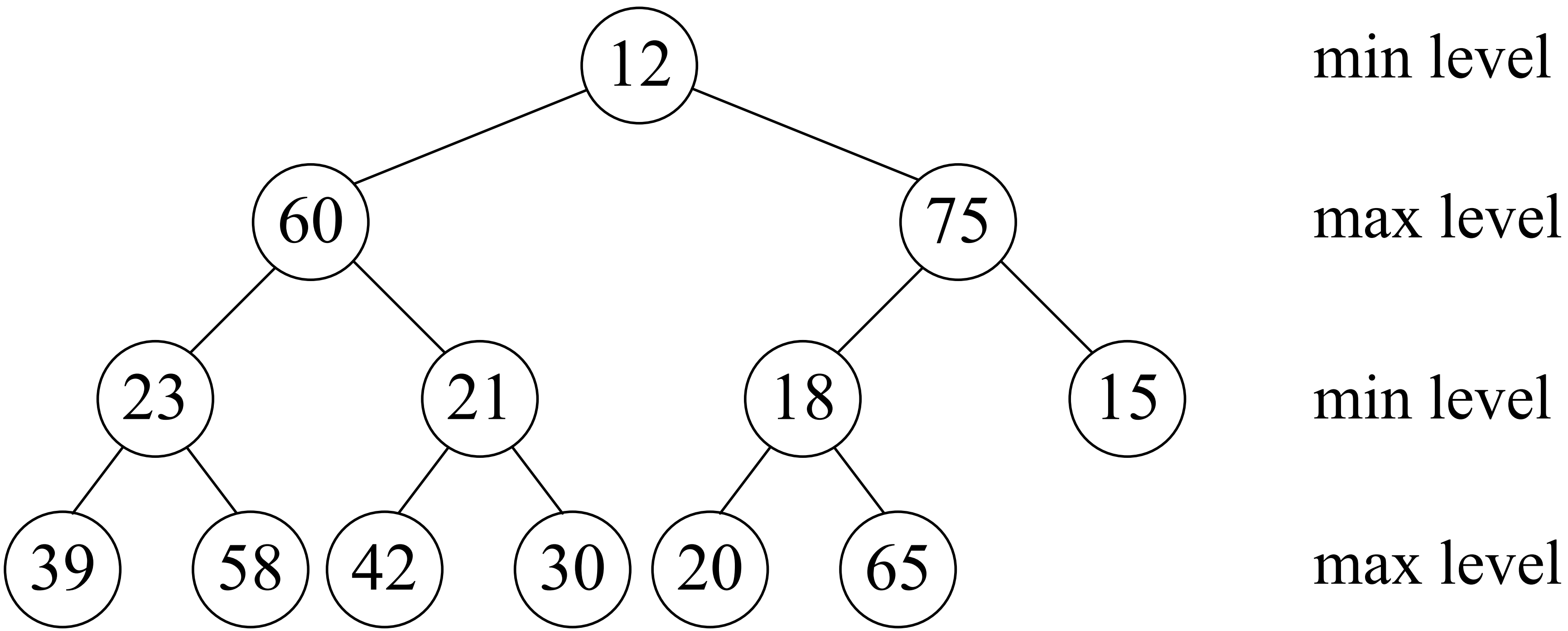
Advanced Data Structures

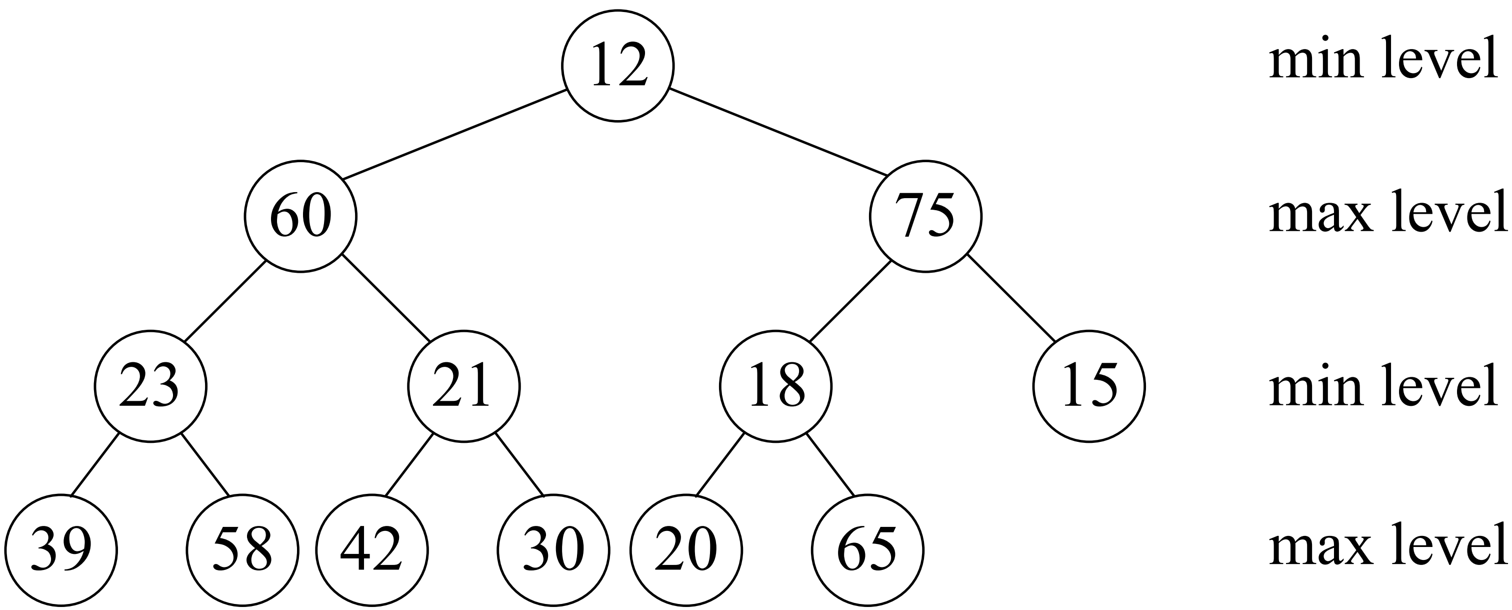
Min-Max Heap

Min-Max Heap



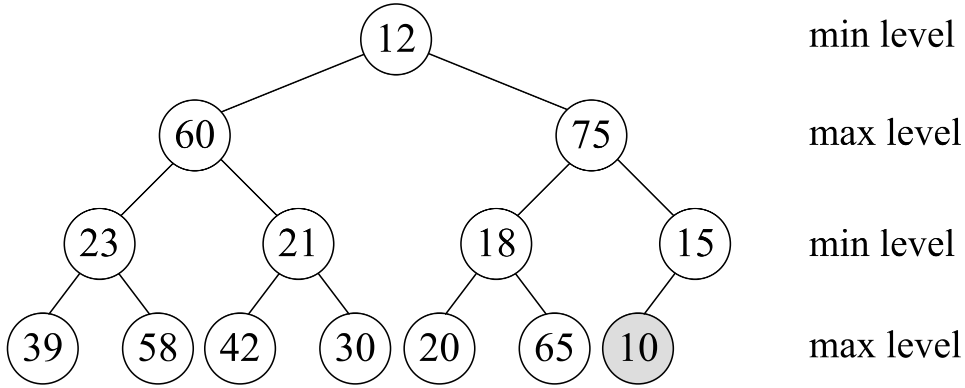
12
60
75
23
21
18
15
39
58
42
30
20
65
min level
max level
min level
max level

- Insert 10， 80

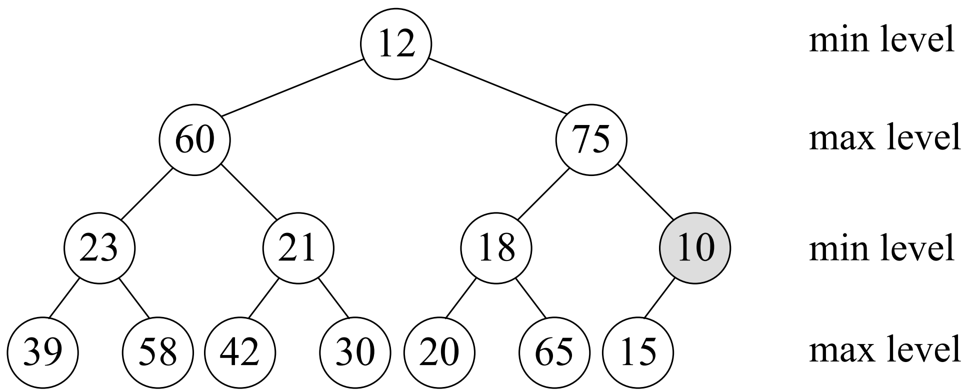


12
60
75
23
21
18
15
39
58
42
30
20
65
min level
max level
min level
max level

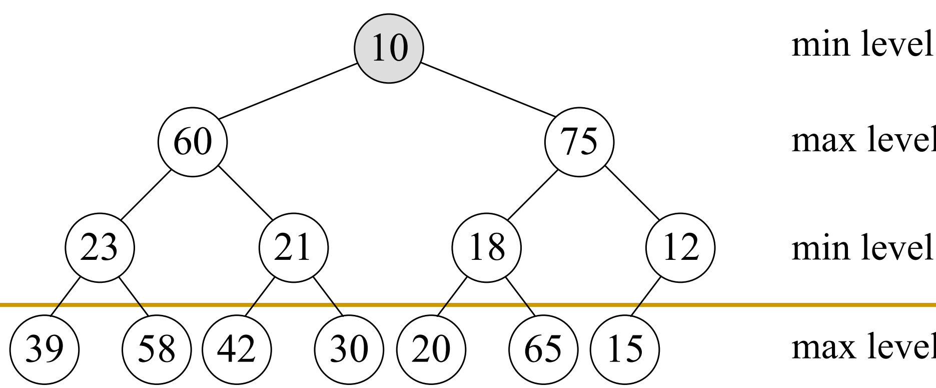
1.


12
60 75
23 21 18 15
39 58 42 30 20 65 10
min level
max level
min level
max level

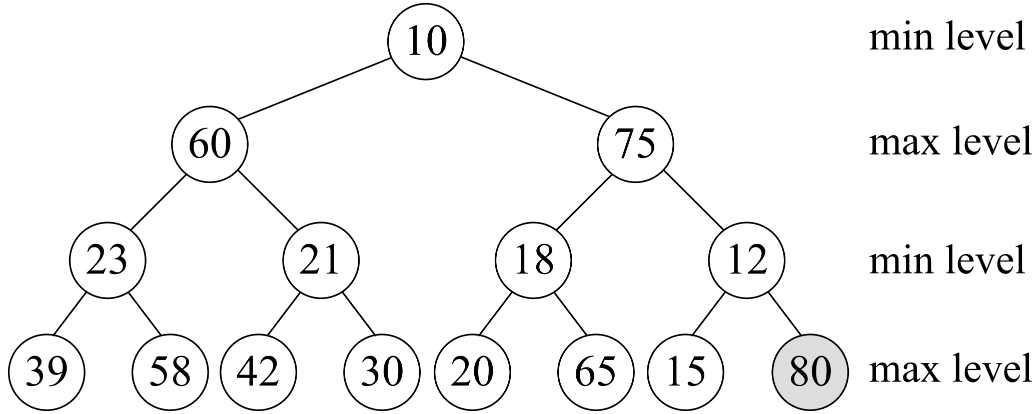
2.


12
60 75
23 21 18 10
39 58 42 30 20 65 15
min level
max level
min level
max level

3.

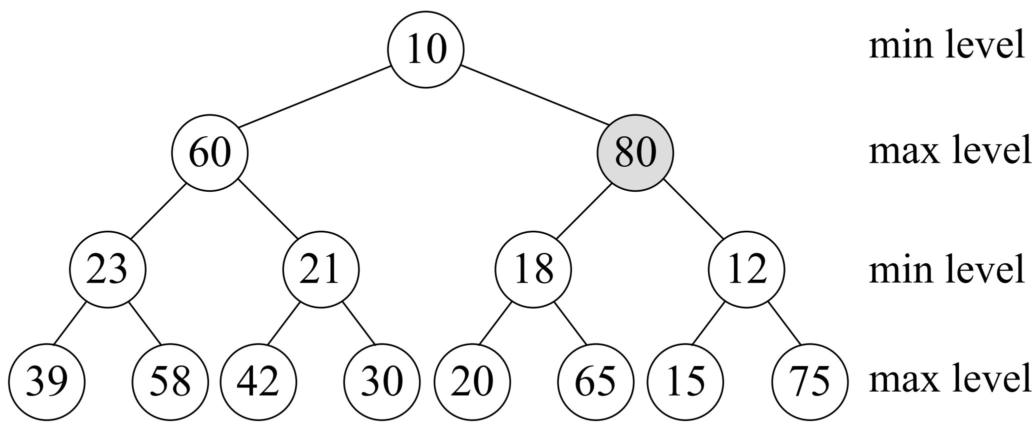

10
60 75
23 21 18 12
39 58 42 30 20 65 15
min level
max level
min level
max level

4.



5. New node is in max level and is larger than its parent，no exchange

6.



```
template <class Type>
void MinMaxHeap<Type>::Insert(const Element<Type>&x ) {
   if (n == MaxSize ) { MinMaxFull( ); return;}
   n++;
   int p = n/2;
   if (!p) {h[1] = x; return;}
   switch (level(p)) {
      case MIN:
         if (x.key < h[p].key) {
            h[n]=h[p];
            VerifyMin(p, x);
         }
```

```
      else VerifyMax(n, x);
      break;
    case MAX:
      if (x.key > h[p].key) {
        h[n]=h[p];
        VerifyMax(p, x);
      }
      else VerifyMin(n, x);
  }
}
```

- Level:

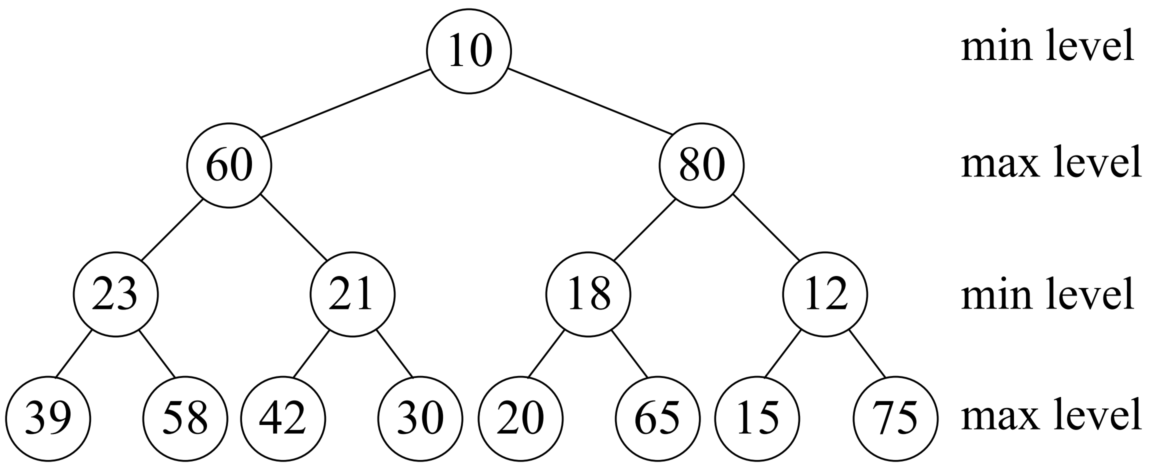
  Max Level:

  $\lceil \log_2(j + 1) \rceil$ is even

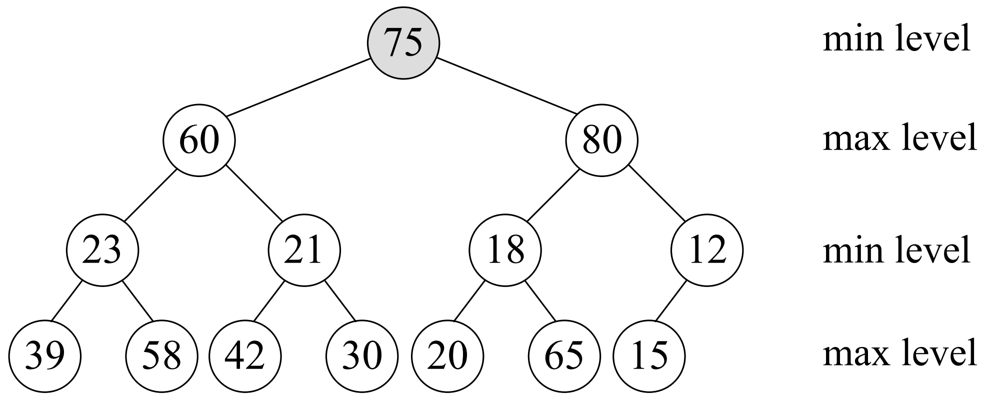
  Min Level:

  $\lceil \log_2(j + 1) \rceil$ is odd
- VerifyMax
- VerifyMin

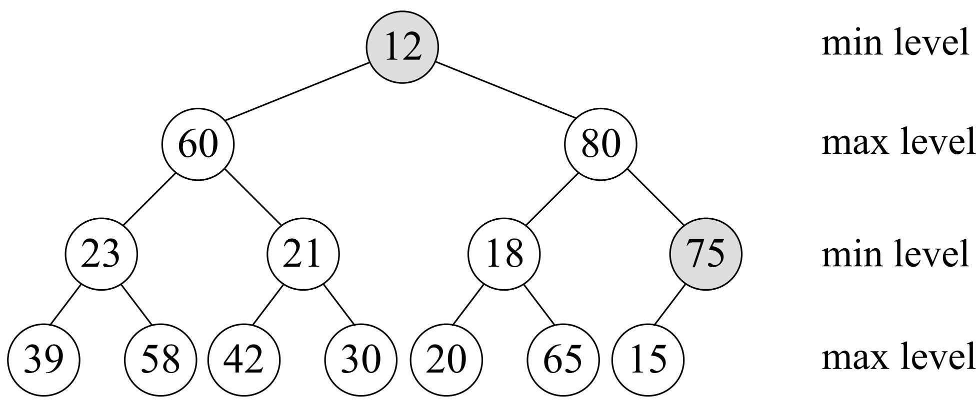
- Delete Min from Min-Max Heap



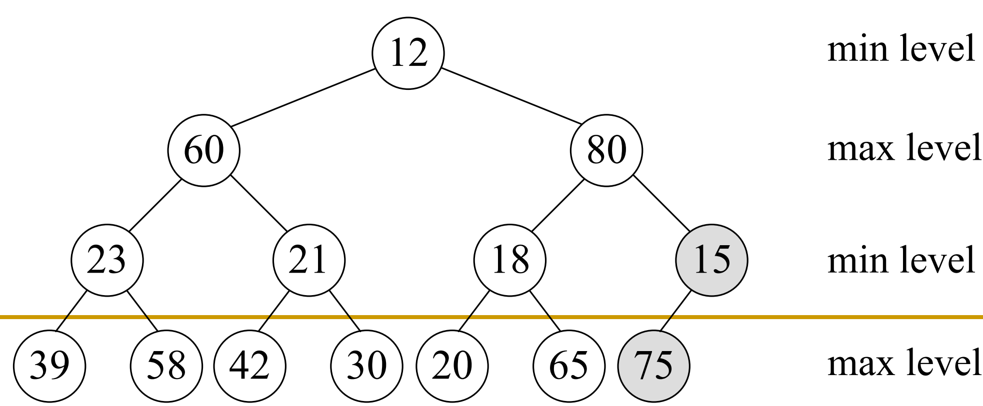
1.


75
60
80
23
21
18
12
39
58
42
30
20
65
15
min level
max level
min level
max level

2.


12
60
80
23
21
18
75
39
58
42
30
20
65
15
min level
max level
min level
max level

3.


12
60
80
23
21
18
15
39
58
42
30
20
65
75
min level
max level
min level
max level

Insert a new node x into a min-max heap with no root:

（1） empty heap: x is the new root;

（2） at least one child of root:

find node k with the smallest data;

(a) x. key≤h[k]. key。

x is the new root;

(b) x.key > h[k].key and k is a child of root:

replace k with x;

k is the new root.

(c) x.key > h[k].key and k is a grandchild of root:

k is the new root

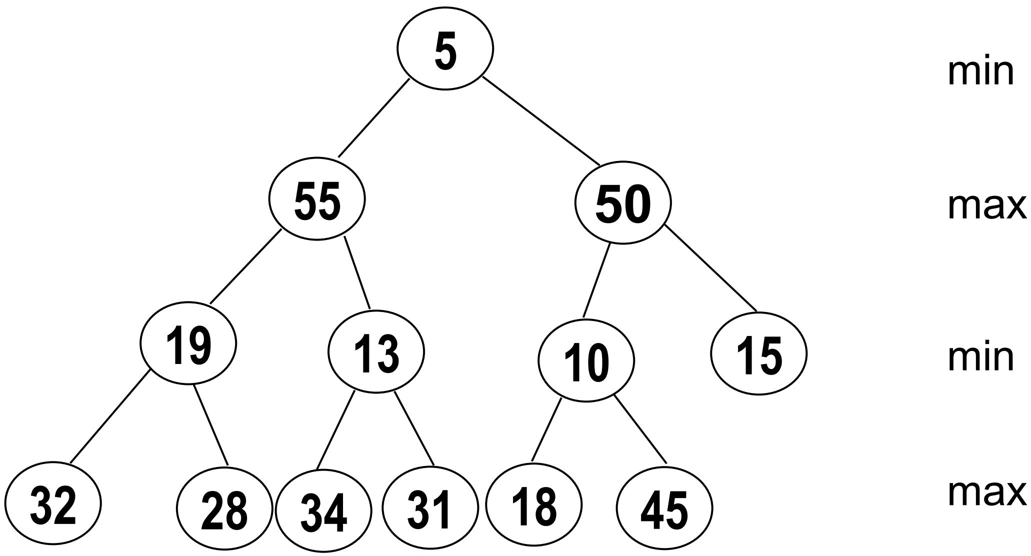
if x.key > h[p].key, exchange x and h[p]

insert x into (sub)min_max heap rooted with k

```
template <class Type>
Element<Type>* MinMaxHeap<Type>::
                                   DeleteMin(Element<Type>&y) {
    if (!n) { MinMaxEmpty( ); return 0; }
    y = h[1];
  Element<Type> x = h[n--];
    int i = 1, j = n/2;
  while (i <= j) {        //  has children
       int k = MinChildGrandChild(i);
       if (x.key <= h[k].key)
          break;          //  2(a)
```

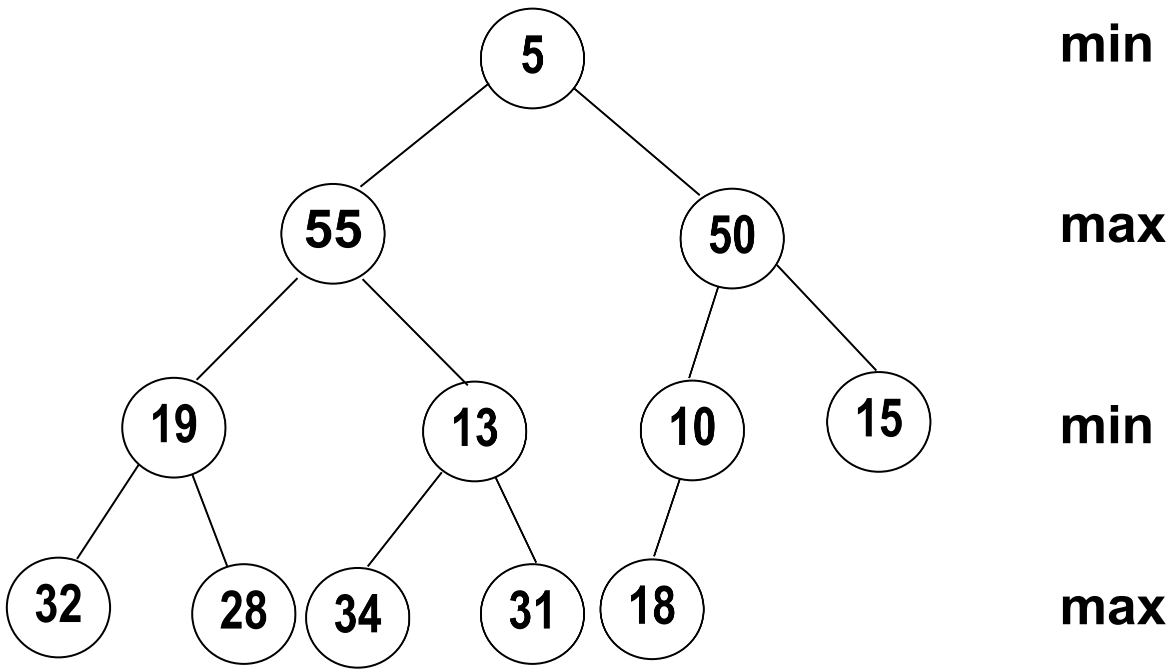
```
    else {             // 2(b) or (c)
        h[i] = h[k];
        if (k <= 2*i+1) {  // 2(b)
          i = k;
          break;
        }
        else {      // 2(c)
          int p = k/2;
          if (x.key > h[p].key) {
            Element<Type> t = h[p]; h[p] = x; x = t;
          }
        }
      i = k;
      }
}
    h[i] = x;
    return &y;
}
```

Min-max heap



- Delete 45

Min-max heap (con.t)



- Delete 55

Advanced Data Structures
—— Deap

A deap is a complete binary tree：

（1） no data in root;

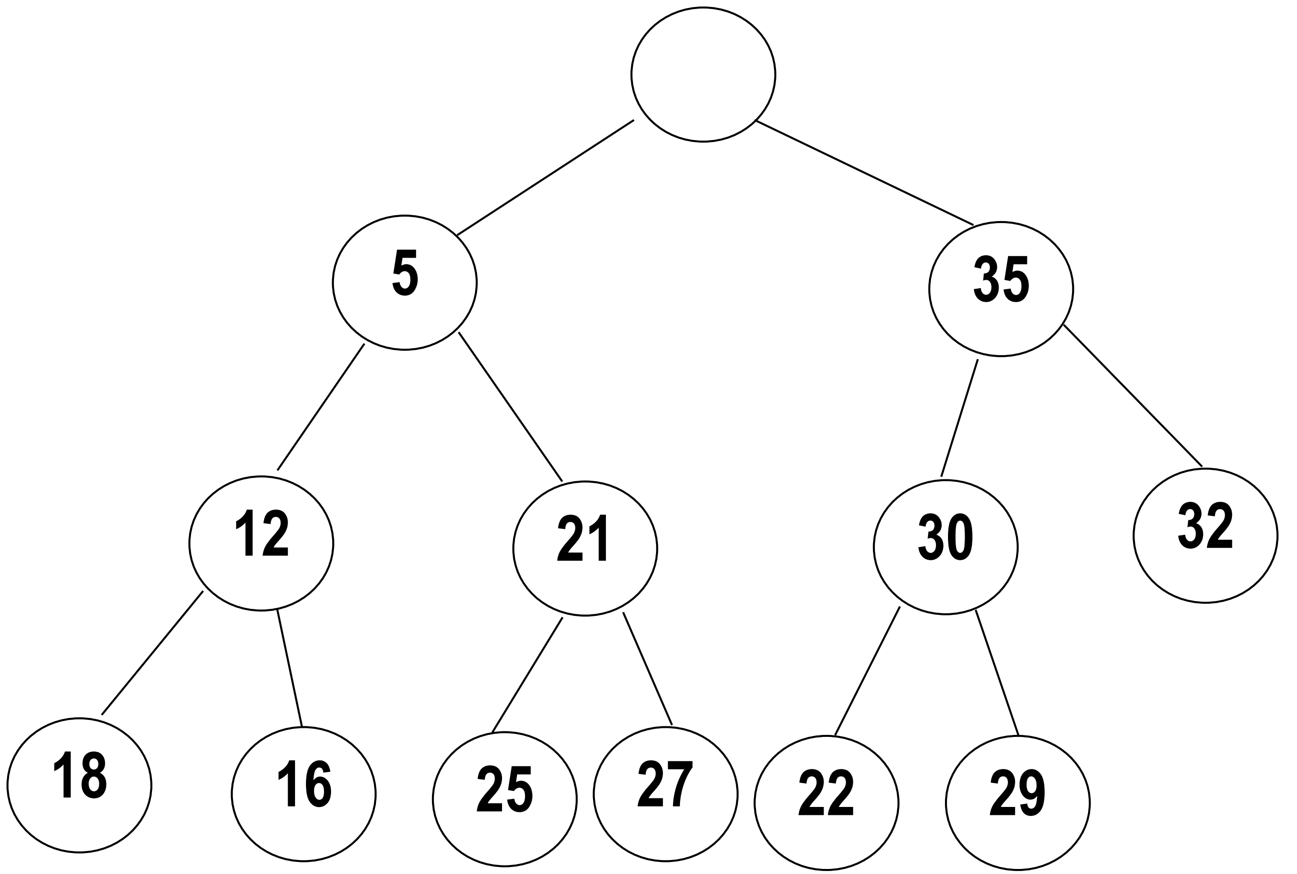
（2） left subtree is a min heap

（3） right subtree is a max heap

（4） every node in left subtree is no bigger than its corresponding node in right subtree

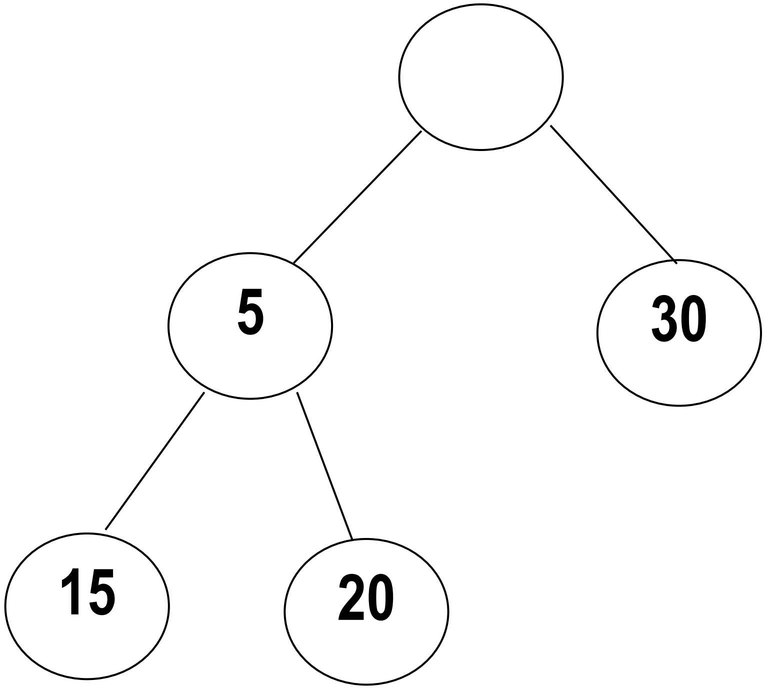
corresponding nodes

Deap



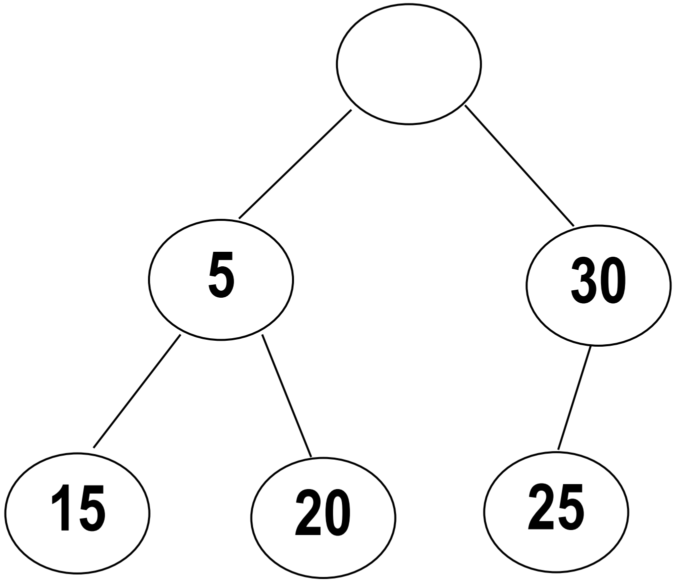
Deap (con.t)

- Insertion
  - Add new node to the end of tree, Heapify



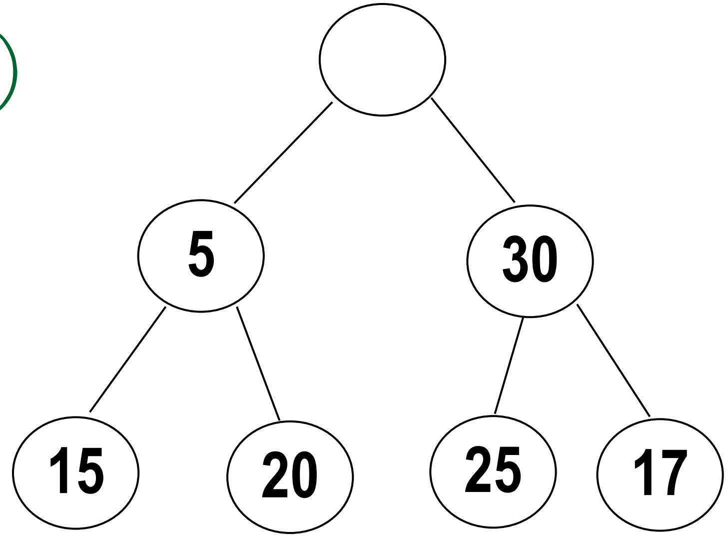
- Insert 25

Deap (con.t)

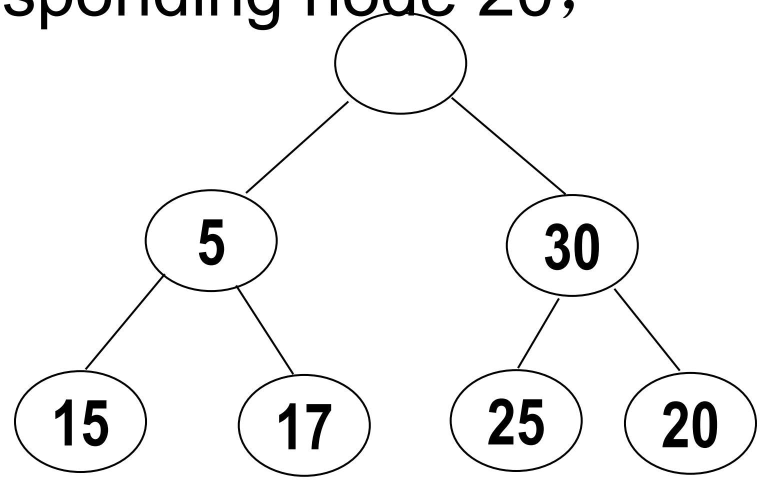


- Insert 17

Deap (con.t)

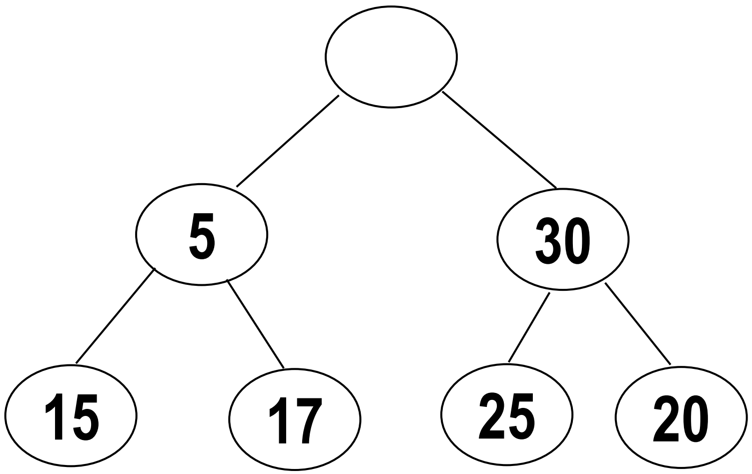


- 17 is smaller than its corresponding node 20，
  - Exchange 17 and 20
- Check the min-heap

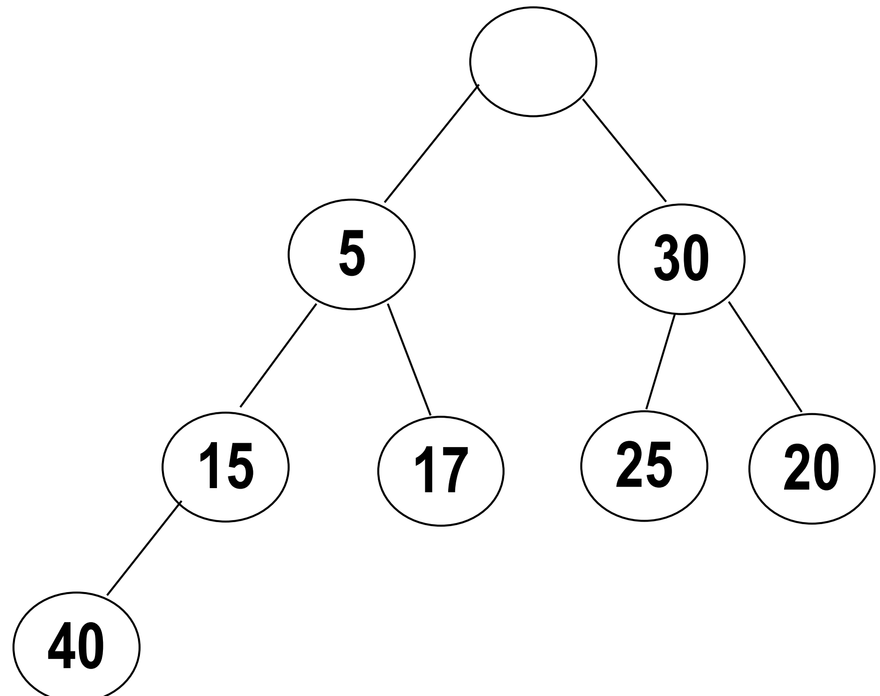


- No need to check the max-heap, WHY？

Deap (con.t)

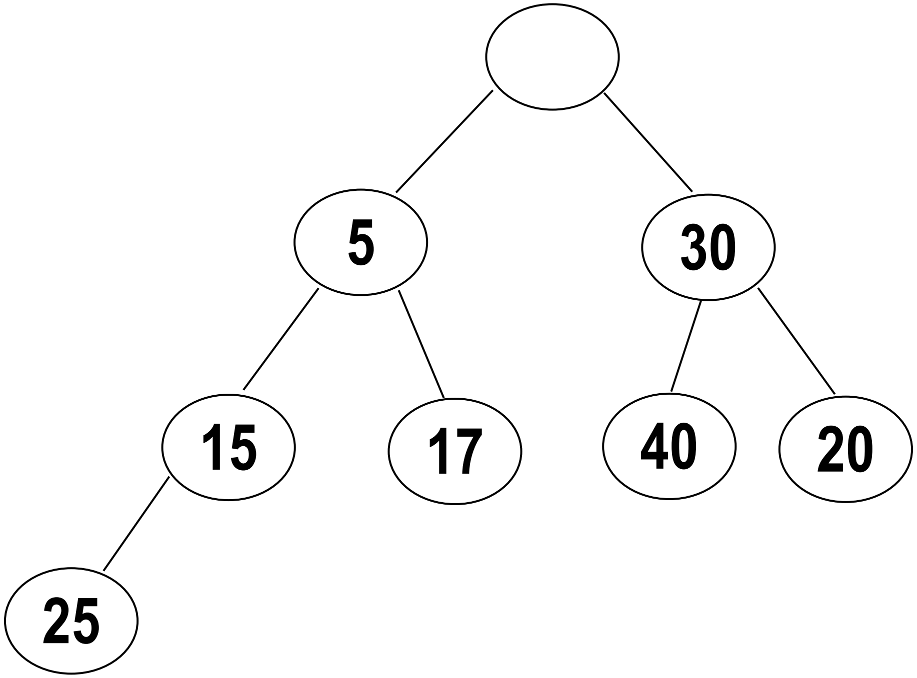


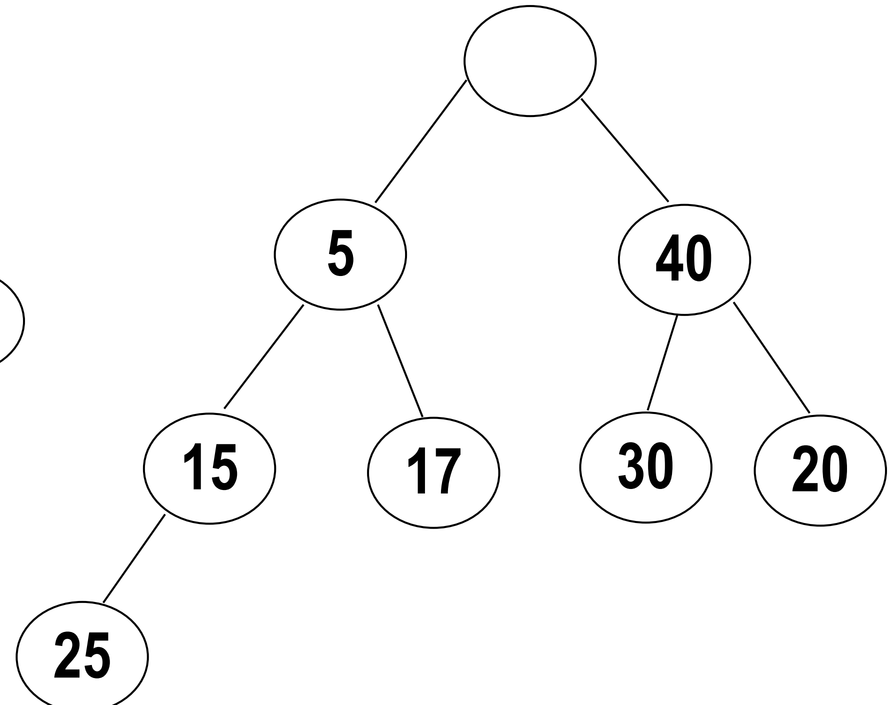
- Insert 40



- 40 is bigger than 25(？)
  - Exchange 40 and 25

Deap (con.t)





- The right subtree is not a max-heap
  - Heapify
- No need to adjust the left subtree, WHY ?

```
template <class Type>
void Deap<Type>::Insert (const Element<Type>&x ) {

  int i;
  if (n == MaxSize ) { DeapFull( ); return;}
  n++;
  if (n == 1) { d[2] = x; return;}
  int p = n + 1;
  switch (MaxHeap(p)) {
    case TRUE:          // p in max-heap
           i = MinPartner(p);
           if (x.key < d[i].key) {
              d[p] = d[i];
              MinInsert(i, x);
           }
```

```
        else MaxInsert(p, x);
        break;
    case FALSE:         // p in min-heap
        i = MaxPartner(p);
        if (x.key > d[i].key) {
          d[p] = d[i];
          MaxInsert(i, x);
        }
        else MinInsert(p, x);
  }                 //
}         // Insert
```

O(log n)

(1) **Deap::MaxHeap(int p)**

for p > 1, if $2^{\lfloor \log_2 p \rfloor} + 2^{\lfloor \log_2 p \rfloor - 1} \le p < 2^{\lceil \log_2 p \rceil}$, p is in the max-heap

(2) **Deap::MinPartner(int p)**

P is in max-heap and P is the last one in Deap:

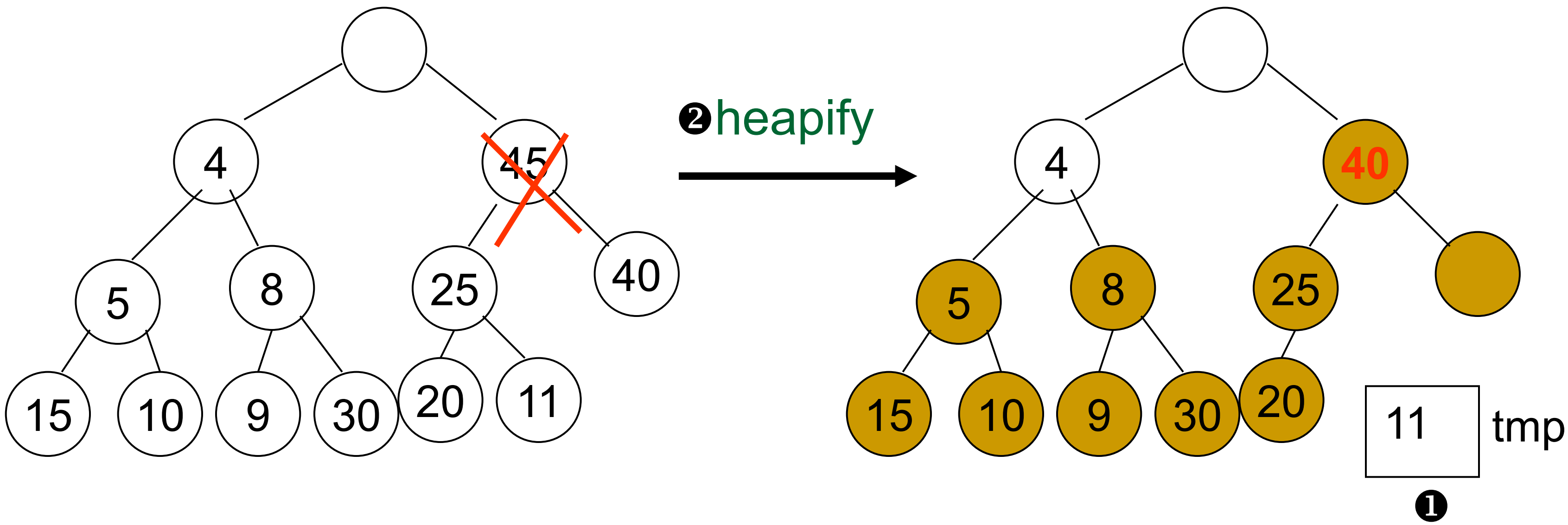
$$p - 2^{\lfloor \log_2 p \rfloor - 1}$$

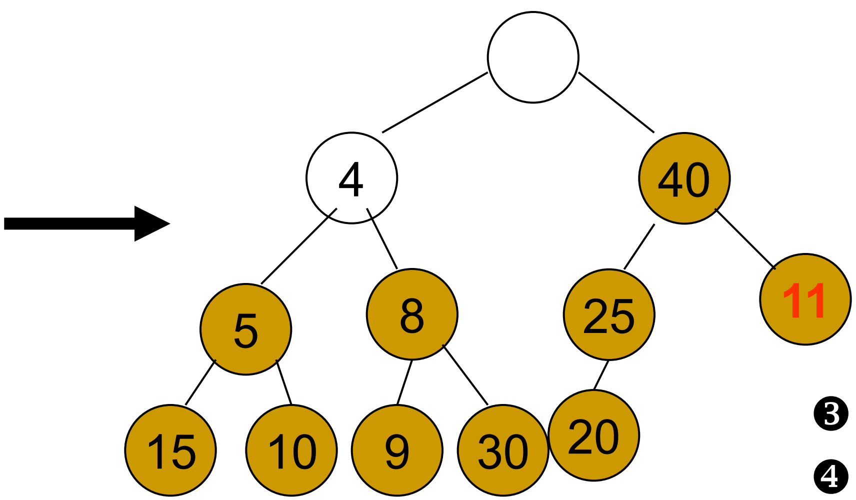
(3) **Deap::MaxPartner(int p)**

P in min-heap and P is the last one in Deap:

$$(p + 2^{\lfloor \log_2 p \rfloor - 1}) / 2$$

Delete: fetch max

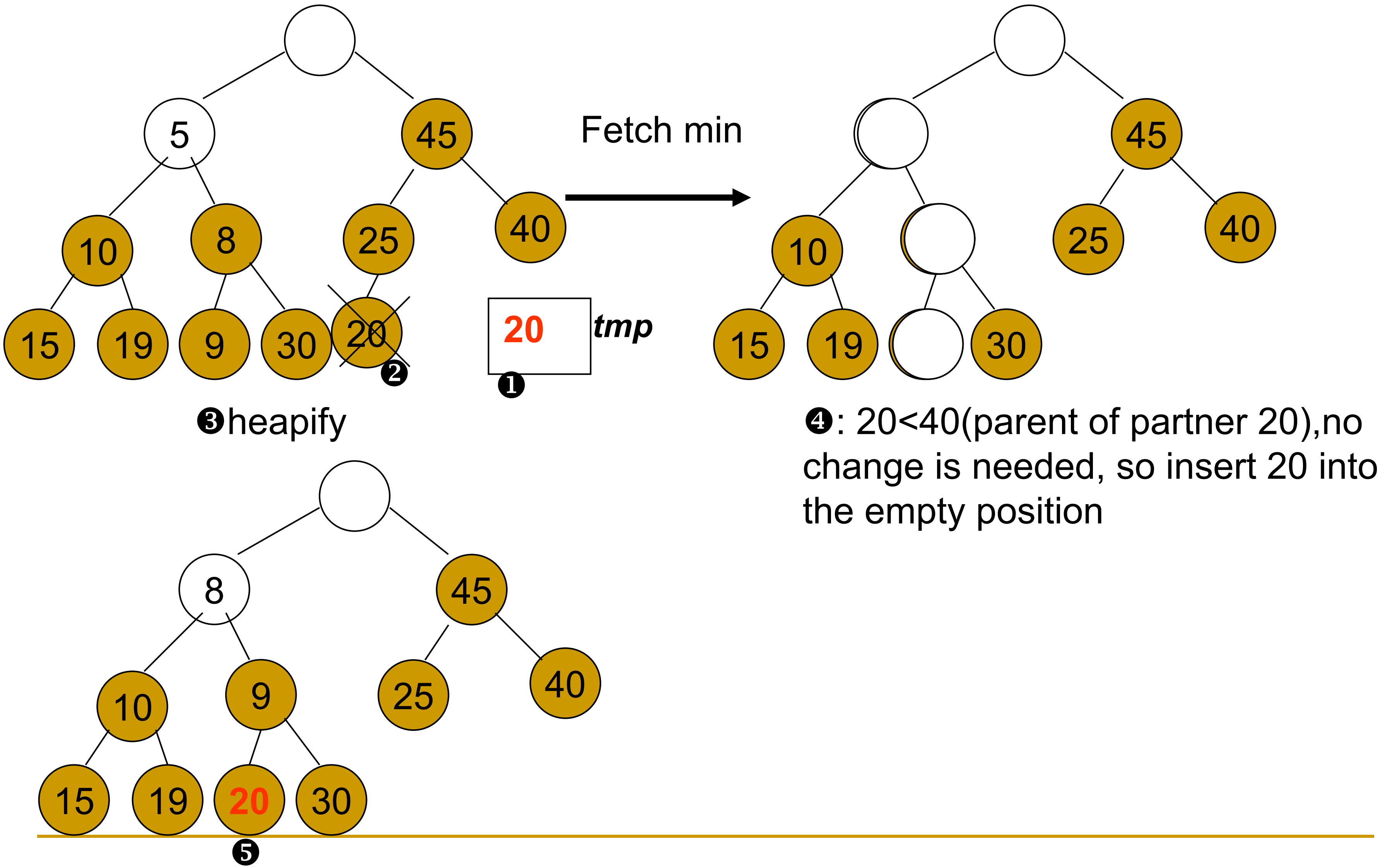




❸: 11>8(partner)

❹: insert tmp into empty position

Delete min



5
45
10
8
25
40
15
19
9
30
20
❷
Fetch min
20
tmp
❶
❸heapify
45
10
25
40
15
19
30
❹: 20<40(parent of partner 20),no change is needed, so insert 20 into the empty position
8
45
10
9
25
40
15
19
20
30
❺