University of Waterloo
CS240, Spring 2014
Assignment 5 - Update 1

Due Date: Wednesday, July 30, at 9:15am

Update 1: We have added Problems 4 to 10 to cover range queries (module 7), text algorithms (module 8) and compression (module 9). All questions are written problems except problem 4.a) which is is a programming problem; submit your solution to 4.a) electronically as a file named `kdpartition.cpp`.

Please read `http://www.student.cs.uwaterloo.ca/~cs240/s14/guidelines.pdf` for guidelines on submission. Submit your solutions to written problems electronically as a PDF file with name `a05wp.pdf` using MarkUs. We will also accept individual question files named `a05q1w.pdf`, `a05q2w.pdf`, ...

Problem 1 Hashing [3+3=6 marks]

Consider a hash table dictionary with universe $U = \{0, 1, 2, \ldots, 24\}$ and size $M = 5$. If items with keys $k = 21, 3, 16, 1$ are inserted in that order, draw the resulting hash table if we resolve collisions using:

- Linear probing with $h(k) = (k + 1)\ mod\ 5$

  - represents empty

0	1
1	-
2	21
3	16
4	3

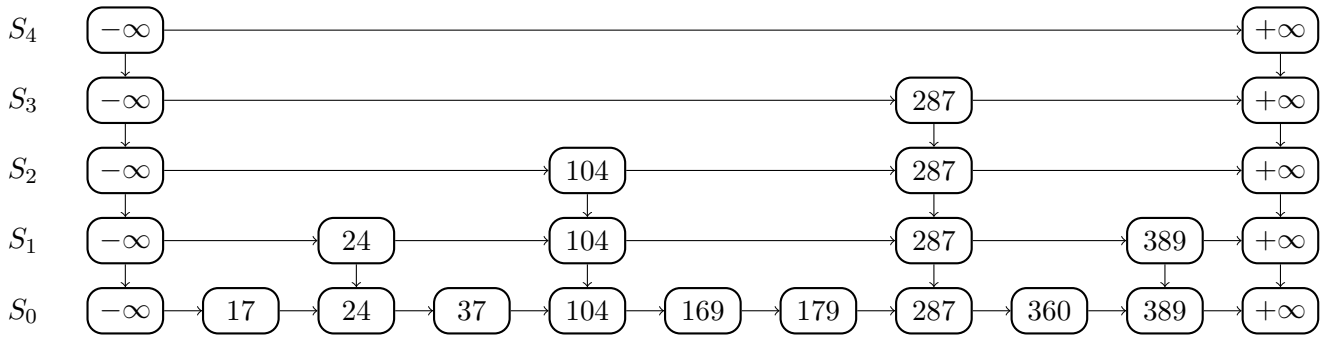
- Cuckoo hashing with $h_1(k) = k\ mod\ 5$ and $h_2(k) = \lfloor k/5 \rfloor$

  - represents empty

0	3
1	1
2	-
3	16
4	21

Problem 2 Skip Lists [6+6+8=20 marks]

a) Consider the skip-list S shown above. Show how $Search(S, 360)$ and $Search(S, 17)$ proceeds. More specifically show at which order search visits the nodes of the skip list.



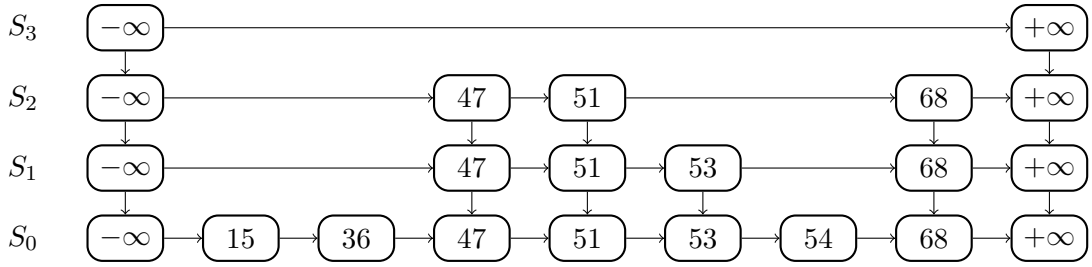
Give only the successful search path, that is only the nodes that result into a 'go right' or 'go down'. You should refer to the nodes using their keys and levels, e.g., you can say node 104 at level 1. The lowest level is 0.

(a) Search(360):
$-\infty$ at level 4, go down
$-\infty$ at level 3, go right
287 at level 3, go down
287 at level 2, go down
287 at level 1, go down
287 at level 0, go right
360 at level 0

(b) Search(17):
$-\infty$ at level 4, go down
$-\infty$ at level 3, go down
$-\infty$ at level 2, go down
$-\infty$ at level 1, go down
$-\infty$ at level 0, go right
17 at level 0

b) Starting with an empty skip list, insert the seven keys $54, 15, 51, 53, 47, 68, 36$. Draw your final answer as on the figure in 4(a). Use the following coin tosses to determine the heights of towers (note, not every toss is necessarily used):

$$T, T, H, H, T, H, T, H, H, T, H, H, T, T, H, T, H, H, T, T, H, H, H, T, \ldots$$



c) The worst case time for searching in a singly linked list is $\Theta(n)$. Now consider a variation of a skip list which has fixed height $h = 3$ even though n can become arbitrarily large. Level S_0 contains the keys $-\infty, k_1, k_2, \ldots, k_n, \infty$. Level S_3 contains only $-\infty$ and ∞. Describe subsets of keys that should be included in levels S_1 and S_2 so that searching in the skip list has worst case cost $\Theta(n^{1/3})$.

Put every multiple of $k_{n^{2/3}}$ from k_0 to k_n on S_2, totalling $n^{1/3} + 1$ of these multiples at this level. Put every multiple of $k_{n^{1/3}}$ from k_0 to k_n on S_1, there would be $n^{2/3} + 1$ of these multiples numbers at this level. No matter what number we search for, in the worst case the cost is $\Theta(n^{1/3})$.

So if n was 64, put 0,16,32,48,64 on S_2 and 0,4,8,16..64 on S_1.

In the worst case, we can only every perform $n^{1/3}$ - 1 shifts on S_2, then at max after moving down from S_2 we can perform $n^{1/3}$ - 1 shifts on S_1, and from S_1 you could then similarly at max perform $n^{1/3}$ - 1 shifts before finding the correct number or failing the search.

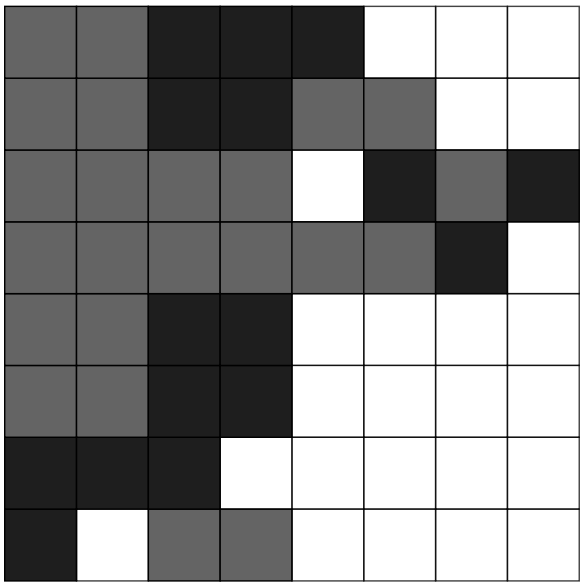
This totals $3n^{1/3}$ + c shifts, where c is a constant, which is $\Theta(n^{1/3})$ complexity.

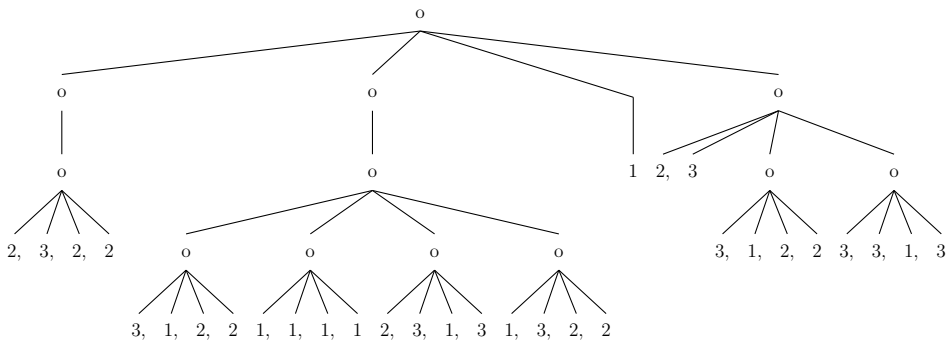
Problem 3 Quad Trees [5+5=10 marks]

For both parts of this question, use the convention that each internal node of a quad tree has exactly four children, corresponding to regions NW, NE, SW and SE, in that order.

a) One of the applications of quad trees is for image compression. An image (picture) is recursively divided into quadrants until the entire quadrant is only one color. Using

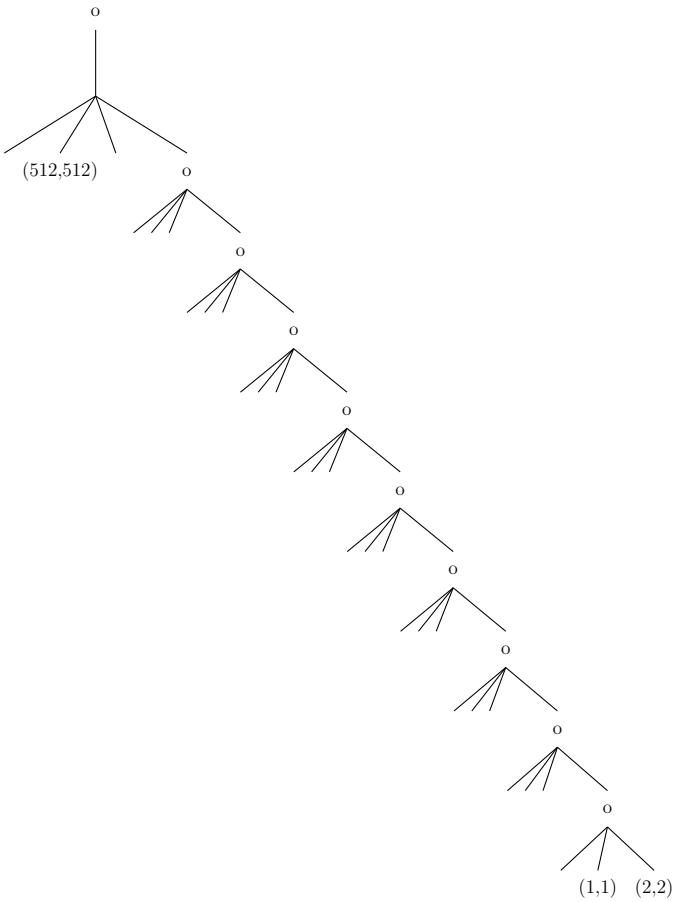
this rule, draw the quad tree of the following image. There are only three colors (shades of gray). For the leaves of the quad tree, use 1 to denote the lightest shade, 2 for the middle shade and 3 for the darkest shade of gray.





b) Give three 2-dimensional points such that the corresponding quad tree has height exactly 9. Give the (x,y) coordinates of the three points and show the quad tree. (Do not give the plane partition.)

points: (1,1), (2,2), (512,512)



Problem 4 *kd*-Tree Construction [15+5=20 marks]

a) Implement an $O(n \log n)$ algorithm to construct a *kd*-tree for dimension 2. Your algorithm should read $2n + 1$ integers from standard input, separated by white space or carriage return. The first integer is the number of points. The remaining $2n$ integers are the n points themselves, according to their x and y coordinates.

Use the recipe on Slide 13 of Module 7 with the following modification on the split: If the array $Px[0..n-1]$ for $n > 1$ is storing points sorted in increasing order according to their x-coordinate, then the vertical splitting line goes through point $Px[mid]$, where $mid = \lfloor n/2 \rfloor$. The root node contains $Px[mid]$, the region to the left of the splitting line include the points $Px[0..mid-1]$, and the region the the right of the splitting line includes the points $Px[mid+1..n-1]$.

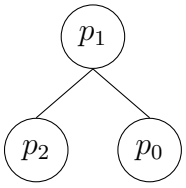
As explained in class, the idea of the algorithm is to first do some preprocessing: sort the points on both their x and y coordinates. For this preprocessing step you may use a standard library function; assume that the sorting algorithm runs in time $O(n \log n)$. You may also need some additional preprocessing. Then call a recursive function (which you create) to produce a tree like the one of Slide 14 of Module 7.

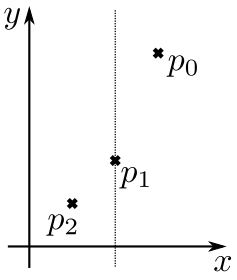
Actually, your program does not need to construct the tree, but rather should just print to standard output the n points stored in the nodes of the tree in the order they are visited during an in-order traversal. The coordinates of the point must be separated by a white space and we want one point per line.

Here is an example input:

```
3
3 4
2 2
1 1
```

The points are $p_0, p_1, p_2 = (3,4), (2,2), (1,1)$. These three points correspond to the following kd-tree:





Thus, for this example your algorithm should print out:

```
1 1
2 2
3 4
```

Your program **must** run in time $O(n \log n)$.

To avoid all ambiguity in the construction of the k-d tree, you can assume that there will only be one point on each separating line. Therefore the choice of the median will always be unique, unlike the example of Slide 14 of Module 7 where p_5 and p_6 are on the same separating line. We will test your program only on such inputs.

b) Justify the $O(n \log n)$ running time of your algorithm.

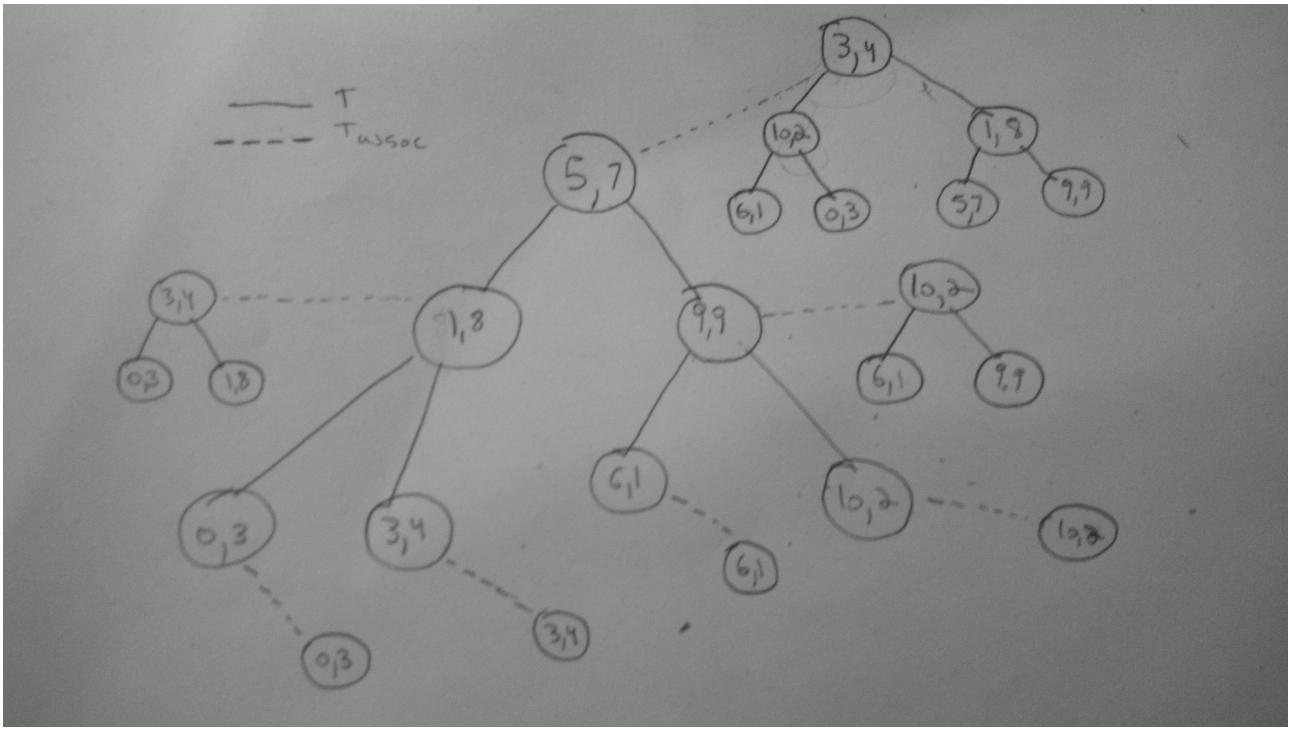
One way to do this is to show that, after a preprocessing that involves sorting, you have a recursive algorithm whose running time on a sub-problem with k points satisfies the following recurrence:

$$T(k) \leq T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil - 1) + O(k).$$

In other words, if you are splitting the region with a vertical line according to x-coordinate, explain how you produce the two sets of points in each region sorted according to their y-coordinate in time $O(k)$.

Problem 5 Range tree [5+5+5=15 marks]

a) Draw the unique 2-dimensional range tree corresponding to points $[3,4]$, $[10,2]$, $[9,9]$, $[5,7]$, $[6,1]$, $[0,3]$, $[1,8]$. More specifically, we want you to draw all the trees τ and $\tau_{assoc}(p)$ for every point p.



b) Assume that we have a set of n numbers (not necessarily integers) and we are interested only in the number of points that lie in a range rather than in reporting all of them. Describe how a 1-dimensional range tree (i.e., a balanced binary search tree) can be modified such that a range counting query can be performed in $O(\log n)$ time (independent of k).

Modify the balanced binary tree T by introducing a counter on each node n that stores the number of nodes beneath the subtree of T with n as the root. When a new value is inserted into the tree, increase the counter on each of its ancestor nodes by one. This preprocessing of the tree allows a range counting query to be performed in $O(\log n)$ time.

Now when a range query is performed, execute a BST-search on the left and right subtrees of the root in order to determine the boundary paths. Add 1 to the total of the range counting query for each boundary node along P1 and P2, which are the left and right paths traversed from the root by the query. For each boundary node visited,

additionally increment the total by the counter stored on the top internal node of the boundary node. This way, rather than having to traverse the inside nodes reporting their values, the top inside node of each boundary node already knows the correct value to return, since every inside node should increment the total.

This modified procedure would then be a constant time operation to increment the count for each boundary node and then for its top internal node, which would run in the worst case the height of the tree for each path:

$$\begin{aligned} & 2\sum_{i=1}^{\log n} O(1) \\ & = 2(O(\log n)) \\ & = O(\log n) \end{aligned}$$

Therefore the modified range counting query can be performed in $O(\log n)$ time.

c) Now consider the 2-dimensional-case: We have a set of n 2-dimensional points. Given a query rectangle R, we want to find the number of points that lie in R. Preprocess the n points (by building an appropriate data structure) such that you can answer any of these counting queries in time $O((\log n)^2)$.

The solution to this problem uses a 2-dimensional range tree T of points (x,y) sorted by x-value, where T has an associated tree $\tau_{assoc}(p)$ sorted by y-value for every point p. Use the same preprocessing algorithm as in b) to store the number of nodes beneath p as a field on p. Perform a BST search based on the x-coordinates to determine the two boundary paths P1 and P2. Every time a boundary node m is visited, check the boundary node itself and if its y-coordinate also satisfies the range, increment the count by 1. Then perform a search just as in b) on the y-coordinates of each top inside node m's $\tau_{assoc}(m)$. The count returned by this operation is the number of inside nodes with a y-coordinate that falls within the range we are searching for, and since we already know that their x-coordinate is within the range, the returned count is added to the total.

This operation would be $O(\log n)$ time to create the boundary paths based on x-coordinate, then for each boundary node's top inside node, it could be at most another $O(\log n)$ complexity again to run the algorithm described in b) on a top inside node m's $\tau_{assoc}(m)$. Since there could be at most $\log n$ nodes in each boundary path P1 and P2 based on x-coordinate, we can describe the total cost as:

$$
\begin{aligned}
&2\sum_{i=1}^{\log n} O(\log n) \\
&= 2O((\log n)^2) \\
&= O((\log n)^2)
\end{aligned}
$$

Therefore in 2-dimensions, the modified range counting query can be performed in $O(\log n)^2$ time.

Problem 6 Tries [3+3+3+3+3=15 marks]

a) Construct a trie on the following eight strings (include edge labels for clarity): 000000, 1111111, 1111100, 011, 11011, 001111, 001100, 000100.

b) Draw the compressed trie equivalent to the trie in the previous part.

c) Draw the compressed trie of the previous part after inserting 00011.

d) Draw the compressed trie of part (b) after inserting 111011.

e) Draw the compressed trie of part (b) after deleting 001100.

Problem 7 KMP [6+6=12 marks]

a) For each of the following pattern strings, determine the Knuth-Morris-Pratt failure array:

1. $P =$ `she sells seashells`

j	P[0 ... j]	P[1 ... j]	F[j]
0	s	-	0
1	sh	h	0
2	she	he	0
3	she_	he_	0
4	she_s	he_s	1
5	she_se	he_se	0
6	she_sel	he_sel	0
7	she_sell	he_sell	0
8	she_sells	he_sells	1
9	she_sells_	he_sells_	0
10	she_sells_s	he_sells_s	1
11	she_sells_se	he_sells_se	0
12	she_sells_sea	he_sells_sea	0
13	she_sells_seas	he_sells_seas	1
14	she_sells_seash	he_sells_seash	2
15	she_sells_seashe	he_sells_seashe	3
16	she_sells_seashel	he_sells_seashel	0
17	she_sells_seashell	he_sells_seashell	0
18	she_sells_seashells	he_sells_seashells	1

2. $P =$ `abracadabracapabra`

j	P[0 ... j]	P[1 ... j]	F[j]
0	a	-	0
1	ab	b	0
2	abr	br	0
3	abra	bra	1
4	abrac	brac	0
5	abraca	braca	1
6	abracad	bracad	0
7	abracada	bracada	1
8	abracadab	bracadab	2
9	abracadabr	bracadabr	3
10	abracadabra	bracadabra	4
11	abracadabrac	bracadabrac	0
12	abracadabraca	bracadabraca	1
13	abracadabracap	bracadabracap	0
14	abracadabracapa	bracadabracapa	1
15	abracadabracapab	bracadabracapab	2
16	abracadabracapabr	bracadabracapabr	3
17	abracadabracapabra	bracadabracapabra	4

3. $P =$ `ababac`

j	P[0 ... j]	P[1 ... j]	F[j]
0	a	-	0
1	ab	b	0
2	aba	ba	1
3	abab	bab	2
4	ababa	baba	3
5	ababac	babac	0

b) Show how to search for pattern $P =$`ababac` in the text $T =$`abcaabaababababacabcaa` using the KMP algorithm. Indicate in a table such as Table 2 which characters of P were compared with which characters of T. Follow the example on slide 25 in module 8. Place each character of P in the column of the compared-to character of T. Put brackets around the character if an actual comparison was not performed. You may not need all space in the table.

a	b	c	a	a	b	a	a	b	a	b	a	b	a	c	a	b	c	a	a
a	b	a																	
		a																	
			a	b															
				a	b	a	b												
						(a)	b												
							a	b	a	b	a	c							
									(a)	(b)	(a)	b	a	c					

Table 1: Table for problem 3(b).

Problem 8 Boyer-Moore [5+5+5=15 marks]

a) Compute the last-occurrence function L for the pattern $P =$ `aabaab`.
Give your answer as a table as shown on slide 32 of module 8.

for each c in the alphabet

c	a	b
L(c)	4	5

b) Compute the suffix skip array S for the pattern $P =$ `aabaab`.
Give your answer as a table as shown on slide 33 of module 8.

i	0	1	2	3	4	5
P(i)	a	a	b	a	a	b
S(i)	-6	-2	-1	-3	-2	4

c) Trace through the execution of Boyer-Moore algorithm for

$P =$ `aabaab`

$T =$ `aaababdaabaaa`

Indicate clearly the sequence of characters compared. Whenever a mismatch occurs, show clearly how the indices to T and P are computed.

1. Row 0: T[3] != P[3]. According to the suffix array, S[3] = -3 so either put P[-3], a at this spot, which would be a shift of 4 to the right, or using the last occurrence array put P[4] here as index 5 is the last occurrence of b, which would shift it 2 in the wrong direction. We therefore use the suffix array option and shift 4 right.
2. Row 1: T[11] != P[5]. According to the suffix array, S[5] = 4 so either put P[4] at this spot, which would be a shift of 1 to the right, or using the last occurrence array put P[4] here as index 4 is the last occurrence of an a. Either one of these options shifts the pattern over one to the right.
3. Row 2: T[12] != P[5]. According to the suffix array, S[5] = 4 so either put P[4] at this spot, which would be a shift of 1 to the right, or using the last occurrence array put P[4] here as index 4 is the last occurrence of an a. Either one of these options shifts the pattern over one to the right.

a	a	a	b	a	b	d	a	a	b	a	a	a
			a	a	b							
											b	
												b

Table 2: Table for problem 3(b).

Problem 9 Suffix Trees [8 marks]

Draw the suffix tree corresponding to the text $T =$ `quisquam`. Use the recipe of Slide 37 of Module 8. Your suffix tree should look like the example on Slide 38. Children of a node should be ordered alphabetically.

Problem 10 Lempel-Ziv compression [8+1+1+1=11 marks]

After writing his memoirs, Foghorn Leghorn has decided to compress them using Lempel-Ziv compression with six bit codes. The code table is initialized with the codes $(a, 000000)$, $(b, 000001)$, $\ldots$, $(z, 011001)$, $(\llcorner, 011010)$, $(', 011011)$, $(,, 011100)$.

a) Give the sequence of substrings that will be used during the compression of the opening sentence "that's a joke, that's a joke, son, that's a joke". If we consider the example of Slide 26 of Module 9, we want you to answer

$$[Y, O, !, \llcorner, YO, U, !\llcorner, YOU, R, \llcorner Y, O, YO, !].$$

You can assume that the dictionary is big enough to store all the substrings that we will encounter.

b) What is the size in bits of the classical encoding of the original string considering that we would code every character from a to ',' on 5 bits if we were not using LZW ?

c) What is the size in bits of the compressed string ?

d) Given the amount of repetition in the sentence above, comment on the quality of compression of LZW on this instance.