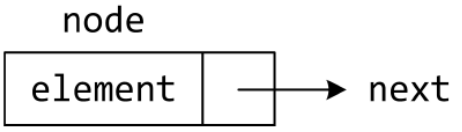
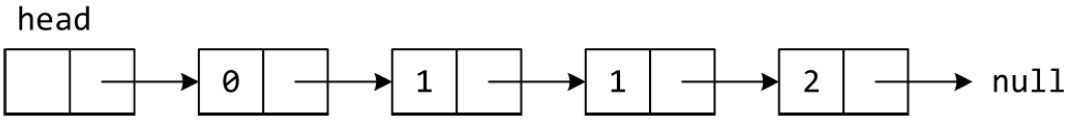
Linked lists…

Linked lists

- A linked list all elements as a sequence of nodes
- Each node stores
  - the data element and
  - a link to the next node

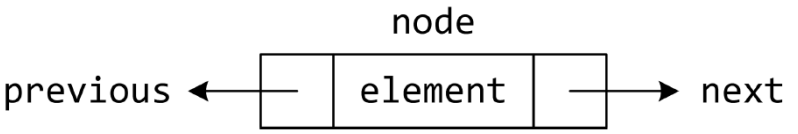


- Normally, a special head node is used

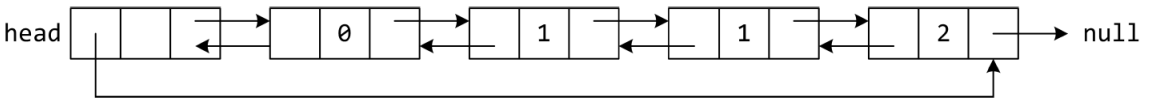


Doubly linked lists

- A doubly linked list is an extension of a (single) linked list
- In addition a node object stores the link to previous node

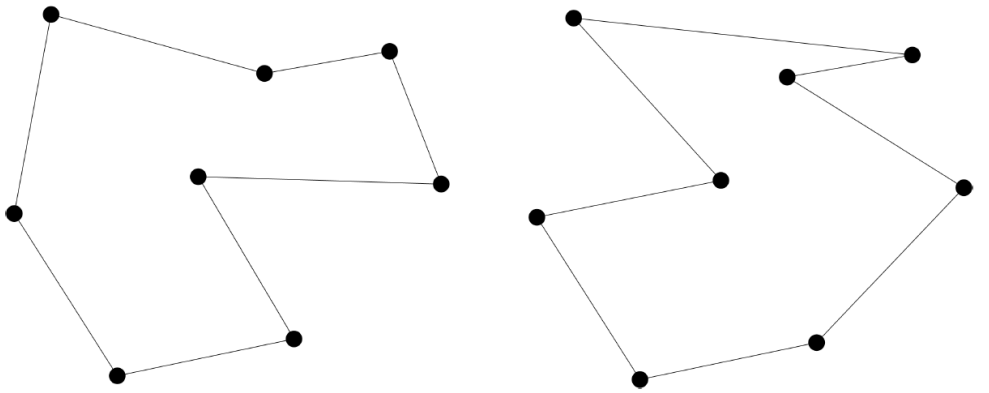


- The two links allow walking along the list in either direction with equal ease.
- The previous node of the head is the last node of the list
- Normally, a special head node is used



Example: Polygon

- A polygon is a closed path of line segments.



Example: Polygon

- A polygon is defined by a list of points. The start point of the first segment is at the same time the end point of the last segment.

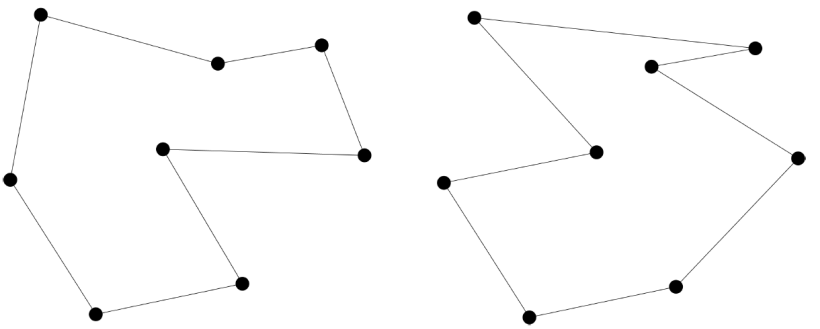
$$\mathbf{P} := < p_0, p_1, \ldots, p_{n-1} >$$

- Often data structures are used within another classes to store and organize the attributes.
- A polygon can be implemented as array list or linked list. The choice depends on the type of usage:
  - If you often need a direct access via an index – use array list implementation.
  - If you often modify the polygon via insert, remove or add operations – use linked list implementation.

  → Here we will implement the polygon class as a linked list

Operations for polygons

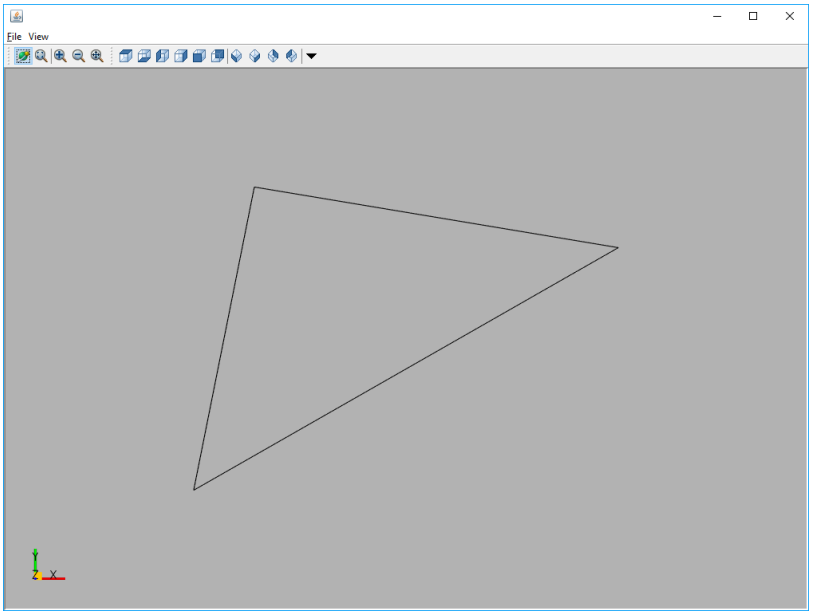
- Add a point at the beginning of the list of points
- Find a node before a specified node
- Remove a specified node
- Draw the polygon on the screen



Polygon demo program

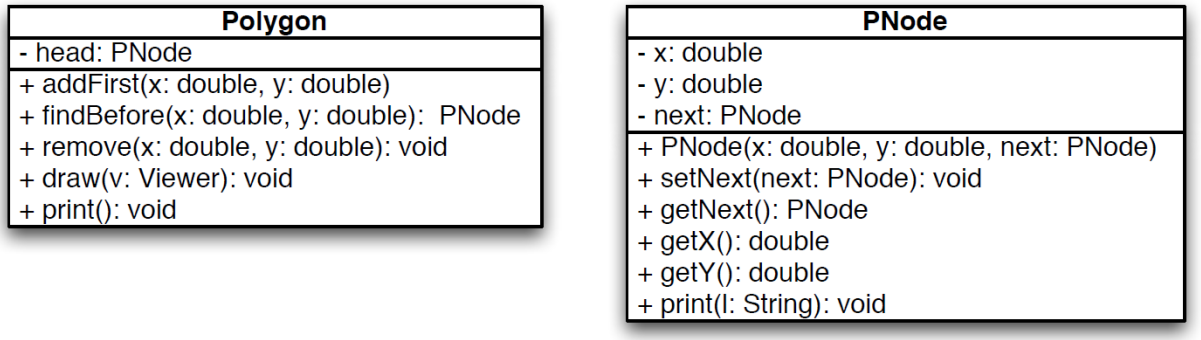

```
public class PolygonDemoProgram {

  public static void main(String[] args) {
    Viewer v = new Viewer();
    Polygon p = new Polygon();

    p.addFirst(1.0, 0.0);
    p.addFirst(2.0, 5.0);
    p.addFirst(8.0, 4.0);
    p.addFirst(7.0, 3.0);
    p.print();

    PNode n = p.findBefore(8.0, 4.0);

    n.print("Found: ");
    System.out.println();

    p.remove(7.0, 3.0);
    p.print();
    p.draw(v);
    v.setVisible(true);
  }
}
```

Polygon demo program: Output



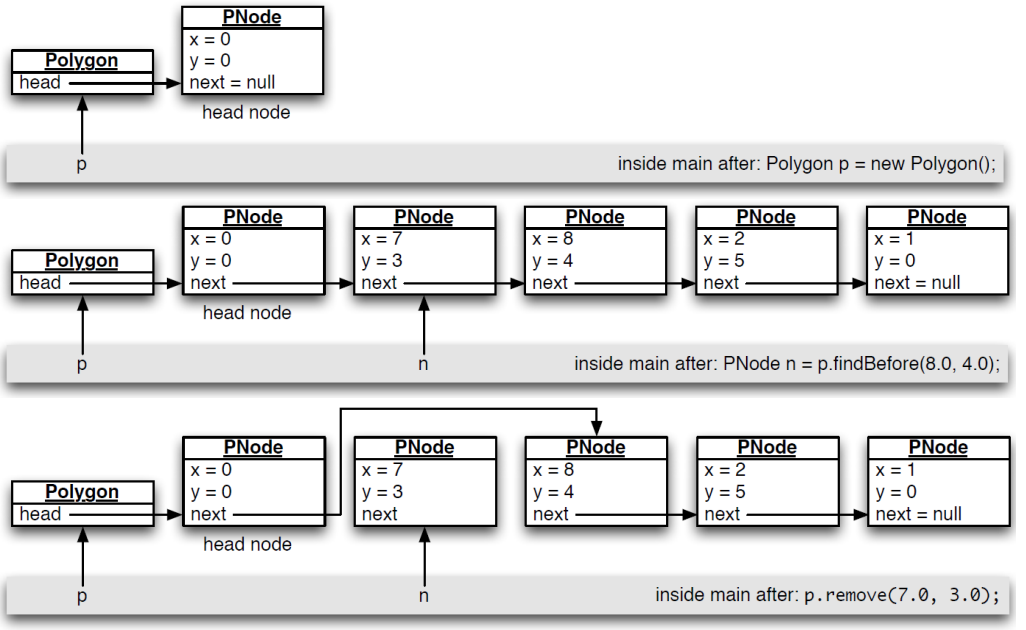

```
->(7.0, 3.0)->(8.0, 4.0)->(2.0, 5.0)->(1.0, 0.0)
Found: (7.0, 3.0)
->(8.0, 4.0)->(2.0, 5.0)->(1.0, 0.0)
```

UML class diagram



- Node class stores point position and pointer to next node (implementation is straightforward, not discussed here)
- Polygon has pointer to head node
- Add first adds node at beginning of chain of nodes
- Find before returns the node before the specified node
- Remove removes specified node

UML class diagram



– Advantage of using a head node for the empty list:
Every node has a predecessor

Implementation of the Polygon class


```
public class Polygon {

  private Node head = new Node(0, 0, null);

  public void addFirst(double x, double y) {
    this.head.setNext(new Node(x, y, this.head.getNext()));
  }

  public void print() {
    Node node = head.getNext();

    while (node != null) {
      node.print("->");
      node = node.getNext();
    }
    System.out.println();
  }

  // other methods come here ...
}
```

Other interesting methods

- Find node before a specified node

```
public PNode findBefore(double x, double y) {
  PNode n = this.head;

  while (n.getNext() != null) {
    if (n.getNext().getX() == x && n.getNext().getY() == y) {
      return n;
    }
    n = n.getNext();
  }
  return null;
}
```

- Remove a node

```
public void remove(double x, double y) {
  PNode n = this.findBefore(x, y);
  PNode newNextNode = n.getNext().getNext();

  n.setNext(newNextNode);
}
```

Other interesting methods

- Draw polygon

```
public void draw(Viewer v) {
  Polyline polyline = new Polyline();
  PNode node = this.head.getNext();

  while (node != null) {
    polyline.addVertex(node.getX(), node.getY(), 0, 0);
    node = node.getNext();
  }
  node = this.head.getNext();
  polyline.addVertex(node.getX(), node.getY(), 0);

  polyline.setColor("black");
  v.addObject3D(polyline);
}
```

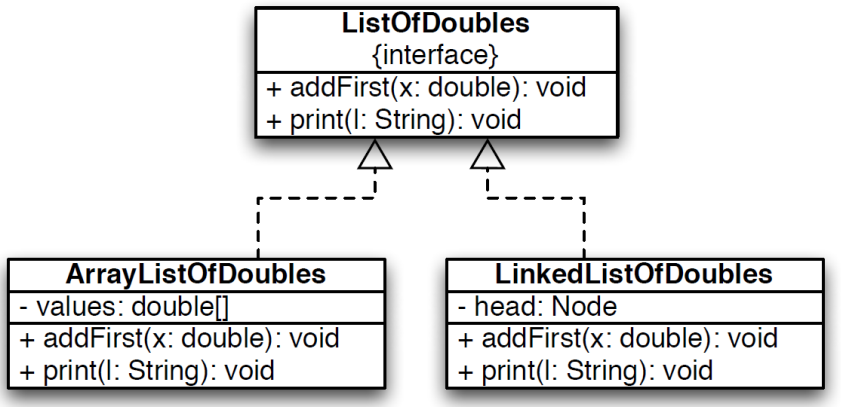
Excursus: Interfaces…

Interfaces by example


```
public interface ListOfDoubles {

  public void addFirst(double x);

  public void print(String l);
}
```

- An interface defines operations without specifying how to implement them
- Interfaces are the Java replacement for multiple inheritance
- An interface can be implemented by many classes

UML class diagram



- The interface is implemented by two classes:
  - One using an array
  - One using a linked list
- Implementing classes are connected to the interface by a dashed line with a triangle pointing to the interface
- Implementing classes provide methods defined by the interface

Array list implementation


```
public class ArrayListOfDoubles implements ListOfDoubles {

  private double[] values = new double[0];

  public void addFirst(double x) {
    int n = this.values.length;
    double[] tmp = new double[n + 1];

    tmp[0] = x;
    System.arraycopy(this.values, 0, tmp, 1, n);
    this.values = tmp;
  }

  public void print(String l) {
    System.out.print(l + "<");
    for (int i = 0; i < this.values.length; i++) {
      System.out.print(this.values[i]);
      System.out.print(" ");
    }
    System.out.println(">");
  }
}
```

Array list implementation


```
public class LinkedListOfDoubles implements ListOfDoubles {

  private Node head = new Node(0, null);

  public void addFirst(double x) {
    Node n = new Node(x, this.head.getNext());
    this.head.setNext(n);
  }

  public void print(String l) {
    Node n = this.head.getNext();

    System.out.print(l + "<");
    while (n != null) {
      System.out.print(n.getValue() + " ");
      n = n.getNext();
    }
    System.out.println(">");
  }
}
```

Demo program


```
public class ListOfDoublesDemoProgram {
  public static void main(String[] args) {
    ListOfDoubles l1 = new ArrayListOfDoubles();
    ListOfDoubles l2 = new LinkedListOfDoubles();

    l1.addFirst(1.1);
    l1.addFirst(6.2);
    l1.addFirst(9.4);
    l2.addFirst(1.1);
    l2.addFirst(6.2);
    l2.addFirst(9.4);
    l1.print("l1=");
    l2.print("l2=");
  }
}
```

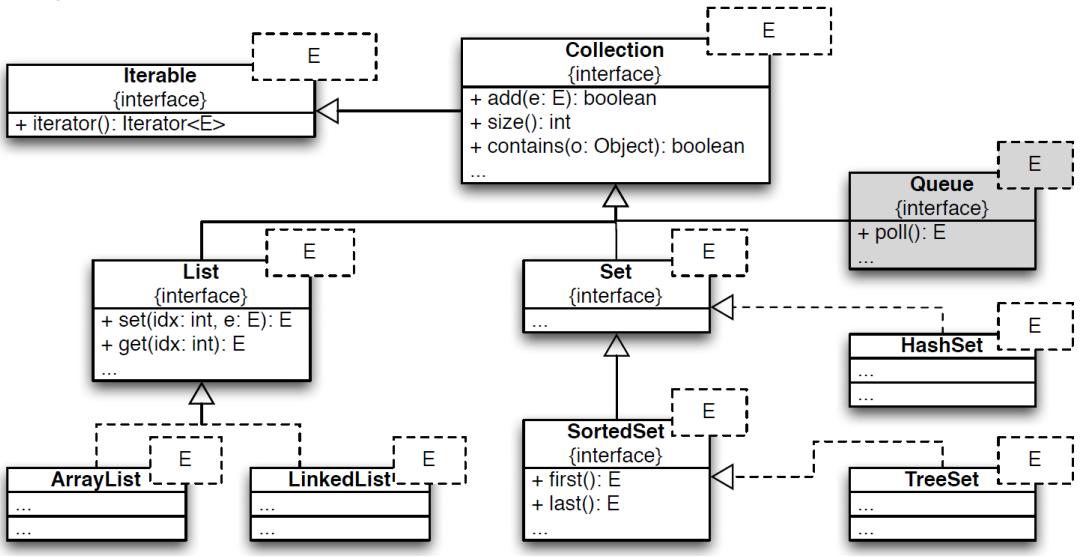
```
l1=<9.4 6.2 1.1 >
l2=<9.4 6.2 1.1 >
```

- Both implementations give the same result
- The interface is used as data type

Java Collections Framework

- Any serious programming projects requires efficient and reliable classes providing data structures to store collections of objects.
- Java provides such classes in the so called Java Collections Framework.
- An excellent comprehensive introduction is given under https://docs.oracle.com/javase/tutorial/collections/
- The framework makes a distinction between:
  - Interfaces defining available operations
  - Classes implementing the interfaces in different ways (ArrayList vs. LinkedList)

Core interfaces and classes



- **E:** The data type, the interface is used for
- **Iterable:** Lets us use `foreach` loop
- **List:** Store elements in an ordered sequence
- **Set:** Store elements without duplicates

Java Generics – Examples

- `LinkedList<String> a = new LinkedList<String>();`
  - This linked list object can only store String objects
  - Generics are specified in angle brackets
- `TreeSet<Point> b = new TreeSet<Point>();`
  - This binary tree objet can only store Point objects
- `ArrayList<Double> c = new ArrayList<Double>();`
  `c.add(1.0); double d = c.get(0);`
  - This array list object converts primitive types into ist correnspondingclasses and the other way round

List demo program


```
List<Double> l1 = new ArrayList<Double>();
List<String> l2 = new LinkedList<String>();

l1.add(1.0);
l1.add(PI);
l1.add(1.0);
l2.add("Beethoven");
l2.add("Strawinsky");
l2.add("Bach");
l2.add("Beethoven");

System.out.print("  Numbers: ");
for (double d : l1) {
  System.out.print(d + " ");
}
System.out.print("\nComposers: ");
for (String s : l2) {
  System.out.print(s + " ");
}
```

```
Numbers: 1.0 3.141592653589793 1.0
Composers: Beethoven Strawinsky Bach Beethoven
```

Set demo program


```
Set<Double> l1 = new HashSet<Double>();
Set<String> l2 = new TreeSet<String>();

l1.add(1.0);
l1.add(PI);
l1.add(1.0);
l2.add("Beethoven");
l2.add("Strawinsky");
l2.add("Bach");
l2.add("Beethoven");

System.out.print("  Numbers: ");
for (double d : l1) {
  System.out.print(d + " ");
}
System.out.print("\nComposers: ");
for (String s : l2) {
  System.out.print(s + " ");
}
```

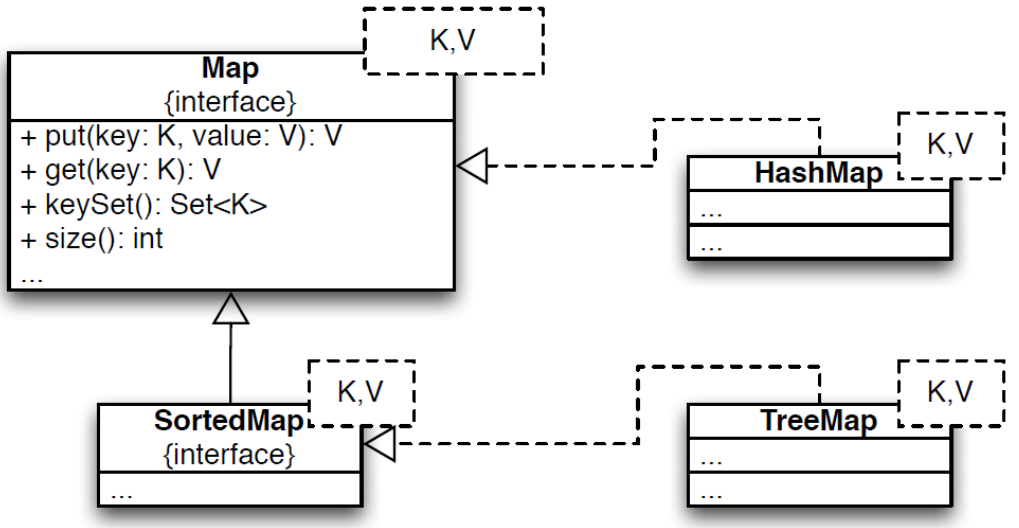
```
Numbers: 3.141592653589793 1.0
Composers: Bach Beethoven Strawinsky
```

Summary

- Specify type of data to be stored in sharp brackets
- For primitive types, use the corresponding wrapper type
- Use `foreach` loop to process all elements efficiently

Maps…

Maps



- Maps store key value pairs
- Uses: Dictionary, Phone book, . . .
- A sorted map is sorted by the keys

Maps

```
Map<Character,Integer> cm = new HashMap<Character,Integer>();
cm.put('a', 99);
cm.put('b', 1);
cm.put('c', 6);
cm.put('d', 85);

for (char key : cm.keySet()) {
  System.out.print(key + "->" + cm.get(key) + " ");
}
```

```
  d->85 b->1 c->6 a->99
```

```
Map<Character,Integer> cm = new TreeMap<Character,Integer>();
cm.put('a', 99);
cm.put('b', 1);
cm.put('c', 6);
cm.put('d', 85);

for (char key : cm.keySet()) {
  System.out.print(key + "->" + cm.get(key) + " ");
}
```

```
  a->99 b->1 c->6 d->85
```

Complexity of operations

- The choice of a certain implementation depends on the desired properties of the collection and the type of usage
- Complexity of operations

Implementation	add/put	get	sorted
ArrayList	O(n)	O(1)	yes
LinkedList	O(1)	O(n)	yes
TreeSet	O(log(n))	O(log(n))	yes
HashSet	O(1)	O(1)	no
TreeMap	O(log(n))	O(log(n))	yes
HashMap	O(1)	O(1)	no

- Hash sets and maps are based on hashing functions. A hashing function maps an object on an integer such that this integer can be used as index for storage in an array.

Application example: Book analyser…

Count appearances of persons in a book

- How often do the names
  - Mulligan
  - Molly
  - Bloom
  - Dedalus

  appear in the book Ulysses by James Joyce?

- Solution ingredients
  - Free books on the web (https://archive.org/)
  - URL: Class to read from a webserver
  - Scanner: Class to tokenise a stream
  - Set: List of persons to consider
  - Map: Count occurences

Demo program


```
String url =
    "http://www.gutenberg.org/files/4300/4300-8.txt";
Book b = new Book(url);

b.addImportantPerson("Mulligan");
b.addImportantPerson("Molly");
b.addImportantPerson("Bloom");
b.addImportantPerson("Dedalus");
b.listAppearances();
```

```
Mulligan: 115
Molly: 41
Dedalus: 116
Bloom: 428
```

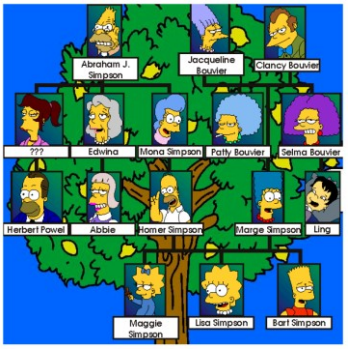
Basic methods


```
public class Book {

  private URL url;
  private Set<String> persons = new HashSet<String>();

  public Book(String url) throws IOException {
    this.url = new URL(url);
  }

  public void addImportantPerson(String p) {
    this.persons.add(p);
  }

  // other methods come here...
}
```

List persons

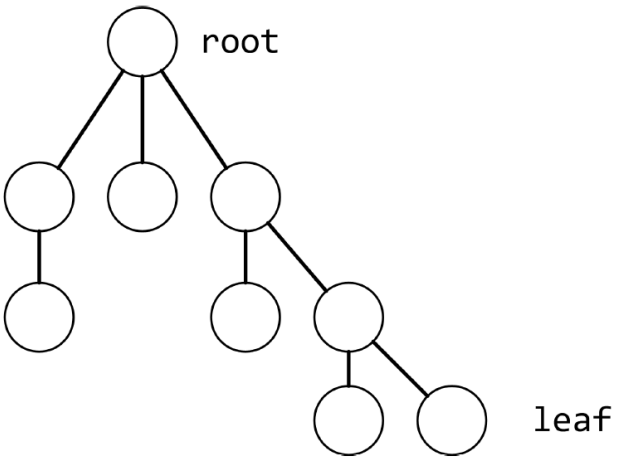

```
public void listAppearances() throws IOException {
  Scanner scanner = new Scanner(this.url.openStream());
  Map<String, Integer> appearances =
                          new TreeMap<String, Integer>();

  for (String p : this.persons) {
    appearances.put(p, 0);
  }

  while (scanner.hasNext()) {
    String s = scanner.next();

    if (this.persons.contains(s)) {
      System.out.println("Found: " + s);
      appearances.put(s, appearances.get(s) + 1);
    }
  }

  for (String person : appearances.keySet()) {
    System.out.println(person + ": "
                              + appearances.get(person));
  }
}
```

Trees…


Abraham J. Simpson
Jacqueline Bouvier
Clancy Bouvier
???
Edwina
Mona Simpson
Patty Bouvier
Selma Bouvier
Herbert Powel
Abbie
Homer Simpson
Marge Simpson
Ling
Maggie Simpson
Lisa Simpson
Bart Simpson

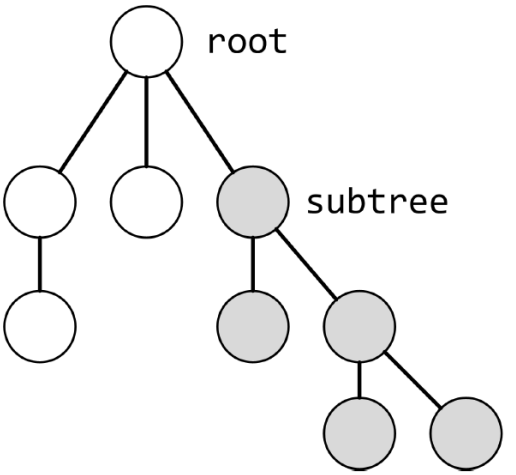
Trees

- Trees describe hierarchical structures of elements based on a set of linked nodes
- Each node in a tree stores an element and has zero or more child nodes, which are below it in the tree
- The topmost node in a tree is called the root node
- Nodes at the bottommost level of the tree are called leaf nodes



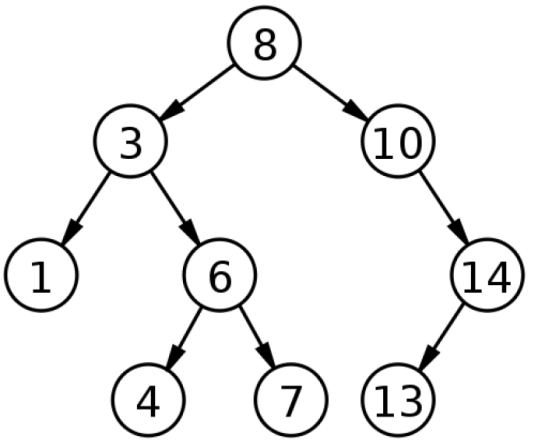
Types of trees

- Any node in a tree T, together with all the nodes below it, comprise a so-called subtree of T
- Different tree types are distinguished by the number of possible child nodes
  - Binary trees
  - Quad trees
  - Oct trees
- Often trees are used to store elements in a certain order



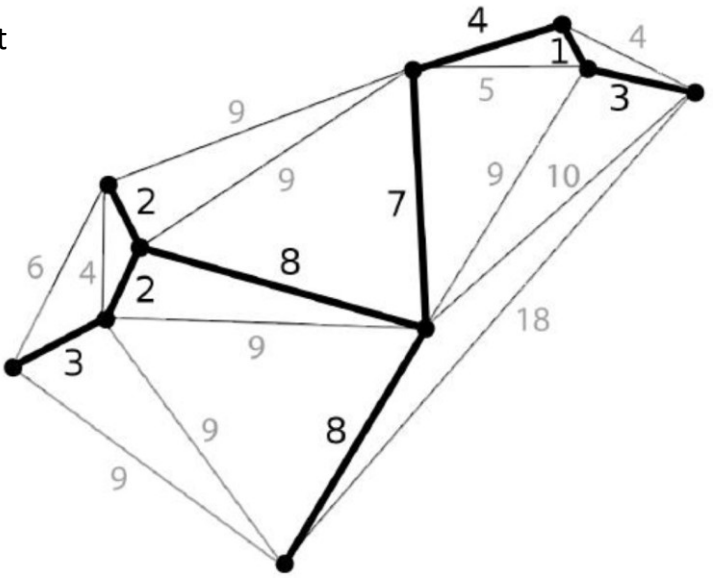
Binary trees

- Often trees are used to store elements in a certain order
- For example the binary search tree
  - The left subtree of a node contains only nodes which are less than the node
  - The right subtree of a node contains only nodes which are not less than the node
  - Both the left and right subtrees must also be binary search trees



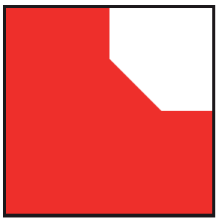
Tree algorithms

- Efficient algorithms can be implemented using tree structures
  - Sorting elements
  - Finding a certain element
  - Clustering of data
  - Minimum spanning trees for connecting vertices

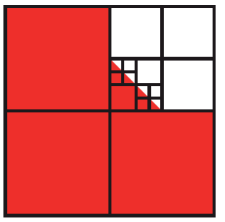


Quadtrees…

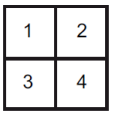
Quadtrees

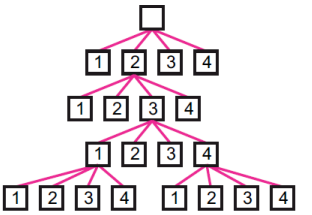


Domain with properties



Partitioning of domain





Corresponding quadtree

- Problem statement
  - **Given**: A domain where each point of the domain is associated with a certain property, e.g. the color.
  - **Needed**: A partitioning of the domain into simple, rectangular subdomains having mostly uniform properties.
- Solution idea
  - Recursively refine region where needed
  - Store rectangles in a tree-like data structure
  - Each node in the tree represents one rectangle
  - A tree node can have either zero or four subnodes

Applications of quadtrees

- Quadtrees are widely used in various fields of computer graphics, engineering and information systems:
  - Grid generation for finite element meshes
  - Collision detection
  - Geoinformation systems: Store data effciently
  - Image analysis

Quadtrees: Java implementation…

Domain class

Domain
- image: BufferedImage - colors: int[]
+ Domain(filename: String, nColors: int) + colorIdxAt(x: double, y: double): int + isMonochromatic(x: double, y: double, s: double): boolean

- Define domain using an image file (.jpg, .png, . . .)
- The domain is of size 1 x 1
- For the image, `nColors` different colors are distinguished
- The `colorIdxAt` returns the index of the color at the specified position
- `isMonochromatic` tests if the specified region contains only colors of one index

Domain class in Java


```
public class Domain {

  private static int N_SAMPLES = 25;

  private int[] colors;
  private BufferedImage image;

  public Domain(String filename, int nColors)
                              throws IOException {
    double dc = 255.0 / nColors;

    this.image = ImageIO.read(new File(filename));
    this.colors = new int[nColors];
    for (int i = 0; i < nColors; i++) {
      this.colors[i] = (int) ((i + 1) * dc);
    }
  }

  // other methods go here ...
}
```

Getting the color index


```
public int colorIdxAt(double x, double y) {
  int iw = this.image.getWidth(null);
  int ih = this.image.getHeight(null);
  int px = (int) (x * iw);
  int py = (int) (ih - y * ih);
  Color c = new Color(this.image.getRGB(px, py));
  int b = (c.getRed() + c.getGreen() + c.getBlue()) / 3;
  int idx = Arrays.binarySearch(this.colors, b);

  if (idx >= 0) {
    return idx;
  } else {
    return -idx - 1;
  }
```

- Parameters *x* and *y* are in [0, 1]
- Map standard domain $[0, 1]^2$ on pixel domain of picture
- Use average color intensity (grayscale image)
- Search color index using binary search

Finding out if a subdomain is monochromatic


```
public boolean isMonochromatic(double x,
                               double y, double s) {
  double d = s / N_SAMPLES;
  int idx = colorIdxAt(x + d / 2, y + d / 2);

  for (int i = 0; i < N_SAMPLES; i++) {
    double px = x + (i + 0.5) * d;
    for (int j = 0; j < N_SAMPLES; j++) {
      double py = y + (j + 0.5) * d;

      if (idx != this.colorIdxAt(px, py)) {
        return false;
      }
    }
  }
  return true;
}
```

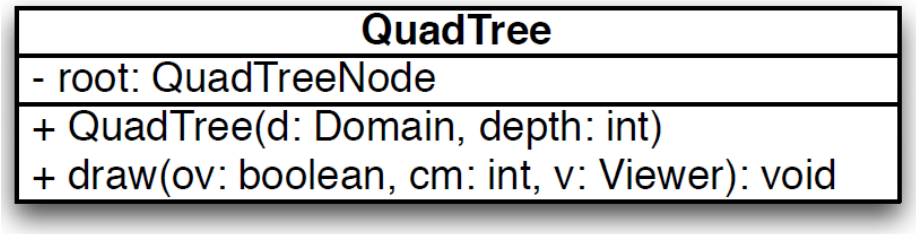
- Parameters x, y and s specify subdomain $S \subseteq [0, 1]^2$
- Check color index at sample points against reference color
- Return false if color index does not match

Using the Domain class


```
Domain d = new Domain("img/section1.png", 10);

System.out.println("      (0.1, 0.1): "
    + d.colorIdxAt(0.1, 0.1));
System.out.println("      (0.1, 0.9): "
    + d.colorIdxAt(0.1, 0.9));
System.out.println("      (0.8, 0.2): "
    + d.colorIdxAt(0.8, 0.2));
System.out.println("(0.1, 0.1, 0.1): "
    + d.isMonochromatic(0.1, 0.1, 0.1));
System.out.println("(0.0, 0.0, 1.0): "
    + d.isMonochromatic(0.0, 0.0, 1.0));
System.out.println("(0.3, 0.3, 0.4): "
    + d.isMonochromatic(0.3, 0.3, 0.4));
```

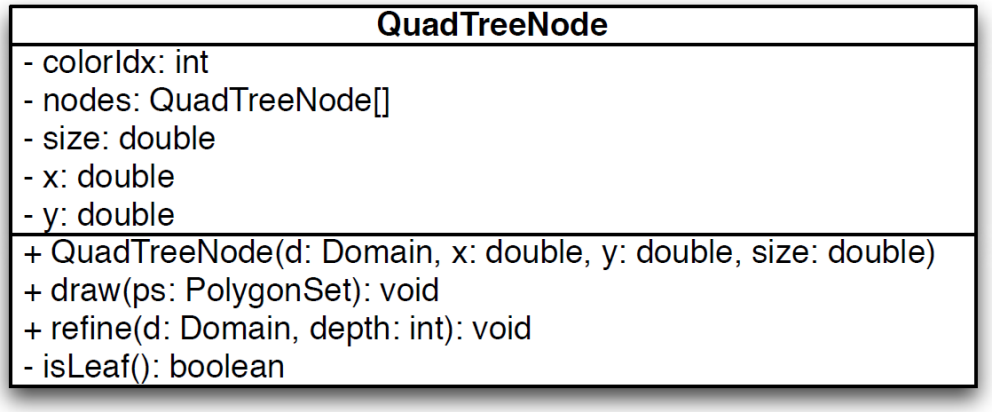
```
      (0.1, 0.1): 4
      (0.1, 0.9): 6
      (0.8, 0.2): 3
(0.1, 0.1, 0.1): true
(0.0, 0.0, 1.0): false
(0.3, 0.3, 0.4): false
```

The QuadTree class



- Pointer to root node
- Specify domain and refinement depth in constructor
- Draw quadtree, parameters determine appearance

The QuadTreeNode class



- Specify domain, position and size in constructor
- Draw using a PolygonSet
- Refine up to the specified level
- Check if the node is a leaf not (i.e. does not have subnodes)

QuadTree in Java


```
public class QuadTree {

  private QuadTreeNode root;

  public QuadTree(Domain d, int depth) {
    this.root = new QuadTreeNode(d, 0.0, 0.0, 1.0);
    this.root.refine(d, depth);
  }

  public void draw(boolean ov, int cm, Viewer v) {
    PolygonSet ps = new PolygonSet();

    this.root.draw(ps);
    ps.setColoringByData(true);
    ps.setOutlinesVisible(ov);
    ps.setColorMode(cm);
    ps.createColors();
    v.addObject3D(ps);
  }
}
```

QuadTreeNode in Java


```
public class QuadTreeNode {

  private int colorIdx;
  private QuadTreeNode[] nodes;
  private double size;
  private double x;
  private double y;

  public QuadTreeNode(Domain d, double x, double y,
                                          double size) {
    double s2 = size / 2;
    this.colorIdx = d.colorIdxAt(x + s2, y + s2);
    this.x = x;
    this.y = y;
    this.size = size;
  }

  private boolean isLeaf() {
    return this.nodes == null;
  }

  // other methods go here ...
}
```

Refinement procedure

- The `refine` method can be implemented using this algorithm:
  - If the subdomain represented by this `QuadTreeNode` is not monochromatic:
    1. Create the array for the four subtree nodes
    2. For $i = 0, 1$ and $j = 0, 1$
       - 2.1 Create a new `QuadTreeNode` at the right position and of the right size
       - 2.2 Refine the new node if `depth` is larger than zero. Pass `depth-1` as parameter
       - 2.3 Store the new node in the array of subtree nodes

Drawing a node

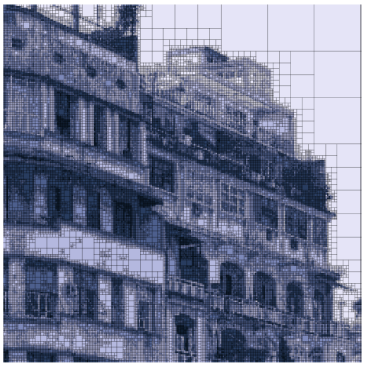
- The draw method can be implemented as follows:
  - If the node is a leaf node
    1. add a polygon to the specified polygon set

    otherwise
    1. invoke the draw method for all subnodes

Example 1

Example 2





Eden building in Saigon, District 1 (demolished)