HASHING

Stefan Grünberger, Martin Schobesberger
Dari Trendafilov, Markus Weninger
Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

Algorithms and Data Structures 2
Exercise – 2021W

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

HASHING :: MOTIVATION

Initial problem example: Storage of 100 student IDs for an exam with direct access.

- Array with a size of 100 million elements would be necessary for direct access (00 000 000 – 99 999 999).
- Remedy: Map the keys to the desired range (e.g., index 0 - 99).
- Problem: Mapping is not unambiguous → handling of collisions.

Aim

- Map keys to specific value range so that elements in the list can be accessed using an index.
- **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)

   Perfect/Injective hashing:
   |***K***| >= number of search keys k to be stored
   |***K***| <= number N of available memory cells

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…
- h(k) = k % N

Example

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

0	1	2	3	4	5	6

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

0	1	2	3	4	5	6
				11		

HASHING :: PRINCIPLE

Example (cont'd):

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

0	1	2	3	4	5	6
				11		27

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

0	1	2	3	4	5	6
21				11		27

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

0	1	2	3	4	5	6
21				11		27

→ at position 4 there is already an entry (collision)
→ different strategies for collision avoidance

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 h(k) and search until the end of the corresponding chain is reached.
- Insertion of a key:
  - Calculate h(k) and append to the end of the corresponding chain.
- Removal of a key:
  - Search and remove from list if found.

Example

- Insert of 11, 27, 21, 18, 32, 44, 55 in hash table

HASHING :: CHAINING

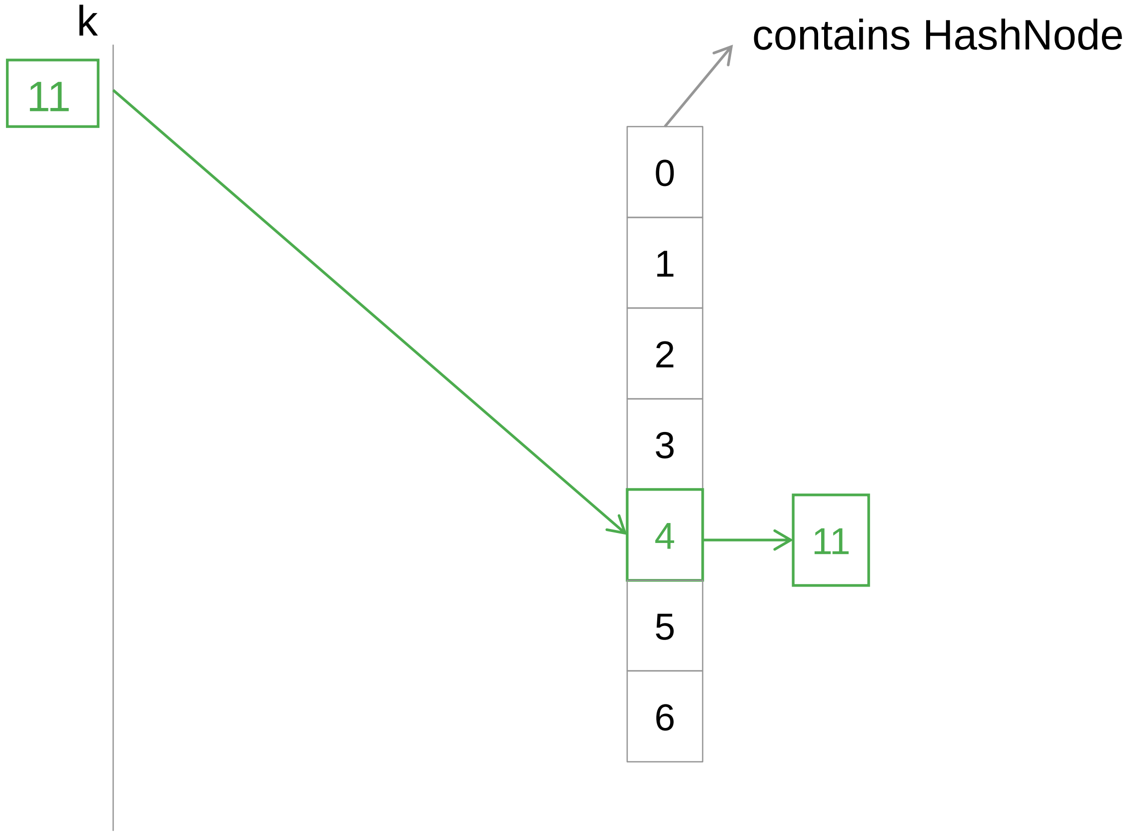
Hash values

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


```
HashNode {
   Integer key
   HashNode next
}
```

HASHING :: CHAINING

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



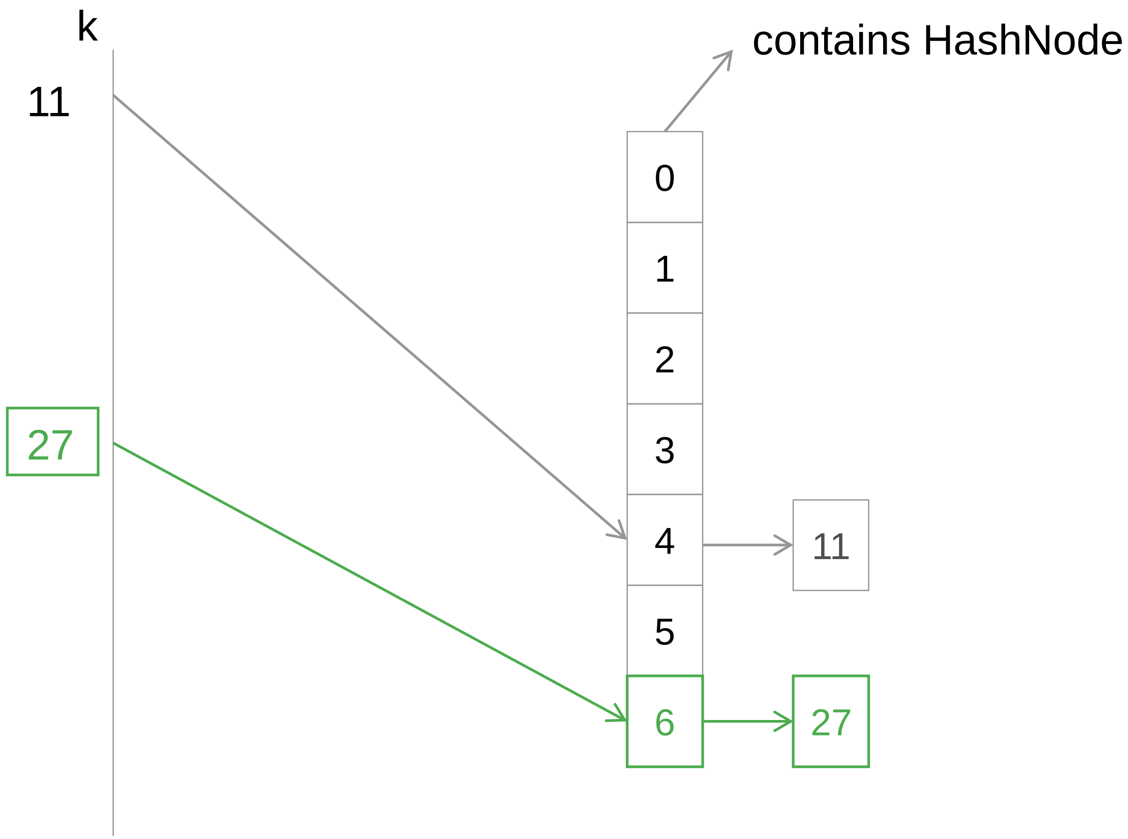
Hash values

h(11) = 11%7 = 4

```
HashNode {
   Integer key
   HashNode next
}
```

HASHING :: CHAINING

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

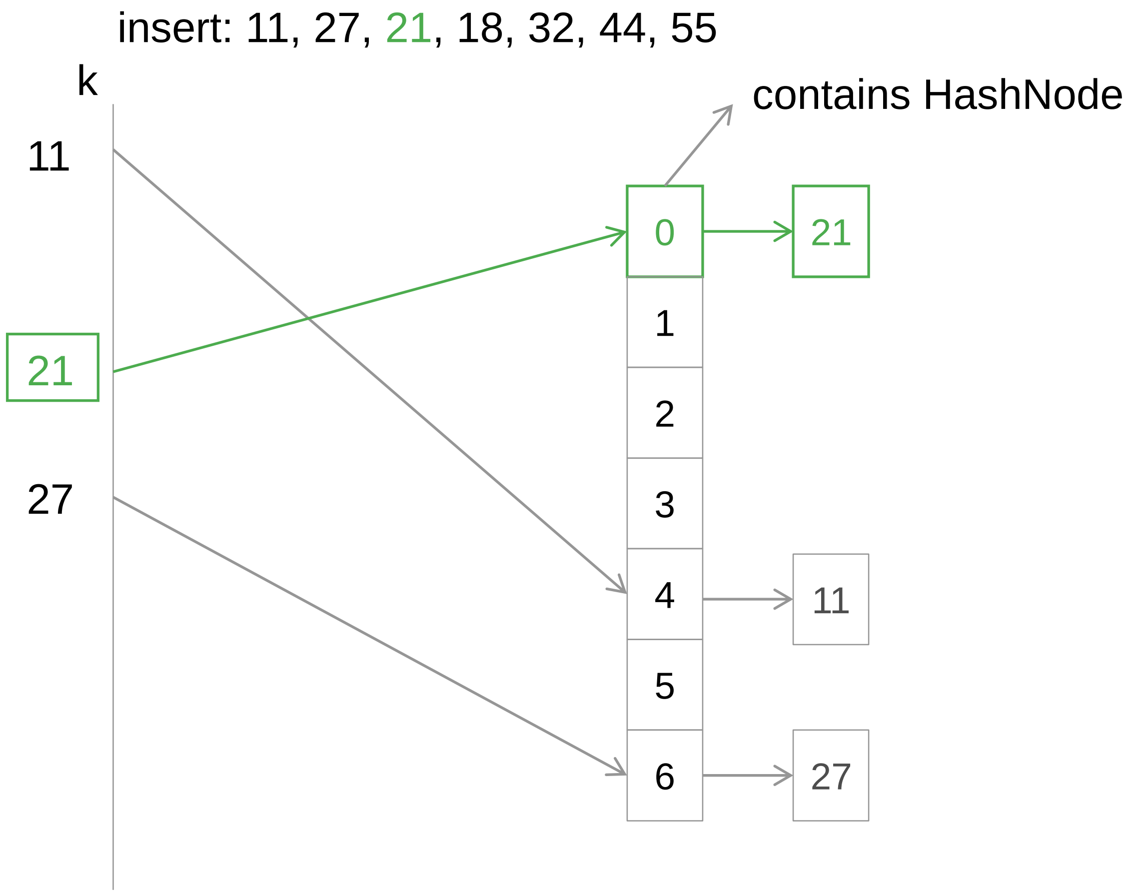



```
HashNode {
   Integer key
   HashNode next
}
```

```
Hash values

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

HASHING :: CHAINING

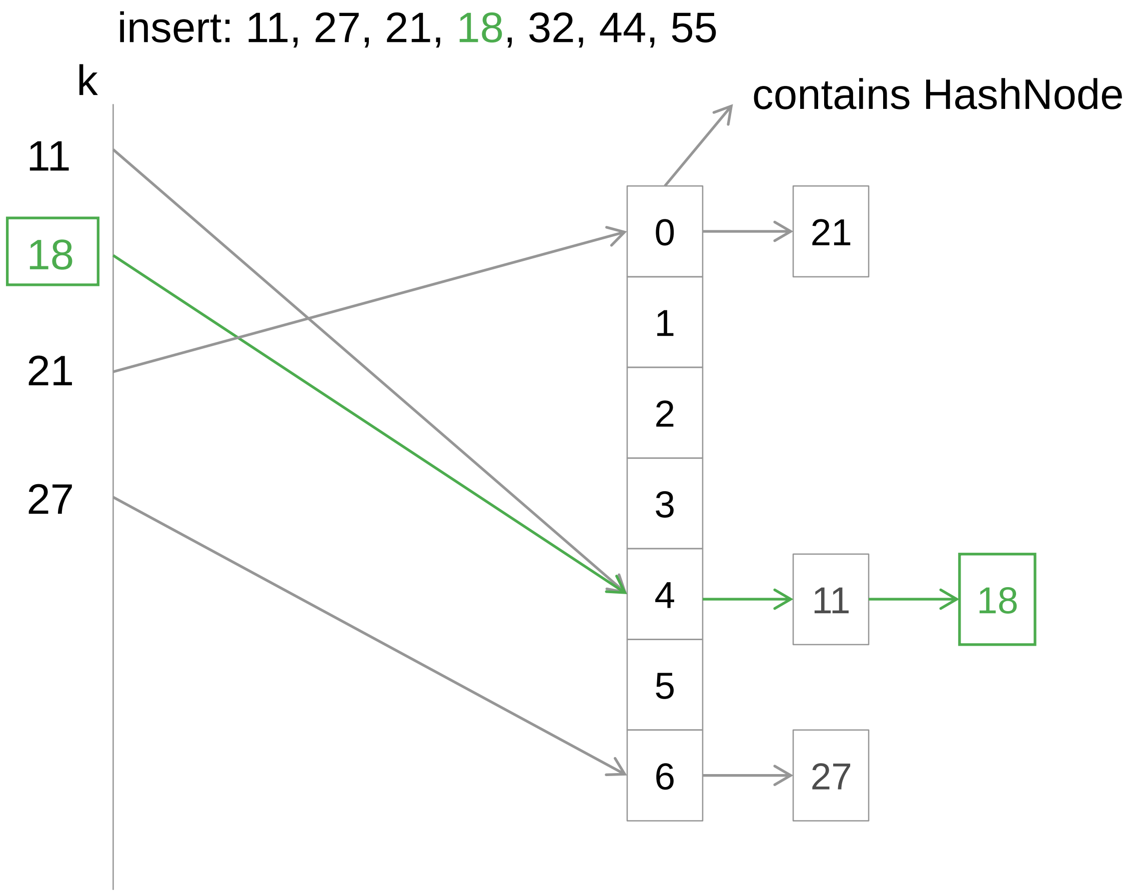



```
Hash values

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

```
HashNode {
   Integer key
   HashNode next
}
```

HASHING :: CHAINING



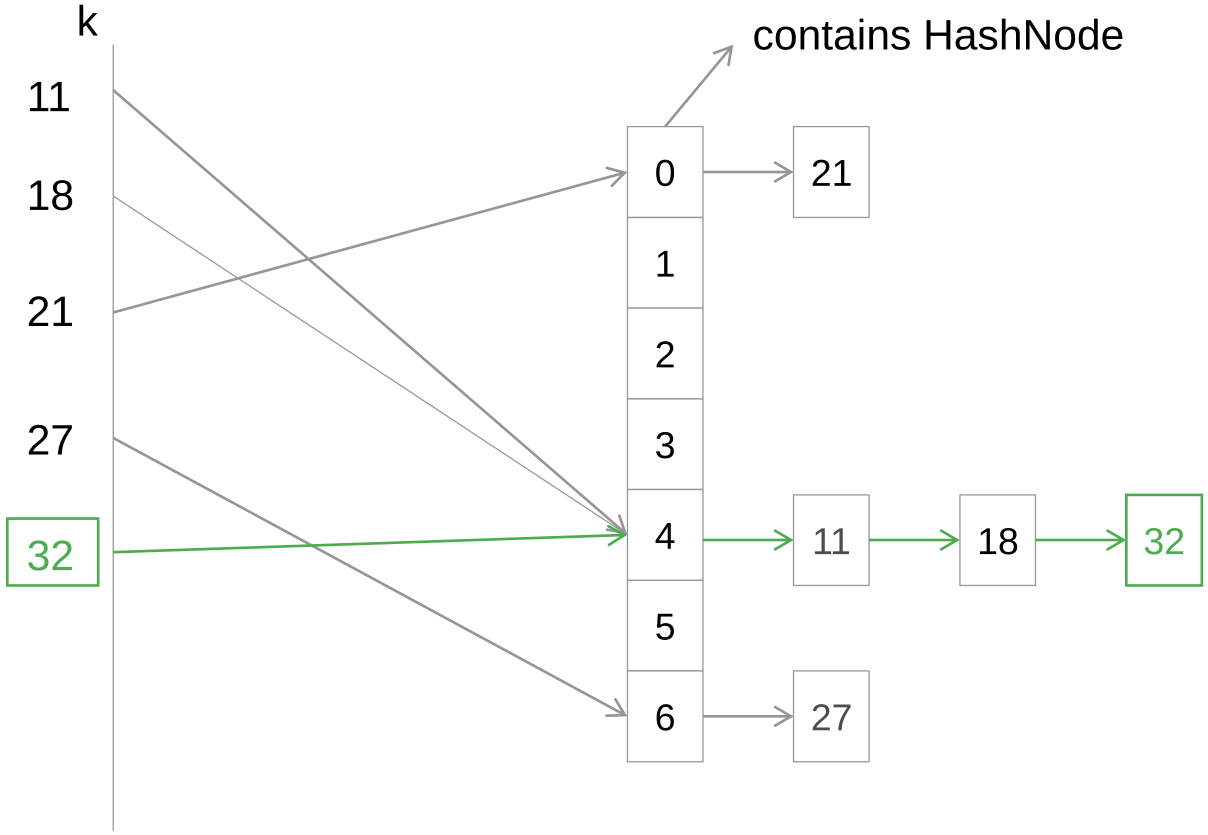

```
HashNode {
   Integer key
   HashNode next
}
```

Hash values

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

HASHING :: CHAINING

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

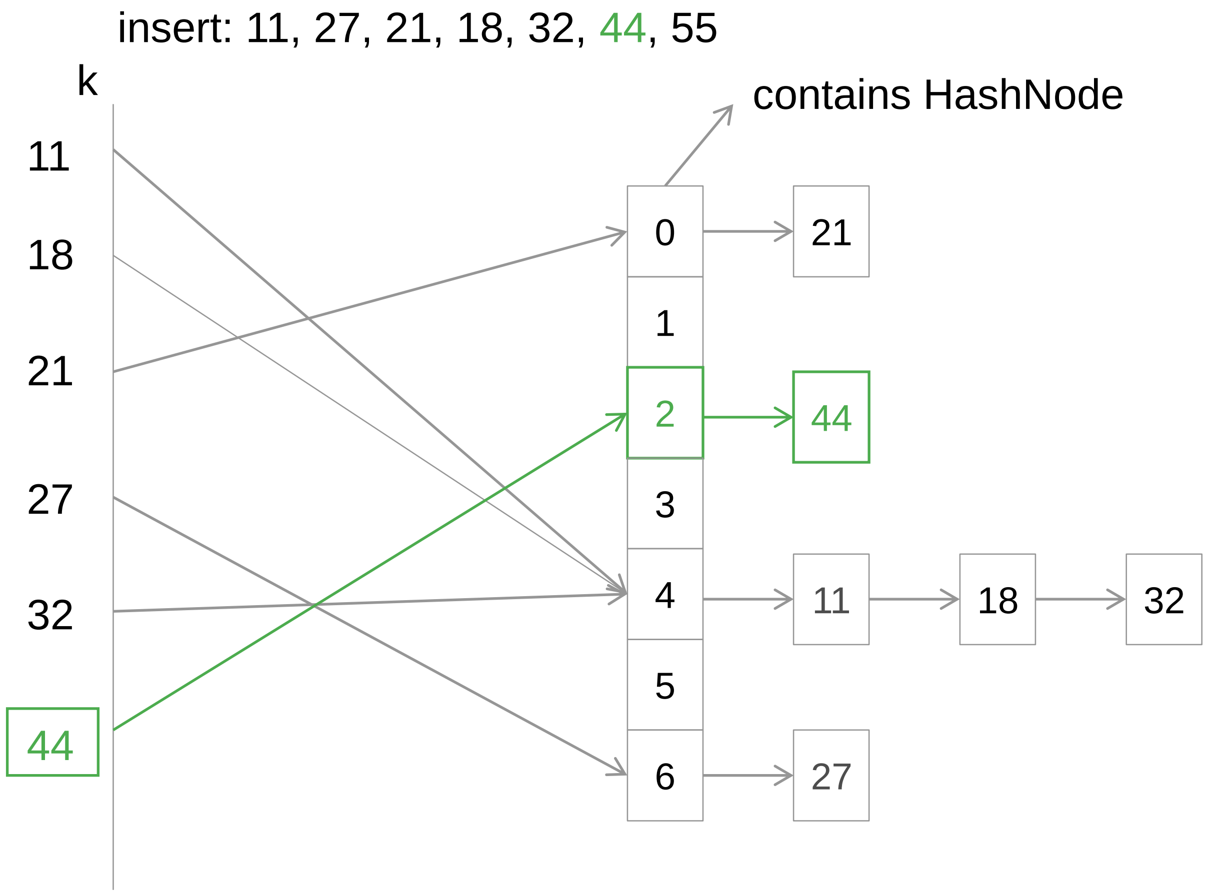



```
HashNode {
    Integer key
    HashNode next
}
```

```
Hash values

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

HASHING :: CHAINING



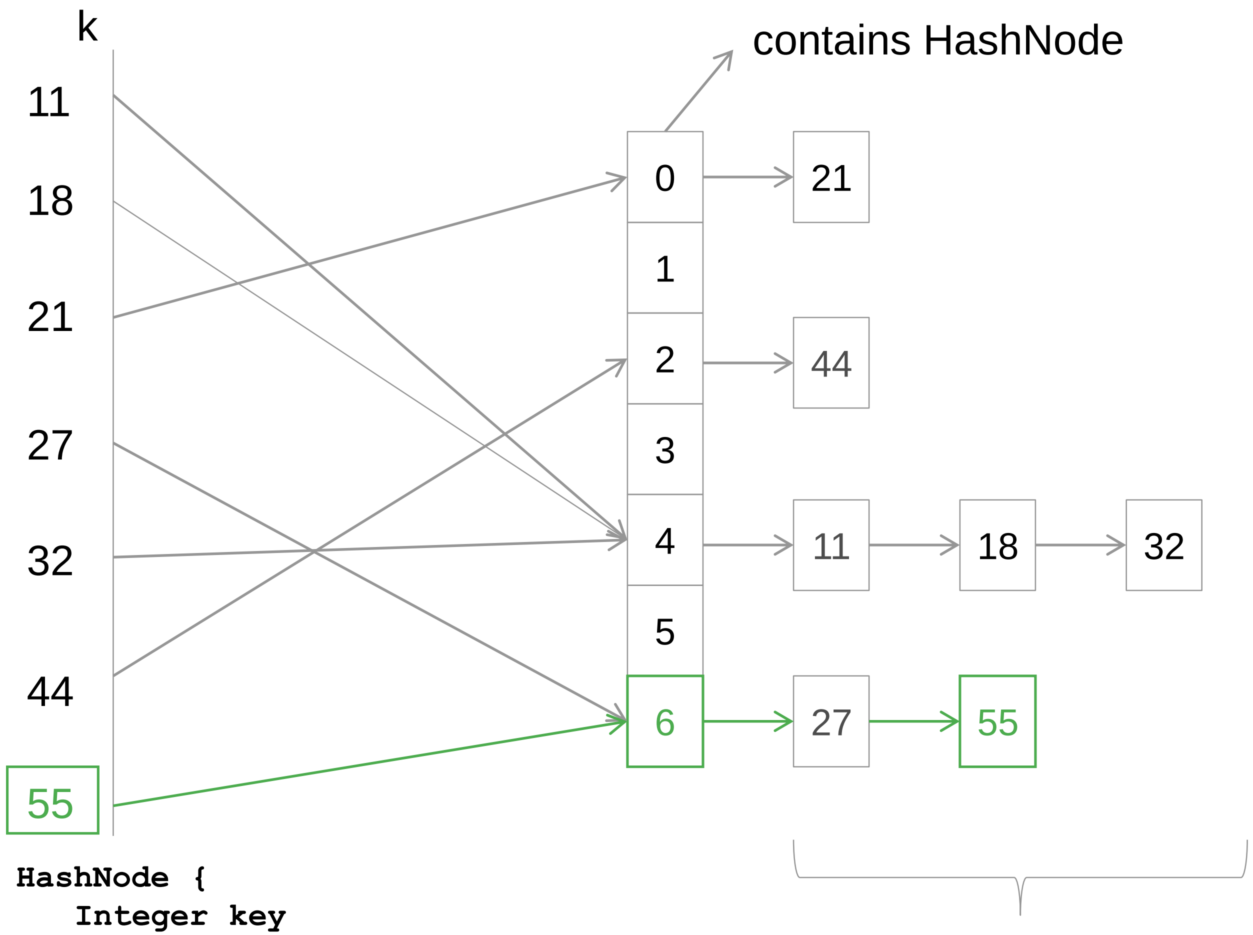

```
HashNode {
   Integer key
   HashNode next
}
```

Hash values

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

HASHING :: CHAINING

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




```
HashNode {
    Integer key
    HashNode next
}
```

Overflow chains

```
Hash values

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

HASHING :: RESOLVING COLLISIONS

2. **Open addressing**

Overflows are stored at vacant positions in the hash table:

- 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:

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 (or left). [→ **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
```

HASHING :: LINEAR PROBING

Example:

0	1	2	3	4	5	6
21				11		27

insert(18): h(18) = 18%7 = 4 → collision '1'
→ (4+1) % 7 = 5 → OK

0	1	2	3	4	5	6
21				11		27

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): h(32) = 32%7 = 4 → collision ‘1’

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): h(32) = 32%7 = 4 → collision ‘1’
→ (4+1) % 7 = 5 → collision ‘2’

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): h(32) = 32%7 = 4 → collision ‘1’
→ (4+1) % 7 = 5 → collision ‘2’
→ (4+2) % 7 = 6 → collision ‘3’

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): h(32) = 32%7 = 4 → collision '1'

→ (4+1) % 7 = 5 → collision '2'

→ (4+2) % 7 = 6 → collision '3'

→ (4+3) % 7 = 0 → collision '4'

0	1	2	3	4	5	6
21				11	18	27

HASHING :: LINEAR PROBING

Example:

insert(32): h(32) = 32%7 = 4 → collision ‘1’
→ (4+1) % 7 = 5 → collision ‘2’
→ (4+2) % 7 = 6 → collision ‘3’
→ (4+3) % 7 = 0 → collision ‘4’
→ (4+4) % 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): h(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): h(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): h(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)*

- h(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

S. Grünberger, M. Schobesberger, D. Trendafilov, M. Weninger

HASHING :: LINEAR PROBING

Example: *contains(32)*

- h(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)*

- h(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 {
    Integer 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

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

S. Grünberger, M. Schobesberger, D. Trendafilov, M. Weninger

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 `(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: $\underbrace{\text{4+1, 4+2, 4+3, 4+4, 4+5, 4+6,}}_{\%\ N}$ ... (= 5, 6, 7, 8, …)

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

Alternative: Instead of this fixed sequence, you can use a second hash function!

HASHING :: DOUBLE HASHING

Principle:

- Reduce clustering by placing different elements with **different step sizes**.
- Definition of the probing sequence by
  - h_1: $k \rightarrow \{0, 1, \ldots, N-1\}$
  - h_2: $k \rightarrow \{1, \ldots, N-1\}$

Pseudo code:

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

Requirements:

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

HASHING :: DOUBLE HASHING

Requirement *(cont'd):*

- Table size should be a prime number or value must be coprime to the table size.

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

HASHING :: DOUBLE HASHING

Requirement *(cont'd):*

- Table size should be a prime number or value must be coprime to the table size.

N=8; offset=4; h=1	*N=7; offset=4; h=1*
(1+4)%8 = 5	(1+4)%7 = 5
(5+4)%8 = 1	(5+4)%7 = 2
(1+4)%8 = 5	(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
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

HASHING :: DOUBLE HASHING

Example:

N=13
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: $h_{new} = h1(h_{old}+offset)$

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

insert(14):
h1(14)=14%13 = 1 [h2(14)=1+(14%12)=3]

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

HASHING :: DOUBLE HASHING

Example:

N=13
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: $h_{new} = h1(h_{old}+offset)$

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

insert(21):
h1(21)=21%13 = 8 [h2(21)=1+(21%12)=10]

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

HASHING :: DOUBLE HASHING

Example:

N=13
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

insert(1):
h1(1)=1%13 = 1 AND h2(1)=1+(1%12)=2
→ (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
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

insert(19):
h1(19)=19%13 = 6 [h2(19)=1+(19%12)=8]

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

HASHING :: DOUBLE HASHING

Example:

N=13
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

insert(10):
h1(10)=10%13 = 10 [h2(10)=1+(10%12)=11]

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

HASHING :: DOUBLE HASHING

Example:

N=13
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

insert(11):
h1(11)=11%13 = 11 [h2(11)=1+(11%12)=12]

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
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: $h_{new} = h1(h_{old}+\text{offset})$

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

insert(6):
h1(6)=6%13 = 6 AND h2(6)=1+6%12=7
→ **(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
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: $h_{new} = h1(h_{old}+offset)$

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

insert(42):
h1(42)=42%13 = 3 AND h2(42)=1+42%12=7
(3+7)%13 = 10
(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
h1(k) = k%N
h2(k) = 1 + k%(N-1) → offset
probing sequence: h_{new} = h1(h_{old}+offset)

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

insert(8):
h1(8)=8%13 = 8 AND h2(8)=1+8%12=9
(8+9)%13 = 4
(4+9)%13 = 0
(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 – 2021W

Stefan Grünberger, Martin Schobesberger
Dari Trendafilov, Markus Weninger
Institute of Pervasive Computing
Johannes Kepler University Linz
teaching@pervasive.jku.at

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