VIETNAM NATIONAL UNIVERSITY - HO CHI MINH CITY
HO CHI MINH CITY UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

DATA STRUCTURES AND ALGORITHMS - CO2003

ASSIGNMENT 2

BUILDING CONCAT_STRING USING TREE AND HASH STRUCTURE

Author: Tien Vu-Van

Ho Chi Minh City, 10/2022

ASSIGNMENT'S SPECIFICATION

Version 1.1

1 Assignment's outcome

After completing this assignment, students review and make good use of:

- Object Oriented Programming (OOP)
- Binary Search Tree and AVL Tree
- Hash data structure.

2 Introduction

In the first assignment, students were asked to implement string literals using a List data structure to reduce the complexity of string concatenating operation. This implementation comes with some limitations: the operation of accessing a single character at a location has high complexity; or a string can only participate in the concatenation operation once.

In this second assignment, students are asked to use tree and hash structures to implement string literals and resolve the above limitations. The class that represents the implemented string literal will be called ConcatStringTree.

3 Description

3.1 Overview

Figure 1 depicts two strings s1, s2 using a tree structure. With a tree structure, string concatenation can be easily done by creating a new node whose left subtree is s1 and the right subtree is s2.

In addition, to facilitate the get-at index operation, the tree structure **Binary Search Tree** is selected to implement. The key at each node of the tree will be chosen as the length of the string on the left. When searching (with a specified index) at each node, if this index is less than the key, then continue searching in the left subtree, otherwise, search in the right subtree.

By choosing the key as the length of the string on the left, we get the key of s3 (join s1 with s2) in Figure 1 has the value of 8. Then, the operation to find the key equals 8 has complexity $O(log(n))$ (try to figure out how to calculate this complexity). To reduce complexity, we will store more information about the total length of the string at each node. Figure 2 suggests the information of a node in the string representation tree s1.

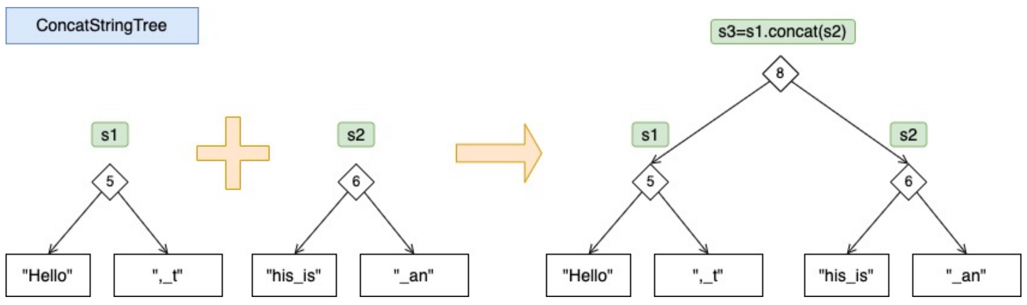

Figure 1: Illustration of the string concatenation

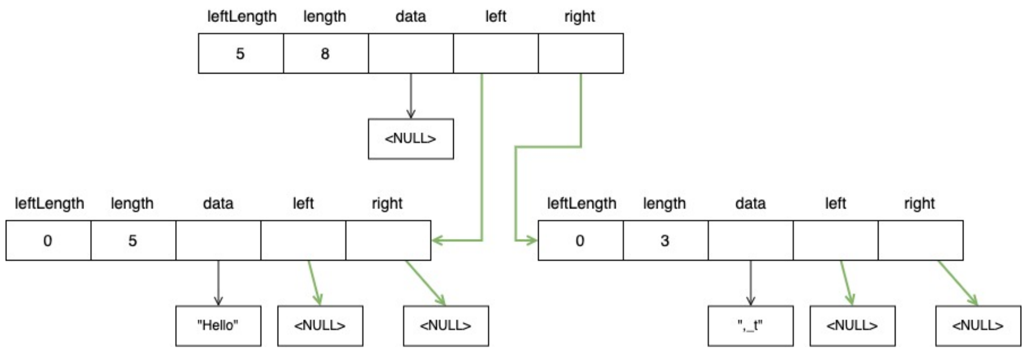

Figure 2: Illustrate a node of the string representation tree

From the above information, the following depictions will describe the leaf nodes (only contain the data) as a diamond with the value 0 (similar to the other nodes, representing the length of the left substring) pointing to a string. The following sections will describe in detail about the class that need to be implemented in this Assignment.

3.2 class ConcatStringTree

Methods to be implemented for the class ConcatStringTree:

1. `ConcatStringTree(const char * s)`
   - Initialize a ConcatStringTree object with the data field that points to an object representing a contiguous string of characters with the same value as the string **s**.

- Complexity (all cases): O(n) where n is the length of the string `s`.

2. `int length() const`

- Returns the length of the string being stored in the ConcatStringTree object.
- Complexity (all cases): O(1).

Example 3.1

In Figure 1:

- s1.length() returns 8.
- s2.length() returns 9.

3. `char get(int index) const`

- Returns the character at position ***index***.
- Exception: If ***index*** has an invalid position in the string, an exception is thrown (via the **throw** command in the C++ language): `out_of_range("Index of string is invalid!")`. ***index*** is a valid position if `index` is within $[0, l-1]$ where l is the length of the string.
- Complexity (average case): O(log(k)) where k is the number of nodes of the ConcatStringTree.

Example 3.2

In Figure 1:

- s1.get(14) throws an exception
  `out_of_range("Index of string is invalid!")`
- s2.get(1) returns character 'i'.

4. `int indexOf(char c) const`

- Returns the position of the first occurrence of ***c*** in the ConcatStringTree. If the character c does not exist, the value -1 is returned.
- Complexity (worst case): $O(l)$ where l is the length of the ConcatStringTree string.

Example 3.3

Trong Hình 1:

- s1.indexOf('i') returns -1.
- s2.indexOf('i') returns 1.

5. `string toStringPreOrder() const`
   - Returns the string representation for the ConcatStringTree object when traversing nodes in the prefix order NLR.
   - Complexity (all cases): $O(l)$ where l is the length of the ConcatStringTree string.

Example 3.4

In Figure 1:

– s1.toStringPreOrder() returns
"ConcatStringTree[(LL=5,L=8,¡NULL¿);(LL=0,L=5,"Hello");(LL=0,L=3,",_t")]"

6. `string toString() const`
   - Returns the string representation for the ConcatStringTree object.
   - Complexity (all cases): $O(l)$ where l is the length of the ConcatStringTree string.

Example 3.5

In Figure 1:

– s1.toString() returns
`"ConcatStringTree["Hello,_t"]"`
– s2.toString() returns
`"ConcatStringTree["his_is_an"]"`

7. `ConcatStringTree concat(const ConcatStringTree & otherS) const`
   - Returns a new ConcatStringTree object as described in Section 3.1.
   - Complexity (all cases): $O(1)$
   - Example: Revisit the Figure 1.
8. `ConcatStringTree subString(int from, int to) const`
   - Returns a new ConcatStringTree object containing characters starting from position `from` (including `from`) to position `to` (excluding `to`).
   - Exception: If **from** or **to** is an invalid position in the string, throw an exception `out_of_range("Index of string is invalid!")`. If $from >= to$ then throw an exception `logic_error("Invalid range!")`.
   - Example: Figure 3 demonstrates the subString operation.
   - Note: The new string needs to ***create new*** nodes (don't reuse the node in the original tree). You should also try to keep the link between the nodes like the original

tree. "Keep the link between the nodes like the original tree" means retaining the link completely, from the root node to leaf nodes containing the characters in the substring range. Leaf nodes with characters outside this range will not be included in the new object.

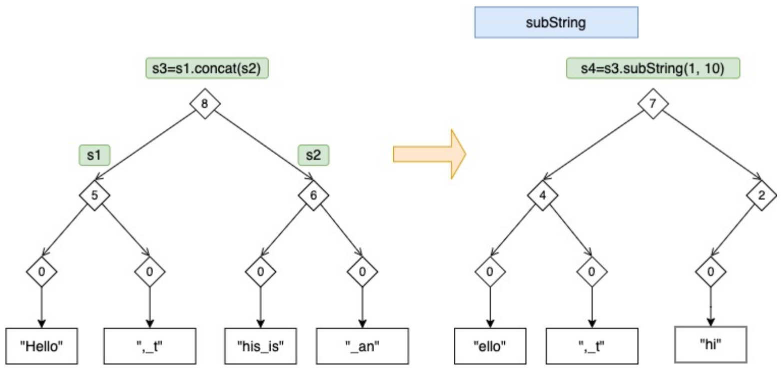

Figure 3: Illustrate the string after performing the subString operation

9. `ConcatStringTree reverse() const`

- Returns a new ConcatStringTree object representing an inverse of the original string.
- Left nodes of old ConcatStringTree object will become right nodes of new ConcatStringTree object and right nodes of old ConcatStringTree object will become left nodes of new ConcatStringTree object.
- Example: Figure 4 illustrates the reverse operation.

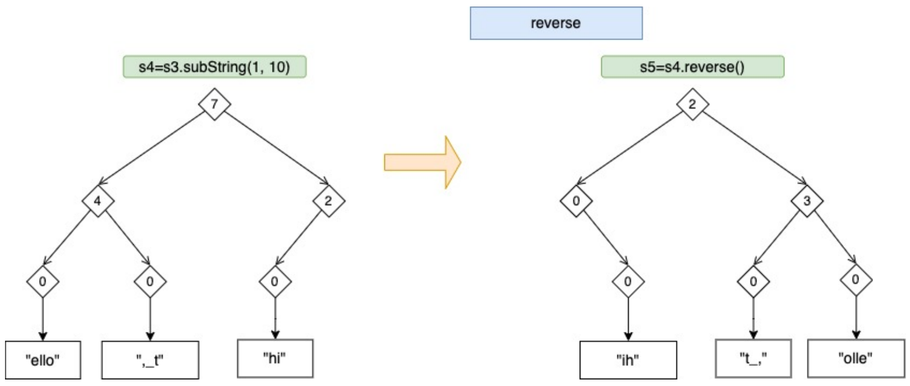

Figure 4: Illustration of reverse operation

10. ~`ConcatStringTree()`

- Implement the destructor so that all dynamically allocated memory must be freed after the program terminates. Considering s1 in Figure 1, normally, if s1 is deleted, the nodes containing the strings "Hello" and ",_r" will be deleted. However, this is not appropriate because these 2 strings are still needed for the formation of string 3. Refer to Section 3.3 for appropriate destructor implementation.

3.3 class ParentsTree

Review Figure 1 illustrating the result of concatenating two strings. When we want to delete s1, if we delete all the nodes in the tree s1, we will lose the strings "Hello" and ",␣r". Hence, the information is lost to represent the tree s3. To solve this problem, at each node, we will store the adjacent parent nodes that point to this node. When performing deletion across nodes, we can check if the current node has no parent, then delete this node and the nodes in the subtree.

Students are required to use the AVL tree structure to store information about the parent nodes in each node of the ConcatStringTree. To use AVL, we need to choose a corresponding key for a node. Students need to implement a simple identifier (id for short) generation mechanism and use this id as the key for the node. The mechanism for generating id is as follows:

- The first node created has the id value 1.
- Nodes are then spawned with an id equal to the previously allocated largest id plus 1.
- Id can only be granted up to 10^7 in value. If the given node id is greater than 10^7, an exception is thrown **overflow_error("Id is overflow!")**.

Class ParentsTree represents the AVL tree used to store the parent nodes of the current node. In ParentsTree, if we delete a node and this node has left and right subtrees, we will take the **largest of the left subtree** node to replace the current node.

When deleting a node in ConcatStringTree, we will remove information about that node from the ParentsTree of the left subtree root node and ParentsTree of the left subtree root node. If a node's ParentsTree is empty then that node is not used to create a larger tree, we can delete that node and the subtree nodes.

On the other hand, in Figure 1, deleting s3 (before s1 and s2) will delete s1 and s2, the representation information for s1 and s2 will be lost. To solve this problem, we will add a value to ParentsTree so that when the delete process s3 passes to s1, the ParentsTree is not empty and will not continue the deletion process. Specifically, when creating a ConcatStringTree node, we will add the node's id to ParentsTree. At the same time, students should note that ParentsTree is updated each time the string concatenation is performed.

Students need to implement the following methods for **class ParentsTree** (these are the methods used in testcases, students implement other methods themselves if needed):

1. **int size() const**
   - Returns the number of nodes in ParentsTree.
   - Complexity: O(1).

   Example 3.6

   In Figure 1:

   – Calling size() on the ParentsTree object in the root node of s1 returns 2.
   – Calling size() on the ParentsTree object in the node with the ”Hello” data of s2 returns 2.

2. **string toStringPreOrder() const**
   - Returns the string representation of the ParentsTree object when traversing nodes in prefix order NLR. The string representation has the following format:
     ”ParentsTree[¡node_list¿]”
     Where ¡node_list¿ is a list of nodes separated by a semicolon. The format of a node is:
     ”(id=¡node_id¿)”
     Where ¡node_id¿ being the id of node.

   Example 3.7

   For example, the result of the toStringPreOrder() method has
   - 0 node (empty):
     `"ParentsTree[]"`
   - 1 node:
     `"ParentsTree[(id=1)]"`
   - 2 node:
     `"ParentsTree[(id=2);(id=3)]"`

Students must implement the following method for **class ConcatStringTree**:

1. **int getParTreeSize(const string & query) const**
   - Returns the number of nodes in ParentsTree at a specified node in ConcatStringTree.
   - The string **query** is used to identify the node that wants to access ParentsTree. **query** contains only two characters ’l’ or ’r’, otherwise throws an exception: **runtime_error(”Invalid character of query”)**. How to determine the node based

on the query: Initially we are at the root node of ConcatStringTree. Iterate through the characters in **query** one by one, if the character 'l' is encountered, we move to the left subtree, if the character 'r' is encountered, we move to the right subtree. If during execution, we encounter a NULL address, we throw an exception: **runtime_error("Invalid query: reaching NULL")**.

2. **string getParTreeStringPreOrder(const string & query) const**
    - Returns the string representation for the ParentsTree object at a specified node in the ConcatStringTree. This string representation is the return result of the **toStringPreOrder** method of the ParentsTree class.
    - The string **query** is used to identify the node that wants to access ParentsTree. The rules and ways to access of **query** are the same as described in the **getParTreeSize** method.

3.4 class ReducedConcatStringTree and class LitStringHash

Considering Figure 5, strings s1 and s2 are objects of class ConcatStringTree. We recognize that the string "hello" appear twice: the left subtree of s1 and the right subtree of s2. In this section, we will find a way to reduce the string storage size by pointing data to an existing memory area if the newly created string is the same as one of the created strings (like the s2 string). The class that represents strings with reduced storage is called ReducedConcatStringTree. ReducedConcatStringTree needs to have all the properties and methods of the ConcatStringTree class.

Let LitString be the strings containing the actual data. In the figure 5, "hello", "there", "here" are LitStrings; node¡5¿ and node¡6¿ are not LitString. We will use the Hash structure to store the LitString, and call the class representing this structure LitStringHash.

- When a string is generated, if that LitString already exists in the LitStringHash, then we point the data to the address of this LitString (do not create a new LitString). Conversely, if the LitString does not already exist in the LitStringHash, then we add (insert) this new LitString to the LitStringHash. Each LitString in the LitStringHash should have additional information about the number of links to it.
- When deleting a string, if a certain LitString no longer has a link to it, we remove it from the LitStringHash. If after deleting and LitStringHash becomes empty, we need to free the allocated memory for LitStringHash. In the following operations (if any), we need to re-allocate memory for LitStringHash if needed.

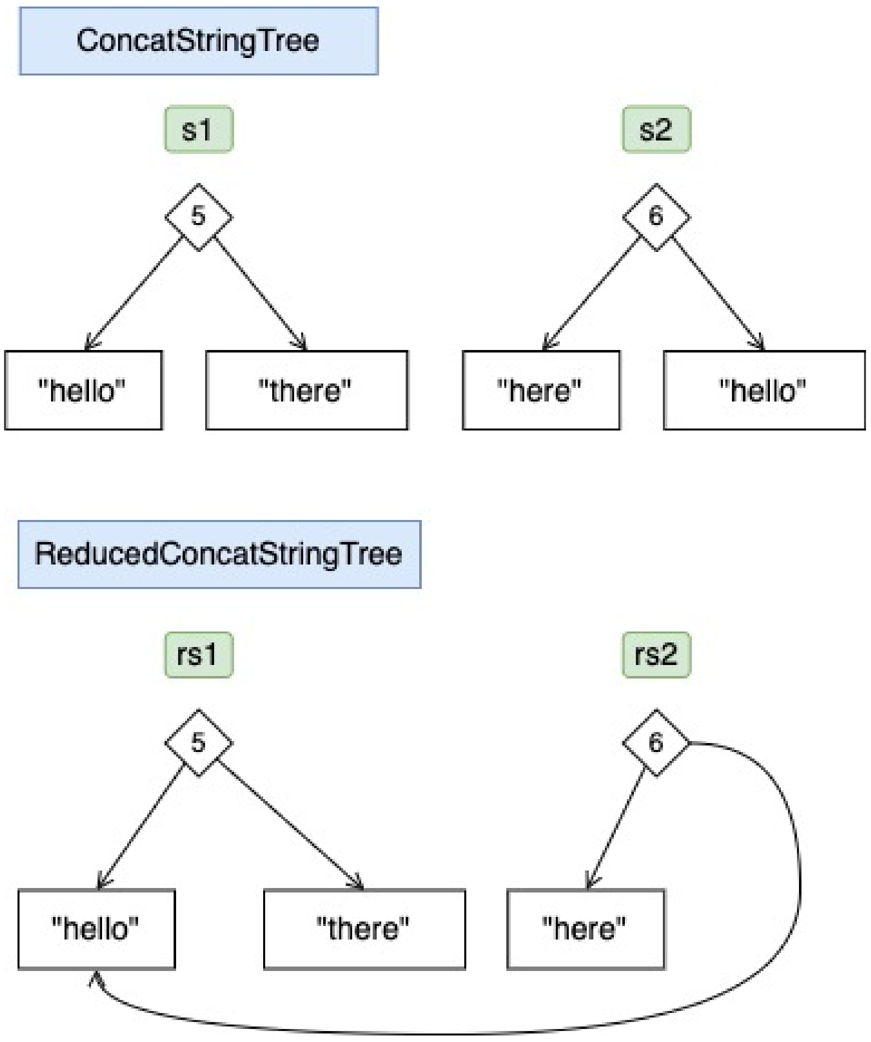

Figure 5: Illustration of reducing memory for strings

The LitStringHash class has the following characteristics:

- Assume **s** is a LitString. The hash of LitStringHash is as follows:

$$h(s) = s[0] + s[1] * p + s[2] * p^2 + ... + s[n-1] * p^{n-1} \mod m$$

  In which:

  - n is the length of the string
  - s[0], s[1],..., s[n-1] are integers representing the corresponding ASCII code of the character at positions 0, 1,..., n-1, respectively
  - m is the size of the hash table
  - p is the configuration parameter which will be described later.

- Probe function: Using quadratic probing:

$$hp(s, i) = (h(s) + c1 * i + c2 * i^2) \mod m$$

In which:

- m is the size of the hash table
- c1, c2 are the configuration parameters which will be described later.

Note: If no available slot is found to add, an exception is thrown:
runtime_error("No possible slot")

- Rehashing: After adding a value to LitStringHash, if LitStringHash has a load factor greater than *lambda* then we need to perform rehash. The load factor is calculated as the ratio between the number of available elements and the size of the LitStringHash. *lambda* is a configuration parameter. Steps to perform a rehash:
  - Create a new LitStringHash of size alpha $*$ size (rounded down) where size is the current size of the LitStringHash.
  - Iterate over the positions of the LitStringHash in turn, if any position has a LitString, we insert it into the new LitStringHash.
- Class HashConfig: This class contains configuration parameters for the above operations, including:
  - p: is an integer used in the hash function.
  - c1, c2: are real numbers (of type double) used in the probe function.
  - lambda: is a real number (of type double) representing the rate at which if the load factor is greater than lambda, it will perform rehashing.
  - alpha: is a real number (of type double) representing the ratio multiplied by the old size to produce a larger new size.
  - initSize: is an integer, representing the size of the LitStringHash when initialized or when adding a new value after freeing the memory for the LitStringHash.

Students needs to implement the following methods for **class LitStringHash**:

1. **LitStringHash(const HashConfig & hashConfig)**
   - Takes in a **hashConfig** object and initializes the necessary configuration parameters for the hash operation.
2. **int getLastInsertedIndex() const**
   - Returns the last position inserted to the LitStringHash and after the rehash operation was performed (if any). If no element has been inserted before, or a memory allocation was just made for LitStringHash, the value -1 is returned.
   - Complexity (all cases): O(1).

3. **string toString() const**
   - Returns the string representation for the LitStringHash object. The string representation has the following format:
     "LitStringHash[¡slot_list¿]"
     Where ¡slot_list¿ is a list of elements separated by a semicolon. The format of an element is:
     - If that position has no value:
       "()"
     - If that position has value:
       "(litS=¡lit_string¿)" Where ¡lit_string¿ is the string representation of the LitString.

Example 3.8

Example of a result of the toString() operation

– `"LitStringHash[();(litS="Hello");();(litS="there")]"`

Students needs to implement the following method for **class ReducedConcatStringTree**:

1. **ReducedConcatStringTree(const char * s, LitStringHash * litStringHash)**
   - Initialize a ReducedConcatStringTree object with the data field that points to an object representing a contiguous string of characters with the same value as the string s (same as ConcatStringTree's constructor). Also, initialize a pointer to the LitStringHash object. This object will store the hash table information shared for ReducedConcatStringTree objects.
   - Complexity (all cases): O(n) where n is the length of the string s.

3.5 Requirements

To complete this assignment, students must:

1. Read entire this description file.
2. Download the initial.zip file and extract it. After extracting, students will receive files including main.cpp, main.h, ConcatStringTree.h, ConcatStringTree.cpp and sample_output folder. Students will only submit 2 files, ConcatStringTree.h and ConcatStringTree.cpp. Therefore, you are not allowed to modify the main.h file when testing the program.

3. Students use the following command to compile:
   g++ -o main main.cpp ConcatStringTree.cpp -I . -std=c++11
   The above command is used in the command prompt/terminal to compile the program. If students use an IDE to run the program, students should pay attention: add all the files to the IDE's project/workspace; change the IDE's compile command accordingly. IDEs usually provide buttons for compiling (Build) and running the program (Run). When you click Build, the IDE will run a corresponding compile statement, normally, only main.cpp should be compiled. Students need to find a way to configure the IDE to change the compilation command, namely: add the file ConcatStringTree.cpp, add the option -std=c++11, -I .
4. The program will be graded on the Unix platform. Students' backgrounds and compilers may differ from the actual grading place. The submission place on BKeL is set up to be the same as the actual grading place. Students must test the program on the submission site and must correct all the errors that occur at the BKeL submission site in order to get the correct results when final grading.
5. Modify the files ConcatStringTree.h, ConcatStringTree.cpp to complete this assignment and ensure the following two requirements:
   - All the methods in this description must be implemented for the compilation to be successful. If the student has not implemented a method yet, provide an empty implementation for that method. Each test case will call some described methods to check the returned results.
   - There is only one **include** directive in the ConcatStringTree.h file **#include "main.h"** and one directive in the ConcatStringTree.cpp file **#include "ConcatStringTree.h"**. Apart from the above directives, no other **#include** is allowed in these files.
6. Students are allowed to write additional methods and properties in classes that are required to be implemented.
7. Students are required to design and use learned data structures.
8. The student must free all the dynamically allocated memory when the program terminates.

4 Submission

Students submit only 2 files: ConcatStringTree.h and ConcatStringTree.cpp, before the deadline given in the link "Assignment 2 - Submission". There are a number of simple test cases used to check student work to ensure that student results are compilable and runnable. Students can submit as many times as they want, but only the final submission will be graded. Since the system cannot bear the load when too many students' submissions at once, students should submit their work as soon as possible. Students do so at their own risk if they submit assignments by the deadline. When the submission deadline is over, the system will close so students will not be able to submit any more. Submissions through other means will not be accepted.

5 Other regulations

- Students must complete this assignment on their own and must prevent others from stealing their results. Otherwise, the student treat as cheating according to the regulations of the school for cheating.
- Any decision made by the teachers in charge of this assignment is the final decision.
- Students are not provided with testcases after grading, but are only provided with information on testcase design strategies and distribution of the correct number of students according to each test case.
- Assignment contents will be harmonized with a question in exam with similar content.

6 Changelog

- Add a note to the ConcatStringTree object illustration.
- Add a description of retaining the association for the subString method.
- Add a description for the reverse method.
- Modify the result of example 3.6 (for the size method).