Student ID: .....

TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

EXAMINATIONS – 2013
TRIMESTER 2 *** WITH SOLUTIONS ***

COMP 261
ALGORITHMS
and
DATA STRUCTURES

Time Allowed: THREE HOURS

Instructions: Closed Book

Attempt ALL Questions.

Answer in the appropriate boxes if possible — if you write your answer elsewhere, make it clear where your answer can be found.

The exam will be marked out of 180 marks.

Only silent non-programmable calculators or silent programmable calculators with their memories cleared are permitted in this examination.

Non-electronic foreign language dictionaries are permitted.

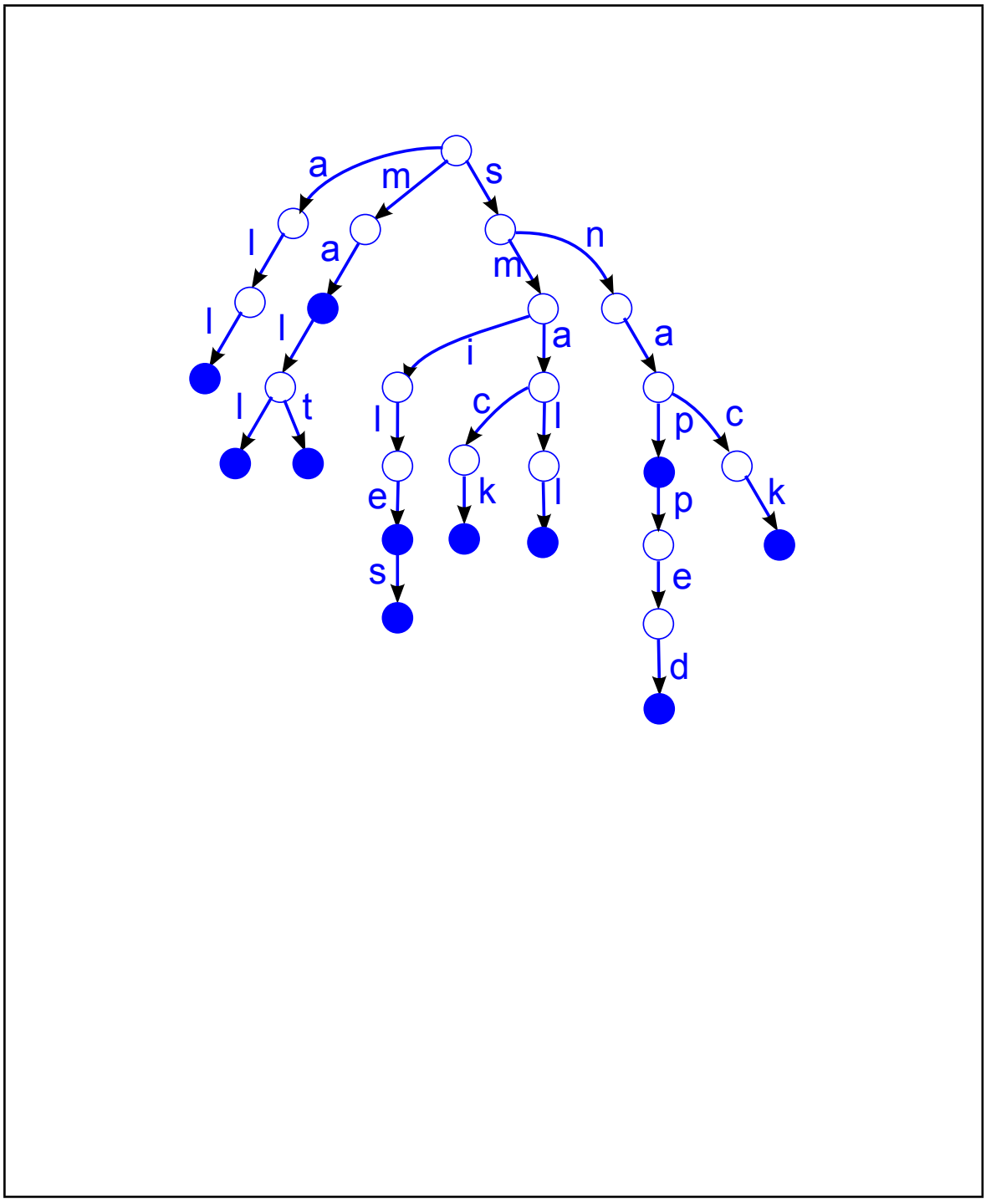
Alphabetic order: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Questions	**Marks**
1. Tries	[10]
2. Articulation Points	[20]
3. Graphics	[30]
4. Graph Algorithms	[34]
5. Parsing	[40]
6. B+ Trees	[26]
7. Maximum Flow	[20]

Question 1. Tries [10 marks]

(a) [5 marks] Draw a Trie containing the following set of strings.
(Note: label the links of the Trie, not the nodes.)

```
small     mall     all      malt     smile     smiles
snapped   snack    ma       smack    snap
```



(Question 1 continued on next page)

(Question 1 continued)

(b) [5 marks] Give an example of a problem where using a Trie would be a more efficient implementation of a set of strings than a HashSet, and explain why the Trie would be more efficient.

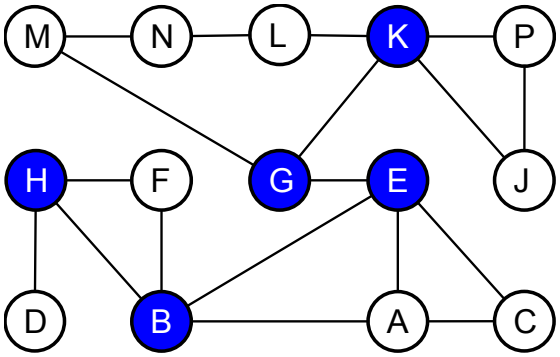
(a) When you need to be able to list all the strings in order (or all the strings between two values, in order). With a trie, you can do a traversal of the trie, starting at the first string. With the HashSet, you would need to extract all the values and sort them.
(b) An autocomplete function that should list all the strings from a dictionary that start with the prefix the user has typed. With a trie, you search down the trie for the prefix, then traverse the subtree under that node listing all the strings. With a Hashset you would need to iterate through every string in the set checking whether hit has the prefix, which would be much slower.
(c) If you need to store a very large number of strings, where there are a lot of shared prefixes (for example, dictionaries or lists of names). The trie may use less memory since it only needs to store any shared prefix once.

Question 2. Articulation Points [20 marks]

(a) [4 marks] Briefly explain the problem that the Articulation Points algorithm solves.

Finds the nodes in an undirected graph whose removal would break a connected component of the graph into two (or more) disconnected components

(b) [4 marks] Circle the articulation points in the following graph.



(c) [4 marks] Outline two examples of applications or problems where the Articulation Points algorithm would be useful.

1: Identifying intersections in a city that are critical for emergency services in that they would cut off access to some roads if they intersection were blocked.
2. Identifying nodes, exchanges, or connection points in a communications network that would disconnect the network if they became disabled.

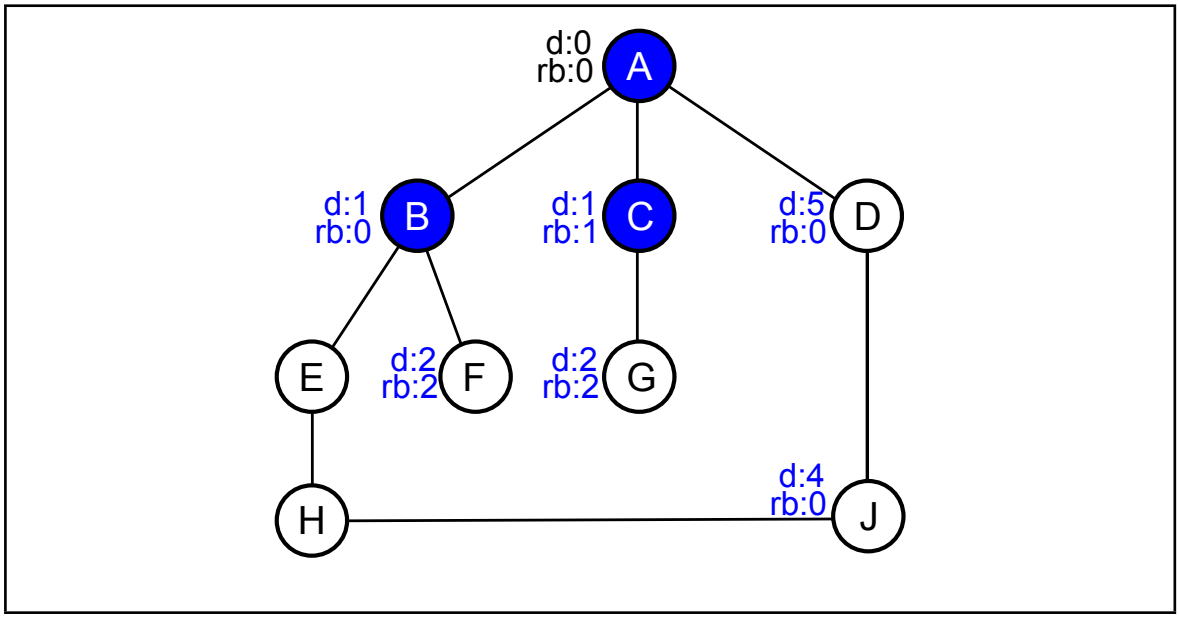
(Question 2 continued on next page)

(Question 2 continued)

(d) [8 marks] Show how the recursive depth-first-search articulation points algorithm would find the articulation points in the following graph, assuming that the search starts with node A and considers neighbours of nodes in alphabetical order. Mark on the graph

- the depth ("d:") and the reach-back ("rb:") of each node (node A is done for you),
- the return values from each recursive call.
- the nodes that are articulation points, indicating why they were marked.

The algorithm is given below for your reference.




```
FindArticulationPoints (graph, start ):
    for each node: node.depth ← ∞, articulationPoints ← { }
    start .depth ← 0, numSubtrees ← 0
    for each neighbour of start
        if neighbour.depth = ∞ then
            RecursiveArtPts( neighbour, 1, start )
            numSubtrees ++
    if numSubtrees > 1 then add start to articulationPoints

RecursiveArtPts(node, depth, fromNode):
    node.depth ← depth, reachBack ← depth,
    for each neighbour of node other than fromNode
        if neighbour.depth < ∞ then
            reachBack ← min(neighbour.depth, reachBack)
        else
            childReach ← recArtPts(neighbour, depth +1, node)
            reachBack ← min(childReach, reachBack )
            if childReach ≥ depth then add node to articulationPoints
    return reachBack
```

Question 3. Graphics [30 marks]

(a) [6 marks] To represent the 3D transformations for the graphics algorithms, we used 4×4 matrices and 4D vectors for the points, even though the points were in 3D space. Explain the advantages of this representation.

With this representation ("homogeneous vectors") (a) all the transformations (*e.g.*, translations, rotations, scaling) could be represented by matrix multiplications, making the code simpler and more consistent; (b) a series of transformations could be composed into a single transformation and applied with one matrix multiplication, which is more efficient than applying each transformation separately.

(b) [6 marks] The 3D rendering algorithm in the lectures constructed edgelists as part of the process of interpolation to compute the depth (z) at each pixel position on the polygon. Explain the role of the edgelists in the algorithm.

The algorithm uses interpolation to work out the z-value of each pixel on the polygon. It splits the interpolation into two phases, first interpolating between the vertices to find the x and z values along the edges at each scan line (y value), then interpolating along the scan line from the left edge to the right edge. The edge lists are the result of the first phase, and specify the "outline" of the polygone; they are used in the interpolation of the second phase.

(Question 3 continued on next page)

(Question 3 continued)

(c) [8 marks] Suppose your Z-buffer rendering program has processed some of the polygons in a model, and is now processing a new polygon. The colour of the new polygon is blue. Part of the edgelists for the polygon are:

	left		right	
y	x	z	x	z
20	11	20	16	30
21	10	17	15	27

The current contents of the Z-buffer are the following. Each cell of the Z-buffer contains a colour and a z value (z increases away from the viewer).

y \ x	10	11	12	13	14	15	16
20	gray ∞	gray ∞	gray ∞	red 27	red 24	red 21	gray ∞
21	green 20	green 20	green 20	gray ∞	yellow 28	yellow 26	yellow 24

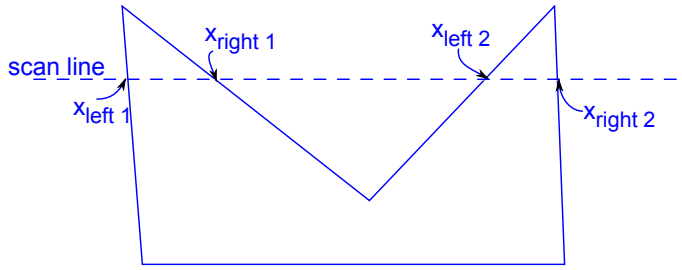
Show the contents of the Z-buffer after your program has rendered the edgelist data into the Z-buffer. State any assumptions you are making.

y \ x	10	11	12	13	14	15	16
20	gray ∞	blue 20	blue 22	blue 24	red 24	red 21	blue 30
21	blue 17	blue 19	green 20	blue 23	blue 25	yellow 26	yellow 24

(Question 3 continued on next page)

(Question 3 continued)

(d) [5 marks] When all the polygons in a 3D model are triangles, the edgelists just contain two x values (left and right) for each value of `y`. Explain, using the following diagram, why the edgelists have to hold more values if the polygon can have more than three sides.



An edge list has to have the left and right edges of a horizontal line across the polygon (a particular scan line (value of y)).
If the polygon has more than three sides, it might not be convex, *e.g.* the v shape at the top of the figure above. In this case, a horizontal line might cross the polygon in two (or more) separate places. If it only uses the left-most and rightmost edge, the rendering will fill in the "gap"; it needs more values in the edge list to specify the start and end of each part of the polygon that it crosses so that it only renders points inside the polygon.

(Question 3 continued on next page)

(Question 3 continued)

(e) [5 marks] The algorithm for rendering 3D models presented in the lectures produced images showing the individual polygons as flat faces with sharp edges. Explain what would be required to render the model as a smoothly rounded surface.

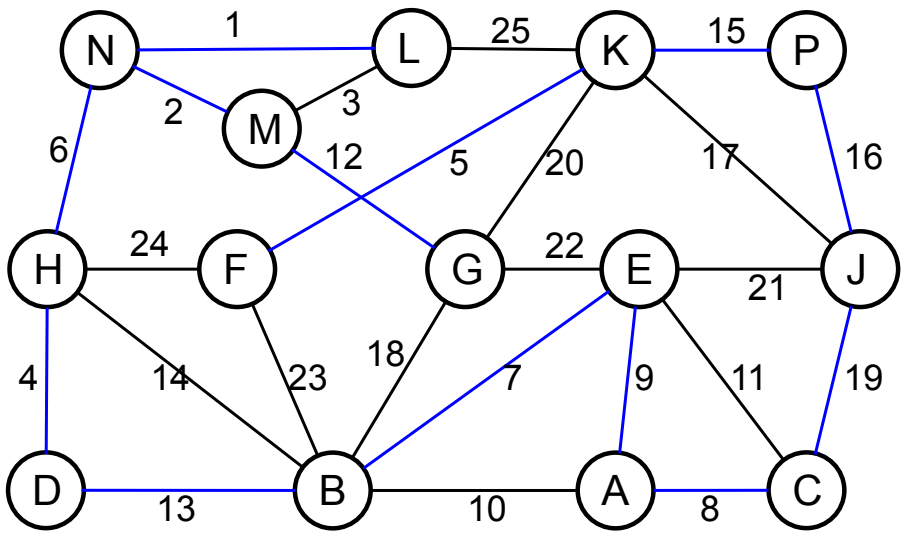
Instead of using a single color value for all points on the polygon, the color value would need to be interpolated, *e.g.*, by interpolating between color values at the vertices that are an average of the colors of the faxes that meet at the vertex. Even better, interpolate the surface normal between surface normals at the vertices, and compute the color at each point using the interpolated normal.

Question 4. Graph Algorithms [34 marks]

(a) [8 marks] Show how Kruskal's Minimum Spanning Tree algorithm will find a minimum spanning tree for the following graph by marking the edges that the algorithm will add to the spanning tree, and listing the edges in the order they are added.

Note: All the edges have different weights, from 1 to 25. You can list the edges by giving their weights.

Reminder: Kruskal's MST algorithm builds up sets of connected nodes, in contrast to Prim's algorithm which builds outwards from a starting node.



Edges added: 1(LN) 2(MN) 4(DH) 5(KL) 6(HN)
7(BE) 8(AC) 9(AE) 12(GM) 13(BD) 15(KP) 16(JP) 19(CJ)

(Question 4 continued on next page)

(Question 4 continued)

(b) [12 marks] Write pseudocode for Kruskal's Minimum Spanning Tree algorithm.

Kruskal's MST algorithm:
given a weighted, undirected graph.
Put all the edges into a priority queue, priority given by weight.
Construct a collection of sets, each node being in its own set
numEdges = 0;
WHILE numEdges ¡ (number of nodes in graph - 1):
. dequeue edge (x, y) from priority queue
. IF x and y are not in the same set THEN
. mark the edge as being part of the spanning tree
. join the sets containing x and y
. increment numEdges

(Question 4 continued on next page)

(Question 4 continued)

(c) [8 marks] Explain why the union-find data structure is important for the efficiency of Kruskal's algorithm.

The three key operations in Kruskal's algorithm are
- dequeuing from the priority queue ($O(\log(e))$)
- determining whether two nodes are in the same set, and
- joining two sets into a single set.

If the last two operations are more expensive than dequeuing, then they will dominate the cost of the algorithm. The union-find data structure allows those two operations to have a small constant cost on average, so that the algorithm will have a total cost of $O(n \log(e))$. Most other data structures for the sets of nodes would have an O(e) cost for at least one of the operations, making the total cost of the algorithm $O(ne)$, which is much slower.

(Question 4 continued on next page)

(Question 4 continued)

(d) [6 marks] Explain how the use of a heuristic can make A* search more efficient than Dijkstra's algorithm for finding the shortest path between two nodes in a graph, and discuss the properties that the heuristic should have.

- uses heuristic to estimate the total cost of the best path through a node, rather than just the cost from the start to the node.
- That means it can choose nodes that are on more promising paths.
- It can also cut off bad paths earlier
- The result is that A* will explore many fewer nodes than Dijkstra's,

- heuristic must be optimistic - underestimate the remaining cost, so that it doesn't cut off a node when it is actually a good path.
- heuristic should be consistent so that the algorithm doesn't have to revisit nodes, because the first time it visits a node, will always be the best path to the node.

Question 5. Parsing [40 marks]

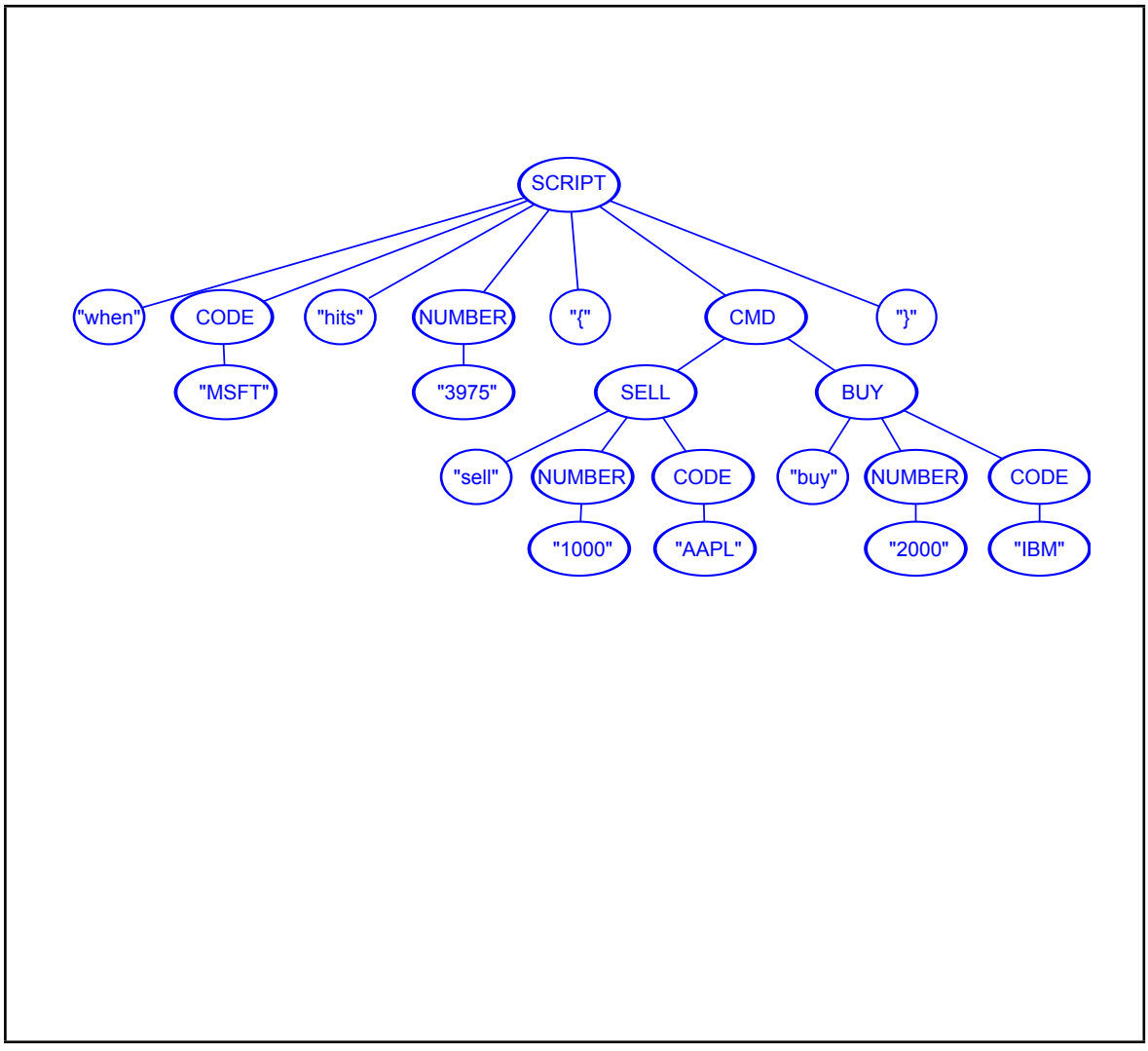
Consider the following grammar for part of a simple scripting language for a stock trading system :

```
SCRIPT ::= "when" CODE "hits" NUMBER "{" [ CMD ]+ "}"
CMD ::=  SELL  |  BUY  |  SCRIPT
SELL ::= "sell" NUMBER CODE
BUY ::= "buy" NUMBER CODE
CODE ::=  three or four alphabetic characters
NUMBER ::=  a positive integer
```

Note: `[ CMD ]+` means one or more repetitions of CMD.

(a) [7 marks] Give a concrete syntax tree for the following script

```
when MSFT hits 3975 { sell 1000 AAPL buy 2000 IBM }
```

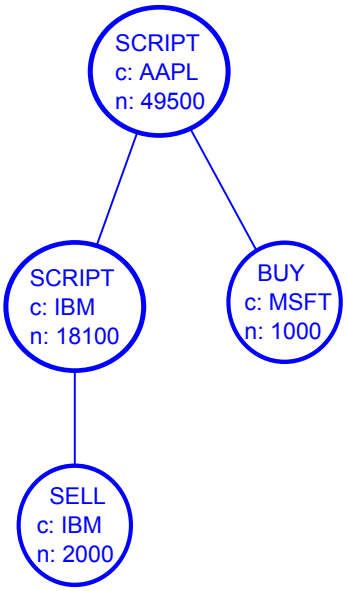


(Question 5 continued on next page)

(Question 5 continued)

(b) [7 marks] Give an abstract syntax tree for the following (different) script, and give a brief justification for what you included and what you ignored in the abstract syntax tree.

```
when AAPL hits 49500 { when IBM hits 18100 { sell 2000 IBM }
                       buy 1000 MSFT }
```



Justification: Only the SCRIPT, BUY, and SELL nodes had any information attached to them. There is no information attached to the CMD node, therefore, it was dropped and the BUY and SELL nodes added directly to the SCRIPT node. The SCRIPT node includes a sequence of commands. The string literals occurring in the rules are only for helping the parser, and all the information in them is implicit in the nodes.
Note that there are other possible trees that would also be acceptable - for example, you could have more of the non-terminal nodes in the AST.

(Question 5 continued on next page)

(Question 5 continued)

(c) [20 marks] Suppose you are writing a recursive descent parser for the script language and you have definitions of the classes, patterns, and methods given below.

Complete the three methods, parseCmd, parseBuy and parseScript on the next two pages. All three have a Scanner parameter, and all three should return an appropriate Node. If the Scanner does not contain a valid CMD, BUY, or SCRIPT, the methods should throw an exception using the fail method. (Explanatory error messages are not necessary.)

Assume that the Scanner has an appropriate delimiter for separating the tokens.

Your methods may (if you wish) call the gobble, fail, and parseSell methods given below.

```
interface Node {}

class ScriptNode implements Node{
    private String code;
    private int price;
    private List<Node> cmds;
    public ScriptNode(String cd, int pr, List<Node> cs){ ... }
}
class BuyNode implements Node{
    private String code;
    private int count;
    public BuyNode(String cd, int ct){code=cd;count=ct;}
}
class SellNode implements Node{
    private String code;
    private int count;
    public SellNode(String cd, int ct){code=cd;count=ct;}
}

private String CodePat = "[A-Z][A-Z][A-Z][A-Z]?";
private String NumPat = "[1-9][0-9]+";
private String OpenPat = "\\{";
private String ClosePat = "\\}";

private boolean gobble(String pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true;}
    return false;
}
private void fail (Scanner s){
   throw new RuntimeException("Parser error at " + s.next());
}
// Parse Methods
private Node parseSell(Scanner s){ ... }  //parses a Sell command
private Node parseBuy(Scanner s){ ... }   //complete on facing page
private Node parseCmd(Scanner s){ ... }   //complete on facing page
private Node parseScript(Scanner s){ ... } //complete on following page.
```

(Question 5 continued on next page)

(Question 5 continued)

The grammar (repeated):

```
SCRIPT ::= "when" CODE "hits" NUMBER "{" [ CMD ]+ "}"
CMD ::=  SELL  |  BUY  |  SCRIPT
SELL ::= "sell" NUMBER CODE
BUY ::= "buy" NUMBER CODE
CODE ::=  three or four alphabetic characters
NUMBER ::=  a positive integer
```

```
private Node parseBuy(Scanner s){
    if (!gobble("buy", s))     { fail (s);}   //Expecting a buy
    if (!s.hasNext(NumPat))  { fail (s);}   //Expecting count in buy
    int num = s.nextInt();
    if (!s.hasNext(CodePat)) { fail (s);}   //Expecting code in buy
    String code = s.next();
    return new BuyNode(code, num);









}
private Node parseCmd(Scanner s){
    if (s.hasNext("buy")) { return parseBuy(s); }
    if (s.hasNext("sell")) { return parseSell(s); }
    if (s.hasNext("when")) { return parseScript(s);}
    fail (s); return null;









}


// Complete parseScript on the next page.
```

(Question 5 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 5 continued)

The grammar (repeated):

```
SCRIPT ::= "when" CODE "hits" NUMBER "{" [ CMD ]+ "}"
CMD ::=  SELL  |  BUY  |  SCRIPT
SELL ::= "sell" NUMBER CODE
BUY ::= "buy" NUMBER CODE
CODE ::=  three or four alphabetic characters
NUMBER ::=  a positive integer
```

```
private Node parseScript(Scanner s){
    if (!gobble("when", s))     { fail (s); }  //Expecting a when
    if (!s.hasNext(CodePat))    { fail (s); }  //Expecting code
    String code =s.next();
    if (!gobble("hits", s))     { fail (s); }  //Missing hits in when
    if (!s.hasNext(NumPat))     { fail (s); }  //Expecting count
    int count = s.nextInt ();
    if (!gobble(OpenPat, s))    { fail (s); }  //Missing { in when
    List<Node> cmds = new ArrayList<Node>();
    cmds.add(parseCmd(s));
    do{
        cmds.add(parseCmd(s));
    }while (!s.hasNext(ClosePat));
    if (!gobble(ClosePat, s))   { fail (s); }  //Missing } in when
    return new ScriptNode(code, count, cmds);
}
```

(Question 5 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 5 continued)

(d) [6 marks] Suppose we wanted to be able to specify that commands must be performed concurrently (specified by ``and'') or one after the other (specified by ``then''). For example, we might want to write:

```
when MSFT hits 3975 { sell 1000 AAPL then sell 2000 IBM
                      and buy 500 HPQ and buy 1000 DELL }
```

A candidate grammar to allow this might be the following:

```
SCRIPT ::= "when" CODE "hits" NUMBER "{" CMDS "}"

CMDS   ::=  CMD  | CMD  "and" CMDS | CMDS "then" CMDS

CMD    ::=  SELL  |  BUY  |  SCRIPT
SELL   ::= "sell" NUMBER CODE
BUY    ::= "buy" NUMBER CODE
CODE   ::=  three or four alphabetic characters
NUMBER ::=  a positive integer
```

Explain why it would now be problematic to construct a recursive descent parser for this grammar, and suggest how you might start addressing the problem(s).

The three alternatives for `CMDS` all start the same way, so you cannot tell which branch to take. Furthermore, CMDS is left recursive, so that we cannot tell whether the first CMD belongs to the current `CMDS`, or a nested sub `CMDS`. What's more, the grammar is ambiguous as to how "and"s and "then"s should be nested - in the example above, is it just selling IBM that follows selling AAPL, or do we also buy HPQ and DELL after selling AAPL?
The grammar needs to be rewritten to show the precedence of "and" and "then", and also needs to be rewritten to remove the ambiguity of which branch of the the CMDS rule should be chosen. It might be better to insist on using brackets around concurrent commands. It would be easier to put the ``and'' and ``then'' in front of the commands.

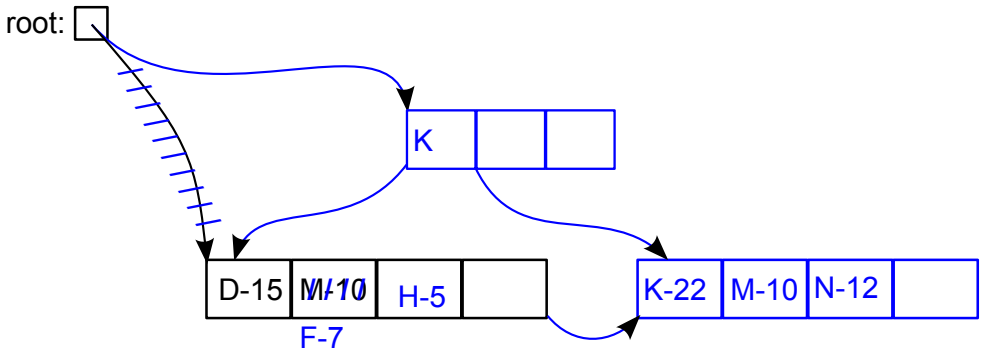
Question 6. B+ Trees [26 marks]

The following subquestions concern a B+ tree that has internal nodes holding up to 3 keys, and leaf nodes holding up to 4 key-value pairs. The keys are letters; the values are numbers.

(a) [5 marks] Show how the B+ tree below would be changed if the following key-value pairs were added to it:

```
H-5  F-7 K-22 N-12
```

State any assumptions you are making.



Assumptions (if any): When the leaf is too full (5 k-v pairs), it could be split 2-3 or 3-2; Here, we have split it 3-2.

Alphabet: A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

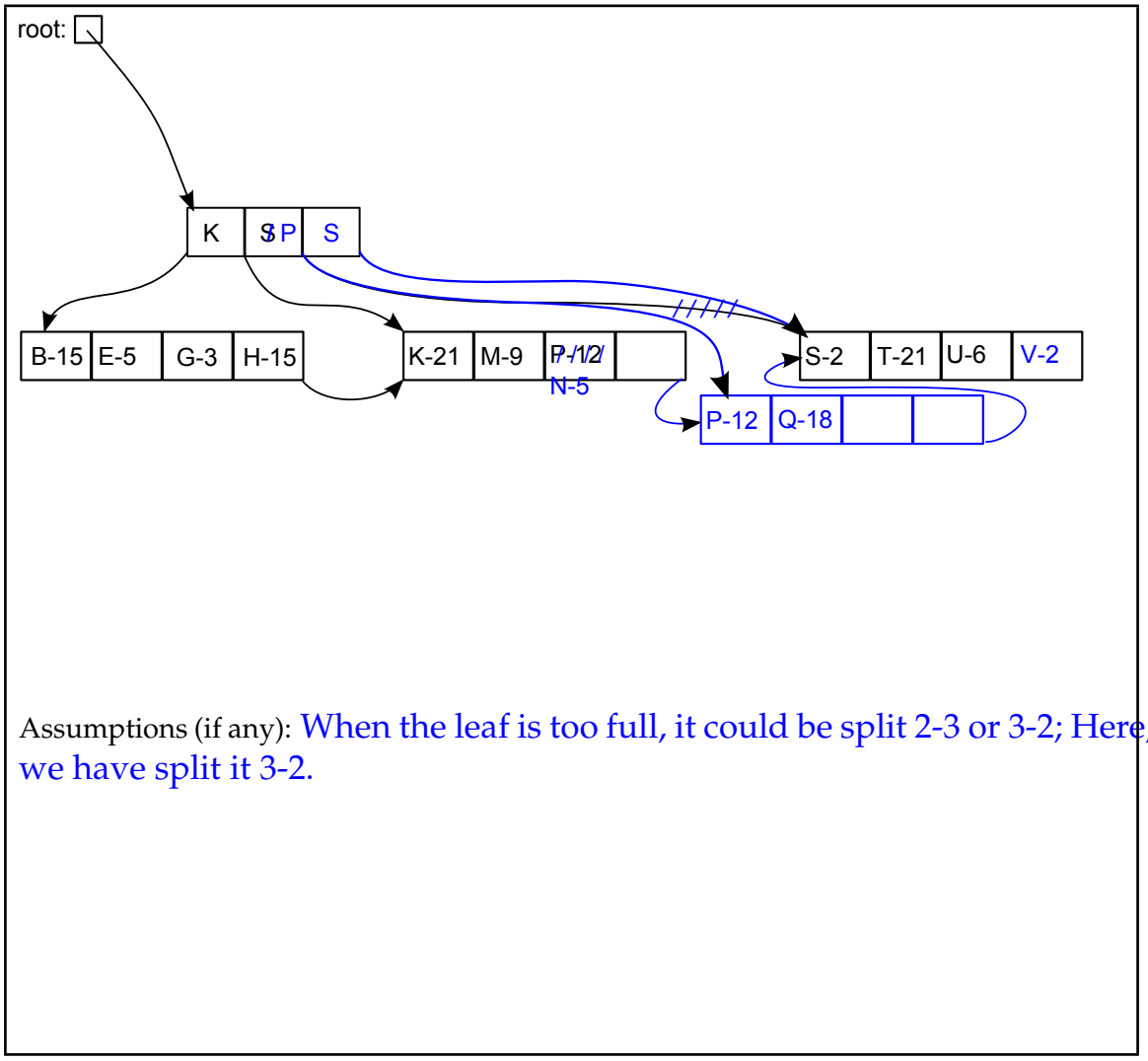
(Question 6 continued on next page)

(Question 6 continued)

(b) [5 marks] Show how the B+ tree below would be changed if the following key-value pairs were added to it:

```
N-5  Q-18 V-2
```

State any assumptions you are making.



Assumptions (if any): When the leaf is too full, it could be split 2-3 or 3-2; Here, we have split it 3-2.

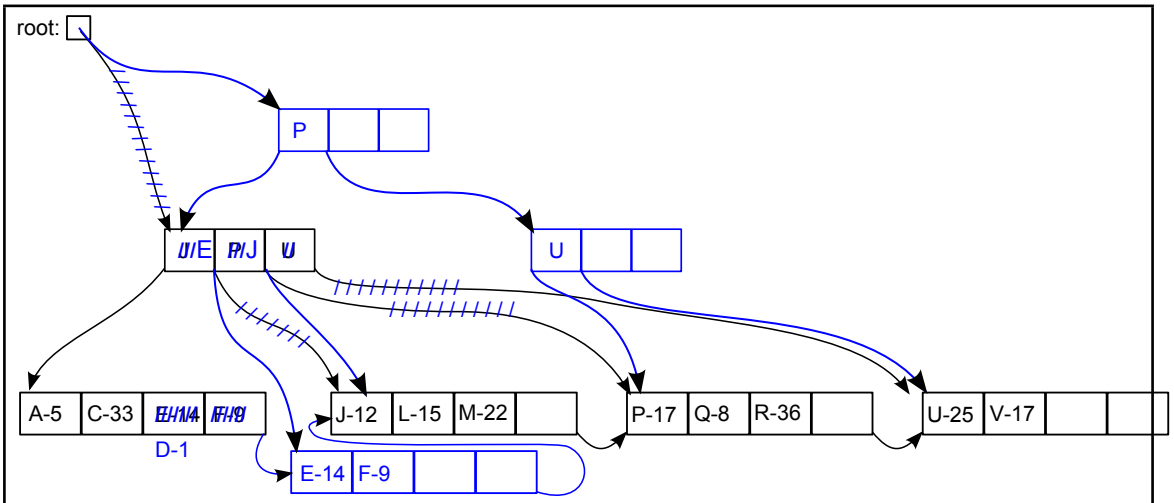
(Question 6 continued on next page)

(Question 6 continued)

(c) [5 marks] Show how the B+ tree below would be changed if the following key-value pair were added to it:

```
D-1
```

State any assumptions you are making.



Assumptions (if any): When the leaf is too full, it could be split 2-3 or 3-2; Here, we have split it 3-2. When the internal node is too full (4 keys, 5 children), it could be split 1-1-2 (the 2nd key being promoted) or 2-1-1 (the third key being promoted). Here, we have split it 2-1-1.

(Question 6 continued on next page)

(Question 6 continued)

(d) [5 marks] Suppose the internal nodes of a B+ tree have at most 6 children (*i.e.*, 5 keys), and the leaves contain at most 4 key-value pairs. The maximum number of key value pairs that can be in a tree is 4×6^h if the height of the tree is h. What is the minimum number of key-value pairs that can be in tree of height h? Assume that $h \geq 1$. Show your working!
Note: the height of a tree is the number of edges (= number of internal nodes) on a path from the root to a leaf.

Internal nodes have at least 3 children, except the root which may have only 2 children.

level:	Minimum # leaf nodes
1:	2
2:	2×3
3:	2×3^2
⋮	⋮
h (leaves):	$2 \times 3^{h-1}$

Each leaf must have at least $4/2 = 2$ key-value pairs
Therefore:
Minimum number of K-V pairs: $2 \times 2 \times 3^{h-1} = 4 \times 3^{h-1}$

(Question 6 continued on next page)

SPARE PAGE FOR EXTRA ANSWERS

Cross out rough working that you do not want marked.
Specify the question number for work that you do want marked.

(Question 6 continued)

(e) [6 marks] Suppose you are implementing a B+ tree in a file, storing each node of the tree in one block. The keys of the B+ tree are share market codes (for example "AAPL"), and the values are company names. Each share market code is up to 6 characters long, and the company names are limited to 66 characters. The blocks are 512 bytes long.

How many key-value pairs can be stored in a leaf node? Explain your reasoning, show your working, and state any assumptions you make about the information stored in the blocks.

6 or 7 key-value pairs in node, depending on bytes for number of pairs.

Leaf needs

- 1 byte to store block type, (or even just one bit!)
- 1, 2, or 4 bytes to store the number of pairs, (or even as few as 3 bits)
- 4 bytes for the index of the next block, and

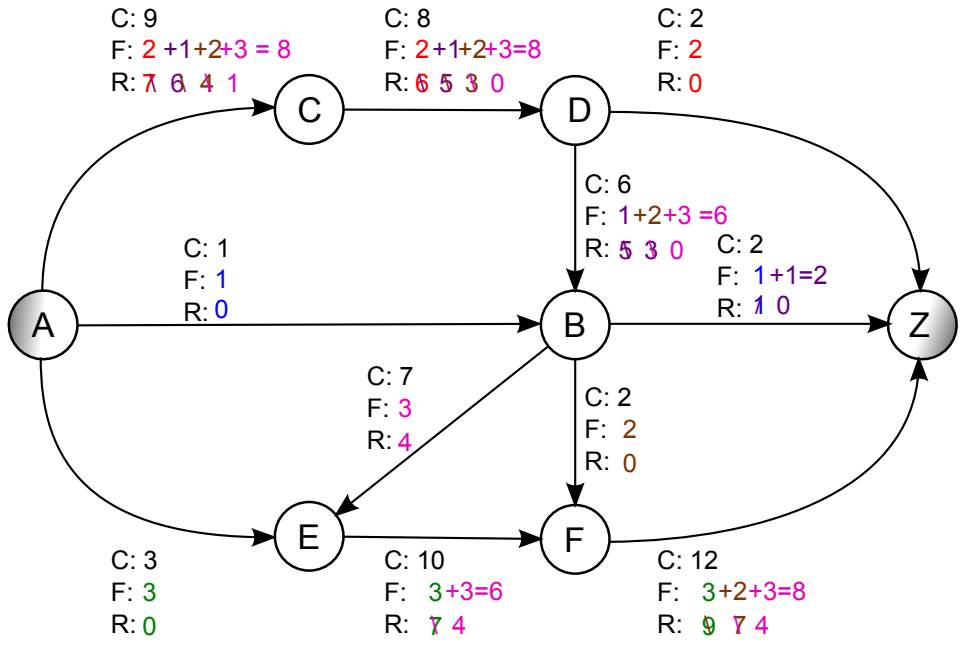
leaving 506, 505, or 503 bytes for the K-V pairs. 72 bytes for each pair means
$\lfloor 506/72 \rfloor = 7$ K-V pairs, if 1 byte for number of pairs
$\lfloor 505/72 \rfloor = 7$ K-V pairs, if 2 bytes for number of pairs
$\lfloor 503/72 \rfloor = 6$ K-V pairs, if 4 bytes for number of pairs

Question 7. Maximum Flow [20 marks]

(a) [13 marks] Show how the Edmonds-Karp algorithm for Maximum Flow would find the maximum flow through the following graph from the node A to the node Z. Each edge is labeled with its capacity, its flow, and its remaining capacity.

1. Show how the flow and remaining capacity change during the algorithm.
2. Below the graph, show the path found in each iteration, along with the flow that can be added along that path.
3. Show the maximum flow from A to Z found by the algorithm.

Hint: Remember that Edmonds-Karp repeatedly uses breadth first search to find a path from source to sink in which every edge has non-zero remaining capacity, sets the new flow to the minimum remaining capacity along the path, and adds this flow to the flow on each edge of the path (and subtracts from each of the reverse edges).



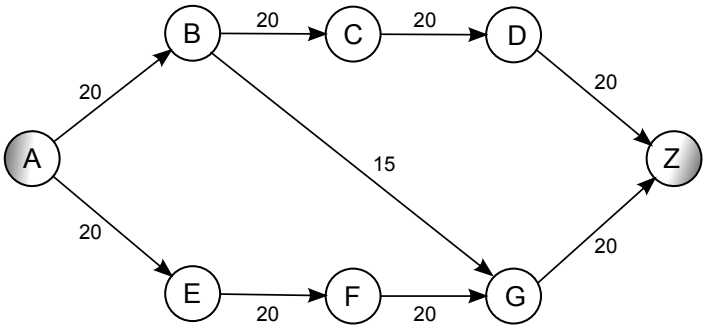
Path	Flow added along path	
A B Z	1	(blue changes)
A C D Z	2	(red changes)
A E F Z	3	(green changes)
A C D B Z	1	(purple changes)
A C D B F Z	2	(brown changes)
A C D B E F Z	3	(pink changes)

Maximum Flow = = 12

(Question 7 continued on next page)

(Question 7 continued)

(b) [7 marks] The graph in part (a) did not need to use any "phantom edges" (though it was not wrong to use them). This is not always the case. Using the following graph as an example, explain why the Edmonds-Karp algorithm needs "phantom edges", and explain how they are used in the algorithm.



The first path that the algorithm chooses includes the edge B-F with a flow of 15, but this will reduce the remaining capacities on the top and bottom paths to just 5 each, giving a total flow of 25. But if it hadn't chosen B-F, it could have had a total flow of 40. To undo the flow along B-F, it needs to put a flow of 15 along the phantom edge from F to B, giving the missing flow of 15.
The phantom edges allow the algorithm to "undo" mistakes it made earlier by pushing flow back along an edge.

****