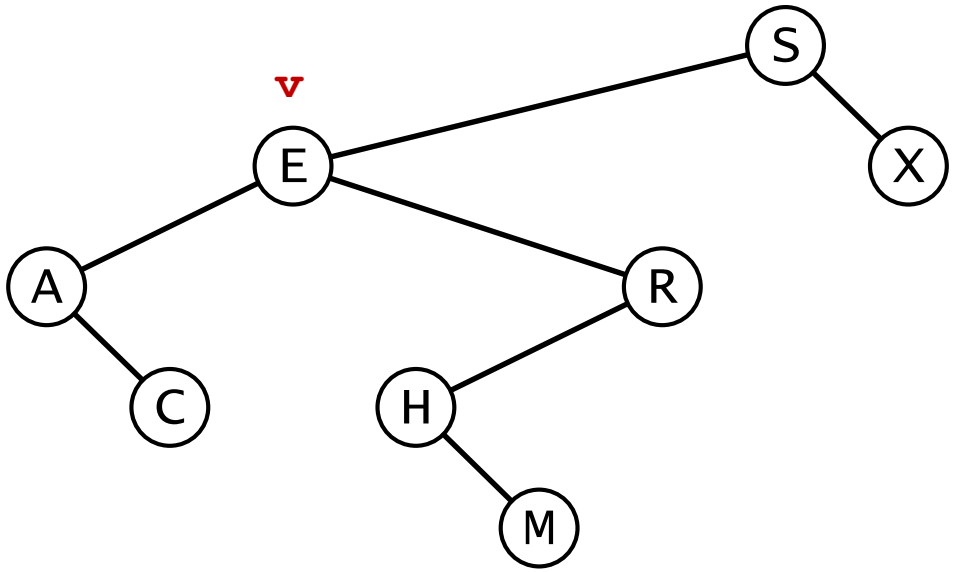
درخت جستجوی دودویی

سید ناصر رضوی www.snrazavi.ir
۱۳۹۵

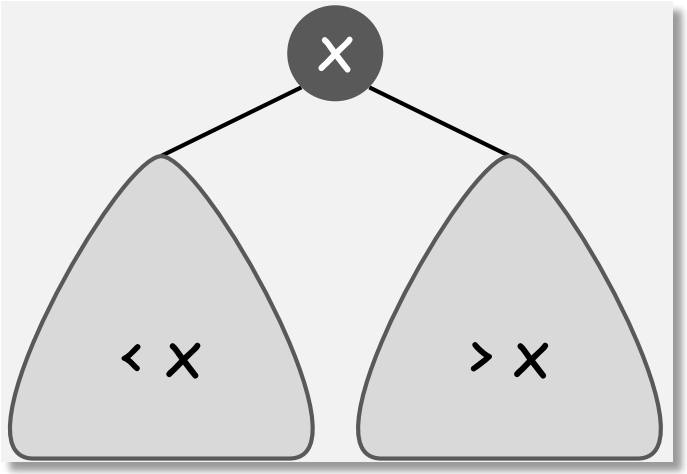
درخت جستجوی دودویی (BST)

- درخت جستجوی دودویی. یک درخت دودویی است به طوری که:
  - هر گره آن شامل یک زوج کلید-مقدار است.
  - به ازای هر گره مانند **v**،
    - کلیدهای ذخیره شده در زیردرخت چپ **v**، کوچک‌تر از کلید گره **v** هستند.
    - کلیدهای ذخیره شده در زیردرخت راست **v**، بزرگ‌تر از کلید گره **v** هستند.



درخت جستجوی دودویی: تعریف بازگشتی

- درخت جستجوی دودویی. یک درخت دودویی است که می‌تواند تهی باشد؛ ولی اگر تهی نباشد دارای ویژگی‌های زیر است:
  - هر گره دارای یک کلید (منحصر به فرد) است.
  - کلیدهای زیردرخت چپ (در صورت وجود) کوچک‌تر از کلید ریشه هستند.
  - کلیدهای زیردرخت راست (در صورت وجود) بزرگ‌تر از کلید ریشه هستند.
  - زیردرخت‌های چپ و راست، درخت جستجوی دودویی هستند.

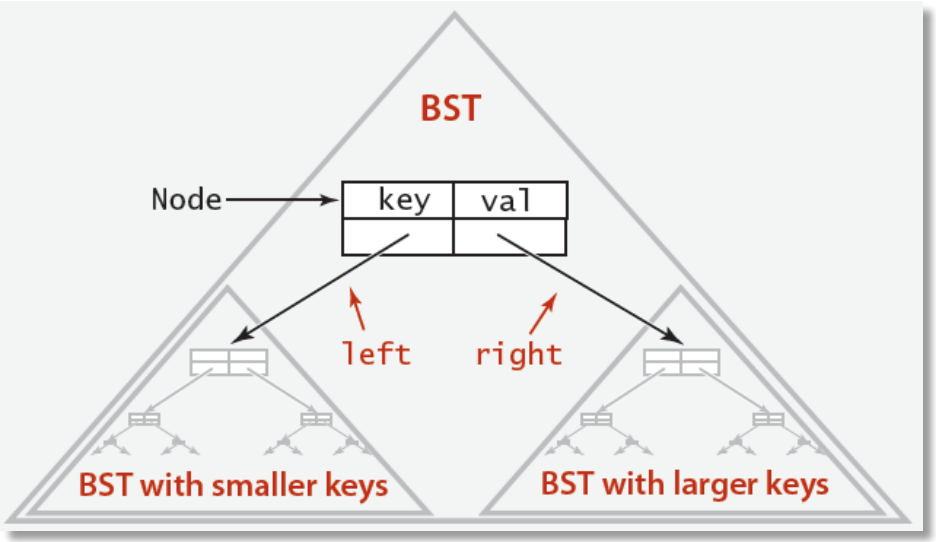


درخت جستجوی دودویی: جاوا

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

- هر گره حاوی چهار مقدار است:
  - یک کلید و مقدار متناظر با آن.
  - یک ارجاع به زیردرخت‌های چپ و راست.

```
private class Node
{
   private Key key;
   private Value val;
   private Node left, right;

   public Node(Key key, Value val)
   {
      this.key = key;
      this.val = val;
   }
}
```

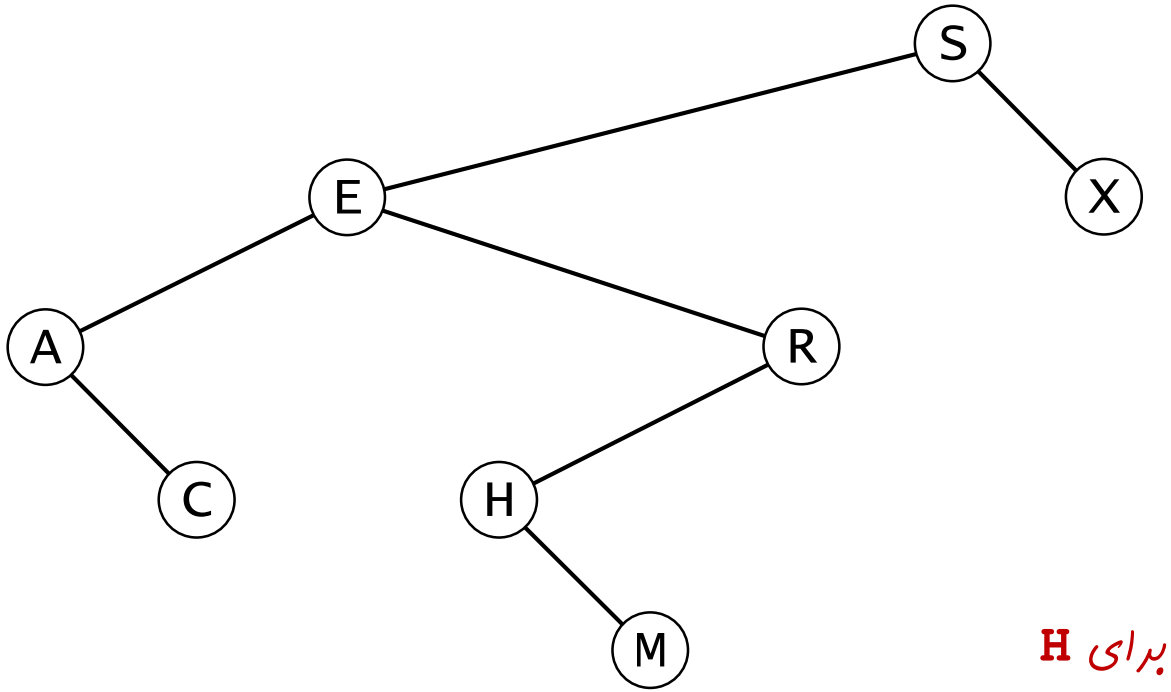


پیاده‌سازی درخت جستجوی دودویی


```
public class BST<Key extends Comparable<Key>, Value>
{
   private Node root;

   private class Node
   { /* see prev slides */ }

   public void put(Key key, Value val)
   { /* see next slides */ }

   public Value get(Key key)
   { /* see next slides */ }

   public void delete(Key key)
   { /* see next slides */ }

   public Iterable<Key> iterator()
   { /* see next slides */ }
}
```

درخت جستجوی دودویی: جستجو

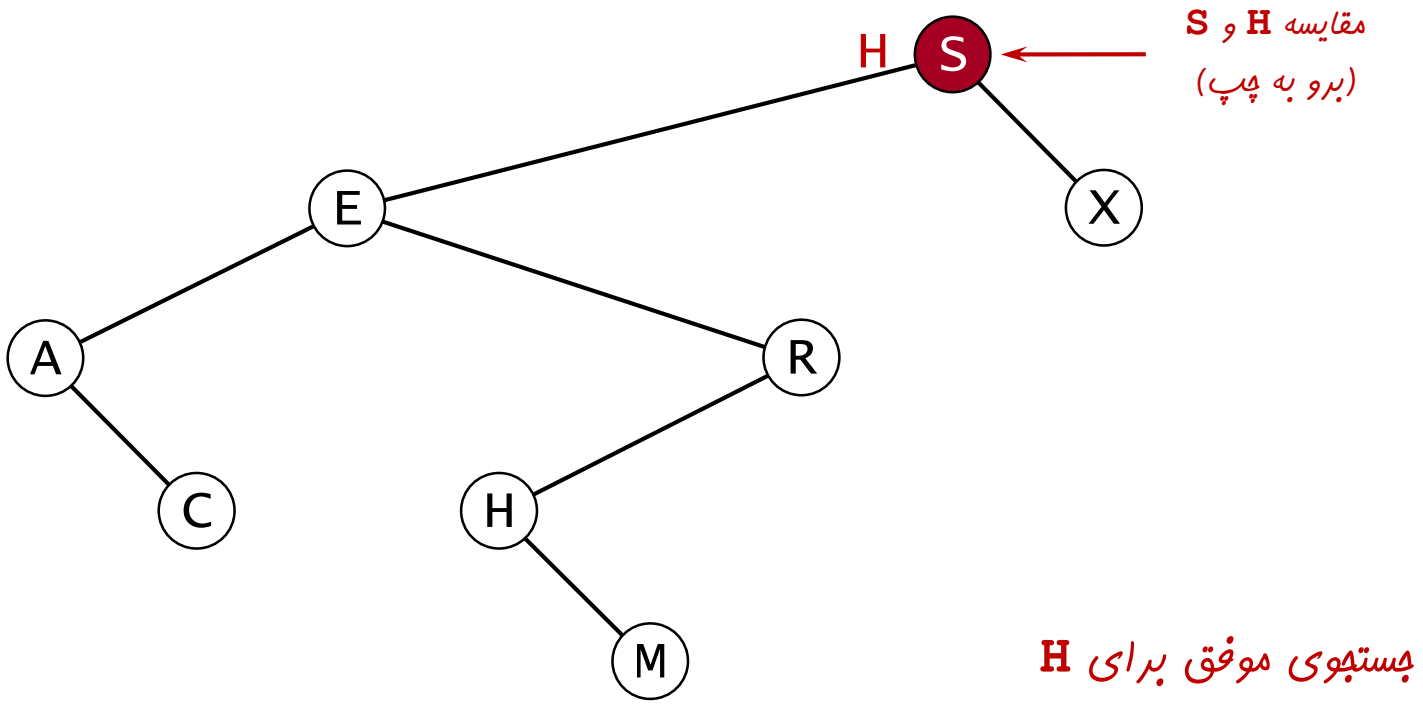
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی موفق برای H

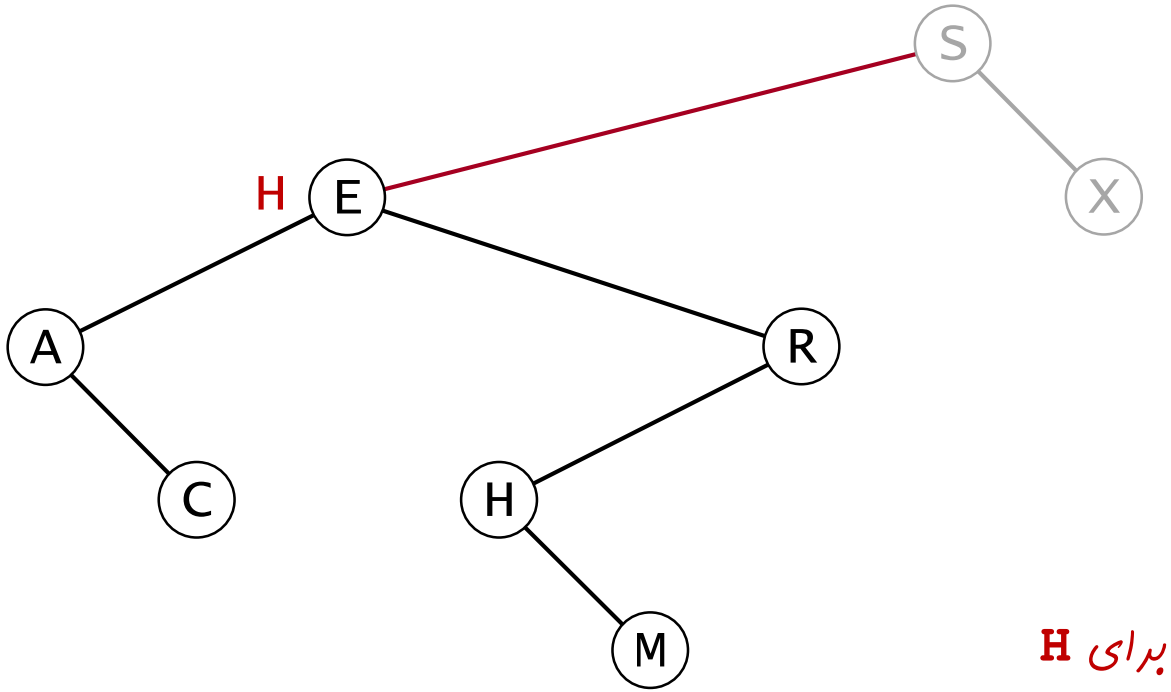
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

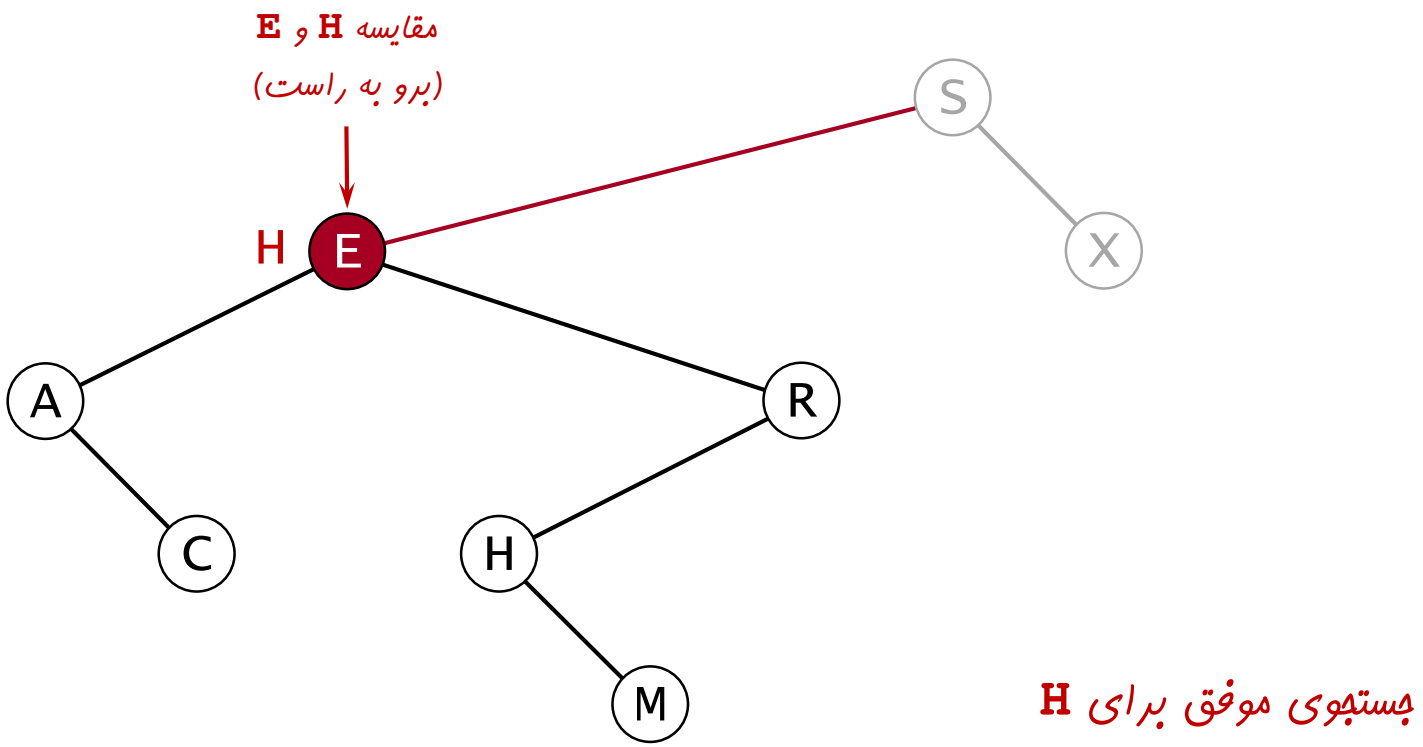
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی موفق برای H

درخت جستجوی دودویی: جستجو

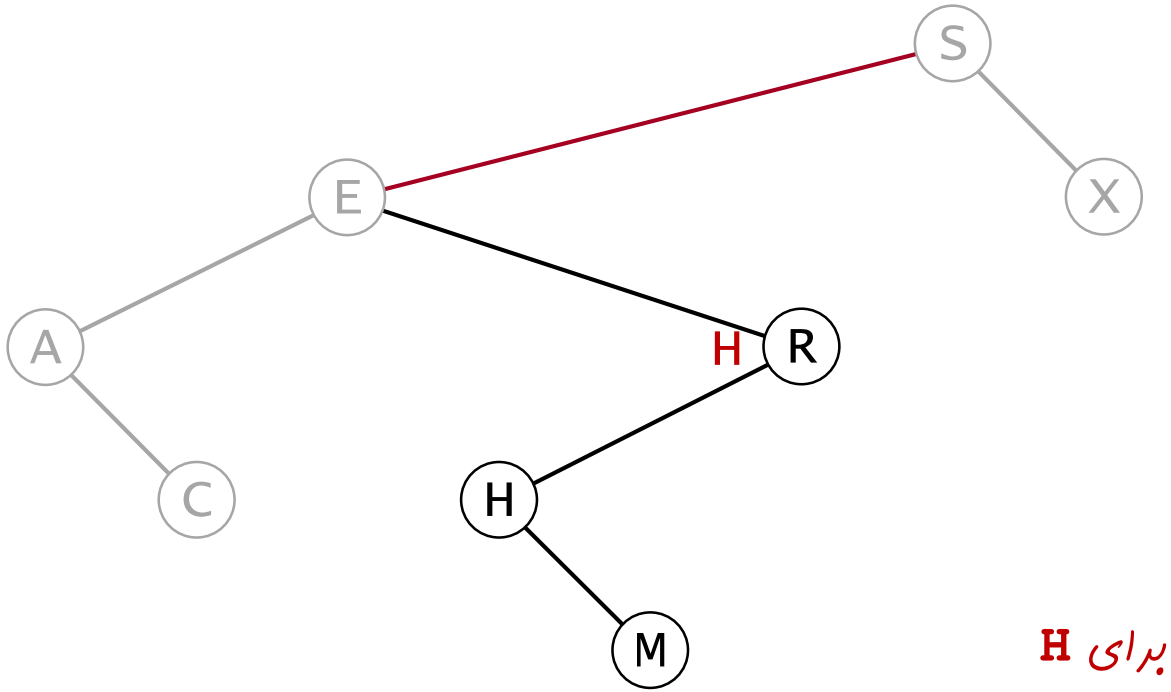
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی موفق برای H

درخت جستجوی دودویی: جستجو

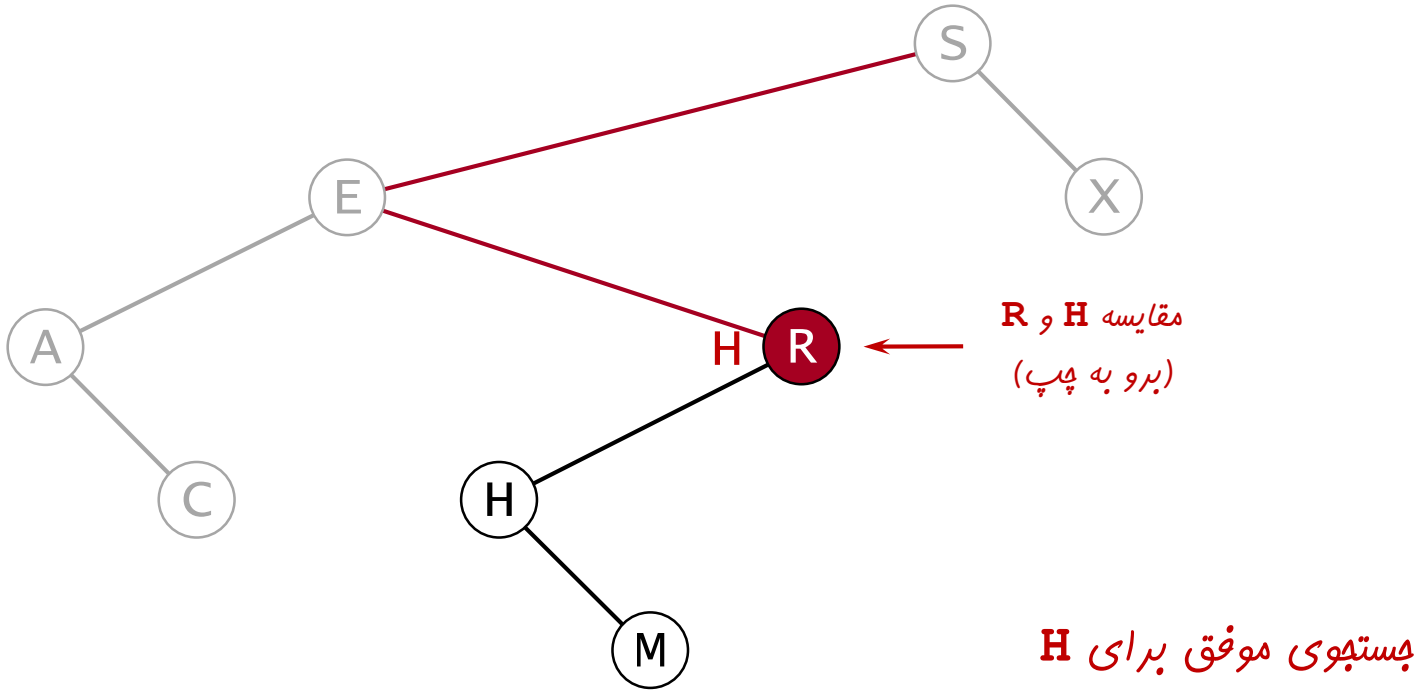
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی موفق برای H

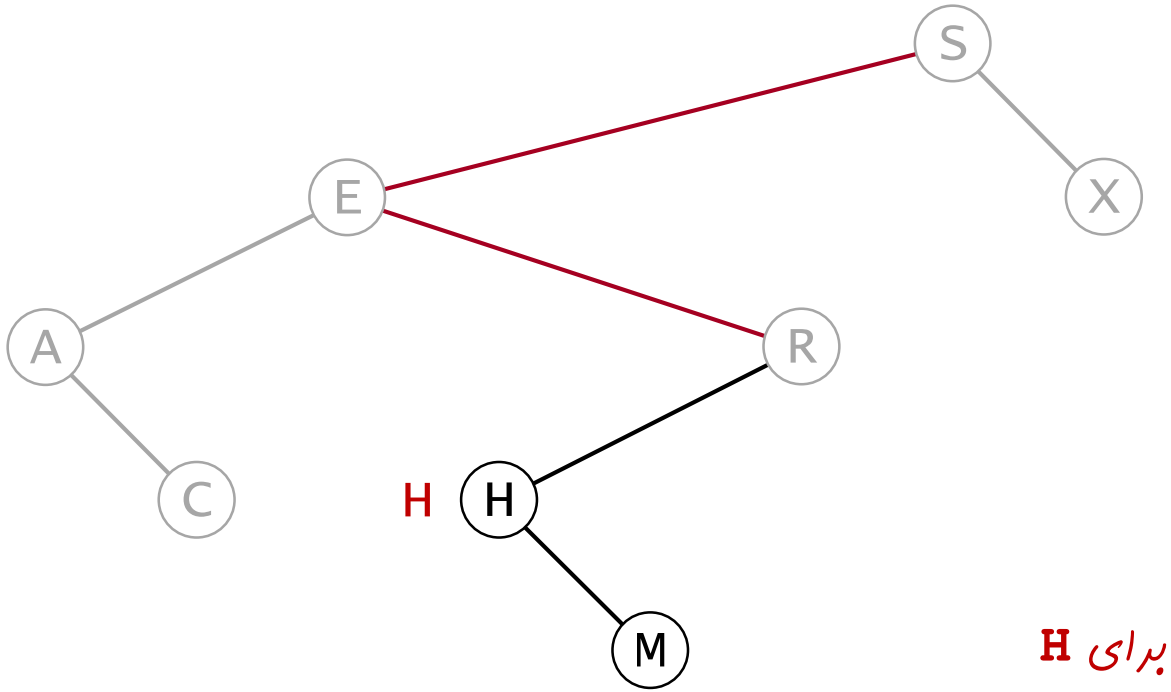
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

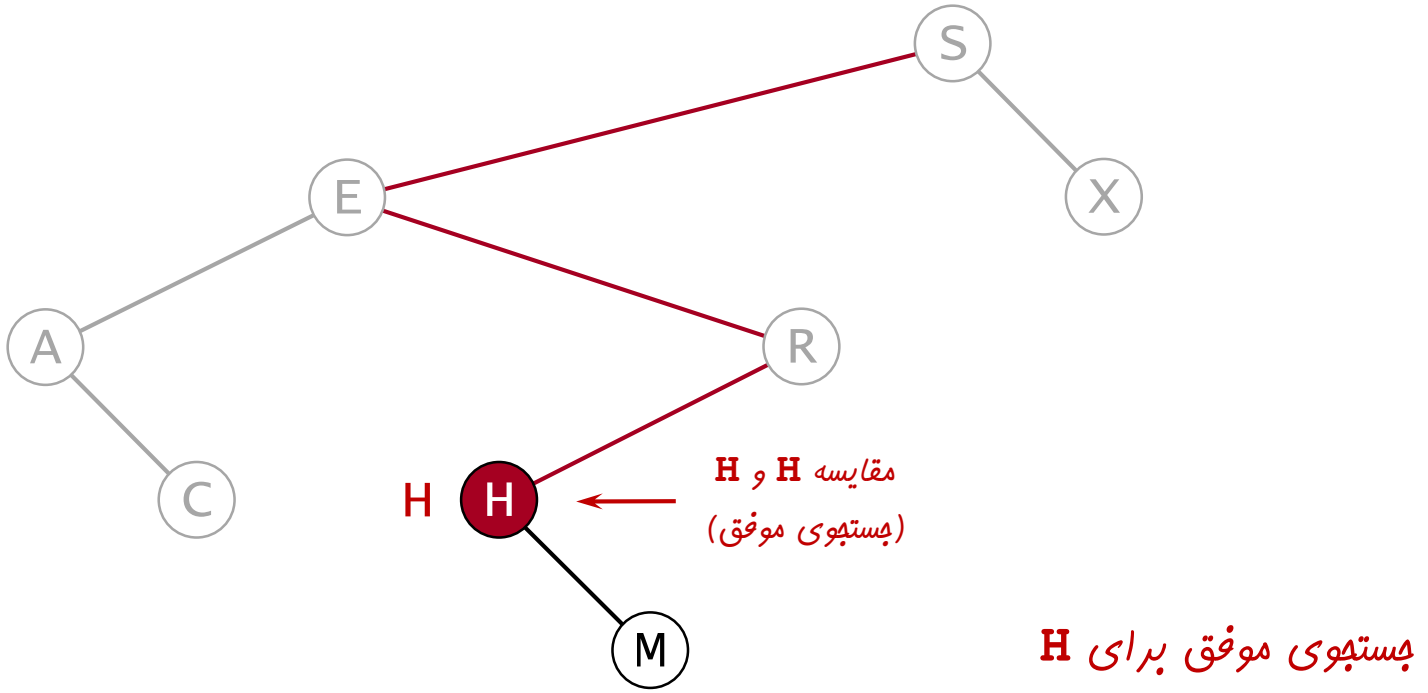
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی موفق برای H

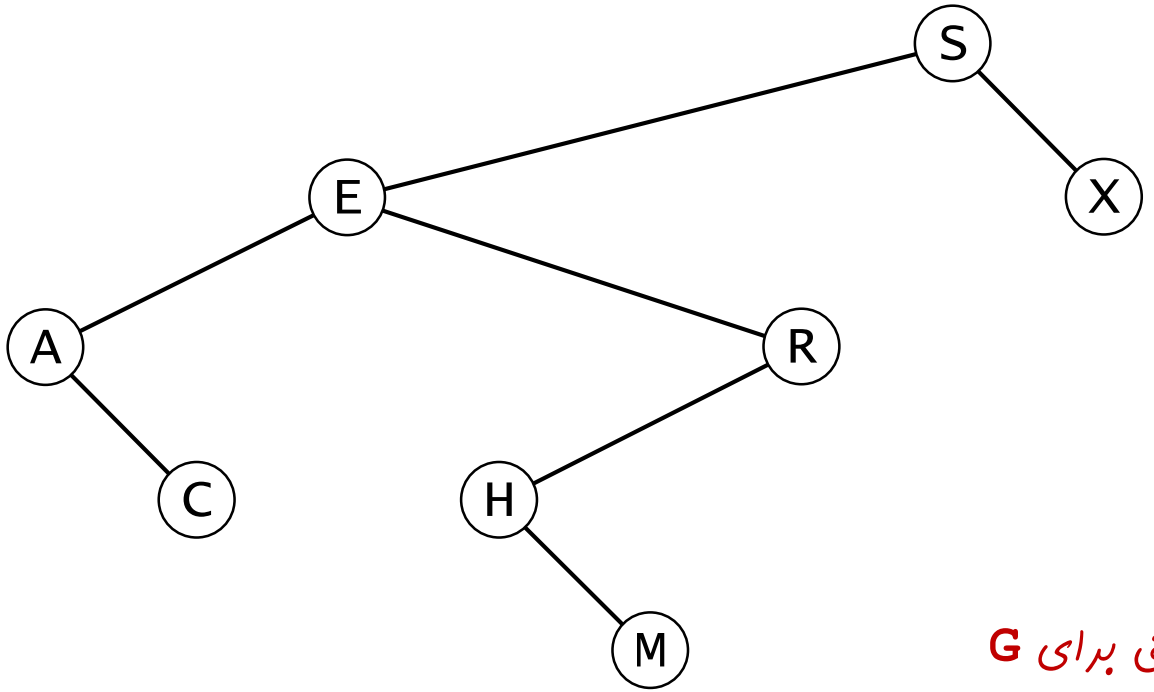
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

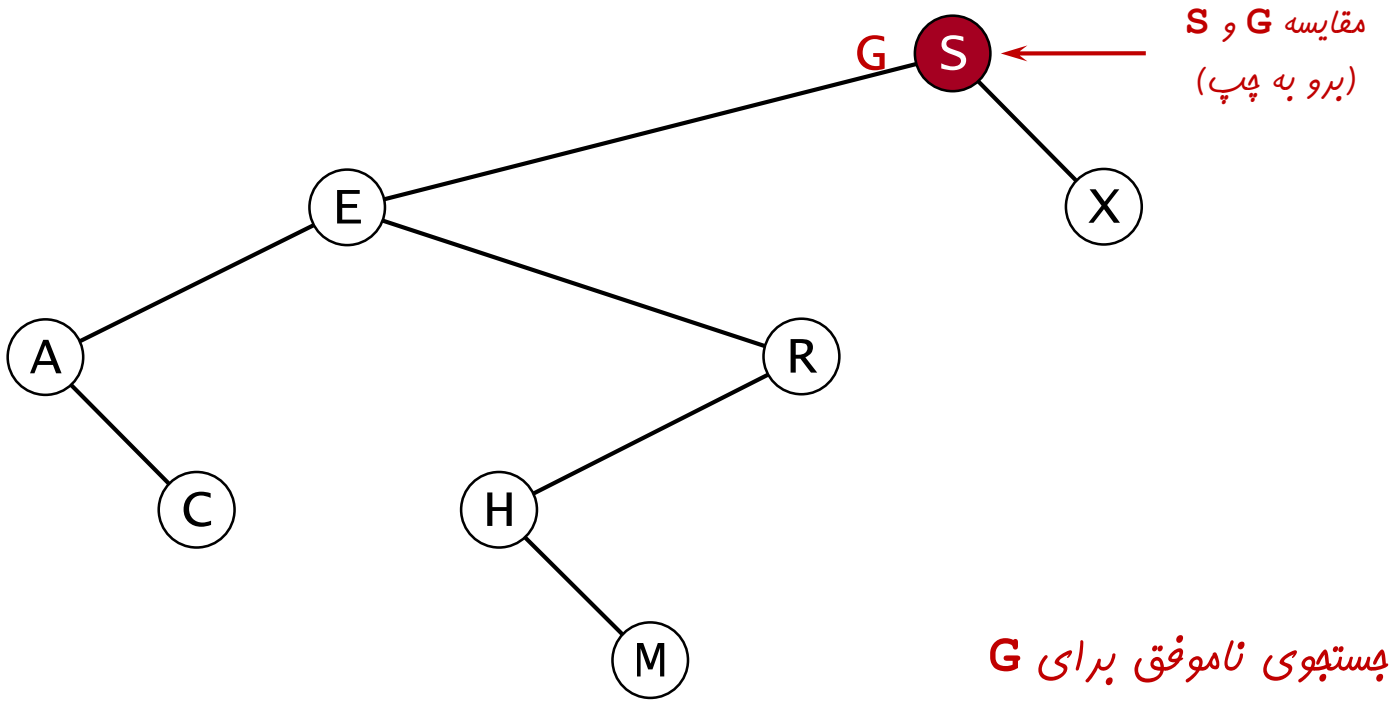
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی ناموفق برای G

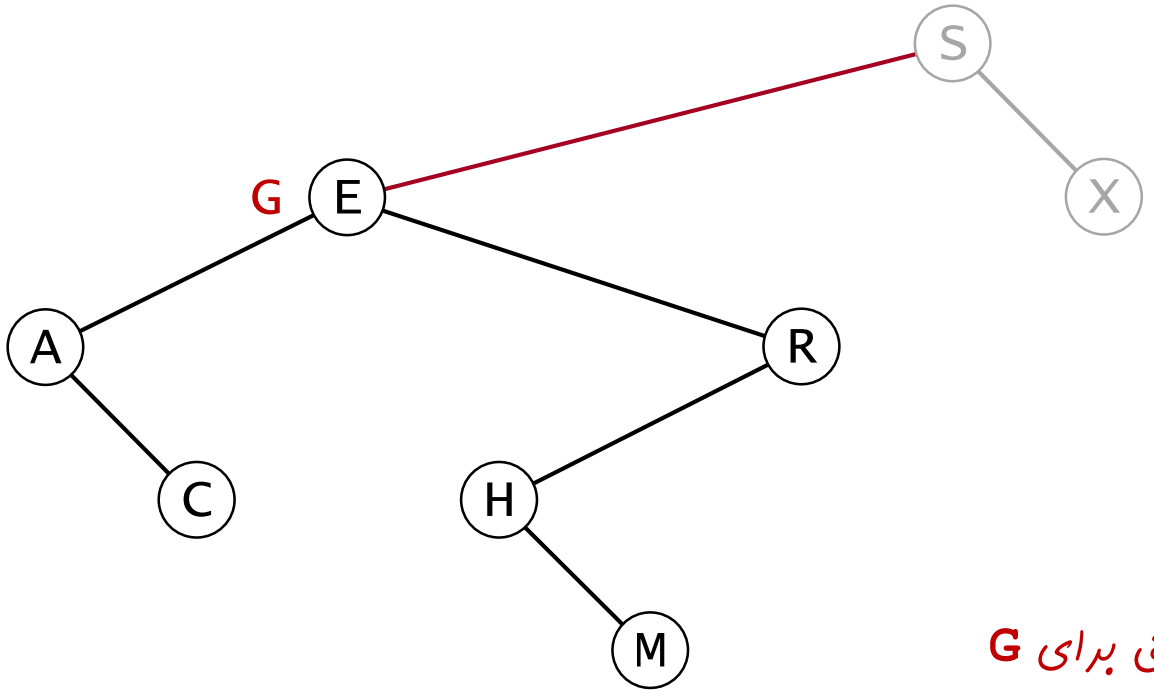
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

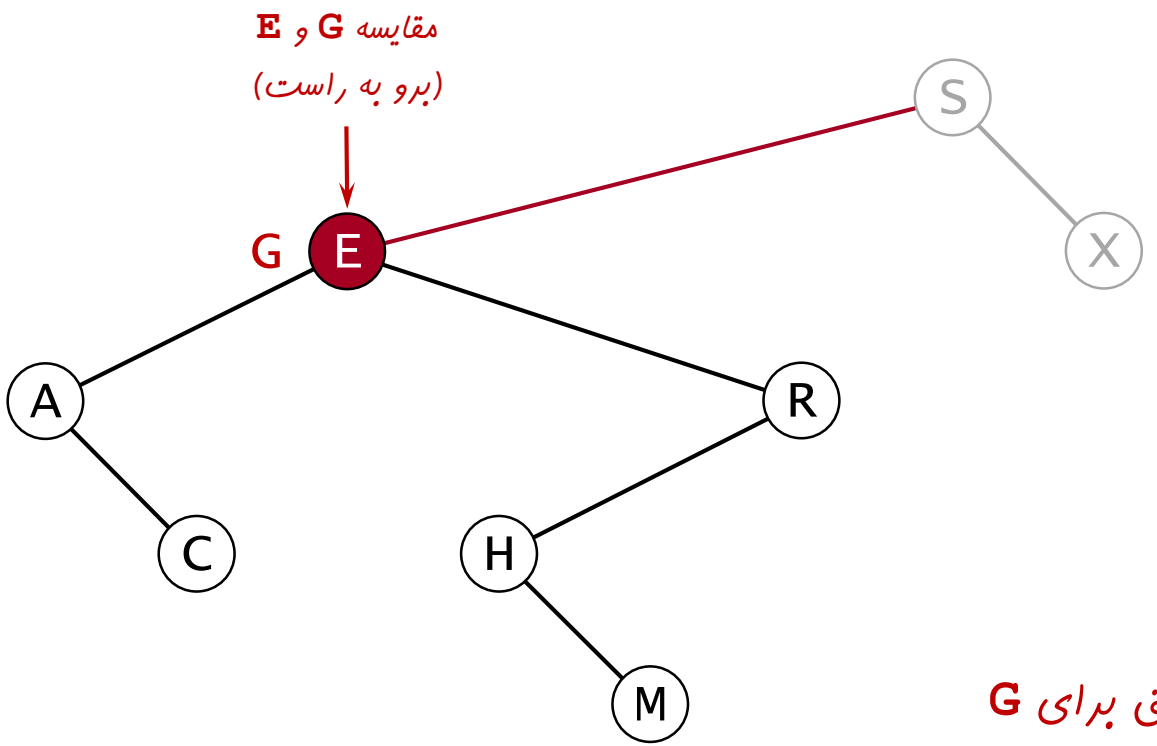
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی ناموفق برای G

درخت جستجوی دودویی: جستجو

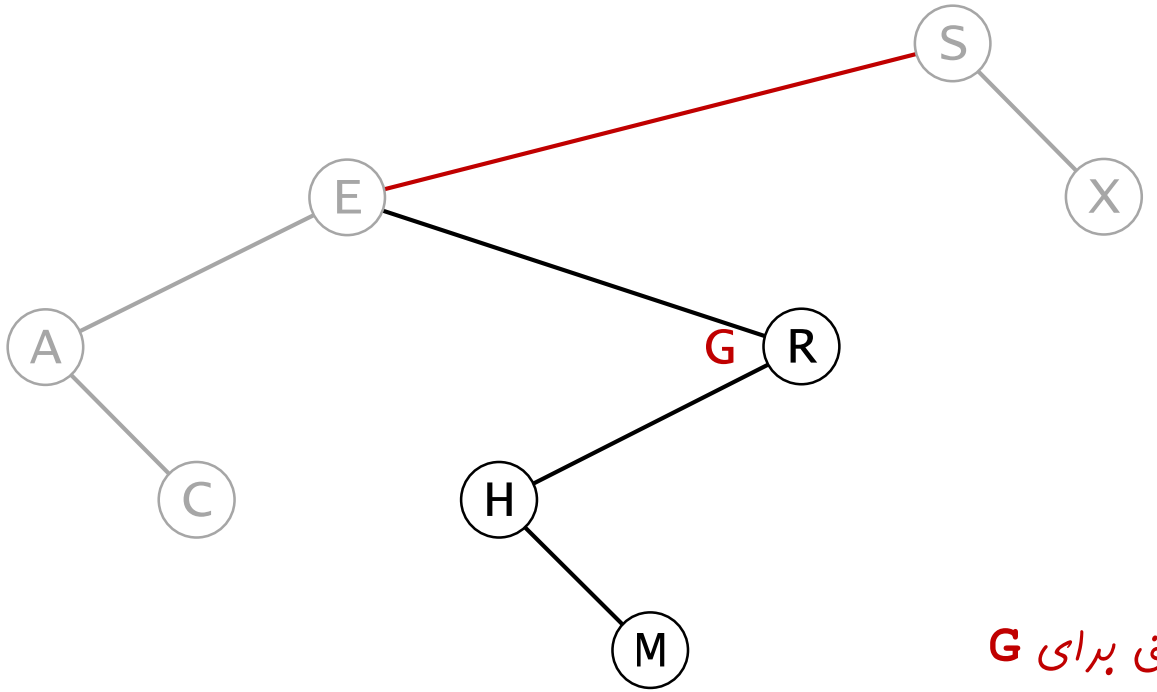
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی ناموفق برای G

درخت جستجوی دودویی: جستجو

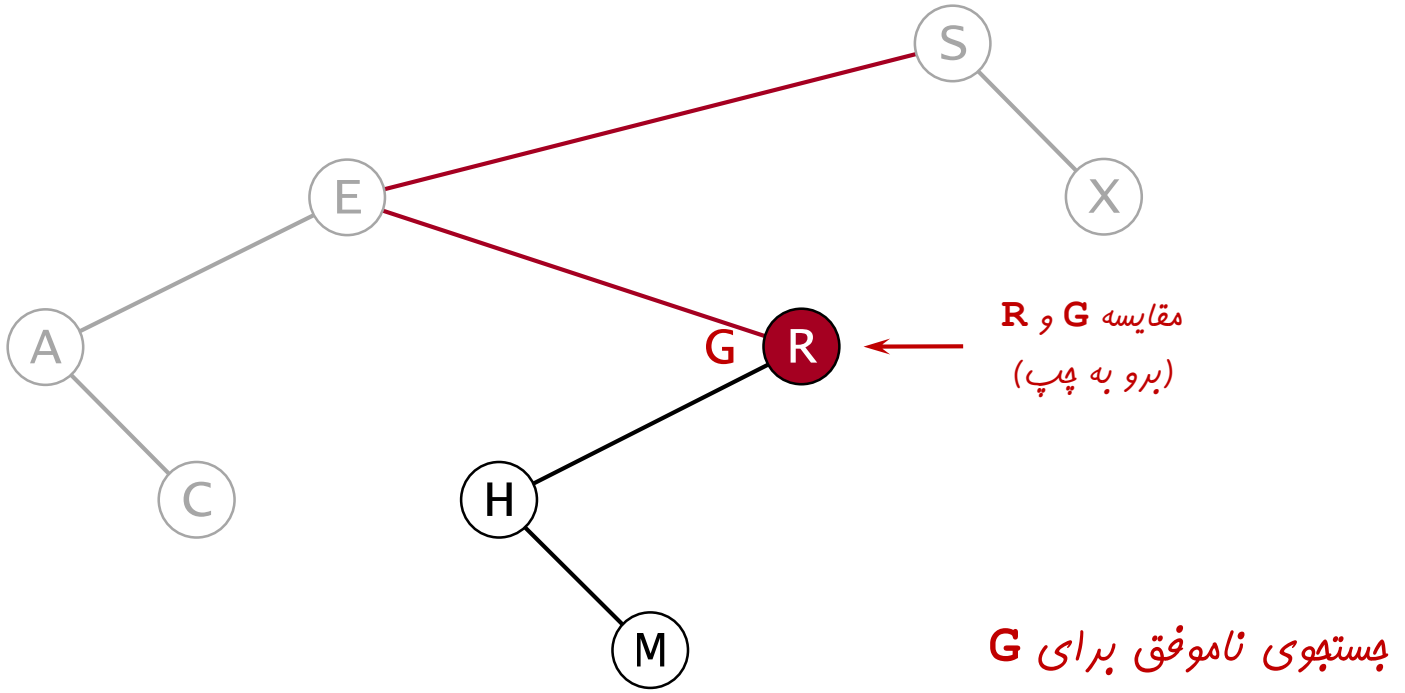
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی ناموفق برای G

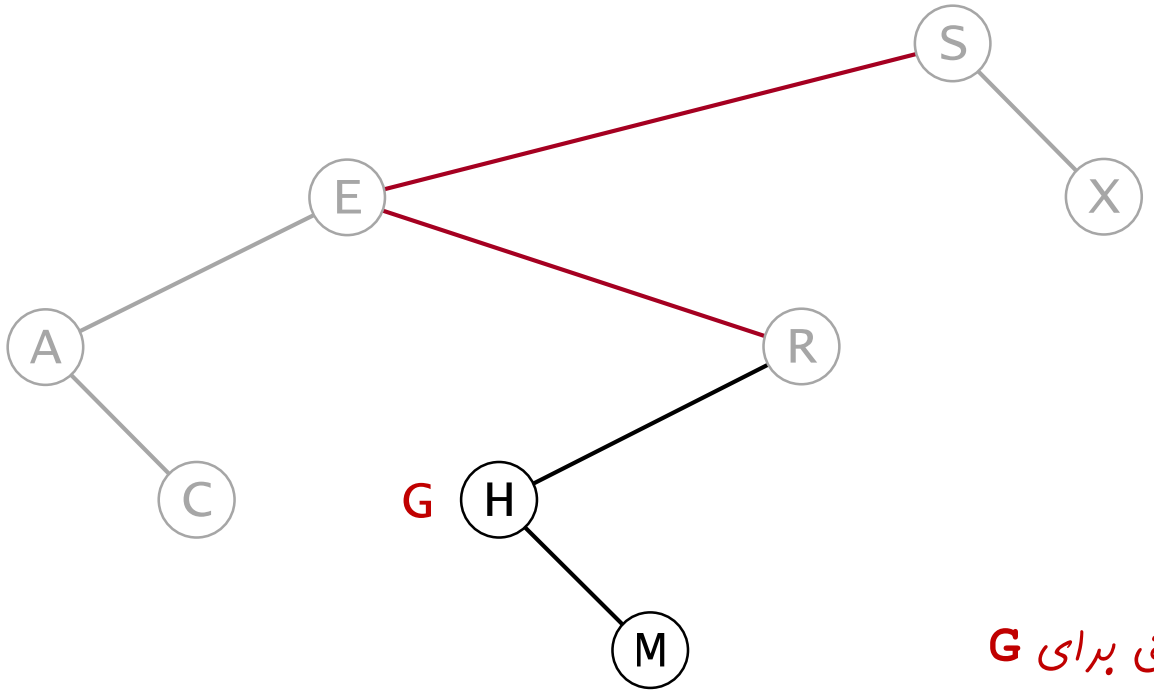
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

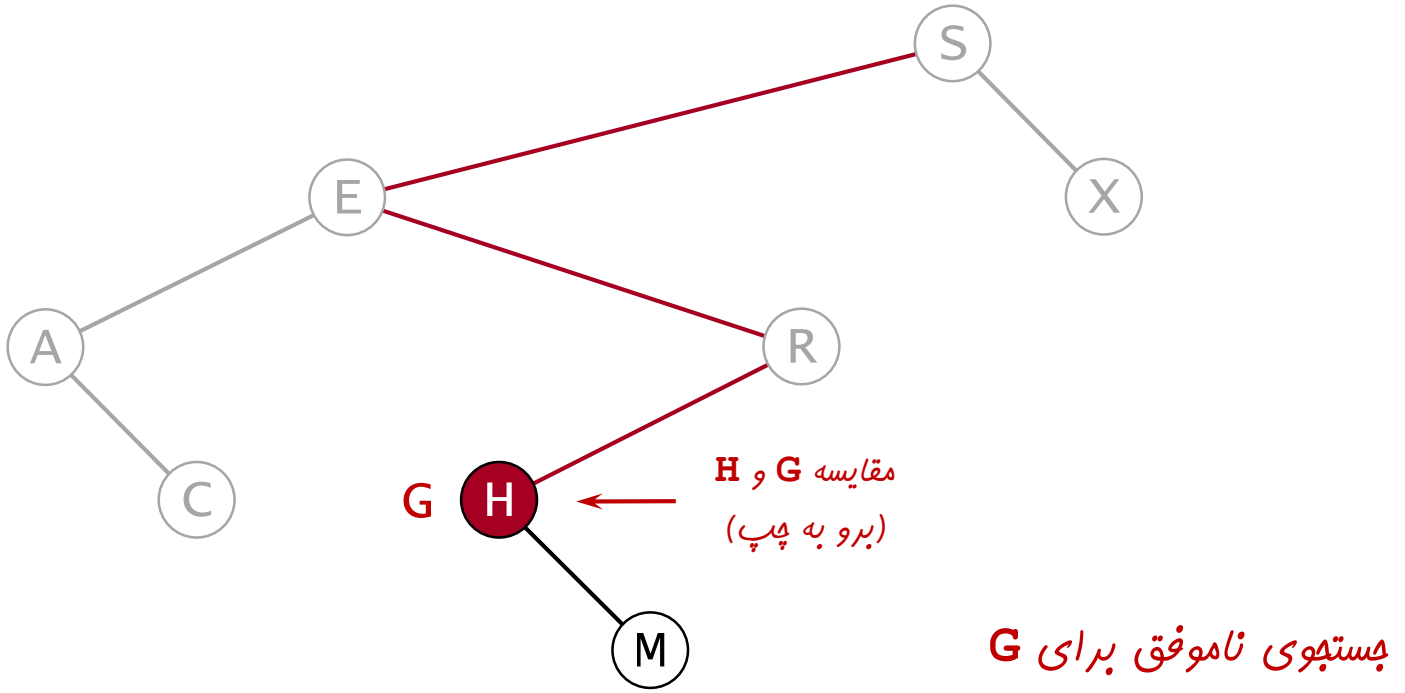
- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجوی ناموفق برای G

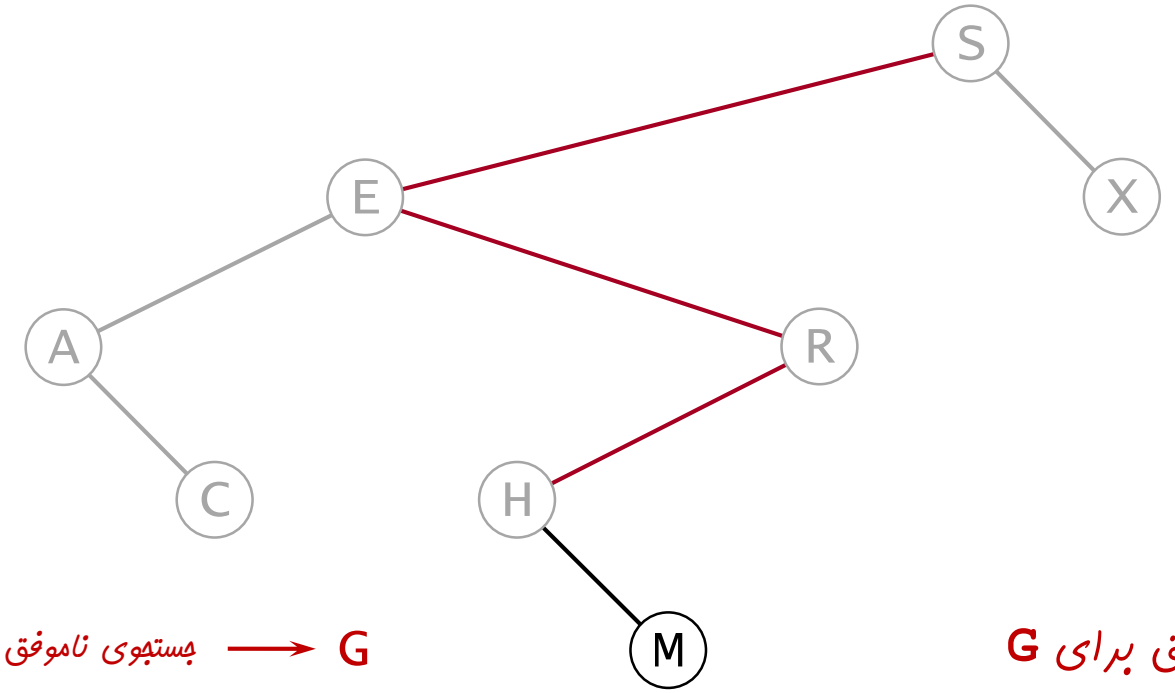
درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درخت جستجوی دودویی: جستجو

- جستجو. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



جستجو: پیاده‌سازی در جاوا

- جستجو. بازگرداندن مقدار متناظر با یک کلید.

```
public Value get(Key key)
{
   Node x = root;
   while (x != null)
   {
      int cmp = key.compareTo(x.key);
      if      (cmp  < 0) x = x.left;
      else if (cmp  > 0) x = x.right;
      else if (cmp == 0) return x.val;
   }
   return null;
}
```

- هزینه. تعداد مقایسه‌ها برابر است با عمق گره به علاوه‌ی یک.

جستجو: پیاده‌سازی در جاوا (بازگشتی)

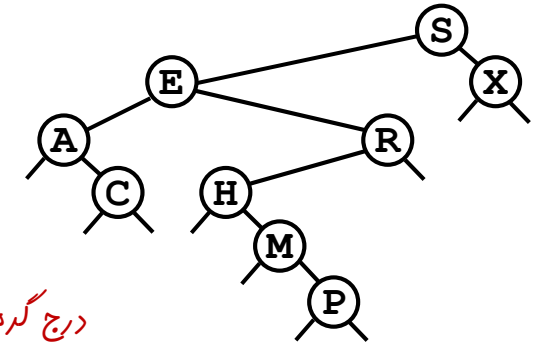
- جستجو. بازگرداندن مقدار متناظر با یک کلید.

```
public Value get(Key key) {
   return get(root, key);
}

public Value get(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return get(x.left,  key);
   else if (cmp  > 0) return get(x.right, key);
   else if (cmp == 0) return x.val;
}
```

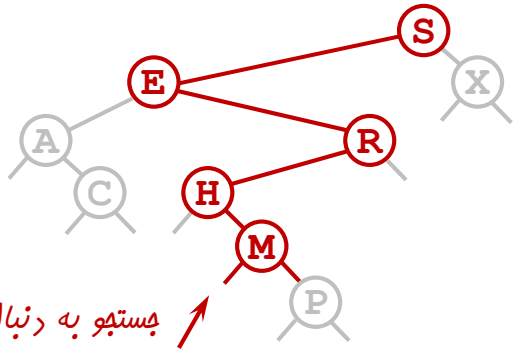
- هزینه. تعداد مقایسه‌ها برابر است با عمق گره به علاوه‌ی یک.

درج در BST

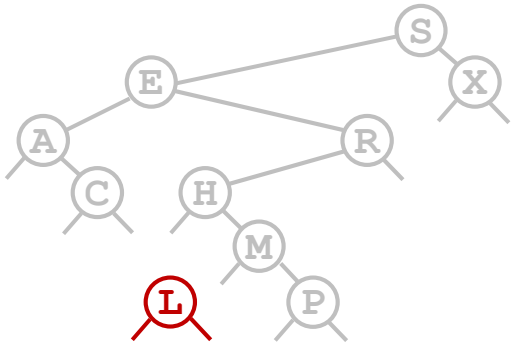
- درج. افزودن یک زوج کلید-مقدار به درخت.
  - ابتدا کلید را جستجو کن:
    - اگر کلید یافت شد، تغییر مقدار متناظر.
    - اگر کلید یافت نشد، افزودن گره جدید.



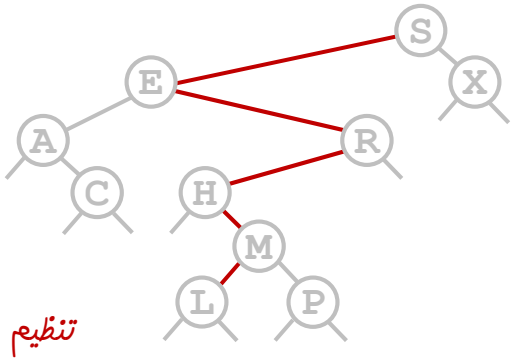
درج گره L



جستجو به دنبال L در این
پیوند پوچ به پایان می‌رسد



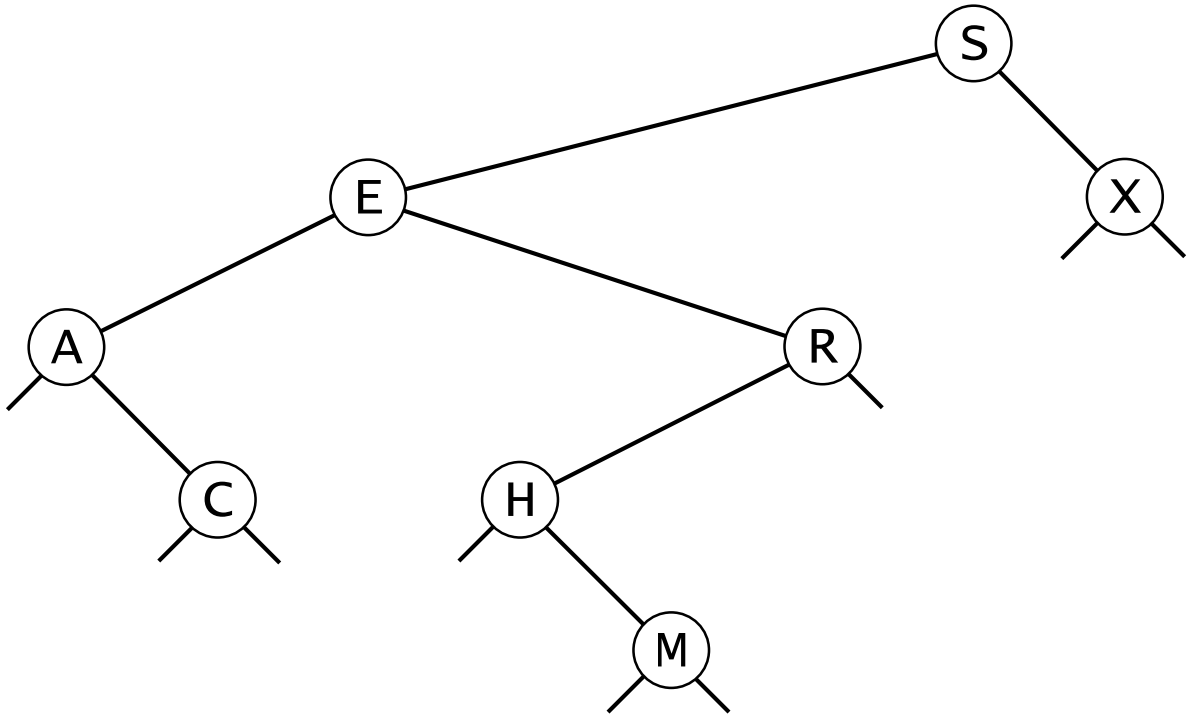
ایجاد گره جدید



تنظیم پیوندها
در مسیر جستجو

درخت جستجوی دودویی: درج

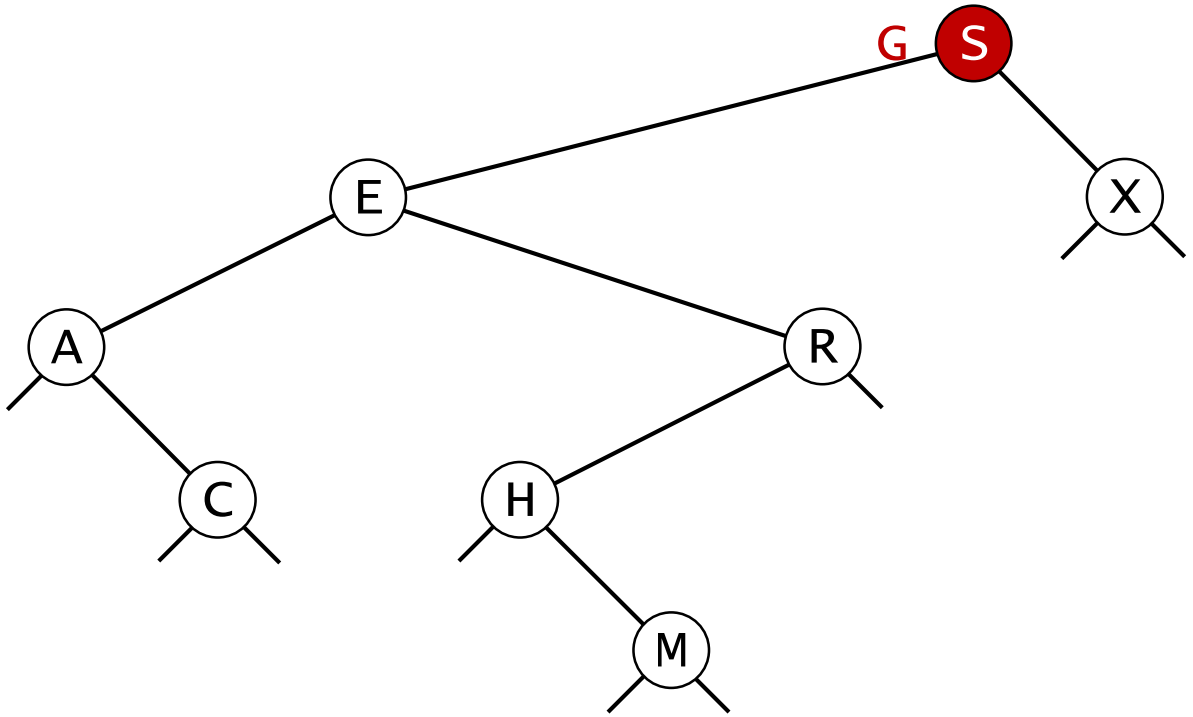
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

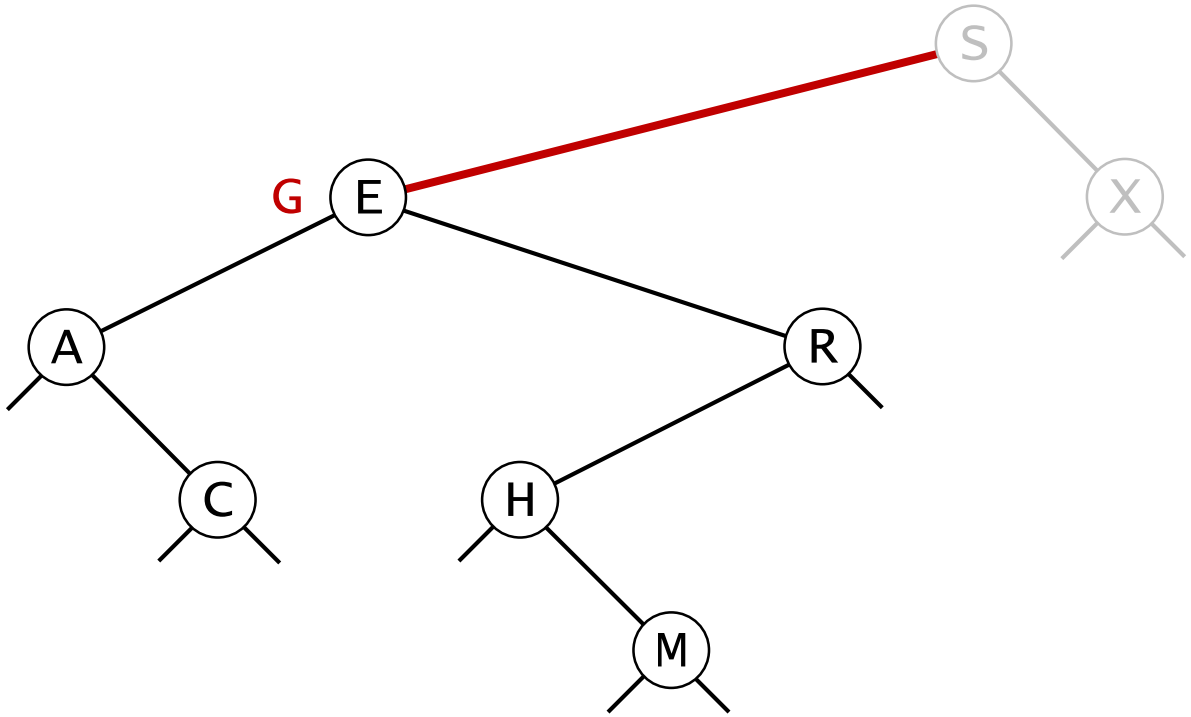
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

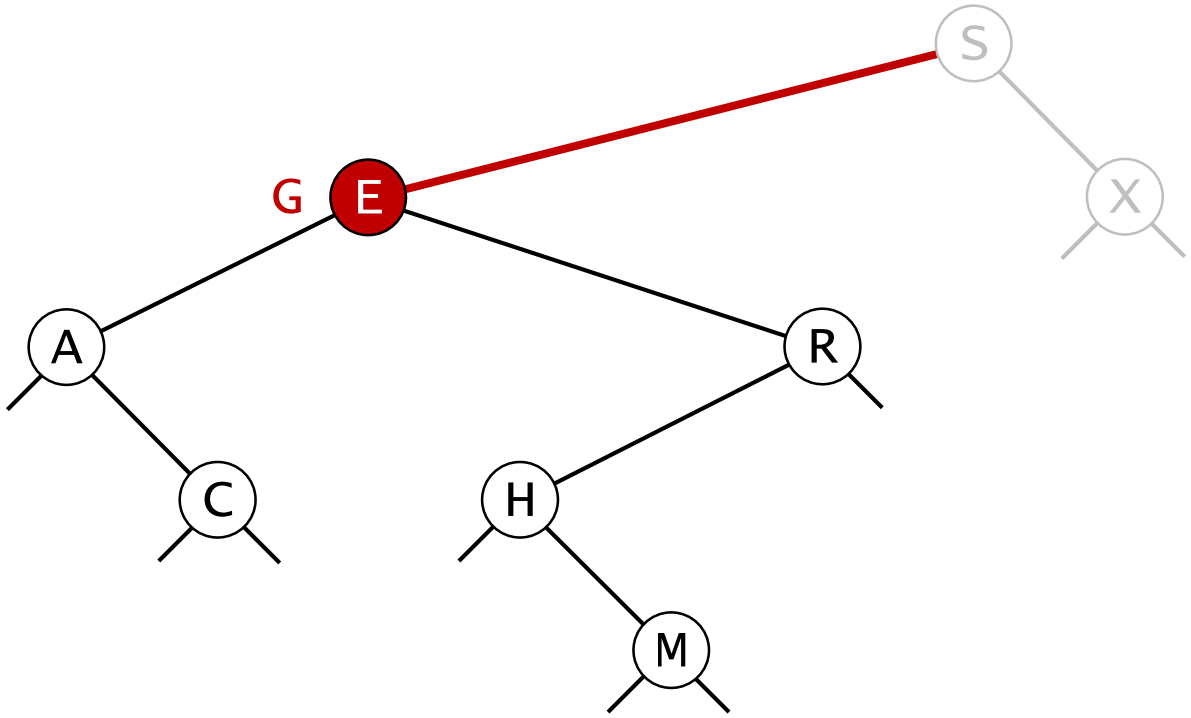
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

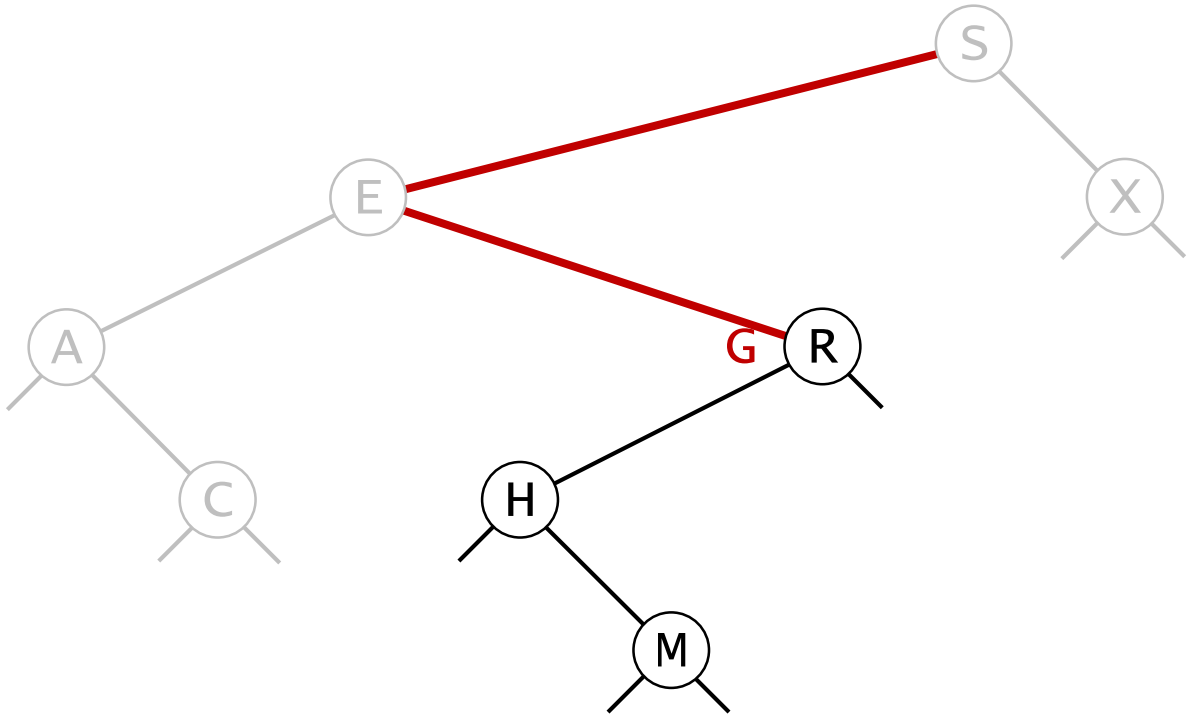
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

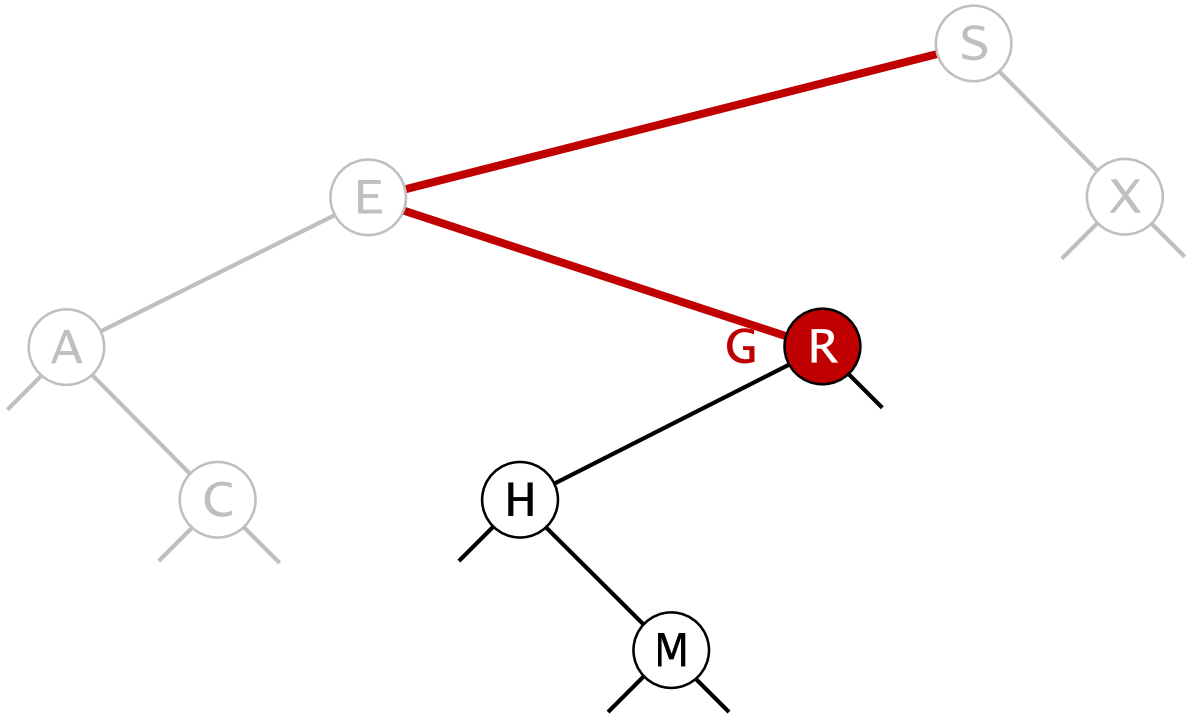
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

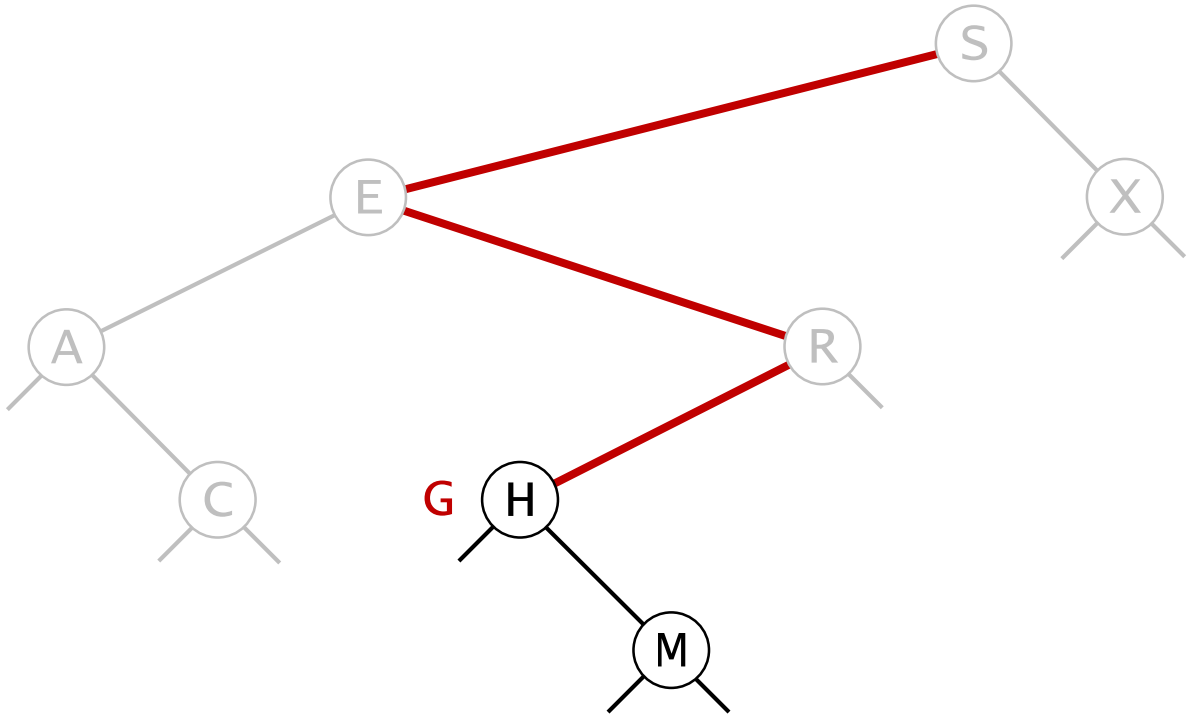
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

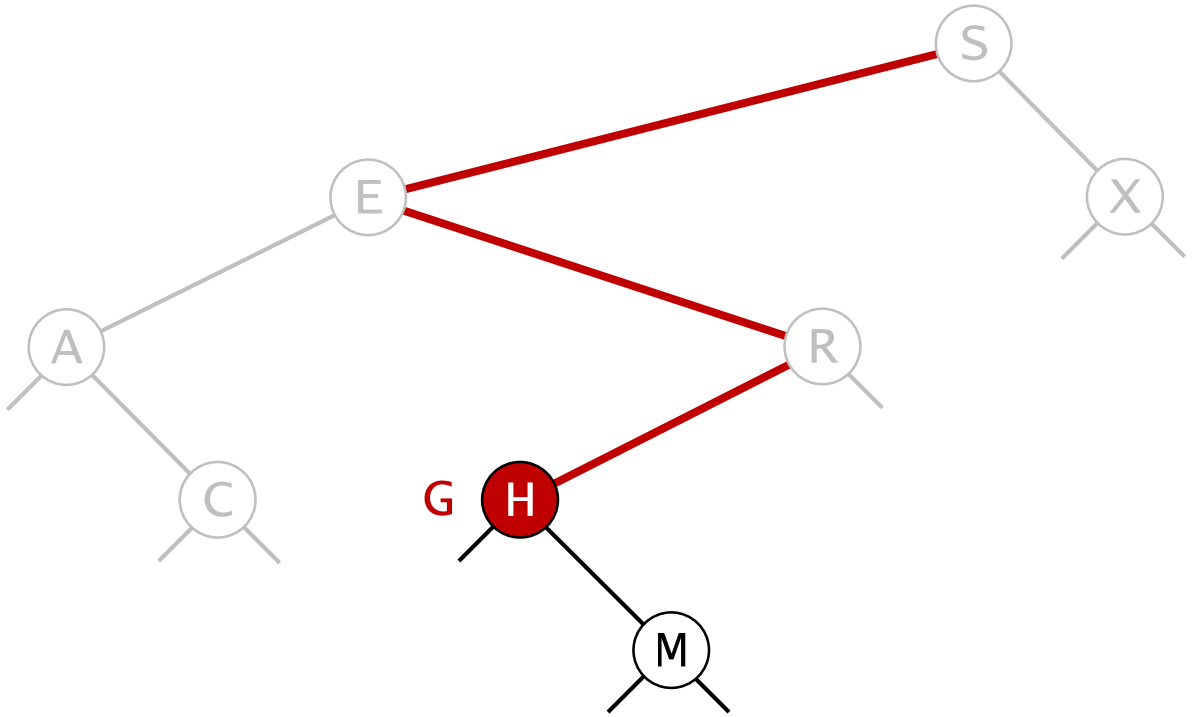
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

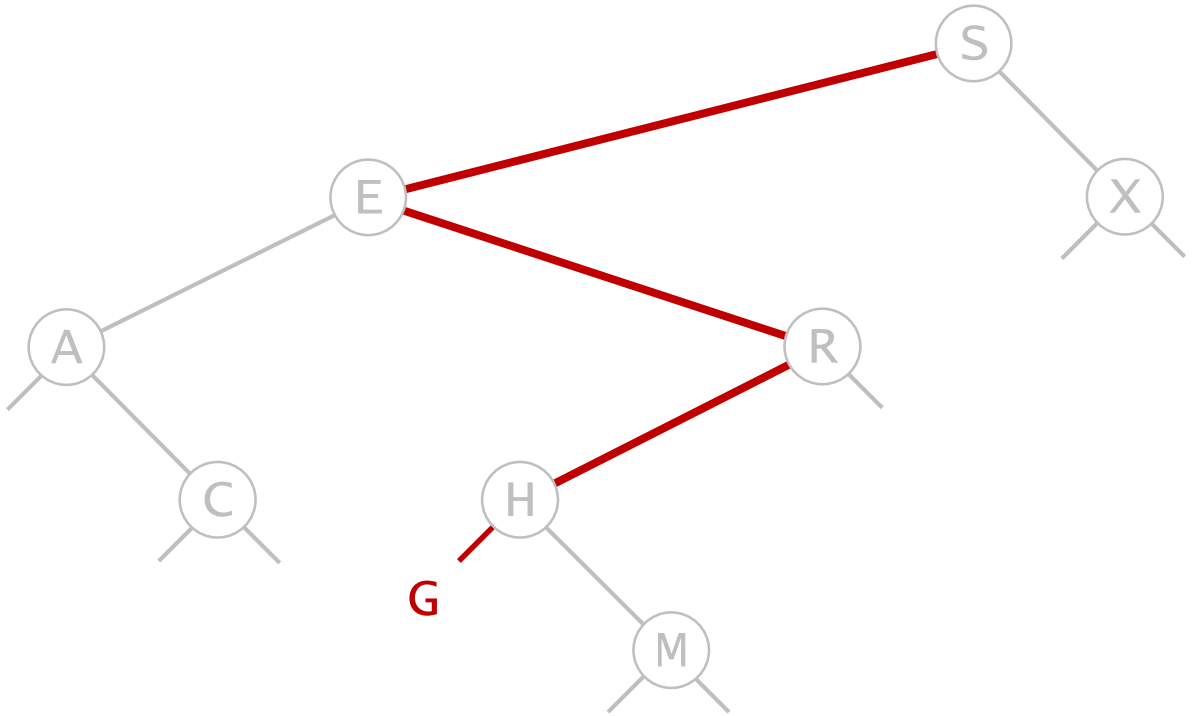
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

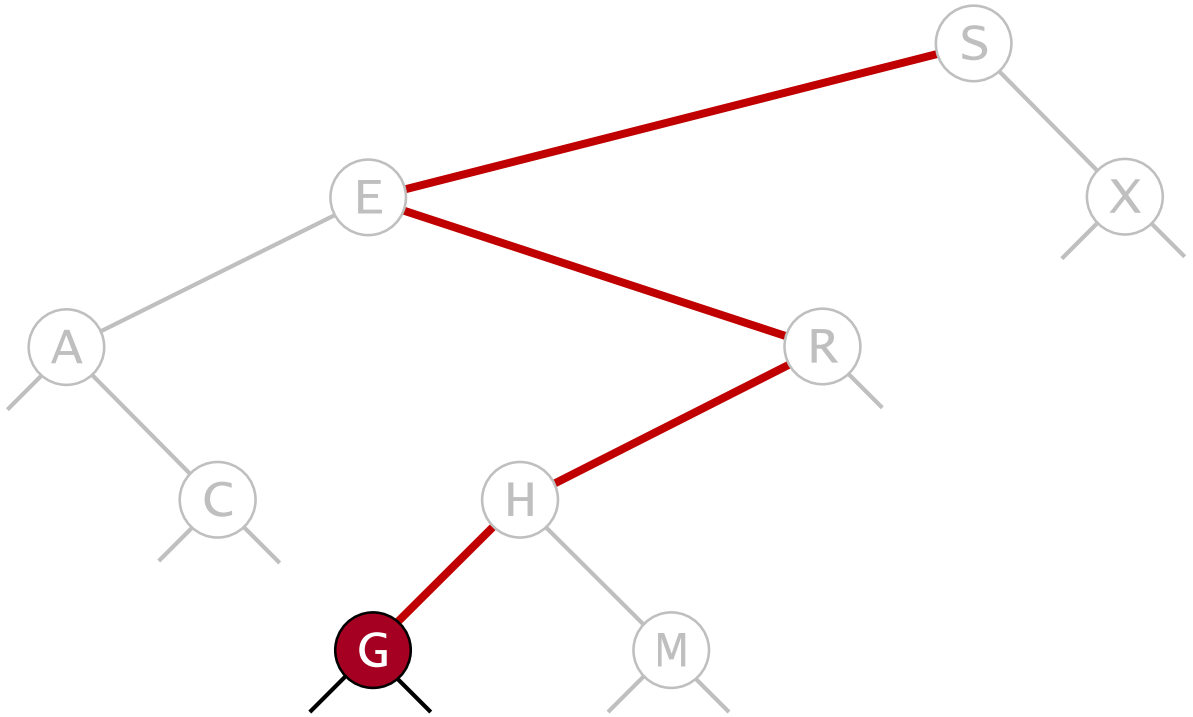
- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درخت جستجوی دودویی: درج

- درج. اگر کوچک‌تر، برو به چپ؛ اگر بزرگ‌تر، برو به راست؛ اگر مساوی، تمام.



درج G

درج: پیاده‌سازی در جاوا

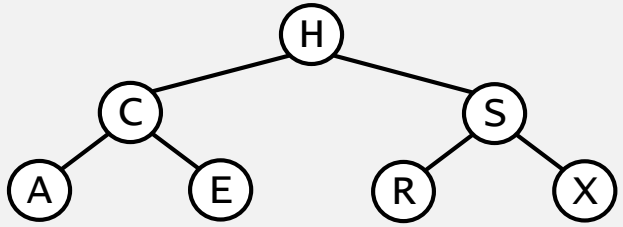
- درج.

```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val = val;
   return x;
}
```

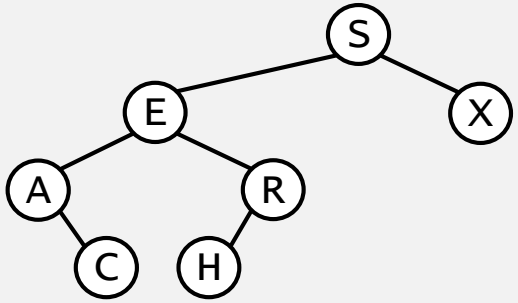
- هزینه. تعداد مقایسه‌ها برابر است با عمق گره به علاوه‌ی یک.

تحلیل زمان اجرای عمل جستجو

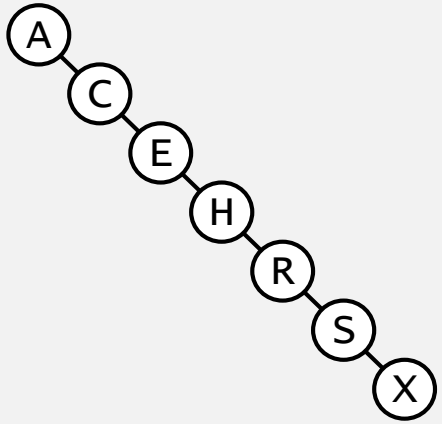
بهترین حالت



حالت متداول



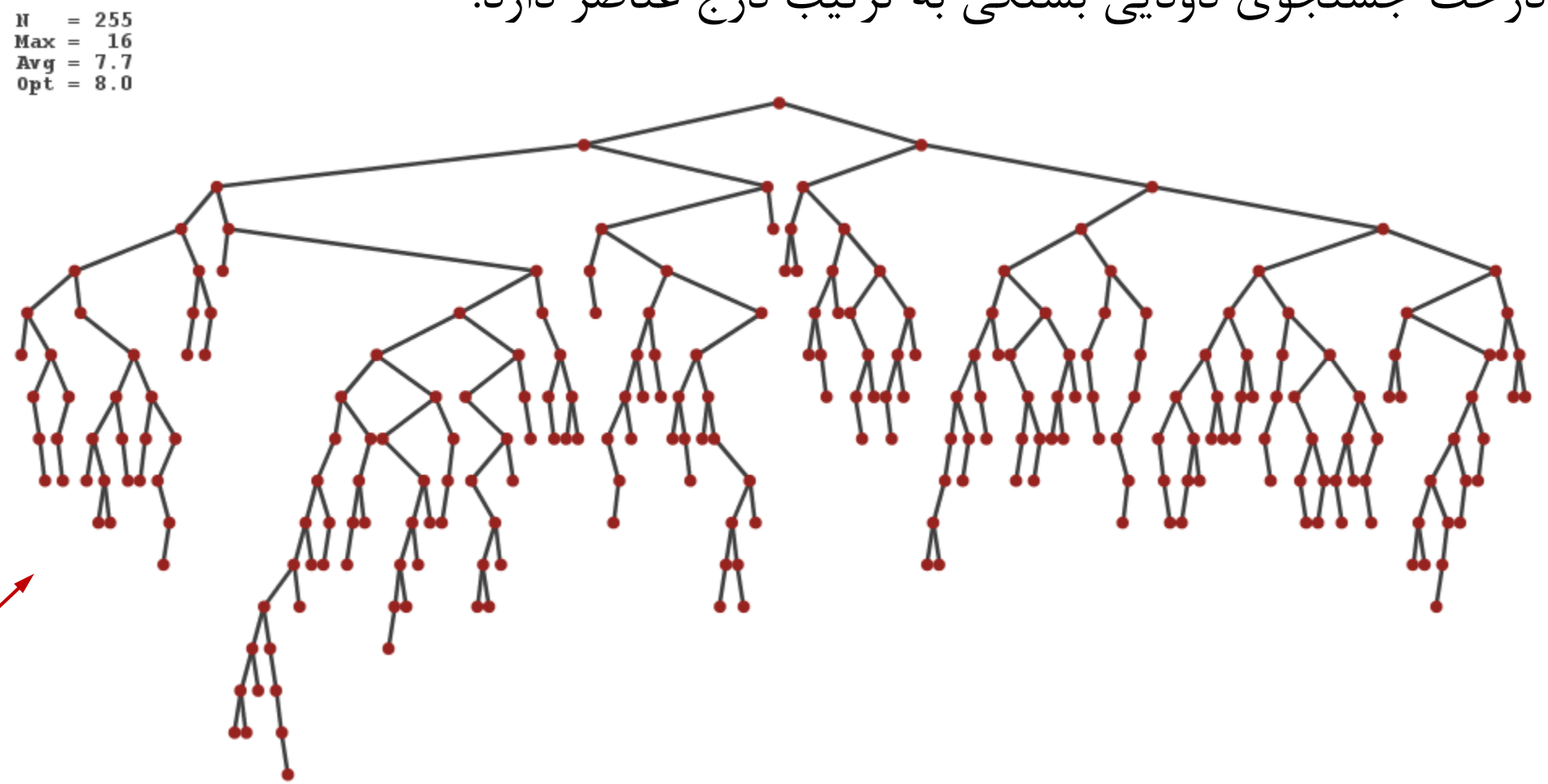
بدترین حالت



- تعداد مقایسه‌های لازم برای جستجوی یک کلید برابر است با: عمق کلید + ۱
- بنابراین، زمان اجرای جستجو در یک درخت با ارتفاع h برابر است با: $O(h)$
- ارتفاع درخت جستجوی دودویی.
  - حداقل: $\lfloor \lg n \rfloor$
  - حداکثر: $n - 1$

شکل درخت جستجوی دودویی

- شکل درخت جستجوی دودیی بستگی به ترتیب درج عناصر دارد.



درج ۲۵۵ کلید به ترتیب تصادفی: اجرای نمایشی

ارتفاع درخت جستجوی دودویی

- گزاره. اگر N کلید متمایز به ترتیب تصادفی در یک درخت دودویی درج شوند، آنگاه تعداد مورد انتظار مقایسه‌ها به منظور جستجو یا درج یک عنصر در درخت حاصل برابر است با:

$$2 \ln N \cong 1.39 \lg N$$

- اثبات. هزینه‌ی N عمل درج

$$T(N) = \frac{1}{N} \sum_{k=1}^{N} [T(k-1) + T(N-k) + (N-1)]$$

$$T(N) \cong 2(N+1) \ln N \sim 2N \ln N$$

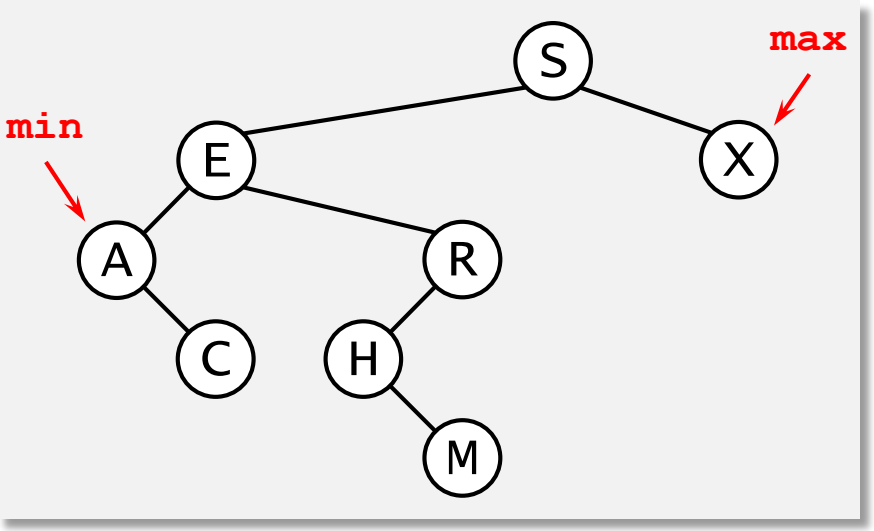
- نتیجه. هزینه‌ی یک عمل درج

$$T(N)/N \sim 2 \ln N$$

عملیات مربوط به ترتیب

یافتن کلیدهای کمینه و بیشینه

- کمینه. کوچک‌ترین کلید در درخت؛ بیشینه. بزرگ‌ترین کلید در درخت.

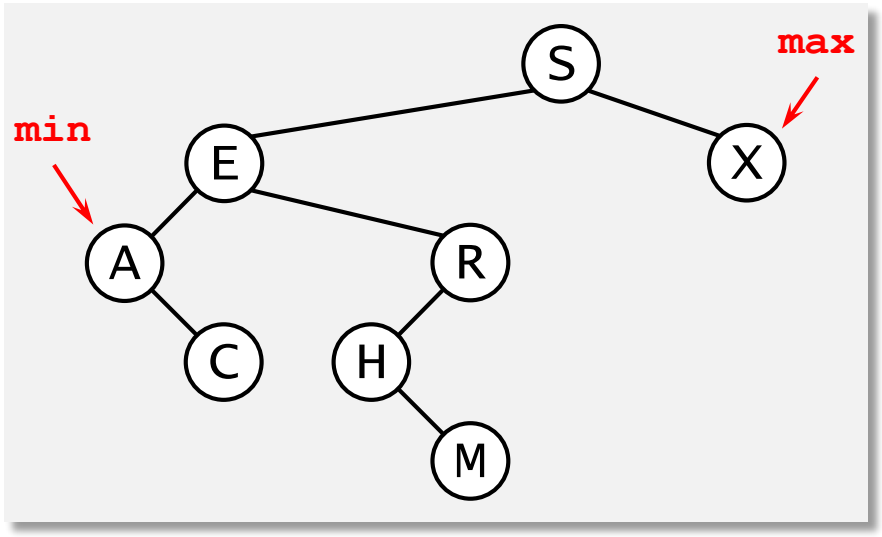



```
public Key min() {
   if (isEmpty()) return null;
   return min(root).key;
}


private Node min(Node x) {
   if (x.left  == null) return x;
   else return min(x.left);
}
```

- هزینه. متناسب با ارتفاع درخت.

یافتن کلیدهای کمینه و بیشینه

- کمینه. کوچک‌ترین کلید در درخت؛ بیشینه. بزرگ‌ترین کلید در درخت.

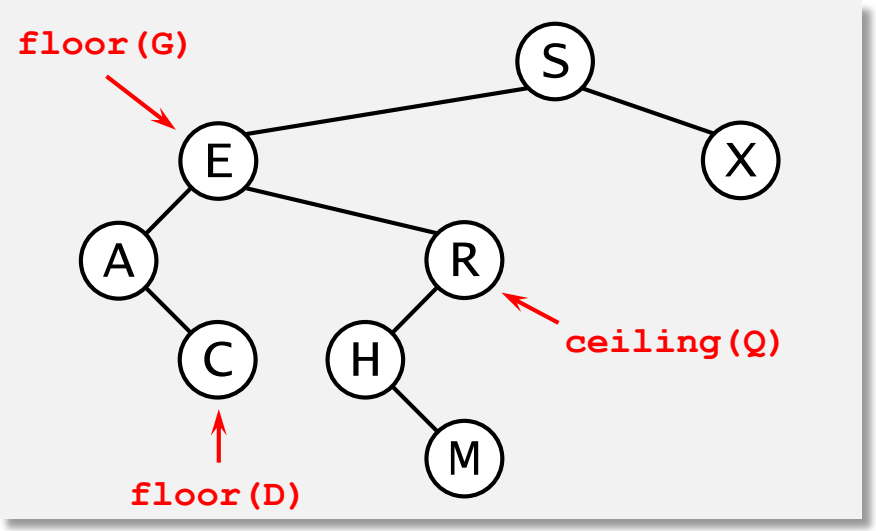



```
public Key max() {
   if (isEmpty()) return null;
   return max(root).key;
}


private Node max(Node x) {
   if (x.right == null) return x;
   else return max(x.right);
}
```

- هزینه. متناسب با ارتفاع درخت.

یافتن کف و سقف یک کلید

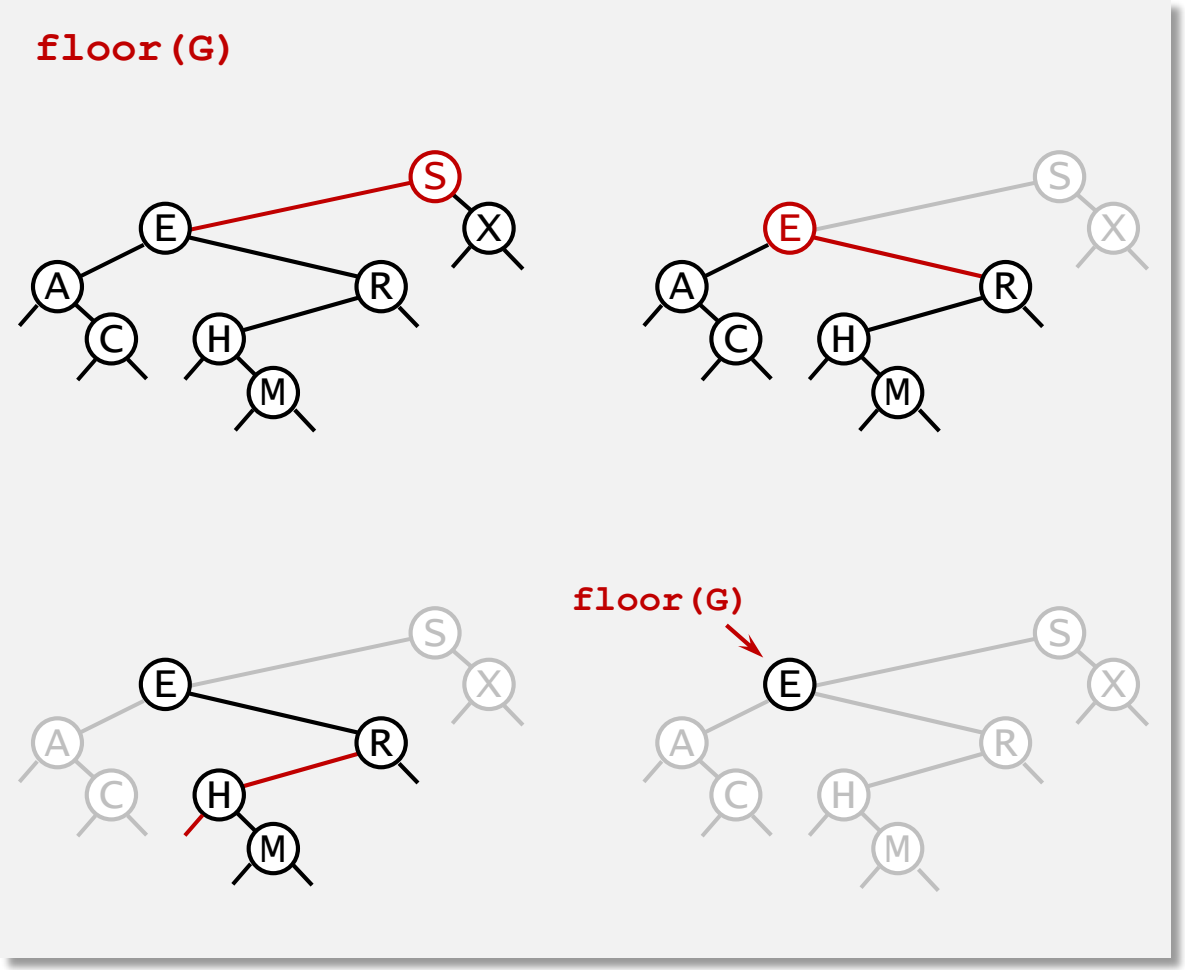
- کف. بزرگ‌ترین کلید در درخت که کوچک‌تر یا مساوی یک کلید داده شده است.
- سقف. کوچک‌ترین کلید در درخت که بزرگ‌تر یا مساوی یک کلید داده شده است.



- س. چگونه می‌توان سقف و کف یک عنصر داده شده را محاسبه نمود؟

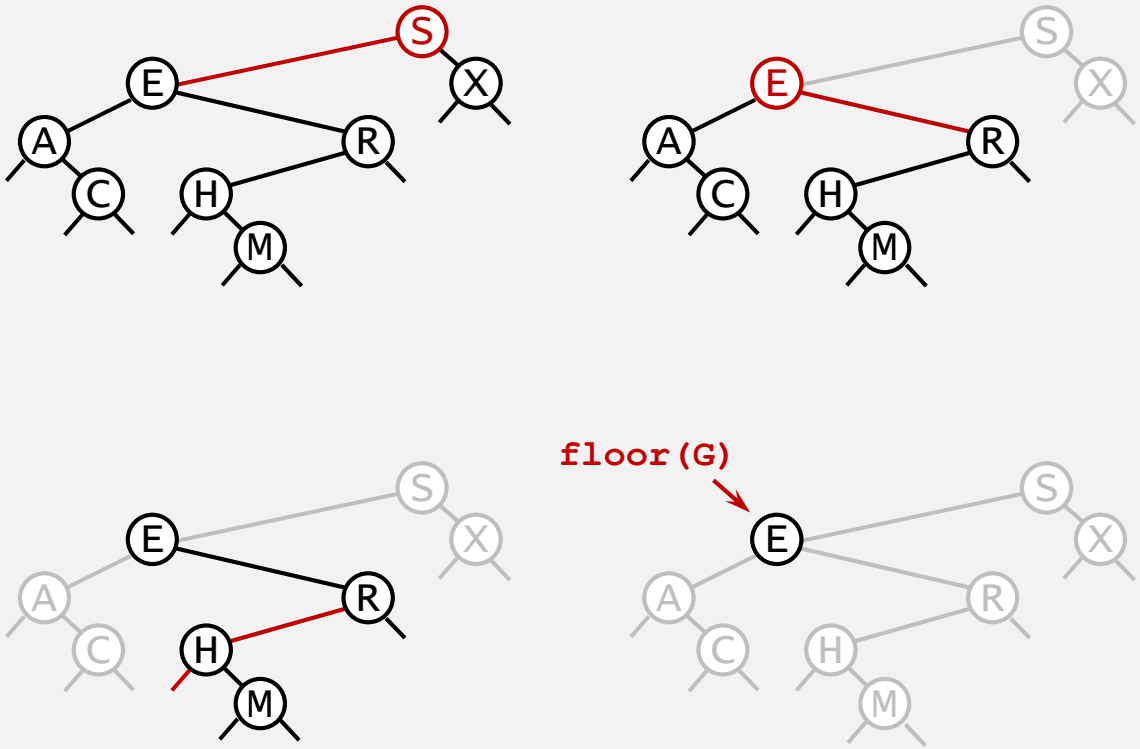
محاسبه کف یک کلید

- حالت ۱. [k برابر با کلید ریشه است]
  - کف k برابر با k است.
- حالت ۲. [k کوچک‌تر از کلید ریشه است]
  - کف k در زیردرخت چپ قرار دارد.
- حالت ۳. [k بزرگ‌تر از کلید ریشه است]
  - کف k در زیردرخت راست قرار دارد، اگر:
    - کلیدی کوچک‌تر یا مساوی k در زیردرخت راست موجود باشد؛
    - در غیر این صورت، کف k برابر با کلید ریشه است.



محاسبه‌ی کف یک کلید

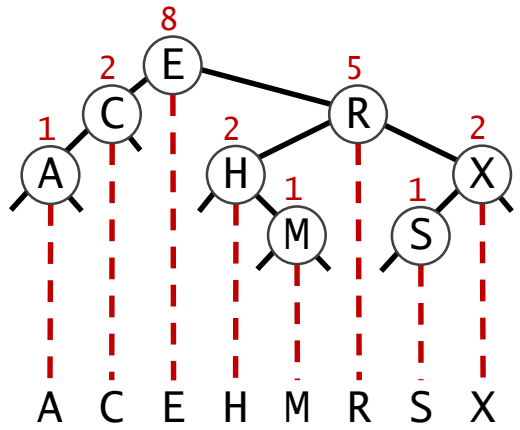
floor(G)

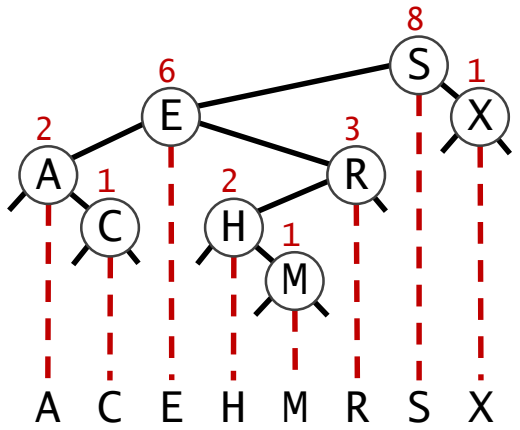



```
public Key floor(Key key)
{
   Node x = floor(root, key);
   if (x == null) return null;
   return x.key;
}

private Node floor(Node x, Key key)
{
   if (x == null) return null;
   int cmp = key.compareTo(x.key);

   if (cmp == 0) return x;

   if (cmp  < 0) return floor(x.left, key);

   Node t = floor(x.right, key);
   if (t != null) return t;
   else           return x;

}
```

تعداد گره‌ها در یک زیردرخت

- در هر گره، تعداد گره‌های موجود در زیردرخت آن گره را ذخیره می‌کنیم.
- برای پیاده‌سازی تابع **`size()`**، عدد مربوط به گره ریشه را برمی‌گردانیم.





- توضیح. این کار باعث می‌شود بتوانیم توابع **`rank()`** و **`select()`** را به طور کارا پیاده‌سازی کنیم.

تعداد گره‌ها در یک زیردرخت

- ساختار یک گره.

```
private class Node
{
   private Key key;
   private Value val;
   private Node left;
   private Node right;
   private int N;
}
```

- تابع `size()`.

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

```
private int size(Node x)
{
   if (x == null) return 0;
   return x.N;
}
```

تعداد گره‌ها در یک زیردرخت

- درج.

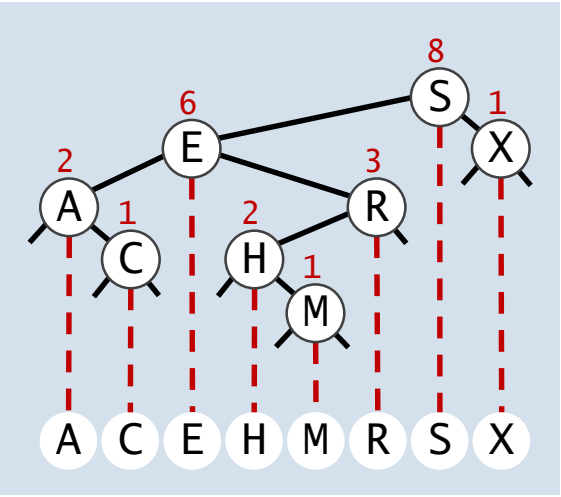
```
public void put(Key key, Value val)
{  root = put(root, key, val);  }

private Node put(Node x, Key key, Value val)
{
   if (x == null) return new Node(key, val);
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = put(x.left,  key, val);
   else if (cmp  > 0) x.right = put(x.right, key, val);
   else if (cmp == 0) x.val   = val;

   x.N = 1 + size(x.left) + size(x.right);

   return x;
}
```

- هزینه. تعداد مقایسه‌ها برابر است با عمق گره به علاوه‌ی یک.

رتبه

- رتبه‌ی کلید k. تعداد کلیدهای کوچک‌تر از k.
  - یک الگوریتم بازگشتی ساده [با چهار حالت]

```
public int rank(Key key)
{  return rank(key, root);  }

private int rank(Key key, Node x)
{
   if (x == null) return 0;
   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) return rank(key, x.left);
   else if (cmp  > 0) return 1 + size(x.left) + rank(key, x.right);
   else if (cmp == 0) return size(x.left);

}
```



انتخاب

- انتخاب. یافتن کلید با رتبه‌ی k.

```
public Key select(int k)
{
   if (k < 0 || k >= size()) return null;
   Node x = select(root, k);
   return x.key;
}

private Node select(Node x, int k)
{
   if (x == null) return null;
   int t = size(x.left);
   if      (k  < t) return select(x.left, k);
   else if (k  > t) return select(x.right, k - t - 1);
   else if (k == t) return x;

}
```

پیمایش میان‌ترتیب

- پیمایش میان‌ترتیب.
  I. پیمایش میان‌ترتیب زیردرخت چپ؛
  II. اضافه کردن ریشه به صف؛
  III. پیمایش میان‌ترتیب زیردرخت راست.

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

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

- نکته. در پیمایش میان‌ترتیب یک درخت جستجوی دودویی، کلیدها به ترتیب افزایشی پیمایش می‌شوند.

درخت جستجوی دودویی: خلاصه

	جستجوی ترتیبی	جستجوی دودویی	درخت جستجوی دودویی
جستجو	N	lg N	h
درج	1	N	h
کوچکترین	N	1	h
کف و سقف	N	lg N	h
رتبه	N	lg N	h
انتخاب	N	1	h
تکرار	N lg N	N	N

h = ارتفاع BST

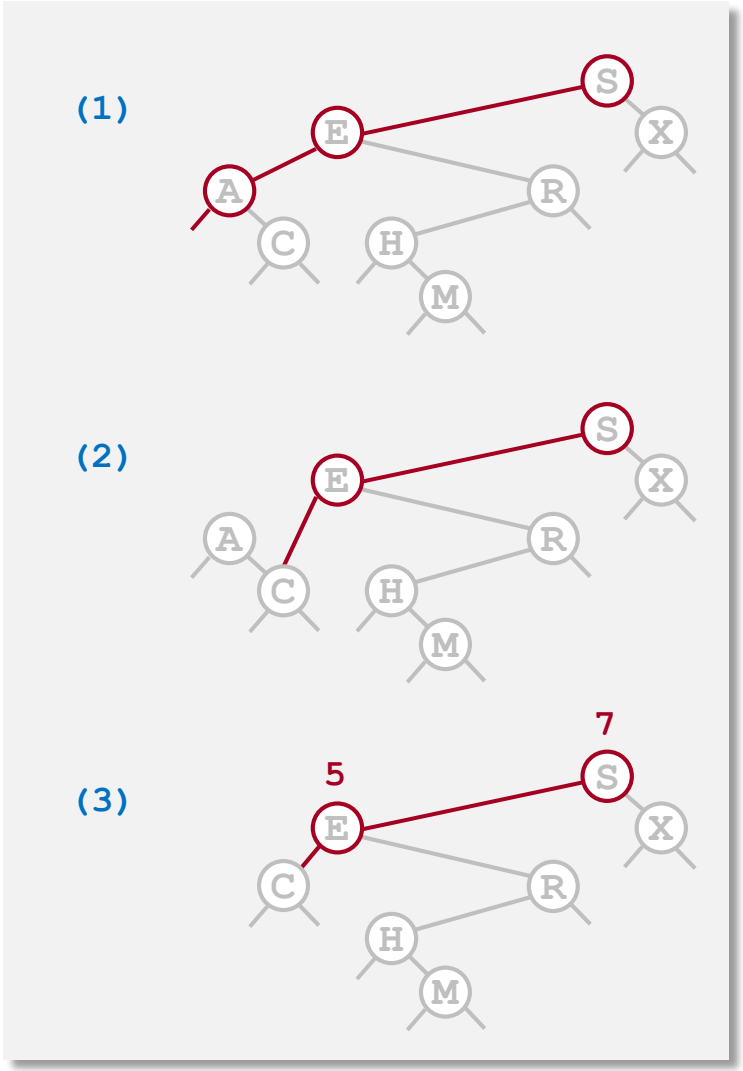
(اگر کلیدها به ترتیب تصادفی درج شوند، ارتفاع لگاریتمی است)

نرخ رشد زمان اجرای مربوط به عملیات مختلف بر روی یک جدول نماد دارای ترتیب.

حذف

حذف کوچک‌ترین کلید

- برای حذف کوچک‌ترین کلید:
  I. به سمت چپ بروید تا به یک گره با پیوند چپ پوچ برسید؛
  II. آن گره را با زیردرخت راستش جایگزین کنید؛
  III. تعداد گره‌ها در زیردرخت‌ها را اصلاح کنید.




```
public void deleteMin () {
   root = deleteMin(root);
}

private Node deleteMin(Node x) {
   if (x.left == null) return x.right;
   x.left = deleteMin(x.left);
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```

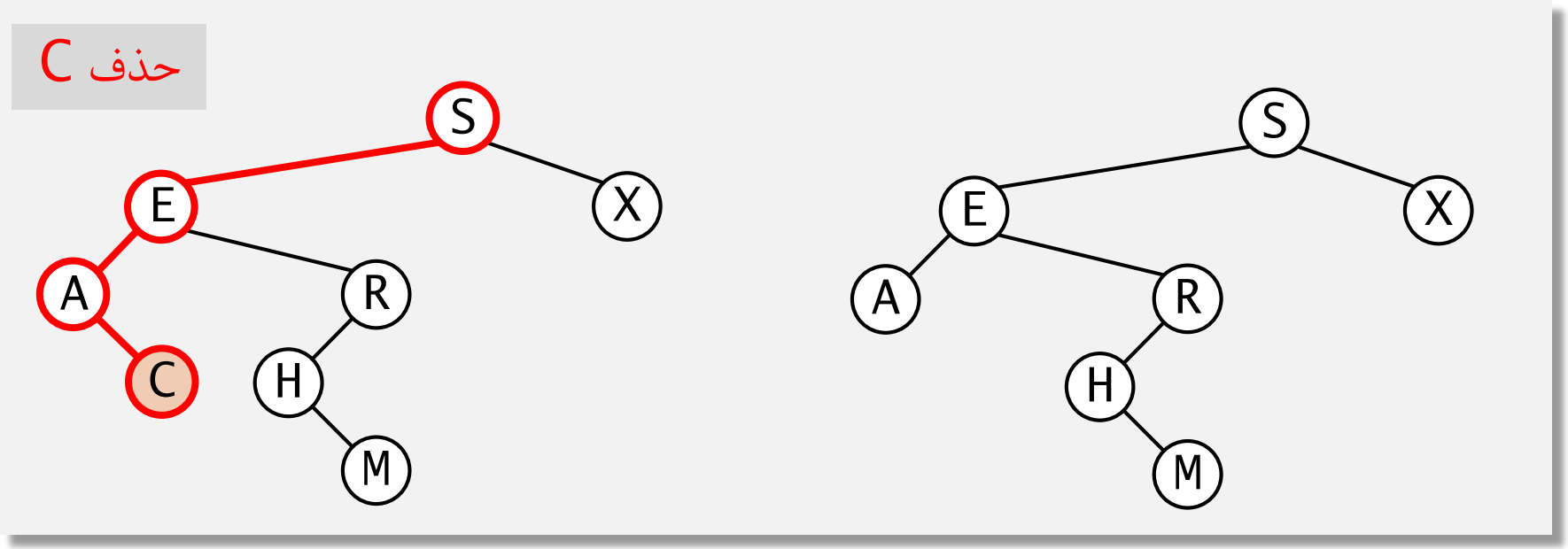
حذف (۱)

- حذف گره x از درخت T.

- در هنگام حذف گره x با یکی از سه حالت زیر مواجه می‌شویم:
  - حالت ۱) گره x فرزند ندارد؛
  - حالت ۲) گره x تنها یک فرزند دارد؛
  - حالت ۳) گره x دو فرزند دارد؛

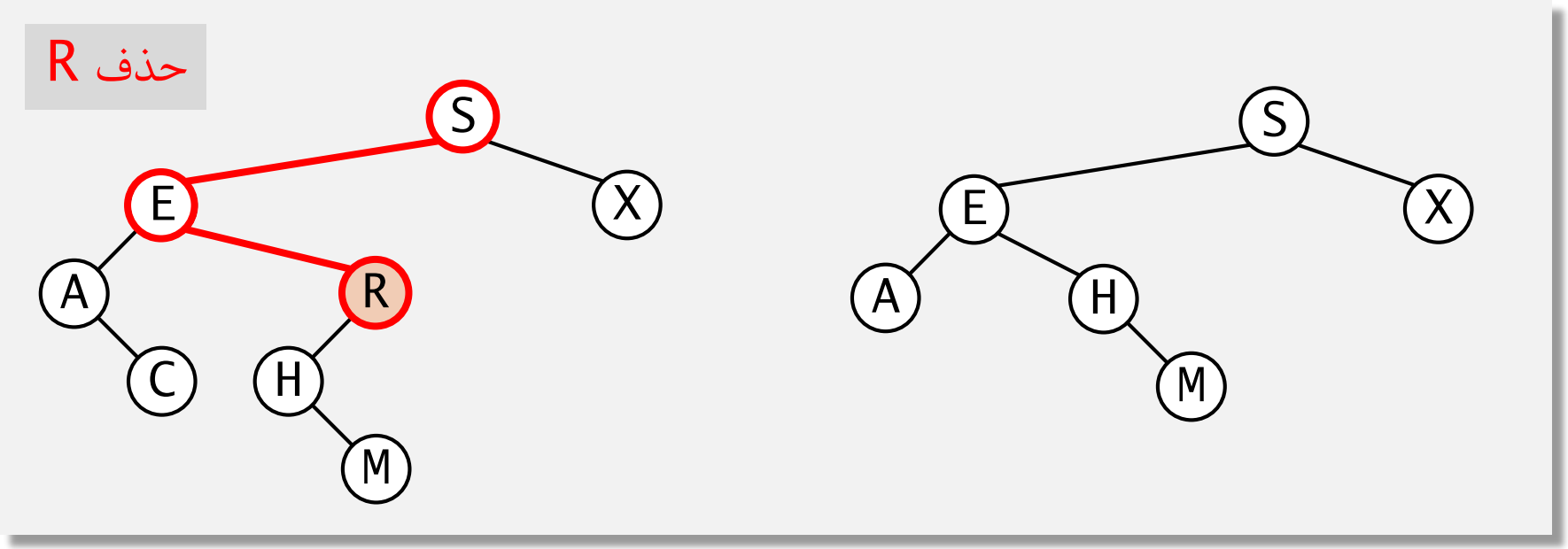
حذف (۲)

- حالت ۱) گره x فرزند ندارد.
  - گره x را با پوچ کردن پیوند پدر حذف کن.



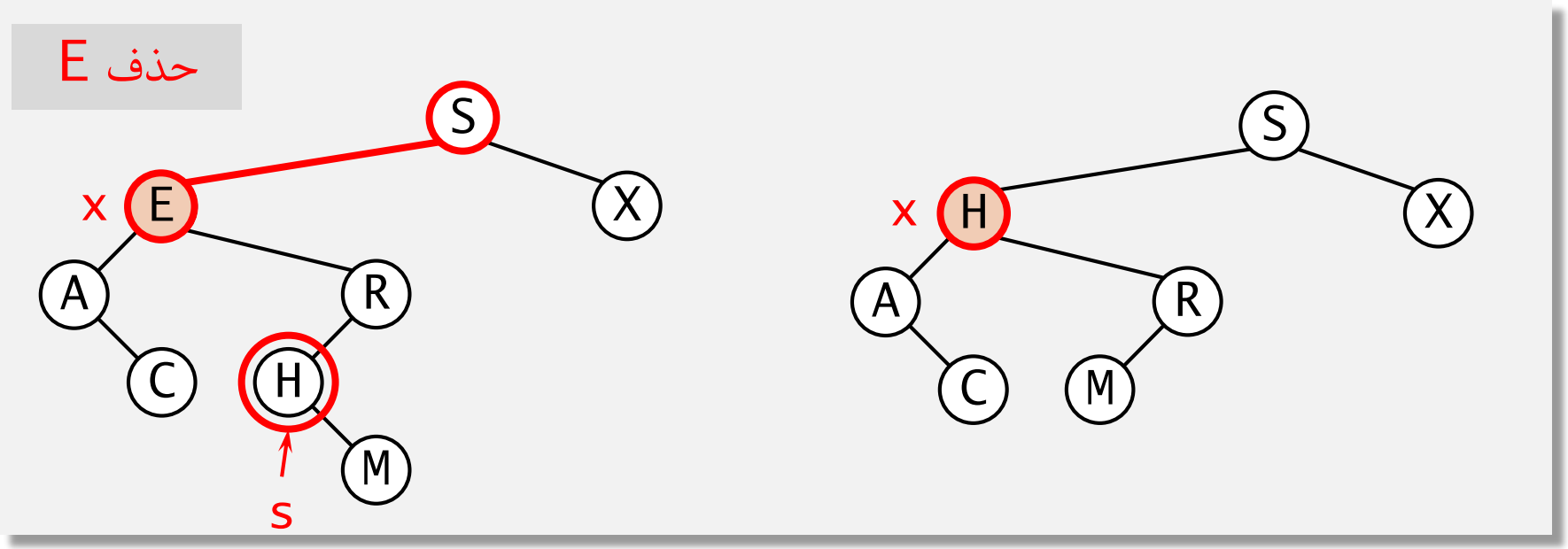
حذف (۳)

- حالت ۲) گره x **یک** فرزند دارد.
  - گره x را با جایگزین کردن آن با تنها فرزندش حذف کن.



حذف (۴)

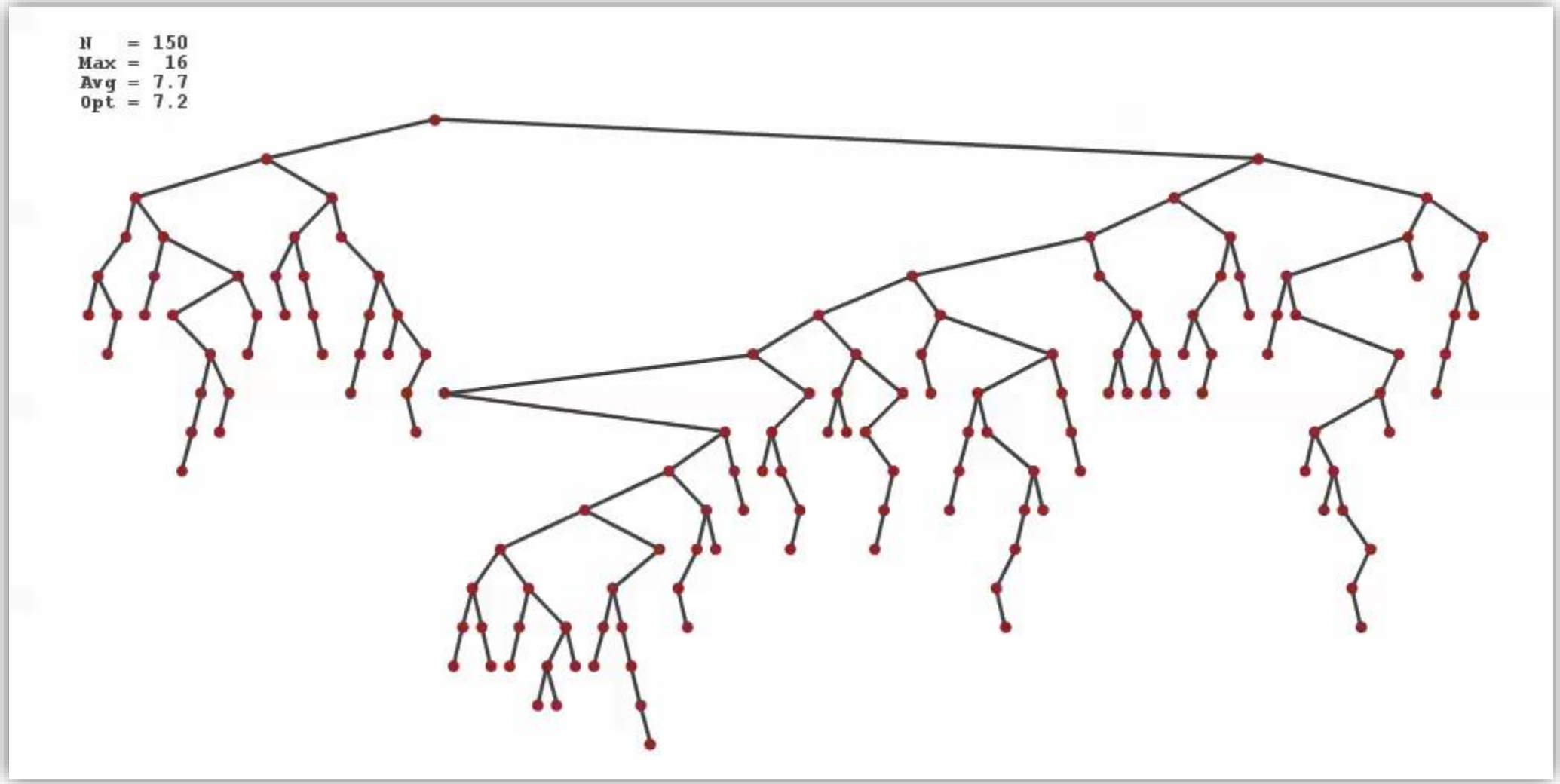
- حالت ۳) گره x **دو** فرزند دارد.
  - گره بعدی x را پیدا کن (s)
  - گره s را از درخت حذف کن (با استفاده از حالت ۱ یا ۲)
  - کلید s را به جای کلید x قرار بده



حذف (۵)


```
public Void delete(Key key)
{  root = delete(root, key);  }

private Node delete(Node x, Key key)
{
   if (x == null) return null;

   int cmp = key.compareTo(x.key);
   if      (cmp  < 0) x.left  = delete(x.left,  key);
   else if (cmp  > 0) x.right = delete(x.right, key);
   else
   {
      if (x.right == null) return x.left;
      if (x.left  == null) return x.right;

      Node t = x;
      x = min(t.right);
      x.right = deleteMin(t.right);
      x.left = t.left;
   }
   x.N = 1 + size(x.left) + size(x.right);
   return x;
}
```

حذف هیبارد: دنباله‌ای از درج و حذف تصادفی



پیاده‌سازی جدول نماد: خلاصه

پیاده‌سازی	بدترین حالت			حالت متوسط			ترتیب؟
	جستجو	درج	حذف	جستجو	درج	حذف	
جستجوی ترتیبی (لیست نامرتب)	N	N	N	N/2	N	N/2	خیر
جستجوی دودویی (آرایه مرتب)	lg N	N	N	lg N	N/2	N/2	بله
درخت جستجوی دودویی	N	N	N	1.39 lg N	1.39 lg N	√N	بله

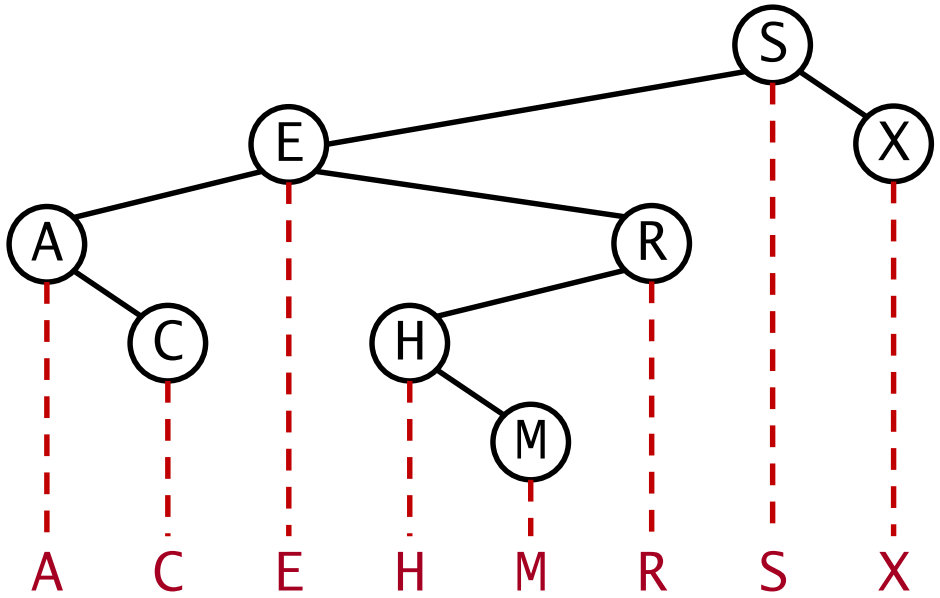
در صورتی که عمل حذف مجاز باشد، زمان اجرای عملیات دیگر **BST** *نیز برابر با* **√N** *خواهد بود!*

- درخت قرمز-سیاه. تضمین زمان اجرای لگاریتمی برای همه‌ی عملیات. [با ما باشید]

مرتب‌سازی درختی

پیمایش میان ترتیب درخت جستجوی دودویی

- گزاره. در پیمایش میان‌ترتیب یک درخت جستجوی دودویی، گره‌ها به ترتیب مقادیر کلیدشان از کوچک به بزرگ ملاقات می‌شوند.

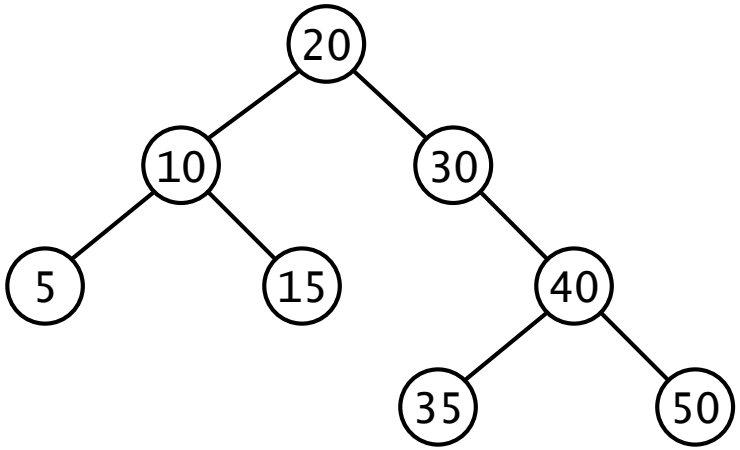


مرتب‌سازی درختی (۱)

- مسئله. مرتب‌سازی n کلید.
  - با استفاده از عناصر داده شده یک درخت جستجوی دودویی ایجاد کن؛
  - درخت حاصل را به روش میان‌ترتیب پیمایش کن.
- مثال.

```
20, 30, 10, 40, 5, 35, 50, 15
```

(1) BST



(2) INORDER

```
5,10,15,20,30,35,40,50
```

مرتب‌سازی درختی (۲)

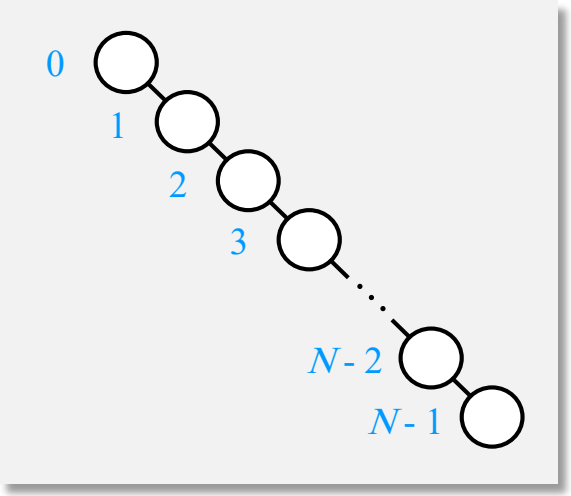

```
public static <Key extends Comparable <Key>> void sort(Key[] a)
{
   BST<Key, Integer> bst = new BST();
   for (int i = 0; i < a.length; i++)
      bst.put(a[i], i);
   int i = 0;
   for (Key key : bst.inorder())
      a[i++] = key;
}
```

	بهترین حالت	حالت متوسط	بدترین حالت
مرتب‌سازی درختی	O(N lg N)	O(N lg N)	$O(N^2)$

مرتب‌سازی درختی (۳)

- گزاره. کل زمان مصرفی به منظور درج n کلید متمایز در یک درخت جستجوی دودویی برابر است با مجموع سطح گره‌ها در درخت حاصل.

- تحلیل پیچیدگی زمانی در بدترین حالت.
  - کلیدها به ترتیب افزایشی (کاهشی) درج شوند.



$$T(N) = 1 + 2 + \cdots (N-1) = \frac{1}{2}N(N-1) \sim \frac{1}{2}N^2$$

مرتب سازی درختی (۴)

- تحلیل پیچیدگی زمانی در حالت متوسط.

$$T(N) = \frac{1}{N}\sum_{i=1}^{N}[T(i-1) + T(N-i) + (N-1)]$$

$$= \frac{2}{N}\sum_{i=1}^{N} T(i-1) + (N-1)$$

$$T(N) = 2(N+1)\ln N \cong 1.39(N+1)\lg N \sim 1.39Nlg\ N$$

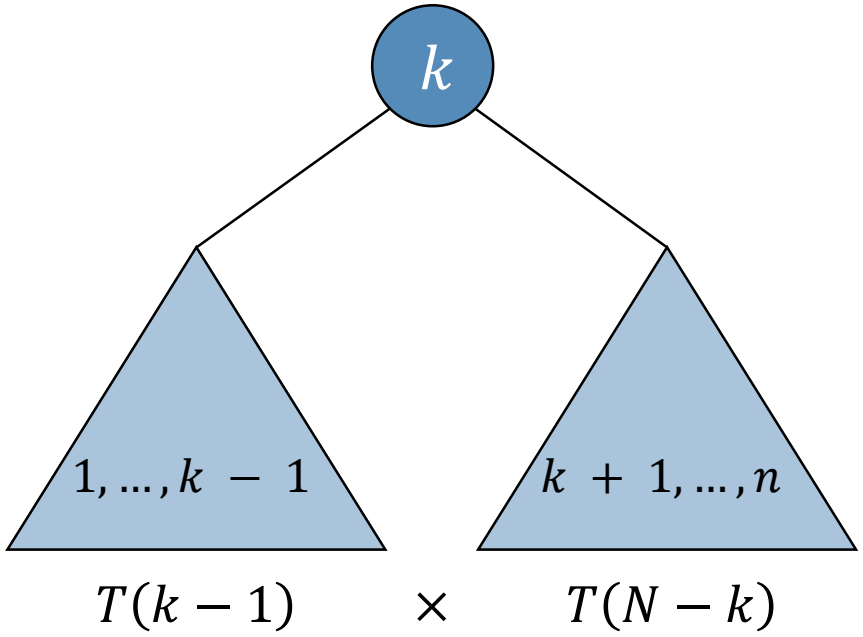
مرتب‌سازی درختی (۵)

- تحلیل پیچیدگی زمانی در بهترین حالت.
  - بهترین حالت زمانی رخ می دهد که درخت حاصل حداقل ارتفاع ممکن را داشته باشد.
    - حداقل ارتفاع برای یک درخت دودویی با n گره برابر است با $\lfloor \lg N \rfloor$
  - در یک درخت دودویی شامل N گره که حداقل ارتفاع ممکن را دارد، تعداد برگها برابر است با $(N/2)$ و درج همین تعداد گره به زمان $(N/2)\lg N$ نیاز دارد.
    - مجموع زمان لازم برای درج بقیه‌ی گره‌ها کمتر از مقدار فوق است (چرا؟).
  - بنابراین درج N کلید در بهترین حالت حداقل به زمان $N \lg N$ نیاز دارد.

یک مسئله‌ی شمارشی

شمارش تعداد درخت‌های جستجوی دودویی

- س. با اعداد ۱ تا n چند درخت جستجوی دودویی متفاوت می‌توان ساخت؟
- ج. تعداد درخت‌های ممکن برابر است با جمله n اُم در دنباله‌ی کاتالان.



$$T(N) = \sum_{k=1}^{N} T(k-1) \cdot T(N-k) = \frac{1}{N+1}\binom{2N}{N}$$