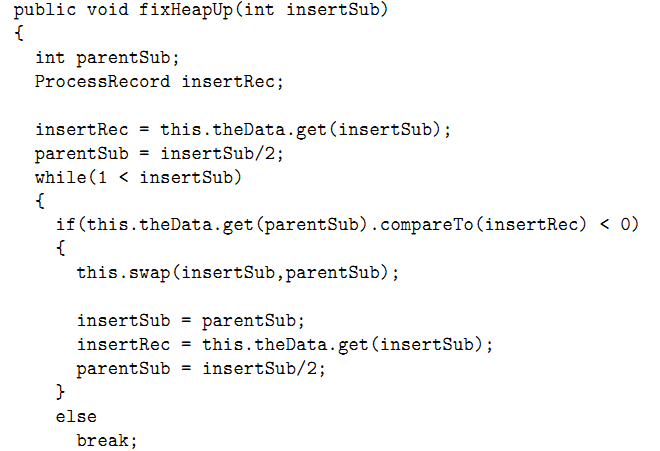
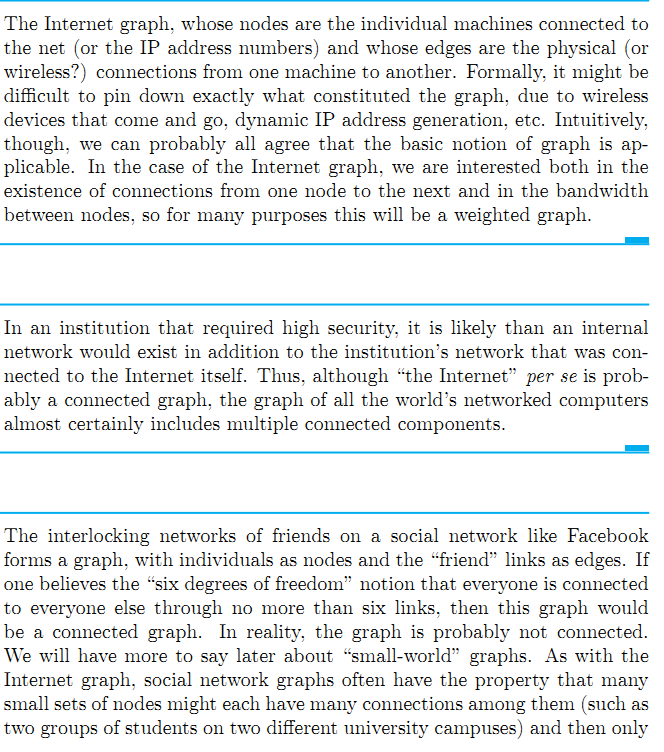
1. Chapter 7 –stacks
   1. Stacks
      1. Two data structures built off a linked list – stack and queue
      2. Can use and arraylist or an array
      3. A stack aka a push down stack –is last in first out LIFO data structure
         1. For example a napkin dispenser
      4. Stack requires two methods – push and pop
         1. Also needs an instance variable of top
         2. Push adds another item before the location pointed by top then adjusts he value of top to the new object
         3. Pop does the reverse in that it removes the top item and adjust the top variable to the object after the removed one
      5. Addathead of a dll is same as push method
      6. Should write an isempty method to see if stack is empty before poping
      7. Peek method - return top item but not actually delete it
   2. Exception handling
      1. Peek and push can cause problems
      2. Peek on an empty list – return null value and require user check for that
      3. Pop – changes structure and returns data
         1. Always check isEmpty before issuing a pop
         2. Or can return a null value and check for null = valid pop is issued after check goes through – should use isempty method
   3. Stacks using nodes
      1. Linked list only requires the new push and pop methods.
         1. Push is already made by addathead method
         2. Pop method requires remove method of dll but requires no search as the node will be always be the first one
         3. Also need a isempty and peek for completeness
      2. Dll is more general as additions/deletions can be made anywhere but the stack object can only add delete from head
   4. Stacks using arrays
      1. You want the top variable to point to the last array subscript
      2. Can increment top when adding a node and when removing don’t need to shift any data down the array
      3. Can easily check for size of array to see if empty before popping
   5. Hp calc and reverse polish arithmetic
      1. Reverse polish notation
         1. 
         2. Add these to a sack in order of operations
            1. 4 2 3 x + 5 7 + 8 \* -
            2. Push the 4, push the 2, push the 3, reach a sign and take out the last two and use that sign on them so then taking out 2 and 3 the next push is 6….
         3. The ability of this is to write and read expressions from left to right without parenthesis and not have to search for closing parenthesis when doing the regular way
   6. Html/xml and stacks
      1. A more standard way of using stacks is xml parsing
      2. The tags need to have open and close for each data.
      3. Push the open tag, read the data and then should reach a close tag and that should match the popped value. Can keep adding open tags but the close tag should be the last open tag pushed onto the stack
      4. A math expression doesnt need a stack, only a counter to make sure the number of open brackets are the same as the number of close brackets
      5. Processing of xml
         1. Begin with empty stack – load an initial null pair
         2. Read the first line and push on stack the tag and its contents.
         3. Read the next line which is another open tag and push that tag with its data
         4. The next line is a close tag and the stack will be popped and the popped tag and the found close and should be the same
         5. If not then the xml data is not correct
   7. Queues
      1. The simplest is data added to the tail and removed from the head
      2. Concept of a queue
         1. Data items are added only at the tail
         2. Data items removed only at the head
         3. User is entitled to peek at data item before removing it
         4. Function exists to tell user if there is nothing to queue
      3. Addbefore method needed and dequeue can be same as pop
   8. Queues using arrays
      1. Use circular arrays for queues
         1. They add to tail and take away from head and continuously do this
         2. Arrays can only be of fixed length and not continuously added to
      2. Top = (top +1) % array.length;
      3. Need to test the size of the array as well
   9. Jfc classes - stacks
      1. Nearly all data structures are in the jfc including stack and queues
      2. The difference in the jfc stack and the one explained is it returns a node and the node must be used to get the data out of instead of just having the data when popping
   10. Jfc queues
       1. Queue is a java interface not like stack or linked list
       2. Requires the methods add, element, offer, peek, poll, and remove
       3. Instead of queue it has add and offer returning Boolean
       4. Instead of dequeue it has poll and remove – both remove elements and poll returns null if queue is empty – remove throws exception
       5. Instead of peek – element and peek methods – element throws the exception if empty
   11. Stacks, queues and deques
       1. The jfc stack and the queue interface – stack is old – jfc is mor general
          1. Queue is a collection designed for holding elements prior to processing
          2. This is done in a fifo fashion
       2. A deque is a double ended queue – instertions and deletions can be made at either end but only at the ends
       3. Jfc arraydeque implements both queue interface and deque subinterface
          1. Has an addfirst, addlast, and well as a removefirst, removelast methods
          2. Arraydeque is made for linked lists
   12. Priority queues
       1. Major feature of linked lists, stacks and queues is dynamic data – structures grow and shrink as data is added/removed
       2. Can add items dynamically and keep them ordered according to priority.
       3. Pop will then always remove the highest priority item from the list
       4. Similar to operating systems with tasks
       5. Entries always added at tail and always removed at head.
       6. The top data item would be the highest priority and that’s the only data that would be in order and matter
       7. Can be sorted with insertionsort – but not all items need to sorted
       8. Need a mechanism that will:
          1. Efficiently rebuild structure after head is removed so that next item is highest priority.
          2. Efficiently inset new itme into structure but it structure still has property of head being highest priority.
   13. The heap
       1. The structure commonly used for this purpose is the heap
       2. A total order is produced when an item is added to the list
       3. The heap represented as a tree is easier to understand
       4. All trees are formed from nodes(circles) and arcs (lines)
          1. A tree with a node with no more than 2 children is called a binary tree
          2. A complete binary tree would have all nodes with 2 children
       5. Max heap – largest value is located at subscript one and the value of any given node is higher than its child nodes
       6. Two features:
          1. Priority queue of N items can be created from scratch in NlogN
          2. Inserting one more item into priority queue can be done in logN time
   14. Creating a heap
       1. An array of N values can be converted into a heap in NlogN times – theorem
       2. 
   15. Maintaining heap
       1. Fix heap up/down both only require log time work
2. Chapter 8
   1. Introduction
      1. Recursive – one that calls, invokes, itself
         1. Examples are Fibonacci sequence and the factorial function
            1. Fib – 1, 1, 2, 3, 5, 8, 13, 21…
            2. N-th fib number Fn is defined by Fn = F(n-1) + F(n +2) subject to base conditions F1 = 1 and F0 = 1
            3. Factorial N! = N\*(N-1)\*…\*(2)\*1 or N!=N\*(N-1)! Subject to base condition 0! = 1
      2. 2 part to the definition of a recursive method
         1. First – steady state function such as N!=N\*(N-1)! –we define function for parameter N in terms of the function when given smaller parameters (N-1 in the case of factorial and both N-1 and N-2 in case of fib)
         2. Second – we define base condition that allows us to know when to stop the recursion. For factorial base condition is 0!=1. For fib base case is both 0 and 1.
         3. Can apply different base cases that create different sequences ie fib sequence with F(0)=2 and F(1)=1 creating the ‘lucas sequence’
            1. Such sequences are ‘linear second order recurrences; linear being F(N) terms appear and second order is needing two consecutive values to define sequence uniquely
   2. The collatz problem
      1. To compute N! requires exactly N function calls
      2. Computing F(N) is a little more complicated and is:
      3. ------------------------------------
      4. Defined as C(n) (= 3n +1 if n is odd) (= n/2 if n is even)
   3. The Ackermann function
      1. Does have a history in computer science
      2. Defined as:
         1. A(m,n) = n + 1 if m=0
         2. =A(m-1,1) if m>0 and n =0
         3. A(m-1,A(m,n-1)) if m>0 and n > 0
      3. Only small numbers used because it will run out of memory being recursively called
   4. Binary divide and conquer algorithms
      1. Can be converted to a recursive version
      2. Set upper and lower limits of array bounds and recall binary search with those limits
   5. Permutations and combinations
      1. Naïve search of every combination of a puzzle will equal N! computations
   6. Gaming and searching applications
      1. Gaming such as chess try to find the best move and this is used recursively
   7. Implementation issues
      1. Heuristic – a notion ought to be true or seems like a reasonable way to trim away unlikely candidates
      2. Ex – throwing away words that obviously aren’t English when search for a word in a jumble of letters
      3. Each recursion is given space allocated for variables, compiler, code, etc
      4. May need so many recursive calls – allocated space – that available space is ran out
   8. Summary
      1. Recursion introduction
         1. Factorial function, Fibonacci sequence, and Ackermann function, and 3n+1 function
      2. Recursion is used extensively in game programs and in search trees, binary divide and conquer althgorithms as well.
      3. Common use of recursion in is in computing/enumerating permutations and combinations.
3. Chapter 9 – graphs
   1. Intro to graphs
      1. A tree is a special kind of graph
      2. A graph is a pir G = (V,E) compromising a set V=(Vi) of nodes (or vertices/vertex) and a set E=(Ek=<Vi,Vj>) of edges(or arcs) with each edge being a pair of nodes.
      3. If an edge exists, we say the node Vi is connected to nodeVj and the edge Ek is incident to node Vi to a node Vj
      4. If there is an edge Ek from a node Vi to Vj then we say the two nodes are adjacent
         1. A standard layout of streets is an example
         2. Nodes being intersects and streets being edges
      5. Graphs can be either directed or undirected
         1. If some streets are one way then its directed graph
         2. If all streets are two way then its indirected
         3. Ability to transverse from node Vi to Vj and vise verse
         4. This requires the edges to be an ordered pair
            1. E<ji> would point to E<ij> and vise verse
      6. Subgraph – is another graph G’=(V’,E’) for which V’ U V and E’ U E and the edges in E’ only connect to nodes in V’.
         1. A subset of nodes, together with some subset of edges that join in the subset.
      7. Path – sequence of edges that connect tail to head
         1. E0=Vi0,Vi1
         2. E1=Vi1,Vi2
         3. E2=Vi2,Vi3
         4. Note: we count the number of n edges not the number of vertices connected by edges
      8. Cycle - path for which the beginning and the ending node are the same
      9. In undirected path, number of edges incident to a particular node is the degree
      10. In directed path need to consider both the indegree and outdgree – direction different
      11. Connect – if there is a path between any pair of nodes in the graph
      12. Connected component – is a subgraph of the nodes that is a connect graph and that is maximal with respect to being connected subgraph
      13. Weighted and unweighted graphs
          1. Unweighted- know only that an edge exists that connects two nodes
          2. Weighted – permitted to assign a weight to each edge –capacity/distance function ex bandwidth between interent nodes or distance in miles
   2. Examples of graphs
      1. 
      2. A tree is a graph that is connected but has no cycles – ex evolutionary tree
      3. Forest of trees – set of disconnected trees
   3. Transitive closure
      1. Following the links of your links and then the links of the links of your links and they all link up in the end until no more new links are there
   4. Theory
      1. Not efficient to keep all data of every single link – only the neighboring links that will most likely get you closer to your destination
      2. 3 examples of what is needed
         1. Underlying real world structure can be represented as a graph
         2. Can store in any given node only local info about nearest neighbors
         3. Need to know global properties about the graph like its connected components
   5. Local-global
      1. Info is what is produced when data is organized
      2. Primary key in which data was organized with – ex the name or number
      3. Problem is what is a start node if all nodes are interconnected
      4. Search pages – ability to compare relevance of given pages against other pages and produce global ranking system on the dynamic system of web pages
   6. Greedy algorithm
      1. The idea that at any stage when a choice is to be made, one takes the greedy, or hopeful whatever looks as the best route at that moment.
      2. Game playing can’t use this
      3. The shortest route at that time may not be the shortest route over all
4. Chapter 9 – trees
   1. trees
      1. Tree- a connected graph with n nodes and n-1 edges and connected with no cycles
         1. Most obvious example is family tree
         2. Viewed in one direction usually from top parent to bottom children
      2. Can have a root – top that all other link to eventually or have no obvious root tree (unrooted trees).
      3. Ancestor node – parent node – child node
      4. A node with only edges on them are called leaf nodes – no children –
         1. Once you get to a leaf there is nowhere to go but back up tree
         2. A node that is not a leaf is called a interior node
      5. Many program make use of the root tree concept
      6. Height and Depth of a node – number of edges between that node and the root of the tree
         1. Ie the number of ancestors between the node and the root
      7. Depth – measurement of its distance from the root
      8. The height of a node is the distance in the opposite direction – the distance to the furthest leaf node
      9. The height of a leaf node is 0 and the height of a non-leaf node is 1 plus the maximum of heights of all the children of that node
      10. Not all trees are balanced – if no leaf node is much farther apart than others
          1. Balanced trees are more efficient
   2. Spanning trees
      1. A tree that includes all the nodes in the graph and a subset of the edges that connects the nodes into that tree
         1. Phone tree that allows one to call other that will in turn call others and therefore reach everyone
      2. Important in wireless devices – they change position relative to fixed towers constantly and you still need to establish efficient networks of the phones
   3. Data structures for trees
      1. Differences in linked list is in the node class the connections from one node to the next that implement the data structure one level higher up
      2. Instead of next and previous – there will be an up –parent- and a down – children- pointers.
         1. The number of children could be variable – need a variable length structure like an arraylist to keep track of the children node pointers
      3. Treenode in java – have a stub, parent node, zero or more children nodes(stored in an array).
   4. Binary tree
      1. A general tree is usually needed because one cannot control the number of children for any given node
      2. A binary tree is a tree in which each node has at most 2 children
         1. Even if missing nodes will have the empty child nodes present
      3. Have appeared in heap structure for a priority queue and the binary decision structure of binary search
      4. Binary tree complete if every interior node has exactly 2 child nodes
         1. Compete tree contains 2^n-1 nodes and 2^(n-1) leaf nodes with n levels
         2. Except possibly for missing nodes on the right hand edge of the lowest level
   5. Implementing binary trees with nodes
      1. Don’t need an array list to keep track of variable amount of child nodes
      2. Can have ‘siblings’ pointers for children to point at each other
      3. If using array, assign subscripts to a constant such as LEFT\_CHILD = 0 and RIGHT\_CHILD = 1
   6. Implement binary trees with arrays
      1. We cannot look locally at a given node in a tree and determine a proper sequence number for that node without looking globally at the rest of tree
      2. Binary trees can establish a fixed subscripting mechanism using an array/list
      3. Start at subscript one as to keep an order
         1. This way the left child is subscript (2\*i) and the right child is (subscript 2\*i+1)
         2. Level k will have nodes subscripted from [2^k] to [(2^k)-1]
   7. The heap
      1. A binary tree whose nodes carry data payloads that have comparable numerical values is said to satisfy the heap property if for all nodes other than the root we have that thisNode.getValue() <= thisNode.getParent().getValue()
         1. If the data payload stored at any given node is less than or equal to the payload value stored at the parent node
      2. Max heap – max values on top , min heap – min values on top
   8. Traversing a tree
      1. Decision tree – of possible moves such as Sudoku
         1. Given decisions we make, options open after a given decision are changed leading to a set of options for the second decision
      2. Transverse a tree – to visit all the nodes of the tree in some predefined order
      3. There are three types:
         1. Preorder = Visit the root-recursive visit the left-recursive visit the right
         2. Postorder = Recursive visit the left-recursive visit the right-visit the root
         3. Inorder = Recursive visit the left-visit the root-recursive visit the right
      4. Example
         1. Preorder – 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
         2. Postorder – 8,9,4,10,11,5,2,12,13,6,14,15,7,3,1
         3. Inorder – 8,4,9,2,10,5,11,1,12,6,13,3,14,7,15
      5. Many standards for laying out web pages or xml documents as tree such as DOM (Document Object Modeling)
      6. Postorder – one version of the use of a stack for processing algebraic expressions or nested markups in markup language like html or xml
      7. Preorder can be found in greatest common divisor or gcd of two positive integers a and b defined as
         1. int gcd(int a, int b){ if(a==b)return a; else if(a<b)return gcd(a, b-a); else if (a>b)return gcd(a-b,b); }
      8. LISP- now in artificial intelligence and compiler writing – is intended to be parsed in preorder fashion
   9. Breadth-first and depth-first traversal
      1. Depth first – the same as an inorder traversal – the order in which the nodes are visited are the same – but is used in many instances as part of a depth-first search
         1. Variations of depth-first search
            1. Game strategy – Sudoku – too many illegal options that don’t need explored
            2. Game such as chess – too many possible moves to explore each move – quantify the goodness of any given move and not go after the ones that are bad
      2. Breadth-first traversal (level traversal) – the nodes of the tree are visited in order – all the nodes of level one followed by all the nodes of level two, etc…
         1. Easy for binary trees but not for general trees in that you need to know the location of the next node in the same level and therefore need global information
         2. Also would have to move the levels around and the node locations if deleting a node
   10. Traversal on parallel computers
       1. Embarrassingly parallel – if problem is absolutely trivial to sketch out an approach for doing independent parts for the computation in parallel with each other
       2. In doing a search on a tree through parallel processors is to perform a hybrid breadth-first and depth-first search.
          1. Locally on each processor, a depth-first exhaustion of the tree is performed
          2. Globally, if all tasks took the same amount of time, then the task to a black of processors would look like a breadth-first search
       3. The art is in load balancing so that individual tasks are in an efficient computation and the cost of new processes and task management is small compared to computing depth first chunks of the tree
   11. Summary
       1. Most general tree structures would be implemented using an arraylist to store pointers to the child nodes for any given node
       2. Binary tree have 2 child nodes for each interior node
          1. Often correspond to binary divide and conquer
          2. Another use of binary tree is a heap
5. Chapter 10 – sorting
   1. Intro
      1. 1/3 of all cpu cycles are spent sorting data
      2. Need a key to use to organize data by that key such as last name
   2. Worst case, best case, average case
      1. Best to discover algorithms that are best even at their worst time
      2. Algorithm chosen for data that is semi sorted will probably be different from the algorithm chosen to sort random data and there need to know best-average-worst case senarios
      3. Bubblesort
         1. This is an O(n^2) computation
         2. The outer loop iterates N-1 times and the inner loop N-2 times, the next iteration of the outer loop the inner loop iterates N-3 times…etc
      4. Insertsort
         1. Similar to bubblesort in that we are fixing only one entry into position with each iteration of the outer loop.
         2. The worst case is O(N^2) – records presented in exact reverse order
         3. Best case – can run in linear time – checks each data and doesn’t sort
         4. Average – we have to progress half way down list and only cut off a factor of ½ and therefore is still O(N^2) comparisons
   3. Improving bubblesort
      1. Keep track of if any swap in a given outer loop – if no swaps, no data out of place, then that part is sorted
   4. Heapsort
      1. Can run in NlogN comparisons for the worst case and the average case
      2. Build a max heap
         1. For(i=N; I > 0; i--)exchange root data item with item at location I – run fix-heap-down on root of tree/heap/array
         2. After building max-heap – largest value is stored at top – exchange this largest value with the value stored at location N so now the max is location N
         3. Shorten array by one and then recreate heap structure using fix-heap-down to push new value stored at root down to proper location
         4. Second largest value is now stored at root – exchange this with value at location N-1, shorten array again, and rebuild heap
         5. Have N items to extract from root and place at end of array – we pay logN to rebuild heap each time thus entire ordering takes NlogN time
   5. Worst, best, average cases
      1. Insertion sort – best case is O(N), but O(N^2) average case and worst case
      2. Insertion and bubble sort – worst case is N^2 and average case is no better
      3. Heap sort is NlogN for best/average/worst case
      4. Quicksort – has become the de facto standard sorting method because although worst case is N^2, average case is NlogN and its operating characteristics make it good for a variety of standard computing platforms
      5. If there are N values, then we can have N! possible permutations of those data values
   6. Merge sort
      1. Divide and conquer algorithm that requires 2N space in order to sort N items but completes in NlogN time
      2. Must be done out of place instead of in place – needs double room to sort
      3. We have a pointer into array A, B, and C each of which is initialized at beginning of array
         1. If value pointed to in the A array is smaller than the value pointed to in the B array, then we copy the value from the array A, bump the A pointer by one and bump the pointer C by one
         2. If the value pointed to in array B is the smaller, then we copy that value into C and bump the pointers for B and C instead of those for A and C
         3. As long as there is data in both arrays, we copy the smaller of the two values into the result array C and thus general merge the two sorted arrays into one sorted array.
         4. If one runs out of data then the other one is all in sorted order and the rest left is added to new array at the end
      4. Three basic facts about this central operation
         1. The question of equal keys doesn’t affect anything – if there are equal keys then we can choose entries from either array without affecting the fact that the eventual array is sorted
         2. No way to avoid extra storage space – best case – we are interleaving A and B other – only need fixed extra storage
            1. Worst case is A is all bigger than B and need to copy all of A over to new array then all of B over to new array.
         3. Merge runs in linear time – with 2 arrays of length N, we can perform merge sort in worst case 2N comparisons – perfectly interleaved data
      5. Coding would look like
         1. For(int phase =1; phase <=n;phase++) { merge the 2^(n-phase) pairs of arrays of length 2^(phase) }
      6. Merge sort is often written recursively
   7. Disk based sorting
      1. Streaming data is much faster than moving read head of disk drive so want to be able to stream data off hard disk
      2. Marge sort excels in reading sequential data at two points into one
   8. Quick sort
      1. De facto standard as the best compromise between worst-average case and execution characteristics on extant machines
      2. Merge sort – by splitting sorting problem in half for each phase we run logN phases
         1. Each phase comprises 2^(n-k) subsorts each is done on 2^k items thus requires N = 2^n = 2^k \* 2^(n-k) giving a running time of NlogN
         2. Negative feature is the double amount of memory needed to do this
      3. Quick sort does this recursive sorting but without the needed extra space
         1. Be able to fix one element in phase one, two elements in phase two, then four elements in phase 3…etc
         2. This makes N = logN phases each of which fixes k elements
         3. This creates NlogN with lgN phases each which take N time
      4. Negative is cannot guarantee recursive breakdown of the array will divide the array into equal halves at each step
      5. If each break has N and N-1 elements, then we will have N phases instead og logN and the total running time will be N^2 time
   9. Quicksort algorithm
      1. Alogorithm
         1. Choose a pivot element whose key is k – can be chosen to be and element in the array but most will chose the first or last
         2. Divide the full array into 3 subarrays – A has less than k, B has those equal to k, and C has those greater than k
         3. Recursively sort subarrays A and C and then return the elements of A and A, B, and C in that order
   10. Magic of quick sort
       1. Most important being the pivot point – a good choice splits the array in half, a bad choice might leave one array empty will all the elements in the other array
       2. Can sample three random values of an array and get the average of that to best guess a pivot point
   11. Average case for quicksort
       1. Ave case is NlgN
       2. Preferred sort in terms of using the physical hardware efficiently
   12. Sorting without comparisons
       1. If we happen to know that all values are between 1- 100 we can use a bucket sort
       2. Setup 100 buckets of variable size and instead of comparing keys, we would look at value and store the record in the right bucket for that key
       3. Only need a Boolean value in the array for duplicates
   13. Experimental results
   14. Auxiliary space and implementation issues
       1. In theory the most important aspect of choice of a sorting algorithm is the running time
       2. In real world applications, you might want space over time – ie very large amounts of data that will be sorted rarely
   15. Space consideration
       1. Merge sort needs double the data but is highly suitable for disk to disk sorting of very large files.
       2. All data is accessed in a purely sequential way
       3. Heapsort – access goes all over the place when sorting
   16. Memory
       1. Quicksort is standard in software libraries
          1. Memory access patterns are far superior than heapsort
          2. It largely runs in cache and not in main memory
       2. Heapsort’s memory access pattern work against the idea of cache
          1. Data accessed randomly and unlikely to be in cache
   17. Multi-level sorts
       1. Merge sort effective on large files, its greater in the later phases when merging large, sorted, arrays than in early phases.
       2. Can use quicksort to take advantage of cache then using mergesort to take advantage of pre organized data
   18. Stability
       1. Stable sort - if 2 records that have the same key end up being stored in the sorted array in the same order in which they are presented in the original array
       2. Heapsort is not stable
       3. Bubblesort and insertion soert are stable provided one gets the ‘less than’ and ‘less than or equal’ comparisons done properly.
          1. Records are not swapped if keys are the same and therefore are kept in original order
          2. If wrote code as if comparing the other way around then the code is not stable
       4. Quicksort is stable in same as bubblesort/insertionsort are stable in if one does not force exchange of records with equal keys
   19. The asymptotic of sorting
       1. Quicksort appears to have a better constant than heapsort and better characteristics on modern computers
   20. Lower bounds on sorting - NlogN
   21. Lower bounds on average case – NlogN
   22. Sorting in parallel
       1. In cluster computer – distributed memory parallel
          1. Head node computer farms out identical tasks to a collection of compute nodes – each an independent computer
          2. Transmission from computer to computer is the bottle neck of the system
          3. Ideally, exchange data not all at the same time so as to not overwhelm bandwidth
       2. Shared memory machine
          1. All processors have access to all memory locations in the machine
          2. The switch between processors and memory is expensive
       3. Sorting with these types
          1. Merge sort – extract sorted lists from processors and put it all back together
          2. Sending messages is expensive task – and others can be waiting while head node is putting pieces back together
   23. Analysis