CS 225 Final Study Guide & Exam Solutions

Table of Contents

Exam Links, Final Details: https://wiki.engr.illinois.edu/display/cs225/Exams
Link to some review slides: https://wiki.engr.illinois.edu/display/cs225/Own+The+Final
[Just Added] Annotated Slides: https://wiki.engr.illinois.edu/display/cs225fa11/Lectures

Suggestions:

1. Make a list of all of our data structures. **Done Courtesy of Michael Miller and Contributors**

2. Make sure you understand the basic function of each (is it a special purpose structure or a dictionary)?
3. Make sure you understand the implementation issues for each (why is it ok to store a heap in an array, but we can't do that for an AVL tree?)
4. Make sure you understand the running times of each of the basic functions (for most, this means the running time of remove, insert, and find)
5. Spend a fair amount of time on graph implementations, making sure you understand the tradeoffs between adjacency lists and arrays, and why each of those is better than just an edge list
6. We did the following graph algorithms: traversals - DFS & BFS, MST - Prim's & Kruskal's. Make sure you understand running times and implementations of each.
7. The few coding questions that will be on the exam will come from MP6 & 7, and also from the lab exercises. make sure you can do all that coding

Notes

Data Structure	Big-O
Unsorted Array	• Insert @ Front: O(1) • Insert @ Arbitrary Spot: O(n) • Insert @ Given Spot: O(n) • Remove @ Front: O(1) • Remove @ Arbitrary Spot: O(n) • Remove @ Given Spot: O(n) • Find: O(n)
Sorted Array	• Insert @ Front: O(1) • Insert @ Arbitrary Spot: O(n) • Insert @ Given Spot: O(n) • Remove @ Front: O(1) • Remove @ Arbitrary Spot: O(n) - can't this be O(log n) using binary search? < no, need to resize • Remove @ Given Spot: O(n) • Find: O(logn)
Singly Linked list	• Insert @ Front: O(1) • Insert @ Arbitrary Spot: O(n) --O(n) w/ find. I went off of the last slide on http://goo.gl/MGIBL for this info. Although I'm struggling to imagine how you could do it without find...--We had to find the errors in that, so the lecture notes are not correct. It was a challenge for the class. So should be O(n) since that was one of the errors. Darn I didn't have that in my notes for some reason, thanks and updated • Insert @ Given Spot: O(1) • Insert @ Back: O(1) -- O(n) if no tail pointer // • Remove @ Front: O(1) • Remove @ Arbitrary Spot: O(n) • Remove @ Given Spot: O(1) w/ hack • Remove @ Back: O(n) -- since you need to fix pointer to tail • Find: O(n)
Doubly Linked List	• Insert @ Front: O(1) • Insert @ Arbitrary Spot: O(n) -- O(n) w/ find. I went off of the last slide on http://goo.gl/MGIBL for this info. It's for SLL's but it should be the same for DLLs too. Although I'm struggling to imagine how you could do it without find... --We had to find the errors in that, so the lecture notes are not correct. It was a challenge for the class. So should be O(n) since that was one of the errors. Darn I didn't have that in my notes for some reason, thanks and updated • Insert @ Given Spot: O(1) • Insert @ Back: O(1) -- O(n) if no tail pointer • Remove @ Front: O(1) • Remove @ Arbitrary Spot: O(n) • Remove @ Given Spot: O(1) • Remove @ Back: O(1) -- O(n) if no tail pointer • Find: O(n)

Stack -
Last In First Out (LIFO), Pez dispenser
	• Array - Preferred because of their speed
	  ○ Push: Amortized O(1) if we double size (and copy) each time we fill the current array.
	  ○ Pop: O(1)
	• List
	  ○ Push: O(1)

	○ Pop: O(1)
Queue - First In First Out (FIFO), Ticket line	• Array - Preferred because of their speed ○ Enqueue: Amortize O(1) if we double size (and copy) each time we fill the current array. ○ Dequeue: O(1) • List ○ Enqueue: O(1) ○ Dequeue: O(1) • Two Stacks ○ Enqueue: O(1) ○ Dequeue: Amortized O(1)
Trees	• Complete: A *tree* in which every *level*, except possibly the deepest, is entirely filled. Complete Binary Tree: A binary tree in which every level (except possibly the last) is completely filled, and ALL the nodes at height h are as far left as possible. n = 2^n (at least). Full Binary Tree: A tree in which every node OTHER THAN THE LEAVES has two children. Perfect Binary Tree: A full binary tree in which all the leaves are at the same level, and in which every parent has two children. n = 2^(h+1) - 1. L = 2^n. Rooted binary tree: A tree with a root node in which every node has at most 2 children. • Binary Tree (In general operations where you might have to end up at a leaf are O(h) for Binary Trees and it's derivatives.) ○ Insert: O(1) ○ Remove: O(n) ○ Find: O(n) ○ Traverse: O(n) • Binary Search Tree //note that these are WORST cases, but on average it should be log(n) for most ○ Insert: O(n) - should be O(log n) for average case. True but these are worst case, which is what we've typically discuss in class, right? Yep ○ Remove: O(n) ○ Find: O(n) ○ Traverse: O(n) • AVL Tree (Balanced BST) ○ Insert: O(log n) ○ Remove: O(log n) ○ Find: O(log n) ○ Traverse: O(n) **Applet:** **http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html**
BTree - block	○ Insert: O(log n) ○ Remove: O(log n) ○ Find: O(log n) ○ Traverse: O(n) ○ B-tree of order m is an m-way tree, which is to say each node can have a maximum of m subtrees/children. ■ m = 2 is NOT technically a binary search tree due to some weird edge case behavior, but Bill Bindi said that this won't get asked because it's too detaily ○ For an internal node, # keys = #children -1 ○ All leaves are on the same level ○ ⌈ m/2 ⌉ - 1 <= # keys per node <= m - 1 ■ all nodes have at most m - 1 keys and at least ⌈ m/2 ⌉ - 1 keys, except the root which may have as few as 1 key ○ Keys in a node are ordered ○ Search:

<table>
<tr>
<td></td>
<td>

O(m) time per node
O($\log_m n$) height implies O(m $\log_m n$) total time but we consider m to be constant so it's O(logn)

Analysis:

Most # Nodes: (m^(h+1) -1)/(m-1)
Most # of Keys: m^(h+1)-1
Least # Nodes: 1+2*[(t^h -1)/(t-1)] where t = ⌈ m/2 ⌉
Least # of Keys: n >= 2*t^h -1
Height: h <= logt (n+1)/2 -> h = O(log n)

We do not need to know insert/delete

Applet: http://people.ksp.sk/~kuko/bak/index.html
</td>
</tr>
<tr>
<td>Hash Table -
For our purposes, if it's under SUHA and alpha (n/N) is constant</td>
<td>

Insert:O(1)
Remove: O(1) --O(n) or O(alpha) under SUHA / alpha is kept under 0.66 so O(1) seems fine. If alpha is kept constant through table resizing then it's constant running time.
Find: O(1)

Hash Tables replace any implementation of a dictionary, I.E. AVL Tree, Array, etc

Why: Unlike AVL, for general hashing the keyspace need not be comparable/defined (<,>)

Assumptions

Hash Fn follows SUHA - Keyspace is evenly distributed - Probability[h(j) == h(k) j != k] = 1/N
Hash Fn is deterministic - Identical inputs always give the same output
Hash Fn runs in constant time
Resizes when load factor alpha (n/N) > ~2/3
Resizes to the next prime > 2 * current size (also prime number) and then rehashed with new hash function

Problems: Anything requiring traversal (e.g. find min/max or find nearest neighbor) takes > O(n)

A Hash Table consists of:

An array

Not an AVL tree because that assumes we can compare the keys using < and > operators.

A Hash Function

A Hash: Function mapping a key to an integer i
A compression: function mapping i into the array cells 0 to N-1. (usually a modulus)

A collision resolution strategy

Separate Chaining - Create separate data structure to hold data (linked list, array, etc)

It's understood to be implemented via linked list though to avoid having to resize arrays and because linked lists give you O(1) insert anyway.
Open Hashing - Data not stored on table I.E pointers to data
O(1) insert
O(n) remove/find without SUHA
O(alpha) remove/find with SUHA

Linear Probing - Insert data at next available spot on table

Closed Hashing - Data must fit in table
H(k,i) = [h(k)+i]%N h(k)=hash fn, i=# probes, N=table size

There can be larger constant steps (such as 2) each probe as well

Problems:

Clustering - Data close together, drastically slows down hashing
Find/Removal may have to look through the entire table

Solutions:

Double Hashing - hash step size when probing I.E. H(k,i) = [h1(k)+i*h2(k)]%N
Keep track of if the cell has ever been occupied
Need to hold load factor (n/N) constant to make find/remove constant (n=# keys, N=table size)

So we need to resize the table when n/N=~.63

</td>
</tr>
</table>

<table>
<tr>
<td></td>
<td>

resize to prime num > current size * 2
re-hash table using original hash function % new_size

Better strategy:

Speed - Linear Probing
Large data sets - Separate Chaining

<table>
<tr><th>Collision Strategy</th><th>Expected number of probes for Find(key) under SUHA.</th></tr>
<tr><td>Linear Probing: Successful</td><td>$\frac{1}{2} * (1 + \frac{1}{1-\alpha})$</td></tr>
<tr><td>Linear Probing: Unsuccessful</td><td>$\frac{1}{2} * (1 + \frac{1}{1-\alpha})^2$</td></tr>
<tr><td>Double Hashing: Successful</td><td>$\frac{-ln(1-\alpha)}{\alpha}$</td></tr>
<tr><td>Double Hashing: Unsuccessful</td><td>$\frac{1}{1-\alpha}$</td></tr>
<tr><td>Separate Chaining: Successful</td><td>$1 + \alpha^2$</td></tr>
<tr><td>Separate Chaining: Unsuccessful</td><td>$1 + \alpha$</td></tr>
</table>
Applet: http://groups.engin.umd.umich.edu/CIS/course.des/cis350/hashing/WEB/HashApplet.htm
</td>
</tr>
<tr>
<td>Dictionary - (Best = via Hash table)</td>
<td>

Insert: O(1)
Remove: O(1)
Find: O(1)

</td>
</tr>
<tr>
<td>Priority Queue - (Best = via heap)</td>
<td>

Insert: O(log n)
Remove: O(log n)

</td>
</tr>
<tr>
<td>Heap - Sorted*ish complete binary tree</td>
<td>

Insert: O(log n)
RemoveMin: O(log n)
HeapifyUp: O(log n)
HeapifyDown: O(log n)

BuildHeap = O(n), use HeapifyDown on root (HeapifyUp does it in O(n lg n)

for (int i = parent(size); i >= 1; i--) heapifyDown(i);

Tree of n nodes: height = log(n)

Implementation with an array, this can be done because a heap is a complete tree:

root @ i = 1
leftChild(i) = 2*i
rightChild(i) = 2*i + 1
parent(i) = floor(i/2)

When the array runs out of room, double the array, and the remaining spaces in the array is equal to the amount of nodes on the last row.

Applet: http://www.cosc.canterbury.ac.nz/mukundan/dsal/MinHeapAppl.html
</td>
</tr>
<tr>
<td>DisjointSets - (Best = via UpTrees)</td>
<td>

w/o path compression and smart union

Find: O(n)

</td>
</tr>
</table>

	○ Union: O(1) ● w/ smart union ○ Find: O(log n) ○ Union: O(1) ● w/ smart union and path compression ○ Find: Basically O(1) ■ For any sequence of m union and find operations, the worst case running time is O(m log*(n)), where n is the number of items. ○ Union: O(1) ● Each node stores parent, find returns OVERALL parent root ● Root stores -(size) or -(height+1) ● Smart Union by size/height to create optimal tree of height <= lg n ● Path Compression - Compress during find to make future finds faster ○ int DS::Find(int i) { if (s[i] < 0) return i; else return **s[i] =** Find(s[i]); } **Applet:** **http://csilm.usu.edu/lms/nav/activity.jsp?sid=__shared&cid=emready@cs2420&lid=28&aid=283765678**
MinimumSpanning Tree - MST	**It's O(mlogn) via Prim's or Kruskall's algo where m represents the # of EDGES and n represents the # of VERTICES.**
Graph - General concepts, runtimes and algorithms below	● n = # nodes (vertices) ● m = # edges ● Connected, Simple(no double edges): ○ n-1 <= m <= n(n-1)/2 = O(n^2) ○ Sum(deg v) = 2m = O(n^2) ● Not connected: ○ ○ m >= 0 ● Implementation: ○ Edge List - Simple, for small graphs, insert = O(1), remove/adjacent/edges = O(m) ○ Adjacency Matrix - n x n matrix with pointers to edges ■ insert vertices/remove vertices/list adjacent edges = O(n) - amortized (from lecture), adjacent = O(1) ○ Adjacency List - in vertex list, each vertex is a linked list of (pointers to?) edges, 2m = O(m) nodes ■ insert vertex = O(1), remove vertex/list adjacent edges = O(deg(v)), adjacent = O(min(deg(v),deg(w))

Graphs

n=vertices, m=edges	Edge List	Adjacency List	Adjacency Matrix
Space	O(n+m)	O(n+m)	O(n^2)
incidentEdges(v)	O(m)	O(deg(v))	O(n)
areAdjacent(v,w)	O(m)	O(min[deg(v), deg(w)])	O(1)
insertVertex(v)	O(1)	O(1)	O(n^2); O(n) - amortized
insertEdge	O(1)	O(1)	O(1)
removeVertex(v)	O(m)	O(deg(v))	O(n^2); O(n) - amortized
removeEdge	O(1)	O(1)	O(1)

Algorithms	
Singly linked list	● insert/remove O(1) ● remove abitrary O(n) due to find()
Array	● insert front O(1) ● insert at given O(1) ● insert/remove given/arbitrary O(n) due to shift
Tree	● Traversals ○ pre-order - act before processing children ○ in-order - act between processing children ○ post-order - act after processing children ● Rotations ○ Left ○ Right ○ LeftRight ○ RightLeft ● B-Tree Search O(mlogmN) / m is treated as constant so O(log N) is also fine
Hash Table	● Collision handling ○ Separate Chaining ○ Linear Probing ● SUHA (Simple Uniform Hashing Assumption) - The assumption states that a hypothetical hashing function will evenly distribute items into the slots of a hash table. Moreover, each item to be hashed has an equal probability of being placed into a slot, regardless of the other elements already placed. ● Hash Function - Ideally Bipartite Function ● load factor *alpha* = n/N, where N is the size of the hash table, and n is the number of current keys in the table. i.e., the average number of elements in a chain. ● *alpha* = n/N should be no more than ~2/3, this is when the algorithms will start to slow down.
Heap	● HeapifyUp - O(log n) - does h constant time swaps from a leaf to the root, h = log n ● HeapifyDown - O(log n) ● HeapSort ● buildHeap - nlogn
DisjointSets	● Find - O(log n) in practice w/ path compression/smart union, O(1) ideal, O(n) no optimization ● Union - O(1) ● Compress Paths - done within find

<table>
<tr><td></td><td>

- Union by size - smart union

</td></tr>
<tr><td>Graphs</td><td>

- InsertVertex/Edge
- RemoveVertex/Edge
- IncidentEdges
- areAdjacent
- origin
- destination
- Kruskal - mark non-cycle creating minimum weighted edges first
- Prim - use partition with PQ
- DFS/BFS - O(m+n)
- Dijkstra
- Single Source Shortest Path - SSSP
- Implementations with adjacency list vs. adjacency matrix : see table above

<table>
<tr><td>Kruskal (MST)</td><td>Create a disjoint set for the vertices with all vertices initialized to -1
Create a set S containing all the edges in the graph (use a priority queue)
Create a blank list to hold your final list of edges
While S is nonempty
remove an edge with minimum weight from S
If that edge connects two different trees, union the vertices
and add the edge to your edge list
otherwise discard that edge
Your edge list and the list of all vertices is your final graph
http://www.youtube.com/watch?v=OWfeZ9uDhdw</td></tr>
<tr><td>Prim (MST)</td><td>Create a tree containing a single vertex chosen arbitrarily from the graph = Vnew
Create a set containing all the edges in the graph
Loop for all vertices
Choose an edge {u, v} with minimal weight such that u is in Vnew
and v is not or vice-versa (if there are multiple edges with the same weight, any of them may be picked)
Add v to Vnew and add {u, v} to Enew
Output: Vnew and Enew describe a minimal spanning tree
http://www.youtube.com/watch?v=BtGuZ-rrUeY</td></tr>
<tr><td>Shortest Path (Dijkstra)</td><td>http://www.youtube.com/watch?v=8Ls1RqHCOPw

This algorithm operates in a very similar manner to Prim’s algorithm. However, when in the “updating”stage of Prim’s algorithm the parent (and distance of the node) are compared with the total distance taken to get to that node. It looks something like this:

for all adjacent vertices W of V:
if(cost(v,w) < d[w]){ // this is prim’s conditional. Dijkstra’s is cost(v,w) + d[v] < d[w]
d[w] = cost(v,w) + d[v];
p[w] = v;
}

This algorithm requires all edge weights positive</td></tr>
</table>

</td></tr>
</table>

BFS Traversal
- Queue

```
Algorithm BFS(G)
        Input: graph G
        Output: labeling of the edges of G as discovery edges and back edges

// initialize the graph
For all u in G.vertices()
        setLabel(u, UNEXPLORED)
For all e in G.edges()
        setLabel(e, UNEXPLORED)
// find and traverse all components in the graph
‘For all v in G.vertices()
        if getLabel(v) = UNEXPLORED
                BFS(G, v)

Algorithm BFS(G, v)
        Input: graph G and starting vertex v
        Output: labelling of the edges of G in the connected component of v as
        discovery edges and cross edges

queue q

setLabel(v, VISITED)
q.enqueue(v)
While (!q.empty())
        q.dequeue()
        For all w in G.adjacentVertices(v)
                if getLabel(w) = UNEXPLORED
                        setLabel( (v, w), DISCOVERY )
                        setLabel(w, VISITED)
                        q.enqueue(w)
                else if getLabel( (v, w) ) = UNEXPLORED
                        setLabel( (v, w), CROSS )
```

DFS Traversal
- Stack
- Recursion

```
Algorithm DFS(G)
        Input: graph G
        Output: labeling of the edges of G as discovery edges and back edges

// initialize the graph
For all u in G.vertices()
        setLabel(u, UNEXPLORED)
For all e in G.edges()
        setLabel(e, UNEXPLORED)
// find and traverse all components in the graph
For all v in G.vertices()
        if getLabel(v) = UNEXPLORED
                DFS(G, v)


Algorithm DFS(G, v)
        Input: graph G and starting vertex v
        Output: labelling of the edges of G in the connected component of v as
        discovery edges and back edges

setLabel(v, VISITED)
For all w in G.adjacentVertices(v)
        if getLabel(w) = UNEXPLORED
                setLabel( (v, w), DISCOVERY )
                setLabel(w, VISITED)
                DFS(G, w)
        else if getLabel( (v, w) ) = UNEXPLORED
                setLabel(e, BACK)
```

Fall 2009 Exam Solutions

---

01. [Choices! Choices! - 20 points] Please add explanations of why it is that one & not other letters if you know.

MC1. C ← Why not A? I agree with A. For list, Just check the vertex you are checking. If it has even 1 link, it is connected. But it says if the graph contains A node that has no edges, not a specific node. So then we have to traverse the nodes right? Yes, cause that would be a different question than the one I read. :)
I chose C because, I thought in worst case, we need to check every vertex to show that there is a vertex without any edge.

MC2.-A <- Why not C? I guess A too, but not positive.I looked at the tree example from class of remove child for the tree. She mentions if she didn't do *& it would make a copy of the pointer.lecture 10/22 It is A, reuse of a question from old midterms with rubric posted.

MC3. C Wouldn't this be A because we have a tail pointer? To push, we just update the tail node pointer and the tail pointer. To pop, we just update the tail pointer to ...oh... we need to search through the list to get the thing before the tail. C it is!

MC4. D iii is wrong<--WHY! an order 2 B tree can only have 1 node so why isnt it a binary tree. <-evert BST is not a B-Tree because it doesn't need to have all leaves at the same level... BSTs do not need to be height balanced<--touche … but B-trees do...which is why every BST is NOT a B-Tree -- I think i and ii are right because i references the total seek time, while ii references the seek time for one level, iii is wrong. :) ----- guys I believe the wrong choice is ii because the number of disk seeks per level of the tree is constant

--The answer is still C despite what the person below mentions. The two questions are different. The one on exam 2009 mentions the bound time of O(log_m(n)) while the midterm mentions m*log_m(n)). The latter has the incorrect value for the height IMO
The answer to MC4 is B, also a reuse of midterm 2 question, rubric somewhere online google the question. link : https://wiki.engr.illinois.edu/download/attachments/81231990/mt2_rubric.pdf?version=1&modificationDate=1307125916000

The question from midterm 2 and this one slightly differs. For example, midterm 2's ii) mentions the number of key comparisons while this one says the number of disk seeks. The number of disk seeks depend on height of the tree. Also in i) it says that the height is O(mlogm n) but, in final practice it is O(logm n).

MC5. C No path compression so can't be log*n. Set Union needs to find root's of two elements O(log n)
setunion can be O(1), if you assume you use find before you setunion (meaning the roots are already found). But generally setunion would call find too, so I would agree with you. -- In class she mentions O(1) for setunion and O(logn) for find for this case.

MC6. B (5n+2) - (n-1) = 4n+3
MC7. A
MC8. C **EXPLAIN**!!?!?? yea<-- in the notes it says O(m + n + mlogn) <-that is for unsorted array --says sorted array in notes -- I would say D -- I believe you would need minimally n^2, since in order to build the array, note: add all the edges, you need to traverse the adjacency matrix. Afterwards its mlogn for the actual algorithm. Making it D, is this incorrect?
MC9. B- both DFS and BFS are O(m+n) //how? <-I think it is because you iterate over all the vertices (n) and edges (m) giving O(m + n)
MC10.D, Why is it D? Since this is basically same as choice B. I thought the answer was E. I believe it is D due to path compression <--can anyone explain further? Start with 7, this points to 6, then 6 points to 5. By path compression 7 will now point to 5. Next step is 5 points to 4, by path compression 6 and 7 both point to 4... etc. So in the end D would have to look like B to be a valid sequence of union and find operations using path compression. You could argue that starting with making 1 point to 0, then 2 point to 1, then 3 point to 2 would be valid under path compression, but this is not how union works as the union will always union the representative members of the set, so again it would end up looking like B.

IT is D because it says path compression is employed.

02. [A Little C++ - 10 points]

a. delete[] orkut; <- why do you use []? its an array of int's / it is a dynamic array, so you need [] / Fairly certain it's a pointer to a non-dynamic array, right? / int* orkut = "new" int[4] which means it is heap-allocated every "new" has to end with every "delete" or the memory will leak I think that is what Chase said during his discussion section / Yeah, but does it have to be delete[] / Yes because what initialization code line is doing is that it creates 4 blocks (each 4 bytes I guess) of spaces on the heap not on the stack frame. So then when is just the keyword delete needed?/ delete[] just deletes spaces for the dynamic array. If the array is dynamic array of pointers like int** arr = new int*[10], then you need to delete each arr[index] using delete command then finally delete [] arr. For

example, for bebo, first bebo[1] is deleted using delete since bebo[1] is pointing at a heap memory (new int). Then bebo is deleted which is storage of 3 blocks of integer pointers.

```
for (int i = 0; i < bebo.size(); i++)
{
        delete bebo[i];
// this is not needed since only bebo[1] has allocated a dynamic memory for int so “delete bebo[1]” seems fine<-this
}
delete[] bebo;
```

So the correct solution is:
delete[] orkut;
delete bebo[1];

//Does hi5 not need to be deleted and set to NULL in this problem? (hi5 is from the 2010 exam)

Why is this here?
Summer 11 MT 1 Rubric
If you look at the past exam solutions, the code does not look like this....
Summer 11 MT 1 has a dynamically allocated array of BMP pointers. BMP ** frames; Not a vector of dynamic arrays of BMPS..

b. vector < BMP * > album; <- not sure on this ← looks fine... dynamically allocated?? wheres the new We were only asked to declare the variable.

c. when the object goes out of scope & when it is explicitly called by deleting a ptr

When dynamically created object is deleted (reword of the last one)

Isn’t this just one instance where the destructor is invoked?

03. [1D-Search - 20 points]

anyone care to explain what is going on in this problem?

a.besides the three function, do we need to write other stuff in the public part? -No

So would A just look like this...

```
class OneDSearch
{
        public:
        OneDSearch();
        void insert(const double x);
        double query(const double y) const;
```

If you have const inside, you dont need it outside? -- Const on the inside means the parameter won’t change, const on the outside means that the function won’t modify class member variables. You can have a function that does one and not the other

//Is it needed to have const for the double parameters? The problem gave:
• OneDSearch(): creates an empty 1-dimensional search structure.
• insert(double x): inserts the real number x into the structure.
• query(double y): returns the real number in the data structure that is closest to y. If
the data structure is empty, return a large number constant called BIG.

no cosnt double.

}
???
Since we are implementing AVL tree which uses dynamically allocated memory like TreeNodes (I cannot think of AVL tree implementation with arrays, it can be possible but... I don’t know), **I think we need Big three(copy constructor, destructor, and assignment operator) <-yes or no? ---------No, the default ones should work fine.**

b. AVL Tree Why is AVL tree best? //this is how we did MP6...we need a BST because we need a good search function. A self

balancing binary search tree is better than simply a BST. But my MP6 was much more complicated than what's given below. Was your nearestneighborsearch not like this? similar, but done both for the left and right subtrees, also I did the temp<best check more often. And where does this return BIG if the structure is empty? there happy yes! ;-)y? and shouldn't the piece below be the helper function? The spec says query only takes a double as a parameter. In query you'd create temp and set currBest to something? And query should return the closest element, where is the distance determined?

c.

```
double OneDSearch::query(TreeNode * temp, double x, TreeNode * currBest) <<currBest needs to be by reference
{
	if (temp==NULL) return BIG;
	if (temp!=NULL) //base case
	{
		if (temp->data == x) return x; //found value
		if (abs(temp->data - x) < abs(currBest->data - x) //the current node is closer than the previous best, update currBest
		{
			currBest = temp;
		}
		if (temp->data < x) return query(temp->left, x, currBest); //search left subtree
		else return query(temp->right, x, currBest); //search right subtree
	}
	return currBest->data;
}
```

d. O(n)

04. [Hashing - 15 points]

a. GBDACEF

works by entering B to 1, then insert D (can't go in 1 so goes in 2) then insert F (can't go to 2 so goes to 3), then insert A (can't go to 3 so goes to 4) then insert C (can't go to 4 so goes to 5) then insert E (can't go to 5 so goes to 6) then finally insert G, can't go to 5, goes to 6 can't go to 6, gos to 0.

b. assumption 1: evenly distribute items into slots of the hash table (SUHA)
assumption 2: deterministic meaning the output is determined by the input
assumption 3: hash function itself must be O(1)
assumption 1(resize): keeps load factor suitably low (~0.6)
assumption 2(resize): resizble x) size doubles to next prime

```
{
	return query(root, x, root);
}
double OneDSearch::query(es by doubling size of array and up to the next prime number and
		rehashing to the new array// actually: moves hash to new array in chunks with every access/store
```

05. [Skip Lists - 20 points] **Did we go over these in class? //pretty sure we didn't**

http://en.wikipedia.org/wiki/Skip_list

06. [Binary Heaps - 20 points]

a. C, H, J, not A.read the block of text here.. //can someone explain why? <- Last spot you insert is where J is. By heapifyup and inserting C and H in the current spot will give you the same minHeap as the picture shown ←- A shouldn't be included because if A was the last one to be added, before it was added, B would be at the top leaving B's right side to not create a complete tree // I also thought A should be included but after I read your argument, I agree that A should not be included., GOOD CATCH!!!!!<-not true A is included

--Why would B be at the top? Whichever is at the top is A's maxPriorityChild, which is not necessarily B. The point is that you insert to the position that J currently occupies, and heapifyUp. The tree's shape and structure is still the same, but the values at the nodes are different. It would still be balanced, and the last node HAS to be inserted as one of H's children, so the right subtree isn't involved. Couldn't C have been at the top instead of B if A was the last one to be included? //How else would you add a value larger than all the rest last?, No I'm saying that A should be included because before adding A, C could have been at the top and that would cause no errors in the tree, whereas if B was at the top there would be errors. If C was at the top then the path to the right subtree wouldn't be

increasing.Why wouldn't it, the heap property would be restored if C was on top before A was added. C > B.so the right subtree isn't a heap anymore C doesn't have to be bigger than B, C would be on top if it was less than B, its a min-heap. Exactly.. I think we're just considering the "alphabet" so C comes AFTER B.. meaning its greater. So C can't be on the top of B. If A wasn't there, B must be the root.Does it say anywhere in the question that we can assume C>B? you've got a point..but I guess it's expectedI think since the question just asks "Which of the keys COULD have been inserted last" A is a valid answer since we can't assume if C> or < B. <it's implied A<B<C I know what you're trying to say, but I would still stick with what's been said. Just because that seems more "natural". Ok, I guess ill make a post on piazza and thatll help. Update this when you find out. sure

i.

```
void Heap<kType>::changeKey(int index, const kType & newKey) {
    kType temp = theHeap[index]; //keep current data safe  O(1) //no reason to do this? need key later
    theHeap[index] = newKey;//update/change key at index O(1)//whether you use heapifyUp or Down depends on whether the newKey > or < the curKey

    if(temp > newKey)
        heapifyUp(index);//O(logn)
    else if( temp < newKey)
        heapifyDown(index);//O(logn)
```

//"Function changeKey(int index, const kType & newKey) changes the key in position index to value newKey and restores the heap property" yea, i think we need to remove the old value and change it to the new value I don't think we can just do heapifyDown(index), imagine if the new key was the smallest value in the heap, then calling heapify down on it would never bring it to the top of the heap. We must instead do buildHeap which is O(n)buildheap is not provided, so would we have to write it? << all you do is check if old value is > or < than newKey and call HeapUp or HeapDown.that would work.

ii. O(logn) <-this depends on your implementation for i. what did you do to get O(logn) by the way?
I think its 0(n), using heapify down... you never go through the whole tree. heapifyup and down run in logn time

iii. ~~public~~ private member function -> private? because the index is passed in and the client won't know which index its at? I also agree this should be private, because the client has no idea about what the index is since the vector will be a private variable. Client won't even know if the implementation of this heap is an array. **This should be private!!!??? <- Agreed**

07. [Priority Queues - 20 points]

a. it prints out the k largest values in the array a ←shouldn't this be GREATER OR EQUAL??? Doesn't it print the K largest numbers. Consider if K was 1 then the top loop runs, and each time removing the minimum when the pq is of size larger than 1, thus the bottom loop will only print 1 value. So while the numbers printed are greater than k, it is just the k largest values. ← I agree, the top loop removes the smallest size - k values and the bottom loop prints out the rest, which is the k largest values

b. O(n log n)

c. prints k numbers that are the smallest Why would this be a different answer than A, implementaiton of a PQ should just change the run time, not the output ← it's also telling you to remove the line if (pq.size() > k) pq.removeMin(); Ohhh thanks, got it

d. O(nk) ← why is this O(nk) there is no loop that runs n times within which k work is done...<--removemin is a O(n) function that is run k times <<according to above tables removeMin is O(logn)

08. [Landmark Paths - 15 points]

a. BFS

b. 1 time from X <--shouldn't this be once to X, then once to W?. Implement the algorithm by starting at X, then marking the distance to V and W, add those together. This way you only run it once, not twice. But you are searching for two distances, isn't that two uses of BFS? or are you keeping track of dist(x,v) and dist(x,w) as you go?

c. X

d. Run BFS at X. Connect path between X to V and path between X to W

e. O(n) Isn't it O(n+m)? since worst case for BFS? yes. O(n+m) n=dist(x,v) m=dist(x,w) ??

09. [Algorithms - 20 points]

a. Prim's Algorithm, which returns minimum spanning tree, so path with lowest weight.

b. m times, this will only execute once for each edge. since you check if the other vertex is labelled, this m times

c.

i. Since graph is not dense for large n, adj. list gives most efficient running time.

ii. Heap ← line 18 clearly shows that they are using an unsorted array... // How? Can you clarify why? d[w] is array for keeping distance of each vertex from starting point. It has no relation with PQ implementation.

iii. O(n log n + m log n) which is O(n log n) since m is smaller than 10n.

Spring 2010 Exam Solutions

---

01. [Choices! Choices! - 20 points] Please add explanations of why it is that one & not other letters if you know.

MC1. A

MC2. C (there is a tail pointer so pushing is O(1), but for pops you must iterate through the linked list to get a pointer to the 2nd to last element.

MC3. E // I chose E can anyone explain which of three statements are wrong? <--I agree with answer as E. every order-2 b-tree is a BST, but not every BST is an order-2 B-tree. in a B-tree, all leaves are on same level, but in a BST they do not need to be. So it should be D?

MC4. C

MC5. D

MC6. B O(klogN) performance w/ BST and removeMin, O(k) for preorder b/c we don't need to care about order! Won't preorder be O(n) not O(k) because we have to process the whole tree/heap for a traversal?, No because we can only process stuff less than x.No, we modify it so that we don't visit children that are greater than the algorithm's input.

MC7. D Not A because...if they are all roots path compression and smart union does nothing :-) it's d because it's just a big chain and the optimizations SHOULD flatten that out

Should be E^ The pattern union(0,1), union(1,2), union(2,3), union(3,4), union(4,5), union(5,6), union(6,7) produces this structure. you can have any number of find statements in between as long as they are either find(0), find(1), are not a number that has been unioned with the main set yet (i.e. union(0,1), find(5), union(1,2)).

MC8. C // Isn't it B? 5n + 2 - (n + 1)<-where does that come from = 4n + 1 this one is different from FA09 one, I think B too // A minimum connected graph has n +1 edges. So there must be only n+1 discovery edges. Edges which exceed that amount should be all back edges. A minimum connected graph has ***n - 1*** edges, not n+1 edges. The correct answer is C.

MC9. A (Back is for DFS and Cross is for BFS)

MC10.C-- It's using an adjacency matrix graph here. Wouldn't that change the running time to be O(n^2 + mlogn)? Wikipedia says its C, you're doing O(m) things and apparently those take O(log(n)) time, not sure why.

MC11.B

MC12. C Why not E? // The second statement is incorrect because x and y can have the same distance from s the phrase "at least" makes this statement false //But doesn't "at least one less than" allow for the situation where they are the same?

02. [A Little C++ - 10 points]

see 2009 exam solutions for this question

03. [1D-Search - 20 points]

see 2009 exam solutions for this question

04. [Hashing - 15 points]

a. NOPBRIG Isn't it GOPBRIN also since it would go to N fist put it in box 6 then G in box 0? I also got NOPBRIG.

Put P in 2, R in 4, O in 1, B in 6 // this is where I'm not sure if I'm moving over enough spaces

Put I in 3 //because 2,4,6,1 are taken

Put N in 0, G in 5 //because 0,1,2,3,4 are taken

Result is NOPIRGB , am I correct?? No... Put N in 0, G in 5 //because 0,1,2,3,4 are taken

Lets just say we do all the non multiples first. R into 4, O into 1. Then you have 5 multiples left.

We are doing linear probing so okay now we say put P into 2. Then B is also 2 so it has to go into next box up. So it goes into 3. Then I goes into 5 since its the next empty box. Then N and G is left. N is box 0, and then G is box 6 since its last box left.

This order is not final though because it doesnt say the letters are called in order but if they are that would be it.

b. Hashing above in 09 exam

05. [Binary Heaps - 15 points]

see 2009 exam solutions for this question

06. [B-Trees - 10 points]

a. (m^(h+1) - 1) / (m-1) can anyone confirm a and b //I agree with your answers**. why isnt this just m^(h+1) - 1?**

b. m^(h+1) - 1 Wikipedia says m^h -1... Different definition of h, whether or not you count the root the root should only add 1 though... so via your logic it would become m^h to the height...

c. number of disk seeks in a search for a particular key in a B-tree dependent on height h where h=log_m n

(isn't c supposed to be log(m^(h+1) - 1 ?) // Nope m^(h+1) - 1 is total number of keys. You definitely don't traverse every single key to find a particular key

07. [Miscellaneous Data Structures - 15 points]

a.

i. hash table, binary search tree, AVL tree, linked lists, arrays, B-Tree.....good luck ..and get

ii. queue, stack, disjoint sets, priority queue, heap, kD-tree, graph.....d

b.

i. Graph (Why graph? I thought it was Disjoint Sets? (I'm with disjoint sets as well)) (It's graph because pals of pals are also your pals. you can't connect children of a root in one dset to another one in a different dset. in graphs you can) --DS anyone in one set becomes friends with everyone in the other set. ←- also in graphs, running time to determine if some other friend is is connected to A. // I also chose DS Pals are sets, the second sentence describes union So... Both... Right? It says data structure(s).

ii. If it is Disjoint Sets then it is O(logbasem(n)) (If its a graph, then O(m+n)) -- for DS I would say mlog*n<- i agree

08. [Graphs - 15 points]

a. No, because you can loop and make length smaller so it can go on forever

b. How do we do this one? We need to find the smallest weight for the unmarked edges. The easiest to start with is the upper right edge. the other edges are 1 and 2, so the upper right edge needs to be at least 4(3? see below). upper left must be 4 so that the path through the center is less than going around the outer edges. left from center must be 7. down right from center must be 8. bottom must be 12. Thoughts?

I put top right 3, top left 4, middle 5, bottom right connection 6, bottom 7 and it worked for MST Kruskals algorithm on an applet. www.cse.iitk.ac.in/users/dsrkg/cs210/applets/minSpTree/MST.html

no other spanning tree has that weight, so upper right must be 4.

c. If it doesn't have a lot of edges Can somone Explain this. It is explained in the second exam that is given to us with solutions.

09. [Algorithms - 20 points]

a. **depth first search and it basically does a traversal or search on a tree/graph** (is it copying g as well? i just thought that because of the add edge and add vertex function calls, but they might be for something else.) I dont think its copying a graph it takes in an input of a graph. If you look on the DFS lecture slide it says it takes in a graph(gotcha, thanks!) Also if you look on the code when it has parameters (graph & G, vertex s) it means that its taking in a graph labeled g and starts vertex v

b. "n" times? ← yikes I said m :(←-hmm i thought m +n ... i put m <--- does it not occur for every single vertex and then for every single edge at that vertex? it doesn't happen if the vertex or edge has already been labeled. So I have numOfEdges which is m in this case.

c. If dense use adj. matrix, if sparse use adj. list? I think this is correct after reading the adjacency list entry on wikipedia

d. O(n+m)

e. O(n+m) also? not sure <--- i actually thought the same but then why would it be same? // Since m is no larger than 10n, it is also OK to assume that it is O(n). --Right, because sum[deg(v)]=10n, so m=5n which is O(n). So the run-time is n+5n, which is O(n).
