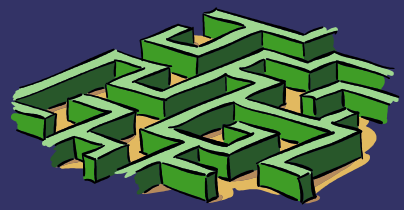
Aufgabe 3

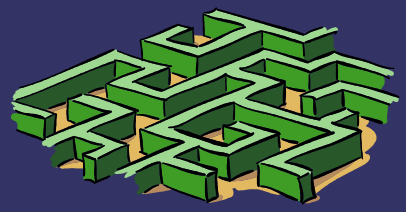
Stellen Sie die Datenstruktur der redistributiven Heaps zur Realisierung von Prioritätswarteschlangen vor. Beschreiben Sie die generelle Funktionsweise der Datenstruktur und welche Laufzeiten für die Priority Queue Operationen erreicht werden. Gehen Sie dabei auch darauf ein, wie diese Laufzeitgarantien gezeigt werden können (Beweisidee).

Grundlage ist folgender Artikel: R. K. Ahuja, K. Mehlhorn, J. B. Orlin, and R. E. Tarjan, Faster algorithms for the shortest path problem, Journal of the ACM, 37(2), pp. 213-223, 1990.



Radix Heaps

- Ein Radix Heap ist eine Datenstruktur zur Realisation der Operationen einer Prioritätswarteschlangen.
- Hiermit kann dann eine Menge von Elementen, denen jeweils ein Schlüssel zugeordnet ist, verwaltet werden.
- Die Laufzeit der Operationen ist abhängig von der Differenz des größten und kleinsten Schlüssels bzw. konstant.
- Die Datenstruktur besteht hauptsächlich aus einer Reihe von Buckets, deren Größe exponentiell zunimmt.



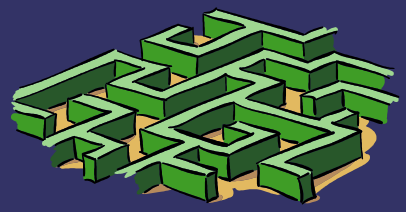
Radix Heaps

- Voraussetzungen:

1. Für jeder Knoten v, falls d(v)=endlich, d(v) ist zw. [0..nC]

2. Für jeder Knoten v!=s, falls v ist labeled, die Kosten d(v) sind zwischen [d(x)...d(x)+C], wo x=neulichst untersuchter Knoten (*)

3. Monotonie von ExtractMin, d.h. die von aufeinander folgenden ExtractMin-Aufrufen zurückgegeben Werte sind monoton steigend



One-Level Radix Heap:

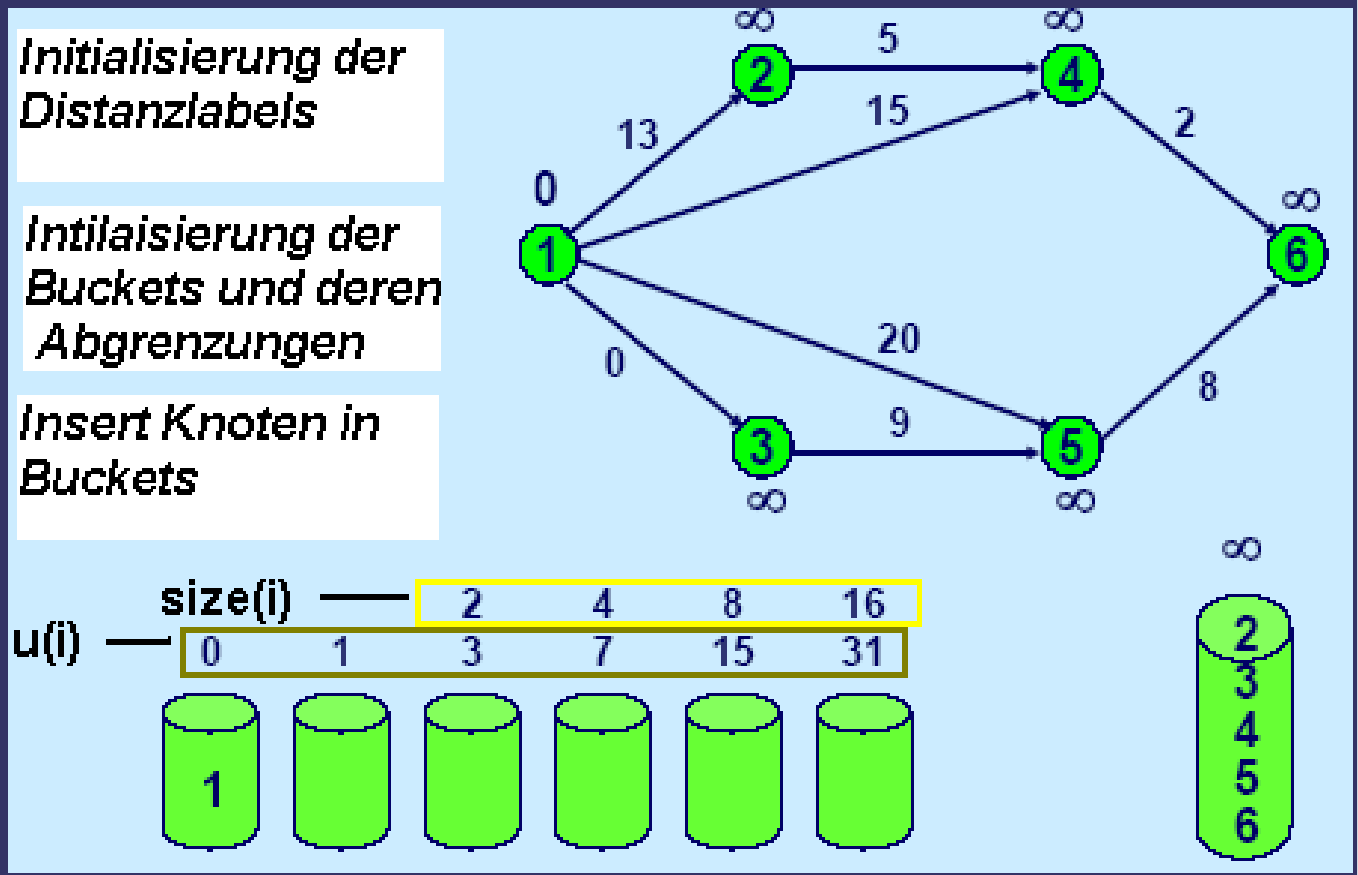
= Sammlung of B=log(C+1)+2 buckets, indexiert von 1 bis B

Jedes Bucket "i "hat eine Größe size(i), die wiefolg definiert ist:

size(1)=1;
size(i)= 2^(i-2)
für 2<=i<=B-1;
...
size(B)=nC+1;

und

u(i)=2^(i-1)-1
für 1<=i<=B-1



One-Level Radix Heap:

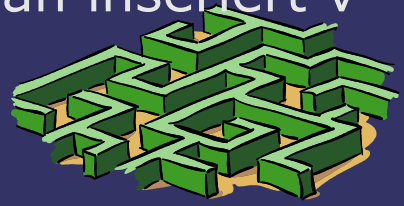
Die drei Heap Operationen werdem implementiert wiefolgt:

1.1. Insert

- um ein neuer Knoten einzufügen untersuchen wir die Werte “i” in fallender Reihenfolge, beginnend mit i=B, bis man den größten “i” findet, s.d. u(i)<d(v)

  d=Kosten;

- insert v in Bucket “i+1”
- um den Wert des Schlüssels (Kosten) abzusetzen entfernt man Knoten v vom aktuellen bucket j.
- danach erfolgt das Reduzieren des Schlüssels von v (man inseriert v im richtigen Bucket, man fängt mit Bucket i=j an)

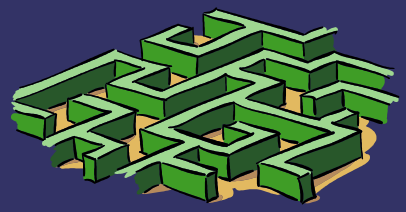


One-Level Radix Heap:

1.2. Insert Analyse

- die benötigte Zeit für insert und all den decrease Operationen für ein einziger Knoten v = O(logC) + O(1) pro decrease

- die Zeit für alle insert Operationen des Dijkstra Algorithm ist O(m + nlog(C))



One-Level Radix Heap:

2.1. Delete min

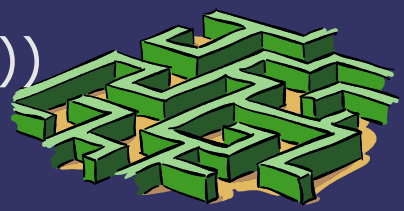
- falls Bucket 1 != leer, return irgendein Bucket vom Bucket 1;

- sonnst: finde ein nonempty Bucket mit dem kleinsten Index, sagen wir Bucket j;

- in Bucket j finde Knoten v mit kleinstem Schlüssel;

- speichere v, als Zurückgabe von delete min und verteile gebliebene Knoten aus Bucket j in den Buckets mit kleineren Indexes wiefolgt:

  - ersetze u(0) mit d(v)-1
  - u(1) mit d(v)
  - für i=2...j-1 : u(i)=min(u(i-1)+size(i), u(j))



One-Level Radix Heap:

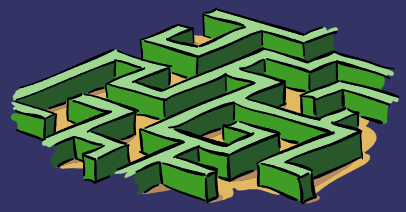
2.2. Delete min Analyse

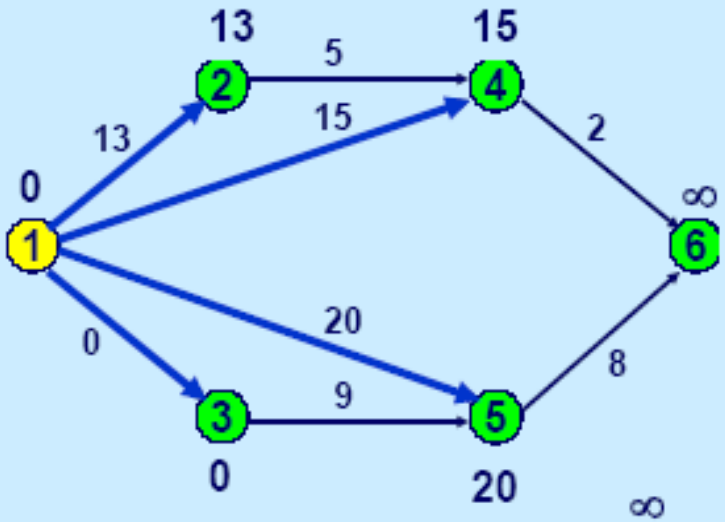
Aus der Ungleichung

$$\sum_{i=1}^{j-1} size(i) \geq \min\{size(j),\ C + 1\}$$

und aus der Eigenschaft(*)

=> wenn in einer delete min Operation j>=2, jeder Knoten von Bucket j wird in einerm Bucket mit streng kleinerem Index verstellt
=> Laufzeit: O(logC) per delete min+ O(logC) per Knoten
=> ein Dijkstra Durchlauf: O(n logC)
=> Total = O(m + n logC)

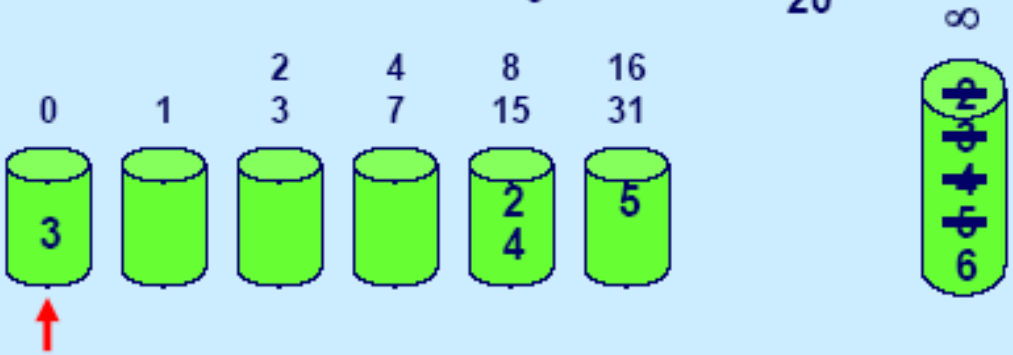




1. **Select Knoten mit kleinstem Schlüssel (hier: Knoten 3)**

2. **analysiere alle Wege aus 3**
=> Knoten 5 aus Bucket 6 in Buckt 5

3.**suche von links-> rechts das erste nicht leere Bucket & dessen min**
=> verteile Bucket Bereich auf den ersten i-1 Buckets um (hier: Startwert=d(5))



Berechung der neuen Abgränzungen der Buckets:

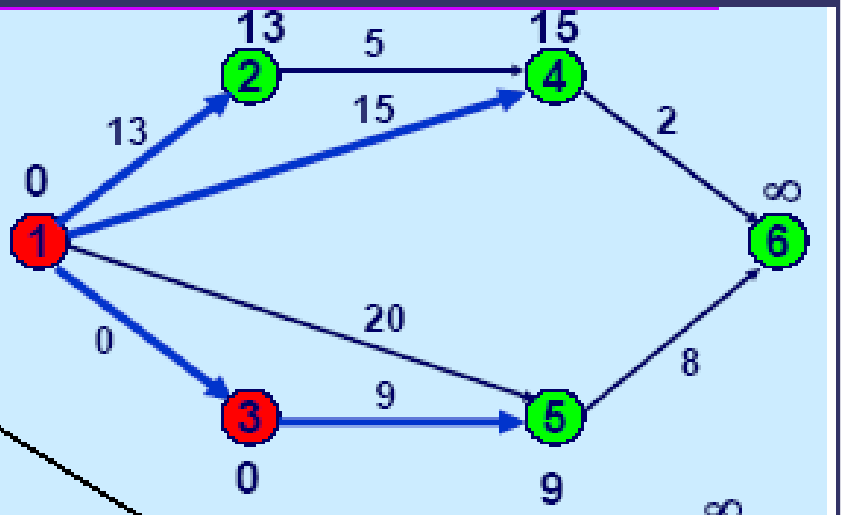
- u(i)=min(u(i-1)+size(i), u(j))

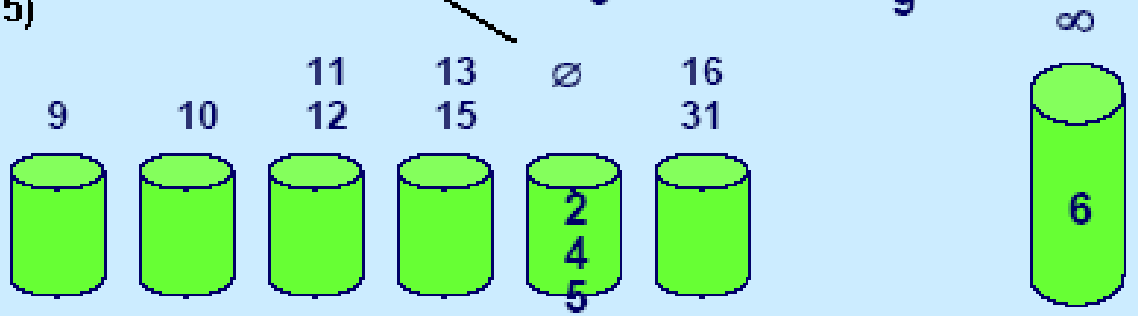
- u(i)=2^(i-1)-1

range(i)= leere Menge

weil: u(i-1)>=u(i)

(Redistribution des Knoten 5)

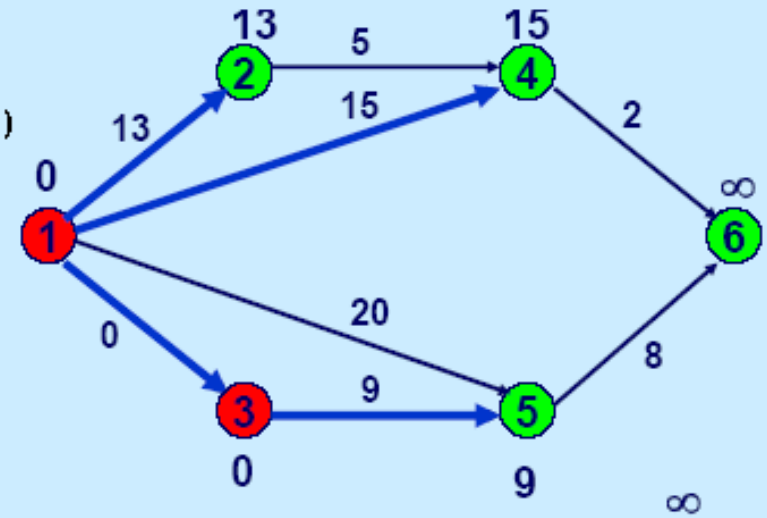


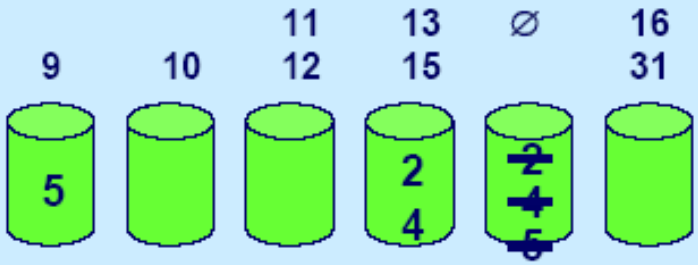


Fuege Knoten in zutreffende Buckets ein (dabei scannt man nach links)

Jetzt ist der meist linkste Bucket nicht mehr leer

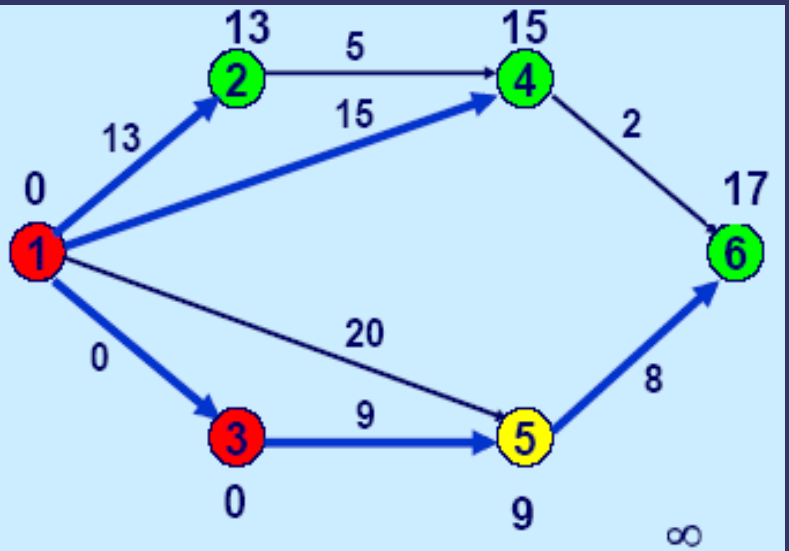
=> man waehlt irgendein Knoten aus diesem Bucket, diese Knoten wird jetzt analysiert (hier: Knoten 5)

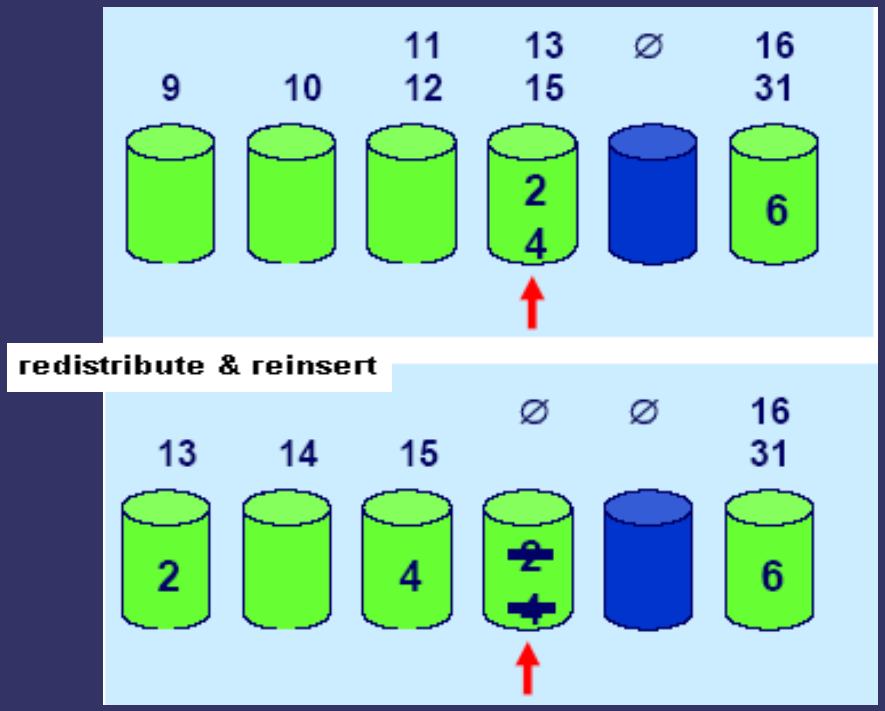


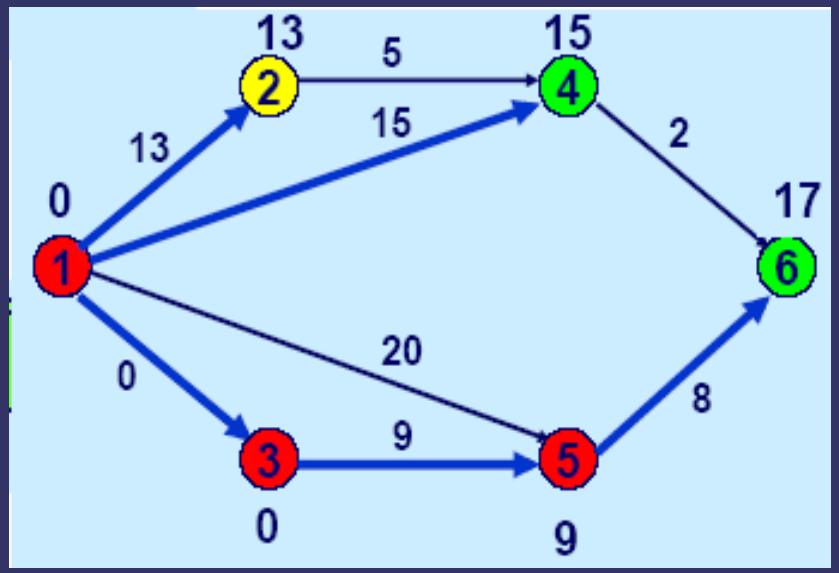


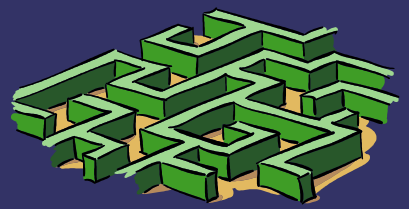
Man scannt jetzt alle Bogen die vom gewaehlten Knoten ausgehen

Fuege Knoten in zutreffende Buckets ein (dabei scannt man die Buckets nach links)




9
10
11
12
13
15
∅
16
31
2
4
6
redistribute & reinsert
13
14
15
∅
∅
16
31
2
4
2
6


13
15
5
13
15
2
0
17
20
0
8
9
0
9



Two-Level Radix Heap:

- Um die Laufzeit von Dijkstra zu senken muss man die Nummer der “reinserts” der Knoten in den Buckets reduzieren

<=> Vergrößerung der Bucketgröße

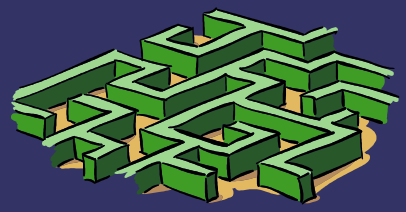
=> Ungleichung (1) nicht mehr erfuellt
=> Teilung der Buckets in K gleichgrosse Segmente

- = Sammlung of B=log(C+1)+1 , (Basis K) Buckets, indexiert von 1 bis B

- Jedes Bucket “i” hat eine Größe size(i), die wiefolgt definiert ist:

size(i)= K^(i)
für 1<=i<=B-1;
size(B)=nC+1

$$u(j) = \sum_{i=1}^{j} K^i - 1$$

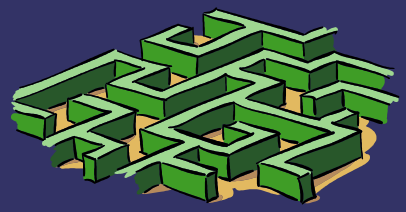


Two-Level Radix Heap:

1. Insert Operation

- wie im One Level Radix Heap, aber jetzt wird v im segment (i, k) eingefügt

=> Laufzeit= O(m + nlogC) (Basis K)



Two-Level Radix Heap:

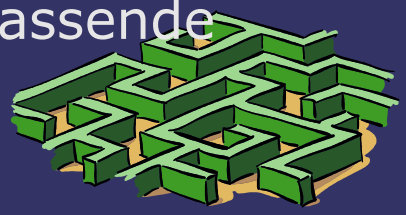
2.1. Delete min Operation

- wie im One Level Radix Heap, aber jetzt nur der Inhalt eines singel Segmentes wird verteilt
- man findet den ersten nicht leeren Bucket j, also (j, k)

  - ist j=B => K=1
  - ist j=1 => entferne & gebe zurück jedwelches Knoten aus dem Segment (j, k)

  sonnst: finde v im Segment (j, k) mit min Schlüssel

- Redefiniere u(i) für 0<=i<=j-1 wie im One Level Heap
- Verteile alle Knoten aus Segment (j, k),außer v, in passende Segmente, die im Bucket von 1..j-1 liegen



Two-Level Radix Heap:

2.2 Delete min Analyse

- ein Bit, dass sagt ob ein Bucket leer ist oder nicht -

=> Laufzeit für finden von j in delete min: O(B)

- jedes verteilte Knoten wird in einem niedrigeren Bucket versetzt
  => Anzahl der Versetzungen: O(Bn)

=> Laufzeit für alle delete min Operationen:
O(Bn)+Zeit für n Schritte der Form "finde das erste nicht leere Segment in einem gegebenen Bucket" (1Schritt: O(k))

=> Laufzeit von Dijkstra: O(m + (B + K)n)

(wenn jedes Segment ist gefunden nachdem man alle Segmente im Bucket durchgeht)

wählt man K proportional mit logC/log log C
=> Laufzeit: O(m + nlogC/log logC)