Homework: Hashing Conflict Resolution

GRADING: A maximum of **100 points** can be obtained on this homework assignment.

Chaining Background

- The hash table is implemented as an array of linked lists
- Inserting an item, ***r***, that hashes at index ***i*** results in the insertion of item ***r*** into the linked list at position ***i*** (of the array)
    - Insertion into the linked list is at the tail end (such that the first item in the linked list was hashed there before the second item in the linked list, and so on…)
- *Synonyms*, items that are hashed to the same index, are chained in the same linked list

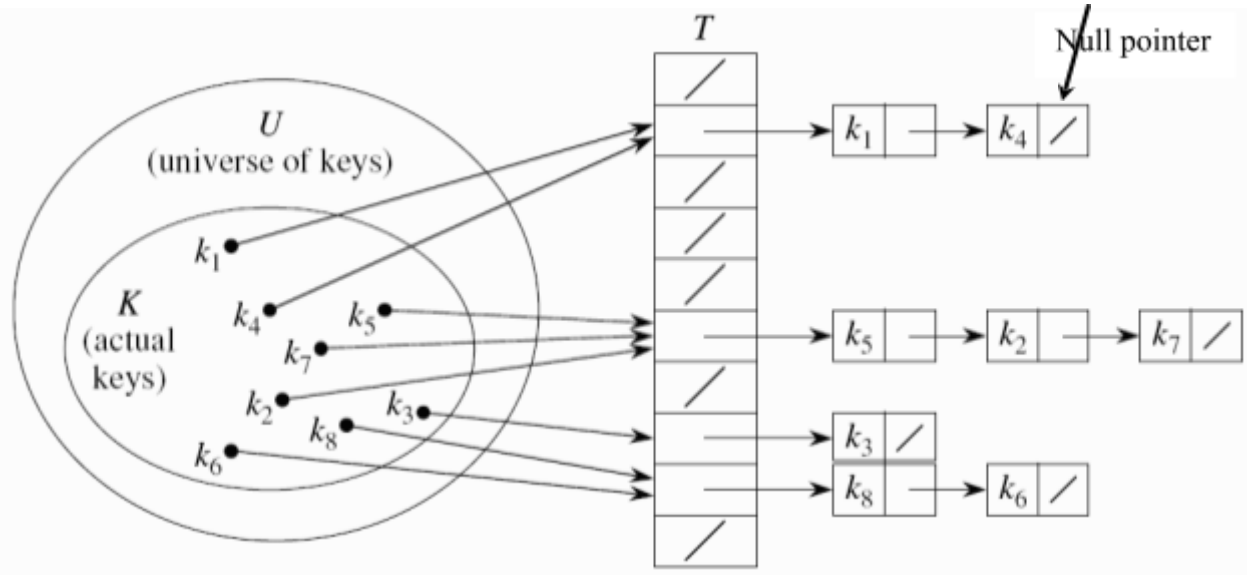


Figure 1: The hash table is array T; keys (k_n) hashed to an integer value; items placed into the linked list at the hash index *i* in array T. (Every element of the array is a linked list, elements that have no keys are simply linked lists of size 0.)

- Retrieval of an item, ***r***, with hash address, ***i***, is a simple process of retrieval from the linked list at position ***i***. Traverse the linked list to find the element you're looking for
- Deletion of an item, ***r***, with hash address, ***i***, is a simple process of deleting ***r*** from the linked list at position ***i***. Traverse the linked list to find the element and then delete it, finally rearrange the pointers to complete the linked list
- A hash function for a string ***s*** = $c_0c_1c_2...c_{n-1}$ can be defined as: hash=($c_0+c_1+c_2+...+c_{n-1}$) % tableSize
- Often the ASCII code of a character is used when hashing a string

EXERCISE 1: Hashing and Chaining with String keys (20 points)

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:

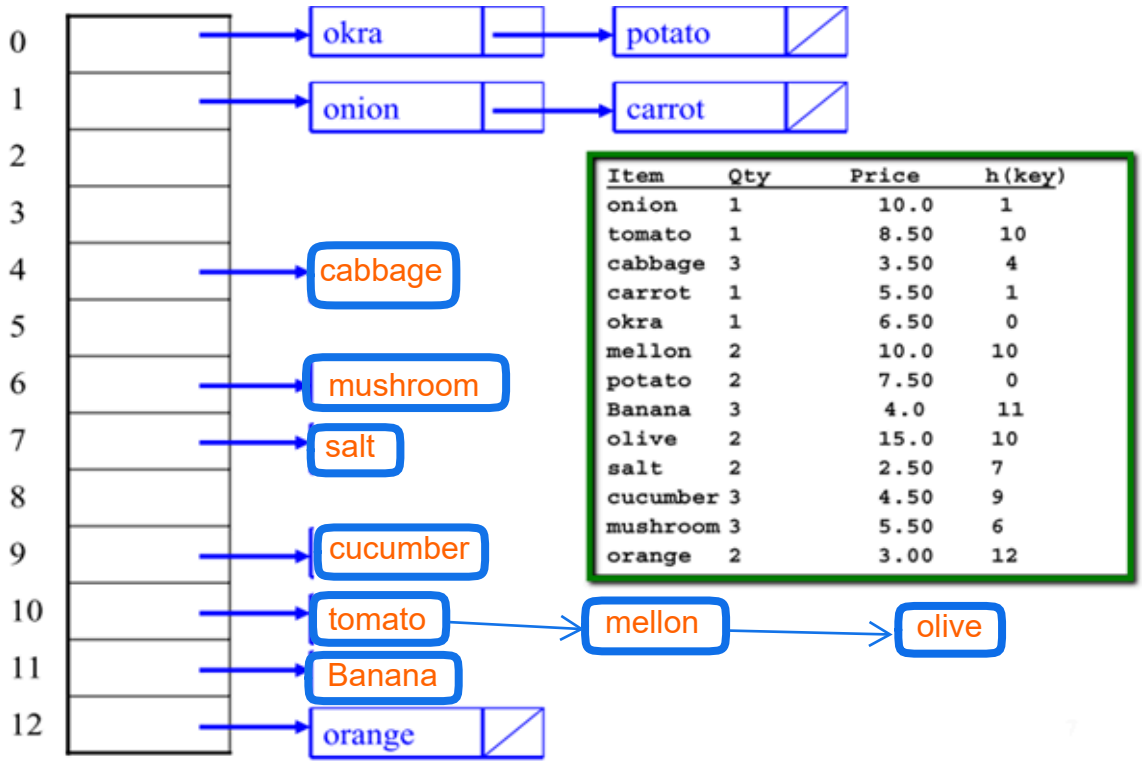
character	a	b	c	e	g	h	i	k	l	m	n	o	p	r	s	t	u	v
ASCII code	97	98	99	101	103	104	105	107	108	109	110	111	112	114	115	116	117	118

For instance:

hash(onion) = (111 + 110 + 105 + 111 + 110) % 13 = 547 % 13 = **1**

hash(orange) = (111 + 114 + 97 + 110 + 103 + 101) % 13 = 636 % 13 = **12**

Complete the diagram below using the Chaining collision resolution technique:



Linear Probing Background

- All items are stored in the hash table itself – no need for another data structure
- While inserting, if a collision occurs, alternative cells (in the array) are tried until an empty cell is found
- It's called "**linear probing**" since finding an empty cell after a collision occurs involves a linear search from the hashed index to the end of the array. The first empty cell found is where the item will be placed
- A "**probe sequence**" is the sequence of array indices that is checked in attempt to search for an empty cell during insertion after a collision has occurred
- The most common probe sequences advance by one index at a time, i.e., the probe sequences are of the form: $\mathbf{h_i(key) = [h(key) + c(i)] \% n}$, for i = 0, 1, …, n-1 where ***h*** is a hash function and ***n*** is the size of the hash table
- The function **c(i)** is used to resolve collisions
- To insert item ***r***, first examine array location $h_0(r) = h(r)$. If there is a collision, array locations $h_1(r)$, $h_2(r)$, …, $h_{n-1}(r)$ are examined until an empty slot is found
- Similarly to find ***r***, the same sequence of locations is examined in the same order
- "Linear" probing is only one kind of "open addressing collision resolution" technique. Other kinds include "quadratic probing" and "double hashing." The only difference in the probe sequence is the definition of the function **c(i)**
- Common definitions of **c(i)** for some *collision resolution techniques* are:

Linear probing	
Quadratic probing	
Double hashing	$i*h_p(key)$, where $h_p(key)$ is another hash function

- Many implementations require each cell to keep one of three states: Empty ("E"), Occupied ("O"), or Deleted ("D")
- While linear probing is more efficient storage-wise (as compared to chaining), its success is dependent upon choosing a proper table size.
- Note, that as the table fills, there can be a sever degradation in performance

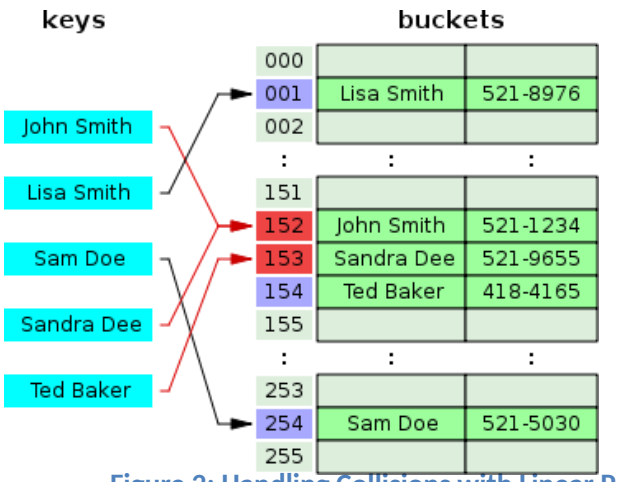


Figure 2: Handling Collisions with Linear Probing

EXERCISE 2: Hashing and Linear Probing (40 points)

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

1. Perform the operations in the table below showing the following two things after each operation:
    a. The hash index or the probe sequence if necessary
    b. A comment "Collision" / "Success" / "Fail" to indicate the appropriate event*
2. 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)	h0(18)=(18 % 13) = 5	Success
Insert(26)	h1(26)=(26%13) = 0	Success
Insert(35)	h2(35) = (35%13) = 9	Success
Insert(9)	h3(9)=[9%13 + c(3)]=9	Collision & Success
Find(15)	h4(15) = [15%13] = 2	Fail
Find(48)	h5(48)=[48%13]=9	Fail
Find(9)	h6(9)=[9%13 + c(6)]=9	Collision & Success
Insert(64)	h7(64)=(64%13)=12	Success
Insert(47)	h8(47)=(47%13)=8	Success
Find(35)	h2(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”)

EXERCISE 3: Additional questions(40 point)

Q1) Name one advantage of Chaining over Linear Probing.

Chaining can handle an arbitrary number of number of entries without having to resize.

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

Chaining requires additional memory for its linked list, leading to higher overall memory usage.

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

Chaining could be made more efficient with a better hash function. This will allow for shorter linked lists, helping with memory issues and improving runtime complexities.

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.

Chaining stores a reference to a linked list in each index of the array, so when a "collision" occures, the linked lists create new nodes to store the new value. There's no reason to use the three-state "flag" for each cell in this case.

With Linear Probing, these flags are required because the cell deals with collisions linearly. Without these flags, it wouldn't be possible to know the current state of each cell in the array.