



**EAST WEST UNIVERSITY**

**PROJECT REPORT**

Course: CSE360 (Section 01)

Semester: Fall2024

Project Group: 08

**Implement a cache replacement policy (e.g., LRU, FIFO, Random)  
in C.**

**Submitted By**

<b>Name</b>	<b>Student ID</b>	<b>Contribution</b>
Nisarga Mridha	2020-2-60-010	Code Implementation

**Submitted To**

Dr. Md. Nawab Yousuf Ali

Professor, Department of Computer Science and Engineering

East West University

**Date of Submission: 21/01/2025**

# *Table of Contents*

<b>Serial No.</b>	<b>Contents</b>	<b>Page Number</b>
1	Title	1
2	Objective	2
3	Theory	3
4	Design	5
5	Implementation	8
6	Debugging-Test-Run	11
7	Result Analysis	17
8	Conclusion and Future Improvements	18
9	Bibliography	19

*Title*

**Implement a cache replacement policy (LRU, FIFO, Random) in C.**

## *Objective*

Our project demonstrates three key cache memory replacement algorithms: **LRU (Least Recently Used)**, **FIFO (First In First Out)**, and **Random Replacement (RR)**. Cache memory boosts system performance by storing frequently accessed data, reducing slower main memory access.

When the CPU requests data, it first checks the cache. A cache hit enables quick data access, while a cache miss triggers replacement in our **3-block** cache. Each algorithm handles replacement differently: LRU tracks and replaces the least used data, FIFO replaces the oldest entries using a circular pointer, and Random replaces entries arbitrarily.

The program features a command-line interface where we can select a replacement policy and observe cache operations in real-time, including hits, misses, and replacements.

## *Theory*

In a study, Hennessy and Patterson (2011) established that cache memory optimization relies on locality principles: temporal (recently used data likely to be reduced) and spatial (nearby data likely to be accessed). The efficiency of cache memory is measured by its **Hit Ratio** - the proportion of successful cache accesses versus total memory accesses (Smith, 1982).

Our implementation focuses on three key cache replacement algorithms such as **LRU (Least Recently Used)**, **FIFO (First-In-First-Out)**, and **Random Replacement (RR)**. We are going to discuss the algorithms in the following section.

1. **LRU (Least Recently Used):** This algorithm tracks data age using counters and replaces the least recently accessed entry. A study by Mattson et al. (1970) demonstrated its near-optimal performance in real-world scenarios. LRU maintains an age counter for each cache block, updating them on every access. It selects the block with the highest age value when replacement is needed. While this provides excellent performance for programs with strong temporal locality, it requires additional memory overhead for age tracking and more complex implementation logic.
2. **FIFO (First-In-First-Out):** This algorithm operates as a circular queue, replacing the oldest entries first. Silberschatz et al. (2018) noted it may suffer from Belady's anomaly where larger cache sizes can increase page faults. FIFO maintains a single pointer that cycles through cache blocks, replacing entries in a circular order regardless of their access frequency. This approach offers simple implementation and minimal overhead but may remove frequently used data if it happens to be the oldest entry.
3. **RR (Random Replacement):** This algorithm uses probabilistic replacement, offering a balance between performance and implementation complexity (Stone et al., 1992). When a replacement is needed, it randomly selects any cache block for eviction using a pseudo-random number generator. Despite its simplicity, studies show it can perform surprisingly well in practice, especially in situations where access patterns are unpredictable or when computational overhead must be minimized. It requires no additional memory or complex logic but sacrifices predictability.

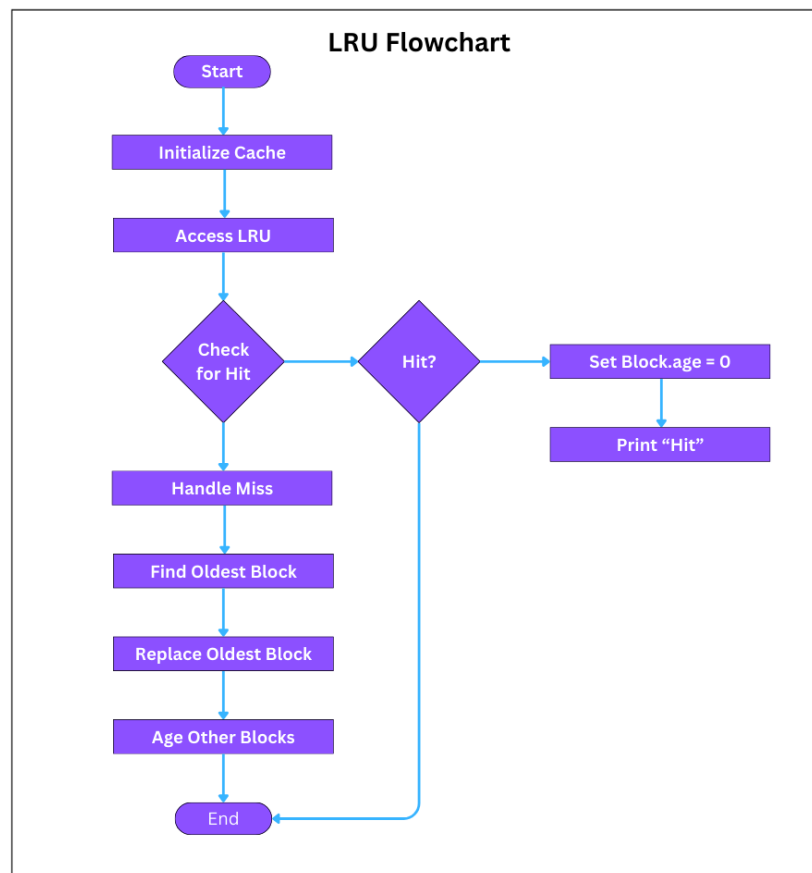
Our implementation required two key metrics of cache memory which was discussed by Jouppi and Wilton in their study. They are **Cache Hit** and **Cache Miss**. A cache hit is the percentage of successful cache accesses and a cache miss is the percentage of main memory accesses required.

## Design

Cache replacement algorithms are efficiently designed to replace the cache when the space is full; old objects must be removed to make space for new ones.

### LRU Design

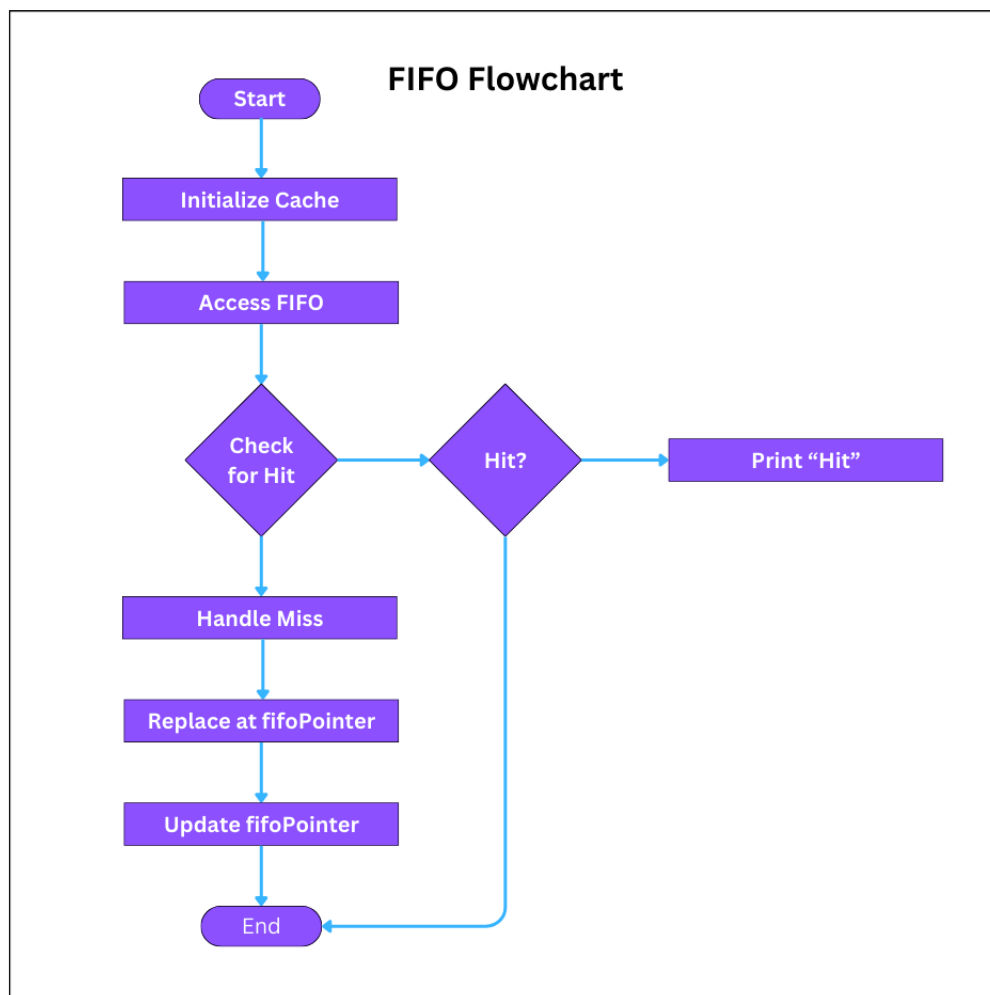
The process starts by initializing a cache of size 3, with each block containing data and age. All blocks are set to empty (data = -1, age = 0). When AccessLRU(data) is called, the system checks if the data exists in the cache. If found (hit), the block's age is reset to 0, and "Hit" is printed. If not found (miss), the system identifies the block with the highest age (least recently used), replaces its data with the new input, and resets its age to 0. The ages of all other blocks are incremented by 1. Finally, the process ends, ensuring the cache prioritizes recently used data and evicts the least recently used when needed.



**Figure 1. Flowchart of LRU Algorithm**

## FIFO Design

The process begins by initializing a cache of size 3, with each block containing data. All blocks are set to empty (data = -1), and a `fifoPointer` is initialized to 0 to track the next block for replacement. When `AccessFIFO(data)` is called, the system checks if the data exists in the cache. If found (hit), it prints "Hit" and ends. If not found (miss), the system replaces the block at the `fifoPointer` position with the new data, prints "Miss", and updates the `fifoPointer` to the next position in a circular manner ( $((\text{fifoPointer} + 1) \% \text{CACHE\_SIZE})$ ). This ensures the cache follows the First-In-First-Out (FIFO) policy, evicting the oldest data when necessary. The process then ends.

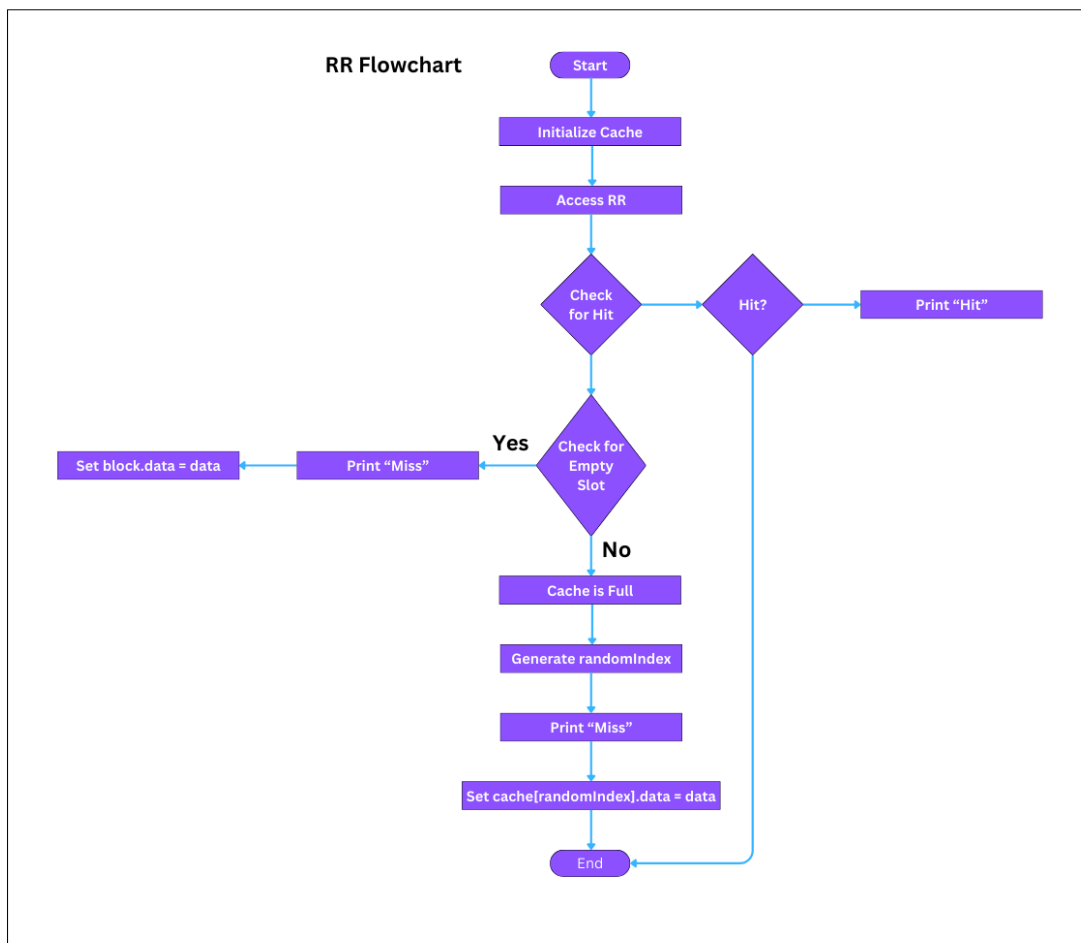


**Figure 2. Flowchart of FIFO Algorithm**



## RR Design

The process begins by initializing a cache of size 3, with each block containing data. All blocks are set to empty (data = -1). When AccessRandom(data) is called, the system checks if the data exists in the cache. If found (hit), it prints "Hit" and ends. If not found (miss), the system first checks for an empty block (data = -1). If an empty block is found, the data is inserted there, and "Miss" is printed. If the cache is full, a random block is selected using a randomly generated index between 0 and 2. The data in the selected block is replaced with the new input, and "Miss" is printed. This ensures the cache follows the Random Replacement policy, evicting data unpredictably when necessary. The process then ends.



**Figure 3. Flowchart of RR Algorithm**

# *Implementation*

Pseudocode of the used algorithms (**LRU**, **FIFO**, and **RR**) are given below.

## **LRU Implementation**

Initialize:

```
Create cache of size 3 with each block having {data, age}
Set all data = -1 (empty)
Set all age = 0
```

Function AccessLRU(data):

```
// Check for hit
FOR each block in cache:
    IF block.data equals data THEN
        Set block.age = 0
        Print "Hit"
        RETURN
    ENDIF
ENDFOR
```

```
// Handle miss - find oldest block
oldest_block = first block
max_age = first block's age
FOR each block in cache:
    IF block.age > max_age THEN
        oldest_block = block
        max_age = block.age
    ENDIF
ENDFOR
```

```
// Replace oldest block
Print "Miss"
Set oldest_block.data = data
Set oldest_block.age = 0
```

```
// Age other blocks
FOR each block in cache:
    IF block is not oldest_block THEN
        Increment block.age
```

```

    ENDIF
  ENDFOR

```

## FIFO Implementation

Initialize:

```

  Create cache of size 3 with each block having {data}
  Set all data = -1 (empty)
  Set fifoPointer = 0

```

Function AccessFIFO(data):

```

  // Check for hit
  FOR each block in cache:
    IF block.data equals data THEN
      Print "Hit"
      RETURN
    ENDIF
  ENDFOR

  // Handle miss - replace at pointer position
  Print "Miss"
  Set cache[fifoPointer].data = data
  fifoPointer = (fifoPointer + 1) % CACHE_SIZE

```

## RR Implementation

Initialize:

```

  Create cache of size 3 with each block having {data}
  Set all data = -1 (empty)

```

Function AccessRandom(data):

```

  // Check for hit
  FOR each block in cache:
    IF block.data equals data THEN
      Print "Hit"
      RETURN
    ENDIF
  ENDFOR

```

```

  // Check for empty slot
  FOR each block in cache:

```

```
    IF block.data equals -1 THEN
        Print "Miss"
        Set block.data = data
        RETURN
    ENDIF
ENDFOR

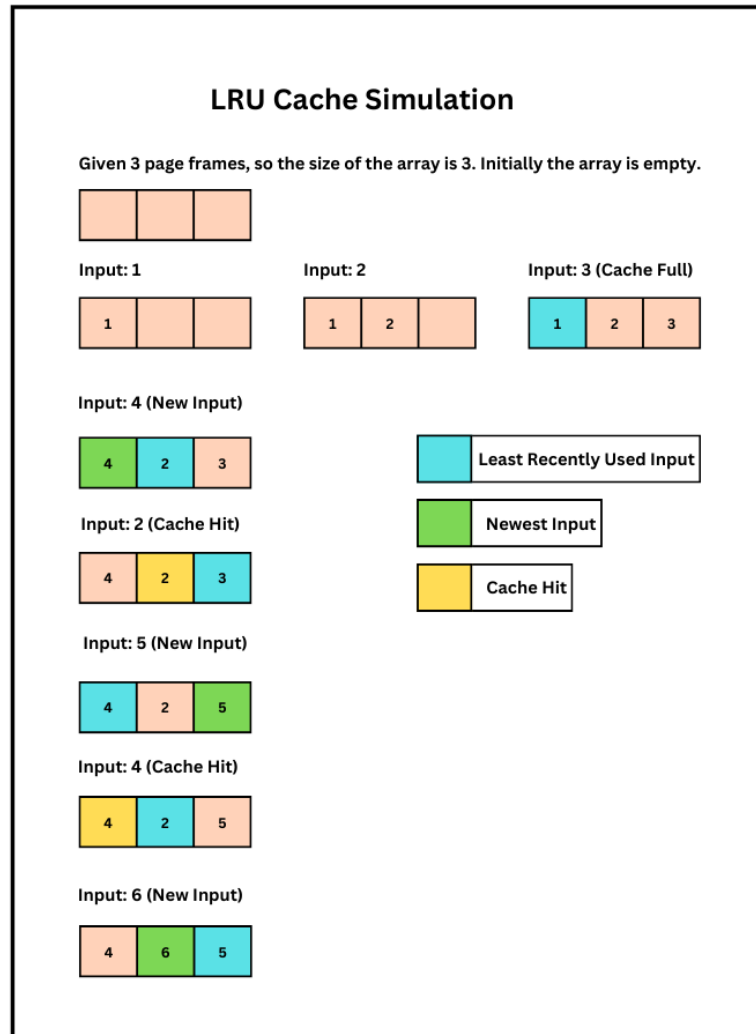
// Cache is full - random replacement
randomIndex = Generate random number between 0 and 2
Print "Miss"
Set cache[randomIndex].data = data
```

## *Debugging-Test-Run*

The Debugging-Test-run section ensures the reliability and accuracy of the cache simulation program through systematic testing and debugging. We have used the test cases: 1, 2, 3, 4, 2, 5, 4, 6, to evaluate cache hits, misses, and replacements.

```
PS C:\Users\nextn\Downloads\Git\Cache Replacement Policies\Code\output> & .\'cache_rep.exe'
Choose Cache Replacement Policy:
1. LRU
2. FIFO
3. Random
1
Enter data to access (-1 to exit): 1 2 3 4 2 5 4 6
Miss: Replacing -1 with 1
Cache contents: 1 - -
Enter data to access (-1 to exit): Miss: Replacing -1 with 2
Cache contents: 1 2 -
Enter data to access (-1 to exit): Miss: Replacing -1 with 3
Cache contents: 1 2 3
Enter data to access (-1 to exit): Miss: Replacing 1 with 4
Cache contents: 4 2 3
Enter data to access (-1 to exit): Hit: 2
Cache contents: 4 2 3
Enter data to access (-1 to exit): Miss: Replacing 3 with 5
Cache contents: 4 2 5
Enter data to access (-1 to exit): Hit: 4
Cache contents: 4 2 5
Enter data to access (-1 to exit): Miss: Replacing 2 with 6
Cache contents: 4 6 5
Enter data to access (-1 to exit): []
```

**Figure 4. Test Result of LRU Algorithm**



**Figure 5. A Simulation of the Above Test Result**

```
PS C:\Users\nextn\Downloads\Git\Cache Replacement Policies\Code\output> & .\'cache_rep.exe'  
Choose Cache Replacement Policy:  
1. LRU  
2. FIFO  
3. Random  
2  
Enter data to access (-1 to exit): 1 2 3 4 2 5 4 6  
Miss: Replacing -1 with 1  
Cache contents: 1 - -  
Enter data to access (-1 to exit): Miss: Replacing -1 with 2  
Cache contents: 1 2 -  
Enter data to access (-1 to exit): Miss: Replacing -1 with 3  
Cache contents: 1 2 3  
Enter data to access (-1 to exit): Miss: Replacing 1 with 4  
Cache contents: 4 2 3  
Enter data to access (-1 to exit): Hit: 2  
Cache contents: 4 2 3  
Enter data to access (-1 to exit): Miss: Replacing 2 with 5  
Cache contents: 4 5 3  
Enter data to access (-1 to exit): Hit: 4  
Cache contents: 4 5 3  
Enter data to access (-1 to exit): Miss: Replacing 3 with 6  
Cache contents: 4 5 6  
Enter data to access (-1 to exit): 
```


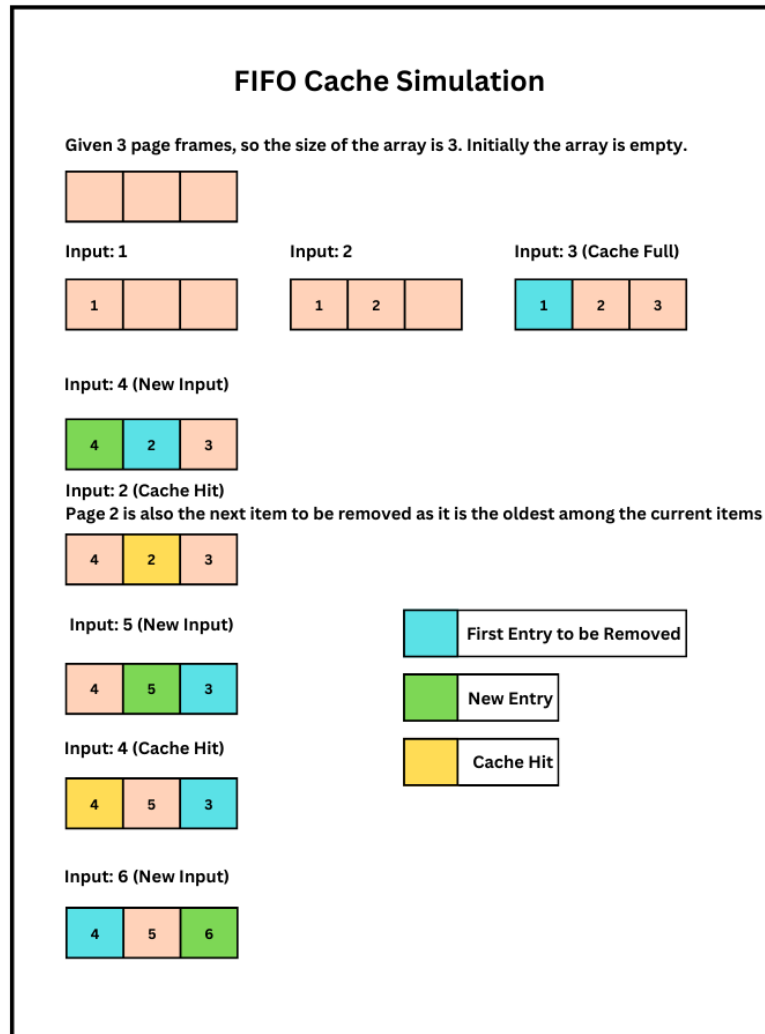
 Compiled successfully!

Figure 6. Test Result of FIFO Algorithm



**Figure 7. A Simulation of the Above Test Result**

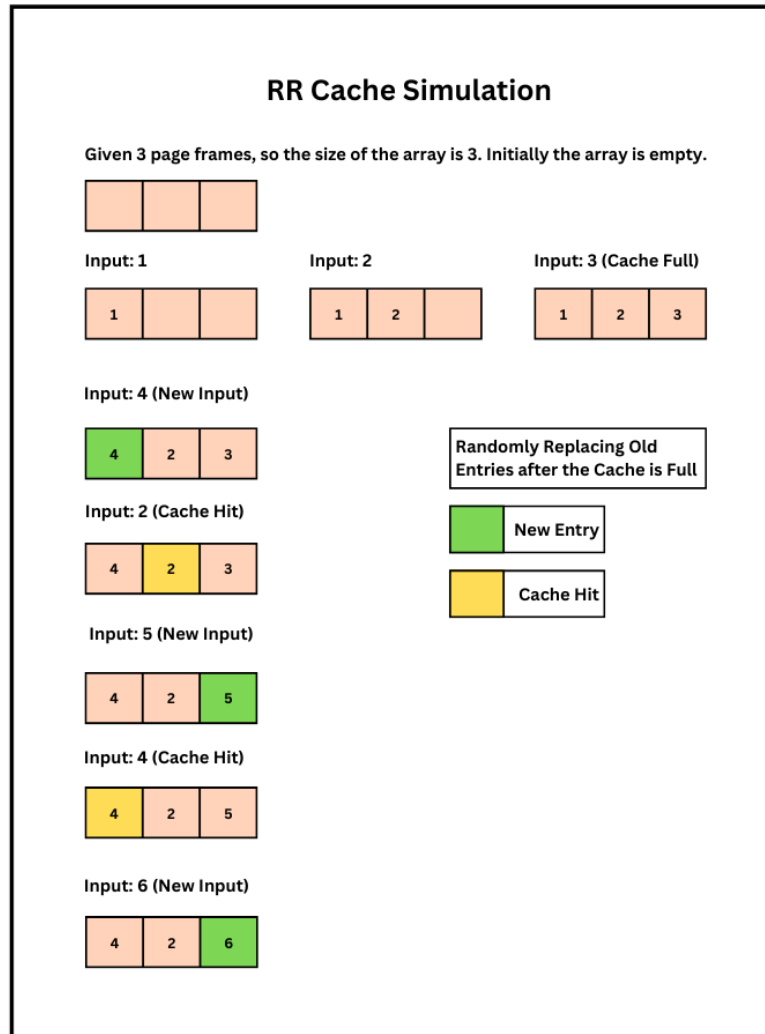


```

PS C:\Users\nextn\Downloads\Git\Cache Replacement Policies\Code\output> cd 'c:\Users\nextn\I
ment Policies\Code\output'
PS C:\Users\nextn\Downloads\Git\Cache Replacement Policies\Code\output> & .\'cache_rep.exe'
Choose Cache Replacement Policy:
1. LRU
2. FIFO
3. Random
3
Enter data to access (-1 to exit): 1 2 3 4 2 5 4 6
Miss: Adding 1 to cache
Cache contents: 1 - -
Enter data to access (-1 to exit): Miss: Adding 2 to cache
Cache contents: 1 2 -
Enter data to access (-1 to exit): Miss: Adding 3 to cache
Cache contents: 1 2 3
Enter data to access (-1 to exit): Miss: Replacing 3 with 4
Cache contents: 1 2 4
Enter data to access (-1 to exit): Hit: 2
Cache contents: 1 2 4
Enter data to access (-1 to exit): Miss: Replacing 4 with 5
Cache contents: 1 2 5
Enter data to access (-1 to exit): Miss: Replacing 2 with 4
Cache contents: 1 4 5
Enter data to access (-1 to exit): Miss: Replacing 4 with 6
Cache contents: 1 6 5
Enter data to access (-1 to exit): 

```

Figure 8. Test Result of RR Algorithm



**Figure 9. A Simulation of the Above Result**

## *Result Analysis*

### **Time Complexity**

All three algorithms (**LRU**, **FIFO**, **Random**) have a time complexity of  $O(n)$  per access, where  $n$  is the cache size. This is because each access requires checking all cache blocks for hits or misses. For LRU, additional operations like finding the least recently used block and updating ages also contribute to the  $O(n)$  complexity. FIFO and Random perform replacements in constant time ( $O(1)$ ), but the initial search for hits remains  $O(n)$ .

### **Space Complexity**

The space complexity is  $O(n)$ , driven by the cache storage, which is an array of `CacheBlock` structures. Each block stores data and, for LRU, an age field. Additional variables like `fifoPointer` use constant space ( $O(1)$ ), making the overall space complexity  $O(n)$ . This design ensures memory efficiency, particularly for small cache sizes.

## *Conclusion and Future Improvements*

This project implemented a cache simulation with three replacement policies: **LRU**, **FIFO**, and **RR**. Our implementation demonstrated cache operations like cache hits, misses, and replacements. Although, our work have a major limitation which is it performs using a cache size of 3 with  **$O(n)$**  time complexity. Our findings say that this issue can be optimized with hash maps or priority queues for larger caches and diverse data types such as strings in future.

## *Bibliography*

1. Belady, L.A. (1966). A study of replacement algorithms for virtual-storage computer. IBM Systems Journal
2. Denning, P.J. (1968). The working set model for program behavior. Communications of the ACM
3. Hennessy, J.L., & Patterson, D.A. (2011). Computer Architecture: A Quantitative Approach
4. Mattson, R.L., et al. (1970). Evaluation techniques for storage hierarchies. IBM Systems Journal
5. Silberschatz, A., et al. (2018). Operating System Concepts
6. Smith, A.J. (1982). Cache Memories. ACM Computing Surveys
7. Tanenbaum, A.S., & Bos, H. (2014). Modern Operating Systems