BÁO CÁO CHƯƠNG TRÌNH CÀI ĐẶT
Bảng băm – Hash Table

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

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

class HashNode                               // Cấu trúc của một nút Hash (từ và con trỏ trỏ đến phần tử kế tiếp)
{
private:
        WORD _Word;
        HashNode * _Next;
}

class HashTable                              // Cấu trúc của một bảng băm (mỗi ô của bảng là một DSLK đơn)
{
private:
        HashNode ** _HTable;
}
```

- **Cài đặt chương trình:**

- **Chương trình trong lớp HashNode**

```
HashNode::HashNode();                        // Khởi tạo mặc định, nút mới có next trỏ đến NULL

HashNode::HashNode(WORD word)                // Khởi tạo nút mới từ một từ đã cho
{
        _Word.m_Word = word.m_Word;
        _Word.m_Meaning = word.m_Meaning;
        _Next = NULL;
}

string HashNode::getWord();                  // Hàm trả về từ của nút đang được trỏ đến
string HashNode::getMeaning();               // Hàm trả về nghĩa của nút đang được trỏ đến
void HashNode::setMeaning(string meaning);   // Hàm gán nghĩa cho nút đang được trỏ đến
void HashNode::changeMeaning(string meaning); // Hàm thay đổi nghĩa của nút đang được trỏ đến
HashNode * HashNode::getNext();              // Hàm trả về nút next của nút đang được trỏ đến
void HashNode::setNext(HashNode * p);        // Hàm gán lại nút next cho nút đang được trỏ đến
```

- **Chương trình trong lớp HashTable:**

```
HashTable::HashTable()                       // Khởi tạo bảng băm ban đầu đều bằng NULL, với kích thước bảng
                                               là TABLE_SIZE = 36433 (số nguyên tố lớn hơn số từ của từ điển)
{
        _HTable = new HashNode *[TABLE_SIZE];
        for (int i = 0; i < TABLE_SIZE; i++)
                _HTable[i] = NULL;
}
```

```
HashTable::~HashTable();                    // Xóa toàn bộ bảng băm
bool HashTable::isEmpty();                  // Kiểm tra xem bảng băm có rỗng không

void HashTable::Delete(string word)         // Xóa một từ trong bảng băm
{
        // Nếu bảng băm trống thì không xử lí
        if (isEmpty())
                return;

        // Tính khóa Hash cho từ để giúp tìm từ nhanh chóng
        int key = HashFunc(word);

        // Nếu ô tại khóa đó rỗng thì không xử lí
        if (_HTable[key] == NULL)
                return;
        else // Ngược lại kiểm tra nút đầu của DSLK có phải từ cần xóa không (xóa đầu của DLSK)
        {
                if (_HTable[key]->getWord() == word)
                {
                        HashNode * tmp = _HTable[key];
                        _HTable[key] = _HTable[key]->getNext();
                        delete tmp;
                }
        }

        // Ngược lại, duyệt từng nút trong DSLK của ô để tìm nút xóa (trường hợp bảng băm không lí tưởng) và thực hiện lệnh xóa giữa và cuối DSLK
        HashNode *node = _HTable[key];
        HashNode *tmp = _HTable[key];
        while (tmp != NULL)
        {
                if (tmp->getWord() == word)
                        break;
                node = tmp;
                tmp = tmp->getNext();
        }
        if (tmp != NULL)
        {
                node->setNext(tmp->getNext());
                delete tmp;
        }
}

HashNode * HashTable::Search(string word)   // Tìm kiếm từ trong bảng băm
{
        // Nếu bảng băm rỗng thì không xử lí
        if (isEmpty())
                return NULL;
        // Tính khóa Hash cho từ cần tìm
        int key = HashFunc(word);
        // Kiểm tra xem ô tại khóa đó có rỗng không. Tìm được thì trả về nút đó
        HashNode *tmp = _HTable[key];
```

```
	while (tmp != NULL)
	{
		if (tmp->getWord() == word)
			return tmp;
		tmp = tmp->getNext();
	}
	return NULL; // Không tìm thấy thì trả về NULL
}

void QS(WORD *a, int l, int r);			// Hàm sort để in các từ ra file theo thứ tự từ điển
void HashTable::Display(string fileName);	// Hàm in bảng ra file, chép bảng vào mảng một chiều sau đó sắp
						xếp mảng vừa tạo

int HashTable::HashFunc(string word)		// Hàm tính khóa Hash cho một từ, tính theo cơ số
					HASH_WEIGHT = 31, sau đó lấy mod cho kích thước bảng để tìm ô phù hợp
{
	unsigned long key = 0;
	for (int i = 0; i < word.length(); i++)
	{
		int temp = 0;
		if ('A' <= word[i] && word[i] <= 'Z')
			temp = word[i] - 64;
		else
			if ('a' <= word[i] && word[i] <= 'z')
				temp = word[i] - 96;

		key = temp + HASH_WEIGHT * key;
	}
	return key % TABLE_SIZE;
}

void HashTable::Insert(const WORD &word) // Hàm thêm một từ vào bảng băm
{
	// Tính khóa Hash cho từ mới
	int key = HashFunc(word.m_Word);
	// Tạo nút mới từ từ đó
	HashNode * k = new HashNode(word);
	// Thêm từ vào ô có khóa vừa tính được
	+ DSLK rỗng
	if (_HTable[key] == NULL)
	{
		_HTable[key] = k;
		return;
	}
	+ DSLK khác rỗng (thêm vào cuối danh sách, vừa kiểm tra xem từ đó đã có chưa)
	HashNode * tmp = _HTable[key];
	while (tmp != NULL)
	{
		if (tmp->getWord() == word.m_Word)
		{
			_HTable[key]->setMeaning(word.m_Meaning);
			return;
		}
```

```
            tmp = tmp->getNext();
        }
        if (tmp == NULL)
            tmp = k;
}

int HashTable::MaxCollision();          // Hàm tính mức đụng độ cao nhất (Kích thước lớn nhất của DSLK)
```

<> Nhận xét bảng băm: Với số từ có được từ file (từ nhiều nghĩa, từ trùng tính là 1) là 36396 từ, thì tìm được số nguyên tố vừa lớn hơn số từ là 36433
=> Chọn kích thước bảng băm là TABLE_SIZE = 36433
* Thuận lợi:
- Hạn chế số lần so sánh => giảm thiểu thời gian truy xuất. Độ phức tạp thường là O(1) và không phụ thuộc vào kích thước của bảng băm
- Dung hòa giữa thời gian truy xuất và dung lượng bộ nhớ: Nếu không có giới hạn bộ nhớ thì có thể xây dựng bảng băm với mội khóa ứng với một địa chỉ với mong muốn thời gian truy xuất tức thời; Nếu dung lượng bộ nhớ có giới hạn thì tổ chức một số khóa có cùng địa chỉ, lúc này thời gian truy xuất có thể bị suy giảm đôi chút

- Nếu băm đều n phần tử vào M bucket, lúc này trung bình mỗi Bucket sẽ có n/M phần tử: (n >= M)
+ Tìm kiếm tuyến tính trên bucket nên thời gian tìm kiếm lúc này có bậc O(n/M)
+ Thêm mất độ phức tạp O(n/M) do duyệt tuyến tính trên bucket để kiểm tra xem có từ trùng không
+ Xóa cũng mất độ phức tạp O(n/M) để tìm được khóa cần xóa.
+ Thời gian Load file là $O(n^2/M)$, do đọc n từ, mỗi từ lại phải duyệt qua bucket với n/M nút để kiểm tra có từ đó hay chưa.
+ Thời gian lưu từ điển là O(n), do duyệt qua M bucket, mỗi bucket lại phải duyệt qua n/M nút.

- Nếu chọn được hàm băm phù hợp thì mỗi ô khóa sẽ có DSLK với độ dài là <= 1 nút (không xảy ra hiện tượng đụng độ), lúc đó: (M >= n)
+ Tìm kiếm sẽ có độ phức tạp O(1)
+ Thêm một phần tử mất độ phức tạp O(1)
+ Xóa một phần tử mất độ phức tạp O(1)
+ Thời gian load file là O(n)
+ Thời gian lưu file là O(M)

* Khó khăn:

+ Lãng phí bộ nhớ (Với n từ nhưng ta có thể cần đến bảng có kích thước M (M > n) để tạo ra trường hợp lí tưởng cho bảng băm (mỗi ô khóa có duy nhiều nhất 1 nút))

+ Để tìm được bảng băm tốt đôi khi rất phức tạp

+ Phải biết trước được số lượng dữ liệu mới xây dựng được bảng băm phù hợp, khác với các cấu trúc cây không cần biết trước

+ Bảng băm không lưu theo thứ tự nào cả, nên muốn in ra file theo thứ tự từ điển phải sort lại và độ phức tạp phụ thuộc vào thuật toán sắp xếp

<> So sánh với cây AVL, cây đỏ-đen: (Xét trường hợp lí tưởng)

Dữ liệu so sánh	**Cây AVL**	**Cây đỏ-đen**	**Bảng băm**
Bộ nhớ sử dụng	O(n)	O(n)	O(M) (M >= n)
Thời gian tìm kiếm trung bình	O(logn)	O(logn)	O(1)
Thời gian chèn, xóa trung bình	O(logn)	O(logn)	O(1)
Thời gian lưu từ điển ra file	O(logn)	O(logn)	O(M) (M >= n)
Thời gian load từ điển	O(nlogn)	O(nlogn)	O(n)