Student name: Quan Hong Doan

Student ID: 986956

Assignment: Lab 8

Week: 09

Due: Dec 16, 2019

**Lab 8**

**Problem 1**:

The following hashtable has size 43; stores integers in the range [0, 999]; and uses the hash function h(k) = k% 43. Collision-handling is accomplished by quadratic probe.

a. What is the expected number of probes when searching for a number that is not in the table? What would it be if you were doing a linear probe? Write your answer as a fraction.

b. Into which slot will the integer 59 be placed? When would it go if you were doing a linear probe?

c. Into which slot will the integer 436 be placed?

d. Assume that we were allowed to physically remove the number 783 from the table. After it is removed, what would be the result if we tried to find and retrieve the number 95 from the table?

e. How is the delete operation handled in quadratic probe? Use part (d) to explain why it is done this way.

**Answer**:

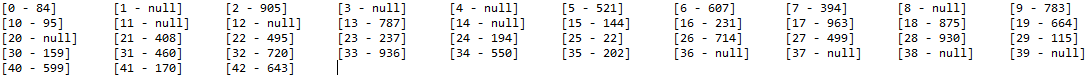
1. The expected number of probes depends on the table size, the algorithm of hashing, the occupants with same hash value.

Our load factor in this case is α=31/43

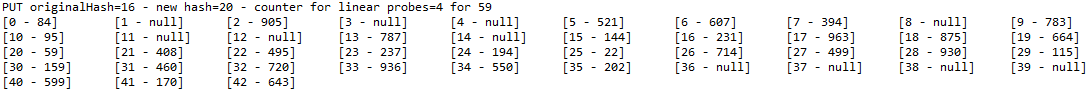
Therefore for the unsuccessful searches and insertions, expected number of probes is

1. In linear probing, the hash value of 59 is 16. The location 16 is occupied by 231, and so on. Therefore we need to check the next available slot. After probing 4 times, the slot 20 is available. Now we can insert.

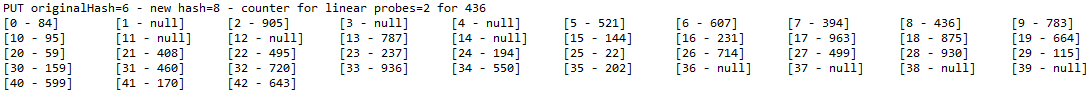
Simulation of the table before inserting 59.



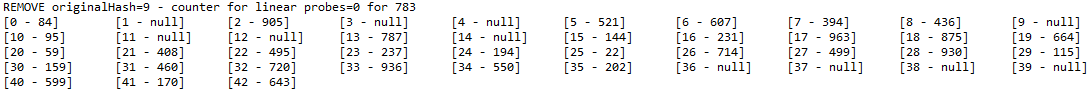
The table after inserting 59.



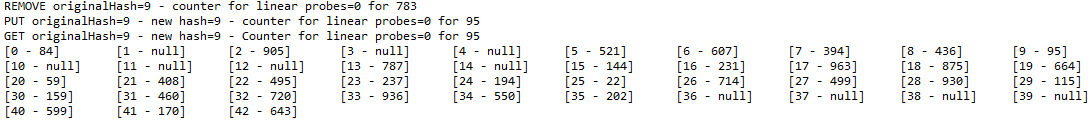
1. Hash value of 436 is 6, which is already occupied, therefore we need to probe to the next available slot, which is slot 8. Now we will insert 436.



1. If we remove 783, then the slot 9 will be emptied. However, 95 also has same hash value 9, which is located at slot 10, next to slot 9 due to the linear probing algorithm. If we just remove 783 only, then when we search 95, we will receive null value, because the hash of 95 is 9, but the location 9 is empty.



To prevent this happening, we have to rehash those values having same hash, and then re-insert them back to the table. Below is a simulation of the rehashing and re-putting sequences.



1. Quadratic probe or linear probe is only about the hashing mechanism. With linear, we move to the next available slot. With quadratic, we move to the next double prime available slot.

Linear probing hash: hash = (hash + 1) % ***TABLE\_SIZE***;

Quadratic probing hash: hash = (hash + h \* h + 1) % ***TABLE\_SIZE***;

(h starts from 1)

So, the deletion is handled in the same way of linear probing: remove the entry, rehash all similar hash valued entries, and re-put them back into the table with the re-calculated hash values.

Below are code for linear and quadratic removing.

**public** **void** removeLinearProbing(**int** key) {

**int** hash = (key % ***TABLE\_SIZE***);

**int** originalHash = hash;

**while** (table[hash] != **null** && table[hash].getKey() != key) {

hash = (hash + 1) % ***TABLE\_SIZE***;

}

table[hash] = **null**;

**for** (hash = (hash + 1) % ***TABLE\_SIZE***; table[hash] != **null**; hash = (hash + 1) % ***TABLE\_SIZE***) {

HashEntry tmp = table[hash];

table[hash] = **null**;

putLinearProbing(tmp.getKey(), tmp.getValue());

}

}

**public** **void** removeQuadricProbing(**int** key) {

**int** hash = (key % ***TABLE\_SIZE***);

**int** h = 1;

**int** originalHash = hash;

**while** (table[hash] != **null** && table[hash].getKey() != key) {

hash = (hash + h \* h + 1) % ***TABLE\_SIZE***;

}

table[hash] = **null**;

**for** (hash = (hash + h \* h + 1) % ***TABLE\_SIZE***; table[hash] != **null**; hash = (hash + h \* h + 1) % ***TABLE\_SIZE***) {

HashEntry tmp = table[hash];

table[hash] = **null**;

putQuadricProbing(tmp.getKey(), tmp.getValue());

}

}

**Problem 2**:

Implement an enhanced stack that supports push, pop, peek, isEmpty and also min, so that worst case running time for any operation is still O(1). Write down your idea and your logic for concluding that operations are in every case O(1). Then implement your idea in a Java class called MinStack.

**Answer**:

For the push or pop or peek or isEmpty operation, the running time is always O(1), because:

* Push: basically we just insert 1 data into a stack, no traversal is needed, it’s O(1).
* Pop: we just remove the top element, if it’s not null, that’s O(1) operation.
* Peek: if it’s not null, we just read the top element data.
* isEmpty: if it’s not null, we return true, otherwise, false.

However, getMin operation is different. There are 2 approaches.

1. Using an external variable along with the stack to store the minimum value after each push or pop operation.

With this approach, there’s a tricky part as when we pop out the top element and if it’s the current minimum value, then we need to check the next minimum value and update the external store of the minimum value. In this case, we can apply a calculation such as:

For PUSH operation:

If the input data < minimum value then

Push (input data x 2 – current minimum) into the stack

Minimum 🡨 input data

For POP operation:

Temp variable 🡨 popped value from the stack

If temp < minimum then

Result 🡨 minimum

Minimum 🡨 2 x minimum – temp

Else

Result 🡨 temp

Return result

The reason of this calculation is to ensure the (2 x input data – current minimum) is always smaller than input data, because: input data = n, current minimum = min

n < min <=> n – min < 0 <=> n – min + n < n <=> 2 x n – min < n

n is new minimum value.

When we pop out, we need to ensure the previous minimum (prevMin) is calculated from the current minimum (min) and the popped value (k).

We know: n is current min, n = min

k = 2 x n – prevMin <=> prevMin = 2 x min – k

Following is the implementation:

**public** **class** MyStack {

Stack<Integer> *myStack*;

**int** *min*;

MyStack() {

*myStack* = **new** Stack<Integer>();

}

**void** push(Integer data) {

**if** (*myStack*.isEmpty()) {

*min* = data;

*myStack*.push(data);

**return**;

}

**if** (data < *min*) {

*myStack*.push(2 \* data - *min*);

*min* = data;

} **else** {

*myStack*.push(data);

}

}

**void** peek() {

**if** (*myStack*.isEmpty()) {

System.***out***.println("Stack empty");

**return**;

}

Integer tmp = *myStack*.peek();

Integer result = **null**;

**if** (tmp > *min*) {

result = tmp;

} **else** {

result = *min*;

}

}

**void** pop() {

**if** (*myStack*.isEmpty()) {

System.***out***.println("Stack empty");

**return**;

}

Integer tmp = *myStack*.pop();

Integer result = **null**;

**if** (tmp < *min*) {

result = *min*;

*min* = 2 \* *min* - tmp;

} **else** {

result = tmp;

}

}

Integer getMin() {

**if** (*myStack*.isEmpty()) {

**return** **null**;

}

**return** *min*;

}

}

1. Using an auxiliary stack to store the minimum value in pair with the actual stack.

With this approach, we don’t need to calculate, we just create another auxiliary stack (called minStack) along with the actual stack. When we insert into the actual stack, we check the value to be inserted with the external minimum value.

If input data < minimum then

Insert “input data” into actual stack

Insert “input data” into minStack

Else

Insert “input data” into actual stack

Insert minimum into minStack

When we pop out, we pop both stack and update the minimum store with the new top element of the minStack.

Following is the implementation.

**public** **class** MyOtherStack {

Stack<Integer> myStack;

Stack<Integer> minStack;

**int** min;

MyOtherStack() {

myStack = **new** Stack<Integer>();

minStack = **new** Stack<Integer>();

}

**void** push(Integer data) {

**if** (myStack.isEmpty() || data < min) {

myStack.push(data);

minStack.push(data);

min = data;

} **else** **if** (data > min) {

myStack.push(data);

minStack.push(min);

}

}

Integer getMin() {

**if** (myStack.isEmpty()) {

**return** **null**;

}

**return** min;

}

Integer pop() {

**if** (myStack.isEmpty()) {

**return** **null**;

}

minStack.pop();

min = minStack.peek();

**return** myStack.pop();

}

}

**Problem 3**:

Start with an empty stack of integers. You will attempt to do a sequence of pushes and pops so that the sequence of pops will be a specified permutation of 1, 2, 3, 4, 5, 6. You will be able to do exactly 6 push operations and 6 pop operations. The first push pushes 1 onto the stack; the next pushes 2; and so forth. The sixth push pushes 6 onto the stack.

For this exercise, we will let S denote a push operation and X a pop operation. Example: The sequence SSSSSSXXXXXX outputs 654321.

a. Describe a sequence of pushes and pops that would produce output 325641 (or explain why it is not possible)

b. Describe a sequence of pushes and pops that would produce output 154623 (or explain why it is not possible)

**Answer**:

We have “325641” and “154623” are permutations of 1, 2, 3, 4, 5, and 6.

If we start the push operations from 1, then 2, then 3, then 4, then 5, then 6, so when we pop the data from the stack, the result will be 654321.

So if we want the result in different permutations, we have to arrange the push operations in proper order.

* For “325641”:
  + The push order will be: 1 > 4 > 6 > 5 > 2 > 3
  + The pop order: 3 > 2 > 5 > 6 > 4 > 1

Testing:

Stack: [1, 4, 6, 5, 2, 3]

Call 6 pop operations: 325641

* For “154623”:
  + The push order will be: 3 > 2 > 6 > 4 > 5 > 1
  + The pop order: 1 > 5 > 4 > 6 > 2 > 3

Testing:

Stack: [3, 2, 6, 4, 5, 1]

Call 6 pop operations: 154623

**Problem 4**:

Devise an algorithm for reversing the elements in a singly linked list. Implement your solution in code. You can use the singly linked list in the lab folder.

**Answer**:

Basically, the linked list before reversing looks like below:

[Head] 🡪 [Node 1] 🡪 [Node 2] 🡪 [Node 3] 🡪 [Node 4] 🡪 null

After reversing, the linked list would be:

[Head] 🡪 [Node 4] 🡪 [Node 3] 🡪 [Node 2] 🡪 [Node 1] 🡪 null

The idea for reversing is:

* Assume, we don’t deal with Head node.
* Using a temporary node to store the next of the current node
* Assign the previous node to the next of the current node
* Assign the current node to the previous node (so we can move backward, or leftward on the list).
* Then assign the temporary node to the current node

Simulation of the 4 steps:

List: 2 > 7 > 9 > 18 > null

temp=current.next: temp=7 >> current=2 >> current.next=7 >> prev=null

current.next=prev: temp=7 >> current=2 >> current.next=null >> prev=null

prev=current: temp=7 >> current=2 >> current.next=null >> prev=2

current=temp: temp=7 >> current=7 >> current.next=9 >> prev=2

The implementation.

**public** **class** MyLinkedList {

**static** Node *head*;

**static** **class** Node {

**int** data;

Node next;

Node(**int** data) {

**this**.data = data;

**this**.next = **null**;

}

}

**void** toString(Node node) {

**while** (**null** != node) {

System.***out***.print(node.data + " > ");

node = node.next;

}

**if** (**null** == node)

System.***out***.println(node);

}

**static** Node reverse(Node node) {

Node prev = **null**;

Node current = node;

Node temp = **null**;

**while** (**null** != current) {

temp = current.next;

current.next = prev;

prev = current;

current = temp;

}

node = prev;

**return** node;

}

**public** **static** **void** main(String[] args) {

MyLinkedList mll = **new** MyLinkedList();

mll.*head* = **new** Node(2);

mll.*head*.next = **new** Node(7);

mll.*head*.next.next = **new** Node(9);

mll.*head*.next.next.next = **new** Node(18);

mll.toString(mll.*head*);

*head* = mll.*reverse*(*head*);

mll.toString(mll.*head*);

}

}

Testing:

2 > 7 > 9 > 18 > null

18 > 9 > 7 > 2 > null