ساختمان داده ها
Data Structure

فصل پنجم
درخت tree

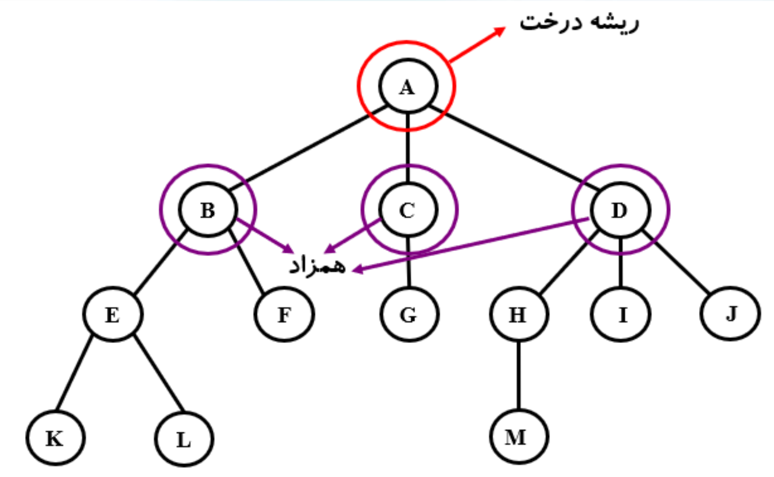
تهیه و تنظیم : محمد نعیمی

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

تهیه و تنظیم : محمد نعیمی

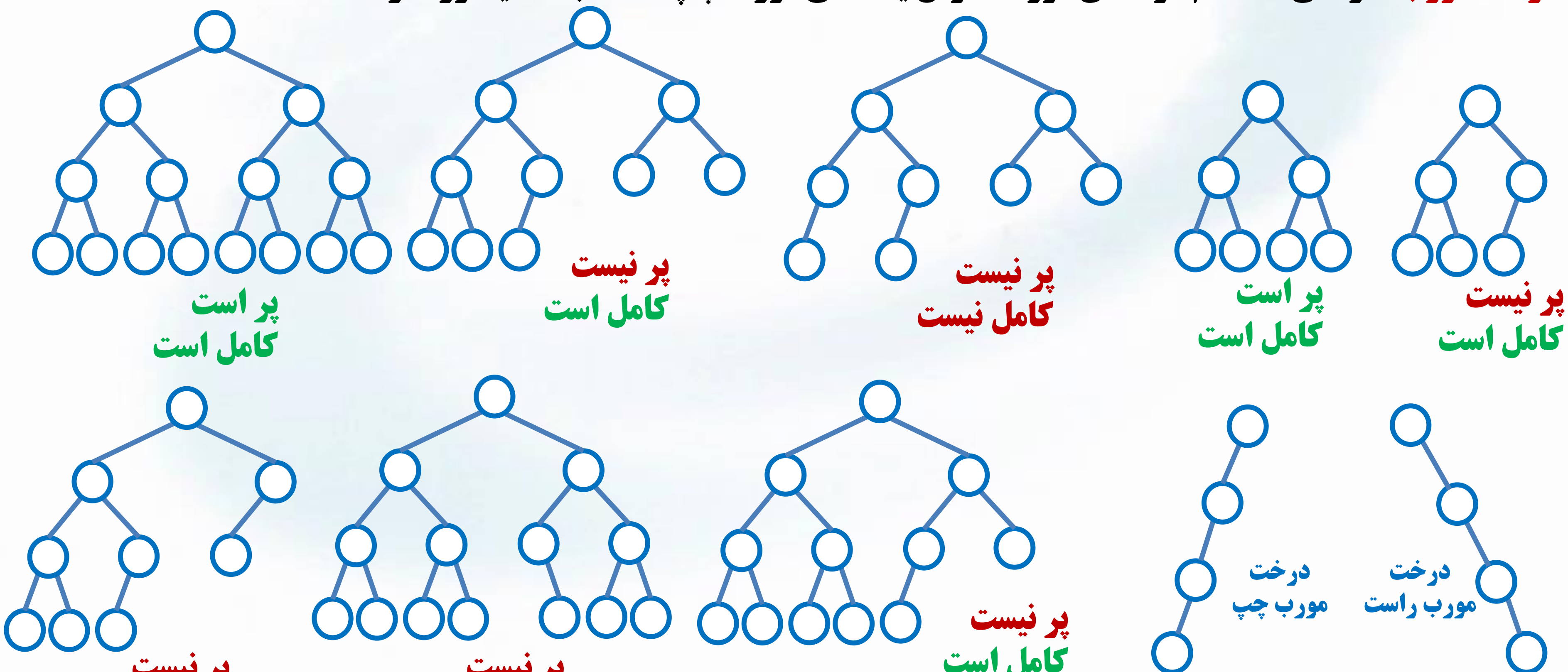
برخی از مدلهای درخت دودویی (binary)

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



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:

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

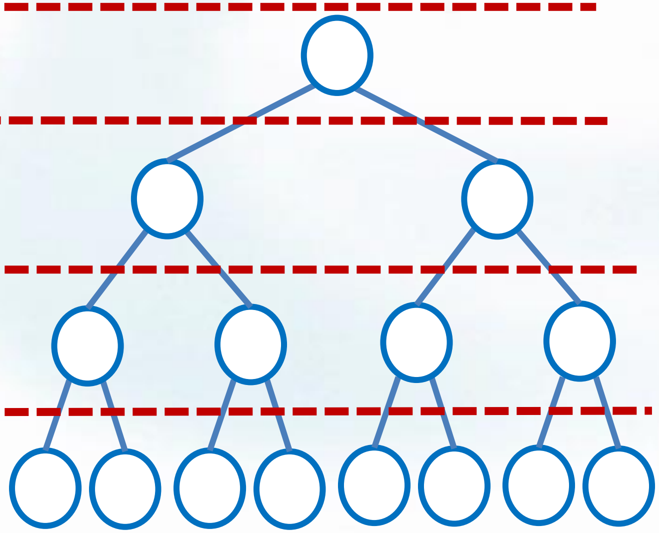
تعداد گره ها میشود تعداد کل فرزندان بعلاوه 1 (ریشه) که فرزند کسی نیست

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

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

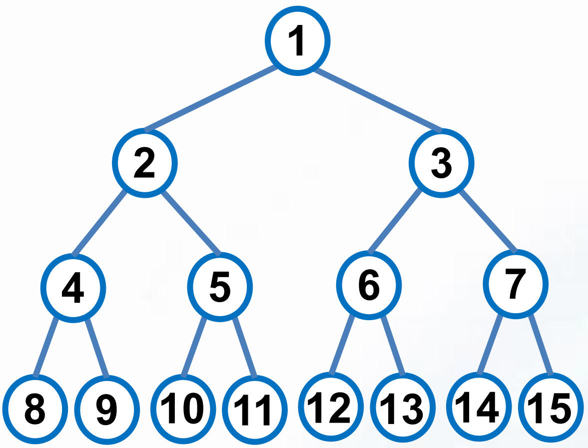
عمق درخت کامل با n گره می شود $\lfloor \log_2 n \rfloor + 1$

تعداد گره سطح	
1	1
2	2
3	4
4	8



تعداد کل گره ها $2^4 - 1 = 15$

نحوه ذخیره سازی درخت دودویی در برنامه (۱)

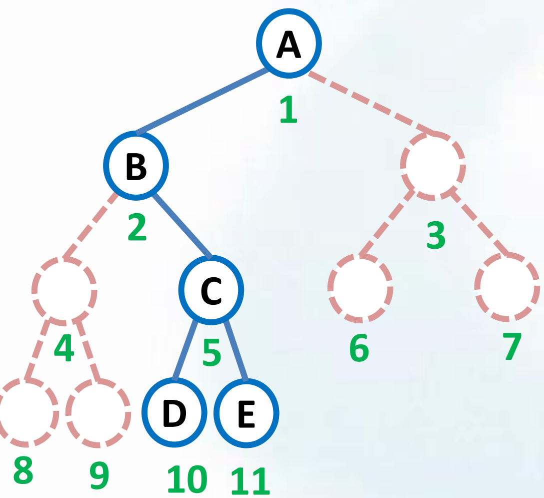


درخت با استفاده از آرایه:

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

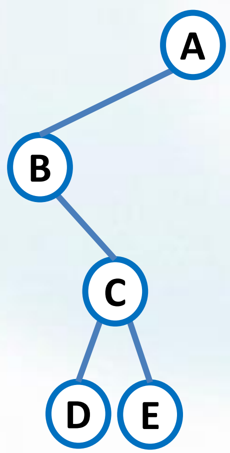
```
struct nodeT {
        type item;
        nodeT *left,*right;
    }
```

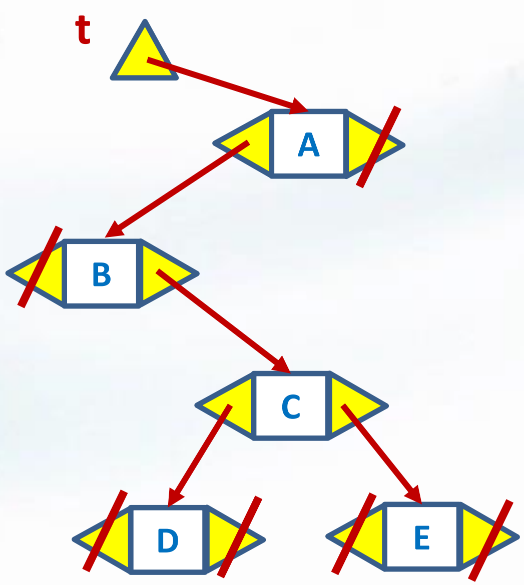
تعریف رکورد گره های درخت

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)

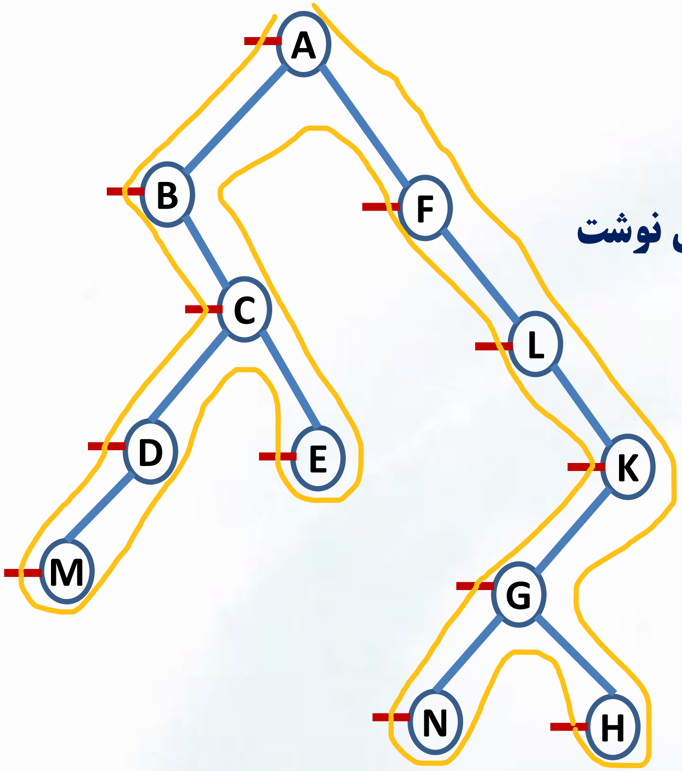
۲– پیمایش میانوندی (inorder) :
این پیمایش به LVR نیز مشهور است. ابتدا کل درختچه چپ به صورت میانوندی پیمایش می شود سپس خود گره رویت می شود و پس از آن کل درختچه راست به صورت پیشوندی

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);
        }
  }
```

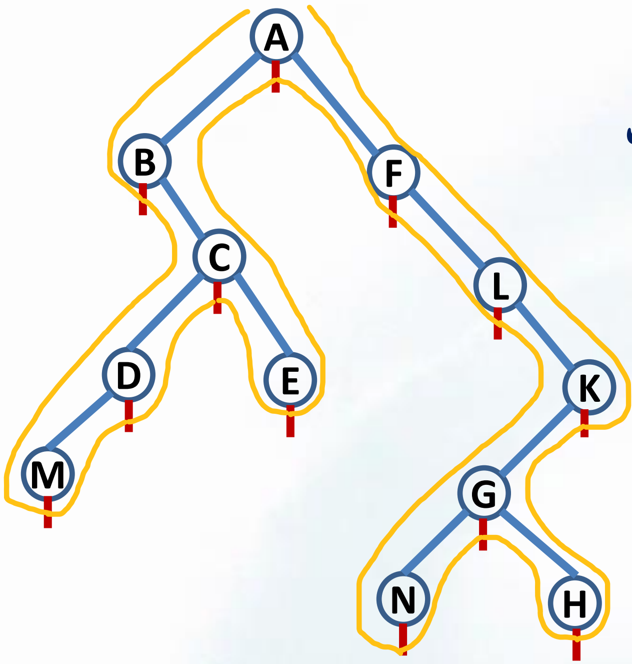
روش سریع:
کافی است در **سمت چپ** گره ها یک خط بکذاریم
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت چپ
گره رسیدیم آن را چاپ میکنیم.

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);
     }
  }
```

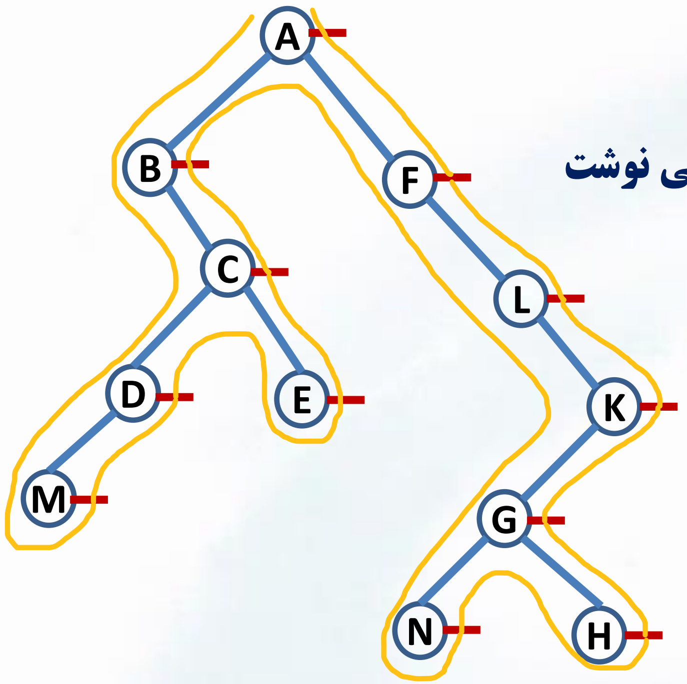
مدل غیر بازگشتی

```
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

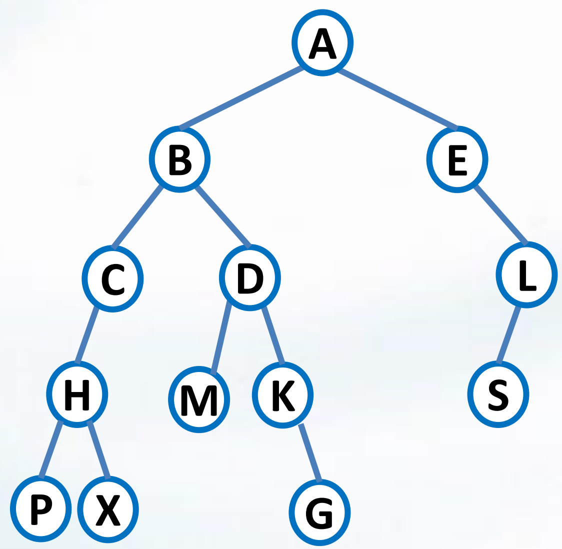
روش سریع:
کافی است در سمت راست گره ها یک خط بکذاریم
دور تا دور درخت را از سمت چپ یک کادر میکشیم و هرگاه به خط سمت راست گره رسیدیم آن را چاپ میکنیم.

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->right);
      }
  }
```

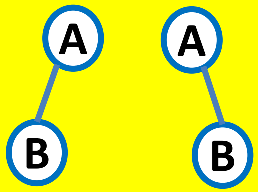
BFS: ABECDLHMKSPXG

ساخت درخت بر اساس پیمایش های آن

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.

این دو درخت متفاوت پیمایش pre و post یکسان دارند:

pre:AB post:BA

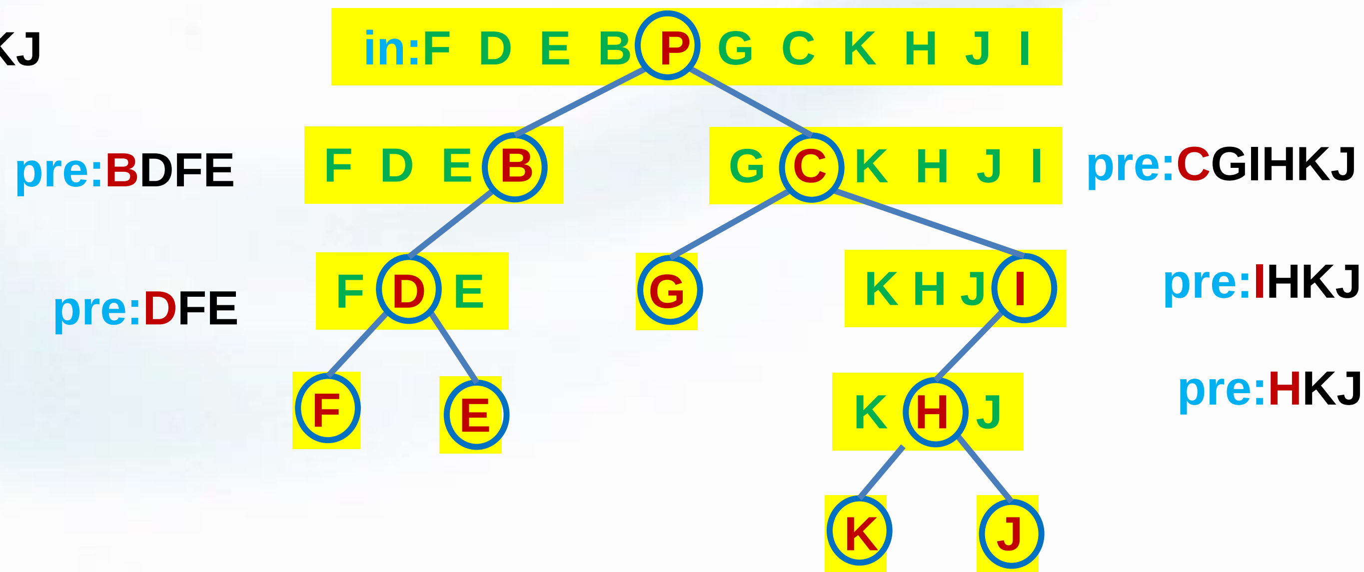


الگوریتم ساخت درخت از پیمایش in و pre:

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.

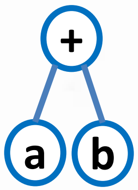
- در صورت داشتن post بجای pre ، آخرین عنصر پیمایش post "ریشه" می باشد.

in:FDEBPGCKHJI
pre:PBDFECGIHKJ

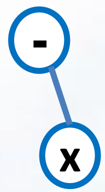


نمایش عبارت محاسباتی با درخت باینری(دودویی)

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.



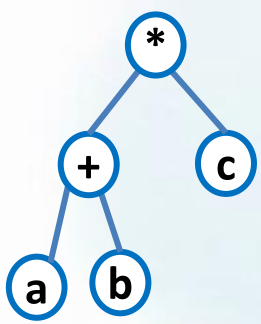
a+b



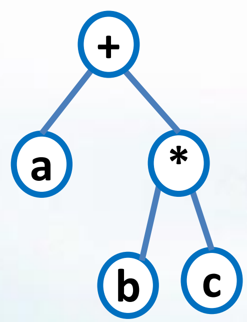
-x

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

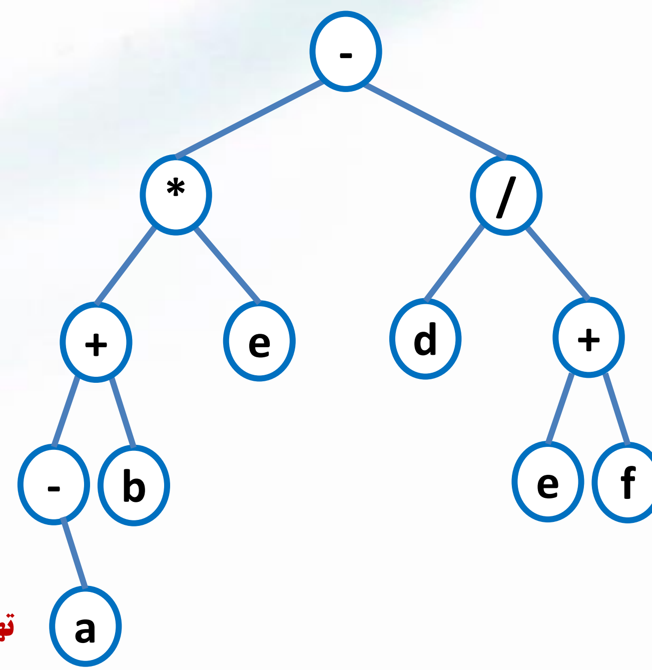
(a+b)*c



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، 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);
  }
```

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

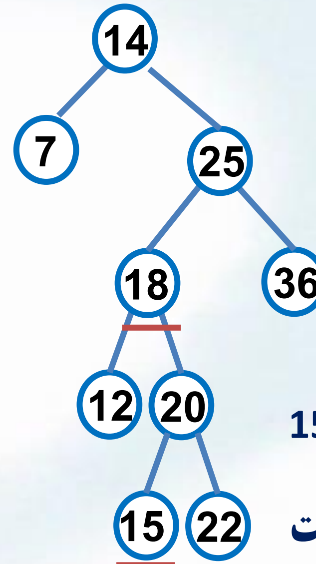
```
int  leaf (nodeT *t)
  {
     if (t==NULL)
         return(0);
     if (t->left==NULL && t->right==NULL)
         return(1);
     return(leaf(t->left) + leaf(t->right));
  }
```

تابع بازگشتی محاسبه تعداد گره های دو فرزندی یک درخت

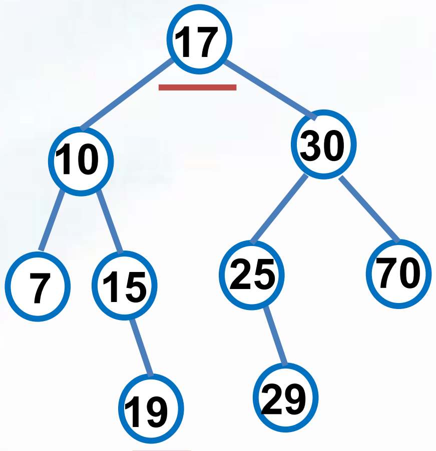
```
int  parent2 (nodeT *t)
  {
     if (t==NULL)
         return(0);
     if (t->left!=NULL && t->right!=NULL)
         return(1+ parent2(t->left) + parent2(t->right));
     return(parent2(t->left) + parent2(t->right));
  }
```

درخت های ویژه– درخت جستجوی دودویی (BST)

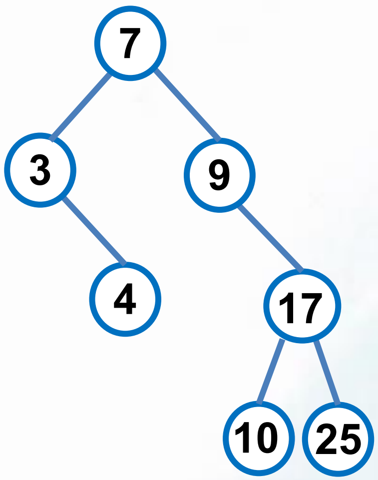
درخت جستجوی دودویی درختی است که در آن تمام گره های سمت چپ هر گره از آن گره کوچکتر و تمام گره های سمت راست گره از آن گره بزرگتر باشند.
نکته: در درخت جستجوی دودویی عنصر تکراری مجاز نیست.



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



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



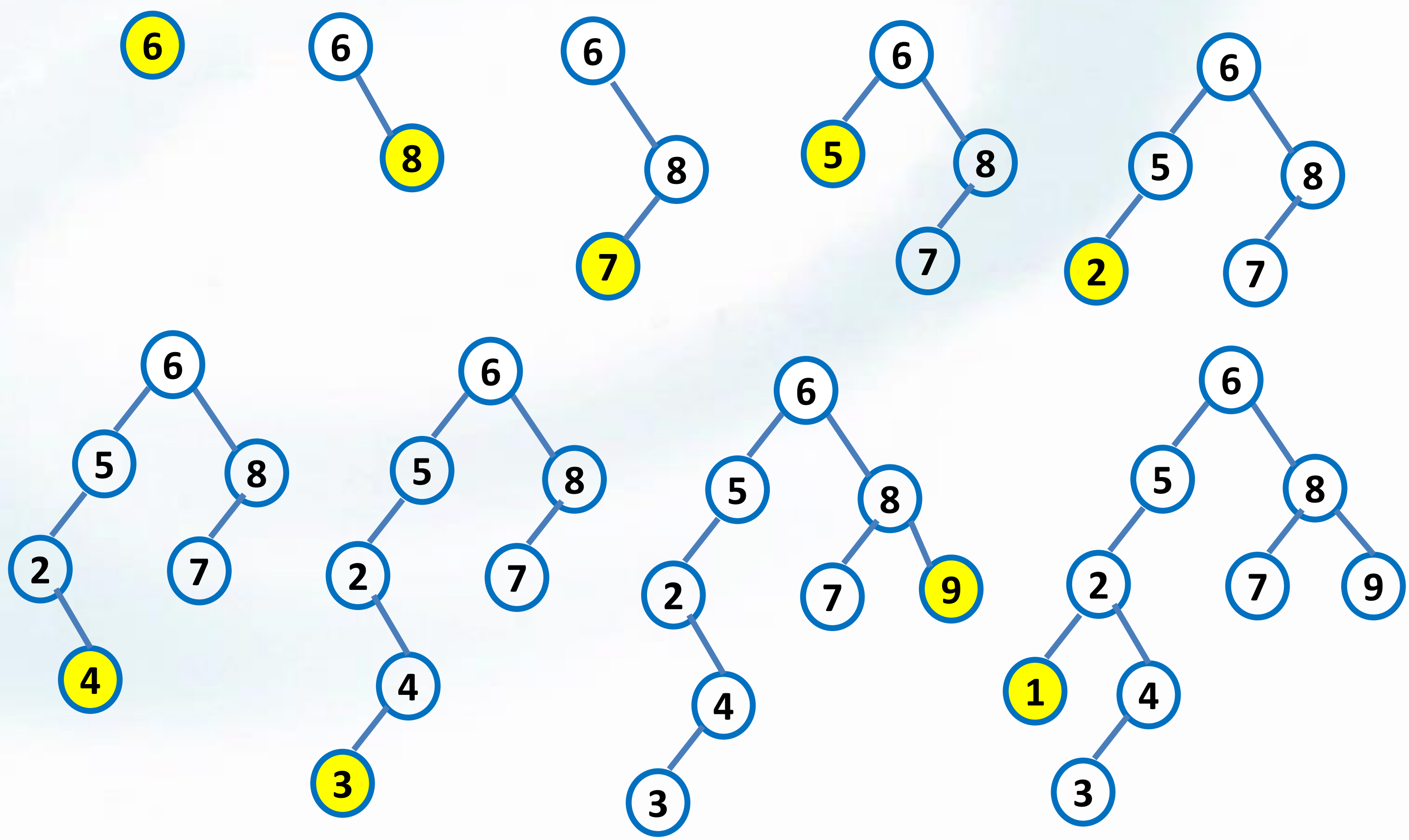
BST است

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.

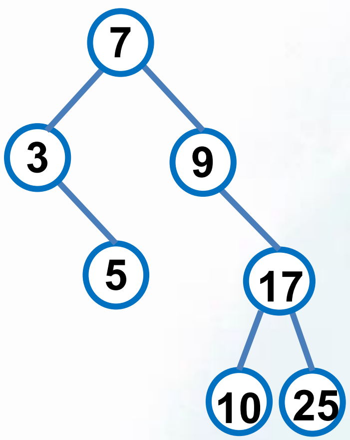
درخت جستجوی دودویی (BST) – افزودن عنصر

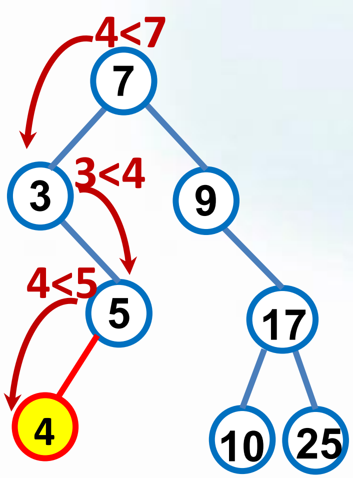
اگر درخت خالی بود عنصر وارد شده ریشه درخت خواهد بود.
اگر درخت خالی نبود، در صورتی که عنصر وارد شده از ریشه کمتر بود به سمت چپ و اگر بزرگتر بود سمت راست میرویم (عنصر مساوی وارد نمی شود و حذف می شود) و همین روند را آنقدر ادامه میدهیم که به جای خالی برسیم. عنصر جدید را در جای خالی اضافه میکنیم

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.
`}`	

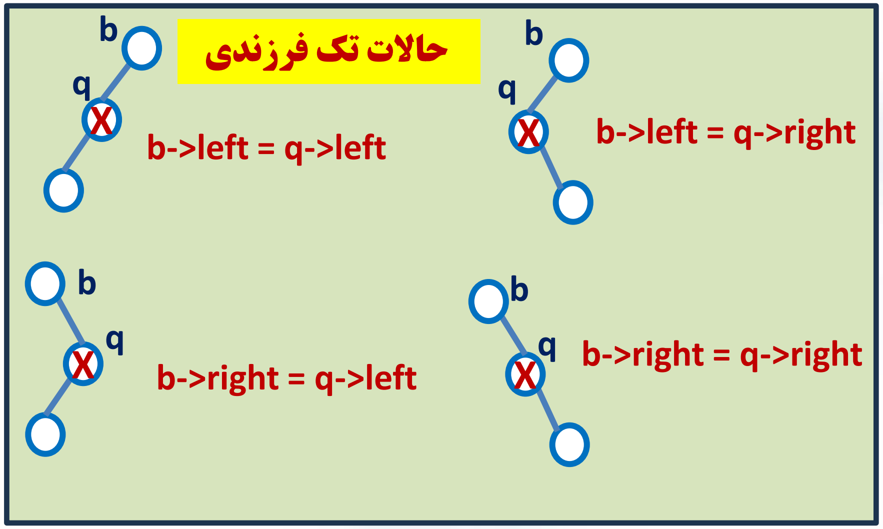
درخت جستجوی دودویی (BST)– تابع جستجو عنصر

```
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));
   }
```

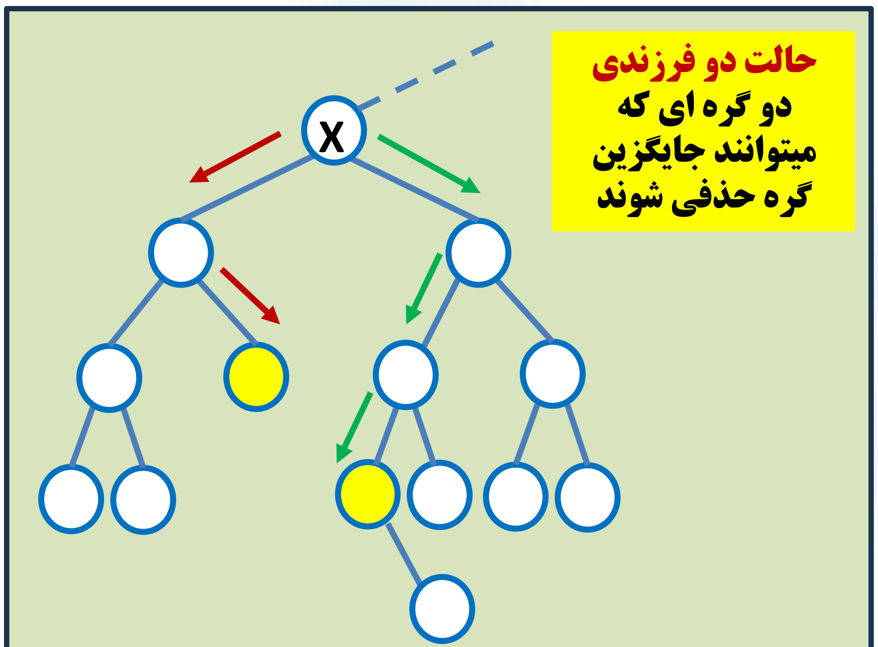
پیچیدگی این دو تابع به اندازه عمق درخت است
دربدترین حالت o(n) و
بهترین حالت و متوسط o(logn)

درخت جستجوی دودویی (BST) – حذف عنصر



سه حالت داریم:

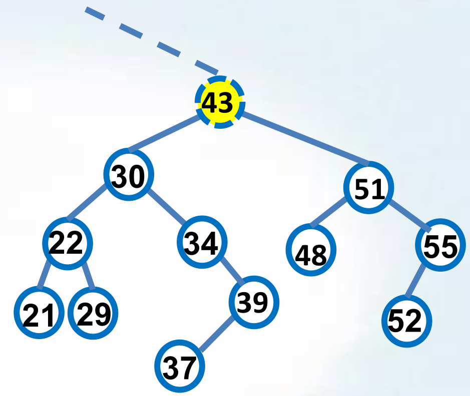
- **گره بچه نداشته باشد:** در قسمت فرزند از گره پدر NULL میگذاریم. اگر فرزند چپ بود فرزند چپ وگرنه فرزند راست
- **گره تک فرزند باشد:** فرزند این گره جایگزین خود گره در گره پدرش می شود
- **گره دو فرزندی باشد:** در این حالت مانند حالت قبل نمیتوان دو فرزند را جای خود گره حذفی قرار داد. از طرفی حذف فیزیکی 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.

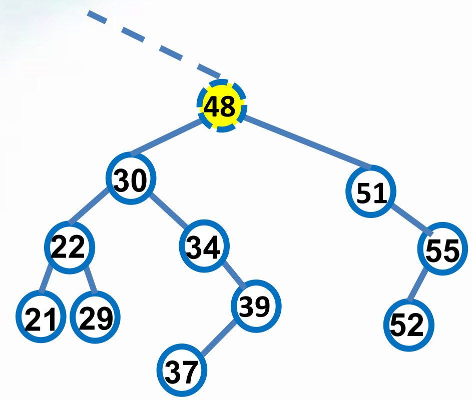


یک حرکت به راست ، متنها علیه چپ یا
یک حرکت به چپ منتها علیه راست

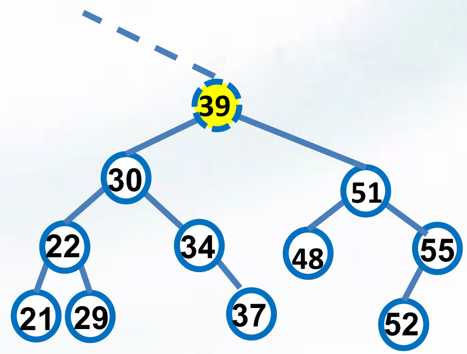
درخت جستجوی دودویی (BST) – حذف عنصر–مثال

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

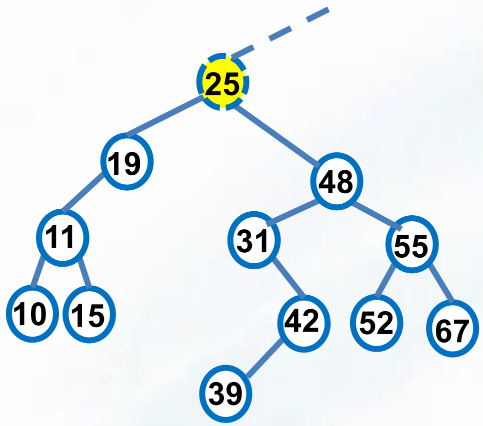


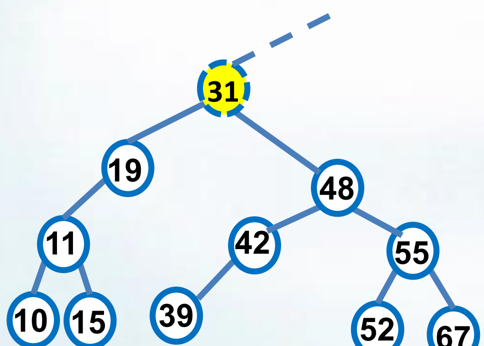


روش ۱
جایگزینی گره 48 و حذف آن

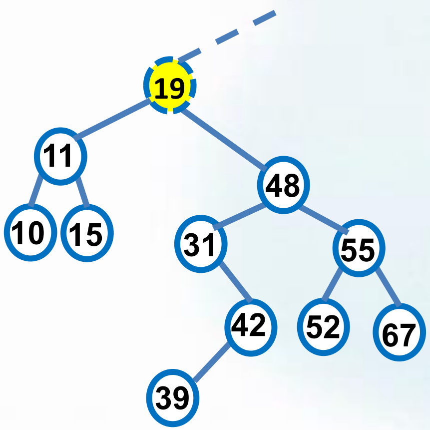


روش ۲
جایگزینی گره 39 و حذف آن





روش ۱
جایگزینی گره 31 و حذف آن



روش ۲
جایگزینی گره 19 و حذف آن

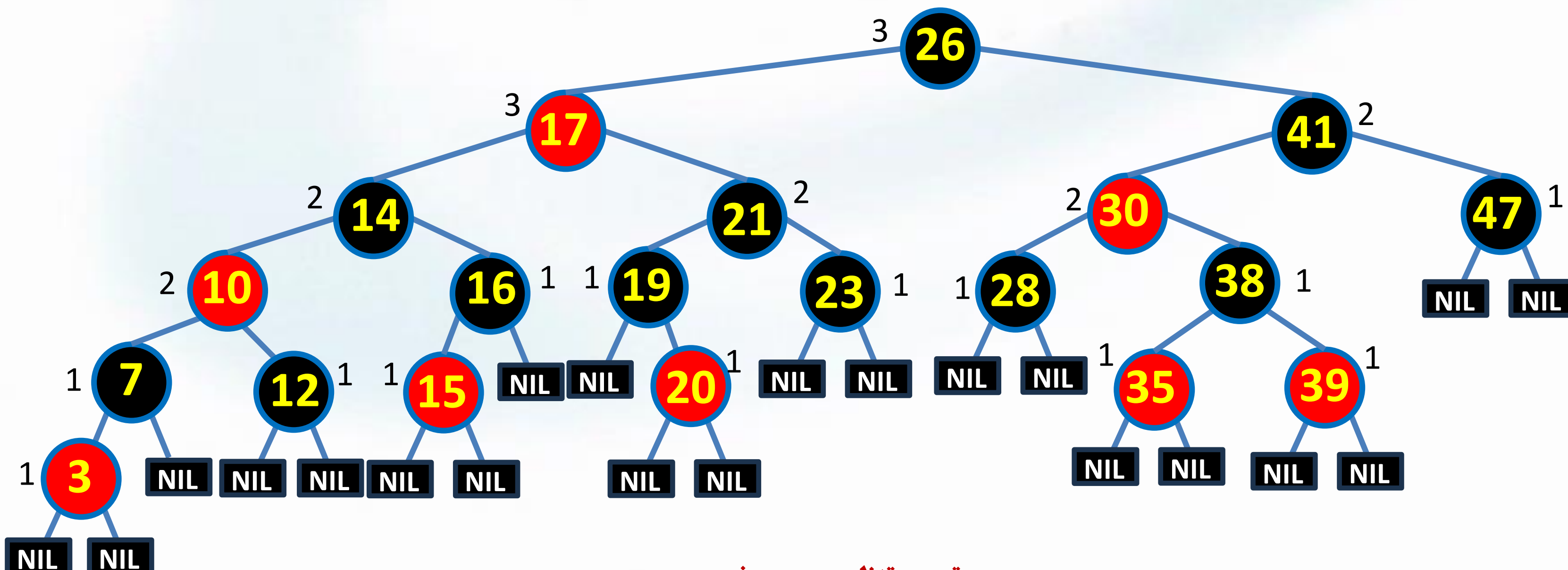
درخت قرمز–سیاه (red-black tree)

❑ درخت قرمز–سیاه یک درخت جستجوی باینری خود متعادل کننده است که در آن هر گره حاوی یک بیت اضافی برای نشان دادن رنگ گره، قرمز یا سیاه است. خصوصیات این درخت عبارت است از:

1. هر گره یا قرمز است یا سیاه.
2. ریشه سیاه است.
3. گره NIL همیشه سیاه است (NIL گره ای است که به عنوان فرزند تمام برگ های درخت گره هایی که یک فرزند ندارند اضافه می شود).
4. اگر گره ای قرمز بود هر دو فرزند آن سیاه می باشند.
5. برای هر گره، تمام مسیر های ساده از آن گره تا NIL تعداد یکسانی گره سیاه داشته باشد(ارتفاع سیاه).

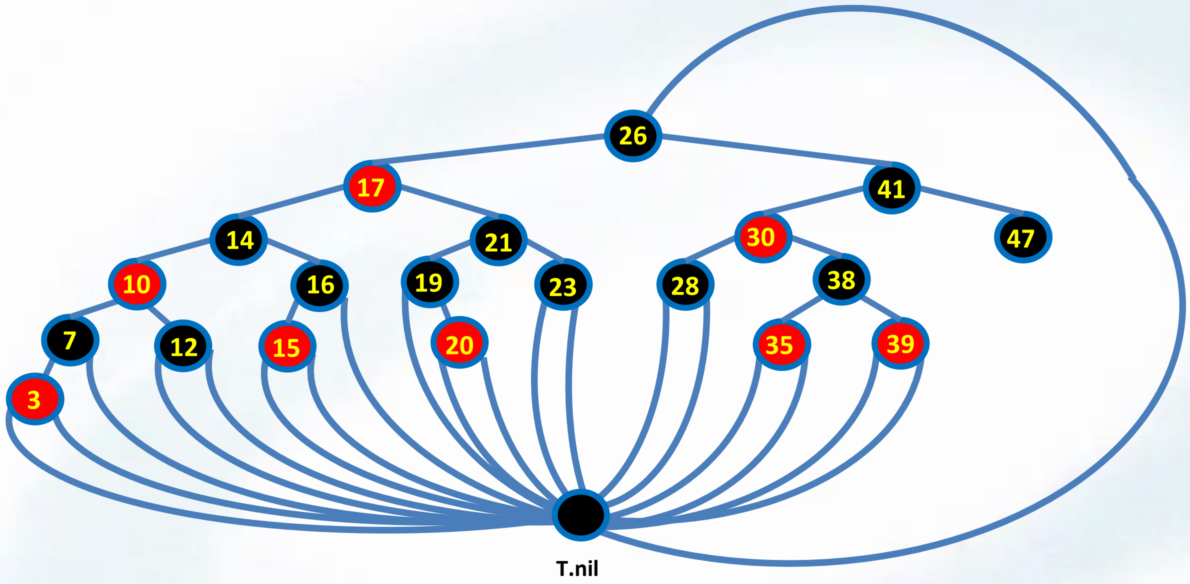
قضیه: درخت قرمز–سیاه با n گره حداکثر عمق 2log(n) خواهد داشت.

هر گره علاوه بر left و right دارای لینک p به پدر گره نیز می باشد.

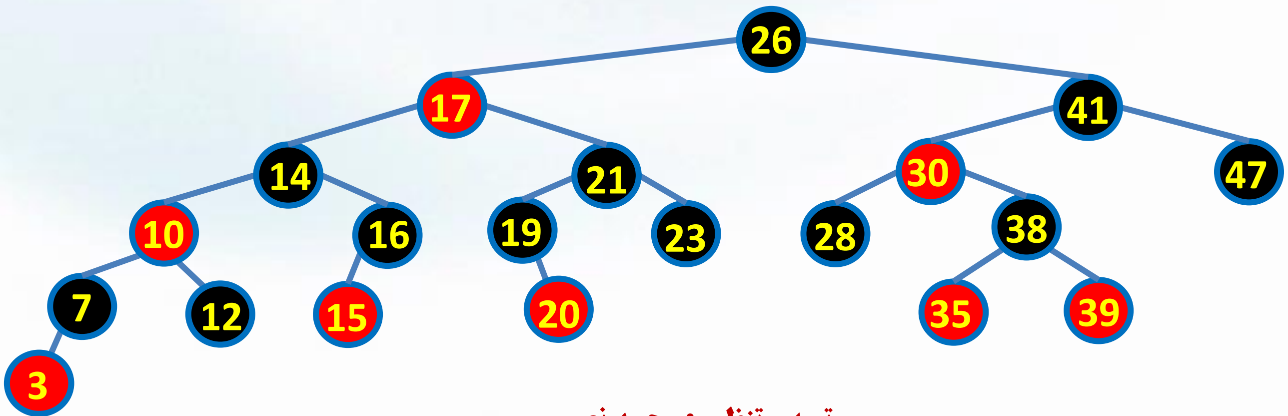


درخت قرمز–سیاه (red-black tree) – ادامه

برای سهولت در ایجاد درخت از یک گره **نگهبان** (T. nil) برای نشان دادن NIL استفاده میکنیم. گره T.nil یک گره با همان ویژگی های سایر گره هاست فقط ویژگی رنگی آن سیاه است. در شکل زیر همه اشاره گره ها به NIL با اشارگر به T.nil جایگزین شده است



به طور کلی هنگام نمایش گره T.nil را رسم نمیکنیم و درخت قرمز–سیاه را به فرم زیر نمایش میدهیم.



درخت قرمز–سیاه (red-black tree)–تابع افزودن عنصر

```
void  add_red_black (nodeT * &t, type x)
  {
    nodeT *z,*q,*b =T.nil;
    q=t;
    while (q!=T.nil)
      {
        if (x.key==q->item.key)
          { cout<< “Repeated number” ; return(); }
        b=q;
        if (x.key<q->item.key)
            q=q->left;
        else
            q=q->right;
      }
    z=new();
    z->item=x;
    z->p=b;
    z->right=p->left=T.nil;
    z->color=RED
    if(t==NULL)
      t=z;
    else
      if(x.key<b->item.key)
         b->left=z;
      else
         b->right=z;
    red_black_fixup(t,z);
  }
```

- افزودن گره به درخت قرمز–سیاه از دو مرحله تشکیل شده است.

۱– مانند افزودن گره به درخت جستجوی دودویی(BST) مکان گره انتخاب و رنگ گره قرمز در نظر گرفته می شود.

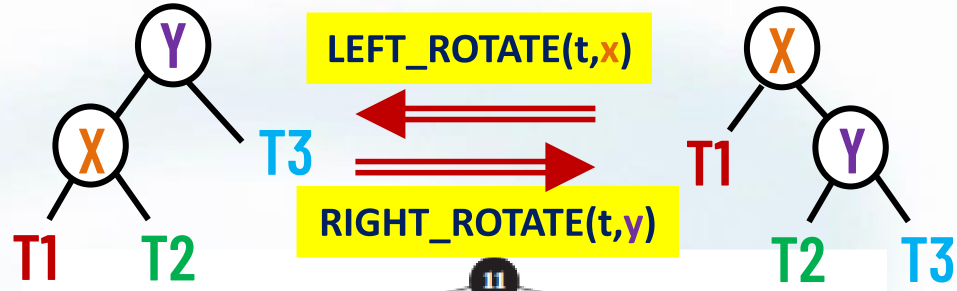
۲– پس از مرحله قبل، تابعی جهت اصلاح مواردی که ممکن است قواعد درخت قرمز–سیاه را نقض کند فراخوان می شود. قواعد ۲ یا ۴ یا ۵ ممکن است نقض شود.

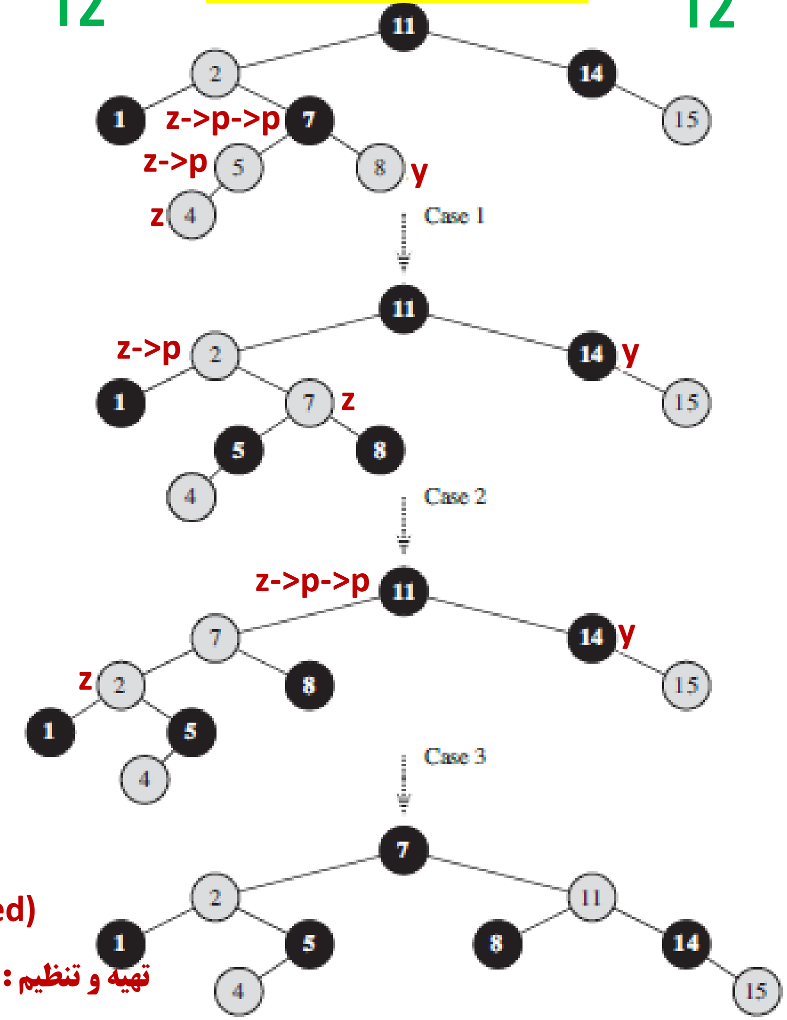
در تابع red_black_fixup(t,z) اصلاح رنگها و ساختار درخت انجام می پذیرد.

- الگوریتم حذف از درخت قرمز–سیاه مانند حذف از درخت جستجوی دودویی بسیار پیچیده تر از عملیات افزودن است و از بیان آن در این درس صرف نظر میکنیم.

درخت قرمز—سیاه (red-black tree)—تابع افزودن عنصر

```
void  red_black_fixup (nodeT * &t, nodeT* z)
  {
     nodeT *x,*y;
     while (z->p->color==RED)
       if (z->p==z->p->p->left)
          {
              y=z->p->p->right;
              if (y->color==RED)
                 {
                    z->p->color=BLACK
                    y->color=BLACK                    case 1
                    z->p->p->color=RED
                    z=z->p->p;
                 }
              else
                 {
                    if (z==z->p->right)
                       {
                          z=z->p;
                          LEFT_ROTATE(t,z);           case 2
                       }
                    z->p.color=BLACK;
                    z->p->p=RED;                       case 3
                    RIGHT_ROTATE(t,z->p->p)
                 }
          }
        else
           (same as if clause with "right" and "left" exchanged)
     t.color=Black;
  }
```

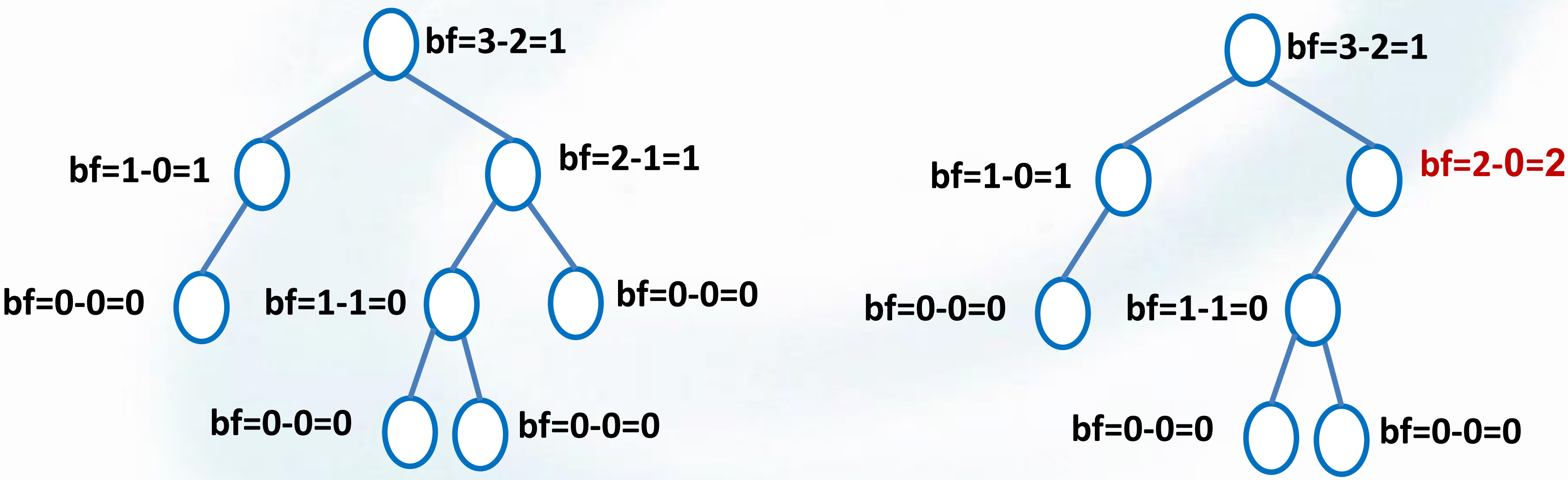




تهیه و تنظیم : محمد نعیمی

درخت AVL

- درخت‌های AVL که به نام خالقانشان Adelson، Velsky و Landis نام‌گذاری شده‌اند.
- درخت AVL یک درخت جستجوی باینری خود متعادل کننده است .
- درخت AVL تضمین می‌کند که عمق زیردرخت راست و چپ هر گره اختلافی بیش‌تر از 1 ندارند.
- این اختلاف به نام عامل تعادل (Balance Factor) نامیده می‌شود.



درخت AVL است

درخت AVL نیست

تهیه و تنظیم : محمد نعیمی

درخت AVL – الگوریتم افزودن گره

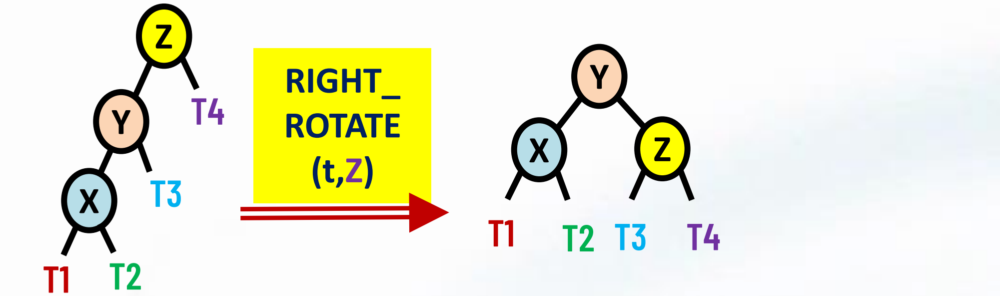
افزودن گره به درخت AVL:

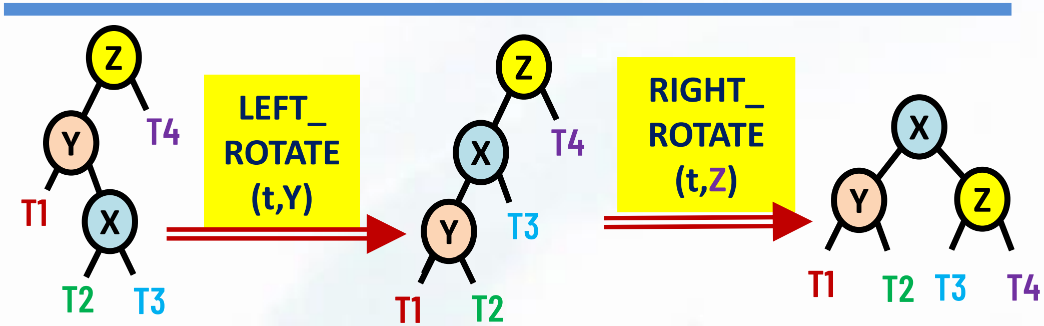
افزودن گره به درخت AVL دو مرحله دارد:

۱– مانند افزودن گره به درخت جستجوی دودویی(BST) مکان گره انتخاب و گره اضافه می شود.

۲– از گره افزوده شده (w) به سمت بالا حرکت میکنیم تا به اولین گره با BF بالای 1 برسیم. این گره را z فرض کنید. فرزند و نوه این گره در مسیر رسیدن به w را به ترتیب y و x می نامیم.

x، y و z با هم یکی از حالات مقابل را دارند و مطابق شکل حالات نشان داده شده را اصلاح میکنیم.





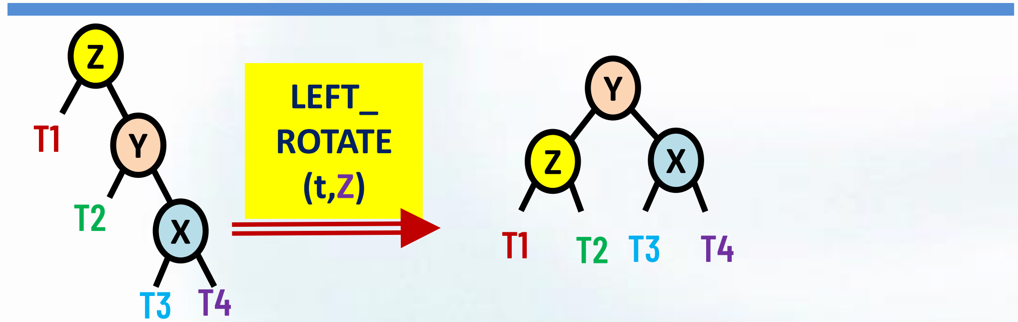
حذف گره از درخت AVL:

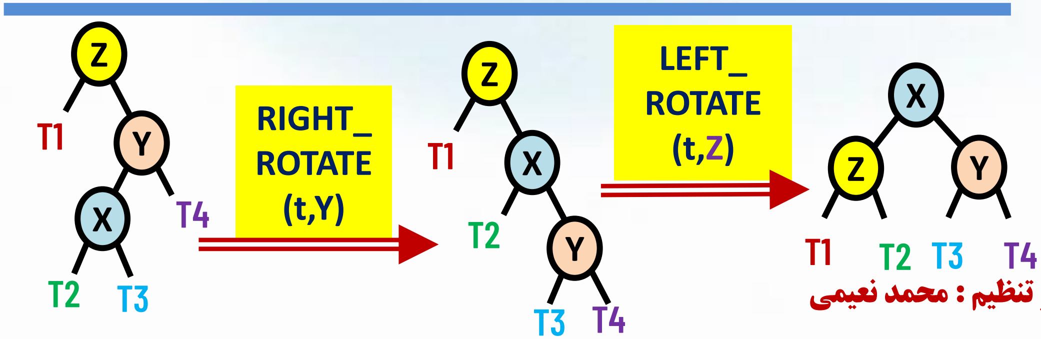
حذف گره از درخت AVL دو مرحله دارد:

۱– مانند حذف گره از درخت جستجوی دودویی (BST) گرهی که به صورت فیزیکی حذف می شود را w می نامیم.

۲– از گره افزوده شده (w) به سمت بالا حرکت میکنیم و x، y و z را طبق الگو افزودن مشخص میکنیم.

x، y و z با هم یکی از حالات مقابل را دارند و مطابق شکل حالات نشان داده شده را اصلاح میکنیم.





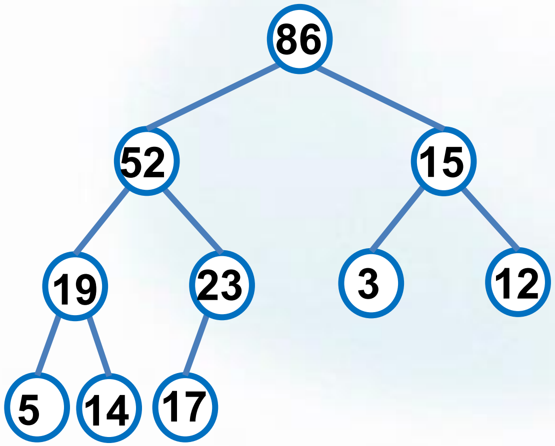
تهیه و تنظیم : محمد نعیمی

درخت های ویژه– درخت هرم (heap)

- درخت هرم به صورت max-heap و min-heap معرفی میشود.
- درخت max-heap یک درخت **کامل** است که هر گره از فرزندانش **بزرگتر یا مساوی** است.
- درخت min-heap یک درخت **کامل** است که هر گره از فرزندانش **کوچکتر یا مساوی** است.
- چون درخت کامل است برای پیاده سازی بهتر است از ساختار آرایه استفاده کنیم.

ما max-heap را معرفی میکنیم. (در min-heap جای عبارت بزرگتر و کوچکتر عوض می شود).
عبارت درخت heap بدون مشخص نمودن نوع max یا min را درخت max-heap در نظر میگیریم.

- بزرگترین عنصر درخت قطعا ریشه می باشد پس یافتن ماکزیمم در max-heap پیچیدگی o(1) دارد.
- کوچکترین عنصر درخت قطعا برگ می باشد (چون برگ فرزندی ندارد). برای گره i اولین فرزند(چپ) در خانه 2i قرار دارد لذا تمام برگها در نیمه دوم آرایه قرار دارد وبرای یافتن آن n/2 خانه ها را باید جستجو کنیم.
- در پیاده سازی بجای خانه ی 0 آرایه، تعداد عناصر را در متغیر n قرار میدهیم تا نوشتار ساده تر شود.



n=10

10	86	52	15	19	23	3	15	5	14	17
0	1	2	3	4	5	6	7	8	9	10

ریشه
ماکزیمم

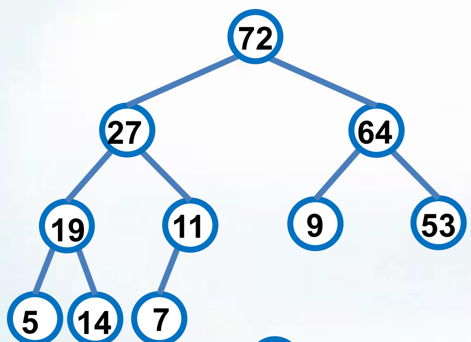
برگها

تهیه و تنظیم : محمد نعیمی

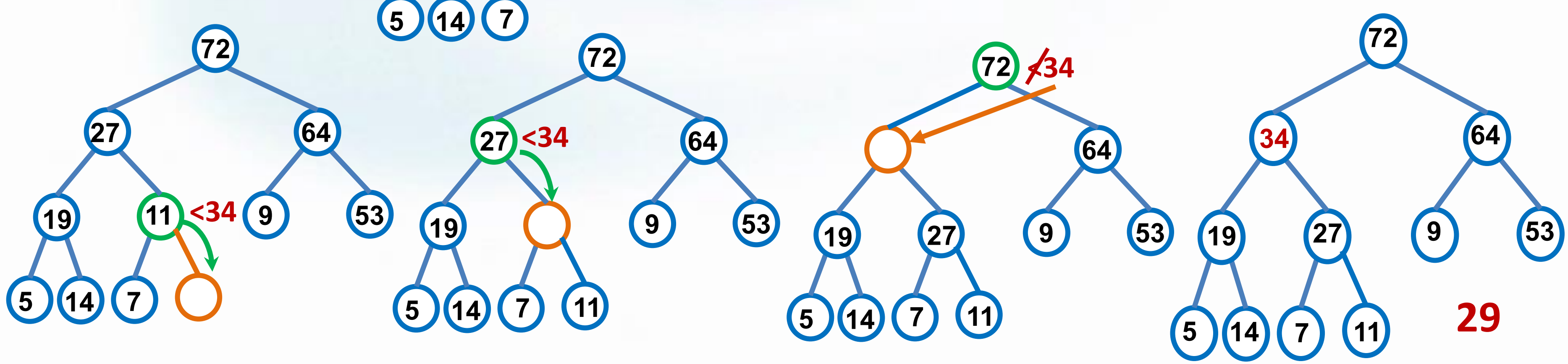
درخت هرم ماکزیمم(max-heap)—افزودن عنصر

افزودن عنصر به درخت max-heap :
از آنجایی که درخت باید کامل باشد گره فیزیکی که ایجاد میگردد دقیقا مشخص است در کجای درخت است. گره جدید بعد از آخرین گره باید قرار بگیرد یعنی در خانه n+1 آرایه.
اما افزودن گره جدید در این خانه ممکن است باعث ایجاد تناقض با پدرش گردد (از پدرش بزرگتر باشد) لذا الگوریتم افزودن مقدار x به درخت به فرم زیر است:

1. تعداد عناصر n+1 می شود و خانه n+1 به عنوان خانه پیشنهادی(k) برای مقدار x در نظر گرفته می شود.
2. اگر خانه پیشنهادی(k)، ریشه بود (k=1 یعنی پدر نداشت) یا مقدار x از مقدار پدرخانه پیشنهادی([k/2]) کمتر یا مساوی بود، x را در خانه پیشنهادی(k) قرار داده کار تمام میشود.
3. اگر مقدار x بیشتر از پدر خانه پیشنهادی([k/2]) بود، مقدار خانه پدر (a[k/2]) را به خانه پیشنهادی(k) منتقل نموده و خانه پدر را به عنوان خانه پیشنهادی در نظر گرفته به مرحله ۲ می رویم



به درخت max-heap مقابل عدد 34 را اضافه کنید



درخت هرم ماکزیمم(max-heap)–تابع افزودن عنصر

```
void  add_heap (type a[], int &n,int x)
  {
    int k;
    k=++n;
    while(k>1 && a[k/2]<x)
      {
        a[k]=a[k/2];
        k=k/2;
      }
    a[k]=x;
  }
```

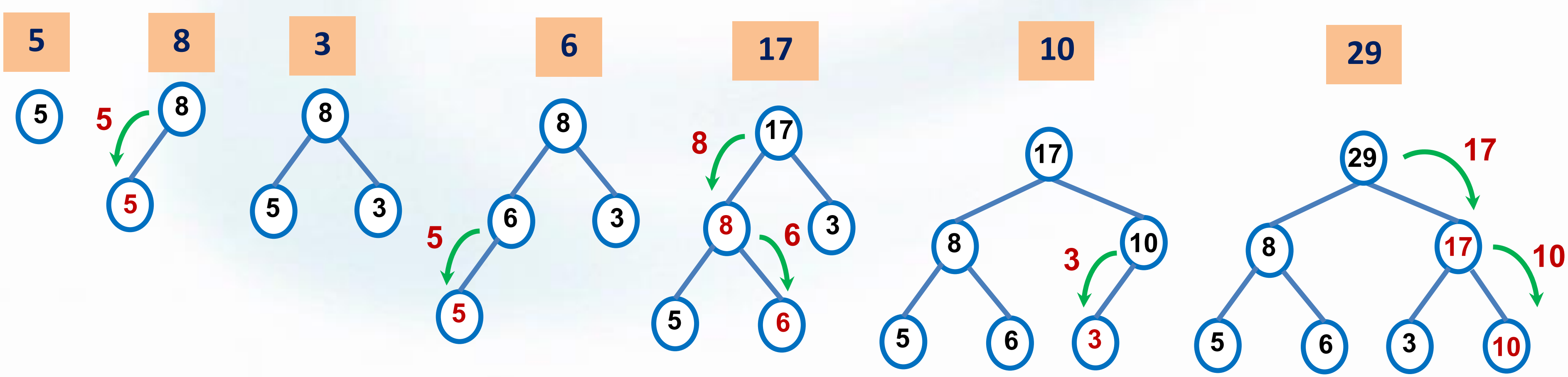
تا وقتی که خانه پیشنهادی، ریشه نباشد و مقدار x از مقدار پدرخانه پیشنهادی بیشتر باشد

انتقال مقدار خانه پدر (a[k/2]) را به خانه پیشنهادی(k)

خانه پدر را به عنوان خانه پیشنهادی در نظر بگیر

در این تابع نوع مقادیر را int فرض کردیم که البته اگر بخواهیم نوع مقادیر دیگر را داشته باشیم به سادگی با تغییر نوع متغیر امکان پذیر است

اعداد زیر را به ترتیب به یک درخت max-heap خالی اضافه کنید(از چپ به راست).



درخت هرم ماکزیمم(max-heap)—حذف عنصر

حذف عنصر از درخت max-heap :

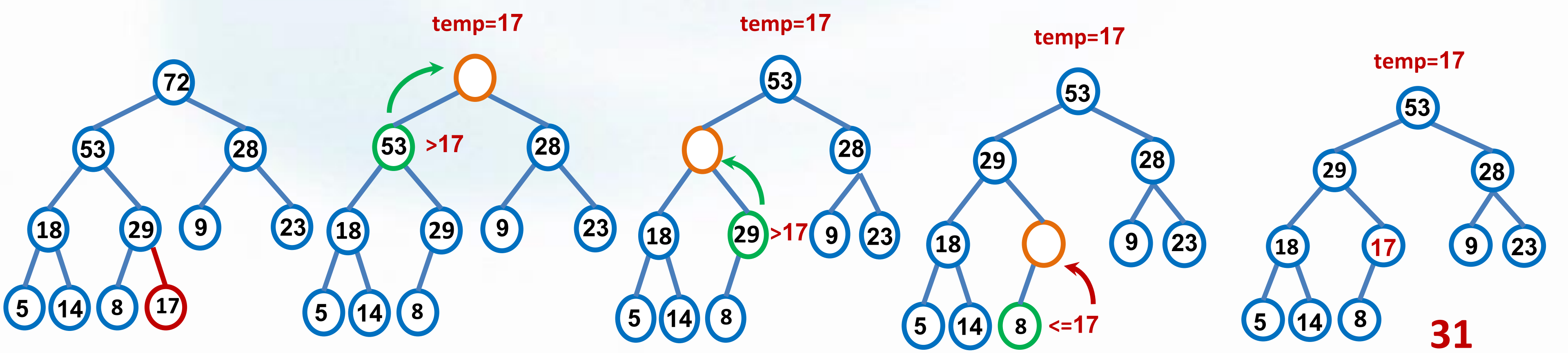
در حذف عنصر از درخت همیشه عنصر ریشه حذف می شود. اما از آنجایی که درخت باید کامل بماند گره آخر از نظر فیزیکی حذف میشود و ریشه جهت قرار گرفتن عنصرموجود در گره آخر پیشنهاد می شود.

درخت باید max-heap بماند لذا عنصر منتقل شده باید با دو فرزند مکان پیشنهادی مقایسه شود.

برای یافتن مکان مناسب برای عنصر آخر(item) الگوریتم زیر پیشنهاد میشود.

1. ریشه به عنوان خانه پیشنهادی(i) معرفی می شود.
2. بین دو فرزند خانه پیشنهادی(i) یعنی گره های 2i و 2i+1 بزرگترین فرزند (j) را پیدا میکنیم.
3. اگر مقدار item بزرگتر یا مساوی از مقدار بزرگترین فرزند (j) بود یا فرزندی نداشت(برگ بود)، مقدار temp در خانه پیشنهادی(i) قرار گرفته و کار تمام می شود.
4. اگر مقدار item کمتر از مقدار بزرگترین فرزند (j) بود، محتوای خانه بزرگترین فرزند (j) را در خانه پیشنهادی(i) قرار می دهیم (چون بزرگترین فرزند است با شاخه کناری که فرزند دیگر است از نظر max-heap بودن مشکلی نخواهد داشت) و خانه بزرگترین فرزند (j) را به عنوان خانه پیشنهادی(i) در نظر گرفته و به مرحله ۲ می رویم

به درخت max-heap مقابل عدد 34 را اضافه کنید



درخت هرم ماکزیمم(max-heap)–تابع حذف عنصر

در این تابع نوع مقادیر را int فرض کردیم که البته اگر بخواهیم نوع مقادیر دیگر را داشته باشیم به سادگی با تغییر نوع منغیر امکان پذیر است

```
int  del_heap (type a[], int &n)
  {
    int i,j,item,root;
    item=a[n--];          // خانه آخر حذف و مقدار آن در item گذاشته می شود
    root=a[1];            // ریشه جهت باز گرداندن در root ذخیره می شود
    i=1;                  // خانه پیشنهادی(i) اولیه خانه ریشه می باشد
    j=2;                  // j ابتدا شماره خانه فرزند چپ است
    while(j<=n)           // تا وقتی که فرزند j وجود دارد (یعنی شماره j در محدوده گره های مجاز است)
      {
        if (j<n)          // چون j فعلا فرزند چپ است باید چک کنیم اگر فرزند راست هم وجود داشت یعنی j+1<=n بود
          if(a[j]<a[j+1]) // و اگر فرزند راست (j+1) بزرگتر بود از چپ (j)
            j++;          // بزرگترین فرزند میشود فرزند راست (از این لحظه j شماره خانه بزرگترین فرزند است)
        if(item>=a[j])    // اگر مقدار Item از بزرگترین فرزند(j) خانه پیشنهادی بزرگتر یا مساوی بود
           break;         // خانه پیشنهادی (i) مناسب است و از حلقه خارج شو
        a[i]=a[j];  // وگرنه(چون در if دستور break داریم نیاز به نوشتن else نیست) بزرگترین فرزند(j) را به خانه پیشنهادی(i) منتقل کن
         i=j;             // خانه بزرگترین فرزند(j) به عنوان خانه پیشنهادی (i) معرفی می شود
         j=j*2;           // j شماره خانه فرزند چپ خانه پیشنهادی(i) می شود
      }
    a[i]=item;            // item در خانه پیشنهادی (i) قرار داده میشود
    return (root);        // مقدار ریشه به عنوان خروجی برگردانده می شود
  }
```

تهیه و تنظیم : محمد نعیمی

درخت هرم ماکزیمم(max-heap)—کاربرد ها

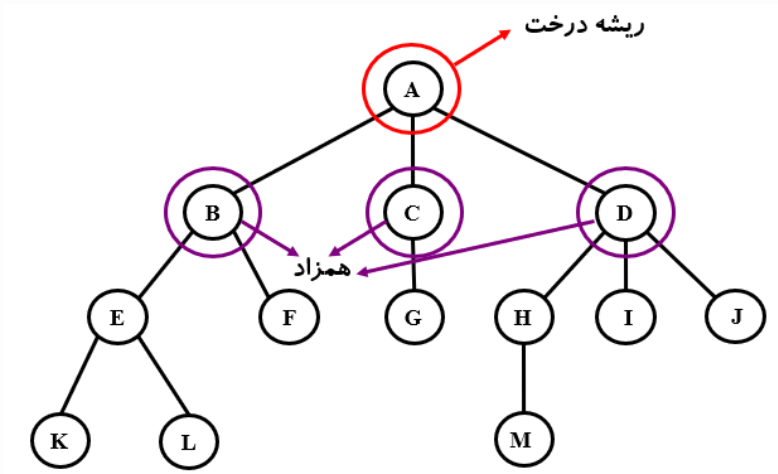
صف اولویت: در صف اولویت ما همیشه عنصری را خارج میکنیم که از همه بزرگتر است.

- ❑ در درخت max-heap یافتن عنصر ماکزیمم نیاز به جستجو ندارد زیرا ریشه ماکزیمم است و تنها پیدا کردن محل مناسب برای گره آخر دارای پیچیدگی o(log(n)) است زیرا در هر مرحله یا سمت چپ میریم یا سمت راست(سمت فرزند بزرگتر) و حداکثر به اندازه عمق درخت حرکت خواهیم کرد.
- ❑ همچنین افزودن عنصر به درخت نیز به اندازه عمق درخت حرکت میکند (از پایین به سمت بالا) و پیچیدگی آن نیز o(log(n)) است

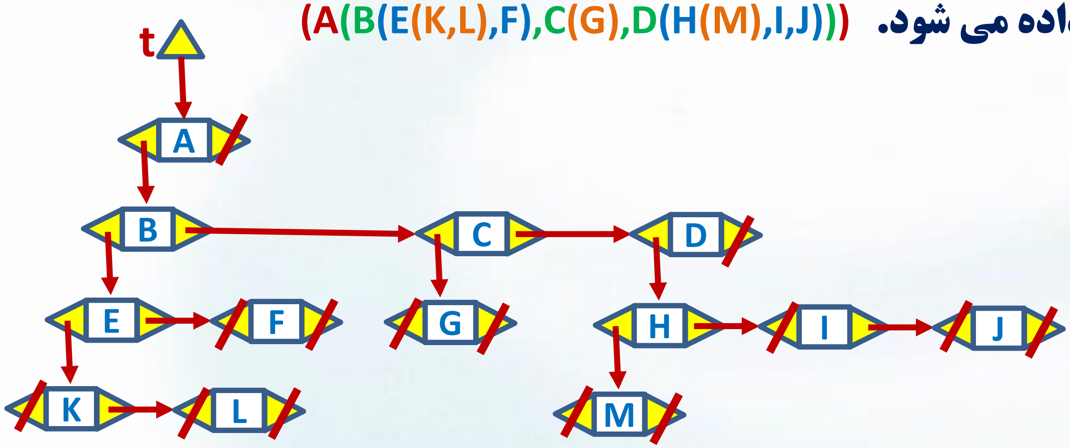
	پیچیدگی افزودن عنصر	پیچیدگی حذف ماکزیمم	پیچیدگی کل
آرایه یا لیست پیوندی نامرتب	o(1)	o(n)	o(n)
آرایه یا لیست پیوندی مرتب	o(n)	o(1)	o(n)
درخت heap	o(log(n))	o(log(n))	o(log(n))

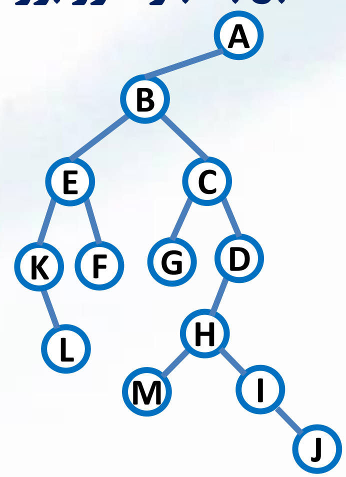
- ❑ یکی دیگر از کاربرد های درخت heap، استفاده در مرتب سازی آرایه ها می باشد. در فصل مرتب سازی در خصوص ایده مربوط به مرتب سازی با درخت heap صحبت خواهیم کرد.

درخت عمومی (غیر باینری)



- برای نمایش درخت عمومی (درخت با درجه بالاتر از 2) میتوان از ساختار لیست پیوندی درختان دودویی استفاده کرد.
- در این ساختار اولین فرزند هر گره در لینک left و همزاد هر گره در لینک right قرار میگیرد.
- درخت غیر باینری مقابل با **روش فرزند چپ–همزاد راست** به فرمهای زیر نمایش داده شده است(هر دو شکل معادل هم هستند)
- همچنین درخت مقابل با عبارت رو برو نمایش داده می شود.   (A(B(E(K,L),F),C(G),D(H(M),I,J)))





میتوان برای درخت M-tree گره هایی با M لینک برای فرزندان گره داشت اما در درختی با n گره فقط n-1 لینک (بجز ریشه که گره ای به آن لینک نمی دهد) استفاده می شوند و Mn – n +1 لینک ها NULL خواهند بود. مثلا برای درختی 4-tree تقریبا $\frac{3}{4}$ لینک ها خالی خواهد بود.

مباحث بیشتر از درختان (مطالعه آزاد)

- **درخت انتخاب:**
  - برای ادغام نمودن n لیست مرتب لازم است مینیمم بین n لیست به آرایه اضافه شود و عدد بعدی لیست جهت مقایسه جایگزین شود. یافتن مینیمم n عدد در حالت عادی به n مقایسه نیاز دارد. اما میتوان با استفاده از درخت min-heap تعداد مقایسه را به log(n) کاهش داد. کافی است ابتدا بر اساس اولین عنصر هر لیست درخت min-heap را ساخت و با حذف مینیمم، عدد بعدی آن لیست به درخت اضافه شود.

- **کد هافمن:**
  - در حالت عادی برای کد گذاری حروف انگلیسی لازم است 5 بیت برای هر حرف در نظر گرفت (تعداد حروف بیت 16 تا 32 می باشد لذا 5 بیت لازم است).
  - الگوریتم هافمن جهت ایجاد کد های با طول متفاوت برای هر حرف می باشد با این شرط که هیچ کد با طول کوچکتری پیش شماره کد های با طول بزرگتر نباشد و همچنین کد حروف پر کاربرد طول کمتری داشته باشند.(چیزی شبیه شماره تلفن های ضروری که سه رقمی هستند و هیچ تلفنی پیش شماره 110 یا 118 ندارد)

- **جنگل:**
  - به مجموعه چند درخت جنگل گفته می شود.

تهیه و تنظیم : محمد نعیمی