ساختمان داده ها

دانشگاه صنعتی نوشیروانی بابل

دکتر حسام عمران پور

طراحان اسلاید:
زهرا ریحانیان و دانیال علیزاده

حل تمرین :
علی باقری

لینک کانال تلگرام اطلاع رسانی و حل تمرین:
t.me/ds_nit_4011

فصل ۸
فرهنگ داده ها

تعریف

به ساختمان داده ای که با اهداف زیر ساخته شود، **فرهنگ داده** گویند :

1. درج: افزودن عنصر جدید به ساختمان
2. حذف: حذف یک عنصر
3. جستجو: k (کلید) به عنوان ورودی داده می شود و در ساختمان، x متناظر پیدا می شود.

انواع فرهنگ داده

- درخت جستجوی دودویی
- درخت AVL
- درخت قرمز – سیاه
- درخت ۳ -۲
- درخت B
- جدول درهم (Hash Table)

فرهنگ داده زبان انگلیسی به فارسی

کلید ← **کلمات انگلیسی (k)** ← از روی کلمات میتوان یک عدد بدست آورد.

برای هر حرفی یک عدد در نظر بگیریم چون ۲۶ حرف انگلیسی داریم یک عدد در مبنای ۲۶ بدست می آید.

داده متناظر ← **ترجمه فارسی (x)**

فرهنگ داده برای ذخیره سازی داده ها به همراه کلید استفاده می شود.کلید معمولا یک عدد یکتاست، میتواند از روی داده دیگری ساخته شود.

توابع ADT

توابع از دیدگاه ADT

Create (**ref** D : Dictionary)

Empty (D : Dictionary) : **Boolean**

MakeNull (**ref** D : Dictionary)

Insert (**ref** D : Dictionary ; k : **KeyType** ; x : **DataType**)

Delete (**ref** D : Dictionary ; Node : **NodeType**)

Search (D : Dictionary ; k : **KeyType**) : **NodeType**

ادامه توابع از دیدگاه ADT

بزرگترین کلید :

Max (D : Dictionary) : NodeType

کوچکترین کلید :

Min (D : Dictionary) : NodeType

عنصر پیشین :

PredeCessor (D : Dictionary ; Node : NodeType) : NodeType

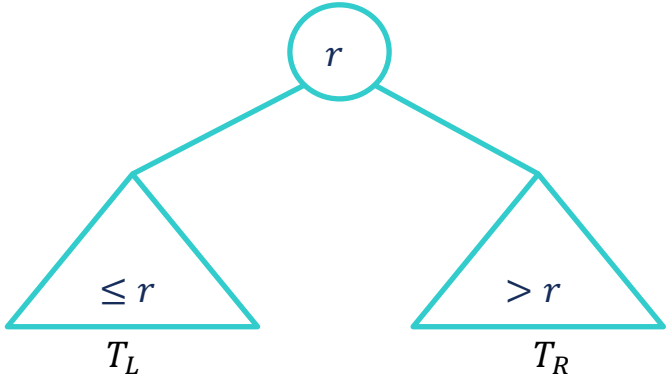
عنصر بعدی :

SucCessor (D : Dictionary ; Node : NodeType) : NodeType

درخت جستجوی دودویی

درخت جستجوی دودویی

درختی دودویی است.

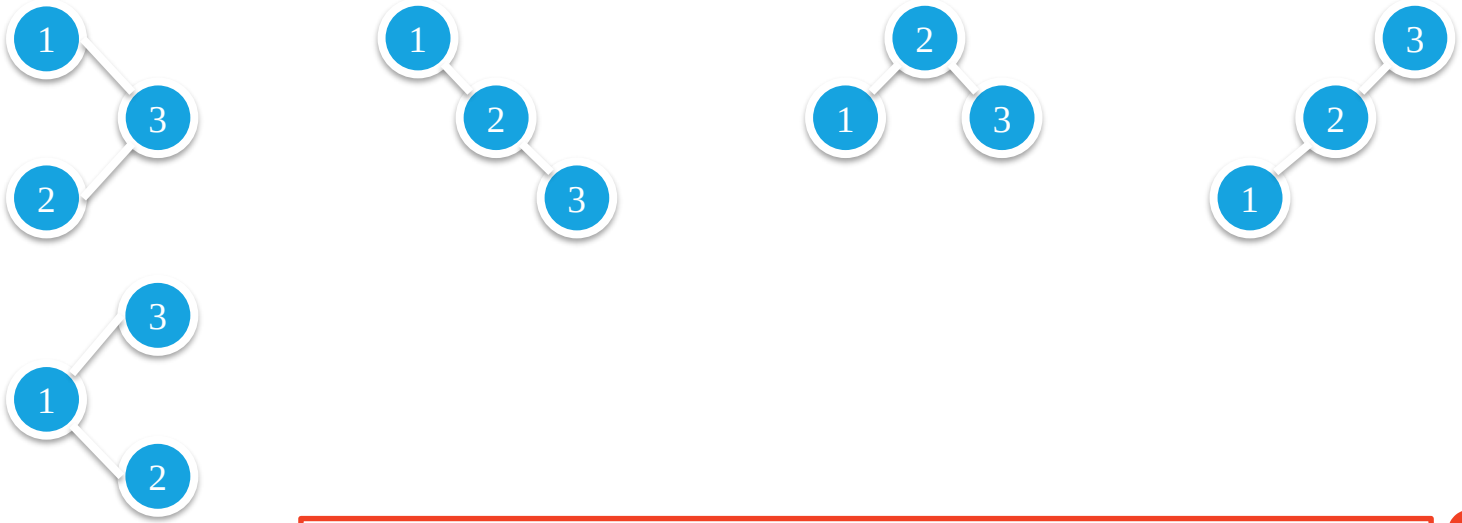


۱. هر عنصر دقیقا یک کلید دارد.

۲. کلید عناصر زیردرخت سمت چپ کوچک تر مساوی کلید ریشه (پدر) است.

۳. کلید عناصر زیردرخت سمت راست بزرگ تر از کلید ریشه (پدر) است.

درخت جستجوی دودویی که میتوان با اعداد ۱ و ۲ و ۳ ساخت را رسم کنید.



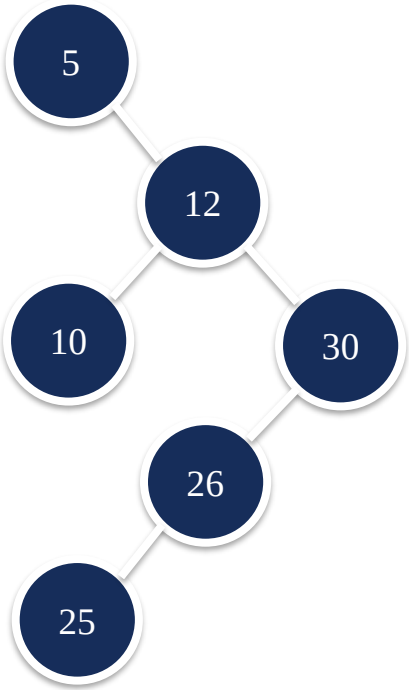
با n کلید، تعداد درخت های جستجوی دودویی که میتوان ساخت برابر است با :

$$C_n = \frac{1}{n+1}\binom{2n}{n} \sim \frac{4^n}{\sqrt{\pi}\, n^{\frac{3}{2}}}$$

پیمایش preorder درخت جستجوی دودویی داده شده است. درخت را رسم کنید.

?

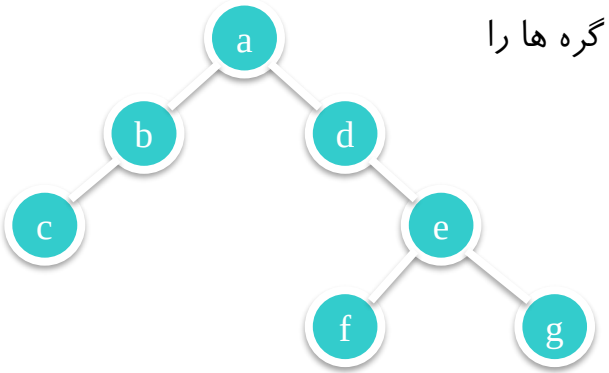
5 , 12 , 10 , 30 , 26 , 25



? پیمایش های pre , post یک درخت دودویی داده شده است. با فرض ذخیره سازی در آرایه (ریشه در خانه ۱ و فرزندان گره اندیسی در 2i , 2i+1) حداکثر تعداد خانه های بلا استفاده قبل از محل آخرین گره در آرایه چند است؟

pre: abcdefg

post: cbfgeda



برای اینکه بیشترین فضای خالی را داشته باشیم، باید تا حد امکان گره ها را در راست ترین جای ممکن قرار دهیم

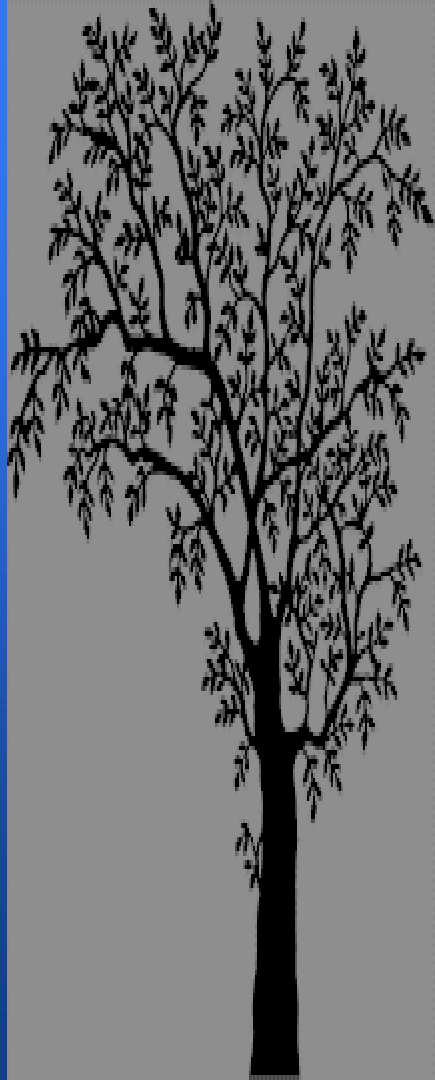
6 + 2 = تا 8

a	b	d	c	*	*	e	*	*	*	*	*	*	f	g
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

پیاده سازی درخت جستجوی دودویی (BST)

```
Type BST = Record
{
        RC, LC : ^CellType;
        Key : KeyType;
        Data : DataType;
}

Type BST : ^CellType
Type NodeType : ^CellType
```



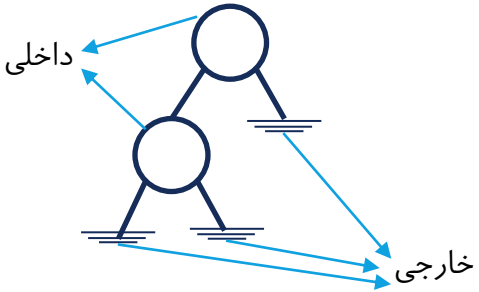
جستجو در درخت جستجوی دودویی

طول مسیر داخلی: مجموع طول مسیرهای گره ریشه تا گره داخلی

طول مسیر خارجی: مجموع طول مسیرهای گره ریشه تا گره خارجی

گره های Null ← خارجی

گره های غیر Null ← داخلی



شبه کد جستجو

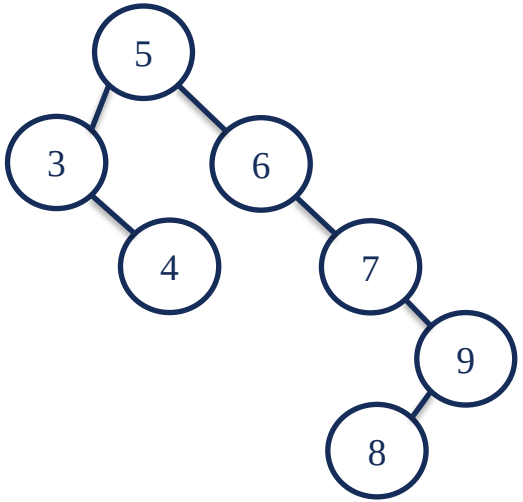
```
Search (T : BST ; k : KeyType) : NodeType
{
        if   T == NULL  then
             return NULL;
        if   T^.Key == k  then
             return  T;
        if   k < T^.Key  then
             return Search(T^.LC, k);
        else
             return Search(T^.RC, k);
}
```

میانگین تعداد مقایسه ها برای جستجوی موفق

اگر احتمال دسترسی به همه ی عناصر ها یکسان باشد :

$$\overline{C} = \frac{1}{n}\sum_{i=1}^{n} d_i$$

d_i عمق گره i_ام



$$\overline{C} = \frac{1}{7}\,(1+2+2+3+3+4+5) = \frac{20}{7}$$

اگر احتمال دسترسی به عناصر متفاوت باشد :

$$\overline{C} = \sum_{i=1}^{n} d_i p_i$$

d_i عمق گره i_ام

p_i احتمال گره i_ام

تعداد مقایسه ها با متوازن تر شدن درخت، کاهش میابد.

یک روش ← استفاده از میانه به عنوان ریشه و سپس همین عمل به صورت بازگشتی بر روی زیردرخت چپ و راست انجام شود.
البته لزومی ندارد که برای بهینه شدن حتما متوازن باشد یا عنصر با بیشترین احتمال در ریشه قرار بگیرد.

پیاده سازی توابع

```
Max (T : BST) : NodeType
{
        if   T == NULL  then
             return NULL;
        else if   T^.RC == NULL   then
             return T;
        else
             return Max (T^.RC);
}
```

راست ترین گره = بزرگترین کلید

چپ ترین گره = کوچکترین کلید

```
Min (T : BST) : NodeType
{
        if   T == NULL  then
             return NULL;
        else if   T^.LC == NULL   then
             return T;
        else
             return Min (T^.LC);
}
```

یافتن پدر یک گره در درخت جستجوی دودویی

```
Parent (T : BST ; Node : NodeType) : NodeType
{
        if   T == NULL or  T == Node then
                return NULL;
        if   T^.LC  ==  Node  or  T^.RC  ==  Node   then
                return T;
        else if  Node^.Data  < T^.Data   then
                return Parent (T^.LC, Node);
        else
                return Parent (T^.RC, Node);
}
```

یافتن عنصر بعدی در درخت جستجوی دودویی

```
SUCCESSOR (T : BST ; Node : NodeType) : NodeType
{
        if  Node^.RC  !=  NULL  then
                return  Min(Node^.RC);
        else {
                parent = Parent(T, Node);
                while  parent != NULL && parent^.RC  == Node
                {
                        Node = parent;
                        parent = Parent(T, Node);
                }
                return parent;
        }
}
```

یافتن عنصر پیشین در درخت جستجوی دودویی

```
PREDECESSOR (T : BST ; Node : NodeType) : NodeType
{
        if  Node^.LC  !=  NULL  then
                return  Max(Node^.LC);
        else {
                parent = Parent(T, Node);
                while  parent != NULL && parent^.LC  == Node
                {
                        Node = parent;
                        parent = Parent(T, Node);
                }
                return parent;
        }
}
```

درج در درخت جستجوی دودویی

```
Insert (ref T : BST ; k : KeyType ; x : DataType)
{
        if   T == NULL
        {
                new(T)
                T^.Key = k;
                T^.Data = x;
                T^.RC = T^.LC = NULL;
        }
        else if  k <= T^.Key
                Insert (T^.LC, k, x);
        else
                Insert (T^.RC, k, x);
}
```

حذف از درخت جستجوی دودویی

برای حذف یک عنصر از درخت جستجوی دودویی سه حالت وجود دارد :

۱. گره مورد نظر x برگ است. در این حالت تنها کافی است که پدر گره ی x یافت شده و مقدار اشاره گر فرزند چپ یا راست آن که به گره ی x اشاره می کند برابر NULL شود. مرتبه ی پیچیدگی زمانی در این حالت به دلیل نیاز به یافتن پدر گره ی x برابر O(h) است.

۲. گره ی مورد نظر x تنها یک فرزند (چپ یا راست) دارد. در این حالت باید فرزند گره ی x جایگزین آن شود. بنابراین ، پس از یافتن پدر این گره ، کافی است مقدار اشاره گر فرزند چپ یا راست که به گره ی x اشاره می کنند ، به تنها فرزند این گره اشاره داده شود. مرتبه ی پیچیدگی زمانی در این حالت نیز ، به دلیل نیاز به یافتن پدر گره ی x برابر O(h) است.

۳. گره ی مورد نظر x دارای دو فرزند (چپ و راست) است . در این حالت ، گره ی پیشین x جایگزین این گره می شود. چون در این حالت گره ی x دارای فرزند چپ است ، پس عنصر پیشین x در زیر درخت چپ آن قرار داشته و راست ترین گره ی زیردرخت چپ آن خواهد بود.

```
Delete (ret T : BST ; Node : NodeType)
{
        p : NodeType
        x : DataType
        k : KeyType
        if  Node^.LC == NULL  or  Node^.RC == NULL
        {
                p = Parent (T, Node)
                if  p == NULL
                {
                    if  Node^.RC  != NULL
                        T = Node^.RC
                    else
                        T = Node^.LC
                }
                else if  p^.RC == NULL
                {
                    if  Node^.RC != NULL
                        p^.RC = Node^.RC
                    else
                        p^.RC = Node^.LC
                }
        …
```

```
        …
                else
                {
                    if  Node^RC != NULL
                        p^.LC = Node^.RC
                    else
                        p^.LC = Node^.LC
                }
                dispose(Node)
        }
        else
        {
                [x, k] = Delete_Right (Node^.LC)
                Node^.Data = x
                Node^.key = k
        }
}
```

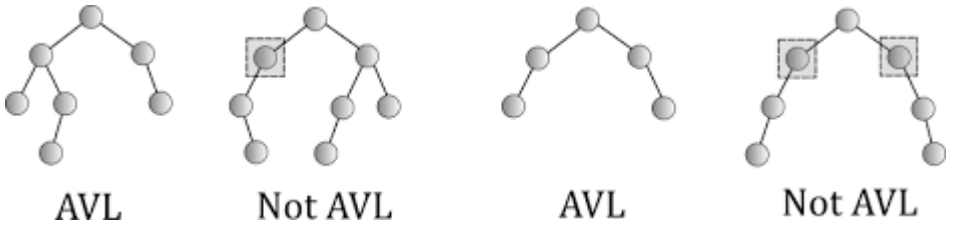
```
Delete_Right (ref T : BST) : [DataType, KeyType]
{
        N : NodeType
        x : DataType
        k : KeyType
        if  T^.RC == NULL
        {
                x = T^.Data
                k = T^.Key
                Delete(T, T)
                return [x, k]
        }
        else
                return Delete_Right (T^.RC)
}
```

درخت های خود متوازن کننده

Self-balancing binary search tree

درخت AVL

درخت جستجوی دودویی که اختلاف ارتفاع زیردرخت های چپ و راست هر گره **حداکثر برابر ۱** باشد.



ارتفاع درخت avl

$\theta(\log n)$

ساختمان داده AVL

```
Type CellType = Record
{
    key : keyType;
    Data : DataType;
    LC : ^CellType;
    RC : ^CellType;
    Height : int;
}
Type AVL = ^CellType;
```

تعریف بازگشتی درخت AVL

پایه : یک درخت خالی (بدون عنصر) درخت ای وی ال است.

بازگشت : درخت جستجوی دودویی T ، درخت ای وی ال است اگر و فقط اگر زیر درخت های چپ و راست (T_L و T_R) ریشه ی T ، درخت های ای وی ال باشند و اختلاف ارتفاع زیر درخت چپ (h_L) و زیر درخت راست (h_R) ریشه ی T ، حداکثر برابر ۱ باشد ، یعنی رابطه زیر برقرار باشد :

$$|h_L - h_R| \leq 1$$

الگوریتم جستجو در درخت AVL

الگوریتم جستجوی یک عنصر در درخت ای وی ال، دقیقا مانند الگوریتم جستجوی عنصر در درخت جستجوی دودویی است.

تنها تفاوت در مرتبه ی پیچیدگی زمانی بدترین حالت

پیچیدگی زمانی بدترین حالت در درخت ای وی ال :

$\theta(\log n)$

انواع چرخش

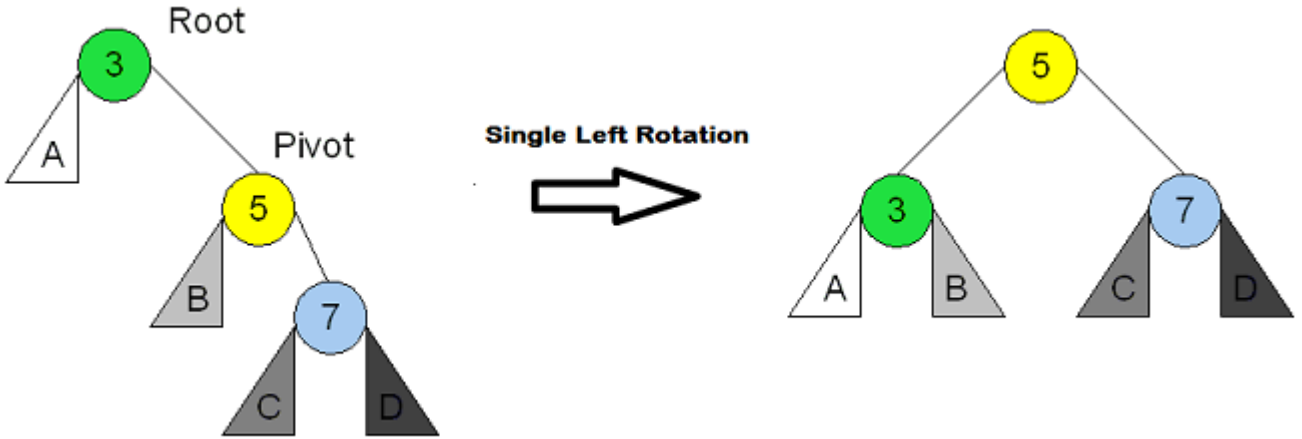
چرخش به راست

چرخش به چپ

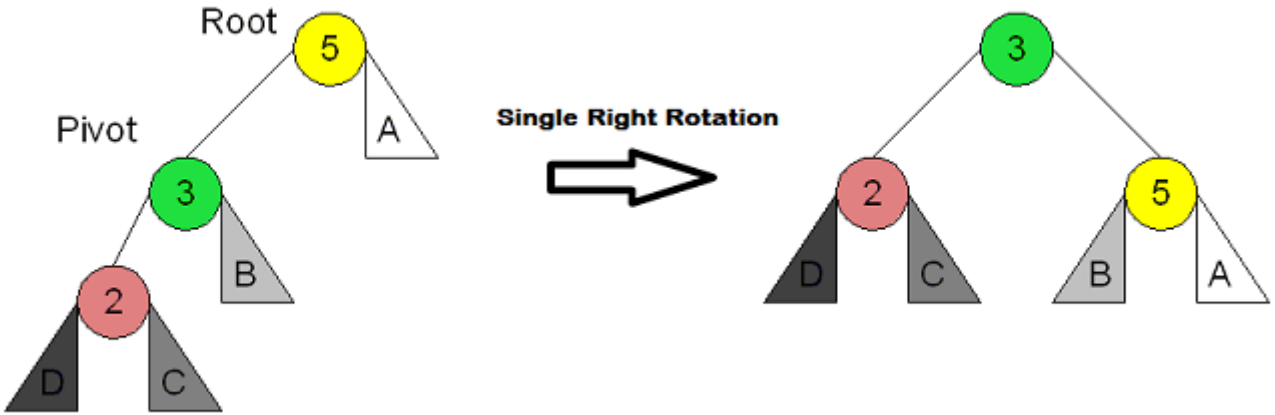
چرخش راست دوگانه

چرخش چپ دوگانه

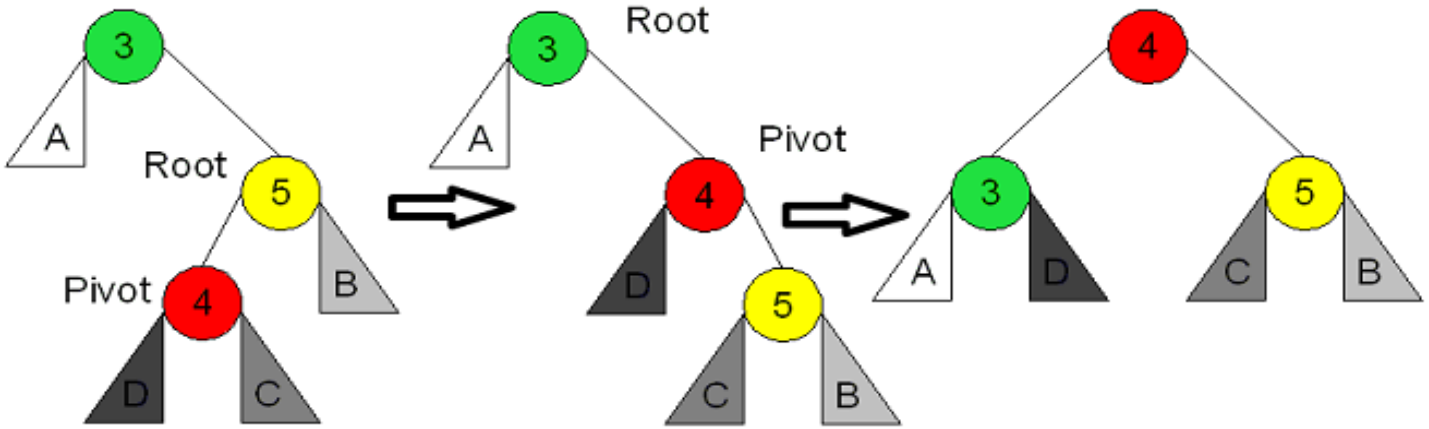
چرخش به چپ



چرخش به راست

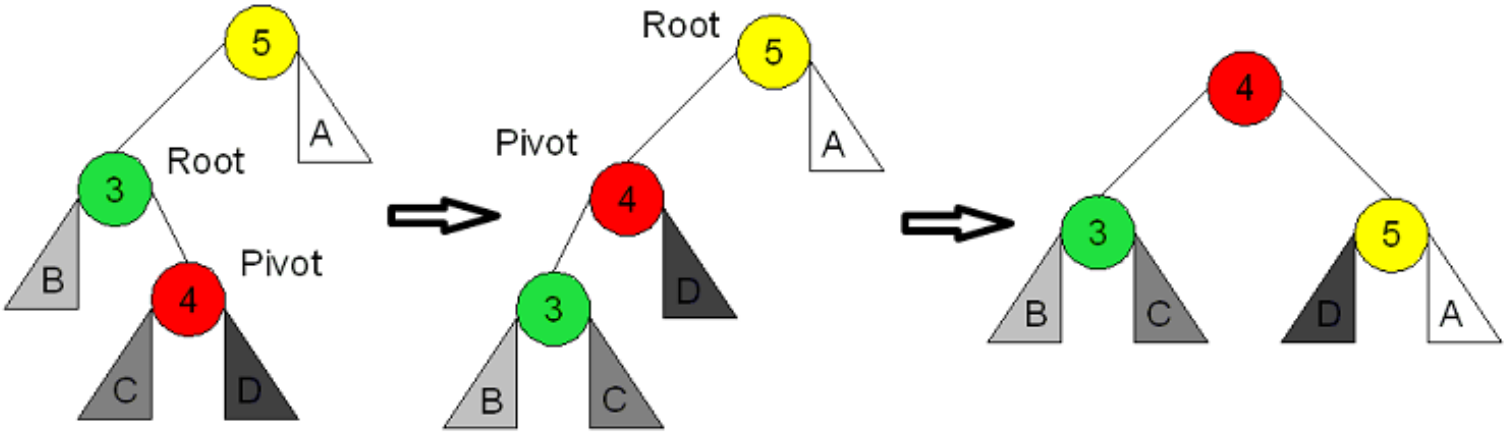


چرخش دوگانه به چپ



* ابتدا یک چرخش به راست سپس چرخش به چپ

چرخش دوگانه به راست



* ابتدا یک چرخش به چپ سپس چرخش به راست

الگوریتم چرخش به چپ

```
LEFT_ROTATE (ref T : BST; x : NodeType)
{
    var y : NodeType;
    var p : NodeType;
    y = x^.RC;
    x^.RC = y^.LC;
    p = Parent (T, x);
    if p == NULL then
        T = y;
    else if x == p^.LC then
        p^.LC = y;
    else
        p^.RC = y;
    y^.LC = x;
}
```

درج در درخت AVL

ابتدا همانند درج در درخت جستجوی دودویی ، یک عنصر در درخت ای وی ال درج می شود و سپس ارتفاع پدر و اجداد این گره یک واحد اضافه می شود.

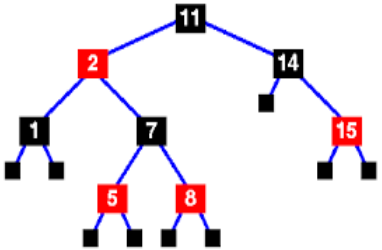
ممکن است پدر یا یکی از اجداد گره درج شده اختلاف ارتفاع فرزندان چپ و راست آنها برابر ۲ شود. لذا باید چرخش روی این اجداد که تعدادشان حداکثر به اندازه ارتفاع درخت $O(\log n)$ است انجام شود. یکی از ۴ حالت زیر را داریم :

1. درج عنصر جدید در زیردرخت چپ فرزند چپ گره ی غیرمتوازن x انجام شده است.
2. درج عنصر جدید در زیردرخت راست فرزند چپ گره ی غیرمتوازن x انجام شده است.
3. درج عنصر جدید در زیردرخت چپ فرزند راست گره ی غیرمتوازن x انجام شده است.
4. درج عنصر جدید در زیردرخت راست فرزند راست گره ی غیرمتوازن x انجام شده است.

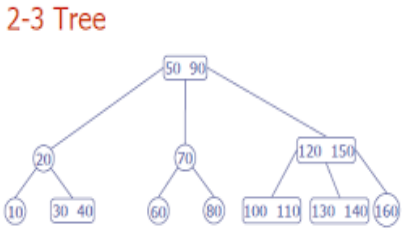
حذف از درخت AVL

برای حذف یک عنصر از درخت ای وی ال ، ابتدا باید عنصر را با استفاده از عمل حذف درخت جستجوی دودویی حذف کرده و سپس توازن گره های غیر متوازن شده را اصلاح کرد. عمل حذف یک عنصر از درخت ای وی ال پیچیده تر از عمل درج بوده و تعداد حالت های زیادی دارد. در عمل حذف ، بر خلاف عمل درج، تعداد گره هایی که نیاز به چرخش پیدا می کنند می تواند حداکثر از مرتبه ی ارتفاع درخت باشد. بنابراین ، مرتبه ی پیچیدگی زمانی عمل حذف نیز برابر $O(h)$ یا همان $O(\log n)$ است.

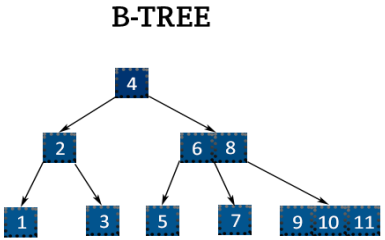
دیگر درخت های خود متوازن کننده



درخت قرمز-سیاه



درخت ۲-۳



درخت بی

جدول های درهم

Hash Table

جدول های درهم

- هدف ما همانند سایر ساختمان داده های Dictionary پایین آوردن مرتبه زمانی می باشد.
در اینجا هدف رساندن مرتبه زمانی میانگین به O(1) است.

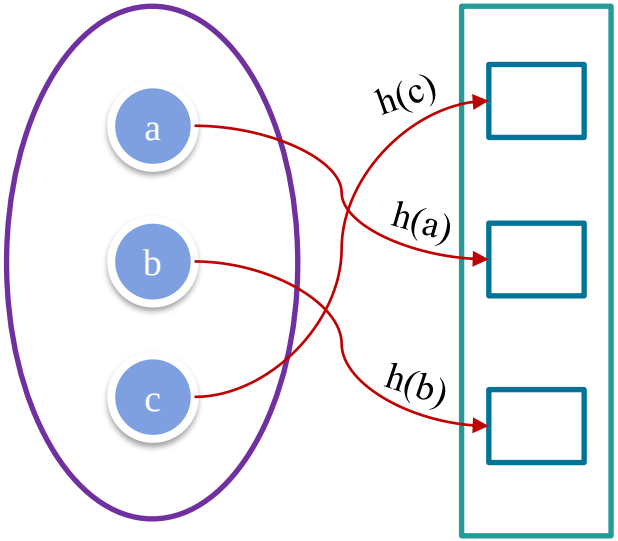
- جدول درهم، یک آرایه به اندازه B است. عناصر مجموعه ی داده ها با استفاده از یک تابع نگاشت به نام تابع درهم ساز، به خانه های مختلف این آرایه نگاشت می شوند. در واقع دامنه ی تابع درهم ساز برابر دامنه ی کلیدهای مجموعه ی داده ها و برد آن برابر اعداد صحیح صفر تا B – 1 است.

- معمولا تابع درهم ساز h(x) طوری انتخاب می شود که بتواند از روی کلید عنصر ورودی، اندیس خانه ی آن در آرایه را با مرتبه O(1) محاسبه کند.

به عمل نگاشت یک عنصر به اندیس متناظر ان با استفاده از تابع h(x)، عمل **درهم سازی** گفته می شود.

یک تابع درهم ساز ایده آل دارای دو شرط زیر است :

۱. **یک به یک باشد.**
۲. **مرتبه پیچیدگی زمانی O(1)**



برخورد (تصادم)

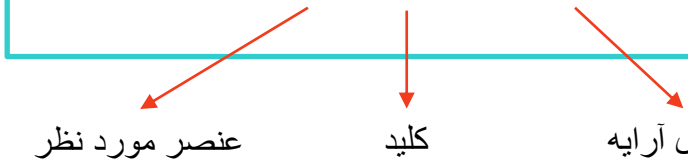
اگر دو آرایه به یک خانه نگاشت شوند، به حالت پیش آمده برخورد گویند.

برای رفع برخورد دو استراتژی کلی وجود دارد :

۱. **درهم سازی باز** : در این نوع درهم سازی، در هر خانه ی جدول می تواند بیش از دو عنصر ذخیره گردد.

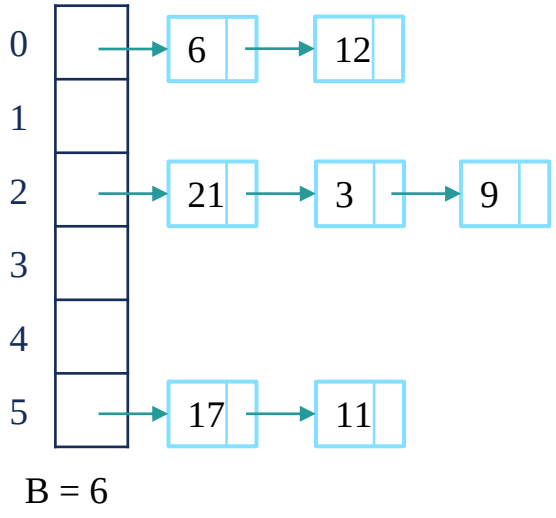
۲. **درهم سازی بسته** : در این نوع درهم سازی، در هر خانه جدول حداکثر یک عنصر میتواند قرار گیرد.

Hash : تابع درهم ساز $\leftarrow h(x) = k \ mode \ B$



درهم سازی باز

هر عنصر از آرایه به یک لیست پیوندی اشاره می کند. برای درج در سلول i از آرایه، عنصر جدید به ابتدای لیست پیوندی اضافه می شود.



فضای اشغال شده

$\theta(B+n)$

مرتبه زمانی

درج O(1)
حذف O(n)
جستجو O(n)

B مناسب O(1)

حافظه ی اشغال شده به طور دقیق

$\alpha \times \beta + (\alpha + \beta + \theta) \times n$

فضای مورد استفاده اشاره گر : α
فضای مورد استفاده کلید : β
فضای مورد استفاده داده : θ

درهم سازی بسته

در این روش هر عنصر در یک سلول از آرایه قرار می گیرد.
(یک کلید و یک داده در هر خانه از آرایه داریم)

در ادامه شیوه های یافتن فضای خالی برای قرار دادن یک عنصر ورودی در آرایه را خواهیم دید.

۱. روش کاوش خطی Linear probing :

$$h'(x) = (h(x) + i)\ mod\ B$$

$$i = 0, 1, 2, \dots, B - 1$$

۲. روش کاوش درجه۲ Quadratic probing :

$$h'(x) = (h(x) + c_1 * i^2 + c_2 * i)\ mod\ B$$

یا

$$h'(x) = (h(x) + i^2)\ mod\ B$$

$$i = 0, 1, 2, \dots, B - 1$$

۳. درهم سازی دوگانه(مضاعف) Double hashing :

$$h'(x) = \big(h_1(x) + i * h_2(x)\big)\ mod\ \ B$$

$$i = 0, 1, 2, \dots, B - 1$$

? با درهم سازی بسته و کاوش خطی (Linear probing) اطلاعات زیر را در آرایه قرار دهید.

B = 8

X	Key
a	2
b	5
c	8
d	10
e	18
f	13
g	4
h	16

0	c
1	h
2	a
3	d
4	e
5	b
6	f
7	g

Linear probing

0	c
1	f
2	a
3	d
4	g
5	b
6	e
7	

Quadratic probing

? برای رفع تصادم از روش کاوش خطی استفاده می شود. اگر تابع درهم ساز و نتیجه حاصل از درج به صورت زیر باشد، چند ترتیب برای ورود عناصر به این جدول می توان تصور نمود؟

x	h(x)
a	3
b	2
c	4
d	2
e	3
f	2

0	a
1	b
2	d
3	f
4	c
5	e

1) c d f e a b
2) d c f e a b
3) d f c e a b

پایان