Algorithms and Data Structures, Fall 2011

Priority queues

Rasmus Pagh

Based on slides by Kevin Wayne, Princeton

Algorithms, 4th Edition · Robert Sedgewick and Kevin Wayne · Copyright © 2002–2011

Priority queue

Collections. Insert and delete items. Which item to delete?

Stack. Remove the item most recently added.

Queue. Remove the item least recently added.

Symbol table. Remove any desired item.

Priority queue. Remove the largest (or smallest) item.

operation	*argument*	*return value*
insert	P	
insert	Q	
insert	E	
remove max		Q
insert	X	
insert	A	
insert	M	
remove max		X
insert	P	
insert	L	
insert	E	
remove max		P

Priority queue API

Requirement. Generic items are `Comparable`.

`public class MaxPQ<Key extends Comparable<Key>>`	
`MaxPQ()`	*create a priority queue*
`MaxPQ(maxN)`	*create a priority queue of initial capacity maxN*
`void insert(Key v)`	*insert a key into the priority queue*
`Key max()`	*return the largest key*
`Key delMax()`	*return and remove the largest key*
`boolean isEmpty()`	*is the priority queue empty?*
`int size()`	*number of entries in the priority queue*

API for a generic priority queue

Priority queue applications

- Event-driven simulation. [customers in a line, colliding particles]
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Computational number theory. [sum of powers]
- Artificial intelligence. [A* search]
- Statistics. [maintain largest M values in a sequence]
- Operating systems. [load balancing, interrupt handling]
- Discrete optimization. [bin packing, scheduling]
- Spam filtering. [Bayesian spam filter]

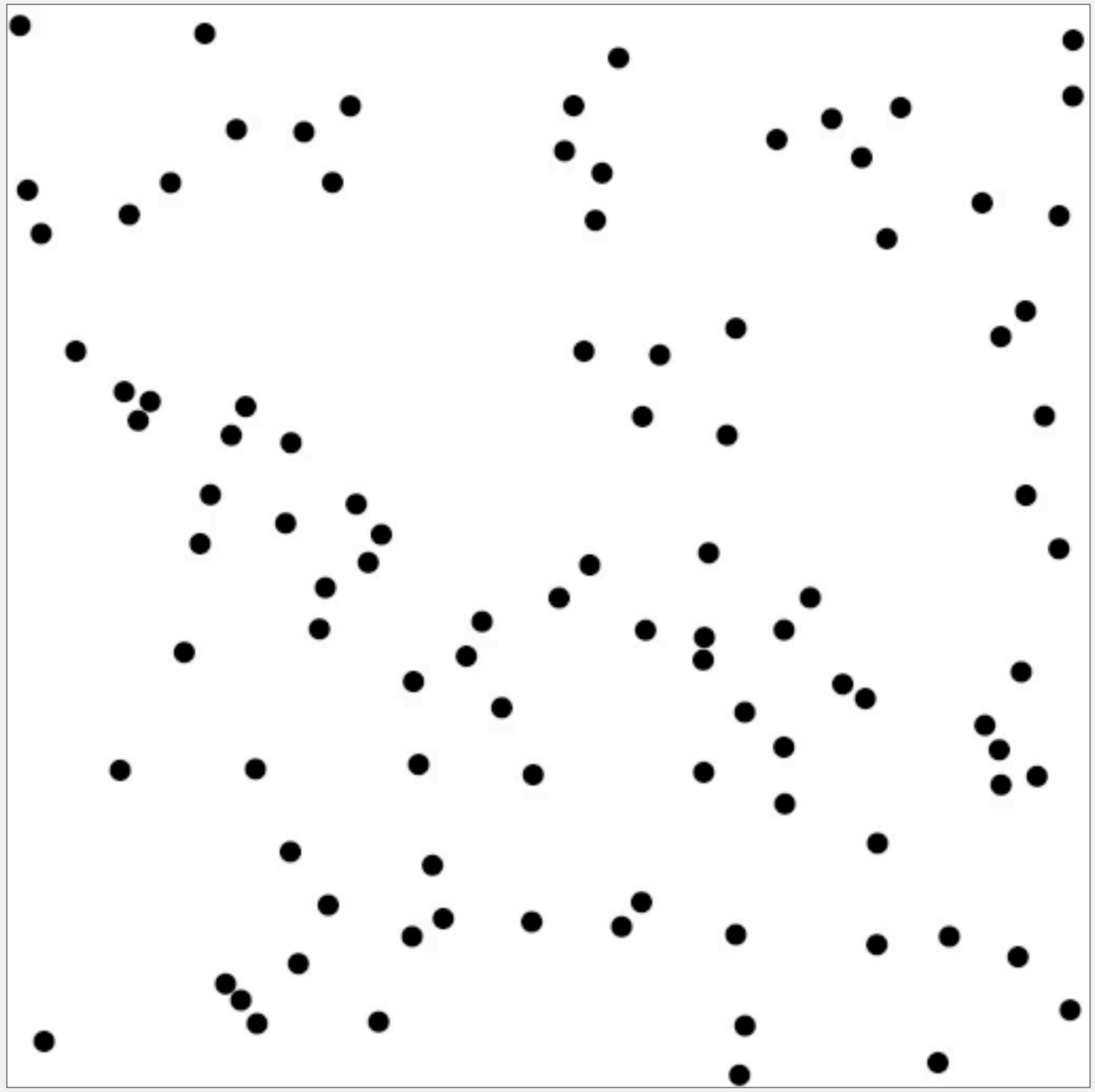
Generalizes: stack, queue (why?)

Simulation example 1

```
% java CollisionSystem 100
```

Simulation example 1

```
% java CollisionSystem 100
```



Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

```
% more tinyBatch.txt
Turing       6/17/1990   644.08
vonNeumann   3/26/2002  4121.85
Dijkstra     8/22/2007  2678.40
vonNeumann   1/11/1999  4409.74
Dijkstra    11/18/1995   837.42
Hoare        5/10/1993  3229.27
vonNeumann   2/12/1994  4732.35
Hoare        8/18/1992  4381.21
Turing       1/11/2002    66.10
Thompson     2/27/2000  4747.08
Turing       2/11/1991  2156.86
Hoare        8/12/2003  1025.70
vonNeumann  10/13/1993  2520.97
Dijkstra     9/10/2000   708.95
Turing      10/12/1993  3532.36
Hoare        2/10/2005  4050.20
```

sort key

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

- Fraud detection: isolate $$ transactions.
- File maintenance: find biggest files or directories.

Constraint. Not enough memory to store N items.

```
% more tinyBatch.txt
Turing       6/17/1990   644.08
vonNeumann   3/26/2002  4121.85
Dijkstra     8/22/2007  2678.40
vonNeumann   1/11/1999  4409.74
Dijkstra    11/18/1995   837.42
Hoare        5/10/1993  3229.27
vonNeumann   2/12/1994  4732.35
Hoare        8/18/1992  4381.21
Turing       1/11/2002    66.10
Thompson     2/27/2000  4747.08
Turing       2/11/1991  2156.86
Hoare        8/12/2003  1025.70
vonNeumann  10/13/1993  2520.97
Dijkstra     9/10/2000   708.95
Turing      10/12/1993  3532.36
Hoare        2/10/2005  4050.20
```

```
% java TopM 5 < tinyBatch.txt
Thompson     2/27/2000  4747.08
vonNeumann   2/12/1994  4732.35
vonNeumann   1/11/1999  4409.74
Hoare        8/18/1992  4381.21
vonNeumann   3/26/2002  4121.85
```

sort key

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

use a min-oriented pq

Transaction data type is Comparable

pq contains largest M items

Exercise: Theoretical analysis of the above, using properties of priority queue implementations.

Priority queue client example

Challenge. Find the largest M items in a stream of N items (N huge, M large).

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();

while (StdIn.hasNextLine())
{
   String line = StdIn.readLine();
   Transaction item = new Transaction(line);
   pq.insert(item);
   if (pq.size() > M)
      pq.delMin();
}
```

use a min-oriented pq

Transaction data type is Comparable

pq contains largest M items

Exercise: Theoretical analysis of the above, using properties of priority queue implementations.

How to implement a priority queue ADT?

Answer 1: Special case of search tree API.

Answer 2: Can do this simpler!

Recursive definition:

A priority queue (PQ) for n elements can be represented as:

- The maximum element, and
- two recursive PQs for remaining about (n-1)/2 elements each, if n>1.

Problem session

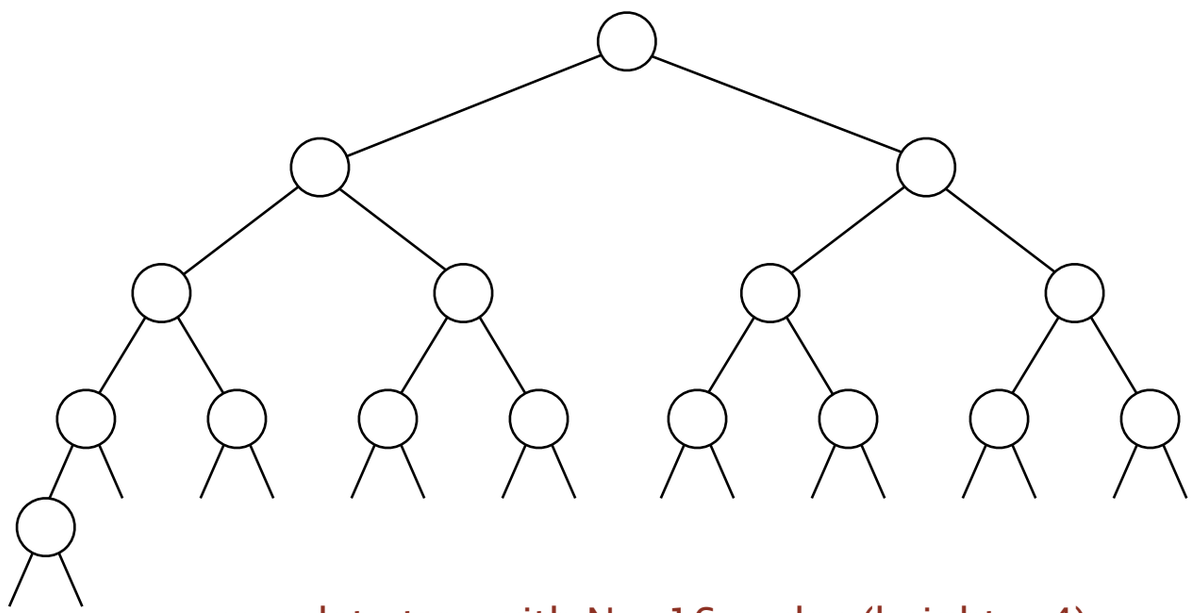
If one defines a PQ recursively as above, how can one implement:

- max?
- insert?
- delMax?

Complete binary tree

For an elegant implementation of the previous idea, we need an alternative tree representation.

Complete tree: Perfectly balanced, except for bottom level.



complete tree with N = 16 nodes (height = 4)

Property. Height of complete binary tree with N nodes is $\lfloor \lg N \rfloor$.

Pf. Height only increases when N is a power of 2.

Binary heap representations

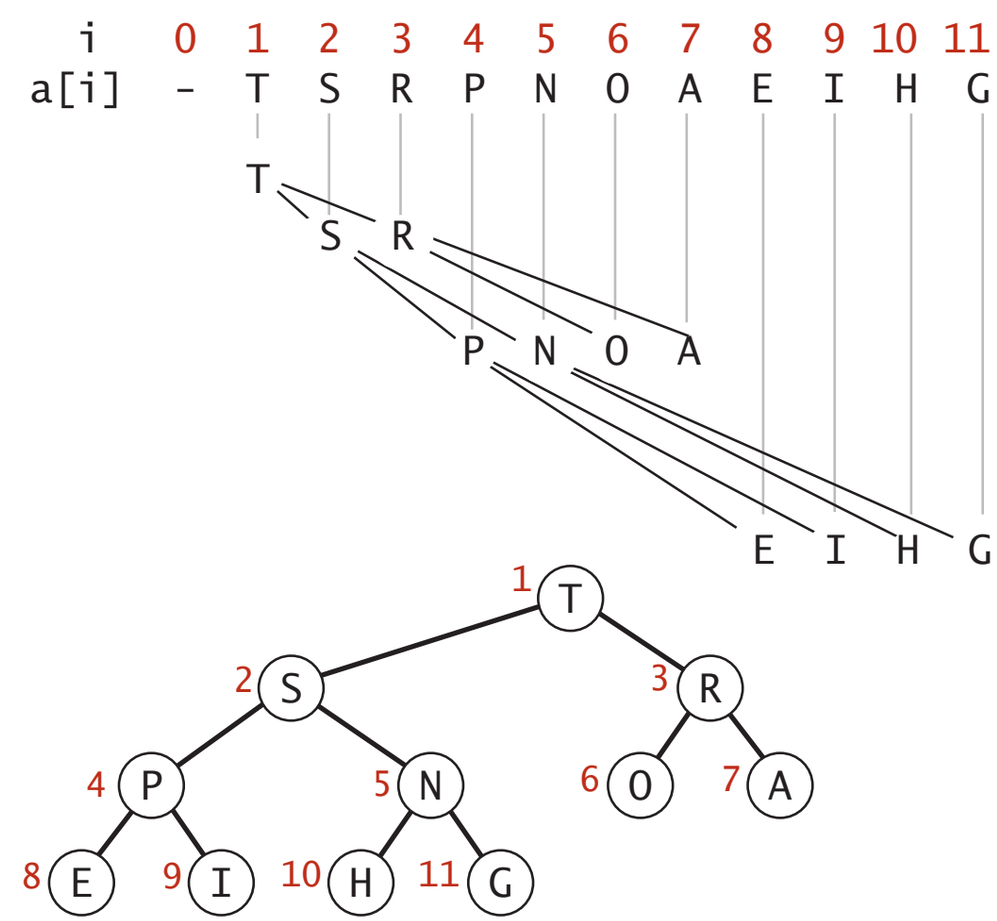
Binary heap. Array representation of a heap-ordered complete binary tree.

Heap-ordered binary tree.

- Keys in nodes.
- No smaller than children's keys.

Array representation.

- Take nodes in level order.
- No explicit links needed!



Heap representations

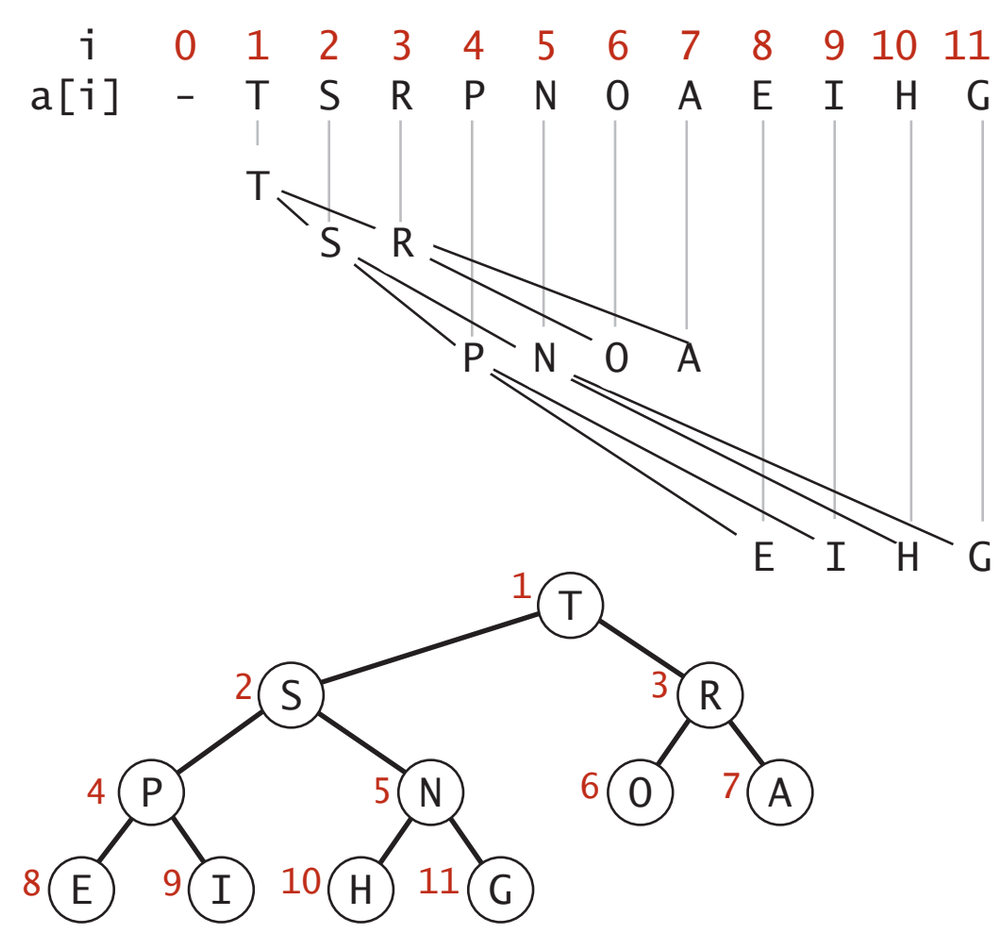
Binary heap properties

Proposition. Largest key is `a[1]`, which is root of binary tree.

indices start at 1

Proposition. Can use array indices to move through tree.

- Parent of node at `k` is at `k/2`.
- Children of node at `k` are at `2k` and `2k+1`.



Heap representations

Promotion in a heap

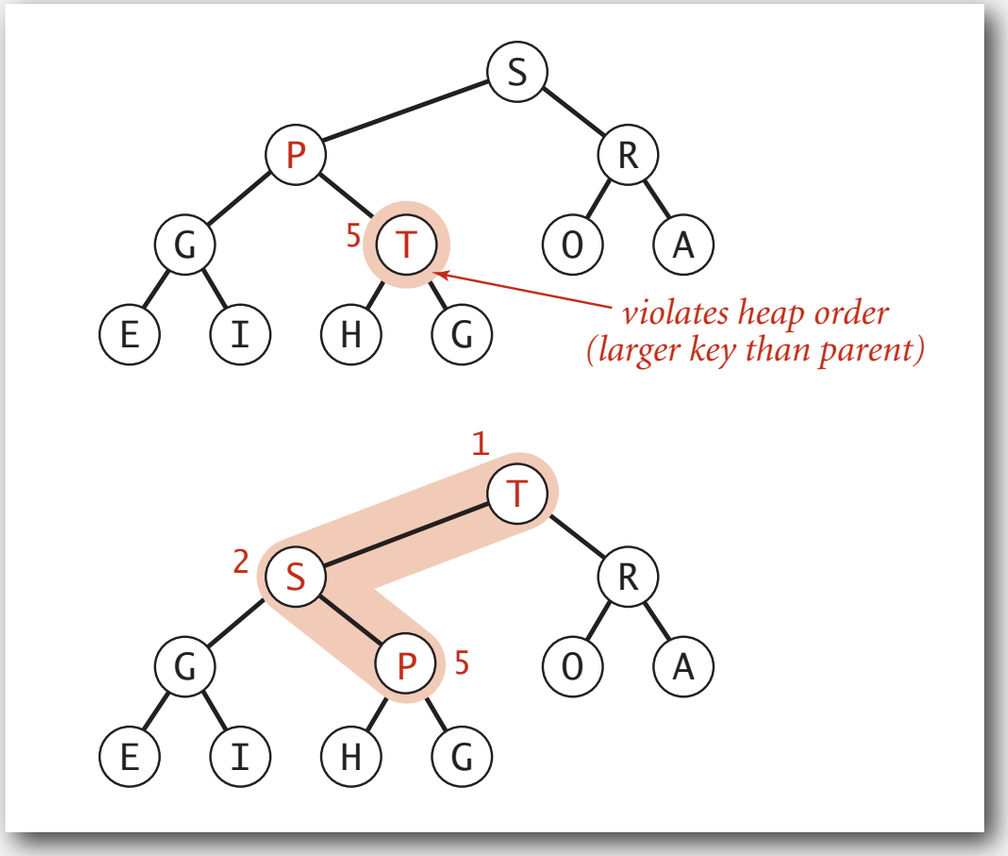
Scenario. Node's key becomes larger key than its parent's key.

To eliminate the violation:

- Exchange key in node with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
   while (k > 1 && less(k/2, k))
   {
      exch(k, k/2);
      k = k/2;
   }
}
```

parent of node at k is at k/2

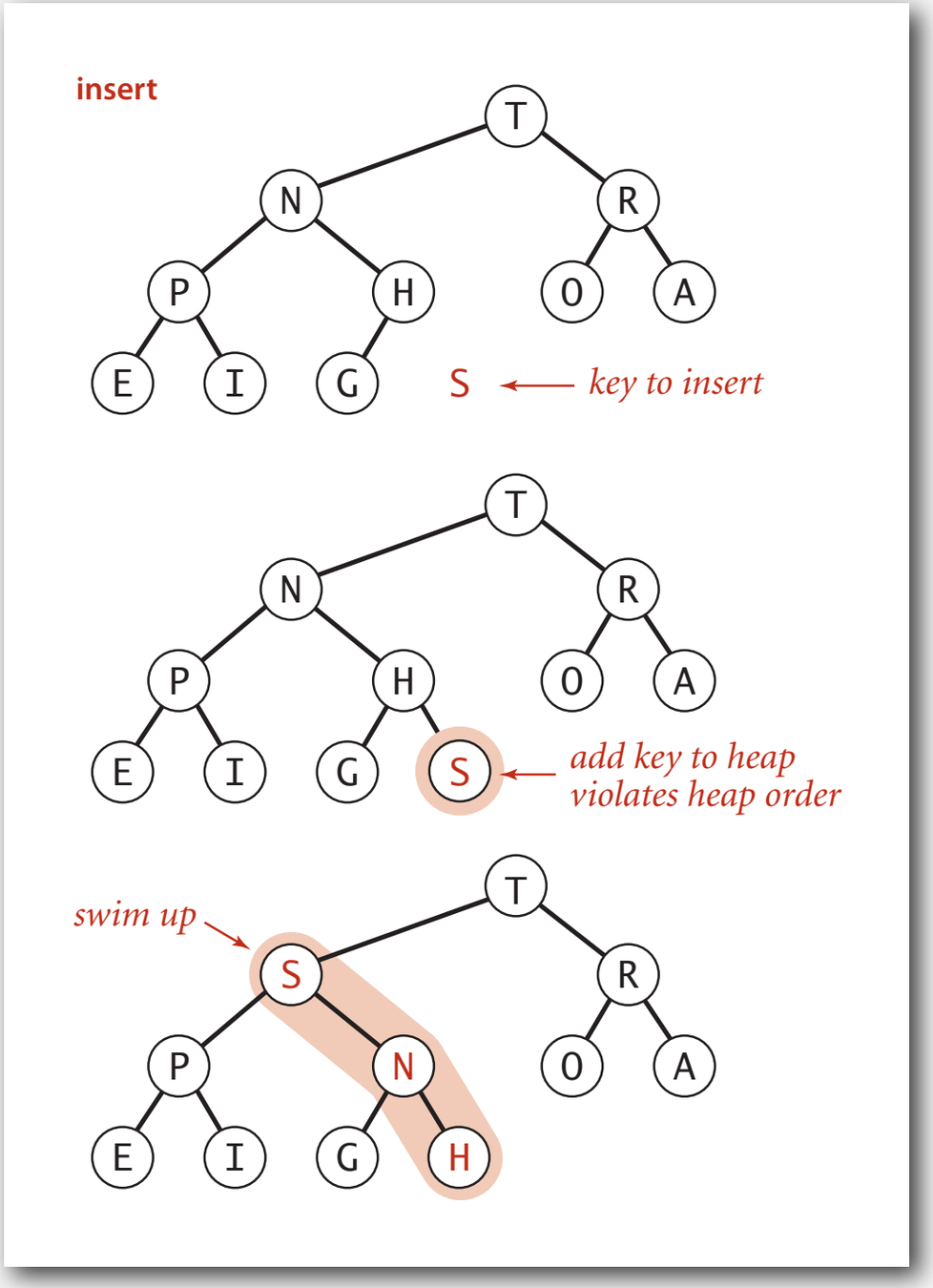


Insertion in a heap

Insert. Add node at end, then swim it up.

Cost. At most $1 + \lg N$ compares.

```
public void insert(Key x)
{
   pq[++N] = x;
   swim(N);
}
```



Demotion in a heap

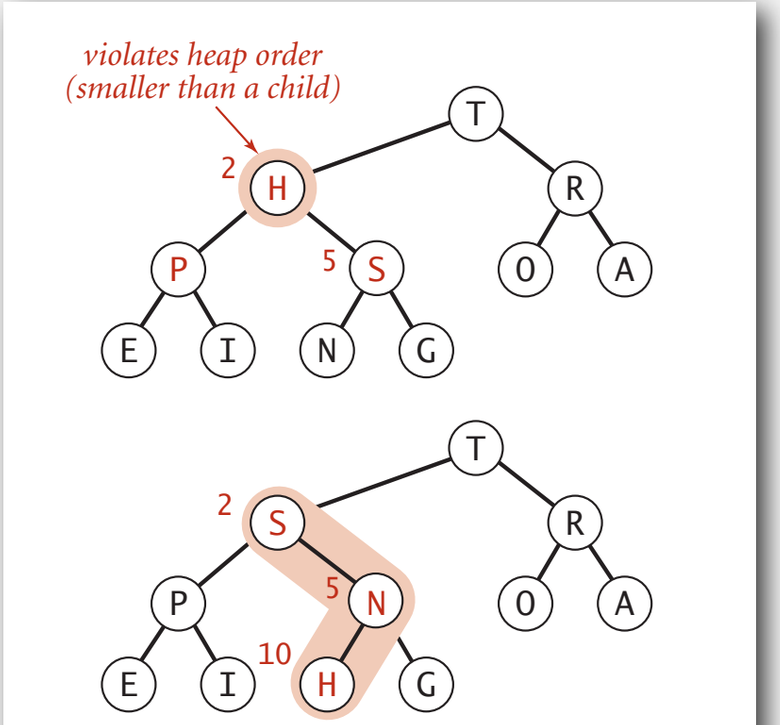
Scenario. Node's key becomes smaller than one (or both) of its children's keys.

To eliminate the violation:

- Exchange key in node with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
   while (2*k <= N)
   {
      int j = 2*k;
      if (j < N && less(j, j+1)) j++;
      if (!less(k, j)) break;
      exch(k, j);
      k = j;
   }
}
```

children of node at k are 2k and 2k+1



Top-down reheapify (sink)

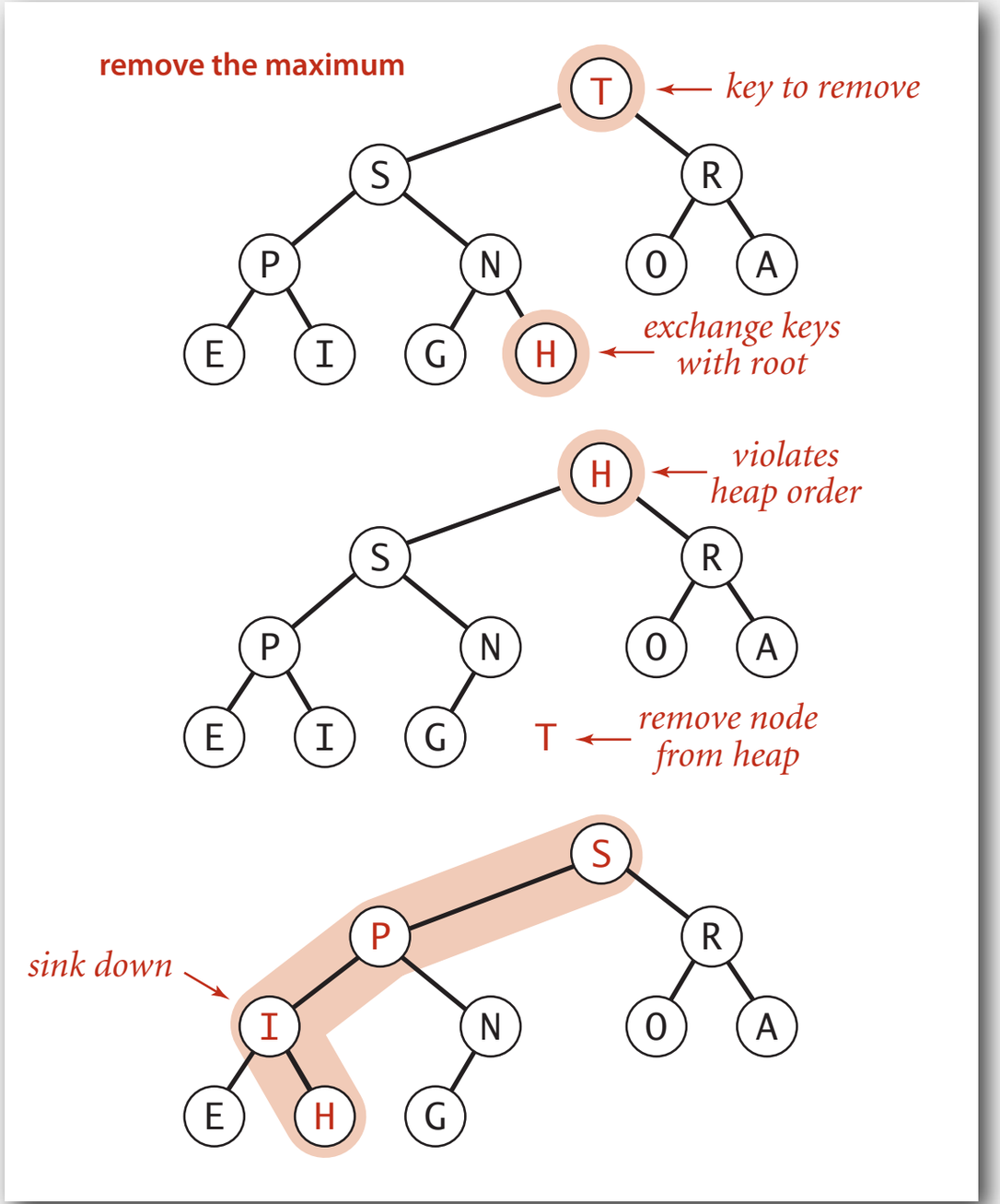
Delete the maximum in a heap

Delete max. Exchange root with node at end, then sink it down.

Cost. At most $2 \lg N$ compares.

```
public Key delMax()
{
   Key max = pq[1];
   exch(1, N--);
   sink(1);
   pq[N+1] = null;
   return max;
}
```

prevent loitering



Priority queues with remove, key updates

- Some priority queue ADTs support the removal of a specific item
  - Need to specify a "handle" to identify the item

- Two ways to do this **efficiently** for a heap:
  - 1. Maintain a hash table that keeps track of the location of each item in the heap.
  - 2. Leave removed items in the heap. When deleting the maximum, check if the item has been deleted (using a hash table).

- Some priority queue ADTs give special methods for changing (increasing/decreasing) the priority of an item.
  - Changes that move items towards the front of the queue are fast!

Priority queues implementation cost summary

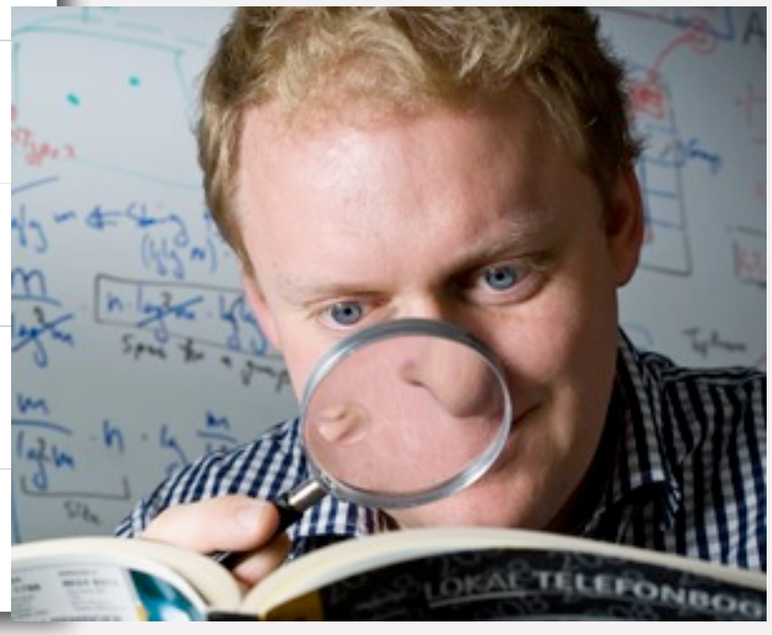
order-of-growth of number of comparisons for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
binary heap	log N	log N	1
d-ary heap	$\log_d N$	$d \log_d N$	1
Fibonacci	1	log N †	1
impossible	1	1	1

† amortized

why impossible?

Gerth Brodal, Aarhus University

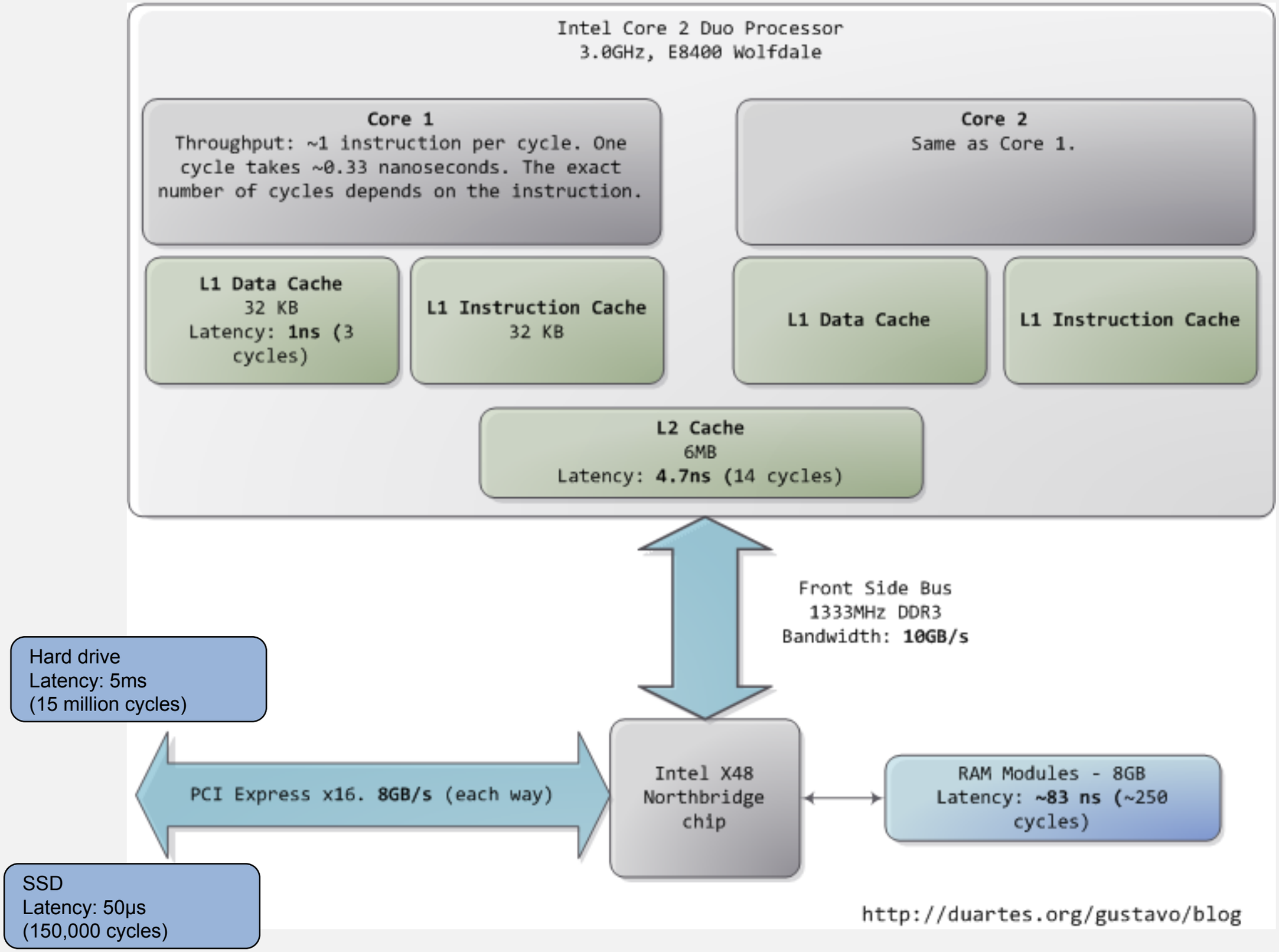


- java.util.PriorityQueue
  - decreaseKey through remove + insert
  - From Java 1.5 API documentation:

> An unbounded priority queue based on a priority heap. … Implementation note: this implementation provides O(log(n)) time for the insertion methods …; **linear time** for the remove(Object) and contains(Object) methods; and constant time for the retrieval methods (peek, element, and size).

- An implementation with better remove performance can be found e.g. in the JDSL library.

Is number of comparisons the right measure of complexity?



Heaps and cache-efficiency

Suppose

- We have a heap of primitive data types (not objects), e.g. integers.
- Our heap is stored in a memory with block size at least d (each memory access transfers a block of d words of memory).

Memory transfers of binary heap

- 1 transfer to access first log d levels
- 1 transfer for each subsequent level
- **Total**: 1+log(n)-log(d)

Memory transfers of d-ary heap

- 1 transfer to access each level
- **Total**: log(n)/log(d)

Problem: Different d needed to optimize for different block sizes.

van Emde Boas (vEB) layout

Presented in master's thesis of Prokop (1999):

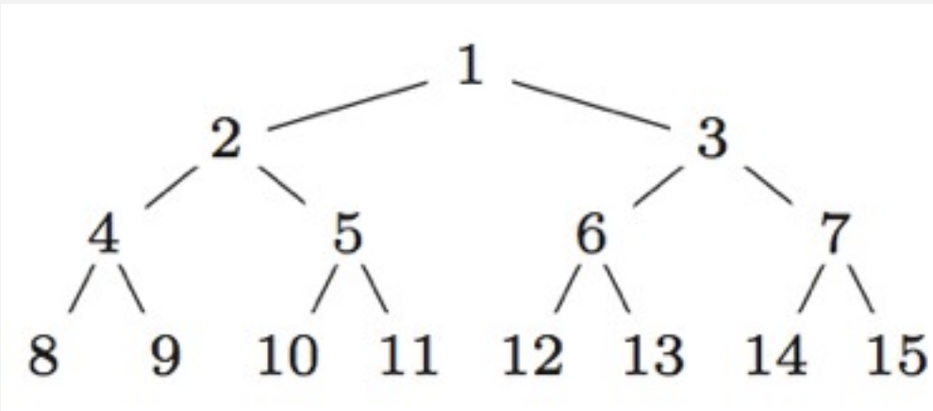
Alternative way of laying out a complete binary tree in an array.

Observation: A tree of depth d can be "cut up" into $2^{d/2}+1$ trees:

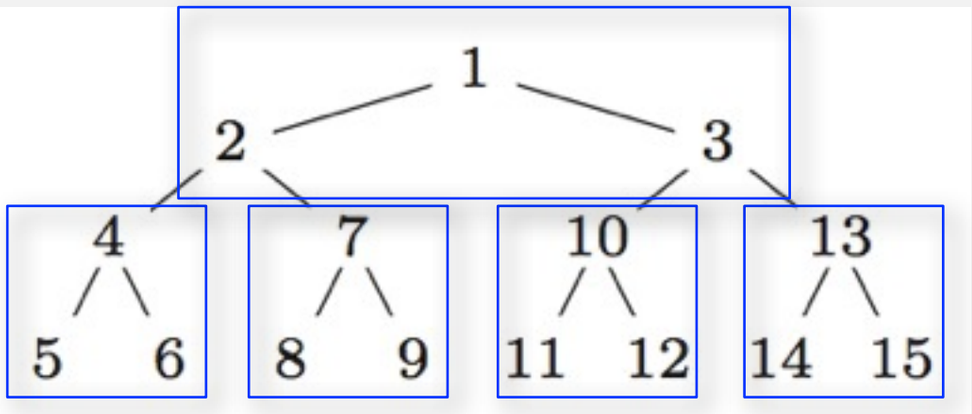
- A top tree of the first d/2 levels
- 2^d "bottom" trees with root at level d/2+1

vEB layout:

- First store top tree (recursively using vEB layout)
- Then store each bottom tree (recursively using vEB layout)



Standard heap layout (level order)



van Emde Boas layout

Properties of van Emde Boas layout

1. Can efficiently compute children/parent of node, given a suitable lookup table.

2. The number of memory transfers at **any level** of the memory hierarchy in a root-to-leaf traversal is within a factor 2 of optimal.

"Cache-oblivious algorithms" make efficient use of caches without considering cache parameters.

There exits heaps (such as the **funnel heap**) that are theoretically better than a binary heap with vEB layout.

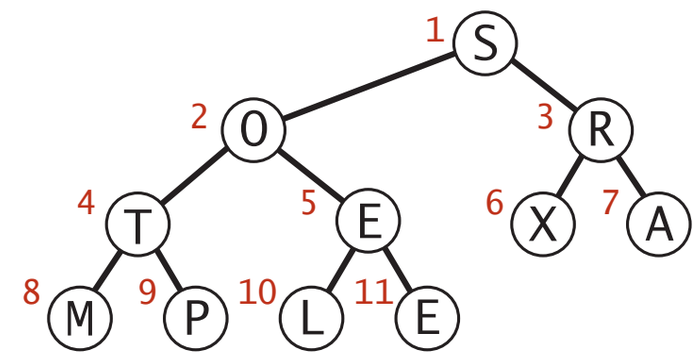
- API
- elementary implementations
- binary heaps
- **heapsort**
- event-driven simulation

Heapsort

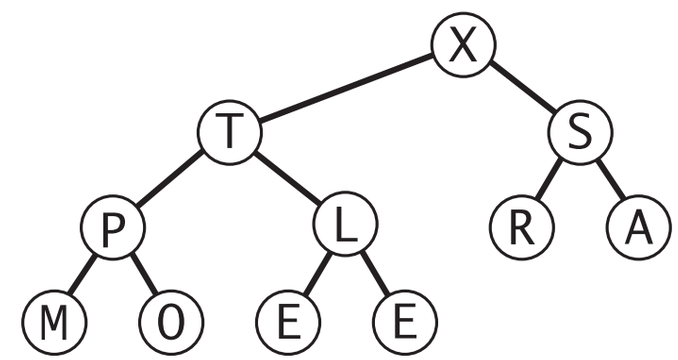
Basic plan for in-place sort.

- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

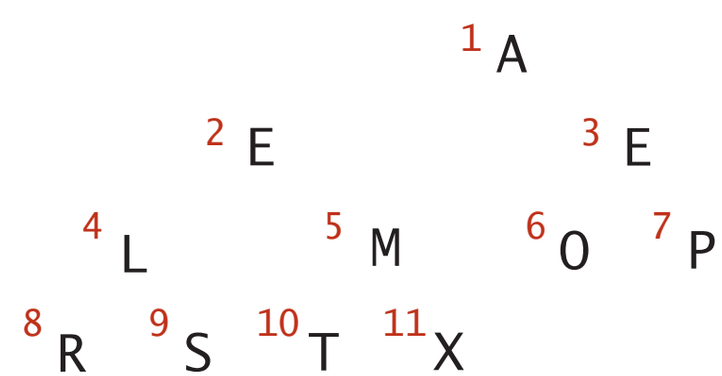
start with array of keys in arbitrary order



build a max-heap (in place)



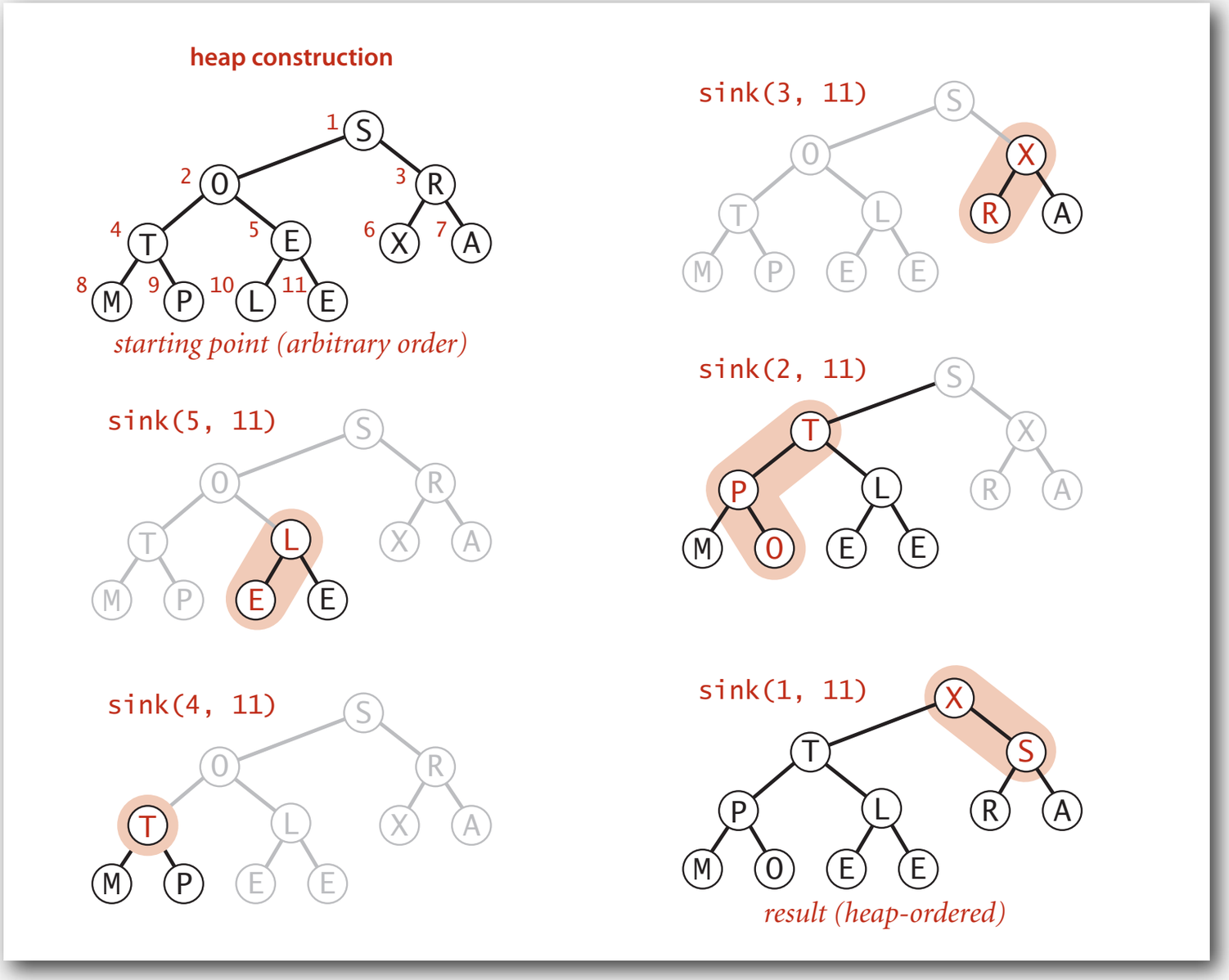
sorted result (in place)



Heapsort: heap construction

First pass. Build heap using bottom-up method.

```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```

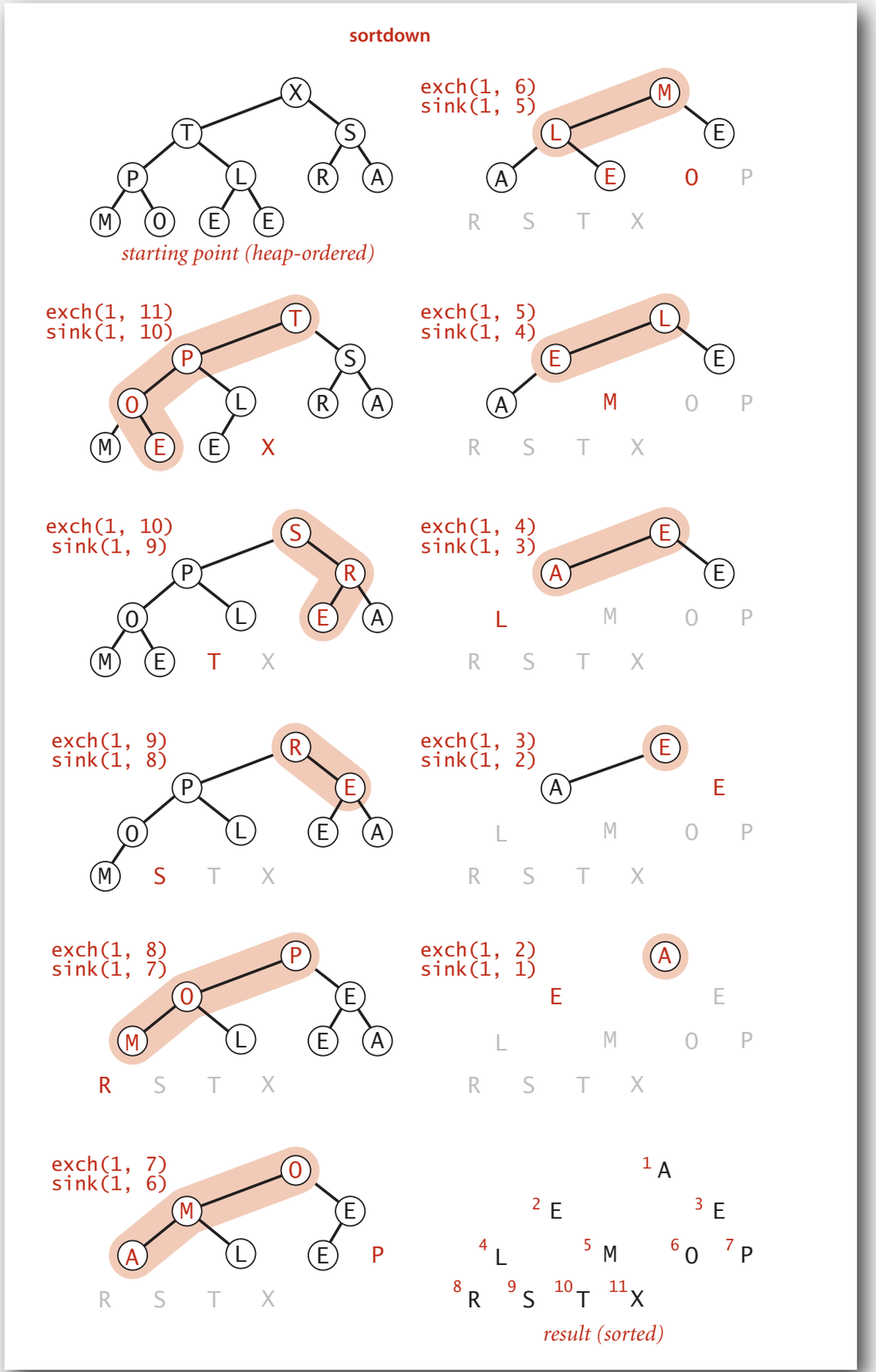


Heapsort: sortdown

Second pass.

- Remove the maximum, one at a time.
- Leave in array, instead of nulling out.

```
while (N > 1)
{
   exch(a, 1, N--);
   sink(a, 1, N);
}
```



Heapsort: mathematical analysis

Proposition. Heap construction uses fewer than $2\,N$ compares and exchanges.

Proposition. Heapsort uses at most $2\,N \lg N$ compares and exchanges.

Significance. In-place sorting algorithm with $N \log N$ worst-case.

- Mergesort: no, linear extra space. ← in-place merge possible, not practical
- Quicksort: no, quadratic time in worst case. ← N log N worst-case quicksort possible, not practical
- Heapsort: yes!

Bottom line. Heapsort is optimal for both time and space, but:

- Inner loop longer than quicksort's.
- Makes poor use of cache memory.
- Not stable.

Conclusion

Heaps give a more space- and time-efficient implementation of the priority queue ADT.

- Yields a very space efficient sorting algorithm.
- Cache-efficiency can be achieved by changing the layout.

Be careful when deleting items from a heap - can take linear time in some implementations!

We will be using priority queues as building blocks in graph algorithms.