Introduction

The following highlights are provided to give you an indication of the topics that you should be knowledgeable about for the midterm. 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 class on Friday, May 3. It covers material through the lecture Wednesday, April 24. This includes chapters 1 through 5.

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

Pointers are a tool in C for pointing to a block of memory. The memory is generally allocated in blocks using malloc or related library functions, and then pointers are used to link the blocks together and access the contents of the blocks (usually referenced as structs). You should be able to read simple code that uses pointers and structs and understand and explain what it is doing. I will not ask you to write code in the exam.

For many reasons it is important that you understand how the number system works using positional notation, exponents and logarithms. In any number base, the position of a digit within a number is determined by the power of the base that it represents. The right-most column in any number represents the units value ($base^0$). The next column to the left represents multiples of the base ($base^1$). The next column represents multiples of the base squared ($base^2$). And so on.

For any given number of digits d, the maximum number that can be represented is $base^d-1$. For example, the maximum base 10 value that can be represented in 3 digits is $10^3-1=999$, and the maximum base 2 value that can be represented in 3 bits is $2^3-1 = 111_2= 7_{10}$.

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. Every group of 4 binary bits maps directly to one of the 16 single-digit hex values.

$Log_2x = y$ means $x=2^y$, or "the log of x, base 2 is the value y that gives $x=2^y$". $log(ab) = log(a)+log(b)$, $log(a/b) = log(a)-log(b)$, and $log(a^b) = b \cdot log(a)$. Unless otherwise

specified, in this class, all logarithms are base 2. You should be able to draw graphs of 2^x and $\log_2 x$ that are correct as to general shape and intersections with the axes.

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). The key idea in asymptotic analysis is to find functions g(x) that can be expressed simply and for which $C_1g(x)$ and $C_2g(x)$ can be shown to be upper and lower bounds of the function of interest f(x), when $x>k_1$ and $x>k_2$, respectively. This means that we can state with certainty how rapidly f(x) grows in proportion to the growth of x.

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), $x>k_2$.

An algorithm's performance can be characterized by counting up the number of operations required to perform the algorithm for a data set, finding a formula to express that number in terms of N, the input data set size, and then finding a way to characterize the formula using O(g(x)), Ω(g(x)), and Θ(g(x)) notation.

Common algorithm performance comparison functions (ie, candidates for g(x)), are g(x)=1 (constant), $g(x)=\log_2 x$ (log), g(x)=x (linear), $g(x)=x\cdot\log_2 x$ (log linear), $g(x)=x^2$ (quadratic), $g(x)=x^3$ (cubic), $g(x)=2^x$ (exponential). You should be able to correctly place these functions in increasing order, as they are shown here, and use them to describe other functions using O(g(x)), Ω(g(x)), and Θ(g(x)) notation.

You should be able to read simple descriptions of algorithms, and provide a simple analysis of the number of operations needed to accomplish the algorithm.

Abstract Data Types

The List Abstract Data Type (ADT) provides Insert, Find, Delete, and Advance operations in the basic interface. These operations are sufficient to allow access to and processing of arbitrary elements in the list, without regard to position in the list.

An array-based implementation of a general purpose list suffers from significant run time issues related to Insert and Delete, because all the elements in the array must be copied in order to Insert or Delete at an arbitrary location. The run time in this case is O(n), where n is the number of elements in the list.

A pointer-based implementation of lists is more flexible, because an arbitrary node in the list can be inserted or deleted with a simple change of pointers. Changing the pointers is a constant time operation O(1). Depending on how the list is organized (single or doubly linked) finding the node in which the pointers are changed may be O(n) or O(1).

You should be able to look at a drawing of a list and show how the pointers would need to be modified in order to add or delete a node.

The Stack ADT is a specialized usage of a list in which elements can only be added at one end of the list (the *top*). This simplifies the design of the implementation, and also removes the high-cost Insert and Delete operations that weighed against an array implementation of lists. The equivalent Push and Pop operations affect only one end of the stack, and consequently an array implementation is appropriate and extremely common. Push and Pop are constant time operations O(1) and you should be able to draw a picture of an array based implementation of a stack and show how you would change values in order to do a push or a pop. Similarly, you should be able to draw a pointer-based implementation of a stack and explain how it is similar and how it is different from the array-based implementation. A stack is Last-In, First-Out, or LIFO.

The Queue ADT is another restricted list. Elements are added at one end of the queue and removed from the other end. This ensures that all elements are processed in the order received. (It doesn't ensure that each element is important to us, but we can't have everything … (joke)) Queues are implemented with pointers or arrays. If pointer-based, then pointers to the head and the tail of the queue are kept and provide fast access. A pointer implementation allows for essentially unlimited queue size, at the cost of the overhead of pointer space and pointer operations. An array implementation uses a current index for the head and tail of the queue and a large array for storing elements. An array implementation requires a fixed maximum queue size, but is very fast and space efficient in operation. In both implementations, add and remove (enqueue and dequeue) are constant time operations because they happen at the ends of the list, and the ends are easily accessed.

The Tree ADT takes many forms. A key structural issue is that all trees have a hierarchical structure and this structure can be used to encode information when the tree is built. Consequently, some operations are much faster than they can be in a linear structure like a List. Tree terms are defined in the lecture of April 15, Trees Intro. Root, node, edge, leaf, parent, child, sibling, ancestor, descendant, subtree, path, pathlength, height, depth.

Trees are often implemented with pointers and linked nodes. When each node can have an arbitrary number of children, then one common link structure is to store two links in each node. The first link points to the first child of the node, the other link points to the next sibling of the node. This is the 1^{st} child / next sibling structure, and it can handle an arbitrary number of children.

A very common tree structure is the binary tree. In this case, each node can have 0, 1 or 2 children. Since the maximum number of children is known, two links are stored in each node and each link points directly to a child node (if it exists). The minimum depth of a binary tree is $O(\log_2 n)$. The maximum depth of binary tree if there are no balance constraints is n, the number of nodes. In this case, the tree is an expensive linked list.

One common element ordering scheme applied to trees is that of the Binary Search Tree. In a Binary Search Tree, the left child is smaller than the parent node, and the right child is larger than the parent node. This is true for all nodes in the tree and so there is an order maintained throughout the tree.

The structure and ordering properties of a binary search tree mean that the minimum element is at the end of the left path and the maximum element is at the end of the right path. In a simple binary tree, insertion is easy; delete is a little harder because nodes need to be rearranged. Binary search trees with no balance constraints can become unbalanced very easily. You should be able to describe balanced and unbalanced trees and provide big-O descriptions of the run-time for various operations on a tree in a particular condition. You should be able to describe the operation of Find, FindMin, FindMax, Insert, on a pointer-based implementation of a binary search tree. You should be able to read simple procedures and recognize and describe the function being performed.

Balanced trees are implemented in a variety of way to eliminate the possibility of the completely degenerate tree with its O(n) access time. An AVL tree maintains balance by enforcing a balance condition after every insert and delete. The balance factor (|height(left subtree) – height(right subtree)|) can be no more than 1. The rotation operation is used to balance a tree after a change that has affected the balance factors. You should be able to take a drawing of a tree such as in the lecture of April 17, slide 13, and recognize whether or not the tree is out of balance. If it is, you should be able to do the appropriate single rotation and draw the new configuration of the tree. I will not ask you to do a double rotation. Find is always O(logn) in an AVL tree because it is always balanced.

Splay trees are not perfectly balanced all the time, but they are adjusted as operations are performed with them. Over a sequence of operations, the restructuring of the tree causes the amortized operation cost to be logn, and possibly better depending on the access patterns. The splay operation is similar to the rotations in AVL trees. You should know that splay trees can be very unbalanced, but that the overall run time is still O(logn).

The Hash table ADT provides constant time Find and Insert. The capability to impose a general order on the elements has been sacrificed, but finding a particular element by key is very fast. The hash table ADT uses a hash function to convert the key to an index, and it uses a collision resolution strategy to resolve the case when two key values hash to the same index location. There are numerous hash functions, and the proper choice is dependent on the speed requirements and the characteristics of the keys. An even distribution of hash index values is usually a key characteristic of a good hash function. Patterns in the keys should not be reflected in patterns in the distribution of hash indexes. A perfect hash is one that provides a collision free hashing of a particular set of keys. You should be able to comment on a particular hash function and a particular set of keys and describe whether or not the function would be a good choice for the set of keys.

Collision resolution is often handled by separate chaining. This means that a list is maintained of all the elements that hash to the same index, and then on Find the list is scanned after hashing to the list head. This adds a small time penalty, but each list should only be N/(table size) if there is an even distribution being provided by the hash function, and so the list search time is not a large burden. The load factor of the hash table is N/(table size). Separate chaining can support load factors greater than one because the lists just get longer.

Another collision resolution strategy is open addressing. In this case, the hash table is the only place to store data elements, and so a probing strategy is required to move from a table entry to the next until an empty entry is found. Linear probing and double hashing are two techniques for finding the next location to store an element in. Open addressing cannot support high load factors because the search for the next open location rapidly approaches a linear search when the table is near full. Open addressing cannot support load factors greater than 1 at all because the table is full and there is no place to put the next entry.

Rehashing is one escape when an open addressing table gets full, because you increase the size of the table and add all the entries in the old table to the new, larger table and start over again.

You should be able to look at drawings of open addressing searches for an open location and comment on the probing technique, the general load factor, and visible clustering.

Binary heaps and binomial queues will not be on this exam (they will be on the final, however).