درخت‌ها (تعریف)

یک گراف هم‌بند و بدون دور («درخت آزاد» (free tree)

یک چنین درختی با n راس دقیقاً دارای $n-1$ یال است.

لم: اگر تعداد یال‌ها (E) تعداد رأس‌ها (V) باشد، داریم $E = V - ۱$

اثبات: با استقرا: برای $n = ۱$ بدیهی است که $E = ۰$،

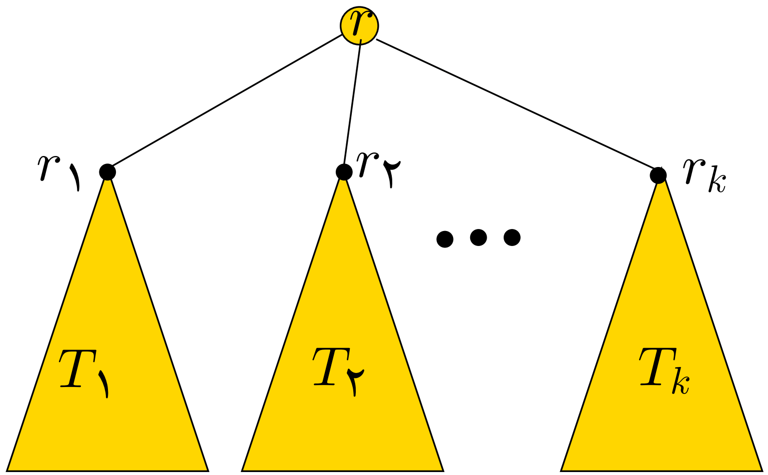
فرض استقراء: برای $V = k$ داریم $E = k - ۱$

حکم استقراء: اگر $V = k + ۱$

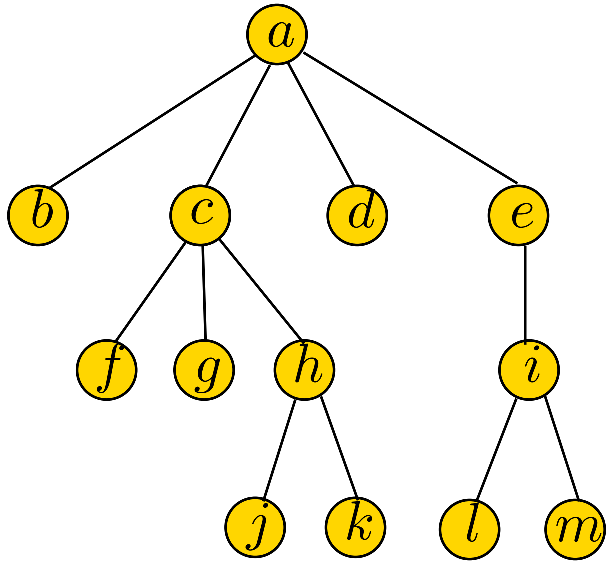
اگر بر روی یال‌های یک درخت آزاد جهت قرار دهیم به‌طوری‌که درجه‌ی ورودی حداکثر یک باشد، به آن یک درخت جهت‌دار با رابطه‌ی «پدر فرزندی» بین رئوس گوییم.

- هر عنصر غیر از ریشه دقیقاً یک پدر دارد
- تعداد فرزندان نامشخص

تعریف درخت به صـــورت بازگشتی



- یک گره به‌تنهایی یک درخت است.
- از k درخت مستقل T_1 تا T_k با ریشه‌های r_1 تا r_k یک درخت بزرگ‌تر T را با ریشه‌ی r بسازیم به‌طوری‌که r پدر r_1 تا r_k باشد.
- در این‌صورت $T_1, ..., T_k$ «زیردرخت‌های» T خواهند بود.


a
b
c
d
e
f
g
h
i
j
k
l
m

تعاریف اولیه در درخت‌های پدر فرزندی

ریشه (root) در درخت جهت‌دار

گره‌ای است که دارای پدر نیست، این گره در هر درخت جهت‌دار یکتا است.

برگ (leaf)

گره بدون فرزند.

برادر (هم‌نیا) (sibling)

گره‌هایی که یک پدر دارند، برادر هم هستند.

گره داخلی (interior node)

گره غیر برگ.

ارتفاع گره v (height)

طول بزرگ‌ترین مسیر از v به برگ w به‌طوری‌که w گره‌ای از زیردرختی به ریشه‌ی v باشد.

ارتفاع درخت

ارتفاع ریشه.

سطح (عمق) یک گره (depth - level)

برابر است با طول مسیری از ریشه درخت به آن گره.

درخت k تایی (k-ary tree)

بیشینه‌ی تعداد فرزندان هر گره یک درخت k باشـد.

درخت k تایی کامل (complete k-ary tree)

درختی است که درآن تعداد فرزندان هر گره برابر k یا صفر (فقط برای برگ) است.

درخت متوازن (balanced tree)

درختی که‌سطح برگ‌های آن حداکثر یک واحد باهم اختلاف داشته باشد.

درخت کاملاً متوازن (completely balanced tree)

درختی که سطـح برگ‌های آن یکسان باشـد .

درخت مرتب (ordered tree)

درختی است که درآن ترتیب فرزندان هر گره مشخص است.

درخت برچسب‌دار (labeled tree)

درختی است که هر گره آن یک برچسب دارد.

درخت دودویی (binary tree)

درخت مرتبی است که هر عنصرآن حداکثر دارای دو فرزند به نام‌های فرزند چپ و راست می باشد. اگر یک گره فقط یک فرزند داشته باشد باید مشخص شود که فرزند چپ است یا راست.

اولاد (نوادگان) یک گره v (descendents)

کلیه‌ی گره‌های موجود در زیردرختی به ریشه v را اولاد v می‌گوییم. با این تعریف هر گره یکی از اولاد خودش است.

اجداد (نیاکان) یک گره (ancestors)

کلیه‌ی گره‌های موجود در مسیری از ریشه به یک گره را اجداد آن گره گوییم. بنابراین هر عنصری جزو اجداد خودش است.

اولاد واقعی (proper descendents)

تمام اولاد یک گره به‌غیر از خود آن گره اولاد واقعی به حساب می آیند.

اجداد واقعی (proper ancestors)

تمام اجداد یک راس به غیر از خود آن راس اجداد واقعی هستند.

زیردرخت (subtree)

یک گره با همه‌ی اولاد واقعی‌اش

درخت پر (full tree)

درخت کامل و کاملاً متوازن

جنگل (forest)

تعدادی درخت!

مسئله

در درختی با n گره که تعداد فرزندان هر گره صفر یا k باشد، تعداد برگ‌های آن چندتاست؟

حل:

- B **تعداد برگ‌ها**
- n **تعداد کل گره‌ها**
- **نعداد یال‌ها** $k * (n-B) = n-1$.
- **پس** $n-B=(n-1)/k$ **و** $B=n-(n-1)/k$ **یا**

$$B=[(k-1)n+1]/k$$

- **یعنی** $(k-1)n+1$ **باید بر** k **بخش‌پذیر باشد.**

پیمایش درخت‌ها

فرض: درخت T **با ریشه** r **و** k **زیر درخت** $T_۱, ..., T_k$

پیمایش پیش‌ترتیب (preorder):

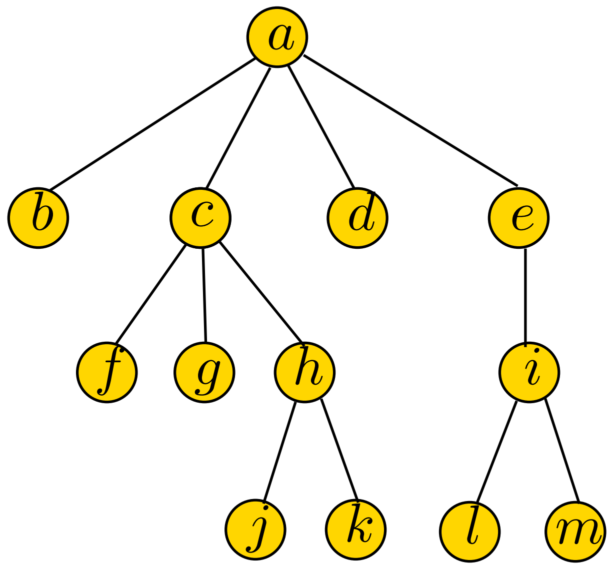
$$Preorder(T) = r, Preorder(T_۱), Preorder(T_۲), \ldots, Preorder(T_k)$$

پیمایش میان‌ترتیب (inorder):

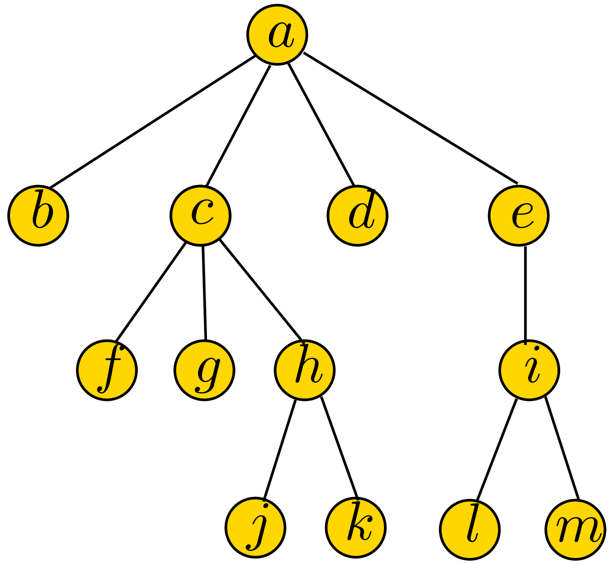
$$Inorder(T) = Inorder(T_۱), r, Inorder(T_۲), \ldots, Inorder(T_k)$$

پیمایش پس‌ترتیب (postorder):

$$Postorder(T) = Postorder(T_۱), Postorder(T_۲), \ldots, Postorder(T_k), r$$

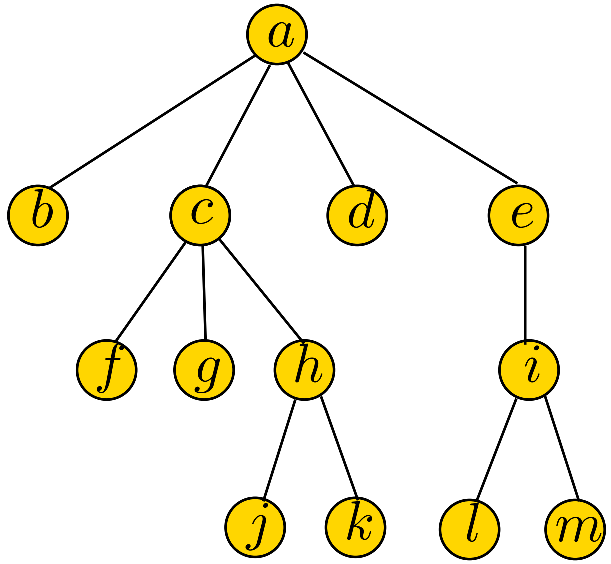


Preorder(T): $a, b, c, f, g, h, j, k, d, e, i, l, m$



Preorder(T): $a, b, c, f, g, h, j, k, d, e, i, l, m$

Inorder(T): $b, a, f, c, g, j, h, k, d, l, i, m, e$

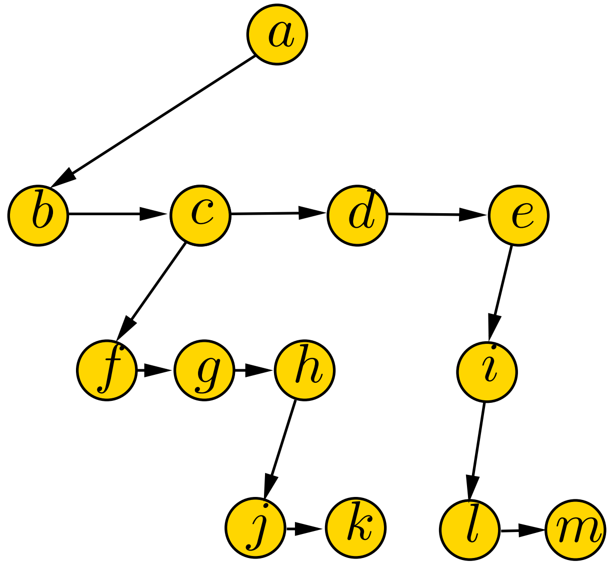


Preorder(T): $a, b, c, f, g, h, j, k, d, e, i, l, m$

Inorder(T): $b, a, f, c, g, j, h, k, d, l, i, m, e$

Postorder(T): $b, f, g, j, k, h, c, d, l, m, i, e, a$

درخت دودویی معادل



اعمال بر روی درخت

- $\textsc{Create}(T)$: درخت تهی T با ویژگی‌های تعریف‌شده را ایجاد کن. ورودی: هیچ، خروجی: درخت
- $\textsc{Root}(T)$: ریشه‌ی درخت T را برمی‌گرداند
  ورودی: درخت، خروجی: گره
- $\textsc{Parent}(T, v)$: پدر گره‌ی v را در درخت T برمی‌گرداند
  ورودی: درخت و گره، خروجی: گره یا null
- $\textsc{Left-Most-Child}(T, v)$: اولین فرزند گره‌ی v را در درخت T برمی‌گرداند
  ورودی: درخت و گره، خروجی: گره یا null
- $\textsc{Right-Sibling}(T, v)$: برادر سمت راست v را در درخت T برمی‌گرداند

ورودی: درخت و گره، خروجی: گره یا null

- $\textsc{Size}(T)$: **تعداد عناصر موجود در درخت**
  ورودی: درخت، خروجی: یک عدد صحیح
- $\textsc{isEmpty}(T)$: **مشخص می‌کند که آیا درخت خالی است**
  ورودی: درخت، خروجی: درست یا نادرست
- $\textsc{Element}(T, n)$: **برچسب عنصر در گره** n **را برمی‌گرداند**
  ورودی: درخت و گره، خروجی: برچسب

پیمایش درخت T از گره p

```
PREORDER(T, p)
1  if p = null
2    then return
3  ELEMENT(T, p)
4  p ← LEFT-MOST-CHILD(T, p)
5  while p ≠ null
6        do PREORDER(T, p)
7           p ← RIGHT-SIBLING(T, p)
```

```
INORDER(T, p)
if p = null
  then return
n ← LEFT-MOST-CHILD(T, p)
INORDER(T, n)
ELEMENT(T, p)
n ← RIGHT-SIBLING(T, n)
while n ≠ null
      do INORDER (T, n)
         n ← RIGHT-SIBLING(T, n)
```

```
POSTORDER(T, p)
if p = null
  then return
n ← LEFT-MOST-CHILD(T, p)
while n ≠ null
      do POSTORDER(T, n)
         n ← RIGHT-SIBLING(T, n)
ELEMENT(T, p)
```

چند عمل دیگر

```
COUNTNODES(T, p)
      ▷ تعداد گره‌های موجود در درخت T با ریشه‌ی p را می‌شمارد
  if T = null
    then return 0
  count ← 1
  p ← LEFT-MOST-CHILD(T, p)
  while p ≠ null
        do count ← count+ COUNTNODES(T, p)
           p ← RIGHT-SIBLING (T, p) return COUNT
```

```
NodeHeight (T, p)
      ▷ ارتفاع گره p را در درخت T محاسبه می‌کند و آن‌را برمی‌گرداند
      ▷ از نحوه‌ی پیاده‌سازی درخت اطلاعی نداریم
  if isEmpty(T)
    then return -1
  height ← 0
  p ← Left-Most-Child(T, p)
  while p ≠ null
        do height ← max{height, NodeHeight (T, p)}
           p ← Right-Sibling(T, p)
  return height + 1
```

تمرین: پیاده‌سازی parent با عمل‌های دیگر:

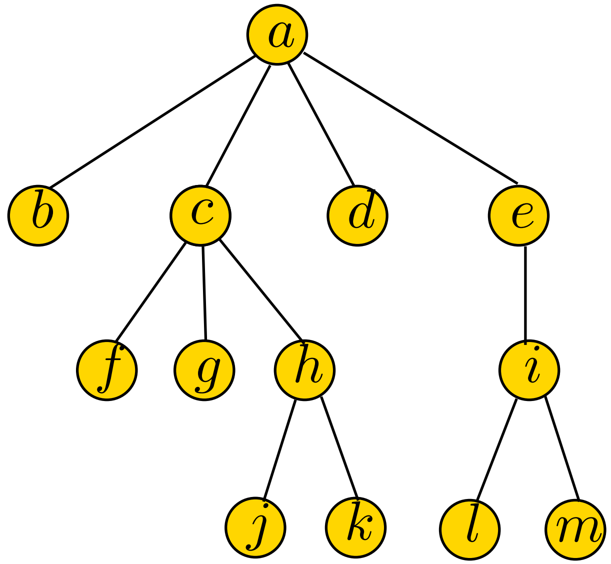
پدر p در درخت T در زیردرخت به ریشه‌ی r

```
FIND-PARENT(T, r, p)
      ▷ گره پدر یک گره p را در درخت T به ریشه‌ی r برمی‌گرداند
      ▷ در صورتی که عنصر در زیردرخت نباشد null برمی‌گرداند
      ▷ فرض می‌کنیم که اشاره‌گر «پدر» وجود ندارد
if p = r
  then return null
q ← Left-Most-Child(T, r)
while q ≠ null
      do if p = q
           then return r
         s ← PARENT(T, q, p)
         if s ≠ null
           then return s
         q ← RIGHT-SIBLING(T, q)
return null
```

پیاده‌سازی درخت‌ها با آرایه

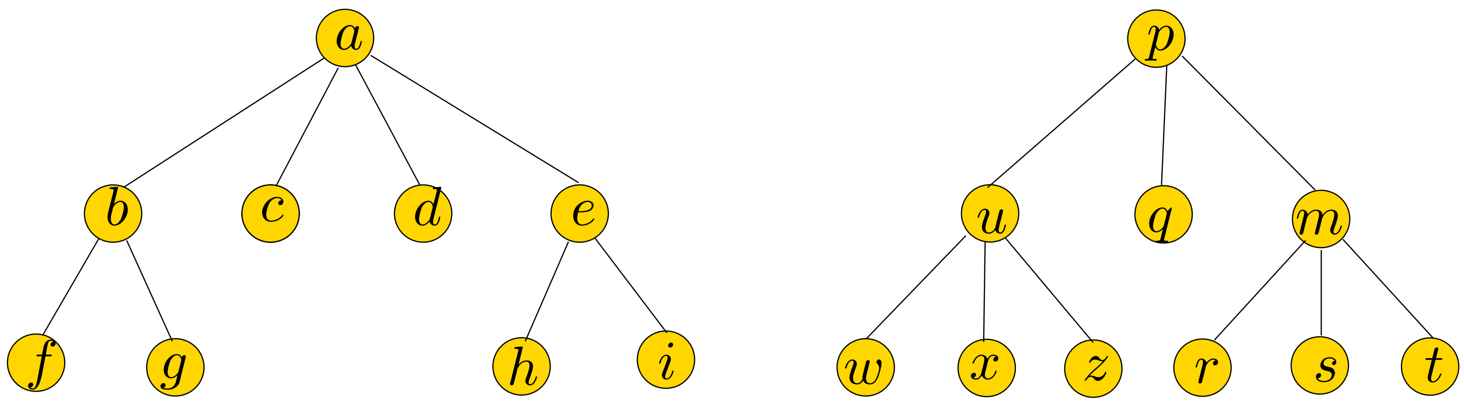
درایه‌ی ریشه: مولفه‌ی Father آن صفر است.

پیاده‌سازی درخت‌های مرتب ؟ ترتیب برادرها باید حفظ گردد.



	1	2	3	4	5	6	7	8	9	10	11	12	13
key	a	b	c	d	e	f	g	h	i	j	k	l	m
$parent$	۰	۱	۱	۱	۱	۲	۲	۲	۵	۸	۸	۹	۹

با این روش می‌توان چند درخت را در یک آرایه پیاده‌سازی کرد.



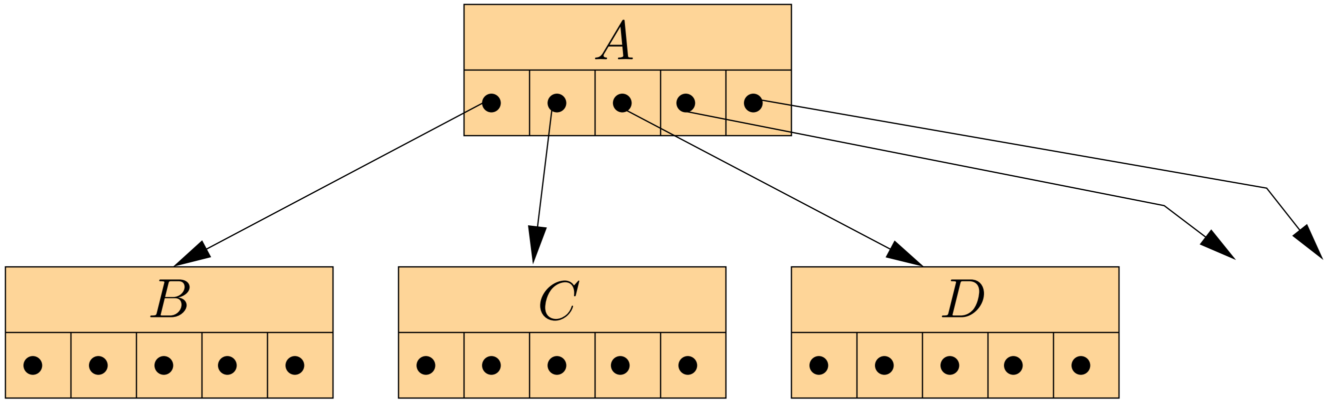
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	c	d	e	f	p	u	g	h	i	q	m	r	s	t	w	x	z
۰	۱	۱	۱	۱	۲	۰	۷	۲	۵	۵	۷	۷	۱۳	۱۳	۱۳	۸	۸	۸

با استفاده از اشاره‌گرها

روش بد:

هر گره

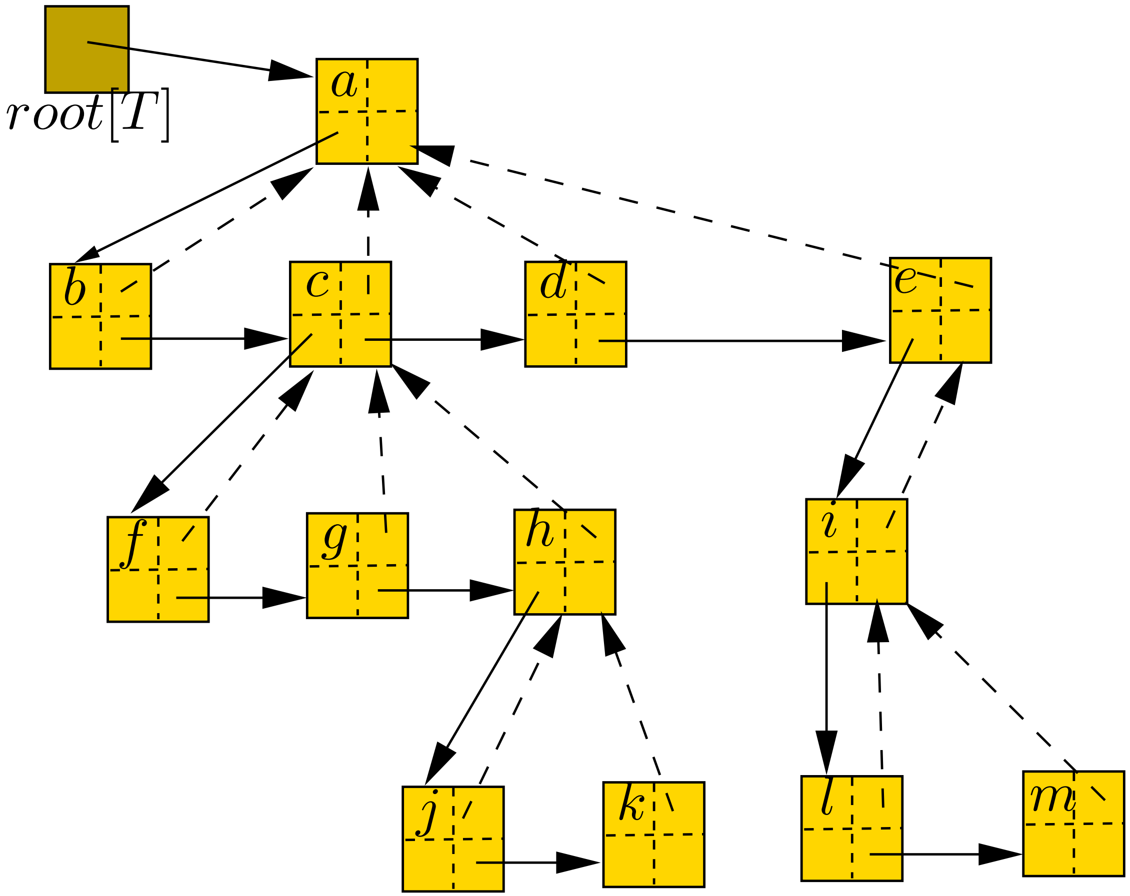
- مولفه‌های label،
- یک آرایه‌ی Child[1..max-child] که child[i] به فرزند iام آن گره اشاره می‌کند.
- مقدار max_child حداکثر تعداد فرزندان یک گره در درخت است.


A
B
C
D

پیاده‌سازی خوب: درخت دودویی معادل

برای هر گره

- مولفه‌ی label
- سه اشاره‌گر left-most-child، right-sibling و parent
- به اولین فرزند سمت‌چپ، برادر سمت راست و به پدر آن گره (در درخت اصلی)



$root[T]$
a
b
c
d
e
f
g
h
i
j
k
l
m

پیاده‌سازی با استفاده از اشاره‌گرهای اندیسی

مقایسه‌ی دو روش

ایجاد و گسترش درخت پیاده‌سازی شده به این روش

$\underline{\text{CREATE2}}\,(x, T_1, T_2)$

▷ درختی به ریشه‌ای که برچسب آن x است ایجاد می‌کند

▷ که زیر درخت‌های T_1 و T_2 به‌ترتیب زیردرخت‌های اول و دوم آن باشند

▷ فرض: T_1 تهی نیست و زیردرخت‌ها به‌درستی با همین روش پیاده‌سازی شده‌اند

$r \leftarrow$ ALLOCATE-NODE $(x, T_1, \textbf{null})$

1. $\textit{parent}[\text{ROOT}(T_1)] \leftarrow r$
2. $\textit{parent}[\text{ROOT}(T_2)] \leftarrow r$
3. $\textit{right-sibling}[\text{ROOT}(T_1)] \leftarrow \text{ROOT}(T_2)$
4. **return** r

$\underline{\textsc{Create3}}\,(x, T_1, T_2, T_3)$

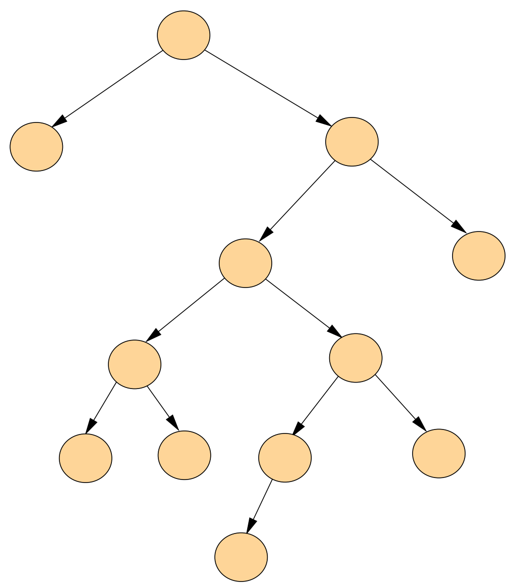
$\triangleright$ همان کار Create2 را با سه زیر درخت T_1، T_2 و T_3 انجام می‌دهد $\triangleleft$

$r \leftarrow$ Allocate-Node $(x, T_1, \textbf{null})$

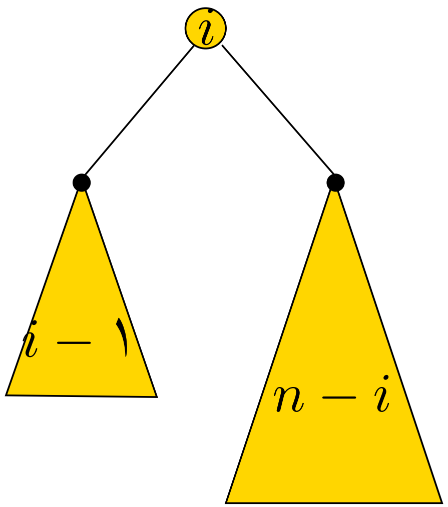
1 $\mathit{parent}[\textsc{Root}(T_1)] \leftarrow r$
2 $\mathit{parent}[\textsc{Root}(T_2)] \leftarrow r$
3 $\mathit{parent}[\textsc{Root}(T_3)] \leftarrow r$
4 $\mathit{right\text{-}sibling}[\textsc{Root}(T_1)] \leftarrow \textsc{Root}(T_2)$
5 $\mathit{right\text{-}sibling}[\textsc{Root}(T_2)] \leftarrow \textsc{Root}(T_3)$
6 **return** r

درخت دودویی

هر گره دو مولفه‌ی left و right دارد که به فرزند چپ و راست آن گره اشاره می‌کند.
ممکن است parent هم داشته باشد.



تعداد درخت‌های دودویی با n گره



$$T(n)=\begin{cases} T(۰)=۱ \\ T(n)=\sum_{i=۱}^{n} T(i-۱)T(n-i),\ \ n\geq ۱ \end{cases}$$

جواب این رابطه‌ی بازگشتی $T(n)=\frac{۱}{n+۱}\binom{۲n}{n}$ **(عدد** n**ام کاتالان)**

```
PREORDER(T, r)
      ▷ پیمایش پیش‌ترتیب درخت T به ریشه‌ی r
  if r = null
    then return
  meet element(T, r)
  PREORDER(T, left[r])
  PREORDER(T, right[r])
```

درخت عبارت (Expression Tree)

نگارش‌های مختلف یک عبارت:

میان‌وندی با پرانتزی کامل (infix with complete paranthesis)

$$
\begin{array}{rcll}
E & \rightarrow & (E\ \langle\beta\rangle\ E\ ) & \\
 & \rightarrow & (\ \langle\alpha\rangle E\ ) & \\
 & \rightarrow & \langle\texttt{operand}\rangle & \\
\langle\alpha\rangle & \rightarrow & \neg \mid ! \mid Sin \mid Log \mid \cdots & \texttt{unary operators} \\
\langle\beta\rangle & \rightarrow & - \mid + \mid * \mid / \mid \hat{}\ \mid \wedge \mid \vee \mid \cdots & \texttt{binary operators}
\end{array}
$$

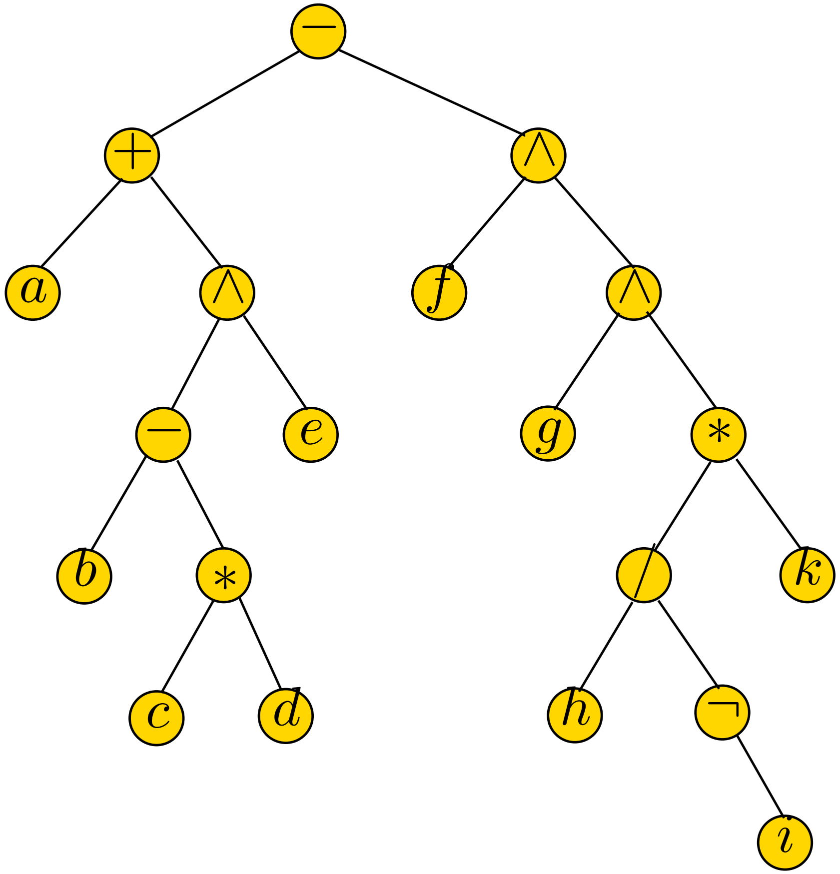
نگارش پسوندی (postfix)

$$
\begin{aligned}
E &\rightarrow EE\langle\beta\rangle \\
&\rightarrow E\langle\alpha\rangle \\
&\rightarrow \langle\texttt{operand}\rangle
\end{aligned}
$$

نگارش پیش‌وندی (prefix)

$$
\begin{aligned}
E &\rightarrow \langle\beta\rangle EE \\
&\rightarrow \langle\alpha\rangle E \\
&\rightarrow \langle\texttt{operand}\rangle
\end{aligned}
$$

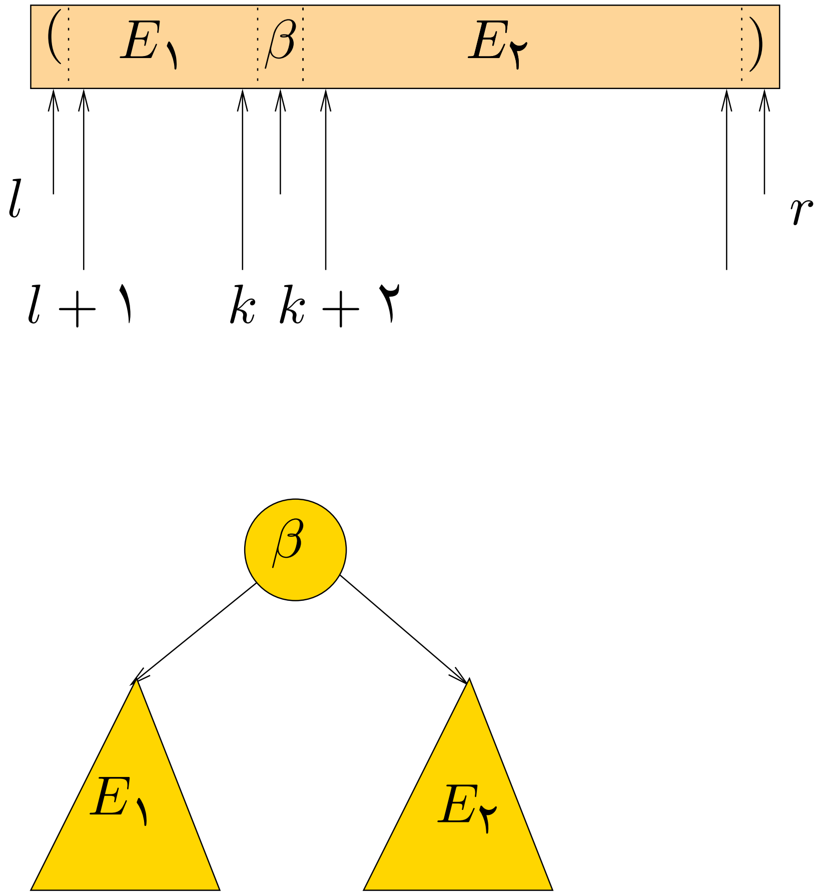
`a+(b-c*d)^e-f^g^(h /-i* k)`

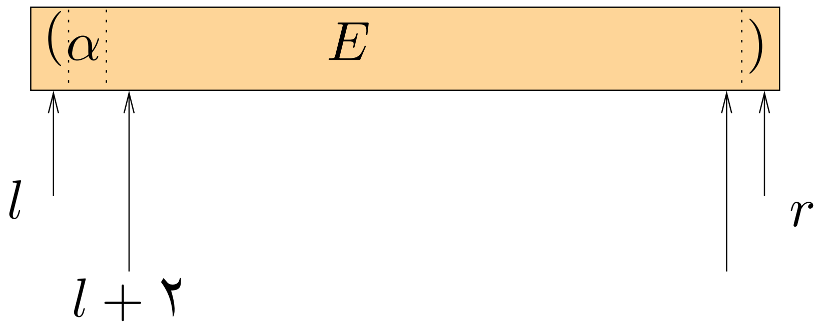


نام روش	نگارش عبارت
میان‌وندی	a+(b-c*d)^e-f^g^(h/¬i*k)
میان‌وندی با پرانتز کامل	((a+((b-(c*d))^e))-(f^(g^((h/(¬i))*k))))
پس‌وندی	abcd*-e^+fghi¬/k*^^-
پیش‌وندی	-+a^-b*cde^f^g*/h¬ik

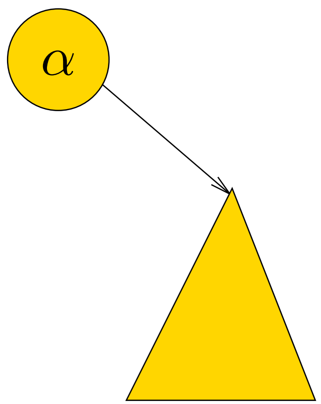
تبدیل نگارش‌های مختلف یک عبارت

میان‌وندی با پرانتز کامل ← درخت عبارت





$(\alpha$
E
$)$
l
$l + ۲$
r



α

P-Infix-to-Tree (l, r)

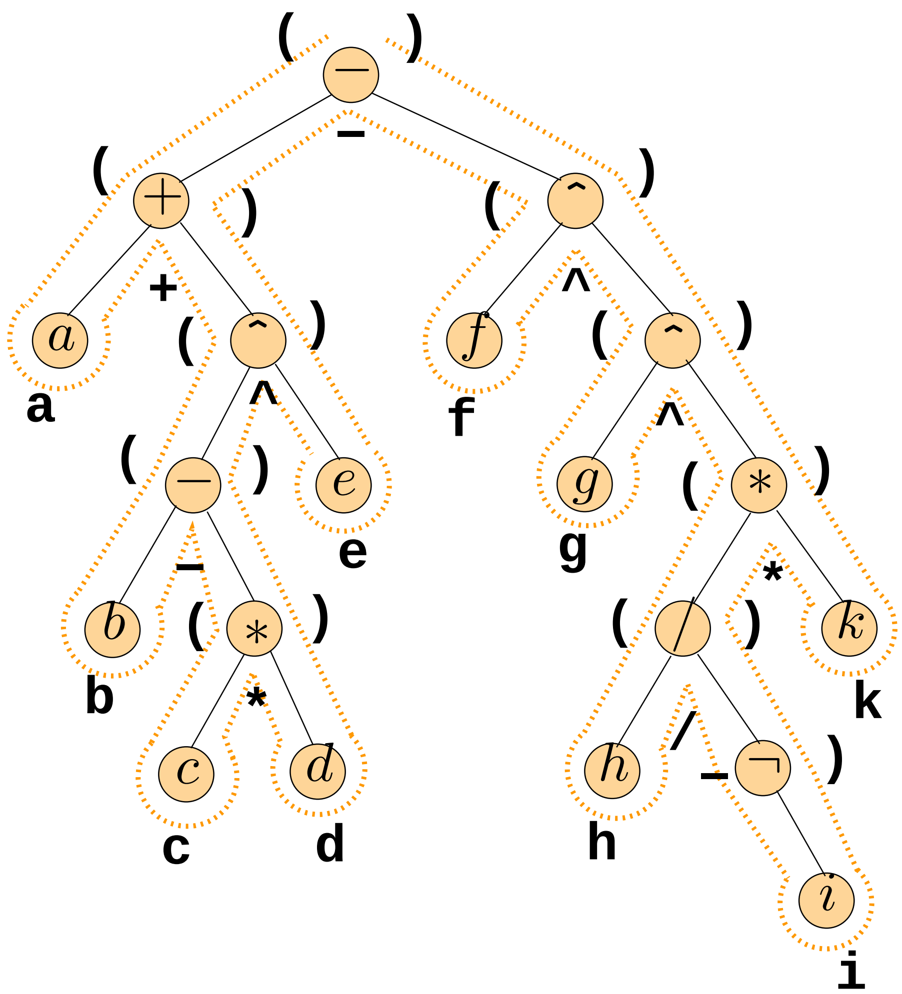
▷ یک عبارت میان‌وندی با پرانتز کامل که از اندیس l تا r در آرایه‌ی *par-infix* قرار دارد را به روش بازگشتی به درخت عبارت معادل آن تبدیل می‌کند

```
if l > r
  then return null
if l = r
  then return Allocate-Node (par-infix[l], null , null )
if par-infix[l + 1] in an unary operator
  then return Allocate-Node (par-infix[l + 1],null ,
              P-Infix-to-Tree(l + 2, r − 1))
k ← FindMatch(l + 1)
return Allocate-Node (par-infix[k + 1],
      P-Infix-to-Tree(l + 1, k), P-Infix-to-Tree(k + 2, r − 1))
```

$\mathcal{O}(n^۲)$

```
NR-P-INFIX-TO-TREE()
      ▷ فرض می‌شود که گره‌های درخت، اشاره‌گر به پدر دارند
CREATE-TREE (T)
p ← ROOT (T)
for i = 1 to length[par-infix]
    do switch
            case par-infix[i] = '('
                    do if par-infix[i + 1] ≠ unary operator
                         then left[p] ← ALLOC-N (null , null , p)
                              p ← left[p]
            case par-infix[i] = ')'
                    do p ← parent[p]
            case par-infix[i] is an operand
                    do label[p] ← par-infix[i]; p ← parent[p]
            case par-infix[i] is a binary or unary operator
                    do label[p] ← par-infix[i]
                       right[p] ← ALLOC-N (null , null , p)
                       p ← right[p]
```

`((a+((b−(c*d))^e))−(f^(g^((h/−i))*k))))`



تبدیل عبارت میان‌وندی (نه لزوماً با پرانتزی کامل) به عبارت پس‌وندی

خروجی	پشته →	← نویسه‌های ورودی	
		a	$+(b-c*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
a		$+$	$(b-c*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
a	$+$	$($	$b-c*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
a	$+($	b	$-c*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
ab	$+($	$-$	$c*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
ab	$+(-$	c	$*d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
abc	$+(-$	$*$	$d)\wedge e-f\wedge g\wedge(h/\neg i*k)$
abc	$+(-*$	d	$)\wedge e-f\wedge g\wedge(h/\neg i*k)$
$abcd$	$+(-*$	$)$	$\wedge e-f\wedge g\wedge(h/\neg i*k)$
$abcd*-$	$+$	$\wedge$	$e-f\wedge g\wedge(h/\neg i*k)$
$abcd*-$	$+\wedge$	e	$-f\wedge g\wedge(h/\neg i*k)$
$abcd*-e$	$+\wedge$	$-$	$f\wedge g\wedge(h/\neg i*k)$

نویسه‌های ورودی $\leftarrow$		پشته $\rightarrow$	خروجی
$\wedge g \wedge (h/\neg\, i * k)$	f	$-$	$abcd * -e \wedge +$
$g \wedge (h/\neg\, i * k)$	$\wedge$	$-$	$abcd * -e \wedge +f$
$\wedge(h/\neg\, i * k)$	g	$-\wedge$	$abcd * -e \wedge +f$
$(h/\neg\, i * k)$	$\wedge$	$-\wedge$	$abcd * -e \wedge +fg$
$h/\neg\, i * k)$	$($	$- \wedge \wedge$	$abcd * -e \wedge +fg$
$/\neg\, i * k)$	h	$- \wedge \wedge($	$abcd * -e \wedge +fg$
$\neg\, i * k)$	$/$	$- \wedge \wedge($	$abcd * -e \wedge +fgh$
$i * k)$	$\neg$	$- \wedge \wedge(/$	$abcd * -e \wedge +fgh$
$*k)$	i	$- \wedge \wedge(/\neg$	$abcd * -e \wedge +fgh$
$k)$	$*$	$- \wedge \wedge(/\neg$	$abcd * -e \wedge +fghi$
$)$	k	$- \wedge \wedge(*$	$abcd * -e \wedge +fghi\neg\, /$
	$)$	$- \wedge \wedge(*$	$abcd * -e \wedge +fghi\neg\, /k$
		$- \wedge \wedge$	$abcd * -e \wedge +fghi\neg\, /k*$
			$abcd * -e \wedge +fghi\neg\, /k * \wedge \wedge -$

t : عمل‌گری که بالای پشته است

c: عمل‌گر ورودی	(	$-$	$+$	$\times$	/	^	$\neg$
(	Push	Push	Push	Push	Push	Push	Push
$-$	Push	Pop	Pop	Pop	Pop	Pop	Pop
$+$	Push	Pop	Pop	Pop	Pop	Pop	Pop
*	Push	Push	Push	Pop	Pop	Pop	Pop
/	Push	Push	Push	Pop	Pop	Pop	Pop
^	Push	Push	Push	Push	Push	Push	Pop
$\neg$	Push	Push	Push	Push	Push	Push	Pop
)	Pop-more	Pop	Pop	Pop	Pop	Pop	Pop

جدول Action[i,j]

```
INFIX-TO-POSTFIX (infix)
     INTIALIZE-ACTIONS()
  while there is token in infix
        do read token c from infix
           if c is an operand
             then write c at the end of postfix
             else done ←  false
                  while  not done
                         do if ISEMPTY(S)
                              then PUSH(c, S); done ←  true
                              else t ←  TOP(S)
                                   if action[c, t] ='PUSH'
                                     then PUSH (S, c); done ←  true
                                     else if t ≠ '('
                                            then write t at postfix
                                          POP(S)
  while  not isEmpty(S)
        do write TOP(S) at the end of postfix; POP(S)
```

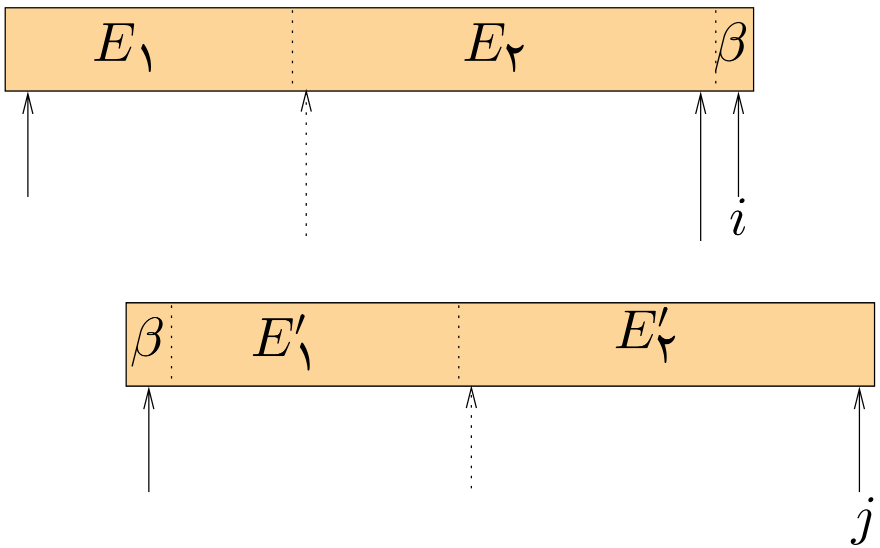
میانوندی ← درخت عبارت

- به‌جای خروجی پشته قرار می‌دهیم که در آن ریشه‌های زیردرخت‌های عبارت‌هایی که تا کنون ساخته شده‌اند قرار می‌گیرد
- چاپ عمل‌وند در خروجی یعنی ایجاد یک گره با عمل‌وند و درج آن در پشته‌ی خروجی
- چاپ عمل‌گر دودویی در خروجی یعنی دو بار Pop از پشته‌ی خروجی و ایجاد یک زیر درخت با این عمل‌گر و درج زیردرخت جدید در پشته‌ی خروجی
- چاپ عمل‌گر یگانی یعنی یک بار Pop و درج زیردرخت جدید به‌جای آن

پس‌وندی ← پیش‌وندی

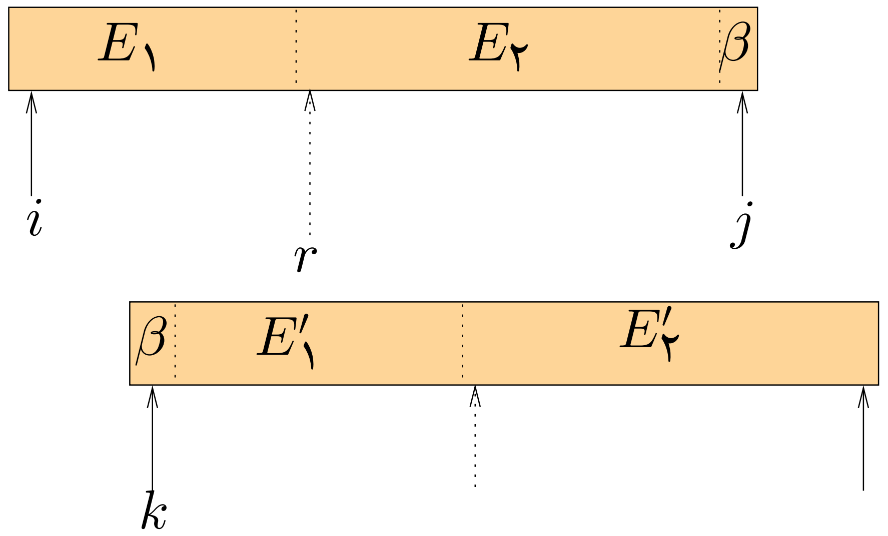
در یک رویه‌ی بازگشتی

- زیرعبارت پس‌وندی $postfix[? \cdots i]$ را به نگارش پیش‌وندی در $prefix[? \cdots j]$ تبدیل می‌کنیم
- در انتها، i و j اندیس‌های شروع این دو زیرعبارت هستند
- $postfix$ و $prefix$ آرایه‌های سراسری فرض می‌شوند


E_1
E_2
β
i
β
E'_1
E'_2
j

```
Postfix-to-Prefix (i, j)
    switch
        case postfix[i] is an operand
            do prefix[j] ← postfix[i]
        case postfix[i] is a binary operator
            do operator ← postfix[i]
               i ← i − 1
               Postfix-to-Prefix (i, j)
               i ← i − 1; j ← j − 1
               Postfix-to-Prefix (i, j)
               prefix[j − 1] ← operator
               j ← j − 1
        case postfix[i] is an unary operator
            do operator ← postfix[i]; i ← i − 1
               Postfix-to-Prefix (i, j)
               prefix[j − 1] ← operator
               j ← j − 1
```

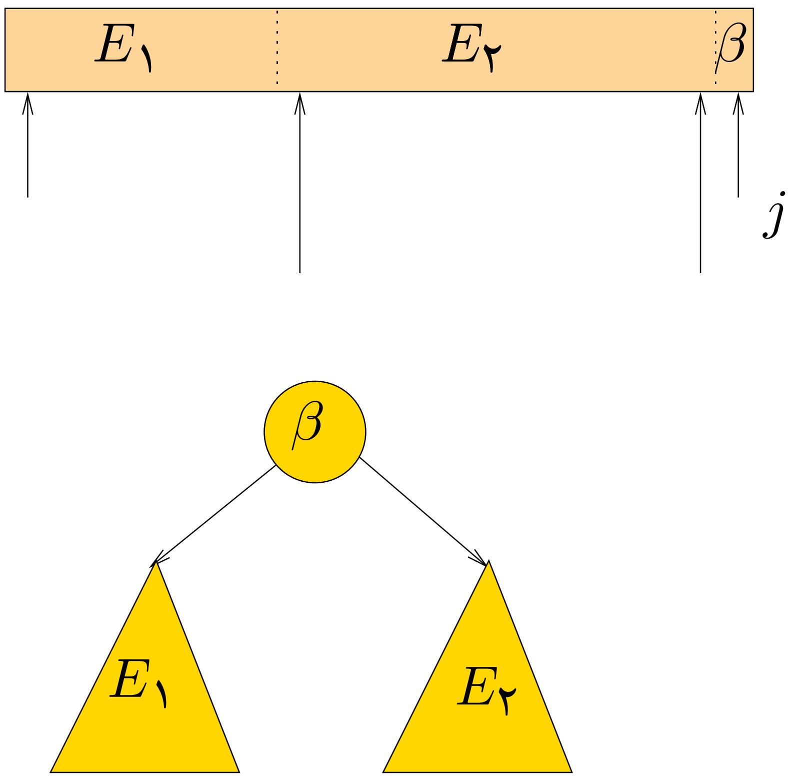
پس‌وندی ← پیش‌وندی (راه دوم)

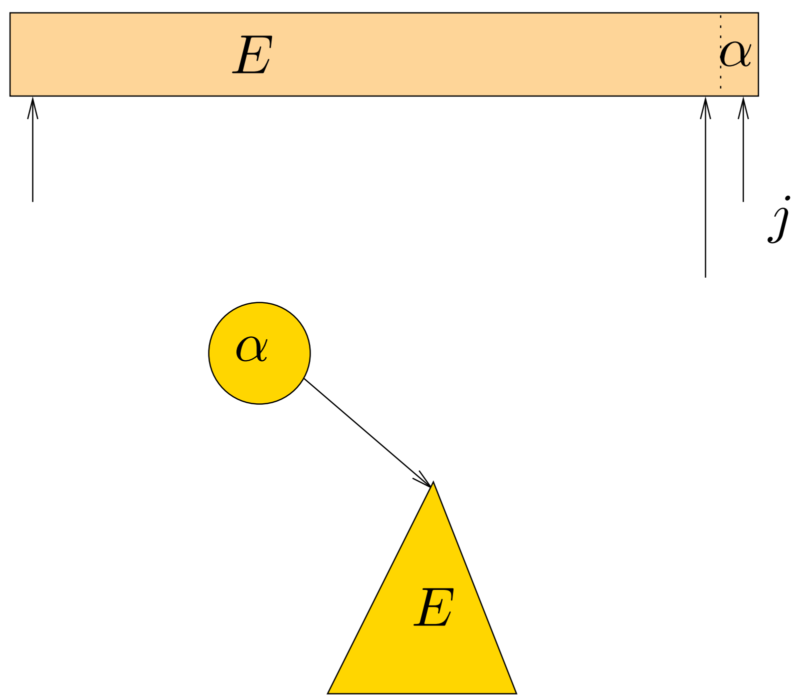


```
POSTFIX-TO-PREFIX (i, j, k)
      ▷ converts postfix[i..j] to prefix[k..?]
if j < i
  then return
if i = j
  then prefix[k] ← postfix[i]
  else prefix[k] ← postfix[j]
       r ← FindR (postfix, j − 1)
       Postfix-to-Prefix (i, r − 1, k + 1)
       Postfix-to-Prefix (r, j − 1, r − i + k + 1)
```

```
FindR (A, j)
switch
        case A[j] is a binary operator
                do count ← 2
        case A[j] is a unary operator
                do count ← 1
        default
                do count ← 1
r ← j
while Count > 0
        do r ← r − 1
            switch
                case A[r] is a binary operator
                        do count ← count + 1
                case A[r] is a unary operator
                        do nothing
                case
                        do count ← count − 1
                            return r
```

پس‌وندی ← درخت عبارت




E
α
j
α
E

```
POSTFIX-TO-TREE-1 (j)
        ▷ یک درخت عبارت برای postfix[?..j] ایجاد می‌کند
        ▷ فرض می‌شود که j یک متغیر آدرسی است و در انتها برابر اندیس شروع خواهد بود
n ← ALLOCATE-NODE(A[j], null , null )
switch
        case postfix[j] is a binary operator
                do j ← j − 1
                   right[n] ← POSTFIX-TO-TREE-1(j)
                   j ← j − 1
                   left[n] ← POSTFIX-TO-TREE-1(j)
        case postfix[j] is a unary operator
                do j ← j − 1
                   right[n] ← POSTFIX-TO-TREE-1(j)
return n
```

```
POSTFIX-TO-TREE-2 (postfix)
        ▷ یک درخت عبارت برای postfix ایجاد می‌کند
  CREATE-STACK(S)
        ▷ عناصر پشته اشاره‌گر به یک زیردرخت هستند
  for i ← 1 to length[postfix]
      do if postfix[i] is an operand
            then PUSH(S, ALLOCATE-NODE(postfix[i], null , null ))
         if postfix[i] is a unary operator
            then r ← ALLOCATE-NODE(postfix[i], null ,POP(S))
                 PUSH(S, r)
         if postfix[i] is a binary operator
            then t ← POP(S)
                 r ← ALLOCATE-NODE(postfix[i],POP(S), t)
                 PUSH(S, r)
  return TOP(S)
```

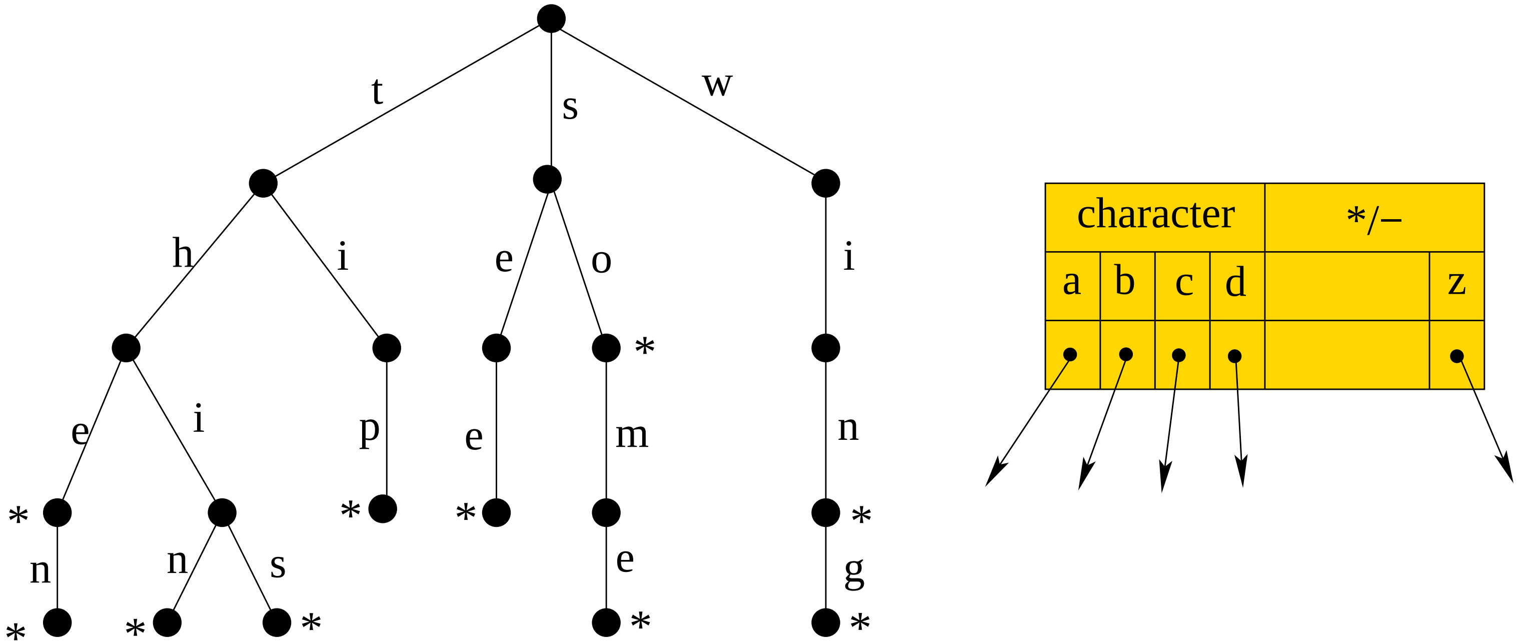
```
Postfix-to-Tree(i, j)
      ▷ Creates a tree for postfix[i..j]
  if j < i
    then return null
  n ← Allocate-Node(A[j], null , null )
  if i < j
    then r ← FindR(postfix, j − 1)
          left[n] ← Postfix-to-Tree (i, r − 1)
          right[n] ← Postfix-to-Tree (r, j − 1)
  return n
```

```
POSTFIX-TO-TREE(j)
        ▷ Makes a tree for postfix[?..j]
        ▷ j is assumed to be a reference variable
  n ← ALLOCATE-NODE(A[j], null , null )
  switch
        case postfix[j] is a binary operator
              do j ← j − 1
                 right[n] ← POSTFIX-TO-TREE(j)
                 j ← j − 1
                 left[n] ← POSTFIX-TO-TREE(j)
        case postfix[j] is a unary operator
              do j ← j − 1
                 right[n] ← POSTFIX-TO-TREE(j)
  return n
```

مثال: تِرای (Trie)

ترای (برگرفته‌شده از کلمه‌ی retrieval) داده‌ساختاری برای ذخیره‌ی تعداد کلمه به‌طوری که بتوان به‌صورت کارا تشخیص داد که یک کلمه در آن هست یا خیر (برای غلط‌یاب‌ها)

اگر کلمات انگلیسی با حروف 'a' تا 'z' باشند، هر گره‌ی تِرای، ۲۶ فرزند دارد هرکدام برای یک حرف.



یک تِرای برای کلمات wing، win، so، some، see، tip، this، thin، then، و the.

```
type Lettertype =  'a' .. 'z';
     Node = ^   Nodetype;
     Nodetype = record
                   letter : Lettertype;
                   isword: ('*', '-');
                   children: array[Lettertype] of Node
                end;
```

اعمال بر روی ترای

درج یک رشته در ترای

حذف یک رشته از ترای

جست‌وجو برای پیداکردن یک رشته در ترای

نوشتن تمام رشته‌های موجود در ترای

درج در ترای

<u>Trie-Insert</u> (T, x)

▷ کلمه‌ی x را در ترای T درج می‌کند

$p \leftarrow$ Root(T)
for $i \leftarrow 1$ **to** $length[x]$
  do if the value of $x[i]$th link of p is **null**
    then $x[i]$th link of $p \leftarrow$
      Allocate-Node(with $x[i]$ as its value)
    $p \leftarrow x[i]$th child of p
$tag[p] \leftarrow$ '*'

درخت دودویی جست‌وجو (تعریف)

-- درخت دودویی

درخت دودویی جست‌وجو (تعریف)

-- درخت دودویی

-- برای هر گره n با برچسب x

درخت دودویی جست‌وجو (تعریف)

– درخت دودویی

– برای هر گره n با برچسب x

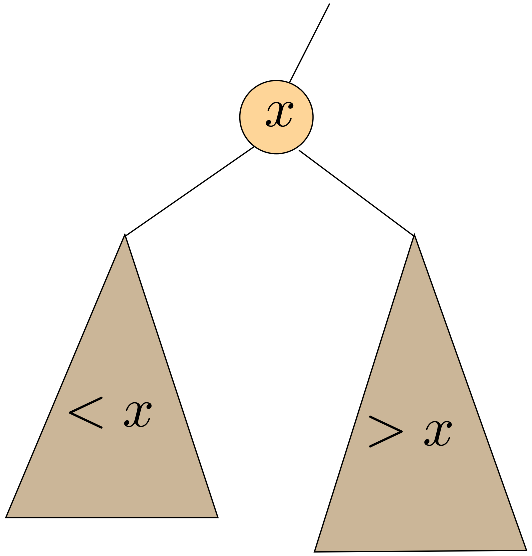
* برچسب کلیه‌ی گره‌های زیردرخت چپ n کم‌تر از x و

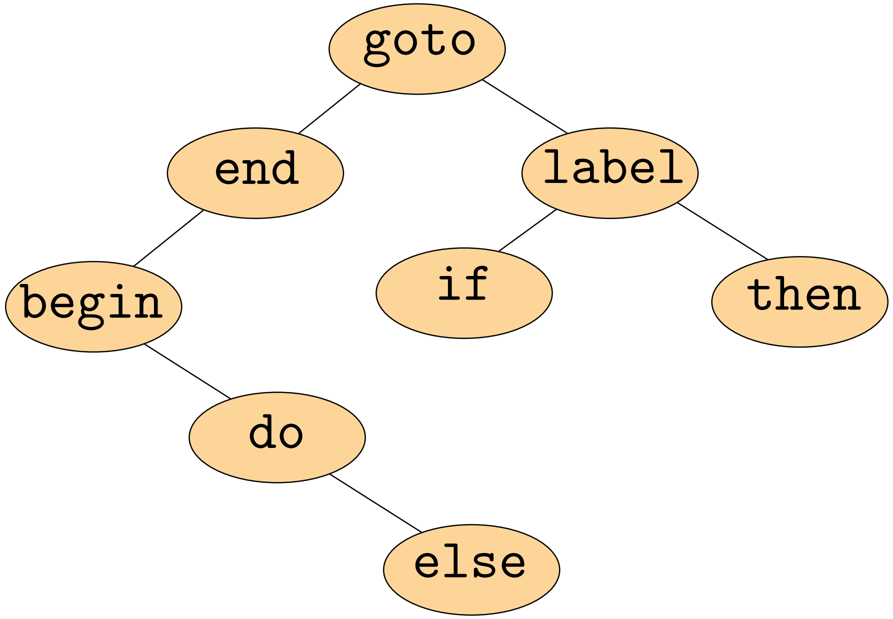
درخت دودویی جست‌وجو (تعریف)

-- درخت دودویی

-- برای هر گره n با برچسب x

* برچسب کلیه‌ی گره‌های زیردرخت چپ n کم‌تر از x و
* برچسب کلیه‌ی گره‌های زیردرخت راست آن بیش‌تر از x است

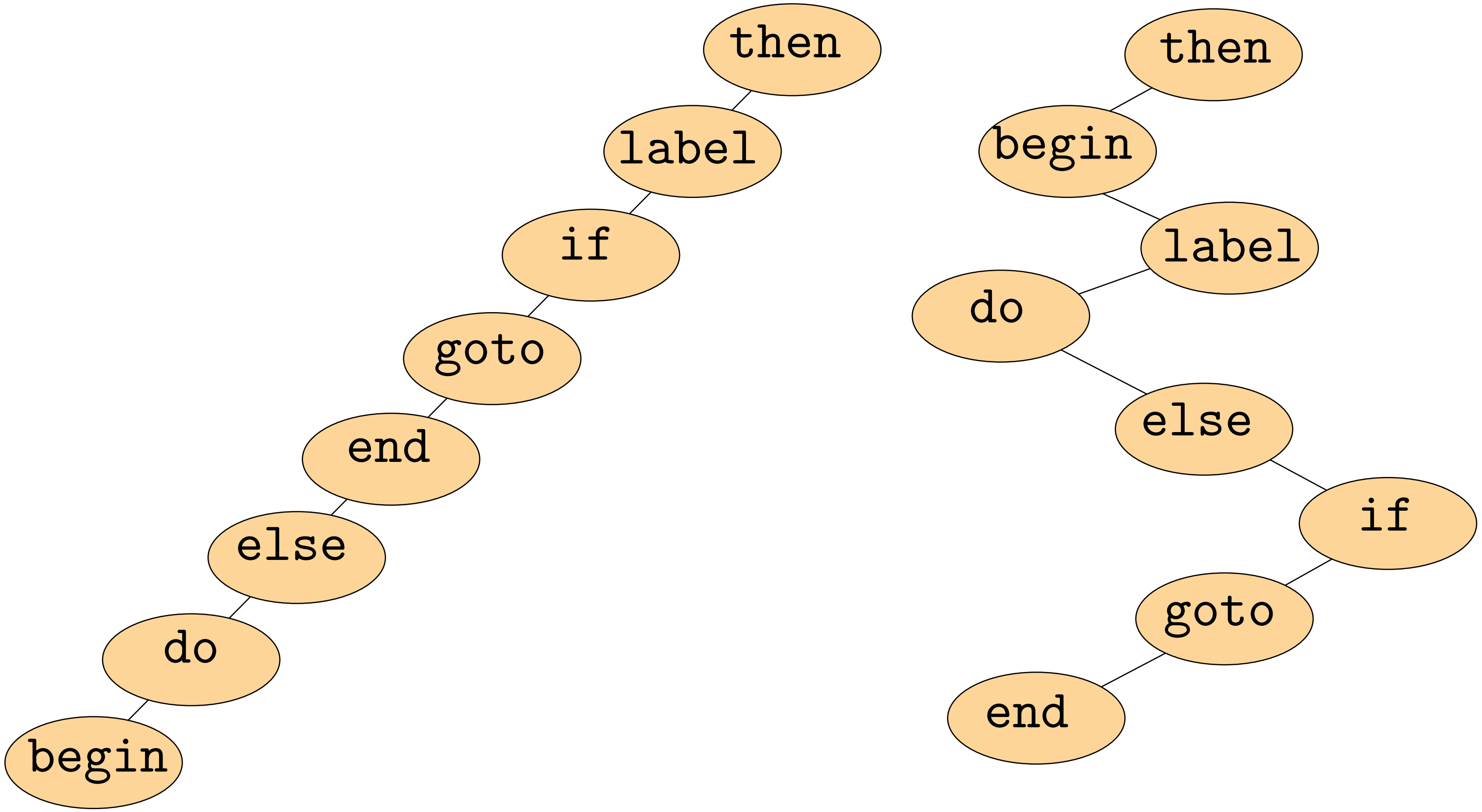




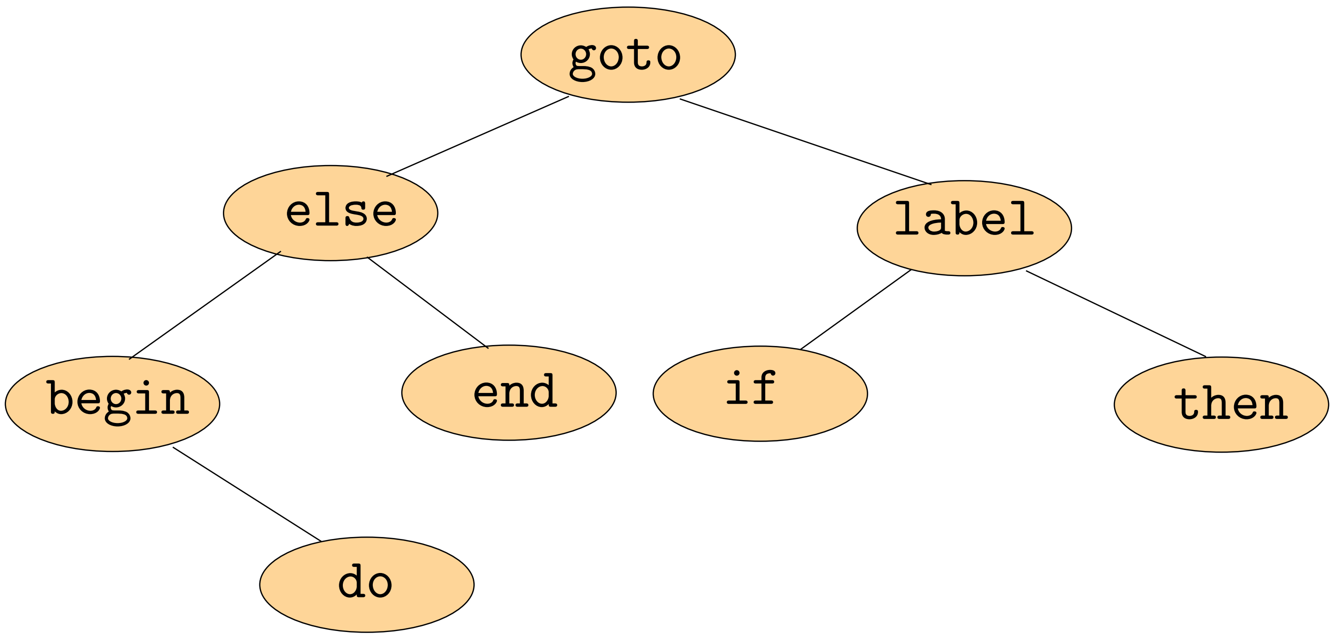
یک درخت دودویی جست‌وجو

پیمایش بین‌ترتیب $\longrightarrow$ مرتب‌شده‌ی عناصر

ترتیب درج در ایجاد درخت:



با بیش‌ترین ارتفاع.



با کم‌ترین ارتفاع.

$\lfloor \lg n \rfloor \le$ **ارتفاع** $\le n - ۱$ **(چرا؟)**

میانگین ارتفاع: $O(\lg n)$

چه تعداد درخت‌های دودویی جست‌وجو با ارتفاع $n-1$؟

چه تعداد درخت دودویی جست‌وجو با ارتفاع $n - ۱$؟

$$۲^{n-۱}$$

تعداد درخت‌های دودویی جست‌وجو

با $a_n < \cdots < a_۲ < a_۱$ چند تا درخت متفاوت می‌توان ساخت؟

تعداد درخت‌های دودویی جست‌وجو

با $a_۱ < a_۲ < \cdots < a_n$ چند تا درخت متفاوت می‌توان ساخت؟

$$\begin{aligned} T(n) &= \sum_{i=۱}^{n} T(i-۱)T(n-i), \\ T(۰) &= ۱ \end{aligned}$$

تعداد درخت‌های دودویی جست‌وجو

با $a_۱ < a_۲ < \cdots < a_n$ چند تا درخت متفاوت می‌توان ساخت؟

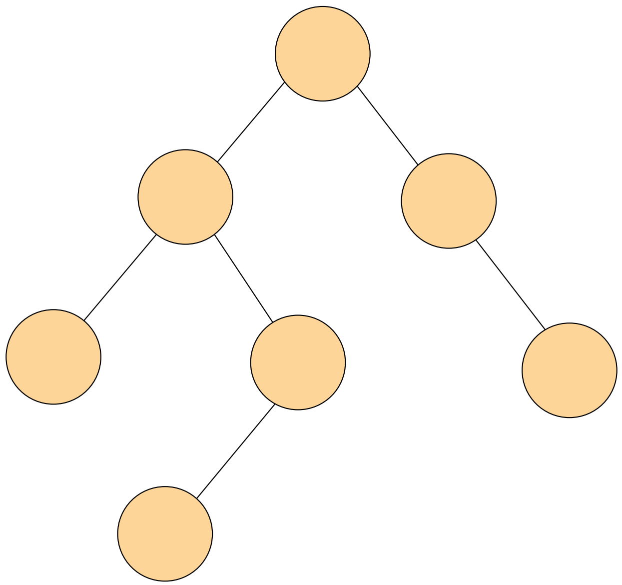
$$\begin{aligned} T(n) &= \sum_{i=۱}^{n} T(i-۱)T(n-i), \\ T(۰) &= ۱ \end{aligned}$$

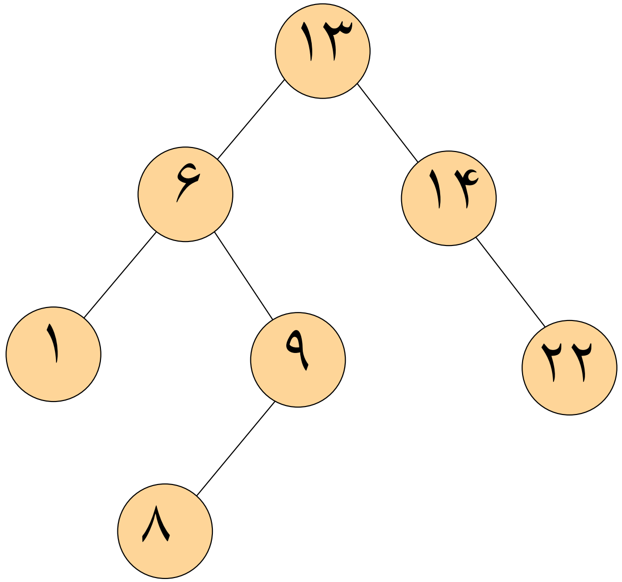
$$T(n) = \frac{۱}{n+۱}\binom{۲n}{n}$$

عدد کاتالان

همین‌جا حل کنید!

اعداد $\{۸, ۱, ۱۳, ۱۴, ۹, ۲۲, ۶\}$ را به درخت زیر طوری نسبت دهید که درخت دودویی جست‌وجو شود.





این دنباله‌های درج در یک درخت تهی درخت فوق را می‌سازد

۱۳, ۶, ۹, ۱, ۱۴, ۸, ۲۲

۱۳, ۱۴, ۶, ۲۲, ۱, ۹, ۸

۱۳, ۶, ۱, ۹, ۸, ۱۴, ۲۲

به‌چند حالت می‌توان اعداد فوق را وارد یک درخت تهی کرد تا در انتها درخت فوق حاصل شود؟ این مقدار را دقیقاً محاسبه کنید.

```
13,      6,   9,   1,    8
     ^           ^
     14          22
```

$$r \ \cup \ l_1 \ \cup \ l_2 \ldots \cup \ l_{n_1} \ \cup$$

$$T(n) \ = \ T(n_1)T(n_2)\frac{(n_1 + n_2)!}{n_1!n_2!}$$

n_1 تعداد گره‌ها در زیردرخت چپ است

n_2 تعداد گره‌ها در زیردرخت راست است

جست‌وجو

```
BST-SEARCH(r, x)
        ▷ r یک گره (یا ریشه‌ی) یک د.د.ج است
  if r = null  or x = key[r]
     then return r
  if x < key[r]
     then return BST-SEARCH(left[r], x)
     else return BST-SEARCH(right[r], x)
```

جست‌وجو (غیر بازگشتی)

NR-BST-SEARCH (r, x)

▷ r یک گره (یا ریشه‌ی) یک د.د.ج است

1 **while** $r \neq$ **null** **and** $x \neq key[r]$
2 **do**
3
4
5 **return** r

جست‌وجو (غیر بازگشتی)

```
NR-BST-Search (r, x)
        ▷ r یک گره (یا ریشه‌ی) یک د.د.ج است
while r ≠ null  and x ≠ key[r]
        do if x < key[r]
              then r ← left[r]
              else r ← right[r]
return r
```

یافتن عنصر کمینه

<u>BST-MINIMUM</u> (r)
1 **if** $left[r] =$ **null**
2 **then return** r BST-MINIMUM$(left[r])$

یافتن کمینه (غیر بازگشتی)

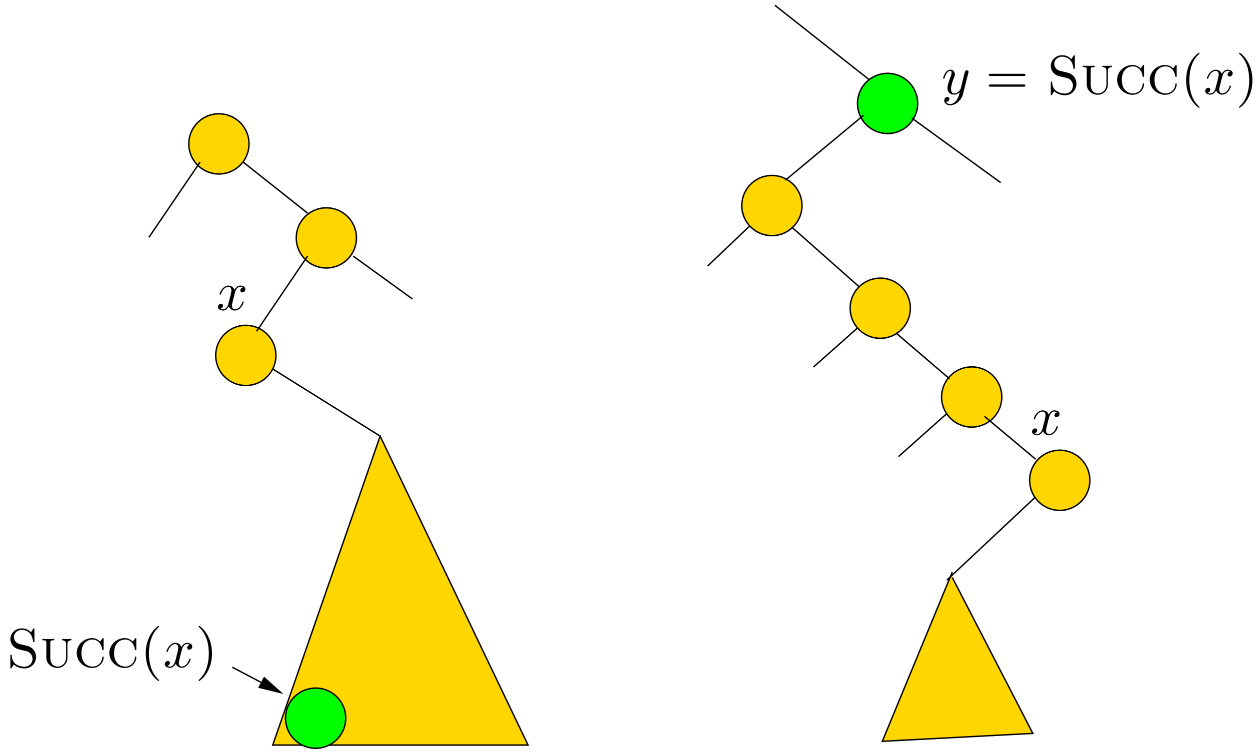
<u>BST-Minimum</u> (r)

1 **while** $left[r] \neq$ **null**

2 **do** $r \leftarrow left[r]$

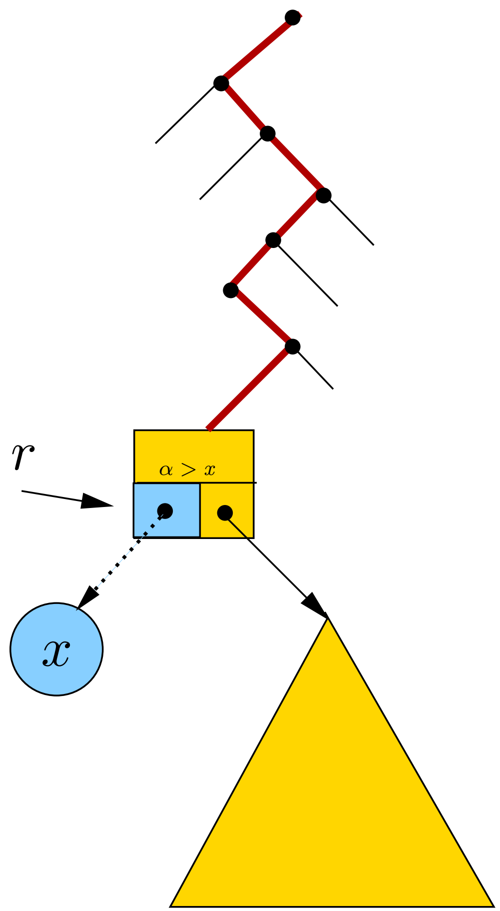
3 **return** r

عنصر بعدی



```
BST-Successor (r)
if right[r] ≠ null
  then return BST-Minimum(right[r])
y ← parent[r]
while y ≠ null and r = right[y]
      do r ← y
         y ← parent[y]
return y
```

درج x



<u>BST-Insert</u> (r, x)

▷ r یک گره (یا ریشه‌ی) د.د.ج است

▷ مهم: r باید یک پارامتر آدرسی باشد

▷ فرض می‌شود مؤلفه‌ی *parent* وجود ندارد

if $r =$ **null**
  then $r \leftarrow$ Allocate-Node(x, **null** , **null**)
if $x < key[r]$
  then BST-Insert($left[r], x$)
  elseif $x > key[r]$
    then BST-Insert($right[r], x$)

درج بازگشتی با مولفه‌ی پدر

BST-INSERT-2 (r, p, x)

▷ r یک گره (یا ریشه‌ی) د.د.ج

▷ p پدر r است

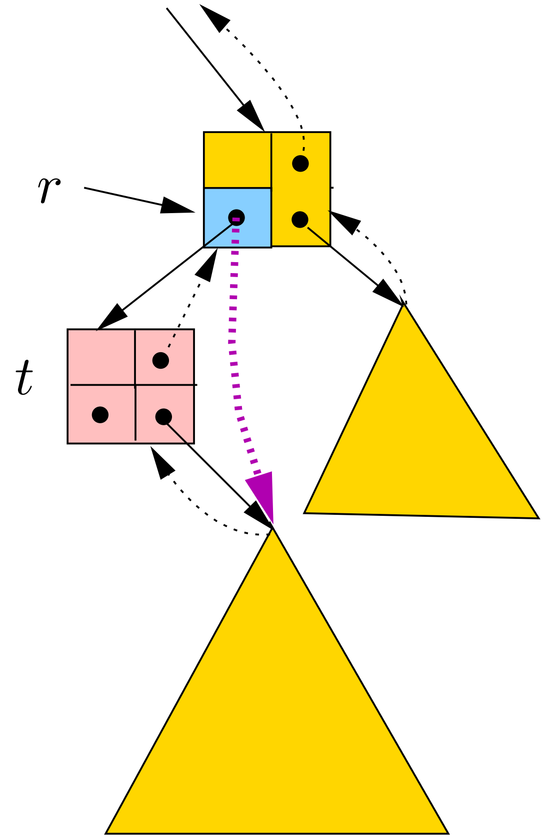
▷ مهم: r باید یک پارامتر آدرسی باشد

```
if r = null
  then r ← ALLOCATE-NODE(x, null , null )
       parent[r] ← p
if x < key[r]
  then BST-INSERT-2(left[r], r, x)
  elseif x > key[r]
       then BST-INSERT-2(right[r], r, x)
```

درج غیر بازگشتی

```
NR-BST-INSERT(T, x)
n ← ALLOCATE-NODE(x, null , null )
prep ← null
p ← root[T]
while p ≠ null
        do prep ← p
            if x < key[p]
              then p ← left[p]
              else if x > key[p]
                      then p ← right[p]
                      else return
parent[n] ← prep
if prep = null
  then root[T] ← n
  else if x < key[prep]
          then left[prep] ← n
          else right[prep] ← n
```

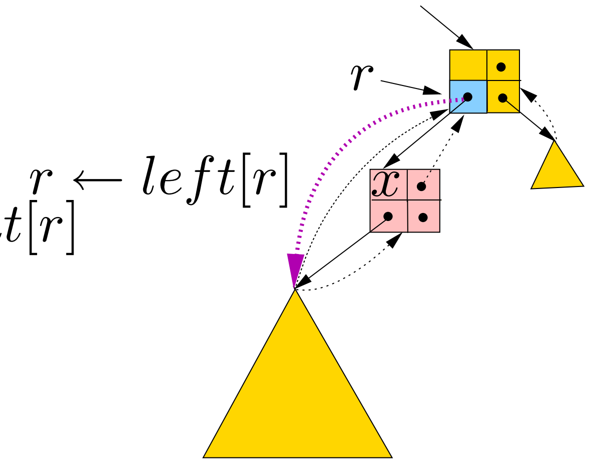
حذف کوچک‌ترین عنصر



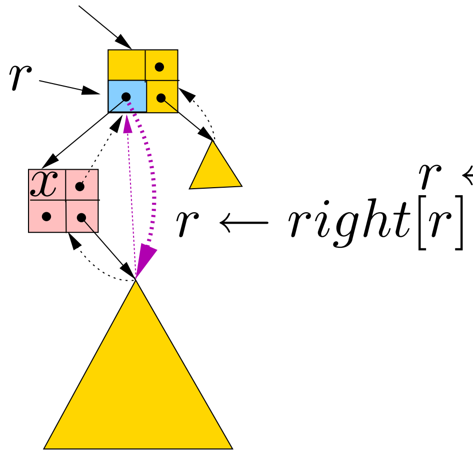
حذف کوچک‌ترین عنصر

```
BST-DeleteMin (r)
      ▷ r یک گره (یا ریشه‌ی) د.د.ج است
      ▷ مهم: r باید یک پارامتر آدرسی باشد
if r = null
  then error tree is empty
if left[r] = null
  then x ← label[r]
       t ← r
       r ← right[r]
       parent[r] ← parent[t]
       Free-Node (t)
       return x
  else return BST-DeleteMin(left[r])
```

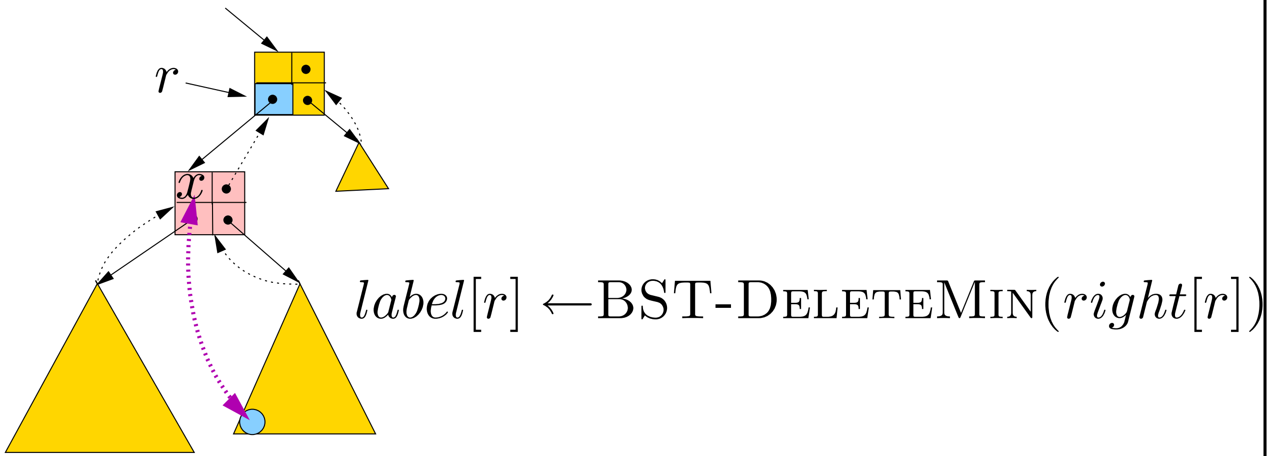
حذف یک عنصر


r
x
$r \leftarrow left[r]$

(a)


r
x
$r \leftarrow right[r]$

(b)


r
x
$label[r] \leftarrow$ BST-DeleteMin$(right[r])$

(c)

```
BST-DELETE (r, x)
      ▷ r is a node (or the root) of a BST
      ▷ r is a reference variable
  if r = null
     then  error ("Tree is Empty")
  if x < label[r]
     then BST-DELETE(left[r], x)
  if x > label[r]
     then BST-DELETE(right[r], x)
  if x = label[r]
     then
```

```
BST-DELETE (r, x)
      ⋮
  if x = label[r]
    then temp ← r
         if left[r] = null
           then r ← right[r]
                parent[r] ← parent[temp]
                FREEE-NODE(temp)
           else if right[r] = null
                  then r ← left[r]
                       parent[r] ← parent[temp]
                       FREEE-NODE(temp)
                  else label[r] ← BST-DELETEMIN (right[r])
```

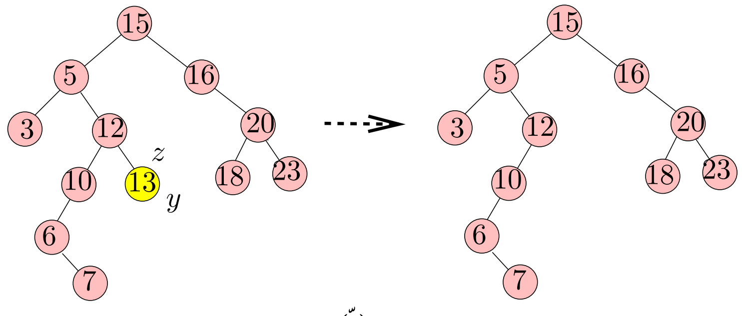
همین‌جا حل کنید!

آیا با داشتن دنباله‌ی بین‌ترتیب از عناصر یک ددج می‌توان آن‌را به‌صورت تک ساخت؟

با پس‌ترتیب چه‌طور؟

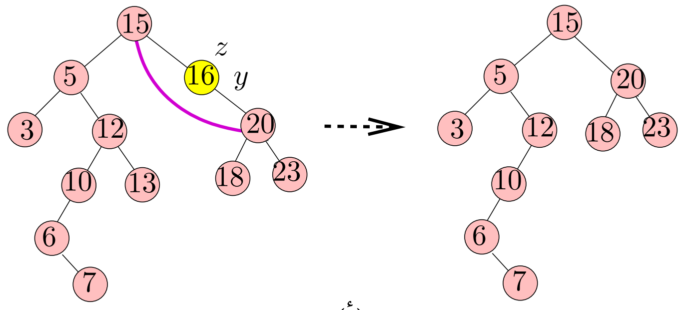
با پیش‌ترتیب چه‌طور؟

حذف غیر بازگشتی یک عنصر



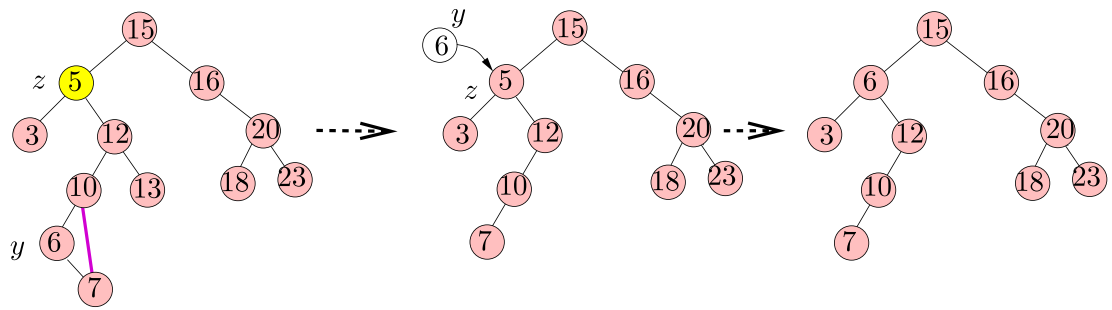
15 5 16 3 12 20 10 13 18 23 6 7
z
y
15 5 16 3 12 20 10 18 23 6 7

( )



15 5 16 3 12 20 10 13 18 23 6 7
z
y
15 5 20 3 12 18 23 10 6 7

( )



15 5 16 3 12 20 10 13 18 23 6 7
z
y
6
15 5 16 3 12 20 10 18 23 7
15 6 16 3 12 20 10 18 23 7

( )

```
NR-BST-Delete(T, z)
if left[z] = null  or rigt[z] = null
  then y ← z
  else y ← BST-Successor(z)
if left[y] ≠ null
  then x ← left[y]
  else x ← right[y]
if x ≠ null
  then parent[x] ← parent[y]
if parent[y] = null
  then root[T] ← x
  else if y = left[parent[y]]
          then left[parent[y]] ← x
          else right[parent[y]] ← x
if y ≠ z
  then key[z] ← key[y]
```

میانگین ارتفاع درخت دودویی جست‌وجوی

فرض: $a_1 < a_2 \cdots < a_n$ به‌صورت تصادفی و با احتمال یک‌سان وارد یک درخت د.د.ج تهی T می‌شوند.

اثبات می‌کنیم که میانگین ارتفاع T برابر $\mathcal{O}(\log n)$ است.

سه متغیر تصادفی:

-- X_n ارتفاع درختی که به‌صورت تصادفی برای n عنصر ایجاد می‌شود.

-- $Y_n = 2^{X_n}$ به‌خاطر سادگی کار با Y_n.

-- R_n اندیس اولین عنصری که وارد T می‌شود و در ریشه قرار می‌گیرد.

-- اگر $R_n = i$، زیردرخت چپ ریشه حتماً $i - ۱$ عنصر زیردرخت راست آن $n - i$ عنصر دارد.

-- هم‌چنین روشن است که ارتفاع T برابر یک واحد بیش‌تر از بیشینه‌ی ارتفاع‌های دو زیر درخت آن است.

-- پس $Y_n = ۲ \max\{Y_{i-۱}, Y_{n-i}\}$ و پایه‌ی این رابطه‌ی بازگشتی $Y_۱ = ۱$.

-- سعی می‌کنیم رابطه‌ای برای مقدار میانگین Y_n، یا $E[Y_n]$ به‌دست آوریم و سپس $E[X_n]$ را محاسبه کنیم.

-- برای این‌کار حالتی را در نظر می‌گیریم که $R_n = i$ باشد و آن‌را با متغیر زیر نمایش می‌دهیم.

$$Z_{n,i} = \{R_n = i\}$$

-- طبق فرض احتمالات یک‌سان برای عناصر درخت، بدیهی است که

$$E[Z_{n,i}] = ۱/n. \qquad (۱)$$

-- و داریم،

$$Y_n = \sum_{i=۱}^{n} Z_{n,i}(۲.\max\{Y_{i-۱}, Y_{n-i}\}). \qquad (۲)$$

ویژگی‌های متغیرهای تصادفی $E[Y_n]$ را مطابق زیر به‌دست می‌آوریم.

$$
\begin{aligned}
E[Y_n] &= E\left[\sum_{i=۱}^{n} Z_{n,i}(۲.\max\{Y_{i-۱}, Y_{n-i}\})\right] \\
&= \sum_{i=۱}^{n} E[Z_{n,i}(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \sum_{i=۱}^{n} E[Z_{n,i}]E[(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \sum_{i=۱}^{n} \frac{۱}{n}E[(۲.\max\{Y_{i-۱}, Y_{n-i}\})] \\
&= \frac{۲}{n}\sum_{i=۱}^{n} E[(\max\{Y_{i-۱}, Y_{n-i}\})] \\
&\leq \frac{۲}{n}\sum_{i=۱}^{n} (E[Y_{i-۱}] + E[Y_{n-i}]).
\end{aligned}
$$

اثبات نامساوی آخر به‌عنوان تمرین داده شده است.

این نامساوی را می‌توان به‌صورت زیر هم نوشت:

$$E[Y_n] \leq \frac{۴}{n} \sum_{i=۱}^{n-۱} (E[Y_i]) \qquad (۳)$$

با استفاده از روش جای‌گذاری می‌توان نشان داد که حاصل رابطه‌ی بازگشتی عبارت زیر است:

$$E[Y_n] \leq \frac{۱}{۴} \binom{n+۳}{۳} \qquad (۴)$$

برای اثبات این فرمول، در تمرین نشان می‌دهیم که

$$\sum_{i=۰}^{n-۱} \binom{i+۳}{۳} = \binom{n+۳}{۳} \qquad (۵)$$

در ابتدا برای $Y_۱$ داریم:

$$۱ = Y_۱ = E[Y_۱] \leq \frac{۱}{۴} \binom{۱+۳}{۳} = ۱$$

و این پایه‌ی استقرا است. برای حالت کلی داریم،

$$\begin{aligned}
E[Y_n] &\leq \frac{۴}{n}\sum_{i=۰}^{n-۱} E[Y_i] \\
&= \frac{۴}{n}\sum_{i=۰}^{n-۱} \frac{۱}{۴}\binom{i+۳}{۳} \\
&= \frac{۱}{n}\sum_{i=۰}^{n-۱} \binom{i+۳}{۳} \\
&= \frac{۱}{n}\binom{n+۳}{۴} \\
&= \frac{۱}{n}\frac{(n+۳)!}{۴!(n-۱)!} \\
&= \frac{۱}{۴}\frac{(n+۳)!}{۳!n!} \\
&= \frac{۱}{۴}\binom{n+۳}{۳}.
\end{aligned}$$

از آن‌طرف می‌دانیم که

$$۲^{E[x_n]} \leq E[۲^{X_n}] = E[Y_n]$$

پس

$$\begin{aligned} ۲^{E[x_n]} &\leq \frac{۱}{۴}\binom{n+۳}{۳} \\ &= \frac{۱}{۴}.\frac{(n+۳)(n+۲)(n+۱)}{۶} \\ &= \frac{n^۳+۶n^۲+۱۱n+۶}{۲۴}. \end{aligned}$$

که اگر از طرفین لگاریتم بگیریم خواهیم داشت، $E[X_n] = \mathcal{O}(\lg n)$، همان‌که می‌خواستیم ثابت کنیم. بنابراین،

قضیه‌ی ۱ میانگین ارتفاع یک درخت دودویی جست‌وجو که به‌صورت تصادفی با n

عنصر ساخته می‌شود برابر $\mathcal{O}\lg n$ است.

صف اولویت (Priority Queue)

از بخش ۶/۵ کتاب CLRS

داده‌ساختاری برای اعمال

-- درج
-- حذف بزرگ‌ترین (کوچک‌ترین) عنصر
-- افزایش (کاهش) مقدار کلید یک عنصر

همه در $O(\lg n)$

صف اولویت (تعریف)

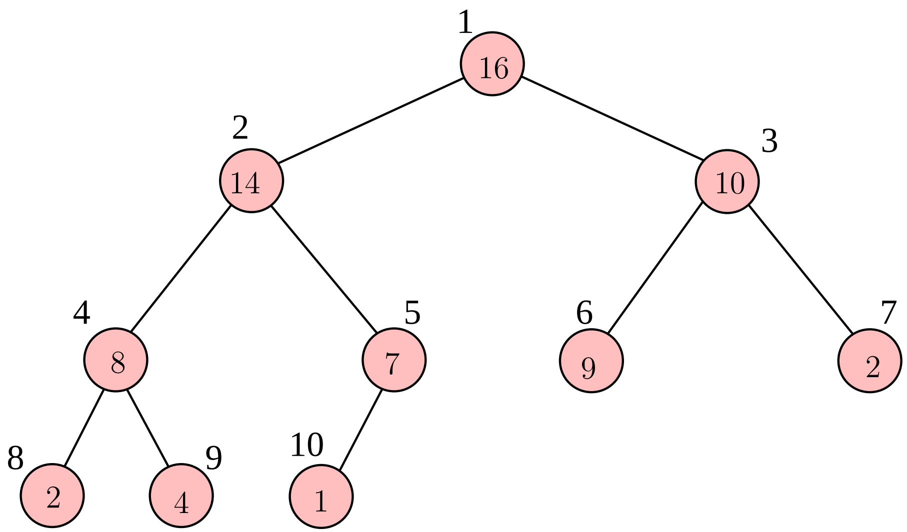
یک درخت دودویی کامل (complete binary tree) (به‌جز حداکثر یک عنصر با یک فرزند)

برگ‌های سطح آخر آن از سمت چپ چیده شده‌اند.

کلید هر عنصر از کلیدهای فرزندانش کوچک‌تر نیست.

به این داده‌ساختار درخت نیمه‌مرتب (Partially Ordered Tree)، max-heap، یا max-priority queue نیز می‌گویند

متناظر با آن min-heap است.

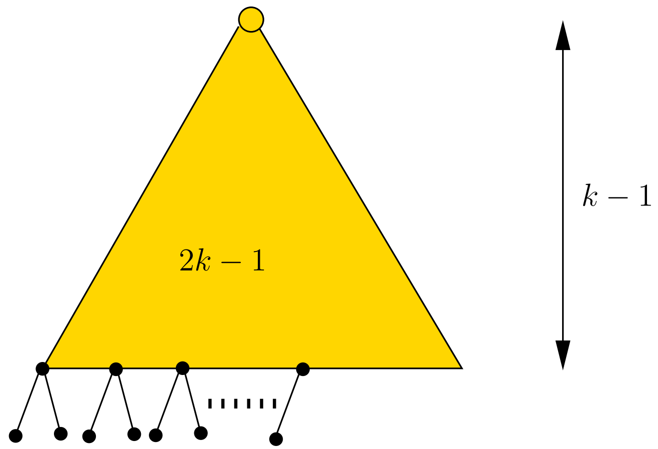


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	2	2	4	1

یک صف اولویت

max-heap (ویژگی‌ها)

- ریشه بزرگ‌ترین عنصر است.
- ارتفاع یک درخت max-heap با n عنصر $\lfloor \lg n \rfloor$ می‌باشد.
- اعمال درج، حذف بزرگ‌ترین و افزایش کلید از مرتبه‌ی $\lg n$ انجام می‌شود



ارتفاع صف اولویت با n گره

min-heap (ویژگی‌ها)

- ریشه کوچک‌ترین عنصر است.
- ارتفاع یک درخت min-heap با n عنصر $\lfloor \lg n \rfloor$ می‌باشد.
- اعمال درج، حذف کوچک‌ترین و کاهش کلید از مرتبه‌ی $\lg n$ انجام می‌شود

پیاده‌سازی max-heap

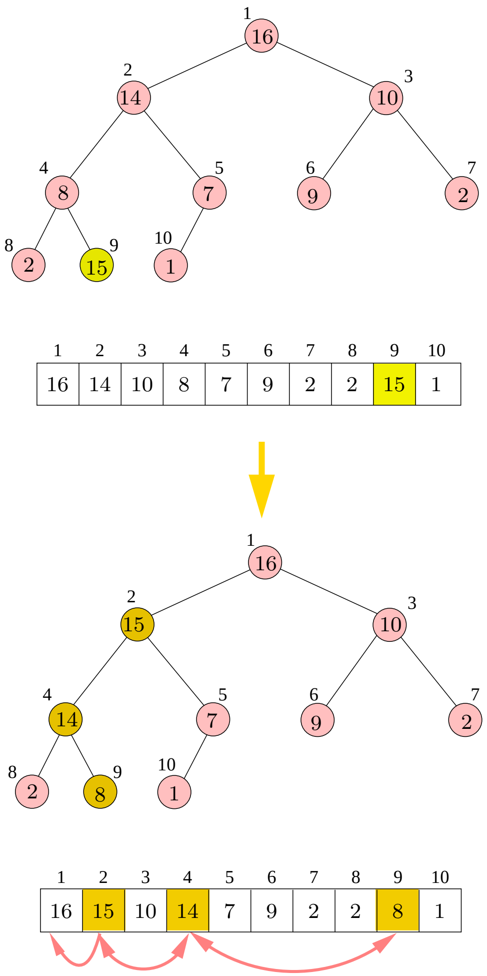
- آرایه‌ی `A[1..n]`
- ریشه در `A[1]`
- فرزند چپ عنصر iاُم در `A[2i]` (اگر $۲i \leq n$)
- فرزند راست آن در `A[2i+1]` (اگر $۲i + ۱ \leq n$)
- پدرش در $\mathtt{A}[\lfloor \frac{\mathtt{i}}{\mathtt{2}} \rfloor]$

$\underline{\textsc{Parent}}\,(i)$
1 **return** $\lfloor \frac{i}{2} \rfloor$

$\underline{\textsc{LeftChild}}\,(i)$
1 **return** $2i$

$\underline{\textsc{RightChild}}\,(i)$
1 **return** $2i + 1$

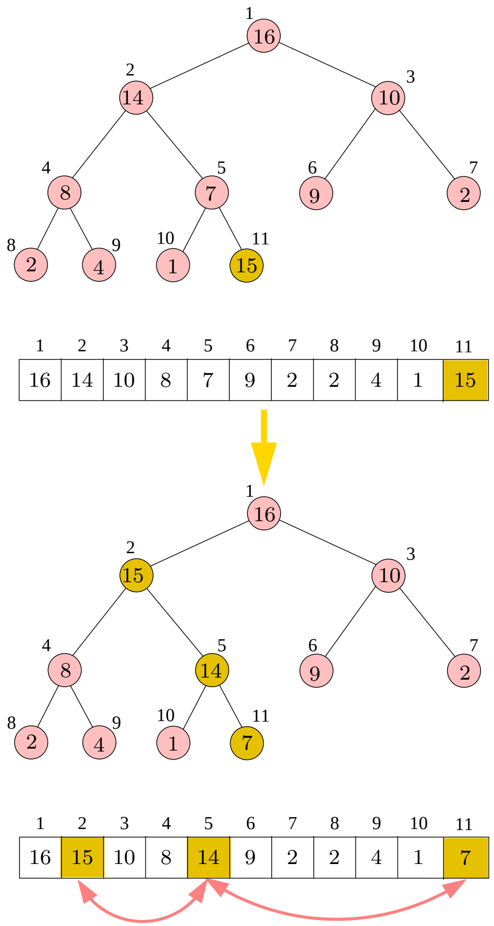
افزایش کلید


1 2 3 4 5 6 7 8 9 10
16 14 10 8 7 9 2 2 15 1
1 2 3 4 5 6 7 8 9 10
16 15 10 14 7 9 2 2 8 1

افزایش کلید

```
MAX-HEAP-INCREASE-KEY(A, i, key)
if key < A[i]
  then  error "new key is smaller than current key"
A[i] ← key
while i > 1 and A[i] > A[PARENT(i)]
      do swap(A[i], A[PARENT(i)] )
         i ← PARENT(i)]
```

درج


1 2 3 4 5 6 7 8 9 10 11
16 14 10 8 7 9 2 2 4 1 15
1 2 3 4 5 6 7 8 9 10 11
16 15 10 8 14 9 2 2 4 1 7

$\underline{\textsc{Max-Heap-Insert}}\,(A, key)$

$length[A] \leftarrow length[A] + 1$

$A[length[A]] \leftarrow -\infty$

$\textsc{Max-Heap-Increase-Key}(A, length[A], key)$

به‌صورت max-heap در آوردن $A[i..length[A]]$

به‌صورت max-heap در آوردن $A[i..length[A]]$

<u>Max-Heapify</u> (A, i)

$l \leftarrow$ LeftChild (i)
$r \leftarrow$ RightChild (i)
if $l \leq length[A]$ **and** $A[l] > A[i]$
  then $bigchild \leftarrow l$
  else $bigchild \leftarrow i$
if $r \leq length[A]$ **and** $A[r] > A[bigchild]$
  then $bigchild \leftarrow r$
if $bigchild \neq i$
  then swap$(A[i], A[bigchild])$
    Max-Heapify $(A, bigchild)$

تبدیل یک آرایه به max-heap

1	2	3	4	5	6	7	8	9	10
3	6	1	9	7	18	9	18	5	8

3	6	1	9	8	18	9	18	5	7

3	6	1	18	8	18	9	9	5	7

3	6	18	18	8	1	9	9	5	7

3	18	18	9	7	1	9	6	5	7

18	9	18	6	7	1	9	3	5	7

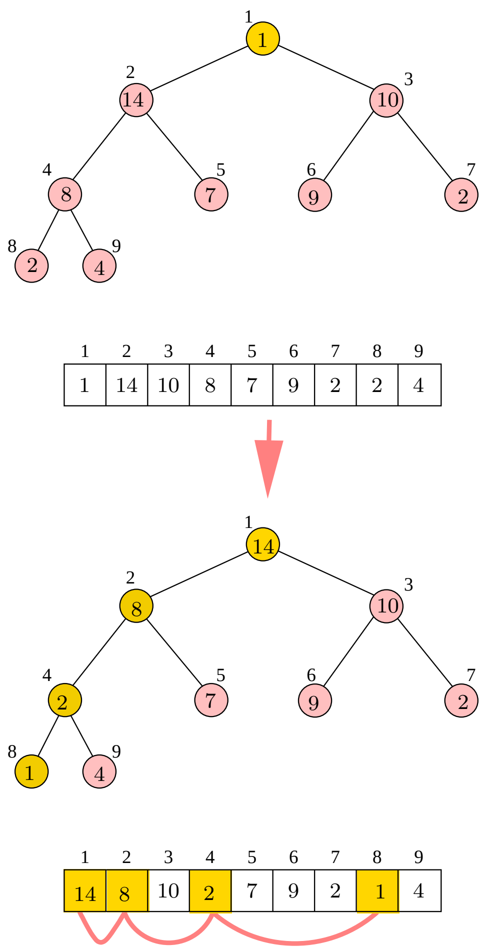
تبدیل یک آرایه به max-heap

<u>BUILD-HEAP</u> (A)

1 **for** $i \leftarrow \lfloor \frac{length[A]}{2} \rfloor$ downto 1

2 **do** HEAPIFY(A, i)

حذف بزرگ‌ترین عنصر در max-heap


1 14 10 8 7 9 2 2 4
14 8 10 2 7 9 2 1 4

حذف بزرگ‌ترین عنصر در max-heap

```
HEAP-DELETE-MAX (A)
if length[A] < 1
  then error "heap underflow"
max ← A[1]
A[1] ← A[length[A]]
A[length] ← A[length] − 1
MAX-HEAPIFY(A, 1)
return max
```

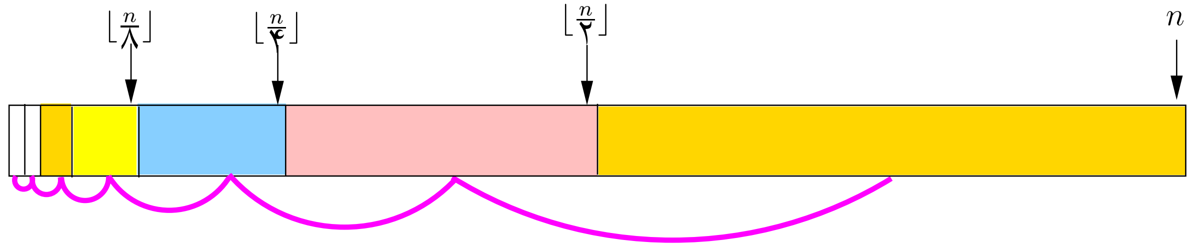
مثال

```
  1   2   3   4   5   6   7   8   9  10
+---+---+---+---+---+---+---+---+---+---+
 18  18  16   9   7   1   9   3   7   5   initial heap

 5   18  16   9   7   1   9   3   7       deletemax

 18   5  16   9   7   1   9   3   7       after heapify (A,1)

 18   9  16   5   7   1   9   3   7       after heapify (A,2)

 18   9  16   7   7   1   9   3   5       after heapify (A,4)

 1   2   3   4   5   6   7   8   9   10   11
+---+---+---+---+---+---+---+---+---+----+----+
 18  18  16   9   7   1   9   3   7    5   13  after Insert 13
 18  18  16   9  13   1   9   3   7    5    7
```

تحلیل

همه‌ی اعمال به‌جز Build-Heap متناسب با ارتفاع heap و از $O(\lg n)$

تحلیل Build-Heap



$$S(i) = \begin{cases} ۱ & i = \lfloor n/۲^۲ \rfloor + ۱ \ldots \lfloor n/۲ \rfloor \quad \text{تعداد} \quad = \lfloor n/۲^۲ \rfloor \\ ۲ & i = \lfloor n/۲^۳ \rfloor + ۱ \ldots \lfloor n/۲^۲ \rfloor \quad \text{تعداد} \quad = \lfloor n/۲^۳ \rfloor \\ \ldots & \ldots \\ k & i = \lfloor n/۲^{k+۱} \rfloor + ۱ \ldots \lfloor n/۲^k \rfloor \quad \text{تعداد} \quad = \lfloor n/۲^{k+۱} \rfloor \\ \ldots & \ldots \\ \lfloor \lg n \rfloor & i = ۱ \quad \text{تعداد} \quad = ۱ \end{cases}$$

$$T(n) \leq \sum_{k=۱}^{\lfloor \lg n \rfloor} k \frac{n}{۲^{k+۱}}$$

$$\sum_{i=۱}^{\infty} i/۲^i = (۱/۲) + (۱/۴ + ۱/۴) + (۱/۸ + ۱/۸ + ۱/۸) + ... = ۲$$

$$\begin{aligned} ۱/۲ + ۱/۴ + ۱/۸ + ۱/۱۶ + \cdots &= ۱ \\ ۱/۴ + ۱/۸ + ۱/۱۶ + \cdots &= ۱/۲ \\ ۱/۸ + ۱/۱۶ + \cdots &= ۱/۸ \\ \cdots &= .. \end{aligned}$$

پس ساخت heap از $O(n)$ است.

همین‌جا حل کنید!

k عدد کوچک‌ترین عناصر n عنصر را می‌خواهیم به‌ترتیب به‌دست آوریم.

همین‌جا حل کنید!

برای یک آرایه به‌صورت heap کارهای زیر را با چه مرتبه‌ای به‌صورت کارا می‌توان انجام داد؟

- جمع همه‌ی اعداد:
- جمع تعداد $\lg n$ بزرگ‌ترین عدد:
- جمع ۱۰ عدد بزرگ:

همین‌جا حل کنید!

برای یک آرایه به‌صورت heap کارهای زیر را با چه مرتبه‌ای به‌صورت کارا می‌توان انجام داد؟

- **جمع همه‌ی اعداد:** n
- **جمع تعداد** $\lg n$ **بزرگ‌ترین عدد: در** $\lg^{۲} n$
- **جمع ۱۰ عدد بزرگ:** $O(۱)$

heapsort

<u>BUILD-HEAP</u> (A)

for $i \leftarrow \lfloor \frac{length[A]}{2} \rfloor$ downto 1
    do HEAPIFY(A, i)

<u>HEAPSORT</u> (A)

BUILD-HEAP(A)
for $i \leftarrow length[A]$ downto 2
    do swap$(A[1], A[length[A]])$
        $length[A] \leftarrow length[A] - 1$
        HEAPIFY $(A, 1)$

	1	2	3	4	5	6	7	8	9	10
SWAP(۱،۱۰)	18	9	18	6	7	1	9	3	5	7
HEAPIFY(۱،۹)	7	9	18	6	7	1	9	3	5	18
SWAP(۱،۹)	18	9	9	6	7	1	7	3	5	18
HEAPIFY(۱،۸)	5	9	9	6	7	1	7	3	18	18
SWAP(۱،۸)	9	7	9	6	5	1	7	3	18	18
HEAPIFY(۱،۷)	3	7	9	6	5	1	7	9	18	18
SWAP(۱،۷)	9	7	3	6	5	1	7	9	18	18
HEAPIFY(۱،۶)	7	7	3	6	5	1	9	9	18	18
SWAP(۱،۶)	7	7	3	6	5	1	9	9	18	18
HEAPIFY(۱،۵)	1	7	3	6	5	7	9	9	18	18
SWAP(۱،۵)	7	6	3	1	5	7	9	9	18	18
HEAPIFY(۱،۴)	5	6	3	1	7	7	9	9	18	18
SWAP(۱،۴)	6	5	3	1	7	7	9	9	18	18
HEAPIFY(۱،۳)	1	5	3	6	7	7	9	9	18	18
SWAP(۱،۳)	5	1	3	6	7	7	9	9	18	18
HEAPIFY(۱،۲)	3	1	5	6	7	7	9	9	18	18
SWAP(۱،۲)	3	1	5	6	7	7	9	9	18	18
	1	3	5	6	7	7	9	9	18	18

```
  heapify (A,i)

  1  2  3  4  5  6  7  8  9 10
+--+--+--+--+--+--+--+--+--+--+
  3  6  1  9  7 18  9  18 5  7  initial array
  3  6  1  9  7 18  9  18 5  7  i:=5

  3  6  1 18  7 18  9  9  5  7  i:=4
         --           --
  3  6 18 18  7  1  9  9  5  7  i:=3
      --        --
  3 18 18  6  7  1  9  9  5  7  i:=2
   --      --
  3 18 18  9  7  1  6  9  5  7
         --      --
 18  3 18  9  7  1  6  9  5  7  i:=1
 -- --
 18  9 18  3  7  1  6  9  5  7
   --    --
 18  9 18  9  7  1  6  3  5  7  heap
         --           --
```

```
  1  2  3  4  5  6  7  8  9 10
+--+--+--+--+--+--+--+--+--+--+
 18  9 18  9  7  1  6  3  5  7           heap

  7  9 18  9  7  1  6  3  5| 18          swap (1,10)
 18  .  7  .  .  .  .  .  .|

  5  9  7  9  7  1  6  3 |18 18          swap (1,9)
  9  5  .  .  .  .  .  .
  .  9  .  5  .  .  .  .

  3  9  7  5  7  1  6 |9  18 18          swap(1,8)
  9  3  .  .  .  .  .
  .  7  3  .  .  .  .
  .  .  6  .  .  .  3

  3  7  6  5  7  1 |9  9  18 18          swap(1,7)
  7  3  .  .  .  .
  7  6  3  .  .  .
```

```
 1   6   3   5   7  |7   9   9   18 18          swap(1,6)
 6   1   .   .   .
 .   7   .   .   1

 1   7   3   5  |6   7   9   9   18 18          swap(1,5)
 7   1   .   .
 .   5   .   1

 1   5   3  |7   6   7   9   9   18 18          swap(1,4)
 5   1   .

 3   1  |5   7   6   7   9   9   18 18          swap(1,3)
 1  |3   5   7   6   7   9   9   18 18          swap(1,2)

|1   3   5   7   6   7   9   9   18 18          sorted
```