Search and Sort	worst	best	average	space
Selection Sort	O(n^2)	O(n^2)	O(n^2)	O(1)
Insertion Sort	O(n^2)	O(n)	O(n^2)	O(1)
Merge Sort	O(nlgn)	O(nlgn)	O(nlgn)	O(n)
Quicksort	O(n^2)	O(nlgn)	O(nlgn)	O(lgn)
Heapsort	O(nlgn)	O(n)	O(nlgn)	O(1)
Counting Sort	O(n)	O(n)	O(n)	O(n)
Radix Sort	O(n)	O(n)	O(n)	O(n)
Bubble Sort	O(n^2)	O(n)	O(n^2)	O(1)
Linear Search	O(n)	O(1)	O(n)	O(1)
Binary Search	O(lgn)	O(1)	O(lgn)	O(1)

Asymptotic Notation

Big O notation: asymptotic "less than": f(n) "≤" g(n)

Omega notation: asymptotic "greater than": f(n) "≥" g(n)

Theta notation: asymptotic "equality": f(n) "=" g(n)

1. Prove that 100n + 5 = O(n^2) => 100n + 5 ≤ 100n + n = 101n ≤ 101n^2 for all n ≥ 5, n0=5 and c=101 is a solution **(Big O)**
2. 5n^2 = Omega(n) 100n + 5 ≠ Omega(n^2) All c, n0 such that: 0 <= cn <= 5n^2 => cn <= 5n^2 => c = 1 and n0 = 1 **(Big Omega)**
3. N^2/2 – n/2 = Theta(n2) => ½ n^2 - ½ n ≤ ½ n^2 in n ≥ 0 => c2= ½ => ½ n^2 - ½ n ≥ ½ n^2 - ½ n * ½ n (all n ≥ 2) = ¼ n^2 => c1= ¼**(Theta)**

- $f(n) = \log n^2$; $g(n) = \log n + 5$ — $f(n) = \Theta(g(n))$
- $f(n) = n$; $g(n) = \log n^2$ — $f(n) = \Omega(g(n))$
- $f(n) = \log \log n$; $g(n) = \log n$ — $f(n) = O(g(n))$
- $f(n) = n$; $g(n) = \log^2 n$ — $f(n) = \Omega(g(n))$
- $f(n) = n \log n + n$; $g(n) = \log n$ — $f(n) = \Omega(g(n))$
- $f(n) = 10$; $g(n) = \log 10$ — $f(n) = \Theta(g(n))$
- $f(n) = 2^n$; $g(n) = 10n^2$ — $f(n) = \Omega(g(n))$
- $f(n) = 2^n$; $g(n) = 3^n$ — $f(n) = O(g(n))$

Mathematical Induction to prove statement is true –

Basis step: prove that the statement is true for n = 1

Inductive step: assume that S(n) is true and prove that S(n+1) is true for all n ≥ 1, then

Find case n "within" case n+1

Prove that: 2n + 1 ≤ 2n for all n ≥ 3 • Basis step: n = 3: 2 * 3 + 1 ≤ 2^3 => 7 ≤ 8 TRUE

• Inductive step: Assume inequality is true for n, and prove it for (n+1): 2n + 1 ≤ 2^n must prove: 2(n + 1) + 1 ≤ 2^(n+1) => 2(n + 1) + 1 = (2n + 1) + 2 ≤ 2^n + 2 ≤ 2^n + 2^n = 2^(n+1), since 2 ≤ 2^n for n ≥ 1

Recurrence

T(n) = T(n-1) + n => O(n^2)

T(n) = T(n/2) + c => O(log(n))

T(n) = T(n/2) + n => O(n)

T(n) = 2T(n/2) + 1 => O(n)

Substitution method - Guess a solution. Use induction to prove that the solution works

1. **T(n) = c + T(n/2)** => **Guess**: T(n) = O(lgn) **Induction goal**: T(n) ≤ d lgn, for some d and n ≥ n0 **Induction hypothesis:** T(n/2) ≤ d lg(n/2) **Proof of induction goal:** T(n) = T(n/2) + c ≤ d lg(n/2) + c = d lgn – d + c ≤ d lgn if: – d + c ≤ 0, d ≥ c
2. **T(n) = 2T(n/2) + n**=>**Guess**:T(n)=O(nlgn) **Induction goal:** T(n) ≤ cn lgn, for some c and n ≥ n0. **Induction hypothesis**: T(n/2) ≤ cn/2 lg(n/2). **Proof of induction goal:** T(n) = 2T(n/2) + n ≤ 2c (n/2)lg(n/2) + n = cn lgn – cn + n ≤ cn lgn if - cn + n ≤ 0 =>c ≥1
3. **T(n) = T(n-1) + n** => **Guess**: T(n) = O(n2) **Induction goal**: T(n) ≤ c n2, for some c and n ≥ n0 **Induction hypothesis:** T(n-1) ≤ c(n-1)2 for all k < n. **Proof of induction goal**: T(n) = T(n-1) + n ≤ c (n-1)2 + n = cn2 – (2cn – c - n) ≤ cn2 if: 2cn – c – n ≥ 0 => c ≥ n/(2n-1) => c ≥ 1/(2 – 1/n). For n ≥ 1 => 2 – 1/n ≥ 1 Þ any c ≥ 1 will work

Master's method - "Cookbook" for solving recurrences of the form - $T(n) = aT\left(\frac{n}{b}\right) + f(n)$ where, a ≥ 1, b > 1, and f(n) > 0.

Case 1: if $f(n) = O(n^{\log_b a - \varepsilon})$ for some ε > 0, then: $T(n) = \Theta(n^{\log_b a})$

Case 2: if $f(n) = \Theta(n^{\log_b a})$, then: $T(n) = \Theta(n^{\log_b a} \lg n)$

Case 3: if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some ε > 0, and if af(n/b) ≤ cf(n) for some c < 1 and all sufficiently large n, then: $T(n) = \Theta(f(n))$

1. **T(n) = 2T(n/2) + root(n)**, a = 2, b = 2, log2(2) = 1 Compare n with f(n) = n^(1/2) => f(n) = O(n^(1-e)) Case 1 => T(n) = theta(n)
2. **T(n) = 2T(n/2) + n**, a = 2, b = 2, log2(2) = 1,Compare nlog2(2) with f(n) = n => f(n) = theta(n) => Case 2 => T(n) = theta(nlgn)
3. $T(n) = 27T(n/3) + \Theta(n^3 \lg n)$
 $n^{\log_3 27} = n^3$ vs. $n^3 \lg n$
 Use Case 2 with $k = 1 \Rightarrow T(n) = \Theta(n^3 \lg^2 n$
4. $T(n) = 5T(n/2) + \Theta(n^3)$
 $n^{\log_2 5}$ vs. n^3
 Now $\lg 5 + \epsilon = 3$ for some constant $\epsilon > 0$
 Check regularity condition (don't really need to since $f(n)$ is a polynomial
 $af(n/b) = 5(n/2)^3 = 5n^3/8 \le cn^3$ for $c = 5/8 < 1$
 Use Case 3 $\Rightarrow T(n) = \Theta(n^3)$

Iteration Method - Iterate the recurrence until the initial condition is reached. Use back-substitution to express the recurrence in terms of n

1. **T(n) = c + T(n/2)** => c + c + T(n/4) = c + c + c + T(n/8). => Assume n = 2^k T(n) = c + c + … + c + T(1) = clgn + T(1) = Θ(lgn)
2. **T(n) = n + 2T(n/2)** = n + 2(n/2 + 2T(n/4)) = n + n + 4T(n/4) = n + n + 4(n/4 + 2T(n/8)) = n + n + n + 8T(n/8) Assume: n = 2k => in + 2i T(n/2i) = kn + 2kT(1) = nlgn + nT(1) = Θ(nlgn)

Recursion-tree method - Convert the recurrence into a tree: Each node represents the cost incurred at various levels of recursion. Sum up the costs of all levels.

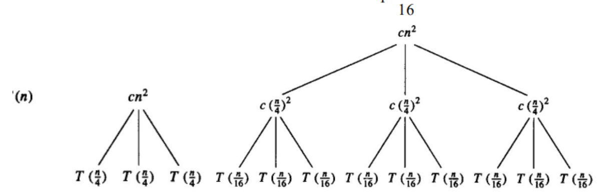
1. **T(n) = 3T(n/4) + cn^2** => Sub problem size at level i is: n/4i.

Sub problem size hits 1 when 1 = n/4i => i = log4n.

Cost of a node at level i = c(n/4^i)^2. Number of nodes at level i = 3i => last level has 3log4 n = nlog4 3 nodes.

Total Cost - T(n) = O(n^2)

$$(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) \le \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i cn^2 + \Theta\left(n^{\log_4 3}\right) = \frac{1}{1 - \frac{3}{16}} cn^2 + \Theta\left(n^{\log_4 3}\right) = O(n^2)$$

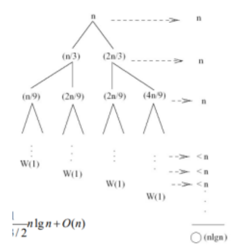


2. **W(n) = W(n/3) + W(2n/3) + n =>** The longest path from the root to a leaf is: (2/3)n -> (2/3)^2(n) -> … -> 1. Sub-problem size hits 1 when 1 = (2/3)^i(n) => i=log3/2(n). Cost of the problem at level i = n.

Total cost: W(n) = O(nlgn)

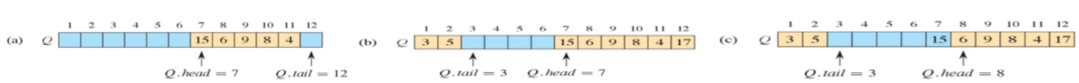
$$W(n) < n + n + \ldots = \sum_{i=0}^{(\log_{3/2} n) - 1} n + 2^{(\log_{3/2} n)} W(1) <$$

$$< n \sum_{i=0}^{\log_{3/2} n} 1 + n^{\log_{3/2} 2} = n \log_{3/2} n + O(n) = n \frac{\lg n}{\lg 3/2} + O(n) = \frac{1}{\lg 3/2} n \lg n + O(n)$$



Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Queue - ENQUEUE: add an element. **DEQUEUE**: remove an element. The queue has a **head and a tail**. When an element is enqueued, it takes its place at the tail of the queue. The element dequeued is always the one at the head of the queue. (first-in, first-out)

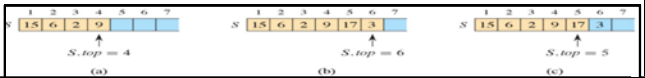


Q.head = Q.tail, => the queue is **empty**

Q.head = Q.tail = 1 => attempt to dequeue – **underflow**

n Q.head = Q.tail + 1 or both Q.head = 1 and Q.tail = Q.size => attempt to enqueue – **overflow**

Stack - INSERT operation on a stack is often called **PUSH**. DELETE operation, which does not take an element argument, is often called **POP. S.top**, indexing the most recently inserted element. **S.size** is the size n of the array**. S[1: S.top],** where **S[1]** is the element at the bottom of the stack and **S[S.top]** is the element at the top.



S.top = 0, the stack contains no elements and is empty. - an attempt to pop – **underflow**
S.top exceeds S.size, the stack **overflows**

Binary Tree - Attributes p, left, and right to store pointers to the parent, left child, and right child of each node in a binary tree.
x.p = NIL, then **x is the root.**
node **x has no left child**, then x.left = NIL.
node **x has no right child**, then x.right = NIL.
T.root = NIL, then the **tree is empty**

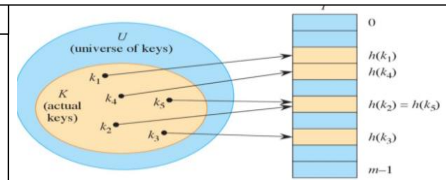
Binary Search Tree - Attributes p, left, and right to store pointers to the parent, left child, and right child of each node in a binary tree.
x.p = NIL, then **x is the root.**
node **x has no left child**, then x.left = NIL.
node **x has no right child**, then x.right = NIL.
T.root = NIL, then the **tree is empty**

Each **node** contains the attributes: Key, left, right, parent
Binary-search-tree property
1. If y is in left subtree of x, **then y.key <= x.key**
2. If y is in right subtree of x, **then y.key >= x.key**

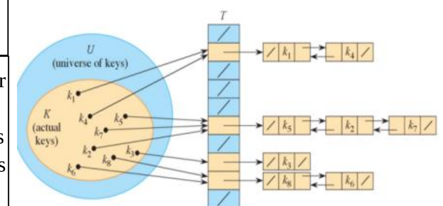
Pre-Order - Display **data** of root**. Traverse left subtree** calling preorder function**. Traverse right subtree** calling preorder function.
In-Order – **Traverse left subtree** calling preorder function. Display **data** of root. **Traverse right subtree** calling preorder function.
Post-Order - Traverse left subtree calling preorder function. Traverse right subtree calling preorder function. Display data of root.
Successor of a node x: node y has the smallest key s.t. y.key > x:key

minimum key of a binary search tree is located at the **leftmost node**, and the **maximum key** of a binary search tree is located at the **rightmost** node.

Hash Table - effective for implementing a **dictionary**. generalization of an ordinary array. , we find the element whose key is k by just looking in the **k-th position of the array** à **direct addressing.** applicable when we **can afford to allocate an array with one position for every possible key**. Use a hash table when the **number of keys actually stored is small** relative to the number of possible keys. Instead of storing an element with key k in slot k, use a function h and store the element in slot h(k). **h(k): Hash function.** k hashes to slot h(k), h(k) is the hash value of key k.

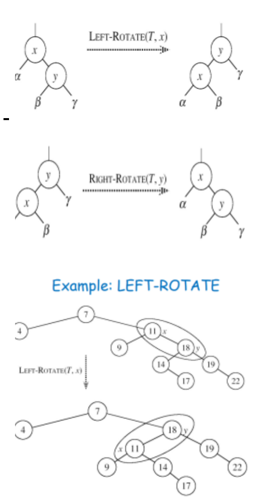


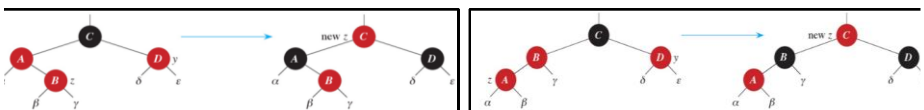
Collision - Two keys may hash to the same slot. • For a given set K of keys with |K| <= m, may or may not happen.
Collision Resolution by Chaining - Each nonempty hash-table slot T[j] points to a linked list of all the keys whose hash value is j



Red-Black Tree - "**Balanced**" binary search trees guarantee **an O(lgn) running time**. Additional attribute for its nodes in tree: color which can be **red or black.** Ensures that **no path is more than twice as long as any other path** -> the tree is balanced.
Red-Black-Trees Properties – 1. Every node is either red or black. 2. The root is black. 3. Every leaf (NIL) is black. 4. If a node is red, then both its children are black. No two consecutive red nodes on a simple path from the root to a leaf. For each node, all paths from that node to descendant leave contain the same number of black nodes. (Satisfy the binary search tree property)
Height of tree - 2lg(n + 1)

Rotation –

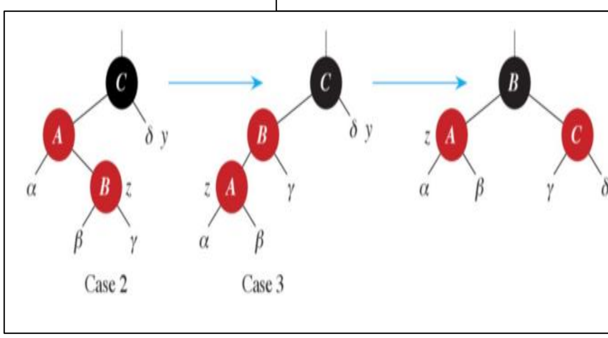




z is a right child

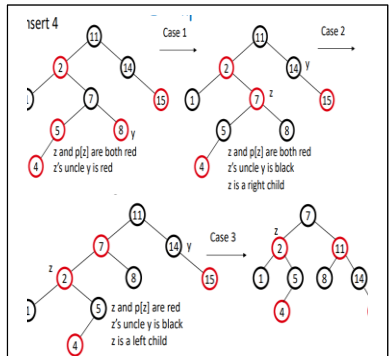
z is a left child

Case 1 (Implmentation)
1. Color p[z] <- black 2. Color y <- black
3. Color p[p[z]] <- red 4. z = p[p[z]]



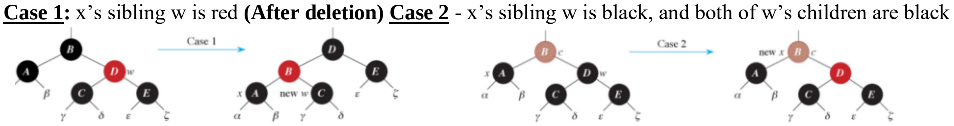
Case 2 (Implementation) - z's "uncle" (y) is black and z is a right child
1. z <- p[z]
2. LEFT-ROTATE(T, z)

Case 3 (Implementation)
1.color p[z] <- black
2.color p[p[z]] <- red
3.RIGHT-ROTATE(T, p[p[z]])

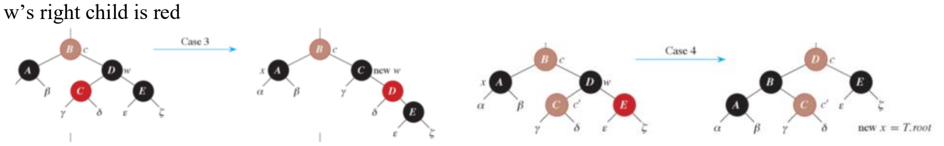


Insertion – insert a node and color it Red And, recolor and rotate node to fix tree.
Deletion – If node to be deleted is red, just delete it and done. But, If node to be deleted is black, then delete it and then perform recolor and rotate node to fix tree

Case 1: x's sibling w is red **(After deletion)** **Case 2** - x's sibling w is black, and both of w's children are black



Case 3: x's sibling w is black, w's left child is red, and w's right child is black **Case 4** - x's sibling w is black, and w's right child is red



Dynamic Programming - Solve problems by **combining the solutions to sub problems**. Used for optimization problems: **Find a solution with the optimal value**. **Four-Step Method**: 1. Characterize the structure of an optimal solution. 2. Recursively defines the value of an optimal solution. 3. Compute the value of an optimal solution, typically in a bottom-up fashion. 4. Construct an optimal solution from computed information.

```
EMOIZED-CUT-ROD(p, n)
let r[0:n] be a new array      // will remember solution values in r
for i = 0 to n
    r[i] = −∞
return MEMOIZED-CUT-ROD-AUX(p, n, r)

EMOIZED-CUT-ROD-AUX(p, n, r)
if r[n] ≥ 0                    // already have a solution for length n?
    return r[n]
if n == 0
    q = 0
else q = −∞
    for i = 1 to n     // i is the position of the first cut
        q = max{q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r)}
r[n] = q               // remember the solution value for length n
return q
```

Heap Sort –1.Create max Heap. 2.Remove largest item. 3.Place item in sorted array

Quick Sort (Pivot)divide-and-conquer

Rod Cutting problem - Rod lengths are integer number of inches and Each cut is free.

an determine optimal revenue r_n by taking the maximum of

p_n: the price we get by not making a cut,

$r_1 + r_{n-1}$: the maximum revenue from a rod of 1 inch and a rod of $n-1$ inches,

$r_2 + r_{n-2}$: the maximum revenue from a rod of 2 inches and a rod of $n-2$ inches, ...

$r_{n-1} + r_1$.

hat is,

$= \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$.

Dynamic solution -
1. Instead of solving the same sub problems repeatedly, **arrange to solve each sub problem just once**. 2. **Save the solution to a sub problem** in a table, and refer back to the table whenever we revisit the sub problem. 3. Two basic approaches: **top-down with memorization, and bottom-up**