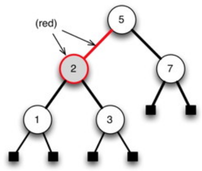
Order of Growth: $n^n > n! > a^n > n^a > n\lg n > n > \lg n$

A **full binary tree** is a binary tree in which each vertex either is a leaf or has exactly two nonempty descendants. In a full binary tree of height *h*:

1. # leaves = (# internal vertices) + 1.
2. # leaves is at least *h*+1 *(first example figure)* and at most 2^h *(second example figure)*.
3. # internal vertices is at least *h* *(first example)* and at most 2^h-1 *(second example)*.
4. Total number of vertices (summing the last two results) is at least 2*h*+1 *(first example)* and at most 2^{h+1}-1 *(second example)*.
5. Height *h* is at least lg(*n*+1)-1 *(second example)* and at most (*n*-1)/2 *(first example)*

Red-Black Tree Properties

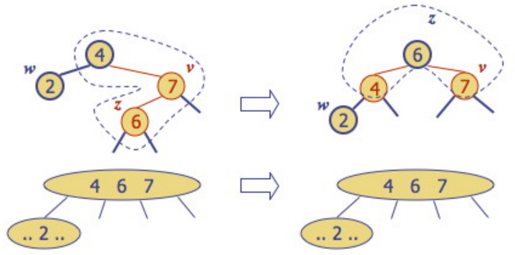
A red-black tree (RBT) is a binary search tree with the following additional properties:



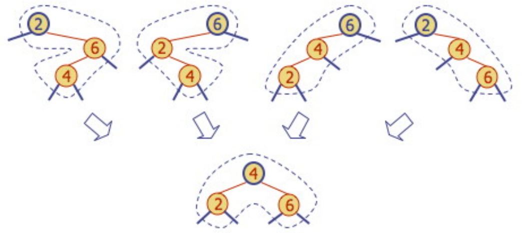
1. **Color property**: Every node is either red or black. *(We can indicate this either by coloring the node or by coloring its parent link.)*
2. **Root property**: The root is black
3. **External property**: Every leaf is black.
4. **Internal property**: If a node is red, then both of its children are black. *(Hence, no two reds in a row are allowed on a simple path from the root to a leaf.)*

Restructuring INSERTION

Restructuring remedies a child-parent double red when the parent red node has a black sibling. It restores the correct representation (internal property) of a 4-node, leaving other RBT and BST properties intact:



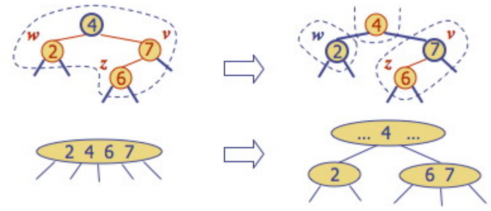
There are four restructuring configurations depending on whether the double red nodes are left or right children. They all lead to the same end configuration of a black with two red children:



After a restructuring, the double red has been remedied without violating any of the other properties *(you should verify this)*: there is no need to propagate changes upwards. Notice that the height of the subtree tree has been reduced by one. ***This is the operation that keeps the trees balanced to within a constant factor of lg(n) height***, by ensuring that the height of the RBT is no more than twice that of the (necessarily balanced) 2-4 tree it represents. *Do you see why?*

Recoloring

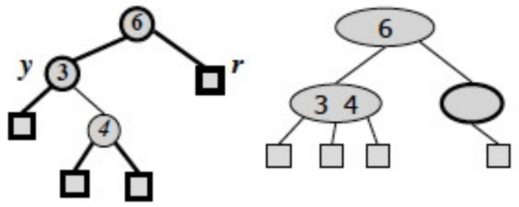
Recoloring remedies a child-parent double red when the parent red node has a red sibling. The parent *v* and its sibling *w* become black and the grandparent *u* becomes red, unless it is the root.



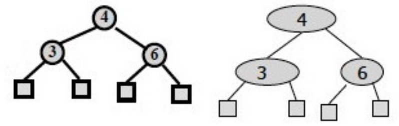
It is equivalent to performing a split on a 5-node in a (2,4) tree. (When there is a double red and yet another red in the parent's sibling, we are trying to collect too many keys under the grandparent.) For example, the RBT recoloring on the top corresponds to the (2,4) transformation on the bottom:

Depth property: For each node, all the paths from the node to descendant leaves contain the same number of black nodes (the **black height** of the node).

DELETION Case 1: *y* is black and has a red child: Perform a RBT **restructuring**, equivalent to a (2,4) **transfer**, and we are done. For example, if we have the RBT on the left corresponding to underflow in the (2,4) tree on the right:

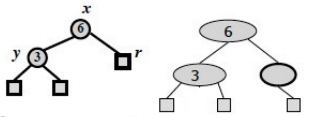


... we do the following restructuring:

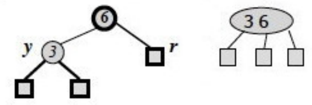


Case 2: *y* is black and its children are both black: Perform a RBT **recoloring**, equivalent to a (2,4) **fusion**, which may propagate up the double black violation. If the double-black reaches the root we can just remove it, as it is now on *all* of the paths from the root to the leaves, so does not affect property 5, the depth property.

For example, if we have the RBT on the left corresponding to underflow in the (2,4) tree on the right:

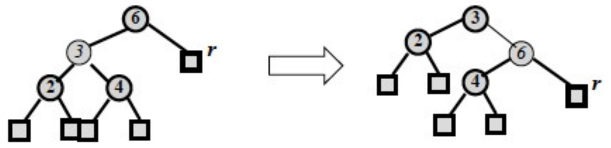


... we do the following recoloring: the black node *y* is colored red, and the double black node *r* is colored ordinary black:



The root of the above subtree takes on an extra black, which propagates only if it was previously black and is not the root. If it was red it merely turns black; if it was the root the extra black no longer affects the balanced black height of the tree.

Case 3: *y* is red: Perform a RBT **adjustment**, equivalent to choosing a different representation of a 3-node, after which either Case 1 or Case 2 applies.



Stable sorts: O(n)

Counting Sort

Radix Sort

Bucket Sort

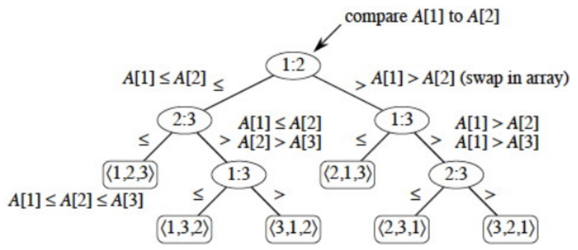
Algorithm	Worst-case running time	Average-case/expected running time
Insertion sort	$\Theta(n^2)$	$\Theta(n^2)$
Merge sort	$\Theta(n \lg n)$	$\Theta(n \lg n)$
Heapsort	$O(n \lg n)$	—
Quicksort	$\Theta(n^2)$	$\Theta(n \lg n)$ (expected)
Counting sort	$\Theta(k + n)$	$\Theta(k + n)$
Radix sort	$\Theta(d(n + k))$	$\Theta(d(n + k))$
Bucket sort	$\Theta(n^2)$	$\Theta(n)$ (average-case)

Quicksort works best on random data

Best/Avg. Case: Θ(n lg n) // Worst Case: $\Theta(n^2)$

Randomized Quicksort for nearly sorted data O(n lg n)

Example Decision Tree



Bounds on Sorting

At least n! leaves: l >= n!

Binary tree: l <= 2^h leaves

Union Find $\Theta(n^2)$

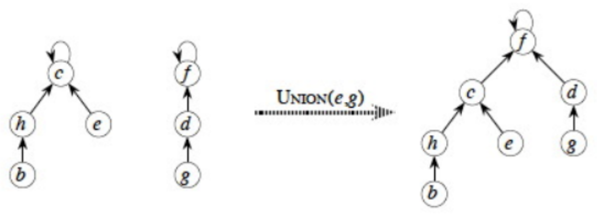
O(m α(n)) for a sequence of m

O(α(n)) per operation

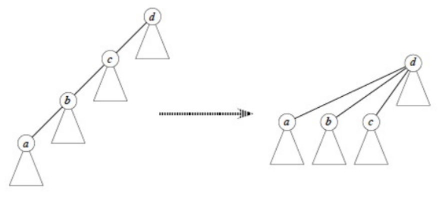
Forest Representations of Disjoint Sets

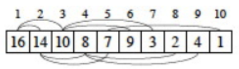
- Each tree represents a set
- The root of the tree is the set representative
- Each node points only to its parent (no child pointers needed)

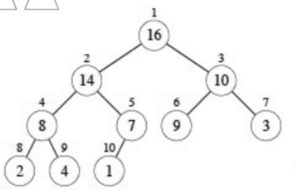
- The root points to itself as parent



Union by Rank: make root of smaller tree a child of the root of the larger tree
Path Compression: When running Find-Set(x), make all nodes on the path from x to the root direct children of the root.





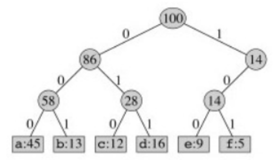


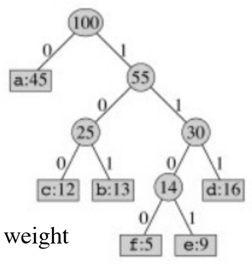
Heap
- Root of the tree is A[1]
- Parent of A[i] is A[i/2]
- Left Child of A[i] is A[2i]
- Right Child of A[i] is A[2i+1]

Optimal Solution requires finding a/the best set of alternatives
Dynamic Programming
Bottom-up: solves subproblems first
Makes choice knowing optimal solutions to subproblems
Greedy Algorithm
Top-down: makes local choice before subproblems
Cannot handle overlapping subproblems
Amortized Analysis guarantees the average case
T(n) of worst case/n = average case
Amortized Cost = amount we charge each operation (overcharges)
Huffman's Code

	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Fixed length: Variable length:

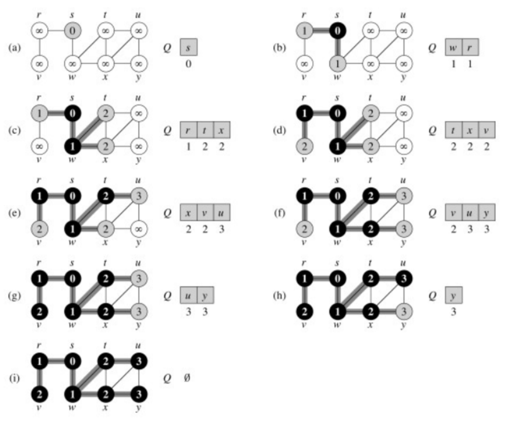




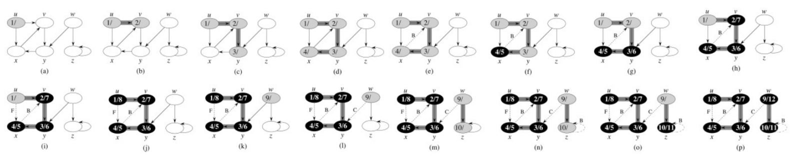
Kruskal's Algorithm O(E lg V)
- Sort by weight
- Build MST starting with lowest weight
- Do not create cycles

Prim's Algorithm O(E lg V)
- Start from root
- Check neighbors and select lowest weight edge
- Do no create cycles

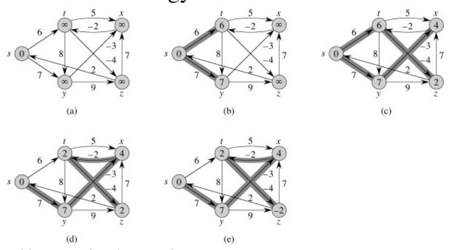
Breadth-First Search O(V + E)



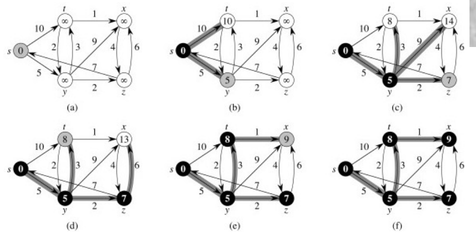
Depth-First Search Θ(V + E): prioritize by alphabetical order



Transpose Graph: prioritize by longest finish time
Topological Sort of a DAG is a linear ordering of vertices such that if (u,v) in E then u appears somewhere before v in the ordering
Shortest Paths Problems
Bellman-Ford Algorithm O(V E)
Allows negative-weight cycles
Brute force strategy



Dijkstra's Algorithm
Cannot handle negative weight cycles
O((V+E)lgV)
Connected: O(ElgV)



All-Pairs Shortest Paths
Johnson's Algorithm O(VElgV)
Make a new vertex 0 that points to all existing vertices
Update vertices to shortest path then apply the following:
prev. weight edge + weight(u) – weight(v)
Floyd-Warshall's Algorithm O(n^3)

$$D_1 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & \infty & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 9 & 0 \end{array} \qquad D_3 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 12 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

$$D_2 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 8 & 9 & 1 \\ 2 & \infty & 0 & 1 & \infty \\ 3 & 4 & 12 & 0 & 5 \\ 4 & \infty & 2 & 3 & 0 \end{array} \qquad D_4 = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 0 & 3 & 4 & 1 \\ 2 & 5 & 0 & 1 & 6 \\ 3 & 4 & 7 & 0 & 5 \\ 4 & 7 & 2 & 3 & 0 \end{array}$$

Maximum Flow

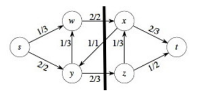
- **Capacity Constraint**: $\forall\, u, v \in V,\ 0 \le f(u, v) \le c(u, v)$.
  (Can't push more over an edge than its capacity.)
- **Flow Conservation**: $\forall\, u \in V - \{s, t\}$,

$$\sum_{v \in V} f(v,u) = \sum_{v \in V} f(u,v) \qquad |f| = \sum_{v \in V} f(s,v) - \sum_{v \in V} f(v,s)$$

(Flow into a vertex must equal flow out of it, except for the source and sink.)

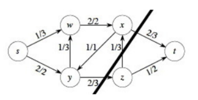
Consider the cut $S = \{s, w, y\}$, $T = \{x, z, t\}$ in the network shown.

$$\begin{aligned} f(S,T) &= \underbrace{f(w,x) + f(y,z)}_{\text{from } S \text{ to } T} - \underbrace{f(x,y)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3\,. \\ c(S,T) &= \underbrace{c(w,x) + c(y,z)}_{\text{from } S \text{ to } T} \\ &= 2 + 3 \\ &= 5\,. \end{aligned}$$



Now consider the cut $S = \{s, w, x, y\}$, $T = \{z, t\}$.

$$\begin{aligned} f(S,T) &= \underbrace{f(x,t) + f(y,z)}_{\text{from } S \text{ to } T} - \underbrace{f(z,x)}_{\text{from } T \text{ to } S} \\ &= 2 + 2 - 1 \\ &= 3\,. \\ c(S,T) &= \underbrace{c(x,t) + c(y,z)}_{\text{from } S \text{ to } T} \\ &= 3 + 3 \\ &= 6\,. \end{aligned}$$



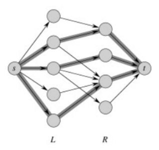
Max-Flow Min-Cut Theorem
1. f is maximum flow
2. G_f has no augmenting path
3. |f| = c(S,T) for some cut (S,T)

Ford Fulkerson Algorithm O(Ef*)
Edmonds-Karp Algorithm O(VE^2)
Maximum Bipartite Matching O(VE)

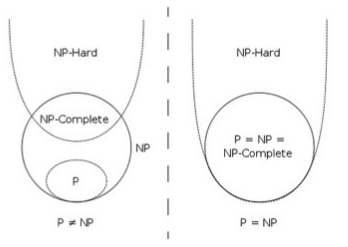
Given G, define flow network $G' = (V', E')$:

- $V' = V \cup \{s, t\}$.
- $E' = E$ augmented with edges from s to every $u \in L$ and from every v in R to t.
- $c(u, v) = 1\ \forall\ (u, v) \in E'$.

Then just run Ford-Fulkerson (Edumunds-Karp is not required, as all edges have unit value):



This works because a maximum flow must use the maximum number of (unitary capacity) edges across the cut (L, R).



NP-Completeness
Tractability and Decidability

- **Tractable problems**, generally considered to be those solvable in polynomial time (most of the problems considered so far in this course).
- Examples: Any problem studied this semester for which we have solutions bounded by nk for a fixed constant k.
- Some solutions are not strictly polynomial but are bounded above by a polynomial (e.g., n lg n for sorting is bounded by n2), so sorting is considered polynomial.

- **Intractable problems** are those solvable in super-polynomial but not polynomial time. Today we will be concerned with the question of which problems are in this class.
    - Examples: Enumerate all binary strings of length n; Compute the power set (set of all sets) of n items. The solutions consist of 2n binary strings, or 2n sets, respectively, so even if the solution were available instantly, exponential time would be required just to output the result!
    - Unknown: Integer linear programming, finding longest simple paths in a graph, finding the optimal way to schedule tasks on factory machines under scheduling and deadline constraints, determining whether certain boolean formulas can be simultaneously satisfied, optimizing circuit layouts under certain constraints, finding the cheapest way to visit all of a set of cities without repeating any and returning to your starting city, finding the optimal partition of graphs under a modularity metric, and many many more ...
- **Unsolvable problems** for which no algorithm can be given guaranteed to solve all instances of the problem. These problems are also known as Undecidable.
    - Example: the halting problem (given a description of a program and arbitrary input, decide whether the program will halt on that input), demonstrated by Alan Turing.

P deonotes the class of problems solvable in polynomial time (P is a subset of NP). **NP** denotes the class of problems for which solutions are verifiable in polynomial time.
If **NP-Hard** problems are solved, then any problem in NP can be solved by reduction to the NP-Hard problem.
NP-Hard problems that are also in NP are said to be **NP Completed (NPC)**, if any NPC problem can be solved in polynomial time then all problem in NP can too

Accepting and Deciding Formal Languages
By casting computational problems as decision problems, theorists can use concepts from formal language theory in their proofs. We are not doing these proofs but you should be aware of the basic concepts and distinctions:
A **language** L over an alphabet Σ is a set of strings made up of symbols from Σ. For example, if Σ = {0, 1}, the set L = {10, 11, 101, 111, 1011, 1101, 10001, ...} is the language of binary representations of prime numbers.
The language that contains all strings over Σ is denoted Σ*. For example, if Σ = {0, 1}, then Σ* = {ε, 0, 1, 00, 01, 10, 11, 000, ...}, where ε denotes an empty string.
An algorithm A **accepts** a string x ∈ {0, 1}* if given x the output of A is 1.
The **language L accepted by A** is the set of strings L = {x ∈ {0, 1}* : A(x) = 1}. But A need not halt on strings not in L. It could just never return an answer. (The existence of problems like the Halting Problem necessitate considering this possibility.)
A language is **decided** by an algorithm A if it accepts precisely those strings in L and rejects those not in L (i.e., A is guaranteed to halt with result 1 or 0).
A language is **accepted in polynomial time** by A if it is accepted by A in time O(nk) for every encoded input of length n and some constant k. Similarly, a language is **decided in polynomial time** by A if it is decided by A in time O(nk).

Polynomial Time Verification
A **complexity class** is a set of languages for which membership is determined by a complexity measure. (Presently we are interested in the running time required of any algorithm that decides L, but complexity classes can also be defined by other measures, such as space required.) For example, we can now define P more formally as:
P = {L ⊆ {0, 1}* : ∃ algorithm A that decides L in polynomial time}.
A **verification algorithm** A(x, y) takes two arguments: an encoding x of a problem and a certificate y for a solution. A returns 1 if the solution is valid for the problem. (A need not solve the problem; only verify the proposed solution.)
The **language verified** by a verification algorithm A is
L = {x ∈ {0, 1}*: ∃ y ∈ {0, 1}* such that A(x, y) = 1}.
We can now define the complexity class **NP** as the class of languages that can be verified by a polynomial time algorithm, or formally:
L ∈ NP iff ∃ polynomial time algorithm A(x, y) and constant c such that:
L = {x ∈ {0,1}* : ∃ certificate y with |y| = O(|x|c) such that A(x,y) = 1}.
The constant c ensures that the size of the certificate y is polynomial in the problem size, and also we require that A runs in time polynomial in its input, which therefore must be polynomial in both |x| and |y|

NP Completeness Defined
A language L ⊆ {0, 1}* is **NP-Complete (in NPC) if**

1. **L ∈ NP, and**
2. **Every L' ∈ NP is polynomial reducible to L.**

Languages satisfying 2 but not 1 are said to be NP-Hard. (This includes optimization problems that can be converted to decision problems in NP.)
The major Theorem of this lecture is:
If any NP-Complete problem is polynomial-time solvable, then P = NP.
Equivalently, **if any problem in NP is not polynomial-time solvable, then no NP-Complete problem is polynomial time solvable.**

Vertex Cover
A vertex cover of an undirected graph G = (V, E) is a subset V' ⊆ V such that if (u, v) ∈ E then u ∈ V' or v ∈ V' or both.
Each vertex "covers" its incident edges, and a vertex cover for G is a set of vertices that covers all the edges in E. For example, in the graph on the right, V' = {v, z} or V' = {v, w, y} are vertex covers. Note V' = V = {u, v, w, x, y, z} is also a vertex cover.
The **Vertex Cover Problem** is to find a vertex cover of minimum size in G. Phrased as a decision problem,
VERTEX-COVER = {⟨G, k⟩ : graph G has a vertex cover of size k}
There is a straightforward reduction of CLIQUE to VERTEX-COVER, illustrated in the figure. Given an instance G=(V,E) of CLIQUE, one computes the complement of G, which we will call Gc = (V,Ē), where (u,v) ∈ Ē iff (u,v) ∉ E. The graph G has a clique of size k iff the complement graph has a vertex cover of size |V| – k. (Note that this is an existence claim, not a minimization claim: a smaller cover may be possible.)

- **If** direction (If G has a k-clique, then Gc has a vertex cover of size |V| – k):
- We show that none of the k vertices in the clique need to be in the cover.
- They are all connected to each other in G, so none of them will be connected to each other in Gc.
- Thus, every edge in Gc must involve at least one vertex not in the clique, so the clique vertices can be excluded from the cover: we can use vertices from the remaining |V| – k vertices to cover all the edges in Gc.
- The minimum vertex cover may be smaller than |V| – k, but we know that |V| – k will work.

Only if direction (If Gc has a vertex cover of size |V| – k, then G has a k-clique):

- We will use proof by contrapositive to show that there is no clique of size k in G, then there is no vertex cover of size |V| – k in Gc.
- Assume for the sake of contradiction that there is no k-clique in G, but there is a vertex cover V' in Gc of size |V'| = |V| – k.
- The non-existence of a k-clique in G means that at least two vertices in every subset of k vertices are not connected in G.
- Consider the subset V \ V', i.e. a subset of k vertices that are not part of the presumed (|V| – k)-sized vertex cover of Gc. By the above, there exist at least two vertices in this subset (let's call them u and v), such that there is no edge (u,v) in G.
- But if edge (u,v) is not in G, it must exist in Gc (the complement of G).
- But if there is an edge (u,v) in Gc, and neither u nor v are in the vertex cover, then edge (u,v) is uncovered and V' is not a valid vertex cover.

We reached a contradiction, therefore, if G does not contain a k-clique, Gc cannot contain a vertex cover of size |V| – k

Clique

A **clique** in an undirected graph G = (V, E) is a subset V' ⊆ V, each pair of which is connected by an edge in E (a complete subgraph of G). (Examples of cliques of sizes between 2 and 7 are shown on the right.)

The **clique problem** is the problem of finding a clique of maximum size in G. This can be converted to a decision problem by asking whether a clique of a given size k exists in the graph:

CLIQUE = {⟨G, k⟩ : G is a graph containing a clique of size k}

One can check a solution in polynomial time. (How?)

3-CNF-SAT is reduced to CLIQUE by a clever reduction illustrated in the figure. Given an arbitrary formula ϕ in 3-conjunctive normal form with k clauses, we construct a graph G and ask if it has a clique of size k as follows:

We create a vertex for every literal in ϕ

For every pair of vertices, we create an edge between them if the corresponding literals are in different triples and the literals are consistent (i.e., one is not the negation of the other).

If there are k clauses in ϕ, we ask whether the graph has a k-clique. The claim is that such a clique exists in G if and only if there is a satisfying assignment for ϕ:

If direction (If clique exists in G, then there is a satisfying assignment in ϕ):

- By definition of k-clique, existence of a k-clique implies there are k vertices in G that are all connected to each other.
- By our construction, the fact that two vertices are connected to each other means that they can receive a consistent boolean assignment (we can assign 1 to all of them), and that they are in different clauses.
- Since there are k vertices in the clique, then at least one literal in each of the k clauses can be assigned a 1, i.e., the formula ϕ can be satisfied.

Only if direction (If ϕ can be satisfied, then there is a clique in G):

- If ϕ can be satisfied, then we can assign values to the literals, such that at least one literal in each clause is assigned value 1. I.e. they are consistent.
- Consider the vertices corresponding to those literals. Since the literals are consistent and they are in different clauses, there is an edge between every pair of them.
- Since there are k clauses in ϕ we have a subset of at least k vertices in the graph with edges between every pair of vertices, i.e., we have a k-clique in G.
- Any arbitrary instance of 3-CNF-SAT can be converted to an instance of CLIQUE in polynomial time with this particular structure. That means if we can solve CLIQUE we can solve any instance of 3-CNF-SAT, and since we know that 3-CNF-SAT is NPC, transitively we can solve any instance in NP in polynomially related time. Be sure you understand why mapping an arbitrary instance of CLIQUE to a specialized instance of 3-CNF-SAT would not work.

Approximation Algorithms

There are three broad approaches to handing NP-Complete or NP-Hard problems in practice:

1. **Stick with small problems**, where the total execution time for an optimal solution is not bad. Your boss rejects this as it would limit the configuration options the company offers.
2. **Find special cases** of the problem that can be solved in polynomial time (e.g., 2-CNF rather than 3-CNF). It requires that we know more about the structure of the problem. We don't know much about iThingies, but we will use some restrictions to help with the third approach ...
3. **Find near-optimal solutions with approximation algorithms**. Your boss thinks it just might work: since the problem is hard, customers won't realize you haven't given them the optimal solution as long as a lot of their requests are met. This is the approach we'll examine today.

Definitions

Let C be the cost of a solution found for a problem of size n and C* be the optimal solution for that problem.

Then we say an algorithm has an **approximation ratio of ρ(n)** (that's "rho") if

C/C* ≤ ρ(n) for minimization problems: the factor by which the actual solution obtained is larger than the optimal solution.

C*/C ≤ ρ(n) for maximization problems: the factor by which the optimal solution is larger than the solution obtained

The CLRS text says both of these at once in one expression shown to the right. The ratio is never less than 1 (perfect performance).

An algorithm that has an approximation ratio of ρ(n) is called a ρ(n)-approximation algorithm.

An **approximation** scheme is a parameterized approximation algorithm that takes an additional input ε > 0 and for any fixed ε is a (1+ε)-approximation algorithm. (ε is how much slack you will allow away from the optimal 1-"approximation".)

An approximation scheme is a **polynomial approximation scheme** if for any fixed ε > 0 the scheme runs in time polynomial in input size n. (We will not be discussing approximation schemes today; just wanted you to be aware of the idea. See section 35.5)

String Matching

Naïve (Brute Force) O((n – m + 1)m)

Finite State Automaton O(m³ |∑|)

Compute-Prefix-Function Θ(m)

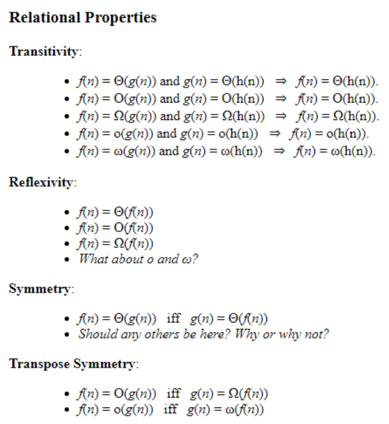
KMP-Matcher Θ(n)

Rabin-Karp Best: O(n) // Worst Θ((n - m + 1)m)

Binary Search Trees O(n lg n)

A **full binary tree** is a binary tree in which each vertex either is a leaf or has exactly two nonempty descendants. In a full binary tree of height h:

1. # leaves = (# internal vertices) + 1.
2. # leaves is at least h+1 (first example figure) and at most 2h (second example figure).
3. # internal vertices is at least h (first example) and at most 2h-1 (second example).
4. Total number of vertices (summing the last two results) is at least 2h+1 (first example) and at most 2h+1-1 (second example).
5. Height h is at least lg(n+1)-1 (second example) and at most (n-1)/2 (first example)



Relational Properties

Transitivity:

- $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$.
- $f(n) = O(g(n))$ and $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$.
- $f(n) = \Omega(g(n))$ and $g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$.
- $f(n) = o(g(n))$ and $g(n) = o(h(n)) \Rightarrow f(n) = o(h(n))$.
- $f(n) = \omega(g(n))$ and $g(n) = \omega(h(n)) \Rightarrow f(n) = \omega(h(n))$.

Reflexivity:

- $f(n) = \Theta(f(n))$
- $f(n) = O(f(n))$
- $f(n) = \Omega(f(n))$
- *What about o and ω?*

Symmetry:

- $f(n) = \Theta(g(n))$ iff $g(n) = \Theta(f(n))$
- *Should any others be here? Why or why not?*

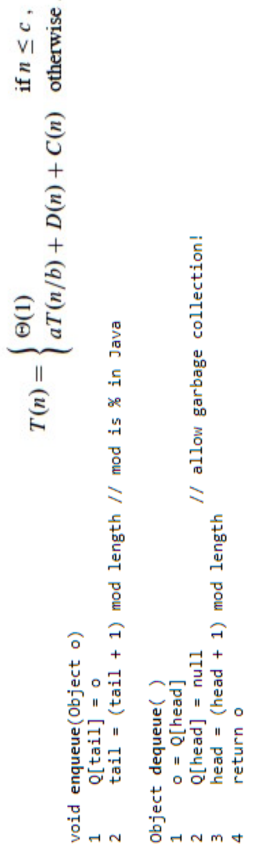
Transpose Symmetry:

- $f(n) = O(g(n))$ iff $g(n) = \Omega(f(n))$
- $f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

```
boolean isEmpty ( )
    if head == tail
      return TRUE
    else
      return FALSE

void enqueue(Object o)
    Q[tail] = o
    if tail == length
      tail = 1           // wrap around
    else
      tail = tail + 1

Object dequeue( )
    o = Q[head]
    if head == length
      head = 1
    else
      head = head + 1
    return o
```



$$T(n) = \begin{cases} \Theta(1) & \text{if } n \le c\,, \\ aT(n/b) + D(n) + C(n) & \text{otherwise}\,. \end{cases}$$

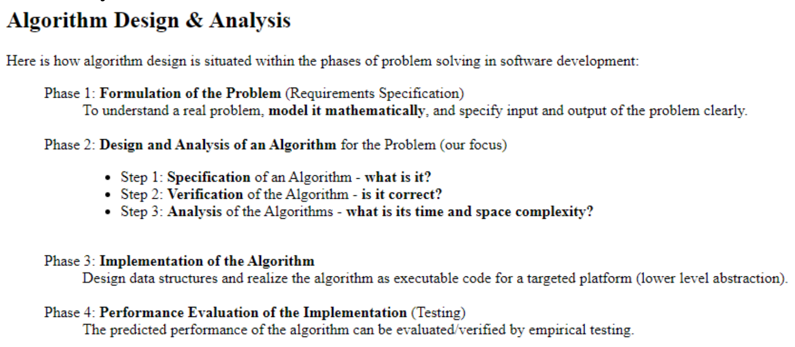
```
void enqueue(Object o)
    Q[tail] = o
    tail = (tail + 1) mod length // mod is % in Java

Object dequeue( )
    o = Q[head]
    Q[head] = null               // allow garbage collection!
    head = (head + 1) mod length
    return o
```

Definity of ADT

An ADT is specified by

1. the type(s) of data objects involved
2. a set of operations that can be applied to those objects, and
3. a set of properties (called axioms) that all the objects and operations must satisfy.



Algorithm Design & Analysis

Here is how algorithm design is situated within the phases of problem solving in software development:

Phase 1: **Formulation of the Problem** (Requirements Specification)
To understand a real problem, **model it mathematically**, and specify input and output of the problem clearly.

Phase 2: **Design and Analysis of an Algorithm** for the Problem (our focus)

- Step 1: **Specification** of an Algorithm - **what is it?**
- Step 2: **Verification** of the Algorithm - **is it correct?**
- Step 3: **Analysis** of the Algorithms - **what is its time and space complexity?**

Phase 3: **Implementation of the Algorithm**
Design data structures and realize the algorithm as executable code for a targeted platform (lower level abstraction).

Phase 4: **Performance Evaluation of the Implementation** (Testing)
The predicted performance of the algorithm can be evaluated/verified by empirical testing.