Data Structures in Java

Session 15

Instructor: Bert Huang

http://www1.cs.columbia.edu/~bert/courses/3134

Announcements

- Homework 4 on website
- Midterm grades almost done
- No class on Tuesday

Review

- Indexing by the key needs too much memory
- Index into smaller size array, pray you don’t get collisions
- If collisions occur,
  - separate chaining, lists in array
  - probing, try different array locations

Today’s Plan

- Rehashing
- Hash functions
- Graphs introduction

Rehashing

- Like ArrayLists, we have to guess the number of elements we need to insert into a hash table
- Whatever our collision policy is, the hash table becomes inefficient when load factor is too high.
- To alleviate load, **rehash**:
  - create larger table, scan current table, insert items into new table using new hash function

When to Rehash

- For quadratic probing, insert may fail if load > 1/2
  - We can rehash as soon as load > 1/2
  - Or, we can rehash only when insert fails
- Heuristically choose a load factor threshold, rehash when threshold breached

Rehash Example

- Current Table:

0	8	7	17	25		
0	1	2	3	4	5	6

- quad. probing with h(x) = (x mod 7)
  8, 0, 25, 17, 7
- New table
  - h(x) = (x mod 17)

0	17						7	8	25							
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Rehash Cost

- No profound algorithm: re-insert each item
- Linear time
- If you rehash, inserting N items costs $O(1)*N + O(N) = O(N)$
- Insert still costs $O(1)$ amortized

Hash function design

- Spread the output as much as possible
- Consider function **h(x) = x mod 5**
- What if our keys are always in tens?
- Less obvious collision-causing patterns can occur
- i.e., hashing images by the intensity of the first pixel if images have border

Hashing a String

- Simple but bad h(x)
  - add up all the character codes (ASCII/ Unicode)
- ASCII 'a' is 97
- If keys are lowercase 5 character words, h(x) > 485

Hashing a String II

- Weiss: Treat first 3 characters of a string as a 3 digit, base 27 number
- Once again, ‘a’ is 97, ‘A’ is 65

String.hashCode()

- Java's built in String hashCode() method
    - `s[0]*31^(n-1) + s[1]*31^(n-2) + ... + s[n-1]`
- nth degree polynomial of base 31
- String characters are coefficients

Hash Function Demo

Built-in Java HashSet

- HashSet stores a set of objects, all hashed by their `hashcode()` method
- `HashSet<String> table = new HashSet<String>();`
- `table.add("Hello");`
- `table.contains("Hello"); // returns true`

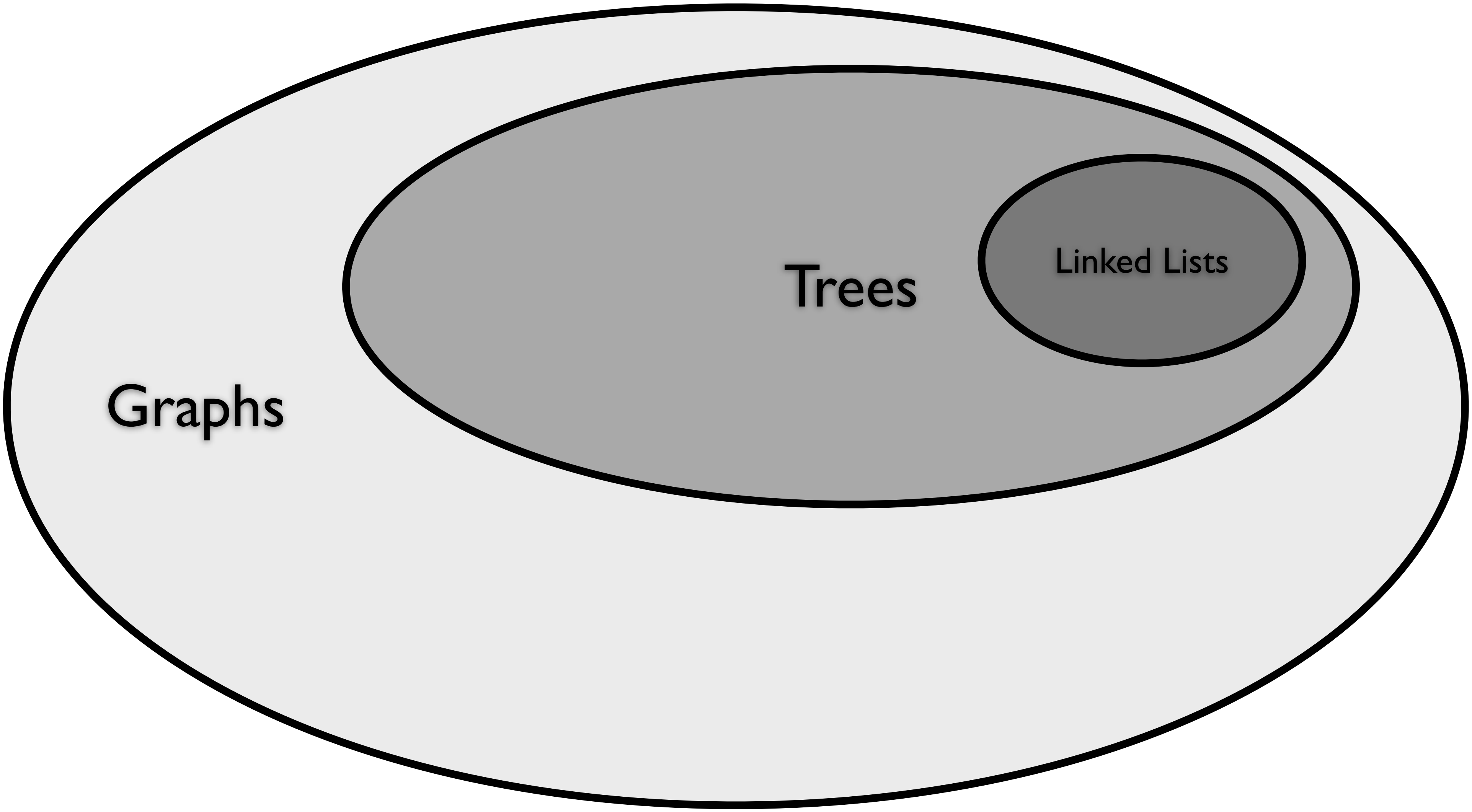
Built-in Java HashMap

- HashMap stores set of <u>pairs of objects</u>,
    - First object is the **key**, second is the value. Hashed by key's `hashcode()`
- HashMap<String,Integer> table = new HashMap<String,Integer>();
- table.set("hello", 42); // pairs "hello" to 42
    - if "hello" is not already in the table, creates new pair. Otherwise, overwrites old Integer
- table.get("hello"); // returns 42

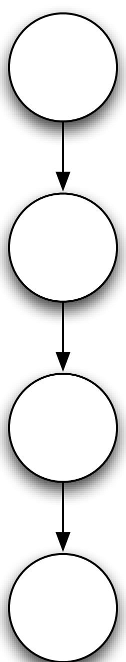
Hashed File Systems

- Gmail and Dropbox (for example) use a hashed file system
- All files are stored in a hash table, so attachments are not stored redundantly
- Saves server storage space and speeds up transactions

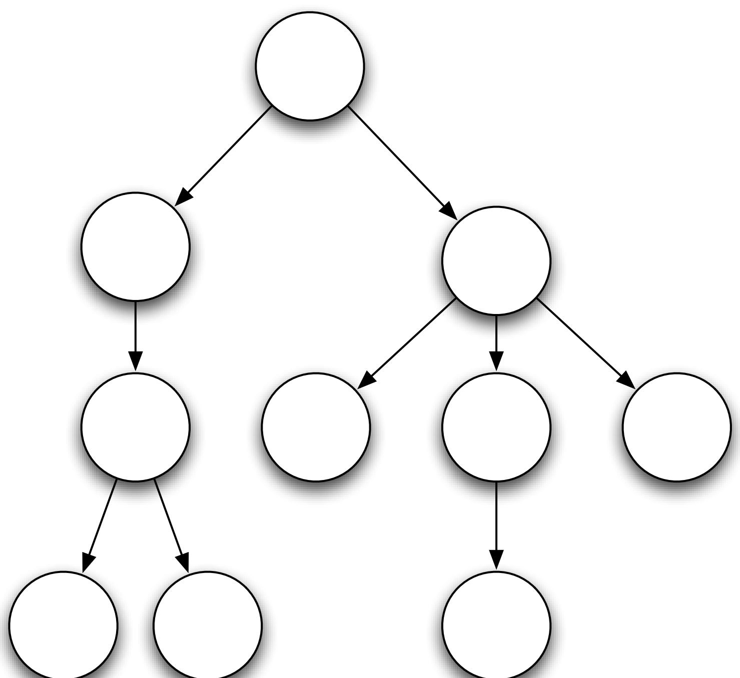
Graphs



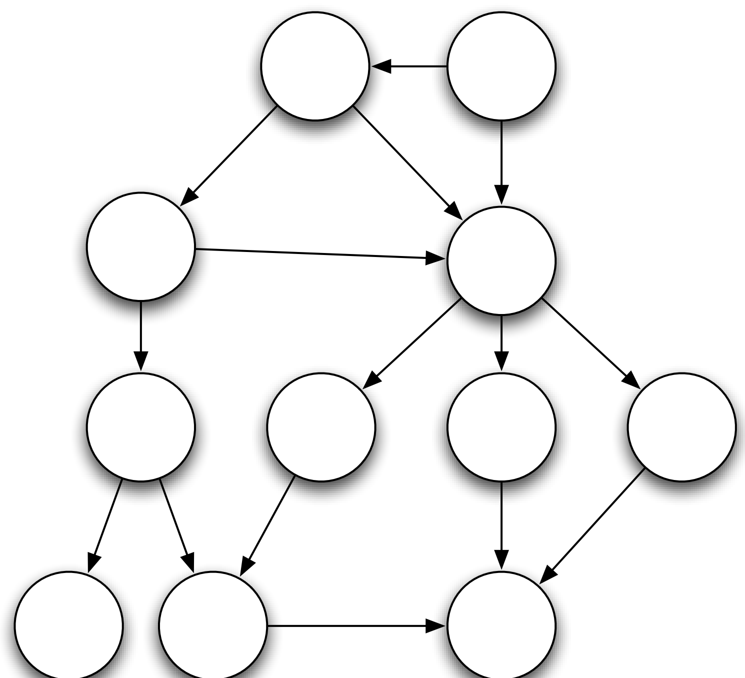
Graphs



Linked List



Tree



Graph

Graph Terminology

- A **graph** is a set of **nodes** and **edges**
  - nodes aka vertices
  - edges aka arcs, links
- Edges exist between pairs of nodes
  - if nodes x and y share an edge, they are **adjacent**

Graph Terminology

- Edges may have **weights** associated with them
- Edges may be **directed** or **undirected**
- A **path** is a series of adjacent vertices
  - the **length** of a path is the sum of the edge weights along the path (1 if unweighted)
- A **cycle** is a path that starts and ends on a node

Graph Properties

- An undirected graph with no cycles is a tree
- A directed graph with no cycles is a special class called a **directed acyclic graph (DAG)**
- In a **connected** graph, a path exists between every pair of vertices
- A **complete** graph has an edge between every pair of vertices

Graph Applications: A few examples

- Computer networks
- The World Wide Web
- Social networks
- Public transportation
- Probabilistic Inference
- Flow Charts

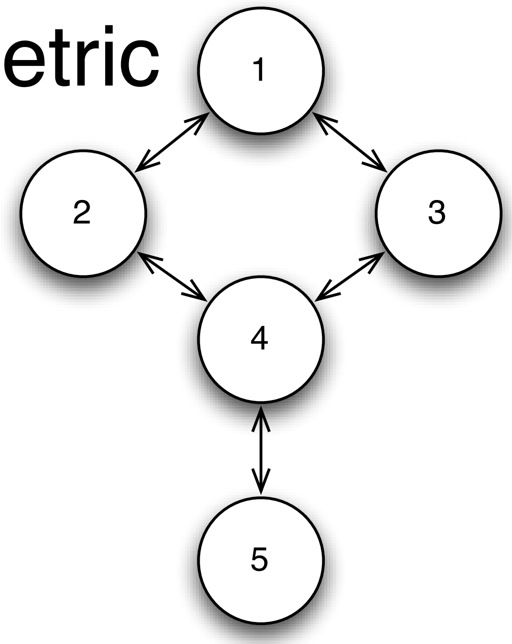
Implementation

- Option 1:
  - Store all nodes in an indexed list
  - Represent edges with **adjacency matrix**
- Option 2:
  - Explicitly store **adjacency lists**

Adjacency Matrices

- 2d-array **A** of boolean variables
- A[i][j] is true when node **i** is adjacent to node **j**
  - If graph is undirected, A is symmetric

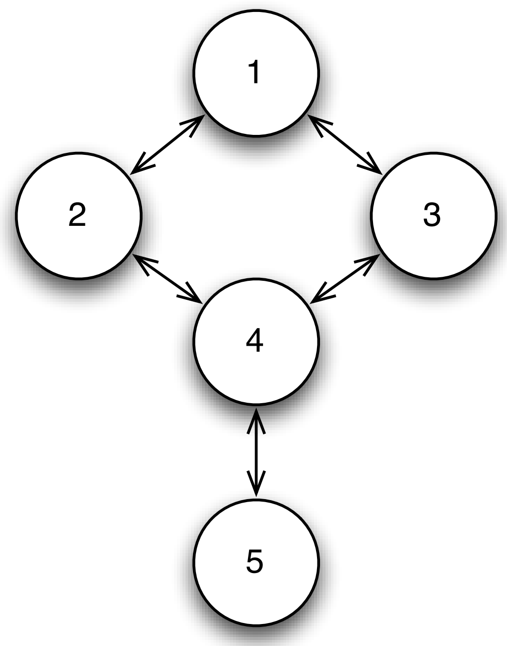
	1	2	3	4	5
1	0	1	1	0	0
2	1	0	0	1	0
3	1	0	0	1	0
4	0	1	1	0	1
5	0	0	0	1	0



Adjacency Lists

- Each node stores references to its neighbors

1	2	3	
2	1	4	
3	1	4	
4	2	3	5
5	4		



Reading

- Weiss Section 5 (Hashing)
- Weiss Section 9.1