# PROGRAMMING-LEAST RECENTLY USED-IMPLEMENTATION

LRU Properties:

* Whenever information in a buffer is accessed, this buffer is brought

to the front of the list.

* The way to ensure that the block is written when necessary, but only when necessary, is to maintain a Boolean variable with the buffer (dirty bit) that is turned on when the buffer’s contents are modified by the client.
* Method acquireBuffer takes a block ID as input and returns a pointer to the

buffer that will be used to store this block.

* Sequential access processes records in order of their logical appearance within the file.

LRUBufferPool Class:

Purpose:

* Implements an LRU (Least Recently Used) buffer pool that caches file data in memory using BufferBlock objects.
* fileStream: A stream to read data from the file.
* blockArray[]: An array of BufferBlock objects (fixed size of POOL\_SIZE).
* file: The file path used to initialize the buffer pool.

Methods:

* Constructor: Initializes the buffer pool with the file data.
* Sets up blockArray[] with BufferBlock objects (setting their IDs, head, and tail).
* Reads the initial blocks of data from the file into the buffer pool.
* Destructor: Currently does nothing. (every time I tried to initialize it to use and delete nullPtr’s, I got some very interesting errors).
* getBytes(char\* space, int sz, int pos): Checks if the data exists in the buffer pool (by comparing pos to the block's head and tail).
  + If the data is found, it is copied to space, and the block is reordered to the front (most recently used).
  + If the data is not in memory, it reads the required bytes from the file into space and adds the block to the buffer pool.
* printBufferBlockOrder(): Prints the current order of the blocks in the buffer pool, from most recently used (front) to least recently used (back).
* getLRUBlockID(): Returns the ID of the least recently used block (the one at the end of the buffer pool).
* LRUNode(string file, int pos): Loads a new block into the buffer pool (if not already in memory), shifting other blocks to make space. Reads the block from the file and stores it at the front of the buffer pool.
* LRUNodeReordered(int block): Reorders the blocks in the buffer pool by moving the specified block to the front (most recently used). Shifts other blocks down and places the given block at index 0.

BufferBlock Class:

Purpose:

* Represents a single block of data in the buffer pool.

Attributes:

* blockID: Unique identifier for the block.
* blockSize: Size of the block (defaulted to BLOCKSIZE).
* head: The starting position of the block in the file.
* tail: The ending position of the block in the file.
* blockPtr: A pointer to the block's data in memory.

Methods:

* Constructor: Initializes blockPtr and blockSize.
* Destructor: Resets head and tail to -1 to indicate that the block is not assigned.
* getData(int pos, int sz, char\* data): Copies sz bytes of data from the block's memory, starting at the position pos.
* getID(): Returns the block's unique identifier.
* setID(int id): Sets the block’s unique identifier.
* setBlock(char\* blk): Assigns the data pointer to blockPtr and calculates blockSize based on the length of the data.
* getBlock(): Returns the pointer to the block's data.
* getBlocksize(): Returns the size of the block.
* setHead(int start): Sets the block’s start position (head).
* getHead(): Returns the block’s start position (head).
* setTail(int end): Sets the block’s end position (tail).
* getTail(): Returns the block’s end position (tail).

1. **Initialization**:
   * The LRUBufferPool initializes an array of BufferBlock objects, each representing a block of data from the file.
   * Data is read from the file and loaded into memory during initialization.
2. **Data Access**:
   * When data is requested, the LRUBufferPool checks if the requested data is already in memory in the BufferBlock objects.
   * If the data is found (cache hit), it is returned, and the corresponding block is moved to the front (most recently used).
   * If the data is not found (cache miss), it is read from the file, and a new BufferBlock is created and placed at the front of the pool.
3. **Eviction (when memory is full)**:
   * If the pool is full, the least recently used block is evicted (the one at the back of the pool), and a new block is loaded into its place.
4. **LRU Order Maintenance**:
   * After each access, the BufferBlock representing the accessed data is moved to the front of the pool (most recently used).
   * The order in the blockArray[] always reflects the LRU order, with the least recently used block at the back.