ساختمان داده ها
Data Structure

فصل پنجم
درخت tree

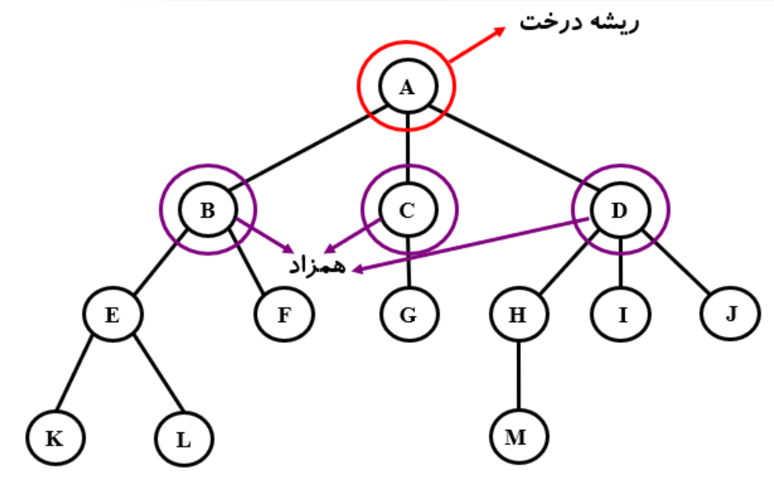
تهیه و تنظیم : محمد نعیمی

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 هرکدام نیز یک درخت می باشند و در درخت دور وجود ندارد.

گره : به عنصر حاوی اطلاعات و لینک به دیگر عناصراطلاق 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 گره‌ها گره های غیرپایانی نامیده میشوند.
همزاد(sibling): فرزندانی 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 اصطلاحا فرزند چپ و فرزند راست مینامیم.

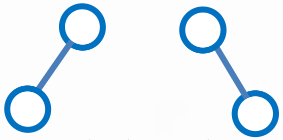


درجه A ، 3 می باشد
عمق M ، 4 می باشد
درجه B ، 2 می باشد
عمق E، 3 می باشد

ارتفاع درخت 4 می باشد
درجه درخت 3 می باشد
D پدر H، I و J 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)

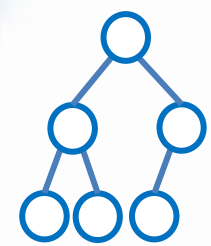
در درخت دودویی فرزند چپ با فرزند راست متفاوت است. این دو درخت متفاوت هستند



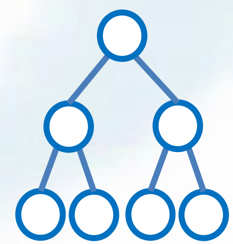
درخت پر(perfect): درختی با عمق d که $2^d + 1$ گره داشته باشد را درخت پر میگوییم. (یعنی در تمام مکانهای ممکن تا عمق d گره وجود داشته باشد. به عبارت دیگر درجه تمام گره ها 2 یا 0 باشد و تمام برگ ها در یک سطح باشند)

درخت کامل(complete): درختی با عمق d کامل است اگر تا عمق d-1 پر باشد و در عمق d بین گره ها از چپ به راست جای خالی وجود نداشته باشد(تمام برگها در چپ ترین مکان ممکن باشند). **هر درختی پری کامل نیز است.**

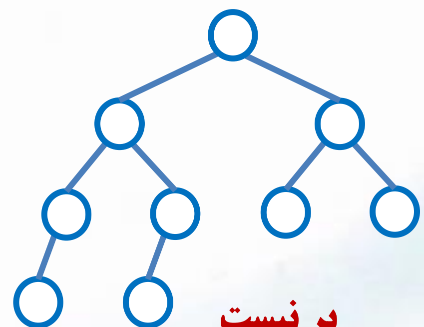
درخت مورب: درختی که تمام گره های فرزند دارش یا همگی فرزند چپ داشته باشند یا فرزند راست



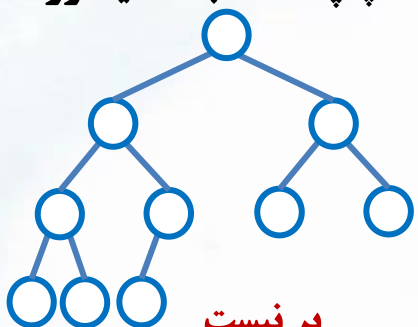
پر نیست
کامل است



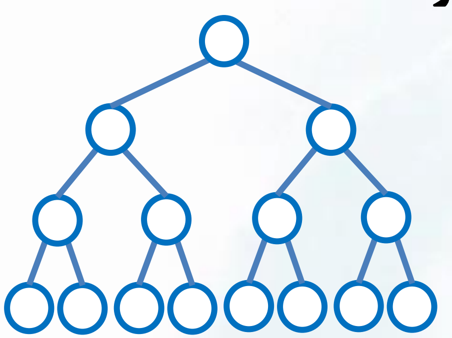
پر است
کامل است



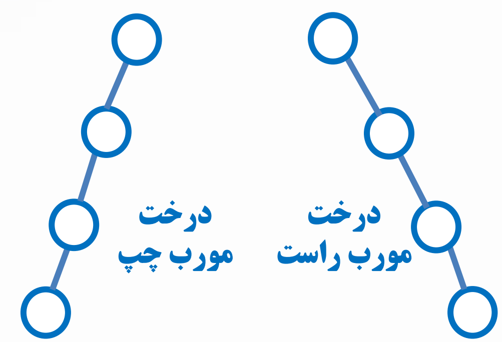
پر نیست
کامل نیست

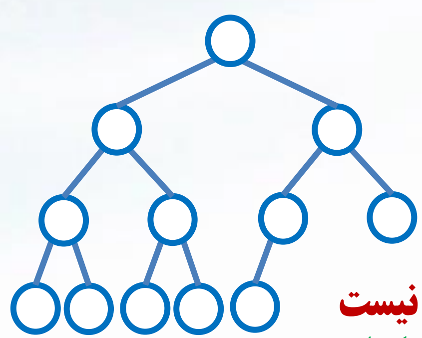


پر نیست
کامل است

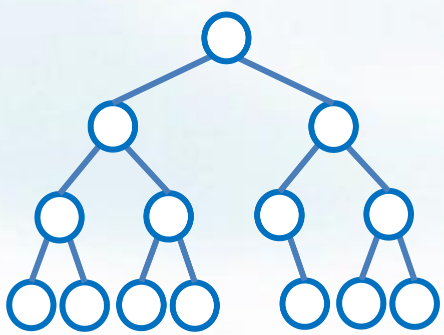


پر است
کامل است

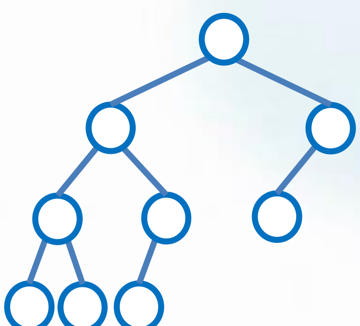




پر نیست
کامل است



پر نیست
کامل نیست



پر نیست
کامل نیست

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:

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

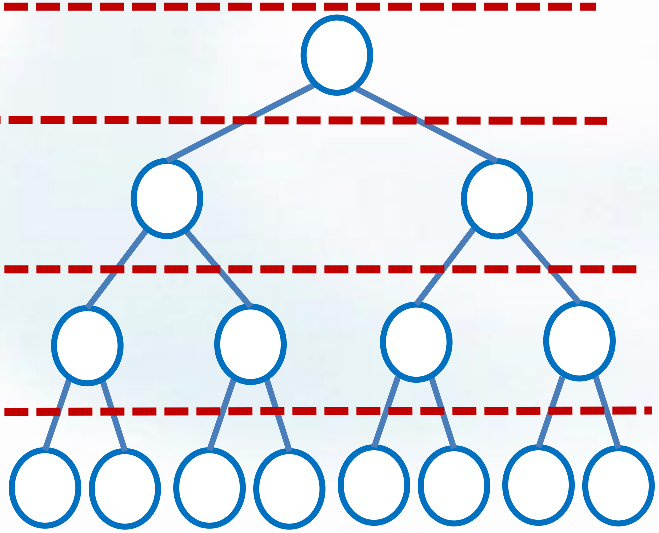
تعداد گره ها میشود تعداد کل فرزندان
بعلاوه 1 (ریشه) که فرزند کسی نیست

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

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

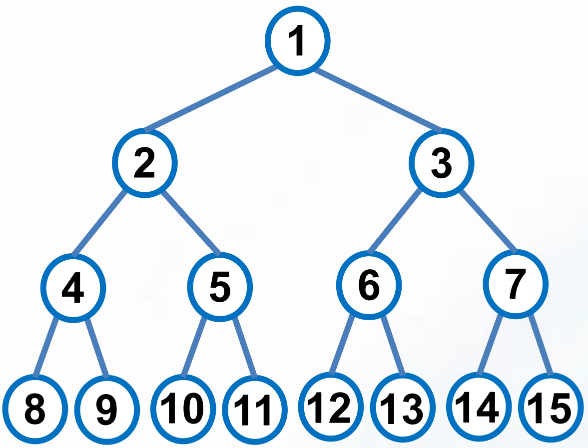
عمق درخت کامل با n گره می شود $[\log_2 n] + 1$

سطح	تعداد گره
1	1
2	2
3	4
4	8



تعداد کل گره ها $2^4 - 1 = 15$

نحوه ذخیره سازی درخت دودویی در برنامه (۱)

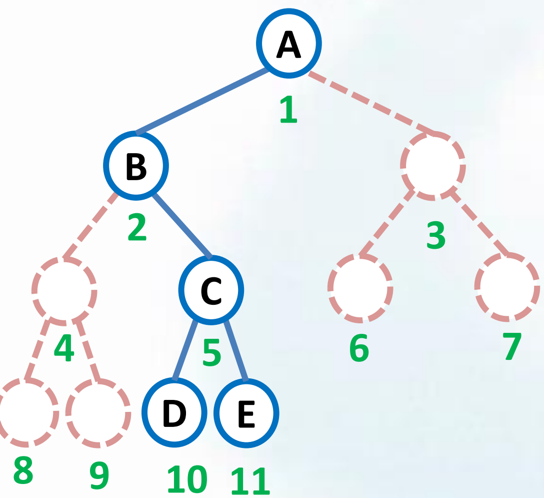


درخت با استفاده از آرایه:

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، اخرین 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

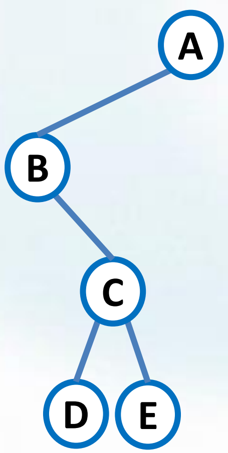
```
struct nodeT {
        type item;
        nodeT *left,*right;
    }
```

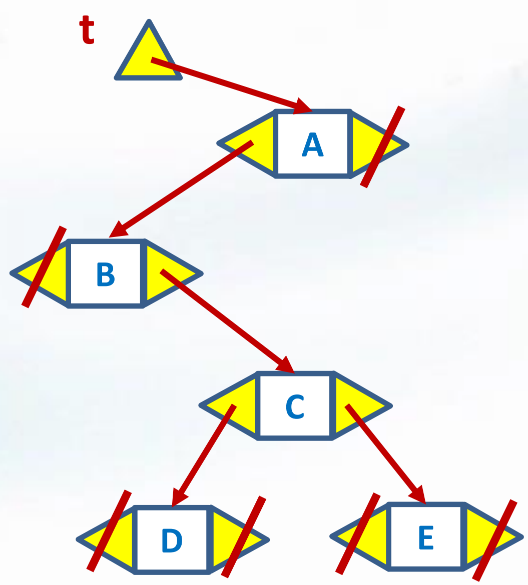
تعریف رکورد گره های درخت

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.

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





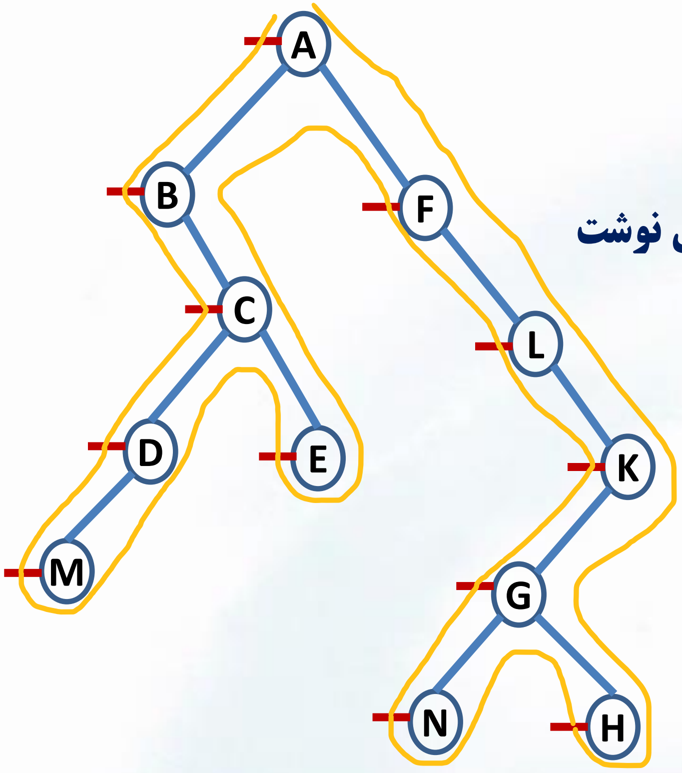
پیمایش درخت

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 دیگردر 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 (شبیه 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);
          }
  }
```

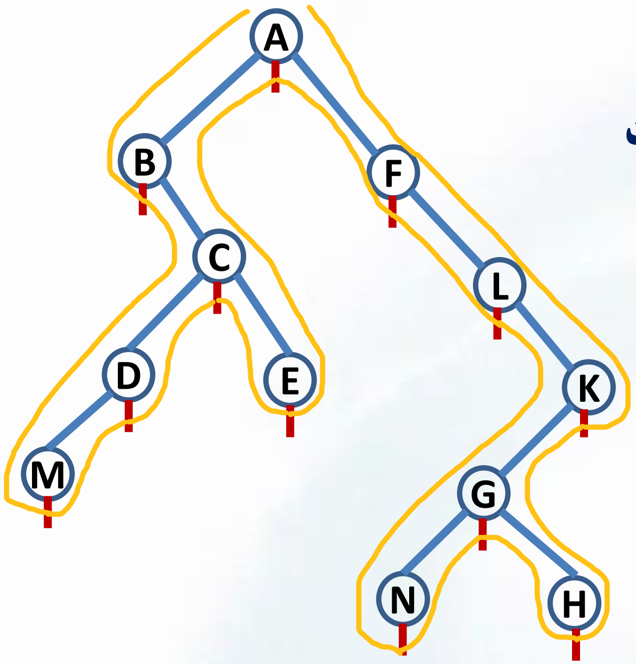
روش سریع:
کافی است در سمت چپ گره ها یک خط بکذاریم
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت چپ گره رسیدیم آن را چاپ میکنیم.

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

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



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

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

مدل غیر بازگشتی

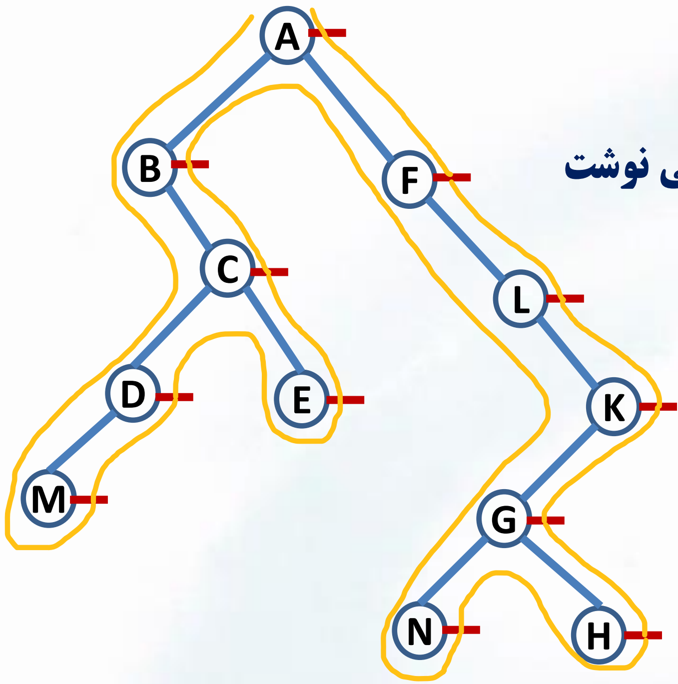
```
void inorder (nodeT *t)
  {
    nodeT *p=t;
    while(p!=NULL || !empty_stack(S))
     {
        while(p!=NULL)
         {
           push(S,p);
           p=p->left;
         }
        if (!empty_stack(S))
         {
           p=pop(S);
           cout<<p->item;
           p=p->right
         }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

روش سریع:
کافی است در پایین گره ها یک خط بکذاریم
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

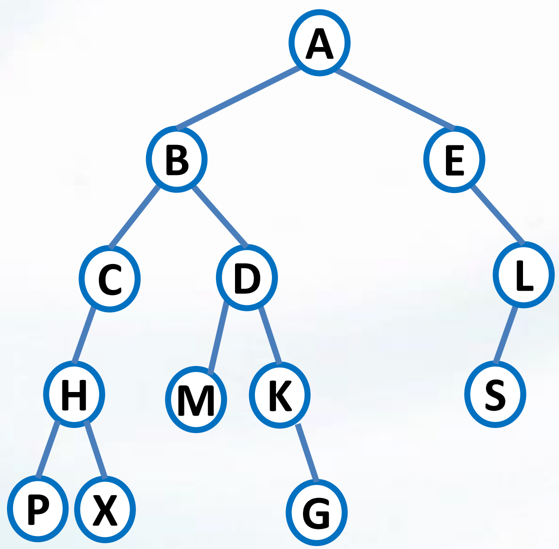
روش سریع:
کافی است در سمت راست گره ها یک خط بکذاریم
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت راست گره رسیدیم آن را چاپ میکنیم.

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

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



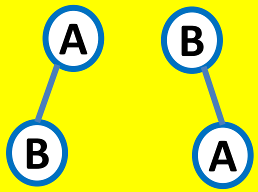

```
void bfs (nodeT *t)
  {
      nodeT *p;
      if (t!=NULL)
         addq(Q,t);
      while(!empty_queue(Q))
      {
         p=delq(Q);
         cout<<p->item;
         if (p->left!=NULL)
            addq(Q,p->left);
         if(p->right!=NULL)
            addq(Q,p->left);
      }
  }
```

BFS: ABECDLHMKSPXG

ساخت درخت بر اساس پیمایش های آن

این دو درخت متفاوت پیمایش pre و post یکسان دارند:

pre:AB post:BA

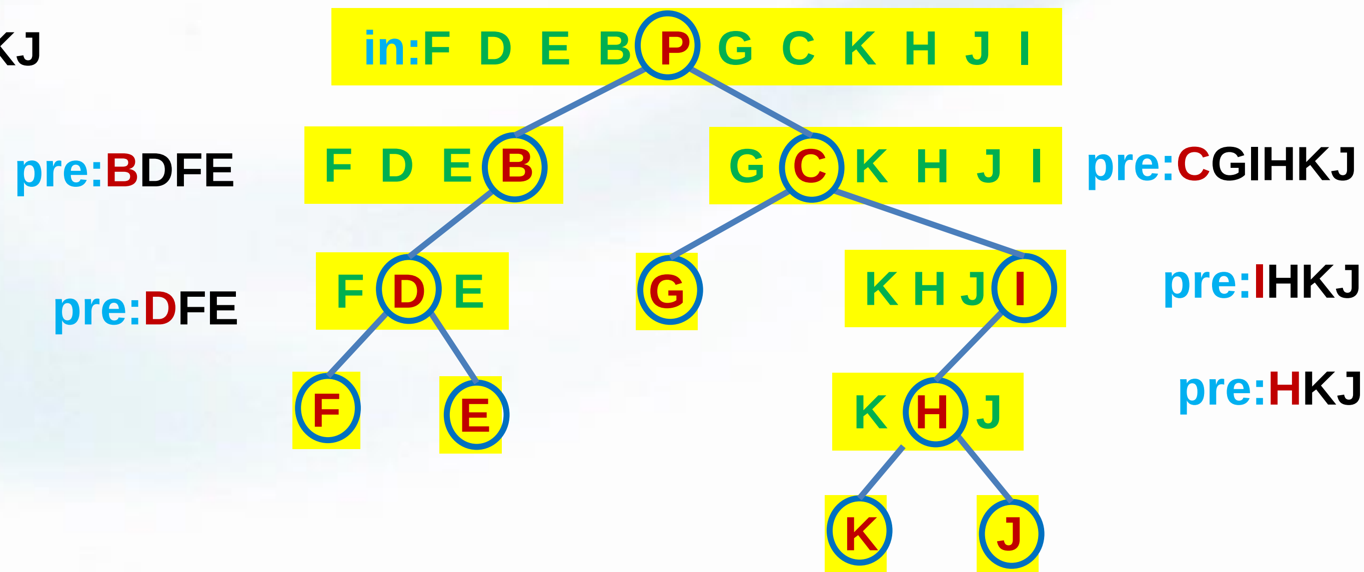


با داشتن پیمایش inorder و preorder می توان یک درخت منحصر به فرد ساخت.
با داشتن پیمایش inorder و postorder می توان یک درخت منحصر به فرد ساخت.
با داشتن پیمایش preorder و postorder نمی توان یک درخت منحصر به فرد ساخت.

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 میدهیم
عملیات فوق را برای تمام قسمت های چپ و راست تا رسیدن به تک عنصر ادامه میدهیم.
در حقیقت ما روی in کار میکنیم و از روی قسمتهای pre ریشه هر قسمت را پیدا میکنیم.

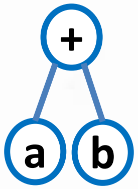
- در صورت داشتن post بجای pre ، آخرین عنصر پیمایش post "ریشه" می باشد.

in:FDEBPGCKHJI
pre:PBDFECGIHKJ

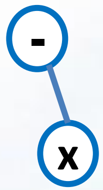


نمایش عبارت محاسباتی با درخت باینری(دودویی)

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



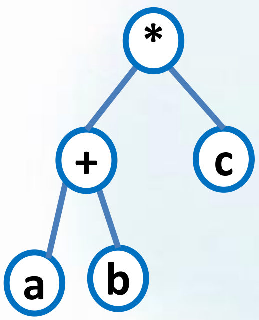
a+b



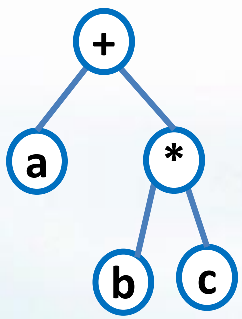
-x

عملگر هایی که در سطح بالاتری قرار دارند دیرتر اعمال می شوند.
برگ ها عملوند و گره های غیر برگ عملگر ها هستند.
preorder، inorder و postorder درخت به ترتیب معادل prefixed، infixed و postfixed عبارت خواهد بود

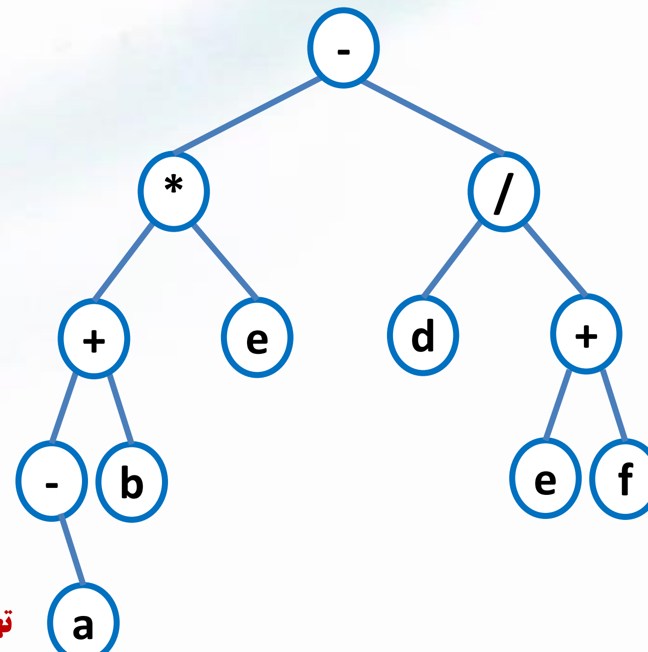
(a+b)*c



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؟

```
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);
  }
```

تابع بازگشتی که ریشه درخت را گرفته از آن درخت کپی گرفته، آدرس 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);
  }
```

توابع بازگشتی برای درخت دودویی(۲)

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 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 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));
  }
```

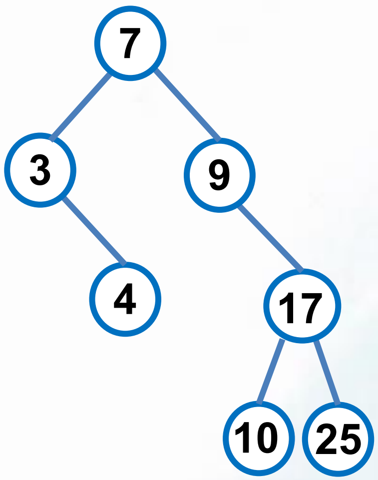
تابع بازگشتی محاسبه تعداد گره های دو فرزندی یک درخت

```
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));
  }
```

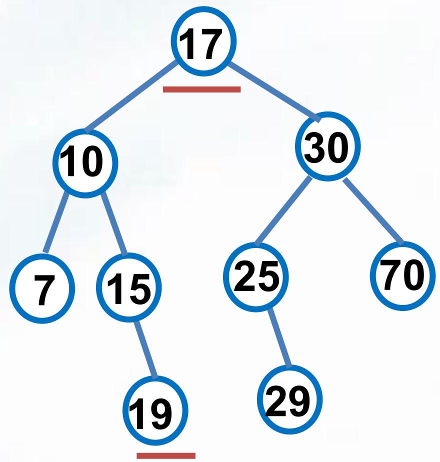
درخت های ویژه– درخت جستجوی دودویی (BST)

درخت جستجوی دودویی درختی است که در آن تمام گره های سمت چپ هر گره از آن گره کوچکتر و تمام گره های سمت راست گره از آن گره بزرگتر باشند.

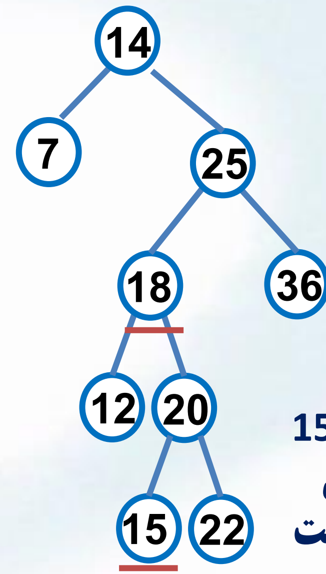
نکته: در درخت جستجوی دودویی عنصر تکراری مجاز نیست.



BST است



BST نیست چون 19>17 اما در فرزندان چپ 17 است



BST نیست 15<18 اما در فرزندان راست گره 18 است

با n گره عمق درخت BST حداقل $\log_2 n$ خواهد بود.

با n گره عمق درخت BST حداکثر n خواهد بود

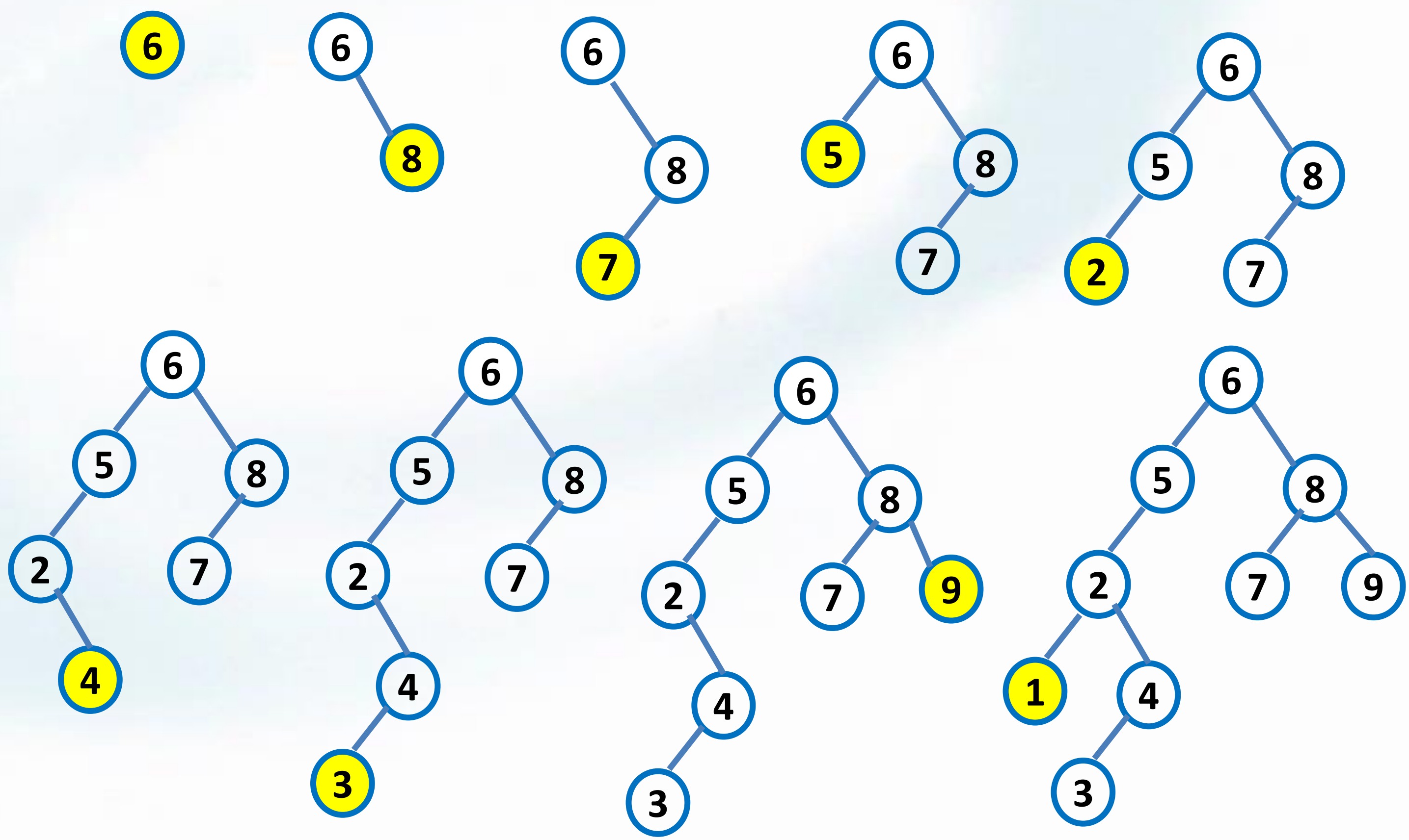
جستجو در درخت BST دارای پیچیدگی به اندازه عمق درخت می باشد.

پیمایش inorder درخت BST باعث چاپ عناصر درخت به صورت مرتب صعودی میشود.

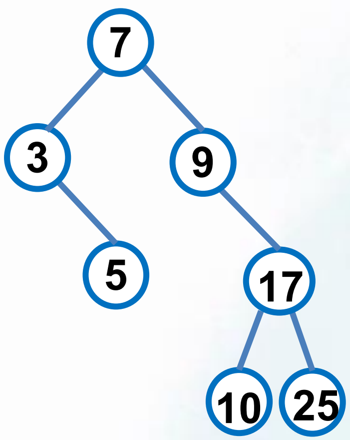
درخت جستجوی دودویی (BST) – افزودن عنصر

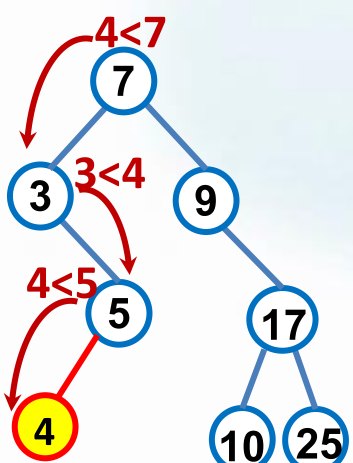
اگر درخت خالی بود عنصر وارد شده ریشه درخت خواهد بود.
اگر درخت خالی نبود، در صورتی که عنصر وارد شده از ریشه کمتر بود به سمت چپ و اگر بزرگتر بود سمت راست میرویم (عنصر مساوی وارد نمی شود و حذف می شود) و همین روند را آنقدر ادامه میدهیم که به جای خالی برسیم. عنصر جدید را در جای خالی اضافه میکنیم

افزودن اعداد مقابل به ترتیب از چپ به راست به درخت BST خالی **6-8-7-5-2-4-3-9-1**



افزودن 4 به این درخت





درخت جستجوی دودویی (BST) – تابع افزودن عنصر

پیچیدگی این تابع به اندازه عمق درخت است
دربدترین حالت o(n) و
بهترین حالت و متوسط o(logn)

`void  add_BST (nodeT * &t, type x)`	
`{`	
`nodeT *q, *p, *b;`	
`q=t;`	
`while (q!=NULL)`	تا وقتی به جای خالی نرسیدی
`{`	
`if (x.key==q->item.key)`	اگر تکراری بود
`{ cout<< “Repeated number”  ;  return();  }`	پیغام و پایان تابع
`b=q;`	b پدر q خواهد بود لذا جای q قرار میگیرد و q پایین می رود
`if (x.key<q->item.key)`	اگر کوچکتر بود
`q=q->left;`	برو سمت فرزند چپ
`else`	وگرنه (بزرگتر بود)
`q=q->right;`	برو سمت فرزند راست
`}`	
`p=new();`	ساخت فیزیکی گره
`p->item=x;`	مقدار دهی به گره
`p->right=p->left=NULL;`	گره برگ است و فرزند راست و چپ ندارد
`if(t==NULL)`	اگر درخت خالی بود
`t=p;`	گره می شود ریشه
`else`	وگرنه
`if(x.key<b->item.key)`	اگر کوچکتر از b بود (q وقتی به جای خالی میرسد b پدرش است)
`b->left=p;`	فرزند چپ b می شود
`else`	وگرنه
`b->right=p;`	فرزند راست b می شود.
`}`	

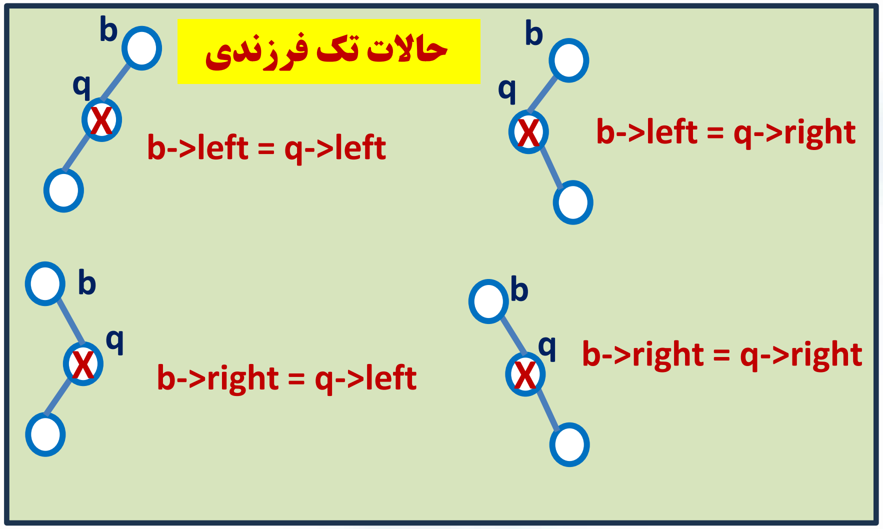
درخت جستجوی دودویی (BST)– تابع جستجو عنصر

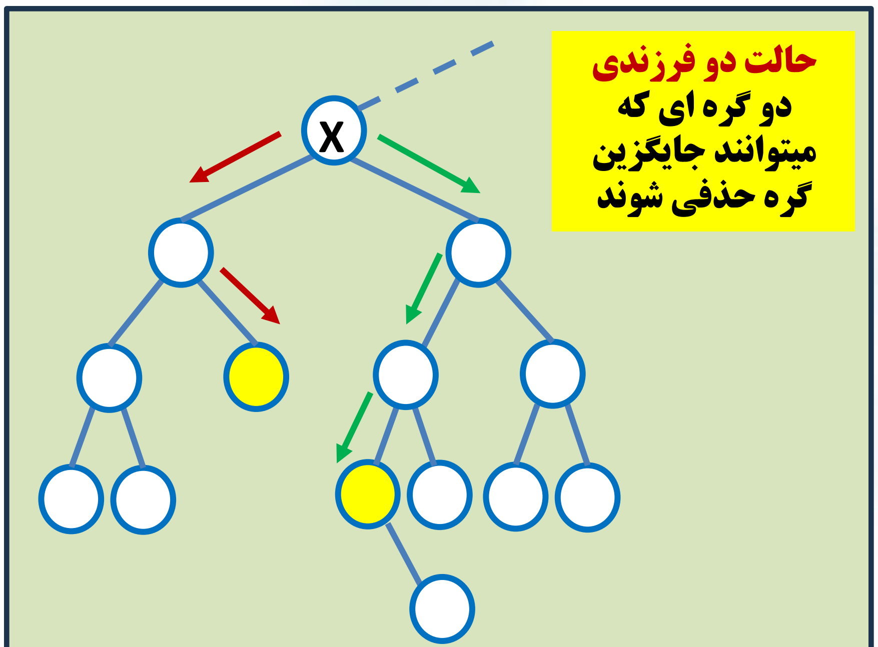
```
nodeT*  search_BST (nodeT * t, int x)
   {
     nodeT *q;
     q=t;
     while (q!=NULL)
       {
         if (x.key==q->item.key)
              return(q);
         if (x.key<q->item.key)
              q=q->left;
         else
              q=q->right;
       }
     return(NULL);
   }
```

```
nodeT*  search_BST_rec (nodeT * t, int x)
   {
     if (t==NULL)
        return(NULL);
     if (x.key==t->item.key)
         return(t);
      if (x.key<q->item.key)
          return( search_BST_rec (q->left,x));
      return( search_BST_rec (q->right,x));
   }
```

پیچیدگی این دو تابع به اندازه عمق درخت است
دربدترین حالت o(n) و
بهترین حالت و متوسط o(logn)

درخت جستجوی دودویی (BST) – حذف عنصر





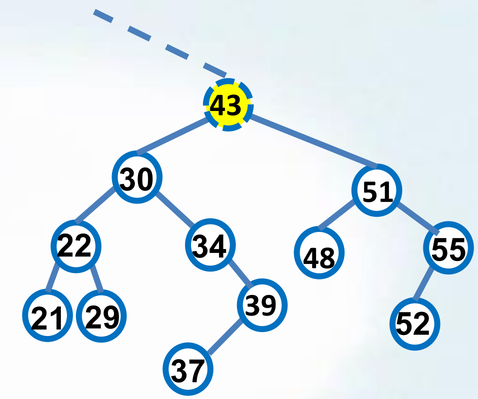
سه حالت داریم:

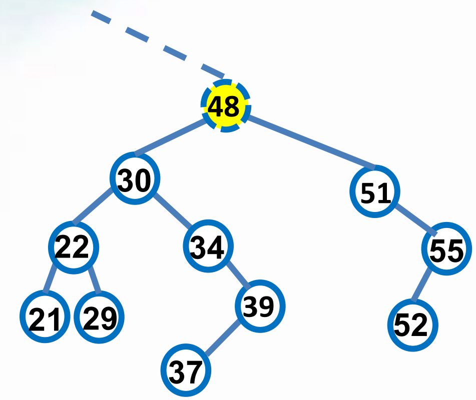
- **گره بچه نداشته باشد:** در قسمت فرزند از گره پدر NULL میگذاریم. اگر فرزند چپ بود فرزند چپ وگرنه فرزند راست
- **گره تک فرزند باشد:** فرزند این گره جایگزین خود گره در گره پدرش می شود
- **گره دو فرزندی باشد:** در این حالت مانند حالت قبل نمیتوان دو فرزند را جای خود گره حذفی قرار داد. از طرفی حذف فیزیکی گره حذفی با باز سازی مجدد درخت همراه خواهد بود. لذا بهترین راه این است که یک گره تک فرزندی یا بدون فرزند را به خانه گره حذفی منتقل کنیم و خانه اش را حذف فیزیکی کنیم.
- هر گرهی از مجموعه فرزندان راست گره حذفی قطعا بزرگتر از فرزندان چپ گره حذفی می باشد پس با فرزندان چپ گره حذفی مشکلی نخواهد داشت لذا باید گرهی انتخاب کرد که از کلیه فرزندان راست گره حذفی کوچکتر باشد. چپ ترین گره در درخت BST کوچکترین گره است. پس چپ ترین گره از فرزندان راست گره حذفی گزینه مناسبی برای جایگزنی با خانه گره حذفی می باشد. چون چپ ترین است فرزند چپ ندارد لذا تک فرزندی یا بدون فرزند است. با همین استدلال راست ترین گره از مجموعه فرزندان چپ نیز میتواند با گره حذفی جایگزین شود.

یک حرکت به راست ، متنها علیه چپ یا
یک حرکت به چپ منتها علیه راست

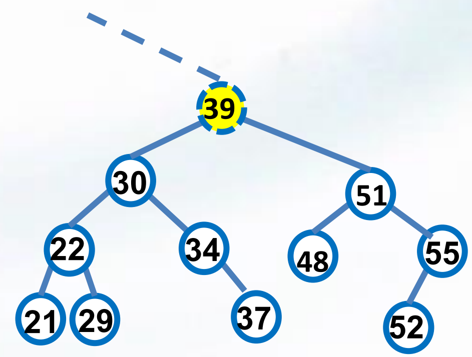
درخت جستجوی دودویی (BST) – حذف عنصر –مثال

برای حذف گره مشخص شده با نقطه چین درخت چگونه خواهد شد؟ (هر دو روش را نمایش دهید)

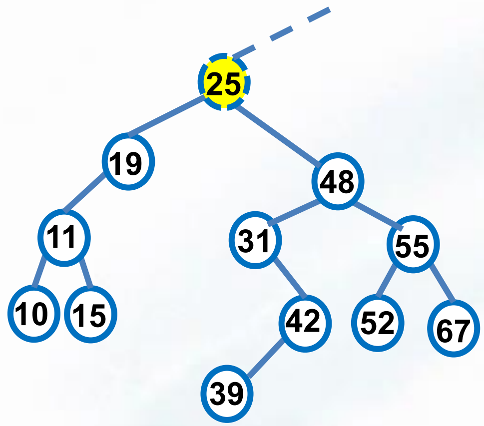


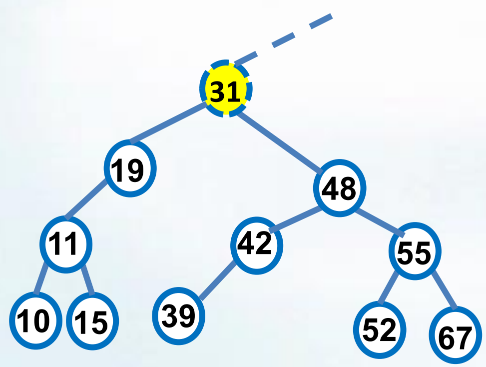


روش ۱
جایگزینی گره 48 و
حذف آن

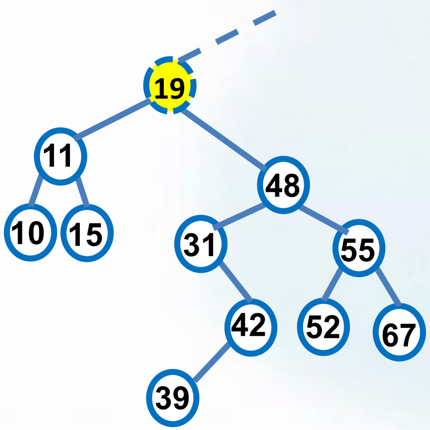


روش ۲
جایگزینی گره 39 و
حذف آن





روش ۱
جایگزینی گره 31 و
حذف آن



روش ۲
جایگزینی گره 19 و
حذف آن