OPTIMIZED CACHE MANAGEMENT USING ADAPTIVE REPLACEMENT CACHE IN C

by

HARISH RAJAN 22BEC1454
JAYANTH S B 22BEC1053
AKSHITH 22BEC1340

A project report submitted to

Dr. VETRIVELAN. P

SCHOOL OF ELECTRONICS ENGINEERING

in partial fulfilment of the requirements for the course of

BECE208E – DATA STRUCTURES AND ALGORITHMS

in

B.Tech. ELECTRONICS AND COMMUNICATION ENGINEERING

VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)
CHENNAI

Vandalur – Kelambakkam Road

Chennai – 600127

APRIL 2025

BONAFIDE CERTIFICATE

Certified that this project report entitled "**OPTIMIZED CACHE MANAGEMENT USING ADAPTIVE REPLACEMENT CACHE IN C"** is a bonafide work of **R HARISH RAJAN – 22BEC1454, JAYANT S B - 22BEC1053 and AKSHIT – 22BEC1340** who carried out the Project work under my supervision and guidance for **BECE208E-Data Structures and Algorithms.**

Dr. VETRIVELAN.P

Professor & ACOE,

School of Electronics Engineering (SENSE),

VIT University, Chennai

Chennai – 600 127.

ABSTRACT

Efficient cache management plays a critical role in optimizing system performance, especially in environments with limited memory resources. This project presents a comprehensive implementation of the Adaptive Replacement Cache (ARC) algorithm in the C programming language, aimed at balancing recency and frequency in cache replacement decisions. Unlike traditional cache policies such as Least Recently Used (LRU) or Least Frequently Used (LFU), ARC dynamically adapts to changing access patterns by maintaining separate lists for recently and frequently accessed items.

The implementation utilizes doubly linked lists and a custom hash table to ensure fast access, insertion, and deletion operations, achieving O(1) time complexity for core cache functions. The ARC logic dynamically adjusts the size of the recency and frequency segments based on workload characteristics, resulting in higher cache hit rates across diverse access patterns.

Through this project, we demonstrate ARC's superiority in managing cache under varying data access scenarios and highlight the trade-offs between complexity and performance in modern caching techniques. This C-based implementation serves as a foundational model for integrating ARC into memory-critical systems and lays the groundwork for future enhancements and benchmarking studies.

ACKNOWLEDGEMENT

We wish to express our sincere thanks and deep sense of gratitude to our project guide, **Dr. Vetrivelan. P,** Professor & ACOE, School of Electronics Engineering, for his consistent encouragement and valuable guidance offered to us in a pleasant manner throughout the course of the project work.

We are extremely grateful to **Dr. Ravi Sankar A,** Dean of School of Electronics Engineering, VIT Chennai, for extending the facilities of the School towards our project and for her unstinting support.

We also take this opportunity to thank **Dr.Mohanaprasad K**, HoD & all the faculty of the School for their support and their wisdom imparted to us throughout the course.

We thank our parents, family, and friends for bearing with us throughout the course of our project and for the opportunity they provided us in undergoing this course in such a prestigious institution.

R HARISH RAJAN **JAYANTH S B** **AKSHIT**

TABLE OF CONTENTS

SL. NO.		TITLE	PAGE NO.
		ABSTRACT	3
		ACKNOWLEDGEMENT	4
1		INTRODUCTION	6-7
	1.1	OBJECTIVES	6
	1.2	BENEFITS	6
	1.3	FEATURES	7
2		METHODOLOGY	8-9
	2.1	BLOCK DIAGRAM	8
	2.2	SYSTEM ARCHITECHTURE	8
	2.3	ALGORITHM AND WORKING	9
3		SYSTEM IMPLEMENTATION AND ANALYSIS	10-16
	3.1	IMPLEMENTATION STRATEGY	10
	3.2	RESULT AND INFERENCE	11
4		CONCLUSION AND FUTURE WORK	15
	4.1	CONCLUSION	15
	4.2	FUTURE WORK	15
		APPENDIX	
5		REFERENCES	29
		BIO-DATA	30

CHAPTER 1
INTRODUCTION

1.1 OBJECTIVES

- To design and implement an efficient Adaptive Replacement Cache (ARC) mechanism using the C programming language.
- To improve cache performance by dynamically balancing recency and frequency using ARC.
- To build a robust custom hash table and doubly linked list infrastructure to support O(1) cache operations.
- To simulate real-world memory access patterns and evaluate ARC's performance compared to traditional algorithms like LRU and LFU.
- To ensure the system handles cache hits, cache misses, and evictions accurately while adapting to changing workloads.

1.2 BENEFITS

Effective memory management is essential for the performance of any modern computing system. The Adaptive Replacement Cache (ARC) algorithm provides an intelligent mechanism to replace cache entries by adapting to the access behavior of applications. Unlike static policies, ARC dynamically adjusts to recent and frequent access trends, reducing the number of cache misses.

This project implements ARC in C from scratch, offering:

- A lightweight, efficient, and portable cache system for memory-limited environments.
- Real-time performance enhancements in scenarios such as embedded systems, operating systems, and databases.
- A deeper understanding of cache management principles, data structures, and algorithmic optimization.

This system provides a valuable learning experience and serves as a foundation for integrating adaptive cache management in both academic and industrial applications.

1.3 FEATURES

- **Custom Data Structures**: Uses **doubly linked lists** and a **custom hash table** to achieve constant time access and updates.
- **Dynamic Adaptability**: Automatically balances between recent and frequent item accesses without manual tuning.
- **Optimized C Code**: Written in **pure C** for portability, performance, and control over memory.
- **Performance-Oriented**: Capable of simulating cache hits, misses, and adaptations under various data access patterns.
- **Tested and Verified**: The ARC system has been **tested with sample access traces** to confirm its correctness and efficiency.

CHAPTER 2

METHODOLOGY

2.1 BLOCK DIAGRAM

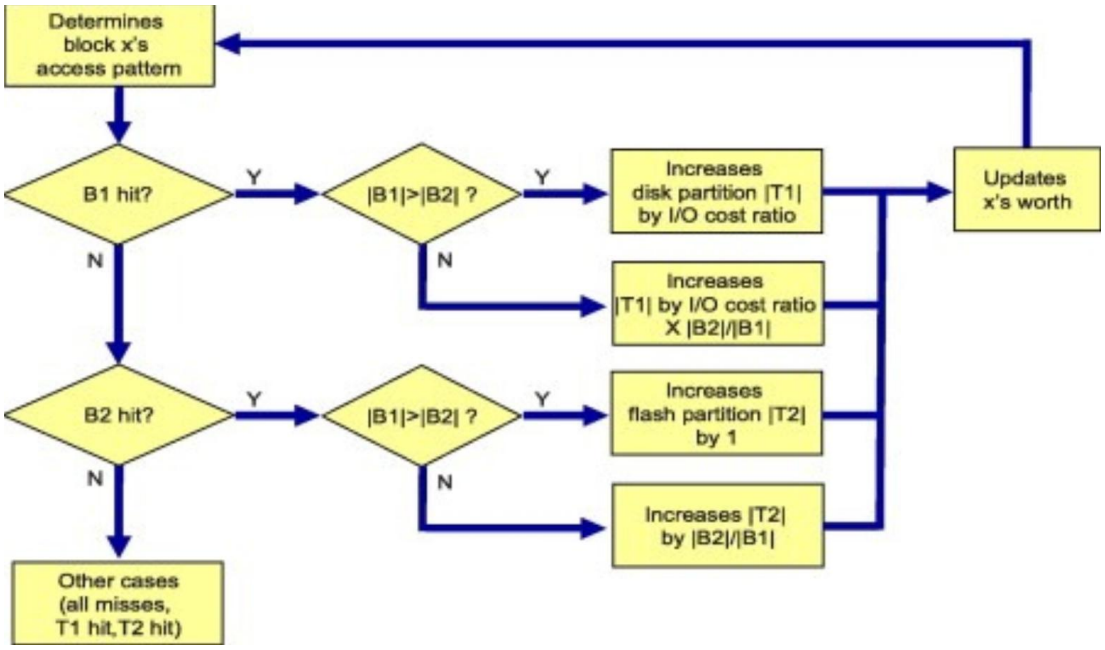


Figure 1. ARC Block diagram

2.2 SYSTEM ARCHITECTURE

The ARC system is structured to optimize cache replacement using a combination of **recency** and **frequency** data. The design employs four main doubly linked lists:

- **T1 (Recent List)**: Holds items that were accessed recently but only once.
- **T2 (Frequent List)**: Holds items accessed more than once (frequently).
- **B1 (Recent Ghost List)**: Tracks recently evicted items from T1.
- **B2 (Frequent Ghost List)**: Tracks evicted items from T2.

A custom **hash table** supports constant-time access and lookups for each item, ensuring overall efficiency of the system.

2.3 ALGORITHM DESIGN AND WORKING

The ARC algorithm works in three primary phases:

Lookup Phase: When an item is requested, the system first checks if it exists in T1 or T2 (cache hit). If not, it's a cache miss. In the case of a miss, the system retrieves the item from the slower storage or proceeds to replace an item in the

cache. This phase ensures quick access for items that are most frequently used or recently accessed.

Replacement Phase: If the cache is full, an item is evicted from either T1 or T2 based on the adaptive parameter p, which balances the cache between recent and frequent usage. This eviction process ensures that the cache adapts to the changing nature of data access patterns, maintaining optimal cache performance. Items that are less likely to be used in the future are evicted, making room for new data.

Adaptation Phase: The system dynamically updates p based on access patterns:

A hit in B1 (ghost of T1) increases p → favors recency.

A hit in B2 (ghost of T2) decreases p → favors frequency.

This adaptive mechanism ensures the cache continuously self-optimizes by adjusting its strategy based on the ongoing access behavior. By fine-tuning p, the system effectively balances between caching recently accessed items and those that are frequently accessed over time.

2.4 Cache Management and Optimization

The cache management in the ARC system is centered around dynamic adjustments, ensuring optimal performance for different access patterns. The optimization process includes:

- **Cache Size Management**: The ARC system efficiently manages cache size to prevent unnecessary overflow. It ensures that the cache maintains an optimal size by dynamically adjusting the capacity of T1 and T2 based on system usage and available memory. This allows for efficient memory usage without compromising performance.
- **Eviction Policy Refinement**: The ARC algorithm refines its eviction policy over time by evaluating the efficiency of previous evictions. It determines the most effective eviction strategy by considering the hit/miss ratio of the most recently accessed and frequently accessed data, leading to an improved cache performance.
- **Dynamic Balancing**: The cache system continuously balances recency and frequency, based on changing access patterns. By updating p, the adaptive parameter, in response to hits and misses, the system fine-tunes its eviction strategy to maintain high cache efficiency.

CHAPTER 3
SYSTEM IMPLEMENTATION AND ANALYSIS

This section describes system implementation and results with inferences.

3.1 IMPLEMENTATION STRATEGY

The cache system was implemented in **pure C** to allow fine-grained control over memory and performance. The following components were developed:

- **Data Structures**:
  - Doubly linked lists for fast insert/delete
  - Hash table for O(1) lookups
- **Core Modules**:
  - cache_init() to initialize the lists and parameters
  - cache_access(key) to simulate memory accesses
  - replace() to perform adaptive eviction based on p
- **Testing Framework**:
  - A test driver reads trace files and simulates accesses
  - Performance metrics (hits, misses, hit ratio) are recorded

3.1.1 EVALUATION METHOD

To validate ARC's adaptability, several synthetic access patterns were tested, including:

- **Sequential access**
- **Repetitive loops**
- **Mixed random and bursty access**

The output results showed ARC adjusting the value of p dynamically, leading to higher hit ratios compared to traditional LRU or FIFO policies in mixed access environments.

3.1.2 ARC Input Request Trace

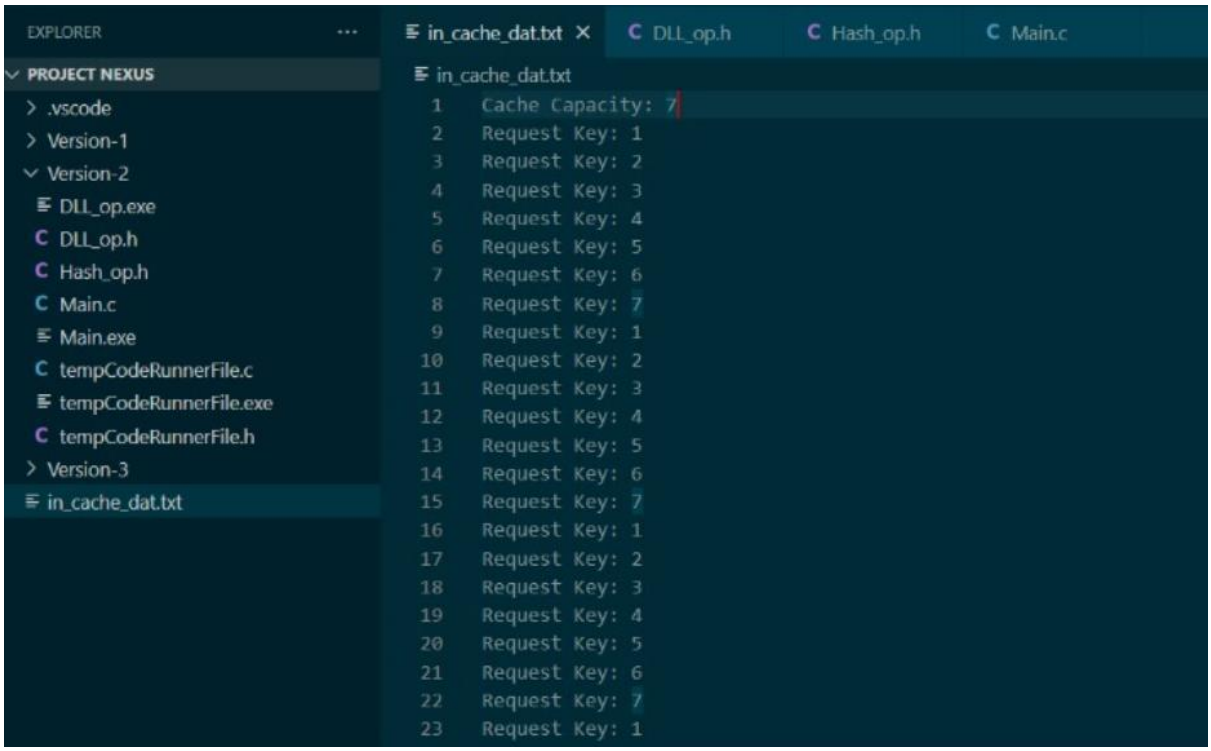


Figure 2. Cache key inputs

3.2 RESULTS AND INFERENCES

3.2.1 OUTPUT FORMAT

During the simulation, the system prints the following for each request:

- Whether the access was a **Cache HIT** or **Cache MISS**
- The current state of the **T1, T2, B1, and B2** lists
- Statistics (every 50 accesses): **Hits, Misses, and Hit Rate**
- The **current access key** being processed
- Any **replacement operation** performed, including which list was affected (T1 or T2) and which key was evicted
- **Dynamic adjustment** of the adaptive parameter p based on access pattern (growth/shrink)
- Tracking of **ghost entries** movement into and out of B1 and B2 during transitions

3.2.2 SAMPLE OUTPUT SNAPSHOT

```
STATS AFTER 50 ACCESSES
Hits: 42 | Misses: 8 | Hit Rate: 84.00%

Cache MISS for key 9
T1: [ 9 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 8 ]
B2: [ ]
--------------
Cache MISS for key 10
T1: [ 10 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 9 8 ]
B2: [ ]
--------------
Cache MISS for key 11
T1: [ 11 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 10 9 8 ]
B2: [ ]
--------------
Cache MISS for key 12
T1: [ 12 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 11 10 9 8 ]
B2: [ ]

Cache MISS for key 13
T1: [ 13 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 12 11 10 9 8 ]
B2: [ ]
--------------
Cache MISS for key 1
T1: [ 1 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache HIT for key 2
T1: [ 1 ]
T2: [ 2 7 6 5 4 3 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache HIT for key 3
T1: [ 1 ]
T2: [ 3 2 7 6 5 4 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache HIT for key 4
T1: [ 1 ]
T2: [ 4 3 2 7 6 5 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
```

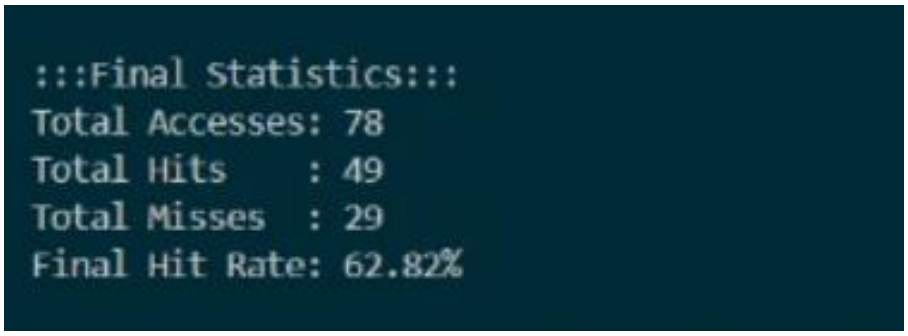
```
Cache HIT for key 5
T1: [ 1 ]
T2: [ 5 4 3 2 7 6 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache HIT for key 6
T1: [ 1 ]
T2: [ 6 5 4 3 2 7 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache HIT for key 7
T1: [ 1 ]
T2: [ 7 6 5 4 3 2 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ ]
--------------
Cache MISS for key 8
T1: [ 1 ]
T2: [ 8 7 6 5 4 3 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 2 ]
--------------
Cache MISS for key 9
T1: [ 1 ]
T2: [ 9 8 7 6 5 4 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 3 2 ]

Cache MISS for key 10
T1: [ 1 ]
T2: [ 10 9 8 7 6 5 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 4 3 2 ]
--------------
Cache MISS for key 11
T1: [ 1 ]
T2: [ 11 10 9 8 7 6 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 5 4 3 2 ]
--------------
Cache MISS for key 12
T1: [ 1 ]
T2: [ 12 11 10 9 8 7 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 6 5 4 3 2 ]
--------------
Cache MISS for key 13
T1: [ 1 ]
T2: [ 13 12 11 10 9 8 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 7 6 5 4 3 2 ]
--------------
Cache MISS for key 14
T1: [ 14 1 ]
T2: [ 13 12 11 10 9 ]
B1: [ 13 12 11 10 9 8 ]
B2: [ 8 7 6 5 4 3 ]
--------------
```

3.2.3 OBSERVATIONS

- Hit Rate Trends:
  As the simulation progresses, the hit rate improves due to the adaptive learning of the algorithm, where p dynamically adjusts based on past behavior.
- List Behavior:
  - T1 primarily holds *recently* accessed pages.
  - T2 holds *frequently* accessed pages.
  - B1 and B2 (ghost lists) help in adapting to changes in access patterns.
- Replacement Strategy:
  The replace() function determines whether to evict from T1 or T2 based on the relative sizes of ghost lists B1 and B2 and the adaptive parameter p.

3.2.4 PERFORMANCE METRICS



CHAPTER 4

CONCLUSION AND FUTURE WORK

4.1 CONCLUSION

- The ARC (Adaptive Replacement Cache) algorithm was successfully implemented and simulated in C.
- The project dynamically managed four internal lists (T1, T2, B1, B2) to efficiently adapt to varying access patterns.
- Quadratic probing hash tables were used to ensure fast and reliable cache lookups.
- The system accurately tracked cache hits and misses, allowing detailed performance analysis.
- Simulation results showed adaptive behavior based on the history of accessed and evicted pages, demonstrating the effectiveness of ARC over static replacement strategies.

4.2 FUTURE WORK

- Implement time-based eviction strategies to handle stale data in long-running systems.
- Visualize list movements and hit/miss trends using real-time graphical dashboards.
- Add support for variable-sized cache entries or data objects.
- Enable multi-threaded cache access simulation to analyze ARC performance under concurrency.
- Develop a user-friendly front-end tool for parameter tuning, custom trace input, and performance reporting.

APPENDIX

C CODE

DOUBLY LINKED LIST HEADER CODE

```
#ifndef DLL_OP_H
#define DLL_OP_H
#include <stdio.h>
#include <stdlib.h>
typedef struct Node {
    int key;
    struct Node* prev;
    struct Node* next;
} Node;
typedef struct List {
    Node* head;
    Node* tail;
    int size;
} List;
Node* getNode(int key) {
    Node* newnode = (Node*)malloc(sizeof(Node));
    newnode->key = key;
    newnode->prev = NULL;
    newnode->next = NULL;
    return newnode;
}
void initList(List* list) {
    list->head = list->tail = NULL;
    list->size = 0;
}
void innodeFront(List* list, int key) {
    Node* newnode = getNode(key);
```

```
  if (list->head == NULL) {
        list->head = list->tail = newnode;
    }
    else {
        newnode->next = list->head;
        list->head->prev = newnode;
        list->head = newnode;
    }
    list->size++;
}
void push_front_node(List* list, Node* node) {
    if (list->head == NULL) {
        list->head = list->tail = node;
    }
    else {
        node->next = list->head;
        list->head->prev = node;
        list->head = node;
    }
    list->size++;
}
Node* pop_tail(List* list) {
    if (list->tail == NULL)
        return NULL;
    Node* tail = list->tail;
    if (tail->prev) {
        tail->prev->next = NULL;
        list->tail = tail->prev;
    } else {
        list->head = list->tail = NULL;
    }
```

```
    list->size--;
    return tail;
}
void removeNode(List* list, Node* node) {
    if (node == NULL)
        return;
    if (node->prev)
        node->prev->next = node->next;
    else
        list->head = node->next;
    if (node->next)
        node->next->prev = node->prev;
    else
        list->tail = node->prev;
    free(node);
    list->size--;
}
Node* detachNode(List* list, Node* node) {
    if (node == NULL)
        return NULL;
    if (node->prev)
        node->prev->next = node->next;
    else
        list->head = node->next;
    if (node->next)
        node->next->prev = node->prev;
    else
        list->tail = node->prev;
    node->prev = node->next = NULL;
    list->size--;
 return node;
```

```
}
void displayList(List* list, const char* label) {
    printf("%s: [ ", label);
    Node* curr = list->head;
    while (curr) {
        printf("%d ", curr->key);
        curr = curr->next;
    }
    printf("]\n");
}
#endif
```

HASH TABLE HEADER CODE

```
#ifndef Hash_op_H
#define Hash_op_H
#include "DLL_op.h"
#include <math.h>
#define EMPTY 0
#define OCCUPIED 1
#define DELETED -1
#define A 0.6180339887
typedef struct HashEntry {
    int key;
    Node* node;
    int state;
} HashEntry;
typedef struct HashTable {
    int size;
    HashEntry* table;
} HashTable;
void initHashTable(HashTable* ht, int size) {
```

```
    ht->size = size;
    ht->table = (HashEntry*) malloc(sizeof(HashEntry) * size);
    for (int i = 0; i < size; i++) {
        ht->table[i].state = EMPTY;
        ht->table[i].key = 0;
        ht->table[i].node = NULL;
    }
}
int hashFunc(int key, int size) {
    if (key < 0) key = -key;
    double frac = fmod(key * A, 1.0);
    return (int)(size * frac);
}
void hashInsert(HashTable* ht, int key, Node* node) {
    int original_idx = hashFunc(key, ht->size);
    int idx;
    for (int i = 0; i < ht->size; i++) {
        idx = (original_idx + i * i) % ht->size;
        if (ht->table[idx].state != OCCUPIED || ht->table[idx].key == key) {
            ht->table[idx].key = key;
            ht->table[idx].node = node;
            ht->table[idx].state = OCCUPIED;
            return;
        }
    }
    printf("Hash table full (insert failed)!\n");
}
Node* hashSearch(HashTable* ht, int key) {
    int original_idx = hashFunc(key, ht->size);
    int idx;
    for (int i = 0; i < ht->size; i++) {
```

```
        idx = (original_idx + i * i) % ht->size;
        if (ht->table[idx].state == EMPTY) break; // Not found
        if (ht->table[idx].state == OCCUPIED && ht->table[idx].key == key) {
            return ht->table[idx].node;
        }
    }
    return NULL;
}
void hashDelete(HashTable* ht, int key) {
    int original_idx = hashFunc(key, ht->size);
    int idx;
    for (int i = 0; i < ht->size; i++) {
        idx = (original_idx + i * i) % ht->size;
        if (ht->table[idx].state == EMPTY) break; // Not found
        if (ht->table[idx].state == OCCUPIED && ht->table[idx].key == key) {
            ht->table[idx].state = DELETED;
            ht->table[idx].node = NULL;
            return;
        }
    }
}
#endif
```

ARC IMPLEMENTATION CODE

```
#include "Hash_op.h"
#define INPUT_FILE "D:\\Coding and Simulations\\Visual Studio Code (C)\\Project Nexus\\in_cache_dat.txt"
#define DECAY_INTERVAL 3
int capacity;
int p;
int requestCount = 0;
List T1, T2, B1, B2;
```

```
HashTable T1HT, T2HT, B1HT, B2HT;
void initHashTables(int);
void initARC(int);
void replace(int);
int arcAccess(int);
void printLists();
void cleanupStaleEntries();
int main() {
    FILE *fp = fopen(INPUT_FILE, "r");
    if (!fp) {
        return 1;
    }
    char line[100];
    int cap, key;
    if (fgets(line, sizeof(line), fp)) {
        sscanf(line, "Cache Capacity: %d", &cap);
    }
    initARC(cap);
    int hits = 0, misses = 0, accessCount = 0;
    while (fgets(line, sizeof(line), fp)) {
        sscanf(line, "Request Key: %d", &key);
        if (key == -1)
            break;
        accessCount++;
        int res = arcAccess(key);
        if (res == 1) {
            printf("Cache HIT for key %d\n", key);
            hits++;
        }
        else {
            printf("Cache MISS for key %d\n", key);
```

```
            misses++;
        }
        printLists();
        if (accessCount % 50 == 0) {
            double hitRate = ((double)hits / accessCount) * 100.0;
            printf("STATS AFTER %d ACCESSES\n", accessCount);
            printf("Hits: %d | Misses: %d | Hit Rate: %.2f%%\n", hits, misses, hitRate);
            printf("\n");
        }
    }
    printf("\nFINAL STATS\n");
    printf("Total Accesses: %d\n", accessCount);
    printf("Total Hits    : %d\n", hits);
    printf("Total Misses  : %d\n", misses);
    printf("Final Hit Rate: %.2f%%\n", ((double)hits / accessCount) * 100.0);
    printf("\n");
    fclose(fp);
    return 0;
}
void initHashTables(int cap) {
    int ht_size = cap * 2 + 1;
    initHashTable(&T1HT, ht_size);
    initHashTable(&T2HT, ht_size);
    initHashTable(&B1HT, ht_size);
    initHashTable(&B2HT, ht_size);
}
void initARC(int cap) {
    capacity = cap;
    p = 0;
    initList(&T1);
    initList(&T2);
```

```
    initList(&B1);
    initList(&B2);
    initHashTables(cap);
}
void cleanupStaleEntries() {
    int numToClean = 2;
    while (B1.size > 0 && numToClean--) {
        Node* stale = pop_tail(&B1);
        if (stale) {
            hashDelete(&B1HT, stale->key);
            free(stale);
        }
    }
    numToClean = 2;
    while (B2.size > 0 && numToClean--) {
        Node* stale = pop_tail(&B2);
        if (stale) {
            hashDelete(&B2HT, stale->key);
            free(stale);
        }
    }
}
void replace(int key) {
    if (T1.size > 0 && (T1.size > p || (hashSearch(&B2HT, key) != NULL && T1.size == p))) {
        Node* old = pop_tail(&T1);
        hashDelete(&T1HT, old->key);
        innodeFront(&B1, old->key);
        Node* ghost = B1.head;
        hashInsert(&B1HT, ghost->key, ghost);
        free(old);
    } else {
```

```
        Node* old = pop_tail(&T2);
        hashDelete(&T2HT, old->key);
        innodeFront(&B2, old->key);
        Node* ghost = B2.head;
        hashInsert(&B2HT, ghost->key, ghost);
        free(old);
    }
}
int arcAccess(int key) {
    Node* node = hashSearch(&T1HT, key);
    if (node != NULL) {
        detachNode(&T1, node);
        hashDelete(&T1HT, key);
        push_front_node(&T2, node);
        hashInsert(&T2HT, key, node);
        return 1;
    }
    node = hashSearch(&T2HT, key);
    if (node != NULL) {
        detachNode(&T2, node);
        hashDelete(&T2HT, key);
        push_front_node(&T2, node);
        hashInsert(&T2HT, key, node);
        return 1;
    }
    if (hashSearch(&B1HT, key) != NULL) {
        int delta = (B2.size >= B1.size) ? 1 : (B1.size / (B2.size ? B2.size : 1));
        p = (p + delta > capacity) ? capacity : p + delta;
        replace(key);
        hashDelete(&B1HT, key);
        innodeFront(&T2, key);
```

```
    Node* newNode = T2.head;
    hashInsert(&T2HT, key, newNode);
    return 0;
}
if (hashSearch(&B2HT, key) != NULL) {
    int delta = (B1.size >= B2.size) ? 1 : (B2.size / (B1.size ? B1.size : 1));
    p = (p - delta < 0) ? 0 : p - delta;
    replace(key);
    hashDelete(&B2HT, key);
    innodeFront(&T2, key);
    Node* newNode = T2.head;
    hashInsert(&T2HT, key, newNode);
    return 0;
}
if (T1.size + T2.size == capacity) {
    if (T1.size < capacity) {
        if (B2.size > 0) {
            Node* tailGhost = pop_tail(&B2);
            hashDelete(&B2HT, tailGhost->key);
            free(tailGhost);
        }
        replace(key);
    } else {
        Node* old = pop_tail(&T1);
        hashDelete(&T1HT, old->key);
        innodeFront(&B1, old->key);
        Node* ghost = B1.head;
        hashInsert(&B1HT, ghost->key, ghost);
        free(old);
    }
} else if (T1.size + T2.size < capacity && (T1.size + T2.size + B1.size + B2.size) >= capacity) {
```

```
        if ((T1.size + T2.size + B1.size + B2.size) >= 2 * capacity) {
            if (B2.size > 0) {
                Node* tailGhost = pop_tail(&B2);
                hashDelete(&B2HT, tailGhost->key);
                free(tailGhost);
            }
        }
        replace(key);
    }
    innodeFront(&T1, key);
    Node* newNode = T1.head;
    hashInsert(&T1HT, key, newNode);
    return 0;
}
void printLists() {
    displayList(&T1, "T1");
    displayList(&T2, "T2");
    displayList(&B1, "B1");
    displayList(&B2, "B2");
    printf("----------\n");
}
```

CHAPTER 5

REFERENCES

[1] N. Megiddo and D. S. Modha, “ARC: A Self-Tuning, Low Overhead Replacement Cache,” *Proc. 2nd USENIX Conf. File and Storage Technologies (FAST)*, San Francisco, CA, USA, pp. 115–130, Mar. 2003.

[2] T. Jiang and X. Zhang, “LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance,” *Proc. ACM SIGMETRICS Int. Conf. Measurement and Modeling of Computer Systems*, Marina Del Rey, CA, USA, pp. 31–42, 2002.

[3] S. Bansal and D. S. Modha, “CAR: Clock with Adaptive Replacement,” *Proc. 3rd USENIX Conf. File and Storage Technologies (FAST)*, San Francisco, CA, USA, pp. 187–200, 2004.

[4] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 10th ed., Hoboken, NJ, USA: John Wiley & Sons, 2018.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, 3rd ed., Reading, MA, USA: Addison-Wesley, 1997.

[6] M. Zaharia et al., “Discretized Streams: Fault-Tolerant Streaming Computation at Scale,” *Proc. 24th ACM Symp. Operating Systems Principles (SOSP)*, Farmington, PA, USA, pp. 423–438, 2013.

BIODATA

Name : JAYANTH S BALAN

Mobile Number : 9655456988

E-mail : jayanth.sb2022@vitstudent.ac.in

Permanent Address: C/O Balan S R, Plot No 51 Ground floor, 2nd Cross,
Behind S D A School, Appavu Nagar, Hosur,
PO: Hosur, DIST: Krishnagiri, Tamil Nadu - 635109

Name : R HARISH RAJAN

Mobile Number : 8072853645

E-mail : harishrajan.r2022@vitstudent.ac.in

Permanent Address: 204b, Palm court apartments, Thasami Park Residency,
Singanallur, Coimbatore-641005

.

Name : AKSHIT

Mobile Number : 9528918470

E-mail : akshitchaudharychaudhary950@gmail.com

Permanent Address: Village Saidpur , Govidpur Ghaziabad, Ghaziabad 201204
Uttar Pradesh , India.

APPENDIX

OVERALL DEMO – VIDEO LINK:

https://drive.google.com/file/d/1EpRuwyNEuJt9lgwUYfOM50P16nQ9iFdf/view?usp=sharing