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 1

BUILDING CONCAT_STRING USING LIST

Author: Tien Vu-Van

Ho Chi Minh City, 09/2022

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)
- List data structures
- Sorting algorithms

2 Introduction

Strings are often used to represent a piece of text (e.g. sentence, paragraph), thereby displaying meaningful information to the user. Typically, string literals are implemented using an array list to store contiguous characters. However, if strings are stored that way, the operation concatenate/join two strings of length m and n respectively has a complexity of $O(m + n)$.

On the other hand, a linked list is a data structure that can perform the join operation of 2 lists with less complexity.

In this assignment, students are asked to implement a string class that efficiently supports string concatenation using list data structures called ConcatStringList.

3 Description

3.1 Overview

Figure 1 shows how to implement ConcatStringList. ConcatStringList s1 has 3 CharALNode, each CharALNode has 2 pieces of information: string literal and link to next CharALNode. The string literal stored as a CharArrayList object, which is an Array List containing the characters of the string. The CharArrayList helps to manipulate characters at random indexes efficiently.

On the other hand, string concatenation is effectively performed by concatenating the last CharALNode of the previous ConcatStringList with the first CharALNode of the following ConcatStringList. For example, in Figure 1, when concatenating the string s1 with s2, we

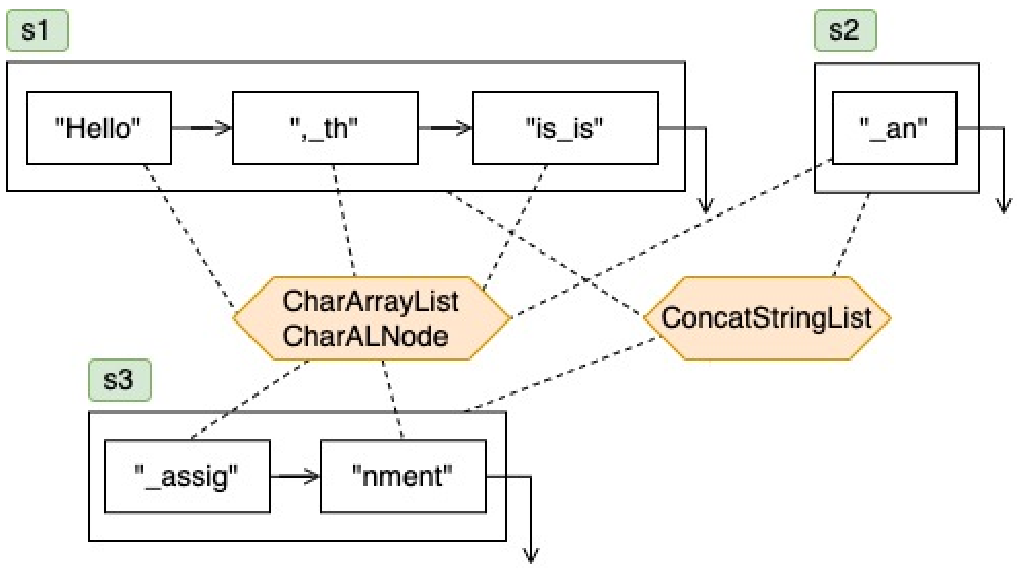

Figure 1: Overview of string representation

simply point the association of CharALNode "is_is" to CharALNode "_an". The result of the join is shown in Figure 2.

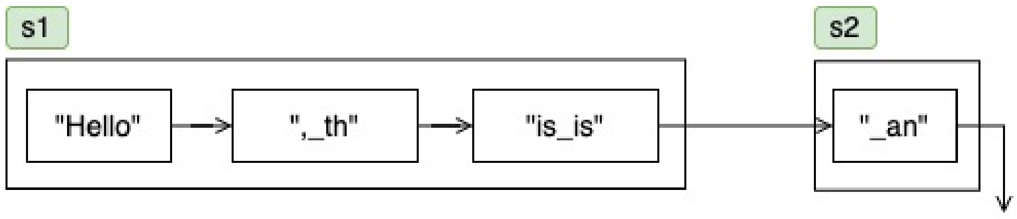

Figure 2: The result of string concatenation

The following sections describe in detail the classes that need to be implemented.

3.2 class ConcatStringList (6 pts)

Methods to be implemented for the ConcatStringList class:

1. `ConcatStringList(const char * s)`
   - Initialize a ConcatStringList object with 1 CharALNode. In CharALNode, there is a CharArrayList initialized with the string literal ***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 ConcatStringList object.

- Complexity (all cases): O(1).

Example 3.1

In Figure 1:

- s1.length() returns 14.
- s2.length() returns 3.

3. `char get(int index) const`

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

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 'a'.

4. `int indexOf(char c) const`

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

Example 3.3

In Figure 1:

- s1.indexOf('i') returns 9.
- s2.indexOf('b') returns -1.

5. `string toString() const`

- Returns the string object representation of the ConcatStringList object.
- Complexity (all cases): $O(l)$ where l is the length of the ConcatStringList string.

Example 3.4

In Figure 1:

- s1.toString() returns `"ConcatStringList["Hello,_this_is"]"`
- s2.toString() returns `"ConcatStringList["_an"]"`

6. `ConcatStringList concat(const ConcatStringList & otherS) const`

   - Returns a new ConcatStringList object and performs the linking of the current object's last CharALNode to the first CharALNode in the `otherS` object.
   - Complexity (all cases): $O(1)$
   - Example: Figures 3 and 4 represent strings before and after the concatenation, respectively. Note: the ConcatStringList class needs to make sure there are two pointers to the first and last CharALNode. These two pointers will be used in some of the upcoming tasks of this assignment.

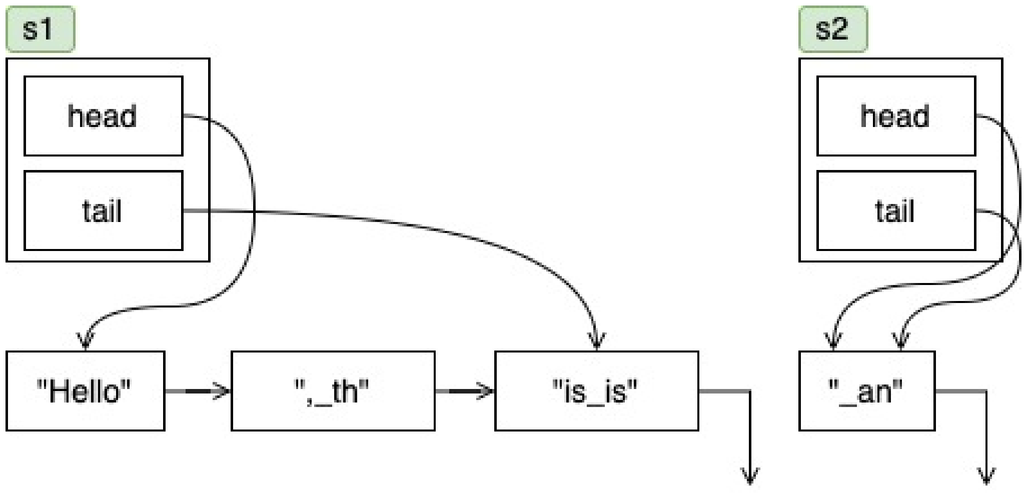

Figure 3: Illustration of the string before concatenation

7. `ConcatStringList subString(int from, int to) const`

   - Returns a new ConcatStringList 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 the exception `out_of_range("Index of string is invalid" )`. If $from >= to$, then throw the exception `logic_error("Invalid range")`.
   - Example: Figure 5 illustrates the subString operation.

8. `ConcatStringList reverse() const`

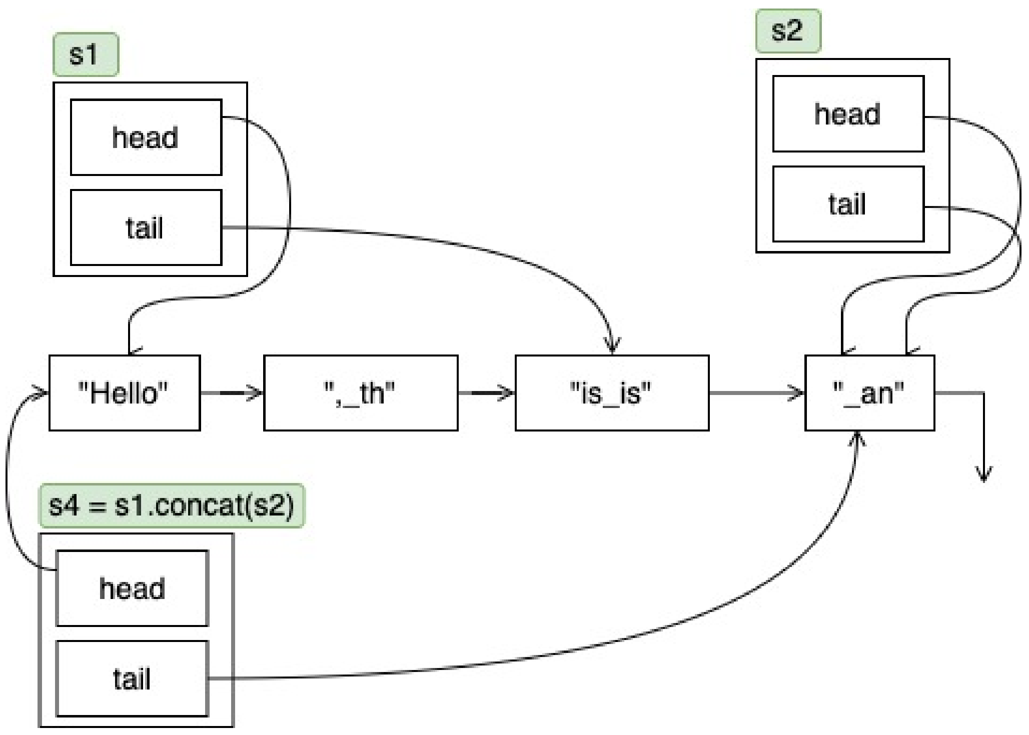

Figure 4: Illustration of the string after concatenation

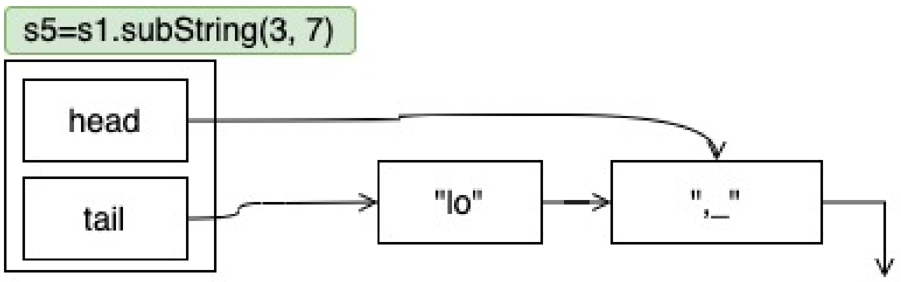

Figure 5: Illustration of the string after performing the subString operation

- Returns a new ConcatStringList object representing the original string after being reversed.
- Example: Figure 6 illustrates the reverse operation.

9. ~`ConcatStringList()`
    - Implement the destructor so that all dynamically allocated memory must be free after the program terminates. Usually, a destructor will free the CharALNodes between head and tail. However, this is not appropriate because some CharALNode may still be referenced as a result of a string join operation. Refer to Section 3.3 for appropriate destructor implementation.

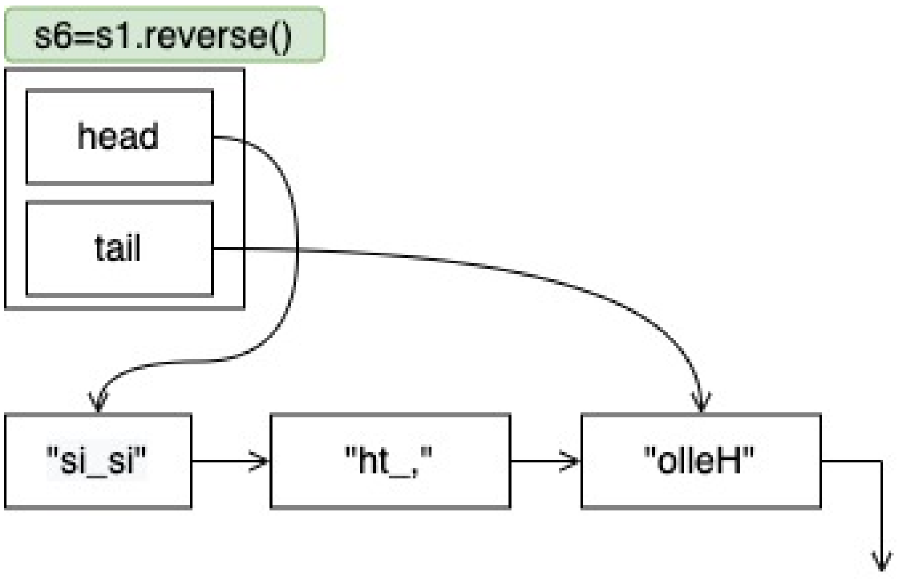

Figure 6: Demonstration of reverse operation

3.3 class ReferencesList và class DeleteStringList (4 pts)

Let us take a look back at Figure 4. The figure illustrates the result of the concatenating operation. Suppose we want to delete the string s2, if the CharALNode "_an" is deleted, the string s4 will only have 3 CharALNode and no longer properly represent the result of the string concatenation. To solve this problem, we will maintain a list of reference counts to the first and last CharALNodes, represented by the ReferencesList class. For example, in Figure 4, the CharALNode of "is_is" has a reference count of 1 (from the tail of s1), the CharALNode of "_an" has a reference count of 3. At the same time, we maintain a list of deleted strings, represented by the DeleteStringList class. Each node of the DeleteStringList holds 2 pieces of information: the first and the last CharALNode of a deleted string. We will traverse through this list, check if both head and tail of a string have zero total references and then delete the CharALNodes between that head and tail (including head and tail). Here are some requirements when implementing the ReferencesList class and the DeleteStringList class:

- Every time a **new** string is created, the first and last CharALNode of this string are added to the ReferencesList for tracking. Of the nine required implementations for the ConcatStringList class, the following methods create a new object:
  - **ConcatStringList(const char *)**
  - **ConcatStringList concat(const ConcatStringList & otherS) const**
  - **ConcatStringList subString(int from, int to) const**
  - **ConcatStringList reverse() const**

- Every time we delete a string, we will reduce the number of references in the ReferencesList corresponding to the CharALNode head and tail of the string by 1 unit. Also, add a new node with this head and tail information at the end of the DeleteStringList. Next, we traverse through the nodes in the DeleteStringList, if any node has a total number of references to head and tail equal to 0, then we delete the CharALNodes between head and tail, and then remove the node from the DeleteStringList. If there are no nodes left after deleting DeleteStringList, we delete all nodes in ReferencesList. Note, when deleting CharALNodes between head and tail, it is necessary to check if head and tail have been deleted before or not.
- When deleting a string, we need to pay attention to the CharALNodes that have a low reference count so that after reducing the number of references, there might be a chance that the number of references is zero, which may leads to the deletion of some CharALNodes. To improve the search efficiency in the ReferencesList, we will always maintain **increasing order** on this list. Then, as a matter of course, nodes with 0 reference will always be at the top of the list. However, those nodes are no longer needed for the search. Therefore, nodes with 0 reference **should be at the end** ReferencesList.

See also Figures 7, 8, 9 illustrating ReferenceList and DeleteStringList through the sequence deletion steps. Note: Figure 9, DeleteStringList does not illustrate the final result. Since the number of references to "_world" is zero, the first node of the DeleteStringList (including "no. references of s1 head" and "no. references of s1 tail") will be deleted. DeleteStringList then has only 1 single node.

Methods to be implemented for **class ReferenceList**:

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

   Example 3.5

   Calling size() with the reference list in Figure 7 returns 2.

2. **int refCountAt(int index) const**
   - Returns the count corresponding to the reference at position **index**.
   - Exception: If **index** has an invalid value, throw an exception: `out_of_range("Index of references list is invalid!")`.

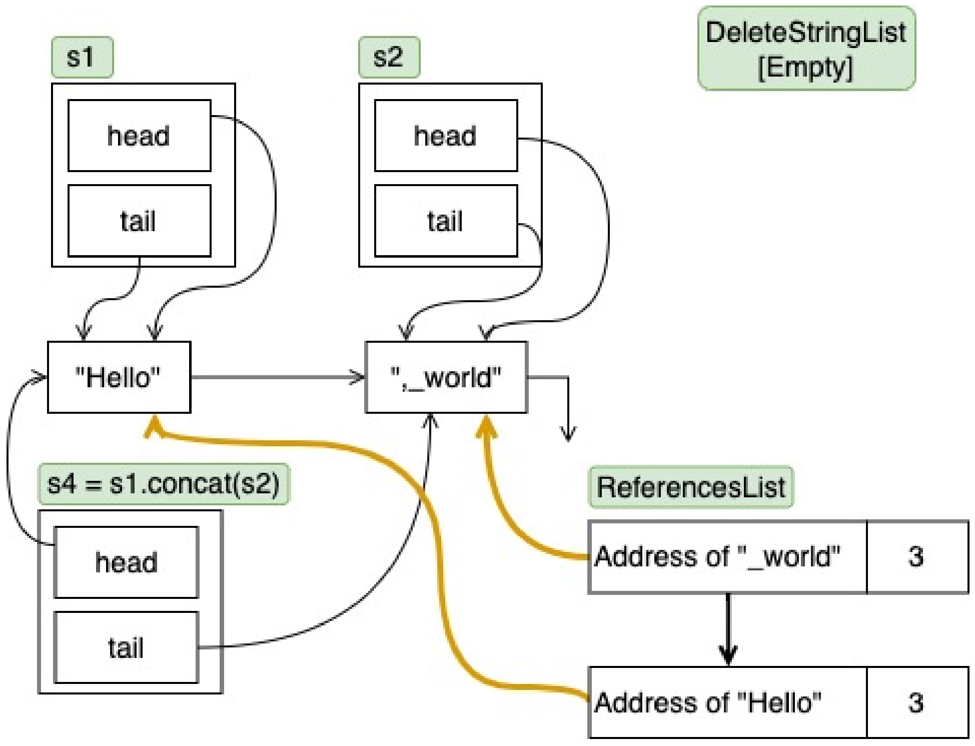

Figure 7: Illustration of ReferenceList

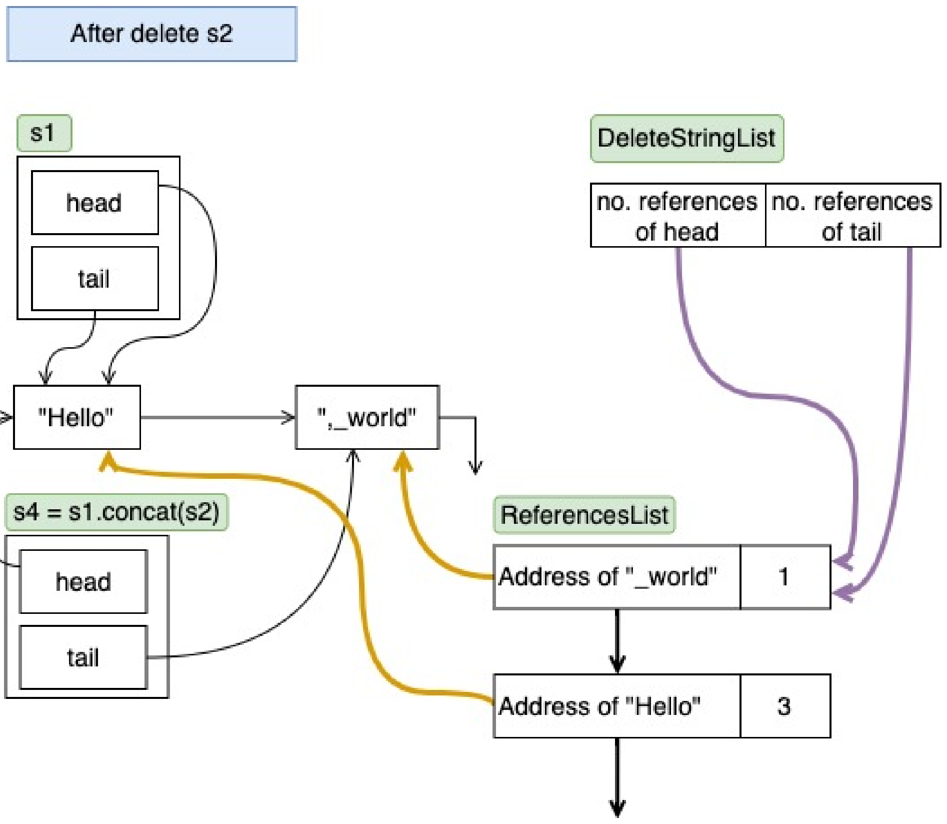

Figure 8: Illustration of ReferenceList after deleting s2

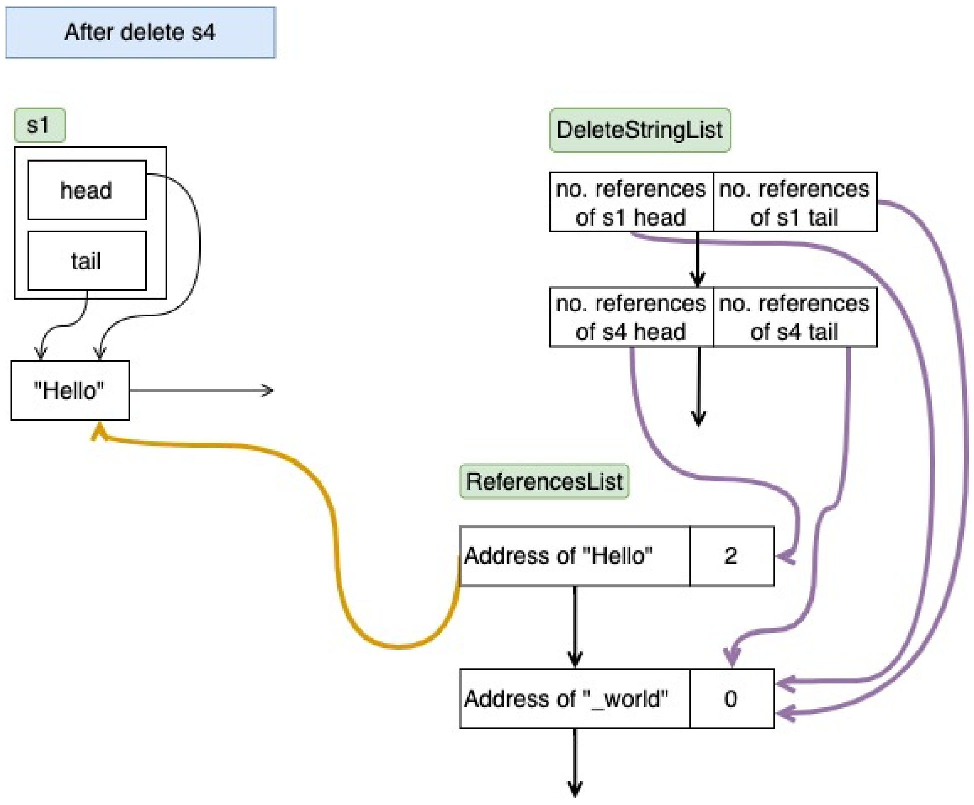

Figure 9: Illustration of ReferenceList after deleting s4

Example 3.6

Calling refCountAt(1) with the reference list in Figure 7 returns 3.
Calling refCountAt(0) with the reference list in Figure 9 returns 1.

3. **string refCountsString() const**

- Returns the string representation of the reference counts in the reference list.
- Complexity: $O(n)$.

Example 3.7

Calling refCountsString() with the reference list in Figure 8 returns **"RefCounts[1,3]"**.
Calling refCountsString() with the reference list in Figure 9 returns **"RefCounts[2]"**.

Methods to implement for **class DeleteStringList**:

1. **int size() const**

- Returns the number of nodes in the list.
- Complexity: O(1).

Example 3.8

Calling size() with DeleteStringList in Figure 7 returns 0.
Calling size() with DeleteStringList in Figure 9 returns 1.

2. **string totalRefCountsString() const**

- Returns the string representing the totals of the nodes' reference counts.
- Complexity: O(n).

Example 3.9

Calling totalRefCountsString() with the list in Figure 8 returns:
"TotalRefCounts[2]"
Calling totalRefCountsString() with the list in Figure 9 returns:
"TotalRefCounts[2]"

3.4 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, ConcatStringList.h, ConcatStringList.cpp and sample_output folder. Students will only submit 2 files, ConcatStringList.h and ConcatStringList.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 ConcatStringList.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 ConcatStringList.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 ConcatStringList.h, ConcatStringList.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 ConcatStringList.h file **#include "main.h"** and one directive in the ConcatStringList.cpp file **#include "ConcatStringList.h"**. Apart from the above directives, no other **#include** is allowed in these files.
6. There are some simple testcases in the main.cpp file in the format **"tc<x>()"**. The output of the function **"tc<x>()"** is recorded in the file **"tc<x>.txt"** in the directory **sample_output**.
7. Students are allowed to write additional methods and properties in classes that are required to be implemented.
8. Students are required to design and use data structures based on learned lists.
9. The student must free all the dynamically allocated memory when the program terminates.

4 Submission

Students submit only 2 files: ConcatStringList.h and ConcatStringList.cpp, before the deadline given in the link "Assignment 1 - 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.

END