CS 261: Data Structures
Week 6: Binary search
Lecture 6a: Balanced trees

David Eppstein
University of California, Irvine

Spring Quarter, 2022

This work is licensed under a Creative Commons Attribution 4.0 International License
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

\( x? \)

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

\[
y = \frac{y_0(x - x_0) + y_1(x_1 - x)}{x_1 - x_0}
\]
Application: Check disjointness of line segments

For \( n \) line segments, find a crossing or report that none exists

Near \( x \)-coordinate of crossing, its two segments are adjacent in vertical ordering of sequences overlapping that coordinate

Process endpoints sorted by \( x \), looking up and down to find nearby segments that might cross its segment or each other

“Looking up and down” = binary search in a dynamic set
Binary search operations

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

- $\text{successor}(q)$: smallest key in $S$ that is $> q$
- $\text{predecessor}(q)$: 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 for static (unchanging) data

Data structure: array of sorted data values

define successor(q,array):
    first = 0 # first and last elements
    last = len(array) - 1 # in not-yet-tested subarr.
    s = infinity # best succ. found so far
    while first <= last:
        mid = (first + last)/2
        if q >= array(mid): # compare against middle
            first = mid + 1 # go to left subarray
        else:
            s = array[mid] # remember better successor
            last = mid - 1 # go to right subarray
    return s

Each step reduces subarray length by factor of two ⇒ \( \log_2 n \) steps
Binary search tree

Data structure that encodes the sequences of comparisons made by the static search

Each node stores
- Value that the query will be compared against
- Left child, what to do when comparison is <
- Right child, what to do when comparison is ≥

Inorder traversal of tree = sorted order of values (traverse left recursively, then root, then right recursively)
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 in same order, but works for any other binary tree with inorder = sorted order
Internal and external nodes

Variation sometimes used in some binary tree data structures

Internal: has data value, always has exactly two children
External: leaf node with no data value
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.)
Random search trees are balanced

Model of randomness: add keys in randomly permuted order
Each new key becomes a leaf of the previous tree
Not all trees are equally likely

When searching for query \( q \), any key \( i \) steps away in sorted order is only searched when random permutation places it earlier than closer keys \( \Rightarrow \) probability \( = \frac{1}{i} \)

Expected number of keys in search \( = \sum \frac{2}{i} \leq 2 \log n \)

Harder to prove: with high probability tree height is \( O(\log n) \)
Two 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
reconnect: parents of $x$ and $y$
left child of $x$, right child of $y$
parent of blue subtree
AVL trees

First known balanced tree structure
Also called height-balanced trees

Georgy Adelson-Velsky and Evgenii Landis, 1962

Each node stores height of its subtree

Constraint: left and right subtree heights must be within one of each other \( \Rightarrow \) height \( \leq \log_\phi n \) (golden ratio again)

Messy case analysis: \( O(\log n) \) rotations per update
Weight-balanced trees

Also called $BB[\alpha]$-trees

Jörg Nievergelt and Ed Reingold, 1973

Each node stores a number, the size of its subtree

Constraint: left and right subtrees at each node have sizes within a factor of $\alpha$ of each other $\Rightarrow$ height $\leq \log\frac{1}{(1-\alpha)} n = O(\log n)$

Original update scheme: rotations, works only for small $\alpha$

Simpler: rebuild unbalanced subtrees, amortized $O(\log n)$/update
(potential function: sum of unbalance amounts at each node)
Red–black trees

Leonidas J. Guibas and Robert Sedgewick, 1978

Each node stores one bit (its color, red or black)

Constraints: Root and children of red nodes are black; all root-leaf paths have equally many black nodes \( \Rightarrow \) height \( \leq 2 \log_2 n \)

Messy case analysis: \( O(\log n) \) time and \( O(1) \) rotations per update
WAVL trees

(WAVL = “weak AVL”, also called rank-balanced trees)

Haeupler, Sen & Tarjan, 2015

Each node stores a number, its rank

Constraints:

- External nodes have rank 0
- Internal nodes with two external children have rank 1
- Rank of parent is rank of child + 1 or + 2

With only insertions, same as AVL tree

In general, same properties as red-black tree with somewhat simpler case analysis
Each node stores a random real number, its priority

Constraint: heap-ordered by priority
⇒ Same as random tree with priority order = insertion order
⇒ Same expected time and high-probability height as random tree

Insert new value: place at a leaf (in correct position for search tree), choose a random priority, and then rotate upwards until heap-ordering constraint is met
Delete value: similar
Zip tree

Not to be confused with zippers in functional programming

Tarjan, Levy, and Timmel, 2019

Each node stores a random positive integer, its rank (probability $1/2^i$ of $rank = i$)

Constraint: max-heap-ordered by rank

Analysis: same as Treap

Update: “unzip” into two trees along search path, add or remove element, zip back together (fewer pointer changes than rotating)
Balance: $O(\log n)$ height while values added and removed

There are many ways of doing this

If your software library includes one, it’s probably fine

Otherwise, if you have to implement it, WAVL trees seem like a good choice (good worst-case performance, simpler case analysis, and $O(1)$ rotates/update)
B-trees
We saw that real-world cached memory hierarchies can make $k$-ary heaps more efficient than binary heaps.

The same is true for search trees: $b$-ary trees can be more efficient than binary trees.

- More complicated nodes $\Rightarrow$ more time per node
- Higher branching factor $\Rightarrow$ fewer nodes per search
- Each node = one cache miss
- For cached memory, cache misses can be much more expensive than processing time
Model of computation

We will assume:

Fast cached memory is small

\[ M << N \]

(similar assumption to streaming algorithms)

We can access slower main memory by moving consecutive blocks of \( B \) words into cache in a single transfer

Goal: Minimize the number of block transfers
In each node, we store:

- Up to \( b - 1 \) search keys (sorted)
- Up to \( b \) pointers to child nodes
  
  \( (b - 2 \) “between” each two keys, one before the first key, and one after the last key)

When a search for \( q \) reaches the node, we:

- Binary search for \( q \) in the keys at the node
- Follow the pointer to the resulting child node

For \( b \approx B/2 \) we can store a single node in a single cache block
B-tree

(Actually this is a $B^+$-tree, one of several variants)

Upper level nodes: $b$-ary nodes with keys and child pointers
Bottom level nodes hold up to $b - 1$ key-value pairs plus pointer to next bottom-level node in global sequence
The same key can appear at multiple levels
All bottom level nodes are at equal distances from root
Keeping the blocks full enough

Constraint: Each block has \( b/2 \leq \) number of children \( \leq b \)
(except we allow root to have fewer)

Equivalently: \( b/2 - 1 \leq \) number of keys \( \leq b - 1 \)

Same constraint on number of keys on bottom level

\[ \Rightarrow \text{number of tree levels} = O(\log_b N) \]

\[ \Rightarrow \text{number of cache misses per search} = O(\log_B N) \]
Updates

To insert a key:

- Search to find bottom-level block where it should go
- Add or remove it to that block
- While some block is too full, split it and insert new key at parent level (recursively splitting as necessary)

To delete a key:

- Find its bottom-level block and remove it
- While some block is too empty:
  - If we can move a key from adjacent child of same parent and keep both blocks full enough, do so
  - Else merge block with adjacent child and remove key from parent (recursively moving keys or merging as necessary)
B-tree summary

Important practical data structure for data too big to fit into cache

Goal is to optimize cache misses rather than runtime

Tree with branching factor between $b/2$ and $b$
with $b$ chosen to make each node $\approx$ one cache block

Splits and merges adjacent leaf nodes to maintain balance