**Analyzing Algorithms**

1. Function Growth
   * + Gives simple characterization of algorithm’s efficiency and allows to compare the relative performance of alternative algorithms
     + When input sizes are large enough to make only the order of growth of runtime relevant, then it is the asymptotic efficiency. This means that we see how the runtime of an algorithm increases with the size of input in the limit, as size of input increases without bound.
     + Asymptotic notations
       - Θ-notation



A picture containing diagram

Description automatically generated

* + - * + A function f(n) belongs to the set Θ((g(n)) if there exists positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n) for sufficiently large n. g(n) is asymptotically tight bound for f(n)
        + The definition requires that every member f(n) belongs to Θ((g(n)) be asymptotically non negative, f(n) ve positive whwnever n is sufficiently large
      * O-notation
        + Aysmptotic upper bound



* + - * + Using O notation we can often describe the running time of an algorithm merely by inspecting the algorithms overall structure
        + O-notation describes an upper bound, when we use it to bound the worst case running time of an algorithm, we have a bound on the running time of the algorithm on every input—the blanket statement we discussed earlier. Thus, the O.n2/ bound on worst-case running time of insertion sort also applies to its running time on every input.
      * Ω notation
        + asymptotic lower bound



* + - * Theorems
        + Logo, company name

          Description automatically generated with medium confidence
      * o-notation
        + Asymptotic upper bound provided by O notation may or may not be asymptotically tight
        + we use smaller notation to denote a upper bound that is not asymptotically tight



* + - * + the definition of O and o notations are similar the main difference is that in f(n) = O(g(n)) bound 0<= f(n) <= cg(n)holds for some constant c > 0, but in f(n) = o(g(n)) the bound holds for all constants c>0
        + intuitively in small or notation the function becomes insignificant relative to geofence as an approaches Infinity
      * w-notation
        + Denotes a lower bound that is not asymptotically tight

Text, letter

Description automatically generated

* + - * Comparing function

Text

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated



1. Time Complexities(Recursion)
   * + Check insists go hand in hand with the divide and conquer paradigm because they give us a natural way to characterize the running times of divide and conquer algorithms
     + a recurrence is an equation or inequality that describes a function in terms of its value on smaller inputs
     + they can take many forms
     + subproblems are not necessarily constrained to being a constant fraction of the original problem size
     + Analyzing an algorithm has come to mean predicting the resources that the algorithm requires. Occasionally, resources such as memory, communication bandwidth, or computer hardware are of primary concern, but most often it is computational time that we want to measure.
     + Generally, by analyzing several candidate algorithms for a problem, we can identify a most efficient one
     + Worst case gives na upper bound on the runtime for any input.
     + An average case is often as bad as worst case.
     + One algorithm is more fficient than the other if its worst case runtime has lower order of growth
     + When an algorithm contains a recursive call to itself, we can often describe its running time by a recurrence equation or recurrence, which describes the overall running time on a problem of size n in terms of the running time on smaller inputs
     + For T(n) runtime on a problem of size n. If problem size is small enough, n<= c for a constant c, the solution takes constant time Θ(1).
     + Division of problem yields ‘a’ subproblems, each of which is 1/b size of original. Then it takes time T(n/b) to solve one subproblem of size n/b. So, to solve a subproblems, it takes aT(n/b) time.
     + If D(n) is time to divide problem into subproblems and C(n) time to combine the solutions to subproblems into solution for the problem then recurrence is

A picture containing company name

Description automatically generated

* + - this chapter offers three methods for solving recurrences that is for obtaining asymptotic big Theta or big O bounds on the same solution
      * the substitution method we guess abound and then use mathematical induction to prove our guess correct
      * the recursion tree method converts the recurrence in a tree whose nodes represent the cost incurred at various levels of the recursion we use techniques for bounding summations to solve the recurrence
      * the master method provides bounds for recurrences of the T(n) = aT(n/b)+f(n), where a >= 1, and b > 1, and f(n) is a given function. This equation characterizes a divide and conquer algorithm that creates ‘a’ sub problems each of which is ‘1/b’ the size of the original problem and in which the divide and combine steps together take f(n) time. To use the master method you will need to memorize 3 cases but once you do that you will be easily able to determine asymptotic bounds for many simple recurrences. We will use the master method to determine the running times of the divide and conquer algorithms for the maximum subarray problem and for matrix multiplication as well as some other algorithms based on the divide and conquer
    - Substitution method for solving recurrences
      * The substitution method for solving recurrences comprises 2 steps
        + guess the form of the solution
        + use mathematical induction to find constant and show that the solution works
      * we substitute the guest solution for the function when applying the inductive hypothesis to smaller values hence the name substitution method this method is powerful but we must be able to guess the form of the on answer in order to apply it
      * we can use the substitution method to establish either upper or lower bounds on a recurrence

Text, letter

Description automatically generated

* + - Recursion tree
      * Although you can use the substitution method to provide a sufficient proof that a solution to a recurrence is correct you might have trouble coming up with a good guess
      * drawing out a recursion tree as we did in our analysis of the merge sort recurrent serves as straightforward way to devise a good guess in a recursion tree each node represents the cost of a single subproblem somewhere in the set of recursive function invocations
      * we sum the cost within each level of the tree to obtain a set of per level cost and then we sum all the per level cost to determine the total cost of all levels of the recursion
      * a recursion tree is best used to generate a good gas which you can then verify by substitution method. When using a recursion tree to generate a good guess you can often tolerate a small amount of sloppiness since you will be verifying your guess later on. If you are very careful when drawing out a recursion tree and summing the cost however you can use a recursion tree as a direct proof of a solution to a recurrence

Diagram

Description automatically generated

* + - Master method for solving recurrences
      * Theorems

Text

Description automatically generated

Text, letter

Description automatically generated

Text

Description automatically generated

Text

Description automatically generated

Text, letter

Description automatically generated

Sorting Algorithms

1. Insertion Sort

* Input: A sequence of n numbers (a1, a2, a3...., an)
* Output: A permutation(reordering) (a1’, a2’, ……an’) of the input sequence, which is sorted
* Keys: Numbers that we wish to sort
* Efficient algorithm for sorting small number of elements.
* Like sorting hands of playing cards. Start with an empty left hand, cards face down on table. Remove one card at a time from table and insert in correct position in the left hand. To find the correct position of the card, compare it with cards already in left hand (right to left). All cards in the left hand are sorted.
* Sorts **in place: rearranges numbers within the array, at most a constant number is stored outside of array**
* Comparison sort: determines sorted output by comparing elements
* Pseudocode
  + Input: An array containing sequence of length ‘n’ to be sorted

Text

Description automatically generated

Text

Description automatically generated

* **Loop invariants**: show correctness of insertion sort. A[1…..j-1] holds sorted elements, A[j+1….n] holds unsorted elements.
  + **Initialization**: It is true prior to the first iteration of the loop.
    - We start by showing that the loop invariant holds before the first loop iteration, when j = 2.The subarray A[1….j-1], therefore, consists of just the single element A[1], which is in fact the original element in A[1]. Moreover, this subarray is sorted (trivially, of course), which shows that the loop invariant holds prior to the first iteration of the loop.
  + **Maintenance**: If it is true before an iteration of the loop, it remains true before the next iteration.
    - Next, we tackle the second property: showing that each iteration maintains the loop invariant. Informally, the body of the for loop works by moving A[j-1], A[j-2], A[j-3] and so on by one position to the right until it finds the proper position for A[j] (lines 4–7), at which point it inserts the value of A[j] (line 8). The subarray A[1…j] then consists of the elements originally in A[1…j], but in sorted order. Incrementing j for the next iteration of the for loop then preserves the loop invariant.
  + **Termination**: When the loop terminates, the invariant gives us a useful property that helps show that the algorithm is correct.
    - Finally, we examine what happens when the loop terminates. The condition causing the for loop to terminate is that j >A.length = n. Because each loop iteration increases j by 1, we must have j = n + 1 at that time. Substituting n + 1 for j in the wording of loop invariant, we have that the subarray A[1….n] consists of the elements originally in A[1…n], but in sorted order. Observing that the subarray A[1….n] is the entire array, we conclude that the entire array is sorted. Hence, the algorithm is correct.
* Analysis
  + The time taken by the INSERTION-SORT procedure depends on the input: sorting a thousand numbers takes longer than sorting three numbers
  + can take different amounts of time to sort two input sequences of the same
  + size depending on how nearly sorted they already are.
  + In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input.
  + The running time of an algorithm on a particular input is the number of primitive operations or “steps” executed

Text

Description automatically generated with low confidence

Text

Description automatically generated with medium confidence

* + Best case: Array is already sorted, linear time O(n)

Text, letter

Description automatically generated

* + Worst case: Array is reverse sorted, quadratic O(n2)

Text, letter

Description automatically generated

* + Running time is fixed for a given input

1. Divide and conquer approach
   1. Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem.

Text

Description automatically generated

1. Merge Sort Algorithm
   1. Follows the divide and conquer paradigm.
   2. Divides n-element sequence to be sorted into subsequences of n/2 elements each. Conquers by sorting the subsequences recursively using merge sort. Combines the sorted subsequences to produce sorted answer.
   3. The recursion “bottoms out” when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.
   4. The key operation of the merge sort algorithm is the merging of two sorted sequences in the “combine” step. We merge by calling an auxiliary procedure MERGE(A, p, q, r), where A is an array and p, q, and r, are indices into the array such that p< q<r. The procedure assumes that the subarrays A(p….q) and A(q+1….r) are in sorted order. It merges them to form a single sorted subarray that replaces the current subarray. This takes Θ(n) where n = r-p+1 is the total number of elements merged

Text, letter

Description automatically generated

Chart, box and whisker chart

Description automatically generated

Chart, box and whisker chart

Description automatically generated

* 1. At the start of each iteration of the for loop of lines 12–17, the subarray A[p…..k-1] contains the k - p smallest elements of L[1….n1+1] and R[1….n2+1],in sorted order. Moreover, L[i] and R[j] are the smallest elements of their arrays that have not been copied back into A.

Text

Description automatically generated

Text, letter

Description automatically generated

Text, letter

Description automatically generated

Diagram

Description automatically generated

* 1. Merge sort runtime can be analyzed using recurrence relation. Here, a = 2, and b = 2.
  2. Merge sort on just element takes constant time. For n>1 elements

Text

Description automatically generated

Worst case runtime of merge sort

A picture containing company name

Description automatically generated

A picture containing logo

Description automatically generated

Diagram, engineering drawing

Description automatically generated

* 1. This worst case runtime is Θ(nlog2n)
  2. Logarithmic functions grow more slowly than any linear function. So, for large enough inputs, merge sort outperforms insertion sort.
  3. Merge sort does not operate in place
  4. Comparison sort: determines sorted output by comparing elements

1. Heap sort
   1. Sorts n numbers in place in O(nlgn) time, using heap data structure which is implemented using priority queue
   2. Comparison sort: determines sorted output by comparing elements
   3. Sorts in place
   4. Heaps
      1. The binary heap data structure is an array object that we can view as a nearly complete binary tree each node of the tree corresponds to an element of the array. The tree is completely filled on all levels except possibly the lowest which is filled from the left up to a point
      2. an array A that represents a heap is an object with two attributes A.length which usually gives the number of elements in the array and the A.heap\_size which represents how many elements in the heap are stored within an array A

Diagram

Description automatically generated with medium confidence

* + 1. There are two kinds of binary heaps that is Max heap and men heaps. In both kinds the value in the node satisfy A heap property the specifics of which depend on the kind of heap. In a Max heap the Max heap property is that for every node I other than the root parent is greater than index that is the value of node is at most the value of its parent. Thus the largest element in a Max heap is stored at the root and the subtree rooted at a node contains values no longer than the contained at the node itself. A minimum heap is organized in the opposite way the minimum heap property is that for every node I other than the root parent is less than index. The smallest element in a min heap is at the root
    2. for the heap sort algorithm we use Max heaps. Minimum heaps commonly implement priority queues.
    3. Viewing A heap as a tree redefined the height of a node in a heap to be the number of edges on the longest simple downward path from the node to a leaf and redefine the height of the heath to be the height of its root
    4. since a heap of N elements is based on a complete binary tree its height is that Θ(lgn)
    5. Methods
       1. MAX\_HEAPIFY : runs in O(lgn) time, key to maintaining the max-heap property
       2. BUILD\_MAX\_HEAP: runs in O(n) produces maz heap from an unordered input array
       3. HEAPSORT: runs in O(nlgn) sorts an array in place
       4. MAXHEAP\_INSERT, HEAP\_EXTRACTMAX, HEAP\_INCREASE\_KEY, HEAP\_MAXIMUM, run in O(lgn) time, allow the heap data structure to implenment priority queue
    6. Maintaining the heap property
       1. In order to maintain the Max heap property we call the procedure Max heapify. Its input are an array A and an index i into the array. When it is called Max heapify assumes that the binary trees rooted at left and right are Max heaps but that A[i] might be smaller than its children thus violating the maxi property. Max heapify lets the value at a I float down in the Max heap so that the subtree rooted at index I obeys the Max heap property
       2. Runtime on a node of height h is O(h)

Text, letter

Description automatically generated

* + 1. Building a heap
       1. We can use the procedural Max heapify in a bottom up manner to convert an array into a Max heap
       2. the procedure built marks heat goes through the remaining nodes of the tree and runs Max heapify on each one

Text, letter

Description automatically generated

* + - 1. Each call to MAXHEAPIFY costs O(lgn) time, and BUILDMAXHEAP makes O(n) such calls. Thus, the runtime is O(nlgn). Its is O(h) for height h
  1. Heapsort algorithm
     1. The heap sort algorithm starts by building the Max heap on the input array. Since the maximum of the array is stored at the root we can put it into its correct final position. If we now discard node N from the heap we can do so by incrementing the heap size we observed that the children of the root remain Max heaps but the new root element might violate the maxi property all we need to do is restore the Max heap property however skull Max heapify which leaves the Max heap in a , 1. Which leaves Max heap in A[1….n-1]. The heap sort algorithm that repeats this procedure for the Max heap of size-n-1 down to a heap of size 2

Text, letter

Description automatically generated

1. Quick sort
   1. Sorts n numbers in place, with worst case runtime of O(n2). Expected runtime is Θ(nlgn), generally outperforms heap sort. Has tight code, so hidden constant factor in runtime is small. Sorts large input arrays
   2. Comparison sort: determines sorted output by comparing elements
   3. Quick sort like merge sort applies the divide and conquer paradigm.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

* 1. Performance
     1. Runtime depends on whether the partitioning is balanced or unbalanced, which depends on elements used for partitioning
     2. If partitioning is balance the algorithm runs asymptotically as fast as merge sort
     3. if partitioning is unbalanced however it can run asymptotically as slowly as insertion sort
     4. worst case partitioning: the worst case behavior of quicksort occurs when the partitioning routine produces once a problem with n-1 elements and one with zero elements. The running time is Θ(n2). This occurs when the array is already sorted.
     5. Best case partitioning: partition produces 2 subproblems, each of size no more than n/2, since one is size n/2 and other is n/2-1. Here it runs much faster. The running time is Θ(nlgn).
     6. Balanced partitioning: Average case runtime is much closer to best case than to worse case.
     7. Average case: still O(nlgn) with slightly larger constant hidden by O-notation

1. Counting sort
   1. Assumes input numbers are in set {0,1….k}. using array indexing as tool for determining relative order
   2. Can sort n numbers in Θ(k+n) time, when k = O(n), runs in time that is linear in size of input array, Θ(n)
   3. Counting sword determines for each input element X the number of elements less than X. It uses this information to place element X directly into its position in the output array.

Text

Description automatically generated

Box and whisker chart

Description automatically generated

* 1. The runtime is Θ(k+n). In practice, we use counting sort when k = O(n), so runtime is Θ(n)
  2. It beats lower bound of OMEGA(nlgn) as its not comparison sort.
  3. It is stable, numbers with same value appear in the output arrat in same order as input array
  4. Used as a subroutine in radix sort

1. Radix sort
   1. Extended counting sort
   2. n integers to sort, each integer has d digits and each digit can take on up to k possible values then radix sort can sort the numbers in Θ(d(n+k)), time
   3. when d is a constant and k is O(n) radix sort runs in linear time
   4. Sorts on the least significant bit

Text

Description automatically generated with medium confidence

Text

Description automatically generated with medium confidence

1. bucket sort
   1. requires knowledge of the probabilistic distribution of numbers in the input array it can sort n real numbers uniformly distributed in the half open interval in alf open interval [0,1) in average case O(n) time
   2. As you know start the input is drawn from a uniform distribution and has an average case running time of O(n). Like counting sword bucket sword is fast because it assumes something about the input
   3. bucket sort assumes that the input is generated by a random process that distributes elements uniformly and independently over the interval [0, 1)
   4. buckets or divides the interval into n equal size sub intervals or buckets and then distribute the n input numbers into the buckets
   5. since the inputs are uniformly and independently distributed over the interval we do not expect many numbers to fall into each bucket
   6. to produce the output we simply sort the numbers in each bucket and then go through the buckets in order listing the elements in each

Graphical user interface, text, application

Description automatically generated with medium confidence

Graphical user interface

Description automatically generated with low confidence

* 1. Runs in linear time THETA(n)
  2. Even if input is not drawn from a uniform distribution, bucket sort may still run in linear time

Table

Description automatically generated

**Trees**

1. **Binary search tree**
   1. Search free data structure support many dynamic set operations including search minimum maximum predecessor successor insert and. Thus we can use a search tree both as a dictionary and a priority queue.
   2. Basic operations on a binary search tree takes time proportional to the height of the tree. For a complete binary tree with N nodes such operations run in worst case runtime. If the tree is a linear chain of N nodes however the same operation take worst case time
   3. the expected height of a randomly built binary search tree is so that the basic dynamic set operations on such a tree take time on average
   4. what is a binary search tree?
      1. Binary search tree is organized as the name suggests in a binary tree. We can represent such a tree by linked data structure in which each node is an object. In addition to a key and satellite data each node contains not to build left right and parent that point to the nodes corresponding to its left child it's the right child and its parent respectively. If a child or the parent is missing the appropriate attribute contains the value nil. The root node noticed the only node in the tree whose parent is nil.
      2. The keys in binary search tree are always stored in such a way as to satisfy the binary search tree property.
      3. if X is the node in a binary search tree. If Y is a node in the left subtree of X, then Y.Key <= X.Key. If Y is a node in the right subtree of X then Y.key >= X.key

A picture containing text, accessory, necklet

Description automatically generated

* + 1. Binary search tree properly allows us to print out all the keys in a binary search tree in sorted order by a simple recursive algorithm, called an inorder tree walk. This algorithm is so named because it prints the key of the root of a subtree. It takes Θ(n) time to walk an n-node binary search tree, since after the initial call, the procedure calls itself recursively exactly twice for each node in the tree-once for its left child and once for its right child.

Text, letter

Description automatically generated

* 1. Querying a binary search tree
     1. We often need to search for a key stored in a binary search tree. Besides the search operation binary search trees can support multiple queries.
     2. Searching

Text, letter

Description automatically generated

Text, letter

Description automatically generated

* + - 1. given a pointer to the root of a tree and a key K the tree search returns it pointer to a node with the key K if one exists otherwise, it returns nil
      2. the procedure begins its search at the root and traces a simple path downward in the tree. For each node X it encounters, it compares the k with x.key if the two keys are equal, the search terminates. If K is smaller than x.key such continues in the left subtree of eggs, since the binary subtree property implies that K could not be stored in the right subtree. Symmetrically if K is larger than X.key the search continues in the right subtree. The nodes encountered during the recursion form is simple path downward from the root of the tree and thus the running trying of tree search is big O(H) where H is the height of the tree
    1. Minimum and maximum
       1. we can always find an element in a binary search tree whose key is a minimum by falling left child pointers from the root until we encounter a nil.
       2. Binary search tree property in guarantees that tree minimum is correct. If a node X has no left subtree then since every key in the right subtree of X is at least as large as X.key the minimum key in the subtree rooted at X is X.key. If null X has the left subtree then since no key and the left right subtree is smaller than its key and every key in the left subtree is not larger than X.key, the minimum key in the subtree rooted at X resides in the subtree rooted at X.left

Text, letter

Description automatically generated

Text, letter

Description automatically generated

* + - 1. both of these procedures run in O(H) time on a tree of height age since as in tree search the sequence of nodes encountered forms a simple downward path from the root
    1. successor and predecessor
       1. given a node in a binary search tree sometimes we need to find its successor in the sorted order determined by an inorder tree walk. If all keys are distinct the successor of a node X is the node with the smallest key greater than X.KEY. The structure of a binary search tree allows us to determine the successor of a node without ever comparing keys. The following procedure returns the successor of a node X in a binary search tree if it exists and nil if X has the largest key in the tree
       2. the code for tree successor is broken into two cases. If the right subtree of node X is non empty then the successor of X is just the left mouse node in X's right subtree, which we find by calling tree minimum on X.right
       3. On the other hand, if the right subtree of node X is empty and X has a successor why, then why is the lowest ancestor of X whose left child is also an ancestor of X. To find Y, these simply go up from X until we encounter a node that is the left child of its parent

Text, letter

Description automatically generated

* + - 1. the running time of three successor on the tree of height age is OH, since we either follow a simple path of the tree or follow a simple path down the tree. The procedure tree predecessor is symmetric to true successor and also run in O(H)
      2. even if the keys are not distinct we define the successor and predecessor of a node X as the node returned by the calls to tree successor and tree predecessor respectively
    1. insertion and deletion
       1. the operations of the insertion and deletion caused the dynamics that represented by a binary search tree to change. The data structure must be modified to reflect this change but in such a way that the binary search group property continues to hold.
       2. Insertion
          1. to insert a new value V into a binary search tree T we used the procedure tree insert. The procedure takes a node Z for which Z.key =V, Z.left = NIL, Z.RIGHT = NIL. Modifies T on some of the attributes of Z in such a way that it inserts Z into an appropriate position in the tree

Text, letter

Description automatically generated

Takes O(h) time

* + - 1. Deletion
         1. The overall strategy for deleting a node Z from a binary search tree T has three base cases

if Z has no children then we simply remove it by modifying its parent to replace Z with nil as its child

if Z has just one child, then we elevate that child to take Z’s position in the tree by modifying Z’s parent to replace Z by Z's child

if Z has two children, we find Z's successor Y, which must be in Z's right subtree and have Y take Z’s position in the tree. The rest of Z’S original right subtree becomes Y’S new right subtree and Z’S left subtree becomes Y’S new left subtree. This case is the tricky one because it matters whether Y is Z ‘S right child

* + - * 1. the procedure for deleting a given node from a binary search tree takes argument pointers to T and Z. It organizes its cases a bit differently from the three cases outlined previously by considering the four cases

if Z has no left child then we will place Z by its right child which may or may not be NIL. When Z’s right child is nil, this case deals with the situation in which Z has no children. When Z’s right child is not empty, this case handles the situation in which Z has just one child which is its right child

if Z has just one child which is its left child then we replace Z by its left child

otherwise Z has both a left and a right child we find Z's successor Y, which lies in Z right subtree and has no left child. We want to splice Y out of its current location and have it replaced Z in the tree

Diagram

Description automatically generated

if Y is Z right child then we replace Z by Y, leaving Y’S right child alone

otherwise Y lies within Z’s right subtree but is not Z's right child. In this case we first replace Y by its own right child and then we replace Z by Y.

In order to move subtrees around within the binary search tree which define a subroutine transplant, which replaces one subtree as a child of its parent with another subtree. Then transplant replaces the subtree rooted at node U with the subtree rooted at node V, node U’S parent become node V's parent and U’S parent ends up having V as its appropriate child.

Text, letter

Description automatically generated

With the transplant procedure in hand your is the procedure that deletes node Z from binary search tree T

Text, letter

Description automatically generated

Takes O(h) time

1. **Red-Black Trees**
   1. A binary search tree of height h can support any of the basic dynamics that operations in O(h) time. Thus they said operations are fast if the height of search tree is small. If its height is large, however the set operations may run no faster than with a linked list. Red black trees are one of many such trees schemes that are balanced in order to guarantee that basic dynamics and operations take o(lgn) time in the worst case
   2. properties of red black trees
      1. red black tree is a binary search tree with one extra bit of storage per node that is its color which can be either red or black. By constraining the node colors on any simple path from the root to a leaf, red black trees ensure that no such path is more than twice as long as any other so that the tree is approximately balanced.
      2. Each node of the tree now contains the attributes color key left right and p if a child or the parent of another does not exist the corresponding pointer attribute of the node contains empty. We shall regard these empty as being pointers to the leaves of the binary search tree and the normal key bearing nodes as being internal nodes of the tree a red black tree is a binary tree that satisfies the following red black properties
         1. every node is either red or black
         2. the root is black.
         3. Every leaf is black
         4. if a node is red then both its children are black
         5. for each node all simple paths from the node to descendant leaves contain the same number of black nodes
   3. Call the number of black nodes on any simple path from but not including in your eggs down to a leaf the black height of the node denoted by bh(x). By property 5, the notion of black height is well defined since all descending simple paths from the node have same number of black nodes. We define the black height of a red black tree to be the black height of its root
   4. Rotations
      1. The SEARCH tree operations(TREE-INSERT, TREE-DELETE) when run on a RBT with n keys, take O(lgn) time. They modify the tree and can violate the RBT property. To restore the properties, some nodes’ colors need to be changed and change some pointer structure. The pointer structure is changed using ROTATION, a local operation which preserves the RBT property.
      2. There are two kinds of rotations
         1. Left rotation: If performed on node X, assume its right child Y is not T.nil, it pivots around the link from X to Y. Makes Y as new root of the subtree and X as Y’s left child, and Y’s OG left child becomes X’s new right child

Diagram

Description automatically generated

Text, letter

Description automatically generated

A picture containing diagram

Description automatically generated

* + - 1. Right rotate is symmetric
    1. Both rotations run in O(1) time, only pointers are changes by a rotation, all other attributes in the node remain the same
  1. Insertion
     1. A node can be inserted in n-node rbt in O(lgn) time
     2. Use slightly modified version of TREE-INSERT, insert node using former, then color new node as red. Then to check RBT property call auxiliary procedure RB\_INSERT\_FIXUP to recolor nodes and perform rotations

Text, letter

Description automatically generated

Text

Description automatically generated

* + 1. INSERT\_FIXUP
       1. Violates the property 2(root node is black) and property 4(red node cannot have a red child), due to Z being red. Property 2 if Z is root and $ if Z’s parent is red

Chart

Description automatically generated

* + - 1. Cases
         1. Case 1: When Z and Z’s parent is red, Z’s uncle is red. Recolor nodes and move pointer Z up the tree

A picture containing chart

Description automatically generated

* + - * 1. Case 2: When Z and Z’s parent are red, Z’s uncle is black, Z is right child of Z.p, perform a left rotation
        2. Case 3: When Z is red, Z’s uncle is black and Z is left child of its parent, recolor and right rotation

Diagram, schematic

Description automatically generated

* + 1. Analysis: Tree’s height for n nodes is O(lgn), RBT\_INSERT takes O(lgn)+O(lgn) = O(lgn) time
  1. Deletion
     1. Takes O(lgn) time
     2. Based on tree-delete procedure but with additional lines of pseudocode to keep track of node y that can violate RBT property

Text, letter

Description automatically generated

* + 1. When deleting node Z who has fewer than two children, Z is removed from tree and y becomes Z.
    2. When Z has two children then Y should be Z’s successor and Y moves into Z’s position in tree. Remember Y’s color before removing or moving within the tree, keep track of X which moves into Y’s position. Then call RB\_DELETE\_FIXUP to change color and perform rotations to restore RBT properties

**A close-up of a document

Description automatically generated with low confidence**

**Table

Description automatically generated with low confidence**

* + 1. Cases
       1. x’s sibling is red
          1. Node w, sibling of x is red. w must have black children, switch colors of w and x.p and perform left rotation on x.p, new sibling of x(which is w’s child prior to rotation) is now black
       2. Node w is black, both w’s children are black
          1. Take one black off both x and w, leaving x with only one black and leaving w red. Add extra black to x.p , new node x is red-black, value c of color attribute of new node x is red, after loop termination, color new node x black
       3. W is black, w’s left is red and right is black
          1. Switch colors of w and w.left, perform right rotation on w, new sibling of x is now black with red right child
       4. W is black, w right is red
          1. Make color changes, perform left roation on x.p, remove extra black on x, make it singly black. Set x to root and terminate.

**Data Structure**

1. **Dynamic Programming**
   1. Applies to optimization problems where we make set of choices to arrive at an optimal solution.
   2. For each choice, subproblems of same form arise
   3. Efficient when given subproblem may arise from more than one partial set of choices, store solution to each subproblem
   4. So;ves problems bt comnining solutions to subproblems
   5. Solving
      1. Characterize structure of an optimal solutopn
      2. Recursively define value of an optimal solution
      3. Compute value of an optimal solution in bottom up fashion
      4. Construct an optimal solution from computed information
   6. Rod cutting
      1. Where to cut steel rods
      2. Rod can be cut in 2n-1 ways
      3. Recursive top-down implementation

Text

Description automatically generated

* + 1. It is inefficient cause it calls itself over and over again with same parameter valyes, T(n) = 2n
    2. Using dynamic programming
       1. Arrange to solve each subproblem only once and save its solution
       2. Uses additional memory to save computation time, time-memory tradeoff
       3. Two approaches – top down with memoization(recursive but saves solution), bottim up(sort problems by size and solve them in size order, smallest first)
       4. Memoized Cut rod

Text, letter

Description automatically generated

Text

Description automatically generated

* + - 1. Bottom up cut rod

Text

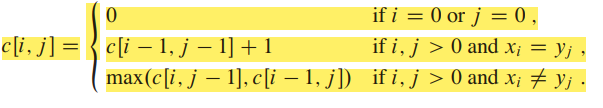
Description automatically generated

* + - 1. Time complezity is THETA(n2)
  1. Longest common subsequence(LCS)
     1. Characterizing LCS
        1. Enumerate all subsequences of X and check if theyre of Y, take exponential time 2n
        2. Recurive solution
           1. If xm = yn, find lcs of xm-1 and yn-1, append this to LCS
           2. If xm =/= ym

Find LCS of Xm-1, Y

LCS of X, Yn-1

Select the longest



* + - 1. Computing length of an LCS

Text, letter

Description automatically generated

* + - * 1. THETA(mn) distinct problems, take 2 sequences as input. Store c[I, j] value in table, compute entries in row-major order, maintain table b to construct an optimal solution, b[I,j] points to table entry corresponding to optimal subproblem solution when computing c[I,j]
      1. Constructing an LCS

A picture containing chart

Description automatically generated

Text, letter

Description automatically generated

* + - * 1. Take THETA(m+n) time

1. **Greedy Algorithm**
   1. Also for optimization, make set of choices to arrive at an optimal solution
   2. Make each choice in locally optimal manner