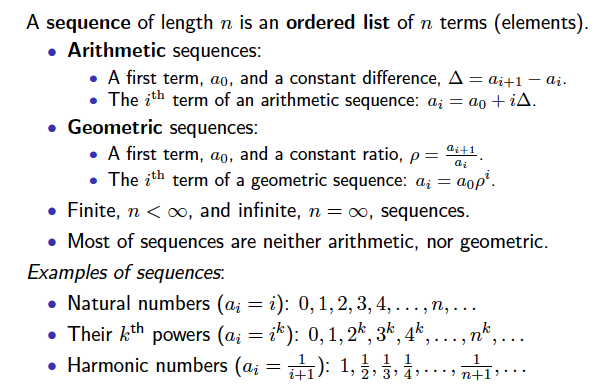
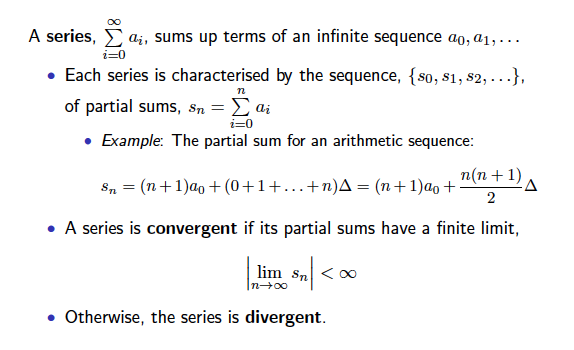
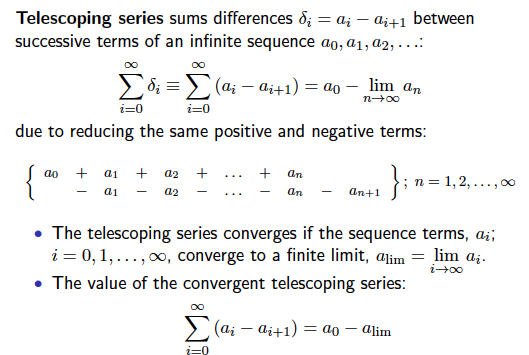
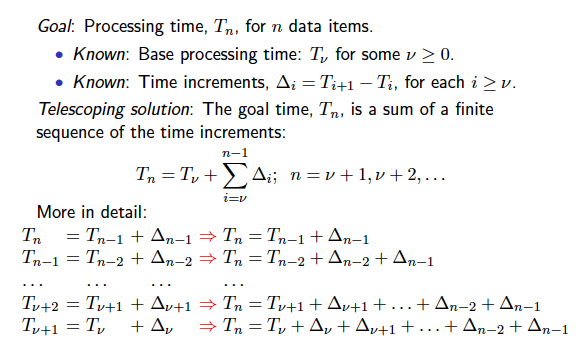
# COMPSCI 220 EXAM NOTES

# **PART 1**

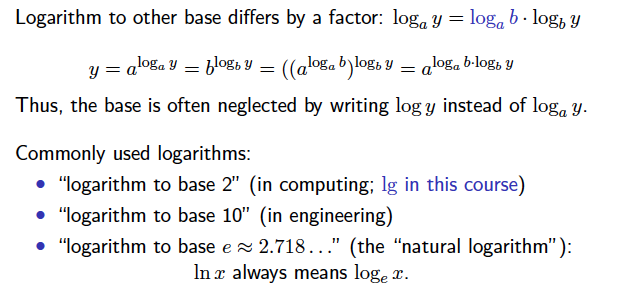
# LECTURE 1: Sets, sums, series

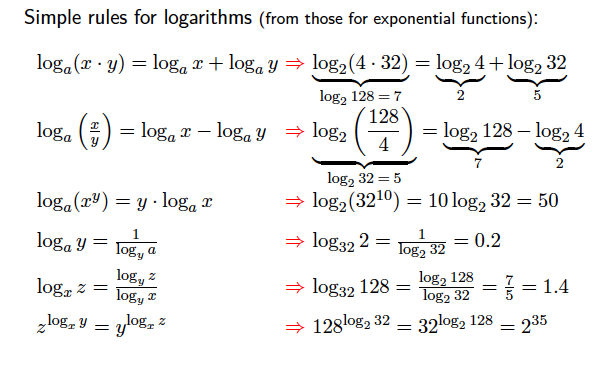
* Sets of elements:
  + A set X is a unordered collection of **zero or more** distinct and well-defined objects.
    - Such objects are called elements of the set.
    - A set can be considered an object by itself.
  + X = {3,4,5,6,7} – the set of integers from 3 to 7.
* Equivalent specifications:
  + Order of elements can be arbitrary so not particular format needed.
    - X = {3,5,6,7,4}, X = {6,7,4,3,5}, X = {7,3,5,6,4} etc.
  + Set elements can be defined implicitly:
    - X = {x | x is an integer and x 3 and x 7}
    - Reads “X is the set of all elements such that x is an integer and x is greater than or equal to 3 and x is less than or equal to 7”
    - Common notation: “{x |…}” or “{x: …}” reads the set of all x for which some condition applies
    - x ∈ X reads “x is an element of X: e.g. 5 ∈ {3,7,6,5,4}
    - x ∉ X reads “x is not an element of X: 8 ∉ {3,7,6,5,4}
* Cardinality of a Set
  + The number of elements of a set X is called the cardinality of X.
    - The cardinality of X is denoted |X|.
    - | {3,4,5,6,7} | = 5
    - Simply amount of numbers, objects in the set
    - Zero cardinality of the empty set, ∅, with no elements: |∅| = 0.
    - Sometimes the cardinality of X is denoted #X or card X.
* Sets and Subsets
  + A set Y **whose all elements** are also elements in a set X is a subset of X.
    - The relation is denoted Y ⊂ X when Y is a subset of X and |Y| < |X| (a “true” subset).
    - Example of a true subset: {3,4} ⊂ {3,5,6,7,4}
    - Example of not a subset: {3,7,9} ⊄ {3,4,5,6,7}
  + A set X is a **superset** of Y if Y is a subset of X:
    - The relation is denoted X ⊃ Y if Y ⊂ X (a “true” superset) or X ⊇ Y if Y ⊆ X.
    - Example of a true superset {3,5,6,7,4} ⊃ {3,4}
    - Example of not a superset: {3,4,5,6,7} not a superset {3,7,9}
* Union of Sets
  + The **Union** of two sets, X and Y denoted X U Y, is the set **containing all the elements of X and all the elements of Y:** 
    - X U Y = {x | x ∈ X OR x ∈ Y}
    - Note that duplicates are not permitted in a set!
    - Z = X U Y is the superset of X and of Y: Z ⊇ X and Z ⊃ Y
    - Examples:
      * X = {3,5,6,7,4}; Y = {2,5,7} => X U Y = {2,3,4,5,6,7}
      * X = {R, A, H, U, L}; Y = {I, S, S, A, R}
      * Z = X U Y = {R, A, H, U, L, I, S, S, A, R}
* Intersection of Sets
  + The intersection of two sets, X and Y, denoted X ∩ Y, is the set of the **elements that are contained in both X and Y:** 
    - X ∩ Y = {x | x ∈ X AND x ∈Y}
    - X = {M, I, K, A}; Y= {H, A, N, U, L} =>
    - Z = X ∩ Y = {A}
  + Two sets, X and Y, with no common elements are disjoint and their intersection is the empty set: X ∩ Y = {}, denoted empty set.
    - X = {3,5,6,7,4}; Y = {2,8,9} => X ∩ Y = empty set.
    - Nothing similar between the two sets
* Complement of sets
  + The complement of a set Y with respect to a set X, denoted X \ Y, is the set X ∩ Y:
    - X \ Y = {x | x ∈ X AND x ∉ Y}
    - The union of the intersection X N Y and the complement X \ Y is the set X:
      * (X ∩ Y) U (X \ Y) = X
    - Examples
      * X = {3,5,6,7,4}; Y = {5,7} => Z = X \ Y = {3,6,4}.
      * {M, I, K, A} \ {H, A, N, U, L} = {M, I, K}





# LECTURE 2: ROUNDING, LOGS, PROOFS

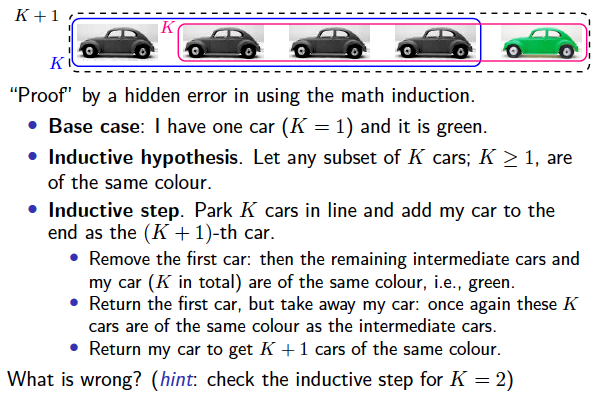
* Rounding real numbers notation:
  + ⎡x⎤ rounds up to the nearest integer larger than or equal to x:
    - x + 1 > i = ⎡x⎤
    - Examples: ⎡3.2⎤ = 4; ⎡10.01⎤ = 11; ⎡10.99⎤ = 11
  + Floor notation: ⎣x⎦ rounds down to the nearest integer smaller than or equal to x:
    - x – 1 < i = ⎣x⎦
  + Examples ⎣3.2⎦ = 3; ⎣10.01⎦ = 10; ⎣10.99⎦ = 10
* Logarithms:
  + The inverse of exponential functions: if = y, then
  +  reads “x is the logarithm to base a of y”

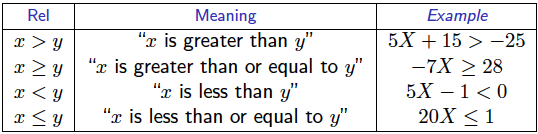


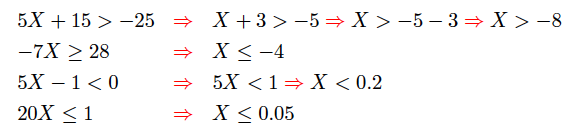
* A **definition** is used to make it clear what a certain term means, what we are going to call something, or how we will be using certain notation.
* A **Theorem** is a statement we claim to be true, and it always requires a **proof**
* A **lemma** is like a small theorem that we prove to lead us up to the proof of a more extensive theorem.
* As a general, we do not claim that something is true, unless we can also prove it.
* To find a statement is true, it should be either deduced (a direct proof), or prove by math induction or by contradiction.
* Mathematical Induction:
  + Proves that some statement is true for all integers, such that n , where is mostly a non-negative constant.
    - If the proof should be for all non-negative integers, = 0
    - If the proof should be for all positive integers, = 1
    - Basis: Prove that the statement is true for
    - Induction hypothesis: Assume that the statement is true for some n
    - Inductive step from n to n + 1: If the induction hypothesis holds, prove that the statement is also true for n + 1
  + The inductive step completes the proof.



* Proving a statement by Contradiction
  + A simple, but quite rare way: Find a counterexample
  + More general framework:
    - Assume the statement is false
    - Derive a logical contradiction from this assumption
    - Conclude the assumption is wrong and the statement is true.
  + Assume there exists the largest integer
  + Consider the number =
    - is integer as the sum of two integers
    - obviously, , that contradicts the assumption.
    - Thus, the assumption is wrong, and the statement is true.



* Inequalities, A Rel B, and their properties
  + Exactly one of three relationships: x < y (less than); x = y(equal to), or x > y (greater than) holds for any two real number x and y.
* Transitivity: If x < y and y < z, then x < z.
* Bias: if x < y and c is a real number, then x + c < y + c
* Positive scaling: if x < y and c > 0, then cx < cy.
* Negative scaling: If x < y and c < 0, then cx > cy.
* The same properties hold for , x > y, and
* Reversal: if x < y, then y > x
* Inverse: When x and y be both positive or both negative, it holds that if x < y then > and if x > y then > .
* Non-negative squares:
* Solution of inequalities – getting numerical bound(s) for the unknown(s) on the left of the inequality symbol:



Solution steps that do not affect direction of inequality:

* Adding a number to both sides:
  + 5X + 15 > -25 => 5X > -25 -15 => 5X > - 40.
  + Scaling with a positive factor: 5X > -40 => X > -8

Solution steps that reverse direction of inequality

* Scaling with a negative factor: -7X 28 => X - 4
* Swapping left – and right – hand sides

Scaling could be done if the sign of the factor is known (thus – do not scale by an unknown variable), e.g. 5xy 15y cannot be solved if y is unknown.

Systems of inequalities may have no solution: 5X > 10 and X < 0.

# LECTURE 3: ANALYSING COMPLEXITY OF ALGORITHMS

Time complexity:

* Constant time:
  + Statement 1;
  + Statement 2;
  + …
  + statement k;
  + This is code that really is exactly k statements;
  + If each statement is “simple” (only involves basic operations) then the time for each statement is constant and the total time is also constant: O(1).
* If-then-else statements
  + if(condition) {

sequence of statements 1

} else {

sequence of statements 2

}

* + Here, either sequence 1 will execute, or sequence 2 will execute. If sequence 1 is O(N) and sequence 2 is O (1) the worst-time case for the whole if-then-else statement would be O(N).
* for loops
  + for (i = 0; i < N; i++) {

sequence of statements

}

* + The loop executes N times, so the sequences of statements also executes N times. Since we assume the statements are O(1), the total time for the for loop is N \* O(1), which is O(N) overall.
* Nested loops
  + First, we’ll consider loops where the number of iterations of the inner loop is independent of the value of the outer loop’s index. For example:

for (i = 0; i < N; i ++) {

for (i = 0; i < N; i ++) {

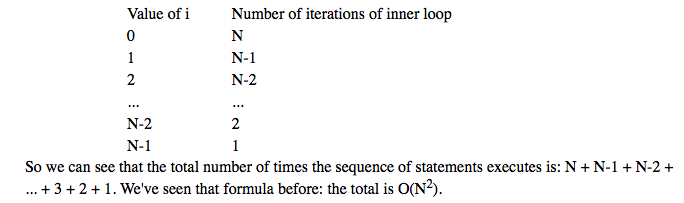
sequence of statements

}

}

The outer loop executes N times. Every time the outer loop executes, the inner loop executes M times. As a result, the statements in the inner loop execute a total of N \* M times. Thus, the complexity is O (N \* M). in a common special case where the stopping condition of the inner loop is j < N instead of j < M, the total complexity for the two loops is O (

Now we can’t just multiple the number of iterations of the outer loop times the number of iterations of the inner loop, because the inner loop has a different number of iterations each time. So, let’s think about how many iterations that inner loop has. That information is given in the following table:

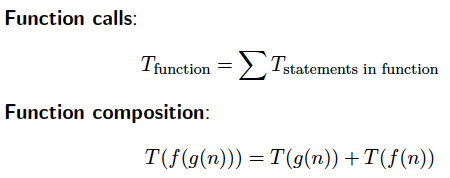


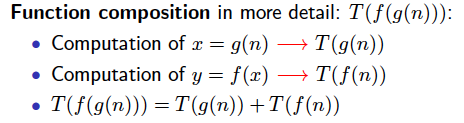


# LECTURE 4: RUNNING TIME EVALUATION

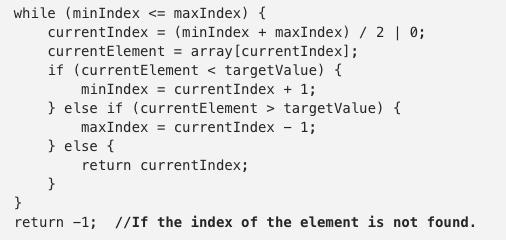
Running time:

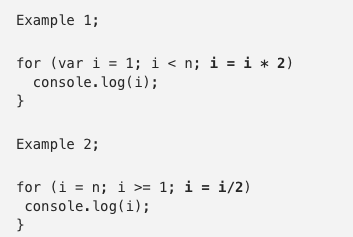
* + Simplifying assumptions: all elementary statements / expressions take the same amount of time to execute, e.g. simple arithmetic assignments, return, etc.
    - A single loop increases in time linearly as lambda \* of a loop where lambda is number of times the loop is executed.
    - Nested loops result in polynomial running time T(n) = if the number of elementary operations in the innermost loop is constant (k is the highest level of nesting and c is some constant)
    - The first three values of k have special names:
      * **Linear time** for k = 1 (a single loop);
      * **Quadratic time** for k = 2 (two nested loops), and
      * **Cubic time** for k = 3 (three nested loops).
  + Conditional / switch statements like
    - if {condition} then {const time T1} else {const time T2} are more complicated
    - One must account for branching frequencies = false = 1 – = true:
      * T = T1 + (1 - ) max {}





* Logarithmic running time:
  + O(log n) – logarithmic time complexity
    - An algorithm has logarithmic time complexity if the time it takes to run the algorithm is proportional to the logarithm of the input size n. An example is binary search, which is often used to search data sets.





* “Big-Oh”, “Big-Omega”, and “Big-Theta” Tools
  + Two algorithms have essentially the same complexity if their running times as functions of n differ only by a constant factor.
* “Big-Oh”
  + Let f(n) and g(n) be non-negative-valued functions, defined on non-negative integers, n.
    - Then g(n) is O(f(n)) (read “g(n) is Big Oh of f(n)) iff there exists a positive real constant, c, and a positive integer, , such that g(n) cf(n) for all n >
* “Big-Omega”
  + If a running time is Ω(f(n)), then for large enough n, the running time is at least k \* f(n) for some constant k.
  + We use the big-Ω notation for asymptotic lower bounds, since it bounds the growth of the running time from below for large enough input sizes.
  + The upper bound tells us what grows asymptotically faster than or at the same rate as our function whereas the lower bound tells us what asymptotically grows slower than or at the same rate as our function.
* “Big-Theta”
  + Is used to describe the asymptotic efficiency of algorithms. It is written Θ(f(n)) where n is an element of N (sometimes sets other than the set of natural numbers, N, are used).
  + Gives both lower and upper asymptotic bound.

# LECTURE 5: “Big-Oh”, “Big-Omega”, and “Big-Theta” Properties and Rules

Running time:

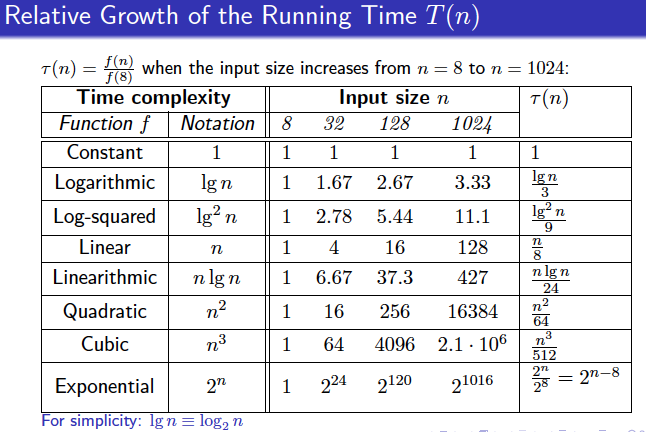
# LECTURE 6: RUNNING TIME EVALUATION

**Algorithm’s Efficiency**

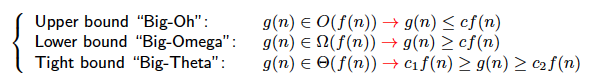
* Efficiency of an algorithm is generally measured in following terms:
  + The running time
  + The amount of memory it requires and other required computation resources
* Memory is relatively cheap at present, so that time complexity of an algorithm is very important.

A function f(n) such that the running time T(n) of a given algorithm is Θ(f(n)) measures the time complexity of the algorithm.

* A polynomial time algorithm: T(n) is O() where k is a fixed positive integer.
* An intractable computational problem: iff no deterministic algorithm with polynomial time complexity exists for it.
  + Many problems are classed as intractable only because a polynomial solution is unknown



**Asymptotic Bounds on running time**

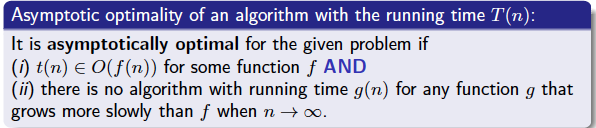
* Asymptotic notation measures the running time of the algorithm in terms of elementary operations, i.e. asymptotic bounds are independent of inputs and implementation:
* In general, the running time varies not only according to the size of the input, but the input itself.
  + Some sorting algorithms take almost no time if the input is already sorted in the desired order, but much longer if it is not.
* The most common performance measure for all inputs:
  + The worst-case running time and the average-case running time.

**The Worst-case Running Time**

* Advantages
  + An upper bound of the worst-case running time is usually easy to find
  + It is essentially for so-called “mission-critical” applications
* Drawbacks:
  + It may be too pessimistic: encountered inputs may lead to much lower running times that the “upper bound”
    - The most popular fast sorting algorithm, quicksort, is Θ() in the worst case, but Θ(nlogn) for “random” inputs, being most frequent in practice.
  + The worst-case input might be unlikely to be met in practice.
  + In many cases, it is difficult to specify the worst-case input.

**The Average-case Running Time**

* A probability distribution on the inputs must be specified
  + Simple assumption: all equally likely inputs of size n
  + Sometimes this assumption may not hold for the real inputs
* The analysis might be a difficult math challenge
* Even if the average-case complexity is good, the worst-case one may be very bad.



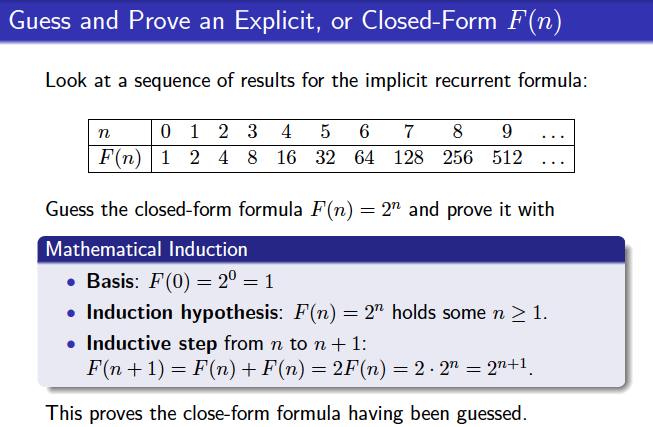
# LECTURE 7: Recurrent Algorithms

**Divide-and-Conquer Principle**

* Divide a large problem into smaller subproblems;
* Recursively solve each subproblem, then
* Combine solutions of them to solve the original problem

Running time: by a recurrence relation accounting for:

1. The size and the number of the subproblems and
2. The cost of splitting the problem into these subproblems



**Telescoping a Recurrence**

* Given: An implicit recurrence relation and its base condition for example:
  + T(n) = 2T (n – 1) + 1; T (0) = 0
* Find: The closed-form (explicit) formula for T(n) by recursive substitution of the same implicit formula:

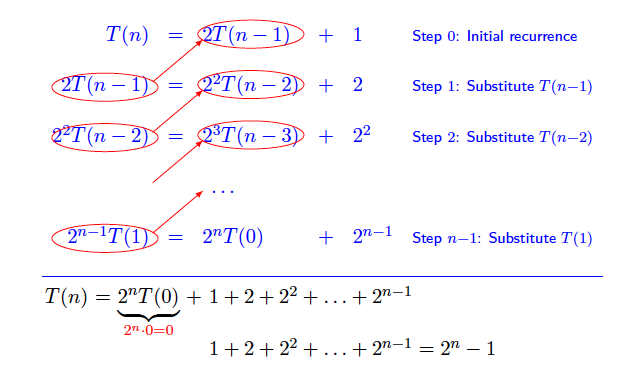
T(n) = 2T (n – 1) + 1

T (n – 1) = 2T (n – 2) + 1

…

T (2) = 2T (1) + 1

T (1) = 2T (0) + 1 = 1



**Capabilities and Limitations**

* Rough time complexity analysis cannot result immediately in an efficient program
  + But it helps to predict empirical running time of the program
  + Limitations of the “Big-oh/Theta/Omega” analysis:
  + It hides the constants crucial for a practical task
  + It is unsuitable for small input
  + It is unsuitable if costs of access to input data items vary
  + It is unsuitable if there is lack of sufficient memory
* However, time complexity analysis provides ideas how to develop new and efficient algorithms.

# LECTURE 8: Mergesort

**Basic idea**

* If the number of items is 0 or 1, return; otherwise:
  + Separate the list into two lists of equal or nearly equal size
  + Recursively sort the first and the second halves separately.
* Finally, merge the two sorted halves into one sorted list.
* Almost all the work is performed in the merge steps.

**Correctness of Mergesort**

Proof: by induction on the size n of the list.

* Basis: if n =0 or 1, merge is correct
* Inductive hypothesis: Mergesort is correct for all m < n
* Inductive step:
  + Mergesort calls itself recursively on two sublists
  + Each of these sublists has size less than n and thus is correctly sorted by induction hypothesis
  + Provided that the merge step is correct, the top-level call of mergesort returns the correct answer.
* Linear time merge, theta(n) yields complexity theta (n log n) for mergesort.
* The merge is at least linear in the total size of the two lists: in the worst case every element must be looked at for the correct ordering

**Analysis of Mergesort**

Theorem 2.11: The running time of mergesort on an input list of size n is Theta(n log n) in the best, worst, and average case.

Proof. The number of comparisons used by mergesort on an input of size n satisfies a recurrence of the form:

Screen%20Shot%202018-06-03%20at%206.30.19%20PM.png

It is straightforward to show that T(n) is theta(n log n)

* The other elementary operations in the divide and combine steps depend on the implementations of the list, but in each case their number is theta(n)
* Thus these operations satisfy a similar recurrence and do not affect the theta (n log n) answer.

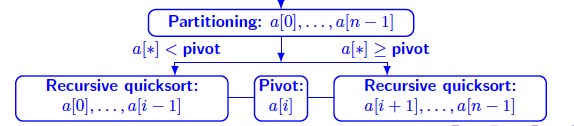
# LECTURE 9: Quicksort

**Basic idea**

* Unlike mergesort, subarrays for sorting and merging are formed dynamically, depending on the input, rather than the predetermined.

**Basic Recursive Quicksort**

* Choose one of the items in the list as a pivot.
* Next, partition the remaining items into two disjoint sublists, such that all items greater than the pivot follow it, and all elements less than the pivot precede it.
* Finally, return the result of quicksort of the “head” sublist, followed by the pivot, followed by the result of quicksort of the “tail” sublist.



Proof: by math induction on the size n of the list.

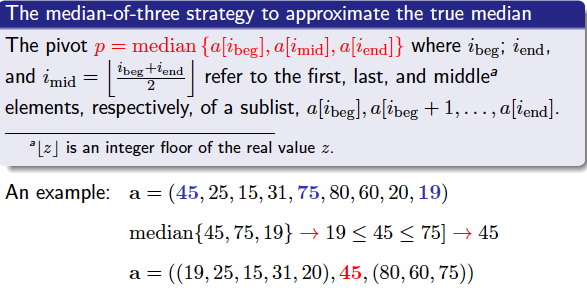
* Basis. If n = 1, the algorithm is correct.
* Hypothesis. It is correct on lists of size smaller than n.
* Inductive step: after positioning, the pivot p at position I;
  + I = 1, … n – 1, splits a list of size n into the head sublist of size i and the tail sublist of size n -1 – i
    - Elements of the head sublist are not greater than p
    - Elements of the tail sublist are not smaller than p
    - By the induction hypothesis, both the head and the tail sublists are sorted correctly.
    - Therefore, the whole list of size n is sorted correctly.
  + Any implementation specifies what to do with items equal to the pivot.

**Analysing quicksort: The worst case**

* The wrong choice may lead to the worst-case quadratic time complexity
* A good choice equalises both sublists in size and leads to linearithmic (n log n) time complexity.
* The “worst-case” choice: the pivot happens to be the largest (or smallest) item
  + Then one subarray is always empty
  + The second subarray contains n – 1 elements, i.e. all the elements other than the pivot
  + Quicksort is recursively called only on this second group
* However, quicksort is fast on the “randomly scattered” pivots.

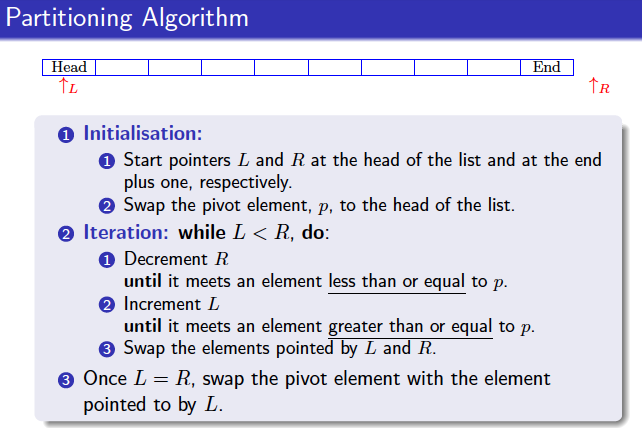
**Active Pivot Strategy**

* The best active pivot – the exact median of the list, dividing it into (almost) equal sized sublists, is computationally inefficient.



**Active Pivot Strategy**

* Bad performance is still possible with the median-of-three strategy, but becomes much less likely, than for a passive strategy.
* Random choice of the pivot
  + The expected running time is theta (n log n) for any given input.
  + No adversary can force the bad behaviour by choosing nasty inputs.
  + A small extra overhead for generating a “random” pivot position.
  + Bad cases: only by bad luck, independent of the input
  + An alternative: to first randomly shuffle the input in linear, theta(n), time and use then the naïve pivot selection.



**Correctness of Partitioning**

* Proof. After each swap of elements, a[L] and a[R],
  + Each element to the left of index L, as well as a[L], is less than or equal to the pivot p;
  + Each element to the right of index R, as well as a[R], is greater than or equal to the pivot p.
* After the final swap of p with a[L], which does not exceed p, all elements smaller than p, are to its left, and all larger are to its right.

Quicksort is easier to program for array, than other types of lists.

Constant-time pivot selection is only for arrays, but not linked lists.

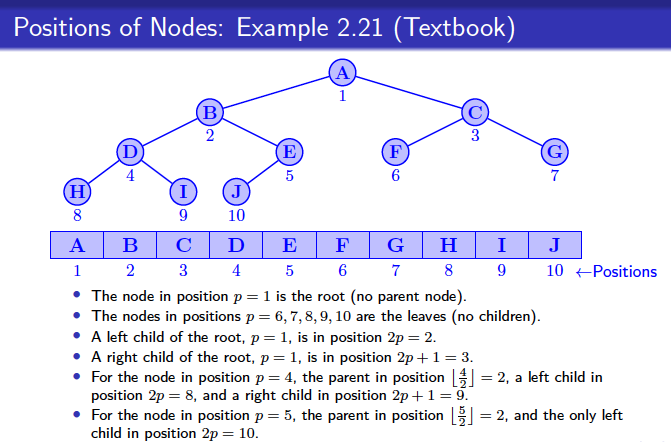
* What time will the median-of-three take for a linked list?

Partition needs a doubly-linked list to scan forward and backward.

# LECTURE 10: Heaps, Heap operations and Heapsort

**Basic idea of heapsort**

* Convert an array into a maximum heap in linear time theta (n)
* Sort the heap in theta (n log n) time by deleting n times the maximum item from the maximum heap (or the minimum item for the minimum heap, respectively).
  + Each deletion of the maximum (or minimum) item takes the logarithmic time, theta (log n)



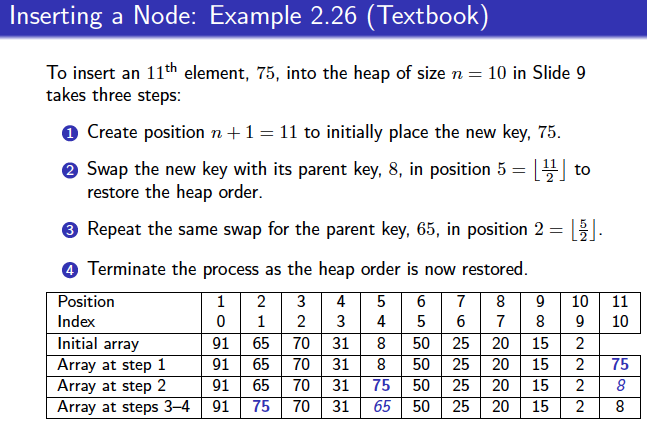
**Binary heap**

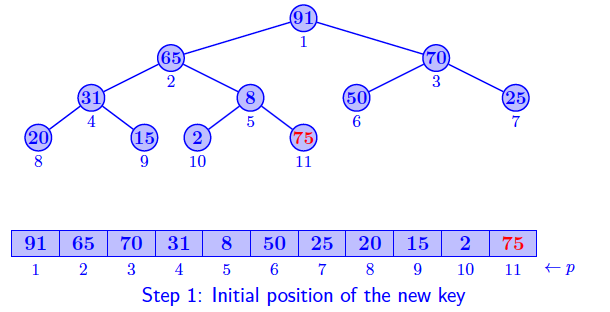
* A (maximum) heap is a complete binary tree having a numerical key associated with each node, such that the key of each parent node is greater than or equal to the keys of its child node.
* The heap order provides easy access to the maximum key associated with the root.
  + Alternatively, a minimum heap has the key of each parent node, which is less than or equal to the keys of its child nodes
  + Then the minimum key is associated with the root

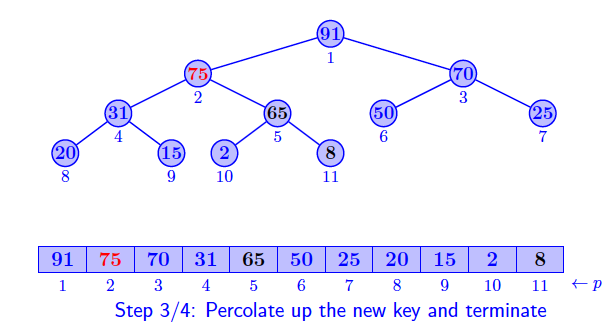
**Inserting a node into a heap**

Proof:

* Create a new, (n + 1)-st, leaf position.
* Place the new node with its associated key in this leaf
* If the inserted key preserves the heap order, the insertion is complete
* Otherwise, bubble up, or percolate up the new key towards the root by repeatedly swapping it with its parent until the heap order is restored.
* There are at most h swaps, where h is the heap height, so that the running time is O (log n)



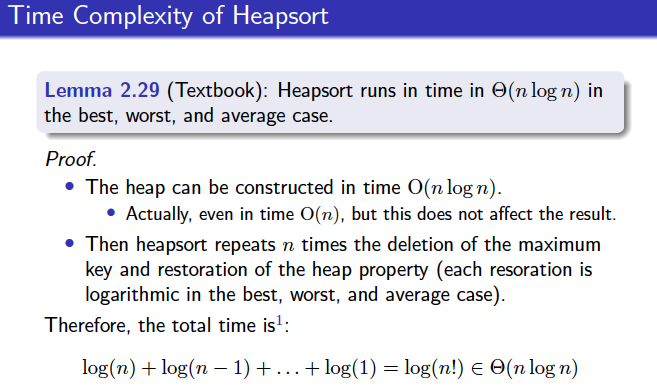




**Deleting the maximum key**

Deleting the maximum key, 91 from the previous example.

* Place key 2 from the eliminated position 10 at the root.
* Percolate the new root key down by comparing to its children 65 and 70 positions 2 = 2 \* 1 and 3 = 2 \* 1 + 1, respectively, and swapping with the larger child, 70, to restore the order.
* Repeat the same swap for the children 50 and 25 in positions, 6 = 2 \* 3 and 7 = 2 \* 3 + 1
* Terminate the process, because the heap order is now correct.



**Building a Heap in Linear Time**

Heap as a recursive structure: left subheap <- root -> root subheap

A heap can be built from a list of size n in theta (n) time.

Proof:

* To form the heap, each of the two subtrees attached to the root are transformed into heaps of height at most h -1
  + The left subtree is always height h -1, whereas the right subtree could be of height h – 2
* In the worst case the root percolates down the tree for at most h steps that takes O (h)
* Thus, the worst-case time T(h) to build a heap of height at most h steps that takes time O(h)

# LECTURE 11: Data selection

Average complexity of selection with sorting O (n log n)

**Quickselect**

Quickselect is a selection algorithm to find the k-th smallest element in an unordered list. The algorithm is like QuickSort. The difference is, instead of recurring for both sides, it recurs only for the part that contains the k-th smallest element. If the index of partitioned element is more than k, then we recur for left part. If index is same as k, we have found the k-th smallest element and we return. If index is less than k, then we recur for right part. This reduces the expected complexity from O(n log n) to O (n), with a worst case of O (n^2).

1. If n = 0 or 1, return “not found” or the list item, respectively.
2. Else choose one of the items as a pivot
3. Partition the remaining items into a disjoint head and tail sublists with j and n – j – 1 items, respectively, separated by the pivot at position with index j.
4. Return the results of Quickselect on the head if k < j + 1; the pivot if k = j + 1, or the result of Quickselect on the tail if k > j + 1.

# LECTURE 12: Data searching and Binary search

**Data search**

Searching in a database D of records, such that each record has a key to use in the search

The search problem: given a key k, either:

* Return the record associated with k in D(a successful search: if k occurs several times, return any occurrence), or
* Indicate that k is not found, without altering D (an unsuccessful search).

The purpose of the search:

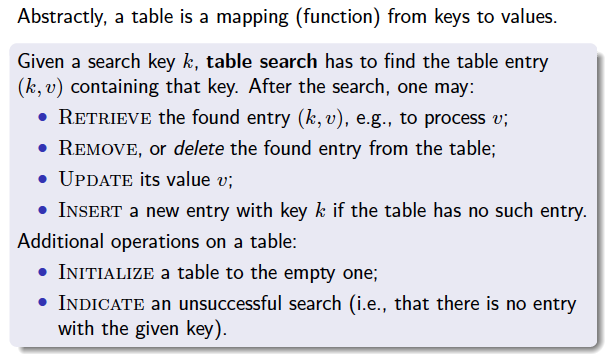
* To access data in the record for processing, or
* To update information in the record, or
* To insert a new record or delete the record found
* An abstract data type (ADT) table is a set of ordered pairs, or table entries (k, v) where k is a unique key and v is a data value associated with the key.
* A table is also called an associative array, or a dictionary.

**Tables**

The ADT table relates a disjoint set of keys to an arbitrary set of values

* A key and a value are linked by association
* Keys of entries may not have any ordering relation
* Keys of entries may be of unknown range
* No upper bound on the table size
  + An arbitrary number of different data items can be maintained simultaneously.
* No analogy with a convention word dictionary, which has a lexicographical order.

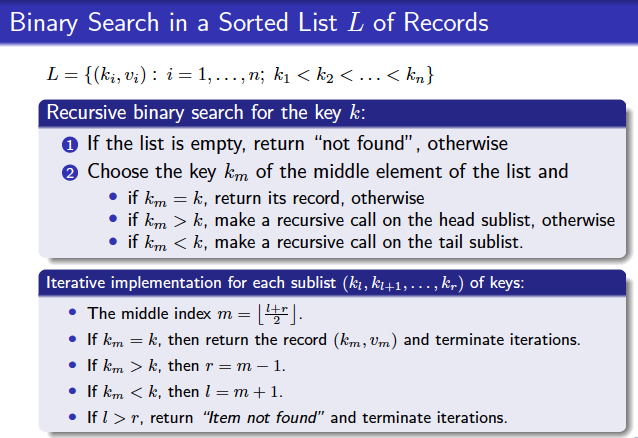
**Basic Operations for Tables**



**Types of search**

* Static search: unalterable (fixed in advance) databases; no updates, deletions, or insertions
* Dynamic search alterable databases (allowable insertions, deletions, and updates)
* Basic implementations of the table ADT: lists and trees
  + An elementary operation: a query or update of a list element or tree node, or comparison of two of them.

**Sequential Search in Unsorted Lists**

* Starting at the head of the list and examining elements one by one until finding the desired key or reaching the end of the list.
* Proof: The unsuccessful search explores each of n keys, so the worst-and-average case time is theta (n)
  + The sequential search is the only option for unsorted arrays and linked lists of records
  + A sorted list implementation allows for much better search based on the divide-and-conquer paradigm.

**Dynamic Binary Tree Search**

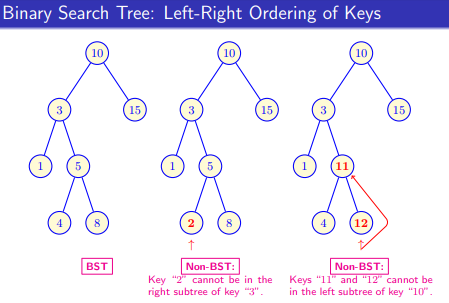
* Static binary search is converted into a dynamic binary tree search by allowing for insertion and deletion of data records.
  + Dynamic binary tree search makes actual use of the binary search tree (BST) data structure
  + The BST data structure is constructed by linking data records.
  + A BST allows for inserting a new node
  + Any existing node of a BST may be removed.
  + Using an array implementation of a sorted list, both successful and unsuccessful search, retrieval, and updating take time in theta (log n) on average and in the worst case.
    - But insertion and deletion are in theta (n) in the worst and average case.
  + Using a linked list, all the above operations take time in theta (n)

# LECTURE 13: Binary Search Trees

**Basics**

Left-to-right numerical ordering in a BST: for every node i,

* The values of all the keys in the left subtree are smaller than the key (root node)
* The values of all the keys in the right subtree are larger than the key (root node)
* Compare to the bottom-up ordering in a heap where the key (root node) of every parent node I is greater than or equal to the keys Kl and Kr in the left and right child node l and r.

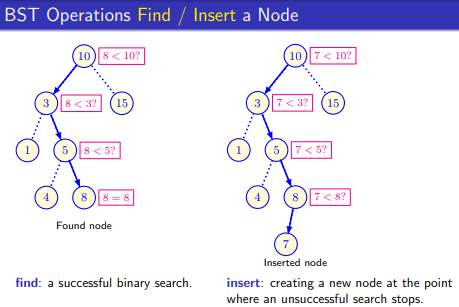


**Basic BST operations**

BST is an explicit data structure implementing the table ADT.

* BST are more complex than heaps: any node may be removed, not only a root or leaves.

Basic operations:

* **Find** a given search key or detect that it is absent in the BST.
* **Insert** a node with a given key to the BST if it is not found
* **FindMin:** Find the minimum key.
* **FindMax:** Find the maximum key
* **Remove:** A node with a given key and restore the BST if necessary

**BST operations: FindMin / FindMax**

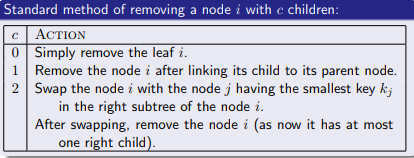
Extremely simple: starting at the root, branch repeatedly left (FindMin) or right (FindMax) as long as a corresponding child exists.

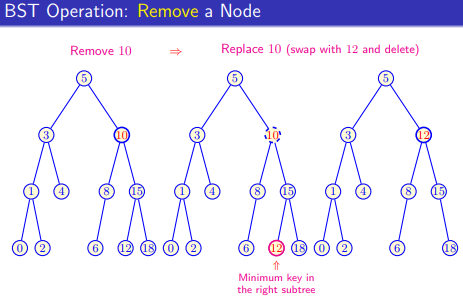
* The root of the tree plays a role of the pivot in quicksort
* As in quicksort, the recursive traversal of the tree can sort the items:
  + First visit the left subtree
  + Then visit the root, and
  + Then visit the right subtree
* O(log n) average-case and O(n) worst-case running time for find, insert, FindMin, and FindMax operations.

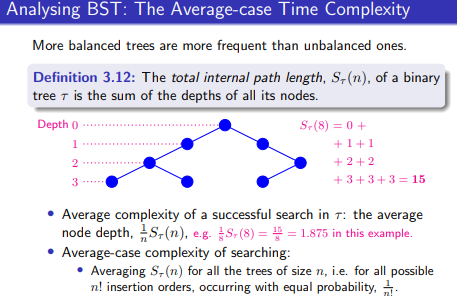
**BST Operation: Remove a Node**

The most complex because the tree may be disconnected.

* Reattachment must retain the ordering condition
* Reattachment should not needlessly increase the tree height.







**Self-balanced Search Trees**

Balancing ensures that the total internal path lengths of the trees are close to the optimal value of n log n

* The average-case and the worst-case complexity of searching is O(log n) due to the resulting balanced structure
* But the insertion and removal operations take longer time on average than for the standard binary search trees.

Balanced BST:

* AVL trees
* Red-black trees
* AA-trees

Balanced multiway search trees:

* B-trees

**Self-balancing BSTs: AVL Trees**

* Complete binary trees have a too rigid balance condition to be maintained when new nodes are inserted.
* For any nodes in the tree, the height of the left and right subtrees can differ by at most 1
* The height of an empty subtree is -1
* Advantages of the AVL balance property
  + Guaranteed height theta (log n) for an AVL tree
  + Less restrictive than requiring the tree to be complete.
* All self-balancing binary search trees use the idea of rotation
* Rotations are mutually inverse and change the tree only locally.
  + If there is a subtree of large height below the node a, the right rotation will decrease the overall tree height.
* Balancing of AVL trees requires extra memory and heavy computations

**Self-balancing BSTs: Red-black Trees**

* Every node is coloured either red or black
* Every non-leaf node has two children
* The root is black
* All children of a red node must be black
* Every path from the root to a leaf must contain the same number of black nodes.

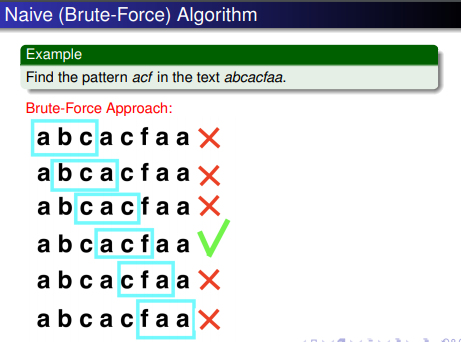
**Self-balancing BSTs: Red-black and AA trees**

* Searching in a red-black tree is logarithmic, O (logn)
* Non-precise average0case analysis
* AA-trees: red-black trees where the left child may not be red – are even more efficient if node deletes are frequent.
  + Used for storing and retrieving ordered data efficiently.
  + Supports efficient addition and deletion of entries.
  + Red nodes can only be added as a right sub child.

# **PART 2**

# LECTURE 1: String searching Algorithms

String searching algorithms are important class of string algorithms that try to find a place where one or several strings (also called patterns) are found within a larger searched text (string).



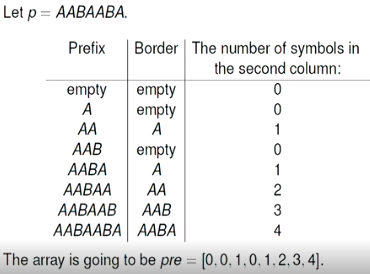
**Native (Brute-Force) Algorithm**

Steps:

* Pick the first m consecutive symbols s. If they equal to p then we found p otherwise go to Step 2
* Pick m consecutive symbols s starting from the second symbol. If they are equal to p then we found p otherwise go to Step 3.
* ….
* Pick m consecutive symbols of s starting from the ith symbol. If they equal to p then we found p otherwise go to next step.
* Pick m last consecutive symbols of s. If they equal to p then we found p otherwise p is not in the string s.
* The running time is O (m(n-m + 1))

**Knuth-Morris-Pratt Algorithm: Pre-processing Phase**

* A proper prefix of p is a prefix which is different from p.
  + Proper prefix of ABCD are empty, A, AB, ABC
  + Everything except the whole word
* The border of p is the largest proper prefix and a suffix of p
  + The border of 101010 is 1010
  + The border of ABCCCCCCAB is AB
* Whenever we detect a mismatch (after some matches), we already know some of the characters in the text of next window
  + We take advantage of this information to avoid matching the characters that we know will anyway match.



# LECTURE 2: String searching Algorithms

**KNP Algorithm**

* Pre-processing phase time complexity is in O(m)
* Searching phase is O (n) time complexity (independent from the alphabet size)
* The total time is in O (m + n)

This lecture just contained plenty of examples. Can go back to it to double check.

# LECTURE 3: String searching Algorithms

**Boyer-Moore Algorithm: idea**

In Boyer-Moore algorithm we move along the string from left to right but start comparing a segment of the string with a pattern from right to left. We have 2 rules:

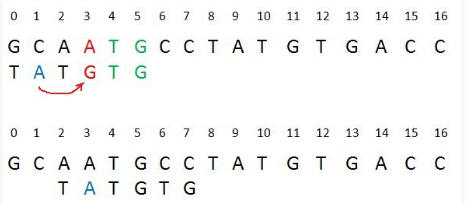
* Bad character rule
* Good suffix rule.

We choose the biggest skip suggested by the two rules.

**Bad character rule**

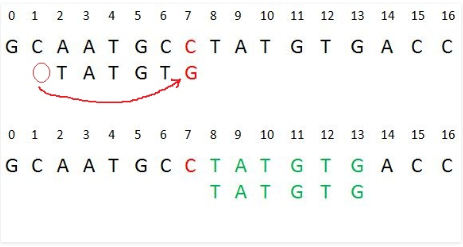
* The idea of a bad character heuristic is simple. The character of the text which doesn’t match with the current character of pattern is called the **Bad character**. Upon mismatch we shift the pattern until
  + The mismatch become a match
  + Patter P move past the mismatch character

Case 1 – mismatch become match

We will look up the position of last occurrence of mismatching character in pattern and if mismatching character exist in patter then we’ll shift the pattern such that it get aligned to the mismatching character in text T.

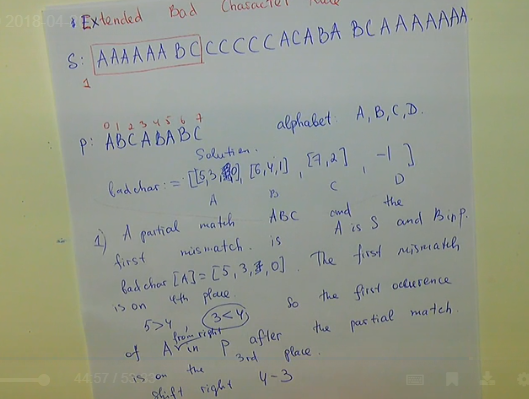
Case 2 – Patter move past the mismatch character

We’ll look up the position of last occurrence of mismatching character in pattern and if character does not exist we will shift pattern past the mismatching character.



With the weak bad character rule we only care about the right most occurrence, where as for the extended bad character rule we care about any occurrence.

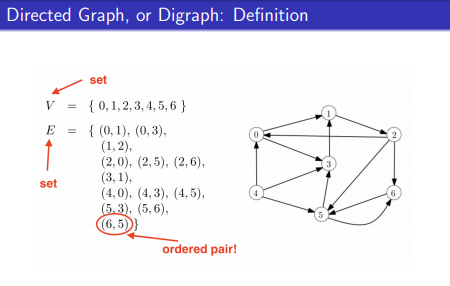
Example of extended bad character rule using the array size. Shifting according to the array.



# LECTURE 4: Graph intro

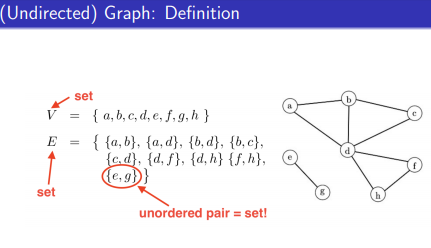
**Directed Graph, or Digraph: Definition**

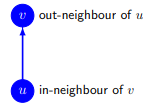
* A directed graph is graph, i.e. a set of objects (called vertices or nodes) that are connected together, where all the edges are directed from one vertex to another. A directed graph can also be called a digraph or a directed network.



**Undirected Graph: Definition**

* An undirected graph is graph, i.e. a set of objects (called vertices or nodes) that are connected together, where all the edges are bidirectional.



**Digraph: Relations of Nodes**

If (u, v) are an element of E,

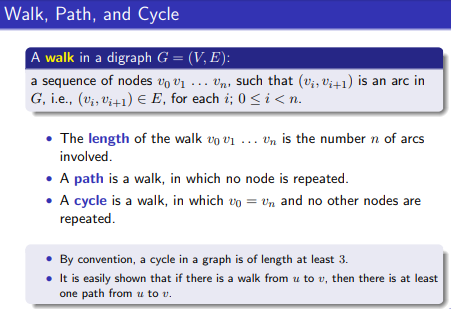
* V is adjacent to u;
* V is an out-neighbour of u, and
* U is an in-neighbour of v.

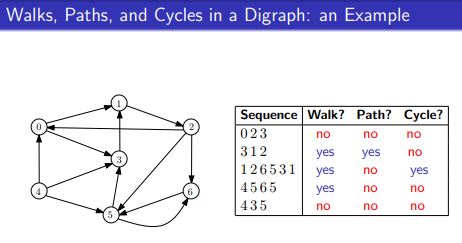


**Order, Size, and In- / Out-degree**

* The order of a digraph G = (V, E) is the number of nodes n = |V| is basically the cardinality of its vertex set
* The size of a digraph G = (V, E) is the number of arcs, m = |E| cardinality of its edge set
* The in-degree or out-degree of a node v is the number of arcs entering or leaving v, respectively.

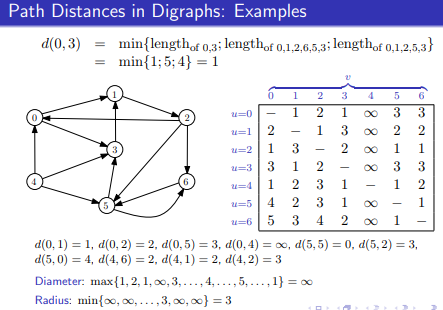
**Walk, Path, and Cycle**



* Walk: follows successive edges can repeat vertices or edges
  + The length of these paths is number of edges we have gone through, not the vertices
* Path: walks through vertices without repeating vertex, can’t go on a repeating vertex.
* A simple cycle is a cycle with no repeated edges or vertices except its first and last vertices.

**Distance and Diameter**

* The distance, d(u,v), from a node u to a node v in G is the minimum length of a path from u to v.
  + If no path exists, the distance is undefined or infinity.
* The diameter of G is the maximum distance between any two nodes
* The eccentricity of a node u is the greatest distance between u and any other node
* The radius of G is the minimum eccentricity over all nodes

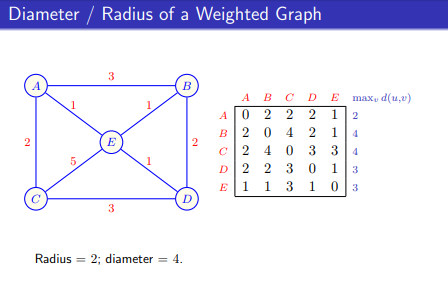


**Weighted Di(graphs)**

A weighted digraph is a pair (G, c), where G is a digraph and c is a function associating a real number to each arc of G. We call c a cost function.

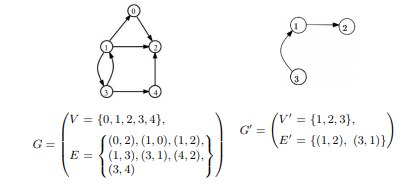
* We interpret c (u, v) as the cost of using arc (u, v) for some purpose
* For weighted (di) graphs, the distance between two nodes d (u, v) is the minimum sum of the weights of the arcs in a path from u to v.
* The path with the minimum sum of weights is not necessarily a path with the fewest arcs.

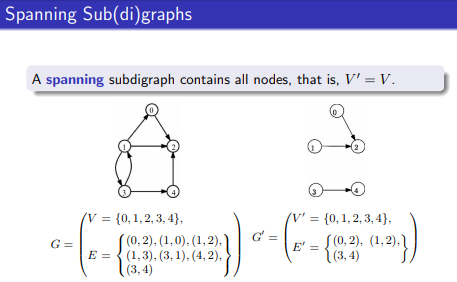
The underlying graph: For a digraph, the underlying graph is the undirected graph created using all of the vertices in and replacing all arcs in with undirected edges.

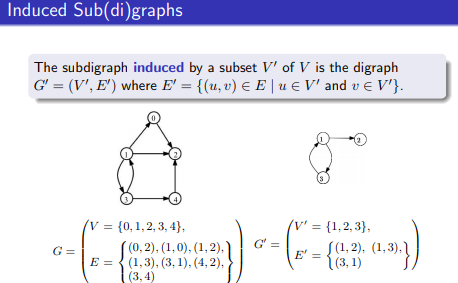


**Sub digraphs**

For a digraph, a sub-digraph of G is a digraph whose vertices and arcs are also G







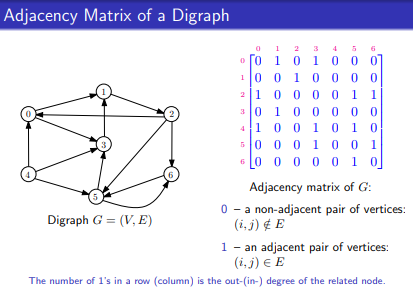
**Adjacency matrix and list**

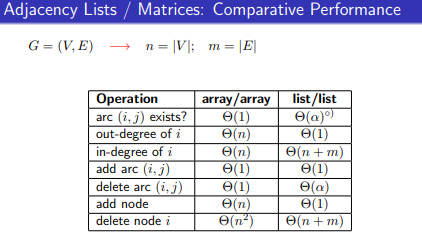
The adjacency matrix of G:

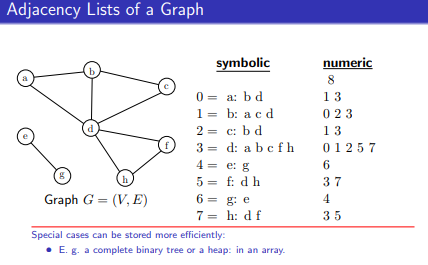
* The n x n Boolean matrix (often encoded with 0’s and 1’s) such that its entry (i, j) is true if and only if there is an arc (i, j) from the node i to node j. So, if there is a neighbour then it will be a 1 if not then it’s a 0.

An adjacency list of G:

* A sequence of n sequences, L0, …, Ln-1, such that the sequence Li contains all nodes of G that are adjacent to the node i.







# LECTURE 5: Introduction to Graph Algorithms

**Motivation: graph traversal**

Want to visit each node of a graph in a systematic and efficient way.

**General graph traversal: Colour scheme**

* White nodes: have not yet been visited
* Grey (frontier) nodes: have been visited but may have adjacent nodes that are white
* Black nodes: have been visited and all their (out-) neighbours have been visited as well.

Visits:

* S is coloured grey and pred[s] = null
* Choose a grey node u
* If u has a white (out)- neighbour v, then colour v grey and pred[v] = u else colour u black
* If we have grey nodes go to (2)

**A search forest**

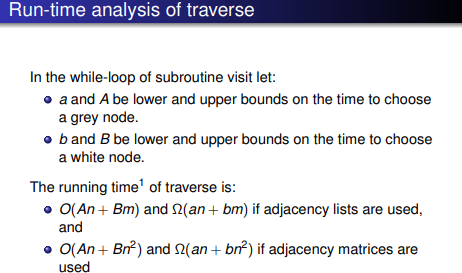
A search forest is a collection of node-disjoint trees that span the digraph and contain, for each node u with pred[u] doesn’t equal NULL, the edge (pred[u], u).

**Traversal arc classifications**

The arc is called a tree arc if it belongs to one of the trees of F. If the arc is not a tree arc, there are three possibilities:

* A forward arc if u is an ancestor of v in F,
* A back arc if u is a descendant of v in F, and
* A cross arc if neither u nor v is an ancestor of the other in F.

# LECTURE 6: Introduction to Graph Algorithms

****

**Graph Traversal**

* Breadth-first search (BFS)
* Depth-first search (DFS)
* Priority-first search (PFS)

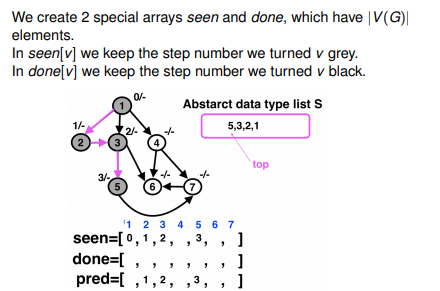
BFS and DFS are dual in some sense. In BFS we choose a new grey node to be the one that has been grey for the longest time. By contrast, in DFS, we choose the one that has been grey for the shortest time.

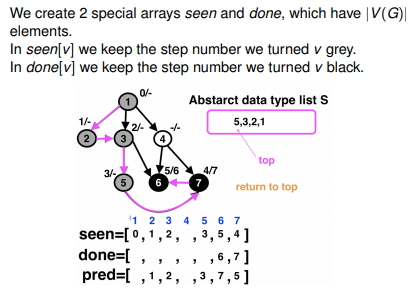
**The abstract data type stack**

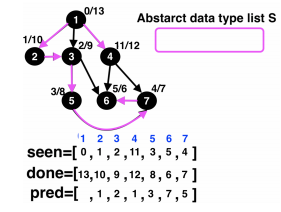
* Special list in which all operations occur at the same end (top) (last in first out)
* Add an element to the list (insert or push)
* Delete an element (delete or pop)
* Return top element without deleting it (GETTOP or PEEK)

**The abstract data type stack in use: dfsvisit(s)**

* We create 2 special arrays seen and done, which have |V(G)| elements
* In seen[v] we keep the step number we turned v grey
* In done[v] we keep the step number we turned v black.







**Basic properties of depth-first search**

* Each call to dfs\_visit(v) terminates only when all nodes reachable from v via a path of white nodes have been seen

# LECTURE 7: Introduction to Graph Algorithms

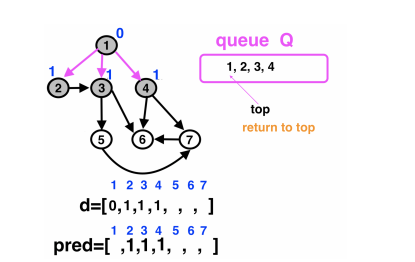
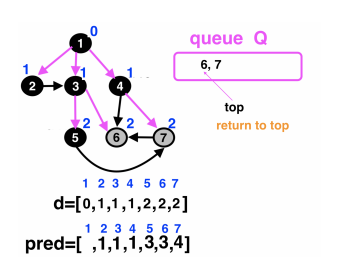
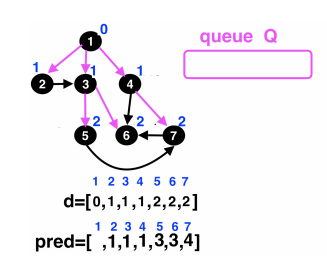
**Breadth First Search**

* BFS assigns two values to each vertex v:
  + A distance, giving the minimum number of edges in any path from the source vertex to vertex v.
  + The predecessor vertex of v along some shortest path from the source vertex.
  + The source vertex’s predecessor is some special value, such as null, indicating that it has no predecessor.
* If there is no path from the source vertex to vertex v, then v’s distance is infinite and its predecessor has the same special value as the source’s predecessor.

**The abstract data type queue in use: bfsvisit(s)**

We create a special array d, which have |V(G)| elements. In d[v] we keep the “level” of v. Q is the abstract data type queue. Bfsvisit(s)

1. S is coloured grey, s is made the top of Q, and d[s] = 0
2. Choose the top of Q, which is a grey node u.
3. If u has a white (out) – neighbour v then colour v grey and pred[v] = u, d[v] = d[u] + 1, add v at the end of Q, go to (2)
4. Colour u black and remove u from Q
5. If Q is not empty go to (2)

Look at distance as per level not how long it takes to get to the node.

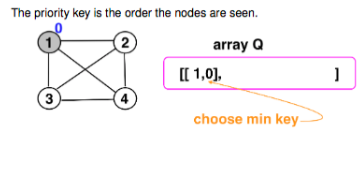
# LECTURE 8: Introduction to Graph Algorithms

**Priority-first-search (PFS) algorithm**

* In BFS we choose a new grey node to be the one that has been grey for the longest time.
* In DFS we choose a new grey node to be the one that has been grey for the shortest time.
* In PFS we choose a new grey node to be the one given to us by priority. The priority rules can be different.
* During PFS we create an array Q, where each element is an array with two elements [ a node, priority key]. Pfsvisit(s)
  + S is coloured grey, [s, key(s)] is added to Q.
  + Choose an element of Q, which has the minimum key value. Let it be a grey node u.
  + If u has a white (out) –neighbour v then colour v grey and add [v, key(v)] to Q, else delete [u, key(u)] from Q and colour u black.
  + If Q is not empty go to (2)

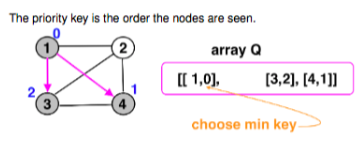
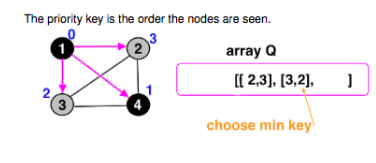
Priority can:

* Be fixed from the very beginning
* Defined during the algorithm run and stay stable.
* Be defined and updated during the algorithm run.



* Be consistent with the white neighbours you choose, like if you choose with the biggest node make sure you keep following that. If it’s random then choose at random.

Once all nodes are grey you can start slowly removing the nodes from the array and making the nodes black.

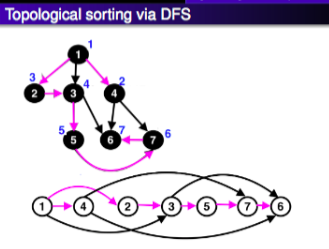


**Cycle detection**

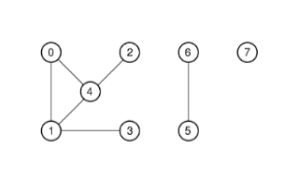
* Suppose that there is a cycle in G and let v be the node in the cycle visited first by DFS. If (u, v) is an arc in the cycle then it must be a back arc.
* Conversely if there is a back arc, we must have a cycle.
* Suppose that DFS is run on a digraph G. Then G is acyclic if and only if G does not contain a back arc.
* An acyclic digraph is called directed acyclic graph(DAG)

**Topological sorting**

* A topological sorting of a digraph G is an ordering on its vertices such that, for each arc (u, v) of, u appears before v in the ordering.
* To place nodes of a digraph on a line so all arcs go in one direction. Possibly if and only if digraph is a DAG
* Main application: scheduling events
* List of finishing times for depth-first search, in reverse order, solves the problem (since there are no back arcs, each node finishes before anything pointing to it ).
* Another solution: zero in-degree sorting. Find node of in-degree zero, delete it and repeat until all nodes listed.



**Graph connectivity**

* A graph G is connected if for each pair of vertices u, v are an element of V(G) there is a path between them.
* A graph G is disconnected if it is not connected and the maximum induced connected subgraphs are called the components of G.
* Connected components shown below:

# LECTURE 9: Introduction to Graph Algorithms

**Nice DFS application: strong components**

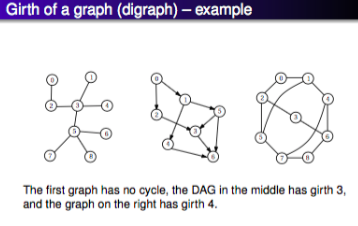
* Nodes v and w and mutually reachable if there is a directed path from v to w and a directed path from w to v. The nodes of a digraph divide up into disjoint subsets of mutually reachable nodes, which induce strong components
* A digraph is up if it has only one strong component.
* Components of a graph are found easily by BFS or DFS. However, this doesn’t work well for digraphs.

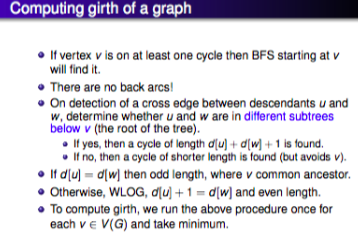
**Strong components algorithm**

* Let G be a digraph, and let Gr, be the reverse digraph of G obtained by reversing all arcs in G.
* Phase 1: Run DFS on G, to get depth-first forest Ft. For each node w in Gr, let done[w] be the time at which “w” turned from grey to black.
* Phase 2: Run DFS on G; when choosing a new root, choose a white node “w” such that done[w] is as large as possible. This gives a forest F.

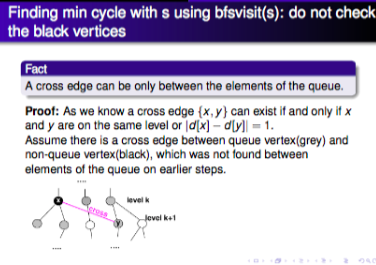
**Girth of a graph (digraph)**

For a graph (with a cycle), the length of the shortest cycle is called the girth of the graph. If the graph has no cycles then the girth is undefined but may be viewed as infinity. For a digraph, we use the term girth for its underlying graph and the term directed graph for the length of the smallest directed cycle.



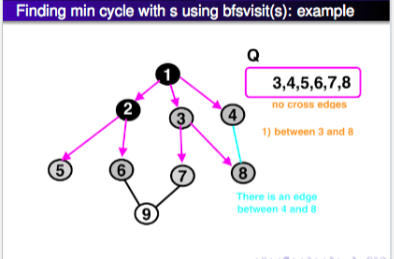


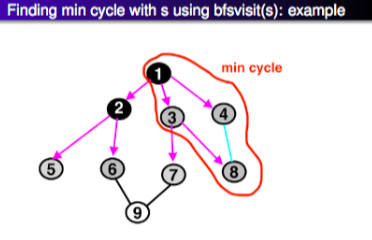
# ../Desktop/Screen%20Shot%202018-06-05%20at%203.33.36%20PM.pngLECTURE 10: Introduction to Graph Algorithms

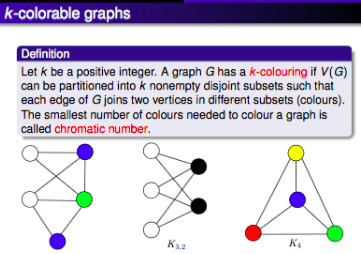


The purple edge is not a tree arc. Hence:

* Case 1: y was coloured grey before x turned black. Thus {x, y} was supposed to be found while x was still in the queue on earlier steps. A contradiction.
* Case 2: y was coloured grey after x turned black. It is not possible as y is adjacent to x, hence, y must be coloured grey before turning x to black.







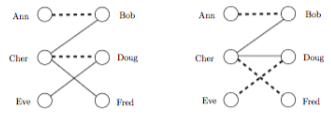
**Bipartite graphs (digraphs)**

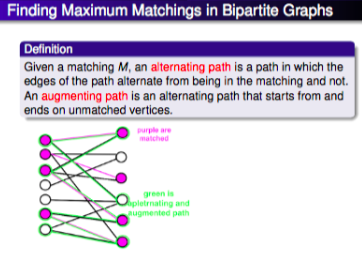
* A graph G is bipartite if V(G) can be partitioned into two nonempty disjoint subsets V1, V2 such that each edge of G has one endpoint in V1 and on in V2
* The following conditions on a graph G are equivalent.
  + G has a 2-colouring
  + G is bipartite
  + G does not contain an odd length cycle

# LECTURE 11: Introduction to Graph Algorithms

**Maximum matchings in graphs**

* A **matching** in a graph is a set of pairwise non-adjacent edges (that is, each vertex can be in at most one edge of the matching)
* A **maximal matching** is a matching such that is not a proper subset of any other matching, cannot add edges to this matching anymore.
* A **maximum matching** is one with the largest possible number of edges (over all possible matchings).
* A **perfect matching** is a matching in which each vertex is matched. Not every graph has such a matching.

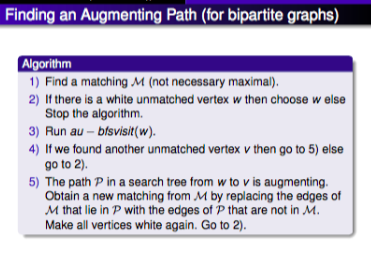


A maximal (left) and maximum (right) matching in a bipartite graph. Use a greedy approach to compute a maximal matching.

**Finding an Augmenting Path: au-BFSvisit**

We have a matching M.

* S is coloured grey, s is made the top of Q, and d[s] = 0
* Choose the top of Q, which is a grey node u.
* If d[u] is even then go to (4) else (in case d[u] is odd) go to (5)
* If u has a white neighbour v such that u, v is unmatched, that is u, v is not an element of M, then colour v grey and pred[v] = u, d[v] = d[u] + 1, add v at the end of Q, go to (4), deal with neighbours of a vertex from even level.
* If u has a white neighbour v such that u, v is matched, that is u, v is an element of M, then colour v grey and pred[v] = u. d[v] = d[u] + 1, add v at the end of Q, go to (5)
* Colour u black and remove u from Q.
* If Q is not empty go to (2)

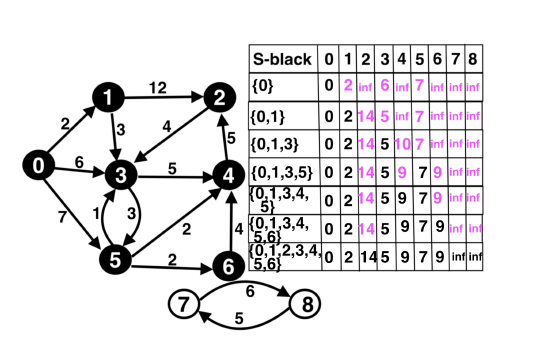


**YouTube more about this!**

# LECTURE 12: Introduction to Graph Algorithms

**Dijkstra’s algorithm**

1. Colour the source black
2. For every adjacent to source node/vertex v dist[v] is the cost c [source v].
3. Choose a white node/vertex u with min value of dist
4. Colour u black
5. For every adjacent to u white x do
   1. dist[x] = mindist[x], dist[u] + c[u, x]
6. If there is a white node/vertex adjacent to black then go to (3) else stop



**Why Dijkstra’s algorithm works?**

Let an S-path be a path starting at node s and ending at node w with all nodes coloured black except possibly w.

Suppose that all weights are non-negative. At the top of while loop, these properties hold for all w is an element of V(G):

* dist[w] is the minimum weight of an S-path to w.
  + This guarantees that, in each iteration of the algorithm, we find a minimum weight path from s to a vertex w ∈V(G) that only uses vertices in S (black vertices) except possibly w.
* if colour[w] = BLACK, dist[w] is the minimum weight.
  + This guarantees that, for each black vertex w, we have already found a minimum weight path from s to w.
* Taken together, these two imply that Dijkstra’s algorithm solves the single-source shortest path problem for weighted digraphs/graphs that do not have any negative arc/edge weights.

# LECTURE 13: Introduction to Graph Algorithms

**Dijkstra’s algorithm**

Dijkstra’s algorithm is PFS with priority function dist. The priority function is updated during PFS run. We check all non-black out neighbours of a grey v. If an out-neighbour of v is white then do the usual things. If an out-neighbour of v is grey then work only with a priority function (array dist in this case).

**Bellman-Ford algorithm**

We are still looking for the min distance from source node to all other nodes. Note, this algorithm works for graphs!

The Bellman-Ford algorithm is an algorithm that computes shortest paths from a single source vertex to all other vertices in a weighted digraph. It is slower than Dijkstra’s algorithm for the same problem, but more versatile, as it is capable of handling graphs in which some of the edge weights are negative numbers. Can’t solve negative cycles only negative numbers.

**Check YouTube for this example!**

# LECTURE 14: Introduction to Graph Algorithms

**Floyd-Warshall algorithm**

* Finding shortest paths in a weighted graph with positive or negative edge weights (but with no negative cycles).
* A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs and vertices.
* Although it does not return details of the path themselves, it is possible to reconstruct the paths with simple modifications to the algorithm.

# LECTURE 15: Introduction to Graph Algorithms

**Minimum spanning tree problem**

* Given a connected weighted graph, find a spanning tree (subgraph containing all vertices that is a tree) of minimum total weight.
* Total efficient greedy algorithms presented here: Prim’s and Krushkal’s
* Each selects edge in order of increasing weight but avoids creating a cycle
* Prim maintains a tree at each stage that grows to span; Krushkal’s maintains a forest whose trees coalesce into one spanning tree.
* Prim implementation very like Dijkstra, runs in time O (m + nlogn); Kruskal can be implemented to run in time O(mlogn)

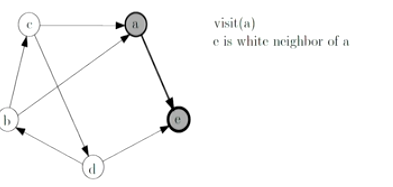
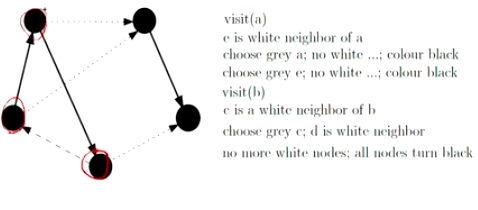
A spanning tree is a subset of graph G, which has all the vertices covered with minimum possible number of edges. In a weighted graph, a minimum spanning tree is a spanning tree that has minimum weight than all other spanning trees of the same graph.

* The most popular algorithms for minimum spanning tree are:
  + Kruskal’s Algorithm: Kruskal’s algorithm can be shown to run in O (E log E) time, or equivalent, O (E log V) time, where E is the number of edges in the graph and V is the number of vertices, all with simple data structures. This algorithm performs better in typical situations because it uses simpler data structures.
  + Prim’s Algorithm: The time complexity is O (VlogV is ElogV) = O(ElogV), making it the same as Kruskal’s algorithm. This algorithm is significantly faster in the limit when you’ve got a dense graph with many more edges than vertices.

# All Algorithms tested

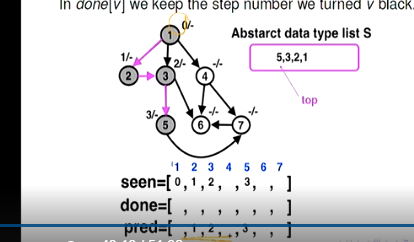
**General tree traversal**

* If we start at node “a”, if you run visit(a) it will go to the node “e” and colour it in grey.
* Since “e” has no white out neighbours, we return to “a”
* “a” has no white out neighbours so we colour “a” black. We traverse to “e” and colour it black as well. We are now done with visit(a)
* We now run visit (b) colour b grey then go to (c), then colour that in grey as well.
* From (c) we move to (d) and colour it in grey.
* Finally, we colour (d), (b) and (c) in black.
* The dark lines at the end will be the search forest whereas the light lines are not needed. The dark lines represent the nodes we have made in grey by traversing.



**Depth-first-search**

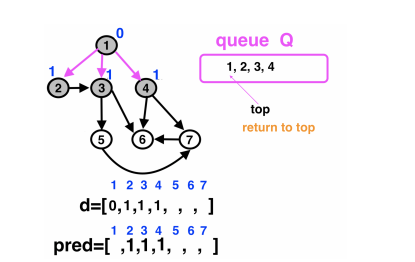
* We have a abstract data type, first array is seen second is done.
* We start with node 1 and make it grey, so we add it to our list. Add 1 to the 0 step on the seen list.
* Now we choose a white out neighbour (not specified which neighbour to choose), so we choose 2, then 3.
* Add them both to the list and also to the seen list.

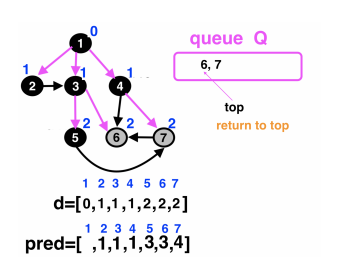


* Once we get to node (6) and is at the top of the list, we are now done since there are no more white out neighbours. So, we turn it black and remove it from the list and add it to done.
* Node 6 will be completed on the 6th step so we write 6 on the done list under 6.
* Node 7 will be completed on the 7th step so we write 7 on the done list under 7.
* Node 5 will be completed on the 8th step so we write 8 on the done list under 5.
* Etc, etc.

**Breadth-first-search**

* We have a queue. Start from d (1) which is 0. Then traverse to the white out neighbour which isn’t specified. So, we go to any of them.
* We go to (2), it is on the first level and add 1 to the 2 on the distance array. The node 2 will be added to the queue at the back.
* 2 has a white out neighbour which is 3, since it’s on level 1 we add that to the distance array, and then add 3 to the queue. Since the top of the queue is 1 we check from there again to see if there are any white out neighbours and notice we have 4.
* 4 is added to the queue and 1 is added to the distance.
* Since 1 has no more white out neighbours, so we colour it black and remove it from the queue. The new top element is now 2.
* 2 doesn’t have any white out neighbours so we colour that black and then remove it from the queue and look at 3.
* 3 has a whiteout neighbour which is 5 so we colour it in grey and add it to the queue.
  + 5 is on the second level so we add 2 to the distance array





**Priority-first-search**

* We start with an array Q, we start with [[1,0]] 0 being our minimum key and 1 being the node we are on.
* Then we go to node 4 and increment the key by 1. So, we will have [[4,1]]
* We can choose the order by random but make sure we are consistently choosing random.
* Since the minimum key is 0 we return to the node with key 0, and choose a white out neighbour which can be 3.
  + Then we add [3,2] to the array then we go back to the minimum key which is still 0.
* Final white out neighbour is 2, we traverse there and add it to the array so [2,3]
* Since minimum key is still 0 we travel to node 1 and check for neighbours, since we have no white out neighbours, so we pop out the node 1 and remove it from the array.
* We do the same with node 4 since it has the minimum key of 1 we go there and look for white out neighbour, since we have none we turn it black and remove it from the array.
* Etc