Chap 5. Trees (4)

Contents

5.6 Heaps

5.6.1 Priority Queues

- *Priority queues*
  - *deletion*: deletes the element with the highest(or the lowest) priority
  - *insertion* : insert an element with arbitrary priority

  (ex: job scheduling in OS)

- We use *max(min) heap* to implement the priority queues

ADT *MaxPriorityQueue* is

objects: a collection of $n > 0$ elements, each element has a key

functions:

for all $q \in$ *MaxPriorityQueue*, *item* $\in$ *Element*, $n \in$ integer

MaxPriorityQueue create(*max_size*)	::=	create an empty priority queue.
Boolean isEmpty(q, n)	::=	**if** ($n > 0$) **return** *FALSE* **else return** *TRUE*
Element top(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q **else return** error.
Element pop(q, n)	::=	**if** (!isEmpty(q, n)) **return** an instance of the largest element in q and remove it from the heap **else return** error.
MaxPriorityQueue push(q, *item*, n)	::=	insert *item* into q and return the resulting priority queue.

ADT 5.2: Abstract data type *MaxPriorityQueue*

5.6.2 Definition of a Max Heap

- **Definition :**
  - A ***max tree*** is a tree in which the key value in each node is no smaller than the key values in its children (if any). *parent's key ≥ children's keys*
  - A ***max heap*** is a complete binary tree that is also a max tree

- **Definition :**
  - A ***min tree*** is a tree in which the key value in each node is no larger than the key values in its children (if any). *parent's key ≤ children's keys*
  - A ***min heap*** is a complete binary tree that is also a min tree.

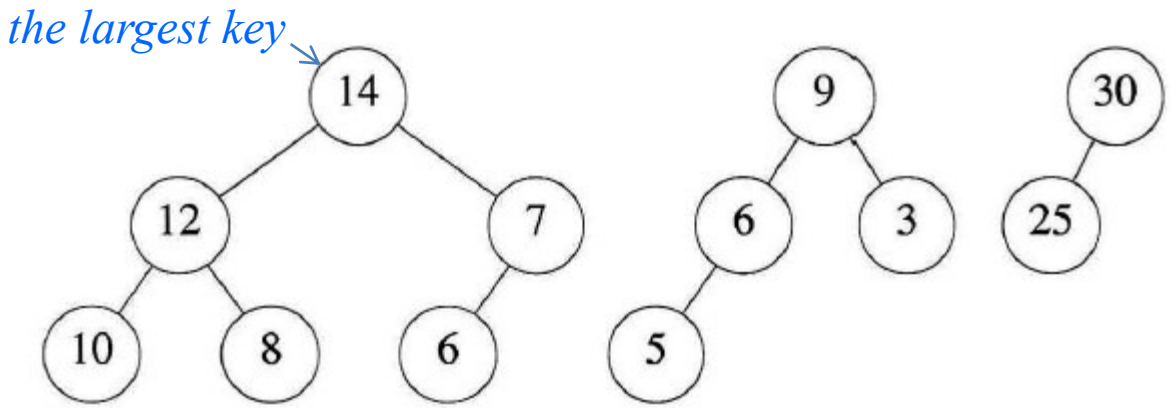


Figure 5.25: Max heaps

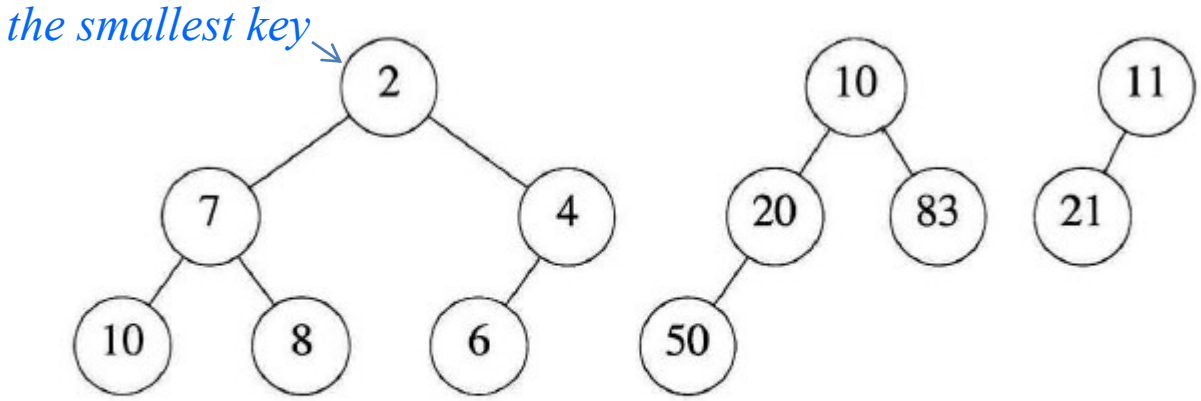


Figure 5.26: Min heaps

5.6.3 Insertion into a Max Heap

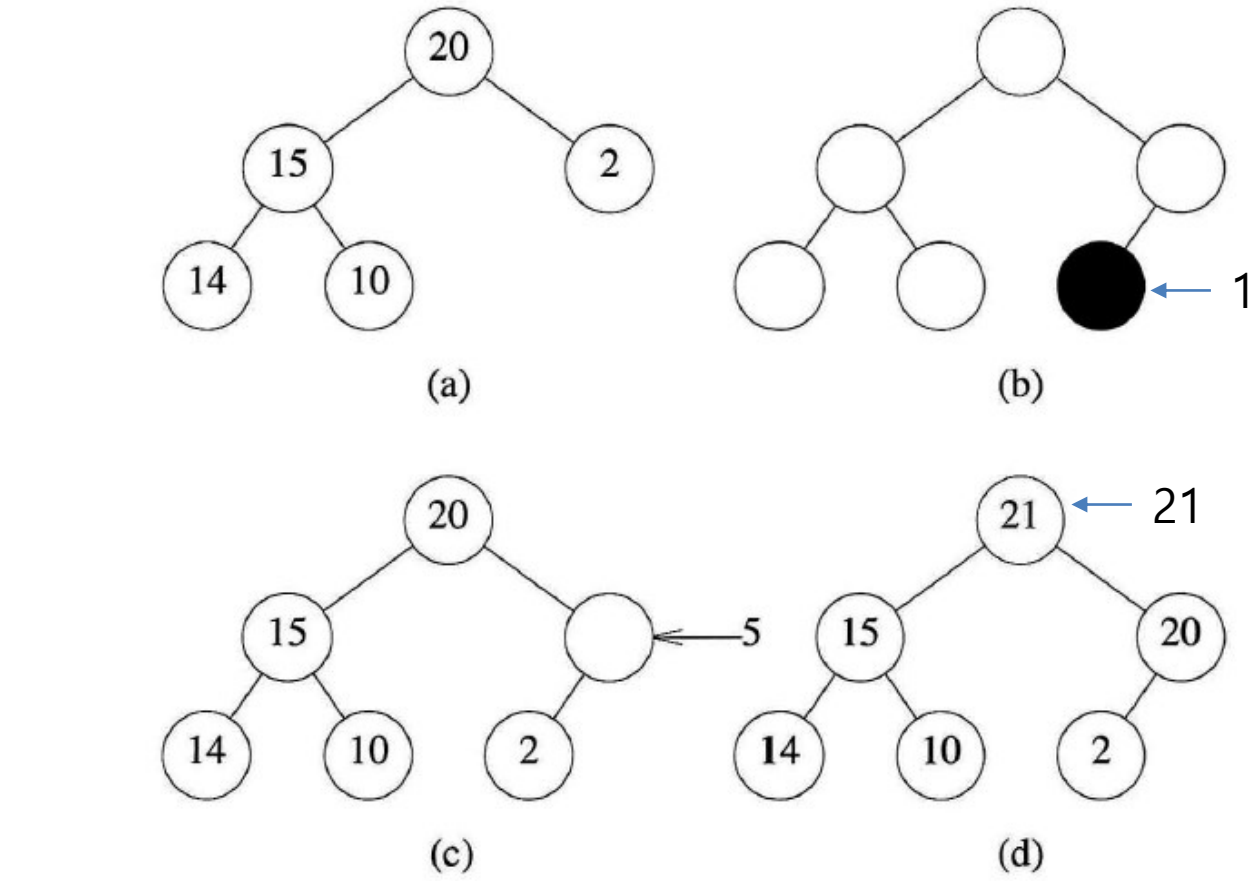


Figure 5.27: Insertion into a max heap

(a)

Insert(1)/
Insert(5)/
Insert(21)

?

```
#define MAX_ELEMENTS 200 /* maximum heap size+1 */
#define HEAP_FULL(n) (n == MAX_ELEMENTS-1)
#define HEAP_EMPTY(n) (!n)
typedef struct {
        int key;
        /* other fields */
        } element;
element heap[MAX_ELEMENTS];
int n = 0;
```

heap size 199

```
void push(element item, int *n)
{/* insert item into a max heap of current size *n */
   int i;
   if (HEAP_FULL(*n)){
      fprintf(stderr, "The heap is full. \n");
      exit(EXIT_FAILURE);
   }
   i = ++(*n);
   while ((i != 1) && (item.key > heap[i/2].key)) {
      heap[i] = heap[i/2];
      i /= 2;
   }
   heap[i] = item;
}
```

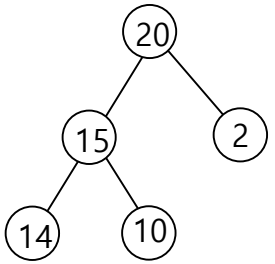
Program 5.13: Insertion into a max heap

```
void push(element item, int *n)
{   ...
   i = ++(*n);
   while ((i != 1) && (item.key > heap[i/2].key)) {
      heap[i] = heap[i/2];
      i /= 2;
   }
   heap[i] = item;
}
```

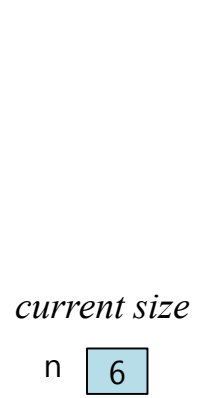
current size

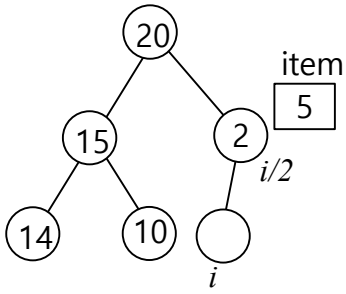
n | 5 |

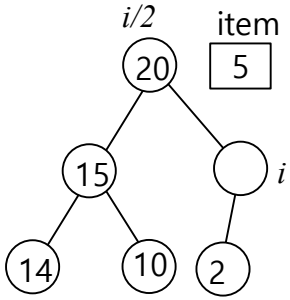
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10		...

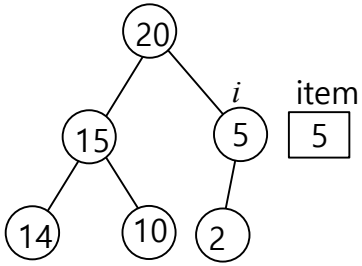


push(5, &n)







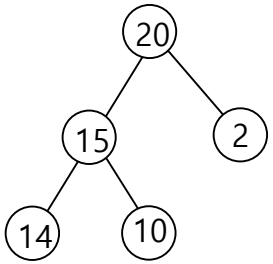


```
void push(element item, int *n)
{   ...
   i = ++(*n);
   while ((i != 1) && (item.key > heap[i/2].key)) {
      heap[i] = heap[i/2];
      i /= 2;
   }
   heap[i] = item;
}
```

current size

n | 5

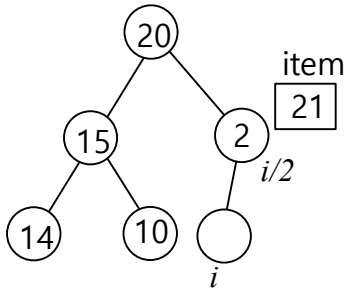
	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10		...

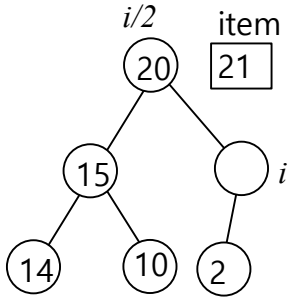


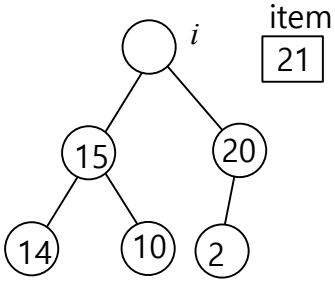
push(21, &n)

current size

n | 6







- Analysis of *push*
  - the height of heap with n elements : $\lceil \log_2(n+1) \rceil$
  - while loop is iterated $O(\log_2 n)$ times
  - time complexity: $O(\log_2 n)$

5.6.4 Deletion from a Max heap

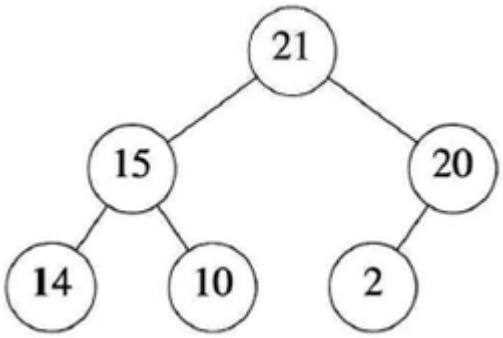


Figure 5.27 (d)

delete( )

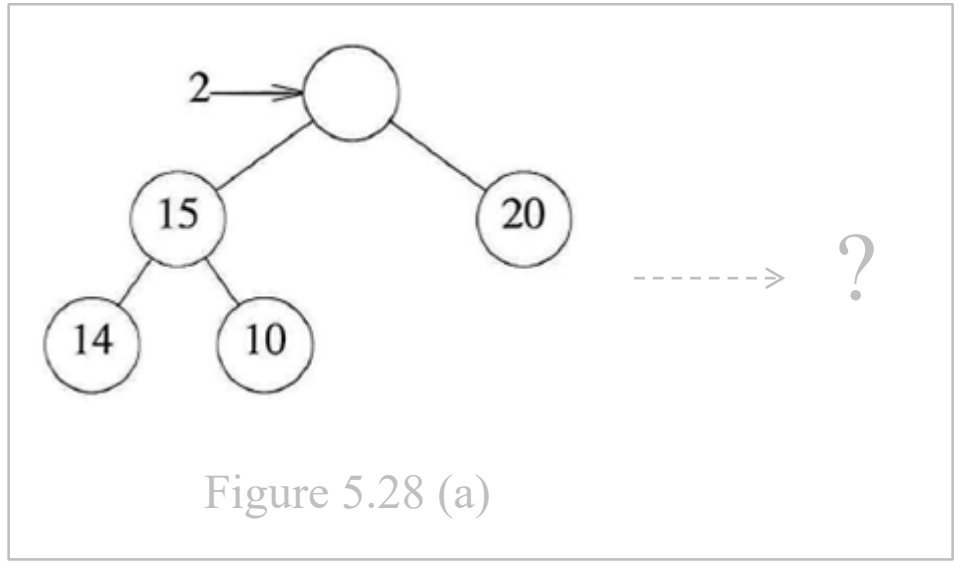
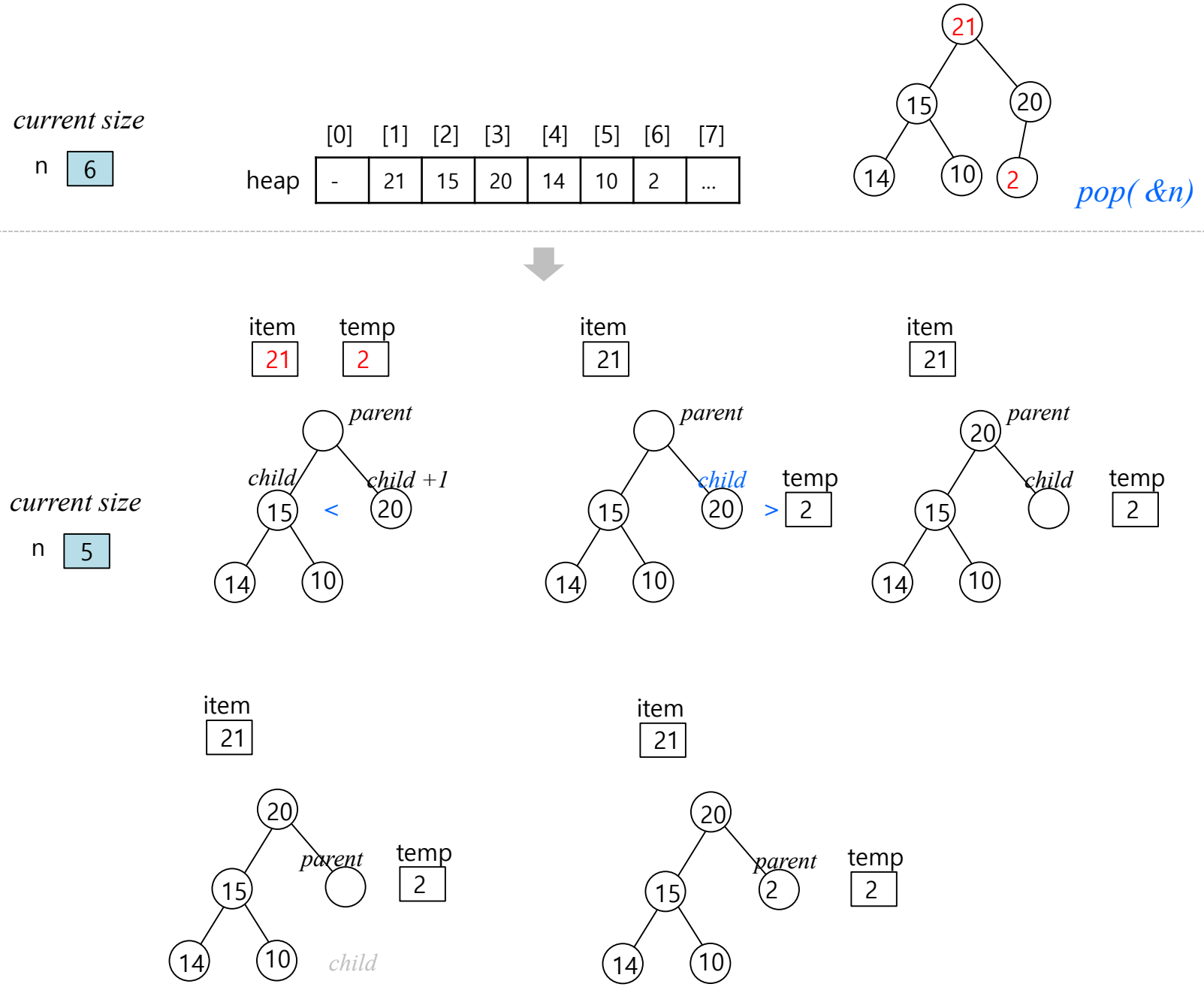


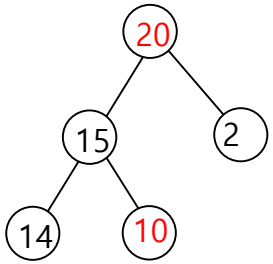
Figure 5.28 (a)


current size
n
6
heap
[0] [1] [2] [3] [4] [5] [6] [7]
- 21 15 20 14 10 2 ...
21
15 20
14 10 2
pop(&n)
current size
n
5
item
21
temp
2
parent
child
child +1
15
<
20
14 10
item
21
parent
child
temp
15
20
>
2
14 10
item
21
20
parent
child
temp
15
2
14 10
item
21
20
parent
temp
15
2
14 10
child
item
21
20
parent
temp
15
2
14 10

current size

n 5

	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]
heap	-	20	15	2	14	10	...	

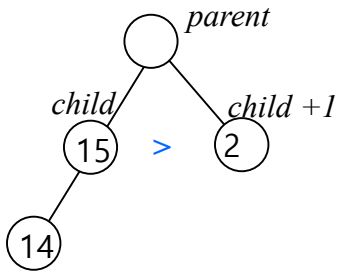


pop(&n)

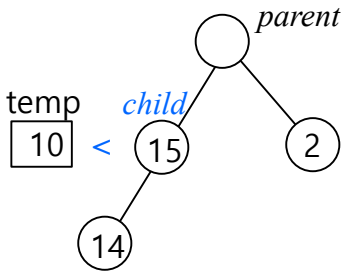
current size

n 4

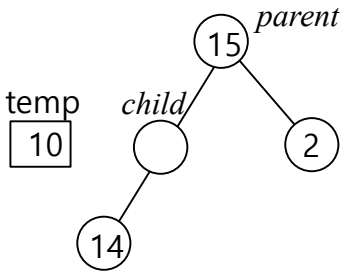
item 20 temp 10



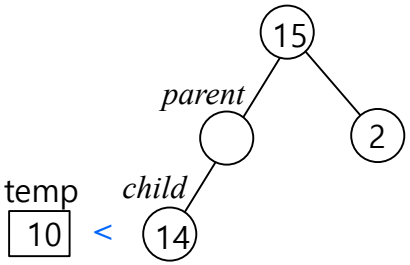
item 20



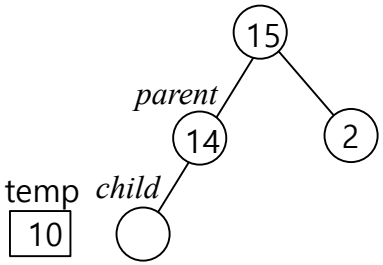
item 20



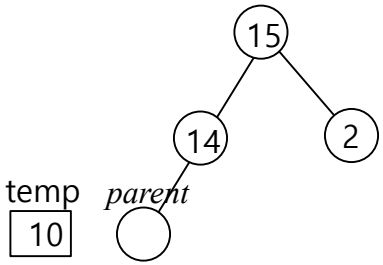
item 20



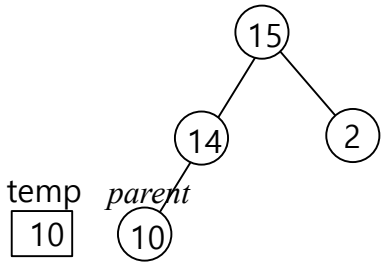
item 20



item 20



item 20



child

```
element pop(int *n)
{/* delete element with the highest key from the heap */
   int parent, child;
   element item, temp;
   if (HEAP_EMPTY(*n)) {
      fprintf(stderr, "The heap is empty\n");
      exit(EXIT_FAILURE);
   }
   /* save value of the element with the highest key */
   item = heap[1];
   /* use last element in heap to adjust heap */
   temp = heap[(*n)--];
   parent = 1;
   child = 2;
   while (child <= *n) {
      /* find the larger child of the current parent */
      if((child < *n) && (heap[child].key < heap[child+1].key))
         child++;
      if (temp.key >= heap[child].key) break;
      /* move to the next lower level */
      heap[parent] = heap[child];
      parent = child;
      child *= 2;
   }
   heap[parent] = temp;
   return item;
}
```

Program 5.14: Deletion from a max heap

- Analysis of *pop*
  - the height of heap with n elements : $\lceil \log_2(n+1) \rceil$
  - while loop is iterated $O(\log_2 n)$ times
  - time complexity: $O(\log_2 n)$