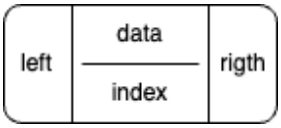
CS 2001 – Data Structures Project

Table of Contents

The problem (Part 1)

Step 1

(Total: 10 points) In this problem, you will create a .cpp program that, given a list of elements (excel data sheet is attached), prints out the positions in the list where the element is found. For this problem, you will use **Binary Search Trees (BST)**, which means you must use a dynamic doubly-linked list to store your data. Your tree will be composed of nodes that hold the `data` and the `left` and `right` subtrees, as well as an `index` that indicates the sequence in which the values are obtained in the original dataset. See the image below:



To create the proposed program, you need to perform the following steps:

1. Create the `search_lib_BST.h` file with the function headers for your problem. Here is what the .h file should have:
    a. The required libraries
    b. The required type definitions, including `dtype` (set to `long int`), `TreeNode`, which is the `struct node` structure for doubly linked lists (with `data` and a pointer to `left` and `right` subtrees), and Stack, which is a `struct stack` structure for a stack. The stack struct is defined as follow:

```
typedef struct stack {
    int top;
    Node *head;

} Stack;
```

    In this struct, Node is the usual singly linked list node we are familiar with (`data` and pointer to `next`). `Node` must also be defined in the .h file.
    c. All the required function headers as defined below (item 2).
2. Include the following function headers to your `search_lib_BST.h` search library:

a. `createTree()`: this function creates an empty tree. It returns a pointer to the root node, which is NULL when the tree is empty.

b. `printTree()`: this function receives a pointer to the root of a BST and prints the elements based on preorder traversal. Preorder traversal means that each visited node is printed out before visiting its subtree. This function must be recursive and follow this sequence: (i) visits the root; (ii) prints out the root; (iii) recursively calls the `printTree()` function to the left; recursively calls the `printTree()` function to the right. Base case is met when the root of a subtree is NULL, meaning that the recursive calls reached a leaf.

c. `addNode()`: this function receives a pointer to the root of a BST, the data to be inserted in the tree angd an index, which indicates the original position of the element in the input dataset. Then, it creates a node, adds the data and the index values to the node and insert the node in the BST. The function returns the pointer to the inserted node. Note that elements are added to the right if they are greater than the node, and to the left if they are lower or equal to the node.

d. `getData()`: this function opens a .csv file (attached to GCR) and reads all the data inside the file into a BST. The file name must be a string constant and be provided as a parameter. The processing steps are as follow: (i) create an empty tree; (ii) open the file; (iii) for each line in the file, read the value and add as a node in the BST. The index for the node will be whatever the sequence in the file is. For example, the first data in the file will be index 0, the second will be index 1, and so on. Once the tree is populated, close the file and return the pointer to the root.

   Each line in the .csv file contains only one value, which should be the `data` for one node in the list.

e. `saveData()`: this function opens a .csv file and saves all the `data` from the tree into the file, in preorder (see printTree function definition to understand preorder traversal). The file name and a pointer to the BST must be provided as parameters. Don't forget to close the file! The function has no required return.

f. `getFirstOccurrence()`: this function receives an element and a pointer to the root of a BST as parameters. Then, it finds and returns the `index` of the first occurrence of the element in the BST. Make sure you check if the tree is not empty before starting the search.

g. `getLastOccurrence()`: this function receives an element and a pointer to the root of a BST as parameters. Then, it finds and returns the `index` of the last occurrence of the element in the BST. Make sure you check if the tree is not empty before starting the search. Also, notice that once the element is found for the first time, the function needs to search for the element in the left subtree, as repeated elements are always added to the left.

h. `getAllOccurrences()`: this function receives an `element`, a pointer to the `root` of a BST, a pointer to an array of `occurrences` (output parameter named `occurrences`) and a pointer to an integer variable that represents the

number of occurrences of the `element` in the BST (output parameter named `numOccurrences`). The function must find all the occurrences of the `element` in the BST and store the `index` in the output parameter `occurrences`. Each time the `element` is found, the function must attribute the `index` to the `occurrences` array, update the `numOccurrences` and recursively call `getAllOccurrences()` to the left in order to find the next occurrence. No return is expected, but make sure the function returns without any value if the tree is empty.

i. `validateTree()`: we want to make sure that the tree we create when we inserted the nodes is a BST. To achieve that, the validate tree function will receive a pointer to the root of a tree and it will verify whether all the elements to the left of a subtree is smaller or equal then the root and all the elements to the right are greater than the root. The function will follow this sequence:

   - The function creates an empty stack and initialize two pointers: current and previous node. Current node starts at the root of the tree and previous starts at NULL.
   - Then, the function enters a loop that iterates until the current pointer is NULL and the stack is empty. Inside this loop, it does the following:
     i. For each node in the left subtree of the current node, push the encountered node onto the stack and move current to the next element to the left. This operation repeats until current pointer reaches a leaf.

ii. Once a leaf is reached, the function pops the top node from the stack and attribute to the current pointer. Then, it checks if its current pointer is less than or equal to the value of in the previous node (if there was one). If so, the tree is not a BST and the function can return FALSE. Otherwise, the function updates the previous pointer to be the current one.

iii. The next step is to set current to the right side of the subtree, which means current will receive whatever its own right node is. By doing that, the next iteration of the loop will traverse its right subtree.

iv. If the loop runs to the end without returning FALSE, then the tree is a BST and the function must return TRUE.

j. `createStack(), push(), pop(), isEmptyStack()`: these are the utility functions that will be necessary to create and use the stack in the `validadeTree()` function.

Notes:

- If you need to create a function that is not listed in this instruction file, add a comment at the bottom and list your own functions separate from the required ones. For example:

```
/*
  Supporting function headers
  Functions createEmpty() and addBeginning()
  Function printList() adapted from …
*/
```

Don't forget to add the comments to your functions and code. Remember that your functions must have the time complexity in the comments as well. Use the template below:

```
/*
  Function: <function name>
  Process: <what the function does>
  Input data (parameter): <input variables listed as arguments>
  Output data(parameter): <output variables listed as arguments>
  Output data (return): <returns>
  Dependencies: <functions called by the function>
  Complexity: <time complexity: O(1), O(n), O(n2), etc.>
*/
```

Step 2

(Total: 30 points) Create the file `search_lib_BST.cpp` file that imports the `search_lib_BST.h` library and implement all the functions specified in the library. Use the definitions provided above to decide how to implement your functions.

Step 3

Time to create the main function!

(Total: 10 points) Create a `BST.cpp` file that contains the following:

- imports the `search_lib_BST.h` library;
- opens a file and reads the data to be sorted from the input file;
- validates whether the data was created as a BST and prints an informative message;
- demonstrate that the functions to get the first, last and all occurrences of a value work for the given data. Search for the following values in the list: `232`, `649`, `989`, `1447`, and `1909`.

This file must also include two constants: `inputFilename` and `outputFilename`. Input file name is the name of the file where the initial data is stored. The input File is provided to you here. The output file will hold the sorted list. You can choose the name for that file as long as it has the .csv extension.

Submission: submit the `search_lib_BST.h`, `search_lib_BST.cpp` , and `BST.h` files as the solution for this part (Problem 1) of project.

Problem 2 (60 points)

Note**:** as usual, divide your program into library (Graph_p1_lib.h), library implementation (Graph_p1_lib.cpp), and main program (Graph_p1.cpp).

Suppose you are given a list of cities and the distance from each other, represented as an undirected graph. You need to find if there is a path between two given cities.

To optimize the search for a path, you decide to implement a hash table to store the distance between two cities. The hash table should take the form of a dictionary, where the keys are represented as the pair of cities (e.g., (*Flagstaff, Sedona*)) and the values are the distances between them.

Step 1:

You program must have at least the following structs:

```
typedef struct {
    char name[50];
    int index;
} City;

typedef struct node {
    int city;
    struct node* next;
} Node;
```

Step 2:

The program should have the following functions:

1. `void insert_edge(Node* adj_list[], int src, int dest)`: this function inserts an edge between vertices `src` and `dest` to the adjacency list received as a parameter. `src` and `dest` are integer values that represents the `index` of a `City`. Follow these procedures:

- The function first allocates memory for a new `Node`, which creates a new node for the linked list of vertex `src`.
- The new node's `city` is then initialized with the index in `dest`, and its `next` pointer points to the current head of the linked list of vertex `src`, which is stored in `adj_list[src]`.
- Finally, the head of the linked list of vertex `src` (`adj_list[src]`) is updated to point to the newly created node.

2. `void print_path(int parent[], int src, int dest, City cities[])`: this function print the path between two cities in a graph represented by an adjacency list. The function takes in four arguments:

- `parent`: an integer array containing the parent of each vertex in the path from source to destination city. If the parent doesn't exist (no incoming edges to destination vertex), `parent[dest]` is set to -1.
- `src`: an integer representing the `index` of the source city in the `cities` array.
- `dest`: an integer representing the `index` of the destination city in the `cities` array.
- `cities`: an array of `City` structs representing the cities in the graph.

The function first checks if there is a path between the source and destination cities by checking if the parent of the destination city is -1. If it is -1, then there is no path, and the function prints a message to indicate that (e.g., *No path exists between Flagstaff and Cairo.*). If there is a path, the function initializes an integer array called `path` and a variable called `path_index` to keep track of the vertices in the path. It then traverses the parent array from the destination city to the source city, storing each vertex in the `path` array. After the loop, the source city is also added to the `path` array.

Then, the function calculates the distance between source and destination nodes and prints a message indicating that a path exists and the distance, followed by the names of the cities in the path. To do so, the function traverses the `path` array in reverse order, accumulates the distance for each edge in the path and concatenate a string with the names of the cities I the path. At the end, the function will print a message such as the following (e.g. *Path between Flagstaff and Phoenix (145 miles): Williams, Rimrock, Camp Verde, Spring Valley, Bumble Bee, Rock Springs, Anthem, Glendale*).

Note: see functions 5 and 6 for distance calculations. As a suggestion (not a requirement!), You may want to write utility functions to support the aggregation of distances. You may want to add the aggregated distances into the distances array so that you can easily find the distances without having to call this function repeated times if needed. For example, if you find the distance between Flagstaff and Phoenix through Williams (Flagstaff – Williams - Phoenix), you may want to add a distance Flagstaff-Phoenix to the distances array.

3. `void bfs(Node* adj_list[], int src, int dest, City cities[], int n)`: this function implements the Breadth-First Search algorithm to traverse an adjacency list represented graph, starting from a given source node (`src`) and searching for a path to a destination node (`dest`). Remember that `src` and `dest` are integer indexes that indicates the position of the nodes in the adjacency list. For example, `Node * temp = adj_list[src]` finds the source node and attributes its address to `temp`.

Follow these procedures:

- The function initializes two arrays, `visited` and `parent`, to keep track of the visited nodes and the parent nodes in the traversal. It then adds `src` to a `queue` and marks it as visited.
- While the `queue` is not empty, the function `dequeues` an index from the `front`, and iterates over its adjacent nodes (using the `next` pointer in the `Node`), and `enqueues`

any unvisited adjacent nodes to the back of the queue. During this process, it updates the `visited` and `parent` arrays accordingly.

- If the destination node is found during the traversal (`city` in the Node is the same as `dest`), the function calls the `print_path` function to print the path from the source node to the destination node and ends. If the destination node is not found, the `print_path` function is called with a special flag value indicating that no path exists between the source and destination nodes (-1), and the function ends.

4. `int hash(char* city1, char* city2)` : given the name of the two cities, the hash function calculates the key by aggregating the ASCII characters in both city names and dividing the result by the maximum length of the hash table. The maximum length is a constant that exists in the program. The function returns the result of the key calculation. For example, assuming that the city names are *Flagstaff* and Sedona, and the maximum length of the hash table is 50, the result would be:

F	l	a	g	s	t	a	f	f
70	108	97	103	115	116	97	102	102

S	e	d	o	n	a
83	101	100	111	110	97

Key = [(70+108+97+103+115+116+97+102+102) + (83+101+100+111+110+97)] % 50

Key = [910 + 602] % 50

Key = 12

5. `int find_distance(double distances[], char* city1, char* city2)`: this function takes an array of distances, representing distances between cities, and two city names as input. It first calculates a key for the pair of cities using the hash function. It then looks for a distance whose position matches the calculated key for the given pair of cities. If a match is found, the function returns the corresponding distance value. If no match is found, the function returns -1, indicating that the distance was not found.

6. `int add_distance(double distances[], char* city1, char* city2, double distance)`: this function takes an array of distances, representing distances between cities, two city names, and a distance as input. It first calculates a key for the pair of cities using the hash function. It then simply sets the value of the distances array at the given key/index to the given distance value. This function must be called every time an edge is create between two cities.

Note: this approach doesn't treat collision. If the position is occupied, the function returns an error code (-1) and the program aborts with a message to the user.

Step 3:

Create a main program that shows that your functions work. Your main program must:

Create a list of at least 5 cities with some edges between them. Make sure you add the distance of the edge for each edge created. Then, the program prints the path between two cities when the path exists and the distance between the two cities using the `print_path` function.

Problem 3 (40 points)

When using hashing, collision may easily become an issue. In the previous problem, we did not resolve collision when they occurred. In this problem, we will modify P1 library to resolve collision using linear probing. Follow the steps bellow:

Step 1:

Create a copy of your solution files for Problem 1 and rename the new files as Graph_p2_lib.h, Graph_p2_lib.cpp, Graph_p2.cpp.

Step 2:

change your array of distances whenever necessary to be of type Distance, which is a user-defined struct as follows:

```
typedef struct {
    char* city1;
    char* city2;
    int cities_key;
    int distance;
} Distance;
```

Step 3:

Change the functions 5 and 6 (find_distance and add_distance) to take Distance distances[] array (instead of double distances[]) as a parameter. Then, modify the functions so that they use linear probing to add and find the distances.

For the add_distance function, starting on the key position, if there is a collision, traverse the distances array linearly until an empty position is found and add the distance. For the find_distance function, starting on the key position, check: (i) if the cities_key is the same as the key generate from the hash function. If so, check if the city names are the same in which case the function returns the distance; (ii) if the cities_key is NOT the same as key generate from the hash function in which case the function traverses the distance array linearly until it finds the correct key. Don't forget to double check the city names. If the city names are not the same, the function keeps looking!

Step 4:

Change the main function to show that the new functions work. The only thing you have to do here is to force some collisions, so that you demonstrate the function execution.

Note: make all the necessary changes in the files so that your program works with the new Distance struct. However, avoid changing more than you have. You are expected to make the least amount of changes you can, with focus on functions 5 and 6 and the functions that call them.

----------Happy coding----------