Introduction

The following highlights are provided to give you an indication of the topics that you should be knowledgeable about for the final. This sheet is not a substitute for the homework and the lectures, and it is possible that there will be some aspects of the questions that are not specifically covered on these pages. However, the test questions are based on the topics identified here.

The exam will be given in MGH 241 on Wednesday, June 12, 2:30-4:20 PM. It covers material from the entire course, with more emphasis on the material starting with lecture 11, Binary Heaps. This includes chapters 6 through 9.

The exam is closed book, no notes, and no calculators.

The questions will be short answer, similar to the problems you have done on the homework.

Fundamentals

Refer to the midterm review sheet and the appropriate lecture notes for more discussion of these fundamental topics.

Pointers are a tool in C for pointing to a block of memory. You should be able to read simple code that uses pointers and structs and understand and explain what it is doing.

It is important that you understand how the number system works using positional notation, exponents and logarithms.

Refer to the charts titled Binary, Hex, and Decimal that I gave in the lectures on April 8, Fundamentals, and April 22, Hashing Appendix. You should be able to translate numbers between 0 and 15 among decimal, binary, and hex values without a calculator, and you should be able to translate any number of hex digits to binary or vice versa without a calculator.

Asymptotic Algorithm Analysis

Asymptotic analysis is a way to characterize the growth of functions with respect to the size of the independent variable (usually the size of the input data set for an algorithm).

f(x) is big-O g(x), ie, O(g(x)), if Cg(x) is an upper bound for f(x), x>k.
f(x) is omega g(x), ie, Ω(g(x)), if Cg(x) is a lower bound for f(x), x>k.
f(x) is theta g(x), ie, Θ(g(x)), if C_1(g(x)) is an upper bound for f(x), x>k_1, and C_2g(x) is a lower bound for f(x), x>k_2.

Abstract Data Types (ADT)

The Binary Heap ADT provides a minimal set of operations for managing a data set in which the important attribute is that a particular element key is the minimum (or maximum) key in the heap. A min-order binary heap provides only FindMin, DeleteMin, and Insert functions as part of the ADT specification. A max-order binary heap provides FindMax, DeleteMax, and Insert, and works exactly the same way with just a reversed comparison. A binary heap is structured as a binary tree and each path from a leaf to the root is sorted and that is all that is required in order to maintain the heap order property. As a result, the root is always the smallest (or largest) node in the heap. The implementation of a binary heap is simplified because the tree can always be maintained as a complete tree (all nodes are in use except for possibly some at the end of the bottom row). Given a drawing of a min-order binary heap, you should be able to redraw the tree after a DeleteMin or an Insert operation as in lecture 11 of April 26. Binary Heap uses a simple implementation of an insertion sort to accomplish the percolate down and percolate up functions.

The Binomial Queue ADT provides the same kind of priority-oriented access to data elements as the Binary Heap, but it also provides fast merging of two prioritized sets of elements. This merging speed is possible because the Binomial Queue is maintained as a forest of heap ordered trees. Each of the trees has a very specific structure according to its position in the forest, and each tree is maintained with the heap order property defined above for Binary Heaps. In the lecture on Binomial Queues on April 29, I showed several examples of merging two queues by "adding" together the trees in the forest to form new trees and a new forest. You should be able to do this too, with the end result being a properly structured Binomial Queue with the elements in the correct places for maintaining the specified heap order property. Note that the trees in a Binomial Queue are not binary trees, because they have more than two children at many of the internal nodes. Consequently the implementation of Binomial Queues uses pointers in a First-Child / Next-Sibling representation, rather than the simple array implementation used for Binary Heaps. FindMin requires scanning the root nodes of the trees in the forest, and consequently is O(logN).

Given a comparison operator that imposes a consistent ordering on the elements of a list **A**, then sorting that list will reorganize the elements of the list such that for any i and j, if $i<j$, then $\mathbf{A}[i] \leq \mathbf{A}[j]$. The key measures of a sorting algorithm are its use of space and time. If the algorithm needs a fixed amount of space to keep track of where it is while operating, then it uses O(1) space and is said to be an in-place algorithm. If, on the other hand, it needs to copy a set (or subset) of the elements being sorted, then it is not an in-place algorithm, and the amount of space required varies with the number of elements to be sorted. The worst case time for sorting algorithms is generally $O(N^2)$, since in that time you can compare all N elements against all the other (N-1) elements. The best case time is shown to be O(NlogN) for any algorithm that performs comparisons between two elements at a time.

Inversions are a useful concept for describing the sorting process. An inversion is a pair of elements that are out of order according to the comparison function being used. In other words, if you have $i<j$ but $\mathbf{A}[i] > \mathbf{A}[j]$, then elements i and j are inverted. Note that the definition of an inversion does not have anything to do with the algorithm you are going to use to sort the list, it is strictly a property of the list (given a particular comparison function). Different comparison functions may give different numbers of inversions for the same list.

The Insertion Sort algorithm is a simple sort that compares one element at a time with the elements in an already sorted list. It operates by comparing the element to be inserted with the last element in the list. If the insert element is smaller, the last element is copied down and the process is repeated with the next element up. Copying continues until the proper spot is found, and the new element is inserted. You should be able to show the comparisons and copies that are needed to do a simple insertion sort on a list of elements. The worst case run time for insertion sort is $O(N^2)$ when the data is in reverse order. The run time is O(N) when the data is already sorted. You should be able to describe why the algorithm runs faster on sorted data.

A sort algorithm that swaps only adjacent elements can only resolve one inversion at a time. Since the maximum number of inversions (reverse order data) is (N-1)N/2, an adjacent element swap sort algorithm is $\Omega(N^2)$. The key to faster sorting is to resolve several inversions at once by moving the elements more than a single position with each swap. This means that elements in widely separated positions must be compared and swapped, rather than only adjacent positions.

The Shell Sort is an early example of a sort algorithm that compares and moves elements distances greater than 1. Using a set of diminishing increments, the Shell sort makes several passes over a list, doing insertion sorts on the subarrays defined by the increments. You should be able to define the subarrays that are defined by a particular increment value. Remember that there are numerous ways to define the set of increments, and some are better than others are. Shell's increments are a sequence in which each increment is half of the one before it (eg, 16, 8, 4, 2, 1). This is not a good choice, since the same subarrays get sorted over and over, and too much work is left for the last pass when the increment equals 1. A better choice is Hibbard's increments which are 2^k-1 (eg, 15, 7, 3, 1). With Hibbard's increments the Shell sort worst case run time can be shown to be $\Theta(N^{1.5})$. (I will not ask you to show this on the exam!)

The Heap Sort algorithm is based on the observation that doing DeleteMax repeatedly on a max-order heap will extract values from the heap in reverse sorted order. Through careful use of the array that stores the heap, we can use a binary heap to do an in-place sort. Referring to the implementation of a binary heap, we remember that it is represented as a complete tree. Consequently, removing one element from the heap with a DeleteMax operation means that the array is one element shorter. The max element that was removed can be held in temporary storage, while the element that is currently last in the tree is removed and reinserted in the binary heap. The max element is placed in the newly vacated array position and the process is repeated.

Merge Sort is a divide and conquer algorithm that divides the array to be sorted in half, then recursively sorts the two halves of the array. Combining the resulting sorted sublists into a single sorted list is fast because it is easy to combine two sorted lists by copying the smaller of the two initial sublist elements to the new combined list. Mergesort is implemented recursively and runs in O(NlogN) time. However, MergeSort requires O(N) extra space for storing the temporary arrays.

Quick Sort uses the same divide and conquer approach, however it does not require extra space because it maintains its sublists in the original array. Thus it is an in-place sort algorithm. The pseudocode for Quicksort is very simple.

```
QuickSort(S)
   if S.length is 0 or 1, return (this array is sorted)
   pick an element v in S (this is the pivot)
   partition the remaining elements in S-{v} into two disjoint sets with
      S1 containing all values x ≤ v and S2 containing all values x ≥ v
   return QuickSort(S1),v,QuickSort(S2)
```

The actual implementation of QuickSort generally uses an Insertion Sort when the number of elements is below some low cutoff value like 20. The low overhead of Insertion sorting makes this more efficient.

Key aspects of the QuickSort algorithm include picking the pivot and implementing the actual partitioning. Picking the pivot is best done by implementing Median3, which picks the middle value of the first, last, and center values and uses that as the pivot. A good pivot value will split S into two approximately equal sized subsets and results in O(NlogN) performance. A very bad pivot will cause one of the two subsets to be empty every time and results in $O(N^2)$ performance. Clearly, choosing a good pivot is critical. Partitioning is implemented as swaps between two elements that are in the wrong subset. Since these are swaps, they are in-place. Since they are elements at a distance, they potentially resolve multiple inversions. Dealing with cases where the element equals the pivot by stopping and swapping is important to maintaining equal sized subsets, which is critically important to obtaining maximum performance for QuickSort.

You should be able to pick the next pivot element out of an array, and show which values will be put in which subset as the result of a given pivot.

The Disjoint Set ADT maintains information about equivalence classes on a set. Equivalence classes are defined by an equivalence relation on a set. An equivalence relation obeys three properties: reflexive, symmetric, transitive. You should be able to state whether or not a defined relation is an equivalence relation. The relation “is part of the same tree” is an equivalence relation that is at the heart of Kruskal’s algorithm for finding minimum spanning trees. The equivalence classes defined by an equivalence relation are disjoint subsets of the original set, and they form a partition of the set. The Disjoint Set ADT is interesting in part because it can be implemented rather simply using an array to implement up-trees. You should be able to read and write drawings of up-trees similar to those in the lecture of May 20, and also be able to conduct Find and Union operations using a diagram of the (virtual) up-tree or the actual array implementation.

You should be able to correctly conduct Union-by-size and path compression during Find and show the effect on an up-tree.

Several of the preceding data structures have nodes linked to each other in a variety of ways. Some of the links are explicit using pointers, others are virtual and implemented using careful calculation of array subscripts. We can generalize this with the study of graph algorithms.

Graph terminology. Node, vertex, edge, directed, undirected, directed graph (digraph), adjacent, degree, in-degree, out-degree, cycle, circuit, directed acyclic graph. Each edge contributes +1 to the degree of each of the two vertices it is incident with. The number of edges is exactly half the sum of the degrees. The sum of the degrees is even.

An undirected graph is connected if there is a path between every pair of distinct vertices in the graph. A graph which is not connected is the union of two or more connected subgraphs, which are called the connected components of the graph.

Graphs are represented with adjacency lists and adjacency matrices. You should be able to create either one from a drawing of a graph, and draw a graph based on either one. Adjacency matrices are very useful mathematically for calculating properties of a graph, but adjacency lists are often used for storing graph information since they are more efficient for sparse matrices, which are very common in practical applications.

A graph is bipartite if its nodes can be partitioned into two disjoint non-empty sets such that every edge in the graph connects a vertex in one set to a vertex in the other set. This also means that no edge connects a vertex in one set to another vertex in the same set. You should be able to state whether or not a graph is bipartite.

A topological sort of a directed graph G=(V,E) is a linear arrangement of the nodes in the graph such that for any edge (v_1,v_2) in the graph, v_1 precedes v_2 in the ordering. A topological sort is not possible if the graph has a cycle.

A simple Breadth First Search (BFS) can be used to find the shortest path length from an initial node to every other node in the graph. You should be able to apply BFS to an unweighted graph and find path lengths and the resulting spanning tree.

Depth First Search (DFS) is another simple algorithm for exploring all the nodes of a graph. It can be easily implemented recursively, thereby (implicitly) using a stack to perform the backtracking that is necessary when a dead end is reached. You should be able to apply DFS to a graph and show the resulting spanning tree.

Dijkstra's algorithm is similar to a breadth first search, except that it is used on weighted graphs to find weighted shortest paths from an initial node to every other node in the graph. You should be able to apply Dijkstra's algorithm to a graph and find the minimum cost paths from an initial node to the other nodes. See the lecture "Graph Paths" of May 24 for the format of the table.

A spanning tree is a tree that touches all the vertices in the graph (spans the graph) and forms a tree (connected and contains no cycles). A minimum spanning tree (MST) is a spanning tree with the least total edge cost.

Prim's algorithm starts with a random node, and then proceeds to build the MST by adding the lowest cost edge in the graph that is incident with one of the nodes in the tree already selected and doesn't cause a cycle. This process is repeated until all vertices have been added. Prim's algorithm is a good example of a use for binary heaps (priority queues). A binary heap is used to maintain prioritized access to the "lowest cost edge in the graph that is incident with one of the nodes in the tree".

Kruskal's algorithm uses the disjoint set ADT to manage the growth of a forest of small trees into a single minimum spanning tree. Initially the algorithm starts with a forest consisting entirely of single node trees. There are as many single node trees as there are nodes in the graph. It then proceeds to combine trees by adding edges between them selected from the graph. The lowest cost edge is always selected unless it would cause a cycle. Cycles are detected by doing an equivalence class Find operation on the two nodes that the edge connects. If the class is the same, then the nodes are part of the same tree and so adding the edge would cause a cycle, and so it is rejected.

You should be able to apply both Prim and Kruskal to simple graphs and derive minimum spanning trees.

Euler paths are paths in a graph that traverse every edge in the graph once and only once. An Euler circuit is an Euler path that starts and ends at the same node. An undirected graph has an Euler path if the degree of all the nodes is even, or if the degree of two of the nodes is odd and for all the other nodes is even. An undirected graph has an Euler circuit if the degree of all the nodes is even. There is a simple algorithm based on Depth First Search for finding Euler paths. Repeated DFS applications give you one or more circuits which can be spliced together to form an Euler path. You should be able to apply these rules, and draw Euler paths and circuits if they exist on a given graph.

A Hamiltonian circuit is one that goes through every node once and only once. There is no known polynomial time algorithm for finding such a circuit.

A problem that can be solved in polynomial time is a member of complexity class P (Polynomial). A problem for which a proposed solution can checked in polynomial time is a member of complexity class NP (Non-deterministic Polynomial). If a problem is in NP, but is not in P, then it is a candidate for the application of advanced algorithms based on heuristics or approximation, rather than a direct solution. If every problem statement S in NP can be converted to a particular problem statement T in NP using a polynomial time reduction, then T is NP-complete. There is a large set of NP-complete problems, and so far as is known, there is no polynomial time algorithm for solving any of them. If any one of them can be solved in polynomial time, they can all be solved in polynomial time. If any one of them can be shown to absolutely not be solvable in polynomial time, then none of them can be solved in polynomial time. This is one of the biggest open questions in computer science today.