- delete node from bst
- hashing
- intro to graphs

Next week and homework

- next week: sorting
- homework by next week:
  - (there are no example programs this week)
  - read Horstmann, sections 14.1 – 14.5, 17.6 – 17.7

Lab

- second part is added to current lab. See Canvas for due date

Review last week

- did simple implementation of binary search tree first
- implemented all the traversals, using recursion
- did a full generic implementation of bst, with an interface, for the current lab

Introduction to this week

- will finish trees, showing how to delete a node from a bst
- begin looking at the next major topic of search
  - we've seen that binary search is much faster than sequential search
  - 'hashing' is a different search technique, potentially faster than binary search
  - re-introduce 'Big-Oh' notation – is how we measure, compare and categorize different algorithms
  - will add hashing to the current lab
- will introduce our final data structure - graphs

Delete node from bst

Objective: we've already seen how to build and search a bst. For practice, will now see how to remove an old node

The three scenarios

- there are three possibilities when removing an old node from a bst
  - (will assume here that a reference has been set to the node to be deleted)

1. delete a node with no children e.g.

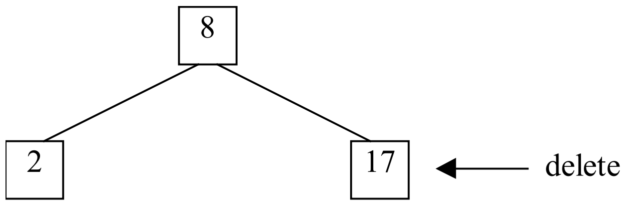

Figure 1 delete a node with no children

   - easy!
   - update the parent

2. delete a node with one child e.g.

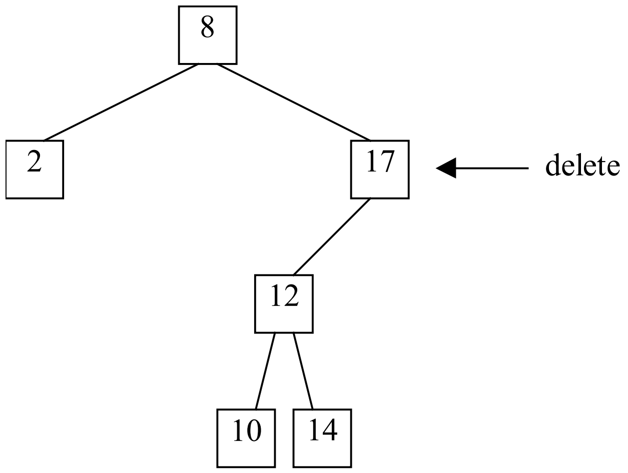

Figure 2 delete a node with one child

   - easy!
   - update the parent to point to the child

3. delete a node with two children

   – this is tricky! e.g.

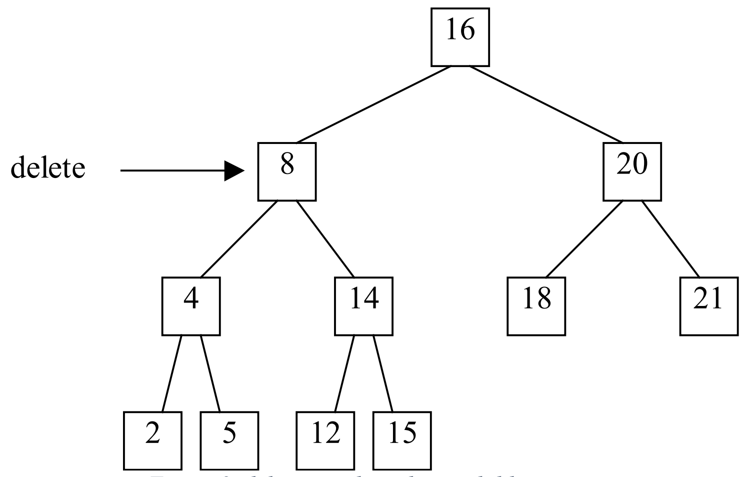

Figure 3 delete a node with two children

   – to which of the subtrees should the parent point, and what happens to the other subtree?

<u>A possible algorithm</u>

- will start with one (bad) approach:

  Step 1: attach right subtree in place of deleted node
  Step 2: attach left subtree to appropriate node of right subtree

  – this gives:

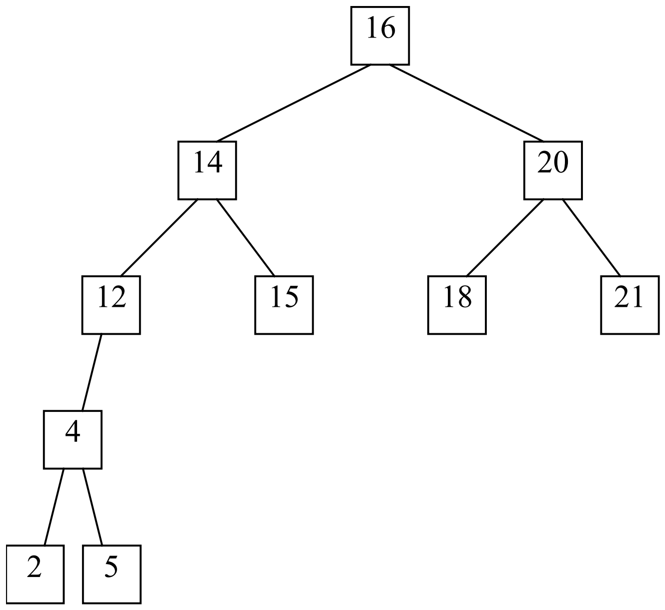

Figure 4 not too bad

- which is not too bad!…

- …however, try the two steps with the following:

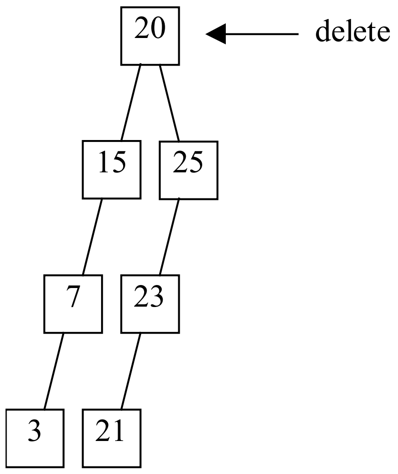

Figure 5 another example

- Step 1 gives:

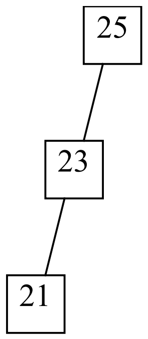

Figure 6 Step 1

– Step 2 gives:

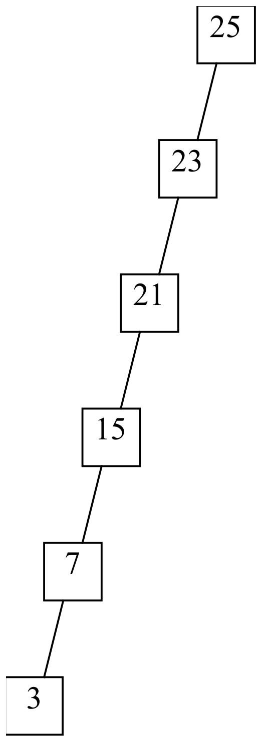

Figure 7 Step 2

– a 'degenerate tree'… really bad for binary search!

<u>A better node delete algorithm</u>

- so we need a better delete algorithm that <u>tends not to increase the height or depth of the binary tree</u>. Here it is:

  – "replace deleted node with its <u>inorder successor</u> from the tree"

  – 'inorder successor' is the node which appears next in an inorder traversal

- e.g. with previous example tree:

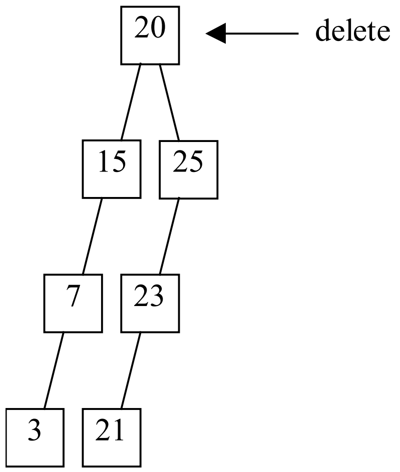

Figure 8 previous example with a better algorithm

- remember, inorder traverses bst in ascending order…
- …so the inorder successor of 20 will be 21 here
- this replaces node being deleted, gives:

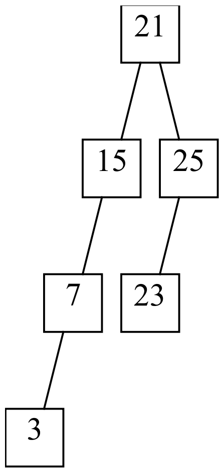

Figure 9 depth is not increased

- much better – depth is not increased!

• let's do another example of this delete algorithm:

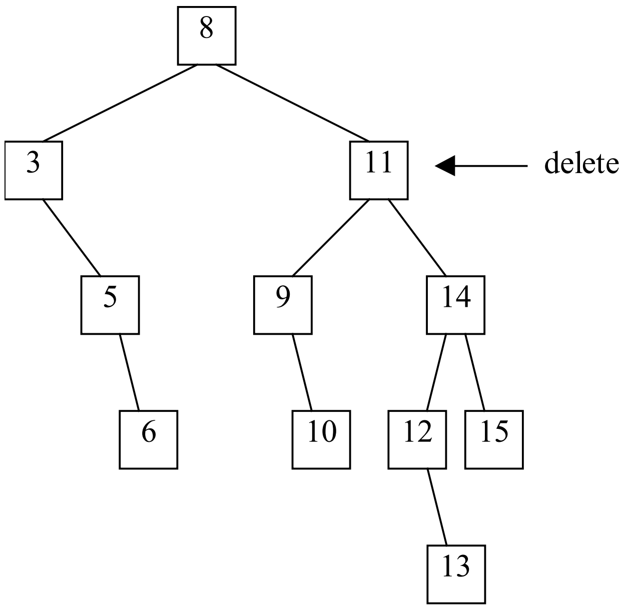

Figure 10 another example

- Q: inorder successor of 11 here is?
- A: 12
- this replaces node being deleted, there's some tree re-arrangement, giving:

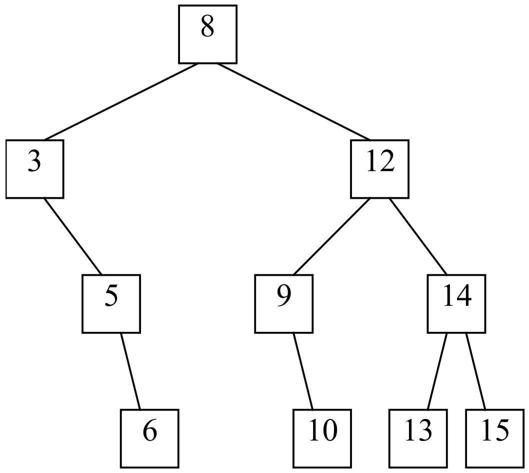

Figure 11 depth of tree is reduced

- depth of tree is reduced

Summary

- for practice, reviewed the design of the algorithm to delete a node from a bst

Hashing

Objective: begin the next major topic of 'searching', and a potentially very efficient search technique called 'hashing'

We use 'big-Oh' notation O() to measure algorithm efficiency

- we've seen some search algorithms and compared their efficiencies:

  sequential
  binary

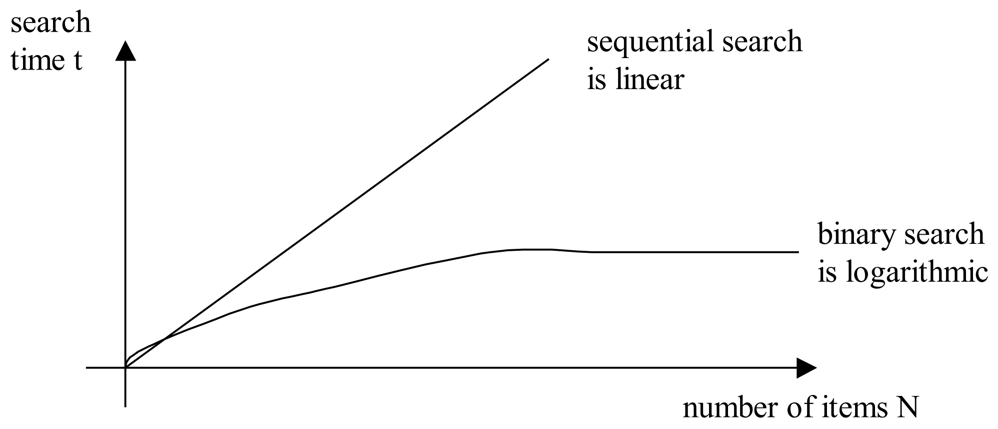

Figure 12 comparing different search algorithms

- efficiency of a search algorithm is expressed using big-Oh notation O() – the 'order of complexity' of the algorithm. We say:
  - sequential O(N) – "is of order N"
  - binary O(logN) – "is of order logN"
  - more on big-Oh next week

Hashing is potentially very fast

- a different search technique called hashing is potentially independent of N!
  - maybe 1 look, no matter how big N is…
  - …no more than 2, or 3, or 4!
  - (otherwise you're doing something wrong)
- so is very fast!

- hashing O(1) of constant order
- ideally, just 1 look no matter how big is N!

How does hashing work?

- idea of hashing is to "map from an object's key directly to its location" e.g.
  - imagine an employee object with SSN as key field
  - imagine an array indexed by SSN(!)

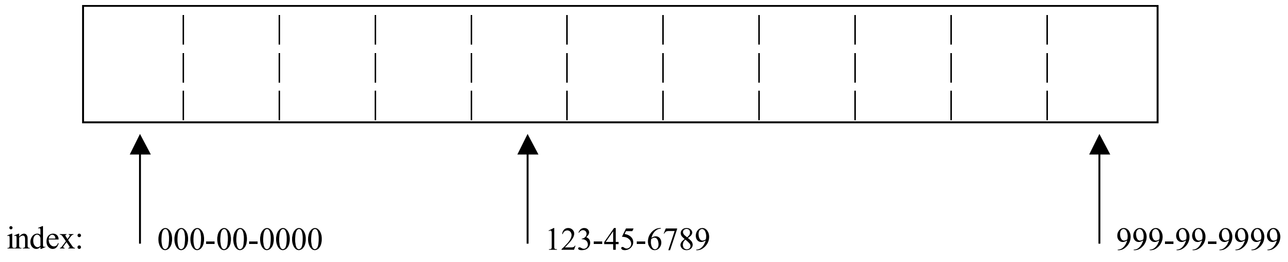

Figure 13 an array indexed by SSN(!)

  - so, retrieving an employee record by SSN – we're guaranteed access in 1 look!
  - extremely fast!
- disadvantage of this illustration of hashing?
  - we can't have an array this big!
  - if employee object is 1 000 bytes
  - requires 1 000 * 1 000 000 000 = 1 000 000 000 000 bytes of storage!

Use a hash table

- so we use a hash table, to reduce the amount of memory required, e.g.
  - instead of 1 000 000 000 locations…
  - …let's say we have only 100 locations:
  - labelled 0 – 99 here:

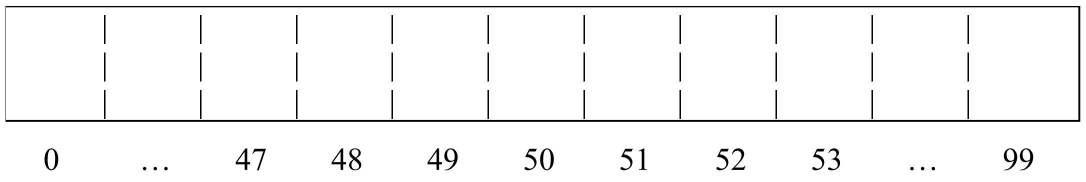

Figure 14 a hash table

- and now use only the last 2 digits of social as index e.g.
- 123-45-67<u>89</u> maps to location 89
- 987-65-43<u>21</u> maps to location 21

<u>Hash functions</u>

- so here, we apply a <u>hash function</u> to the key, to map directly to hash table location
  - a simple hash function is '<u>key mod tablesize</u>'
  - hash(key) = key % 100
  - hash(123-45-6789) to 89

<u>A 'collision' is where different keys hash to the same location</u>

- problem with hash tables?
  - what happens when "different keys hash to the same location"?
  - e.g. hash(111-11-1189) – hashes to the same location, 89
  - so a completed different key value hashes to the same hash table location
  - this is known as a collision
- at least 3 different ways to handle collisions
  - will look at each, using the example of a hash table with 7 locations (table size is 7):

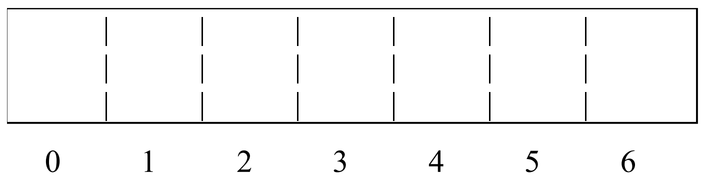

Figure 15 a hash table with 7 locations

- hash function: hash(key) = key % 7
- inserting keys: 3 10 4 6 13
- then will search for: 10 7
- and consider how to delete an object

<u>Linear probing</u>

- with <u>linear probing</u>, "if there's a collision, put new key into next available place", e.g.
  - inserting keys: 3 10 4 6 13
  - gives:

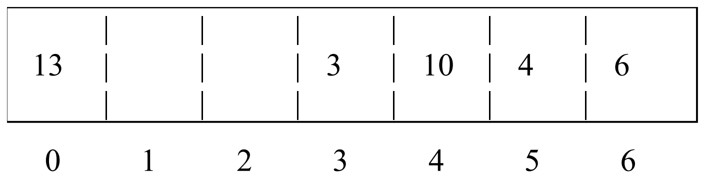

Figure 16 linear probing example

- how does a lookup work here?
  - hash the key, then sequential search from here until found or an empty location
  - HINT: when handling collisions, be careful to wrap the index around from the last element of the array to the first, if necessary
  - e.g. work through finding 10, 7, 13
- be careful when deleting an object!
  - e.g. if we delete 3 – still have to be able to find 10!

– so overwrite a deleted item with a special value such as –999, which means continue linear probing

Rehashing

- with rehashing, "if there's a collision, apply a different hash function", e.g.

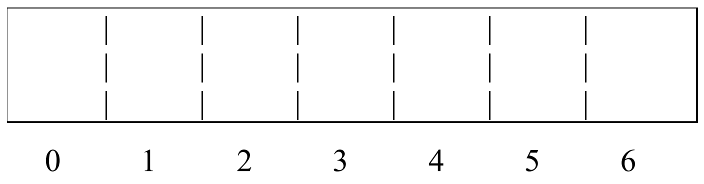

Figure 17 a hash table with 7 locations

– first hash function: hash(key) = key % 7

– second hash function: hash(key) = (key * key) % 7

– inserting keys: 3 10 4 6 13

– gives:

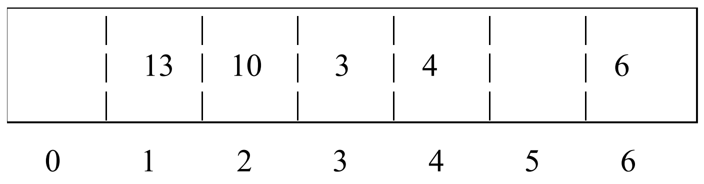

Figure 18 rehashing example

- how does a lookup work?

  – hash the key, then rehash if necessary until found or an empty location

  – e.g. find 10, 7

- what if a rehash collides???

  – could rehash again, and do on

  – but eventually have to fall back to linear probe!

Chaining

- with chaining, "each location in the hash table holds a linked list of all keys that hash to that location"

  – here, each location in the hash table is known as a bucket, each bucket is effectively of unlimited size e.g.

  – hash table is a table of references to lists:

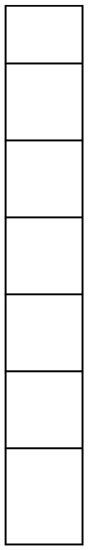

Figure 19 hash table of 7 buckets

  – each element in a list holds a key, a reference to the next element, and the location on disk of the object (call this the Disk Block Address)

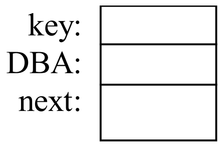

Figure 20 an element in a list

  – so a hash table with 7 locations (table size is 7) is:

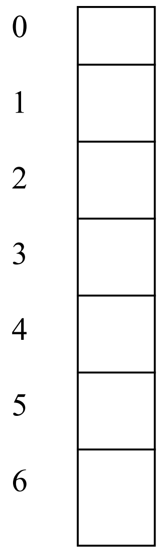

Figure 21 hash table of 7 buckets

- hashing function: hash(key) = key % 7
- inserting keys: 3 10 4 6 13
- gives:

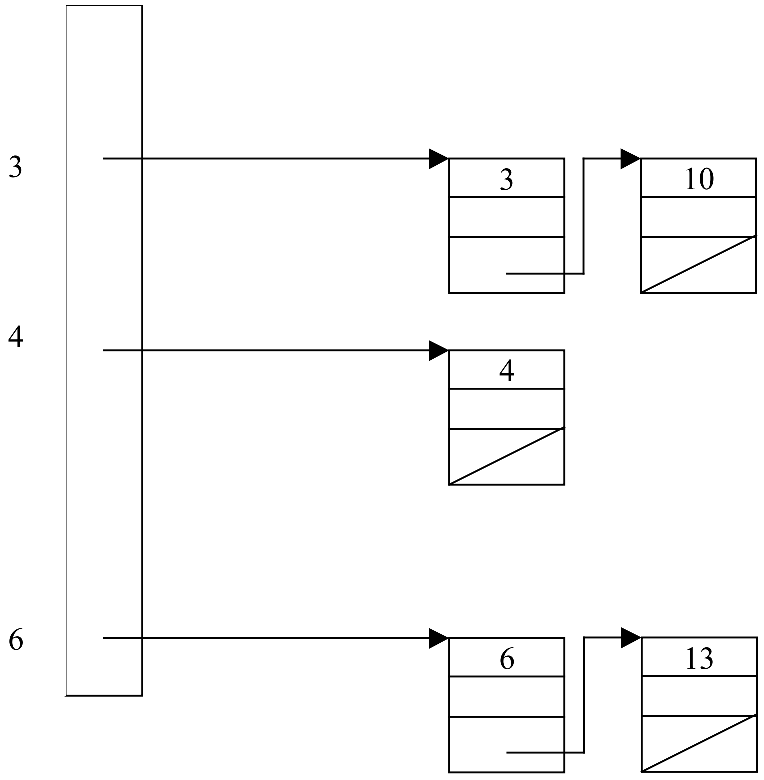

Figure 22 chaining example

- how does a lookup work?
  - hash the key, then sequential search within bucket
  - e.g. search for 10, 7
  - (BTW, could implement the linked list as binary search tree for faster searches within buckets!)

Hash table optimum size

- search time for hashing is relatively independent of hash table size…
  - is actually limited instead by how full the table gets – called the '<u>load factor</u>'
  - typically, a hash table should never get more than 80% full – a load factor of 80%
  - best performance is with load factor around 10%
  - also, research says to make hash table size a prime number, to avoid clustering
  - so estimate your max number of keys, then set hash table size to a prime number 10 times this
- ensure that the hash table is small enough to fit into main memory, for fast access
  - so the hash table contains location of the objects on disk (DBAs), as shown for chaining above, rather than the objects themselves
  - (typically the entire data file of objects is too big to fit in memory)

An example hash function for words

- we want to choose a hash function that evenly distributes hashed keys over the hash table, i.e. avoids '<u>primary clustering</u>'
  - there are very many different hash functions, some examples:
  - we've seen a very simple remainder-based one
  - could use a name as a key field – hash based on ASCII values, e.g.:

	S	m	i	t	h		
ASCII:	83 +	109 +	105 +	116 +	104	=	517 then mod tablesize

- or weight by character position within word:

```
*    *    *    *    *
1    2    3    4    5
```

- and so on and on and on

• here's an example hash function that works well for words

  - add each character's Unicode value to a running total, multiplying by some constant (such as `Math.PI`) after each addition

  - discard the integer part of the final result

  - multiply by the size of the hash table (`MAX`) and discard the remainder

• see my implementation in Java:

```java
public int hash(String word)
{
    char ch;
    double total;

    total = 0.0;
    for (int i = 0; i < word.length(); ++i) {
        ch = word.charAt(i);
        total += (double) ch;
        total *= Math.PI;
    }
    double decimal = total - (int) total;
    int hash = (int) (MAX * decimal);
    return hash;
}
```

  - will use this hash function in the lab

<u>A perfect hash function</u>

• a perfect hash function for a set of keys generates a unique hash value for each key

  - so guaranteed never any collisions, the perfect scenario

  - is only possible with a known, limited set of keys…

  - e.g. compilers use hashing to identify their small, known, limited set of reserved words

Summary

- hashing potentially allows very fast search
  - the big idea is to reduce the number of collisions
  - and choose a hash function appropriate to your key values, that gives a good, broad distribution of mappings across the hash table
- BTW, a disadvantage of hashing
  - can't easily access data in sorted order
- will add hashing to the current lab
  - to quickly identify a small, known set of commonly-occurring, uninteresting words in the text
  - e.g. a, after, all, and, because, every, etc etc

Intro to graphs
Objective: our final data structure. Will introduce primitive operators, traversals and an example application, but no implementations

<u>A graph consists of vertices connected by edges</u>

- a graph consists of "a set of nodes called vertices and a set of lines called edges or arcs that connect the vertices"
    - there are no restrictions on how the edges connect the vertices
    - the graph represents information in terms of its edges between vertices, e.g. say routes between destinations here:

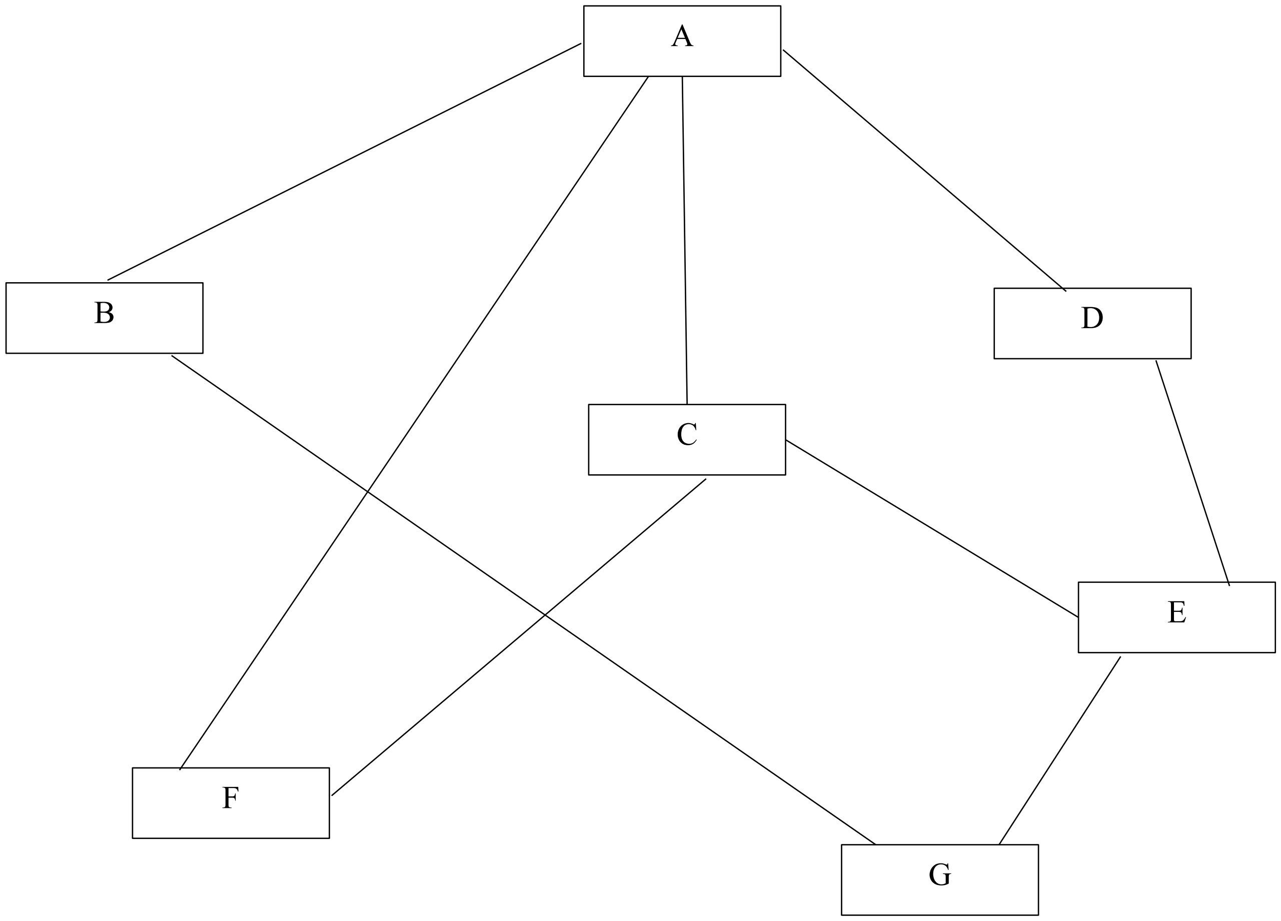

Figure 23 an undirected graph

- these edges are 'undirected', so can travel from A to B and B to A

Weighted graphs

- a weighted graph is a graph "where each edge has a value"

  – e.g. a weighted graph of the flights between cities that an airline offers, where the value of an edge shows the air distance between pairs of cities (not all edges have been labelled)

  – also, these edges are now 'directed'. Can travel directly from Dallas to Denver, but not directly from Denver to Dallas

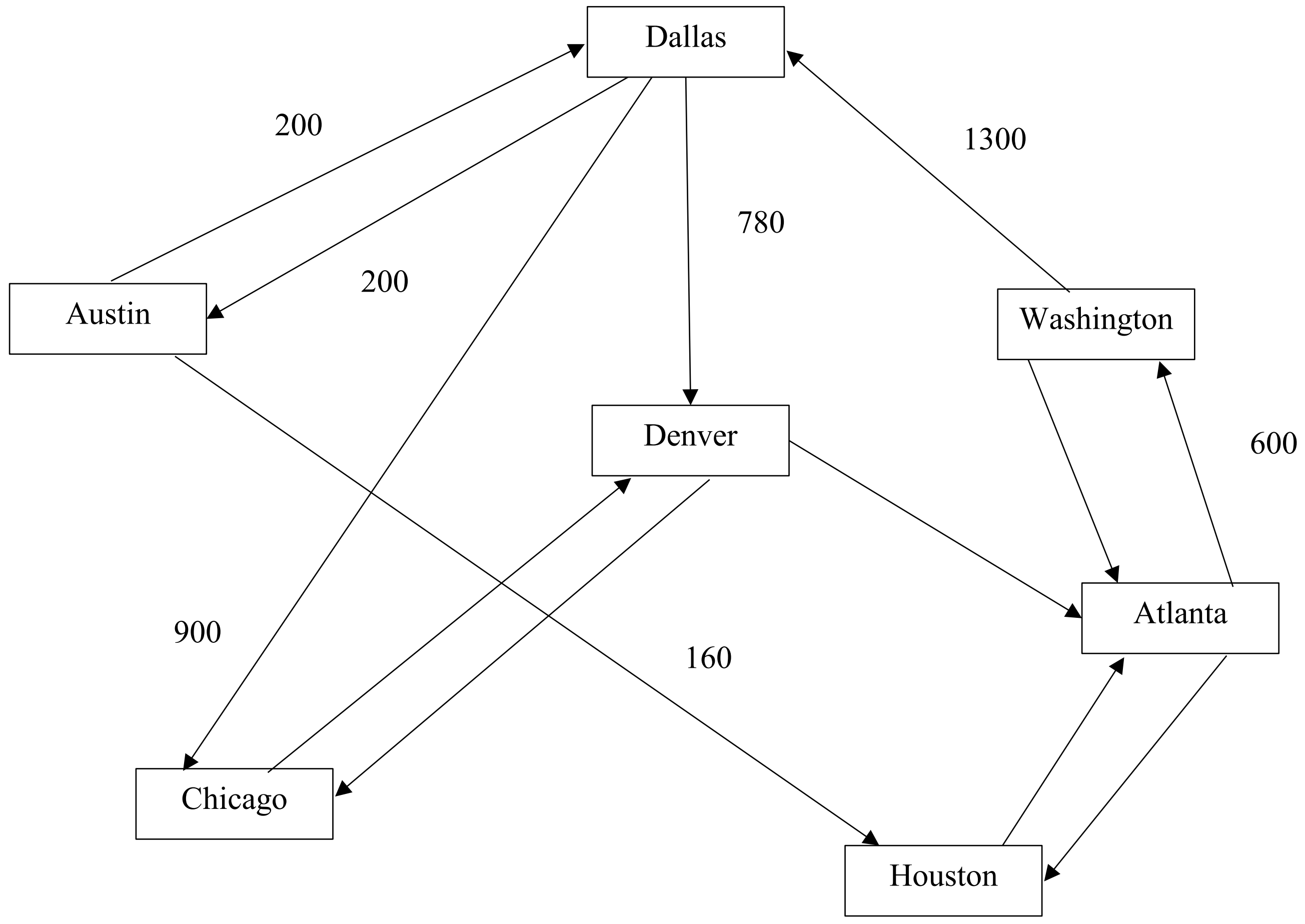

Figure 24 a directed graph

- we use weighted graphs for finding and comparing different routes between vertices

  – will see different graph traversals to find routes

Primitive operations

- there will be the expected primitives, such as:

  - addVertex() – add a vertex to the graph
  - addEdge() – add an edge between two vertices and its weight
  - hasVertex() – search the graph for a vertex
  - weightIs() – return the weight of an edge

- then a very important operator used when finding routes: getToVertices()
  - takes a vertex
  - returns a list of adjacent vertices i.e. one hop from this vertex
  - this idea is essential, as we build routes from one vertex to another
- we also need to be able to 'mark' a vertex as visited, so that we don't get trapped in infinite circular routes, so:
  - clearMarks() – clear all marks from the graph, to start building a new route
  - markVertex() – mark a vertex as visited
  - isMarked() – return whether this vertex has been visited

<u>Graph traversals</u>

- different styles of traversal through the graph can be used to answer import questions about routes
  - in a binary tree, the 'depth-first' traversals we have seen repeatedly go as deep as they can in a tree and then back up
  - we have not seen a 'breadth-first' traversal, where the traversal fans out across the tree level by level
  - similar traversals are possible with graphs
- depth first graph traversal
  - from the starting vertex, we use getToVertices() and push to a stack all of the adjacent places we can get to with one hop

- then we pop the stack for the top place, and repeat the process, until either we reach the target, or a dead end
- in this strategy, we back up as little as possible when stuck, by popping the next alternative from the stack, and so on
- (must be careful to use marking to avoid a literal infinite loop)
- in this way we traverse depth-first through the graph until reaching the destination without regard to the number of hops

- breadth-first graph traversal
  - a breadth-first traversal looks at all the possible paths at the same depth, before it goes to a deeper level
  - would use queues to implement this, where every element in every queue is checked before moving on to the next level of queues if necessary
  - in depth-first search we back up as little as possible when we reach a dead end, in breadth-first we back up as far as possible
  - so breadth-first would be preferred in our airline application, since it minimizes the number of hops between destinations
- both these styles of traversal can be used when searching for routes in graphs, you choose an approach based on the graph, and the information required
  - use depth-first if there are several solutions, far from the start
  - use breadth-first to minimize the number of hops, as here
- BTW, notice that these traversal algorithms are just brute force, with backtracking. There's no intelligence here, just exhaustive search
  - some other important graph algorithms to be aware of:

<u>Dijkstra's algorithm</u>
- Dijkstra's algorithm finds the shortest paths in a weighted graph
  - e.g. in our airline example, searching for paths from Austin to Washington, say:

Austin to Houston is 160 miles +	Austin to Dallas is 200 miles +
Houston to Atlanta is 800 miles +	Dallas to Denver is 780 miles +
Atlanta to Washington is 600 miles	Denver to Atlanta is 1400 miles +
total: 1560 miles	Atlanta to Washington is 600 miles
	total: 2980 miles

Figure 25 comparing paths

- fewer total miles, so the first here is a shorter path than the second

- depending on implementation detail, algorithm is something like O(V^2), where V is the number of vertices

Floyd-Warshall algorithm

- the Floyd-Warshall algorithm finds the shortest path for all pairs in a weighted graph
  - so output is a matrix of the minimum distances from any node to all other nodes in the graph
  - algorithm is O(V^3), where V is the number of vertices

Summary

- a graph is less constrained than a tree, since any node can connect to any other
- very versatile, an important way to model many real-world situations
- many advanced algorithms have been invented to process graphs

Next week and homework

- next week: sorting

- homework by next week:

  - (there are no example programs this week)

  - read Horstmann, sections 14.1 – 14.5, 17.6 – 17.7

Lab

- second part is added to current lab. See Canvas for due date