Priority queues and Heapsort

Feb. 25, 2016

Acknowledgement: The course slides are adapted from the slides prepared by R. Sedgewick and K. Wayne of Princeton University.

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.

Randomized queue. Remove a random item.

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

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td>Q</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td>L</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>E</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td>L</td>
</tr>
</tbody>
</table>

Priority queue API

Requirement. Generic items are comparable.

Key must be Comparable (bounded type parameter)

public class MaxPQ<Key extends Comparable<Key>>

MaxPQ() create an empty priority queue

MaxPQ(Key[] a) create a priority queue with given keys

void insert(Key v) insert a key into the priority queue

Key delMax() return and remove the largest key

boolean isEmpty() is the priority queue empty?

Key max() return the largest key

int size() number of entries in the priority queue
Priority queue applications

- Event-driven simulation.
- Numerical computation.
- Data compression.
- Graph searching.
- Computational number theory.
- Artificial intelligence.
- Statistics.
- Operating systems.
- Discrete optimization.
- Spam filtering.

Generalizes: stack, queue, randomized queue.

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

```java
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();
}
pq contains largest $M$ items
```

sort key

PRIORITY QUEUES AND HEAPSORT

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation
Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return value</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td>P</td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td>P</td>
<td>2</td>
<td>Q</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>P</td>
<td>3</td>
<td>E</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td>E</td>
<td>3</td>
<td>X</td>
<td>E</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td>A</td>
<td>4</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td>L</td>
<td>6</td>
<td>L</td>
<td>L</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td>E</td>
<td>7</td>
<td>E</td>
<td>E</td>
</tr>
<tr>
<td>remove max</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue

Priority queue: unordered array implementation

```java
public class UnorderedMaxPQ<Key extends Comparable<Key>> {
    private Key[] pq; // pq[i] = ith element on pq
    private int N; // number of elements on pq

    public UnorderedMaxPQ(int capacity) {
        pq = (Key[]) new Comparable[capacity];
    }

    public boolean isEmpty() {
        return N == 0;
    }

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

    public Key delMax() {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max, i)) max = i;
        exch(max, N-1);
        return pq[--N];
    }
}
```

Priority queue elementary implementations

**Challenge.** Implement all operations efficiently.

```
<table>
<thead>
<tr>
<th>implementation</th>
<th>insert</th>
<th>del max</th>
<th>max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>goal</td>
<td>log N</td>
<td>log N</td>
<td>log N</td>
</tr>
</tbody>
</table>
```

Priority Queues and Heapsort

- Heapsort
- API
- Elementary implementations
- Binary heaps
- Heapsort
- Event-driven simulation
Binary tree

**Binary tree.** Empty or node with links to left and right binary trees.

**Complete tree.** Perfectly balanced, except for bottom level.

![Complete tree](image)

**Property.** Height of complete tree with $N$ nodes is $\lceil \lg N \rceil$.

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

A complete binary tree in nature

![A complete binary tree in nature](image)

Binary heap representations

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

**Heap-ordered binary tree.**
- Keys in nodes.
- Parent’s key no smaller than children’s keys.

**Array representation.**
- Indices start at 1.
- Take nodes in level order.
- No explicit links needed!

Binary heap properties

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

**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 $2k+1$. 
Promotion in a heap

Scenario. Child’s key becomes larger key than its parent’s key.

To eliminate the violation:
- Exchange key in child with key in parent.
- Repeat until heap order restored.

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

Peter principle. Node promoted to level of incompetence.

Insertion in a heap

Insert. Add node at end, then swim it up.
Cost. At most $1 + \lg N$ compares.

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

Demotion in a heap

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

To eliminate the violation:
- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```java
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;
    }
}
```

Power struggle. Better subordinate promoted.

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.

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

1. **Insert.** Add node at end, then swim it up.
2. **Remove the maximum.** Exchange root with node at end, then sink it down.

---

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

Remove the maximum. Exchange root with node at end, then sink it down.

---

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

Remove the maximum. Exchange root with node at end, then sink it down.

---

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

Remove the maximum. Exchange root with node at end, then sink it down.
Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**Insert.** Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**Binary heap operations**

**Insert.** Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**Binary heap operations**

**Insert.** Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**Binary heap operations**

**Insert.** Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

**Binary heap operations**
Binary heap operations

**Insert.** Add node at end, then swim it up.
*Remove the maximum.* Exchange root with node at end, then sink it down.

---

**Insert.** Add node at end, then swim it up.
*Remove the maximum.* Exchange root with node at end, then sink it down.

---

**Insert.** Add node at end, then swim it up.
*Remove the maximum.* Exchange root with node at end, then sink it down.

---

**Insert.** Add node at end, then swim it up.
*Remove the maximum.* Exchange root with node at end, then sink it down.
Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

heap ordered

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

remove the maximum
Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

1. Remove the maximum.

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

---

Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

1. Remove the maximum.

2. Exchange root with node at end, then sink it down.
Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.

Binary heap operations

Insert. Add node at end, then swim it up.
Remove the maximum. Exchange root with node at end, then sink it down.
Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

```
        1
       / \
      S   R
      / \  / \
     2   O N
    /  \ /  \
   E   C P
```

- Violates heap order (swim up)

```
S R O N P G A E I H
```

Binary heap operations

**Insert.** Add node at end, then swim it up.
**Remove the maximum.** Exchange root with node at end, then sink it down.

```
        S
       /|
      R | O
     / |  |
    N | P | C
   / |   / |
  E |   G   A
```

- Heap ordered

```
S R O N P G A E I H
```

Binary heap: Java implementation

```
public class MaxPQ<Key extends Comparable<Key>> {
private Key[] pq;
private int N;

public MaxPQ(int capacity) {
    pq = (Key[]) new Comparable[capacity+1];
}
	public boolean isEmpty() {
	    return N == 0;
	}
	public void insert(Key key) {
	    /* see previous code */
	}
	public Key delMax() {
	    /* see previous code */
	}

private void swim(int k) {
    /* see previous code */
}

private void sink(int k) {
    /* see previous code */
}

private boolean less(int i, int j) {
    return pq[i].compareTo(pq[j]) < 0;
}

private void exch(int i, int j) {
    Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;
}
}
```

Priority queues implementation cost summary

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Insert</th>
<th>Del Max</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>unordered array</td>
<td>N</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>ordered array</td>
<td>N</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>binary heap</td>
<td>(\log N)</td>
<td>(\log N)</td>
<td>1</td>
</tr>
<tr>
<td>d-ary heap</td>
<td>(\log_d N)</td>
<td>(d \log_d N)</td>
<td>1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1</td>
<td>(\log N)</td>
<td>1</td>
</tr>
<tr>
<td>impossible</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

† Amortized

Why impossible?
Binary heap considerations

Immutability of keys.
- Assumption: client does not change keys while they’re on the PQ.
- Best practice: use immutable keys.

Underflow and overflow.
- Underflow: throw exception if deleting from empty PQ.
- Overflow: add no-arg constructor and use resizing array.

Minimum-oriented priority queue.
- Replace `less()` with `greater()`.
- Implement `greater()`.

Other operations.
- Remove an arbitrary item.
- Change the priority of an item.
  
  can implement with `sink()` and `swim()` [stay tuned]

Immutability: implementing in Java

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

```java
public final class Vector {
    private final int N;
    private final double[] data;

    public Vector(double[] data) {
        this.N = data.length;
        this.data = new double[N];
        for (int i = 0; i < N; i++)
            this.data[i] = data[i];
    }
    ...
}
```

Immutable. String, Integer, Double, Color, Vector, Transaction, Point2D.

Mutable. StringBuilder, Stack, Counter, Java array.

Immutability: properties

Data type. Set of values and operations on those values.

Immutable data type. Can’t change the data type value once created.

Advantages.
- Simplifies debugging.
- Safer in presence of hostile code.
- Simplifies concurrent programming.
- Safe to use as key in priority queue or symbol table.

Disadvantage. Must create new object for each data type value.

"Classes should be immutable unless there's a very good reason to make them mutable.... If a class cannot be made immutable, you should still limit its mutability as much as possible."

— Joshua Bloch (Java architect)
Heapsort

Basic plan for in-place sort.
- Create max-heap with all $N$ keys.
- Repeatedly remove the maximum key.

Heap construction. Build max heap using bottom-up method.

Starting point. Array in arbitrary order.

Heapsort: constructing (left) and sorting down (right) a heap.
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 5

Heap construction. Build max heap using bottom-up method.

sink 5

Heap construction. Build max heap using bottom-up method.

sink 4

Heap construction. Build max heap using bottom-up method.
Heapsort

Heap construction. Build max heap using bottom-up method.

sink 3

sink 2
Heap construction. Build max heap using bottom-up method.

sink 2

7-node heap

sink 2

sink 1
**Heapsort**

**Heap construction.** Build max heap using bottom-up method.

sink 1

```
X T S P L R A M O E E
1 3
```

**Heapsort**

**Heap construction.** Build max heap using bottom-up method.

end of construction phase

```
X T S P L R A M O E E
```

**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 11

```
X T S P L R A M O E E
```

**Heapsort**

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 11

```
X T S P L R A M O E E
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

1 E

2 T

3 S

4 P

5 L

6 O

7 R

8 A

9 X

ETSPLRAMEOX

1

2

4

1 E

2 P

3 L

4 R

5 A

6 X

7 T

8 S

9 M

10 O

11 E

12 X

13 T

14 E

15 A

16 S

17 R

18 L

19 P

20 E

21 X

22 T

23 E

24 A

25 S

26 R

27 L

28 P

29 E

30 X

31 T

32 E

33 A

34 S

35 R

36 L

37 P

38 E

39 X

40 T

41 E

42 A

43 S

44 R

45 L

46 P

47 E

48 X

49 T

50 E

51 A

52 S

53 R

54 L

55 P

56 E

57 X

58 T

59 E

60 A

61 S

62 R

63 L

64 P

65 E

66 X

67 T

68 E

69 A

70 S

71 R

72 L

73 P

74 E

75 X

76 T

77 E

78 S

79 P

80 L

81 R

82 A

83 X

84 T

85 E

86 A

87 S

88 R

89 L

90 P

91 E

92 X

93 T

94 E

95 A

96 S

97 R

98 L

99 P

100 E

101 X
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

Sortdown. Repeatedly delete the largest remaining item.

Sortable.

Exchange 1 and 10

Sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 9

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 8

sink 1

exchange 1 and 8

sink 1
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

sink 1

```
    P
   / \  
  O   E
 /   /  
M   L   A
```

Heapsort

```
POEML最早xingずSTX
1 2 4
```

Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 7

```
    1
   / \  
  O   P
 /   /  
M   L   E
 /   /   /
R   S   T  X
```

Heapsort

```
POEML最早xingずSTX
1 2
```

Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

exchange 1 and 7

```
    1
   / \  
  A   E
 /   /  
M   L   E
```

Heapsort

```
AOEML最早xingずPRSTX
1 7
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

A O E M L E P R S T X

1 2 4

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

A O E M L E P R S T X

1 2 4
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

sink 1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 6

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

exchange 1 and 5
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

A M O P
R S T X

E L E A M O P R S T X

1

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

sink 1

A M O P
R S T X

L E L E A M O P R S T X

1 2

Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 4

A M O P
R S T X

L E L E A M O P R S T X

1 4

111
Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

1. Exchange 1 and 4

```
A E E L M O P R S T X
```

2. Sink 1

```
E A E L M O P R S T X
```

Heapsort

**Sortdown.** Repeatedly delete the largest remaining item.

1. Sink 1

```
E A E L M O P R S T X
```

2. Sink 1

```
E A E L M O P R S T X
```
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 3

sink 1
Heapsort

Sortdown. Repeatedly delete the largest remaining item.

exchange 1 and 2

1 2

A E

E L M O P

R S T X

A E E L M O P R S T X

end of sortdown phase

A E

E L M O P

R S T X

A E E L M O P R S T X
Heapsort

Ending point. Array in sorted order.

```
A
E
E
L
M
O
P
R
S
T
X
```

1  2  3  4  5  6  7  8  9  10  11

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: heap construction

First pass. Build heap using bottom-up method.

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

Heapsort: Java implementation

```
public class Heap
{
    public static void sort(Comparable[] pq)
    {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq, k, N);
        while (N > 1)
        {
            exch(pq, 1, N);
            sink(pq, 1, --N);
        }
    }
    private static void sink(Comparable[] pq, int i, int N)
    { /* as before */ }
    private static boolean less(Comparable[] pq, int i, int j)
    { /* as before */ }
    private static void exch(Comparable[] pq, int i, int j)
    { /* as before */ }
}
```

but convert from 1-based indexing to 0-base indexing
Heapsort: trace

<table>
<thead>
<tr>
<th>N</th>
<th>k</th>
<th>a[1]</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>5</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>SORT L X A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>S O X T L R A M P E E</td>
</tr>
<tr>
<td>11</td>
<td>2</td>
<td>S T K P L R A M O E E</td>
</tr>
<tr>
<td>11</td>
<td>1</td>
<td>X T S P L R A M O E E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>a[1]</th>
<th>heap-ordered</th>
</tr>
</thead>
<tbody>
<tr>
<td>X T S P L R A M O E E</td>
<td></td>
</tr>
<tr>
<td>T P S O L R A M E E X</td>
<td></td>
</tr>
<tr>
<td>S P R O L E A M T X</td>
<td></td>
</tr>
<tr>
<td>R P E O L E A R S T X</td>
<td></td>
</tr>
<tr>
<td>P D E M L E A R S T X</td>
<td></td>
</tr>
<tr>
<td>G M E A L E P R S T X</td>
<td></td>
</tr>
<tr>
<td>M L E A E O P R S T X</td>
<td></td>
</tr>
<tr>
<td>L E E A M O P R S T X</td>
<td></td>
</tr>
<tr>
<td>E A E L M O P R S T X</td>
<td></td>
</tr>
<tr>
<td>E A E L M O P R S T X</td>
<td></td>
</tr>
<tr>
<td>A E E L M O P R S T X</td>
<td></td>
</tr>
<tr>
<td>A E E L M O P R S T X</td>
<td></td>
</tr>
</tbody>
</table>

Heapsort trace (array contents just after each sink)

Heapsort animation

Heapsort: mathematical analysis

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

**Proposition.** Heapsort uses at most $2N \log N$ compares and exchanges.

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

- **Mergesort:** no, linear extra space.
- **Quicksort:** no, quadratic time in worst case.
- **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.

Sorting algorithms: summary

<table>
<thead>
<tr>
<th>in-place?</th>
<th>stable?</th>
<th>worst</th>
<th>average</th>
<th>best</th>
<th>remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>selection</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N^2/2$</td>
<td>$N$ exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$N^2/4$</td>
<td>N</td>
</tr>
<tr>
<td>shell</td>
<td>x</td>
<td>?</td>
<td>?</td>
<td>N</td>
<td>tight code, subquadratic</td>
</tr>
<tr>
<td>quick</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>$N \log N$</td>
<td>probabilistic guarantee in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>$N^2/2$</td>
<td>$2N \ln N$</td>
<td>N</td>
<td>improves quicksort in presence of duplicate keys</td>
</tr>
<tr>
<td>merge</td>
<td>x</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>$2N \log N$</td>
<td>$2N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$ guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
<td>$N \log N$</td>
</tr>
</tbody>
</table>
Molecular dynamics simulation of hard discs

**Goal.** Simulate the motion of $N$ moving particles that behave according to the laws of elastic collision.

**Hard disc model.**
- Moving particles interact via elastic collisions with each other and walls.
- Each particle is a disc with known position, velocity, mass, and radius.
- No other forces.

**Significance.** Relates macroscopic observables to microscopic dynamics.
- Einstein: explain Brownian motion of pollen grains.

Warmup: bouncing balls

**Time-driven simulation.** $N$ bouncing balls in the unit square.

```java
public class BouncingBalls {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        Ball[] balls = new Ball[N];
        for (int i = 0; i < N; i++)
            balls[i] = new Ball();
        while (true) {
            StdDraw.clear();
            for (int i = 0; i < N; i++)
                balls[i].move(0.5);
            for (int i = 0; i < N; i++)
                balls[i].draw();
            StdDraw.show(50);
        }
    }
}
```

% java BouncingBalls 100

**main simulation loop**
Warmup: bouncing balls

```java
public class Ball {
    private double rx, ry; // position
    private double vx, vy; // velocity
    private final double radius; // radius
    public Ball() {
        /* initialize position and velocity */
    }
    public void move(double dt) {
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1.0 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1.0 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }
    public void draw() {
        StdDraw.filledCircle(rx, ry, radius);
    }
}
```

Check for balls colliding with each other.
• Physics problems: when? what effect?
• CS problems: which object does the check? too many checks?

Time-driven simulation

Main drawbacks.
• $\sim N^2/2$ overlap checks per time quantum.
• Simulation is too slow if $dt$ is very small.
• May miss collisions if $dt$ is too large.
  (if colliding particles fail to overlap when we are looking)

• Discretize time in quanta of size $dt$.
• Update the position of each particle after every $dt$ units of time, and check for overlaps.
• If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.

Event-driven simulation

Change state only when something happens.
• Between collisions, particles move in straight-line trajectories.
• Focus only on times when collisions occur.
• Maintain PQ of collision events, prioritized by time.
• Remove the min = get next collision.

Collision prediction. Given position, velocity, and radius of a particle, when will it collide next with a wall or another particle?

Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.
Particle-wall collision

Collision prediction and resolution.
• Particle of radius $s$ at position $(rx, ry)$.
• Particle moving in unit box with velocity $(vx, vy)$.
• Will it collide with a vertical wall? If so, when?

Collision prediction.
$\Delta t = \frac{d}{v} = \frac{|x_2 - x_1|}{v}$

Position after collision: $(x_1 + v \Delta t, y_1)$

Velocity after collision: $(-v_y, v_x)$

Particle-particle collision prediction

Collision prediction.
• Particle $i$: radius $s_i$, position $(rx_i, ry_i)$, velocity $(vx_i, vy_i)$.
• Particle $j$: radius $s_j$, position $(rx_j, ry_j)$, velocity $(vx_j, vy_j)$.
• Will particles $i$ and $j$ collide? If so, when?

Collision resolution. When two particles collide, how does velocity change?

$\Delta v_i = (\Delta vx, \Delta vy) = (vx_i - vx_j, vy_i - vy_j)$

$\Delta v_j = (\Delta vx, \Delta vy) = (vx_j - vx_i, vy_j - vy_i)$

$\Delta v = (\Delta vx, \Delta vy) = (\Delta vx)^2 + (\Delta vy)^2$

$\Delta v = (\Delta vx)^2 + (\Delta vy)^2$

Impulse due to normal force (conservation of energy, conservation of momentum)

$J_x = \frac{m_j \Delta v_x}{\sigma} = \frac{m_j \Delta v_x}{\sigma(m_i + m_j)}$

Important note: This is high-school physics, so we won’t be testing you on it!
Particle data type skeleton

```java
public class Particle {
    private double rx, ry; // position
    private double vx, vy; // velocity
    private final double radius; // radius
    private final double mass; // mass
    private int count; // number of collisions

    public Particle(...) { }
    public void move(double dt) { }
    public void draw() { }
    public double timeToHit(Particle that) { }
    public double timeToHitVerticalWall() { }
    public double timeToHitHorizontalWall() { }
    public void bounceOff(Particle that) { }
    public void bounceOffVerticalWall() { }
    public void bounceOffHorizontalWall() { }
    public Particle(...) { }
}
```

Particle-particle collision and resolution implementation

```java
public double timeToHit(Particle that) {
    if (this == that) return INFINITY;
    double dx  = that.rx - this.rx, dy  = that.ry - this.ry;
    double dvx = that.vx - this.vx; dvy = that.vy - this.vy;
    double dvdr = dx*dvx + dy*dvy;
    if( dvdr > 0) return INFINITY;
    double dvdv = dvx*dvx + dvy*dvy;
    double drdr = dx*dx + dy*dy;
    double sigma = this.radius + that.radius;
    double d = (dvdr*dvdr) - dvdv * (drdr - sigma*sigma);
    if (d < 0) return INFINITY;
    return -(dvdr + Math.sqrt(d)) / dvdv;
}
```

Collision system: event-driven simulation main loop

Initialization.
- Fill PQ with all potential particle-wall collisions.
- Fill PQ with all potential particle-particle collisions.

“potential” since collision may not happen if some other collision intervenes

Main loop.
- Delete the impending event from PQ (min priority = t).
- If the event has been invalidated, ignore it.
- Advance all particles to time \( t \), on a straight-line trajectory.
- Update the velocities of the colliding particle(s).
- Predict future particle-wall and particle-particle collisions involving the colliding particle(s) and insert events onto PQ.

Event data type

```java
private class Event implements Comparable<Event> {
    private double time; // time of event
    private Particle a, b; // particles involved in event
    private int countA, countB; // collision counts for a and b

    public Event(double t, Particle a, Particle b) { }
    public int compareTo(Event that) { return this.time - that.time; }
    public boolean isValid() { }
}
```

Conventions.
- Neither particle null ➞ particle-particle collision.
- One particle null ➞ particle-wall collision.
- Both particles null ➞ redraw event.
public class CollisionSystem
{
  private MinPQ<Event> pq;  // the priority queue
  private double t = 0.0;    // simulation clock time
  private Particle[] particles; // the array of particles

  public CollisionSystem(Particle[] particles) { }

  private void predict(Particle a)
  {
    if (a == null) return;
    for (int i = 0; i < N; i++)
    {
      double dt = a.timeToHit(particles[i]);
      pq.insert(new Event(t + dt, a, particles[i]));
    }
    pq.insert(new Event(t + a.timeToHitVerticalWall(), a, null));
    pq.insert(new Event(t + a.timeToHitHorizontalWall(), null, a));
  }

  private void redraw() { }
  public void simulate() {
    /* see next slide */
  }
}

Collision system implementation: skeleton

Collision system implementation: main event-driven simulation loop

public void simulate()
{
  pq = new MinPQ<Event>();
  for(int i = 0; i < N; i++) predict(particles[i]);
  pq.insert(new Event(0, null, null));

  while(!pq.isEmpty())
  {
    Event event = pq.delMin();
    if(!event.isValid()) continue;
    Particle a = event.a;
    Particle b = event.b;
    for(int i = 0; i < N; i++)
      particles[i].move(event.time - t);
    t = event.time;
    if      (a != null && b != null) a.bounceOff(b);
    else if (a != null && b == null) a.bounceOffVerticalWall();
    else if (a == null && b != null) b.bounceOffHorizontalWall();
    else if (a == null && b == null) redraw();
    predict(a);
    predict(b);
  }
}

Particle collision simulation example 1

% java CollisionSystem 100

Particle collision simulation example 2

% java CollisionSystem < billiards.txt
Particle collision simulation example 3

% java CollisionSystem < brownian.txt

Particle collision simulation example 4

% java CollisionSystem < diffusion.txt