Đại học quốc gia TP.HCM
Trường đại học công nghệ thông tin

Ngành: KHOA HỌC MÁY TÍNH

CS523. CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT NÂNG CAO

CẤU TRÚC DỮ LIỆU TREAP

Sinh viên:
Trần Thị Cẩm Tú - 23521704
Mai Lê Bá Vương - 23521821

Giảng viên:
Ths Nguyễn Thanh Sơn

Hồ Chí Minh, Tháng 6/2025

Contents

1 GIỚI THIỆU **4**

2 LÝ THUYẾT VỀ TREAP **4**

2.1 Nhắc lại 4

2.2 Định nghĩa 5

2.3 Tính chất 6

3 CẤU TRÚC DỮ LIỆU **7**

4 CÁC THAO TÁC CƠ BẢN **7**

4.1 Xoay cây 7

4.2 Thêm phần tử 7

4.3 Xóa phần tử 8

4.4 Tìm kiếm 8

5 MINH HỌA **8**

5.1 Thao tác chèn 8

5.2 Thao tác xóa 10

5.3 Thao tác Merge và Split 10

6 ƯU, NHƯỢC ĐIỂM CỦA TREAP **11**

6.1 Ưu điểm 11

6.2 Nhược điểm 11

7 BIẾN THỂ CỦA TREAP **11**

7.1 Implicit Treap 11

7.1.1 Khái niệm 11

7.1.2 Kiểu dữ liệu biểu diễn 12

7.1.3 Thành phần bên trong của mỗi nút 12

7.1.4 Tác dụng của biến thể 12

7.1.5 Thao tác cơ bản 12

7.1.6 Ứng dụng thực tế 13

7.2 Pesistent treap 13

7.2.1 Khái niệm 13

7.2.2 Kiểu dữ liệu biểu diễn 13

7.2.3 Thành phần bên trong của mỗi nút 13

7.2.4 Tác dụng của biến thể 14

7.2.5 Thao tác cơ bản 14

7.2.6 Ứng dụng thực tế 14

8 KẾT LUẬN 14

9 TREAP VISUALIZATION 15

10 TÀI LIỆU THAM KHẢO 16

LỜI CẢM ƠN

Trước tiên, em xin chân thành cảm ơn quý thầy Nguyễn Thanh Sơn, người đã tận tình hướng dẫn em trong suốt quá trình thực hiện đồ án này.

1 GIỚI THIỆU

Cấu trúc dữ liệu là một thành phần cốt lõi trong khoa học máy tính, đặc biệt quan trọng trong việc xây dựng các thuật toán có hiệu suất cao. Trong số đó, cây tìm kiếm nhị phân (Binary Search Tree - BST) là một trong những cấu trúc cơ bản và được sử dụng phổ biến nhất để lưu trữ dữ liệu có tổ chức và cho phép tìm kiếm, chèn, xóa nhanh chóng. Tuy nhiên, nhược điểm chính của BST là khi các phần tử được chèn theo thứ tự tăng hoặc giảm, cây trở nên mất cân bằng và hiệu suất giảm xuống mức tuyến tính O(n).

Để giải quyết vấn đề này, nhiều cấu trúc cây nhị phân tự cân bằng đã được phát triển như cây AVL (Adelson-Velsky and Landis), cây Đỏ-Đen (Red-Black Tree), cây Splay, v.v. Tuy nhiên, các cấu trúc này thường có cài đặt phức tạp, đòi hỏi người lập trình phải quản lý nhiều trường hợp đặc biệt khi thực hiện các thao tác chèn và xóa. Điều này không chỉ làm tăng độ khó trong việc lập trình mà còn ảnh hưởng đến tính linh hoạt của các ứng dụng thời gian thực hoặc hệ thống nhúng có tài nguyên hạn chế.

Treap (kết hợp giữa **Tree** và **Heap**) là một lựa chọn thay thế đáng chú ý nhờ vào sự đơn giản trong cài đặt mà vẫn đạt được hiệu suất trung bình tương đương các cây cân bằng phức tạp như AVL hoặc Red-Black Tree. Ý tưởng chính của Treap là gán cho mỗi nút một giá trị **priority** ngẫu nhiên, qua đó cây tự cân bằng theo xác suất. Điều này cho phép Treap giữ được chiều cao trung bình là O(log n) với độ phức tạp đơn giản hơn nhiều so với các cây cân bằng truyền thống.

2 LÝ THUYẾT VỀ TREAP

2.1 Nhắc lại

Trước khi đi sâu vào cấu trúc dữ liệu Treap, chúng ta cần ôn lại một số kiến thức nền tảng liên quan đến cây nhị phân, Heap, cây nhị phân cân bằng và các thao tác xoay cây – những khái niệm này sẽ giúp làm rõ cơ chế hoạt động của Treap.

Cây nhị phân tìm kiếm (Binary Search Tree - BST) là một loại cây nhị phân trong đó mỗi nút chứa một khóa, các khóa ở cây con bên trái nhỏ hơn khóa của nút cha, còn các khóa ở cây con bên phải thì lớn hơn. BST hỗ trợ các thao tác như tìm kiếm, chèn và xóa với độ phức tạp trung bình là $O(\log n)$ nếu cây được cân bằng. Tuy nhiên, nếu dữ liệu được chèn theo thứ tự không ngẫu nhiên, cây dễ bị lệch hẳn sang một phía và biến thành danh sách liên kết, dẫn đến độ phức tạp tăng lên $O(n)$.

Heap là một cấu trúc dữ liệu dạng cây gần đầy, tuân theo một trong hai quy tắc: với Min-Heap thì mỗi nút có giá trị nhỏ hơn hoặc bằng các nút con; còn với Max-Heap thì mỗi nút có giá trị lớn hơn hoặc bằng các nút con. Heap thường được dùng trong các bài toán như sắp xếp (Heap Sort) hoặc cài đặt hàng đợi ưu tiên.

Cây nhị phân cân bằng (Balanced Binary Tree) là loại cây trong đó độ cao của hai cây con của mỗi nút không chênh lệch quá nhiều. Một số ví dụ điển hình là cây AVL, trong đó độ cao của hai nhánh con chênh nhau tối đa là một, và cây Đỏ-Đen (Red-Black Tree), sử dụng quy tắc màu sắc để đảm bảo cây không bị lệch quá mức. Cả hai cấu trúc này đều đảm bảo chiều cao cây luôn giữ ở mức $O(\log n)$, nhờ đó duy trì hiệu suất tốt cho các thao tác.

Cuối cùng, để duy trì sự cân bằng của cây nhị phân sau khi thực hiện chèn hoặc xóa, ta thường sử dụng kỹ thuật xoay cây. Có hai loại xoay chính là xoay trái và xoay phải. Trong thao tác xoay, các con trỏ giữa nút cha và nút con được điều chỉnh lại nhằm thay đổi cấu trúc cây mà vẫn đảm bảo tính chất của BST. Xoay cây là thành phần quan trọng trong các cây tự cân bằng như AVL, Red-Black Tree và cả Treap – nơi mà việc xoay cây sẽ được thực hiện dựa trên thứ tự ưu tiên (priority) của nút.

Những kiến thức trên là nền tảng quan trọng để hiểu rõ Treap – một cấu trúc đặc biệt kết hợp tính chất của cả BST và Heap trong cùng một cây nhị phân.

2.2 Định nghĩa

Treap là một cấu trúc dữ liệu cây nhị phân tự cân bằng được phát minh bởi Cecilia R. Aragon và Raimund Seidel vào năm 1989, được giới thiệu trong bài báo nổi tiếng "Randomized Search Trees". Treap kết hợp một cách sáng tạo hai ý tưởng quan trọng trong cấu trúc dữ liệu: cây nhị phân tìm kiếm (Binary Search Tree – BST) và Heap (có thể là Min-Heap hoặc Max-Heap). Nhờ sự kết hợp này, Treap thừa hưởng được ưu điểm của cả hai loại cây, đồng thời duy trì được hiệu suất trung bình tốt một cách tự nhiên thông qua tính ngẫu nhiên.

Về mặt cấu trúc, mỗi nút trong Treap bao gồm hai thông tin chính: khóa (key) và độ ưu tiên (priority). Tính chất BST được áp dụng dựa trên trường khóa, tức là với mỗi nút, tất cả các nút trong cây con bên trái có khóa nhỏ hơn, và các nút trong cây con bên phải có khóa lớn hơn. Đồng thời, tính chất Heap được áp dụng trên trường độ ưu tiên, đảm bảo rằng với Min-Heap (hoặc Max-Heap), độ ưu tiên của nút cha luôn nhỏ hơn (hoặc lớn hơn) độ ưu tiên của các nút con. Thông thường, giá trị độ ưu tiên được sinh ngẫu nhiên, nhờ đó Treap trở thành một cây nhị phân tự cân bằng theo xác suất, thay vì dựa trên các phép tính toán phức tạp như trong AVL hay Red-Black Tree.

Tính ngẫu nhiên trong lựa chọn độ ưu tiên giúp Treap đạt được độ cao kỳ vọng là $O(\log n)$, đảm bảo rằng các thao tác tìm kiếm, chèn và xóa đều có độ phức tạp trung bình tốt. Đây là lý do vì sao Treap thường được sử dụng như một giải pháp thay thế đơn giản nhưng hiệu quả cho các cây tự cân bằng truyền thống trong nhiều ứng dụng thuật toán và hệ thống thời gian thực.

Vậy, Treap là một cây nhị phân, trong đó mỗi nút có hai giá trị:

- **Key** – Dùng để duy trì tính chất BST.
- **Priority** – Dùng để duy trì tính chất Heap (Max-Heap hoặc Min-Heap).

2.3 Tính chất

Treap sở hữu một số tính chất đặc trưng nhờ vào sự kết hợp giữa hai cấu trúc dữ liệu nền tảng là Binary Search Tree (BST) và Heap. Mỗi tính chất đóng vai trò thiết yếu trong việc đảm bảo hiệu suất cũng như sự tự cân bằng của cây theo cách đơn giản và hiệu quả.

Thứ nhất, Treap thỏa mãn tính chất của một cây nhị phân tìm kiếm. Điều này có nghĩa là với mỗi nút trong cây, tất cả các nút thuộc cây con bên trái sẽ có khóa nhỏ hơn khóa của nút đó, trong khi các nút thuộc cây con bên phải sẽ có khóa lớn hơn. Nhờ đó, các thao tác tìm kiếm trên Treap vẫn tuân theo nguyên tắc quen thuộc như trong BST và có thể thực hiện một cách trực quan.

Thứ hai, Treap đồng thời thỏa mãn tính chất của một Heap dựa trên trường độ ưu tiên (priority). Cụ thể, trong Treap kiểu Min-Heap, độ ưu tiên của mỗi nút luôn nhỏ hơn hoặc bằng độ ưu tiên của các nút con; ngược lại, trong Treap kiểu Max-Heap, độ ưu tiên của mỗi nút lớn hơn hoặc bằng độ ưu tiên của các nút con. Điều đáng chú ý là các giá trị độ ưu tiên thường được sinh ra một cách ngẫu nhiên khi tạo nút, điều này tạo ra hiệu ứng cân bằng xác suất cho cây.

Sự phối hợp giữa hai tính chất nói trên mang lại một điểm mạnh vượt trội cho Treap: đó là khả năng cân bằng cấu trúc cây mà không cần tới các quy tắc phức tạp như trong AVL hay Red-Black Tree. Trong khi các cây tự cân bằng truyền thống yêu cầu theo dõi và điều chỉnh độ cao hoặc màu sắc, thì Treap đơn giản chỉ cần thực hiện các phép xoay cây để duy trì cả hai tính chất khi chèn hoặc xóa nút.

Một tính chất quan trọng khác là Treap có chiều cao trung bình kỳ vọng là $O(\log n)$, trong đó n là số lượng nút của cây. Điều này đảm bảo rằng các thao tác cơ bản như tìm kiếm, chèn, xóa đều có độ phức tạp trung bình hiệu quả. Dù vẫn có khả năng xảy ra trường hợp xấu với chiều cao $O(n)$, nhưng xác suất để cây trở nên mất cân bằng là rất thấp nếu độ ưu tiên được sinh ra ngẫu nhiên và không trùng lặp.

Nhờ các tính chất này, Treap không chỉ là một lựa chọn thay thế đơn giản cho các cây cân bằng truyền thống mà còn là một công cụ linh hoạt để xây dựng các thuật toán nâng cao như Split, Merge, Range Query, hay thậm chí trong các hệ thống thời gian thực và lập trình thi đấu.

Một Treap thỏa mãn:

- Theo **key**: left.key $<$ root.key $<$ right.key (BST)
- Theo **priority**: root.priority $>$ con.priority (Heap)

3 CẤU TRÚC DỮ LIỆU

```
struct Node {
    int key;
    int priority;
    Node* left;
    Node* right;

    Node(int k) {
        key = k;
        priority = rand();
        left = right = nullptr;
    }
};
```

4 CÁC THAO TÁC CƠ BẢN

4.1 Xoay cây

Xoay trái:

```
Node* rotateLeft(Node* root) {
    Node* R = root->right;
    root->right = R->left;
    R->left = root;
    return R;
}
```

Xoay phải:

```
Node* rotateRight(Node* root) {
    Node* L = root->left;
    root->left = L->right;
    L->right = root;
    return L;
}
```

4.2 Thêm phần tử

```
Node* insert(Node* root, int key) {
    if (!root) return new Node(key);
    if (key < root->key) {
        root->left = insert(root->left, key);
        if (root->left->priority > root->priority)
            root = rotateRight(root);
    } else {
```

```
        root->right = insert(root->right, key);
        if (root->right->priority > root->priority)
            root = rotateLeft(root);
    }
    return root;
}
```

4.3 Xóa phần tử

```
Node* deleteNode(Node* root, int key) {
    if (!root) return nullptr;
    if (key < root->key)
        root->left = deleteNode(root->left, key);
    else if (key > root->key)
        root->right = deleteNode(root->right, key);
    else {
        if (!root->left)
            root = root->right;
        else if (!root->right)
            root = root->left;
        else if (root->left->priority > root->right->priority) {
            root = rotateRight(root);
            root->right = deleteNode(root->right, key);
        } else {
            root = rotateLeft(root);
            root->left = deleteNode(root->left, key);
        }
    }
    return root;
}
```

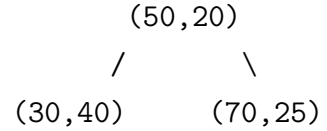
4.4 Tìm kiếm

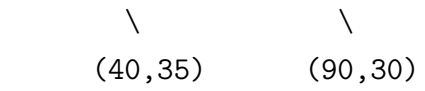
Duyệt theo quy tắc của cây BST.

5 MINH HỌA

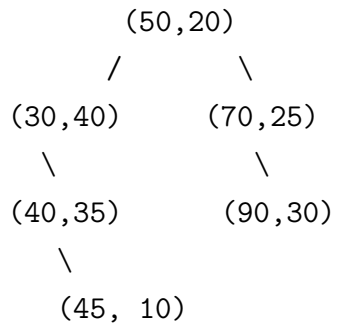
5.1 Thao tác chèn

Giả sử chèn (45, 10) vào cây sau :

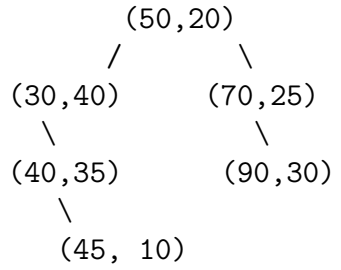


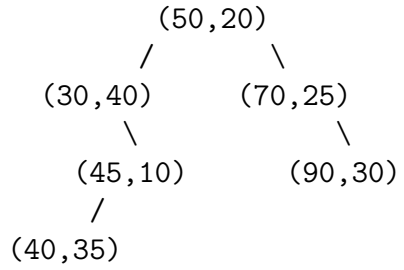


Bước 1: Chèn cặp (K, P) như một lá mới (leaf)

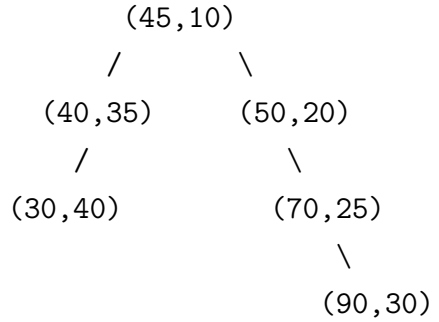


Bước 2: Sau đó, xoay nút đó lên (rotate up) bằng cách sử dụng các phép xoay của AVL nếu cần thiết





Tiếp tục xoay cho đến khi cây đảm bảo các tính chất của Treap và được kết quả như sau:



Độ phức tạp:

- Mỗi phép xoay là O(1).
- Trường hợp trung bình là O(log n)
- Trường hợp tệ hơn vẫn là O(n)

5.2 Thao tác xóa

- Bước 1: Tìm nút X chứa K bằng cách sử dụng thuật toán tìm kiếm thông thường của BST.
- Bước 2: Nếu nút X là một lá (leaf), chỉ cần xóa nút đó (gỡ liên kết với nút cha).
- Bước 3: Nếu nút X không phải là lá, sử dụng các phép xoay của AVL (AVL rotations) để xoay nút đó xuống dưới (rotate down) cho đến khi nó trở thành một lá. Sau đó, xóa nó đi.

Độ phức tạp:

- Mỗi phép xoay là O(1).
- Đối với việc xóa, giống như tìm kiếm, trường hợp trung bình là O(log n)
- Nhưng trường hợp tệ hơn vẫn là O(n) vì mặc dù trung bình cây khá cân bằng nhưng vẫn có khả năng cây bị mất cân bằng nghiêm trọng.

5.3 Thao tác Merge và Split

Thao tác Merge

Kết hợp hai Treap T1 và T2 thành một Treap mới, đảm bảo:

- Mọi key trong T1 $<$ mọi key trong T2
- Treap kết quả vẫn thỏa mãn tính chất BST và Heap

Thao tác Split

- Cho một cây và một giá trị khóa K không có trong cây.
- Tạo ra hai cây: Một cây chứa các khóa nhỏ hơn K, và một cây chứa các khóa lớn hơn K.
- Insert (K, $INFINITIY$) vào Treap.
- Tương tự như phép insert đã thực hiện. Vì INFINITY (vô cực) có độ ưu tiên cao hơn bất kỳ nút nào trong heap, nên nó sẽ trở thành gốc (root) của Treap sau khi chèn.

6 ƯU, NHƯỢC ĐIỂM CỦA TREAP

6.1 Ưu điểm

Tiêu chí	**Ưu điểm**
Cấu trúc kết hợp	Kết hợp giữa BST và Heap nên có khả năng tự cân bằng xác suất cao nhờ `priority` ngẫu nhiên.
Thao tác cơ bản	Hỗ trợ nhanh các thao tác `insert`, `delete`, `split`, `merge` với độ phức tạp trung bình $O(\log n)$.
Cài đặt	Cài đặt đơn giản hơn AVL và Red-Black Tree, không cần xử lý nhiều trường hợp xoay phức tạp.
Mở rộng	Dễ biến đổi thành Implicit Treap, Persistent Treap, phù hợp nhiều ứng dụng thực tế.

Table 1: Ưu điểm của Treap

6.2 Nhược điểm

Tiêu chí	**Nhược điểm**
Độ cân bằng	Không đảm bảo luôn cân bằng tuyệt đối như AVL Tree do phụ thuộc vào `priority` ngẫu nhiên.
Bộ nhớ	Sử dụng con trỏ động, không tối ưu về cache locality, có thể gây phân mảnh bộ nhớ.
Độ tin cậy	Do dùng random nên kết quả thao tác có thể dao động. Không phù hợp hệ thống cần kết quả ổn định tuyệt đối.
Truy vấn đoạn	Không mạnh như Segment Tree trong các bài toán truy vấn đoạn (tổng, max, min, v.v.).

Table 2: Nhược điểm của Treap

7 BIẾN THỂ CỦA TREAP

7.1 Implicit Treap

7.1.1 Khái niệm

Implicit Treap là một biến thể của Treap trong đó khóa (key) không được lưu trữ trực tiếp trong mỗi nút mà được ẩn đi ("implicit") thông qua vị trí của nút trong cây. Nói cách khác, thứ tự của phần tử trong một dãy số (hoặc mảng) chính là khóa để sắp xếp cây. Điều này cho phép Treap mô phỏng các thao tác trên mảng động với độ phức tạp trung bình là $\mathcal{O}(\log n)$.

7.1.2 Kiểu dữ liệu biểu diễn

Implicit Treap thường được sử dụng để biểu diễn một dãy số (hoặc chuỗi ký tự). Mỗi nút của cây đại diện cho một phần tử trong dãy.

7.1.3 Thành phần bên trong của mỗi nút

Một nút trong Implicit Treap thường bao gồm các trường:

- `value`: Giá trị tại vị trí tương ứng trong dãy.
- `priority`: Một số ngẫu nhiên để duy trì tính chất heap ngẫu nhiên.
- `left, right`: Con trái và con phải.
- `size`: Tổng số nút trong cây con gốc tại nút hiện tại.
- `rev` (tuỳ chọn): Biến cờ dùng cho lazy propagation để hỗ trợ đảo đoạn.

7.1.4 Tác dụng của biến thể

Implicit Treap cho phép thực hiện các thao tác phổ biến trên mảng như:

- Chèn một phần tử vào vị trí bất kỳ trong dãy.
- Xoá phần tử tại một vị trí bất kỳ.
- Truy cập phần tử theo chỉ số.
- Đảo ngược một đoạn trong dãy (với lazy propagation).

Tất cả các thao tác trên đều đạt độ phức tạp trung bình $\mathcal{O}(\log n)$ nhờ vào phép chia cây (split) và trộn cây (merge).

7.1.5 Thao tác cơ bản

- **split**: Chia cây thành hai cây con theo chỉ số k.
- **merge**: Trộn hai cây lại thành một cây hợp lệ.
- **insert**: Chèn giá trị x vào vị trí k bằng cách split cây tại k, chèn nút mới và merge lại.
- **erase**: Xoá phần tử tại vị trí k.
- **reverse**: Đảo đoạn từ vị trí l đến r bằng cách split thành 3 cây rồi đảo phần giữa và merge lại.

7.1.6 Ứng dụng thực tế

- Làm mảng động hỗ trợ chèn/xoá/truy cập hiệu quả.
- Xử lý chuỗi (biến thể như Rope).
- Giải quyết các bài toán thao tác đoạn (segment rearrangement) trong lập trình thi đấu.
- Demo trực quan các thuật toán đảo đoạn, undo thao tác trên chuỗi (khi kết hợp với Persistent Treap).

7.2 Pesistent treap

7.2.1 Khái niệm

Persistent Treap là một biến thể của Treap hỗ trợ lưu lại toàn bộ lịch sử của các phiên bản cây sau mỗi lần chỉnh sửa (chèn, xoá). Khi thực hiện thao tác, cây không bị thay đổi trực tiếp mà một bản sao của cây mới (chỉ thay đổi các nút cần thiết) sẽ được tạo ra. Điều này giúp ta có thể "quay lại" bất kỳ phiên bản nào trước đó – còn gọi là "undo/redo" hoặc "truy xuất phiên bản".

7.2.2 Kiểu dữ liệu biểu diễn

Tương tự như Treap cơ bản, Persistent Treap cũng có thể biểu diễn:

- Cây nhị phân tìm kiếm (theo khoá)
- Một dãy số (nếu kết hợp thêm các kỹ thuật implicit)

7.2.3 Thành phần bên trong của mỗi nút

Một nút trong Persistent Treap thường có cấu trúc tương tự Treap cơ bản:

- `key`: Khoá giá trị.
- `priority`: Giá trị ngẫu nhiên để đảm bảo tính chất heap.
- `left, right`: Con trái và con phải (điểm quan trọng là chúng có thể trỏ tới nút ở các phiên bản trước).
- `size, rev` (tuỳ chọn): Nếu cần dùng thêm cho implicit treap hoặc lazy propagation.

Tuy nhiên, sự khác biệt quan trọng là:

- Các thao tác không chỉnh sửa trực tiếp cây gốc.

- Khi cần thay đổi, tạo bản sao của nút thay đổi (copy-on-write), giúp giữ nguyên phiên bản cũ.

7.2.4 Tác dụng của biến thể

Persistent Treap giúp:

- Quản lý phiên bản của cây nhị phân.
- Hỗ trợ quay lui (undo), tiến tới (redo).
- Tối ưu hoá bộ nhớ: các phiên bản chỉ khác nhau ở một số ít nút nên phần lớn dữ liệu được tái sử dụng.

7.2.5 Thao tác cơ bản

- **split**: Giống Treap cơ bản nhưng trả về các cây ở phiên bản mới.
- **merge**: Trả về cây mới mà không làm thay đổi hai cây con ban đầu.
- **insert, erase**: Trả về cây mới sau khi thêm/xoá phần tử.
- **versioning**: Mỗi lần thao tác sinh ra một root mới đại diện cho phiên bản tiếp theo.

7.2.6 Ứng dụng thực tế

- Hệ thống cần lưu lịch sử các thao tác (như lịch sử chỉnh sửa trong tài liệu).
- Các bài toán dạng "undo/redo" nhiều bước, ví dụ trong phần mềm soạn thảo, chỉnh sửa ảnh, hay mô phỏng hành động.
- Cấu trúc dữ liệu hỗ trợ truy vấn thông tin tại bất kỳ thời điểm nào trong quá khứ.
- Nền tảng cho các thuật toán "time-travel" trong lập trình cạnh tranh.

8 KẾT LUẬN

Treap là một cấu trúc dữ liệu hiệu quả, vừa giữ được tính chất của BST, vừa duy trì cân bằng trung bình nhờ tính ngẫu nhiên. Cấu trúc này phù hợp cho các bài toán yêu cầu hiệu suất ổn định nhưng không quá phức tạp trong cài đặt.

9 TREAP VISUALIZATION

Demo chương trình và tham số liên quan: Chương trình trực quan hóa Treap được triển khai dưới dạng một ứng dụng web, bao gồm hai thành phần chính: backend sử dụng Flask và frontend xây dựng bằng ReactJS. Người dùng có thể tương tác trực tiếp thông qua giao diện đồ họa để thực hiện các thao tác như chèn, xóa nút trên cây Treap, đồng thời quan sát quá trình tự cân bằng thông qua các hiệu ứng trực quan sinh động.

Về mặt tổ chức, chương trình được chia thành hai thư mục chính là `backend/` và `frontend/`. Trong đó, `backend/` chứa các tập tin như `app.py` (chạy server Flask và định nghĩa API), `treap.py` (triển khai logic chèn, xóa, xoay cây Treap), cùng với tập tin `requirements.txt` khai báo các thư viện cần thiết. Thư mục `frontend/` chứa mã nguồn React, bao gồm các thành phần hiển thị cây, xử lý điều khiển người dùng, cũng như kết nối tới backend thông qua các hàm API. Toàn bộ cấu trúc được tổ chức rõ ràng nhằm đảm bảo dễ dàng trong việc phát triển, bảo trì và mở rộng ứng dụng.

Trong quá trình vận hành, người dùng nhập vào các giá trị như `key` và `priority` để chèn một nút vào cây. Nếu người dùng không cung cấp giá trị độ ưu tiên, hệ thống sẽ sinh ngẫu nhiên nhằm đảm bảo tính ngẫu nhiên của cấu trúc Treap. Ngoài ra, người dùng có thể chỉ định thao tác cụ thể như chèn (`insert`), xóa (`delete`) hoặc làm trống toàn bộ cây (`clear`) và chọn chế độ (`Max Heap`) hoặc (`Min Heap`).

Về mặt kỹ thuật, mỗi thao tác từ giao diện người dùng sẽ sinh ra một truy vấn HTTP gửi đến backend Flask. Chẳng hạn, thao tác chèn sẽ phát sinh truy vấn `POST /insert` kèm theo dữ liệu JSON chứa `key` và `priority`. Backend sẽ tiếp nhận yêu cầu, xử lý việc cập nhật cây, thực hiện các bước cân bằng như xoay trái hoặc xoay phải nếu cần thiết, sau đó trả về cây Treap hiện tại dưới định dạng JSON. Tương tự, thao tác xóa gửi yêu cầu `POST /delete`, và thao tác làm trống cây gửi đến `POST /clear` nếu được hỗ trợ.

Dữ liệu phản hồi từ backend có định dạng JSON theo cấu trúc đệ quy, biểu diễn đầy đủ cây nhị phân. Mỗi nút bao gồm khóa (`key`), độ ưu tiên (`priority`) cùng với các con trỏ `left` và `right`. Frontend sẽ sử dụng dữ liệu này để dựng lại cây bằng SVG hoặc thư viện vẽ như D3.js. Mỗi nút được hiển thị kèm thông tin và hiệu ứng di chuyển, giúp người dùng dễ dàng quan sát cấu trúc và thay đổi của cây theo thời gian thực.

Giao diện người dùng hỗ trợ nhập dữ liệu trực tiếp, lựa chọn thao tác tương ứng, và quan sát kết quả ngay trên màn hình. Ngoài ra, hệ thống còn hiển thị thông báo lỗi nếu người dùng nhập sai định dạng hoặc thao tác trên nút không tồn tại.

Kết quả demo cho thấy ứng dụng đáp ứng đầy đủ các yêu cầu đặt ra, bao gồm khả năng duy trì tính chất BST theo khóa và Heap theo độ ưu tiên, thực hiện các thao tác cơ bản, và đặc biệt là trực quan hóa rõ ràng quá trình tự cân bằng của Treap. Việc

cung cấp khả năng điều chỉnh tham số và phản hồi theo thời gian thực giúp nâng cao khả năng học tập, trải nghiệm và phân tích thuật toán một cách trực quan và hiệu quả.

Figure 1: Giao diện web

10 TÀI LIỆU THAM KHẢO

- Cormen, Leiserson, Rivest, Stein - Introduction to Algorithms
- https://www.geeksforgeeks.org/treap-set-1-introduction-and-insertion/
- https://cp-algorithms.com/data_structures/treap.html
- CMSC Treaps, Justin Wyss-Gallifent
- Lecture 3: Treaps and Skip Lists [Sp'17]
- "Randomized Search Trees" paper (Aragon and Seidel)
- https://www.coursehero.com/file/236213930/treapspdf/