CS 261: Graduate Data Structures

Week 6: Binary search trees

David Eppstein
University of California, Irvine

Spring Quarter, 2021

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

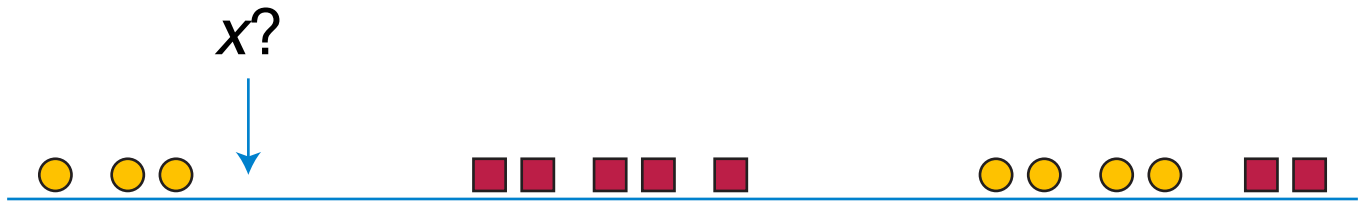
Hashing scrambles order $\Rightarrow$ not useful for nearby keys

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



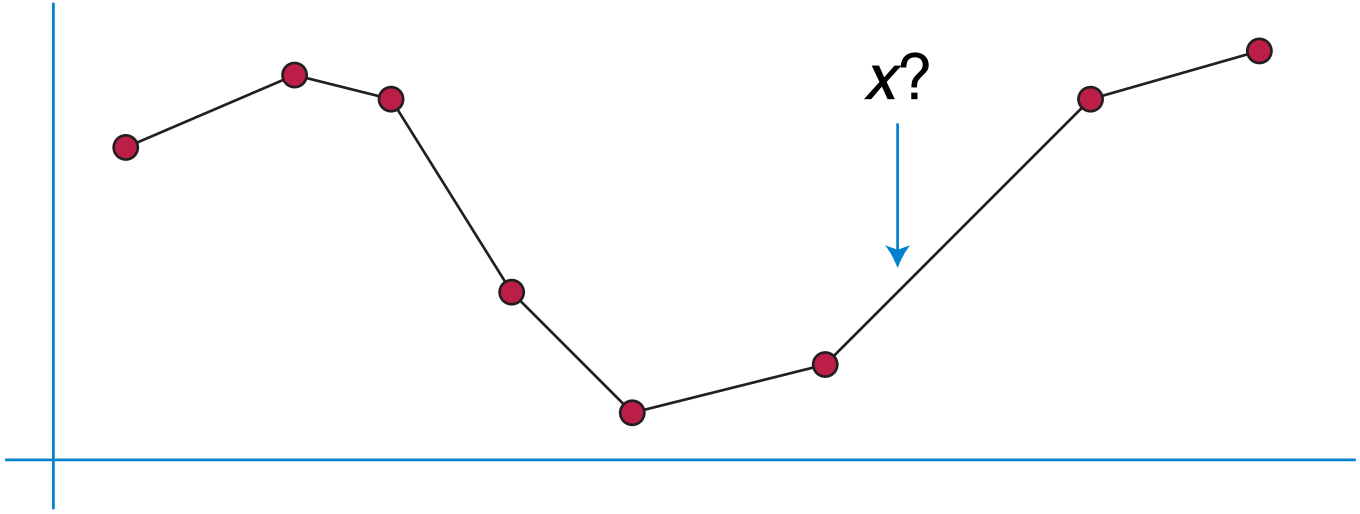
Binary search can be used for finding nearest value
but only when the data is only one-dimensional (unrealistic)

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 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
```

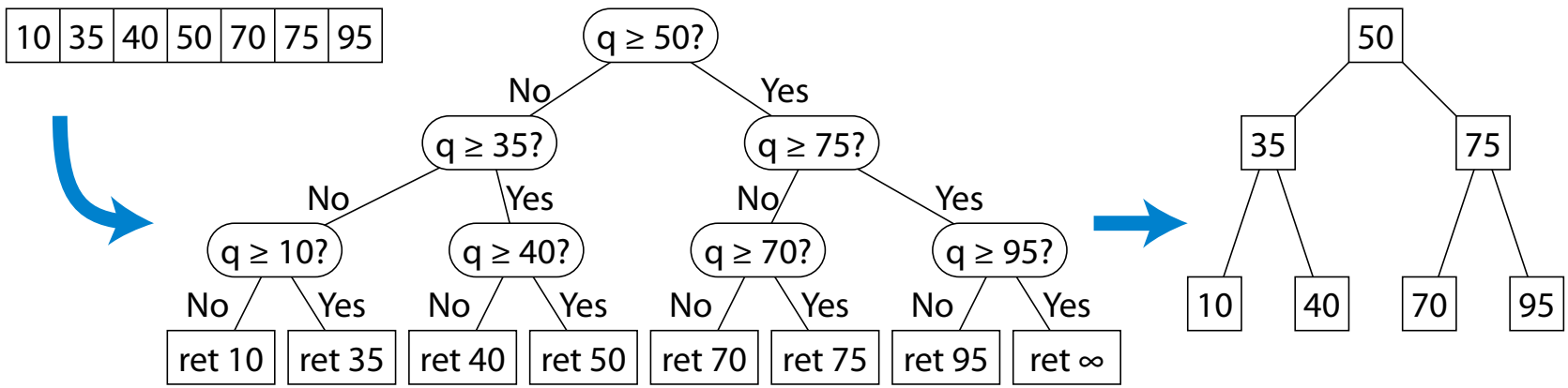
Each step reduces subarray length by factor of two $\Rightarrow$ $\log_2 n$ steps

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 $\geq$



Inorder traversal of tree = sorted order of values
(traverse left recursively, then root, then right recursively)

Successor in binary search trees

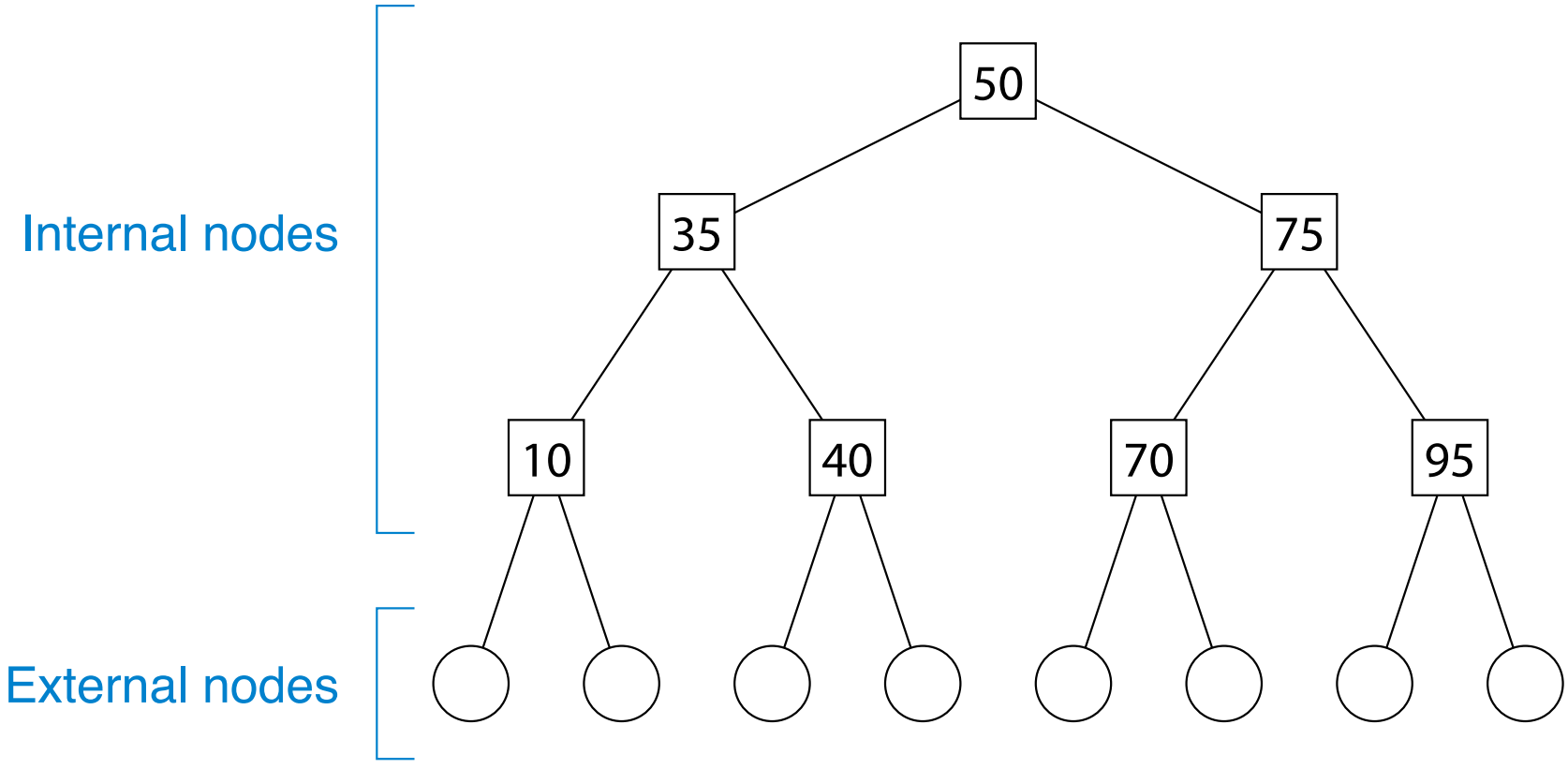
```
define successor(q,tree):
    s = infinity
    node = tree.root
    while node != null:
        if q >= node.value:
            node = node.right
        else:
            s = node.value
            node = node.left
    return s
```

For tree derived from static array, does same steps 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 $= 1/i$

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

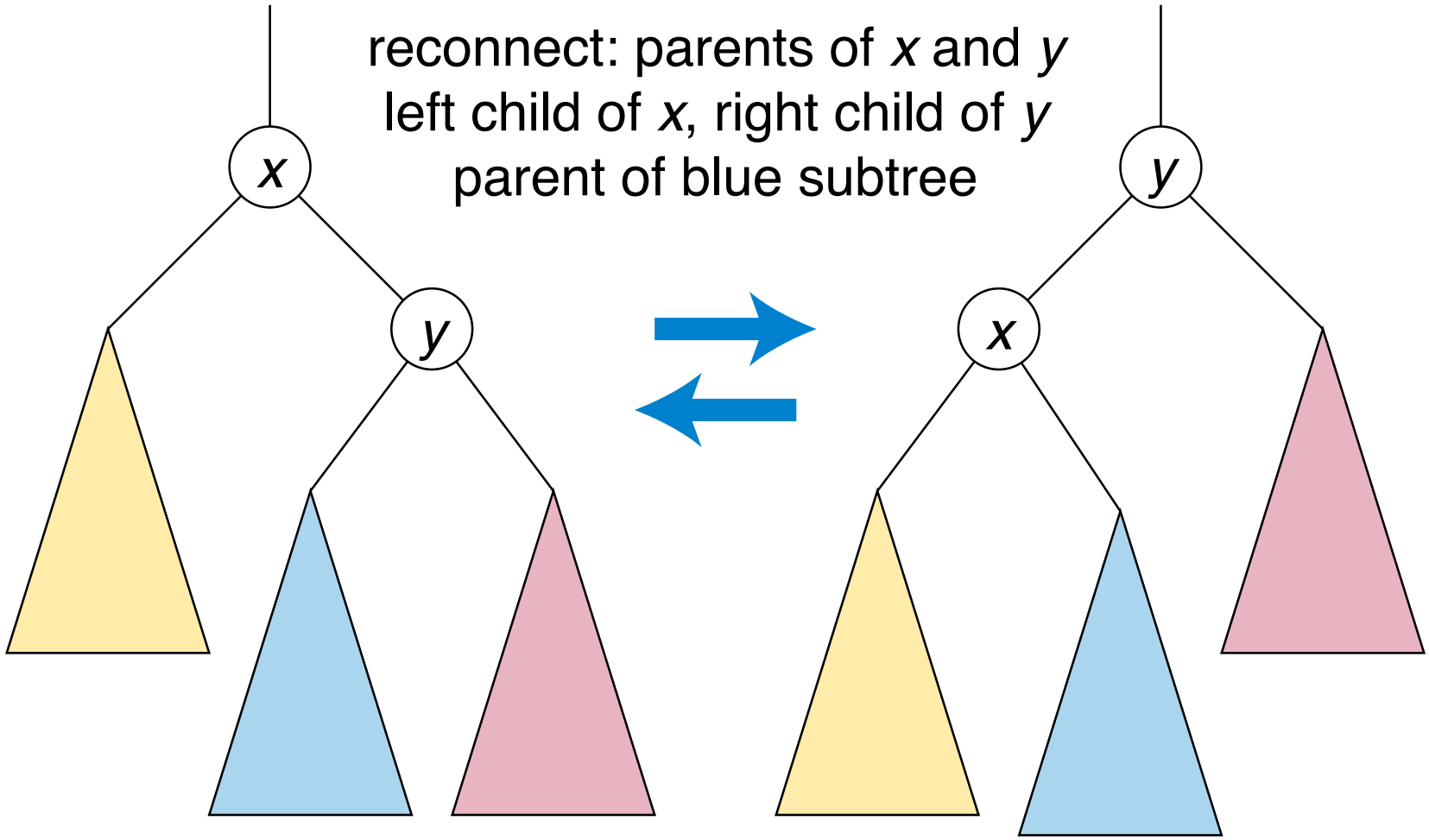
Usually amortized; can get very close to $\log_2 n$ height

Rotate

Local changes to structure that preserve search tree ordering

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

Rotation


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

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 $\leq \log_\varphi n$ (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 α of each other $\Rightarrow$ height $\leq \log_{1/(1-\alpha)} n = O(\log n)$

Original update scheme: rotations, works only for small α

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

Treap

Raimund Seidel and Cecilia R. Aragon, 1989

Each node stores a random real number, its priority

Constraint: heap-ordered by priority

$\Rightarrow$ Same as random tree with priority order = insertion order

$\Rightarrow$ 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 $\mathit{rank} = i$)

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)

Balanced tree summary

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

Intuition

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

Size $M << $ data set size 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

b-ary search tree: single node

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

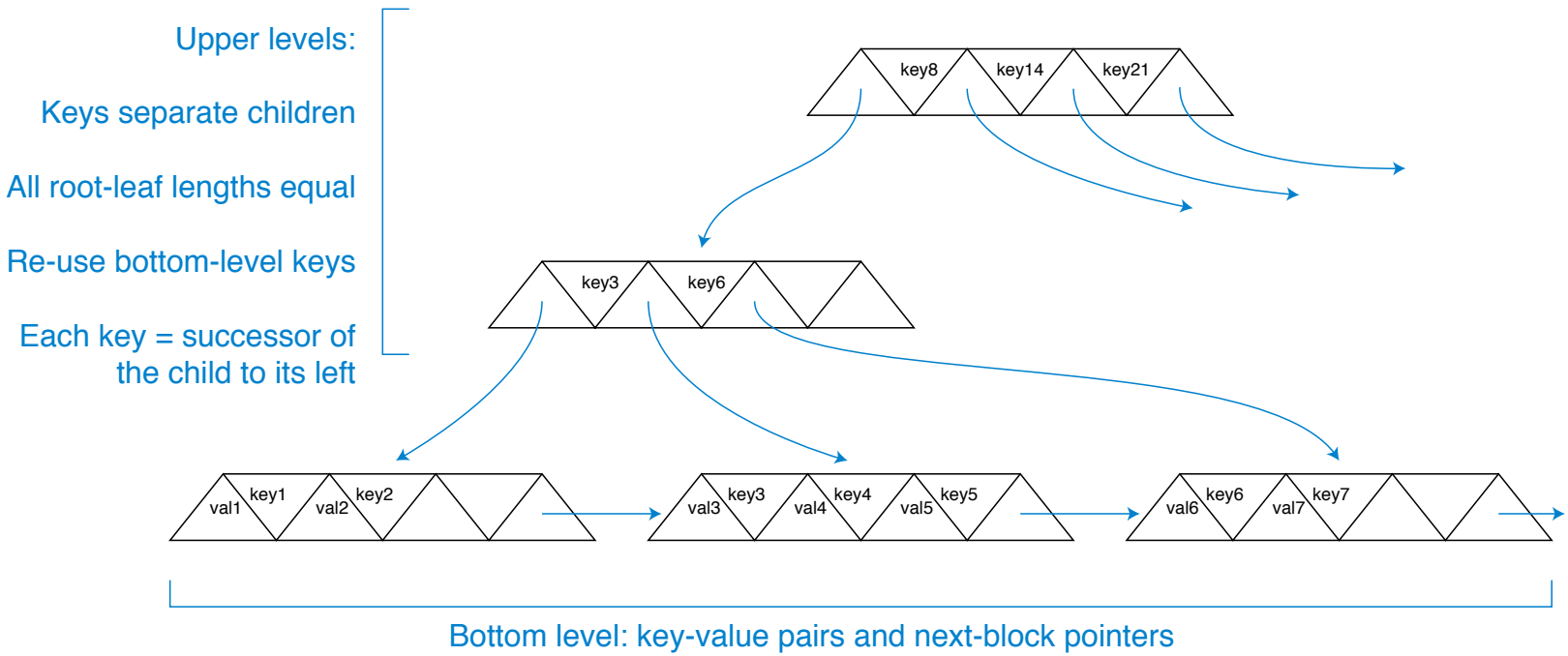
(Actually this is a B^+-tree, one of several variants)

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

$\Rightarrow$ number of tree levels $= O(\log_b N)$

$\Rightarrow$ 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

Optimal binary search trees

Static optimality

Suppose we know the frequencies p_i
of each search outcome (external node x_i)

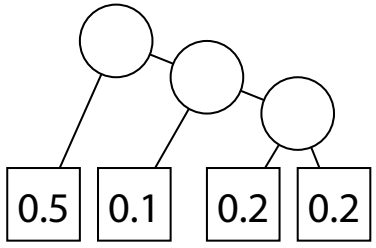
Then quality of a tree = average length of search path

$$= \sum_i p_i \times (\text{length of path to } x_i)$$

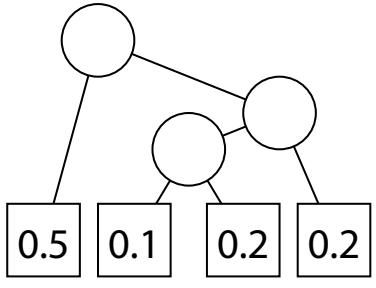
Uniformly balanced tree might not have minimum average length!

Example

With external node frequencies 0.5, 0.1, 0.2, 0.2:

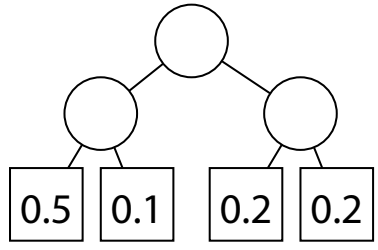


average height = 1.9
(1 x 0.5 + 2 x 0.1 + 3 x 0.2 + 3 x 0.2)

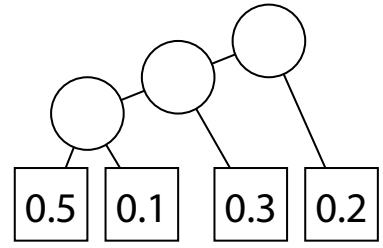


average height = 1.8
(1 x 0.5 + 3 x 0.1 + 3 x 0.2 + 2 x 0.2)

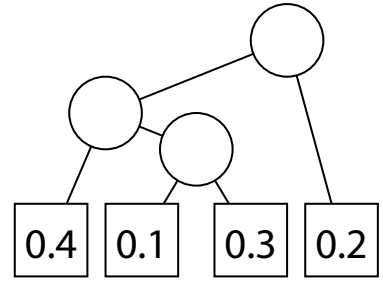
Optimal!



average height = 2
(2 x 0.5 + 2 x 0.1 + 2 x 0.3 + 2 x 0.1)



average height = 2.4
(3 x 0.5 + 3 x 0.1 + 2 x 0.2 + 1 x 0.2)



average height = 2.1
(2 x 0.5 + 3 x 0.1 + 3 x 0.2 + 1 x 0.2)

Dynamic program for optimal trees

For each subarray, in order by length:

- For each partition into two smaller subarrays:
  - Height $= 1 +$ weighted average of subarray heights
- Choose partition giving smallest height
- Remember its height for later lookup

Optimal tree is given by best partition for full array,
and by recursive optimal choices for each subarray

Time for naive implementation: $O(n^3)$

Improved by Knuth 1971 to $O(n^2)$

Garsia–Wachs algorithm for optimal trees

Adriano Garsia and Michelle L. Wachs, 1977,
simplifying T. C. Hu and A. C. Tucker, 1971

Very rough sketch of algorithm:

- Add frequency values $+\infty$ at both ends of the sequence
- Use a dynamic balanced binary tree to implement a greedy algorithm that repeatedly finds the first consecutive triple of frequencies x, y, z with $x \leq z$, replaces x and y with $x + y$, and moves replacement earlier in the sequence
  (after rightmost earlier value that is $\geq x + y$)
- The tree formed by these replacements has optimal path lengths but is not a binary search tree (leaves are out of order); find a binary search tree with the same path lengths

Time is $O(n \log n)$

Dynamic optimality

But now suppose:

- We are adding and removing items as well as searching
- Different items are "hot" at different times

Maybe we can do better than a static tree?

(Idea: rearrange tree to move currently-hot items closer to root)

Competitive ratio

Let A be an online algorithm
(one that doesn't have information about future operations)

Let B be an algorithm that performs the same computation optimally (somehow), using information about future

Let S denote any sequence of operations

Then the *competitive ratio* is

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

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 $O(1)$

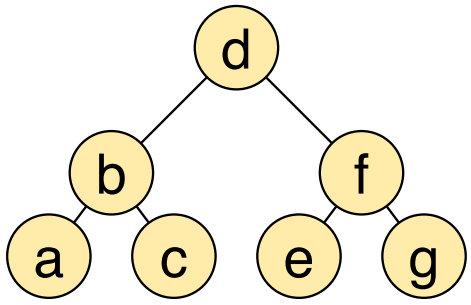
(I.e. it gets same O-notation as the best dynamic tree structure optimized for any specific input sequence)

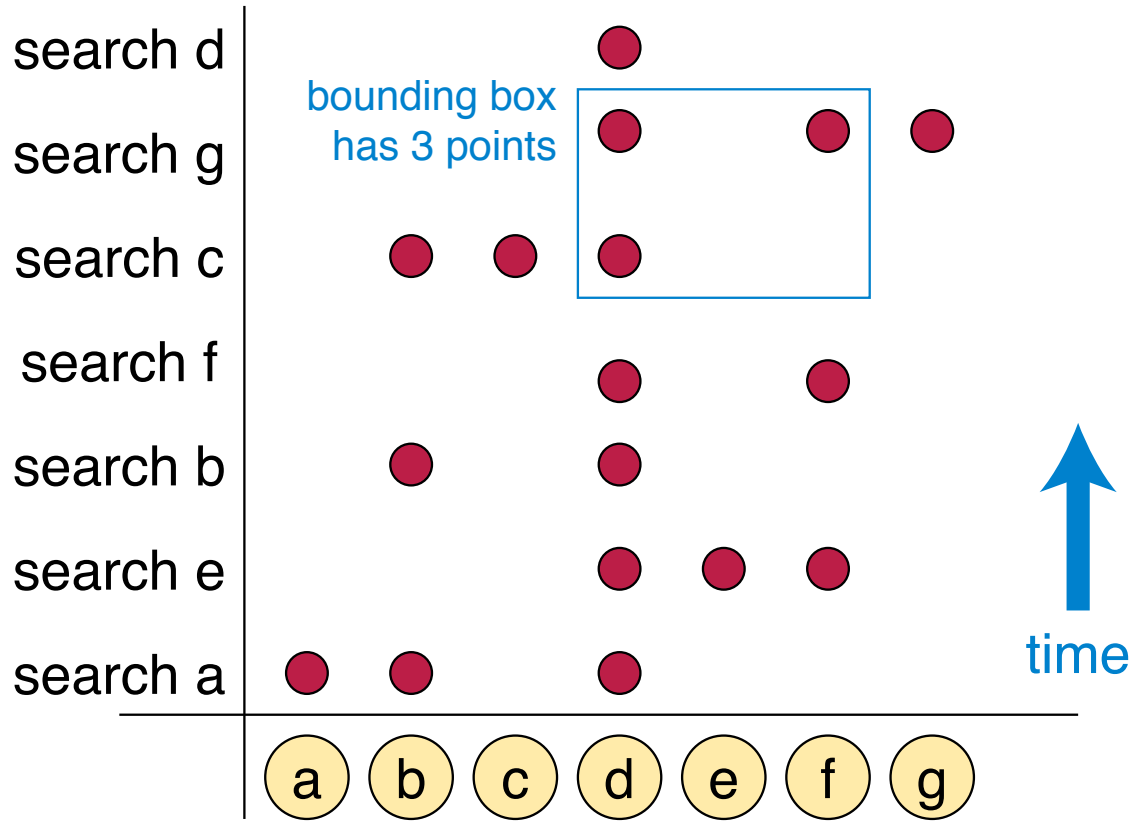
Two candidates for good tree structures:

- Splay trees (next section of notes)
- GreedyASS trees (sketched now)

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)

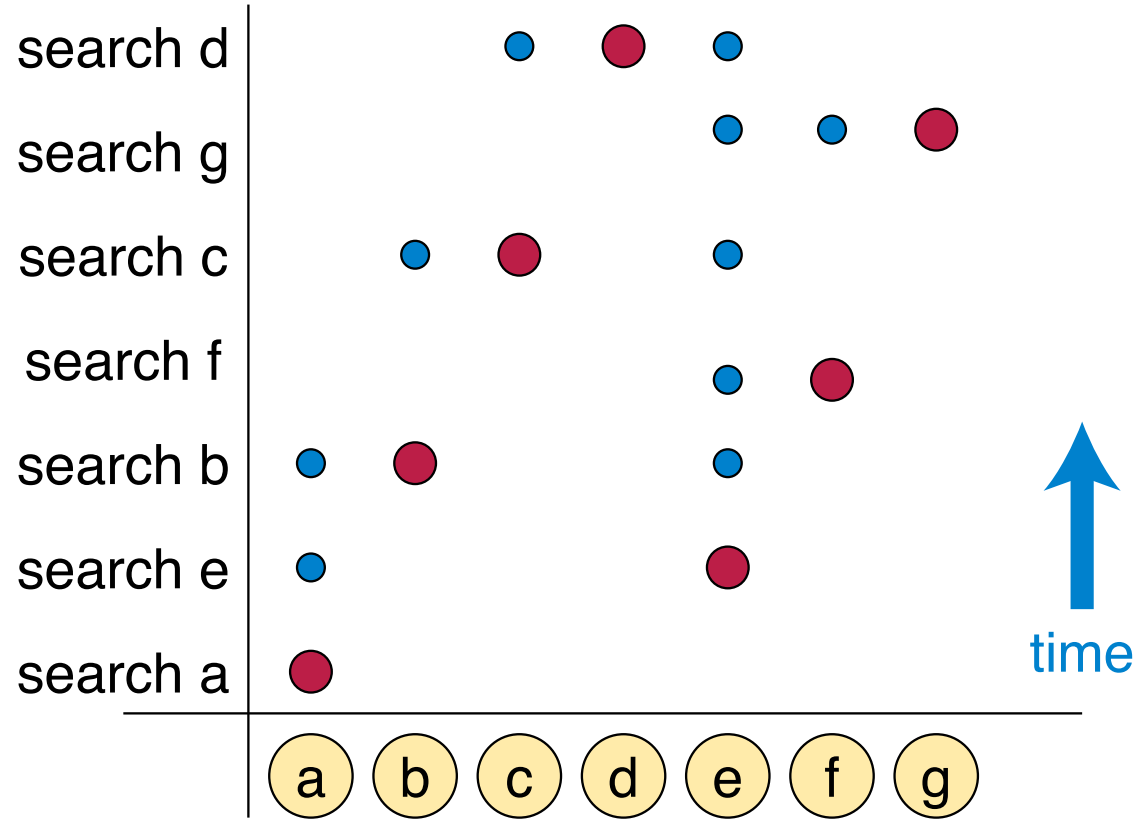




"Arborially satisfied set": Every two points not both on same row or column have a bounding box containing at least one more point

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 near-optimal number of total points

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

Demaine, Harmon, Iacono, Kane, and Pătraşcu, 2009

Splay trees

The main idea

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

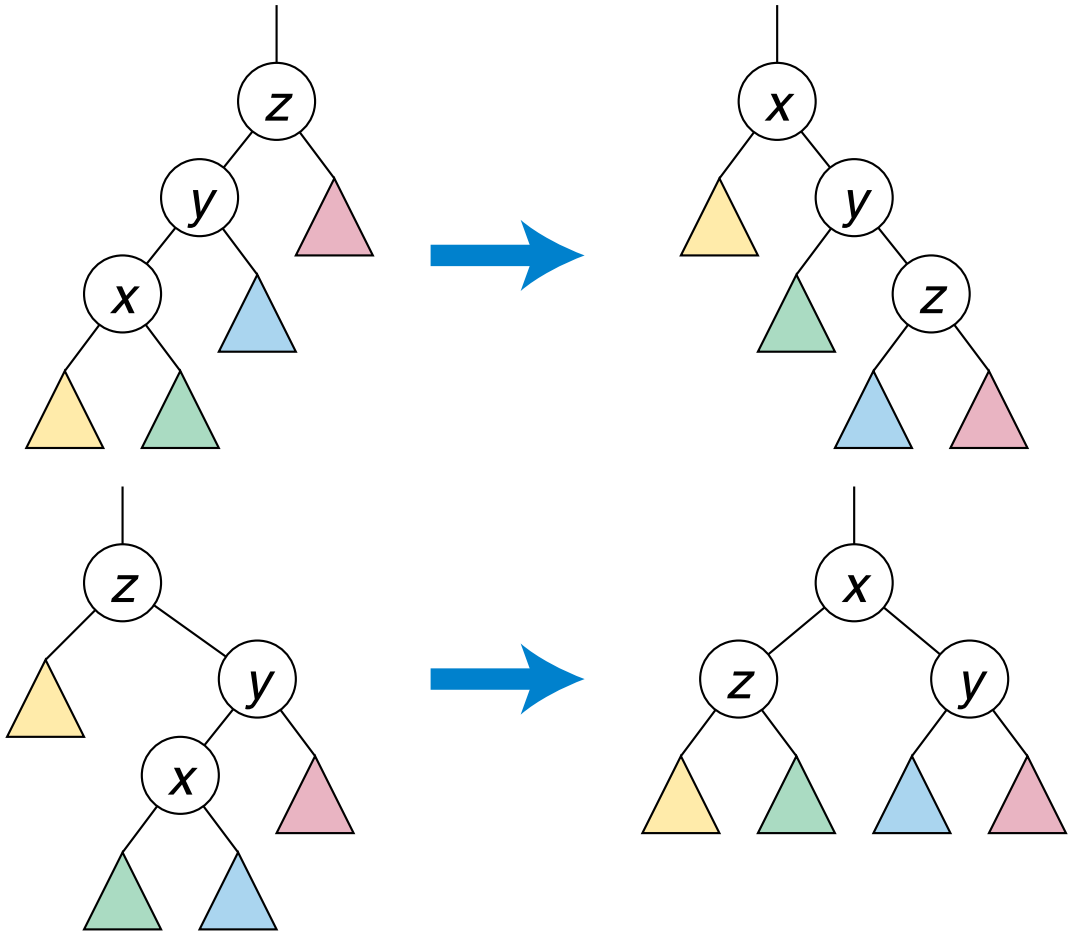
This operation is called "splaying"

Daniel Sleator and Robert 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 on 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

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 $W = \sum w_i$

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

Potential function Φ = sum of ranks of all nodes

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 $O(1)$ 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

Same analysis is valid regardless of what the weights w_i are!

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

If we set all $w_i = 1 \Rightarrow$ amortized time is $O(\log n)$

For any static binary search tree T, with $w_i = 1/3^j$,
where j is the height of i in $T \Rightarrow$ sum of weights is $O(1)$
$\Rightarrow$ amortized time is $O(\text{height in } T)$

Splay trees are as good as static optimal tree!

For search items drawn randomly with probability p_i,
set $w_i = p_i \Rightarrow$ expected amortized time is
$O(\sum p_i \log 1/p_i) = O(\text{entropy})$

Suppose x_i denotes the ith most frequently accessed item
Set $w_i = 1/i^2 \Rightarrow$ sum of weights is $O(1) \Rightarrow$ time is $O(\log i)$

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
- Dynamic optimality conjecture and competitive ratios
- Splay trees and their amortized analysis
- Static optimality of splay trees