Mastering Data Structures and Algorithms: A Practical Approach

Priority Queues, Hashing

By Dr. Juan C. Gomez
Fall 2018

Santa Cruz extension in Silicon Valley

Overview

- Priority Queues or Heaps
  - Basic operations
  - Binary Heap
    - Insert Operation
    - Delete Operation
    - Other Operations
  - Applications
  - d-Heaps
  - Leftist Heaps
  - Skew Heaps
  - Binomial Queues
  - Priority Queues in the Standard Library
  - Examples

Overview

- Hashing
  - Hash Function
  - Rehashing
  - Universal Hashing
  - Extendible Hashing
  - Examples

Heaps: Basic Operations

- Min-Heap:

A priority queue is a data structure that allows at least the following two operations: insert, which does the obvious thing; and `deleteMin`, which finds, returns, and removes the minimum element in the priority queue. The `insert` operation is the equivalent of `enqueue`, and `deleteMin` is the priority queue equivalent of the queue's `dequeue` operation.

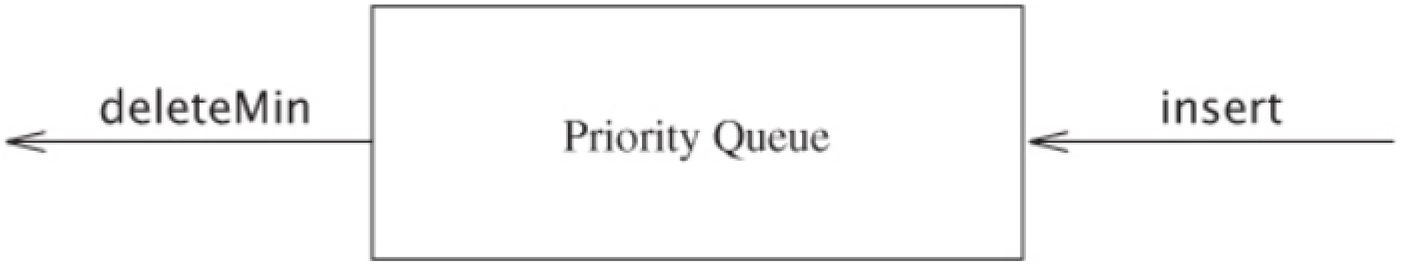


Figure 6.1 Basic model of a priority queue

Heaps: Basic Operations

- Min-Heap: Operations time complexity:
  - Insert: O(log(n))
  - deleteMin: O(log(n))
  - peekMin: O(1)
- Max-Heap:
  - Insert: O(log(n))
  - deleteMax: O(log(n))
  - peekMax: O(1)

Heaps: Binary Heap

A heap is a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right. Such a tree is known as a **complete binary tree.** Figure 6.2 shows an example.

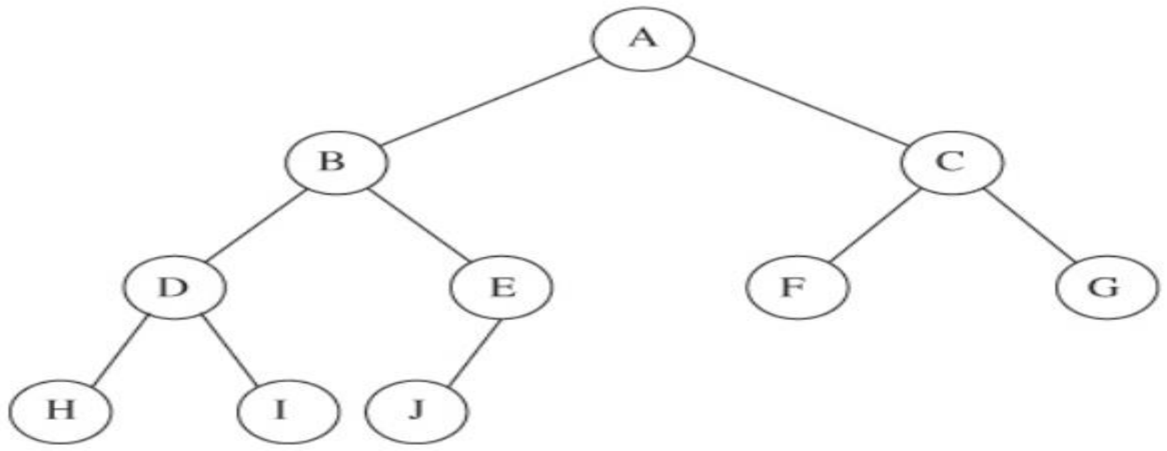


Figure 6.2 A complete binary tree

	A	B	C	D	E	F	G	H	I	J			
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Figure 6.3 Array implementation of complete binary tree

Heaps: Binary Heap

- Heap Order-Property:
  - Min-Heap:
    - Every node is smaller than its children
  - Max-Heap:
    - Every node is larger than its children
  - This order property must be preserved through insert and delete operations

Heaps: Binary Heap

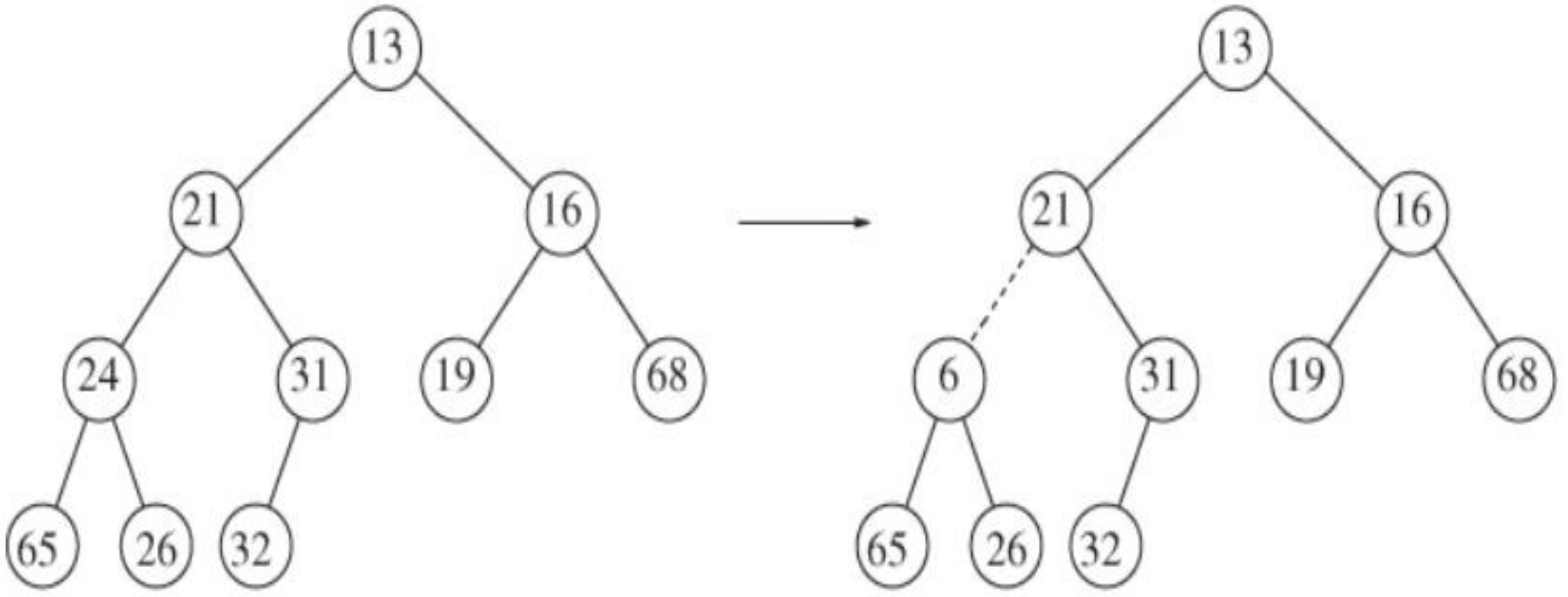
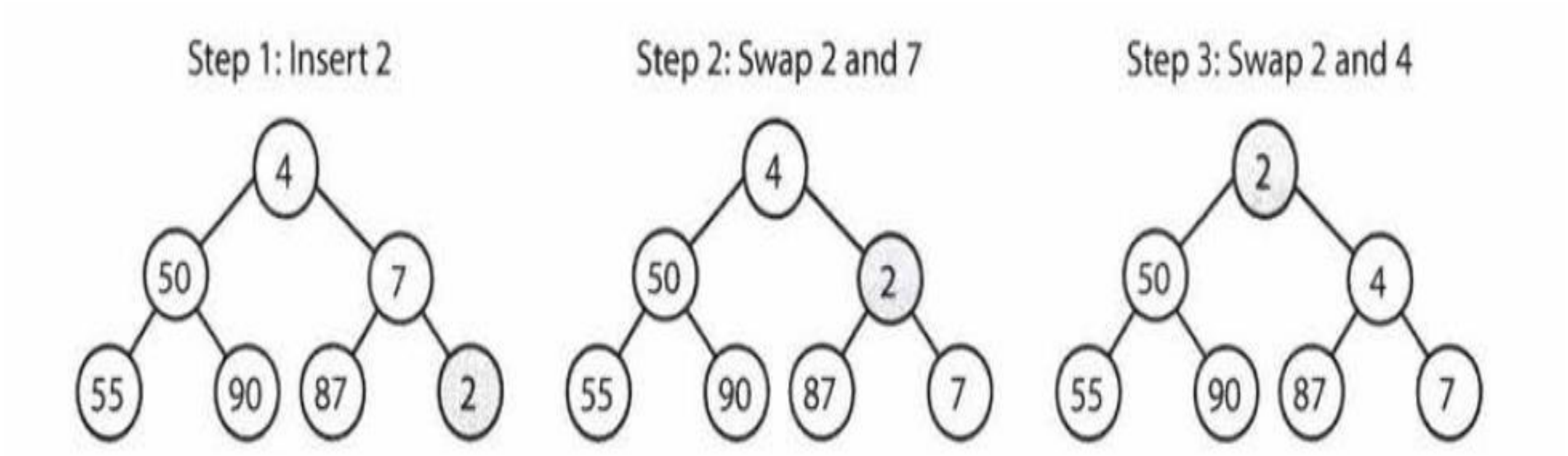


Figure 6.5 Two complete trees (only the left tree is a heap)

Heaps: Insert Operation



This takes O(log n) time, where n is the number of nodes in the heap.

Heaps: Insert Operation

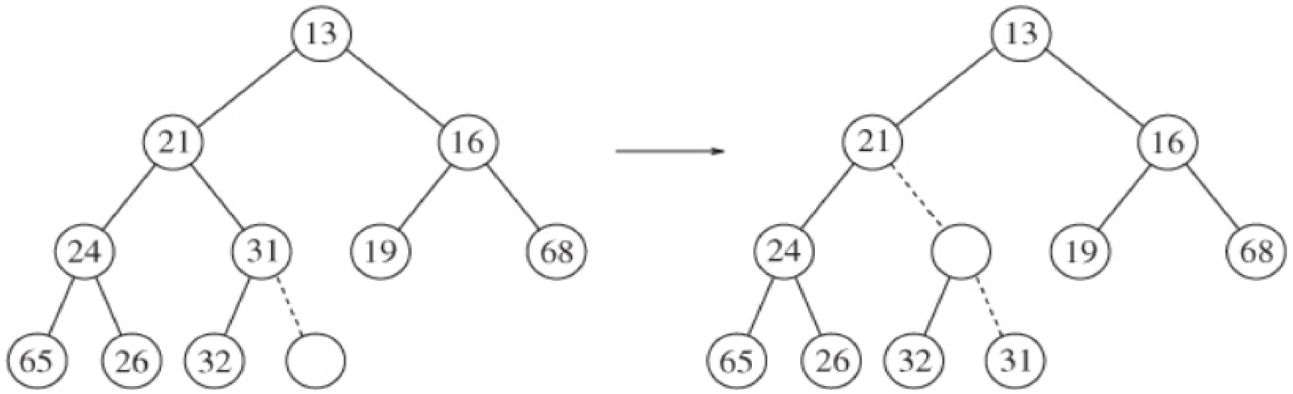


Figure 6.6 Attempt to insert 14: creating the hole and bubbling the hole up

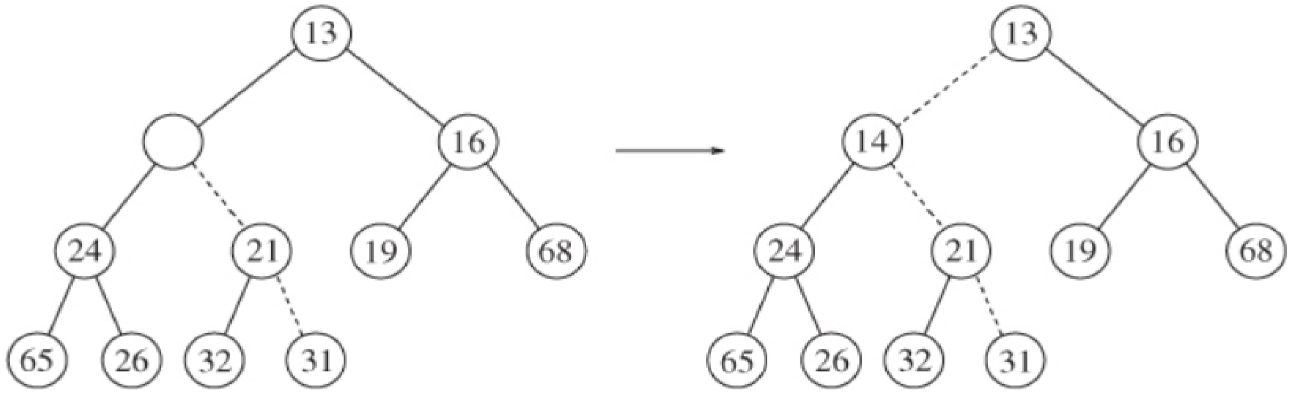
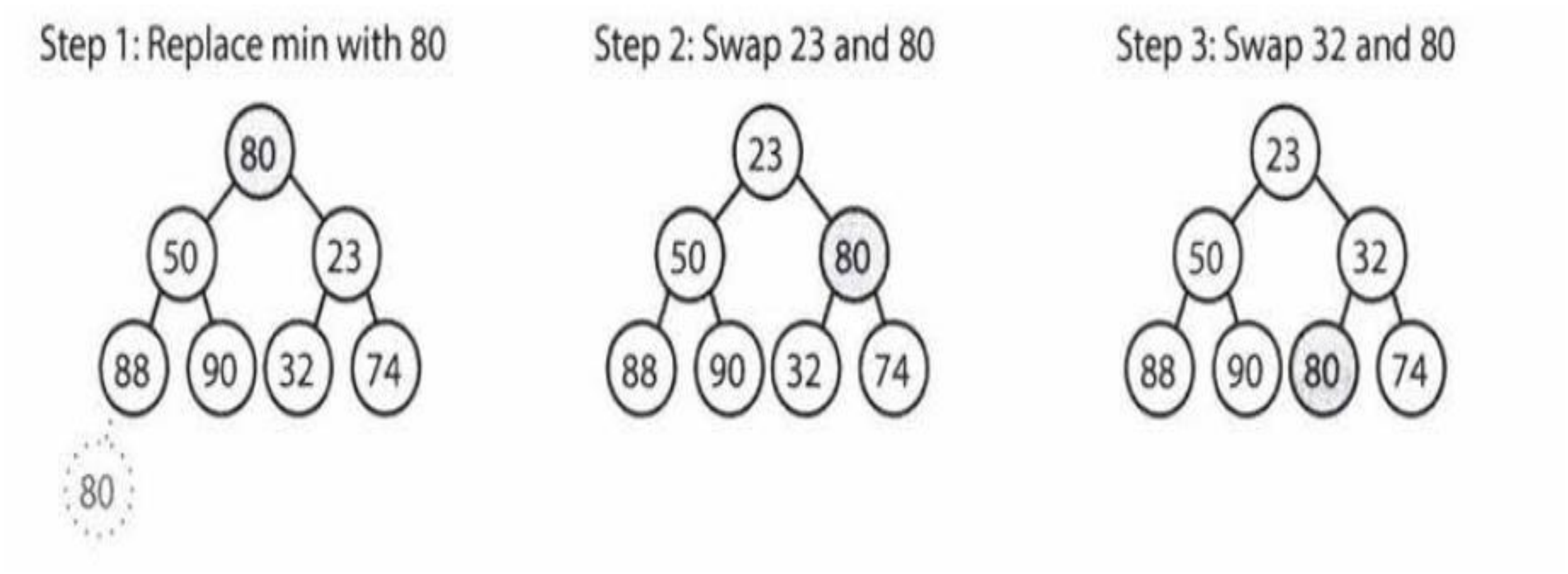


Figure 6.7 The remaining two steps to insert 14 in previous heap

Heaps: Insert Operation

```
    /**
     * Insert into the priority queue, maintaining heap order.
     * Duplicates are allowed.
     * @param x the item to insert.
     */
    public void insert( AnyType x )
    {
        if( currentSize == array.length - 1 )
            enlargeArray( array.length * 2 + 1 );

            // Percolate up
        int hole = ++currentSize;
        for( array[ 0 ] = x; x.compareTo( array[ hole / 2 ] ) < 0; hole /= 2 )
            array[ hole ] = array[ hole / 2 ];
        array[ hole ] = x;
    }
```

Heaps: Delete Operation



Heaps: Delete Operation

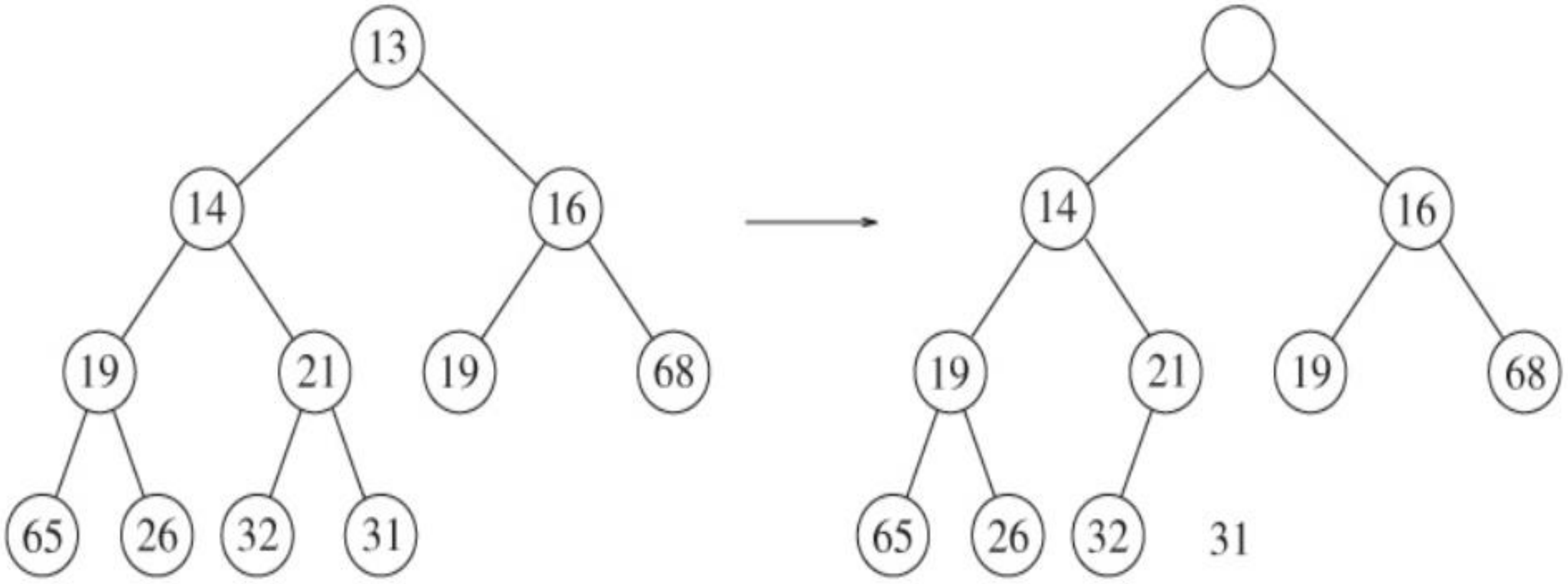


Figure 6.9 Creation of the hole at the root

Heaps: Delete Operation

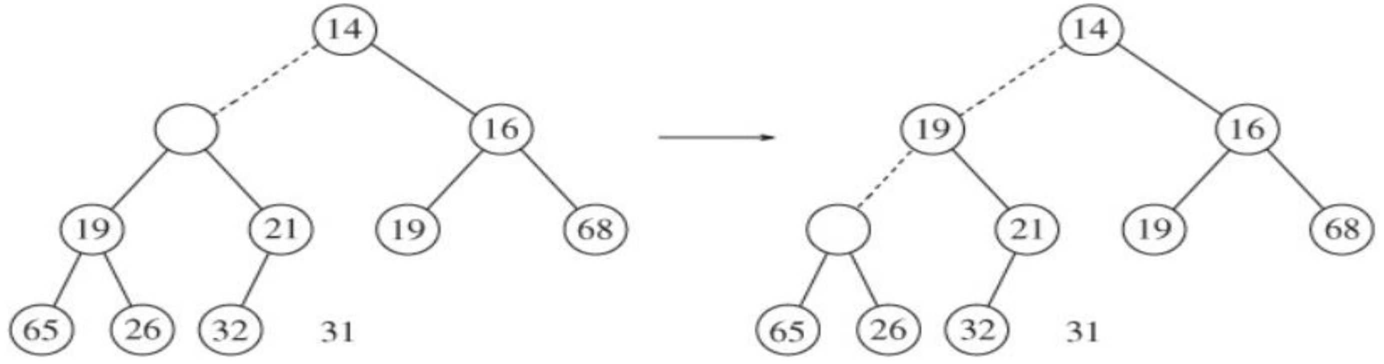


Figure 6.10 Next two steps in `deleteMin`

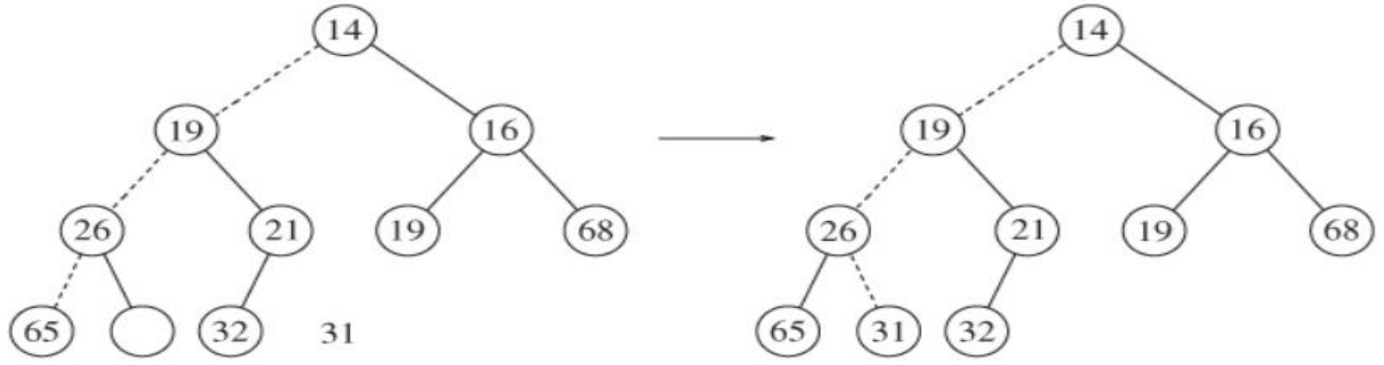


Figure 6.11 Last two steps in `deleteMin`

Heaps: Delete Operation

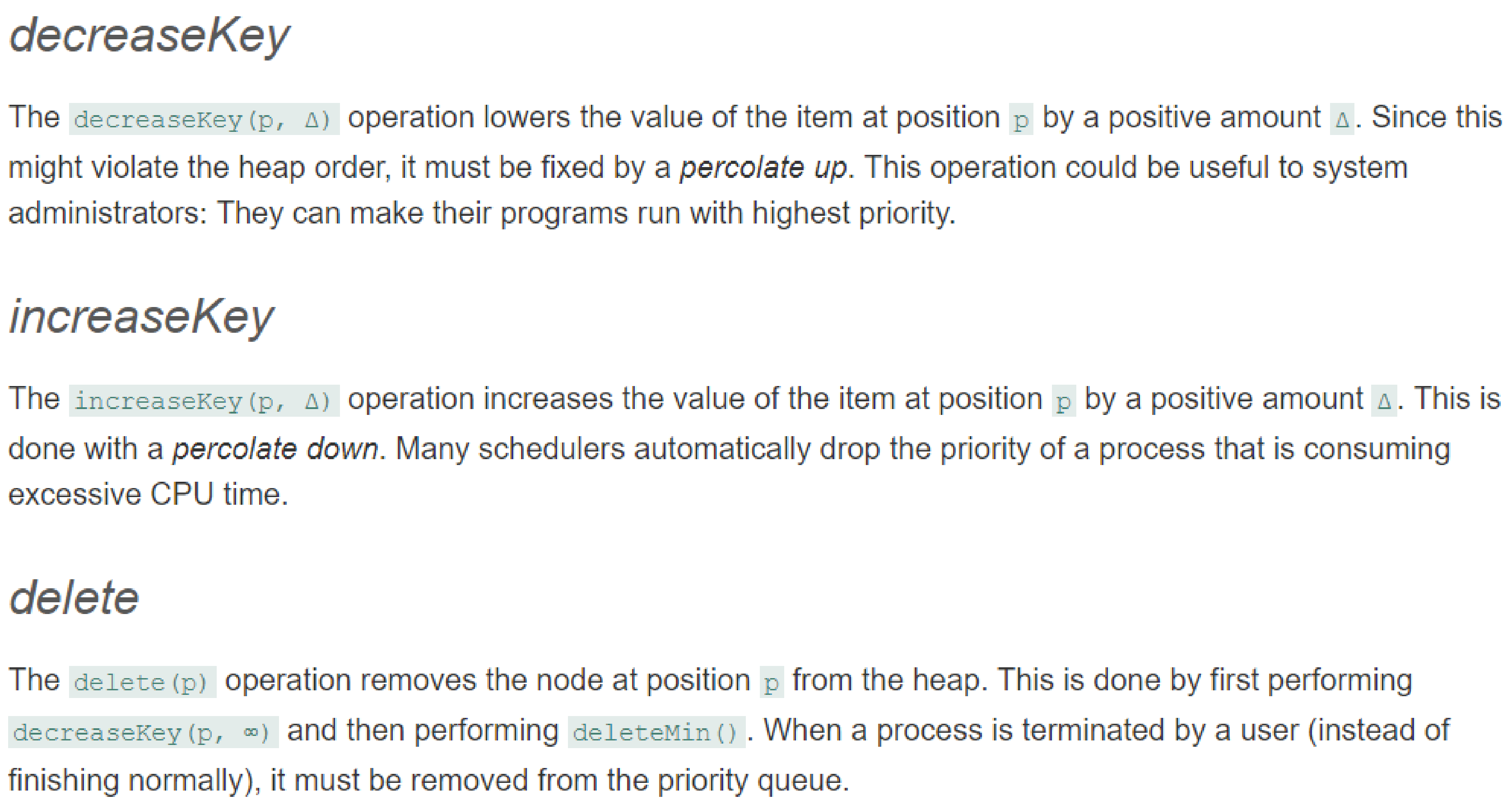
```
/**
 * Remove the smallest item from the priority queue.
 * @return the smallest item, or throw UnderflowException, if empty.
 */
public AnyType deleteMin( )
{
    if( isEmpty( ) )
        throw new UnderflowException( );

    AnyType minItem = findMin( );
    array[ 1 ] = array[ currentSize-- ];
    percolateDown( 1 );

    return minItem;
}
```

Heaps: Delete Operation

```java
    /**
     * Internal method to percolate down in the heap.
     * @param hole the index at which the percolate begins.
     */
    private void percolateDown( int hole )
    {
        int child;
        AnyType tmp = array[ hole ];

        for( ; hole * 2 <= currentSize; hole = child )
        {
            child = hole * 2;
            if( child != currentSize &&
                    array[ child + 1 ].compareTo( array[ child ] ) < 0 )
                child++;
            if( array[ child ].compareTo( tmp ) < 0 )
                array[ hole ] = array[ child ];
            else
                break;
        }
        array[ hole ] = tmp;
    }
```

Santa Cruz
UC extension
in Silicon Valley

Heaps: Other Operations



decreaseKey

The `decreaseKey(p, Δ)` operation lowers the value of the item at position `p` by a positive amount `Δ`. Since this might violate the heap order, it must be fixed by a *percolate up*. This operation could be useful to system administrators: They can make their programs run with highest priority.

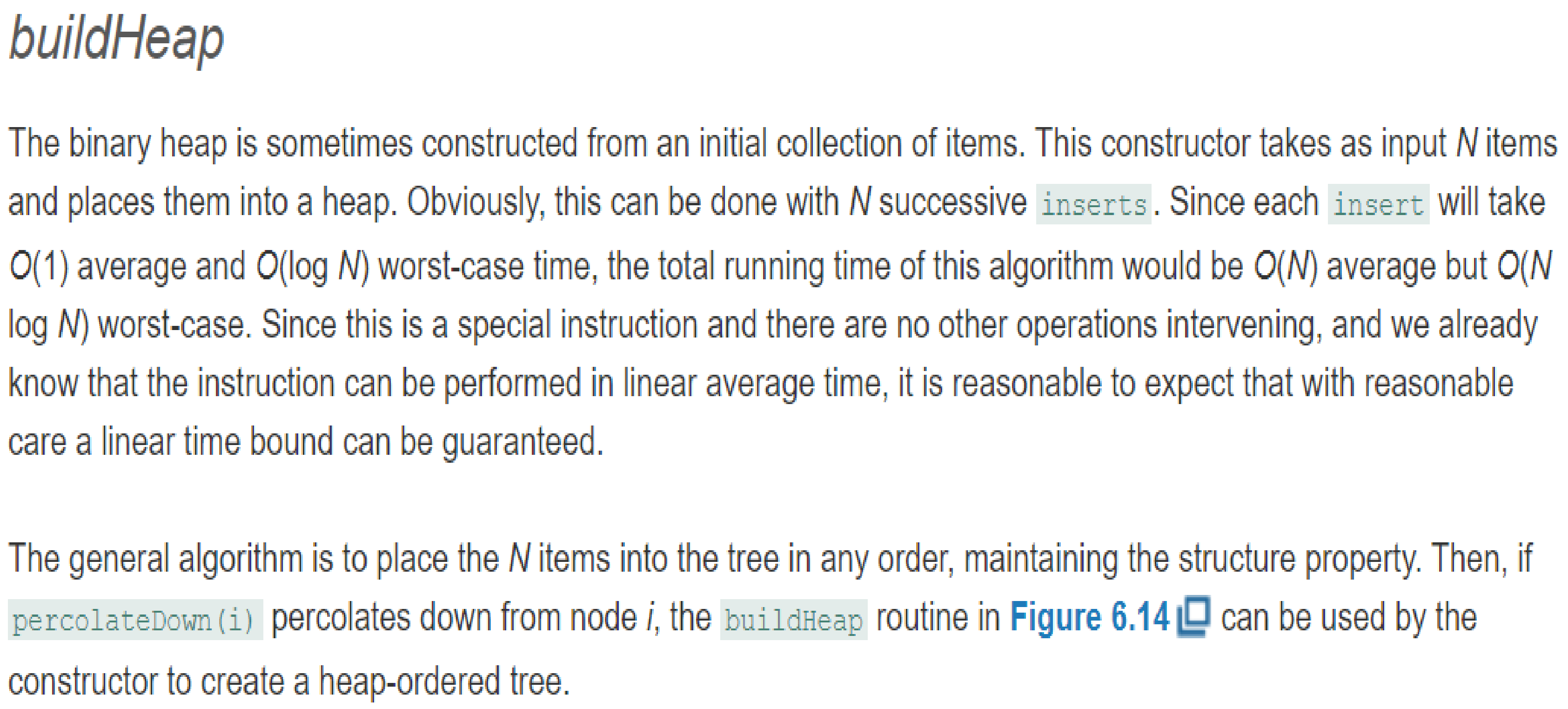
increaseKey

The `increaseKey(p, Δ)` operation increases the value of the item at position `p` by a positive amount `Δ`. This is done with a *percolate down*. Many schedulers automatically drop the priority of a process that is consuming excessive CPU time.

delete

The `delete(p)` operation removes the node at position `p` from the heap. This is done by first performing `decreaseKey(p, ∞)` and then performing `deleteMin()`. When a process is terminated by a user (instead of finishing normally), it must be removed from the priority queue.

Heaps: Other Operations

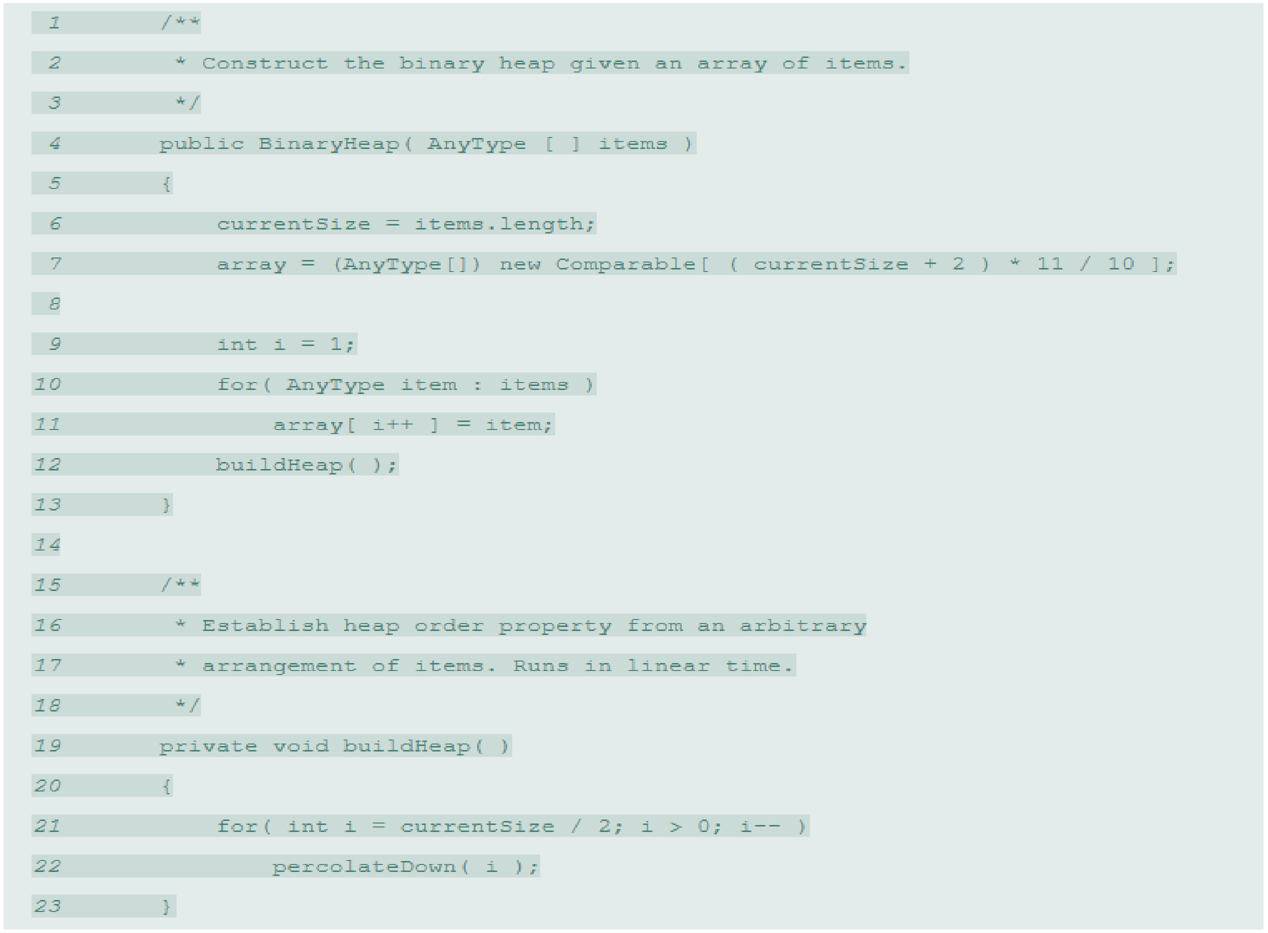


buildHeap

The binary heap is sometimes constructed from an initial collection of items. This constructor takes as input *N* items and places them into a heap. Obviously, this can be done with *N* successive `inserts`. Since each `insert` will take *O*(1) average and *O*(log *N*) worst-case time, the total running time of this algorithm would be *O*(*N*) average but *O*(*N* log *N*) worst-case. Since this is a special instruction and there are no other operations intervening, and we already know that the instruction can be performed in linear average time, it is reasonable to expect that with reasonable care a linear time bound can be guaranteed.

The general algorithm is to place the *N* items into the tree in any order, maintaining the structure property. Then, if `percolateDown(i)` percolates down from node *i*, the `buildHeap` routine in **Figure 6.14** can be used by the constructor to create a heap-ordered tree.

Heaps: Other Operations



```
    /**
     * Construct the binary heap given an array of items.
     */
    public BinaryHeap( AnyType [ ] items )
    {
        currentSize = items.length;
        array = (AnyType[]) new Comparable[ ( currentSize + 2 ) * 11 / 10 ];

        int i = 1;
        for( AnyType item : items )
            array[ i++ ] = item;
        buildHeap( );
    }

    /**
     * Establish heap order property from an arbitrary
     * arrangement of items. Runs in linear time.
     */
    private void buildHeap( )
    {
        for( int i = currentSize / 2; i > 0; i-- )
            percolateDown( i );
    }
```

Heaps: Applications

- Selection Problem
- Event Simulation

Heaps: d-Heaps

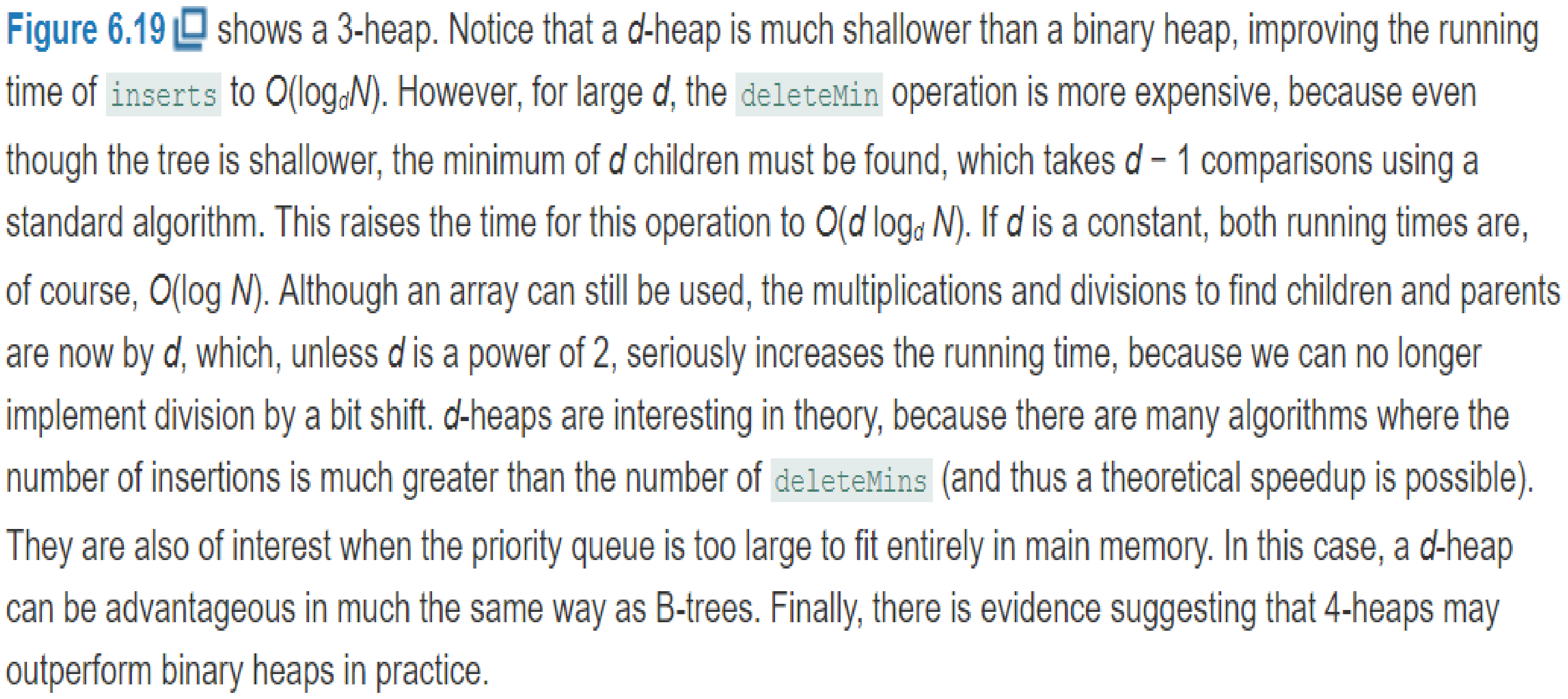


Figure 6.19 shows a 3-heap. Notice that a *d*-heap is much shallower than a binary heap, improving the running time of `inserts` to $O(\log_d N)$. However, for large *d*, the `deleteMin` operation is more expensive, because even though the tree is shallower, the minimum of *d* children must be found, which takes $d - 1$ comparisons using a standard algorithm. This raises the time for this operation to $O(d \log_d N)$. If *d* is a constant, both running times are, of course, $O(\log N)$. Although an array can still be used, the multiplications and divisions to find children and parents are now by *d*, which, unless *d* is a power of 2, seriously increases the running time, because we can no longer implement division by a bit shift. *d*-heaps are interesting in theory, because there are many algorithms where the number of insertions is much greater than the number of `deleteMins` (and thus a theoretical speedup is possible). They are also of interest when the priority queue is too large to fit entirely in main memory. In this case, a *d*-heap can be advantageous in much the same way as B-trees. Finally, there is evidence suggesting that 4-heaps may outperform binary heaps in practice.

Heaps: d-Heaps

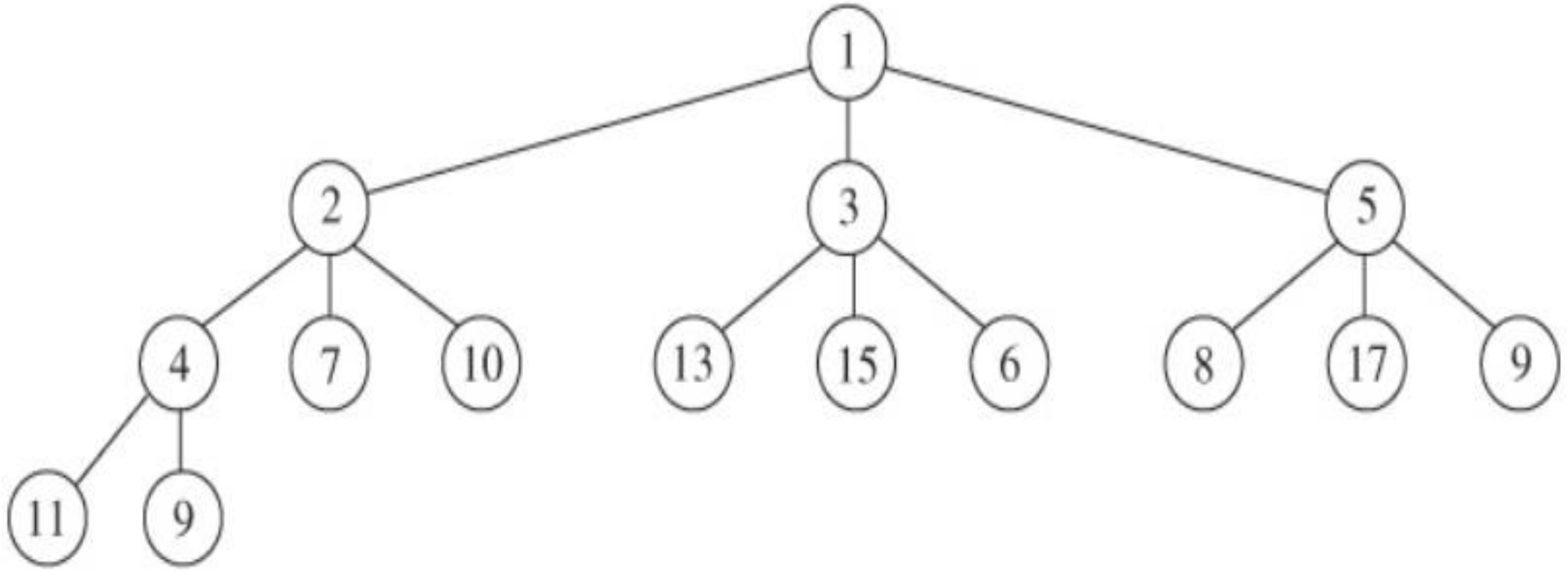
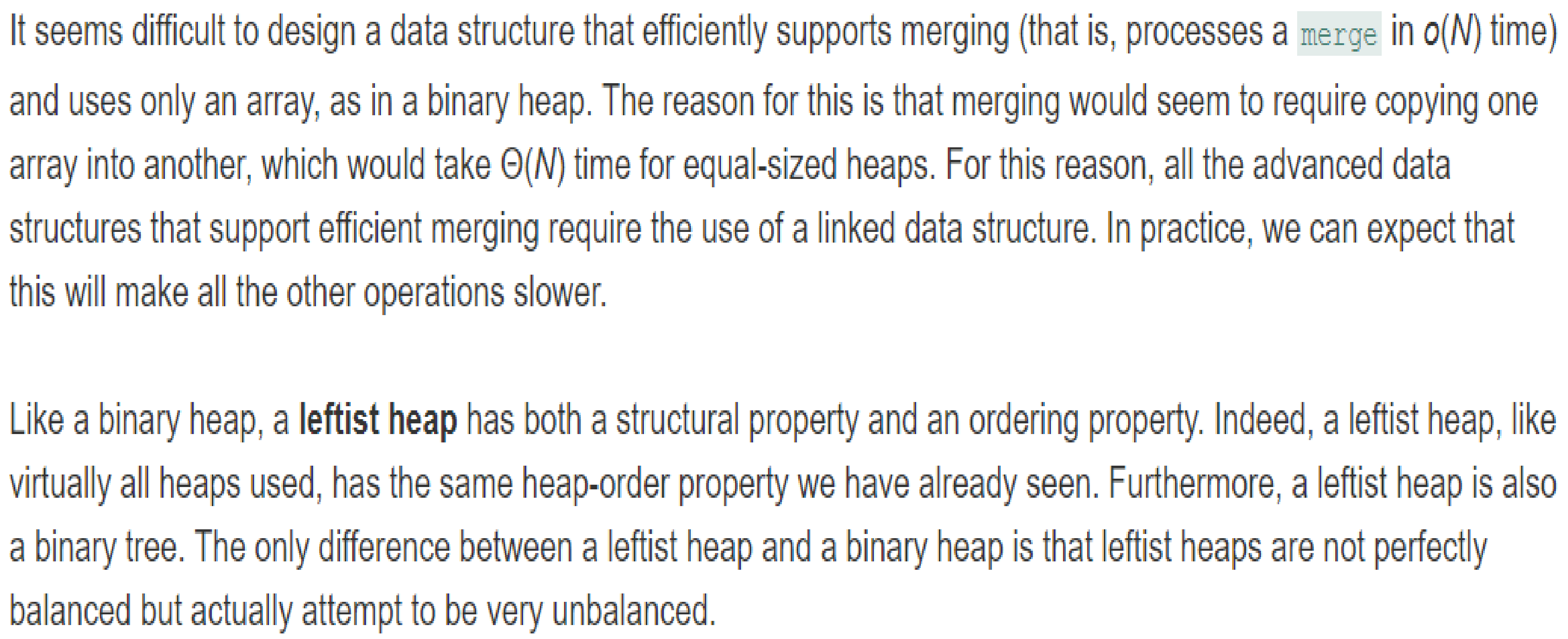


Figure 6.19 A *d*-heap

Heaps: Leftist Heaps



It seems difficult to design a data structure that efficiently supports merging (that is, processes a `merge` in $o(N)$ time) and uses only an array, as in a binary heap. The reason for this is that merging would seem to require copying one array into another, which would take $\Theta(N)$ time for equal-sized heaps. For this reason, all the advanced data structures that support efficient merging require the use of a linked data structure. In practice, we can expect that this will make all the other operations slower.

Like a binary heap, a **leftist heap** has both a structural property and an ordering property. Indeed, a leftist heap, like virtually all heaps used, has the same heap-order property we have already seen. Furthermore, a leftist heap is also a binary tree. The only difference between a leftist heap and a binary heap is that leftist heaps are not perfectly balanced but actually attempt to be very unbalanced.

Heaps: Leftist Heaps

We define the **null path length**, *npl*(*X*), of any node *X* to be the length of the shortest path from *X* to a node without two children. Thus, the *npl* of a node with zero or one child is 0, while *npl*(`null`) = –1. In the tree in **Figure 6.20**, the null path lengths are indicated inside the tree nodes.

Notice that the null path length of any node is 1 more than the minimum of the null path lengths of its children. This applies to nodes with less than two children because the null path length of `null` is –1.

The leftist heap property is that for every node *X* in the heap, the null path length of the left child is at least as large as that of the right child. This property is satisfied by only one of the trees in **Figure 6.20**, namely, the tree on the left. This property actually goes

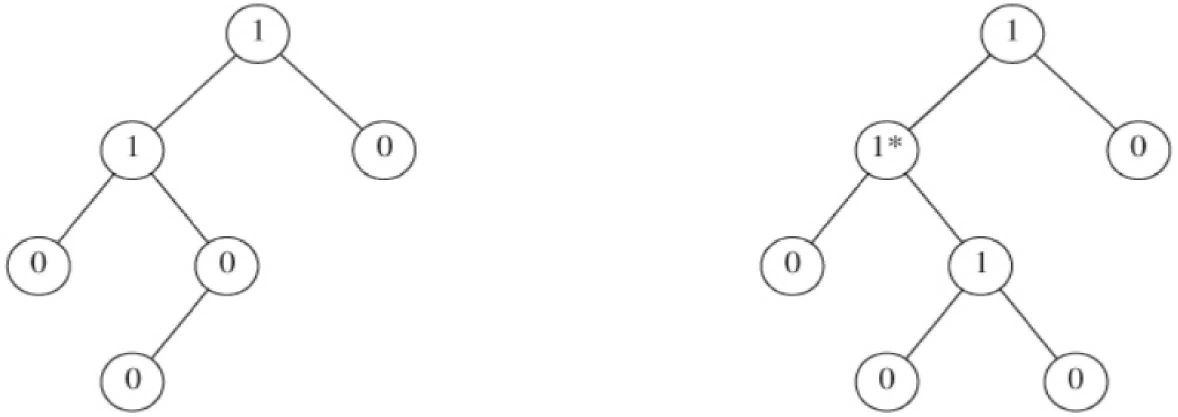


Figure 6.20 Null path lengths for two trees; only the left tree is leftist

Heaps: Leftist Heaps

- Right path of the tree is ensured to have at most floor(Log(N+1)) nodes
- If you do all operations on the right side, you are ensured worst case O(log(n))
- Fundamental operation is merge

Heaps: Leftist Heaps

- Recursive definition of merge:
  - Base Case: if either heap is empty return the other heap.
  - Merge heap with larger root with right subheap of smaller root

Heaps: Leftist Heaps

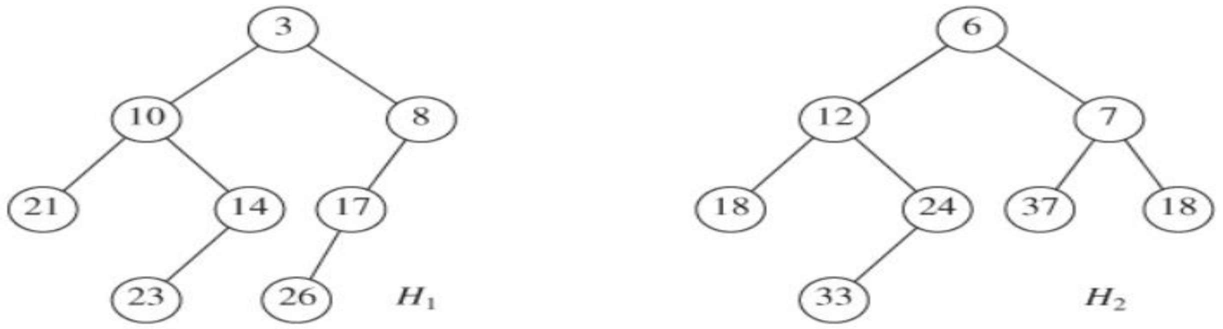


Figure 6.21 Two leftist heaps H_1 and H_2

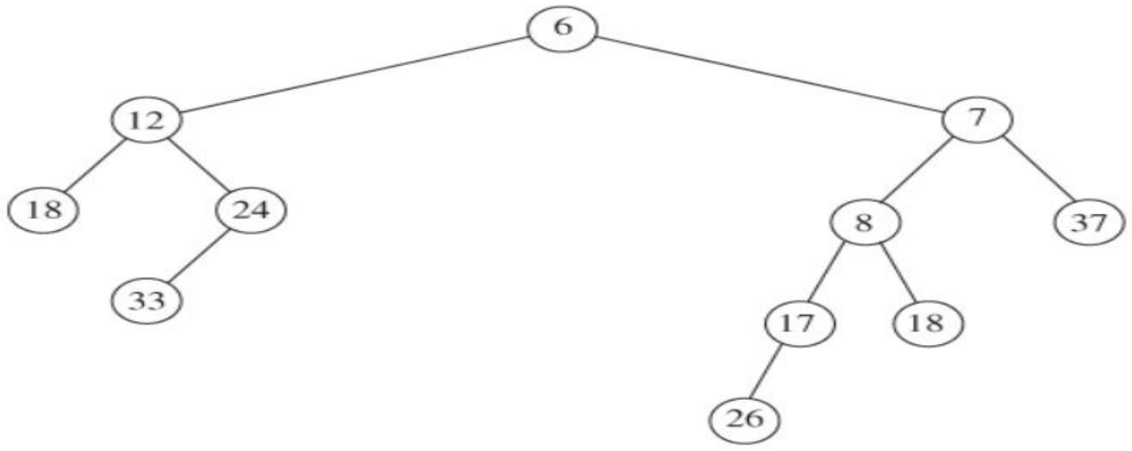


Figure 6.22 Result of merging H_2 with H_1's right subheap

Heaps: Leftist Heaps

- If resulting heap is not leftist (check and maintain null-path-length): swap right and left children.

Heaps: Skew Heaps

- No null-path-length maintain to do.
- Always swap children except for largest node in right path.
- Cost of M consecutive operations is M*log(N): Amortized cost of operations O(log(N))
- Single operation can be O(N)!

Heaps: Binomial Queues

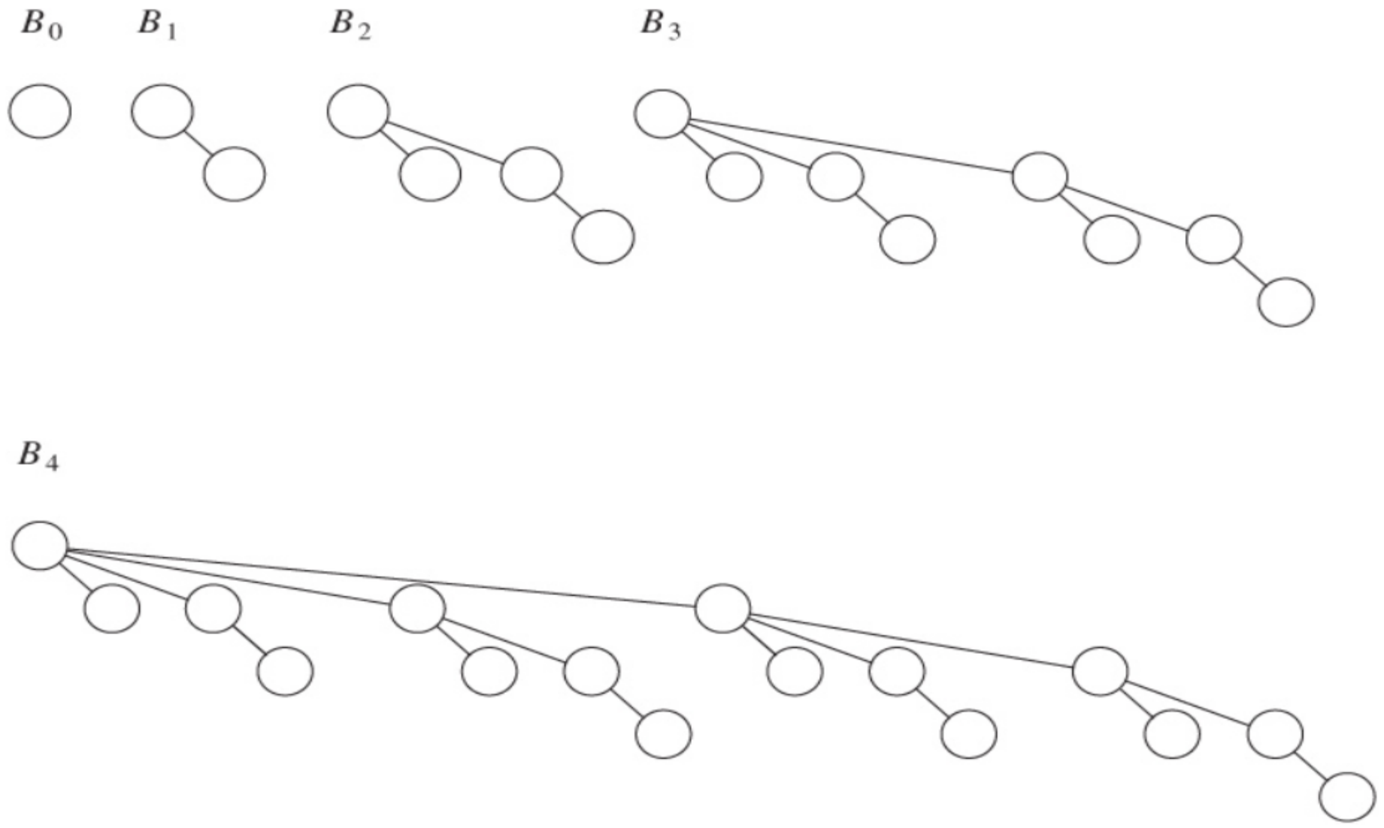


Figure 6.34 Binomial trees B_0, B_1, B_2, B_3, and B_4

Heaps: Binomial Queues

Binomial queues differ from all the priority queue implementations that we have seen in that a binomial queue is not a heap-ordered tree but rather a *collection* of heap-ordered trees, known as a **forest.** Each of the heap-ordered trees is of a constrained form known as a **binomial tree** (the reason for the name will be obvious later). There is at most one binomial tree of every height. A binomial tree of height 0 is a one-node tree; a binomial tree, B_k, of height k is formed by attaching a binomial tree, B_{k-1}, to the root of another binomial tree, B_{k-1}. **Figure 6.34** shows binomial trees B_0, B_1, B_2, B_3, and B_4.

From the diagram we see that a binomial tree, B_k, consists of a root with children B_0, B_1, …, B_{k-1}. Binomial trees of height k have exactly 2^k nodes, and the number of nodes at depth d is the binomial coefficient $\binom{k}{d}$. If we impose heap order on the binomial

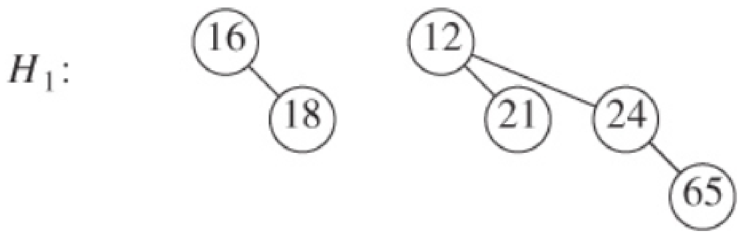


Figure 6.35 Binomial queue H_1 with six elements

Heaps: Binomial Queues

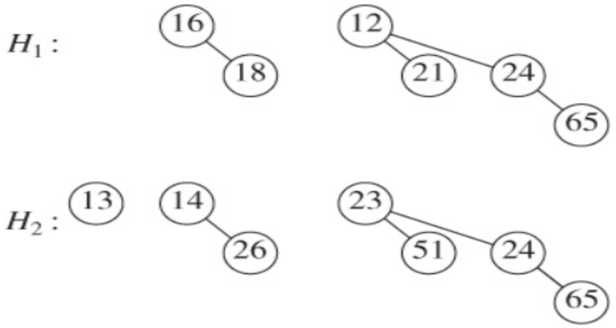


Figure 6.36 Two binomial queues H_1 and H_2

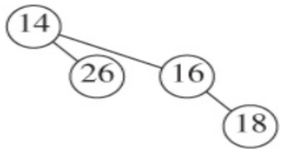


Figure 6.37 Merge of the two B_1 trees in H_1 and H_2

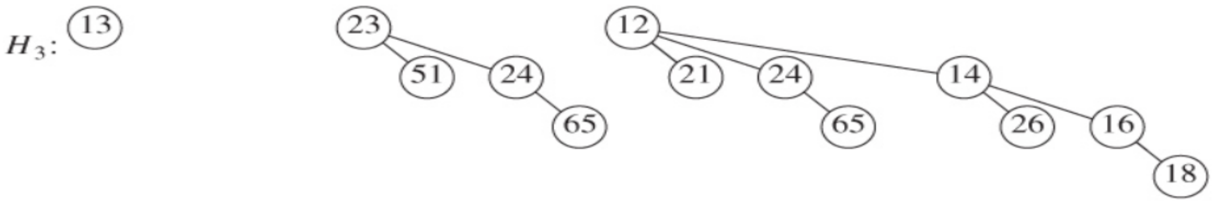


Figure 6.38 Binomial queue H_3: the result of merging H_1 and H_2

Heaps: Binomial Queues

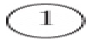


Figure 6.39 After 1 is inserted

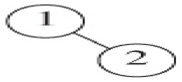


Figure 6.40 After 2 is inserted

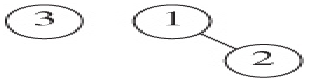


Figure 6.41 After 3 is inserted

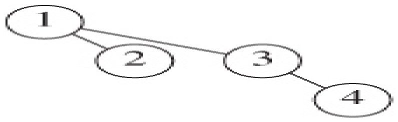


Figure 6.42 After 4 is inserted

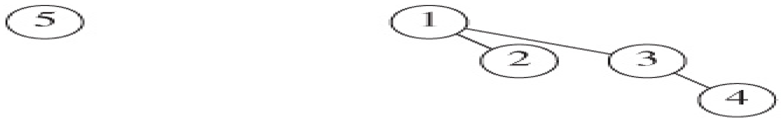


Figure 6.43 After 5 is inserted

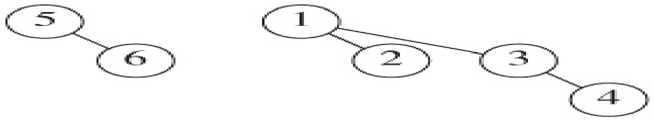


Figure 6.44 After 6 is inserted

Heaps: Binomial Queues

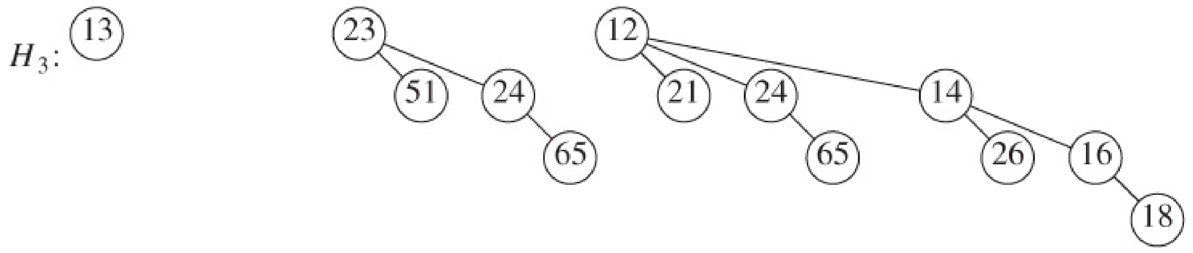


Figure 6.46 Binomial queue H_3

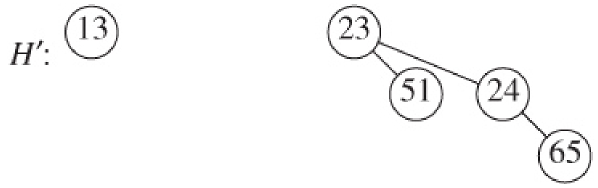


Figure 6.47 Binomial queue H'', containing all the binomial trees in H_3 except B_3

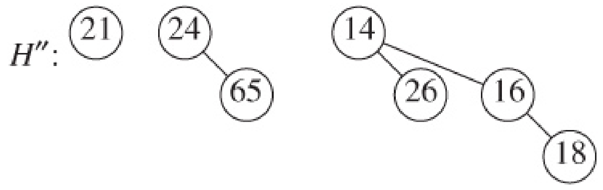
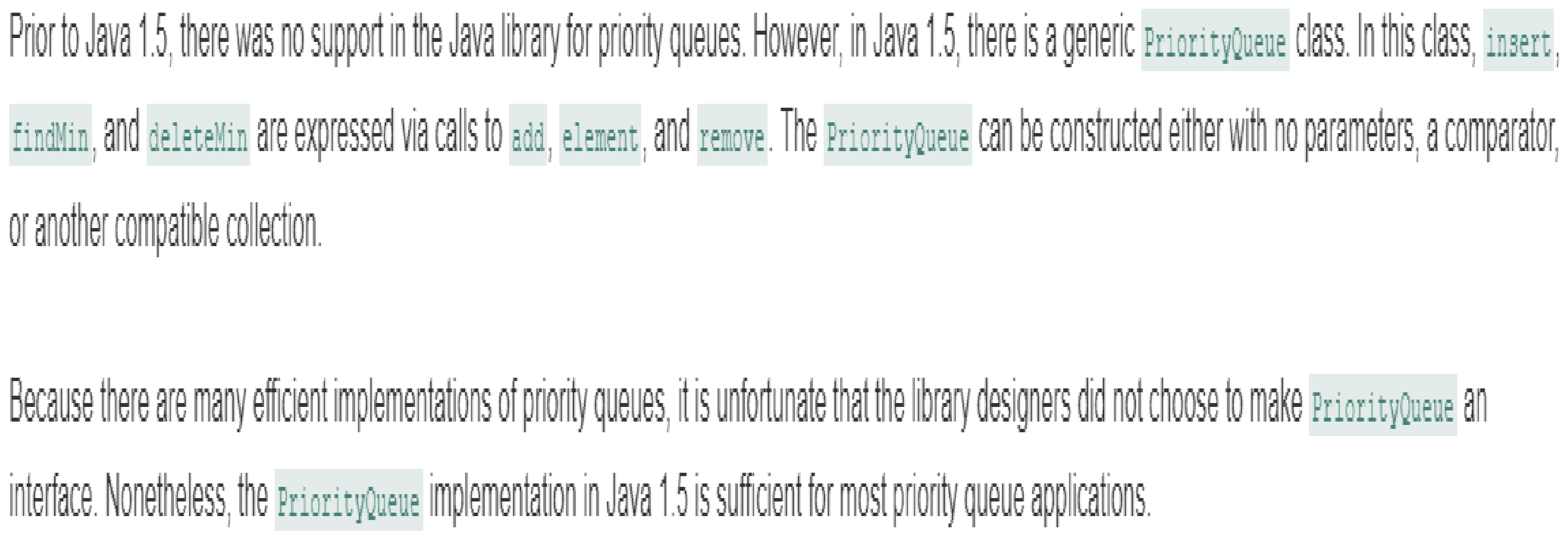


Figure 6.48 Binomial queue H'': B_3 with 12 removed

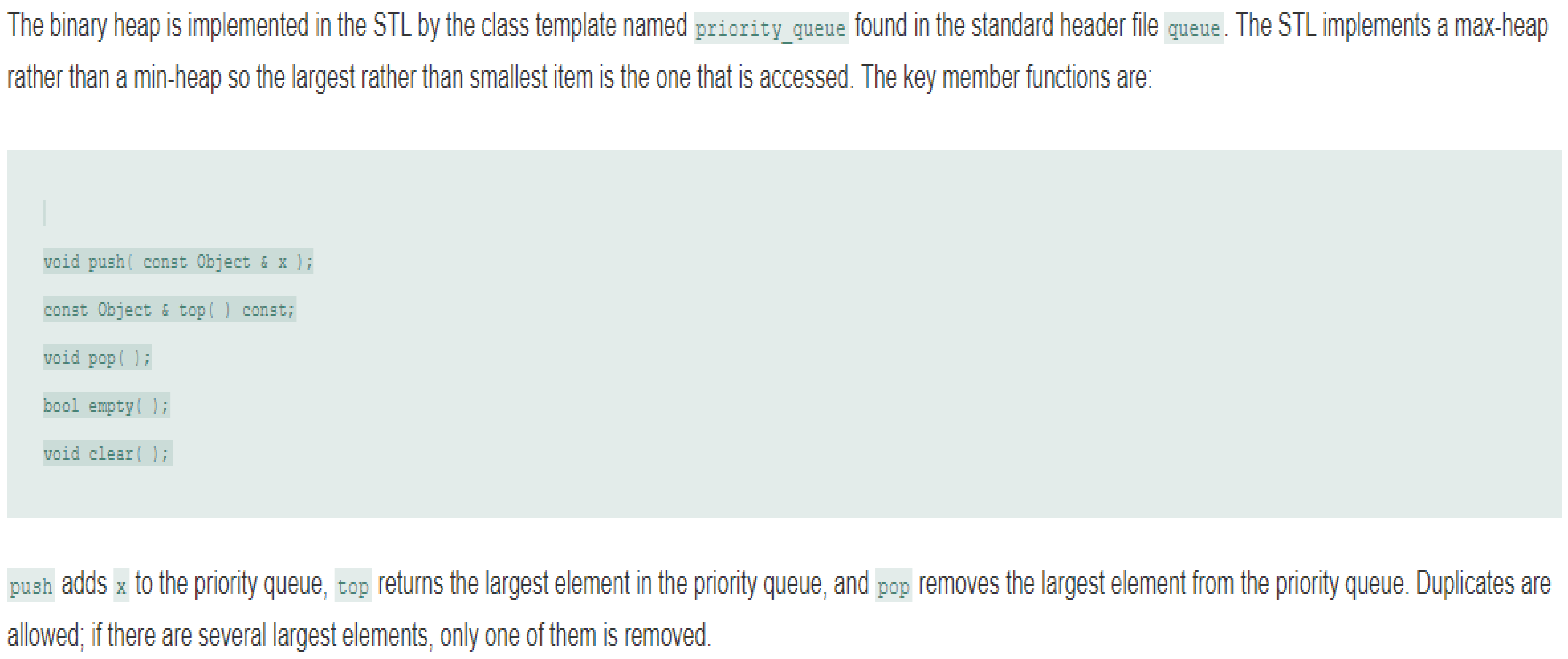
Heaps: Standard Library



Prior to Java 1.5, there was no support in the Java library for priority queues. However, in Java 1.5, there is a generic `PriorityQueue` class. In this class, `insert`, `findMin`, and `deleteMin` are expressed via calls to `add`, `element`, and `remove`. The `PriorityQueue` can be constructed either with no parameters, a comparator, or another compatible collection.

Because there are many efficient implementations of priority queues, it is unfortunate that the library designers did not choose to make `PriorityQueue` an interface. Nonetheless, the `PriorityQueue` implementation in Java 1.5 is sufficient for most priority queue applications.

Heaps: Standard Library



The binary heap is implemented in the STL by the class template named `priority_queue` found in the standard header file `queue`. The STL implements a max-heap rather than a min-heap so the largest rather than smallest item is the one that is accessed. The key member functions are:

```
void push( const Object & x );
const Object & top( ) const;
void pop( );
bool empty( );
void clear( );
```

`push` adds `x` to the priority queue, `top` returns the largest element in the priority queue, and `pop` removes the largest element from the priority queue. Duplicates are allowed; if there are several largest elements, only one of them is removed.

Heaps: Examples

- Priority Queue using doubly linked list: Given Nodes with their priority, implement a priority queue using doubly linked list.
- Operations on Priority Queue :
  - push(): This function is used to insert a new data into the queue.
  - pop(): This function removes the element with the lowest priority value form the queue.
  - peek() / top(): This function is used to get the lowest priority element in the queue without removing it from the queue.

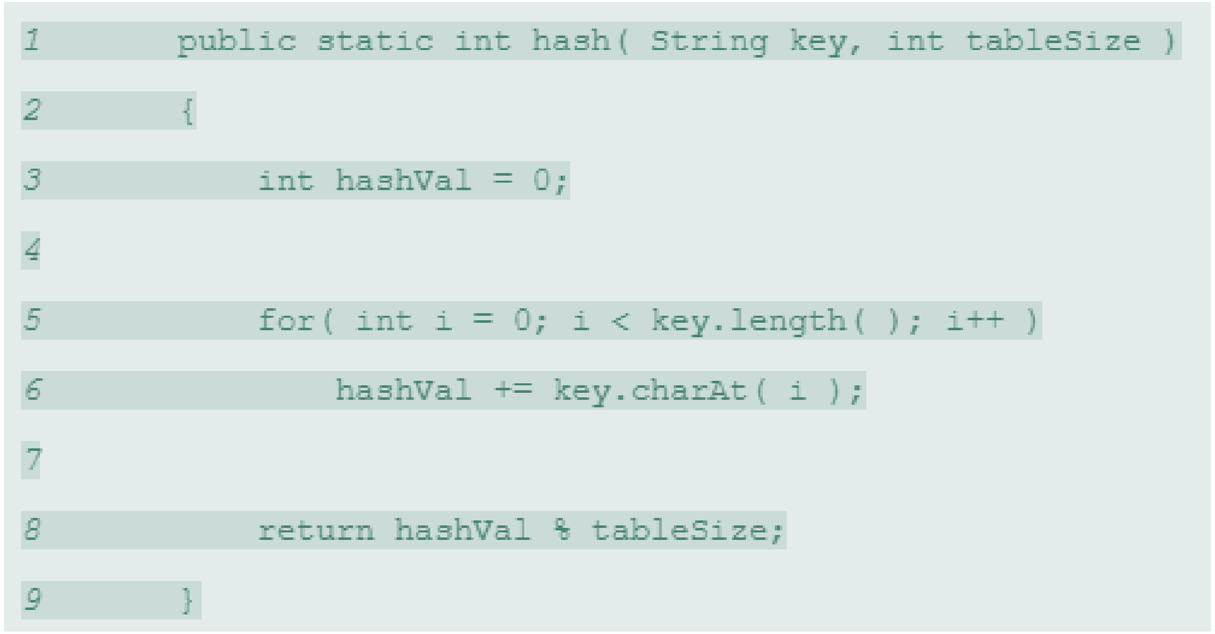
Heaps: Examples

- Print Binary Tree levels in sorted order
  - Given a Binary tree, the task is to print its all level in sorted order

Heaps: Examples

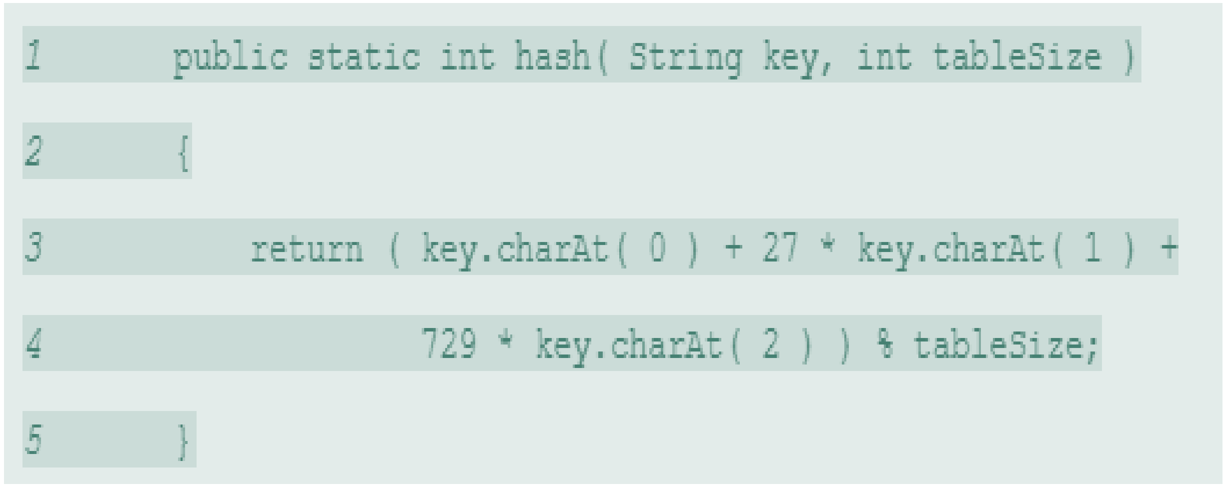
- How to implement stack using priority queue or heap?
  - (using min heap).

Hashing: Hash Function



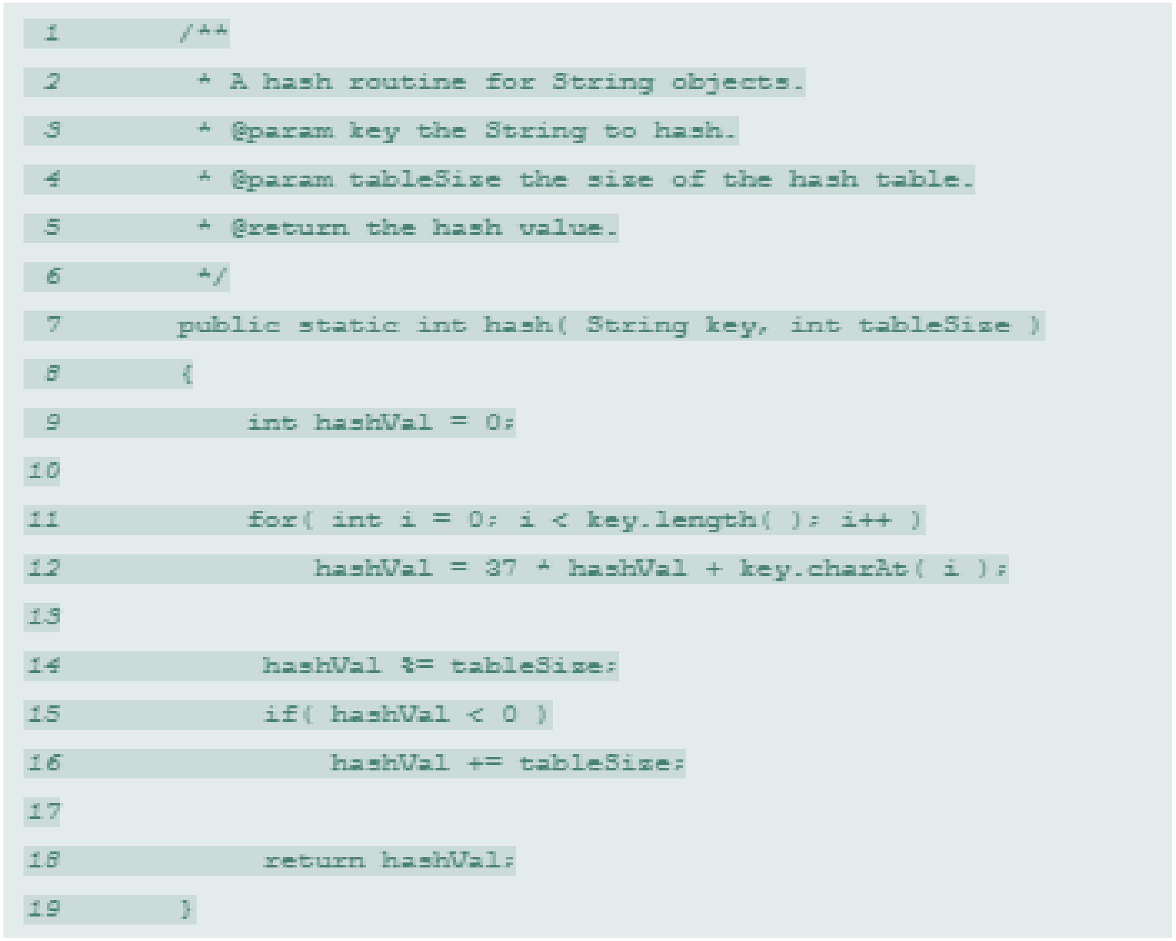
```
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal += key.charAt( i );

    return hashVal % tableSize;
}
```

Hashing: Hash Function



```
public static int hash( String key, int tableSize )
{
    return ( key.charAt( 0 ) + 27 * key.charAt( 1 ) +
                 729 * key.charAt( 2 ) ) % tableSize;
}
```

Hashing: Hash Function



```
/**
 * A hash routine for String objects.
 * @param key the String to hash.
 * @param tableSize the size of the hash table.
 * @return the hash value.
 */
public static int hash( String key, int tableSize )
{
    int hashVal = 0;

    for( int i = 0; i < key.length( ); i++ )
        hashVal = 37 * hashVal + key.charAt( i );

     hashVal %= tableSize;
     if( hashVal < 0 )
         hashVal += tableSize;

     return hashVal;
}
```

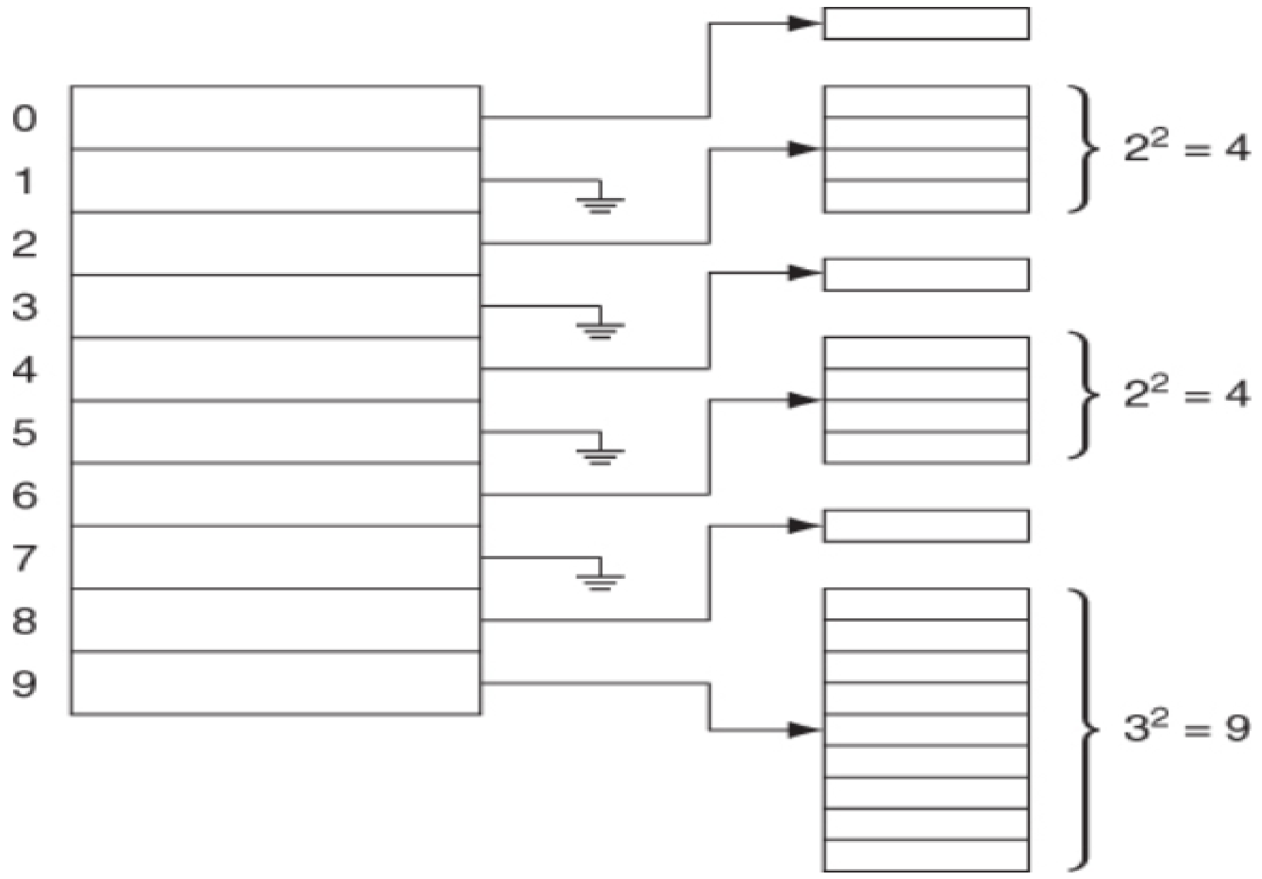
Hashing: Rehashing

- When:
  - load factor is too large. N/TableSize
  - insertion fails to find spot for new entry.
- How:
  - Double size and reinsert N entries as they will map to different spaces.
- Cost:
  - Amortized: only needed when there are O(N) entries and takes O(N) so add constant time only.

Hashing: O(1) worst case

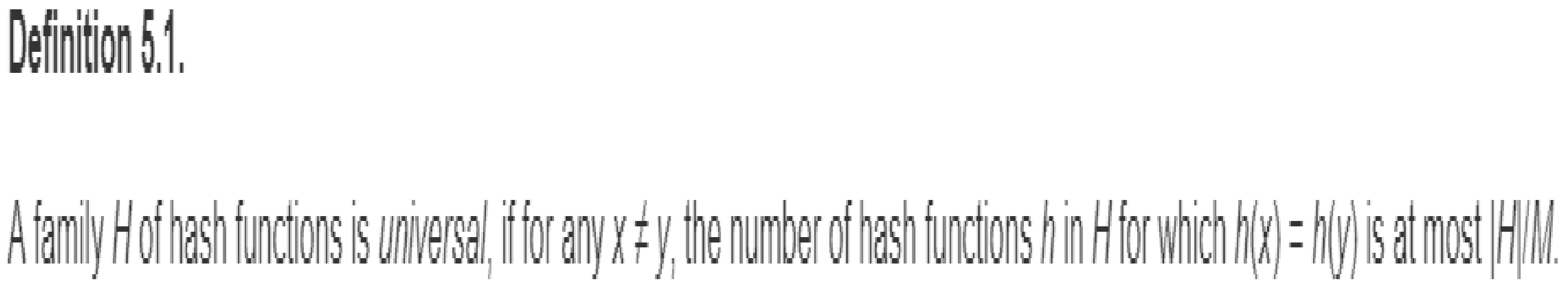
- O(log(N)/log(logN)) for uniform distribution hashes.
- Perfect Hashing
- Universal Hashing
- Cuckoo Hashing

Hashing: Perfect Hashing



Hashing: Universal Hashing

- Randomly select hash function from a set of universal hashes (little collisions)



Definition 5.1.

A family H of hash functions is *universal*, if for any $x \neq y$, the number of hash functions h in H for which $h(x) = h(y)$ is at most $|H|/M$.

Hashing: Cuckoo Hashing

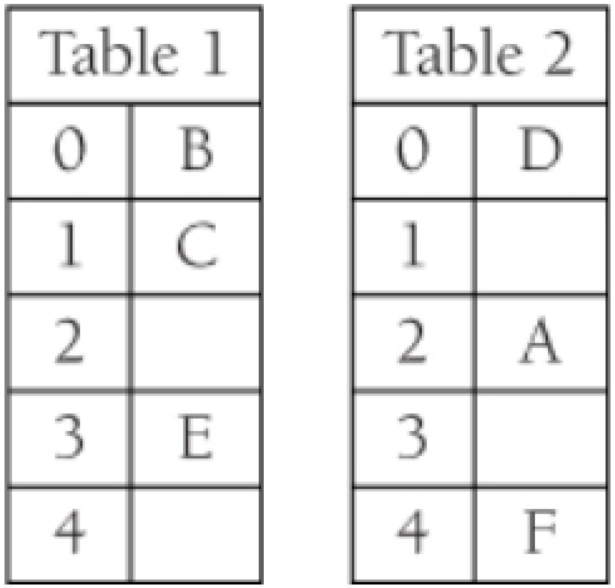


Table 1	
0	B
1	C
2	
3	E
4	

Table 2	
0	D
1	
2	A
3	
4	F

A: 0, 2
B: 0, 0
C: 1, 4
D: 1, 0
E: 3, 2
F: 3, 4

Hashing: Extendible Hashing

- When data is too big to fit in memory, place on disk.

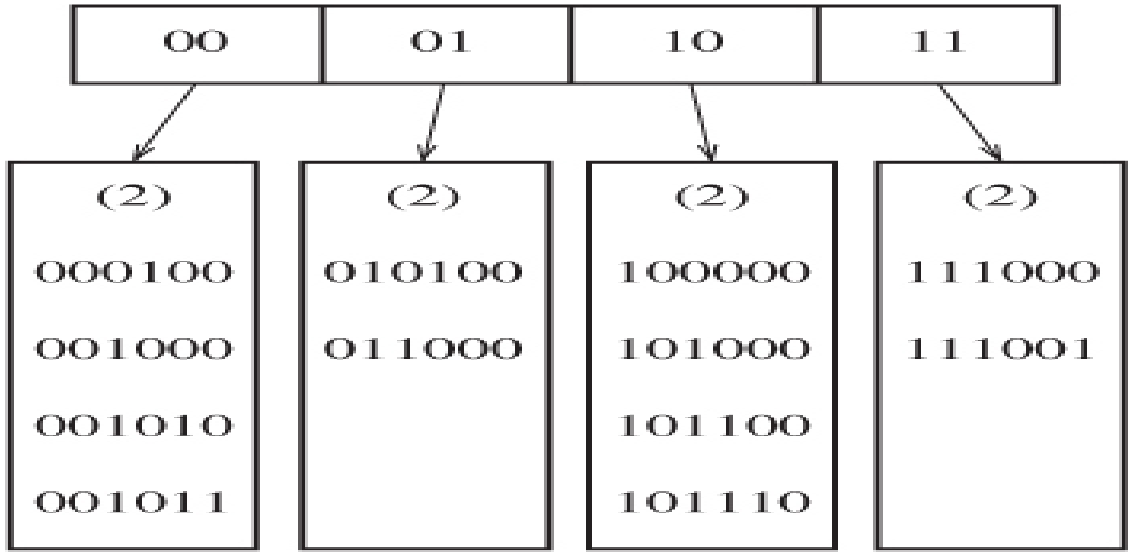


Figure 5.52 Extendible hashing: original data

Hashing: Examples

- Find whether an array is subset of another array
- Union and Intersection of two Linked Lists
- Given an array A[] and a number x, check for pair in A[] with sum as x
- Find four elements a, b, c and d in an array such that a+b = c+d
- Count distinct elements in every window of size k