Dokumentation

ADT MultiMap

–

Implementierung

auf einer Hashtabelle,

Kollisionsbehandlung mit unabhängige

Verkettung

# Inhalt:

1. Beschreibung der Aufgabe, die mit dem ADT gelöst wird.
2. ADT – Spezifikation und Interface
3. Repräsentierung des Containers und des Iterators
   1. MultiMap
   2. Iterator
4. Implementierung in Pseudocode
   1. MultiMap
   2. Iterator
5. Teste
6. Warum der ADT geeignet für die ausgewählte Aufgabe ist
7. Schluss

# Beschreibung der Aufgabe

Ein Transportunternehmen möchte ein Bus- / Straßenbahn-Überwachungssystem in der Stadt implementieren. Sie wollen wissen wie viele Busse / Straßenbahnen eine Linie hat. Der Transportunternehmen kann, zu eine Linie , zusätzliche Busse / Straßenbahnen hinzufügen oder entfernen.

# Spezifikation und Interface des ADT

Domäne: MM = {mm| mm ist ein MultiMap mit Elementen e = (k, v), wobei k ∈ TKey und v ∈ TValue[]}

* **Init**(mm)
  + **Descr**: erstellt eine leere MultiMap
  + **Pre**: Wahr
  + **Post**: mm ∈ MM, m ist eine leere MultiMap
  + **Komplexität** ϴ(1)
* **Destroy**(mm)
  + **Desc:** zerstört eine MultiMap
  + **Pre:** mm ∈ MM
  + **Post:** m wurde zerstört
  + **Komplexität** ϴ(1)
* **Add**(mm,k,v)
  + **Descr:** fügt ein neues key-value Paar zu dem MultiMap ein
  + **Pre:** mm ∈ MM, k ∈ TKey, v ∈ TValue
  + **Post:** mm’ ∈ MM, mm’ = mm ∪ <k,v>
  + **Komplexität** O(n)
* **Remove**(mm,k,v)
  + **Descr:** löscht ein Element mit einem gegebenen Schlüssel aus dem MultiMap
  + **Pre:** mm ∈ MM, k ∈ TKey
  + **Post:** v ∈ TValue, wobei:

**Komplexität** O(mm)

* **Search**(mm,k,v)
  + **Descr:** sucht den Wert, der dem angegebenen Schlüssel entspricht
  + **Pre:** mm ∈ MM, k ∈ TKey
  + **Post:** v ∈ TValue wobei:

**Komplexität** O(mm)

* **Iterator**(mm, it)
  + **Descr:** gibt ein Iterator für eine MultiMap zurück
  + **Pre:** mm ∈ MM
  + **Post:** it ∈ I, it ist ein Iterator für m
  + **Komplexität** ϴ(1)
* **Size**(mm)
  + **Descr:** gibt die Anzahl der Paare in der MultiMap zurück
  + **Pre:** mm ∈ MM
  + **Post:** size ← Anzahl der Paare in m
  + **Komplexität** ϴ(1)
* **Keys**(mm, s)
  + **Descr:** gibt die Menge der Schlüssel aus der MultiMap zurück
  + **Pre:** mm ∈ MM
  + **Post:** s ∈ S, s ist ein Set der alle Schlüssel aus mm enthält
  + **Komplexität** O(mm)
* **Values**(mm, b)
  + **Descr:** gibt ein Bag von Werten aus der MultiMap zurück
  + **Pre:** mm ∈ MM
  + **Post:** b ∈ B, b ist ein Bag der alle Werte aus mm enthält
  + **Komplexitat** O(mm)

# Repräsentierung des Containers und des Iterators:

## MultiMap

1. **Beschreibung**
   1. **MultiMap** enthält Paare der Form Schlüssel-Werte (key-values[])
   2. Ein Schlüssel kann mehrere zugehörigen Werte enthalten (eine List von Werten)
2. **Repräsentierung**

**Node:**

Key: TKey

Values: TValue[]

Next: ↑Node

**HashTable**

Table: ↑Node[]

Size: Integer

HashF: TFunction

## Iterator

1. **Beschreibung**
   1. **Der Iterator** gibt uns das Element in der MultiMap zurück
   2. Der Iterator ist eine Mischung zwischen ein Iterator für einen Array und einen für eine Liste
2. **Repräsentierung**

**Iterator:**

mm: MultiMap

currPos: Integer

currElem: Integer

currNode: ↑Node

# Implementierung in PSEUDOCODE

## MultiMap

**Subalgorithm** init() is:

Table = @allocate an Array of map.size Nodes

For (i =0; i < map.size ; i++)

Table[i] = NIL;

Endfor

**Endsubalgorithm** Komplexitat ϴ(n)

**Subalgorithm** destroy() is:

For (i=0; i < table.size ; i++)

If (table[i] != NIL) then

prevNode = NIL

currNode = table[i]

while (currNode != NIL) do

prevNode = currNode

currNode = currNode.next

@deallocate prevNode

Endwhile

Endif

Endfor

@Delete table

**Endsubalgorithm** Komplexitat ϴ(n)

**Function** hash(key) is:

Hash = key % map.size

**Endfunction** Komplexitat ϴ(1)

**Function** search(key,value)

H = hash(key)

If (table[h] == NIL) then

Search = false

Else

currNode = table[h]

while ((currNode.key != key) AND (currNode.next != NIL) do

currNode = currNode.next

endwhile

if (currNode.key == key) then

size = currNode.values.size

for (i=0; i<size; i++)

if (currNode.values[i] == value) then

search = True

endif

endfor

endif

search = false

endif

**endfunction**

**Komplexität:** O(n)

Kalkül:

**Function** getValues(key) is:

H = hash(key)

If (table[h] == NIL) then

@Key doesn’t exist

Else

currNode = table[h]

while ((currNode.key != key) AND (currNode.next != NIL) do

currNode = currNode.next

endwhile

if (currNode == NIL) then

@Key doesn’t exist

Else

getValues = currNode.values

endif

endif

**endfunction** Komplexitat O(n)

**Function** getKeys() is:

i = 0

getKeys = @create empty array with TKey

while (i < mm.size) do

currNode = mm.table[i]

if (currNode == NULL) then

i = i+1

else

while(currNode.next != NIL) do

getKeys.add(currNode.key)

endwhile

if (currNode.next == NULL) then

getKeys.add(currNode.key)

endif

i = i+1;

endif

endwhile

**endfunction** Komplexitat O(n)

**subalgorithm** add(key,value) is:

check = search(key,value)

if (check == False) then

h = hash(key)

if (table[h] == NIL) then

#*Der neue Knoten wird das Head der neue Liste sein*

values = @create array of values

values.add(value)

table[h] = @allocate new Node with key and newly created array of values

else

#*Wir müssen ein Platz finden wobei die Value eingefügt sein muss.*

*#Wir unterscheiden zwei fälle: Es gibt schon ein Knoten die die Key enthält*

*#und wir müssen nur die Value einfügen,*

*#oder müssen wir ein neues Knoten an ende der Liste einfügen.*

added = False

currNode = table[h]

while ((currNode.next != NULL) || (added == false)) do

if (currNode.key == key) then

added = true

currNode.values.add(elem)

else

currNode = currNode.next

endif

endwhile

if ((currNode.next == NIL) && (added == false)) then

*#Wir sind an der Ende der Liste angelangen, wir fügen einen neuen #Knoten hinzu*

Values = @create array of values

Values.add(value)

newNode = @allocate newNode with key and values

newNode.next = NIL

currNode.next = newNode

endif

endif

else

@Value already exists

Endif

**Endsubalgorithm** Komplexitat O(n)

**Subalgorithm** remove(key,value) is:

H = hash(key)

If (table[h] != NIL) then

prevNode = NIL

currNode = table[h]

while ((currNode.next != NIL) AND (currNode.key != key)) do

prevNode = NIL

currNode = currNode.next

endwhile

if (currNode.key == key) then

currNode.values.delete(value)

if (currNode.values.size == 0) then

*# wir prüfen ob jetzt das Array von Values leer ist*

if (prevNode == NIL) then

*# wir müssen der Head der Liste löschen*

Table[h] = currNode.next

@free currNode

Else if (currNode.next == NIL) then

*# wir müssen der letzten Knoten der Liste löschen*

prevNode.next = NIL

@free currNode

Else

*# Der Knoten befindet sich etwa in die Mitte*

nextNode = currNode.next

prevNode.next = nextNode

@free currNode

Endif

Endif

Else

@Key doesn’t exist

Endif

else

@Key doesn’t exist

Endif

**Endsubalgorithm** Komplexitat O(n)

## Iterator

**Subalgorithm** init(map) is:

Mm = map

currPos = 0

currElem = 0

while ((currPos < map.size) && (map.table[currPos] == NULL) do

currPos = currPos + 1

endwhile

if (currPos < map.size) then

currNode = map.table[currPos]

else

currNode = NULL

endif

**Endsubalgorithm** Komplexitat O(n)

**Function** getCurrent() is:

getCurrent = currNode.values[currElem]

**endfunction** Komplexitat ϴ(1)

**Subalgorithm** next() is:

If (currElem < currNode.values.size) then

currElem = currElem +1

else

currElem = 0

if (currNode.next != NULL) then

currNode = currNode.next

else if (currNode.next == NULL) then

currPos = currPos + 1

while ((currPos < hm.size) && (hm.table[currPos] == NULL)) do

currPos = currPos + 1

endwhile

if (currPos != NULL) then

currNode = hm.table[currPos]

else

currNode = NULL

endif

endif

endif

**Endsubalgorithm** Komplexitat O(n)

**Function** valid() is:

If (currNode == NIL)

Valid = False

Else

Valid = True

Endif

**Endfunction** Komplexitat ϴ(1)

## Teste

**Subalgorithm** testFunctionality() is:

testMap = map.init()

@assert testMap.size == TABLE\_SIZE

@assert testMap.getValues is Empty

@assert testMap.getKeys is Empty

testBus = object Bus („EX-00-MPL”)

@assert testMap.hash(5) = 0

testMap.add(5, testBus)

@assert testMap.getValues[0] == „EX-00-MPL”

@assert testMap.getKeys[0] == 5

@assert testMap.table[0].next == NIL

testBus2 = object Bus („EX-11-MPL”)

testMap.add(10, testBus2)

@assert testMap.table[0].next != NIL

Check = testMap.search(10, „EX-11-MPL”)

@assert check == True

testMap.remove(10, „EX-11-MPL”)

check = testMap.search(10, „EX-11-MPL”)

@assert check == False

@assert testMap.table[0].next == NIL

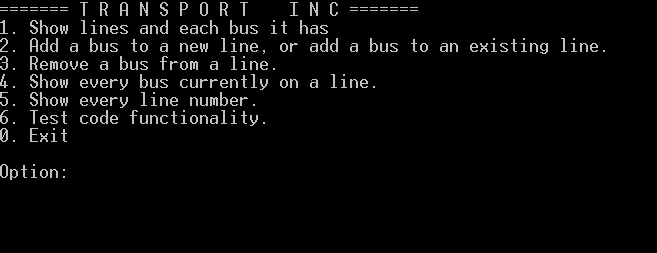
@destroy testMap

**endsubalgorithm**

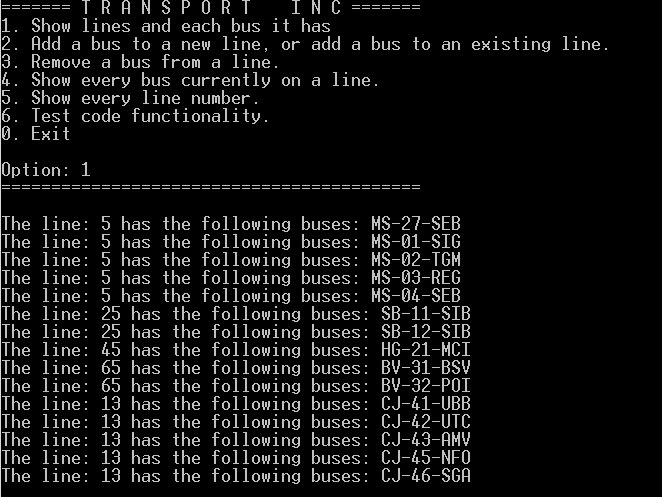
# Warum der ADT geeignet für die ausgewählte Aufgabe ist

Der ADT ist geeignet für die ausgewählte Aufgabe, da das <Key, Values[]> Konzept zu unsere Problem umgewandelt sein kann. Wir können Buslinien als „Key” betrachten und weil eine Buslinie mehrere Busse haben kann, können wir diese Busse, oder das ID dieser Busse, in die „Values” speichern. Außerdem, benützen wir ein viel kleineres Speicherplatz. Wir können wenigstens 10 Speicherplätze für die Hashtabelle haben, indem wir auf jeden Slot, mehrere Paare speichern können. Weil wir die Buslinien als Schlüsseln speichern, können wir effizient alle Buslinien auf den Hashtabelle speichern mit die Hilfe des Hashfunktions. Bedauerlicherweise, falls 2 Paare den selben Hashwert haben, brauchen wir zusätzliches Speicherplatz. Das heißt, wir müssen ein neues Knoten bilden und ihm zu den existierenden Knoten auf dem Slot binden. Das wird eine verkettete Liste bilden. Die Komplexität der Algorithmen ADD, REMOVE und SEARCH hängen von der Länge jeder verkettete Liste ab.

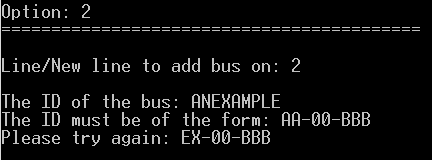
Die Problem wird so ansehen, implementiert in C++ und mit Visual Studio 2017 kompiliert.



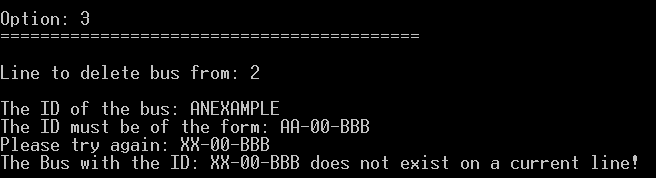
1. Für die erste Option, drucken wir die Buslinien und die dazugehörigen Busse mit hilfe des Iterators. Das wird so ansehen:



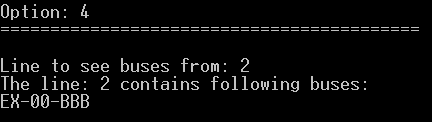
1. Für die zweite Option, benutzen wir die ADD Funktion des Containers. Die Daten müssen gültig sein, dass heißt, sie müssen der Form „AA-00-BBB” sein. Falls sie falsch eingetragen waren, wird das Program eine Nachricht drucken. Der Program wird erstens mit hilfe der Funktion SEARCH durchsuchen, ob das Element der eingefügt werden muss, schon in der MultiMap ist. Falls das der Fall ist, wird das Program eine Nachricht drucken; falls nicht, wird das Element eingefügt.



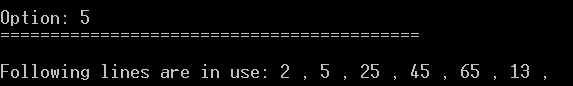
1. Als für die dritte Option, benutzen wir de DELETE Funktion des Containers. Wie bei der zweite Option, müssen hier die Daten gültig sein.



1. Die vierte Option druckt alle Busse die auf eine Buslinie sind. Der Program lauft hier die Funktion „GETVALUES” durch.



1. Die fünfte Option druckt mit hilfe des Funktions „GETKEYS” alle Schlüssel aus dem MultiMap.



1. Die sechste Option durchlauft alle Teste für das Container.
2. Die letzte Option bringt das Program zu ende.

# Schluss

Somit können wir eine MultiMap auf einen Hashtabelle mit unabhängige Verkettung als Kollisionsbehandlung implementieren. Auf einer Seite benutzen wir keine zusätzliche Algorithmen oder Funktionen um unsere Problem zu lösen, aber auf der andere Seite verlieren wir in der effizienz des Containers falls zu viele Daten eingefügt werden, da die verkettung der Listen zu lang wird.