14. 히프와 우선순위 큐

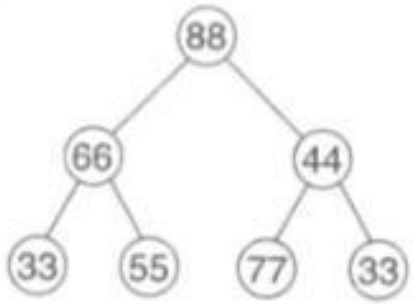
14.1 히프

- 히프의 정의
  - 리프-루트 경로를 따라가는 모든 키가 오름차순으로 되어 있는 완전 이진 트리
  - 이러한 순서에 대한 제약은 어떤 키도 부모보다 크지 않다고 말하는 것과 동일함
  - 이것은 히프를 그것의 키에 대한 ≤ 관계를 가지는 부분 순서로 만듦

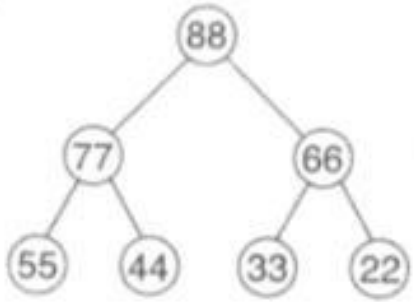
- 히프화(heapify) 연산
  - 히프 특성을 만족하도록 시이퀀스의 원소들을 재배열함
  - 리프-루트 경로를 따라 인접해 있는 두 개 이상의 원소를 회전시킴

다음 중 어떤 이진 트리가 히프인지 결정하라.

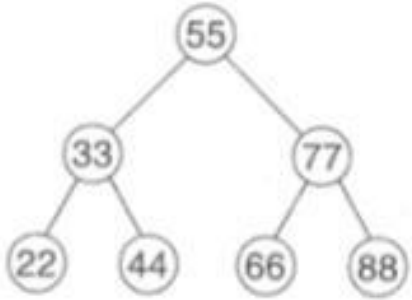
a.



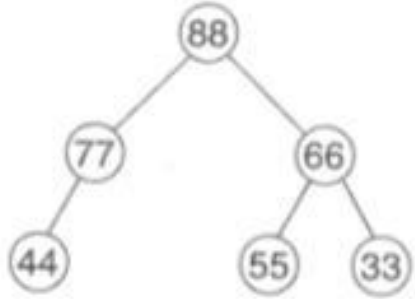
b.



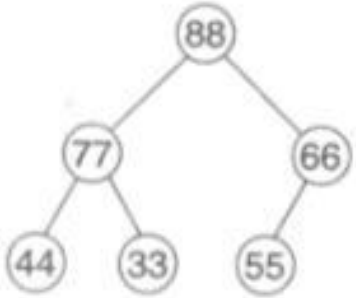
c.



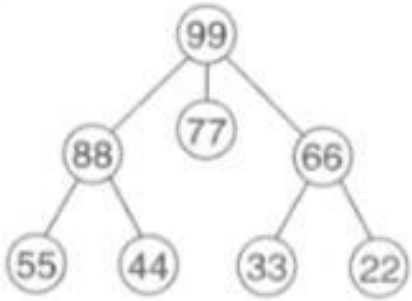
d.



e.



f.

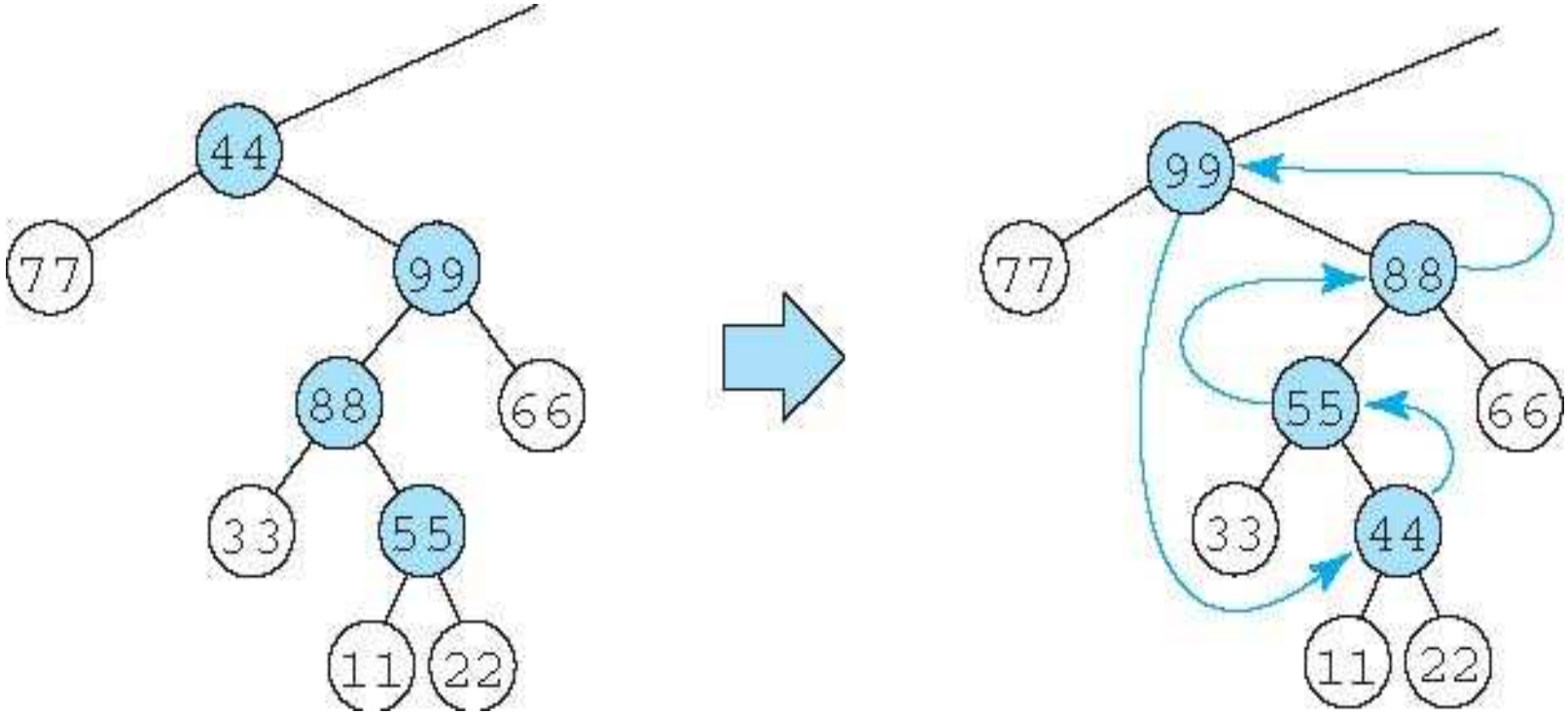


14.2 히프 알고리즘

- 히프화 연산 알고리즘
  - 입력 : 완전 이진 트리에 있는 노드 x.
  - 선조건 : x의 두 서브트리가 히프임.
  - 후조건 : x가 루트인 서브트리가 히프임.

  1. *temp*=*x.key*로 설정.
  2. *x*가 리프가 아닌 동안, 단계 3-4를 수행.
  3. *y*를 *x*의 큰 자식으로 설정.
  4. 만일 *y.key* > *temp*이면 단계 5-6을 수행.
  5. *y*를 *x*로 복사.
  6. *x*=*y*로 설정.
  7. *temp*를 *x*로 복사.

히프화 경로



히프화 메소드

- LISTING 14.1: The heapify() Method

```
1 void heapify(int[] a, int i, int n) {
2     int ai = a[i];
3     while (i < n/2) {                  // while a[i] is not a leaf
4       int j = 2*i + 1;                 // a[j] is ai's left child
5       if (j+1 < n && a[j+1] > a[j])
            ++j;                         // a[j] is ai's larger child
6       if (a[j] <= ai) break;           // a[j] is not out of order
7       a[i] = a[j];                     // promote a[j]
8       i = j;                           // move down to next level
9     }
10    a[i] = ai;
11 }
```

히프 구축 알고리즘

- ALGORITHM 14.2: The Build Heap Algorithm
  - 입력 : 완전 이진 트리 T.
  - 후조건 : T가 히프임.

    1. 만일 T가 단독 트리이면, 리턴.
    2. 왼쪽 서브트리에 대해 히프 구축 알고리즘 적용.
    3. 오른쪽 서브트리에 대해 히프 구축 알고리즘 적용.
    4. 루트에 대해 히프화 알고리즘 적용.

다른 방법으로 히프 구축

- LISTING 14.2: The buildHeap() Method

```
1 void buildHeap(int[] a, int i, int n) {
2         if (i >= n/2) return;
3         buildHeap(a, 2*i+1, n);
4         buildHeap(a, 2*i+2, n);
5         heapify(a, i, n);
6 }
```

Complexity of buildHeap(a, 0, n) : $\theta(n)$

히프 구축 메소드

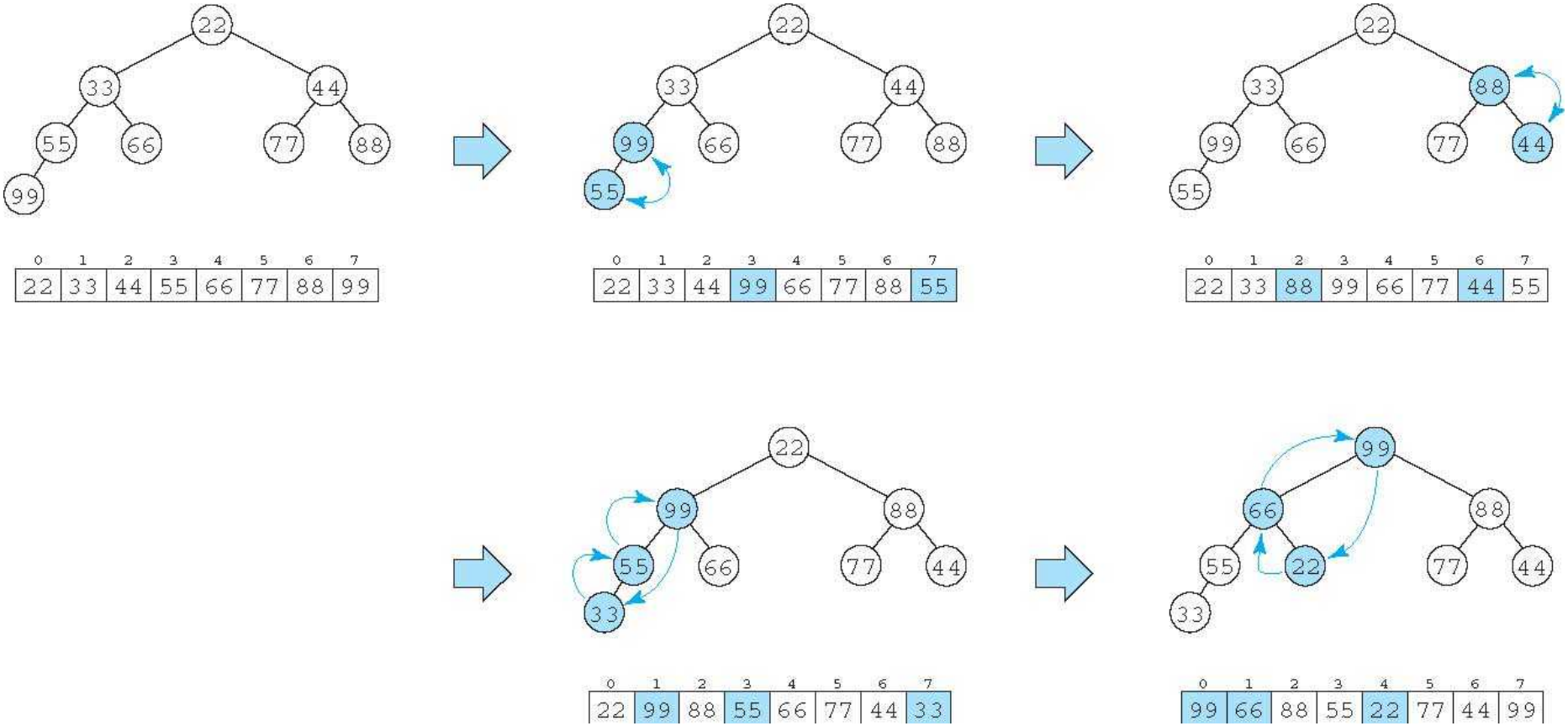
- Rewrite buildHeap() in the iterative fashion

```
void buildHeap(int[] a, int i, int n) {



}
```

트리에 있는 각각의 내부 노드에 적용되는 히프화 연산



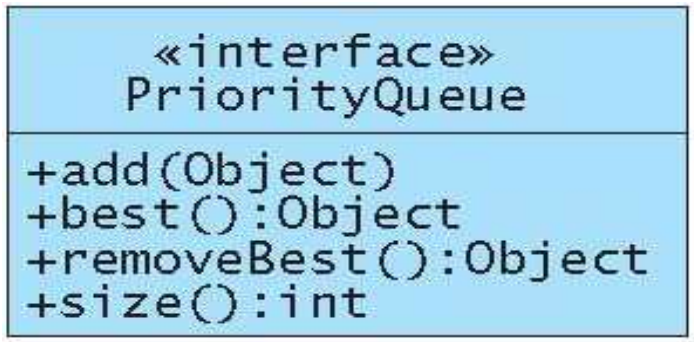
14.3 우선순위 큐

- 정의
  - 원소의 우선순위에 삭제 연산이 수행되는 큐
  - 어떤 것이 높은 우선순위를 가지는지 결정하기 위해 원소들 간의 비교가 가능하다는 것을 가정하고 있음
  - 만일 일반적인 큐를 선입선출(first-in, first-out) 자료 구조(FIFO)로 생각한다면, 우선순위 큐는 최적입선출(best-in, first-out)인 자료 구조(BIFO)로 생각할 수 있음

- 힙으로 우선순위큐를 구현
  - 우선순위가 가장 높은 원소가 루트에 위치함

PriorityQueue ADT

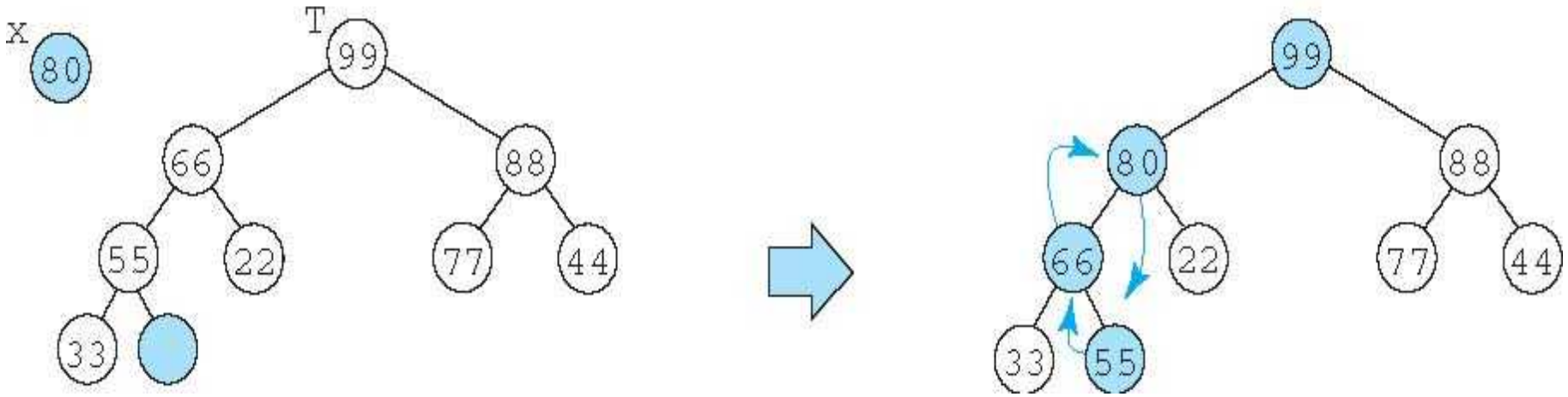
- 우선순위 큐는 BIFO 접근 프로토콜을 유지하는 원소의 컬렉션임
- 연산
  1. Add: 주어진 원소를 큐에 삽입한다.
  2. Best: 큐가 공백이 아니면, 최고 우선순위를 가지는 원소를 리턴한다.
  3. RemoveBest: 큐가 공백이 아니면, 최고 우선순위를 가지는 원소를 삭제해서 리턴한다.
  4. Size: 큐에 있는 원소의 수를 리턴한다.
- PriorityQueue ADT의 인터페이스



PriorityQueue 인터페이스

```
1 public interface PriorityQueue {
2       public void add(Object object);
3           // POSTCONDITION: the given object is in this queue;
5       public Object best();
6           // RETURN: the highest priority element in this queue;
7           // PRECONDITION: this queue is not empty;
9       public Object removeBest();
10          // RETURN: the highest priority element in this queue;
11          // PRECONDITION: this queue is not empty;
12          // POSTCONDITION: the returned object is not in thisqueue;
14      public int size();
15          // RETURN: the number of elements in this queue;
16 }
```

우선순위 큐에 80을 삽입



삽입 알고리즘

- 우선순위 큐를 위한 삽입 알고리즘
  - 입력 : 완전 이진 트리 T와 새로운 노드 x.
  - 후조건 : x가 T에 삽입됨.

1. 만일 *T*가 공백이면, *T*를 다시 *x*를 포함하는 단독 트리로 만들고, 리턴.
2. *T*의 마지막에 새로운 노드 z=x를 추가.
3. *y=z.parent*로 설정.
4. *y.key<x.key* 인 동안, 단계 5-7을 수행.
5. *z.key=y.key*로 설정.
6. *z=y*로 설정.
7. 만일 *y*가 루트가 아니라면, 단계 8을 수행.
8. *y=y.parent*로 설정.
9. *z.key=x.key*로 설정하고 리턴.

HeapPriorityQueue 클래스 (1)

- LISTING 14.4: A HeapPriorityQueue Class

```
1 public class HeapPriorityQueue implements PriorityQueue {
2     private static final int CAPACITY = 100;
3     private Comparable[] a;
4     private int size;
5
6     public HeapPriorityQueue() {
7         this(CAPACITY);
8     }
9
10    public HeapPriorityQueue(int capacity) {
11        a = new Comparable[capacity];
12    }
13
```

```
14   public void add(Object object) {
15      if (!(object instanceof Comparable))
16                throw new IllegalArgumentException();
17      Comparable x = (Comparable)object;
18      if (size == a.length) resize();
19      int i = size++;
20      while (i > 0) {
21            int j = i;
22            i = (i-1)/2;
23            if (a[i].compareTo(x) >= 0) {
24                  a[j] = x;       return;     }
27            a[j] = a[i];
28      }
29      a[i] = x;
30   }
```

```
32   public Object best() {
33       if (size == 0) throw new
                              java.util.NoSuchElementException();
34       return a[0];
35    }
37    public Object remove() {
38        Object best = best();
39        a[0] = a[--size];
40        heapify(0, size);
41        return best;
42    }
44    public int size() {
45      return size;
46   }
```

```
48   public String toString() {
49      if (size == 0) return "{}";
50      StringBuffer buf = new StringBuffer("{" + a[0]);
51      for (int i = 1; i < size; i++)
52                buf.append("," + a[i]);
53      return buf + "}";
54   }
56   private void heapify(int i, int n) {
57        Comparable ai = a[i];
58        while (i < n/2) {
59           int j = 2*i+1;
60           if (j+1 < n && a[j+1].compareTo(a[j]) > 0) ++j;
61           if (a[j].compareTo(ai) <= 0) break;
62           a[i] = a[j];
63           i = j;
64        }
65        a[i] = ai;
66   }
```

```
68    private void resize() {
69       Comparable[] aa = new Comparable[2*a.length];
70       System.arraycopy(a, 0, aa, 0, a.length);
71       a = aa;
72    }
73}
```

HeapPriorityQueue 클래스의 테스팅 (1)

```
1 public class TestHeapPriorityQueue {
3       public TestHeapPriorityQueue() {
4           PriorityQueue queue = new HeapPriorityQueue();
5           int[] pages = {7,3,2,8,3,4,1,3};
6           for (int i = 0; i < pages.length; i++) {
7                   queue.add(new PrintJob(null, pages[i]));
8                   System.out.println("q: " + queue);
9           }
10          while (queue.size() > 0) {
11                  Systeout.println("queue.remove(): " + queue.remove());
12                  System.out.println("q: " + queue);
13          }
14      }
16      public static void main(String[] args) {
17          new TestHeapPriorityQueue();
18      }
19 }
```

```
21 class PrintJob implements Comparable {
22        private java.io.File file;
23        private int pages;
24        private String id;
25        private static int n = 100;
27        public PrintJob(java.io.File file, int pages) {
28              this.file = file;
29              this.pages = pages;
30              this.id = "ID" + n++;
31        }
33        public int compareTo(Object object) {
34              if (!(object instanceof PrintJob))
35                    throw new IllegalArgumentException();
36              PrintJob that = (PrintJob)object;
37              return that.pages - this.pages;      //페이지 크기가 작을 수록 우선순위가 높다
38        }
40        public String toString() {
41             return id + "(" + pages + ")";
42         }
43 }
```

출력 결과

q: {ID100(7)}
q: {ID101(3),ID100(7)}
q: {ID102(2),ID100(7),ID101(3)}
q: {ID102(2),ID100(7),ID101(3),ID103(8)}
q: {ID102(2),ID104(3),ID101(3),ID103(8),ID100(7)}
q: {ID102(2),ID104(3),ID101(3),ID103(8),ID100(7),ID105(4)}
q: {ID106(1),ID104(3),ID102(2),ID103(8),ID100(7),ID105(4),ID101(3)}
q: {ID106(1),ID104(3),ID102(2),ID107(3),ID100(7),ID105(4),ID101(3),ID103(8)}
queue.remove(): ID106(1)
q: {ID102(2),ID104(3),ID101(3),ID107(3),ID100(7),ID105(4),ID103(8)}
queue.remove(): ID102(2
q: {ID104(3),ID107(3),ID101(3),ID103(8),ID100(7),ID105(4)}
queue.remove(): ID104(3)
q: {ID107(3),ID105(4),ID101(3),ID103(8),ID100(7)}queue.remove(): ID107(3)
q: {ID101(3),ID105(4),ID100(7),ID103(8)}queue.remove(): ID101(3)
q: {ID105(4),ID103(8),ID100(7)}queue.remove(): ID105(4)
q: {ID100(7),ID103(8)}queue.remove(): ID100(7)
q: {ID103(8)}queue.remove(): ID103(8)
q: {}