Amortized Analysis

Definition 1
A data structure with operations \(op_1(), \ldots, op_k()\) has amortized running times \(t_1, \ldots, t_k\) for these operations if the following holds.

Suppose you are given a sequence of operations (starting with an empty data-structure) that operate on at most \(n\) elements, and let \(k_i\) denote the number of occurrences of \(op_i()\) within this sequence. Then the actual running time must be at most
\[
\sum_{i=1}^{k} k_i \cdot t_i(n).
\]

Potential Method

Introduce a potential for the data structure.
- \(\Phi(D_i)\) is the potential after the \(i\)-th operation.
- Amortized cost of the \(i\)-th operation is
  \[
  \hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).
  \]
- Show that \(\Phi(D_i) \geq \Phi(D_0)\).

Then
\[
\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c_i + \Phi(D_k) - \Phi(D_0) = \sum_{i=1}^{k} \hat{c}_i.
\]

This means the amortized costs can be used to derive a bound on the total cost.

Example: Stack

Stack
- \(S.\ push()\)
- \(S.\ pop()\)
- \(S.\ multipop(k)\): removes \(k\) items from the stack. If the stack currently contains less than \(k\) items it empties the stack.
- The user has to ensure that pop and multipop do not generate an underflow.

Actual cost:
- \(S.\ push()\): cost 1.
- \(S.\ pop()\): cost 1.
- \(S.\ multipop(k)\): cost \(\min\{\text{size}, k\} = k\).

Example: Stack

Use potential function \(\Phi(S) = \text{number of elements on the stack}\).

Amortized cost:
- \(S.\ push()\): cost
  \[
  \hat{C}_{\text{push}} = C_{\text{push}} + \Delta \Phi = 1 + 1 \leq 2.
  \]
- \(S.\ pop()\): cost
  \[
  \hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta \Phi = 1 - 1 \leq 0.
  \]
- \(S.\ multipop(k)\): cost
  \[
  \hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta \Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \leq 0.
  \]

Note that the analysis becomes wrong if pop() or multipop() are called on an empty stack.
Example: Binary Counter

Incrementing a binary counter:
Consider a computational model where each bit-operation costs one time-unit.

Incrementing an \( n \)-bit binary counter may require to examine \( n \)-bits, and maybe change them.

Actual cost:
- Changing bit from 0 to 1: cost 1.
- Changing bit from 1 to 0: cost 1.
- Increment: cost is \( k + 1 \), where \( k \) is the number of consecutive ones in the least significant bit-positions (e.g, 001101 has \( k = 1 \)).

Amortized cost:
- Changing bit from 0 to 1:
  \[
  \hat{C}_{0 \rightarrow 1} = C_{0 \rightarrow 1} + \Delta \Phi = 1 + 1 \leq 2.
  \]
- Changing bit from 1 to 0:
  \[
  \hat{C}_{1 \rightarrow 0} = C_{1 \rightarrow 0} + \Delta \Phi = 1 - 1 \leq 0.
  \]
- Increment: Let \( k \) denotes the number of consecutive ones in the least significant bit-positions. An increment involves \( k \) (1 \( \rightarrow \) 0)-operations, and one (0 \( \rightarrow \) 1)-operation.
  
  Hence, the amortized cost is \( k\hat{C}_{1 \rightarrow 0} + \hat{C}_{0 \rightarrow 1} \leq 2 \).

8.3 Fibonacci Heaps

Collection of trees that fulfill the heap property.

Structure is much more relaxed than binomial heaps.

Additional implementation details:
- Every node \( x \) stores its degree in a field \( x.\text{degree} \). Note that this can be updated in constant time when adding a child to \( x \).
- Every node stores a boolean value \( x.\text{marked} \) that specifies whether \( x \) is marked or not.
8.3 Fibonacci Heaps

The potential function:
- \( t(S) \) denotes the number of trees in the heap.
- \( m(S) \) denotes the number of marked nodes.
- We use the potential function \( \Phi(S) = t(S) + 2m(S) \).

The potential is \( \Phi(S) = 5 + 2 \cdot 3 = 11 \).

8.3 Fibonacci Heaps

We assume that one unit of potential can pay for a constant amount of work, where the constant is chosen “big enough” (to take care of the constants that occur).

To make this more explicit we use \( c \) to denote the amount of work that a unit of potential can pay for.

8.3 Fibonacci Heaps

\( S. \) minimum()

- Access through the min-pointer.
- Actual cost \( \Theta(1) \).
- No change in potential.
- Amortized cost \( \Theta(1) \).

8.3 Fibonacci Heaps

\( S. \) merge\((S')\)

- Merge the root lists.
- Adjust the min-pointer

Running time:
- Actual cost \( \Theta(1) \).
- No change in potential.
- Hence, amortized cost is \( \Theta(1) \).
8.3 Fibonacci Heaps

S. insert \( x \)
- Create a new tree containing \( x \).
- Insert \( x \) into the root-list.
- Update min-pointer, if necessary.

Running time:
- Actual cost \( \mathcal{O}(1) \).
- Change in potential is +1.
- Amortized cost is \( c + \mathcal{O}(1) = \mathcal{O}(1) \).

8.3 Fibonacci Heaps

S. delete-min \( x \)
- Delete minimum; add child-trees to heap; time: \( D(\text{min}) \cdot \mathcal{O}(1) \).
- Update min-pointer; time: \( (t + D(\text{min})) \cdot \mathcal{O}(1) \).

During the consolidation we traverse the root list. Whenever we discover two trees that have the same degree, we use an array that contains for every degree value \( d \) a pointer to a tree left of the current pointer whose root has degree \( d \) (if such a tree exist).
8.3 Fibonacci Heaps

Consolidate:

Consolidate:

Consolidate:

Consolidate:
8.3 Fibonacci Heaps

Consolidate:

current

min → 7

© Harald Racke

8.3 Fibonacci Heaps

Consolidate:

current

min → 7

© Harald Racke

8.3 Fibonacci Heaps

Consolidate:

current

min → 7

© Harald Racke

8.3 Fibonacci Heaps

Consolidate:

current

min → 7

© Harald Racke
8.3 Fibonacci Heaps

Actual cost for delete-min()
- At most $D_n + t$ elements in root-list before consolidate.
- Hence there exists $c_1$ s.t. actual cost is at most $c_1 \cdot (D_n + t)$.

Amortized cost for delete-min()
- $t' \leq D_n + 1$ as degrees are different after consolidating.
- Therefore $\Delta \Phi \leq D_n + 1 - t$;
- We can pay $c \cdot (t-D_n-1)$ from the potential decrease.
- The amortized cost is
  
  \[
  c_1 \cdot (D_n + t) - c \cdot (t-D_n-1) 
  \leq (c_1 + c)D_n + (c_1 - c)t + c \leq 2c(D_n + 1) \leq \Theta(D_n)
  \]
  
  for $c \geq c_1$.

Fibonacci Heaps: decrease-key(handle h, v)

Case 1: decrease-key does not violate heap-property
- Just decrease the key-value of element referenced by $h$.
  Nothing else to do.

Case 2: heap-property is violated, but parent is not marked
- Decrease key-value of element $x$ reference by $h$.
- If the heap-property is violated, cut the parent edge of $x$, and make $x$ into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of $x$ (unless it’s a root).
Fibonacci Heaps: decrease-key\((\text{handle } h, v)\)

Case 2: heap-property is violated, but parent is not marked
- Decrease key-value of element \(x\) reference by \(h\).
- If the heap-property is violated, cut the parent edge of \(x\), and make \(x\) into a root.
- Adjust min-pointers, if necessary.
- Mark the (previous) parent of \(x\) (unless it’s a root).

Case 3: heap-property is violated, and parent is marked
- Decrease key-value of element \(x\) reference by \(h\).
- Cut the parent edge of \(x\), and make \(x\) into a root.
- Adjust min-pointers, if necessary.
- Continue cutting the parent until you arrive at an unmarked node.

Marking a node can be viewed as a first step towards becoming a root. The first time \(x\) loses a child it is marked; the second time it loses a child it is made into a root.
Fibonacci Heaps: decrease-key(handle h, v)

Actual cost:
- Constant cost for decreasing the value.
- Constant cost for each of ℓ cuts.
- Hence, cost is at most \( c_2 \cdot (ℓ + 1) \), for some constant \( c_2 \).

Amortized cost:
- \( t' = t + ℓ \), as every cut creates one new root.
- \( m' \leq m - (ℓ - 1) + 1 = m - ℓ + 2 \), since all but the first cut unmarks a node; the last cut may mark a node.
- \( ∆Φ \leq ℓ + 2(-ℓ + 2) = 4 - ℓ \)
- Amortized cost is at most \( c_2(ℓ + 1) + c(4 - ℓ) \leq (c_2 - c)ℓ + 4c + c_2 = O(1) \), if \( c \geq c_2 \).

Delete node

\( H. \, delete(x) \):
- decrease value of \( x \) to \(-∞\).
- delete-min.

Amortized cost: \( O(D_n) \)
- \( O(1) \) for decrease-key.
- \( O(D_n) \) for delete-min.

8.3 Fibonacci Heaps

Lemma 2
Let \( x \) be a node with degree \( k \) and let \( y_1, \ldots, y_k \) denote the children of \( x \) in the order that they were linked to \( x \). Then

\[
\text{degree}(y_i) \geq \begin{cases} 
0 & \text{if } i = 1 \\
\text{degree}(y_i) - 2 & \text{if } i > 1
\end{cases}
\]

The marking process is very important for the proof of this lemma. It ensures that a node can have lost at most one child since the last time it became a non-root node. When losing a first child the node gets marked; when losing the second child it is cut from the parent and made into a root.

Proof
- When \( y_1 \) was linked to \( x \), at least \( y_1, \ldots, y_{i-1} \) were already linked to \( x \).
- Hence, at this time degree(x) ≥ i − 1, and therefore also degree(y_i) ≥ i − 1 as the algorithm links nodes of equal degree only.
- Since, then \( y_1 \) has lost at most one child.
- Therefore, degree(y_i) ≥ i − 2.
8.3 Fibonacci Heaps

Let $s_k$ be the minimum possible size of a sub-tree rooted at a node of degree $k$ that can occur in a Fibonacci heap.

- $s_k$ monotonically increases with $k$
- $s_0 = 1$ and $s_1 = 2$.

Let $x$ be a degree $k$ node of size $s_k$ and let $y_1, \ldots, y_k$ be its children.

\[
s_k = 2 + \sum_{i=2}^{k} \text{size}(y_i) \\
\geq 2 + \sum_{i=2}^{k} s_{i-2} \\
= 2 + \sum_{i=0}^{k-2} s_i
\]

\[
F_k = \begin{cases} 
1 & \text{if } k = 0 \\
2 & \text{if } k = 1 \\
F_{k-1} + F_{k-2} & \text{if } k \geq 2 
\end{cases}
\]

Facts:

1. $F_k \geq \phi^k$.
2. For $k \geq 2$: $F_k = 2 + \sum_{i=0}^{k-2} F_i$.

The above facts can be easily proved by induction. From this it follows that $s_k \geq F_k \geq \phi^k$, which gives that the maximum degree in a Fibonacci heap is logarithmic.