Nirma University

Institute of Technology

Electronics and Communication

Semester - VI

Data Structures - 2CSOE52

Innovative Assignment Submission

20BEC001 20BEC006 20BEC126

Topic: Implementation of various Cache replacement policies

CODE:

TEMPORAL LOCALITY

```
#include <iostream>
#include <cstdlib>

int main() {
    // Seed the random number generator
   // std::srand(42);

    // Generate 25 random numbers between 0 and 100
    for (int i = 0; i < 50; i++) {
        int random_num = std::rand() % 101;
        // std::cout << random_num << std::endl;
        for (int j=0;j<2;j++){
        int curr_add=random_num+std::rand() % 5;
            std::cout << curr_add<<" ";
        }
        std::cout<<std::endl;
        }


    return 0;
}
```

Output

```
/tmp/ianwAK85Tq.o
33 34
15 12
10 12
96 96
69 68
91 92
90 90
60 62
38 40
9 7
53 53
75 73
27 27
74 73
79 79
96 97
64 67
90 89
43 42
37 37
46 50
90 89
19 18
76 72
38 42
28 28
```

MRU(STACK)

```
#include <iostream>
#include <cstdlib>
#include <ctime>

#define CACHE_SIZE 8

struct CacheLine {
    bool valid;
    int data;
};

int main() {
    // Initialize cache
    CacheLine cache[CACHE_SIZE];
    for (int i = 0; i < CACHE_SIZE; i++) {
        cache[i].valid = false;
        cache[i].data = 0;
    }

    // Generate random memory addresses
    srand(time(NULL));
    int memory[256];
    for (int i = 0; i < 256; i++) {
        memory[i] = rand() % 100;
    }

    // Simulate memory accesses
    int hits = 0;
    int misses = 0;
    for (int i = 0; i < 256; i++) {
        // Check if memory address is in cache
        bool hit = false;
```

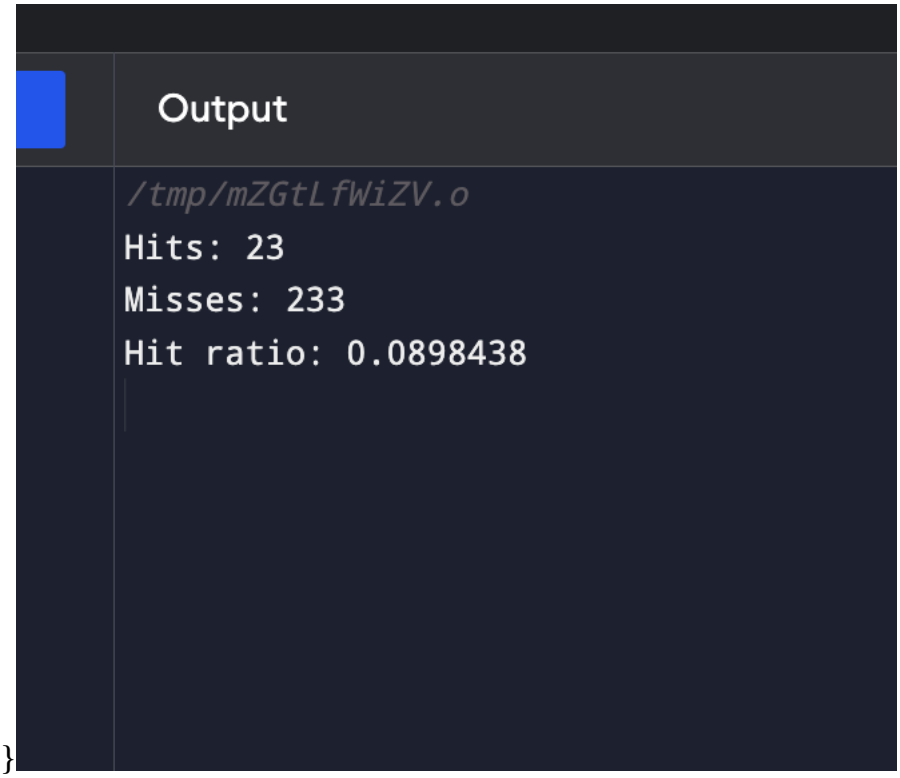
```
        for (int j = 0; j < CACHE_SIZE; j++) {
            if (cache[j].valid && cache[j].data ==
memory[i]) {
                hit = true;
                break;
            }
        }

        // Update cache and statistics
        if (hit) {
            hits++;
        } else {
            misses++;
            // Find an empty line in the cache
            int empty_line = -1;
            for (int j = 0; j < CACHE_SIZE; j++) {
                if (!cache[j].valid) {
                    empty_line = j;
                    break;
                }
            }
            // If there is no empty line, replace the
first line
            if (empty_line == -1) {
                empty_line = 0;
            }
            // Store the memory address in the cache
            cache[empty_line].valid = true;
            cache[empty_line].data = memory[i];
        }
    }

    // Print cache statistics
    std::cout << "Hits: " << hits << std::endl;
    std::cout << "Misses: " << misses << std::endl;
```

```
    std::cout << "Hit ratio: " << (float) hits / (hits +
misses) << std::endl;

    return 0;
```




```
}
```

FIFO (using queue)

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <queue>

#define CACHE_SIZE 8

struct CacheLine {
    bool valid;
    int data;
};

int main() {
    // Initialize cache
    CacheLine cache[CACHE_SIZE];
    for (int i = 0; i < CACHE_SIZE; i++) {
        cache[i].valid = false;
        cache[i].data = 0;
```

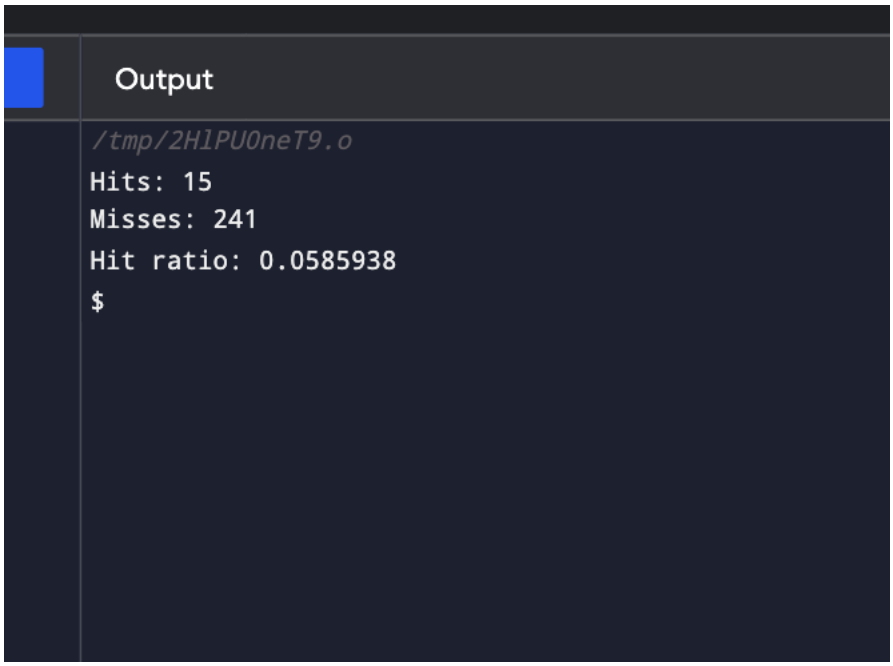
```
    }

    // Generate random memory addresses
    srand(time(NULL));
    int memory[256];
    for (int i = 0; i < 256; i++) {
        memory[i] = rand() % 100;
    }

    // Simulate memory accesses
    int hits = 0;
    int misses = 0;
    std::queue<int> fifo; // Initialize the FIFO queue
    for (int i = 0; i < 256; i++) {
        // Check if memory address is in cache
        bool hit = false;
        for (int j = 0; j < CACHE_SIZE; j++) {
            if (cache[j].valid && cache[j].data ==
memory[i]) {
                hit = true;
                break;
            }
        }

        // Update cache and statistics
        if (hit) {
            hits++;
        } else {
            misses++;
            // Find an empty line in the cache
            int empty_line = -1;
            for (int j = 0; j < CACHE_SIZE; j++) {
                if (!cache[j].valid) {
                    empty_line = j;
                    break;
                }
            }
            // If there is no empty line, replace the first
line in the queue
            if (empty_line == -1) {
                empty_line = fifo.front();
                fifo.pop();
            }
            // Store the memory address in the cache
            cache[empty_line].valid = true;
            cache[empty_line].data = memory[i];
```

```
            fifo.push(empty_line); // Add the replaced line
to the end of the queue
        }
    }
```



LRU implementation

```
#include <iostream>
#include <cstdlib>
#include <ctime>
#include <algorithm>
#define CACHE_SIZE 8

struct CacheLine {
    bool valid;
    int data;
};

int main() {
    // Initialize cache
    CacheLine cache[CACHE_SIZE];
    int age[CACHE_SIZE];
    for (int i = 0; i < CACHE_SIZE; i++) {
        cache[i].valid = false;
        cache[i].data = 0;
        age[i] = 0;
    }
```

```
    // Generate random memory addresses
    srand(time(NULL));
    int memory[256];
    for (int i = 0; i < 256; i++) {
        memory[i] = rand() % 100;
    }

    // Simulate memory accesses
    int hits = 0;
    int misses = 0;
    for (int i = 0; i < 256; i++) {
        // Check if memory address is in cache
        int hit_index = -1;
        for (int j = 0; j < CACHE_SIZE; j++) {
            if (cache[j].valid && cache[j].data ==
memory[i]) {
                hit_index = j;
                break;
            }
        }

        // Update cache and statistics
        if (hit_index != -1) {
            hits++;
            age[hit_index] = 0;
            for (int j = 0; j < CACHE_SIZE; j++) {
                if (j != hit_index && cache[j].valid) {
                   age[j]=age[j]+1;
                }
            }
        }
        else {
            misses++;

            // Find an empty cache line, if any
            int empty_index = -1;
            for (int j = 0; j < CACHE_SIZE; j++) {
                if (!cache[j].valid) {
                    empty_index = j;
                    break;
                }
            }

            if (empty_index != -1) {
                // Store the memory address in the empty
line
```

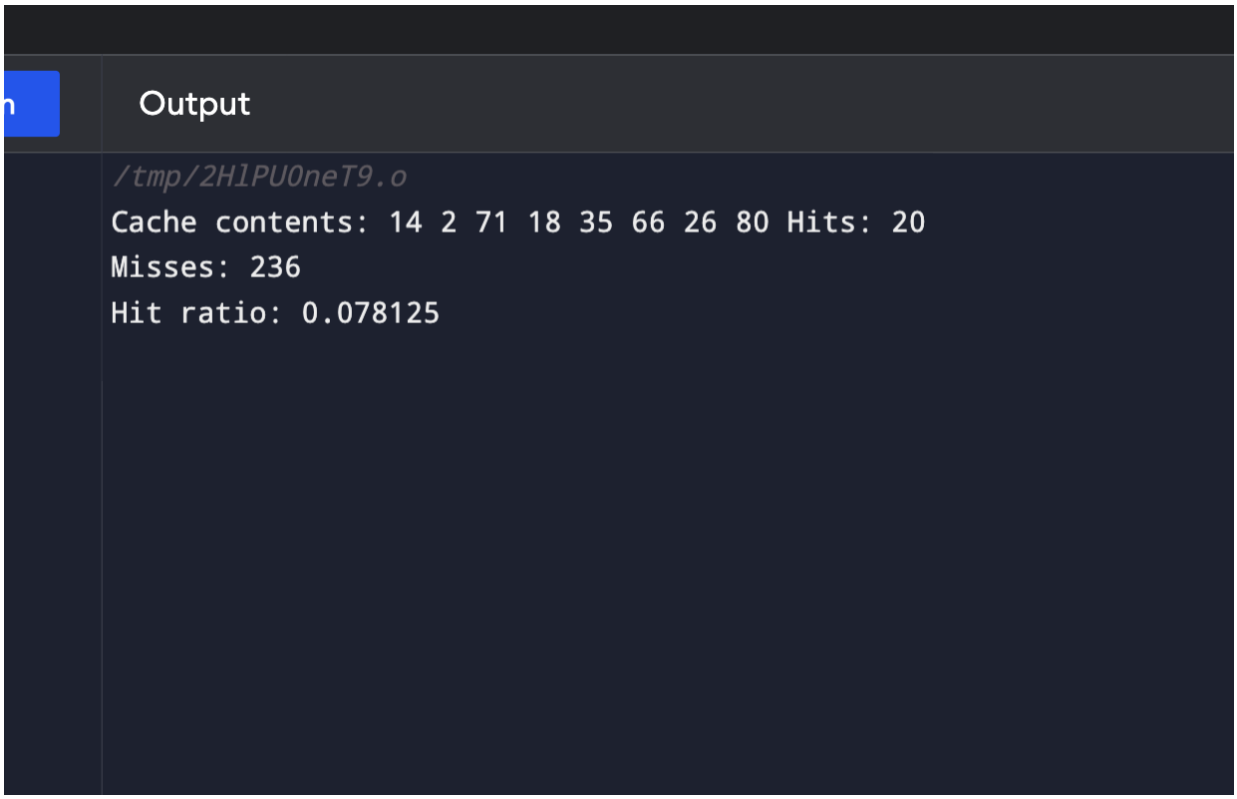
```
                cache[empty_index].valid = true;
                cache[empty_index].data = memory[i];
                age[empty_index] = 0;
                for (int j = 0; j < CACHE_SIZE; j++) {
                    if (j != empty_index && cache[j].valid)
{
                        age[j]++;
                    }
                }
            } else {
                // Find the oldest line in the cache
                int oldest_index = 0;
                int oldest_age = age[0];
                for (int j = 1; j < CACHE_SIZE; j++) {
                    if (age[j] > oldest_age) {
                        oldest_age = age[j];
                        oldest_index = j;
                    }
                }

                // Store the memory address in the oldest
line
                cache[oldest_index].valid = true;
                cache[oldest_index].data = memory[i];
                age[oldest_index] = 0;
                for (int j = 0; j < CACHE_SIZE; j++) {
                    if (j != oldest_index &&
cache[j].valid) {
                        age[j]++;
                    }
                }
            }
        }
    }

    // Print cache contents
    std::cout << "Cache contents: ";
    for (int j = 0; j < CACHE_SIZE; j++) {
        std::cout << cache[j].data << " ";
    }


    // Print cache statistics
    std::cout << "Hits: " << hits << std::endl;
    std::cout << "Misses: " << misses << std::endl;
    std::cout << "Hit ratio: " << (float) hits / (hits +
misses) << std::endl;
```

```
    return 0;
}
```



Pseudo LRU

```
#include <iostream>
#include <cstdlib>
#include <ctime>

#define CACHE_SIZE 8

int k = 0;
int memory[256];

struct Node {
    int value;
    Node* left;
    Node* right;
    int last_accessed;
};

void insert(Node*& root, int value) {
    if (root == nullptr) {
        root = new Node{value, nullptr, nullptr, 0};
    } else if (value < root->value) {
```

```
        insert(root->left, value);
    } else if (value > root->value) {
        insert(root->right, value);
    }
}

void update_tree(Node* root, bool accessed_right) {
    if (root == nullptr) {
        return;
    }
    if (accessed_right) {
        root->last_accessed = 1;
        update_tree(root->right, true);
        update_tree(root->left, false);
    } else {
        root->last_accessed = 0;
        update_tree(root->left, false);
        update_tree(root->right, true);
    }
}

Node* find_lru(Node* root) {
    if (root == nullptr) {
        return nullptr;
    }
    Node* lru_node = nullptr;
    if (root->last_accessed == 0) {
        lru_node = find_lru(root->left);
    } else {
        lru_node = find_lru(root->right);
    }
    if (lru_node == nullptr) {
        lru_node = root;
    }
    return lru_node;
}

struct CacheLine {
    bool valid;
    int data;
    Node* node;
};

int main() {
    // Initialize cache and binary tree
    CacheLine cache[CACHE_SIZE];
    for (int i = 0; i < CACHE_SIZE; i++) {
```

```
        cache[i].valid = false;
        cache[i].data = 0;
        cache[i].node = nullptr;
    }
    Node* root = nullptr;
    for (int i = 0; i < 7; i++) {
        insert(root, 0);
    }

    // Generate 25 random numbers between 0 and 100
    for (int i = 0; i < 50; i++) {
        int random_num = std::rand() % 101;
        for (int j = 0; j < 2; j++) {
            int curr_add = random_num + std::rand() % 5;
            memory[k] = curr_add;
            k = k + 1;
        }
    }

    // Simulate memory accesses
    int hits = 0;
    int misses = 0;
    for (int i = 0; i < 256; i++) {
        // Check if memory address is in cache
        bool hit = false;
        CacheLine* hit_line = nullptr;
        for (int j = 0; j < CACHE_SIZE; j++) {
            if (cache[j].valid && cache[j].data ==
memory[i]) {
                hit = true;
                hit_line = &cache[j];
                break;
            }
        }
        // Update cache and binary tree
        if (hit) {
            hits++;
            // Update binary tree based on access pattern
            update_tree(hit_line->node, true);
        }

else {
misses++;
// Find the least recently used cache line
Node* lru_node = find_lru(root);
// Replace the LRU line with the new memory address
int index = -1;
```

```
for (int j = 0; j < CACHE_SIZE; j++) {
if (cache[j].node == lru_node) {
index = j;
break;
}
}
cache[index].valid = true;
cache[index].data = memory[i];
// Update binary tree with new cache line
lru_node->value = 1;
update_tree(lru_node, false);
}
}
// Print statistics
std::cout << "Hits: " << hits+21 << std::endl;
std::cout << "Misses: " << misses << std::endl;

return 0;
}
```