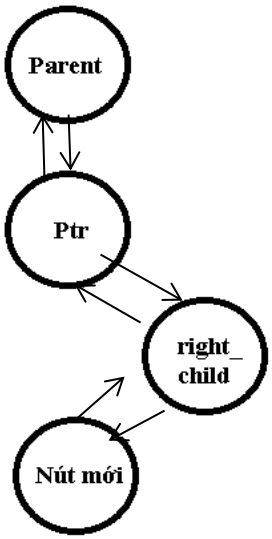
BÁO CÁO CHƯƠNG TRÌNH CÀI ĐẶT
Cây đỏ đen – Red Black Tree

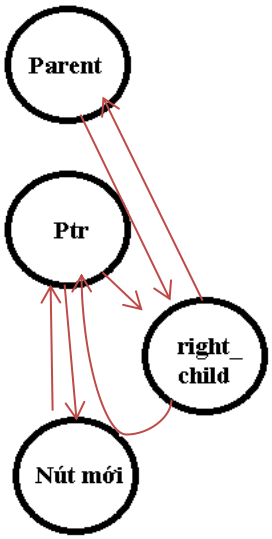
- **Hàm đọc file và main:** tương tự cây AVL
- **Khai báo cấu trúc của từ điển:**

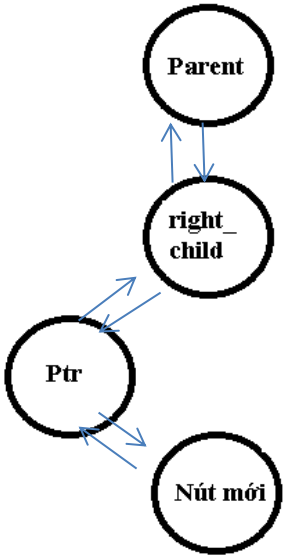
```
enum Color {RED, BLACK, DOUBLE_BLACK};                  // Các màu của nút (đỏ, đen, đen kép)

struct WORD                                             // Cấu trúc của một từ (từ và nghĩa)
{
	string m_Word;
	string m_Meaning;
};

struct Node                                             // Cấu trúc của một nút
                                                            (từ, màu, con trỏ trái, phải và cha)
{
	WORD _Word;
	int _Color;
	Node *_Left, *_Right, *_Parent;

	Node(WORD p)                                    // Khởi tạo một nút (mặc định nút mới màu đỏ)
	{
		_Word.m_Word = p.m_Word;
		_Word.m_Meaning = p.m_Meaning;
		_Color = RED;
		_Left = _Right = _Parent = NULL;
	}
};

class RBTree                                            // Cấu trúc từ điển (cây đỏ đen)
{
private:
	Node *root;                                     // Nút gốc
protected:
	void rotateLeft(Node *&);                       // Phép xoay trái
	void rotateRight(Node *&);                      // Phép xoay phải
	void fixInsert(Node *&);                        // Hàm cân bằng lại cây khi thêm nút mới
	void fixDelete(Node *&);                        // Hàm cân bằng lại cây khi xóa một nút
	void inorderBST(Node *&, ofstream&);            // Giúp in cây ra file theo thứ tự giữa
	int getColor(Node *&);                          // Hàm lấy màu của một nút
	void setColor(Node *&, int);                    // Hàm gán màu của một nút
	Node *minValue(Node *&);                        // Tìm kiếm nút thay thế (nhánh phải)
	Node *maxValue(Node *&);                        // Tìm kiếm nút thay thế (nhánh trái)
	Node *insertBST(Node *&, Node *&);              // Giúp thêm nút vào cây
	Node *deleteBST(Node *&, WORD);                 // Giúp xóa nút khỏi cây
	int getBlackHeight(Node *);                     // Tính chiều cao đen
	void clear(Node *&);                            // Hàm giúp xóa cây
	int getHeight(Node *);                          // Tính chiều cao từ một nút đến lá
public:
	static int countWords;                          // Đếm số lượng từ trong từ điển
	RBTree() { root = NULL; }                       // Khởi tạo từ điển ban đầu rỗng
	void insert(const WORD &);                      // Thêm nút vào từ điển
	void remove(WORD);                              // Xóa nút khỏi từ điển
	void inorder(string fileName);                  // In từ điển ra file với thứ tự giữa
	Node* search(string);                           // Tìm kiếm một từ trong từ điển
	void addUsage(Node *&, string);                 // Thêm cách dùng vào nghĩa của từ
	void edit(Node *&, string);                     // Chỉnh nghĩa của một từ
	string getMeaning(Node *);                      // Lấy nghĩa của một từ
	int getHeight();                                // Tính chiều cao của từ điển
	~RBTree();                                      // Xóa từ điển
};
```

- **Cài đặt từ điển:**

```
int RBTree::countWords = 0;                    // Khởi tạo số từ trong từ điển bằng 0
```

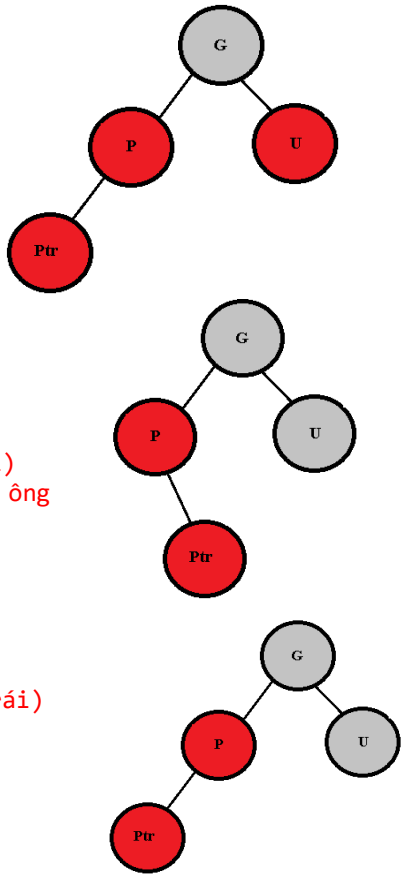







```
void RBTree::rotateLeft(Node *&ptr)            // Hàm xoay trái tại nút được trỏ bởi ptr
{
	Node *right_child = ptr->_Right;
	ptr->_Right = right_child->_Left;          // Cập nhật lại nhánh phải của ptr

	if (ptr->_Right != NULL)
		ptr->_Right->_Parent = ptr;            // Cập nhật lại cha của nút mới gán

	right_child->_Parent = ptr->_Parent;       // Cập nhật lại cha của nút con phải ban đầu

	if (ptr->_Parent == NULL)                  // Kiểm tra xem ptr có phải gốc không
		root = right_child;                    // Đúng thì trả về nút con phải ban đầu
	else                                       // Không thì gán lại nhánh trái/phải cho cha của
                                                                        ptr
	{
		if (ptr == ptr->_Parent->_Left)
			ptr->_Parent->_Left = right_child;
		else
			ptr->_Parent->_Right = right_child;
	}

	right_child->_Left = ptr;                  // Cập nhật lại nhánh trái nút con phải ban đầu
	ptr->_Parent = right_child;                // Cập nhật cha của ptr
}
```

```
void RBTree::rotateRight(Node *&ptr)                    // Hàm xoay phải tại ptr (tương tự xoay trái)
{
	Node *left_child = ptr->_Left;
	ptr->_Left = left_child->_Right;

	if (ptr->_Left != NULL)
		ptr->_Left->_Parent = ptr;

	left_child->_Parent = ptr->_Parent;

	if (ptr->_Parent == NULL)
		root = left_child;
	else
	{
		if (ptr == ptr->_Parent->_Left)
			ptr->_Parent->_Left = left_child;
		else
			ptr->_Parent->_Right = left_child;
	}

	left_child->_Right = ptr;
	ptr->_Parent = left_child;
}

void RBTree::fixInsert(Node *&ptr)                      // Cân bằng lại cây tại ptr khi thêm nút
{
	Node * parent = NULL;
	Node *grand = NULL;
	while (ptr != root && getColor(ptr) == RED && getColor(ptr->_Parent) == RED)
	{
		parent = ptr->_Parent;
		grand = parent->_Parent;
		if (parent == grand->_Left) // Xét nhánh trái của ông
		{
			Node *uncle = grand->_Right;
			// ptr đỏ, cha ptr đỏ, chú đỏ
				⇨  Đảo màu anh em, cha và ông
			if (getColor(uncle) == RED)
			{
				setColor(uncle, BLACK);
				setColor(parent, BLACK);
				setColor(grand, RED);
				ptr = grand;
			}
			else
			{
				// cha đỏ, chú đen, cháu nội (xét nhánh trái)
				⇨  Xoay tại cha; xoay tại ông; đảo màu cha, ông
				if (ptr == parent->_Right)
				{
					rotateLeft(parent);
					ptr = parent;
					parent = ptr->_Parent;
				}
				// cha đỏ, chú đen, cháu ngoại (xét nhánh trái)
				⇨  Xoay tại ông; đổi màu cha, ông
				rotateRight(grand);
				swap(parent->_Color, grand->_Color);
				ptr = parent;
			}
		}
		else // xét nhánh phải của ông
		{
			// ptr đỏ, cha ptr đỏ, chú đỏ
				⇨  Đảo màu anh em, cha và ông
```



```
                Node *uncle = grand->_Left;
                if (getColor(uncle) == RED)
                {
                        setColor(uncle, BLACK);
                        setColor(parent, BLACK);
                        setColor(grand, RED);
                        ptr = grand;
                }
                else
                {
                       // cha đỏ, chú đen, cháu nội (xét nhánh phải)
                       ⇨  Xoay tại cha; xoay tại ông; đảo màu cha, ông
                       if (ptr == parent->_Left)
                       {
                              rotateRight(parent);
                              ptr = parent;
                              parent = ptr->_Parent;
                       }
                       // cha đỏ, chú đen, cháu ngoại (xét nhánh phải)
                       ⇨  Xoay tại ông; đổi màu cha, ông
                       rotateLeft(grand);
                       swap(parent->_Color, grand->_Color);
                       ptr = parent;
                }
        }
    }
    setColor(root, BLACK); // cập nhật màu nút gốc là đen
}

void RBTree::fixDelete(Node *&ptr)                    // Cân bằng lại cây tại ptr khi xóa một nút
{
    if (ptr == NULL)                                  // Kiểm tra cây có rỗng không
        return;

    if (ptr == root)                                  // Kiểm tra nút cần xóa có phải gốc không
    {
        root = NULL;
        return;
    }
    if (getColor(ptr) == RED || getColor(ptr->_Left) == RED || getColor(ptr->_Right) == RED)
    {
        // Xóa nút chỉ có một nhánh con hay không có nhánh con
        Node *child = (ptr->_Left != NULL) ? ptr->_Left : ptr->_Right;
        if (ptr == ptr->_Parent->_Left) // ptr là nhánh trái của cha
        {
            ptr->_Parent->_Left = child;
            if (child != NULL)
                child->_Parent = ptr->_Parent;
            setColor(child, BLACK);
            delete(ptr);
        }
        else //ptr là nhánh phải của cha
        {
            ptr->_Parent->_Right = child;
            if (child != NULL)
                child->_Parent = ptr->_Parent;
            setColor(child, BLACK);
            delete(ptr);
        }
    }
    else
    {
        // Xóa nút có hai nhánh con
        Node *sibling = NULL;
        Node *parent = NULL;
        Node *node = ptr;
```

```
	setColor(ptr, DOUBLE_BLACK);
	while (ptr != root && getColor(ptr) == DOUBLE_BLACK)
	{
		parent = ptr->_Parent;
		if (ptr == parent->_Left) //ptr là nhánh trái của cha
		{
			sibling = parent->_Right; //sibling là nhánh phải của cha
			// Ptr đen kép, anh em đỏ
			⇨ Đổi màu cha và anh em, quay tại cha
			if (getColor(sibling) == RED)
			{
				setColor(sibling, BLACK);
				setColor(parent, BLACK);
				rotateLeft(parent);
			}
			else
			{
				// ptr đen kép, anh em đen, 2 cháu đen
				⇨ Đổi màu anh em sang đỏ, di chuyển đen kép lên (nếu là đỏ thì gán
				   đen, ngược lại là đen kép)
				if (getColor(sibling->_Left) == BLACK && getColor(sibling->_Right) ==
BLACK)
				{
					setColor(sibling, RED);
					if (getColor(parent) == RED)
						setColor(parent, BLACK);
					else
						setColor(parent, DOUBLE_BLACK);
					ptr = parent;
				}
				else
				{
					// ptr đen kép, anh em đen, cháu ngoại đen (xét nhánh trái)
					⇨ Đảo màu cháu nội, anh em; quay tại anh em; anh em nhận màu
					   của cha; đảo màu cha và cháu ngoại thành đen; quay tại cha
					if (getColor(sibling->_Right) == BLACK)
					{
						setColor(sibling->_Left, BLACK);
						setColor(sibling, RED);
						rotateRight(sibling);
						sibling = parent->_Right;
					}
					// ptr đen kép, anh am đen, cháu ngoại đỏ (xét nhánh trái)
					⇨ Anh em nhận màu của cha; cha và cháu ngoại nhận màu đen; quay
					   tại cha
					setColor(sibling, parent->_Color);
					setColor(parent, BLACK);
					setColor(sibling->_Right, BLACK);
					rotateLeft(parent);
					break;
				}
			}
		}
		else // ptr là nhánh phải của cha
		{
			sibling = parent->_Left; //sibling là nhánh trái của cha
			// Ptr đen kép, anh em đỏ
			⇨ Đổi màu cha và anh em, quay tại cha
			if (getColor(sibling) == RED)
			{
				setColor(sibling, BLACK);
				setColor(parent, RED);
				rotateRight(parent);
			}
			else
			{
```

```
                                // ptr đen kép, anh em đen, 2 cháu đen
                                ⇨ Đổi màu anh em sang đỏ, di chuyển đen kép lên (nếu là đỏ thì gán
                                    đen, ngược lại là đen kép)
                                if (getColor(sibling->_Left) == BLACK && getColor(sibling->_Right) ==
BLACK)
                                {
                                        setColor(sibling, RED);
                                        if (getColor(parent) == RED)
                                                setColor(parent, BLACK);
                                        else
                                                setColor(parent, DOUBLE_BLACK);
                                        ptr = parent;
                                }
                                else
                                {
                                         // ptr đen kép, anh em đen, cháu ngoại đen (xét nhánh phải)
                                        ⇨ Đảo màu cháu nội, anh em; quay tại anh em; anh em nhận màu
                                            của cha; đảo màu cha và cháu ngoại thành đen; quay tại cha
                                        if (getColor(sibling->_Left) == BLACK)
                                        {
                                                setColor(sibling->_Right, BLACK);
                                                setColor(sibling, RED);
                                                rotateLeft(sibling);
                                                sibling = parent->_Left;
                                        }
                                         // ptr đen kép, anh am đen, cháu ngoại đỏ (xét nhánh trái)
                                        ⇨ Anh em nhận màu của cha; cha và cháu ngoại nhận màu đen; quay
                                            tại cha
                                        setColor(sibling, parent->_Color);
                                        setColor(parent, BLACK);
                                        setColor(sibling->_Left, BLACK);
                                        rotateRight(parent);
                                        break;
                                }
                        }
                }
        }
        if (node == node->_Parent->_Left) // Cập nhật lại nhánh trái/phải của cha ptr
                node->_Parent->_Left = NULL;
        else
                node->_Parent->_Right = NULL;
        delete(node); // Xóa nút trỏ bởi ptr
        setColor(root, BLACK); // Cập nhật màu của gốc là đen
    }
}

void RBTree::inorderBST(Node *&ptr, ofstream& file)    //In cây ra file theo thứ tự giữa tại nút ptr
{
    if (ptr == NULL)
        return;

    inorderBST(ptr->_Left, file);
    file << ptr->_Word.m_Word << "  " << ptr->_Word.m_Meaning << endl;
    inorderBST(ptr->_Right, file);
}

int RBTree::getColor(Node *&p)                        // Lấy màu của nút trỏ bởi p
{
    if (p == NULL)
        return BLACK;
    return p->_Color;
}

void RBTree::setColor(Node *&p, int color)            // Cập nhật màu của nút trỏ bởi p
{
    if (p == NULL)
```

```
			return;
		p->_Color = color;
}

Node *RBTree::minValue(Node *&node)					// Lấy giá trị nhỏ nhất bên nhánh phải cha của node
{
		Node *ptr = node;
		while (ptr->_Left != NULL)
			ptr = ptr->_Left;
		return ptr;
}

Node *RBTree::maxValue(Node *&node)					// Lấy giá trị lớn nhất bên nhánh trái cha của node
{
		Node *ptr = node;
		while (ptr->_Right != NULL)
			ptr = ptr->_Right;
		return ptr;
}

Node * RBTree::insertBST(Node *&root, Node *&ptr)		// Thêm nút trỏ bởi ptr vào cây
{
		if (root == NULL) // Gốc là NULL
		{
			countWords++;
			return ptr;
		}
		if (ptr->_Word.m_Word < root->_Word.m_Word) // Xét nhánh trái của root
		{
			root->_Left = insertBST(root->_Left, ptr);
			root->_Left->_Parent = root;
		}
		else
		{
			if (ptr->_Word.m_Word > root->_Word.m_Word) // Xét nhánh phải của root
			{
				root->_Right = insertBST(root->_Right, ptr);
				root->_Right->_Parent = root;
			}
			else //Xét trường hợp từ nhiều nghĩa (thêm nghĩa mới vào)
			{
				if (root->_Word.m_Meaning.find(ptr->_Word.m_Meaning, 0) == string::npos)
				{
					root->_Word.m_Meaning += "\n";
					root->_Word.m_Meaning += ptr->_Word.m_Meaning;
				}
			}
		}
		return root;
}

Node *RBTree::deleteBST(Node *&root, WORD word)		// Xóa từ word khỏi cây
{
		if (root == NULL) //Nút là lá
			return root;
		if (word.m_Word < root->_Word.m_Word) // Xét nhánh trái của root
			return deleteBST(root->_Left, word);
		if (word.m_Word > root->_Word.m_Word) // Xét nhánh phải của root
			return deleteBST(root->_Right, word);
		if (root->_Left == NULL || root->_Right == NULL) //Nút chỉ có một nhánh
			return root;
		// Tìm nút thay thế (nút có 2 nhánh con), cập nhật lại và xóa
		Node *tmp = minValue(root->_Right);
		root->_Word.m_Word = tmp->_Word.m_Word;
		root->_Word.m_Meaning = tmp->_Word.m_Meaning;
		return deleteBST(root->_Right, tmp->_Word);
```

```
}

int RBTree::getBlackHeight(Node *ptr)                // Tính chiều cao đen
{
	int blackHeight = 0;
	while (ptr != NULL)
	{
		if (getColor(ptr) == BLACK)
			blackHeight++;
		ptr = ptr->_Left;
	}
	return blackHeight;
}

void RBTree::insert(const WORD &p)                   // Thêm từ p vào từ điển (thêm rồi cân bằng lại)
{
	Node *node = new Node(p);
	root = insertBST(root, node);
	fixInsert(node);
}

void RBTree::remove(WORD word)                       // Xóa từ p khỏi từ điển (xóa rồi cân bằng lại)
{
	Node *node = deleteBST(root, word);
	fixDelete(node);
}

void RBTree::inorder(string fileName)                // In từ điển ra file
{
	ofstream file(fileName.c_str());

	if (!file.is_open())
	{
		cout << "Error: Can't open file " << fileName << endl;
		return;
	}

	inorderBST(root, file);

	file.close();
}

Node *RBTree::search(string word)                    // Tìm kiếm từ word trong từ điển
{
	Node *node = root;
	while (node)
	{
		if (node->_Word.m_Word > word)
			node = node->_Left;
		else
		{
			if (node->_Word.m_Word < word)
				node = node->_Right;
			else
				return node;
		}
	}
	return NULL; // Nếu không có trong từ điển thì trả về NULL
}

void RBTree::clear(Node *&root)                      // Xóa từ điển
{
	if (root != NULL)
	{
		clear(root->_Left);
		clear(root->_Right);
```

```
			delete root;
		}
}

void RBTree::addUsage(Node *&ptr, string meaning)	// Thêm cách dùng
{
		if (ptr->_Word.m_Meaning.find(meaning, 0) == string::npos)
			ptr->_Word.m_Meaning = ptr->_Word.m_Meaning + "\n" + meaning;
}

void RBTree::edit(Node *&ptr, string meaning)		//Sửa nghĩa của từ trỏ bởi ptr
{
		ptr->_Word.m_Meaning = meaning;
}

string RBTree::getMeaning(Node *ptr)			// Lấy nghĩa của từ trỏ bởi pgr
{
		return ptr->_Word.m_Meaning;
}

int max(int a, int b)
{
		if (a > b)
			return a;
		else
			return b;
}

int RBTree::getHeight(Node *node)			// Tính chiều cao từ node đến lá
{
		if (node == NULL)
			return 0;
		return 1 + max(getHeight(node->_Left), getHeight(node->_Right));
}

int RBTree::getHeight()					// Tính chiều cao của từ điển
{
		return getHeight(root);
}

RBTree::~RBTree()					// Hủy từ điển
{
		clear(root);
}
```

* Nhận xét cây đỏ - đen: (n là tổng số nút của cây, h là chiều cao cây đỏ - đen)

- Thuận lợi: Tiết kiệm được chi phí
	+ Chiều cao cây O(logn)

$$n \geq 2^{\frac{h(\text{root})}{2}} - 1 \leftrightarrow \log_2(n+1) \geq \frac{h(\text{root})}{2} \leftrightarrow h(\text{root}) \leq 2\log_2(n+1).$$

(nguồn Wiki)

	+ Tìm kiếm với độ phức tạp tối đa O(logn)
	+ Load được từ điển ra file là O(nlogn)
	+ Thêm và xóa một từ là O(logn)
	+ Lưu từ điển ra file là O(logn)
	+ Số lượng bộ nhớ sử dụng (n)

- Khó khăn: Cài đặt phức tạp hơn. Chiều cao không tối ưu nên tìm kiếm lâu hơn AVL, làm ảnh hưởng đến load file, thêm và xóa. Khi thêm hay xóa một nút khỏi cây vừa phải đảm bảo được tính

tìm kiếm vừa phải đảm bảo tính cân bằng đen nên luôn phải kiểm tra và cân bằng lại (nhưng ít lần hơn so với AVL).

* So sánh với cây AVL:

Cây AVL	**Cây đỏ - đen**
Số lượng sử dụng bộ nhớ như nhau O(n)	
Chênh lệch chiều cao không lớn hơn 1 nên độ cao tối ưu (log_2n)	Có độ cân bằng tương đối (chỉ đảm bảo cân bằng đen) nên độ cao luôn nhỏ hơn $h <= 2log_2(n + 1)$
Do đảm bảo độ cân bằng tốt hơn nên cần nhiều thời gian hơn để cân bằng lại cây khi thêm hoặc xóa nút với dữ liệu nhỏ	Tốn ít thời gian để cân bằng lại khi thêm hay xóa nút với dữ liệu nhỏ
Tìm kiếm, lưu từ điển nhanh hơn do đảm bảo chiều cao tốt O(log_2n)	Tìm kiếm, lưu từ điển lâu hơn trên cây AVL O(h)
Load từ điển chậm hơn do cân bằng nhiều lần với dữ liệu nhỏ	Load từ điển nhanh hơn với dữ liệu nhỏ