HASHING

Algorithms and Data Structures 2
Exercise – 2023W

Martin Schobesberger, Markus Weninger, Markus Jäger,
Florian Beck, Achref Rihani

Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at

HASHING :: MOTIVATION

Initial problem example: Storage student IDs for an exam with direct access **(=O(1))**.

- Each ID has 8 digits → ID is used as index in the array (direct access)
- Array with a size of **100 million** elements would be necessary for direct access (00 000 000 – 99 999 999).
- If there are 100 IDs to store, **100 indices of 100 million are needed – the rest is unused**.

HASHING :: MOTIVATION

Initial problem example: Storage student IDs for an exam with direct access.

- Each ID has 8 digits → ID is used as index in the array (direct access)
- Array with a size of **100 million** elements would be necessary for direct access (00 000 000 – 99 999 999).
- If there are 100 IDs to store, **100 indices of 100 million are needed – the rest is unused**.

Aim

- Map keys to specific range so that elements in the list can be accessed using an index.
- e.g. 00 000 000 = index 0; 11 222 334 = index 1; 99 999 999 = index 99
- **Best case O(1)**.

HASHING :: MOTIVATION

Initial problem example: Storage student IDs for an exam with direct access.

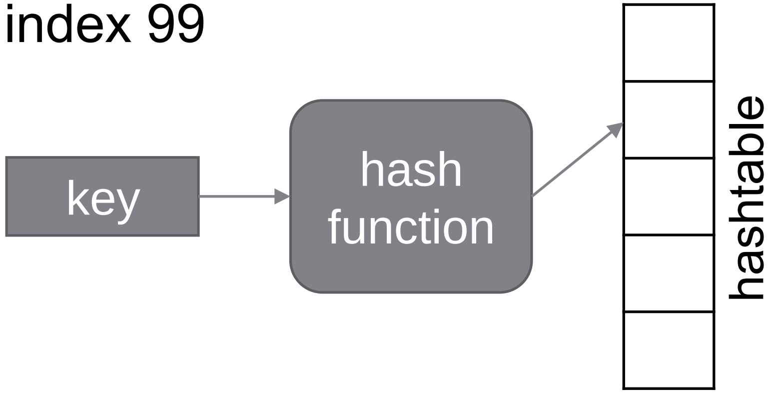
- Each ID has 8 digits → ID is used as index in the array (direct access)
- Array with a size of **100 million** elements would be necessary for direct access (00 000 000 – 99 999 999).
- If there are 100 IDs to store, **100 indices of 100 million are needed – the rest is unused**.

Aim

- Map keys to specific range so that elements in the list can be accessed using an index.
  e.g. 00 000 000 = index 0; 11 222 334 = index 1; 99 999 999 = index 99
- **Best case O(1)**.

Hashing: Compromise between time and memory requirements

- No time problem: sequential search.
- No memory problem: use keys as memory addresses.



HASHING :: PRINCIPLE

Algorithms based on hashing consist basically of 2 parts:

1. **Transformation** of key *k* (must be unique) into a table address (from the set of possible hashes K)

   `h: k → {0, …, N-1}` … N should be a prime number (→ equal distribution → see example later)

2. **Collision avoidance**
   - Transformation might result in same index for different keys
   - Store elements at different positions

HASHING :: PRINCIPLE

For this exercise we use the modulo operation (%) as hash function.

- However, it is also possible to define other hash functions…

Example

hash(k) = k % N

  - Hashtable (n = 7), in which Integer values are stored

0	1	2	3	4	5	6

HASHING :: PRINCIPLE

Example

hash(k) = k % N

- Hashtable (n = 7), in which Integer values are stored

0	1	2	3	4	5	6

- *insert(11):* hash(11) = 11 % 7 = 4 → insert 11 at index position 4

0	1	2	3	4	5	6
				11		

HASHING :: PRINCIPLE

Example (cont'd):

hash(k) = k % N

- insert(27): hash(27) = 27 % 7 = 6

0	1	2	3	4	5	6
				11		27

HASHING :: PRINCIPLE

Example (cont'd):

hash(k) = k % N

- insert(27): hash(27) = 27 % 7 = 6

0	1	2	3	4	5	6
				11		27

- insert(21): hash(21) = 21 % 7 = 0

0	1	2	3	4	5	6
21				11		27

HASHING :: PRINCIPLE

Example (cont'd):

hash(k) = k % N

- insert(27): hash(27) = 27 % 7 = 6

0	1	2	3	4	5	6
				11		27

- insert(21): hash(21) = 21 % 7 = 0

0	1	2	3	4	5	6
21				11		27

- insert(18): hash(18) = 18 % 7 = 4

0	1	2	3	4	5	6
21				11		27

→ at position 4 there is already an entry (collision)

HASHING :: PRINCIPLE

Example (cont'd):

hash(k) = k % N

- insert(27): hash(27) = 27 % 7 = 6

0	1	2	3	4	5	6
				11		27

- insert(21): hash(21) = 21 % 7 = 0

0	1	2	3	4	5	6
21				11		27

- insert(18): hash(18) = 18 % 7 = 4

0	1	2	3	4	5	6
21				11		27

→ at position 4 there is already an entry (collision)

We discuss
- **Chaining and**
- **3 types of Open Adressing**

HASHING :: RESOLVING COLLISIONS

1. **Chaining**

Overflow chains are attached at the corresponding index positions.

Each element is a reference to an overflow chain:

- Table can never overflow.
- Long chains behave like list processing without direct access (→ worse performance).
- Searching of a key:
  - Calculate hash(k) and search until the end of the corresponding chain is reached.
- Insertion of a key:
  - Calculate hash(k), iterate and append to the end of the corresponding chain if not already contained.
- Removal of a key:
  - Calculate hash(k), search and remove from list if found.

Example

- Insert of 11, 27, 21, 18, 32, 44, 55 in hashtable
- using N=7 and same hash function as before: hash(k) = k % N

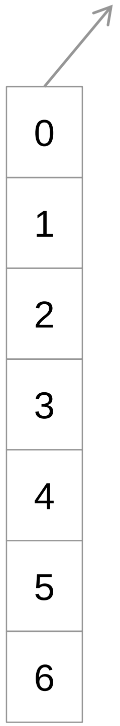
HASHING :: CHAINING

Hash values

insert: 11, 27, 21, 18, 32, 44, 55

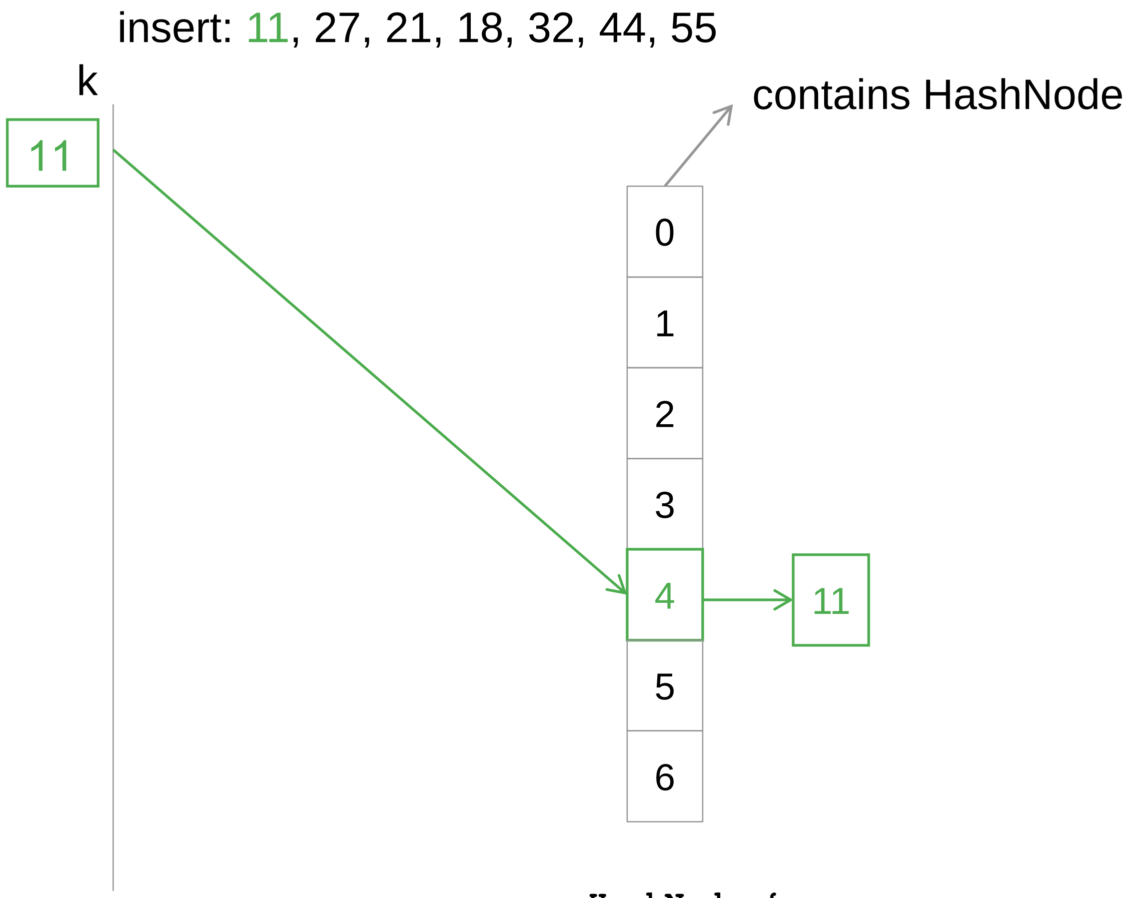
k

contains HashNode




```
HashNode {
   int key
   HashNode next
}
```

HASHING :: CHAINING



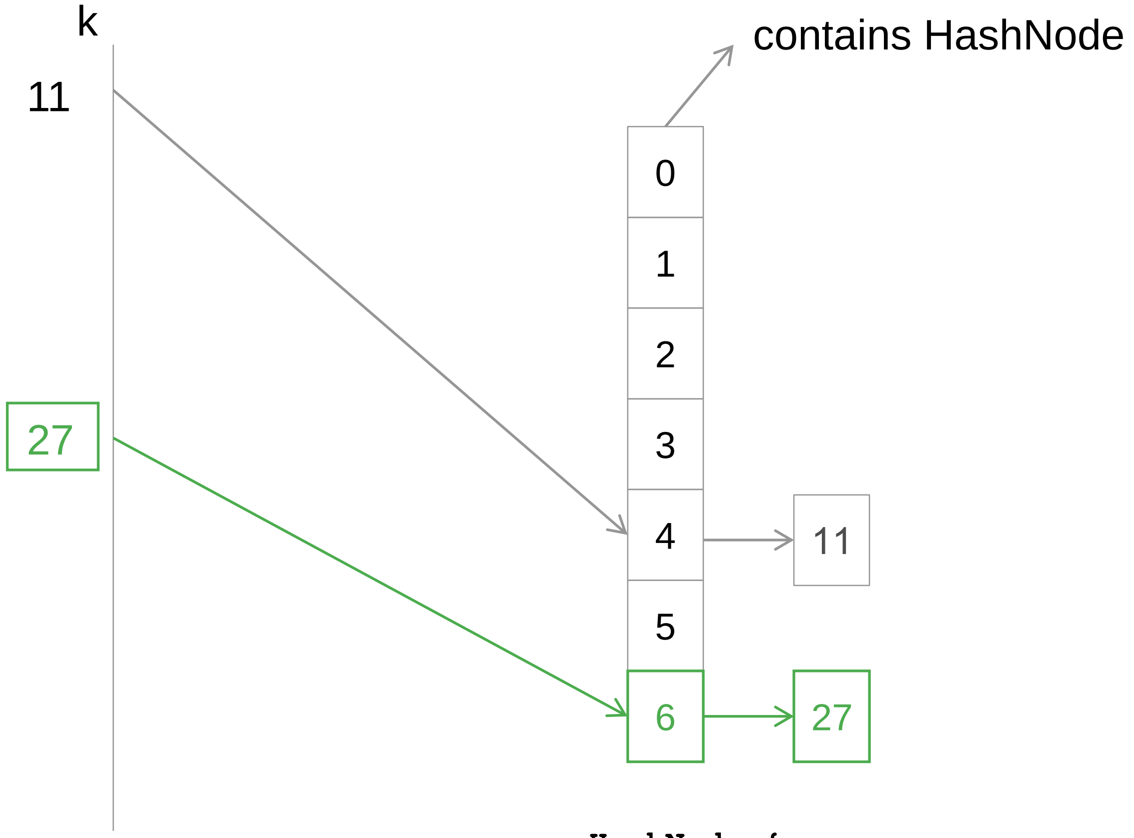

```
HashNode {
    int key
    HashNode next
}
```

Hash values

`hash(11) = 11%7 = 4`

HASHING :: CHAINING

insert: 11, 27, 21, 18, 32, 44, 55



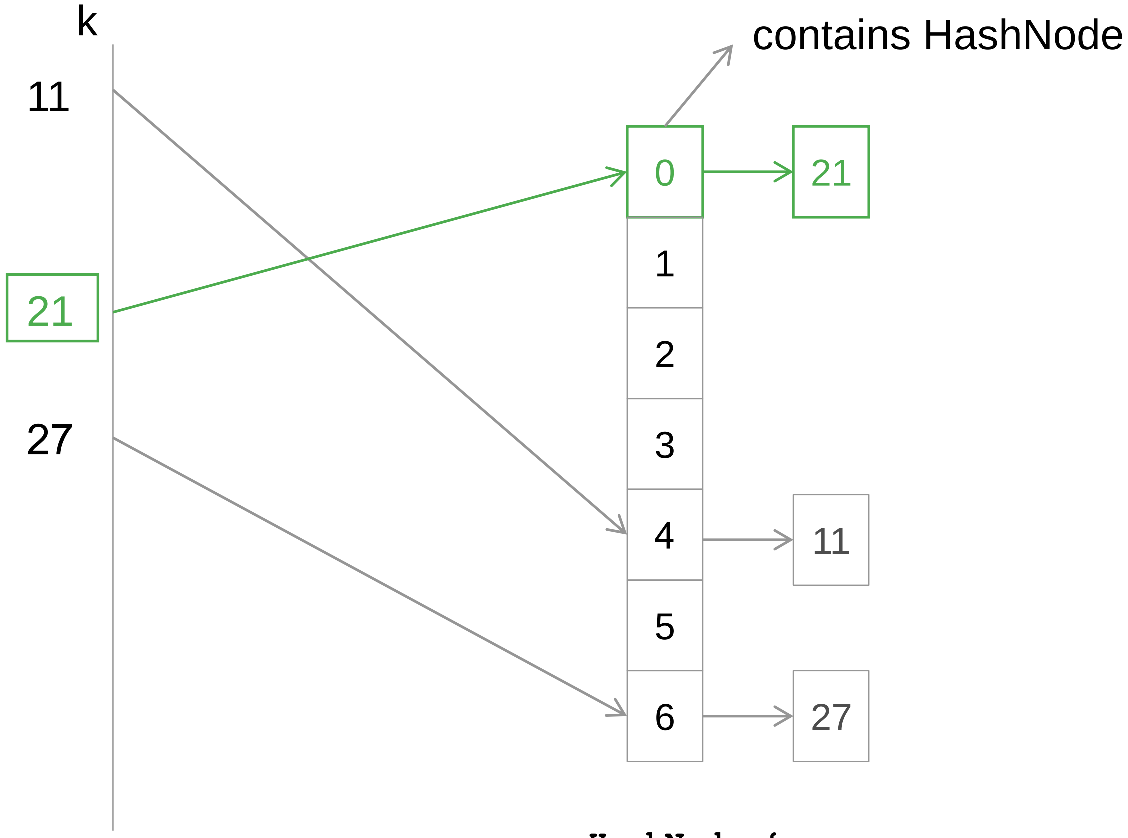

```
HashNode {
    int key
    HashNode next
}
```

```
Hash values

hash(11)  = 11%7  = 4
hash(27)  = 27%7  = 6
```

HASHING :: CHAINING

insert: 11, 27, 21, 18, 32, 44, 55

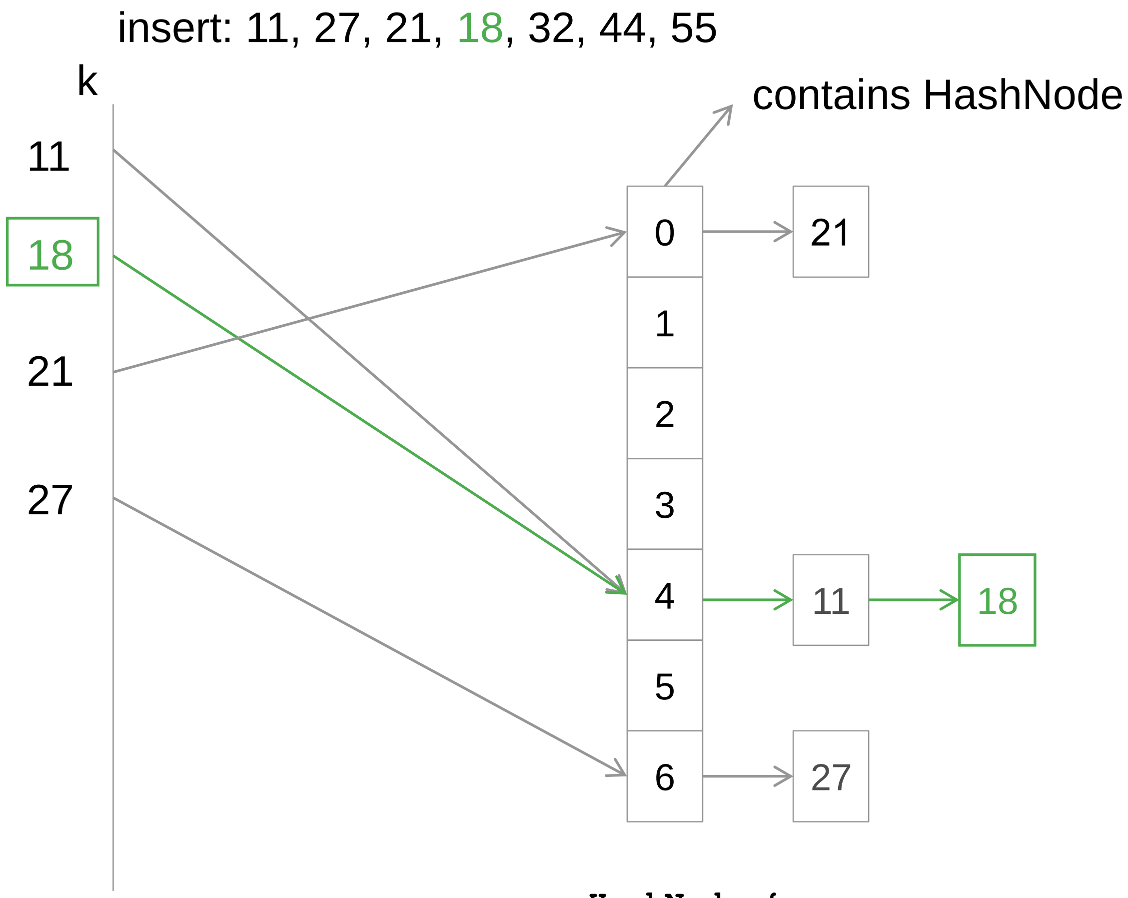



```
HashNode {
   int key
   HashNode next
}
```

```
Hash values

hash(11) = 11%7 = 4
hash(27) = 27%7 = 6
hash(21) = 21%7 = 0
```

HASHING :: CHAINING

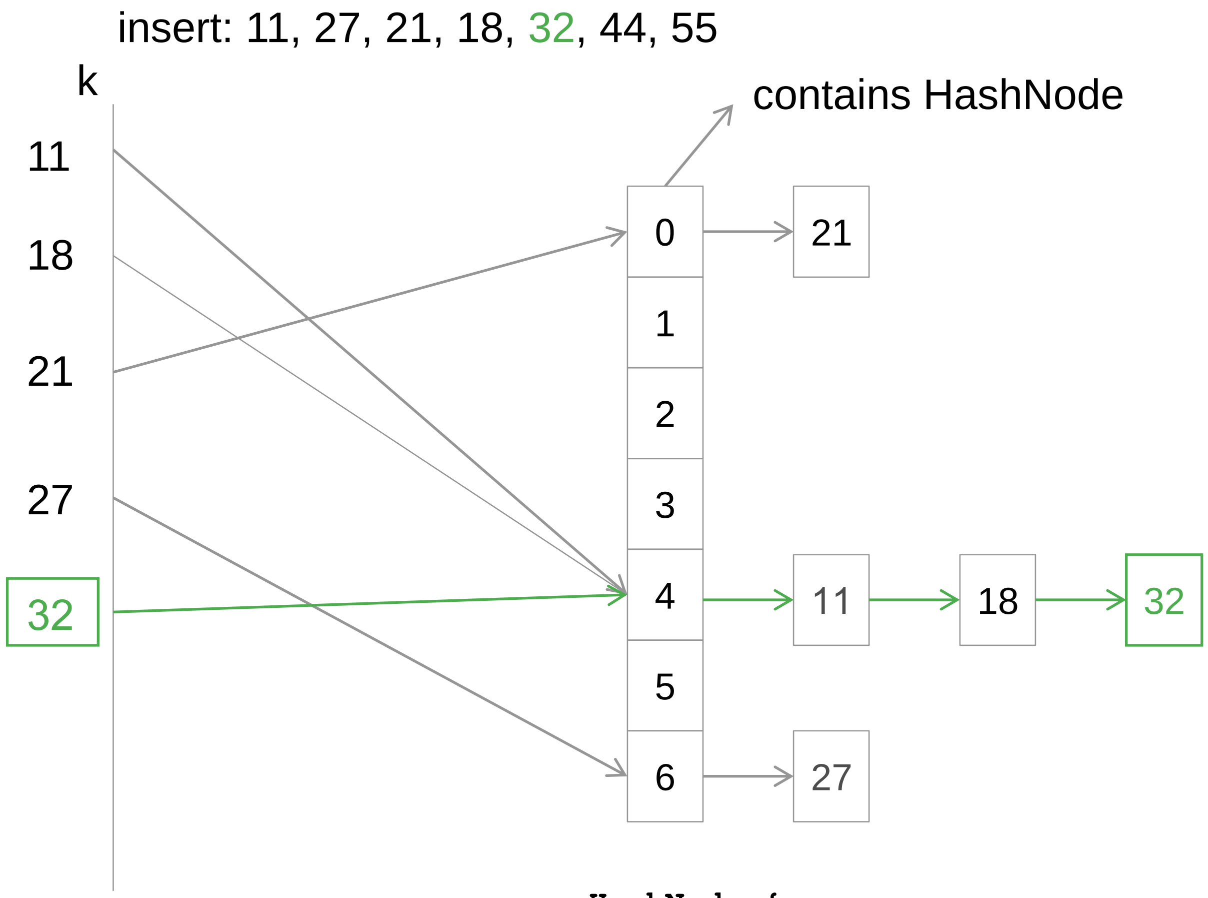



```
HashNode {
    int key
    HashNode next
}
```

```
Hash values

hash(11) = 11%7 = 4
hash(27) = 27%7 = 6
hash(21) = 21%7 = 0
hash(18) = 18%7 = 4
```

HASHING :: CHAINING

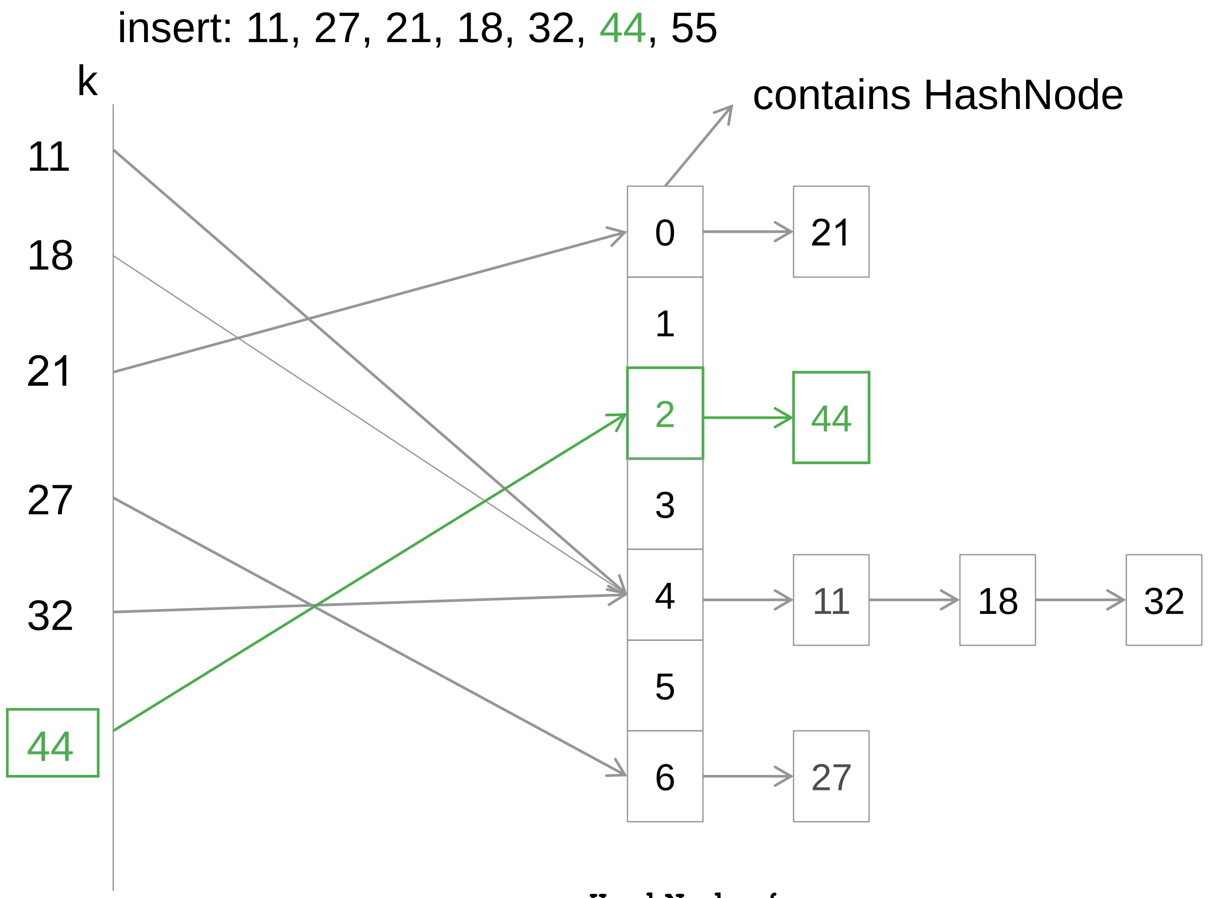



```
HashNode {
    int key
    HashNode next
}
```

Hash values

```
hash(11)  = 11%7  = 4
hash(27)  = 27%7  = 6
hash(21)  = 21%7  = 0
hash(18)  = 18%7  = 4
hash(32)  = 32%7  = 4
```

HASHING :: CHAINING



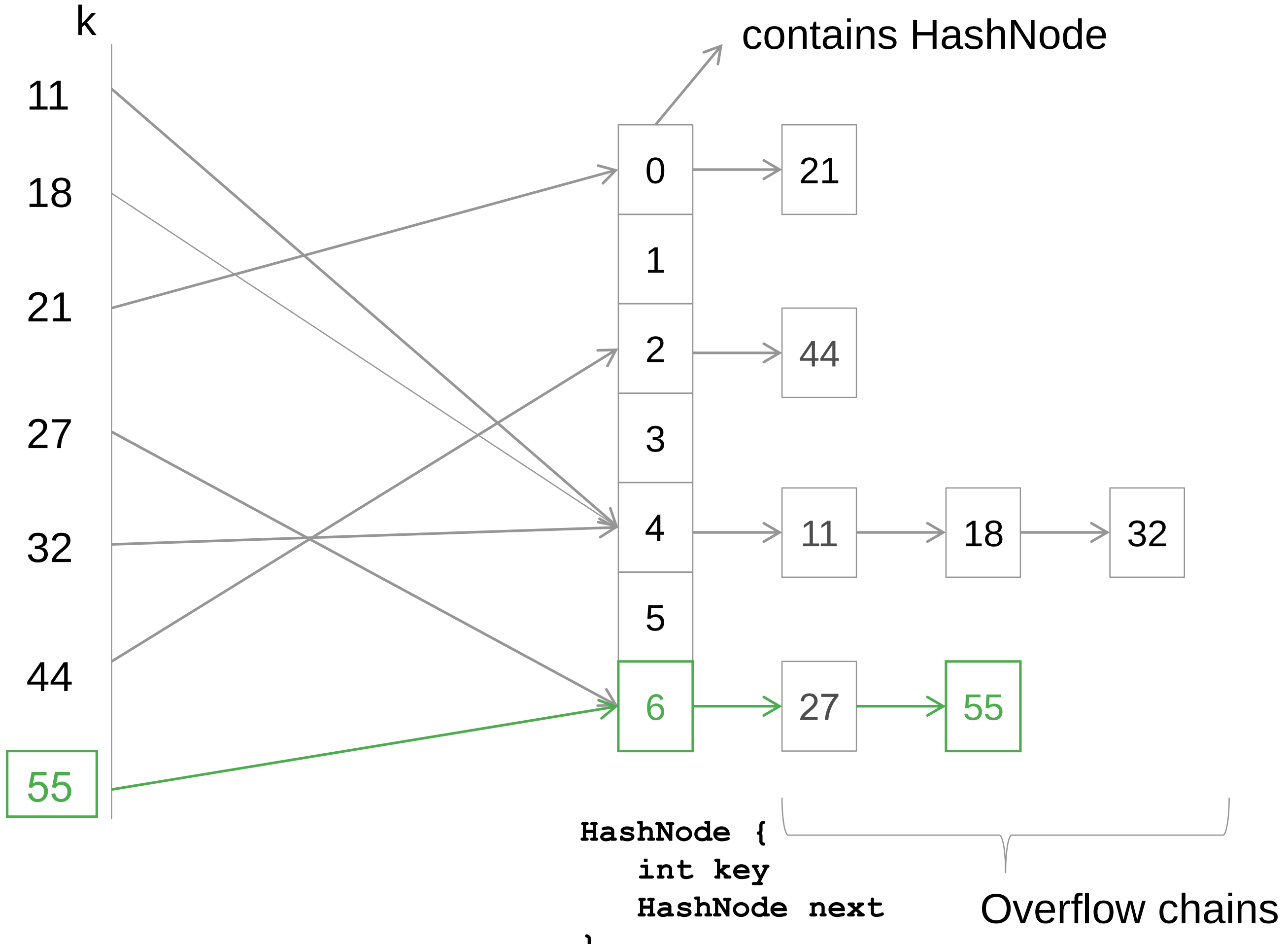

```
HashNode {
    int key
    HashNode next
}
```

```
Hash values

hash(11) = 11%7 = 4
hash(27) = 27%7 = 6
hash(21) = 21%7 = 0
hash(18) = 18%7 = 4
hash(32) = 32%7 = 4
hash(44) = 44%7 = 2
```

HASHING :: CHAINING

insert: 11, 27, 21, 18, 32, 44, 55




```
HashNode {
    int key
    HashNode next
}
```

```
Hash values

hash(11)  = 11%7  = 4
hash(27)  = 27%7  = 6
hash(21)  = 21%7  = 0
hash(18)  = 18%7  = 4
hash(32)  = 32%7  = 4
hash(44)  = 44%7  = 2
hash(55)  = 55%7  = 6
```

HASHING :: RESOLVING COLLISIONS

2. **Open addressing**

Overflows are stored at vacant positions in the hashtable:

- The sequence of the positions considered is referred to as the **probing sequence.**
- Follow this probing sequence until the first vacant position is found.

In the exercise we discuss 3 methods of open addressing:

2a) linear probing
2b) quadratic probing
2c) double hashing

HASHING :: LINEAR PROBING

Principle:

- If the calculated position is already occupied, move 1 element to the right / left (keep moving direction!) [→ **probing sequence**], until
  - a vacant position is found, or
  - the original element is found again (→ table is full)

Pseudo code:

```
insert(key)
  h = hash function(key)
  while(occupied(hashtable[h]))  //collision
     h = hash function(h + 1)
     if(h == original index) return

  hashtable[h] = key // insert key at first vacant position
```


M. Schobesberger, M. Weninger, M. Jäger, F. Beck, A. Rihani

HASHING :: LINEAR PROBING

Example:

0	1	2	3	4	5	6
21				11		27

insert(18): hash(18) = 18%7 = 4 → 1st collision

0	1	2	3	4	5	6
21				11		27

HASHING :: LINEAR PROBING

Example:

0	1	2	3	4	5	6
21				11		27

insert(18): hash(18) = 18%7 = 4 → 1st collision

0	1	2	3	4	5	6
21				11		27

→ (4+1) % 7 = 5 → OK

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): hash(32) = 32%7 = 4 → 1st collision

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): hash(32) = 32%7 = 4 → 1^{st} collision
→ (4+1) % 7 = 5 → 2^{nd} collision

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): hash(32) = 32%7 = 4 → 1st collision
→ (4+1) % 7 = 5 → 2nd collision
→ (5+1) % 7 = 6 → 3rd collision

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): hash(32) = 32%7 = 4 → 1^{st} collision
→ (4+1) % 7 = 5 → 2^{nd} collision
→ (5+1) % 7 = 6 → 3^{rd} collision
→ (6+1) % 7 = 0 → 4^{th} collision

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): hash(32) = 32%7 = 4 → 1st collision
→ (4+1) % 7 = 5 → 2nd collision
→ (5+1) % 7 = 6 → 3rd collision
→ (6+1) % 7 = 0 → 4th collision
→ (0+1) % 7 = 1 → OK

0	1	2	3	4	5	6
21	32			11	18	27

Primary clustering

HASHING :: LINEAR PROBING

Example: *remove(18)*

Begin with search (like for insert):

hash(18)=4

(1) If the element to be removed is at the first position → *remove*

(2) If not→ search along the probing sequence until empty element is found, or entire table has been traversed.

0	1	2	3	4	5	6
21	32			11	18	27

HASHING :: LINEAR PROBING

Example: *remove(18)*

Begin with search (like for insert):

hash(18)=4

(1) If the element to be removed is at the first position → *remove*

(2) If not→ search along the probing sequence until empty element is found, or entire table has been traversed.

0	1	2	3	4	5	6
21	32			11	18	27

0	1	2	3	4	5	6
21	32			11	~~18~~	27

HASHING :: LINEAR PROBING

Example: *remove(18)*

Begin with search (like for insert):

hash(18)=4

(1) If the element to be removed is at the first position → *remove*

(2) If not→ search along the probing sequence until empty element is found, or entire table has been traversed.

0	1	2	3	4	5	6
21	32			11	18	27

0	1	2	3	4	5	6
21	32			11	~~18~~	27

0	1	2	3	4	5	6
21	32			11		27

HASHING :: LINEAR PROBING

Example: *contains(32)*

- hash(32)=4
- Element is not found at calculated position.
- Search along the probing sequence (4+1)%7=5
- Position ‚5‘ is empty. Search is terminated, although element 32 is stored in list!

0	1	2	3	4	5	6
21	32			11		27

HASHING :: LINEAR PROBING

Example: *contains(32)*

- hash(32)=4
- Element is not found at calculated position.
- Search along the probing sequence (4+1)%7=5
- Position ‚5‘ is empty. Search is terminated, although element 32 is stored in list!

0	1	2	3	4	5	6
21	32			11		27

Problem: Keys with equal hash positions can be detached.

HASHING :: LINEAR PROBING

Example: *contains(32)*

- hash(32)=4
- Element is not found at calculated position.
- Search along the probing sequence (4+1)%7=5
- Position ‚5' is empty. Search is terminated, although element 32 is stored in list!

0	1	2	3	4	5	6
21	32			11		27

Problem: Keys with equal hash positions can be detached.

Solution: Differ between EMPTY and REMOVED elements
→ Status flag for each entry

```
HashNode {
    int key
    boolean removed
}
```

HASHING :: LINEAR PROBING

Initial situation

0	1	2	3	4	5	6
21	32			11	18	27
F	F	F	F	F	F	F

HASHING :: LINEAR PROBING

Initial situation

0	1	2	3	4	5	6
21	32			11	18	27
F	F	F	F	F	F	F

remove(18): 18%7 → (4+1)%7=5

→ found and mark position 5 as deleted

0	1	2	3	4	5	6
21	32			11		27
F	F	F	F	F	T	F

HASHING :: LINEAR PROBING

Initial situation

0	1	2	3	4	5	6
21	32			11	18	27
F	F			F	F	F

remove(18): 18%7 → (4+1)%7=5

→ found and mark position 5 as deleted.

0	1	2	3	4	5	6
21	32			11		27
F	F			F	T	F

contains(32): 32%7 = 4

Cancel search only if (1) EMPTY, (2) element is found, or (3) table is traversed entirely.

HASHING :: QUADRATIC PROBING

Principle:

- Instead of `(h + i) % N` we use $\mathbf{(h \pm i^2)\ \%\ N}$
- i.e.: for h = 4 we get
  - Quadratic: 4+1, 4-1, 4+4, 4-4, 4+9, 4-9, … (= 5, 3, 8, 0, …) instead of
  - Linear: $\underbrace{4+1,\ 4+2,\ 4+3,\ 4+4,\ 4+5,\ 4+6,}_{\%\ N}$ ... (= 5, 6, 7, 8, …)

HASHING :: QUADRATIC PROBING

Principle:

- Instead of `(h + i) % N` we use `(h ± i²) % N`
- i.e.: for h = 4 we get
  - Quadratic: 4+1, 4-1, 4+4, 4-4, 4+9, 4-9, … (= 5, 3, 8, 0, …) instead of
  - Linear: 4+1, 4+2, 4+3, 4+4, 4+5, 4+6, ... (= 5, 6, 7, 8, …)

% N

Example:

insert(32): 32 % 7 = 4 → collision

→ $(4 + 1^2)$ % 7 = 5 → collision

→ $(4 – 1^2)$ % 7 = 3 → 4 collisions before, now only 2 collisions

0	1	2	3	4	5	6
21			32	11	18	27

HASHING :: DOUBLE HASHING

Principle:

- Reduce clustering by placing different elements with **different step sizes**.
- Definition of the probing sequence by
  - $\texttt{hash}_1$: k → {0, 1, …, N-1} -> hash
  - $\texttt{hash}_2$: k → {1, …, N-1} → offset

Pseudo code:

```
insert(key)
  hash = hash function 1(key)
  offset = hash function 2(key)
  while(occupied(hashtable[hash]))  // collision
    hash = hash function 1(hash + offset)
    if(hash == original index) return
  hashtable[hash] = key // insert key at first vacant position
```

- It is possible to always use hash_2 and probing sequence
  - In this exercise, we use hash_2 and probing sequence only if hash_1 causes a collision

HASHING :: DOUBLE HASHING

Requirements for Double Hashing:

- h_2 must not return 0 (would result in an endless loop on first collision).
- The offset must be coprime to the table size; therefore, table size should be a prime number.

N=8; offset=4; hash=1

(1+4)%8 = 5

(5+4)%8 = 1

(1+4)%8 = 5

HASHING :: DOUBLE HASHING

Requirement *(cont'd):*

- The offset must be coprime to the table size; therefore, table size should be a prime number.

N=8; offset=4; h=1
(1+4)%8 = 5
(5+4)%8 = 1
(1+4)%8 = 5

N=7; offset=4; h=1
(1+4)%7 = 5
(5+4)%7 = 2
(2+4)%7 = 6
(6+4)%7 = 3
(3+4)%7 = 0
(0+4)%7 = 4
(4+4)%7 = 1
(1+4)%7 = 5

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

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

probing sequence: hash = hash1(hash+offset)

M. Schobesberger, M. Weninger, M. Jäger, F. Beck, A. Rihani

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

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

probing sequence: hash = hash1(hash+offset)

insert(14):
hash1(14)=14%13 = 1

0	1	2	3	4	5	6	7	8	9	10	11	12
	14											

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14											

probing sequence: hash = hash1(hash+offset)

insert(21):
hash1(21)=21%13 = 8

0	1	2	3	4	5	6	7	8	9	10	11	12
	14							21				

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14							21				

probing sequence: hash = hash1(hash+offset)

insert(1):
hash1(1)=1%13 = 1

0	1	2	3	4	5	6	7	8	9	10	11	12
	14							21				

M. Schobesberger, M. Weninger, M. Jäger, F. Beck, A. Rihani

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14							21				

probing sequence: hash = hash1(hash+offset)

insert(1):
hash1(1)=1%13 = 1 AND hash2(1)=1+(1%12)=2
probing sequence → **(1+2)%13 = 3**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1					21				

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1					21				

probing sequence: hash = hash1(hash+offset)

insert(19):
hash1(19)=19%13 = 6

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21				

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21				

probing sequence: hash = hash1(hash+offset)

insert(10):
hash1(10)=10%13 = 10

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10		

M. Schobesberger, M. Weninger, M. Jäger, F. Beck, A. Rihani

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10		

probing sequence: hash = hash1(hash+offset)

insert(11):
hash1(11)=11%13 = 11

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(6):
hash1(6)=6%13 = 6

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(6):
hash1(6)=6%13 = 6 AND hash2(6)=1+6%12=7
probing sequence → **(6+7)%13 = 0**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(42):
hash1(42)=42%13 = 3

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21			11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(42):
hash1(42)=42%13 = 3 AND hash2(42)=1+42%12=7
probing sequence → (3+7)%13 = 10

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1			19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(42):
hash1(42)=42%13 = 3 AND hash2(42)=1+42%12=7
probing sequence → (3+7)%13 = 10
probing sequence → **(10+7)%13 = 4**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(8):
hash1(8)=8%13 = 8

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1			19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(8):
hash1(8)=8%13 = 8 AND hash2(8)=1+8%12=9
probing sequence → (8+9)%13 = 4

0	1	2	3	4	5	6	7	8	9	10	11	12
	14		1	42		19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(8):
hash1(8)=8%13 = 8 AND hash2(8)=1+8%12=9
probing sequence → (8+9)%13 = 4
probing sequence → (4+9)%13 = 0

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

HASHING :: DOUBLE HASHING

Example:

N=13
hash1(k) = k%N **-> hash**
hash2(k) = 1 + k%(N-1) → **offset**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21		10	11	

probing sequence: hash = hash1(hash+offset)

insert(8):
hash1(8)=8%13 = 8 AND hash2(8)=1+8%12=9
probing sequence → (8+9)%13 = 4
probing sequence → (4+9)%13 = 0
probing sequence → **(0+9)%13 = 9**

0	1	2	3	4	5	6	7	8	9	10	11	12
6	14		1	42		19		21	8	10	11	

ASSIGNMENT 03

HASHING

Algorithms and Data Structures 2
Exercise – 2023W

Martin Schobesberger, Markus Weninger, Markus Jäger, Florian Beck, Achref Rihani

Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

JOHANNES KEPLER UNIVERSITY LINZ
Altenberger Straße 69
4040 Linz, Austria
jku.at