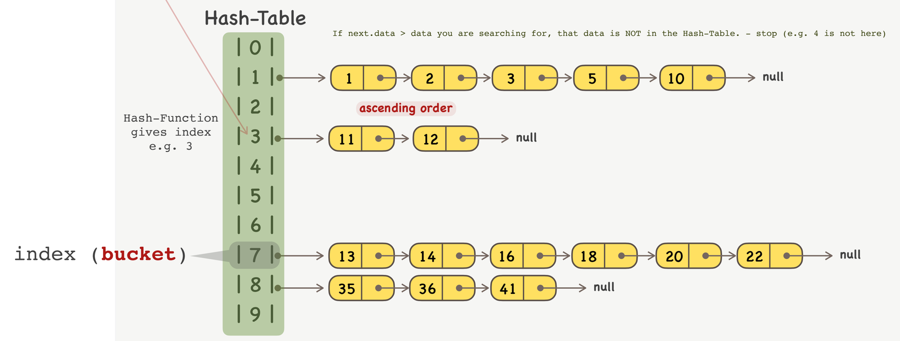
Hash Table

By Martin Gregory Sendrowicz · Fall 2019 edited for Spring 2020 class, edited again in 2023

Def: **Hash-Table (Map)** is an **array of linked-lists**. Meaning, each element is a pointer to a linked list.

Every Hash Table has two principle components in:

(1) **Hash-Function**: which main role is to give us the **index (bucket)** of the Hash-Table. At this **index (bucket)** there is a pointer to the Linked-List into/from which we insert/retrieve our data. Hence, Hash-Function tells us in which bucket we should search for our data.



(2) **Size of Hash Table** - is the number of buckets. E.g in our example there are 10 buckets, hence the size is 10. One must know the size in order to allocate the necessary space. Also size is used by Hash-Function in order to find the index/bucket (modular arithmetic).

application:
Radix Sort

purpose:
Quick information storage and retrieval - we always strive for **constant time** of data access

The **best Hash-Function** is the one that evenly distributes data among all of its buckets. Meaning, the bucket associated linked-lists should be of similar length. An ideal Hash-Function distributes the same number of data in each bucket. An ideal Hash-Function does not exist. A squirrel is a really good Hash-Function.

VS

The ideally **worst Hash-Function** is the one that puts everything into same bucket. A realistically **bad Hash-Function** is: if there are b-buckets and n-data, data is assigned to very few buckets.

MOD function is present in every Hash-Function (ALWAYS)

examples of Hash-Functions:

Hash(x) = **mod(x, size)**

Hash(x) = **mod(sum_digs(x), size)**
e.g.:
Hash(123) = mod(sum_digs(1,2,3), 10) = mod(6, 10) = 6
Hash(10) = mod(sum_digs(1,0), 19) = mod(1, 19) = 1
Hash(423) = mod(sum_digs(4,2,3,2), 10) = mod(11, 10) = 1

Hash-Table Insertion O(n)

Algorithm - steps from **storing** data into Hash-Table using an array of Linked-Lists:

Step 0: Establish **Hash-Function** and Hash-Table of a given **Size**

Step 1: data ← get data   e.g. from file.txt or some other source

Step 2: Get index (bucket) where to store data
index ← **Hash-Function**(data)

Step 3: Store data at the derived index and into Linked-List
insert(HashTable[index] , data)
*Insertion into Linked-List should be done in **ascending order**.

Step 4: repeat step 1 to step 3 until no more data

Hash-Table Retrieval O(n)

Algorithm - steps from **retrieving** data from Hash-Table:

Step 1: Get index (bucket) where the data is stored
index ← **HashFunction**(data)

Step 2: Got to the index (bucket) and search for data
true/false ← **search**(HashTable[index], data)

Array Sorting Algorithms

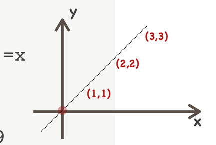
algorithm	Time Complexity best	average	worst	Space Complexity worst
Heap Sort	Ω(nlogn)	θ(nlogn)	**O(nlogn)**	O(1)
Merge Sort	Ω(nlogn)	θ(nlogn)	O(nlogn)	O(n)
Bubble Sort	Ω(n)	θ(n²)	O(n²)	O(1)
Selection Sort	Ω(n²)	θ(n²)	O(n²)	O(1)
Quick Sort	Ω(nlogn)	θ(nlogn)	O(n²)	O(logn)
Tim Sort	Ω(n)	θ(nlogn)	O(nlogn)	O(n)
Tree Sort	Ω(nlogn)	θ(nlogn)	O(n²)	O(n)
Shell Sort	Ω(nlogn)	θ(n(logn)²)	O(n(logn)²)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n²)	O(n)
Radix Sort	Ω(nk)	θ(nk)	**O(nk)**	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cube Sort	Ω(n)	θ(nlogn)	O(nlogn)	O(n)
Insertion Sort	Ω(n)	θ(n²)	O(n²)	O(1)

Radix Sort

Radix Sort is the best sort ever—it uses **Hash Table**. Two principle components of **Radix Sort Hash-Table** are:

(1) **Radix Sort Hash-Function** is the **identity** function i.e. f(x)=x

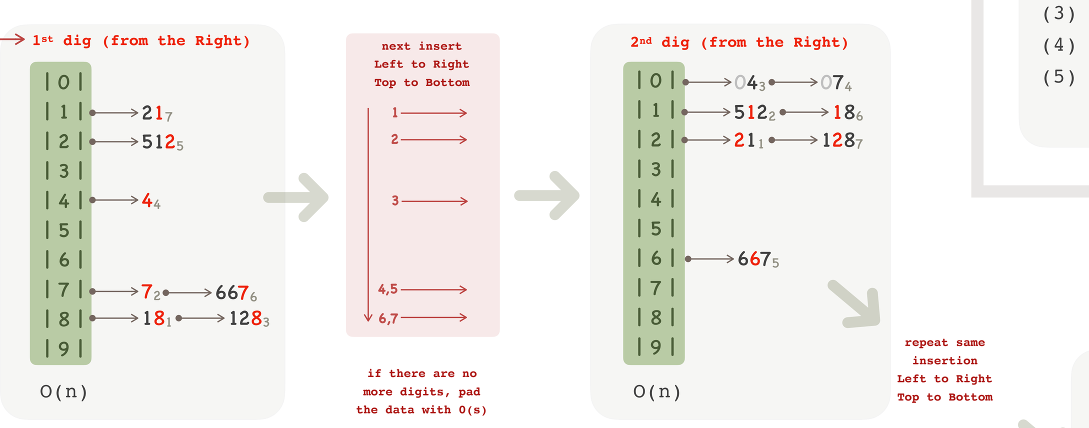
(2) **Size n** i.e. for:
- integers **n=10** meaning indices from 0 to 9
- chars **n=256** for each ASCII code

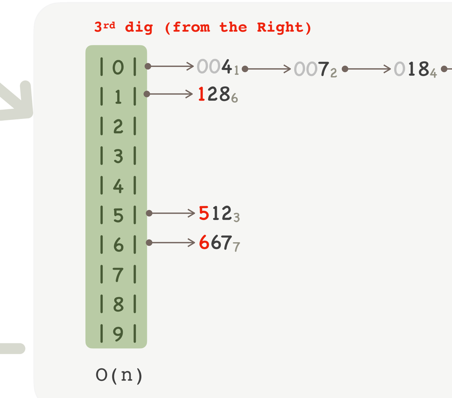


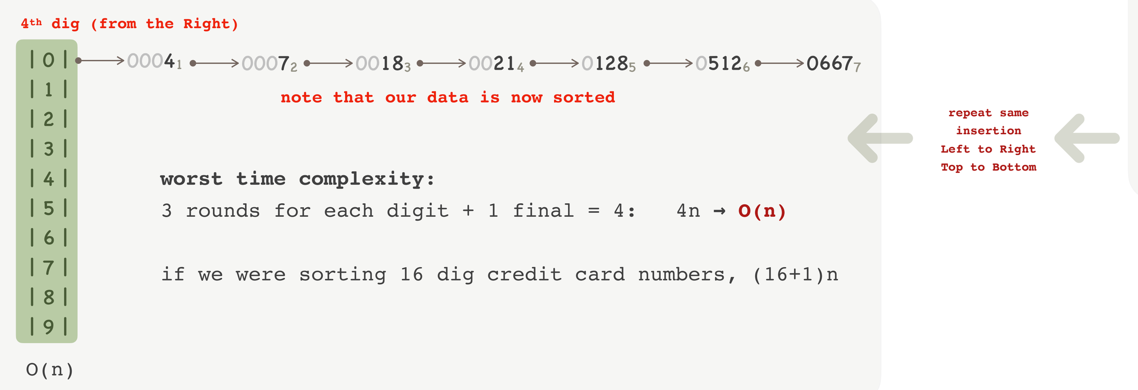
example

data: 18, 7, 128, 4, 512, 667, 21

Look at **1st digit** of each integer (**starting from right** to left) and insert it into Radix Hash-Table at the value/index/bucket equal to that digit (identity function). E.g., in our dataset, the 1st digit is 18; hence, we insert 18 into the linked-list located at index 8 of Radix Hash-Table. Then we insert the consecutive numbers into their corresponding indices. Note that we ALWAYS insert them at the end of the linked-list. The final output is sorted in ascending order.







worst time complexity:
3 rounds for each digit + 1 final = 4: 4n → **O(n)**

if we were sorting 16 dig credit card numbers, (16+1)n

sorting floats

- **if sorting floating numbers**, Size n=10

steps:
(1) find the **largest** decimal part among all numbers e.g. 85.**125** -> 3 digits
(2) multiply each number by 1000 (3 digits) e.g. 85.125 → 85125
(3) sort in the usual fashion using Radix Sort
(3) divide the final result by 1000 e.g. 85125÷1000 → 85.125

sorting negatives

- **if sorting negative numbers**, Size n=10

steps:
(1) find the **smallest** number among all negatives e.g. −79
(2) take absolute value of that number e.g. abs(−79) = 79
(3) add 79 to every negative number making them non-negative
(4) sort in the usual fashion using Radix Sort
(5) subtract 79 from all previously negative numbers making them negative again

Program Output

```
Enter input file name (e.g. input.txt): input.txt
thanks you entered: input.txt
Below is the extracted data:
18 7 128 4 512 667 21
 - - - - - - - - - - - - - - - - -
 H A S H - T A B L E - 0
bucket 0 : The Queue is empty
bucket 1 : Front-->(21)->NULL
bucket 2 : Front-->(512)->NULL
bucket 3 : The Queue is empty
bucket 4 : Front-->(4)->NULL
bucket 5 : The Queue is empty
bucket 6 : The Queue is empty
bucket 7 : Front-->(7)->(667)->NULL
bucket 8 : Front-->(18)->(128)->NULL
bucket 9 : The Queue is empty
 - - - - - - - - - - - - - - - - -
 H A S H - T A B L E - 1
bucket 0 : Front-->(4)->(7)->NULL
bucket 1 : Front-->(512)->(18)->NULL
bucket 2 : Front-->(21)->(128)->NULL
bucket 3 : The Queue is empty
bucket 4 : The Queue is empty
bucket 5 : The Queue is empty
bucket 6 : Front-->(667)->NULL
bucket 7 : The Queue is empty
bucket 8 : The Queue is empty
bucket 9 : The Queue is empty
 - - - - - - - - - - - - - - - - -
 H A S H - T A B L E - 0
bucket 0 : Front-->(4)->(7)->(18)->(21)->NULL
bucket 1 : Front-->(128)->NULL
bucket 2 : The Queue is empty
bucket 3 : The Queue is empty
bucket 4 : The Queue is empty
bucket 5 : Front-->(512)->NULL
bucket 6 : Front-->(667)->NULL
bucket 7 : The Queue is empty
bucket 8 : The Queue is empty
bucket 9 : The Queue is empty
 - - - - - - - - - - - - - - - - -
 H A S H - T A B L E - 1
bucket 0 : Front-->(4)->(7)->(18)->(21)->(128)->(512)->(667)->NULL
bucket 1 : The Queue is empty
bucket 2 : The Queue is empty
bucket 3 : The Queue is empty
bucket 4 : The Queue is empty
bucket 5 : The Queue is empty
bucket 6 : The Queue is empty
bucket 7 : The Queue is empty
bucket 8 : The Queue is empty
bucket 9 : The Queue is empty
Enter output file name (e.g. output.txt): output.txt
thanks you entered: output.txt

Program ended with exit code: 0
```

Note to Future Self:
* How to make this program run fast.
Dynamic Heap allocation slows things down. Is it possible, to run RadixSort only on the Stack? Yes, it is. The only problem we are facing is the unknown size of the data. So run a different program that iterates through the data to learn its size. This preprocessing program uses a single incrementing variable that counts the data. Then once the data size is known, recompile RadixSort specifically for that size—allocate ALL data structures on the Stack.

Radix Sort algorithm steps

Step 0: Establish **Hash-Function** and **Hash-Table** of a given Size

Hash-Function ← identity function. The Hash-Function will insert the elements of data according to digit-index **r**, where the value of **r** starts incrementing from the right hand side of each number—i.e. from the least significant digit to the most significant one.

size ← 10 meaning the **Hash-Table size**. E.g. for sorting numerical data (i.e. digits 0..9) we only need 10 buckets.

Get the to be sorted data e.g. from *input.txt* file or some other source

data ← load data

iterations ← e.g 4 while loading data determine its largest value, e.g. 667 → 3 digits so 3+1=4 iterations

data_size ← determine while loading data

Hash-Table₀ ← array of Queues (as Linked List) (FIFO)
Hash-Table₁ ← array of Queues (as Linked List) (FIFO)

h ← 0 which Hash-Table to use, where 0 ≤ h ≤ 1
r ← 1 index of the **right most digit**, where **1 ≤ r ≤ iterations**
i ← 1 index of the data element, where **1 ≤ i ≤ data_size**

Step 1: Extract data one by one.
dataᵢ ← data

Step 2: Get the Hash-Table **bucket** (index) telling you where to store **dataᵢ**.
bucket ← Hash-Function(dataᵢ[r])

Step 3: Store **dataᵢ** into **Hash-Table-0** at the derived **bucket**.
insert(Hash-Table₀[bucket] , dataᵢ)
i++

Step 4: Repeat step 1 to step 3 until no more data (**i=data_size**)

Step 5: Update the right most digit **r** .
r ← r + 1

Step 6: Extract data one by one from **Hash-Table-h**.
dataᵢ ← Hash-Tableₕ

Step 7: Get the **bucket** (index) telling you where to store **dataᵢ**.
bucket ← Hash-Function(dataᵢ[r])

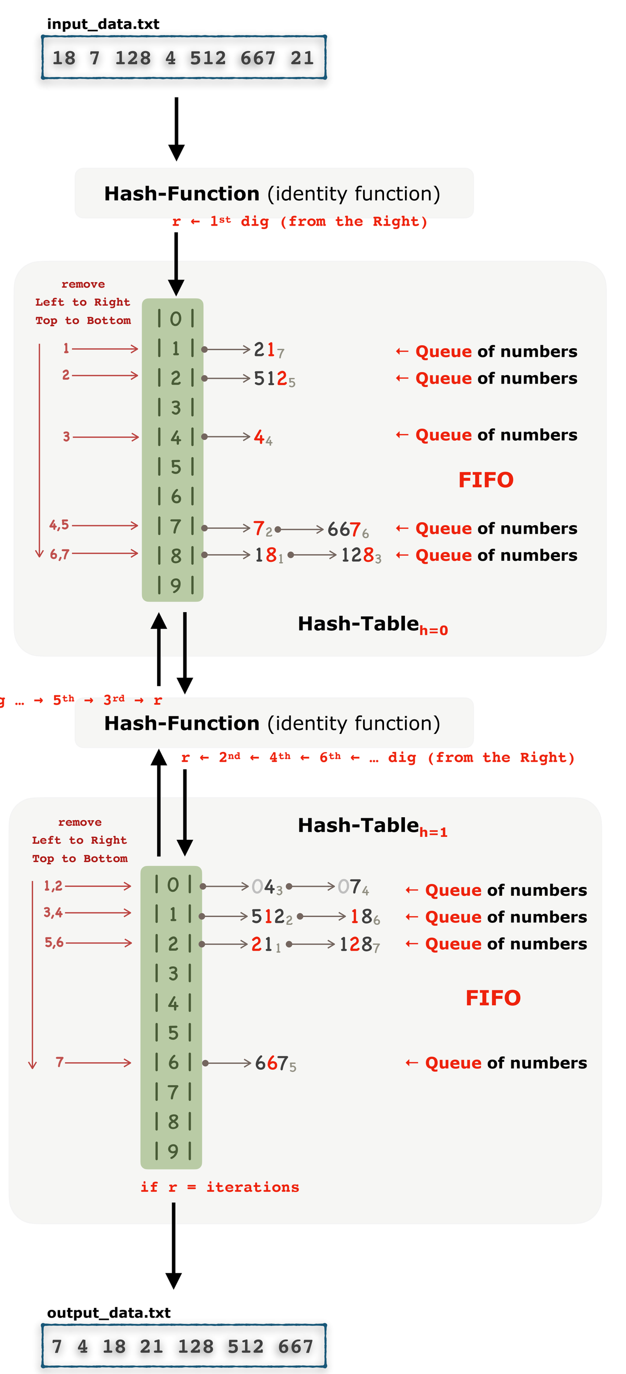
Step 8: Store **dataᵢ** into **Hash-Table-h** at the derived **bucket**.
h ← ((h+1) mod 2) i.e. switch to destination table
insert(Hash-Tableₕ[bucket] , dataᵢ)
h ← ((h+1) mod 2) i.e. switch back to source table
i++

Step 9: Repeat step 6 to step 8 until no more data (**i=data_size**)

Step 10: Update the right most digit **r** and switch to the other Hash-Table **h**.
r ← r + 1
h ← ((h+1) mod 2) i.e. destination table becomes source table

Step 11: Repeat step 6 to step 10 until **r = iterations**

Step 12: Return the sorted data.



The above section of the notes are based on the lectures given by Dr. Tsaiyun Phillips in Fall 2018, I was an undergraduate student at the time.