Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

- after access, an element is moved to the root; splay(x)
- repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

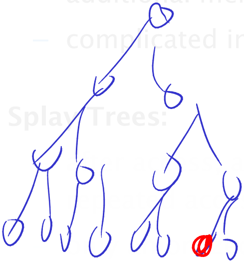
Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

$n \qquad m$

$\underbrace{}$

$O(n \log n) + O(m)$



Splay Trees:

- after access, an element is moved to the root; splay(x)
- repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

- after access, an element is moved to the root; splay(x)
- repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

- after access, an element is moved to the root; splay(x)
- repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

+ after access, an element is moved to the root; splay(x)
  repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

+ after access, an element is moved to the root; splay(x)
  repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

\+ after access, an element is moved to the root; splay(x)
  repeated accesses are faster

- only amortized guarantee
- read-operations change the tree

Splay Trees

Disadvantage of balanced search trees:

- worst case; no advantage for easy inputs
- additional memory required
- complicated implementation

Splay Trees:

+ after access, an element is moved to the root; splay(x)
  repeated accesses are faster
- only amortized guarantee
- read-operations change the tree

Splay Trees

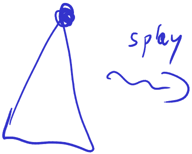
find(x)

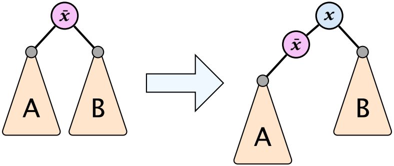
- search for x according to a search tree
- let $\bar{x}$ be last element on search-path
- splay($\bar{x}$)

Splay Trees

insert(x)

- search for x; $\bar{x}$ is last visited element during search (successer or predecessor of x)
- splay($\bar{x}$) moves $\bar{x}$ to the root
- insert x as new root

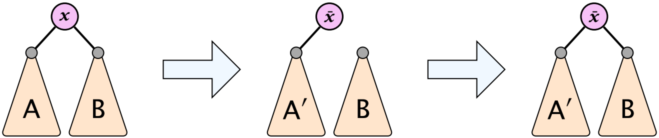




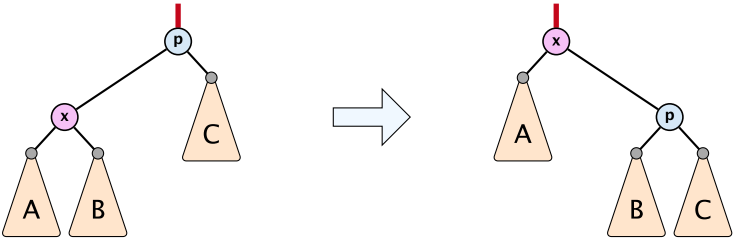
Splay Trees

delete(x)

- search for x; splay(x); remove x
- search largest element $\bar{x}$ in A
- splay($\bar{x}$) (on subtree A)
- connect root of B as right child of $\bar{x}$



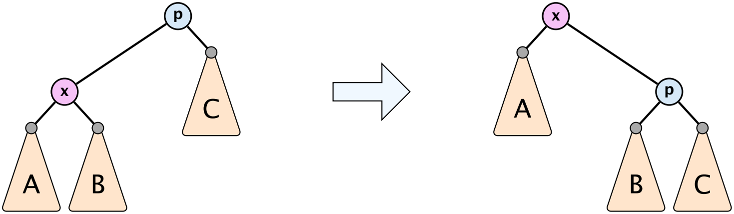
Move to Root



How to bring element to root?

- one (bad) option: moveToRoot(x)
- iteratively do rotation around parent of x until x is root
- if x is left child do right rotation otw. left rotation

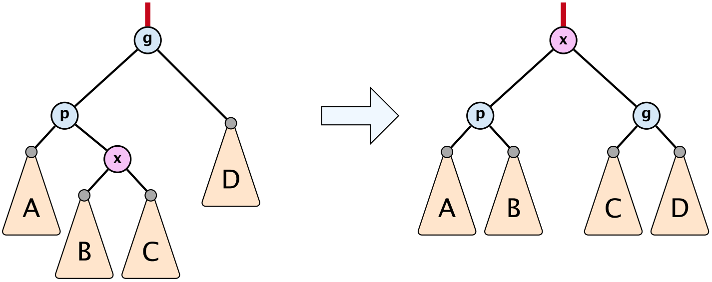
Splay: Zig Case



better option splay(x):

- zig case: if x is child of root do left rotation or right rotation around parent

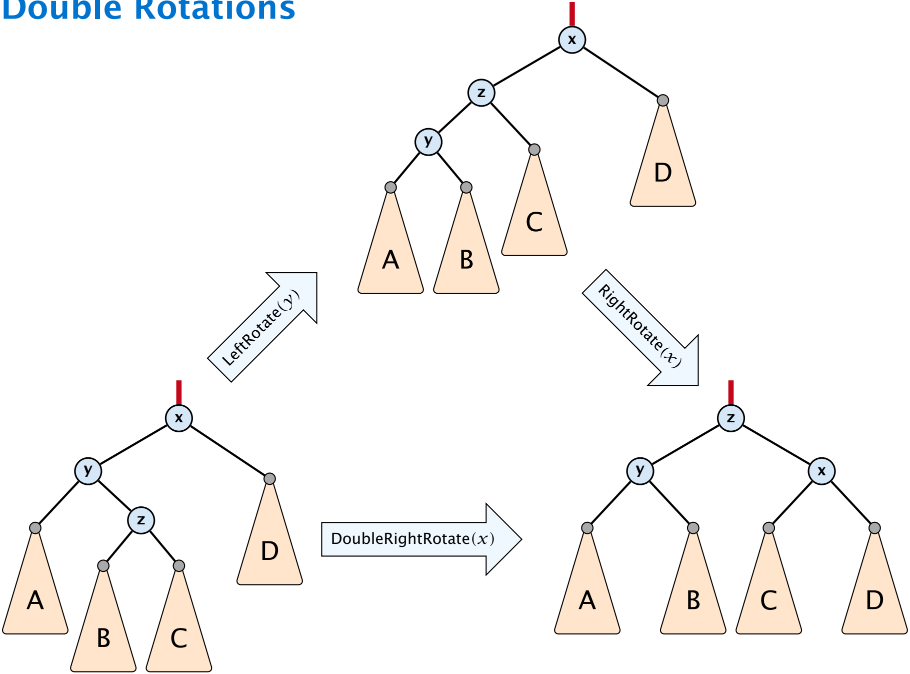
Splay: Zigzag Case



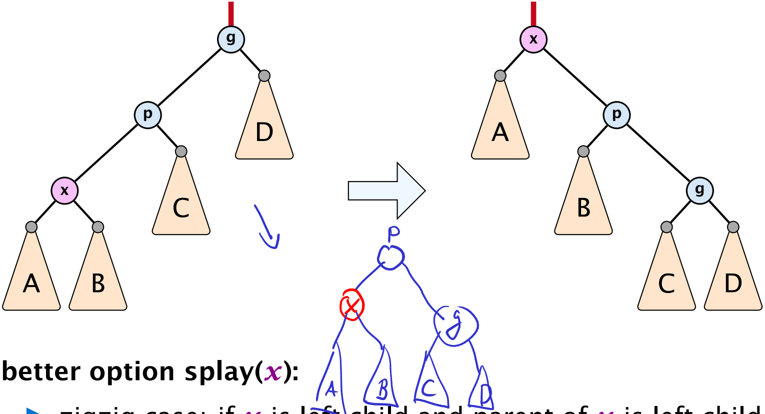
better option splay(x):

- zigzag case: if x is right child and parent of x is left child (or x left child parent of x right child)
- do double right rotation around grand-parent (resp. double left rotation)

Double Rotations



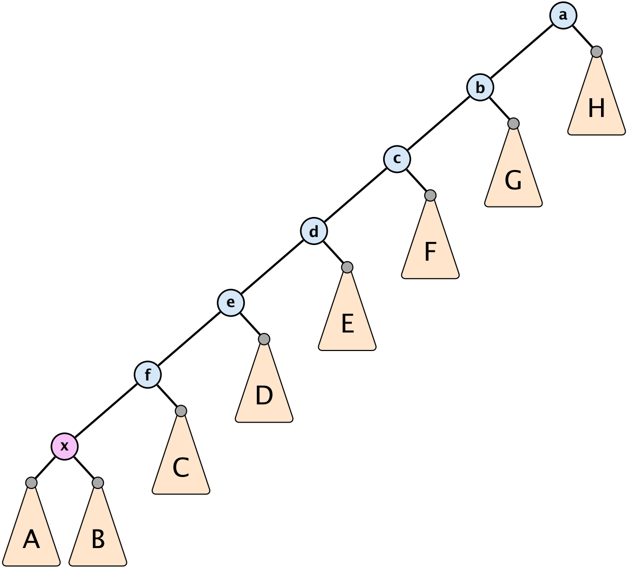
Splay: Zigzig Case



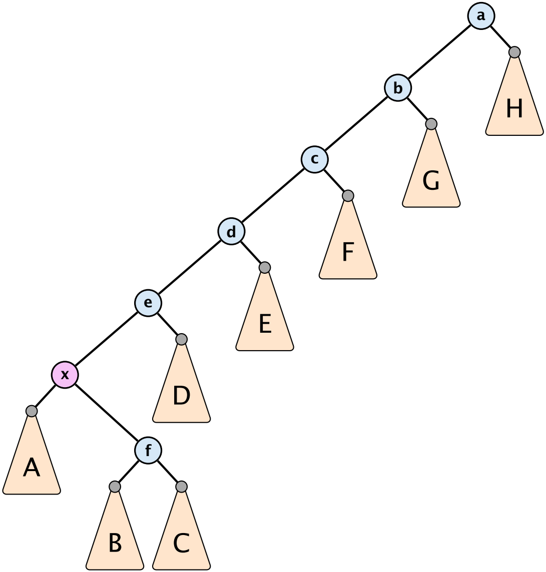
better option splay(x):

- zigzig case: if x is left child and parent of x is left child (or x right child, parent of x right child)
- do right roation around grand-parent followed by right rotation around parent (resp. left rotations)

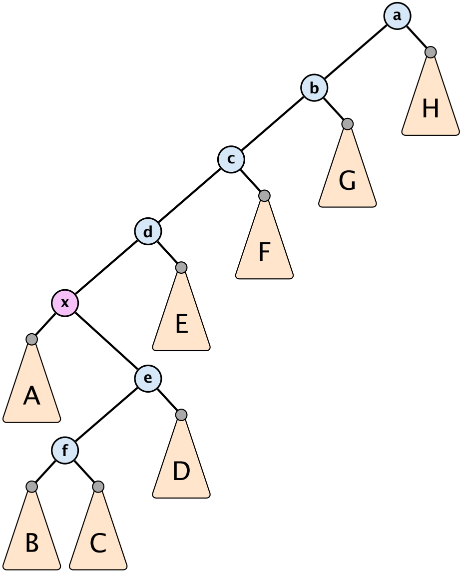
Splay vs. Move to Root



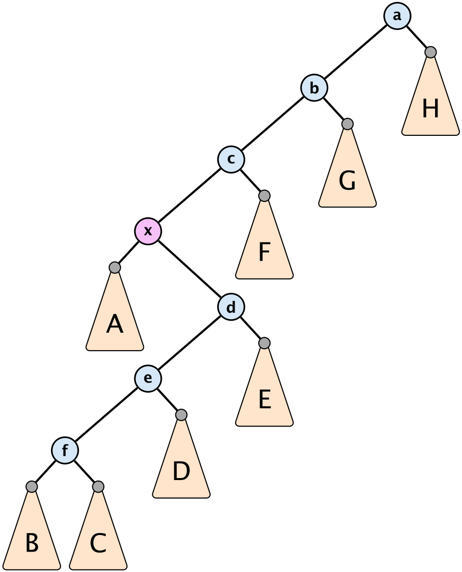
Splay vs. Move to Root



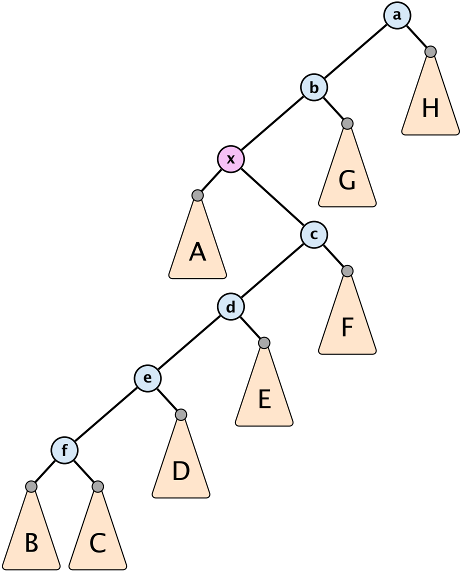
Splay vs. Move to Root



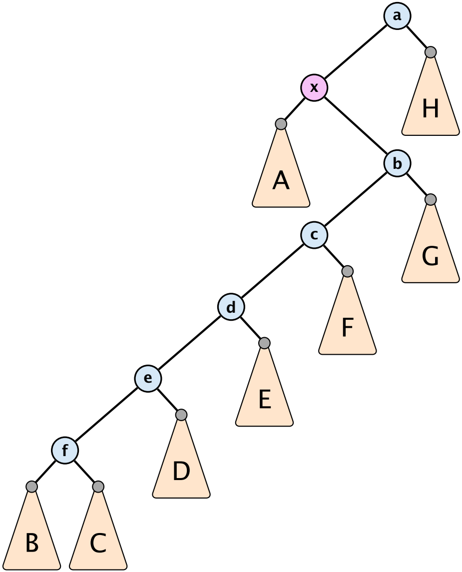
Splay vs. Move to Root



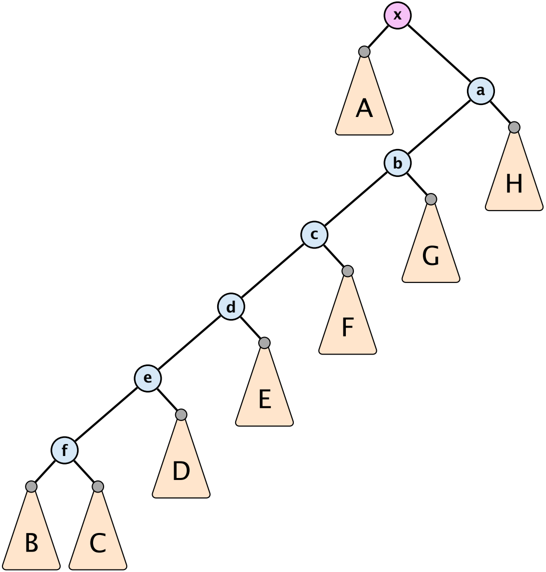
Splay vs. Move to Root



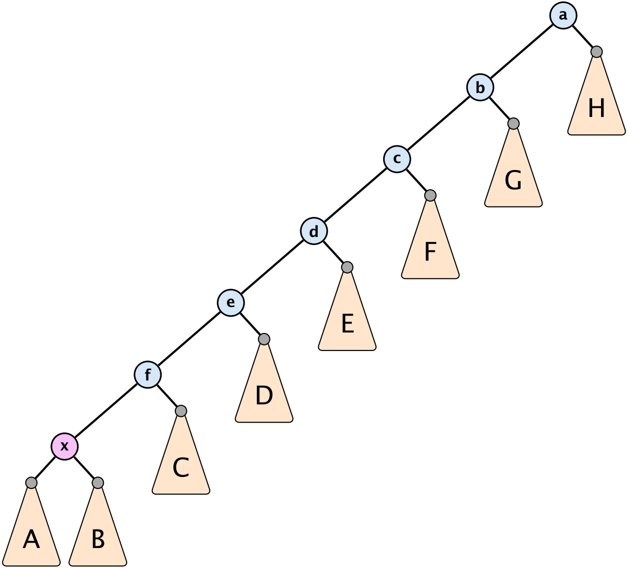
Splay vs. Move to Root



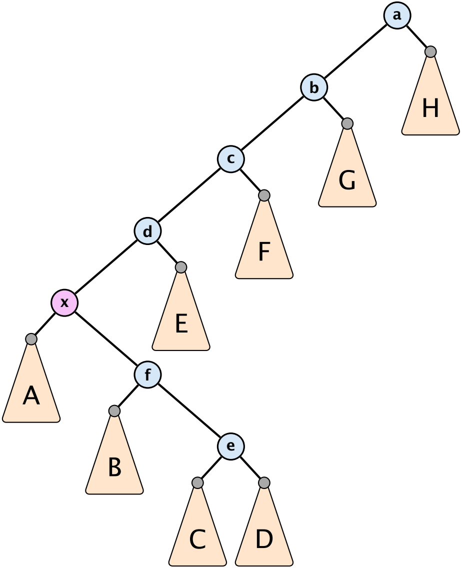
Splay vs. Move to Root



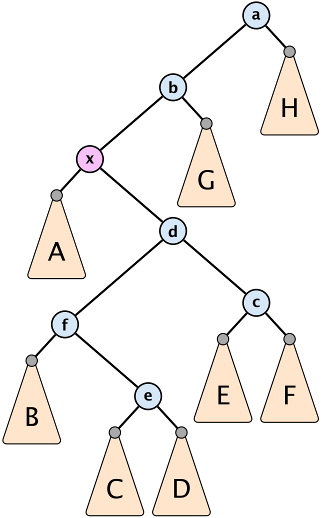
Splay vs. Move to Root



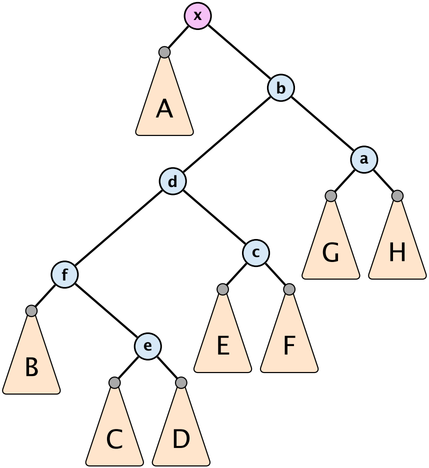
Splay vs. Move to Root



Splay vs. Move to Root



Splay vs. Move to Root



Static Optimality

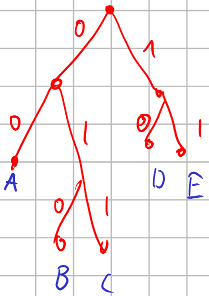
Suppose we have a sequence of m find-operations. find(x) appears h_x times in this sequence.

The cost of a static search tree T is:

$$\text{cost}(T) = m + \sum_x h_x \,\text{depth}_T(x)$$

The total cost for processing the sequence on a splay-tree is $\mathcal{O}(\text{cost}(T_{\text{min}}))$, where T_{min} is an optimal static search tree.

A B D A A B D D E C C



$$\sum_{x} h_x \cdot \mathrm{depth}_T(x)$$

011

010

Dynamic Optimality

Let S be a sequence with m find-operations.

Let A be a data-structure based on a search tree:

- the cost for accessing element x is $1 + \text{depth}(x)$;
- after accessing x the tree may be re-arranged through rotations;

Conjecture:

A splay tree that only contains elements from S has cost $\mathcal{O}(\text{cost}(A, S))$, for processing S.

Lemma 16

Splay Trees have an amortized running time of $\mathcal{O}(\log n)$ *for all operations.*

Amortized Analysis

Definition 17

A data structure with operations $\mathrm{op}_1(), \ldots, \mathrm{op}_k()$ 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 occurences of $\mathrm{op}_i()$ within this sequence. Then the actual running time must be at most $\sum_i k_i \cdot t_i(n)$.

Potential Method

Introduce a potential for the data structure.

Potential Method

Introduce a potential for the data structure.

- $\Phi(D_i)$ is the potential after the i-th operation.

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \ .$$

real cost

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 $\Phi(D_i) \geq \underbrace{\Phi(D_0)}_{=0}$.

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

$$\sum_{i=1}^{k} c_i$$

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \enspace .$$

- Show that $\Phi(D_i) \geq \Phi(D_0)$.

Then

$$\sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} c_i + \overbrace{\Phi(D_k) - \Phi(D_0)}^{\geq 0}$$

$$= \sum_i c_i + \sum_{i>0} \left(\Phi(D_i) - \Phi(D_{i-1}) \right)$$

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

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \enspace .$$

- 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

- $S.\mathbf{push}()$
- $S.\mathbf{pop}()$
- $S.\mathbf{multipop}(k)$: 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:

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

Example: Stack

Stack

- $S.\mathbf{push}()$
- $S.\mathbf{pop}()$
- $S.\mathbf{multipop}(k)$: 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:

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

Example: Stack

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

Amortized cost:

- $S.$ push(): cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \le 2 .$$

- $S.$ pop(): cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \le 0 .$$

- $S.$ multipop(k): cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min(\text{size}, k) - \min(\text{size}, k) \le 0 .$$

Example: Stack

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

Amortized cost:

- $\mathbf{S.\,push()}$: cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- $\mathbf{S.\,pop()}$: cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- $\mathbf{S.\,multipop(\mathit{k})}$: cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \le 0 \ .$$

Example: Stack

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

Amortized cost:

- $\boldsymbol{S.\,\mathbf{push}()}$: cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \le 2 \enspace .$$

- $\boldsymbol{S.\,\mathbf{pop}()}$: cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \le 0 \enspace .$$

- $\boldsymbol{S.\,\mathbf{multipop}(k)}$: cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \min\{\text{size}, k\} \le 0 \enspace .$$

Example: Stack

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

Amortized cost:

- $S.\textbf{push}()$: cost

$$\hat{C}_{\text{push}} = C_{\text{push}} + \Delta\Phi = 1 + 1 \le 2 .$$

- $S.\textbf{pop}()$: cost

$$\hat{C}_{\text{pop}} = C_{\text{pop}} + \Delta\Phi = 1 - 1 \le 0 .$$

- $S.\textbf{multipop}(k)$: cost

$$\hat{C}_{\text{mp}} = C_{\text{mp}} + \Delta\Phi = \min\{\text{size}, k\} - \overset{k}{\overset{\shortparallel}{\min\{\text{size}, k\}}} \le 0 .$$

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 $k = 1$).

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 $k=1$).

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 $k = 1$).

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from $0 \to 1$:

$$\hat{C}_{0\to1} = C_{0\to1} + \Delta\Phi = 1 + 1 \leq 2 .$$

- Changing bit from $1 \to 0$:

$$\hat{C}_{1\to0} = C_{1\to0} + \Delta\Phi = 1 - 1 \leq 0 .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to0} + \hat{C}_{0\to1} \leq 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

1 11011

- Changing bit from 1 to 0:

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \le 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$$\hat{C}_{0\to1} = C_{0\to1} + \Delta\Phi = 1 + 1 \le 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to0} = C_{1\to0} + \Delta\Phi = 1 - 1 \le 0 \ .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to0} + \hat{C}_{0\to1} \le 2$.

Example: Binary Counter

Choose potential function $\Phi(x) = k$, where k denotes the number of ones in the binary representation of x.

Amortized cost:

- Changing bit from 0 to 1:

$11\;0\;\underbrace{1111}_{0000}$

$$\hat{C}_{0\to 1} = C_{0\to 1} + \Delta\Phi = 1 + 1 \leq 2 \ .$$

- Changing bit from 1 to 0:

$$\hat{C}_{1\to 0} = C_{1\to 0} + \Delta\Phi = 1 - 1 \leq 0 \ .$$

- Increment: Let k denotes the number of consecutive ones in the least significant bit-positions. An increment involves k $(1 \to 0)$-operations, and one $(0 \to 1)$-operation.

  Hence, the amortized cost is $k\hat{C}_{1\to 0} + \hat{C}_{0\to 1} \leq 2$.