CS 261: Data Structures

Week 6–7: Binary search

Lecture 6a: Balanced trees

David Eppstein
University of California, Irvine

Spring Quarter, 2025

Binary search

Exact versus binary

Exact search

We are given a set of keys (or key-value pairs)

Want to test if given query key is in the set (or find value)

Usually better to solve with hashing (constant expected time)

Binary search

The keys come from an ordered set (e.g. numbers)

Want to find a key near the query key

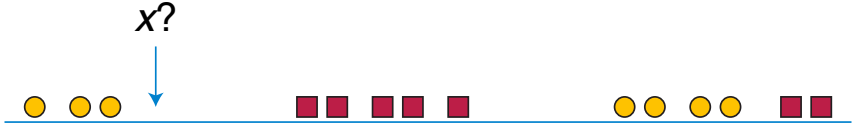
Hashing scrambles order $\Rightarrow$ not useful for nearby keys

Application: Nearest neighbor classification

Given training set of (data,classification) pairs

Want to infer classification of new data values

Method: Find nearest value in training set, copy its classification



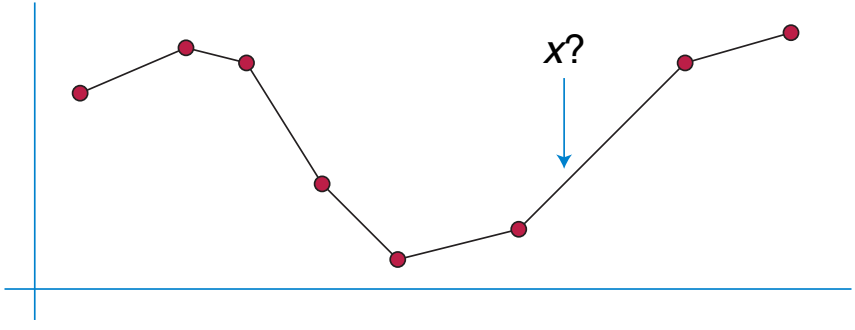
Binary search can be used for finding nearest value
but only when the data is only one-dimensional (unrealistic)

Application: Function interpolation

Given x, y pairs from unknown function $y = f(x)$

Compute approximate values of $f(x)$ for other x

Method: assume linear between given pairs



Find two pairs x_0 and x_1 on either side of given x and compute

$$y = \frac{y_0(x - x_0) + y_1(x_1 - x)}{x_1 - x_0}$$

Binary search operations

Given a S of keys from an ordered space (e.g. numbers, strings; sorting order of whole space should be defined):

- successor(q): smallest key in S that is $> q$
- predecessor(q): largest key in S that is $< q$
- nearest neighbor: must be one of q (if it is in S), successor, predecessor

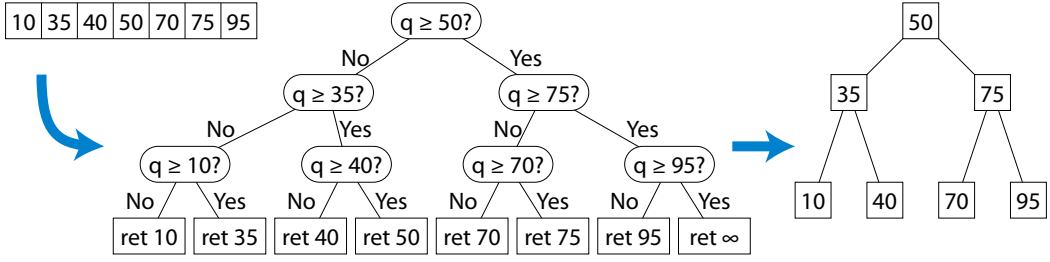
We will mainly consider successor; predecessor is very similar

Binary search tree

Data structure that encodes the sequences of comparisons made in a binary search (for instance, when searching a static array)

Each node stores

- Value that the query will be compared against
- Left child, where to go when comparison is $<$
- Right child, where to go when comparison is $\geq$



To recover sorted array, use inorder traversal:
recurse in left subtree, then root, then recurse in right subtree

Successor in binary search trees

```
define successor(q,tree):
    s = infinity
    node = tree.root
    while node != null:
        if q >= node.value:
            node = node.right
        else:
            s = node.value
            node = node.left
    return s
```

For tree derived from static array, does same steps as binary search of array, but works for any binary tree with inorder = sorted order

Balanced binary search trees

Balance

For static data, sorted array achieves $O(\log n)$ search time

For a binary search tree, search time is $O(\text{tree height})$

Balanced binary search tree: a search tree data structure for dynamic data (add or remove values) that maintains $O(\log n)$ (worst case, amortized, or expected) search time and update time.

Typically, store extra structural info on nodes to help balance

(The name refers to a different property, that the left and right sides of a static binary search tree have similar sizes, but a tree can have short search paths with subtrees of different sizes.)

Three strategies for maintaining balance

Rebuild

Let the tree become somewhat unbalanced, but rebuild subtrees when they get too far out of balance

Usually amortized; can get very close to $\log_2 n$ height

Rotate

Local changes to structure that preserve search tree ordering

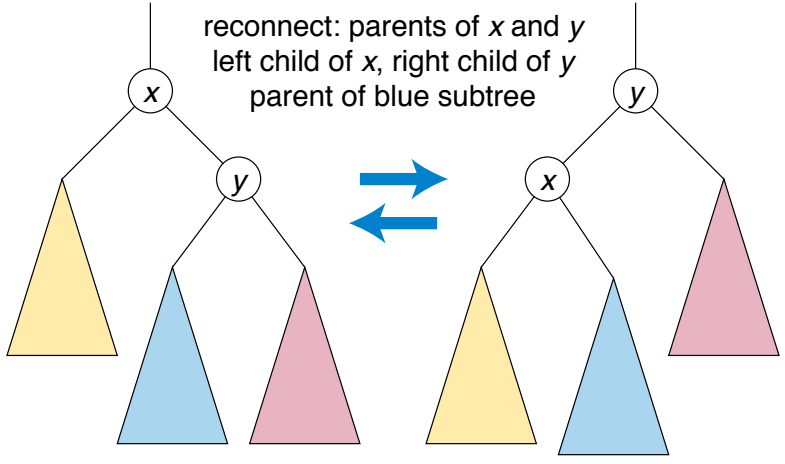
Can give worst case $O(\log n)$ with larger constant in height

Zipping

Cut into two trees along a path and then rejoin

Used in some recent structures [Tarjan et al. 2021; Gila et al. 2023]

Rotation


reconnect: parents of x and y
left child of x, right child of y
parent of blue subtree
x
y
y
x

Self-adjusting dynamic trees

The main idea

If a sequence of queries has repeating patterns or skewed item frequencies, we may be able to get faster than logarithmic queries

When an operation follows a search path to node x, rotate x to the root of the tree so that the next search for it will be fast

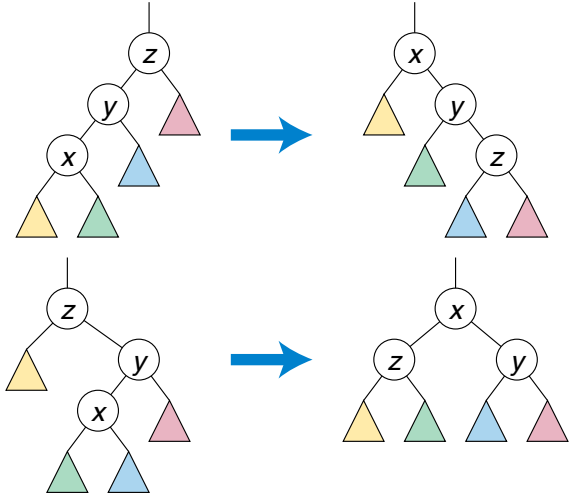
This operation is called "splaying"

[Sleator and Tarjan 1985]

Splay(x)

While x is not root:

If parent is root, rotate x and parent, else...



(and their mirror images)

Splay tree operations

Search

- Usual binary tree search (e.g. for successor)
- Splay the lowest interior node on the search path

Split into two subtrees at some key

- Splay the key
- Break link to its left child

Concatenate two subtrees

- Splay leftmost key in right subtree
- Add left subtree as its child

Add or remove item: split and concatenate

Simplifying assumptions for analysis

No insertions or deletions, only searches for members of an unchanging set of keys

- Deletion is similar to searching for the key and then not searching for it any more
- Insertion is similar to having a key in the initial set that you never searched for before
- Search for a missing key is similar to having another key where that key would be

We only need to analyze the time for a splay operation

- Actual time for search is bounded by time for splay

Amortized time for of weighted items

Suppose item x_i has weight $w_i > 0$, and let $W = \sum w_i$

Choose scale factor $s = \frac{1}{\min \text{ rank}}$ so that $s \cdot w_i \geq 1$ for all x_i

For a node x_i with subtree T_i (including x_i and all its descendants), define rank $r_i = \lfloor \log_2 s \cdot (\text{sum of weights of all nodes in } T_i) \rfloor$

Potential function Φ = sum of ranks of all nodes

Scale factor causes $\Phi \geq 0$ but doesn't affect $\Delta\Phi$ so we can mostly ignore it

Claim: The amortized time to splay x_i is $O(\log(W/w_i))$

Amortized analysis (sketch)

Main idea: look at the path from the previous root to x_i

Separate splay steps along path into two types:

- Steps where x and its grandparent z have different rank
- Steps where ranks of x and grandparent are equal

Rank at $x \geq \log_2 w_i$ and rank at root $\approx \log_2 W$ so number of different-rank steps is $O(\log(W/w_i))$

Each takes actual time $O(1)$ and can add $O(\Delta\text{rank})$ to Φ

There can be many equal-rank steps but each causes Φ to decrease
(if rank is equal, most weight in grandparent's subtree is below x, so rotation causes parent or grandparent to decrease in rank)

Decrease in Φ cancels actual time for these steps

Consequences for different choices of weights

$O(\log(W/w_i))$ time is valid regardless of what the weights w_i are!
We can set w_i however we like; algorithm doesn't know or care

Uniform weights:

Set all $w_i = 1$

$$W = \sum w_i = n$$

$$W/w_i = n$$

Amortized time is $O(\log n)$

Consequences for different choices of weights

$O(\log(W/w_i))$ time is valid regardless of what the weights w_i are!

We can set w_i however we like; algorithm doesn't know or care

Optimal weights:

Let T be an optimal static binary tree

Set $w_i = 1/3^h$ where h is height of same node in T

$$W = \sum w_i = \sum_h \frac{\#\text{ nodes at height h}}{3^h} \leq \sum_{h=0}^{\infty} \frac{2^h}{3^h} = 3$$

$$W/w_i \leq 3^{h+1}$$

Amortized time is $O(\log 3^{h+1}) = O(h)$

Splay trees are as good as static optimal tree!

Consequences for different choices of weights

$O(\log(W/w_i))$ time is valid regardless of what the weights w_i are!

We can set w_i however we like; algorithm doesn't know or care

Weights from probabilities:

Suppose each search item is chosen randomly, independently of previous search items, with probability p_i of choosing key i

Set $w_i = p_i$

$W = 1$

Expected amortized time is $O(\sum p_i \log 1/p_i)$

This is the entropy of the distribution!

Consequences for different choices of weights

$O(\log(W/w_i))$ time is valid regardless of what the weights w_i are!

We can set w_i however we like; algorithm doesn't know or care

Weights from ranks:

Set weight of ith most frequently accessed item to $1/i^2$

$$W = \sum_{i=1}^{n} \frac{1}{i^2} \leq \sum_{i=1}^{\infty} \frac{1}{i^2} = \frac{\pi^2}{6}$$

$$\log W/w_i = O(\log i^2) = O(\log i)$$

Amortized time is $O(\log i)$

Consequences for different choices of weights

$O(\log(W/w_i))$ time is valid regardless of what the weights w_i are!

We can set w_i however we like; algorithm doesn't know or care

Weights from access times:

Set $w_i = 1/t_i^2$ where t_i = number of searches since last access

Weights are dynamic $\Rightarrow$ amortized analysis needs to include the change in potential caused by any change in weights

Weights change by increasing weight of (just-accessed) tree root, decreasing everything else $\Rightarrow$ change of weights cannot increase Φ

Amortized time is $O(\log t_i)$

Dynamic optimality

Competitive ratio

Question: How valuable is knowledge of the future?

Let A be any algorithm for handling a sequence S of dynamic requests, one at a time, without knowledge of future requests

Let OPT be an algorithm that can see the whole sequence of requests and then chooses optimally (somehow) what to do

Then the competitive ratio of A is

$$\max_S \frac{\text{cost of } A \text{ on sequence } S}{\text{cost of } OPT \text{ on sequence } S}$$

Dynamic optimality conjecture

Allow dynamic search trees to rearrange any contiguous subtree containing the root node, with cost per operation:

- Length of search paths for all operations, plus
- Sizes of all rearranged subtrees

Conjecture: There is a structure with competitive ratio $O(1)$

(I.e. it gets same O-notation as the best dynamic tree structure optimized for any specific input sequence)

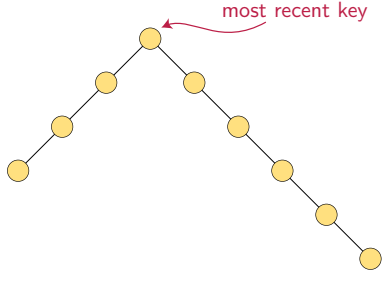
A simple example

For search sequences S where each search is previous ± 1:

Use a tree rooted at most recent search key, with two paths going left and right

For general searches this is a bad structure but for S it takes $O(1)$ per search (one rotation)

A competitive tree must also get $O(1)$ per search on S



Candidates for good competitive ratio

Splay trees

Conjectured to have competitive ratio $O(1)$

GreedyASS trees (next slides)

Conjectured to have competitive ratio $O(1)$

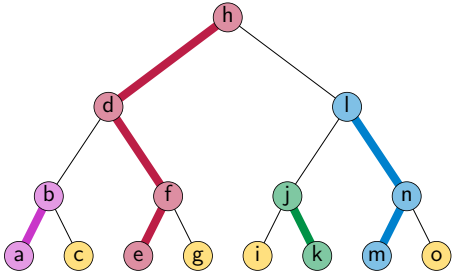
Tango trees (next slides)

Proven to have competitive ratio $O(\log \log n)$

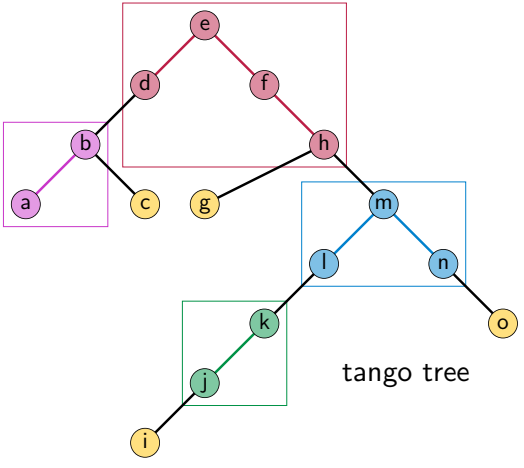
Tango trees

Consider a complete binary search tree on the keys ("reference tree")
+ "preferred paths" to most recently accessed descendant

Replace each preferred path by a balanced tree structure that can support cutting and linking operations (like a splay tree); it has $O(\log n)$ nodes so time $O(\log \log n)$



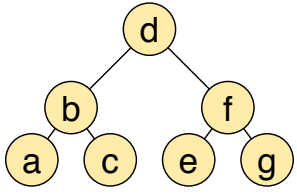
reference tree with paths to recently accessed descendants

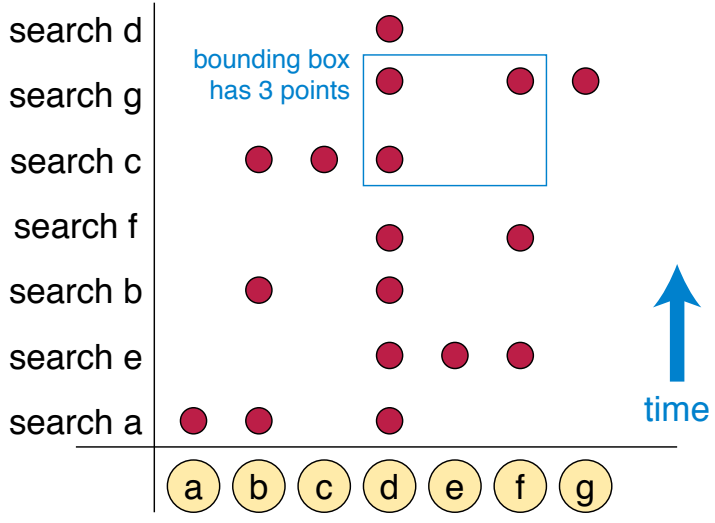


tango tree

The geometry of binary search trees

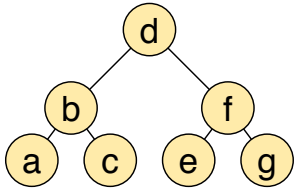
Given any (static or dynamic) binary search tree,
plot access to key i during operation j as a point (i, j)

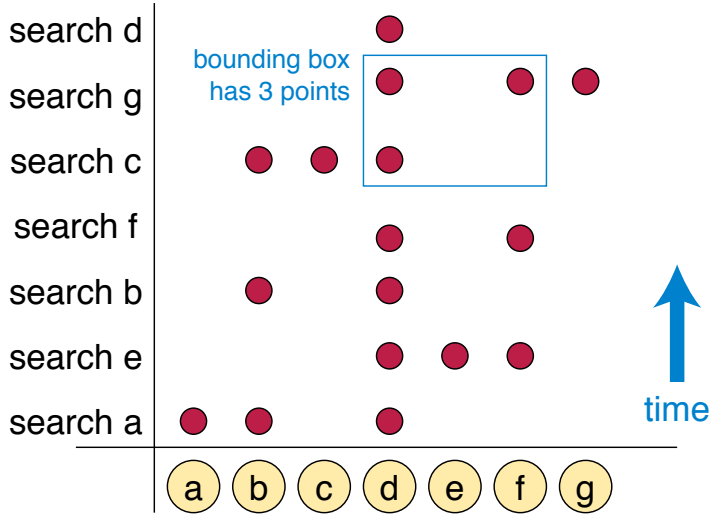




Arborially satisfied sets

Key property: Every two points not both on same row or column have a bounding box containing at least one more point

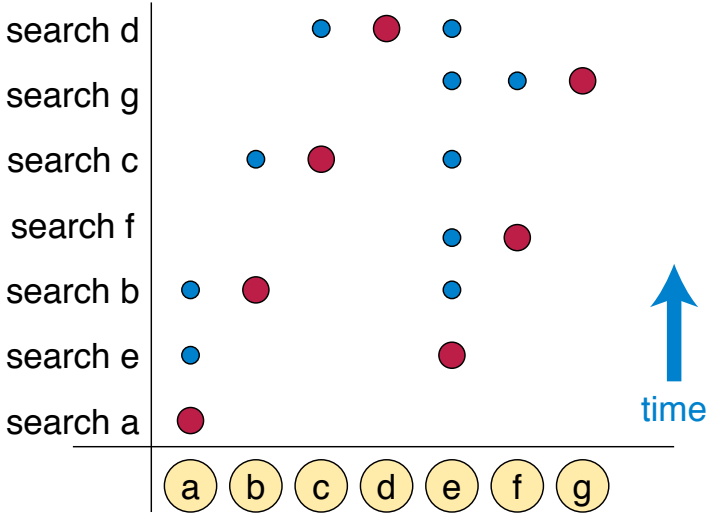




Interpretation: if search reaches x and later reaches y, it must pass through a common

Greedy arborially satisfied sets

In each row (bottom-up order) add the minimum number of extra points (blue) to make every bounding box have ≥ 3 points



Conjecture: uses $O(1) \times$ optimal # points

Can be turned into a dynamic tree algorithm (GreedyASS tree)

From geometry back to trees

Offline (if we know the future)

$\forall$ arborially satisfied set $\Rightarrow$ sequence of tree operations

Idea: Treap (a binary search tree that is heap-ordered by priorities), but replace random priority by next access time

Online

Can convert any row-by-row construction of arborially satisfied sets into a dynamic tree algorithm

Complicated

Greedy arborially satisfied set $\Rightarrow$ GreedyASS tree

[Demaine et al. 2009]

Summary

Summary

- Hashing is usually a better choice for exact searches, but binary searching is useful for finding nearest neighbors, function interpolation, etc.
- Similar search algorithms work both for static data in sorted arrays and explicit tree structures
- Balanced trees: maintain log-height while being updated
- Many variations of balanced trees
- Static versus dynamic optimality
- Construction of static binary search trees
- Splay trees and their amortized analysis
- Static optimality of splay trees
- Dynamic optimality conjecture and competitive ratios

References

Erik D. Demaine, Dion Harmon, John Iacono, Daniel Kane, and Mihai Puatracscu. The geometry of binary search trees. In Claire Mathieu, editor, *Proceedings of the Twentieth Annual ACM–SIAM Symposium on Discrete Algorithms, SODA 2009, New York, NY, USA, January 4–6, 2009*, pages 496–505. SIAM, 2009. doi: 10.1137/1.9781611973068.55.

Ofek Gila, Michael T. Goodrich, and Robert E. Tarjan. Zip-Zip Trees: Making Zip Trees More Balanced, Biased, Compact, or Persistent. In Pat Morin and Subhash Suri, editors, *Algorithms and Data Structures – 18th International Symposium, WADS 2023, Montreal, QC, Canada, July 31 – August 2, 2023, Proceedings*, volume 14079 of *Lecture Notes in Computer Science*, pages 474–492. Springer, 2023. doi: 10.1007/978-3-031-38906-1_31.

Daniel D. Sleator and Robert E. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985. doi: 10.1145/3828.3835.

Robert E. Tarjan, Caleb C. Levy, and Stephen Timmel. Zip Trees. *ACM Transactions on Algorithms*, 17(4):34:1–34:12, 2021. doi: 10.1145/3476830.