built on 2017/02/12 at 09:33:23 ***due:*** *Wed, Feb 22 @ 11:59pm*

Be sure to read this problem set thoroughly, especially the sections related to collaboration and the hand-in procedure.

Collaboration

We interpret collaboration very liberally. You may work with other students. However, each student ***must*** write up and hand in his or her assignment separately. Let us repeat: You need to write your own code. You must not look at or copy someone else's code. You need to write up answers to written problems individually. The fact that you can recreate the solution from memory will be taken as proof that you actually understood it, and you may actually be interviewed about your answers.

Be sure to indicate who you have worked with (refer to the hand-in instructions).

Hand-in Instructions

To submit this assignment, please follow the steps below:

1. Make sure your programs compile and run on Hamachi.
2. Check for any memory leak e.g. `valgrind --tool=memcheck --leak-check=full ./box < test.in`
3. Zip up all `.c` files and name the zip file `a3.zip` i.e.

```
> zip a3.zip box.c queue.c btree.c mergesort.c
```

4. Find out the MD5 hash of your zip file. You will need to submit this code on Canvas. We use it for keeping track of your submission time. You may resubmit your work but the MD5 hash has to match.

```
> md5sum a3.zip
```

5. Copy the zip file to the directory `/subm/u5712345` where `u5712345` is your student ID.

```
> cp a3.zip /subm/u5712345
```

6. Log on to Canvas, go to assignment 3 submission page and enter the MD5 hash.

Task 1: Magic Box (10 points)

In this problem, you will work on a data structure called (Magic) Box. This Box is magical because it can hold as many items as you want! (well, not really but as long as memory permits)

Your Task: Complete the implement of the following functions in `box.c`,

- `void createBox(Box **b, int init_cap)` – Creates a new box with capacity of `init_cap`. If the box `b` has already been allocated, this function does nothing.
- `void insert(Box *b, int elem)` – Inserts an element into a given box. If the box is full, you must fist increase its capacity by **doubling the capacity** with `realloc` and then add the element.
- `void removeAll(Box *b, int elem)` – Removes all occurances of an element from a box. After removal, the elements in the box must be condensed i.e. no gap between any two elements. Also, this function must not change the capacity of the box.

- `void printBox(Box *b)` – Prints out elements of a box, one element per line.
- `double getMean(Box *b)` – Returns the mean of data in a box.
- `void dealloc(Box **b)` – Deallocates the box along with its data.

Expected Behavior: After you have finish implementing `box.c`, you can check your own work by running:

```
> ./box < test.in
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) and the `main` function will be ignored in the grading environment. So you should not need to and must not modify the header files and the `main` function! Your program should not leak any memory.

Task 2: Queue of Words (10 points)

Queue (or a FIFO) is another basic data structure that is very common in low-level system implementations i.e. system libraries and OS kernels. You will work on a simple implementation of a dynamic queue where each element is a string. The underlying implmentation of your queue will be similar to a listed list.

Your Task: Complete the implement of the following functions in `queue.c`,

- `void push(Queue **q, char *word)` – Pushes a string `word` to the back of a queue `q`. Instead of keeping the pointer to the array, you must instead keep a **COPY** of the word inside the queue. Also, if `q` hasn't been allocated i.e. `q` is NULL, you must allocate space for it here as well.
- `char *pop(Queue *q)` – Returns the pointer to the string at the front of the queue and remove it from the queue. The caller of `pop` is responsible for freeing the returned pointer. When `q` is empty, this function returns NULL.
- `int isEmpty(Queue *q)` – Returen `true` if the queue is empty and `false` otherwise.
- `void print(Queue *q)` – Prints out the elements of a give queue `q`, front to back. If `q` is empty, prints out `No items`.
- `void delete(Queue *q)` – Deallocates the queue as well as all items in it.

Expected Behavior: After you have finish implementing `queue.c`, you can check your own work by running:

```
> ./queue
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) and the `main` function will be ignored in the grading environment. So you should not need to and must not modify the header files and the `main` function! Your program should not leak any memory.

Task 3: B(e)ST tree (10 points)

Next data structure that you will work on is Binary Search Tree (BST). A binary search tree is a dynamic data structure that organizes its internal structure as a tree. In a good scenario, you can insert, remove, and then look up for item very efficiently. You can read more about it here: `http://algs4.cs.princeton.edu/32bst/`

Your Task: Complete the implement of the following functions in `btree.c`,

- `void insert(Node **tree, int val)` – Inserts a value into the tree rooted at a node `tree`. If `tree` has not been allocated i.e. is NULL, it is allocated here and `tree` becomes a new root node.
- `void delete(Node *tree)` – Deallocates ALL nodes under a give node `tree`. If `tree` is a root node, the entire tree is deallocated. If `tree` is not a root node, the subtree rooted at `tree` is deallocated.
- `Node *lookup(Node **tree, int val)` – Returns the pointer to the node containing `val` or, NULL if the given value is not found.
- `void print(Node *tree)` – Prints out the tree in ascii format. You must print out the tree using the in-order traversal i.e. root-left-right. White-spaces and `|-` symbols are used to indicate level of the tree and parent-child relationship. For example, a tree with 5 nodes where the root node has value of 6 with 2 children: 2, 9 as left child and right child respectively. The node 2 also has 2 children of its own: 1 and 4. The tree can be printed as follows:

```
6
|-2
  |-1
  |-4
|-9
```

Expected Behavior: After you have finish implementing `btree.c`, you can check your own work by running:

```
> ./btree
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) and the `main` function will be ignored in the grading environment. So you should not need to and must not modify the header files and the `main` function! Your program should not leak any memory.

Task 4: Mergesort (10 points)

In this task, you will be implementing your beloved sorting algoithm *Merge Sort*, but, this time, it must be in C. To keep this things simple, we have provided you with the pseudo code; you only need to translate it to C.

Let `Entry` be a C-struct with the following fields: data (`int`), name (`char *`). We have already defined them for you in `mergesort.h`. Entries are sorted by using `data` as the key. There are 3 functions that you have to complete: `merge`, `mergesort` and `main`.

- `merge` – Merges two sorted arrays into a single array

```
// L and R are array of Entry structures.
// nL and nL are the number of items in L and R respectively.
void merge(Entry *temp, Entry *L, int nL, Entry *R, nR) {
  Let i,j,k be 0
  While (k is smaller than nL + nR) {
    if ((L[i] is valid and L[i] is smaller than R[j]) or R[j] is invalid) {
      Set temp[k] to L[i]
      Increase i by 1
    } else {
      Set temp[k] to R[j]
      Increase j by 1
    }
    Increase k by 1
  }
}
```

- `mergesort` – Partition.. Sort.. Merge

```
// entries is an array of Entry structures.
// n is the number of items in entries.
void mergesort(Entry *entries, int n) {
  if (n > 1) {
    Split the array entries into 2 arrays: L and R
    Create a temp array that can hold n entries
    Call mergesort() on L
    Call mergesort() on R
    Call merge() with temp, L and R along with the size of L and R
    Copy over sorted items from temp to entries
    Free the temp array
  }
}
```

- `main` – The "main" function

```
main() {
         Read the number of items from the STDIN.
        Allocate an array for the input.
        Read data from STDIN and store them the array.
        Call mergesort() on the array.
        Print out the entries.
        Free all the memory that you have allocated.
}
```

Input & Output Your program should read input from the **standard input**. The format of the input is as follows:

```
<number of items>
<data_1> <name_1>
<data_2> <name_2>
...
<data_n> <name_n>
```

where `data_i` is an `int` and `name_i` is a string of length strictly smaller than `MAX_NAME_LENGTH` which is defined in `mergesort.h`. Here is an example input:

```
4
4 aa
7 bb
2 cc
1 dd
```

After you are done, you can check your own work by running:

```
> ./mergesort < test.in
```

and compare the output with the provided expected output in the starter code.

Important: Any modification made to the: header files (.h files) will be ignored in the grading environment. So you should not need to and must not modify the header files! For this problem, the `main` function will not be replaced. Your program should not leak any memory.