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 $E = V - 1$

اثبات: با استقرا: برای $n = 1$ بدیهی است که $E = 0$،

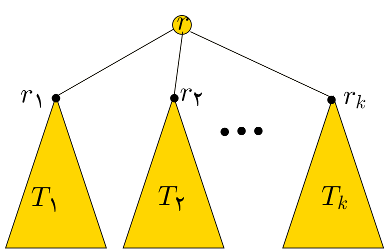
فرض استقراء: برای $V = k$ داریم $E = k - 1$

حکم استقراء: اگر $V = k + 1$

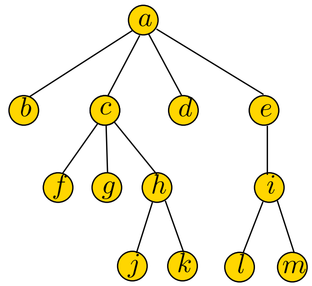
اگر بر روی یال‌های یک درخت آزاد جهت قرار دهیم به‌طوری‌که درجه ورودی حداکثر یک باشد، به آن یک درخت با رابطه‌ی «پدر فرزندی» بین رئوس گوییم.

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

تعریف درخت به صورت 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.



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 (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 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)

حداکثر تعداد فرزندان هر گره یک درخت k باشد.

درخت k تایی کامل (complete k-ary tree)

درختی است که درآن تعداد فرزندان هر گره برابر k یا صفر (برای برگ) و همه‌ی برگ‌ها در یک سطح هستند.

درخت متوازن (balanced tree)

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

درخت کاملاً متوازن (completely balanced tree)

درختی که سطح برگ‌های آن یکسان باشد .

درخت مرتب (ordered tree)

درختی است که درآن ترتیب فرزندان هر گره مهم باشد.

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 ک فرزند چپ است یا راست.

اولاد 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 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** $n - ۱ = (n - B) * k$.
- **پس** $n - B = (n - ۱)/k$ **و** $B = n - (n - ۱)/k$ **یا**

$B = [(k - ۱)n + ۱]/k$

- **یعنی** $(k - ۱)n + ۱$ **باید بر** k **بخش‌پذیر باشد.**

پیمایش درخت‌ها

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

روش پیش‌ترتیب (preorder):

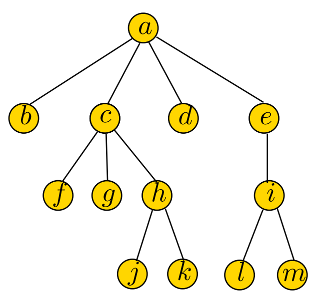
$$Pre(T) = r, Pre(T_۱), Pre(T_۲), \ldots, Pre(T_k)$$

روش بین‌ترتیب (inorder):

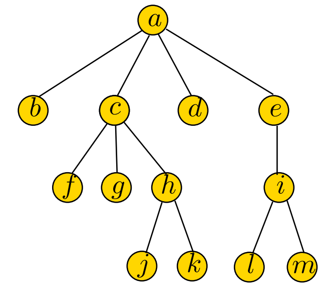
$$Inorder(T) = Inorder(T_۱), r, Inorder(T_۲), \ldots, Inorder(T_k)$$

روش پس‌ترتیب (postorder):

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

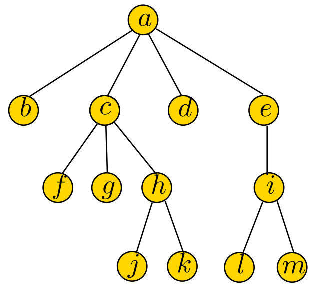


reorder(A): a, b, c, f, g, h, j, k, d, e, i, l, m



preorder(A): a, b, c, f, g, h, j, k, d, e, i, l, m

inorder(A): b, a, f, c, g, j, h, k, d, l, i, m, e

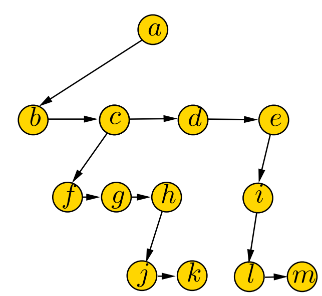


reorder(A): a, b, c, f, g, h, j, k, d, e, i, l, m

norder(A): b, a, f, c, g, j, h, k, d, l, i, m, e

ostorder(A): b, f, g, j, k, h, c, d, l, m, i, e, a

درخت دودویی معادل



اعمال بر روی درخت

- $\textsc{MakeEmpty}(T)$: یک درخت تهی T ایجاد می‌کند ($root[T]$ تهی است.)
  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

- RIGHT-SIBLING(T, v): برادر سمت راست v را در درخت T برمی‌گرداند
  ورودی: درخت و گره، خروجی: گره یا null
- SIZE(T): تعداد عناصر موجود در درخت
  ورودی: درخت، خروجی: یک عدد صحیح
- ISEMPTY(T): مشخص می‌کند که آیا درخت خالی است
  ورودی: درخت، خروجی: درست یا نادرست
- 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)
if p = null
   then return
PRINT ELEMENT(T, p)
p ← LEFT-MOST-CHILD (T, p)
while p ≠ null
        do PREORDER (T,P)
           p ← RIGHT-SIBLING (T, p)
```



```
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)
PRINT ELEMENT(T, p)
```



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

```
CountNodes(T, p)
      ▷ counts the number of nodes in T with root p
if isEmpty(T)
  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)
      ▷ returns the height of p in tree T
if isEmpty(T)
  then error
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
```

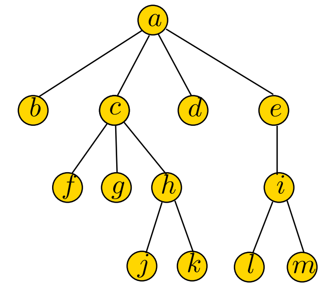

تمرین: پیاده‌سازی parent با عمل‌های دیگر:
پدر p در درخت T در زیردرختی به ریشه‌ی r

```
Find-Parent(T, r, p)
   if p = r
     then return null
q ← Left-Most-Child (T, r)
while q ≠ null
        do if p = q
             then return r
           s ← Find-Parent(T, q, p)
           if s ≠ null
             then return s
           q ← Right-Sibling (T, q)
return null
```


پیاده‌سازی درخت‌ها با آرایه

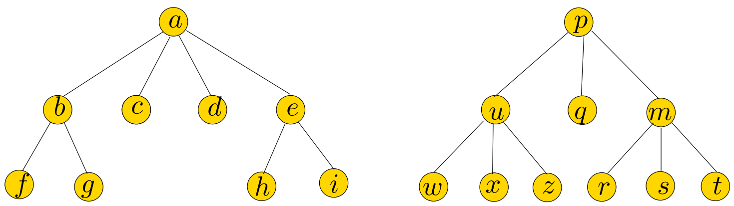
درایه‌ی ریشه: مولفه‌ی `Father` آن صفر است.

پیاده‌سازی درخت‌های مرتب ؟ ترتیب برادرها باید حفظ گردد.



1	2	3	4	5	6	7	8	9	10	11	12	13
a	b	c	d	e	f	g	h	i	j	k	l	m
0	1	1	1	1	3	3	3	5	8	8	9	9

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



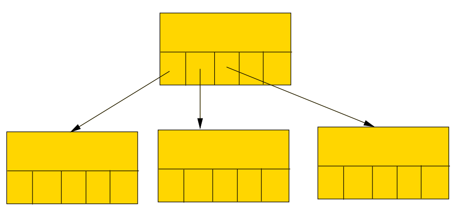
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	c	d	e	f	g	h	i	p	u	q	m	r	s	t	w	x	z
0	1	1	1	1	2	2	5	5	0	10	10	10	13	13	13	11	11	11

با استفاده از اشاره‌گرها

روش بد:

هر گره

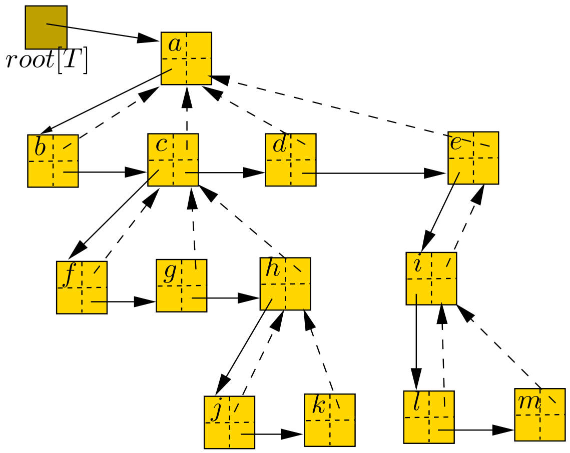
- **مولفه‌های** label،
- **یک آرایه‌ی** `Child[1..max-child]` **که** `child[i]` **به فرزند** i**ام آن گره اشاره می‌کند.**
- **مقدار** `max_child` **حداکثر تعداد فرزندان یک گره در درخت است.**



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)



پیاده‌سازی با استفاده از اشاره‌گرهای اندیسی

مقایسه‌ی دو روش

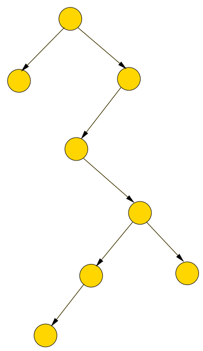
اعمال مختلف

CREATE2(x, T_1, T_2)
 ▷ creates a tree with x as the label of its root
 ▷ and two subtrees T_1 and T_2 ($T_1 \neq$ **null**)
MAKEEMPTY (T)
$root[T] \leftarrow$ *Allocate-Node* $(x, root[T_1],$ **null** , **null**)
RIGHT-SIBLING$[root[T_1]] \leftarrow root[T_2]$
$parent[root[T_1]] \leftarrow root[T]$
$parent[root[T_2]] \leftarrow root[T]$
$size[T] \leftarrow 1 + size[T_1] + size[T_2]$
return T

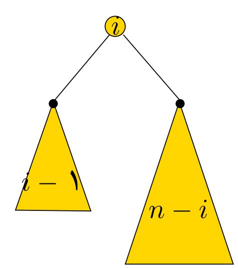
```
Create3(x, T1, T2, T3)
    MakeEmpty (T)
    root[T] ← Allocate-Node (x, root[T1], null , null )
    right-sibling[root[T1]] ← root[T2]
    right-sibling[root[T2]] ← root[T3]
    parent[root[T1]] ← root[T]
    parent[root[T2]] ← root[T]
    size[T] ← 1 + size[T1] + size[T2] + size[T3]
    return T
```


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



$$T(n) = \begin{cases} T(0) = 1 \\ T(n) = \Sigma_{i=1}^{n} T(i-1)T(n-i), \; n \geq 1 \end{cases}$$

جواب این رابطه‌ی بازگشتی $T(n) = \frac{1}{n+1}\binom{2n}{n}$ **(عدد nام کاتالان)**


```
Preorder(T, r)
    ▷ It is assumed that r is the root of T
    if r = null
        then return
    meet element(T, r)
    Preorder(T, left[r])
    Preorder(T, right[r])
```

درخت عبارت (Expression Tree)

گونه‌های عبارت:

میانوندی با پرانتزی کامل (infix with complete paranthesis)

$$\begin{array}{rcl} E & \rightarrow & (E_1 < \beta > E_2) \\ & \rightarrow & (< \alpha > E) \\ & \rightarrow & < \texttt{operand} > \\ < \alpha > & \rightarrow & \texttt{unary operators} \\ < \beta > & \rightarrow & \texttt{binary operators} \end{array}$$

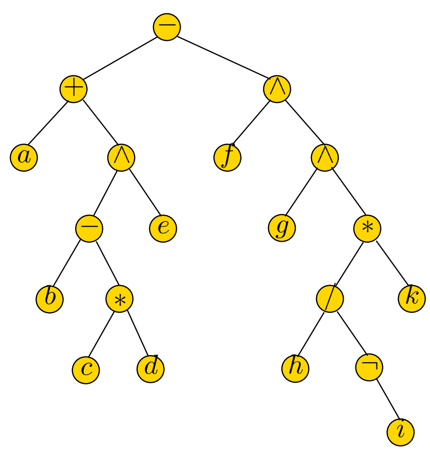
پسوندی (postfix)

$$\begin{array}{rcl} E & \rightarrow & E_1 E_2 < \beta > \\ & \rightarrow & E < \alpha > \\ & \rightarrow & < \texttt{operand} > \end{array}$$

پیشوندی (prefix)

$$\begin{array}{rcl} E & \rightarrow & < \beta > E_1 E_2 \\ & \rightarrow & < \alpha > E \\ & \rightarrow & < \texttt{operand} > \end{array}$$

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



فرم عبارت	عبارت
میانوندی	$a+(b-c*d)\wedge -f\wedge g\wedge (h/\neg i*k)$
میانوندی با پرانتز کامل	$((a+((b-(c*d))\wedge e)-(f\wedge (g\wedge ((h/(\neg i))*k))))$
پسوندی	$abcd*-e\wedge +fghi\neg /k*\wedge \wedge -$
پیشوندی	$-+a\wedge -b*cde\wedge f\wedge g*/h\neg ik$

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	$-f\wedge g\wedge (h/\neg i*k)$
$abcd*-e$	$+\wedge$	$-$	$f\wedge g\wedge (h/\neg i*k)$

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

on the top of stack

		(	−	+	×	/	∧	¬
	(	Push	Push	Push	Push	Push	Push	Push
i	−	Push	Pop	Pop	Pop	Pop	Pop	Pop
n	+	Push	Pop	Pop	Pop	Pop	Pop	Pop
p	∗	Push	Push	Push	Pop	Pop	Pop	Pop
u	/	Push	Push	Push	Pop	Pop	Pop	Pop
t	∧	Push	Push	Push	Push	Push	Push	Pop
	¬	Push	Push	Push	Push	Push	Push	Pop
	)	Pop-more	Pop	Pop	Pop	Pop	Pop	Pop

جدول Action[i,j]

```
INFIX-TO-POSTFIX(infix)
      ▷ Uses stack S, and matrix action
intialize-actions()
while there is token in infix
        do read token c from infix
            if c is an operand
               then write c to postfix
               else   done ← false
                      while  not done
                             do if isEMPTY(S)
                                   then Push (c, S)
                                          done ← true
                                   else   s ← TOP(S)
                                          if c =')'  and s =' ('
                                             then POP(S)
                                                    done ← true
                                          if action[c, s] =' push'
                                             then Push (S, c)
```



```
                                                    done ← true
                                             else   write s to postfix
                                                    POP(S)
while  not isEMPTY(S)
       do write TOP(S) to postfix
           POP(S)
```



```
POSTFIX-TO-PREFIX(i, j)
      ▷ converts correct postfix[?..i] to prefix[?..j]
      ▷ at the end, i and j are either 0 or
      ▷ the last index of the preceeding expressions.
      ▷ postfix and prefix are global arrays
▷ i , j are refrenced variables switch
        case postfix[i] is operand
               do prefix[j] ← postfix[i]
                   i ← i − 1; j ← j − 1
        case  postfix[i] is binary operator
               do operator ← postfix[i]; i ← i − 1
                   POSTFIX-TO-PREFIX (i, j)
                   POSTFIX-TO-PREFIX (i, j)
                   prefix[j] ← operator
                   j ← j − 1
        case  postfix[i] is unary operator
```



```
               do operator ← postfix[i]; i ← i − 1
                   POSTFIX-TO-PREFIX (i, j)
                   prefix[j] ← operator
                   j ← j − 1
```

```
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
```



```
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
```