Dynamic Set

* Mathematical sets are unchanging
* Sets used in computers can grow or shrink i.e. they are DYNMIC

Elements {

Key,

SatelliteData

}

Key data

The key is needed for the operation of the tree. Everything else isn't

Satellite data

Refers to any "payload" data which you want to store in your data structure and which is not part of the structure of the data structure. It can be anything you want. It can be a single value, a large collection of values, or a pointer to some other location that holds the value

Example

Node {

Node next; // key

int value; // satellite data

}

Queries operations

* Search(S, K)
* Minimum(S)
* Maximum(S)
* Successor(S, x)
* Predecessor(S, x)

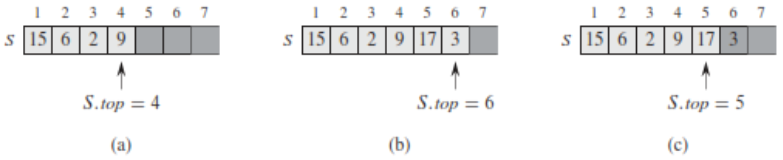
In which, S is a Dynamic Set, K is the key and x is satellite data

Modifying operations

* Insert(S, x)
* Delete(S, x)

Stack

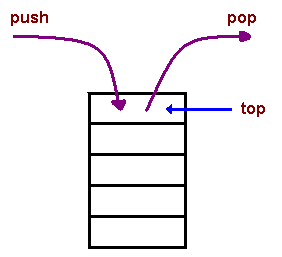
LIFO - Last In First Out



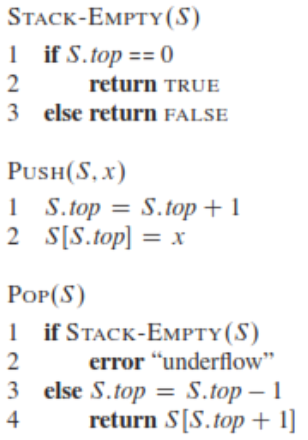
a) S has 4 elements and the top element is 9

b) After called PUSH(S, 17) and PUSH(S, 3)

c) After called POP(S)



Pseudo code



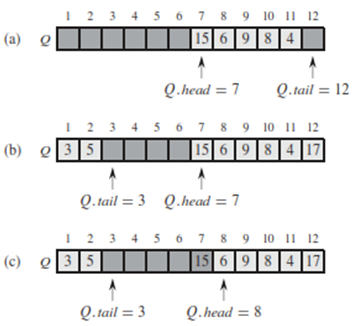
See more

lec8.ArrayStack.java

https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html

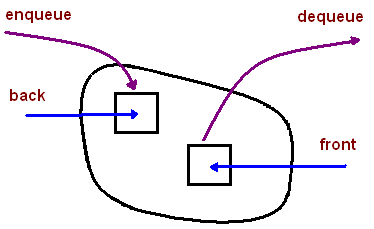
Queue

FIFO – First In First Out

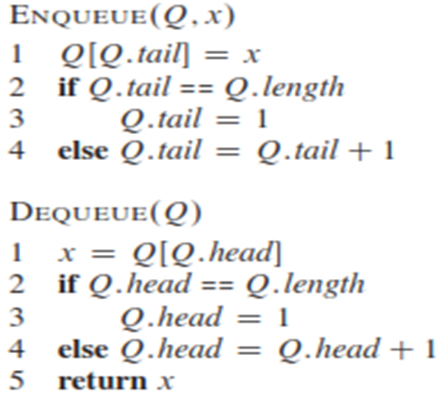


Q [1…12]

1. Queue has 5 elements from 7…11
2. Call enqueuer(17), enqueuer(3) and enqueuer(5)
3. Calls dequeue(Q), returns the value 15 at the head of the queue. New head has value 6



Pseudo code



See more

lec8.ArrayQueue.java

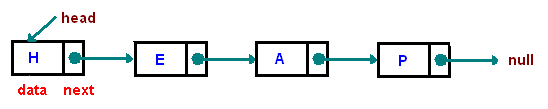
<https://www.cs.cmu.edu/~adamchik/15-121/lectures/Stacks%20and%20Queues/Stacks%20and%20Queues.html>

Linked List

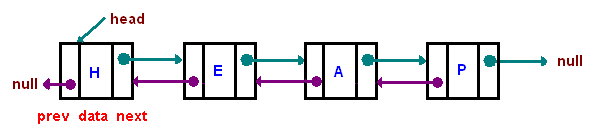
One disadvantage of using arrays to store data is that arrays are static structures and therefore cannot be easily extended or reduced to fit the data set. Arrays are also expensive to maintain new insertions and deletions.

One disadvantage of a linked list against an array is that it does not allow direct access to the individual elements

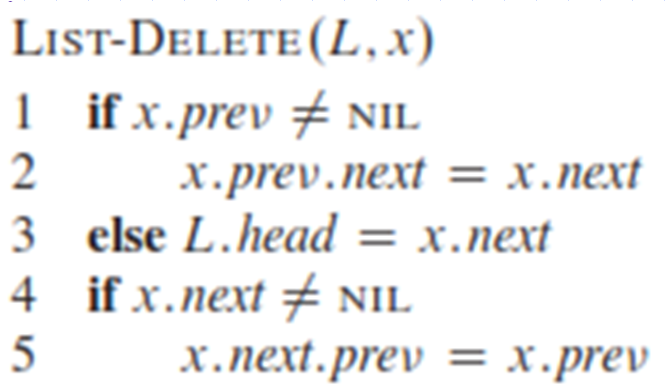
Singly Linked List



Doubly Linked List

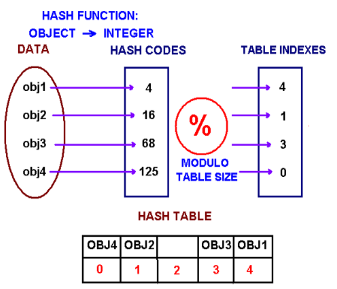


Pseudo code



Hashing

The problem at hands is to speed up searching if we know in advance the index at which that value is located in the array. With this magic function our search is reduced to just one probe, giving us a constant runtime O(1). Such a function is called a hash function. A hash function is a function which when given a key, generates an address in the table.



In the above image, create an array of size M *(= 5 in this case)*. Choose a hash function h that is a mapping from objects into integers 0, 1...M-1. Put these objects into an array at indexes computed via the hash function index = h(object). Such array is called a hash table.

Division method

h(k) = k mod m

Example:

Insert the following sequence of numbers 23, 46, 12, 21, 75, 5, 3 into a hash table of size 9 using h(x) = x%9 as a hash function, where % mean “mod”. Use Chaining with Linked List to avoid collision.

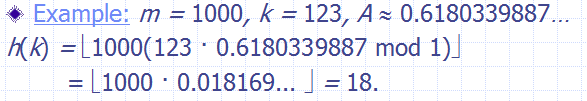
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|  | 46 |  | 12 |  | 23 |  |  |  |
|  |  |  | 21 |  | 5 |  |  |  |
|  |  |  | 75 |  |  |  |  |  |
|  |  |  | 3 |  |  |  |  |  |

Multiplication method



* Disadvantage: Slower than the division method.
* Advantage: Value of m is not critical. Typically chosen as a power of 2, i.e., m = 2p, which makes implementation easy.

Example:



A hash function that returns a unique hash number is called a universal hash function which properties as below:

* It always returns a number for an object.
* Two equal objects will always have the same number
* Two unequal objects not always have different numbers

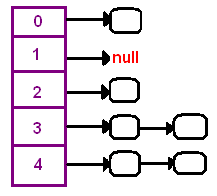
For Division or Multi. Method, the worst case is that all keys map to ONE Slot! To avoid so, Universal Hashing selects hash functions randomly. The key idea is to select Hash Function at random at run time from set of selected functions, provides good average case performance

Collisions

When we put objects into a hash table, it is possible that different objects (by the equals() method) might have the same hash code. This is called a collision. Here is the example of collision. Two different strings ""Aa" and "BB" have the same key:

"Aa" = 'A' \* 31 + 'a' = 2112

"BB" = 'B' \* 31 + 'B' = 2112



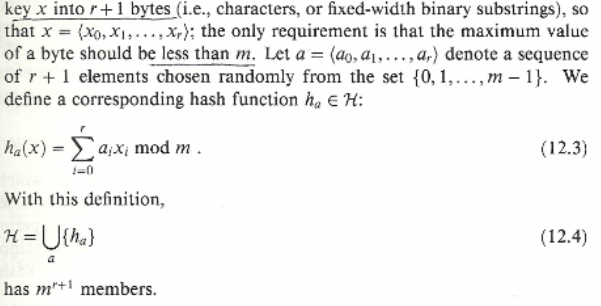
If the number of hash functions h, for which h(x) = h(y) is |H| / m , then h is a Universal Hash Function. Chance of collision between x and y is 1/m.

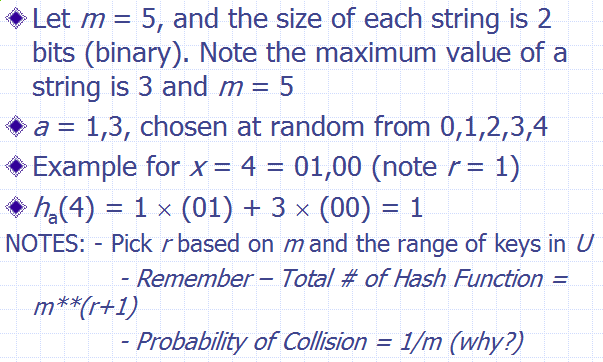
There are several approaches in dealing with collisions. One of them is based on idea of putting the keys that collide in a linked list! A hash table then is an array of lists!! This technique is called a separate chaining collision resolution.

The big attraction of using a hash table is a constant-time performance for the basic operations add, remove, contains, size.

Another technique of collision resolution is a linear probing. If we cannot insert at index k, we try the next slot k+1. If that one is occupied, we go to k+2, and so on.

Decompose





Probability of Collision = 1/m. Why?

We have:

* m way to take a0
* m way to take a1
* m way to take ar

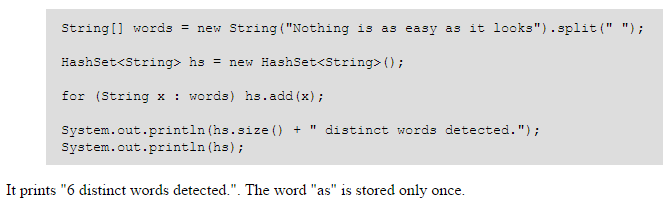
🡪 mr+1

Proof of Universal Hashing

HashSet

HashSet is a regular set - all objects in a set are distinct.

HashSet stores and retrieves elements by their content, which is internally converted into an integer by applying a hash function. Elements from a HashSet are retrieved using an Iterator. The order in which elements are returned depends on their hash codes.

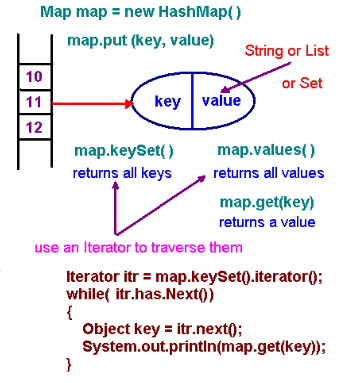


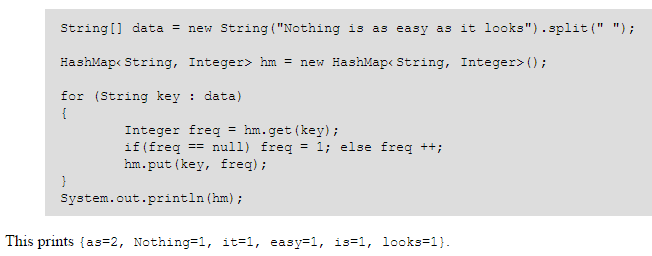
HashSet methods:

* set.add(key) -- adds the key to the set.
* set.contains(key) -- returns true if the set has that key.
* set.iterator() -- returns an iterator over the elements

HashMap

HashMap is a collection class that is designed to store elements as key-value pairs. Maps provide a way of looking up one thing based on the value of another.





HashMap methods

* map.get(key) -- returns the value associated with that key. If the map does not associate any value with that key then it returns null. Referring to "map.get(key)" is similar to referring to "A[key]" for an array A.
* map.put(key,value) -- adds the key-value pair to the map. This is similar to "A[key] = value" for an array A.
* map.containsKey(key) -- returns true if the map has that key.
* map.containsValue(value) -- returns true if the map has that value.
* map.keySet() -- returns a set of all keys
* map.values() -- returns a collection of all value

NOTE: HashSet and HashMap will be printed in no particular order. If the order of insertion is important in your application, you should use LinkeHashSet and/or LinkedHashMap classes. If you want to print dtata in sorted order, you should use TreeSet and or TreeMap classes

Binary Search Tree

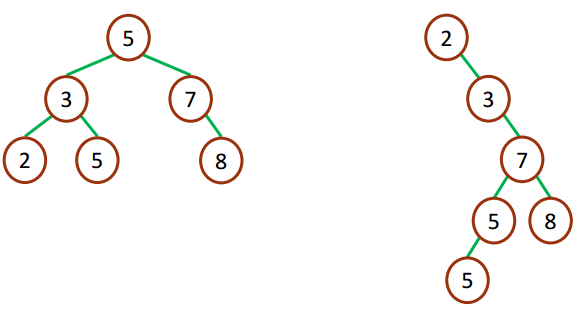
A binary tree is a tree such that each node has *at most* 2 children

BST property

key[leftSubtree(x)] <= key[x] <= key[rightSubtree(x)]

* The key of a node is always greater than the keys of the nodes in its left subtree.
* The key of a node is always smaller than the keys of the nodes in its right subtree.

Example

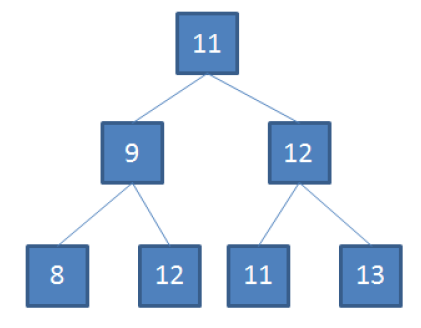


Traversals

* Pre-order tree walk: print root, then left, then right
* In-order tree walk: print left, then parent and then right
* Post-order tree walk: print left, then right, then root
* Level-order tree walk: print the root, and then its children, then its grandchildren

Example

Use Pre-Order, In-Order and Post-Order Tree-Walk to print all keys in the following Tree:



* Pre-order tree walk: 11,9,8,12,12,11,13
* In-order tree walk: 8,9,12,11,11,12,13
* Post-order tree walk: 8,12,9,11,13,12,11
* Level-order tree walk: 11, 9, 12, 8, 12, 11, 13

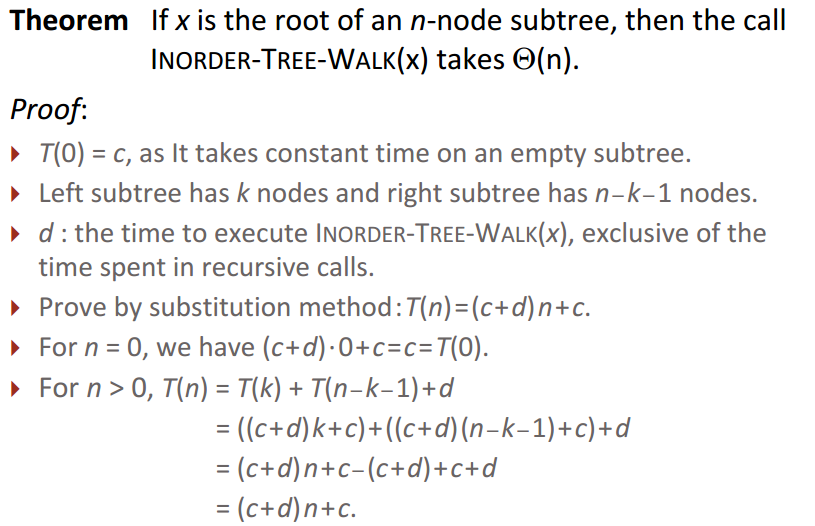


Basic operations take time proportional to the height of the tree, i.e., O(h)

We represent a binary tree by a linked data structure in which each node is an object. Each node contains the fields

* key and possibly other satellite data.
* left: points to left child.
* right: points to right child.
* p: points to parent. p[root[T]] = NIL.

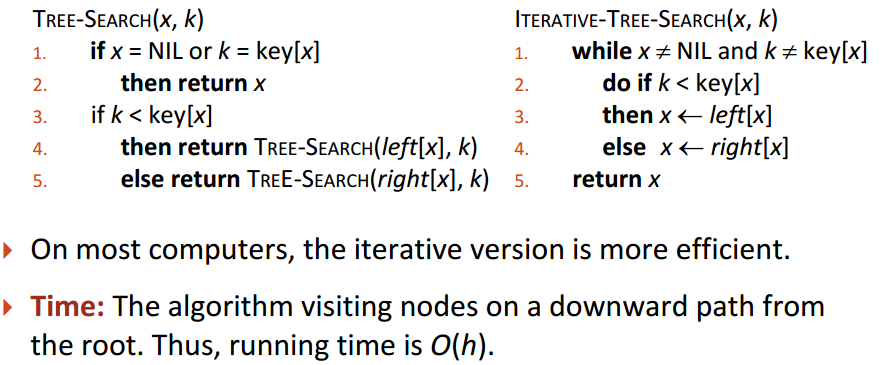
How long will a tree walk take?



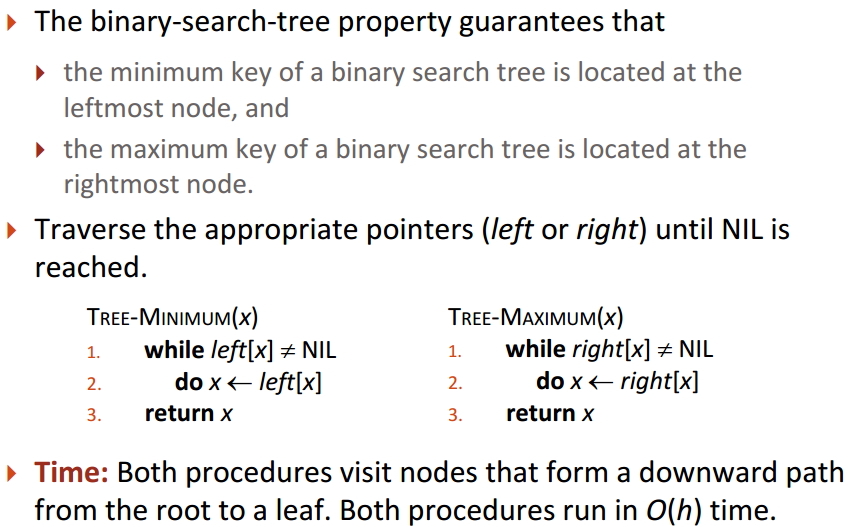
Prove that inorder walk prints in monotonically increasing order



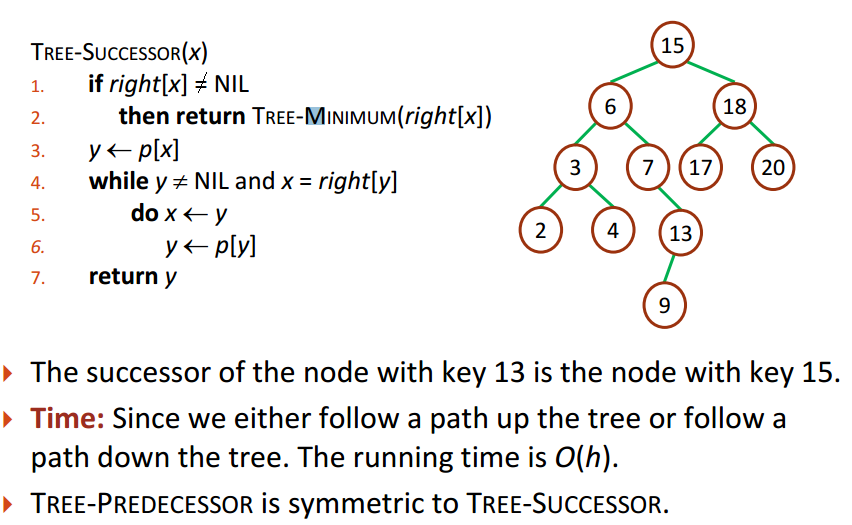
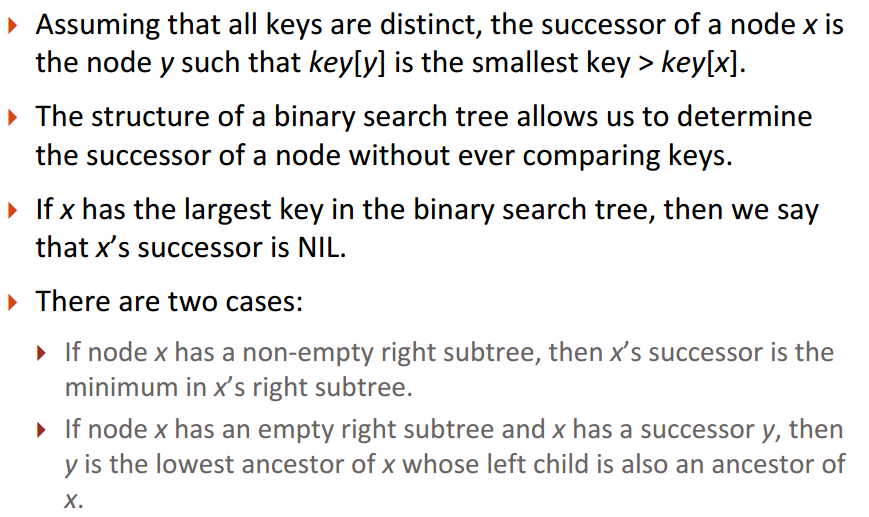
Which of these two functions is more efficient?



Minimum and maximum



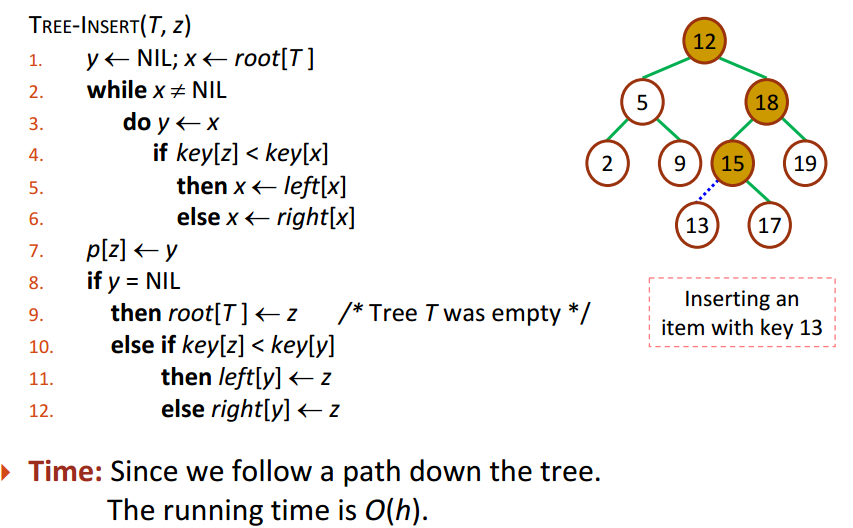
Successor and predecessor



Predecessor:

* + Predecessor is the left subtree's right-most child or
  + The node with largest key that belongs to the tree and that is strictly less than x’s key. Predecessor of 17 is 15 , predecessor of 6 is 4 and predecessor of 13 is 9

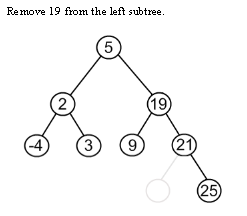
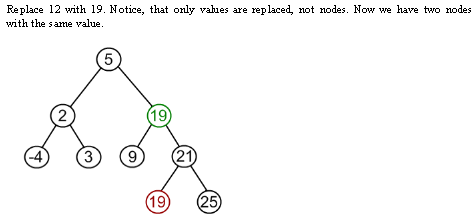
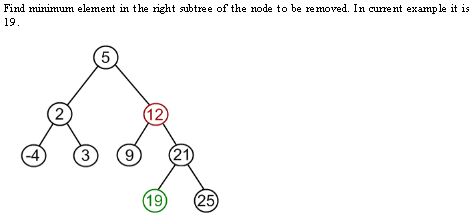
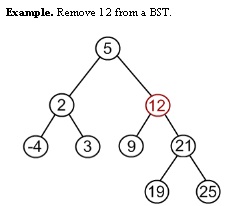
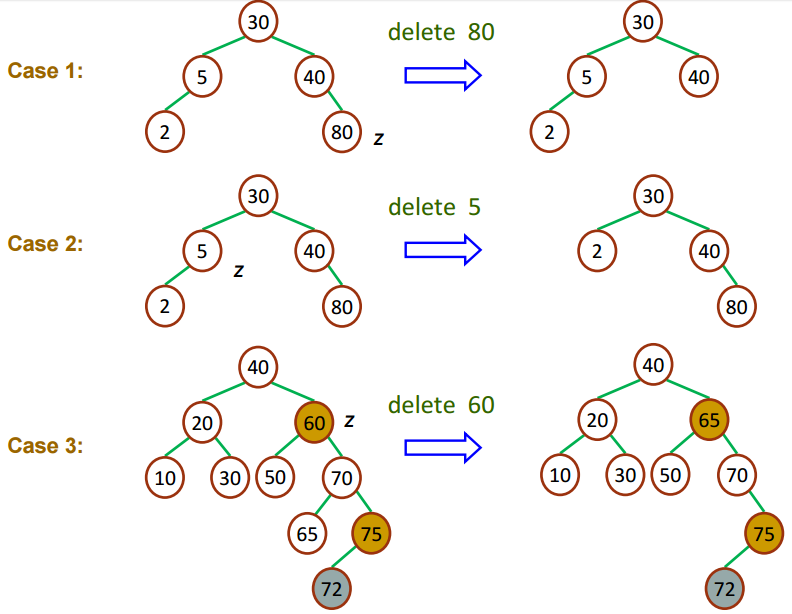
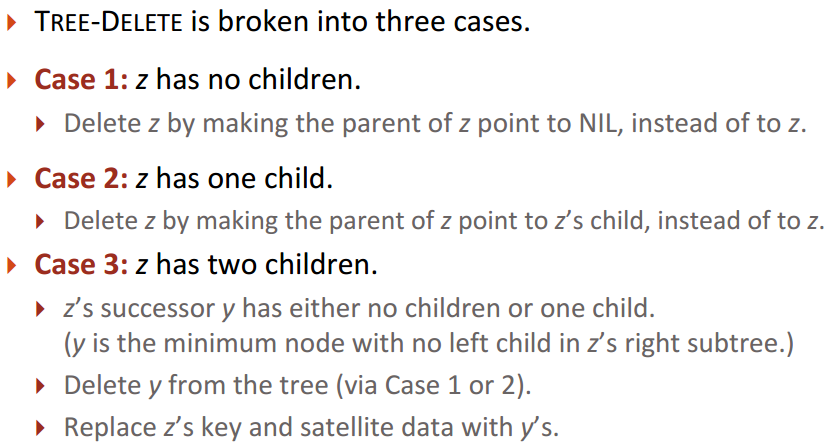
Insertion



Illustrate: insertion of 13

1. 13 > 12 🡪 go to sub-right
2. 13 < 18 🡪 to to sub-left
3. 13 < 15 🡪 to to sub-left
4. sub-left == NIL 🡪 assign 13 to the left of 15

Deletion

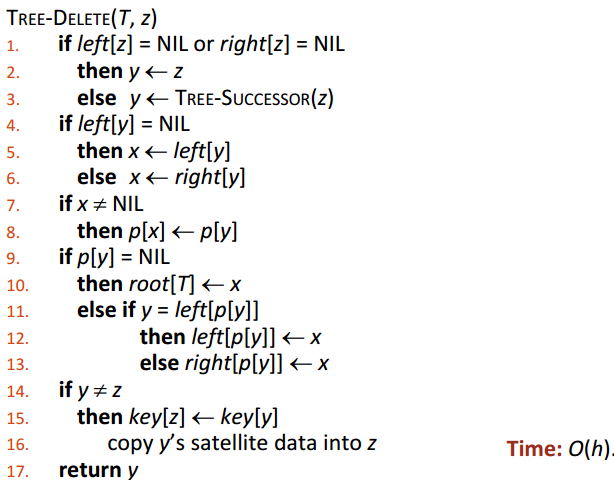


Why will case 2 always go to case 0 or case 1?

A: Because when x has 2 children, its successor is the minimum in its right subtree, and that successor has no left child (hence 0 or 1 child)

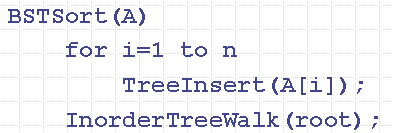
Could we swap x with predecessor instead of successor?

A: Equivalently, we could swap with predecessor instead of successor. It might be good to alternate to avoid creating lopsided tree

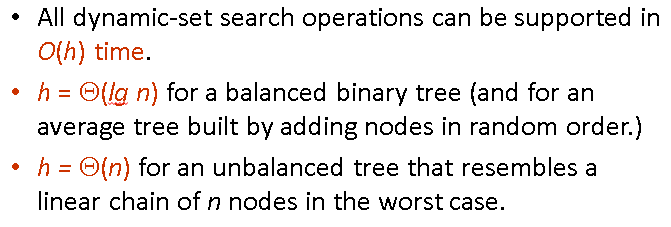
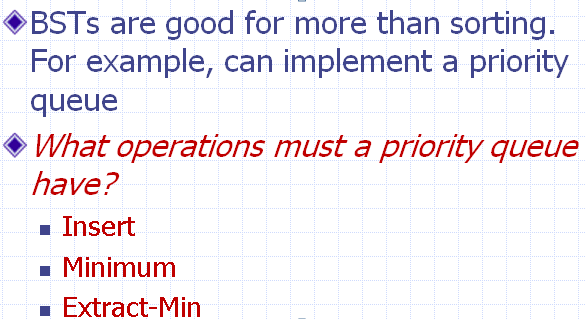
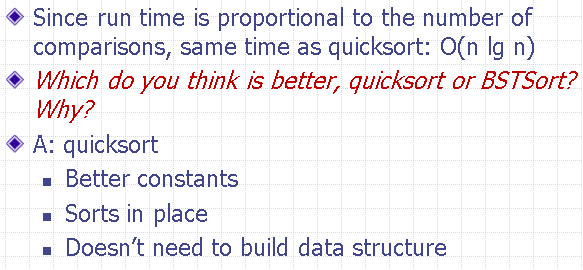


Sorting BST

Pseudo code



Run time



Red Black Tree

The red-black properties

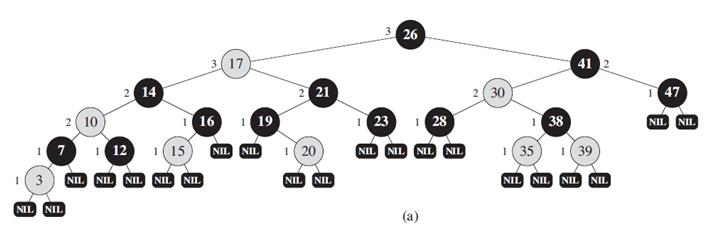
* Every node is either red or black
* Every leaf (NULL pointer) is black

Note: this means every “real” node has 2 children

* If a node is red, both children are black (the reason why parent of red node is black)

Note: can’t have 2 consecutive reds on a path

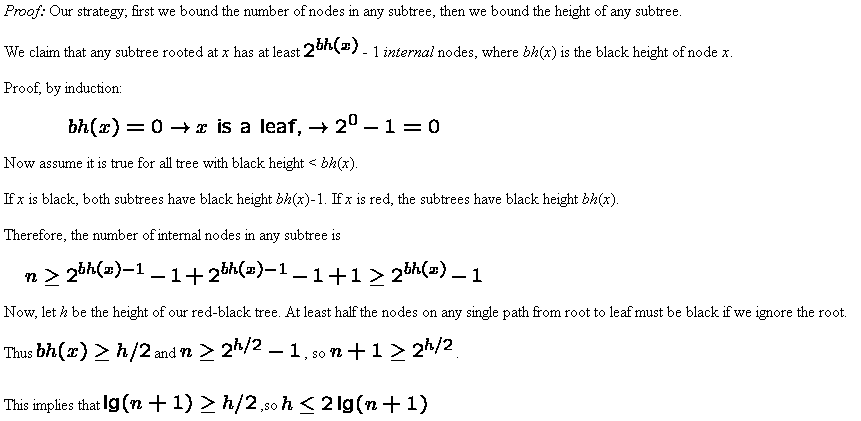
* Every path from node to descendent leaf contains the same number of black nodes (Số lượng node đen trên một đường dẫn từ gốc đến lá được gọi là chiều cao đen (black height). Ta có thể phát biểu quy tắc (4) theo một cách khác là mọi đường dẫn từ gốc đến lá phải có cùng chiều cao đen)
* The root is always black



What is the minimum black-height of a node with height h?

A: a height-h node has black-height >= h/2, means bh(x) >= h/2

Show that a Red-Black tree with n-internal nodes has height at most 2 lg(n+1).



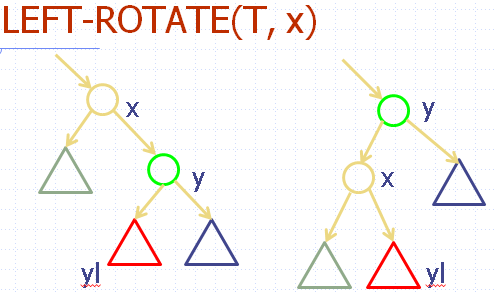
Left Rotate

Let X be the parent and Y be the right child.

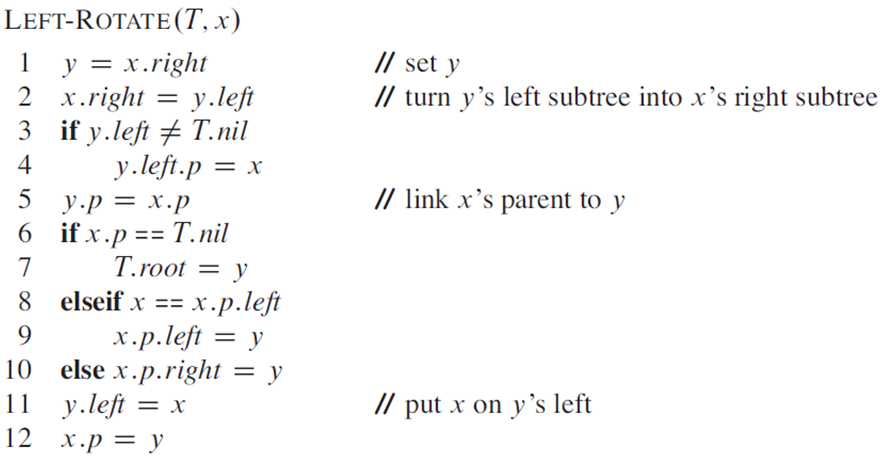
TO DO:

1. Make X Left Child of Y

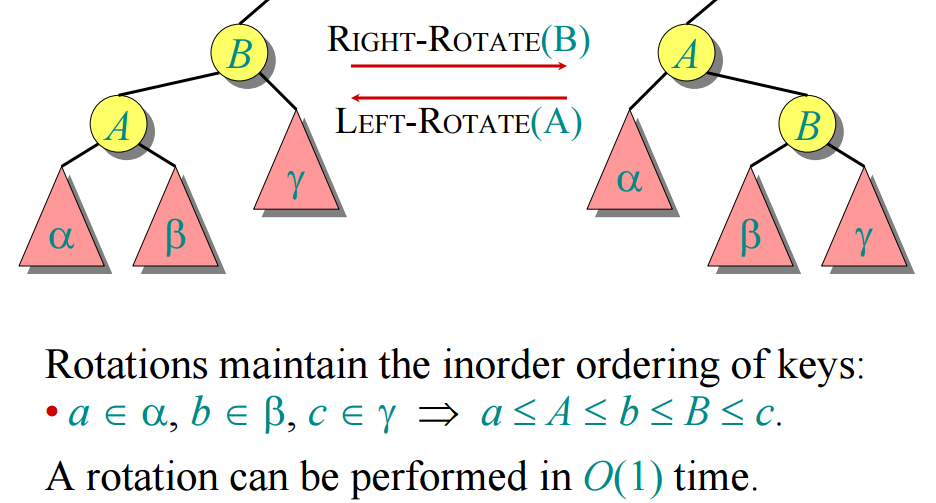
2. Make “previous Left Child of Y, as the Right Child of X



Pseudo code



Right Rotate



Insert New Node

Z is the node that is RED and Z’s Parent is also RED.

What follows is when parent is a left child of the grandparent. Interchange left and right if parent is a right child.

Case 1: Z has a RED uncle.

* + Change Parent, Uncle to BLACK
  + Change Grand Parent to RED
  + Z <- Z.Parent.Parent

Case 2: Z has a no RED uncle and Z is RIGHT child

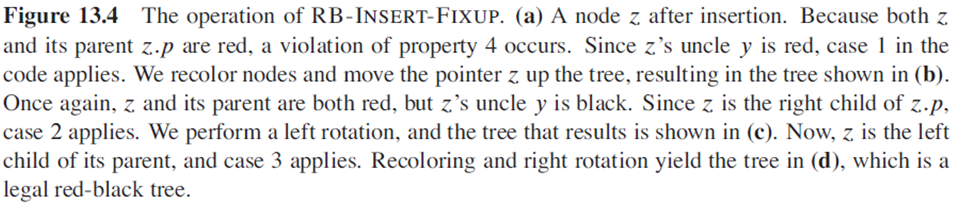
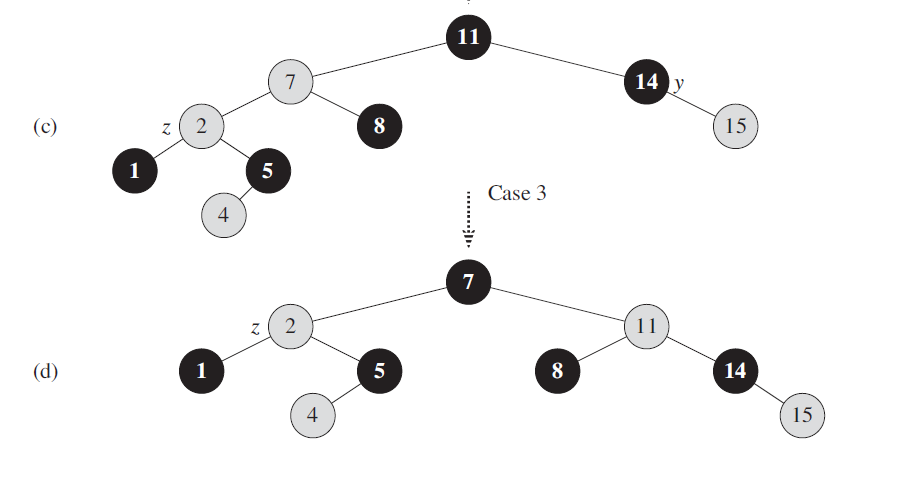
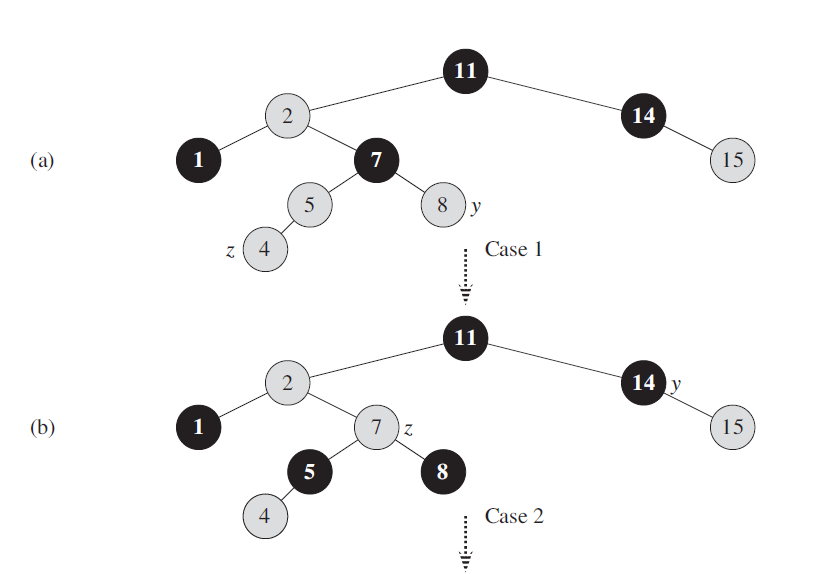
* + Z <- Z.Parent
  + LEFT ROTATION(T, Z)

Case 3: Z has a no RED uncle and Z is LEFT child.

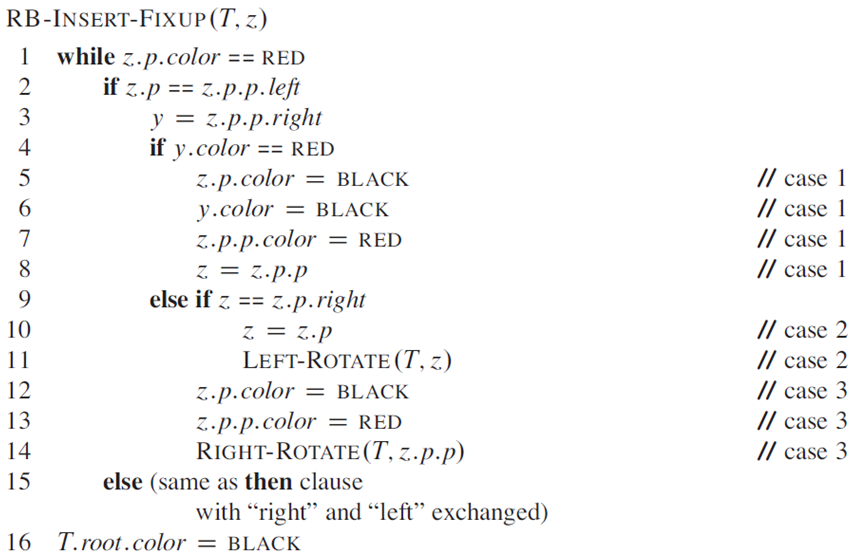
* + Change Color of Parent to BLACK, Grand Parent to RED
  + RIGHT ROTATION(T, Z.Parent.Parent)

At the end, assign BLACK color to Root.

Illustrate



Pseudo code



Red Black Tree – Key comment

The trick is to use Red and Back labels to nodes and use appropriate rules to manipulate nodes in such a way that makes the tree balanced after inserting a new value.

Worst case time

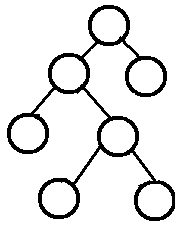
We’ve proved that a red-black tree has O(lg n) height

These operations take O(log n) time: Minimum(), Maximum(), Successor(), Predecessor(), Search().

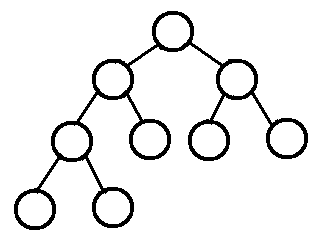
Insert() and Delete(): also take O(log n) time (but will need special care since they modify tree).

Heap Sort

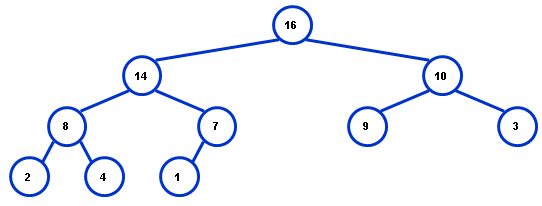
A full binary tree (sometimes proper binary tree or 2-tree or strictly binary tree) is a tree in which every node other than the leaves has two children



A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible



Is it a complete tree?



Yes, nearly complete.

Heap

To represent a complete binary tree as an array

* The root node is A[1]
* Node i is A[i]
* The parent of node i is A[i/2] (note: integer divide)
* The left child of node i is A[2i]
* The right child of node i is A[2i + 1]



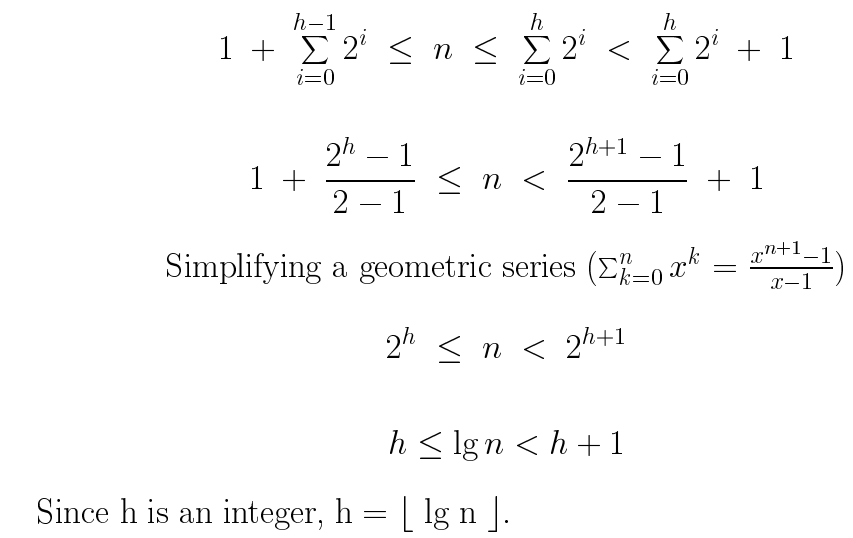
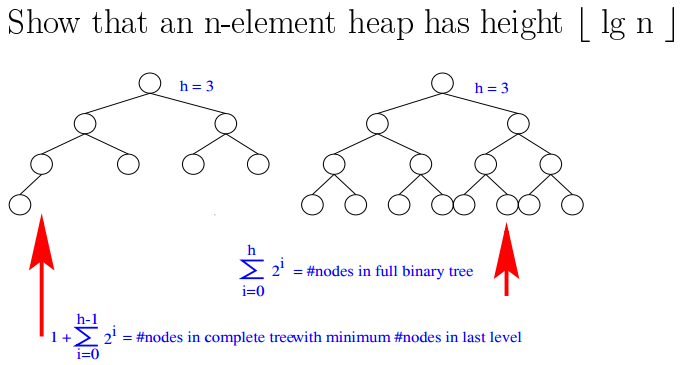
Heap properties

A[Parent(i)] >= A[i] for all nodes i > 1

What is the largest element in a heap stored? A[1] as root, of course.

Heap height

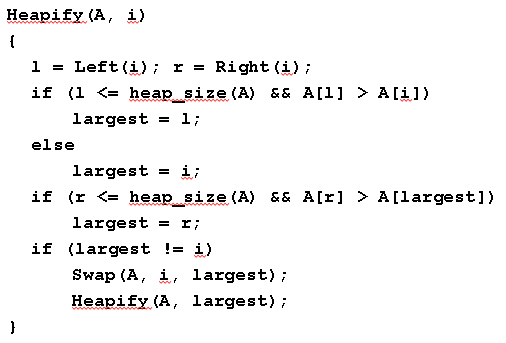
The height of a node in the tree = the number of edges on the longest downward path to a leaf



Heapify() method

Maintain the heap property

* Given: a node i in the heap with children l and r
* Given: two subtrees rooted at l and r, assumed to be heaps
* Problem: The subtree rooted at i may violate the heap property (How?)
* Action: let the value of the parent node “float down” so subtree at i satisfies the heap property



Main idea

From root, find the largest between LEFT and RIGHT, if found then

* Swap i and largest
* Then call Heapify from the largest position

Example: see slides 11 🡪 19 in Heap\_Sort\_EK.ppt file.

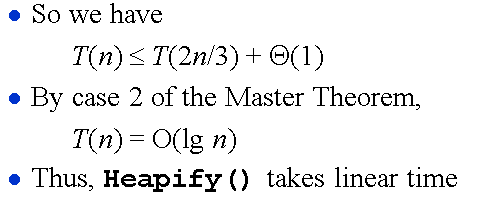
Aside from the recursive call, what is the running time of Heapify()?

Aside from the recursive call, Heapify() fixes up relationships between i, l, and r takes Theta(1) time

If the heap at i has n elements, how many elements can the subtrees at l or r have?

Draw it, the answer is 2n/3

What is the worst-case running time of Heapify() on a heap of size n?



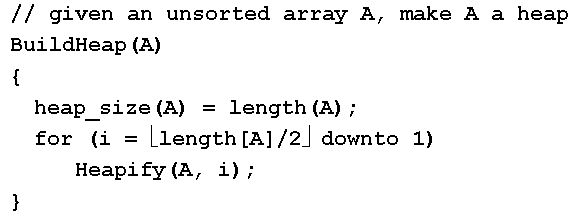
BuildHeap()

Build a heap in a bottom-up manner by running Heapify() on successive subarrays

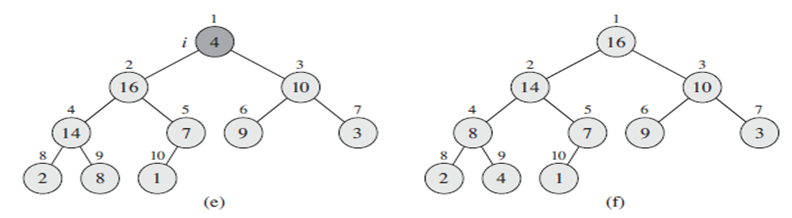
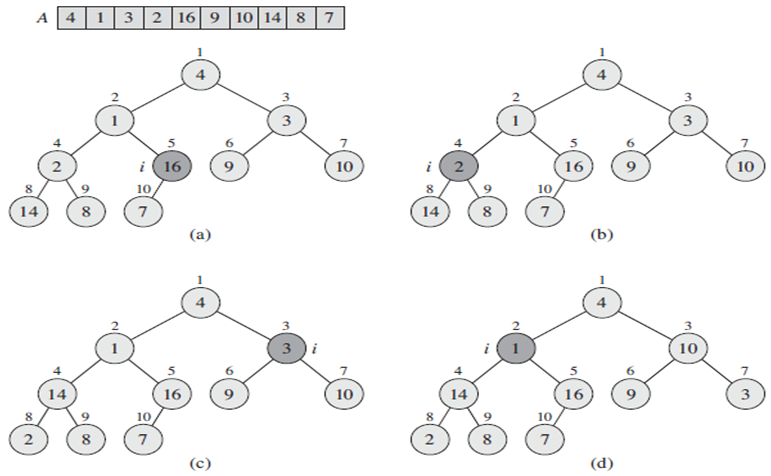
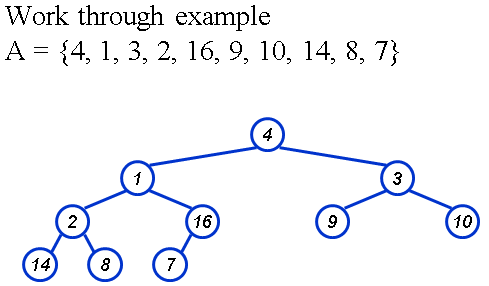
Fact: for array of length n, all elements in range A[((n/2) + 1) .. n] are heaps (Why?)

A: Because the Heap property A[Parent(i)] >= A[i] for all nodes i > 1

Pseudo code



Example

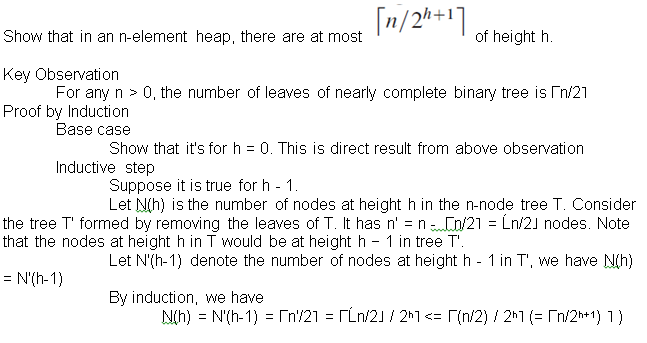


Run time

* Each call to Heapify() takes O(lg n)
* For… loop takes O(n) (actually, it is n/2)

🡪 buildHeap() takes O (n lg n)

Prove that h >= upper bound of (n/2h+1)?

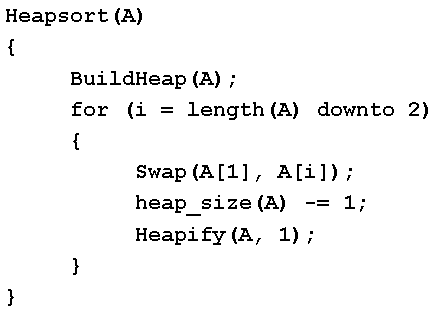


HeapSort()

BuildHeap() – In-place sorting algorithm

* Maximum element is at A[1]
* Discard by swapping with element at A[n]
  + Decrement heap\_size[A]
  + A[n] now contains correct value
* Restore heap property at A[1] by calling Heapify()
* Repeat, always swapping A[1] for A[heap\_size(A)]

Pseudo code

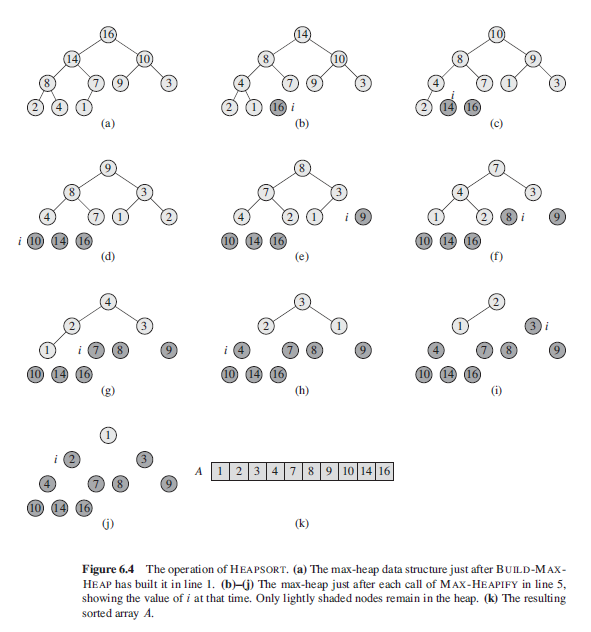


Run time

* BuildHeap() takes O(n) time
* Each of the n - 1 calls to Heapify() takes O(lg n) time

🡪HeapSort() = O(n) + (n-1) O(lg n) = O(n) + O (n lg n) = O (n lg n)

Example



Priority queues

Heapsort is a nice algorithm, but in practice Quicksort usually wins

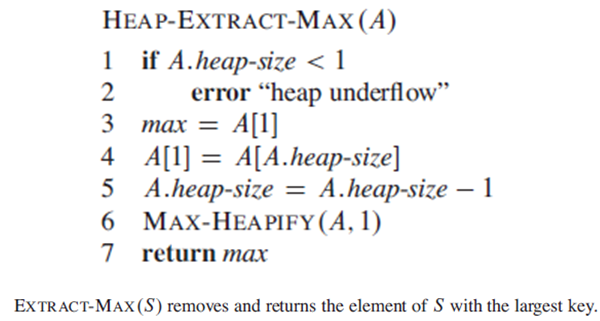
But the heap data structure is incredibly useful for implementing priority queues

* A data structure for maintaining a set S of elements, each with an associated value or key
* Supports the operations Insert(), Maximum(), and ExtractMax()

What might a priority queue be useful for?

Supports the notion that highest priority objects are removed first

Pseudo code



Algorithm Design

1. Devide and Conquer

Involves solving a particular computational problem by dividing it into one or more subproblems of smaller size, recursively solving each subproblem, and then “merging” the solutions to the subproblem(s) to produce a solution to the original problem

The methods

* Divide the problem into subproblems (divide input array into left and right halves)
* Conquer the subproblems by solving them recursively (search recursively in whichever half could potentially contain target element)
* Combine the solutions to the subproblems into a solution to the problem (return value found or indicate not found)

It doesn’t work if

* For Divide and Conquer to be effective, it must be possible to break up the original problem into non-overlapping subproblems (Ex: In MergeSort, the steps of recursive sorting of the left half of the list do not affect, and are not affected by, the steps of the sorting of the right half of the list)
* If something similar to Divide and Conquer is attempted when problems are overlapping, it may result in many redundant computations (Ex: Recursive Fibonacci)
* Binary Search (and operations on a BST)
* MergeSort
* QuickSort
* QuickSelect

1. Dynamic Programming

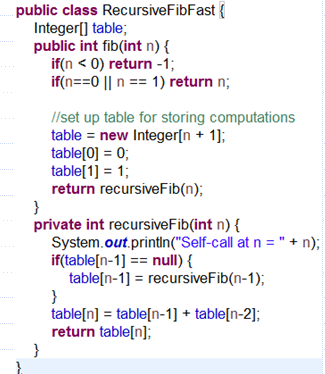
Sometimes problems can be broken down into overlapping subproblems, which can be solved, and whose solutions can be combined in some way to obtain a solution to the main problem. Solutions to subproblems are stored and combined stage by stage to produce a solution to the main problem

When such a problem exhibits the following characteristics, it can in many cases be tackled using dynamic programming:

* Overlapping subproblems – the subproblems “overlap” – the recursion tends to solve the same subproblems over and over (example: recursive fibonacci)
* Optimal substructure – an optimal solution is composed of a combination of optimal solutions to subproblems

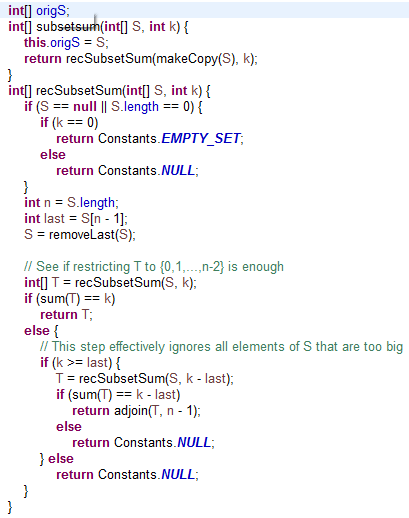
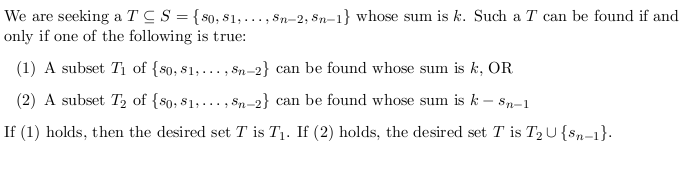
1. Revised Recursive Fibonacci

* To generate the nth Fibonacci number, the subproblems are computation of the kth Fibonacci numbers for k < n.
* To prevent redundant computation, solutions to subproblems can be stored in a table and accessed whenever needed during execution of the algorithm



* SubsetSum

The Subset Sum optimization problem says: We have set S = {s0, s1 … sn-1} of n positive integers and a non-negative integer k. Find a subset T of S so that the sum of the s(r) in T is k



Explanation

The recursive algorithm tries to find a solution T for ({s0, s1, …, sn-2, sn-1}, k) by checking if a solution exists for either of the subproblems

* ({s0, s1, …, sn-2}, k)
* ({s0, s1, …, sn-2}, k - sn-1)

To find these, it seeks solutions to smaller subproblems

As n gets larger, the recursive solution will repeatedly recalculate solutions for the smaller subproblems (recall how this happened with recursive Fibonacci)

We can speed up the recursive approach by storing solutions to subproblems in a table (memoization). See code Demo

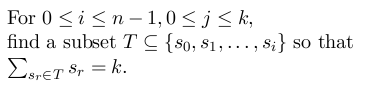
We can organize the stored computations in the recursive algorithm in a table (see RecDynamSubsetSum-Demo.pdf)

Typically, the amount of work done to fill in the table is polynomial bounded, and the rest of the running time is insignificant.

A “bottom-up” approach is typically used to fill in the table from the 0th row to the last row. The correct output is then read from the bottom right corner of the table. See DynamicSubsetSum-Demo.pdf

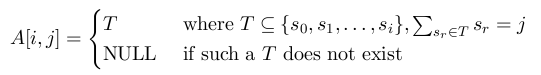
Bottom up approach

There are only (k+1) \* n problems to solve, namely:

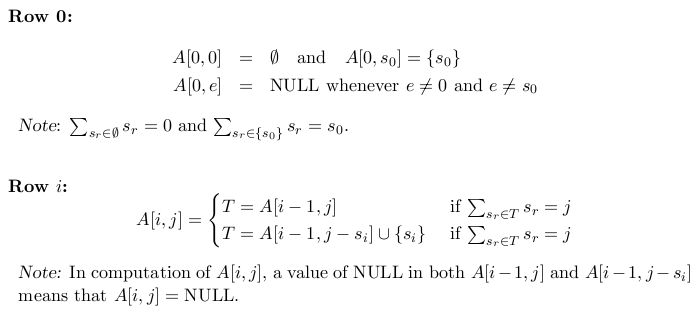


Build a solution for bigger values of i and j using stored solutions for smaller values of i and j.

Obtain a 2-dimensional array (a matrix) A so that



* If S contains values > k, we ignore them since they don’t contribute to the solution (computations for which j is too big are skipped – see the implementation in code)
* Fill row i = 0 first, then fill later rows based on values of earlier rows.

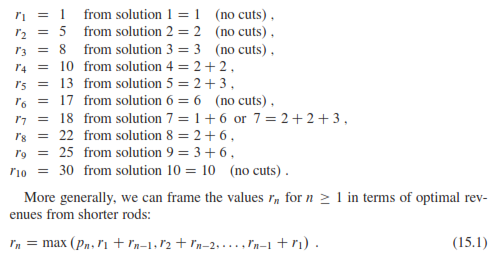


Pseudo-polynomial time algorithm

The dynamic programming solution to SubsetSum runs in O(kn). However, k may be much bigger than n, and even if k is Θ(n), the true running time is based on the number of bits in k, not on the value of k. So even this algorithm runs in exponential time in terms of input size

Note: For a pseudo-polynomial time algorithm, its running time is polynomial in the numeric value of the input, but is exponential in the length of the input – the number of bits required to represent it

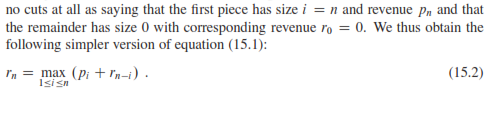
1. Rod Cutting



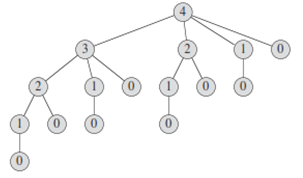
Ex: r(10) = max (p10, r1 + r9, r2 + r8, r3+r7, r4+r6, r5 + r5,…)

= max (30, 26, 27, 26, 27, 26…)

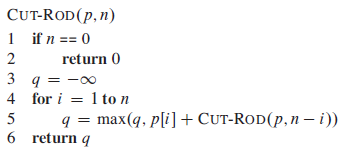
= 30



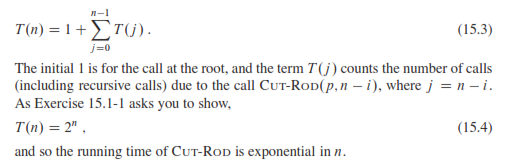
In general, the recursive tree has 2n nodes and 2n-1 leaves. A path from the root to a leaf corresponds to one of the 2n-1 ways of cutting up a rod of length n.



Pseudo code (top-down)



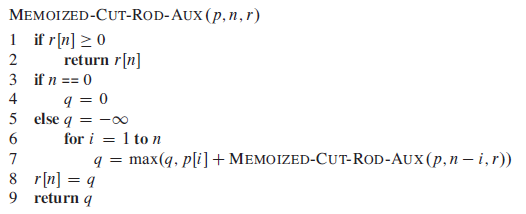
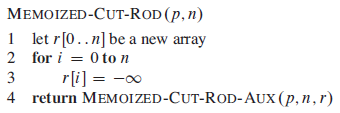
Rod cutting running time



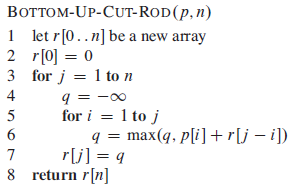
Memoization technique

Memoization comes from memo, since the technique consists of recording a value so that we can look it up later

Memoized top-down approach with Dynamic Programming (DP)



Memoized bottom-up approach with DP (even simplier)



Line 1 of procedure BOTTOM-UP-CUT-ROD creates a new array r[0…n]in which to save the results of the subproblems

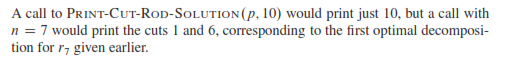
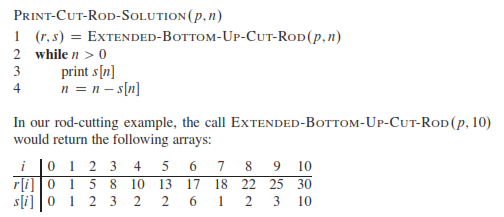
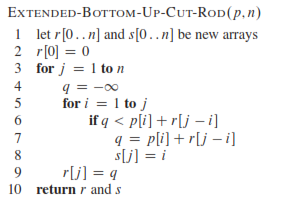
Line 2 initializes r[0] to 0, since a rod of length 0 earns no revenue.

Lines 3–6 solve each subproblem of size j, for j = 1, 2… n, in order of increasing size. The approach used to solve a problem of a particular size j is the same as that used by CUT-ROD, except that line 6 now directly references array entry r[j – i] instead of making a recursive call to solve the subproblem of size j - i.

Line 7 saves in r[j] the solution to the subproblem of size j.

Finally, line 8 returns r[n], which equals the optimal value r(n)

Optimal Memoized bottom up approach



Computing a binomial coefficient by DP

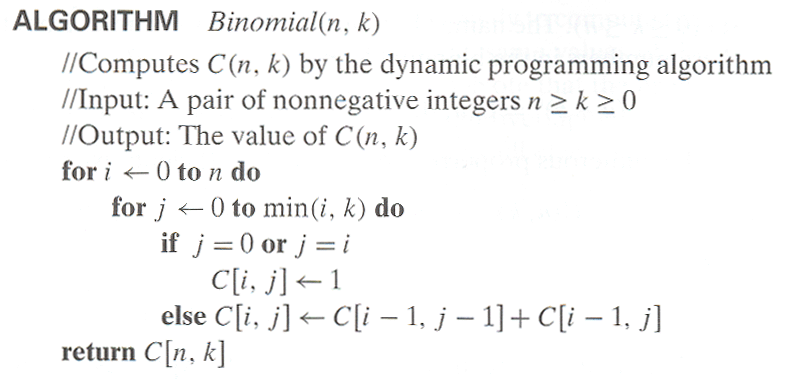
Binomial coefficients are coefficients of the binomial formula:

(a + b)n = C(n,0)anb0 + . . . + C(n,k)an-kbk + . . . + C(n,n)a0bn

Recurrence:

C(n,k) = C(n-1,k) + C(n-1,k-1) for n > k > 0

C(n,0) = 1, C(n,n) = 1 for n >= 0



Runtime complexity: O (nk)

Space complexity: O (nk)

1. Knapsack

Given a knapsack with maximum capacity W, and a set S consisting of n items

Each item i has some weight wi and benefit value b[i] (all w[i], b[i] and W are integer values)

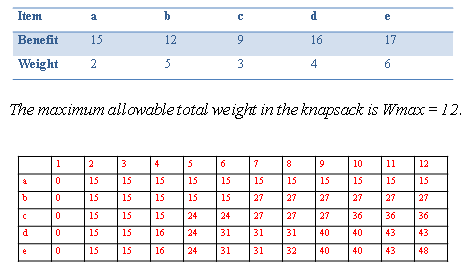
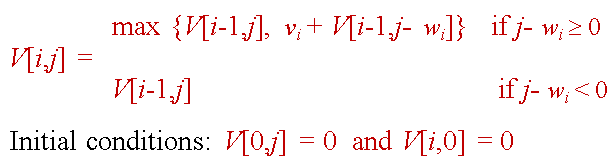
Problem: How to pack the knapsack to achieve maximum total value of packed items?

Given n items of

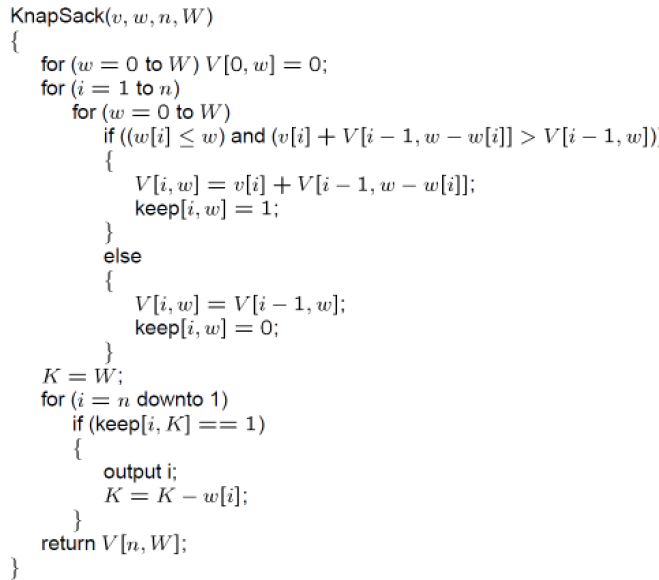
integer weights: w1 w2 … wn

values: v1 v2 … vn

a knapsack of integer capacity W, find most valuable subset of the items that fit into the knapsack.



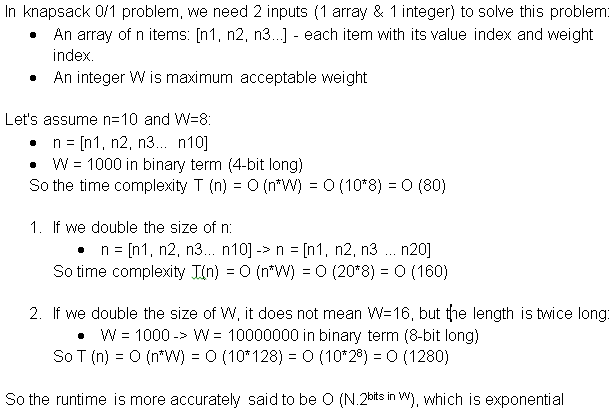
Pseudo code



Running time is O(nW) in terms of values, but, in terms of input size, it’s   
O(n \* 2length(W)), which is potentially exponential in n (whenever n <= W)

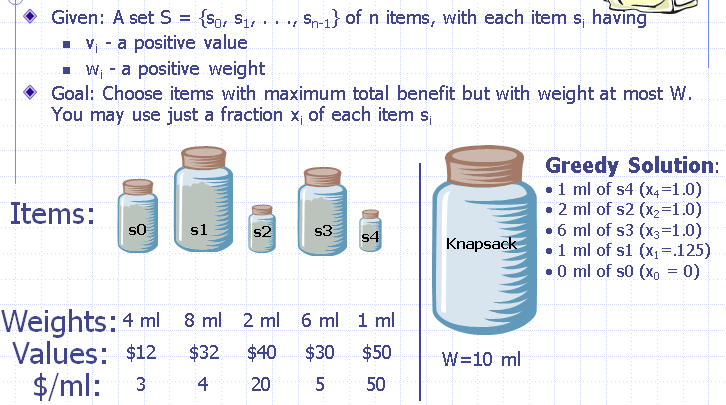
Explain 1: The number of items, n usually is not very large (although it can be theoretically). But the weight W can be very large. To represent this W we would need size = lg b(W) where "b" is the base and "size" is the number of bits to represent "W". This means that W = bsize. If b = 2, we have the running time as O(n x 2size) which is exponential.

Explain 2:

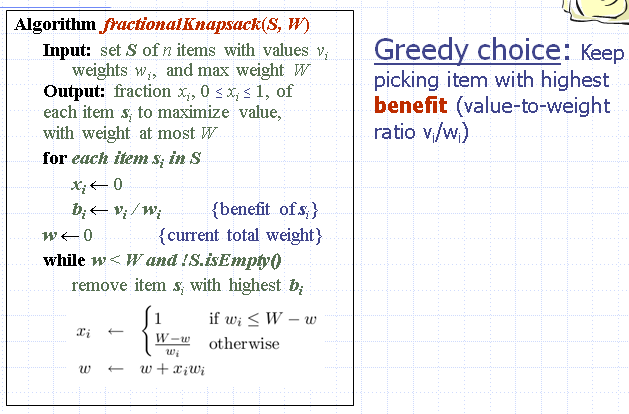


1. Shortest Path (in a graph - later)
2. The Greedy Method
3. Fractional Knapsack

Begin with a max weight W and a set S = {s0, s1…s(n-1)} of n items having weights given in the weights array w[] = {w0, w1...w(n-1)} and values in the vales array v[] = {v0, v1, . . ., v(n-1)}  
  
The objective is to come up with fractions x0, x1, . . ., x(n-1), each in the range [0,1], so that the sum of the numbers x(i)w(i) for s(i) in T is ≤ W and the sum of the numbers x(i)v(i) for s(i) in T is the maximum possible (Note that some of the fractions may equal 0)



Pseudo code



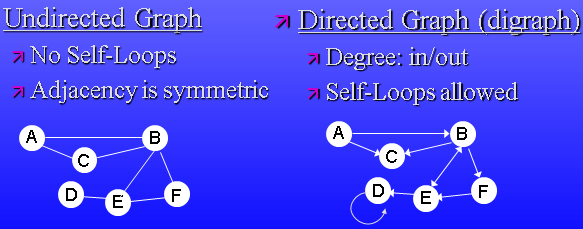
Running time

It takes O(n log n) to sort the benefits and O(n) to scan the sorted list of benefits and perform the needed computations. Therefore, FractionalKnapsack runs in O(n log n)

1. Shortest Path (in a graph - later)
2. Minimum Spanning Tree (in a graph - later)

Graph Concepts

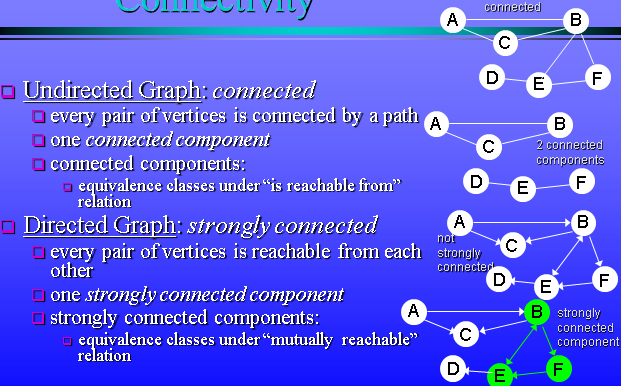
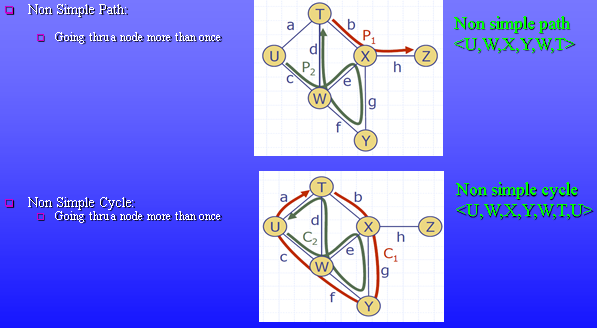
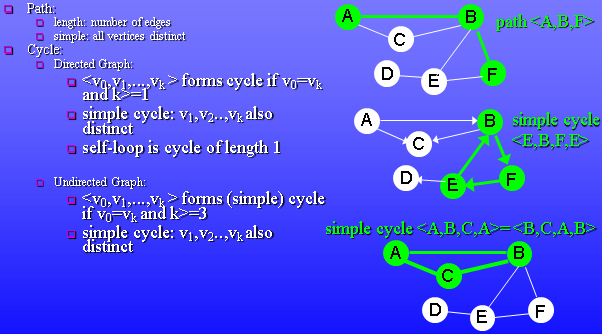
A graph is a pair (V, E), where V is a set of node – called Vertices, E is a collection of pairs of vertices – called Edges.



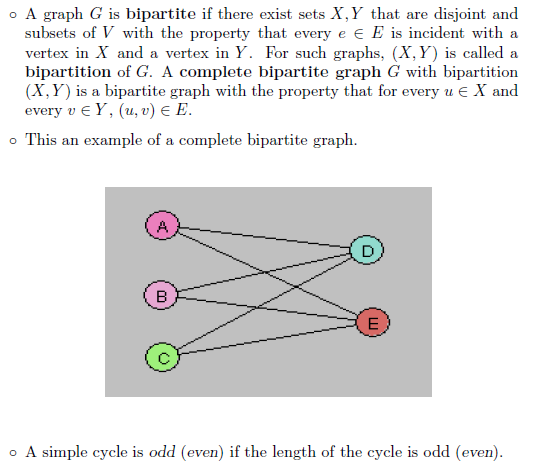
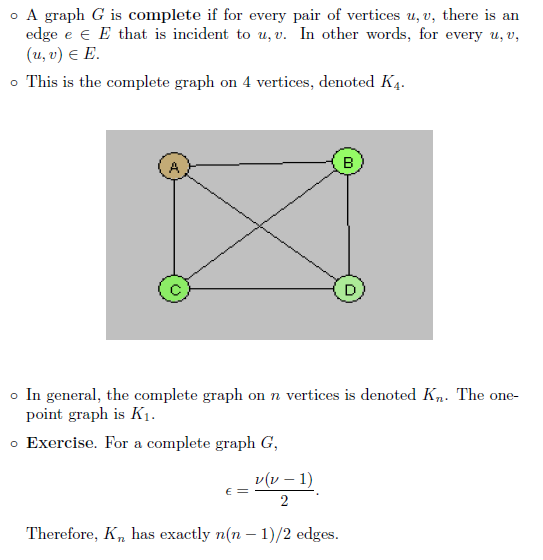
Representation: using List or Matrix



Path & Cycle



Complete graph



Vertex Color Changes

* Vertex is WHITE if it has not yet been encountered during the search.
* Vertex is GRAY if it has been encountered but has not yet been fully explored.
* Vertex is BLACK if it has been fully explored

Edge Classification Legend

When vertex u is being explored, edge e = (u,v) is classified based on the color of v when the edge is first explored

* T: Tree edge if v is WHITE [for DFS and BFS]
* B: Back edge if v is GRAY [for DFS only], for DFS this means v is an ancestor of u in the DFS tree
* F: Forward edge if v is BLACK and [for DFS only] v is a descendent of u in the DFS tree
* C: Cross edge if v is BLACK and [for DFS only] there is no ancestor or descendent relationship between u and v in the DFS tree

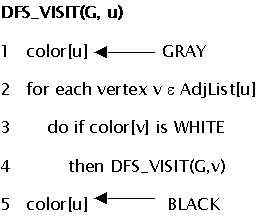
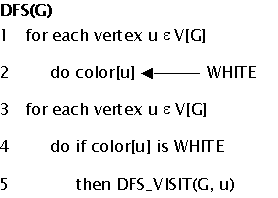
Note that

* For BFS we’ll only consider tree edges.
* For DFS we consider all 4 edge types.
* In DFS of an undirected graph, every edge is either a tree edge or a back edge

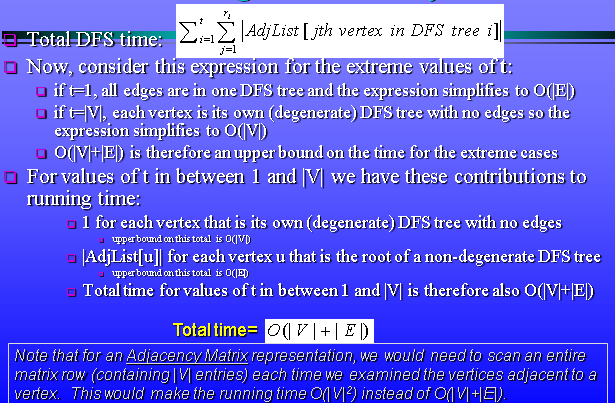
Depth First Search

Pseudo code

We assume an Adjacency List representation



Time complexity

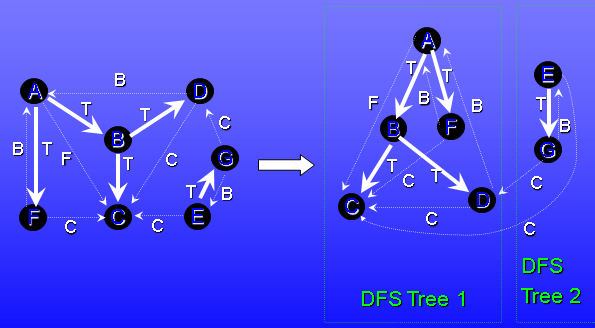


Depth First Search – Directed Graph

Example: See slides 16 🡪 43 in Lec\_12\_Graph\_Algo.ppt file to know more detail.

Blacks node in order: C D B F A (start from A and done at A)

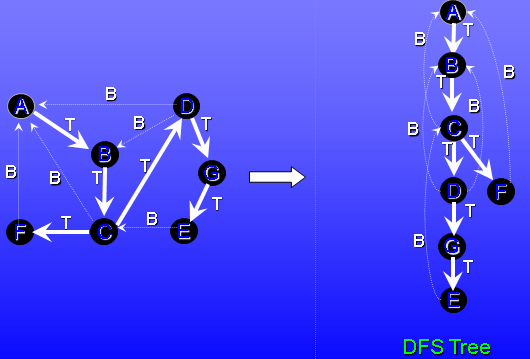
Blacks node in order: G E (start from E and done at E)



DFS – Undirected Graph

Example: See slides 44 🡪 69 in Lec\_12\_Graph\_Algo.ppt file to know more detail.

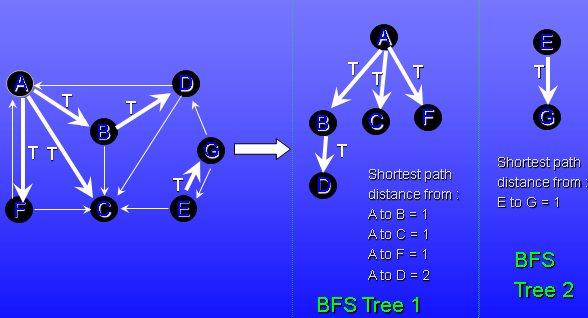
DFS Tree



Breadth First Search – Directed Graph (only tree edges are used)

Example: See slides 75 🡪 88 in Lec\_12\_Graph\_Algo.ppt file to know more detail. Note that Queue is updated whenever vertex is changed color

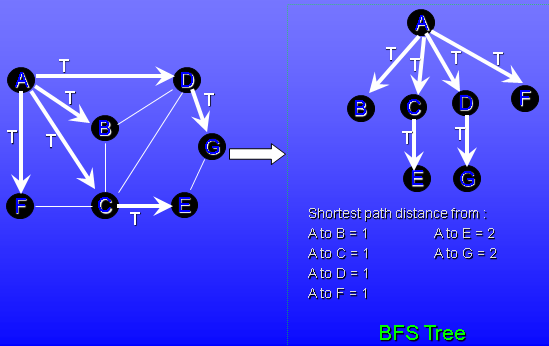
BFS Trees



Breadth First Search – Undirected Graph (only tree edges are used)

Example: See slides 91 🡪 104 in Lec\_12\_Graph\_Algo.ppt file to know more detail. Note that Queue is updated whenever vertex is changed color

BFS Trees



Using DFS & BFS

Using DFS to detect cycles: a directed graph G is acyclic if and only if a DFS of G yields no back edges (Note: DFS can also be used to detect cycles in undirected graph if notion of cycles is defined appropriately)

Using BFS for shortest paths: a BFS of G yields shortest path information: for each BFS tree, the path from its root u to a vertex v yields the shortest path from u to v in G

Elementary Graph Algorithm

For unweighted directed or undirected graph G=(V,E)



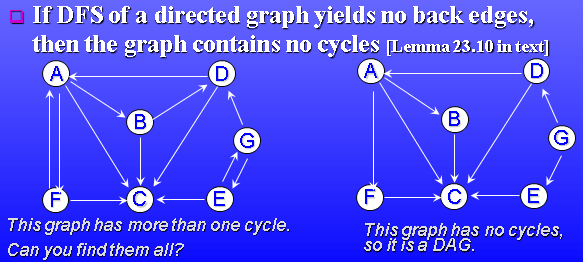
Time complexity

Adjacency list: O(|V| + |E|)

Adjacency matrix: O(|V|2)

Topological Sort

Directed Acyclic Graph – DAG



A topological sort of a DAG G = (V, E) is a linear ordering of all its vertices such that if G contains an edge (u, v), then u appears before v in the ordering.

* If the graph is not acyclic, then no linear ordering is possible.
* A topological sort of a graph can be viewed as an ordering of its vertices along a horizontal line so that all directed edges go from left to right.
* Topological sorting is thus different from the usual kind of "sorting"

Main ideas

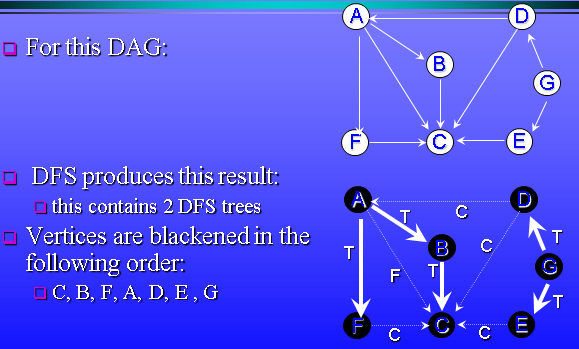
Produces linear ordering of vertices

For edge (u,v), u is ordered before v

TOPOLOGICAL-SORT(G)

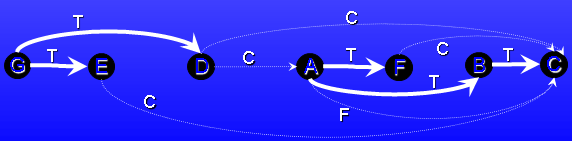
1. DFS(G) computes “finishing times” for each vertex: call DFS(G) to compute finishing times f[v] for each vertex v (this is equal to the order in which vertices change color from gray to black)
2. As each vertex is finished, insert it onto front of list: as each vertex is finished (turns black), insert it onto the front of a linked list
3. Return list: return the linked list of vertices

Example



Final result

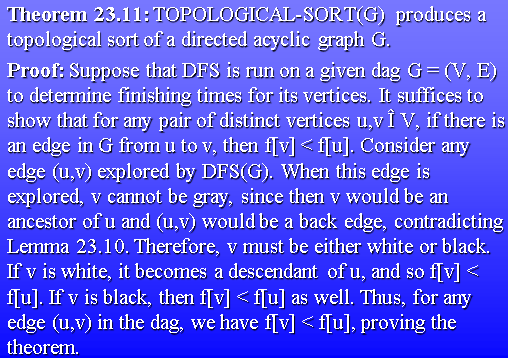
* Vertices are added to front of a linked list in the blackening order.
* Final result is shown below
* Note that all tree edges and non-tree edges point to the right



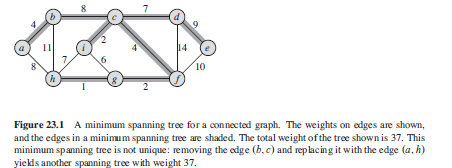
Time Complexity

We can perform a topological sort in time Q(V + E), since depth-first search takes Q(V + E) time and it takes 0(1) time to insert each of the |V| vertices onto the front of the linked list

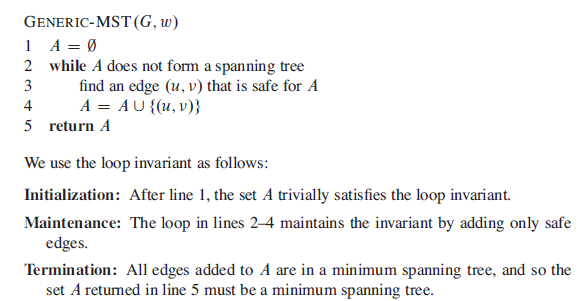
Note



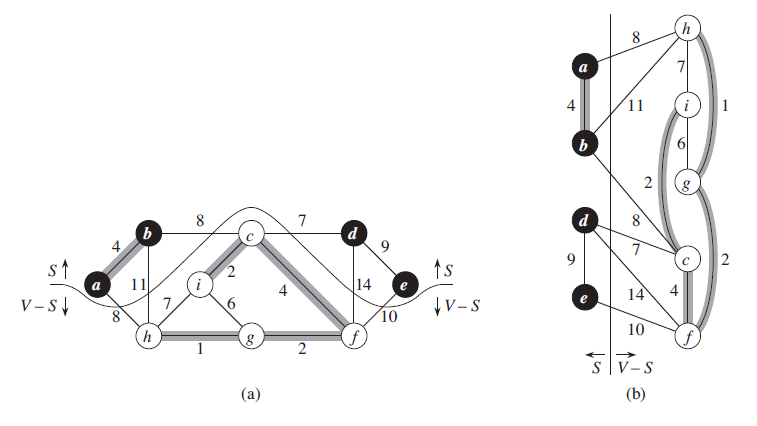
Minimum Spanning Tree



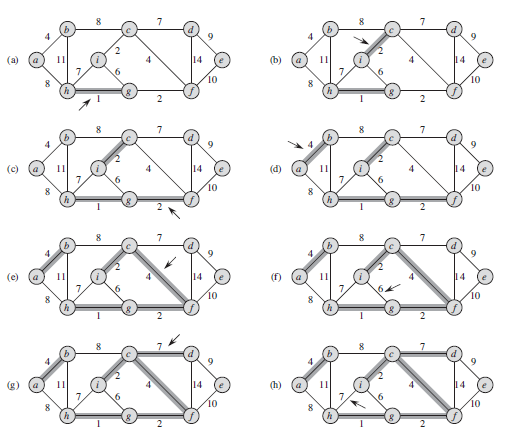
Pseudo code



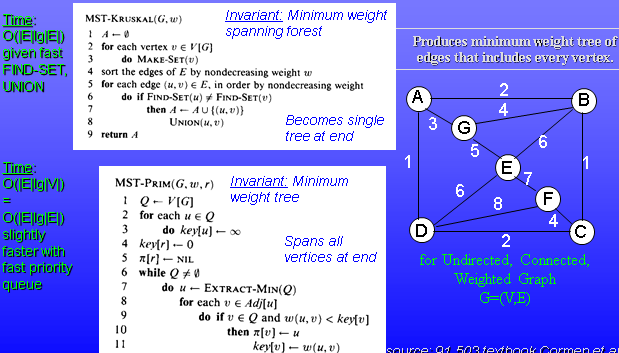
Concept of CUT



Kruskal



Greedy Algorithm



MST-KRUSKAL

* Line 2 – 3: O (V)
* Line 4: E lg E
* Line 5 – 8: O ( E lg E )
  + Line 5: O (E)
  + Line 6: finite O (E, V) = lg E

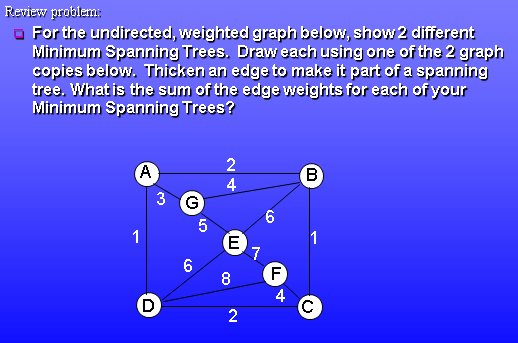
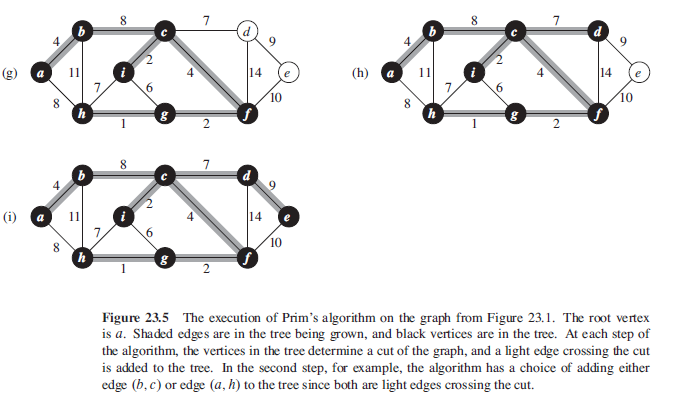
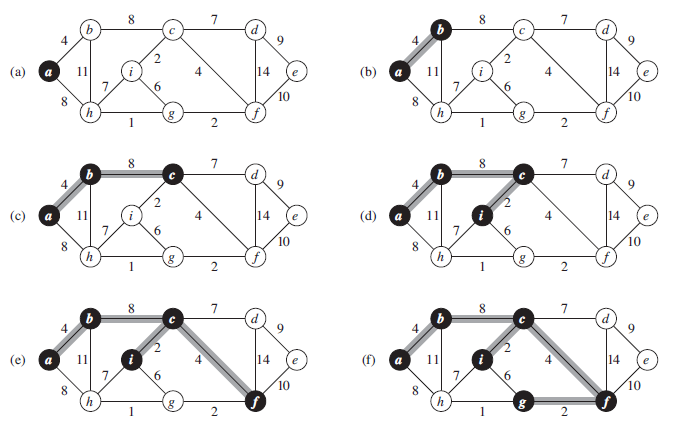
🡪 O ( E lg E ) + O ( E lg E ) = O ( E lg E )

MST-PRIM

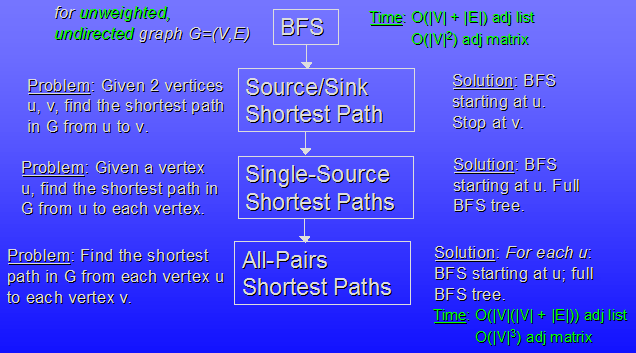
* Line 2 – 3: O (V)
* Line 7: O (V lg V)
* Line 8 – 11: O (E lg V)
  + Line 8: E
  + Line 11: lg V

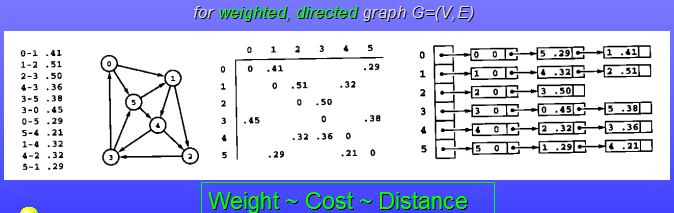
🡪 O ( V lg V + E lg V )

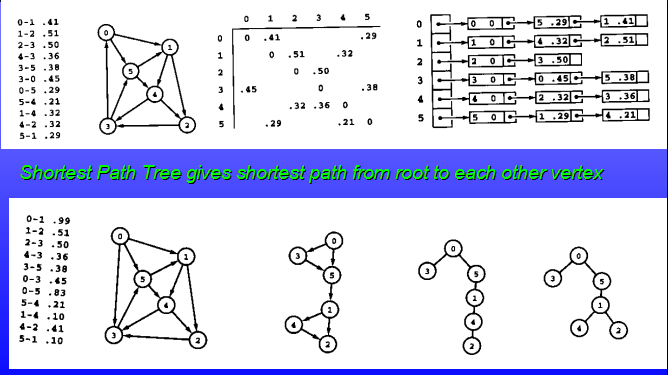
🡪 O ( E lg V ) (when |E| >> |V|)



Shortest Paths

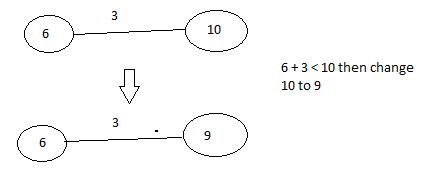






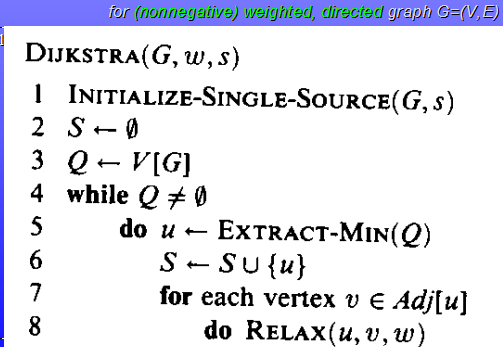
Relaxation of an Edge (u,v)

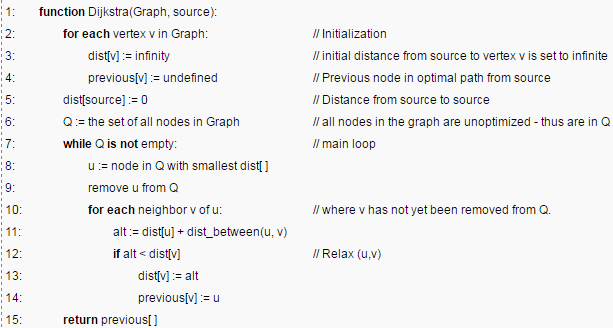
Test if shortest path to v [found so far] can be improved by going through u

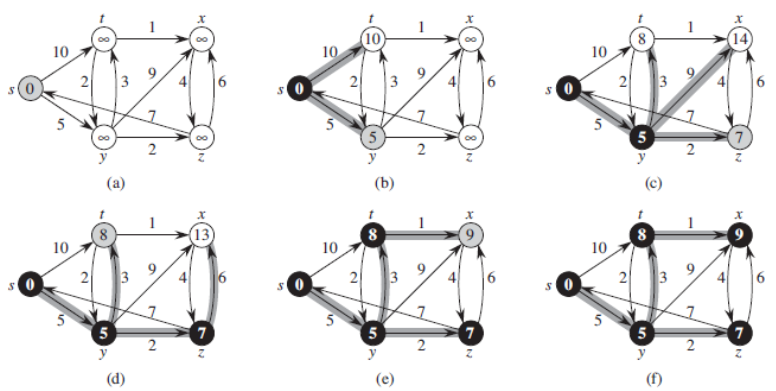


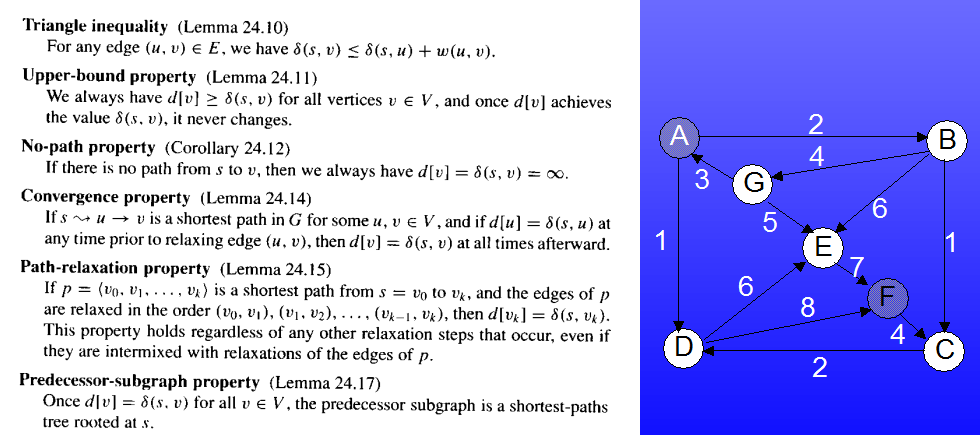
Dijkstra

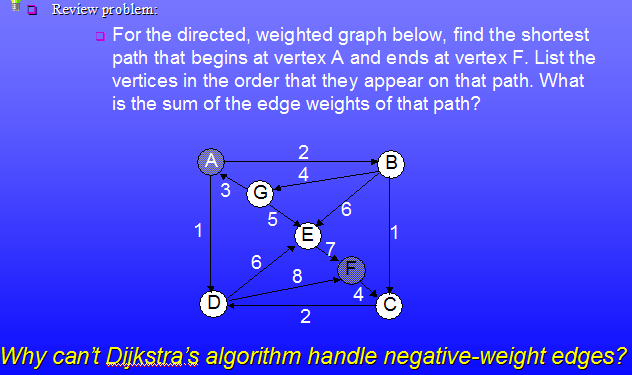
Pseudo code

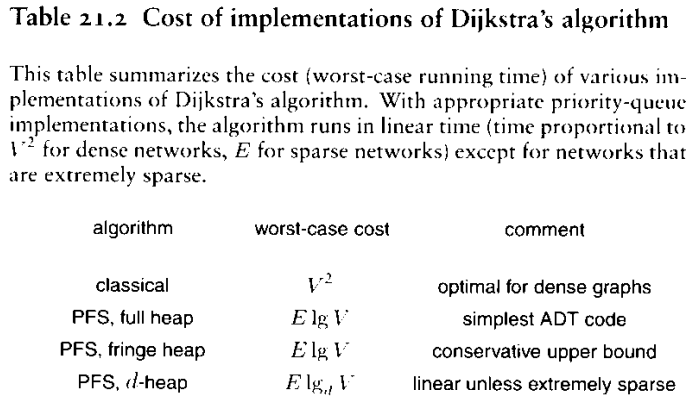


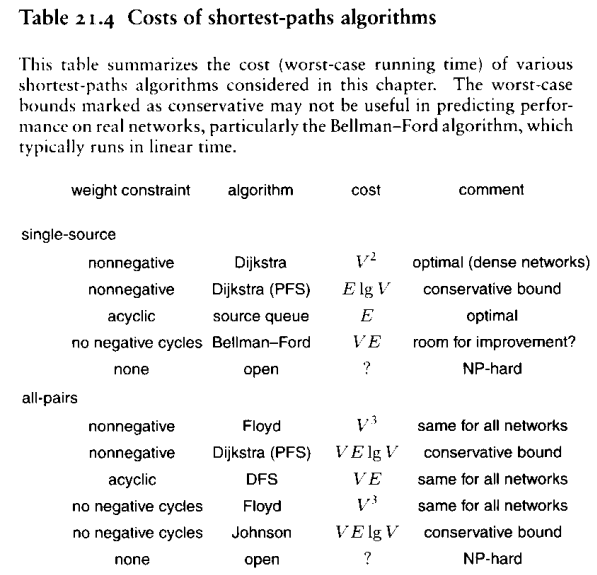




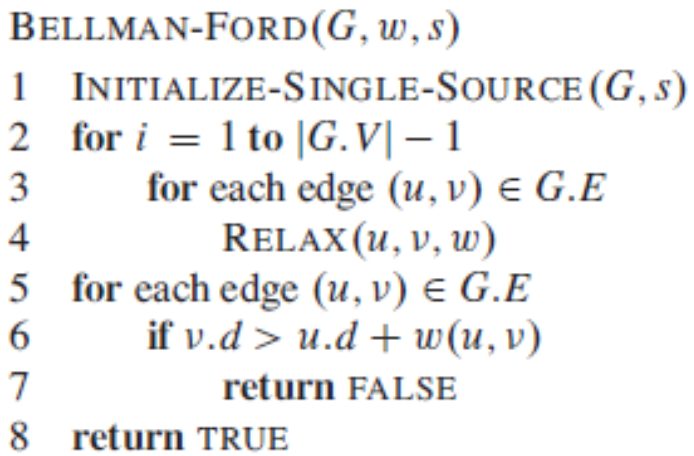


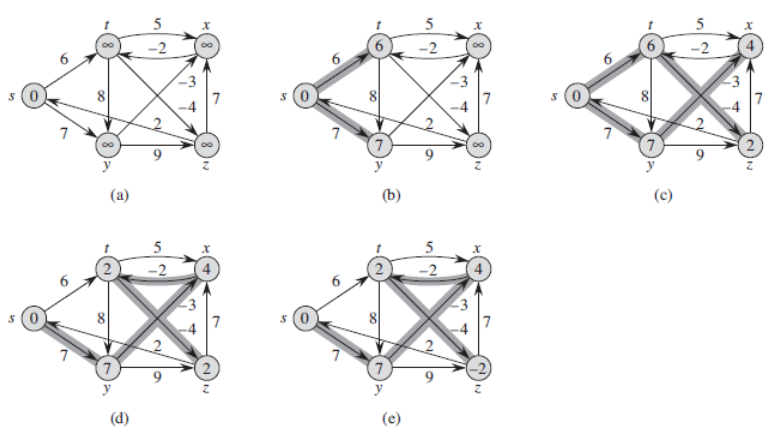






Bellman-ford





Hard Problem

* Decision problems that have no known polynomial time solution are considered hard, but hard problems can be further classified to determine their degree of hardness.
* A decision problem belongs to NP if there is a polynomial p and an algorithm A such that for any instance of the problem of size n, a correct solution to the problem can be verified using A in at most p(n) steps.
* In addition, the problem is said to be NP-complete if it belongs to NP and every NP problem can be polynomial-reduced to it

Are some problems solvable in polynomial time?

Of course: many algorithms we’ve studied provide polynomial-time solutions to some problems

Are all problems solvable in polynomial time?

No: Turing’s “Halting Problem” is not solvable by any computer, no matter how much time is given

Most problems that do not yield polynomial-time algorithms are either optimization or decision problems

How Do We Know When a Problem Does Not Belong to P?

Hard to know for sure because even if there is no known polynomial time algorithm today, tomorrow someone may come up with one

Modern-day example: The IsPrime problem. Before 2002, all known deterministic algorithms to solve this problem ran in superpolynomial time.

EXPTIME-Complete problems, like nxn chess, do require superpolynomial time and therefore do not belong to P

The class NP

* To understand decision problems that may not belong to P, one approach is to see how hard it is to check whether a given solution is correct
* The class of decision problems with the property that a solution can be verified in polynomial time is denoted NP.

Optimization Problems

An optimization problem is one which asks, “What is the optimal solution to problem X?”

Examples:

* Knapsack
* Fractional Knapsack
* Minimum Spanning Tree

Decision Problems

A decision problem is one with yes/no answer

Examples:

* Does a graph G have a MST of weight  W?

Many problems will have decision and optimization versions

Example: Traveling salesman problem

* optimization: find hamiltonian cycle of minimum weight
* decision: is there a hamiltonian cycle of weight <= k

Some problems are decidable, but intractable: as they grow large, we are unable to solve them in reasonable time

Is there a polynomial-time algorithm that solves the problem?

An instance of a Decision Problem

Is said to have a solution or be a solvable instance if “true” is the correct answer to the problem.

Example: “Is there a vertex cover of size at most 3 for the complete graph on 5 vertices?” has a solution (so “true” is the correct answer)

Example: “Is there a vertex cover of size at most 3 for the complete graph on 8 vertices?” does not have a solution (so “true” is not correct)

An instance I of a problem has input data (e.g. (V,E,k) for VertexCover); any candidate solution for an instance also consists of data (e.g. a subset W of V for VertexCover) called solution data.

Need to know the size of input data.

For graphs, the convention is that the number of vertices is the size that is used

NP-complete problems are a set of problems that any other NP-problem can be reduced to in polynomial time, but retain the ability to have their solution verified in polynomial time. Alternatively, a problem (in NP) is NP-complete if any problem in NP is reducible to it

Examples: Traveling Salesman Problem & Hamiltonian Cycle problem

NP-hard problems are those at least as hard as NP-complete problems, meaning all NP-problems can be reduced to them, but not all NP-hard problems are in NP, meaning not all of them have solutions verifiable in polynomial time

Definition of P

A decision problem Q is said to belong to P if there exists a polynomial p(y) (“polynomial witness”) and an algorithm A such that for any instance I of Q that has a solution and that has input data X of size n, when A is run on input X, A outputs “true” in O(p(n)) time.   
We say A and p(y) witness that Q belongs to P.

Definition of NP

A decision problem Q is said to belong to NP if there exists a polynomial p(y) (“polynomial witness”) and an algorithm B such that for any instance I of Q that has a solution, if X is input data of size n and Y is a solution data (called a certificate), then when B is run with input (X, Y), B outputs the message “verified” in O(p(n)) time. (Time includes the effort to read the certificate Y). We say that B and p(y) witness that Q belongs to NP

Yet another definition of NP

NP: the class of decision problems those are solvable in polynomial time on a nondeterministic machine (or with a nondeterministic algorithm)

* + A deterministic computer is what we know
  + A nondeterministic computer is one that can “guess” the right answer or solution

Think of a nondeterministic computer as a parallel machine that can freely spawn an infinite number of processes (to some extent similar to Quantum Computing).

Note that NP stands for “Nondeterministic Polynomial-time”

Quantum Computing and NPC

* Uses Q bit that can super impose inputs in a probabilistic way
* Can store 2n in a probabilistic way in n qbits etc.

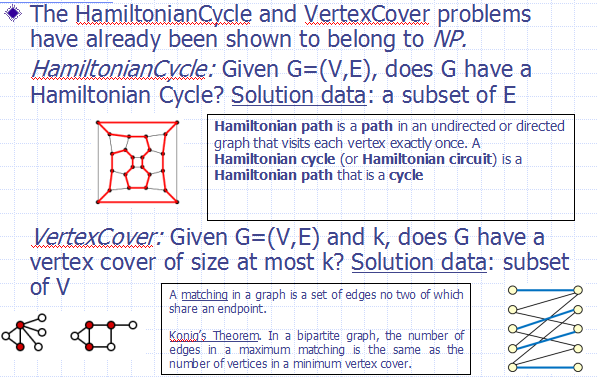
Thus, bits, algorithms, results are all probabilistic. At this time only some classes of problems can be solved fast in a quantum machine.

Quantum computer cannot solve NPC or NP-Hard problems in Polynomial time (in general, may be some specific problems can be solved in P time)

Quantum computing Complexity

* BQP - : bounded error quantum polynomial time is the class of decision problems solvable by a quantum computer in polynomial time, with an error probability of at most 1/3 for all instances.
* QMA - Quantum Merlin Arthur, is the quantum analog of the nonprobabilistic complexity class NP or the probabilistic complexity class MA.
* QMA is related to BQP in the same way NP is related to P, or MA is related to BPP.
* (BPP is bounded-error probabilistic polynomial time)

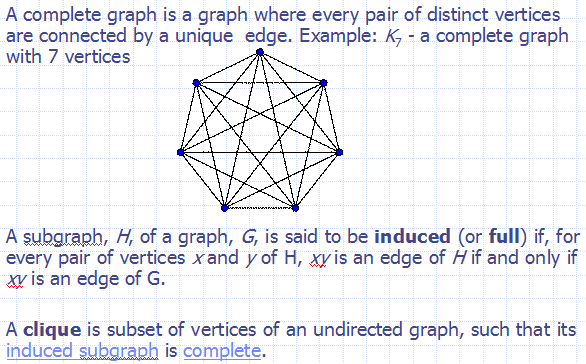
The HamiltonianCycle and VertexCover problems have already been shown to belong to NP.



Traveling Salesman Problem (TSP)



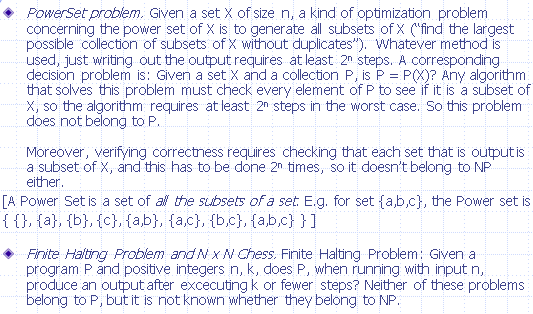
Complete Graph, Induced Graph and Clique



TSP belongs to NP

* Given an instance I of TSP with input data a complete graph G with n vertices, a cost function of c: E 🡪 N and a positive integer k
* Given a certificate E0 (a subset of E) - the algorithm B checks that E0 is a Hamiltonian Cycle for G
* Computes the sum of c(e) over the edges e in E0 to verify that it is at most k.

Example not in NP



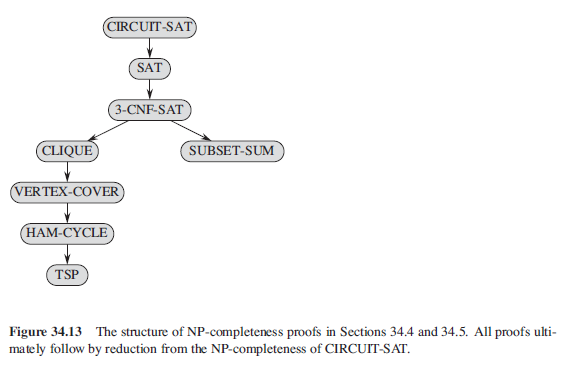
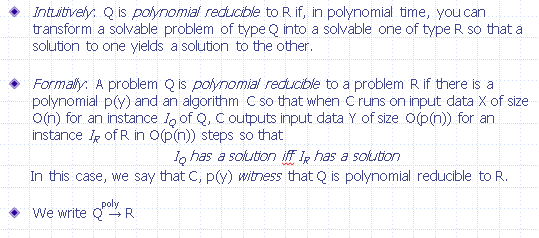
Is P = NP?

The answer is not known. If it were true, then thousands of problems that were believed to have infeasible solutions would suddenly have feasible solutions

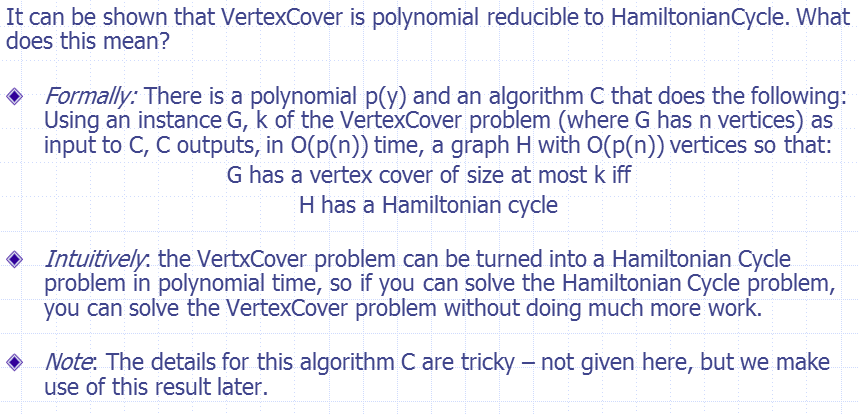
A Remarkable Fact about NP

* If someone ever figures out a polynomial-time algorithm to solve an NP-complete problem, then all problems in NP will also have polynomial time solutions
* One consequence: If anyone finds a polynomial time solution to an NP-complete problem, then P = NP

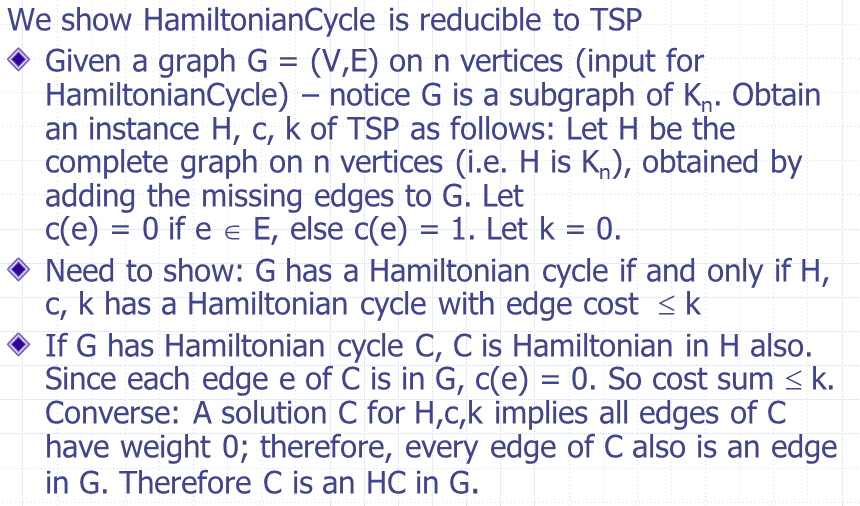
Reducibility



VertexCover 🡪 Hamiltonian Cycle



Hamiltonian Cycle 🡪 TSP



EXPtime and Exptime-Complete

* A problem Q is EXPTIME-complete if Q belongs to EXP, and for every problem R in EXP, R is polynomial time reducible to Q.
* Notice that although we don't know if NP is equal to P or not, we do know that EXPTIME-complete problems are not in P; it has been proven that these problems cannot be solved in polynomial time. These are harder than NP-Complete problems.

SAT - Satisfiability Problem (the first NP-Complete problem)

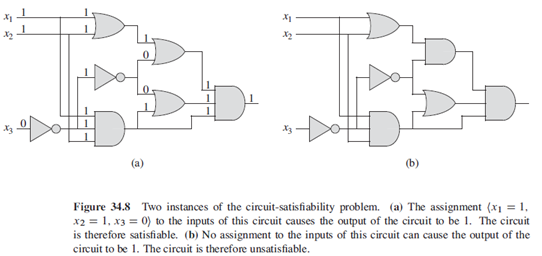
SAT is the following problem: Given an expression using boolean variables p,q,r,… joined together using connectives AND, OR, NOT, is there a way to assign values “true” or “false” to the variables so that the expression evaluates to true?  
  
Example: Consider   
 p AND NOT (q OR (NOT r))

Assigning T to p and to r and assigning F to q causes the expression to evaluate to T (i.e. “true”)

= T AND NOT (F OR (NOT T))

= T AND NOT (F OR F)

= T AND (NOT F) = T AND T = True



NP-Complete Problems (VertexCover is NP-complete)

VertexCover can be shown to be NP-complete. The main idea is to prove that SAT is polynomial reducible to VertexCover. Then: Given any NP problem Q, to show Q is polynomial reducible to VertexCover, observe:

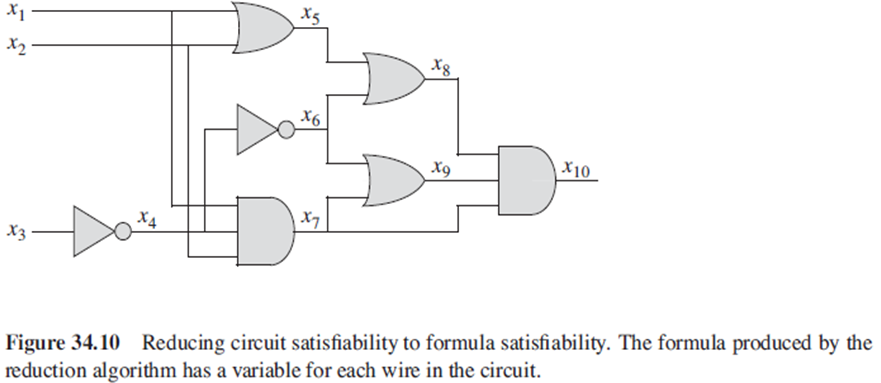
* Q is polynomial reducible to SAT
* SAT is polynomial reducible to VertexCover
* Therefore, Q is polynomial reducible to VertexCove

Hamiltonian Cycle is NP-Complete

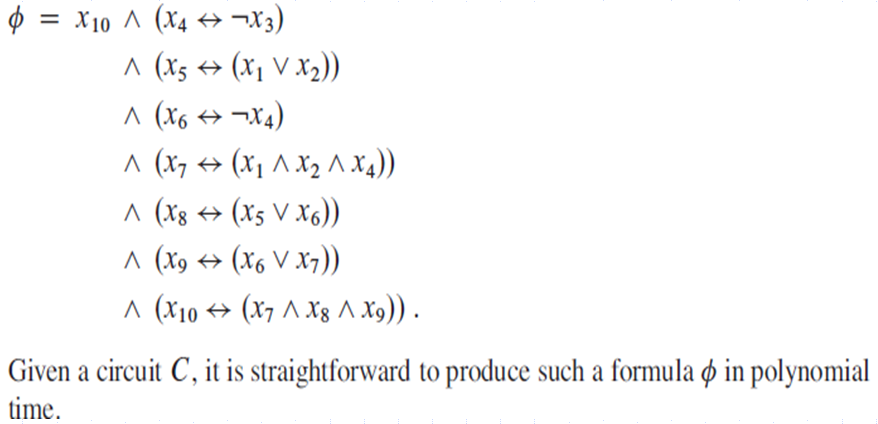
This is an outline of a proof that HamiltonianCycle is NP-Complete under the assumption that VertexCover is NP-complete:

* Recall that VertexCover is polynomial reducible to HamiltonianCycle.
* Therefore, since VertexCover is NP-complete, given any NP problem Q, Q is polynomial reducible to VertexCover
* Therefore, any such Q is polynomial reducible to HamiltonianCycle as well

CSAT is NP-Complete



SAT is NP-Complete

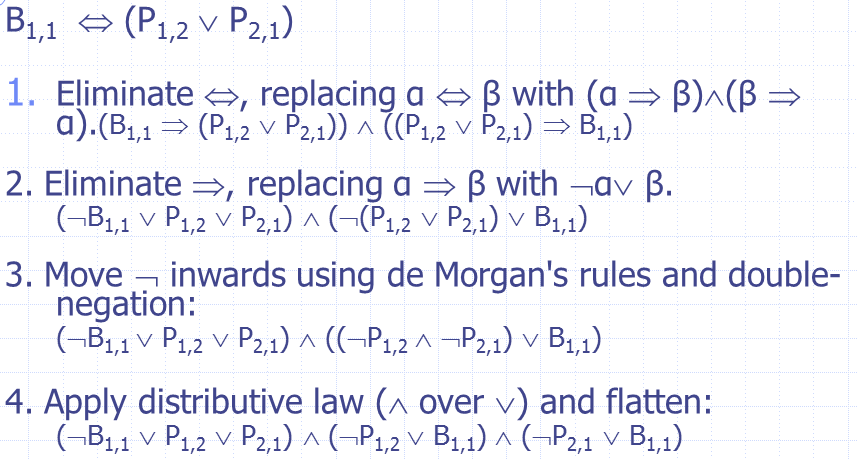


3-CNF is NP-Complete

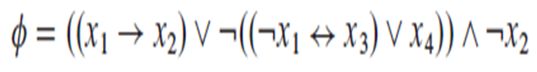
The following Boolean Formula is in 3-CNF (Conjunction of Disjunctions) form



Any Boolean Expression can be converted to CNF form



Thus the following formula can be converted to 3-CNF in Polynomial time



Thus, SAT can be reduced to 3-CNF in Polynomial time. Hence 3-CNF is NP-Complete as SAT is NP-Complete

Solving One NP-Complete Problem Solves Them All

* Suppose Q is an NP-complete problem and someone finds a polynomial-time algorithm A that solves it in O(p(n)) time
* Let R be any problem in NP. R is polynomial reducible to Q, with witness polynomial q(y)
* Polynomial-time algorithm to solve R: Given an instance IR of R, create an instance IQ of Q (in O(q(n)) time) that has a solution iff IR does. Solve IQ in O(p(q(n))) time. Output solution to IR. The algorithm runs in O(q(n) + q(p(n)))