FINAL EXAM REVIEW

CS 200 RECITATIOIN 14

I don't *think* the final is cumulative. Sangmi will say for sure during her review sessions. Either way, keep in mind that this course's material is naturally cumulative because later concepts often build upon earlier ones. For example, graph traversals use stacks and queues and Dijkstra's algorithm commonly uses a priority queue.

Table

A table is a value oriented abstract data type:

Student ID	First Name	Last Name	Exam 1	Exam 2
001245	Jake	Glen	67	89
001247	Parastoo	Mahgreb	87	78
001256	Wayn	Dwyer	90	96
012345	Bob	Harding	95	97
022356	Richard	Nixon	95	50

- Data is arranged by Search Key
  - Student ID is the search key in this table
  - Search key must be unique (it identifies a single record)
  - Implementation designed to retrieve records using the search key
  - Search key must not change once it is established
- Each column is a filed
- Each row is a record
- Possible implementations:
  -

	Search	Add	Remove
Sorted Array	O(log n)	O(n)	O(n)
Unsorted Array	O(n)	O(1)	O(n)
BST	O(log n), [O(n)]	O(log n), [O(n)]	O(log n), [O(n)]

    * values in square brackets are worst case costs
- Hashing techniques provide a better performing table implementation
- Used when someone needs to store a bunch of records, random access is important and they have a piece of data suitable for use as a unique search key.
- Operations
  - add a record
  - retrieve a record
  - delete a record

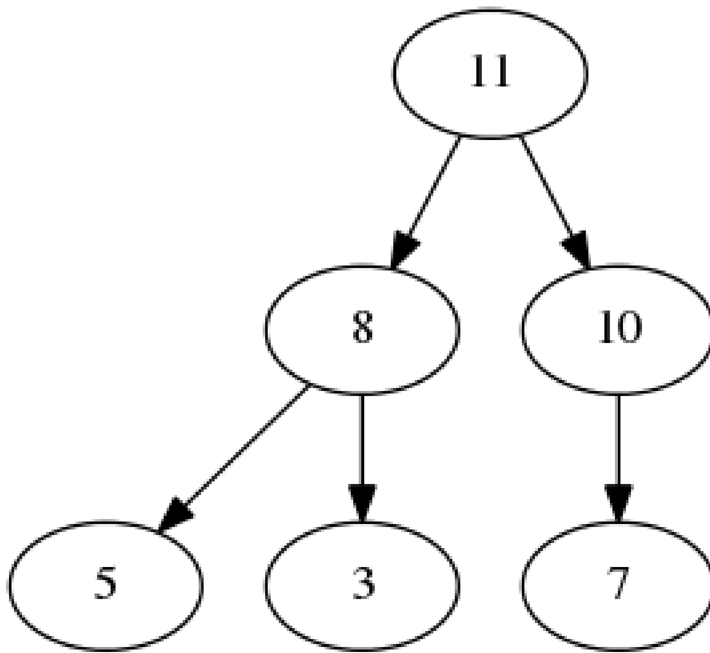
Priority Queue

Another value oriented ADT.

- It's a queue with one big difference: an element's place in line is determined not by when it was added, but by it's priority value. Higher priority elements are higher in the queue and will be retrieved before lower priority elements.
- Uses:
  - Operating systems
    - sometimes used when scheduling the processor
  - Networks
    - traffic is commonly assigned priorities, and network devices need to give higher priority traffic a better place in line
- Implementations:
  - Sorted array
    - may need a lot of shifting of elements
  - Sorted linked list
  - Binary Search Tree
    - Highest value is easily found (all the way to the right, has at most one child)
  - Heap
    - preferred in most cases
- Operations:
  - enqueue
  - dequeue
  - peek

Heap

Yet another value oriented abstract data type.



- Heaps are complete trees which follow the heap property
  - root contains a key greater than or equal to the keys of it's children
    - This is a maxheap. A minheap has the same idea but the above inequality is inverted: root's key is less than or equal to the keys of it's children.
  - left and right sub trees are also heaps
  - Heap property implications for a maxheap:
    - root has the highest value in the heap
    - values are in descending order along every path from root to a leaf
  - Implications for a minheap are similar:
    - root has the lowest value in the heap
    - values are in ascending order along every path from root to a leaf
- NOT a Binary Search Tree! Just a regular, but complete, tree
- Can be stored in an array efficiently:
  - because it's a complete tree
  - eliminates need for a separate 'node' class, the location of parent and child nodes can be directly computed.
  - Think of concatenating each row into the array. Above tree is:
    - [11, 8, 10, 5, 3, 7]
  - Array may be longer than number of elements in the tree, simply keep track of where the last one is. There will be no gaps.
  - How to compute addresses (array indices) relative to node N, which is at index I:
    - N's left child: $2I + 1$
    - N's right child: left child + 1 or: $2(I + 1)$
    - N's parent node: $\lfloor (I + 1)/2 \rfloor$, doing an integer division will accomplish that floor operation nicely.

- Root is always at index 0

- Operations
  - insert: add new node, by storing it in the first open slot in the array.
    * new node may be out of position (violating the heap property), swap with parent node as necessary until it is in position.
    * Usually called "percolating up", or "trickling up"
    * an added node may percolate up far enough to become the new root.
  - delete: behaves much like a stack's pop operation: root is removed from tree and returned. It's always the root because that is the only node that has an absolute position (the other nodes don't have to be in order, just less than the root)
    * swap root with right-most leaf, then delete leaf. Much like tree deletes. This keeps the tree complete and prevents the array from ever having to shift
    * after swapping, the heap will no longer be a correct heap (it will be psudoheap), we need to restore it's heap property. This is done by swapping the root with it's larger child as many times as is needed until it ends up in the right place

- Complexity
  -

	average	worst case
insert	O(log n)	O(log n)
delete	O(log n)	O(log n)

- Uses:
  - priority queues!
  - sorting

- Heapsort:
  - store all elements to be sorted in a heap
  - iteratively call delete on the heap until it is empty, elements will come out in sorted order
  - can be done in place: use part of the array for the heap and the rest of it for the sorted list

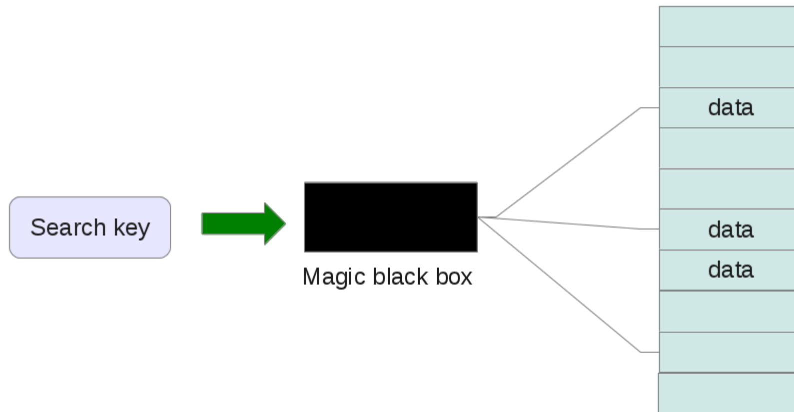
Hashing

Hashing is is not a data type by itself, but a technique for building data types like hash maps and hash tables. Recall the previous discussion of the Table ADT, its performance is generally a bit underwhelming (OK, the sorted tree based implementation is not bad, but we can still do better).

-

	Search	Add	Remove
Sorted Array	O(log n)	O(n)	O(n)
Unsorted Array	O(n)	O(1)	O(n)
BST	O(log n), [O(n)]	O(log n), [O(n)]	O(log n), [O(n)]

Notice that the fundamental problem for a table is how to quickly find which record we want to operate on. Enter hashing techniques, which are designed for exactly that:



That magic box, a.k.a. the hash function, can convert a search key into an array index. And do it pretty quickly. Inserting something in a table just got a lot easier:

- call the hash function with the record's search key
- it returns an index
- store element at array[index]

Hash Functions

There are two main types of hash functions:

- Perfect: maps every possible search key to a unique value. Never has a collision, no empty space necessary in the array.
  - only practical if the entire domain of the search key is known beforehand. For example: credit card and telephone numbers.
- Imperfect: maps search keys to addresses, but not necessarily unique ones. Collisions are possible, and some empty space is needed in the array
  - general purpose, these are the most common type of hash function.

Desirable properties of a hash function:

- easy and fast to compute
- values evenly distributed (over the array's indices)

Example hash functions:

- choose some digits from the search key:

  - 001364825, fourth and last digit makes hash of: 35
  - simple and fast, but does not spread values evenly

- folding
  - sum all the digits of the key
  - or group the digits and sum the groups

- modulo arithmetic
  - $x \,\%\, \textit{table size}$where x is the search key. Prime table sizes work the best

Strings can be hashed as well. They are usually converted to a number and then hashed with on of the above hash functions

Collisions

A collision happens when an imperfect hash function tries to put two things in the same index. They are pretty common actually, see the slides about the birthday problem. Dealing with collisions:

- technique 1: open addressing
  - probe for an empty index
  - linear probing: if location h is occupied, check h+1, h+2, etc... until and empty space is found. Same technique used to search for an element.
    - deleting can cause problems: if it removes an element that was previously in a solid block (cluster) of elements, subsequent attempts at linear probing may be fooled into thinking the empty space is the end of the cluster. This is usually solved by marking the index as deleted in some way.
    - susceptible to primary clustering: items tend to cluster together in the array. Large clusters tend to grow.
  - quadratic probing: if location h is occupied: check $h + 1^2$, $h + 2^2$, $h + 3^2$, etc...
    - eliminates primary clustering
    - but causes secondary clustering because the search sequence is the same for every element
  - double hashing: use a second hash function to choose a step (number of spaces) to use for probing

- technique 2: restructure the table
  - make each location able to store multiple elements
  - if array[i] is an array of fixed size, it's called a bucket. Choosing a good bucket size can be tricky
  - if array[i] is a linked list, you are using separate chaining. when a collision happens, simple add the offending element to the end of the linked list in that location.

Performance

Performance of a hash table is dependent on how full it's array is, called the load factor. it is simply the ratio of full spaces to total space in the array. Smaller load factors are better. Most systems try to keep their load factors below 2/3. In the following equations, $\alpha$is the load factor:

- number of comparisons for a search:
  - linear probing

    * successful: $\frac{1}{2}\left[1+\dfrac{1}{1-\alpha}\right]$
    * unsuccessful: $\dfrac{1}{2}\left[1+\dfrac{1}{(1-\alpha^2)}\right]$
  - quadratic probing
    * successful: $\dfrac{-log_e(1-\alpha)}{\alpha}$
    * unsuccessful: $\dfrac{1}{1-\alpha}$
  - chaining
    * successful: $1+\dfrac{\alpha}{2}$
    * unsuccessful: α

Relations

A relation aggregates members of a set into tuples according to some relationship (any one will do!) between those members.

- $\{0, 1, 2, 3\}$ is a set. Set is a list of elements in no particular order with no duplicates.
- $(0, 2)\,(1, 3)$ are two tuples drawn from that set
- $\{(0, 2)\,, (1, 3)\,, (1, 1)\}$ is a relation on that set (relationship = one I made up). Note that this is also a set, a set of tuples.

Properties of Relations

- Reflexive
  - $(a, a) \in R$ for every element $a \in R$. That means that for every member X in the set, the relation must have an (X,X) tuple
- Symmetric
  - $(b, a) \in R$ whenever $(a, b) \in R$. If my relation contains (1,2) it must also contain (2,1) to be symmetric.
- Anti-symmetric
  - if $(b, a) \in R$ then (a, b) is not in R. $a = b$ is a special case, (1,1) is a tuple which is both symmetric and anti-symmetric
- Transitive
  - if $(a, b) \in R$ and $(b, c) \in R$, then $(a, c) \in R$. If I have (1,2) and (2,3) I must also have (1,3) for it to be transitive.

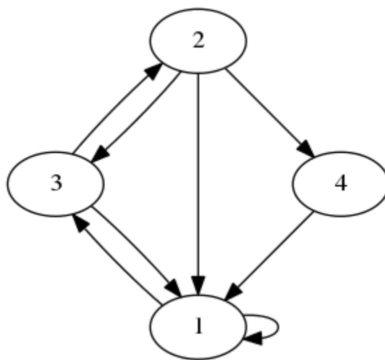
These properties are pretty much all-or-nothing. Either the entire relation is reflexive or it isn't. A single counter example is all that is necessary to remove one of these properties from a relation.

Combining Relations

- Union
  - $R1 \cup R2$. Both sets concatenated, duplicates removed
- Intersection
  - $R1 \cap R2$. Set of elements in both relations
- Subtraction
  - $R1 - R2$. Removes elements common to both relations from R1.
- Composite
  - $R2 \circ R1$. Find tuples in R1 with second elements which are the same as first elements in tuples from R2. Make a new tuple from the unused elements in this process. This new tuple may not be present in R1 or R2, it can be completely new.
  - $\{(1,1), (1,4), (2,3), (3,1), (3,4)\}$= R1
  - $\{(1,0), (2,0), (3,1), (3,2), (4,1)\}$= R2
  - $\{(1,0), (1,1), (2,1), (2,2), (3,0), (3,1)\} = R2 \circ R1$

Representing Relations

- As a Matrix
  - $\{(2,1), (3,1), (3,2)\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$
  - when a relation is expressed as a matrix, the composite can be computed using a technique similar to matrix multiplication called the boolean product. We did not present this technique in class, and you are not expected to know it. If you are given matrices, it is perfectly OK to convert them back into list-of-tuples notation before doing the composite.
- As a Digraph, this is graph of $\{(1,1), (1,3), (2,1), (2,3), (2,4), (3,1), (3,2), (4,1)\}$, though it looks different than the book's version (sorry)



- General form:
  - $\{(a,b) \mid condition\}$
  - (a,b) describes the format of the relation's tuples
  - | - the vertical bar is a separator here, not a logical OR

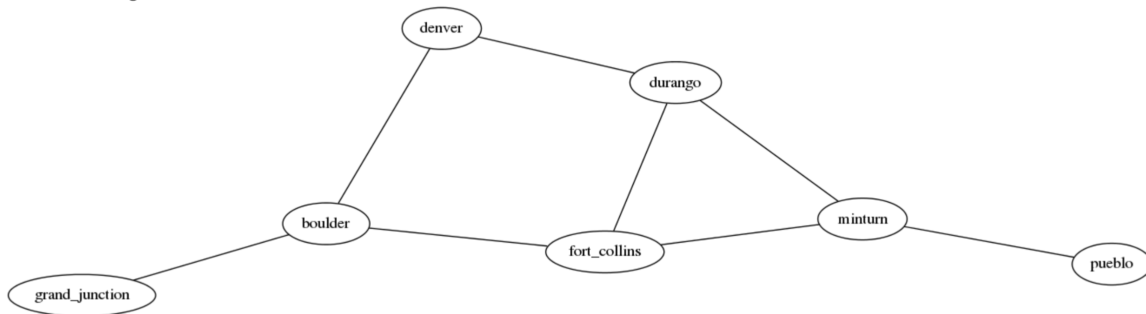
– the condition specifies rules for building tuples in the relation. Multiple conditions can be used with logical operators:
  * $\{(a, b) \mid condition\ or\ condition2\ and\ condition3\}$
  * $\{(a, b) \mid a\ is\ odd\ and\ b\ is\ even\}$
– try to get all the conditions into one set of {} braces.

Closures

A closure on a relation means: add whatever tuples are necessary to make the relation have the specified property. So a reflexive closure on a relation means add whatever tuples are necessary to that relation to make it reflexive

Graphs

Abstract data type with nodes and edges, kind of like a tree except that graphs do not have the hierarchy that trees do. In general trees are viewed as a subset of graphs (they are a graph with extra rules). Graphs look something like this:



Nodes and edges, like a tree, but with out any overall ordering. These edges have no direction, so this graph is undirected. A directed graph's edges have direction (indicated with an arrow), and can only be traversed in the correct direction.

Terminology

- Vertex - a location in the graph (usually containing data)
- Node - another name for a vertex
- Edge - connects vertices
- Adjacent - two vertices (nodes) are adjacent if they are directly connected by an edge
- Incident - an edge is incident on two nodes (the ones it touches)
- Degree - of a vertex, the number of edges incident upon it (how many edges connect to this vertex)
- Self Loop - an edge which connects a vertex to itself
- Simple Graph - no self loops, no two edges connect same vertex pair
- Multigraph - may have multiple edges between same vertex pair
- Psudograph - multigraph with self loops.
- complete graph - simple graph which contains exactly one edge between each pair of distinct vertices
- Cycle - repeatable path through the graph.
- Path - sequence of edges (or sequence of vertices in a simple graph)

Traversals

- Depth first search: go as deep as you can visiting nodes along the way, then backtrack to catch any missed nodes.
  - Psudocode:
  - ```
    dfs(in v:Vertex)
      s - stack for keeping track of active vertices
      s.push(v)
      mark v as visited
      while(!s.isEmpty()) {
        if (no unvisited vertices adjacent to the vertex
        on top of the stack) {
          s.pop() \\backtrack
        else {
          select unvisited vertex u adjacent to vertex on
          top of the stack.
          s.push(u)
          mark u as visited
        }
      }
    ```
- Breadth first search: visit every adjacent node, then go one level deeper and repeat
  - Psudocode:
  - ```
    bfs(in v:Vertex)
      q - queue of nodes to be
      processed
      q.enque(v)
      mark v as visited
      while(!q.isEmpty()) {
        w = q.dequeue()
        for (each unvisited vertex u
        adjacent to w) {
          mark u as visited
          q.enqueue(u)
        }
    }
    ```

Implementation

3 ways to implement a graph ADT:

- Adjacency matrix
  - use a matrix to store which nodes are connected by an edge.
  -

	first	second
first	0	1
second	1	0

  - first is connected to second
  - and second is connected to first

  - matrix[i][j] tells you if there is connection between i and j
  - for today, the row label will be the 'from ' node and the column label will be the 'to' node. This is not set in stone, but I had to pick one.

- Adjacency list

  - each node keeps a list of outgoing edges (if undirected, it's just a list of edges). This stores edge data on the nodes as opposed to the above matrix which stores it independently of the nodes.

- Nodes and References (like out trees from before, not used much)

  - variant of the adjacency list.

Topological Sorting

Graphs are commonly used to describe some sort of precedence relationship between their nodes. The example from the slides about Batman's costume is a good analogy, he has to don his costume in the correct order.

- DAG: directed acyclic graph. Is commonly used with topological sorting (cycles are bad...).
- A topological sort is a listing of nodes such that: if (a,b) is an edges, then a appears before b in the list
- the sort is a path through the graph which illustrates the precedence of the nodes
- depth first search can be modified to do a topological sort:
-
```
topSort2( in theGraph:Graph):List
  s.createStack()
  for (all vertices v in the graph theGraph)
    if (v has no predecessors)
      s.push(v)
      Mark v as visited
  while (!s.isEmpty())
    if (all vertices adjacent to the vertex on top of the
    stack have been visited)
      v = s.pop()
      aLlist.add(1, v)
    else
      Select an unvisited vertex u adjacent to vertex on
      top of the stack
      s.push(u)
      Mark u as visited
  return aList
```

Shortest Path

Dijkstra's shortest path algorithm is pretty dominant. The main idea:

- maintain an array, d, of minimum distance estimates:

  - initi it as:
  - d[start] = 0

  - d[unvisited] = infinity

- make a priority queue of vertices not yet visited
- select minimum distance vertex (delete from the queue), visit it, and update distances to it's neighbors
- for each vertex, store a pointer which tells you the vertex from which you arrived. By traversing backwards along these pointers, kinda like a linked list, you can get the path

We want to compute the shortest path from our start vertex to every other vertex in the graph. Usually described by showing successive states of the array d for each step in the algorithm:

a	b	c	d	e
0	–	–	–	–
0	10/a	5/a	–	–
0	8/c	5/a	7/c	14/c
0	8/c	5/a	7/c	11/d

Spanning Trees

Remember that a tree is a also a graph, it is an undirected connected graph with no simple circuits.

- Spanning tree: A sub graph of a connected undirected graph G that contains all of G's vertices and just enough edges to form a tree
  - remove edges until you have a tree
  - add back any edges necessary to make it reach all vertices

Minimum Spanning Tree

A spanning tree which minimizes the total edge weight of all edges in the tree. Note that these do not guarantee shortest paths, in fact they can make some stupid paths, but they will give you the minimum units of edge weight necessary to connect all vertices. Helpful if you are planning a physical network, like sewer pipes, and construction is expensive; You would want to minimize the total amount of pipe needed.

- Prim's algorithm:
  - weighted graph
  - start at arbitrary vertex, or user's vertex
  - keep list of visited vertices, and a list of visited edges
  - loop until visited list contains the whole graph:
    * Look at all the edges which connect any visited vertex to any unvisited vertex
    * Choose the edge in the above set with the smallest weight
    * add that edge to the list of edges, and the vertex it connects to to the list of visited vertices
  - output the list of visited vertices and the list of visited edges. The list of vertices should contain all vertices in the graph, the list of edges need not contain all edges in the graph.

Other resources:

- answers and solutions to the recitation exercises are available on the RamCT discussion board.
- recitation worksheets are intended as practice and study aids, look up the answers to them if you haven't already
- look over the written assignments and quizzes, they have good example problems
- I think Sangmi is planning to post a study guide
- Lectures this week will be review

Grading:

Attendance only, come sign the sheet.