amortize: to put money aside at intervals for gradual payment (of a debt, etc.)

(Webster's)

idea: to average over time.

Motivation for Amortization

In many uses of data structures, a <u>sequence</u> of operations (rather than just one) is performed.

We are interested in the <u>total</u> time of the sequence.

Motivation:

In many uses of data structures, a sequence of operations (rather than just one) is performed.

We are interested in the total time of the sequence.

[Next page]

Worst-case time per operation may be unduly pessimistic, because of correlated effects of operations on data structure.

Average-case time may be inaccurate, since the probabilistic assumptions needed to carry out the analysis may be false.

→ Amortized time (time per operation averaged over a worst-case sequence) is both realistic and robust.

Worst-case time per operation may be unduly pessimistic, because of correlated effects of operations on data structure.

Average-case time may be inaccurate since the probabilistic assumptions needed to carry out the analysis may be false.

→ Amortized time (time per operation averaged over a worst-case sequence) is both realistic and robust.

An example: stack manipulation

Unit-time primitives:

push (an item onto the stack)

pop (an item off of the stack)

Operation:

carry out zero or more pops, followed by a push.

Beginning with an empty stack, carry out a sequence of n operations

Question: How many pushes and pops total?

Answer: 2n

Each operation causes one push (immediately) and possibly one pop (later)

(After i operations, there have been $2i-k$ pushes and pops, where k is the stack size)

Applications:

Linear-time string-matching
(Knuth, Morris, Pratt)

Planarity testing
(Hopcroft, Tarjan)

etc.

How can we formalize this phenomenon
and exploit it in
the design and analysis of algorithms

A Banker's View of Amortization

Credits: One will pay for a unit-time computation.

Credits can sit in the data structure, representing time saved.

Debits: Each represents an excess unit of time spent.

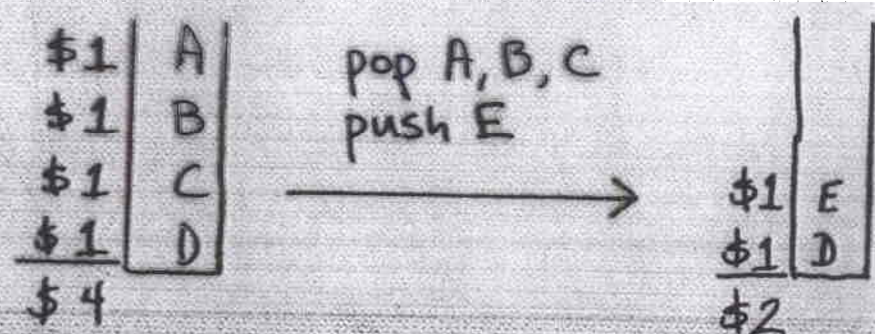
Can also sit in data structure, must be accounted for at end of computation.

A banker's analysis of the stack

Each operation gets two credits.

Number of saved credits equals stack size.

⇒ Each pop paid for by a saved credit.



\$4 + \$2 − \$2 = \$4

pays for three pops, one push

A Physicist's View of Amortization

Each configuration of data structure has a <u>potential</u> Φ.

a_i (amortized time of operation i) $\equiv$

t_i (actual time of operation i)

$+ \Phi_i$ (potential after operation)

$- \Phi_{i-1}$ (potential before operation)

$$a_i \equiv t_i + \Phi_i - \Phi_{i-1}$$

$$\sum_{i=1}^{m} t_i = \sum_{i=1}^{m} (a_i + \Phi_{i-1} - \Phi_i)$$

$$= \sum_{i=1}^{m} a_i + \Phi_0 - \Phi_m$$

$$\le \sum_{i=1}^{m} a_i \text{ if } \Phi_0 = 0 \text{ and } \Phi_m \ge 0.$$

Definition of potential is arbitrary,

but only a good choice gives

useful results.

A Physicist's analysis of the stack

Potential of stack equals stack size

⇒ Amortized time of an operation with k pops

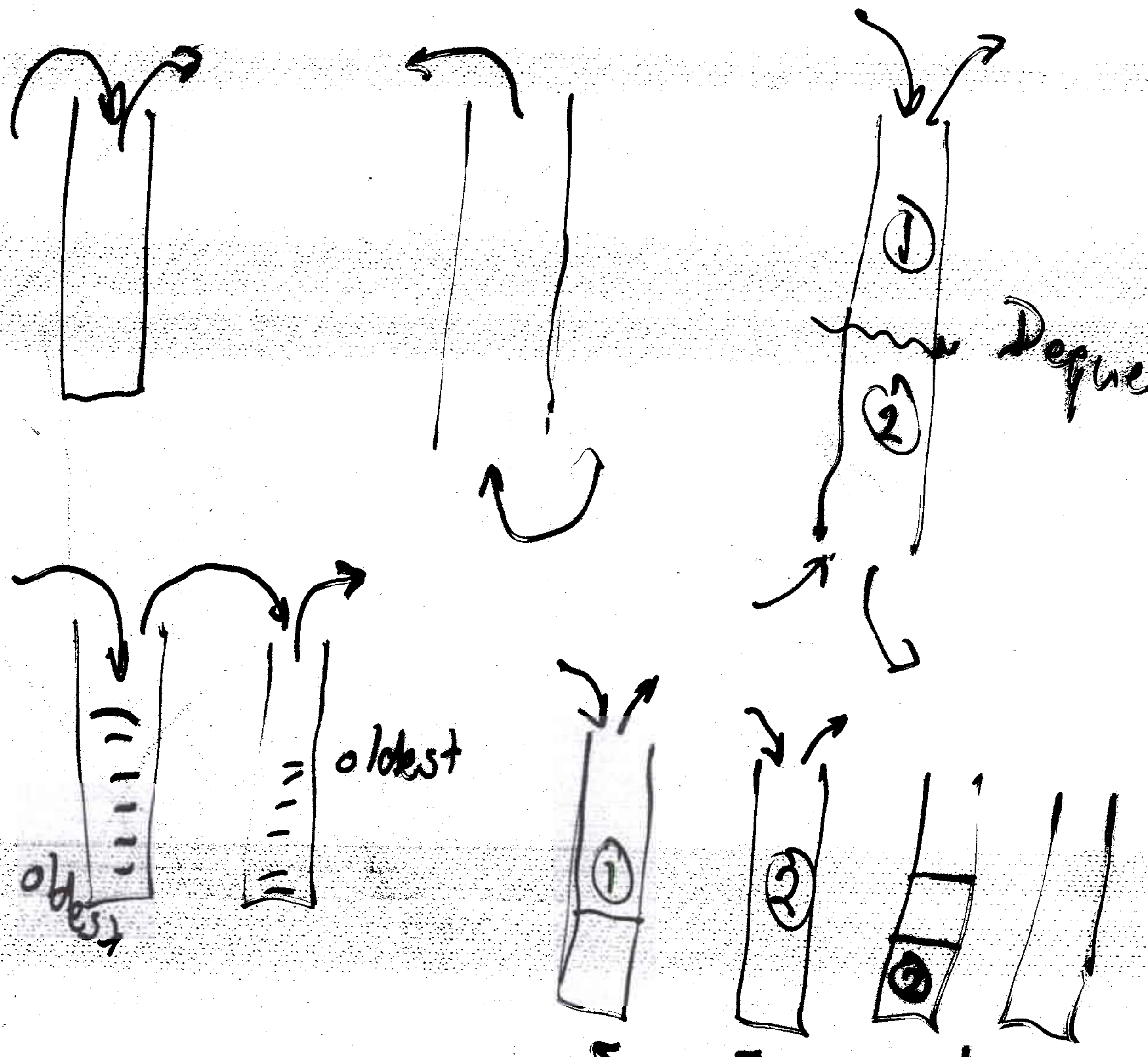
$$
\begin{aligned}
&\bullet \quad k+1 && \text{(actual time)} \\
&+ \quad s-k+1 && \text{(new potential)} \\
&- \quad s && \text{(old potential)} \\
&= \quad 2.
\end{aligned}
$$

Uses of Amortization

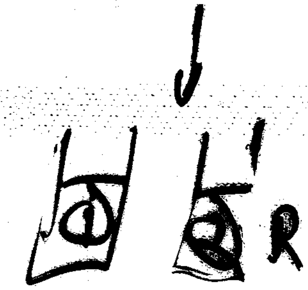
As an analytical tool: obtain new bounds for known algorithms: self-organizing sequential search, disjoint set union, etc.

As a design tool: obtain new "self-adjusting" data structures that are simple and have good amortized performance.

Stacks vs. Queues



$\Phi = |\text{diff in stack ht}|$



Competitiveness

On-line vs. off-line algorithms:

How much does knowing the future help?

The skier's dilemma:

Renting skis costs \$x per ski trip

Buying skis costs \$y

When to buy?

Goal: minimize the cost ratio as compared to the best policy when the number of trips is known.

Solution: Buy when total rent equals cost of buying

performance ratio (competitive factor) = 2

An on-line algorithm is k-competitive if its performance is within a factor of k of that of the optimum off-line algorithm on any sequence of operations.

Self-organizing linear lists

Data structure: n items stored in a linear list.

Object: perform m access operations.

Cost of accessing i^{th} item $= i$.

Update primitive: Swap any two adjacent items, at a cost of 1. Can be performed at any time.

$a, b, c, d, e, f, \ldots$

access e

$e, a, b, c, d, f, \ldots$

Move-to-front heuristic (MTF): Move each accessed item to front of list (via $i-1$ swaps)

Total cost to access item $i = 2i-1$.

Single exchange heuristic (SE): Swap accessed item with its predecessor.

Frequency count heuristic (FC): Keep items in decreasing order by access frequency.

Most previous results are average-case:

Fixed access probabilities $P_1, P_2, \ldots, P_n$;

each access is independent.

Optimum algorithm: static list with items arranged in decreasing access probability.

Classic result: Average asymptotic access cost
of FC is optimum,
of MTF is within a factor of 2 of optimum (not counting update cost).

Rivest: SE asymptotically better than MTF on average.

Various results about different heuristics, rate of convergence, etc.

Bentley, McGeough: Amortized cost of MTF within a factor of 2 of any static-list algorithm.

Sleator-Tarjan: Amortized cost of MTF within a factor of 2 of any algorithm.

(both results do not count update cost of MTF, increases constant factor to 4).

Experiments (Bentley, McGeough) show that MTF is sometimes <u>better</u> than FC on realistic data.

Potential = 2 × # inversions in MTF list vs. adversary list (A)

$$\leq \binom{n}{2} = \frac{n(n-1)}{2}$$

A: . . . i . . . j . . .

MTF: . . . . j . . i . . . .

Access i :

A: 1 2 3 . . . i . . .

MTF: $\underbrace{. \; . \; . \; . \; . \; . \; . \; . \; .}_{k}$ i . .

At least k-i items > i, $\Phi \downarrow 1$ for each

At most i-1 items < i, $\Phi \uparrow 1$ for each

Actual cost = 2k-1

$$\Delta\Phi \leq 2(i-1) - 2(k-i)$$

$$\text{Am. cost (MTF)} = 2k-1+\Delta\Phi \leq 4i-3$$

$$\leq 4 \text{ Act. cost (A)}$$

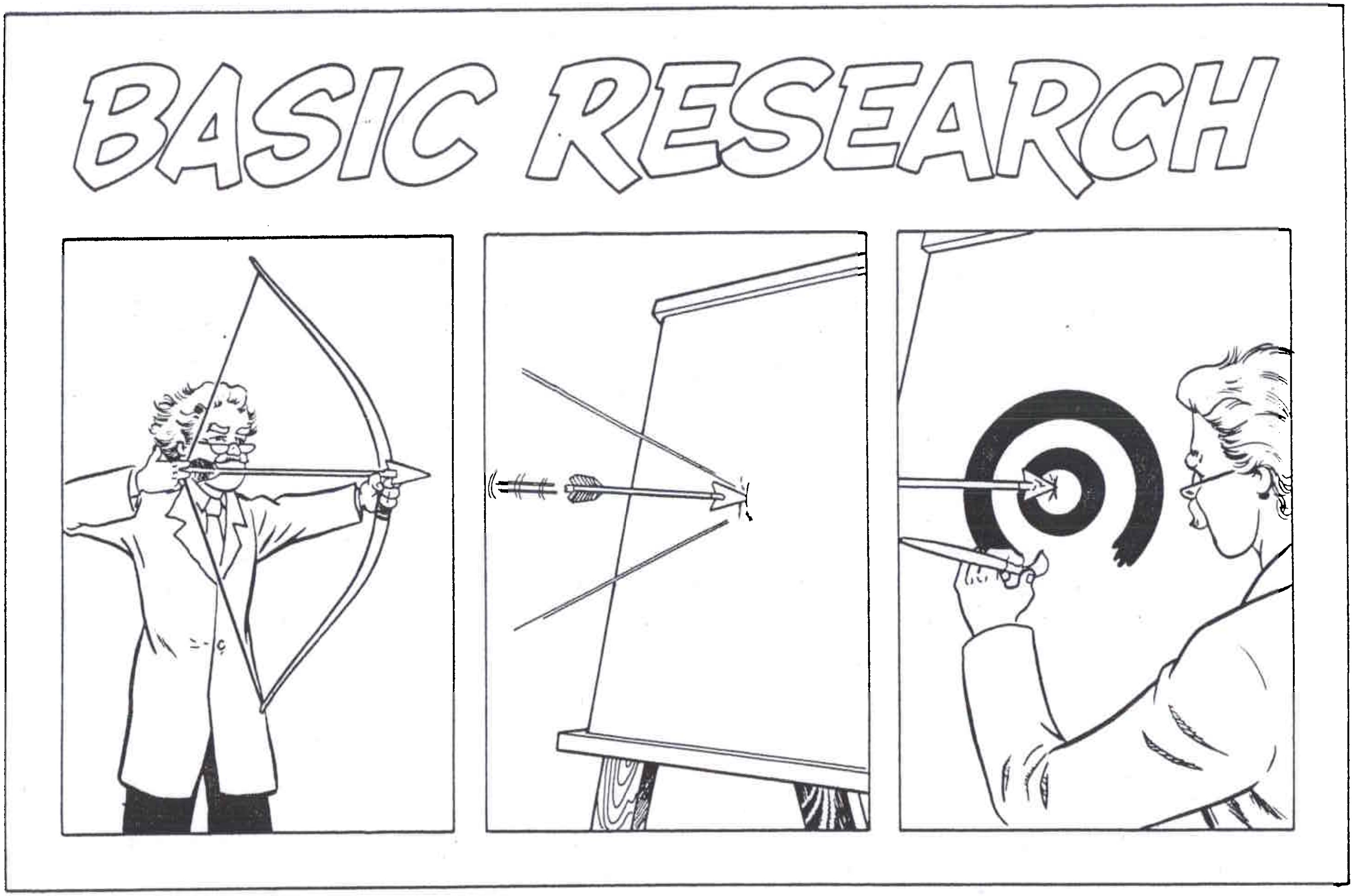
Different cost model:

arbitrary exchanges cost 1

Then off-line can beat on-line by a factor of n (always access last in on-line's list)

Other settings for competitive analysis:

caching, paging, etc.


BASIC RESEARCH