TRƯỜNG ĐẠI HỌC GIAO THÔNG VẬN TẢI

KHOA CÔNG NGHỆ THÔNG TIN

---------------o0o---------------

**Bài tập lớn môn học**

Giảng viên hướng dẫn: TS. Hoàng Văn Thông

Sinh viên thực hiện: Trương Quang Vinh

Mã sinh viên: 221231053

Lớp: CNTT 3 – K63

**CẤU TRÚC DỮ LIỆU VÀ GIẢI THUẬT**

**Hà Nội tháng 11 năm 2023**

# MỤC LỤC

[MỤC LỤC 2](#_Toc149906892)

[BÀI 1 (BÀI SỐ 6 TRONG DANH SÁCH BÀI TẬP) 3](#_Toc149906893)

[1.1. Đề bài 3](#_Toc149906894)

[1.2. Phân tích bài toán 3](#_Toc149906895)

[1.2. Cài đặt các lớp và hàm main bằng C++ 8](#_Toc149906896)

[1.3. Phân tích thời gian chạy của từng phương thức có trong các lớp 8](#_Toc149906897)

[BÀI 2 (BÀI SỐ 36 TRONG DANH SÁCH BÀI TẬP) 20](#_Toc149906898)

[2.1. Đề bài 20](#_Toc149906899)

[2.2. Phân tích bài toán 21](#_Toc149906900)

[2.3. Cài đặt các lớp và hàm main bằng C++ 25](#_Toc149906901)

[2.4. Phân tích thời gian chạy của từng phương thức có trong các lớp 25](#_Toc149906902)

[DANH SÁCH TÀI LIỆU THAM KHẢO 37](#_Toc149906903)

# 

# BÀI 1 (BÀI SỐ 6 TRONG DANH SÁCH BÀI TẬP)

## Đề bài

1. Xây dựng lớp cây tìm kiếm nhị phân.

2. Xây dựng một lớp quản lý một từ điển Anh-Việt, từ điển được lưu vào cây nhị phân mà tại mỗi nút của nó lưu một từ tiếng Anh và nghĩa tiếng Việt tương ứng có các phương thức:

a. Đọc từ điển từ file để xây dựng cây.

b. Ghi từ điển trong cây vào file (duyệt theo thứ tự giữa, duyệt đến nút nào thì ghi vào file giávtrị của nút đó).

c. Thêm một từ mới vào từ điển đang lưu trong cây.

d. Xóa bỏ một từ của từ điển đang lưu trong cây.

e. Cập nhật lại nghĩa một từ đang lưu trong cây.

f. Tìm kiếm một từ trong cây.

3. Viết chương trình kiểm tra sự hoạt động của lớp từ điển đã xây dựng.

## 1.2. Phân tích bài toán

* Yêu cầu của bài toán
* Xây dựng lớp cây tìm kiếm nhị phân.
* Xây dựng một lớp quản lý một từ điển Anh-Việt, từ điển được lưu vào cây nhị phân mà tại mỗi nút của nó lưu một từ tiếng Anh và nghĩa tiếng Việt tương ứng.
* Viết chương trình kiểm tra sự hoạt động của lớp từ điển đã xây dựng.
* Xác định các lớp, các thuộc tính, phương thức của lớp
* Lớp BSTNode
* Các thuộc tính
* Keys key;
* T element;
* BSTNode<Keys, T> \*parent, \*left, \*right;
* Các phương thức
* BSTNode();
* BSTNode(Keys k, T e, BSTNode<Keys, T> \*parent);
* BSTNode<Keys, T> \*getParent();
* BSTNode<Keys, T> \*getLeft();
* BSTNode<Keys, T> \*getRight();
* void setLeft(BSTNode<Keys, T> \*p);
* void setRight(BSTNode<Keys, T> \*p);
* void setParent(BSTNode<Keys, T> \*p);
* int hasLeft();
* int hasRight();
* T getElement();
* Keys getKey();
* void setElement(T e);
* void setKey(Keys k);
* Lớp BSTree
* Các thuộc tính
* BSTNode<Keys, T> \*root;
* int n;
* Các phương thức
* BSTree();
* int size();
* int isEmpty();
* int isInternal(BSTNode<Keys, T> \*p);
* int isExternal(BSTNode<Keys, T> \*p);
* int isRoot(BSTNode<Keys, T> \*p);
* BSTNode<Keys, T> \*search(Keys k, BSTNode<Keys, T> \*p);
* BSTNode<Keys, T> \*getRoot();
* bool search(Keys k);
* BSTNode<Keys, T> \*insert(Keys k, T e, BSTNode<Keys, T> \*temp, BSTNode<Keys, T> \*parent);
* void insert(Keys k, T e);
* BSTNode<Keys, T> \*getMinRight(BSTNode<Keys, T> \*p);
* void remove(BSTNode<Keys, T> \*tmp);
* void remove(Keys k);
* void preOrder(BSTNode<Keys, T> \*p, ofstream &f);
* void preOrder();
* void inOrder(BSTNode<Keys, T> \*p, ofstream &f);
* void inOrder();
* void postOrder(BSTNode<Keys, T> \*p, ofstream &f);
* void postOrder();
* Lớp từ điển
* Các thuộc tính
* BSTree<string, string> b;
* Các phương thức
* void docFile();
* void ghiFile();
* void themTuMoi(string key, string elem);
* void xoaMotTu(string key);
* void capNhatNghia(string key);
* void timKiem(string key);
* Mô tả chức năng của từng lớp, từng phương thức
* Lớp BSTNode
* Lớp BSTNode định nghĩa ra một nút trong cây nhị phân.
* **Chức năng của các phương thức trong lớp**
* BSTNode(): Hàm tạo không đối gán giá trị khởi đầu cho các thuộc tính trong lớp.
* BSTNode(Keys k, T e, BSTNode<Keys, T> \*parent): Hàm tạo có đối gán giá trị cho một nút mới trong cây.
* BSTNode<Keys, T> \*getParent(): Trả lại con trỏ trỏ đến cha của nút hiện tại.
* BSTNode<Keys, T> \*getLeft(): Trả lại con trỏ trỏ đến nút con trái của nút hiện tại.
* BSTNode<Keys, T> \*getRight(): Trả lại con trỏ trỏ đến nút con phải của nút hiện tại.
* void setLeft(BSTNode<Keys, T> \*p): Gán giá trị nút trái cho nút hiện tại.
* void setRight(BSTNode<Keys, T> \*p): Gán giá trị nút phải cho nút hiện tại.
* void setParent(BSTNode<Keys, T> \*p): Gán giá trị nút cha cho nút hiện tại.
* int hasLeft(): Kiểm tra có con bên trái hay không.
* int hasRight(): Kiểm tra có con bên phải hay không.
* T getElement(): Trả về nghĩa tiếng Việt của từ tiếng Anh lưu ở nút hiện tại.
* Keys getKey(): Trả về từ tiếng Anh đang lưu ở nút hiện tại.
* void setElement(T e): Gán nghĩa của từ tiếng Anh lưu ở nút hiện tại.
* void setKey(Keys k): Gán từ tiếng Anh lưu ở nút hiện tại.
* Lớp BSTree
* Lớp BSTree quản lí các nút có trong cây
* **Chức năng của các phương thức trong lớp**
* BSTree(): Gán giá trị khởi đầu cho nút gốc và giá trị n để đếm số nút hiện tại trong cây.
* int size(): Trả về số nút có trong cây.
* int isEmpty(): Kiểm tra cây rỗng.
* int isInternal(BSTNode<Keys, T> \*p): Kiểm tra nút trong.
* int isExternal(BSTNode<Keys, T> \*p): Kiểm tra nút lá.
* int isRoot(BSTNode<Keys, T> \*p): Kiểm tra nút gốc.
* BSTNode<Keys, T> \*search(Keys k, BSTNode<Keys, T> \*p): Trả về con trỏ đến nút có key k nếu tìm thấy và NULL nếu ngược lại.
* BSTNode<Keys, T> \*getRoot(): Trả về con trỏ đến nút gốc của cây
* bool search(Keys k): Trả về true nếu tìm thấy và false nếu ngược lại.
* BSTNode<Keys, T> \*insert(Keys k, T e, BSTNode<Keys, T> \*temp, BSTNode<Keys, T> \*parent): Thêm một nút vào cây.
* void insert(Keys k, T e): Gọi hàm thêm một nút vào cây với đối truyền vào là gốc.
* BSTNode<Keys, T> \*getMinRight(BSTNode<Keys, T> \*p): Trả về nút có giá trị lớn hơn nhỏ nhất của nút hiện tại.
* void remove(BSTNode<Keys, T> \*tmp): Xóa nút không có con hoặc có một trong hai con.
* void remove(Keys k): Xóa nút có cả hai con.
* void preOrder(BSTNode<Keys, T> \*p, ofstream &f): Duyệt đầu cây nhị phân tìm kiếm và ghi vào file output.txt.
* void preOrder(): Gọi hàm duyệt đầu.
* void inOrder(BSTNode<Keys, T> \*p, ofstream &f): Duyệt giữa cây nhị phân tìm kiếm và ghi vào file output.txt.
* void inOrder(): Gọi hàm duyệt giữa.
* void postOrder(BSTNode<Keys, T> \*p, ofstream &f): Duyệt cuối cây nhị phân tìm kiếm và ghi vào file output.txt.
* void postOrder(): Gọi hàm duyệt cuối.
* Lớp từ điển
* Lớp từ điển giúp người dùng quản lí từ điển sau khi đọc từ file
* **Chức năng của các phương thức trong lớp**
* void docFile(): Đọc từ điển từ file vào cây.
* void ghiFile(): Ghi từ điển từ cây vào file.
* void themTuMoi(string key, string elem): Thêm một từ vào cây lưu từ điển.
* void xoaMotTu(string key): Xóa một từ trong cây từ điển.
* void capNhatNghia(string key): Cập nhật nghĩa cho một từ có trong cây.
* void timKiem(string key): Tìm kiếm một từ trong cây từ điển.

## Cài đặt các lớp và hàm main bằng C++

Link to code: <https://github.com/vinhtruong204/DSA-UTC>.

## Phân tích thời gian chạy của từng phương thức có trong các lớp

* Lớp BSTNode

template <class Keys, class T>

BSTNode<Keys, T>::BSTNode()

{

// Gán các giá trị khởi đầu cho các thuộc tính O(1).

this->parent = NULL;

this->right = NULL;

this->left = NULL;

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T>::BSTNode(Keys k, T e, BSTNode<Keys, T> \*parent) {

// Gán các giá trị được truyền vào O(1).

this->key = k;

this->element = e;

this->parent = parent;

this->right = NULL;

this->left = NULL;

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTNode<Keys, T>::getParent()

{

return this->parent; // Truy cập thuộc tính parent và trả về kết quả O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTNode<Keys, T>::getLeft()

{

return this->left; // Truy cập thuộc tính left và trả về kết quả O(1);

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTNode<Keys, T>::getRight()

{

return this->right; // Truy cập thuộc tính right và trả về kết quả O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTNode<Keys, T>::setLeft(BSTNode<Keys, T> \*p)

{

this->left = p; // Truy cập left và gán giá trị p cho left O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTNode<Keys, T>::setRight(BSTNode<Keys, T> \*p)

{

this->right = p; // Truy cập right và gán giá trị p cho right O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTNode<Keys, T>::setParent(BSTNode<Keys, T> \*p)

{

this->parent = p; // Truy cập parent và gán giá trị p cho parent O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTNode<Keys, T>::hasLeft()

{

return this->left != NULL; // Truy cập left, so sánh và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTNode<Keys, T>::hasRight()

{

return this->right != NULL; // Truy cập right, so sánh và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

T BSTNode<Keys, T>::getElement()

{

return this->element; // Truy cập element và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

Keys BSTNode<Keys, T>::getKey()

{

return this->key; // Truy cập key và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTNode<Keys, T>::setElement(T e)

{

this->element = e; // Truy cập element và gán giá trị O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTNode<Keys, T>::setKey(Keys k)

{

this->key = k; // Truy cập key và gán giá trị O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).
* Lớp BSTree

template <class Keys, class T>

BSTree<Keys, T>::BSTree()

{

root = NULL; // Gán giá trị cho root O(1).

n = 0; // Gán giá trị cho n O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTree<Keys, T>::getRoot()

{

return this->root; // Truy cập root và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTree<Keys, T>::isEmpty()

{

return this->root != NULL; // Truy cập root, so sánh và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTree<Keys, T>::size()

{

return this->n; // Truy cập n và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTree<Keys, T>::isInternal(BSTNode<Keys, T> \*p)

{

return (p->hasLeft() || p->hasRight()); // Gọi phương thức hasLeft có 4 phép toán, hasRight có 4 phép toán sử dụng toán tử || và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTree<Keys, T>::isExternal(BSTNode<Keys, T> \*p)

{

return !(p->hasLeft() && p->hasRight()); // 11: Gọi phương thức hasLeft có 4 phép toán, hasRight có 4 phép toán sử dụng toán tử &&, phép phủ định và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

int BSTree<Keys, T>::isRoot(BSTNode<Keys, T> \*p)

{

return p->getParent() == NULL; // Gọi phương thức getParent có 3 phép toán, so sánh với NULL và trả về O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTree<Keys, T>::search(Keys k, BSTNode<Keys, T> \*p)

{

if (p == NULL) // So sánh con trỏ p với NULL O(1).

{

return NULL; // Trả về con trỏ NULL O(1).

}

if (p->getKey() == k) // So sánh key của p với key cần tìm kiếm O(1).

{

return p; // Trả về con trỏ p O(1).

}

else if (p->getKey() < k) // So sánh key của p với key cần tìm O(1).

{

return search(k, p->getRight()); // Tìm kiếm cây con bên phải O(h), h là chiều cao của cây.

}

else

{

return search(k, p->getLeft()); // Tìm kiếm cây con bên trái O(h), h là chiều cao của cây.

}

}

* Phương thức có độ phức tạp tiệm cận O(h).

template <class Keys, class T>

bool BSTree<Keys, T>::search(Keys k)

{

return search(k, root) != NULL; // Gọi hàm search(k, root), so sánh và trả về O(h).

}

* Phương thức có độ phức tạp tiệm cận O(h).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTree<Keys, T>::insert(Keys k, T e, BSTNode<Keys, T> \*temp, BSTNode<Keys, T> \*parent)

{

if (temp == NULL) // Kiểm tra xem nút hiện tại (temp) có phải là NULL hay không, có độ phức tạp O(1).

{

// Make root

temp = new BSTNode<Keys, T>(k, e, parent); // Nếu nút hiện tại là NULL, tạo một nút mới và trả về nút mới, có độ phức tạp O(1).

}

else

{

if (k < temp->getKey())

{

// Nếu giá trị khóa của nút mới nhỏ hơn giá trị khóa của nút hiện tại, gọi phương thức insert đệ quy để chèn nút mới vào cây con bên trái của nút hiện tại.

temp->setLeft(insert(k, e, temp->getLeft(), temp));

}

else if (k > temp->getKey())

{

// Nếu giá trị khóa của nút mới nhỏ hơn giá trị khóa của nút hiện tại, gọi phương thức insert đệ quy để chèn nút mới vào cây con bên trái của nút hiện tại.

temp->setRight(insert(k, e, temp->getRight(), temp));

}

}

* Độ phức tạp của hai phép so sánh trên là O(log n).

return temp; // Trả về nút hiện tại có độ phức tạp là O(1).

}

* Phương thức có độ phức tạp tiệm cận O(log n).

template <class Keys, class T>

void BSTree<Keys, T>::insert(Keys k, T e)

{

root = insert(k, e, root, NULL); // Gọi hàm insert và truyền vào các đối có độ phức tạp trung bình là O(log n);

n++; // Tăng gia trị biến đếm có độ phức tạp là O(1).

}

* Phương thức có độ phức tạp tiệm cận O(log n).

template <class Keys, class T>

BSTNode<Keys, T> \*BSTree<Keys, T>::getMinRight(BSTNode<Keys, T> \*p)

{

BSTNode<Keys, T> \*temp = p; // Gán giá trị p cho temp O(1).

while (temp->getLeft() != NULL) // Lấy ra con trỏ left và so sánh h lần (h là chiều cao của cây) O(h).

{

temp = temp->getLeft(); // Gán temp bằng con trỏ bên trái O(1).

}

return temp; // Trả về temp O(1).

}

* Phương thức có độ phức tạp tiệm cận O(h).

template <class Keys, class T>

void BSTree<Keys, T>::remove(BSTNode<Keys, T> \*tmp)

{

if (tmp == NULL) // So sánh con trỏ và trả về O(1).

return;

// Xoa nut la

if (!tmp->hasLeft() && !tmp->hasRight()) // Kiểm tra cây con bên trái, phải O(1).

{

// Lấy ra nút cha O(1).

BSTNode<Keys, T> \*p = tmp->getParent();

// Kiểm tra nút đang cần xóa bên trái hay phải nút cha O(1).

if (p->getLeft() == tmp)

{

p->setLeft(NULL);

}

else

{

p->setRight(NULL);

}

}

// Xóa và dồn cây bên trái lên O(1).

else if (tmp->hasLeft() && !tmp->hasRight())

{

// Lấy ra nút cha của nút hiện tại O(1).

BSTNode<Keys, T> \*p = tmp->getParent();

// Cập nhật nút cha cho nút con bên trái hiẹn tại O(1).

tmp->getLeft()->setParent(p);

// Kiểm tra nút con bên trái hay phải nút cha và tiến hành dồn cây O(1).

if (p->getLeft() == tmp)

{

p->setLeft(tmp->getLeft());

}

else

{

p->setRight(tmp->getLeft());

}

}

// Dồn cây bên phải lên O(1).

else if (!tmp->hasLeft() && tmp->hasRight())

{

// Lấy ra nút cha O(1).

BSTNode<Keys, T> \*p = tmp->getParent();

// Cập nhật nút cha cho nút con bên trái hiẹn tại O(1).

tmp->getRight()->setParent(p);

// Kiểm tra nút con bên trái hay phải nút cha và tiến hành dồn cây O(1).

if (p->getLeft() == tmp)

{

p->setLeft(tmp->getRight());

}

else

{

p->setRight(tmp->getRight());

}

}

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class Keys, class T>

void BSTree<Keys, T>::remove(Keys k)

{

// Tìm nút có khóa k O(log n).

BSTNode<Keys, T> \*tmp = search(k, root);

// So sánh con trỏ vừa tìm được O(1).

if (tmp == NULL)

return;

// Trường hợp cần xóa cả hai con O(h).

if (tmp->hasLeft() && tmp->hasRight())

{

// Lấy ra nút nhỏ nhất bên phải nút cần xóa O(h)

BSTNode<Keys, T> \*minRight = getMinRight(tmp->getRight());

// Gán giá trị nút cần xóa là giá trị nút minRight O(1).

tmp->setKey(minRight->getKey());

tmp->setElement(minRight->getElement());

// Xóa nút minRight O(1).

remove(minRight);

}

// Xóa nút trong các trường hợp còn lại O(1).

else

{

remove(tmp);

}

n--; // Giảm giá trị của biến lưu các nút O(1).

}

* Phương thức có độ phức tạp tiệm cận O(h) vì cây tìm kiếm không cân bằng.

template <class Keys, class T>

void BSTree<Keys, T>::preOrder(BSTNode<Keys, T> \*p, ofstream &f)

{

// Root -> Left -> Right

if (p != NULL) // So sánh giá trị p với NULL O(1).

{

f << p->getKey() << "\t" << p->getElement() << "\n"; // Ghi giá trị nút hiện tại vào file O(1).

preOrder(p->getLeft(), f); // Gọi đệ quy đến cây con bên trái O(n / 2).

preOrder(p->getRight(), f); // Gọi đệ quy đến cây con bên phải O(n / 2).

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

template <class Keys, class T>

void BSTree<Keys, T>::inOrder(BSTNode<Keys, T> \*p, ofstream &f)

{

// Left -> Root -> Right

if (p != NULL) // So sánh giá trị p với NULL O(1).

{

inOrder(p->getLeft(), f); // Gọi đệ quy đến cây con bên trái O(n / 2).

f << p->getKey() << "\t" << p->getElement() << "\n"; // Ghi giá trị nút hiện tại vào file O(1).

inOrder(p->getRight(), f); // Gọi đệ quy đến cây con bên phải O(n / 2).

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

template <class Keys, class T>

void BSTree<Keys, T>::postOrder(BSTNode<Keys, T> \*p, ofstream &f)

{

// Left -> Right -> Root

if (p != NULL) // So sánh giá trị p với NULL O(1).

{

postOrder(p->getLeft(), f); // Gọi đệ quy đến cây con bên trái O(n / 2).

postOrder(p->getRight(), f); // Gọi đệ quy đến cây con bên phải O(n / 2).

f << p->getKey() << "\t" << p->getElement() << "\n";

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

template <class Keys, class T>

void BSTree<Keys, T>::preOrder()

{

ofstream file("../Datas/output.txt"); // Mở file output O(1).

preOrder(root, file); // Gọi hàm preOrder và truyền đối ghi vào file O(n).

file.close(); // Đóng file.

}

* Phương thức có độ phức tạp tiệm cận O(n).

template <class Keys, class T>

void BSTree<Keys, T>::inOrder()

{

ofstream file("../Datas/output.txt"); // Mở file output O(1).

inOrder(root, file); // Gọi hàm inOrder và truyền đối ghi vào file O(n).

file.close(); // Đóng file.

}

* Phương thức có độ phức tạp tiệm cận O(n).

template <class Keys, class T>

void BSTree<Keys, T>::postOrder()

{

ofstream file("../Datas/output.txt"); // Mở file output O(1).

postOrder(root, file); // Gọi hàm postOrder và truyền đối ghi vào file O(n).

file.close(); // Đóng file.

}

* Phương thức có độ phức tạp tiệm cận O(n).
* Lớp từ điển

void TuDien::docFile()

{

fstream file("../Datas/input.txt"); // Mở file input O(1).

if (file.fail()) // Kiểm tra file chưa mở được O(1).

return;

string key = "", elem = ""; // Gán giá trị cho hai biến tạm O(1).

while (getline(file, key) && getline(file, elem)) // Đọc file O(n)

{

b.insert(key, elem); // Thêm một nút vào cây O(log n).

}

file.close(); // Đóng file O(1)

}

* Phương thức có độ phức tạp tiệm cận O(n).

void TuDien::ghiFile()

{

b.inOrder(); // Gọi hàm duyệt giữa O(n).

}

* Phương thức có độ phức tạp tiệm cận O(n).

void TuDien::themTuMoi(string key, string elem)

{

b.insert(key, elem); // Thêm một từ mới vào cây O(log n).

}

* Phương thức có độ phức tạp tiệm cận O(log n).

void TuDien::xoaMotTu(string key)

{

b.remove(key); // Xóa một từ trong cây O(h).

}

* Phương thức có độ phức tạp tiệm cận O(h).

void TuDien::capNhatNghia(string key)

{

if (b.search(key)) // Tìm kiếm từ O(h)

{

// Khai báo và nhập nghĩa mới O(1).

string temp = "";

cout << "Nhap nghia moi: ";

getline(cin, temp);

// Tìm theo key cập nhật nghĩa mới O(h).

b.search(key, b.getRoot())->setElement(temp);

}

else

{

cout << "Khong tim thay tu " << key << " trong tu dien!\n";

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

void TuDien::timKiem(string key)

{

if (b.search(key)) // Tìm kiếm một từ O(h).

{

cout << "Tim thay tu " << key << " trong tu dien:\n";

cout << b.search(key, b.getRoot())->getKey() << ": " << b.search(key, b.getRoot())->getElement() << "\n"; // Tìm kiếm và in ra từ và nghĩa của từ O(h).

}

else

{

cout << "Khong tim thay tu " << key << " trong tu dien!\n";

}

}

* Phương thức có độ phức tạp tiệm cận O(h).

# BÀI 2 (BÀI SỐ 36 TRONG DANH SÁCH BÀI TẬP)

## 2.1. Đề bài

1. Cài đặt cấu trúc dữ liệu trừu tượng Vector.
2. Ứng dụng Vector xây dựng lớp quản lý học sinh. Mỗi đối tượng học sinh có các thuộc tính sau: Mã học sinh, họ tên, giới tính, tuổi, điểm toán, điểm lý, điểm hóa, điểm trung bình và xếp loại.

* Xếp loại được tính như sau:
* Giỏi: nếu điểm trung bình lớn hơn hoặc bằng 8.
* Khá: nếu điểm trung bình nhỏ hơn 8 và lớn hơn hoặc bằng 7.
* Trung Bình: nếu điểm trung bình nhỏ hơn 7 và lớn hơn hoặc bằng 5.
* Yếu: nếu điểm trung bình nhỏ hơn 5.
* Lớp có các phương thức sau:

1. Đọc danh sách học sinh từ file (pupil.txt) chứa danh sách học sinh.
2. Thêm một học sinh.
3. Cập nhật thông tin học sinh khi biết Mã học sinh.
4. Xóa thí sinh bởi Mã học sinh.
5. Tìm kiếm thí sinh theo tên.
6. Sắp xếp thí sinh theo điểm trung bình (GPA).
7. Sắp xếp thí sinh theo tên.
8. Hiển thị danh sách thí sinh.
9. Ghi danh sách thí sinh vào file "pupil.txt".

## 2.2. Phân tích bài toán

* Yêu cầu của bài toán
* Cài đặt cấu trúc dữ liệu trừu tượng Vector.
* Ứng dụng Vector xây dựng lớp quản lý học sinh. Mỗi đối tượng học sinh có các thuộc tính sau: Mã học sinh, họ tên, giới tính, tuổi, điểm toán, điểm lý, điểm hóa, điểm trung bình và xếp loại.
* Xác định các lớp, các thuộc tính, phương thức của lớp
* Lớp vector
* Các thuộc tính
* int capacity, number;
* T \*arr;
* Các phương thức
* MyVector();
* MyVector(int k, T x);
* ~MyVector();
* int getCapacity();
* int size();
* bool empty();
* void pop\_back();
* void erase(T \*it);
* void resize(int newCapacity);
* T &back();
* T &operator[](int k);
* void push\_back(T x);
* void insert(int k, T x)
* MyVector &operator=(MyVector<T> a);
* T\* begin();
* T\* end();
* T\* rbegin();
* T\* rend();
* Lớp học sinh
* Các thuộc tính
* int maHocSinh, tuoi;
* string hoTen, gioiTinh, xepLoai;
* float diemToan, diemLy, diemHoa, diemTB;
* Các phương thức
* void xetXepLoai();
* HocSinh(int maHocSinh = 0, string hoTen = "", string gioiTinh = "", int tuoi = 0, float diemToan = 0, float diemLy = 0, float diemHoa = 0);
* int getMaHocsinh();
* string getHoTen();
* string getTen();
* float getDiemTb();
* ~HocSinh();
* Lớp quản lý
* Các thuộc tính
* MyVector<HocSinh> ds;
* int n;
* Các phương thức
* void docDanhSachHocSinh();
* void themMotHocSinh(HocSinh a);
* void capNhatThongTin(int maHocSinh);
* void xoaHocSinh(int maHocSinh);
* void timKiem(string tenHocSinh);
* void sapXepTheoDiemTb();
* void sapXepTheoTen();
* void hienDanhSachHocSinh();
* void ghiDanhSachHocSinh();
* Mô tả chức năng của từng lớp, từng phương thức
* Lớp vector
* Lớp vector định nghĩa cấu trúc dữ liệu vector trừu tượng giúp người dùng quản lí danh sách học sinh.
* **Chức năng của các phương thức trong lớp**
* MyVector(): Hàm tạo không đối gán giá trị khởi đầu cho các thuộc tính.
* MyVector(int k, T x): Hàm tạo có đối cấp phát và gán k giá trị x cho mảng
* ~MyVector(): Giải phóng bộ nhớ.
* int getCapacity(): Trả về sức chứa hiện tại của vector.
* int size(): Trả về số phần tử hiện đang được lưu trong vector.
* bool empty(): Kiểm tra vector rỗng.
* void pop\_back(): Xóa phần tử cuối cùng.
* void erase(T \*it): Xóa phần tử ở một vị trí bất kì.
* void resize(int newCapacity): Cấp phát lại bộ nhớ cho mảng dữ liệu với sức chứa mới.
* T &back(): Trả về phần tử cuối của mảng.
* T &operator[](int k): Nạp chồng toán tử truy xuất dữ liệu thông qua vị trí.
* void push\_back(T x): Đẩy phần tử mới vào cuối vào vector.
* void insert(int k, T x): Chèn phần tử vào vị trí k.
* MyVector &operator=(MyVector<T> a): Nạp chồng toán tử bằng cho hai vector.
* T\* begin(): Trả về con trỏ đến đầu mảng của vector.
* T\* end(): Trả về con trỏ sau phần tử cuối của mảng.
* T\* rbegin(): Trả về con trỏ sau phần tử cuối của mảng.
* T\* rend(): Trả về con trỏ đến đầu mảng của vector.
* Lớp học sinh
* Lớp học sinh giúp người dùng quản lý các thông tin của một đối tượng học sinh.
* **Chức năng của các phương thức trong lớp**
* void xetXepLoai(): Xếp loại học sinh dựa trên điểm nhập vào.
* HocSinh(int maHocSinh = 0, string hoTen = "", string gioiTinh = "", int tuoi = 0, float diemToan = 0, float diemLy = 0, float diemHoa = 0): Hàm tạo có đối mặc định truyền gán giá trị thuộc tính vào cho đối tượng học sinh.
* int getMaHocsinh(): Trả về mã học sinh.
* string getHoTen(): Trả về họ và tên của học sinh.
* string getTen(): Trả về tên của học sinh.
* float getDiemTb(): Trả về điểm trung bình ba môn của học sinh.
* ~HocSinh(): Gán giá trị mặc định cho học sinh.
* Lớp quản lý
* Giúp người dùng quản lý các đối tượng học sinh.
* **Chức năng của các phương thức trong lớp**
* void docDanhSachHocSinh(): Đọc danh sách học sinh từ file pupil.txt vào vector học sinh.
* void themMotHocSinh(HocSinh a): Thêm một học sinh vào vector học sinh.
* void capNhatThongTin(int maHocSinh): Cập nhật thông tin một học sinh.
* void xoaHocSinh(int maHocSinh): Xóa một học sinh khỏi danh sách.
* void timKiem(string tenHocSinh): Tìm kiếm học sinh theo tên.
* void sapXepTheoDiemTb(): Sắp xếp danh sách học sinh theo điểm trung bình giảm dần.
* void sapXepTheoTen(): Sắp xếp danh sách học sinh theo điểm trung bình tăng dần.
* void hienDanhSachHocSinh(): Hiển thị thông tin danh sách học sinh lên màn hình.
* void ghiDanhSachHocSinh(): Ghi danh sách học sinh vào file pupil\_output.txt.

## 2.3. Cài đặt các lớp và hàm main bằng C++

Link to code: <https://github.com/vinhtruong204/DSA-UTC>.

## 2.4. Phân tích thời gian chạy của từng phương thức có trong các lớp

* Lớp vector

template <class T>

MyVector<T>::MyVector() // Gán giá trị khởi đầu cho các thuộc tính O(1).

{

this->capacity = 0;

this->number = 0;

this->arr = NULL;

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

MyVector<T>::MyVector(int k, T x)

{

this->capacity = k; // Gán giá trị cho sức chứa vector O(1).

this->number = k; // Gán giá trị cho biến biểu diễn số phần tử hiện tại trong vector O(1).

this->arr = new T[k]; // Cấp phát bộ nhớ cho k phần tử O(k).

for (int i = 0; i < k; i++) // Gán giá trị cho k phần tử O(k).

{

this->arr[i] = x;

}

}

* Phương thức có độ phức tạp tiệm cận O(k).

template <class T>

MyVector<T>::~MyVector()

{

if (arr != NULL) // So sánh giá trị arr với NULL O(1).

{

delete[] arr; // Giải phóng bộ nhớ O(capacity) capacity là sức chứa.

}

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

template <class T>

int MyVector<T>::getCapacity()

{

return this->capacity; // Trả về giá trị sức chứa O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

int MyVector<T>::size()

{

return this->number; // Trả về biến lưu số phần tử hiện tại trong vector O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

bool MyVector<T>::empty()

{

return this->number == 0; // Trả về giá trị sức chứa và so sánh với 0 O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

void MyVector<T>::pop\_back()

{

if (!empty()) // Kiểm tra vector rỗng O(1)

{

this->number--; // Giảm số lượng phần tử O(1).

}

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

void MyVector<T>::resize(int newCapacity)

{

if (newCapacity < this->capacity) // Kiểm tra sức chứa mới và sức chứa hiện tại O(1).

{

return; // Nếu sức chứa mới nhỏ hơn hiện tại thì trả về O(1).

}

this->capacity = newCapacity; // Gán giá trị sức chứa mới cho sức chứa hiện tại O(1).

T \*temp = new T[this->capacity]; // Cấp phát bộ nhớ cho mảng tạm O(capacity).

for (int i = 0; i < number; i++)

{

temp[i] = arr[i]; // Gán các giá trị của mảng cũ cho mảng mới O(number).

}

if (this->arr != NULL) // Kiểm tra mảng cũ khác rỗng O(1).

{

delete[] this->arr; // Xóa mảng cũ O(capacity).

}

this->arr = temp; // Gán mảng cũ bằng mảng có sức chứa mới O(1).

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

template <class T>

T &MyVector<T>::back()

{

return this->arr[number - 1]; // Trả về phần tử cuối vector O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

T &MyVector<T>::operator[](int k)

{

return this->arr[k]; // Trả về phần tử thứ k O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

void MyVector<T>::push\_back(T x)

{

if (this->number == this->capacity) // Kiểm tra giới hạn lưu trữ vector O(1).

{

resize(this->capacity \* 2 + 2); // Gọi hàm resize mở rộng vùng nhớ O(capacity \* 2 + 2).

}

arr[number] = x; // Gán giá trị vào cuối mảng O(1).

number++; // Tăng số phần tử trong vector O(1).

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

template <class T>

void MyVector<T>::insert(int k, T x)

{

if (this->number == this->capacity) // Kiểm tra giới hạn lưu trữ vector O(1).

{

resize(this->capacity \* 2 + 2); // Gọi hàm resize mở rộng vùng nhớ O(capacity \* 2 + 2).

}

for (int i = this->number; i > k; i--)

{

this->arr[i] = this->arr[i - 1]; // Tiến hành dồn mảng về sau O(n – k).

}

this->arr[k] = x; // Gán giá trị vào vị trí thứ k O (1).

this->number++; // Tăng biến đếm số lượng phần tử O(1).

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

template <class T>

MyVector<T> &MyVector<T>::operator=(MyVector<T> a)

{

this->capacity = a.capacity; // Gán giá trị sức chứa của vector hiện tại với sức chứa của vector được truyền vào O(1).

this->number = a.number; // Gán giá trị số phần tử của vector hiện tại với số phần tử vector được truyền vào O(1).

if (capacity) // Kiểm tra sức chứa O(1).

{

this->arr = new T[capacity]; // Cấp phát bộ nhớ cho vector hiện tại O(capacity).

for (int i = 0; i < number; i++)

{

this->arr[i] = a.arr[i]; // Gán giá trị cho mảng của vector hiện tại O(number).

}

}

else

{

this->arr = 0; // Gán giá trị mặc định cho mảng dữ liệu O(1);

}

return \*this; // Trả về vector hiện tại O(1).

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

template <class T>

T \*MyVector<T>::begin()

{

return this->arr; // Trả về con trỏ đầu tiên của mảng dữ liệu O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

T \*MyVector<T>::end()

{

return this->arr + this->number; // Trả về con trỏ sau phần tử cuối cùng của mảng dữ liệu O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

T \*MyVector<T>::rbegin()

{

return this->arr + this->number; // Trả về con trỏ sau phần tử cuối cùng của mảng dữ liệu O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

T \*MyVector<T>::rend()

{

return this->arr; // Trả về con trỏ đầu tiên của mảng dữ liệu O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

template <class T>

void MyVector<T>::erase(T \*it)

{

if (number == 0) // Kiểm tra số phần tử của mảng dữ liệu O(1).

return; // Trả về nếu như mảng không có phần tử nào O(1).

for (it; it != arr + number; it++)

{

\*it = \*(it + 1); // Dồn mảng từ vị trí cuối mảng về vị trí it O(number).

}

number--; // Giảm giá trị của biến lưu số lượng phần tử còn lại O(1).

}

* Phương thức có độ phức tạp tiệm cận O(number).
* Lớp học sinh

HocSinh::HocSinh(int maHocSinh, string hoTen, string gioiTinh, int tuoi, float diemToan, float diemLy, float diemHoa)

{

// Gán các giá trị khởi đầu và xét xếp loại O(1).

this->maHocSinh = maHocSinh;

this->hoTen = hoTen;

this->gioiTinh = gioiTinh;

this->tuoi = tuoi;

this->diemToan = diemToan;

this->diemLy = diemLy;

this->diemHoa = diemHoa;

this->diemTB = (diemToan + diemLy + diemHoa) / 3;

this->xetXepLoai();

}

* Phương thức có độ phức tạp tiệm cận O(1).

void HocSinh::xetXepLoai()

{

// So sánh điểm và xếp loại O(1)

if (diemTB >= 8.0f && diemTB <= 10.0f)

{

xepLoai = "Gioi";

}

else if (diemTB >= 7.0f && diemTB <= 8.0f)

{

xepLoai = "Kha";

}

else if (diemTB >= 5.0f && diemTB <= 7.0f)

{

xepLoai = "Trung binh";

}

else if (diemTB >= 0.0f && diemTB <= 5.0f)

{

xepLoai = "Yeu";

}

}

* Phương thức có độ phức tạp tiệm cận O(1).

HocSinh::~HocSinh()

{

// Gán các giá trị mặc định cho đối tượng O(1).

this->maHocSinh = 0;

this->hoTen = "";

this->gioiTinh = "";

this->tuoi = 0;

this->diemToan = 0;

this->diemLy = 0;

this->diemHoa = 0;

this->diemTB = 0;

this->xepLoai = "";

}

string HocSinh::getHoTen()

{

return this->hoTen; // Trả về tên của học sinh O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

string HocSinh::getTen()

{

stack<string> x; // Khởi tạo một stack O(1).

stringstream ss(hoTen); // Khởi tạo stringstream để phân tách chuỗi họ tên O(n), n là độ dài của họ tên.

string temp; // Khai báo chuỗi temp O(1).

while (ss >> temp) // Vòng lặp đến khi tách hết chuỗi họ tên O(n).

{

x.push(temp); // Đẩy chuỗi con vừa đọc vào stack O(1).

}

return x.top(); // Trả về tên ở đỉnh stack O(1).

}

* Phương thức có độ phức tạp tiệm cận O(n), n là độ dài của họ tên.

int HocSinh::getMaHocsinh()

{

return this->maHocSinh; // Trả về mã học sinh O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

float HocSinh::getDiemTb()

{

return this->diemTB; // Trả về điểm trung bình O(1).

}

* Phương thức có độ phức tạp tiệm cận O(1).

istream &operator>>(istream &inp, HocSinh &a)

{

// Nhập vào các thông tin O(1).

cout << "\nMa hoc sinh: ";

inp >> a.maHocSinh;

inp.ignore();

cout << "Ho ten: ";

getline(inp, a.hoTen);

cout << "Gioi tinh: ";

getline(inp, a.gioiTinh);

cout << "Tuoi: ";

inp >> a.tuoi;

cout << "Diem toan: ";

inp >> a.diemToan;

cout << "Diem ly: ";

inp >> a.diemLy;

cout << "Diem hoa: \n";

inp >> a.diemHoa;

inp.ignore();

a.diemTB = (a.diemHoa + a.diemLy + a.diemToan) / 3; // Tính điểm trung bình O(1).

a.xetXepLoai(); // Gọi hàm xét xếp loại O(1).

return inp;

}

* Phương thức có độ phức tạp tiệm cận O(1).

ostream &operator<<(ostream &out, HocSinh a)

{

// Xuất ra màn hình các thông tin O(1).

out << fixed << setprecision(1) << left << setw(10) << a.maHocSinh << "\t|" << setw(25) << a.hoTen << "\t|" << setw(10) << a.gioiTinh << "\t|" << setw(5) << a.tuoi << "\t|" << setw(5) << a.diemToan << "\t|" << setw(5) << a.diemLy << "\t|" << setw(5) << a.diemHoa << "\t|" << setw(5) << a.diemTB << "\t|" << setw(10) << a.xepLoai << "\t\n";

return out;

}

* Phương thức có độ phức tạp tiệm cận O(1).
* Lớp quản lý

void QuanLy::docDanhSachHocSinh()

{

fstream file("../Datas/pupil.txt"); // Mở file O(1).

if (file.fail()) // Kiểm tra file đã được mở hay chưa O(1).

{

cout << "Khong the mo file!\n";

return; // Trả về nếu như file chưa được mở O(1).

}

file >> n; // Đọc n từ file O(1);

cin.ignore();

for (int i = 0; i < n; i++)

{

HocSinh t;

file >> t; // Nhập học sinh vào file O(1).

ds.push\_back(t); // Đẩy học sinh vừa đọc được vào vector O(capacity).

}

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

void QuanLy::themMotHocSinh(HocSinh a)

{

ds.push\_back(a); // Đẩy học sinh vừa đọc được vào vector O(capacity).

}

* Phương thức có độ phức tạp tiệm cận O(capacity).

void QuanLy::capNhatThongTin(int maHocSinh)

{

for (HocSinh &x : ds) // Duyệt danh sách học sinh O(n), n là số học sinh.

{

if (x.getMaHocsinh() == maHocSinh) // So sánh mã học sinh O(1).

{

cout << "Cap nhat thong tin cho hoc sinh " << x.getHoTen() << ":\n";

cin >> x; // Nhập thông tin nếu tìm thấy học sinh O(1).

}

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

void QuanLy::xoaHocSinh(int maHocSinh)

{

for (int i = 0; i < n; i++) // Duyệt danh sách học sinh O(n).

{

if (ds[i].getMaHocsinh() == maHocSinh) // So sánh mã học sinh O(n).

{

ds.erase(ds.begin() + i); //Xóa phần tử tại vị trí thứ i trong vector O(n);

}

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

void QuanLy::timKiem(string tenHocSinh)

{

// Gán các giá trị tìm kiếm O(1).

bool flag = false;

int cnt = 0;

for (auto x : ds) // Duyệt danh sách học sinh O(n).

{

if (x.getHoTen() == tenHocSinh) // So sánh hai chuỗi O(1).

{

// Nếu tìm thấy thì gán các giá trị tìm kiếm O(1).

flag = true;

cnt++;

}

}

if (flag) // Kiểm tra tìm thấy hay không O(1).

{

cout << "Tim thay " << cnt << " hoc sinh co ho ten " << tenHocSinh << ":\n";

for (auto x : ds) // Duyệt học sinh trong vector O(n).

{

if (x.getHoTen() == tenHocSinh) // So sánh tên học sinh O(1).

cout << x; // In ra các học sinh cần tìm O(1).

}

}

else

{

cout << "404 - Not found!\n"; // Thông báo lỗi nếu không tìm thấy O(1).

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

void QuanLy::sapXepTheoDiemTb()

{

for (int i = 0; i < n - 1; i++) // Duyệt danh sách n – 1 học sinh O(n - 1).

{

for (int j = i + 1; j < n; j++) // Duyệt danh sách n – i – 1 học sinh O(n - i - 1).

{

if (ds[i].getDiemTb() < ds[j].getDiemTb()) // So sánh điểm trung bình O(1).

{

// Hoán đổi hai đối tượng học sinh O(1).

HocSinh temp = ds[i];

ds[i] = ds[j];

ds[j] = temp;

}

}

}

}

* Phương thức có độ phức tạp tiệm cận O().

void QuanLy::sapXepTheoTen()

{

for (int i = 0; i < n - 1; i++) // Duyệt danh sách n – 1 học sinh O(n - 1).

{

for (int j = i + 1; j < n; j++) // Duyệt danh sách n – i – 1 học sinh O(n - i - 1).

{

if (ds[i].getTen() > ds[j].getTen()) // So sánh tên O(1).

{

// Hoán đổi hai đối tượng học sinh O(1).

HocSinh temp = ds[i];

ds[i] = ds[j];

ds[j] = temp;

}

}

}

}

* Phương thức có độ phức tạp tiệm cận O().

void QuanLy::hienDanhSachHocSinh()

{

for (HocSinh x : ds) // Duyệt qua danh sách học sinh O(n).

{

cout << x; // In ra học sinh O(1).

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

void QuanLy::ghiDanhSachHocSinh()

{

fstream file("../Datas/pupil\_output.txt"); // Mở file output O(1).

for (auto x : ds) // Duyệt qua danh sách học sinh O(n).

{

file << x; // Ghi học sinh vào file O(n).

}

}

* Phương thức có độ phức tạp tiệm cận O(n).

# DANH SÁCH TÀI LIỆU THAM KHẢO

1. Slide bài giảng trong trang web [lms.utc.edu.vn](https://lms.utc.edu.vn/) – Ngày tham khảo 27/10/2023.
2. Video bài giảng trong trang web [lms.utc.edu.vn](https://lms.utc.edu.vn/) – Ngày tham khảo 27/10/2023.
3. [mv15](https://auth.geeksforgeeks.org/user/mv15) (21/06/2022). [How to implement our own Vector Class in C++?](https://www.geeksforgeeks.org/how-to-implement-our-own-vector-class-in-c/) – Ngày tham khảo 28/10/2023.
4. [ProtoGO](https://www.youtube.com/@ProtoGO)(12/11/2020). [Implementation of Binary Search Tree using Template in C++(2020)](https://youtu.be/i0t63WWkfow?si=SVOoILRufgRBUsGx) – Ngày tham khảo 31/10/2023.