Algorithmen und Datenstrukturen SoSe25

-Assignment 7-

Moritz Ruge
Matrikelnummer: 5600961
Lennard Wittenberg
Matrikelnummer: —

Juni 2025

Problem 1: Hashing im Selbstversuch II

a) Fügen Sie nacheinander die Schlüssel 10, 22, 31, 4, 15, 28, 17, 88, 59 in eine Hashtabelle der Größe 11 ein. Die Hashfunktion sei $h(k) = k \mod 11$. Die Konflikte werden durch offene Adressierung mit linearem Sondieren gelöst.

Lösung:
" " : Empty, * : Deleted

Index	0	1	2	3	4	5	6	7	8	9	10
T =											

1. insert (10,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =											10

2. insert (22,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22										10

3. insert (31,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22									31	10

4. insert (4,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22				4					31	10

5. insert (15,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22				4	15				31	10

$\Rightarrow$ index 4: $4 \neq 15 \Rightarrow$ Index 5: Empty $\rightarrow$ Index 5 = 15

6. insert (28,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22				4	15	28			31	10

7. insert (17,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22				4	15	28	17		31	10

$\Rightarrow$ Index 6: $28 \neq 17 \rightarrow$ Index 7: Empty $\rightarrow$ Index $7 = 17$

8. insert (88,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22	88			4	15	28	17		31	10

$\Rightarrow$ Index 0: $22 \neq 88 \rightarrow$ Index 1: Empty $\rightarrow$ Index $1 = 88$

9. insert (59,v)

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22	88			4	15	28	17	59	31	10

$\Rightarrow$ Index 4: $4 \neq 59 \rightarrow$ Index 8: Empty $\rightarrow$ Index $8 = 59$

b) Fügen Sie nacheinander die Schlüssel 10, 22, 31, 4, 15, 29, 17, 88, 59 in eine Hashtabelle der Größe 11 ein. Die Konflikte werden durch Kuckuck gelöst, mit $h1(k) = k \mod 11$ und $h2(k) = (k \mod 13) \mod 11$.

Illustrieren Sie jeweils die einzelnen Schritte.
Hinweis: Pseudocode für die Hashoperation findet sich im Skript.

Lösung:
" " : Empty, * : Deleted

Index	0	1	2	3	4	5	6	7	8	9	10
T =											

1. insert (10)

   $\Rightarrow h_1(k) = k \mod 11 = 10 \mod 11 = 10$

Index	0	1	2	3	4	5	6	7	8	9	10
T =											10

2. insert (22)

   $\Rightarrow h_1(k) = k \mod 11 = 22 \mod 11 = 0$

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22										10

3. insert (31)

   $\Rightarrow h_1(k) = k \mod 11 = 31 \mod 11 = 9$

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22									31	10

4. insert (4)

   $\Rightarrow h_1(k) = k \mod 11 = 4 \mod 11 = 4$

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22				4					31	10

5. insert (15)

   $\Rightarrow h_1(15) = 15 \mod 11 = 4$ Platziere in Index 4, $k' = 4$.
   $\rightarrow h_1(4') = 4 \mod 11 = 4$. Anwendung $h_2(k')$
   $\rightarrow h_2(4') = (4 \mod 13) \mod 11 = 4$ Platz in Index 4, $k' = 15$.
   $\Rightarrow h_1(15') = 15 \mod 11 = 4$. Anwendung $h_2(k')$
   $\rightarrow h_2(15') = (15 \mod 13) \mod 11 = 2$ Platz in Index 2.

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22		15		4					31	10

6. insert (29)

   $\Rightarrow h_1(k) = 29 \mod 11 = 7$ Platziere in Index 7.

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22		15		4			29		31	10

7. insert (17)

   $\Rightarrow h_1(k) = 17 \mod 11 = 6$ Platziere in Index 6.

Index	0	1	2	3	4	5	6	7	8	9	10
T =	22		15		4		17	29		31	10

8. insert (88)

   $\Rightarrow h_1(88) = 88 \mod 11 = 0$ Platziere in Index 0, $k' = 22$.
   $\rightarrow h_1(22') = 22 \mod 11 = 0$. Anwendung $h_2(k')$
   $\rightarrow h_2(22') = (22 \mod 13) \mod 11 = 9$ Platz in Index 9, $k' = 31$.
   $\Rightarrow h_1(31') = 31 \mod 11 = 9$ Anwendung $h_2(k')$
   $\rightarrow h_2(31') = (31 \mod 13) \mod 11 = 5$ Platz in Index 5 $\rightarrow$ Empty.

Index	0	1	2	3	4	5	6	7	8	9	10
T =	88		15		4	31	17	29		22	10

9. insert (59)

   $\Rightarrow h_1(59) = 59 \mod 11 = 4$ Platziere in Index 4, $k' = 4$.
   $\rightarrow h_1(4') = 4 \mod 11 = 4$. Anwendung $h_2(k')$
   $\rightarrow h_2(4') = (4 \mod 13) \mod 11 = 4$ Platz in Index 4, $k' = 59$.
   $\Rightarrow h_1(59') = 59 \mod 11 = 4$ Anwendung $h_2(k')$.
   $\rightarrow h_2(59') = (59 \mod 13) \mod 11 = 7$ Platz in Index 7, $k' = 29$.
   $\Rightarrow h_1(29') = 29 \mod 11 = 7$ Anwendung $h_2(k')$.
   $\rightarrow h_2(29') = (29 \mod 13) \mod 11 = 3$ Platz in Index 3 $\rightarrow$ Empty.

Index	0	1	2	3	4	5	6	7	8	9	10
T =	88		15	29	4	31	17	59		22	10

Problem 2: Implementierung einer Hashtabelle

a) Implementieren Sie eine Hashtabelle mit Verkettung in Scala. Benutzen Sie dazu die Funktion hashCode, die von allen Objekten in Scala zur Verfügung gestellt wird.
Gestalten Sie Ihre Implementierung so, dass sich die Größe der Hashtabelle wählen lässt, und implementieren Sie mit mindestens zwei verschiedenen Kompressionsfunktionen.

b) Erweitern Sie Ihre Implementierung so, dass die Größe der Hashtabelle dynamisch angepasst wird, um einen Ladefaktor zwischen 1 und 3 zu garantieren (sobald mindestens 20 Einträge in der Hashtabelle vorhanden sind). Welche Strategie wählen Sie, um Ihre Hashtabelle anzupassen?

Datei: Hashtable.scala

```scala
class HashTable(size : Int, compressionFunktion: Int => Int):
    // public changable Attribute to alter, hold and display the (Key,Values) pairs.
    // Using Array two a fixed size for the Hashtable and dynamic List Chains.
    private var Table = Array.fill[List[(Int,Int)]](size)(Nil)
    private var numEntries = 0 // counter pairs in Table
    private var loadFactorCheck = 20 // indicator when the Table needs to be resized

    private def resize(newSize : Int) : Unit =
        val oldTable = this.Table
        this.size = newSize
        this.Table = Array.fill[List[(Int,Int)]](this.size)(Nil)
        this.numEntries = 0
        this.loadFactorCheck = loadFactorCheck * 2
        // copy all old pairs into the resized Table.
        for chain <- oldTable do
            for (k,v) <- chain do
                put(k,v)

    // Two-Part hash-function 1. builtin hashCode and 2. class instances given
    compression function to create a index for the Table
    def hash(key : Int) : Int =
        val hashedPosition = key.hashCode()
        val index = compression_division(hashedPosition,size)

    // main 3 Operations
    // get(k,v)
    def get(key : Int) : Int =
        val index : Int = hash(key) // 1. get the starting index
        //2.Search the linked list at T[i] for an entry with key k
        //3.If found, return the associated value
        //4.If not found, throw NoSuchElementException in this case std error value
        var chain : List[(Int,Int)] = this.Table(index)
        var res : Int = -1 // std return value -> handling in main
        for (k,v) <- chain do
            if k == key then
                res = v
        res

    //remove(k)
    //1.Calculate index i = h(k)
    //2.Search the linked list at T[i] for an entry with key k
    //3.If found, remove that node from the linked list
    //4.If not found -> nothing happens in this implementation
    def remove(key : Int) : Unit =
        val index : Int = hash(key)
        val initalLength : Int = this.Table(index).length
        Table(index) = Table(index).filter(_._1 != key)
```

```
        if Table(index).length != initalLength then
            this.numEntries -= 1 // keeping track of the number of entries

    // put(k,v)
    //  1.Calculate index i = h(k)
    //  2.Search the linked list at T[i] for an entry with key k
    //  3.If found, update its value to v
    //  4.If not found, append a new node with (k,v) to the list at T[i]
    def put(key : Int, value : Int): Unit =
        val index : Int = hash(key)
        var chain : List[(Int,Int)] = this.Table(index)
        chain = chain.map(x = > if(x._1 == key) then (key,value) else x)    // map over chain if entry.key == wanted key -> update value
        if chain.filter(_._1 == key).length == 0 then                       // After maping no entry with key present -> append
            chain = chain ::: List((key, value))
            this.numEntries += 1                                            // keeping track of the number of entries
        this.Table(index) = chain                                           // update the chain in Table with the new altered Chain
        if numEntries >= loadFactorCheck then   // if more Entires than the current loadFactorCheck allows are in the Table -> resize accordingly
            val loadFactor = this.numEntries.toDouble / this.size
            if loadFactor < 1 then      // loadFactor too small
                resize(this.size/2)
            if loadFactor > 3 then      // loadFactor too high
                resize(this.size*2)

//compression functions modeled after the script
def compression_division(hash : Int, size : Int): Int =
    hash % size
def compression_multiplication(hash : Int, size : Int): Int =
    val A : Float = 0.6180339887 // 0.6180339887 is the standard value from the script.
    val product = hash * A
    val fraction = product - Math.floor(product)
    (size * fraction).toInt

// main-function for debugging and testing
@main def main(): Unit =
    var HT : HashTable = HashTable(10,compression_division)
```

Problem 3: Lineares Sondieren und Löschen

a) In der Vorlesung haben Sie eine Strategie gesehen, mit der das Löschen in einer Hashtabelle mit linearem Sondieren umgesetzt werden kann: Gelöschte Elemente werden durch einen eigenen Eintrag markiert und in den Einfüge- und Lookup Routinen speziell behandelt.
Beschreiben Sie eine alternative Methode, bei welcher der gelöschte Eintrag ggf. durch einen geeigneten anderen Eintrag ersetzt wird. Beschreiben sie Ihre Methode verbal und geben Sie Pseudocode. Geben Sie auch zwei interessante Beispiele, die zeigen, wie Ihre Methode funktioniert.

Lösung: Robin-Hood-Hashing mit Backwards-Shifting

Prinzip: Es handelt sich hierbei um eine Hashtabelle mit *Open Addressing*.
Beim Einfügen von neuen Schlüsseln wird für jeden Schlüssel der PSL (Probe Sequenz Lenght) berechnet um anzugeben, wie weit der Schlüssel von seiner eigentlichen Hashposition entfernt ist. Ziel ist es nun, das alle Schlüssel so nah wie möglich an ihrer "Home" Position bleiben. Beim Löschen von Schlüsseln wird geschaut ob es nachfolgend noch andere Schlüssel gibt, wenn nicht sind wir fertig, wenn doch wir auf deren PSL-Wert geschaut, ist dieser Größer als 0 Shiften wir den Schlüssel nach "Links" (auf die Position des gelöschten Schlüssels) und gegen in der Tabelle weiter, bis wir ein PSL-Wert von 0 oder einen Leeren Slot haben.

Einfügen:

⇒ insert: $h(8) = 1$

Index	0	1	2	3	4	5
T =		8_0				

⇒ insert: $h(20) = 1$

Index	0	1	2	3	4	5
T =		8_0				
		$20_0 \rightarrow$				

Index	0	1	2	3	4	5
T =		8_0	20_1			

⇒ Wir verschieben 20 nach rechts und erhöhen den PSL um 1

⇒ insert: $h(3) = 1$

Index	0	1	2	3	4	5
T =		8_0	20_1			
		$3_0 \rightarrow$	$3_1 \rightarrow$			

Index	0	1	2	3	4	5
T =		8_0	20_1	3_2		

⇒ Wir verschieben 3 2x nach rechts und erhöhen den PSL um 2

⇒ insert: $h(9) = 0$

Index	0	1	2	3	4	5
T =	9_0	8_0	20_1			

$\Rightarrow$ insert: $h(66) = 0$

Index	0	1	2	3	4	5
T =	9_0	8_0	20_1			
	$66_0 \rightarrow$					

Index	0	1	2	3	4	5
T =	9_0	66_1	20_1	8_2		
		$8_0 \rightarrow$	$8_1 \rightarrow$			

$\Rightarrow$ Wir verschieben 66 nach rechts und erhöhen den PSL um $1 \rightarrow 66_1$
$\rightarrow$ Da 66_1 höher ist als 8_0 tauschen wir die Werte und verschieben die 8 nach rechts
$\rightarrow$ Da $8_1 = 20_1$ ist schieben wir den Schlüssel weiter nach rechts und erhöhen den PSL