NATIONAL INSTITUTE OF TECHNOLOGY SIKKIM
Department of Computer Science and Engineering

CS13201 Data Structure and Algorithms Laboratory

Odd Semester, July–December 2024

Laboratory Assignment #5

NB: Solve the following programming problems according to instructions given during the laboratory sessions. Along with the requirements mentioned in the text of the questions, additional instructions may be given by the course instructor to produce the desired output. All programs should be written in the C programming language. Do not use any external variable unless allowed by the instructor. Do not use any header other than *stdio.h* and *stdlib.h*.

1. Write a program to take a list of real numbers, *A*, as input from the user, and then sort *A* by bubblesort. After sorting *A*, take a value *key* as input from the user and search for *key* in *A* by binary search. Write two functions called **bubblesort** and **binary_search** to implement the sorting and search operations respectively. Print the result of the search, i.e. whether *key* is "found" or "not found" in *A*, as the final output of your program. Write both a non-recursive version as well as a recursive version of the **binary_search** function, and call both functions to verify their correctness.

2. Write a program to take a list of real numbers as input from the user as command-line arguments (where each number is a command-line argument), and then sort the list by insertion sort. Write a function called **insertion_sort** to implement the sorting operation. Print the sorted list as the final output of your program. The sample execution given below is self-explanatory.

   Sample Execution

```
$ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
7 11.22 14.13 15.98 17.40 25 50
```

3. Write a program to take a list of real numbers as input from the user as command-line arguments (where each number is a command-line argument), and then sort the list by selection sort. Write a function called **selection_sort** to implement the sorting operation. Print the sorted list as the final output of your program. The sample execution given below is self-explanatory.

   Sample Execution

```
$ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
7 11.22 14.13 15.98 17.40 25 50
```

4. Write a program to take a list of real numbers as input from the user as command-line arguments (where each number is a command-line argument), and then sort the list by quicksort. Write a function called **quicksort** to implement the sorting operation. Print the sorted list as the final output of your program. The sample execution given below is self-explanatory.

   Sample Execution

```
$ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
7 11.22 14.13 15.98 17.40 25 50
```

5. Write a program to take a list of real numbers as input from the user as command-line arguments (where each number is a command-line argument), and then sort the list by merge sort. Write a function called **merge_sort** to implement the sorting operation. Print the sorted list as the final output of your program. The sample execution given below is self-explanatory.

   Sample Execution

```
$ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
```

```
7 11.22 14.13 15.98 17.40 25 50
```

6. Write a program to create a binary search tree by taking the keys of the tree as input from the user. Initially take a series of *n* values as input from the user and create a binary search tree *T* by inserting into *T* each of the entered values as a key of *T*, with the **Insert** function defined below. Display the keys of *T* by the following three types of traversal as the final output of your program: i) Inorder tree walk, ii) Preorder tree walk, and iii) Postorder tree walk. You need to implement the following functions (abstract definitions given below) in your program and then call those functions in your **main** function to satisfy the requirements of this problem.

   Insert(*T*, *x*) := inserts node *x* into the binary search tree *T* (so that the binary-search-tree property is maintained for *T*).

   Inorder-tree-walk(*x*) := Prints the keys of the tree rooted at *x* by an inorder tree walk.

   Preorder-tree-walk(*x*) := Prints the keys of the tree rooted at *x* by a preorder tree walk.

   Postorder-tree-walk(*x*) := Prints the keys of the tree rooted at *x* by a postorder tree walk.

7. Write a program to create a binary search tree and implement the functions defined below (abstract definitions given) on the tree. Create a menu-based interface which will ask the user to select an option and then display the required output; the menu will be displayed repeatedly until the user chooses to quit, and will show the user the following options: i) Create binary search tree, ii) Delete tree, iii) Insert a node, iv) Search a key, v) Delete a node, vi) Tree Minimum, vii) Tree Maximum, viii) Get height, ix) Count nodes, x) Count leaves, xi) Display in inorder, xii) Display in preorder, xiii) Display in postorder, xiv) Display leaves, xv) Display internal nodes. The aforementioned menu options respectively correspond to the functions defined below, where *list* is an array of *n* double-precision real numbers (taken as input from the user), *T* is a binary search tree, *n* is a natural number, *x* is a pointer to a node in a binary tree, and *k* is a key value that may be stored in *T*.

   i) **createBST**(*list*, *n*) := Creates a binary search tree *T* with an *n*-element array *list*, reading *list* from *list*[1] to *list*[*n*], by calling the **insert** function repeatedly, starting with **insert**(*T*, x_1) and ending with **insert**(*T*, x_n), where x_1 is a node with *list*[1] as its key, and x_n is a node with *list*[*n*] as its key. The function then returns *T* (or the root of *T*).

   ii) **deleteTree**(*T*) := Deletes the whole tree *T*, freeing the storage held by every node of *T*.

   iii) **insert**(*T*, *x*) := Inserts node *x* into the binary search tree *T* (maintaining the binary-search-tree property for *T*).

   iv) **search**(*T*, *k*) := Searches for the key *k* in *T*, and returns “true” if *k* is present in *T*, and returns “false” otherwise.

   v) **deleteNode**(*T*, *k*) := Deletes a node *x* from *T* where *x.key* = *k* if such a node *x* exists in *T* (freeing the storage held by *x*).

   vi) **treeMinimum**(*T*) := Returns the node with the smallest key in *T*.

   vii) **treeMaximum**(*T*) := Returns the node with the largest key in *T*.

   viii) **getHeight**(*T*) := Returns the height of *T*.

   ix) **getNodeCount**(*T*) := Returns the number of nodes in *T*.

   x) **getLeafCount**(*T*) := Returns the number of leaves in *T*.

   xi) **inorder**(*T*) := Prints the keys of *T* by an inorder tree walk of *T*.

   xii) **preorder**(*T*) := Prints the keys of *T* by a preorder tree walk of *T*.

   xiii) **postorder**(*T*) := Prints the keys of *T* by a postorder tree walk of *T*.

   xiv) **printLeaves**(*T*) := Prints the keys of the leaves of *T*.

   xv) **printInternalNodes**(*T*) := Prints the keys of the internal nodes (non-leaf nodes) of *T*.

8. For the program of Question no. 6, write three versions for each of the three functions **Inorder-tree-walk**, **Preorder-tree-walk**, **Postorder-tree-walk**: i) a recursive version, ii) a non-recursive version using a stack (and without using a parent pointer for each node), iii) a non-recursive version without using a stack. Create a binary search tree as done for Question no. 6, and then print the tree with all the aforementioned functions as the final output of your program to verify the correctness of these functions defined by you.

9. Write a program to take a sentence as input from the command line, where each word of the sentence is a command-line argument, and then create a binary search tree by storing each character of the sentence in a node of the tree (excluding blank spaces, but including other special characters like punctuation symbols), following the binary search tree property where the characters are related to each other with respect to their ASCII values, i.e. every ordered pair of characters has an "is less than", "is greater than", or "is equal to" relation according to their equivalent integer values. The characters are to be inserted into the binary search tree by reading the sentence from left to right. For example, if the given character sequence is "bat", then 'b' will be stored as the root, 'a' will be stored as the left child of 'b', and 't' will be stored as the right child of 'b' (since, 'a' < 'b' < 't'). Print the keys of the binary tree by preorder tree walk as the final output of your program. The sample execution given below is self-explanatory.

   Sample Execution

```
$ ./mybst.out go, and play ↵
g , a a d o n l p y
```

10. Modify the program written for Question no. 9 so that all the vowels of the binary tree are removed, i.e. after constructing a binary search tree with characters taken from the command-line, as described in the previous question, delete from the binary tree all nodes containing vowels. Print the binary tree by preorder tree walk before and after making the modification as the final output of your program. The sample execution given below is self-explanatory.

    Sample Execution

```
$ ./mybst.out go, and play ↵
Binary tree by preorder tree walk before modification:
g , a a d o n l p y
Binary tree by preorder tree walk after modification:
g , d p n l y
```

11. Write a program to test whether a given array of numbers is a binary heap, and show what kind of binary heap it is if it is a binary heap. Take an array of real numbers *A* as input from the user, and print whether *A* is a max-heap, *A* is a min-heap, or *A* is NOT a binary heap. The sample final output given below is self-explanatory.

    Sample Final Output 1

    Input array A: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
    A is NOT a binary heap.

    Sample Final Output 2

    Input array A: 16, 14, 10, 8, 7, 9, 3, 2, 4, 1
    A is a max-heap.

    Sample Final Output 3

    Input array A: 2, 4, 7, 14, 5, 25, 10, 15, 17, 13
    A is a min-heap.

12. Write a program to build a max-heap out of the elements of a given array of numbers with a function called **buildMaxHeap**. **buildMaxHeap** will take an array of real numbers *A* as a parameter and then convert *A* into a max-heap. Print, as the final output of your program, the elements of *A* both before and after *A*

becomes a heap. [NB: You may need to define a function called **maxHeapify** before defining **buildMaxHeap**, as described in the textbook.]

Sample Final Output

Original array: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7
Modified array representing a max-heap: 16, 14, 10, 8, 7, 9, 3, 2, 4, 1

13. Write a program to build a min-heap out of the elements of a given array of numbers with a function called **buildMinHeap**. **buildMinHeap** will take an array of real numbers *A* as a parameter and then convert *A* into a min-heap. Print, as the final output of your program, the elements of *A* both before and after *A* becomes a heap. [NB: You may need to define a function called **minHeapify** before defining **buildMinHeap**, similarly as in Question no. 12 above.]

    Sample Final Output

    Original array: 17, 4, 25, 14, 13, 7, 10, 15, 2, 5
    Modified array representing a min-heap: 2, 4, 7, 14, 5, 25, 10, 15, 17, 13

14. Write a program to take an array of real numbers *A* as input from the user as command-line arguments (where each number is a command-line argument), and then sort *A* in ascending order by heapsort. Write a function called **heapsort** to implement the sorting operation. Print *A* after sorting as the final output of your program. The sample execution given below is self-explanatory. [NB: Use the functions **buildMaxHeap** and **maxHeapify** from Question no. 12 to define **heapsort**, as described in the textbook.]

    Sample Execution

    ```
    $ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
    7 11.22 14.13 15.98 17.40 25 50
    ```

15. Write a program to take an array of real numbers *A* as input from the user as command-line arguments (where each number is a command-line argument), and then sort *A* in descending order by heapsort. Write a function called **heapsort** to implement the sorting operation. Print *A* after sorting as the final output of your program. The sample execution given below is self-explanatory. [NB: Use the functions **buildMinHeap** and **minHeapify** from Question no. 13 to define **heapsort**, as described in the textbook.]

    Sample Execution

    ```
    $ ./sort.out 15.98 14.13 50 17.40 25 7 11.22 ↵
    7 11.22 14.13 15.98 17.40 25 50
    ```

---------------------------------------x---------------------------------------