Sorts

Insert: Treats the left most element of the array as a sorted list with one element. Goes through the rest of the array moving elements in the right array to a sorted position in the left array. | worst case n(n-1)/2/O(n^2)| O(n) for a sorted list.

Selection: Swaps the highest value in the unsorted list with the value at the end of the list. |n^2 bad sort

Bubble: looks at the next cell and compares values. If n+1>n it swaps into the correct order|n^2|worst of the sorts

Merge: A divide and conquer sort. It splits the list at the midpoint and sorts both halves. When recombining it compares the elements in each split of the array to decide the order. Needs a second array. n-1 comparisons O(nlog(n)) for best case, average, and worst. Splitting the list is simple, recombining is harder.

Quick: Pick a value from the list to be the pivot, optimally pick three values and choose the middle of the three. Split the list into two parts, elements lower than the pivot to the left of it and higher to the right. Once partitioned it sorts the two halves | |O(nlog(n)) for average and best case. Has O(n^2) with a sorted list.

Searches

Sequential: Steps through the list. O(n)

Sentinel: Removes the comparison for sequential. Able to add/remove from the list (this must be faster than O(n)) adds the key to the end of the list. O(n)

Probability: put more likely targets up front. 2 Types: Static- arrange then sort Dynamic-start with sequential or sentinel then if the key is found move it up the list. O(n)

Binary: Has to be ordered. Split the list in half and check if your key is higher or lower repeat till key is found or no more data. Forgetful: doesn't stop, is faster because it does one less comparison, and regular binary is only faster when it does half the searches, the chance of that happening is less than 3/sqrt(n). O(log(n))

Hashing: Table must not exceed x% full/must distribute keys evenly/can have O(1)/constants should be prime/Hash table needs to be fixed length/Simple functions: Mod division: key = c%x where x is prime/Pseudorandom: p(key*a+c)%c(a and c should be large and prime)/Less simple folding: break about key into chunks then add the chunks/Rotation: Split the key and move some parts/Collision resolution: Rehash or extend the table/clustering: keys keep getting put in the same spot(bad hash) use a second hash function

Stacks: Hold one data type/ only open on one end(push)/remove from the top(pop)/look at element(peek)/used in backtracking/

Queues: Hold one type of data of any type/Enqueue (adds to one end)/Dequeue (remove from other end)/Queues can wrap on themselves/need to keep track of the head, tale, and size. Can be implemented with a circular array or with a linked list that has a pointer pointing to the end of the list so enqueue/dequeue still have O(n).

Linear/Linked Lists: Holds searchable data types/Pos are numbered/search[]/traverse: looks at all values in the list/insert: adds something to the list(any spot)/remove: removes item from list(any spot)/How to interst:make sure pos is valid>Create new node>Find node that comes before>new node points to where old node points to>Old node points to new node>update count/How to remove: make sure pos is valid>find the node before the one you want removed>remember the ntbd>give the prev node the ntbd pointer>delete node>update count

Trees: It is a collection of nodes or a collection of edges/there is only one path between two nodes/all nodes need to be connected/doesn't matter how many nodes are connected to a single node/there are n-1 edges where n is the # of nodes/there are different ways to judge the "size" of a tree. height: # of edges size: # of nodes airty: max# of children for a node (binary have no more than 2)/There are 6 ways to processes a tree but normally L before R. To add an item to a binary tree you essentially search through the tree for what you want to add, when you find where it should be you add it to that position. Removing is more complicated. When removing a node with only a right sub tree, just make the right child the new root node, the best way to do this would be to choose the right most child as the new root and restructure the tree. Works if there is no child as well as it sets it to null. If there is a left and right child, the right most child to the left of the root is chosen as it is the highest value on the left.
Implementing: A tree can be implemented with an array, where position zero is the root and the following positions are the following levels. If the root is A and it's left child is C and right child D then it would go A,C,D in that pattern. Nodes: uses pointers to point from the root to its children.
Graphs: a collection of nodes or edges (edges often carry info)/no one path rule/ all trees are graphs/There are two ways to store: adjacency matrix or list matrix/List are more involved but it is easier to add/remove data has a O(V+E)/Two types of searches Depth and Breadth
Implementation: Adjacency matrix. It uses a two dimensional array to show what is connected to what. Essentially if two positions are adjacent to each other, they are marked as true, if not false.
Depth: it has a starting vertice and travels from along an edge to an unmarked neighbor. If a dead end is met it goes back to the last vertex, if a vertex has no other neighbors it goes back to the last vertex, it can only traverse vertex's that have a path from the start position to it.
Breadth: Uses a queue. Adds the start vertex to the queue. Marks the position and adds any connected vertices to the queue. Repeats until no unmarked vertex remain.
Header: include files, constants/defineds,data struct classes, global vars, and function prototypes
Big-O: O(1)> O(logn)>O(n)>O(nlogn) or O(lg(n!))>O(n^x)> O(2^n)>O(x^n)>O(n!) O(n^n)
Pointers: Is the memory address of a variable. Tells you where something is instead of what it is. The * to declare a pointer the & to tell it to point to something, example int *pnt, I; pnt=&i; this has pnt point to i. To expand on this. Int i=42, *p1, *p2; p1=&I; | p2=p1; | then cout<<*p1<<" "<<*p2<<endl; will print out 42 twice.
Heaps: A heap is always a complete binary tree. A heap is priority based. The value in a node is always larger than the value in its children. A complete binary tree has no nodes with a single right child, there is either two children or only a left child. Adding: When adding to a heap, the value is placed as the left child of a node in a way that keeps the tree complete. It is then compared to its parent, and if larger swaps places. This is done until it is in the proper position. Removing: you remove the value with the highest priority. You then choose the last entry in the last level as the new root, you then swap it with its child if one is larger than it. This is repeated until it has no children larger than it.