Algorithms and Data Structures

Open Addressing, Priority Queue

Albert-Ludwigs-Universität Freiburg

UNI FREIBURG

Prof. Dr. Rolf Backofen

Bioinformatics Group / Department of Computer Science

Algorithms and Data Structures, November 2018

Structure

Hashing
- Recapitulation
- Treatment of hash collisions
- Open Addressing
- Summary

Priority Queue
- Introduction

Hashing:

Hashing:

- No hash function is good for all key sets!

Hashing:

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

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 functions work, e.g.

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

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 functions work, e.g.

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

  - Then the random keys make sure that it is distributed evenly

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 functions 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

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 functions 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:

Rehashing:

- It is possible to get bad hash functions with universal hashing, but it is unlikely

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

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

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

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

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 does not happen often

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 does not happen often
  - Therefore the average cost is low

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 does not happen often
  - Therefore the average cost is low
  - Look at amortized analysis in the next lecture

Structure

Hashing

- Recapitulation
- Treatment of hash collisions
- Open Addressing
- Summary

Priority Queue

- Introduction

Buckets as linked list:

Buckets as linked list:

- Each bucket is a linked list

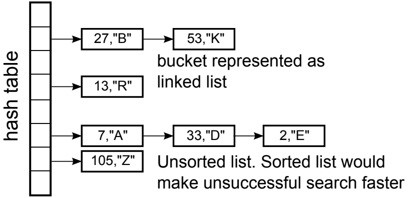
UNI FREIBURG

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

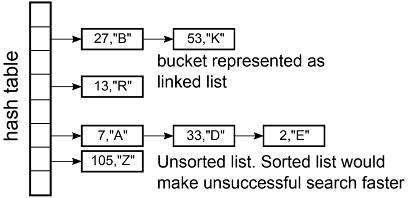
Buckets as linked list:

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



Buckets as linked list:

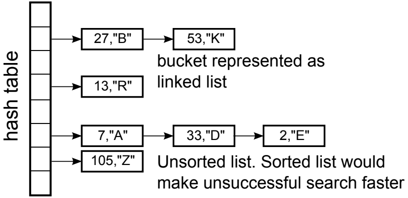
- Each bucket is a linked list
- Colliding keys are inserted into the linked list of a bucket, either sorted or appended at the end



- Operations in $O(1)$ are possible if a suitable table size and hash function is selected

Buckets as linked list:

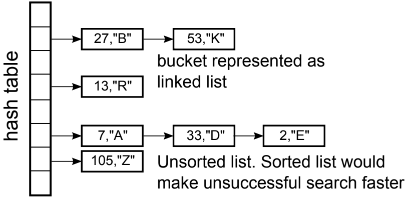
- Each bucket is a linked list
- Colliding keys are inserted into the linked list of a bucket, either sorted or appended at the end



- Operations in $O(1)$ are possible if a suitable table size and hash function is selected
- Worst case $O(n)$, e.g. table size of 1

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 table size and hash function is selected
- Worst case $O(n)$, e.g. table size of 1
- Dynamic number of elements is possible

Structure

Hashing
Recapitulation
Treatment of hash collisions
Open Addressing
Summary

Priority Queue
Introduction

- For colliding keys we choose a new free entry

- For colliding keys we choose a new free entry
- Static, fixed number of elements

- 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

- 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 an entry is already occupied, then iteratively the following entry is checked. If a free entry is found the element is inserted

- 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 an entry is already occupied, then iteratively the following entry is 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 has been found

Definitions:

Definitions:

$h(s)$ Hash function for key s

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

Hashing
Open Addressing

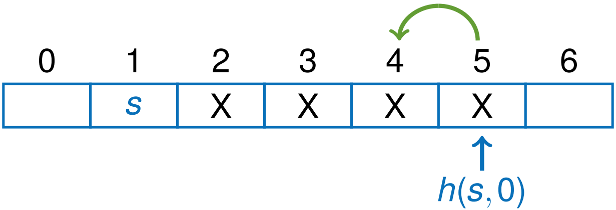
Definitions:

$h(s)$ 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:
                        j += 1
        if t[(h(s) - g(s, j)) mod m][0] == s:
            return t[(h(s) - g(s, j)) mod m]
    return None
```

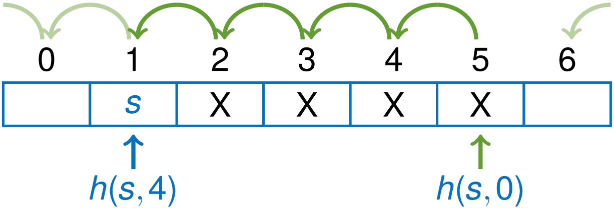


Figure: Linear probe sequence

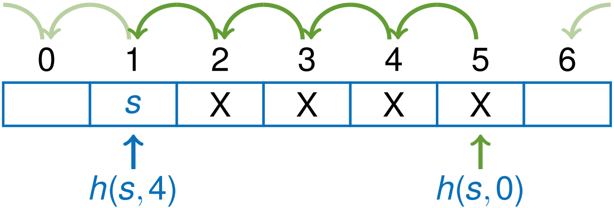


Figure: Linear probe sequence

- Check the element with lower index: $g(s,j) := j$
  $\Rightarrow$ Hash function: $h(s,j) = (h(s) - j) \mod m$

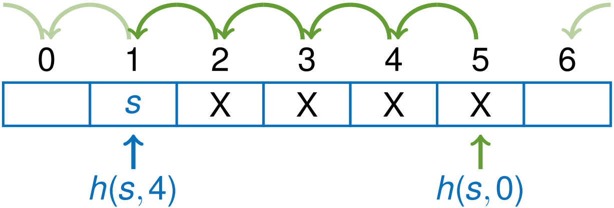


Figure: Linear probe sequence

- 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

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

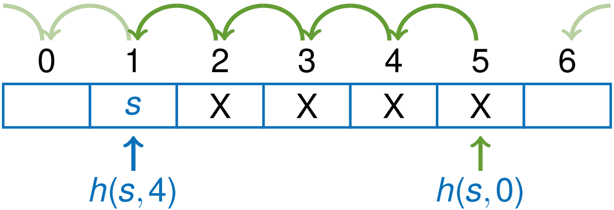


Figure: Linear probe sequence

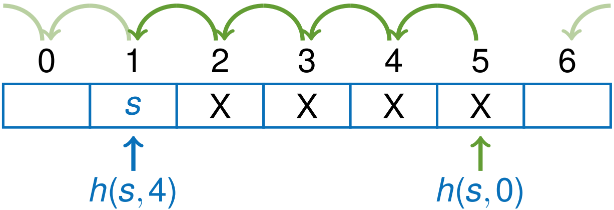


Figure: Linear probe sequence

- Can result in primary clustering

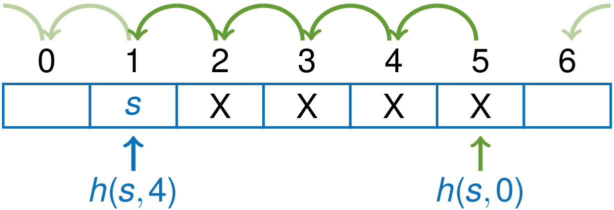


Figure: Linear probe sequence

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

Example:

Example:

- Keys: $\{12, 53, 5, 15, 2, 19\}$

Example:

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

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	

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

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	

Hashing

Open Addressing - Linear Probing

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

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	

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:

UNI FREIBURG

Squared probing:

- Motivation: avoid local clustering

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

Hashing

Open Addressing - Squared Probing

Squared probing:

- Motivation: avoid local clustering

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

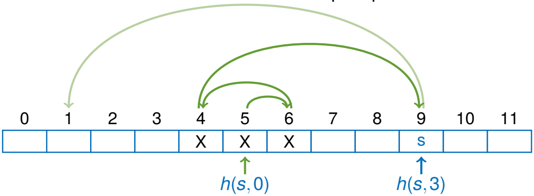


Figure: Squared probe sequence

Squared probing:

- Motivation: avoid local clustering

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

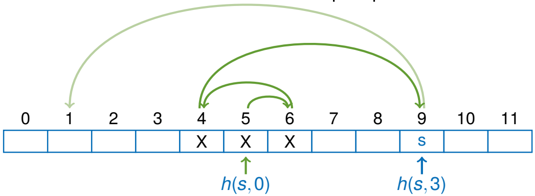


Figure: Squared probe sequence

- This leads to the following probe sequence

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

UNI FREIBURG

Squared probing:

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

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

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: $h(s,j) := (h(s) - c_1 \cdot j + c_2 \cdot j^2) \mod m$

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:

Uniform Probing:

- Motivation: so far function $g(s, j)$ uses only the step counter j for linear and squared probing
  $\Rightarrow$ The probe sequence is independent of the key s

Uniform Probing:

- Motivation: so far function $g(s,j)$ uses 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 dependent on key s

Uniform Probing:

- Motivation: so far function $g(s,j)$ uses 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 dependent on key s
- **Advantage:** prevents clustering because different keys with the same hash value do not produce the same probe sequence

Uniform Probing:

- Motivation: so far function $g(s,j)$ uses 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 dependent 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:

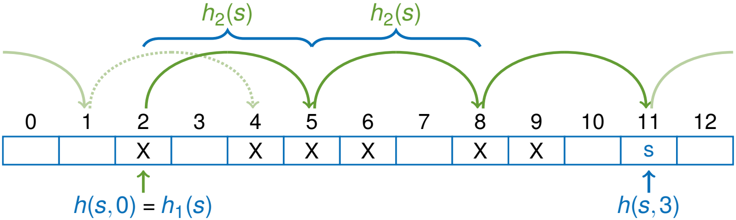


Figure: double hashing probe sequence

Double Hashing:

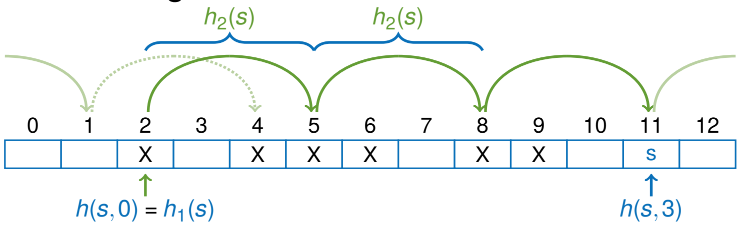


Figure: double hashing probe sequence

- Motivation: consider key s in probe sequence

Double Hashing:

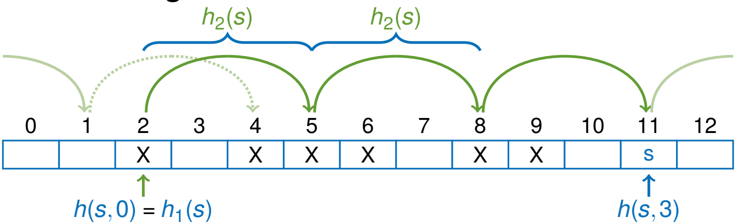


Figure: double hashing probe sequence

- Motivation: consider key s in probe sequence
- Use two independent hash functions $h_1(s), h_2(s)$

Hashing

Open Addressing - Double Hashing

Double Hashing:

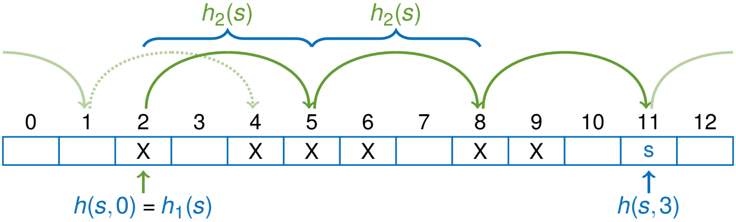


Figure: double hashing probe sequence

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

UNI FREIBURG

Double Hashing:

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

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

Double Hashing:

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

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

- Works well in practical use

Double Hashing:

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

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

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

Example:

Example:

$$
\begin{aligned}
h_1(s) &= s \mod 7 \\
h_2(s) &= (s \mod 5) + 1 \\
h(s,j) &= h_1(s) + j \cdot h_2(s) \mod 7
\end{aligned}
$$

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 $h_1(s) \neq h_2(s)$

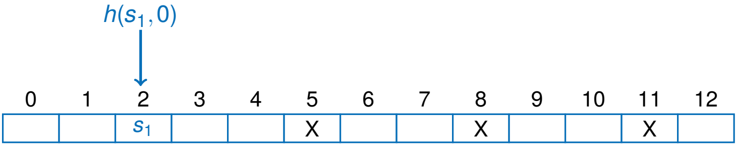


Figure: double hashing

Double hashing by Brent:

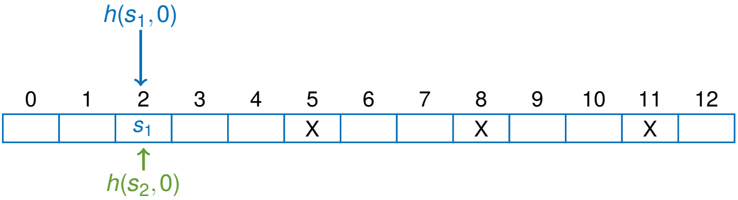


Figure: double hashing

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

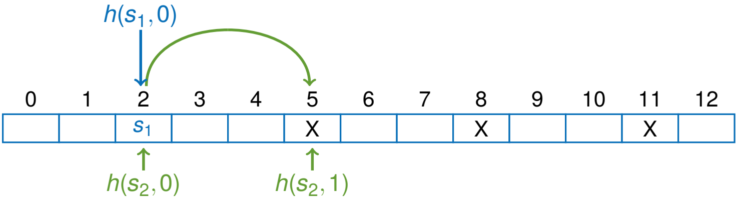


Figure: double hashing

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

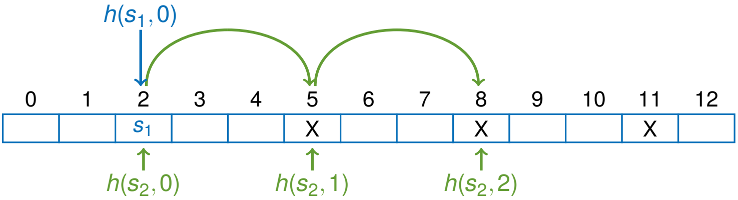


Figure: double hashing

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

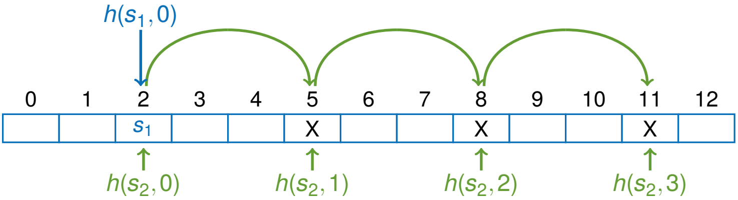


Figure: double hashing

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

Hashing

Open Addressing - Double Hashing - Optimization

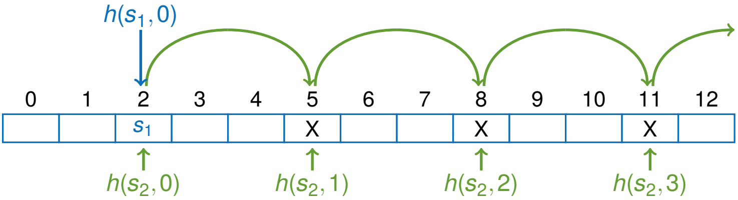


Figure: double hashing

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

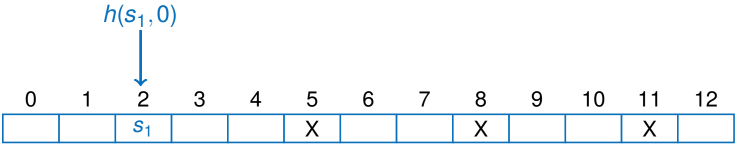


Figure: double hashing

Example:

- The key s_1 is inserted at position $p_1 = h(s_1, 0)$

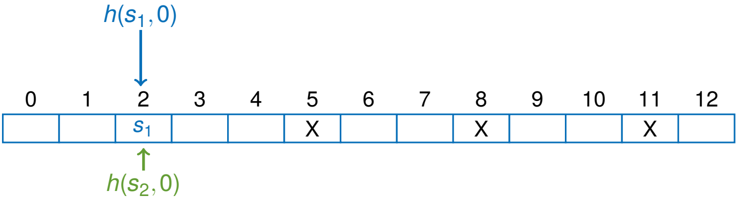


Figure: double hashing

Example:

- The key s_1 is inserted at position $p_1 = h(s_1, 0)$
- The hash function for s_2 also results in $p_2 = h(s_2, 0) = p_1$

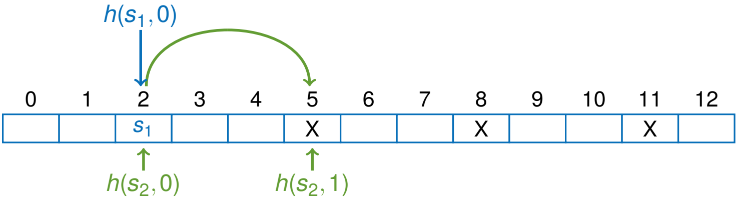


Figure: double hashing

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

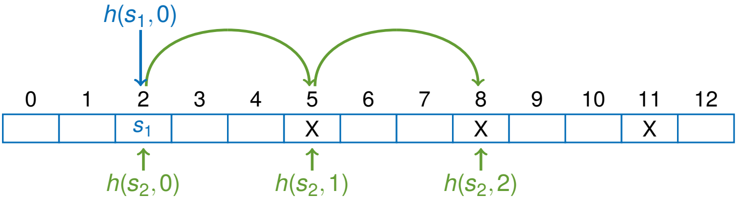


Figure: double hashing

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

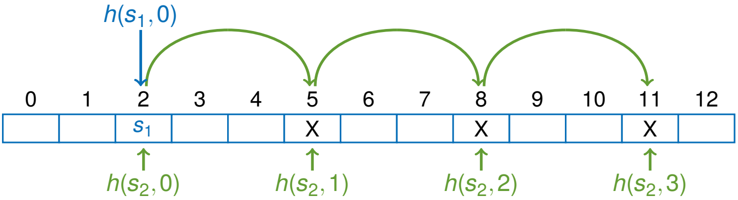


Figure: double hashing

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

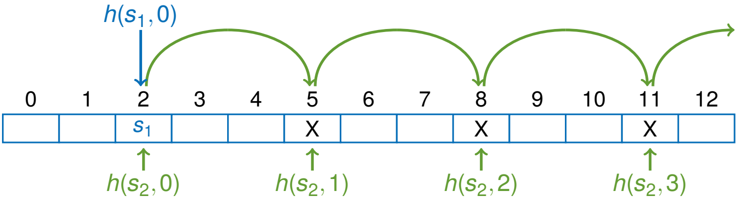


Figure: double hashing

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 $h(s_2, j)$, $j \in \{1, \dots, n\}$ are also occupied
- If we insert s_2 at position $h(s_2, n+1)$ the search will be inefficient

0	1	2	3	4	5	6	7	8	9	10	11	12
		s_2			X			X			X	

$h(s_2, 0)$

Figure: double hashing by Brent

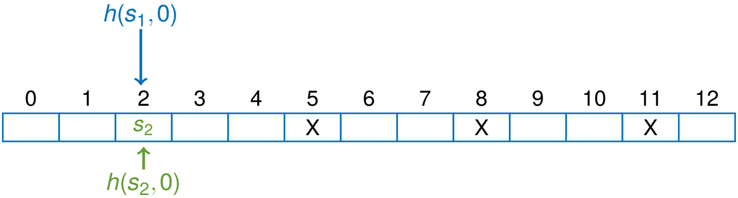


Figure: double hashing by Brent

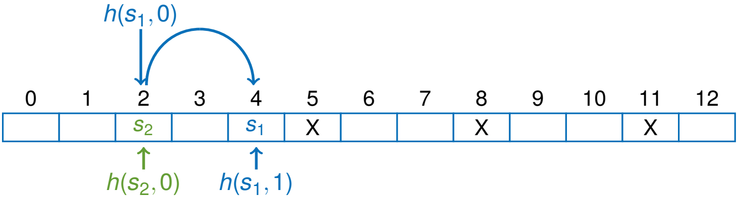


Figure: double hashing by Brent

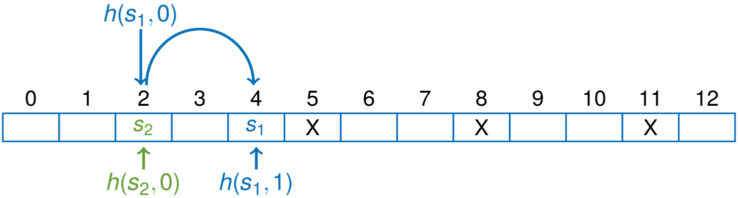


Figure: double hashing by Brent

- Reversed sequence of keys would have been better

Hashing

Open Addressing - Double Hashing - Optimization

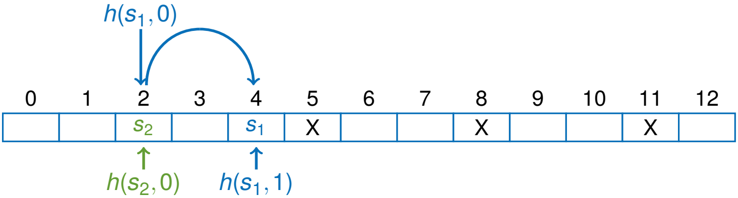


Figure: double hashing by Brent

- Reversed sequence of keys would have been better
- **Brent's idea:**

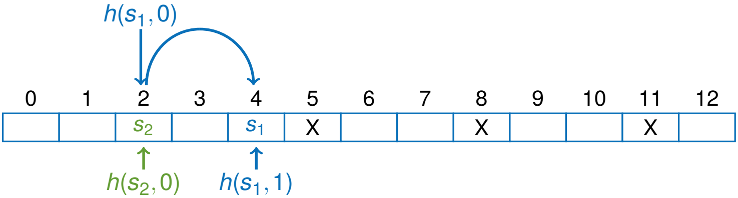


Figure: double hashing by Brent

- Reversed sequence of keys would have been better
- **Brent's idea:**
  - Test if location $h(s_1, 1)$ is free

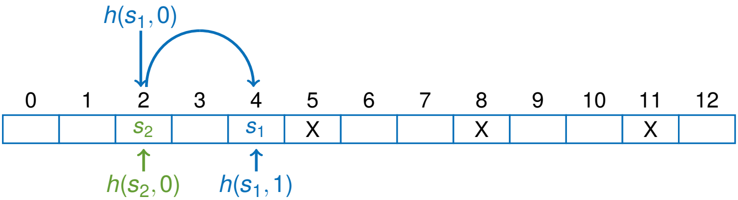


Figure: double hashing by Brent

- Reversed sequence of keys would have been better
- **Brent's 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 hash table sorted.
- Therefore, in case of an unsuccessful search of elements in combination with linear probing or double hashing, aborting is possible earlier 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

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

Motivation:

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

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:

UNI FREIBURG

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)

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 to 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 $j_1 < j_2$ $(0 < 7)$ key 12 stays at position p_1
- For 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

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:

UNI FREIBURG

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

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

Structure

Hashing

- Recapitulation
- Treatment of hash collisions
- Open Addressing
- Summary

Priority Queue

- Introduction

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
  - Raises the probability of collisions because probing order does not depend on the key

Open hashing: (static, number of elements fixed)

Open hashing: (static, number of elements fixed)

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

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:

UNI FREIBURG

Hashing:

- Efficiency of dictionary operations:

  `Insert`: $O(1) \ldots O(n)$

  `Search`: $O(1) \ldots O(n)$

  `Remove`: $O(1) \ldots O(n)$

UNI FREIBURG

Hashing:

- Efficiency of dictionary operations:

  `Insert`: $O(1) \ldots O(n)$

  `Search`: $O(1) \ldots O(n)$

  `Remove`: $O(1) \ldots O(n)$

- Direct access oto all elements in a hash table

Hashing:

- Efficiency of dictionary operations:
  `Insert`: $O(1) \ldots O(n)$
  `Search`: $O(1) \ldots O(n)$
  `Remove`: $O(1) \ldots O(n)$
- Direct access oto all elements in a hash table
- Using a hash function to find the position (hash value) in the hash table

UNI FREIBURG

Hashing:

- Efficiency of dictionary operations:
  `Insert:` $O(1) \ldots O(n)$
  `Search:` $O(1) \ldots O(n)$
  `Remove:` $O(1) \ldots O(n)$
- Direct access oto 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 all influence the efficiency of the data structure

Structure

Hashing

Recapitulation

Treatment of hash collisions

Open Addressing

Summary

Priority Queue

Introduction

Definition:

Definition:

- A priority queue saves a set of elements

Definition:

- A priority queue saves a set of elements
- Each element contains a key and a value like a map

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

Definition:

Definition:

- The priority queue supports the following operations:

Definition:

- The priority queue supports the following operations:

`insert(key, value):` inserts a new element into the queue

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

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:

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

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:

Special features:

- Multiple elements with the same key

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`

UNI FREIBURG

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

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 an element in the queue
  - That is 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()
```

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	...

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_2 :

4	5	6	7	...

L_3 :

1	10	11	24	...

$\Rightarrow R$:

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

Figure: 3-way merge

Example 1:

Example 1:

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

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 $\Theta(k)$

UNI FREIBURG

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 $\Theta(\log k)$

UNI FREIBURG

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:

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 (following lecture)

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 (following lecture)
- Among other applications it can be used for sorting

Idea:

Idea:

- Save elements as tuples in a binary heap

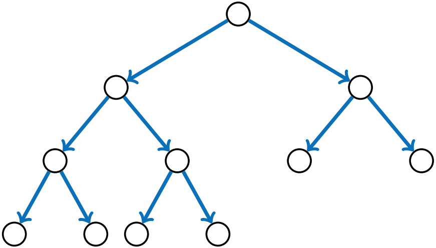


Figure: heap with 11 nodes

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

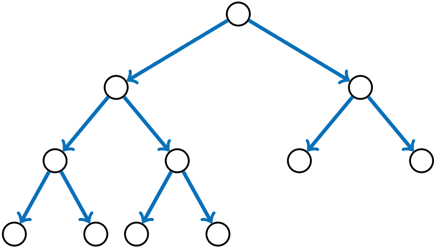
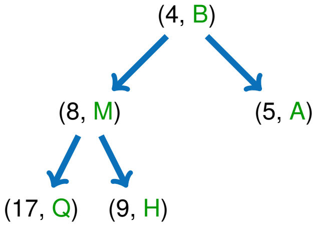
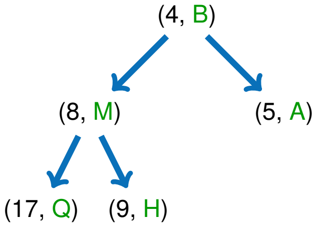


Figure: heap with 11 nodes



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

Figure: min heap stored in array

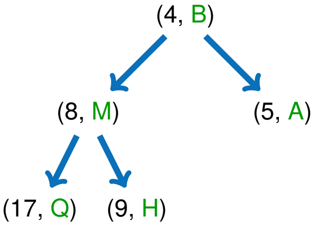


0	1	2	3	4
4, B	8, M	5, A	17, Q	9, H

Figure: min heap stored in array

Storing a binary heap:

Priority Queue
Implementation



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

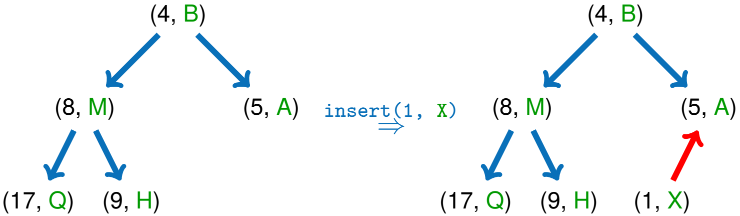
Figure: min heap stored in array

Storing a binary heap:

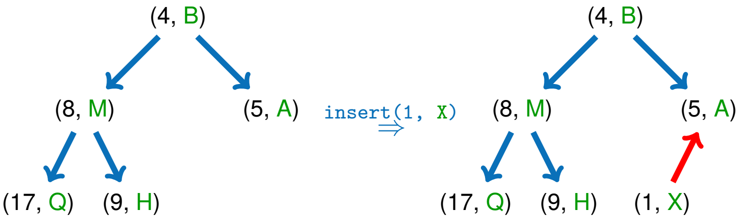
- Number nodes from top to bottom and left to right starting with 0 and store entries in array
- Children of node i are the nodes $2i+1$ and $2i+2$
- Parent node of node i is $\mathrm{floor}((i-1)/2)$

UNI FREIBURG

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

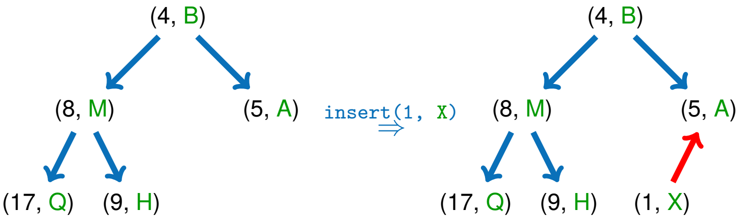


Inserting an element: `insert(key, item)`



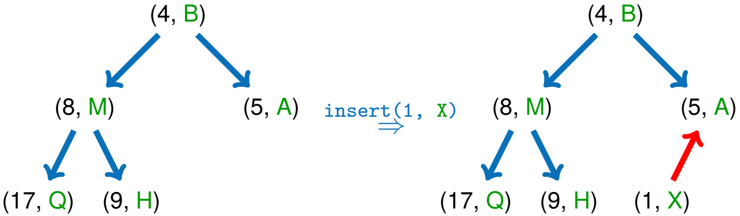
- Append the element at the end of the array

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



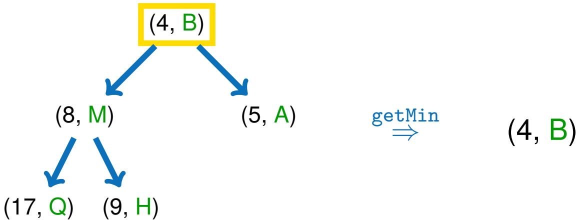
- Append the element at the end of the array
- The heap condition may be violated, but only at the last index

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

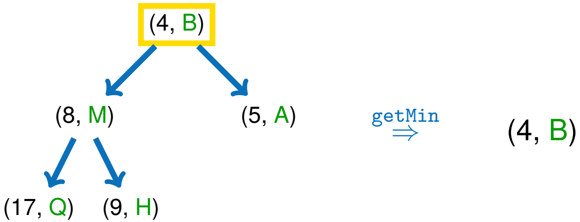


- Append the element at the end of the array
- The heap condition may be violated, but only at the last index
- Repair heap condition $\Rightarrow$ We will see later how to do this

Returning the minimum: `getMin()`

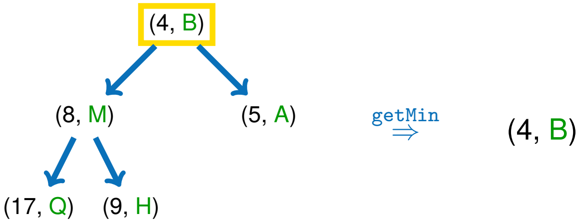


Returning the minimum: `getMin()`



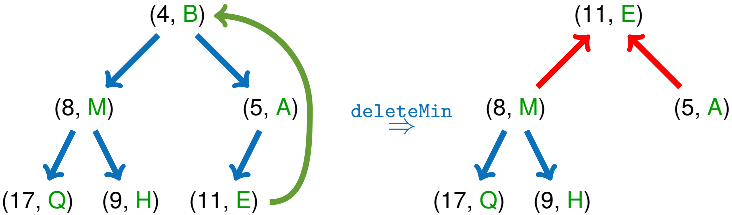
- Else return the first element

Returning the minimum: `getMin()`

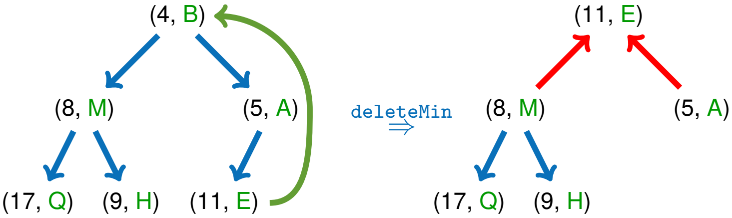


- Else return the first element
- If the heap is empty return None

Removing the minimum: `deleteMin()`

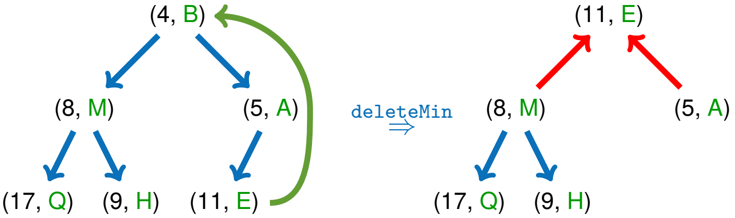


Removing the minimum: `deleteMin()`



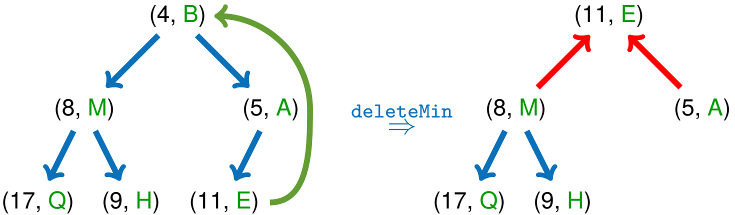
- Deleting the element with the lowest key

Removing the minimum: `deleteMin()`



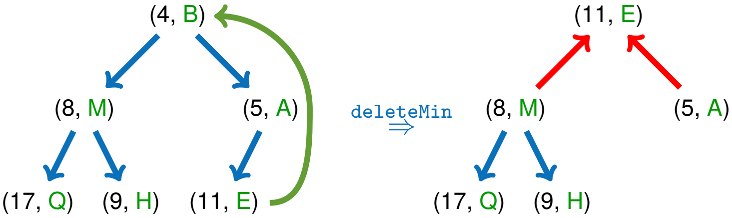
- Deleting the element with the lowest key
- Swap the last element with the first element and shrink the heap by one

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

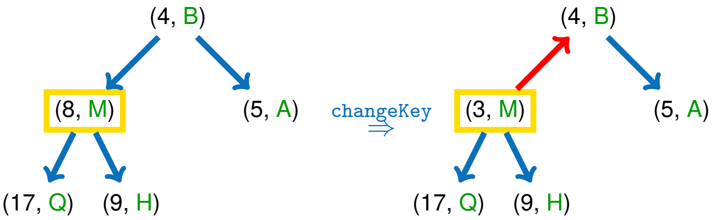
Removing the minimum: `deleteMin()`



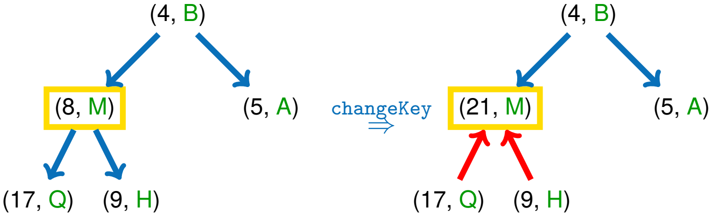
- Deleting the element with the lowest key
- Swap the last element with the first element and shrink the heap by one
- The heap condition may be violated, but only at the first index
- Repair heap condition

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

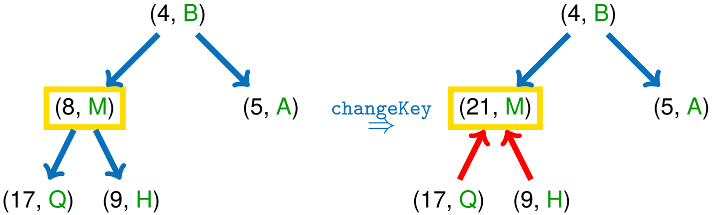
- The element (queue item) is given as argument
- Replace the key of the element
- The heap condition may be violated, but only at the element index and only in one direction (up / down)
- Repair heap condition



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

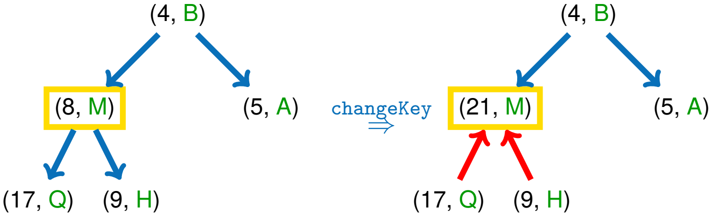


Changing the key (priority): `changeKey(item, key)`



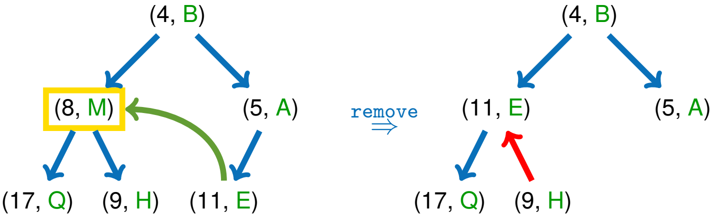
- The heap condition may be violated, but only at the element index and only in one direction (up / down)

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

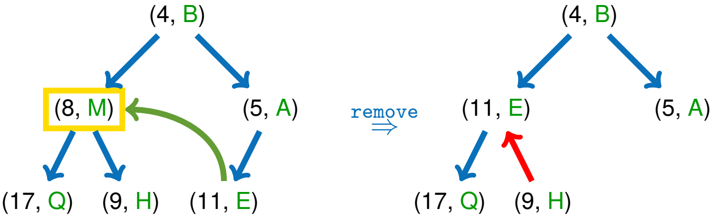
Removing an element: `remove(item)`



Priority Queue

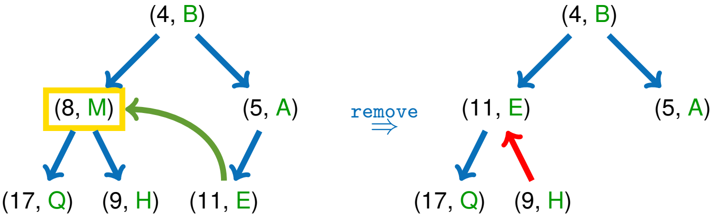
Implementation

Removing an element: `remove(item)`



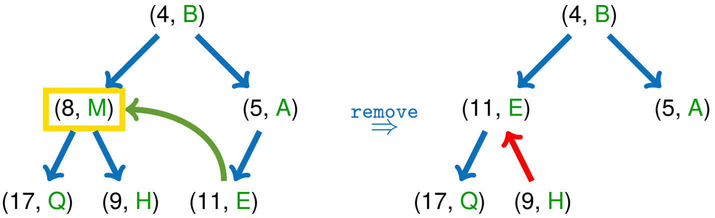
- The element (queue item) is given as argument

Removing an element: `remove(item)`



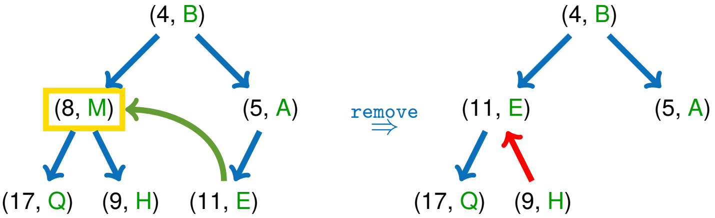
- The element (queue item) is given as argument
- Replace the element with the last element and shrink the heap by one

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)

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:

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*

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:

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

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

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:`

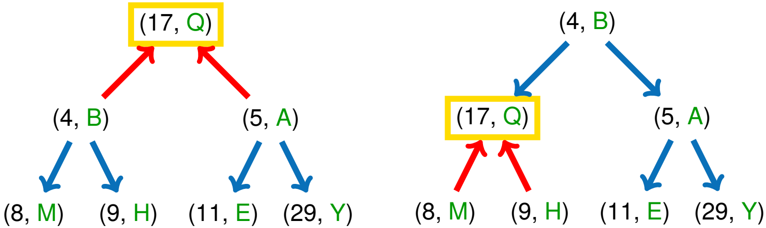


Figure: repairing the heap downwards

`repairHeapDown:`

- Sift the element until the heap condition is valid

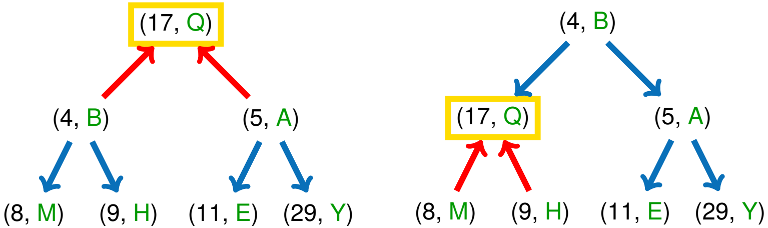


Figure: repairing the heap downwards

`repairHeapDown:`

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

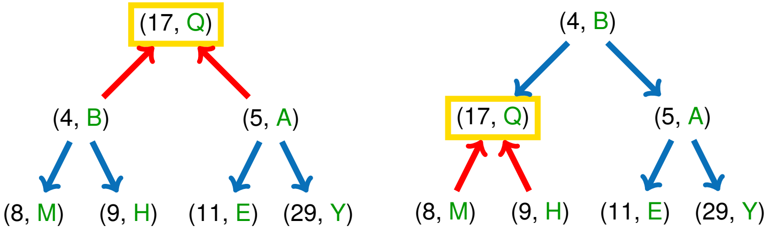


Figure: repairing the heap downwards

`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

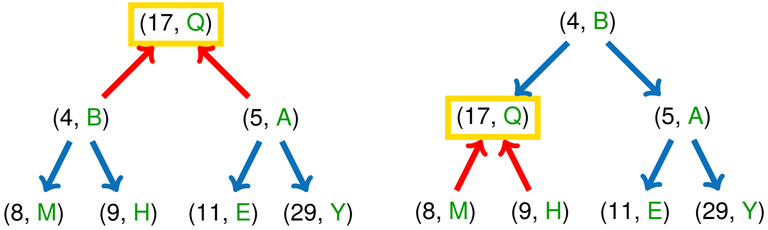


Figure: repairing the heap downwards

`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

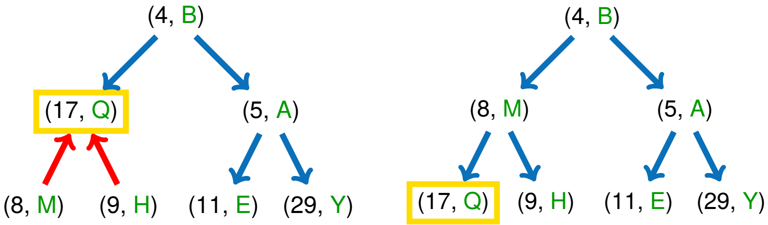


Figure: repairing the heap downwards

`repairHeapUp:`

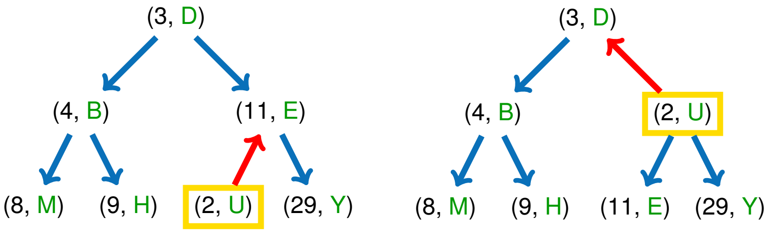


Figure: repairing the heap upwards

`repairHeapUp:`

- Change node with parent

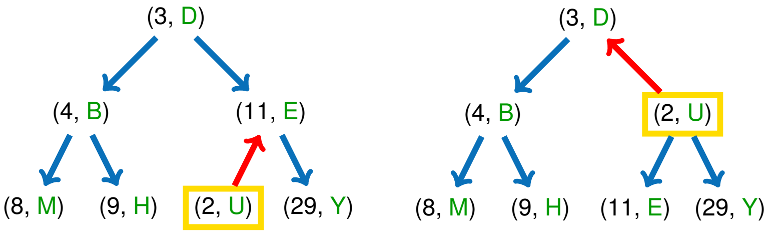


Figure: repairing the heap upwards

`repairHeapUp:`

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

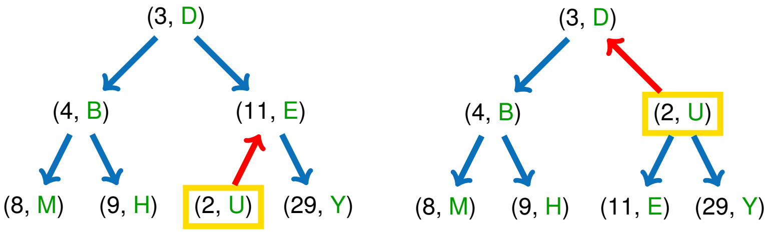


Figure: repairing the heap upwards

`repairHeapUp:`

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

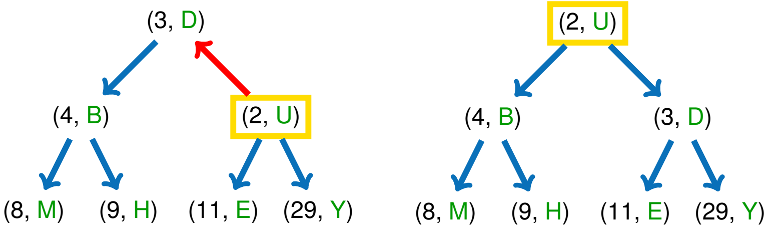


Figure: repairing the heap upwards

Index of a priority queue item:

Index of a priority queue item:

- Attention: for `changeKey` and `remove` the item has to "know" where it is located in the heap

UNI FREIBURG

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:

Summary lecture 1:

- A full binary tree with n elements has a depth of $O(\log n)$

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

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: $O(\log n)$

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

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: $O(\log n)$

UNI FREIBURG

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: $O(1)$

UNI FREIBURG

Improvements (Fibonacci heaps):

Improvements (Fibonacci heaps):

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

Improvements (Fibonacci heaps):

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

Improvements (Fibonacci heaps):

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

Practical experience:

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

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
- The difference is negligible if the number of elements is relatively small

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
- The difference is negligible if the number of elements is relatively small
- Example:
  - For $n = 2^{10} \approx 1,000$, the depth $\log_2 n$ is only 10
  - For $n = 2^{20} \approx 1,000,000$, the depth $\log_2 n$ is only 20

Further Literature

- **Course literature**

[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