درخت ها

درخت های دودویی (Binary tree)

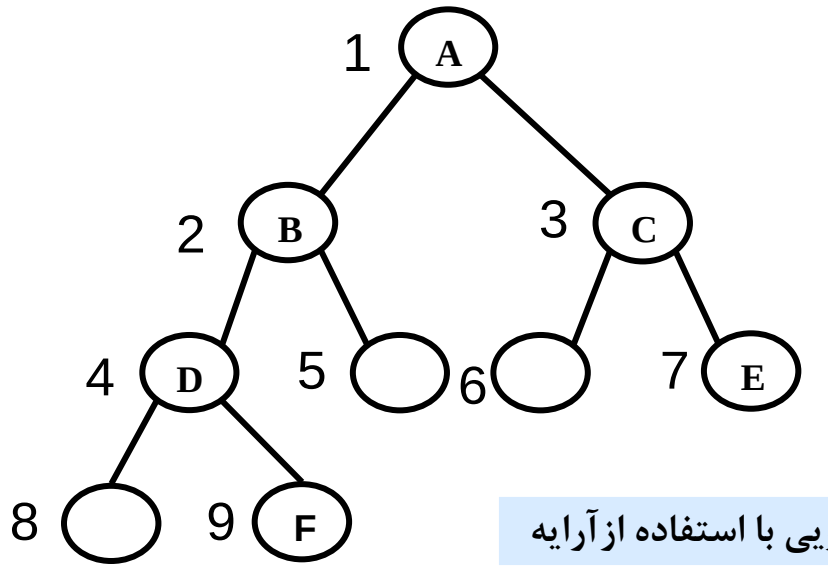
درختی مرتب که هر گره آن حداکثر دو فرزند دارد با نام های فرزند چپ و فرزند راست. اگر گرهی یک فرزند داشته باشد باید مشخص باشد که فرزند چپ است یا راست.

نمایش درخت های دودویی با استفاده از آرایه

۱- با استفاده از گره های مصنوعی (کمکی)، درخت را به یک درخت دودویی کامل تبدیل می کنیم.

۲- گره های درخت را با ترتیبی که در پیمایش اول سطحی ظاهر (BFS) می شوند در آرایه ذخیره می کنیم.

نمایش درخت های دودویی با استفاده از آرایه

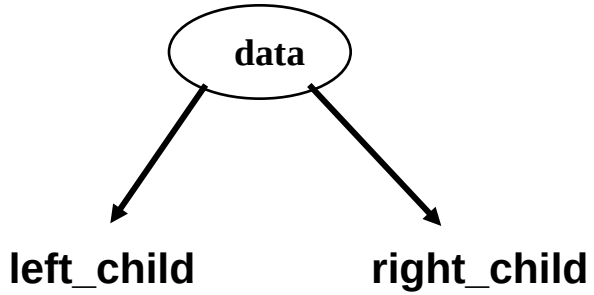


نمایش درخت دودویی با استفاده ازآرایه

A
B
C
D
-
-
E
-
F

اگر شماره گرهی i باشد والد آن دارای شماره [i/2] است و فرزندان چپ و راست آن به ترتیب دارای شماره های 2i+1, 2i هستند.

نمایش پیوندی



نمایش یک گره درخت دودویی

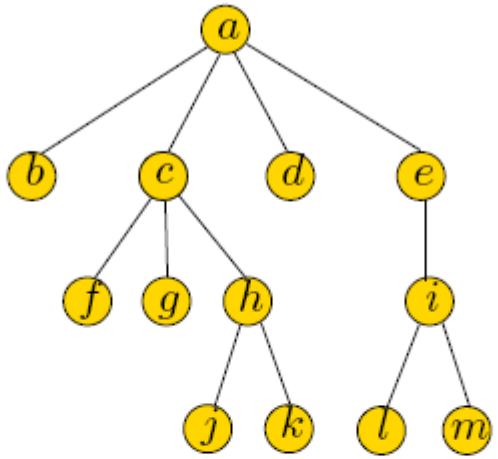
نمایش درخت ها

پیاده سازی با :

آرایه

لیست های پیوندی

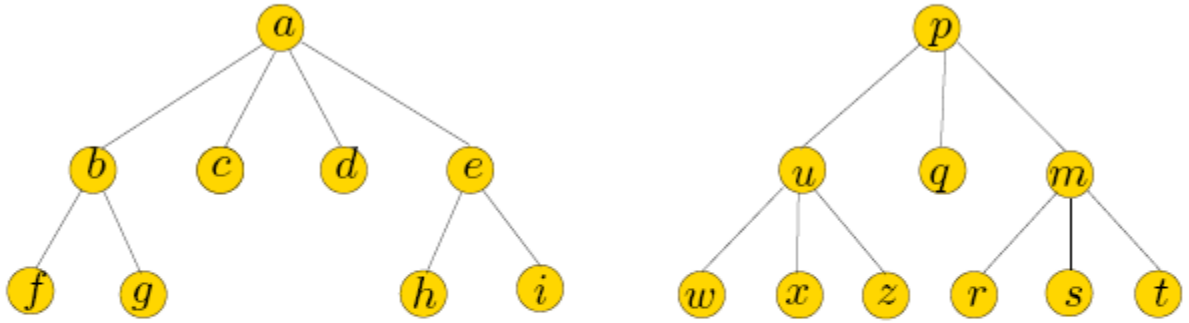
نمایش درخت ها با استفاده از آرایه



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

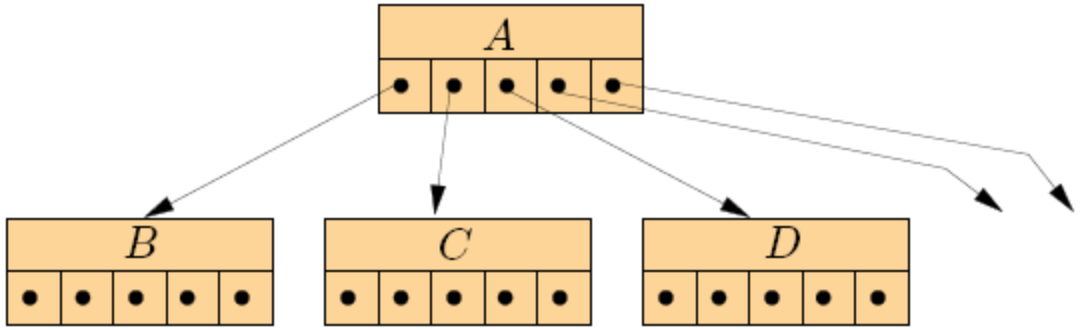
نمایش درخت ها با استفاده از آرایه

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

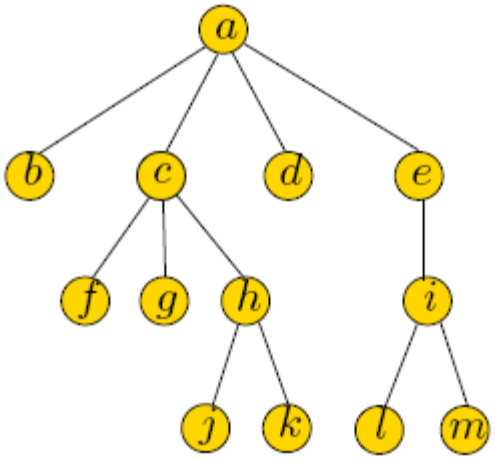


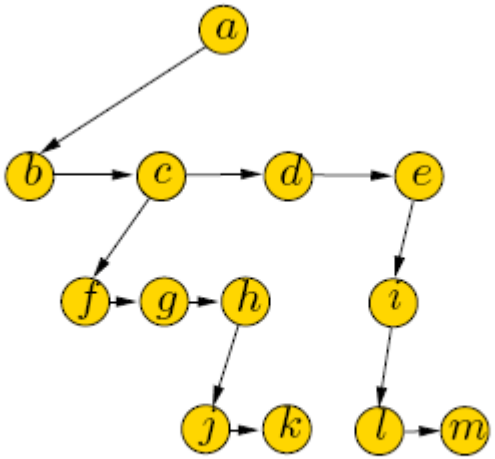
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
a	b	c	d	e	f	p	u	g	h	i	q	m	r	s	t	w	x	z
۰	۱	۱	۱	۱	۲	۰	۷	۲	۵	۵	۷	۷	۱۳	۱۳	۱۳	۸	۸	۸

نمایش درخت ها با استفاده از نمایش پیوندی



نمایش درخت ها با استفاده از نمایش پیوندی



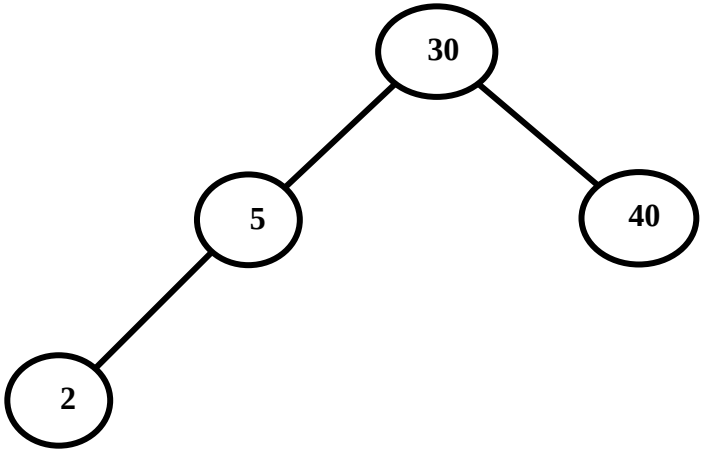


درختان جستجوی دودویی (BST)

یک درخت جستجوی دودویی یک درخت دودویی است که اگر درخت تهی نباشد خصوصیات زیر را برآورده می کند :

- **کلیدهای واقع در زیردرخت غیرتهی چپ هر گره باید کوچکترمساوی مقدار کلید واقع در آن گره باشد .**
- **کلیدهای واقع در زیردرخت غیرتهی راست هر گره باید بزرگترمساوی مقدار کلید واقع در آن گره باشد.**

درختان جستجوی دودویی



مثال

درخت جستجوی دودویی

مجموعه عناصر:

درخت جستجوی دودویی

عملیات:

درج مقدار در درخت.

حذف گره ای از درخت.

جستجوی درخت

پیمایش درخت

یافتن گره بعدی و گره قبلی

یافتن کلید مینیمم

حذف کلید مینیمم

پیمایش در یک درخت جستجوی دودویی O(n)

- پیمایش پس ترتیب
- پیمایش پیش ترتیب
- پیمایش میان ترتیب

```
INORDER-TREE-WALK(x)
1  if x ≠ NIL
2      INORDER-TREE-WALK(x.left)
3      print x.key
4      INORDER-TREE-WALK(x.right)
```

جستجو در یک درخت جستجوی دودویی O(h)

```
TREE-SEARCH(x, k)
1   if x == NIL or k == x.key
2       return x
3   if k < x.key
4       return TREE-SEARCH(x.left, k)
5   else return TREE-SEARCH(x.right, k)
```

جستجو در یک درخت جستجوی دودویی O(h)

```
ITERATIVE-TREE-SEARCH(x, k)
1  while x ≠ NIL and k ≠ x.key
2      if k < x.key
3          x = x.left
4      else x = x.right
5  return x
```

یافتن گره با مقدار مینیمم و گره با مقدار ماکزیمم در یک درخت جستجوی دودویی

```
TREE-MINIMUM(x)
1  while x.left ≠ NIL
2      x = x.left
3  return x
```

```
TREE-MAXIMUM(x)
1  while x.right ≠ NIL
2      x = x.right
3  return x
```

یافتن گره بعدی و گره قبلی یک گره در یک درخت جستجوی دودویی

```
TREE-SUCCESSOR(x)
1  if x.right ≠ NIL
2        return TREE-MINIMUM(x.right)
3  y = x.p
4  while y ≠ NIL and x == y.right
5        x = y
6        y = y.p
7  return y
```

درج عنصری به داخل درخت جستجوی دودویی

```
TREE-INSERT(T, z)
 1  y = NIL
 2  x = T.root
 3  while x ≠ NIL
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == NIL
10        T.root = z          // tree T was empty
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
```

حذف گره x از درخت جستجوی دودویی

۱- گرهی که می خواهیم حذف کنیم برگ است.

(در این حالت اشاره گری از پدر x را که به x اشاره می کند را NULL کنید.)

۲- فقط یک فرزند دارد.

(در این حالت ، آدرس فرزند x را در اشاره گری از پدر x که به x اشاره می کند قرار دهید.)

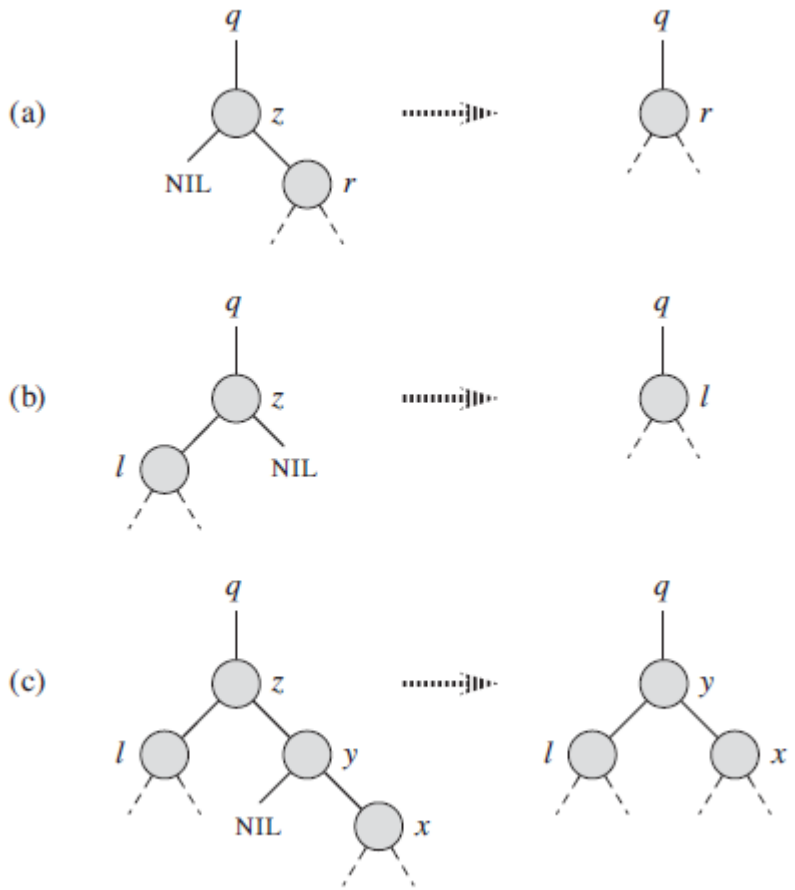
۳- دو فرزند دارد.

(در این حالت، مینیمم زیردرخت ریشه دار در فرزند راست x را در گره x قرار بده و گره مینیمم زیردرخت ریشه دار در فرزند راست x را حذف کنید)

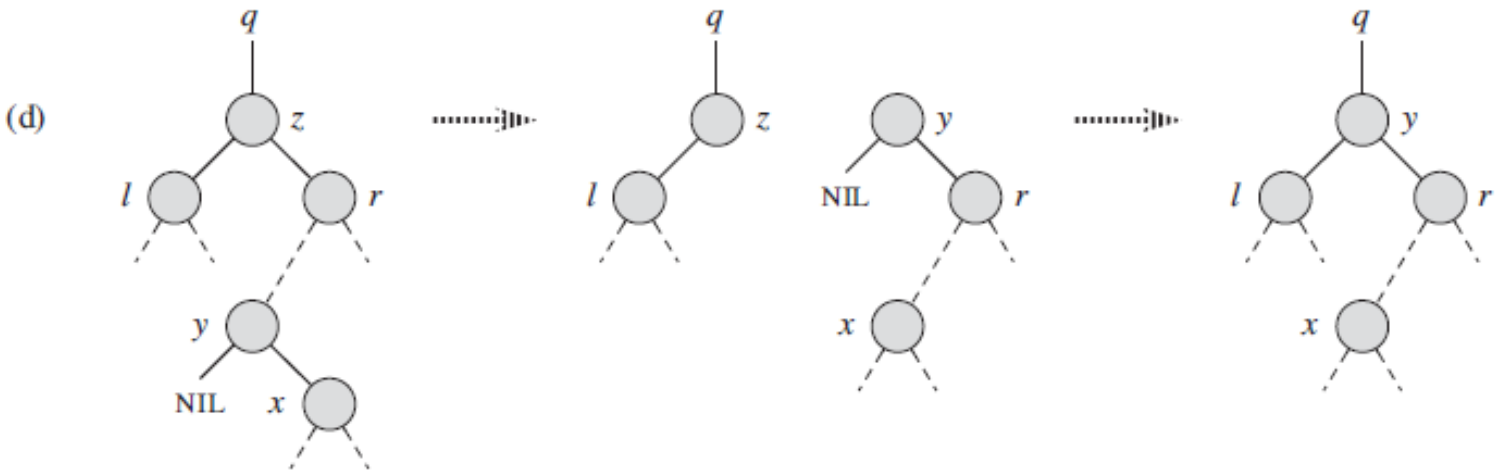
حذف عنصری از درخت جستجوی دودویی

```
TRANSPLANT(T, u, v)
1  if u.p == NIL
2        T.root = v
3  elseif u == u.p.left
4        u.p.left = v
5  else u.p.right = v
6  if v ≠ NIL
7        v.p = u.p
```

حذف عنصری از درخت جستجوی دودویی



حذف عنصری از درخت جستجوی دودویی



حذف عنصری از درخت جستجوی دودویی

```
TREE-DELETE(T, z)
 1  if z.left == NIL
 2      TRANSPLANT(T, z, z.right)
 3  elseif z.right == NIL
 4      TRANSPLANT(T, z, z.left)
 5  else y = TREE-MINIMUM(z.right)
 6      if y.p ≠ z
 7          TRANSPLANT(T, y, y.right)
 8          y.right = z.right
 9          y.right.p = y
10      TRANSPLANT(T, z, y)
11      y.left = z.left
12      y.left.p = y
```

درختان قرمز-سیاه

یک درخت جستجوی دودویی که :

۱- هرگره یا قرمز است و یا سیاه

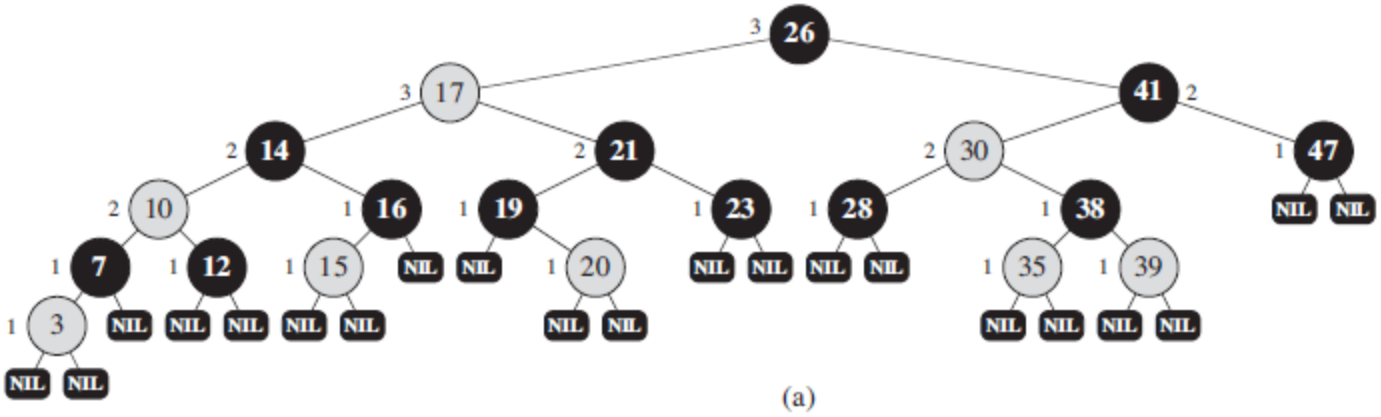
۲- ریشه سیاه است.

۳- تمام برگ ها سیاه هستند.

۴- اگر یک گره قرمز باشد، آنگاه هردو فرزند آن سیاه است.

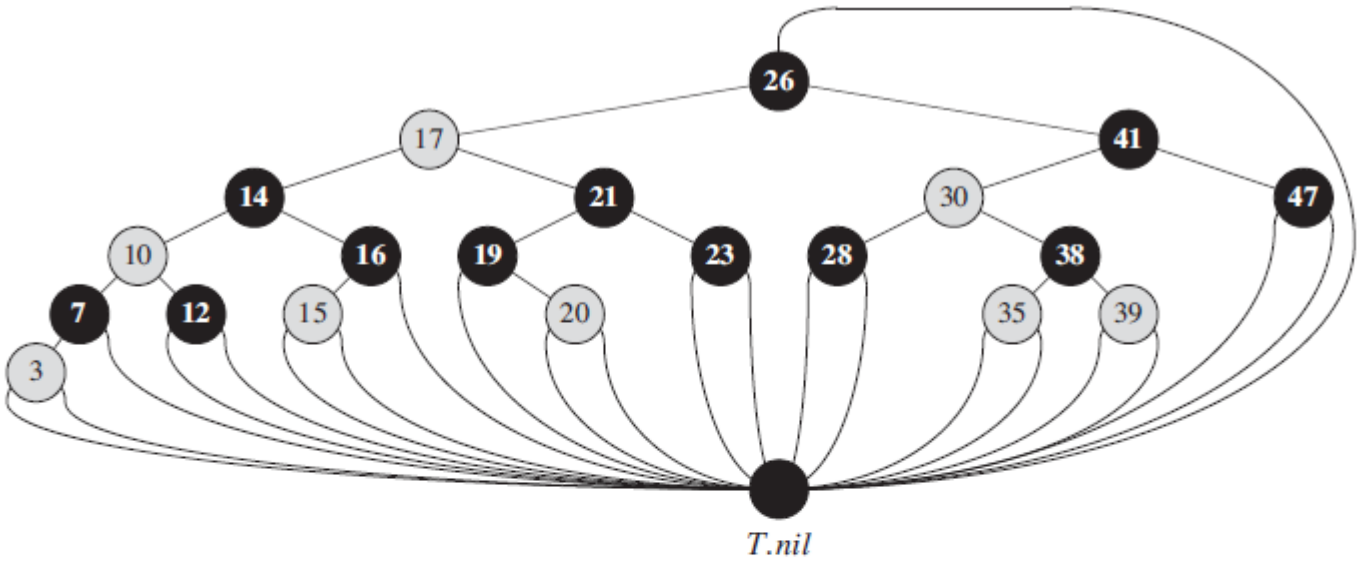
۵- برای هر گره، تمام مسیرها از آن گره تا برگهایی که نواده آن هستند، حاوی تعدادی مساوی گره ی سیاه است.

درختان قرمز-سیاه

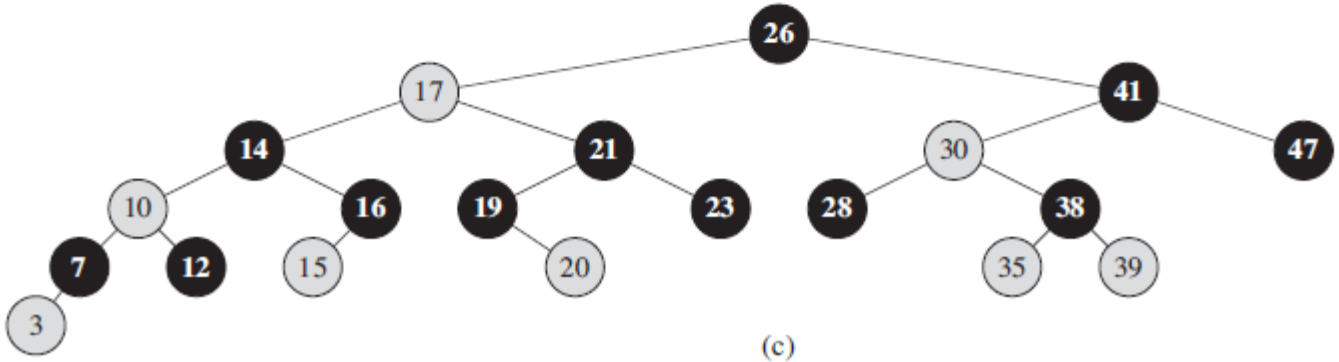


(a)

درختان قرمز-سیاه



(b)



(c)

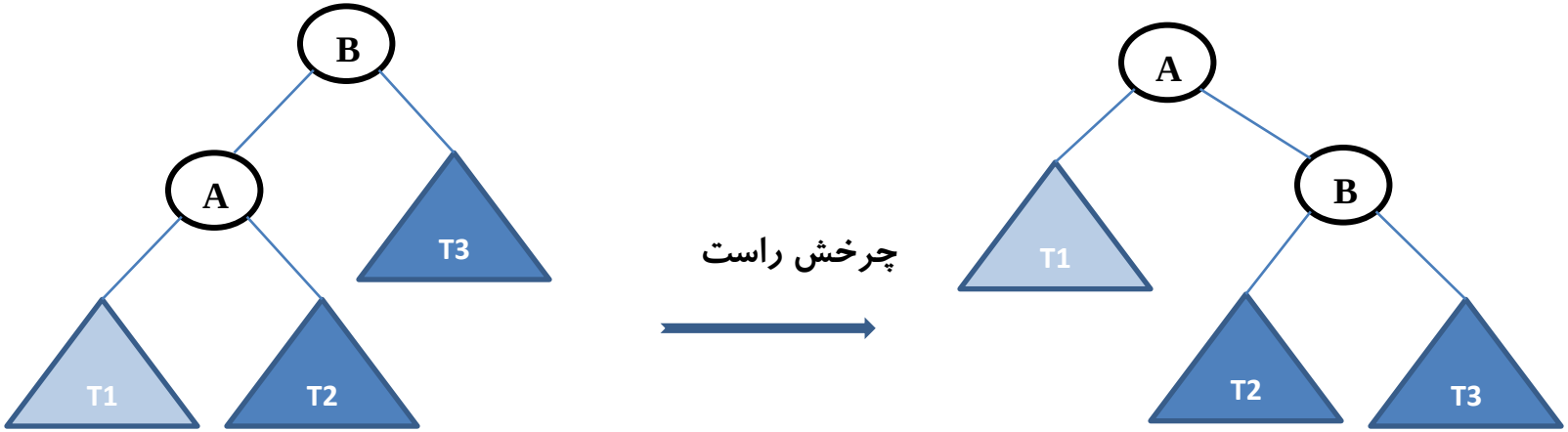
ارتفاع درختان قرمز-سیاه

قضیه: ارتفاع یک درخت قرمز-سیاه با n گره داخلی حداکثر 2log(n+1) است.

bh(x): ارتفاع سیاه: تعداد گره های سیاه (بجز خود گره) هر مسیراز x تا هر برگ اززیر درخت x.

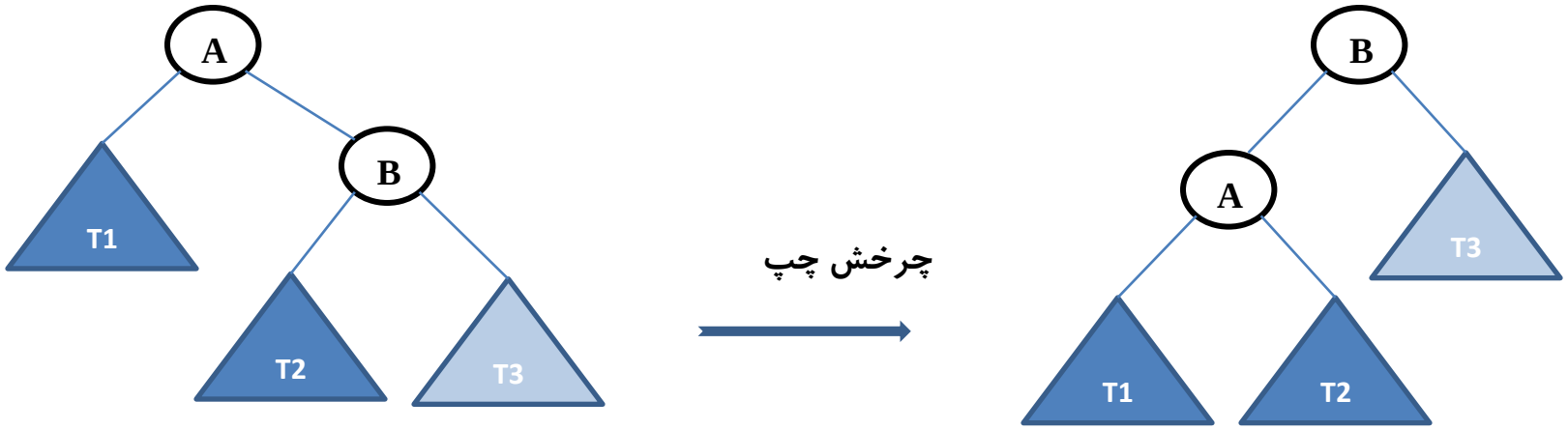
زیر درخت x حداکثر 2^bh(x)-1 گره داخلی دارد.

چرخش ها



$$((T1 \ \textcircled{A} \ T2) \ \textcircled{B} \ T3) = (T1 \ \textcircled{A} \ (T2 \ \textcircled{B} \ T3))$$

چرخش ها

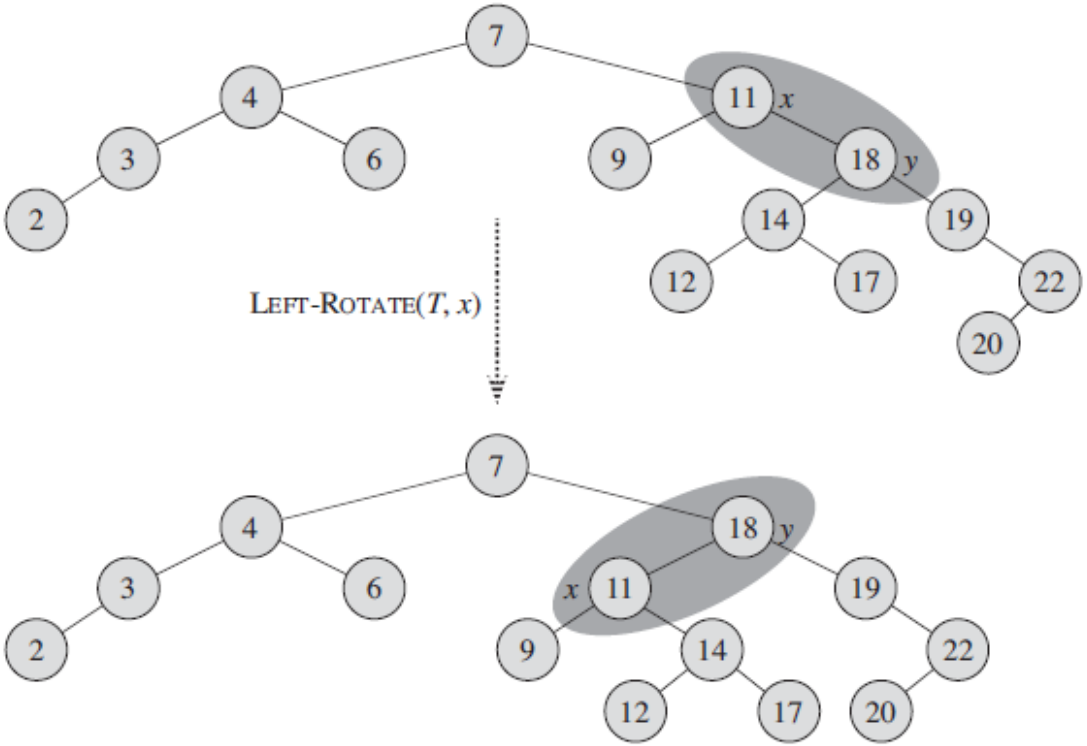


$$(T1 \ \textcircled{A} \ (T2 \ \textcircled{B} \ T3)) = ((T1 \ \textcircled{A} \ T2) \ \textcircled{B} \ T3)$$

چرخش چپ

```
LEFT-ROTATE(T, x)
 1  y = x.right                  // set y
 2  x.right = y.left             // turn y's left subtree into x's right subtree
 3  if y.left ≠ T.nil
 4       y.left.p = x
 5  y.p = x.p                    // link x's parent to y
 6  if x.p == T.nil
 7       T.root = y
 8  elseif x == x.p.left
 9       x.p.left = y
10  else x.p.right = y
11  y.left = x                   // put x on y's left
12  x.p = y
```

چرخش چپ


7
4
11
x
3
6
9
18
y
2
14
19
12
17
22
20
LEFT-ROTATE(T, x)
7
4
18
y
3
6
x
11
19
2
9
14
22
12
17
20

درج گره جدید

```
RB-INSERT(T, z)
 1  y = T.nil
 2  x = T.root
 3  while x ≠ T.nil
 4        y = x
 5        if z.key < x.key
 6              x = x.left
 7        else x = x.right
 8  z.p = y
 9  if y == T.nil
10        T.root = z
11  elseif z.key < y.key
12        y.left = z
13  else y.right = z
14  z.left = T.nil
15  z.right = T.nil
16  z.color = RED
17  RB-INSERT-FIXUP(T, z)
```

درختان قرمز-سیاه

یک درخت جستجوی دودویی که :

۱- هرگره یا قرمز است و یا سیاه

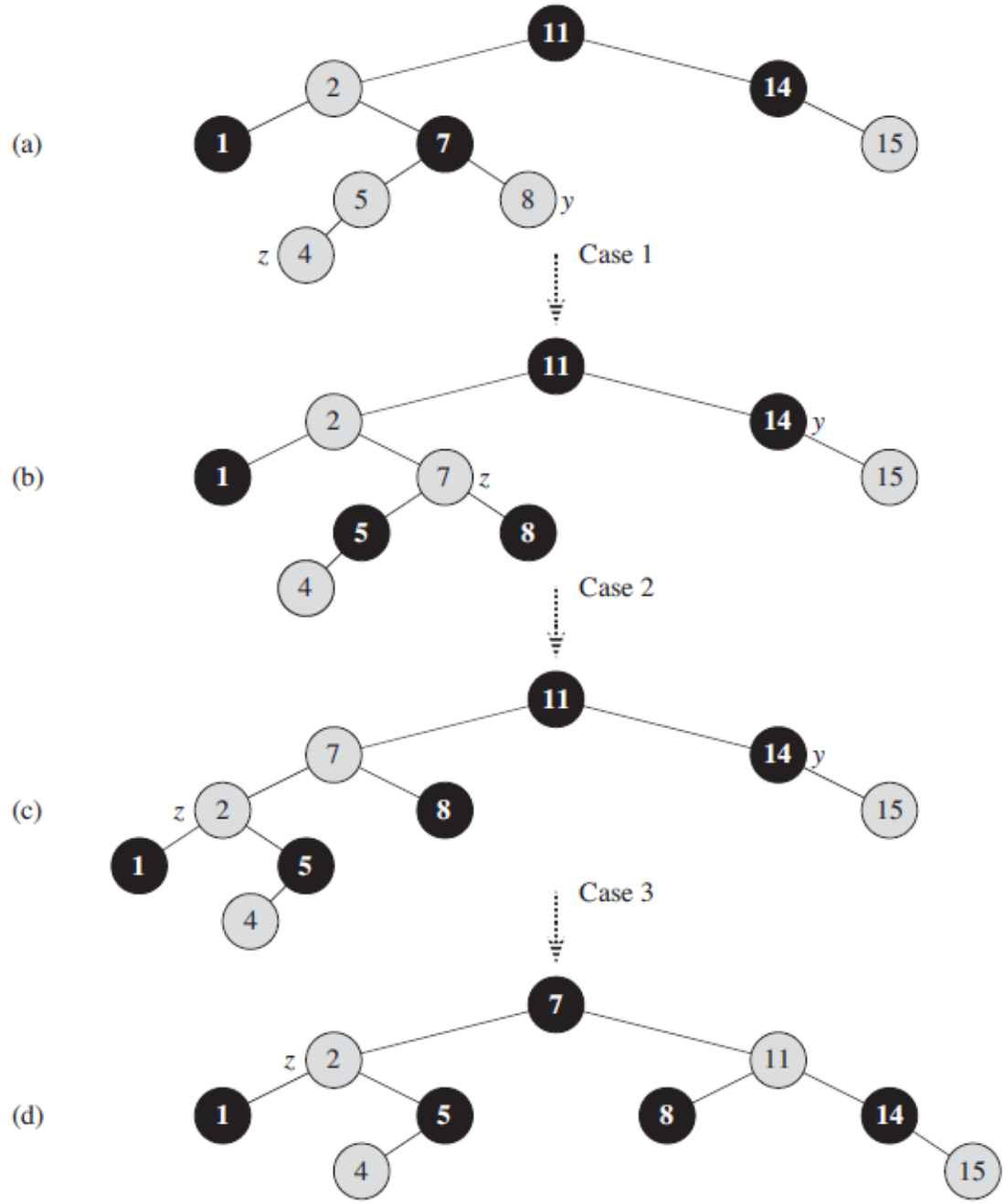
۲- ریشه سیاه است.

۳- تمام برگ ها سیاه هستند.

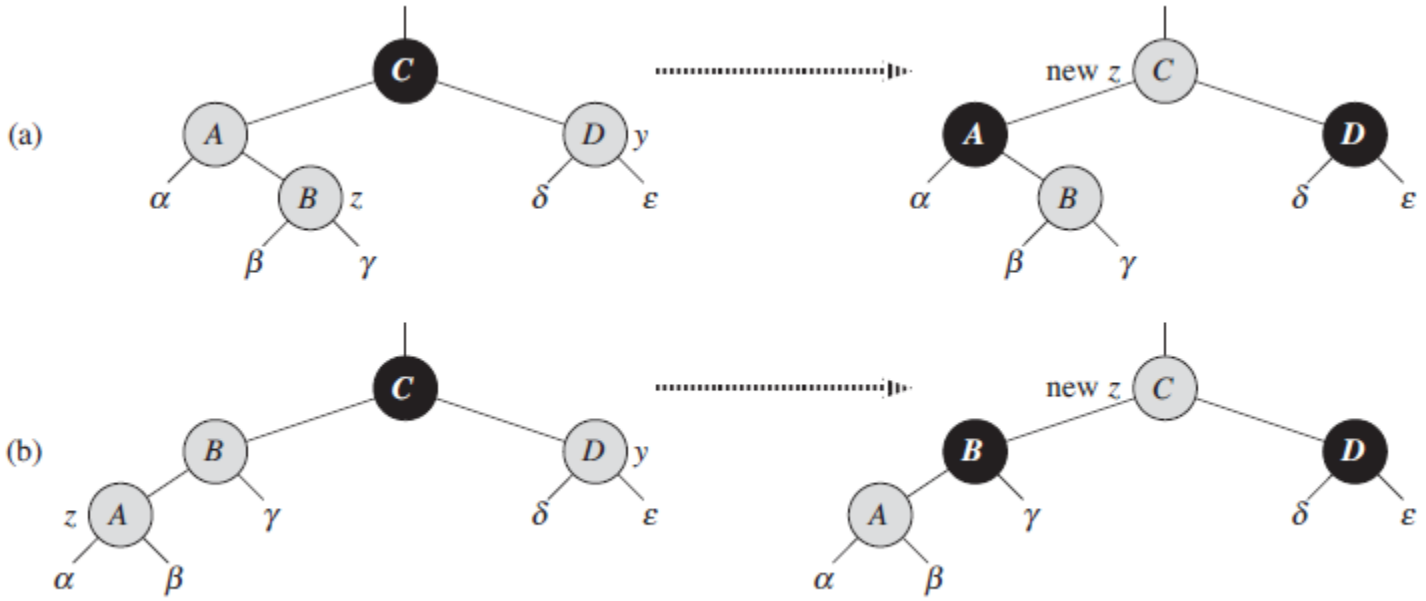
۴- اگر یک گره قرمز باشد، آنگاه هردو فرزند آن سیاه است.

۵- برای هر گره، تمام مسیرها از آن گره تا برگهایی که نواده آن هستند، حاوی تعدادی مساوی گره ی سیاه است.

درج گره جدید


(a)
11
2
14
1
7
15
5
8
y
z
4
Case 1
(b)
11
2
14
y
1
7
z
15
5
8
4
Case 2
(c)
11
7
14
y
z
2
8
15
1
5
4
Case 3
(d)
7
z
2
11
1
5
8
14
4
15

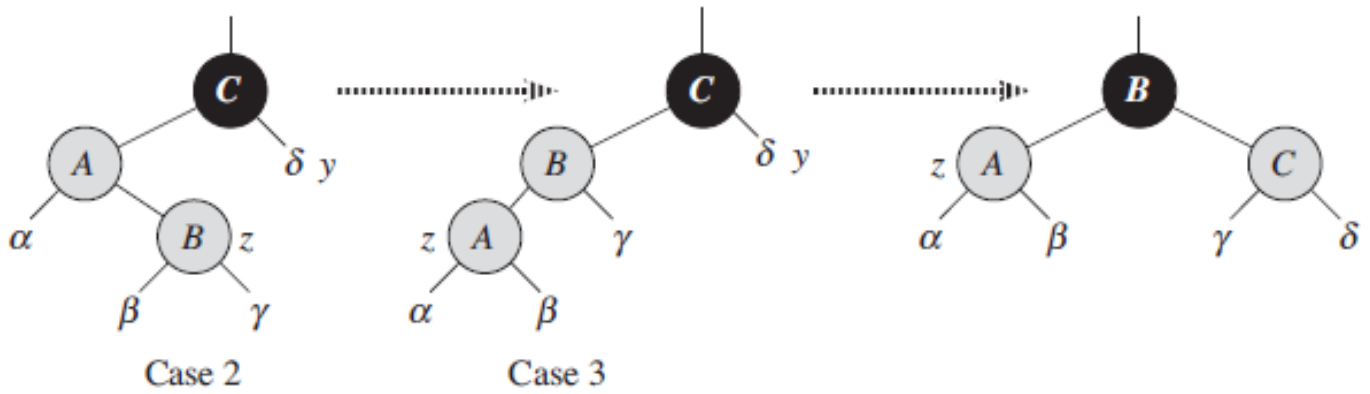
درج گره جدید
حالت ۱: عموی z قرمز باشد



درج گره جدید

حالت ۲: عموی z سیاه و z فرزند راست است

حالت ۳: عموی z سیاه و z فرزند چپ است



درج گره جدید

```
RB-INSERT-FIXUP(T, z)
 1  while z.p.color == RED
 2      if z.p == z.p.p.left
 3          y = z.p.p.right
 4          if y.color == RED
 5              z.p.color = BLACK
 6              y.color = BLACK
 7              z.p.p.color = RED
 8              z = z.p.p
 9          else if z == z.p.right
10                  z = z.p
11                  LEFT-ROTATE(T, z)
12              z.p.color = BLACK
13              z.p.p.color = RED
14              RIGHT-ROTATE(T, z.p.p)
15      else (same as then clause
                with "right" and "left" exchanged)
16  T.root.color = BLACK
```

حذف یک گره

```
RB-TRANSPLANT(T, u, v)
1   if u.p == T.nil
2       T.root = v
3   elseif u == u.p.left
4       u.p.left = v
5   else u.p.right = v
6   v.p = u.p
```

حذف یک گره

```
RB-DELETE(T, z)
 1  y = z
 2  y-original-color = y.color
 3  if z.left == T.nil
 4       x = z.right
 5       RB-TRANSPLANT(T, z, z.right)
 6  elseif z.right == T.nil
 7       x = z.left
 8       RB-TRANSPLANT(T, z, z.left)
 9  else y = TREE-MINIMUM(z.right)
10       y-original-color = y.color
11       x = y.right
12       if y.p == z
13            x.p = y
14       else RB-TRANSPLANT(T, y, y.right)
15            y.right = z.right
16            y.right.p = y
17       RB-TRANSPLANT(T, z, y)
18       y.left = z.left
19       y.left.p = y
20       y.color = z.color
21  if y-original-color == BLACK
22       RB-DELETE-FIXUP(T, x)
```

حذف یک گره

- اگر گرهی که حذف و یا جابجا می شود سیاه باشد ممکن است یکی از مشکلات زیر به وجود آید.
- ۱: y ریشه باشد و گره x که جایگزین آن می شود قرمز باشد. (x سیاه می کنیم)
- ۲: گره جایگزین y(x) و پدر جدید آن (x.p) قرمز باشند.(مشکل را درخت بالا می بریم تا به ریشه برسد و ریشه را سیاه می کنیم)
- ۳: مسیرهای شامل y از نظر داشتن تعداد گره سیاه درست نباشند. (یک رنگ سیاه اضافی به x می دهیم)

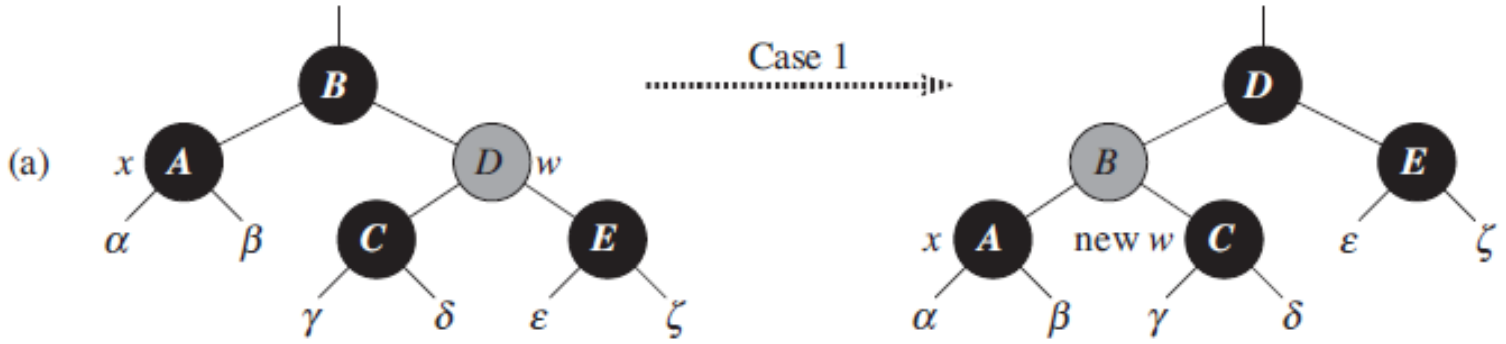
حذف یک گره

- فرض کنیم x فرزند چپ پدر جدیدش باشد و رنگ x سیاه باشد . w
- نکته: اگر x سیاه دوگانه باشد برادر x برگ نیست. برادر جدید x باشد.
- حالت ۱: برادر x قرمز باشد.
- حالت ۲: برادر x و هر دو فرزند w سیاه باشد.
- حالت ۳: برادر x سیاه و فرزند چپ w قرمز و فرزند راست w سیاه باشد.
- حالت ۴: برادر x سیاه و فرزند راست w قرمز باشد.

حذف یک گره

- حالت ۱: برادر X قرمز باشد.

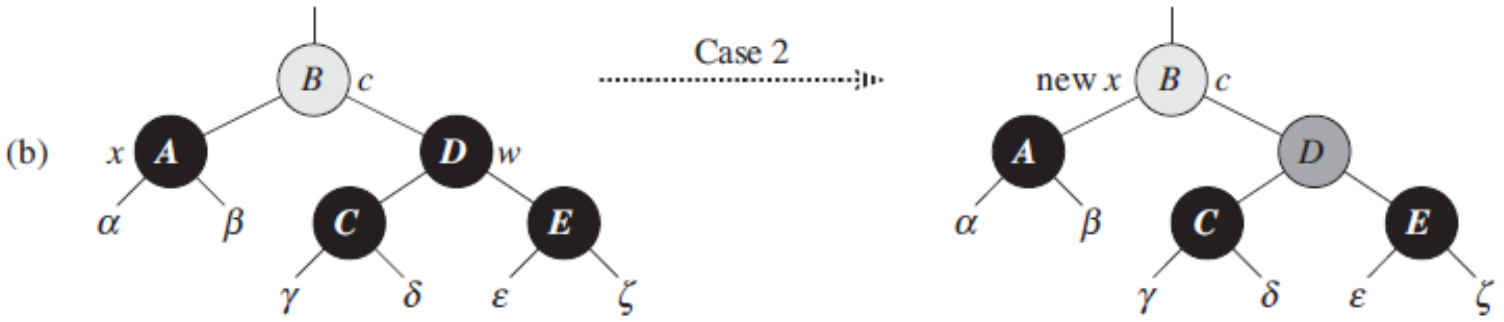
پدر Xسیاه است رنگ W و پدر X را باهم جابجا می کنیم و یک چرخش به چپ روی پدر X انجام می دهیم.



حذف یک گره

- حالت ۲: برادر X و هر دو فرزند W سیاه باشد.

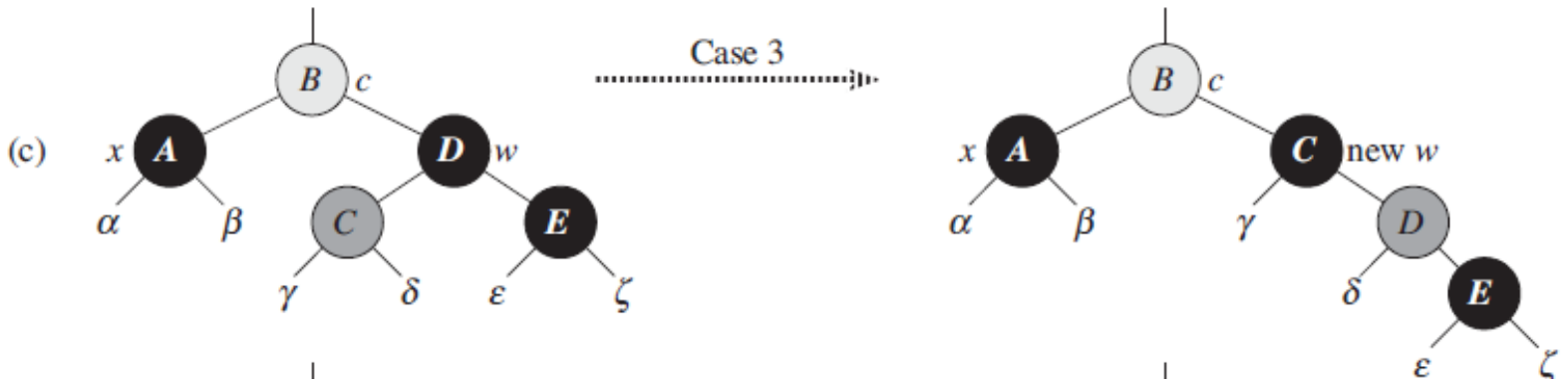
W را قرمز می کنیم و سیاهی اضافی X را به پدرش می دهیم، پدرX را بعنوان X جدید معرفی می کنیم.



حذف یک گره

- حالت ۳: برادر X سیاه و فرزند چپ W قرمز و فرزند راست W سیاه باشد.

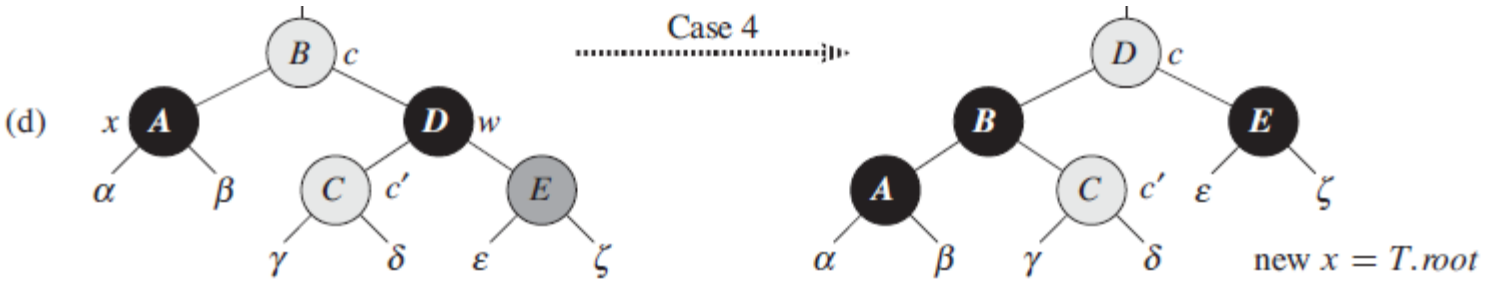
رنگ W و فرزند چپش را باهم عوض می کنیم و سپس یک چرخش چپ بر روی W انجام می دهیم.

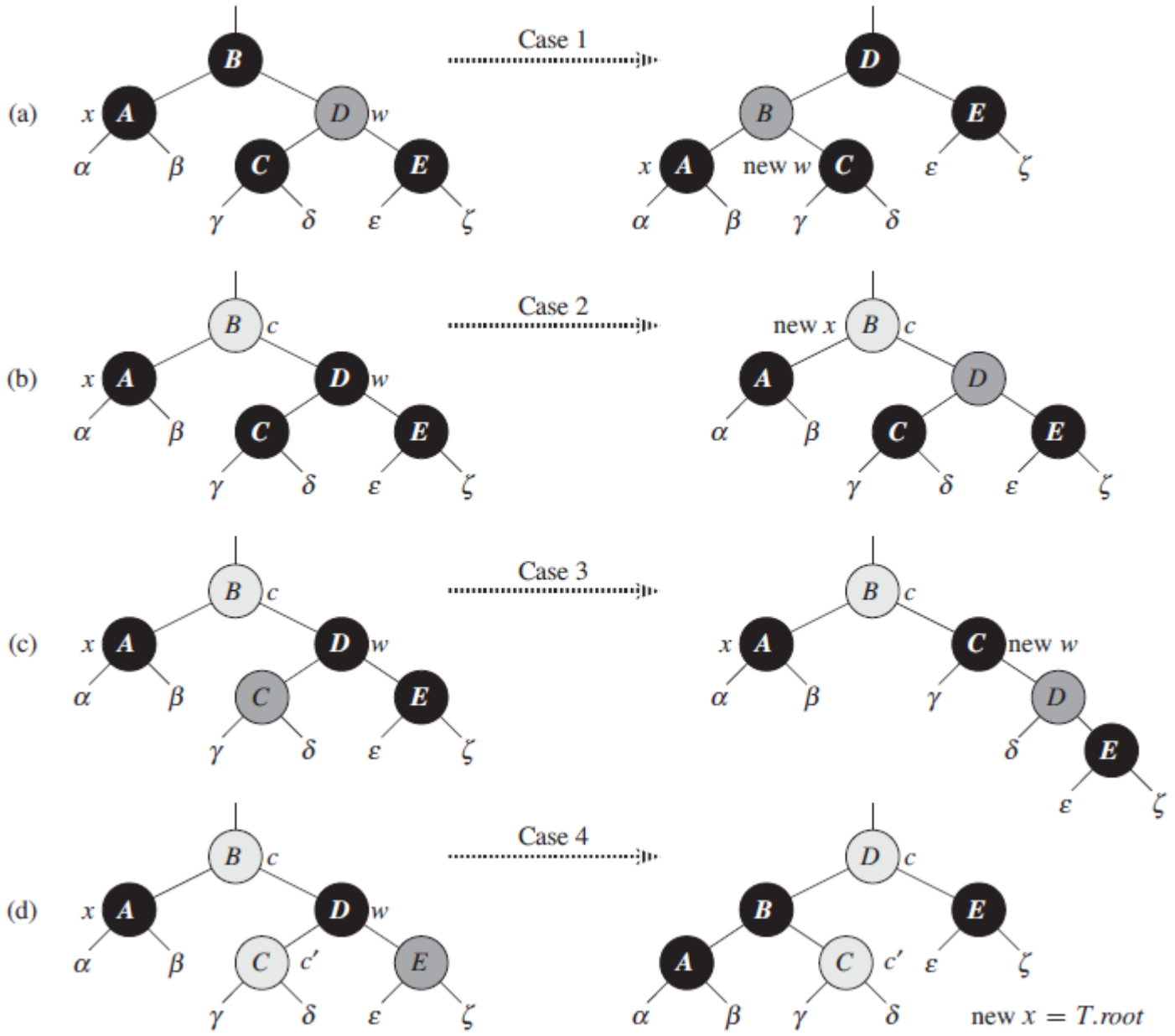


حذف یک گره

- حالت ۴ : برادر X سیاه و فرزند راست W قرمز باشد.

فرزند راست W را سیاه می‌کنیم و یک چرخش چپ روی پدر X انجام می دهیم و رنگ W و پدر آن را جابجا می کنیم. ریشه را بعنوان X جدید در نظر می‌گیریم.




(a)
Case 1
(b)
Case 2
(c)
Case 3
(d)
Case 4
new x = T.root

حذف یک گره

```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ T.root and x.color == BLACK
 2      if x == x.p.left
 3          w = x.p.right
 4          if w.color == RED
 5              w.color = BLACK                          // case 1
 6              x.p.color = RED                          // case 1
 7              LEFT-ROTATE(T, x.p)                      // case 1
 8              w = x.p.right                            // case 1
 9          if w.left.color == BLACK and w.right.color == BLACK
10              w.color = RED                            // case 2
11              x = x.p                                  // case 2
12          else if w.right.color == BLACK
13                  w.left.color = BLACK                 // case 3
14                  w.color = RED                        // case 3
15                  RIGHT-ROTATE(T, w)                   // case 3
16                  w = x.p.right                        // case 3
17              w.color = x.p.color                      // case 4
18              x.p.color = BLACK                        // case 4
19              w.right.color = BLACK                    // case 4
20              LEFT-ROTATE(T, x.p)                      // case 4
21              x = T.root                               // case 4
22      else (same as then clause with "right" and "left" exchanged)
23  x.color = BLACK
```

نوع داده مجردصف اولویت(priority queue)

- مجموعه عناصر:

مجموعه ای از داده ها که برای هرکدام اولویتی در نظرگرفته شده است.

- مجموعه عملیات ها:

پیدا کردن عنصر با بیشترین اولویت

حذف عنصر با بیشترین اولویت

درج یک عنصر

پیاده سازی صف اولویت

- استفاده از آرایه مرتب
- (درج O(n) ، یافتن عنصری با بیشترین اولویت O(1) ، حذف عنصری با بیشترین اولویت O(1))
- استفاده از آرایه نامرتب
- (درج O(1) ، یافتن عنصری با بیشترین اولویت O(n) ، حذف عنصری با بیشترین اولویت O(n))
- استفاده از آرایه نیمه مرتب (هرم های دودویی)
- (درج O(logn) ، یافتن عنصری با بیشترین اولویت O(1) ، حذف عنصری با بیشترین اولویت O(logn))

هرمها (heaps)

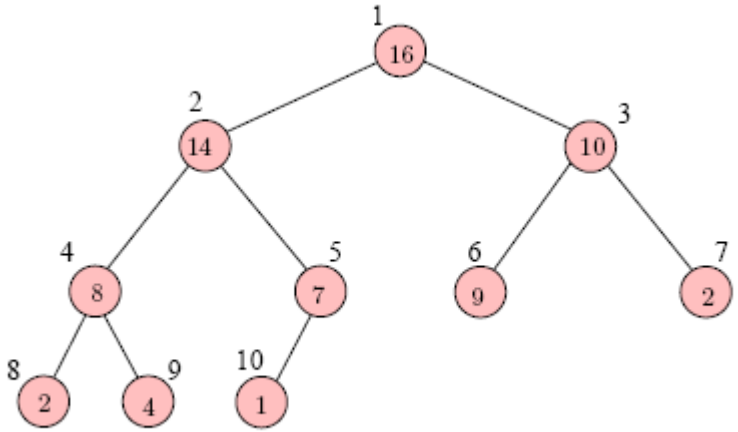
- هرم ماکزیمم (max-priority-tree, max-heap)
- (درختی که درآن مقدار هر گره از مقادیر فرزندانش کمتر نیست)

- هرم مینیمم (min-priority-tree, min-heap)
- (درختی که درآن مقدار هر گره از مقادیر فرزندانش بیشتر نیست)

هرم های ماکزیمم دودویی

- یک درخت دودویی کامل را که درآن مقدار هر گره از مقادیر فرزندانش بیشتر است، یک هرم ماکزیمم دودویی گویند.

هرم های ماکزیمم دودویی (Binary-max- heaps)



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

هرم های ماکزیمم دودویی (Binary-max- heaps)

max-heap (ویژگی‌ها)

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

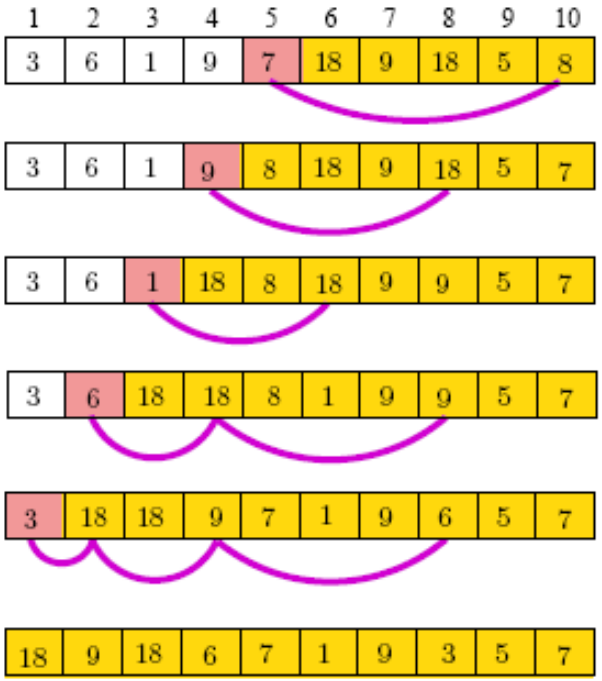
به‌صورت max-heap در آوردن $A[i..length[A]]$

```
MAX-HEAPIFY (A, i)
 1  l ← LEFTCHILD (i)
 2  r ← RIGHTCHILD (i)
 3  if l ≤ length[A]  and A[l] > A[i]
 4    then bigchild ← l
 5    else bigchild ← i
 6  if r ≤ length[A]  and A[r] > A[bigchild]
 7    then bigchild ← r
 8  if bigchild ≠ i
 9    then swap(A[i], A[bigchild])
10         MAX-HEAPIFY (A, bigchild)
```

تبدیل یک آرایه به max-heap

```
BUILD-HEAP(A)
1  for i ← ⌊length[A]/2⌋ downto 1
2      do HEAPIFY(A, i)
```

تبدیل یک آرایه به max-heap



تحلیل Build-Heap

$k = \log n$

$i = 1$ O(k)

$i = 2,3$ O(k-1)

$i = 4,5,6,7$ O(k-2)

.

.

.

$i = 2^s, 2^s + 1, ..., 2^{s+1} - 1$ O(k-s)

.

.

.

$i = 2^{k-1}, 2^{k-1} + 1, ..., 2^k - 1$ O(1)

تحلیل Build-Heap

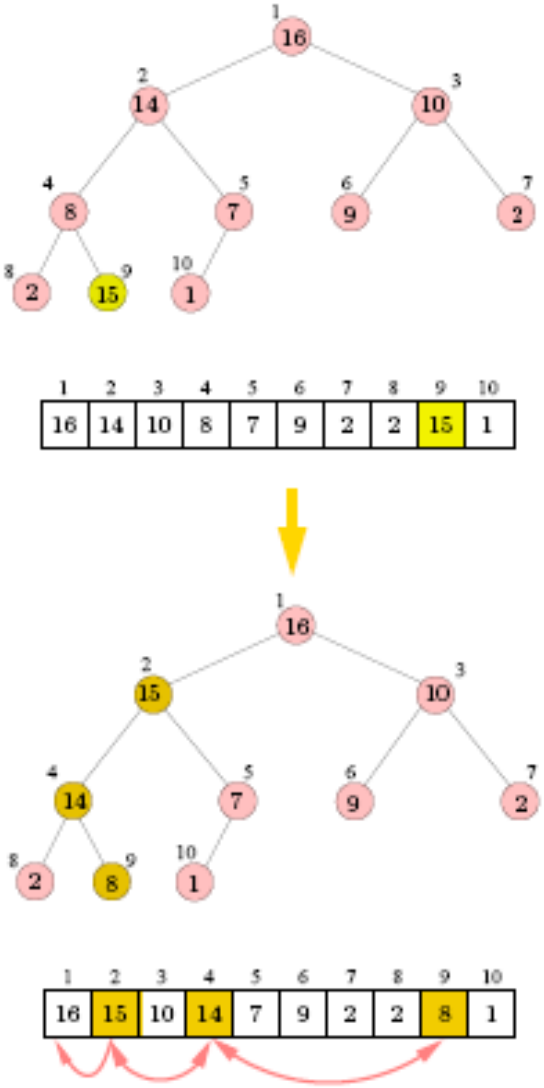
$$\sum_{s=0}^{k-1} 2^s (k-s) = n \sum_{s=0}^{k-1} 2^{s-k} (k-s) = n \sum_{i=1}^{k} \frac{i}{2^i}$$

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

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

پس ساخت heap از $O(n)$ است.

افزایش مقدار یک

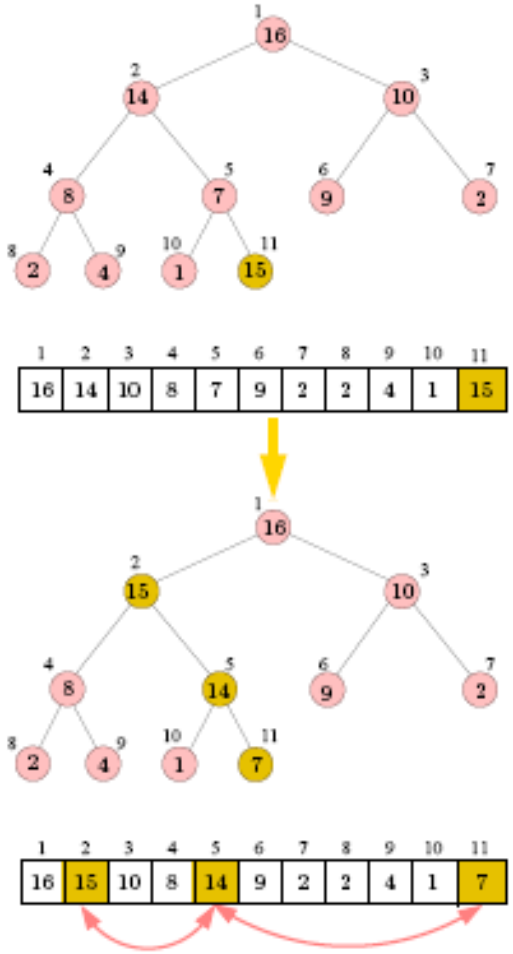


افزایش کلید

افزایش مقدار یک گره

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

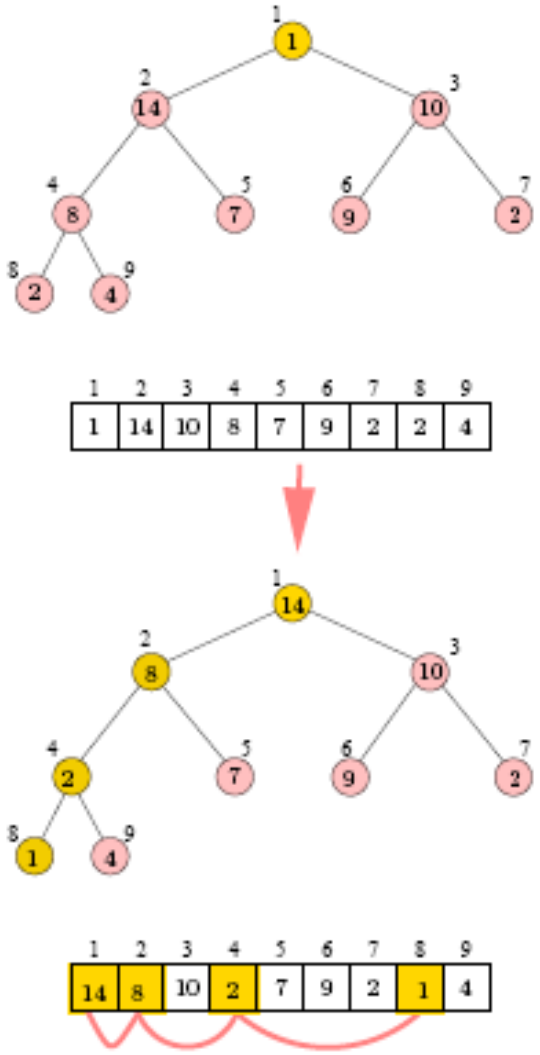
درج یک گره جدید



درج یک گره جدید

```
MAX-HEAP-INSERT(A, key)
1  length[A] ← length[A] + 1
2  A[length[A]] ← −∞
3  MAX-HEAP-INCREASE-KEY(A, length[A], key)
```

حذف گره ماکزیمم



حذف گره ماکزیمم

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