Algorithms

ROBERT SEDGEWICK | KEVIN WAYNE

http://algs4.cs.princeton.edu

COLLECTIONS, IMPLEMENTATIONS, PRIORITY QUEUES

- *collections*
- *priority queues, sets, symbol tables*
- *heaps and priority queues*
- *heapsort*
- *event-driven simulation (optional)*

http://algs4.cs.princeton.edu

COLLECTIONS, IMPLEMENTATIONS, PRIORITY QUEUES

- *collections*
- *priority queues, sets, symbol tables*
- *heaps and priority queues*
- *heapsort*
- *event-driven simulation (optional)*

Collections

> *"The difference between a bad programmer and a good one is whether [the programmer] considers code or data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships."*
>
> *— Linus Torvalds (creator of Linux)*

Things we might like to represent

- Sequences of items.
- Sets of items.
- Mappings between items, e.g. jhug's grade is 88.4

Terminology

Abstract Data Type (ADT)

- A set of abstract values, and a collection of operations on those values.
- Operations:
  - Queue: enqueue, dequeue
  - Stack: push, pop
  - Union-Find: union, find, connected

Example: queue of integers

- A sequence of integers
  - Mathematical sequence, not any particular data structure!
- create: returns empty sequence.
- enqueue x: puts x on the right side of the sequence.
- dequeue x: removes and returns the element on the left-hand side of the sequence.

For more: COS326

Terminology

Abstract Data Type (ADT)
- A set of abstract values, and a collection of operations on those values.

Collection
- An abstract data type that contains a collection of data items.

Data Structure
- A specific way to store and organize data.
- Can be used to implement ADTs.
- Examples: Array, linked list, binary tree.

Terminology

Implementation
- Data structures are used to implement ADTs.
- Choice of data structure may involve performance tradeoffs.
  - Worst case vs. average case performance.
  - Space vs. time.
- Restricting ADT capabilities may allow better performance. [stay tuned]

Examples
- Queue
  - Linked list
  - Resizing array
- Randomized Queue
  - Linked list (slow, but you can do it!)
  - Resizing array

http://algs4.cs.princeton.edu

Collections, Implementations, Priority Queues

- collections
- *priority queues, sets, symbol tables*
- heaps and priority queues
- heapsort
- event-driven simulation (optional)

List (Java terminology)

(some) List operations

operation	parameters	returns	effect
contains	Item	boolean	checks if an item is in the list
add	Item		appends Item at end
add	index, Item		adds Item at position index
set	index, Item		replaces item at position index with Item
remove	index	boolean	removes item at index
remove	Item	boolean	removes Item if present in list
get	index	Item	returns item at index

Java implementations
- ArrayList
- LinkedList

Caveat
- Java list does not match standard ADT terminology.
- Abstract lists don't support random access.

Task

NSA Monitoring

- You receive 1,000,000,000 unencrypted documents every day.
- You'd like to save the 1,000 documents with the highest score for manual review.

In real code, pick a list implementation, e.g. LinkedList

```
public void process(List<Document> top1000, Document newDoc) {
   Document lowest = top1000.get(0);
   for (Document d : top1000)
      if (d.score() < lowest.score())
         lowest = d;

   if (newDoc.score() > lowest.score()) {
      top1000.remove(lowest);
      top1000.add(newDoc);
   }
}
```

List based solution

Priority queue

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

- MaxPQ: Supports largest operations.
- MinPQ: Supports smallest operations.

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

Min PQ operations.

operation	parameters	returns	effect
insert	Item		adds an item
min		Item	returns minimum Item
delMin		Item	deletes and returns minimum Item

Priority queue

operation	parameters	returns	effect
insert	Item		adds an item
min		Item	returns minimum Item
delMin		Item	deletes and returns minimum Item

Actual algs4 class

```
public void process(MinPQ<Document> top1000, Document newDoc) {
   top1000.insert(newDoc);
   top1000.delMin(newDoc);
}
```

PQ based solution

Advantages

- Much simpler code.
- ADT is problem specific. May be faster.

Priority queue

Implementation

- We'll get to that later.

> *"Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: **premature optimization is the root of all evil**. Yet we should not pass up our opportunities in that critical 3%" — Donald Knuth, Structured Programming with Go To Statements*

Sets

operation	parameters	returns	effect
add	Item		adds an item, only one copy may exist
contains		boolean	returns true if item is present
delete			deletes item

In real code, pick a set implementation, e.g. TreeSet

```
public Set<String> wordsInFile(In file) {
   Set<String> s = new Set<String>();

   while (!file.isEmpty())
      s.add(file.readString());

   return s;
}
```

Finding all words in a file

Application

Genre identification

- Collect set of all words from a song's lyrics.
- Compare against large dataset using machine learning techniques.
  - Guess genre.

Confusion Matrix

decision \ correct answer	Alt Country	Alt Rock	Bluegrass	Blues	Xtian Ska	Country	Deathmetal	Gothmetal	Gothrock	HiphopGrp	Indie Rock	Industrial	Pop Punk	Prog Rock	Wave1Punk	Wave2Punk	R&B	Rappers	Ska
Alt Country	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Alt Rock	0	6	0	0	0	0	0	0	1	0	0	0	1	1	0	0	0	0	0
Bluegrass	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Blues	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Xtian Ska	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Country	7	13	0	2	0	39	0	0	0	3	5	0	1	4	2	1	4	0	0
Deathmetal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gothmetal	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Gothrock	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
HiphopGrp	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0
Indie Rock	0	54	0	6	0	2	0	0	0	1	44	8	8	6	3	8	14	0	1
Industrial	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Pop Punk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Prog Rock	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Wave1Punk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Wave2Punk	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
R&B	0	0	0	4	0	0	0	0	0	1	1	0	0	0	0	0	18	0	0
Rappers	0	2	0	0	0	2	0	0	0	13	4	1	0	0	1	0	21	67	0
Ska	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Symbol tables

operation	parameters	returns	effect
put	Key, Value		associates Value with Key
contains	Key	boolean	returns true if Key is present
get	Key	Value	returns Value associated with Key (if any)
delete	Key	Value	deletes Key and returns Value

In real code, pick a symbol table implementation, e.g. TreeMap

```
public void countChars(SymT<Character, Integer> charCount, String s) {
   for (Character c : s.toCharArray())
      if (charCount.contains(c))
         charCount.put(c, charCount.get(c) + 1);
      else
         charCount.put(c, 1);
}
```

Adding letter counts to array of strings

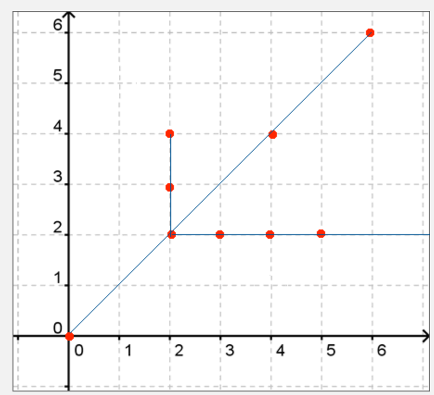
Other names

- Associative array, map, dictionary

Collinear

Collinear revisited (on board / projector)

- Collections make things easier.
- Likely to be slower and use more memory.



Design Problem

Solo in Groups

- **Erweiterten Netzwerk** is a new German minimalist social networking site that provides only two operations for its logged-in users.
  - Neu : Enter another user's username and click the **Neu** button. This marks the two users as friends.
  - Erweiterten Netzwerk : Type in another user's username and determine whether the two users are in the same extended network (i.e. there exists some chain of friends between the two users).

pollEv.com/jhug   text to **37607**

Identify at least one ADT that **Erweiterten Netzwerk** should use:

```
A. Queue        [879345]    D. Priority Queue        [879348]
B. Union-find   [879346]    E. Symbol Table          [879349]
C. Stack        [879347]    F. Randomized Queue      [879350]
```

Note: There may be more than one 'good' answer.

Implementations

Collection	Java Implementations	algs4 Implementations
List	LinkedList, ArrayList, Stack (oops)	None
MinPQ MaxPQ	PriorityQueue	MinPQ MaxPQ
Set	TreeSet, HashSet, CopyOnWriteArraySet, ...	SET (note: ordered)
Symbol Table	TreeMap, HashMap, ConcurrentHashMap, ...	RedBlackBST, SeparateChainingHashST, LinearProbingHashST, ...

Implementations

- Use algs4 classes when possible in COS226.
- When performance matters, pick the right implementation!
- Next two weeks: Implementation details of these collections.
- More collections to come.

http://algs4.cs.princeton.edu

Collections, Implementations, Priority Queues

- *collections*
- *priority queues, sets, symbol tables*
- *heaps and priority queues*
- *heapsort*
- *event-driven simulation (optional)*

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. [customers in a line, colliding particles] ← See optional slides / Coursera lecture.
- Numerical computation. [reducing roundoff error]
- Data compression. [Huffman codes]
- Graph searching. [Dijkstra's algorithm, Prim's algorithm]
- Number theory. [sum of powers]
- Artificial intelligence. [A* search] ← Assignment 4.
- 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, randomized queue.

Priority queue client example

Challenge. Find the largest M items in a stream of N items.

order of growth of finding the largest M in a stream of N items

implementation	time	space
sort	N log N	N
elementary PQ	M N	M
binary heap	N log M	M
best in theory	N	M

Priority queue: unordered and ordered array implementation

operation	*argument*	*return value*	*size*	*contents (unordered)*	*contents (ordered)*
insert	P		1	P	P
insert	Q		2	P Q	P Q
insert	E		3	P Q E	E P Q
remove max		Q	2	P E	E P
insert	X		3	P E X	E P X
insert	A		4	P E X A	A E P X
insert	M		5	P E X A M	A E M P X
remove max		X	4	P E M A	A E M P
insert	P		5	P E M A P	A E M P P
insert	L		6	P E M A P L	A E L M P P
insert	E		7	P E M A P L E	A E E L M P P
remove max		P	6	E M A P L E	A E E L M P

A sequence of operations on a priority queue

Priority queue: unordered array implementation

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

   public UnorderedArrayMaxPQ(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];
   }
}
```

no generic array creation (← `public UnorderedArrayMaxPQ(int capacity)`)

less() and exch() similar to sorting methods (but don't pass pq[]) (← `delMax()`)

should null out entry to prevent loitering (← `return pq[--N];`)

Priority queue elementary implementations

Challenge. Implement all operations efficiently.

order of growth of running time for priority queue with N items

implementation	insert	del max	max
unordered array	1	N	N
ordered array	N	1	1
goal	log N	log N	log N

Binary heap

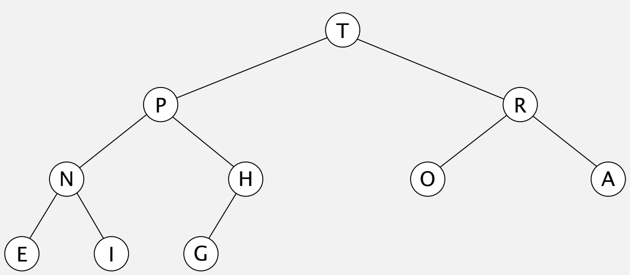
Basic idea on board

Binary heap demo

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

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

heap ordered

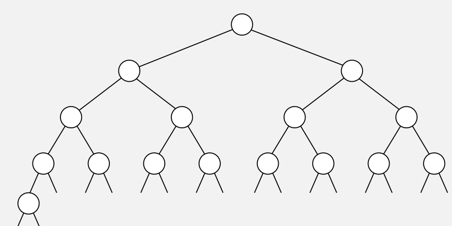


T	P	R	N	H	O	A	E	I	G	

Complete 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 with N = 16 nodes (height = 4)

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

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

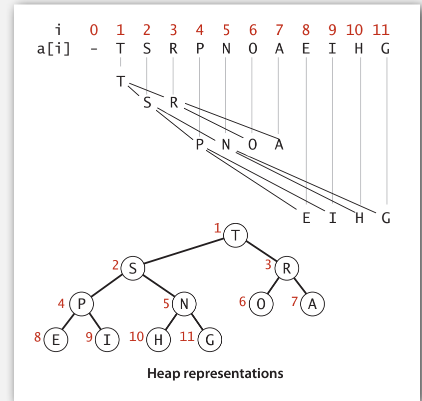
A complete binary tree in nature

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!

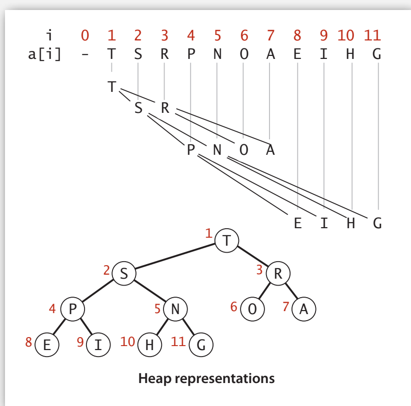


Heap representations

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`.



Heap representations

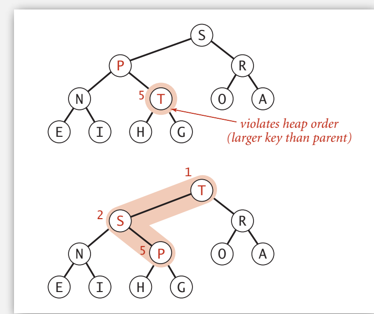
Promotion in a heap

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.

```
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



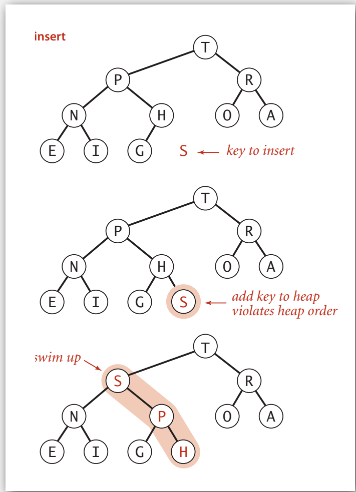
Peter principle. Node promoted to level of incompetence.

Insertion in a heap

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

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

```
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.

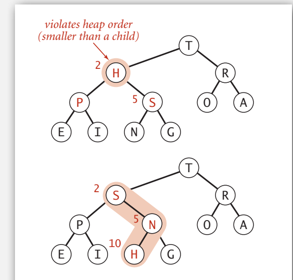
To eliminate the violation:

why not smaller child?

- Exchange key in parent 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)

Power struggle. Better subordinate promoted.

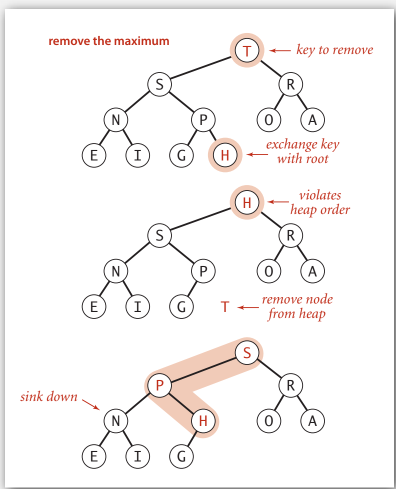
Delete the maximum in a heap

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

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

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

prevent loitering



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)
   public Key delMax()
   {   /* see previous code */   }

   private void swim(int k)
   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;  }
}
```

fixed capacity (for simplicity)

PQ ops

heap helper functions

array helper functions

Priority queues implementation cost summary

order-of-growth of running time 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
Brodal queue	1	log N	1
impossible	1	1	1

← why impossible?

† amortized

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.

leads to log N amortized time per op (how to make worst case?)

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.

```
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];
   }

   …
}
```

- `final class` ← can't override instance methods
- `private final` fields ← instance variables private and final
- copying loop ← defensive copy of mutable instance variables
- `…` ← instance methods don't change instance variables

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)*

http://algs4.cs.princeton.edu

Collections, Implementations, Priority Queues

- collections
- priority queues, sets, symbol tables
- heaps and priority queues
- *heapsort*
- event-driven simulation (optional)

Challenge

Design a sorting algorithm

- Given an Iterable<Comparable>.
- Design a sorting algorithm that only uses methods from the Set collection to print the items in order.

Challenge

Design a sorting algorithm

- Given an Iterable<Comparable>.
- Design a sorting algorithm that only uses methods from the Set collection to print the items in order.

```
public void HeapSort(Iterable<Comparable> a) {
   MaxPQ<Comparable> mpq = new MaxPQ<Comparable>();
   for (Comparable c : a)
      mpq.insert(c);

   for (Comparable c : a)
      System.out.println(mpq.delMax());
}
```

Performance

- Order of growth of running time: N lg N.
- Lots of unnecessary data movement and memory usage.

Heapsort

Observation

- Our heaps are represented with arrays.
  - Any array is just a messed up heap!

Heapsort can be done in place.

- Step 1: Heapify the array.
  - In place.
- Step 2: Delete the max repeatedly.
  - Largest element is swapped to the end.
  - Once completed, array is in order.
- Items take a round trip, but only a logarithmic distance.

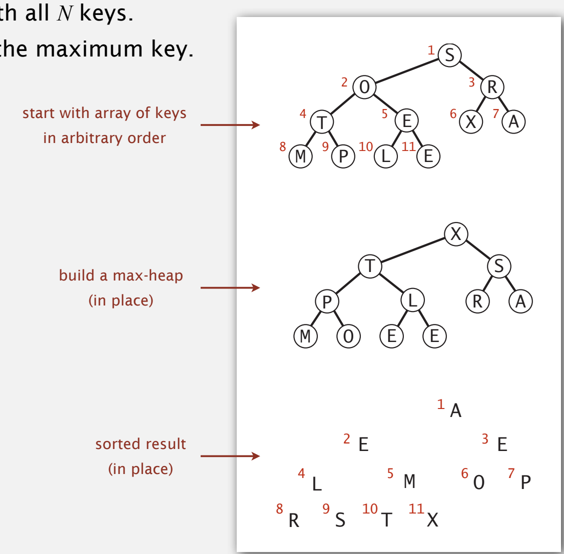
Heapsort has sometimes been described as the “[symbol]” algorithm, because of the motion of l and r. The upper triangle represents the heap creation phase, when $r = N$ and l decreases to 1; and the lower triangle represents the selection phase, when $l = 1$ and r decreases to 1.

Donald Knuth - The Art of Computer Programming Volume 3

Heapsort

Basic plan for in-place sort.

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

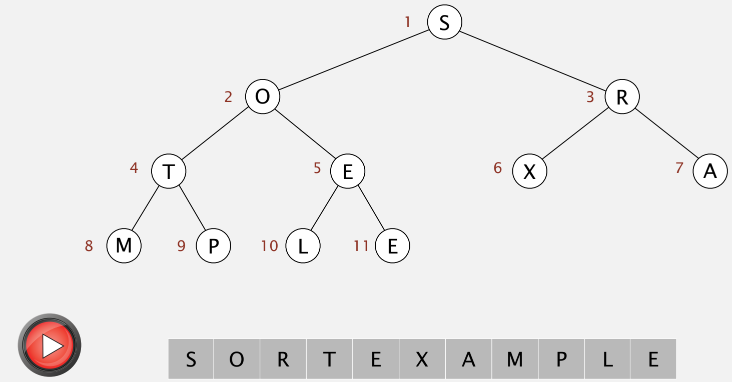


Heapsort demo

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

we assume array entries are indexed 1 to N

array in arbitrary order

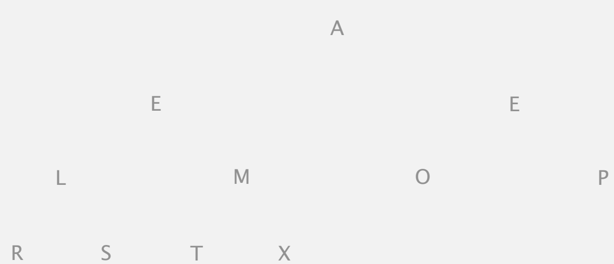


S	O	R	T	E	X	A	M	P	L	E
1	2	3	4	5	6	7	8	9	10	11

Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order

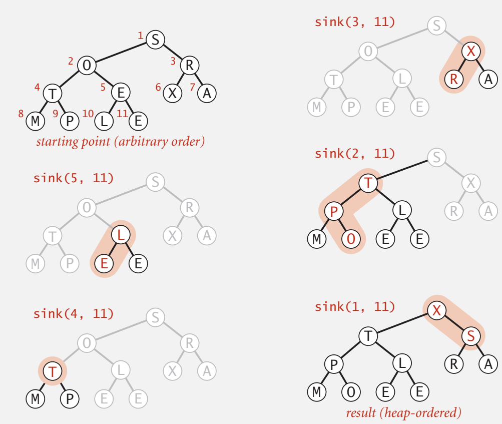


A	E	E	L	M	O	P	R	S	T	X
1	2	3	4	5	6	7	8	9	10	11

Heapsort: heap construction

First pass. Heapify using **bottom-up** method.

- Linear time (see book or optional slides).
- Top-down method is N lg N (see book or optional slides).



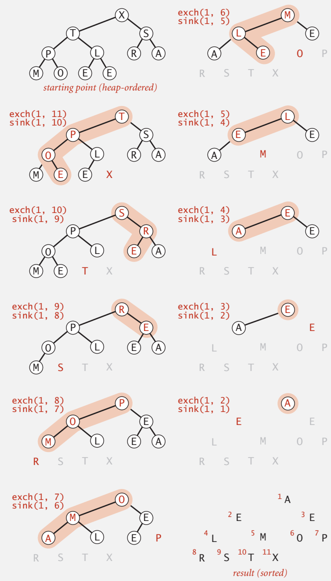

```
for (int k = N/2; k >= 1; k--)
   sink(a, k, N);
```

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: Java implementation

```
public class Heap
{
   public static void sort(Comparable[] a)
   {
      int N = a.length;
      for (int k = N/2; k >= 1; k--)
         sink(a, k, N);
      while (N > 1)
      {
         exch(a, 1, N);
         sink(a, 1, --N);
      }
   }

   private static void sink(Comparable[] a, int k, int N)
   {  /* as before */  }

   private static boolean less(Comparable[] a, int i, int j)
   {  /* as before */  }

   private static void exch(Object[] a, int i, int j)
   {  /* as before */  }

}
```

but make static (and pass arguments)

but convert from 1-based indexing to 0-base indexing

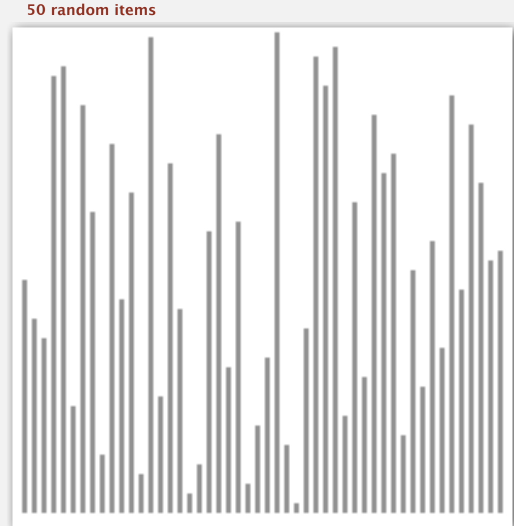
Heapsort: trace

N	k	0	1	2	3	4	5	6	7	8	9	10	11
initial values			S	O	R	T	E	X	A	M	P	L	E
11	5		S	O	R	T	L	X	A	M	P	E	E
11	4		S	O	R	T	L	X	A	M	P	E	E
11	3		S	O	X	T	L	R	A	M	P	E	E
11	2		S	T	X	P	L	R	A	M	O	E	E
11	1		X	T	S	P	L	R	A	M	O	E	E
heap-ordered			X	T	S	P	L	R	A	M	O	E	E
10	1		T	P	S	O	L	R	A	M	E	E	X
9	1		S	P	R	O	L	E	A	M	E	T	X
8	1		R	P	E	O	L	E	A	M	S	T	X
7	1		P	O	E	M	L	E	A	R	S	T	X
6	1		O	M	E	A	L	E	P	R	S	T	X
5	1		M	L	E	A	E	O	P	R	S	T	X
4	1		L	E	E	A	M	O	P	R	S	T	X
3	1		E	A	E	L	M	O	P	R	S	T	X
2	1		E	A	E	L	M	O	P	R	S	T	X
1	1		A	E	E	L	M	O	P	R	S	T	X
sorted result			A	E	E	L	M	O	P	R	S	T	X

Heapsort trace (array contents just after each sink)

Heapsort animation

50 random items



http://www.sorting-algorithms.com/heap-sort

algorithm position
in order
not in order

Heapsort: mathematical analysis

Proposition. Heap construction uses $\leq 2N$ compares and exchanges.
Proposition. Heapsort uses $\leq 2N \lg N$ compares and exchanges.

algorithm be improved to ~ 1 N lg N

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.

Heapsort summary

The good news:

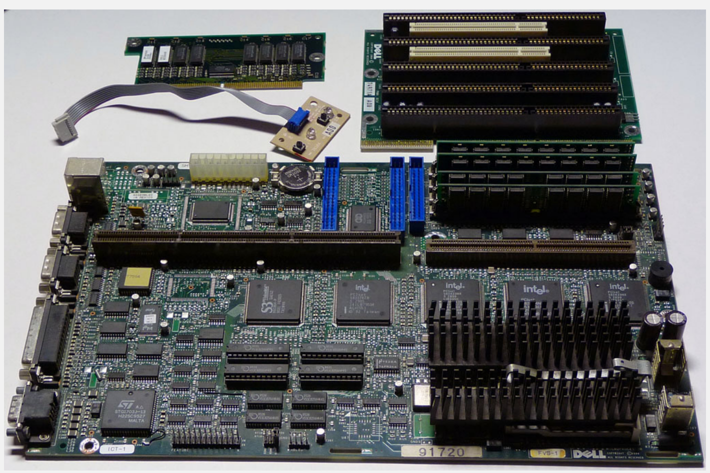
- Heap sort: In place and theoretically fast (not in place)

Mergesort	Quicksort	Heapsort
		

The bad news:

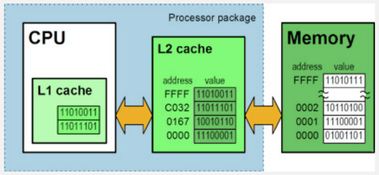
- (Almost) nobody uses Heapsort in the real world. Why?
  - Like Miss Manners, Heapsort is very well-behaved, but is unable to handle the stresses of the real world
  - In particular, performance on real computers is heavily impacted by really messy factors like cache performance

What does your computer look like inside?



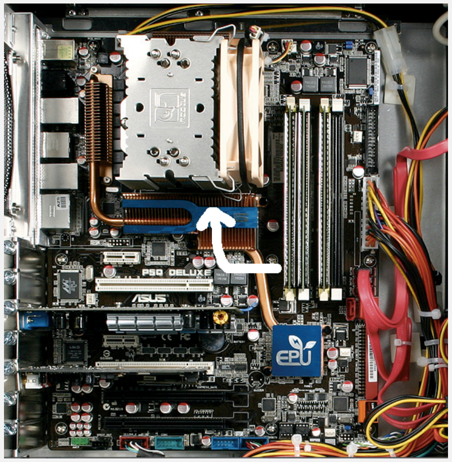
Play with it!

Levels of caches



We'll assume there's just one cache, to keep things simple.

Key idea behind caching



When fetching one memory address, fetch **everything** nearby.

- Because memory access patterns of most programs/algorithms are highly localized!

http://media.soundonsound.com/sos/dec08/images/DigitalVillageSynergyPC_04.jpg

Which of these is faster?

A.
```
sum=0
for (i = 0 to size)
  for (j = 0 to size)
    sum += array[i][j]
```

B.
```
sum=0
for (i = 0 to size)
  for (j = 0 to size)
    sum += array[j][i]
```

Answer: A is faster, sometimes by an order of magnitude or more.

Cache and memory latencies: an analogy

Cache
Get up and get something from the kitchen

RAM
Walk down the block to borrow from neighbor

Hard drive
Drive around the world...

...twice

Sort algorithms and cache performance

Mergesort: sort subarrays first 

Quicksort: partition into subarrays 

Heapsort: all over the place 

Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		½ N^2	½ N^2	½ N^2	N exchanges
insertion	x	x	½ N^2	¼ N^2	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		½ N^2	2 N ln N	N lg N	N log N probabilistic guarantee fastest in practice
3-way quick	x		½ N^2	2 N ln N	N	improves quicksort in presence of duplicate keys
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2 N lg N	2 N lg N	N lg N	N log N guarantee, in-place
???	x	x	N lg N	N lg N	N lg N	holy sorting grail

Optional slides on heapification running time

- Or see textbook section 2.4

Sink-based (bottom up) heapification

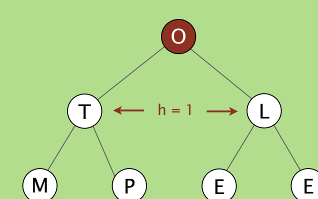
Observation

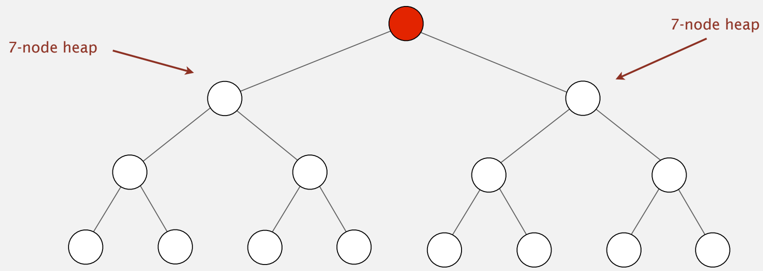
- Given two heaps of height 1.
- A heap of height 2 results by:
  - Pointing the root of each heap at a new item.
  - Sinking that new item.
- Cost: 4 compares (2 * height of new tree).

pollEv.com/jhug text to **37607**

Q: How many compares are needed to sink the O into the correct position in the worst case?

A. 1 [676050]
B. 2 [676051]
C. 3 [676052]
D. 4 [676053]





pollEv.com/jhug text to **37607**

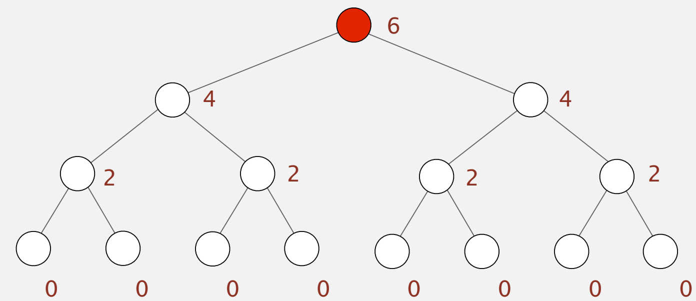
Q: How many worst-case compares are needed to form a height 3 heap by sinking an item into one of two perfectly balanced heaps of height 2?

A. 4 [676057]
B. 6 [676058]
C. 8 [676059]

Sink-based (bottom up) heapification

Observation

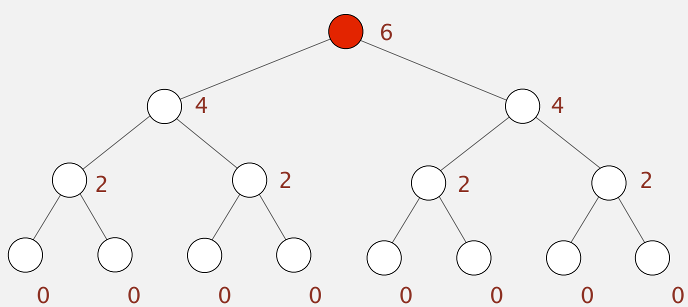
- Given two heaps of height h-1.
- A heap of height h results by
  - Pointing the root of each heap at a new item.
  - Sinking that new item.
- Cost to sink: At most 2h compares.
- Total heap construction cost: 4*2 + 2*4 + 6 = 22 compares



Sink-based (bottom up) heapification

Total Heap Construction Cost

- For h=1: $C_1 = 2$
- For h=2: $C_2 = 2C_1 + 2*2$
- For h: $C_h = 2C_{h-1} + 2h$
- Total cost: Doubles with h (plus a small constant factor): Exponential in h
- Total cost: Linear in N



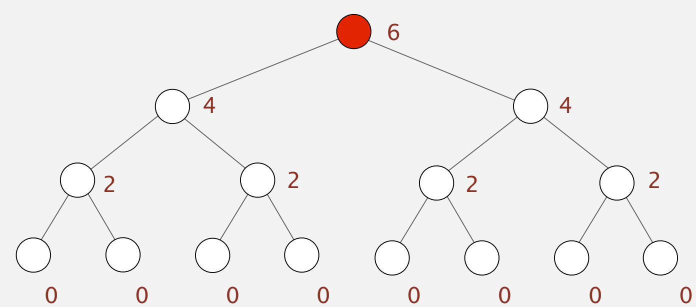
Heapsort

Order of growth of running time

- Heap construction: N
- N calls to delete max: N lg N

Total Extra Space

- Constant (in-place)



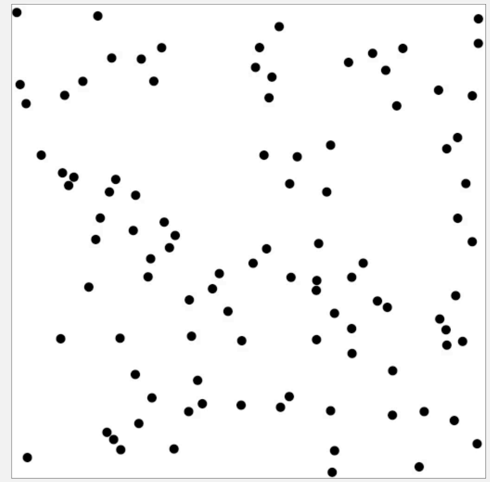
http://algs4.cs.princeton.edu

Collections, Implementations, Priority Queues

- collections
- priority queues, sets, symbol tables
- heaps and priority queues
- heapsort
- *event-driven simulation (optional)*

Molecular dynamics simulation of hard discs

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



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.

temperature, pressure, diffusion constant

motion of individual atoms and molecules

Significance. Relates macroscopic observables to microscopic dynamics.

- Maxwell-Boltzmann: distribution of speeds as a function of temperature.
- Einstein: explain Brownian motion of pollen grains.

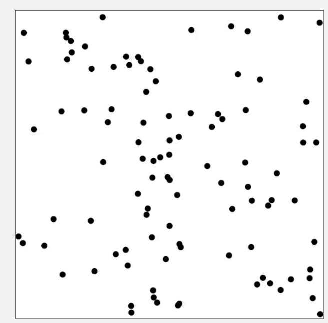
Warmup: bouncing balls

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

```
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);
            balls[i].draw();
         }
         StdDraw.show(50);
      }
   }
}
```

main simulation loop

% java BouncingBalls 100



Warmup: bouncing balls

```
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);  }
}
```

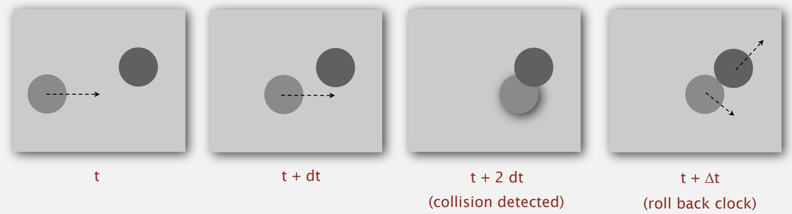
check for collision with walls

Missing. 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

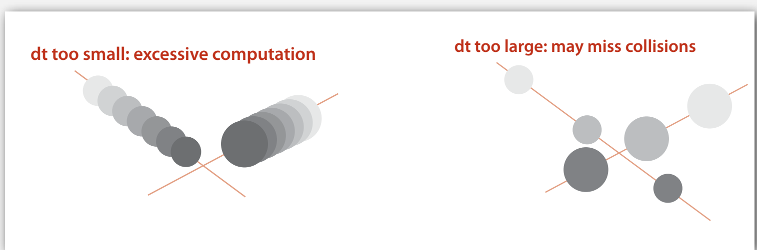
- Discretize time in quanta of size dt.
- Update the position of each particle after every dt units of time, and check for overlaps.
- If overlap, roll back the clock to the time of the collision, update the velocities of the colliding particles, and continue the simulation.



Time-driven simulation

Main drawbacks.

- ~ $N^2 / 2$ 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)



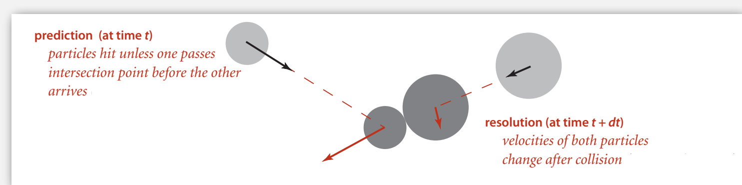
Event-driven simulation

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?

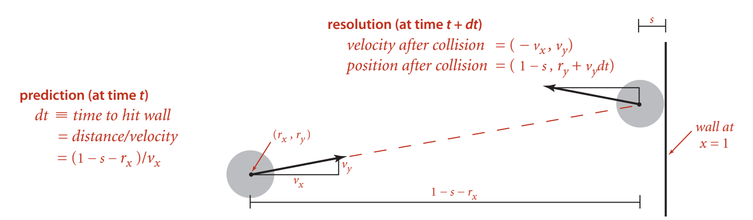
Collision resolution. If collision occurs, update colliding particle(s) according to laws of elastic collisions.



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?

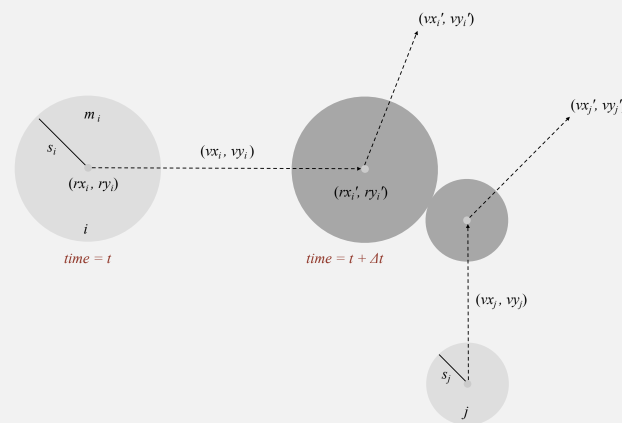


Predicting and resolving a particle-wall collision

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?



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?

$$\Delta t = \begin{cases} \infty & \text{if } \Delta v \cdot \Delta r \ge 0 \\ \infty & \text{if } d < 0 \\ -\dfrac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise} \end{cases}$$

$$d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v)\,(\Delta r \cdot \Delta r - \sigma^2) \qquad \sigma = \sigma_i + \sigma_j$$

$$\Delta v = (\Delta vx,\ \Delta vy) = (vx_i - vx_j,\ vy_i - vy_j)$$
$$\Delta r = (\Delta rx,\ \Delta ry) = (rx_i - rx_j,\ ry_i - ry_j)$$

$$\Delta v \cdot \Delta v = (\Delta vx)^2 + (\Delta vy)^2$$
$$\Delta r \cdot \Delta r = (\Delta rx)^2 + (\Delta ry)^2$$
$$\Delta v \cdot \Delta r = (\Delta vx)(\Delta rx) + (\Delta vy)(\Delta ry)$$

Important note: This is high-school physics, so we won't be testing you on it!

Particle-particle collision resolution

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

$$\begin{aligned} vx_i' &= vx_i + Jx / m_i \\ vy_i' &= vy_i + Jy / m_i \\ vx_j' &= vx_j - Jx / m_j \\ vy_j' &= vy_j - Jy / m_j \end{aligned}$$

Newton's second law (momentum form)

$$Jx = \frac{J\,\Delta rx}{\sigma},\quad Jy = \frac{J\,\Delta ry}{\sigma},\quad J = \frac{2\,m_i\,m_j\,(\Delta v \cdot \Delta r)}{\sigma(m_i + m_j)}$$

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

Important note: This is high-school physics, so we won't be testing you on it!

Particle data type skeleton

```
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()   { }

}
```

predict collision with particle or wall

resolve collision with particle or wall

Particle-particle collision and resolution implementation

```
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;
}
```

no collision

```
public void bounceOff(Particle that)
{
   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;
   double dist = this.radius + that.radius;
   double J = 2 * this.mass * that.mass * dvdr / ((this.mass + that.mass) * dist);
   double Jx = J * dx / dist;
   double Jy = J * dy / dist;
   this.vx += Jx / this.mass;
   this.vy += Jy / this.mass;
   that.vx -= Jx / that.mass;
   that.vy -= Jy / that.mass;
   this.count++;
   that.count++;
}
```

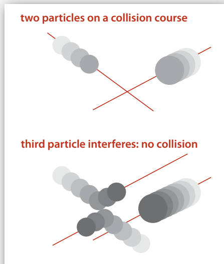
Important note: This is high-school physics, so we won't be testing you on it!

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



An invalidated event

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

Conventions.

- Neither particle `null` ⇒ particle-particle collision.
- One particle `null` ⇒ particle-wall collision.
- Both particles `null` ⇒ redraw event.

```
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()
    {   }
}
```

create event

ordered by time

invalid if intervening collision

Collision system implementation: skeleton

```
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 */  }
}
```

add to PQ all particle-wall and particle-particle collisions involving this particle

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);
   }
}
```

initialize PQ with collision events and redraw event

get next event

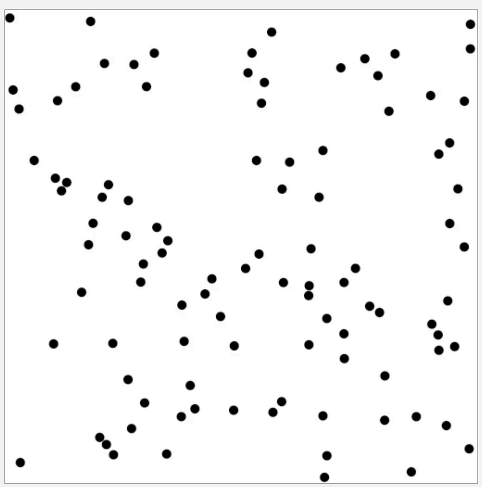
update positions and time

process event

predict new events based on changes

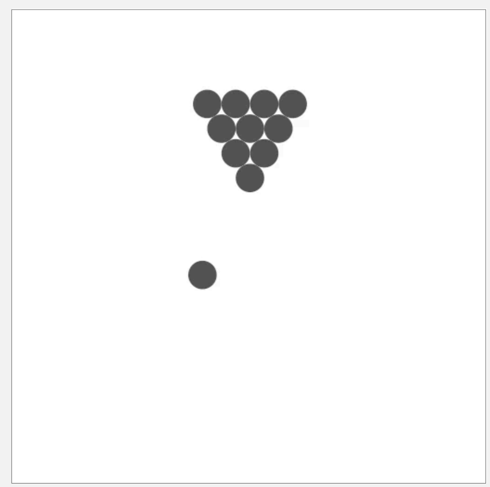
Particle collision simulation example 1

```
% java CollisionSystem 100
```



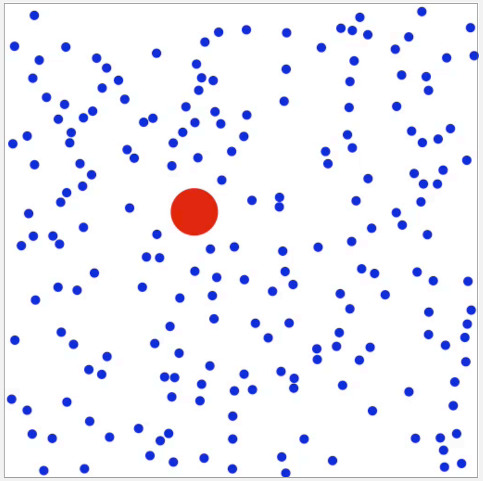
Particle collision simulation example 2

```
% java CollisionSystem < billiards.txt
```



Particle collision simulation example 3

```
% java CollisionSystem < brownian.txt
```



Particle collision simulation example 4

```
% java CollisionSystem < diffusion.txt
```