INDIAN STATISTICAL INSTITUTE

MTech(CS) I year 2025-2026

Subject: Computing Laboratory

Lab Test 3 (November 10, 2025)

Total: 60 marks Duration: 4 hours

GENERAL INSTRUCTIONS

1. All programs should take the required input via the standard input (terminal/keyboard), and print the desired output to the terminal.
2. Please make sure that your programs adhere strictly to the specified input and output format. Do not print extra strings asking the user for input, debugging messages, etc. These will cause the automatic checking system to fail.
3. Please make sure that the programs are free from memory errors and leaks, you will lose marks if they are not.

For all problems on binary search trees, you may assume that the nodes store 32-bit integers. You may use the type `int32_t` and the corresponding limits `INT32_MIN` and `INT32_MAX` (defined in `stdint.h`) if necessary.

Q1. (10 marks) Implement the following functions for a binary search tree (BST).

(a) `int min_node(TREE *t, int *index)`

(b) `int max_node(TREE *t, int *index)`

The functions should *return* the minimum and maximum value stored in the tree, respectively; the index of the node that stores the minimum / maximum value should be saved in `*index`.

For full credit, your program should use $O(h)$ time and $O(1)$ extra space (h denotes the height of the given tree).

Input format: The standard format supported by the `read_tree(TREE *t)` function discussed in class, viz., a non-negative integer n denoting the number of nodes in the tree, followed by n lines (one per node), each having 3 fields: data, left child index, and right child index.

Output format: Two lines in the example format shown below.

```
Minimum value: -20, node index = 6
Maximum value: 132, node index = 16.
```

Q2. (10 marks) Write a program that takes 2 BSTs, T_1 and T_2, and an integer x, and reports all pairs (v_1, v_2) of nodes (with $v_1 \in T_1$ and $v_2 \in T_2$) such that the sum of the values stored in v_1 and v_2 is x.

For full credit, your program should use $O(\max(n_1, n_2) \lg(\max(h_1, h_2))$ time and $O(\min(h_1, h_2))$ extra space (h_1 and h_2 denote the heights of T_1 and T_2).

Input format: Similar to the format described above, except that you will be given **two** trees, one after the other. These two trees will be followed by some values of the target x.

Output format: For each target x given after the two trees, your output should be of the form given below.

```
Target: 112
Index in T1: 6, value = -20; index in T2: 16, value = 132.
Index in T1: 10, value = 0; index in T2: 10, value = 112.
...
```

The targets should be processed in the order in which they are listed in the input file. For a particular target, if there are multiple pairs, the corresponding lines should be printed in increasing order of the **DATA** stored in the node $v_1 \in T_1$.

Q3. (8+12 = 20 marks) Suppose we give you a binary tree and claim that it is a **balanced** binary search tree. Implement the following functions to check whether the given tree satisfies the corresponding property.

(a) `is_BST(TREE *t)` : returns 1 if `t` is a binary search tree (BST), and 0 otherwise. [8]

[You will lose 1-2 marks if your algorithm requires $O(n)$ instead of $O(h)$ extra space (n and h denote the number of nodes in the tree, and its height, resp.)]

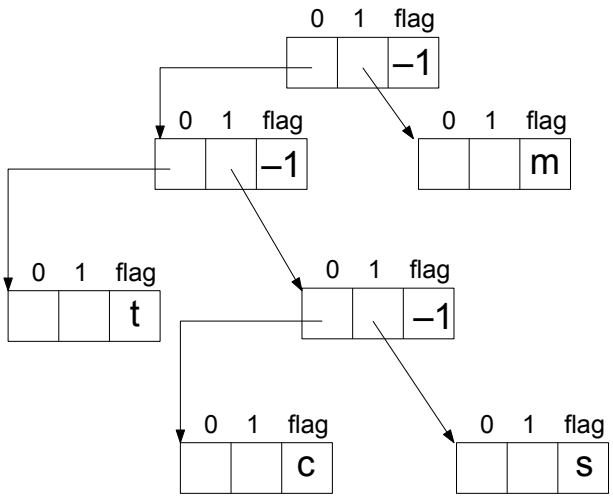
(b) `is_balanced(TREE *t)` : checks whether, at each node, the heights of the left and right sub-trees of that node differ by at most 1. [12]

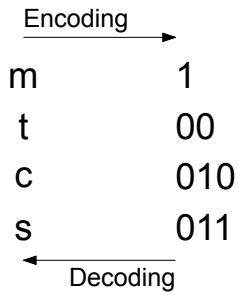
Input format: As for Q1.

Output format: As in the example given below.

```
BST: YES, Balanced: NO.
```

Q4. (12+8 = 20 marks) Consider the following binary trie.





Observe the following properties.

- The underlying alphabet is $\{0, 1\}$. Thus, T stores only binary strings, over $\{0, 1\}$, and each node has at most 2 child pointers (in the example we discussed in class, each node had up to 26 child pointers). Of course, each node also has a third field to store the END-OF-WORD flag (or something similar; see below).

- **Prefix Property:** No word / string stored in the trie is a prefix of any other word / string stored in the trie, i.e., the END-OF-WORD flag only appears at leaf nodes in the trie.
- The END-OF-WORD field either stores -1 (for non-leaf nodes), or **a single ascii character**, for leaf nodes.

  We say that the (binary) string that takes us to a leaf node is an **encoding** for the ascii character stored in that leaf node. Thus, `m` and `t` may be encoded as 1 and 00, respectively. Conversely, the strings 1, 00, 010 and 011 may be decoded as `m`, `t`, `c` and `s`, respectively.

Encoding of strings: Given the above trie, the string `mtt` can be **encoded** as `10000`,

Decoding: Now consider a binary string s. Because of the **Prefix Property** stated above, we can easily decode it in one pass, as follows: start a search for s in the trie as usual. If s is found in the TRIE, then we output the character stored at the leaf node corresponding to s, and we are done. Otherwise, if we reach an END-OF-WORD flag (leaf node), but we have not yet reached the end of s, we output the character stored at the leaf node, and start another fresh search with the remainder of s. If anything else happens, s **cannot** be expressed as a concatenation of the strings stored in T; so it is **not a valid encoding**.

For example: the string 01100100011 can be **uniquely decoded as `stmts`**, while the string 0110010 cannot be decoded properly.

Write functions that take such a trie, along with

(a) a binary string s and either correctly decodes it, or prints `Not a valid encoding` if that cannot be done. [12]

(b) a string consisting only of printable, non-whitespace ascii characters, and prints its binary encoding. If the code for any input letter is not stored in the trie, your program should print `Unexpected input`. [8]

Input format: The first part of the input will consist of a non-negative integer n denoting the number of nodes in the trie, followed by n lines (one per trie node), each having 3 fields: left child index, right child index, and an END-OF-WORD flag whose value is either -1, or a single printable ascii character (other than digits and whitespace).

The trie will be followed by some strings, one per line. The number of input strings will not be given to you in advance, but each string will contain no more than 200 characters. Encoded and decoded strings are over disjoint alphabets; thus, your program should be able to determine the type of input and the desired output, simply by looking at the first character of each input string.

Output format: For each input string, your program should print, on a separate line, the output (or an error message) obtained by encoding / decoding the input string.

Sample input:

⟨ representation of the trie used in the example ⟩

```
01100100011
mtcs
abc
stmts
0110010
```

Sample output:

```
stmts
100010011
Unexpected input
01100100011
Not a valid encoding.
```

Suggested prototypes for the functions:

- `char *decode(TRIE *t, char *input);` : returns a string containing the decoded input. If the input is not a valid encoding, it should return `NULL` and print the message given above.

  The calling function is responsible for freeing the memory allocated by `decode()` to store the decoded input.

- `char *encode(TRIE *t, char *input);` : returns a string containing the encoded input. If the input contains any letter outside a–z, it should return `NULL` and print the message given above.

  The calling function is responsible for freeing the memory allocated by `encode()` to store the encoded input.