# درخت

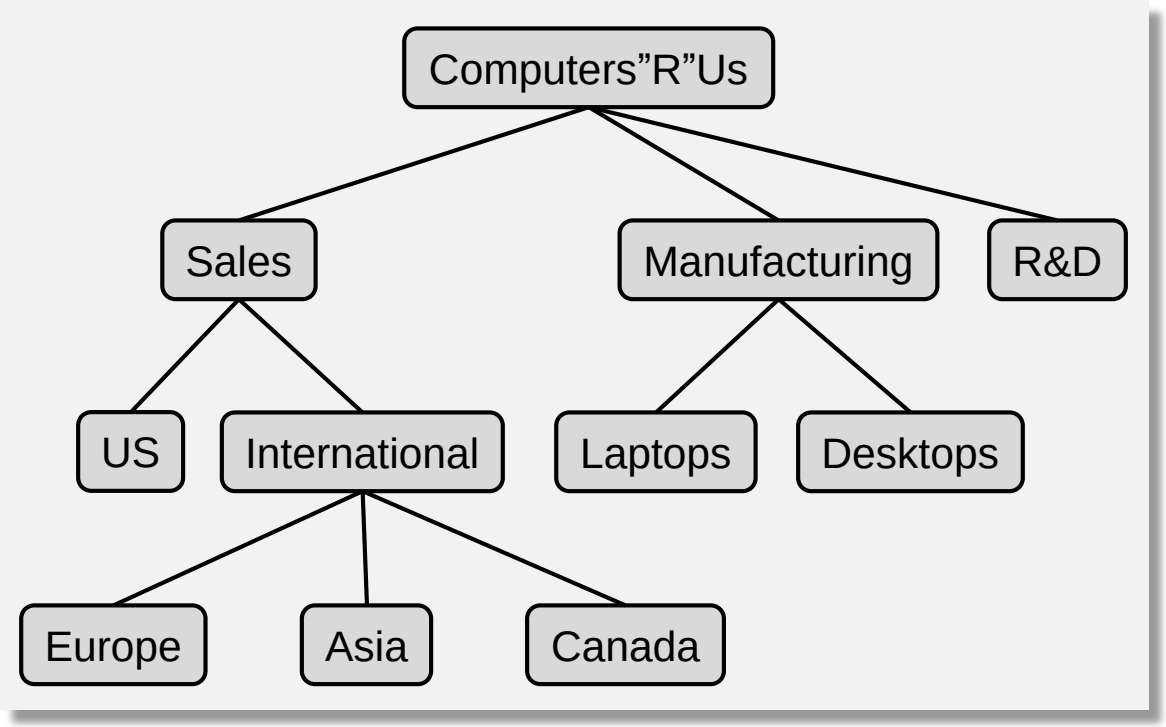
سید ناصر رضوی www.snrazavi.ir
۱۳۹۵

# فهرست مطالب

# درخت



- درخت. یک مدل انتزاعی برای نمایش ساختارهای سلسله‌مراتبی.



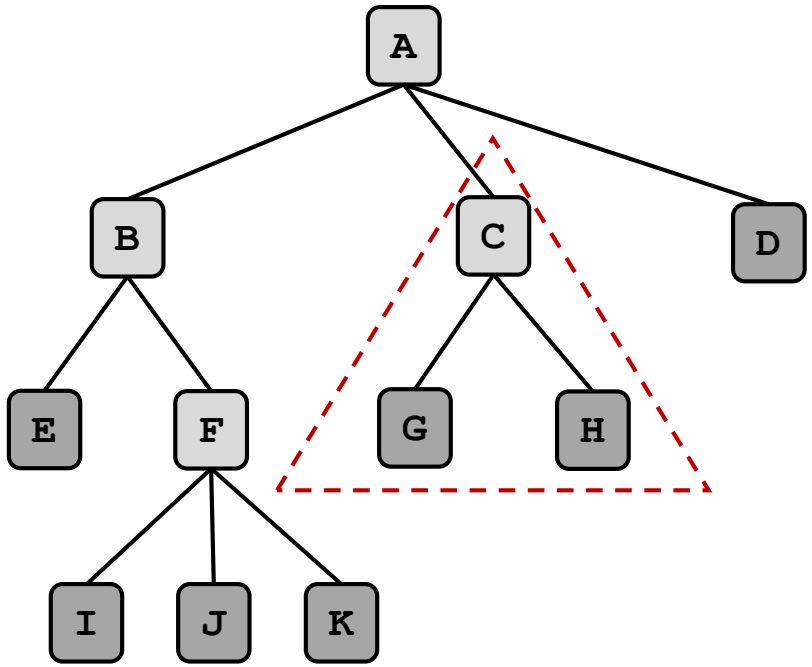
# درخت



- در علوم محاسباتی، درخت یک مدل انتزاعی برای نمایش ساختارهای سلسله‌مراتبی است.
- یک درخت شامل گره‌هایی است که بین آنها روابط پدر- فرزندی برقرار است (ساختار غیر خطی)
  - به جز بالاترین عنصر (ریشه)، هر عنصر دقیقاً یک پدر دارد.
  - همچنین، هر عنصر در درخت می‌تواند صفر یا چند فرزند داشته باشد.
- یک مثال معروف. شجره‌نامه‌های خانوادگی
- برخی از کاربردها.
  - چارت‌های سازمانی
  - سیستم‌های فایل
  - محیط‌های برنامه‌نویسی

# اصطلاحات و تعاریف



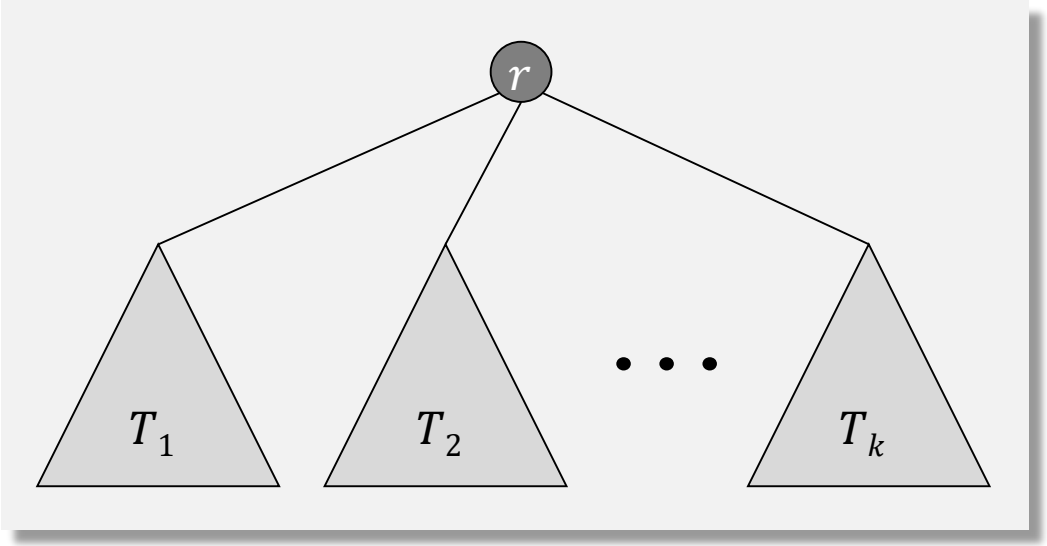



- ریشه. گره بدون والد (**A**)
- گره داخلی. گره‌ای که حداقل یک فرزند دارد. (**A, B, C, F**)
- برگ. [گره خارجی] گره بدون فرزند. (**D, E, G, H, I, J, K**)
- گره‌های هم نیا. گره‌هایی که والد یکسان دارند. (**I, J, K**)
- عمق گره. طول مسیر از ریشه تا آن گره.
- ارتفاع درخت. حداکثر عمق گره‌ها در درخت.
- اجداد یک گره. پدر، پدر بزرگ، پدرِ پدر بزرگ و ...
- نسلهای یک گره. فرزند، فرزندِ فرزند و ...

# تعریف رسمی درخت (تعریف بازگشتی)



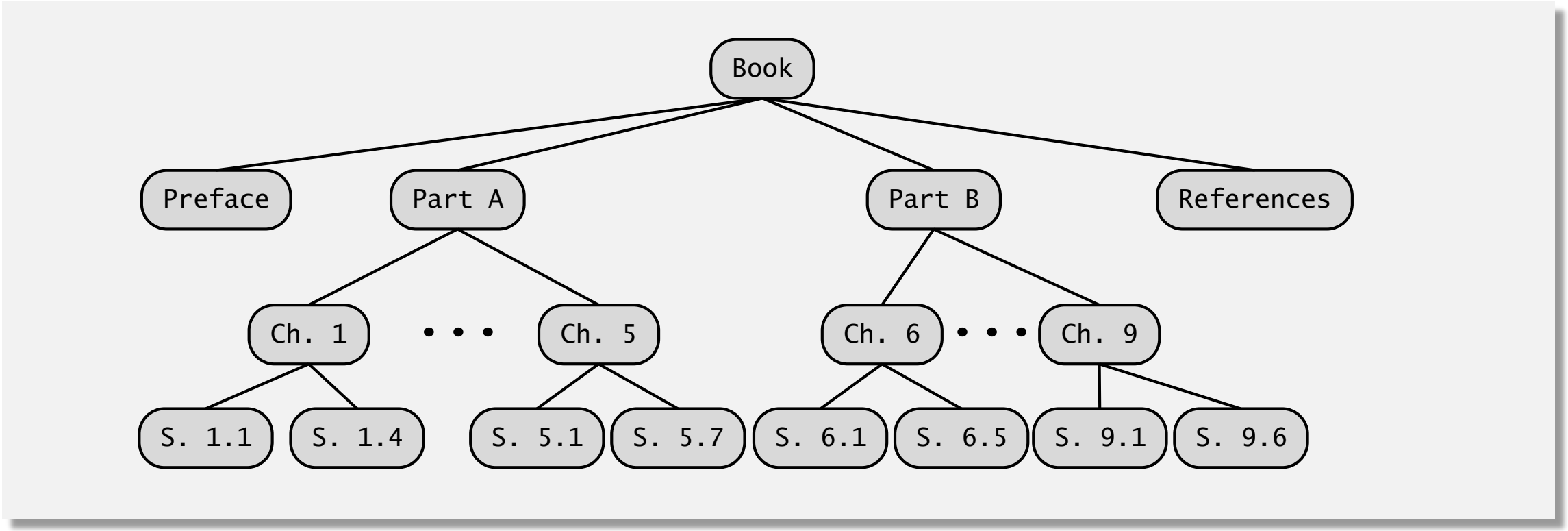
- درخت. یک مجموعه‌ی محدود از گره‌ها با ویژگی‌های زیر است:
  - اگر تهی نباشد، دارای یک گره خاص به نام ریشه است.
  - سایر گره‌ها به $k \geq 0$ مجموعه‌ی مجزا مانند $T_1$، $T_2$، ... و $T_k$ تقسیم می‌شوند به گونه‌ای که هر یک از این مجموعه‌ها خود یک درخت هستند. [$T_1$، $T_2$، ... و $T_k$ را زیردرخت‌های ریشه می‌نامیم]



# کاربرد درخت‌ها



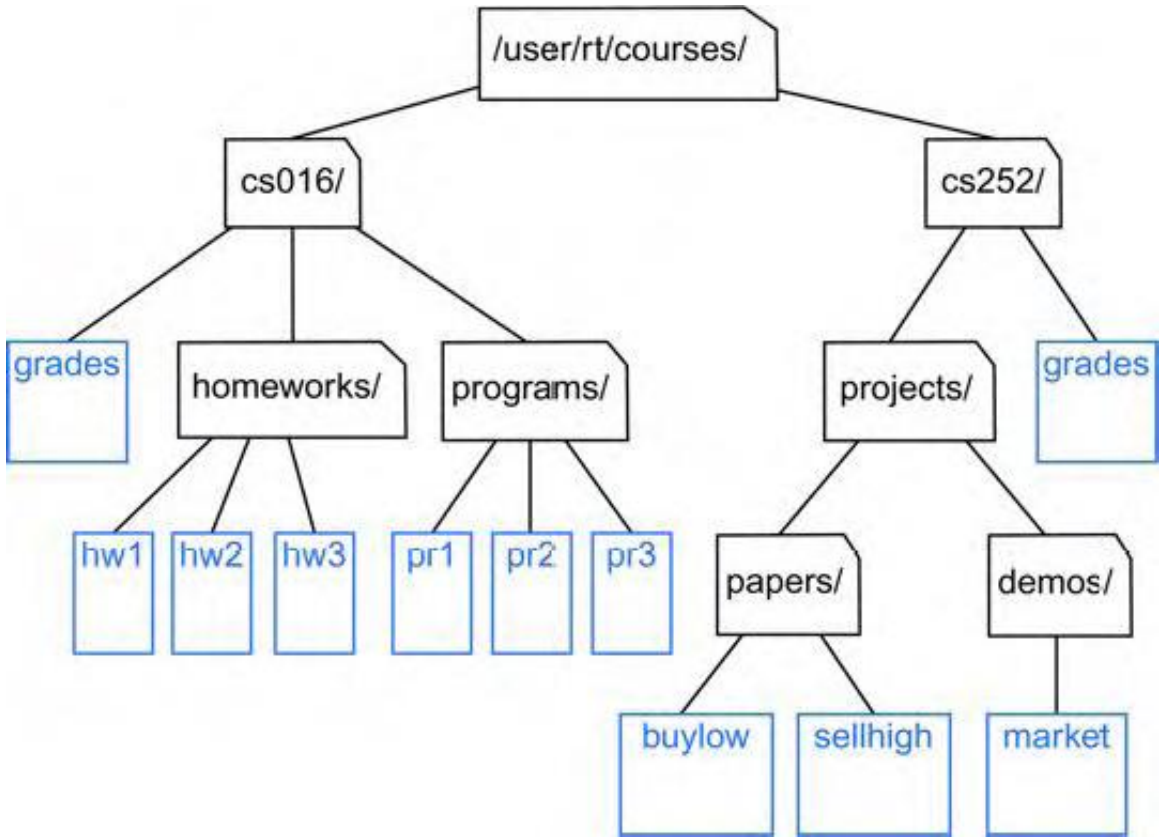
- نمایش ساختار یک کتاب.



# کاربرد درخت‌ها



- نمایش ساختار یک سیستم فایل.



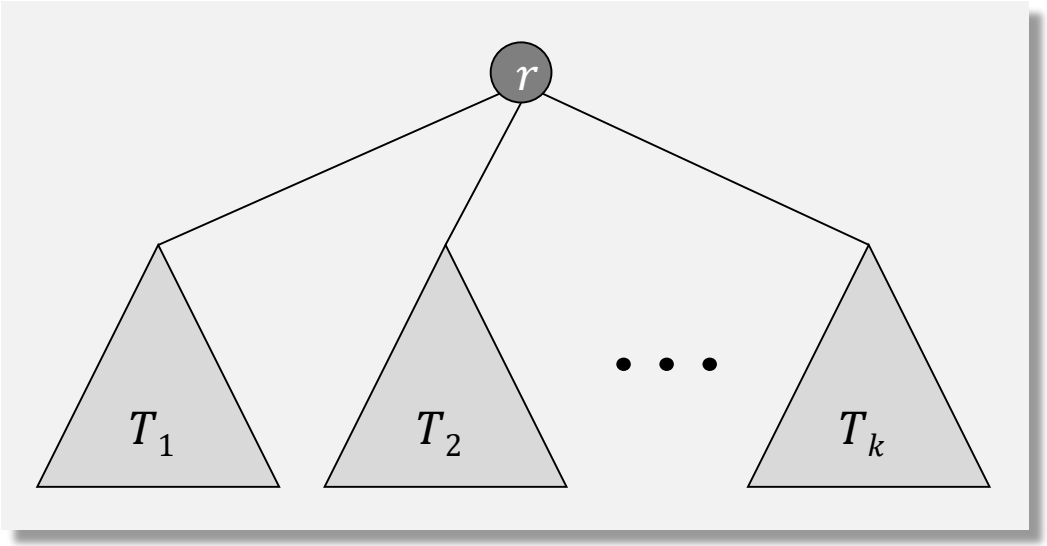
# درخت مرتب



- درخت مرتب. یک درخت مرتب است اگر به ازای هر گره، یک ترتیب خطی بر روی فرزندان آن گره تعریف شده باشد.
  - در یک درخت مرتب می‌توان به فرزندان یک گره به صورت فرزند اول، فرزند دوم و ... ارجاع نمود.
  - در درخت‌های مرتب، فرزندان یک گره بر اساس ترتیبی که دارند از چپ به راست قرار داده می‌شوند.
- مثال. شجره‌نامه‌های خانوادگی و درخت‌های مربوط به ساختار کتاب‌ها .

# روشهای پیمایش درخت





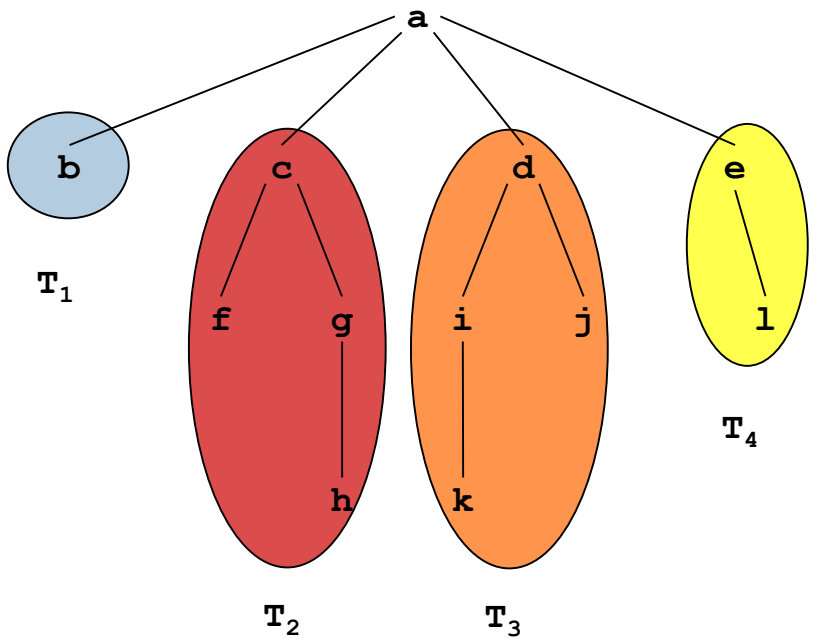

Preorder(T) : r, Pre($T_1$), Pre($T_2$), …, Pre($T_k$)

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 پیش‌ترتیب



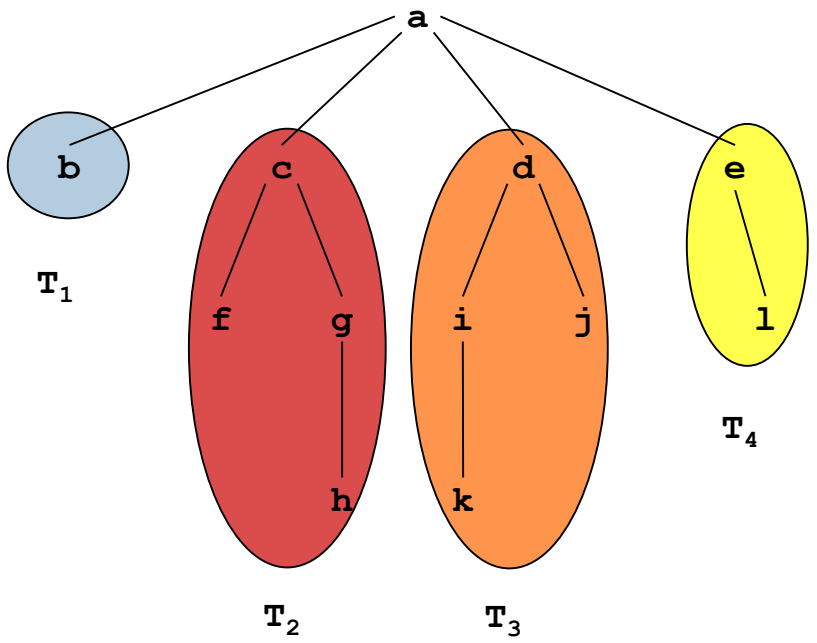



Preorder: a b c f g h d i k j e l

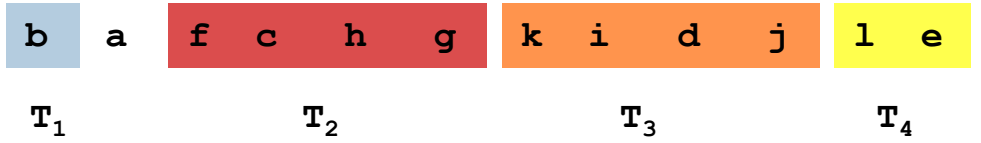
$T_1$ $T_2$ $T_3$ $T_4$

# مثال: پیمایش میان‌ترتیب



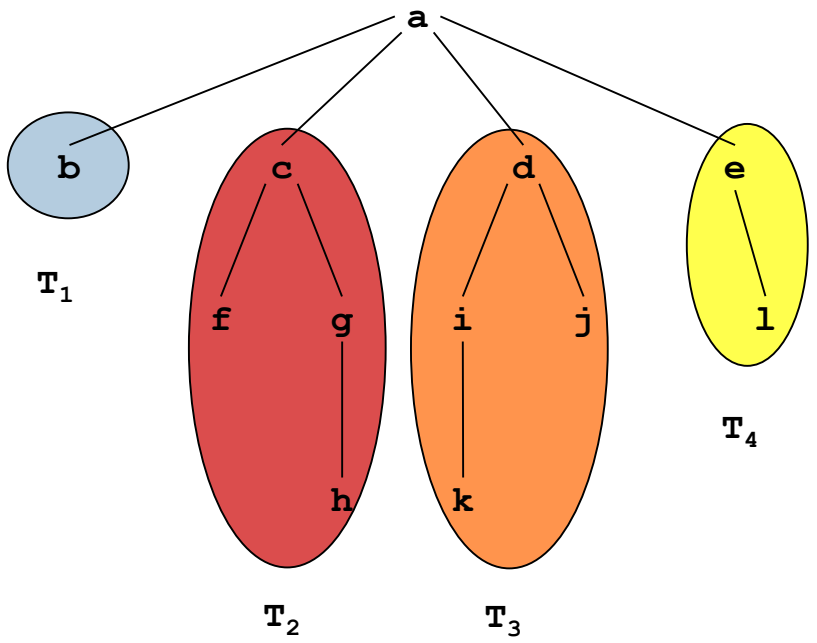



Inorder:



# مثال: پیمایش پس‌ترتیب






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

$T_1$ $T_2$ $T_3$ $T_4$

# پیمایش پیش‌ترتیب درخت عمومی



```
public Iterable<Item> preorder()
{
   Queue<Item> q = new Queue<Item>();
   preorder(root, q);
   return q;
}
```

```
public void preorder(Node x, Queue<Item> q)
{
   q.enqueue( x.item );
   Node y = x.lmc;
   while (y != null)
   {
      preorder(y, q);
      y = y.rsb;
   }
}
```

```
private class Node
{
   private Item item;
   private Node parent;
   private Node lmc;
   private Node rsb;
}
```

# پیمایش میان‌ترتیب درخت عمومی (غیر بازگشتی)



```
public void inorder(Node x, Queue<Item> q)
{
   Node y = x.lmc;
   if (y != null)
   {
      inorder(y, q);
      q.enqueue(x.item);
      y = y.rsb;
      while (y != null)
      {
         inorder(y, q);
         y = y.rsb;
      }
   }
   else
      q.enqueue(x.item);
}
```

# پیمایش پس‌ترتیب درخت عمومی



```
public Iterable<Item> postorder()
{
   Queue<Item> q = new Queue<Item>();
   postorder(root, q);
   return q;
}
```

```
public void postorder(Node x, Queue<Item> q)
{
   Node y = x.lmc;
   while (y != null)
   {
      postorder(y, q);
      y = y.rsb;
   }
   q.enqueue(x.item);
}
```

# پیاده‌سازی درخت عمومی (روش بد)



- پیاده‌سازی گره‌ها.
  - یک فیلد برای ذخیره‌ی اطلاعات (برچسب)
  - **k** فیلد اشاره‌گر برای ذخیره‌ی آدرس فرزندان

| item | Child 1 | Child 2 | Child 3 | … | Child k |
|---|---|---|---|---|---|

- ایراد این روش پیاده‌سازی.
  - تعداد کل فیلدهای اشاره‌گر به فرزندان: `n * k`
  - تعداد فیلدهای اشاره‌گر غیر پوچ: `n - 1`
  - تعداد اشاره‌گر پوچ برابر است با:

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

# پیاده‌سازی چپ‌ترین فرزند-برادر راست (LMC-RSB)



- در این روش، هر گره دارای سه فیلد آدرس است:
  - یکی برای ذخیره‌ی آدرس پدر
  - یکی برای ذخیره‌ی آدرس چپ ترین فرزند
  - یکی برای ذخیره‌ی آدرس نزدیک ترین برادر راست

# مثال: محاسبه‌ی تعداد گره‌های یک درخت LMC-RSB



```
public int size() {
   return size(root);
}

private int size(Node x)
{
   if (x == null) return 0;
   int count = 1;
   Node y = x.lmc;
   while (y != null)
   {
      count += size(y);
      y = y.rsb;
   }
   return count;
}
```

# مثال: محاسبه‌ی ارتفاع یک درخت LMC-RSB



```
public int height() {
   return height(root);
}

private int height(Node x)
{
   if (x == null) return -1;
   int h = -1;
   Node y = x.lmc;
   while (y != null)
   {
      h = Math.max(h, height(y));
      y = y.rsb;
   }
   return 1 + h;
}
```

# تمرین: محاسبه‌ی تعداد برگ‌های یک درخت LMC-RSB



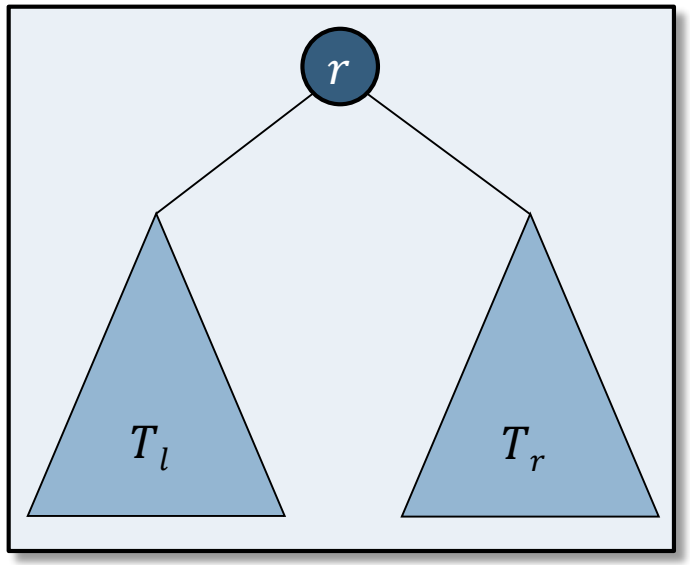
```
public int numLeaves() {
   return numLeaves(root);
}

private int numLeaves(Node x)
{



   .
   .
   .



}
```

# درخت دودویی

# درخت دودویی (تعریف بازگشتی)



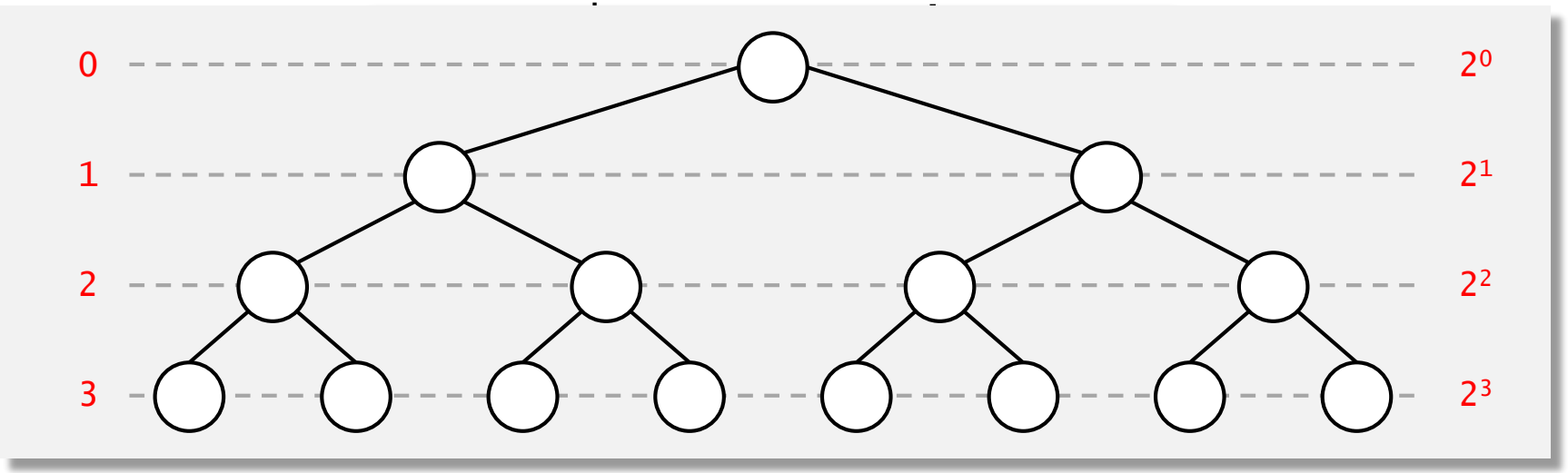
- درخت دودویی. یا تهی است و یا حاوی مجموعه‌ی محدودی از گره‌ها شامل ریشه و دو درخت دودویی است. این درخت‌ها، زیردرخت‌های چپ و راست نامیده می‌شوند.




- تفاوت‌های درخت دودویی با درخت عمومی:
  - درخت دودویی می‌تواند تهی باشد.
  - در درخت دودویی ترتیب فرزندان اهمیت دارد.

# خواص درخت‌های دودویی



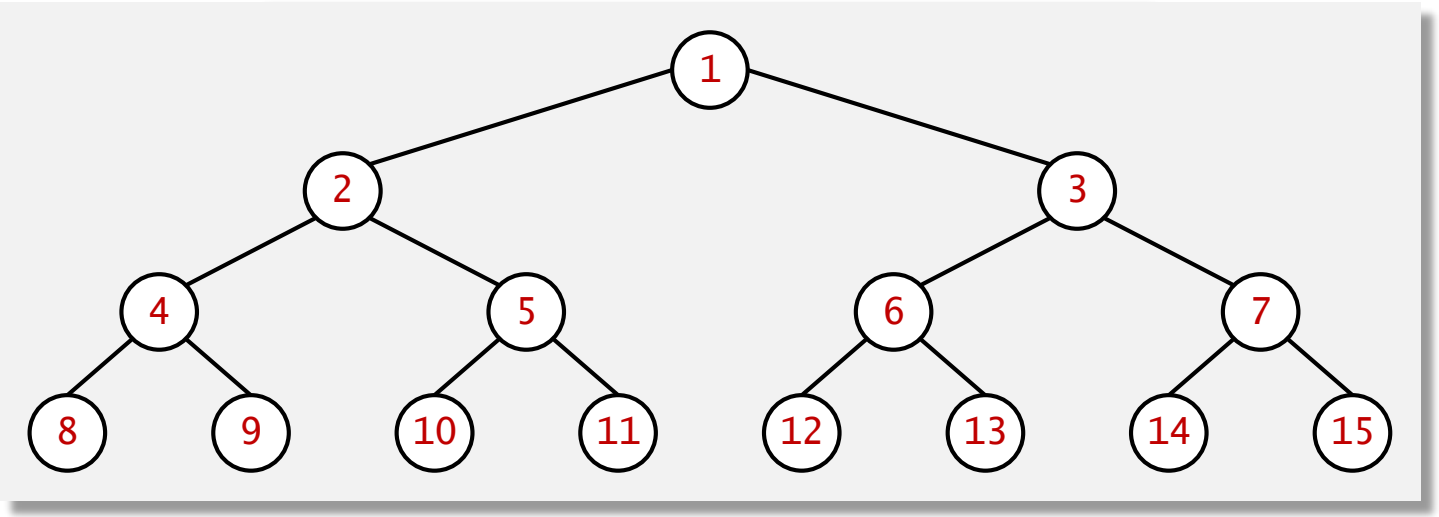



- حداکثر تعداد گره‌ها در سطح i برابر است با : $2^i$
- حداکثر تعداد گره‌ها در یک درخت دودویی با ارتفاع h برابر است با : $2^{h+1} - 1$

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

# نمایش درخت دودویی به وسیله‌ی آرایه

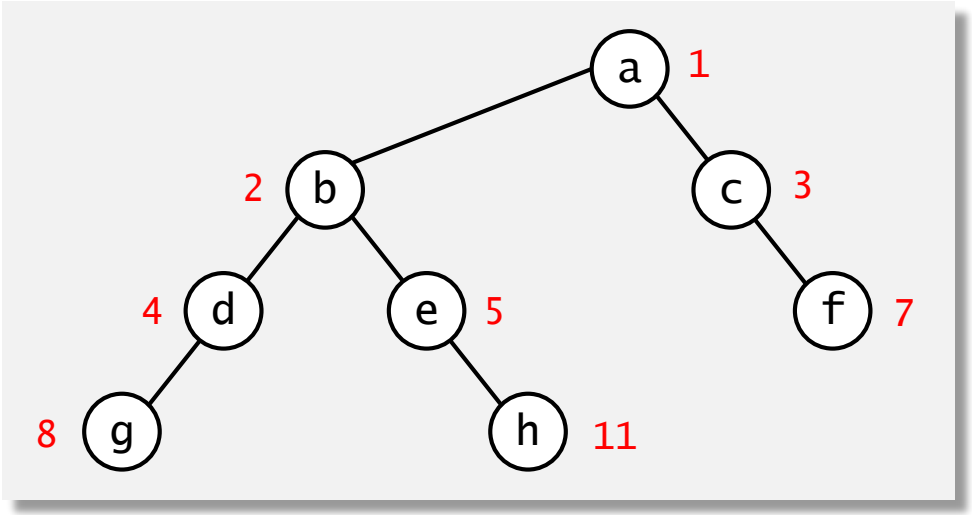





- ریشه در خانه‌ی ۱ قرار دارد.
- اگر یک گره در خانه i ام باشد، آنگاه:
  - پدر آن گره در خانه‌ی $\lfloor i/2 \rfloor$ (i > 1)
  - فرزند چپ آن در خانه‌ی $2i$ (2i ≤ n)
  - فرزند راست آن در خانه‌ی $2i + 1$ (2i + 1 ≤ n)

# نمایش درخت دودویی به وسیله آرایه

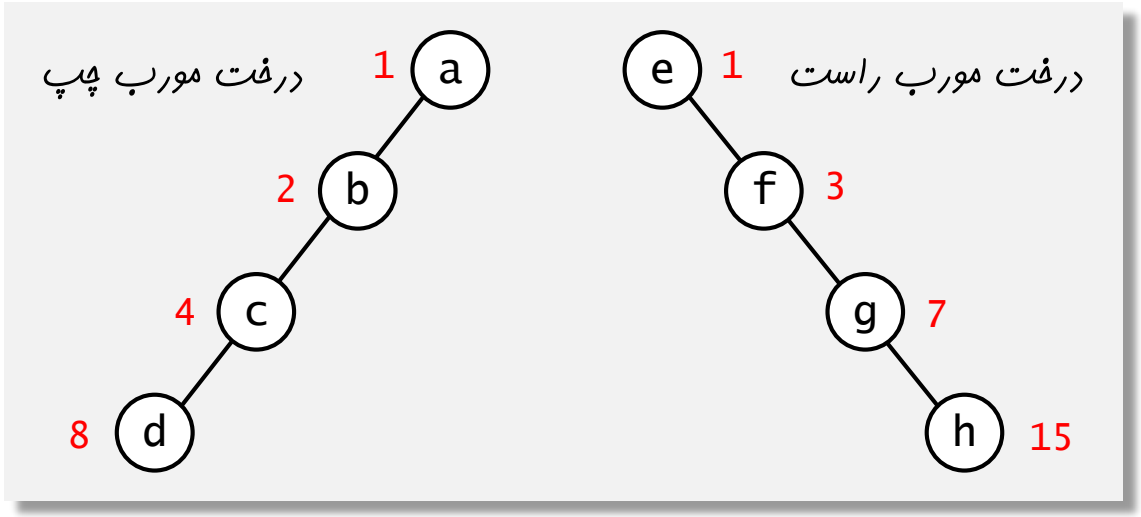





| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| a | b | c | d | e | | f | g | | | h | | | | |

# 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 | | | | | | | |

| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| e | | f | | | | g | | | | | | | | h |

# معایب پیاده‌سازی درخت دودویی به وسیله‌ی آرایه



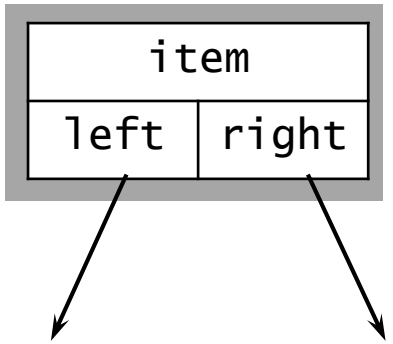
- اتلاف حافظه.
  - این روش فقط برای درخت‌های دودویی پر و تقریباً پر مناسب است.
  - در بدترین حالت، یک درخت مورب به ارتفاع h به $2^{h+1} - 1$ خانه نیاز دارد که از این تعداد فقط $h + 1$ خانه استفاده می‌شود و بقیه‌ی خانه‌ها خالی می‌مانند.
- کند بودن عمل درج و حذف.
  - در این روش، درج و حذف یک گره نیاز به جابجایی گره‌های دیگر دارد که این مسئله باعث کندی عمل درج و حذف می‌گردد.

# پیاده‌سازی درخت دودویی به وسیله‌ی اشاره‌گر



تعریف جاوا. درخت دودویی یک ارجاع به گره ریشه است.

```
private class Node
{
   private Item item;
   private Node left, right;

   public Node(Item item) {
      this(item, null, null);
   }

   public Node(Item item, Node left, Node right)
   {
      this.item  = item;
      this.left  = left;
      this.right = right;
   }
}
```

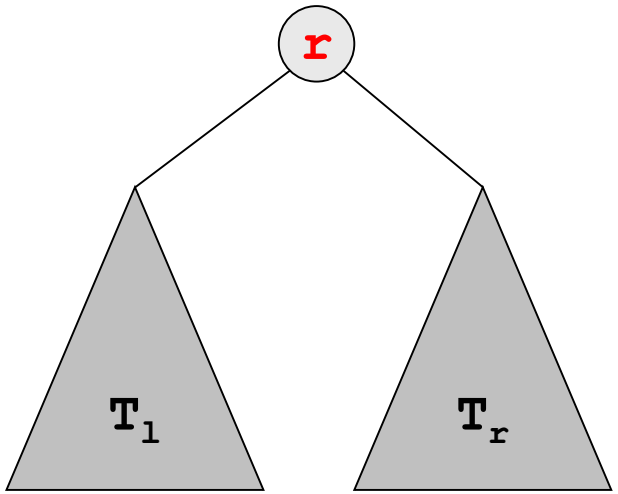


# پیمایش درخت دودویی

پیمایش پیش‌ترتیب
پیمایش میان‌ترتیب
پیمایش پس‌ترتیب
پیمایش به ترتیب سطح
ترسیم درخت دودویی به کمک پیمایش‌های آن

# روشهای پیمایش درخت دودویی





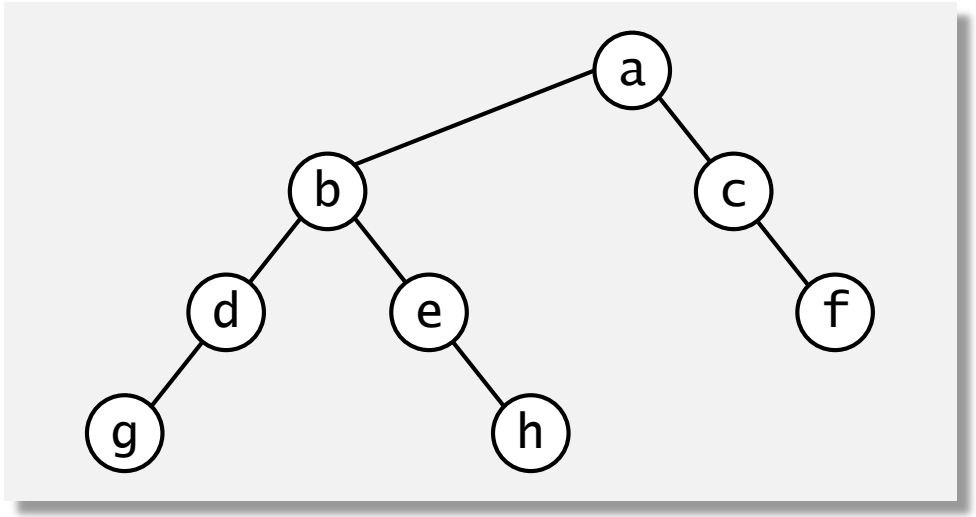

Preorder(T) : r, preorder($T_l$), preorder($T_r$)

Inorder(T) : inorder($T_l$), r, inorder($T_r$)

Postorder(T) : postorder($T_l$), postorder($T_r$), r

# 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 پیش‌ترتیب درخت دودویی



```
public Iterable<Item> preorder()
{
   Queue<Item> q = new Queue<Item>();
   preorder(root, q);
   return q;
}

private void preorder(Node x, Queue<Item> q)
{
   if (x == null) return;
   q.enqueue(x.item);
   preorder(x.left,  q);
   preorder(x.right, q);
}
```

# پیمایش میان‌ترتیب درخت دودویی



```
public Iterable<Item> inorder()
{
   Queue<Item> q = new Queue<Item>();
   inorder(root, q);
   return q;
}

private void inorder(Node x, Queue<Item> q)
{
   if (x == null) return;
   inorder(x.left,  q);
   q.enqueue(x.item);
   inorder(x.right, q);
}
```

# پیمایش پس‌ترتیب درخت دودویی

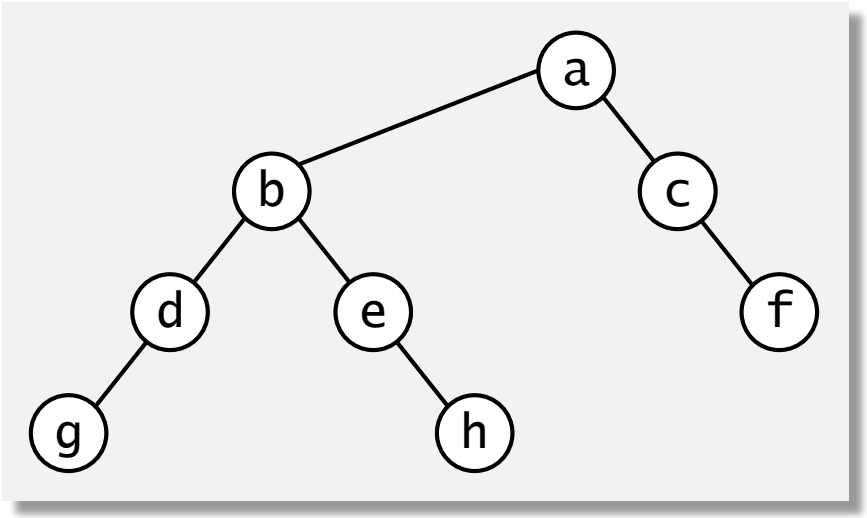


```
public Iterable<Item> postorder()
{
   Queue<Item> q = new Queue<Item>();
   postorder(root, q);
   return q;
}

private void postorder(Node x, Queue<Item> q)
{
   if (x == null) return;
   postorder(x.left,  q);
   postorder(x.right, q);
   q.enqueue(x.item);
}
```

# پیمایش میان‌ترتیب درخت دودویی: غیربازگشتی



```
private NR-Inorder(Node x) {

   Stack<Node> s = new Stack<Node>();
   boolean done = false;

   while (!done) {
      while (x != null) {
         s.push(x);
         x = x.left;
      }

      if (!s.isEmpty()) {
         x = s.pop();
         StdOut.print(x.item);
         x = x.right;
      }

      else done = true;
   }
}
```

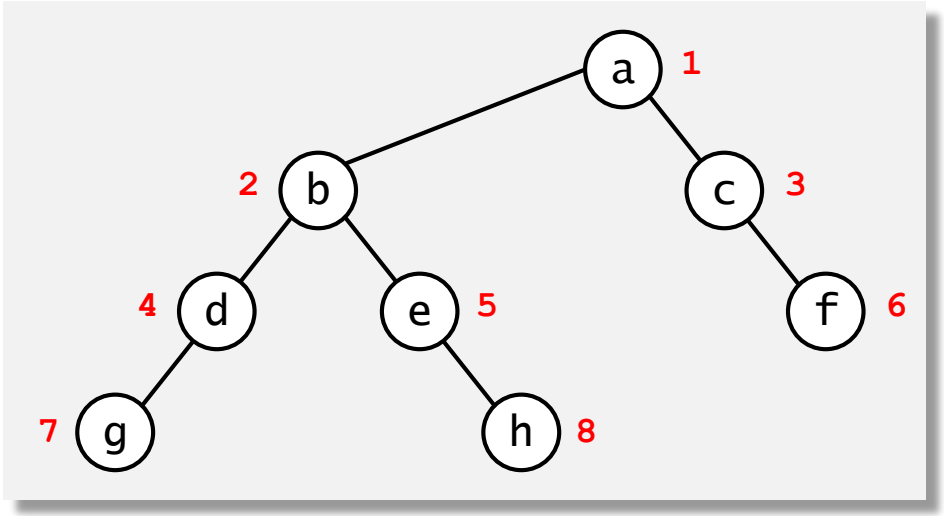


# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```



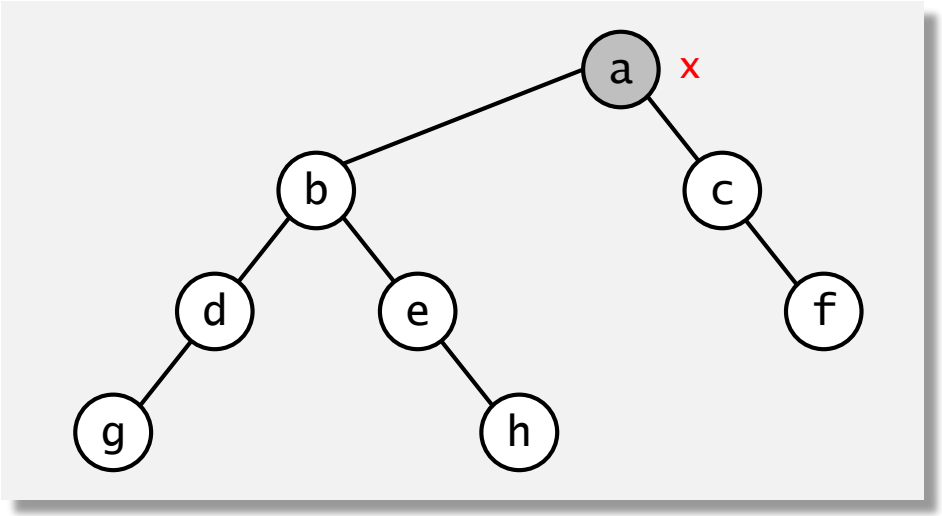

a

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```




a

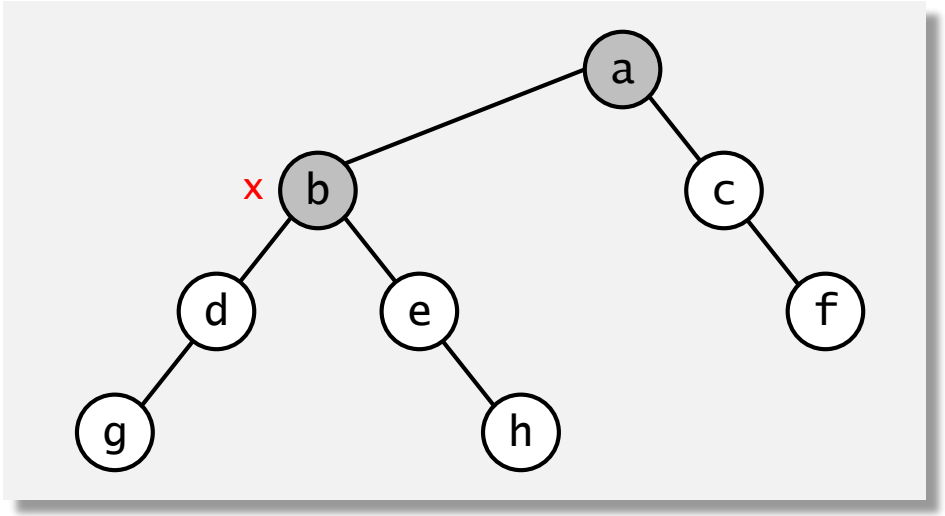
a

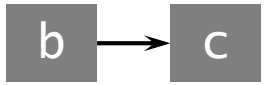
# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```



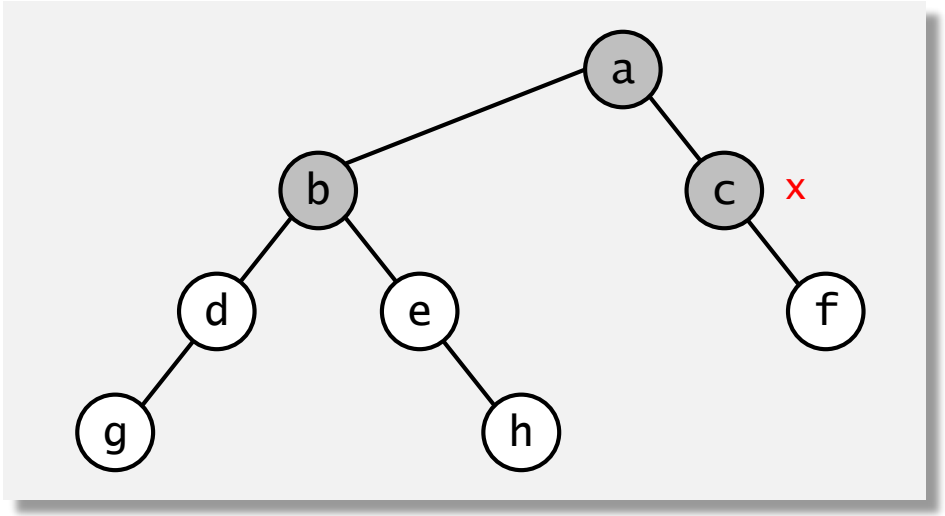

a b

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```





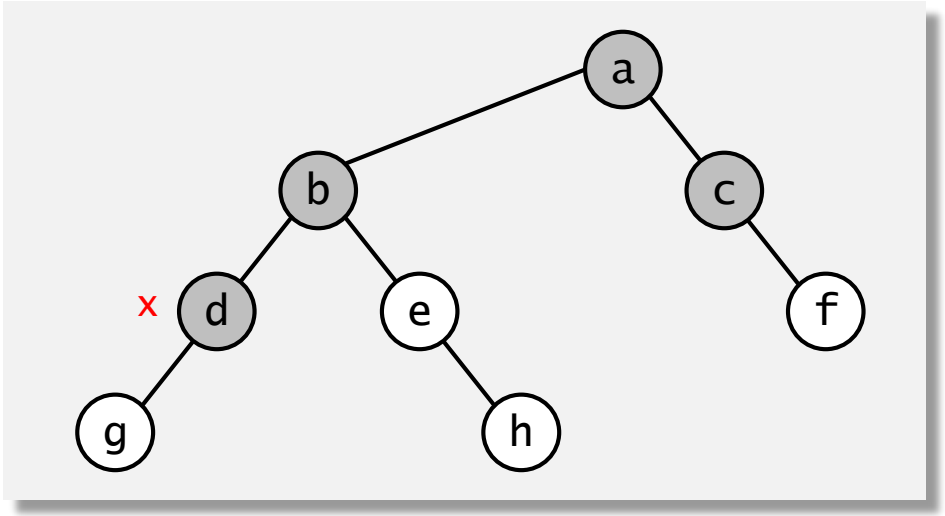
a b c

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```





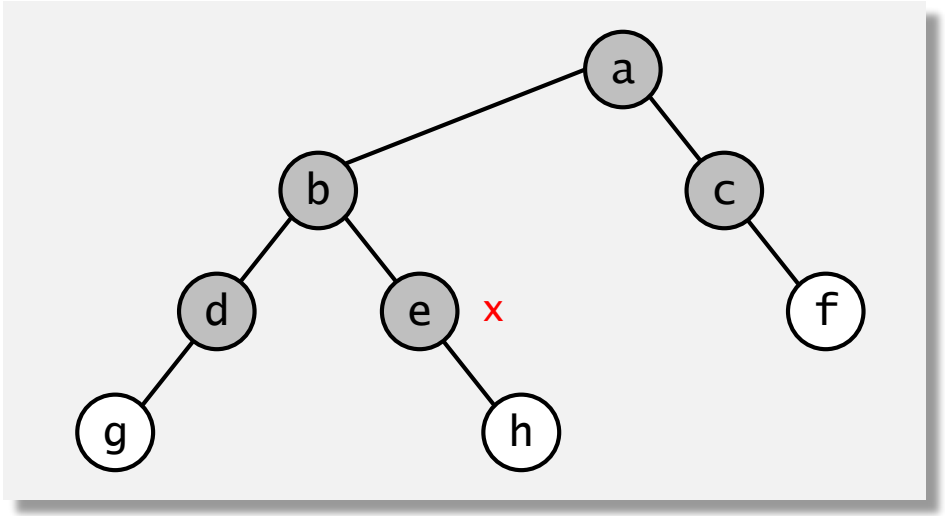
a b c d

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```





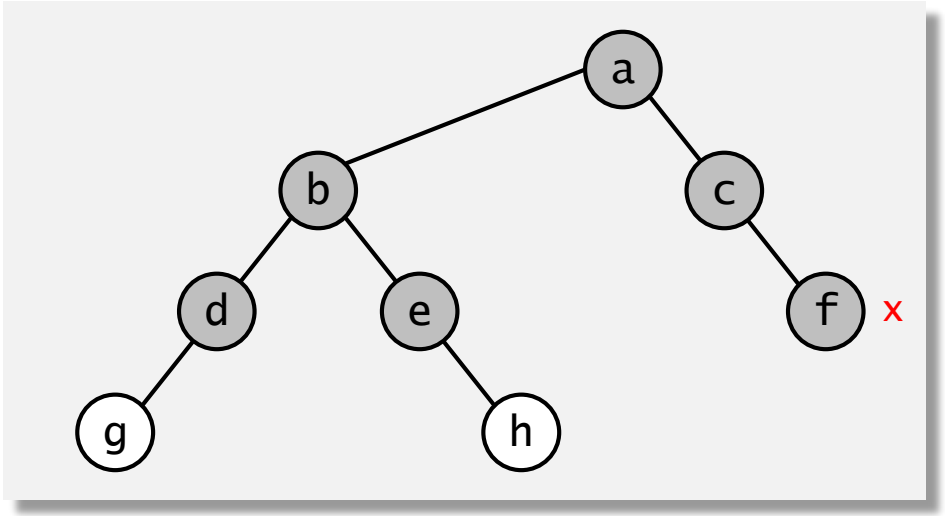
a b c d e

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```





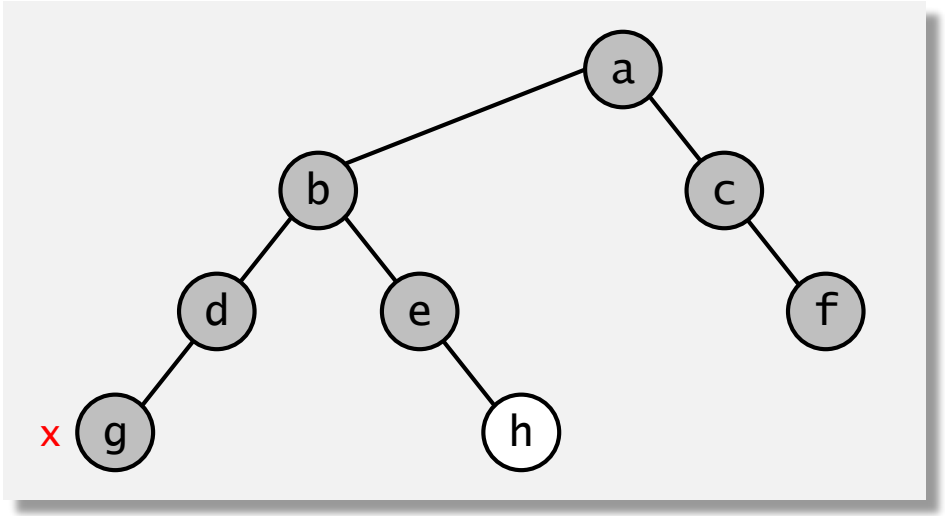
a b c d e f

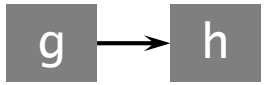
# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```



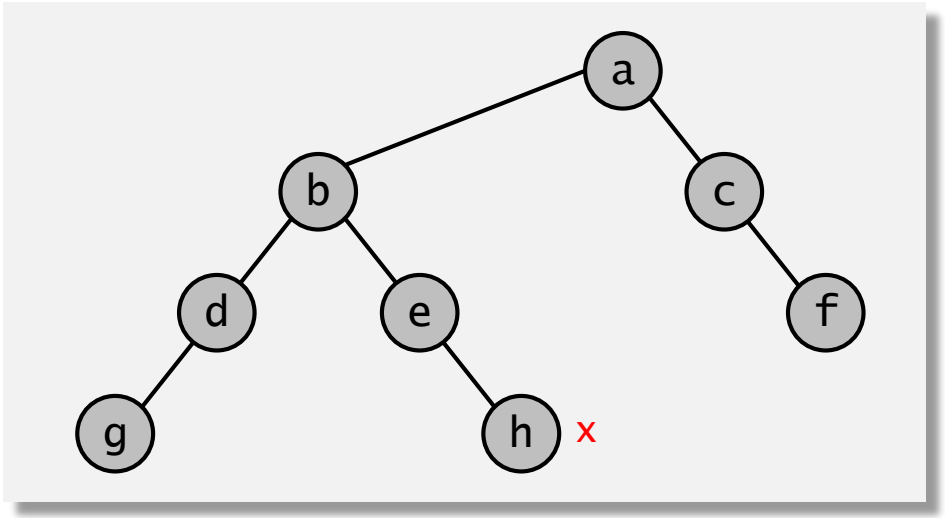

a b c d e f g

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```




h

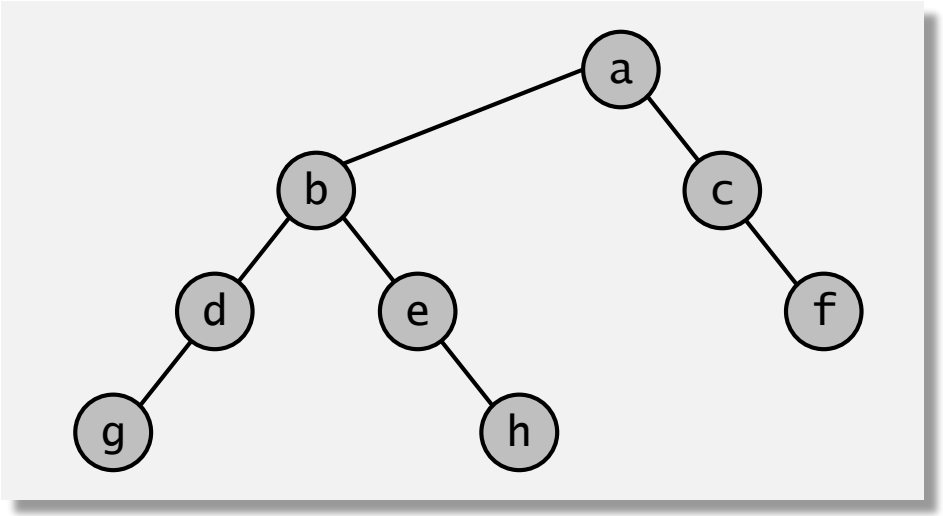
a b c d e f g h

# پیمایش درخت دودویی به ترتیب سطح



- پیمایش به ترتیب سطح.

```
private void levelOrder()
{
   if (isEmpty()) return;
   Queue<Item> q = new Queue<Item>();
   q.enqueue(root);
   while (!q.isEmpty())
   {
      Node x = q.dequeue();
      StdOut.print(x.item + " ");
      if (x.left  != null) q.enqueue(x.left);
      if (x.right != null) q.enqueue(x.right);
}
```

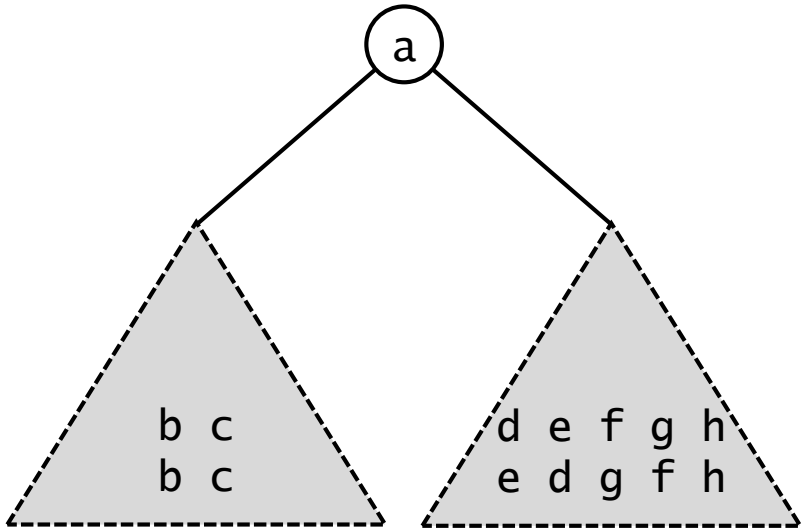



a b c d e f g h

# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



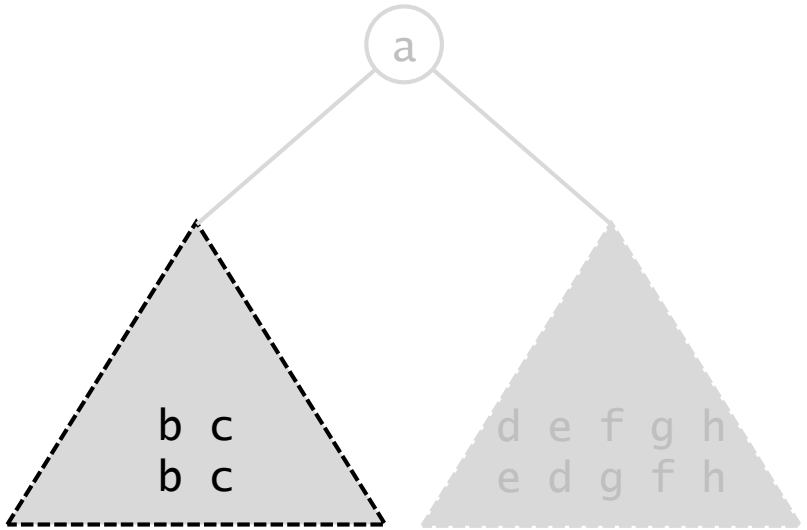
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | d | e | f | g | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



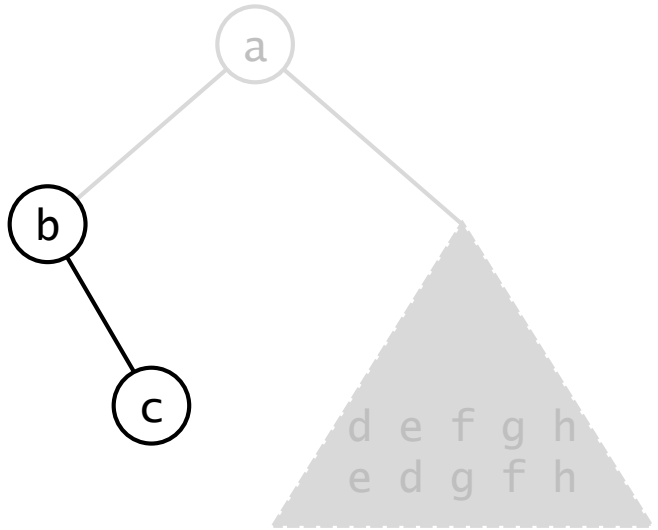
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



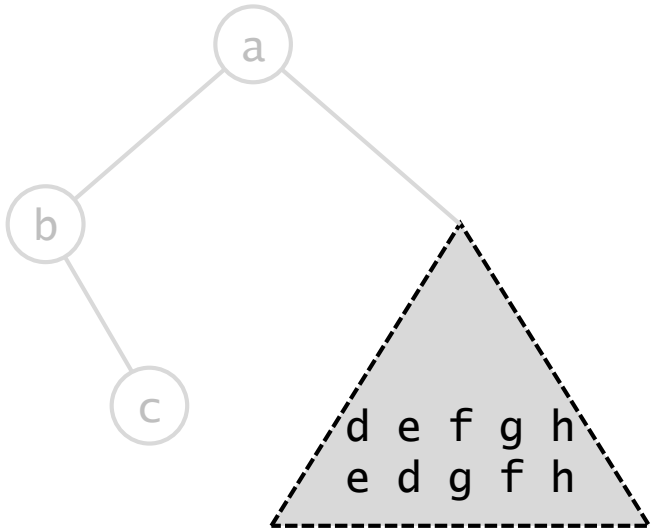
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



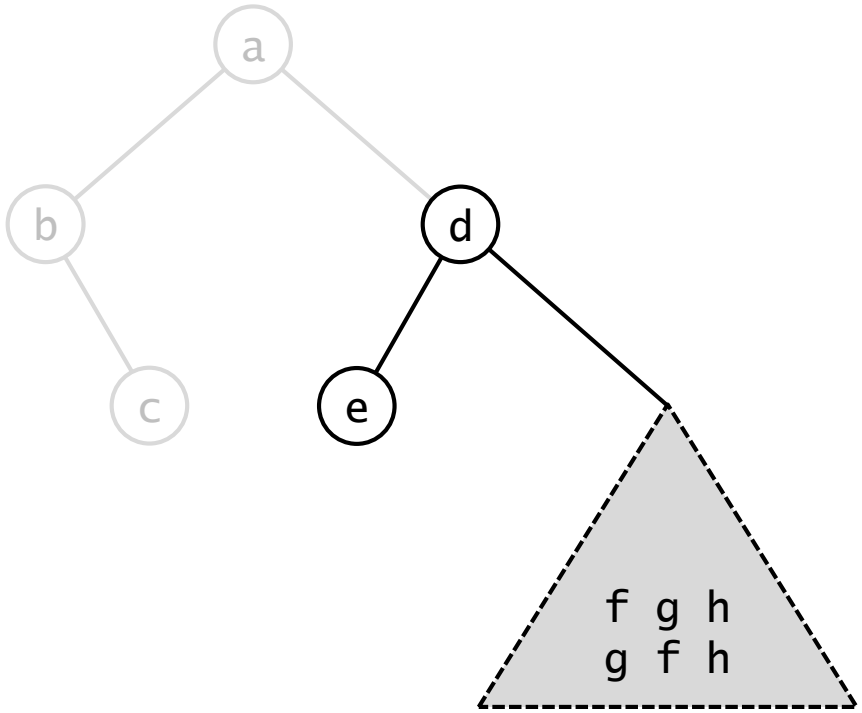
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



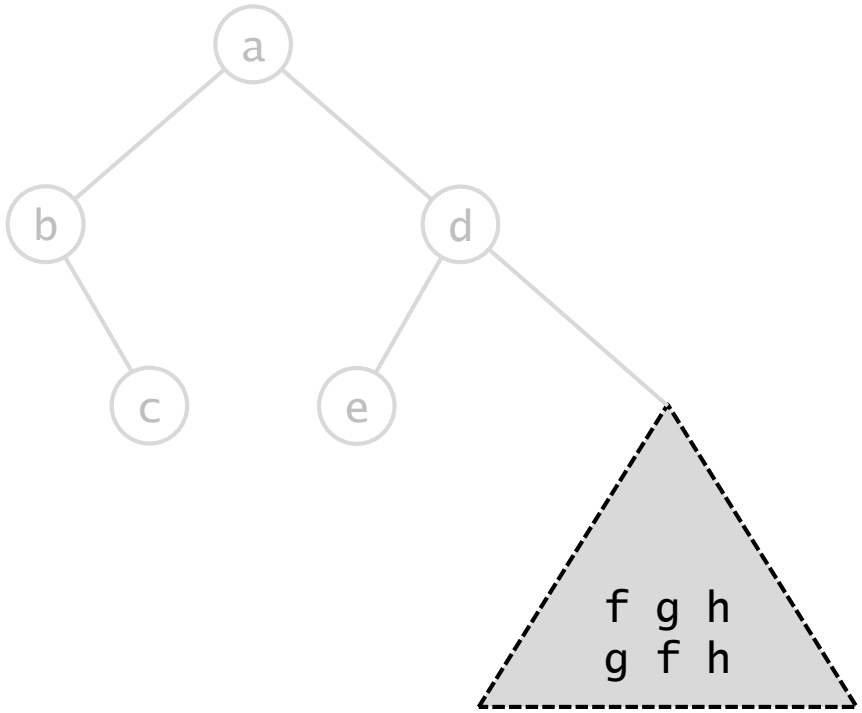
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



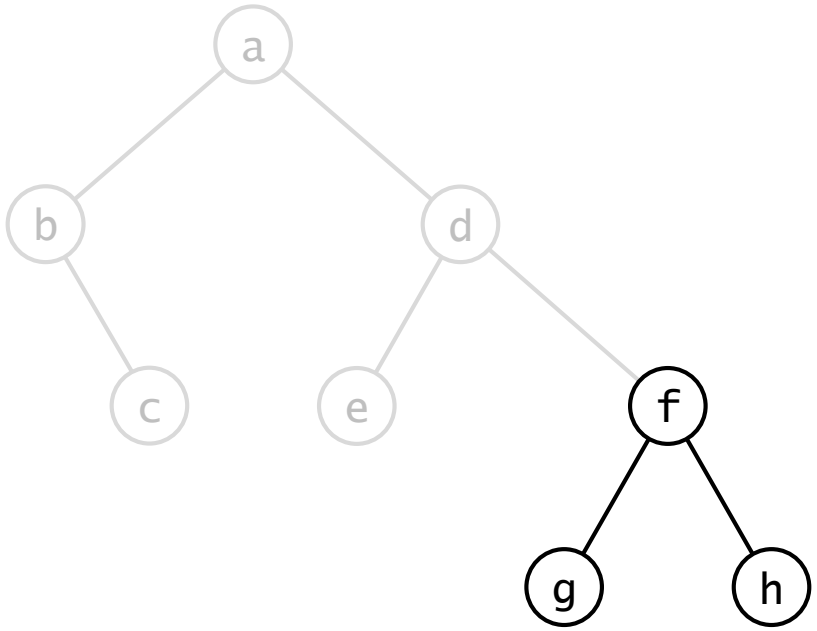
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | g | f | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



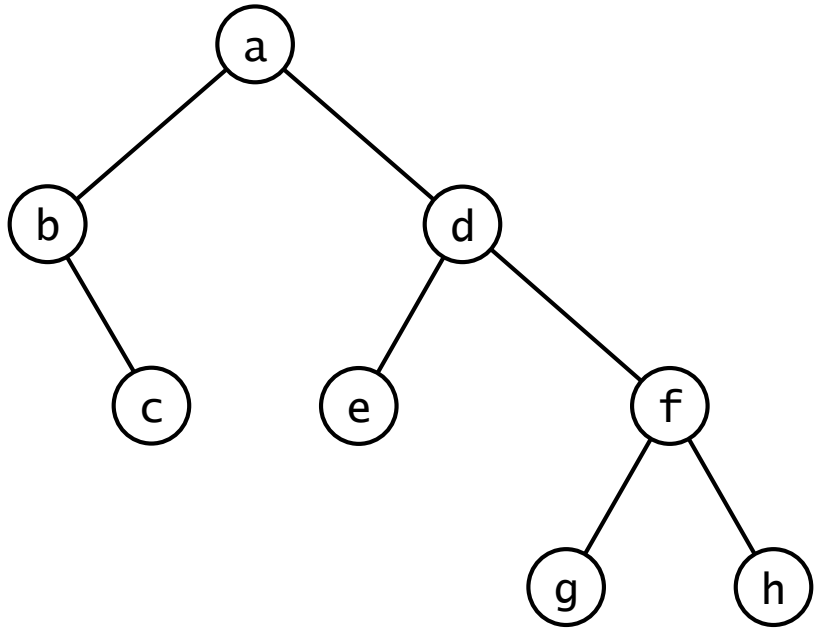
| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |



# مسئله: رسم درخت دودویی به کمک پیمایش‌های آن



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| pre | a | b | c | d | e | f | g | h |
| in | b | c | a | e | d | g | f | h |
| pos | 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



```
public BinaryTree<Item> copy()
{
   BinaryTree<Item> T = new BinaryTree<Item>();
   T.root = copy(root);
   return T;
}
```

```
private Node copy(Node x)
{
   if (x == null) return null;
   Node y = new Node(x.item);
   y.left  = copy(x.left );
   y.right = copy(x.right);
   return y;
}
```

# تمرین



- تابعی به صورت زیر به منظور بررسی تساوی دو زیردرخت دودویی بنویسید.

```
private boolean equals(Node x, Node y) { … }
```

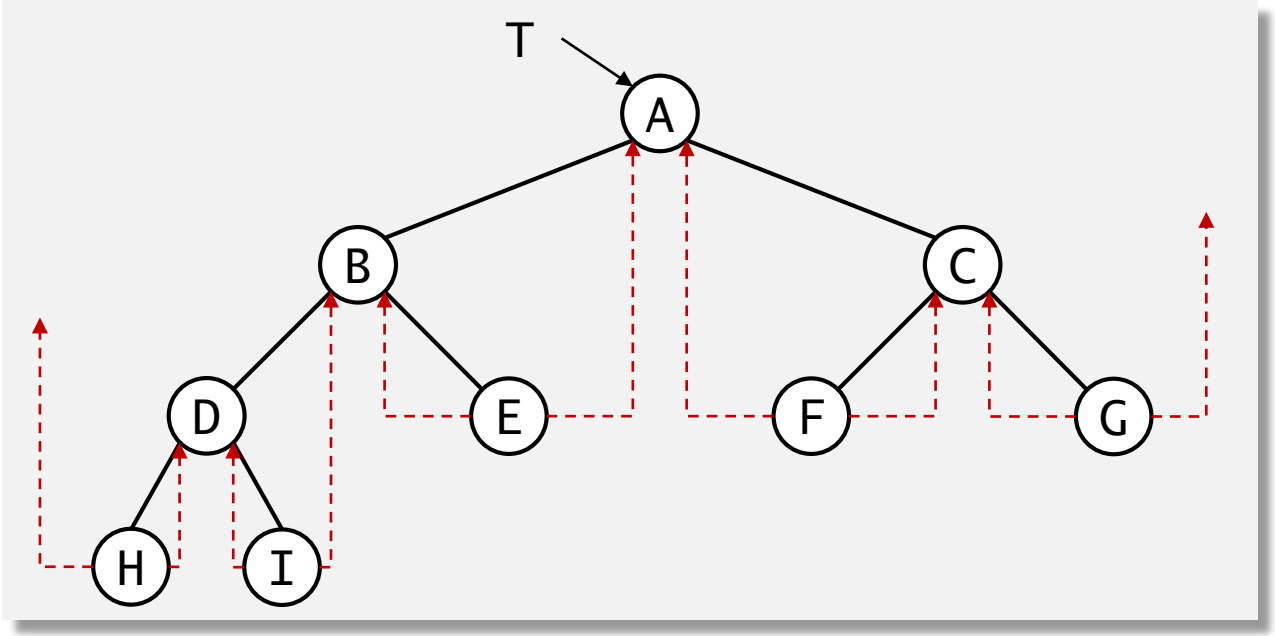
- تابعی به صورت زیر بنویسید به طوری که برای هر گره در درخت دودویی، جای دو زیردرخت چپ و راست آن گره را با هم تعویض نماید.

```
private Node swap(Node x) { … }
```

# درخت نخی دودویی



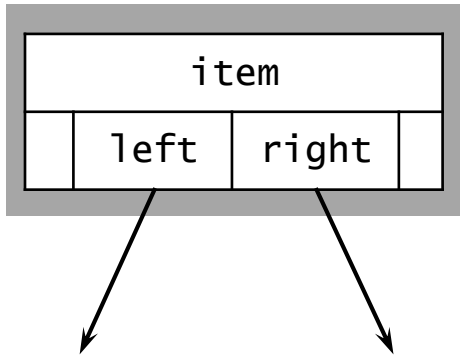
- انگیزه. استفاده‌ی هوشمند از فیلدهای اشاره‌گر پوچ.
  - در یک درخت دودویی با n گره، 2n فیلد اشاره‌گر وجود دارد به طوری که n − 1 فیلد غیر پوچ و n + 1 فیلد دارای مقدار پوچ هستند.
  - هر فیلد پوچ می‌تواند به گره بعدی (در یک پیمایش خاص) اشاره کند.



# پیاده‌سازی درخت نخی دودویی



```
private class Node
{
   private Item item;
   private Node left;
   private Node right;
   private boolean leftThread;
   private boolean rightThread;
   ...
}
```



# درخت عبارت

# درخت عبارت



- عبارت کاملاً پرانتزی.

```
E ::= a | (α E) | (E β E)
a ::= variable
α ::= unary operator [~, log, sin, cos, …]
β ::= binary operator [^, ×, /, +, -, …]
```

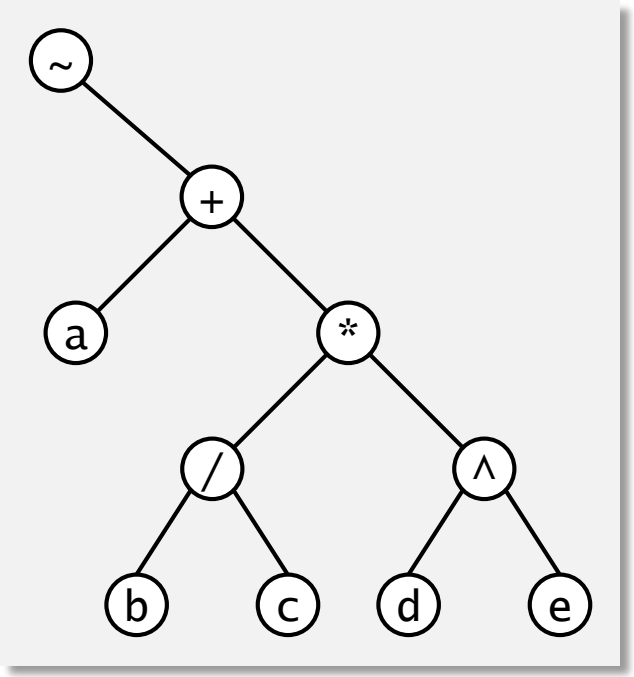
- مثال.

| ( | ~ | ( | a | + | ( | ( | b | / | c | ) | * | ( | d | ^ | e | ) | ) | ) | ) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|

# درخت عبارت



- بهترین مدل برای یک عبارت ریاضی استفاده از درخت است.

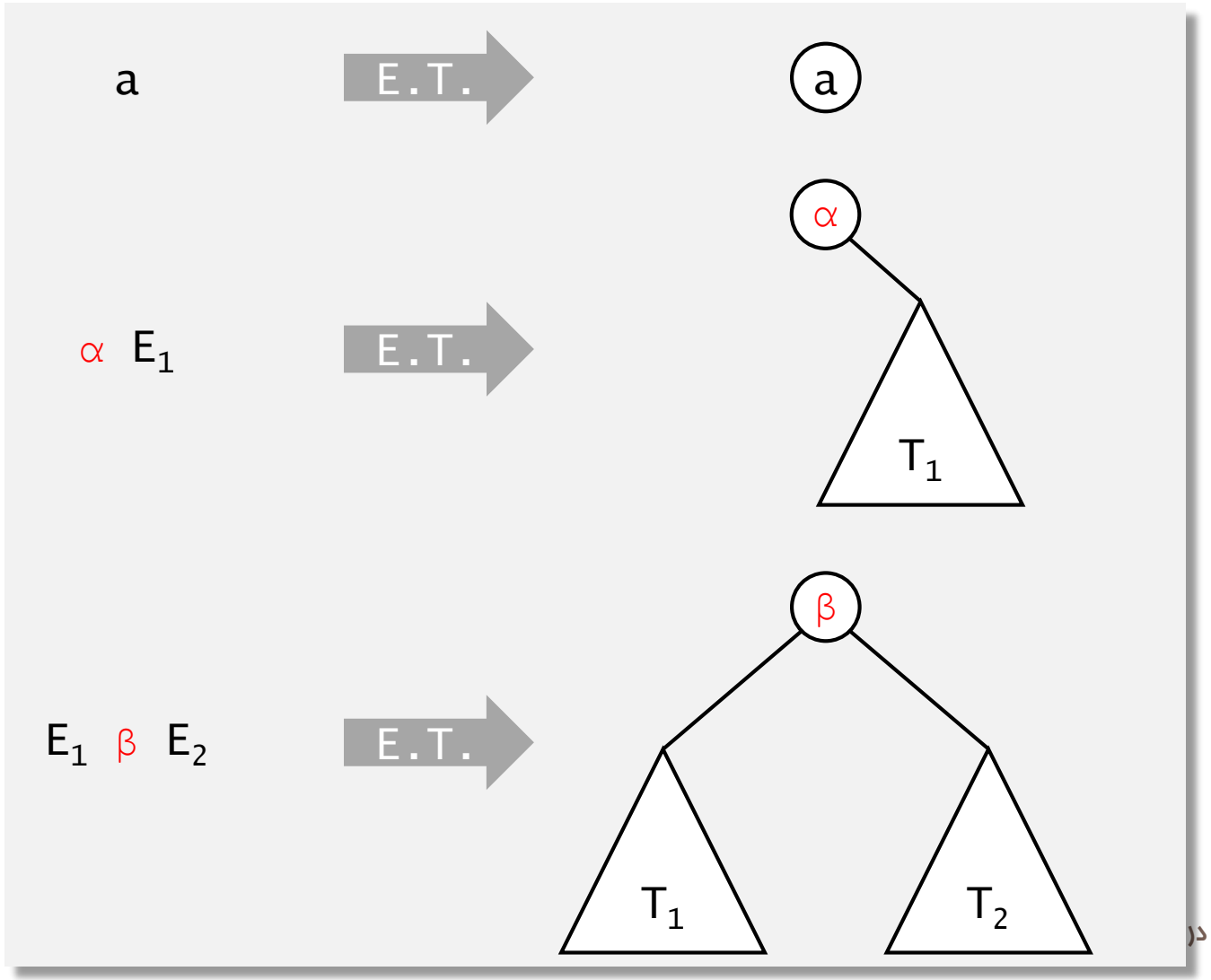



( ~ ( a + ( ( b / c ) * ( d ^ e ) ) ) )

- نکته. پیمایش‌های پیش‌ترتیب، میان‌ترتیب و پس‌ترتیب درخت عبارت به ترتیب معادل فرمهای پیشوندی، میانوندی و پسوندی عبارت هستند.

# تبدیل عبارت کاملاً پرانتزی به درخت عبارت





# الگوریتم تبدیل عبارت کاملاً پرانتزی به درخت عبارت

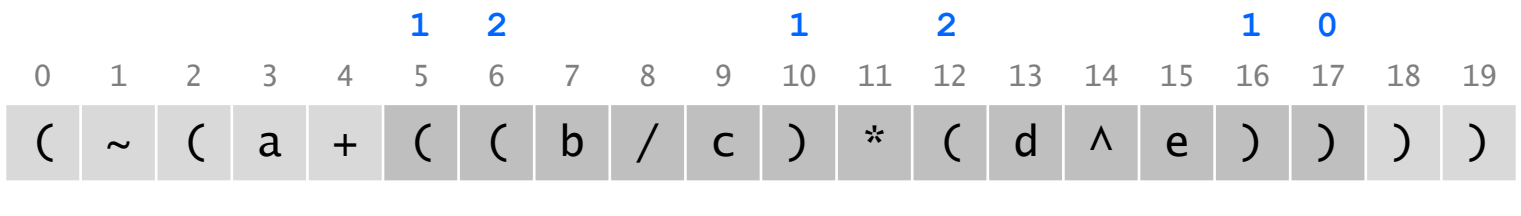


```
private Node expressionTree(String[] a, int lo, int hi)
{
   if (lo == hi) return New Node(a[lo]);
   else if (unaryOps.contains(a[lo + 1]))
   {
      Node x  = new Node(a[lo + 1]);
      x.right = expressionTree(a, lo + 2, hi - 1);
      return x;
   }
   else {
      int k   = match(a, lo + 1);
      Node x  = new Node(a[k + 1]);
      x.left  = expressionTree(a, lo + 1, k);
      x.right = expressionTree(a, k  + 2, hi - 1);
      return x;
   }
}
```

# الگوریتم تبدیل عبارت کاملاً پرانتزی به درخت عبارت



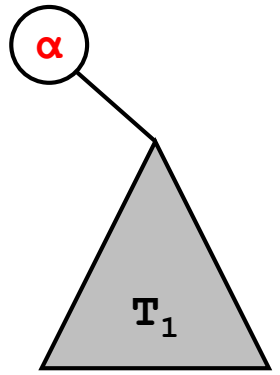
```
private int match(String[] a, int lo)
{
   int count = 0;
   int j = lo;
   while (true)
   {
      if      (a[j].equals("(")) count++;
      else if (a[j].equals(")")) count--;
      j++;
      if (count == 0) break;
   }
   return j - 1;
}
```



# حالت ۲



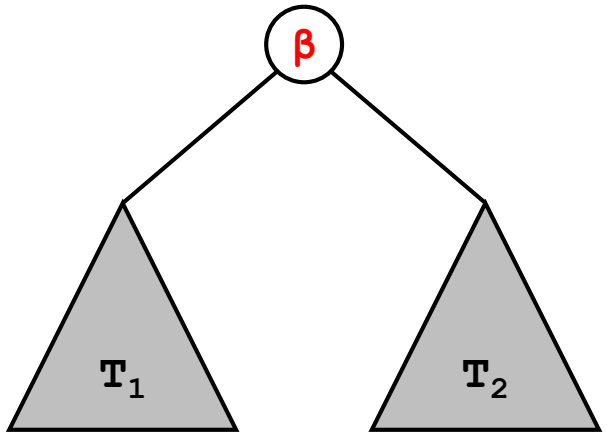
```
else if (unaryOps.contains(a[lo + 1]))
{
   Node x  = new Node(a[lo + 1]);
   x.right = expressionTree(a, lo + 2, hi - 1);
   return x;
}
```




# حالت ۳

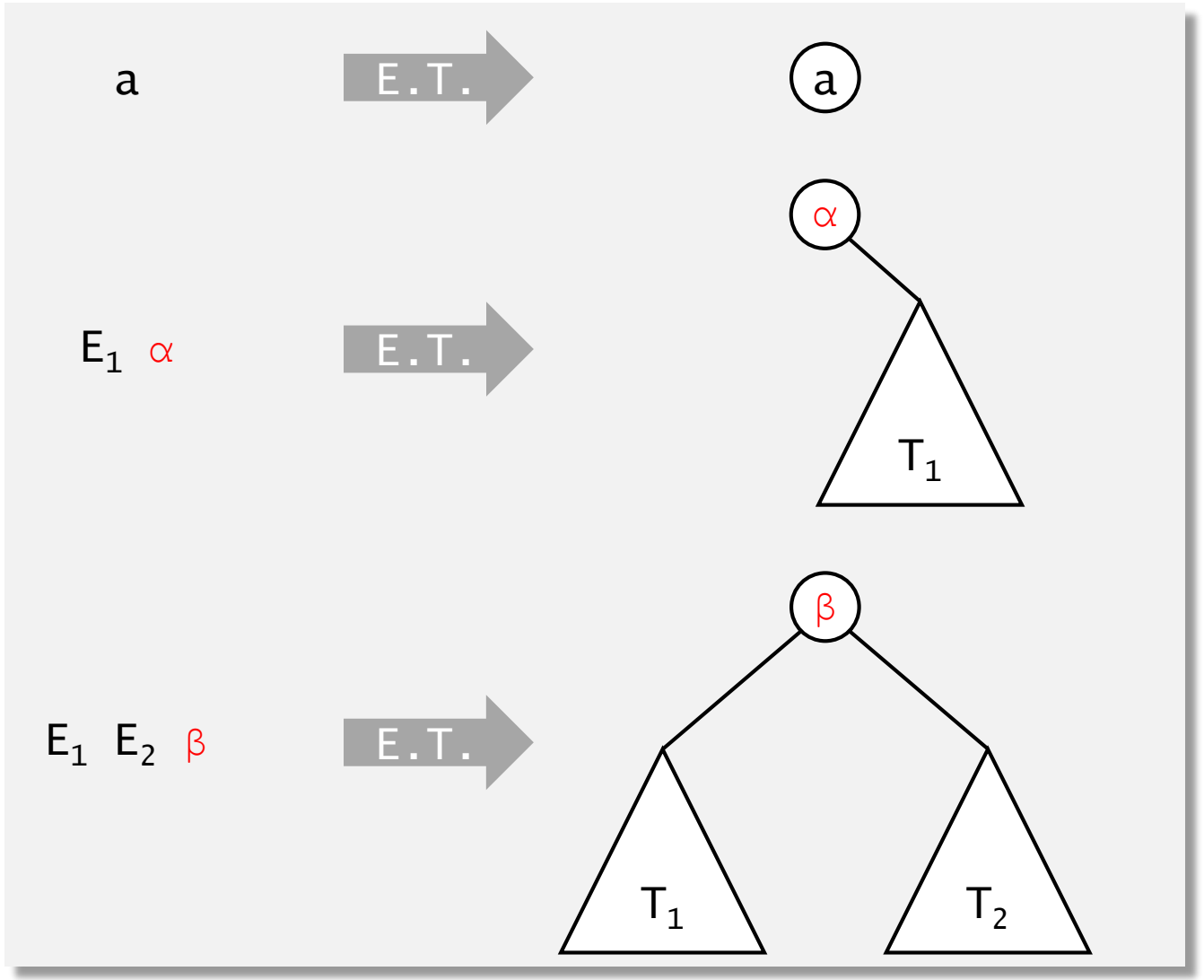


```
else {
      int k   = match(a, lo + 1);
      Node x  = new Node(a[k + 1]);
      x.left  = expressionTree(a, lo + 1, k);
      x.right = expressionTree(a, k  + 2, hi - 1);
      return x;
   }
```




# تمرین: ایجاد درخت عبارت از روی فرم پسوندی





# تبدیل درخت عبارت به فرم کاملاً پرانتزی



```
public void printInfix(Node x);
{
   if (unaryOps.contains(x.item)) {
      StdOut.print("(" + x.item);
      printInfix(x.right);
      StdOut.print(")");
   }
   else if (binaryOps.contains(x.item)) {
      StdOut.print("(");
      printInfix(x.left);
      StdOut.print(x.item);
      printInfix(x.right);
      StdOut.print(")");
   }
   else
      StdOut.print(x.item);
}
```

# الگوریتم فشرده‌سازی هافمن



- مسئله.
  - ورودی: تعدادی کاراکتر و احتمال وقوع هر یک از کاراکترها.
  - خروجی: کدهای کاراکترها به طوری که ضریب فشرده‌سازی حداقل شود.

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

- مثال.

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

# طرح اول برای کد گذاری: کد دودویی با طول ثابت



- در این روش اگر n کاراکتر مختلف داشته باشیم، به هر کدام یک کد دودویی با طول ثابت زیر نسبت می‌دهیم:

| کاراکتر | فراوانی | کد دودویی 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

کتاب کد

$$\text{طول کد} = \lceil \lg n + 1 \rceil$$

- محاسبه‌ی اندازه‌ی فایل کد شده [بر حسب بیت]:

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

# طرح دوم برای کد گذاری: کد دودویی با طول متغیر



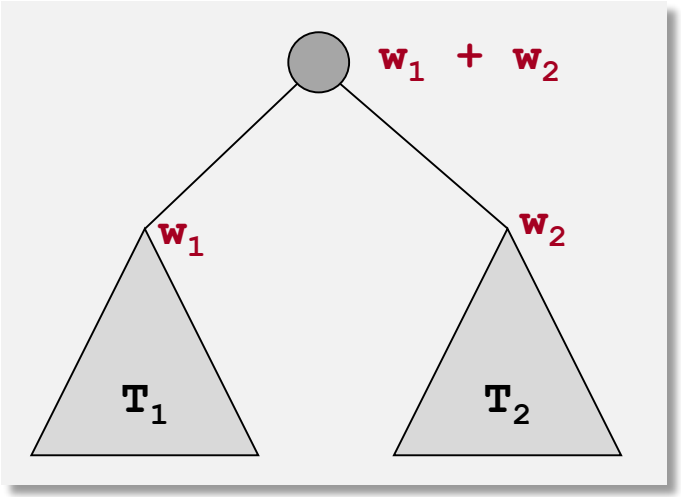
- انگیزه. استفاده از کلمات کد با طول کمتر برای کاراکترهایی که فراوانی بیشتری دارند.

- راه حل. استفاده از روش کدگذاری هافمن:
  - مرحله ۱) به ازای هر کاراکتر یک درخت دودویی تک گره‌ای ایجاد و وزن ریشه‌ی آن درخت را برابر با احتمال کاراکتر مربوطه قرار می‌دهیم.

# 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 می‌کنیم.
  - یک درخت دودویی جدید ایجاد می‌کنیم که $T_1$ و $T_2$ زیردرختان چپ و راست آن باشند و وزن ریشه‌ی آن را برابر با $w_1 + w_2$ قرار می‌دهیم.

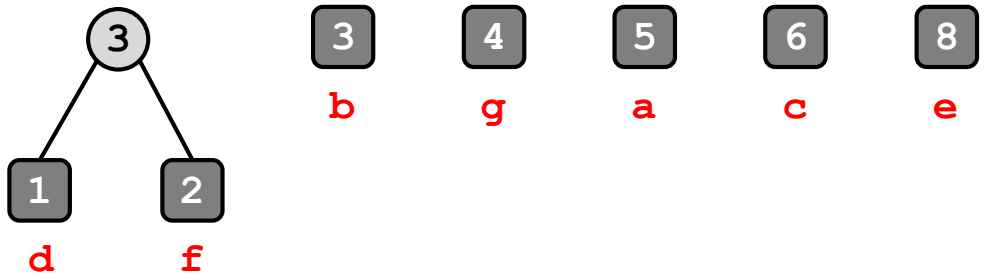


# روش کد گذاری هافمن: مرحله ۲



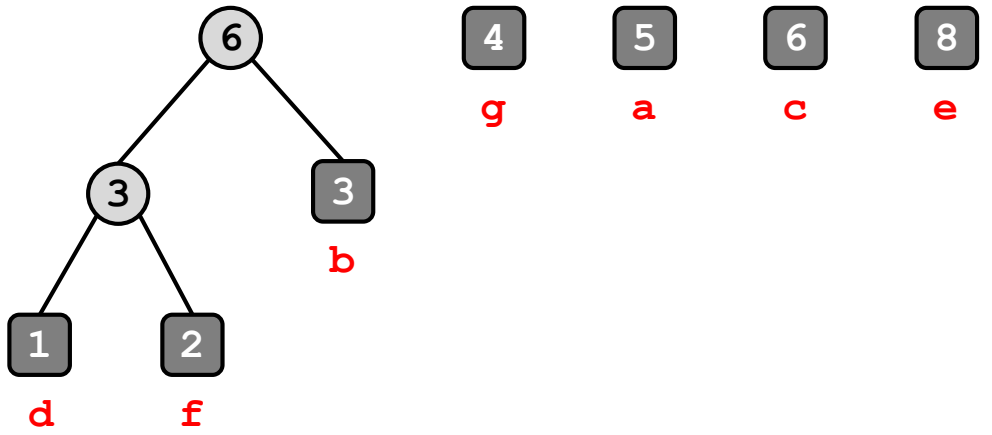
# روش کد گذاری هافمن: مرحله ۲





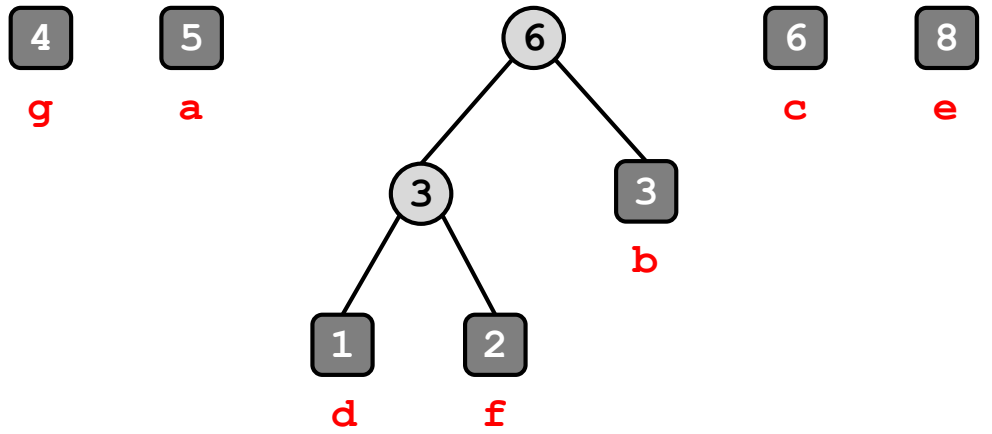
# روش کد گذاری هافمن: مرحله ۲





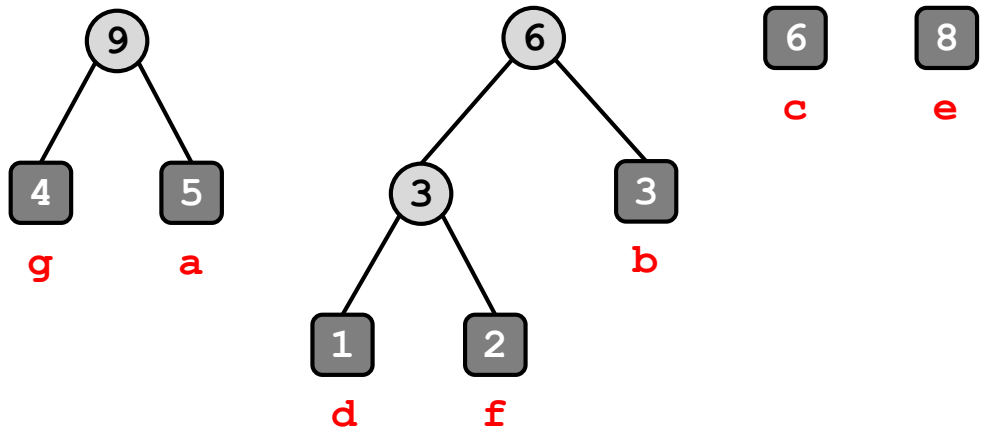
# روش کد گذاری هافمن: مرحله ۲





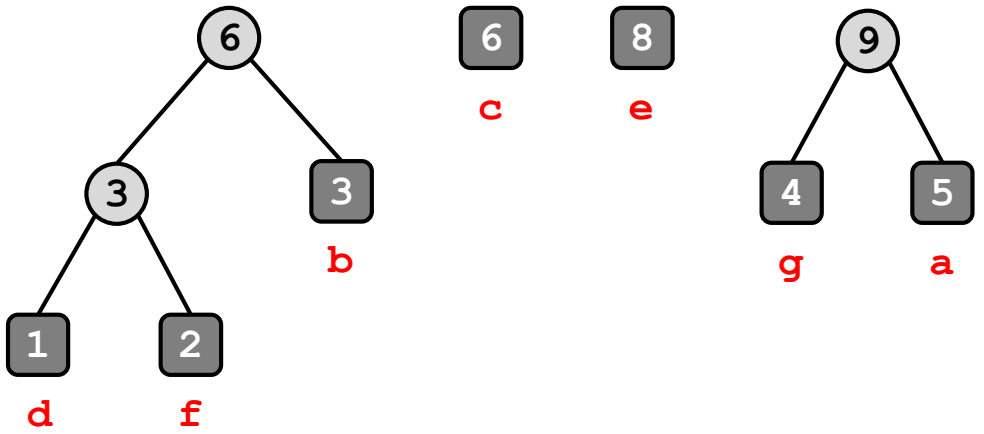
# روش کد گذاری هافمن: مرحله ۲





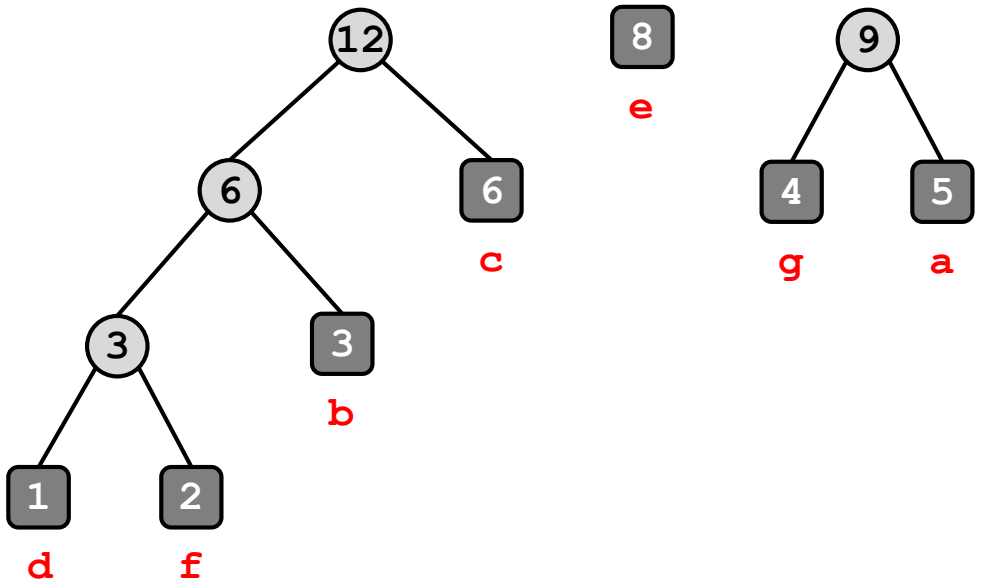
# روش کد گذاری هافمن: مرحله ۲





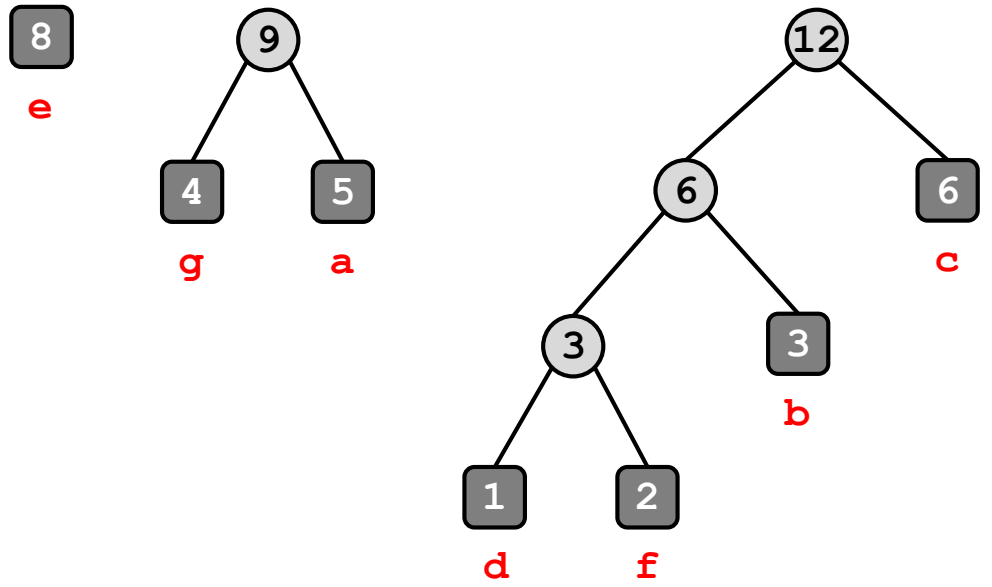
# روش کد گذاری هافمن: مرحله ۲





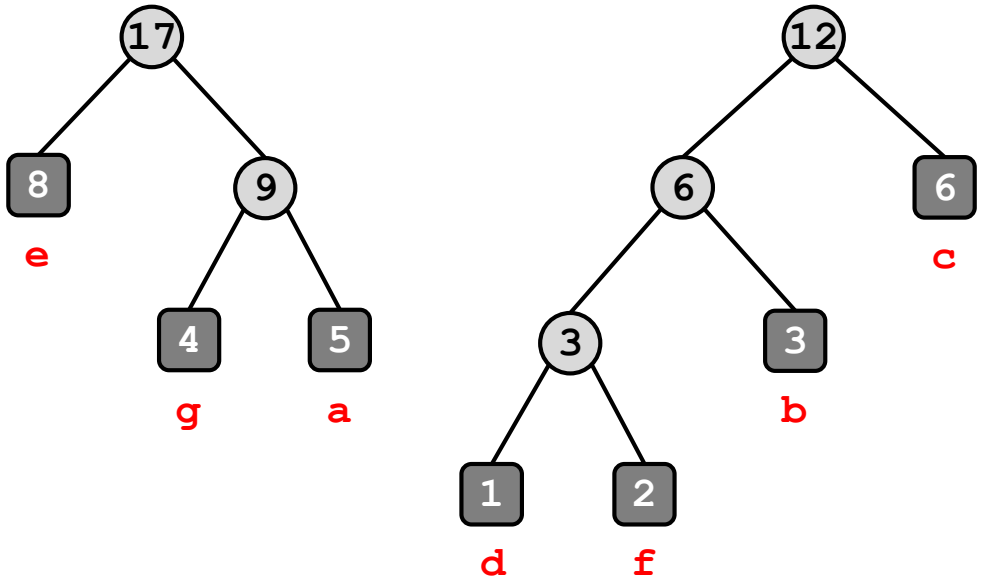
# روش کد گذاری هافمن: مرحله ۲





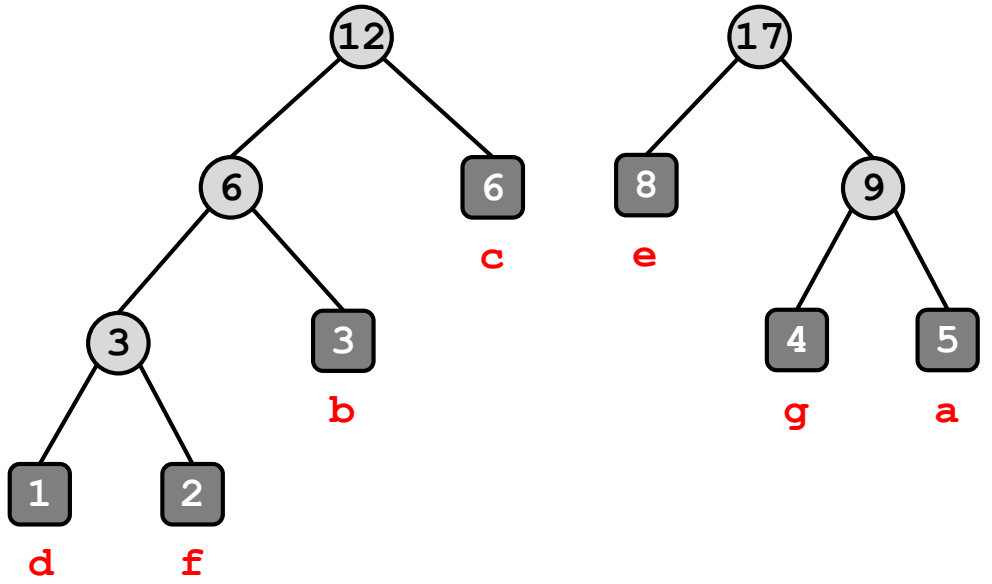
# روش کد گذاری هافمن: مرحله ۲





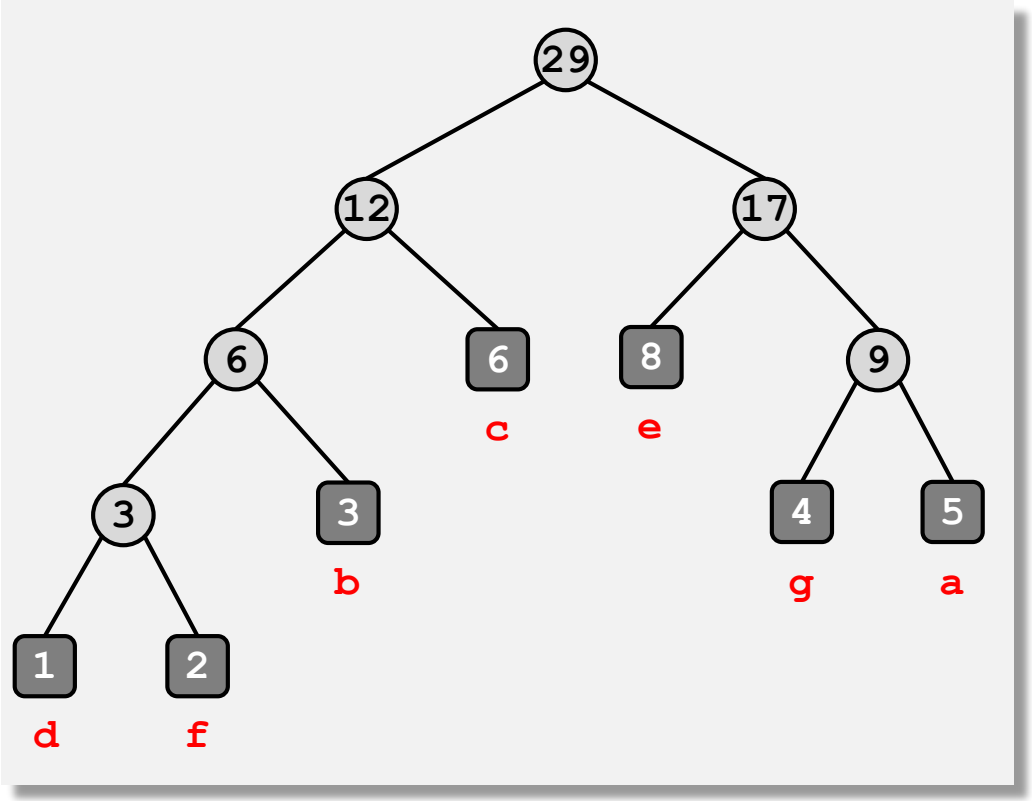
# روش کد گذاری هافمن: مرحله ۲





# روش کد گذاری هافمن: مرحله ۲

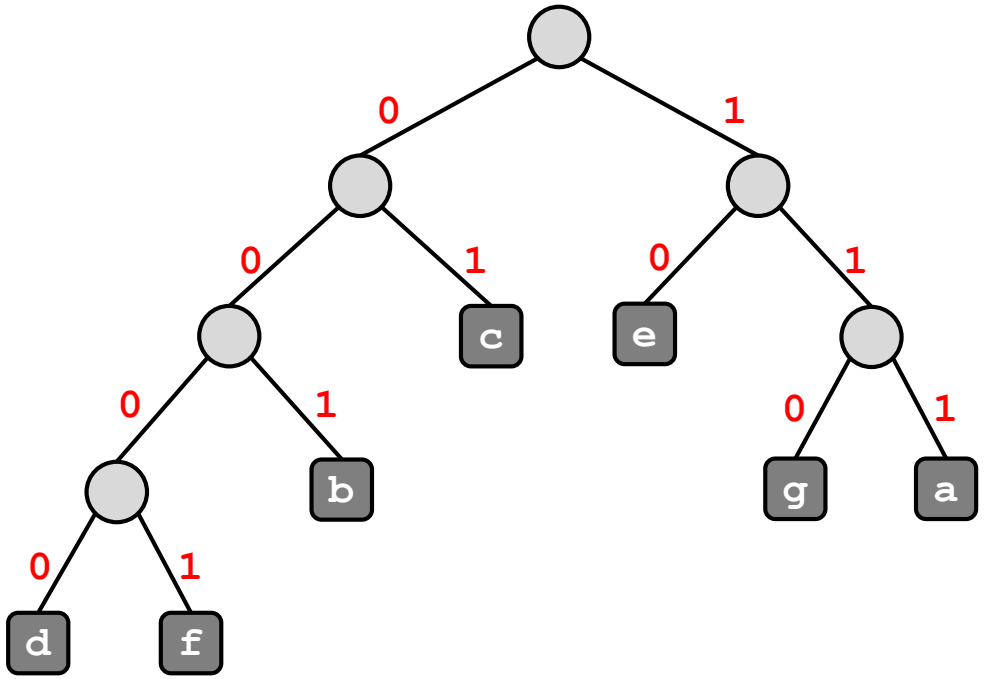




# روش کد گذاری هافمن



- مرحله ۳) تولید کد کاراکترها.
  - به شاخه‌های چپ برچسب 0 و به شاخه‌های راست برچسب 1 (و یا برعکس) می‌دهیم.



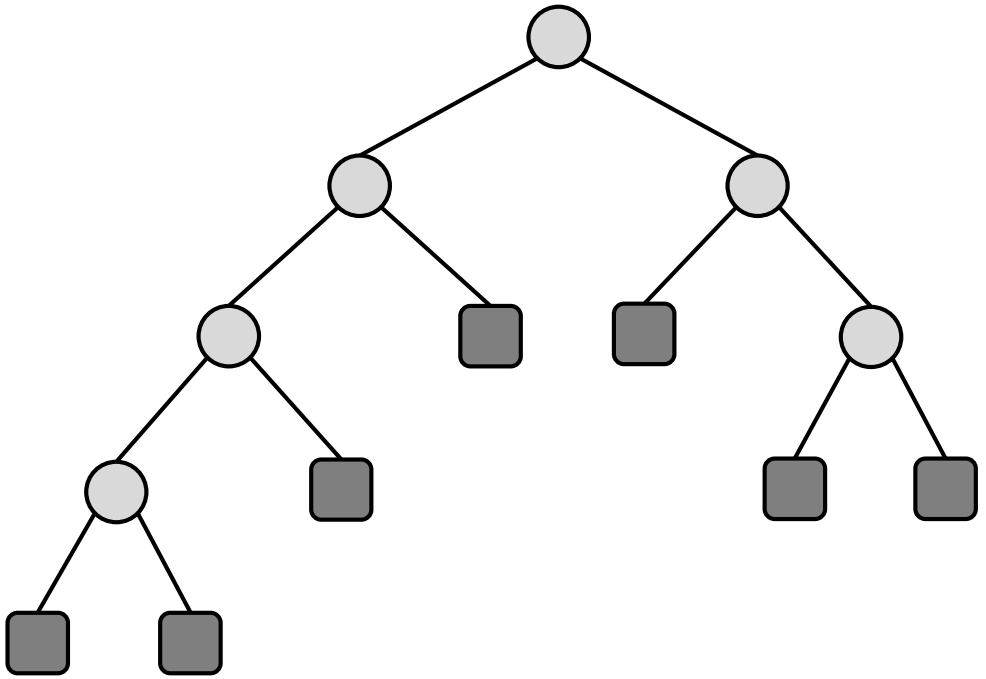

ضریب فشرده سازی = $\frac{76}{87}$

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

# یک قضیه مرتبط



- قضیه. یک درخت دودویی کامل با n گره برگ، دارای n - 1 گره داخلی است.



- اثبات. از طریق استقرا روی n
  - پایه استقرا به ازای n = 1 برقرار است

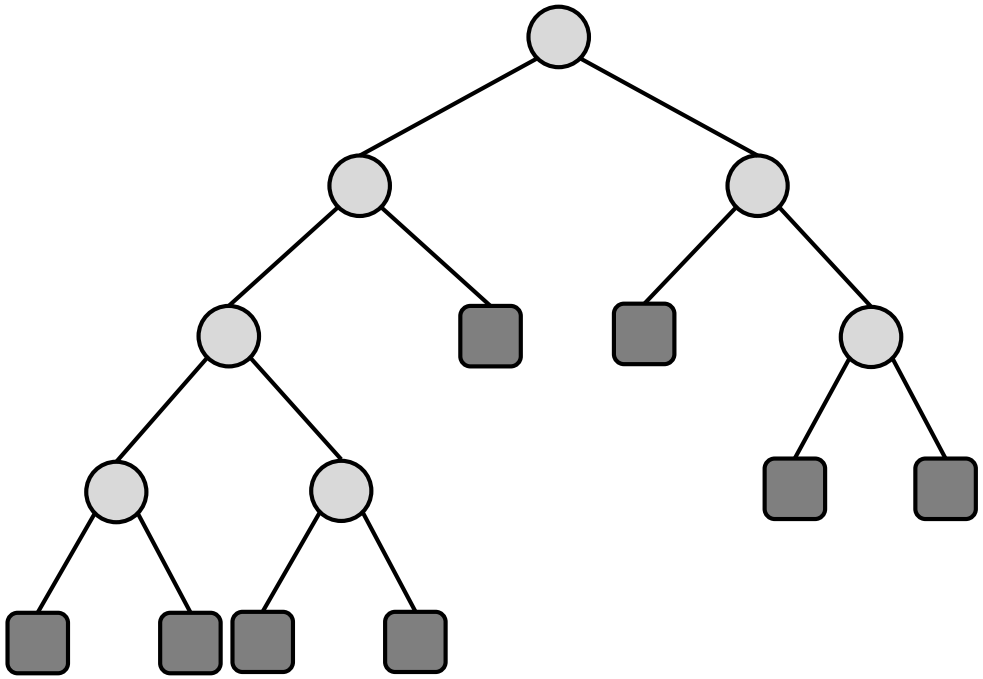
*تعداد برگها:* n

*تعداد گره‌های داخلی:* n - 1

# یک قضیه مرتبط



- قضیه. یک درخت دودویی کامل با n گره برگ، دارای n - 1 گره داخلی است.



- اثبات. از طریق استقرا روی n
  - پایه استقرا به ازای n = 1 برقرار است

*تعداد برگها:* n + 1

*تعداد گره‌های داخلی:* n

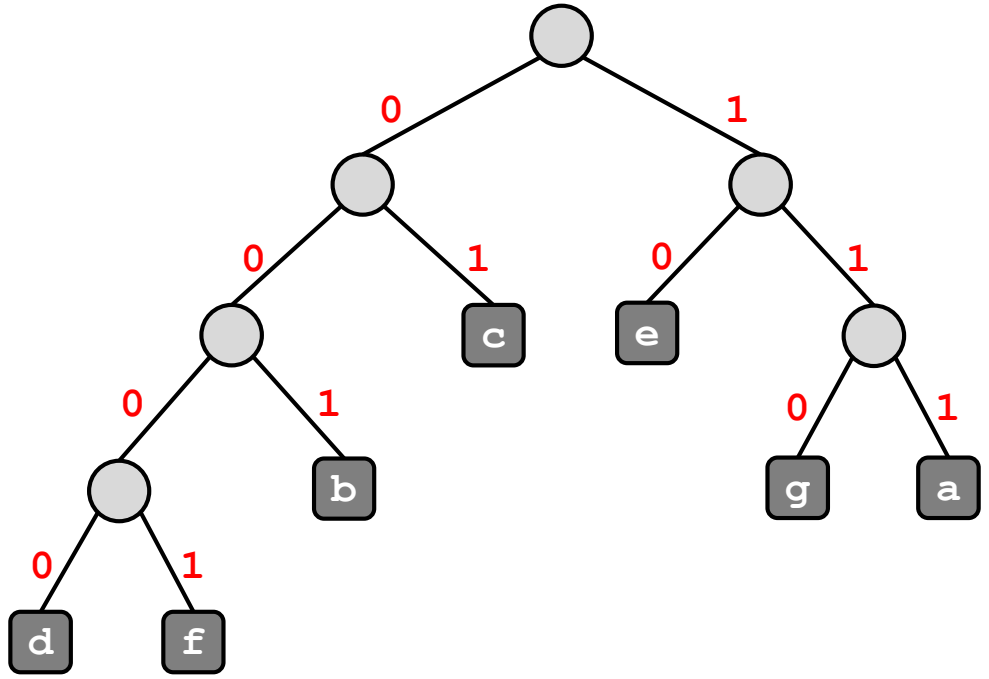
# ۲ نکته در مورد روش کد گذاری هافمن



- بهینگی کد هافمن.
  - روش کدگذاری هافمن بهینه است، یعنی با این روش ضریب فشرده‌سازی حداقل می‌شود.
- خاصیت پیشوندی.
  - کدهای به دست آمده خاصیت پیشوندی دارند،
  - یعنی در کتاب کد، هیچ کلمه‌ی کدی پیشوند کلمه‌ی کد دیگری نیست.
  - در نتیجه رمزگشایی با یک بار پویش فایل رمز شده به سادگی قابل انجام است.
- مثال. رشته‌ی 0001111000010 را رمزگشایی نمایید.

# رمزگشایی






| 0 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| f | | | | a | | | d | | | | e | |

# پیاده‌سازی الگوریتم هافمن



- تمرین. الگوریتم فشرده‌سازی هافمن را پیاده‌سازی نمایید.

| | root | weight |
|---|---|---|
| 1 | | |
| 2 | | |
| | . | . |
| | . | . |
| | . | . |
| n | | |

FOREST

| | symbol | prob | leaf |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| | . | . | . |
| | . | . | . |
| | . | . | . |
| n | | | |

ALPHABET

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

TREE