Course DVG B03: Data Structures & Algorithms – Compendium

Course Structure

1. Introduction: basic data structures and operations (5 lectures)
2. Sequences: sorting, searching and hashing (2 lectures)
3. Performance (1 lecture)
4. Trees: general, binary, BST, AVL, B-trees (3 lectures)
5. Graphs: directed, undirected (4 lectures)
6. Revision and exam technique (1 lecture)

Course goals

1. Background: Abstraction, modelling & collections
2. Data structures (set, sequence, tree, graph) and operations
3. Algorithms: Tree: AVL add, heap, Graph: Dijkstra, Floyd, Warshall, Prim, Kruskal, Strong Components, Articulation Point
4. Labs: Understanding and implementation of the above
5. Abstraction: Modelling, Implementation, Collection
6. Data Structures: Set, Sequence (List, Stack, Queue), Tree, (General, Binary, Binary Search, AVL, B-tree), Graph (Directed, Undirected)
7. **Introduce ABSTRACTION & ABSTRACT THINKING**
8. Create a mental toolbox
9. Improve C programming
10. **Introduce the concept of Abstract Programming**
    (Language & implementation independent)

Data structures and operations – these can be generalised as collections

1. set: is_empty, add, remove, is_member, display, cardinality
2. sequence: is_empty, add, remove, find, display, cardinality
3. tree: is_empty, add, remove, find, display, navigation, cardinality
4. graph: is_empty, add, remove, find (node, edge), display, cardinality, searches

Abstraction

1. modelling abstraction: real world ➔ entities, attributes, relationships
2. abstract data types: implementation independent
3. collection abstraction: set, sequence, tree, graph as collections
   Collection = entities & attributes + relationships

Abstraction – working definitions

1. ADT (abstract data type) = ADS (abstract data structure) + operations
2. ADS = abstract set, sequence, tree graph
3. DT (data type) = DS (data structure) + operations
4. DS (data structure) = implementations of set, sequence, tree, graph – the most common DSs in programming languages are arrays and structures (records)

Computer Science

1. frequently deals with **collections of information** and how to organise information
2. most important operation is **SEARCHING**
3. SORTING is an aid to improved searching
4. requires efficient methods of organising and searching information
5. this in turn gave rise to the study of the **data structures**: set, sequence, tree, graph
6. these data structures are used in every branch of computer science

Modelling

1. consists of abstracting information from the **real world** to make **a computer model** which may then be represented by data structures and manipulated via algorithms
2. from the database world we can use the **Entity-Relationship model**[1] (Chen) where real world objects are represented as **entities** with **attributes** and where **relationships** (with attributes) may exist between entities
3. Example: students and courses – the model contains only relevant information
    a. real world student: (name, address, hair colour, eye colour, university…)
    b. model student: (sname, address, telephone number, subject, id number)
    c. model course: (cname, code, level, offering, lecturer)
    d. courses/student relationship: (id number, code)
    e. each of these (b, c, d) may be represented by sets

Language & Modelling

1.	noun	➔ entity	e.g. student
2.	adjective	➔ attribute	e.g. third year student
3.	verb	➔ relationship	e.g. student takes course

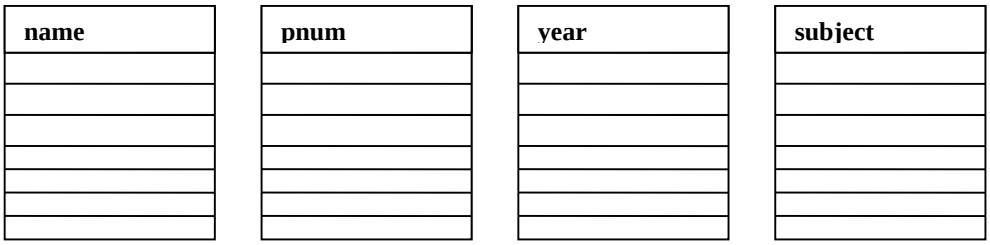
Programming Languages

1. provide **(atomic / predefined) data types**: integer, real, character, Boolean
2. provide **constructed data types** usually via array and structure (record)
3. provide **basic operations** on predefined data types: arithmetic, logic
4. provide **functions/procedures**/methods as ways of defining new operations using previously defined operations – basic + user defined
5. example: linked lists using structures + pointers
6. example: display the list as a function “display()”
7. an “object” in OO programming is an entity + operations (methods)

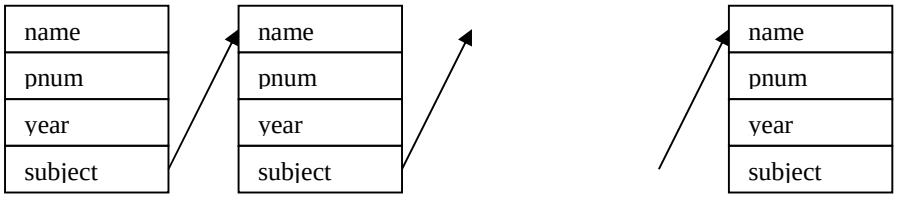
[1] http://en.wikipedia.org/wiki/Entity%E2%80%93relationship_model

Implementing collections of entities and attributes

1. e.g. student: (string: name, string: pnum, integer: year, integer: subject)
2. arrays – one per attribute



3. structures and pointers



4. similarly trees and graphs may be implemented
   (Usually with structures & pointers)
5. set & get operations may then be implemented for each attribute to keep the rest of the code independent of the implementation details (array/structure)
6. using the set/get operations, other operations may be implemented
7. this may be done in a "backend" module as in C or using an object (class) in OO
8. this backend is the implementation of our ADT (➔ ADS + operations)

ADTs, properties and CS applications

SET[2]

Definition:	a collection of objects (entities)
Operations:	union, intersection, difference, subset, is_member, cardinality (number of objects), is_empty, add (member), remove (member), find (member i.e. is_member), display
Properties:	unique entities, unordered
CS applications:	relational databases[3]

SEQUENCE[4]

Definition:	an ordered collection of objects (entities)
Relationships:	successor / predecessor
Operations:	concatenate, difference, sub-sequence, is_member, cardinality, is_empty, add (entity), remove (entity), find (entity), display, add, remove & find by position (first, nth, last)
Properties:	non-unique entities (duplicates), ordered (position attribute) may be sorted (**note the difference between ordered and sorted!)**
CS applications:	string, array, program (sequence of instructions)

Recursive defn:

Sequence ::= Head Tail | empty
Head ::= element
Tail ::= Sequence

Restrictions: add, remove at position first ➔ stack[5] (LIFO)
add at position last, remove at position first ➔ queue[6] (FIFO)

[2] http://en.wikipedia.org/wiki/Set_theory
[3] http://en.wikipedia.org/wiki/Relational_model
[4] http://en.wikipedia.org/wiki/Sequence
[5] http://en.wikipedia.org/wiki/Stack_%28abstract_data_type%29
[6] http://en.wikipedia.org/wiki/Queue_%28abstract_data_type%29

TREE[7]

Definition: A hierarchical collection of objects, each node may have n children

Operations: traversals: pre-, in- post-order, general traversal, depth-first, breadth-first
cardinality, height, add (node), remove (node), find (value), display

Properties: unordered, ordered, root node, leaf nodes (no children)

CS applications: parse/syntax trees in compiling

BINARY TREE[8]

Definition: A hierarchical collection of objects, each node may have at most 2 children

Operations: traversals: pre-, in- post-order, general traversal, depth-first, breadth-first
cardinality, height, add (node), remove (node), find (node), display

Properties: ordered (Left_child, Right_child), root node, leaf nodes (no children), height (number of levels / longest path length)
full: every node has zero or two children
perfect: for height h there are $2^h - 1$ nodes
complete perfect on the next lowest level and filled from the left on the lowest level

CS applications: parse/syntax trees in compiling, arithmetic expression trees, File system organisation (directories)

Recursive defn:

BT	**::=**	**Left_child Node Right_child | empty**
Left_child	**::=**	**BT**
Node	**::=**	**element**
Right_child	**::=**	**BT**

Algorithms: AVL add, heap

[7] http://en.wikipedia.org/wiki/Tree_%28data_structure%29
[8] http://en.wikipedia.org/wiki/Binary_tree

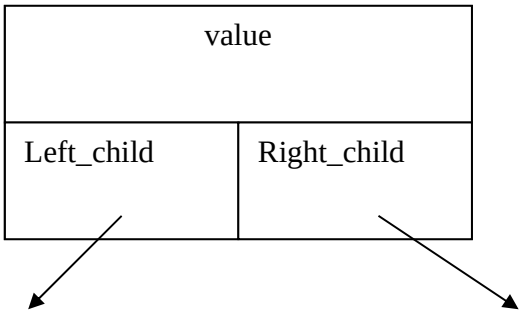
GRAPH[9]

Definition:	**G = (V, E)** **V is a set of Vertices[10] (nodes),** **E is a set of Edges**
Operations:	node-cardinality, edge-cardinality, is_empty, display, add (node/edge), remove (node/edge), find (node/edge), is_edge(A, B), is_path(A, B), neighbours(N), traversals: depth-first, breadth-first in-degree(N), out-degree(N)
Properties:	unordered (set), directed, undirected
CS applications:	modelling networks (computer, transport), decision & planning systems (e.g. PERT[11] charts)
Algorithms:	Dijkstra[12], Dijkstra shortest path tree, Floyd, Warshall[13] Prim's[14], Kruskal's[15]

[9] http://en.wikipedia.org/wiki/Graph_%28abstract_data_type%29
[10] One vertex, two vertices or two vertexes
[11] http://en.wikipedia.org/wiki/Program_Evaluation_and_Review_Technique
[12] http://en.wikipedia.org/wiki/Dijkstra%27s_algorithm
[13] http://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm
[14] http://en.wikipedia.org/wiki/Prim%27s_algorithm
[15] http://en.wikipedia.org/wiki/Kruskal%27s_algorithm

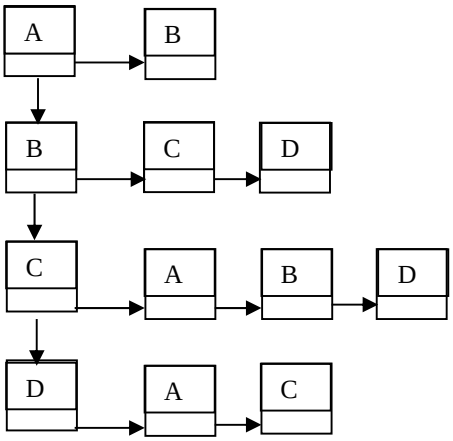
BT Implementation

Typically a structure (value, left_child, right_child).



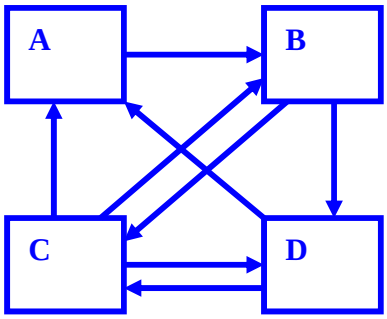
Graph Implementation

Adjacency List



Adjacency Matrix

	A	B	C	D
A		1		
B			1	1
C	1	1		1
D	1		1	



SEQUENCE – Ordered – possibly sorted

Implementations:	arrays (value, next, (previous)), structure & pointers (ditto)			
Operations:	is_empty:	S	➔ Boolean	
	cardinality:	S	➔ integer	
	add_pos:	S x value x pos	➔ S	
	rem_pos:	S x pos	➔ S	
	find_pos:	S x pos	➔ value	
	display:	S	➔ S	
	make_el:	value	➔ element	
Operations: (**Stack**)	add_pos:	S x value x first	➔ S	(push)
	rem_pos:	S x first	➔ S	(pop)
	find_pos:	S x first	➔ value	(TOS)
Operations: (**Queue**)	add_pos:	S x value x last	➔ S	(enqueue)
	rem_pos:	S x first	➔ S	(dequeue)
Operations: (**sorted**)	add:	S x value	➔ S	
	rem:	S x value	➔ S	
	find:	S x value	➔ Boolean	
Operations: (**sequential** view)	get_first:	S	➔ element	
	get_next:	S x pos	➔ element	(pos implicit)*
	is_EOL:	S x pos	➔ Boolean	(pos implicit)

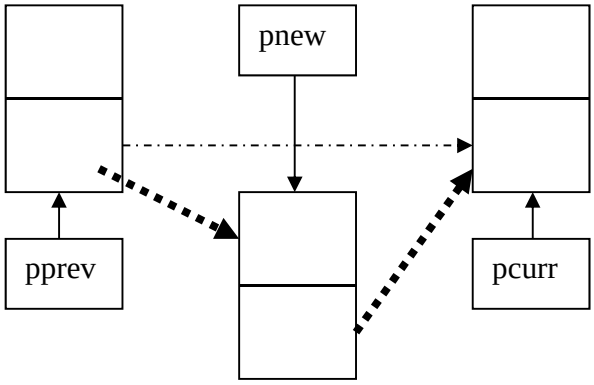
* usually the backend keeps 2 references previous and current which are moved by get_next – these references are private attributes

Operations: (Recursive view)	**head:**	**S**	**➔ S**
	tail:	**S**	**➔ S**
	cons:	**element x S**	**➔ S**

In the recursive versions of the operations, the sequence is DECONSTRUCTED into a head + tail and then RECONSTRUCTED on the return from the recursive calls – each call returns a “new” sequence. Note that the reference to an element and to S, are the same type. The element may be envisaged as a sequence of 1 element.

New Sequences are always **CONS**tructed by adding an element at the head of the sequence (this avoids the need for the previous and current references in the sequential version).

SEQUENCE – Iterative add



Note

1. add at beginning pprev = NULL; pcurr = NULL (empty sequence) / = not NULL (non-empty sequence)
2. add in middle pprev = not NULL, pcurr = not NULL
3. add at end pcurr = NULL, pprev = NULL (empty sequence) / = not NULL (non-empty sequence)

```
set_pos (tvalue value) {  // first_el() initializes pprev and pcurr
                          // next_el() moves pprev and pcurr forward
   /* link in ascending order */
   first_el(); while(!is_eol() && (fvalue>get_value())) next_el();
   return;
}

void linknew() {  // between pprev and pcurr

   if (is_prevempty()) liststart = pnew;      //1
   else set_next(pprev, pnew);                //2
   set_next(pnew, pcurr);                     //3
   return;
   }
```

add(tvalue value) { set_pos(value); pnew = make_el(value); linknew(); }

1. **add at beginning**
   liststart = pnew (//1) and set_next(pnew, pcurr) (//3) -- pprev == NULL

2. **add in middle**
   set_next(pprev, pnew) (//2) and set_next(pnew, pcurr) (//3)

3. **add at end**
   set_next(pprev, pnew) (//2) and set_next(pnew, pcurr) (//3) -- pcurr == NULL

SEQUENCE – Recursive add

```
seqref add(seqref S, valuetype value)
{
        return is_empty(S)                      ? make_el(value);          //1

                : value < get_value(head(S))    ? cons(make_el(value), S); //2

                : cons(head(S), add(tail(S), value));                      //3
}
```

1. **add at the beginning – add((2, 4), 1);**
   - **a.** is_empty(S) ➔ false //1
   - **b.** 1 is less than 2 ➔ return cons(make_el(1), (2, 4)) ➔ (1, 2, 4) //2
   - **c.** finished!

2. **add in the middle – add((2, 4), 3);**
   - **a.** is_empty(S) ➔ false //1
   - b. 3 not less than 2 //2
   - c. cons((2), **add**((4), 3)) //3
     - i. is_empty ➔ false **//1**
     - ii. 3 less than 4 return cons(make_el(3), (4)) ➔ (3, 4) **//2**
   - d. cons((2), (3, 4)) ➔ (2, 3, 4) //3
   - e. finished!

3. **add at the end – add((2, 4), 5);**
   - **a.** is_empty(S) ➔ false //1
   - b. 5 not less than 2 //2
   - c. cons((2), **add**((4), 5)) //3
     - i. is_empty ➔ false **//1**
     - ii. 5 not less than 4 **//2**
     - iii. cons((4), **add**((), 5)) **//3**
       1. is_empty(()) ➔ true, return (5) **//1**
     - iv. cons((4), (5)) ➔ (4, 5) **//3**
   - d. cons((2), (4, 5)) ➔ (2, 4, 5) //3
   - e. finished!

Recursive remove is similar – in step //2 : value == get_value(head(S)) ? tail(S);

STACK LIFO (Last in First out)

Operations: Push (add); Pop (remove); Top of Stack (ToS) – top (first) element (peek)

May be implemented using a sequence with position ➔

- push = add(value, **first**)
- pop = remove(**first**)
- Tos = find(**first**)

Often associated with depth-first searches

Queue (First in First out)

Operations: Enqueue (add); Dequeue (remove);

May be implemented using a sequence with position ➔

- enqueue = add(value, **last+1**)
- dequeue = remove(**first**)

Often associated with breadth-first searches

General Tree ➔ Binary Tree

1. The first child becomes the left child of the parent
2. The subsequent children become the right child of their predecessor

Binary Tree (BT) / Binary Search Tree (BST)

Implementations: usually structures & pointers;
3 arrays (i) value (ii) left child (iii) right child;

array (heap structure) where the root is in A[1] and in general the left child is in position 2 * position(parent) and the right child in position 2 * position(parent) + 1 so for example the left/right child of the root are in A[2], A[3] respectively; the children of A[2] will be in A[4], A[5]; those of A[3] in A[6], A[7].

Note that the BST is sorted and thus it is possible to add/remove nodes without reference to position.

Operations: is_empty, cardinality (nodes), height
add (node), remove (node), find (value), display
traversals: pre-, in-, post-order, depth-first, breadth-first

Properties: ordered, each node has max 2 children, BST is sorted

Full:	BT every node has either 2 or 0 children
Perfect:	BT of height h has exactly 2^h -1 elements
Complete:	BT perfect on the next lowest level and the lowest level is filled from the left

CS applications: parse/syntax trees in compiling, arithmetic expression trees,
File system organisation (directories)

Operations are usually implemented recursively. Example – number of nodes

```
int card(BT)
{   if is_empty(BT) return 0; else return 1 + card(LC(BT)) + card(RC(BT)); }
```

OR

```
int card(BT) { return is_empty(BT) ? 0 : 1 + card(LC(BT)) + card(RC(BT)); }
```

BST Implementation – Binary Search Tree

Usually with a structure node (LCref, value, RCref)

The cons operation now becomes **cons(LC, Node_el, RC)**
Cf head/tail for a list – now we have LC, RC ("tail"), Node ("head")

The add operation becomes

```
BST add(BST T, int v) {
  if is_empty(T) return create_el(v);
  if v < value(node(T))  return cons ( add(LC(T), v), node(T), RC(T))
  if v > value(node(T))  return cons ( LC(T), node(T), add(RC(T), v))
  return T; // no duplicates
}
```

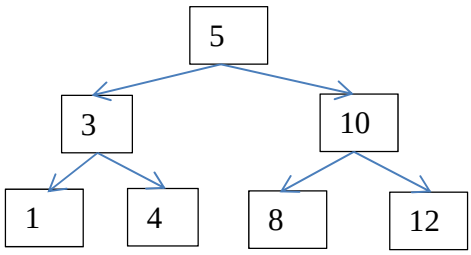
See the example below

Firstly consider the add function

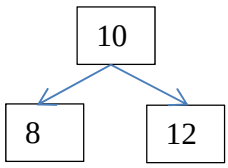
```
BST add(BST T, int v) {
  if is_empty(T) return create_el(v);                                  //case 1
  if v < value(node(T))  return cons ( add(LC(T), v), node(T), RC(T))  //case 2
  if v > value(node(T))  return cons ( LC(T), node(T), add(RC(T), v))  //case 3
  return T;  // no duplicates                                          //case 4
}
```

Note that below, the left and right sub-trees are represented by their root node value

Initial call
Start with an example BST and add 9 (see below and note the case number from above
case 3: **cons (LC(T), node(T), add(RC(T), v))** **i.e. cons (3, 5, add(10, 9))**



Step 1: The first recursive call is **case 3** which "rephrases the question" as add 9 to the following BST
case 2 : cons (add(LC(T), v), node(T), RC(T)) **i.e. cons (add(8, 9), 10, 12)**

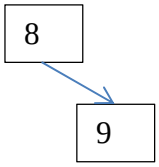


Step 2: The next recursive call is **case 2** which "rephrases the question" as add 9 to the following BST
case 3 : (LC(T), node(T), add(RC(T), v)) **i.e. cons (¤, 8, add(¤, 9))**

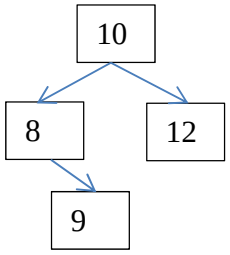
This becomes **case 1** which returns a tree containg a single node (9) to **step 2** giving

case 1: if is_empty(BST) return create_el(v); //case 1

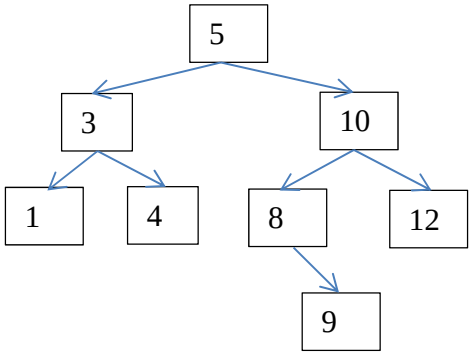
Step 2 then returns the following tree to **step 1**
cons (LC(T), node(T), add(RC(T), v)) **i.e. cons (¤, 8, add(¤, 9))**



Step 1 then returns the following tree to the **initial call**
cons (add(LC(T), v), node(T), RC(T)) **i.e. cons (add(8, 9), 10, 12)**



which in turn returns the tree below as the final result.
cons (LC(T), node(T), add(RC(T), v)) **i.e. cons (3, 5, add(10, 9))**



And the process is complete

Note that in the above, the left and right sub-trees are represented by their root node value

The remove operation is a little more tricky – see the literature.

Swap the value with a leaf node, remove the leaf node and check that any constraints on the tree (BST, AVL) are still met.

```
BST remove(BST T,  int v)
{
  if IsEmpty(T) then return T                                  //case 1
  if v < value(T) then return cons(remove(LC(T), v), T,  RC(T))  //case 2
  if v > value(T) then return cons(LC(T), T, remove(RC(T), v))   //case 3
  return remove_Root(T);   // return a BST with the (local) root removed  //case 4
}
```

LC, RC = return left and right child respectively

Where remove_Root is left as an exercise for the lab. Work through an example of this and note what is passed in the recursive calls AND what is returned on the way back from the recursive calls i.e. the deconstruction / reconstruction process.

Remove_Root has 4 cases

i) the tree is a leaf node i.e. no left nor right child
ii) the tree has a left child only
iii) the tree has a right child only
iv) the tree has both a left and right child

The solution for cases (i), (ii) and (iii) are

i) return the empty tree
ii) return the left child
iii) return the right child

N.B. check that you understand why this is the case!
Draw the corresponding pictures

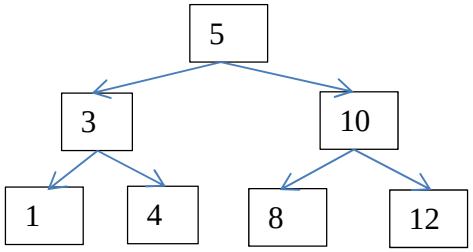
Case 4 has 2 solutions

a) replace the node (local root) value with the maximum value of the left child
b) replace the node (local root) value with the minimum value of the right child

and then reconstruct the tree as

a) cons(remove(LC(T), max), create_el(max), RC(T))
b) cons(LC(T), create_el(min), remove(RC(T), min))

Now take an example and work through it as was done above for add



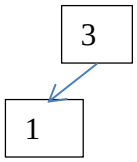
remove(5,**5**)
remove_Root(5,**5**)
cons(remove(3, **4**), 4, 10)
cons(1, 3, remove(4, **4**))
remove_Root(4,**4**) **STOP!**
return ¤
return (1,3,¤)
return (3,4,10) ➔result
root values given for LC, RC

And remove the root (5)

For case (a) in remove_Root we have
cons(remove(LC(T), max), create_el(max), RC(T)) i.e. cons(remove(3, 4), 4, 10)

for the remove(3, 4) this gives
if v > value(T) then return cons(LC(T), T, remove(RC(T), v)) **//case 3**
i.e. cons(LC(T), T, remove(RC(T), v)) i.e. cons(1, 3, remove(4, 4))

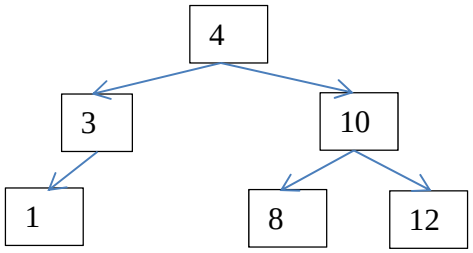
this calls remove_Root again with case (i) that the node is a leaf node in which case the empty tree is returned so the cons above gives cons(1, 3, ¤) giving the tree



Which in turn is returned to the first cons above –

cons(remove(LC(T), max), create_el(max), RC(T)) i.e. cons(remove(3, 4), 4, 10)

to give the tree



Which is the required result.

Exercise repeat the above for the case where the minimum of the right child is used instead.

Graphs: Directed & Undirected

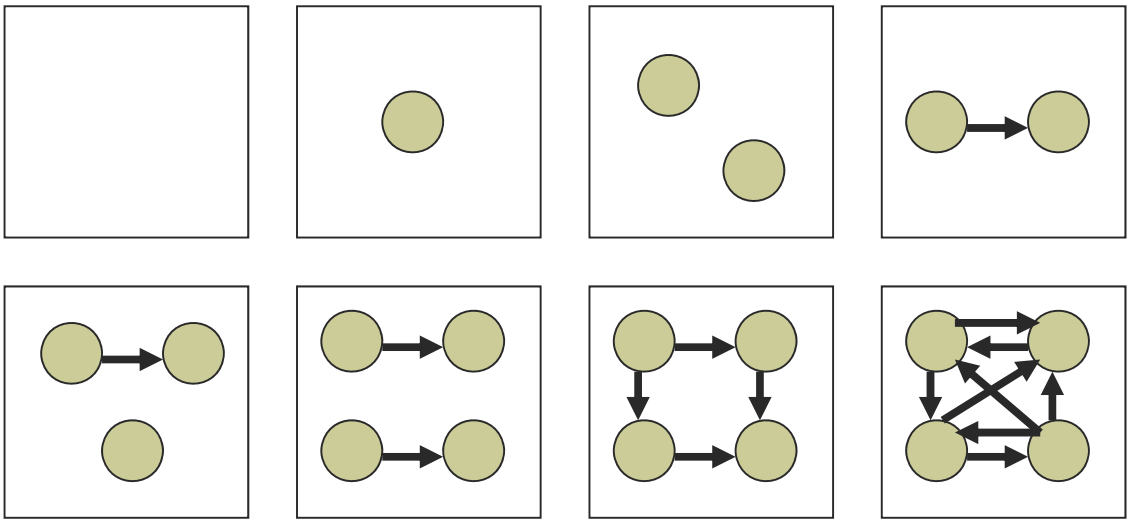
G=(V, E) – set of vertices + set of edges

Graphs may be implemented with adjacency lists or adjacency matrices. (See above).

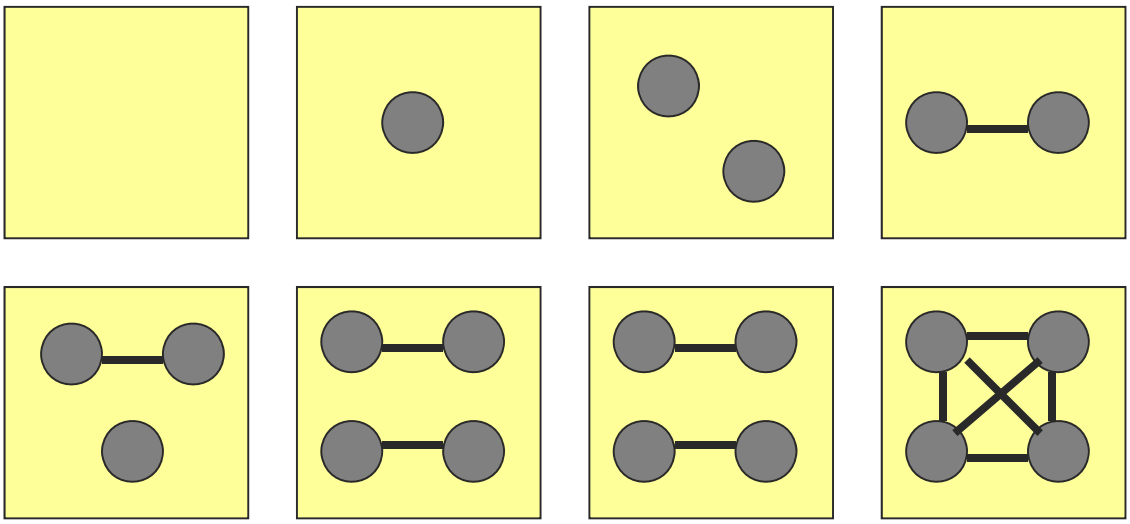
Note that a graph is a collection of vertices (nodes) + edges. Edges require 2 nodes for their existence and may be specified by (a, b) where a and b are nodes.

The edge may be directed (one way) (a, b) or undirected (two-way) {a, b} with the latter being implemented by 2 directed edges (a,b) and (b,a). This shows up as symmetry in the adjacency matrix, about the diagonal top-left to bottom-right.

Examples – Directed Graphs



Examples – Undirected Graphs



For terminology and more information see Wikipedia Graphs[16],[17]

In computer science graphs are used to model computer networks. They may also model transport networks. Graphs have also been used to model systems in other disciplines.

Examples of issues involving graphs are

CONNECTIVITY[18] – the degree to which nodes are connected
e.g. in a network (computer, transport) it is desirable that each node is connected to every other node for sending messages from A to B or travelling from A to B. Dijkstra's algorithm gives the shortest path from a given node to all other nodes. Warshal's algorithm gives the transitive closure of a graph showing whether all nodes are connected (the resultant matrix contains only one's) or not

The A2B (Dijkstra/Floyd) problem is one we look at in the labs.

Since graphs are slightly more complex data structures, there is more terminology[19] associated with a graph. Learn the following:-

- Vertex (Node)
  - Source node – a node with only outgoing edges
  - Sink node – a node with only incoming edges
  - Adjacent nodes – endpoints of the same edge
  - Articulation point – a node which when removed results in a disconnected graph
- Edge – connection between 2 nodes, a and b
  - Directed edge (a, b) a, b are nodes
  - Undirected edge {a, b} a, b are nodes
  - Weighted edge – the edge has a cost c associated with it
  - Parallel edges – multiple edges of the same type and end nodes
  - Self loop – an edge with the start and end vertex
  - Bridge edge – an edge which when removed results in a disconnected graph
- Graph
  - Simple Graph – no parallel edges nor self-loops
  - Directed Graph (DiGraph) – a graph with directed edges only
  - Undirected Graph – a graph with undirected edges only
  - Weighted Graph – a graph with weighted edges
  - Mixed Graph - a graph with both directed and undirected edges
  - Sub-graph – a subset of vertices and edges
  - Spanning sub-graph – a sub-graph containing all the vertices

[16] http://en.wikipedia.org/wiki/Graph_%28mathematics%29
[17] http://en.wikipedia.org/wiki/Graph_theory
[18] http://en.wikipedia.org/wiki/Connectivity_%28graph_theory%29
[19] http://www.csl.mtu.edu/cs2321/www/newLectures/24_Graph_Terminology.html

    - Spanning Tree – a spanning sub-graph that is also a tree
    - Tree – an undirected connected graph with no cycles
    - Acyclic Graph – a graph with no cycles
- Degree
    - In-degree – number of edges arriving at a node
    - Out-degree – number of edges leaving a node
- Path - a sequence of alternating vertices and edges starting and ending on a vertex
    - Simple path – a path with distinct vertices
    - Directed path – a path containing only directed edges
    - Length of a path is the number of edges in the path
- Cycle – a path that starts and ends on the same vertex
    - Simple Cycle – a path with unique vertices except for the first and last
    - Directed cycle – a cycle containing only directed edges
- Connected Graph – is any 2 vertices can be joined by a path
- Disconnected graph (has several components) – not connected

If you find any more definitions, bring then to the class and we can add these to the list.

SEQUENCES

Sorting and Searching

For efficient search a sorted collection is often preferred. There are 2 classes of sorting algorithms for sequences

1. **swap** methods: e.g. Bubble[20], **insertion**[21], selection[22], shellsort[23]
2. **divide and conquer** methods: **quicksort**[24], mergesort[25]

Hashing[26]

Hash function is a mapping of an input value (key) to an index value (usually integer). A Simple hashing function may be for example. H(key) → n where H is key mod p and p is usually a prime number. The distribution of the index values should be relatively uniform. If not collisions may occur i.e. H(key1) → and H(key2) → n

Collision handling and resolution is another area of Hashing. If collisions arise, the resolution algorithm must find a new space for the information i.e. a different index value. Usually this is H(key) + f(i) where I is the number of the collision (1, 2, …)

Some techniques are

1. separate chaining – create a list of elements at i
   (disadvantage search → O(n) instead of O(1))
2. linear probing – try n+1, n+2, … until a free position is found **f(i) = i**
   (disadvantage primary clustering)
3. quadratic probing – try $n+1^2$, $n+2^2$, … **$f(i) = i^2$**
   (disadvantage secondary clustering especially if load > 50%)
4. double hashing – f(i) is another hash function H_2

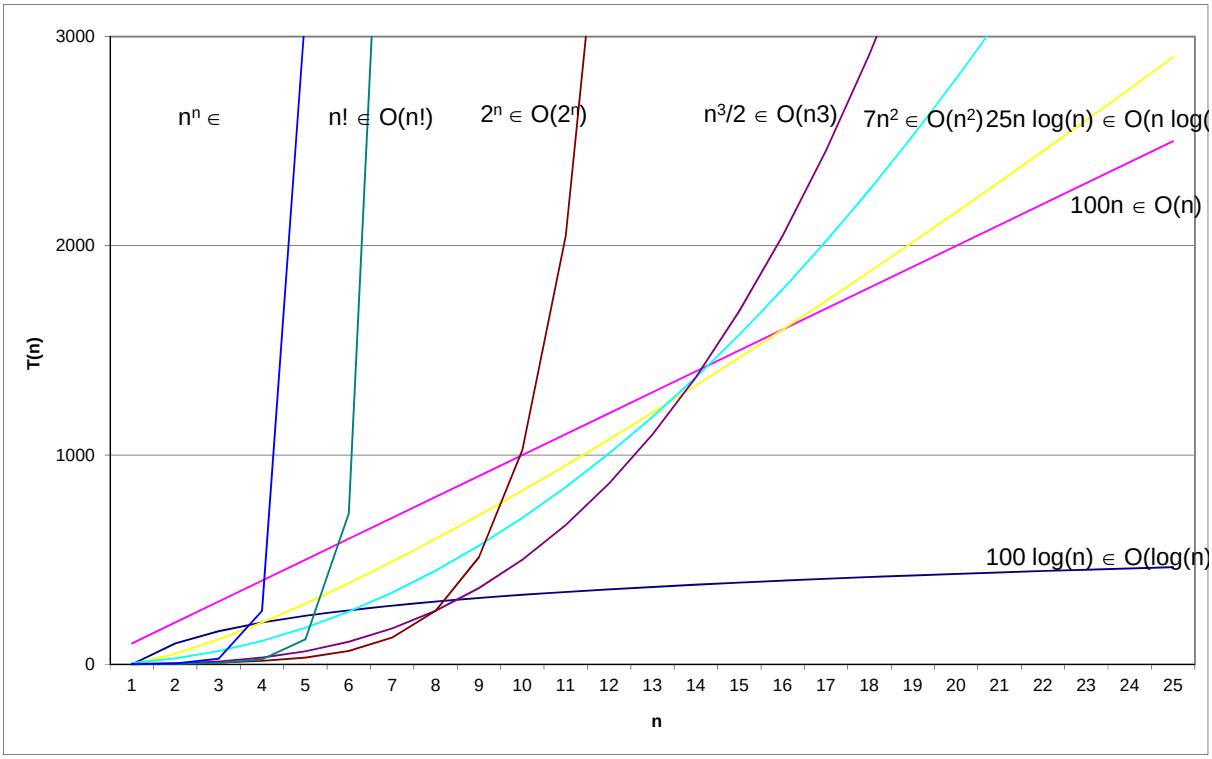
Other methods for disk systems involves overflow slots in the same page and on different pages.

Much research has been carried out on more sophisticated hash functions.

[20] http://en.wikipedia.org/wiki/Bubble_sort
[21] http://en.wikipedia.org/wiki/Insertion_sort
[22] http://en.wikipedia.org/wiki/Selection_sort
[23] http://en.wikipedia.org/wiki/Shellsort
[24] http://en.wikipedia.org/wiki/Quicksort
[25] http://en.wikipedia.org/wiki/Merge_sort
[26] http://en.wikipedia.org/wiki/Hash_function

PERFORMANCE

Performance is often stated in terms of **big-oh** – for example **O(1)** (constant), **O(log n)** (logarithmic), **O(n)** (linear), **O(n log n)** (loglinear), **O(n^2)** (quadratic), **O(n^3)** (cubic), **O(n^c)** (polynomial), **O(c^n)** (exponential), **O(n!)** (factorial) where n is the number of items in the collection. Usually this is a time performance e.g. time to sort n elements quicksort (O(n log n)), insertsort (O(n^2)). See the diagram below.



ALGORITHMS

TREES (1)

Convert a General Tree to a Binary Tree:

1. The **first child** becomes the **left child** of the parent
2. The **subsequent children** become the **right child** of their **predecessor**

Breadth-first Traversal:

```
BreadthFirst(T) {
        if T is not Empty {
                Q = Empty;
                Q = AddQ(Q, T);
                while(Q != Empty) {
                        p = front(Q); Q = deQ(Q);
                        process(Root(p));
                        if(Left(p)  != Empty) Q = AddQ(Q, Left(p));
                        if(Right(p) != Empty) Q = AddQ(Q, Right(p));
                }
        }
}
```

Depth-first Traversal: Pre-order, In-order and Post-order:

```
PreOrder(T) {
if !is_empty(T) { process(Root(T)); PreOrder(Left(T)); PreOrder(Right(T)); }
}

InOrder(T) {
if !is_empty(T) { InOrder(Left(T)); process(Root(T)); InOrder(Right(T)); }
}

PostOrder(T) {
if !is_empty(T) { PostOrder(Left(T)); PostOrder(Right(T)); process(Root(T)); }
}
```

HASHING

For a hash function HF: **0 <= HF(key) <= M-1**

Collision handling:-

Separate chaining:- the collisions are handled by building a separate list from the position given by HF(key)

OR
a new position is calculated using HF(key) + f(i) where i is the i^{th} collision

1. Linear probing f(i) = i
2. Quadratic probing f(i) = i^2
3. Double hashing f(i) = HF_2(key)

Examples of HF:- key mod x (x is usually a prime number)
key mod 10 (used in the examples for teaching purposes since the calculation is easy)

Example of HF_2:- **R - (key mod R)** where R is a prime number < size(Hash Table)

In the lecture notes R = 7 was used (R < 10).

Disadvantages:-

1. Linear probing ➔ primary clusters (around the first position found) ➔ linear search
2. Quadratic probing ➔ secondary clusters (requires a rehash when load > 50%

HEAP

Heapify & Build

```
Heapify(A, i)
```

Determine the left and right children - "l" and "r"

```
    l = Left(i)
    r = Right(i)
```

Determine **if the left child exists** (i.e. that i is **not** a leaf node) and if so whether the value at the left child is greater than than the value at the parent - otherwise largest is the parent node

```
    if l <= A.size and A[l] > A[i] then largest = l else largest = i
```

Determine **if the right child exists** and if so whether the value at the right child is greater than than the value of largest.
Now we have the biggest value for the parent and left and right children

```
    if r <= A.size and A[r] > A[largest] then largest = r
```

If the largest value is **NOT** the parent then swap the parent value with the child value and heapify the child sub-tree

```
    if largest != i then
      swap(A[i], A[largest])
      Heapify(A, largest)
      end if
end Heapify


Build(A)
    for i = [A.size / 2] downto 1 do Heapify(A, i)
    end Build
```

Why does the algorithm start with A.size/2?

See the revision notes for a worked example
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/index.php?heapify=1
See the animation on
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/Heapify.pps

Add

```
Add(H, v)
        let A = H.array
        A.size++
        i = A.size
        while i > 1 and A[Parent(i)] < v
                do      A[i] = A[Parent(i)]
                        i = Parent(i)
          end while
        A[i] = v
end Add
```

See the animation on
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/HAdd.pps

Remove

```
Remove(H)
        let     A = H.array
        A[1] = A[A.size]
        A.size--
        Heapify(A, 1)
end Remove
```

See the animation on
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/HRemove.pps

TREES (2)

AVL-add

```
void AvlTree::insert( const Comparablke & x, AvlNode * & t ) const

{ if (t == NULL)    t = new AvlNode(x, NULL, NULL);

else if ( x < t->element) {    /*** add to left child ***/

        insert(x, t->left);

        if ( height(t->left) - height(t->right) == 2 )
                if ( x < t->left->element ) rotateWithLeftChild(t);  /*LSTof LC*/
                else doubleWithLeftChild(t);                         /*RSTof LC*/
                }

else if ( t->element < x) {    /*** add to right child ***/

        insert(x, t->right);

        if ( height(t->right) - height(t->left) == 2 )
                if (t->right->element < x) rotateWithRightChild(t); /*RSTof RC */
                else doubleWithRightChild(t);                        /*LSTof RC */
                }

else ; /*** duplicate - do nothing ***/

t->height = max( height(t->left), height(t->right)) + 1;   /*** recalculate height ***/

}
```

LST = Left sub tree **LC = Left child**
RST = Right sub tree **RC = Right child**

REMEMBER:

Adding to the **"outside"** requires a **single** rotation
Adding to the **"inside"** requires a **double** rotation

Rotations – remember that these are "mirror images" (right ⬅➡ left)

SINGLE ROTATIONS:-

```
Void rotateWithLeftChild(AvlNode * & k2 ) {     // single right rotation

    AvlNode *k1       = k2->left;

    k2->left          = k1->right;
    k1->right         = k2;
    k2->height        = max(height(k2->left), height(k2->right)) + 1;
    k1->height        = max(height(k1->left), k2->height) + 1;
    k2                = k1;
}

Void rotateWithRightChild(AvlNode * & k2 ) {   // single left rotation

    AvlNode *k1       = k2->right;

    k2->right         = k1->left;
    k1->left          = k2;
    k2->height        = max(height(k2->right), height(k2->left)) + 1;
    k1->height        = max(height(k1->right), k2->height) + 1;
    k2                = k1;
}
```

DOUBLE ROTATIONS:-

```
Void doubleWithLeftChild( AvlNode * & k3 ) {          // double right rotation

        rotateWithRightChild( k3->left );             // single left rotation
        rotateWithLeftChild( k3 );                    // single right rotation
}

Void doubleWithRightChild( AvlNode * & k3 ) {         // double left rotation

        rotateWithLeftChild( k3->right );             // single right rotation
        rotateWithRightChild( k3 );                   // single left rotation
}
```

GRAPHS

Dijkstra

```
Dijkstra ( a )
{
        S = {a}                          -- start node
        for (i in V-S) D[i] = C[a, i]    -- initialise D

        -- D[i] represents the distance from node a to the remaining
        -- nodes in the graph (i.e. the edges. Distance = infinity if
        -- there is no edge from node a to some node x)

        while (!is_empty(V-S)) {         -- "unvisited nodes"
                                         -- i.e. nodes not in the
                                         -- component (S)

                choose w in V-S such that D[w] is a minimum
                -- i.e. D[w] is the shortest path from a to w (so far)

                S = S + {w}
                -- add node w to the component (i.e. visited nodes)

                foreach ( v in V-S ) D[v] = min(D[v], D[w]+C[w,v])*
                -- check if there is a shorter path VIA node w
                -- than that already calculated – if so update D[v]
                }
        }
```

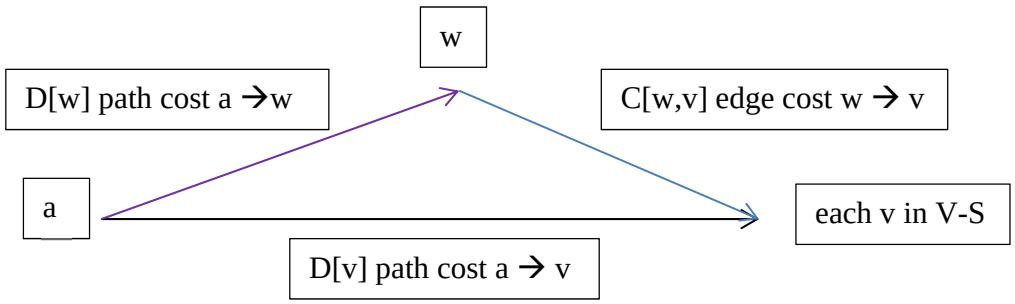
*could also be written

```
foreach ( v in V-S ) if (D[w]+C[w,v] < D[v]) D[v] = D[w]+C[w,v]
```

The "essence" of this algorithm is the row

```
foreach ( v in V-S ) D[v] = min(D[v], D[w]+C[w,v])
```

this may be visualised as



See the worked example in the revision notes
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/DijkstraEx4.pdf
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/DijkstraEx.pdf

Dijkstra + SPT (Shortest Path Tree)

```
Dijkstra_SPT ( a )
{
        S = {a}                 -- start node

        for (i in V-S) {
           D[i] = C[a, i]       --- initialise D - (edge cost)

        -- D[i] represents the distance from node a to the remaining
        -- nodes in the graph (i.e. the edges. Distance = infinity if
        -- there is no edge from node a to some node x)

           E[i] = a             --- initialise E - SPT (edge)

        -- E[i] represents the edge from E[i] to each node in V-a

           L[i] = C[a, i]       --- initialise L - SPT (edge cost)

        -- L[i] represents the edge COST from E[i] to each node in V-a

        -- together E[i] & L[i] represent the SPT in its different
        -- stages of development

           }

        while (!is_empty(V-S)) {

                choose w in V-S such that D[w] is a minimum
                -- i.e. D[w] is the shortest path from a to w (so far)

                S = S + {w}
                -- add node w to the component (i.e. visited nodes)

                foreach ( v in V-S )
                    if (D[w]+C[w,v] < D[v]) {
                        -- check if there is a shorter path VIA node w

                       D[v] = D[w]+C[w,v]       -- update path cost to v
                       E[v] = w                 -- save the edge w → v
                       L[v] = C[w,v]            -- save the COST w → v

                -- i.e. the edge w → v gives a shorter path a → v and
                -- thus should be added to the SPT

                       }
                }
        }
```

See the worked example in revision notes
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/DijkstraSPTEx4.pdf

GRAPHS

Floyd's Algorithm – all pairs shortest path

```
Floyd ( )
{

        for (i in 1..n) for (j in 1..n) if (i <> j) A[i, j] = C[i, j]          -- initialisation
        for (i in 1..n) A[i, i] = 0

        for (k in 1..n) for (i in 1..n) for (j in 1..n)
                if ( A[i, k] + A[k, j] < A[i, j])  A[i, j] = A[i, k] + A[k, j]
        }
```

Warshall's Algorithm – transitive closer (i.e. does a path exist between nodes x and y?)

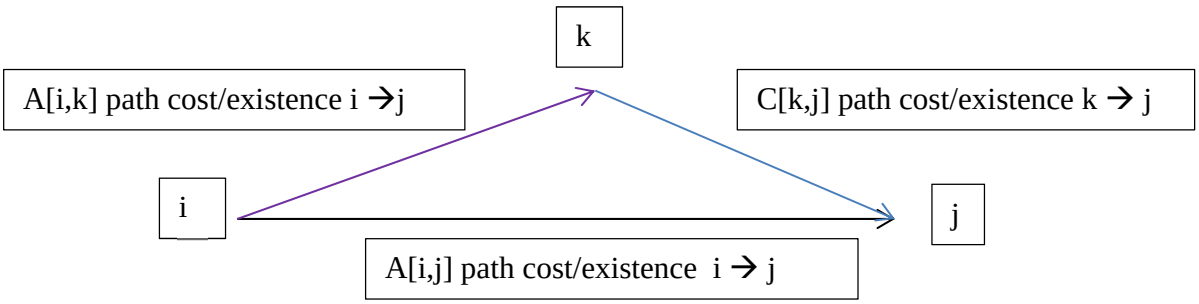
```
Warshall ( )
{

        for (i in 1..n) for (j in 1..n) A[i, j] = C[i, j]          -- initialisation
         for (i in 1..n) A[i, i] = 0

         for (k in 1..n) for (i in 1..n) for (j in 1..n)
                if (A[i, j] = 0) A[i, j] = A[i, k] and A[k, j]
         }
```

Note that both use a similar principle to Dijkstra's algorithm

this may be visualised as



DIGRAPHS (Directed Graphs)

Depth-first search:-

```
select one v in V and mark as visited; enqueue v in Q

        while not is_empty(Q) {

                x = front(Q); dequeue(Q);

                for each y in adjacent (x) if unvisited (y) {
                        mark(y); enqueue y in Q; process (x,y);      // (e.g. add to tree)

                        }
```

Topological Sort:-

```
tsort(v)        {

        mark v visited
                for each w adjacent to v if w unvisited tsort(w);

                display(v);

                }
```

NB: prints **reverse** topological order of a DAG from v

Strong Components Algorithm:-

- Perform a dfs and assign a number to each vertex

```
dfs(v) {    mark v visited
            for each w adjacent to v if w unvisited dfs(w)
            number v
            }
```

- construct digraph G_r by reversing every edge in G
- perform a dfs on G_r starting at highest numbered vertex (repeat on next highest if all vertices not reached)
- each tree in resulting spanning forest is an SCC of G

GRAPHS

Kruskal's

S = set of connected components (V from G=(V,E))

merge(A, B, S) -- merge components A & B in S - rename A
find(v, S) -- return name of component X in S : v in X
initial('A', v, S) -- make A the name of component in S
containing only vertex v initially
insert(e, S) -- add a given edge to S
remove_pq() -- remove an edge from the PQ
(x, y, c) -- edge (x, y) in PQ with cost c

```
for each v in S initial ( next(name), v, S)    -- initiliase

while (size(S) > 1 {                           -- size = number of components
        get_PQ ( );                            -- get (x, y, c) from PQ

        if ( find(x, S) != find(y, S) ) {      -- x, y in different components
                merge ( find (x, S), find (y, S), S );
                insert (get_PQ ( ), S);
                }
        remove_pq( );
        }
```

-- this is a "less obvious" statement of what is a relatively simple algorithm if you look at
-- the explanation in the lecture notes
-- http://www.cs.kau.se/cs/education/courses/dvgb03/lectures/graphs3.pdf
-- and the summary on
-- http://www.cs.kau.se/cs/education/courses/dvgb03/revision/Kruskal.pdf

Prim's

```
Prim ( node v)   -- v is the start node

    { U = {v};  for i in (V-U) { low-cost[i] = C[v,i]; closest[i] = v; }

    -- U is the set of "visited nodes" i.e. the component which will eventually
    -- become the MST
    -- V is the set of nodes in the graph G = (V, E)
    -- V-U is the set of "unvisited nodes" i.e. nodes which are NOT part of the
    -- component

    -- low_cost[i] represents the cost of an edge from x → y in the graph
    -- initially this is the cost from the start node v to the remaining nodes
    -- closest[i] is the value of this edge (initially may be infinite)

    while (!is_empty (V-U) ) {              -- find the closest vertex in V-U

        i = first(V-U); min = low-cost[i]; k = i; -- minimum cost edge
        for j in (V-U-k) if (low-cost[j] < min) {min = low-cost[j]; k = j; }

        -- this computes the closest (least cost) node NOT in the component

        display(k, closest[k]);                 -- display edge (just to check)
        U = U + k;                              -- k added to U i.e. the component

        for j in (V-U) if ( C[k,j] < low-cost[j] ) )     -- readjust costs
                {low-cost[j] = C[k,j]; closest[j] = k; }
        }

        -- if there is an edge from k to any (unvisited) node in V-U (i.e. not in
        -- the component) replace the current edge with the edge k → j
        -- (j in V-U) and update low_cost[j]

    }
```

See the worked example in the revision notes
http://www.cs.kau.se/cs/education/courses/dvgb03/revision/PrimExa.pdf

Articulation Points Algorithm:-

- Perform a **dfs of the graph**, computing the df-number for each vertex v (df-numbers order the vertices as in a pre-order traversal of a tree)
- for each vertex v, compute low(v) - the smallest df-number of v or any vertex w reachable from v by following down 0 or more tree edges to a descendant x of v (x may be v) and then following a back edge (x, w)
- compute **low(v)** for each vertex v by visiting the vertices in post-order traversal
- when v is processed, **low(y)** has already been computed for all children y of v

Note that the post-order traversal implies a bottom-up solution

- the root is an AP iff it has 2 or more children
  - since it has no **cross edges**, removal of the root must disconnect the sub-trees rooted at its children
  - removing a => {b, d, e} and {c, f, g}
- a vertex v **(other than the root)** is an AP iff there is some child w of v such that low(w) >= df-number(v)
  - v disconnects w and its descendants from the rest of the graph
  - if **low(w) < df-number(v)** there must be a way to get from w down the tree and back to a proper ancestor of v (the vertex whose df-number is low(w)) and therefore deletion of v does not disconnect w or its descendants from the rest of the graph