

# INF102

## Algorithms, Data Structures and Programming

Marc Bezem<sup>1</sup>

<sup>1</sup>Department of Informatics  
University of Bergen

Fall 2015

## INF102, practical stuff

- ▶ Lecturer: Marc Bezem; Team: see homepage
- ▶ Homepage: [INF102](#) (hyperlinks in red)
- ▶ Also: [GitHub](#) (recommended); Dropbox: [slides](#), [schedule](#)
- ▶ Textbook: [Algorithms, 4th edition](#)
- ▶ Prerequisites: INF100 + 101 ( $\approx$  Ch. 1.1 + 1.2)
- ▶ Syllabus (pensum): Ch. 1.3 – 1.5, Ch. 2 – 4
- ▶ Exam: three compulsory exercises and a [written exam](#)
- ▶ Old exams: [2004–2013](#), [2014](#)
- ▶ [Table of Contents of these slides](#)

# Resources

- ▶ Good textbook, USA-style: many pages, exercises etc.
- ▶ Average speed must be ca 50 pages p/w
- ▶ Lectures (ca 24) focus on the essentials
- ▶ Slides (ca 120, dense!) summarize the lectures
- ▶ Prepare yourself by reading in advance
- ▶ Workshops: selected exercises
- ▶ Test yourself by trying some exercises in advance
- ▶ If you can do the exercises (incl. compulsory), you are fine
- ▶ Review of exercises on Friday morning

## Generic Bags, Queues and Stacks

- ▶ Generic programming in Java, example: **PolyPair.java**
- ▶ Bag, Queue and Stack are generic, iterable collections
- ▶ Queue and Stack: Ch. 9 in textbook INF100/1
- ▶ APIs include: `boolean isEmpty()` and `int size()`
- ▶ All three support adding an element
- ▶ Queue and Stack support removing an element (if any)
- ▶ FIFO Queue (en/dequeue), LIFO Stack (push/pop)
- ▶ Dijkstra's Two-Stack Expression Evaluation **Movie**
- ▶ Example:  $( 1 + ( ( 2 + 3 ) * ( 4 * 5 ) ) )$

## Implementations

- ▶ `ResizingArray_Stack.java`
- ▶ Arrays give direct access, but have fixed size
- ▶ Resizing takes time and space proportional to size
- ▶ `LinkedList_Stack.java`
- ▶ No fixed size, but indirect access
- ▶ Pointers take space and dereferencing takes time
- ▶ Programming with pointers: make a picture
- ▶ `LinkedList_Queue.java`

## Computation time and memory space

- ▶ Two central questions:
  - ▶ How long will my program take?
  - ▶ Will there be enough memory?
- ▶ Example: **ThreeSum.java**
- ▶ Inner loop (here  $a[i] + a[j] + a[k] == 0$ ) is important
- ▶ Sorting helps: **ThreeSumOptimized.java**
- ▶ Run some experiments: `1Kints.txt`, `2Kints.txt`, ...

## Methods of Analysis

- ▶ Empirical:
  - ▶ Run program with randomized inputs, measuring time & space
  - ▶ Run program repeatedly, doubling the input size
  - ▶ Measuring time: **StopWatch**
  - ▶ Plot, or log-log plot and **linear regression**
- ▶ Theoretical:
  - ▶ Define a cost model by abstraction (e.g., array accesses, comparisons, operations)
  - ▶ Try to count/estimate/average this cost as function of the input (size)
  - ▶ Use  $O(f(n))$  and  $f(n) \sim g(n)$

## ThreeSum, empirically

- ▶ Input sizes 1K, 2K, 4K, 8K take time 0.1, 0.8, 6.4 ,51.1 sec
- ▶ The log's are 3, 3.3, 3.6, 3.9 and -1, -0.1, 0.8, 1.71
- ▶ Basis of the logarithm should be the same for both
- ▶ Linear regression gives  $y \approx 3x - 10$
- ▶  $\log(f(n)) = 3 \log(n) - 10$  iff

$$f(n) = 10^{\log(f(n))} = 10^{3 \log(n) - 10} = n^3 * 10^{-10}$$

- ▶ Conclusion: cubic in the input size, with constant  $\approx 10^{-10}$
- ▶ Strong dependence on input can be a problem
- ▶ Constant  $10^{-10}$  depends on computer, exponent 3 does not



## ThreeSum, theoretically

- ▶ Number of different picks of triples:  $g(n) = n(n-1)(n-2)/6$
- ▶ Inner loop  $a[i] + a[j] + a[k] == 0$  executed  $g(n)$  times
- ▶  $g(n) = n^3/6 - n^2/2 + n/3$
- ▶ Cubic term  $n^3/6$  wins for large  $n$
- ▶ Computational model # array accesses:  $3 * n^3/6 = n^3/2$
- ▶ Cost array access  $t$  sec: time  $t * n^3/2$  sec
- ▶ Cost models are abstractions! (NB cache)

## Big Oh, and $\sim$

- ▶ Q: 'wins for large  $n$ ' uhh???
- ▶ A: Big Oh, and  $\sim$  will clear this up
- ▶ Costs are positive quantities, so  $f, g, \dots : \mathbb{N} \rightarrow \mathbb{R}^+$
- ▶ MNF130:  $f(n)$  is  $O(g(n))$  if there exist  $c \in \mathbb{R}^+$ ,  $N \in \mathbb{N}$  such that  $f(n) \leq cg(n)$  for all  $n \geq N$  (that is,, for  $n$  large enough)
- ▶ Example:  $n^2$  and even  $99n^3$  are  $O(n^3)$ , but  $n^3$  is not  $O(n^{2.9})$
- ▶ INF102:  $f(n) \sim g(n)$  if  $1 = \lim f(n)/g(n)$
- ▶ If  $f(n) \sim g(n)$ , then  $f(n)$  is  $O(g(n))$  and  $g(n)$  is  $O(f(n))$
- ▶ Big Oh and  $\sim$  aim to capture 'order of growth'
- ▶ Big Oh abstracts from constant factors,  $\sim$  does not
- ▶ Large constant factors are important!

## Important orders of growth

- ▶ constant:  $c$ ,  $f(n) = c$  for all  $n$
- ▶ linear:  $n$  (compare all for  $n = 20$  sec)
- ▶ linearithmetic:  $n \log n$
- ▶ quadratic:  $n^2$
- ▶ cubic:  $n^3$
- ▶ exponential:  $2^n$
- ▶ general form:  $an^b(\log n)^c$

## Logarithms and Exponents

- ▶ Definition:  $\log_x z = y$  iff  $x^y = z$  for  $x > 0$
- ▶ Inverses:  $x^{\log_x y} = y$  and  $\log_x x^y = y$
- ▶ Exponent:  $x^{(y+z)} = x^y x^z$ ,  $x^{(yz)} = (x^y)^z$
- ▶ Logarithm:  $\log_x(yz) = \log_x y + \log_x z$ ,  $\log_x z = \log_x y \log_y z$
- ▶ Base of logarithm: the  $x$  in  $\log_x$
- ▶ Various bases:  $\log_2 = \lg$ ,  $\log_e = \ln$ ,  $\log_{10} = \log$
- ▶ Double exponent: e.g.  $2^{(2^n)}$  (not used in INF102)
- ▶ Double logarithm:  $\log(\log n)$  (not used in INF102)

## Worst case, average case, amortized cost

- ▶ Worst case: guaranteed, independent of input; Examples:
  - ▶ Linked list implementations of Stack, Queue and Bag: all operations take constant time in the worst case
  - ▶ Resizing array implementations of Stack, Queue and Bag: adding and deleting take linear time in the worst case (easy)
- ▶ Average case: not guaranteed, dependent of input *distribution*
- ▶ Amortized: worst-case cost *per operation*. E.g., each 10-th operation has cost  $\leq 21$ , all others cost 1, amortized  $\leq 3$  p/o.
- ▶ Resizing arrays: adding and deleting take constant time *per operation* in the worst case (proof is difficult)
- ▶ Special case of resizing array that is only growing:  
 $1(2)2(4)3(4)4(8)5(6)6(8)7(8)8(16)9 \dots 16(32) \dots$ , with  $(n)$  the new size.  
 Resizing to  $(n)$  costs  $2n$  array accesses, so in total  
 $(1+4)+(1+8)+(2+16)+(4+32)+(8+64) \dots$ , so 9 p/push.

## Staying Connected

- ▶ We want efficient algorithms and datastructures for testing whether two objects are 'connected'
- ▶ MNF130: relation  $E \subseteq V \times V$  is an *equivalence* if
  - ▶  $E$  is *reflexive*:  $\forall x \in V. E(x, x)$
  - ▶  $E$  is *symmetric*:  $\forall x, y \in V. E(x, y) \rightarrow E(y, x)$
  - ▶  $E$  is *transitive*:  $\forall x, y, z \in V. E(x, y) \wedge E(y, z) \rightarrow E(x, z)$
- ▶ We assume connectedness to be an equivalence
- ▶ Dynamic connectivity means (here) that  $E$  can grow
- ▶ Clear relationship with paths in graphs, (connected) components (MNF130)
- ▶ Input:  $N$  and pairs in  $V = \{0, \dots, N-1\}$  defining  $E$
- ▶ Challenge: efficient `boolean connected(int p, int q)`
- ▶ Example:  $N = 10$ , 4 3, 3 8, ... (`algs4-data/tinyUG.txt`)
- ▶ Picture on blackboard (don't print pairs that are already connected)

## Union-Find

- ▶ Find, idea: every component has one element as its identifier, `int find(int n)` computes this identifier
- ▶ Union, idea: for any new pair  $n\ m$  that are not already connected, `union(int n, int m)` takes the union of the two components, ensuring `find(n) == find(m)`
- ▶ API: **UF**; Cost model: number of array accesses
- ▶ Implementations:
  - ▶ **SlowUF.java**: `id[p]` identifier of  $p$   
`find()`  $\sim 1$ , `union()`  $\sim$  between  $n+3$  and  $2n+1$
  - ▶ **FastUF.java**: `int[] id` pointers, `id[p]==p`: identifier  
`find()`  $\sim 1+2d$ , `union()`  $\sim 1 + \text{two find}()$ 's
  - ▶ **WeightedUF.java**: `int[] id` pointers, `int[] sz` subtree sizes  
`find()` and `union()` both  $\sim \lg n$
- ▶ WeightedUF: height of subtree of size  $k$  is at most  $\lg k$
- ▶ Path-compression: ultimate improvement of UF (almost  $O(1)$ , amortized)

# Sorting

- ▶ Sorting: putting objects in a certain order
- ▶ MNF130: relation  $R \subseteq V \times V$  is a *total order(ing)* if
  1.  $R$  is *reflexive*:  $\forall x \in V. R(x, x)$
  2.  $R$  is *transitive*:  $\forall x, y, z \in V. R(x, y) \wedge R(y, z) \rightarrow R(x, z)$
  3.  $R$  is *antisymmetric*:  $\forall x, y \in V. R(x, y) \wedge R(y, x) \rightarrow x = y$
  4.  $R$  is *total*:  $\forall x, y \in V. R(x, y) \vee R(y, x)$
- ▶ Natural orderings:
  - ▶ Numbers of any type: ordinary  $\leq$  and  $\geq$
  - ▶ Strings: lexicographic
  - ▶ Objects of a Comparable type: `v.compareTo(w) <= 0`



## Sorting (ctnd)

- ▶ Bubble sort: `ExampleSort.java`
- ▶ Certification: `assert isSorted(a)` in `main()`
- ▶ No guarantee against modifying the array (but `exch()` is safe)
- ▶ Costmodel 1: number of `exch()`'s and `less()`'s
- ▶ Costmodel 2: number of array accesses
- ▶ Pitfalls: cache misses, expensive `v.compareTo(w) < 0`
- ▶ Why studying sorting? (`java.util.Arrays.sort()`)
- ▶ Comparing sorting algorithms: `SortCompare.java`

## Selection Sort

- ▶ Bubble sort:  $\sim n^2/2$  compares,  $0 \leq \text{exchanges} \leq \sim n^2/2$
- ▶ Selection sort:
  - ▶ Find index of a minimum in  $a[0..n-1]$ , exchange with  $a[0]$
  - ▶ Find index of a minimum in  $a[1..n-1]$ , exchange with  $a[1]$
  - ▶ ... until  $n-2$
- ▶ Selection sort:  $\sim n^2/2$  compares,  $0 \leq \text{exchanges} \leq n-1$  (!)

```
public static void sort(Comparable[] a) {  
    int N = a.length;  
    for (int i=0; i<N-1; i++){  
        int min=i;  
        for (int j=i+1; j<N; j++) if (less(a[j],a[min])) min=j;  
        if (i != min) exch(a,i,min);  
    }  
}
```

## Insertion sort

- ▶ Insertion sort:
  - ▶ Insert  $a[1]$  on its correct place in (sorted)  $a[0..0]$
  - ▶ Insert  $a[2]$  on its correct place in (sorted)  $a[0..1]$
  - ▶ ... until  $a[n-1]$
- ▶ Very good for partially sorted arrays, costs:
  - ▶ Best case:  $n-1$  compares and 0 exchanges
  - ▶ Worst case:  $\sim n^2/2$  compares and exchanges
  - ▶ Average case:  $\sim n^2/4$  compares and exchanges (distinct keys)

```
public static void sort(Comparable[] a) {  
    int N = a.length;  
    for (int i=1; i<N; i++){  
        for (int j=i; j>0 && less(a[j],a[j-1]); j--)  
            exch(a,j,j-1);  
    }  
}
```

## Shell sort

- ▶ Insertion sort:
  - ▶ Very good for partially sorted arrays
  - ▶ Slow in transport: step by step `exch(a,j,j-1)`
- ▶ Idea: h-sort, `a[i], a[i+h], a[i+2h], ...` sorted (any `i`)

```
public static void hsort(int h, Comparable[] a) {  
    int N = a.length;  
    for (int i=h; i<N; i++)  
        for (int j=i; j-h>=0 && less(a[j],a[j-h]); j-=h)  
            exch(a,j,j-h);  
}
```

- ▶ Insertion sort: `hsort(1,a)`
- ▶ Shell sort: e.g., `hsort(10,a); hsort(1,a)`

## Shell sort (ctnd)

- ▶ `hsort(10,a); hsort(1,a)` faster than just `hsort(1,a)` !
- ▶ Q: How is this possible?
- ▶ A: `hsort(10,a)` transports items in steps of 10, which would be done by `hsort(1,a)` in 10 steps of 1.
- ▶ What about `hsort(100,a); hsort(10,a); hsort(1,a)`?
- ▶ To be expected: depends on the length  $N$  of the array
- ▶ Best practice:  $h = N/3, N/9, \dots, 364, 121, 40, 13, 4, 1$

# Mergesort

- ▶ Top-down (recursive) algorithm:
  - ▶ Mergesort left half, mergesort right half
  - ▶ Merge the results
- ▶ Using an auxiliary array: [TopDownMergeSort.java](#), [Movie](#)
- ▶ Bottom-up algorithm (16 elements):
  - ▶ Merge  $a[0], a[1]$ , so  $a[2], a[3]$ , so  $a[4], a[5]$ , so ...
  - ▶ Merge  $a[0..1], a[2..3]$ , so  $a[4..5], a[6..7]$ , so ...
  - ▶ Merge  $a[0..3], a[4..7]$ , so  $a[8..11], a[12..15]$
  - ▶ Merge  $a[0..7], a[8..15]$ , done!
- ▶ Also using an auxiliary array: [BottomUpMergeSort.java](#)

## Run-time and memory use of mergesort

- ▶ Mergesort uses between  $\sim (N/2) \lg N$  and  $\sim N \lg N$  compares. Proof on bb. Important formula ( $N = 2^n$ ):

$$2C(2^{n-1}) + 2^{n-1} \leq C(2^n) \leq 2C(2^{n-1}) + 2^n$$

- ▶ Mergesort uses at most  $\sim 6N \lg N$  array accesses
- ▶ Mergesort uses  $\sim 2N$  space (plus some var's)
- ▶ Q: How fast can compare-based sorting of  $N$  distinct keys be?
- ▶ A:  $\lg N! \sim N \lg N$ ; Proof in book and on bb. Keywords: binary *compare tree*, inner nodes for each `compare(a[i], a[j])`, permutations in the leaves,  
 $N! = \text{number of permutations} \leq \text{number of leaves} \leq 2^{\text{height of tree}}$

# Quicksort

- ▶ Top-down (recursive) algorithm:
  - ▶ Choose a (pivot) value  $v$  in the array
  - ▶ Partition the array in non-empty parts  $\leq v$  and  $\geq v$
  - ▶ Quicksort the two parts
- ▶ Pros: in-place, average computation time  $O(n \log n)$
- ▶ Cons: stack space for recursion, worst-case  $O(n^2)$ , not stable
- ▶ Implementation: **QuickSort.java**
- ▶ BTW: **Bug in java.util.Arrays.sort**



## Quicksort, details

- ▶ Subtleties in `sort()`: shuffling protects against worst-case behaviour
- ▶ Termination of recursive `quicksort()`
- ▶ Subtleties in `partition()`:
  - ▶ Invariants  $l \leq h$  in the two inner loops
  - ▶ Postcondition after the two inner loops
  - ▶ Invariant of the `for(;;)` loop
  - ▶ Termination of the `for(;;)` loop
  - ▶ There are some variations that are also correct

## Run-time and memory use of quicksort

- ▶ Compare Quicksort to other sorts ( $n = 10^2, 10^3, \dots$ )
- ▶ Quicksort: time  $O(n^2)$  if pivot is always smallest (or largest)
- ▶ Randomization: choose pivot randomly, or shuffle array
- ▶ If all keys are distinct and randomization is perfect, then quicksort uses on average  $\sim 2n \ln n$  compares and  $\sim (n/3) \ln n$  exchanges (proofs in book, complicated)
- ▶ Relevant improvements:
  - ▶ Cut-off to insertion sort for sizes  $\leq 15$  (ca.)
  - ▶ Median-of-three pivot
  - ▶ Taking advantage of duplicate keys (3-way partitioning)
- ▶ Quicksort is generally quite good
- ▶ In special situations other sorts are better (e.g., countsort)

## Priority Queues

- ▶ Assume collecting and processing items having keys
- ▶ Examples of keys: time-stamp, price-tag, priority-tag
- ▶ Assume: keys can be ordered
- ▶ Reasonable: processing currently highest (or lowest)
- ▶ Special cases: items time-stamped when added
  - ▶ Queue: dequeue currently oldest (lowest time-stamp)
  - ▶ Stack: pop currently newest (highest time-stamp)
- ▶ Priority queue generalizes this
- ▶ Examples: highest priority, largest transaction, lowest price
- ▶ Abstract from 'item' and use only 'key' (in applications: use objects with fields `item` and `key` and compare on `key`)

## Priority Queues

- ▶ Good info: [Wikipedia](#); API (the bare essentials):

```
public class  ArrayListPQ<Key extends Comparable<Key>>

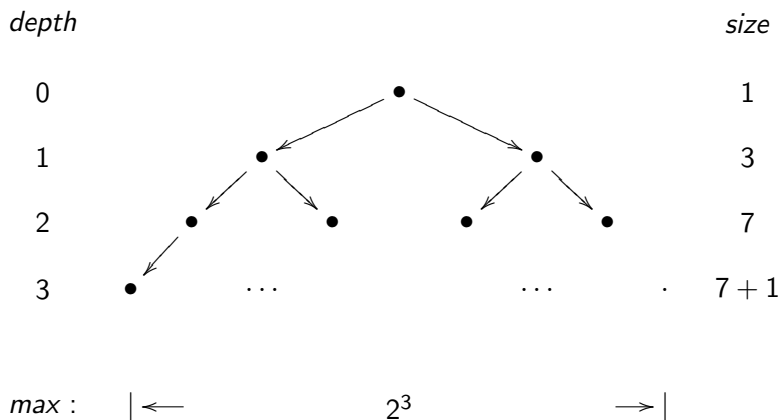
void          insert(Key v) // insert a key
Key           delMax() // delete a largest key, if any
boolean       isEmpty()
int           size()
```

- ▶ Aim: operations in logarithmic time, no extra space
- ▶ In case of duplicate keys: 'a' largest, not 'the'
- ▶ Typical application: the 1K largest keys of 1G unsorted keys
- ▶ Client: [BottomM.java](#) (Q: why is the output slowing down?)

## Binary Trees

- ▶ MNF130: Tree *size* is number of nodes, *depth* of a node is number of links to the root, tree *height* is maximum depth.
- ▶ MNF130: A binary tree is *complete* if all levels are filled. So, a complete binary tree of height  $h$  has  $2^{h+1}-1$  nodes.
- ▶ INF102: A binary tree is (left-) *complete* if all levels  $< h$  are filled, level  $h$  may be partly empty from the right (picture bb). A (left-)complete binary tree of height  $h$  has between  $2^h$  and  $2^{h+1}-1$  nodes (from now on we leave out '(left-)').
- ▶ A complete binary tree of  $n$  nodes has height  $\lfloor \lg n \rfloor$

# Picture



## Heap-ordered Binary Trees

- ▶ A binary tree is *heap-ordered* if the key in each node is  $\geq$  the keys in its children (if any). Thus the root has a maximal key.
- ▶ Array representation of heap-ordered complete binary tree (bb)
- ▶ Methods `swim()` and `sink()`: picture on bb, code below
- ▶ Implementation: `ArrayListPQ.java`

## Run-time and memory use of heaps, applications

- ▶ In a heap of  $n$  elements (since height is  $\leq \lfloor \lg n \rfloor$ ):
  - ▶ `swim()`, and hence `insert()`, takes  $\leq 1 + \lfloor \lg n \rfloor$  compares and  $\leq \lfloor \lg n \rfloor$  exchanges
  - ▶ `sink()`, and hence `delMax()`, takes  $\leq 2\lfloor \lg n \rfloor$  compares
  - ▶ `sink()` takes  $\leq \lfloor \lg n \rfloor$  exchanges, and `delMax()`  $\leq 1 + \lfloor \lg n \rfloor$
- ▶ Heap construction by `insert()` can sometimes be improved
- ▶ Given an array of keys, right-to-left heap construction (bb) takes  $< 2n$  compares and  $< n$  exchanges
- ▶ Applications: **heapsort** and merging sorted streams (bb)
- ▶ Many variations with extended API (indexed priority queue)



## Purpose of Sorting

- ▶ Sorting makes the following easier and more efficient:
  - ▶ Searching (binary search, example: `ThreeSumOptimized`)
  - ▶ Searching and looking up, e.g., the `pagenumber` in an index
  - ▶ Removing duplicates
  - ▶ Finding the median, quartiles etc.
- ▶ Our sorting algorithms are generic: `sort(Comparable[] a)`, for any user-defined data type with a `compareTo()` method
- ▶ We do *pointer sorting*, manipulating refs to objects.
  - ▶ Pro: not moving full objects
  - ▶ Cons: pointer dereferencing, no `sort(int[] a)`
- ▶ More flexibility: pass a `Comparator` object to `sort()`

## Comparator object

- ▶ API: `void sort(Object[] a, Comparator c)`
- ▶ Call, e.g.: `sort(a, new Transaction.WhenOrder())`
- ▶ Call, e.g.: `sort(a, new Transaction.SizeOrder())`
- ▶ Obs: `import java.util.Comparator`
- ▶ Obs: `less(Object o1, Object o2, Comparator c)`
- ▶ Priority queues also with `Comparator`

```
public class Transaction {  
    ...  
    public static class MyOrder {  
        implements Comparator<Transaction>  
        public int compare(Transaction t, Transaction v){...}  
    } // End of Myorder  
    ...// similarly: WhenOrder, SizeOrder  
} // End of Transaction
```

## Applications of Sorting

- ▶ Consider sorting first to make other problems easier
- ▶ Commercial computing (sort on price, departure time, ...)
- ▶ Search for information: web-indexing, search engines
- ▶ Job scheduling heuristic: longest processing time first
- ▶ To come: Prim's, Dijkstra's and Kruskal's algorithms
- ▶ Huffman compression: a lossless compression based on using the shortest codes for the symbols that occur often.  
Frequency counter: next chapter!
- ▶ Cryptology and genomics (e.g., longest repeated substring)

# Symbol Tables

- ▶ Symbol table associates *keys* with *values*: *key-value pairs*
- ▶ Examples: keyword-page number, ID number-personal data
- ▶ Important operations:
  - ▶ Insert a key-value pair in the symbol table: `void put(k,v)`
  - ▶ Search the value for a given key (if any): `Value get(k)`
- ▶ Important conventions:
  - ▶ Inserting key-value for existing key: overwriting the value
  - ▶ No duplicate keys, no null keys
  - ▶ Value null: no value for this key
  - ▶ Lazy deletion: insert key-null; Eager: really delete key
- ▶ **API** of unordered symbol table
- ▶ Aim: all operations in time  $\sim c \lg n$  with constant  $c$  small

# ST Basics

- ▶ Archetypical ST-client: frequency counter (code: `main`)
- ▶ Cost model: number of compares
- ▶ Naive ST: unordered linked list, linear search (INF101, Ch.9)
  - ▶ Search miss:  $\sim n$  compares
  - ▶ Search hit: between 1 and  $\sim n$  compares
  - ▶ Random search hit:  $(1 + \dots + n)/n \sim n/2$  compares
  - ▶ Inserting  $n$  distinct keys:  $(1 + \dots + (n-1)) \sim n^2/2$  compares
- ▶ `algs4-data/leipzig1M.txt`: 21M words, 500K distinct
- ▶ Naive ST impracticable for genomics, internet
- ▶ Scale: G-T keys, M-G distinct (Kilo,Mega,Giga,Tera)
- ▶ Better for unordered ST: hashing (in Ch. 3.4)

## Ordered Symbol Table

- ▶ Ordered ST: keys are ordered
- ▶ **API** of ordered symbol table
- ▶ Binary search: `get(Key k)` takes  $\sim \lg n$  comparisons
- ▶ What about `put(Key k, Value v)`? **ArrayListST**, good!
- ▶ TODO: test that `add(int i, E e)` is amortized  $O(1)$
- ▶ Implementation with binary search in **ArrayListST.java**
- ▶ Trace of inserts on bb: S E A R C H E X A M P L E
- ▶ Experiments with `tinyTale.txt`, `tale.txt`, ...

## Binary Search Trees

- ▶ Binary *search* tree: for every node, all keys to the left of this node are smaller, and all keys to the right are larger
- ▶ Search time: length of the path to the node where the key 'should' be
- ▶ Balanced binary tree with  $n$  keys has  $\lg n$  height
- ▶ Unbalanced binary trees can have height  $n$  (max depth)
- ▶ Search hits in a binary search tree, built without rebalancing, of  $n$  random keys take on average  $\sim 2 \ln n$  compares
- ▶ **UBST.java**: `put()`, `get()`, `size()`, `isEmpty()`
- ▶ Trace of inserts on bb: S E A R C H E X A M P L E

## Binary Search Trees (ctnd)

- ▶ Interrelated, increasing difficulty: `min(Node x)`, `deleteMin(Node x)`, `delete(Node x, Key k)`
- ▶ Node of minimum key: not `null`, and has left child `null`, and is root or left child of parent (picture on bb)

```
public Node min(Node x){ // subtree under x
    while (x!=null) x = x.left;
    return x;
} // cf. tail recursive min() in Alg. 3.3
```

- ▶ Delete minimum key, two cases:  
(1) both children `null`; (2) left child `null`
- ▶ Delete is really difficult: bb + **BST.java**
- ▶ Don't forget: update `x.N` along the path to the root!



## Balanced Search Trees: keep paths short!

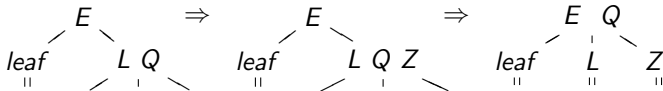
- ▶ NB tree balancing not as easy as in UF and Heap (4hrs!)
- ▶ A 2-3 search tree consists of 2-nodes and 3-nodes:
  - ▶ Each 2-node has two children and a key  $k$  such that all keys in the left subtree are  $< k$ , and all keys in the right subtree  $> k$
  - ▶ Each 3-node has three children and two keys  $k_1, k_2$  such that all keys in the left subtree are  $< k_1$ , all keys in the middle subtree  $> k_1$  and  $< k_2$ , and all keys in the right subtree  $> k_2$
- ▶ Examples and pictures on bb
- ▶ *Perfect* 2-3 search tree: paths from root to leaves equally long
- ▶ Search: compare key with key(s) in node, if equal return corresponding value, else search in one of left, middle, right subtree where the key should be (if it occurs at all)
- ▶ Insert should keep tree perfect, rough idea:
  - ▶ into a 2-leaf: make it into a 3-leaf
  - ▶ into a 3-node: do something clever (explained below)

## Insert in Balanced Search Trees

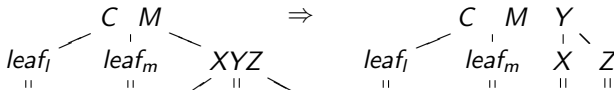
- ▶ Terminology: a *leaf* is a node all whose children are null
- ▶ Data invariant 1: 2-3 search tree
- ▶ Data invariant 2: paths from root to leaves equally long

- ▶ Insert into a 2-leaf  $L$  either  $\begin{array}{c} A \\ / \quad | \quad \backslash \\ L \end{array}$  or  $\begin{array}{c} L \\ / \quad | \quad \backslash \\ Z \end{array}$

- ▶ into a 3-leaf whose parent is a 2-node: with new key  $Z$



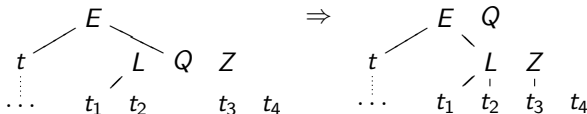
- ▶ into a 3-leaf whose parent is a 3-node: with new key  $Z$



- ▶ into a 3-node whose parent is a 3-node: move up middle key!

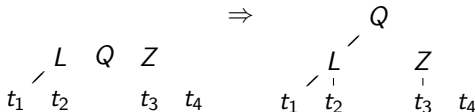
## Insert (ctnd)

- ▶ Data invariant 1: 2-3 search tree
- ▶ Data invariant 2: paths from root to leaves equally long
- ▶ Insert works up from the leaf where the key 'should' be
  - ▶ if 2-node on path to root: make it into a 3-node (two cases)



Equivalent

- ▶ otherwise: split the root

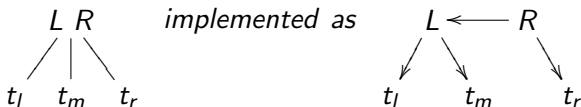


## Insert, summary and examples

- ▶ Six operations for eliminating 4-nodes: Equivalent
  - ▶ if parent is root: root split
  - ▶ if parent is 2-node: move middle key up (left and right case)
  - ▶ if parent is 3-node: move middle key up (left, middle, right)
- ▶ Search and insert visit at most  $\lfloor \lg n \rfloor$  nodes
- ▶ Proof: maximal path length is  $\geq \lfloor \log_3 n \rfloor$  and  $\leq \lfloor \log_2 n \rfloor$
- ▶ Trace of inserts on bb: S E A R C H (E) X (A) M P L (E)
- ▶ Trace of inserts on bb: A C E H L M P R S X

## Red-black trees

- ▶ Red-black trees implement 2-3 trees
- ▶ Idea: one 3-node = two 2-nodes + extra info
- ▶ Extra info coded in color, picture:



- ▶ A *red-black tree* is a binary search tree with red and black links such that:
  - ▶ Only left links can be red (but need not be)
  - ▶ Never  $\leftarrow \leftarrow$
  - ▶ Perfect black balance (all paths from root to leaves same number of black links; this number is called the *black height*)
- ▶ Equivalent: red-black tree and perfect 2-3 search tree


## Red-black trees (ctnd)

- Color is attribute of *incoming* link (why?)

```
private class Node {  
    Key key;  
    Value value;  
    Node left, right;  
    boolean color; // true for red, false for black  
    int N;  
}  
private boolean isRed(Node n) {  
    if (n==null) {return false;} else {return x.color}
```

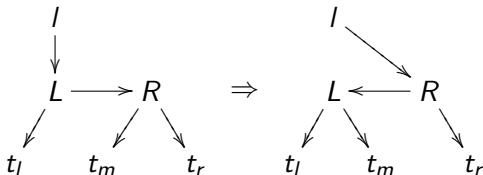
## Rotating and Color Flipping

- ▶ Aim: restoring the data invariants of red-black search trees
  1. Only left red links, but never two
  2. Search tree invariant
  3. Perfect black balance
- ▶ Invariants get violated by temporary 4-nodes, e.g.,
  - ▶ inserting  $Z$  in  $L \leftarrow R : L \leftarrow R \rightarrow Z$
  - ▶ inserting  $A$  in  $L \leftarrow R : A \leftarrow L \leftarrow R$
  - ▶ inserting  $M$  in  $L \longleftarrow R : L \longleftarrow R$ 


- ▶ Restoring:
  - ▶ Color flip  $L \leftarrow R \rightarrow Z : L \leftarrow R \rightarrow Z$
  - ▶ Rotation right + color flip  $A \leftarrow L \leftarrow R : A \leftarrow L \rightarrow R$
  - ▶ Rotation left into  $M \leftarrow L \leftarrow R$ , then as previous

## Left Rotation

Call: `l = rotateLeft(l);`

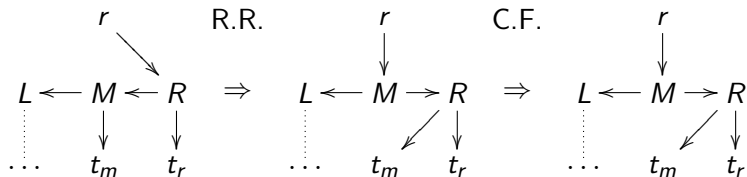


```
private Node rotateLeft(Node l){
    Node r = l.right; l.right = r.left; r.left = l key;
    r.color = l.color; l.color = true // == RED
    r.N = l.N; l.N += size(l.right); // Why?
    return r;
}
```



## Right Rotation and Color Flip

Typically in the following situation (e.g., after insert(L) in a 3-leaf):



- ▶ Code of `rotateRight()` like that of `rotateLeft()`
- ▶ NB1: operations are local (here only  $r$ ,  $M$ ,  $R$ )
- ▶ NB2: operations preserve data invariants
- ▶ NB3: root is a special case (always black)
- ▶ Deletions: complicated, but doable (Exc. 3.3.39–41)

## Run-time and memory use of Red-Black BSTs

- ▶ The height of a red-black BST with  $n$  nodes is  $\leq 2 \lg n$   
Proof: the worst-case is one 3-node path and the rest 2-nodes
- ▶ The average length of path from the root to a *node* (?) in a red-black BST with  $n$  nodes is  $\lg n$  ('empirical fact')
- ▶ In a red-black BST, search, insertion, ..., and delete, take logarithmic time in the worst-case. Proof: a constant amount of work is done per visited node.
- ▶ For red-black BSTs, logarithmic time is guaranteed!

# Hashing

- ▶ A *hash function* maps keys to a (as unique as possible) array index
- ▶ A (hash) *collision* occurs when different keys are mapped to the same index
- ▶ In such a case we need *collision resolution*
- ▶ Hashing = hash function + collision resolution
- ▶ Symbol tables based on hashing: fast, but no support for `max, ...`

## TODOs Chapter 1–3

- ▶ Find out if `add(int i, E e in ArrayList` is amortized  $O(1)$
- ▶ Explain *indexed* priority queues
- ▶ Explain `delete(Key k` in red-black trees

## ToC and topics of general interest

- ▶ Table of Contents on next slide (all items clickable)
- ▶ Practical stuff: slide 2

Introduction

Ch.1.3 Bags, Queues and Stacks

Ch.1.4 Analysis of Algorithms

Ch.1.5 Case Study: Union-Find

Ch.2.1 Elementary Sorts

Ch.2.2 Mergesort

Ch.2.3 Quicksort

Ch.2.4 Priority Queues

Ch.2.5 Applications

Ch.3.1 Symbol Tables

Ch.3.2 Binary Search Trees

Ch.3.3 Balanced Search Trees

Ch.3.4 Hash Tables

Ch.3.5 Applications

Ch.4.1 Undirected Graphs

Ch.4.2 Directed Graphs

Ch.4.3 Minimum Spanning  
Trees

Ch.4.4 Shortest Paths

Table of Contents