Algorithms, Princeton University

Guidelines

1. Compile errors are more welcomed, than runtime errors.
2. Avoid casting (create new types) in codes.

3. Use theory as a guide
    a. Don’t try to design sorting algorithms that guarantees N logN/2 compares
    b. Design sorting algorithm that is both time and space optimal
4.

Interview Questions

1. **Queue with two stacks.** Implement a queue with two stacks so that each queue operations takes a constant amortized(緩衝) number of stack operations.

   Ans: If you push elements onto a stack and then pop them all, they appear in reverse order. If you repeat this process they’re now back in order.

2. **Stack with max.** Create a data structure that efficiently supports the stack operations and also a return the maximum operation. Assume the elements are real numbers so that you can compare them.

   Ans: Use two stacks, one stores all of the items and a second stack to store to maximum.

3. **Java Generics.** Explain why Java prohibits generic array creation.

   Ans: Java arrays are covariant(協變) but Java generics are not: that is String[] is a subtype of Object[], but Stack<String> is not a subtype of Stack<Object>.

Part I

Stacks

Examine the item most recently added. (Last in first out)

---

public class StackOfString

Create an empty stack: StackOfStrings()

Insert an new string onto stack: void push(String item)

Remove and return the string most recently added: String pop()

Check if the stack is empty: boolean isEmpty()

```
public static void main(String[] args) {
    //Create a new stack
    StackOfStrings stack = new StackOfStrings();
    while (!StdIn.isEmpty()){
        String s = StdIn.readString();
        if (s.equals("-") StdOut.print(stack.pop());
        else stack.push();
        }
}
```

Input: to be or not to - be - - that - - - is

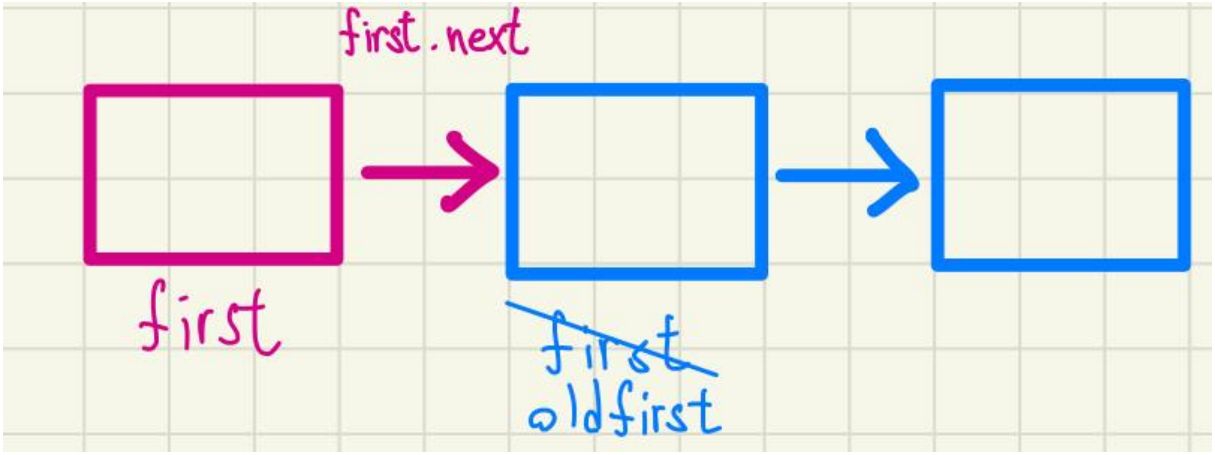
Output: to be not that or be

Preimplement

```
private Node first = null;
private class Node{
    String item;
    Node next;
}
```

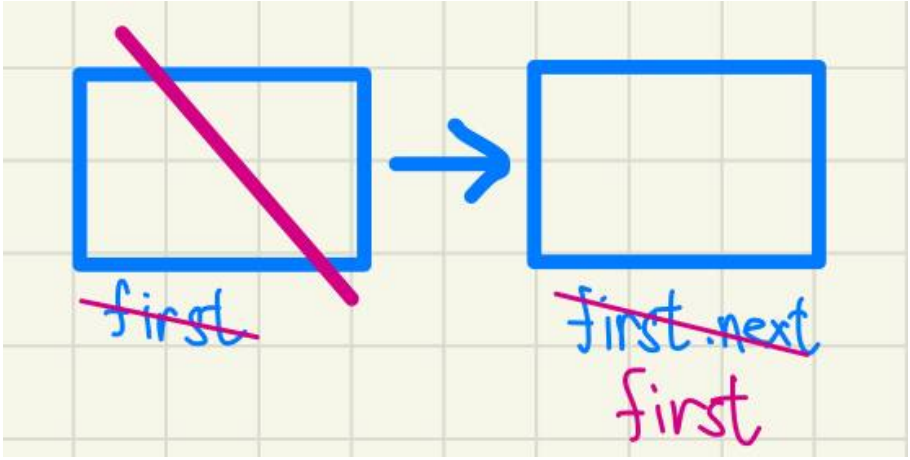
Implement push and pop by linked list:

```
public void push(String item){
    Node oldfirst = first;
    first = new Node();
    first.item = item;
    first.next = oldfirst;
}
```




```
public String pop(){
    String item = first.item;
    first = first.next;
    return item
}
/*
```

```
after popping an item, we can avoid loitering by letting array[N]=null,
so that, the memory is released.
*/
```



Most of the time, we don;t know the size of the array we need. How are we going to resize the arrays we need?

1. Increase the size of array when push(), and decrease the size of the array when pop()

   Too expensive, everytime we do push() or pop(), we have to copy all items to a new array. (N^2) = 1+2+3+…+N

2. Repeated doubling

   If an array is full, create a new array of twice the size, and copy it.

   If an array is one-quarter full, halve size the array. (The array is always between 25%~100% full)

```
public ResizingArrayStackOfStrings(){
    s = new String[1];
}

public void push (String item) {
    if (N == s.length) resize(2 * s.length):
    s[N++] = item;
}

public String pop(){
    String item = s[--N];
    //to avoid loitering
    s[N] = null;
    if (N > 0 && N == s.length/4){
        resize(s.length/2);
    }
    return item;
}

private void resize(int capacity){
    String[] copy = new String[capacity];
    for (int i = 0; i<N; i++){
        copy[i] = s[i];
    }
```

```
    s = copy;
}
```

(N) = 2+4+8+…+N = 3*N

Queue

Examine the item least recently added. (First in first out)

Last (Linear time to enqueue) First (Const time to dequeue)

public class of QueueOfStrings

Create an empty queue: QueueOfStrings()

Insert a new string onto queue: void enqueue(String item)

Remove and return the string least recently added: String dequeue()

Check if the queue is empty: boolean isEmpty()

```
public class LinkedQueueOfStrings{
    private Node first, last;

    private class Node{
        String item;
        Node item;
    }

    public boolean isEmpty(){
        return first == null;
    }

    public void enqueue(String item){
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else oldlast.next = last;

    public String dequeue() {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
}
```

Generics

We have implemented StackOfStrings, we also want StackOfURLs, StackOfInts, …

1. Implement a seperate stack for each type of data.
2. Implement a stack with items of type object

   Casting is required in client, and it is error-prone (when type mismatches).
3. Java generics

   Avoid casting in client, discover an error when compile.

linked-list implementation

```
public class Stack<Item>{
    private Node first = null;

    private class Node{
        Item item;
        Node next;
    }

    public boolean isEmpty(){
        return first == null;
    }

    public void push(Item item){
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }

    public Item pop(){
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

Array implementation

```
public class FixedCapacityStack<Item>{
    private Item[] s;
    private int N = 0;

    public FixedCapacityStack(int capacity){
        s = (Item[]) new Object[capacity];
    }

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

    public void push(Item item){
    s[N++] = item;
    }

    public Item pop(){
    return s[--N];
    }
}
```

Iteration

Support iteration over stack items by client, without revealing the internal representation of the stack.

```
//Iterable interface, Iterable means that there is method to return an Iterator.
public interface Iterable<Item>{
    Iterator<Item> iterator();
}

//Iterator interface, Iterator has methods hasNext() and next()
public interface Iterator<Item>{
    boolean hasNext();
    Item next();
}
```

```
//Shorthand
for (String s : stack) StdOut.println(s);
//Longhand
Iterator<String> i = stack.iterator();
while (i.hasNext()) {
    String s = i.next();
    StdOut.println(s);
}
```

Linked-list implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item>{

    public Iterator<Item> iterator(){
        return new ListIterator();
    }

    private class ListIterator implements Iterator<Item>{
        private Node current = first;

        public boolean hasNext(){
            return current != null;
        }

        public void remove(){
        /*not supported*/
        }

        public Item next(){
            Item item = current.next;
            current = current.next;
            return item;
        }
    }
}
```

Array implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item>{
    public Iterator<Item> iterator(){
        return new ReverseArrayIterator();
    }

    private class ReversArrayIterator implements Itera
        private int i = N;

        public boolean hasNext(){
            return i>0;
        }

        public void remove(){
            /*not supported*/
        }

        public Item next(){
            return s[--i];
        }
    }
}
```

Applications

Dijkstra’s two-stack algorithm

1. Value: push onto the value stack.
2. Operator: push on to the operator stack.
3. Left parenthesis: ignore.
4. Right parenthesis: pop operator and two values, push the result of applying that operator to those value onto the operand stack.

(1 + ((2+3) * (4 * 5)))

```
public class Evaluate {
    public static void main(String[] args) {
```

```
        Stack<String> ops = new Stack<String>();
        Stack<Double> vals = new Stack<Double>();
        while (!StdIn.isEmpty()) {
            String s = StdIN.readString();
            if (s.equals("(") ;
            else if (s.equals("+") ops.push(s);
            else if (s.equals("*") ops.push(s);
            else if (s.equals(")") {
                String op = ops.pop();
                if (op.equals("+")) vals.push(vals.pop() + vals.pop());
                else if (op.equals("*") vals.push(vals.pop() * vals.pop());
            }
            else vals.push(Double.parseDouble(s));
        }
        StdOut.println(vals.pop());
    }
}
```

Sorting

Goal, sort any type of input array. ⇒ Callback = reference to executable code.

1. Client passes array of objects to sort() function.
2. The sort() function calls back object's compareTo() method as needed.

RoadMaps:

1. Client

```
import java.io.File;
public class FileSorter {
    public static void main(String[] args) {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++) {
            StdOut.println(files[i].getName());
        }
    }
}
```

2. Comparable interface (built in to java)

```
public interface Comparable<Item> {
    public int compareTo(Item that);
}
```

3. Object implementation

```
public class File implements Comparable<File> {
    ...
    public int compareTo(File b) {
        ...
        return -1;
```

```
        ...
        return +1;
        ...
        return 0;
    }
}
```

4. Sort implementation

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

Total Order:

1. Antisymemetry: if V ≤ W and W ≤ V, then V=W.
2. Transitivity: if V ≤ W and W ≤ X, then V ≤ X.
3. Totality: if either V≤ W or W ≤ V or both.

Selection sort

Start with the first index, the find the min of the rest of the card, and compare. If it is smaller, swap the two cards, then move on to the next card.

Runtime: Quadratic, It doesn't matter whether the array is sorted or not.

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

private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}

private static void exch(Comparable[] a, int i, int j){
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

Insertion sort

In iteration i, swap a[i] with each larger entry to its left.

Runtime: Quadratic/4

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

private static boolean less(Comparable v, Comparable w) {
    return v.compareTo(w) < 0;
}

private static void exch(Comparable[] a, int i, int j){
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

Shell sort

Move entries more than one position at a time by h-sorting the array.

Big increments (big h) ⇒ small subarrays, Small increments (small h) ⇒ nearly in order.

Prove: A g-sorted array remains g-sorted after h-sorting it.

However which increments should we choose?

1. Power of two? ⇒ Noooooo
2. Powers of two minus one? ⇒ Maybe
3. 3x + 1? ⇒ Easy to compute
4. Sedgewick (1, 5, 19, 41, 109, 209,…)? ⇒ Good, tough to beat in emperical studies.

```
public class Shell {
    public static void sort(Comparable[] a) {
        int N = a.length;

        int h = 1;
        while (h < N/3) h - h * 3 + 1;
        while (h >= 1) {
            for (int i = h; i < N; i++) {
                for (int j = i; j >= h && less(a[j], a[j-h]); j -= h) exch(a, k, j-h);
            }
            h = h/3;
        }
    }
    private static boolean less(Comparable v, Comparable w) {
    }
    private static void exch(Comparable[] a, int i, int j) {
```

```
    }
}
```

Shuffle sort

Generate a random real number for each array entry, then sort the array.

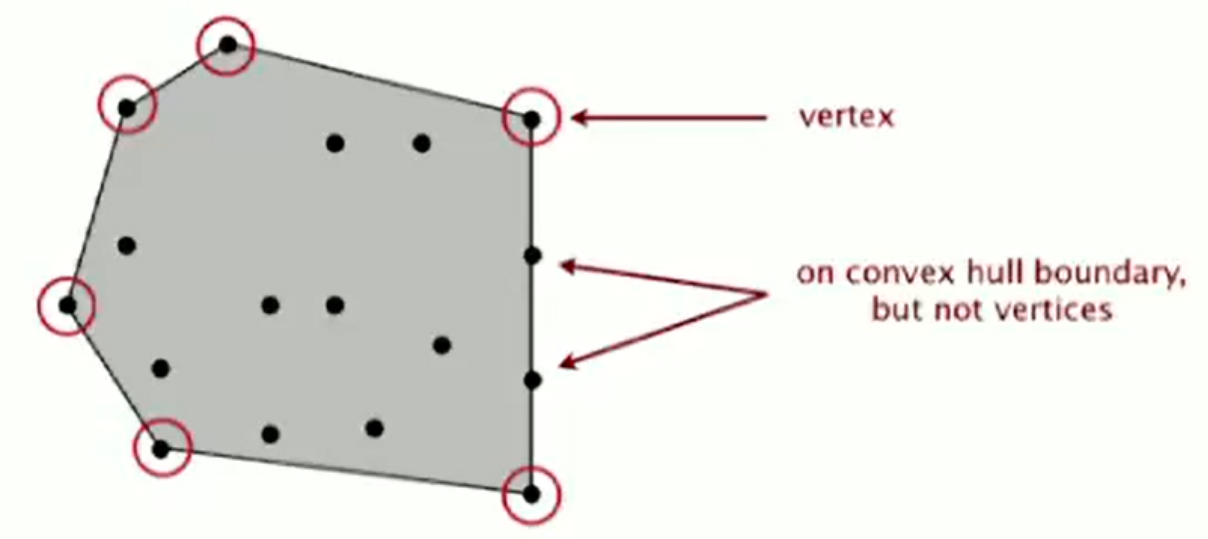
Knuth shuffle (It takes linear time to shuffle)

In iteration i, pick integer r between 0 and i uniformly at random, then swap a[i] and a[r].

```
public static StdRandom {
    public static void shuffle (Object[] a) {
        int N = a.length;
        for (int i = 0; i < N; i++) {
            int r = StdRandom.uniform(i + 1);
            exch(a, i, r);
        }
    }
}
```

Convex hull

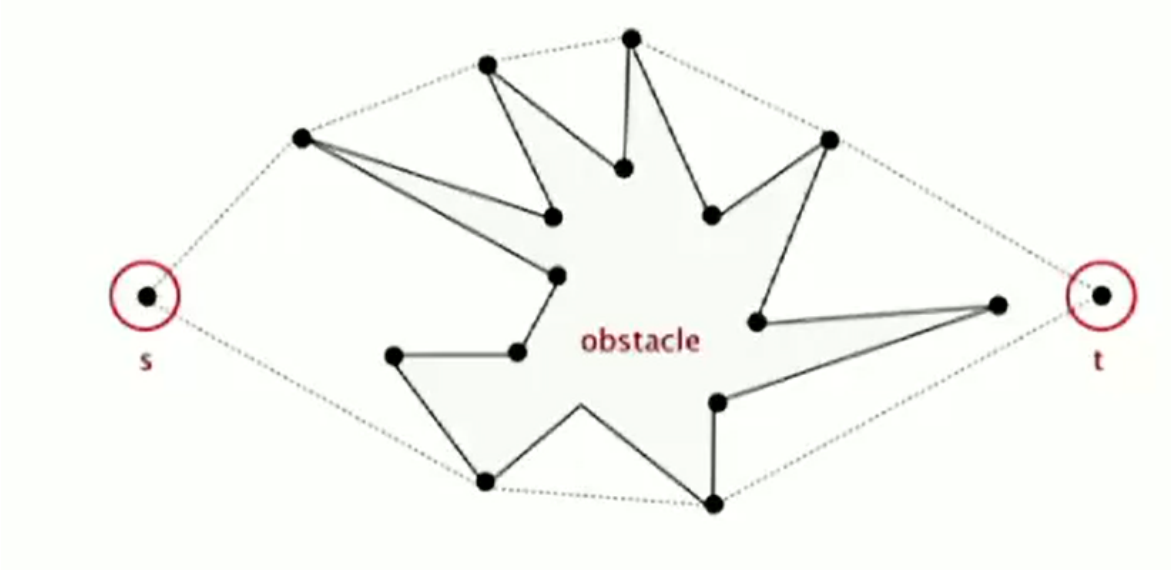
The convex hull of a set of N points is the smalllest perimeter fence enclosing the points.



Usage:

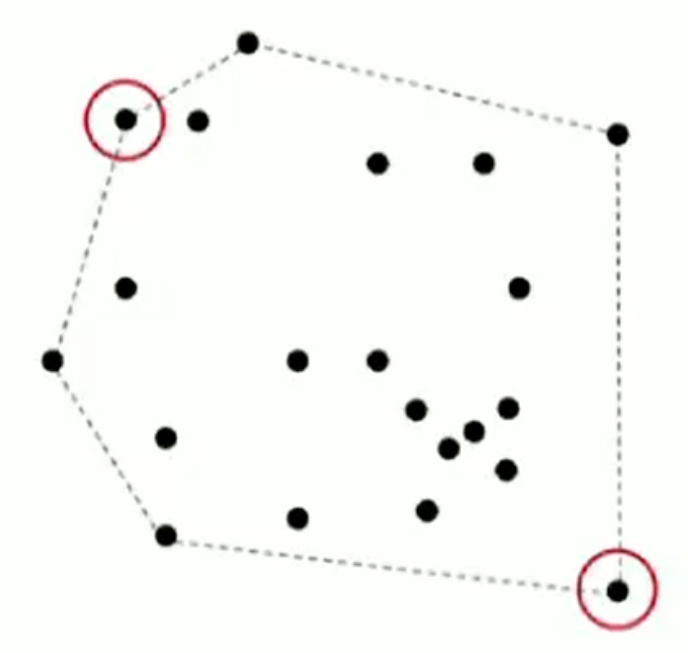
Robot motion planning

The shortest path is either straight or the polygonal chains of convex hull.



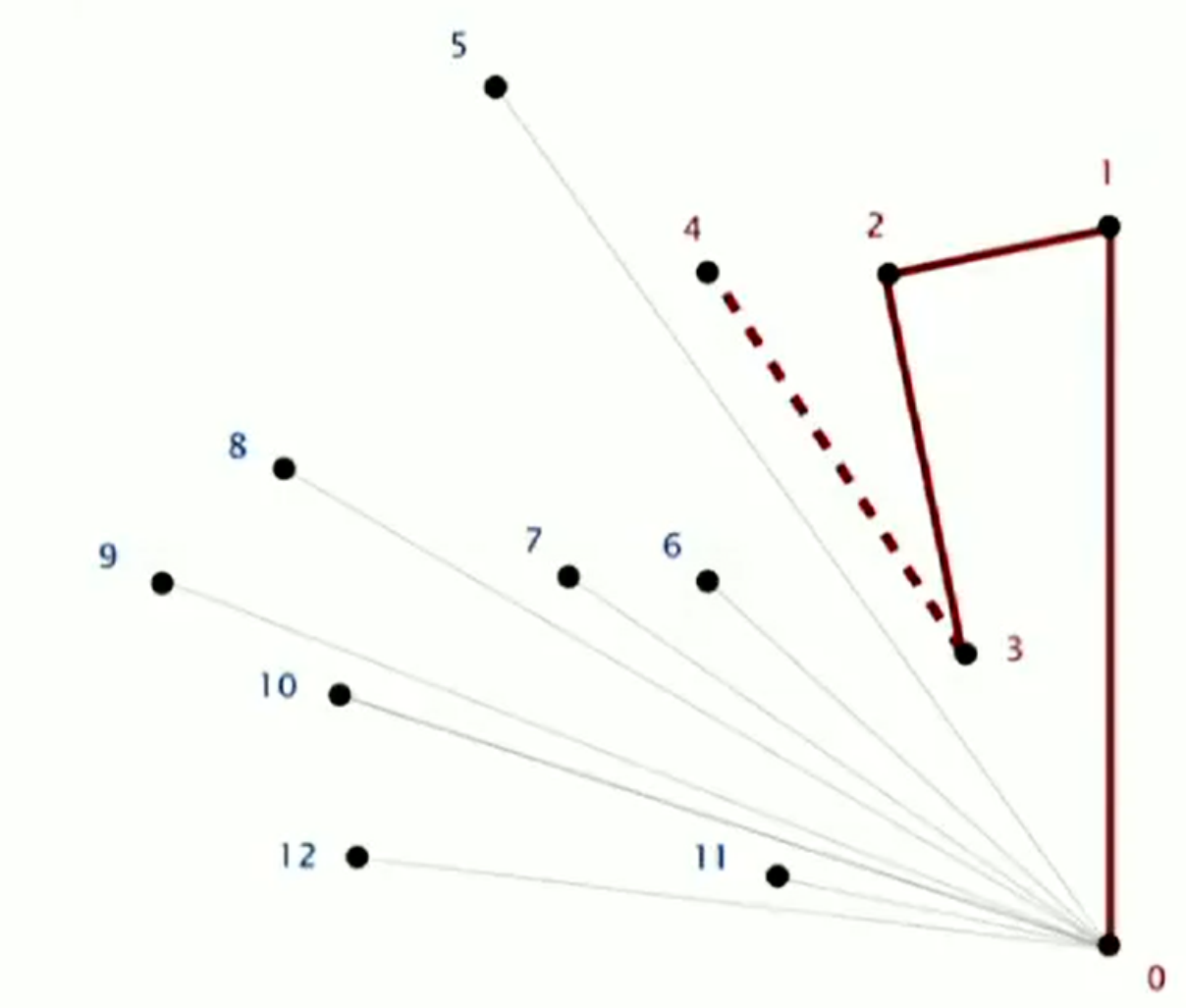
Farthest pair problem

Given N points on the plane, find a pair of points with the largest Euclidean distance between them.



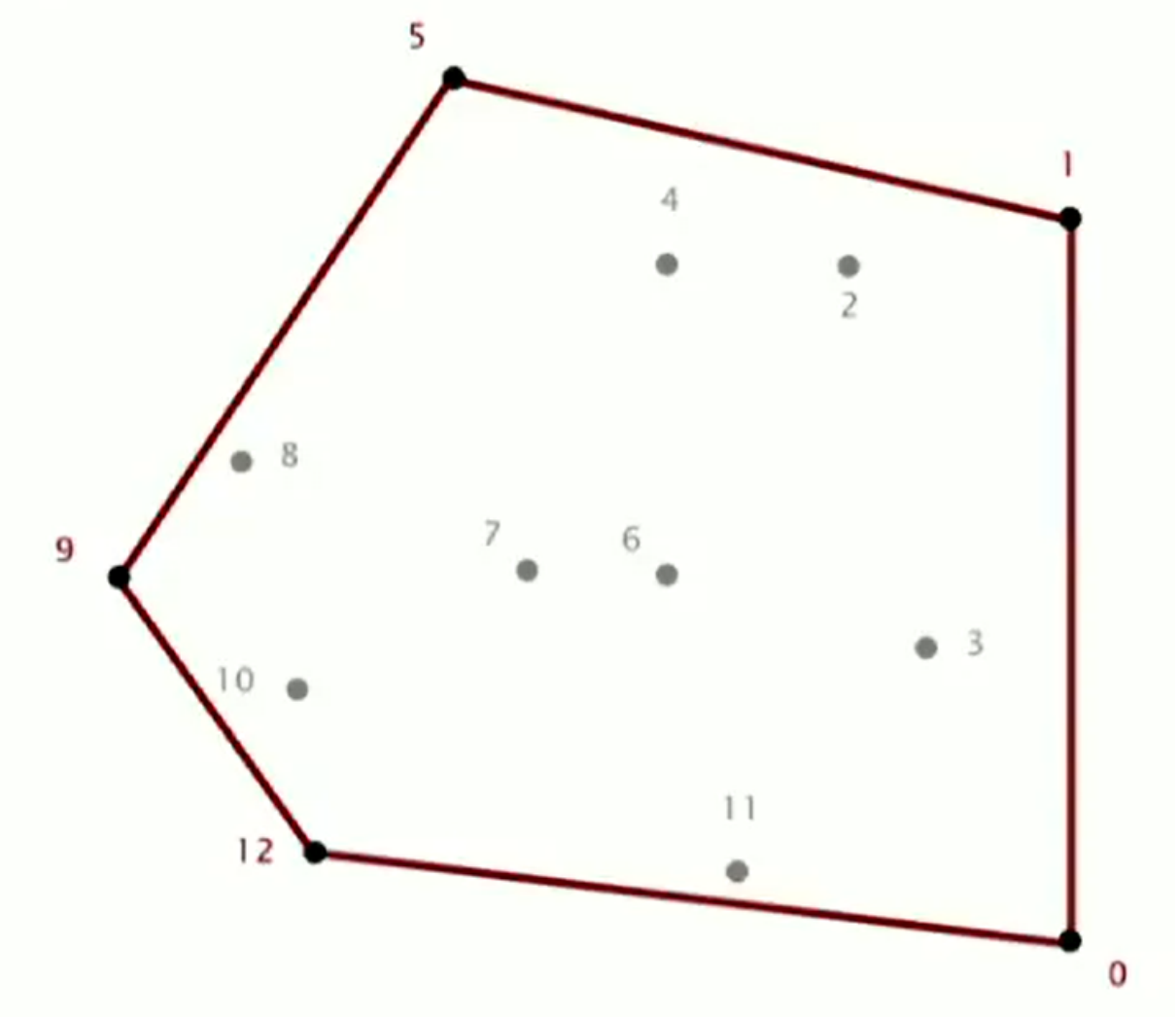
Graham scan demo

1. Choose point p with the smallest y-coordinate.
2. Sort points by polar angle with p.
3. Consider points in order, discard unless it create a ccw turn.



3 → 4 is not a ccw turn, so we discard 3.

After going through all the points, we get the convex hull.




```
Stack<Point2D> hull = new Stack<Point>();

Arrays.sort(p, Point2D.Y_ORDER); //Find out the lowest y-coordinate
Arrays.sort(p, p[0].BY_POLAR_ORDER); //Sort by polar angle with respect to p[0]

hull.push(p[0]);
hull.push(p[1]);

for (int i = 2; i < N; i++) {
    Point2D top = hull.pop();
    //compare the top two points on the hull to check if it is a ccw turn.
    //Point2D.ccw(hull.peek(), top, p[i]) > 0) is ccw
    while (Point2D.ccw(hull.peek(), top, p[i]) <= 0)
        top = hull.pop();
    hull.push(top);
    hull.push(p[i]);
}
```

Check if it is CCW

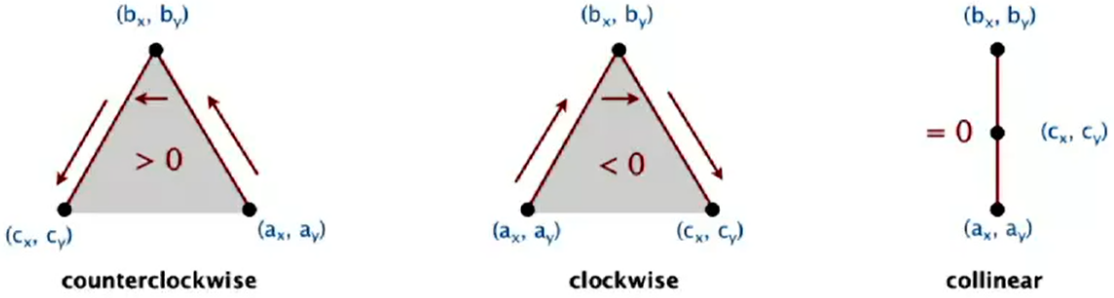
CCW. Given three points a, b, and c, is $a \rightarrow b \rightarrow c$ a counterclockwise turn?

- Determinant (or cross product) gives 2x signed area of planar triangle.

$$2 \times Area(a,b,c) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = (b_x - a_x)(c_y - a_y) - (b_y - a_y)(c_x - a_x)$$

(b − a) × (c − a)

- If signed area > 0, then $a \rightarrow b \rightarrow c$ is counterclockwise.
- If signed area < 0, then $a \rightarrow b \rightarrow c$ is clockwise.
- If signed area $= 0$, then $a \rightarrow b \rightarrow c$ are collinear.



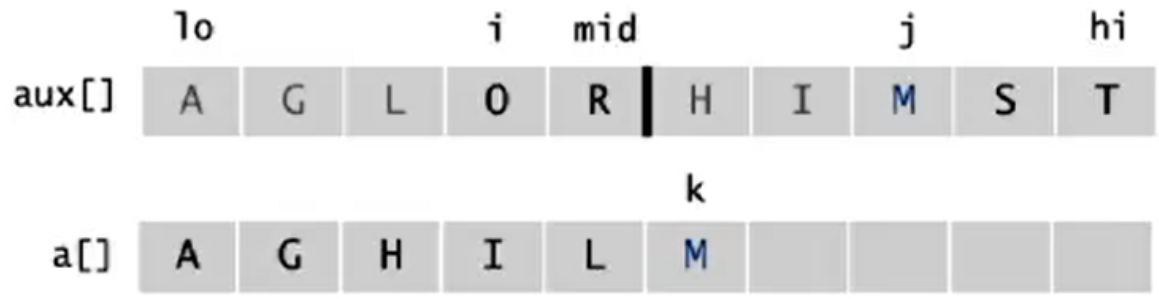

```
public class Point2D {
    private final double x;
    private final double y;

    public Point2D (double x, double y) {
        this.x = x;
        this.y = y;
    }
    ...
    public static int ccw (Point2D a, Point2D b, Point2D c) {
        double area2 = (b.x - a.x)*(c.y - a.y) - (b.y - a.y)*(c.x - a.x);
        if (area2 < 0) return -1; //clockwise
        else if (area2 > 0) return 1; //ccw
        else return 0; //collinear
    }
}
```

Mergesort

Divide array into two halves, then recursively merge each half, last merge two halves.

```
public class Merge {
    private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
        //precondition a[lo....mid] sorted
        assert isSorted(a, lo, mid);
```

```
        //precondition a[mid+1..hi] sorted
        assert isSorted(a, mid+1, hi);

        //copy the array
        for (int k = lo; k <= hi; k++)
            aux[k] = a[k];

        //merge
        int i = lo, j = mid+1;
        for (int k = lo; k <= hi; k++) {
            if (i > mid) a[k] = aux[j++];
            else if (j > hi) a[k] = aux[i++];
            else if (less(aux[j], aux[i])) a[k] = aux[j++];
            else a[k] = aux[i++];
        }
        //postcondition a[lo...hi] sorted
        assert isSorted(a, lo, hi);
    }

    private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
        if (hi <= lo) return;
        int mid = lo + (hi - lo) / 2;
        sort (a, aux, lo, mid);
        sort (a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }

    public static void sort(Comparable[] a) {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length-1);
    }
}
```



Improvement for mergesort:

1. Use insertion sort for small subarrays
    a. Mergesort have too much overhead for tiny subarrays.
2. Stop if already sorted
    a. Check if the biggest item in first half ≤ smallest item in second half
3. Eliminate the copy to the auxiliary array
    a. Save time but not space

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi) {
    //merge
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        if (i > mid) aux[k] = a[j++];
        else if (j > hi) aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else aux[k] = a[i++];
    }
}

private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort (aux, a, lo, mid);
    sort (aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Assertions

Statements to test assumptions about your program.

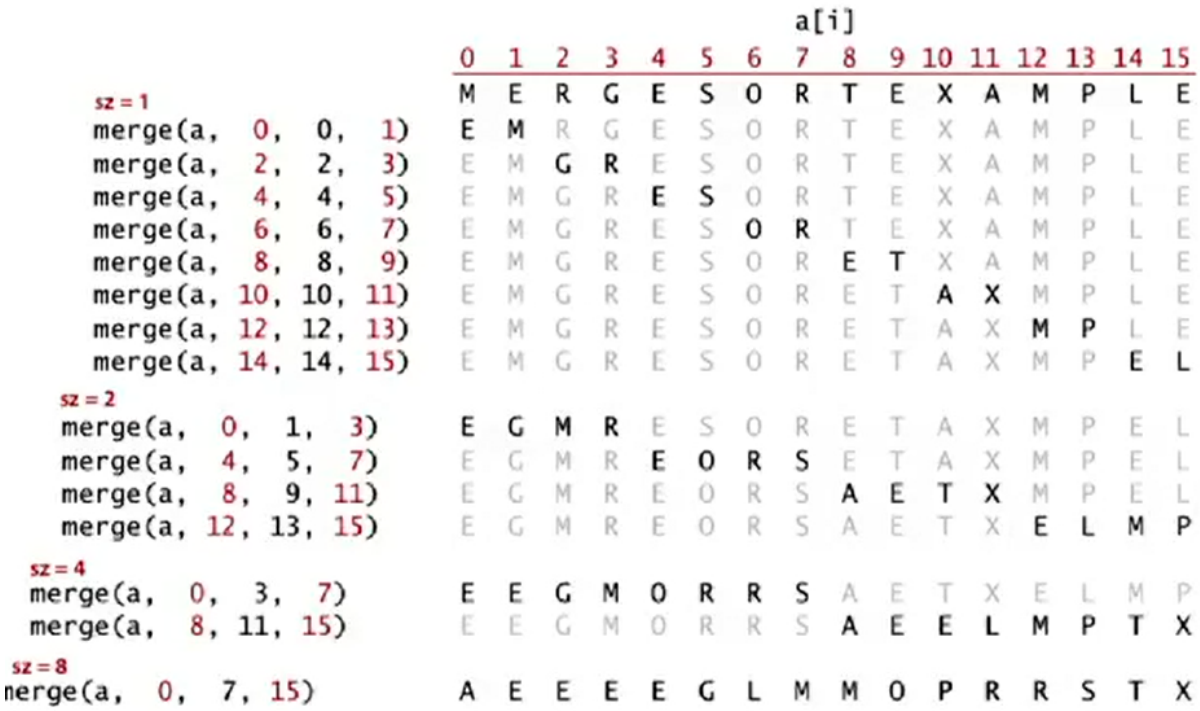
Java assert statements throw exceptions unless boolean condition is true.

Can enable or disable at runtime ⇒ No cost in production code.

```
java -ea MyProgram //assertion enable
java -da MyProgram //assertion disable
```

Bottom-up Mergesort

Pass through array, merging subarrays of size 1, then repeat for subarrays of size 2, 4, 6, 8, ….




```
public class MergeBU {
    private static Comparable[] aux;

    private static void merge(Comparable[] a, int lo, int mid, int hi) {
    /* as before*/
    }

    public static void sort(Comparable[] a) {
        int N = a.length;
        aux = new Comparable[N];

        for(int sz = 1; sz < N; sz = sz+sz) {
            for(int lo = 0; lo < N-sz; lo += sz+sz) {
                merge(a, lo, lo+sz-1, Math.min(lo+sz+sz-1, N-1));
            }
        }
    }
}
```

Complexity of sorting

1. Computational complexity

   Framework to study efficiency of algorithms for solving a particular problem X.

2. Model of computation

   Allowable computation

3. Cost model

Operation counts

4. Upper bound

   Cost guarantee provided by some algorithm for X

5. Lower bound

   Proven limit on cost guarantee of all algorithms for X

6. Optimal algorithm

   Algorithm with best possible cost guarantee for X

Example:

- Model of computation: Decision tree
- Cost model: #Compares
- Upper bound: ~N logN from mergesort
- Lower bound: ~N logN
- Optimal algorithm: mergesort

Comparators

Comparator interface: sort using an alternate order.

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

private static boolean less(Comparator c, Object v, Object w) {
    return c.compare(v, w) < 0;
}

private static void exch(Object[] a, int i, int j) {
    Object swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

To implement a comparator

```
public class Student {
    public static final Comparator<Student> BY_NAME = new ByName();
    public static final Comparator<Student> BY_SECTION = new BySection();
    private final String name;
    private final int section;

    private static class ByName implements Comparator<Student> {
        public int compare(Student v, Student w) {
            return v.name.compareTo(w.name);
        }
```

```
    }

    private static class BySection implements Comparator<Student> {
        public int compare(Student v, Student w) {
            return v.section - w.section;
        }
    }
}
```

Comparator for 2D point: Polar order

```
public class Point2D {
    public final Comparator<Point2D> POLAR_ORDER = new PolarOrder();
    private final double x, y;

    private static int ccw(Point2D a, Point2D b, Point2D c) {
        /* as in previous lecture*/
    }

    private class PolarOrder implements Comparator<Point2D> {
        public int compare(Point2D q1, Point2D q2) {
            double dy1 = q1.y - y;
            double dy2 = q2.y - y;

            if (dy1 == 0 && dy2 == 0) {/*horizontal*/}
            else if (dy1 >= 0 && dy2 < 0) return -1; // q1 above p, p above q2, CW
            else if (dy2 >= 0 && dy1 < 0) return +1; // q2 above p, p above q1, CCW
            else return -ccw(Point2D.this, q1, q2);  // both above or below p
        }
    }
}
```

Stability

A stable sort preserves the relative order of items with equal keys.

Insertion sort and mergesort are stable, but not selection sort or shellsort.

Proposition: Insertion sort is stable

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

Proof: Equal items never move pass each other.

Selection sort and shellsort is not stable due to they all have long distance changes.

If the merging operation is stable, then mergesort is stable.

Quicksort

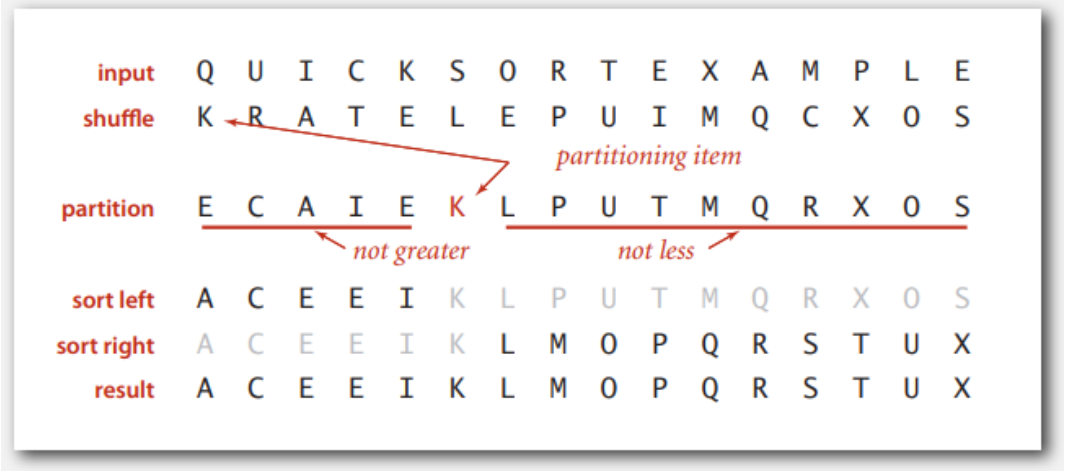
1. Shuffle the array.
2. Partition so that, for some j, entry[j] is in place, and no larger entry to the left of j, and no smaller entry to the right of j.
   a. Scan i from the left to the right as long as (a[i] < a[lo]).
   b. Scan j from the right to the left as long as (a[lo] < a[j]).
   c. Exchange a[i] with a[j].

   Repeat this until i and j meets.

   Then exchange a[j] and a[lo].


```
private static int partition(Comparable[] a, int lo, int hi) {
    int i = lo, j = hi+1;
    while (true) {
        //find item on left to swap
        while (less(a[++i], a[lo]))
            if (i == hi) break;
        //find item on right to swap
        while (less(a[lo], a[--j]))
            if (j == lo) break;

        //partition done!!!
        if (i >= j) break;
        exch(a, i, j);
    }

    //swap the partitioning item
    exch(a, lo, j);
    return j;
}
```

3. Sort each piece recursively.




```
public class Quick {
    private static int partition(Comparable[] a, int lo, int hi) {
        /* see above */
    }

    public static void sort(Comparable[] a) {
        StdRandom.shuffle(a);
        sort(a, 0, a.length-1);
    }

    private static void sort(Comparable[] a, int lo, int hi) {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

Properties

- Quicksort is an in-place sorting (doesn't use other space) algorithm.
- Quicksort is not stable (Partitioning moves elements with long distance).

Practical Improvements

- Insertion sort small arrays
- esitimate the partition elements at the middle (let the partition element to be medium).

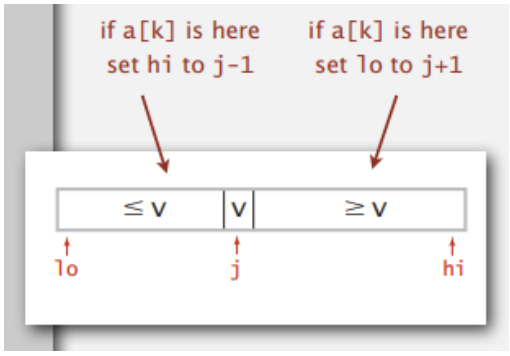
Selection

Given an array of N items, find a kth smallest item.

Quick-select

Partition the array, then repeat on one subarray till we find the value.

```
public static Comparable select(Comparable[] a, int k) {
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length-1;
    while (hi > lo) {
        int j = partition(a, lo, hi);
        if (j < k) lo = j+1;
        else if (j > k) hi = j-1;
        else return a[k];
    }
    return a[k];
}
```



Duplicate keys

Mergesort with duplicate keys uses NlgN/2 ~ NlgN time.

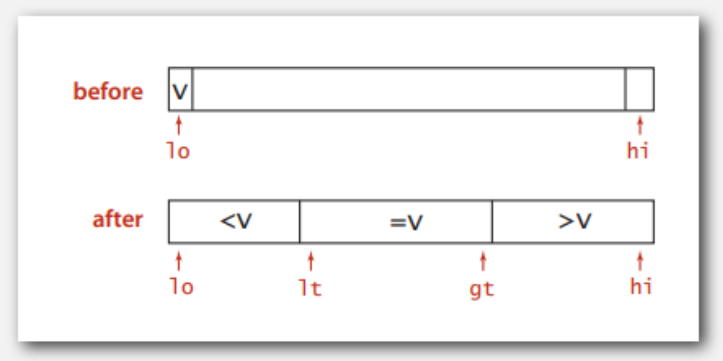
Quicksort with duplicate keys goes quadratic unless partitioning stops on equal keys

Solution:

1. Stop scans on items equal to the partitioning item.
2. Put all items equal to the partitioning item in place. (Desirable)

3-way partitioning

Seperate the array into 3 parts so that, entries between lt and gt equal to partition item v, no larger entries to the left of lt, no smaller entries to the right of gt.



System Sort

Java system sorts: Arrays.sort( ).

- Has different method method for each primitive type.
- Has a method for data types that implement Comparable.
- Has a method that uses a Comparator.
- Uses tuned quicksort for primitive types; tuned mergesort for objects.

Priority Queues

Delete the largest or smallest item in the array.

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

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

Priority queues for simulation (event driven)

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

Warmup: Bouncing balls, N bouncing balls in the unit square.

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

    public void move(double dt) {
```

```
        if ((rx + vx*dt < radius) || (rx + vx*dt > 1 - radius)) { vx = -vx; }
        if ((ry + vy*dt < radius) || (ry + vy*dt > 1 - radius)) { vy = -vy; }
        rx = rx + vx*dt;
        ry = ry + vy*dt;
    }

    public void draw() {
        StdDraw.filledCircle(rx, ry, radius);
    }
}

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);
        }
    }
}
```

Binary Heap

Empty or node with links to left and right binary trees. Heap-ordered binary trees, parent's key is no smaller than children's key.

Properties:

> The largest key is a[1], which is the root of binary tree.
>
> The parent of node k is at k/2.
>
> The children of node k is at 2k and 2k+1.

Senarios

If a child key is greater than its parent key, we exchange the key with the parent key until heap order is restored. (swim operation)

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

Insert a key into the binary heap.

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

```
    swim(N);    // check whether it needs to be moved.
}
```

Parent's key becomes smaller than one (or both) of its children's, we exchange key in parent with key in larger child until heap order is restored. (sink operation)

```
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;
    }
}
```

Delete the maximum in a heap, we exchange root with node at the end, then sink it down.

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

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) {
        pq[++N] = key;
        swim(N);
    }

    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;
    }
}
```

Immutables

Can’t change the data type value once created.

```
// can't override instance methods
public final class Vector {
    // all instance variables private and final
    private final int N;
    private final double[] data;

    public Vector(double[] data) {
        this.N = data.length;
        // defensive copy of mutable instance variables
        this.data = new double[N];
        for (int i = 0; i < N; i++) {
            this.data[i] = data[i];
        }
    }
}
```

Heapsort

Heap construction:

1. Build max heap using bottom-up method.
2. Remove the maximum, one at a time. Leave in array, instead of nulling out.

Implementation

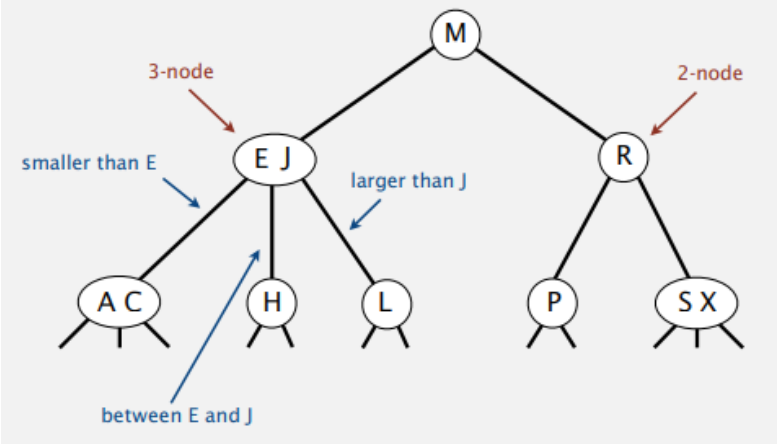
```
public class Heap {
    public static void sort(Comparable[] a) {
        int N = a.length;
        for (int k = N/2; k >= 1; k--) {
            sink(a, k, N);
        }
        while (N > 1) {
            exch(a, 1, N);
            sink(a, 1, --N);
        }
    }

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

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

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

Balanced Search Tree

2-3 tree

Allow 1 or 2 keys per node. 2-node: one key, two children. 3-node: two keys, three children.



Search: Compare search key against keys in node, then determine to go left, right (or middle).

Insertion: Add the new key to the root, then search for its place. If you created a 4-node, move the middle key to the parent and split it into three two nodes.

Perfect balance: Every path from root to node null link has the same length.

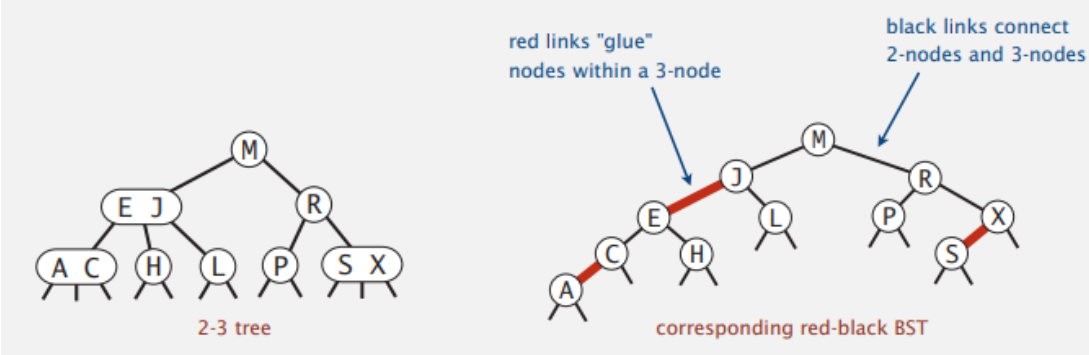
Worse case: logN (all 2-nodes), Best case: log3N (all 3-nodes).

Guaranteed logarithmic performance for search and insert.

red-black BSTs (left-leaning)

Use “internal” left-leaning links as “glue” for 3-nodes. (red link connects a 3-nodes, black link connects 2-nodes and 3-nodes)

The longest path (alternating red and black) can be no more twice as long as the shortest path (all black).



No node has 2 red links connected to it, and every path from root to null link has the same number of black links. Red link leans left.

Search: The same as elementary BST. (but runs faster)

```
public val get(Key key) {
    Node x = root;
    while (x != null) {
        int tmp = key.compareTo(x.key);
        if (tmp < 0) x = x.left;
        else if (tmp > 0) x = x.right;
        else return x.val;
    }
    return null;
}
```

left rotation: orient a temporarily right-leaning red link to lean left.

```
private Node rotateLeft(Node h) {
    Node x = h.right;
    h.right = x.left;
    x.left = h;
    x.color = h.color;
    h.color = RED;
    return x;
}
```

Color flip: Recolor to spilt a temporary 4-node into two 2-node.

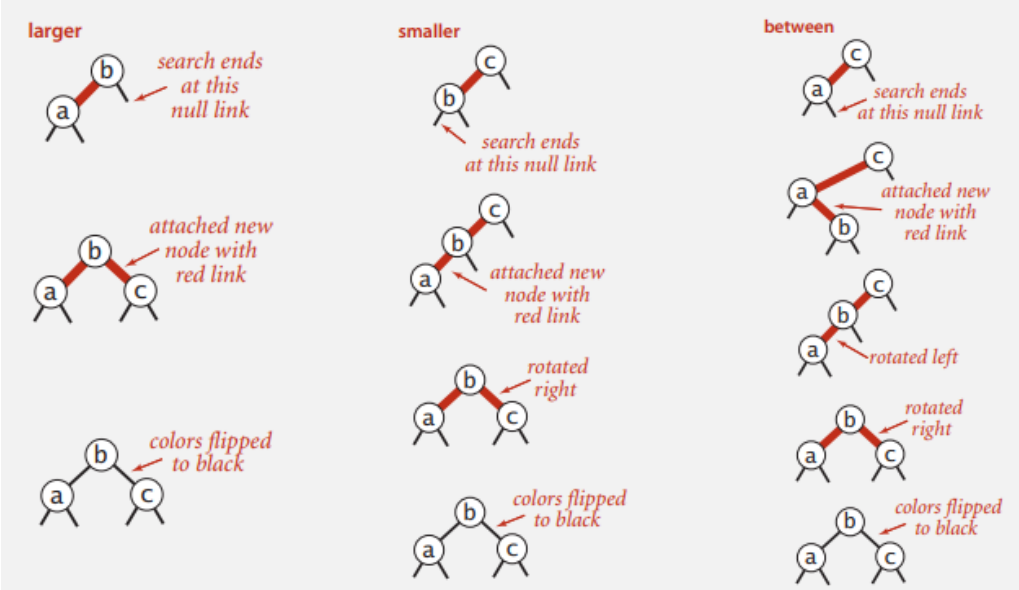
```
private void flipColors(Node h) {
    h.color = RED;
    h.left.color = BLACK;
    h.right.color = BLACK;
}
```

Left-leaning red black tree

Everytime we insert a node into the tree, we change the link of the node to its parent red, and if the red is on the right, flip it.

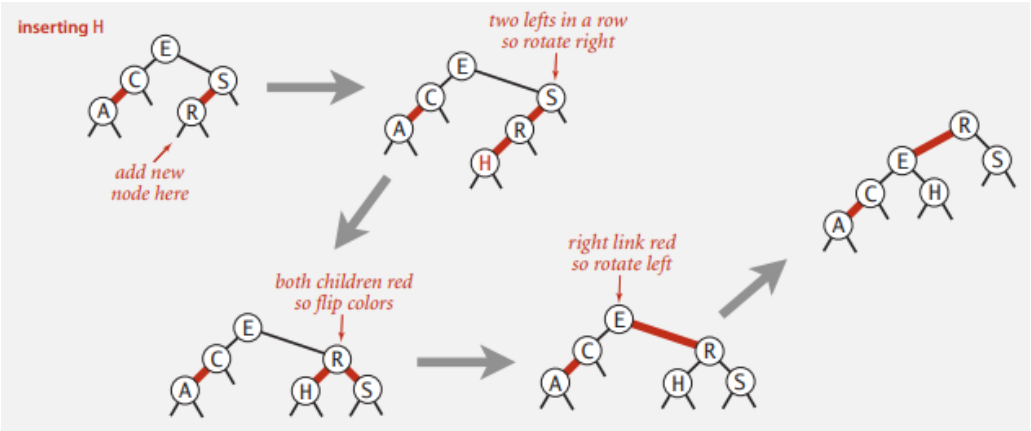
Case.1: Insert into a 2-node at the bottom.

Do standard BST insert, and color the new link red, if the new red link is a right link, rotate left.



Case.2: Insert into a 3-node at the bottom.

Do standard BST insert, color the new link red. (Rotate to balance the 4-node, if needed.) Flip colors to pass the red link up one level. (Rotate to make lean left, if needed.) (Repeat case 1 or case 2 if needed.)



Same code handles all cases:

- Right child red, left child black: rotate left.
- Left child red, left-left grandchild red: rotate right.
- Both children red: flip colors.

```
private Node put(Node h, Key key, Value val) {
    if (h == null) return new Node(key, val, RED);
    int tmp = key.compareTo(h.key);
    if (tmp < 0) h.left = put(h.left, key, val);
```

```
    else if (tmp > 0) h.right = put(h.right, key, val);
    else h.val = val;

    // lean left
    if (isRed(h.right) && !isRed(h.left)) h.rotateLeft(h);
    // balance 4-node
    if (isRed(h.left) && isRed(h.left.left)) m.rotateRight(h);
    // split 4-node
    if (isRed(h.left) && isRed(h.right)) flipColors(h);

    return h;
}
```

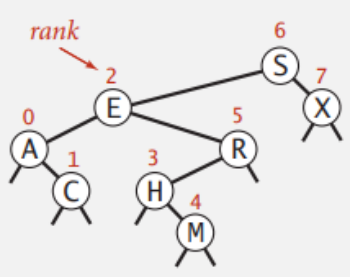
Geometric Applications of BSTs

1D range search

API:

- Insert key-value pair
- Search for key k
- Delete key k
- Range search: find all keys between k1 and k2.
- Range count: number of keys between k1 and k2.

implementing BST



1D range count

```
public int size(Key lo, Key hi) {
    if (contains(hi)) return rank(hi)-rank(lo)+1;
    else return rank(hi)-rank(lo);
}
```

1D range search: Find all keys between lo and hi.

- Recursively find all keys in left subtree
- check key in current node
- Recursively find all keys in right subtree

Line segment intersection

Given N horizontal and vertical line segments, find all intersections.

Solution: Sweep vertical line from left to right.

- x-coordinates define events → NlogN
- h-segment (left endpoint): insert y-coordinate into BST. → NlogN
- h-segment (right endpoint): remove y-coordinate from BST. → NlogN
- v-segment: range search for interval of y-endpoints. → NlogN+R

2D trees

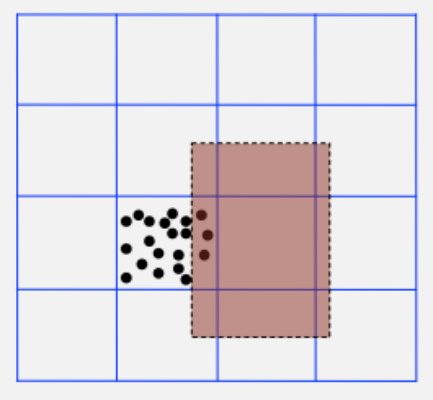
API:

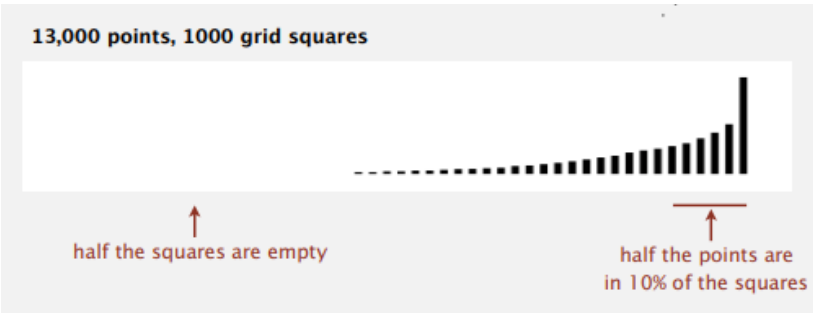
- Insert a 2D key
- Search for a 2D key.
- Range search: find all keys that lie in a 2D range.
- Range count: numver of keys that lie in a 2D range.

Grid implementation: split the area into M by M grids

Space: M*M + N, Time: 1+ N/M^2 per square examined, on average

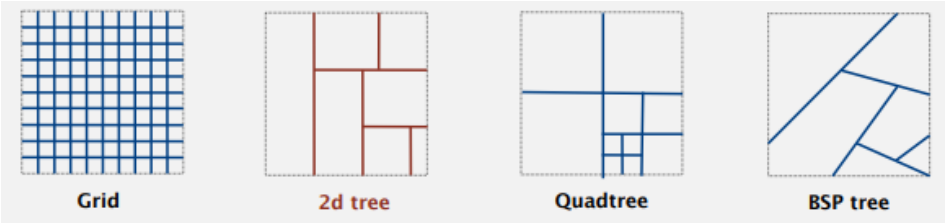
Problem: Clustering, the list is too long even though average length is small.



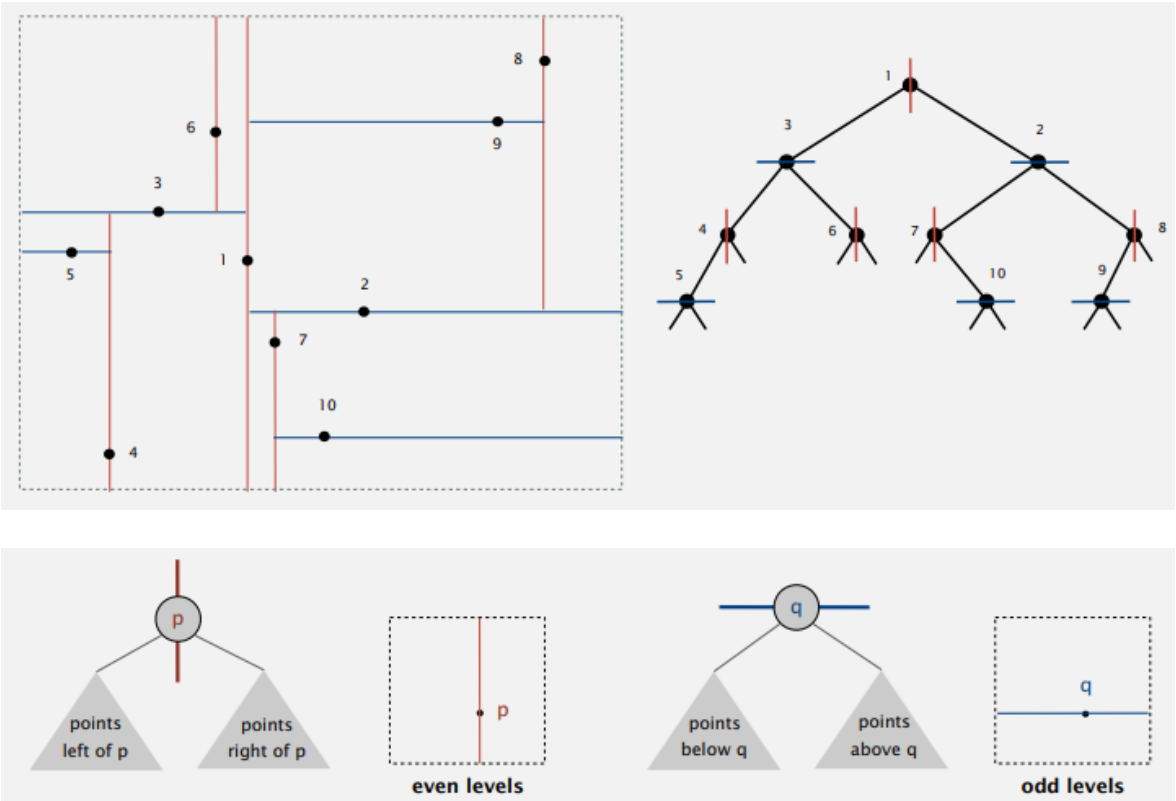


Use a tree to represent a recursive subdivision of 2D space.

- Grid: Divide space uniformly into squares.
- 2D tree: Recursively divide space into two halfplanes.
- Quadtree: Recursively divide space into four quadrants.
- BSP tree: Recursively divide space into two regions.

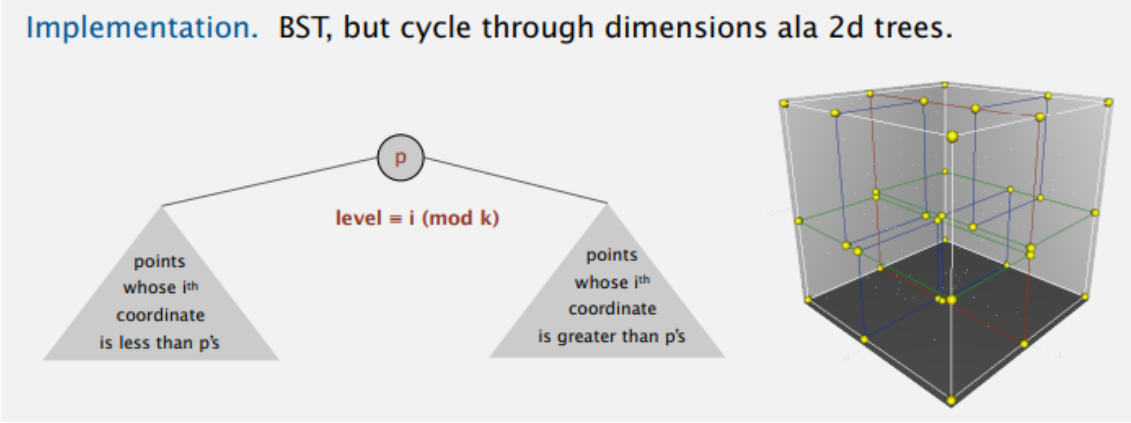


2D tree construction:



KD tree

Recursively partition k-dimensional space into 2 halfspaces.

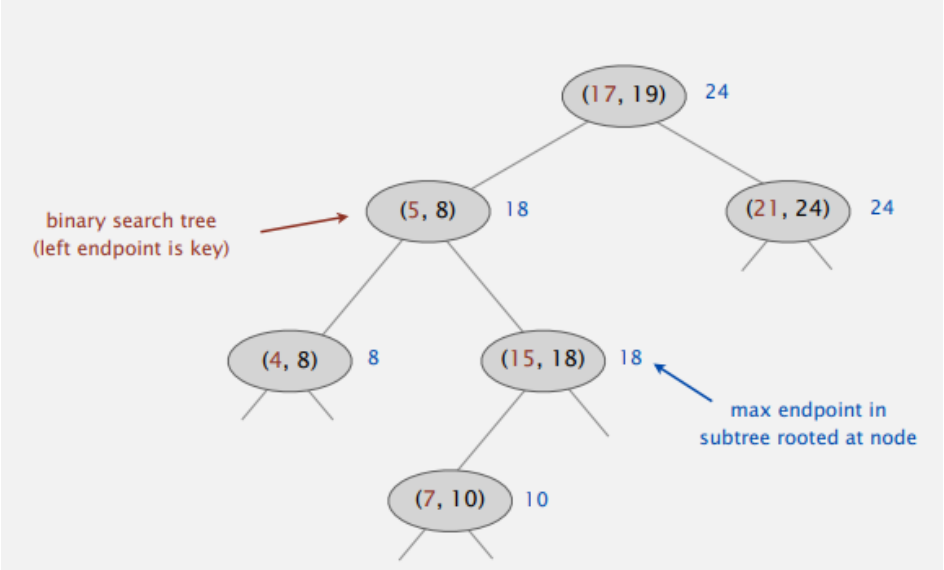


1D interval search

API:

- Insert an interval (lo, hi)
- Search for an interval (lo, hi)
- Delete an interval (lo, hi)
- Interval intersection query: given an interval (lo, hi), find all intervals in data structure that intersects (lo, hi)

Create BST, where each node stores an interval (lo, hi). Use left endpoint as BST key. Store max endpoint in subtree rooted at node.



Insert node:

Insert into BST, use lo as the key. Update max in each node on search path.

Search tree:

If interval in node intersects query interval, return it. Else if left subtree is null, go right. Else if max endpoint in left subtree is less than lo, go right. Else go left.

```
Node x = root;
while (x != null) {
    if (x.interval.intersects(lo, hi)) return x.interval;
    else if (x.left == null) x = x.right;
    else if (x.left.max < lo) x = x.right;
    else x = x.left;
}
```
