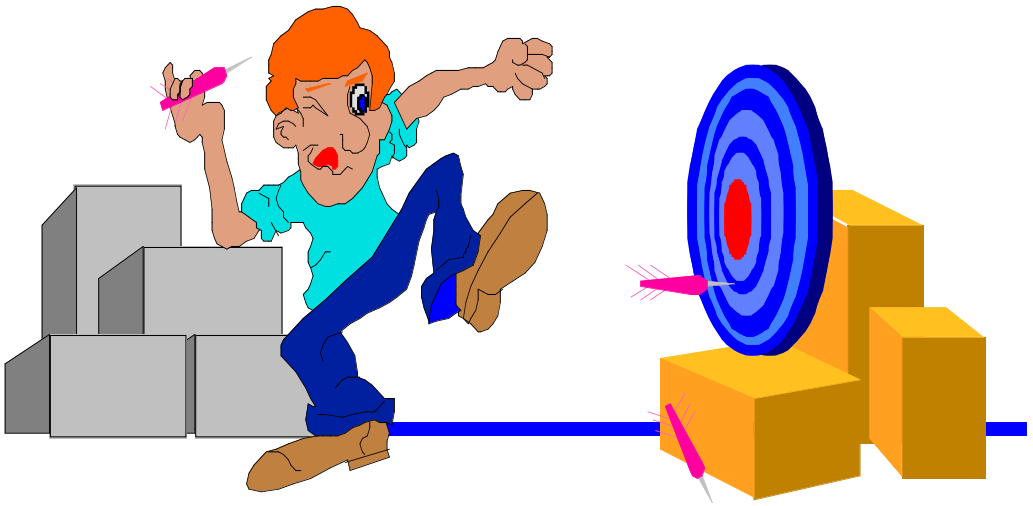
Implementations III

Agenda

- **Hash tables**
  - Implementation methods
  - Applications
  - Hashing vs. binary search trees

- **The binary heap**
  - Properties
  - Logarithmic-time operations
  - Heap construction in linear time
  - Java implementation of `PriorityQueue`
  - Heapsort
  - External sorting

Hashing



Hashing
Search by key transformation

Search in a balanced search tree requires $O(\log N)$ *comparisons* of keys.

Is $O(\log N)$ the best achievable complexity?

No.

How can we decrease the complexity?

With **hashing** - a technique that applies transformations of keys in order to be able directly look them up in a table.

With hashing $O(1)$ complexity can be achieved.

hashing, **hashes**
1. To chop into pieces
2. *Informal* To make a mess of

Basic idea

Store each record in a **table** at an index computed from the key.

A **hash function** is a function for computing a table index from a key.

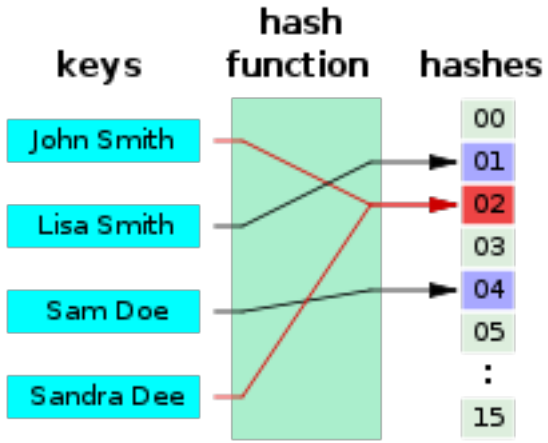
Mathematically: A hash function h is a **map** from the set of possible keys K into an integer interval I:

$$h: K \curvearrowright I$$

Idealistically, two different keys should map to two different indices. However, this is seldom possible to achieve. If two or more keys hash out to the same index, we have a **collision**.

A **collision strategy** is an algorithm for handling collisions.

Hash function example



A hash function that maps names to integers from 0 to 15. There is a **collision** between keys "John Smith" and "Sandra Dee".

Space-time tradeoff

No **space** limits:

use the key as index (trivial hash function)

No **time** limits:

use sequential search

If there are both space and time limits:

use hashing

The hashing technique

Let h denote the hash function.

Insertion:

A record with key k is stored in the table at index $h(k)$, unless there already is a record at that index. In the latter case, the record must be stored in another way (how, depends on the collision strategy).

Search:

When searching for a record with key k we first examine the table at index $h(k)$. If it contains a record with key k, the search is terminated successfully. Otherwise, the search continues (how, depends on the collision strategy).

Good hash functions

- Collision must be avoided as far as possible
  The hash function should map the expected keys as evenly as possible to the index interval.

- The hash function should be computationally cheap.

Design of hash functions

(short keys)

Short keys (keys that fit a machine word):

Treat key k as an integer and compute

$h(k) = k \bmod M$ (in Java: `k % M`)

where M is the size of the table

$h(k) \curvearrowright [0;M\text{-}1]$

Example with short keys

Four-character keys, table size 101.

```
ASCII        a        b        c        d
hex      6| 1| 6| 2| 6| 3| 6| 4|
bin   01100001011000100110001101100100
```

```
0x61626364 = 1633831724
1633831724 % 101 = 11
```

Key "abcd" hashes to 11.

```
0x64636261 = 1684234849
1684234849 % 101 = 57
```

Key "dcba" hashes to 57.

```
0x61626263 = 1633837667
1633837667 % 101 = 57
```

Key "abbc" also hashes to 57. Collision!

Design of hash functions

(long keys)

Long keys (keys that do not fit a machine word):

Treat key k as integer and compute

$$h(k) = k \bmod M$$

where M is the size of the table

That is, in principle, as for short keys.

Example with long keys

Example with four-character keys. But the method works for any length.

Use Horner's rule:

```
0x61626364 =
97*256^3 + 98*256^2 + 99*256^1 + 100 =
((97*256 + 98)*256 + 99)*256 + 100
```

Take modulo after each addition to avoid arithmetic overflow:

```
(97*256 + 98  = 24930) % 101 = 84
(84*256 + 99  = 21603) % 101 = 90
(90*256 + 100 = 23140) % 101 = 11
```

Table size

Choose table size to be a prime.

Why?

In the previous example we had

```
"abcd" = 0x61626364 =
97*256^3 + 98*256^2 + 99*256^1 + 100
```

If the table size is chosen to be 256, only the last character (d) will contribute to the result.

A simple method for assuring that all characters contribute is to choose the table size to be a prime.

figure 20.1

A first attempt at a hash function implementation

```
1     // Acceptable hash function
2     public static int hash( String key, int tableSize )
3     {
4         int hashVal = 0;
5
6         for( int i = 0; i < key.length( ); i++ )
7             hashVal = ( hashVal * 128 + key.charAt( i ) )
8                                                 % tableSize;
9         return hashVal;
10    }
```

ASCII characters can typically be represented in 7 bits as a number between 0 and 128. Unfortunately, the repeated multiplication will shift early characters to the left - out of the answer. Furthermore, the modulus computation (%) after each addition is expensive.

figure 20.2

A faster hash function that takes advantage of overflow

```
1       /**
2        * A hash routine for String objects.
3        * @param key the String to hash.
4        * @param tableSize the size of the hash table.
5        * @return the hash value.
6        */
7       public static int hash( String key, int tableSize )
8       {
9           int hashVal = 0;
10
11          for( int i = 0; i < key.length( ); i++ )
12              hashVal = 37 * hashVal + key.charAt( i );
13
14          hashVal %= tableSize;
15          if( hashVal < 0 )
16              hashVal += tableSize;
17
18          return hashVal;
19      }
```

```
1       // A poor hash function when tableSize is large
2       public static int hash( String key, int tableSize )
3       {
4           int hashVal = 0;
5
6           for( int i = 0; i < key.length( ); i++ )
7              hashVal += key.charAt( i );
8
9           return hashVal % tableSize;
10      }
```

figure 20.3

A bad hash function if `tableSize` is large

`hashCode` in `java.lang.String`

```
final class String {
    public int hashCode() {
        if (hash != 0)
            return hash;
        for (int i = 0; i < length(); i++)
            hash = hash * 31 + (int) charAt(i);
        return hash;
    }

    private int hash = 0;
}
```

1. The constant 37 has been replaced by 31.
2. A computed hash value is remembered (cached).

Frequency of collisions

The birthday paradox:

How many persons should be invited to a party so that there is at least 50% chance that at least two people have the same birthday?

Answer: 23.

Let M be the table size. How many insertions until the first collision?

M	$\sqrt{\pi M / 2}$
100	12
365	23
1000	40
10000	125
100000	396
1000000	2353

Collision strategies

Number of keys: N
Table size: M

Option 1 (**open addressing**):

Keep $N < M$:

Put keys somewhere in the table.

Option 2 (**separate chaining**):

Allow $N > M$:

Put keys that hash to the same index in a list (about N/M keys per list).

Open addressing
Linear probing

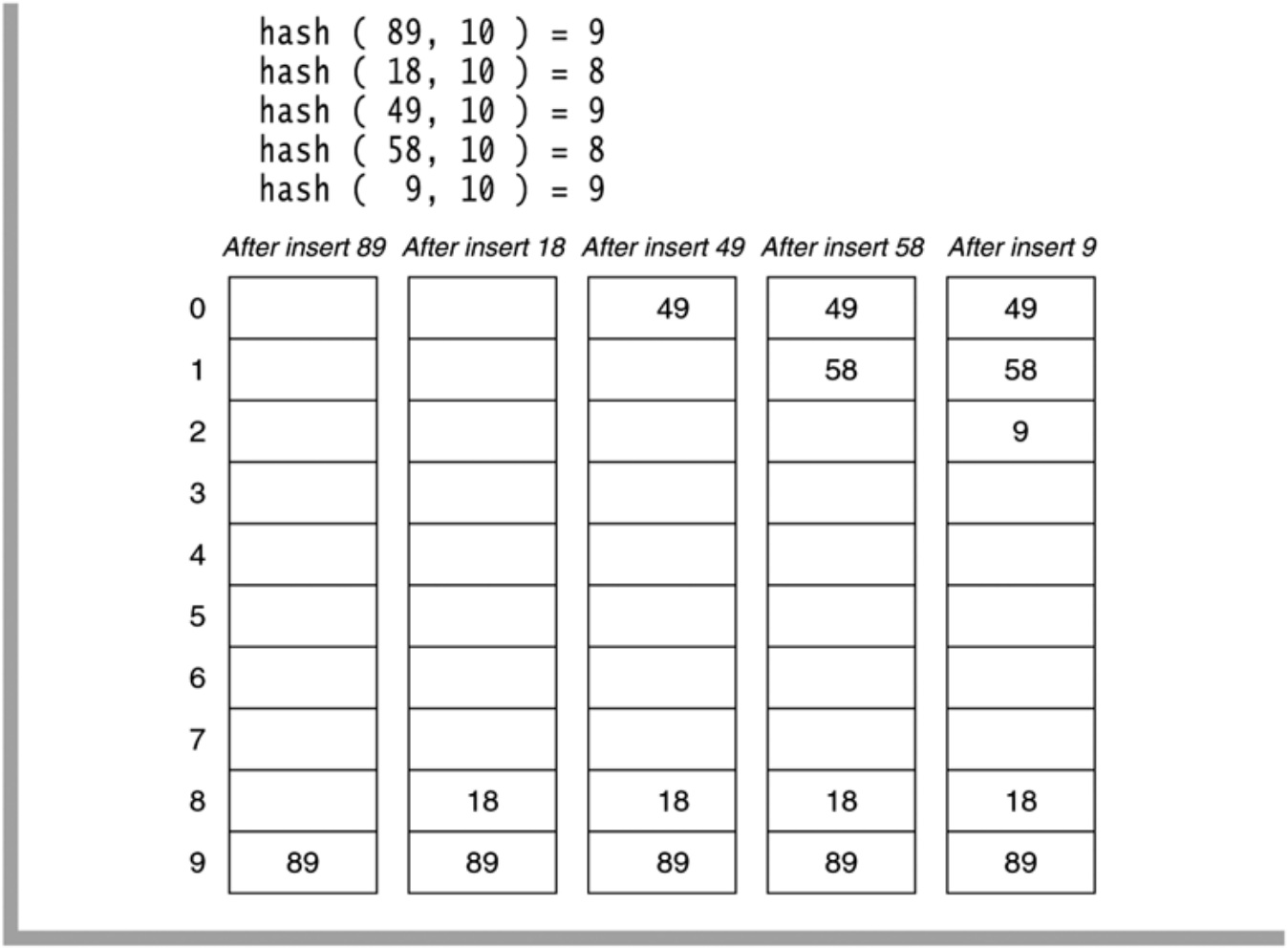
Open addressing:
No links. Everything is kept in the table.

Linear probing:
Start linear search at hash position (stop when an empty position is hit).

Constant time if table is sparse.

figure 20.4

Linear probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).



A hash table interface

```
interface HashTable<K,V> {
    void put(K key, V value);
    V get(K key);
    void remove(K key);
}
```

Implementation of open addressing

```
abstract class ProbingHashTable<K,V> implements HashTable<K,V> {
    ProbingHashTable()
      { array = new HashEntry[DEFAULT_TABLE_SIZE]; }

    void put(K key, V value) { ... }
    V get(K key) { ... }
    void remove(K key) { ... }

    protected abstract int findPos(K key);

    protected HashEntry[] array;
    private int currentSize;
    private static final int DEFAULT_TABLE_SIZE = 101;
}
```

```
class HashEntry {
    HashEntry(Object k, Object v)
      { key = k; value = v; }
    Object key, value;
    boolean isActive = true;
}
```

```
V get(K key) {
    int pos = findPos(key);
    if (array[pos] == null || !array[pos].isActive)
        return null;
    return (V) array[pos].value;
}
```

```
void remove(K key) {
    int pos = findPos(key);
    if (array[pos] != null)
        array[pos].isActive = false;
}
```

```
void put(K key, V value) {
    int pos = findPos(key);
    array[pos] = new HashEntry(key, value);
    if (++currentSize < array.length / 2)
        return;
    // rehash
    HashEntry[] oldArray = array;
    array = new HashEntry[nextPrime(2 * oldArray.length)];
    currentSize = 0;
    for (int i = 0; i < oldArray.length; i++)
        if (oldArray[i] != null && oldArray[i].IsActive)
            put((K) oldArray[i].key, (V) oldArray[i].value);
}
```

Running time for `nextPrime` is $O(\sqrt{n}\log n)$.

Simple copying when `rehash`ing does not work.

```
int nextPrime(int n) {
    if (n % 2 == 0)
        n++;
    while (!isPrime(n))
        n += 2;
    return n;
}
```

Class `LinearProbingHashTable`

```
class LinearProbingHashTable<K,V> extends ProbingHashTable<K,V> {
    protected int findPos(K key) {
        int pos = Math.abs(key.hashCode()) % array.length;
        while (array[pos] != null && !array[pos].key.equals(key))
            if (++pos >= array.length)
                pos = 0;
        return pos;
    }
}
```

Efficiency of linear probing

Linear probing uses less than 5 probes for searching a hash table that is less than 2/3 full.

The precise expressions are

$$\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$$

probes on average for an *unsuccessful* search, and

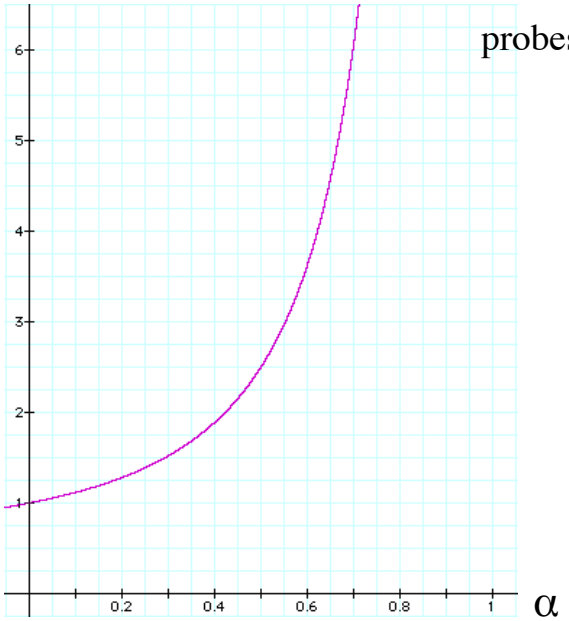
$$\frac{1}{2} + \frac{1}{2(1-\alpha)}$$

probes on average for a *successful* search, where $\alpha = N/M$ denotes the **load factor**.

Efficiency curves for linear probing

$$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$$

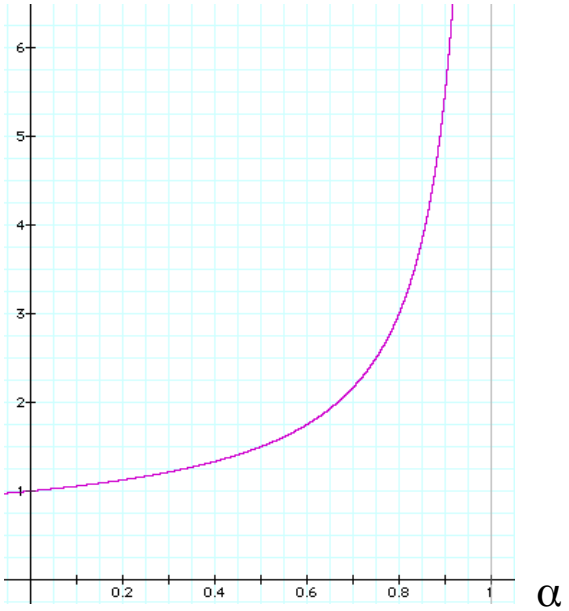
probes



Unsuccessful search

$$\frac{1}{2}+\frac{1}{2(1-\alpha)}$$

probes



Successful search

Clustering

Bad phenomenon: records clumps together.

Long clusters tend to get longer.

Average search cost grows to M as the table is filled.

Linear probing is too slow when the table is 70-80% full.

Argument for clustering tendency

Suppose all positions [i:j] store records, whereas i-1, j+1, and j+2 are empty.

Then the chance that a new record is stored at position j+1 is equal to the chance that its key hashes to one of the values in [i:j+1].

For the new record to be stored at position j+2 its key must hash to exactly j+2.

Quadratic probing

(reduces the risk of clustering)

Probing sequence:

Linear probing: $pos, pos+1, pos+2, pos+3, ...$

Quadratic probing: $pos, pos+1^2, pos+2^2, pos+3^2, ...$

Squaring of the step number can be avoided:

Let H_{i-1} be the most recently computed probe (H_0 is the original hash position) and H_i be the probe we are trying to compute. Then we have

$$H_{i-1} = pos + (i - 1)^2 = pos + i^2 - 2i + 1 = H_i - 2i + 1$$

and obtain $H_i = H_{i-1} + 2i - 1$.

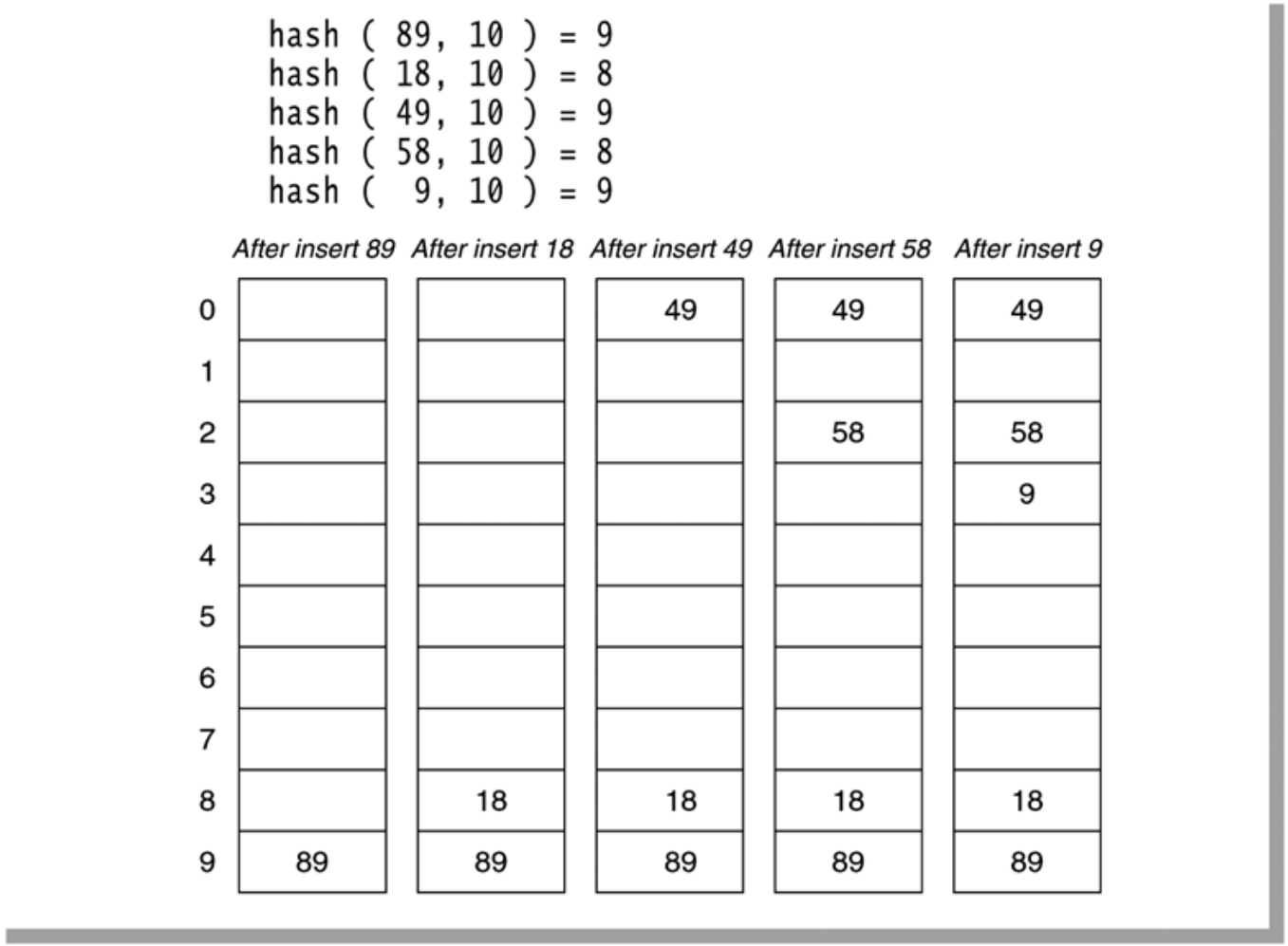


figure 20.6

A quadratic probing hash table after each insertion (note that the table size was poorly chosen because it is not a prime number).

Implementation of quadratic probing

```
class QuadraticProbingTable<K,V> extends ProbingHashTable<K,V> {
    protected int findPos(K key) {
        int pos = Math.abs(key.hashCode()) % array.length;
        int i = 0;
        while (array[pos] != null &&
               !array[pos].element.equals(key)) {
            if ((pos = pos + 2 * ++i - 1) >= array.length)
                pos = 0;
        return pos;
    }
}
```

It has been proven that

> If the table size is prime and the table is at least half empty, then a new element can always be inserted, and no cell is probed twice.

Double hashing

Reduces the risk of clustering by using a second hash function.

The strategy is the same as for linear probing; the only difference is that, in stead of examining each successive table position following a collision, we use a second hash function to get a fixed increment to use for the probe sequence.

By this means the chance of finding empty cells during insertion is increased.

Implementation of double hashing

```
class DoubleHashTable<K,V> extends ProbingHashTable<K,V> {
    protected int findPos(K key) {
        int pos = Math.abs(key.hashCode()) % array.length;
        int k = Math.abs(key.h2());
        while (array[pos] != null &&
               !array[pos].element.equals(key))
            pos = (pos + k) % array.length;
        return pos;
    }
}
```

Requirements for the second hash function

- It must never return 0.
- It must always return values that are relatively prime to M.

  This can be achieved by choosing M as prime and letting $h_2(k) < M$ for every k.
- It must differ from the first hash function.

A simple and fast method is

$h_2(k) = 8 - k \% 8$ ($k \% 8$ is equal to the last 3 bits of k)

Efficiency of double hashing

Double hashing uses fewer probes on average than linear probing. Less than 5 probes in a table that is 80% full, and less than 5 probes for a successful search in a table that is 99% full.

The precise expressions are

$$\frac{1}{1-\alpha}$$

probes on average for an *unsuccessful* search, and

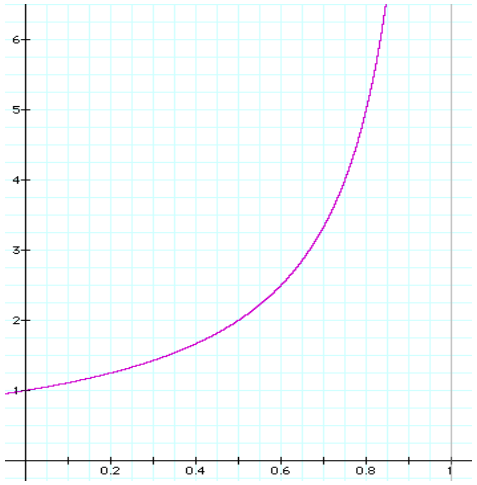
$$\frac{-\ln(1-\alpha)}{\alpha}$$

probes on average for a *successful* search, where $\alpha = N/M$ denotes the **load factor**.

Double hashing versus linear probing

Double hashing

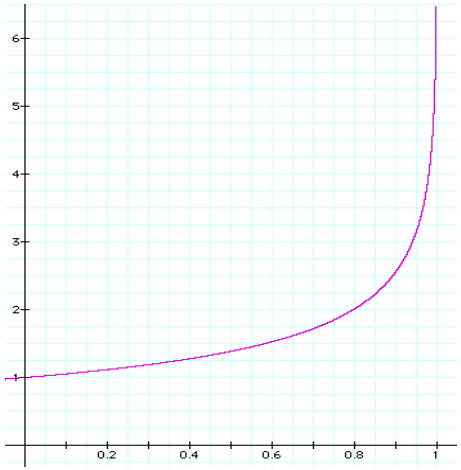
$\frac{1}{1-\alpha}$



α

Unsuccessful search

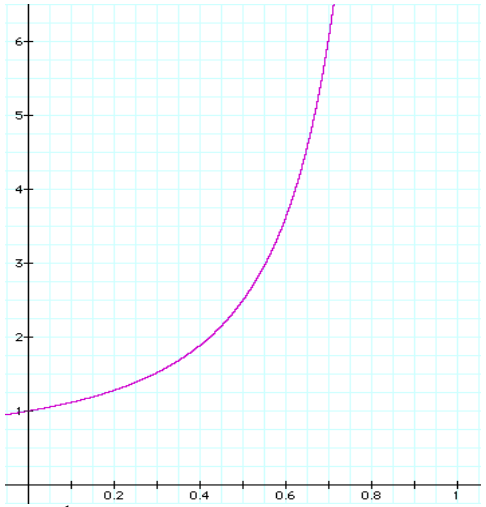
$-\frac{\ln(1-\alpha)}{\alpha}$



α

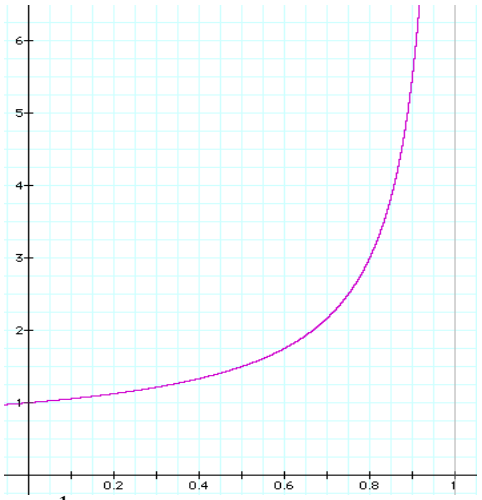
Successful search

Linear probing



α

$\frac{1}{2}+\frac{1}{2(1-\alpha)^2}$



α

$\frac{1}{2}+\frac{1}{2(1-\alpha)}$

Separate chaining hashing

Simple, practical and widely used.
Method: M linked lists, one for each table slot.

0:	*
1:	L A W *
2:	M X *
3:	N C *
4:	*
5:	E P *
6:	*
7:	G R *
8:	H S *
9:	I *
10:	*

($M = 11$)
($N = 14$)

Cuts search time by a factor of M over sequential search.

Implementation of separate chaining hashing

```
interface HashTable<K,V> {
    void put(K key, V value);
    V get(K key);
    void remove(K key);
}
```

```
public class SeparateChainingHashTable<K,V> implements
             HashTable<K,V> {
    private HashEntry[] array;
    ...
}
```

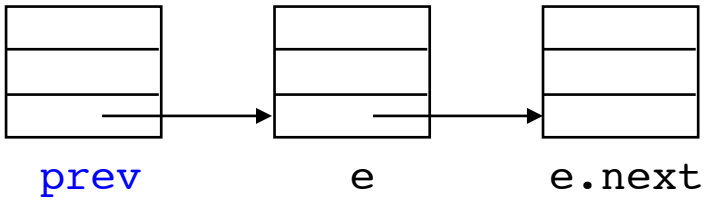
```
class HashEntry {
    HashEntry(Object k, Object v, HashEntry n)
      { key = k; value = v; next = n; }

    Object key, value;
    HashEntry next;
}
```

```
 V get(K key) {
     int pos = Math.abs(key.hashCode()) % array.length;
     for (HashEntry e = array[pos]; e != null; e = e.next)
         if (key.equals(e.key))
             return e.value;
     return null;
 }
```

```
void put(K key, V value) {
    int pos = Math.abs(key.hashCode()) % array.length;
    for (HashEntry e = array[pos]; e != null; e = e.next)
        if (key.equals(e.key)) {
            e.value = value;
            return;
        }
    array[pos] = new HashEntry(key, value, array[pos]);
}
```

```
void remove(K key) {
    int pos = Math.abs(key.hashCode()) % array.length;
    HashEntry prev = null;
    for (HashEntry e = array[pos]; e != null; prev = e, e = e.next)
        if (key.equals(e.key)) {
            if (prev != null)
                prev.next = e.next;
            else
                array[pos] = e.next;
            return;
        }
    }
}
```



Efficiency of separate chaining hashing

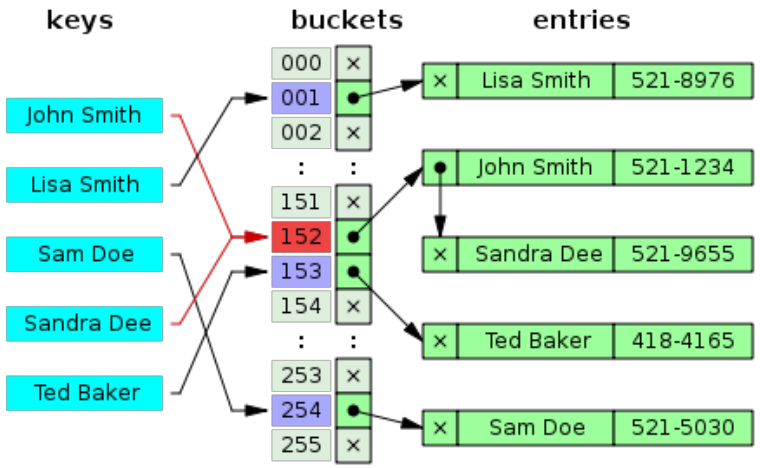
Average search cost (successful):	$N/M/2$
Average search cost (unsuccessful):	N/M
Insertion cost:	N/M
Worst case ("probabilistically" unlikely):	N

If the lists are kept sorted:

Cuts average unsuccessful search time to $N/M/2$.
Cuts average insertion time to $N/M/2$.

Advantages of separate chaining hashing

- Idiot proof (doesn't break down)
- Deletion is simple



Reasons not to use hashing

Hashing allows search and insertion to run in constant time.

Why use other methods?

- There is no performance guarantee
- Too much arithmetic if keys are long
- Takes extra space
- Does not support sorting

Hash tables versus binary search trees

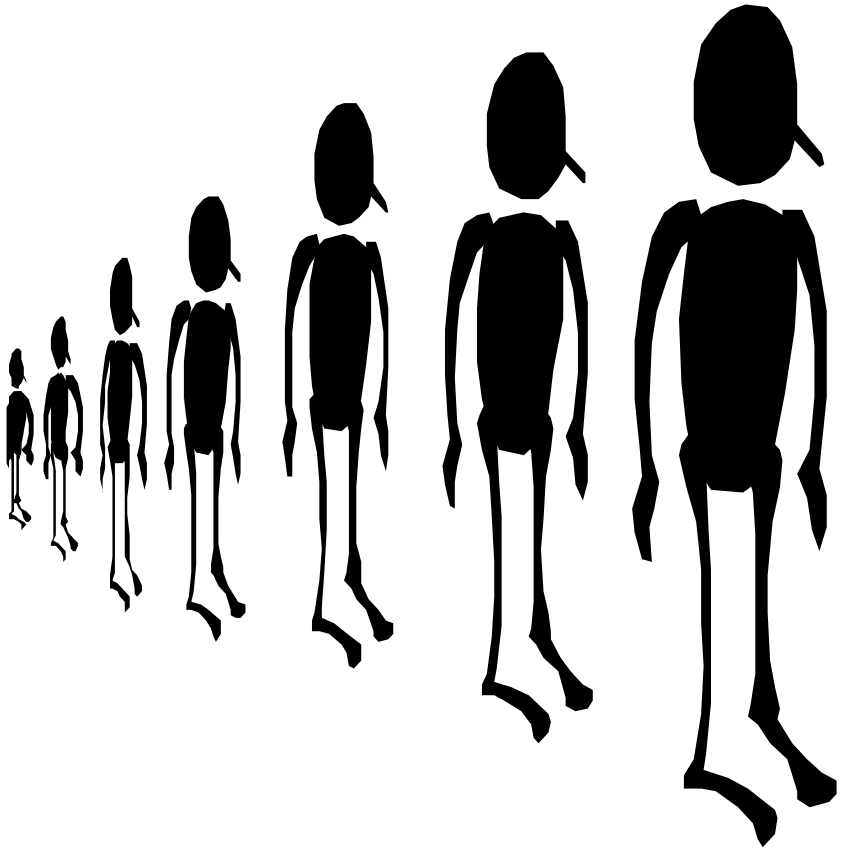
Experimental results

Insertion of 10,000,000 different integers into an initially empty set, followed by deletion of each element, in random order.
2.8 GHz MacBook Pro.

Red-black tree (`java.util.TreeSet`)	57.9 seconds
AA-tree (`weiss.util.TreeSet`)	65.0 seconds
Hash set (`java.util.HashSet`)	45.2 seconds

```
java -Xmx1G
```

Priority queues



Priority queues

A **priority queue** is an abstract data type that supports the following two operations:

insert(x):	add the element x to the priority queue with an associated *priority*
deleteMin:	remove the element with the lowest priority, and return it.

Ordinary queues and stacks are special cases of priority queues.

Applications of priority queues

- operating systems
- graph search
- file compression
- discrete event simulation
- sorting

Specification in Java

```
interface PriorityQueue {
    void insert(Comparable x);
    Comparable deleteMin();
}
```

Sometimes it is appropriate to add futher operations, e.g.:

`boolean isEmpty()`:
    return true if the priority queue contains no elements

`Comparable getMin()`:
    return the element with the lowest priority

`void merge(PriorityQueue pq)`:
    merge this priority queue with another one (`pq`).

Implementation using an *unordered* array

```
class ArrayPriorityQueue implements PriorityQueue {
    private Comparable[] array;
    private int currentSize;

    ArrayPriorityQueue()
        { array = new Comparable[DEFAULT_CAPACITY]; }

    public void insert(Comparable x)
      { checkSize(); array[currentSize++] = x; }

    public Comparable deleteMin() {
        if (currentSize == 0)
            throw new UnderflowException();
        int min = 0;
        for (int i = 1; i < currentSize; i++)
            if (array[i].compareTo(array[min]) < 0)
                min = i;
        swapReferences(array, min, currentSize - 1);
        return array[--currentSize];
    }
}
```

$O(1)$

$O(N)$

Implementation using an *ordered* array

The array is kept sorted in *decreasing order*

```
void insert(Comparable x) {
    checkSize();
    int i = currentSize;
    while (i > 0 && array[i - 1].compareTo(x) < 0)
        { a[i] = a[i - 1]; i--; }
    array[i] = x; currentSize++;
}

Comparable deleteMin() {
    if (currentSize == 0)
        throw new UnderflowException();
    return array[--currentSize];
}
```

$O(N)$

$O(1)$

Other implementations: unordered lists, ordered lists.

Sorting using a priority queue

Sorting an array `a` in increasing order:

```
PriorityQueue pq = new TypePriorityQueue();
for (int i = 0; i < a.length; i++)
    pq.insert(a[i]);
for (int i = 0; i < a.length; i++)
    a[i] = pq.deleteMin();
```

If the priority queue is implemented using an *unordered* array, the algorithm corresponds to *selection sort*.

If the priority queue is implemented using an *ordered* array, the algorithm corresponds to *insertion sort*.

Implementation using a search tree

```
void insert(Comparable x) {
    searchTree.insert(x);
}

Comparable deleteMin() {
    return searchTree.removeMin();
}
```

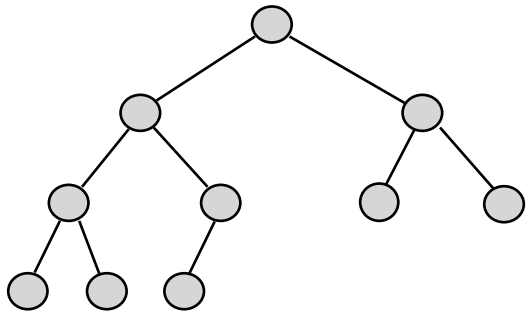
If the search tree is kept balanced, running time for both operations is $O(\log N)$.

However, implementation is difficult (particularly `removeMin`).

Binary heap

A binary **heap** is a **complete** binary tree (*structure property*) in which the key in every node is less than or equal to the keys of its children (*heap-order property*).

Complete tree: All levels are filled, with the possible exception of the bottom level, which is filled from left to right.



We can conclude that the smallest key is in the root.

figure 21.2

Heap-order property

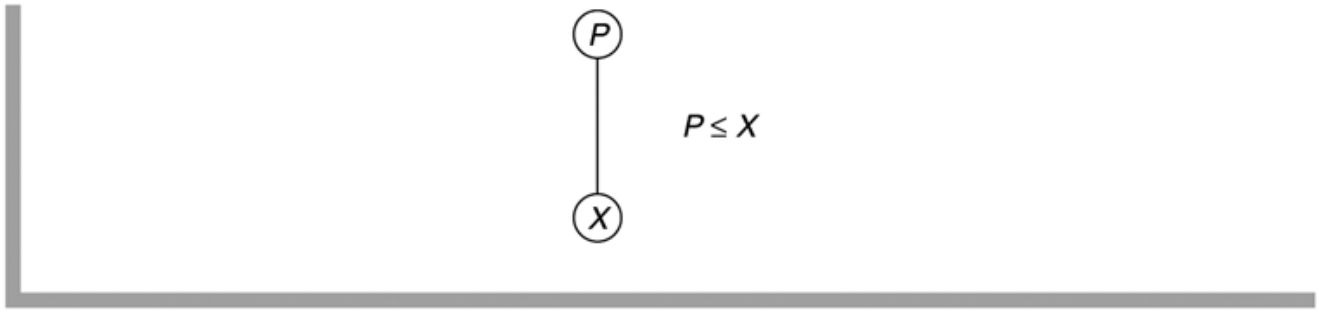
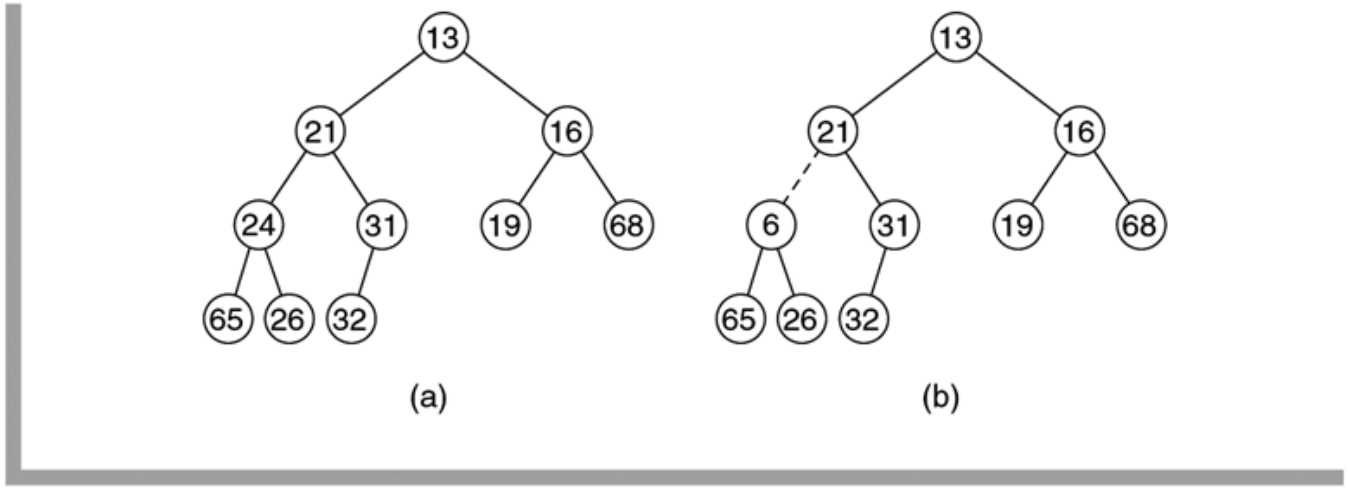
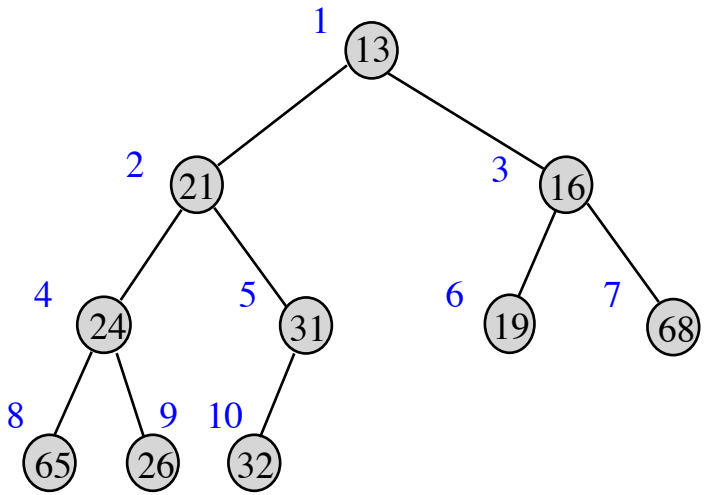


figure 21.3

Two complete trees: (a) a heap; (b) not a heap



Heap representation



A heap can be represented in an array (no explicit links needed):

root:	`array[1]`
children of root:	`array[2]` and `array[3]`
children of `i`:	`array[2*i]` and `array[2*i+1]`
parent of `i`:	`array[i/2]`

```
i:      0  1  2  3  4  5  6  7  8  9 10
array:    13 21 16 24 31 19 68 65 26 32
```

(level order, implicit representation)

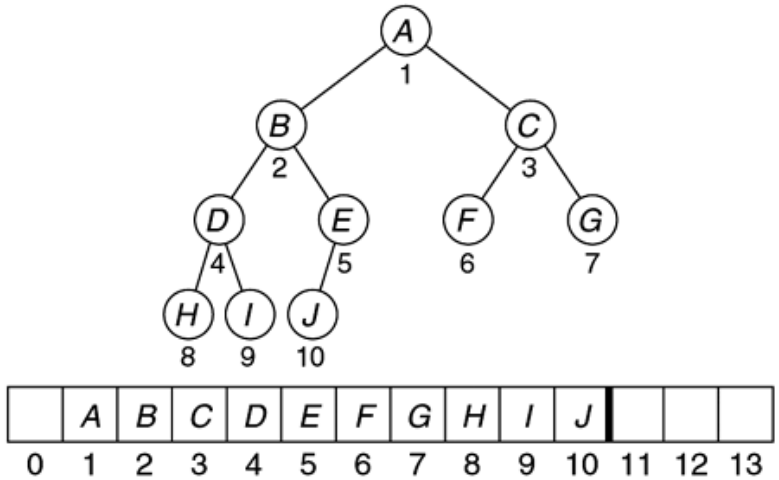


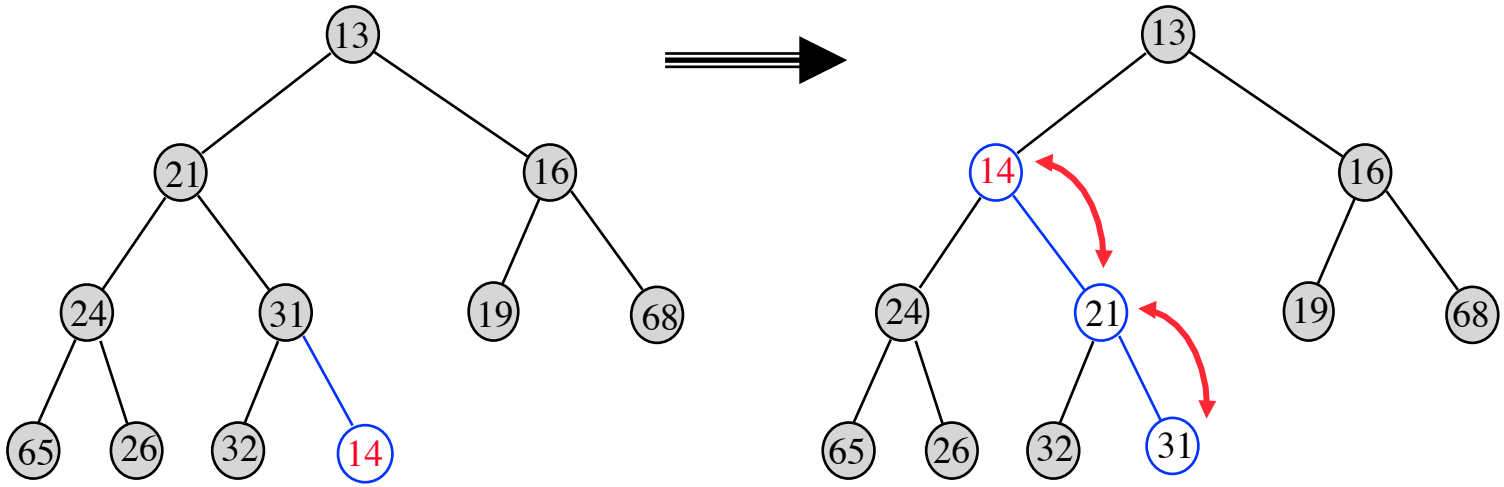
figure 21.1

A complete binary tree and its array representation

Insertion

Insert the element as the last one in the heap. This does not violate the structure property.

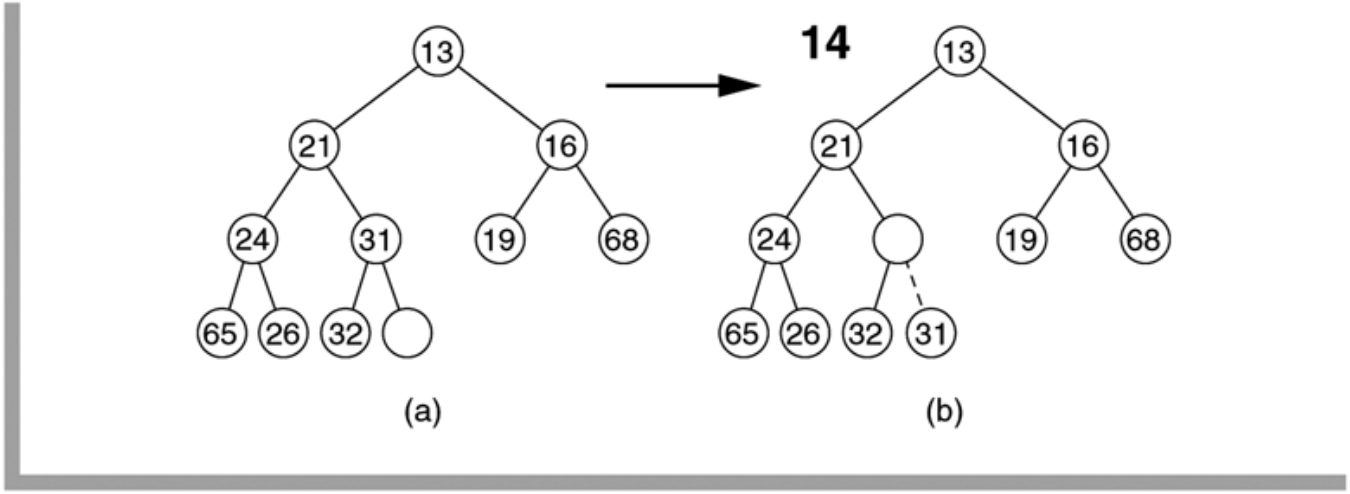
Maintain the heap order property by exchanging the new node with its parent as long as the heap-order property is violated.



Insertion of 14

figure 21.7

Attempt to insert 14, creating the hole and bubbling the hole up



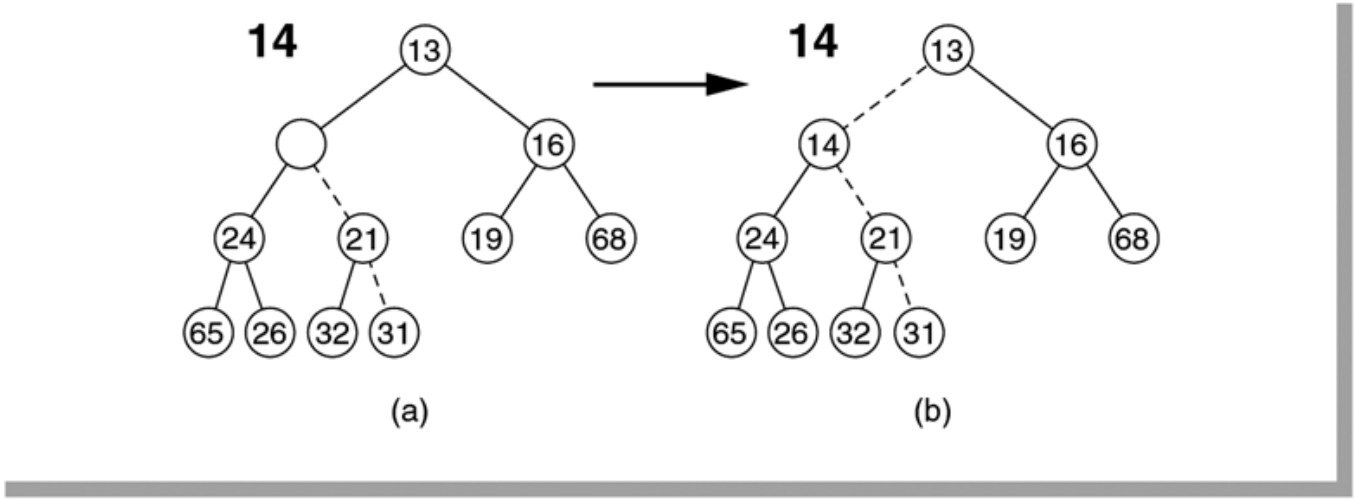


figure 21.8

The remaining two steps required to insert 14 in the original heap shown in Figure 21.7

Implementation of `insert`

```
void insert(Comparable x) {
   checkSize();
   array[++currentSize] = x;
   percolateUp(currentSize);
}
```

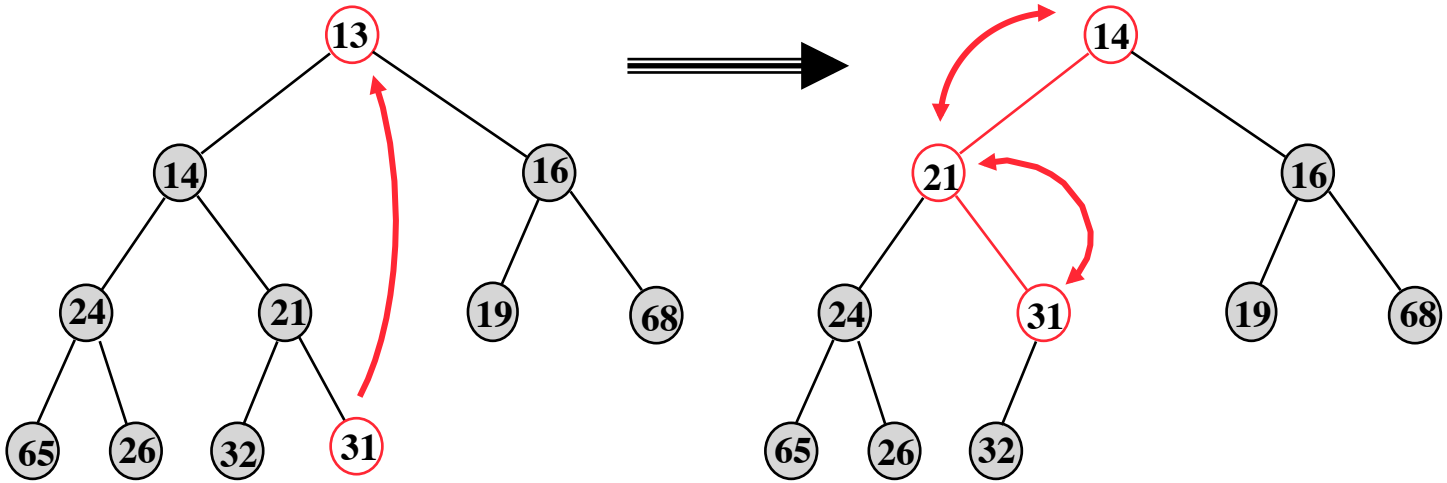
```
void percolateUp(int hole) {
    Comparable x = array[hole];
    array[0] = x;
    for ( ; x.compareTo(array[hole / 2]) < 0; hole /= 2)
       array[hole] = array[hole / 2];
    array[hole] = x;
}
```

Time complexity: $O(\log N)$

Deletion of the root element

Replace the root by the last element of the heap.

Maintain the heap order property by exchanging this node by the smallest of its children as long as the heap-order property is violated.



Deletion of 13

figure 21.10

Creation of the hole at the root

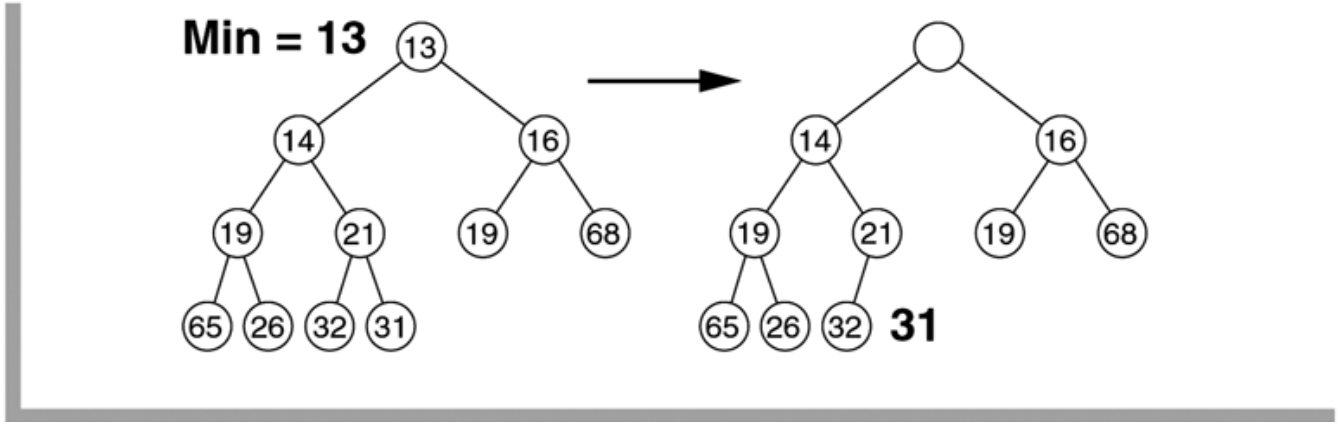
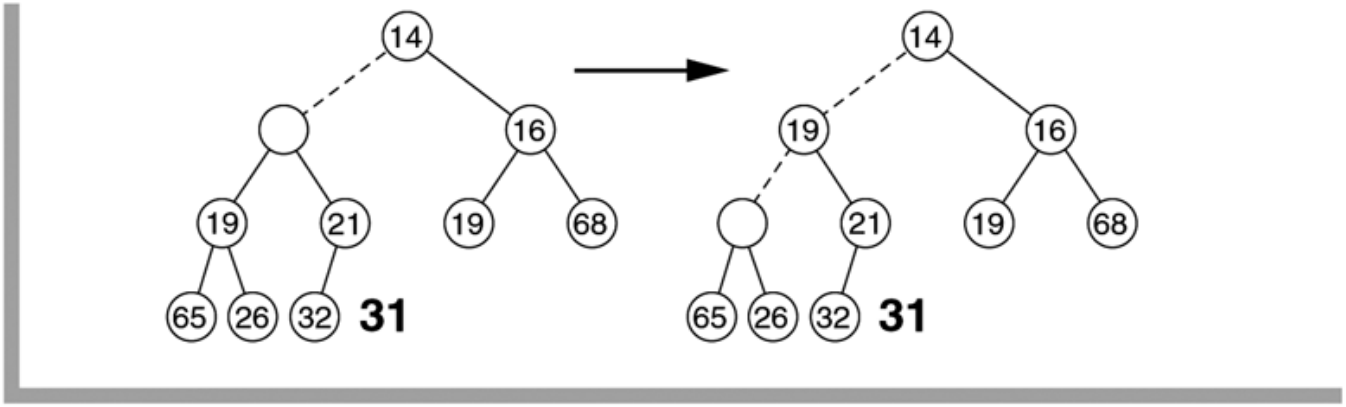


figure 21.11

The next two steps in the `deleteMin` operation



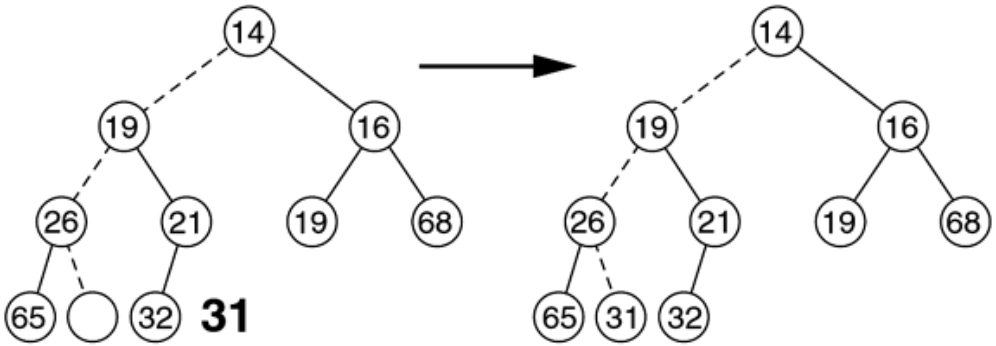


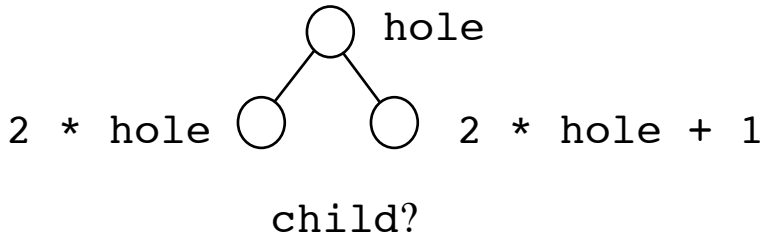
figure 21.12

The last two steps in the `deleteMin` operation

Implementation of `deleteMin`

```
Comparable deleteMin() {
    if (isEmpty())
        throw new UnderflowException();
    Comparable minItem = array[1];
    array[1] = array[currentSize--];
    percolateDown(1);
    return minItem;
}
```

```
void percolateDown(int hole) {
    int child;
    Comparable tmp = array[hole];
    for ( ; hole * 2 <= currentSize; hole = child) {
        child = 2 * hole;
        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;
}
```

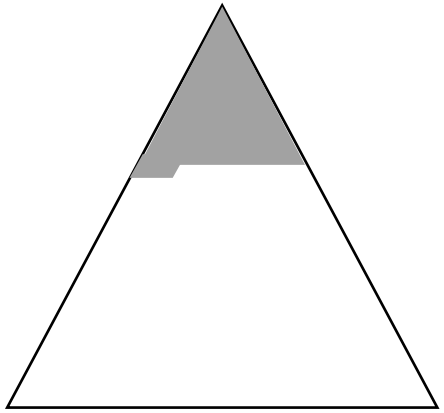


Time complexity: $O(\log N)$

Heap construction

Problem: Given an array `array`[1:*N*] of elements in arbitrary order, rearrange the elements so that the array is a heap.

Induction hypothesis (**top-down**): `array`[1:*i*] is a heap.



Top-down heap construction

```
for (int i = 2; i <= N; i++)
    percolateUp(i);
```

Time complexity:

$$O(\sum_{i=2..N} \log i) = O(N \log N)$$

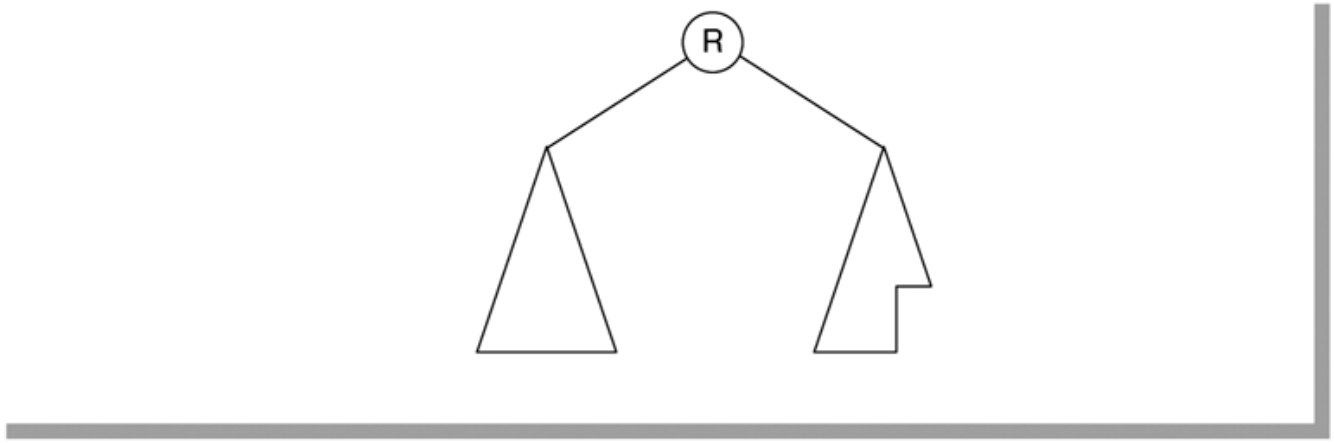
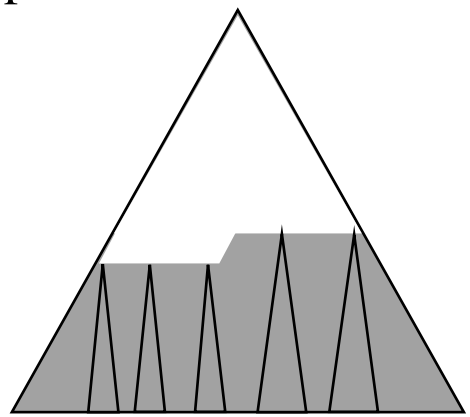


figure 21.15

Recursive view of the heap

Induction hypothesis (**bottom-up**): All the trees represented by `array`[*i*:*N*] are heaps.



`array`[*N*/2+1:*N*] represent heaps (they are leaves in the final heap).

```
for (int i = N / 2; i >= 1; i--)
    percolateDown(i);
```

Time complexity:

$$O(\sum_{i=1..N/2} \log(N/i)) = O(N)$$

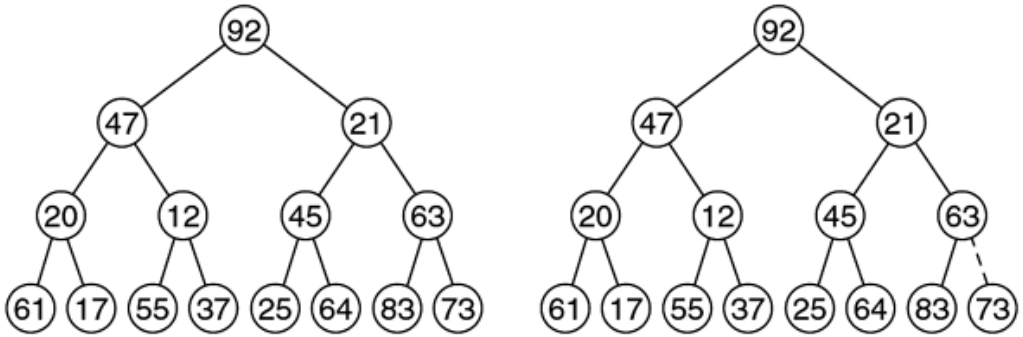


figure 21.17

(a) Initial heap;
(b) after
`percolateDown(7)`

figure 21.18

(a) After
`percolateDown(6);`
(b) after
`percolateDown(5)`

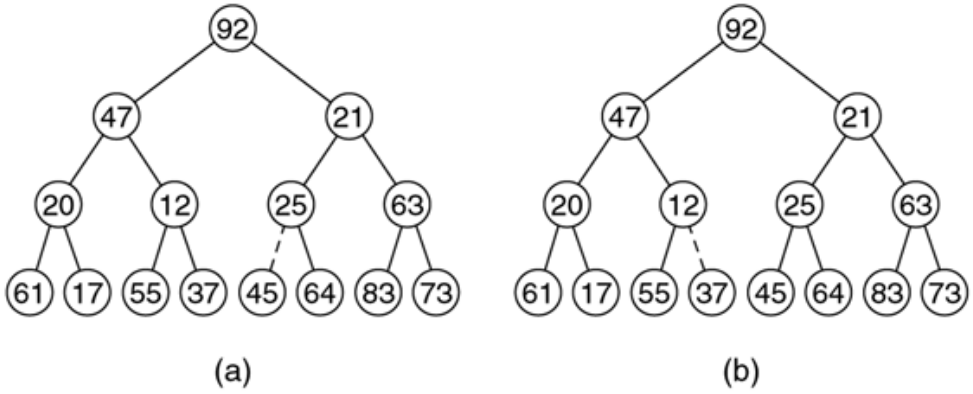


figure 21.19

(a) After `percolateDown(4);` (b) after `percolateDown(3)`

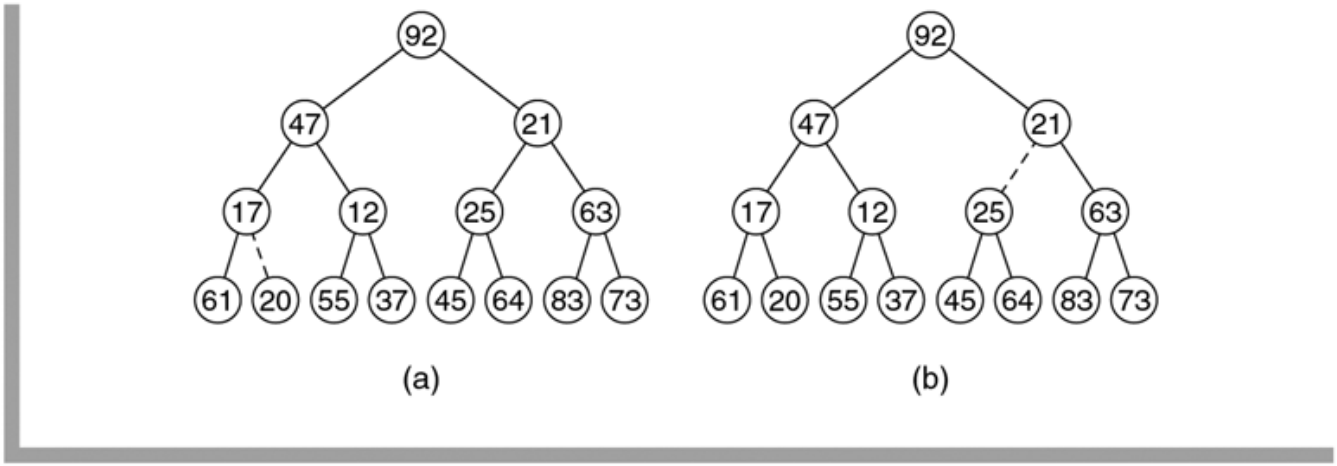
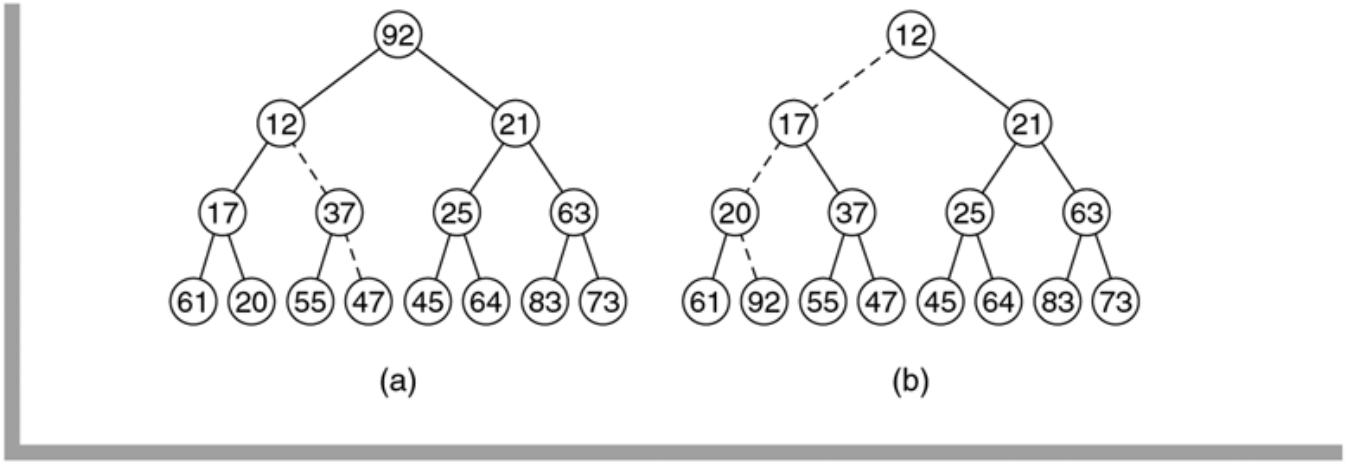


figure 21.20

(a) After `percolateDown(2);` (b) after `percolateDown(1)` and `buildHeap` terminates



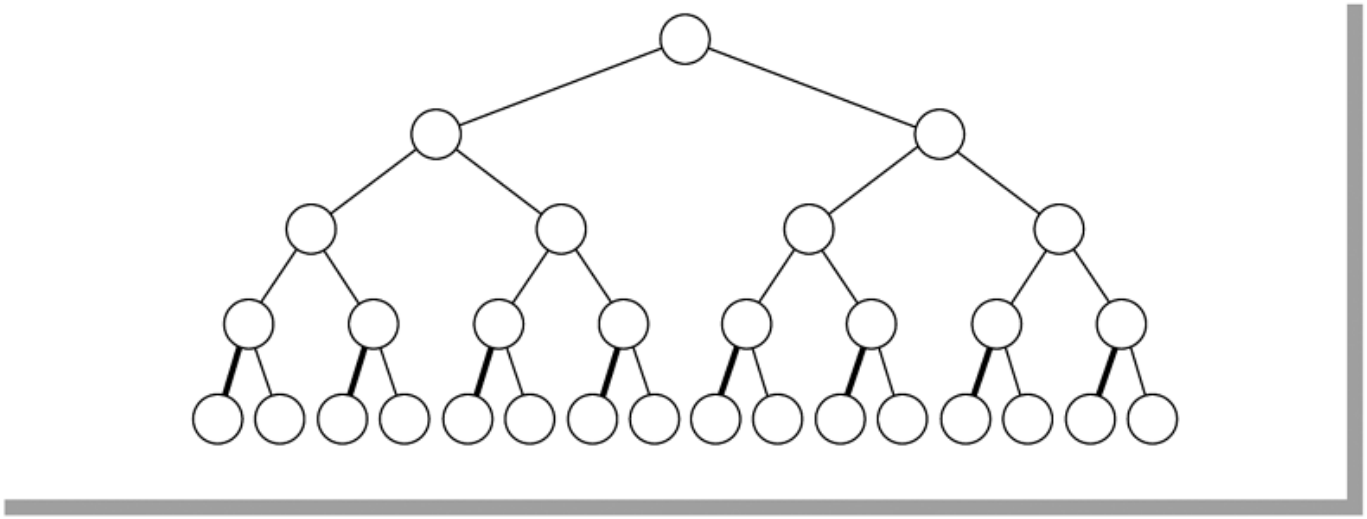


figure 21.21

Marking the left edges for height 1 nodes

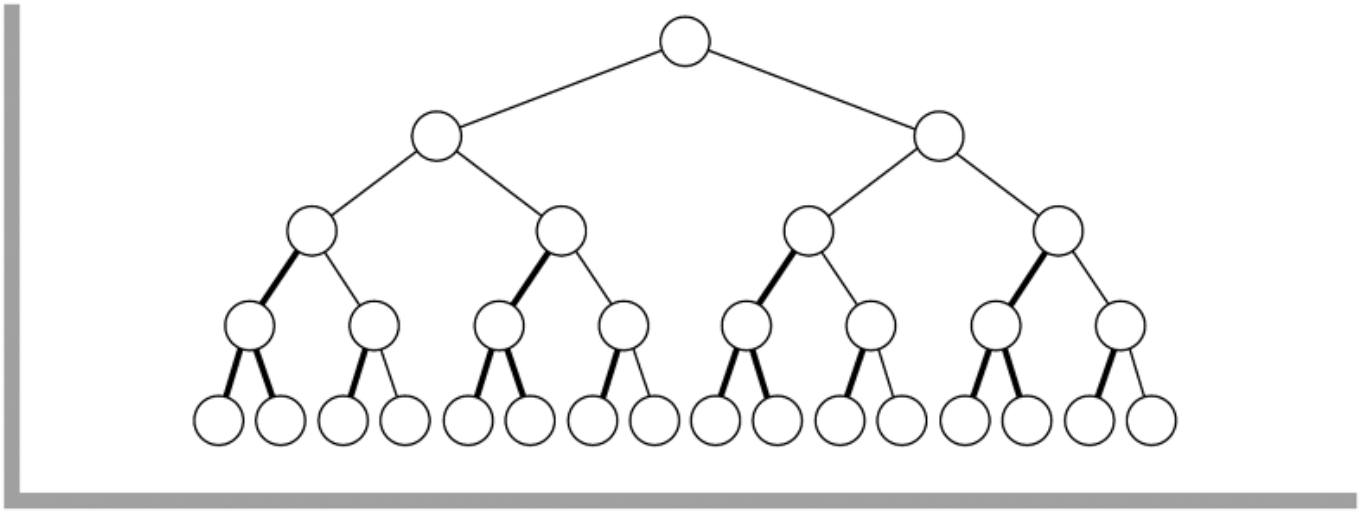


figure 21.22

Marking the first left edge and the subsequent right edge for height 2 nodes

figure 21.23

Marking the first left edge and the subsequent two right edges for height 3 nodes

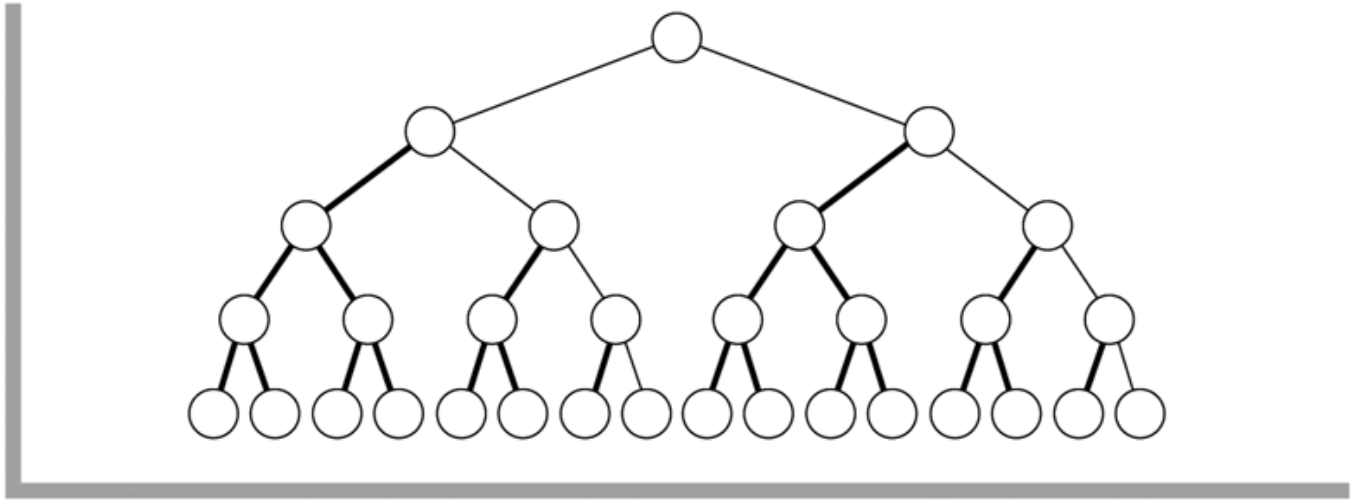
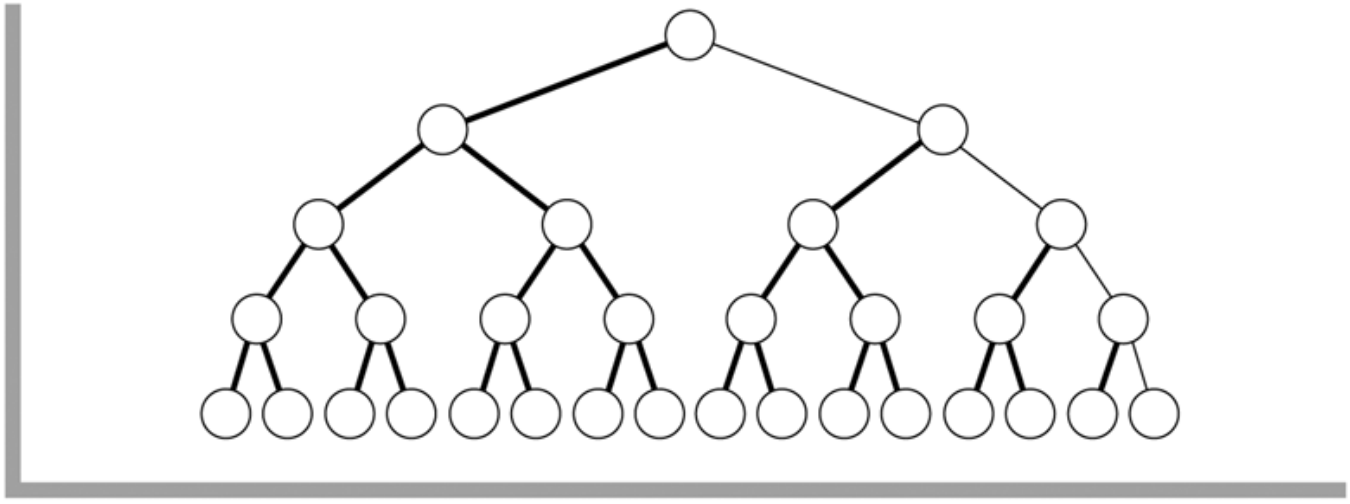


figure 21.24

Marking the first left edge and the subsequent two right edges for the height 4 node



Heapsort

(using a **max-heap**)

heap construction

```
void heapsort(Comparable[] a) {
    for (int i = a.length / 2 - 1; i >= 0; i--)
        percDown(a, i, a.length);
    for (int i = a.length - 1; i > 0; i--) {
        swapReferences(a, 0, i);
        percDown(a, 0, i);
    }
}
```

$O(N)$

$O(N \log N)$

root: `a[0]`
current heap size: `i`
children of `i`: `array[2*i+1]` and `array[2*i+2]`
parent of `i`: `array[(i-1)/2]`

- Time complexity of heapsort is $O(N \log N)$.
- No extra space needed.

Implementation of percDown

(solution of Exercise 21.23)

```
private static <AnyType extends Comparable<? super AnyType>>
void percDown(AnyType[] a, int index, int size) {
    int child;
    AnyType tmp;

    for (tmp = a[index]; 2 * index + 1 < size; index = child) {
        child = 2 * index + 1;
        if (child + 1 < size && a[child].compareTo(a[child + 1]) < 0)
            child++;
        if (tmp.compareTo(a[child]) < 0)
            a[index] = a[child];
        else
            break;
    }
    a[index] = tmp;
}
```

Heapsort example

59	26	58	21	41	97	21	16	26	53

Construct max-heap:

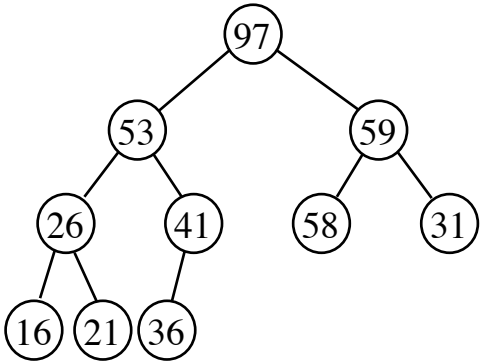
97	53	59	26	41	58	31	16	21	36

59	53	58	26	41	36	31	16	21	97

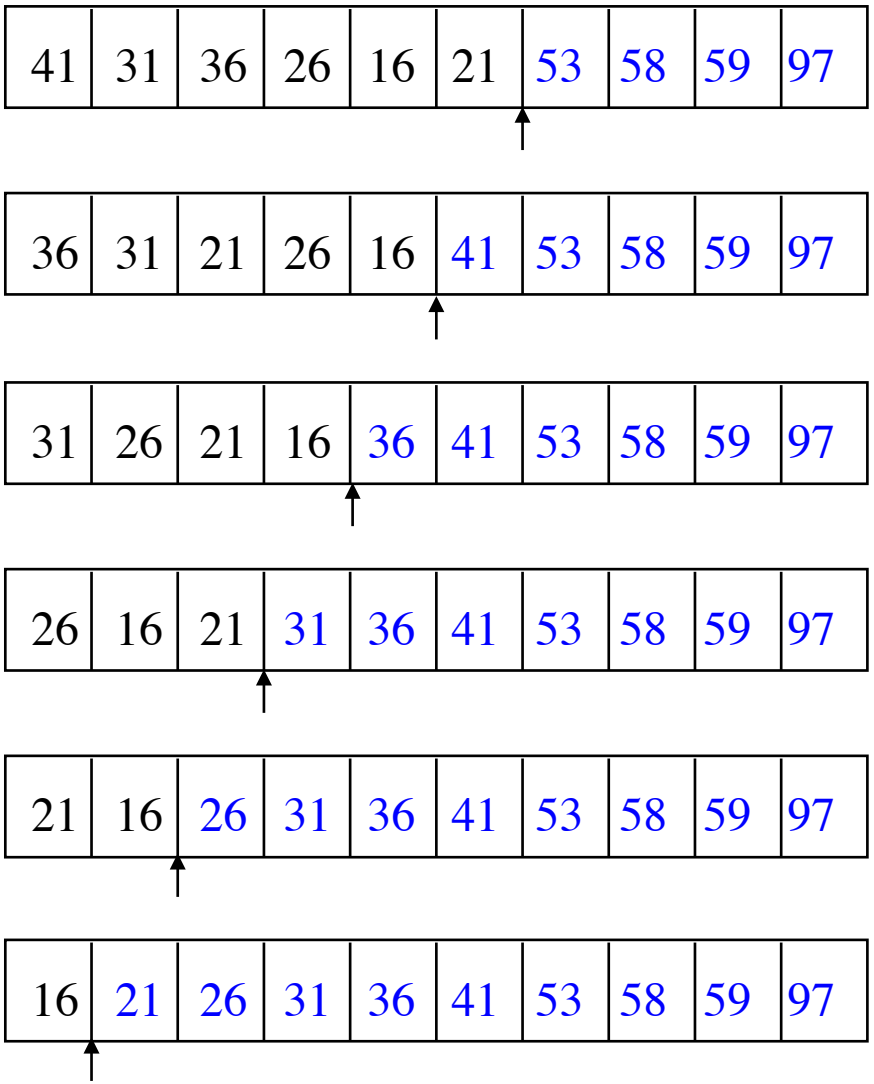
58	53	36	26	41	21	31	16	59	97

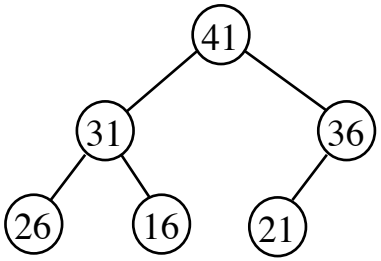
53	41	36	26	16	21	31	58	59	97

41	31	36	26	16	21	53	58	59	97



to be continued

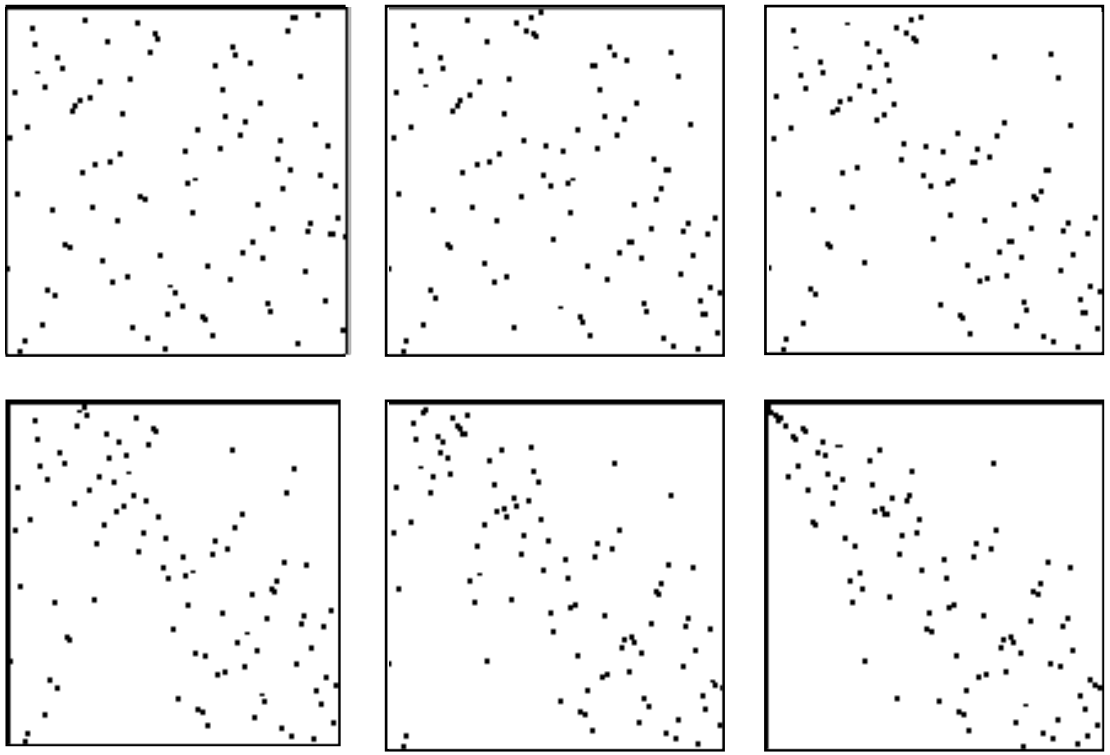

41 31 36 26 16 21 53 58 59 97
36 31 21 26 16 41 53 58 59 97
31 26 21 16 36 41 53 58 59 97
26 16 21 31 36 41 53 58 59 97
21 16 26 31 36 41 53 58 59 97
16 21 26 31 36 41 53 58 59 97


41
31
36
26
16
21

end

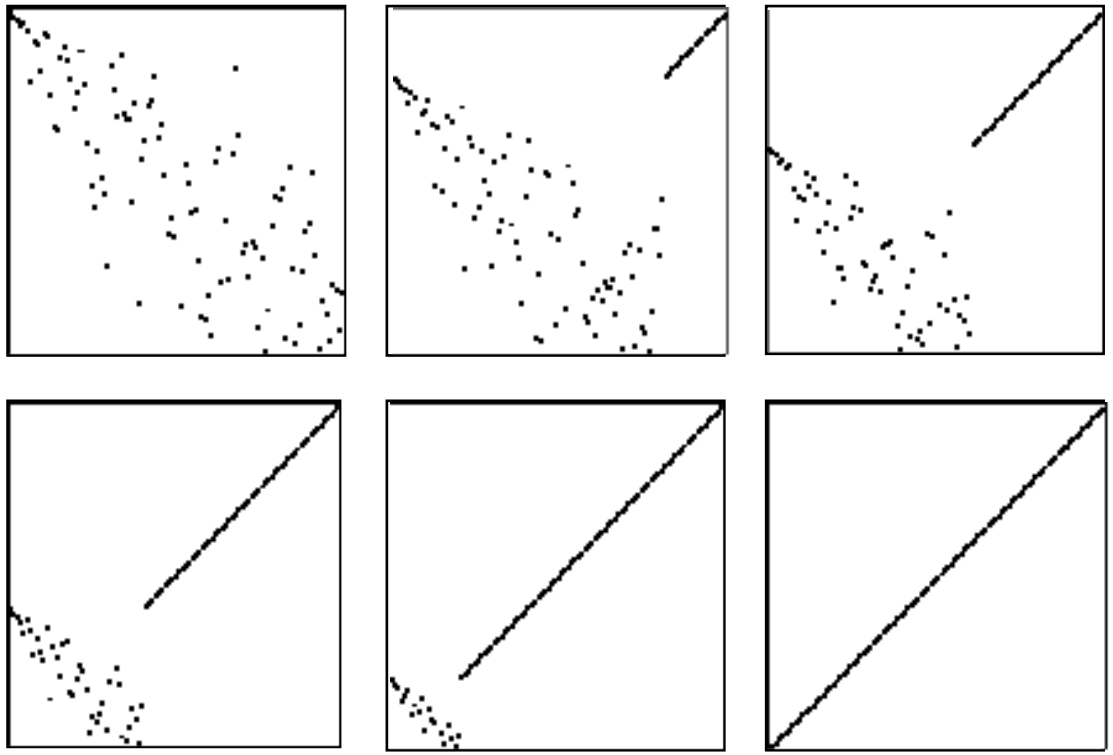
Animation of heapsort

(heap construction phase)



Animation of heapsort

(sorting phase)



External sorting

(sorting of data on external memory)

Special considerations:

(1) It is very time consuming to access a data element

(2) There can be restrictions on how data can be accessed, e.g., data on a magnetic tape can only be read sequentially.

Distribute and merge

Distribute:

Divide the file to be sorted into blocks, each of size equal to the internal memory.
Sort each block and distribute them to two or more temporary files.

Merge:

Merge the sorted blocks (*runs*) into longer sorted blocks.
Continue in this way until the original file is one sorted block.

Balanced multiway merge

Example: 3-way sorting of 81 records

Sorted blocks (number of records)

1	**9** (3)	**0**	**1** (27)	**0**
2	**9** (3)	**0**	**1** (27)	**0**
3	**9** (3)	**0**	**1** (27)	**0**
4	**0**	**3** (9)	**0**	**1** (81)
5	**0**	**3** (9)	**0**	
6	**0**	**3** (9)	**0**	

It takes 3 passes to sort 81 records

The merging may be performed by a priority queue.

Balanced k-way merge

N: number of records
M: size of internal memory (measured in number of records)

Use half of the $2k$ files as input files, the rest as output files.

Pass 0: divide the file into blocks of size M, sort each block and distribute the sorted blocks to files $1, 2, ..., k$.
Pass 1: k-merge the blocks from files $1, 2, ..., k$ into blocks of size kM and write them to files $k+1, k+2, ..., 2k$.
Pass 2: k-merge the blocks from files $k+1, k+2, ..., 2k$ into blocks of size k^2M and write them to files $1, 2, ..., k$.
...
Pass p: k-merge the blocks from the input files to one block of size k^pM and write it to one of the output files.

The file is sorted when

$k^p M \geq N$

i.e, after

$p = \log_k(N/M)$ passes.

Examples:

file size (N)	10^9 records
memory size (M)	10^6 records

number of temporary files ($2k$)		4
number of passes (p)	$\log_2 10^3 \approx$	**10**

number of temporary files (2k)		20
number of passes (p)	$\log_{10} 10^3 =$	**3**

The file is sorted in a time that is 4-11 times longer than the time is takes to read or write it.

Polyphase merge

Reduces the number of temporary files to about the half the number of temporary files needed for balanced multiway merge.

Principle:

Always use k-1 input files and 1 output file.

Algorithm:

Merge from the k-1 input files to the output file, until the end of one of the input files is reached.
Use the latter file as new output file for merging from the other k-1 files.
Continue in this way until the file is sorted.

Run distribution for polyphase merge

Distribute the runs on k-1 files such that the last merge causes the end to be reached on all input files simultaneously.

The initial run distribution can be determined using *generalized* Fibonacci numbers.

1	21	8	0	5	3	1	0	1
2	13	0	8	3	0	2	1	0
3	0	13	5	0	2	0	1	0

3 files, 34 runs, 7 passes

$$F^k(N) = F^k(N\text{-}1) + F^k(N\text{-}2) + \ldots + F^k(N\text{-}k)$$
$$F^k(0 \le N \le k\text{-}2) = 0,\ F^k(k\text{-}1) = 1$$

Replacement selection

A technique that produces initial runs of average length $2M$ if the input is randomly distributed.

Read M records into a priority queue.

Delete the smallest record from the priority queue and write it out.

Read a record from the input file. If the new element is smaller than the last one output, it cannot become part of the current run. Mark it as belonging to the next run and treat it as greater than all the unmarked elements in the queue.

Terminate the run when a marked element reaches the top of the queue.

Organize the marked records as priority queue.

	Three Elements in Heap Array				
	array[1]	array[2]	array[3]	Output	Next Item Read
Run 1	11	94	81	11	96
	81	94	96	81	12
	94	96	12	94	35
	96	35	12	96	17
	17	35	12	End of Run	Rebuild
Run 2	12	35	17	12	99
	17	35	99	17	28
	28	99	35	28	58
	35	99	58	35	41
	41	99	58	41	75
	58	99	75	58	15
	75	99	15	75	End of Tape
	99		15	99	
			15	End of Run	Rebuild
Run 3	15			15	

figure 21.38

Example of run construction

$M = 3$