COMP2421—DATA STRUCTURES AND ALGORITHMS

Hashing

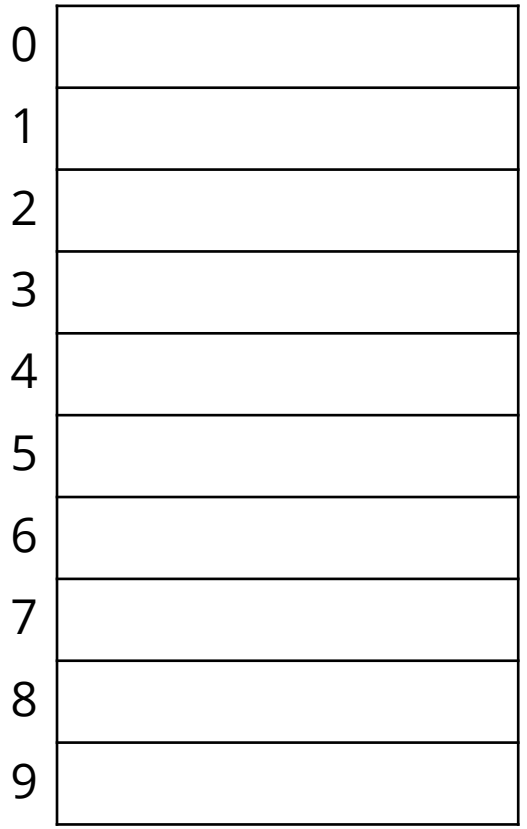
Dr. Radi Jarrar
Department of Computer Science
Birzeit University

Hashing

- A method to save/retrieve data quickly.
- Idea: have an array of some fixed size. The array contains the keys. Each key is associated with a value.
- The indexing/retrieval is done via **Hash Function**
- Hash function: should be simple enough to compute and should ensure that any 2 distinct keys get different cells in the array.
- E.g., data: 16, 1, 30, 19, 163, 677, 328
  - Array size of 10

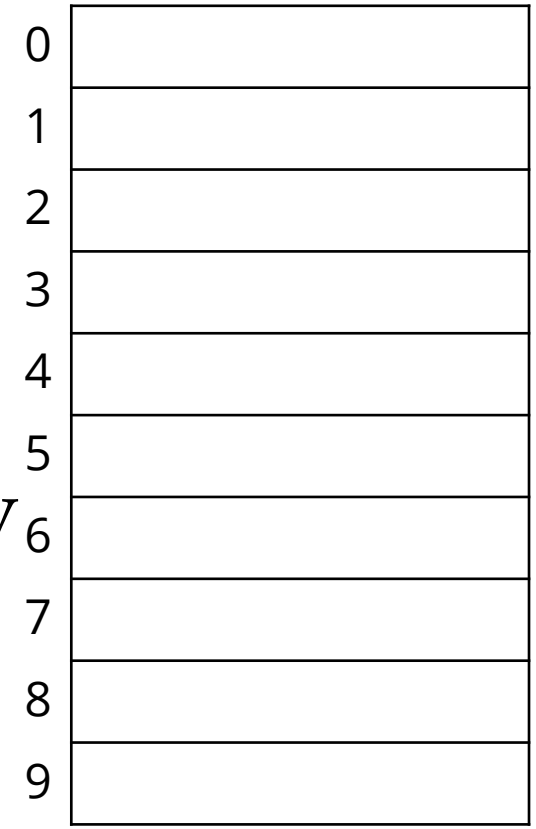
Hashing

- E.g., data: 16, 1, 30, 19, 163, 677, 328
  - Array size of 10
  - Hash function: h(x) = x % 10, where x is key; and 10 is the table size



Hashing

- There is a finite number of cells & infinite supply of keys.
- E.g., 10, 20, 30, 40, 50, 100, 1000, …
  - If h(x) = x % 10 → this would cause collision!
- Collision: the case in which a newly inserted key maps to an already occupied slot in the hash table.
- In this case, 10 is a bad choice. A good strategy is to have the size of table as a Prime Number.



Hashing

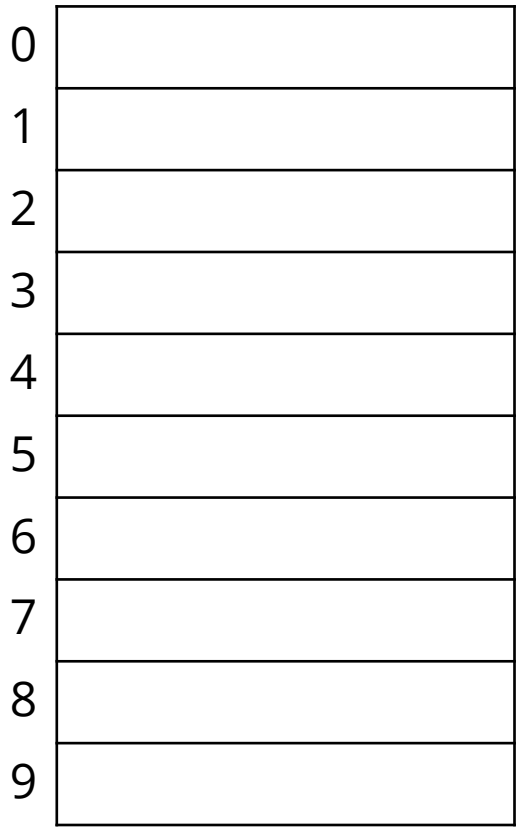
- Two things to take into consideration to solve collision:

1. The best function that gives the least number of collisions.
2. Once a collision encountered, what is the best solution.

- Solving collision can be done through Separate chaining & Open addressing.

Separate Chaining

- Keep a list of all elements that hash to the same value
- Array of linked lists
- E.g., data: 11, 21, 35, 4, 6, 46, 56, 7, 147, 99 and h(x) = x%10



Separate Chaining (2)

- Pros:
  - Easy to reach for keys (find)
  - Divides data into groups
- Cons:
  - Slow (linked list, pointers, …)
  - With larger number of collisions, it gets slower

- Avoid to use if the space is tight!

Open Addressing

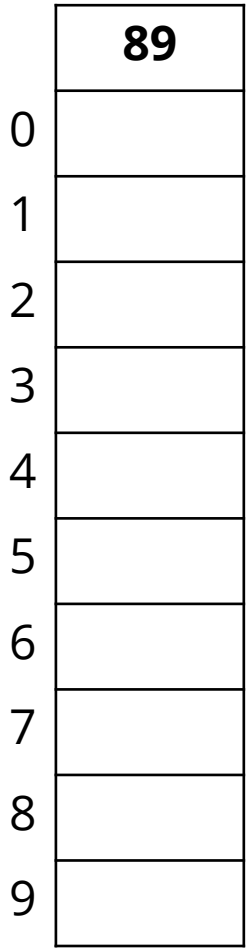
- Separate chaining requires pointers which slows the algorithm a bit because of the time required to allocate new cells
- Open addressing is an alternative to resolving the collision with linked lists
- Strategy: once a collision occurs, alternative cells are tried until an empty cell is found
- H_i(x) = (Hash(x) + F(i)) mod TableSize

Open Addressing (2)

- Because all data goes inside the table, a bigger table is needed for open addressing hashing than for separate chaining
- Types:
  - Linear hashing
  - Quadratic hashing
  - Double hashing

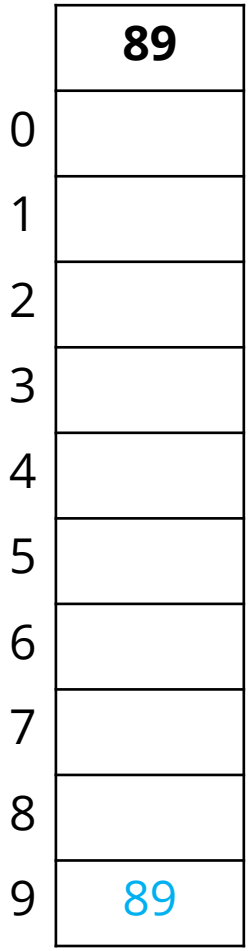
Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69



Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69



Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		
9	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that 0 ≤ i ≤ size
- E.g., 89, 18, 49, 58, 69

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		18
9	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69

	89	18	**49**
0			
1			
2			
3			
4			
5			
6			
7			
8		18	18
9	89	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69

	89	18	**49**
0			49
1			
2			
3			
4			
5			
6			
7			
8		18	18
9	89	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69

	89	18	49	**58**
0			49	49
1				
2				
3				
4				
5				
6				
7				
8		18	18	18
9	89	89	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq size$
- E.g., 89, 18, 49, 58, 69

	89	18	49	**58**
0			49	49
1				58
2				
3				
4				
5				
6				
7				
8		18	18	18
9	89	89	89	89

Linear Hashing (Linear probing)

- Try all cells sequentially with wraparounds in search of an empty cell.
- h(x) = (function + i) % size such that $0 \leq i \leq$ size
- E.g., 89, 18, 49, 58, 69

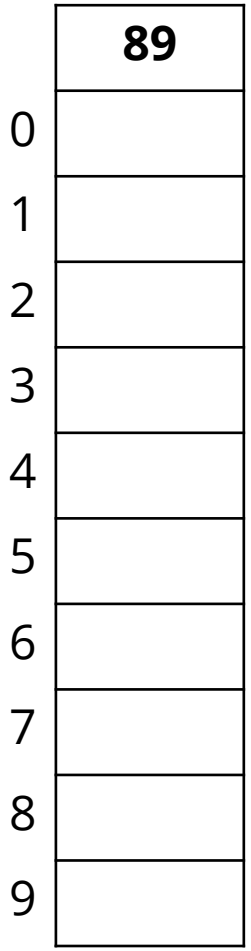
	89	18	49	58	**69**
0			49	49	49
1				58	58
2					
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Linear Hashing (Linear probing)

- As long as the table is big enough, a free cell can always be found, BUT the time to do so gets quite large.
- Another problem: Primary clustering
  - Blocks of occupied cells start forming up.
  - This means that any key that hashes into the cluster will require several attempts to resolve the collision & then it will add to the cluster

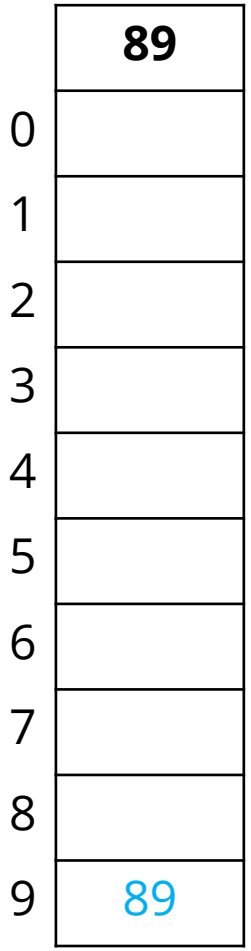
Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`



Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`



Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		
9	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		18
9	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	**49**
0			
1			
2			
3			
4			
5			
6			
7			
8		18	18
9	89	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	**49**
0			49
1			
2			
3			
4			
5			
6			
7			
8		18	18
9	89	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	49	**58**
0			49	49
1				
2				
3				
4				
5				
6				
7				
8		18	18	18
9	89	89	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	49	**58**
0			49	49
1				
2				58
3				
4				
5				
6				
7				
8		18	18	18
9	89	89	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	49	58	**69**
0			49	49	49
1					
2				58	58
3					
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic Hashing

- `h(x) = (function + i²) % size` such that `0 ≤ i ≤ size`
- E.g., `89, 18, 49, 58, 69`

	89	18	49	58	**69**
0			49	49	49
1					
2				58	58
3					69
4					
5					
6					
7					
8		18	18	18	18
9	89	89	89	89	89

Quadratic Hashing

- This method eliminates the primary clustering problem of linear probing.
- $F(i) = i^2$
- For linear probing, it is bad to get the table nearly full because the performance will degrade.
- For quadratic, there is no guarantee of finding an empty cell once the table gets more than half full or before getting half full if the table size is not prime. This is because at most half of the table can be used as alternative locations to resolve collisions.

Quadratic Hashing

- If the table is half empty and the table size is prime, we are always guaranteed to be able to insert a new element.
- It is crucial that the table size be prime.
- If the table size is not prime, the number of alternative locations can be severely reduced.

Quadratic Hashing

- E.g., If the table size = 16, so alternative locations will be distances 1, 4, or 9 away.
- $h(x) = (x \% 16) + i^2$ values: 0, 1, 4, 9, 16

Index	0	1	2	3	4	5	6	7	8	9	10	11	12
	0	1			4					9			

- Hash size = first prime > size * 2
  - Such that h(x) = x % hash size

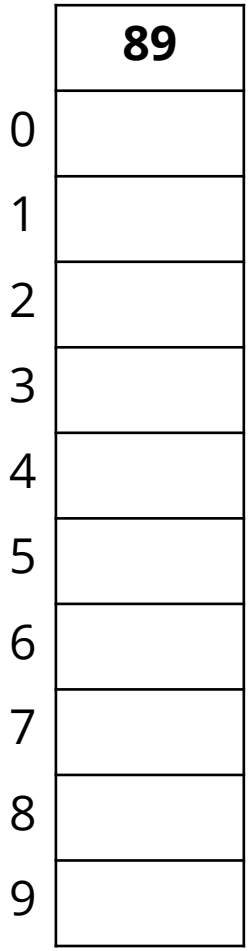
Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$ $0 \leq i \leq size$
- It means we apply a second hash function to x and probe at a distance $h_2(x)$, $2h_2(x)$, …, and so on.
- $h_2(x)$ should be chosen carefully. The function must never evaluate to zero.
- A function such as $h_2(x) = R – (x \% R)$ where R is a prime smaller than TableSize will work properly.

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

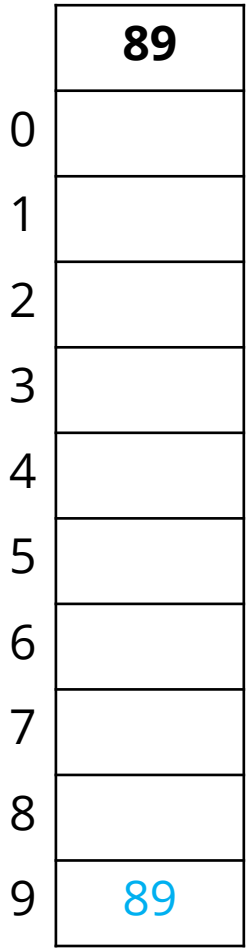
Hash size = 10, prime < 10 = 7



Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7



Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		
9	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	**18**
0		
1		
2		
3		
4		
5		
6		
7		
8		18
9	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	**49**
0			
1			
2			
3			
4			
5			
6			
7			
8		18	18
9	89	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	**49**
0			
1			
2			
3			
4			
5			
6			49
7			
8		18	18
9	89	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	49	**58**
0				
1				
2				
3				
4				
5				
6			49	49
7				
8		18	18	18
9	89	89	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	49	**58**
0				
1				
2				
3				58
4				
5				
6			49	49
7				
8		18	18	18
9	89	89	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	49	58	**69**
0					
1					
2					
3				58	58
4					
5					
6			49	49	49
7					
8		18	18	18	18
9	89	89	89	89	89

Double Hashing

- $h(x) = h_1(x) + i * h_2(x)$
- $h_2(x) = 7 - (x \% 7)$

Hash size = 10, prime < 10 = 7

	89	18	49	58	**69**
0					69
1					
2					
3				58	58
4					
5					
6			49	49	49
7					
8		18	18	18	18
9	89	89	89	89	89

Rehashing

- If the table gets too full, the running time of the operations will start taking too long and insert might fail for open addressing hashing with quadratic resolution.
- Solution: Build another table that is about twice the size and scan down the entire original hash table, computing new hash value for each element and insert it in the new table.
- New size = the first prime > (old size * 2)

Rehashing

- Hash function: if the input keys are integers, then returning key mod TableSize is a reasonable strategy.
- The hash function has to be carefully considered
  - → if the table size is 10 and the keys end with zeros, then key % 10 is a bad choice.
  - Good idea: is to ensure that the table size is prime
    - It distribute keys evenly

Searching for Elements

- When searching for an entry, the table is scanned the same sequence as the collision was solved until either the target record is found or an unused array slot is found, which indicates that there is no such key in the table.