Computation I 5EIA0
Homework 5: Malloc and Sorting (v3.6, October 2, 2022)
Deadline Tuesday 11 October 13:30

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

In this homework 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
q	quit program
n	new array
p	print the array
i	insert a new person in the array
h	print smallest and largest heights and the range of heights
r	replace element
b	sort on value with bubble sort
m	sort on value with merge sort

function	1	2	3	4	5	6	7	8	9	10	11	12	% per fn	cumulative %
quit	1	1	1	1	1	1	1	1	1	1	1	1	8%	8%
new		1	1	1	1	1	1	1	1	1	1	1	17%	25%
print				1	1	1	1	1	1	1	1	1	8%	33%
insert					1	1	1	1	1	1	1	1	8%	42%
min/max						1	1		1	1	1	1	8%	50%
replace							1	1		1		1	8%	58%
bubble sort								1	1	1			25%	83%
merge sort											1	1	17%	100%

Figure 2: Test cases and points per task

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

```
Command? z
Unknown command 'z'
Command? q
Bye!
```

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

Task 2. In this homework we'll use `malloc` and `free` to dynamically ask the system for space. Include the `stdlib.h` system library (as well as `stdio.h`) at the start of your program. We will keep track of names and heights of persons in an array of structures called `persons`. When we want to declare a variable or function parameter we can write `{struct { char *name; float height; }`, but that is very long. It is better to declare the structure once, like this:

```
struct person {
  char name[20];
  float height;
};
```

at the start of the program, before the first function. In the remainder of the program we can then use `struct person` instead of `struct { char name[20]; float height; }` everywhere. For example, to declare a single person we would use:

```
struct person one_person;
```

To declare an array of 10 persons we would use:

```
struct person person[10];
// instead of struct { char name[20]; float height; } persons[10];
```

The `name` field of the structure is an array of characters with a fixed length of 20. This is quite restrictive and possibly wasteful of space (if most names are shorter). For that reason we will use `malloc` (and `free`) to dynamically ask for exactly the right amount of space for each name. The declaration of the structure then changes to:

```
struct person {
  char *name;
  float height;
};
struct person one_person; // unchanged
struct person person[10]; // unchanged
```

Similarly, instead of declaring an array with a fixed size we will allow the user to dynamically change the size of the array. This requires `malloc` (and `free`). As you recall from the lecture, `int *i;` is a pointer to a `int`. If we declare

```
int *intarray = (int *) malloc (nrInts * sizeof(int));
```

then `intarray` points to `nrInts` integers, and we can use `intarray` just like any other array. e.g. we can use the 10th element with `intarray[9]`. Note that if we free `intarray` and then `malloc` with a different number of entries, then the size of the array has changed. (Unfortunately, we will also have lost all the contents of the array!) Now in your main function declare the variable `persons` that is a pointer to a person (i.e. `struct person`) and initialise it to NULL.

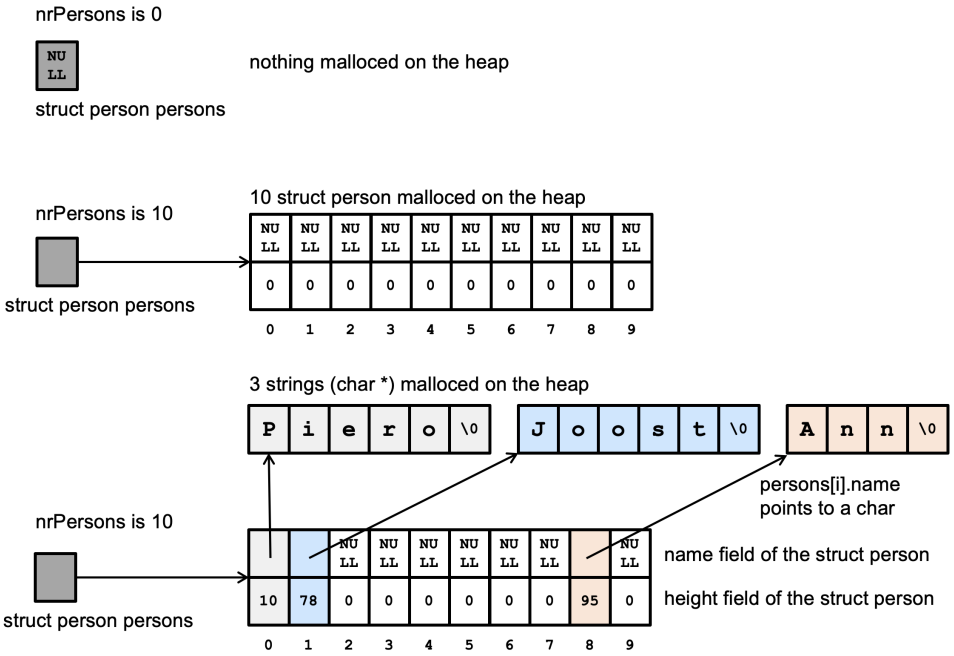

Figure 3: Example of how the `persons` arrays and the strings are laid out in memory.

1. `persons[0].name` contains the address of the first character of the string `"Piero"`, i.e. the address of character `'P'`.
2. `persons[1].name` contains the address of the first character of the string `"Joost"`, i.e. the address of character `'J'`.
3. `persons[8].name` contains the address of the first character of the string `"Ann"`, i.e. the address of character `'A'`.
4. `persons[0].name+1` contains the address of the second character of the string `"Piero"`, i.e. the address of char. `'i'`.
5. `persons[0].name+2` contains the address of the third character of the string `"Piero"`, i.e. the address of character `'e'`.
6. `persons[1].name+4` contains the address of the last character of the string `"Joost"` before the terminating null character, i.e. the address of character `'t'`.
7. And so on.

Task 3. We will now add the 'new' command (character 'n') to the command loop in the main function. This command will initialise the persons arrays with a certain number of entries. Ask for the number of `persons` that can be stored (`nrPersons` variable). Then malloc `nrPersons` struct persons and assign the result to the `persons` arrray. If `malloc` returned NULL then give an error message (`malloc returned NULL`) and exit the program using a `return` statement. All entries of the `persons` array must be initialised with 0.0 for the height field and NULL for the name field. If the number of entries is less than one, then there's no need to malloc any space and `persons` is initialised with NULL. This is an example output:

```
Command? n
Entries? 10
Command? n
Entries? 0
Command? q
Bye!
```

Task 4. Is your program correct when you execute the new command? Hint: does it contain a memory leak? You will probably get an error like this:

```
Command? n
Entries? 10
Command? q
Bye!

=================================================================
==86226==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 160 byte(s) in 1 object(s) allocated from:
    #0 0x7f1541238808 in __interceptor_malloc ../../../../src/libsanitizer/asan/
asan_malloc_linux.cc:144
    #1 0x560ab98ad398 in main /home/p10257/test.c:208
    #2 0x7f1540e090b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24
0b2)

SUMMARY: AddressSanitizer: 160 byte(s) leaked in 1 allocation(s).
```

The reason is that you malloc'd space for the persons array but did not free it before finishing the program. Note that there's a hint that second function on the stack (main) on line 208 in the program (test.c) is involved in the leak.

You should `free` any existing strings in the `persons` array before you re-initialise. The space for the `persons` array should also be freed before you re-initialise. Fix your program!

You should also free all malloc'ed space when executing the quit command. For that reason it may be easiest to have a helper function that you can call twice:

```
struct person *removeAllPersons(struct person *persons, int nrPersons) ...
```

What happens when you call the new command multiple times?

Task 5. Write the function
`void printPersons (struct person persons[], int nrPersons, int from, int to)`
that prints the name and height of each person. Only print the entries `from` to `to` (inclusive). If the entry of the names array is NULL then don't print it. For the print ('p') command print all the persons (from 0 to `nrPersons`). 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. The example output below already uses the insert command that you will implement next, to illustrate the output format.

```
Command? p
[]
Command? n
Entries? 10
Command? p
[]
Command? n
Entries? 3
Command? i
Index? 0
Name? First
Height? 10
Command? i
Index? 1
Name? Second
Height? 11
Command? i
Index? 2
Name? Third
Height? 12
Command? p
[("First",10.000),("Second",11.000),("Third",12.000)]
Command? q
Bye!
```


Task 6. Implement the insert command ('i') in the function

```
void insertPerson(struct person persons[], int nrPersons,
                  int newEntry, char *newName, float newHeight)
```

to insert a new person at the `newEntry` position in the arrays. The height must be larger than zero; give an error message if this is not the case. You can assume that names are at most 80 characters long (incl. null character). Give the error message shown below if the position is already occupied.

```
Command? n
Entries? 10
Command? p
[]
Command? i
Index? 3
Name? MinThree
Height? -3
Height must be larger than zero
Command? i
Index? 3
Name? Three
Height? 3
Command? p
[("Three",3.000)]
Command? i
Index? 2
Name? Two
Height? 2
Command? p
[("Two",2.000),("Three",3.000)]
Command? i
Index? 3
Name? Wrong
Height? 10
Entry 3 is already occupied by ("Three",3.000)
Command? p
[("Two",2.000),("Three",3.000)]
Command? q
Bye!
```


Which of the following C code fragments do you think you should use, in the `insertPerson` function? What goes wrong in the other code fragments?

```
// ---------- code fragment A ---------- //
persons[newEntry].name = newName;

// ---------- code fragment B ---------- //
persons[newEntry].name = (char *) malloc (100);
strcpy (persons[newEntry].name, newName);

// ---------- code fragment C ---------- //
persons[newEntry].name = (char *) malloc (strlen(newName)*sizeof(char));
strcpy (persons[newEntry].name, newName);

// ---------- code fragment D ---------- //
persons[newEntry].name = (char *) malloc ((strlen(newName)+1)*sizeof(char));
strcpy (persons[newEntry].name, newName);

// ---------- code fragment E ---------- //
persons[newEntry].name = (char **) malloc ((strlen(newName)+1)*sizeof(char));
strcpy (persons[newEntry].name, newName);

// ---------- code fragment F ---------- //
persons[newEntry].name = (char **) malloc ((strlen(newName)+1)*sizeof(char *));
strcpy (persons[newEntry].name, newName);
```

Hint: Have a look at the end of this document for an explanation.

Task 7. Does your program have a different memory leak now? You probably got an error like this:

```
Command? n
Entries? 10
Command? i
Index? 0
Name? hello
Height? 10
Command? q
Bye!

=================================================================
==86487==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 6 byte(s) in 1 object(s) allocated from:
    #0 0x7f4fbb076808 in __interceptor_malloc ../../../../src/libsanitizer/asan/
asan_malloc_linux.cc:144
    #1 0x56303ed7b6ef in insertPerson /home/p10691/test.c:62
    #2 0x56303ed7c996 in main /home/p10691/test.c:248
    #3 0x7f4fbac470b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x24
0b2)

SUMMARY: AddressSanitizer: 6 byte(s) leaked in 1 allocation(s).
```

Why and where are the 6 bytes leaked?
The reason is that you malloc'd space for the person's name but did not free it before finishing the program. You should `free` any existing strings in the `persons` array before you re-initialise. Fix your `removeAllPersons` function!

Task 8. Write the function `void maxHeight (struct person persons[], int nrPersons, float *max)` that computes the largest height contained in first `size` elements of the array, and a similar function `float minHeight (struct person persons[], int nrPersons)`
Note that `maxHeight` returns its value through a call-by-reference argument, while `minHeight` has a float return value. 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: 0.000
Max: 0.000
Range: 0.000
Command? n
Entries? 4
Command? i
Index? 0
Name? Tall
Height? 212
Command? i
Index? 1
Name? Small
Height? 121
Command? i
Index? 2
Name? Kees
Height? 186
Command? p
[("Tall",212.000),("Small",121.000),("Kees",186.000)]
Command? h
Min: 121.000
Max: 212.000
Range: 91.000
Command? q
Bye!
```

Task 9. Implement the replace ('r') command by writing the function

```
void replacePerson(struct person persons[], int nrPersons,
                   int newEntry, char *newName, float newHeight)
```

Ask for an index, name, and height as shown below. If the current entry is empty then just insert the new entry. The height must be larger than zero; give the same error message as for the insert command if this is not the case.

```
Command? n
Entries? 5
Command? i
Index? 0
Name? Newton
Height? 9.8
Command? i
Index? 1
Name? Fourier
Height? 45
Command? i
Index? 2
Name? Pascal
Height? 121
Command? p
[("Newton",9.800),("Fourier",45.000),("Pascal",121.000)]
Command? r
Index? 1
Name? Transform
Height? 54
Command? p
[("Newton",9.800),("Transform",54.000),("Pascal",121.000)]
Command? r
Index? 3
Name? Insert
Height? 1
Command? q
Bye!
```

Hint: If you got the following error message then you should reread Tasks 3 and 6.

```
      =================================================================
==571946==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 17 byte(s) in 3 object(s) allocated from:
    #0 0x7f5e7dcf6808 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan
    #1 0x557a7c9ad6ef in insertPerson /home/p11574/test.c:62
    #2 0x557a7c9ae9b4 in main /home/p11574/test.c:248
    #3 0x7f5e7d8c70b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x240b2)

Direct leak of 13 byte(s) in 2 object(s) allocated from:
    #0 0x7f5e7dcf6808 in __interceptor_malloc ../../../../src/libsanitizer/asan/asan
    #1 0x557a7c9ad897 in replacePerson /home/p11574/test.c:90
    #2 0x557a7c9ae872 in main /home/p11574/test.c:239
    #3 0x7f5e7d8c70b2 in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x240b2)

SUMMARY: AddressSanitizer: 30 byte(s) leaked in 5 allocation(s).
```


Task 10. Do you have a memory leak in your `replacePerson` function? What happened to the `name` string that you replaced? Fix your program if necessary.

The following tasks ask to implement a non-recursive and recursive sorting function. Bubble sort is good to practice basic programming skills. Merge sort is good to practice recursion.

Task 11. Implement bubble sort ('b' command) with the function
`void bubbleSort(struct person persons[], int nrPersons, int *swapped)`
Bubble sort is described on Wikipedia (`https://en.wikipedia.org/wiki/Bubble_sort`) and also in the lecture notes. 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. You can assume that all entries in the `persons` array are occupied (i.e. not NULL). That saves some coding complexity.

```
...
Command? p
[("person 3",3.000),("person 2",2.000),("person 1",1.000),("person 0",0.000)]
Command? b
after swapping: [("person 2",2.000),("person 3",3.000),("person 1",1.000),("person 0",0.000)]
after swapping: [("person 2",2.000),("person 1",1.000),("person 3",3.000),("person 0",0.000)]
after swapping: [("person 2",2.000),("person 1",1.000),("person 0",0.000),("person 3",3.000)]
after swapping: [("person 1",1.000),("person 2",2.000),("person 0",0.000),("person 3",3.000)]
after swapping: [("person 1",1.000),("person 0",0.000),("person 2",2.000),("person 3",3.000)]
after swapping: [("person 0",0.000),("person 1",1.000),("person 2",2.000),("person 3",3.000)]
Swapped 6 times
Command? p
[("person 0",0.000),("person 1",1.000),("person 2",2.000),("person 3",3.000)]
Command? b
Swapped 0 times
Command? q
Bye!
```

Task 12. When you can sort the arrays on height, make sure that if there are multiple persons with the same height, then they are ordered on name. Consider `Wacko` and `Xuxa` that both have `height` 2 in the trace below.

```
...
Command? p
[("Zebob",10.000),("Xuxa",2.000),("Wacko",2.000),("Beeblebrox",1.000)]
Command? b
after swapping: [("Xuxa",2.000),("Zebob",10.000),("Wacko",2.000),("Beeblebrox",1.000)]
after swapping: [("Xuxa",2.000),("Wacko",2.000),("Zebob",10.000),("Beeblebrox",1.000)]
after swapping: [("Xuxa",2.000),("Wacko",2.000),("Beeblebrox",1.000),("Zebob",10.000)]
after swapping: [("Wacko",2.000),("Xuxa",2.000),("Beeblebrox",1.000),("Zebob",10.000)]
after swapping: [("Wacko",2.000),("Beeblebrox",1.000),("Xuxa",2.000),("Zebob",10.000)]
after swapping: [("Beeblebrox",1.000),("Wacko",2.000),("Xuxa",2.000),("Zebob",10.000)]
Swapped 6 times
Command? p
[("Beeblebrox",1.000),("Wacko",2.000),("Xuxa",2.000),("Zebob",10.000)]
Command? q
Bye!
```

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

Task 13. Implement the recursive merge sort ('m' command) with the function
`void mergeSort(struct person persons[], int left, int right)`
Merge sort is described in the lecture notes. You can assume that all entries in the `persons` array are occupied (i.e. not NULL). That saves some coding complexity.

```
...
Command? p
[("person 3",3.000),("person 2",2.000),("person 1",1.000),("person 0",0.000)]
Command? m
Command? p
[("person 0",0.000),("person 1",1.000),("person 2",2.000),("person 3",3.000)]
Command? m
Command? q
Bye!
```

Submission: Submit your file `sorting.c` on Oncourse. You can resubmit as often as you want until the deadline.

Answers to the question which code fragment is right in Task 5:

Code fragment A

```
persons[newEntry].name = newName;
```

This assigns the start address of newName to persons[newEntry].name. But the function parameter `newName` contains the address of the local variable `newName` in the main function and it will be overwritten on the next insert command. This would be the result:

```
Command? n
Entries? 4
Command? i
Index? 1
Name? Kees
Height? 186
Command? p
[("Kees",186.000)]
Command? i
Index? 2
Name? Joris
Height? 188
Command? p
[("Joris",186.000),("Joris",188.000)]
Command? q
Bye!
```

Notice how the string Kees has been replaced by Joris, which is wrong. The height is still correct since that is a floating-point value rather than a pointer. The figure shows what goes wrong. Refer to the figure in Task 2 for how it should be.

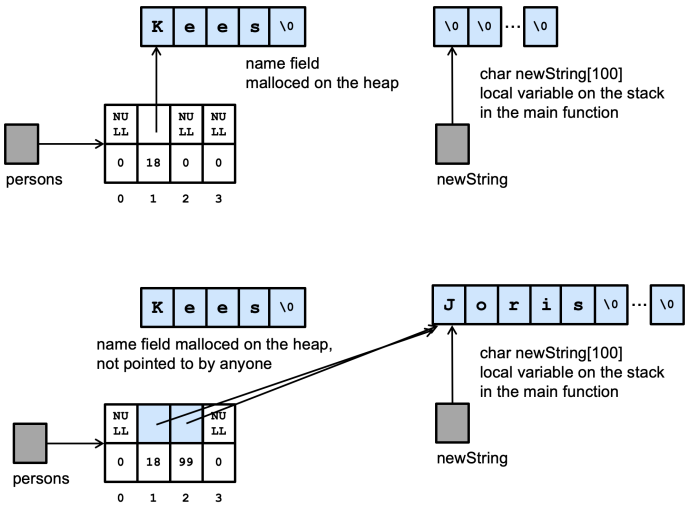

Figure 4: Code fragement A

Code fragment B

```
persons[newEntry].name = (char *) malloc (100);
strcpy (persons[newEntry].name, newName);
```

This code does not make the same mistake as fragment A because it creates a new string on the heap. However, since the malloc is for a fixed 100 characters, if the user supplies a longer string (newName) then this will go wrong. `strcpy` assumes that newName will fit in persons[newEntry].name, but it won't and some memory will be overwritten.

Code fragment C

```
persons[newEntry].name = (char *) malloc (strlen(newName)*sizeof(char));
strcpy (persons[newEntry].name, newName);
```

This code does not make the same mistake as fragment B because it creates a new string, but it is one character too short. Recall that `strlen` returns the length of the string *without* the null character. Again `strcpy` assumes that newName will fit in persons[newEntry].name, but it won't and one byte after the end of the malloced space will incorrectly overwritten.

Code fragment D

```
persons[newEntry].name = (char *) malloc ((strlen(newName)+1)*sizeof(char));
strcpy (persons[newEntry].name, newName);
```

This code is correct.

```
Command? n
Entries? 4
Command? i
Index? 1
Name? Kees
Height? 186
Command? p
[("Kees",186.000)]
Command? i
Index? 2
Name? Joris
Height? 188
Command? p
[("Kees",186.000),("Joris",188.000)]
Command? q
Bye!
```

Code fragment E

```
persons[newEntry].name = (char **) malloc ((strlen(newName)+1)*sizeof(char));
strcpy (persons[newEntry].name, newName);
```

This code is incorrect because the types don't match. Recall from the slides that this is the correct way:

```
TYPE *var = (TYPE *) malloc (number*sizeof(TYPE));
```

Code fragment F

```
persons[newEntry].name = (char **) malloc ((strlen(newName)+1)*sizeof(char *));
strcpy (persons[newEntry].name, newName);
```

Although the types match on the right-hand side in this code fragment, the left and right-hand side don't match:

```
TYPE *var = (TYPE *) malloc (number*sizeof(TYPE));
```


- 17/9 v1.4 Removed the text in X.
- 30/9 v1.5 Function name consistency.
- 4/8 v2.0 Almost completely new version.
- 23/9 v2.1 Updated minigrind.
- 27/9 v2.2 Minor updates.
- 28/9 v2.3 Minor updates.
- 29/9 v2.4-5 Include minigrind as .h file to avoid compilation issues on Oncourse. Explicit instruction that minigrind.h needs to be downloaded.
- 30/9 v2.6 Explained min/maxHeight output.
- 30/9 v2.7 Min/max now work since insertPerson must be ¿0.
- 29/7 v3.0 Use (simpler) struct instead of two arrays. Removed PYNQ graphics. Added merge sort. Removed minigrind.
- 4/8 v3.1 Added removeAllPersons.
- 11/8 v3.2 More instructions for when using libasan.
- 22/9 v3.3 Removed old negative height in example.
- 2/10 v3.6 Clarified negative height for replace command, fixed table & typos.

Input / output test cases

Long lines have been wrapped at 70 characters for legibility. When your program output is compared to the expected output lines will not be wrapped.

Case 01

Input:

```
X
q
```

Output:

```
Command? Unknown command 'X'
Command? Bye!
```


Case 02

Input:

```
n
10
q
```

Output:

```
Command? Entries? Command? Bye!
```


Case 03

Input:

```
n
10
n
0
n
2
n
0
n
0
n
10000
n
0
q
```

Output:

```
Command? Entries? Command? Entries? Command? Entries? Command?
Entries? Command? Entries? Command? Entries? Command? Entries?
Command? Bye!
```


Case 04

Input:

```
p
n
10
p
n
0
p
n
0
q
```

Output:

```
Command? []
Command? Entries? Command? []
Command? Entries? Command? []
Command? Entries? Command? Bye!
```


Case 05

Input:

```
n
10
p
i
3
Three
3
p
i
2
Two
2
p
i
3
Wrong
10
p
n
0
p
n
1
i
0
Zero
0
p
i
0
ZeroAgain
0
p
n
0
p
q
```

Output:

```
Command? Entries? Command? []
Command? Index? Name? Height? Command? [("Three",3.000)]
Command? Index? Name? Height? Command? [("Two",2.000),("Three",3.000)]
Command? Index? Name? Height? Entry 3 is already occupied by
("Three",3.000)
Command? [("Two",2.000),("Three",3.000)]
Command? Entries? Command? []
Command? Entries? Command? Index? Name? Height? Height must be larger
than zero
Command? []
Command? Index? Name? Height? Height must be larger than zero
Command? []
Command? Entries? Command? []
Command? Bye!
```


Case 06

Input:

```
h
n
0
h
n
10
h
i
3
Three
3
p
h
i
2
Two
2
h
i
0
AlmostZero
0.1
i
9
Eight
8
h
i
8
Nine
9
h
i
7
S7ven
7
h
i
5
Five
5
i
4
Four
5
p
h
n
0
q
```


Output:

```
Command? Min: 0.000
Max: 0.000
Range: 0.000
Command? Entries? Command? Min: 0.000
Max: 0.000
Range: 0.000
Command? Entries? Command? Min: 0.000
Max: 0.000
Range: 0.000
Command? Index? Name? Height? Command? [("Three",3.000)]
Command? Min: 3.000
Max: 3.000
Range: 0.000
Command? Index? Name? Height? Command? Min: 2.000
Max: 3.000
Range: 1.000
Command? Index? Name? Height? Command? Index? Name? Height? Command?
Min: 0.100
Max: 8.000
Range: 7.900
Command? Index? Name? Height? Command? Min: 0.100
Max: 9.000
Range: 8.900
Command? Index? Name? Height? Command? Min: 0.100
Max: 9.000
Range: 8.900
Command? Index? Name? Height? Command? Index? Name? Height? Command?
[("AlmostZero",0.100),("Two",2.000),("Three",3.000),("Four",5.000),("F
ive",5.000),("S7ven",7.000),("Nine",9.000),("Eight",8.000)]
Command? Min: 0.100
Max: 9.000
Range: 8.900
Command? Entries? Command? Bye!
```


Case 07

Input:

```
n
10
i
3
Three
3
p
i
2
Two
2
i
0
Zero
0.1
i
9
Eight
8
i
8
Nine
9
i
7
S7ven
7
i
5
Five
5
i
4
Four
5
p
h
r
3
NewThree
3
p
h
r
3
NewerThree
33
p
h
r
0
First
100
p
r
9
Last
99
p
h
n
1
```

Output:

```
Command? Entries? Command? Index? Name? Height? Command?
[("Three",3.000)]
Command? Index? Name? Height? Command? Index? Name? Height? Command?
Index? Name? Height? Command? Index? Name? Height? Command? Index?
Name? Height? Command? Index? Name? Height? Command? Index? Name?
Height? Command?
[("Zero",0.100),("Two",2.000),("Three",3.000),("Four",5.000),("Five",5
.000),("S7ven",7.000),("Nine",9.000),("Eight",8.000)]
Command? Min: 0.100
Max: 9.000
Range: 8.900
Command? Index? Name? Height? Command?
[("Zero",0.100),("Two",2.000),("NewThree",3.000),("Four",5.000),("Five
",5.000),("S7ven",7.000),("Nine",9.000),("Eight",8.000)]
Command? Min: 0.100
Max: 9.000
Range: 8.900
Command? Index? Name? Height? Command?
[("Zero",0.100),("Two",2.000),("NewerThree",33.000),("Four",5.000),("F
ive",5.000),("S7ven",7.000),("Nine",9.000),("Eight",8.000)]
Command? Min: 0.100
Max: 33.000
Range: 32.900
Command? Index? Name? Height? Command?
[("First",100.000),("Two",2.000),("NewerThree",33.000),("Four",5.000),
("Five",5.000),("S7ven",7.000),("Nine",9.000),("Eight",8.000)]
Command? Index? Name? Height? Command?
[("First",100.000),("Two",2.000),("NewerThree",33.000),("Four",5.000),
("Five",5.000),("S7ven",7.000),("Nine",9.000),("Last",99.000)]
Command? Min: 2.000
Max: 100.000
Range: 98.000
Command? Entries? Command? Index? Name? Height? Command?
[("Uno",1111.000)]
Command? Index? Name? Height? Command? [("Due",2222.000)]
Command? Min: 2222.000
Max: 2222.000
Range: 0.000
Command? Entries? Command? Bye!
```


Case 08

Input:

```
p
b
p
n
1
i
0
Three
3
p
b
p
n
2
i
0
Three
3
i
1
Two
2
p
b
p
n
2
i
1
Three
3
i
0
Two
2
p
b
p
n
0
q
```


Output:

```
Command? []
Command? Swapped 0 times
Command? []
Command? Entries? Command? Index? Name? Height? Command?
[("Three",3.000)]
Command? Swapped 0 times
Command? [("Three",3.000)]
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? [("Three",3.000),("Two",2.000)]
Command? after swapping: [("Two",2.000),("Three",3.000)]
Swapped 1 times
Command? [("Two",2.000),("Three",3.000)]
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? [("Two",2.000),("Three",3.000)]
Command? Swapped 0 times
Command? [("Two",2.000),("Three",3.000)]
Command? Entries? Command? Bye!
```


Case 09

Input:

```
n
5
i
0
One
1
i
1
One
1
i
2
Two
2
i
3
One
2
i
4
Four
0.4
i
3
Five
0.5
p
b
p
n
0
q
```


Output:

```
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? Index? Name? Height? Command? Index? Name? Height?
Command? Index? Name? Height? Command? Index? Name? Height? Entry 3
is already occupied by ("One",2.000)
Command?
[("One",1.000),("One",1.000),("Two",2.000),("One",2.000),("Four",0.400
)]
Command? after swapping:
[("One",1.000),("One",1.000),("One",2.000),("Two",2.000),("Four",0.400
)]
after swapping:
[("One",1.000),("One",1.000),("One",2.000),("Four",0.400),("Two",2.000
)]
after swapping:
[("One",1.000),("One",1.000),("Four",0.400),("One",2.000),("Two",2.000
)]
after swapping:
[("One",1.000),("Four",0.400),("One",1.000),("One",2.000),("Two",2.000
)]
after swapping:
[("Four",0.400),("One",1.000),("One",1.000),("One",2.000),("Two",2.000
)]
Swapped 5 times
Command?
[("Four",0.400),("One",1.000),("One",1.000),("One",2.000),("Two",2.000
)]
Command? Entries? Command? Bye!
```


Case 10

Input:

```
n
6
i
0
Hello
20
i
1
Hello
10
i
2
Aargh
10
i
3
Aargh
10
i
4
Aaah!
3
i
5
Aaah!
1
p
b
p
n
0
n
1
i
0
test
0.1
p
b
p
n
0
q
```


Output:

```
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? Index? Name? Height? Command? Index? Name? Height?
Command? Index? Name? Height? Command? Index? Name? Height? Command?
[("Hello",20.000),("Hello",10.000),("Aargh",10.000),("Aargh",10.000),(
"Aaah!",3.000),("Aaah!",1.000)]
Command? after swapping:
[("Hello",10.000),("Hello",20.000),("Aargh",10.000),("Aargh",10.000),(
"Aaah!",3.000),("Aaah!",1.000)]
after swapping:
[("Hello",10.000),("Aargh",10.000),("Hello",20.000),("Aargh",10.000),(
"Aaah!",3.000),("Aaah!",1.000)]
after swapping:
[("Hello",10.000),("Aargh",10.000),("Aargh",10.000),("Hello",20.000),(
"Aaah!",3.000),("Aaah!",1.000)]
after swapping:
[("Hello",10.000),("Aargh",10.000),("Aargh",10.000),("Aaah!",3.000),("
Hello",20.000),("Aaah!",1.000)]
after swapping:
[("Hello",10.000),("Aargh",10.000),("Aargh",10.000),("Aaah!",3.000),("
Aaah!",1.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Hello",10.000),("Aargh",10.000),("Aaah!",3.000),("
Aaah!",1.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Aargh",10.000),("Hello",10.000),("Aaah!",3.000),("
Aaah!",1.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Aargh",10.000),("Aaah!",3.000),("Hello",10.000),("
Aaah!",1.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Aargh",10.000),("Aaah!",3.000),("Aaah!",1.000),("H
ello",10.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Aaah!",3.000),("Aargh",10.000),("Aaah!",1.000),("H
ello",10.000),("Hello",20.000)]
after swapping:
[("Aargh",10.000),("Aaah!",3.000),("Aaah!",1.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
after swapping:
[("Aaah!",3.000),("Aargh",10.000),("Aaah!",1.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
after swapping:
[("Aaah!",3.000),("Aaah!",1.000),("Aargh",10.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
after swapping:
[("Aaah!",1.000),("Aaah!",3.000),("Aargh",10.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
Swapped 14 times
Command?
[("Aaah!",1.000),("Aaah!",3.000),("Aargh",10.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
Command? Entries? Command? Entries? Command? Index? Name? Height?
Command? [("test",0.100)]
Command? Swapped 0 times
Command? [("test",0.100)]
Command? Entries? Command? Bye!
```


Case 11

Input:

```
n
5
i
0
One
1
i
1
One
1
i
2
Two
2
i
3
One
2
i
4
Four
0.4
i
3
Five
0.5
p
m
p
n
0
q
```

Output:

```
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? Index? Name? Height? Command? Index? Name? Height?
Command? Index? Name? Height? Command? Index? Name? Height? Entry 3
is already occupied by ("One",2.000)
Command?
[("One",1.000),("One",1.000),("Two",2.000),("One",2.000),("Four",0.400
)]
Command? Command?
[("Four",0.400),("One",1.000),("One",1.000),("One",2.000),("Two",2.000
)]
Command? Entries? Command? Bye!
```


Case 12

Input:

```
n
6
i
0
Hello
20
i
1
Hello
10
i
2
Aargh
10
i
3
Aargh
10
i
4
Aaah!
3
i
5
Aaah!
1
p
m
p
n
0
n
1
i
0
test
0.1
p
m
p
n
0
q
```


Output:

```
Command? Entries? Command? Index? Name? Height? Command? Index? Name?
Height? Command? Index? Name? Height? Command? Index? Name? Height?
Command? Index? Name? Height? Command? Index? Name? Height? Command?
[("Hello",20.000),("Hello",10.000),("Aargh",10.000),("Aargh",10.000),(
"Aaah!",3.000),("Aaah!",1.000)]
Command? Command?
[("Aaah!",1.000),("Aaah!",3.000),("Aargh",10.000),("Aargh",10.000),("H
ello",10.000),("Hello",20.000)]
Command? Entries? Command? Entries? Command? Index? Name? Height?
Command? [("test",0.100)]
Command? Command? [("test",0.100)]
Command? Entries? Command? Bye!
```

