CS 261: Data Structures

Week 8: Navigating in trees

Lecture 8c: Maintaining order in a list

David Eppstein
University of California, Irvine

Spring Quarter, 2023

The list ordering problem

List ordering vs tree ancestors

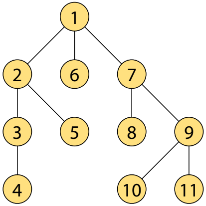
Simplification of common ancestor problem:

Test whether one tree vertex is an ancestor of another, or not

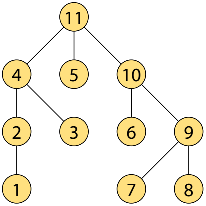
Data structure:

Number in preorder

Number in postorder



preorder



postorder

Ancestor = earlier in preorder and later in postorder

Dynamic order comparison

Maintain a collection of elements ordered into a list

Operations:

- Insert x at the start or end of the list
- Insert x immediately before or after another element y
- Find the element immediately before or after x
- Remove x
- Test whether element x is earlier than or later than element y

Most operations can easily be done in constant time,
for example by using doubly linked lists

The only missing one: testing relative ordering

House numbering problem

Typical properties of numbers of buildings in US streets:

- Ordered: number tells you relative position along street
- Usually small integers
- Not necessarily consecutive: there may be gaps in the numbering
- Renumbering is expensive, so don't do it very often

Intuition: apply similar scheme to list ordering by numbering list elements and using numbers to test relative position

Partial history

- Dietz [1982]: logarithmic update, $O(1)$ order-comparison
- Tsakalidis [1984]: constant amortized update and comparison
- Dietz and Sleator [1987]: maintain ordered numbering, all numbers polynomially large, constant amortized update, complicated
- Bender et al. [2002]: simplification of same results
- Bender et al. [2017]: worst case rather than amortized
- Devanny et al. [2017]: few relabelings per element

We will follow Bender et al. [2002]

Application of house numbering: Dynamic arrays, revisited

Dynamic arrays with insertion and deletion

Suppose we want to maintain a sequence of values with the following operations:

- Look up the value at position i in the sequence
- Change the value at position i in the sequence
- Add a new value at position i in the sequence, shifting all later values to higher positions
- Remove the value at position i in the sequence, shifting all later values to earlier positions

Dynamic arrays allow only the first two operations, and add/remove at the end of the array; adding and remove fast lookup of the element at position i, and fast insertion or deletion at the end of the array

What if we want to extend arrays to allow insertion or deletion at other positions?

Dynamic arrays from augmented trees

Store the sequence in a binary search tree augmented for ranking and unranking

(The sequence order is the left-to-right tree order; we don't need to store keys with the tree nodes, only the associated array values, so house numbering not needed.)

To find or change the value at position i: use unrank to find its tree node

To add or remove a value: standard binary search tree insertion/deletion operations

Dynamic arrays from house numbering

Maintain:

- House-numbering solution for sequence of elements
- Dictionary mapping house numbers to linked list nodes
- Structure for ranking and unranking house numbers with $O(\log n / \log\log n)$ time per operation (mentioned briefly last week)

To find or change the value at position i: use unrank to find its house number and then use that number as a dictionary key

To add or remove a value, update the house numbers and propagate any changes in numbering to the ranking/unranking structure

House numbering solution

Terminology

We will store a doubly linked list, whose elements are called keys

Because it's stored as a linked list, we can quickly find adjacent keys
The two adjacent keys are the predecessor and successor

We wish to assign numbers to the keys to allow fast comparisons of their positions; these numbers are called tags

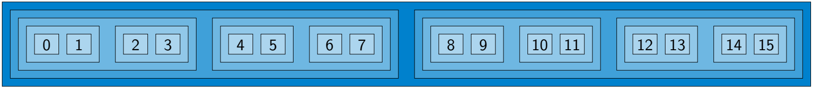
We want to maintain a correspondence keys $\rightarrow$ tags so that the numerical ordering of tags $=$ the list ordering of keys

Main idea

Delete a key $\Rightarrow$ do not renumber other tags

If we insert key x, and there is any available tag i between tags of its predecessor and successor, set $\mathrm{tag}(x) = i$

Remaining case: Partition possible tag values recursively into ranges of tags with power-of-two sizes



Find the smallest range of tags (size 2^k) surrounding new element location that is used by few keys: fewer than c^k

Renumber the keys evenly within this range (including x)

Main idea implementation details

The parameter c can be any fixed number in range $1 < c < 2$ (smaller $c \Rightarrow$ bigger tags; larger $c \Rightarrow$ more renumberings)

To find a range of tags that is used by few keys: scan left and right from x in the sequence, finding increasingly large ranges in hierarchical partition of tags, until finding a range with few keys

Let $k = \log_c n$, and let $\alpha = \log_c 2$. If max tag $> n^\alpha$, then range of all tags is bigger than 2^k and holds only $n = c^k$ keys, so $\exists$ range with few keys and search terminates

When search terminates, time it took to find the range and renumber its elements are both proportional to # keys in it

Main idea analysis

When we renumber a range of size 2^i, left or right half-range is full.

(If both half-ranges had few elements, we would have renumbered one of them before getting to the larger range.)

Therefore, when we renumber a range of size 2^k, we renumber between c^{k-1} and c^k keys and it takes total time $\Theta(c^k)$

After renumber, each half-range has $\leq \frac{c}{2} c^{k-1}$ keys, below full by a factor of $c/2 \Rightarrow$ cannot fill up again before we do another $\Omega(c^k)$ insertions $\Rightarrow$ amortized time for ranges of size 2^i is $O(1)$

The same analysis holds separately for each choice of i, but there are $O(\log n)$ choices $\Rightarrow$ total amortized time per update is $O(\log n)$

Log-shaving

To achieve constant instead of logarithmic amortized time per update, again use a blocking strategy:

- Group keys into dynamic blocks of logarithmic length (similar to bottom level of a B-tree with $B = \Theta(\log n)$)
- Use main idea to number blocks
- Allocate polynomially many tags within each block
- New key in a block gets average of predecessor and successor
- Renumber all keys in a block when block structure changes or when a new element has no tag; this happens only $O(1)$ times per $O(\log n)$ insertions

Summary

Summary

- Representation of trees and binary trees with $2n$ bits
- Blocking and table lookup strategy for saving logarithmic factors in the space bounds for many data structures
- Common ancestor problem and its applications to shortest paths and bandwidth maximization
- Equivalence between common ancestors and range minima
- Common ancestors in $O(n)$ space and $O(1)$ query time
- Maintaining order in a list in $O(1)$ amortized time
- Level ancestors in $O(n)$ space and $O(1)$ query time

References and image credits, I

Michael A. Bender, Richard Cole, Erik D. Demaine, Martin Farach-Colton, and Jack Zito. Two simplified algorithms for maintaining order in a list. In Rolf H. Möhring and Rajeev Raman, editors, *Algorithms – ESA 2002, 10th Annual European Symposium, Rome, Italy, September 17–21, 2002, Proceedings*, volume 2461 of *Lecture Notes in Computer Science*, pages 152–164. Springer, 2002. doi: 10.1007/3-540-45749-6_17.

Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, Tsvi Kopelowitz, and Pablo Montes. File maintenance: When in doubt, change the layout! In Philip N. Klein, editor, *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2017, Barcelona, Spain, Hotel Porta Fira, January 16–19*, pages 1503–1522. Society for Industrial and Applied Mathematics, 2017. doi: 10.1137/1.9781611974782.98.

References and image credits, II

William E. Devanny, Jeremy T. Fineman, Michael T. Goodrich, and Tsvi Kopelowitz. The online house numbering problem: Min-max online list labeling. In Kirk Pruhs and Christian Sohler, editors, *25th Annual European Symposium on Algorithms, ESA 2017, September 4–6, 2017, Vienna, Austria*, volume 87 of *LIPIcs*, pages 33:1–33:15. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.ESA.2017.33.

P. Dietz and D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC '87)*, pages 365–372. ACM, 1987. doi: 10.1145/28395.28434.

Paul F. Dietz. Maintaining order in a linked list. In *Proceedings of the 14th Annual ACM Symposium on Theory of Computing (STOC '82)*, pages 122–127. ACM, 1982. ISBN 978-0897910705. doi: 10.1145/800070.802184.

References and image credits, III

Infrogmation. 2825 Bell Street, New Orleans. CC-BY-SA image, February 27 2019. URL https://commons.wikimedia.org/wiki/File:Bell_Street_2800_Block_New_Orleans_Feb_2019_09.jpg.

liempdma. Cutting lumber with a swingblade sawmill. CC-BY-SA image, September 4 2018. URL https://commons.wikimedia.org/wiki/File:Cutting_lumber_with_a_swingblade_sawmill.jpg.

Athanasios K. Tsakalidis. Maintaining order in a generalized linked list. *Acta Informatica*, 21(1):101–112, 1984. doi: 10.1007/BF00289142.