Basic Data-Structures & Algorithms Complexity

This document:

https://github.com/Arnaud-Nauwynck/presentations/blob/main/java/Basic-Data-Structures.pdf

Outline

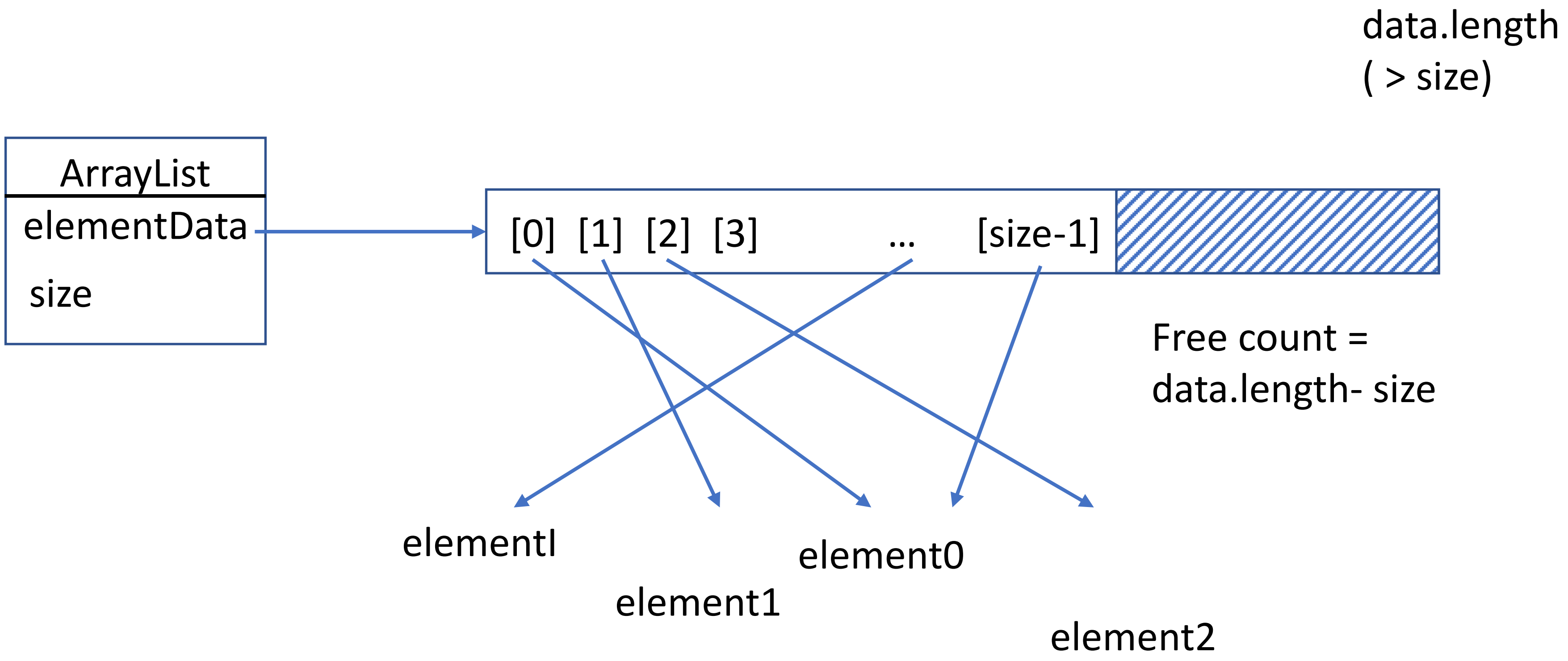
- ArrayList
  - add / indexOf / remove
  - resize
- LinkedList
- Arrays.sort, Tim Sort
- HashMap
  - hashCode, modulo or power 2, collisions
  - resize
  - Put, get, remove, iterate
- TreeMap
  - balanced tree, red-black tree
  - Put, get, remove, iterate
- PriorityQueue
  - Offer, poll

All with
Schemas
+ Algorithms
+ Associated Complexity
O(1), O(log N), O(N), O(N log N), O(n2)..

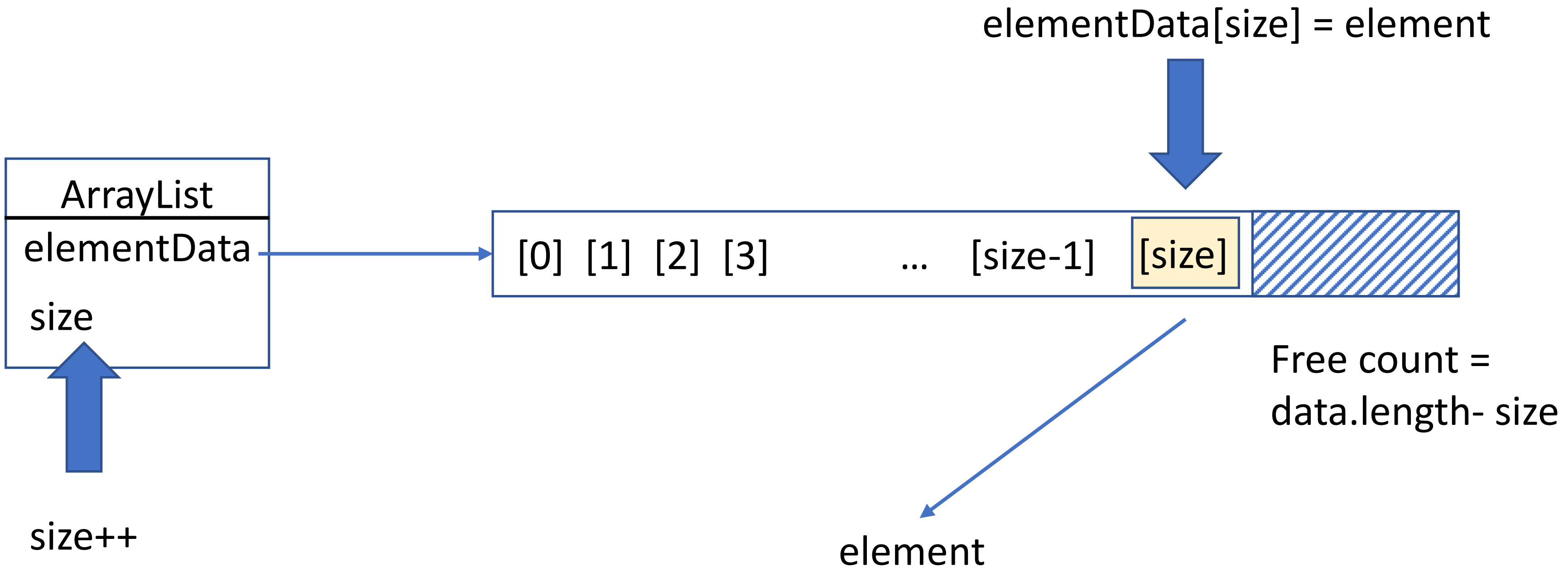
ArrayList<T>

ArrayList

```
 * The array buffer into which the elements of the ArrayList are stored.
transient Object[] elementData; // non-private to simplify nested class access

 * The size of the ArrayList (the number of elements it contains).
private int size;
```



ArrayList.add(element)
If Fast case : enough allocated length



Complexity = O(1) ... 1 logical operation

Big O Notation

f(x) is O(g(x)) means: f(x) < M g(x) for x > x0 and M>0

Big *O* notation

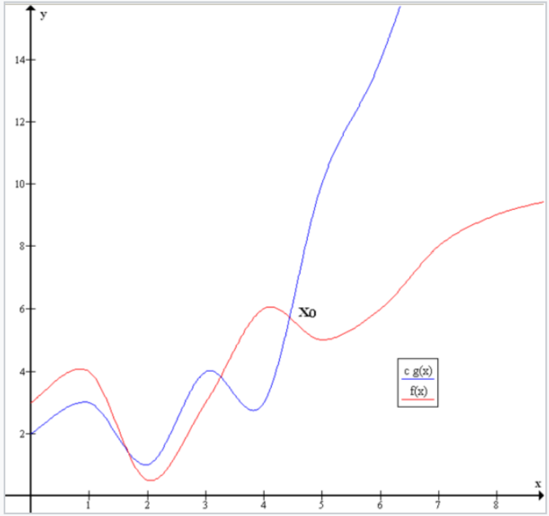
From Wikipedia, the free encyclopedia

Big *O* notation is a mathematical notation that describes the limiting behavior of a function when the argument tends towards a particular value or infinity. Big O is a member of a family of notations invented by Paul Bachmann,[1] Edmund Landau,[2] and others, collectively called **Bachmann–Landau notation** or **asymptotic notation**. The letter O was chosen by Bachmann to stand for *Ordnung*, meaning the order of approximation.

In computer science, big O notation is used to classify algorithms according to how their run time or space requirements grow as the input size grows.[3] In analytic number theory, big O notation is often used to express a bound on the difference between an arithmetical function and a better understood approximation; a famous example of such a difference is the remainder term in the prime number theorem. Big O notation is also used in many other fields to provide similar estimates.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as the **order of the function**. A description of a function in terms of big O notation usually only provides an upper bound on the growth rate of the function.

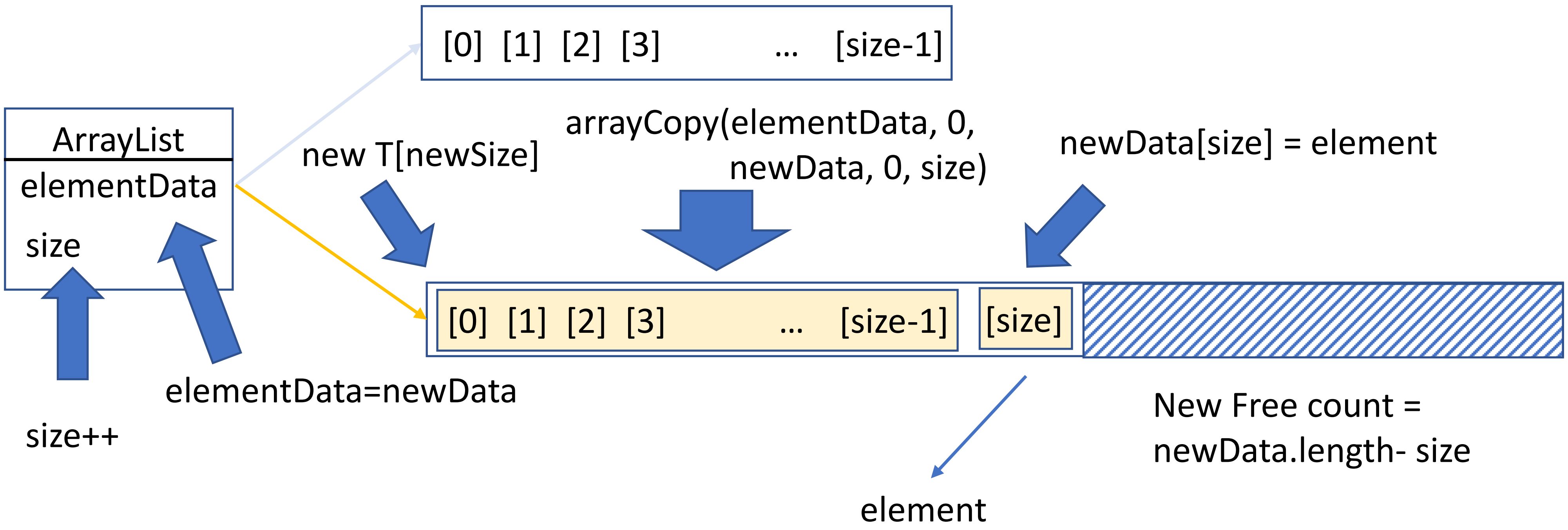
Associated with big O notation are several related notations, using the symbols o, Ω, ω, and Θ, to describe other kinds of bounds on asymptotic growth rates.



Example of Big O notation: $f(x) \in O(g(x))$ as there exists $M > 0$ (e.g., $M = 1$) and x_0 (e.g., $x_0 = 5$) such that $f(x) \leq Mg(x)$ whenever $x \geq x_0$.

Contents [hide]

ArrayList.add(element)
If Slow case : NOT enough allocated length



Complexity = O(size) ... need copy all existing element pointers

How much to allocate in advance?

```java
 * Increases the capacity to ensure that it can hold at least the
private void grow(int minCapacity) {
    // overflow-conscious code
    int oldCapacity = elementData.length;
    int newCapacity = oldCapacity + (oldCapacity >> 1);
    if (newCapacity - minCapacity < 0)
        newCapacity = minCapacity;
    if (newCapacity - MAX_ARRAY_SIZE > 0)
        newCapacity = hugeCapacity(minCapacity);
    // minCapacity is usually close to size, so this is a win:
    elementData = Arrays.copyOf(elementData, newCapacity);
}
```

compromize

+1 +2 +3 … +size/2 +… ???

Bad compromize:
Need copy every time adding

Bad compromize:
Lot of wasted memory if no more add

Memory preference

Compute preference

ArrayList initialCapacity : 0 .. then 10

```java
 * Default initial capacity.
private static final int DEFAULT_CAPACITY = 10;

public ArrayList() {
    this.elementData = DEFAULTCAPACITY_EMPTY_ELEMENTDATA;
}

private static int calculateCapacity(Object[] elementData, int minCapacity) {
    if (elementData == DEFAULTCAPACITY_EMPTY_ELEMENTDATA) {
        return Math.max(DEFAULT_CAPACITY, minCapacity);
    }
    return minCapacity;
}
```

```java
 * Constructs an empty list with the specified initial capacity.
public ArrayList(int initialCapacity) {
    if (initialCapacity > 0) {
        this.elementData = new Object[initialCapacity];
```

Use explicit if known size

Loop … add(element)
=> How many waste copies / times ?

```java
ArrayList<Object> ls = new ArrayList<>();
for(int i = 0; i < 1_000_000; i++) {
    ls.add(null);
}
// => count equivalent grows + copies
int countGrow = 0;
int totalCopy = 0;
for(int capacity = 10; capacity < 1_000_000; capacity = capacity + (capacity >> 1)) {
    countGrow++;
    totalCopy += capacity;
    System.out.println("[" + countGrow + "] " + capacity + " .. " + totalCopy);
}
```

Count :	1	2	3	4	5	6		28	29
Capacity :	10	15	22	33	49	73	….	540217	810325
Total Copies:	10	10+15= 25	25+22=47	80	129	202		1620647	2430972

Total : 2.4 Million copies
To add 1 Million values

Average Cost

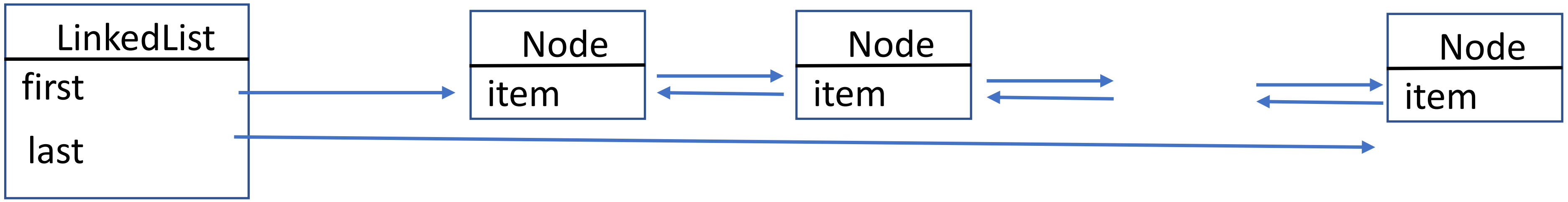
```
* <p>The <tt>size</tt>, <tt>isEmpty</tt>, <tt>get</tt>, <tt>set</tt>,
* <tt>iterator</tt>, and <tt>listIterator</tt> operations run in constant
* time.  The <tt>add</tt> operation runs in <i>amortized constant time</i>,
* that is, adding n elements requires O(n) time.  All of the other operations
* run in linear time (roughly speaking).  The constant factor is low compared
* to that for the <tt>LinkedList</tt> implementation.
```

Total copies = O(N) .... 2.4 * N for 1M

Number of reallocations: O(log N)

LinkedList … even worse !

```
* time.  The <tt>add</tt> operation runs in <i>amortized constant time</i>,
* that is, adding n elements requires O(n) time.  All of the other operations
* run in linear time (roughly speaking).  The constant factor is low compared
* to that for the <tt>LinkedList</tt> implementation.
```




```
 * Pointer to first node.
transient Node<E> first;

 * Pointer to last node.
transient Node<E> last;
```

```
private static class Node<E> {
    E item;
    Node<E> next;
    Node<E> prev;
```

Problem … « RAM » Random Access Memory … cache L1, L2

Typical access time :

RAM is ~100 nanos … 1000 x SLOWER than L1

But Typical capacity :

L1 is ~15 Ko … 1M x SMALLER than Ram (~100 Go)

Algorithm cost is driven by **Cache misses + Look Aheads**

ArrayList.indexOf(element)

```java
 * Returns the index of the first occurrence of the specified element
public int indexOf(Object o) {
    if (o == null) {
        for (int i = 0; i < size; i++)
            if (elementData[i]==null)
                return i;
    } else {
        for (int i = 0; i < size; i++)
            if (o.equals(elementData[i]))
                return i;
    }
    return -1;
}
```

Average Cost indexOf()

Positive find => … return break loop maybe at position 1, or 2, … or N

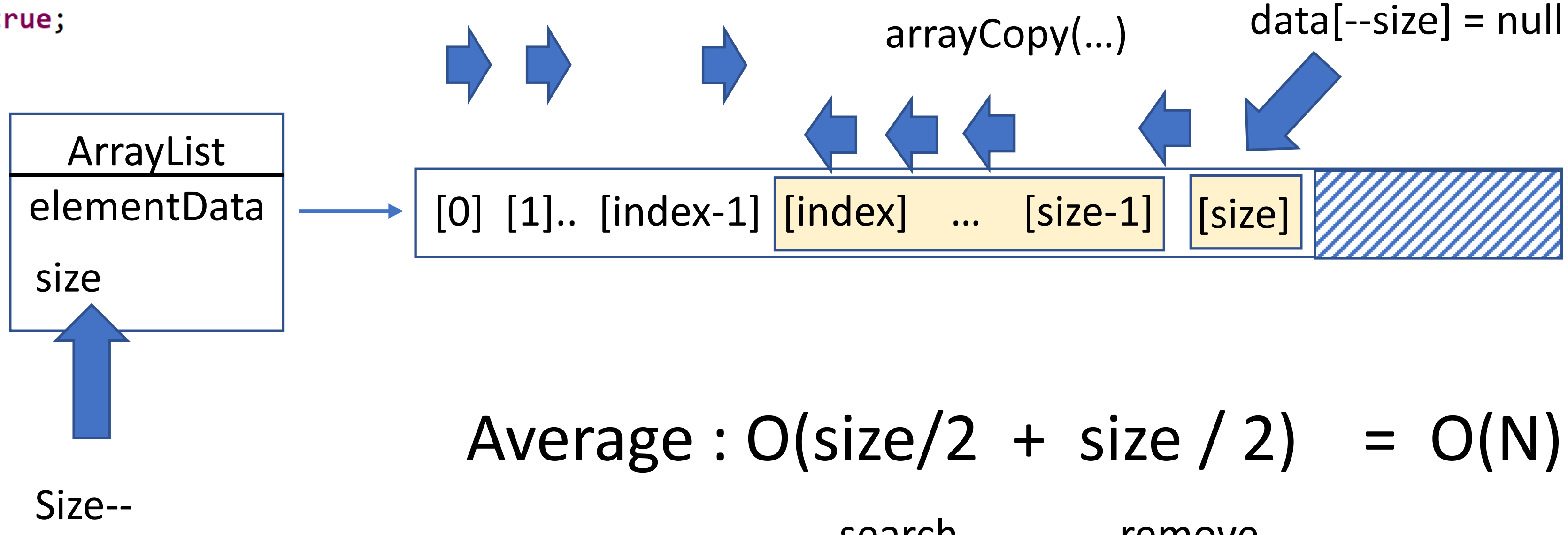
$$\frac{1+2+3+ \dots + N}{N} = \frac{(N+1) * N}{2\,N} = \frac{N+1}{2}$$ = half scan in average

Negative find => … always full scan $= N$

ArrayList.remove(element) = indexOf + fastRemove by index

```java
 * Removes the first occurrence of the specified element from this list,
public boolean remove(Object o) {
    if (o == null) {
        for (int index = 0; index < size; index++)
            if (elementData[index] == null) {
                fastRemove(index);
                return true;
            }
    } else {
        for (int index = 0; index < size; index++)
            if (o.equals(elementData[index])) {
                fastRemove(index);
                return true;
            }
    }
    return false;
}
```

```java
private void fastRemove(int index) {
    modCount++;
    int numMoved = size - index - 1;
    if (numMoved > 0)
        System.arraycopy(elementData, index+1, elementData, index,
                         numMoved);
    elementData[--size] = null; // clear to let GC do its work
}
```



Average : O(size/2 + size / 2) = O(N)

search remove

ArrayList.remove(index)

list.remove(0)

WORSE CASE !!! O(N)
Need shifting ALL elements

list.remove(list.size()- 1)

OK… O(1)
Always prefer treating « removing » elements at end

HashMap<K,V>

HashMap<K,V>

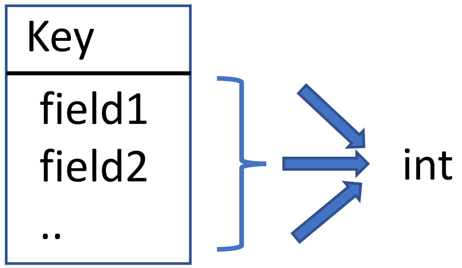
```java
 * The table, initialized on first use, and resized as
transient Node<K,V>[] table;

 * Holds cached entrySet(). Note that AbstractMap fields are used
transient Set<Map.Entry<K,V>> entrySet;

 * The number of key-value mappings contained in this map.
transient int size;

 * The number of times this HashMap has been structurally modified
transient int modCount;

 * The next size value at which to resize (capacity * load factor).
// (The javadoc description is true upon serialization.
// Additionally, if the table array has not been allocated, this
// field holds the initial array capacity, or zero signifying
// DEFAULT_INITIAL_CAPACITY.)
int threshold;

 * The load factor for the hash table.
final float loadFactor;
```

```java
/**
 * Basic hash bin node, used for most entries.  (See below for
 * TreeNode subclass, and in LinkedHashMap for its Entry subclass.)
 */
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;
```

… LinkedHashMap<K,V>

```java
 * HashMap.Node subclass for normal LinkedHashMap entries.
static class Entry<K,V> extends HashMap.Node<K,V> {
    Entry<K,V> before, after;
```

hashCode()
injective function for Key -> int



Mix all bits information
Example using « ^ » (xor) :
int x,int y => x ^ y
long value => (value ^ (value>>>32))
Example using prime: x*31+y

Expected properties:

1/ repeatable (same object => same result .. No time or address dependent)

2/ when key1.equals(key2) => same hashCode

3/ when different key => expect BUT NOT mandatory to have different hashCode

4/ use all bits information on key   (no naive return 123, even if legal)

5/ try to reach all possible values of « int » range: [-2^31… +2^31]

Hashing Function,
then Modulo « % » or Bitwise « & »

```java
 * Computes key.hashCode() and spreads (XORs) higher bits of hash
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}
```

```java
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
```

.. Mostly equivalent to « modulo »

hash = Key.hashCode()

… « int » in range [-2^31, +2^31]

hash2 = Math.abs(hash)

… in [0, +2^31] .. Lost 1 bit

index = hash2 % M

Randomly distributed when M is Prime

… in range [0, size-1]
OK to lookup in array « table[size] »

Jdk .. Use capacity=2^power

```java
/**
 * The default initial capacity - MUST be a power of two.
 */
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16
```

Hash to index .. using « **hash & (n-1)** »

Example: (in base 2)

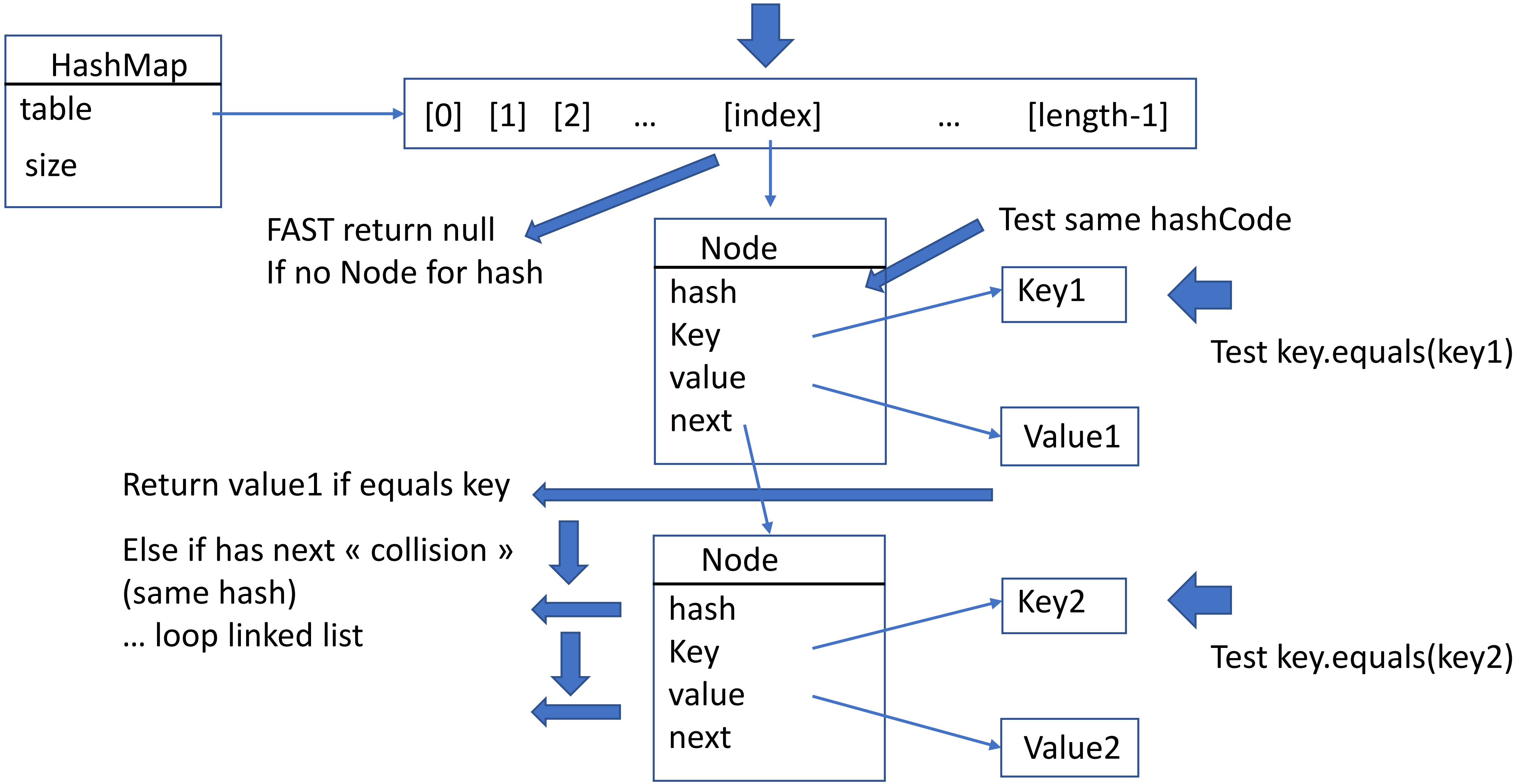
N	=32 = (1 << 5) =	0 0 0 1 0 0 0 0 0
n-1 = 31 =		0 0 0 0 1 1 1 1 1
Hash =		y y y y x x x x x
Hash&(n-1)=		0 0 0 0 x x x x x

Resize: capacity *2 .. using « **capacity << 1** »
(jdk : no decrease capacity)

Re-hash after resize => index change to index or index*2

hashCode()
… lookup index for finding value by key if present

index = abs(key.hashCode()) % M



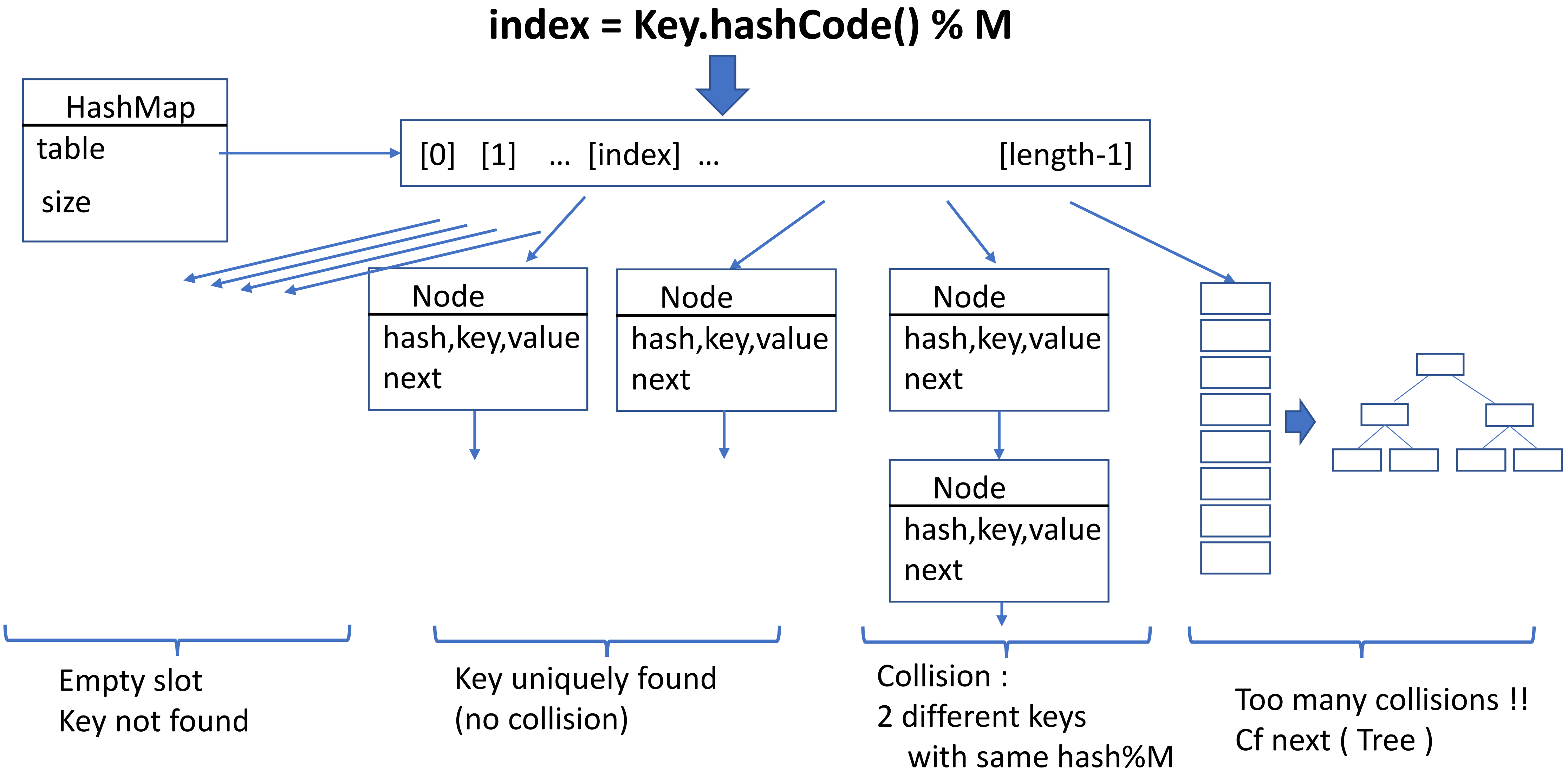
Strategy to Reduce Collisions Occurrences

1/ use « GOOD » hashCode() function …. With hopefully equally distributed values

2/ use « BIG » M allocated tables length (example: M > 2 * N)
… wasting memory, but improving selectivity (avoid %M collisions)

3/ if using « hashCode % M » … prefer M as Prime number
(NOT the implementation of jdk)

Same « hashCode()%M » : collisions



LinkedList to Tree … >= 8

```
/**
 * The bin count threshold for using a tree rather than list for a
 * bin.  Bins are converted to trees when adding an element to a
 * bin with at least this many nodes. The value must be greater
 * than 2 and should be at least 8 to mesh with assumptions in
 * tree removal about conversion back to plain bins upon
 * shrinkage.
 */
static final int TREEIFY_THRESHOLD = 8;
```

HashMap.get(key) ... O(1)

```java
 * Returns the value to which the specified key is mapped,
public V get(Object key) {
    Node<K,V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

 * Implements Map.get and related methods.
final Node<K,V> getNode(int hash, Object key) {
    Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
    if ((tab = table) != null && (n = tab.length) > 0 &&
        (first = tab[(n - 1) & hash]) != null) {
        if (first.hash == hash && // always check first node
            ((k = first.key) == key || (key != null && key.equals(k))))
            return first;
        if ((e = first.next) != null) {
            if (first instanceof TreeNode)
                return ((TreeNode<K,V>)first).getTreeNode(hash, key);
            do {
                if (e.hash == hash &&
                    ((k = e.key) == key || (key != null && key.equals(k))))
                    return e;
            } while ((e = e.next) != null);
        }
    }
    return null;
}
```

Fast path
O(1) ...no collision

many collisions
O(log(C)) ...

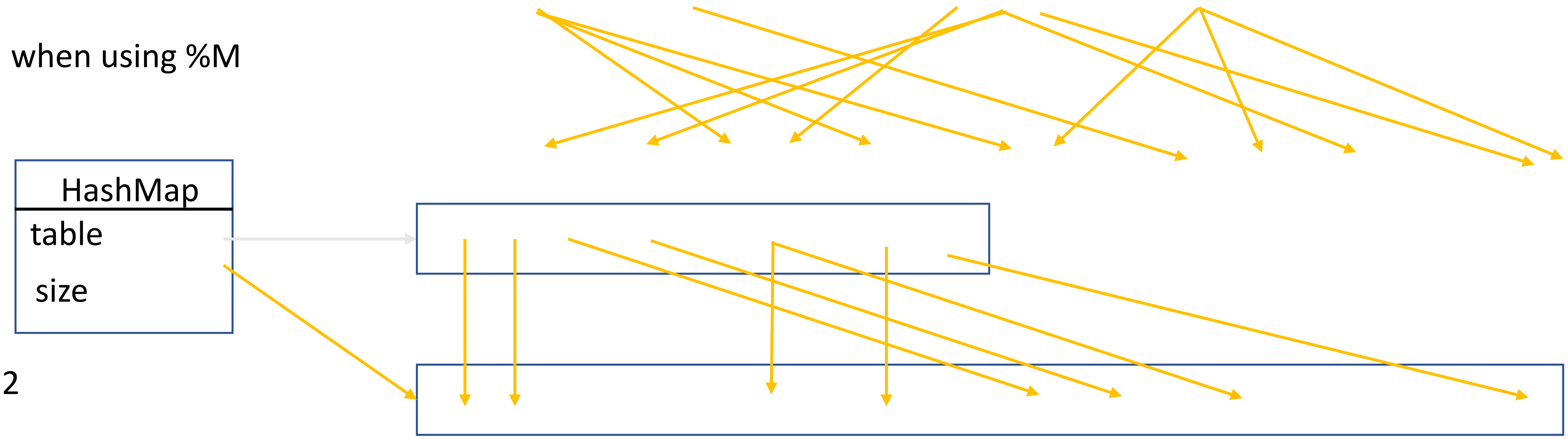
medium path
O(8/2) ...few collision(s)

HashMap.put() … may resize

Hash size « M » should be adapted to number of element « N » greater (but not too much)

.. Sometimes re-copy all ! … reduce collisions

General rehash when using %M



FAST rehash
Using power of 2

Rehash is FAST … (not recomputed, and index unchanged OR *2)

HashMap.resize(): ?>threshold and capacity*2

```java
    if (++size > threshold)
        resize();
    afterNodeInsertion(evict);
    return null;
}

 * Initializes or doubles table size.  If null, allocates in
final Node<K,V>[] resize() {
    Node<K,V>[] oldTab = table;
    int oldCap = (oldTab == null) ? 0 : oldTab.length;
    int oldThr = threshold;
    int newCap, newThr = 0;
    if (oldCap > 0) {
        if (oldCap >= MAXIMUM_CAPACITY) {
            threshold = Integer.MAX_VALUE;
            return oldTab;
        }
        else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                 oldCap >= DEFAULT_INITIAL_CAPACITY)
            newThr = oldThr << 1; // double threshold
    }
    else if (oldThr > 0) // initial capacity was placed in threshold
        newCap = oldThr;
    else {               // zero initial threshold signifies using defaults
        newCap = DEFAULT_INITIAL_CAPACITY;
        newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
    }
    if (newThr == 0) {
        float ft = (float)newCap * loadFactor;
        newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY ?
                  (int)ft : Integer.MAX_VALUE);
    }
```

May resize after put()

capacity*2, threshold*2

Init : capacity=16,
loadFactor=0.75
..threshold=12

Loop put(k,v) ... how many resize() + total Re-hashes?

```java
HashMap<Integer,Integer> map = new HashMap<>();
for(int i = 0; i < 1_000_000; i++) {
    map.put(i, null);
}
// => count equivalent grows + rehashes
int capacity = 16;
int threshold = (int) (capacity * 0.75);
System.out.println("init " + capacity + ", threshold:" + threshold);
int countGrow = 0;
int totalRehash = 0;
for(; capacity < 1_000_000;) {
    countGrow++;
    totalRehash += threshold;
    capacity = capacity << 1;
    threshold = threshold << 1;
    System.out.println("[" + countGrow + "] " + capacity
            + ", " + threshold + " .. " + totalRehash);
}
```

```
init 16, threshold:12
[1] 32, 24 .. 12
[2] 64, 48 .. 36
[3] 128, 96 .. 84
[4] 256, 192 .. 180
[5] 512, 384 .. 372
[6] 1024, 768 .. 756
[7] 2048, 1536 .. 1524
[8] 4096, 3072 .. 3060
[9] 8192, 6144 .. 6132
[10] 16384, 12288 .. 12276
[11] 32768, 24576 .. 24564
[12] 65536, 49152 .. 49140
[13] 131072, 98304 .. 98292
[14] 262144, 196608 .. 196596
[15] 524288, 393216 .. 393204
[16] 1048576, 786432 .. 786420
```

Put 1 Million values => 16 resizes … 700k total rehashes

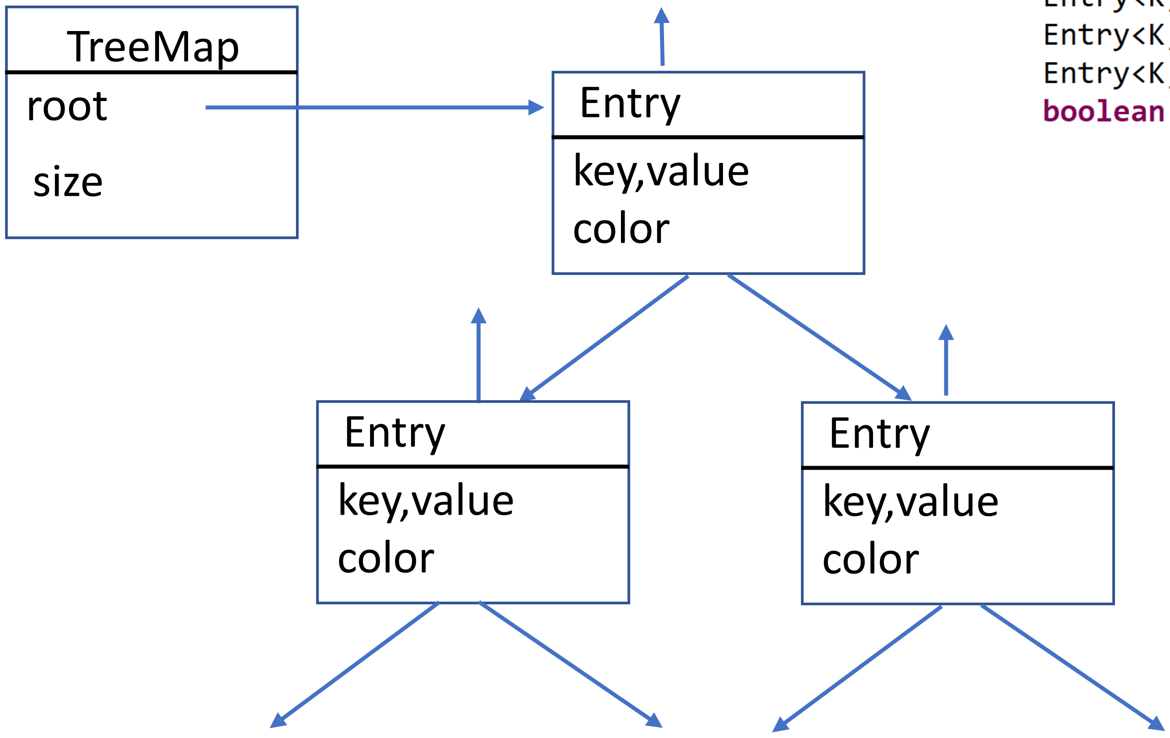
~ 0.8 N … O(N)

TreeMap<K,V>

TreeMap<K,V>

```java
private transient Entry<K,V> root;

 * The number of entries in the tree
private transient int size = 0;

 * The number of structural modifications to the tree.
private transient int modCount = 0;
```

```java
// Red-black mechanics

private static final boolean RED   = false;
private static final boolean BLACK = true;

 * Node in the Tree.  Doubles as a means to pass key-value pairs back to

static final class Entry<K,V> implements Map.Entry<K,V> {
    K key;
    V value;
    Entry<K,V> left;
    Entry<K,V> right;
    Entry<K,V> parent;
    boolean color = BLACK;
```



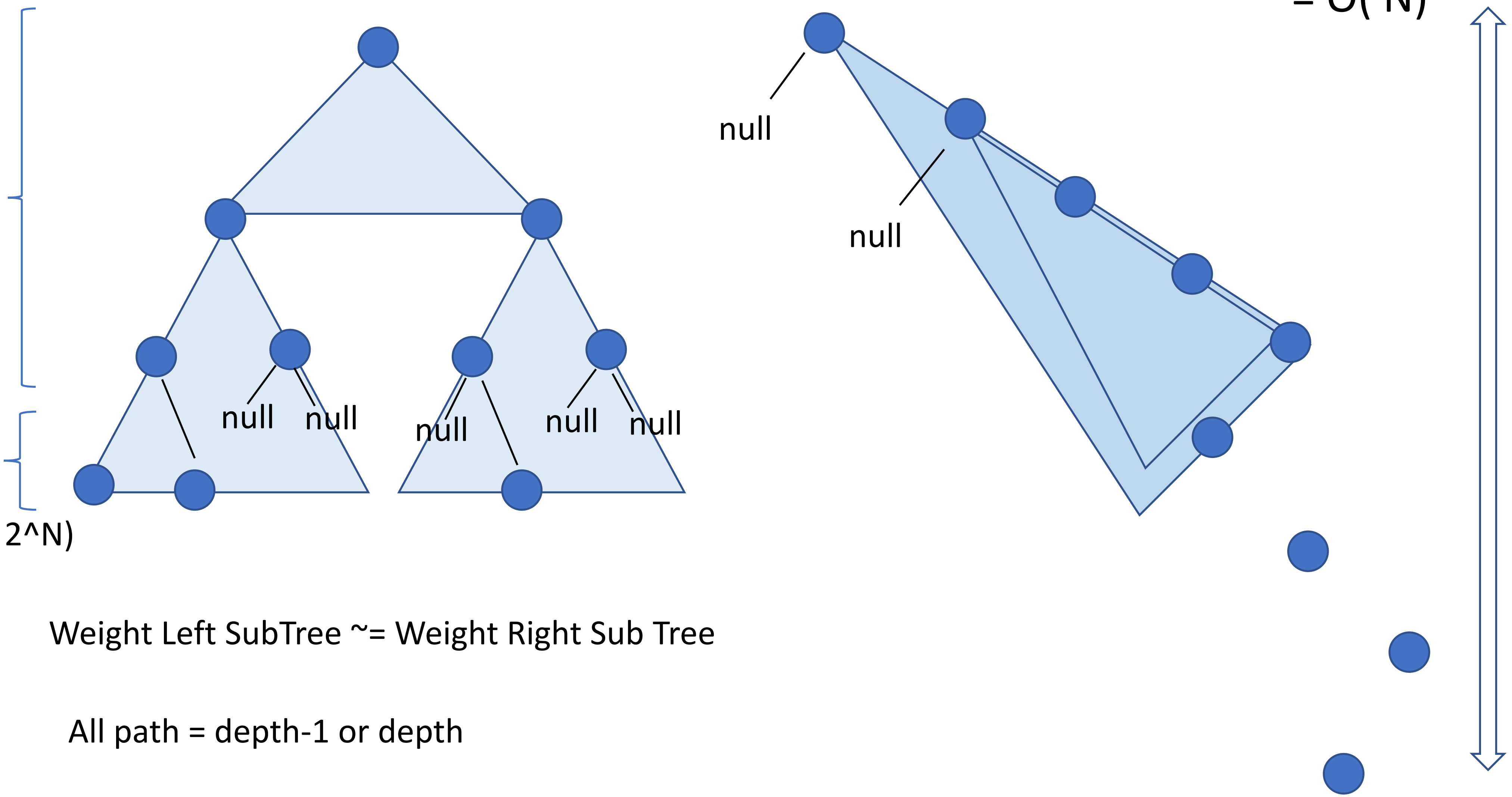
Balanced / Un-balanced Trees

tree depth
= O(log N)

intermediate
levels:
Hopefully
Entierly filled

leaf levels
may be empty
(not multiple of 2^N)

tree depth
= O(N)



Weight Left SubTree ~= Weight Right Sub Tree

All path = depth-1 or depth

TreeMap.get(key)

```java
 * Returns this map's entry for the given key, or {@code null} if the map
final Entry<K,V> getEntry(Object key) {
    // Offload comparator-based version for sake of performance
    if (comparator != null)
        return getEntryUsingComparator(key);
    if (key == null)
        throw new NullPointerException();
    @SuppressWarnings("unchecked")
        Comparable<? super K> k = (Comparable<? super K>) key;
    Entry<K,V> p = root;
    while (p != null) {
        int cmp = k.compareTo(p.key);
        if (cmp < 0)
            p = p.left;
        else if (cmp > 0)
            p = p.right;
        else
            return p;
    }
    return null;
}
```

Max operations
= tree depth
= O(log N)

N=2^depth

Level 0: root

Compare rootKey <-> key

Compare : > 0
=> go right

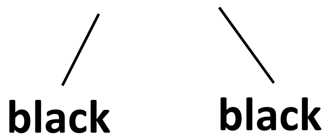
Compare < 0
=> go left

Compare = 0
=> found
OR no more child
=> stop

Red-Black Rules

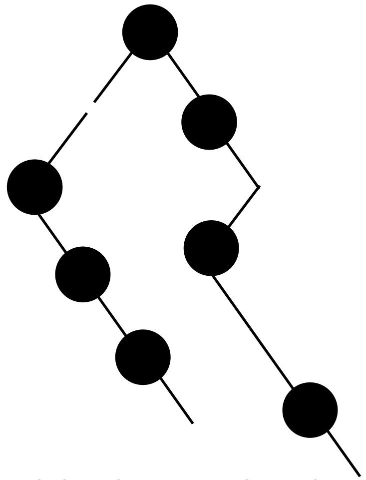
1 Every node is either red or black.

2 All NIL nodes (figure 1) are considered black.



3 A red node does not have a red child.

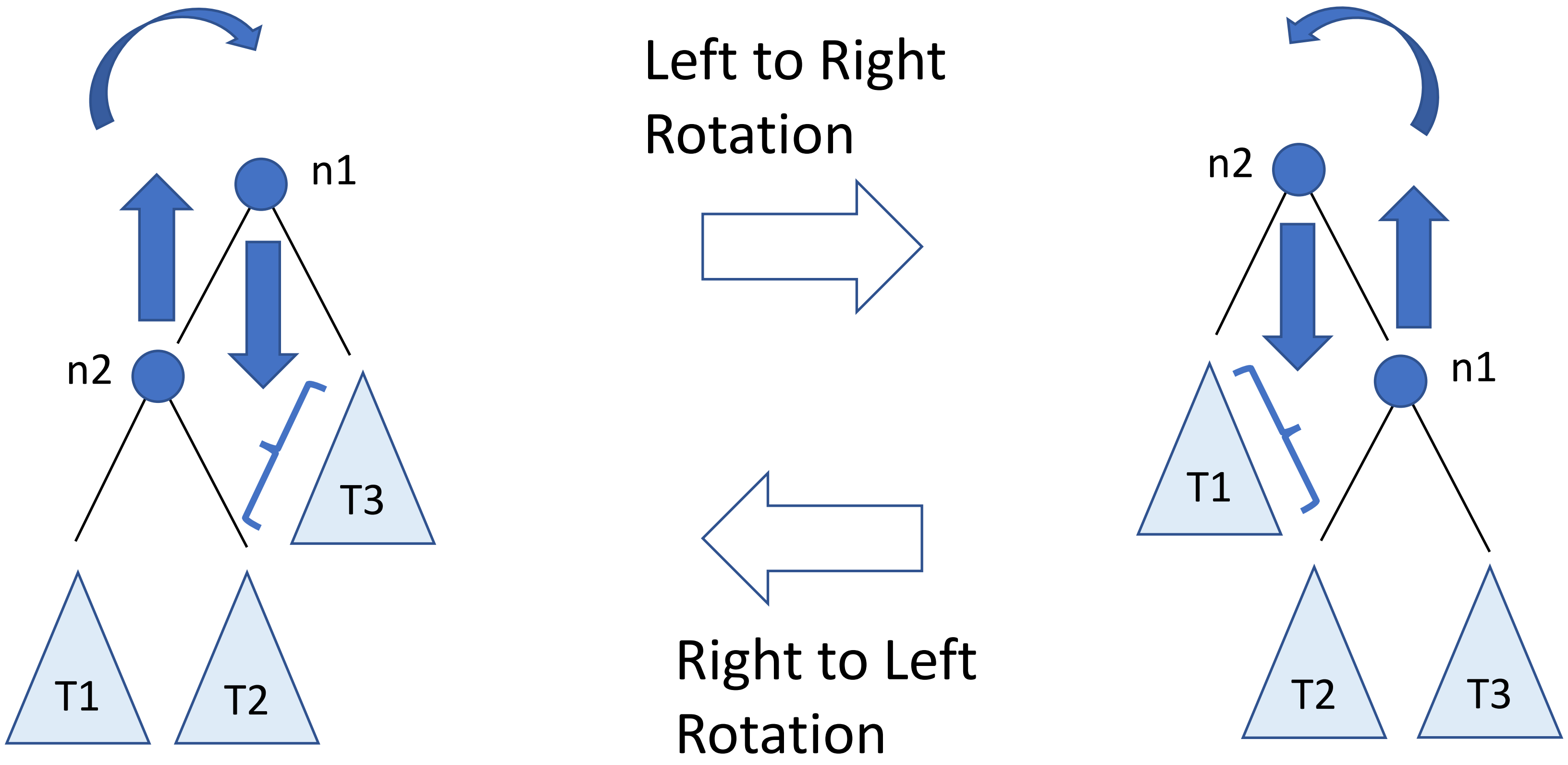
4 Every path from a given node to any of its descendant NIL nodes goes through the same number of black nodes.



n1 black = n2 back

=> the path from the root to the farthest leaf is no more than twice as long as the path from the root to the nearest leaf

Rotations



n1
n2
T1
T2
T3
Left to Right
Rotation
Right to Left
Rotation
n2
n1
T1
T2
T3

TreeMap.put(key, value) .. [1/2] find down insert point

```java
 * Associates the specified value with the specified key in this map.
public V put(K key, V value) {
    Entry<K,V> t = root;

        .....

    else {
        if (key == null)
            throw new NullPointerException();
        @SuppressWarnings("unchecked")
            Comparable<? super K> k = (Comparable<? super K>) key;
        do {
            parent = t;
            cmp = k.compareTo(t.key);
            if (cmp < 0)
                t = t.left;
            else if (cmp > 0)
                t = t.right;
            else
                return t.setValue(value);
        } while (t != null);
    }
    Entry<K,V> e = new Entry<>(key, value, parent);
    if (cmp < 0)
        parent.left = e;
    else
        parent.right = e;
    fixAfterInsertion(e);
    size++;
    modCount++;
    return null;
```

Level 0: root

Compare rootKey <-> key

Compare : > 0
=> go right

Compare < 0
=> go left

Compare = 0
=> setValue .. END
OR no more child
=> insert + fix

TreeMap.put(key, value) .. [2/2] fix up color+rotate

```java
private void fixAfterInsertion(Entry<K,V> x) {
    x.color = RED;

    while (x != null && x != root && x.parent.color == RED) {
        if (parentOf(x) == leftOf(parentOf(parentOf(x)))) {
            Entry<K,V> y = rightOf(parentOf(parentOf(x)));
            if (colorOf(y) == RED) {
                setColor(parentOf(x), BLACK);
                setColor(y, BLACK);
                setColor(parentOf(parentOf(x)), RED);
                x = parentOf(parentOf(x));
            } else {
                if (x == rightOf(parentOf(x))) {
                    x = parentOf(x);
                    rotateLeft(x);
                }
                setColor(parentOf(x), BLACK);
                setColor(parentOf(parentOf(x)), RED);
                rotateRight(parentOf(parentOf(x)));
            }
        } else {
            Entry<K,V> y = leftOf(parentOf(parentOf(x)));
            if (colorOf(y) == RED) {
                setColor(parentOf(x), BLACK);
                setColor(y, BLACK);
                setColor(parentOf(parentOf(x)), RED);
                x = parentOf(parentOf(x));
            } else {
                if (x == leftOf(parentOf(x))) {
                    x = parentOf(x);
                    rotateRight(x);
                }
                setColor(parentOf(x), BLACK);
                setColor(parentOf(parentOf(x)), RED);
                rotateLeft(parentOf(parentOf(x)));
            }
        }
    }
    root.color = BLACK;
}
```

Finished on black parent node
(or root)

Maybe fix color
+ maybe rotate

Maybe fix color
+ maybe rotate

Time cost … O(log N)

```
* <p>This implementation provides guaranteed log(n) time cost for the
* {@code containsKey}, {@code get}, {@code put} and {@code remove}
* operations.  Algorithms are adaptations of those in Cormen, Leiserson, and
* Rivest's <em>Introduction to Algorithms</em>.
```

.get(key)
.containsKey(key)

Read-only

.put(key, value)
.remove(key)

Unitary write
Idempotent
(not atomic, need synchronize)

.put() and .remove() slightly slower than get
: need fixup .. But still very fast

TreeMap.iterator / subMap / headMap / tailMap

```java
 * Returns the successor of the specified Entry, or null if no such.
static <K,V> TreeMap.Entry<K,V> successor(Entry<K,V> t) {
    if (t == null)
        return null;
    else if (t.right != null) {
        Entry<K,V> p = t.right;
        while (p.left != null)
            p = p.left;
        return p;
    } else {
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        while (p != null && ch == p.right) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

```java
 * Returns the predecessor of the specified Entry, or null if no such.
static <K,V> Entry<K,V> predecessor(Entry<K,V> t) {
    if (t == null)
        return null;
    else if (t.left != null) {
        Entry<K,V> p = t.left;
        while (p.right != null)
            p = p.right;
        return p;
    } else {
        Entry<K,V> p = t.parent;
        Entry<K,V> ch = t;
        while (p != null && ch == p.left) {
            ch = p;
            p = p.parent;
        }
        return p;
    }
}
```

Unitary successor() and predecessor()   run in O(log N)

… scanning all TreeMap runs in O(N log N)
not optimal compared to Arrays.sort() + scanning in ArrayList

Arrays.sort()

Arrays.sort() ... O(N log N)

Many algorithms

Buble Sort (..bad O(N^2)) / Insert Sort / Quick Sort / Merge Sort ... not used in java

ParallelSort .. => using common ForkJoinPool + Dual Pivot Quick Sort

Dual Pivot Quick Sort (called from Arrays.sort() ... on primitive types)

Tim Sort ... default in ArrayList and Arrays.sort(.. Comparator)

```java
 * Sorts the specified range of the specified array of objects according
public static <T> void sort(T[] a, int fromIndex, int toIndex,
                            Comparator<? super T> c) {
    if (c == null) {
        sort(a, fromIndex, toIndex);
    } else {
        rangeCheck(a.length, fromIndex, toIndex);
        if (LegacyMergeSort.userRequested)
            legacyMergeSort(a, fromIndex, toIndex, c);
        else
            TimSort.sort(a, fromIndex, toIndex, c, null, 0, 0);
    }
}
```

```java
 * Sorts the specified array into ascending numerical order.
public static void sort(int[] a) {
    DualPivotQuicksort.sort(a, 0, a.length - 1, null, 0, 0);
}
```

Tim Peter’s Sort

```
/**
 * A stable, adaptive, iterative mergesort that requires far fewer than
 * n lg(n) comparisons when running on partially sorted arrays, while
 * offering performance comparable to a traditional mergesort when run
 * on random arrays.  Like all proper mergesorts, this sort is stable and
 * runs O(n log n) time (worst case).  In the worst case, this sort requires
 * temporary storage space for n/2 object references; in the best case,
 * it requires only a small constant amount of space.
 *
 * This implementation was adapted from Tim Peters's list sort for
 * Python, which is described in detail here:
 *
 *   http://svn.python.org/projects/python/trunk/Objects/listsort.txt
```

PriorityQueue<T>

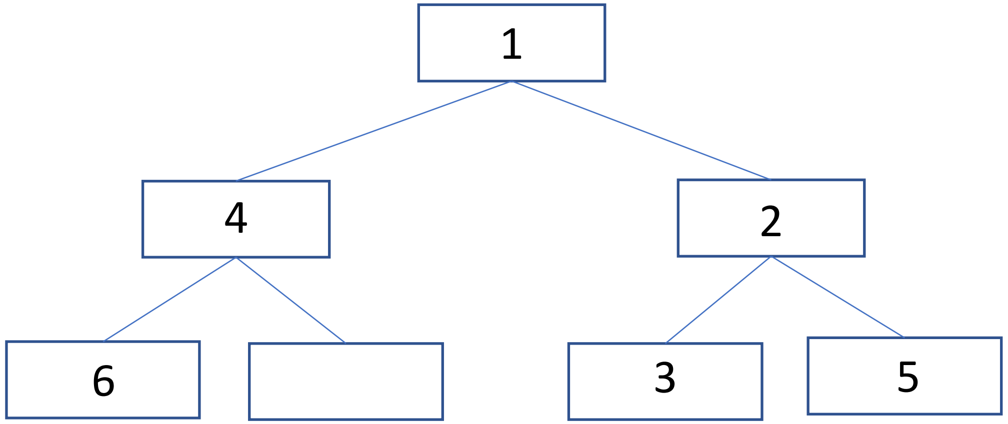
PriorityQueue Property:
for all node, parent value <= child value

Compared to TreeMap ... no constraint between left child and right child

Root contains the min value

... all others looks « unordered »

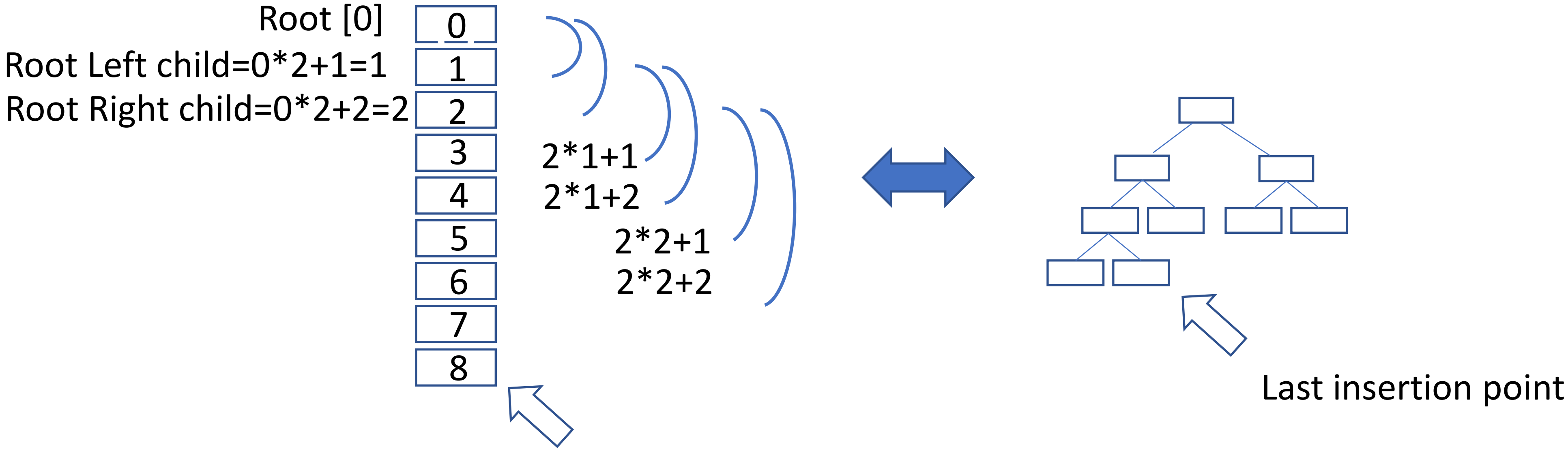
Example for
1, 2, 3, 4, 5, 6



PriorityQueue … Balanced Binary Heap as Array

```
/**
 * Priority queue represented as a balanced binary heap: the two
 * children of queue[n] are queue[2*n+1] and queue[2*(n+1)].  The
 * priority queue is ordered by comparator, or by the elements'
 * natural ordering, if comparator is null: For each node n in the
 * heap and each descendant d of n, n <= d.  The element with the
 * lowest value is in queue[0], assuming the queue is nonempty.
 */
transient Object[] queue; // non-private to simplify nested class access

 * The number of elements in the priority queue.
private int size = 0;
```

Tree as Array



Fast arithmetic base 2:

« i *2 » : i << 1

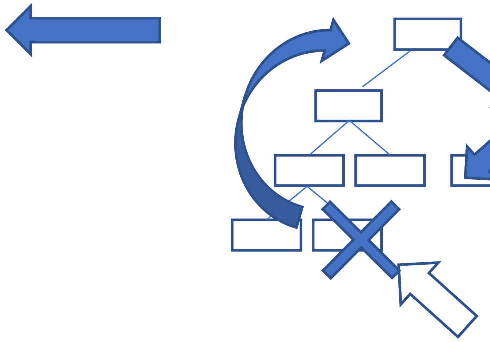
« i /2 » : i >>> 1

PriorityQueue.peek() … O(1)

peek() : return but do not consume (no remove) min value
… simply look at root !

```
public E peek() {
    return (size == 0) ? null : (E) queue[0];
}
```

PriorityQueue.poll() ... O(log N)




```java
public E poll() {
    if (size == 0)
        return null;
    int s = --size;
    modCount++;
    E result = (E) queue[0];
    E x = (E) queue[s];
    queue[s] = null;
    if (s != 0)
        siftDown(0, x);
    return result;
}
```

```java
/**
 * Inserts item x at position k, maintaining heap invariant by
 * demoting x down the tree repeatedly until it is less than or
 * equal to its children or is a leaf.
 */
private void siftDown(int k, E x) {
    if (comparator != null)
        siftDownUsingComparator(k, x);
    else
        siftDownComparable(k, x);
}
```

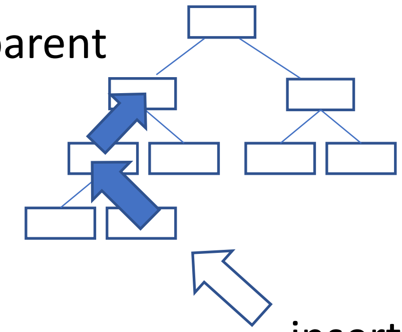
```java
private void siftDownUsingComparator(int k, E x) {
    int half = size >>> 1;
    while (k < half) {
        int child = (k << 1) + 1;
        Object c = queue[child];
        int right = child + 1;
        if (right < size &&
            comparator.compare((E) c, (E) queue[right]) > 0)
            c = queue[child = right];
        if (comparator.compare(x, (E) c) <= 0)
            break;
        queue[k] = c;
        k = child;
    }
    queue[k] = x;
}
```

PriorityQueue.add(v) / .offer(v) … O(log N)

```java
 * Inserts the specified element into this priority queue.
public boolean offer(E e) {
    if (e == null)
        throw new NullPointerException();
    modCount++;
    int i = size;
    if (i >= queue.length)
        grow(i + 1);
    size = i + 1;
    if (i == 0)
        queue[0] = e;
    else
        siftUp(i, e);
    return true;
}
```

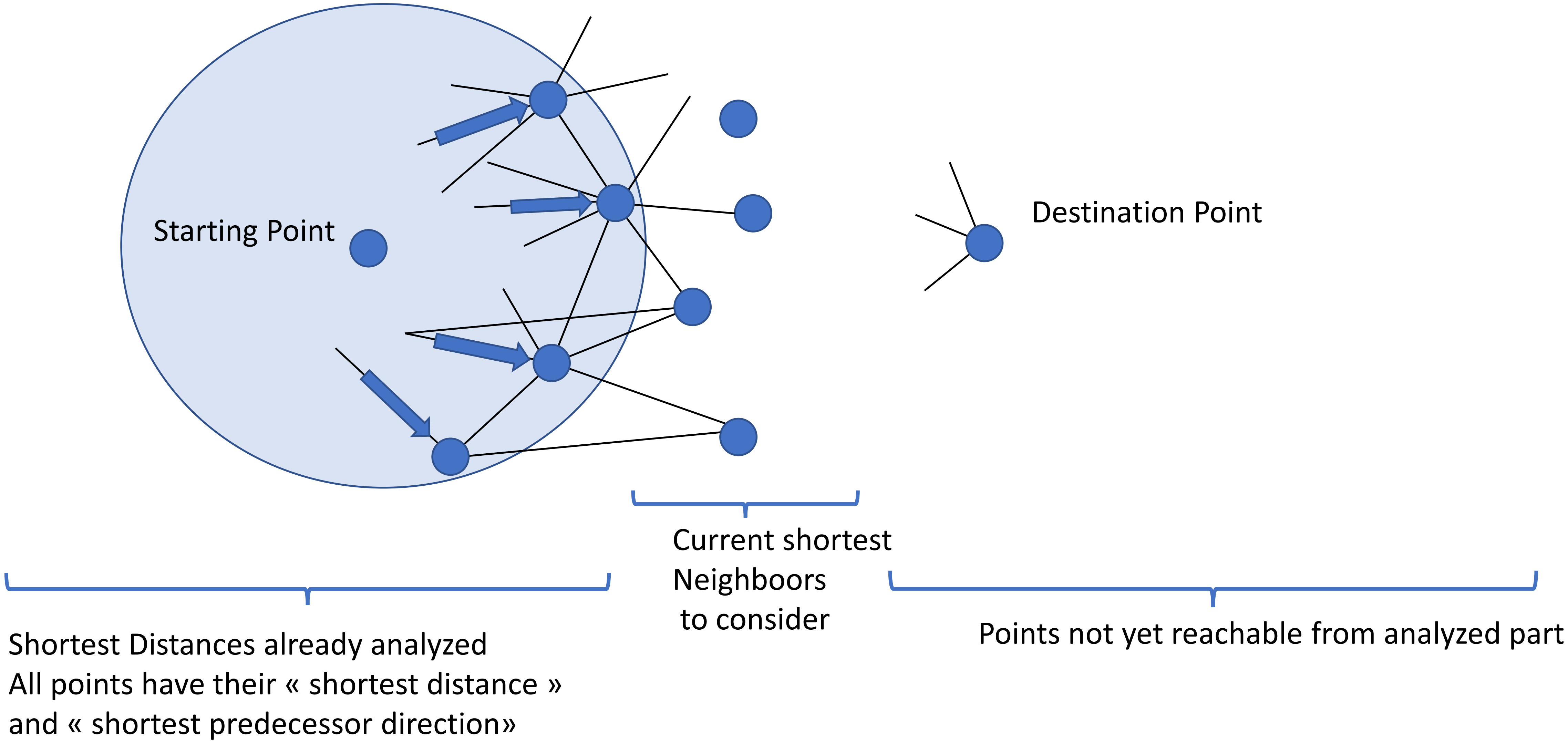
```java
private void siftUpComparable(int k, E x) {
    Comparable<? super E> key = (Comparable<? super E>) x;
    while (k > 0) {
        int parent = (k - 1) >>> 1;
        Object e = queue[parent];
        if (key.compareTo((E) e) >= 0)
            break;
        queue[k] = e;
        k = parent;
    }
    queue[k] = key;
}
```

Loop Swap parent
until lower



insert last

Typical PriorityQueue Usages
… Shortest Path Finding



Shortest Path Finding Algorithms
Bellman-Ford, Dijkstra, ..

```
procedure Shortest-Path-Faster-Algorithm(G, s)
    for each vertex v ≠ s in V(G)
        d(v) := ∞
    d(s) := 0
    push s into Q
    while Q is not empty do
        u := poll Q
        for each edge (u, v) in E(G) do
            if d(u) + w(u, v) < d(v) then
                d(v) := d(u) + w(u, v)
                if v is not in Q then
                    push v into Q
```

Q is « queue »
Works with FIFO queue
… but optimal with PriorityQueue (Dijkstra)

Libraries for Collection & Algorithms
big ecosystem, but jdk is often enough

Libraries of Interrest :

Guava (for Immutable Types)

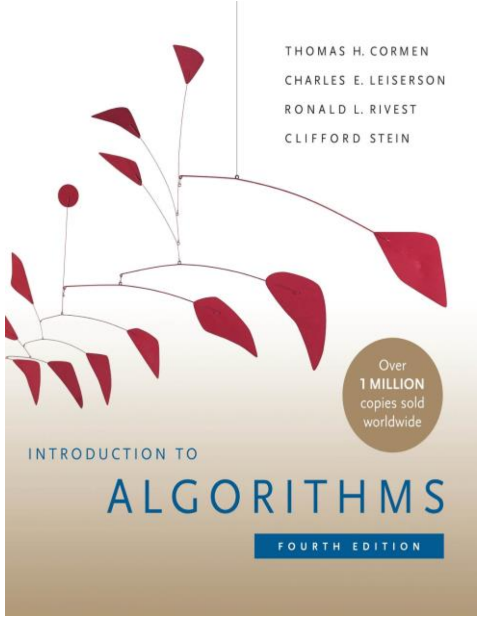
Specialized data structures for native types « ListInt » instead of « List<Integer> »

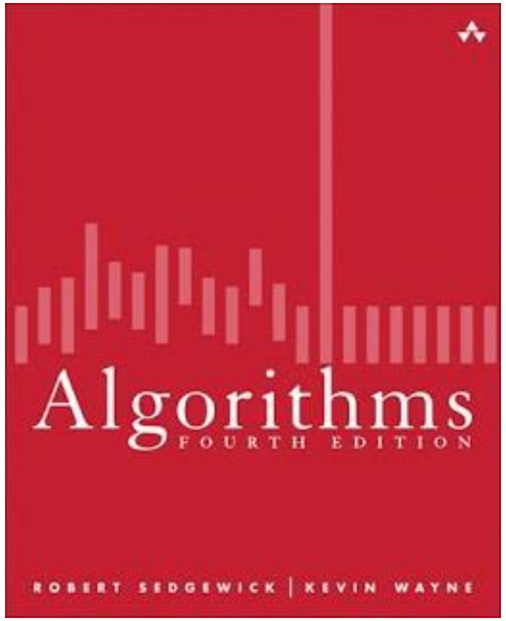
Specialized « Off-Heap Memory » collections … for huge in-memory collections,
but no GC problems with big jvm heap

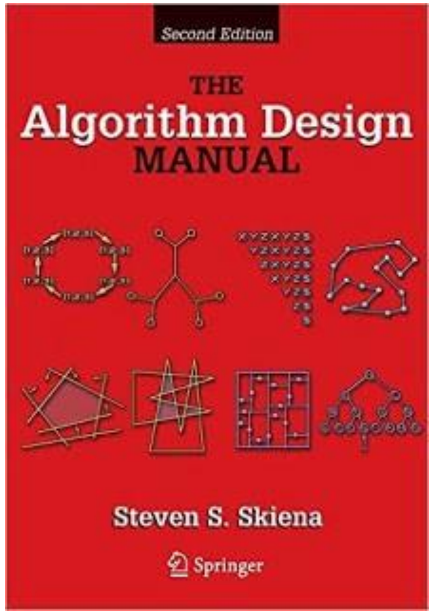
Disitributed collection … Spark Datasets

Specialized persistent collections (save to File ~Database)

More Links:
Books, Google, SourceCode, StackOverflow, …







Questions ?