Data Structures And Algorithms

CSC2001F

Data Structures

Flashcards: https://quizlet.com/_7zu8eq?x=1jqt&i=2g4hog

Abbreviations:

1. Str = structure
2. Obj = object
3. Ops = operations
4. Algo = algorithm
5. Calcs = calculations
6. LHS = left hand side
7. RHS = right hand side

- Example: Facebook stores "likes" – need to store it somewhere it can be quickly accessed on demand; need a mechanism to do so.
    - i.e mechanism which allows for large amounts of data with fast access
- This can be stored in a data str.
- Data structure: efficient mechanism to store data and provide access operations.
- Stored (advantages):
    - in-memory: faster VS
    - in-disk: larger storage, non-volatile
- The "conceptual" view:
    - A data str is a "black box" – data in and data out.
    - You can perform ops on this box

DATA ITEM

- Can be many things e.g. obj instance.
    - Need to contain at *least* one data value
    - Need to contain at *least* one "key"
        - Key: a distinct thing about a data item that allows for differentiation between data items.
            - E.g. ID #
            - Used for data str obj ops
    - Contains ops to manage data item.

DATA STRUCTURES OPERATIONS

- Can include *create* (initialise empty data str); *traverse* (perform an op on all data items); *merge* (make multiple DSes into one) and so on

Algorithm Analysis

- Determine speed of algorithm and memory needed by algorithm.
- Amount of time it takes algo to run depends on amount of input it must process
    - Thus, running time is function of input size.
    - Running time varies due to factors like host machine speed, compiler quality and program quality.

- Empirical analysis: measuring resource usage of algorithm experimentally.
- Asymptotic analysis: determining bounds on resources needed as size of data increases
    - Upper bound: think of it as overestimating the amount of processing power needed
    - Lower bound: not often used as it can be exceeded.

TIME CALCULATIONS

- T(n) = time for an operation
    - Use an "expensive" op e.g. comparison
- Time calculations rely on a relationship between hardware and software
- Linear algo: time is directly proportional to amount of input
    - Most efficient algo
- Non-linear algos often lead to large running times.

TIME COMPLEXITY

- The amount of time an algorithm takes to solve a problem expressed as a function of the problem's size.
- Some ops have the same running time i.e. constant time.
- Decisions affect number of operations – affects time complexity
- If algo isn't constant time, increasing resources WON'T increase efficiency
- Worst case:
    - Highest possible time
- Average case:
    - Average of time across all values of n
    - DON'T assume average behaviour
- Best case:
    - Lowest possible time.

DOMINANT TERMS

- The operation that will have the most effect and minimise the rest
- Dominant term will affect time complexity
- By means of an example:
    - E.g. $f(x) = x^3 + 2x$ -----> as x gets large, x^3 dominates the function.
      ==> f(x) is order x^3
      ==> $f(x) = O(x^3)$

- We measure a function's rate of growth.
    - We have some value n.
        - For a large enough n, the value of a function is largely determined by the dominant term – "large enough" is a description that'll vary from function to function.
    - Function's growth should ideally be less than n.
- Big-Oh notation – places dominant term in a function and represents the growth rate
    - Show's computational complexity of algorithm i.e order of a function
    - Used to –
        - order functions by comparing dominant terms e.g. O(nlogn) vs O(n).

        - see how changes in number of data items (inputs) affects how many units of computation are performed for very large n.
- Small input amounts make comparisons between functions difficult – leading constants become very significant.

ASYMPTOTIC FUNCTIONS

- f(x) < g(x) ----> fundamental basis
- f(x) = O(g(x))
    - f(x) is bounded by multiple of g(x), for large x
    - |f(x)| <= M|g(x)| for all x > x0
    - Big O
- F(x) = o (g(x))
    - Small o notation
    - Tighter bound
    - For all M, there is an x0.
- Everything that satisfies small o (2nd condition), satisfies big o (1st condition) but NOT vice versa.

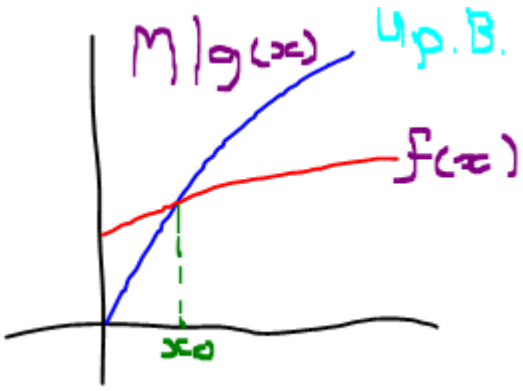


Figure 1: graph of asymptotic behaviour. Up B = Upper Bound

ALGORITHM TRADEOFFS

- Time-space:
    - Using extra memory to make ops faster
        - E.g. pre-computing values and saving it to a database.
    - Using slower ops to reduce space requirements
        - E.g. unindexed database
- Time-time: use time of one op to make another faster
    - i.e. use time upfront to make later calcs faster
- Single vs double linked lists: not much diff bet two in terms of operation time complexity but doubly is O(1) for delete (single is O(1)/O(n))

Trees

Abbreviations

1. Bin tree = binary tree
2. --> means “implies”
3. WC = worst case
4. BC = best case
5. AC = average case

Note to self: insert diagrams and code for written notes

- Tree: simplest example of non-linear data struc.
    - Use? Some operations are more efficient – faster search, organising files etc.
    - Applications – e.g. used in directory structure in OSes, like Unix.
- Properties:
    - Set of nodes and set of directed edges
    - Single root node – unique path from root to each node
    - Every node (except root) has *only* one incoming link and zero/more outgoing links.
    - No cycles
- Have parent and child nodes – edge connects parent to child.
- Leaf: node that has no children.
- Siblings: nodes with the same parent
- Path length: # of edges that must be followed.
- Node size: # of descendants node has – including node itself.
- Height of tree: max distance frm root to node in tree.
    - Zero height ==> single node
    - Tree with no nodes has height val -1
    - Calculated using height of subtrees
    - Can only calc height given root node.

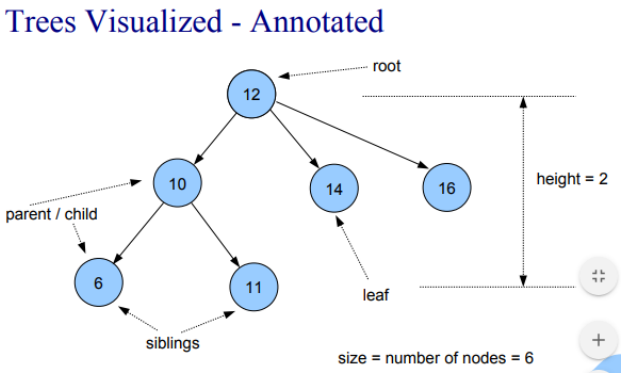


Figure 2: Trees

GENERICS DATA STRUCTURE

- Data struct itself is indep
    - i.e. separation between data and data struc

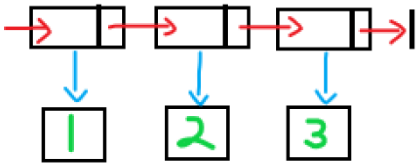


Figure 3: A visual representation of a data structure.

- Useful because if you're manipulating data struc (say, for example the data is stored in a node in the data struc), may not want to manipulate/change data too (as doing so can be costly)

Binary trees

- Trees with 2 potential links:
    - i.e. 0-2 children possible per node
    - Children called left and right.
- Construction of bin. Tree – need root (i.e. thing that points to key/main element of bin tree)
    - If it has root, can get to every other element.
- Recursive thinking: bin tree either is empty or has a root, left and right tree.

OPERATIONS

- Traversals:
    - PreOrder
    - InOrder
    - PostOrder
    - LevelOrder
- Calculating:
    - Height
    - Size

TRAVERSAL

- The Visitor Pattern: abstract away function you want to perform with every item into a "visitor method".
    - Makes it easier in future to change method
    - E.g if you want to iterate over every item and print, define visitor method that will print item – then if necessary, in future, only need to change visitor method
- In all cases of traversal, we start at the root node, and we use a stack data struc.
- PreOrder: visit node before children
    - Processes one subtree at a time.
    - Node has been "visited" when it is popped off stack for 1st time
    - Before returning, iterator pushes right child, then left child onto stack.
        - Why? Want left child to be processed before right.

- PostOrder: visit node after children

IN-DEPTH PROCESS OF POSTORDER TRAVERSAL

- Not examinable
- PostOrder traversal implemented using stack to store current state.
- Initialise algo by pushing reference to root node onto stack.
- Top of stack = node you're visiting at a step in the traversal.
    - Can be in one of 3 steps:
        1. Make recursive call to left subtree
        2. Make recursive call to right subtree
        3. Process current node
- Means each node is placed on stack 3 times during course of traversal.
- If node popped 3rd time --> mark it as current node to be visited.
- Else, if node is being popped for 1st/2nd time, means it's not ready to be visited, so push back onto stack and simulate recursive call.
    - If node popped for 1st time, push left child (if it exists) onto stack.
    - Otherwise, if node popped for 2nd time, push right child (if it exists) onto stack.
- Pop the stack and do same test.
    - By popping, simulating recursive call to correct child.
    - If child doesn't exist, then it was never pushed onto stack and so when the stack gets popped, pop the original node again.
- Process result with popping node for 3rd time/ stack empties.
    - Empty stack --> iterated over entire tree.

- InOrder: visit node after left child but before right child.
    - Useful for printing things in left to right order.
- LevelOrder:
    - Level: nodes that are same distance away frm root.
    - Can't use same recursive technique as for other traversals.
    - Need to use queue data strc
        - Queues stores nodes that are yet to be visited.

          1. Add root to queue

          While queue is not empty:

          2. Check if node has children (pop a node i.e. go to front of queue and visit it), if

          Yes, add children to queue.

COMPLEXITY ANALYSIS

- Best Case = Worst Case = Avg case = O(n)
    - Applies to height, size and traversal ops.
- Lvl order space requirements:
    - At most, can have all elements on one lvl i.e. all elements in queue at one time.
    - Need n/2 space for lvl order traversal, i.e. max amount of elements on one lvl.

Binary Search Trees

- Bin search tree is alternative to linked list – extends bin search algo to allow for insertion and deletion
- Type of binary tree with extra properties – nodes are ordered.
    - i.e. nodes have vals which can be used to compare them
    - condition has to apply to every node of the tree – not just root.
    - All nodes in left subtree are < node.
    - All nodes in right subtree are > node.
        - Can also define one side as equal
- Pros: faster search
- Cons: more work for adding and removing nodes
- Use: want to be able to have a data strc such that, if changes need to be made to the data, it can be done while the program is executing.
- Basics:
    - Search for item using its key.
    - Bin search tree satisfies *search order property*: for every node X in tree, values of all keys in left subtree are smaller than key in X and all values of all keys in right subtree are larger than X.
        - Property implies that all items in tree can be ordered consistently.
        - InOrder traversal yields all items in order
        - Property also doesn't allow duplicate items.

Basic Operations

- Every bin search tree is a binary tree BUT not every binary tree is a bin search tree.
    - Implies can use bin tree ops on a bin search tree.
- Necessary tools: findmin method, removemin method, findmax method
- Insert:
    - If root is null, new node becomes entire tree
    - Add node recursively
        - If val<node, add new node to left.
        - If val>node, add new node to right.
        - Recurse until node gets added as new leaf
    - Every node that gets inserted *only* gets inserted as leaf

{Insert code}
INSERTION DIAG

    - Have 2 base cases for recursive algo – insert node for both L and R
    - Code key: {(d,null,null)} - have nulls because added as leaves.
- Find:
    - Make comparisons at each node
    - Start at root
    - If search == root
        - Result is root elem
    - If search<root

- Recurse into left subtree
- If search<root
  - Recurse into right subtree
- With duplicates, no guarantee which of dups you'll find
  - ==> avoid inserting dups for this reason.

{Find code}
FIND DIAG

- Remove:
  - Find elem to del
  - If leaf, just del
  - If 1 child, replace node with child
  - If 2 children:
    - Find X = min node in right subtree
      - Alternatively, can find max left subtree.
    - Remove X frm right/ leftsubtree
    - Replace deltd elem with X
  - {remove code}
  - *findmin* method: while left node not null, follow chain (of nodes) while there's a left hand side pointer.
  - *removemin* method: using recursion, navigate down, updating left hand side pointers
    - Ensures tree can be reconnected again.

{Remove code}

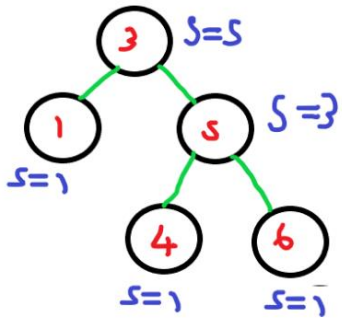
DUPLICATE KEYS

- Assume all keys unique
- Many issues:
  - Is it identical item copies OR
  - Different items with identical keys
- If you store dup normally in tree, no guarantee as to which one will you get when you perform operations on the tree.
- Options:
  - Put dups in adjacent tree positions
    - Must modify all ops to handle this
  - Counter for dups in each node
  - Create a list of dup items and attach to each node (Hussein recommends)

ORDER STATISTICS (FIND KTH VALUE)

- Maintain size of subtree at each node
- Every func that changes tree needs to update these sizes – recalculate height and other sizes.
- Putting precalculated size into every node.
  - Because of precalculated size, can efficiently implement certain algos.
- Algo:
  - SL = left subtree size

- SR = right subtree size
- If K= SL +1, root node is answer
- If K<SL+1, find K(K) in left subtree
- If K>SL+1, find K(K-SL-1) in right subtree



If k = 2:

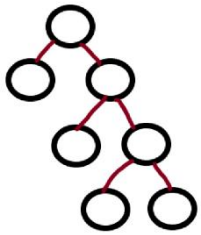
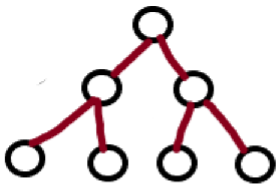
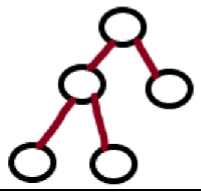
2 = SL +1
= 1+1 == k is root node

Figure 4: board example

Complexity Analysis

- Assume tree with n items; counting comparisons
- Best case: O(1) - search, insert and del
- Worst case: O(n) - i.e. linear config of nodes (all nodes in "one line")
- Number of ops is related to height of tree and height is related to $\log_2 n$
  - Log is bound of complexity analysis for all ops performed on bin search tree.
- Many algos can be made as a combination of other algos e.g. Update = del + insert.

BINARY SEARCH TREE ANALYSIS

Full tree: no single child nodes ➔no linear trees	
Complete tree: all levels have every node except last, and last is filled from left. ➔i.e. every lvl full and filled left to right ➔complete tree has minimum height which implies faster performance of trees	 

Balanced tree: nodes distributed according to a balancing criteria ➔ balancing tree guarantees O(log n) performance.	

PROOF BY CONTRADICTION: MINIMAL HEIGHT FOR COMPLETE TREE

- Assume we have a complete tree.
- Suppose it is possible to reduce the height.
- This implies that it is possible to move one or more nodes at the deepest level higher up the tree, to reduce its height.
- But this would imply that the tree is not complete.
- This is a contradiction.
- Our original assumption is incorrect. It is not possible to reduce the height.

HEIGHTS AND SIZES OF TREE

- Height and size are related:
  - Height (max # edges from root to leaf) = h
  - Size (number of nodes) = s
  - Level (number of rows) = l = h+1
- Max height:
  - Related to O(n) func
  - Nodes are linear
  - Bound: h<= s-1
- Min height (ie have a complete tree):
  - Related to O(log n) func
  - Bound: h>= [log(s+1)] -1 – here [] implies “ceiling of’
- Min # nodes:
  - Linear tree
  - s>=h+1
- Max # nodes:
  - Assume full and complete
  - $s<=2^{h+1}-1$

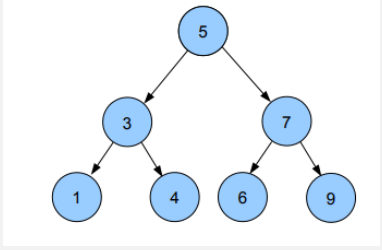
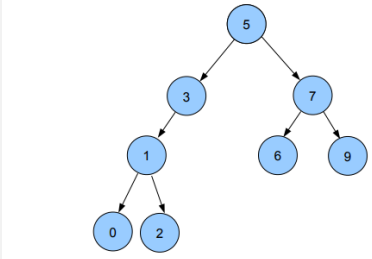
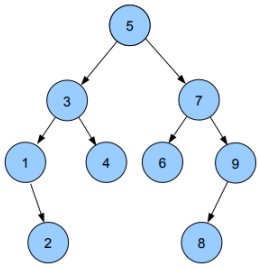
AVL

Abbreviations

1. BST = binary search tree
2. BT = binary tree
3. Bal = balance
4. Bals = balances
5. Balnd = balanced

- BST with additional balance properties:
  - Every node balanced (in AVL terms)
    - Height of left and right sub-trees differ by at most 1
  - Every node must meet this balance condition ie at every node, balance condition must hold.
    - Applies to all nodes in tree because each node is itself a root of some subtree.
  - Height of empty subtree is –1.

- All complete trees are AVL.
- Advantages:
  - Balanced tree in WC
  - Faster ops
- Disadvantages:
  - More work for adding/removing nodes
- Containment relationship
  - Inheritance of algos - every AVL tree is BST is BS is T
- Why balance trees?
  - Guarantee of performance ie clear bounds so that tree doesn't "degenerate" into linear time complexity
  - Balance condition ensures depth of tree is logarithmic

AVL	NOT AVL
	
	

Basic Operations

- Insertion/deletion can break the balance of several tree nodes.
    - Balance has to be restored before opn can be considered complete.
- Key idea: after an insertion, only nodes that are on the path from the insertion point to the root might have their balances altered.
    - As you traverse the path up to the root and update balancing info, may find a node whose new balance violates AVL condition.
- Auxiliary routines
    - Manage height
        - Precalculate it
        - Fix it after changes
    - Rebalance
    - Rotate left/right
- Additional info:
    - Maintain height at each subtree
        - Use instance variable in node
    - Determine heights between left and right sub-trees
        - Balance factor
    - {height code slide 9}
    - {balanceFactor and fixHeight code slide10}
- Insert algo:
    - Use same BST insertion algo
    - Rebalance all nodes potentially affected
        - Apply to all nodes from insertion pt to root.
        - Use tree rotations at each node as necessary

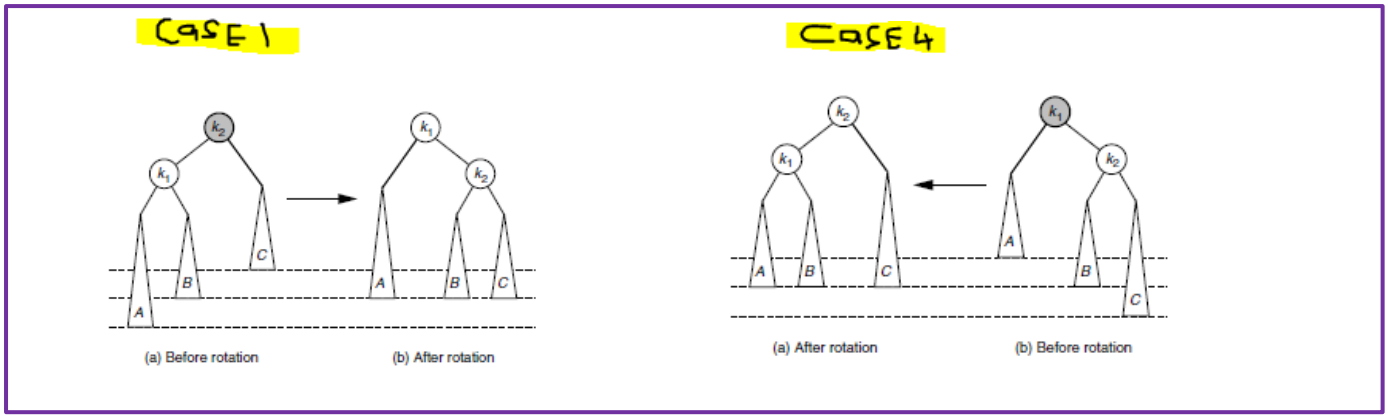
Tree Rotations

- Can be root/lower node that changes – find which pt conditions has been violated and perform rotation there.
- 4 cases for height imbalance – ie height of two subtrees (with parent X) differs by 2.
    1. Insertion in left subtree of left child of X
    2. Insertion in right subtree of left child of X
    3. Insertion in left subtree of right child of X
    4. Insertion in right subtree of right child of x
        - Cases 1 and 4 are mirror images of each other; cases 2 and 3 too.
- Balance is restored by tree rotations

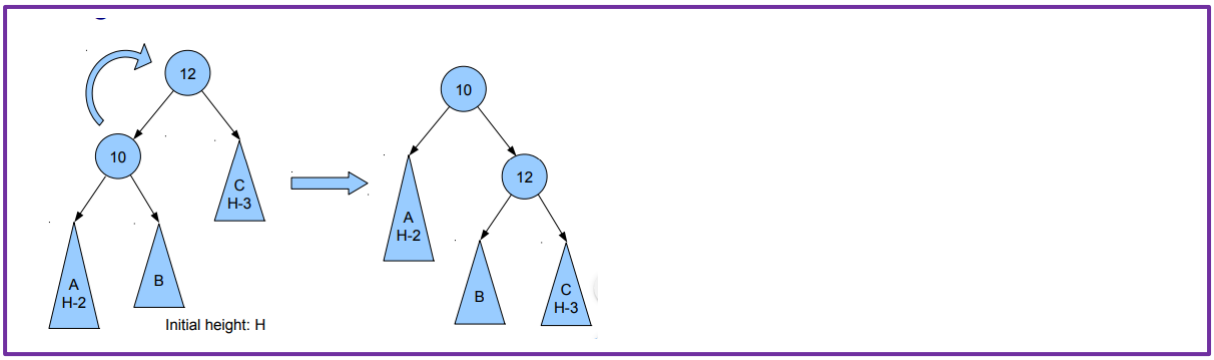
SINGLE ROTATIONS

- Switches roles of parent and child while maintaining search order

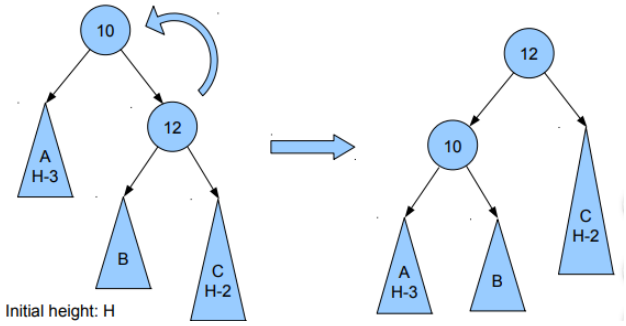
- Insertion occurs on the outside ie right subtree-right child, left subtree- left child.



- Insertion into left subtree of left child
  - rotateWithLeftChild – rotate right
  - {rotateRight code slide 16}

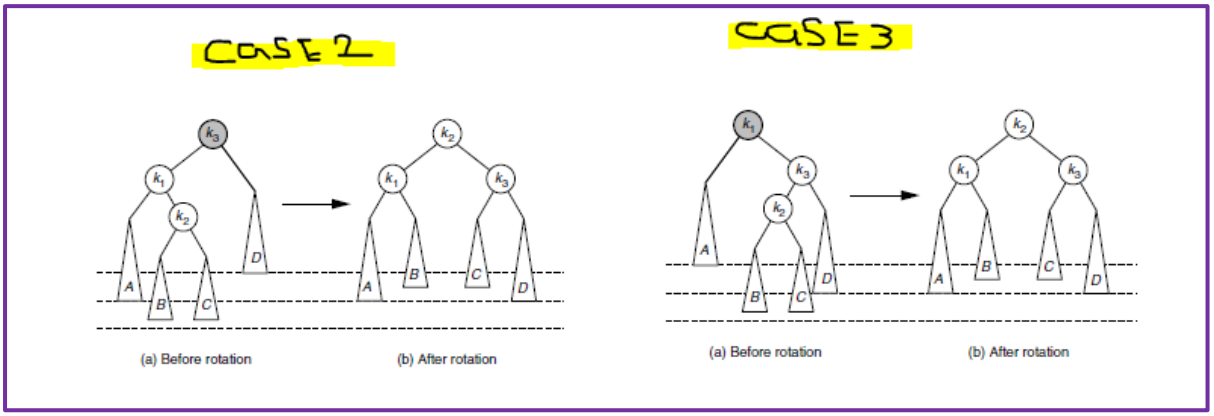


- Insertion into right subtree of right child
  - rotateWithRightChild – rotate left
  - {rotateLeft code slide 17}

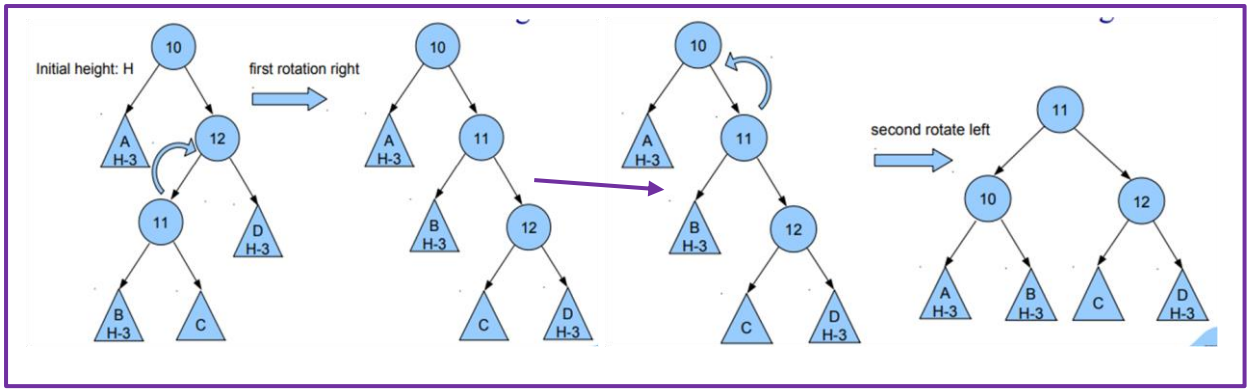


DOUBLE ROTATIONS

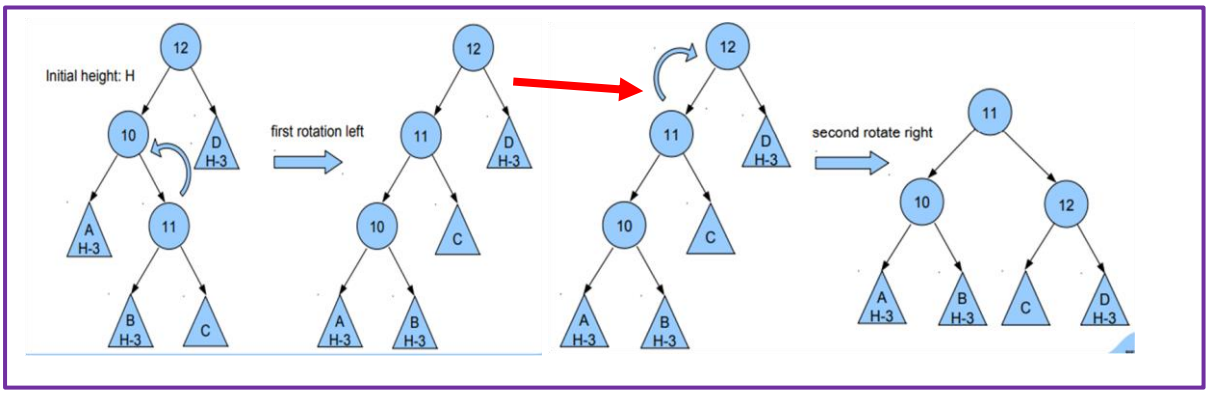
- Has the sequence:
  - Rotation between X's child and grandchild
  - Rotation between X and its new child.
- Insertion into right subtree of left child
- Insertion into left subtree of right child
- Double rotation is equivalent to 2 single rotations.



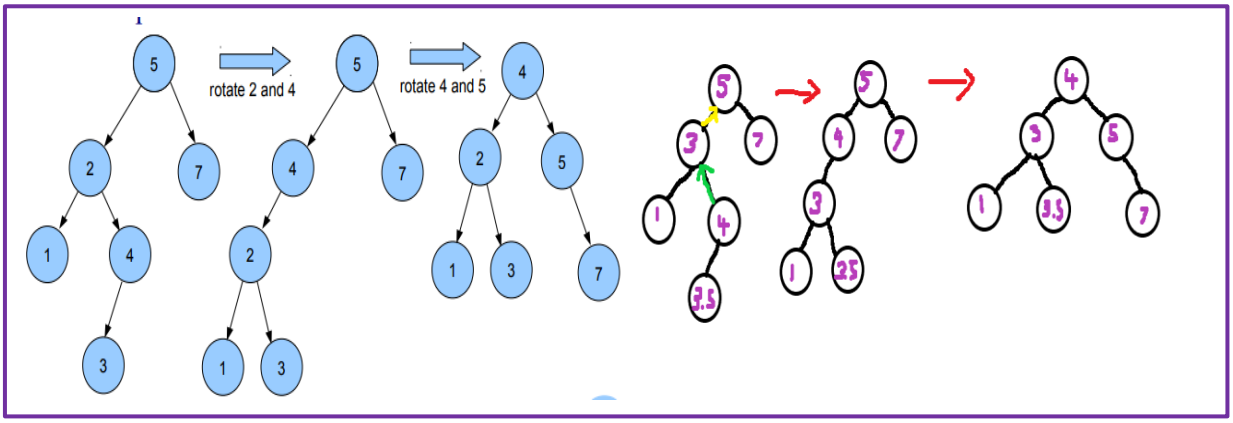
doubleRotateWithLeftChild



doubleRotateWithRightChild



- Examples:



- {balance code slide 23}
- {insert code slide 24}
- Delete algo:
  - Rebalance nodes all the way from node to root.
  - Rebalance nodes also when removing the min
  - {delete code 26 to 27}
  - {removeMin code slide 28}

Complexity analysis

- WC has search, insert and delete: O(log n)
- Max depth of n-item tree is O(log n)

Red-Black trees

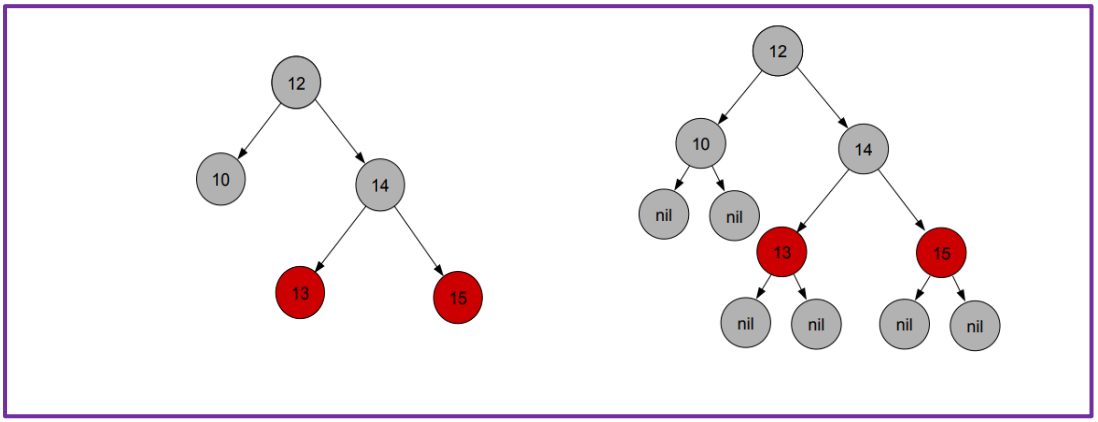
- BST with additional properties:
  - Every node has a colour: red/black
  - Every leaf (nil) is black and so is root node
  - If node is red, both its children are black
  - Every path from root to leaves contains the same # of black nodes.
- Single top down pass used during insertion and deletion routines.
  - Differs to AVL, as pass down the tree is used to establish the insertion pt and a second pass up the tree to updates heights and rebalance if necessary.
- Non-recursive implementation of red black tree is simpler and faster than an AVL tree implementation.

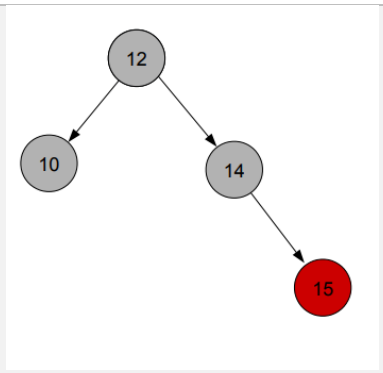
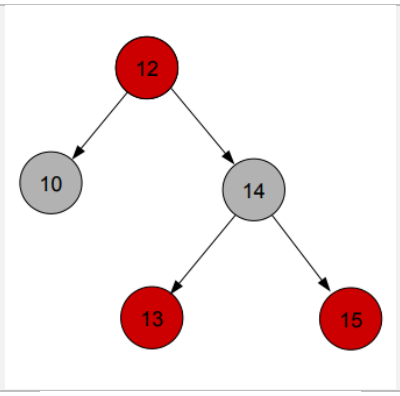
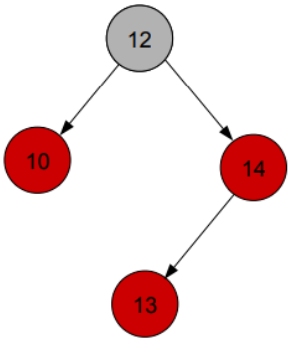
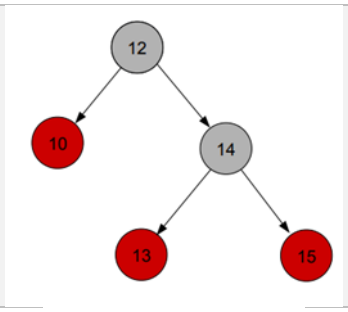
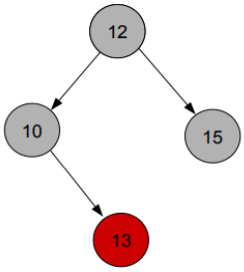
Analysis

- All ops : O(log n)
  - Can prove tree with n nodes has height O(log n)
- Similar to AVL but with colour changes and possibly less rotations
  - Weaker balance condition than avl
  - Same advantages and disadvantages
- Advantage:
  - Relatively low overhead required to perform insertion and that in practice, rotations occur relatively infrequently.
  -
- One bit per node to colour (vs heights for AVL)

Basics

- Assume every nil pter black
- Every inserted node is red
- No node and its parent can both be red
  - Violation of condition 3
  - Fix colours and rotate tree if necessary, to maintain properties 3 and 4
- Examples



RB	NOT RB
	
	
	
	

Basic operations

- Insert, delete
- Auxiliary ops:
  - Manage colours
    - Updates colours. Fix colours after changes
  - Single/double rotate left/right

- Maintain colour at each node
    - Use an instance variable in node
- Determine if any red-black conditions have been violated.

TOP-DOWN RBTS

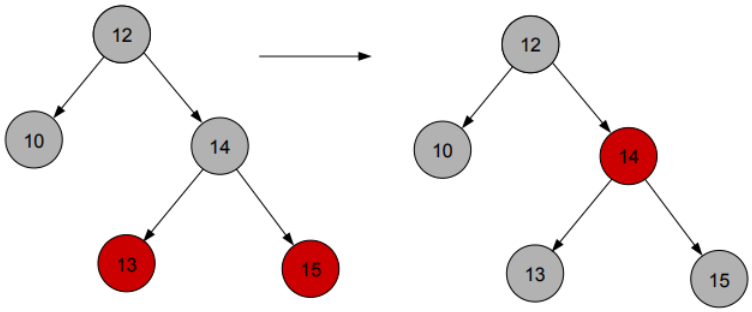
- Apply top-down procedure while looking for insertion pt
    - Avoids possibility of having to iterate up the tree.
    - While descending, ensure that the sibling's parent isn't red either with colour flips/rotations.
    - Guarantee that, when arriving at a leaf and insert node, S is not red.
    - Can just add leaf and if needed use one rotation (single/double).
- Procedure:
    - See a node A that has two red children – make A red and its two children black.
    - Then recolour A to black
    - If A's parent is red, that introduces 2 consecutive red nodes – can apply either single/double rotation.
    - What if A's parent's sibling is also red?
        - Impossible, since, if on the way down the tree, see that node B has 2 red children, then that means B's grandchildren must be black.
        - As B's children are also made black via a colour flip – even after rotation may occur, this means there won't be another red node for 2 lvls.
        - Hence, when encountering A, if A's parent is red, A's parent's siblings can't also be red.

figure 19.38

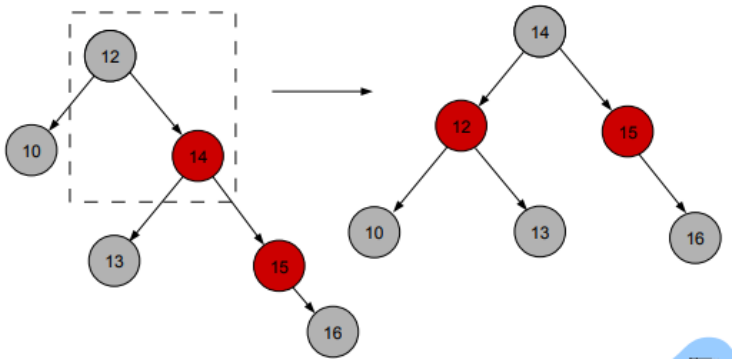
Color flip: Only if X's parent is red do we continue with a rotation.

INSERT (TOP DOWN)

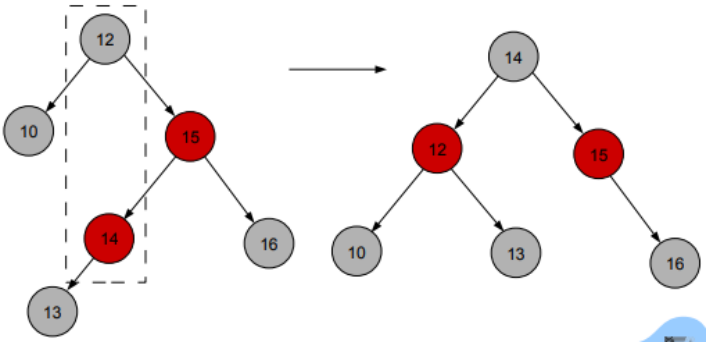
- Traverse down from root to find insertion pt
- At each node:
    - Step 1 - If node is black and both children are red, flip colours.



    - If node's parent is also red:
        - Step 2 - If outside node, do single rotation and recolour

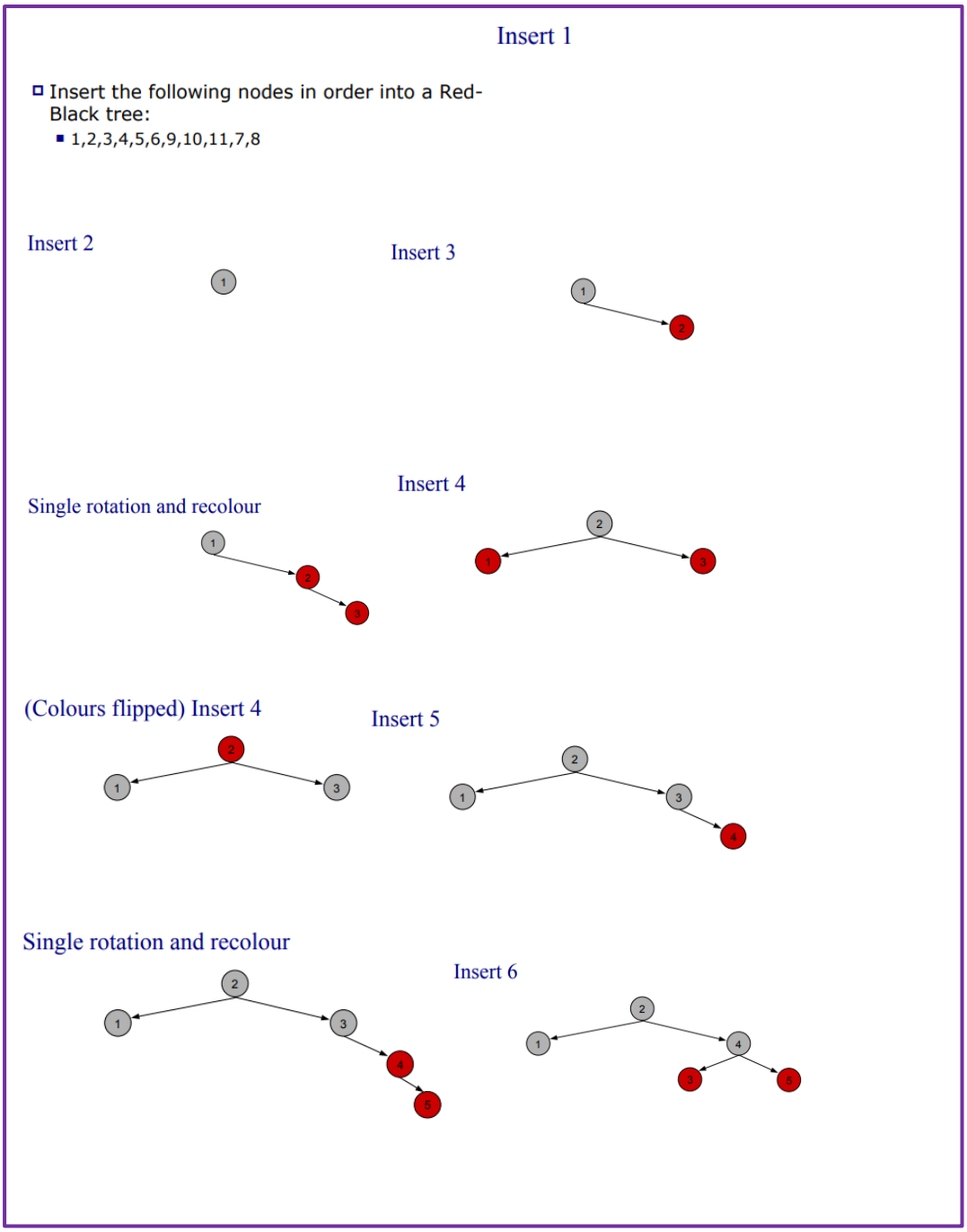


- Step 3 -If inside node, do double rotation and recolour

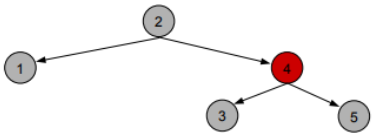


- Step 4 - After inserting node (as red), if its parent is also red, do a single/double rotation.
- Colour root black

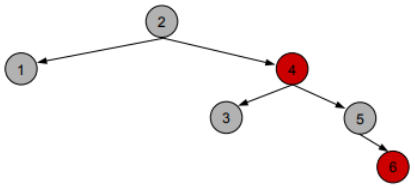
- Example:


Insert 1
Insert the following nodes in order into a Red-Black tree:
1,2,3,4,5,6,9,10,11,7,8
Insert 2
Insert 3
Insert 4
Single rotation and recolour
(Colours flipped) Insert 4
Insert 5
Single rotation and recolour
Insert 6

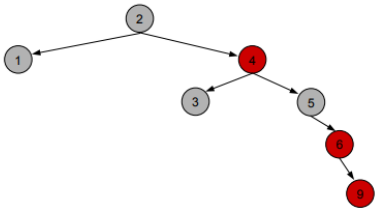
(Colours flipped) Insert 6



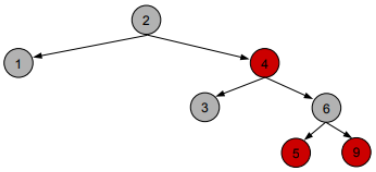
Insert 9



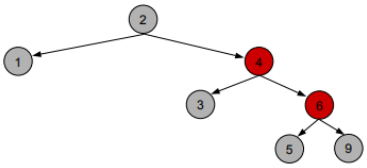
Single rotation and recolour



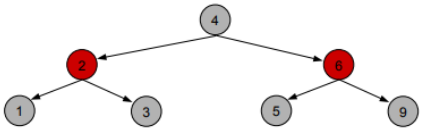
Insert 10



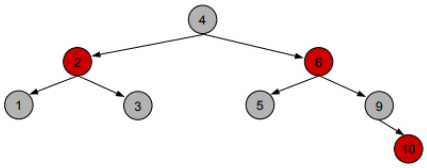
(Flipped colours) Insert 10



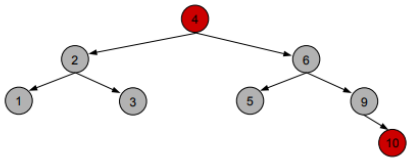
(Single rotation, recolour) Insert 10



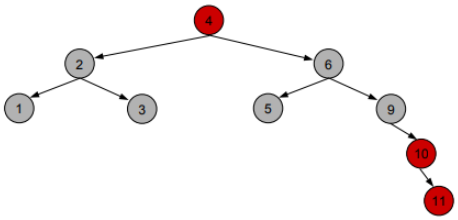
Insert 11



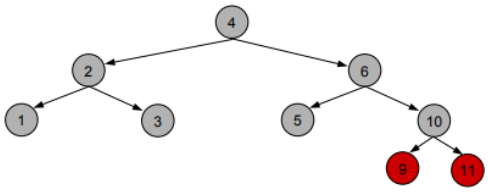
(Colours flipped) Insert 11



Single rotation, then recolour

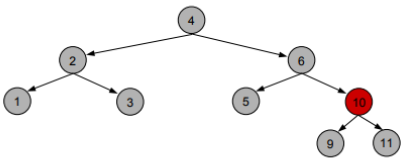


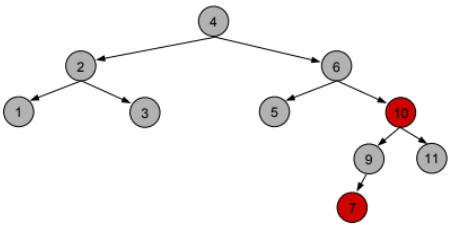
Insert 7

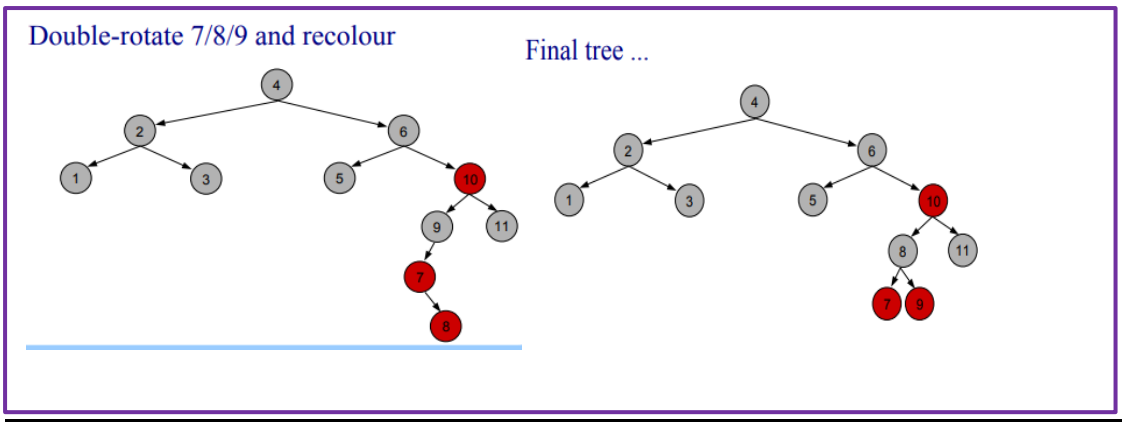


Insert 8

(Flipped colours) Insert 7






Double-rotate 7/8/9 and recolour
Final tree ...
4
2
6
1
3
5
10
9
11
7
8
4
2
6
1
3
5
10
8
11
7
9

Bottom-up insertion

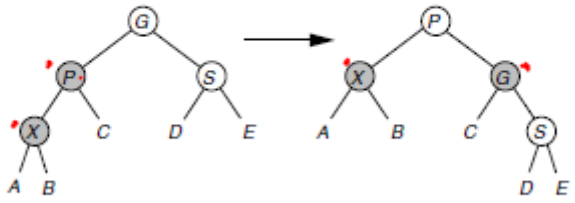
- New item is always inserted as leaf in a tree.
- So new item is inserted as red – if it was inserted as black, violates property 4 since it creates a longer path of black nodes.
- Basic opns: colour changes and tree rotations.

CASES FOR IF THE PARENT IS RED

*Shaded nodes are red

- Sibling of parent is black (S)
- X is newly added lead, P is its parent, S is sibling of parent (if it exists) and G is grandparent.
- Only X and P are red in this case; G is black.,
- G, relative to x, can be an inside/outside node,
- If X is outside grandchild, single rotation of its parent and grandparent, and some colour changes, restores property 3.

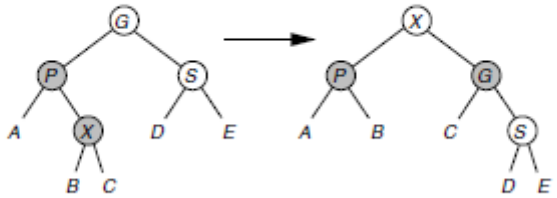
X=outside g



(a) Before rotation (b) After rotation

- If X is outside grandchild, double rotation and colour changes are needed.

X=inside g



(a) Before rotation (b) After rotation

- Why are these rotations correct?
  - Need to ensure never have two consecutive red nodes.
  - Only possible instances of consecutive red nodes would be between P and one of its children or between G and C
  - Roots of A, B and C must be black
- If S is red:
  - Insertion attempts will fail – neither single/double rotation works because both result in consecutive red nodes.
  - So, both S and the subtree's new root must be coloured red.
    - Induce consecutive red nodes at a higher level.
    - Need to iterate up the tree to fix things.

DELETE

- Deleted nodes have at most one child
  - 2 node deletion is actually replacement, so colours are unchanged
- Want to delete a red node
  - Will maintain black node path lengths
- Traverse tree from top to bottom
  - Ensure that current node is red
    - Otherwise, rotate and recolour without violating conditions.

Hash Tables

- Hash table: data strc where items are stored in a location determined by their content
  - Content-based index VS
  - Comparison-based index
- Want to quickly access arbitrary amounts of data in quick time.
  - To go beyond O(log n), forget about comparisons
- Every hash table is fundamentally composed of:
  - Array of items
  - Hash func: item -> index
- Issues with hash tables:
  - More keys than slots = overflow
  - Key isn't an integer = map
  - Keys are same but values are different = collision

Hash Function

- Hash function: a mapping from an item to an integer
  - One-way function: item to index ONLY.
  - Can cause collisions.
- Collision: two/more different items can # to the same position
  - Situation can never be avoided because there are many more items than positions
- Resolving a collision, can use:
    - Linear, quadratic probing
    - Chaining
  - Each method yields a different performance, depending on how full the array is.
- A hash function is NOT an array but an array part of the data strc.
- Many ways to go about creating one:
  - Hashing strings – can add together Unicode values and mod tablesize.
    - E.g. h("abc") = (97+98+99)% tableSize
    - Bad # func for larger tablesize and small keys
    - "abc" = "cab" ==> can cause collisions
  - Could build on previous function by multiplying/shifting each char by some value.
    - Increases # values
    - Solves uniqueness problem.
- A good # func:
  - Fast to compute
  - Deterministic (and consistent)
  - Spreads keys evenly in # table.
- Perfect # func:
  - Maps every distinct key onto a distinct integer
  - Minimal perfect # func has no holes on the array.

Collision Resolution

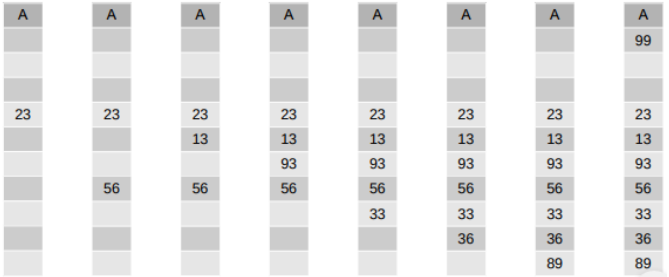
- Open Addressing:
  - Linear probing
  - Quadratic probing
- Closed addressing:
  - Chaining
- Load factor (λ): proportion of the table that is full
  - Load factor ranges from 0 (empty) to 1 (full)

Linear probing

- Searching sequentially in the array until empty cell found
  - Search wraps around from last position to first if necessary.
- So long as table is large enough, free cell can always be found.
  - Time needed to find free cell can be long.
- Insertion algo:
  - Generate hashcode h = hash(key)
  - While A[h] contains a key:
    - H = (h+1) % tableSize
    - A[h] = {key, value}
- Find algo:
  - Generate hashcode h = hash(key)
  - While A[h] contains a key:
    - If (A[h] {key} == key) return A[h] {value}
    - H = (h+1) % tableSize
    - Return Not Found
- Analysis of "find":
  - Two types of find operations: unsuccessful and successful
  - Cost of a successful search of X is equal to the cost of inserting X at the time X was inserted.
  - To find the average time to perform a successful search in a table with load factor λ, must compute the average insertion cost by averaging over all the load factors leading to λ

Linear Probing Example

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99



ISSUES

- Primary clustering: multiple adjacent items and slow performance
    - Effect of primary clustering - formation of large clusters of occupied cells, making insertions into the cluster expensive (and then the insertion makes the cluster even larger).
    - Primary clustering not only makes the average probe sequence longer, but it also makes a long probe sequence more likely
    - Main problem with primary clustering thus is that performance degrades severely for insertion at high load factors.
- Items that collide because of identical # funcs cause degenerate performance
- Also an item that collides with an alternative location for another item causes poor performance.
- Uneven gaps in table
- Table full
    - What to do when table full?
        - Create new and large table – items will need to be rehashed
- Gaps – very important; need to be 1 less than size of array; gap are the stopping condition for insert operations.

ANALYSIS

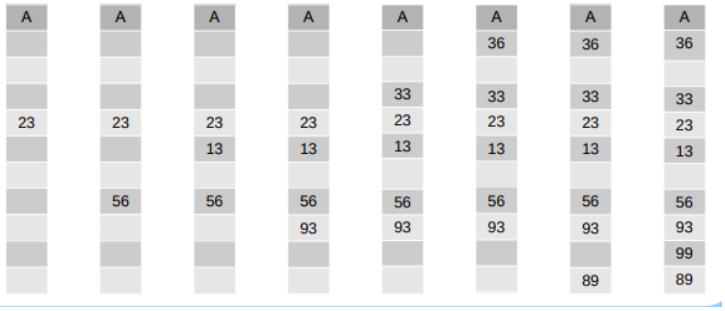
- Load factor (λ):
    - Proportion of empty table is (1-λ)
    - Probability of cell being empty is (1- λ)
- Average # of cells examined for insert (failed find):
    - $T(\lambda) = (1+1/(1- \lambda)^2)/2$
    - E.g. λ = 0. 5;
      $T(0.5) = (1+1/0.5^2)/2 = 2.5$
- Average # of cells examined for successful find:
    - $T(\lambda) = (1+1/(1- \lambda))/2$
    - E.g. λ = 0. 5;
      $T(0.5) = (1+1/0.5)/2 = 2.5$

Quadratic probing

- Eliminates problem of primary clustering by examining certain cells away from the original probe point.
- If the hash function evaluates to H and a search in cell H is inconclusive, try cells $H+1^2$, $H+2^2$ in sequence.
- Insertion algorithm:
    - Generate hashcode h = H = hash(key), i = 1
    - While A[h] contains a key:
        - h = (H+i*i) % tableSize
        - i++
        - A[h] = {key, value}
- Find algo:
    - Generate hashcode h = H = hash(key), i = 1

- While A[h] contains a key:
  - If (A[h] {key} == key) return A[h] {value}
  - h = (H+i*i) % tableSize
  - i++
  - Return Not Found

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99



ANALYSIS

- If table size prime and $\lambda < 0.5$, can never check same cell twice and new element can always be inserted
- Rough proof:
  - Choose prime M (= tableSize)
  - Choose 2 distinct I and j values < M/2
  - Suppose $H+i^2 \equiv H+j^2 \pmod{M}$
  - Then, $(i-j)(i+j) \equiv 0 \bmod M$, so M has a factor
  - By contradiction, first M/2 positions checked are distinct

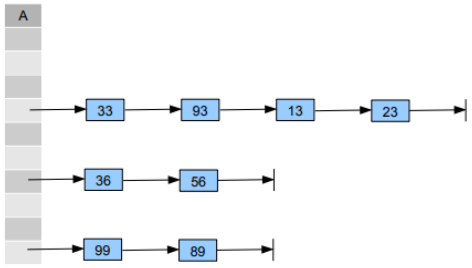
ISSUES

- Elements that hash to the same position probe the same alternative cells, which is known as secondary clustering.
- Techniques that eliminate secondary clustering:
  - Double hashing: a second hash function is used to drive the collision resolution.

Chaining

- Use table of pters/references
- Each new item must be added to a linked list at that position in the table
- Insertion algo:
  - Generate hashcode h = hash(key), p = new Node
  - p.data = {key, value}; p.next = A[h]
  - A[h] = p
- Find algo:
  - Generate hashcode h = hash(key), p = A[h]
  - While p! = null and p{key} != key
    - p = p.next
  - return p

- Using tableSize=10 and h(x)=x % 10
  - insert: 23, 56, 13, 93, 33, 36, 89, 99



ANALYSIS

- Hash function shows in which list to insert an item X and then, during a find, which list contains X.
  - Although searching a linked list is a linear operation, if lists are sufficiently short, search time will be very fast
- Appeal of chaining- performance is not affected by moderately increasing load factor; thus rehashing can be avoided.
- Assume N items
- Assume tableSize = M
- Assume even distribution of # values and use of all possible # values
- Then, on avg, each linked list is fo size N/M
- Avg time to search = ½ N/m
- WC O(N)
- BC O(N/M)
  - All about choice of M relative to N

Deletions

- Open addressing:
  - Deletion not easily possible
  - As with BST, item in # table not only represents itself but also connects other items by serving as a placeholder during collision resolution.
  - So, implement lazy deletion – marking items as deleted rather than physically removing them from the table
    - This info is recorded it extra member – each item is either active/deleted.
  - Add flag to item to mark it as deleted
    - Skip deleted items during search
    - Unmark and overwrite deleted items on insert
- Chaining:
  - Use linked list deletion opns

Other variations

- Double hashing
  - Secondary clustering is where a key generates the same sequence of locations to check

    - Use a second # func when there's a collision
    - h = H1 +H2, where H2 is the rehashing func
    - A different sequence is checked for each key
- Can implement chaining using BSTs/ other data strcs

Potential test questions (Hussein mentioned in class)

- {code} Height of binary trees
- {code} Size of binary trees
- Do an insertion process for an RB tree.
- What the mv command does on cmd line

Relational Database Design

Abbreviations

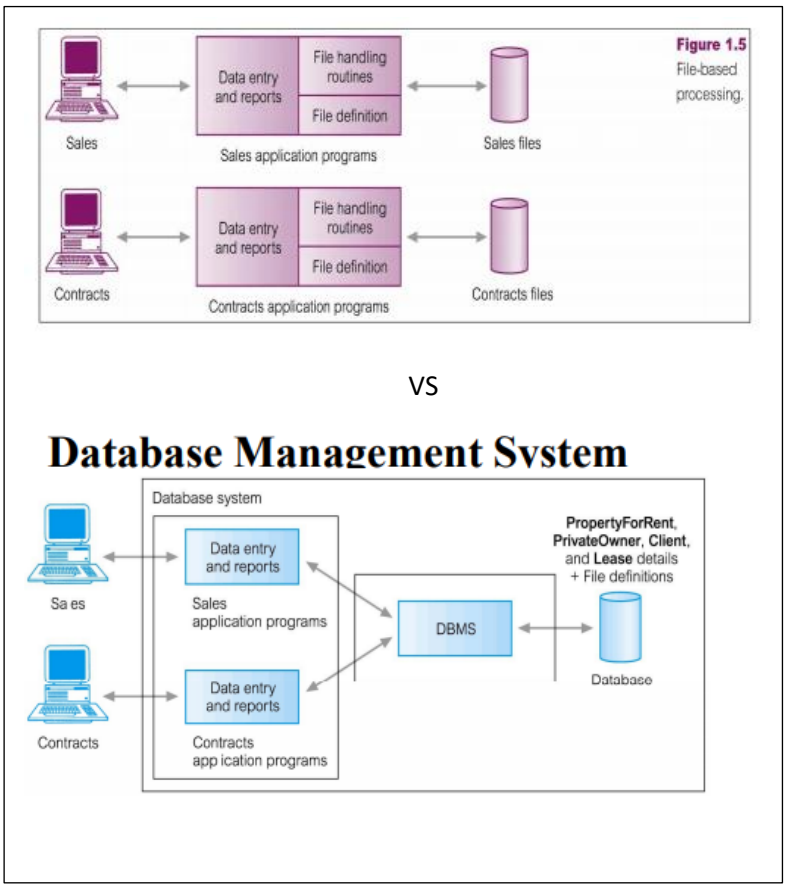
1. dtb = database
2. tbl = table
3. cols = columns
4. fk = foreign key
5. pk = primary key
6. attb/attrb (any variation) = attribute

- Without database, different systems in an organisation end up having their own programs with separate data files.
    - This makes it difficult to work with all the data together – results in many values being stored in multiple places and usually ends up with different copies of the same fact getting out of sync, making it impossible to know which value is correct.

Database Management System

- Database Management System: software system that's responsible for storing and manipulating data in a database.
    - Enables users to store, modify and extract info from the dtb as per the requirements.
    - Acts as an intermediate layer between programs and data – progs access DBMS, which then access the data.
    - Some examples: MYSQL, Oracle.
- A database ensures that there's only 1 system for storing the data of an organisation all together in one place
    - This is far better than having many systems that are separate like what can happen when programmers use files instead of databases.
- With a DBMS, programs are easier to write because they just send statements (like SQL) to the DBMS.
    - Actually working with data on disk is done by DBMS.
- DBMS provides ACID properties and keeps everything together in one place.
    - This system gives safe and easy way to access the data.
- Different users want to see different but overlapping parts of the database and want to see the data presented to the differently i.e. suit their own exact needs.
    - E.g. for UCT, Finance and Libraries need to have different views and needs with regards to the student data they access.

- **A visual guide:**



DATABASES VS FILES

- Files are usually designed for then certain applications, not for the whole organisation.
- Different apps often duplicate data in separate files and inconsistencies arise when the files disagree.
- For every new need, a new program must be written and tested
- Difficult to provide efficiency, security and reliability.

THE DISADVANTAGES OF PROG LANG FILES AND EXCEL

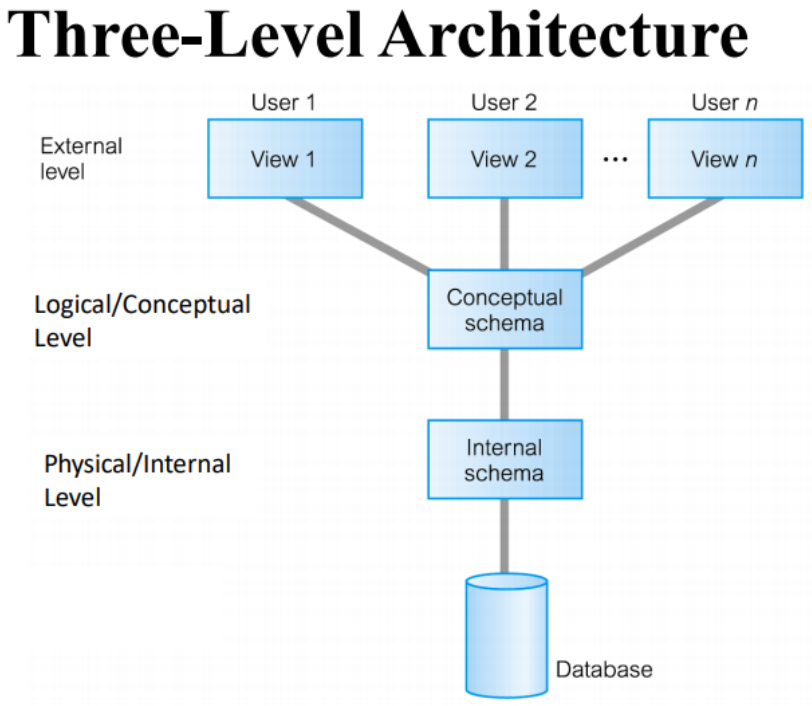
Mnemonic: **C**an **F**inding **I**diots **A**lways **S**tay **C**omplicated? (CFIAS)
UCT examples included

1. Consistency – if some fact stored in more than one place, all copies of the fact should always be the same.
    a. Same student number associated with 2 faculties?
2. Flexibility – if someone wants to ask an unprecedented/unexpected question about the data, they shouldn't have to wait for an app to be written
    a. Can't easily find who deserves funding/bursaries (complex)
3. Integrity – only *valid* data that obeys business rules should be allowed into the database. (ie no nonsense)

    a. Can't give transcripts to those who owe UCT money
4. Atomicity – only transactions that do complete fully and successfully should make changes to dtb (i.e ALL OR NOTHING)
    a. Move RXX from Stats to CS and loadshedding happens during this?
5. Security – unauthorised access and change should never be possible.
    a. How to limit access to salaries, fees owed, marks etc?
6. Concurrency – several transactions using the same data at the same time should *never* interfere with each other.
    a. Student and bursar pay into fee account simultaneously

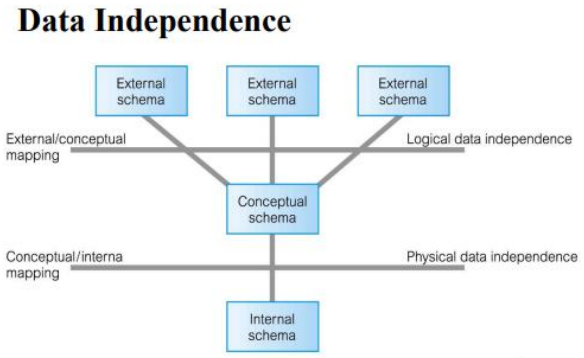
Architecture

- To simplify dtb design, 3 level architecture used.



- Only the dtb admin/ dtb design team needs to know about the 2 lower lvls – users and programmers just need to know top/external levels
- Dtb design team first designs conceptual level first
    - In relational dtbs, this is seen as being tables comprising of rows and columns.
    - Designing the conceptual level is deciding what tables to have and what columns to have in each table.
- Then, they decide on the physical schema i.e. how tbls will be stored on disk.
    - Dtb can then be created
- Finally, design team designs external level, which is how users will see parts of the dtb that are relevant to them and see it in a way that suits them best.

Data Independence



- Physical schema: specifies how data must be organised on disk, blocks, indexes etc
- Logical schema: relations and attributes
    - aka tables and columns.
- External schema/view: logical description of virtual relations that exposes only these parts of a dtb needed by user(s) as if the data is stored in a way that suits them.
- Physical data independence: if physical schema changes, logical schema doesn't need to change.
- Logical data independence: if logical schema changes (e.g. attributes removed or modified), the programs don't have to change because views can be defined instead.
- The dtb admin who is in charge of the dtb from an organisation will specify *mapping* from conceptual schema to each of the external schema – using views
    - Will also specify mapping from conceptual schema to internal/physical schema.
- Mappings can be changed if any of the lvls are changed – means can change any of the lvls without having to redo other lvls (because can just change the mappings)
    - This is called *data independence*.

A.C.I.D transactions

- Transaction: sequence of statements that together comprise one logical unit of work.
    - *Each* transaction should have these ACID properties.
- Mnemonic: **A**ll **C**ats **I**rritate **D**ogs

1. Atomicity: either whole transaction makes changes to dtb or none of it does
    a. All or nothing
2. Consistency: dtb never has contradicting facts
3. Isolation: many transactions can use same data at the same time and the effect is as if they'd run one after another
4. Durability: if transaction completes successfully, its dtb changes will never be lost due to errors/sys crash.
    a. BMDS writes everything to a log before it changes dtb and so log can always be used to "record" things if anything should go wrong later.

- Transaction example:
    - Mom gives R50 to her Son's bank account

- TStart; Read (Mom); Read(Son); Mom = Mom – 50; Son += 50; Write (Mom); Write(Son); Tend

1. **Atomicity**: ensures that if sys crashes during this transaction, then what will NOT happen is that only Write(Mom) changes the dtb but Write(son) did not. *Both* will change the dtb or none will change the dtb
    a. So, the R50 *won't* be lost.
2. **Consistency**: If Son's account has R200 in this instance of the dtb, then it won't be anything else (e.g. R250) anywhere else in the dtb.
3. **Isolation**: if 2 people deposit money into the Son's account (e.g. Mom and Grandmother) at the same time, it will be as if they did so one after another.
4. **Durability**: if, after the deposit takes place i.e. completes successfully, loadshedding happens, the transaction won't be lost, because the DBMS writes everything to a log *before* it changes the dtb.

Relational Data Model

- Tbls are called "relations" and each row has the data for a different real-world person/thing.
- Relations: collections of info of the same structure and meaning
    - Each of the items is comprised of a number of values called "attributes"
- Every Row has the same columns
- Alternative terminology:
    - Relation == table
    - Tuple == row
    - Attribute = column
- Value in each cell (each col of each row) is a simple value
    - Usually a string, date, #
    - These values are NOT structured values like arrays/similar.
- This model is intuitive and easy to use – people are familiar with spreadsheets and so used to seeing data like that.
- In terms of OOP:
    - A relation is like a data type
    - A tuple is like an instance of a type
    - An attribute is a single value.
- Relational model examples:

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Hypothetical Relational Database Model

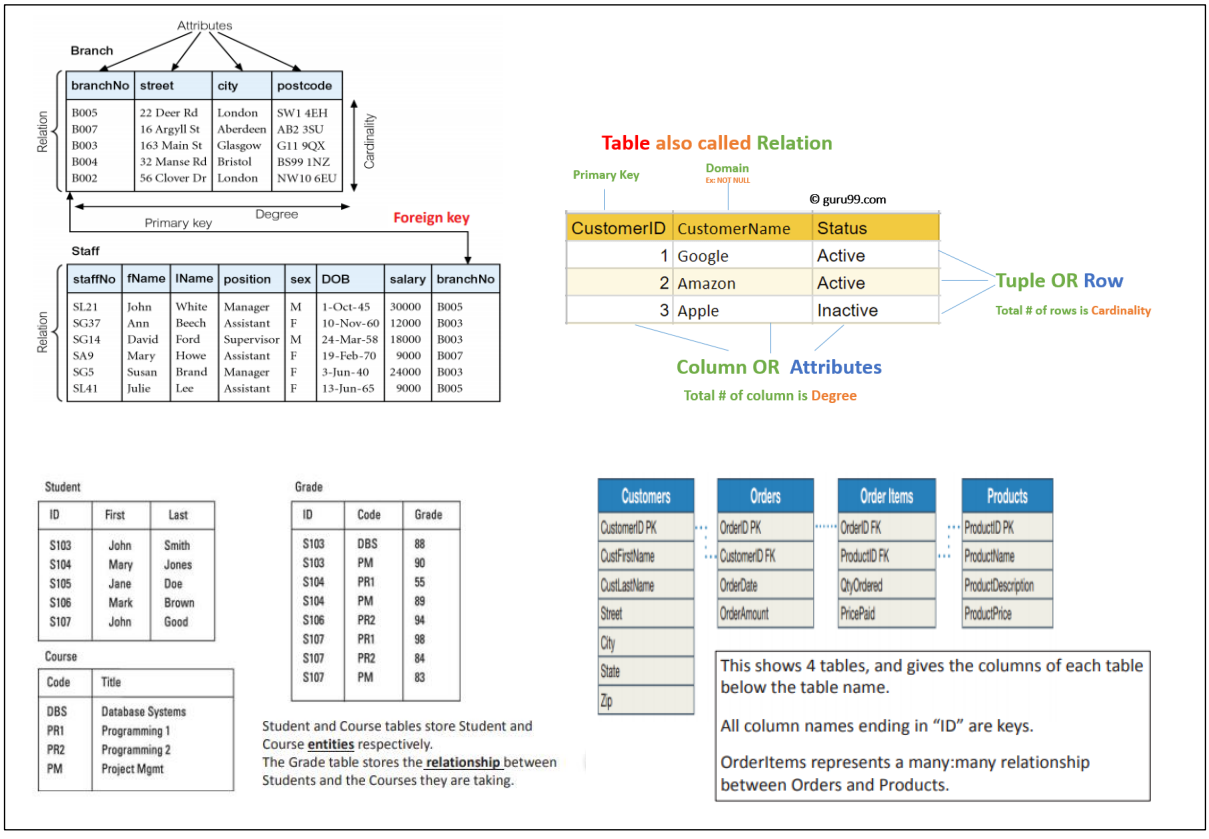
PubID	Publisher	PubAddress
03-4472822	Random House	123 4th Street, New York
04-7733903	Wiley and Sons	45 Lincoln Blvd, Chicago
03-4859223	O'Reilly Press	77 Boston Ave, Cambridge
03-3920886	City Lights Books	99 Market, San Francisco

AuthorID	AuthorName	AuthorBDay
345-28-2938	Haile Selassie	14-Aug-92
392-48-9965	Joe Blow	14-Mar-15
454-22-4012	Sally Hemmings	12-Sept-70
663-59-1254	Hannah Arendt	12-Mar-06

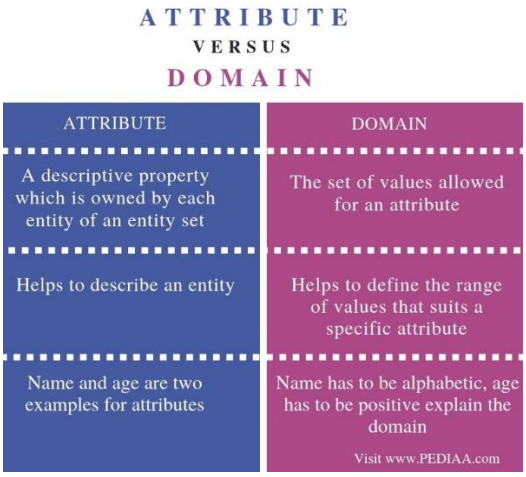
ISBN	AuthorID	PubID	Date	Title
1-34532-482-1	345-28-2938	03-4472822	1990	Cold Fusion for Dummies
1-38482-995-1	392-48-9965	04-7733903	1985	Macrame and Straw Tying
2-35921-499-4	454-22-4012	03-4859223	1952	Fluid Dynamics of Aquaducts
1-38278-293-4	663-59-1254	03-3920886	1967	Beads, Baskets & Revolution

TERMINOLOGY

- Cardinality: the # of tuples (rows) in a relation (table).
- Degree: the # of attributes (cols) in a relation (table).
- Primary key (pk): unique identifier for a row – distinguishes tuples of a relation from each other.
- Foreign key (fk): a reference to a row of another table.
    - Used to reference dtb tuples by means of their primary key.
- SQL is the lang used by all relational DBMS to access data.
- There are 2 kinds of tbls/relations in a relationship table:
    1. One-one-one
    2. One-to-many
- There are 2 things to model in *any* dtb – namely, entities and relationships
    - In relational databases, the tbl is used to represent both
    - Columns are properties of that entity/relationship.
- Examples:



ATTRIBUTE DOMAINS



- It's better to use attribute domains rather than simply saying if values in a column stores a specific data type e.g. strings
- Domain can restrict the possible values to be meaningful ones rather than nonsense values.
- Examples:

Entity Set	Attribute	Domain	Description
STUDENTS	stno	CHAR(10)	college-assigned student ID number
	name	CHAR(35)	full name
	addr	CHAR(35)	street address
	city	CHAR(20)	home city
	state	CHAR(2)	home state
	zip	CHAR(10)	home zip
COURSES	cno	CHAR(5)	college-assigned course number
	cname	CHAR(30)	course title
	cr	SMALLINT	number of credits
	cap	INTEGER	maximum number of students
INSTRUCTORS	empno	CHAR(11)	college-assigned employee ID number
	name	CHAR(35)	full name
	rank	CHAR(12)	academic rank
	roomno	INTEGER	office number
	telno	CHAR(4)	office telephone number

Attributes of Sets of Entities

Attribute	Domain Name	Meaning	Domain Definition
branchNo	BranchNumbers	The set of all possible branch numbers	character: size 4, range B001–B999
street	StreetNames	The set of all street names in Britain	character: size 25
city	CityNames	The set of all city names in Britain	character: size 15
postcode	Postcodes	The set of all postcodes in Britain	character: size 8
sex	Sex	The sex of a person	character: size 1, value M or F
DOB	DatesOfBirth	Possible values of staff birth dates	date, range from 1-Jan-20, format dd-mmm-yy
salary	Salaries	Possible values of staff salaries	monetary: 7 digits, range 6000.00–40000.00

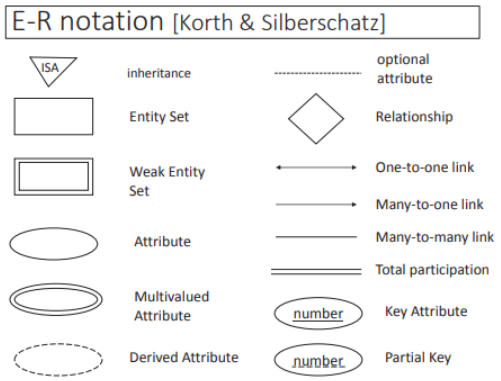
Schemas and Instances

- Schema: a definition of the type of data that will be stored.
- Instance: an actual collection of values that is stored in a dtb at some point in time.
- Schema = cookie cutter; instance = cookie
- For any schema, there can be many different instances of it over time
    - Think of it like OOP, you have a type like String, and there can be many different instances of that type.
- Schema of a relational dtb shows what the tbl(s) and their cols are
    - Defines relations and their attributes.
- Instance of a relational dtb is a collection of data in that dtb
    - i.e. values that are currently stored in all tbls of that dtb.
- Organisations A and B can have that same schema BUT have different dtb instances – data for A is on instance, data for B is another.

- Data in a dtb will change over time – schema usually stays the same (i.e. same tbls and cols) but data *inside* those tbls (dtb instances) changes.
- Specify constraints on attributes to ensure users can't enter nonsense in the data.
    - E.g. CustomerID must have numbers and letters – users can't just enter "peanut butter" and have that be the customer ID

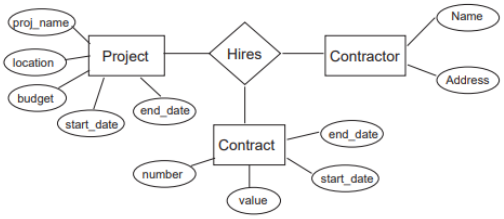
ER Models

- Why model data?
  - Communication
  - Document requirements, suggestions and decisions
  - Manage complexity
  - Identify significant features and hide unimportant ones
  - Identify gaps and errors.
- Entity set: a type of person/thing



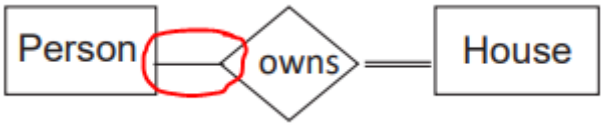
- Don't put key/any other attbs of entities as attbs of rel
- Relationships can have named lines (roles) to explain wha the entity is doing in the relationship.
- Ternary relationships:
  - Relationship can involve 1/ 2 entity sets.
    - Relationships with more than 2 entity sets aren't often needed.

Ternary Relationships can be used if you find this easier to show your understanding to the client

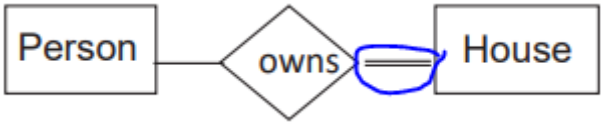


Cardinality and participation constraints

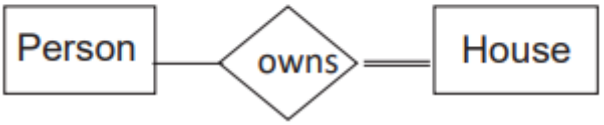
- Partial participation: entity doesn't have to participate in such a relationship i.e. optional.
  - Single line frm rectangle to diamond



- Total participation: entity *must* participate in this relationship.
    - Every entity will be in this relationship.
    - Double line from rectangle to diamond.

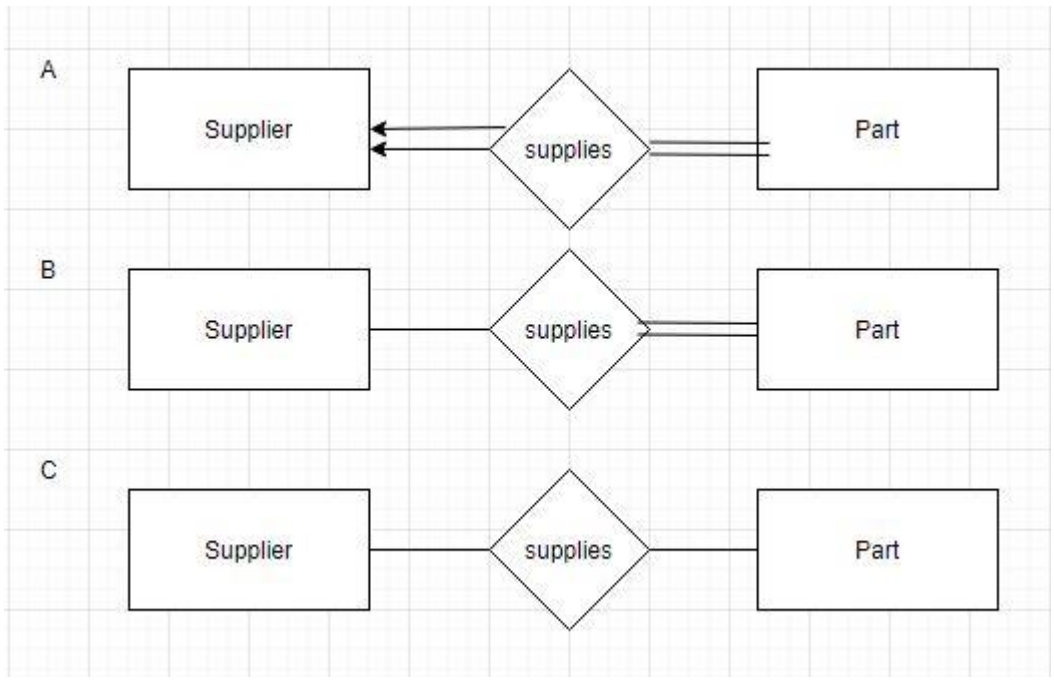


- Cardinality:
    - Arrowhead on entity means the relationship is “to one”
        - ie. “at most one”
    - No arrowhead => “to many”
        - Ie. “any number of”
    - Can find cardinality of the participating entity by reading from left to right or vice versa.
        - Read from left to right if finding cardinality of entity on LHS (left hand side).
        - Read from right to left if finding cardinality of entity on RHS.



    - Example can be read as :
        - Top: School must enrol many pupils
          AND A pupil must enrol in a school
        - Bottom: A person can own a house
          AND A house *must* be owned by a person

ADDITIONAL EXAMPLES



The example is of a business keeping track of the Parts they use and the Suppliers they buy these from.

(A) Each part is bought from exactly one supplier, and Suppliers are only in the database if we buy one or more parts from them.

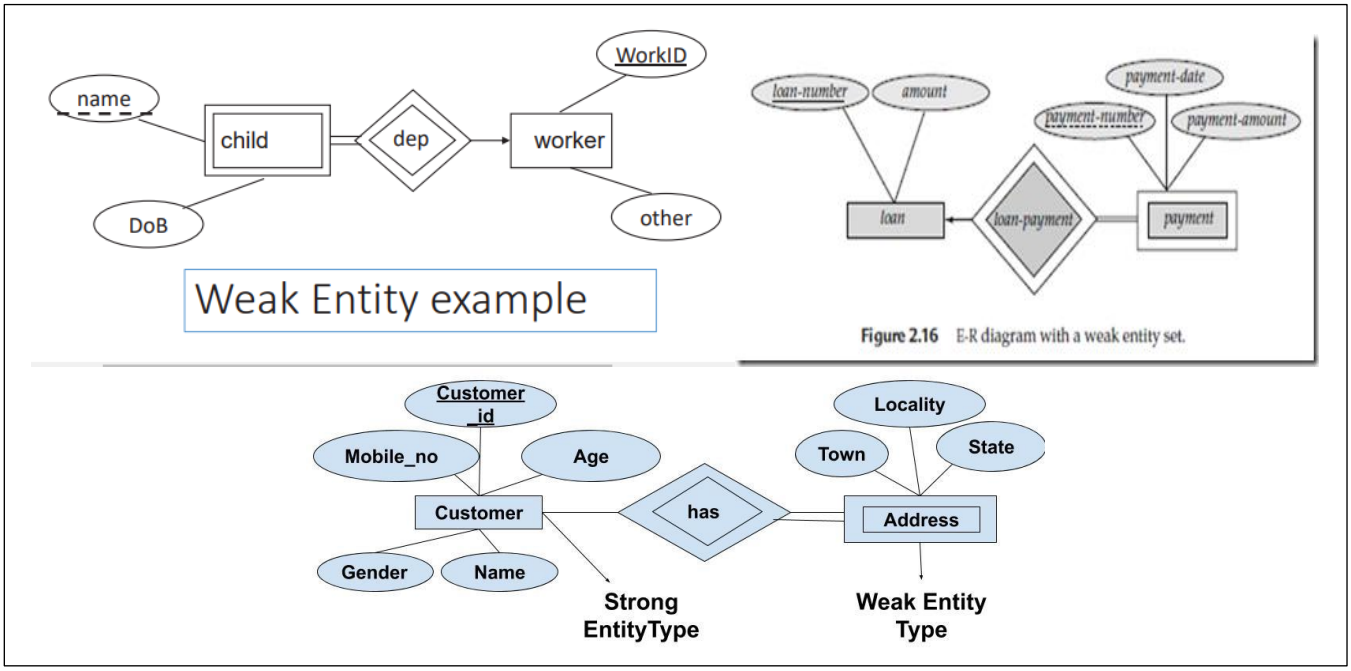
(B) Parts must all be buyable (able to be bought) from at least one Supplier, and sometimes parts are buyable from more than one Supplier. The database keeps information on all Suppliers the business knows about, some of them the business buys lots of different parts from, and some we don't buy any parts from at all.

(C) The database has data on Suppliers and Parts and who we buy them from, but there are no constraints on what happens. So some parts are not bought at all, some are bought from only 1 supplier and some are bought from many suppliers; some suppliers we don't buy any parts from at all, some suppliers we buy just 1 part from, and other suppliers we buy many parts from.

Weak entity

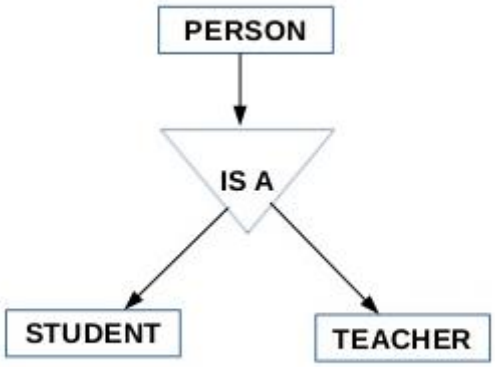
- Existence of weak entity set depends existence of *identifying entity set* (a strong entity set)
  - It's thus existence dependent on identifying entity set.
- Weak entity set: entity set that doesn't have a primary key.
  - Relationship must be *many-to-one* from weak entity to its identifying entity.
- Participation of weak entity set into the relationship must be **total**.
  - *Compulsory* association with its identifying entity (dbl line).
  - Weak entity is only of interest *because* it's *identifying entity* is of interest – thus has 'to-one participation".

EXAMPLES



Specialisation and Generalisation

- Specialisation looks at using "IS A" (inheritance) to add more detail to model so that important differences within one type of entity set are included in the model.
    - Specialisations have all the same attributes and relationships as generalisations, as well their own attrbs and/ relationships.
        - Specialisation <=> inheritance in OOP.



- Generalisation uses "IS A" conversely – if find diff parts of ER diag are quite similar , may be clearer for everyone if name a general case as new entity set that has all their common attbs and relationships and then ISA to name different kinds along with their additional relationships and/ attrbs.

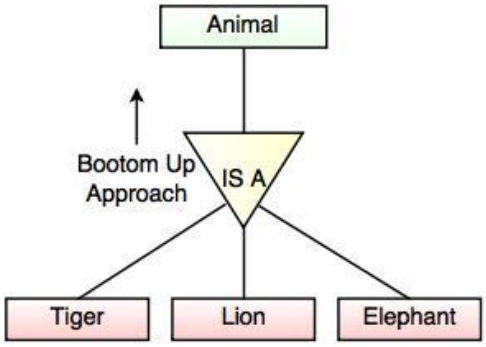


Fig. Generalization

- Summary:
    - If parts of ER model are similar = generalisation
        - Model will clearer
    - If entity has optional attrbs and / partial rels = specialisation.

EXAMPLES

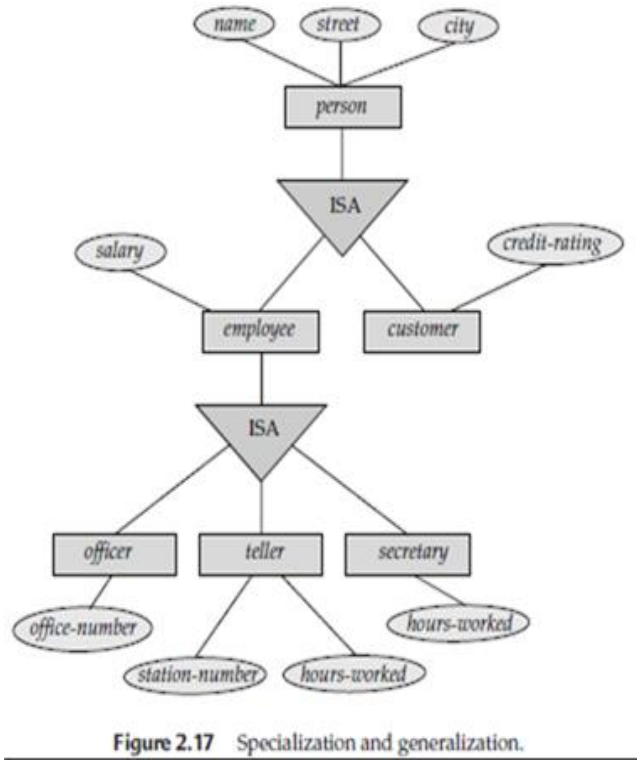
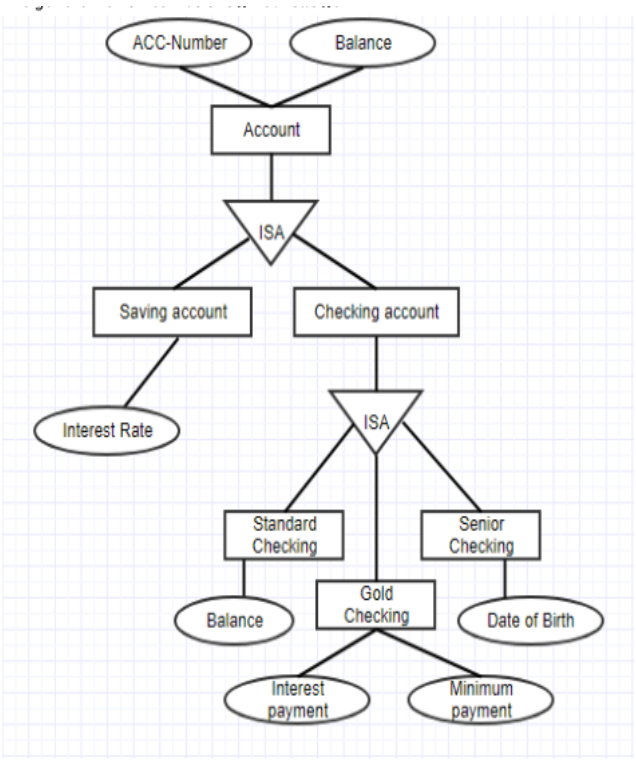
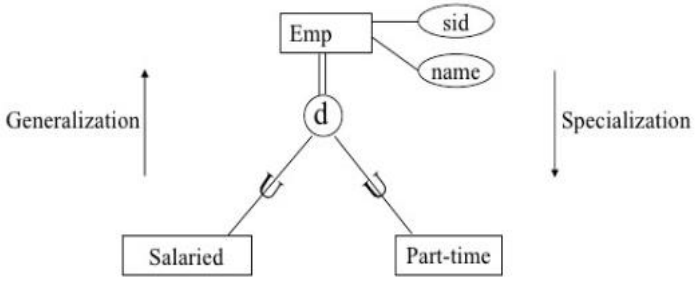


Figure 2.17 Specialization and generalization.

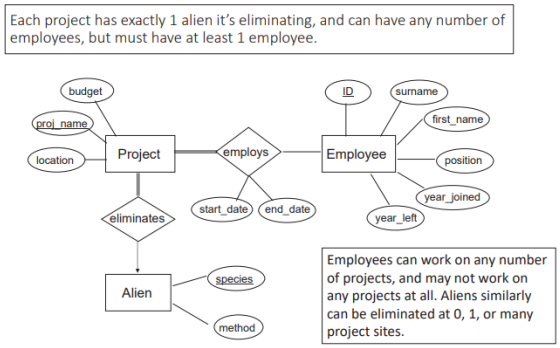


GENERALIZATION AND SPECIALIZATION



Inheritance constraints

- Determining membership:
  - Condition defines
    - E.g. dancer is a ballerina where style = “ballet”
      Tanker is-a ship where cargo= “oil”
  - User defined (no condition to check)
- Limiting membership:
  - Disjoint
    - E.g. undergrads or postgrads – can’t be both at the same time
      Vegan or vegetarian – can’t be both at the same time
      Driver and passenger – can’t be both at the same time
  - Overlapping
    - E.g student and staff – some are both
      Swimmer and swim teacher – some people are both
- Completeness:
  - Total – each higher-level entity must belong to a lower-level entity set
    - E.g. student *must* be postgrad/undergrad
  - Partial– some may not belong to a lower-level entity set
    - E.g. some persons at UCT are neither student/staff.
- Constraints aren’t shown on ER diag; but written alongside it.



ER Design

1. Identify entity sets
2. Add relationships between them
3. Attach attrbs
4. Choose and underline keys of entities
5. Check for weak entity sets
6. Specify ISA where appropriate
7. Check for total and to-one participation in relationships
8. Check for multivalued; optional; derived attbs

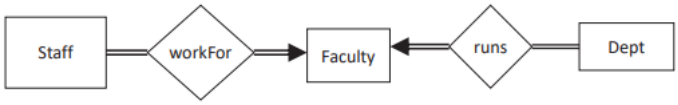
DESIGN ISSUES

- Entity or Attrb?
  - Usually something which *doesn't have* it's own attrb/relationships is an attrb rather than an entity
  - Usually nouns
  - Is it multivalued/needs its own attrbs/relationships?
- Entity or relationship?
  - Can only link entities in a relationship, *can't* link relationships.
  - Often verbs are relationships
  - Does it participate in relationship itself?
- Binary vs n-ary relationships?
  - Participation, cardinality etc.
- ISA
  - *Don't* use ISA unless diff. kinds are sufficiently diff in terms of the attrbs/relationships in the model.

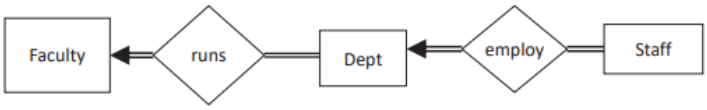
TRAPS

- "Fan trap"
  - Two M:1 relations
  - When have two different relations linked to the same entity, check if there's hidden hierarchical relationships that may've been missed.

Two M:1 relations in the "fan trap"

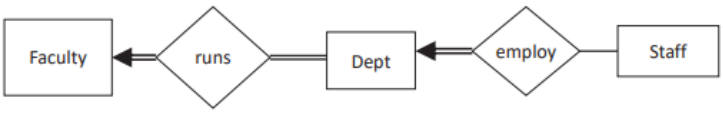


"fan trap" fixed: model the hierarchy



- "Chasm" trap:
  - Some entities aren't always part of the relationship
    - Another relationship for those entities may be required.

partial relationships in a “chasm trap”



- In above example, since there’s only *one* line from Staff to Employ, means not all staff are employed by the Department.

“chasm trap” fixed: add a relationship

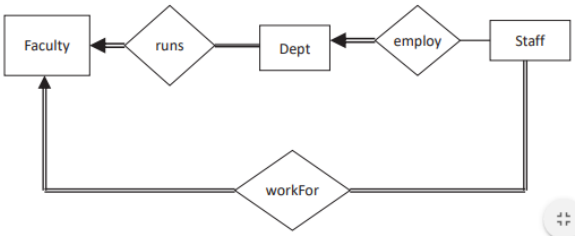


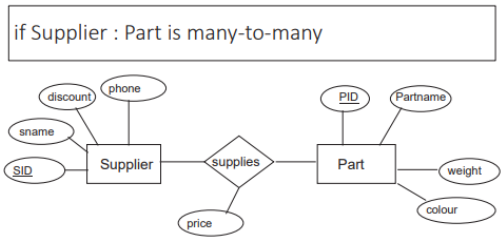
Diagram to Relation

- Translate ER diags to tbls
  1. Each entity becomes a tbl
  2. For one-to-one relationship:
     a. Use extra attbs instead of a tbl for the relationship.
  3. For weak entity:
     a. Instead of tbl for relaltionship, include key of identifying entity in the tbl for the weak entity.
  4. For any other relationship: extra table.

Relationships

M:N

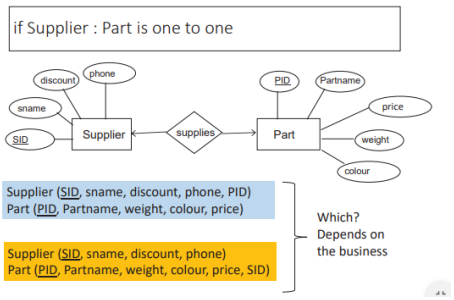
- Attributes connected to an entity become columns of its tbl, underlined attbs become pks of tbls.
- Tbl for a binary (2-way) relationship has a col for each of the entities being related , as well as a col for each of the relationship’s own attbs (if any)
- Fks *must* be used in a relationship tbl – fks can be pks of the other tbls involved in the relationship – both needed to uniquely identify row.



Supplier (SID, sname, discount, phone)
Part (PID, Partname, weight, colour)
supplies (SID, PID, price)

1:1

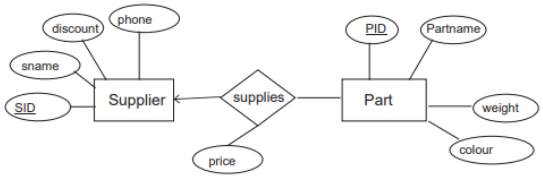
- Inefficient to have 3 tbls if relationship 1:1
  - Just store reference to one of the tbls inside the other.
    - Reference in this case will be a fk col
- In relational dtbs, references are the key of the entity tbl they refer to.
- DON'T store a reference to the other tbls in *both* tables
  - Why is this bad? The *same fact* is then stored in two different places (*must* avoid this – remember ACID properties?)
    - If two places don't agree, have no way of knowing which of the two places has the right info and which is wrong
- Choose a reference according to how it is expected this data will most often be used when dtb is up and running.



1: MANY

- If a one-to-many relationship *isn't* linking a weak entity to its identifying entity, make a separate table for it
- Still have to decide what key is for such a tabl
  - In relationship tbl, can be pk of the "to many" part of the relationship

if one-to-many (each Part has1 Supplier)



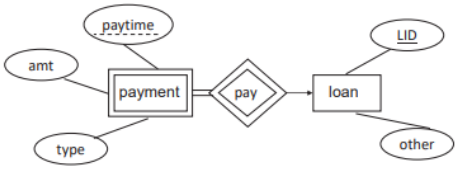
Supplier (SID, sname, discount, phone)
Part (PID, Partname, weight, colour)
Supplies (SID, PID, price)

WEAK ENTITY SET

- Weak entity can have partial key
  - Basically, for a given identifying entity, the partial key value will uniquely distinguish it from other weak entities associated with it.
- Thus, need to include key of identifying entity as an extra col in the tbl for the weak entity
  - These two keys *together* uniquely identify particular row in that tbl.

Weak Entity Set

LOAN (LID, other)
PAYMENT (LID, paytime, amt, type)

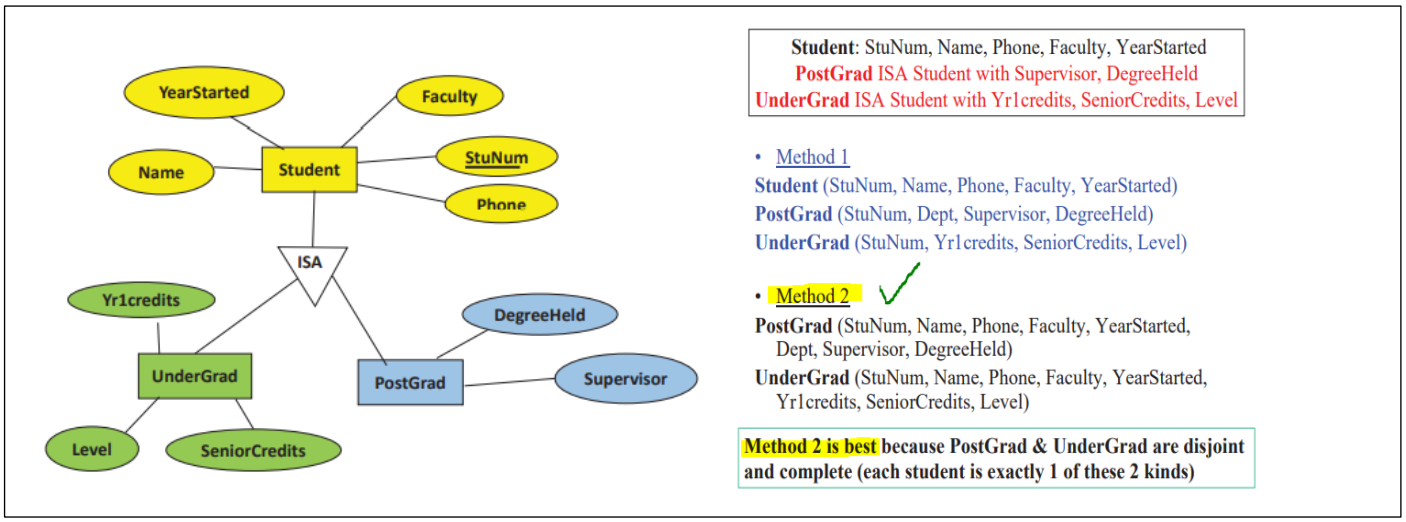


Attributes and ISA

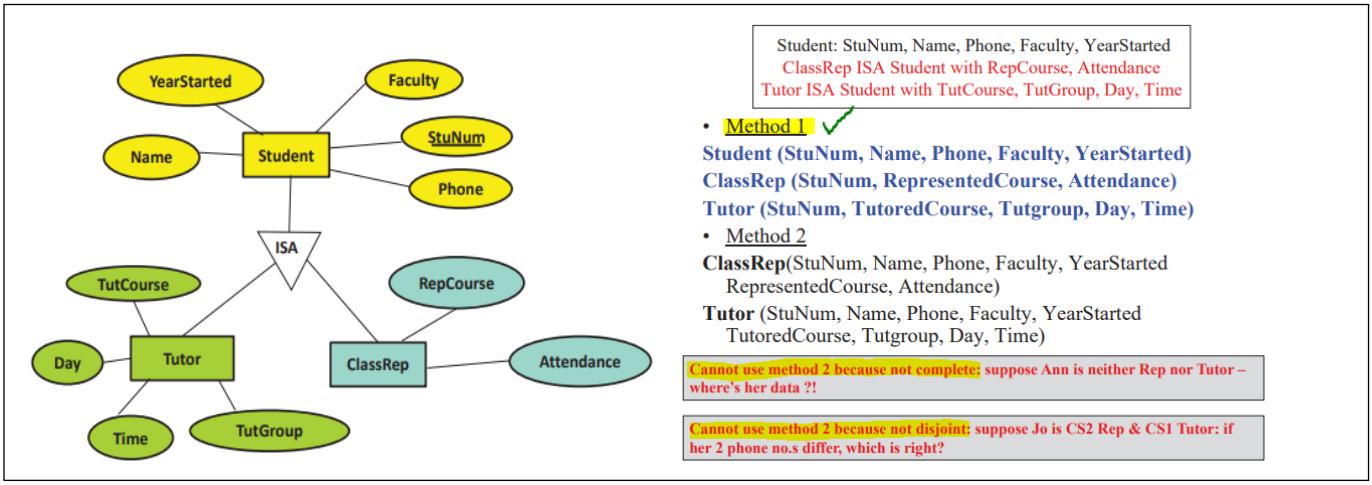
- Composite attributes: just have cols for component attrbs, not composite attrb.
  - E.g. if there's a composite attrb "Name" and it's composed of firstName and Surname, just have the 2 cols (one for firstName, the other for Surname).
- Multivalue attrbs:
  - Must have a tbl of its own.
  - Can only have 1 single val in any cell of the table.
  - Don't have separate cols – have *a separate table*.
    - Should hardly ever have separate columns rather than a separate table: e.g. if you think you'll have 3 columns for say PhoneNum of a person because nobody has more than 3, sooner or later you'll find someone who has 4 or 5 and then what?
- ISA (generalisation/specialisation)
  - Create tbl for supertype ("general") entity and for each "specialised" (subtype) entity.
    UNLESS specialisation is disjoint and total, then:
  - Create tbl for each specialised subtype entity but *no table* for "general" (supertype) entity.

        - Everything you would have put there (in additional table) is in one of the other tables (specialisation is ***total***). And each entity will be in just one of the specialised tables (because it's ***disjoint***) and not in more than 1 place.
- All-key table: all cols are needed to uniquely identify a row.
- There are 2 different ways to translate ER model with an ISA relationship into a relational dtb schema.
    1. 1 tbl per entity
    2. Add key of generalised entity to tbls for its specialised entity and *remove* general entity.
        - No danger of loss IF specialisation is complete
- Note:
    - Even if specialisation is disjoint but NOT complete, (or complete but not disjoint), *must still* use method 1.

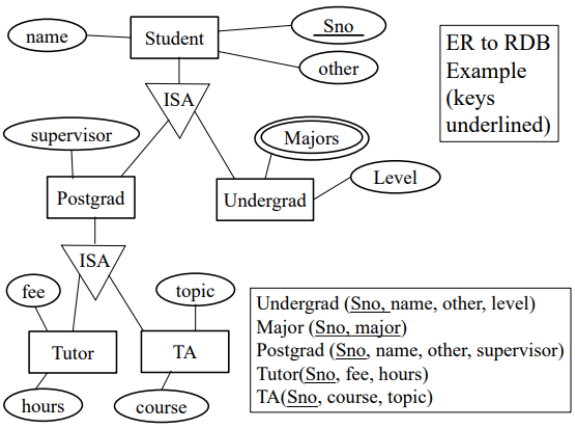
- Method 1:



- Method 2:

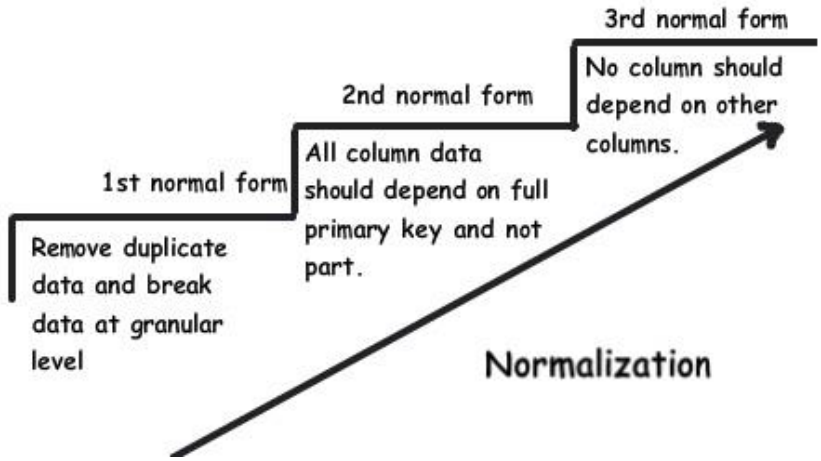


- Combined example:



Normal form theory

- Normalisation: every attrb is dependent on *the key* and ***nothing but the key***.
- Key: uniq identifier of a row in the tbl.
    - Is associated with 1 uniq val of every other col in that tbl.
    - Determines (uniquely) every val in its row.
- If there's more than 1 attb in the key, the tbl is a **relationship** tbl and so only *attbs of the relationship* belong there.
    - i.e. if we have the tables Customer, Rents and Video and the rents tbl has CustomerID and VideoID as it's key, then we know it's a relationship table
- If there's a relationship between 2 attrbs that isn't about the key, it needs its own table.
- A bad ER diagram leads to a bad schema



Why study NF theory?

Examples

- Suppose have a table as follows:

StaffBranch

staffNo	sName	position	salary	branchNo	bAddress
SL21	John White	Manager	30000	B005	22 Deer Rd, London
SG37	Ann Beech	Assistant	12000	B003	163 Main St, Glasgow
SG14	David Ford	Supervisor	18000	B003	163 Main St, Glasgow
SA9	Mary Howe	Assistant	9000	B007	16 Argyll St, Aberdeen
SG5	Susan Brand	Manager	24000	B003	163 Main St, Glasgow
SL41	Julie Lee	Assistant	9000	B005	22 Deer Rd, London

- The table "staffBranch" stores all info about the staff.
    - The problems: if for example, want to change the address from "163 Main Street" to "163 Main Road", have to then make sure it changes ***all occurrences*** of the address. This problem also applies to deleting a row – if we want to remove branchNo 003 from the database, have to find and remove ***all occurrences*** of it.

Another problem: if a staff with ID SA9 resigns, all info about the branchNo and bAddress is lost.

One more problem: if want to insert a new person to manage a new branch, but we don't have the address for the branch just yet. In this situation, we cannot appoint the new person *until* we have the bAddress for the branch.

- Suppose have another table:

FULL NAMES	PHYSICAL ADDRESS	MOVIES RENTED	SALUTATION
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

  - Problems: if "Robert Phil" changes his address, once again, have to **find all occurrences** and update them. The same applies if the salutation for "Janet Jones" changed to "Mrs".
  - The problems of deletion and insertion mentioned in the previous example can also be applied to this one.
- It would thus be best to have this table instead (Example 1):

Staff

staffNo	sName	position	salary	branchNo
SL21	John White	Manager	30000	B005
SG37	Ann Beech	Assistant	12000	B003
SG14	David Ford	Supervisor	18000	B003
SA9	Mary Howe	Assistant	9000	B007
SG5	Susan Brand	Manager	24000	B003
SL41	Julie Lee	Assistant	9000	B005

Branch

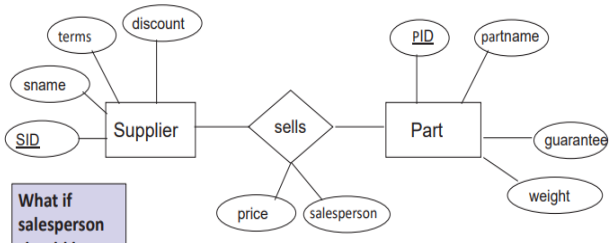
branchNo	bAddress
B005	22 Deer Rd, London
B007	16 Argyll St, Aberdeen
B003	163 Main St, Glasgow

have 1 fact in 1 place, not in many places

  - Using two tables will eliminate the aforementioned problems.
    - How? For example, if we want to change the address for branch B005, we can do it and the changes will reflect across the database. Since the address of B005 is only stored in one location, we only have to update it in the Branch table and not multiple times throughout the table (if we only had one table, like we had originally).
- For the 2nd example, it would be best to split table into Customer, with attrbs Full Name, Physical Address and Salutation; and Movies (with movie names). Ideally there should also be a table for the relationship between the two tables - we'll get to this later.

- One more example:

What if the discount depends only on the terms?



What if salesperson should be with Supplier?

Supplier (SID, sname, terms, discount)
Part (PID, partname, guarantee, weight)
Sells (SID, PID, price, salesperson)

Supplier (SID, sname, terms, discount)
Part (PID, partname, guarantee, weight)
Sells (SID, PID, price, salesperson)

So what if salesperson should be with Supplier?

SID	*PID*	*Price*	*Salesperson*
1	11	90	Ndaba
1	12	15	Ndaba
2	12	25	Fuller
3	13	40	Doe
3	14	75	Doe
4	14	15	Li
5	15	10	Baxter

repetition of Salesperson causes problems:

- if Doe is replaced we must **update** all rows of SID 3;
- if we **delete** the last row because no longer want PID 15, we lose info about SID 5;
- we cannot **insert** the name of a salesperson at a new supplied until we buy a part from them

Supplier (SID, sname, terms, discount)
Part (PID, partname, guarantee, weight)
Sells (SID, PID, price, salesperson)

So what if the discount depends only on the terms?

repetition of terms-discount relationship gives problems:

- if we now get a discount of 12 for terms of 30 days, must **update** all rows where terms is 30 days;
- if we **delete** the middle row because we stop dealing with Pinchers, we lose info about that terms of 90 days is discount of 5
- can't **insert** that terms of 15 is discount of 12 until we get supplier data for it

SID	*sname*	*terms*	*discount*
1	Dodgies	30	10
2	Parts-R-Us	60	8
3	Pinchers	90	5
4	Ripovs	60	8
5	Mr Priceless	30	10

terms 30 discount 10 means if we pay within 30 days we get a discount of 10%

Keys

- If attbs(X) are a unique identifier for the tuples (rows) of relation R and no subset of those attbs are a unique identifier , X is then a candidate key for R.
    - Don't want to define something as a key if *part* of those attrbs alone would be sufficient to uniquely identify a row – thus have a *minimal* set of attrbs to uniquely identify a row.
- Prime attribute (of R): A is an attrb of any candidate key for R.
- One of the candidate keys is chosen as the primary key for a relation
    - Will be used to represent that tuple/row in any other relation
        - i.e. will be the foreign key (fk) used to reference it in all its relationships

Functional Dependencies

- An FD
  - A → B means each instance of A is associated with *at most* one unique value of B in the real world.
    - The relationship from A to B is a ***"to-one"*** mapping
    - A in this instance is usually the pk
- A "functional determines" B.
- Functional dependency: a relationship between two attrbs, typically between the pk (primary key) and other non-key attrbs within a tbl.
  - Examples: StudentNumber → Student name
    CourseCode → CourseName, CourseConvenor
- Determinant: the determinant of an FD is the attrb/ group of attrbs on LHS of arrow
- Dependant: attrb/group of attrbs on the RHS of arrow.

WHY BOTHER WITH FDS?

- They are an important constraint to enforce on data in the dtb.
- Used to find keys for relations
- Used to check if relation scheme is okay.

FORMAL DEFINITION

- The FD
  - α → β holds on the relation R ***if and only if*** for any legal relations r(R), whenever any two tuples (rows) t1 and t2 of r agree on attrbs α , they also agree on attrbs β
    - That is, t1[α] = t2[α] => t1[β] = t2[β]

KEYS

- If X → R AND there's no Y ⊂ X and Y → R , then X is a candidate key for R
  - i.e. if we have StudentNumber, StudentAddress → StudentName, StudentSurname
    At UCT, all you need to find a student's name and surname is their student number. So, in this stance, { StudentNumber, StudentAddress } can NOT be a candidate key. Since we can technically use just StudentNumber to find a student's name and surname.
    - If it was just StudentNumber → StudentName, StudentSurname, then StudentNumber IS a candidate key.
- Prime attribute: any attrb that's part of a candidate key.
  - If we have for a video shop for example, VidID → VidName , the "VidID" is a prime attrb.
- Note: just because we can have an attrb be a primary key one way, does NOT mean it's reversible!
  - For example: StudentNumber → StudentName works because student numbers are ***unique***
    - However, StudentName → StudentNumber does NOT work because many people can have the name "Kim Jones".
      - This relationship implies that each name will have a unique student number but not in the way you think!

- Basically, this relationship implies that if a person has the name "Sam Smith", they should all have the *same student number*. But this is not the case at UCT. Many people can have the same name but all have different student numbers.

FULLY FUNCTIONALLY DEPENDANT

- A is fully functionally dependant on X if:
  - X → A AND for all Y ⊂ X, Y --/--> A
- Otherwise, if A is functionally dependent on X and A is *also functionally dependent* on a subset of X, then A is ***partially dependant on X***.
- An example to illustrate both these points:
  - Consider relation Employee(StaffNumber, Name, Office, DeskNumber, PhoneNumber, TaxNumber, MedicalAidNumber, JobTitle, Salary) with 4 candidate keys shown below:
    - **StaffNumber**
    - **TaxNumber**
    - **{MedicalAidNumbe**r , **Name}**
    - **{Office** , **DeskNumber}**
  - Salary is **fully functionally dependent** on {Office, DeskNumber} because
  - Office --/--> Salary and DeskNumber --/-->Salary, but Office,DeskNumber→Salary.
    - If you only know the Office not the desk, you can't know the salary because there can be many employees in the same office, and they'll have different salaries.
    - If you only know the DeskNumber like 4 then you can't know the salary because there are desks numbered 4 in many large offices, and those people will have different salaries.
  - Salary is **partially dependent** on {StaffNumber, PhoneNumber} because StaffNumber → Salary. You just need the StaffNumber to find out the Salary.

TRANSITIVE DEPENDENCIES

- If A, B and C are attrbs of a relation, such that A→B and B→C, then C is transitively dependent on A through B.
  - Example: assume every person who has medical aid gets a unique medical aid number. And to get medical aid you need to have an ID with an ID number (naturally). It thus follows:
  - MedicalAidNum → IDNum and IDNum → PersonName
  - Consequently, we can say: MedicalAidNum → PersonName
  - Another example: Suppose we are told that StaffNumber → JobTitle and that JobTitle → Salary. Then it follows that StaffNumber → Salary, and we say Salary is transitively dependent on StaffNumber through JobTitle
  - Final example: Suppose we are told StaffNo → Branch and we are told that Branch → Address. Then it follows that StaffNo → Address, and we say Address is transitively dependent on StaffNo through Branch

Attribute Closure

- Closure of X, X^+, are all the attrbs functionally determined by X.
    - i.e. X^+ gives all values that follow uniquely from X.
- Attribute closure: the properties of an attrb.
    - Closure of a set of attrbs i.e. {Y,X} is everything the dtb can give about any given XY pair.
    - Basically, it's all the info that the database can give you about an attrb/attrbs.
        - Examples: closure of a set of attributes like say {StuNum, CourseCode} is everything the database can give you about any given StuNum-CourseCode pair. At UCT the only thing it can tell you is whether or not that is a compulsory course in that student's degree, and whether or not that student has ever registered for that course.
        - The closure of {StuNum, CourseCode, Offering} can also tell you a great deal about what happened to that student in that particular offering of that course – percent, DP/DPR, class, etc.
- Attrb closure is used to find keys and to see if a FD is true/false.
    - Is X a key for R? } is X^+ the whole of R?
    - Is X → Y true? } is Y in X^+?

FINDING X^+

ALGORITHM

1. ans = X
2. For every Y → Z such that Y ⊆ ans, add Z to ans
3. Repeat until no more changes to X^+ possible.

EXPLANATION

- Algo builds up ans step by step
- Start with X itself
- At each step, check FDs to see if any of them has a LHS that is already in ans
    - If yes, implies already know that X → {LHS} and so ***by transitivity***, can add RHS to ans.
- Transitivity applies like this:
    - Since LHS is in ans and ans is everything uniquely determined by X:
        - X→ {LHS}
    - Have been given this FD:
        - {LHS} → {RHS}
        - So, by transitivity X → {RHS}
    - So, add RHS to ans.
- Note: although in tbls for entities, one column is the key, in tbls for relationships (i.e. the "buys" tbl in a Customer-Buys-Item ER model), generally have *more than one column*.

Examples

Example

SC → PMG
SL → C
CT → L
TL → C
SP → C

1. **Is SL a key for R**(SCPMGLT)?
2. **Does SL → PG?**

Example : attribute closure

SC → PMG SL → C CT → L TL → C SP → C
What is the closure of SL ?
start with ans = {SL}
using 2nd FD, ans = {SLC}
using 1st FD, ans = {SLCPMG}
no more attributes can be added so (SL)+ is SLCPMG
Is SL a key for R(SCPMGLT)?
no, SL ↛ T
Does SL → PG?
yes, because PG are in $(SL)^+$

Example: is AG a key for R ?

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B; A \rightarrow C; CG \rightarrow H; CG \rightarrow I; B \rightarrow H\}$
- Find $(AG)^+$
  1. $result = AG$
  2. $result = ABCG$ $(A \rightarrow C$ and $A \rightarrow B)$
  3. $result = ABCGH$ $(CG \rightarrow H$ and $CG \subseteq AGBC)$
  4. $result = ABCGHI$ $(CG \rightarrow I$ and $CG \subseteq AGBCH)$
- Is AG a candidate key?
  1. Is AG a key? Yes since $(AG)^+ = R$
  2. Is any subset of AG a key?
     1. Does $A \rightarrow R$? *no*
     2. Does $G \rightarrow R$? no

  So AG is a candidate key.

First Normal Form

- Un-normalised: a relation that isn't in 1NF
- In first normal form, only single values are permitted in columns and rows.
  - i.e. if we have for a client in a video shop:
    - Janet Jones: UP, Bee Movie, Taken in ONE column in ONE row, then this is a violation of 1NF (first normal form).
- To correct this, ensure that there is only a single value for every column AND every row
  - Split up any multivalued rows.
  - For the example above, should be:
    - Janet Jones: UP
    - Janet Jones: Bee Movie
    - Janet Jones: Taken

Example

Students

FirstName	LastName	Knowledge
Thomas	Mueller	Java, C++, PHP
Ursula	Meier	PHP, Java
Igor	Mueller	C++, Java

Startsituation

Result after Normalisation

Students

FirstName	LastName	Knowledge
Thomas	Mueller	C++
Thomas	Mueller	PHP
Thomas	Mueller	Java
Ursula	Meier	Java
Ursula	Meier	PHP
Igor	Mueller	Java
Igor	Mueller	C++

Second Normal Form

- 2NF relation is in 1NF and every non-prime attb is fully functionally dependent on the primary key.

CONVERTING 1NF TO 2NF

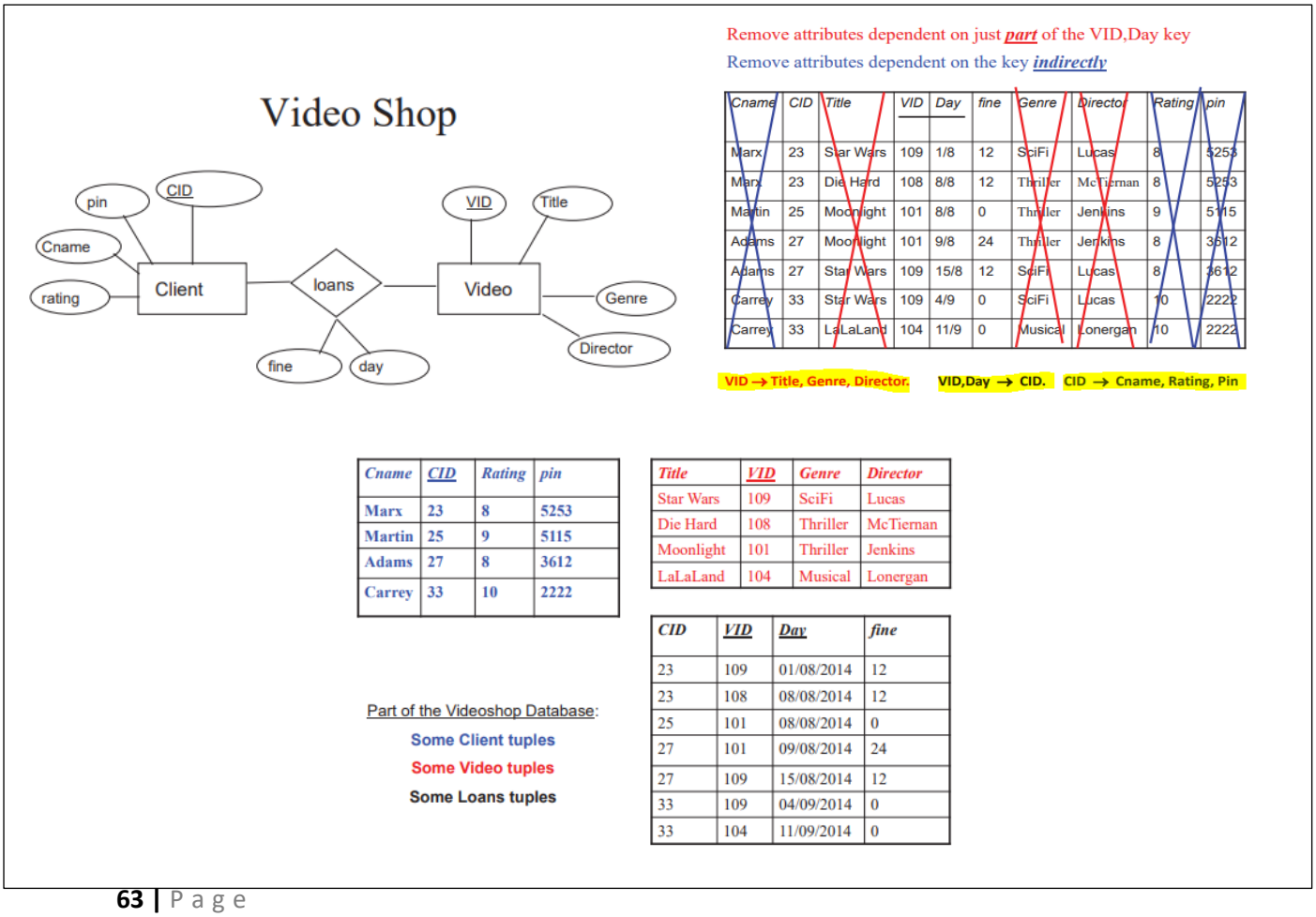
ALGORITHM

1. Find candidate keys of relation
2. Identify FD's
3. If any non-prime attrbs are *partially dependent* on pk, remove them and place them in a new relation along with a copy of their determinant.

EXPLANATION

- Once have candidate keys, we'll know which attrbs are prime.
  - No problem with prime attrbs being in the tbl
  - ONLY need to check if ***non-prime*** attrbs should be in the tbl or not.
- Look at FDs that have a LHS that **isn't** a key.
  - Those with keys on LHS don't worry about them, as attbs definitely belong in the tbl.
- If LHS of a FD is ***only part*** of a key for tbl, FD is saying the tbl is a relationship tbl, BUT there's an attrb that's a property of a participating entity and NOT of the relationship.
  - MUST remove it.

EXAMPLE



Cname	CID	Rating	pin
Marx	23	8	5253
Martin	25	9	5115
Adams	27	8	3612
Carrey	33	10	2222

Title	VID	Genre	Director
Star Wars	109	SciFi	Lucas
Die Hard	108	Thriller	McTiernan
Moonlight	101	Thriller	Jenkins
LaLaLand	104	Musical	Lonergan

CID	VID	Day	fine
23	109	01/08/2014	12
23	108	08/08/2014	12
25	101	08/08/2014	0
27	101	09/08/2014	24
27	109	15/08/2014	12
33	109	04/09/2014	0
33	104	11/09/2014	0

Part of the Videoshop Database:

Some Client tuples

Some Video tuples

Some Loans tuples

- The example has 3 FDS: (VID) → {VID, Title, Genre}; (VID,Day) →(CID); (CID) → {CID, Cname, Raing, Pin}
- Need to check FDs for problems:
    - (VID)+ is {VID, Title, Genre} so it is not a key …… [1]
        - This FD has ***part*** of the key as its left hand side, because Title,Genre,Director are attributes of the video *entity* only and not of the "loan" *relationship,* and so they *don't* belong in this table.
    - (CID)+ is {CID, Cname, Raing, Pin} so it is not a key ….[2]
        - This FD has a LHS that isn't part of any key, it is *a separate relationship* about clients and their attributes and doesn't belong in this table.
    - (VID,Day)+ is {VID, Day, CID, Cname, Rating, Pin, Title, Genre, Director} …..[3]
- Is (VID,Day) a key? VID alone is not a key (from[1] above) and (Day)+ is {Day}.
    - So (VID,Day) is a key because it's a unique identifier and no subset is a key.
        - i.e. can't just use either VID or Day to identify something.
    - Thus, this FD is not problem because its LHS is a key for the table.
        - The table is about *the relationship* between a video and the day it was loaned.

Third Normal Form

- A 3NF relation is in 2NF and there is no non-prime attrb that is transitively dependent on primary key.

CONVERT 2NF TO 3 NF

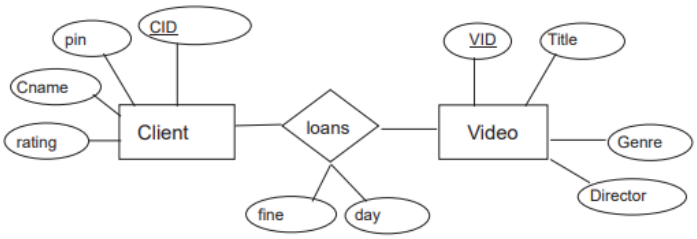
ALGORITHM

1. Find candidate keys of relation
2. Identify FD's
3. If transitive dependencies exist on pk, remove them by placing them in a *new relation* along with a copy of their determinant.

EXPLANATION

- Once have candidate keys, we'll know which attrbs are prime.
    - No problem with prime attrbs being in the tbl
    - ONLY need to check if *non-prime* attrbs should be in the tbl or not
- Look at FDs that have a LHS that **isn't** a key.
    - Those with keys on LHS don't worry about them, as the attbs on the left and RHS of those FDs definitely belong in the tbl.
- If LHS of an FD is NOT a key for tbl, FD is telling us that there's a relationship that doesn't belong in this tbl
    - MUST make a separate table for that relationship.

EXAMPLE



Remove attributes dependent on just ***part*** of the VID,Day key

Remove attributes dependent on the key ***indirectly***

Cname	*CID*	*Title*	*VID*	*Day*	*fine*	*Genre*	*Director*	*Rating*	*pin*
Marx	23	Star Wars	109	1/8	12	SciFi	Lucas	8	5253
Marx	23	Die Hard	108	8/8	12	Thriller	McTiernan	8	5253
Martin	25	Moonlight	101	8/8	0	Thriller	Jenkins	9	5115
Adams	27	Moonlight	101	9/8	24	Thriller	Jenkins	8	3612
Adams	27	Star Wars	109	15/8	12	SciFi	Lucas	8	3612
Carrey	33	Star Wars	109	4/9	0	SciFi	Lucas	10	2222
Carrey	33	LaLaLand	104	11/9	0	Musical	Lonergan	10	2222

VID → Title, Genre, Director. VID,Day → CID. CID → Cname, Rating, Pin

Cname	*CID*	*Rating*	*pin*
Marx	23	8	5253
Martin	25	9	5115
Adams	27	8	3612
Carrey	33	10	2222

Title	*VID*	*Genre*	*Director*
Star Wars	109	SciFi	Lucas
Die Hard	108	Thriller	McTiernan
Moonlight	101	Thriller	Jenkins
LaLaLand	104	Musical	Lonergan

CID	*VID*	*Day*	*fine*
23	109	01/08/2014	12
23	108	08/08/2014	12
25	101	08/08/2014	0
27	101	09/08/2014	24
27	109	15/08/2014	12
33	109	04/09/2014	0
33	104	11/09/2014	0

Part of the Videoshop Database:

Some Client tuples

Some Video tuples

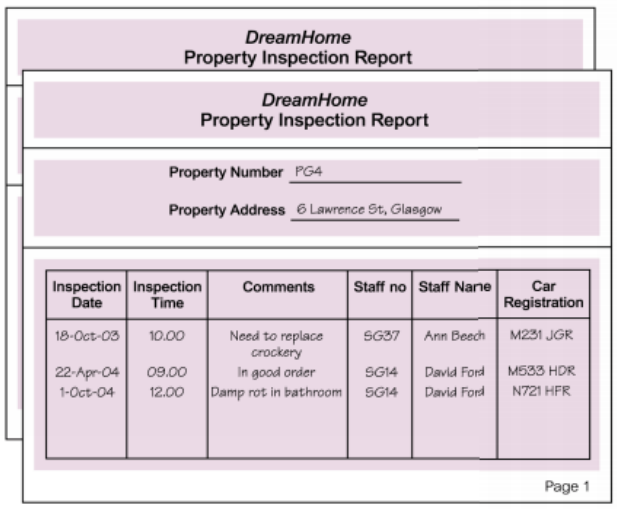
Some Loans tuples

CONVERT TO 3NF

ALGORITHM

1. Find candidate keys of relation
2. Check all FD's
3. For each X→ A where X is NOT a candidate key AND A is NOT prime.
    a. Remove A frm relation
    b. Make a new relation (X,A)

EXAMPLE



DreamHome
Property Inspection Report

DreamHome
Property Inspection Report

Property Number PG4

Property Address 6 Lawrence St, Glasgow

Inspection Date	Inspection Time	Comments	Staff no	Staff Name	Car Registration
18-Oct-03	10.00	Need to replace crockery	SG37	Ann Beech	M231 JGR
22-Apr-04	09.00	In good order	SG14	David Ford	M533 HDR
1-Oct-04	12.00	Damp rot in bathroom	SG14	David Ford	N721 HFR

Page 1

StaffPropertyInspection Data is not in 1NF (some cells are arrays/sets)

propertyNo	pAddress	iDate	iTime	comments	staffNo	sName	carReg
PG4	6 Lawrence St, Glasgow	18-Oct-03 22-Apr-04 1-Oct-04	10.00 09.00 12.00	Need to replace crockery In good order Damp rot in bathroom	SG37 SG14 SG14	Ann Beech David Ford David Ford	M231 JGR M533 HDR N721 HFR
PG16	5 Novar Dr, Glasgow	22-Apr-04 24-Oct-04	13.00 14.00	Replace living room carpet Good condition	SG14 SG37	David Ford Ann Beech	M533 HDR N721 HFR

StaffPropertyInspection Data is now in 1NF (all single-value cells)

propertyNo	iDate	iTime	pAddress	comments	staffNo	sName	carReg
PG4	18-Oct-03	10.00	6 Lawrence St, Glasgow	Need to replace crockery	SG37	Ann Beech	M231 JGR
PG4	22-Apr-04	09.00	6 Lawrence St, Glasgow	In good order	SG14	David Ford	M533 HDR
PG4	1-Oct-04	12.00	6 Lawrence St, Glasgow	Damp rot in bathroom	SG14	David Ford	N721 HFR
PG16	22-Apr-04	13.00	5 Novar Dr, Glasgow	Replace living room carpet	SG14	David Ford	M533 HDR
PG16	24-Oct-04	14.00	5 Novar Dr, Glasgow	Good condition	SG37	Ann Beech	N721 HFR

propertyNo,iDate,iTime → pAddress,comments,staffNo,sName,carReg
propertyNo → pAddress staffNo → sName staffNo,iDate → carReg

Remove attributes partially dependent on the key

P(propertyNo, pAddress)

Remove attributes transitively dependent on the key

S(staffNo, sName)
C(staffNo, iDate, carReg)

StaffPropertyInspection

propertyNo	iDate	iTime	pAddress	comments	staffNo	sName	carReg
PG4	18-Oct-03	10.00	6 Lawrence St, Glasgow	Need to replace crockery	SG37	Ann Beech	M231 JGR
PG4	22-Apr-04	09.00	6 Lawrence St, Glasgow	In good order	SG14	David Ford	M533 HDR
PG4	1-Oct-04	12.00	6 Lawrence St, Glasgow	Damp rot in bathroom	SG14	David Ford	N721 HFR
PG16	22-Apr-04	13.00	5 Novar Dr, Glasgow	Replace living room carpet	SG14	David Ford	M533 HDR
PG16	24-Oct-04	14.00	5 Novar Dr, Glasgow	Good condition	SG37	Ann Beech	N721 HFR

- First FD shows that {propertyNo,iDate,iTime} is a unique identifier for the rows of this table
- propertyNo→pAddress means this table is not 2NF because propertyNo is just ***part*** of the key and pAddress is not prime (is not part of the key).

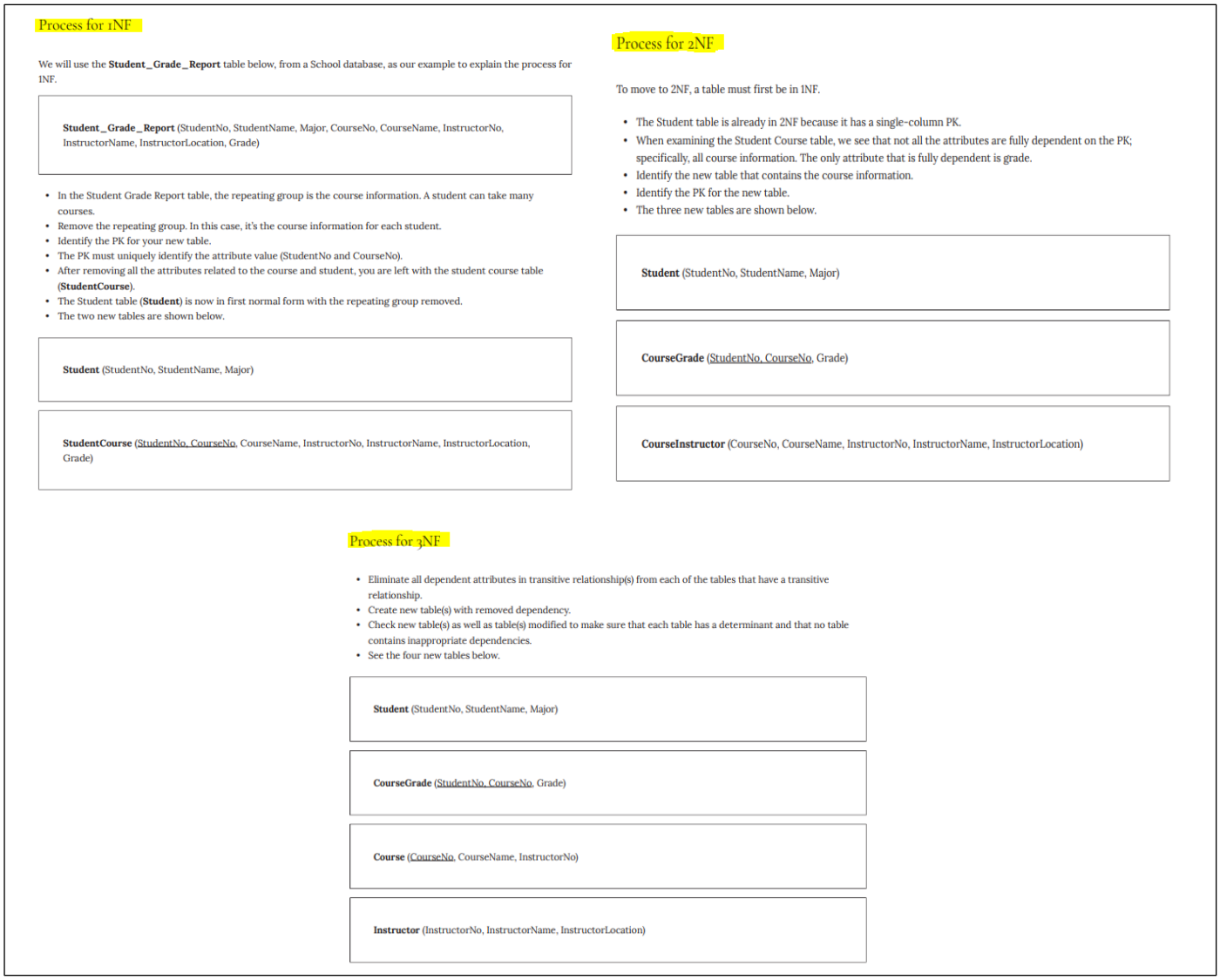
- Need to remove pAddress from our table and make a separate table for this FD. We chose to call that table P.

- The last 2 FDs do not have the key on their LHS, and their left hand side is not a subset of the key, so those last 2 FDs mean this tables is not 3NF.
  - Have to remove sName and carReg from the table because they are not prime (but keep iDate, because it is part of the key) AND
  - Have to make a new table for each of these 2 FDs. We called the new tables S and C.

Summary of Violations for Normal Forms

- 2NF: {just *part* of key} → {not prime}
  - Attrbs ***partially dependent*** on key
- 3NF: {not key AND not subset of key} → X
  - Attrbs ***transitively dependent*** on key.
- Example:

NOTE: this is NOT given in Sonia's notes



Process for 1NF

We will use the **Student_Grade_Report** table below, from a School database, as our example to explain the process for 1NF.

Student_Grade_Report (StudentNo, StudentName, Major, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

- In the Student Grade Report table, the repeating group is the course information. A student can take many courses.
- Remove the repeating group. In this case, it's the course information for each student.
- Identify the PK for your new table.
- The PK must uniquely identify the attribute value (StudentNo and CourseNo).
- After removing all the attributes related to the course and student, you are left with the student course table (**StudentCourse**).
- The Student table (**Student**) is now in first normal form with the repeating group removed.
- The two new tables are shown below.

Student (StudentNo, StudentName, Major)

StudentCourse (StudentNo, CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation, Grade)

Process for 2NF

To move to 2NF, a table must first be in 1NF.

- The Student table is already in 2NF because it has a single-column PK.
- When examining the Student Course table, we see that not all the attributes are fully dependent on the PK; specifically, all course information. The only attribute that is fully dependent is grade.
- Identify the new table that contains the course information.
- Identify the PK for the new table.
- The three new tables are shown below.

Student (StudentNo, StudentName, Major)

CourseGrade (StudentNo, CourseNo, Grade)

CourseInstructor (CourseNo, CourseName, InstructorNo, InstructorName, InstructorLocation)

Process for 3NF

- Eliminate all dependent attributes in transitive relationship(s) from each of the tables that have a transitive relationship.
- Create new table(s) with removed dependency.
- Check new table(s) as well as table(s) modified to make sure that each table has a determinant and that no table contains inappropriate dependencies.
- See the four new tables below.

Student (StudentNo, StudentName, Major)

CourseGrade (StudentNo, CourseNo, Grade)

Course (CourseNo, CourseName, InstructorNo)

Instructor (InstructorNo, InstructorName, InstructorLocation)

- Additional examples of converting relations to NFs
  https://vula.uct.ac.za/access/lessonbuilder/item/336947/group/a7dd7d20-ea3e-4096-b4f0-5ecdc36d348e/Databases/DatabasesTerm2/part15-more_3NF_examples-SLIDES.pdf

SQL

- Structured Query Language
  - SQL is a declarative language
- [x] used to indicate optional function
- Query: a request for info from a database
  - The info we want to get as a result is often subject to certain criteria
    - E.g. Display all customers who owe the company more than R1000

Select

- SELECT column(s) FROM Table [LIMIT X]
  - Can select any # of cols from table, in any order
  - [LIMIT X] – often used when working with large data sets, limits number of rows displayed
- SELECT column(s) FROM Table [ORDER BY A]
  - Gives all columns in ascending order of column A.
- AS is used to give a meaningful name to a column
  - Especially useful when columns have long names and will be used in specifying criteria in the query.
    - E.g. SELECT amountOutstanding AS AO FROM Customer
      WHERE AO>1000
    - It would be tedious to have to type “amountOutstanding” over and over again, much easier to use AO.
- SELECT DISTINCT
  - SQL by default allows duplicates
  - Use this to force elimination of duplicates.

EXAMPLES

```
select branchNo, street FROM BRANCH
WHERE city = 'London';
```

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Students

Sname	*SID*	*Uni*	*GPA*
Marx	23	UCT	52
Martin	25	WITS	71
Adams	27	UCT	66
Carrey	33	UJ	82

Profs

Pname	*PID*	*Papers*	*Topic*
Li	109	10	Java
Berman	108	50	Databases
Doe	101	40	Java
Roy	104	20	Java

SELECT * FROM Students - - gives the whole Students relation as result

SELECT SID, Sname, GPA/10 FROM Students ORDER BY SID DESC

SELECT SNAME FROM Students WHERE GPA <= 70

SELECT SUM (Papers) AS VALUE, COUNT (Papers) AS AMOUNT FROM Profs

Students

Sname	*SID*	*Uni*	*GPA*
Marx	23	UCT	52
Martin	25	WITS	71
Adams	27	UCT	66
Carrey	33	UJ	82

Students

Sname	*SID*	*Uni*	*GPA*
Marx	23	UCT	52
Martin	25	WITS	71
Adams	27	UCT	66
Carrey	33	UJ	82

SELECT Sname, SID, GPA/10 FROM Students ORDER BY SID DESC

Sname	*SID*	*GPA/10*
Carrey	33	8
Adams	27	6
Martin	25	7
Marx	23	5

SELECT SNAME FROM Students WHERE GPA <= 70

Sname
Marx
Adams

Where

- WHERE clause is condition which *row* must satisfy if it is to be included in the result.
  - Can be used with numeric, string type values etc.
  - E.g. SELECT numWins FROM Games
    WHERE numWins > 2 AND numWins<5
  - E.g SELECT names FROM Pets
    WHERE names = 'Spot'

Wildcards

- Used mainly in pattern matching
- LIKE used instead of "=" to do pattern matching.
- % matches any substring
    - E.g. SELECT animals FROM Zoo
      WHERE animals = "a%" ----> will select all animals that start with 'a'.
- _ matches a character
    - Can be used to specify length of what we're searching for
    - E.g SELECT names FROM Pets
      WHERE names like "_ _ _ _" ----> will select all names that are 4 letters long
- Wildcards can be combined :
    - SELECT animals FROM Zoo
      WHERE animals = "b_ _ _" ----> will select all animals that are exactly 4 chars in length and start with b.

EXAMPLES

```
select distinct  name, email
from             organisation
where            email like "%uct.ac.za"
```

Null values

- Null signifies a value that doesn't exist (i.e. missing value) or an unknown value.
- Null values introduce difficulties and complications in arithmetic, comparison and aggregate opns.
- If a tuple (row) is inserted and leave out one of its values. DBMS puts NULL into that cell.
- NULL can be used in SQL statements to mean empty/missing cell.
- Any comparison involving null value is treated as "false"
- Result of an *arithmetic* expression involving null is NULL
    - Any comparisons with null returns UNKNOWN
- If a WHERE clause evaluates to UNKNOWN, it's treated as false
- MIN, MAX, AVG, SUM, COUNT ignore tuples with null values
    - *Except* COUNT(*)

Aggregate functions

- Computes/returns a single value from a set of values.
- Appears in the SELECT clause
- AVG, MIN, MAX, SUM, COUNT are all aggregate functions

EXAMPLES

Aggregate Functions

```
select  avg(Papers ) as Average_Java_Papers
from    Profs
where   Topic = "Java"
```

```
select  count(*) as
from    Number_of_Profs Profs
```

```
select  min (Attended) as Lowest_attendance
from    Ratings
```

```
select  max (Attended) as Best_attendance
from    Ratings
```

```
select  sum(Papers) as Total_Papers
from    Profs
```

```
SELECT Color, SUM(ListPrice), SUM(StandardCost)
FROM Production.Product
WHERE Color IS NOT NULL
```

COUNT

- COUNT(*) gives the # of items altogether, including nulls i.e. the # of rows.
- COUNT(someCol) gives # non-null items i.e. # of non-null rowS
- COUNT(DISTINCT someCol) returns the # of *distinct* values excluding null and duplicates.

Set operations

- UNION, INTERSECT, EXCEPT (MINUS)
- Relations must be compatible i.e. must have *exactly* the same attbs.
- Using ALL in set operations will retain duplicates.
- Examples:

same schema

Profs

Pname	PID	Papers	Topic
Li	109	100	Java
Berman	108	500	Databases
Doe	101	400	Java
Roy	104	200	Java

Tutors

Pname	PID	Papers	Topic
Hu	211	10	Java
Fox	212	50	Databases
Codd	213	40	Java
Ben	214	20	Java

Students

Sname	SID	Uni	GPA
Marx	23	UCT	52
Martin	25	WITS	71
Adams	27	UCT	66
Carrey	33	UJ	82

Ratings

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	4	20
33	109	5	80
33	104	1	40

(SELECT * FROM Profs) UNION (SELECT * FROM Tutors)

~~(SELECT * FROM Students) EXCEPT (SELECT * FROM Ratings)~~

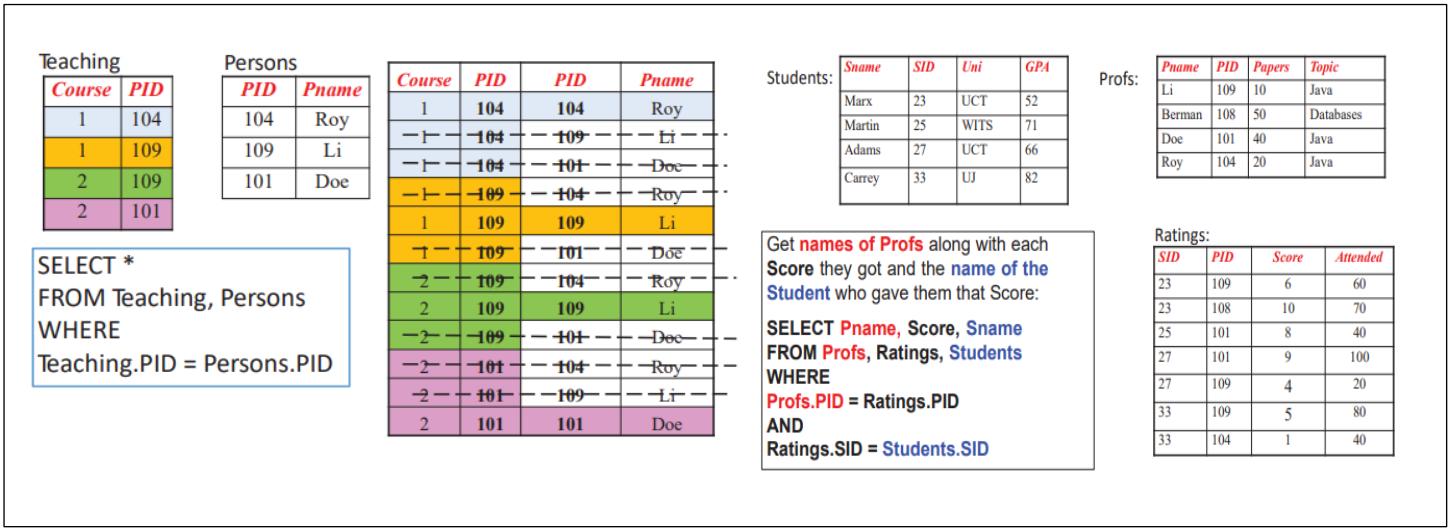
(SELECT SID FROM Students) EXCEPT (SELECT SID FROM Ratings)

The two relations to which UNION/EXCEPT/INTERSECT are applied must be **union compatible** – that means they must have the same relation scheme

CARTESIAN PRODUCT

- FROM clause used to access 2/more tbls – put all tbl names in clause.
    - If you just specify this, the result is the *cartesian product* of both tables
        - i.e. every row of the first table joined to every row of the other tables.
        - Not generally what is wanted
- To *prevent* getting cartesian product, use WHERE clause.
- NB to use AS to rename columns

EXAMPLES



Group by

- Use GROUP BY when want to know some aggregate function for each value in column.
  - Or when want to calc some aggregate function per column value.
- Imagine DBMS first taking all rows and sorting them so they're in order of GROUP BY column and then finding the aggregate function for each group.

EXAMPLES

Ratings

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	4	20
33	109	5	80
33	104	1	40

find the average score given **to each prof**

PID	mean
101	8.5
104	1
108	10
109	5

SELECT *PID, AVG(Score) AS mean*
FROM *Ratings*
GROUP BY *PID*

Ratings

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
~~25~~	~~101~~	~~8~~	~~40~~
27	101	9	100
~~27~~	~~109~~	~~4~~	~~20~~
33	109	5	80
~~33~~	~~104~~	~~1~~	~~40~~

Considering only majority-ratings, find the average score given **to each prof**

(a majority-rating is one where percent attended exceeds 50)

SELECT *PID, AVG(Score) AS mean*
FROM *Ratings*
WHERE *Attended* > 50
GROUP BY *PID*

PID	mean
101	9
108	10
109	5.5

HAVING

- Can put conditions on groups wanted in the result.
- Use after SELECT statement.
- HAVING is a condition put on the *group* , NOT a condition put on a row.

EXAMPLES

Problem: List the number of customers in each country. Only include countries with more than 10 customers.

```
1.  SELECT COUNT(Id), Country
2.    FROM Customer
3.   GROUP BY Country
4.  HAVING COUNT(Id) > 10
```

Results: 3 records

Count	Country
11	France
11	Germany
13	USA

Ratings

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	4	20
33	109	5	80
33	104	1	40

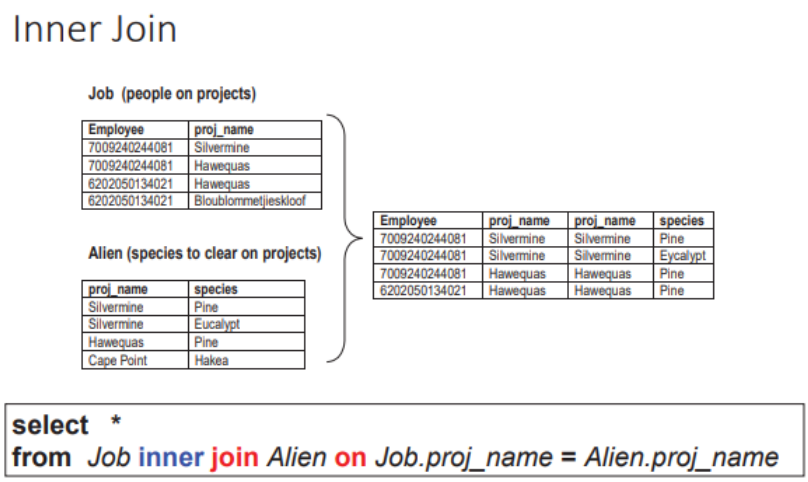
find the average score given **to each prof who has been rated more than once**

SELECT *PID, AVG(Score) AS mean*
FROM *Ratings*
GROUP BY *PID*
HAVING COUNT (*) > 1

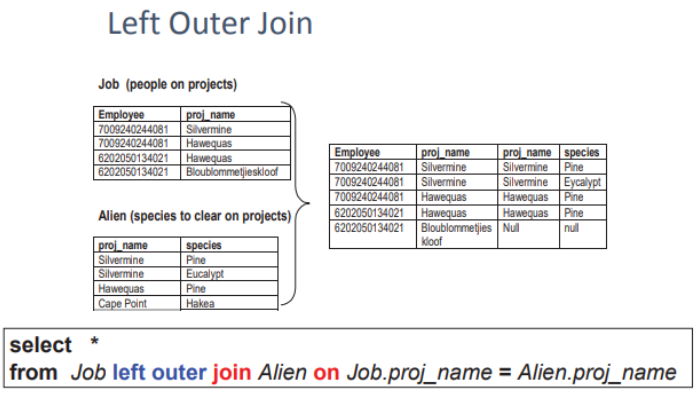
PID	mean
101	8.5
~~104~~	~~1~~
~~108~~	~~10~~
109	5

Join

- Can join tbls by referring to their pk and fks.
    - Paired rows can be tested against a WHERE clause
- Can use JOIN when working with 2/more relations in one query.
- Inner join:
    - Will leave out any row of either of the 2 tbls that doesn't join with any row in the other tbl
    - Example

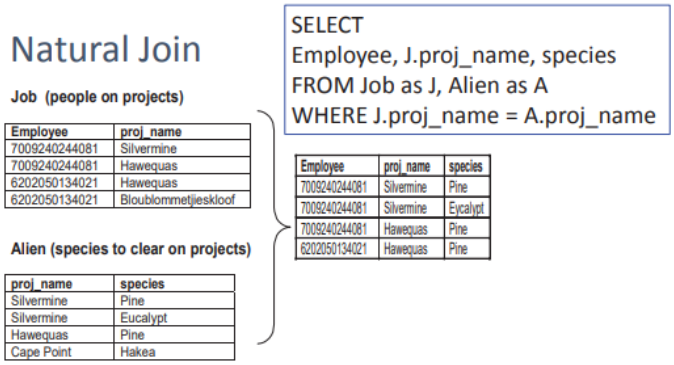


- Outer join:
    - For left outer join – each row of LH tbl will be in result.
        - If there's nothing to join to in the other tbl, it gets null in those columns instead.
    - Same logic but reversed for right outer join
    - Full outer join – all rows frm both tbls in result, with nulls where there's nothing to join to in the other tbl.
    - Example



- Natural join:
    - Automatically joins tbls where cols with the same name have matching values.
    - *Only* matches rows on equal same-col-same name value; whereas other types of join can use an expression in the ON part.

EXAMPLES




```
select * from Job natural inner join Alien
```

- Join types: defines how tuples/rows which don't match are handled
  - Types = inner, right/left/full outer, natural
- Join conditions: defines which tuples/rows in the two relations match.
  - Natural – uses all attbs that are common to both
  - ON <predicate> - used when attbs have different names.
  - USING(A1,A2,…,An) – similar to natural join condition
    - A1,…,An are join attbs
    - Specifies common attbs to join on; rather than all attbs common to both.
    - Why have USING?

Have 2 tables, Orders & Deliveries, which both include 3 columns called OrderNum, SupplierNum and Amount (among many other columns they have). Want to see all orders and their matching deliveries, but sometimes the Amount they charged us when we ordered is different from what they charge when they deliver. Natural join would not pick up orders that have been delivered with a new Amount. So we can say USING(OrderNum,SupplierNum) and then it will only join where those 2, not all 3, values match.

Nested select

- Make a set of values to compare with by enclosing them in round brackets and using IN
  - Here "IN" implies "is an element of".
  - Can have SELECT within these brackets.

EXAMPLES

Students

Sname	*SID*	*Uni*	*GPA*
Marx	23	UCT	52
Martin	25	WITS	71
Adams	27	UCT	66
Carrey	33	UJ	82

Ratings

SID	*PID*	*Score*	*Attended*
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	5	20
33	109	4	80
33	104	1	40

```
select Sname from Students where Uni not in ("UJ", "WITS")
```

Which students with a GPA above 70 have given rating(s) below 3?

```
select Sname from Students where GPA > 70 and
    SID in (select SID from Ratings where Score < 3)
```

- SOME = existential quantifier
  - Needs to fit one/more of the criteria
- ALL = universal quantifier
  - Needs to fit all criteria
- Example

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	5	20
33	109	4	80
33	104	1	40

```
- - who scored below some score of 109?
select distinct PID from Ratings
where Score < some (select Score from
Ratings where PID = 109)

- - who scored below all 109's scores?
select distinct PID from Ratings
where Score <= all (select Score from
Ratings where PID = 109)
```

- EXISTS usually used with * because it just tests to see if the result is the empty set or not.
- UNIQUE usually used with * because it just tests to see if the set is 1 element or not.
- Example

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	5	20
33	109	4	80
33	104	1	40

```
- - who has not been rated:
select distinct Pname from Profs as P
where not exists (select * from Ratings
where Ratings.PID = P.PID)

- - who has been rated exactly once:
select distinct Pname from Profs as P
where unique (select * from Ratings
where Ratings.PID = P.PID)
```

- FROM – SELECT clause can use an intermediate result of a nested select instead of giving the name of an actual dtb in the FROM part.
  - This nested select must be followed by AS and then a tbl name and col names for its intermediate/temporary result.
- Example

SID	PID	Score	Attended
23	109	6	60
23	108	10	70
25	101	8	40
27	101	9	100
27	109	5	20
33	109	4	80
33	104	1	40

```
select PID, total / howMany as mean
from (
        select PID, count( * ), sum(score)
        from Ratings where Attended > 50
        group by PID
      )
     as result (PID, howMany, total)
where howMany > 1
```

Considering only majority-ratings, find the average score given to each prof who has been majority-rated more than once
(a majority-rating is one where percent attended exceeds 50)

```
Select PID, count(*) / sum(Score) as mean
from Ratings
where Attended > 50
group by PID
having count(*) > 1
```

Table Creation

```
create table    Ratings
      (SID      integer  not null,
       PID      integer  not null,
       Score    enum (0,1,2,3,4,5,6,7,8,9,10) not null,
       Attended integer,
      primary key (SID, PID),
       check (PID in (select PID from Profs)),
       check (SID in (select SID from Students) )
      )
```

- "not null" – if used for any col, DBMS won't allow row to be inserted/updated unless col has a value.
- "check" ensures that values is valid
- There are SQL data types which cols can have, such as:
  - Date and time
  - Numeric
  - String
  - See Week 1 part8 slides 2-4 (ula.uct.ac.za/access/lessonbuilder/item/335972/group/a7dd7d20-ea3e-4096-b4f0-5ecdc36d348e/Databases/DatabasesTerm2/part8-other-SQL-statements-SLIDES.pdf) for more in depth info

DELETE

- Removes from tbl all rows for which a specified criterion is true.
  - In no criteria specified, all rows are delete BUT *tables remains* – it is just empty.

EXAMPLES

```
DELETE FROM STAFF WHERE staffNo = 'SA9';
```

Deleting Rows

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

```
DELETE FROM
BRANCH WHERE
branchNo = 'B002' OR
branchNo = 'B004'
```

```
delete from  Profs
where        Pname = 'Roy'
```

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005


```
delete from  Profs
where        Papers < (select avg(Score) from Profs  )
```

INSERT

- Can insert into one tbl by selecting data from another table.

EXAMPLES

```
INSERT INTO BRANCH
VALUES ('B008', '4 Ivy Rd', 'London', 'NW4');
```

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Inserting Rows

Pname	*PID*	*Papers*	*Topic*
Li	109	10	Java
Berman	108	50	Databases
Doe	101	40	Java
Roy	104	20	Java

```
insert into Profs
values ('Marais',111, 57, 'Unix')
```

```
insert into Profs
( select * from Tutors where Topic like 'Database%')
```

UPDATE

- Can use WHERE clauses to say which rows must be changed

EXAMPLES

```
UPDATE STAFF SET salary = salary * 1.1
WHERE position = 'Assistant';
```

Branch

branchNo	street	city	postCode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse Rd	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNo	fName	lName	position	sex	DOB	salary	branchNo
SL21	John	White	Manager	M	1-Oct-45	30000	B005
SG37	Ann	Beech	Assistant	F	10-Nov-60	12000	B003
SG14	David	Ford	Supervisor	M	24-Mar-58	18000	B003
SA9	Mary	Howe	Assistant	F	19-Feb-70	9000	B007
SG5	Susan	Brand	Manager	F	3-Jun-40	24000	B003
SL41	Julie	Lee	Assistant	F	13-Jun-65	9000	B005

Updating Rows

```
update Profs
set    Papers = Papers + 1
where  Pname in ('Li', 'Roy')
```

Views

- Used for external layout of 3-layer architecture
- Any select can be used to define what data is needed in the view
- Views should only be used for fetching data from database and NOT for inserting, updating or deleting data.
- Views <=> 'virtual tables'
- There are 2 main reasons for using views:

- Security
  - E.g. columns like “salary” that are in tbl aren’t included in view (for those who aren’t meant to see them).
- Ease of use
  - E.g. no need for complex SQL like “join”.

Views

```
create view Staff as
        select Pname, PID from Profs

insert into Staff values ('Hu', 110)

create view Teaching as
        select Pname, Topic from Profs
- - the view above doesn't include PID which is the key of People, so the DB
- - Administrator must specify only select operations allowed on PeopleJobs

create view JavaTeachers as
        select Pname, PID from People where Topic = 'Java'
```

Pname	*PID*	*Papers*	*Topic*
Li	109	10	Java
Berman	108	50	Databases
Doe	101	40	Java
Roy	104	20	Java

Manipulating schemas

- Can remove/redefine columns in tbl – ALTER TABLE
- Can also remove rows and tbl itself from dtb – DROP TABLE
- Note: drop =/= delete!

Manipulating Schemas

```
alter table Students add Degree varchar(40)

alter table Ratings drop Attended
```

```
drop table    Tutors

delete from   Tutors
```

⇒ different operations

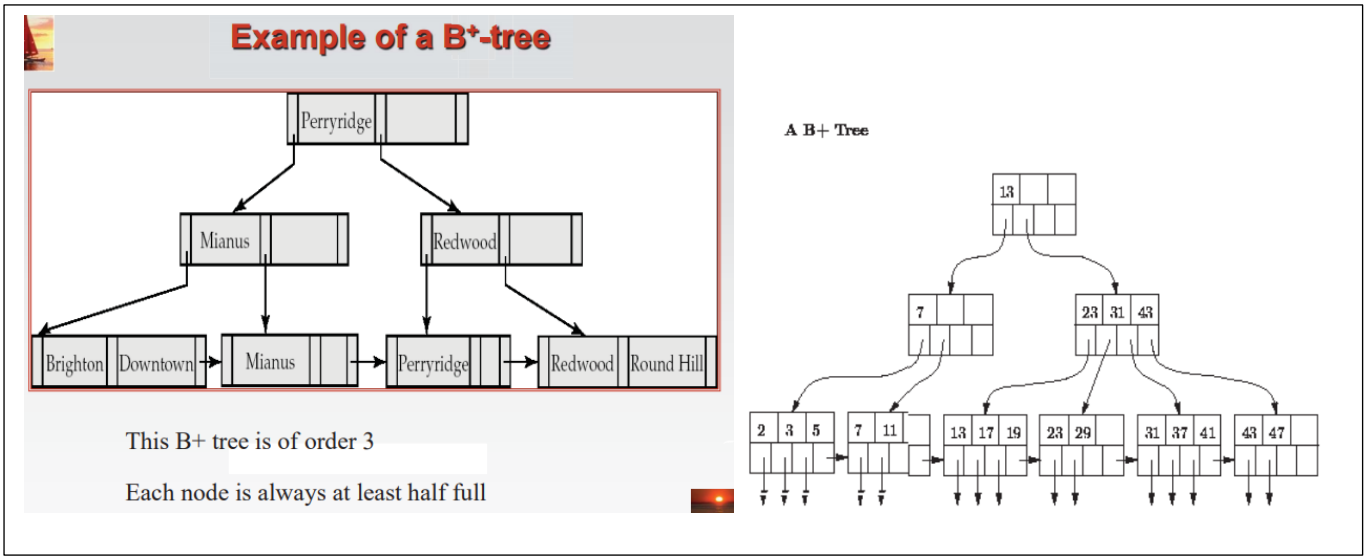
B+ Trees

Abbreviations

1. FQ = forum question with answers

- https://www.cs.nmsu.edu/~hcao/teaching/cs582/note/DB2_4_BplusTreeExample.pdf
- https://www.cs.princeton.edu/courses/archive/fall08/cos597A/Notes/BplusInsertDelete.pdf
- https://iq.opengenus.org/b-tree-search-insert-delete-operations/
- https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html
- https://www.db-book.com/db7/slides-dir/PDF-dir/ch14.pdf
- "Most queries can be executed more quickly if the values are stored in order. But it's not practical to hope to store all the rows in the table one after another, in sorted order, because this requires rewriting the entire table with each insertion or deletion of a row.
    - This leads us to instead imagine storing our rows in a tree structure. Our first instinct would be a balanced binary search tree like a red-black tree, but this really doesn't make much sense for a database since it is stored on disk. You see, disks work by reading and writing whole blocks of data at once — typically 512 bytes or four kilobytes. A node of a binary search tree uses a small fraction of that, so it makes sense to look for a structure that fits more neatly into a disk block.
    - Hence the B+-tree, in which each node stores up to d references to children and up to d − 1 keys. Each reference is considered "between" two of the node's keys; it references the root of a subtree for which all values are between these two keys." (http://www.cburch.com/cs/340/reading/btree/)
- Widely used in dtbs – given *key* of desired obj/row, index gives fast access to dtb obj/row with that key.
    - For example, in a UCT database, an index on Student Number can give fast access to the data of any student.
- Any key can be found in *log* time.
    - Automatic reorganisation on deletion and insertion ensures tree stays *balanced*.
        - Done "cheaply" as only a tiny fraction of key is changed
        - Unbalanced tree => some keys quicker to find than other
        - Balanced tree ensures no key will take longer to find than others
- Data can be accessed in ***any order***.
- Automatically reorganise themselves whenever necessary after insertion/deletion
    - Reorganisations are small, localised changes
    - Reorganisations of the entire relation file is *never necessary*.
- Tree nodes are size of a disk block to **optimise disk IO**.
    - Block: sequence of bytes/bits with a max length called a *block size*.
- All branches i.e. all the values store within non-leaf nodes are ALSO stored in leaf nodes.
    - Each leaf points to its *neighbour*.
- Each node is **always** at least half full.
    - Order of a B+ tree is the # of pointer fields in each node.

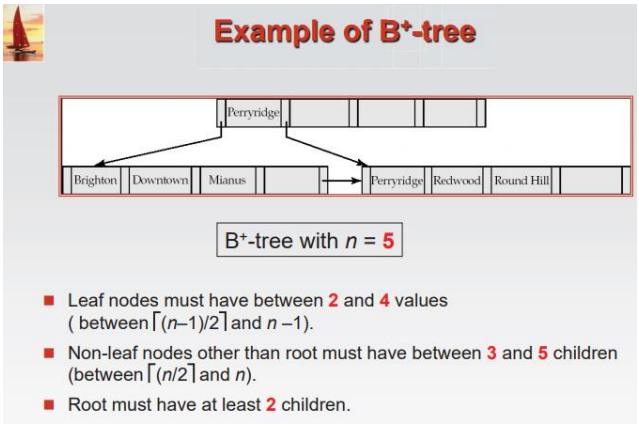
EXAMPLES



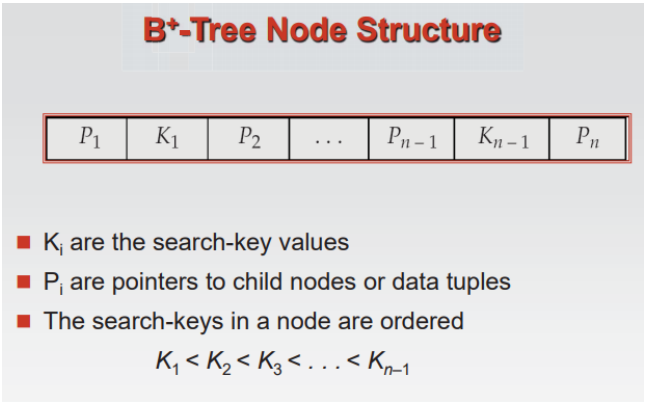
Properties

- Each node is **always** at least half full.
- Each node that is NOT a root/leaf has between [n/2] and n children
  - Note: [] here denotes "ceiling of" i.e. always rounds up to nearest integer.
- Leaf node has between [(n-1)/2] and (n-1) values.
- The root, if it is not also a leaf, has at least 2 children.
- **Balanced**- all paths from root to leaf have same length.
- Path to leaf is not longer than: $[\log_{[n/2]} K]$
  - K denotes total # of keys (values) in B+ tree.
- In a dtb, each B+ tree node is the size of a disk block/ multiple thereof
  - Every time access anything from disk, it reads a disk block at a time – typically 4000 bytes in size.
  - There are usually about 50 to 100 pointers and 50 to 100 children per node, not just 3 or 4 as in the examples.
  - So even when there are millions of keys, ***the B+-tree will not be very deep***. Here we see the difference in depth compared to binary search trees which are so many more levels deep because they have only 2 pointers per node.

EXAMPLE



Node Structure

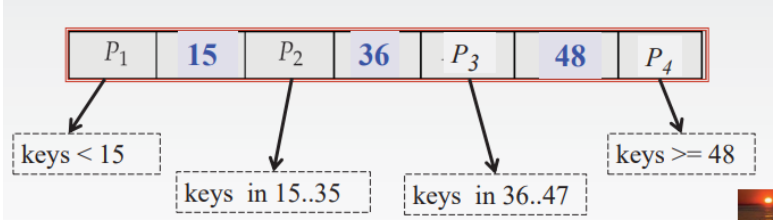


- In a B+ tree of order N, each node has N pointer fields and (N-1) key fields.

NON-LEAF NODES

- All keys in subtree to which P1 points (see diagram above) are <u>less than K1</u>
  - Would be waste of space storing 1st key because it gives no new info – P1 points what is less than K1
- All keys in subtree to which Pn points have values greater than/equal to Kn-1.
- For 2 < j < n-1, all keys in subtree tp which Pj points have values <u>greater than/equal to Ki-1 and less than Kj</u>

EXAMPLE



Find

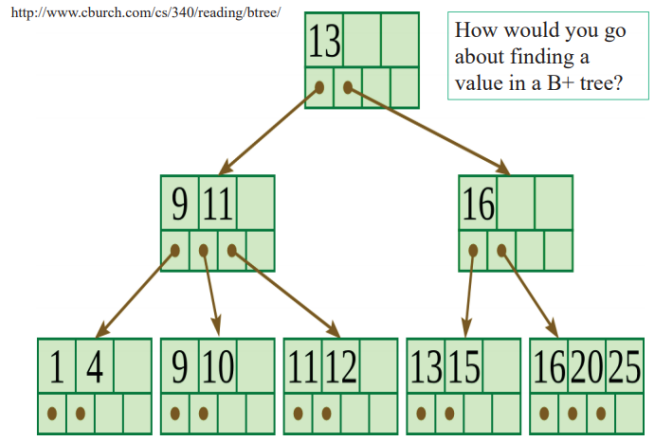
- Find (V) for tree of order M
- In tiny B+ tree, can find value just by looking through leaves.
    - Not feasible for large B+ tree that have e.g. millions of values in the leaves.
        - Much quicker for you to use the properties of the B+ tree to find specific value.

ALGORITHM

1. N = root
2. While N is NOT a leaf {
    a. Let Kj = smallest key in N greater than V
    b. If (no such K exists) N = Pm
        i. Else N = node pointed to by Kj-1 }

- Line 2b: If (no such K exists) N = Pm
    - Means if there's no key smaller than V, follow the last pointer because V is in the child with the largest value.
- Algo continues until it reaches a leaf, because it's the leaves that store the pointer to the database record for the key that needs to be found (that key is called V here).
- If find algo doesn't locate the value it searches for, value is defs not in tree.

EXAMPLE



How would you use the B+ tree properties to find 10? to find 15? to find 20?

To find 10: Start at the root. 10 < 13 so go left. There 10 is between 9 and 11 so follow the 2nd pointer, and the leaf you reach is where 10 is.

To find 15: Start at the root. 15 >= 13 so go right. There 15 < 16 so go left, and the leaf you reach is where 15 is.

To find 25: Start at the root. 25 >= 13 so go right. There 25 >= 16 so go right, and the leaf you reach is where 25 is.

Indexes

CREATE UNQIUE INDEX index_name
ON table_name(column1, col2,…,col N)

(UNIQUE would be omitted if duplicate keys are expected in the index.)

- Index is in col1,…,col N order => quick to search for specific values of col1,col2, etc
    - Also quick to search for specific values of col1 OR col1 and col2 etc.
- Indexes are an overhead when there are many inserts, deletes and updates to values being indexed; so often end up dropping indexes.

- i.e. ALTER TABLE table_name DROP INDEX index_name

DROPPING INDEXES (FQ)

- When do we drop the indexes? Is it after a certain number of inserts/deletes and updates have been done or just before we want to search for something? Is it a "one time" thing (i.e. only drop it once), or after we've dropped it do we then create the index again and then it becomes a process of dropping and creating?
    - This is not something one should be doing, but a way of remedying a database design error. Some database designers will think they should have lots and lots of indexes for their data so that all kinds of queries will be fast. So for example if the data is for a bookstore, they may create a B+-tree index on title and another index on author and another index on category (e.g. Dictionaries) and another index on publisher and ....
    - But if they do that they will probably find that updating, inserting and deleting books in the database requires so many index changes that, while searches are now faster, changes are way too slow. So they then realise it is not worth having so many indexes, and so they drop some indexes and never create them again, having learnt the lesson that Too many indexes is not good either.
    - There is no rule of thumb and certainly no formula for how many indexes to have.
    - Generally a DB designer will start by just specifying an index on each primary key (because these will be looked up in a join whenever info in this table is needed in conjunction with info in some other table; and also because we normally lookup e.g. students by Student Number, accounts by account number, etc). Then generally they will create an index on a non-key field for some table if an important DB application is running too slowly and this index would speed it up.

CONNECTION BETWEEN SQL INDEXES AND B+ TREES (FQ)

- Are the indexes stored in the B+ Trees with an indexed table at each pointer in the leaf? Or are the B+ Trees the indexes themselves, and if so how does one join or group them?
    - A SQL index is usually implemented by a DBMS using a B+-tree. An index on say empID gives us fast access to a row of say an Employee table if the empID is known, right. How does saying we want an index make empID access faster? Because the DBMS keeps a B+-tree containing all empID values in the Employee table, and gets to the leaf for any empID like say 123 in logarithmic time. Here it finds 123 and a pointer to the place on disk where the row for that employee with empID 123 is stored. Without this index, the disk blocks containing the Employee table would have to be fetched one by one until the row for 123 is found - much slower.
- If a relation has 2 indexes (e.g. one on EmpID and one on EmpName) then 2 separate B+-trees are used accordingly.
- And if an index has two attributes does that just make two B+ Trees?
    - If a relation has an index on two attributes like Dept,Section then the Dept and Section values for each row are effectively concatenated into a single key that then becomes the key for the single B+-tree for that one index.

WHEN INDEXES ARE USEFUL VS NOT

- Indexes used to find rows matching WHERE conditions and for JOIN, ORDER, GROUP BY, MIN, MAX.
- Indexes can speed up searches
    - E.g. for LIKE “asdf%” but not for LIKE “%asdf%” i.e. *only if the start of the string of interest is given*
- A composite index e.g. INDEX tbl(col1,col2) can speed up searches – col1 is 1st part and thus index is kept in col1 order (and within this col1 order, it’s kept in col2 order).
    - E.g. An index ON moduleOffering (Topic, Lecturer, Year) can be used by the DBMS to quickly find a specific Offering of a given Topic taught by a specific Lecturer in a specific Year, but can also quickly find all the offerings of a particular Topic by a specific Lecturer (for each Year they taught it in order), and also to quickly find all the offerings of a particular Topic (by each lecturer & year in turn). But it will not speed up finding all offerings in a particular year nor will it speed up finding all offerings by a particular lecturer.
- For 2 indexes to speed up a join, must have values of the *same type*.
- Indexes can be used to speed up a join but only if they have values of the ***same type***.
    - , e.g. if one is numeric and one VARCHAR, the number 4 can match with “4”, “ 4”, “00004”, etc. so indexes cannot be used.
    - Indexes of type VARCHAR(16) and CHAR(16) can be used together in a join, but not VARCHAR(20) and CHAR(16)

Insert

- Insert V in tree of order M

ALGORITHM

1. L = Find(V) //leaf that should contain V
2. If (L has fewer than M-1 keys) //new key fits
    a. insert_in_leaf(L,V) //keep keys in order
3. Else { R = new node;
    a. Split(L,R,V); //put M/2 keys in L, rest in R
    b. Insert_in_parent(L,R< smallest key in R) }
- Put simply:

```
find(V);
if (there's room) insert;
else {make new node; // split
      distribute half-half;
      insert <key,ptr> in parent}
```

```
find;
insert OR
{split; recurse}
```

EXPLANATION

- The leaves must have all values in order, so Find algo is used to get the correct leaf where new value must go.
    - If there’s space in that leaf, value is inserted and it’s done.

- If, on the other hand, the leaf is full because it already has M-1 keys in it, then new node is created and in order to keep the "half-full" property of B+ trees
    - M key split half-half between old and new nodes.
    - Having made a new node, have to insert pointer to it in parent node – else new node can't be found in future.

Insert in parent

ALGORITHM

1. If (L is root)
    a. { N = new node (L,R,S) // has key value S and pointer L and R
    b. Make N the new root
    c. Return }
2. Let P = parent if L;
    a. If (P has fewer than M-1 keys) //new key fits
        i. Insert S, and pointer R, in P //key and pointer of new child.
    b. Else { U = new node;
        i. Split(P,U,S); //put smallest m/2 items in P, rest in U
        ii. insert_in_parent(P,U, smallest key in U) }

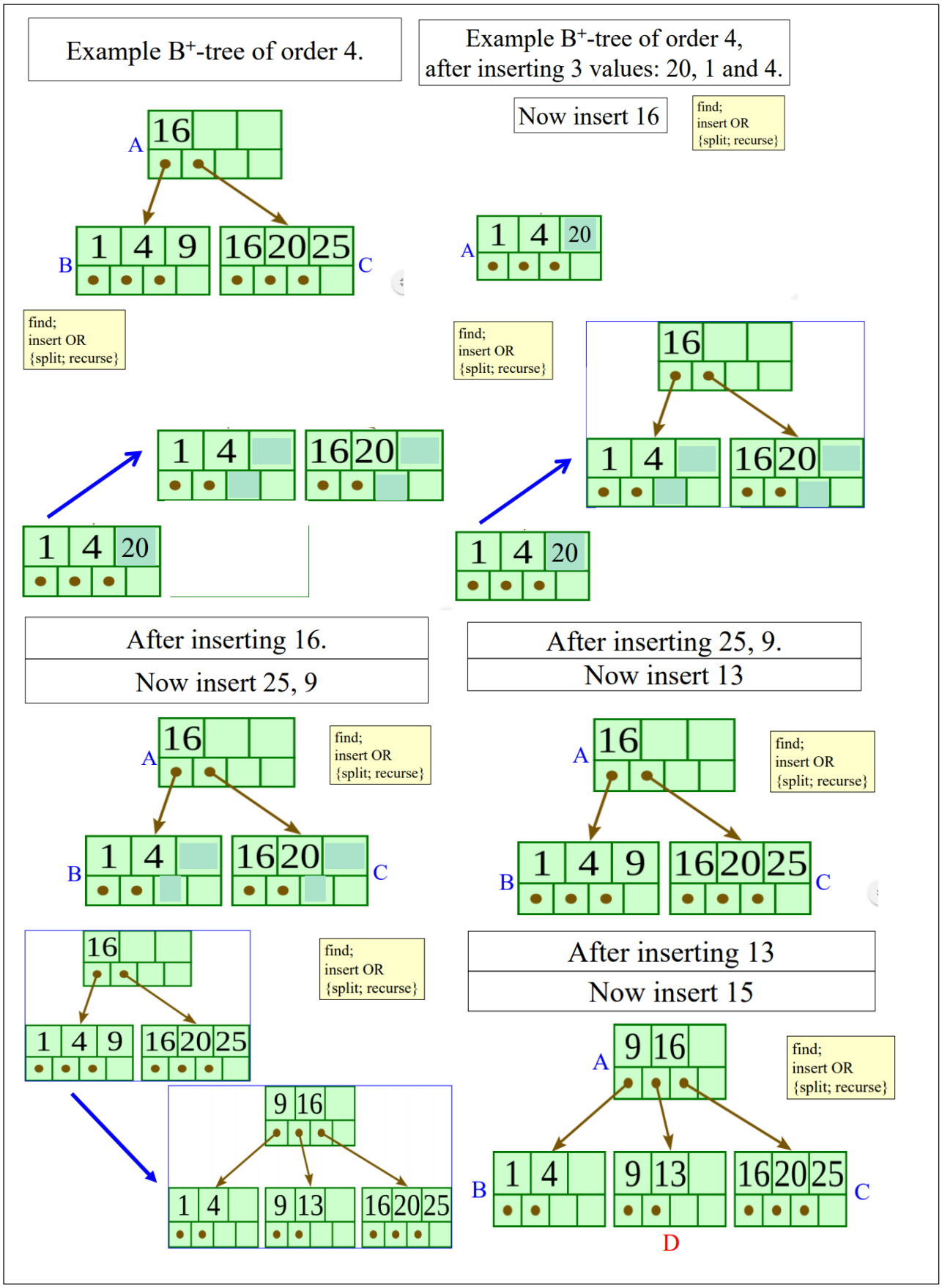
EXPLANATION

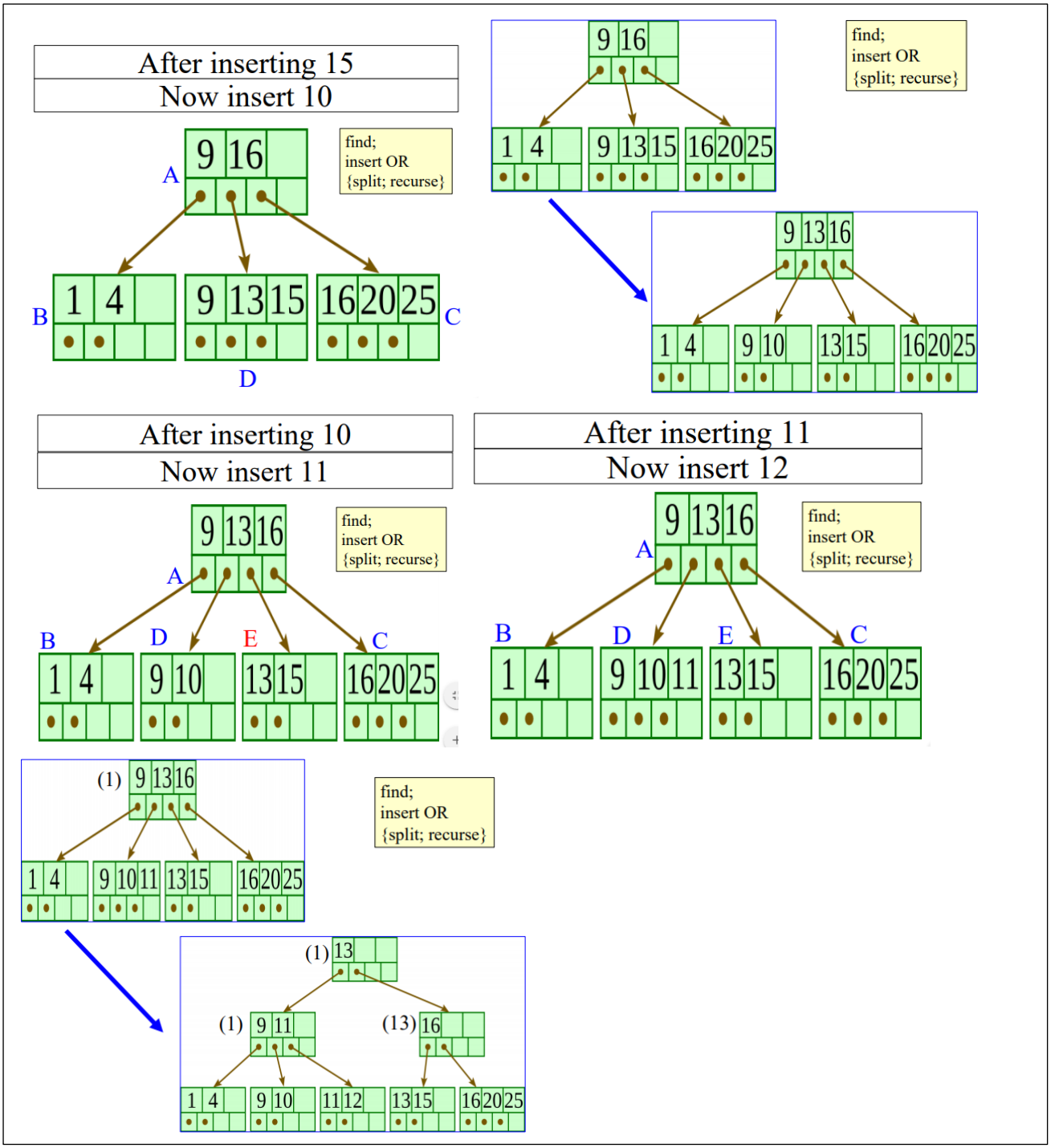
- Insert in Parent (oldchild, newChild, split value)
- Insert in parent called (line 1) because node L full and so M keys split between node L and new node R, with S being the smallest key in R.
    - There's a special case where L is only node in tree handled first.
        - L would then be root but also a leaf as it has no children.
    - In this special case, a new node must be made that points to L and R , and S is the first value in R, so S is first value that is stored in this new root.
- Otherwise, if L wasn't the root (line 2), then it has a parent, called P.
    - If there's space for the new info (key is S, pointer is R), then it's stored in P.
    - Else if P is also full – repeat procedure
        - Split half-half and call insert-in-parent again.

Splitting values between nodes

- If node being split is **leaf**, put first M/2 **keys** in 1st node, with their pointers, and then rest of the keys in the 2nd node.
- If node being split if non-leaf, put first M/2 **pointers** with their keys in the 1st node, the rest in the 2nd node.

EXAMPLE (Insertion)





EXPLANATION

- Had to insert into root node to insert 16, but root node was full, so had to make a new node and distribute values half-half.
    - 2 smaller values remains in old node and 2 bigger values go in new node.
    - Do insert in parent with 16, because 16 is smallest value in new node and so must be inserted in parent.
    - Tree only had one node => special case – no parent, so a new root is created.
- Insert 25 and 9 no problem.
- To insert 13:

- 13>16, so goes into first leaf.
- First leaf full, so split leaf and make new node – {1,4} stays in old node, {9,13} go into new node.
- Insert in parent called – 9 stored in parent 9 (smallest value in new node).

- Insert 15 no problem.
- Insert 10:
  - 10> 9 so into second child – but second child full so split half-half once more with {9,10} staying in original node and {13,15} in new node.
  - Insert into parent called – 13 stored in parent (smallest value in new node).
- Insert 11 no problem.
- Insert 12:
  - 12> 9, so into second child.
  - Second child full so split half-half with {9,10} and {12,13}.
  - Insert in parent called – 12 added to root.
    - BUT root full, so split half-half {9,12} in old node and {13,16} in new node
  - Insert in parent called once more – this time need a new root.
  - 13 gets placed in new root node and no longer in same node as 16 – remember that first value in non-leaf node is never stored.

Delete

- Delete value V

ALGORITHM

1. N = Find (V)
2. drop(N,v); //drop V and its pointer from N
3. if fullEnough(N) return;
4. if cantakeFromSibling(N) return;
5. if hasLeftSibling(N)
   a. mergeWithPrevSibling(N)
6. else
   a. mergeWithNextSibling(N)

- Put simply:

EXPLANATION

- To delete a value, need to first find V and remove V and its pointer from that leaf.
  - Note: even though the same values will appear in both leaf and non-leaf nodes, we will often not both to delete the value in both the leaf and non-leaf – will only delete in the leaf.

- If leaf is still at least half full, then B+ tree property maintained and you're done.
    - UNLESS the deleted value was the smallest in that leaf and the parent *must* be updated with the new smallest value there.
- If deleting a value from a node N makes N less than half-full, there's 4 ways to maintain half-full property.
    a. If it has sibling on its left, that can spare a value and still be at least half full, then move a value from that sibling to N
    b. If it has sibling on its right, that can spare a value and still be at least half full, then move a value from that sibling to N
    c. If it has a sibling on its left, then move N's remaining values there.
        i. They'll fit, otherwise (a) would have succeeded.
        ii. Delete N altogether.
    d. If it has a sibling on its right, then move N's remaining values there.
        i. They'll fit, otherwise (b) would have succeeded.
        ii. Delete N altogether.

Take from sibling

ALGORITHM (CANTAKEFROMSIBLING(N))

1. If leftSiblingmoreThanHalfFull(N, L)
    a. { move last <key,ptr> entry in L to N;
    b. Update N's key in parent;
    c. Return true; }
2. Else if rightSiblingMoreThanHalfFull(N, R)
    a. { move last <key,ptr> entry in R to N;
    b. Update R's key in parent;
    c. Return true; }
    d. Note: taking value from right sibling means that you're taking the smallest value in that node.
        i. If that sibling is NOT a leaf, then that smallest value isn't there
        ii. In that case, smallest value can be found in parent – remember that first value in non-leaf node is never stored.
3. Else return false; //couldn't take from sibling.

EXPLANATION

- Moving a *value from a sibling node to node N* because N needs another value – in order for N to remain half-full because one of N's values has just been deleted.
- When moving a value from a sibling node to node N, **parent must be updated** accordingly.
    - If value V moves from *sibling on left to N*, V is new smallest value in N, so key associated with N in parent must change to V.
    - If value V moves from *sibling on right to N*, **V is no longer the smallest** value in sibling node, so key associated with that sibling in parent must change to whatever is the smallest value in that sibling.
- When a node N becomes less than half full after a deletion, and it can't take a value from either of it siblings, then it MUST merge with one of its siblings.

Merging with siblings

ALGORITHMS

With Previous (mergeWithPrevSibling(N))

1. Move contents of N to its left sibling;
2. delete N's pointer and key from parent // recurse

With Previous (mergeWithNextSibling(N))

1. Move contents of N to its right (next) sibling;
2. delete N's pointer and key from parent // recurse
3. Note: taking a value from the sibling on the right, it has to be the smallest value in that sibling, and if that sibling is not a leaf then that smallest value is not there!
    a. But in that case the smallest value can be found in the parent.
    b. In other words – for nonleaves, get missing 1st key of righthand sibling from parent.

EXPLANATIONS

Previous

- Merging with its previous (left) sibling
    - Whatever the smallest value in that sibling is still the smallest value in that sibling
        - Have to delete <key, ptr> entry for N from its parent.

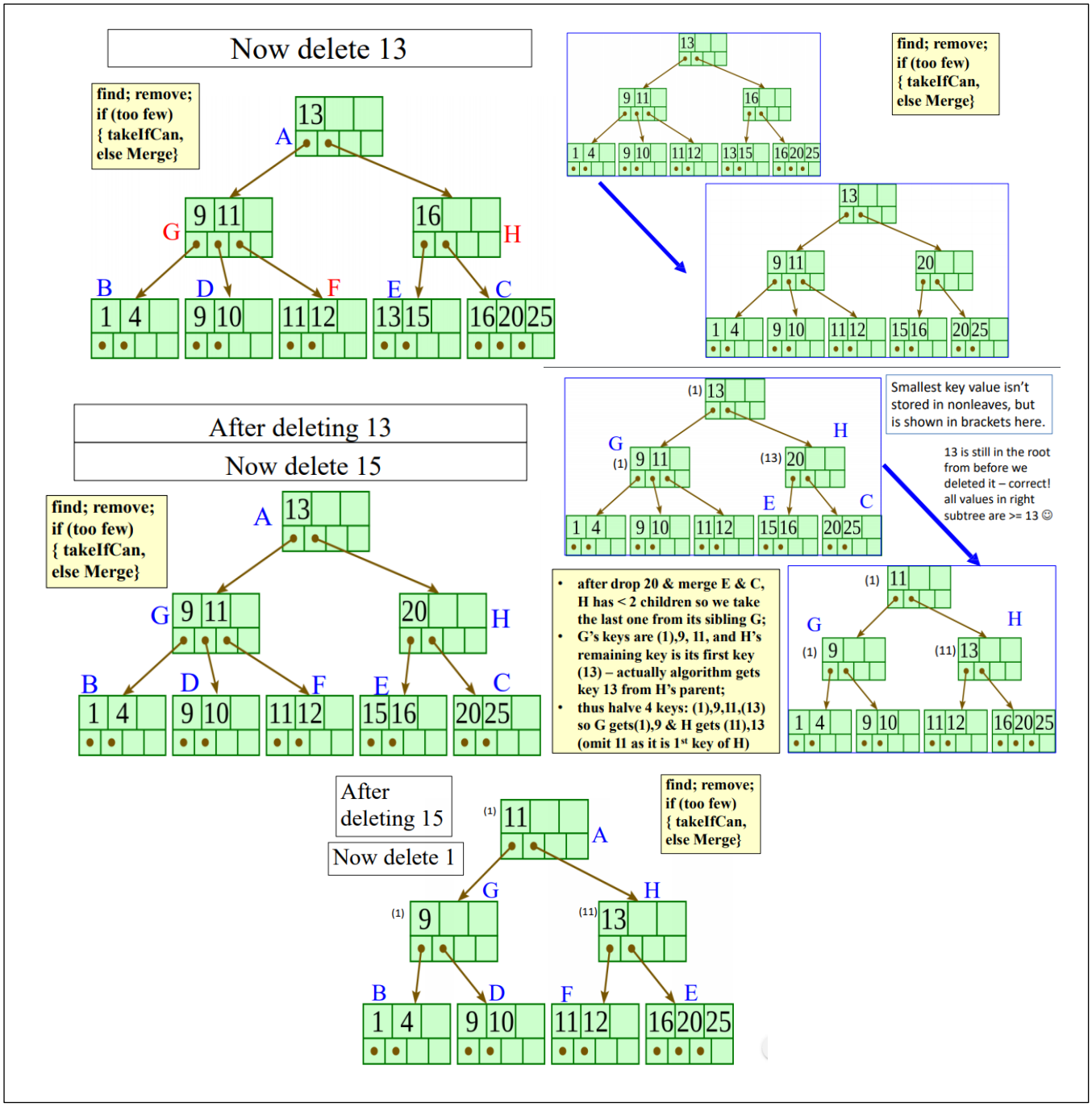
Next

- Merging with its next (right) sibling
    - Smallest value in N will become smallest value in combined/merged node, so N's pointer and key are deleted from parent and key for its right sibling is updated in parent.

MERGING NODES

- If a **leaf** has fewer than [(n-1)/2] keys, it must merge with its sibling
- If a **non-leaf** node has fewer than [n/2] pointers, it must merge with its sibling
- Usually merge with previous sibling because it's easier – if there's none then next sibling is used.

EXAMPLE



EXPLANATION

- Find value 13 and reach node E – remove 13 and pointer for database record for key 13.
- Node E now has only one value – half full property violated.
  - Take value from left sibling – does not exist
  - Thus, take value from sibling C
    - C has 3 values, so can take without violating half-full (HF) property.
  - E becomes (15,16) and C becomes (20,25)
  - New smallest value in C is 20, so parent H is updated to replace 16 by 20.

- 13 is no longer in the leaf, but it is still in the root
    - No point wasting effort changing nodes higher up, because without doing so the tree is still a valid (correct) B+ tree
- Find value 15 and reach node E once more - remove 15 and pointer for database record for key 15.
- E is no longer half-full – sibling C isn't full enough to take value from
    - Have to merge E and C, since C is E's only sibling, and delete pointer from H.
    - E and C now one node with (16,20,25)
- Deleting from H means it now only has one pointer – violates HF property
    - Have to take from its sibling G.
    - G has 3 pointers – so can take.
    - G has values (1,9,11) and H just has 13.
    - Split half-half – G becomes (1,9) and H becomes (11,13)
        - Remember, don't store first key value in non-leaf nodes, so G will only show 9.
        - H only stores 13 and 11 is in the parent.
- Find value 1 and reach node B - remove 1 and pointer for database record for key 1.
    - B less than half-full – sibling D isn't full enough to take value from.
    - Have to merge B and D into one node with (4,9,10)
- Parent G must be updated to contain value 4 – BUT only has one pointer left.
    - H with keys (11,13), G's sibling, is not full enough to take, so must merge G and H into one node.
        - 11 is gotten from parent
    - Keys in merged node are (4,11,13) – NB that 4 isn't stored (1st node in non-leaf …)
- Having merged G and H, must update parent A.
    - Parent only has 1 child – node with only 1 child is useless and is deleted – <u>this only happens at the root</u>
- This deletion shrinks the tree level by 1

Priority Queues

- Extra ppts used:
  - https://www.cs.cmu.edu/~adamchik/15-121/lectures/Binary%20Heaps/heaps.html
  - https://courses.cs.washington.edu/courses/cse373/06sp/handouts/lecture10.pdf

- Queues are First In First Out (FIFO)
  - PQS differ to this
- PQs provide a way of dealing with things in a specific order – in order of priority rather than time in which it arrives.
- Think of sending jobs to printer – usually placed in queue but that might not be best.
  - e.g. if a job is really important, might want to give it precedence
  - Or if the printer is free and there is either several 1page jobs to do, or one 100 page job, it makes sense to do the one page jobs first (even if they weren't the first job submitted).
- Priority Queue: a data strc that stores tasks based on some priority and supports access and deletion of the min and max item (priority metric).
  - Good for apps/situations in which some partial ordering is required and where access to the max/min item needs to be done quickly (0- high,1,2,… low)
  - Performance goal is for opns to be "fast"
- Priority metric: can be either max (larger vales are more urgent) or min (lower values more urgent).
- PQ algo work the same, it's just that they'll use different operators (greater than, less than) depending on the metric.
  - Example: a hospital has a severity of injury rating from 1 (minor injury) to 9 (dying) and will always choose the max

Why study PQs

- Prioritisation: adhering to real-time constraints
- Ordering ops: minimise execution delays
- Efficiency: need for data strcs that are computationally efficient.

Operations

- PQS support opns on a set S.
  - Insert (S,x): inserts x into the set.
  - Max(S): returns the max element in S
  - Extract-Max(S): removes and return the element of S with the largest key.
    - If several items have same max, any of them could be extracted.
  - Increase-key(S,x,k): increases the value of x's key to the new value k (k is assumed to be as large as x)
- Similar opns possible with **min** e.g.
  - Min(S): returns the minimum element in S

- Extract-Min(S): removes and return the element of S with the smallest key.
  - If several items have same min, any of them could be extracted.

Applications

- Priority criterion: typically decided by the app/scenario for which app is designed.
- Example: (Job Scheduling in OS)
  - PQ holds jobs to be performed and their priority values as the jobs arrive
  - When job is completed/ interrupted, highest priority job is chosen.
  - Scheduler ensures the highest PQ job is at the head of a queue (new jobs can be added).
  - Aim: avoids delays and deadlocks.

Implementation strategy

OPTION 1: UNORDERED LINKED LISTS

- Remember that PQ metric is max or min.
- Insertions (at front) are O(1) – so, constant
- Searching and/or deleting the min/max O(n) – so, linear
  - On avg will find max/min about halfway through so, on avg n/2 opns.
- Problem: finding and/or deletions of max/min requires a linear scan of list.

OPTION 2: ORDERED LINKED LIST

- Ensures list is always sorted
- Makes for cheap access (find) and deletions – O(1)
- *However*, insertions still require scanning the list (linear scan) i.e O(n).
  - On avg, have to go n/2 (halfway) through list before can insert item in correct place.
- Not really much better than unordered list.

OPTION 3: BST

- Gives O(log n) avg running times for find, delete and insert opns
- Better than scanning through linked list.
  - Avg run time is O(log n) – much better than O(n) especially when n is large.
- Problems:
  - Input not sufficiently random – BST can be very unbalanced, with some branches much longer than others.
  - Can lead to O(n) (linked list) running time for opns.

OPTION 4: BALANCED BST (LIKE AVL)

- Implementation is cumbersome/more effort than it's worth and complex.
- Balanced trees are designed to be able to quickly find *any* value.
  - PQ doesn't need that capability, only needs to extract-max/min
    - Doesn't need to be able to find any other value.

OPTION 5: BINARY HEAP (FOCUS)

- PQ data strc
- Compromise between queue and search tree.
- Classic approach to implementing PQ.

- Performance is a compromise between constant-time queue and log time set.
    - Basic PQ supports all opns in O(log n) WC time, uses only an array, supports insertion in constant avg time, is simple to implement and is known as BINARY HEAP

Binary Heap

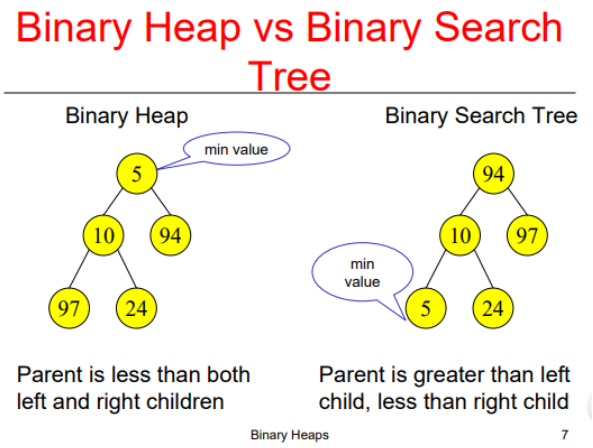
- BT with two properties:
  - Structure property: if a complete BT is used all algo executions need to maintain this data strc.
  - Heap order property: if max/min element is at the root this should always be true.
- Efficient: allow insertion (new items) and deletions (max/min element) in log WC time (balanced tree).

The reason we re-implement a priority queue is to improve its efficiency. When we implemented a priority queue with an array or a linked list, the efficiency of some operations were O(n).

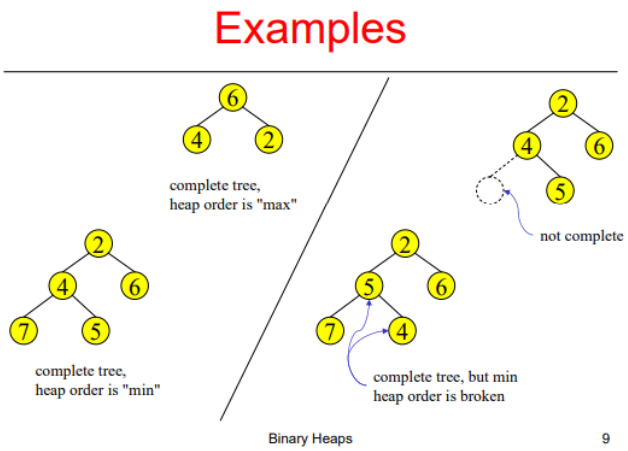
	insert	deleteMin	remove	findMin
ordered array	O(n)	O(1)	O(n)	O(1)
ordered list	O(n)	O(1)	O(1)	O(1)
unordered array	O(1)	O(n)	O(1)	O(n)
unordered list	O(1)	O(n)	O(1)	O(n)

Using a binary heap, the runtime of both the deleteMin and insert operations is O(log n).

	insert	deleteMin	remove	findMin
binary heap	O(log n)	O(log n)	O(log n)	O(1)

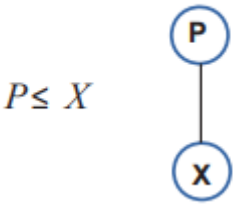


- Some examples: BH vs NOT BH



Ordering property

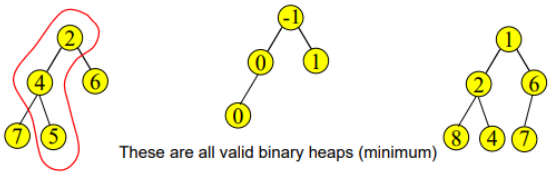
- Heap-order property: in a min heap, for every node X with parent P, the key in P is smaller than/equal to key in X
    - (P <= X)
    - In min heap, the smaller the value for an item, the more important that item is
        - So, want small values higher up => quicker to reach
        - Every parent is smaller than/equal to its children i.e. no child is smaller than its parent.
        - Every node is smallest value in its subtree



    - When "min" at root:
        - Ensure each parent key (P) is less than/equal to keys at two other specific (children positions)
    - That is, a complete BST with each "key' less than/equal to its two children
        - (P <= X)
- A BT is heap ordered if the key in each node is less than/equal the key's in that node's two children (if any)
    - Similar properties applies to max heap, but there P>=X
- Conversely, max heap supports access to the max.
    - Parents are >= their children.
        - Max heap, every node is biggest in its subtree

Heap order property

- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - › A binary heap is NOT a binary search tree



These are all valid binary heaps (minimum)

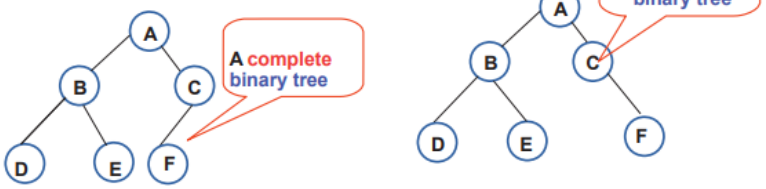
Binary Heaps 6

Structure property

- Heap is a *complete* BT
  - Want PQ to be better strc than ordinary BST i.e. want to ensure it's balanced
  - Complete BT has each lvl filled left to right, with exception of maybe the bottom last (which is still filled in left to right but may not be full).
  - Requiring BT to be complete ensures it is always perfectly balanced.
    - Since a heap is a complete BT, has smallest possible height – heap with N nodes has O(log n) height.

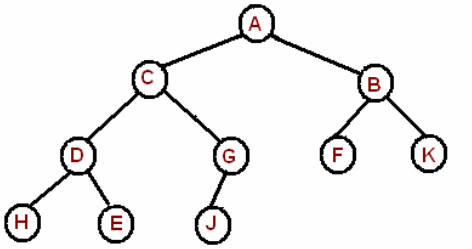
Complete Binary Tree:

Each level is completely filled, with the possible exception of the bottom level (**filled from** left to right)



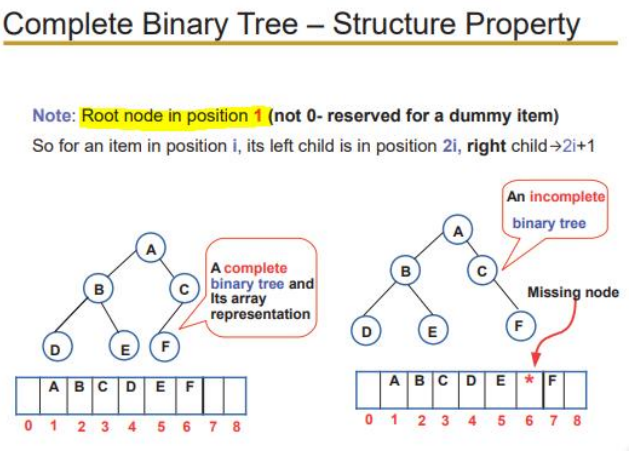
Array representation

Complete BT can be represented by storing its lvl order traversal in an array

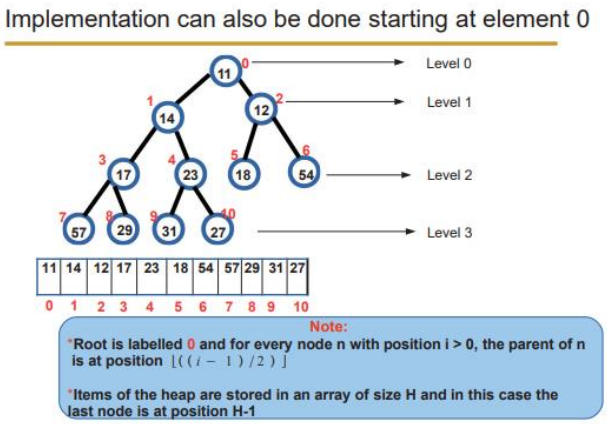


0	1	2	3	4	5	6	7	8	9	10
	A	C	B	D	G	F	K	H	E	J

- BH can be stored in arrays
    - First value is in element 1.
        - Why? Makes it easy and quick to know which elements of the array contain the 2 children on any node
        - Item is stored in position i
            - Left child is in position 2i
            - Right child is 2i+1
            - Doubling is quick for computer to do



- Can also have first value in element 0
    - Root is 0
    - For every node n with position i > 0:
        - Parent of n is at position floor[((i-1)/2)]
    - Items of the heap are stored in an array of size H



- Arrays can be used to store a tree e.g. instead of using a linked list.
    - Ensures log depth
- Advantages:
    - No child links required
    - Performance – opns required to traverse tree are simple to implement and efficient.
    - Heap entity: array of objs and an integer (current heap size).
- Disadvantage:
    - Dynamic adjustments of table size can become expensive.

Summary of BK

- Both properties: structuring and ordering need to be satisfied in ordr to avoid violating either one (opns on a heap could violate either one).
    - BH opns should *only terminate* when **both** properties are satisfied.
- For this course: will implement BHs using arrays that store 1st value in element 1.
- These BH algos can be used to implement PQS
    - Algos use both strc and ordering property and thus make PQs very efficient.
- Remember, a PQ is NOT a heap – rather, it is something that can be implemented with a BH.
    - Like how a "list" can be implemented as a linked list or array.
    - BH is just the classical, usual way of implementing PQ because it is the most efficient data strc for the task
        - BH **min** heap is a BT that is always balanced and ensures that no value in the subtree of each node is *smaller* than it
        - BH **max** heap is a BT that is always balanced and ensures that no value in the subtree of each node is *bigger* than it

Insertions

ALGORITHM

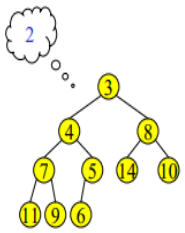
- NB: strc and order properties must always be obeyed.
- Algo:
    - https://youtu.be/t0Cq6tVNRBA : YouTube vid for visual representation
    - Create new node in tree in next available position i.e. position that has to be filled t satisfy strc property (to avoid violating strc property – complete BT).
        - Since heap is stored in array, just put it in the first unused element
    - Check to ensure ordering property satisfied
- "Percolate up" - general strategy
    - Create hole at next available location
        - If heap order isn't violated, place item in hole
        - Else "bubble up" the hole toward the root.
            - i.e. try swap it with its parent, if that isn't possible, swap places with grandparent
                - just swap places with each of its ancestors until order property satisfied

- A visual representation:

NOTE: this is NOT given in Sonia's notes

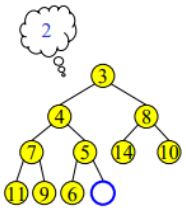
Insert

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done



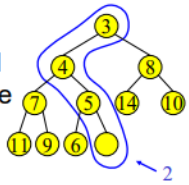
Maintain the Structure Property

- The only valid place for a new node in a complete tree is at the end of the array
- We need to decide on the correct value for the new node, and adjust the heap accordingly

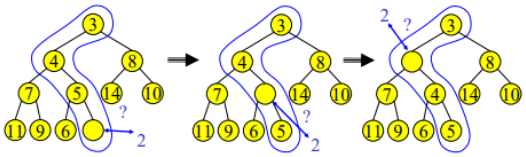


Maintain the Heap Property

- The new value goes where?
- We can do a simple insertion sort operation on the path from the new place to the root to find the correct place for it in the tree

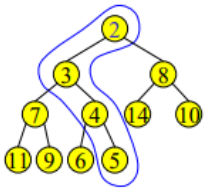


Insert: Percolate Up



- Start at last node and keep comparing with parent A[i/2]
- If parent larger, copy parent down and go up one level
- Done if parent ≤ item or reached top node A[1]
- Run time?

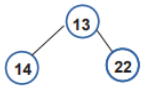
Insert: Done



- Another example:

Heap Operations – Insertion (Example)

- Consider a binary heap formed from the set {14, 13, 22}

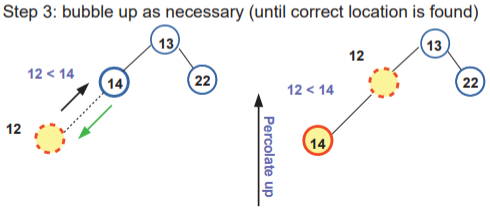


- Insert the elements {12, 11, 10, 20} into the heap above

Exercise

Note:
* All operations are aimed at finding a new slot for "12"
* Ordering and structure property must be strictly obeyed

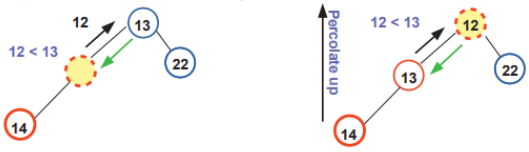
- **Case 1**: Inserting 12
- Step 1: add a node (hole) at next available location
- Step 2: compare "12" to immediate parent node
- Step 3: bubble up as necessary (until correct location is found)



Exercise

Note:
* All operations are aimed at finding a new slot for "12"
* Ordering and structure property must be strictly obeyed

- **Case 1**: Inserting 12
- Step 4: compare "12" to immediate parent node
- Step 5: bubble up as necessary

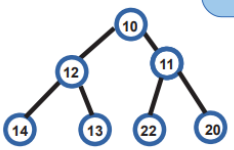


12 is finally at the correct position.
Note: structure and order properties obeyed

Exercise

- Repeat the procedure for other items
- Resulting binary heap...

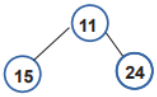
Note:
* All operations are aimed at finding a new slot for "an item"
* Ordering and structure property must be strictly obeyed



- One more example:

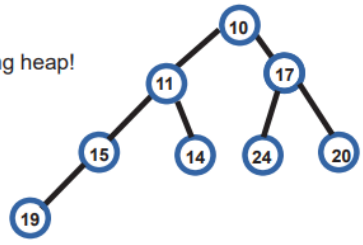
Exercise Solution – Insertion Operation

- Consider a binary heap formed from the set {15, 11, 24}
- Insert the elements {14, 10, 17, 20, 19} into the heap above

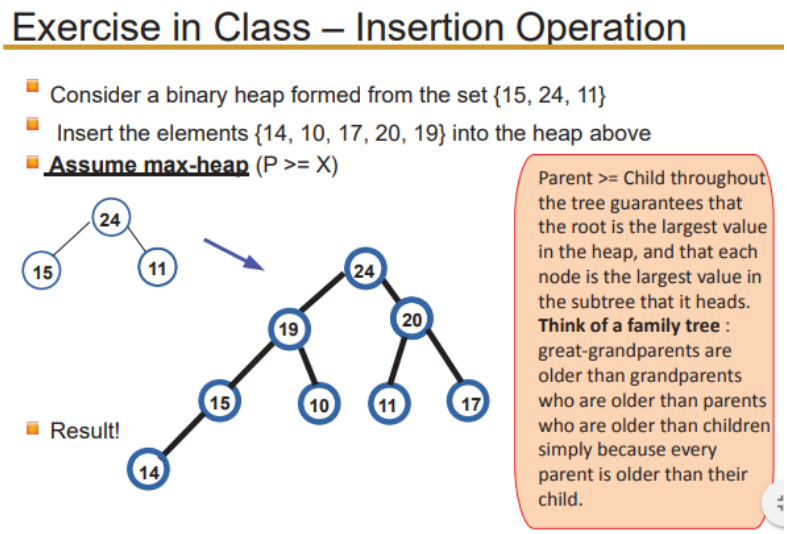


Note:
* All operations are aimed at finding a new slot for "an item"
* Ordering and structure property must be strictly obeyed

- Resulting heap!

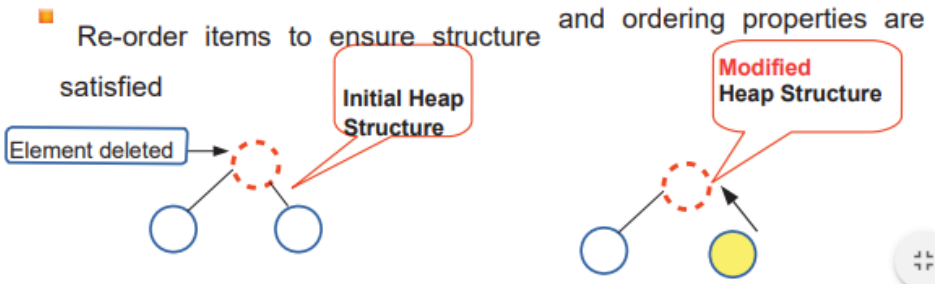


- An example with a **max heap**:

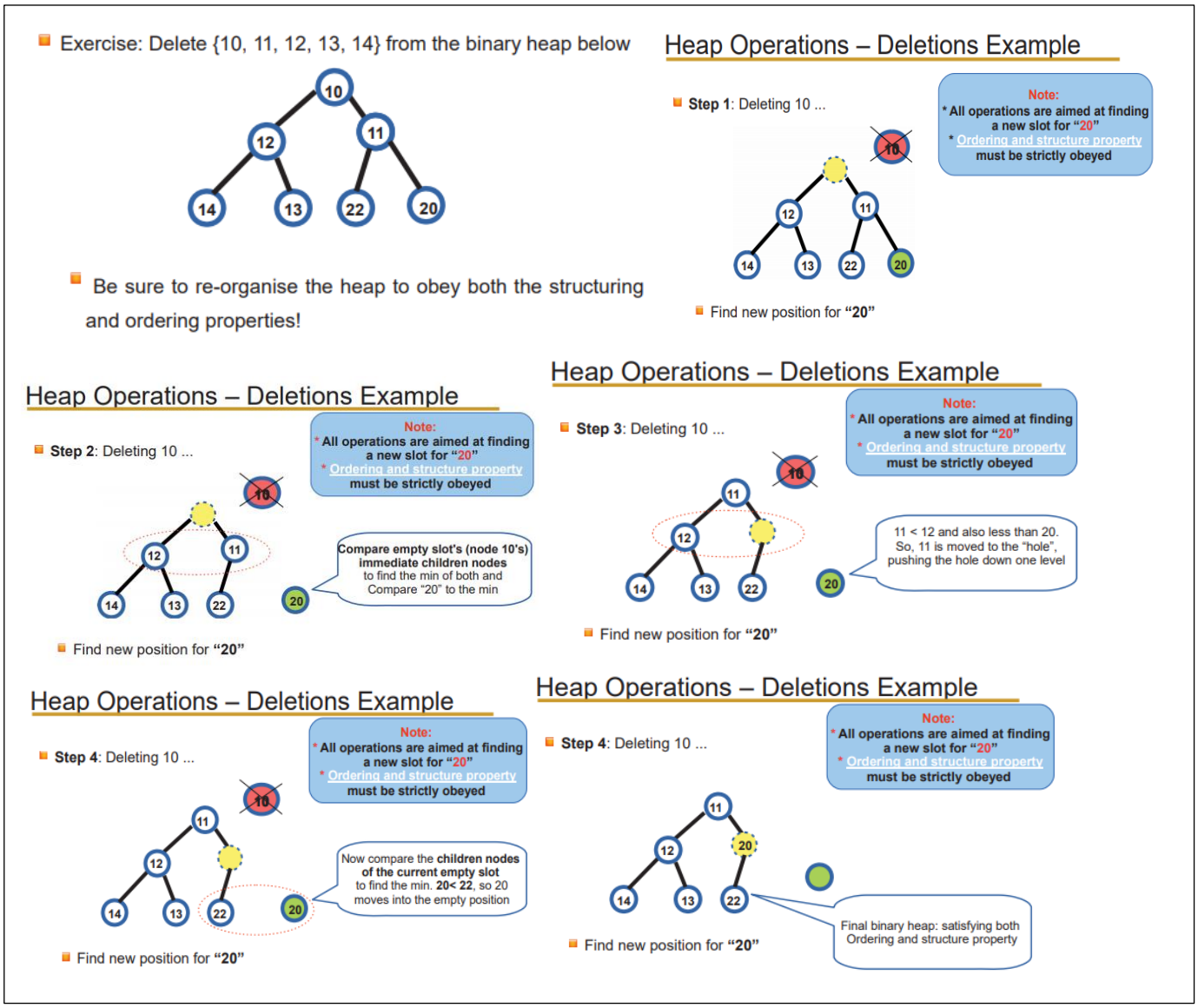


Deletions

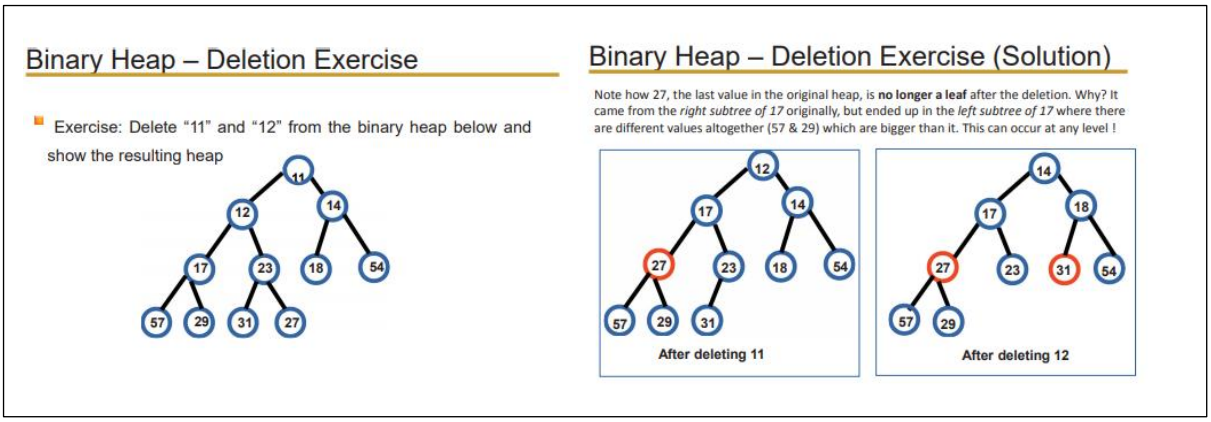
- Easy to find min/max at root.
  - But deletion of value at root creates hole at root
    - No longer have BH because it violates strc property.
  - Heap shrinks by 1 (find new item at bottom lvl for open slot)
- However, we know that since there'll be one less value in the heap, last node must be emptied, so take that value out
  - This is called the "orphan"
  - Temporary fix is to put orphan in hole so that strc property is fixed
    - Then, just have to sort out ordering property.
- Have Restructuring Principle – "Percolate down"
  - Re-order items to ensure strc and ordering properties are satisfied.



- An example:
  - For min heap, when comparing nodes while “bubbling down”, always take ***lowest*** value.



- Another example:

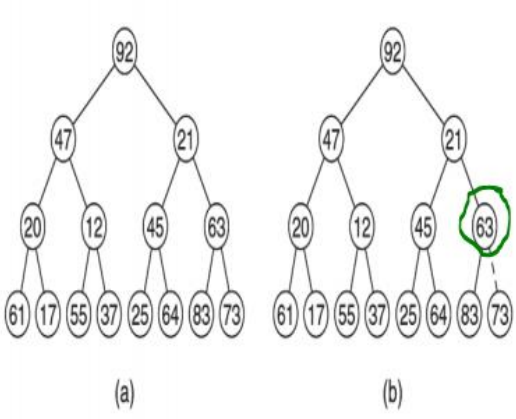


Heapify

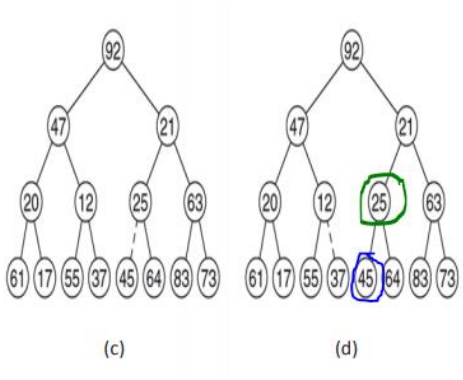
- Take BH that violates the heap order and reinstates it.
- Advantage:
  - Reduce cost on insertions from O(log n) to O(n).
- An insertion take O(log n) time
  - Implies n insertions take O(n log n) time
  - Why? Insertion algo puts new value in last position at very bottom of tree and bubbles it up until it's in the right place.
    - The worst case for this is if the new value to be inserted is the new max/min (i.e. has to go to the root) – thus has to "bubble up" through all n levels of the tree.
    - Insertions also costly as heap order *must* be maintained after every insertion.

ALGORITHM

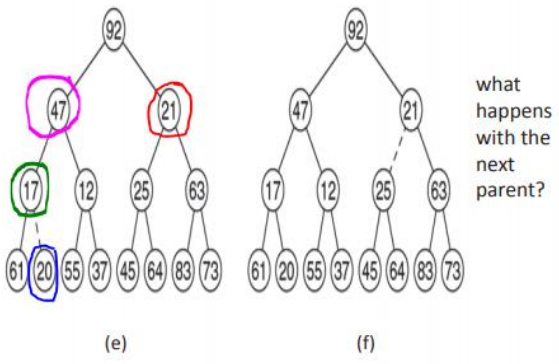
- Put given values into the array exactly in the order in which they're input.
  - The initiate bubble down process for each element, working backwards from last non-leaf (rightmost) node up to the root.
    - Non-leaves are checked but not leaves because leaves have no children and so cannot break the ordering property.
    - The criterion for checking non-leaves will depend on whether we have a min/max heap.
- By means of an example (**min heap**):
  - Step 1 goes from a to b
    - Start at last non-leaf node up to root.
      - 63 is smaller than it's children so it stays where it is.



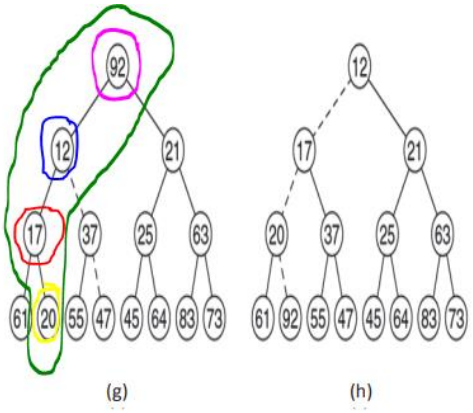
  - Step 2 goes from c to d
    - 63 needed no swaps – so move backwards to previous non-leaf
      - 45 is not smaller than its children so it swaps with 25.
      - 25 is smaller so we leave as is
    - Move to 12, which is less than both its children and so can stay where it is.



- Step 3 goes from e to f:
  - 20 is NOT less than its children so need to bubble down to keep ordering property.
    - We swap 20 with its smallest child, 17.
  - Now finished with that lvl of tree – everything from that lvl downwards obeys ordering property.
    - Move up one lvl of tree.
  - Move to the far right again, to 21 and check its ordering property.
    - 21 is less than both its children, so it stays where it is.
  - Move to 47 – it is NOT less than its children
    - Swap with 12 – must check order property once more since 47 still has children
    - 47>37 and so must swap again.
    - 47 is now leaf with no children and so it is left where it is.



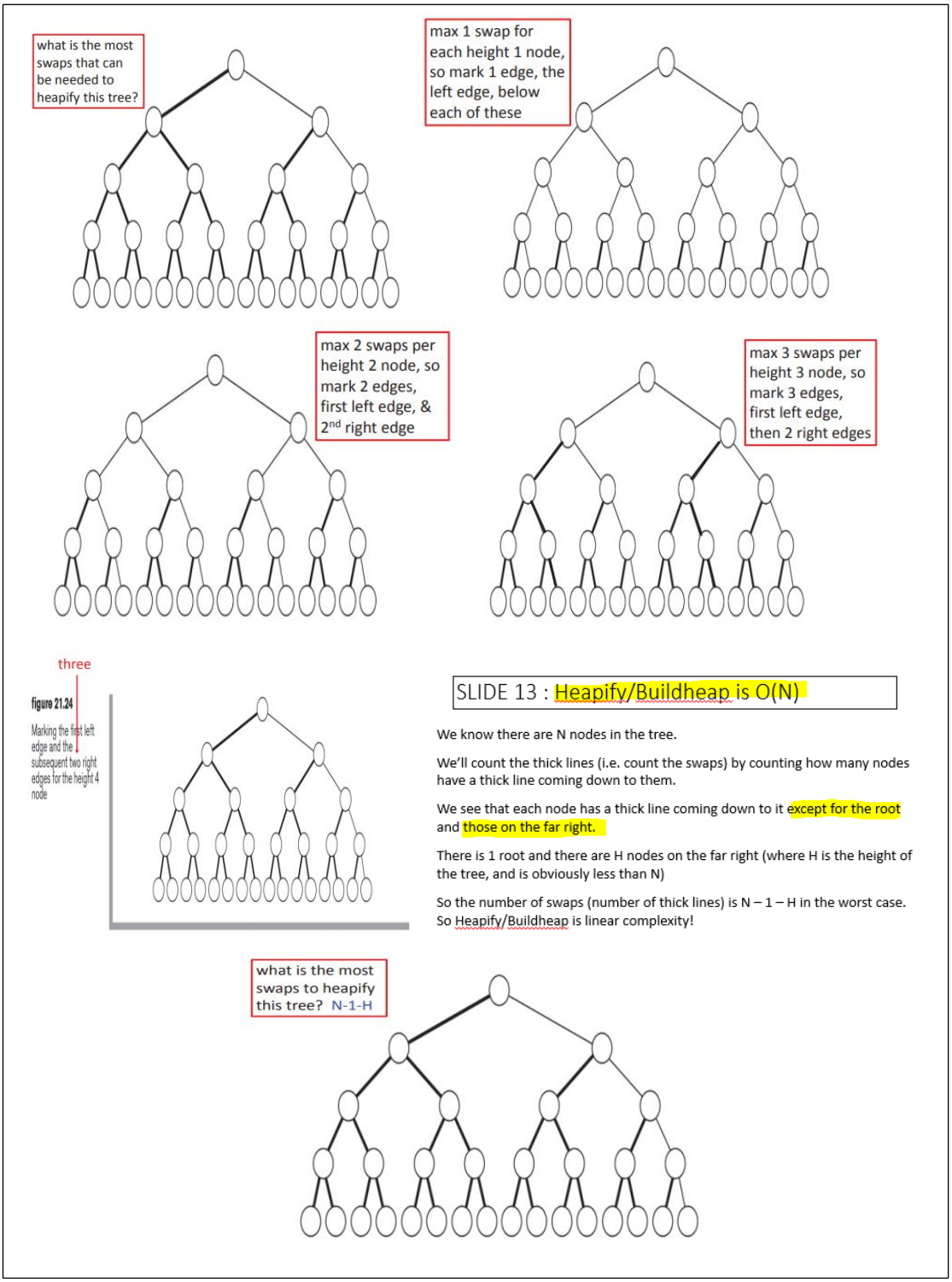
- Step 4 moves from g to h:
  - Have now finished 2 lvl of tree and move upwards (to root).
  - Move to 92 and check ordering property
    - 92 is NOT less than both its children and so it gets moved – swap with smaller child 12.
    - 92 still has children so swap once more
      - Swap 92 with 17 – still has children so process is repeated
    - 92 has children 61 and 20 now – so check ordering property and see that must swap 20 and 92.
    - 92 is now a leaf node i.e. has no children and so we can stop.
  - Everything from top downwards obeys ordering property => have a min heap.



COMPLEXITY

- O(N) in terms of number of swap operations required to make heap in WC.
    - Never work on leaves so lowest level of tree is level *above* leaves where heapify/buildheap algo begins.
    - Each node will move down 1 lvl in WC (lowest level) – so each of those requires one swap in WC
        - Can swap with either child, it's inconsequential (for the purposes of proving why O(N) is WC – has to obey rules of ordering property of min/max heap in real life).
    - After finishing lowest non-leaf level, goes to lvl above it and in the WC each of these nodes needs to go all the way to the bottom and so each requires 2 swaps
    - After finishing that lvl, algo goes to lvl above it to work.
        - The WC for each of these nodes is that they need to go all the way to the bottom and so require 3 swaps.
    - The same idea is applied to all the lvls above in the tree
        - Once we've reached the route, we're finished.
- We know there's N nodes in tree, and height of tree is H
    - Number of swaps is thus N-1-H in WC
        - Implies heapify/buildheap is linear complexity

- By means of an example:



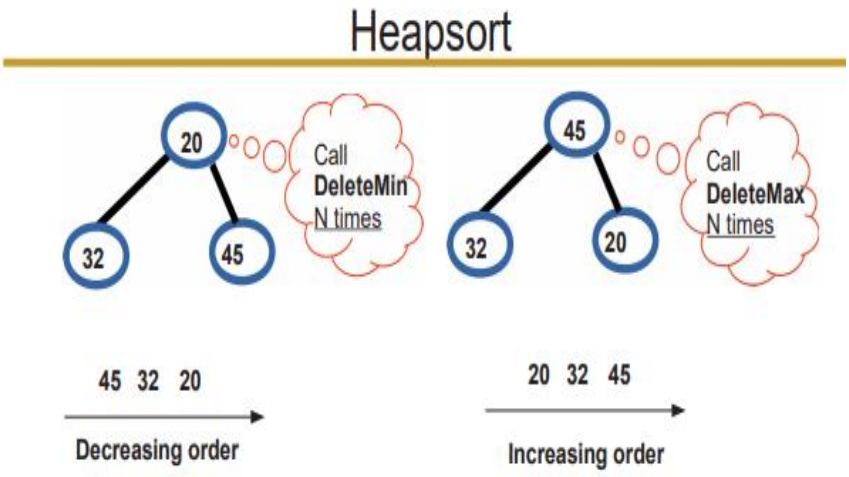
- Some notes:
  - Could have marked *any* edge for the lvl just above the leaves – just chose left arbitrarily.
    - Thereafter, especially for the higher lvls, the lines that weren't already bold were then chosen when a swap was needed (so chose between left and right for each lvl).
  - Notice how lines are far right are not bold i.e no swaps occurred there in this instance

HeapSort

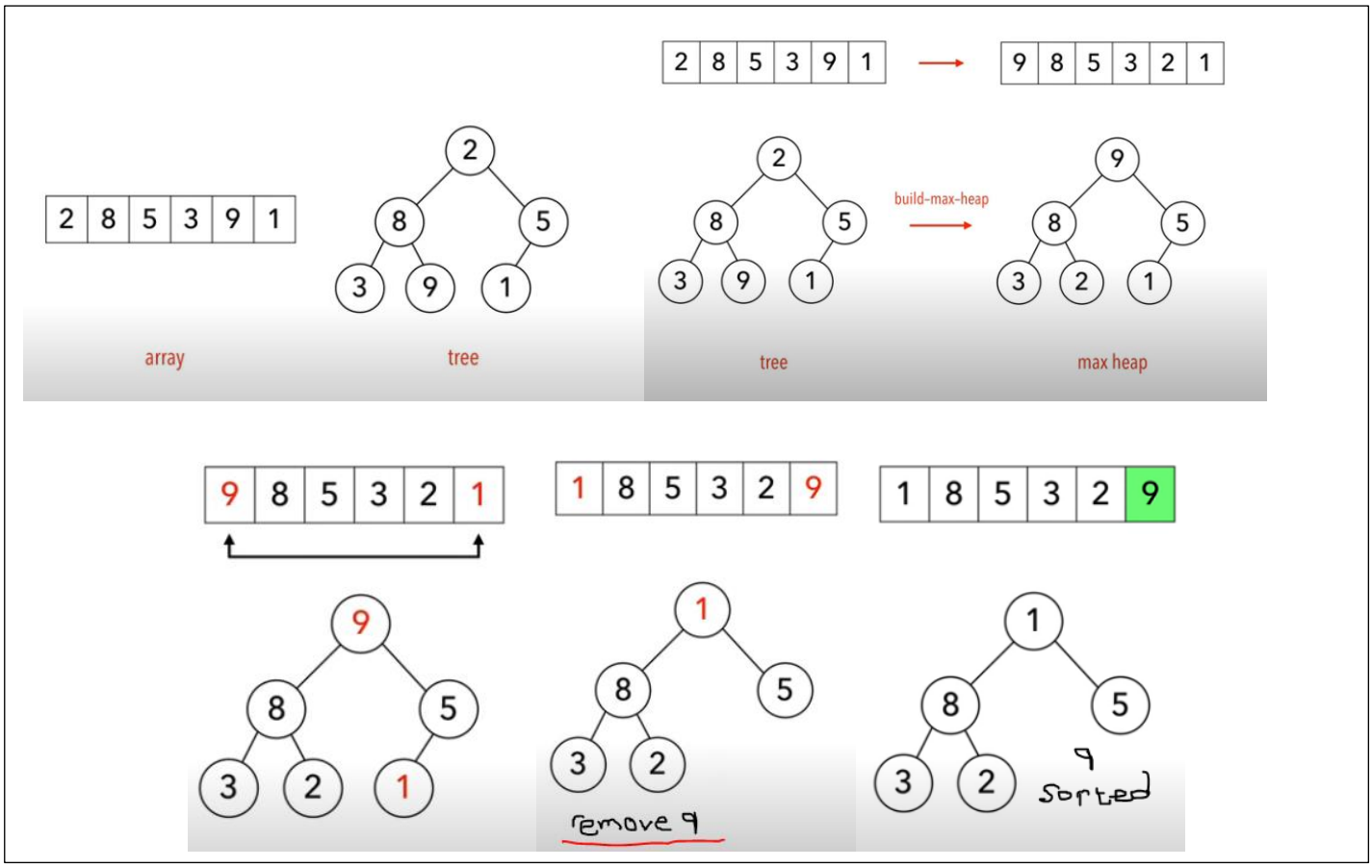
- https://www.youtube.com/watch?v=2DmK_H7IdTo (youtube)
- http://ind.ntou.edu.tw/~litsnow/al98/pdf/Algorithm-Ch6-Heapsort.pdf (notes)
- DISCLAIMER:

A warning for Heapsort section: I looked up different algorithms online and choose the one that I understood best. The method names are different to that of Sonia's notes, but I asked her on the forums if the algos i picked was okay and she confirmed it is (Vula thread for proof: https://vula.uct.ac.za/portal/site/a7dd7d20-ea3e-4096-b4f0-5ecdc36d348e/tool/8fe5970b-0d87-45f6-8740-19e0b58bf37e/discussionForum/message/dfViewThread).

- Heap designed to return smallest/biggest element each time "extract" method called, can use heap strc to efficiently sort an array of values.
- Put values in array as they're input, then call heapify and remove an element one at a time.
    - Where to put each element as it's removed from heap of size n?
        - There is one less value now, so can put it in nth position which would otherwise be empty.
- Can remove each element from min heap in this way – remove each element from heap in turn, filling up array backwards from nth place the 1st place.
    - nth value will be the smallest and 1st value will be biggest
    - If want to get vals in decreasing order, use a min heap
    - For increasing order, use max heap
- Basically, we're using the array to sort items in place if we place each value we pop (delete) in the newly opened (vacated) array element.
    - This then fills in the sorted list from back to front – use max/min heap depending on whether want items in increasing/decreasing order.



- A simple example to show this procedure (more detailed one follows later):
  NOTE: this is NOT given in Sonia's notes



- NOTE: this is NOT given in Sonia's notes
- An explanation of above example:
    - We have an array and its tree representation – first, we run heapify for a *max heap*.
    - Then we take element at root (first position of array) – 9 in this case and switch it with the last element of array – element 1 here. We consider 9 to now be sorted, so it gets removed from the tree.
    - This process of heapify, swap, remove from tree gets repeated until the array is sorted.
    - Final product:

PSEUDO CODE

```
Heapsort (A as array)
    BuildMaxHeap(A)
    for i = n to 1
        swap (A[1], A[i])
        n = n - 1
        Heapify (A, 1)

BuildMaxHeap (A as array)
    n = elements_in(A)
    for i = floor (n/2) to 1
        Heapify (A,i)
```

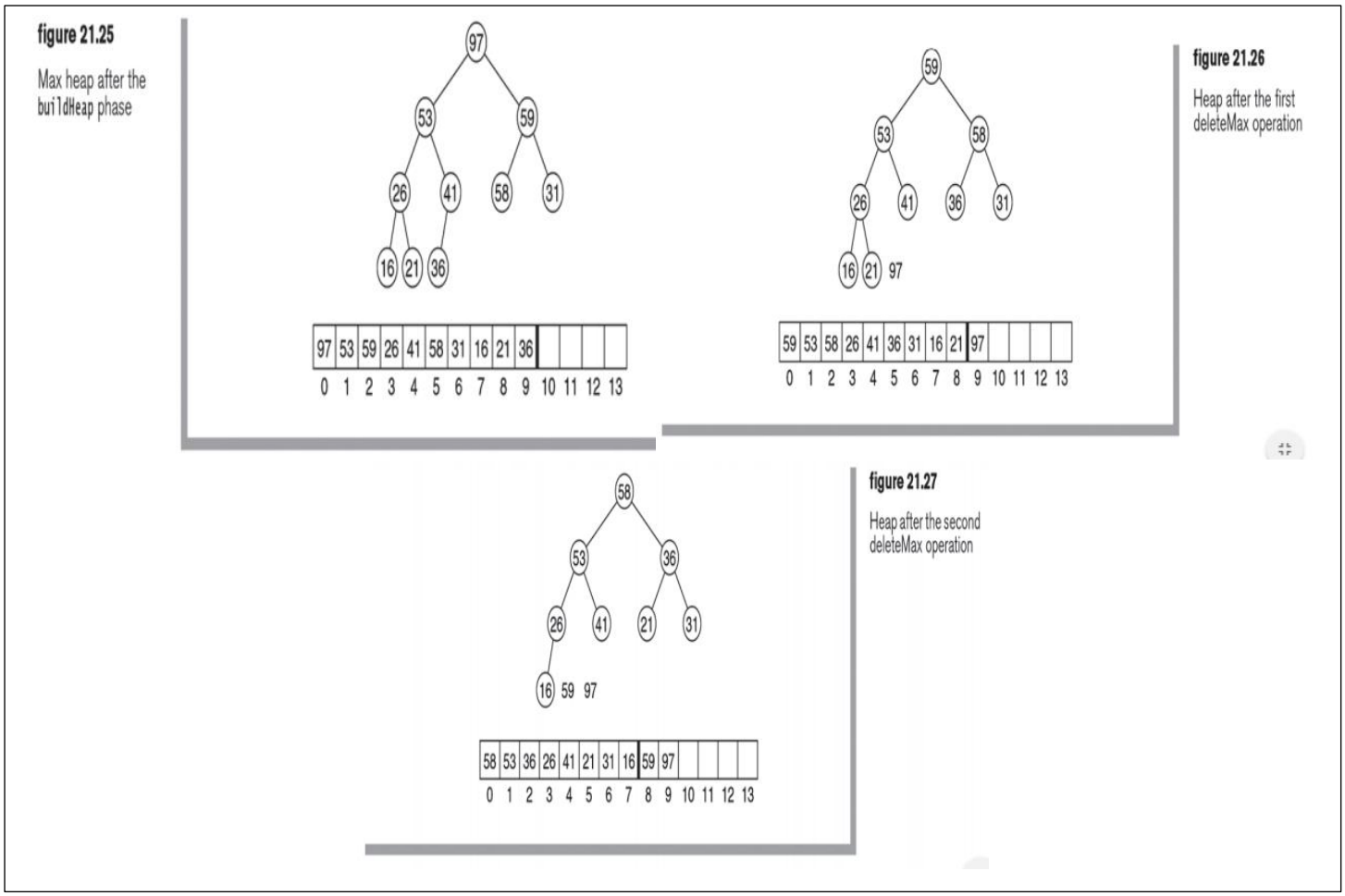
```
Heapify (A as array, i as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
        max = left
    else
        max = i

    if (right<=n) and (A[right] > A[max])
        max = right

    if (max != i)
        swap (A[i], A[max])
        Heapify (A, max)
```

NOTE: this is NOT given in Sonia's notes

ALGORITHMS

- Heapsort algo for getting values in increasing order:
    1. Fill array
    2. Run heapify algo to create max heap
    3. Call deleteMax n times
- Heapsort algo for getting values in decreasing order:
    1. Fill array
    2. Run heapify algo to create min heap
    3. Call deleteMin n times

EXAMPLE



- An explanation of the example:
  - First delete root (97) from heap and put it where last element used to be.
    - NB that Delete algo causes orphan 36 to be removed from last position and then the orphan is placed at root and swapped downwards until order property is satisfied.
  - After removing 97, note that orphan 36 was too small to remain at the root, so it was swapped with biggest child of the root i.e. 69
    - Orphan 36 was again compared with its new children and swapped with the biggest of them until ordering property was satisfied.
      - 36 is now a leaf
  - 97, which was removed, was placed in the empty element in the array.
  - This process of deletion, bubbling down and swapping is repeated until array is sorted.

OBSERVATIONS

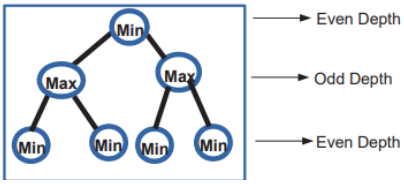
- By using empty slots in array, can perform sort in place.
- Heapsort algo is optimal for sorting values internally BUT algo is Not stable.
    - Why isn't it stable? Duplicates may not retain their original ordering amongst themselves.
        - E.g. if we are sorting person records by person name, and the "Jo Li" living at 4 Doe St came before the "Jo Li" at 8 Ivy Rd, the Ivy Rd person may come before the Doe St one
    - Why is it optimal? Its complexity is O(n log n)
        - There is a proof that sorting can't be done using fewer comparisons.
- Heapsort algo also makes an *assumption* for sorting: assumes all data will fit in memory.

Double ended PQS

- Double-ended PQ allows insertions and extraction (deletion) of either smallest/largest value.
- Double-ended PQ supports these opns:
    - Inserting element with an arbitrary key
    - Deleting element with smallest key
    - Deleting element with largest key
- Use: for systems that use PQs, want to attend to most important/urgent items first, but ALSO don't want items with low priority to wait around forever (called starvation).
    - These systems will usually extract most important item M times, and then give the least important items a chance.
- Double-ended PQ is implemented most efficiently with a min-max heap.

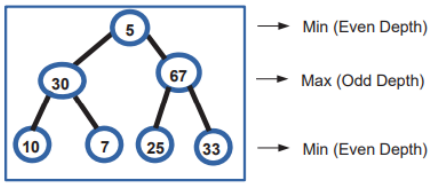
A double ended priority queue (PQ) supports the following:
Inserting an element with an arbitrary key
Deleting an element with the smallest key
Deleting an element with the largest key

A Min-Max Heap supports all of the above operations.



- Can get max-min heap
    - Similar to min-max heap but the even depths would be max nodes and then odd depths would be min nodes.

EXAMPLE (min-max heap)



- For every node X at even depth, the key stored at X is the smallest in its subtree
- For every node X at odd depth, the key stored at X is the largest in its subtree (root is at even depth)

    - For every leaf in above example:
        - Every leaf is >= its grandparent (5) and is <= its parent (e.g. 10 <30)
    - Range of values that could be stored in rightmost (last) leaf?
        - Let last leaf value be L
            - This implies 5<= L <= 67 for ordering property to be maintained.

INSERTIONS

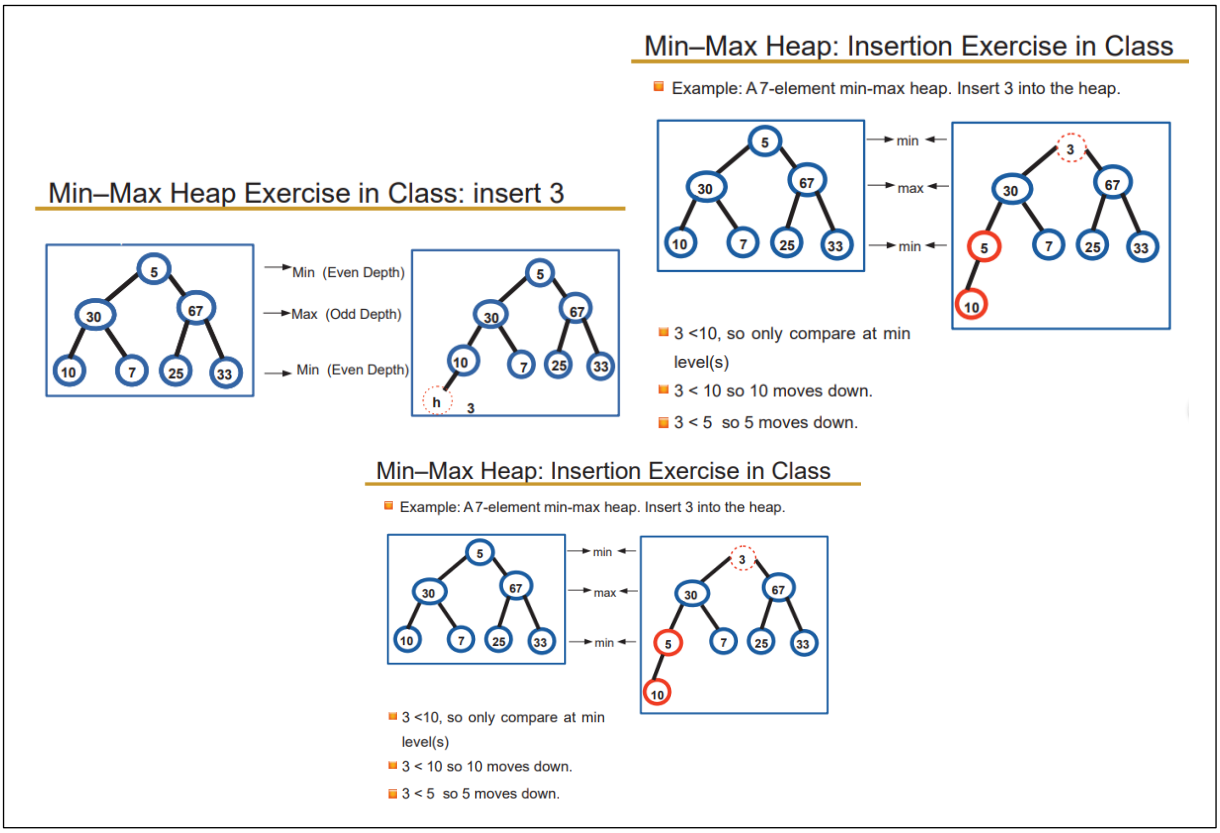
- NB that strc and ordering properties must be obeyed.
- Process:
    - Create new node in tree in nxt available position
        - Maintains strc property – ensures tree is complete BT.
    - Check to ensure min-max / max-min property is satisfied.
- "Percolate up" - general strategy
    - Create hole at next available location
        - If heap order isn't violated, place item in hole
        - Else "bubble up" the hole toward the root.

ALGORITHM

- For min-max heap
- X is new item
- Place X at nxt available location (maintains structure property).
- If heap order (ordering property) violated, "bubble up":
    - Compare X with its parent P
        - If **X < P**, then X is guaranteed < all max-level nodes above it.
            - It's true for P, so it will be true for X since X smaller than P.
            - Only check nodes on *min levels*
            - Why? X may be **too small** for where it is, since X < P.
        - If **X > P**, then X is guaranteed > all min-level nodes above it
            - It's true for P and X is bigger than P
            - Only checks nodes on max lvls
            - Why? X may be **too big** for where it is, since X > P
- Insertion requires checking ***either*** min lvls OR max lvls only – NEVER all lvls
- The above algo, put simply:

- Compare node with its parent – if it's smaller than its parent, just need to check min lvls.
- If it's bigger, just need to check max lvls.

EXAMPLE

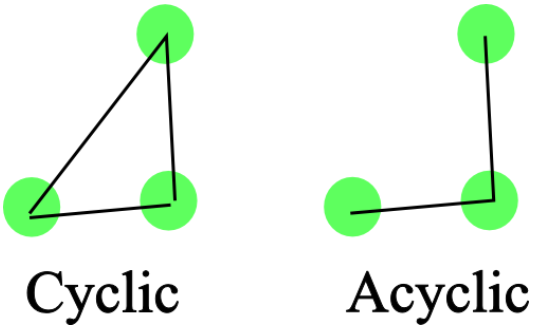


- An explanation for above example:
  - Put 3 in nxt available position
    - Gets put as child of 10.
  - Compare 3 with 10
    - 3 < 10, so must check against min lvls
  - The first min lvl above 3 is lvl 2, so compare 3 with 10
    - 3 < 10, so swap them.
  - Nxt min lvl is 0 (root), so compare 3 and 5
    - 3 < 5 so swap them.
  - 3 is now at root

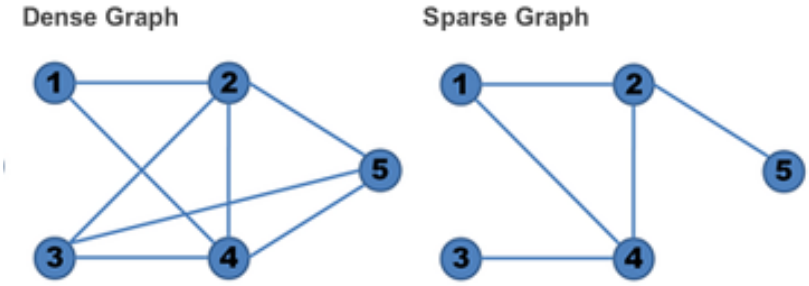
Graphs

- A graph consists of a set of vertices (nodes) and a set of edges(arcs) that connect the vertices
    - Edges should connect one vertex to another, or a vertex to itself (M)
- Loop: an edge with just one endpoint (M)
- G = (V,E) where V is the set of vertices and E, the set of edges
- An edge can have a 3rd component: a cost/weight
    - This can be in the form of time, distance etc.
- Directed graph (digraph): a graph where the edge pair is ordered
    - (u,v) => u --> v
    - Consists of non-empty set of vertices V and set of edges E that is a subset of V x V set (M)
    - Each edge has a direction (M)

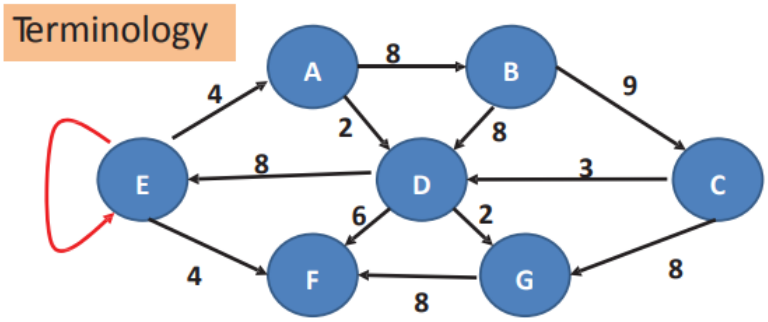
- Undirected graph: order of the vertices in the edge set doesn't matter
    - u – v => {(u,v)(v,u)}
    - No self-loops (M)
- Path: sequence of vertices connected by edges
    - X and Y are vertices of a graph – between X and Y is a motion from x and y along edges of the graph (M).
        - X is the initial vertex (M)
        - Y is the terminal vertex (M)
- Unweighted path length: the # of edges in a path.
- Simple path: path where all vertices are distinct, **except** that first and last vertices can be the same.
- A cycle (in directed graph): a path that begins and ends at the same vertex and contains *at least* one edge.
    - Vertices and edges aren't repeated (M).
    - Starts and finishes at same vertex (M).
- Cyclic graph: directed graph with *at least one* cycle



- Directed acyclic graph (DAG): directed graph with no cycles.
- Sparse graph: the # of edges leaving a vertex is much less than the number of vertices in that graph.



- A diagram to illustrate above:



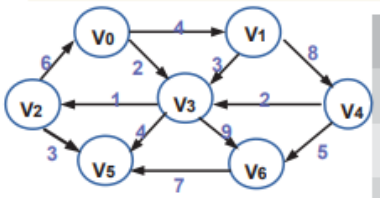
Vertex (Vertices) / Node(s)	A, B, C, D, E, F, G
Edge / Arc	(E,A) (E,F) (A,B) (A,D) (D,E) (D,F) (D,G) (B,D) (B,C) (G,F) (C,D) (C,G)
Digraph	Otherwise, it's an "undirected graph"
Cost / weight	E.g. (A,B) weight is 8
Path	E.g. E→ A → D → E → F (what is the weight of this path?)
Simple path	E.g. E → A → B → C → D → G → F (its weight is what?)
Cycle	E.g. E → A → D → E
Loop	E.g. if we had (A,A) (B,B) (C,C) (D,D) (E,E) (F,F) OR (G,G)
Acyclic graph	E.g. without (E,A) or (A,D) or (D,E) this graph is acyclic
DAG	Directed Acyclic Graph

Representation

ADJACENCY MATRIX

- Best used for dense graphs where $|E| = O(|V^2|)$ i.e. quadratic
- Can store graph using 2D array
- This representation wastes a lot of space – most graphs are spare.
- Example:
    - Notice how there are no numbers on the diagonal? That means there are no loops!

This is best for dense graphs: graphs where $|E| = O(|V^2|)$ i.e. quadratic
But our example is a sparse graph: $|E| = O(|V|)|$ i.e. linear



Graph

	V0	V1	V2	V3	V4	V5	V6
V0	∞	4	∞	2	∞	∞	∞
V1	∞	∞	∞	3	8	∞	∞
V2	6	∞	∞	∞	∞	3	∞
V3	∞	∞	1	∞	∞	4	9
V4	∞	∞	∞	2	∞	∞	5
V5	∞	∞	∞	∞	∞	∞	∞
V6	∞	∞	∞	∞	∞	7	∞

Adjacency Matrix

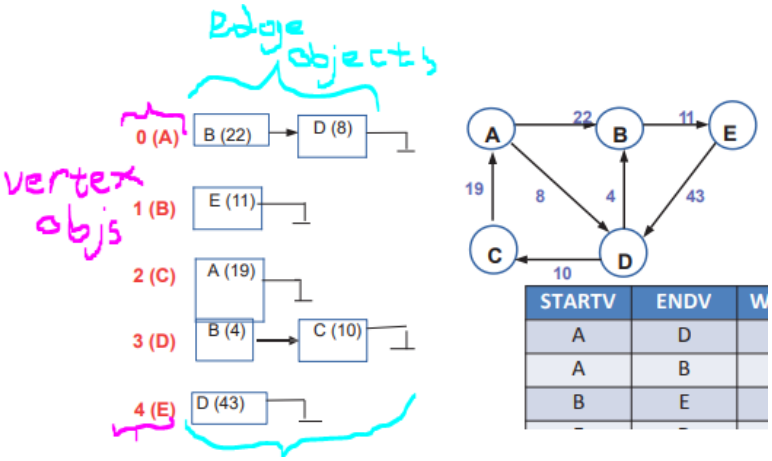
Initialisation is $O(|V^2|)$

Space cost is $O(|V^2|)$

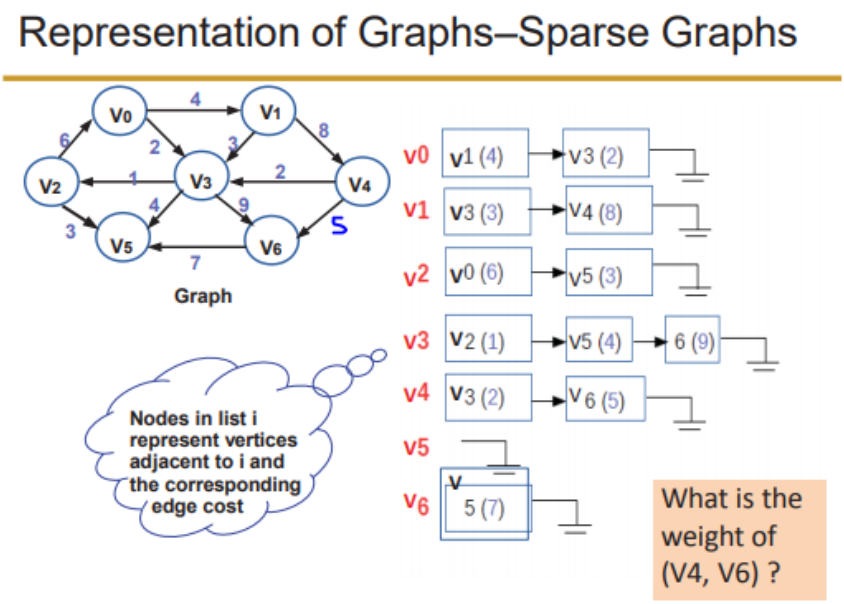
Cost of adding/removing a vertex is high

ADJACENCY LISTS

- Each vertex has a list of edges that leave from there, with the list elements showing the destination vertex and weight of that edge.
- Good for sparse graphs
- In adjacency lists, space is taken up only for edges that *actually exist* in the graph.
- Adjacency lists store **numbers** not names – so can go directly to that node's array entry next.
    - E.g. a node A will have number 0 and so be represented by 0 in the adjacency list.
- Adjacency list objects store the location of the vertex
    - Collection of the vertices would be stored in an array
    - Adjacency list vertices would store the element number of each destination vertex.
    - Also store edge objects
- To find cost of a path, start at element 0 and see what edge node it has, then go that node and so on.

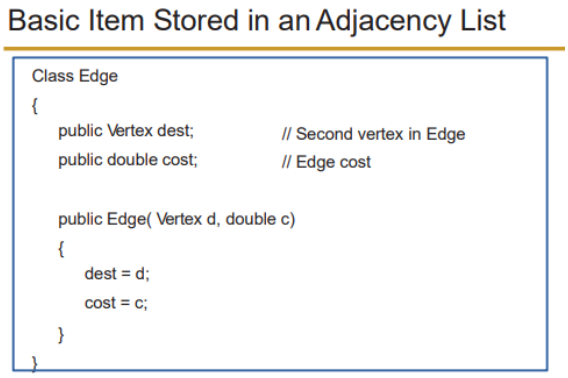


- Example:
  - The weight of (V0, V1) is 4; (V4,V6) is 5.



EDGE OBJECT

- Each obj in adjacency list contains destination vertex and weight of that edge.
- Simple code for Edge class:



VERTEX CLASS

- Need to store vertex name and its adjacency list.
- Weiss's class:

```
1  // Represents a vertex in the graph.
2  class Vertex
3  {
4      public String     name;   // Vertex name
5      public List<Edge> adj;    // Adjacent vertices
6      public double     dist;   // Cost
7      public Vertex     prev;   // Previous vertex on shortest path
8      public int        scratch;// Extra variable used in algorithm
9
10     public Vertex( String nm )
11       { name = nm; adj = new LinkedList<Edge>( ); reset( ); }
12
13
14
15 }
```

From Weiss textbook

Its adjacency list

that our algorithms will calculate

- Will need:
    - Variable to store the best cost of reaching a vertex that the algo has determined thus far
        - **dist** in this code.
    - Variable to store the vertex we came from when reaching this vertex, in the smallest (i.e. cheapest) way the algo has found so far.
        - **prev** in this code
    - Variable to store a count of how many times we've reached this vertex since the algo started running
        - **scratch** in this code.

Breadth-first search

- https://www.cs.usfca.edu/~galles/visualization/BFS.html
- https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/
- Can find shortest path in unweighted graphs i.e. the path with the fewest edges.
- Breadth-First search algo finds shortest paths between any 2 vertices.
- Implemented using queue (FIFO, not priority)
- Nodes processed in order in which we reach them.
    - Guarantees that paths we find are shortest paths.
- Can create tree when implementing algo for a visual representation.
- Because there's no weights, sooner reach a node ==> cheaper it is to reach from start node
    - Process vertices in order in which reach them – no possibility of getting to any node later on by a cheaper path , because future paths are longer.

TERMINOLOGY

- Adjacent: two vertices that are connected by an edges (M).
    - E.g. if W is adjacent to V, there's an edge from V → W.
- Note that 'node' and 'vertex' used interchangeably.

Algorithm

1. Mark source node (src vertex) as reachable in 0 steps
2. Mark all nodes reachable in 1 step (frm src, in 1 step)
    a. All adjacent nodes marked with 1
3. Mark all nodes reachable in 2 steps
    a. All nodes marked with 1 have their adjacent nodes marked with 2
4. Mark all nodes reachable in 3 steps.
    a. All nodes with a 2 have their adjacent nodes marked with a 3.
5. Etc
    a. Continues until no new nodes can be marked, as nothing can change after that

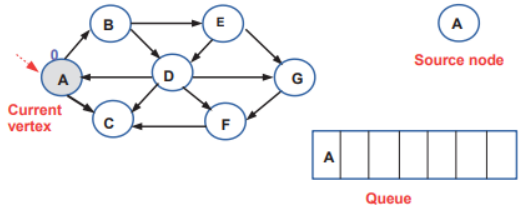
PSEUDO CODE

- All nodes have distance set to infinity before algo runs

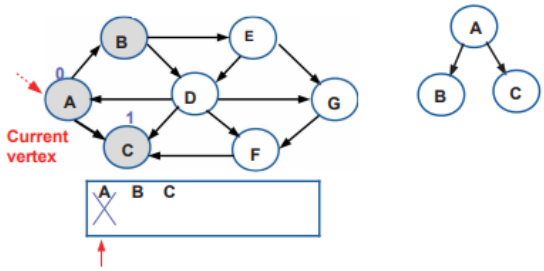
```
getStartVertex() srcVertex
if srcVertex is null
        throw exception
else
        create new LinkedList queue
        add srcVertex to queue
        startDistance = 0
while queue not empty
        current vertex V = remove item from queue
        for (edge E to V adjacent)
                next vertex W = E destination
                        if W distance == INFINITY
                                {
                                        W distance = V distance +1
                                        W previous = v
                                        add W to queue
                                }
```

EXAMPLE

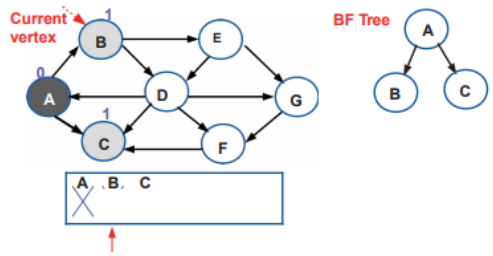
Breadth First Search - Example



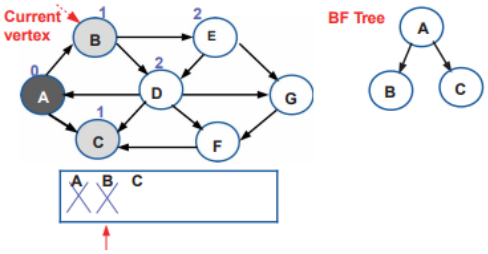
Breadth First Search - Example



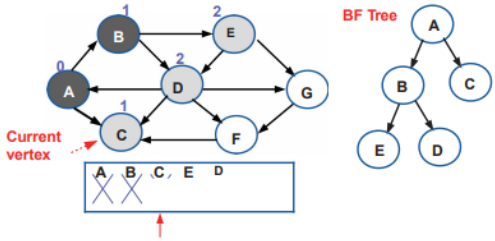
Breadth First Search - Example



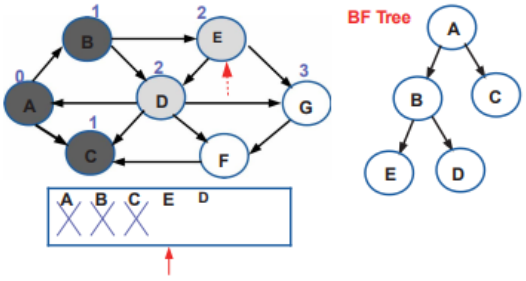
Breadth First Search - Example



Breadth First Search - Example

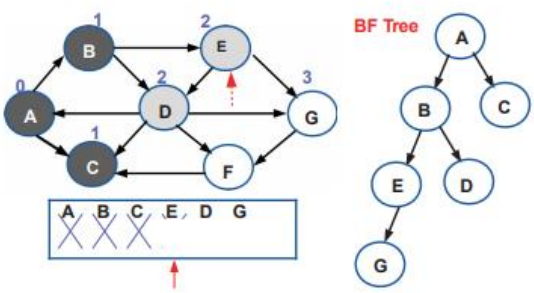


Breadth First Search - Example

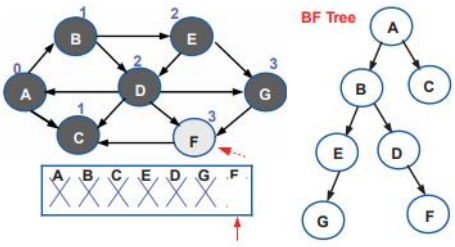


Final

Breadth First Search - Example



Breadth First Search - Example



EXPLANATION

- Nodes adj to A are B and C, so they're marked with 1.
  - Remove A frm queue since it has been dealt with
  - Place B and c in queue, since they're been reached.
- General convention is to *queue in alphabetical order*.
- Process is repeated using whatever node is at **the front** of the queue
  - When the queue is empty, we're finished.
- Next, B is popped off the front of the queue.
  - Handle nodes adj to B that have NOT been reached before.
    - Mark those nodes with the (cost of B) +1
    - Add them to back of queue ('enqueue them') – added to the back because they are newly reached and we have other nodes to do before them (like node C)
- Pop C from front of queue
  - No nodes are adj to C, so there's nothing more to be done.
  - C is removed (popped off) queue
- Algo continues since queue isn't empty
- Pop E from queue
  - Handle adj nodes
  - Mark these nodes with (cost of E) +1
  - Add them to back of queue
- Repeat until all nodes have been processed.

- How can we be certain paths found are the shortest?
  - Mark all nodes Nj that are adj to current node Ni with Ni's distance + 1
    - Only one edge gets added – if there was a shorter path, it would have less edges BUT then it also would have been marked in an earlier iteration of the algo.
  - This algo works inductively and so it's true for every step
    - This implies all paths are shortest paths.

Djikstra

- http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html
- http://math.mit.edu/~rothvoss/18.304.3PM/Presentations/1-Melissa.pdf
- http://www.ifp.illinois.edu/~angelia/ge330fall09_dijkstra_l18.pdf
- BFS doesn't work for weighted graphs – in the case of weighted graphs, shortest path is the one where the cost from one node to another is cheapest

Main ideas

- Keep queue of nodes as we reach them, start with start node, then nodes adj to start node, then nodes adj to those, etc – as done in BFS.
- As in BFS, proceed in order of nearness i.e from nearest to furthest nodes.
- Since edges have different costs, the nxt nearest node will depend on the costs/distances to those nodes., and NOT in order we reached them – so use a PQ (NOT a normal FIFO queue).
- Can only finalise the shortest-distance path to any node when it is the min/top of the PQ
    - Why? Because an even shorter path to it may otherwise exist that goes through one of those nodes in the queue that is nearer to the start node (will only discover this when we process that nearer node).
    - But for the node at top/min of PQ, there's no pt waiting to see if a shorter path exists via other nodes in the PQ, because they're all further away and thus more costly.
- Work backwards through the previous values of the nodes to obtain the final path
    - Each node will have a value for it's "previous" i.e. the node we used to get to our current node.
    - Start at final node and work backwards to start node.

PseudoCode

```
1:   function Dijkstra(Graph, source):
2:       for each vertex v in Graph:                // Initialization
3:           dist[v] := infinity                    // initial distance from source to vertex v is set to infinite
4:           previous[v] := undefined               // Previous node in optimal path from source
5:       dist[source] := 0                          // Distance from source to source
6:       Q := the set of all nodes in Graph         // all nodes in the graph are unoptimized - thus are in Q
7:       while Q is not empty:                      // main loop
8:           u := node in Q with smallest dist[ ]
9:           remove u from Q
10:          for each neighbor v of u:              // where v has not yet been removed from Q.
11:              alt := dist[u] + dist_between(u, v)
12:              if alt < dist[v]                   // Relax (u,v)
13:                  dist[v] := alt
14:                  previous[v] := u
15:      return previous[ ]
```

NOTE: this is NOT given in Sonia's notes

CODE

```
/**
 * Single-source weighted shortest-path algorithm.
 */
public void dijkstra( String startName )
{
    PriorityQueue<Path> pq = new PriorityQueue<Path>( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    clearAll( );
    pq.add( new Path( start, 0 ) ); start.dist = 0;

    int nodesSeen = 0;
    while( !pq.isEmpty( ) && nodesSeen < vertexMap.size( ) )
    {
        Path vrec = pq.remove( ); // pop the node with the shortest distance
        Vertex v = vrec.dest;
        if( v.scratch != 0 )  // already processed v
            continue;

        v.scratch = 1;
        nodesSeen++;

        for( Edge e : v.adj ) // look at each edge leaving that node
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( cvw < 0 )          // get the destination node that edge goes to
                throw new GraphException( "Graph has negative edges" );

            if( w.dist > v.dist + cvw ) // update & Q it if this route better
            {
                w.dist = v.dist + cvw;
                w.prev = v;
                pq.add( new Path( w, w.dist ) );
            }
        }
    }
}
```

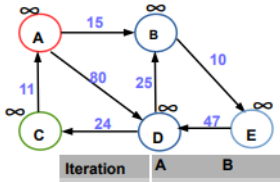
EXAMPLE AND EXPLANATION

1. Start at node A – mark with zero, since there's no cost getting to where we already are.
    a. 0 is now min of PQ..
    b. All other nodes marked with ∞
2. Nodes adj to A are B and D
    a. Algo checks if these give a better path to B and D than it had found for them before.
    b. B is updated with (Cost to reach A) + (edge weight of A to B) = 0+15; since 15 < ∞, B becomes 15.
    c. The same is done for node D – which is updated to 80.
    d. A is marked off as done
    e. PQ is now [B,D]
3. The cheapest vertex that hasn't been done is next selected
    a. thus B.
    b. Examine all vertices adj to B, which is E only.
    c. E is updated with (cost to reach B) + (edge weight of B to E) = 15+10 =25, since 25 < ∞
    d. Mark B off as done
    e. PQ is now [E, D]
4. The cheapest vertex that hasn't been done is next selected
    a. E is used
    b. Total cost to get to D is now (cost to reach E) + (edge weight of E to D) = 25 +47 = 72
    c. Since 72 < 80, update value of D to 72.
    d. Mark E off as done
    e. PQ is [D]
5. The cheapest vertex that hasn't been done is next selected
    a. Use D

b. Nodes adj to D are B and C, so algo checks if these give a better path to B and C than it had found for them before.
c. D → B edge has weight 25, and so getting to B from this edge is (Cost to reach D) + (edge weight from D to B) = 72 + 25 = 97
    i. Since 97> 15 (current cost of B), no changes made to vertex/node B.
d. Total cost to get to C is (Cost to reach D) + (edge weight of D to C) = 96, since 96 < ∞, C updated to 96.
e. Mark D off as done.
f. PQ is now [C]

6. The cheapest vertex that hasn't been done is next selected
    a. Next is C
    b. Since we're finding cheapest path to C, and C is now top of PQ, we stop.
7. The path is thus A → B → E → D → C

Example: Compute the shortest path from A to C



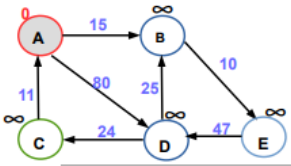
Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Iteration	A	B	C	D	E
0	∞	∞	∞	∞	∞

Weiss has shown us code for the algorithm. When applying the algorithm to an example, use a table with a column for each node, as in these slides – clearest for tests & exams.

Example: Compute the shortest path **from A** to C

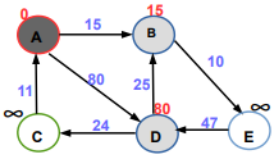


Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞

Now add nodes adjacent to A. i.e. (A → B) and (A → D)

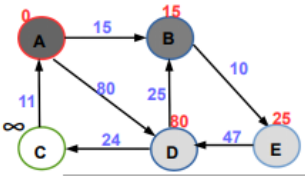


Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞

Now select the node closest to A (i.e. minimum distance), which is node B
Check other reachable vertices from "current vertex", B

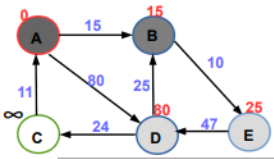


Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞
B		15_A	∞	80_A	25_B

Now select the minimum distance, that is node E
Check other reachable vertices from "current vertex", E

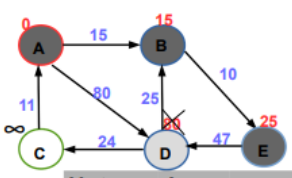


Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞
B		15_A	∞	80_A	25_B

Continue the process...E is new min-node, update cost for node D



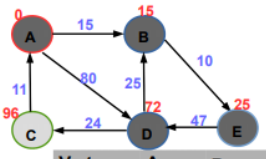
Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞
B		15_A	∞	80_A	25_B
E			∞	72_E	25_B

If the newly computed cost is < than stored cost, update (else don't)

Continue the process...D is new min-node, update cost for node C



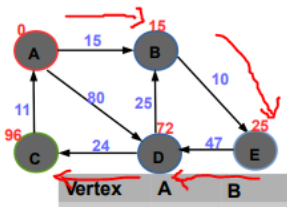
Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞
B		15_A	∞	80_A	25_B
E			∞	72_E	25_B
D			96_D	72_E	

If the newly computed cost is < than stored cost, update (else don't)

What is the cost of the minimum weighted path from A to C?
What is this path?



Adjacency List

A → B → D
B → E
C → A
D → B → C
E → D

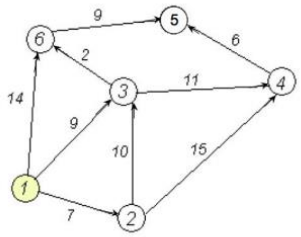
Vertex	A	B	C	D	E
A	0_A	∞	∞	∞	∞
A	0_A	15_A	∞	80_A	∞
B		15_A	∞	80_A	25_B
E			∞	72_E	25_B
D			96_D	72_E	
C			96_D		

- Another example:

NOTE: this is NOT given in Sonia's notes

Lecture 18

Dijkstra's Algorithm: Example

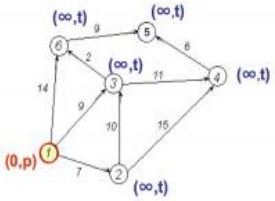


We want to find the shortest path from node 1 to all other nodes using Dijkstra's algorithm.

Lecture 18

Initialization - Step 1

- Node 1 is designated as the current node
- The state of node 1 is $(0, p)$
- Every other node has state (∞, t)

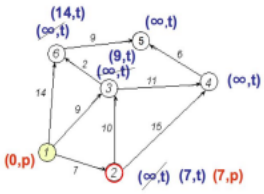


Lecture 18

Step 2

- Nodes 2, 3,and 6 can be reached from the current node 1
- Update distance values for these nodes

$$d_2 = \min\{\infty, 0+7\} = 7$$
$$d_3 = \min\{\infty, 0+9\} = 9$$
$$d_6 = \min\{\infty, 0+14\} = 14$$



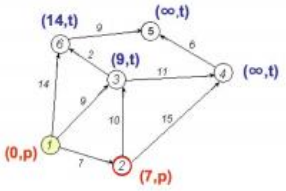
- Now, among the nodes 2, 3, and 6, node 2 has the smallest distance value
- The status label of node 2 changes to permanent, so its state is $(7, p)$, while the status of 3 and 6 remains temporary
- Node 2 becomes the current node

Operations Research Methods 13

Lecture 18

Step 3

Graph at the end of Step 2

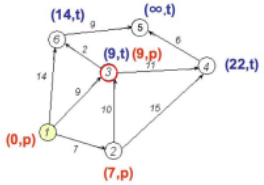


We are not done, not all nodes have been reached from node 1, so we perform another iteration (back to Step 2)

Operations Research Methods 14

Another Implementation of Step 2

- Nodes 3 and 4 can be reached from the current node 2
- Update distance values for these nodes



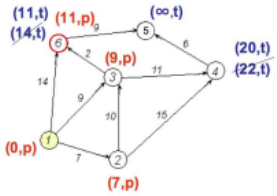
$d_3 = \min\{9, 7+10\} = 9$
$d_6 = \min\{\infty, 7+15\} = 22$

- Now, between the nodes 3 and 4 node 3 has the smallest distance value
- The status label of node 3 changes to permanent, while the status of 6 remains temporary
- Node 3 becomes the current node

We are not done (Step 3 fails), so we perform another Step 2

Another Step 2

- Nodes 6 and 4 can be reached from the current node 3
- Update distance values for them



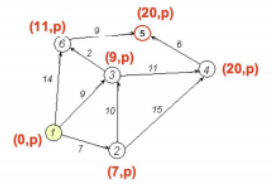
$d_4 = \min\{22, 9+11\} = 20$
$d_6 = \min\{14, 9+2\} = 11$

- Now, between the nodes 6 and 4 node 6 has the smallest distance value
- The status label of node 6 changes to permanent, while the status of 4 remains temporary
- Node 6 becomes the current node

We are not done (Step 3 fails), so we perform another Step 2

Another Step 2

- Node 5 can be reached from the current node 6
- Update distance value for node 5



$d_5 = \min\{\infty, 11+9\} = 20$

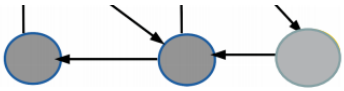
- Now, node 5 is the only candidate, so its status changes to permanent
- Node 5 becomes the current node

From node 5 we cannot reach any other node. Hence, node 4 gets permanently labeled and we are done.

Comparison of BFS and Djikstra

- Both algos:
  - Start with setting all vertices *except start* to infinity (-9999 in slides) i.e. all vertices unreachable.
  - Put start vertex in a queue
  - Loop.
- Differences:
  - Take nxt item off queue
    - BFS takes **front**
    - Djikstra takes **min**
  - For each adjacent node, in order for it's cost to change:
    - BFS is because node was not reached before
    - Djikstra is because its cheaper than its cost found so far

- Update its cost and add it to queue.



```
set each vertex cost to 999999999
v.cost = 0
initialize queue with v
while queue not empty DO
    a = cheapest in queue
    for each each edge a → w
        c =  a.cost + (weight of a→w)
        if c < w.cost
                w.cost = c
                add w to queue
    remove a from queue
```

```
bfs(v): // shortest paths from v
set each vertex cost to -9999
v.cost = 0
initialize queue with v
while queue not empty DO
    a = front of queue
    for each edge a → w
        if w.cost = -9999
                c = a.cost + 1
                w.cost = c
                add w to queue
    remove a from queue
```

Bellman-Ford

- https://www.youtube.com/watch?v=obWXjtg0L64
- https://courses.csail.mit.edu/6.006/fall11/lectures/lecture17.pdf
- http://www.cs.tulane.edu/~carola/teaching/cs5633/spring05/slides/Lecture-18.pdf
- Dijkstra's algo fails when weights are negative
    - Algo uses PQ so that it doesn't have to compute cost of all possible paths in the graph.
    - By going in priority order, Djikstra computes shorter paths *before* longer (costlier) paths
        - Means that first time each node is reached => that's definitely the shortest possible path by which it can be reached.
- If you have a cycle where the **sum of the weights** on the edges of that cycle are negative, then an algo to find the shortest path will go into an infinite loop.
    - Why? Each time it goes around the cycle, cost path decreases. Trying to minimise cost will thus make it go forever.
    - A cycle with a negative weight is NOT a problem for BF.
- Any algo to find shortest paths in negative weighted graph must be able to detect negative cycles to avoid falling into an infinite loop.

Detect Negative Cycles

- An edge involves a src and destination node i.e. 2 nodes.
- If no cycle in graph:
    - Path with 2 edges has 3 nodes, 3 edges has 4 nodes etc
- If graph with 2 nodes has path 2 edges long, it has to be a cycle.
- Have N nodes in a graph and shortest path has N edges, then it must visit one of the nodes more than once.
    - Basically, the path *must* have a cycle in it somewhere.
    - As that cycle is part of the shortest path to that node, it must be a negative cost cycle, otherwise it would be cheaper to *not* use that cycle.
- Any algo to find shortest paths in negative weighted graph needs to check none of the shortest paths is finding more than N edges
    - As soon as path with N edges is found, algo must stop and output that there's a negative cost cycle.

Algorithm Overview

- Start node initialised to zero – all other nodes have distance value of infinity.
- Take start node and put nodes reached from start node in queue.
    - Keep popping nodes off the queue and adding their adj nodes (Nj) onto q whenever this new path to Nj is cheaper than paths seen before.
- NB that need to able to detect negative cycles
    - BF keeps track of negative cycles in graph of N vertices by tracking how many times each node is put on q – if val ever reaches N, then a path of N edges (i.e. path containing cycle) has been found so algo will stop.

- Work backwards through the previous values of the nodes to obtain the final path
    - Each node will have a value for its "previous" i.e. the node we used to get to our current node.
    - Start at final node and work backwards to start node.

PSEUDOCODE

Bellman-Ford in pseudocode

```
initialise( );
enqueue(S);
while QnotEmpty( ) do  popQandProcess( );


popQandProcess( ):
{
F = popQ( );
checkNegCycle(F); // if F queued more than N times it's a -ve cycle
for every edge F →T do
        checkCost(T,F);    // update T's entry to come via F if better
}
```

CODE

From Weiss book

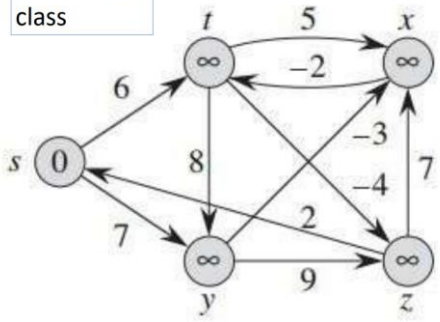
```
/**
 * Single-source negative-weighted shortest-path algorithm.
 */
public void negative( String startName )
{
    clearAll( );

    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    Queue<Vertex> q = new LinkedList<Vertex>( );
    q.add( start ); start.dist = 0; start.scratch++; //Start is only node to process at first

    while( !q.isEmpty( ) )  //while there are more nodes to process
    {
        Vertex v = q.removeFirst( );
        if( v.scratch++ > 2 * vertexMap.size( ) )
            throw new GraphException( "Negative cycle detected" );

        for( Edge e : v.adj )  //for each edge leaving the node we are currently processing
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( w.dist > v.dist + cvw ) // if edge v->w gives a new cheapest path to w
            {
                w.dist = v.dist + cvw; // update the 2 items storing cheapest path detail for w
                w.prev = v;
                  // Enqueue only if not already on the queue
                if( w.scratch++ % 2 == 0 )
                    q.add( w );  // ensure w queued for processing, as we have new data for it
                else
                    w.scratch--;  // undo the enqueue increment
            }
        }
    }
}
```

- Queue (q) used go keep track of all nodes as they're reached.
- q starts with start vertex at beginning.
- Algo continues until q is empty.
- **Pop node** V from q
- **For every edge** adj to V i.e. every edge from V to W:
    - If (cost of reaching V) + (edge weight) is new cheapest path to W, update W and ensure W is on q since it has new cheapest cost.
- Scratch is a 0 for all nodes
    - Gets increased by 1 each time node is put on q
    - Also increased by 1 each time it's taken off the q
    - If scratch odd => node is already on the q.
    - If scratch == 2* numNodes, there's a negative cycle.
        - Node's cheapest cost has decreased numNodes times.

EXAMPLES

(Brief)

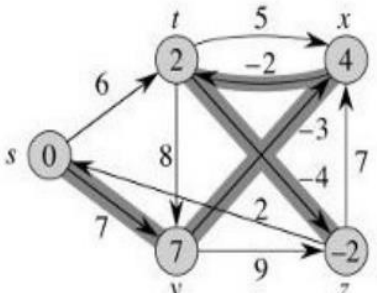
Exercise in class



SLIDE 12 : Exercise – we use tables to keep track again

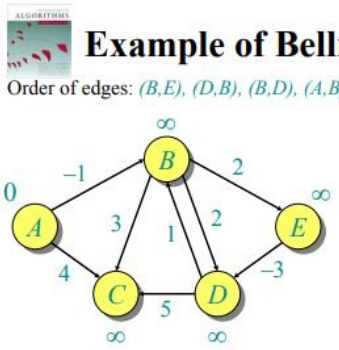
Pop (& go via)	Q after this row	s	t	x	y	z
	s	0	Infinity	Infinity	Infinity	infinity
s	t,y	0	6	Infinity	7	infinity
t	y,x,z	0	6	6+5=11	7 (as 7 < 6+8)	6+(-4)=2
y	x,z	0	6	7+(-3)=4	7	2 (as 2 < 7+9)
x	z,t	0	4+(-2)=2	4	7	2
z	t	0	2	4 (4 < 2+7)	7	2
t	z	0	2	4 (4 < 2+5)	7 (as 7 < 2+8)	2+(-4)= -2
z		0	2	4 (4 < (-2)+7)	7	-2

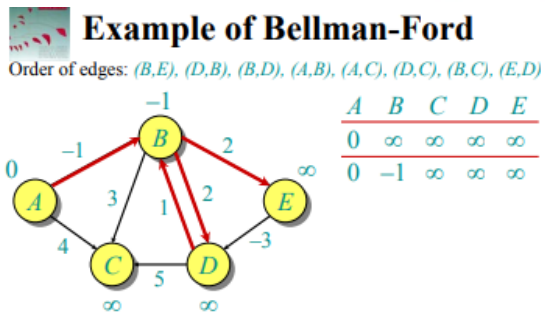
The idea is just to pop the front of the queue, add its cost (in bold) to the cost of each edge leaving that node and see if that total is less than the current cost of the destination node – and if cheaper, add destination to Q. Stop when Q empty.

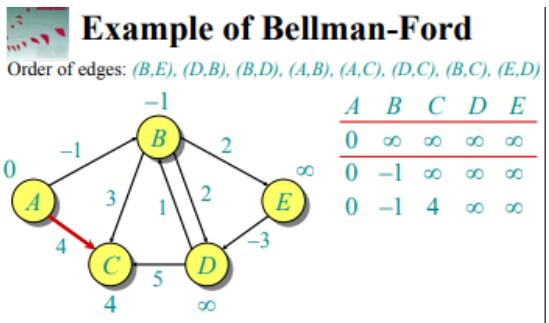


(In-depth)

NOTE: this is NOT given in Sonia's notes

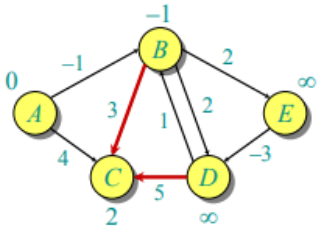






Example of Bellman-Ford

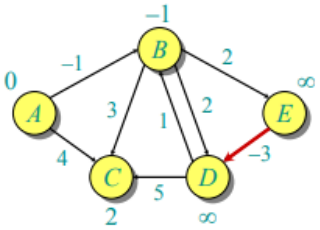
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

Example of Bellman-Ford

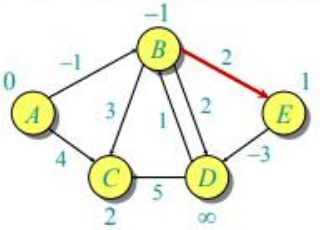
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞

Example of Bellman-Ford

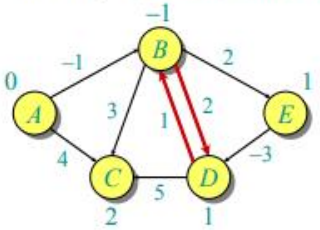
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1

Example of Bellman-Ford

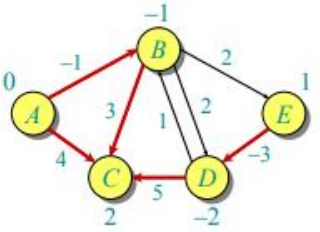
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1

Example of Bellman-Ford

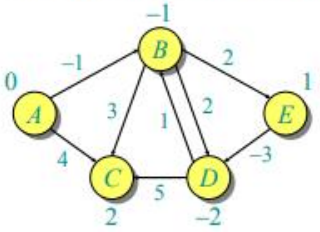
Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

Example of Bellman-Ford

Order of edges: *(B,E), (D,B), (B,D), (A,B), (A,C), (D,C), (B,C), (E,D)*



Note: Values decrease monotonically.

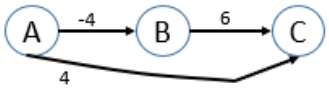
A	B	C	D	E
0	∞	∞	∞	∞
0	−1	∞	∞	∞
0	−1	4	∞	∞
0	−1	2	∞	∞
0	−1	2	∞	1
0	−1	2	1	1
0	−1	2	−2	1

… and 2 more iterations

Changing weights

- Why not add a constant N value to all weights in graph to make them all positive? Then can use Djikstra, which is more efficient to compute shortest path?
    - Cost will then be affected by number of edges in path.
- An example for this:

Here is an example of why it does NOT work to add the same large number to all edges to make them all positive (in order to use Dijkstra's algorithm instead of Bellman-Ford):



If we add a value like 4 to every edge, then A->B->C is 0 + 10 = 10 and A->C is 8, so A->C becomes the shortest path, when actually A->B->C is shortest. This is because the path with 2 edges had 4+4=8 added, and the path with 1 edge only had 4 added to it. Similarly for any other example.

Summary

- If there's negative weights, Dijkstra breaks:
    - Can process node V before U
    - Then can later find a negative path from V to U, such that this new path to V costs less than its current best one.
    - Also implies that other nodes going via V in their shortest paths are wrong too.
        - What this means is that all the nodes we reached by going through V will have smaller costs too (V will change and so consequently all nodes reached via V will change too).
- A negative cost cycle is bad:
    - Shortest paths are undefined (path will get smaller and smaller with no end).
    - Results in infinite loop.
    - Can check for this by checking if node has been removed from queue (dequeued) |V| times OR
      By doing loop |V| times and seeing if an improvement (decrease in cost of path) occurs – path of more than |V| edges has a cycle; cycle may be a -ve cycle.
- BF uses q like BFS and cost-update like Dijkstra:
    - But can be slower, as it has O(|E|*|V|)
    - Goes through all edges |V|-1 times

Directed Acyclic Graph

- DAGs can have weighted (positive and negative) edges

Topological sorting

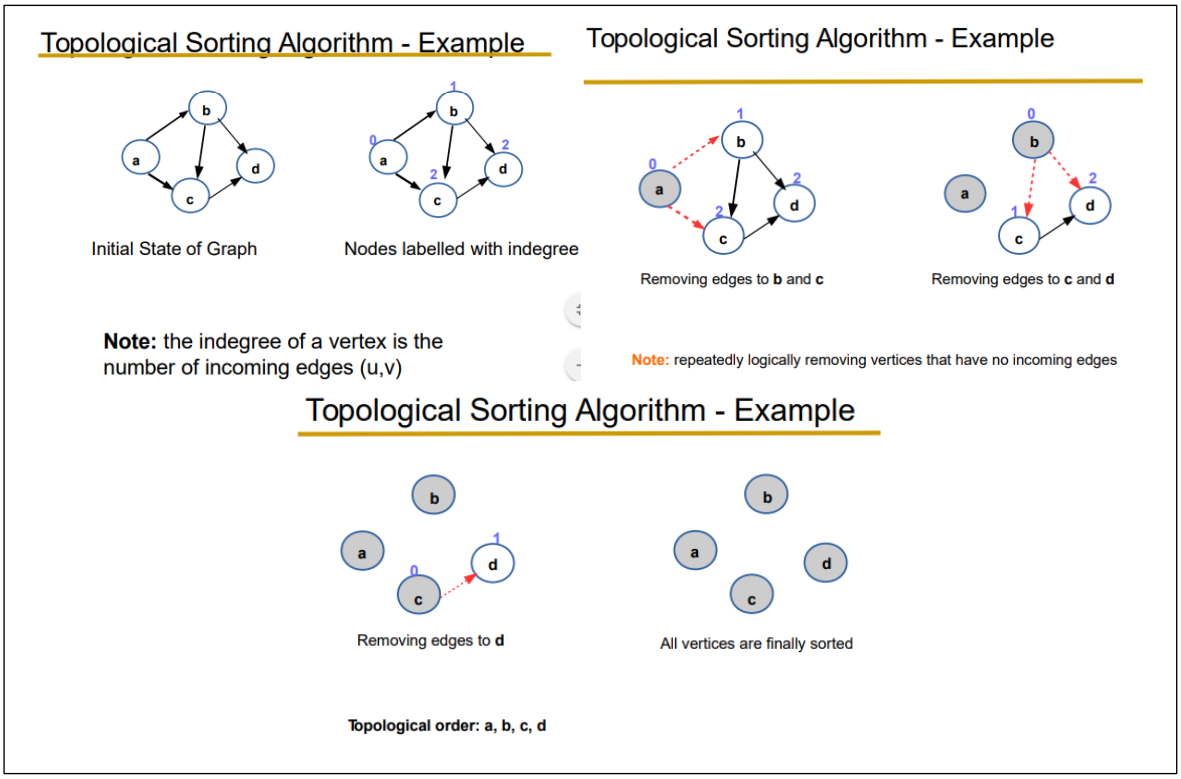
- Topological sort **orders vertices** in a *directed graph* such that if there's a path from u to v, then v appears after u in ordering.
    - i.e. same as when we can only start a task after all other tasks that had to finish before it had already been done.
- General idea: ordering where no node N is visited before another node Ni if there's a path from Ni to N.
    - i.t.o ordering tasks, means no task N is done before another task N1 if there's a path from N1 to N – if there IS such a path, N1 should have finished before N started.
- What to do when there's a cycle?
    - Topological sorting **only defined** for acyclic graphs
    - Any graph with a cycle in it ***can't*** have topological ordering.
        - Algo for topological sorting must be able to detect cycles and stop (else it'll run forever).
- Graphs can have more than one topological order.
- Implemented using a queue.
- Indegree(of a vertex): the # of incoming edges (u,v) ; v and u are vertices.

Algorithm

1. Find all vertices in a graph, compute all indegrees
2. Find vertices with indegree of 0
    a. Add to queue
3. If none exists (no vertices with indegree == 0), then have graph with a cycle
4. Else remove the vertices with indegree 0 as well as their edges
    a. Code processes their adjacency list and decreases the indegree value of the destination nodes by 1.
5. Repeat from (2) onwards until no more vertices to process.

- Note: don't really remove edges, rather just keep indegree count with node
- Algo also detects if graph has cycle - if there's a cycle:
    - There'll be unprocessed nodes; none of them will have an indegree of zero
    - If there are N such nodes, there there's at least N incoming edges – BUT the *only* way there can be N/more edges when there are only N nodes is if a cycle exists.
        - E.g. if have a graph of 3 vertices and each have an indegree of 2 then there definitely a cycle (and thus the graph is cyclic)

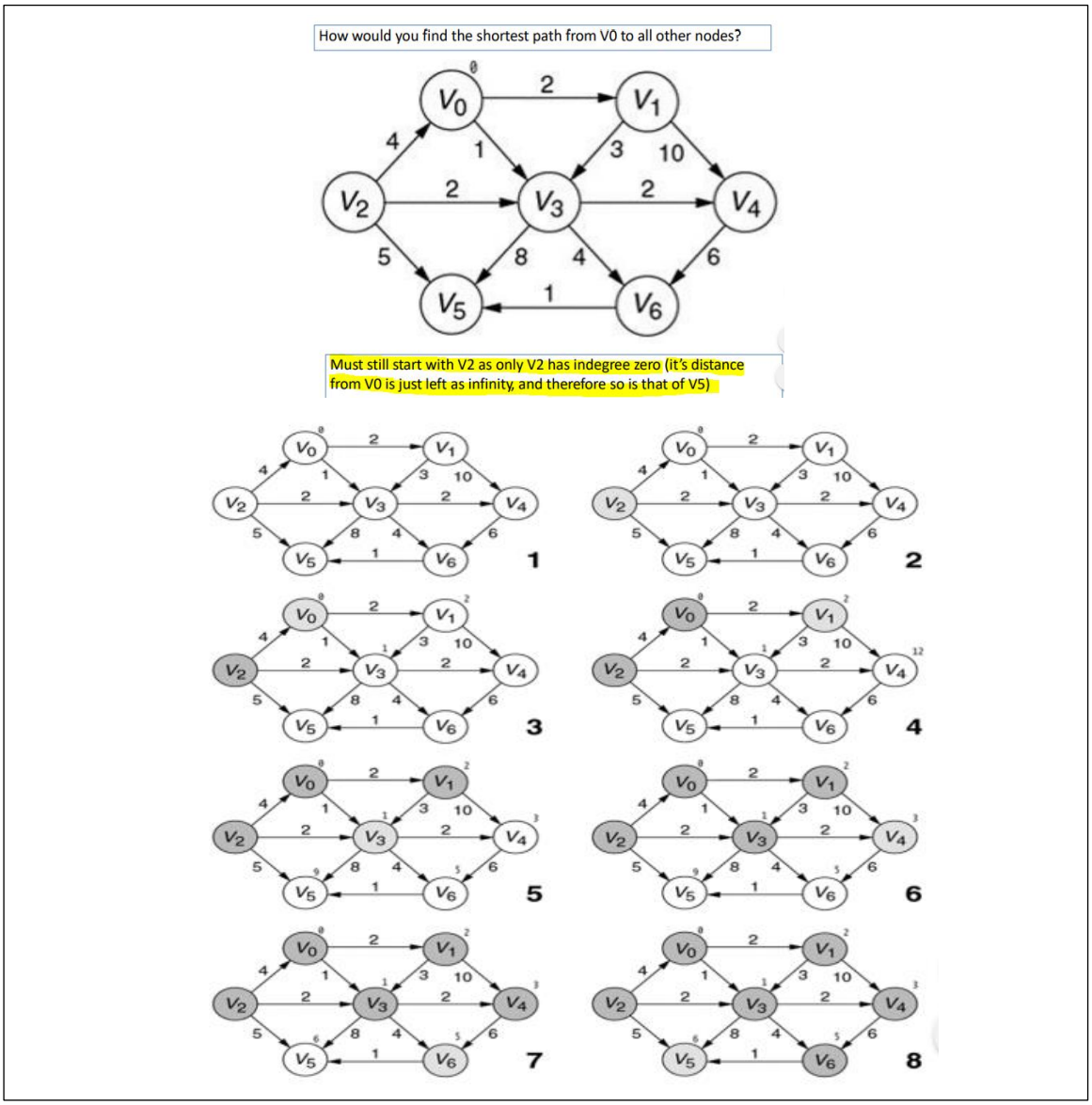
EXAMPLE



Shortest Paths

- Topological ordering can be used to find shortest path.
- Start node initialised to zero – all other nodes have distance value of infinity.
- Nodes processed in topological order.
    - Like with Dijkstra, each time process a node (V), cost of each adjacent node Nj is compared to (cost to V) + (edge weight V -> Nj).
    - If this new edge gives lower cost, new cost is stored and the previous for that node set to V.
- NB that there's no cycles since DAG:
  ==> there will be topological ordering, so **every node** will be processed and since we examine all edges leaving a node each time, **every edge** will be considered too.
- After processing node, pop it off q
    - Node only gets processed when its indegree is zero
      ==> no path to it from nodes not yet processed (no more paths definitely implies *no cheaper paths*).
- Remember that algo *only starts* with vertex that has indegree zero
    - This means that even if we have a graph e.g. with nodes A to D (not in that order) and we're trying to find the cheapest path from A -> D, if A starts with indegree =/= 0, we CANNOT start there.
    - We would thus have to find the node with indegree 0 and start there
        - If e.g. this node was C, then A would still start with cost 0 and C with cost infinity, and C's cost would *stay at infinity*. This implies we wouldn't be able to get from C to A in the DAG.

EXAMPLE



- Even though we want shortest paths from V0, the algorithm can only start with nodes of indegree 0, so in this graph, cannot start with V0 because it has indegree = 1.
    - BUT the cost of V0 starts at zero and the cost of all other nodes at infinity.
- V2 is the only node with indegree 0 it will be added to q, processed first, then popped off the q.
    - Its cost will remain infinity - we ***cannot*** get to V2 from V0 in this DAG.

CODE

```
/**
 * Single-source negative-weighted acyclic-graph shortest-path algorithm.
 */
public void acyclic( String startName )
{
    Vertex start = vertexMap.get( startName );
    if( start == null )
        throw new NoSuchElementException( "Start vertex not found" );

    clearAll( );
    Queue<Vertex> q = new LinkedList<Vertex>( );
    start.dist = 0;

      // Compute the indegrees
    Collection<Vertex> vertexSet = vertexMap.values( );
    for( Vertex v : vertexSet )
        for( Edge e : v.adj )
            e.dest.scratch++;

      // Enqueue vertices of indegree zero
    for( Vertex v : vertexSet )
        if( v.scratch == 0 )
            q.add( v );

    int iterations;
    for( iterations = 0; !q.isEmpty( ); iterations++ )
    {
        Vertex v = q.remove( );

        for( Edge e : v.adj )
        {
            Vertex w = e.dest;
            double cvw = e.cost;

            if( --w.scratch == 0 )
                q.add( w );

            if( v.dist == INFINITY )
                continue;

            if( w.dist > v.dist + cvw )
            {
                w.dist = v.dist + cvw;
                w.prev = v;
            }
        }
    }

    if( iterations != vertexMap.size( ) )
        throw new GraphException( "Graph has a cycle!" );
}
```

Weiss code (from book)

- First compute indegree of all nodes – store in “scratch”
- Then, put all nodes with indegree 0 in queue q
- Loop until q is empty, process each node as follows:
    - Pop node off q and store in variable v;
    - For each node w adj to v:
        - Decrease indegree of w by 1
        - If indegree of w is now 0, put w in q
        - If (distance to v + cost of v-> w edge) > distance to w, store new shorter distance and make v “prev” of w.
    - Last if loop can be skipped if dist to v is infinity (waste of time to compare if infinity smaller than w.dist)

Notes

- Can’t use BFS for topological sort – **edges weighted**
- Can’t always use Dijkstra algo for shortest paths – **less efficient** (than topological sort) and can’t work with **negative weights**
    - Topological sort is more efficient than Dijkstra because each node is enqueued only once ==> each edge only processed once
      ==> linear time algo
    - Dijkstra takes log N opns – because it uses a PQ
        - HOWEVER, Dijkstra can be used in more cases, since it can handle cyclic graphs.
- Topological sort may not visit nodes in same order as Dijkstra.

- But it CAN work on graphs with negative weights (unlike Dijkstra) – because nodes aren't processed in priority order i.e. algo **doesn't** assume that paths found later are longer than earlier ones (Dijkstra's does make this assumption).

Time complexity

- |E| is the number of edges in the graph and |V| the number of vertexes.
- BFS and Topological sort both process each edge once, has O(|E|) complexity.
- Dijkstra processes each edge once BUT takes log|V| time to pop each node off q because it's a PQ.
  - Min heap used for implementation (most efficient) - deletion requires log|V| opns to reorganise heap after the min removed.
- Bellman-Ford in WC has to loop through all edges |V| times , because negative weights can keep causing decreased costs.
  - In WC, algorithm is only able to terminate after |V| loops, with the last loop being done to ensure there are no negative cycles.

Worst-Case Running Times

S/N	Graph Problem Type	Running Time	Comments
1	Unweighted	O(|E|)	Breadth-first search
2	Weighted, no edge negative edges	O(|E| log |V|)	Dijkstra's algorithm
3	Weighted, negative edges	O(|E| . |V|)	Bellman-Ford algorithm
4	Weighted, acyclic	O(|E|)	Topological sort

Integrity Constraints

- Integrity in this context means the data is accurate and meaningful.
- An example of dtb integrity constraint is an FD like A→ B
    - Specifies that for A and B values to be meaningful, the same A value must always be associated with the same B value (else data is inaccurate/meaningless).

Domain constraints

- If domain types used instead of built-in types like VARCHAR or SMALLINT, then meaningful values for cols of that type can be more accurately defined and this will then be enforced by DBMS.
- Even queries can benefit from this – they can have less to check through when looking through rows.
- An example:
    - mark smallint
      constraint valid_percentage (optional name – useful to have name because in an error msg, the name indicates the problem i.e error msg will include constraint name in display.
      check ((mark >= 0) and (mark <=100))
    - When in this used by DBMS?
        - Checking updates and inserts – MUST obey constraints
        - Checking queries – e.g. xxx AND mark > 100 is empty set.
    - Queries can benefit – e.g. a SELECT that includes a condition like mark >100, DBMS won't bother going through the table for such rows – it would be a waste of time with the constraint in place already.

CREATING TABLES

- Should always specify NOT NULL for pk attrbs.
- For other attrbs, check with client before specifying NOT NULL.
    - Only specify NOT NULL if the client doesn't want the rest of the info (i.e. the other attrbs that have data in them for a row) stored in the dtb at all *unless* that value is included.
        - i.e. If have atrrbs Book, ISBN Number (NOT NULL), Author: if we have values for Book and author, but not ISBN, then we *cannot* enter that book into the dtb.
- Default values can be used too especially when the value will be the same for every instance of data entry
    - E.g. at a fire station, every fireman has completed fireman training, and so we can have "completed" as the default value for Training Status attrb.
- Can have artificial keys for everything and rather have DBMS use its own unique identifiers.
    - Each time row inserted, DBMS will set that field to nxt available value
    - Can do this with AUTO_INCREMENT.

EXAMPLES

MySQL / SQL Server / Oracle / MS Access:

```
CREATE TABLE Persons (
    ID          int NOT NULL,
    LastName    varchar(255) NOT NULL,
    FirstName   varchar(255),
    Age         int,
    City        varchar(255) DEFAULT 'CT',
    CONSTRAINT Chk_Person
        CHECK (Age>=18
               AND City IN ('DBN', 'CT', 'JBG')
);
```

```
CREATE TABLE Persons (
    ID int NOT NULL AUTO_INCREMENT,
    LastName varchar(255) NOT NULL,
    FirstName varchar(255),
    Age int,
    PRIMARY KEY (ID)
);
```

MySQL

Using system keys for tables (MySQL notation, differs slights for different vendor's DBMS). To insert a new record into the "Persons" table, we do NOT specify a value for the "ID" column (a unique value, the next integer, will be added automatically)

Referential integrity

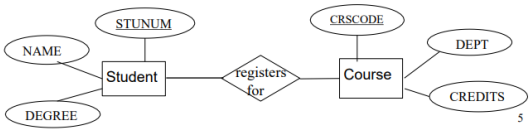
- Let R1 be a relation with PK K1
- Let A be an attrb of another relation R2 that is a FK referencing K1
- Referential integrity ensures that every value of A in R2 exists in column k1 of relation R1.
- Ensures fk values are valid – i.e references an *actual entity* in the table for those entities.
- Not only for insert opns that DBMS needs to enforce referential integrity.
    - Any delete/update (basically anything that *changes a row*) in one of the tables will also reflect in the other table and BOTH tables need to be left in a **valid state**

EXAMPLES

Referential Integrity

- Student (Stunum, Name, Degree)
- Course (Crscode, Dept, Credits)
- Registers (Stunum, Crscode)

- INSERT INTO Registers
  VALUES ("PRCMAX001", "STA5018F") ??????



Slide 5: while the Stunum PRCMAX001 looks like a valid UCT student number, if there is no such Stunum in the Student table, then the DBMS should not allow this insert, because it is meaningless to say that a non-existent student is taking STA5018F.

Referential integrity ensures every Pno in Orders appears as a Pno in Parts.
Put another way: referential integrity ensures foreign key values are valid.

Orders

Order	Pno	Qty
1231	101	12
1232	102	12
1233	204	24
1234	204	12
1235	402	24

Parts

Pno	Name	Cost	Size
101	axle	4	1
102	cog	4.50	1
203	nut	9	5
204	screw	12.50	7
402	gizmo	22	12

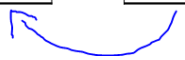
Referential integrity ensures every Pno in Orders appears as a Pno in Parts. So if part 402 is deleted, or the Pno value 402 is changed in Parts, something must be done: e.g. the delete/update must be disallowed, or order 1235 must be updated/deleted.

Orders

Order	Pno	Qty
1231	101	12
1232	102	12
1233	204	24
1234	204	12
1235	402	24

Parts

Pno	Name	Cost	Size
101	axle	4	1
102	cog	4.50	1
203	nut	9	5
204	screw	12.50	7
402	gizmo	22	12



FORMAL DEFINITION

- Let P and O be relations with PKS P# and O# respectively.
- PF of O is a fk referencing P# in P, if for every tuple T in O, there must be a tuple Q in P such that:
  - Q[P#] = T[PF]

Database Modification

- DBMS *must* check referential integrity constraints on:
  - Insert
  - Delete
    - What if other tuples reference deleted tuple?
      - Delete rejected – delete doesn't occur; display an error.
      - Delete cascades – apply same operation to row referencing deleted tuple
      - References nullified – replace value with NULL (unknown).
  - Update
    - What if FK attrb changed?
    - What if other tuples reference a tuple whose key is changing?

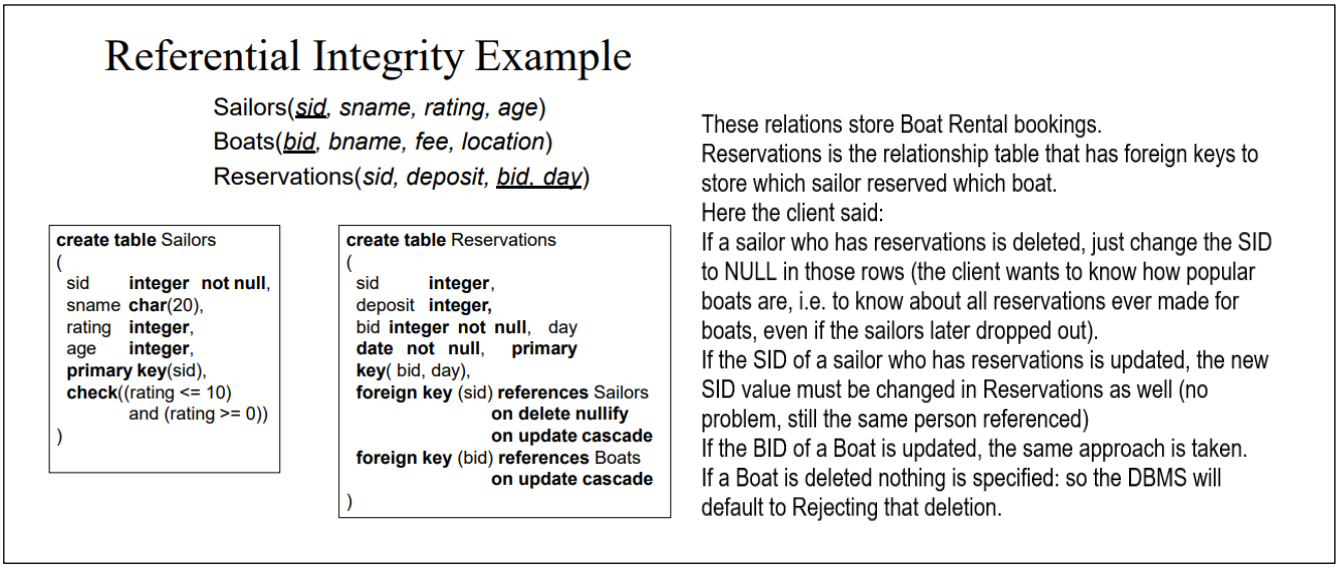
- Update rejected – update doesn't occur; display an error.
- Update cascades - apply same operation to row referencing updated tuple
- References nullified – replace value with NULL (unknown).

FORMAL DEFINITIONS

- DBMS *must* check referential integrity constraints on:
  - Insert: if tuple t2 in inserted into r2, the sys must ensure that there's a tuple t1 in r1 such that t1[K] = t2 [α].
  - Delete: if tuple is deleted from r1, the sys must compute the set of tuples in r2 that reference t1
    - If this set is *not empty*, either the delete command is rejected as an error, OR, the tuples that reference R1 must themselves be deleted - cascading deletions; or nullified.
    - Delete requires DBMS to find all rows in dtb that reference one being deleted and to see what the **schema specifies** must be done with each of these rows.
    - An example, with the Orders and Parts tables:
      - Designer asks client what they want the dtb to store when a Part that is still in the Orders table is removed from the database.
        - Reject => client says don't allow Parts still on Order to be deleted;
        - Cascade => client says just delete that Order row (risky in this example, just now these orders are delivered, and they don't want that part!)
        - Nullify => client says keep that Order row, but use Pno value NULL so I know it's an old order for a Part I no longer want.
  - Update:
    - Requires DBMS to find all rows in dtb that reference the one being updated and to see what the **schema specifies** must be done about each of those rows.
    - There are two cases
      - If update mods values for FK α, sys must ensure new value is valid FK.
      - If update mods values for a PK (K), then sys must first look for FKs containing old value.
        - If this set isn't empty, then update may be rejected as an error, or update may cascade to tuples in set, or tuples in set may be deleted.
    - An example, with the Orders and Parts tables:
      - Designer asks client what they want the database to store when a Part in the Orders table has its Pno value changed.
        - Reject => client says don't allow a Pno in the Order to be changed ("this is a serious act I want to check up on this").
        - Cascade => client says just update that Pno in all Order rows where it appeared (no problem, still point to same Parts row)

- Nullify => client says keep that Order row, but use Pno value NULL so I know it's an old order for a Part where the Pno was changed after the Order was placed (risky here!)

EXAMPLE



SQL Constraints

- SQL constraints specify rules for data in a table and limit type of data that can go into a table.
- If any action violates the constraint, action is aborted.
  - Ensures accuracy and reliability of data.
- Col lvl constraints apply to a col; tbl lvl constraints apply to a whole relation.
- Some SQL constraints:
  - NOT NULL – ensures column can't have null value.
  - UNIQUE – ensures that all values in col are different
  - PRIMARY KEY – a combo of a NOT NULL and UNIQUE; uniquely identifies each row in a table.
  - FOREIGN KEY - uniquely identifies a row/record in another table.
  - CHECK – ensures all values in a col satisfies a specific condition.
  - DEFAULT – sets a value for a col to use when no value is given.
  - INDEX – used to create and retrieve data from the dtb very quickly.

TRIGGERS

- Trigger is automatically executed as a side effect of a dtb mod
  - It's basically a piece of code that the dtb designer specified that will execute if specific conditions become true in the dtb.
- To design a trigger:
  - Specify conditions under which trigger must be executed
  - Specify actions to be taken when trigger executes.
- NB that triggers are *extra work* for the DBMS , so should not be made indiscriminately/over-used.

Example

- Suppose when a final mark between 45 and 49 is entered, the university:
  - Makes the result symbol “FS”
  - Creates a new registration for that student
  - The course name in the new registration has “SUP” added to it
  - The result symbol for the new registration is made “AB” for the moment
- In PL/SQL (not covered in course – prog lang used in triggers and stored procedures):
  - The part in red is the condition which triggers the actions.
  - The actions are in black and are written using PL/SQL code.
  - The trigger states if the actions must be run before or after the SQL command; and the command can be any operator, not just update.

Trigger Example (uses PL/SQL;
PL/SQL is not covered in this course)

```
create trigger obtain_supp after update on result
referencing new row as nrow
for each row
when nrow.final < 50  and nrow.final > 44
begin atomic
   insert into registration values
        ( nrow.regno,  concat(nrow.course, “SUP”),  “AB”);
   update result set symbol = “FS”
        where result.regno = nrow.regno
        and result.course = nrow.course
end
```

SQL INJECTION ATTACKS

- https://www.youtube.com/watch?v=ciNHn38EyRc
- Queries can include user-supplied text
  - Queries that include this need to prevent users from supplying text that lets them see and/or change data they don’t have access to
- SQL injection attack: when a user tries to get access to and/or change data that they don’t have permission to access by means of submitting queries with malicious SQL statements in them.

Examples

SQL injection attacks

```
uId = getRequestString("UserId");
q = "SELECT * FROM Uscrs WHERE UscrId = " + uId;
```

SQL injection can result in e.g. this:

```
SELECT * FROM Users WHERE UserId = 105 OR 1=1;
```

In the first example, q is meant to give those with the required permission to select data from Users, the ability to see that table. By inputting any random integer followed by OR TRUE, anyone can see that table.

If that doesn't work, it could be because the query was looking for part of a string (i.e. user's random value was inside quotation marks). In that case, by putting in the close-quote character first and then OR followed by any operation on strings that returns true when followed by a close-quote (which you do not supply), the table can be seen by anyone.

```
uName = getRequestString("username");
uPass = getRequestString("userpassword");
sql = " SELECT * FROM Users WHERE Name = ' " +
uName + " ' AND Pass = ' " + uPass + " ' " ;
```

SQL injection can result in e.g. this:

```
SELECT * FROM Users WHERE Name =
' ' or 'x' = 'x' AND Pass = ' ' or 'x' = 'x' ;
```

Another SQL injection attack

```
uId= getRequestString("UserId");
q = "SELECT * FROM Users WHERE UserId = " + uId;
```

SQL injection can result in e.g. this:

```
SELECT * FROM Users WHERE UserId =
105; DROP TABLE Suppliers;
```

In this example any User who is able to select from this table, even just their own row say, can do any malicious damage to the database
All they have to do is supply, as input, not just their uID but their uID followed by a semicolon and any malicious database alteration statement.

Prevention

- DBMS provide functions to prevent these attacks – these should be used when working with user supplied input.
- MYSQL function (a real example):

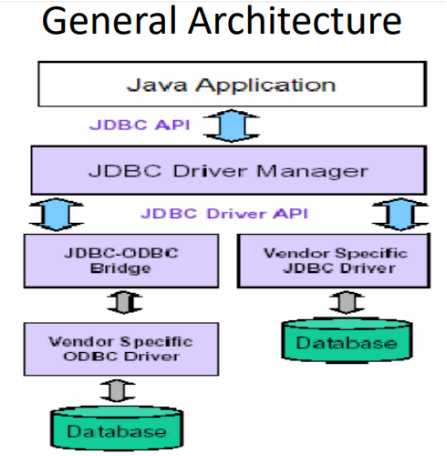
Use mysql_real_escape_string for MySQL to remove injection attacks from input strings

```
$name_in = getRequestString("UserId");
$name_in = mysql_real_escape_string($name_in);
$query_safe = "SELECT * FROM customers
        WHERE username = $name_in";
```

Embedded SQL and JDBC

- http://java2novice.com/jdbc/
- Can use diff prog langs to acces dtb.
- Why embedded SQL?
    - Embedded SQL is a method of combining the computing power of a programming language and the database manipulation capabilities of SQL.
    - Used for complex queries and apps
    - Basically, the capabilities of SQL and a high-level programming lang like Java.

General Architecture



- ODBC driver: used for dtb access from C and C++.
- DBMS vendors provide ODBC and JDBC drivers.
- JDBC API is used in all other statements, irrespective of which driver is interfacing with the dtb.
- Prerequisites:
    - A DBMS
    - Java and JDBC (JDBC API)
    - Drivers

JDBC

- The JDBC API consists of a set of interfaces and classes written in the Java programming language.
- A set of Java interfaces – Driver, Connection, Statement, ResultSet, etc.
- Database vendors implement these interfaces
- If switch from one DBMS to another, just load a different driver and don't need change the rest of the code.

Terminology

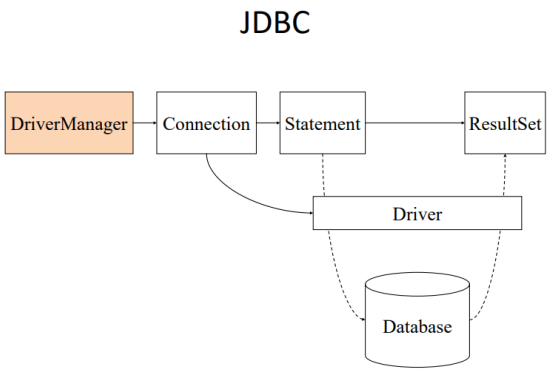
- JDBC: Java API for connecting to databases.
    - Defines behaviour.
- JDBC Driver: an implementation of the JDBC driver interface provided by the database vendor
- Database URL: string to uniquely determine a dtb server and dtb.

Programming with JDBC

1. Import **JDBC classes**
    a. Import java.sql.*
2. Load a **driver**
3. Make a **connection**
4. Interact with dtb:
    a. Create JDBC statements
    b. Execute JDBC statements
    c. Process JDBC ResultSet (an interface and is also used to retrieve SQL select query results).
5. Close the connection

- These are the 4 main types of action commonly used with JDBC.
    - There are others.

LOAD A DRIVER



- Best to store driver name as string – if driver is replaced, only 1 line of code where driver name is supplied needs to be changed.
- Class.forName(driverNameStringVariable) – JDBC call that prog must make if it wants to use dtb.
    - JDBC driver will be loaded into class dynamically at *run time*.
    - Enrichment: forName() method creates Driver class obj and registers it with the DriverManager service.
        - forName() method is static

Loading a Driver

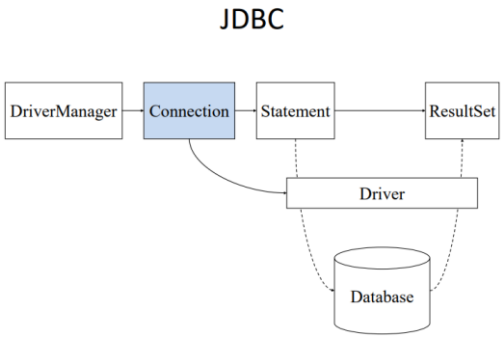
Tells which DBMS
actually which Driver Manager

Class.forName([location of driver]);

To use JDBC-ODBC bridge driver
Class.forName("sun.Jdbc.odbc.jdbcodbcDriver");
To use vendor-specific driver
Class.forName("oracle.Jdbc.Driver.OracleDriver);

Best to use a constant:
String DRIVER = " - - - ";
Class.forName (DRIVER);

MAKING A CONNECTION

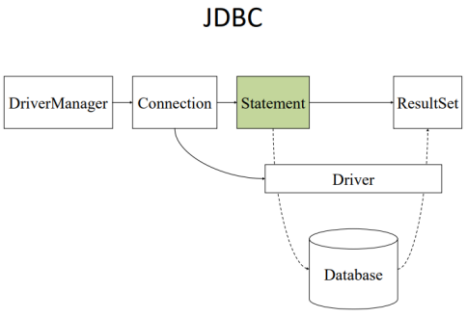


- Obtain dtb URl from dtb admin - dtb URL is first par of getConnection() method.
  - URL for dtb includes protocol (JDBC), vendor, driver server and port #
  - It is once again better to save these values to variables as strings (for the same reasons as mentioned for loading a driver).
  - Connection conn = DriverManager.getConnection(dtb URL)
- Can make and close connection multiple times within a program so as to reduce the load being placed on the dtb.
  - Doing so frees up the dtb connection when the processing doesn't require dtb access.

Making a Connection

```
import java.sql.*;
Connection conn= DriverManager.getConnection
("jdbc:oracle:thin:@server.cs.uct.ac.za:1521:CS01",
 loginname, passwd);
```

CREATING JDBC STATEMENT



- JDBC statement obj:
  - Is associated with an open connection
  - Is like a channel on the connection for passing statements to the DBMS.
  - Has no SQL specified or executed *yet*.
  - Statement stmt = conn.createStatement();
- createStatement() usually doesn't need parameters – it's enough to just call the createStatement() method of the obj returned by the getConnection call.

Examples

Creating a statement

```
Statement st = conn.createStatement( );  // example 1

Statement st =conn.createStatement(
        ResultSet.TYPE_SCROLL_SENSITIVE,
        ResultSet.CONCUR_UPDATABLE ); // example 2
```

TYPE_SCROLL_SENSITIVE means can do myResultSet.previous() – if left out this defaults to TYPE_FORWARD_ONLY

CONCUR_UPDATABLE means the result set can be updated – if left out this defaults to CONCUR_READ_ONLY i.e. no database changes

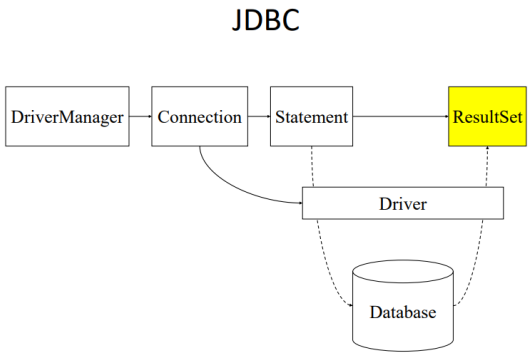
- ResultSet objects hold the results of a query. They have a cursor.
- TYPE_FORWARD_ONLY: The result set cannot be scrolled; its cursor moves forward only, from before the first row to after the last row. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved. This is the default.
- TYPE_SCROLL_INSENSITIVE: The cursor can move forward and backward, and to an absolute position. The result set is insensitive to changes made to the underlying data source while it is open. It contains the rows that satisfy the query at either the time the query is executed or as the rows are retrieved.
- TYPE_SCROLL_SENSITIVE: The cursor can move both forward and backward, and to an absolute position. The result set reflects changes made to the underlying data source while the result set remains open.
- CONCUR_READ_ONLY: The ResultSet object cannot be updated using the ResultSet interface.
- CONCUR_UPDATABLE: The ResultSet object can be updated using the ResultSet interface
- **Note**: Not all databases and JDBC drivers support all ResultSet types.

EXECUTING

- Statement object: obj returned by createStatement() method, contains different methods to run the different types of SQL statements.
- Create a standard SQL syntax statement and *pass it to statement object* as a string.
- This statement will execute one of two things:
  - An update – the statement will **change the database**
    - executeUpdate() statement used – takes string that'll be given to DBMS to compile and run in order to change the data.
      - Changes the state of the dtb.
    - Examples:
      String SQL = "INSERT INTO rel" + " VALUES ('246')";
      st.executeUpdate(SQL);

        - stmt.executeUpdate (“Create table Member “ +
          “(Member_Id number(4) primary key, ” +
          “Member_Name Varchar2(15))”);
        - stmt.executeUpdate("insert into Member values” + “(12, ‘Jim Hu’)");
    - A query – statement fetches values
        - This is a *problem,* because query results are <u>always relations</u>.
            - This implies that a ResultSet obj is returned.
            - ResultSet obj can be used to process the rows of that table one by one.
        - Example:
          String SQL = "SELECT snum FROM suppliers";
          ResultSet rec = st.executeQuery(SQL);

USING RESULTSET



- executeQuery() returns ResultSet obj.
- resultSet Obj has methods
    - rs.next(): method returns the first row when it’s first called, then 2nd when it’s called second and so on.
        - It returns the next row OR zero if there’s no more rows next.
    - getString, getInt etc are methods that take either a col name/col number as an argument and return the corresponding value in that column of the current row.

Examples

Example of executeQuery( )

```
ResultSet rs;
rs = stmt.executeQuery ("select * from Member");
while ( rs.next ( ) )
{
   int memId = rs.getInt (1);
   String mName = rs.getString (2);
   System.out.println ("Member Id:" + memId +
                       "Member Name:"+ mName);
}
```

Using ResultSet to print all tuples of a relation

```
String query = "select * from Passing";

ResultSet rs = stmt.executeQuery(query);

while ( rs.next( ) ) {
   String ID = rs.getString("ID");
   String name = rs.getString("NAME");
   int marks = rs.getInt("MARKS");
    System.out.println(ID+","+name+":"+marks)
}
```

Extract attributes of interest from each tuple by giving the column name or column number

```
while( rs.next( ) )
{ System.out.println( rs.getString("ID") );
}
// or
while( rs.next( ) )
 { System.out.println( rs.getString(1) );
  // first column of the resultset
}
```

Scrollable and Updateable

- Prg processes rows in different order, rather than each row in order – specify this in createStatement() with TYPE_SCROLL_SENSITIVE (TYPE_FORWARD_ONLY is default)
- Some of the rows retrieved will be updated - specify this in createStatement() with CONCUR_UPDATABLE (CONCUR_READ_ONLY is default).

Examples

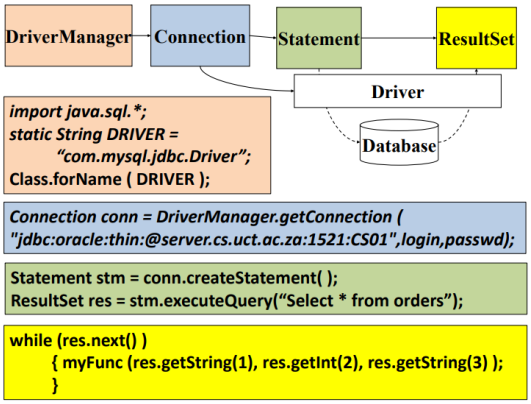
Scrollable Result Set

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_SENSITIVE,
                    ResultSet.CONCUR_READ_ONLY);

String query = "select students from class where type='not sleeping' ";
ResultSet rs = stmt.executeQuery( query );

rs.previous();  / / go back to previous tuple in the ResultSet
rs.relative(-5); / / go back 5 tuples
rs.relative(7); / / go forward 7 tuples
rs.absolute(10); / / go to 10th tuple of ResultSet
```

Updatable ResultSet

```
...
Statement stmt =
con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
                    ResultSet.CONCUR_UPDATABLE);
String query = " select students, grade from class
                 where type='attending' ";
ResultSet rs = stmt.executeQuery( query );
...
while ( rs.next() )
{
   int grade = rs.getInt("grade");
   rs.updateInt("grade", grade+10);
   rs.updateRow();
}
```

CLOSING UP



- There is the process – loading driver, connecting to dtb, creating and executing statements, processing rows of any resulting tbl.
- This process if thus concluded with closing resources no longer needed – achieved by using their close() methods.
    - StatementName.close(); ConnectionName.close()

Optimisation

Query optimisation

1. When using a SELECT statement and just want to find if any row matches a criteria e.g. String query = "SELECT city FROM t where city = 'PE' ";
    a. Best to use a SELECT --- LIMIT 1 rather than wasting DBMS time by looking for all.
    b. Cheaper way of finding out if such data exists, because table scan will stop as soon as first such tuple is found.
    c. Example:
        i. Recall that rs.next( ) returns 0 if there is no row to fetch.
        ii. This example is using a return value of 0, meaning no such row, to determine if there are any customers from PE.
        iii. String query = "SELECT city FROM t where city = 'PE' LIMIT 1";
        ResultSet rs = stmt.executeQuery(query);
        IF (rs.next( ) > 0) { - - - }
2. Avoid using SELECT * FROM ---
    a. Wastes memory space, network traffic and disk reads.
    b. Instead use SELECT col1, col2 etc FROM ---
3. For mySQL, use mySql unbuffered
4. query to start working on result tuples as soon as they're in memory, rather than wait for the whole result to be fetched first.
5. Do large INSERTS, DELETES and UPDATES in small batches – avoids locking tables for too long.
    a. Can do this by adding LIMIT to statement
    b. Example:
    while (1) { mysql_query("DELETE FROM t WHERE - - - LIMIT 10000"); if (mysql_affected_rows() == 0) { break;} // break out of loop, done usleep(60000); }
    c. Smaller batches are processed separately
    d. This can ONLY be done when logic allows for this.
        i. Can do: UPDATE R SET COURSE = 'CSC2001F' WHERE COURSE = 'CS2' LIMIT 10000
            1. Safe to do in batches because each row will only be changed once. When update executed again, row no longer has "CS2" value.
        ii. CANNOT do: UPDATE R SET MARK = MARK * 2 LIMIT 10000
            1. Not safe because will be applied to the same 10000 rows repeatedly.
    e. Advantage of limiting operations to small batches is that it gives other concurrent apps a chance to use that table.
        i. If update is critical, may not want to do that but for a lot of situations it won't matter.
6. One INSERT of many VALUES lists (many rows) at a time is much faster than many INSERTS of 1 row at a time.
    a. This is because each dtb operation requires connecting to dtb, sending query, parsing query, inserting data and updating index.

b. In the same manner, LOAD DATA INFILE (from text/csv file) is faster than using inserts.

Schema optimisation

1. Data types must be carefully chosen so as to save space and time wherever possible.
    a. So, for example, using VARCHAR instead of CHAR is not worthwhile unless VARCHAR saves a considerable amount of space.
    b. This is especially needed when tables are large and/or frequently accessed.
    c. Time taken to read and write to disk is the slowest part of any dtb application.
2. Use ENUM instead of VARCHAR or CHAR when domain has few possible values and these values can all be listed
    a. Saves space
    b. Faster to query.
    c. Example:
    Faculty ENUM('SCI', 'HUM', 'COMM', 'HEALTH', 'LAW', 'EBE')
3. For tables representing entities, use UNSIGNED integers as pk rather than VAR/CHAR properties of entities - VAR/CHAR fields are slower.
4. Use SMALLINT, MEDIUMINT, TINYINT where possible – saves space and time.
    a. Also use DATE instead of DATETIME for the same reasons, where possible.
5. Tables with rows all same length are faster to use – position of any row can be calculated and sys can go straight there without having to lookup where to go in an index first.
    a. Use CHAR instead of VARCHAR if it'll make rows fixed size.
    b. Cols of type TEXT/BLOB means rows will NOT be fixed size.
    c. Can also keep variable sized columns in separate table IF it won't need too many joins.
6. If some cols are updated far more frequently than others, or are read far less than others, put them in a separate table – UNLESS joins will often be needed.
    a. Data that's large, seldom changes and seldom used (a Comment field, for example) – usually kept in separate table.
    b. JOIN to that table will be needed when need that data, it will not be needed often, and so this rare event is not as important as making frequent queries run faster because table takes up fewer disk blocks.

Optimisation Assistance

- DBMS will usually provide tools and functions to improve dtb performance.
    - DBMS keeps metadata about tables, cols, indexes and so forth – uses this when planning fastest way to execute dtb statement.
    - Can access metadata and the plans DBMS evals for your statements before choosing best plan to execute.
- Some examples of functions:
    - Run PROCEDURE_ANALYSE() to get suggestions of how to improve dtb structure – data must be already loaded into dtb.
    - ANALYZE TABLE tableName; will cause mySQl to access table and show distribution of its keys.

- Run SQl with EXPLAIN added in front of a statement – shows how mySQL will execute it. Can then maybe see how to make it more efficient.

EXPLAIN OUTPUT

- EXPLAIN lists tables in order they'd be read.
- Key: which of these indexes is chosen to use in this query
- Key_len: size of the key used (e.g. only first part of composite key)
- Ref: the columns that will be looked up in this index
- Rows: estimate of how many rows will be fetched
- Type:
    - CONST (at most that number of rows fetched)
    - EQ_REF (only the 1 row with the required index value will be fetched)
    - RANGE (only the rows with index value in the required range will be fetched)
    - REF (only those rows with the required index value will be fetched)
    - ALL (all rows in the table will be fetched) or INDEX (the entire index will be scanned instead of the entire table - index smaller and contains all the values that are needed)
- The *product of the values* in the **rows** column gives an idea of query performance.

Example

```
EXPLAIN SELECT a,b FROM R JOIN T ON R.id=T.id
WHERE R.id BETWEEN 4 AND 8
```

Plan 1: Explain output shows index usage for 1 table only, hence expensive

id	---	table	Type	possible_keys	key	key_len	ref	rows	Extra
1		R	ALL	NULL	NULL	NULL	NULL	8884	using where
1		T	eq_ref	PRIMARY	PRIMARY	4	Mydb.T.id	1	

Plan 2: Explain output shows index usage for both tables, thus much faster to execute

id	---	table	type	possible_keys	key	key_len	ref	rows	Extra
1		T	range	PRIMARY	PRIMARY	4	NULL	5	using where
1		R	ref	R.Id	R.Id	4	Mydb.R.id	20	

In the example on the last slide, Plan 1 is to go through all rows in relation R and lookup the id in T using the primary key index of T. It is thus 8884 rows of R and of T, hence double 8884 rows fetched altogeth, total 17 768.
Plan 2 is to use the primary key index of T to find id values 4 to 8, and then use the R index on id to fetch the matching R rows for each of those 5 T tuples. Clearly R is a relationship table, and it is estimated each T has about 4 rows that reference it in R. So 5 T rows and about 20 R rows so total about 25.

XML