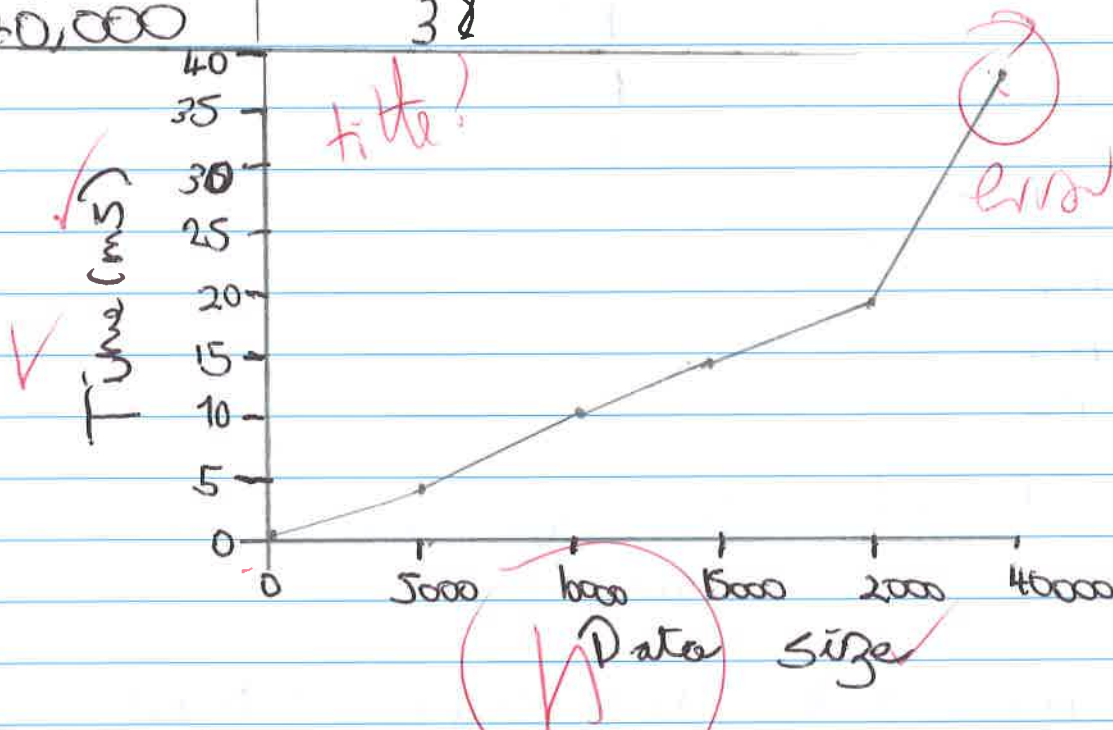


PULL & TEAR

FOR ADMINISTRATIVE USE ONLY, CAREFULLY REMOVE PERFORATED STRIP

Experiment 1

Size Array	Time (ms)
5,000	4
10,000	10
15,000	13
20,000	19
40,000	38



Looking at this graph, I think it's fair to state that `afun1` has a time complexity of $O(n^2)$. I'm able to come to such a ~~conclusion~~ conclusion due to the simple fact that when the data is doubled in size, so is the time taken. If you specifically look at test 4 and test 5, you can see that the size of the data is twice as large / as is the time taken.

Experiment 2

aFunc2

Size	Time Taken (ms)
5,000	14
10,000	13
15,000	13
20,000	15
40,000	14



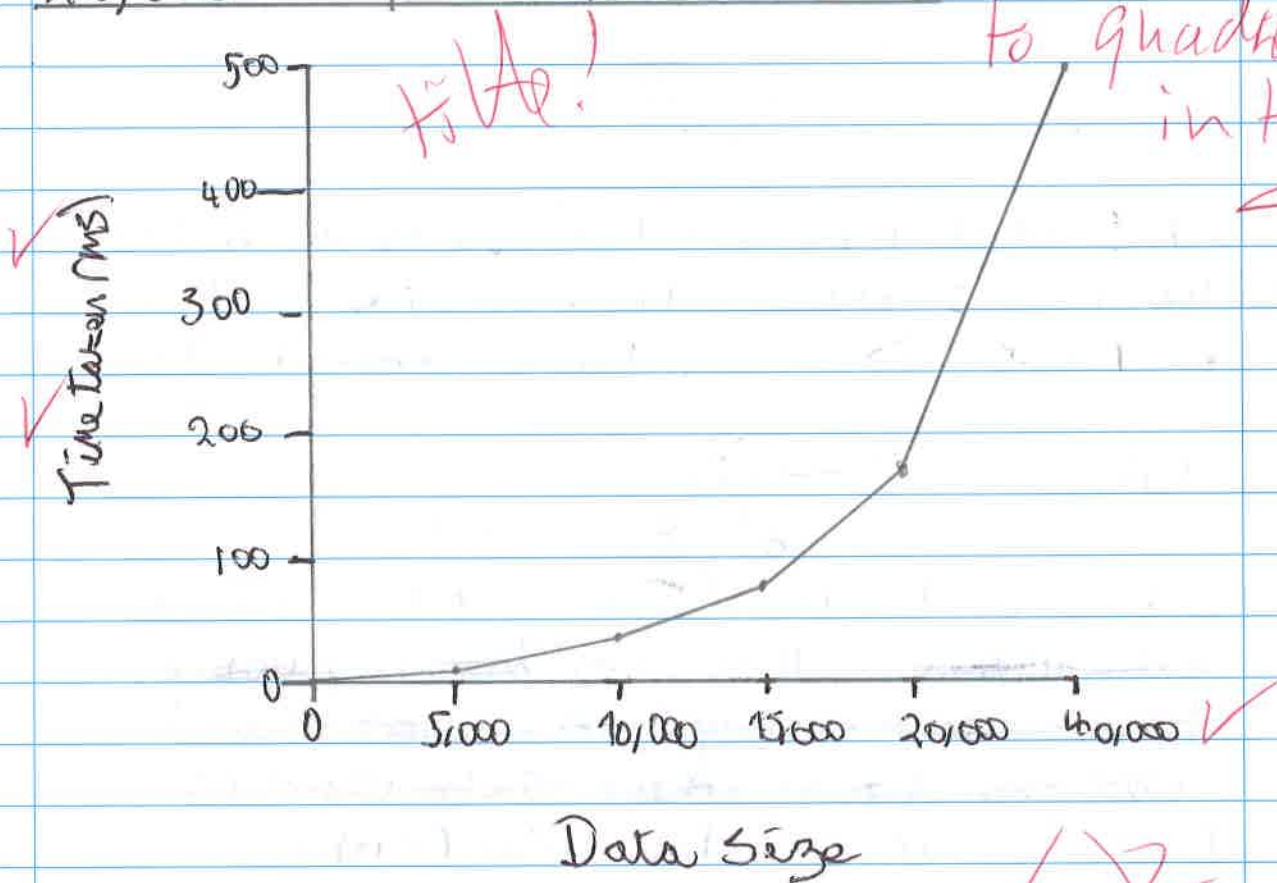
Looking at the above graph, I think it's fair to say that aFunc2 has a time complexity of $O(1)$. I've come to this conclusion due to the fact that there's no REAL difference in the time taken to execute when looking at the different tests. Yes, there is a minute time difference, with some of the results, but at most it's 2 ms. The 2 ms difference could be a number of things, and considering test 5 took less time than test 4 and the same time as test 1, it's only fair to come to such a conclusion, when you consider the different data sizes.

Experiment 2

AFunc3

Size	Time Taken (ms)
5,000	8
10,000	30
15,000	63
20,000	161
40,000	494

doubling
data leads
to quadrupling
in time.



Looking at the graph above, it would appear that AFunc3 has a time complexity of $O(n^3)$. I've come to such a conclusion due to looking at the curve on the graph firstly.

A curve such as this heavily implies a complexity of $O(n^3)$. However, when looking at the data, whenever you double the data, the time taken increases by $(x \times 3)$. Take the difference of test 4 and

test 5. Test 5 takes near enough 3 times the time test 4 took.

13
25

13
25

Question 3

a) Pre-Order [A, C, E₂, E₃, E₁, R₁, H, R₂, S, T] ✓

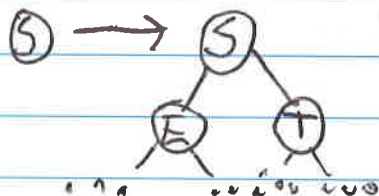
~~[A, C]~~

Post-Order [T, R₂, H, R₁, E₃, E₂, C, A, E₁, S] X

In-Order [E₃, E₂, C, A, R₂, H, R₁, E₁, T, S] ✓

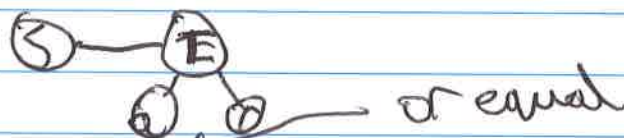
4/12

b) Step 1.



If S, which is the inserted value is less than or equal to the root's value, then go left. Otherwise go right, so $S = S$, we go right left.

Step 2.

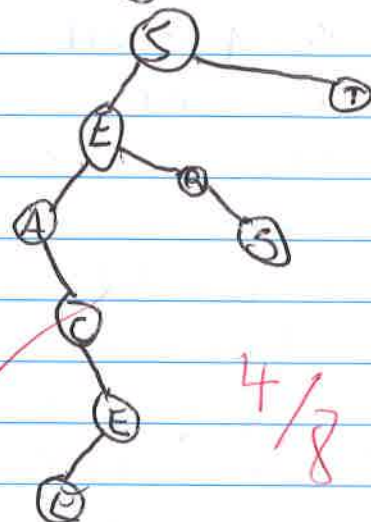


If S is less than ~~E~~ go left, if greater or equal, then go ~~right~~ ^{left} ~~just as before~~ ~~don't anywhere to go from E~~ ~~we can have a S as the left child~~
As S is greater then go right.

Step 3. $S \rightarrow R$

If S is greater than R place it on the right of R, has no ~~the~~ child nodes/leaves. otherwise place it on the left, BUT because ~~R~~ $3 > R$, go right & now exit the algorithm

Result



4/8

① If H were to be deleted from the tree, the tree would need to be redrawn. This is due to the fact that H has one leaf. If the tree is not redrawn, then this means that R_2 would be lost too. Therefore when H gets deleted, R_2 should get moved to where H used to be, as the logic is holding to that of which is applied when inserting. R_2 is equal to R_1 , therefore R_2 can be placed next to R_1 , which prevents R_2 from being lost and the ~~structure~~ structure's order remains intact.

... and a confide algorithm that implements this.

$2\frac{1}{2}$
5

$10\frac{1}{2}$
25

Question 3 B continued

When inserting a node into the tree, at worst, it has a time complexity of $O(\log n)$, because it doesn't need to go through the entire tree to insert the data. However to insert the data, it does need to go through x amount of nodes, hence why it's $O(\log n)$ rather than $O(n)$, or worse.

Question 2

- relationship with compilation?
- a) Static memory allocation defines something which is fixed within memory, an example being a fixed length for an array. Once the array is full, it cannot add more data due to the fixed size. Dynamic memory allocation is the opposite, if you have a dynamic array with an initial length of 10, but you insert an 11th element, the array would grow thanks to the dynamic memory allocation.

- b) A linearly linked list is a simple data structure, it consists of nodes and pointers, where the pointers point to the next node in the linked list.



Add data is an essential, it simply with add data to the end of the list / this can have a complexity of $O(n)$, as you may need to search the entire list to get to the end. Or, it could be $O(1)$, provided you store the last known node in the list.

could place anywhere! in list
you complexity distribution suggests you know this

Another essential operation for a linked list is to search, this however has a complexity of $O(n)$, as you'd go from one node, to the next.

However, you could be creative and decrease the time, however for a basic implementation, to add data to a linked list has a complexity of $O(n)$. 3/5

© Function Insert (data) {

Var copy = this;

If (data < ~~this.data[0]~~ this.data[0]) {

~~this.data[0] = data;~~

~~this.data[0] =~~

this.data += copy;

this.data[0].relation =

this.data[1];

} else { var clone = new DynamicArray(0);

var len = this.data.length; var found = 0;

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

var current = this.data[i];

if (data < ~~this.data[i].current~~ current) {

this.data[i] = data; found = i;

} else { if (found > i) {

clone.add(current);

}

}

this.data[this.data.length].relation =

clone[0];

}

The only major issue that I can see with my implementation is that it would perform poorly if the data set is large. Or if you want to insert something that's bang in the middle. As if you wanted to insert an element (node) in the middle, it would take $O(n)$ time as it would loop through the entire data set, regardless.

7/10

- ① A ll link would be needed for the linked list to implement ~~as~~ a stack. It is to store the end of the list. This way, when you add (Push) / remove (Pop) data to / from the stack, it could do it ~~also~~ looking at the end of the list, and reapplying the end of the list.

explains what you mean by end

An example:

class linkedList {

properties = {

data = dynamic array;

start = this.data[0];

end = this.data[this.data.length];

Size = this.data.length; }

}

~~no~~ need ~~for~~ size checker.

Then the stack's push could be done like so:

function Push(data) {

this.data[this.Size+1] = data;

~~this~~ this.end = this.data[this.Size];

or could use front
put at front

~~this is~~

Then for POP, you could do something like so:

function pop() {

var del = ~~return~~ this.data[this.size];

this.data[this.size] = null;

this.size--;

this.end = this.data[this.size];

return del;

}

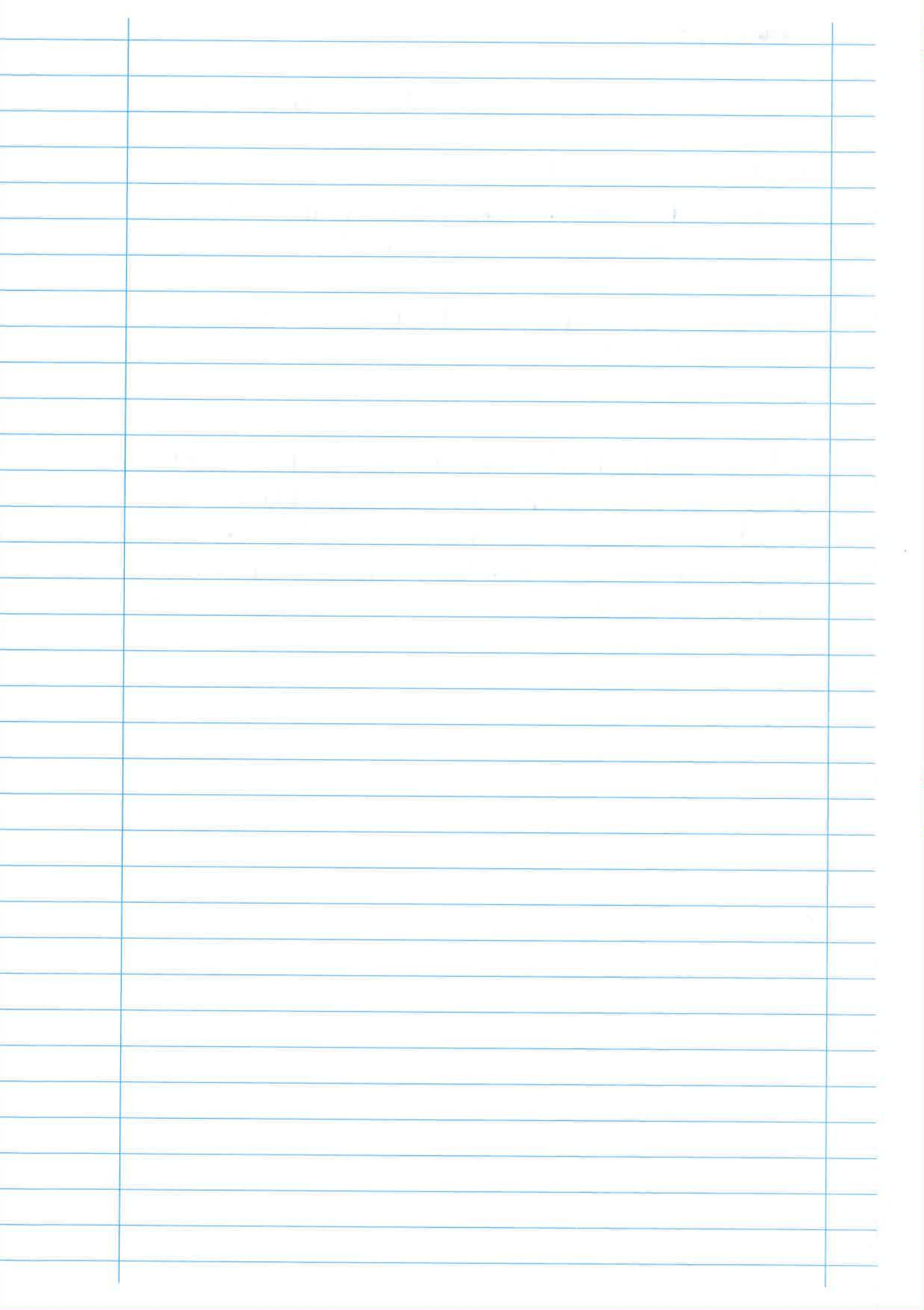
The above function holds all of the functionality that you'd expect to see when removing data from a stack, only it's implemented using a very raggle ^{mmm} implementation of a linked list.

help!

34/6

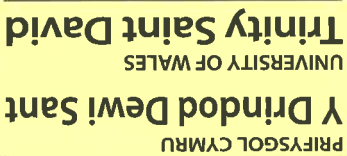
use diagrams
them.

16/25



TYNNWCH A
RHWYGWCH

AR GYFER DEFNYDD GWEINYDDOL YN UNIG, TYNNWCH Y STRIBYN TYLLOG I FFWRDD YN OFALUS



24 Tudalen

Rhaid dychwelyd pob llyfr. Ni chaniaateir mynd â llyfrau atebion, wedi eu defnyddio neu heb eu defnyddio, allan o'r lleoliad arholi o dan unrhyw amgylchiadau.

Rhif Myfyritwr	Teil y Papur Arholiad	Cod y Modwl	Dyddiad ac Amser yr Arholiad	Lleoliad yr Arholiad	Rhif Sedd	Nifer y Llyfrau Atebion a gyflwynwyd
----------------	--------------------------	-------------	---------------------------------	-------------------------	-----------	--

Rhestwch nïau'r cwestiyonau a ymgeisiwyd gennydh		Cyfanswm
At Ddbentlon y Swyddfa		

(See the other side for English version)

Cyfenw

(LLYTHRENNAU BREISION) Enw(au) cyntaf

PLYGWCH