Computation I 5EIA0
Exercise 4: Sorting (v1.4, September 17, 2020)
Deadline Wednesday 30 September 23:55

Figure 1: The sorting hat: `https://www.youtube.com/watch?v=xQZFWA2KDbw`

In this exercise you will develop a program that can be used to sort an array of person records based on their height. Sorting an array has many practical applications. For example, many organisations and web based applications have to deal with huge amounts of data. The data will often have to be accessed multiple times. Keeping the data in a sorted format allows for quick and easy recovery of data.
You will write an interactive program that you can give commands. These are the commands that your program will support.

command	operation
i	initialise array to new size
p	print all the array values
h	print smallest and largest heights and the range of heights
r	replace element
s	sort on value with insertion sort
m	sort on value with merge sort
b	sort on value with bubble sort
d	display array graphically
q	quit program

All of the C exams will use a similar style, so it's a good idea to do this exercise yourself.

Task 1. For the first step, implement the quit command. Print the prompt `Command:` and ask for a single-character input. Print `Bye` when you read the 'q' character and exit the program. The output of your program should look as follows:

```
Command: q
Bye
```

Hint: Use `scanf(" %c",&cmd);` to read a single character, skipping all whitespace (spaces, newlines, tabs).

Task 2. Outside your `main` function define a structure `struct person_t` with a `char *name` field and a `float height` field. Note that you can define a type without defining a variable.

```
struct person_t {
  char *name;
  float height;
};
```

In your `main` function define an array `database` of `MAXLEN` person records, with initial `size` 0.

Task 3. Implement the initialise command in the function `int initialise(struct person_t database[], int size)`
Add `#define MAXLEN 30` to your program. Ask for the size of the array, and check that it's not less than 1 or larger than MAXLEN. If the array size is invalid then the existing array is unchanged. Initialise the array with the person records `("person 0",size-1)` (first element) to `("person X",size-1)` (last element, where X equals size-1). The function returns the new size of the array.
To automatically generate the `size` strings `"person 0"` to `"person X"` you can use the following code.

```
a[i].name = (char *) malloc (10*sizeof(char)); // ok for size <= 100
sprintf(a[i].name, "person %d",i);
```

First you must allocate sufficient space on the heap for a string. Then you fill the space with the `"person X"` string using the `sprintf` function. The `sprintf` function is just like the `printf` function, except that it puts the string it prints not on the screen but in its first element. Of course, you need to do this for all `size` elements in a loop.
Note that it should be possible to re-initialise the array, i.e. execute the 'i' command multiple times. The output of your program should look as follows:

```
Command: i
Size? 1
Command: i
Size? 10
Command: i
Size? 0
Array size must be from 1 to 30.
Command: q
Bye
```

Task 4. Is your program correct when you execute the initialise command multiple times? Hint: does it contain a memory leak?
You should `free` any existing strings in the database before you malloc space for new strings. Fix your program!

Task 5. Write a function `void printValues(struct person_t database[], int from, int to)` that prints the values of elements `values[from]` to `values[to]` (inclusive). For the 'p' command print all the elements of the array (with its current size). Make sure that exactly three digits after the decimal point are printed. Use round brackets around every person, and square brackets and separating commas for the array as a whole.

```
Command: p
[]
Command: i
Size? 3
Command: p
[("person 0",2.000),("person 1",1.000),("person 2",0.000)]
Command: q
Bye
```

Task 6. Write a function `float largestElement(struct person_t database[], int size)` that returns the largest height contained in first `size` elements of the `database` array and a similar function `smallestElement`. These two functions allow you to compute range of elements (max-min). The minimum and maximum are 0 when the database is empty. The output of your program should look as follows:

```
Command: h
Min: 0000
Max: 0.000
Range: 0.000
Command: i
Size? 4
Command: h
Min: 0000
Max: 3.000
Range: 3.000
```

Task 7. Implement the replace command by writing a function `void replaceElement(struct person_t database[])` First ask for an index `i` and give an error message if the index is out of range. Then ask for a new name and a new height. Update `a[i].name` and `a[i].height`. Which of the following C code fragments do you think you should use? What goes wrong in the other code fragments?

```
// ---------- code fragment A ---------- //
char newstring[100];
printf ("Name? ");
scanf (" %[^\n]s", newstring);
a[i].name = newstring;

// ---------- code fragment B ---------- //
char newstring[100];
printf ("Name? ");
scanf (" %[^\n]s", newstring);
a[i].name = (char *) malloc (100);
strcpy (a[i].name,newstring);

// ---------- code fragment C ---------- //
char newstring[100];
printf ("Name? ");
scanf (" %[^\n]s", newstring);
a[i].name = (char *) malloc (strlen(newstring));
strcpy (a[i].name,newstring);

// ---------- code fragment D ---------- //
char newstring[100];
printf ("Name? ");
scanf (" %[^\n]s", newstring);
a[i].name = (char *) malloc (strlen(newstring)+1);
strcpy (a[i].name,newstring);
```

```
Command: i
Size? 4
Command: p
[("person 0",3.000),("person 1",2.000),("person 2",1.000),("person 3",0.000)]
Command: r
Index? -1
Index must be from 0 to 3.
Command: r
Index? 2
Name? Kees
Height? 186
Command: p
[("person 0",3.000),("person 1",2.000),("Kees",186.000),("person 3",0.000)]
Command: r
Index? 2
Name? Harry
Height? 125
Command: p
[("person 0",3.000),("person 1",2.000),("Harry",125.000),("person 3",0.000)]
Command: q
Bye
```

Note that in order to replace the second element, the user has to input the index 1! This is because the first element of an array is placed at index 0.

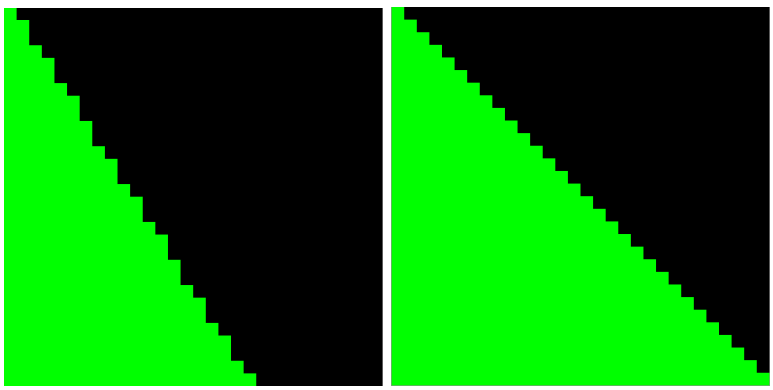
Task 8. Do you have a memory leak in your `replaceElement` function? What happened to the `name` string that you replaced? Fix your program if necessary.

Task 9. In this task you will display the array graphically using the 'd' command. Write a function `void display_array (struct person_t database[], int size, int from, int to)` that displays a bar graph of the `size` elements in the array `values`. The bars from element `from` to element `to` (inclusive) are red, the remaining bars are green. Normalise the height of the bars such that the smallest element has height 0 and the largest element has height `HEIGHT`. Here are some code snippets to help:

```
#define MAXLEN 30
#define WIDTH MAXLEN
#define HEIGHT 30
...
void displayValues (struct person_t database[], int size, int from, int to)
{
  const float min = minValue (a,size);
  const float max = maxValue (a,size);
  const float range = max-min;
  // draw the bar of the appropriate height
  // float with minValue becomes bar with height 0
  // float with maxValue becomes bar with height HEIGHT-1
  for (int x = 0; x < size; x++) {
    for (int y = 0; y < HEIGHT; y++) {
      // if pixel is not part of the vertical bar then display a black pixel
      // else if pixel is part of the vertical bar and between from and to then display a red pixel
      // else if pixel is part of the vertical bar and not between from and to then display a green
    }
  }
  draw_display();
  sleep_msec(500);
}

int main (void)
{
  pixel display[HEIGHT][WIDTH];
  init_display (HEIGHT, WIDTH, 20, display);
  ...
}
```

These are example graphical outputs: on the left an array with 20 elements (19 downto 0) and on the right an array with 30 elements (30 down to 0). For the 'd' command all bars are green (hint: set `from=to=-1`). Note that if there are less than 30 elements, the right hand side of the window is not used.



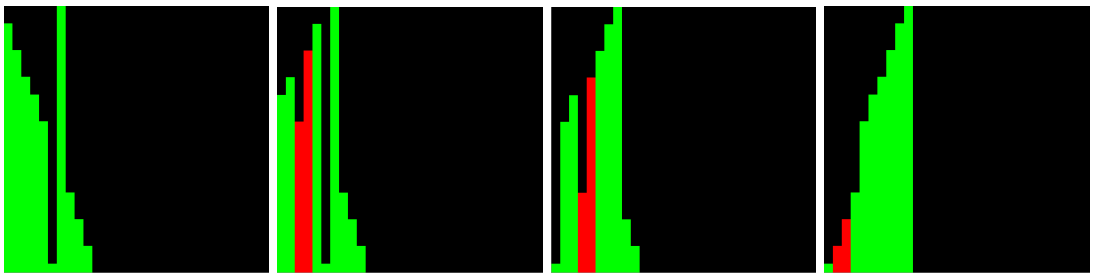
Hint: If your graphics seem slow then it may be faster to login to the PYNQ with `ssh -X` rather than `ssh -XC`. If you find graphics difficult then you can do Tasks 10 & 11 first, and then return to this task.

Task 10. We now implement the first sorting function for the 's' command. Write a function `void insertionsort(struct person_t database[], int size)` that orders all elements inside the array according to their height (low to high), using the insertion sort algorithm. See Wikipedia `https://en.wikipedia.org/wiki/Insertion_sort` for a description. In essence, the first i elements of the array are kept sorted and the i^{th} element is inserted in the right place (by swapping elements). As i goes from 0 (no elements have been sorted yet), to 1 (the first 1 elements have been sorted), to 2 (the first two elements have been sorted), etc. the sorted part of the array grows until the entire array is sorted. The following trace (that your program should produce too) illustrates this for an array of length 4. Notice that sorting the array again takes no work.

```
Command: i
Size? 4
Command: p
[("person 0",3.000),("person 1",2.000),("person 2",1.000),("person 3",0.000)]
Command: s
after swapping: [("person 1",2.000),("person 0",3.000),("person 2",1.000),("person 3",
after swapping: [("person 1",2.000),("person 2",1.000),("person 0",3.000),("person 3",
after swapping: [("person 2",1.000),("person 1",2.000),("person 0",3.000),("person 3",
after swapping: [("person 2",1.000),("person 1",2.000),("person 3",0.000),("person 0",
after swapping: [("person 2",1.000),("person 3",0.000),("person 1",2.000),("person 0",
after swapping: [("person 3",0.000),("person 2",1.000),("person 1",2.000),("person 0",
Command: s
Command: p
[("person 3",0.000),("person 2",1.000),("person 1",2.000),("person 0",3.000)]
Command: q
Bye
```

Hint: Call printValues and displayValues whenever you swap two person records. For displayValues highlight the two swapped values in red.

The images below are for the initial array with heights `[9,8,7,6,5,-1,10,2,1,0]`:



Task 11. When you can sort the database on height, make sure that if there are multiple persons with the same height, then they are ordered on name. Consider `Person 3` and `Person 9` that both have `height 0` in the trace below.

```
Size? 4
Command: p
[("person 0",3.000),("person 1",2.000),("person 2",1.000),("person 3",0.000)]
Command: r
Index? 2
Name? person 9
Height? 0
Command: p
[("person 0",3.000),("person 1",2.000),("person 9",0.000),("person 3",0.000)]
Command: s
after swapping: [("person 1",2.000),("person 0",3.000),("person 9",0.000),("person 3",
after swapping: [("person 1",2.000),("person 9",0.000),("person 0",3.000),("person 3",
after swapping: [("person 9",0.000),("person 1",2.000),("person 0",3.000),("person 3",
after swapping: [("person 9",0.000),("person 1",2.000),("person 3",0.000),("person 0",
after swapping: [("person 9",0.000),("person 3",0.000),("person 1",2.000),("person 0",
after swapping: [("person 3",0.000),("person 9",0.000),("person 1",2.000),("person 0",
Command: p
[("person 3",0.000),("person 9",0.000),("person 1",2.000),("person 0",3.000)]
Command: q
Bye
```

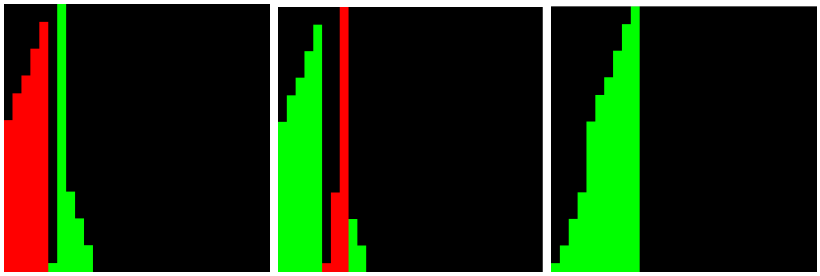
Hint: Use `strcmp` from the `string.h` standard library to compare strings (see Kernighan & Ritchie Appendix B3.)

Task 12. This is an optional task. Insertion sort can be very slow. Recursive sorting methods such as merge sort that was discussed in the lecture often perform much better. Now for the 'm' command implement merge sort with the `void mergesort(struct person_t database[], int size, int left, int right)` that sorts the values `values[left]` to `values[right]`. If `left` is larger than `right` you need to do nothing (this is the end of the recursion). (The `size` parameter is not needed for the sorting, but only for the `printDisplay` below.) To understand how the sorting works you should add several print statements to your code, resulting in the following output:

```
Command: i
Size? 4
Command: p
[3.000,2.000,1.000,0.000]
Command: m
need to sort                  [ 0, 3]: [3.000,2.000,1.000,0.000]
need to sort                  [ 0, 1]: [3.000,2.000]
after sorting left hand side  [ 0, 0]: [3.000]
after sorting right hand side [ 1, 1]: [2.000]
after merging                 [ 0, 1]: [2.000,3.000]
after sorting left hand side  [ 0, 1]: [2.000,3.000]
need to sort                  [ 2, 3]: [1.000,0.000]
after sorting left hand side  [ 2, 2]: [1.000]
after sorting right hand side [ 3, 3]: [0.000]
after merging                 [ 2, 3]: [0.000,1.000]
after sorting right hand side [ 2, 3]: [0.000,1.000]
after merging                 [ 0, 3]: [0.000,1.000,2.000,3.000]
Command: p
[0.000,1.000,2.000,3.000]
Command: q
Bye
```

Hint: The printValues and printDisplay calls are placed immediately after the recursive calls to the `mergesort` and `merge` functions. The numbers `[ 0, 1]` numbers are the `left`, `mid`, and `right` values.

The images below are for the initial array with heights `[9,8,7,6,5,-1,10,2,1,0]`:



Task 13. This is an optional task. Also implement bubble sort, with the 'b' command. Bubble sort is described on Wikipedia (`https://en.wikipedia.org/wiki/Bubble_sort`) and also in the lecture notes.

```
Command: i
Size? 5
Command: p
[("person 0",4.000),("person 1",3.000),("person 2",2.000),("person 3",1.000),("person 4",0.000)]
Command: b
after swapping: [("person 1",3.000),("person 0",4.000),("person 2",2.000),("person 3",1.000),("
after swapping: [("person 1",3.000),("person 2",2.000),("person 0",4.000),("person 3",1.000),("
after swapping: [("person 1",3.000),("person 2",2.000),("person 3",1.000),("person 0",4.000),("
after swapping: [("person 1",3.000),("person 2",2.000),("person 3",1.000),("person 4",0.000),("
after swapping: [("person 2",2.000),("person 1",3.000),("person 3",1.000),("person 4",0.000),("
after swapping: [("person 2",2.000),("person 3",1.000),("person 1",3.000),("person 4",0.000),("
after swapping: [("person 2",2.000),("person 3",1.000),("person 4",0.000),("person 1",3.000),("
after swapping: [("person 3",1.000),("person 2",2.000),("person 4",0.000),("person 1",3.000),("
after swapping: [("person 3",1.000),("person 4",0.000),("person 2",2.000),("person 1",3.000),("
after swapping: [("person 4",0.000),("person 3",1.000),("person 2",2.000),("person 1",3.000),("
Command: p
[("person 4",0.000),("person 3",1.000),("person 2",2.000),("person 1",3.000),("person 0",4.000)]
Command: q
Bye
```

Submission: Submit your file `sorting.c` that implements task 11 (or later tasks if you did them) on Oncourse (`Exercise 4: Sorting (due 30 Sep 23:55)`). You can resubmit as often as you want until the deadline.

- **17/9 v1.4** Removed the text in X.