ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Cấu trúc dữ liệu và giải thuật - CO2003

Bài tập lớn 2

XÂY DỰNG CONCAT_STRING BẰNG CẤU TRÚC CÂY VÀ HASH

Tác giả: Vũ Văn Tiến

TP. HỒ CHÍ MINH, THÁNG 10/2022

ĐẶC TẢ BÀI TẬP LỚN

Phiên bản 1.1

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên ôn lại và sử dụng thành thục:

- Lập trình hướng đối tượng.
- Cấu trúc dữ liệu cây nhị phân tìm kiếm và cây AVL.
- Cấu trúc dữ liệu Hash.

2 Dẫn nhập

Trong Bài tập lớn 1, SV được yêu cầu hiện thực chuỗi ký tự bằng cấu trúc dữ liệu danh sách để giảm độ phức tạp của thao tác nối chuỗi. Cách hiện thực này có một số hạn chế: thao tác truy cập một ký tự tại một vị trí có độ phức tạp cao; hoặc một chuỗi chỉ được tham gia vào thao tác nối một lần.

Trong Bài tập lớn (BTL) này, sinh viên được yêu cầu sử dụng cấu trúc cây và Hash để hiện thực chuỗi ký tự và giải quyết các hạn chế trên. Class biểu diễn cho chuỗi ký tự cần hiện thực trong BTL có tên là ConcatStringTree.

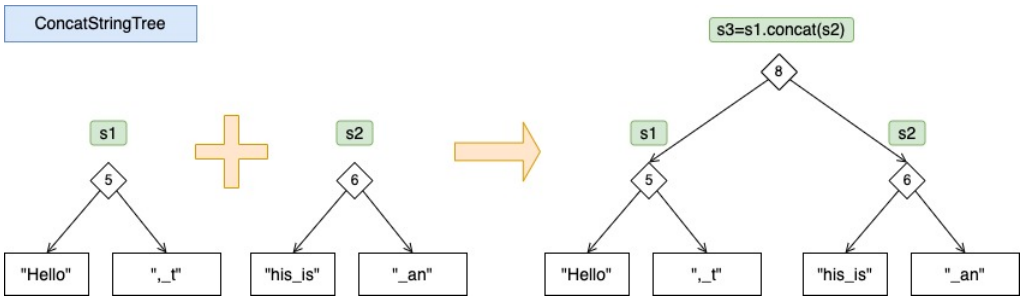
3 Mô tả

3.1 Tổng quan

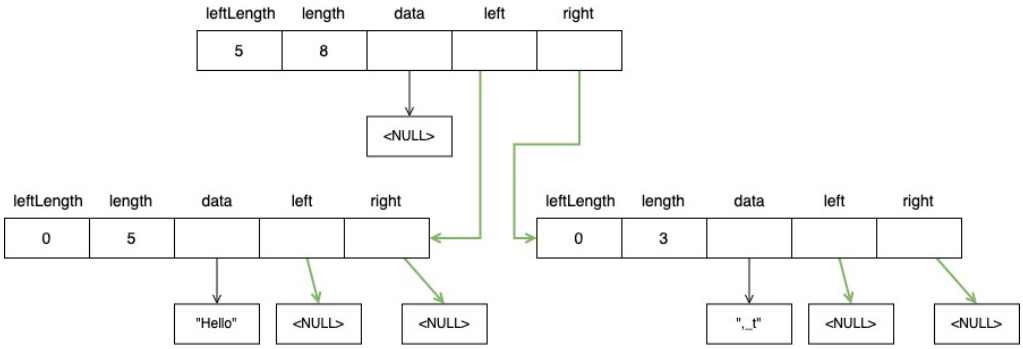
Hình 1 có minh hoạ 2 chuỗi s1, s2 bằng cấu trúc cây. Với cấu trúc cây, thao tác nối chuỗi có thể được thực hiện dễ dàng bằng cách tạo một node mới có cây con trái là s1 và cây con phải là s2.

Bên cạnh đó, để hỗ trợ thao tác truy cập bằng vị trí, cấu trúc cây được lựa chọn là **cây tìm kiếm nhị phân (Binary Search Tree)**. Key tại mỗi node của cây sẽ được chọn là độ dài của chuỗi bên trái. Khi tìm kiếm (với vị trí xác định) tại mỗi node, nếu vị trí đang tìm kiếm nhỏ hơn key thì tiếp tục tìm kiếm ở cây bên trái, ngược lại thì tìm kiếm ở cây bên phải.

Với cách chọn key là độ dài của chuỗi bên trái ta được key của s3 (nối s1 với s2) trong Hình 1 có giá trị là 8. Khi đó, thao tác để tìm ra key bằng 8 có độ phức tạp $O(log(n))$ (SV hãy thử tìm cách tính độ phức tạp này). Để giảm độ phức tạp, ta sẽ lưu thêm thông tin về tổng độ dài của chuỗi ở mỗi node. Hình 2 đề xuất các thông tin của các node trong cây biểu diễn s1, biết rằng s1 là kết quả của thao tác nối giữa 2 chuỗi "Hello" và ",_r".



Hình 1: Minh hoạ thao tác nối chuỗi



Hình 2: Minh hoạ các node của cây biểu diễn chuỗi s1

Từ thông tin lưu trữ như trên, các hình minh hoạ trở về sau sẽ mô tả node lá (chỉ chứa dữ liệu) bằng một hình thoi chứa số 0 (tương tự các node khác, thể hiện độ dài chuỗi bên trái) trỏ thẳng xuống một chuỗi (thể hiện trường data). Các phần sau sẽ mô tả chi tiết về các class cần được hiện thực trong BTL này.

3.2 class ConcatStringTree

Các phương thức cần hiện thực cho class ConcatStringTree:

1. `ConcatStringTree(const char * s)`
   - Khởi tạo đối tượng ConcatStringTree với trường data trỏ đến một đối tượng biểu diễn cho một chuỗi ký tự liên tục có giá trị giống với chuỗi **s**.

- Độ phức tạp (mọi trường hợp): O(n) với n là độ dài của chuỗi **s**.

2. `int length() const`
   - Trả về độ dài của chuỗi đang được lưu trữ trong đối tượng ConcatStringTree.
   - Độ phức tạp (mọi trường hợp): O(1).

Ví dụ 3.1

Trong Hình 1:

- s1.length() trả về 8.
- s2.length() trả về 9.

3. `char get(int index) const`
   - Trả về ký tự tại vị trí ***index***.
   - Ngoại lệ: Nếu ***index*** có giá trị là 1 vị trí không hợp lệ trong chuỗi, ném ra ngoại lệ (thông qua lệnh **throw** trong ngôn ngữ C++): `out_of_range("Index of string is invalid!")`. ***index*** có giá trị là vị trí hợp lệ nếu `index` nằm trong đoạn $[0, l-1]$ với l là độ dài của chuỗi.
   - Độ phức tạp (trường hợp trung bình): O(log(k)) với k là số lượng node của ConcatStringTree.

Ví dụ 3.2

Trong Hình 1:

- s1.get(14) ném ra ngoại lệ `out_of_range("Index of string is invalid!")`
- s2.get(1) trả về ký tự 'i'.

4. `int indexOf(char c) const`
   - Trả về vị trí xuất hiện đầu tiên của ***c*** trong ConcatStringTree. Nếu không tồn tại ký tự c thì trả về giá trị -1.
   - Độ phức tạp (trường hợp tệ nhất): $O(l)$ với l là chiều dài của chuỗi ConcatStringTree.

Ví dụ 3.3

Trong Hình 1:

- s1.indexOf('i') trả về -1.
- s2.indexOf('i') trả về 1.

5. `string toStringPreOrder() const`

- Trả về chuỗi biểu diễn cho đối tượng ConcatStringTree khi duyệt các node một cách tiền thứ tự NLR.
- Độ phức tạp (mọi trường hợp): $O(l)$ với l là chiều dài của chuỗi ConcatStringTree.

Ví dụ 3.4

Trong Hình 1:

- s1.toStringPreOrder() trả về "ConcatStringTree[(LL=5,L=8,<NULL>);(LL=0,L=5,"Hello");(LL=0,L=3,",_t")]"

6. `string toString() const`

- Trả về chuỗi biểu diễn cho đối tượng ConcatStringTree.
- Độ phức tạp (mọi trường hợp): $O(l)$ với l là chiều dài của chuỗi ConcatStringTree.

Ví dụ 3.5

Trong Hình 1:

- s1.toString() trả về `"ConcatStringTree["Hello,_t"]"`
- s2.toString() trả về `"ConcatStringTree["his_is_an"]"`

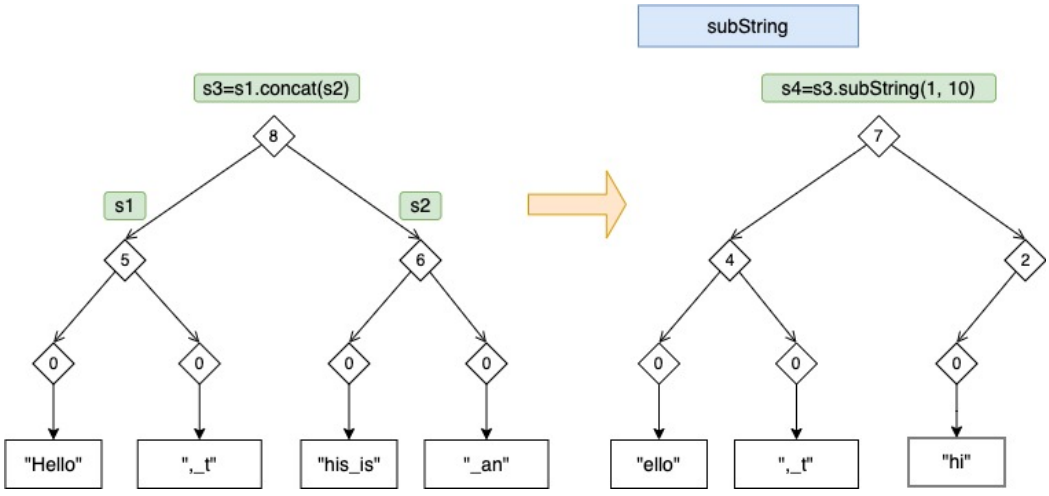
7. `ConcatStringTree concat(const ConcatStringTree & otherS) const`

- Trả về đối tượng ConcatStringTree mới như mô tả ở Mục 3.1.
- Độ phức tạp (mọi trường hợp): $O(1)$
- Ví dụ: Xem lại Hình 1.

8. `ConcatStringTree subString(int from, int to) const`

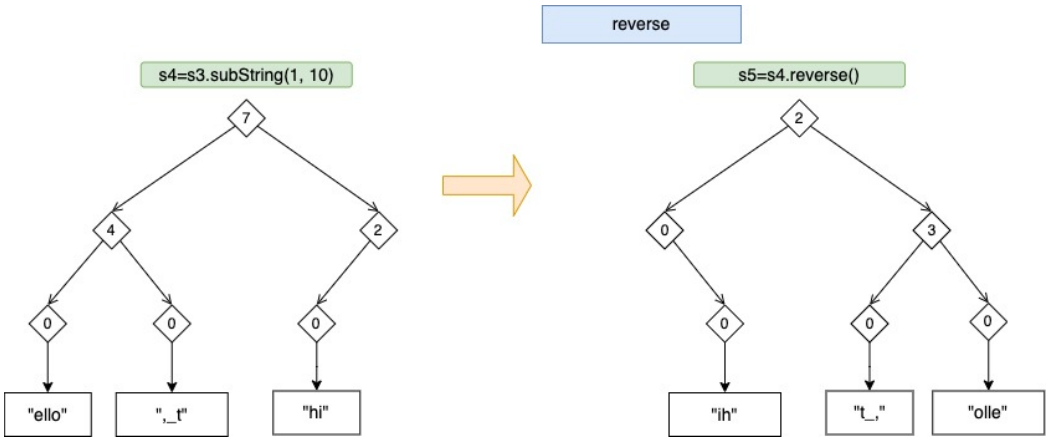
- Trả về đối tượng ConcatStringTree mới chứa các ký tự bắt đầu từ vị trí `from` (bao gồm `from`) đến vị trí `to` (không bao gồm `to`).

- Ngoại lệ: Nếu **from** hoặc **to** là một vị trí không hợp lệ trong chuỗi thì ném ra ngoại lệ `out_of_range("Index of string is invalid!")`. Nếu $from >= to$ thì ném ra ngoại lệ `logic_error("Invalid range!")`.
- Ví dụ: Hình 3 minh hoạ cho thao tác subString. Lưu ý: số ghi trong hình thoi là độ dài của chuỗi bên trái.
- Lưu ý: Chuỗi mới cần ***tạo mới*** các node (không sử dụng lại node trong cây gốc) và cố gắng giữ lại liên kết giữa các node giống như cây gốc. Giữ lại liên kết ở đây có nghĩa là giữ lại liên kết một cách đầy đủ, nhỏ nhất từ node gốc đến các node lá chứa ký tự nằm trong khoảng subString. Những node lá có ký tự nằm ngoài khoảng này sẽ không được bao gồm trong đối tượng mới.



Hình 3: Minh hoạ chuỗi sau khi thực hiện thao tác subString

9. `ConcatStringTree reverse() const`
   - Trả về đối tượng ConcatStringTree mới biểu diễn một chuỗi nghịch đảo của chuỗi gốc.
   - Các node là con bên phải của đối tượng ConcatStringTree cũ sẽ thành con bên trái của đối tượng mới và ngược lại.
   - Ví dụ: Hình 4 minh hoạ thao tác reverse.
10. `~ConcatStringTree()`
    - Hiện thực hàm huỷ để tất cả vùng nhớ được cấp phát động phải được thu hồi sau khi chương trình kết thúc. Xem xét s1 trong Hình 1, thông thường, nếu xoá s1 thì các node chứa chuỗi "Hello" và ",_r" sẽ bị xoá. Điều này không phù hợp vì 2 chuỗi này vẫn cần đến cho sự tạo thành của chuỗi 3. Tham khảo Mục 3.3 để hiện thực hàm hủy cho phù hợp.



Hình 4: Minh hoạ thao tác reverse

3.3 class ParentsTree

Xem xét lại Hình 1 minh hoạ kết quả của thao tác nối 2 chuỗi. Khi muốn xoá s1, nếu ta xoá toàn bộ các node trong cây s1 thì sẽ làm mất các chuỗi "Hello" và ",_r". Từ đó làm mất thông tin để biểu diễn cho cây s3. Để giải quyết vấn đề này, tại mỗi node, ta sẽ lưu trữ các node cha liền kề có trỏ đến node này. Khi thực hiện xoá qua các node, ta có thể kiểm tra nếu node hiện tại không có node cha nào thì thực hiện xoá node này và các node trong cây con.

Sinh viên được yêu cầu sử dụng cấu trúc cây AVL để lưu trữ thông tin về các node cha trong mỗi node của ConcatStringTree. Để sử dụng AVL, ta cần chọn một key tương ứng cho một node. SV cần hiện thực một cơ chế sinh ra id đơn giản và dùng id này làm key cho node. Cơ chế sinh ra id như sau:

- Node đầu tiên được tạo ra có id là giá trị 1.
- Các node sau đó được sinh ra có id bằng với id lớn nhất đã cấp trước đó cộng thêm 1.
- Id chỉ được cấp tối đa 10^7 giá trị. Nếu id cấp cho node lớn hơn 10^7, ném ra ngoại lệ **overflow_error("Id is overflow!")**.

Class ParentsTree biểu diễn cây AVL được dùng để lưu trữ các node cha của node hiện tại. Trong ParentsTree, nếu thực hiện thao tác xoá node và node này có 2 cây con trái và phải, ta sẽ lấy node **lớn nhất của cây con bên trái** để thay thế cho node hiện tại.

Khi xoá một node trong ConcatStringTree ta sẽ xoá thông tin về node đó ra khỏi ParentsTree của node gốc thuộc cây con bên trái và node gốc thuộc cây con bên phải. Nếu ParentsTree của node là rỗng thì node đó không được sử dụng để tạo thành cây lớn hơn, ta có thể xoá node đó và node cây con.

Mặt khác, trong Hình 1, nếu xoá s3 (trước s1 và s2) thì sẽ xoá s1 và s2, thông tin biểu

diễn cho s1 và s2 sẽ bị mất. Để giải quyết vấn đề này, ta sẽ thêm giá trị vào ParentsTree để khi quá trình xoá s3 truyền đến s1 thì ParentsTree không bị rỗng và sẽ không tiếp tục quá trình xoá. Cụ thể, khi tạo node của ConcatStringTree, ta sẽ thêm id của node vào ParentsTree. Đồng thời, sinh viên cần lưu ý cập nhật vào ParentsTree mỗi lần thực hiện nối chuỗi.

SV cần hiện thực các phương thức sau cho **class ParentsTree** (đây là các phương thức được sử dụng trong testcases, SV tự hiện thực các phương thức khác nếu cần):

1. **int size() const**
   - Trả về số node trong ParentsTree.
   - Độ phức tạp: O(1).

   Ví dụ 3.6

   Trong Hình 1, sau khi áp dụng concat:
   - Gọi size() với đối tượng ParentsTree trong node gốc của s1 trả về 2.
   - Gọi size() với đối tượng ParentsTree trong node có data "Hello" của s1 trả về 2.

2. **string toStringPreOrder() const**
   - Trả về chuỗi biểu diễn cho đối tượng ParentsTree khi duyệt các node theo thứ tự tiền tố NLR. Chuỗi biểu diễn có định dạng sau:
     "ParentsTree[<node_list>]"
     Với <node_list> là danh sách các node được phân cách nhau bởi 1 ký tự chấm phẩy. Định dạng của 1 node là:
     "(id=<node_id>)"
     Với <node_id> là id của node.

     Ví dụ 3.7

     Ví dụ kết quả của thao tác toStringPreOrder() có
     - 0 node (rỗng):
       `"ParentsTree[]"`
     - 1 node:
       `"ParentsTree[(id=1)]"`
     - 2 node:
       `"ParentsTree[(id=2);(id=3)]"`

SV phải hiện thực thêm phương thức sau cho **class ConcatStringTree**:

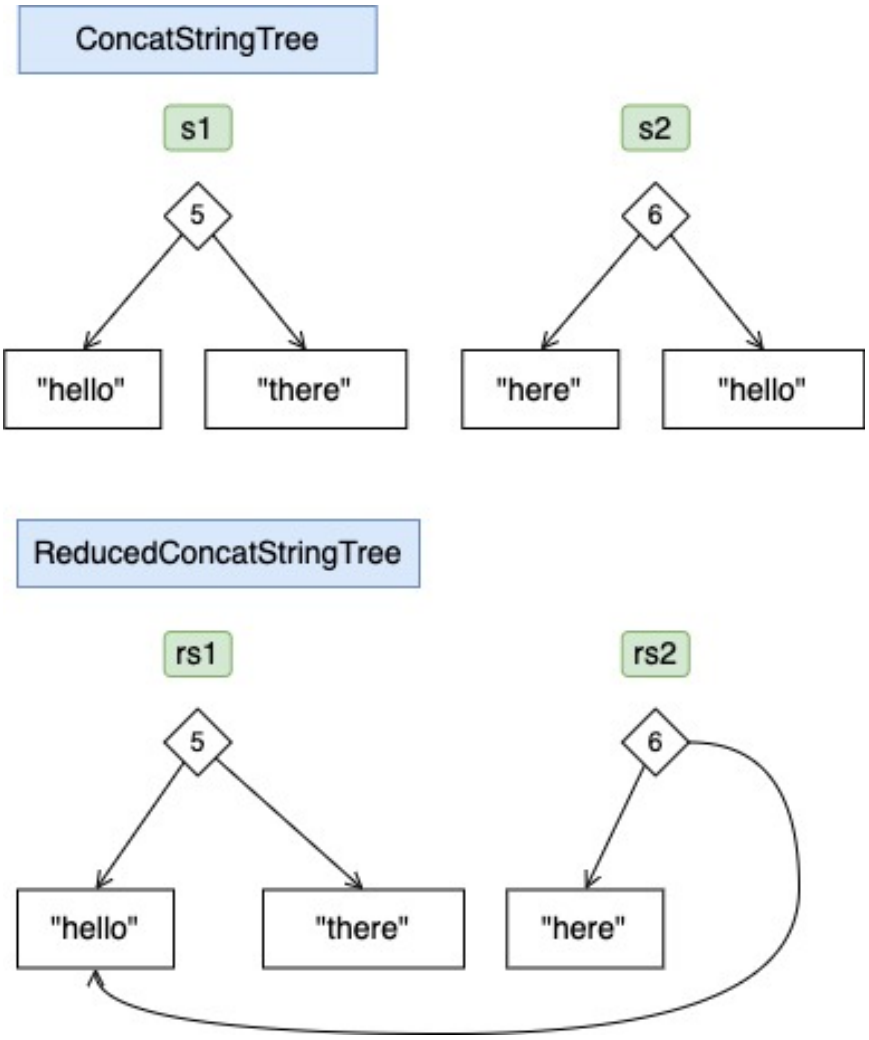
1. **int getParTreeSize(const string & query) const**
   - Trả về số lượng node trong ParentsTree ở một node xác định trong ConcatStringTree.
   - Chuỗi **query** được dùng để xác định node muốn truy cập ParentsTree. **query** chỉ gồm các ký tự thuộc 2 ký tự 'l' hoặc 'r', ngược lại ném ra ngoại lệ: **runtime_error("Invalid character of query")**. Cách xác định dựa vào query: Ban đầu ta ở node gốc của ConcatStringTree. Lần lượt duyệt qua các ký tự trong **query**, nếu gặp ký tự 'l', ta di chuyển đến cây con bên trái, nếu gặp ký tự 'r', ta di chuyển đến cây con bên phải. Nếu trong quá trình thực hiện, ta gặp địa chỉ NULL thì ném ra ngoại lệ: **runtime_error("Invalid query: reaching NULL")**.
2. **string getParTreeStringPreOrder(const string & query) const**
   - Trả về chuỗi biểu diễn cho đối tượng ParentsTree ở một node xác định trong ConcatStringTree. Chuỗi biểu diễn này là kết quả trả về của phương thức **toStringPreOrder** của class ParentsTree.
   - Chuỗi **query** được dùng để xác định node muốn truy cập ParentsTree. Quy tắc và cách truy cập của **query** giống như mô tả trong phương thức **getParTreeSize**.

3.4 class ReducedConcatStringTree và class LitStringHash

Xem xét Hình 5, chuỗi s1 và s2 là các đối tượng của class ConcatStringTree. Ta thấy chuỗi "hello" xuất hiện 2 lần: cây con bên trái của s1 và cây con bên phải của s2. Trong mục này, ta sẽ tìm cách giảm kích thước lưu trữ chuỗi bằng cách trỏ dữ liệu đến vùng nhớ đã tồn tại nếu chuỗi tạo mới giống với một trong các chuỗi đã tạo (như chuỗi s2). Class biểu diễn chuỗi với khả năng giảm vùng nhớ lưu trữ được gọi là ReducedConcatStringTree. ReducedConcatStringTree cần có tất cả các thuộc tính và phương thức của class ConcatStringTree. kế thừa của concatstringtree

Gọi các chuỗi chứa dữ liệu thật sự là LitString. Trong hình 5, "hello", "there", "here" là các LitString; node <5> và node <6> không phải là LitString. Ta sẽ dùng cấu trúc Hash lưu trữ các LitString, gọi class biểu diễn cấu trúc này là LitStringHash.

- Khi một chuỗi được tạo ra, nếu LitString đó đã tồn tại trong LitStringHash thì ta trỏ data đến địa chỉ của LitString này (không tạo ra LitString mới). Ngược lại, nếu LitString chưa tồn tại trong LitStringHash thì ta thêm (insert) LitString mới này vào LitStringHash. Mỗi LitString trong LitStringHash nên có thêm thông tin về số liên kết đến nó.



Hình 5: Minh hoạ giảm vùng nhớ cho chuỗi

- Khi xoá một chuỗi, nếu LitString nào đó không còn liên kết nào đến nó nữa thì ta sẽ xoá nó ra khỏi LitStringHash. Nếu sau khi xoá LitStringHash trở thành rỗng thì ta cần thu hồi vùng nhớ cấp phát cho LitStringHash. Ở các thao tác sau đó (nếu có), ta cần cấp phát lại vùng nhớ cho LitStringHash.

Class LitStringHash có các đặc điểm sau:

- Giả sử **s** là một LitString. Hàm băm của LitStringHash như sau:

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

Trong đó:

  - n là chiều dài của chuỗi
  - s[0], s[1],..., s[n-1] là lần lượt là số nguyên biểu diễn mã ASCII tương ứng của ký tự tại vị trí 0, 1,..., n-1

  - m là kích thước của bảng hash.
  - p là thông số cấu hình sẽ được mô tả sau.

- Hàm dò tìm: sử dụng phương pháp dò tìm bậc 2 (quadratic probing):

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

  Trong đó:

  - m là kích thước của bảng hash.
  - c1, c2 là thông số cấu hình sẽ được mô tả sau.

  Lưu ý: Nếu không tìm thấy vị trí trống để thêm vào thì ném ra ngoại lệ:
  runtime_error("No possible slot")

- Rehashing: Sau khi thêm 1 giá trị vào LitStringHash, nếu LitStringHash có hệ số tải (load factor) lớn hơn *lambda* thì ta cần thực hiện rehash. Hệ số tải được tính bằng tỉ lệ giữa số phần tử đang có và kích thước của LitStringHash. *lambda* là một thông số cấu hình. Các bước thực hiện rehash:
  - Tạo LitStringHash mới với kích thước alpha * size (làm tròn xuống) với size là kích thước hiện tại của LitStringHash.
  - Lần lượt lặp qua các vị trí của LitStringHash, nếu vị trí nào đó có LitString thì ta chèn nó vào LitStringHash mới.

- Class HashConfig: class này chứa các thông số cấu hình cho các thao tác ở trên, gồm có:
  - p: là một số nguyên sử dụng trong hàm băm.
  - c1, c2: là các số thực (kiểu double) sử dụng trong hàm dò tìm.
  - lambda: là số thực (kiểu double) thể hiện tỉ lệ mà nếu hệ số tải lớn hơn nó sẽ thực hiện rehashing.
  - alpha: là số thực (kiểu double) thể hiện tỉ lệ nhân với kích thước cũ để tạo ra kích thước mới lớn hơn.
  - initSize: là số nguyên, thể hiện kích thước của LitStringHash khi vừa khởi tạo hoặc khi thêm một giá trị mới sau khi vừa thu hồi vùng nhớ cho LitStringHash.

SV cần hiện thực các phương thức sau cho **class LitStringHash**:

1. **LitStringHash(const HashConfig & hashConfig)**
   - Nhận vào đối tượng **hashConfig** và khởi tạo các tham số cấu hình cần thiết cho thao tác băm.
2. **int getLastInsertedIndex() const**

- Trả về vị trí cuối cùng được thêm vào LitStringHash và sau khi thao tác rehash được thực hiện (nếu có). Nếu trước đó chưa có phần tử nào được chèn, hoặc vừa thực hiện thu hồi vùng nhớ cho LitStringHash thì trả về giá trị -1.
- Độ phức tạp (mọi trường hợp): O(1).

3. **string toString() const**

- Trả về chuỗi biểu diễn cho đối tượng LitStringHash. Chuỗi biểu diễn có định dạng sau:
  "LitStringHash[<slot_list>]"
  Với <slot_list> là danh sách các phần tử được phân cách nhau bởi 1 ký tự chấm phẩy. Định dạng của 1 phần tử là:
  - Nếu vị trí đó không có giá trị:
    "()"
  - Nếu vị trí đó có giá trị:
    "(litS=<lit_string>)" Với <lit_string> là chuỗi biểu diễn LitString.

Ví dụ 3.8

Ví dụ về 1 kết quả của thao tác toString()

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

SV cần hiện thực phương thức sau cho **class ReducedConcatStringTree**:

1. **ReducedConcatStringTree(const char * s, LitStringHash * litStringHash)**

- Khởi tạo đối tương ReducedConcatStringTree với trường data trỏ đến một đối tượng biểu diễn cho một chuỗi ký tự liên tục có giá trị giống với chuỗi s (giống như phương thức khởi tạo của ConcatStringTree). Đồng thời, khởi tạo một con trỏ đến đối tượng LitStringHash. Đối tượng này sẽ lưu trữ các thông tin của bảng băm dùng chung cho các đối tượng ReducedConcatStringTree.
- Độ phức tạp (mọi trường hợp): O(n) với n là độ dài của chuỗi s.

3.5 Yêu cầu

Để hoàn thành bài tập lớn này, sinh viên phải:

1. Đọc toàn bộ tập tin mô tả này.

2. Tải xuống tập tin initial.zip và giải nén nó. Sau khi giải nén, sinh viên sẽ nhận được các tập tin: main.cpp, main.h, ConcatStringTree.h, ConcatStringTree.cpp và thư mục sample_output. Sinh viên sẽ chỉ nộp 2 tập tin là ConcatStringTree.h và ConcatStringTree.cpp nên không được sửa đổi tập tin main.h khi chạy thử chương trình.
3. Sinh viên sử dụng câu lệnh sau để biên dịch:
   g++ -o main main.cpp ConcatStringTree.cpp -I . -std=c++11
   Câu lệnh trên được dùng trong command prompt/terminal để biên dịch chương trình. Nếu sinh viên dùng IDE để chạy chương trình, sinh viên cần chú ý: thêm đầy đủ các tập tin vào project/workspace của IDE; thay đổi lệnh biên dịch của IDE cho phù hợp. IDE thường cung cấp các nút (button) cho việc biên dịch (Build) và chạy chương trình (Run). Khi nhấn Build IDE sẽ chạy một câu lệnh biên dịch tương ứng, thông thường, chỉ biên dịch file main.cpp. Sinh viên cần tìm cách cấu hình trên IDE để thay đổi lệnh biên dịch: thêm file ConcatStringTree.cpp, thêm option -std=c++11, -I .
4. Chương trình sẽ được chấm trên nền tảng Unix. Nền tảng và trình biên dịch của sinh viên có thể khác với nơi chấm thực tế. Nơi nộp bài trên BKeL được cài đặt giống với nơi chấm thực tế. Sinh viên phải chạy thử chương trình trên nơi nộp bài và phải sửa tất cả các lỗi xảy ra ở nơi nộp bài BKeL để có đúng kết quả khi chấm thực tế.
5. Sửa đổi các file ConcatStringTree.h, ConcatStringTree.cpp để hoàn thành bài tập lớn này và đảm bảo hai yêu cầu sau:

   - Tất cả các phương thức trong mô tả này đều phải được hiện thực để việc biên dịch được thực hiện thành công. Nếu sinh viên chưa thể hiện thực được phương thức nào, hãy cung cấp một hiện thực rỗng cho phương thức đó. Mỗi testcase sẽ gọi một số phương thức đã mô tả để kiểm tra kết quả trả về.
   - Chỉ có 1 lệnh **include** trong tập tin ConcatStringTree.h là **#include "main.h"** và một include trong tập tin ConcatStringTree.cpp là **#include "ConcatStringTree.h"**. Ngoài ra, không cho phép có một **#include** nào khác trong các tập tin này.

6. Sinh viên được phép viết thêm các phương thức và thuộc tính khác trong các class được yêu cầu hiện thực.
7. Sinh viên được yêu cầu thiết kế và sử dụng các cấu trúc dữ liệu dựa trên các loại danh sách đã học.
8. Sinh viên phải giải phóng toàn bộ vùng nhớ đã xin cấp phát động khi chương trình kết thúc.

4 Nộp bài

Sinh viên chỉ nộp 2 tập tin: ConcatStringTree.h và ConcatStringTree.cpp, trước thời hạn được đưa ra trong đường dẫn "Assignment 2 - Submission". Có một số testcase đơn giản được sử dụng để kiểm tra bài làm của sinh viên nhằm đảm bảo rằng kết quả của sinh viên có thể biên dịch và chạy được. Sinh viên có thể nộp bài bao nhiêu lần tùy ý nhưng chỉ có bài nộp cuối cùng được tính điểm. Vì hệ thống không thể chịu tải khi quá nhiều sinh viên nộp bài cùng một lúc, vì vậy sinh viên nên nộp bài càng sớm càng tốt. Sinh viên sẽ tự chịu rủi ro nếu nộp bài sát hạn chót. Khi quá thời hạn nộp bài, hệ thống sẽ đóng nên sinh viên sẽ không thể nộp nữa. Bài nộp qua các phương thức khác đều không được chấp nhận.

5 Một số quy định khác

- Sinh viên phải tự mình hoàn thành bài tập lớn này và phải ngăn không cho người khác đánh cắp kết quả của mình. Nếu không, sinh viên sẽ bị xử lý theo quy định của trường vì gian lận.
- Mọi quyết định của giảng viên phụ trách bài tập lớn là quyết định cuối cùng.
- Sinh viên không được cung cấp testcase sau khi chấm bài mà chỉ được cung cấp thông tin về chiến lược thiết kế testcase và phân bố số lượng sinh viên đúng theo từng testcase.
- Nội dung Bài tập lớn sẽ được Harmony với một câu hỏi trong bài kiểm tra với nội dung tương tự.

6 Thay đổi so với phiên bản trước

- Bổ sung ghi chú cho hình minh hoạ ConcatStringTree.
- Bổ sung mô tả giữ lại liên kết cho phương thức subString.
- Bổ sung mô tả cho phương thức reverse.
- Sửa lại kết quả ví dụ 3.6 (cho phương thức size).