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

Bài tập lớn 2 - Phần 1

BKU TREE

TP. HỒ CHÍ MINH, THÁNG 11/2020

ĐẶC TẢ BÀI TẬP LỚN 2 (Phần 1)

Phiên bản 1.1

1 Chuẩn đầu ra

Sau khi hoàn thành bài tập lớn này, sinh viên sẽ có khả năng:

- Hiện thực thành thạo một cấu trúc dữ liệu cây bằng ngôn ngữ lập trình C/C++.
- Hiểu và thao tác được hai loại cây nhị phân tìm kiếm là AVL và Splay Tree.

2 Dẫn nhập

Cây nhị phân tìm kiếm là nhóm các cây có khả năng lưu trữ để tìm kiếm các phần tử với độ phức tạp trung bình $O(logN)$ với N là số điểm dữ liệu đã lưu. Trong đó, AVL thuộc nhóm cây cân bằng với độ phức tạp khi tìm kiếm trong mọi trường hợp đều là $O(logN)$ nhờ vào các toán tử tự điều chỉnh chiều cao để tại mỗi nút đều thỏa mãn tính chất về sự chêch lệch của các cây con.

Tuy nhiên, nếu có những điểm dữ liệu thường xuyên được tìm kiếm và luôn nằm ở nút lá trên cây AVL thì việc sử dụng cấu trúc AVL để lưu trữ lại tỏ ra kiếm hiệu quả. Splay Tree ra đời để giải quyết bài toán trên với đặc điểm tự đưa những nút thường xuyên được tìm kiếm lên gần nút gốc giúp cho việc tìm kiếm những điểm dữ liệu này lại trở nên hiện quả.

Điểm bất lợi lớn nhất của Splay Tree là không cân bằng về chiều cao nên nếu việc tìm kiếm trên diễn ra ngẫu nhiên thì việc tìm kiếm trên Splay Tree thiếu hiệu quả do các thao tác điều chỉnh cây liên tục xảy ra. Nếu có một ứng dụng đòi hỏi tại những giai đoạn khác nhau thì chuỗi các giá trị tìm kiếm sẽ ngẫu nhiên hoặc tập trung vào một vài điểm dữ liệu thì cả hai loại cây trên đều gây bất lợi trong các giai đoạn khác nhau.

Vì vậy, cấu trúc dữ liệu cây **BKU Tree** được đưa ra trong bài tập lớn này như một ý tưởng để giải quyết sự thiếu hiệu quả về mặt thời gian cho cả hai loại cây trên.

3 Cây nhị phân tìm kiếm BKU Tree

Cây nhị phân tìm kiếm BKU Tree là một loại cây trong đó mang cả hai cây AVL và cây Splay trong đó để giúp cho chương trình vừa có thể tìm kiếm như trên một AVL Tree, vừa có thể tìm

kiếm như trên một Splay Tree. Cụ thể, trong cấu trúc của BKU Tree sẽ lưu trữ như sau:

1. Cây Splay.
2. Cây AVL.
3. Một hàng đợi các khóa tìm kiếm gần nhất (Số lượng phần tử tối đa của hàng đợi được người dùng khởi tạo).

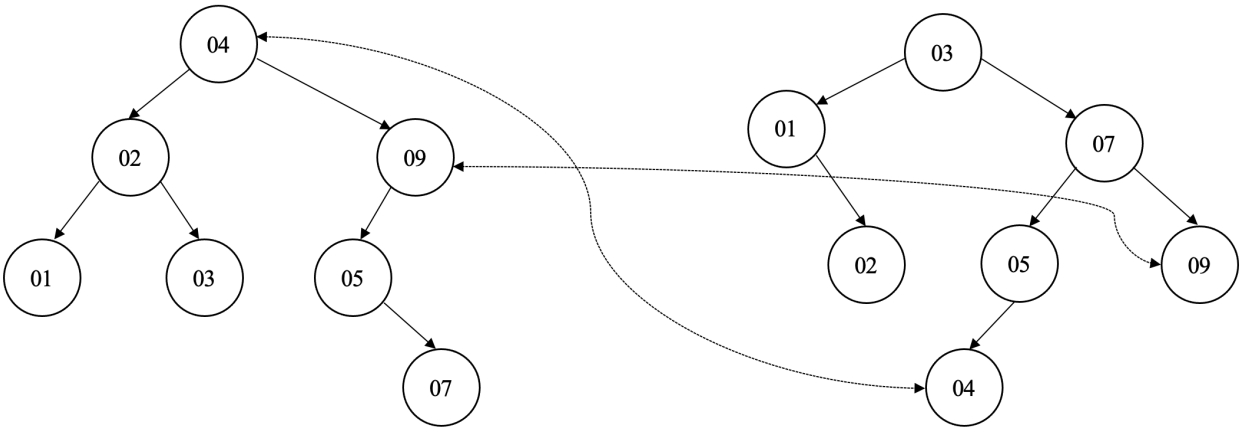
Trong đó, mỗi node trên cây Splay đều lưu địa chỉ của (hay có môt con trỏ trỏ đến) node tương ứng trên cây AVL và ngược lại.

3.1 Thao tác thêm (Insertion) và thao tác xóa (Deletion)

Khi diễn ra thao tác thêm và thao tác xóa trên BKU Tree, thao tác tương ứng phải được diễn ra trên cây Splay và cây AVL.

- Trong trường hợp thêm: Khóa được thêm phải được thêm vào hàng đợi, nhưng nếu hàng đợi đã đầy thì ta xóa một phần tử ở đầu hàng đợi (dequeue) và thêm khóa mới thêm vào.
- Trong trường hợp xóa: Nếu trong hàng đợi các khóa tìm kiếm gần nhất có tồn tại khóa này, ta phải xóa khóa đó ra khỏi hàng đợi và thêm khóa hiện tại nằm tại gốc của cây Splay (sau khi xóa).

Ví dụ: Lần lượt thêm các nút mang giá trị khóa (là số nguyên) là [1, 3, 5, 7, 9, 2, 4] vào một BKUTree (kích thước hàng đợi là 5) ta được cây theo mô hình sau:



Hình 1: Kết quả sau khi thêm các phần tử [1, 3, 5, 7, 9, 2, 4] vào BKU Tree

Lưu ý: trong hình vẽ trên để tránh rối hình, người ta chỉ vẽ sự tương ứng của một vài nút, sinh viên phải tự hiểu rằng, các nút có mang cùng giá trị sẽ có sự lưu trữ địa chỉ để kết nối.

3.2 Thao tác tìm kiếm (Search)

Khi tìm kiếm một phần tử mang khóa k trên BKU Tree, ta thực hiện theo các bước sau:

1. Nếu khóa đó ở tại gốc của Splay Tree, ta trả về kết quả tìm thấy và kết thúc.
2. Nếu khóa đó nằm trong hàng đợi, ta thực hiện tìm kiếm trên Splay Tree. Thực hiện thao tác splay đúng **1 lần** tại nút vừa được tìm thấy (không đệ quy lại).
3. Nếu khóa không nằm trong hàng đợi, ta thực hiện theo các bước sau:
   (a) Từ nút gốc r của Splay Tree, ta tham khảo đến nút tương ứng r' trên AVL Tree.
   (b) Thực hiện tìm kiếm trên cây AVL con mà nút r' là nút gốc.
   (c) Nếu tìm thấy ở cây AVL ở bước (b), trả về nút cần tìm đã được tìm kiếm ở nút f'.
   (d) Nếu không tìm thấy, ta thực hiện tìm kiếm từ gốc của cây AVL tổng. Trong quá trình tìm kiếm, nếu phải đi qua nút tìm kiếm ở nút r' thì trả về kết quả là không tìm thấy. Ngược lại tìm thấy nút cần tìm được là nút f'.
   (e) Từ nút f' của AVL Tree, ta tham khảo đến nút tương ứng f trên Splay Tree. Thực hiện thao tác **1 lần** splay tại nút f để đưa nút này gần nút gốc hơn (không đệ quy lại).
4. Đưa khóa vừa tìm kiếm vào cuối hàng đợi (enqueue) (kể cả đã tồn tại trong hàng đợi). Trong trường hợp hàng đợi quá tải, ta xóa khóa ở đầu hàng đợi (dequeue) ra khỏi trước khi thêm khóa này vào.

4 Nhiệm vụ

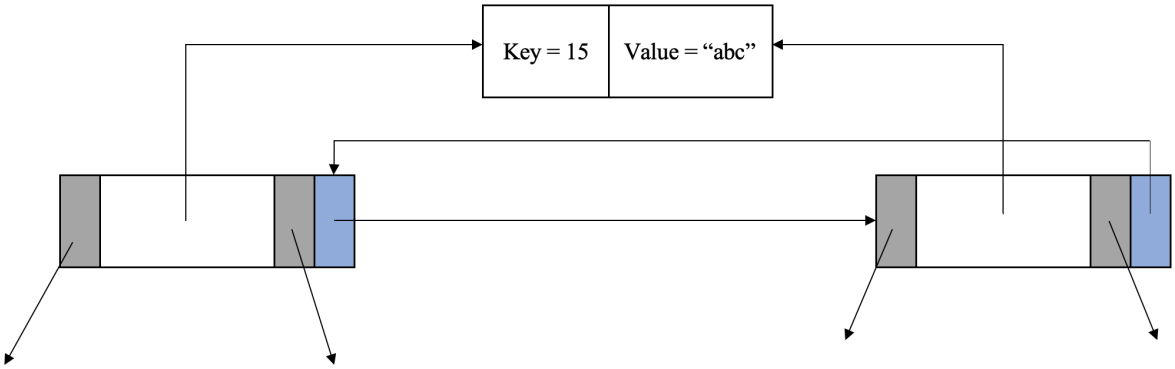
Sinh viên được yêu cầu xây một chương trình trên C++ để hiện thực cấu trúc dữ liệu BKU Tree được đề xuất ở trên. Sinh được được cung cấp hai file 201.dsa-a1p1-des.pdf để mô tả bài tập lớn và file BKUTree.cpp chứa code khởi tạo, trong đó:

- Lớp `BKUTree` (với template `K` và template `V`) thể hiện cấu trúc dữ liệu BKUTree đã đề cập với các thuộc tính (attributes) và phương thức (methods):
  - Thuộc tính (attributes):
    1. `AVLTree* avl` dùng để lưu trữ cây AVL.
    2. `SplayTree* splay` dùng để lưu trữ cây Splay.
    3. `queue<K>* keys` dùng để lưu trữ hàng đợi các khóa được tìm kiếm gần đây với `queue` là cấu trúc hàng đợi trong STL C++.
  - Phương thức (methods):

1. `BKUTree(int maxNumOfKeys = 5)`: hàm dựng để thiết lập số khóa tối đa có thể lưu trữ trong hàng đợi. Nếu người dùng không truyền vào, số khóa tối đa sử dụng là 5.
2. `void add(K key, V value)`: thêm một phần tử với khóa `key` và giá trị `value` vào cấu trúc BKU Tree. Nếu khóa đã tồn tại, chương trình xuất ra một ngoại lệ "Duplicate key".
3. `void remove(K key)`: xóa một phần tử mang khóa `key` ra khỏi cấu trúc dữ liệu BKU Tree. Nếu khóa không tồn tại, chương trình xuất ra một ngoại lệ "Not found".
4. `V search(K key, vector<K>& traversedList)`: tìm kiếm một phần tử mang khóa *key* trong cấu trúc dữ liệu BKU Tree. Nếu khóa không tồn tại chương trình xuất ra một ngoại lệ "Not found", ngược lại trả về giá trị tương ứng với khóa đó. Ngoài ra, `traversedList` sẽ lưu trữ các khóa lần lượt đã đi qua cho trước khi tìm gặp phần tử mang khóa cần tìm hoặc cho đến khi kết thúc trong trường hợp không tìm thấy.
5. `traverseNLROnAVL` và `traverseNLROnSplay`: là hai phương thức dùng để duyệt cây tiền thứ tự tương ứng trên hai cây AVL và cây Splay mà BKUTree đang lưu trữ với `func` là con trỏ hàm lưu trữ thao tác xử lý tại mỗi khi duyệt trên từng nút.

- Lớp `Entry` lồng bên trong `BKUTree` là một cấu trúc dùng để lưu trữ khóa và giá trị tương ứng của một điểm dữ liệu.
- Lớp `AVLTree` lồng bên trong `BKUTree` là một cấu trúc dùng để lưu trữ cây AVL với lớp `Node` dùng để lưu trữ một nút trên cây AVL bao gồm các thông tin: địa chỉ của entry đang lưu trữ, địa chỉ của con bên trái và bên phải, chỉ số cân bằng của nút và địa chỉ của nút tương ứng trên cây Splay.
- Lớp `SplayTree` lồng bên trong `BKUTree` là một cấu trúc dùng để lưu trữ cây SplayTree với lớp `Node` dùng để lưu trữ một nút trên cây SplayTree bao gồm các thông tin: địa chỉ của entry đang lưu trữ, địa chỉ của con bên trái và bên phải, chỉ số cân bằng của nút và địa chỉ của nút tương ứng trên cây AVL.
- Các phương thức của `AVLTree` và `SplayTree` mang dữ liệu đầu vào và đầu ra giống nhau:
  1. `void add(K key, V value)`: dùng để thêm một nút trong đó phần entry trỏ đến có chứa khóa `key` và `value`. Nếu khóa đã tồn tại, trả ra một ngoại lệ "Duplicate key".
  2. `void add(Entry* entry)`: dùng để thêm một nút trong đó phần entry trỏ đến chính là entry được truyền vào. Nếu entry này đã tồn tại trong bất kỳ nút nào của cây, trả ra một ngoại lệ "Duplicate key".

3. `void remove(K key)`: xóa một nút ra khỏi cây trong trường hợp không tồn tại nút có mang `key`, trả ra một ngoại lệ "Not found".
4. `V search(K key)`: tìm kiếm và trả về phần giá trị tương ứng của nút mang khóa `key`. Trong trường hợp không tồn tại nút có mang `key`, trả ra một ngoại lệ "Not found".
5. `void traverseNLR(void (*func)(K key, V value))`: dùng để duyệt tiền thứ tự các phần tử trên cây với thao tác khi duyệt được định nghĩa bởi con trỏ hàm `func`.
6. `void clear()`: dùng để xóa và giải phóng hết vùng nhớ đã cấp phát cho cây.



Hình 2: Ví dụ về sự chia sẻ của hai nút trên hai cây khi mang cùng một entry

Trong hình 2, tại mỗi nút, phần màu xám là hai con trỏ đến các cây con bên trái và bên phải, phần màu xanh lưu trữ địa chỉ tương ứng với nút trên cây con lại và phần màu trắng lưu trữ địa chỉ entry đang lưu trữ.

Sinh viên được phép thêm các phương thức phụ trợ, thay đổi kiểu dữ liệu trả về của các phương thức đã được đề cập hiện đang có kiểu trả về là `void` (tức là không được phép thay đổi trên các phương thức `search`) nhưng tên các lớp và các phương thức trên không được phép thay đổi.

5 Chiến lược chấm bài

Chiến lược thiết kế testcase chấm bài sẽ được chia thành 3 phần:

1. Testcase 00 - 24: Kiểm tra tính đúng đắn của cây AVL mà sinh viên đã hiện thực.
2. Testcase 25 - 59: Kiểm tra tính đúng đắn của cây Splay mà sinh viên đã hiện thực.
3. Testcase 60 - 99: Kiểm tra tính đúng đắn của BKUTree mà sinh viên đã hiện thực.

Ngoài ra, sinh viên được yêu cầu phải giải phóng hết vùng nhớ khi chương trình kết thúc được kiểm tra từ testcase 80 - 99. Nếu không giải phóng hết, chương trình của sinh viên được đánh giá là không đạt testcase.

6 Quy định về thắc mắc và nộp bài

Sinh viên sẽ được cung cấp khung code khởi tạo trên site chung của môn học. Sinh viên làm bài trực tiếp trên hệ thống của BKeL và bài làm cuối cùng của sinh viên được sử dụng để chấm điểm.

Sinh viên được giải đáp thắc mắc trên diễn đàn trên site môn học, TUYỆT ĐỐI KHÔNG GỬI EMAIL cho giảng viên phụ trách để đặt câu hỏi.

————————HẾT————————