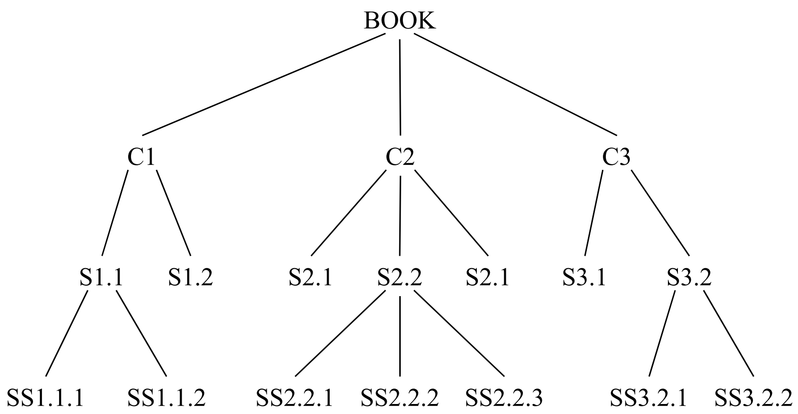
درخت (TREE)

سید ناصر رضوی
email: razavi@comp.iust.ac.ir
۱۳۸۶

درخت

- تعدادی عنصر که بین آنها رابطه پدر-فرزندی برقرار است. در درخت هر عنصر به غیر از ریشه دقیقاً یک پدر دارد.
- تعریف رسمی: یک درخت مجموعه محدودی از یک یا چند گره به صورت زیر می باشد:
  - دارای گره خاصی به نام **ریشه** است.
  - بقیه گره ها به $k \geq 0$ مجموعه مجزا مانند T_1، T_2، ... و T_k تقسیم می شوند به گونه ای که هریک از این مجموعه ها خود یک درخت می باشند. (T_1، T_2، ... و T_k را **زیردرخت** های ریشه می گوییم.)

مثال



تعاریف پایه ای

- مسیر: دنباله ای از گره ها مانند $n_1, n_2, \ldots, n_k$ می باشد که در این دنباله به ازای هر $1 \leq i < k$ از گره n_i به گره n_{i+1} یک شاخه و جود دارد.
- طول مسیر: برابر است با تعداد شاخه های موجود در آن مسیر (یا تعداد گره های موجود در مسیر منهای یک).
- مسیر بدیهی: مسیری است که طول آن برابر صفر می باشد.
  - بنابراین از هر کره به خود آن گره یک مسیر بدیهی وجود دارد.

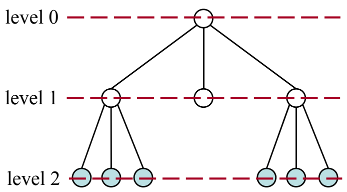
تعاریف پایه ای

- جد (ancestor) و نسل (descendant): اگر در درخت مسیری از گره a به گره b وجود داشته باشد، در این صورت a را **جد** b و b را **نسل** a می گوییم.
- پدر (parent) و فرزند (child): اگر در درخت مسیری به طول یک از گره a به گره b وجود داشته باشد، در این صورت a را **پدر** b و b را **فرزند** a می گوییم.
- گره های sibling (همزاد، هم نیا): گره هایی که پدر آنها یکسان باشد.

تعاریف پایه ای

- درجه گره: برابر است با تعداد فرزندان آن گره.
- درجه درخت: برابر است با حداکثر درجه گره های موجود در آن درخت.
- گره برگ (گره خارجی): گره ای که درجه آن برابر با صفر باشد (گره ای که تعداد فرزندانش برابر با صفر باشد).
- گره غیربرگ (گره داخلی): گره ای که درجه آن بزرگتر از صفر باشد (گره ای که حداقل یک فرزند داشته باشد).

تعاریف پایه ای



- سطح (عمق) گره: برابر است با طول مسیر از ریشه تا آن گره.
- ارتفاع گره: برابر است با حداکثر طول یک مسیر از آن گره تا یکی از برگ های اولاد.
- ارتفاع درخت: برابر است با ارتفاع ریشه (حداکثر طول یک مسیر از ریشه تا یکی از برگ های درخت)

تعاریف پایه ای

- درخت k-تایی: درختی است که در آن درجه هر گره حداکثر برابر با k باشد.
- قضیه: در هر درخت k-تایی داریم:

$$n_0 = (k-1)\, n_k + (k-2)\, n_{k-1} + \dots + 2n_3 + n_2 + 1$$

$B = n - 1 = n_0 + n_1 + \dots + n_k - 1$

$B = 0 \times n_0 + 1 \times n_1 + 2 \times n_2 \dots + k \times n_k$

$n_0 = (k-1)\, n_k + (k-2)\, n_k\text{-}1 + \dots + 2n_3 + n_2 + 1$

نتیجه: در هر درخت k-تایی تعداد برگ ها مستقل از تعداد گره های تک فرزندی می باشد.

تعاریف پایه ای

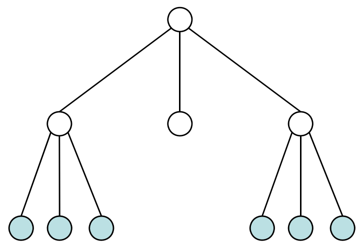
- درخت k-تایی کامل: درختی است که در آن درجه هر گره یا برابر با k و یا برابر با صفر باشد.
- قضیه: در هر درخت k-تایی کامل داریم:

$$n_0 = (k-1)\, n_k + 1$$

$B = n - 1 = n_0 + n_k - 1$

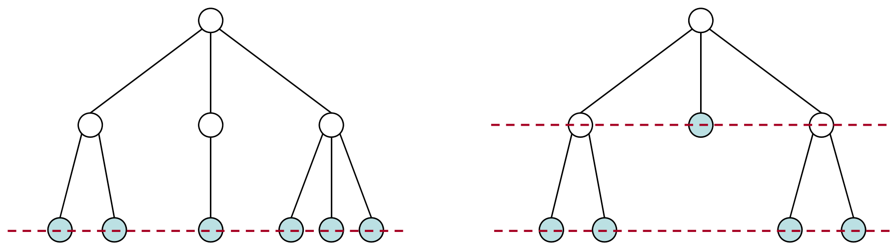
$B = 0 \times n_0 + k \times n_k$

$n_0 = (k-1)\, n_k + 1$



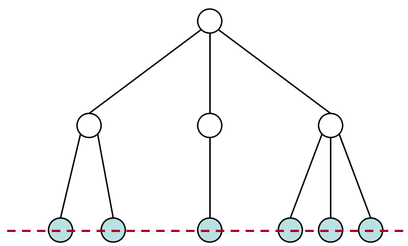
تعاریف پایه ای

- درخت متوازن: درختی است که در آن اختلاف سطح برگ ها حدکثر برابر با یک باشد.



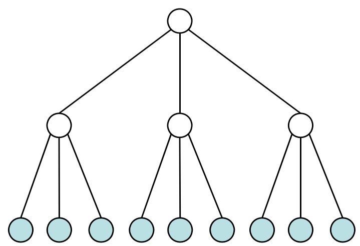
تعاریف پایه ای

- درخت کاملاً متوازن: درختی است که در آن اختلاف سطح برگ ها دقیقاً برابر با صفر باشد.



تعاریف پایه ای

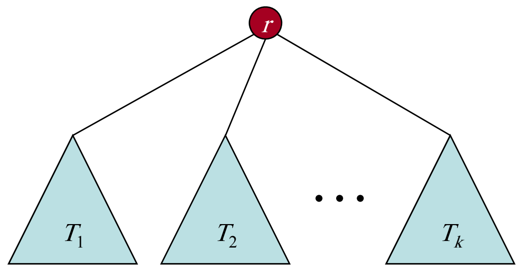
- درخت پر: درختی است که هم کامل باشد و هم کاملاً متوازن.



- تعداد کل گره ها در یک درخت پر با درجه d و ارتفاع h برابر است با:

$$n = \sum_{i=0}^{h} d^i = d^0 + d^1 + \cdots + d^h = \frac{d^{h+1}-1}{d-1}$$

روش های پیمایش درخت

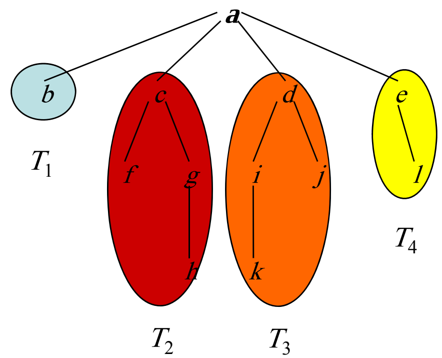


Preorder (T): r, Pre (T_1), Pre (T_2), …, Pre (T_k)

Inorder (T): In (T_1), r, In (T_2), …, In (T_k)

Postorder (T): Post (T_1), Post (T_2), …, Post (T_k), r

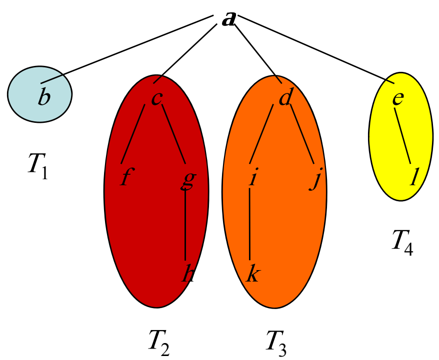
مثال: پیمایش پیش ترتیب



Preorder: ***a*** | *b* | *c f g h* | *d i k j* | *e l*

| | T_1 | T_2 | T_3 | T_4 |

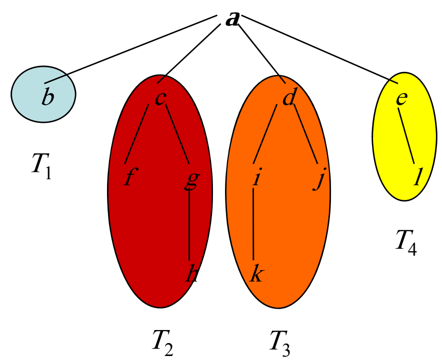
مثال: پیمایش میان ترتیب



Inorder: *b* | ***a*** | *f c h g* | *k i d j* | *l e*

T_1 T_2 T_3 T_4

مثال: پیمایش پس ترتیب



Postorder: *b* | *f h g c* | *k i j d* | *l e* | ***a***

T_1 T_2 T_3 T_4

نوع داده انتزاعی درخت (TREE ADT)

- انواع داده ای

- T: TREE
- n: node

- عملیات درخت

- MAKENULL (T)
- ROOT (T)
- PARENT (n, T)
- LABEL (n, T)
- LEFT-MOST-CHILD (n, T)
- RIGHT-SIBLING (n, T)
- CREATE$_k$(r, T_1, T_2, …, T_k)

پیمایش درخت عمومی به صورت پیش ترتیب

```
procedure PRE-ORDER (n: node);
var
    c: node;

begin
    print (LABEL (n, T));
    c := LEFT-MOST-CHILD (n, T);
    while ( c <> NULL ) do begin
        PRE-ORDER (c);
        c := RIGHT-SIBLING (c, T)
    end
end;
```

$T(n) \in \Theta(n)$

پیمایش درخت عمومی به صورت میان ترتیب

```
procedure IN-ORDER (n: node);
var
    c: node;

begin
    c := LEFT-MOST-CHILD (n, T);
    if ( c <> NULL ) then begin
        IN-ORDER (c);
        print (LABEL (n, T));
        c := RIGHT-SIBLING (c, T)
        while ( c <> NULL ) do begin
            IN-ORDER (c);
            c := RIGHT-SIBLING (c, T)
        end
    else
    print (LABEL (n, T));
end;
```

$T(n) \in \Theta(n)$

پیمایش درخت عمومی به صورت پس ترتیب

```
procedure POST-ORDER (n: node);
var
    c: node;

begin
    c := LEFT-MOST-CHILD (n, T);
    while ( c <> NULL ) do begin
        POST-ORDER (c);
        c := RIGHT-SIBLING (c, T)
    end;
    print (LABEL (n, T))
end;
```

$T(n) \in \Theta(n)$

تمرین: پیمایش به ترتیب سطح (level order)

- رویه ای برای پیمایش یک درخت به ترتیب سطح بنویسید. (راهنمایی: از صف استفاده نمایید).

```
procedure LEVEL-ORDER (n: node);
var
    c: node;
    Q: QUEUE;
begin
    MAKENULL (Q);
    ENQUEUE (n, Q);
    while not EMPTY (Q) do begin
        n := DEQUEUE (Q);
        print ( LABEL (n, T) );
        c := LEFT-MOST-CHILD (n, T);
        while ( c <> NULL ) do begin
            ENQUEUE (c, Q);
            c := RIGHT-SIBLING (c, T)
        end
end;
```


پیاده سازی درخت عمومی (روش بد)

- پیاده سازی گره ها:
  - یک فیلد برای ذخیره اطلاعات (برچسب)
  - k فیلد اشاره گر برای ذخیره آدرس فرزندان

label فیلد اشاره گر k

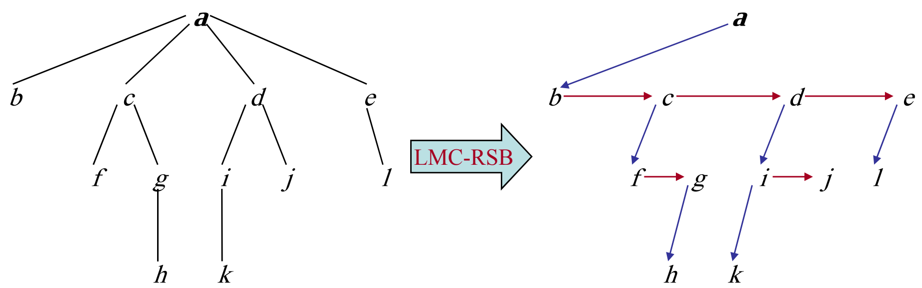
	Child 1	Child 2	Child 3	…	Child k

- عیب این روش پیاده سازی
  - تعداد کل فیلدهای اشاره گر به فرزندان: nk
  - تعداد فیلدهای اشاره گر غیر **nil**: $n-1$
  - بنابراین تعداد فیلدهای اشاره گر **nil** برابر است با:

$$nk-(n-1)=n(k-1)+1$$

پیاده سازی درخت عمومی: چپ ترین فرزند-برادر راست (LMC-RSB)

- در این روش، هر گره دارای ۲ فیلد آدرس می باشد:
  - یکی برای ذخیره آدرس چپ ترین فرزند
  - یکی برای ذخیره آدرس نزدیک ترین برادر راست

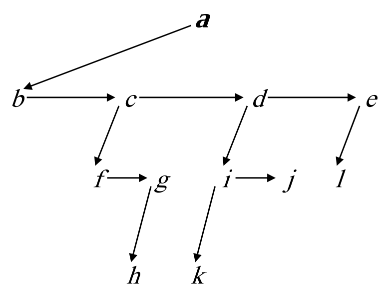


مثال: پیاده سازی LMC-RSB

T: 5 → 5

	Label	LMC	RSB
1	f	NULL	2
2	g	10	NULL
3	c	1	11
4		NULL	NULL
5	a	9	NULL
6	k	NULL	NULL
7		NULL	NULL
8	e	12	NULL
9	b	NULL	3
10	h	NULL	NULL
11	d	13	8
12	l	NULL	NULL
13	i	6	14
14	j	NULL	NULL
⋮			
MAX			

cellspace



پیاده سازی LMC-RSB

```
const
    MAX = …;
    NULL = 0;

type
    TREE = 0 .. MAX;
    node = TREE;

var
    cellspace : array [1..MAX] of record
        label: labeltype;
        LMC, RSB: node
    end;
```


پیاده سازی عملیات LMC-RSB

```
procedure MAKENULL (var T: TREE);
begin
    T := NULL
end;
```

$$T(n) \in \Theta(1)$$

پیاده سازی عملیات LMC-RSB

```
function ROOT (T: TREE): node;
begin
    return (T)
end;
```

$$T(n) \in \Theta(1)$$

پیاده سازی عملیات LMC-RSB

```
function LABEL (n: node; T: TREE): labeltype;
begin
    return (cellspace [n].lable)
end;
```

$$T(n) \in \Theta(1)$$

پیاده سازی عملیات LMC-RSB

```
function LEFT-MOST-CHILD (n: node; T: TREE): node;
begin
    return (cellspace [n].LMC)
end;

function RIGHT-SIBLING (n: node; T: TREE): node;
begin
    return (cellspace [n].RSB)
end;
```

$$T(n) \in \Theta(1)$$

پیاده سازی عملیات LMC-RSB

```
function PARENT (n: node; T: TREE): node;
var
    c, p: node;

begin
    if ( n = ROOT (T) ) then return (NULL);
    c := LEFT_MOST_CHILD ( ROOT(T), T);
    while ( c <> NULL) do begin
        if (n = c ) then return ( ROOT (T) );
        p := PARENT (n, c);
        if ( p <> NULL ) then return (p);
        c := RIGHT-SIBLING (c, T)
    end;
    return ( NULL )
end;
```

$$T(n) \in \Theta(n)$$

پیاده سازی عملیات LMC-RSB

```
function CREATE2 (r: labeltype; T1, T2: TREE): TREE;
var
    T: TREE;

begin
    MYNEW (T);
    cellspace [T].label = r;
    cellspace [T].LMC = T1;
    cellspace [T].RSB = NULL;
    cellspace [T1].RSB = T2;
    return ( T )
end;
```

$$T(n) \in \Theta(n)$$

مثال: محاسبه تعداد گره های یک درخت LMC-RSB

```
function SIZE (n: node): integer;
var
    count: integer;
    c: node;

begin
    count := 1;
    c := LEFT-MOST-CHILD (n, T);
    while ( c <> NULL ) do begin
        count := count + SIZE (c);
        c := RIGHT-SIBLING (c, T)
    end;
    return (count)
end;
```

مثال: محاسبه ارتفاع یک درخت LMC-RSB

```
function HEIGHT (n: node): integer;
var
    h: integer;
    c: node;

begin
    h := 0;
    c := LEFT-MOST-CHILD (n, T);
    if ( c = NULL ) then return (0);
    while ( c <> NULL ) do begin
        h := MAXIMUM (h, HEIGHT (c));
        c := RIGHT-SIBLING (c, T)
    end;
    return (h + 1)
end;
```

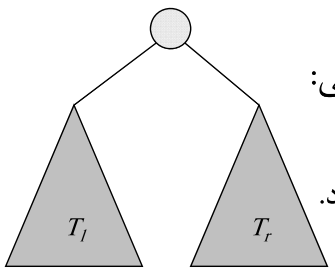

تمرین

- تابعی برای محاسبه تعداد برگ های یک درخت LMC-RSB بنویسید.

درخت دودویی (Binary Tree)

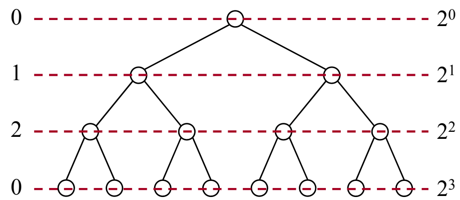
- تعریف: یک **درخت دودویی** یا تهی است و یا حاوی مجموعه محدودی از گره ها شامل ریشه و دو درخت دودویی است. این درخت ها، زیردرخت های چپ و راست نامیده می شوند.

- تفاوت های درخت دودویی با درخت عمومی:
  - درخت دودویی می تواند تهی باشد.
  - در درخت دودویی ترتیب فرزندان اهمیت دارد.



خواص درخت های دودویی

- حداکثر تعداد گره ها در سطح i ام برابر است با: 2^i



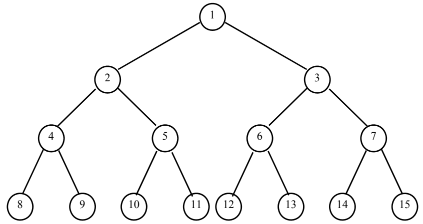
- حداکثر تعداد گره ها در یک درخت دودویی با ارتفاع h برابر است با : $2^{h+1} - 1$

$$n = \sum_{i=0}^{h} 2^i = 2^0 + 2^1 + \cdots + 2^h = \frac{2^{h+1}-1}{2-1} = 2^{h+1} - 1$$

- رابطه بین تعداد برگ ها و تعداد گره های دو فرزندی:

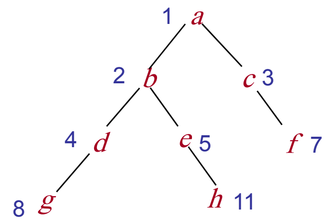
$$n_0 = n_2 + 1$$

نمایش درخت دودویی به وسیله آرایه

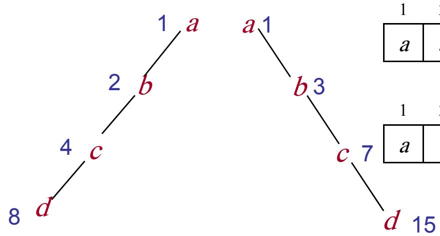


- ریشه در خانه ۱ قرار دارد.
- اگر یک گره در خانه i ام باشد، آنگاه:
  - پدر آن گره در خانه $\lfloor i/2 \rfloor$ $(i > 1)$
  - فرزند چپ آن در خانه $2i$ $(2i \le n)$
  - فرزند راست آن در خانه $2i + 1$ $(2i + 1 \le n)$

نمایش درخت دودویی به وسیله آرایه



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



درخت مورب چپ   درخت مورب راست

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

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

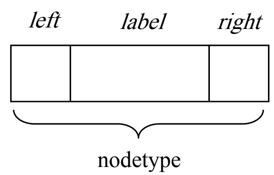
معایب پیاده سازی درخت دودویی به وسیله آرایه

- این روش فقط برای درخت های دودویی پر و تقریباً پر مناسب است. در بدترین حالت، یک درخت مورب به ارتفاع h به 2^{h+1}-1 خانه نیاز دارد که از این تعداد فقط $h + 1$ خانه استفاده می شود و بقیه خال می مانند.

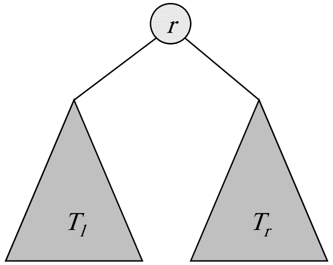
- در این روش، درج و حذف گره ها نیاز به جابجایی بقیه گره ها دارد که باعث کندی عمل درج و حذف می گردد.

پیاده سازی درخت دودویی به وسیله اشاره گر

```
type
    TREE = ^nodetype;
    node = TREE;
    nodetype = record
        label: labeltype;
        left, right: node
    end;
```



روش های پیمایش درخت دودویی



Preorder (T): r, Pre (T_l), Pre (T_r)

Inorder (T): In (T_l), r, In (T_r)

Postorder (T): Post (T_l), Post (T_r), r

پیمایش درخت دودویی به روش پیش ترتیب

```
procedure PRE-ORDER (n: node);
begin
   if ( n <> nil ) then begin
      write (n^.label);
      PRE-ORDER (n^.left);
      PRE-ORDER (n^.right)
   end
end;
```

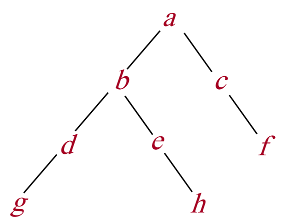

پیمایش درخت دودویی به روش میان ترتیب

```
procedure IN-ORDER (n: node);
begin
   if ( n <> nil ) then begin
      IN-ORDER (n^.left);
      write (n^.label);
      IN-ORDER (n^.right)
   end
end;
```


پیمایش درخت دودویی به روش پس ترتیب

```
procedure POST-ORDER (n: node);
begin
   if ( n <> nil ) then begin
      POST-ORDER (n^.left);
      POST-ORDER (n^.right);
      write (n^.label)
   end
end;
```

مثال: پیمایش درخت دودویی



Preorder (*T*): *a, b, d, g, e, h, c, f*

Inorder (*T*): *g, d, b, e, h, a, c, f*

Postorder (*T*): *g, d, h, e, b, f, c, a*

پیمایش میان ترتیب درخت دودویی به صورت غیر بازگشتی به کمک پشته

```
procedure NR-INORDER (n: node);
var
    S: STACK;
    done: boolean;
begin
    MAKENULL (S);
    done := false;
    repeat
        while ( n <> nil ) do begin
            PUSH (n, S);
            n := n^.left
        end;
        if not EMPTY (S) then begin
            n := TOP (S); POP (S);
            write (n^.label);
            n := n^.right
        end else
            done := true;
    until done
end;
```

$$T(n) \in \Theta(n)$$

تمرین: پیایش به ترتیب سطح (level order)

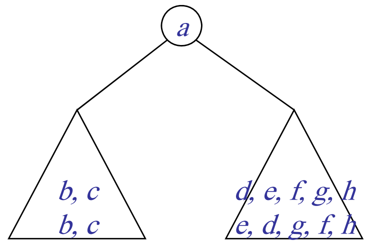
- رویه ای برای پیمایش یک درخت دودویی به ترتیب سطح بنویسید.

```
procedure LEVEL-ORDER (n: node);
var
    Q: QUEUE;

begin
    MAKENULL (Q);
    ENQUEUE (n, Q);
    while not EMPTY (Q) do begin
        n := DEQUEUE (Q);
        write ( n^.label );
        if ( n^.left <> nil )   then ENQUEUE (n^.left, Q);
        if ( n^.right <> nil ) then ENQUEUE (n^.right, Q)
end;
```


مسأله: رسم درخت دودویی به کمک پیمایش های آن

Pre: $a, \underbrace{b, c}, \underbrace{d, e, f, g, h}$

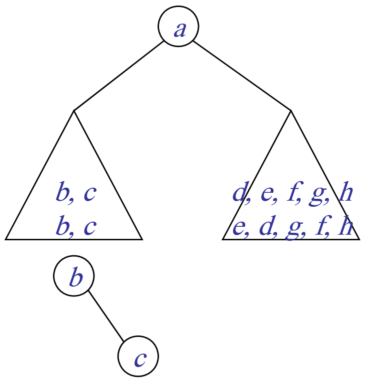
In: $\underbrace{b, c}, a, \underbrace{e, d, g, f, h}$



مسأله: رسم درخت دودویی به کمک پیمایش های آن

Pre: *b, c*

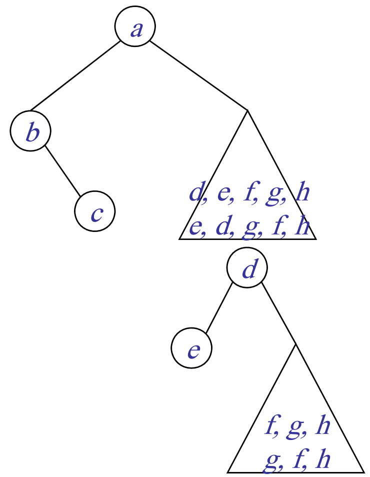
In: *b, c*



مسأله: رسم درخت دودویی به کمک پیمایش های آن

Pre: *d, e, f, g, h*

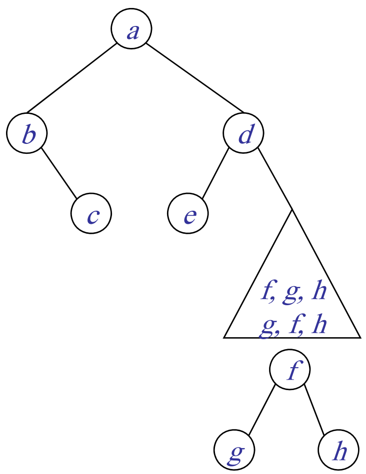
In: *e, d, g, f, h*



مسأله: رسم درخت دودویی به کمک پیمایش های آن

Pre: *f, g, h*

In: *g, f, h*

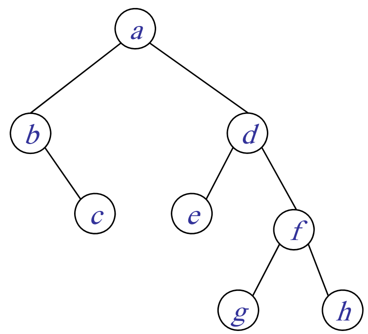


مسأله: رسم درخت دودویی به کمک پیمایش های آن

Pre: *a, b, c, d, e, f, g, h*

In: *b, c, a, e, d, g, f, h*

Post: *c, b, e, g, h, f, d, a*



بنابراین با داشتن پیمایش Inorder و حداقل یکی از پیمایش های دیگر درخت دودویی قابل ترسیم است.

اما با داشتن پیمایش Preorder و Postorder درخت دودویی به صورت یکتا قابل ترسیم نیست. چرا؟

مثال: کپی کردن یک درخت دودویی

```
function COPY (n: node): TREE;
var
    T: TREE;

begin
    if n = nil then return (nil);
    new (T);
    T^.label := n^.label;
    T^.left := COPY (n^.left);
    T^.right := COPY (n^.right);
    return (T)
end;
```


تمرین

۱. تابعی به صورت زیر به منظور بررسی نمودن تساوی دو درخت دودویی بنویسید.

function EQUAL-TREE ($n1$, $n2$: node): boolean;

۲. تابعی به صورت زیر بنویسید به طوری که برای هر گره در درخت دودویی، جای دو زیر درخت چپ و راست آن گره را با هم تعویض نماید.

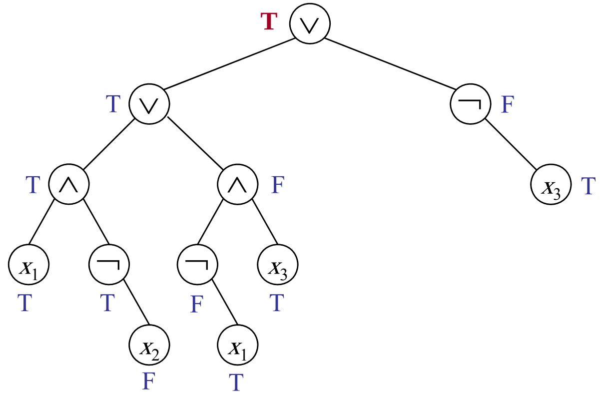
function SWAP-TREE (n: node): TREE;

تمرین

۳. رویه ای برای ارزشیابی یک فرمول گزاره ای بنویسید. (راهنمایی: رویه postorder را اصلاح نمایید.

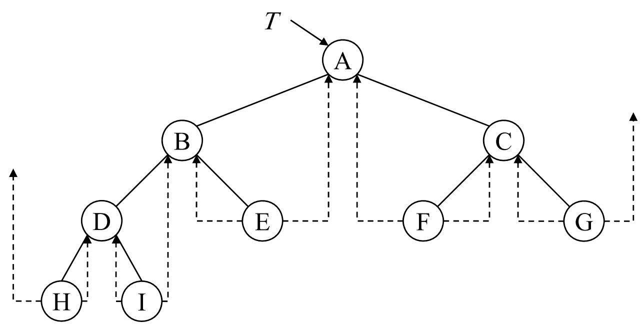
$$(x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_3) \vee \neg x_3$$

x_1	True
x_2	False
x_3	True

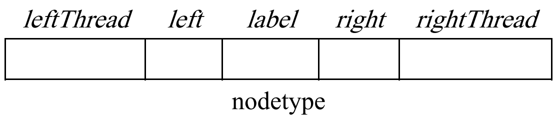


درخت نخی دودویی

- انگیزه: استفاده هوشمند از فیلدهای اشاره گر nil
  - در یک درخت دودویی با n گره، کلاً $2n$ فیلد اشاره گر وجود دراد به طوری که $n-1$ فیلد غیر nil و $n+1$ فیلد دارای مقدار nil می باشند.
  - هر فیلد nil می تواند به گره بعدی (در یک پیمایش خاص) اشاره کند.

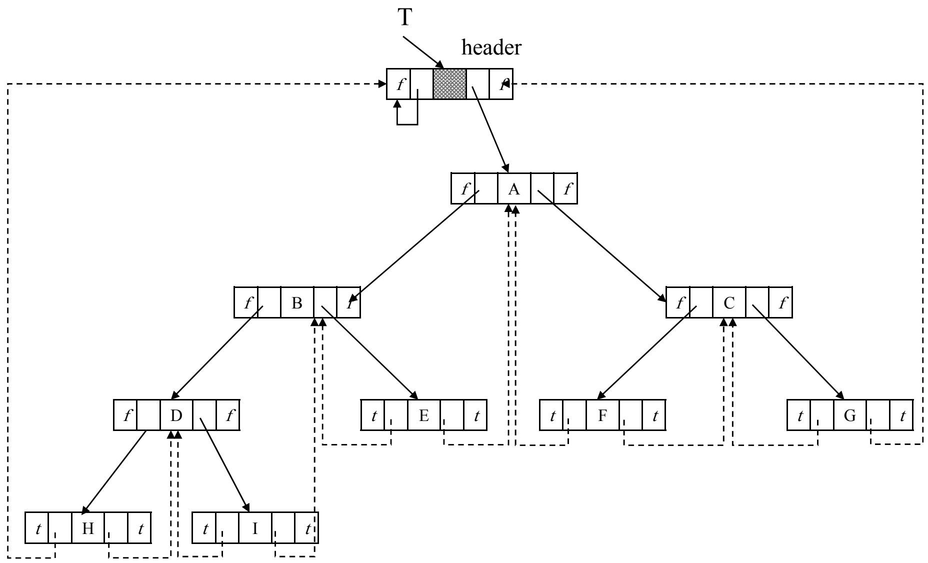


پیاده سازی درخت نخی دودویی



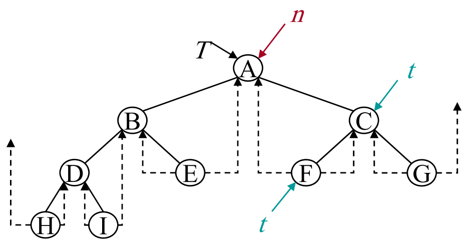

```
type
    TREE = ^nodetype;
    node = ^nodetype;
    nodetype = record
        leftThread: boolean;
        left: node;
        label: labeltype;
        right: node;
        rightThread: boolean
    end;
```


پیاده سازی درخت نخی دودویی



پیمایش میان ترتیب درخت نخی دودویی

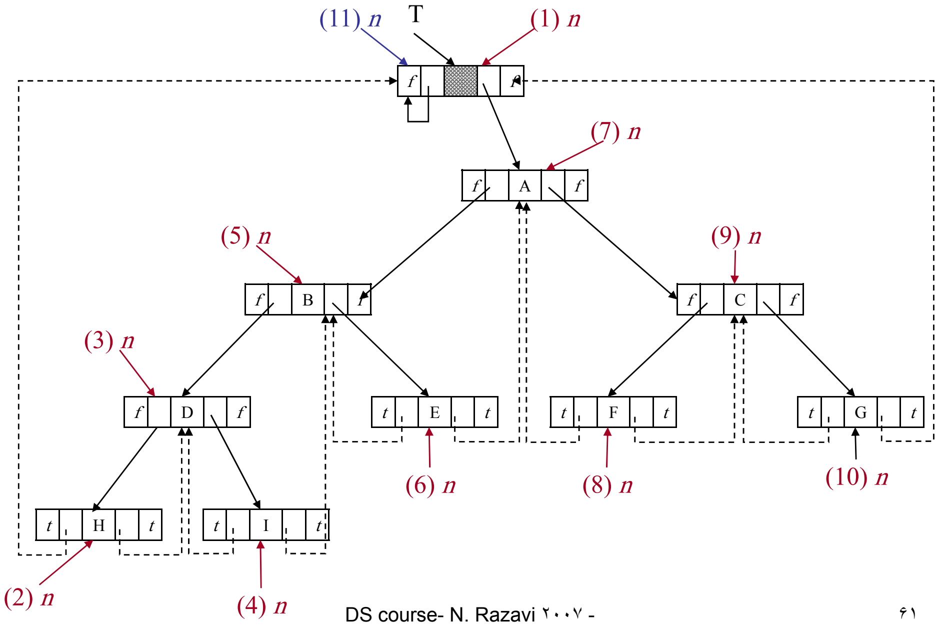
```
function INORDER-SUCC (n: node): node;
var
    t: node;

begin
    t := n^.right;
    if not n^.rightThread then
        while not t^.leftThread do
            t := t^.left;
    return (t)
end;
```



پیمایش میان ترتیب درخت نخی دودویی

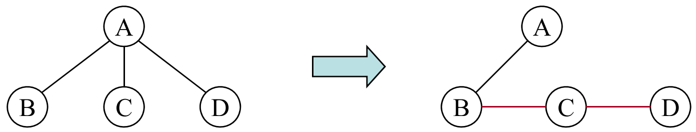
```
function INORDER-THREAD (T: TREE): node;
var
    n: node;

begin
    n := T;
    repeat
        n := INORDER-SUCC (n);
        if ( n <> T) then
            write (n^.label);
    until n = T
end;
```

مثال: پیمایش درخت نخی دودویی

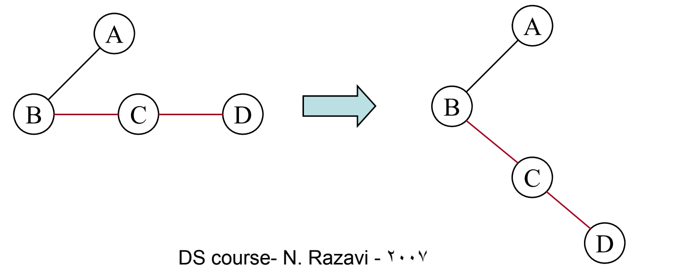


تبدیل درخت عمومی به درخت دودویی

۱. تبدیل درخت عمومی به شکل LMC-RSB

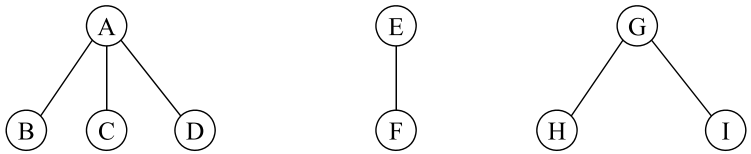


۲. چرخاندن اتصالات RSB به اندازه ۴۵ درجه در جهت گردش عقربه های ساعت



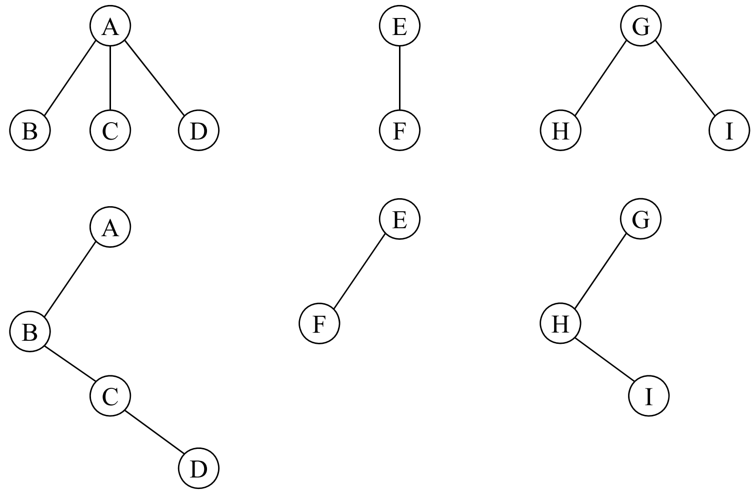
جنگل (forest)

- تعریف: جنگل یک مجموعه مرتب از $k \geq 0$ درخت مجزا است.



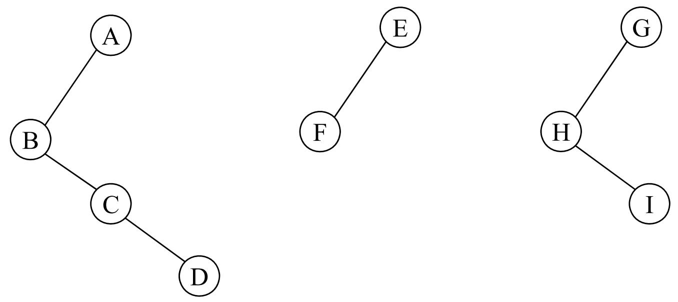
تبدیل جنگل به درخت دودویی

۱. تبدیل هر درخت در جنگل به یک درخت دودویی



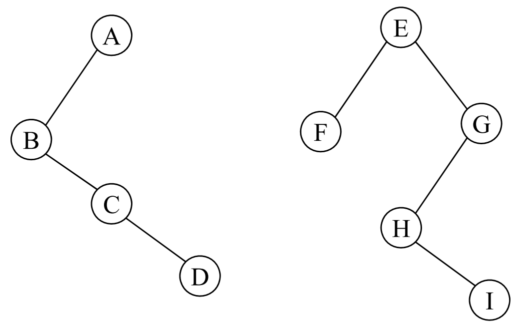
تبدیل جنگل به درخت دودویی

۲. قرار دادن هر یک از درخت های دودویی به عنوان زیردرخت راست ریشه درخت قبلی



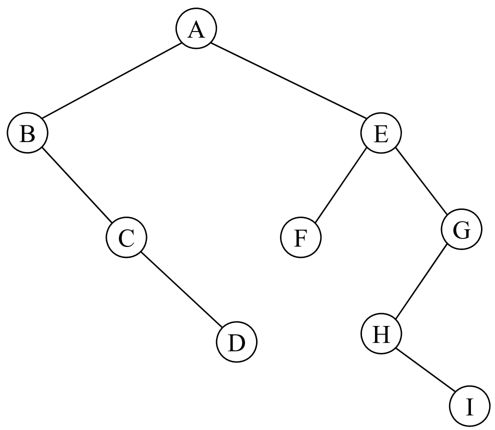
تبدیل جنگل به درخت دودویی

۲. قرار دادن هر یک از درخت های دودویی به عنوان زیردرخت راست ریشه درخت قبلی



تبدیل جنگل به درخت دودویی

۲. قرار دادن هر یک از درخت های دودویی به عنوان زیردرخت راست ریشه درخت قبلی



پیمایش جنگل F به روش پیش ترتیب

۱. اگر F تهی است، برمی گردیم.
۲. ملاقات ریشه اولین درخت در جنگل F
۳. پیمایش زیردرخت های اولین درخت در جنگل F به صورت پیش ترتیب
۴. پیمایش بقیه درخت های جنگل F به صورت پیش ترتیب

• نکته: پیمایش پیش ترتیب جنگل و پیمایش پیش ترتیب درخت دودویی متناظر با آن، یکسان **است**.

پیمایش جنگل F به روش میان ترتیب

۱. اگر F تهی است، برمی گردیم.
۲. پیمایش زیردرخت های اولین درخت در جنگل F به صورت میان ترتیب
۳. ملاقات ریشه اولین درخت در جنگل F
۴. پیمایش بقیه درخت های جنگل F به صورت میان ترتیب

- نکته: پیمایش میان ترتیب جنگل و پیمایش پیش ترتیب درخت دودویی متناظر با آن، یکسان **است**.

پیمایش جنگل F به روش پس ترتیب

۱. اگر F تهی است، برمی گردیم.
۲. پیمایش زیردرخت های اولین درخت در جنگل F به صورت پس ترتیب
۳. پیمایش بقیه درخت های جنگل F به صورت پس ترتیب
۴. ملاقات ریشه اولین درخت در جنگل F

- نکته: پیمایش پس ترتیب جنگل و پیمایش پیش ترتیب درخت دودویی متناظر با آن، یکسان **نیست**.

پیمایش جنگل F به ترتیب سطح

۱. ملاقات ریشه درختان از چپ به راست
۲. ملاقات گره ها ی سطح اول از چپ به راست
۳. ملاقات گره ها ی سطح دوم از چپ به راست
۴. ...

- نکته: پیمایش به ترتیب سطح جنگل و پیمایش به ترتیب سطح درخت دودویی متناظر با آن، یکسان **نیست**.

یک کاربرد درخت: نمایش زیرمجموعه های مجزا (disjoint subsets)

$$S = \{1, 2, \ldots, n\}$$

- یک مثال از زیرمجموعه های مجزا (برای $n = 10$):

$S_1 = \{1, 7, 8, 9\}$

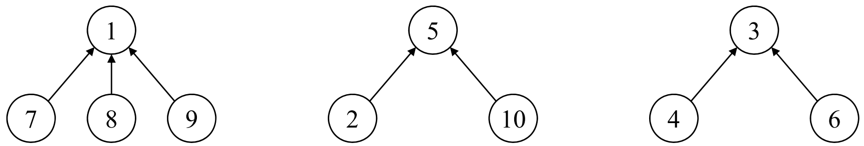
$S_2 = \{2, 5, 10\}$

$S_3 = \{3, 4, 6\}$

- در زیرمجموعه های مجزا داریم:

- $S_i \subseteq S$
- $S_i \cap S_j = \varnothing \quad (i \neq j)$

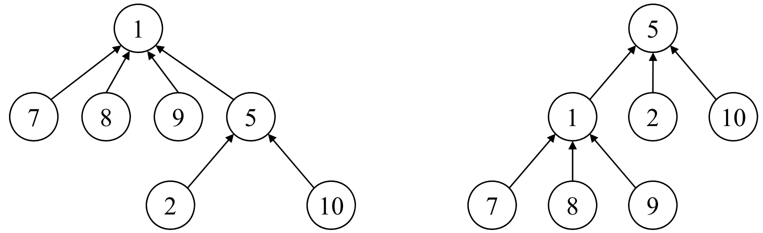
نمایش زیرمجموعه های مجزا



S_1 S_2 S_3

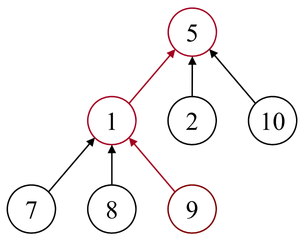
عملیات زیرمجموعه های مجزا:
UNION (i, j)

- در این عمل تنها کافیست که ریشه یک درخت را فرزند ریشه درخت دیگر قرار دهیم.
- مثال: نمایش های ممکن برای $S_1 \cup S_2$:



عملیات زیرمجموعه های مجزا:
FIND (i)

- از گره i و از طریق اشاره گرهایی که به پدر گره ها وجود دارد، به سمت ریشه حرکت می کنیم تا به ریشه درخت برسیم.
- ریشه درخت را به عنوان نام مجموعه حاوی عنصر i برمی گردانیم.
- مثال: FIND (9)

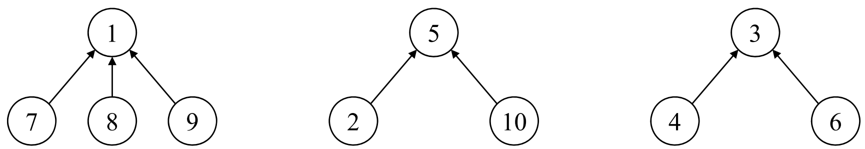


پیاده سازی ساختمان داده زیرمجموعه های مجزا

	1	2	...	MAX
parent				

مثال:

	1	2	3	4	5	6	7	8	9	10
parent	0	5	0	3	0	3	1	1	1	5



پیاده سازی عملیات زیرمجموعه های مجزا

```
procedure UNION (i, j: integer);
begin
    parent [i] := j
end;

function FIND (i: integer): integer;
begin
    while parent [i] > 0 do
        i := parent [i];
    return (i)
end;
```


مثال

- فرض کنید که در ابتدا

$$\forall\ 1 \leq i \leq n \qquad S_i = \{i\}$$

- حال دنباله عملیات زیر را انجام می دهیم:

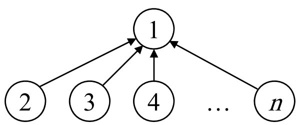
UNION (1, 2) UNION (2, 3) UNION (3, 4) … UNION (n - 1, n)
FIND (1) FIND (2) … FIND (n)

- در نتیجه درخت زیر حاصل می شود:
  - هزینه تمام عملیات اجتماع: O(n)
  - هزینه تمام عملیات یافتن: O(n^2)

بهبود عملیات یافتن و اجتماع

- تعریف (قانون وزنی برای اجتماع i و j): اگر تعداد گره ها در درختی با ریشه i، کمتر از تعداد گره ها در درختی با ریشه j باشد، j والد i می شود و در غیر این صورت i والد j خواهد شد.

- اجرای دنباله عملیات مثال قبل:

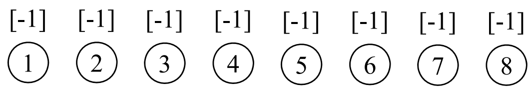


پیاده سازی عمل اجتماع

```
procedure WEIGHTED-UNION (i, j: integer);
var
    temp: integer;

begin
    temp := parent [i] + parent [j];
    if parent [i] > parent [j] then begin
        parent [i] := j;
        parent [j] := temp
    end else begin
        parent [j] := i;
        parent [i] := temp
    end
end;
```

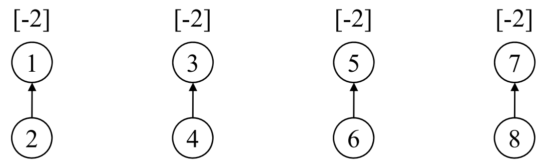
مثال: عملکرد الگوریتم WEIGHTED-UNION

- $1 \le i \le n = 8, \quad parent[i] = -\ count[i] = -\ 1$
- UNION (1, 2), UNION (3, 4), UNION (5, 6), UNION (7, 8), UNION (1, 3), UNION (5, 7), UNION (1, 5)



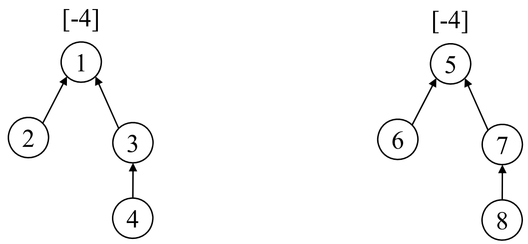
مثال: عملکرد الگوریتم WEIGHTED-UNION

- UNION (1, 2), UNION (3, 4), UNION (5, 6), UNION (7, 8)



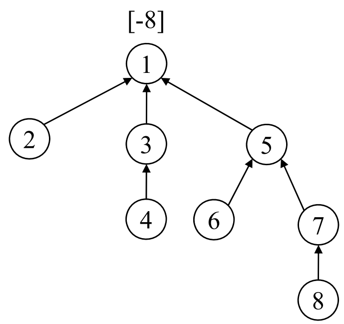
مثال: عملکرد الگوریتم WEIGHTED-UNION

- UNION (1, 3), UNION (5, 7)



مثال: عملکرد الگوریتم WEIGHTED-UNION

- UNION (1, 5)



قانون تخریب

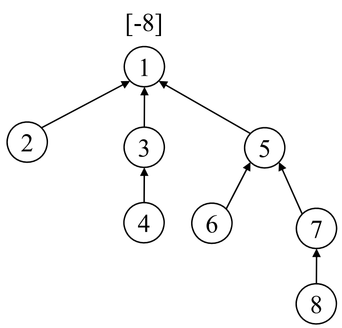
- تعریف قانون تخریب: اگر j گره ای روی مسیر i تا ریشه اش باشد و $parent[i] \neq root(i)$، در این صورت $parent[j]$ را برابر $root(i)$ قرار خواهیم داد.

پیاده سازی عمل COLLAPSING-FIND

```
function COLLAPSING-FIND (i: integer): integer;
var
   r, s: integer;

begin
   r := i;
   while parent [r] > 0 do
      r := parent [r];
   while i <> r do begin
      s := parent [i];
      parent [i] := r;
      i := s
   end;
   return (r)
end;
```


مثال

- FIND (8), FIND (8), FIND (8), FIND (8), FIND (8), FIND (8), FIND (8), FIND (8)



- تعداد حرکت ها اگر از FIND استفاده شود:
  ۳ × ۸ = ۲۴
- اگر از COLLAPSING-FIND استفاده شود:
  ۳ + ۳ + ۷ = ۱۳

تحلیل عملیات WEIGHTED-UNION و COLLAPSING-FIND

- فرض کنید که با جنگلی از درخت ها، که هر کدام یک گره دارد شروع کنیم. $T(f, u)$ حداکثر زمان لازم برای پردازش ترکیبی از f عمل یافتن و u عمل اجتماع است. اگر $u \geq n/2$ باشد، آنگاه به ازای مقادیر مثبت k_1 و k_2:

$$k_1 (n + f\alpha(f + n, n)) \leq T(f, u) \leq k_2 (n + f\alpha(f + n, n))$$

تعریف توابع آکرمان و $\alpha\ (p, q)$

$A\ (1, j) = 2^j, \qquad j \geq 1$

$A\ (i, 1) = A\ (i - 1, 2) \qquad i \geq 2$

$A\ (i, j) = A\ (i - 1, A\ (i, j - 1)) \qquad i, j \geq 2$

$\alpha(p, q) = \min\ \{z \geq 1 \mid A\ (z, \text{floor}\ (p\ /\ q)) > \log_2 q\}, p \geq q \geq 1$

۳ مسأله شمارشی

- شمارش تعداد درخت های دودویی مجزا با اعداد ۱ تا n
- شمارش تعداد جایگشت های پشته ای اعداد ۱ تا n
- شمارش تعداد روش های ممکن برای ضرب $n+1$ ماتریس

$$T(n) = \frac{1}{n+1}\binom{2n}{n} = O(4^n / n^{1.5})$$

درخت عبارت (Expression Tree)

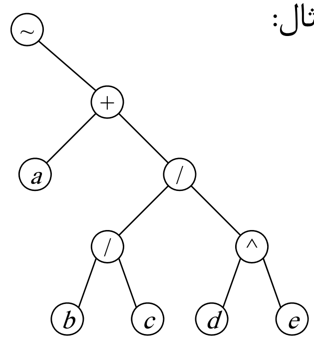
- تعریف فرم عبارت کاملاً پرانتزی
- $E ::= a \mid (\alpha\ E) \mid (E\ \beta\ E)$
- a ::= **variable**
- α ::= **unary operator** (~, sin, cos, …)
- β ::= **binary operator** (^, ×, /, +, -, …)
- مثال:

$$(\sim (a + (b\ /\ c))\ /\ (d \wedge e))$$

درخت عبارت (Expression Tree)

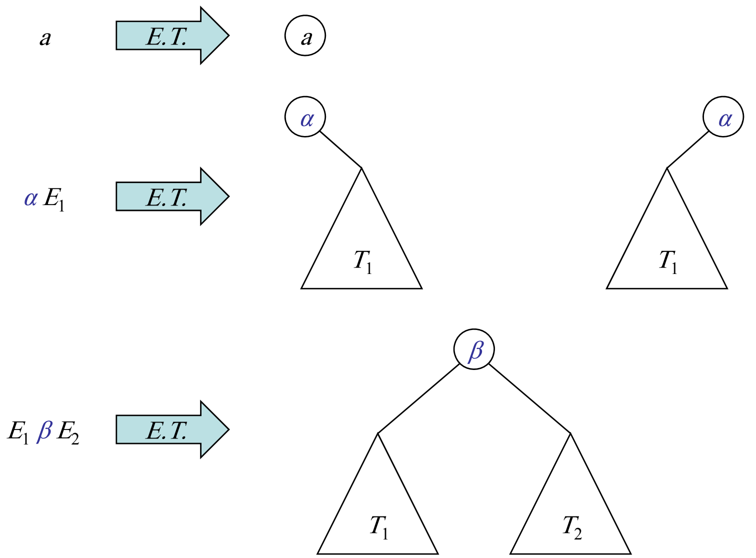
- بهترین مدل برای یک عبارت ریاضی استفاده از درخت می باشد.
- مثال:

$(\sim (a + (b\ /\ c))\ /\ (d \wedge e))$



- نکته: پیمایش های پیش ترتیب، میان ترتیب و پس ترتیب درخت عبارت به ترتیب معادل فرمهای پیشوندی، میانوندی و پسوندی عبارت می باشند.

تبدیل عبارت کاملاً پرانتزی به درخت عبارت



الگوریتم تبدیل عبارت کاملاً پرانتزی به درخت عبارت

function EXP-TREE (*i*, *j*: integer): TREE;
var
T: TREE;
k: integer;
begin
if (*i* = *j*) **then begin**
new (*T*);
T^.*label* := *A* [*i*];
T^.*left* := **nil**;
T^.*right* := **nil**
end

		i, *j*	
A	…	*a*	…

الگوریتم تبدیل عبارت کاملاً پرانتزی به درخت عبارت

else if (*A* [*i* + 1] is a unary operator) **then**
begin
new (*T*);
T^.*label* := *A* [*i* + 1];
T^.*left* := **nil**;
T^.*right* := EXP-TREE (*i* + 2, *j* – 1)
end

		i	*i* + 1	$\overbrace{i+2 \quad\quad j-1}$	*j*	
A	…	(	α	*E*	)	…

الگوریتم تبدیل عبارت کاملاً پرانتزی به درخت عبارت

else begin
new (*T*);
k := MATCH (*i* + 1);
T^.*label* := *A* [*k* + 1];
T^.*left* := EXP-TREE (*i* + 1, *k*);
T^.*right* := EXP-TREE (*k* + 2, *j* – 1)
end;
return (*T*)
end;

		i	$\overbrace{i+1 \quad\quad k}$	*k* + 1	$\overbrace{k+2 \quad\quad j-1}$	*j*	
A	…	(	*E*	β	*E*	)	…

پیاده سازی تابع MATCH (*i*)

```
function MATCH (i: integer): integer;
var
    c, j: integer;
begin
    c := 0;
    j := i;
    repeat
        if A [j] = '(' then c := c + 1;
        if A [j] = ')' then c := c - 1;
        j := j + 1;
    until c = 0;
    return (j – 1)
end;
```


تمرین

- الگوریتمی برای ایجاد درخت عبارت از روی فرم پسوندی بنویسید.

تبدیل درخت عبارت به فرم کاملاً پرانتزی

```
procedure PRINT-INFIX (T: TREE);
begin
    if (T^.label is a unary operator) then begin
        write ('(');
        write (T^.label);
        PRINT-INFIX (T^.right);
        write (')')
    end
```


تبدیل درخت عبارت به فرم کاملاً پرانتزی

```
    if (T^.label is a binary operator) then begin
        write ('(');
        PRINT-INFIX (T^.left);
        write (T^.label);
        PRINT-INFIX (T^.right);
        write (')')
    end else
        write (T^.label)
end;
```

الگوریتم فشرده سازی هافمن

• مسأله:

– ورودی: تعدادی کاراکتر و احتمال وقوع هر یک از کاراکترها

– خروجی: کدهای کاراکترها به طوری که ضریب فشرده سازی حداقل شود.

$$\text{ضریب فشرده سازی} = \frac{\text{اندازه فایل فشرده شده}}{\text{اندازه فایل اصلی}}$$

• مثال:

کاراکتر	a	b	c	d	e	f	g
فراوانی	5	3	6	1	8	2	4

طرح اول برای کد گذاری: کد باینری با طول ثابت

• در این روش اگر n کاراکتر مختلف داشته باشیم، به هر کدام یک کد باینری با طول ثابت زیر نسبت می دهیم:

$$\text{طول کد} = \lfloor \lg n \rfloor + 1$$

کاراکتر	فراوانی	کد باینری با طول ۳
a	5	000
b	3	001
c	6	010
d	1	011
e	8	100
f	2	101
g	4	110

کلمه کد (Code word)

کتاب کد (Code book)

• محاسبه اندازه فایل کد شده (بر حسب بیت):

اندازه فایل کد شده =

$$3 \times (5+3+6+1+8+4+2) = 87 \text{ bits}$$

طرح دوم برای کد گذاری: کد باینری با طول متغیر

• انگیزه: استفاده از کلمات کد با طول کمتر برای کاراکترهایی که فراوانی بیشتری دارند.

• راه حل: استفاده از روش کد گذاری هافمن:

– مرحله ۱) به ازای هر کاراکتر یک درخت دودویی تک گره ای ایجاد و وزن ریشه آن درخت را برابر با احتمال کاراکتر مربوطه قرار می دهیم.

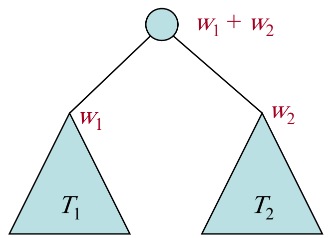
1	2	3	4	5	6	8
d	f	b	g	a	c	e

روش کد گذاری هافمن

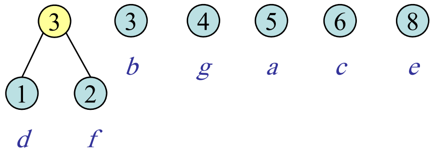
• مرحله ۲) عملیات زیر را $n-1$ بار تکرار کن:

– هر بار دو درخت T_1 و T_2 را که ریشه آنها دارای کمترین وزنهای w_1 و w_2 هستند انتخاب می کنیم.

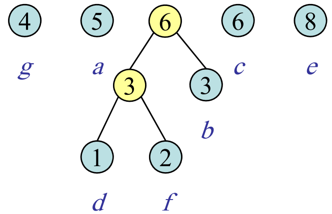
– یک درخت دودویی جدید ایجاد کن که T_1 و T_2 زیردرختان چپ و راست آن باشند و وزن ریشه آن $w_1 + w_2$ باشد.



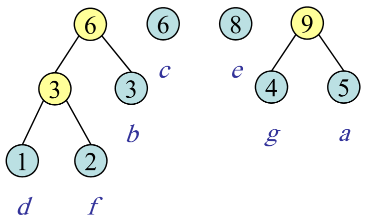
روش کد گذاری هافمن



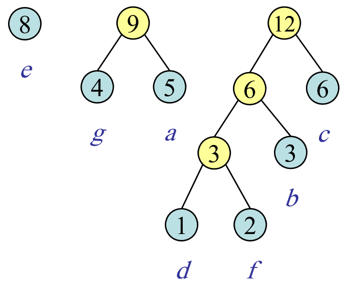
روش کد گذاری هافمن



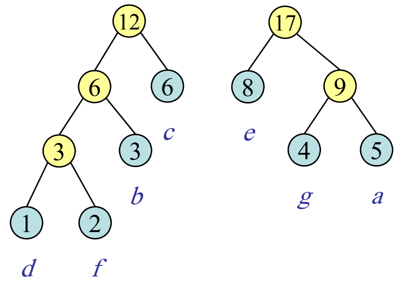
روش کد گذاری هافمن



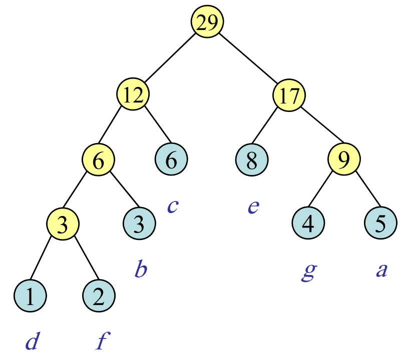
روش کد گذاری هافمن



روش کد گذاری هافمن

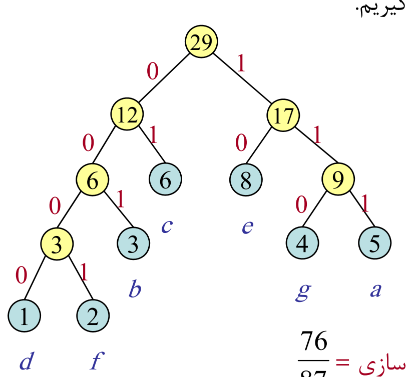


روش کد گذاری هافمن



روش کد گذاری هافمن

- مرحله ۳) تولید کد کاراکترها:
  - در درخت نهایی، به شاخه های چپ برچسب 0 و به شاخه های راست برچسب 1 (و یا برعکس) می دهیم.
  - به ازای هر کاراکتر، مسیر از ریشه تا برگ مربوط به آن کاراکتر را پیموده و برچسب های موجود در مسیر را به عنوان کد آن کاراکتر در نظر می گیریم.



ضریب فشرده سازی = $\frac{76}{87}$

کاراکتر	کد هافمن
a	111
b	001
c	01
d	0000
e	10
f	0001
g	110

یک قضیه مرتبط

- قضیه: یک درخت دودویی کامل با n گره برگ، دارای $n-1$ گره داخلی می باشد.

- اثبات: از طریق استقرا روی n

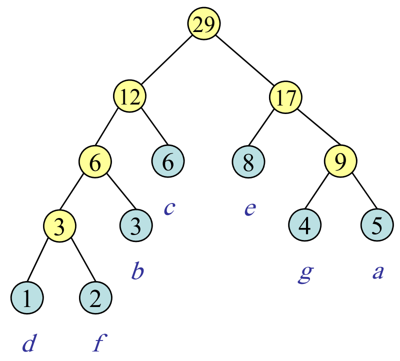
۲ نکته در مورد روش کد گذاری هافمن

- روش کد گذاری هافمن بهینه می باشد، یعنی با این روش ضریب فشرده سازی حداقل می شود.

- کدهای به دست آمده خاصیت پیشوندی دارند، یعنی در کتاب کد هیچ کلمه کدی پیشوند کلمه کد دیگری نمی باشد.
  – در نتیجه دیکود کردن با یک بار پویش فایل کد شده به سادگی قابل انجام می باشد.
  – مثال: رشته 0001111000010 را دیکود نمایید.

دیکود کردن (رمزگشایی)

0001111000010



پیاده سازی الگوریتم هافمن

FOREST

	root	*weight*
1		
2		
	. . .	. . .
n		

ALPHABET

	symbol	*prob*	*leaf*
1			
2			
	. . .	. . .	. . .
n			

TREE

	left	*right*	*parent*
1			
2			
	. . .	. . .	. . .
2*n*-1			

تمرین

- الگوریتم فشرده سازی هافمن را پیاده سازی نمایید.