Assignment 1

Introduction to Asymptotic analysis

Welcome to your course on Design and Analysis of Algorithms. We hope you will enjoy this course. The first assignment is on asymptotic analysis. As you should know by now, Asymptotic analysis is the study of the number of critical operations[1] in an algorithm as the input size grows very large.

Your first assignment will be a practical demonstration of the asymptotic behaviors of operations on different data structures. You will be required to implement the basic operations of a Max-Heap and a Binary Search tree, 2 structures that you should be familiar with. More on the specifics in the following sections.

Please note to follow the rules, failure to comply with any of the rules will lead to the student losing all the marks for the assignment.

Problem Statement 1

You are given a header file that contains the functions for some operations on a Max-heap. You are expected to implement these functions to carry out the expected operations. The given functions should also store the **number of operations** carried out, in the location pointed to by count_ptr. For the Heap, you are expected to use ***iterative*** method, no recursive functions.

Max-heap data structure has the following operations defined on it:

- Insert:- Inserts a given key into a heap while ensuring the heap property still holds after insertion
- Extract_max:- Finds max element, removes it from the heap, and returns the maximum value to the client, returns -1 if the heap is empty
- Find max:- Finds max element and returns it, DOES NOT delete the element, returns -1 if the heap is empty
- Find min:- Finds the minimum element in the heap, returns -1 if the heap is empty
- Search:- Returns the key if it is present in the heap, returns -1 if the element is not found

In the header file given to you, you will find these function with the following signatures respectively:

```
// Insert element "key" to heap "heap"
// and store the number of operations made in the
// location pointed to by count_ptr.
void insert(heap_t *heap, int key, int *count_ptr);

// *Removes and Returns* the maximum element in the heap
// Returns -1 if the element is not present
int extract_max(heap_t *heap, int *count_ptr);

// Searches for the element key in the heap
// Returns the element if found, else -1
int search(const heap_t *heap, int key, int *count_ptr);

// Returns the maximum value of the element in the heap
// and store the number of key comparisons made in the
// location pointed to by count_ptr.
// Returns -1 if the heap is empty
int find_max(const heap_t *heap, int *count_ptr);

// Returns the minimum value in the heap, -1 if the heap is empty
int find_min(const heap_t *heap, int *count_ptr);
```

You will also need to implement some simple initialization and de-initialization code to create the data structures. Details about these functions and the structures can be found in the file `heap.h`. The file `<SRN>_heap.c` contains the skeletons for all the functions you will need to implement. Implementation details are also mentioned there.

You will finish the implementations for these functions and use `heap_client.c` to run a few sample test cases and print some stats. It prints the number of comparisons for various inputs in the form “n,c” where n is the input size and c is the number of comparisons.

__Note__: The test cases are not exhaustive in any way. You are supposed to test your functions more rigorously.

You will run the programs as:

```
$ gcc -c <SRN>_heap.c # <SRN> would be replaced with your SRN
$ gcc -c heap_client.c
$ gcc <SRN>_heap.o heap_client.o
```

```
$ ./a.out # This would be to test your functions
```

Input Constraints:

0 <= max_size of heap < 10000
0 <= key value in the heap < 20000

Marking Scheme:

This problem statement will be graded for 15 marks. The distribution is as follows:

- 5 marks: Correct implementation of the data structure
- 5 marks: Correct counting for all operations
- 5 marks: Following all programming guidelines mentioned below in the rules section

Problem Statement 2

You are given a header file that contains the functions for operations on a binary search tree(BST). You are expected to implement these functions to carry out the given expected operations. The given functions should also store the **number of operations** carried out in the location pointed to by count_ptr. For the BST, you are expected to use ***recursive*** functions.

All implementations of the operations on the BST must be ***recursive*** in nature. This should help you understand how to implement counting in the case of recursive functions.

Binary Search Trees have the following operations defined on them:

- Insert: Inserts a key into the binary search tree.
- Delete Deletes a key from the binary search tree.
- Search: Searches for the node with a key in the tree, if found, it returns the key, else it returns -1.
- Find_max: Returns the maximum element in the tree. You may assume the maximum element of an empty tree is -1.

In the implementation file given to you, you will find these functions modified into the following signatures respectively:

```
// Inserts element key into the Binary search tree
```

```
void insert(bst_t *tree, int key, int *count_ptr);

// Delete key from the BST
// Replaces node with in-order successor
void delete_element(bst_t *tree, int key, int *count_ptr);

// Searches for the element key in the bst
// Returns the element if found, else -1
int search(const bst_t *tree, int key, int *count_ptr);

// Returns the maximum element in the BST
// Returns -1 if the tree is empty
int find_max(const bst_t *tree, int *count_ptr);
```

You will also need to implement some simple initialization and de-initialization code to create the data structures. Details about these functions and the structures can be found in the file `bst.h`. The file `<SRN>_bst.c` contains the skeletons for all the functions you will need to implement. Implementation details are also mentioned there.

You will finish the implementations for these functions and use `bst_client.c` to run a few sample test cases and print some stats. It prints the number of comparisons for various inputs in the form “n,c” where n is the input size and c is the number of comparisons.

Note: The test cases are not exhaustive in any way. You are supposed to test your functions more rigorously.

You will run the programs as:

```
$ gcc -c <SRN>_bst.c # <SRN> would be replaced with your SRN
$ gcc -c bst_client.c
$ gcc <SRN>_bst.o bst_client.o
$ ./a.out # This would be to test your functions
```

__Note__: You may make the functions declared in the header file as wrapper functions that call a recursive function[2].

Input Constraints:

0 <= key queried on the tree < 20000
0 <= maximum number of elements in the tree <=10000

Marking Scheme:

This problem statement will be graded for 15 marks. The distribution is as follows:

- 5 marks: Correct implementation of the data structure
- 5 marks: Correct counting for all operations
- 5 marks: Following all programming guidelines mentioned below

An example: A sorted array

As an example, we have implemented all the operations mentioned above for a sorted array. This program is what we want you to emulate for the other data structures. You may use this sample program to guide you on the different ways you may implement your programs. You will find `array.c` containing our implementation.

Rules

1. **Plagiarism Policy**
   You will lose all marks for the assignment if the code is found to be plagiarized.

2. Do ***not*** modify the given header file, you may however write your own client file to try out various test cases.

3. Programming guidelines:
   - Do *not* use global variables
   - Do *not* use static variables
   - Do *not* use any libraries other than stdlib.h
   - Do *not* use goto statements
   - Do *not* use break statements unless in a switch-case clause

- Do *not* use continue statements
- If you define any helper functions that are not in the header, mark the functions as static
- Document your code with helpful comments

4. Do *not* print anything in your functions, you will be awarded 0 in this case

5. Follow the function signatures given in the header file. Do *not* modify the function signatures, you will be awarded 0 in this case

6. Follow the naming convention for the submissions correctly. We will not allow for re-submission, be careful. FAILING TO FOLLOW THE NAMING CONVENTION WILL LEAD TO 0 MARKS

Submission

The deadline for submission is **11:59 PM 26th February 2022**. You will not be granted any extension for the deadline.

You will need to submit 2 files, one for each problem statement.

- For problem statement 1, the submission shall be the C file containing the implementation of problem statement 1(Max Heap). It must be named `<SRN>_heap.c`, with the student's SRN. Example: `PES1UG20CS000_heap.c`

- For problem statement 2, the submission shall be the C file containing the implementation of problem statement 2(Binary Search Tree). It must be named `<SRN>_bst.c`, with the student's SRN. Example: `PES1UG20CS000_bst.c`

- Ensure the SRN is capitalized in both files

- Add your name as a comment on the first line of your program

- Add your SRN as a comment on the second line of your program

You will be awarded marks based on the correctness of all your programs and you will lose marks if you have not followed any of the above rules. We will be checking all of the above criteria. You will lose all your marks if the naming conventions are not followed.

Please submit your files to PESU Academy.

Resources

- Drive Folder containing all the aforementioned files.
- A Doubt sheet to post queries regarding the assignment.
- Heaps: https://en.wikipedia.org/wiki/Binary_heap
- Binary Search Trees: https://en.wikipedia.org/wiki/Binary_search_tree

Footnotes

[1] The term “critical operations” is purposefully ambiguous. You will need to decide on the complexity of certain simple operations to decide a meaningful lower bound or upper bound for an algorithm. Any bound is meaningless without knowing what it is a bound of.

[2] Refer to the recursive binary search in the sample problem. Ensure to mark these functions static.