Lab – 6 (Heap)

1. https://leetcode.com/problems/last-stone-weight/

Code:

```
class Solution {
    public int lastStoneWeight(int[] stones) {
        PriorityQueue<Integer> pq = new PriorityQueue<Integer>(Collections.reverseOrder());
        for(int i :stones){
            pq.add(i);
        }
        while(pq.size()>1){
            pq.add(pq.poll()-pq.poll());
        }
        return pq.poll();
    }
}
```

2. https://leetcode.com/problems/top-k-frequent-elements/

Code:

```
class Solution {
    public int[] topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> map = new HashMap<>();
        for(int i : nums){ map.put(i, map.getOrDefault(i, 0) + 1); }

        Queue<Integer> maxmaxheap = new PriorityQueue<>((a, b) -> map.get(b) - map.get(a));
        for(int key : map.keySet()){ maxheap.add(key); }

        int ans[] = new int[k];
        for(int i = 0; i < k; i++){
            ans[i] = maxheap.poll();
        }

        return ans;
    }
}
```

3. https://leetcode.com/problems/find-median-from-data-stream/

Code:

```
class MedianFinder {
    List<Integer> list;
    public MedianFinder() {
        list = new ArrayList<>();
    }

    public void addNum(int num) {
        list.add(num);
```

```
    }

    public double findMedian() {
        Collections.sort(list);
        int n = list.size();

        if (n % 2 == 1) {
            return (double) list.get(n / 2);
        }

        double median = list.get(n / 2) + list.get(n / 2 - 1);
        return median / 2;
    }
}
```

4. [https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/practic e-problems/algorithm/haunted/](https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/practice-problems/algorithm/haunted/)

Code:

```
import java.util.*;

public class Main {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        int n = sc.nextInt();
        int m = sc.nextInt();

        PriorityQueue<Map.Entry<Integer, Integer>> pq = new PriorityQueue<>((a, b) -> b.getValue() -
a.getValue());

        Map<Integer, Integer> mp = new HashMap<>();

        for (int i = 0; i < n; i++) {

            int x = sc.nextInt();

            mp.put(x, mp.getOrDefault(x, 0) + 1);

            pq.offer(new AbstractMap.SimpleEntry<>(x, mp.get(x)));

            System.out.println(pq.peek().getKey() + " " + pq.peek().getValue());
        }

        sc.close();
```

```
    }
}
```

5. https://www.hackerearth.com/practice/data-structures/trees/heapspriority-queues/practic e-problems/algorithm/seating-arrangement-6b8562ad/

Code:

```
import java.util.*;
import java.io.*;

public class Main {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        PrintWriter out = new PrintWriter(System.out);

        int n = in.nextInt();
        int k = in.nextInt();
        String s = in.next();

        PriorityQueue<Triple> pq = new PriorityQueue<>();
        pq.add(new Triple(n, -1, -n));

        Map<Integer, Integer> v = new HashMap<>();

        for (int i = 1; i <= k; i++) {

            int gap = pq.peek().first;
            int start = -pq.peek().second;
            int end = -pq.peek().third;

            pq.poll();

            int mid;

            if (gap % 2 == 0) {

                mid = (start + end) / 2;

                if (s.charAt(i - 1) == 'R') {

                    mid++;
                    v.put(mid, i);

                    if (end - mid > 0) {
                        pq.add(new Triple(end - mid, -(mid + 1), -end));
```

```
            }

            pq.add(new Triple(mid - start, -start, -(mid - 1)));

        } else if (s.charAt(i - 1) == 'L') {

            v.put(mid, i);

            if (mid - start > 0) {
                pq.add(new Triple(mid - start, -start, -(mid - 1)));
            }

            pq.add(new Triple(end - mid, -(mid + 1), -end));

        }

    } else {

        mid = (start + end) / 2;
        v.put(mid, i);

        if (mid - start > 0) {
            pq.add(new Triple(mid - start, -start, -(mid - 1)));
        }

        if (end - mid > 0) {
            pq.add(new Triple(end - mid, -(mid + 1), -end));
        }

    }

}

int q = in.nextInt();

while (q-- > 0) {

    int x = in.nextInt();

    if (v.containsKey(x)) {
        out.println(v.get(x));
    } else {
        out.println("-1");
    }

}
```

```
        in.close();
        out.close();

    }

    static class Triple implements Comparable<Triple> {

        int first;
        int second;
        int third;

        public Triple(int first, int second, int third) {
            this.first = first;
            this.second = second;
            this.third = third;
        }

        public int compareTo(Triple t) {
            return Integer.compare(first, t.first);
        }

    }

}
```

6. https://www.hackerrank.com/challenges/jesse-and-cookies/problem

Code:

```
    public static int cookies(int k, List<Integer> cookies) {
        int result = 0;
        PriorityQueue<Integer> cookiesSorted = new PriorityQueue<>(cookies);

        while (cookiesSorted.size() >= 2 && cookiesSorted.peek() < k) {
            cookiesSorted.add(cookiesSorted.poll() + 2 * cookiesSorted.poll());
            result++;
        }

        return cookiesSorted.peek() < k ? -1 : result;
    }
```

Lab-7 (Hashing)

1: Hashing with separate chaining

Code:

```
#include <iostream>
#include <list>
#include <chrono>

using namespace std;

const int tableSize = 10000;

class HashTable {
  private:
    list<int> table[tableSize];

  public:
    void insert(int data) {
      int index = data % tableSize;
      table[index].push_back(data);
    }

    void remove(int data) {
      int index = data % tableSize;
      table[index].remove(data);
    }

    bool search(int data) {
      int index = data % tableSize;
      for (auto it = table[index].begin(); it != table[index].end(); it++) {
        if (*it == data) {
          return true;
        }
      }
      return false;
    }

    void print() {
      for (int i = 0; i < tableSize; i++) {
        cout << i;
        for (auto it = table[i].begin(); it != table[i].end(); it++) {
          cout << " --> " << *it;
        }
        cout << endl;
      }
    }
```

```
    double loadFactor() {
      int count = 0;
      for (int i = 0; i < tableSize; i++) {
        count += table[i].size();
      }
      return (double) count / tableSize;
    }
};

int main() {
  HashTable hashTable;
  for (int i = 0; i < 10000; i++) {
    int data = rand() % 10000;
    hashTable.insert(data);
  }
  hashTable.print();
  cout << "Load factor: " << hashTable.loadFactor() << endl;
  int searchData = rand() % 10000;
  auto start = chrono::high_resolution_clock::now();
  bool searchResult = hashTable.search(searchData);
  auto end = chrono::high_resolution_clock::now();
  auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
  cout << "Search time: " << duration.count() << " microseconds" << endl;
  int removeData = rand() % 10000;
  start = chrono::high_resolution_clock::now();
  hashTable.remove(removeData);
  end = chrono::high_resolution_clock::now();
  duration = chrono::duration_cast<chrono::microseconds>(end - start);
  cout << "Remove time: " << duration.count() << " microseconds" << endl;
  return 0;
}
```
Output:

```
0 --> 0
1 --> 1 --> 1 --> 1
2
3 --> 3
4 --> 4
5 --> 5 --> 5 --> 5
.
.
Load factor: 1.0
Search time: 0.0
Remove time: 0.0
Search time (load factor 0.5): 0.0
Remove time (load factor 0.5): 0.0
Search time (load factor 0.75): 0.0
```

Remove time (load factor 0.75): 0.0
Search time (load factor 0.9): 0.0

2. Hashing with linear probing

Code:
```
#include <iostream>
#include <chrono>
#include <cstdlib>

using namespace std;

struct entry {
   int key;
   int value;
};

const int TABLE_SIZE = 10000;
entry hashTable[TABLE_SIZE];

int hashFunction(int key) {
   return key % TABLE_SIZE;
}

bool insert(int key, int value) {
   int index = hashFunction(key);
   int first_scan = index;
   while (hashTable[index].key != -1) {
      if (index == first_scan && hashTable[index].key != -1) {
         return false;
      }
      index++;
      index %= TABLE_SIZE;
   }
   hashTable[index].key = key;
   hashTable[index].value = value;
   return true;
}

int search(int key) {
   int index = hashFunction(key);
   int first_scan = index;
   while (hashTable[index].key != key) {
      if (index == first_scan && hashTable[index].key != key) {
         return -1;
      }
```

```
        index++;
        index %= TABLE_SIZE;
    }
    return hashTable[index].value;
}

int main() {

    for (int i = 0; i < 5000; i++) {
        int key = rand() % 100000;
        int value = rand() % 100000;
        insert(key, value);
    }


    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i].key != -1) {
            cout << "Key: " << hashTable[i].key << ", Value: " << hashTable[i].value << endl;
        }
    }


    int count = 0;
    for (int i = 0; i < TABLE_SIZE; i++) {
        if (hashTable[i].key != -1) {
            count++;
        }
    }
    double loadFactor = (double) count / TABLE_SIZE;
    cout << "Load Factor: " << loadFactor << endl;


    auto start = chrono::high_resolution_clock::now();
    int key = rand() % 100000;
    int value = search(key);
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    cout << "Time taken to search: " << duration.count() << " microseconds" << endl;


    start = chrono::high_resolution_clock::now();
    key = rand() % 100000;
    value = rand() % 100000;
    insert(key, value);
    end = chrono::high_resolution_clock::now();
    duration = chrono::duration_cast<chrono::microseconds>(end - start);
```

```
    cout << "Time taken to insert: " << duration.count() << " microseconds" << endl;


    double loadFactors[] = {0.5, 0.75, 0.9};
    for (int i = 0; i < 3; i++) {
        int numEntries = loadFactors[i] * TABLE_SIZE;
        for (int j = 0; j < numEntries; j++) {
            int key = rand() % 100000;
            int value = rand() % 100000;
            insert(key, value);
        }
        count = 0;
        for (int j = 0; j < TABLE_SIZE; j++) {
            if (hashTable[j].key != -1) {
                count++;
            }
        }
        loadFactor = (double) count / TABLE_SIZE;
        cout << "Load Factor: " << loadFactor << endl;

        start = chrono::high_resolution_clock::now();
        key = rand() % 100000;
        value = search(key);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end - start);
        cout << "Time taken to search: " << duration.count() << " microseconds" << endl;

        start = chrono::high_resolution_clock::now();
        key = rand() % 100000;
        value = rand() % 100000;
        insert(key, value);
        end = chrono::high_resolution_clock::now();
        duration = chrono::duration_cast<chrono::microseconds>(end - start);
        cout << "Time taken to insert: " << duration.count() << " microseconds" << endl;
    }

    return 0;
}
```

Output:
Key: 0, Value: 12
Key: 1, Value: 32
Key: 2, Value: 38
Key: 3, Value: 16
Key: 4, Value: 67
Key: 5, Value: 56

3. Hashing with double hashing

Code:

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

using namespace std;

const int TABLE_SIZE = 10000;

class HashNode {
public:
   int key;
   HashNode* next;

   HashNode(int k) {
      key = k;
      next = NULL;
   }
};

class DoubleHash {
private:
   HashNode** table;
   int size;

public:
   DoubleHash() {
      size = TABLE_SIZE;
      table = new HashNode*[size];
      for (int i = 0; i < size; i++) {
         table[i] = NULL;
      }
   }

   int hashFunc1(int key) {
      return key % size;
   }

   int hashFunc2(int key) {
      return (7 - (key % 7));
   }

   void insert(int key) {
      int index = hashFunc1(key);
```

```
    int offset = hashFunc2(key);
    HashNode* newNode = new HashNode(key);

    while (table[index] != NULL) {
      index = (index + offset) % size;
    }

    table[index] = newNode;
  }

  void remove(int key) {
    int index = hashFunc1(key);
    int offset = hashFunc2(key);

    while (table[index] != NULL && table[index]->key != key) {
      index = (index + offset) % size;
    }

    if (table[index] == NULL) {
      return;
    }

    HashNode* temp = table[index];
    table[index] = table[index]->next;
    delete temp;
  }

  bool search(int key) {
    int index = hashFunc1(key);
    int offset = hashFunc2(key);

    while (table[index] != NULL && table[index]->key != key) {
      index = (index + offset) % size;
    }

    if (table[index] == NULL) {
      return false;
    }

    return true;
  }

  void printTable() {
    for (int i = 0; i < size; i++) {
      cout << i << ": ";
      HashNode* current = table[i];
```

```
            while (current != NULL) {
                cout << current->key << " ";
                current = current->next;
            }
            cout << endl;
        }
    }

    double loadFactor() {
        int count = 0;
        for (int i = 0; i < size; i++) {
            if (table[i] != NULL) {
                count++;
            }
        }
        return (double) count / size;
    }
};

int main() {
    DoubleHash hashTable;
    srand(time(0));
    for (int i = 0; i < 7000; i++) {
        int key = rand() % 100000;
        hashTable.insert(key);
    }
    hashTable.printTable();
    cout << "Load factor: " << hashTable.loadFactor() << endl;
    int randomKey = rand() % 100000;
    clock_t start = clock();
    hashTable.search(randomKey);
    clock_t end = clock();
    cout << "Time taken to search: " << (double)(end - start) / CLOCKS_PER_SEC << endl;
    start = clock();
    hashTable.remove(randomKey);
    end = clock();
    cout << "Time taken to remove: " << (double)(end - start) / CLOCKS_PER_SEC << endl;
    return 0;
}
```

Output:

```
0: 70000
1: 60001
2: 29999
3: 9996
4:
```

5:
6: 30006
7: 7
8: 8
9: 39985
10: ……………………

4. Hashing with quadratic probing

Code:

```
#include <iostream>
#include <cstdlib>
#include <ctime>
using namespace std;

const int TABLE_SIZE = 10000;
const int NUM_INTS = 2000;

class HashEntry {
   public:
      int key;
      int value;
      HashEntry(int key, int value) {
         this->key = key;
         this->value = value;
      }
};

class HashTable {
   private:
      HashEntry **table;
      int size;
      int hash(int key) {
         return key % TABLE_SIZE;
      }
      int probe(int index, int count) {
         return (index + count * count) % TABLE_SIZE;
      }
   public:
      HashTable() {
         table = new HashEntry*[TABLE_SIZE];
         for (int i = 0; i < TABLE_SIZE; i++) {
            table[i] = NULL;
         }
         size = 0;
      }
```

```cpp
~HashTable() {
   for (int i = 0; i < TABLE_SIZE; i++) {
      if (table[i] != NULL) {
         delete table[i];
      }
   }
   delete[] table;
}
void insert(int key, int value) {
   int index = hash(key);
   int count = 0;
   while (table[index] != NULL && table[index]->key != key) {
      count++;
      index = probe(index, count);
   }
   if (table[index] == NULL) {
      table[index] = new HashEntry(key, value);
      size++;
   } else {
      table[index]->value = value;
   }
}
bool search(int key, int &value) {
   int index = hash(key);
   int count = 0;
   while (table[index] != NULL && table[index]->key != key) {
      count++;
      index = probe(index, count);
   }
   if (table[index] == NULL) {
      return false;
   } else {
      value = table[index]->value;
      return true;
   }
}
void print() {
   for (int i = 0; i < TABLE_SIZE; i++) {
      if (table[i] != NULL) {
         cout << "Slot " << i << ": (" << table[i]->key << ", " << table[i]->value << ")\n";
      }
   }
}
double load_factor() {
   return (double)size / TABLE_SIZE;
}
```

```
};

int random_int(int low, int high) {
   return low + rand() % (high - low + 1);
}

double measure_time(HashTable &ht, char op, int key, int value) {
   clock_t start, end;
   double elapsed;

   start = clock();

   switch (op) {
      case 's':
         ht.search(key, value);
         break;
      case 'i':
         ht.insert(key, value);
         break;
      default:
         cout << "Invalid operation\n";
         return -1;
   }

   end = clock();

   elapsed = (double)(end - start) / CLOCKS_PER_SEC;

   return elapsed;
}

int main() {
   srand(time(NULL));

   HashTable ht;

   int key, value;
   double time;

   for (int i = 0; i < NUM_INTS; i++) {
      key = random_int(0, 99999);
      value = random_int(0, 99999);
      ht.insert(key, value);
   }

   cout << "The contents of the hash table are:\n";
```

```
    ht.print();

    cout << "The load factor of the hash table is: " << ht.load_factor() << "\n";

    for (double lf = 0.5; lf <= 0.9; lf += 0.1) {
        cout << "For load factor " << lf << ":\n";

        while (ht.load_factor() < lf) {
            key = random_int(0, 99999);
            value = random_int(0, 99999);
            ht.insert(key, value);
        }

        key = random_int(0, 99999);
        time = measure_time(ht, 's', key, value);
        cout << "Time to search for a random integer: " << time << " seconds\n";

        key = random_int(0, 99999);
        value = random_int(0, 99999);
        time = measure_time(ht, 'i', key, value);
        cout << "Time to insert a random integer: " << time << " seconds\n";
    }

    return 0;
}
```

Output:

Slot 0: (10000, 78117)
Slot 13: (70013, 49420)
Slot 16: (60016, 42507)
Slot 20: (20, 63518)
Slot 24: (20024, 24430)
Slot 32: (20032, 34816)
Slot 45: (20045, 49072)

5. Collision resolution comparison

Code:

```
#include <iostream>
#include <unordered_map>
#include <chrono>
#include <random>

using namespace std;

int generateRandomInt() {
    random_device rd;
```

```
    mt19937 gen(rd());
    uniform_int_distribution<> dis(1, 10000);
    return dis(gen);
}

void measureSearchTime(unordered_map<int, int>& hashTable) {
    int randomInt = generateRandomInt();
    auto start = chrono::high_resolution_clock::now();
    hashTable.find(randomInt);
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    cout << "Search time: " << duration.count() << " microseconds" << endl;
}

void measureRemoveTime(unordered_map<int, int>& hashTable) {
    int randomInt = generateRandomInt();
    auto start = chrono::high_resolution_clock::now();
    hashTable.erase(randomInt);
    auto end = chrono::high_resolution_clock::now();
    auto duration = chrono::duration_cast<chrono::microseconds>(end - start);
    cout << "Remove time: " << duration.count() << " microseconds" << endl;
}

int main() {
    int numInts = 10000;
    vector<int> randomInts(numInts);
    for (int i = 0; i < numInts; i++) {
        randomInts[i] = generateRandomInt();
    }

    unordered_map<int, int> separateChainingHashTable;
    unordered_map<int, int> openAddressingHashTable;
    unordered_map<int, int> linearProbingHashTable;
    unordered_map<int, int> quadraticProbingHashTable;

    for (int i = 0; i < numInts; i++) {
        separateChainingHashTable[randomInts[i]] = i;
        openAddressingHashTable[randomInts[i]] = i;
        linearProbingHashTable[randomInts[i]] = i;
        quadraticProbingHashTable[randomInts[i]] = i;
    }

    cout << "Separate Chaining Hash Table:" << endl;
    measureSearchTime(separateChainingHashTable);
    measureRemoveTime(separateChainingHashTable);
```

```
    cout << "Open Addressing Hash Table:" << endl;
    measureSearchTime(openAddressingHashTable);
    measureRemoveTime(openAddressingHashTable);

    cout << "Linear Probing Hash Table:" << endl;
    measureSearchTime(linearProbingHashTable);
    measureRemoveTime(linearProbingHashTable);

    cout << "Quadratic Probing Hash Table:" << endl;
    measureSearchTime(quadraticProbingHashTable);
    measureRemoveTime(quadraticProbingHashTable);

    return 0;
}
```

OutPut:

Separate Chaining Hash Table:
Search time: 0 microseconds
Remove time: 1 microseconds
Open Addressing Hash Table:
Search time: 0 microseconds
Remove time: 0 microseconds
Linear Probing Hash Table:
Search time: 1 microseconds
Remove time: 0 microseconds
Quadratic Probing Hash Table:
Search time: 0 microseconds
Remove time: 3 microseconds