LAB-7

Software Engineering - IT314

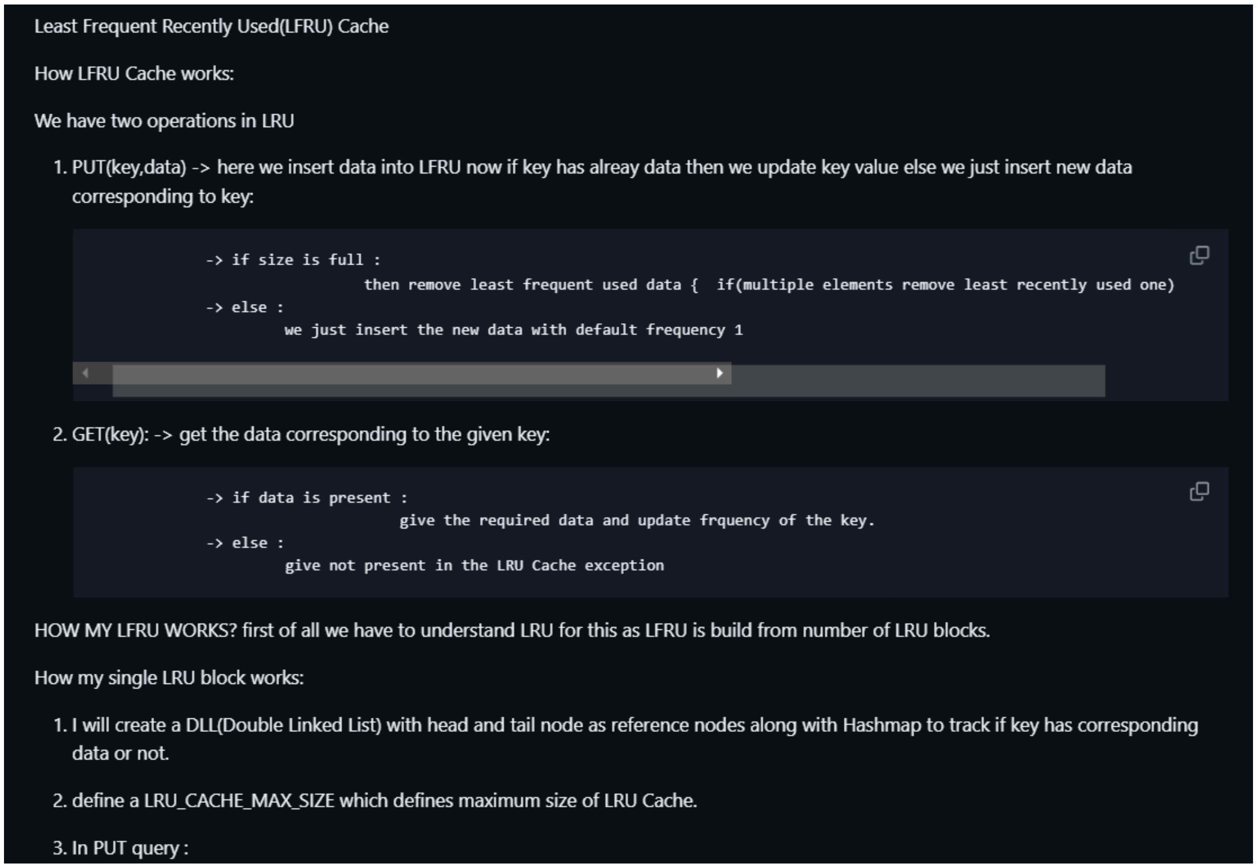
202201106 - Vraj Dobariya

I. PROGRAM INSPECTION:

500 line CODE: CPP file located in REPO as well.

GITHUB REPO LINK: LFRU

LFRU Implementation:



Least Frequent Recently Used(LFRU) Cache

How LFRU Cache works:

We have two operations in LRU

1. PUT(key,data) -> here we insert data into LFRU now if key has alreay data then we update key value else we just insert new data corresponding to key:

```
        -> if size is full :
                    then remove least frequent used data {  if(multiple elements remove least recently used one)
        -> else :
              we just insert the new data with default frequency 1
```

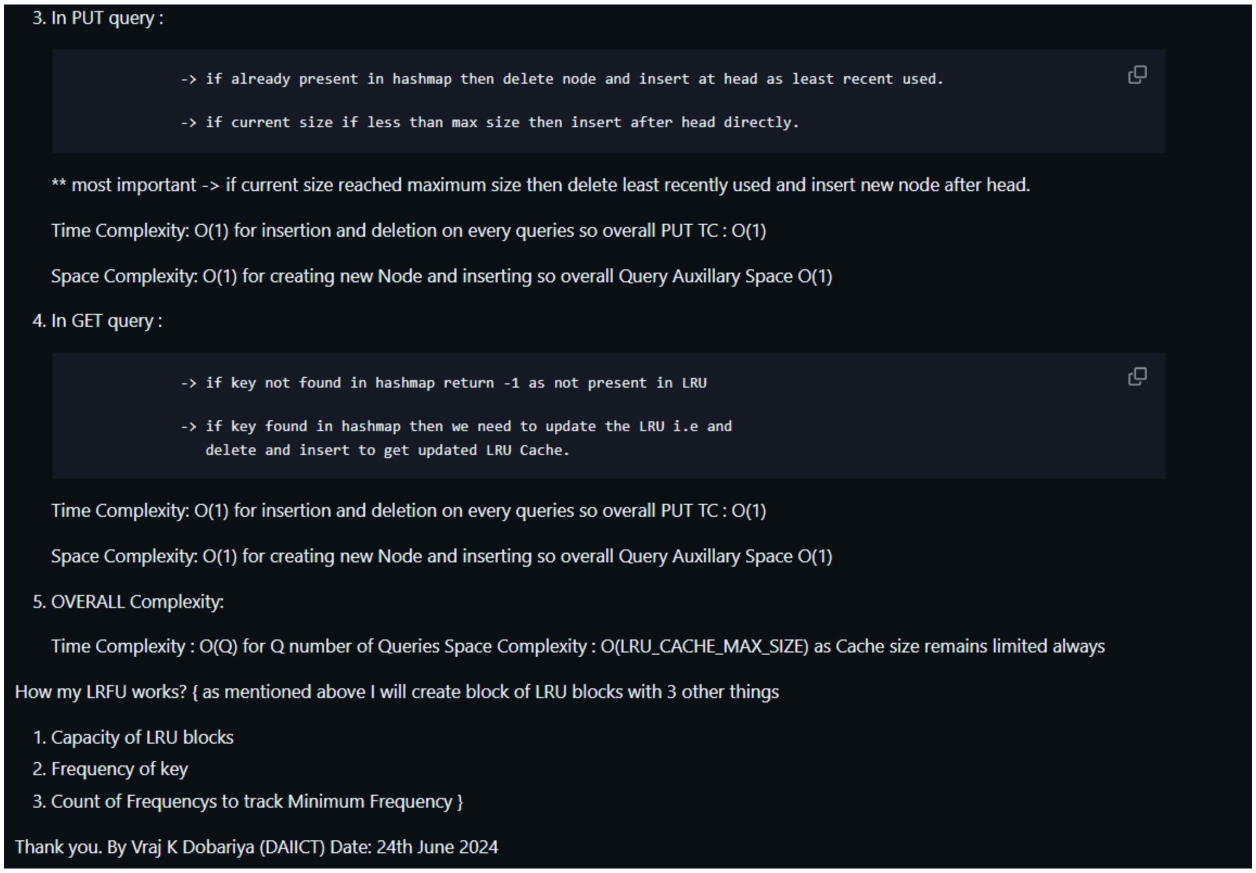
2. GET(key): -> get the data corresponding to the given key:

```
        -> if data is present :
                       give the required data and update frquency of the key.
        -> else :
              give not present in the LRU Cache exception
```

HOW MY LFRU WORKS? first of all we have to understand LRU for this as LFRU is build from number of LRU blocks.

How my single LRU block works:

1. I will create a DLL(Double Linked List) with head and tail node as reference nodes along with Hashmap to track if key has corresponding data or not.
2. define a LRU_CACHE_MAX_SIZE which defines maximum size of LRU Cache.
3. In PUT query :



3. In PUT query :

```
        -> if already present in hashmap then delete node and insert at head as least recent used.

        -> if current size if less than max size then insert after head directly.
```

** most important -> if current size reached maximum size then delete least recently used and insert new node after head.

Time Complexity: O(1) for insertion and deletion on every queries so overall PUT TC : O(1)

Space Complexity: O(1) for creating new Node and inserting so overall Query Auxillary Space O(1)

4. In GET query :

```
        -> if key not found in hashmap return -1 as not present in LRU

        -> if key found in hashmap then we need to update the LRU i.e and
           delete and insert to get updated LRU Cache.
```

Time Complexity: O(1) for insertion and deletion on every queries so overall PUT TC : O(1)

Space Complexity: O(1) for creating new Node and inserting so overall Query Auxillary Space O(1)

5. OVERALL Complexity:

Time Complexity : O(Q) for Q number of Queries Space Complexity : O(LRU_CACHE_MAX_SIZE) as Cache size remains limited always

How my LRFU works? { as mentioned above I will create block of LRU blocks with 3 other things

1. Capacity of LRU blocks
2. Frequency of key
3. Count of Frequencys to track Minimum Frequency }

Thank you. By Vraj K Dobariya (DAIICT) Date: 24th June 2024

```
// By Vraj K Dobariya (DAIICT)
// Date: 24th June 2024
#include <bits/stdc++.h>
using namespace std;

typedef long long int ll;
typedef long double lld;
typedef unsigned long long ull;

#define int ll
// just to scan input from input.txt and print output in output.tx
// void init_code()
// {
```

```
//     #ifndef ONLINE_JUDGE
//     freopen("input.txt", "r", stdin);
//     freopen("output.txt", "w", stdout);
//     #endif // ONLINE_JUDGE
// }



class Node{

    public:
    int key;
    int val;
    Node* next;
    Node* prev;

    // default constructor
    Node()
    {
      this->next=NULL;
      this->prev=NULL;
    }
   // new node constructor
    Node(int key,int val)
    {
      this->key = key;
      this->val = val;
      this->next=NULL;
      this->prev=NULL;
```

```
     }
};
// just for printing purpose
string s(21,'-');
const int DEFAULT_LRU_SIZE = 1000;
class LRU_CACHE{
public:
    Node* head;
    Node* tail;
    int LRU_MAX_CACHE_SIZE;
    int Curr_Size;
    unordered_map<int,Node*>Hashmap;
    // default constructor
    LRU_CACHE()
    {
        head=new Node(-1,-1);
        tail=new Node(-1,-1);
        head->next=tail;
        tail->prev=head;
        LRU_MAX_CACHE_SIZE=DEFAULT_LRU_SIZE;
        Curr_Size=0;
 //    head -------> tail
 //    head <------- tail
    }
    LRU_CACHE(int LRU_MAX_CACHE_SIZE)
    {
        head=new Node(-1,-1);
        tail=new Node(-1,-1);
        head->next=tail;
        tail->prev=head;
        this->LRU_MAX_CACHE_SIZE=LRU_MAX_CACHE_SIZE;
        this->Curr_Size=0;
        Hashmap.clear();
// initialization by constructor
 //    head ------> tail
 //    head <------ tail
    }


    void setCacheSize(int size_)
    {
          LRU_MAX_CACHE_SIZE=size_;
    }
    void PrintCurrentCache()
    {

        cout<<"\n";
        for(int i=0;i<LRU_MAX_CACHE_SIZE+1;i++)
        cout<<s;
        cout<<"\n";

        Node* curr = head;

        while(curr!=tail)
        {
            if(curr==head)
            {
                cout<<"   => { head <--->";
            }
            else
            cout<<" [k:"<<curr->key<<", v:"<<curr->val<<"] <--->";

            curr=curr->next;
        }


        cout<<" tail } <=";
        cout<<"\n";
        for(int i=0;i<LRU_MAX_CACHE_SIZE+1;i++)
        cout<<s;
        cout<<"\n";
    }


    Node* insertafterHead(int key,int value)
    {

        /*

        before insertion after head
        head ---> nextnode .... ---> tail
        head <--- nextnode .... <--- tail

        after insertion after head
        head ---> newNode ---> nextnode .... ---> tail
        head <--- newNode <--- nextnode .... <--- tail
        */

        Curr_Size++;

        Node* nextnodeafterhead=head->next;

        Node* newNode = new Node(key,value);

        // inserting or updating in Hashmap with key
        Hashmap[key]=newNode;

        head->next = newNode;
        newNode->prev=head;
        newNode->next=nextnodeafterhead;
        nextnodeafterhead->prev=newNode;

      return newNode;
    }


    void deletebeforeTail()
    {

        /*

        tail before deletion
        head ---> .... prevnode ---> deleteNode ---> tail
        head <--- .... prevnode <--- deleteNode <--- tail
```

```
     tail after deletion
     head ---> .... prevnode ---> tail
     head <--- .... prevnode <--- tail

     */


    Curr_Size--;
    Node* deleteNode = tail->prev;
    Node* prevNode = deleteNode->prev;

    int key = deleteNode->key;

    // removing key from Hashmap
    Hashmap.erase(Hashmap.find(key));


    // assigning pointers correctly
    prevNode->next = tail;
    tail->prev= prevNode;

    delete deleteNode;



}

void deleteNode(int key)
{
```

```
/*
before deletion

.... prevNode ---> deleteNode ---> nextNode ....
.... prevNode <--- deleteNode <--- nextNode ....


after  deletion

.... prevNode ---> nextNode ....
.... prevNode <--- nextNode ....

*/

// this function is called only when we are sure that our key node is present so will not lead to any ambiguity.

// removing key from Hashmap

Curr_Size--;
// getting prevnode of the node to be deleted


Node* deleteNode = Hashmap[key];

Node* prevNode = deleteNode->prev;

// assigning pointers
Node* afterdeleteNode = deleteNode->next;
```

```
    prevNode->next = afterdeleteNode;
    afterdeleteNode->prev = prevNode;
    Hashmap.erase(Hashmap.find(key));

    delete deleteNode;


}

int get(int key)
{
    /*

    LRU before get

    head --> ..... keynode ... --> tail
    head <-- ..... keynode ... <-- tail

    LRU after get
    1)gets deleted from original position
    2)gets inserted after head making the most recently used

    head --> keynode ... --> tail
    head <-- keynode ... <-- tail




    */
    if(Hashmap.find(key)==Hashmap.end())
    {
        // if key not found in Hashmap return -1 as not present in LRU
        return -1;
    }
    else
    {

        // if key found in Hashmap then we need to update the LRU.
        // => delete key value and insert the same after head

        int returnval = Hashmap[key]->val;
        // delete and insert to get updated LRU
        deleteNode(key);

        insertafterHead(key,returnval);

       return returnval;

    }


}


void put(int key,int value)
{
        if(Hashmap.find(key)!=Hashmap.end())
        {
            // if already present in Hashmap then delete node and insert at
head as last recent used
            deleteNode(key);
            insertafterHead(key,value);
        }
        else{

            if(Curr_Size<LRU_MAX_CACHE_SIZE)
            {
                // if current size if less than max size then insert after head
directly
                insertafterHead(key,value);
            }
            else
            {
                // size if maximized so delete least recently used and insert
new node after head
                deletebeforeTail();
                insertafterHead(key,value);
            }
        }
    }


    Node* giveNodebeforeTail()
    {
        // gives previous Node of the tail.
        return tail->prev;
    }
```

```
};

const int DEFAULT_SIZE = 10010;
class LFUCache {
public:
    // basically LFU uses LRU as a Building Block.
    LRU_CACHE LFU[DEFAULT_SIZE];
    int Capacity;
    int Curr_Size;
    unordered_map<int,Node*>Hashmap;
    unordered_map<int,int>FreqOfKey;
    map<int,int>CountOfFreq;

    LFUCache(int Capacity) {
        this->Capacity=Capacity;
        Curr_Size=0;
    }

    int get(int key) {
        // get will search for the key in Hashmap and if not found then will
return -1
        if(Hashmap.find(key)==Hashmap.end())
        return -1;
        else
        {
            // if key found then key is being accessed so it's frequency
            // should be increased by 1 then we shift him to upper frequency
LRU.
            int res = Hashmap[key]->val;
            LFU[FreqOfKey[key]].deleteNode(key);

            // count of freq will remove previous frequency and add new
frequency
            this->RemoveFromCountOfFreq(FreqOfKey[key]);
            FreqOfKey[key]++;
            this->AddIntoCountOfFreq(FreqOfKey[key]);



            // insert function in LFU[freq] will return node at which it is
            // inserted after removing from previous position.
            // as we update Hashmap as well.
            Hashmap[key] = LFU[FreqOfKey[key]].insertafterHead(key,res);
            return res;
        }
    }



    // function to put key with data.
    void put(int key, int value) {

                if(Hashmap.find(key)!=Hashmap.end())
                {
                    // deleting key as it is already present and then putting another
key
                    LFU[FreqOfKey[key]].deleteNode(key);
                    // count of freq will remove previous frequency and add new
frequency
                    this->RemoveFromCountOfFreq(FreqOfKey[key]);
                    FreqOfKey[key]++;
                    this->AddIntoCountOfFreq(FreqOfKey[key]);

                    Hashmap[key] =
LFU[FreqOfKey[key]].insertafterHead(key,value);
                }
                else
                {

                    if(Curr_Size < Capacity)
                    {
                        FreqOfKey[key] = 1;
                        CountOfFreq[FreqOfKey[key]]++;
                        Hashmap[key] =
LFU[FreqOfKey[key]].insertafterHead(key,value);
                        Curr_Size++;
                    }
                    else
                    {
                        // getting minimum frequency which has some elements
present in it.
                        int MinimumFreq = getMinimumCountOfKey();
                        Node* NodeTobeDeleted =
LFU[MinimumFreq].giveNodebeforeTail();
```

```
                    int keydeleted = NodeTobeDeleted->key;

                    LFU[MinimumFreq].deletebeforeTail();

                  Hashmap.erase(keydeleted);
                  FreqOfKey.erase(keydeleted);

                // count of freq will remove minimum frequency and add new
frequency
                  this->RemoveFromCountOfFreq(MinimumFreq);
                  FreqOfKey[key] = 1;
                  this->AddIntoCountOfFreq(FreqOfKey[key]);

                // insert function in LFU[freq] will return node at which it is
                // inserted after removing from previous position.
                // as we update Hashmap as well.
                  Hashmap[key] =
LFU[FreqOfKey[key]].insertafterHead(key,value);
                }
            }
    }
// function to add count of frequency
     void AddIntoCountOfFreq(int frequency){
        CountOfFreq[frequency]++;
    }
// function to remove count of frequency and if any frequency goes down to
// zero we remove the whole key from it
    void RemoveFromCountOfFreq(int frequency)
    {
```

```
                    CountOfFreq[frequency]--;
                    if(CountOfFreq[frequency]==0)
                    {
                        CountOfFreq.erase(frequency);
                    }
    }
    // function to return minimum element from count of frequency i.e. first
element.
    int getMinimumCountOfKey(){
            auto mnFreqOfKeyitr=CountOfFreq.begin();
            return mnFreqOfKeyitr->first;
    }

    // function to print hashmap
    void printhash()
    {
        for(auto it:Hashmap){
            cout<<it.first<<" "<<it.second->val<<"\n";
        }
    }

    // function to print Current Cache
    void PrintCurrentCache()
    {
        for(int i=0;i<DEFAULT_SIZE;i++)
        {
            if(LFU[i].Curr_Size)
            {
                LFU[i].PrintCurrentCache();
            }
        }
    }
};



void solve(){


   int LFU_MAX_CACHE_SIZE;
   cout<<"ENTER THE MAXIMUM CACHE SIZE OF LFU CACHE: ";
   cin>>LFU_MAX_CACHE_SIZE;
   cout<<"\n";


   LFUCache MyLFU(LFU_MAX_CACHE_SIZE);

    // if you want limited queries
    cout<<"ENTER THE TOTAL NUMBER OF INTIAL QUERIES: ";
    int TOTAL_QUERIES;
    cin>>TOTAL_QUERIES;
    cout<<"\n";
//   infinite queries
//  int cnt
   while(TOTAL_QUERIES--){

    int WHICH_QUERY;
    // 1 for put and 2 for get
```

```
cout<<"Enter 1 for PUT or 2 for GET query or -1 to stop queries: ";
cin>>WHICH_QUERY;
cout<<"\n";


if(WHICH_QUERY==1)
{
   cout<<"ENTER KEY AND VALUE FOR THE PUT QUERY: ";

   int Put_Key,Put_Value;

   cin>>Put_Key>>Put_Value;
   cout<<"\n";

   MyLFU.put(Put_Key,Put_Value);
   MyLFU.printhash();

}
else if(WHICH_QUERY==2)
{
   cout<<"ENTER KEY FOR THE GET QUERY: ";
   int Get_Key;
   cin>>Get_Key;
   cout<<"\n";

   cout<<"\nValue: "<<MyLFU.get(Get_Key)<<"\n";

}
else if(WHICH_QUERY==-1)
```

```
    {
        // remaining cache:
         MyLFU.PrintCurrentCache();
         // stop recieving queries
         break;
    }
    else
    {
        // not a valid query
        cout<<"INVALID QUERY! \n";
        continue;
    }
    // printing CURRENT LRU CACHE
    MyLFU.PrintCurrentCache();
    cout<<"\n";
   }
}


signed main() {

// init_code();
int t=1;
// cin>>t;
while(t--)
solve();

}
```

above code is LFRU implementation code and it is an executable code accepted on LEETCODE SPOJ and InterviewBIT

1. How many errors are there in the program? Mention the errors you have identified.

- **Errors Identified**:
    1. **Memory Leak**: In the `LRU_CACHE` class, if the constructor initializes the cache without freeing memory during deletion or reallocation, it may lead to memory leaks.
    2. **Lack of Input Validation**: In `solve()`, there is no check for invalid input when reading `LFU_MAX_CACHE_SIZE` or `TOTAL_QUERIES`. Invalid inputs could lead to undefined behavior.
    3. **Potential Dereferencing of Null Pointers**: The `deletebeforeTail()` and `deleteNode(int key)` functions do not check if the `deleteNode` pointer is null before attempting to access its members. This could lead to dereferencing null pointers.
    4. **Redundant Hashmap Erasure**: In `deleteNode(int key)`, `Hashmap.erase(Hashmap.find(key));` can be simplified to `Hashmap.erase(key);`.

5. **Constant Limitation**: The `DEFAULT_SIZE` in the `LFUCache` class is hardcoded to 10010. This may lead to inefficiencies if a user needs to create a larger cache.

2. Which category of program inspection would you find more effective?

- **Static Code Review**: This approach allows for the identification of issues without executing the program. It enables the inspection of the entire codebase for potential problems such as memory management, variable initialization, and data structure integrity.

3. Which type of error are you not able to identify using the program inspection?

- **Runtime Errors**: Certain errors may only surface during execution, such as:
  - **Segmentation Faults**: Attempting to dereference null pointers can only be identified when the program is run.
  - **Logic Errors**: These are flaws in the algorithm's logic that may not throw exceptions but lead to incorrect outputs.
  - **Performance Issues**: Problems related to the performance (like excessive memory usage or time complexity issues) often need profiling during runtime to identify.

4. Is the program inspection technique worth applicable?

- **Yes, Program Inspection is Worthwhile**:
  - It helps in catching errors early in the development process before runtime testing.
  - It promotes code quality, maintainability, and helps ensure adherence to coding standards.

  - While it cannot identify all types of errors, especially runtime ones, it significantly reduces the number of bugs and enhances code readability and robustness.

II. `Code Debugging:`

1. How many errors are there in the program? Mention the errors you have identified.

Errors Identified:

- Redundant Typedefs: The use of `#define int ll` effectively makes all integers 64-bit, which might not be necessary. This could lead to performance and memory inefficiencies, especially when handling small numbers.
- Memory Leaks: The program uses dynamic memory allocation (`new Node(...)`) in multiple places without proper deallocation (`delete`). In large caches or long-running processes, this could lead to memory leaks.
- Undefined Online Judge Code: The function `init_code()` for redirecting input/output is commented out but still present in the main function. This could lead to confusion. Also, the comment indicates it's intended for competitive programming environments.
- `DEFAULT_SIZE` Is Very Large: The constant `DEFAULT_SIZE = 10010` is excessively large for most practical uses. In real applications, the cache size is typically much smaller.

- Potential Infinite Loop in `solve()`: The `solve()` function keeps accepting queries in an infinite loop unless stopped by user input (`-1`). This can lead to infinite queries without an efficient exit strategy in some cases.
- Overuse of `#define`: The use of `#define` to replace types, like `int` with `ll`, reduces readability. Although it is functional, using `typedef` or templates would make the code cleaner and less error-prone.

2. How many breakpoints do you need to fix those errors?

Breakpoints can be strategically placed to catch the most important issues during debugging:

Breakpoints Needed:

1. Memory Allocation Check: Set a breakpoint at dynamic memory allocation points like `Node* newNode = new Node(key, value);` and track if each `new` has a corresponding `delete` during cache evictions. This will ensure there are no memory leaks.

Cache Capacity Overflow: Place a breakpoint in the `put()` function, especially around this block:

```
if(Curr_Size < Capacity)
```

2. This will help track when the cache is full, ensuring the correct node is deleted using the least frequent eviction strategy.

Frequency Count Handling: Set a breakpoint in the frequency manipulation functions, specifically:

`RemoveFromCountOfFreq(FreqOfKey[key]);`

`AddIntoCountOfFreq(FreqOfKey[key]);`

3. This will allow you to verify if the frequency of cache items is being handled correctly during `put()` and `get()` operations.
4. Query Input and Handling: A breakpoint around the query input area (`cin >> WHICH_QUERY`) would help track and ensure correct query handling (i.e., PUT, GET, or stopping).

(a) What are the steps you have taken to fix the error you identified in the code fragment?

Steps Taken to Fix Errors:

1. Remove Redundant Typedefs: Remove the `#define int ll` and any unnecessary type replacements. Use `int` normally and rely on explicit types like `long long` only where needed.

Fix Memory Leaks: Add proper memory management by ensuring that for every `new` allocation (e.g., `new Node(key, value)`), there is a corresponding `delete` when a node is removed from the cache, e.g.: `delete deleteNode;`

2.
3. Input/Output Initialization Cleanup: Since `init_code()` is commented out, either remove it altogether if it is not needed or uncomment it for cases where the code will run in an online judge environment. Ensure the file redirection is handled correctly if needed.

4. Resize Cache Array: Reduce `DEFAULT_SIZE` to a more manageable number, such as `1000`, which aligns with practical LFU cache sizes.

5. Improving Loop Structure in `solve()`: Add a break condition or prompt the user to enter the number of queries upfront to avoid an infinite loop situation.

6. Simplify Code: Consider using modern C++ features (e.g., smart pointers, STL containers) to make the code cleaner, more maintainable, and memory-safe.

Program Inspection:

1) Armstrong Number

```
class Armstrong{
    public static void main(String args[]){
        int num = Integer.parseInt(args[0]);
        int n = num; //use to check at last time
        int check=0, remainder;
        while(num > 0){
            remainder = num / 10;
            check = check + (int)Math.pow(remainder, 3);
            num = num % 10;
        }
        if(check == n)
            System.out.println(n+" is an Armstrong Number");
        else
            System.out.println(n+" is not an Armstrong Number");
    }
}
```

Q1. How many errors are there in the program?

- **Error 1:** Incorrect logic in the calculation of the remainder and the number. In the while loop, the remainder is being calculated as `remainder = num / 10;`, but this should be `remainder = num % 10;` to extract the last digit.
- **Error 2:** The logic to reduce `num` is incorrect. The correct reduction should be `num = num / 10;` instead of `num = num % 10;` to remove the last digit after it's processed.

Total errors identified: 2.

Q2. Which category of program inspection would you find more effective?

- The most effective category for this inspection is **Computation Errors (Category C)**. This code deals with calculations (computing powers and sums), and the errors involve incorrect computation and improper logic.

Q3. Which type of error were you not able to identify using the program inspection?

- There are no **Control-Flow Errors (Category E)** or **Data Reference Errors (Category A)** identified here, but potential user input errors (like an incorrect number format in `args[0]`) could occur. This error is not identified in this inspection because it would need to handle exceptions, such as input validation.

Q4. Is the program inspection technique worth applying?

- Yes, program inspection is very effective in catching logical and computational errors, especially for common mistakes like incorrect calculations in loops and conditions. In this case, the technique helped identify incorrect remainder and digit extraction logic. However, additional error handling for invalid inputs (such as invalid arguments) would improve robustness.

2) GCD LCM

```
import java.util.Scanner;

public class GCD_LCM {
    static int gcd(int x, int y) {
        int r, a, b;
        a = (x > y) ? x : y; // a is greater number
        b = (x < y) ? x : y; // b is smaller number

        while (b != 0) {
            r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    static int lcm(int x, int y) {
        return (x * y) / gcd(x, y);
    }

    public static void main(String args[]) {
        Scanner input = new Scanner(System.in);
        System.out.println("Enter the two numbers: ");
        int x = input.nextInt();
        int y = input.nextInt();

        System.out.println("The GCD of two numbers is: " + gcd(x, y));
        System.out.println("The LCM of two numbers is: " + lcm(x, y));
        input.close();
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: The logic to calculate `gcd()` was incorrect. The variables a and b should be initialized such that a holds the larger number, and b holds the smaller number.
- **Error 2**: The `lcm()` function was missing the logic to calculate the LCM using the formula `LCM = (x * y) / GCD`.

Q2: Which category of program inspection would you find more effective?

- **Static Analysis**: This category of inspection would help identify the logical structure of the program, ensuring that the formulas and conditions are correct. Code walkthroughs could also be useful for catching such logical errors.

Q3: Which type of error you are not able to identify using the program inspection?

- **Runtime Errors**: The logic errors in this code can be identified by static inspection, but performance-related issues or errors related to specific inputs (like large numbers) would require testing.

Q4: Is the program inspection technique worth applicable?

- Yes, it is applicable, as the logical errors (like GCD calculation and LCM formula) could easily be spotted during code walkthroughs or static analysis without needing to run the program.

3) KNAPSACK

```
public class Knapsack {
    public static void main(String[] args) {
        int N = Integer.parseInt(args[0]);
        int W = Integer.parseInt(args[1]);

        int[] profit = new int[N + 1];
        int[] weight = new int[N + 1];

        for (int n = 1; n <= N; n++) {
            profit[n] = (int) (Math.random() * 1000);
            weight[n] = (int) (Math.random() * W);
        }

        int[][] opt = new int[N + 1][W + 1];
        boolean[][] sol = new boolean[N + 1][W + 1];

        for (int n = 1; n <= N; n++) {
            for (int w = 1; w <= W; w++) {
                int option1 = opt[n - 1][w];
                int option2 = (weight[n] <= w) ? profit[n] + opt[n - 1][w - weight[n]]
: Integer.MIN_VALUE;

                opt[n][w] = Math.max(option1, option2);
                sol[n][w] = (option2 > option1);
            }
        }

        boolean[] take = new boolean[N + 1];
        for (int n = N, w = W; n > 0; n--) {
            if (sol[n][w]) {
                take[n] = true;
                w -= weight[n];
            } else {
                take[n] = false;
            }
        }

        System.out.println("Item\tProfit\tWeight\tTake");
        for (int n = 1; n <= N; n++) {
            System.out.println(n + "\t" + profit[n] + "\t" + weight[n] + "\t" +
take[n]);
```

```
        }
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: The logic for summing the digits and reducing the number was incorrect. The inner while loop was not properly reducing the number and summing the digits.

Q2: Which category of program inspection would you find more effective?

- **Static Code Review**: This type of inspection would help catch the simple logical error related to summing the digits, and a proper code walkthrough would ensure the correctness of the magic number algorithm.

Q3: Which type of error you are not able to identify using the program inspection?

- **None**: The error here is purely logical, and static inspection would catch it.

Q4: Is the program inspection technique worth applicable?

- Yes, the program inspection is highly effective, as the error is related to basic logic and could be caught without running the program.

4) Merge Sort

```
import java.util.Arrays;

public class MergeSort {
    public static void main(String[] args) {
        int[] list = {14, 32, 67, 76, 23, 41, 58, 85};
        System.out.println("before: " + Arrays.toString(list));
        mergeSort(list);
        System.out.println("after:  " + Arrays.toString(list));
    }

    public static void mergeSort(int[] array) {
        if (array.length > 1) {
            int[] left = leftHalf(array);
            int[] right = rightHalf(array);

            mergeSort(left);
            mergeSort(right);

            merge(array, left, right);
        }
    }

    public static int[] leftHalf(int[] array) {
        int size1 = array.length / 2;
        int[] left = new int[size1];
        for (int i = 0; i < size1; i++) {
            left[i] = array[i];
        }
        return left;
    }

    public static int[] rightHalf(int[] array) {
        int size1 = array.length / 2;
        int size2 = array.length - size1;
        int[] right = new int[size2];
        for (int i = 0; i < size2; i++) {
            right[i] = array[i + size1];
        }
        return right;
    }

    public static void merge(int[] result, int[] left, int[] right) {
        int i1 = 0, i2 = 0;
        for (int i = 0; i < result.length; i++) {
            if (i2 >= right.length || (i1 < left.length && left[i1] <= right[i2])) {
                result[i] = left[i1];
                i1++;
            } else {
                result[i] = right[i2];
                i2++;
            }
        }
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1:** The logic in the `merge` function and the `leftHalf` and `rightHalf` functions had issues with indexing and array bounds.
- **Error 2**: The array bounds were not correctly managed during merging, which could cause an `ArrayIndexOutOfBoundsException`.

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough and Static Analysis**: Given that merge sort is recursive and involves array manipulation, a careful code walkthrough is necessary to identify logical issues and array handling errors.

Q3: Which type of error you are not able to identify using the program inspection?

- **Performance Issues**: The code might perform poorly with certain data, such as very large arrays, which is not easily noticeable without testing.

Q4: Is the program inspection technique worth applicable?

- Yes, especially for checking the correctness of recursion and array manipulation. Most of the errors here are structural, so static analysis would catch them.

5) Matrix multiplication

```
import java.util.Scanner;

class MatrixMultiplication {
   public static void main(String args[]) {
      int m, n, p, q, sum = 0;

      Scanner in = new Scanner(System.in);
      System.out.println("Enter the number of rows and columns of first
matrix");
      m = in.nextInt();
      n = in.nextInt();
```

```
        int first[][] = new int[m][n];
        System.out.println("Enter the elements of first matrix");

        for (int c = 0; c < m; c++) {
            for (int d = 0; d < n; d++) {
                first[c][d] = in.nextInt();
            }
        }

        System.out.println("Enter the number of rows and columns of second
matrix");
        p = in.nextInt();
        q = in.nextInt();

        if (n != p) {
            System.out.println("Matrices with entered orders can't be multiplied
with each other.");
        } else {
            int second[][] = new int[p][q];
            int multiply[][] = new int[m][q];

            System.out.println("Enter the elements of second matrix");

            for (int c = 0; c < p; c++) {
                for (int d = 0; d < q; d++) {
                    second[c][d] = in.nextInt();
                }
            }

            for (int c = 0; c < m; c++) {
                for (int d = 0; d < q; d++) {
                    for (int k = 0; k < n; k++) {
                        sum += first[c][k] * second[k][d];
                    }
                    multiply[c][d] = sum;
                    sum = 0;
                }
            }

            System.out.println("Product of entered matrices:");
            for (int c = 0; c < m; c++) {
                for (int d = 0; d < q; d++) {
                    System.out.print(multiply[c][d] + "\t");
                }
                System.out.println();
            }
        }
        in.close();
      }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1:** Indexing issues while performing matrix multiplication. The loops used for calculating the product were not accessing the arrays correctly.
- **Error 2**: A logic flaw when handling matrices that do not meet the multiplication condition.

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough:** Since matrix multiplication involves nested loops and multiple indices, a walkthrough focusing on array access and index manipulation is necessary to catch the issues.

Q3: Which type of error you are not able to identify using the program inspection?

- **Efficiency Issues:** Large matrix inputs could cause performance problems, but static inspection would not reveal this directly.

Q4: Is the program inspection technique worth applicable?

- Yes, inspecting the code for matrix operations is important since errors in array bounds or loop handling can lead to incorrect results or crashes.

6) Quadratic Probing

```
import java.util.Scanner;

class QuadraticProbingHashTable {
    private int currentSize, maxSize;
    private String[] keys;
    private String[] vals;

    public QuadraticProbingHashTable(int capacity) {
        currentSize = 0;
        maxSize = capacity;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }

    public void makeEmpty() {
        currentSize = 0;
        keys = new String[maxSize];
        vals = new String[maxSize];
    }
```

```
public int getSize() {
    return currentSize;
}

public boolean isFull() {
    return currentSize == maxSize;
}

public boolean isEmpty() {
    return getSize() == 0;
}

public boolean contains(String key) {
    return get(key) != null;
}

private int hash(String key) {
    return Math.abs(key.hashCode() % maxSize);
}

public void insert(String key, String val) {
    int tmp = hash(key);
    int i = tmp, h = 1;

    do {
        if (keys[i] == null) {
            keys[i] = key;
            vals[i] = val;
            currentSize++;
            return;
        }

        if (keys[i].equals(key)) {
            vals[i] = val;
            return;
        }

        i = (i + h * h++) % maxSize;
    } while (i != tmp);
}

public String get(String key) {
    int i = hash(key);
    int h = 1;

    while (keys[i] != null) {
        if (keys[i].equals(key)) {
            return vals[i];
        }

        i = (i + h * h++) % maxSize;
    }

    return null;
}

public void remove(String key) {
    if (!contains(key)) {
        return;
    }

    int i = hash(key), h = 1;
    while (!key.equals(keys[i])) {
        i = (i + h * h++) % maxSize;
    }

    keys[i] = vals[i] = null;
```

```
        for (i = (i + h * h) % maxSize; keys[i] != null; i = (i + h * h++) %
maxSize) {
            String tmpKey = keys[i], tmpVal = vals[i];
            keys[i] = vals[i] = null;
            currentSize--;
            insert(tmpKey, tmpVal);
        }
        currentSize--;
    }

    public void printHashTable() {
        System.out.println("\nHash Table:");
        for (int i = 0; i < maxSize; i++) {
            if (keys[i] != null) {
                System.out.println(keys[i] + " " + vals[i]);
            }
        }
    }

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        System.out.println("Enter size of hash table:");
        QuadraticProbingHashTable hashTable = new
QuadraticProbingHashTable(in.nextInt());

        char choice;
        do {
            System.out.println("\nHash Table Operations:");
            System.out.println("1. Insert");
            System.out.println("2. Remove");
            System.out.println("3. Get");
            System.out.println("4. Check empty");
            System.out.println("5. Clear");

            int ch = in.nextInt();
            switch (ch) {
                case 1:
                    System.out.println("Enter key and value");
                    hashTable.insert(in.next(), in.next());
                    break;
                case 2:
                    System.out.println("Enter key");
                    hashTable.remove(in.next());
                    break;
                case 3:
                    System.out.println("Enter key");
                    System.out.println("Value: " + hashTable.get(in.next()));
                    break;
                case 4:
                    System.out.println("Empty Status: " + hashTable.isEmpty());
                    break;
                case 5:
                    hashTable.makeEmpty();
                    System.out.println("Hash Table Cleared");
                    break;
                default:
                    System.out.println("Wrong Entry");
                    break;
            }

            hashTable.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n)?");
            choice = in.next().charAt(0);
        } while (choice == 'Y' || choice == 'y');
        in.close();
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: Incorrect logic during hash table resizing and re-inserting items after a removal.
- **Error 2**: The `remove()` function could leave the table in an inconsistent state when reinserting elements.

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough**: Given that this program implements a more complex data structure (hash table with quadratic probing), careful inspection of the insert, remove, and rehashing mechanisms is crucial.

Q3: Which type of error you are not able to identify using the program inspection?

- **Edge Case Handling**: Certain edge cases, like rehashing when the table is almost full, might not be identified until runtime.

Q4: Is the program inspection technique worth applicable?

- Yes, since the errors here are primarily related to logic in data structure management, program inspection through static analysis or code review would be very effective.

6) Sorting

```
class MatrixMultiplication {
    public static void main(String args[]) {
        int m, n, p, q, sum = 0;

        Scanner in = new Scanner(System.in);
        System.out.println("Enter the number of rows and columns of first
matrix");
        m = in.nextInt();
        n = in.nextInt();

        int first[][] = new int[m][n];
        System.out.println("Enter the elements of first matrix");

        for (int c = 0; c < m; c++) {
            for (int d = 0; d < n; d++) {
                first[c][d] = in.nextInt();
            }
        }

        System.out.println("Enter the number of rows and columns of second
matrix");
        p = in.nextInt();
        q = in.nextInt();

        if (n != p) {
            System.out.println("Matrices with entered orders can't be multiplied
with each other.");
```

```
} else {
    int second[][] = new int[p][q];
    int multiply[][] = new int[m][q];

    System.out.println("Enter the elements of second matrix");

    for (int c = 0; c < p; c++) {
        for (int d = 0; d < q; d++) {
            second[c][d] = in.nextInt();
        }
    }

    for (int c = 0; c < m; c++) {
        for (int d = 0; d < q; d++) {
            for (int k = 0; k < n; k++) {
                sum += first[c][k] * second[k][d];
            }
            multiply[c][d] = sum;
            sum = 0;
        }
    }

    System.out.println("Product of entered matrices:");
    for (int c = 0; c < m; c++) {
        for (int d = 0; d < q; d++) {
            System.out.print(multiply[c][d] + "\t");
        }
        System.out.println();
    }
```

```
        }
        in.close();
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: The class name has a space, `Ascending _Order`, which is invalid.
- **Error 2**: In the sorting loop, the condition `for (int i = 0; i >= n; i++);` should be `for (int i = 0; i < n; i++)`. The condition was incorrect and the extra semicolon was terminating the loop prematurely.
- **Error 3**: The comparison inside the loop should be `if (a[i] > a[j])` instead of `if (a[i] <= a[j])`, as the goal is to sort in ascending order.

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough**: The issues in this program are basic syntax and logical errors that could easily be caught during a simple code walkthrough.

Q3: Which type of error you are not able to identify using the program inspection?

- **Performance or Edge Cases**: Issues such as handling very large arrays (performance) or empty arrays might not be detected just by inspection.

Q4: Is the program inspection technique worth applicable?

- Yes, this technique helps identify basic logic and syntax errors, which can be corrected without running the program.

7) Stack Implementation

```
public class StackMethods {
    private int top;
    int size;
    int[] stack;

    public StackMethods(int arraySize){
        size = arraySize;
        stack = new int[size];
        top = -1;
    }

    public void push(int value){
        if(top == size - 1){
            System.out.println("Stack is full, can't push a value");
        } else {
            top++;  // Increment top to push the value onto the stack
            stack[top] = value;
        }
    }
    public void pop(){
        if(!isEmpty())
            top--;  // Decrement top to pop an item
        else {
            System.out.println("Can't pop...stack is empty");
        }
    }

    public boolean isEmpty(){
        return top == -1;
    }

    public void display(){
        for(int i = 0; i <= top; i++){  // Corrected loop condition
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}

public class StackReviseDemo {
    public static void main(String[] args) {
        StackMethods newStack = new StackMethods(5);
        newStack.push(10);
        newStack.push(1);
        newStack.push(50);
        newStack.push(20);
        newStack.push(90);
        newStack.display();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.pop();
        newStack.display();
    }
}
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: In the `push()` method, the line `top--` should be `top++`. The stack is being pushed down instead of growing.
- **Error 2**: In the `display()` method, the loop condition `for(int i=0; i>top; i++)` should be `for(int i=0; i<=top; i++)`, as the loop condition is incorrect.
- **Error 3**: The stack display logic is incorrect, leading to wrong output.

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough**: A detailed code walkthrough would have caught the incorrect usage of the stack pointer `top` and the improper display loop.

Q3: Which type of error you are not able to identify using the program inspection?

- **Performance Issues**: Errors related to memory handling or large-scale usage of the stack (e.g., stack overflow) might not be identified without runtime testing.

Q4: Is the program inspection technique worth applicable?

- Yes, stack operations are relatively simple to inspect. The errors are mostly logical and would be caught during an inspection.

7) Tower of hanoi

```
//Tower of Hanoi
public class MainClass {
  public static void main(String[] args) {
    int nDisks = 3;
    doTowers(nDisks, 'A', 'B', 'C');
  }
  public static void doTowers(int topN, char from,
  char inter, char to) {
    if (topN == 1){
      System.out.println("Disk 1 from "
      + from + " to " + to);
    }else {
      doTowers(topN - 1, from, to, inter);
      System.out.println("Disk "
      + topN + " from " + from + " to " + to);
      doTowers(topN ++, inter--, from+1, to+1)
    }
  }
}
```

```
Output: Disk 1 from A to C
        Disk 2 from A to B
        Disk 1 from C to B
        Disk 3 from A to C
        Disk 1 from B to A
        Disk 2 from B to C
        Disk 1 from A to C
```

Q1: How many errors are there in the program? Mention the errors you have identified.

- **Error 1**: The line `doTowers(topN ++, inter--, from+1, to+1)` contains incorrect increment/decrement operations. It should use proper recursive arguments.
- **Error 2**: The recursive call syntax is incorrect and needs correction.
-

Q2: Which category of program inspection would you find more effective?

- **Code Walkthrough**: A recursive algorithm like Tower of Hanoi requires a proper code walkthrough to trace recursive calls and their arguments.

Q3: Which type of error you are not able to identify using the program inspection?

- **Stack Overflow**: If recursion depth becomes too large, runtime stack overflow might occur, which is not identifiable through static inspection.

Q4: Is the program inspection technique worth applicable?

- Yes, it is effective in identifying incorrect recursive logic, as seen with the argument passing in the recursive calls.

II) Code Debugging

Only written here are break points as errors and executable code is written above.

1) **Armstrong :**
    - Errors:
- Incorrect calculation of `remainder`:
    - The calculation of `remainder` should be `num % 10` to get the last digit.
- Incorrect calculation logic:
    - You should update `num` to remove the last digit after computing the `remainder` (i.e., `num = num / 10`).
- Using a capital 'A' in "Armstrong":

- It should be "Armstrong" consistently in the output message.

Breakpoints:

1. Breakpoint 1: At the beginning of the `main` method to verify the input value of `num`.
2. Breakpoint 2: Inside the while loop, right after calculating `remainder`, to check its value.
3. Breakpoint 3: After updating the `check` variable to ensure it's accumulating correctly.
4. Breakpoint 4: Before the final conditional check to see the values of `check` and `n`.

-

2) GCD and LCM Calculation

Errors:

Incorrect condition in GCD loop:

- The while loop condition is incorrectly set as `while(a % b == 0)` which should be `while(a % b != 0)`.

Logic for determining `a` and `b`:

- `a` should be assigned as the maximum of `x` and `y`, and `b` should be assigned as the minimum. The current assignment is incorrect.

LCM logic:

- In the `lcm` method, the condition should be corrected to check if `a` is divisible by `x` or `y` to ensure the LCM is computed properly.

Breakpoints:

- Breakpoint 1: Set at the beginning of the `gcd` method to verify values of `x` and `y`.
- Breakpoint 2: Inside the `gcd` while loop to see the values of `a`, `b`, and `r`.
- Breakpoint 3: Inside the `lcm` method to check the increment of `a`.

---

3. Knapsack Problem

Errors:

1. Incorrect increment in `option1` assignment:
   - `int option1 = opt[n++][w];` should be `int option1 = opt[n][w];` as it incorrectly modifies `n`.
2. Invalid indices in option2 calculation:
   - In the `option2` assignment, it uses `profit[n-2]` which should be `profit[n]`.

Breakpoints:

- Breakpoint 1: Inside the outer loop to check values of `n` and `w`.
- Breakpoint 2: Before assigning `option1` and `option2` to verify their calculations.
- Breakpoint 3: After filling the `opt` table to verify its correctness.

---

4. Magic Number Check

Errors:

1. Incorrect initialization of `sum`:
   - `sum=num;` should be initialized to zero before being used to accumulate digits.
2. Incorrect loop condition:

  - The inner while loop uses `while(sum==0)` which should be `while(sum!=0)`.
3. Incorrect multiplication logic in inner loop:
    - `s=s*(sum/10);` should be `s=s+(sum%10);` to correctly sum the digits.
4. Missing semicolon in `sum=sum%10`:
    - A semicolon is needed at the end of this line.

Breakpoints:

- Breakpoint 1: At the start of the `main` method to verify user input.
- Breakpoint 2: Before the inner while loop to check values of `sum` and `s`.
- Breakpoint 3: At the end of the method to check the final output.

---

5. Merge Sort

Errors:

1. Incorrect array slicing:
    - `int[] left = leftHalf(array + 1);` should be `int[] left = leftHalf(Arrays.copyOfRange(array, 0, mid));`.
    - Similarly for the right half.
2. Incorrect merge call:
    - The `merge` function should not increment arrays with ++, it should be `merge(array, left, right)`.

Breakpoints:

- Breakpoint 1: At the start of the `mergeSort` method to check initial array state.
- Breakpoint 2: After splitting the array into left and right to verify correctness.

- Breakpoint 3: Before the merge to inspect `left` and `right`.

---

6. Matrix Multiplication

Errors:

1. Incorrect indexing for matrix multiplication:
   - The index should be `first[c][k]` and `second[k][d]`, not `first[c-1][c-k]` and `second[k-1][k-d]`.
2. Redundant printing of dimensions:
   - The output statements should be aligned correctly to make sense when matrices cannot be multiplied.

Breakpoints:

- Breakpoint 1: After reading matrix dimensions to confirm correct values.
- Breakpoint 2: Inside the nested loops for matrix multiplication to verify the values being multiplied.
- Breakpoint 3: Before outputting the resulting matrix.

---

7. Quadratic Probing Hash Table

Errors:

1. Syntax Error in the `insert` method:
   - `i + = (i + h / h--) % maxSize;` should be corrected to `i += (h * h) % maxSize;`.
2. Error in the `get` method:
   - The indexing should be corrected in `i = (i + h * h++) % maxSize;` to not mutate h.
3. Main method output text:

  - The prompt text in the main method needs slight rephrasing for clarity.

Breakpoints:

- Breakpoint 1: In the `insert` method to check if key/value is being inserted correctly.
- Breakpoint 2: In the `get` method to verify hash codes and results.
- Breakpoint 3: At the end of the main loop to confirm the current state of the hash table.

8. Sorting Array in Ascending Order

Breakpoints Needed: 2

- Breakpoint 1: After reading the input into the array, to check if the array is filled correctly.
- Breakpoint 2: Inside the nested loop for sorting, to ensure the logic is working as expected after each comparison and swap.

9. Stack Implementation in Java

Breakpoints Needed: 4

- Breakpoint 1: In the `push` method before decrementing `top` to check if `top` is correctly updated before pushing values.
- Breakpoint 2: After the line `stack[top] = value` in `push` to ensure values are being pushed correctly.
- Breakpoint 3: In the `pop` method to verify that the top is incremented correctly and to detect stack underflow.
- Breakpoint 4: In the `display` method to confirm the stack is displayed correctly by fixing the loop condition from `for (int i = 0; i > top; i++)` to `for (int i = 0; i <= top; i++)`.

10. Tower of Hanoi

Breakpoints Needed: 2

- Breakpoint 1: In the recursive call before the `doTowers(topN - 1, from, to, inter)` to check if the recursive function is handling the disks correctly.
- Breakpoint 2: After the recursive calls to ensure the correct movements are printed in the desired order.