1.
struct { int x, y; } x;
struct { int x, y;} y;
Are these declarations legal on an individual basis? Could both declarations appear as shown in a program?

Yes it is legal because we are declaring two separate variables as x and y. For example, this is like calling x = x+y and y = x+y. Although they share the same stuff inside, it is separated by the variable defining them which is the x and y. Therefore, this declaration in the structure is fine.

2.double a takes 8 bytes, Inside the Union: b[4] takes 4 bytes(because its declared that it takes 4 bytes by [4]), double c takes 8 bytes, int d takes 4 bytes because its integer, so the function inside the union takes a 8 bytes because that's the highest amount of bytes it that's called. Char f[4] takes 4 bytes. So in conclusion, we have 8 byes from double a, 8 bytes from union, and 4 bytes from char f[4]
8+8+4 = 20, so in total, s takes 20 bytes

3. 16 bytes for u because the code provided is a union that takes the bytes of the largest member inside of it. Initially, we would think that the largest member is the double a because of the fact that it takes 8 bytes, but since there is a structure in the Union, we are gonna need to evaluate the space of the structure. Luckily for us, the structure is the same as shown for the question #2 except it's a structure instead of a nested Union. So just add up the bytes, so once again, inside the struct, we have a double=8 bytes, char b[4] = 4 bytes, and int which takes 4 bytes. 8+4+4=16, so the structure is 16 bytes in total. Because it is the largest byte inside the Union, the Union allocates 16 bytes for u.

4.

a) Yes it is legal and safe as b now stores the Boolean FALSE
b) You can assign b = i, and when b is called, it will return i as an integer. Only issue is with the way the current code is structured, we didn't define what integer i is, so it can return unpredicted results so right now, b=i is NOT SAFE
c) It is definitely not safe when incrementing b because it is an enum storing TRUE and FALSE BOOLEANS which can only return 0 or 1. Incrementing it can lead to a result of 2 which is outside of the boolean range, so the result will not be meaningful. It will be perfectly legal if the enum is an integer though.
d) Yes safe and legal, this is one of those scenarios where you i = b is fine and acceptable but not b = i. The reason why i = b is fine is because we already defined our enum value for b to be either 0 or 1 for True or False Boolean values, so depending on the result, it will return a valid predictable result of 0 or 1.
e) i = 2 * b + 1, Ok, looks complex but it is not too bad as this mostly deals with arithmetic simple math so this result is both legal and safe. Explanation is, we already mentioned that b can be either 0 or 1 due to it being assigned as a TRUE or FALSE Boolean. So it will look like this i = 2(0)+1, i =1 or i = 2(1) +1, i=3. So we can expect a result of either i =

1 or i = 3 depending on the result of the boolean function that is enumerated. Therefore, since the result is meaningful, it makes this safe.

5.

a) Not legal because b is defined as an integer, but that integer is both inside the struct and nested into a union that is called d. Therefore, to reference the structure you need to call the Union as well before you can access the integer b. So the corrected version will be p->d.b = ' ';
b) Legal because e [3] is inside the Union but not inside any defined structures so no need to call any of the other structure functions to access it. It looks like we are accessing an array at the 3rd index, and calling it 10 which is perfectly valid
c) (*p).d.a = '*'; legal because it is access the char a which is inside of the union d using the pointer *p
d) Not legal because p->c means p referring to c, but since c is actually inside of d because d is a union, you have to call it using a dot like this, d.c, so the Corrected is p->d.c = 20;,

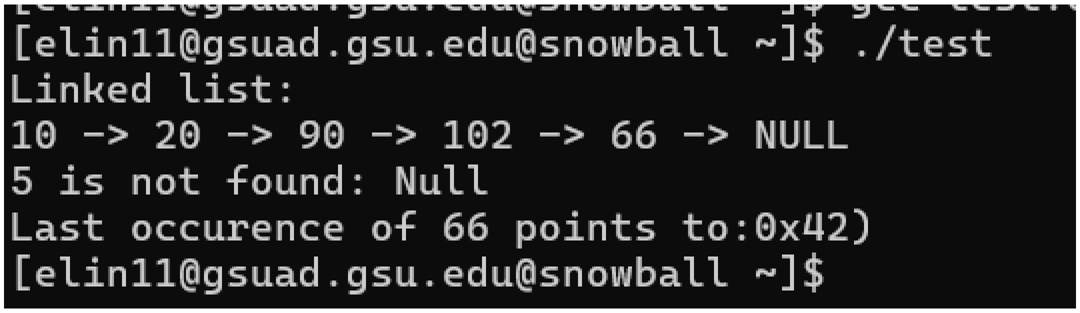
6. The issue with the code is trying to run p = p->next in the same loop in one line as this can cause an unpredictable outcome. The reason it produced an unpredictable outcome is when the loop runs and it calls free(p), it will still continue to try to go next because this is written in the same condition as the checking for a value that isn't NULL.To fix this error just write that condition inside the for loop.

```
for (p = first; p != NULL;) {
p = p->next
free(p)}
```

7.while (cur->value <= new_node->value), what happens when the loop ends here? When Null reaches the end of the iteration, it becomes Null ->value, but when it arrives at NULL the loop is supposed to stop but it does not because as soon as it reaches null, it points to another value. This can cause an infinite loop. We can fix this issue by adding another condition, for example, Curr != NULL && cur->value <= new_node->value which sets the requirement for if it reaches null, the condition fails and won't continue to execute the loop.

8.

```
struct node *find_last(struct node *list, int n) {
   struct node *result = NULL;
   while (list != NULL) {
      if (list->value == n)
         result = list;
      list = list->next;
   }
   return result;
}
```



```
[elin11@gsuad.gsu.edu@snowball ~]$ gcc test.
[elin11@gsuad.gsu.edu@snowball ~]$ ./test
Linked list:
10 -> 20 -> 90 -> 102 -> 66 -> NULL
5 is not found: Null
Last occurence of 66 points to:0x42)
[elin11@gsuad.gsu.edu@snowball ~]$
```