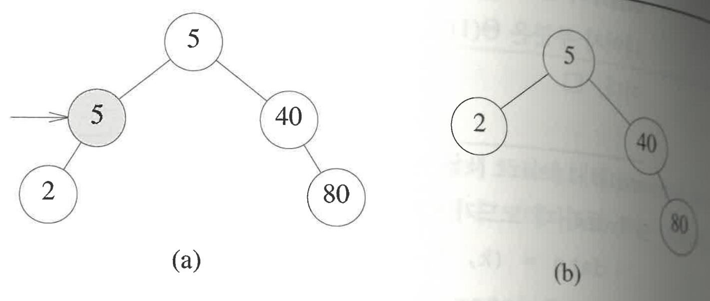

그림 5.31 이원 탐색 트리에서의 삭제

2개의 자식을 가진 비리프 노드를 삭제할 때는 그 원소를 왼쪽 서브트리에서 가장 큰 원소이거나 오른쪽 서브트리에서 가장 작은 원소로 대체한다. 그리고 대체된 서브트리에서 대체한 원소의 삭제 과정을 진행한다. 예로 그림 5.30(a) 트리에서 키 30을 가진 원소를 삭제한다면 이것을 왼쪽 서브트리에서 가장 큰 원소인 5 또는 오른쪽 서브트리에서 가장 작은 원소인 40 중 하나와 대체해야 한다. 왼쪽 서브트리에서 가장 큰 원소로 대체시킨다고 가정하면, 5가 트리의 루트로 이동하여 그림 5.31(a)의 트리로 된다. 다음에는 두 번째 5를 삭제해야 하는데, 이 노드는 하나의 자식만 가지고 있으므로 부모의 포인터가 이 노드의 자식을 가리키도록 변경하기만 하면 된다. 그 결과 그림 5.31(b)의 트리가 얻어진다. 왼쪽 서브트리에서 가장 큰 원소와 오른쪽 서브트리에서 가장 작은 원소 중 어느 것으로 대체해도, 대체되는 노드는 항상 차수가 1이거나 0인 노드라는 사실을 증명할 수 있다. 그러므로 대체된 노드를 삭제하는 것은 아주 쉽다. 삭제 함수를 작성하는 것은 연습문제로 남겨놓기로 한다. 높이가 h인 탐색 트리에서 삭제가 $O(h)$ 시간 내에 수행된다는 것은 분명하다.

5.7.5 이원 탐색 트리의 조인과 분할

탐색·삽입·삭제가 이원 탐색 트리에서 가장 빈번하게 수행되지만, 다음과 같은 추가적인 연산들도 어떤 응용에서는 유용하다.

(a) *threeWayJoin(small, mid, big)*: 이 연산은 이원 탐색 트리 *small*과 *big*에 있는 쌍들과 쌍 *mid*로 구성되는 하나의 이원 탐색 트리를 생성한다. *small*에 있는 키는 모두 *mid.key*보다 작고 *big*에 있는 키는 모두 *mid.key*보다 큰 것으로 가정한다. 이 조인 연산이 종료되면 *small*과 *big*은 공백이 된다.

(b) *twoWayJoin(small, big)*: 이 연산은 두 이원 탐색 트리 *small*과 *big*을 조인하여 이 *small*과 *big*에 있는 모든 쌍들을 포함하는 하나의 이원 탐색 트리를 생성한다. *small*에 있는 모든 키들은 *big*에 있는 모든 키들보다 작고 이 조인 연산이 종료되면 *small*과 *big*은 공백이 된다고 가정한다.

(c) *split(theTree, k, small, mid, big)*: 이원 탐색 트리 *theTree*를 세 부분으로 분할한다. *small*은 *k*보다 작은 키를 가지고 있는 *theTree*의 모든 쌍을 포함하는 이원 탐색 트리이다. 만일 *theTree*가 키 *k*를 가진 쌍을 포함하고 있으면 이 쌍은 참조 매개변수 *mid*에 반환된다. *big*은 *k*보다 큰 키를 가지고 있는 *theTree*의 모든 쌍을 포함하는 이원 탐색 트리이다. 이 분할 연산이 종료되면 *theTree*는 공백이 된다. *theTree*에 키가 *k*인 쌍이 없는 경우에는 *mid.key*는 −1로 설정된다.(이는 −1이 사전 쌍에 대한 유효한 키가 아니라고 가정한 것이다.)

*threeWayJoin*은 수행하기가 특별히 쉽다. 새로운 노드를 하나 얻어 데이타 필드를 *mid*로, 왼쪽 자식 포인터는 *small*로, 오른쪽 자식 포인터는 *big*으로 설정하면 된다. 이 새로운 노드는 생성되는 이원 탐색 트리의 루트가 된다. 끝으로 *small*과 *big*은 *NULL*로 설정된다. 이 연산에 필요한 시간은 $O(1)$이며 새로운 트리의 높이는 $\max\{height(small), height(big)\} + 1$이 된다.

*twoWayJoin*을 고려해보자. 만일 *small* 또는 *big*이 공백이면, 결과는 공백이 아닌 것이 바로 이원 탐색 트리가 된다. 어느 것도 공백이 아닌 경우 먼저 *small*에서 가장 큰 키 값을 가진 *mid* 쌍을 삭제한다. 이 이원 탐색 트리를 *small′*라 하자. 연산을 완수하기 위해 *threeWayJoin(small′, mid, big)*를 수행하면 된다. *twoWayJoin*을 수행하는 데 필요한 총 시간은 $O(height(small))$이고, 결과 트리의 높이는 $\max\{height(small'), height(big)\} + 1$이 된다. 만일 각 트리가 높이를 유지하는 경우 실행 시간은 $O(\min\{height(small), height(big)\})$이 될 수 있다. 즉, *small*의 높이가 *big*의 높이보다 크지 않으면 *small*에서 제일 큰 키를 가진 쌍을 삭제하고, 그렇지 않으면 *big*에서 제일 작은 키를 가진 쌍을 삭제한다. 그 뒤에 *threeWayJoin* 연산을 수행한다.

*split*을 하기 위해, 먼저 루트에서 분할할 때(즉, k = *theTree*→*data.key*일 때) 다음을 관찰할 수 있다. 이 경우 *small*은 *theTree*의 왼쪽 서브트리이며, *mid*는 루트에 있는 쌍이고, *big*은 *theTree*의 오른쪽 서브트리가 된다. 만일 k가 루트의 키보다 작으면 루트와 오른쪽 서브트리는 *big*에 속하게 되고, 만일 k가 루트의 키보다 크면 루트와 왼쪽 서브트리는 *small*에 속하게 된다. 이러한 관찰을 이용하여 키 k를 가진 쌍을 찾기 위해 탐색 트리 *theTree*를 따라 밑으로 이동하면서 분할을 수행한다. 또 밑으로 이동하면서 2개의 탐색 트리 *small*과 *big*을 구성한다. *theTree*를 분할하는 함수는 프로그램 5.18에 제