Initial Review

ADT — Abstract Data Type – a class, has behaviors and values
memory allocation, value interpretation, allowable operators

collection — (ADT data structure) single reference (name) many same typed (usually) values

generic — defer type specification of collection until construction.

```
public class GenericClass <T> { ...}
...
GenericClass <String> aStr =
   new GenericClass <String>();
```

array — statically allocated collection of same typed values
accessed by an index (subscript)

ArrayList — dynamically allocated **generic** collection
accessed by an index or Iterator
or bi-directionally with a ListIterator

ArrayListDemo.java

```
import java.util.ArrayList;
import java.util.Iterator;

public class ArrayListDemo {
  ArrayList<Integer> anArrayList;

  public void show() {
    Iterator <Integer> iterator = anArrayList.iterator();
    while (iterator.hasNext()) System.out.print(iterator.next() + " ");
    System.out.println();
    }

   public ArrayListDemo(int n) {
    anArrayList = new ArrayList<Integer>();
    for(int i = 0; i < n; i++)
      anArrayList.add(new Integer((int) (Math.random() * 100)));
    }

  public static void main(String[] arg) {
    if (arg.length == 1) {
      ArrayListDemo ald = new ArrayListDemo(Integer.parseInt(arg[0]));
      ald.show(); }
    else
      System.out.println("example:  java ArrayListDemo 10");
   }
}
```

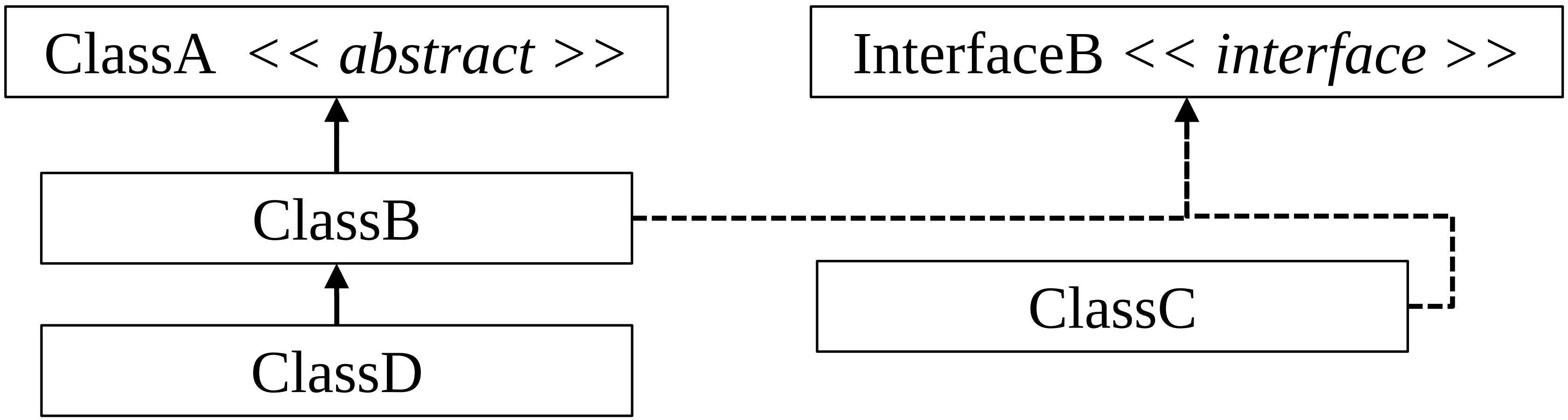
Java's class and interface ADTs

class **single** inheritance ADT
“extends” class can extend (inherit) from one super class

abstract “virtual” class – can’t instantiate objects
class w/ abstract method must be an abstract class
abstract classes enable polymorphism

interface **multiple** “behavioral” inheritance ADT
class can “implement” many interfaces ("mix-ins")
interface is ADT with method signatures (no definitions)
implementors of an interface must define the “inherited” interface methods
increases collection types for membership

InterfaceTypeDemo



Object name	**Type membership** (collection type)
aClassB	ClassB, ClassA, InterfaceB
aClassC	ClassC, InterfaceB
aClassD	ClassD, ClassB, ClassA, InterfaceB

ClassB and ClassD objects can belong to collections of ClassA
ClassB, ClassC and Class D can belong to collections of InterfaceB

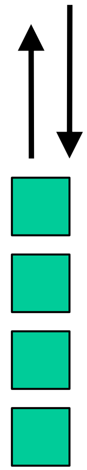
```
ArrayList <InterfaceB> aList;
```

see InterfaceTypeDemo.java example

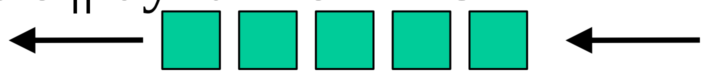
182 Data Structures

linked list — dynamically allocated collection
navigable w/ iterator

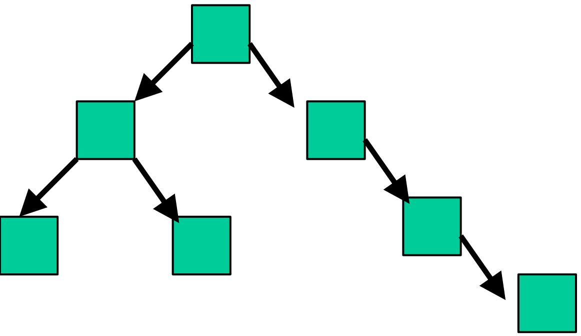
stack — static || dynamic LIFO ADT



queue — static || dynamic FIFO ADT



binary tree — dynamic, ordered (on comparable Key)
hierarchical collection rooted w/ first insertion
Each cell has a less-than and greater-than subtree.
[Key, Value(s), < subtree, > <subtree>]
“leaves” are non-navigable (null) subtrees.



balanced binary search tree — tree w/ all leaves at 2 (adjacent) levels
height and height -1 where height = $\log_2 (n + 1)$
average retrieve, insert, delete $O(\log n)$

non-balanced — worst case retrieve, insert, delete $O(n)$

Search performance

Equivalent search performance

JCF ArrayList – sort then retrieve using a binary search

subsequent additions (end, or at index)

no search method (linear with iterator)

subsequent removals OK

Binary tree – balance, retrieve

subsequent insertions and deletions allowable

degrade performance as tree becomes less balanced.

Collection Goals:

provides storage and operations (insert, remove, retrieve, update...)

maintains performance on modifications.

Java Collections:

red-black trees ➔ JCF TreeMap

hashing ➔ JCF HashMap

http://docs.oracle.com/javase/7/docs/technotes/guides/collections/index.html

182 Algorithms

Comparable Interface int compareTo(<E>) returns < 0, 0, or > 0
<E> must implement comparable for most data structure algorithms

Recursion initial (public) ➔ recursive (private) ➔ halting condition

Search compare all values with "target" until found / ! found

linear	O(n)	iteratively compare items 0..n-1 unordered collection
binary	O(log n)	partition search space, compare ordered collection

Sorts compare two values, swap when out of order

insertion	$O(n^2)$	swap i^{th} item with min value (i+1 .. n-1) selection sort variants ....
quicksort	O(n log n)	recursively partition, pivot and swap merge sort

182 Algorithm evaluation

$O(1)$	constant	index into an array, hash functions, pop/push stack, add/remove queue
$O(n)$	linear	search unsorted array or linked list array, list, or BST traversal
$O(n^2)$	quadratic	simple sorts: exchange, selection 2 nested loop through n items: duplicates ! sort
$O(\log n)$	logarithmic	Binary search in a sorted list BST insert, find, Heap insert, remove
$O(n \log n)$	"n log n"	merge, quicksort
$O(a^n)$	exponential where $a > 1$	tower of Hanoi, recursive Fibonacci, permutations

Big 0

Big O is "on order" analysis, efficiency of algorithms

estimate, statement of algorithms *efficiency* growth rate

Time && Memory 🡸 fn (critical operations, frequency of operations)

O(...) 🡸 simplification of fn(critical operation, frequency of operation)

remove constants, assume critical operation(s) are constant

ignore constants $O(n) \leftarrow O(n - 1)$

ignore low order terms $O(n^3) \leftarrow O(n^3 + n^2 + n)$

ignore multiplicative constants $O(n^2) \leftarrow O(7 * n^2)$

combine growth rates $O(n^2 + n) \leftarrow O(n^2) + O(n)$

282 Data Structures

Lectures and Data Structures to be covered

SimulationFrameworkV3: GUI, 2D graphics, synchronized methods

Internal Data Structures:

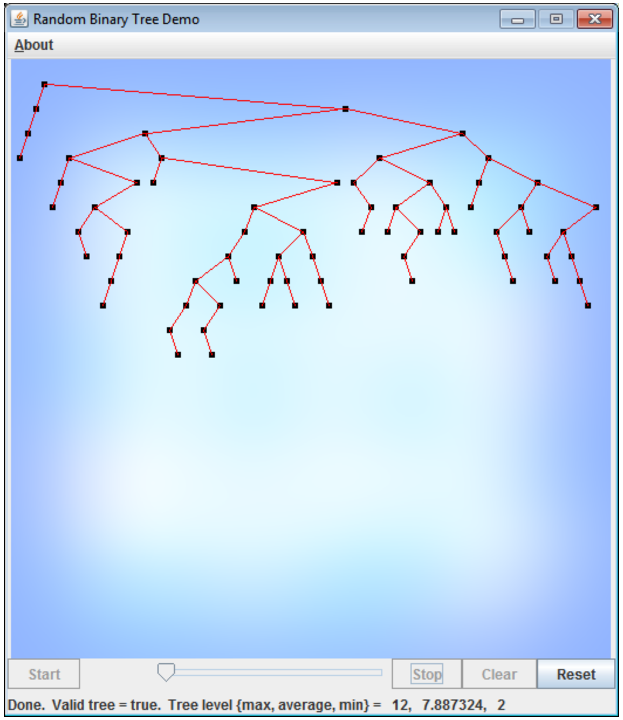
- Hashing JCF HashMap<K, V> (P1, P2)
- Graphs (P2)
- PriorityQueue JCF PriorityQueue (P2)
- AVL Tree
- Red-Black Tree ➔ JCF TreeMap

External Data Structures:

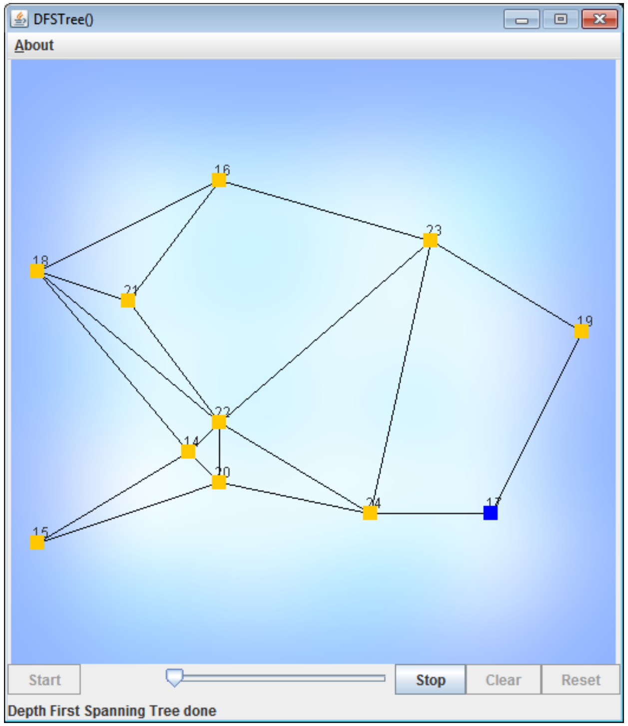
- serialization, random access files, external hashing
- Relational Database MySQL (P3)
- SQL (Structured Query Language)
- B-Tree

Projects 1, 2, and 3 will use JCF and SimulationFrameworkV3
Project 3 will also use MySQL and JDBC (Java database connectivity)

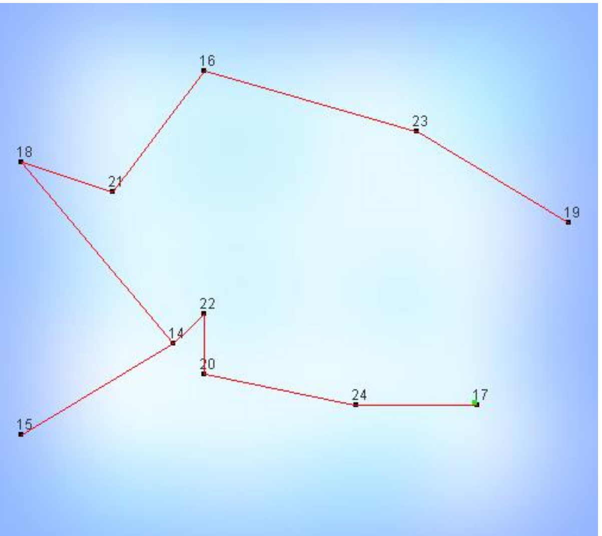
Visualizing Data Structures



random binary tree



graph



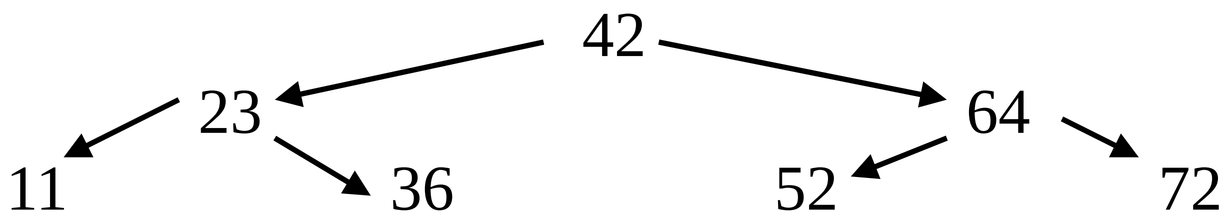
spanning tree

Simulation Framework V3

Balanced binary tree exercise

Design an “high level” algorithm (not a program) to build a balanced binary search tree that contains n unique random numbers.

How could you printout a level-by-level listing of its keys?



42
23 64
11 36 52 72

we’ll discuss in class next time