Informatics II, Spring 2024, Solution Exercise 10

Publication of exercise: May 6, 2024
Publication of solution: May 13, 2024
Exercise classes: May 13 - 17, 2024

Learning Goal

- Understanding hash tables, hash functions and how to deal with collisions.
- Knowing how to use hash tables to solve problems.

Task 1 [Easy]

What is the resulting hashtable `HT` after running the script?

Assume `HTInsert()` is called with `value_array = {3, 14, 33, 27, 9, 30, -2, 0, 2, 16, 8}`, thus also `n = 11`. You can also assume that `HT` is an array of type pointers to `struct Node` and has a length of `n = 11`. It is initialized with `NULL` values.

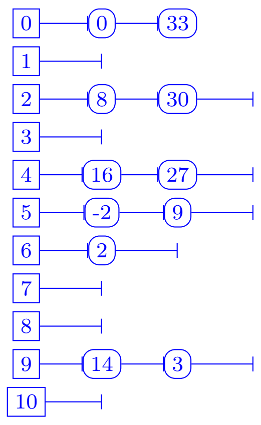
```
Algorithm: HTInsert(*HT, *value_array, n)
for i=0 to n do
    k = (value_array[i] * 3) mod n;
    new_node = malloc(sizeof(struct Node));
    new_node->val = value_array[i];
    new_node->next = HT[k];
    HT[k] = new_node;
```

```
Algorithm: HTInsert(*HT, *value_array, n)
for i=0 to n do
    // calculate the hash function with value to get key;
    k = (value_array[i] * 3) mod n;
    // allocate element in linked list;
    new_node = malloc(sizeof(struct Node));
    // set the value of the node ;
    new_node->val = value_array[i];
    // set the next to the current linked list which is saved at key k ;
    new_node->next = HT[k];
    // overwrite current pointer saved at position ;
    HT[k] = new_node;
```

This is the more simplified version of the algorithm above. However, this version is based on the book, not the lecture slides.

```
Algorithm: HTInsert(*HT, *value_array, n)
for i=0 to n do
    k = (value_array[i] * 3) mod n;
    List-Prepend(HT[k], value_array[i]);
```

This is the hash table visualized. Because `n=11`, we know that `HT` has indexes `1..10`.



Task 2 [Medium]

You are provided the script `hash_functions.c`. This script already sets up the following functions:

- `main()`: Declares some variables needed for the hash functions and run time calculations. You do not need to change something here.
- `initHT(int *HT)`: Initializes the hash table `HT`, i.e, fills it with `-1` which mark unused slots. You do not need to change something here.
- `printHT(int *HT)`: Prints the content of the hash table `HT`. You do not need to change something here.
- `insertKeyH1(int *HT, int k)`: See taks **(a)**.
- `insertKeyH2(int *HT, int k)`: See task **(b)**.

The hash table HT is initialized in the `main()` function. It has a size of `N` slots. Implement the tasks **(a)** and **(b)** in order to insert the 100 keys given in `keys`. In the end we want to compare the relative runtime, which is why the `start_time` and `end_time` are declared. Additionally, we want to compare the number of collisions.

Hint: For this task, do not use compiler optimizations, in order to better see the effect on the runtime.

(a) Implement a Hash Function

Consider the following **hash function**: $h_1(k) = k \mod 7$. Your task is now to implement this hash function to implement the `insertKeyH1()` function. This function takes a pointer tho the hash table `HT` array and the key `k`, which should be inserted into the hash table. Use **linear probing** to resolve conflicts and return the number of conflicts occured in the insertion for the key `k`. (These are then added up for the 100 keys to get the total number of conflicts, which is already implemented.)

Compile and run the script. **Answer the following questions**

- What is the runtime?
- What is the number of collisions?
- How are the 100 given keys distributed in the hash table?
- Is this a good hash function?

See `./code_solutions/hash_functions_solution.c` for solution implementations. Goal was to try different hash functions and see the effect on the runtime.

- What is the runtime?
  - *$\approx$ 30 clocks*
- What is the number of collisions?
  - *4662 collisions*
- How are the 100 given keys distributed in the hash table?
  - *Not well distributed; All clustered into the first 2/3 of the hash table*
- Is this a good hash function?
  - *No*

(b) Implement a Second Hash Function

Implement a second hash function $h_2(k)$, also with **linear probing**, in the `insertKeyH2()`. You can design this hash function.

- Find a hash function that is better than $h_1(k)$. *Hint: Try different values and methods for the hash function.*

Compare the two implementations regarding clock time and number of collisions.

- What can you conclude?

Goal was to find an improved hash function. There is no incorrect solution as long as the runtime is faster and there are fewer collisions. Goal was to see the impact of different hash functions on the solution. Here are a few examples. The speedup is measured with a base of 34 clocks for h_1.

- $h_2(k) = (23k) \mod 97$ with linear probing
  - Note that these are prim numbers. 97 is far away from the next 2^p but still does not distribute the numbers well. That is why ther are still 564 collisions. Nevertheless, the speedup is $\approx 34/9 \approx 3.7$ times faster
- $h_2(k) = (23k) \mod 127$ with linear probing
  - Note that 127 is close to 2^7 but does distribute better. That is why there are only 106 collisions with a speedup of $\approx 34/5 \approx 6.8$. Better, but there are still the last 30 slots of the hash table unused
- $h_2(k) = (23 * k) \mod 127$ with double hashing $h_2(k)' = (73k) \mod 101$
  - This time double hashing is used, which clearly does distribute better, since there are no more than 6 consecutive slots empty. This is reflected in the number of collisions as well, only 53 collisions this time. The speedup improves just slightly to $\approx 34/4 \approx 8.5$ compared to h_1

Note on using the solution code: Comment or uncomment the following lines in the `InsertKeyH2()` function to use linear probing or double hashing.

```
/* int step = 1;              /\* linear probing *\/ */
int step = h2Step(k);       /* double hasing */
```

And modify the h_2 or the h_2' to see the effects yourself.

```
int h2(int k) {
    return (23 * k) % 127;
    /* it is possible to go higher, thus no collision, e.g.:  */
}

int h2Step(int k) {
    return (73 * k) % 101;
}
```

Task 3 [Medium/Hard]

A hash table of length 10 uses open addressing with hash function $h(k) = (k + 4) \mod 10$, and linear probing (probing goes to the right). After inserting five values into an empty hash table, the table is as shown below.

slot	0	1	2	3	4	5	6	7	8	9
key	96						32	52	33	74

(a) Find Insertion Order

Which of the following choice(s) give possible order(s) in which the key values could have been inserted in the table?

1. 32, 33, 52, 96, 74
2. 32, 52, 33, 74, 96
3. 32, 52, 74, 96, 33
4. 96, 32, 52, 33, 74

Solution: 2., 4.

(b) Delete Keys

Implement a function `HTDelete()` which takes a key `k` to be deleted. Upon the deletion the succeding keys are moved to the right place, as if `k` was never inserted in the first place.

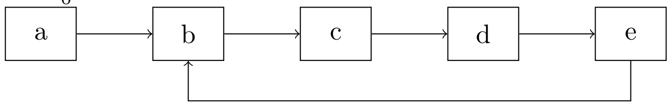
For example on the above table, we call `HTDelete(33)`, we then get the following table:

slot	0	1	2	3	4	5	6	7	8	9
key	96						32	52	**74**	

See `./code_solutions/hash_table_delete.c`

Task 4 [Easy]

Consider the following linked list. For simplicity assume that the values, a, b, c, d, e, in the list are distinct and $\in \mathbb{N}_0^+$



As one can see, there is a cycle in it. Implement a function `detectCycle()` which detects such cycles in a linked list. This function should return `0` if there is no cycle, and `1` if there is. The worst case runtime for `detectCycle()` should be $O(n)$, where n is the number of distinct elements.. *Hint: Think about how hashing can help here.*

Key idea is to use a hash function. If the same key is already in the hash table, we know that this key was already encountered, thus there is a cycle. See `./code_solutions/detect_cycles.c`.

Task 5 [Medium]

Cuckoo hashing has two hash tables each with n elements. There are two hash functions h_1 and h_2 to map the domain into n. Every element x in the domain will either be at position $h_1(x)$ in the first table or $h_2(x)$ in the second table.

To insert an element x, start by inserting it into the first table. If $h_1(x)$ is empty, x is placed there. Otherwise, place x there by pushing the previous element y into the table 2. Repeat this process, bouncing between tables, until all elements stabilize.

(a) Questions

Answer the following:

- What is the worst case lookup time for a key k?
- What is the worst case deletion time for a key k?

These questions can be answered with the introduction text or quickly looking up the wikipedia article.

- What is the worst case lookup time for a key k?
  - *Answer*: Thus lookups are worst case of $O(1)$ (only two locations must be checked).
- What is the worst case deletion time for a key k?
  - *Answer*: Thus deletions are worst case of $O(1)$ (only two locations must be checked).

(b) Cuckoo Insert

Write a `cuckooInsert()` function, which takes a key `k` and inserts it into the table 1. This function should also handle collisions. It returns the table number and which index in that table. Use the following hash function $h_1(x) = x \mod 10$ and $h_2(x) = \lfloor x/10 \rfloor \mod 10$

Additionally, find an example of insert keys which result in an infinite cycle.

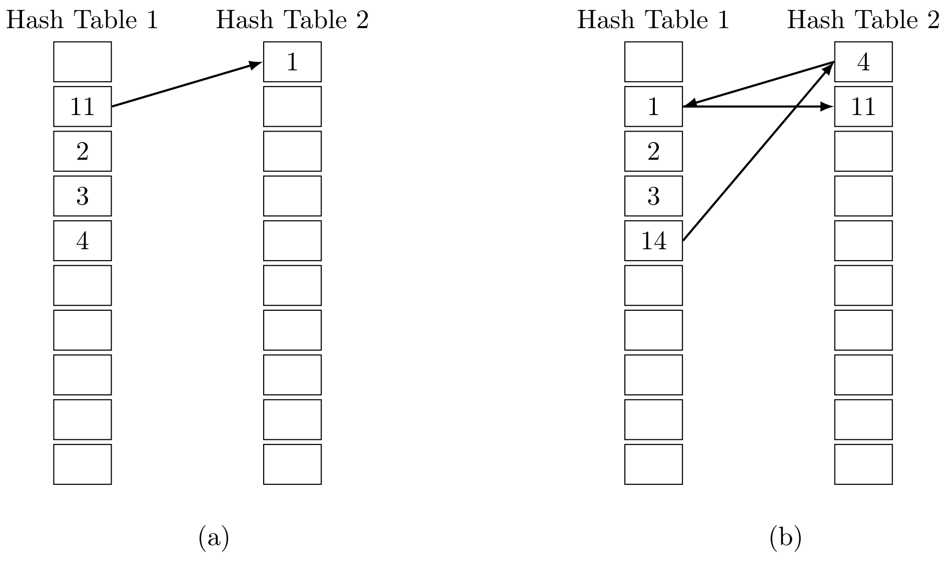


Figure 1: Hash tables for cuckoo hashing. **Situation (a)** after inserting 1, 2, 3, 4, 11. Note how the 11 pushes the 1 into the second hash table (marked by arrow). **Situation (b)** after inserting 14. Note how the 14 pushes the 4 into the second hash table, which pushes the 1 back to the first hash table, which pushes the 11 into the second hash table.

See `cuckooInsert()` in `./code_solutions/cuckoo_hashing.c`.

- Easy example for infinite cycle based on the given example: $12, 112, 1112$
- Other, more complicated example: $1, 11, 2, 12, 112$

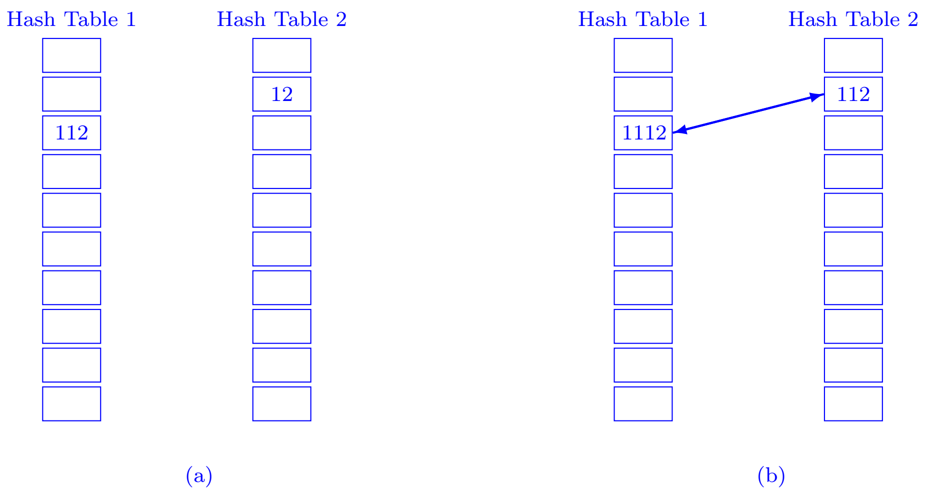


Figure 2: Hash tables for cuckoo hashing. **Situation (a)** after inserting $12, 112$. **Situation (b)** leads to an infinite cycle on inserting 1112.

(c) Cuckoo Search

Write a `cuckooSearch()` function, which takes a key `k` and searchs for it. It prints the table number and which index in that table. Keep in mind that the asymptotic runtime should adhere to your answer of **(a)**.

See `cuckooSearch()` in `./code_solutions/cuckoo_hashing.c`. This is fairly easy, due to the $O(1)$ looup time.