Algorithms and Datastructures

Open Addressing, Priority Queue

Albert-Ludwigs-Universität Freiburg

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Datastructures, November 2017

Structure

Hashing:

- No hash function is good for all key sets!
  - This cannot work, because a big universe is mapped onto a small set: $|\mathbb{U}| > m$
- For random key sets also simple hash function work, e.g.

$$\Rightarrow h(x) = x \mod m$$

  - Then the random keys make sure that it is distributed evenly
- To find a good hash function for every key set universal hashing is needed
  - Then however, for a fixed set of keys not every hash function is suitable, but only some

Rehashing:

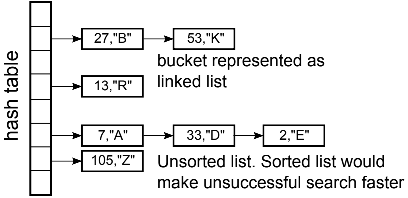
- It is possible to get bad hash functions with universal hashing, but it is unlikely
- This is determinable by monitoring the maximum bucket size
- If a pre-defined level is exceeded, then a rehash is performed

How to rehash?

- New hash table with a new random hash function
- Copy elements into the new table
  - Expensive but happens not often
  - Therefore the average cost is low
  - Look at amortized analysis in the next lecture

Buckets as linked list:

- Each bucket is a linked list
- Colliding keys are inserted into the linked list of a bucket, either sorted or appended at the end



- Operations in $O(1)$ are possible if a suitable tablesize and hashfunction is selected
- Worst case $O(n)$, e.g. tablesize of 1
- Dynamic number of elements is possible

- For colliding keys we choose a new free entry
- Static, fixed number of elements
- The **probe sequence** determines for each key, in which sequence all hash table entries are searched for a free bucket
  - If a entry is already occupied, then iteratively the following entry can be checked. If a free entry is found the element is inserted
  - If element is not found at the corresponding table entry, even if the entry is occupied, then probing has to be performed until the element or a free entry have been found

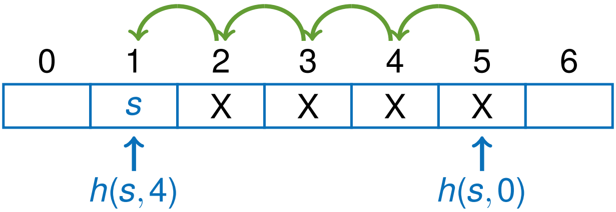
Definitions:

$h(s)$ Hash function for key s

$g(s,j)$ Probing function for key s with overflow positions $j \in \{0,\ldots,m-1\}$ e.g. g(s,j)=j

- The **probe sequence** is calculated by

$$h(s,j) = (h(s) - g(s,j)) \mod m \in \{0,\ldots,m-1\}$$



```
def insert(s, value):
    j = 0

    while t[(h(s) - g(s, j)) mod m] \
            is not None:
        j += 1

    t[(h(s) - g(s, j)) mod m] \
        = (s, value)
```

```
def lookup(s):
    j = 0

    while t[(h(s) - g(s, j)) mod m] \
            is not None:

        if t[(h(s) - g(s, j)) mod m][0] == s:
            return t[(h(s) - g(s, j)) mod m]

        j += 1

    return None
```

Hashing

Open Addressing - Linear Probing

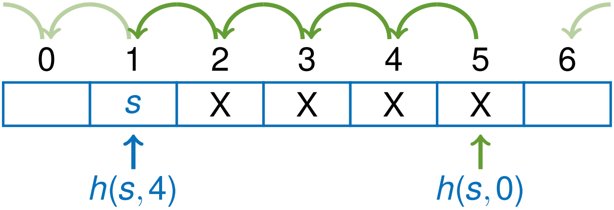


Figure: Linear probe sequence

- Check the element with lower index: $g(s,j) := j$
  $\Rightarrow$ Hash function: $h(s,j) = (h(s) - j) \mod m$
- This leads to the following probe sequence

$$h(s), h(s)-1, h(s)-2, \ldots, \underbrace{0, m-1}_{\text{clipping}}, m-2, \ldots, h(s)+1$$

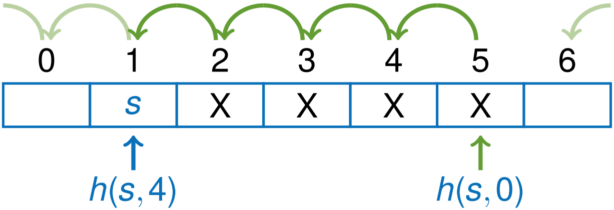


Figure: Linear probe sequence

- Can result in primary clustering
- Dealing with a hash collision will result in a higher probability of hash collisions in close entries

Hashing

Open Addressing - Linear Probing

Example:

- Keys: $\{12, 53, 5, 15, 2, 19\}$
- Hash function: $h(s,j) = (s \mod 7 - j) \mod 7$

- t.insert(12, "A"), $h(12,0) = 5$

0	1	2	3	4	5	6
					12, A	

- t.insert(53, "B"), $h(53,0) = 4$

				53, B	12, A	

Figure: Probe/Insertion sequence on a hash map

Example:

- Hash function: $h(s, j) = (s \mod 7 - j) \mod 7$

- t.insert(5, "C"), $h(5,0) = 5$, $h(5,1) = 4$, $h(5,2) = 3$

0	1	2	3	4	5	6
			5, C	53, B	12, A	

- t.insert(15, "D"), $h(15,0) = 1$

	15, D		5, C	53, B	12, A	

Figure: Probe/Insertion sequence on a hash map

Example:

- Hash function: $h(s,j) = (s \mod 7 - j) \mod 7$

- t.insert(2, "E"), $h(2,0) = 2$

0	1	2	3	4	5	6
	15, D	2, E	5, C	53, B	12, A	

- t.insert(19, "F"), $h(19,0) = 5$, $h(19,1) = 4$, $h(19,2) = 3$, $h(19,3) = 2$, $h(19,4) = 1$, $h(19,5) = 0$

19, F	15, D	2, E	5, C	53, B	12, A	

Figure: Probe/Insertion sequence on a hash map

Squared probing:

- Motivation: Avoid local clustering

$$g(s,j) := (-1)^j \left\lceil \frac{j}{2} \right\rceil^2$$

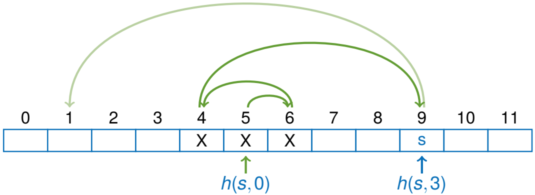


Figure: Squared probe sequence

- This leads to the following probe sequence

$h(s),\ h(s)+1,\ h(s)-1,\ h(s)+4,\ h(s)-4,\ h(s)+9,\ h(s)-9,\ \ldots$

Squared probing:

$$g(s,j) := (-1)^j \left\lceil \frac{j}{2} \right\rceil^2$$

- If m is a prime number for which $m = 4 \cdot k + 3$ then the probe sequence is a permutation of the indices of the hash tables
- Alternatively: $h(s,j) := (h(s) - c_1 \cdot j + c_2 \cdot j^2) \mod m$
- Problem of secondary clustering
  No local clustering anymore, but keys with same hash value have similar probe sequence

Uniform Probing:

- Motivation: So far uses function $g(s,j)$ only the step counter j for linear and squared probing
  $\Rightarrow$ The probe sequence is independent of the key s
- Uniform probing computes the sequence $g(s,j)$ of permutations of all possible indices in dependency on key s
- **Advantage:** Prevents clustering because different keys with the same hash value do not produce the same probe sequence
- **Disadvantage:** Hard to implement

Double Hashing:

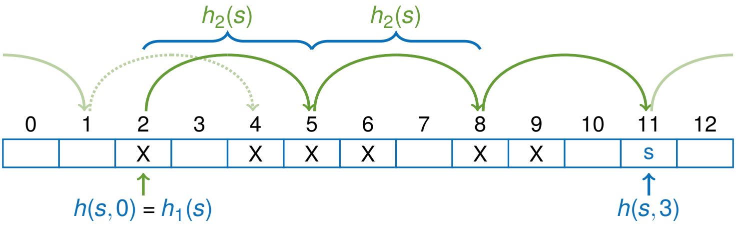


Figure: Double hashing probe sequence

- Motivation: Consider key s in probe sequence
- Use two independent hash functions $h_1(s), h_2(s)$
- Hash function: $h(s,j) = (h_1(s) + j \cdot h_2(s)) \mod m$

Double Hashing:

- Hash function: $h(s,j) = (h_1(s) + j \cdot h_2(s)) \mod m$
- probe sequence:

$$h_1(s),\ h_1(s) + h_2(s),\ h_1(s) + 2 \cdot h_2(s),\ h_1(s) + 3 \cdot h_2(s),\ \dots$$

- Works well in practical use
- This method is an approximation of uniform probing

Example:

$$h_1(s) = s \mod 7$$
$$h_2(s) = (s \mod 5) + 1$$
$$h(s,j) = h_1(s) + j \cdot h_2(s) \mod 7$$

Table: Comparing both hash functions

s	10	19	31	22	14	16
$h_1(s)$	3	5	3	1	0	2
$h_2(s)$	1	5	2	3	5	2

- The efficiency of double hashing is dependent on $h_1(s) \neq h_2(s)$

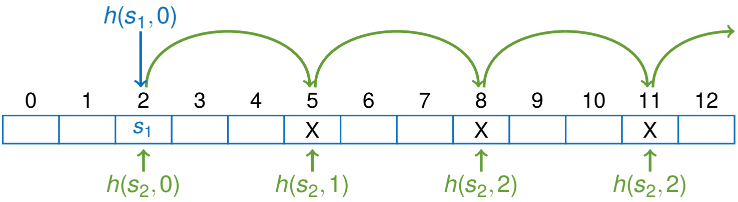


Figure: Double hashing

Double hashing by Brent:

- Motivation:
  Because different keys have different probe sequences, the sequence of the insertions has impact on efficiency of a sucessful search

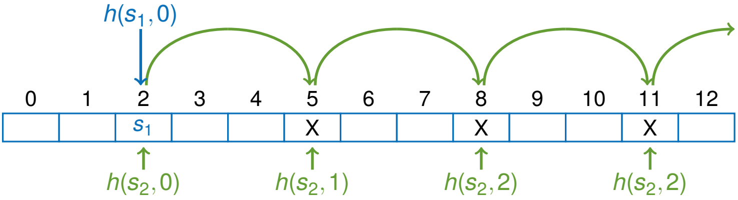


Figure: Double hashing

Example:

- The key s_1 is inserted at position $p_1 = h(s_1, 0)$
- The hash function for s_2 also results in $p_2 = h(s_2, 0) = p_1$
- The locations $h(s_2, j)$, $j \in \{1, \ldots, n\}$ are also occupied
- If we insert s_2 at position $h(s_2, n+1)$ the search will be inefficient

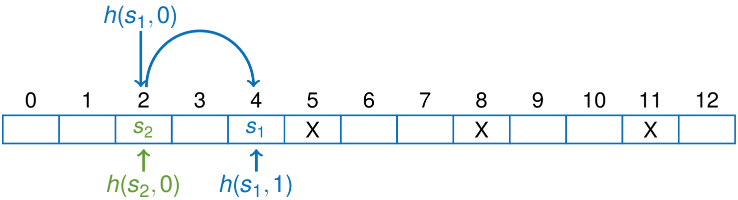


Figure: Double hashing by Brent

- Reversed sequence of keys would have been better
- **Brents Idea:**
  - Test if location $h(s_1, 1)$ is free
  - If yes, move s_1 from $h(s_1, 0)$ to $h(s_1, 1)$ and insert s_2 at $h(s_2, 0)$

Idea:

- Motivation: Colliding elements are inserted in the hashtable sorted.
- Therefore, in case of an unsuccessful search of elements in combination with linear probing or double hashing, aborting is earlier possible because single probing steps have a fixed length

Implementation:

- Compare both keys if a collision occurs
- Insert the smaller key at p_1
- Search a position based on the diversion order for the bigger key

Example:

- The key 12 is saved at position $p_1 = h(12, 0)$
- We insert the key 5 into the hash map
- We assume $h(5, 0)$ results in location p_1
- Because $5 < 12$ we insert the key 5 at position p_1
- For the key 12 we iterate through the sequence

$$h(12, 1), h(12, 2), h(12, 3), \dots$$

Motivation:

- Having similiar length of probe sequences for all elements. Total costs stay the same, but they are distributed evenly. Results in approximately similar search times for all elements

Implementation:

- If two keys s_1, s_2 collide ($p_1 = h(s_1, j_1) = h(s_2, j_2)$) we compare the length of the sequence (j_1 or j_2)
- The key with the bigger search sequence is inserted at p_1 The other key is assigned a new location based on the sequence

Example:

- The key 12 is saved at position $p_1 = h(12,7)$
- We insert the key 5 into the hash map
- We assume $h(5,0)$ results in location p_1
- Because $j_1 < j_2$ $(0 < 7)$ the key 12 stays at position p_1
- For the key 5 we iterate through the sequence

$$h(5,1),\ h(5,2),\ h(5,3),\ \ldots$$

Problem:

- The key s_1 is inserted at position p_1
- The key s_2 returns the same hash value, but is inserted at position p_2 because of the probing order
- If s_1 is removed, it is impossible to find s_2

Solution:

- **Remove:** Elements are marked as removed, but not deleted
- **Inserting:** Elements marked as removed will we overwritten

Bucket as linked list: (dynamic, number of elements variable)

- Save colliding elements as linked list

Open hashing: (static, number of elements fixed)

- Determine a probe sequence, permutation of all hash values
- Linear, quadratic probing:
  - Easy to implement
  - Raise the probability of collisions because probing order does not depend on the key

UNI FREIBURG

Open hashing: (static, number of elements fixed)

- Uniform probing, double hashing:
  - Different probing orders for different keys
  - Avoids clustering of elements

Improving efficiency: (Brent, Ordered Hashing)

- Improve search efficiency by sorting colliding insertions
  - Abortion of unsuccessfull search
  - Search sequence length balancing

Hashing:

- Efficient fo dictionary operations:
  `Insert:` $O(1) \ldots O(n)$
  `Search:` $O(1) \ldots O(n)$
  `Remove:` $O(1) \ldots O(n)$
- Direct access of all elements in a hash table
- Using a hash function to find the position (hash value) in the hash table
- Hash function, size of the hash table and strategy to avoid hash collisions influence the efficiency of the datastructure

Definition:

- A priority queue saves a set of elements
- Each element contains a key and a value like a map
- There is a total order (like $\leq$) defined on the keys

UNI FREIBURG

Definition:

- The priority queue supports the following operations:

`insert(key, value)`: Inserts a new element into the queue
`getMin()`: Returns the element with the smallest key
`deleteMin()`: Removes the element with the smallest key

- Sometimes additional operations are defined:

`changeKey(item, key)`: Changes the key of the element
`remove(item)`: Removes the element from the queue

Special features:

- Multiple elements with the same key
  - No problem and for many applications necessary
  - If there is more than one element with the smallest key

    `getMin()`: Returns just one of the possible elements

    `deleteMin()`: Deletes the element returned by `getMin`

- Argument of `changeKey` and `remove` operations
  - There is no **quick-access** to a element in the queue
  - Thats why `insert` and `getMin` return a reference (handle, accessor object)
  - `changeKey` and `remove` take this reference as argument
  - Therefore each element has to store its current position in the heap.

Priority Queue

Python

```
from queue import PriorityQueue

q = PriorityQueue()

e1 = (5, "A") # element with priority 5
q.put(e1); # insert element e1

# remove and return the lowest item
e2 = q.get()
```

Priority Queue

Application Example

Example 1:

- Calculation of the sorted union of k sorted lists (multi-way merge or k-way merge)

L_1 :

3	5	8	12	...

L_3 :

1	10	11	24	...

L_2 :

4	5	6	7	...

$\Rightarrow R$:

1	3	4	5	5	6	7	8	10	...

Figure: 3-way merge

Example 1:

- Calculation of the sorted union of k sorted lists (multi-way merge or k-way merge)
- Runtime: N = length of resulting list
  - Trivial: $\Theta(N \cdot k)$, minimum calculation $\Theta(k)$
  - Priority queue: $\Theta(N \cdot \log k)$, minimum calculation $\Theta(\log k)$

Example 2:

- For example Dijkstra's algorithm for computing the shortest path ($\leftarrow$ following lecture)
- Among other applications it can be used for sorting

Idea:

- Save elements as tuples in a binary heap
- Summary from lecture 1 (*HeapSort*):
  - Nearly complete binary tree
  - Heap condition:
    The key of each node $\leq$ the keys of the children

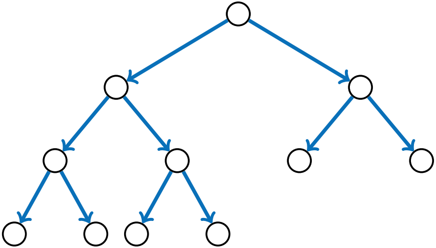
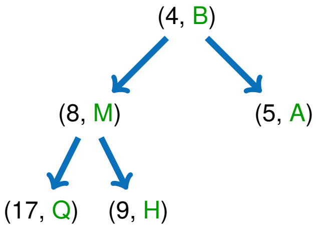


Figure: Heap with 11 nodes

Priority Queue
Implementation



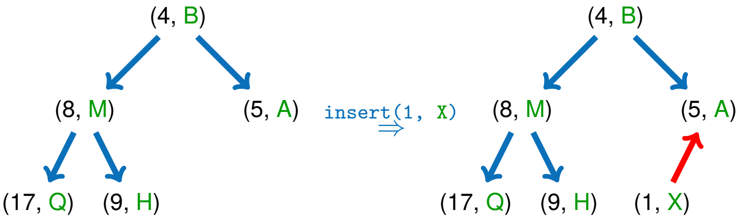
0	1	2	3	4
4, B	8, M	5, A	17, Q	9, H

Figure: Min heap stored in array

Storing a binary heap:

- Number nodes from top to bottom and left to right starting with 0 and store entries in array
- Children of node i are the nodes $2i+1$ and $2i+2$
- Parent node of node i is $\mathrm{floor}((i-1)/2)$

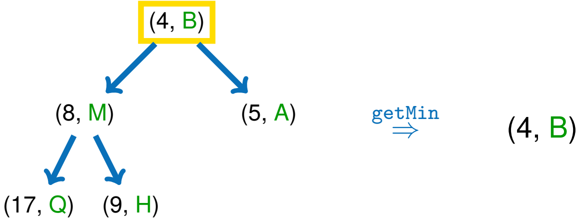
Inserting an element: `insert(key, item)`



- Append the element at the end of the array
- The heap condition may be violated, but only at the last index
- Repair heap condition ⇒ We will see later how to do this

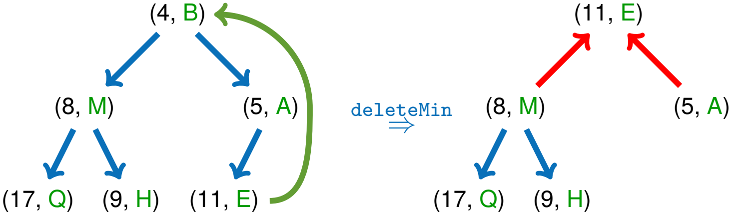
Priority Queue
Implementation

Returning the minimum: `getMin()`



- Else return the first element
- If the heap is empty return None

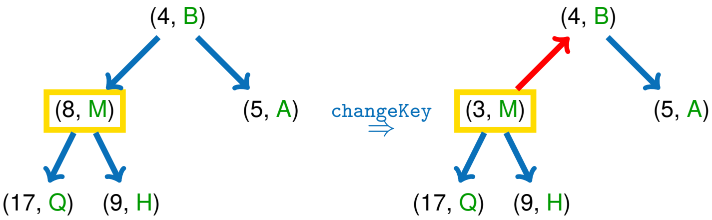
Removing the minimum: `deleteMin()`



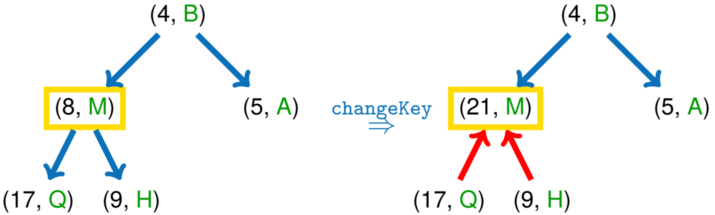
- Deleting the element with the lowest key
- Swap the last element with the first element and shrink the heap by one
- The heap condition may be violated, but only at the first index
- Repair heap condition

Changing the key (priority): `changeKey(item, key)`

- The element (queue item) is given as argument
- Replace the key of the element
- The heap condition may be violated, but only at the element index and only in one direction (up / down)
- Repair heap condition

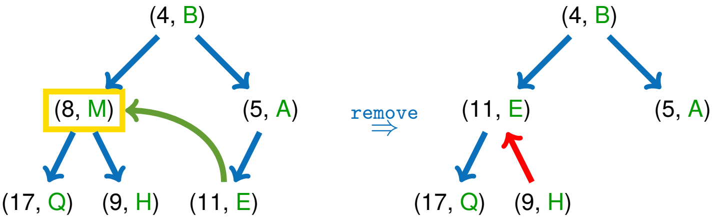


Changing the key (priority): `changeKey(item, key)`



- The heap condition may be violated, but only at the element index and only in one direction (up / down)
- Repair heap condition

Removing an element: `remove(item)`



- The element (queue item) is given as argument
- Replace the element with the last element and shrink the heap by one
- The heap condition may be violated, but only at the element index and only in one direction (up / down)
- Repair heap condition

Repairing after modifying operations:

- The heap condition can be violated after using `insert`, `deleteMin`, `changeKey`, `remove`, but only at one known position with index i
- Heap conditions can be violated in two directions:
  - Downwards: The key at index i is not $\leq$ than the value of its children
  - Upwards: The key at index i is not $\geq$ than the value of its parent
- We need two repair methods: `repairHeapUp`, `repairHeapDown`

`repairHeapDown:`

- Sift the element until the heap condition is valid
  - Change node with child, which has the lower key of both children
  - If the heap condition is violated repeat for the child node

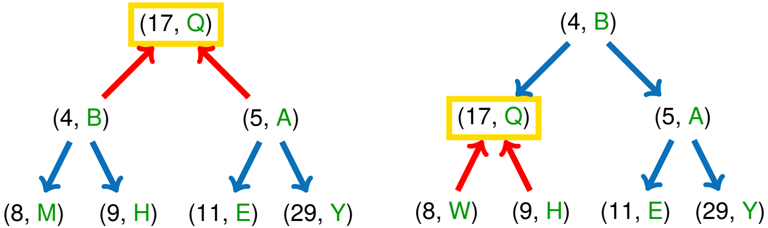


Figure: Repairing the heap downwards

`repairHeapDown:`

- Sift the element until the heap condition is valid
  - Change node with child, which has the lower key of both children
  - If the heap condition is violated repeat for the child node

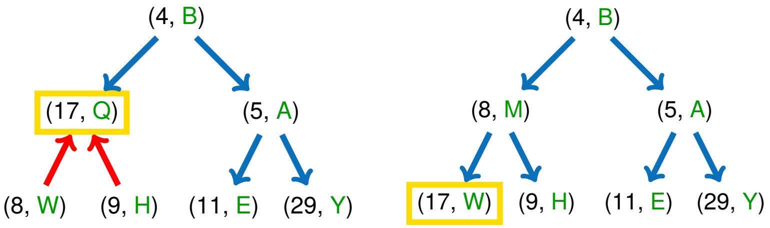


Figure: Repairing the heap downwards

`repairHeapUp:`

- Change node with parent
- If the heap condition is violated repeat for parent node

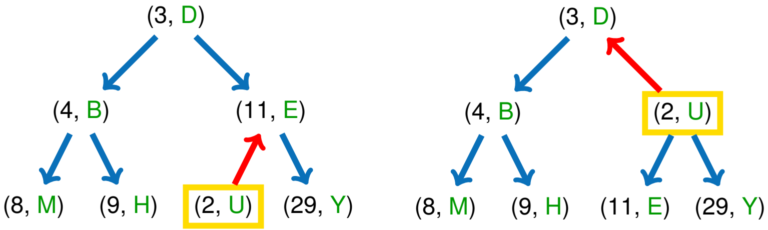


Figure: Repairing the heap upwards

`repairHeapUp:`

- Change node with parent
- If the heap condition is violated repeat for parent node

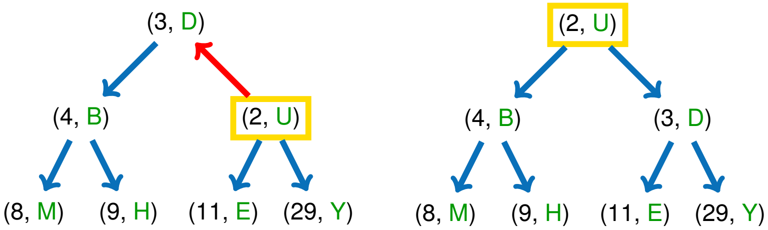


Figure: Repairing the heap upwards

Index of a priority queue item:

- Attention: For `changeKey` and `remove` the item has to "know" where it is located in the heap
- Remember for `repairHeapUp` and `repairHeapDown`: Update the index if moving an heap element

```python
class PriorityQueueItem:

    """Provides a handle for a queue item.

    This handle can be used to remove or
    update the queue item.
    """

    def __init__(self, key, value, index):
        self.key = key
        self.value = value
        self.index = index
```

Summary lecture 1:

- A full binary tree with n elements, has a depth of $O(\log n)$
- The maximum distance from the root to a leaf can be $O(\log n)$ elements
- Repairing the heap upwards and downwards:
  We have only one path to traverse: $O(\log n)$

Runtime for methods

- `insert`, `deleteMin`, `changeKey`, `remove`:
  We have to repair the heap: $O(\log n)$
- `getMin`: Return the element at index 0: $O(1)$

Priority Queue
Complexity

Improvements (Fibonacci heaps):

- `getMin`, `insert` and `decreaseKey` in amortized time of $O(1)$
- `deleteMin` in amortized time $O(\log n)$

Practical experience:

- The binary heap is simpler: Costs for managing the structure are low
- If the number of elements is relatively small so the difference is negligible
- Example:
  - For $n = 2^{10} \approx 1,000$ is the the depth $\log_2 n$ only 10
  - For $n = 2^{20} \approx 1,000,000$ is the depth $\log_2 n$ only 20

Further Literature

- **General**

[CRL01] Thomas H. Cormen, Ronald L. Rivest, and Charles E. Leiserson.
Introduction to Algorithms.
MIT Press, Cambridge, Mass, 2001.

[MS08] Kurt Mehlhorn and Peter Sanders.
Algorithms and data structures, 2008.
https://people.mpi-inf.mpg.de/~mehlhorn/ftp/Mehlhorn-Sanders-Toolbox.pdf.

Further Literature

- **Priority Queue - Implementations / API**

[Cpp] C++ - priority_queue
http://www.sgi.com/tech/stl/priority_queue.html

[Jav] Java - PriorityQueue
https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html

[Pyt] Python - PriorityQueue
https://docs.python.org/3/library/queue.html#queue.PriorityQueue