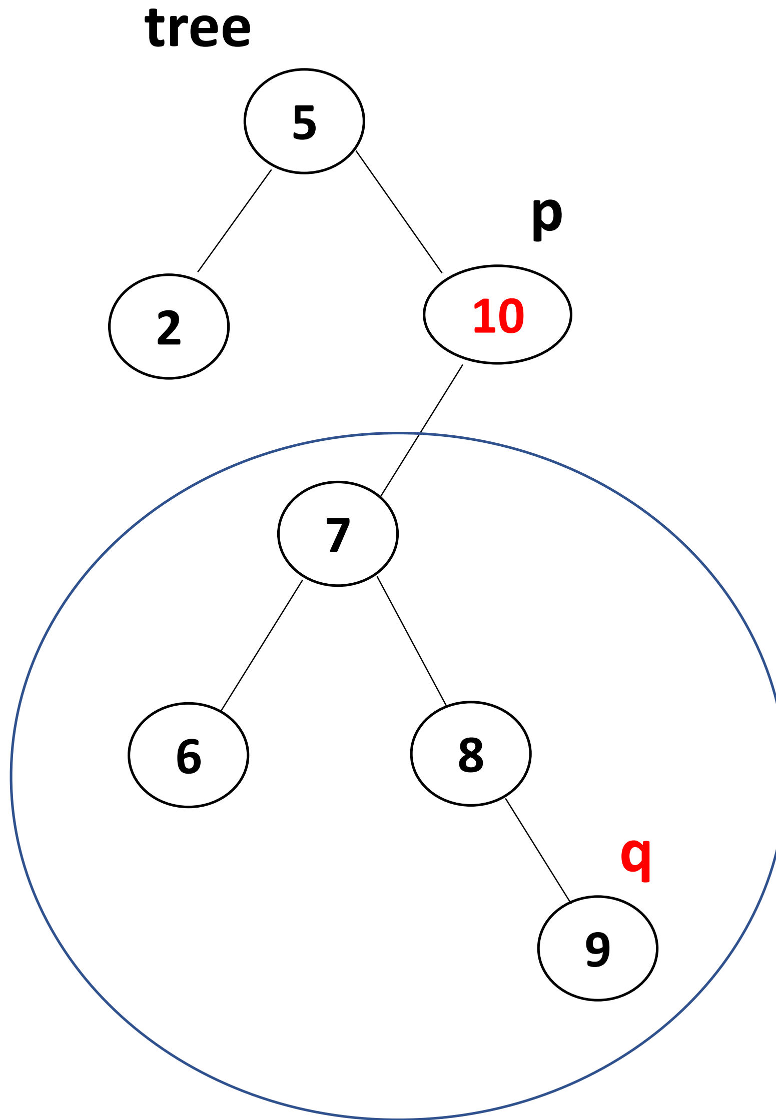
به عنوان مثال: در درخت زیر، گره با مقدار X=10 را حذف کنید.



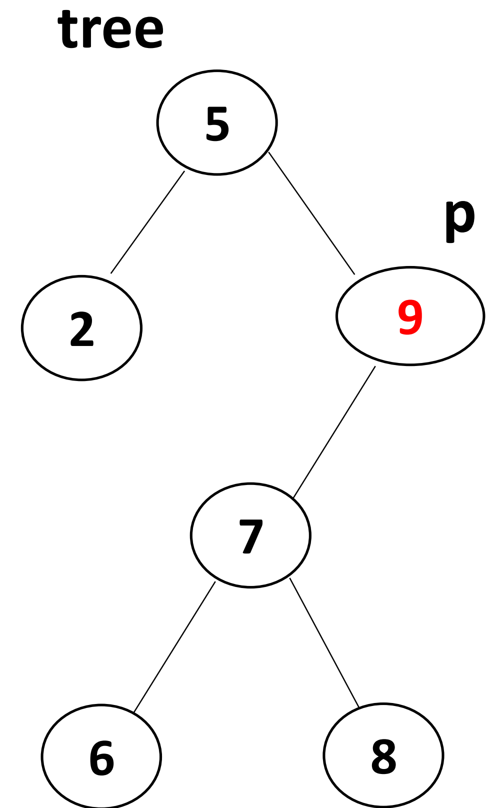
مرحله ۱: یافتن x



مرحله ۲: یافتن
بزرگترین فرزند چپ
گره با مقدار x=10

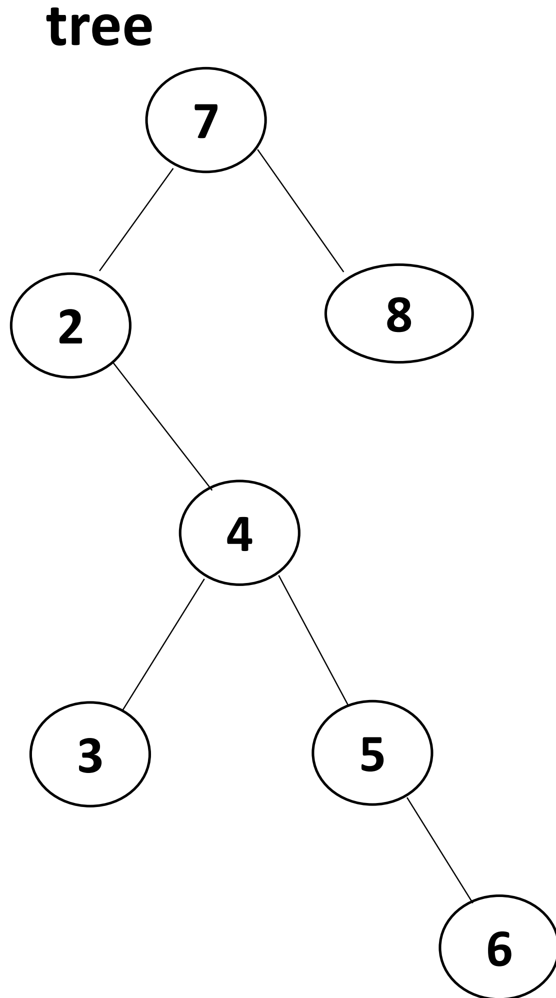


مرحله ۳: مقدار 9 را
بجای 10 می نویسیم و
سپس مقدار 9 را حذف
می کنیم.

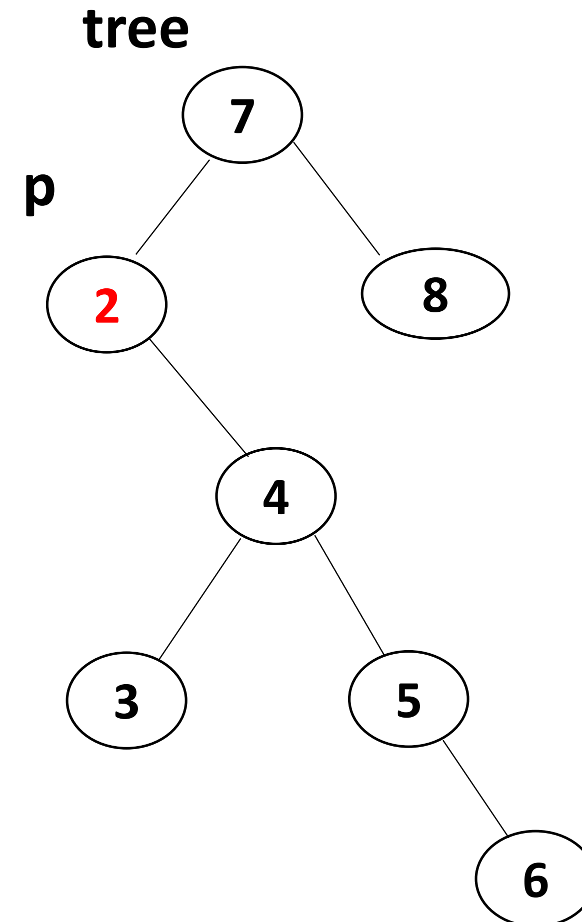


حالت ۳: X فقط فرزند راست دارد: برای این حالت، کافی است گره حاوی X را پیدا کنیم. سپس، کوچکترین بچه زیردرخت راست را پیدا کرده و مقدار آن را بجای X نوشته و آن بچه کوچکتر را که اگر برگ باشد به روش حالت ۱ حذف اش می کنیم در غیر اینصورت بسته به اینکه چه حالتی است به روش بازگشتی آن را حذف می کنیم.

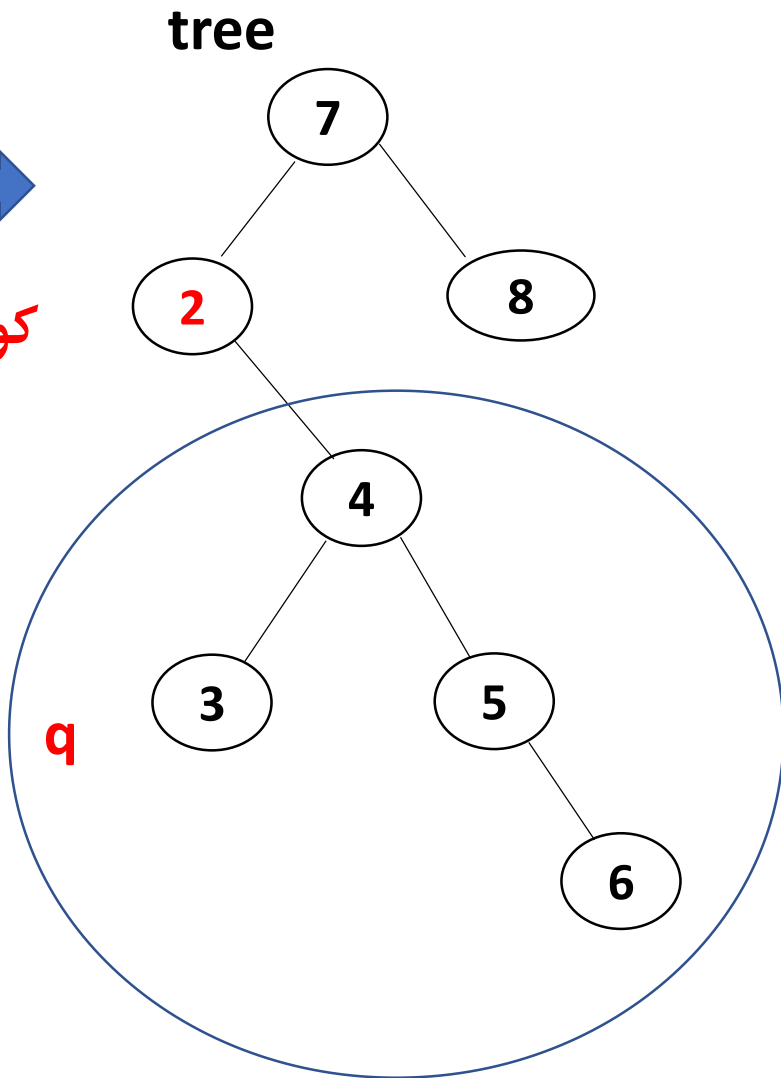
به عنوان مثال: در درخت زیر، گره با مقدار X=2 را حذف کنید.



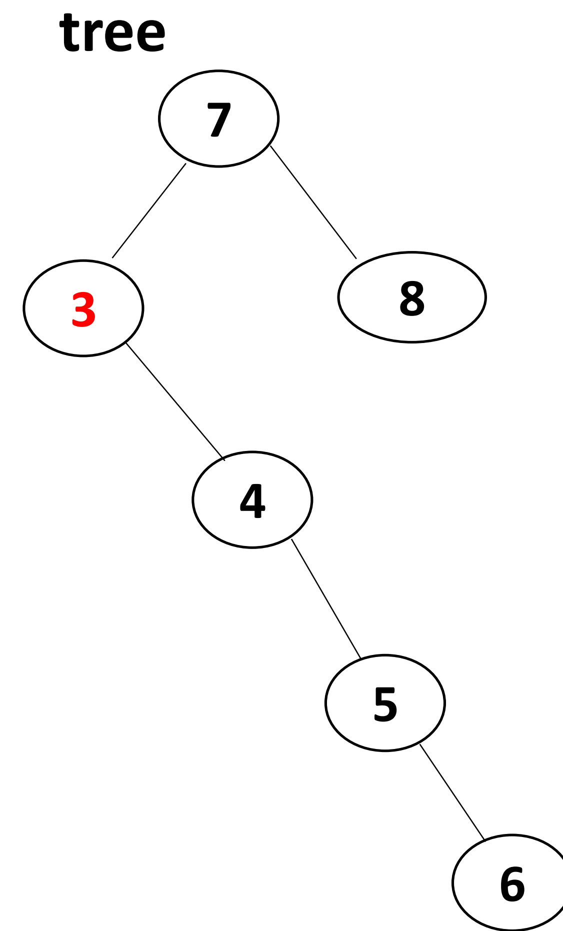
مرحله ۱: یافتن x



مرحله ۲: یافتن کوچکترین فرزند راست گره با مقدار x=2

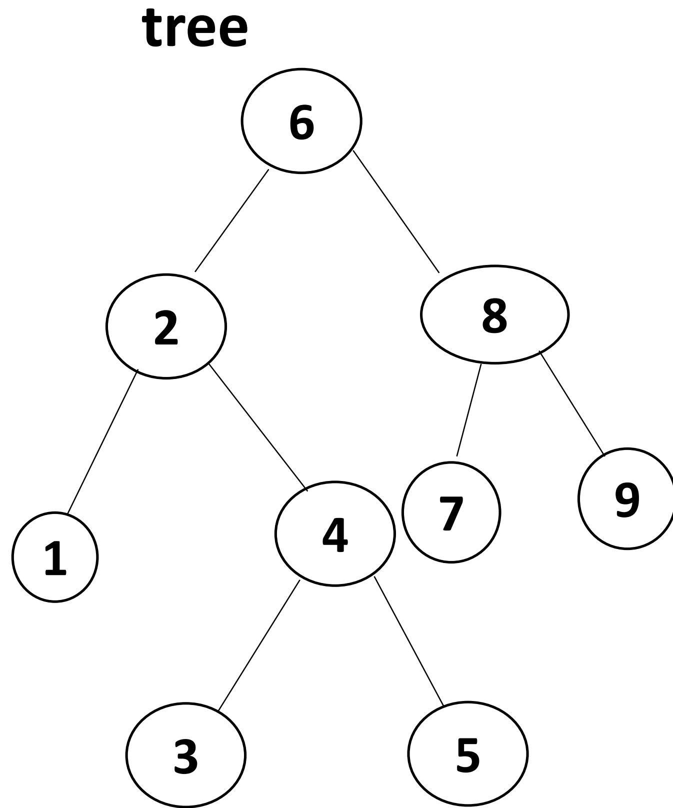



مرحله ۳: مقدار 3 را بجای 2 می نویسیم و سپس 3 را حذف می کنیم.

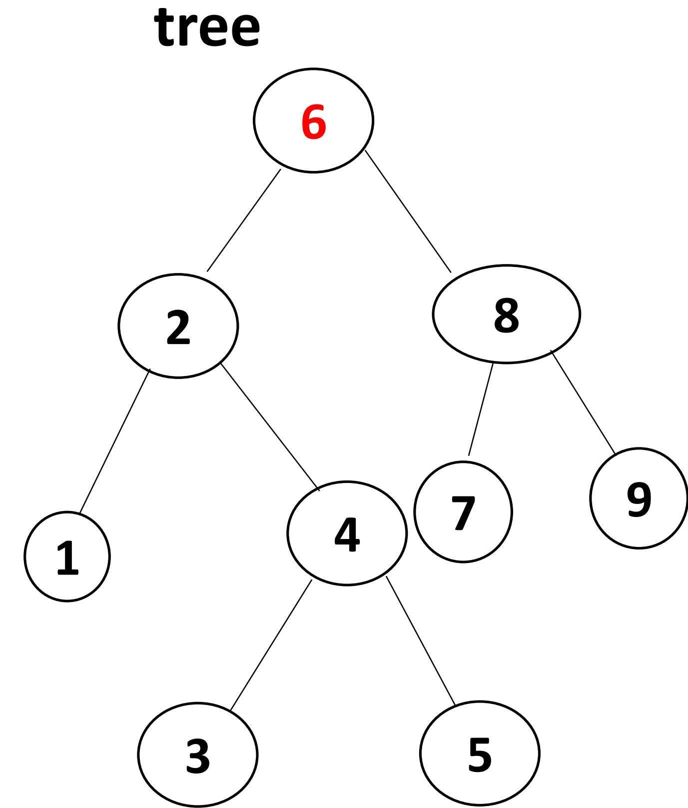


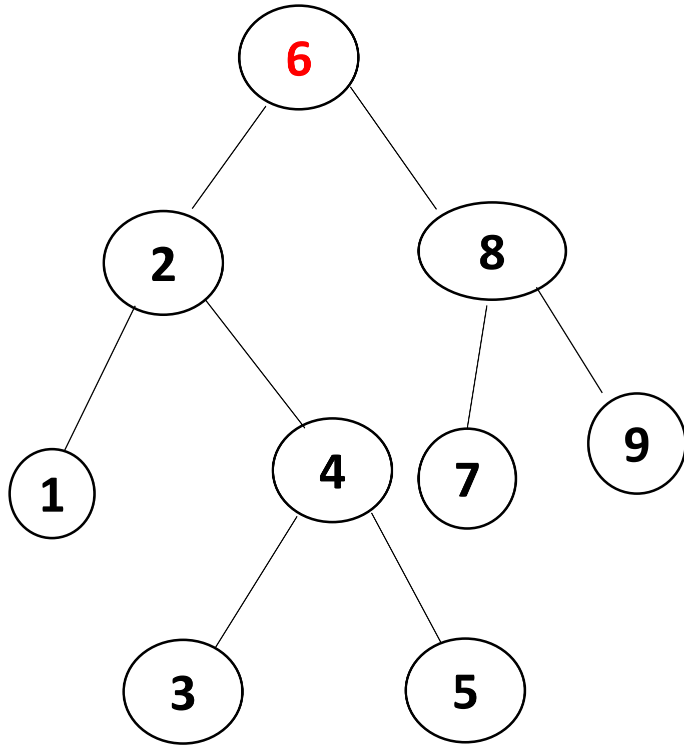
حالت ۴: $x$ دو فرزند دارد: مطابق حالت ۲ یا حالت ۳ عمل می کنیم.

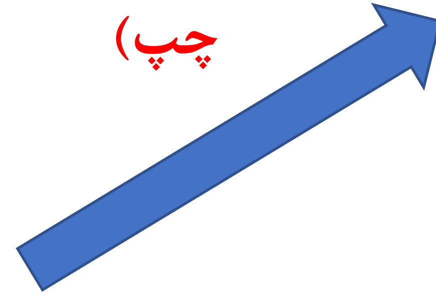
به عنوان مثال: در درخت زیر، گره با مقدار $x=6$ را حذف کنید.



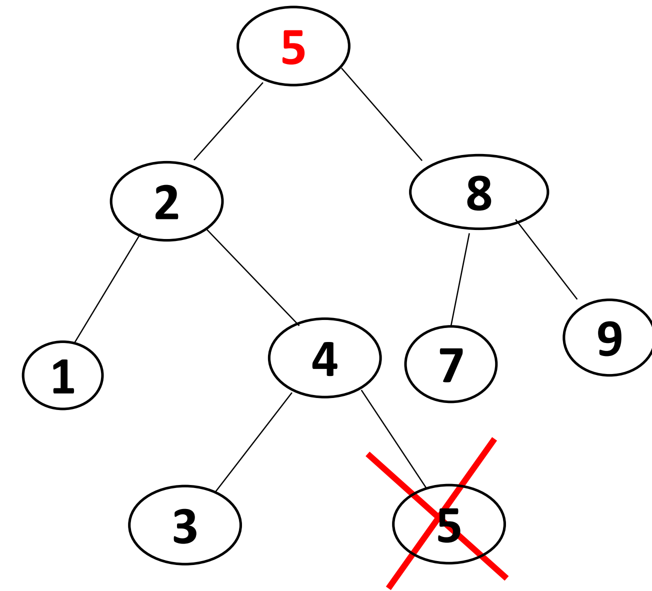
مرحله ۱: یافتن x

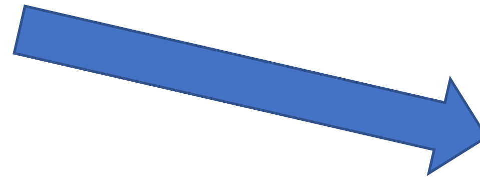




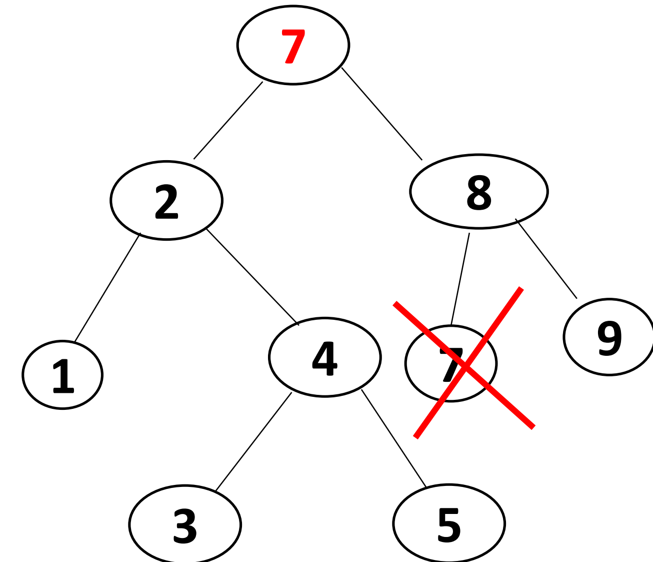



مطابق حالت ۲ عمل می کنیم (بزرگترین فرزند چپ)



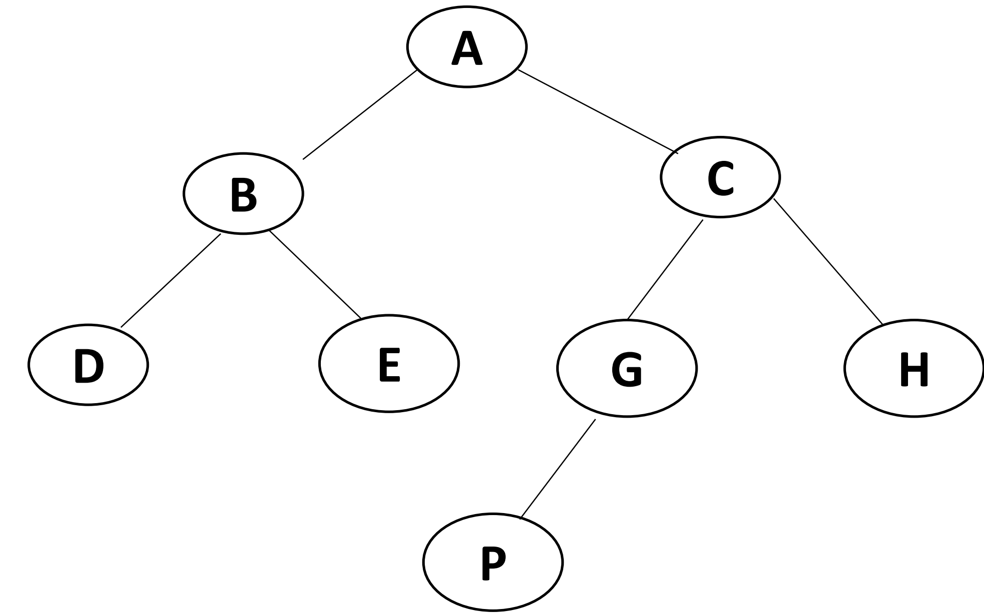
مطابق حالت ۳ عمل می کنیم (کوچکترین فرزند راست).

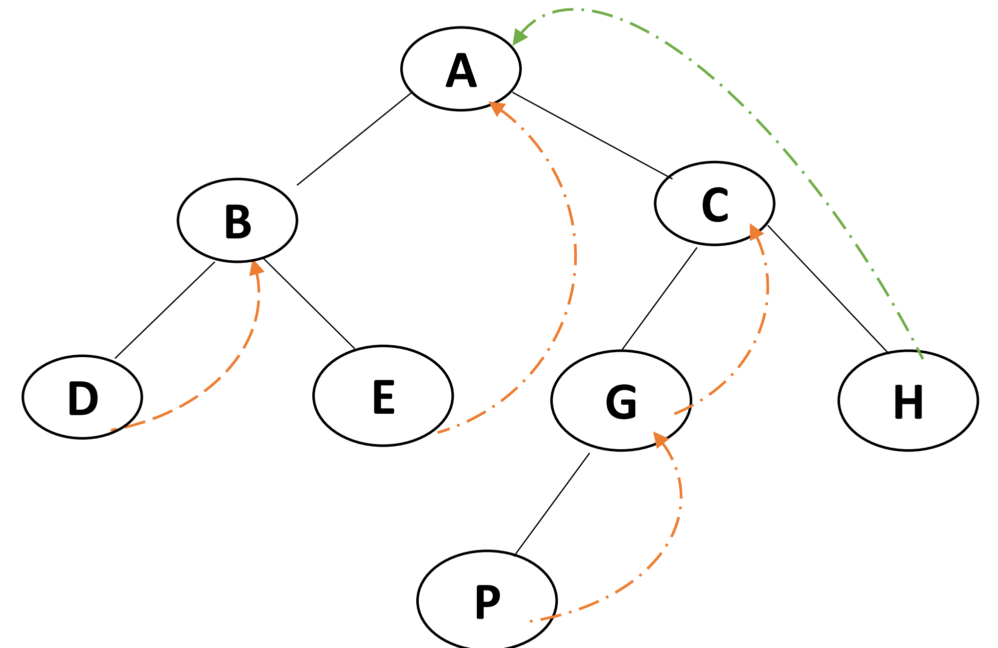


- زمان اجرای الگوریتم حذف به ارتفاع درخت بستگی دارد.
- در بدترین حالت، ارتفاع درخت از مرتبه $O(n)$ است بنابراین زمان اجرای الگوریتم حذف نیز از مرتبه $O(n)$ خواهد بود.
- در بهترین حالت یا حالت متوسط، ارتفاع درخت از مرتبه $\Theta(\log n)$ است بنابراین زمان اجرای الگوریتم حذف نیز از مرتبه $\Theta(\log n)$ خواهد بود.

# درخت دودویی نخی (Threaded Binary Tree:TBT)

- درختی است که در آن، یک نخ (اشاره گر)، هر گره را به گره بعدی در پیمایش inorder پیوند می دهد.
- برای تبدیل به درخت نخی، برای هر گره که اشاره گر راست آن برابر NULL باشد را به گره بعدی در پیمایش inroder وصل می کنیم.






- برای مشخص کردن اینکه، آیا اشاره گر راست، معمولی است یا نخ، یک فیلد منطقی rThread به گره ها اضافه می کنیم. مقدار true به معنای اشاره گر نخ و مقدار false به معنای اشاره گر معمولی است.

- فرض کنید می خواهیم پیغامی متشکل از تعدادی نماد را رمزگذاری کرده و بفرستیم و گیرنده نیز با توجه به جدول رمزگذاری، پیغام را از حالت رمز در بیاورد.

- مثال:

پیغام: ABACCDA

جدول رمز ۱

| نماد | A | B | C | D |
|---|---|---|---|---|
| رمز | 00 | 01 | 10 | 11 |

پیغام رمزگذاری شده : 00 01 00 10 10 11 00

۱۴ بیت

جدول رمز ۲

| نماد | A | B | C | D |
|---|---|---|---|---|
| رمز | 0 | 110 | 10 | 111 |

۱۳ بیت

پیغام رمزگذاری شده : 0 110 0 10 10 111 0

جدول رمز ۲ بهینه است.

- نکته مهم این است که رمزها نباید پیشوند هم باشند چون موقع رمزگشایی مشکل پیش می آید.

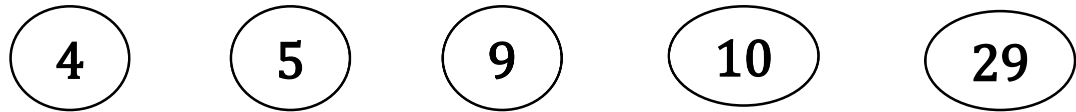
بعنوان مثال: اگر پیغام ABC را با کدهای A=0 و B=01 و C=1 در نظر بگیریم

ABC= 0011 → AACC or ABC ????

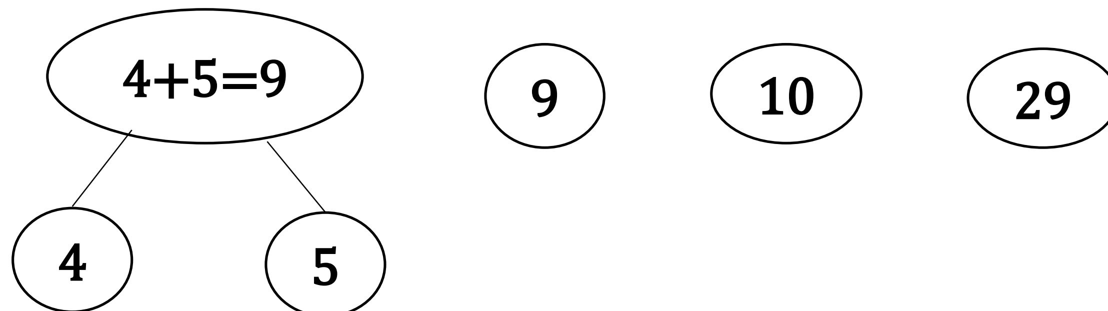
- الگوریتم رمزگذاری هافمن را با مثال توضیح می دهیم.
- مثال:

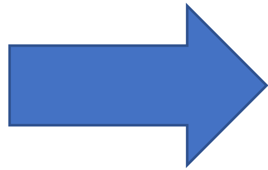
| نماد | N | I | S | E | T |
|---|---|---|---|---|---|
| وزن (تعداد تکرار در پیغام) | 9 | 5 | 4 | 29 | 10 |

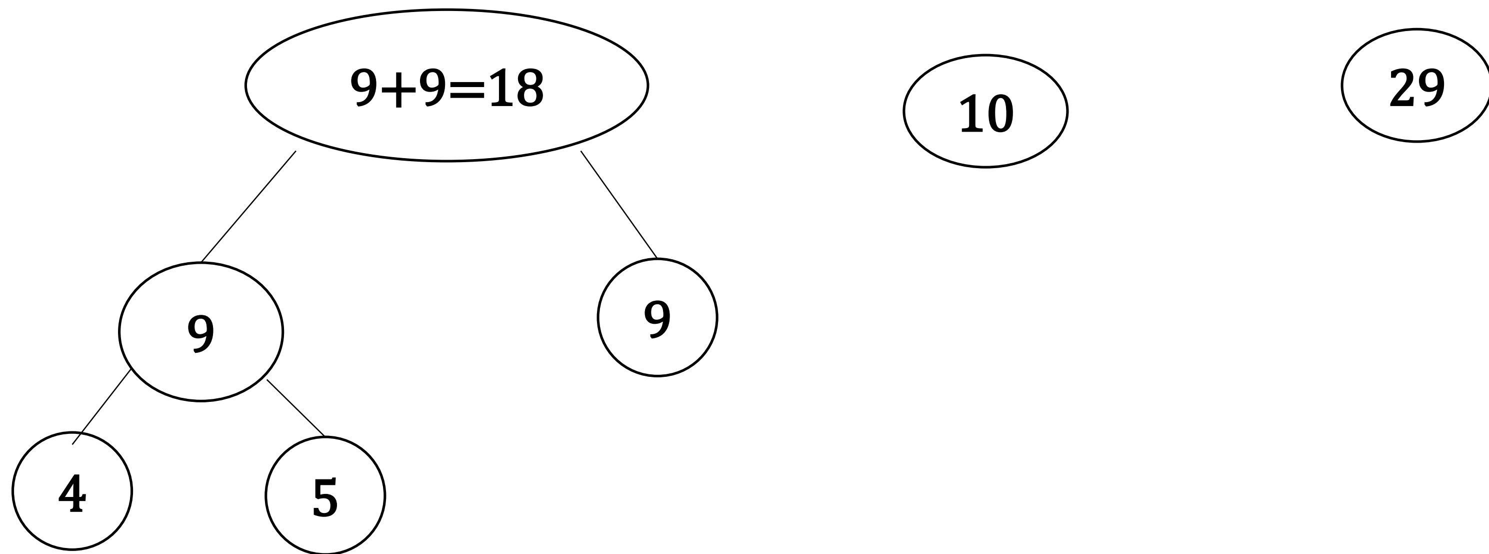
جواب: ابتدا وزن ها را بطور صعودی مرتب کرده و برای هر وزن یک درخت با یک گره در نظر می گیریم.

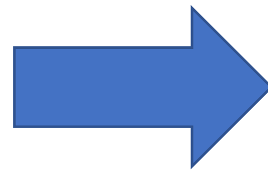


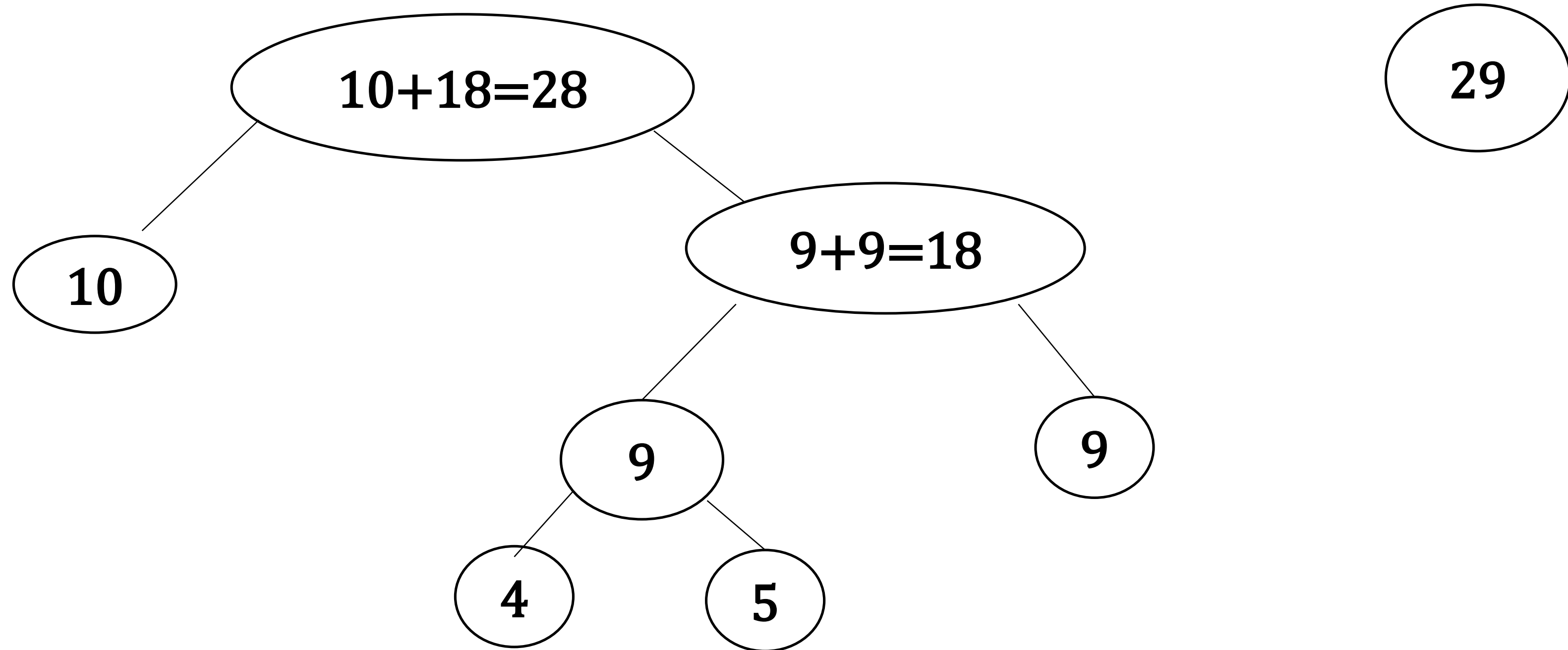

دو درخت با کمترین وزن را انتخاب کرده و و یک درخت جدید با آنها ساخته و مجموع وزن آنها را در ریشه درخت جدید قرار می دهیم و دو درخت قبلی را حذف می کنیم.

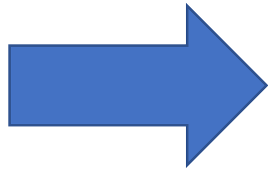


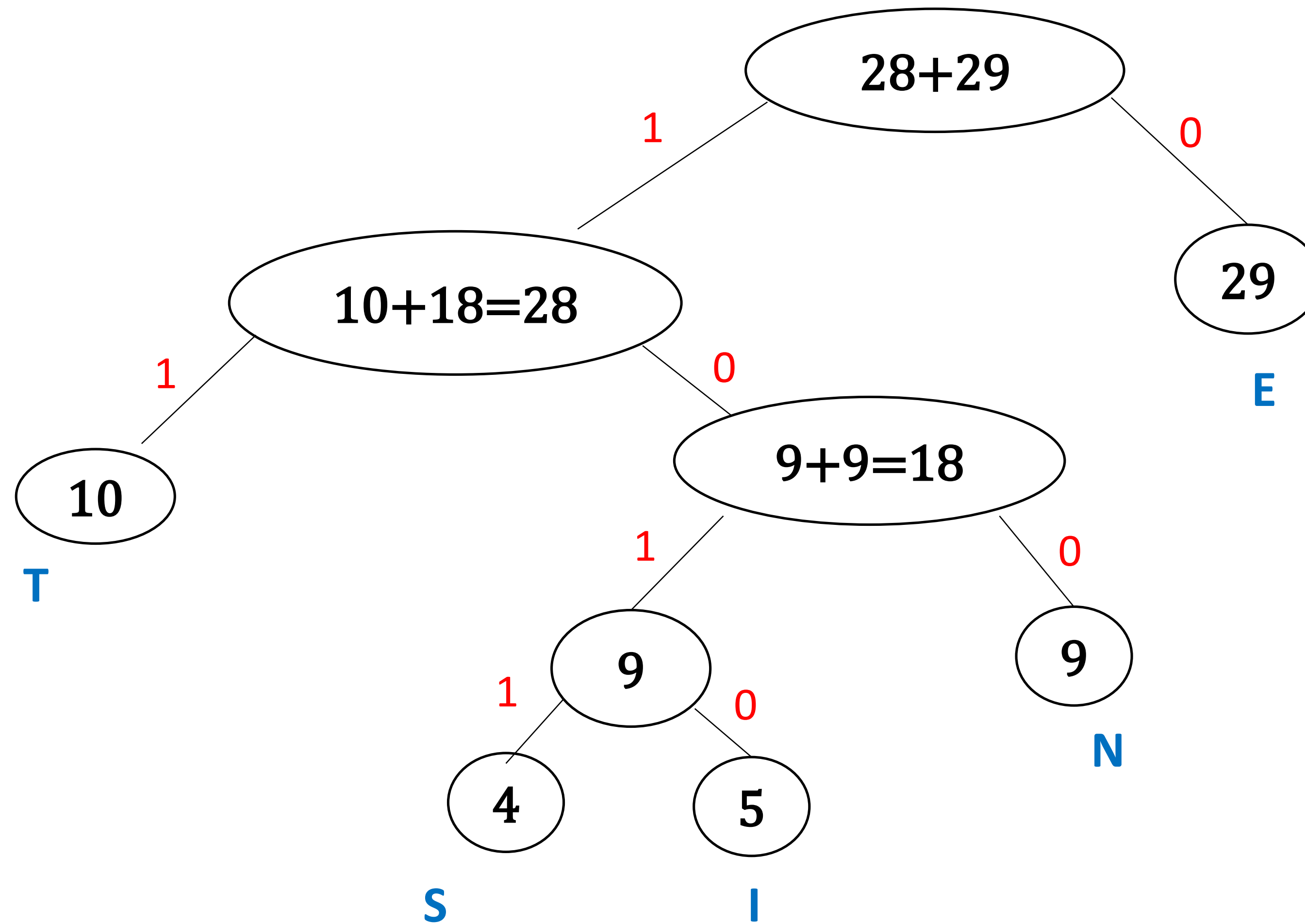



9+9=18
10
29
9
9
4
5




10+18=28
29
9+9=18
10
9
9
4
5






| T | E | S | I | N | نماد |
|---|---|---|---|---|---|
| 11 | 0 | 1011 | 1010 | 100 | رمز |

پیچیدگی الگوریتم: $\theta(n\log n)$

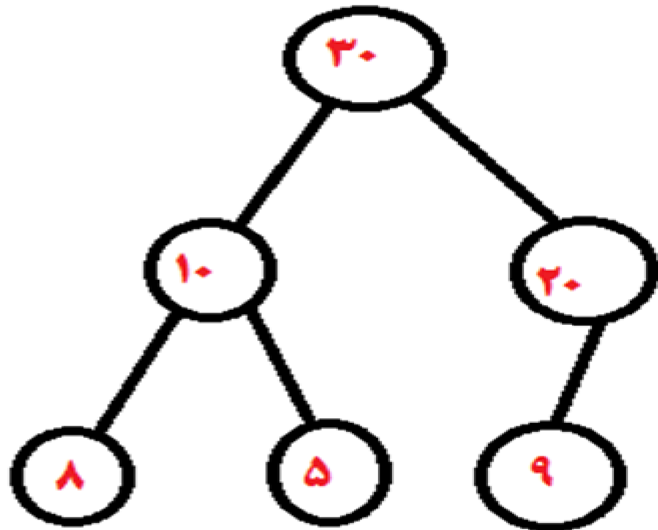
# هرم (Heap)

- هرم یک درخت دودویی با ویژگیهای زیر است:

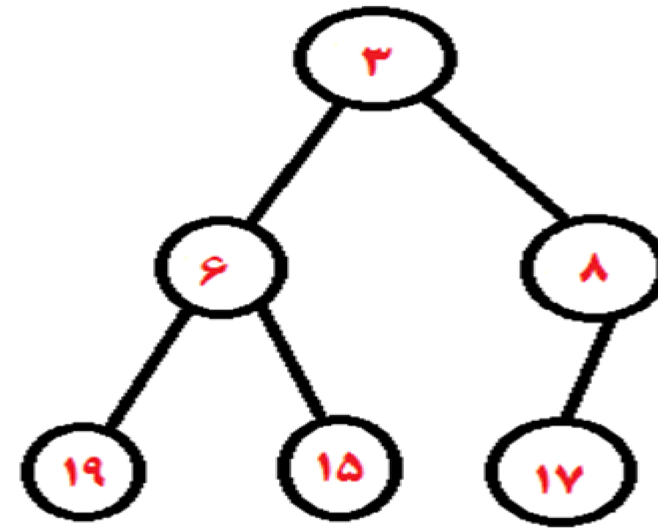
۱- کامل است.

۲-۱: مقدار هر گره، بزرگتر یا مساوی مقادیر گره های فرزندانش است ( هرم بیشینه: MaxHeap).

۲-۲: مقدار هر گره، کوچکتر یا مساوی مقادیر گره های فرزندانش است (هرم کمینه: MinHeap).




MaxHeap




MinHeap

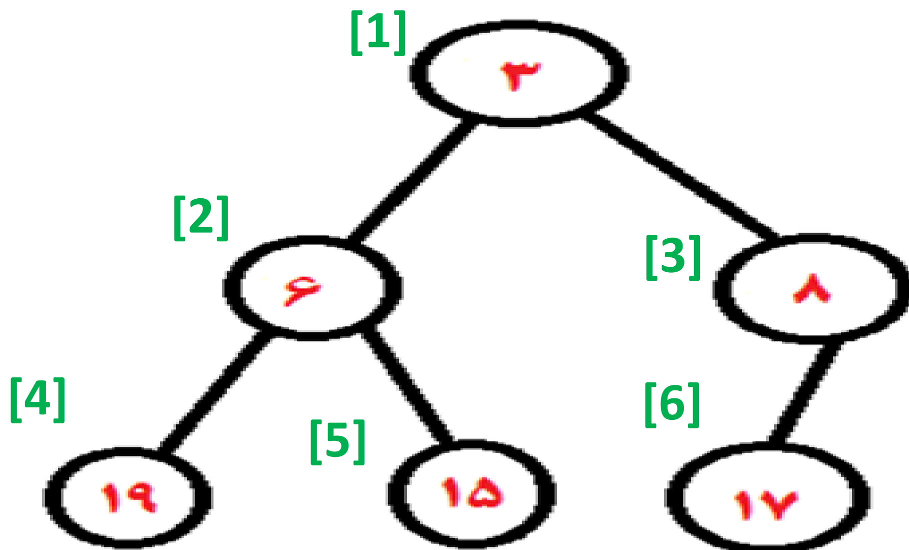
❑ هرم را با آرایه پیاده سازی می کنند.

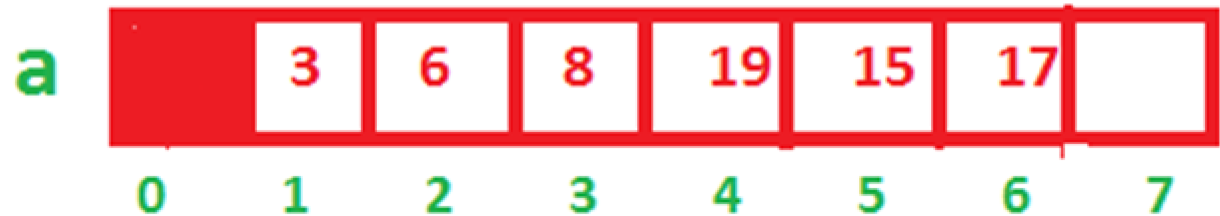
✓ a[0] خالی می ماند.

✓ ریشه در a[1] قرار می گیرد.

✓ برای هر گره در مکان k : فرزندان چپ و راست (در صورت وجود) به ترتیب در مکان های 2k و 2k+1 قرار می گیرند.

❑ بعنوان مثال:



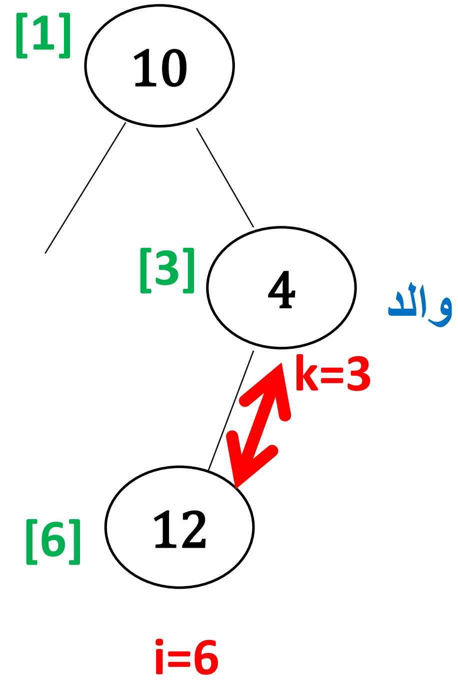



- یک هرم با n گره دارای ارتفاع $\theta(\log n)$ می باشد چون درخت کامل می باشد.
- پیدا کردن کوچکترین عنصر در یک MinHeap و یا بزرگترین عنصر در یک MaxHeap دارای پیچیدگی $\theta(1)$ خواهد بود.
- پیدا کردن بزرگترین عنصر در یک MinHeap و یا کوچکترین عنصر در یک MaxHeap دارای پیچیدگی $\theta(n)$ خواهد بود.

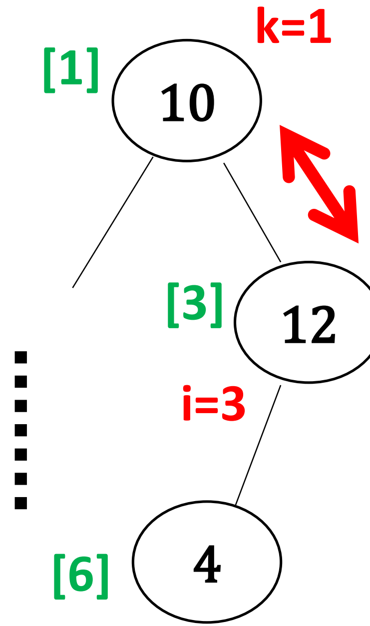
❑ توابع مورد نیاز که الگوریتم های آنها در اسلایدهای بعدی آمده اند (این توابع برای هرم بیشینه نوشته شده اند می توان آنها را با کمی تغییر برای هرم کمینه نیز استفاده کرد)

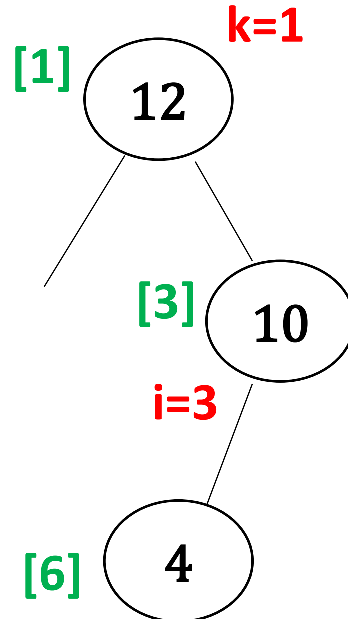
1. moveUp(a,i) : عنصر موجود در مکان i ام آرایه a (هرم) را طوری به سمت ریشه هرم حرکت می دهد که خاصیت بیشینه بودن هرم برقرار شود.
2. insert(a,i,x) : عنصر جدید x را به مکان i ام هرم اضافه کرده سپس تابع moveUp را فراخوانی می کند تا خاصیت بیشینه بودن هرم برقرار شود.
3. moveDown(a,i) : عنصر موجود در مکان i ام آرایه a (هرم) را طوری به سمت پایین هرم حرکت می دهد که خاصیت بیشینه بودن هرم برقرار شود.
4. delete(a) : ریشه را حذف کرده و بجای آن، آخرین عنصر آرایه را کپی کرده (در واقع جای ریشه و آخرین عنصر هرم را باهم جابجا می کند) و سپس تابع moveDown(a,1) را فراخوانی می کند تا خاصیت بیشینه بودن هرم برقرار شود.
5. makeHeap(a,n) : یک آرایه را به heap تبدیل می کند.

# تابع moveUp




هدف این است که عنصر موجود در مکان i=6 به سمت بالای هرم حرکت داده شود تا خاصیت بیشینه بودن هرم برقرار شود





# تابع moveUp

```
void moveUp(int a[], int i){
    int  k=i/2;
    while(k>=1 && a[k]<a[i])
    {
        swap(a[k],a[i]);
        i=k;
        k=i/2;
    }
}
```

**والدِ مکان شماره i را محاسبه کرده و در متغیر k قرار می دهد.**

**تا موقعی که والد (a[k]) از فرزند (a[i])  کوچکتر است و همچنین به ریشه نرسیده ایم جای والد و فرزند عوض شده و این روند ادامه می یابد.**

اگر  n تعداد گره ها باشد پیچیدگی این تابع  در بدترین حالت از مرتبه $O(\log n)$ خواهد بود.

اگر  n تعداد گره ها باشد پیچیدگی این تابع  در بهترین حالت از مرتبه $\Omega(1)$ خواهد بود.

# تابع insert

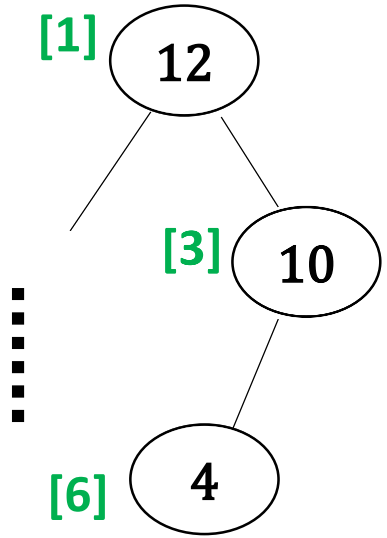
این تابع، عنصر جدید x را به مکان i ام آرایه (هرم) اضافه کرده و سپس تابع moveUp را فراخوانی می کند تا خاصیت بیشینه بودن هرم برقرار شود.

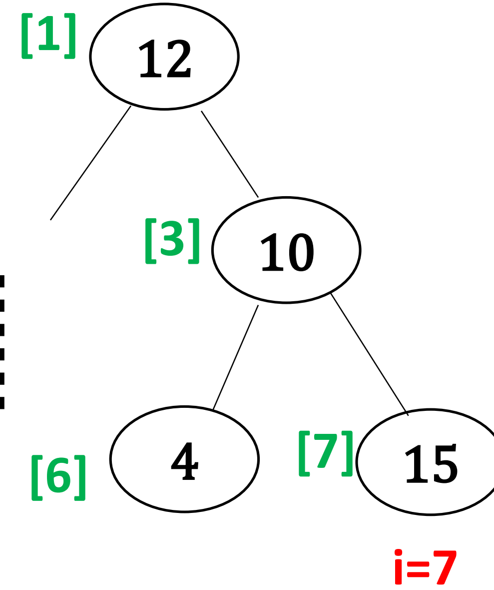
```
void insert(int a[], int i,int x){
    a[i]=x;
    moveUp(a,i);
}
```

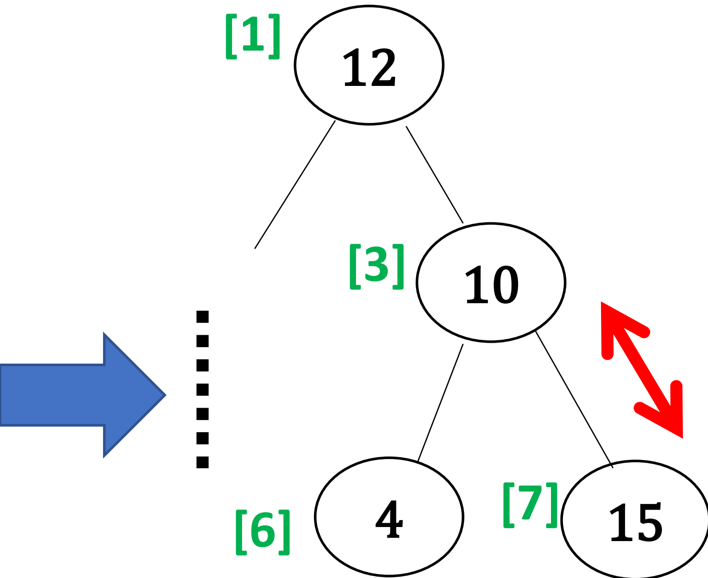
اگر n تعداد گره ها باشد پیچیدگی این تابع در بدترین حالت از مرتبه $O(\log n)$ خواهد بود.

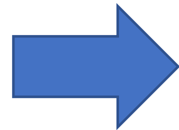
اگر n تعداد گره ها باشد پیچیدگی این تابع در بهترین حالت از مرتبه $\Omega(1)$ خواهد بود.

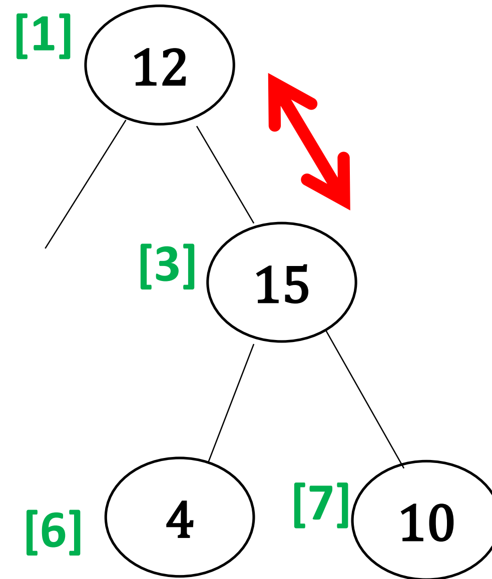
کار در کلاس: عنصر جدید x=15 را به مکان i=7 هرم زیر اضافه کنید.

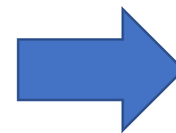


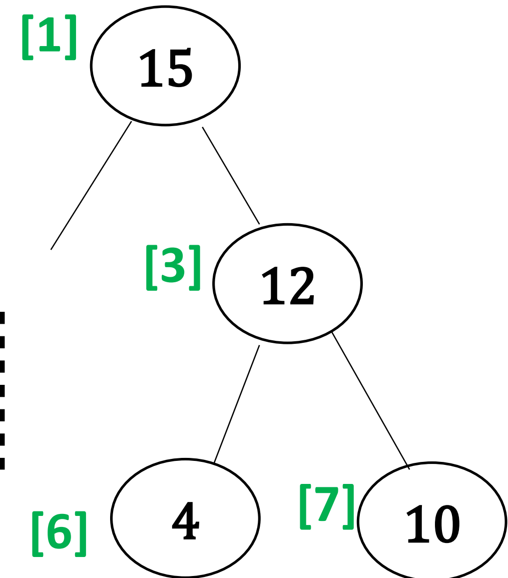


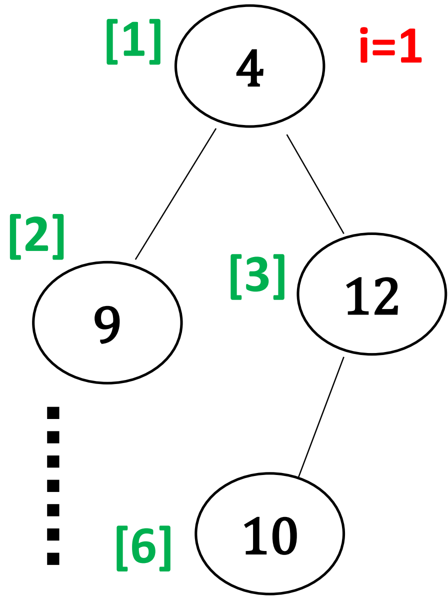





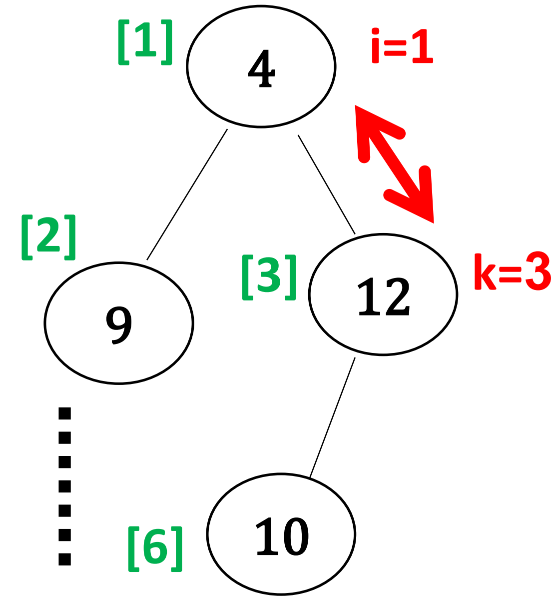



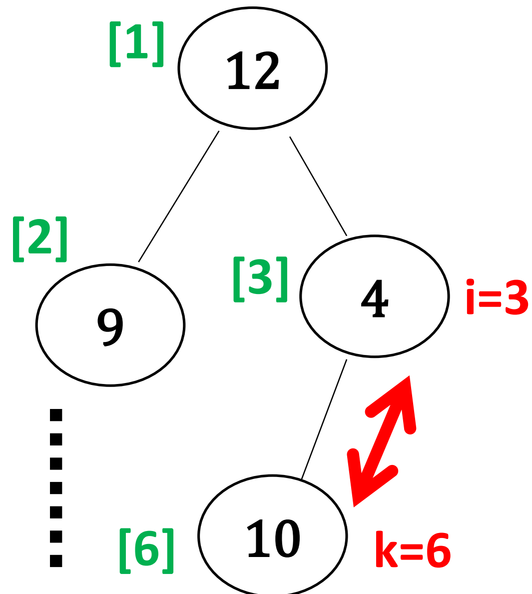


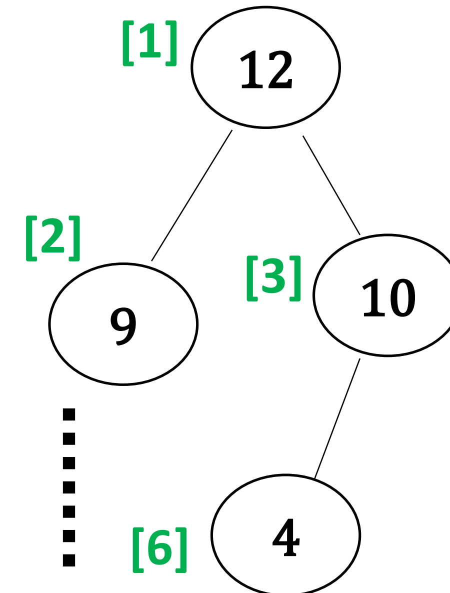
# تابع moveDown




هدف این است که عنصر موجود در مکان i=1 به سمت پایین هرم حرکت داده شود تا خاصیت بیشینه بودن هرم برقرار شود.
عنصر ۴ با فرزندانش مقایسه شده و با بزرگترین آنها جابجا می شود.



# تابع moveDown

در الگوریتم زیر، n تعداد عناصر هرم را نگهداری می کند.

```
void moveDown(int a[], int n, int i ){
    while(2*i<=n)
    {
        int k=2*i;
        if(k<n && a[k]<a[k+1])  k++;
        if(a[i]>a[k])    break;
        swap(a[i],a[k]);
        i=k;
    }
}
```

**حلقه تا جایی تکرار می شود که گره مکان i ام فرزند چپ داشته باشد چون اگر فرزند چپ نداشته باشد مطمئنا فرزند راست هم ندارد چون درخت کامل است.**

**فرزند چپ گره i محاسبه شده و در متغیر k قرار داده می شود.**

**اگر فرزند چپ از فرزند راست کوچکتر باشد k یک واحد افزایش می یابد تا فرزند راست را نشان دهد.**

**اگر فرزند (a[k]) از والد (a[i]) کوچکتر بود از حلقه خارج می شود.**

**جای والد با فرزند بزرگتر جابجا می شود.**

اگر n تعداد گره ها باشد پیچیدگی این تابع در بدترین حالت از مرتبه $O(\log n)$ خواهد بود.

اگر n تعداد گره ها باشد پیچیدگی این تابع در بهترین حالت از مرتبه $\Omega(1)$ خواهد بود.