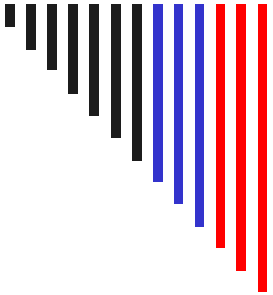
BLM212 Veri Yapıları

Heaps



Heap

Hedefler

- **Heap yapılarını tanımlamak ve gerçekleştirmek**
- **Heap ADT nin işleyişini ve kullanımını anlamak**

Heap

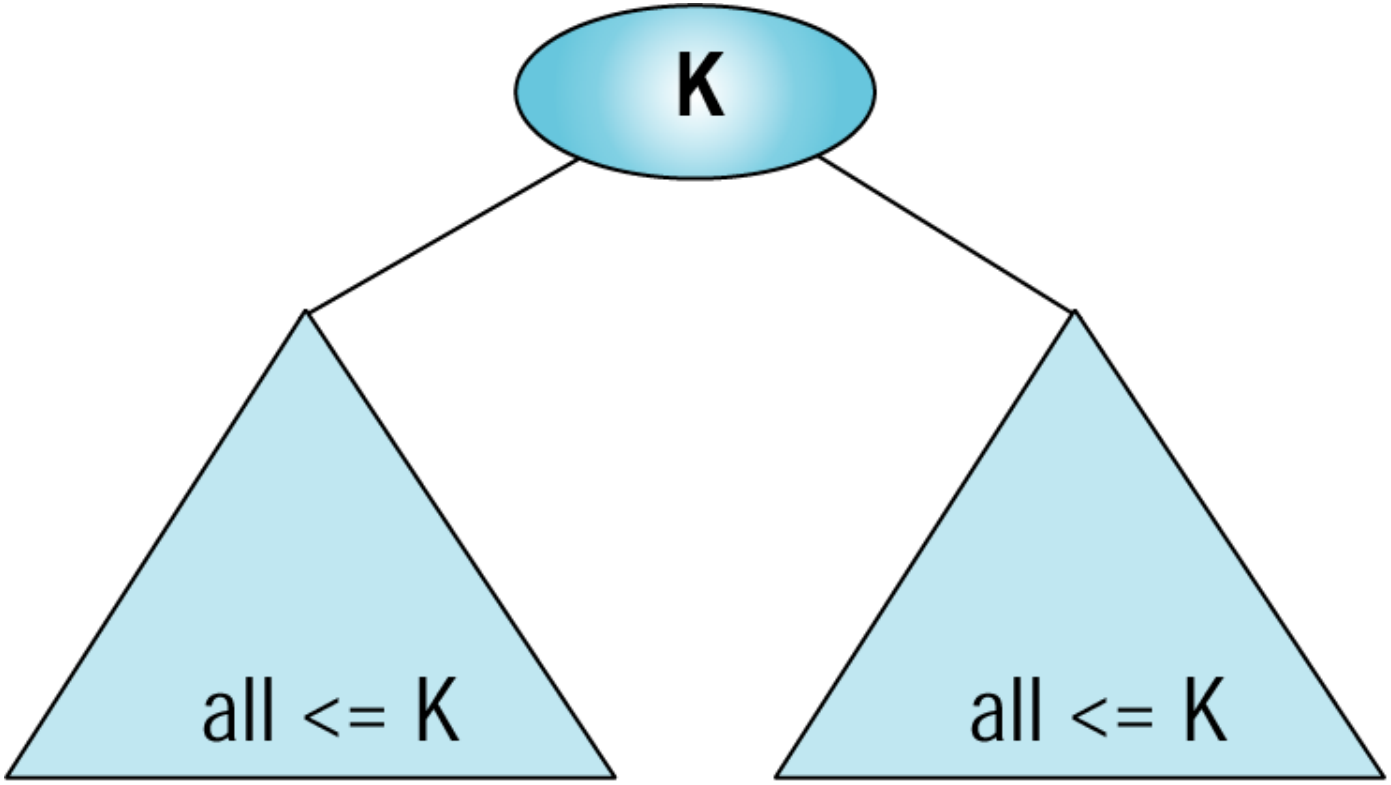
- **Heap** şu kurallara sahip bir ağaç yapısıdır:

1. Ağaç «complete» veya «nearly complete» dir
2. Her bir düğümün anahtar değeri, soyundan olan her bir düğümün anahtar değerinden büyük veya ona eşittir.

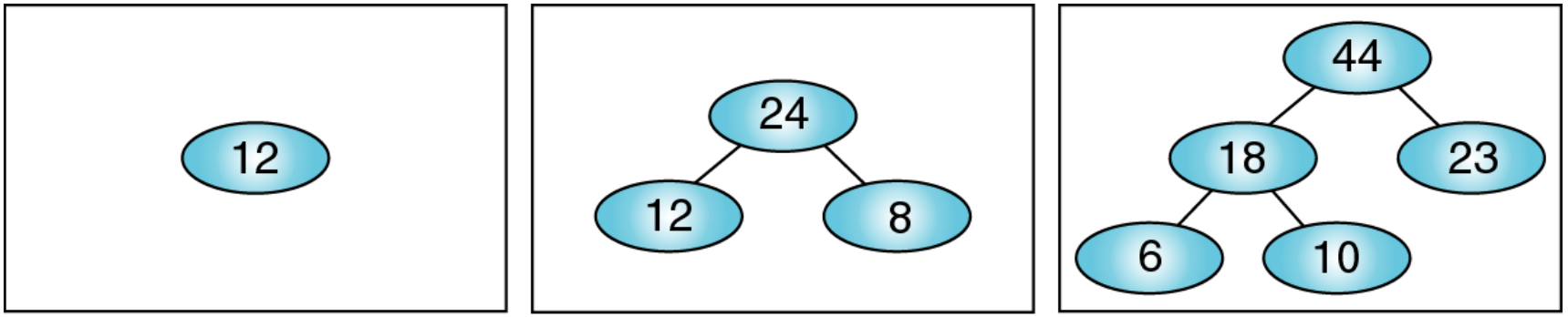
- **Heap** genellikle bağlantılı bir liste yerine bir dizi kullanarak gerçekleştirilir.
  - Diziler kullanıldığında, sol ve sağ alt ağaçların konumunu hesaplayabiliriz.
  - Veya tam tersi ebeveynin adresini hesaplayabiliriz.

Heaps

Heap, ağacın en büyük düğümünün kökte tutulmasını garanti altına alır. Daha küçük düğümler ağacın sol veya sağ alt ağaçlarına yerleştirilir.



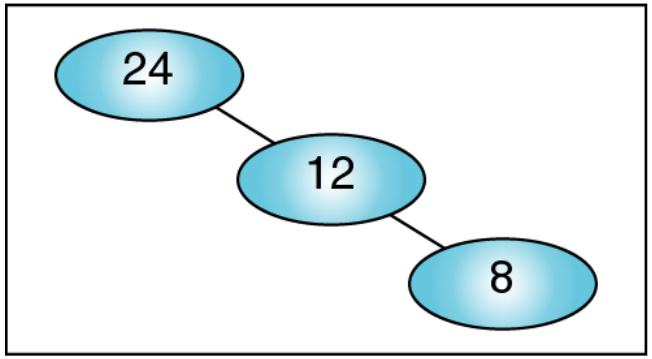
Heaps



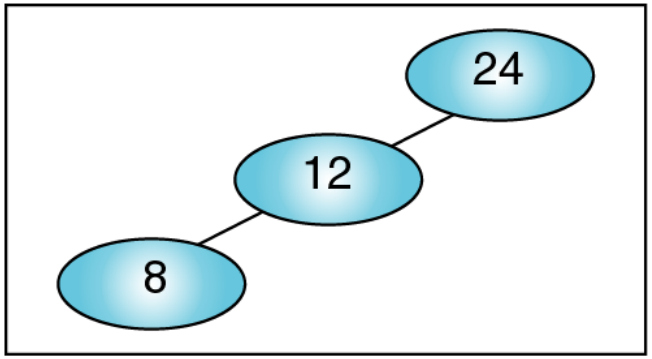
(a) Root-only heap (b) Two-level heap (c) Three-level heap

Heap Trees (*Kümeleme Ağaçları*)

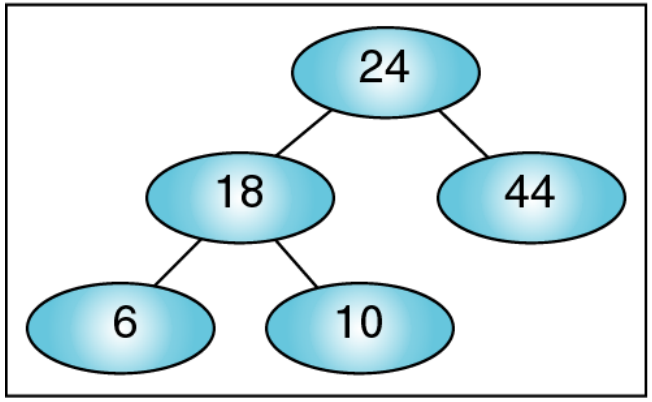
Heaps



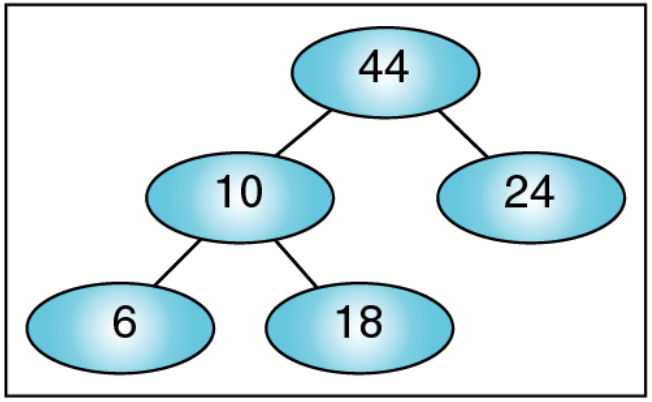
(a) Not nearly complete (rule 1)



(b) Not nearly complete (rule 1)

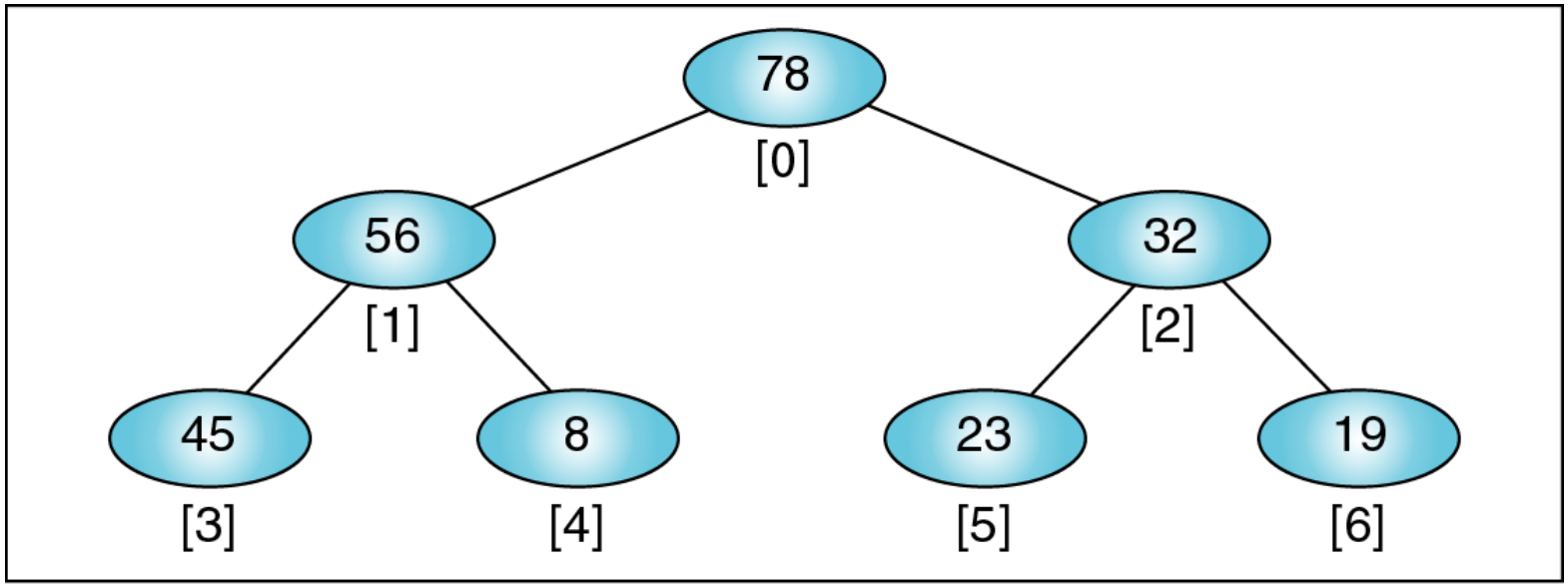


(c) Root not largest (rule 2)

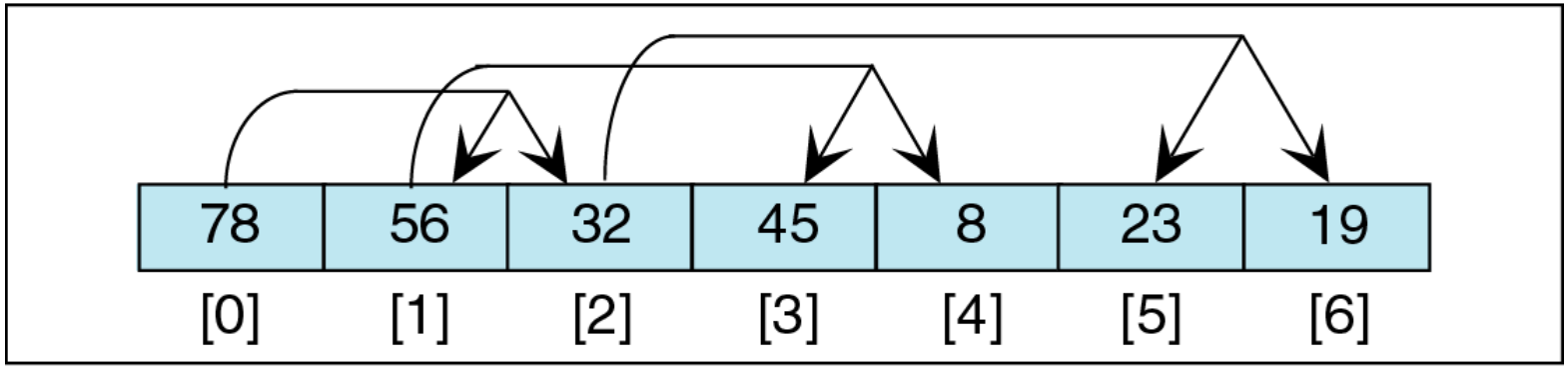


(d) Subtree 10 not a heap (rule 2)

Invalid Heaps



(a) A heap in its logical form

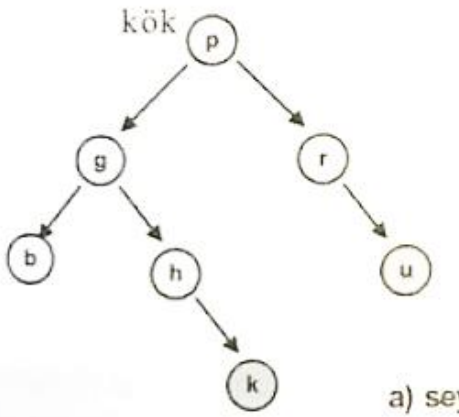


(b) A heap in an array

Figure 9-8

Dizi (indis bağıntısı) yönteminin olumsuz tarafı

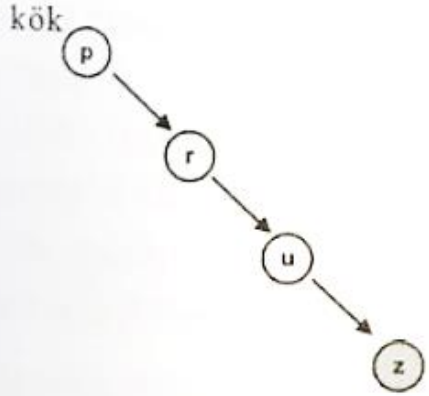
Bu yöntem seyrek özellikte ağaçlar için fazla bellek harcar.



	0	1	2	3	4	5	6	7	8	9	10
A	p	g	r	b	h	-	u	-	-	-	k

4 tane yer boş kaldı!

a) seyrek bir ikili ağaç için



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
A	p	-	r	-	-	-	v	-	-	-	-	-	-	-	z

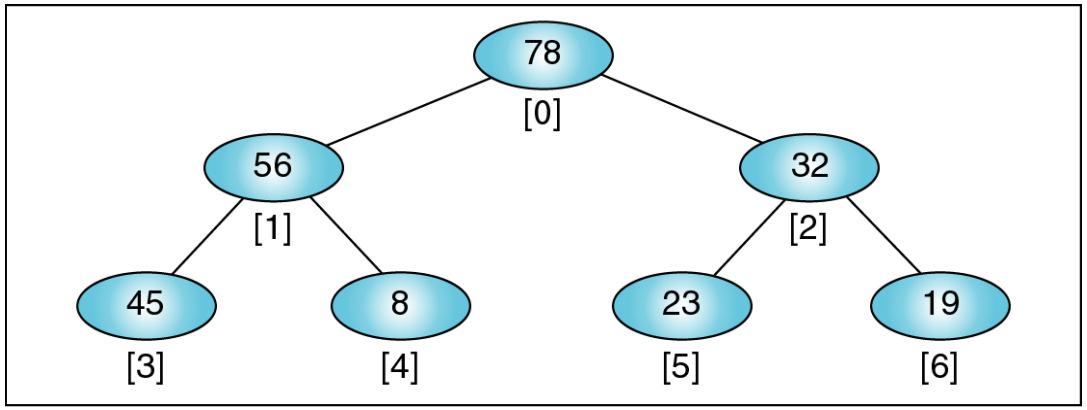
11 tane yer boş kaldı!

a) bağlantılı liste gibi bir ikili ağaç için

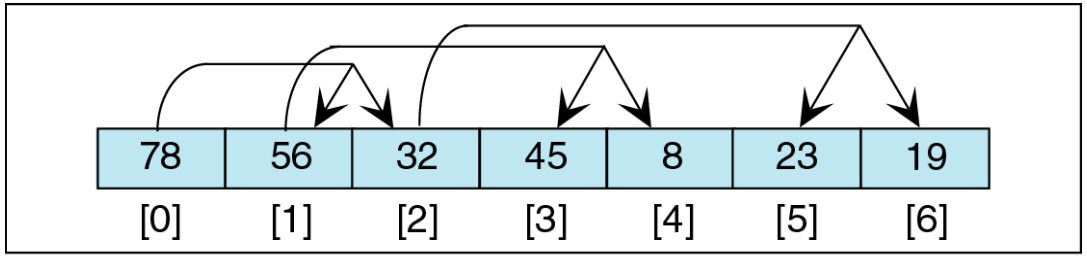
İndis bağıntısı yönteminin belleği boşa harcaması durumu

Heap Data Structure

- Genellikle diziyle gerçekleştirilir.
  - Mümkündür çünkü heap, ya «**complete**» ya da «**nearly complete**» dir.
- Dolayısıyla, bir düğüm ve çocukları arasındaki ilişki sabittir ve aşağıda gösterildiği gibi hesaplanabilir:



(a) A heap in its logical form

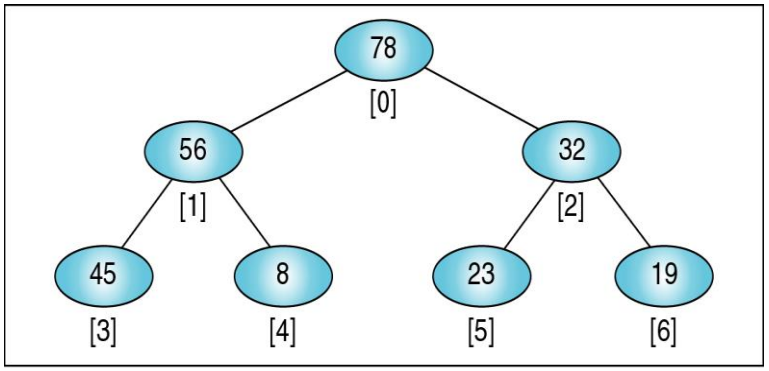


(b) A heap in an array

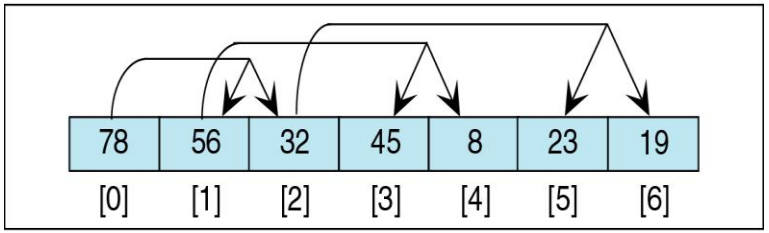
Heap Data Structure

İndeks *i* lokasyonundaki bir düğüm için,

- çocukları şu lokasyonlarda:
  - ***Left child**: 2i + 1*
  - ***Right child**: 2i + 2*
- ebeveyni, **parent:** *[(i – 1)/2]* konumunda
- *j* indeksindeki bir sol çocuk verildiğinde;
  - onun sağ kardeşi, **right sibling:** *(j + 1)*
- *k* indeksindeki bir sağ çocuk verildiğinde;
  - onun sol kardeşi, **left sibling:** (k – 1)
- Boyutu *n* olan bir eksiksiz (complete) heap verildiğinde, ilk yaprağın konumu: *[n/2]*.



(a) A heap in its logical form



(b) A heap in an array

Basic Heap Algorithms

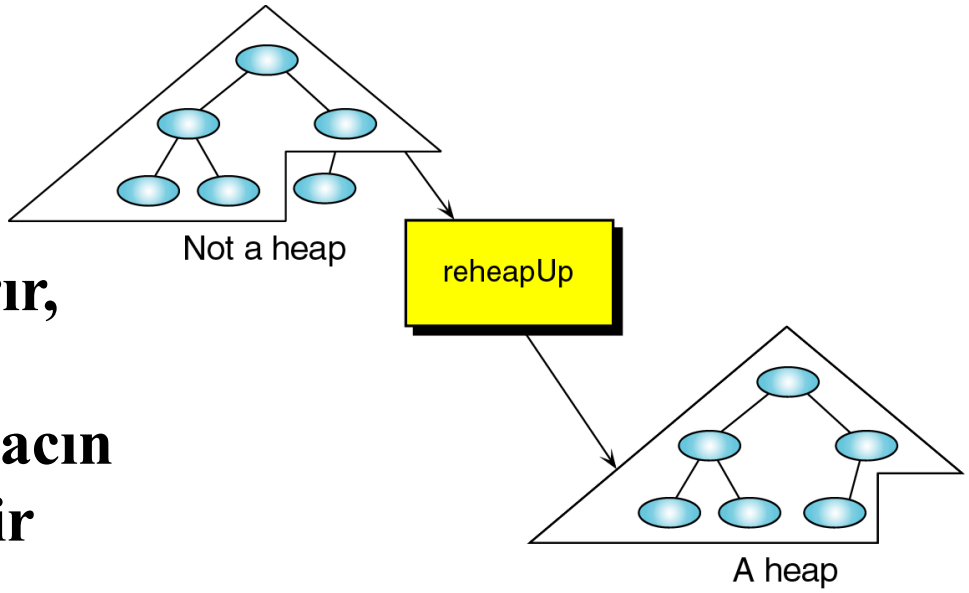
Heap üzerinde gerçekleştirilen iki temel bakım işlemi vardır:

- Düğüm ekleme (**insert**),
- Düğüm silme (**delete**).

Ekleme ve silme işlemlerini gerçekleştirmek için 2 temel algoritmaya ihtiyaç duyulur:

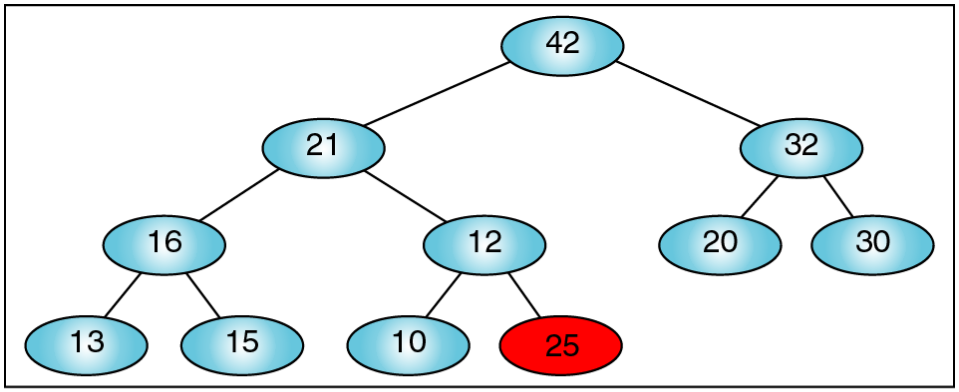
- reheap up
- reheap down

Heap Algorithms - ReheapUp

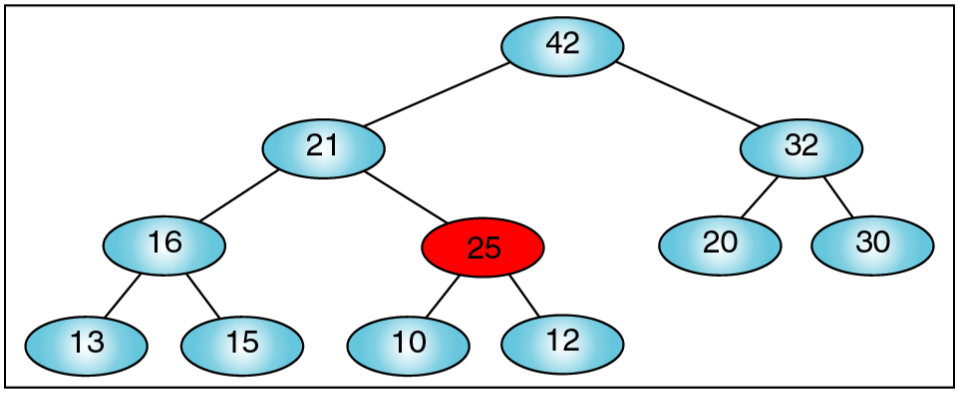


ReheapUp işlemi yapıyı onarır, böylece son eleman doğru konuma gelene kadar onu ağacın üzerinde yüzdürerek ağacı bir heap haline getirir.

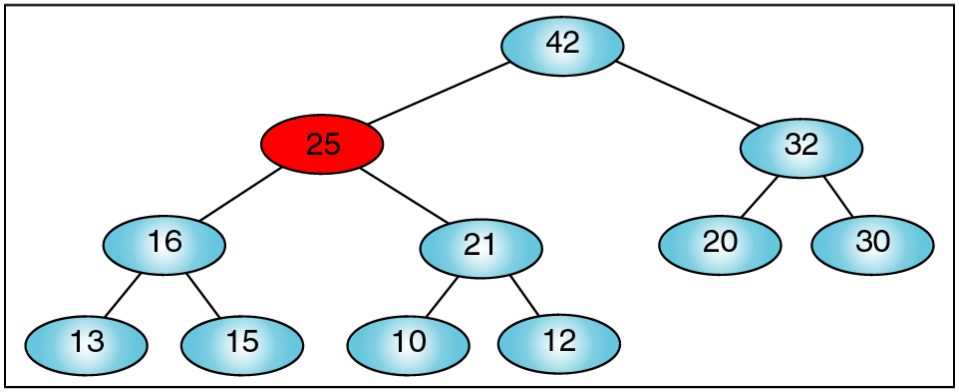
- Ekleme, ilk boş pozisyonda bir yaprakta gerçekleşir.
- Bu, yeni düğümün anahtarının ebeveyninden büyük olduğu bir durum yaratabilir.
- Bu durumda ebeveyn ve çocuk düğümlerin anahtar ve dataları değiştirilerek düğüm ağaç üzerinde yukarı doğru yüzdürülür.



(a) Original tree: not a heap



(b) Last element (25) moved up



(c) Moved up again: tree is a heap

Heap Algorithms - ReheapUp

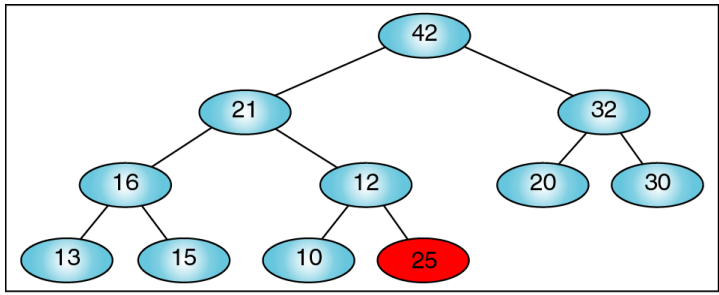
algorithm **reheapUp** (ref heap <array>, val newnode <index>)

Reestablishes heap by moving data in child up to its correct location in the heap array.

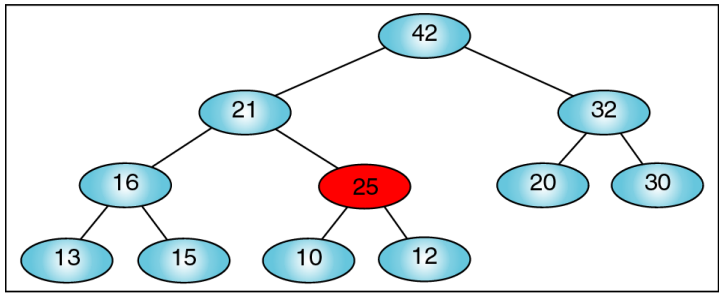
PRE heap is array containing an invalid heap.
newNode is index location to new data in heap.

POST newNode has been inserted into heap.

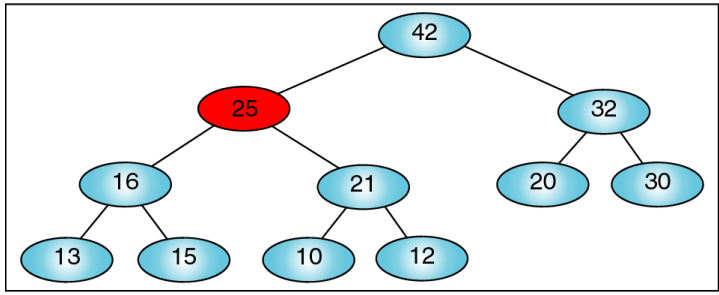
```
1 if (newNode not zero) //if (newNode not the root)
    1 parent = (newNode – 1) / 2
    2 if (heap[newNode].key > heap[parent].key)
        1 swap(newNode, parent)
        2 reheapUp(heap, parent)
2 return
end reheapUp
```



(a) Original tree: not a heap

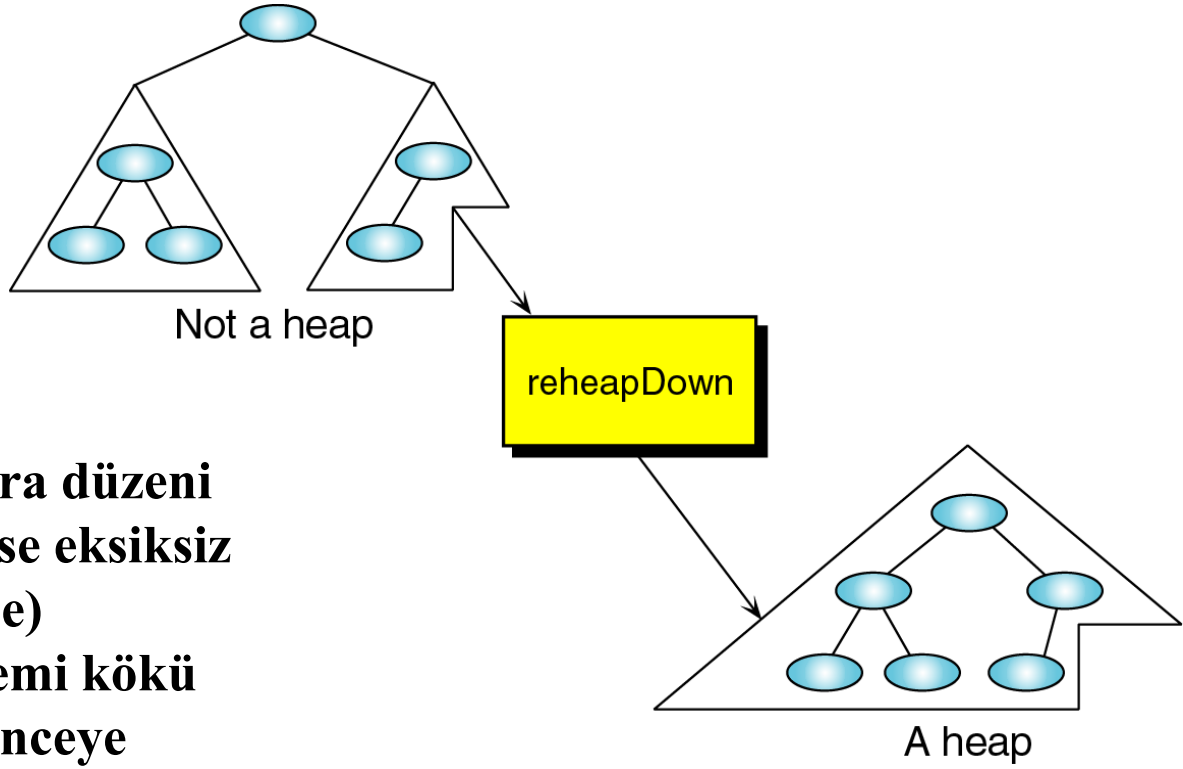


(b) Last element (25) moved up



(c) Moved up again: tree is a heap

Heap Algorithms - ReheapDown



Heap kök konumu dışında sıra düzeni özelliğini karşılayan neredeyse eksiksiz bir ağaç (nearly complete tree) olduğunda, ReheapDown işlemi kökü heap'te doğru konuma getirinceye kadar ağaçta aşağı hareket ettirerek bozuk heap'i düzene sokar.

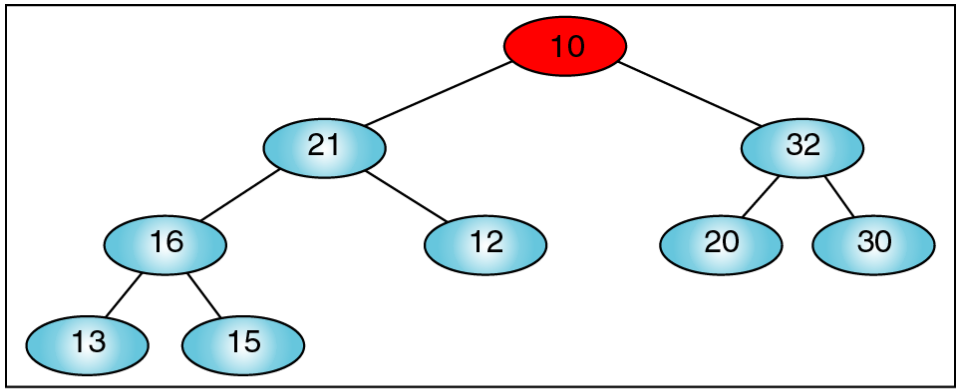
Düğümleri heap'ten aşağı iterken, geçerli girişin çocuklarından daha küçük mü (bir ya da her ikisi) belirlememiz gerekir.

Başladığımızda, kök (10) alt ağaçlardan daha küçüktür.

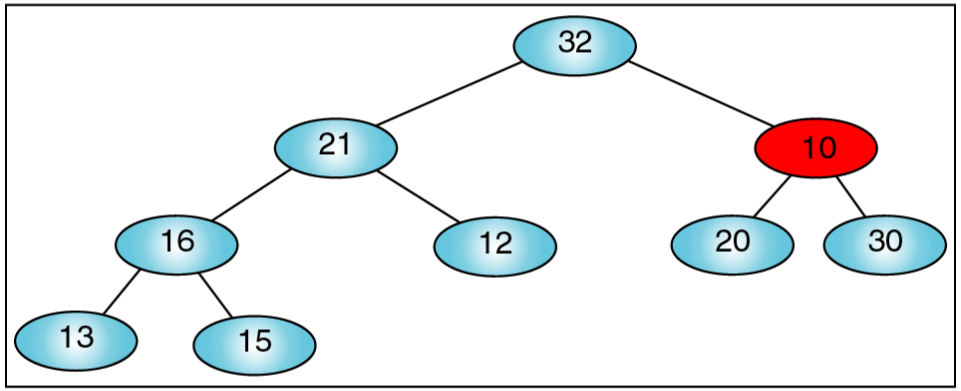
Alt ağaçları inceliyoruz ve kök ile değiş tokuş yapmak için ikisinin büyüğünü seçiyoruz, bu durumda 32.

Şekil (b) 'de değiş tokuş yaptıktan sonra, bitip bitmediğini görmek için alt ağaçlara bakarız ve 10'un alt ağaçlarının anahtarlarından daha küçük olduğunu görürüz.

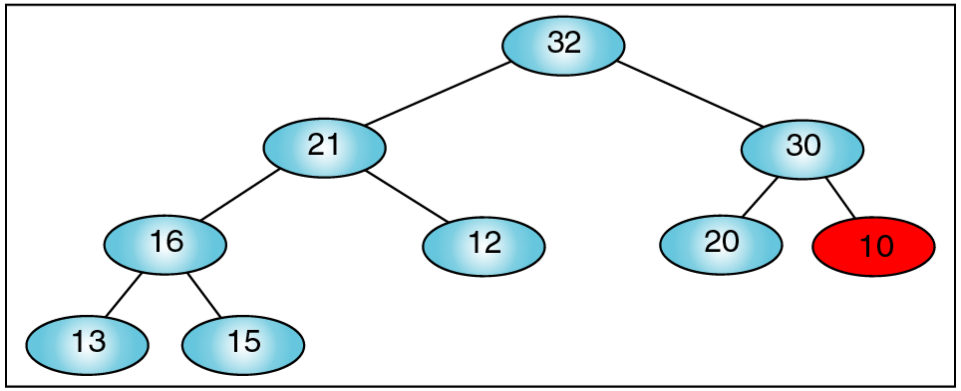
Bir kez daha, alt ağaçlardan daha büyük olanı, 10'la değiş-tokuş ediyoruz, 30. Bu noktada bir yaprağa ulaştık ve işlem bitti.



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

Heap Algorithms - ReheapDown

algorithm **reheapDown** (ref heap <array>, val root <index>, val last <index>)

Reestablishes heap by moving data in root down to its correct location in the heap array.

PRE heap is an array data.

root is root of heap or subheap.

last is an index to the last element in heap.

POST heap has been restored.

Heap Algorithms - ReheapDown

algorithm **reheapDown** (ref heap <array>, val root <index>,
val last <index>)

Determine which child has larger key.

1 if (root *2 + 1 <= last)

There is at least one child.

1 leftKey = heap[root * 2 +1].data.key

2 rightkey = heap[root *2 + 2].data.key

3 if (leftKey > rightKey)

1 largeChildKey = leftKey

2 largeChildIndex = root * 2 + 1

4 else

1 largeChildKey = rightKey

2 largeChildIndex = root * 2 + 2

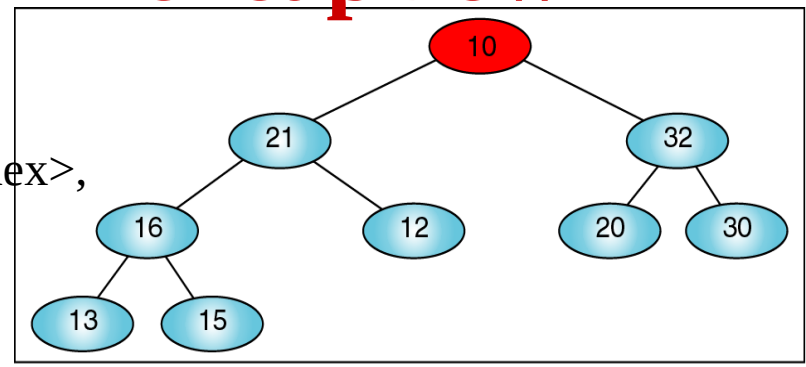
Test if root > larger subtree.

5 if (heap[root].data.key < heap[largeChildIndex].data.key)

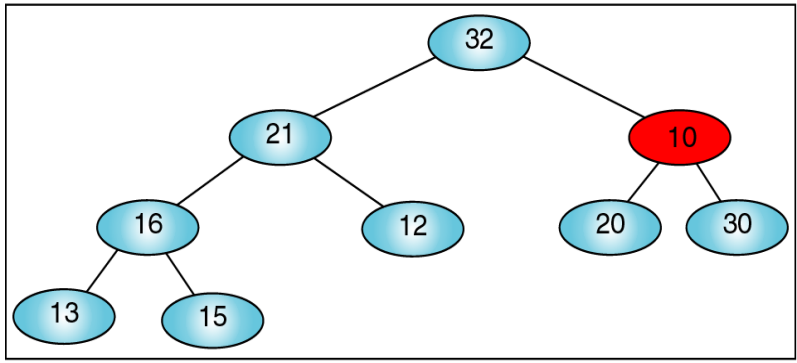
1 swap(root, largeChildIndex)

2 **reheapDown**(heap, largeChildIndex, last)

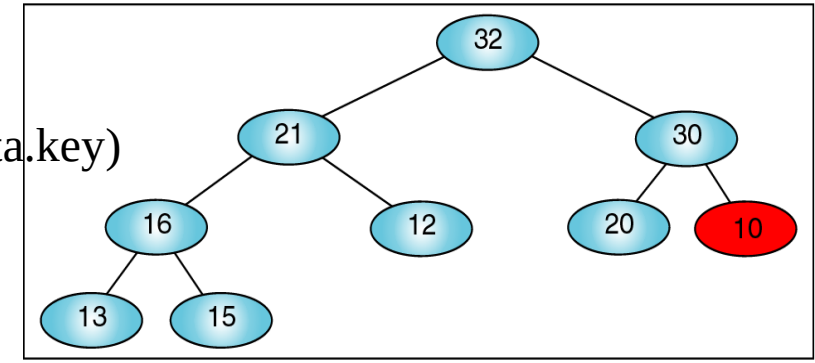
end **reheapdown**



(a) Original tree: not a heap



(b) Root moved down (right)



(c) Moved down again: tree is a heap

Heap Algorithms - Build Heap

Heap build algoritması oldukça basittir.

Heap'e dönüştürülmesi istenen bir dizi verildiğinde,

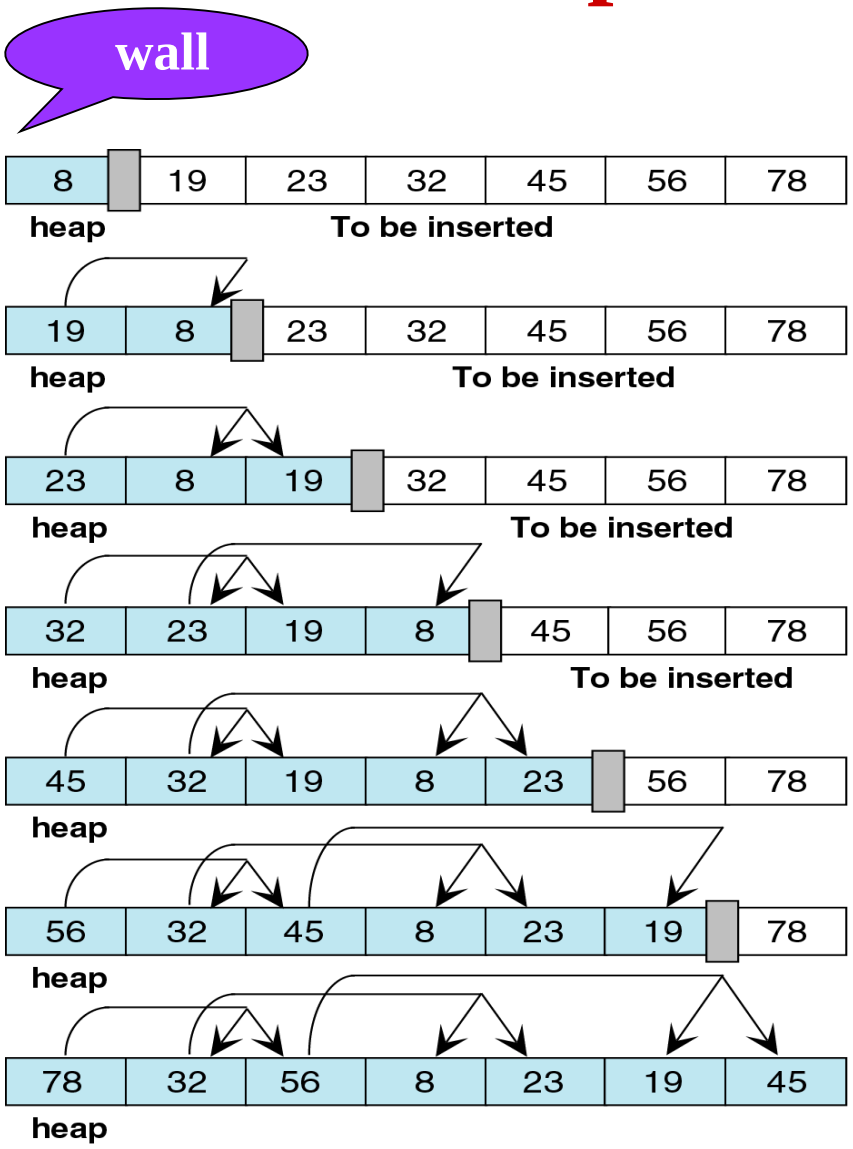
- ikinci elemandan başlayarak ve
- her seferinde eklenecek eleman için reheapUp fonksiyonunu çağırarak

bu dizi üzerinden geçmek gerekir.

algorithm **buildHeap**(ref heap <array>,
val size <integer>)

heap: heap düzeninde olmayan verileri tutan dizi
size: dizideki eleman sayısı

```
1 walker = 1
2 loop (walker < size)
    1 reheapUp(heap, walker)
    2 walker = walker + 1
3 return
end buildHeap
```



Heap Algorithms - Insert Heap

•Bir düğüm eklemek için dizideki ilk boş yaprağı bulmamız gerekir.

•Yeni veriyi ilk boş yaprağa taşır ve ardından reheapUp işlemi yaparız.

algorithm **insertHeap**(ref heap <array of dataType>,
ref last <index>,
ref data <dataType>)

```
1 if (heap full)
        1 return false
2 last=last + 1
3 heap[last] = data
4 reheapUp(heap, last)
5 return true
end insertHeap
```

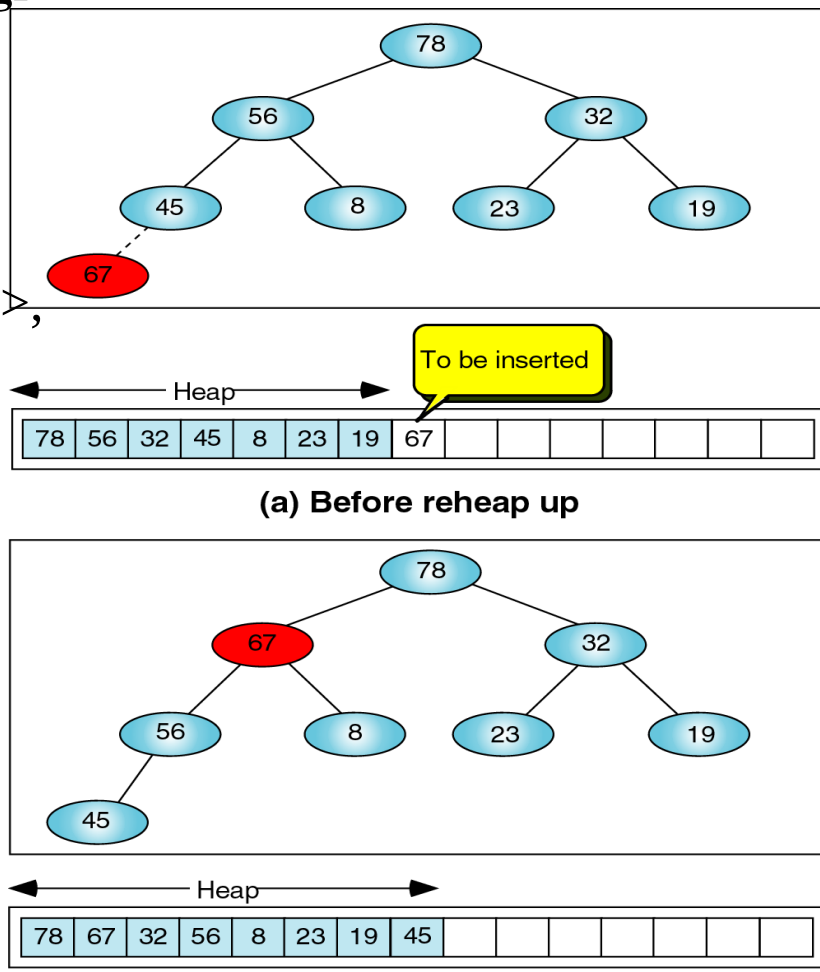
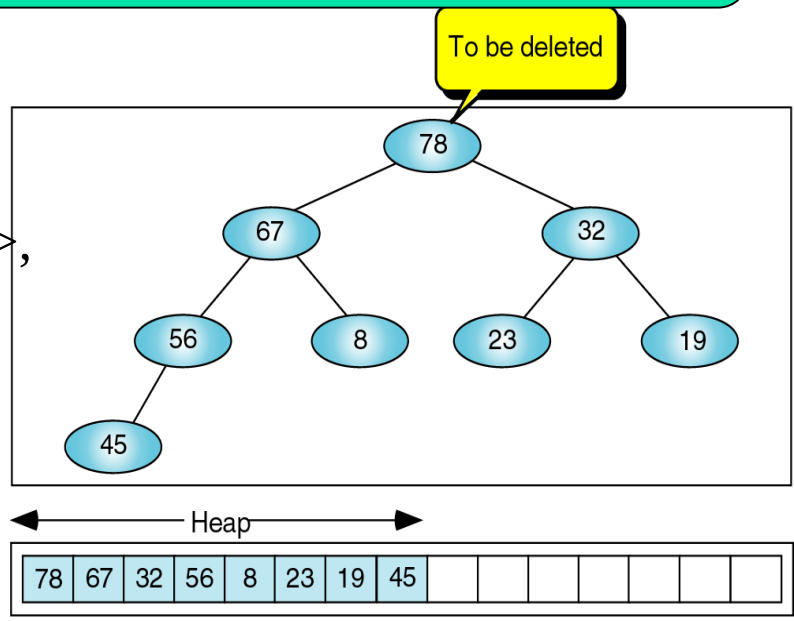


Figure 9-10

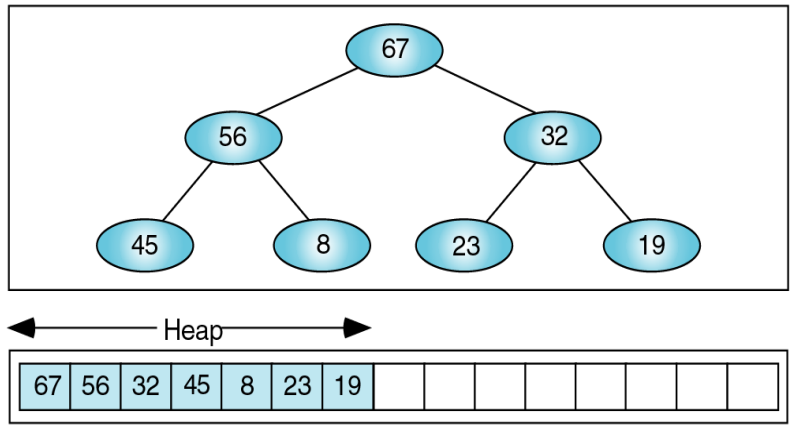
Heap Algorithms - Delete Heap

Bir düğümü heap'ten silerken, en yaygın ve anlamlı mantık kökü silmektir.
Aslında, bir heap'in gerekçesi/mantığı, en büyük elemanı, yani kökü belirlemek/seçip çıkarmaktır.

Silme işleminin ardından heap köksüz kalır.
Heap'i yeniden kurmak için, son yığın düğümündeki veriyi köke taşır ve ardından reheapDown işlemi yaparız.



(a) Before delete



(b) After delete

algorithm **deleteHeap**(ref heap <array of dataType>,
ref last <index>,
ref dataOut <dataType>)

Deletes root of heap and passes data back to caller.
Root has been deleted from heap and root data
placed in dataOut.

```
1 if (heap empty)
        1 return false
2 dataOut= heap[0]
3 heap[0] = heap[last]
4 last = last - 1
5 reheapDown(heap, 0, last)
6 return true
end deleteHeap
```

Implementation of Heap ADT

- Heap Structure
- Heap Algorithms

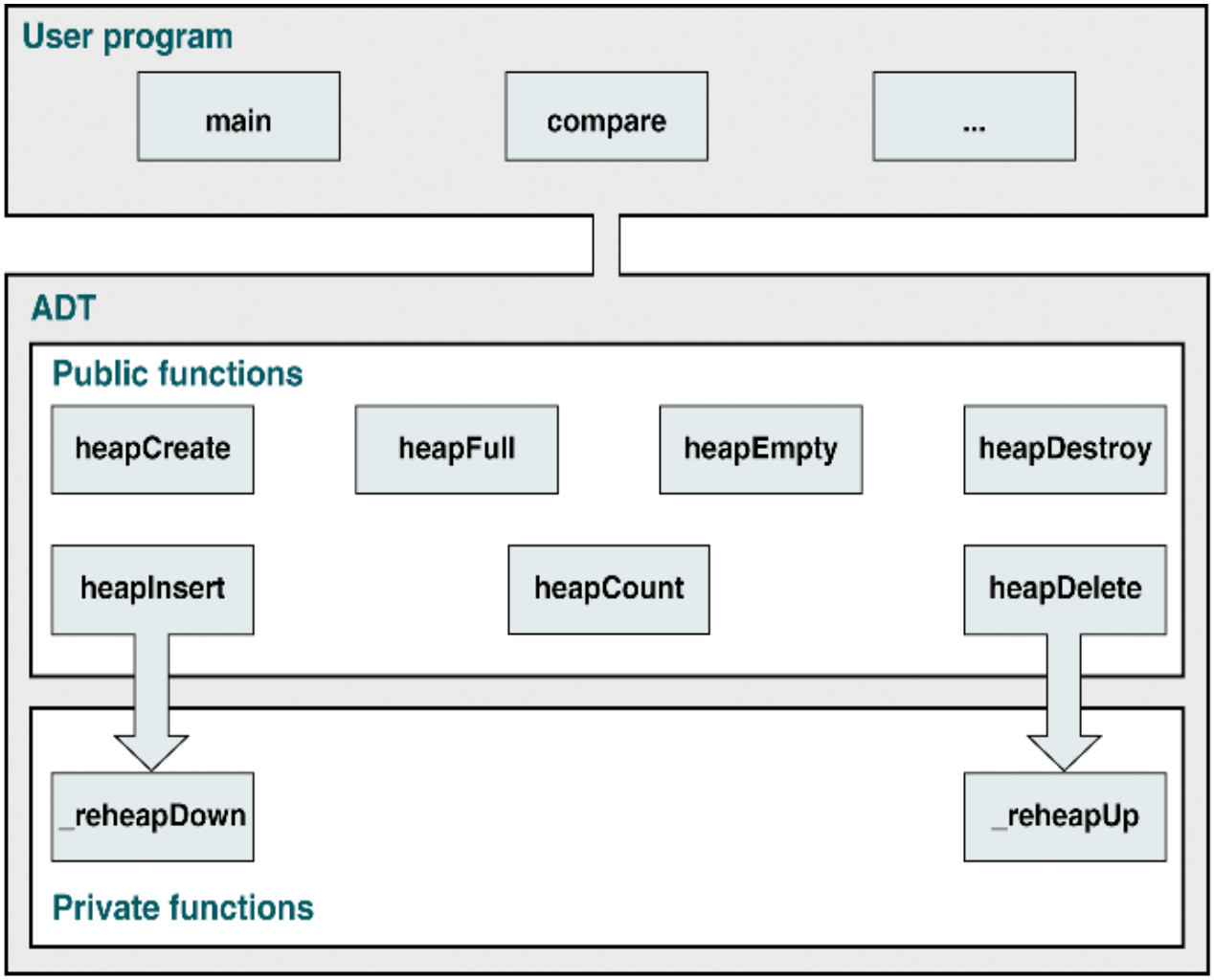
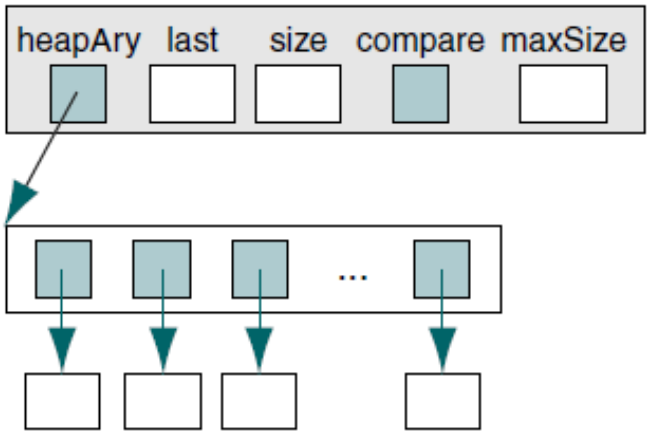


FIGURE 9-12 Heap ADT Design




```
typedef struct
{
   void** heapAry;
   int    last;
   int    size;
   int   (*compare)(void* argu1, void* argu2);
   int    maxSize;
} HEAP;
```

Heap ADT Structure

PROGRAM 9-1 Heap Declaration *(continued)*

```
 5  #include <stdbool.h>
 6
 7  typedef struct
 8  {
 9     void** heapAry;
10     int    last;
11     int    size;
12     int    (*compare) (void* argu1, void* argu2);
13     int    maxSize;
14  } HEAP;
15
16  // Prototype Definitions
17  HEAP* heapCreate (int maxSize,
18                    int (*compare) (void* arg1, void* arg2));
19  bool  heapInsert  (HEAP* heap, void*  dataPtr);
20  bool  heapDelete  (HEAP* heap, void** dataOutPtr);
21  int   heapCount   (HEAP* heap);
22  bool  heapFull    (HEAP* heap);
23  bool  heapEmpty   (HEAP* heap);
24  void  heapDestroy (HEAP* heap);
25
26  static void _reheapUp   (HEAP* heap, int childLoc);
27  static void _reheapDown (HEAP* heap, int root);
```

PROGRAM 9-2 Create Heap Application Interface

```
/* ==================== heapCreate ====================
   Allocates memory for heap and returns address of
   heap head structure.
      Pre  Nothing
      Post heap created and address returned
                if memory overflow, NULL returned
*/
#include <math.h>

HEAP* heapCreate (int maxSize,
              int  (*compare) (void* argu1, void* argu2))
{
// Local Definitions
   HEAP* heap;

// Statements
   heap = (HEAP*)malloc(sizeof (HEAP));
   if (!heap)
      return NULL;

   heap->last    = -1;
   heap->compare = compare;

   // Force heap size to power of 2 -1
   heap->maxSize =
            (int) pow (2, ceil(log2(maxSize))) - 1;
   heap->heapAry = (void*)
                calloc(heap->maxSize, sizeof(void*));
   return heap;
}  // createHeap
```

PROGRAM 9-3 Insert Heap Application Interface

```
/* ================== heapInsert ====================
   Inserts data into heap.
      Pre    Heap is a valid heap structure
             last is pointer to index for last element
             data is data to be inserted
      Post   data have been inserted into heap
      Return true if successful; false if array full
*/
bool heapInsert (HEAP* heap, void* dataPtr)
{
// Statements
   if (heap->size == 0)                    // Heap empty
      {
       heap->size                 = 1;
       heap->last                 = 0;
       heap->heapAry[heap->last] = dataPtr;
       return true;
      } // if
   if (heap->last == heap->maxSize - 1)
     return false;
   ++(heap->last);
   ++(heap->size);
   heap->heapAry[heap->last] = dataPtr;
   _reheapUp (heap, heap->last);
   return true;
} // heapInsert
```

PROGRAM 9-4 Internal Reheap Up Function

```
1  /* ==================== reheapUp ====================
2     Reestablishes heap by moving data in child up to
3     correct location heap array.
4        Pre  heap is array containing an invalid heap
5             newNode is index to new data in heap
6        Post newNode inserted into heap
7  */
8  void  _reheapUp  (HEAP* heap, int childLoc)
9  {
10 // Local Definitions
11    int    parent;
12    void** heapAry;
13    void*  hold;
14
15 // Statements
16    // if not at root of heap -- index 0
17    if (childLoc)
18       {
19        heapAry = heap->heapAry;
20        parent = (childLoc - 1)/ 2;
21        if (heap->compare(heapAry[childLoc],
22                          heapAry[parent]) > 0)
23            // child is greater than parent -- swap
24            {
25             hold              = heapAry[parent];
26             heapAry[parent]   = heapAry[childLoc];
27             heapAry[childLoc] = hold;
28            _reheapUp (heap, parent);
29            } // if heap[]
30       } // if newNode
31    return;
32 }  // reheapUp
```

PROGRAM 9-5 Delete Heap Application Interface

```
1  /*  ================== heapDelete ===================
2     Deletes root of heap and passes data back to caller.
3        Pre    heap is a valid heap structure
4               last is reference to last node in heap
5               dataOut is reference to output area
6        Post   last deleted and heap rebuilt
7               deleted data passed back to user
8        Return true if successful; false if array empty
9  */
10 bool heapDelete (HEAP* heap, void** dataOutPtr)
11 {
12 // Statements
13    if (heap->size == 0)
14        // heap empty
15        return false;
16    *dataOutPtr = heap->heapAry[0];
17    heap->heapAry[0]  = heap->heapAry[heap->last];
18    (heap->last)--;
19    (heap->size)--;
20    _reheapDown (heap, 0);
21    return true;
22 }  // heapDelete
```

PROGRAM 9-6 Internal Reheap Down Function

```
1  /* ================== reheapDown ==================
2     Reestablishes heap by moving data in root down to its
3     correct location in the heap.
4        Pre  heap is array of data
5             root is root of heap or subheap
6             last is an index to last element in heap
7        Post heap has been restored
8  */
9  void _reheapDown (HEAP* heap, int root)
10 {
11 // Local Definitions
12    void* hold;
13    void* leftData;
14    void* rightData;
15    int   largeLoc;
```

PROGRAM 9-6 Internal Reheap Down Function *(continued)*

```
16      int   last;
17
18  // Statements
19      last = heap->last;
20      if ((root * 2 + 1) <= last)          // left subtree
21          // There is at least one child
22         {
23          leftData   = heap->heapAry[root * 2 + 1];
24          if ((root * 2 + 2) <= last)  // right subtree
25             rightData = heap->heapAry[root * 2 + 2];
26          else
27             rightData = NULL;
28
29          // Determine which child is larger
30          if ((!rightData)
31                || heap->compare (leftData, rightData) > 0)
32             {
33              largeLoc = root * 2 + 1;
34             } // if no right key or leftKey greater
35          else
36             {
37              largeLoc = root * 2 + 2;
38             } // else
39          // Test if root > larger subtree
40          if (heap->compare (heap->heapAry[root],
41              heap->heapAry[largeLoc]) < 0)
42              {
43               // parent < children
44               hold = heap->heapAry[root];
45               heap->heapAry[root] =
46                     heap->heapAry[largeLoc];
47               heap->heapAry[largeLoc] = hold;
48               _reheapDown (heap, largeLoc);
49              } // if root <
50      } // if root
51      return;
52  }  // reheapDown
```

Eğer sağ çocuk yoksa veya sol çocuk, sağ çocuktan büyükse

Heap Applications

- Common applications of Heaps are;

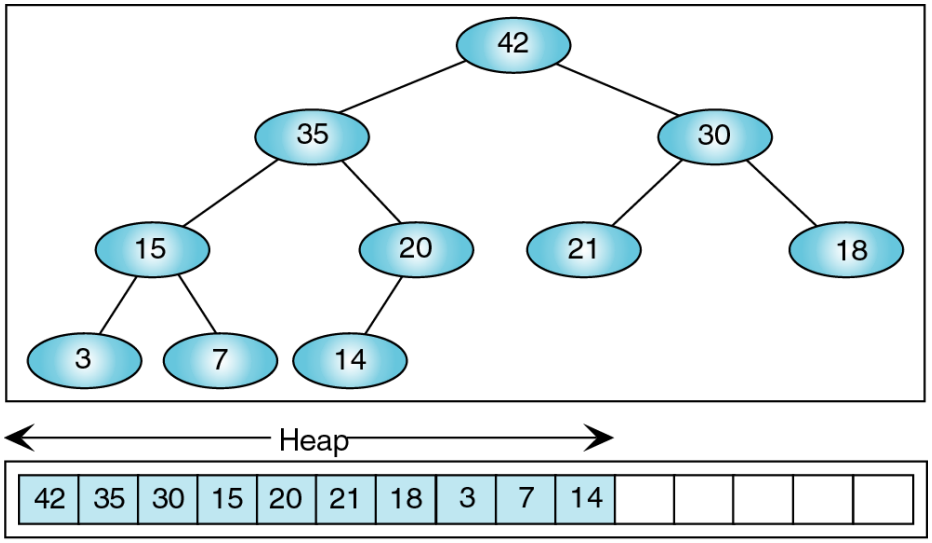
1. Selection algorithms,
2. Priority queues
3. Sorting.

Heap Applications – **Selection Algorithms**

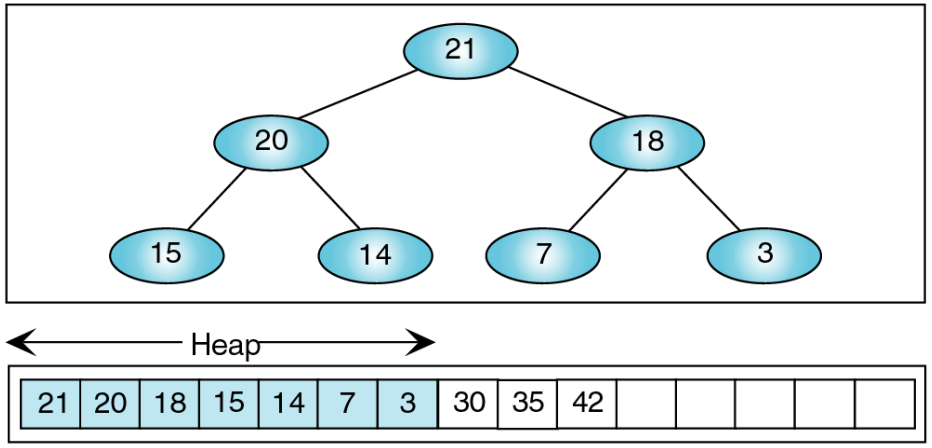
- Sıralanmamış listedeki *k*. elemanı belirlemek için iki çözüm vardır:
  1. Önce listeyi sıralayın ve öğeyi *k*. konumdaki elemanı seçin veya
  2. **bir heap oluşturun ve bundan *k*-1 adet eleman silin**

Heap Applications – **Selection Algorithms**

Örnek: Listenin en büyük dördüncü elamanını öğrenmek istiyorsak:



(a) Original heap



(b) After three deletions

Heap Applications – **Priority Queues**

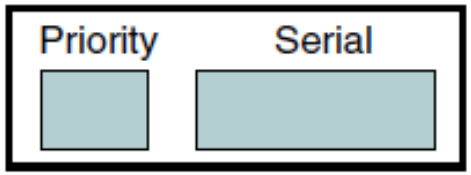
- Heap, öncelikli kuyruk için mükemmel bir yapıdır.

- Yaygın olarak kullanılan bir teknik, önceliğini + olayın kuyruktaki konumunu temsil eden bir seri numaradan oluşan bir kodlanmış **öncelik numarası** kullanmaktır.
  - **Seri numarası azalan sırada olmalıdır.**

Heap Applications – **Priority Queues**

Herhangi bir anda herhangi bir öncelik için maksimum 1000 olay olacağını varsayarsak, 1999'dan 1000'e kadar olan sıralı sayılara en düşük önceliği atayabiliriz,

Priority number



Priority	Serial	Priority	Serial	Priority	Serial
1	999	3	999	5	999
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
1	000	3	000	5	000
2	999	4	999		
.	.	.	.		
.	.	.	.		
.	.	.	.		
2	000	4	000		

Priority Queue Priority Numbers

Heap Applications – **Priority Queues**

- Bir olay kuyruğa girerken, kuyruktaki diğer olaylara göre konumunu belirleyen bir öncelik numarası atanır.
- Yeni olay (**event**) heap'e herhangi bir zamanda yalnızca bir yerde, ilk boş yaprak olarak girebilse de, bir öncelik numarası atanır.
- Bununla birlikte, sıraya girdikten sonra, bu yeni olay heap'teki diğer tüm olaylara göre hızlı bir şekilde doğru konumuna yükselir.
- En yüksek önceliğe sahipse, heap'in tepesine çıkar ve işlenecek bir sonraki olay haline gelir.
  - Düşük önceliğe sahipse heap'te daha aşağılarda kalır ve sırasının gelmesini bekler.

Priority Queues -Example

- Üç öncelikten oluşan bir öncelik sırasına sahip olduğumuzu varsayalım: yüksek (3), orta (2) ve düşük (1).

- Gelen ilk beş müşteriden ikincisi ve beşinci yüksek öncelikli müşteriler, üçüncüsü orta öncelikli ve birincisi ve dördüncüsü de düşük önceliklidir.
- Bunlar için öncelik numarası atamaları şu şekilde gerçekleştirilir.

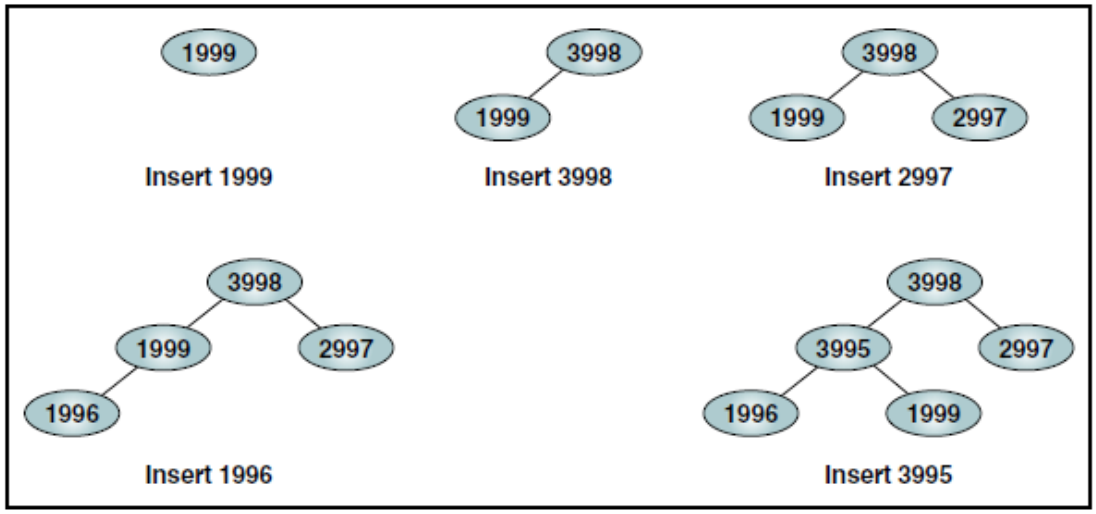
Arrival	Priority	Priority
1	low	1999 (1 & (1000 – 1)
2	high	3998 (3 & (1000 – 2)
3	medium	2997 (2 & (1000 – 3)
4	low	1996 (1 & (1000 – 4)
5	high	3995 (3 & (1000 – 5)

Priority Number Assignments

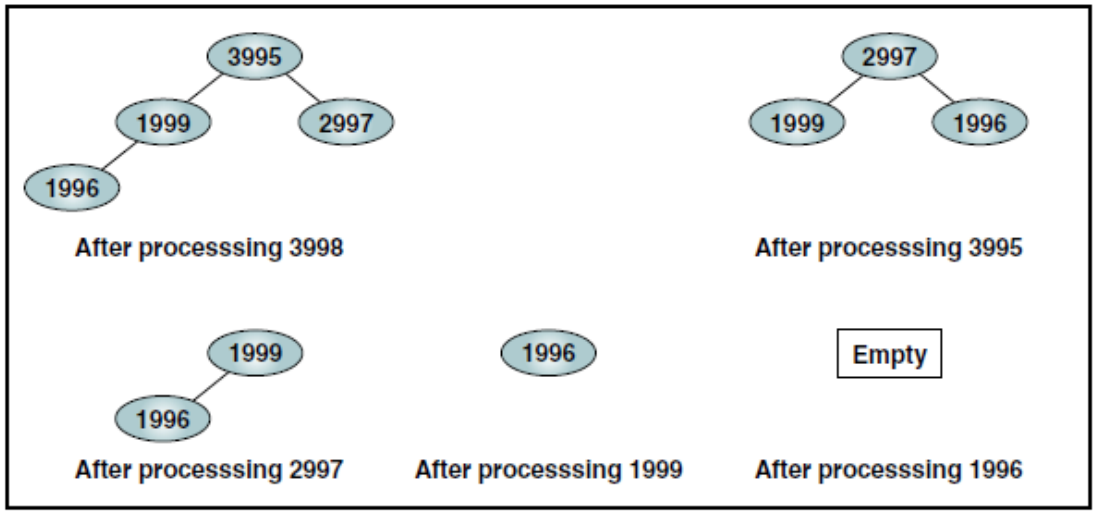
Priority Queues -Example

Arrival	Priority	Priority
1	low	1999 (1 & (1000 – 1)
2	high	3998 (3 & (1000 – 2)
3	medium	2997 (2 & (1000 – 3)
4	low	1996 (1 & (1000 – 4)
5	high	3995 (3 & (1000 – 5)

Priority Number Assignments



(a) Insert customers



(b) Process customers

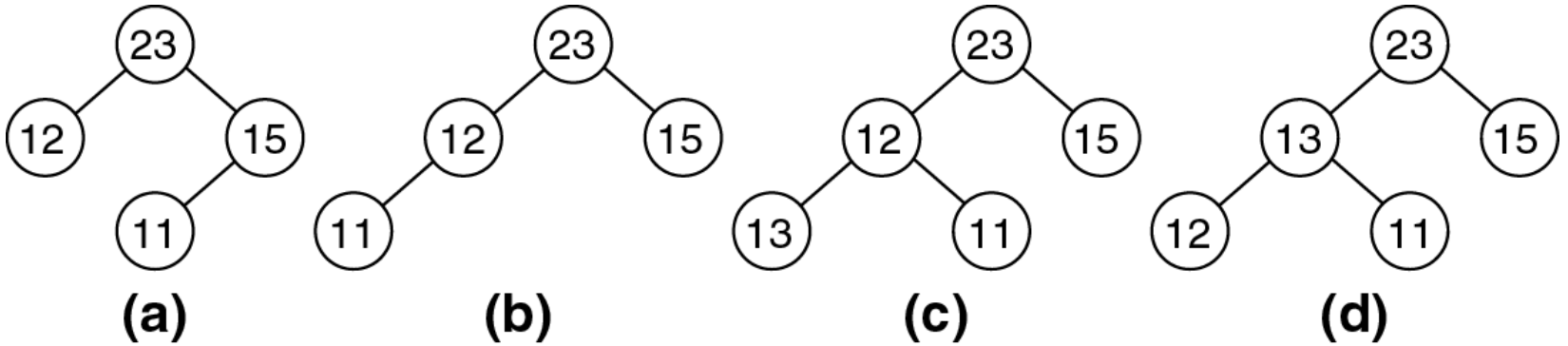
Priority Queue Example

Lab Uygulaması

Kitaptaki PROGRAM 9-7 Priority Queue Implementation örnek uygulamasında **Heap ADT**'nin kullanım biçimi incelenecek ve uygulama çalıştırılacak.

Alıştırma

Aşağıdaki yapıların hangisi heap'tır, hangisi değildir?



Heap değil

Çünkü «nearly complete» değil, yani solda boş düğüm var.

Heap

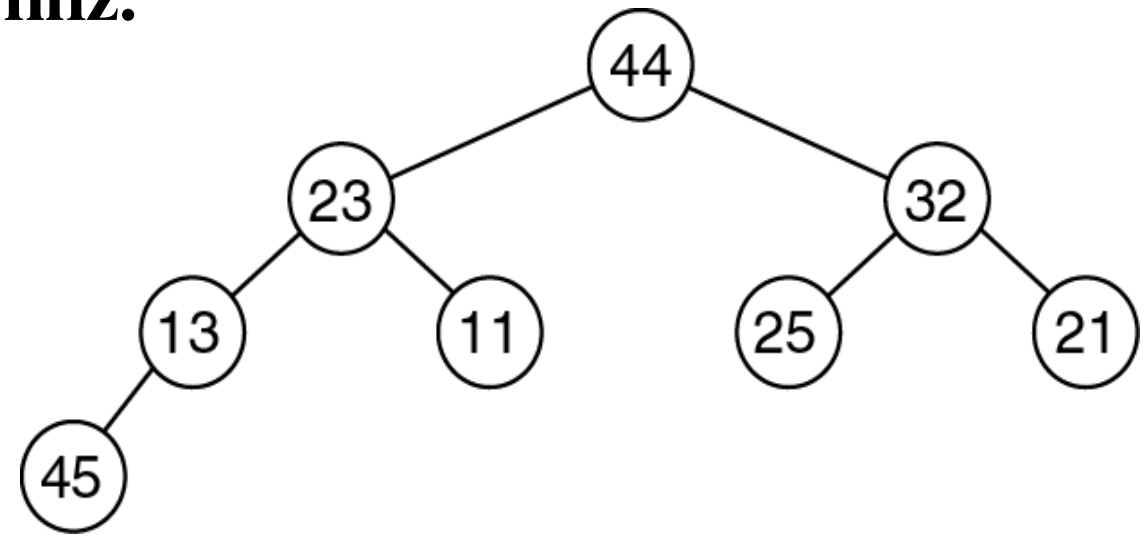
Heap değil

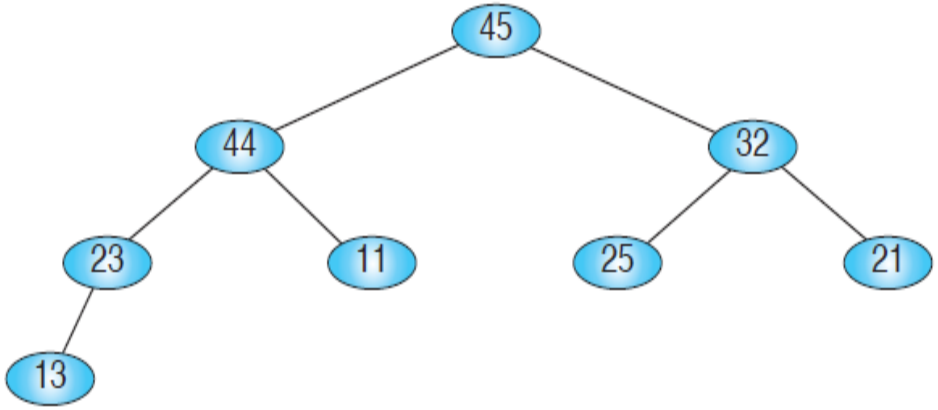
Çünkü düğüm 13 ebeveyninden (12) büyük.

Heap

Alıştırma

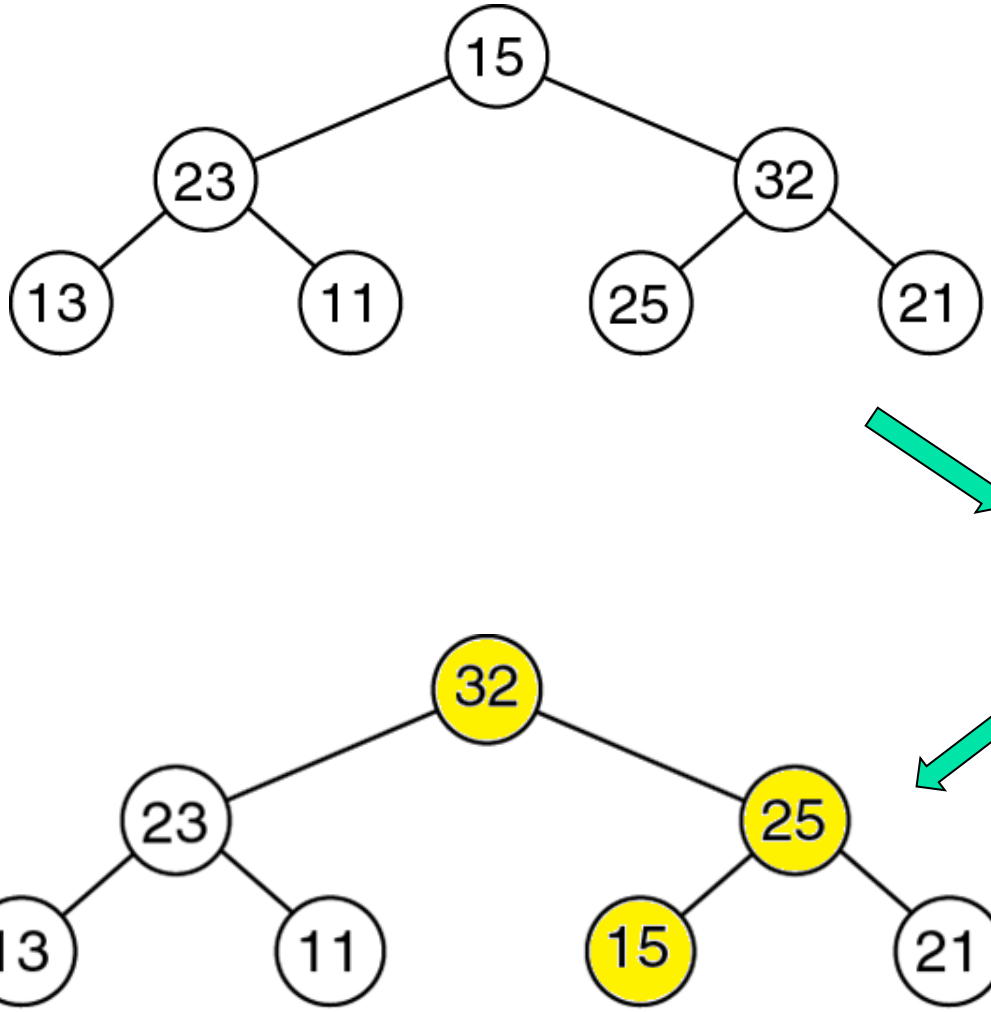
Aşağıdaki heap olmayan yapıya reheapUp algoritmasını uygulayınız.





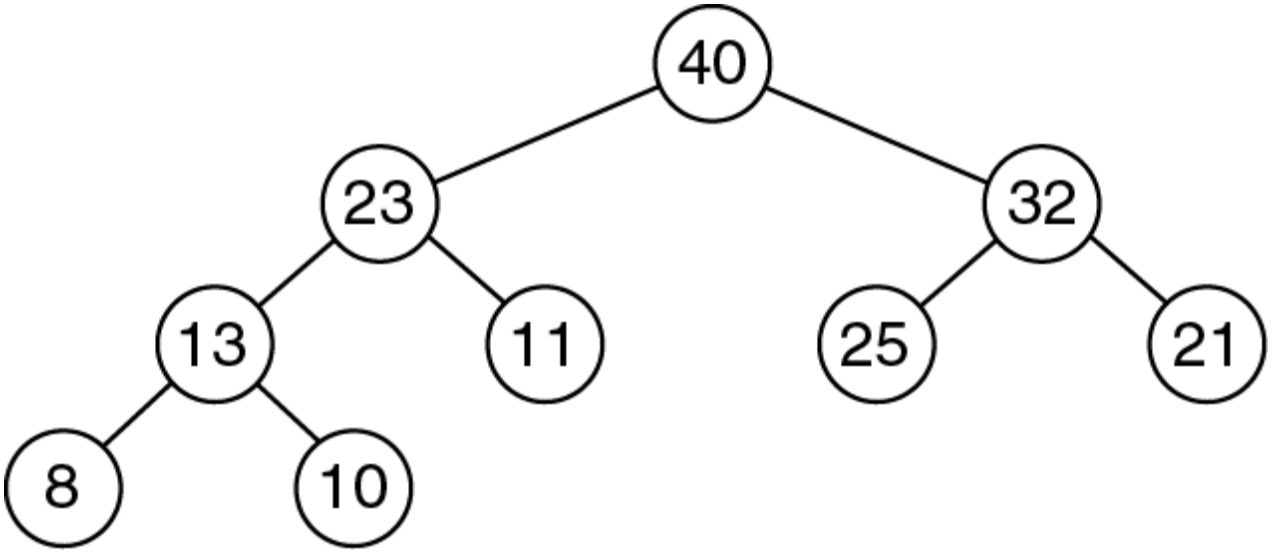
Alıştırma

Aşağıdaki heap olmayan yapıya reheapDown algoritmasını uygulayınız.



Alıştırma

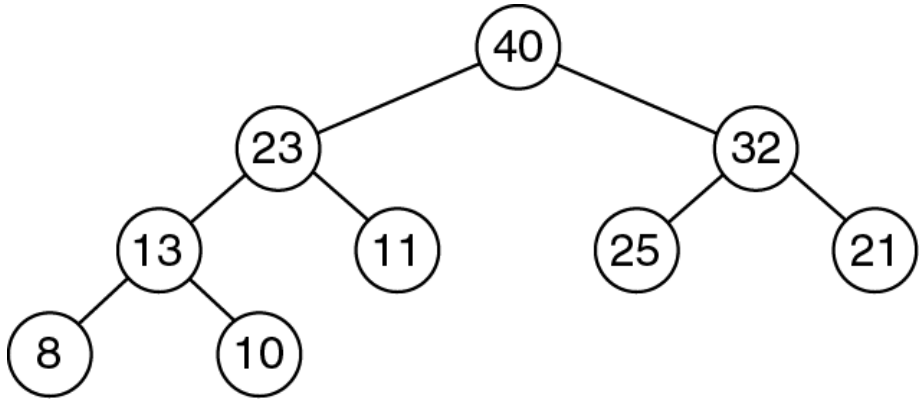
Aşağıdaki heap'in dizi implementasyonunu gösteriniz/çiziniz.



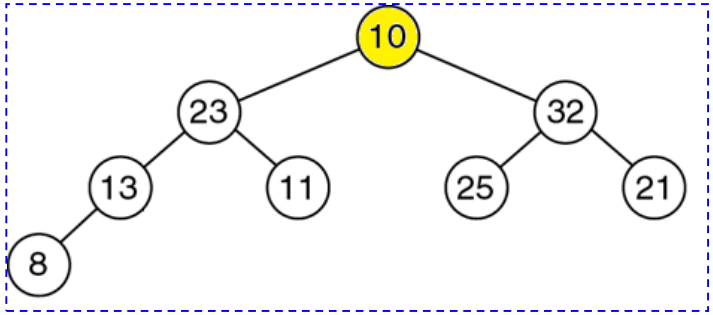
40	23	32	13	11	25	21	8	10					
[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]					

Alıştırma

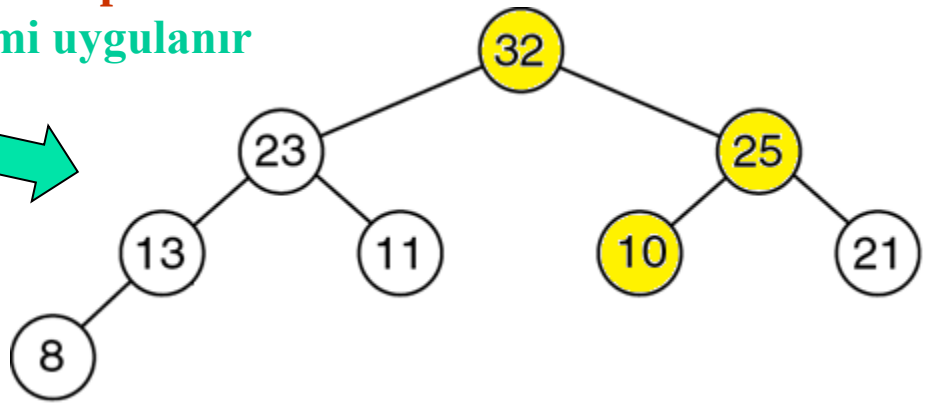
Şilme işlemi uygulayın. Silme işleminin ardından heap'ı onarın.



1) Kök düğüm silinir ve son düğüm köke taşınır

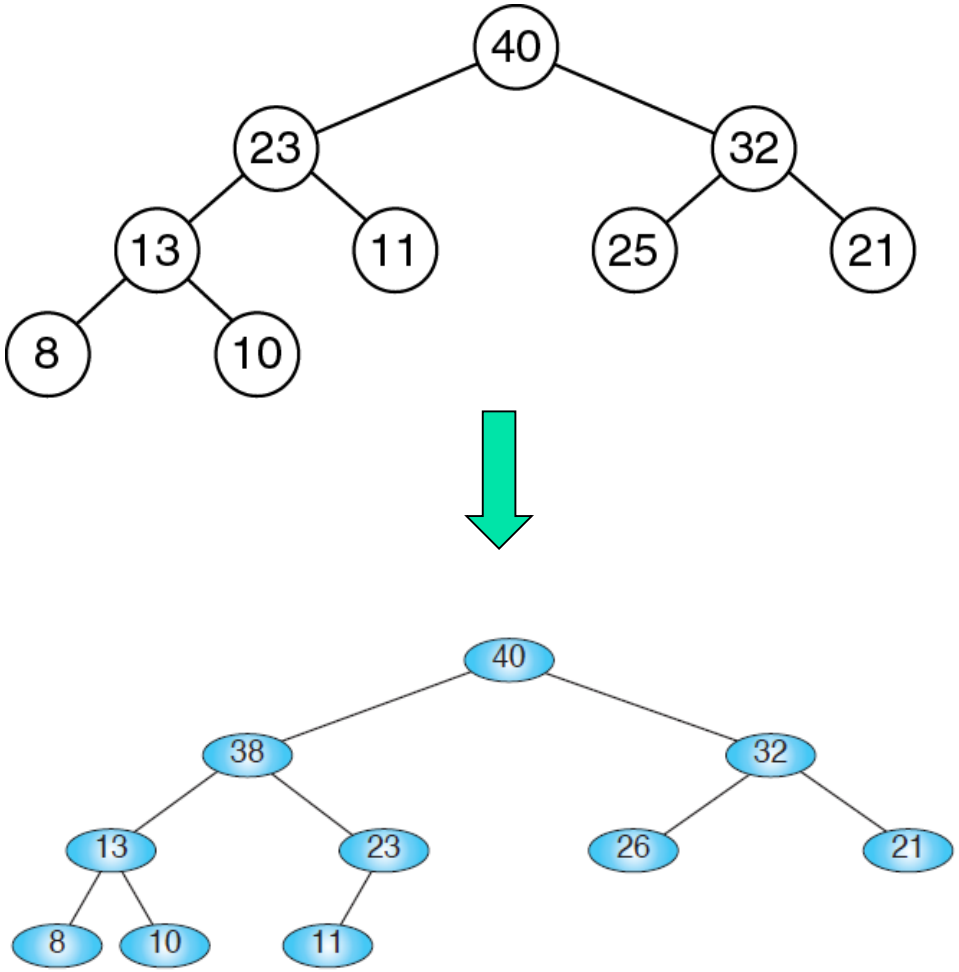


2) ReheapDown işlemi uygulanır



Alıştırma

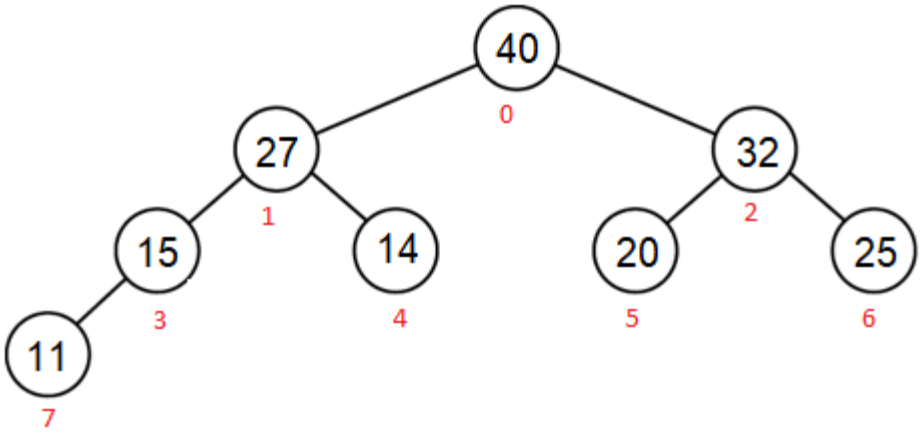
38'i heap'e ekleyin. Ekleme işleminin ardından heap'ı onarın.



Alıştırma

1. **Heap'te 32 ve 27'nin sol ve sağ çocuklarını gösteriniz.**
2. **14 ve 40'ın sol çocuklarını gösteriniz.**
3. **11'in ebeveynini, 20'nin ebeveynini ve 25'in ebeveynini gösteriniz.**

40	27	32	15	14	20	25	11
[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]



Alıştırma

Aşağıdakilerden hangileri heap’tir.

a. 42 35 37 20 14 18 7 10

b. 42 35 18 20 14 30 10

c. 20 20 20 20 20 20