COLLECTIONS, IMPLEMENTATIONS, PRIORITY QUEUES

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

<table>
<thead>
<tr>
<th>operation</th>
<th>parameters</th>
<th>returns</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>contains</td>
<td>Item</td>
<td>boolean</td>
<td>checks if an item is in the list</td>
</tr>
<tr>
<td>add</td>
<td>Item</td>
<td></td>
<td>appends Item at end</td>
</tr>
<tr>
<td>add</td>
<td>index, Item</td>
<td></td>
<td>adds Item at position index</td>
</tr>
<tr>
<td>set</td>
<td>index, Item</td>
<td></td>
<td>replaces item at position index with Item</td>
</tr>
<tr>
<td>remove</td>
<td>index</td>
<td>boolean</td>
<td>removes item at index</td>
</tr>
<tr>
<td>remove</td>
<td>Item</td>
<td>boolean</td>
<td>removes Item if present in list</td>
</tr>
<tr>
<td>get</td>
<td>index</td>
<td>Item</td>
<td>returns item at index</td>
</tr>
</tbody>
</table>

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

```java
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.

Min PQ operations.

<table>
<thead>
<tr>
<th>operation</th>
<th>parameters</th>
<th>returns</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>Item</td>
<td></td>
<td>adds an item</td>
</tr>
<tr>
<td>min</td>
<td>Item</td>
<td></td>
<td>returns minimum Item</td>
</tr>
<tr>
<td>delMin</td>
<td>Item</td>
<td></td>
<td>deletes and returns minimum Item</td>
</tr>
</tbody>
</table>
Priority queue

<table>
<thead>
<tr>
<th>operation</th>
<th>parameters</th>
<th>returns</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>Item</td>
<td></td>
<td>adds an item</td>
</tr>
<tr>
<td>min</td>
<td>Item</td>
<td></td>
<td>returns minimum Item</td>
</tr>
<tr>
<td>delMin</td>
<td>Item</td>
<td></td>
<td>deletes and returns minimum Item</td>
</tr>
</tbody>
</table>

Actual algs4 class

```java
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

<table>
<thead>
<tr>
<th>operation</th>
<th>parameters</th>
<th>returns</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>Item</td>
<td></td>
<td>adds an item, only one copy may exist</td>
</tr>
<tr>
<td>contains</td>
<td>boolean</td>
<td></td>
<td>returns true if item is present</td>
</tr>
<tr>
<td>delete</td>
<td></td>
<td></td>
<td>deletes item</td>
</tr>
</tbody>
</table>

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

```java
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.
Symbol tables

<table>
<thead>
<tr>
<th>operation</th>
<th>parameters</th>
<th>returns</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>put</td>
<td>Key, Value</td>
<td></td>
<td>associates Value with Key</td>
</tr>
<tr>
<td>contains</td>
<td>Key</td>
<td>boolean</td>
<td>returns true if Key is present</td>
</tr>
<tr>
<td>get</td>
<td>Key</td>
<td>Value</td>
<td>returns Value associated with Key (if any)</td>
</tr>
<tr>
<td>delete</td>
<td>Key</td>
<td>Value</td>
<td>deletes Key and returns Value</td>
</tr>
</tbody>
</table>

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

```java
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).

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

- 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.

<table>
<thead>
<tr>
<th>Collection</th>
<th>Java Implementations</th>
<th>algs4 Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>List</td>
<td>LinkedList, ArrayList, Stack (oops)</td>
<td>None</td>
</tr>
<tr>
<td>MinPQ, MaxPQ</td>
<td>PriorityQueue</td>
<td>MinPQ, MaxPQ</td>
</tr>
<tr>
<td>Set</td>
<td>TreeSet, HashSet, CopyOnWriteArraySet, ...</td>
<td>SET (note: ordered)</td>
</tr>
<tr>
<td>Symbol Table</td>
<td>TreeMap, HashMap, ConcurrentHashMap, ...</td>
<td>RedBlackBST, SeparateChainingHashST, LinearProbingHashST, ...</td>
</tr>
</tbody>
</table>
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.

```java
public class MaxPQ<Key extends Comparable<Key>> {
    // create an empty priority queue
    MaxPQ()
    MaxPQ(Key[] a)

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

    // return and remove the largest key
    Key delMax()

    // is the priority queue empty?
    boolean isEmpty()

    // return the largest key
    Key max()

    // number of entries in the priority queue
    int size()
}
```

Key must be Comparable (bounded type parameter)
Priority queue applications

Generalizes: stack, queue, randomized queue.

- 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]
- 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]

See optional slides / Coursera lecture.

Assignment 4.
### 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**

<table>
<thead>
<tr>
<th>implementation</th>
<th>time</th>
<th>space</th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>$N \log N$</td>
<td>$N$</td>
</tr>
<tr>
<td>elementary PQ</td>
<td>$M$</td>
<td>$M$</td>
</tr>
<tr>
<td>binary heap</td>
<td>$N \log M$</td>
<td>$M$</td>
</tr>
<tr>
<td>best in theory</td>
<td>$N$</td>
<td>$M$</td>
</tr>
</tbody>
</table>
## Priority queue: unordered and ordered array implementation

<table>
<thead>
<tr>
<th>operation</th>
<th>argument</th>
<th>return</th>
<th>size</th>
<th>contents (unordered)</th>
<th>contents (ordered)</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>1</td>
<td>P</td>
<td>P</td>
</tr>
<tr>
<td>insert</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>P Q</td>
<td>P Q</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>3</td>
<td>P Q E</td>
<td>E P Q</td>
</tr>
<tr>
<td>remove max</td>
<td>Q</td>
<td></td>
<td>2</td>
<td>E P</td>
<td>E P</td>
</tr>
<tr>
<td>insert</td>
<td>X</td>
<td></td>
<td>3</td>
<td>E P X</td>
<td>E P X</td>
</tr>
<tr>
<td>insert</td>
<td>A</td>
<td></td>
<td>4</td>
<td>A E P X</td>
<td>A E P X</td>
</tr>
<tr>
<td>insert</td>
<td>M</td>
<td></td>
<td>5</td>
<td>A E M P X</td>
<td>A E M P X</td>
</tr>
<tr>
<td>remove max</td>
<td>X</td>
<td></td>
<td>4</td>
<td>A E M P</td>
<td>A E M P</td>
</tr>
<tr>
<td>insert</td>
<td>P</td>
<td></td>
<td>5</td>
<td>A E M P</td>
<td>A E M P P</td>
</tr>
<tr>
<td>insert</td>
<td>L</td>
<td></td>
<td>6</td>
<td>A E L M P P</td>
<td>A E L M P P</td>
</tr>
<tr>
<td>insert</td>
<td>E</td>
<td></td>
<td>7</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
<tr>
<td>remove max</td>
<td>P</td>
<td></td>
<td>6</td>
<td>A E E L M P P</td>
<td>A E E L M P P</td>
</tr>
</tbody>
</table>

A sequence of operations on a priority queue
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]; // should null out entry to prevent loitering
    }
}
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>
</tbody>
</table>

**goal**: \( \log N \) \( \log N \) \( \log N \)

order of growth of running time for priority queue with N items
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
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)](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

Hyphaene Compressa - Doum Palm

© Shlomit Pinter
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.

```
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.

**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 \(2 \log N\) compares.

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

```java
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;
    }

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;
}
}
```
**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>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>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 \dagger)</td>
<td>1</td>
</tr>
<tr>
<td>Brodal queue</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>

\(\dagger\) amortized

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

- unordered array: \(O(1)\) for insert, \(O(N)\) for del max, \(O(N)\) for max
- ordered array: \(O(N)\) for insert, \(O(1)\) for del max, \(O(1)\) for max
- binary heap: \(O(\log N)\) for insert, \(O(\log N)\) for del max, \(O(1)\) for max
- d-ary heap: \(O(\log_d N)\) for insert, \(O(d \log_d N)\) for del max, \(O(1)\) for max
- Fibonacci: \(O(1)\) for insert, \(O(\log N \dagger)\) for del max, \(O(1)\) for max
- Brodal queue: \(O(1)\) for insert, \(O(\log N)\) for del max, \(O(1)\) for max
- impossible: \(O(1)\) for insert, \(O(1)\) for del max, \(O(1)\) for max

**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.

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

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)
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.

```java
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 “◊” 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
Heapsort demo

Sortdown. Repeatedly delete the largest remaining item.

array in sorted order
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).

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

starting point (arbitrary order)

result (heap-ordered)
Heapsort: sortdown

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

```plaintext
while (N > 1)
{
    exch(a, 1, N--);
    sink(a, 1, N);
}
```
Heapsort: Java implementation

```java
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

Heapsort trace (array contents just after each sink)
Heapsort animation

50 random items

http://www.sorting-algorithms.com/heap-sort
Proposition. Heap construction uses $\leq 2N$ compares and exchanges.

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

algorithm be improved to $\sim 1N \lg N$

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.
Heapsort summary

The good news:
• Heap sort: In place and theoretically fast (not in place)

<table>
<thead>
<tr>
<th>Mergesort</th>
<th>Quicksort</th>
<th>Heapsort</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image1.jpg" alt="Mergesort" /></td>
<td><img src="image2.jpg" alt="Quicksort" /></td>
<td><img src="image3.jpg" alt="Heapsort" /></td>
</tr>
</tbody>
</table>

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!

Which of these is faster?

A. \[
\text{sum}=0 \\
\text{for } (i = 0 \text{ to } \text{size}) \\
\quad \text{for } (j = 0 \text{ to } \text{size}) \\
\quad \quad \text{sum } += \text{array}[i][j]
\]

B. \[
\text{sum}=0 \\
\text{for } (i = 0 \text{ to } \text{size}) \\
\quad \text{for } (j = 0 \text{ to } \text{size}) \\
\quad \quad \text{sum } += \text{array}[j][i]
\]

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

<table>
<thead>
<tr>
<th>inplace?</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>½ (N^2)</td>
<td>½ (N^2)</td>
<td>½ (N^2)</td>
<td>N exchanges</td>
</tr>
<tr>
<td>insertion</td>
<td>x</td>
<td>x</td>
<td>½ (N^2)</td>
<td>¼ (N^2)</td>
<td>N use for small N or partially ordered</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>½ (N^2)</td>
<td>2 (N \ln N)</td>
<td>(N \lg N)</td>
<td>N (\log N) probabilistic guarantee, fastest in practice</td>
</tr>
<tr>
<td>3-way quick</td>
<td>x</td>
<td>½ (N^2)</td>
<td>2 (N \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 \lg N)</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>N (\log N) guarantee, stable</td>
</tr>
<tr>
<td>heap</td>
<td>x</td>
<td>2 (N \lg N)</td>
<td>2 (N \lg N)</td>
<td>(N \lg N)</td>
<td>N (\log N) guarantee, in-place</td>
</tr>
<tr>
<td>???</td>
<td>x</td>
<td>x</td>
<td>(N \lg N)</td>
<td>(N \lg N)</td>
<td>(N \lg N) holy sorting grail</td>
</tr>
</tbody>
</table>
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).

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]
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: $4 \times 2 + 2 \times 4 + 6 = 22$ compares
Sink-based (bottom up) heapification

Total Heap Construction Cost
- For h=1: $C_1 = 2$
- For h=2: $C_2 = 2C_1 + 2 \times 2$
- 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)
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.

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

% java BouncingBalls 100

main simulation loop
Warmup: bouncing balls

public class Ball
{
    private double rx, ry;    // position
    private double vx, vy;    // velocity
    private final double radius; // radius
    public Ball(...) // radius
    { /* 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); }
}

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.

- \( \sim \frac{N^2}{2} \) 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?

Prediction (at time $t$)

$dt \equiv \text{time to hit wall}$
$\equiv \text{distance/velocity}$
$\equiv (1 - s - rx)/vx$

Resolution (at time $t + dt$)

velocity after collision $= (-vx, vy)$
position after collision $= (1 - s, ry + vydt)$

Predicting and resolving a particle-wall collision
Collision prediction.

- Particle $i$: radius $s_i$, position $(r_{xi}, r_{yi})$, velocity $(v_{xi}, v_{yi})$.
- Particle $j$: radius $s_j$, position $(r_{xj}, r_{yj})$, velocity $(v_{xj}, v_{yj})$.
- Will particles $i$ and $j$ collide? If so, when?
Collision prediction.

- Particle $i$: radius $s_i$, position $(r_{xi}, r_{yi})$, velocity $(v_{xi}, v_{yi})$.
- Particle $j$: radius $s_j$, position $(r_{xj}, r_{yj})$, velocity $(v_{xj}, v_{yj})$.
- Will particles $i$ and $j$ collide? If so, when?

\[
\Delta t = \begin{cases} 
\infty & \text{if } \Delta v \cdot \Delta r \geq 0 \\
\infty & \text{if } d < 0 \\
- \frac{\Delta v \cdot \Delta r + \sqrt{d}}{\Delta v \cdot \Delta v} & \text{otherwise}
\end{cases}
\]

\[
d = (\Delta v \cdot \Delta r)^2 - (\Delta v \cdot \Delta v) (\Delta r \cdot \Delta r - \sigma^2) \quad \sigma = \sigma_i + \sigma_j
\]

\[
\Delta v = (\Delta v_x, \Delta v_y) = (v_{xi} - v_{xj}, v_{yi} - v_{yj}) \quad \Delta v \cdot \Delta v = (\Delta v_x)^2 + (\Delta v_y)^2
\]

\[
\Delta r = (\Delta r_x, \Delta r_y) = (r_{xi} - r_{xj}, r_{yi} - r_{yj}) \quad \Delta r \cdot \Delta r = (\Delta r_x)^2 + (\Delta r_y)^2
\]

\[
\Delta v \cdot \Delta r = (\Delta v_x)(\Delta r_x) + (\Delta v_y)(\Delta r_y)
\]

Important note: This is high-school physics, so we won’t be testing you on it!
Collision resolution. When two particles collide, how does velocity change?

\[
\begin{align*}
vx_i' &= vx_i + Jx / m_i \\
vy_i' &= vy_i + Jy / m_i \\
vx_j' &= vx_j - Jx / m_j \\
vy_j' &= vy_j - Jy / m_j
\end{align*}
\]

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!
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

```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;
}
```

```java
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.

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.

"potential" since collision may not happen if some other collision intervenes
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()
    {   }
}
Collision system implementation: skeleton

```java
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

```java
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