

### EXERCISE 1: Hashing and Chaining with String keys

Let's assume the hash table size = 13

Use the hash function to load the following commodity items into the hash table:

onion	1	10.0			
tomato	1	8.50	Banana	3	4.00
cabbage	3	3.50	olive	2	15.0
carrot	1	5.50	salt	2	2.50
okra	1	6.50	cucumber	3	4.50
mellon	2	10.0	mushroom	3	5.50
potato	2	7.50	orange	2	3.00

Will use ASCII code for the characters as follows:

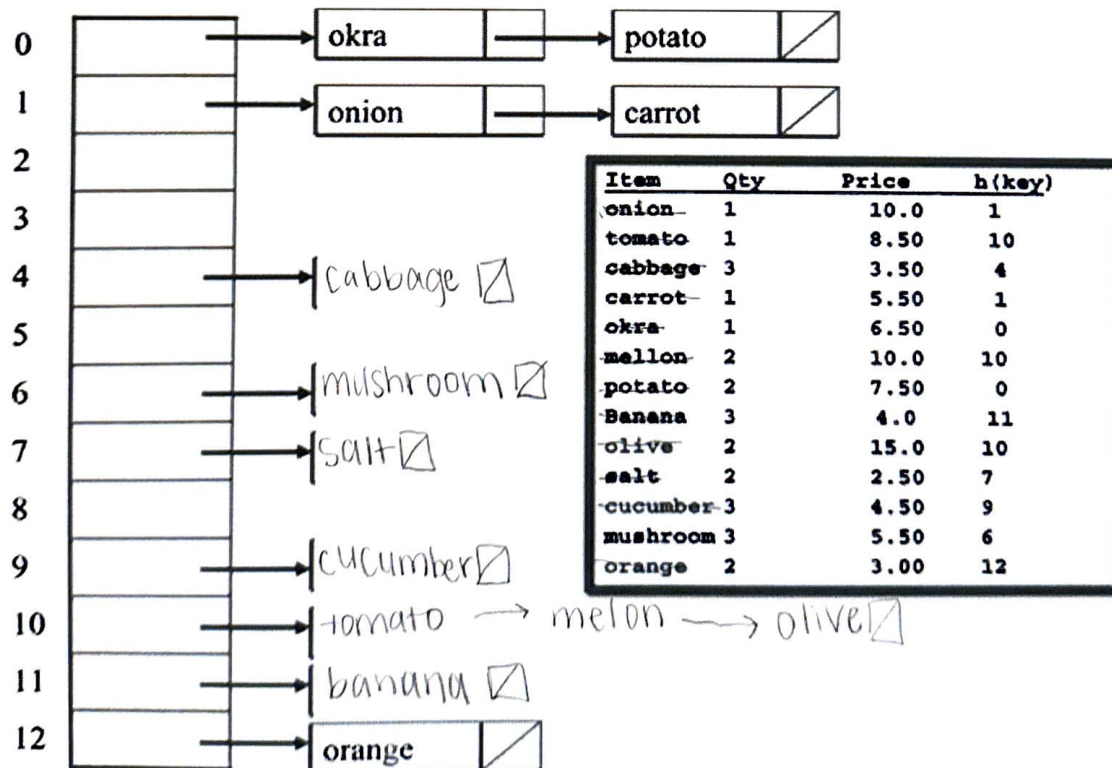
character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

For instance:

$$\text{hash}(\text{onion}) = (111 + 110 + 105 + 111 + 110) \% 13 = 547 \% 13 = 1$$

$$\text{hash}(\text{orange}) = (111 + 114 + 97 + 110 + 103 + 101) \% 13 = 636 \% 13 = 12$$

Complete the diagram below using the Chaining collision resolution technique:



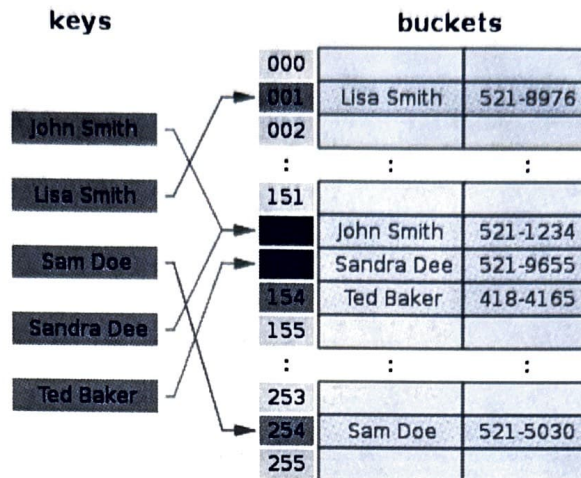


Figure 2: Handling Collisions with Linear Probing

### EXERCISE 2: Hashing and Linear Probing

Given this hash table's initial configuration: (Note: size of table = 13, "E" = Empty state)

Index	Status	Value
0	E	26
1	E	
2	E	
3	E	
4	E	
5	E	18
6	E	
7	E	
8	E	47
9	E	35
10	E	9
11	E	
12	E	64

- Perform the operations in the table below showing the following two things after each operation:
  - The hash index or the probe sequence if necessary
  - A comment "Collision" / "Success" / "Fail" to indicate the appropriate event\*
- Show the final hash table after all the operations have been performed

The first operation has been done for you:

Operation	Index or Probe Sequence	Comment
Insert(18)	$h_0(18) = (18 \% 13) = 5$	Success
Insert(26)	$h_0(26) = (26 \% 13) = 0$	Success
Insert(35)	$h_0(35) = (35 \% 13) = 9$	Success
Insert(9)	$h_0(9) = (9 \% 13) = 9$ $h_1(9) = ((9+1) \% 13) = 10$	Collision Success
Find(15)	$h_0(15) = (15 \% 13) = 2$	Fail
Find(48)	$h_0(48) = (48 \% 13) = 9$ $h_1(48) = ((48+1) \% 13) = 10$	$h_2(48) = ((48+2) \% 13) = 11$ Collision, Collision, Fail
Find(9)	$h_0(9) = (9 \% 13) = 9$ $h_1(9) = ((9+1) \% 13) = 10$	Collision Success
Insert(64)	$h_0(64) = (64 \% 13) = 12$	Success
Insert(47)	$h_0(47) = (47 \% 13) = 8$	Success
Find(35)	$h_0(35) = (35 \% 13) = 9$	Success

One entry in final hash table (notice the change of status from "E" to "O"):

Index	Status	Value
0	E	
...	E	
5	O	18
...	E	

\* Note for Exercise 2 on when to use the "Collision" / "Success" / "Fail" comments:

- "Collision"
  - Inserting but cell is occupied
  - Finding but something else is there
- "Success"
  - Able to insert
  - Able to find
- "Fail"
  - Unable to find (while linear probing you find a cell with status "Empty" ("E"))



Alanna Hazlett  
Uwa6xv  
9/19/24

I pledge that I have neither given nor received help on this assignment.

EXERCISE 3: Additional questions

Alanna K. M. Hazlett

Q1) Name one advantage of Chaining over Linear Probing.

Control of data structure in effort to reduce search times. Saves time.

Q2) Name one disadvantage of Chaining that isn't a problem in Linear Probing.


Memory usage, when collisions occur more memory is requested/needed.

Q3) If using Chaining, how can finding an element in the linked list be made more efficient?

Using different data structures, like binary search trees, reduce search time.

Q4) Why does Linear Probing require a three-state (Occupied, Empty, Deleted) "flag" for each cell, but Chaining does not? You may use an example as an illustration to your argument.


For linear probing, if we remove an element from the array we mark it as deleted, because if another item had previously had a collision at that bucket it would now occupy a bucket after that. This means when searching if it was marked as empty, it would determine that the item was not in the array, even though it was. When using the deleted marker the linear probing knows to continue searching the buckets either until it finds the item or until it reaches an empty bucket (indicating the item is not in the array).



0	1	2	3	4	5	6	7
E	O	O	E	O	O	E	E

Arrows point from bucket 1 to bucket 2, and from bucket 2 to bucket 3.

If the hash function determines item should be in bucket 1, but it is occupied by a different item it will continue searching. In this case it reaches an empty bucket, which indicates that the item is not in the array.



0	1	2	3	4	5	6	7
E	O	O	D	O	O	E	E

Arrows point from bucket 1 to bucket 2, from bucket 2 to bucket 3, and from bucket 3 to bucket 4.

If in reality our item originally hashes to bucket one, but had multiple collisions until it reaches bucket 4 and then the item in bucket 3 was deleted prior to our search the deleted notation allows us to continue our linear probe until we find it (in this case bucket 4) or reach an empty bucket.