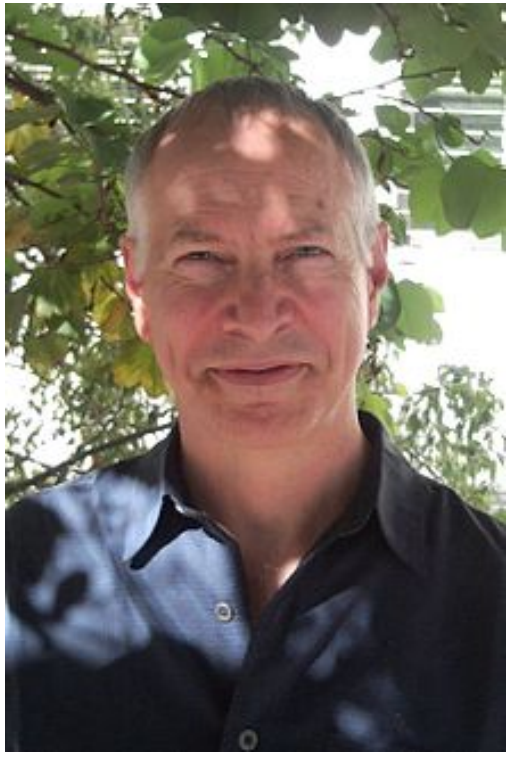
ACM Turing Award Winners 1986: John Edward Hopcroft and Robert Endre Tarjan

Venkatesh Raman
The Institute of Mathematical Sciences, Chennai

Talk at ACM Chennai Chapter
September 27, 2018



The Turing Award Citation 1986

for fundamental achievements in the design and analysis of algorithms and data structures.

Early Life and Career (Hopcroft)

Early Life and Career (Hopcroft)

- Born on October 7, 1939 in Seattle, Washington
- Bachelor's from Seattle University, Masters from Stanford 1962
- PhD in (Electrical Engineering) from Stanford 1964

Early Life and Career (Hopcroft)

- Born on October 7, 1939 in Seattle, Washington
- Bachelor's from Seattle University, Masters from Stanford 1962
- PhD in (Electrical Engineering) from Stanford 1964
- At Princeton for 3 years and developed the first course on theory of computation (thanks to McCluskey).

Early life and career (Hopcroft)

Early life and career (Hopcroft)

- Taught the first course on automata theory

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey,

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.
- Introduction to Automata Theory, Languages and Computation – Hopcroft and Ullman (1969, 1979, 2000; 2001 and 2003 with Rajeev Motwani)

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.
- Introduction to Automata Theory, Languages and Computation – Hopcroft and Ullman (1969, 1979, 2000; 2001 and 2003 with Rajeev Motwani)
- Won IEEE John Von Neumann Medal along with Ullmann in 2010.

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.
- Introduction to Automata Theory, Languages and Computation – Hopcroft and Ullman (1969, 1979, 2000; 2001 and 2003 with Rajeev Motwani)
- Won IEEE John Von Neumann Medal along with Ullmann in 2010.
- Moves to Cornell (on invitation from Hartmanis) and is there since 1967.

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.
- Introduction to Automata Theory, Languages and Computation – Hopcroft and Ullman (1969, 1979, 2000; 2001 and 2003 with Rajeev Motwani)
- Won IEEE John Von Neumann Medal along with Ullmann in 2010.
- Moves to Cornell (on invitation from Hartmanis) and is there since 1967.
- Sabbatical to Stanford in 1972.

Early life and career (Hopcroft)

- Taught the first course on automata theory on the advice of Edward McCluskey, based on papers
  - by McCulloch and Pitts (neurophysiology, networks, regular expression),
  - Rabin and Scott,
  - Bachus and Naur,
  - Chomsky,
  - Turing and
  - Hartmanis and Stearns.
- Introduction to Automata Theory, Languages and Computation – Hopcroft and Ullman (1969, 1979, 2000; 2001 and 2003 with Rajeev Motwani)
- Won IEEE John Von Neumann Medal along with Ullmann in 2010.
- Moves to Cornell (on invitation from Hartmanis) and is there since 1967.
- Sabbatical to Stanford in 1972. Turing award work with Tarjan happens here.

Early Life (Tarjan)

Early Life (Tarjan)

- Born on April 30, 1948 in Pomona, California
- BMath from Caltech 1969
- Masters from Stanford 1971

Early Life (Tarjan)

- Born on April 30, 1948 in Pomona, California
- BMath from Caltech 1969
- Masters from Stanford 1971
- PhD from Stanford 1972 (supervised by Knuth and Floyd, but worked with Hopcroft who was on a sabbatical from Cornell)
- PhD thesis was on a planarity algorithm that won him the Turing Award

Algorithms research at that time

Algorithms research at that time

- Knuth's fundamental algorithms book released –

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical,

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical, asymptotics only for approximation of summations

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical, asymptotics only for approximation of summations
- Turing machine algorithms

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical, asymptotics only for approximation of summations
- Turing machine algorithms
- Cook, Karp, Edmonds view on polyonomial time algorithms

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical, asymptotics only for approximation of summations
- Turing machine algorithms
- Cook, Karp, Edmonds view on polyonomial time algorithms
- Most papers had a program, table and graphs

Algorithms research at that time

- Knuth's fundamental algorithms book released – mostly numerical, asymptotics only for approximation of summations
- Turing machine algorithms
- Cook, Karp, Edmonds view on polyonomial time algorithms
- Most papers had a program, table and graphs
- Graph, algorithms for graph exploration were there (for example, in AI), but no systematic analysis.

Planarity Algorithms at that time

Planarity Algorithms at that time

- Kuratowski's theorem (G is non-planar if and only if it has a subdivision of K_5 or $K_{3,3}$)

Planarity Algorithms at that time

- Kuratowski's theorem (G is non-planar if and only if it has a subdivision of K_5 or $K_{3,3}$)
- Algorithms based on finding a cycle, deleting it, and embedding each of the remaining pieces with the cycle.

Planarity Algorithms at that time

- Kuratowski's theorem (G is non-planar if and only if it has a subdivision of K_5 or $K_{3,3}$)
- Algorithms based on finding a cycle, deleting it, and embedding each of the remaining pieces with the cycle.
- No theoretical analysis

Areas of Contributions (Hopcroft and Tarjan)

Areas of Contributions (Hopcroft and Tarjan)

- Laying the foundations of the model for asymptotic analysis of algorithms ("Big-Oh" notation, RAM model, Worst Case Analysis)

Areas of Contributions (Hopcroft and Tarjan)

- Laying the foundations of the model for asymptotic analysis of algorithms ("Big-Oh" notation, RAM model, Worst Case Analysis)
- Formally analysing and recognising adjacency list and Depth First Search as important paradigms.

Areas of Contributions (Hopcroft and Tarjan)

- Laying the foundations of the model for asymptotic analysis of algorithms ("Big-Oh" notation, RAM model, Worst Case Analysis)
- Formally analysing and recognising adjacency list and Depth First Search as important paradigms.
- Devising a linear time algorithm for Planarity testing
- Planar graph isomorphism

Areas of Contributions (Hopcroft and Tarjan)

- Laying the foundations of the model for asymptotic analysis of algorithms ("Big-Oh" notation, RAM model, Worst Case Analysis)
- Formally analysing and recognising adjacency list and Depth First Search as important paradigms.
- Devising a linear time algorithm for Planarity testing
- Planar graph isomorphism

- Efficient Planarity Testing (JACM 1974, IFIP 1971) with John Hopcroft
- Depth First Search and Linear Graph Algorithms (FOCS 1971, SIAM J. Computing 1972)
- Dividing a graph into triconnected components (SIAM J. Computing 1973) with John Hopcroft
- Isomorphism of planar graphs (CCC 1972, JCSS 1973) with John Hopcroft

Other Contributions of Hopcroft

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)
  - Data Structures and Algorithms – Aho, Hopcroft and Ullman (1983)

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)
  - Data Structures and Algorithms – Aho, Hopcroft and Ullman (1983) and more recently

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)
  - Data Structures and Algorithms – Aho, Hopcroft and Ullman (1983) and more recently
  - Foundations of Data Science – Blum, Hopcroft and Kannan (2017)

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)
  - Data Structures and Algorithms – Aho, Hopcroft and Ullman (1983) and more recently
  - Foundations of Data Science – Blum, Hopcroft and Kannan (2017)
- Hopcroft-Karp $O(m\sqrt{n})$ algorithm for maximum matching in bipartite graphs (1973).

Other Contributions of Hopcroft

- Developed algorithms course at Cornell and wrote
  - Design and Analysis of Computer Algorithms – Aho, Hopcroft and Ullman (1974)
  - Data Structures and Algorithms – Aho, Hopcroft and Ullman (1983) and more recently
  - Foundations of Data Science – Blum, Hopcroft and Kannan (2017)
- Hopcroft-Karp $O(m\sqrt{n})$ algorithm for maximum matching in bipartite graphs (1973).
- Automata, Complexity theory, Algorithms, Computational Geometry and Robotics, Social Networks, AI, Deep Learning.....

Notable Awards and Students (Hopcroft)

Notable Awards and Students (Hopcroft)

- ACM Fellow (1994)
- Karl V Karlstrom Outstanding Educator Award (ACM) (2008)
- IEEE John Von Neumann Medal (2010) (with Ullmann)
- *Students* Alfred Aho, Gilles Brassard, Zvi Galil, Cynthia Dwork, 30 of them.

Career (Tarjan)

Career (Tarjan)

- Cornell 1972-73
- UC Berkeley 1973-75
- Stanford 1974-80
- New York University 1981-85
- Princeton since 1985
- Also at Bell Labs (1980-1989), NEC (1989-97), Intertrust technologies (1997-2001, 2014-present), Compaq (2002), HP (2006-2013), Microsoft (2013).

Notable Awards and Students (Tarjan)

Notable Awards and Students (Tarjan)

- Nevanlinna (1983) – the first receipent
- ACM Fellow (1994)
- Paris Kannellakis (1999), ACM – theory and practice (with Sleator for Splay Trees)
- *Students* Daniel Sleator, Samuel Bent, Haim Kaplan, Monika Henzinger, Neil Sarnak, (26 of them)

Other Areas of Contributions (Tarjan)

Other Areas of Contributions (Tarjan)

Algorithms

Other Areas of Contributions (Tarjan)

Algorithms

- Linear time algorithm for finding the median (or any k-th smallest) of a list (along with Blum, Floyd, Pratt and Rivest).

Other Areas of Contributions (Tarjan)

Algorithms

- Linear time algorithm for finding the median (or any k-th smallest) of a list (along with Blum, Floyd, Pratt and Rivest).
- Fast algorithms for network flows (with Goldberg, Ahuja, Orlin, ...)
- Randomized Linear time algorithm for Minimum Spanning Trees (JACM 1995) with Klein and Karger

Other Areas of Contributions (Tarjan)

Other Areas of Contributions (Tarjan)

Data Structures

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity
- Amortized analysis of Union-Find Data Structures, Fibonacci heaps

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity
- Amortized analysis of Union-Find Data Structures, Fibonacci heaps
- Self-organizing data structures (lists, splay trees, skew heaps, ...)

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity
- Amortized analysis of Union-Find Data Structures, Fibonacci heaps
- Self-organizing data structures (lists, splay trees, skew heaps, ...)
- Persistent data structures (data structures that remember history),

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity
- Amortized analysis of Union-Find Data Structures, Fibonacci heaps
- Self-organizing data structures (lists, splay trees, skew heaps, ...)
- Persistent data structures (data structures that remember history),
- Link-Cut trees to maintain a forest of trees to support find, union, lca operations.

Other Areas of Contributions (Tarjan)

Data Structures

- Amortized Complexity
- Amortized analysis of Union-Find Data Structures, Fibonacci heaps
- Self-organizing data structures (lists, splay trees, skew heaps, ...)
- Persistent data structures (data structures that remember history),
- Link-Cut trees to maintain a forest of trees to support find, union, lca operations.
- Efficient data structure to support least common ancestor

Book by Tarjan

- Data Structures and Network Algorithms (SIAM publications 1983)

Amortized Complexity, Splay Trees, Dynamic Optimality Conjecture

Data Structuring Problem

Data Structuring Problem

Given some data (a set/sequence/graph ...),

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

- To search from a list of elements from a total order,

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

- To search from a list of elements from a total order, keep it sorted, so queries take $O(\log n)$ time.

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

- To search from a list of elements from a total order, keep it sorted, so queries take $O(\log n)$ time.
- To support insert and deletemax (as needed for implementing jobs in a printer or CPU),

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

- To search from a list of elements from a total order, keep it sorted, so queries take $O(\log n)$ time.
- To support insert and deletemax (as needed for implementing jobs in a printer or CPU), keep the elements in a heap and so queries take $O(\log n)$ time.

Data Structuring Problem

Given some data (a set/sequence/graph ...), organize it to support some specific queries fast.

For example,

- To search from a list of elements from a total order, keep it sorted, so queries take $O(\log n)$ time.
- To support insert and deletemax (as needed for implementing jobs in a printer or CPU), keep the elements in a heap and so queries take $O(\log n)$ time.
- Keeping the elements in a balanced binary search tree makes dictionary queries (insert, search, delete) possible in logarithmic time.
- *All times are worst case*

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.
- Used in algorithms

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.
- Used in algorithms
  - Shortest Paths, Minimum Spanning Tree algorithms use some data structure query (deletemin, ...) repeatedly

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.
- Used in algorithms
  - Shortest Paths, Minimum Spanning Tree algorithms use some data structure query (deletemin, ...) repeatedly
  - To minimize time for the algorithm, worst case for a single query is NOT that important, but we need to optimize worst case time for a *sequence* of operations.

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.
- Used in algorithms
  - Shortest Paths, Minimum Spanning Tree algorithms use some data structure query (deletemin, ...) repeatedly
  - To minimize time for the algorithm, worst case for a single query is NOT that important, but we need to optimize worst case time for a *sequence* of operations.
- *Amortized Cost of the operation* = max $\{$(cost for a sequence of m operations)$/m$ over all sequences $\}$

Worst Case vs Amortized Case

Data Structuring problem comes up in two kinds of applications ...

- It is itself the problem
  - For example, we need fast response to Google queries
  - so worst case for a single query is important.
- Used in algorithms
  - Shortest Paths, Minimum Spanning Tree algorithms use some data structure query (deletemin, ...) repeatedly
  - To minimize time for the algorithm, worst case for a single query is NOT that important, but we need to optimize worst case time for a *sequence* of operations.
- *Amortized Cost of the operation* = max $\{$(cost for a sequence of m operations)$/m$ over all sequences $\}$
- Amortized Cost $\leq$ Worst case cost, but it can be a gross overestimate

Amortized Complexity – Motivation and Challenge

Amortized Complexity – Motivation and Challenge

- *Motivation* Expensive operation may not be that common,

Amortized Complexity – Motivation and Challenge

- *Motivation* Expensive operation may not be that common, in particular expensive operation maybe preceded/succeeded by several cheap operations.

Amortized Complexity – Motivation and Challenge

- *Motivation* Expensive operation may not be that common, in particular expensive operation maybe preceded/succeeded by several cheap operations.
  Example – incrementing a k-bit counter, worst case is k,

Amortized Complexity – Motivation and Challenge

- *Motivation* Expensive operation may not be that common, in particular expensive operation maybe preceded/succeeded by several cheap operations.
  Example – incrementing a k-bit counter, worst case is k, amortized is 2.

Amortized Complexity – Motivation and Challenge

- *Motivation* Expensive operation may not be that common, in particular expensive operation maybe preceded/succeeded by several cheap operations.
  Example – incrementing a k-bit counter, worst case is k, amortized is 2.
- *Challenge* – Capturing the interdependence of expensive vs cheap operations.

Amortized Complexity – Estimation (Sleator and Tarjan)

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure;

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure
- At any point, the charge in the data structure is the (non-negative) "potential" in the data structure.

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure
- At any point, the charge in the data structure is the (non-negative) "potential" in the data structure.
- Charge with the operation is its amortized complexity (which is also the same as actual cost minus the potential difference).

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure
- At any point, the charge in the data structure is the (non-negative) "potential" in the data structure.
- Charge with the operation is its amortized complexity (which is also the same as actual cost minus the potential difference).
- Example: Incrementing a k-bit counter

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure
- At any point, the charge in the data structure is the (non-negative) "potential" in the data structure.
- Charge with the operation is its amortized complexity (which is also the same as actual cost minus the potential difference).
- Example: Incrementing a k-bit counter
  Potential – number of 1s in the bit-vector

Amortized Complexity – Estimation (Sleator and Tarjan)

- Each operation comes with a "charge" – its amortized complexity
- If it uses less, then keeps the excess in the data structure; if it uses more, draw from the data structure
- At any point, the charge in the data structure is the (non-negative) "potential" in the data structure.
- Charge with the operation is its amortized complexity (which is also the same as actual cost minus the potential difference).
- Example: Incrementing a k-bit counter
  Potential – number of 1s in the bit-vector
  Amortized complexity – 2 (one to change a 0 to 1, and one with the new 1).

Spinoff of Amortized Analysis

Spinoff of Amortized Analysis

- Resulted in amortized (hence tight) analysis of some data structures and algorithms

Spinoff of Amortized Analysis

- Resulted in amortized (hence tight) analysis of some data structures and algorithms
- New (simpler) data structures were designed keeping amortized complexity in mind (Splay Trees, Skew Heaps, Fibonacci heaps, ...)

Spinoff of Amortized Analysis

- Resulted in amortized (hence tight) analysis of some data structures and algorithms
- New (simpler) data structures were designed keeping amortized complexity in mind (Splay Trees, Skew Heaps, Fibonacci heaps, ...)
- Explained some practical heuristics

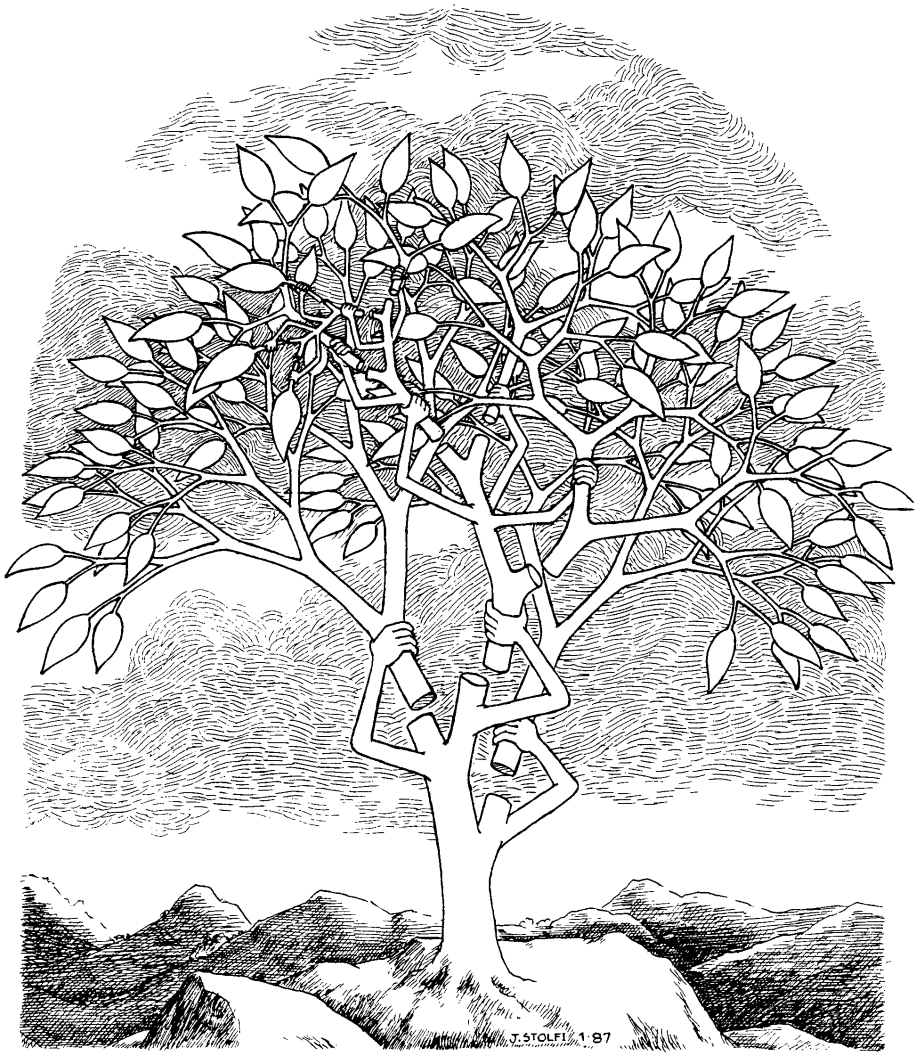
Spinoff of Amortized Analysis

- Resulted in amortized (hence tight) analysis of some data structures and algorithms
- New (simpler) data structures were designed keeping amortized complexity in mind (Splay Trees, Skew Heaps, Fibonacci heaps, ...)
- Explained some practical heuristics
  For example, Move to Front (MTF) heuristic for lists has been shown to be 2-competitive with any dynamic list heuristic.

Spinoff of Amortized Analysis

- Resulted in amortized (hence tight) analysis of some data structures and algorithms
- New (simpler) data structures were designed keeping amortized complexity in mind (Splay Trees, Skew Heaps, Fibonacci heaps, ...)
- Explained some practical heuristics
  For example, Move to Front (MTF) heuristic for lists has been shown to be 2-competitive with any dynamic list heuristic.
- Dynamic Optimality Conjecture (of Splay Trees)

Self-adjusting trees



A Self-Adjusting Search Tree

Splay Trees – an alternative to balanced binary search trees

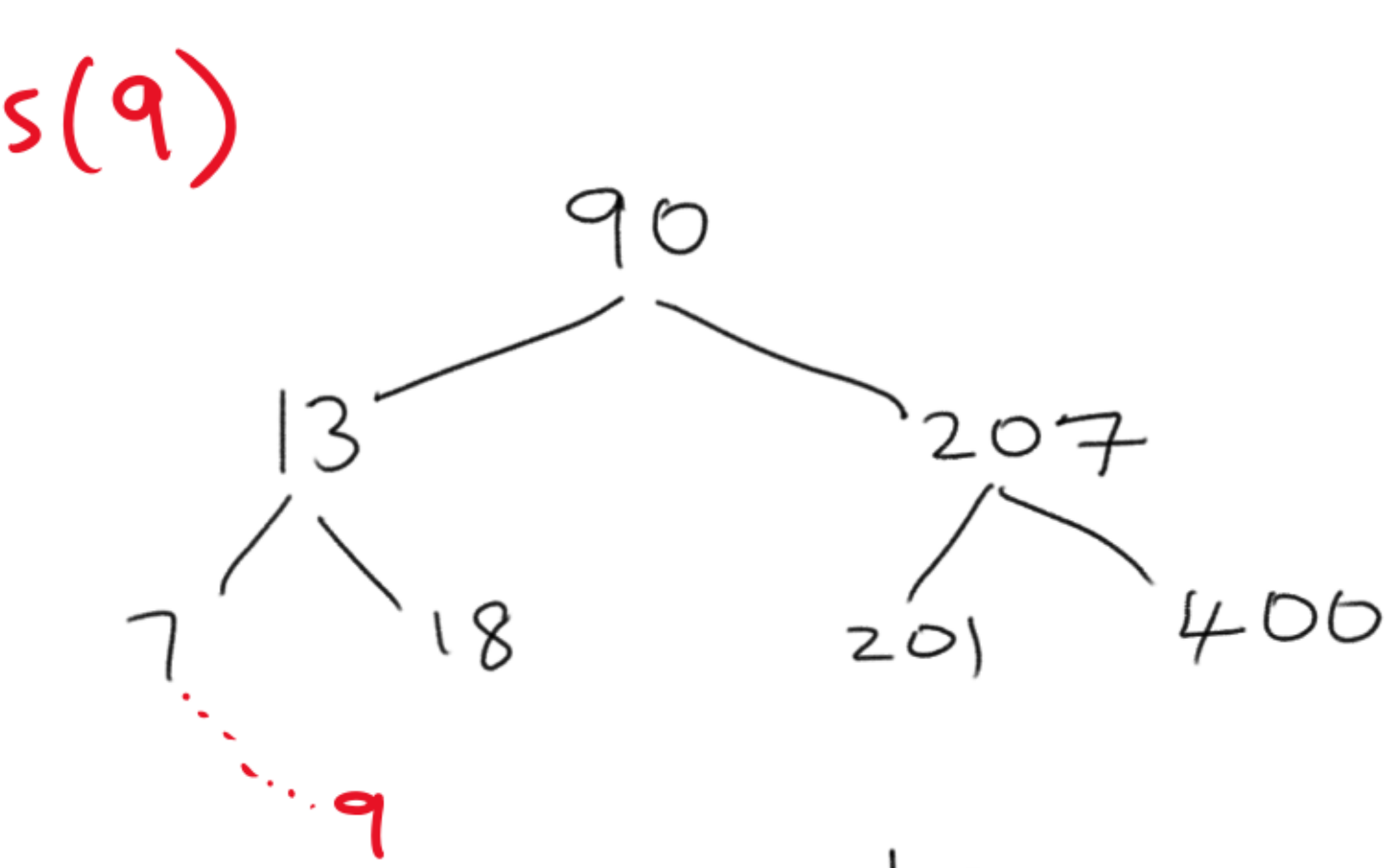
Binary Search

7, 13, 18, 90, 201, 207, 400

Search – $O(\log n)$

Insert – $\Theta(n)$

ins(9)



Binary Search tree

Values (left subtree (x)) < Value (x)

< Values (rt subtree of x)

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees
  - Maintain some local balance criteria (for example, $|height(left(x) - height(right(x))| \leq 1$ for every x, in the case of AVL trees) to ensure logarithmic height.

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees
  - Maintain some local balance criteria (for example, $|height(left(x) - height(right(x))| \leq 1$ for every x, in the case of AVL trees) to ensure logarithmic height.
  - When an insertion causes violation of the balance criteria, fix that using what are called *rotations*.

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees
  - Maintain some local balance criteria (for example, $|height(left(x) - height(right(x))| \leq 1$ for every x, in the case of AVL trees) to ensure logarithmic height.
  - When an insertion causes violation of the balance criteria, fix that using what are called *rotations*.
  - Uses additional space (at least a bit per node, in addition to the pointers) to maintain balance criteria

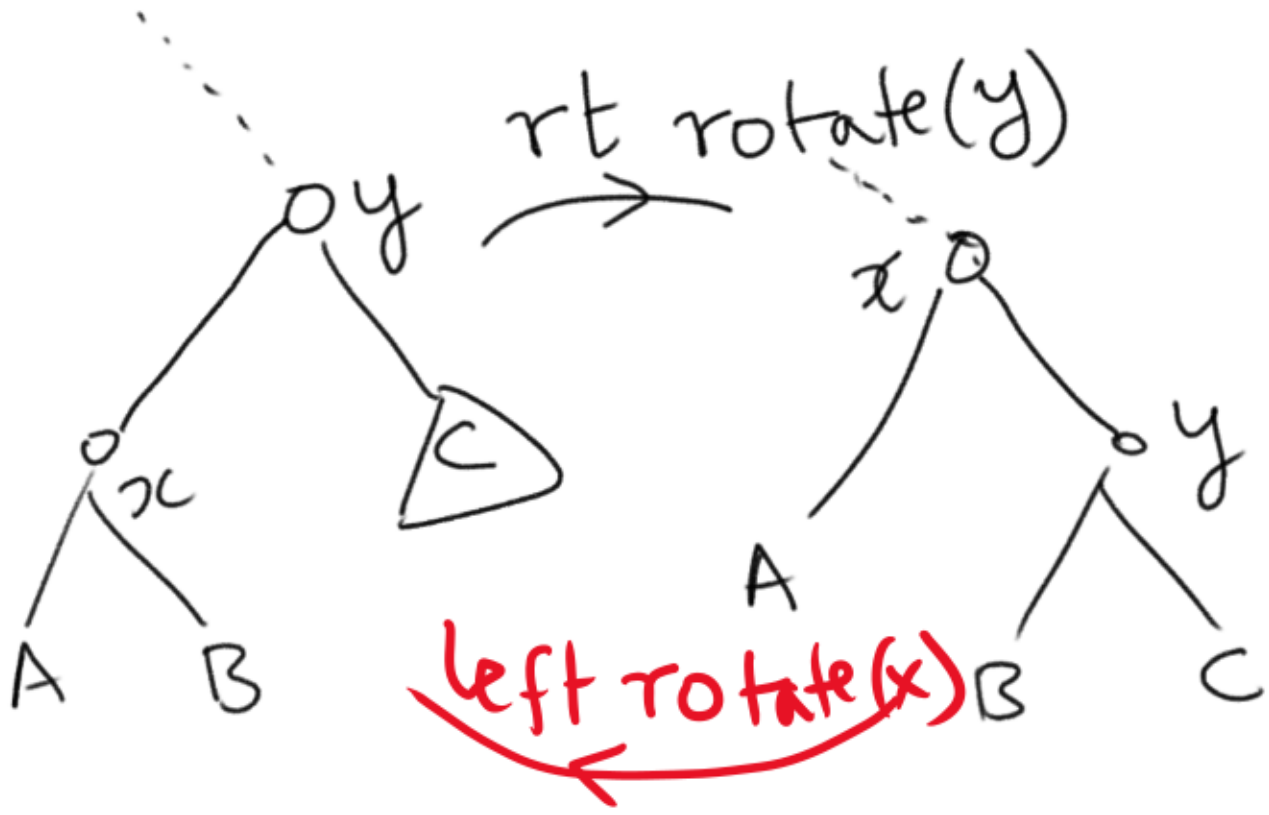
Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees
  - Maintain some local balance criteria (for example, $|height(left(x) - height(right(x))| \leq 1$ for every x, in the case of AVL trees) to ensure logarithmic height.
  - When an insertion causes violation of the balance criteria, fix that using what are called *rotations*.
  - Uses additional space (at least a bit per node, in addition to the pointers) to maintain balance criteria
  - Involved case analysis for insertion (and deletion)

Balanced Binary Search Trees

- Subsequent insertions into the binary search tree can make the tree unbalanced resulting in $\Theta(n)$ height, and hence time for insertions and search.
- Balanced binary search trees
  - Maintain some local balance criteria (for example, $|height(left(x) - height(right(x))| \leq 1$ for every x, in the case of AVL trees) to ensure logarithmic height.
  - When an insertion causes violation of the balance criteria, fix that using what are called *rotations*.
  - Uses additional space (at least a bit per node, in addition to the pointers) to maintain balance criteria
  - Involved case analysis for insertion (and deletion)
  - AVL trees, Red-Black Trees, B-trees, ...

ROTATIONS



- Constant number of pointer changes per rotation
- Binary search tree property maintained

Splay Trees

Splay Trees

- After every insertion or search (or delete), we will *splay* at that node looking at its parent and grand parent, using double rotations, moving it eventually to the root.

Splay Trees

- After every insertion or search (or delete), we will *splay* at that node looking at its parent and grand parent, using double rotations, moving it eventually to the root.
- Independent of the shape of the tree.

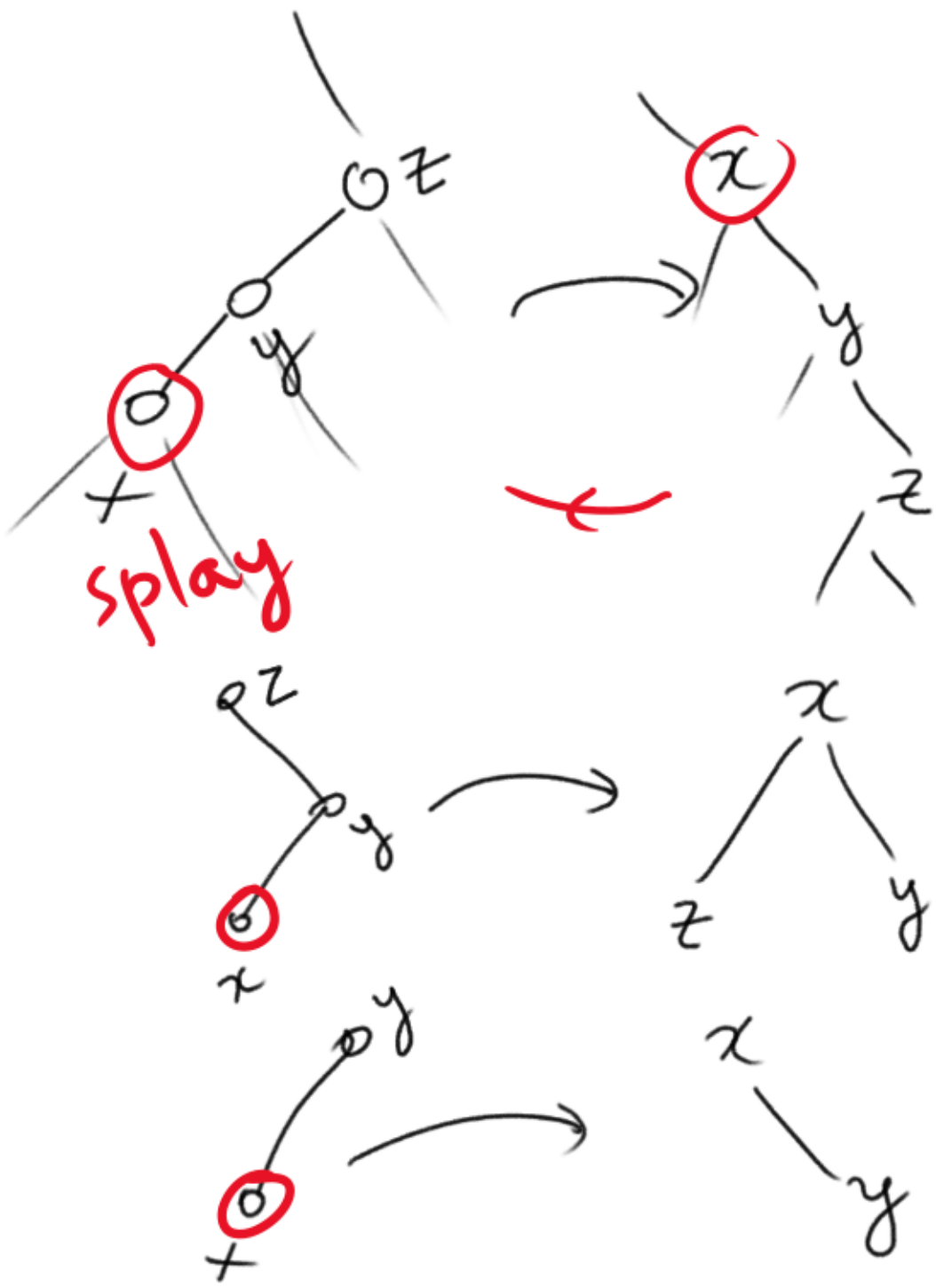
Splay Trees

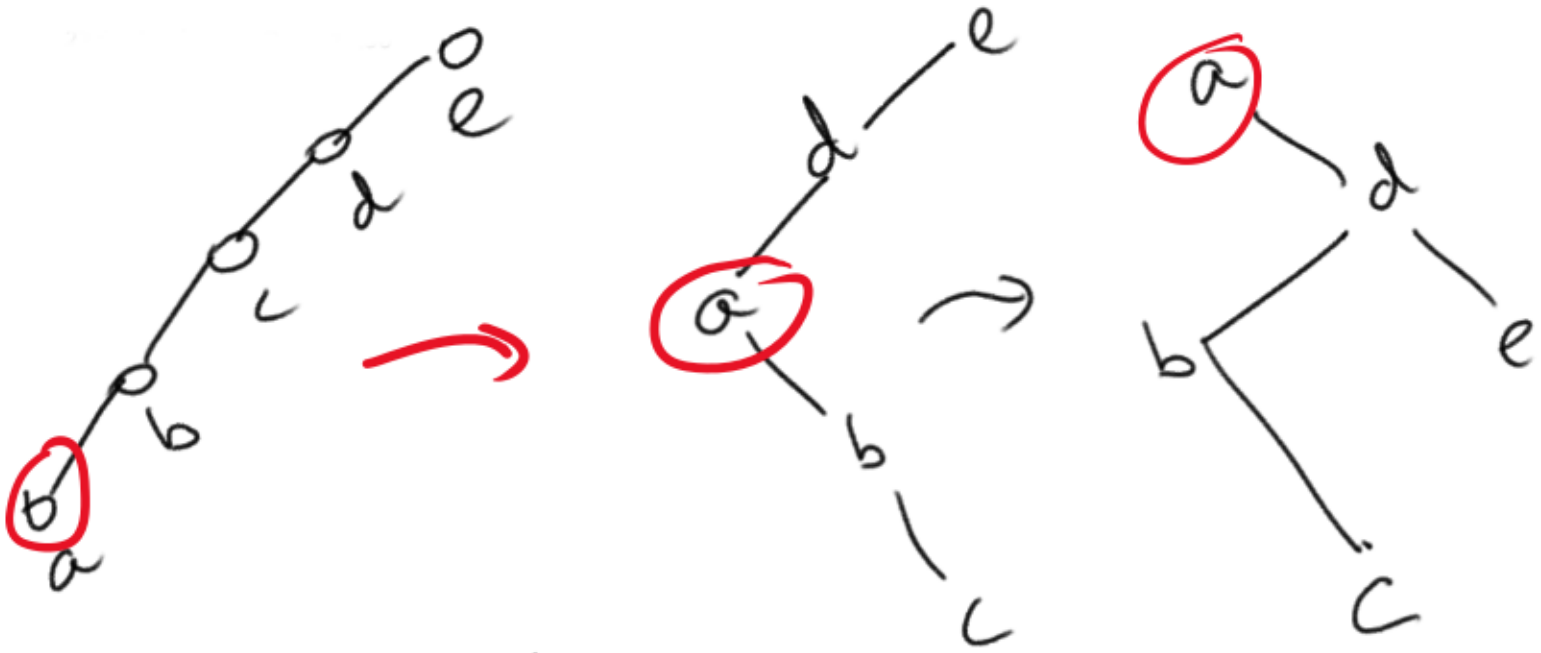
- After every insertion or search (or delete), we will *splay* at that node looking at its parent and grand parent, using double rotations, moving it eventually to the root.
- Independent of the shape of the tree.
- Each splay cuts the depth of each node in the access path by half, and increases the depths of a few others by one or two.

Splay Trees

- After every insertion or search (or delete), we will *splay* at that node looking at its parent and grand parent, using double rotations, moving it eventually to the root.
- Independent of the shape of the tree.
- Each splay cuts the depth of each node in the access path by half, and increases the depths of a few others by one or two.
- (Sleator-Tarjan) Amortized Complexity of splaying (hence access/insert/delete) is $O(\log n)$ on any starting tree on n nodes.

SPLAYING





single rotations would have yielded

Static Optimality of Splay Trees

Static Optimality of Splay Trees

- Adapts to the access distribution. I.e. if the i-th smallest element is accessed f_i times, then the total cost of a sequence of m operations is proportional to

Static Optimality of Splay Trees

- Adapts to the access distribution. I.e. if the i-th smallest element is accessed f_i times, then the total cost of a sequence of m operations is proportional to $\sum_{i=1}^{n} f_i \lg(m/f_i)$ (related to the entropy of the distribution).

Static Optimality of Splay Trees

- Adapts to the access distribution. I.e. if the i-th smallest element is accessed f_i times, then the total cost of a sequence of m operations is proportional to $\sum_{i=1}^{n} f_i \lg(m/f_i)$ (related to the entropy of the distribution). Note that the algorithm need not know the f_i's.

Static Optimality of Splay Trees

- Adapts to the access distribution. I.e. if the i-th smallest element is accessed f_i times, then the total cost of a sequence of m operations is proportional to $\sum_{i=1}^{n} f_i \lg(m/f_i)$ (related to the entropy of the distribution). Note that the algorithm need not know the f_i's.
  (Static Optimality Theorem of Sleator and Tarjan)

Static Optimality of Splay Trees

- Adapts to the access distribution. I.e. if the i-th smallest element is accessed f_i times, then the total cost of a sequence of m operations is proportional to $\sum_{i=1}^{n} f_i \lg(m/f_i)$ (related to the entropy of the distribution). Note that the algorithm need not know the f_i's.
  (Static Optimality Theorem of Sleator and Tarjan)
- If the f_is are known, such an optimum binary search tree can be constructed by a dynamic programming based algorithm (Knuth, Cormen et al.)

Dynamic Optimality Conjecture

Dynamic Optimality Conjecture

- Consider a heuristic which after accessing an element, reorganizes the search tree (possibly based on future request) by performing rotations with an additional cost of number of rotations performed. We call this a self-adjusting search heuristic.

Dynamic Optimality Conjecture

- Consider a heuristic which after accessing an element, reorganizes the search tree (possibly based on future request) by performing rotations with an additional cost of number of rotations performed. We call this a self-adjusting search heuristic.
- *Dynamic Optimality Conjecture* (Sleator and Tarjan): Total cost of m splay operations on an initial binary search tree on n elements is at most constant times the cost of any other self-adjusting heuristic.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is to arrange them in decreasing order of the frequencies.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is to arrange them in decreasing order of the frequencies.
- We can define "self-adjusting heuristic" in the same way –

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is to arrange them in decreasing order of the frequencies.
- We can define "self-adjusting heuristic" in the same way – after accessing an element at position i, reorder the elements up to position i with cost proporitional to the number of moves.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is to arrange them in decreasing order of the frequencies.
- We can define "self-adjusting heuristic" in the same way – after accessing an element at position i, reorder the elements up to position i with cost proporitional to the number of moves.
- *Move to Front heuristic:* After accessing an element, move it to the front of the list.

Dynamic Optimality of Move to Front heuristic on lists

Consider organising a list of n elements where accessing the element at position i costs i.

- If we know the frequencies of the items to be accessed, the static optimal way is to arrange them in decreasing order of the frequencies.
- We can define "self-adjusting heuristic" in the same way – after accessing an element at position i, reorder the elements up to position i with cost proporitional to the number of moves.
- *Move to Front heuristic:* After accessing an element, move it to the front of the list.
- Theorem (Sleator and Tarjan): For a list, cost of a sequence of accesses using Move to Front heuristic is at most twice the cost of any self-organizing heuristic for any sequence of accesses in an initial list of n elements.

Thank You