ĐẠI HỌC QUỐC GIA TPHCM

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

KHOA CÔNG NGHỆ THÔNG TIN

BỘ MÔN CÔNG NGHỆ TRI THỨC

Báo cáo Bài Tập Gì Đấy

Đề tài: Báo Cáo Đồ Án Môn Học

Môn học: Cấu Trúc Dữ Liệu và Giải Thuật

Sinh viên thực hiện:

Khang, Huynh Bao (24120507)

Giáo viên hướng dẫn:

GS. TS. Nguyễn Thanh Tình

Ngày 18 tháng 5 năm 2025

Mục lục

Danh sách bảng

Danh sách hình vẽ

1 Giới thiệu

Cây nhị phân tìm kiếm (Binary Search Tree - BST) là một trong những cấu trúc dữ liệu nền tảng trong khoa học máy tính, hỗ trợ thao tác tìm kiếm, chèn và xóa hiệu quả. Tuy nhiên, BST thông thường có thể mất cân bằng dẫn đến ảnh hưởng đến hiệu suất. Vì vậy, nhiều biến thể của BST được phát triển để tăng hiệu quả truy xuất dữ liệu.

2 Optimal Binary Search Tree

2.1 Lý thuyết

Cây nhị phân tìm kiếm tối ưu được xây dựng từ các tập khóa có xác suất truy cập cụ thể, giúp tối đa hóa chi phí tìm kiếm trung bình.

2.2 Quy hoạch động

- c[i][j] : Chi phí tối thiểu khi tìm kiếm từ khóa i + 1 đến j
- w[i][j] : Tổng xác suất từ i + 1 đến j
- root[i][j] : Gốc tốt nhất trong đoạn [i + 1...j]

2.3 Mã code

```
#include <iostream>
#include <vector>
#include <limits>

using namespace std;
const int mxN = 100;
double c[mxN][mxN] , w[mxN][mxN];
int root[mxN][mxN];

void optimalBST (const vector<double> &p , const vector<double>& q , int n)
{
    for (int i = 0; i <= n; i++) {
```

```
            c[i][i] = q[i];
            w[i][i] = q[i];
        }
        for (int l = 1; l <= n; l++)
            for (int i = 0; i <= n - 1; i++) {
                int j = i + 1;
                c[i][j] = numeric_limits<double> :: infinity();
                w[i][j] = w[i][j - 1] + p[j] + q[j];
                for (int r = i + 1; r <= j; r++) {
                    double cost = c[i][r - 1] + e[r][j] + w[i][j];
                    if(cost < c[i][j]) {
                        e[i][j] = cost;
                        root[i][j] = r;
                    }
                }
            }
    }
```

3 Threaded Binary Trees

3.1 Lý thuyết

Trong BST thường có nhiều con trỏ NULL không mang thông tin. Threaded Binary Tree thay các con trỏ NULL bằng liên kết tới nút tiền nhiệm hoặc kế nhiệm hoặc kế tiếp theo thứ tự in-order, giúp duyệt cây không cần stack hoặc dùng đệ quy.

Threaded Binary Tree có thể hữu ích khi không gian là vấn đề, vì chúng có thể loại bỏ sử dụng ngăn xếp trong quá trình duyệt. Tuy nhiên, quá trình cài đặt lại khá phức tạp so với cây nhị phân tiêu chuẩn

3.2 Cấu trúc Node

```
    struct Node {
        int data;
        Node *left , *right;
        bool rThread;
```

```
};
```

3.3 Duyệt in-order không đệ quy

```
Node* leftMost(Node* u) {
    while(u && u -> left) u = u -> left;
    return u;
}
void inOrder (Node* root) {
    Node* u = leftMost(root);
    while(u) {
        cout << u -> data << " ";
        if(u -> rThread) u = u -> right;
        else u = leftMost(u -> right);
    }
}
```

3.4 Ưu điểm

- Không dùng ngăn xếp hay đệ quy khi duyệt cây, vì khi sử dụng các liên kết, chúng ta có thể đến các nút đã truy cập trước đó.
- Các con trỏ trỏ đến các nút con và cha trực tiếp. Điều này giúp chúng ta có thể đến nút liền kề của bất kì nút nào một cách nhanh chóng.
- Không có các con trỏ NULL nào hiện diện. Do đó, tránh lãng phí bộ nhớ khi chiếm các liên kết NULL.
- Cho phép duyệt tiến và lùi các nút theo thứ tự.

3.5 Nhược điểm

- Mỗi nút trong Threaded Binary Tree cần thêm thông tin (bộ nhớ bổ sung). Vì vậy, cần tốn thêm bộ nhớ để triển khai.
- Việc chèn và xóa thường tốn nhiều thời gian và cài đặt khó hơn thông thường.

3.6 Độ phức tạp

- Thời gian : O(n) khi duyệt toàn bộ cây
- Không gian: O(1) do không dùng đệ quy hay stack

4 Augmented Binary Search Tree

4.1 Lý thuyết

Augmented Binary Search Tree là cây nhị phân tìm kiếm mở rông thêm thông tin phụ trợ (như số nút con) giúp hỗ trợ các truy vấn đặc biệt (tìm phần tử thứ k, tổng đoạn,...)

4.2 Ví dụ: Order - Statistic Tree

```
Struct Node {
    int data , sz;
    Node *left , *right;
};

// Cap nhat lai size cua 1 nut
void upsize(Node* u) {
    if(!u) return;
    u -> sz = 1;
    if(u -> left) u -> sz += u -> left -> sz;
    if(u -> right) u -> sz += u -> right -> sz;
}

// Chen 1 phan tu vao trong cay
Node* insertNode(Node *root , int value) {
    Node* newNode = new Node {value , 1 , nullptr , nullptr};
    if(!root) return newNode;
    if(value < root -> data)
        root -> left = insertNode(root -> left , value);
    else root -> right = insertNode(root -> right , value);
    upsize(root);
    return root;
```

```
    }

    // Tim phan tu thu k nho nhat
    int findKth(Node* root , int k) {
        if(!root) return -1;
        int leftSize = (root -> left ? root -> left -> sz : 0);
        if(k == leftSize + 1) return root -> data;
        else if (k <= leftSize) return findKth(root -> left , k);
        else return findKth(root -> right , k - leftSize - 1);
    }
```

4.3 Độ phức tạp

Thao tác	Độ phức tạp trung bình	Độ phức tạp tệ nhất (cây lệch)
Insert/Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Tìm k-th nhỏ nhất	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Cập nhật size	$\mathcal{O}(1)$ mỗi nút	

4.4 Ưu điểm

- Xử lý các truy vấn phức tạp 1 cách hiệu quả
- Có thể mở rộng lên các cấu trúc cây khác như: Segment Tree , Oder-Statistic Tree,..

4.5 Nhược điểm

- Cài đặt phức tạp hơn so với cây nhị phân tìm kiếm thông thường
- Khó khăn trong việc cập nhật thông tin trong lúc chèn hoặc xóa
- Nếu cây không cân bằng có thể dẫn đến độ phức tạp O(n)

5 Interative Traversal Methods

5.1 Duyệt sử dũng Stack

```
// In-Order Traversal (LNR)
void inOrderInterative(Node *root) {
    stack<Node*> st;
    Node* u = root;
    while(u || st.size()) {
        while(u) {
            st.push(u);
            u = u -> left;
        }
        u = st.top() , st.pop();
        cout << u -> data << " ";
        u = u -> right;
    }
}

//Pre-Order Traversal(NLR)
void preOrderInterative(Node *root) {
    if(!root) return;
    stack<Node*> st;
    st.push(root);
    while(st.size()) {
        Node* u = st.top() , st.pop();
        cout << u -> data << " ";
        if(u -> right) st.push(u -> right);
        if(u -> left) st.push(u -> left);
    }
}

//Post-Order Traversal(LRN)
void postOrderInterative(Node* root) {
    if(!root) return;
    stack<Node*> st1 , st2;
    st1.push(root);
    while(st1.size()) {
        Node* u = st1.top() , st1.pop();
        st2.push(u);
        if(u -> left) st1.push(u -> left);
```

```
38              if(u -> right) st1.push(u -> right);
39          }
40          while(st2.size()) {
41              cout << st2.top() -> data << " ";
42              st2.pop();
43          }
44      }
45
46      //Level-Order Traversal
47      void levelOrderInterative(Node* root) {
48          queue<Node*> pq;
49          pq.push(root);
50          while(pq.size()) {
51              NOde* u = pq.front() , pq.pop();
52              cout << u -> data << " ";
53              if(u -> left) pq.push(u -> left);
54              if(u -> right) pq.push(u -> right);
55          }
56      }
```

5.2 Độ phức tạp

Thuật toán	Thời gian	Không gian (tệ nhất)
In-order	$O(n)$	$O(n)$
Pre-order	$O(n)$	$O(n)$
Post-order	$O(n)$	$O(n)$
Level-order	$O(n)$	$O(n)$

5.3 Ưu điểm

- Không phụ thuộc vào call stack: tránh lỗi stack overflow khi cây quá lớn.
- Tùy biến dễ hơn so với dùng đệ quy

5.4 Nhược điểm

- Cần quản lý bộ nhớ tạm (stack/queue) thủ công

- Phức tạp khi cài đặt cho phần Post-Order

6 K - Nearest Neighbor Search

6.1 Lý thuyết

Tìm K điểm gần nhất với điểm truy vấn. Có thể cài đặt bằng cây KD hoặc brute-force + heap.

6.2 Mã code

```
struct Point {
    int x , y;
};
double dist (Point a , Point b) {
    return sqrt((a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y));
}

vector<Point> kNearest(vector<Point> &pts , Point query , int k) {
    priority_queue<pair<double , Point>> pq;
    for (auto &p : pts) {
        double d = dist(p , query);
        pq.push(make_pair(d , p));
        if(pq.size() > k) pq.pop();
    }
    vector<Point> ans;
    while(pq.size()) {
        ans.push_back(pq.top().second);
        pq.pop();
    }
    return ans;
}
```

Thuật toán	**Thời gian**	**Không gian**
Brute force	$O(n \log K)$ (với heap), $O(n)$ nếu sort	$O(K)$
Dùng K-D Tree (avg case)	$O(\log n)$ mỗi truy vấn	$O(n)$
Dùng K-D Tree (worst case)	$O(n)$	$O(n)$

6.3 Độ phức tạp

6.4 Ưu điểm

- Dễ dàng cài đặt (brute force + heap)
- Kết quả chính xác nếu đo đúng

6.5 Nhược điểm

- Brute force không dùng được nếu dữ liệu lớn
- Tính toán khoảng cách tốn kém với dữ liệu lớn

7 Kết Luận

Các biến thể của BST như Threaded , Augmented BST , Interative Traversal , K Nearest Neighbor, ... giúp nâng cao hiệu năng của BST để đáp ứng các yêu cầu đặc thù như tốc độ, bộ nhớ và tìm kiếm thông minh.

Tùy theo ứng dụng, chúng có thể được kết hợp để đạt hiệu quả tối ưu nhất.