CSE 830:
Design and Theory of Algorithms

Quicksort!

Dr. Charles Ofria

Quicksort

Although mergesort is O(n log n), it is difficult to implement on arrays since we need space to merge. In practice, *Quicksort* is the fastest sorting algorithm.

Quicksort chooses a pivot point and partitions the other numbers before or after the pivot in sorted order, then recurses.

Example: Pivot about 10

17 12 6 23 19 8 5 10 - before

6 8 5 10 17 12 23 19 - after

The pivot point is now in the correctly sorted position, and all other numbers are in the relative correct position, before or after.

Quicksort Walkthrough

17 12 6 23 19 8 5 10

6 8 5 10 17 12 23 19

5 6 8 17 12 19 23

6 8 12 17 23

6 17

5 6 8 10 12 17 19 23

Pseudocode

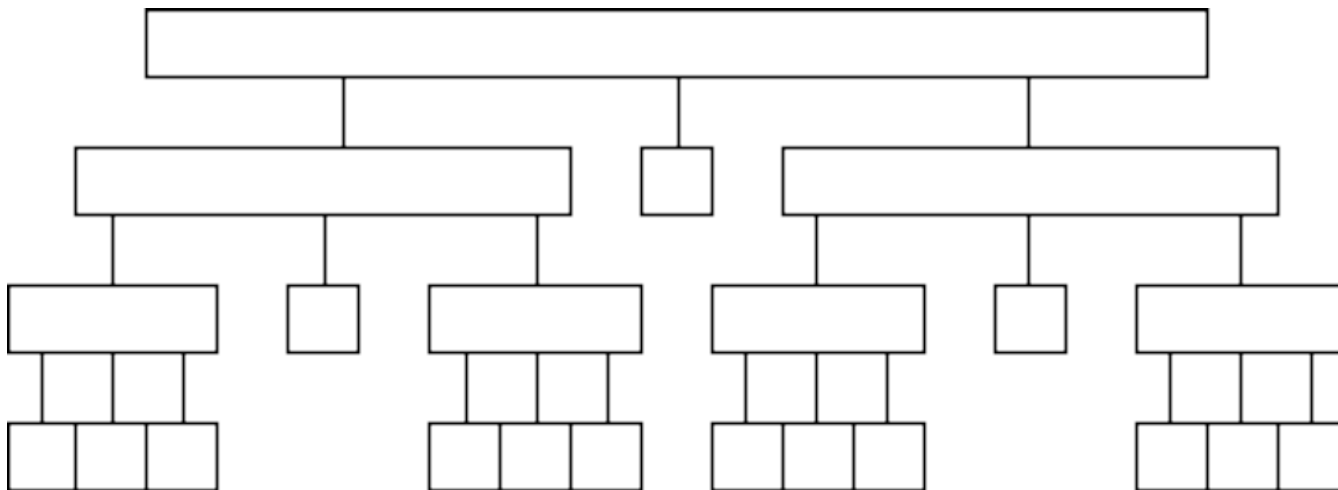
```
Sort(A) {
  Quicksort(A,1,n);
}

Quicksort(A, low, high) {
  if (low < high) {
    pivotLocation = Partition(A,low,high);
    Quicksort(A,low, pivotLocation - 1);
    Quicksort(A, pivotLocation+1, high);
  }
}
```

Pseudocode

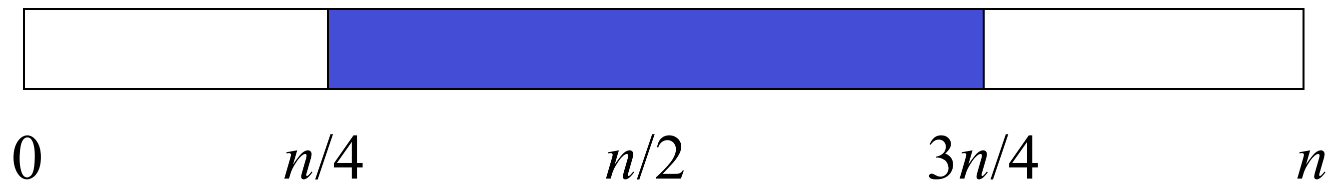
```
Partition(A,low,high) {
   pivot = A[low];
   leftwall = low;
   for i = low+1 to high {
      if (A[i] < pivot) then {
         leftwall = leftwall+1;
         swap(A[i],A[leftwall]);
      }
   }
   swap(A[low],A[leftwall]);
   return leftwall;
}
```

Best Case for Quicksort



Worst Case for Quicksort

Intuition: The Average Case



Anywhere in the middle half is a *decent* partition

$(3/4)^h n = 1 \quad \Rightarrow \quad n = (4/3)^h$

$\log(n) = h \log(4/3)$

$h = \log(n) / \log(4/3) < 2 \log(n)$

What have we shown?

At most $2\log(n)$ *decent* partitions suffices to sort an array of n elements.

But if we just take arbitrary pivot points, how often will they, in fact, be decent?

Since any number ranked between $n/4$ and $3n/4$ would make a decent pivot, we get one half the time on average.

Therefore, on average we will need $2 \times 2\log(n) = 4\log(n)$ partitions to guarantee sorting.

Quicksort in the real world…

Average Case Analysis

$$T(n) = \sum_{p=1}^{n} \frac{1}{n}(T(p-1) + T(n-p)) + (n-1)$$

$$\approx 1.38 n \log n$$

What *is* the worst case?

A B D F H J K

B D F H J K

D F H J K

F H J K

H J K

J K

K

Having the worst case occur is very bad because that is a likely case for many applications.

How can we pick a better pivot?

- Use the middle Element of the sub-array as the pivot.
- Use a random element in the array.
- Use the median element of (first, middle, last) to make sure to avoid any kind of pre-sorting.

What if your worst enemy supplies the data?

Randomization

Randomization of data means that you have to be very *unlucky* for your algorithm to run in worst case time (as opposed to ill prepared or unpopular.)

Randomization is a general tool to improve algorithms with bad worst-case but good average-case complexity. The worst-case is still there, but we almost certainly won't see it.

Is Quicksort really faster than Heapsort?

Since Quicksort is $\Theta(n \log n)$ and Selection Sort is $\Theta(n^2)$, there isn't any debate about which is faster.

How can we compare two $\Theta(n \log n)$ algorithms to know which one is faster?

Using the RAM model and the big-Oh notation, we can't!

If all of the algorithms are well implemented, Quicksort is at least 2-3 times faster than any of the others, but this only has to do with implementation details.

What do we know about our data?

- Is the data already partially sorted?
- Do we know the distribution of the keys?
- Are your keys very long or hard to compare?
- Is the range of possible keys very small?

Optimizing Quicksort

Using randomization: guarantees never to never have worst-case time due to bad data.

Median of three: Can be slightly faster than randomization for somewhat sorted data.

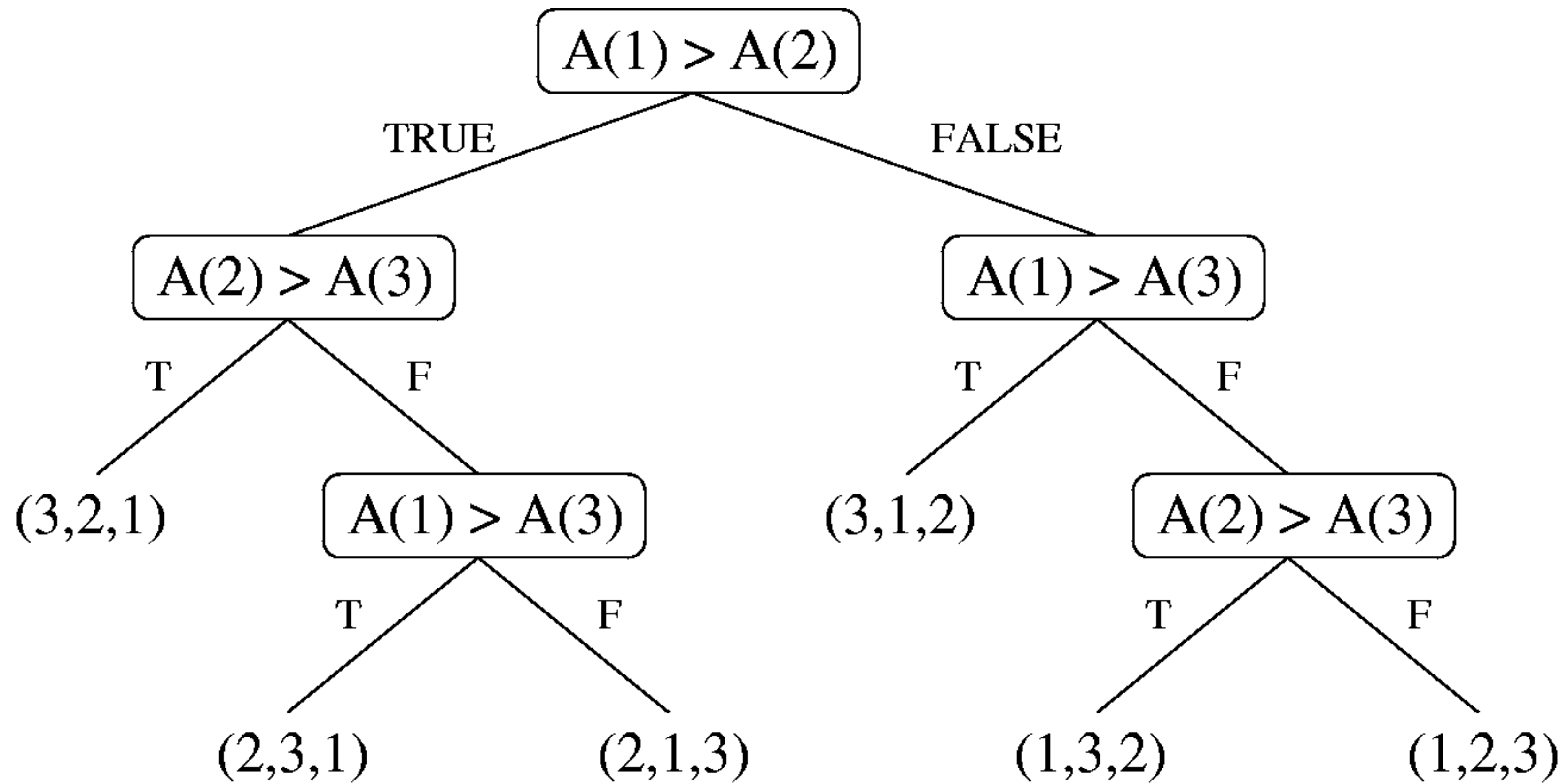
Leave small sub-arrays for insertion sort: Insertion sort can be faster, in practice, for small values of n.

Do the smaller partition first: minimize runtime memory.

Is Linear Sorting Possible?

Any comparison-based sorting program can be thought of as defining a decision tree of possible executions.

Example Decision Tree



How big is the decision tree?

Since different permutations of *n* elements requires a different sequence of steps to sort, there must be at least *n*! different paths from the root to leaves in the decision tree, ie. at least *n*! different leaves in the tree.

Since a binary tree of height *h* has at most 2^h leaves, we know that $n! \leq 2^h$, or $h \geq \log(n!)$

By inspection, $n! > (n/2)^{n/2}$ since the last n/2 elements of the product are greater than n/2. Thus $h > (n/2)\log(n/2)$

Non-comparison Based Sorting

All the sorting algorithms we have seen assume binary comparisons as the basic primitive, questions of the form "is x before y?".

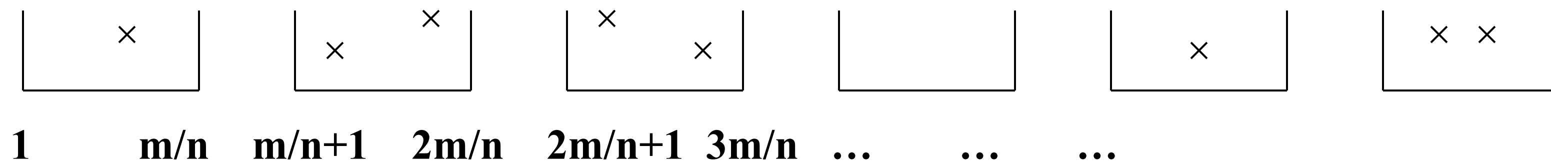
Suppose you were given a deck of playing cards to sort. Most likely you would set up 13 piles and put all cards with the same number in one pile.

A 2 3 4 5 6 7 8 9 10 J Q K

Bucketsort

Suppose we are sorting *n* numbers from *1* to *m*, where we know the numbers are approximately uniformly distributed.

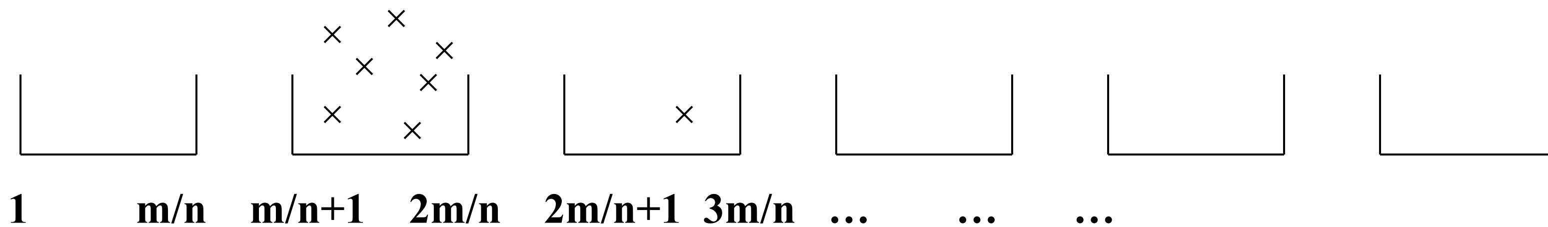
We can set up *n* buckets, each responsible for an interval of *m*/*n* numbers from *1* to *m*



Bucketsort

We can use bucketsort effectively whenever we understand the distribution of the data.

However, bad things happen when we assume the wrong distribution.



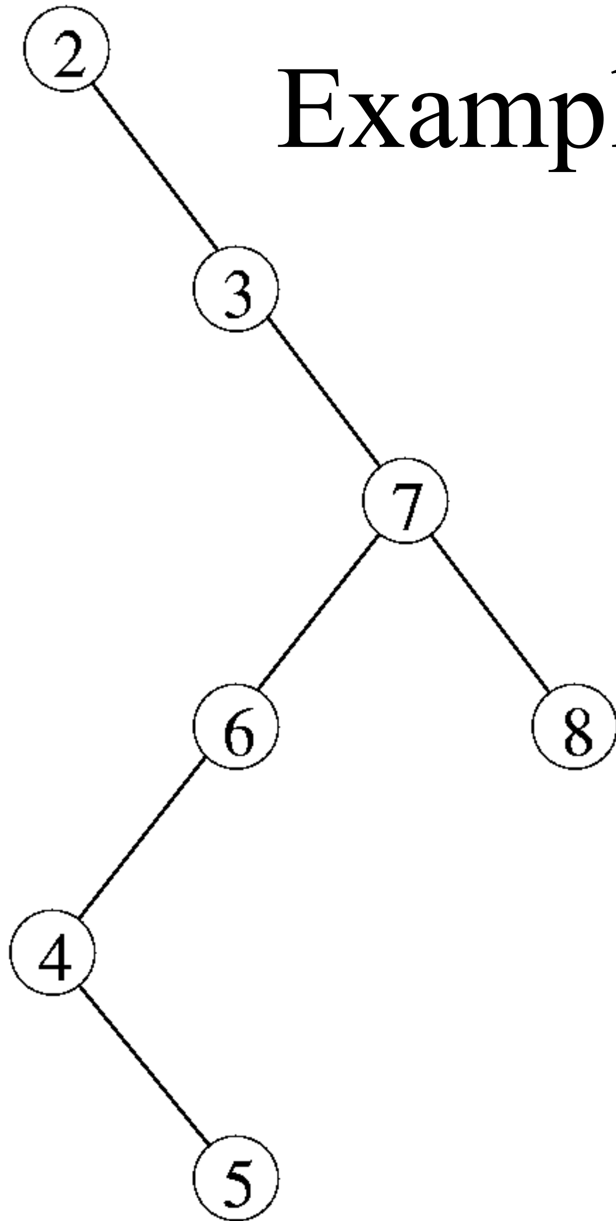
Real World Distributions

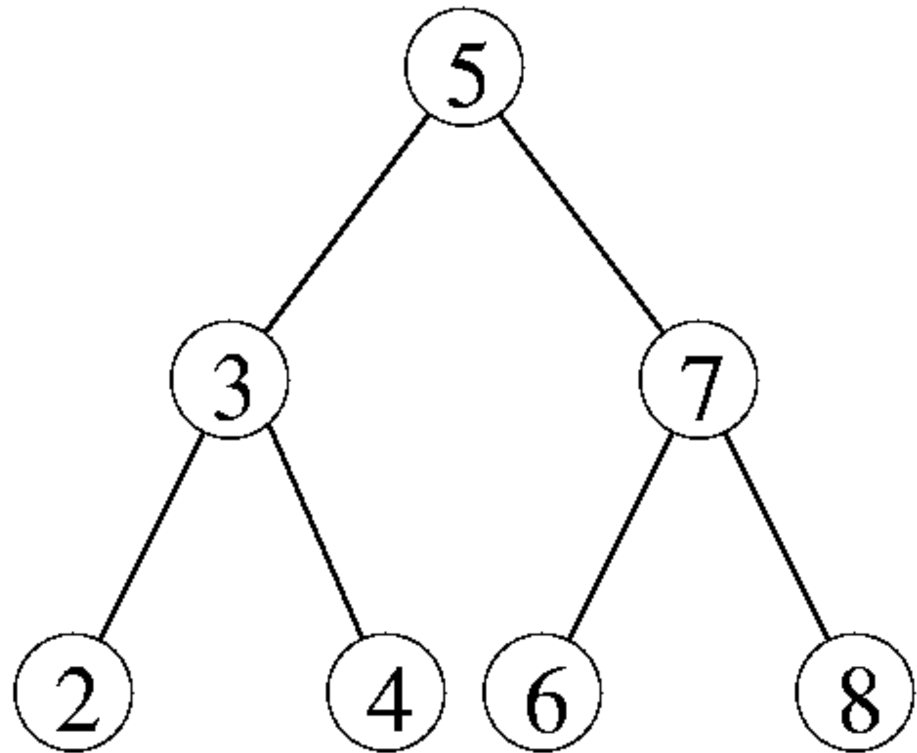
Consider the distribution of names in a telephone book.

- Will there be a lot of Ofria’s?
- Will there be a lot of Smith’s?
- Will there be a lot of Whitley’s?

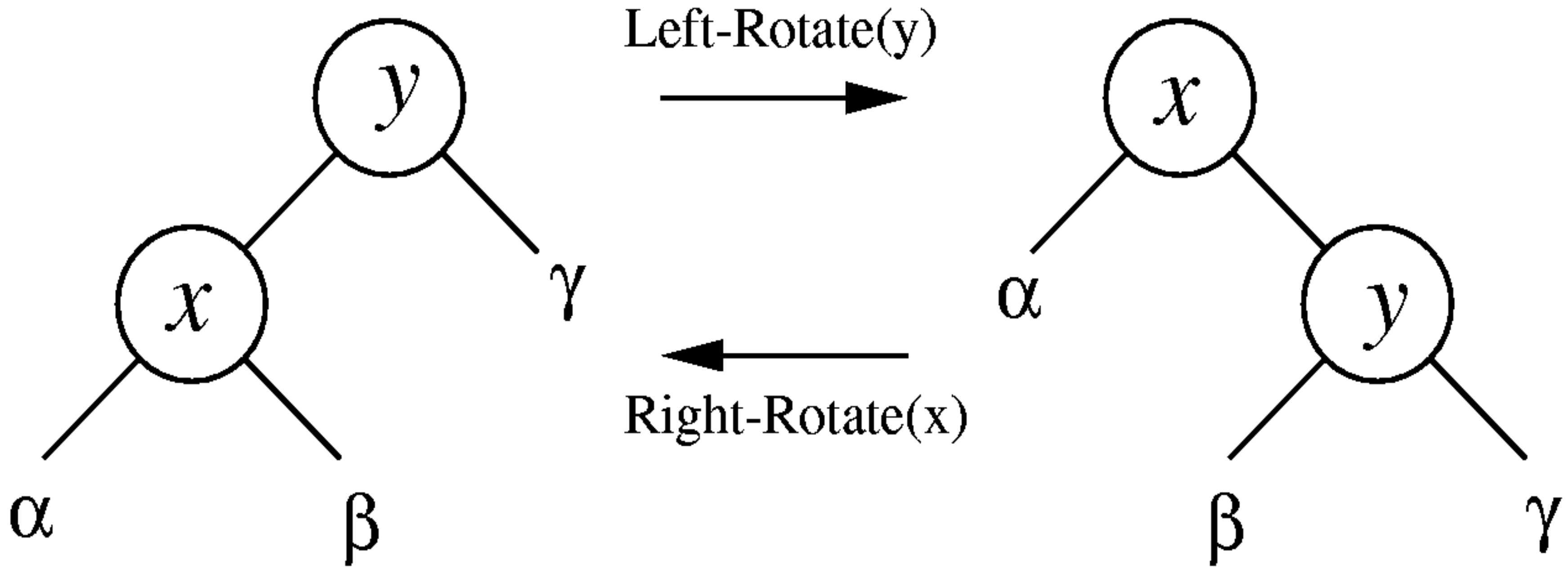
Make *sure* you understand your data, or use a good worst-case or randomized algorithm!

Example Search Trees





Manipulating Search Trees



Tree-Balancing Algorithms

- AVL Trees
- Splay Trees
- 2-3 Trees and 2-3-4 Trees
- Red-Black Trees

AVL Trees

The two sub-trees in an AVL tree differ in height by at most 1, and are in turn both AVL trees.

How can we be sure to maintain this property when inserting and deleting elements in the tree?

Does this guarantee us a “good” binary search tree?

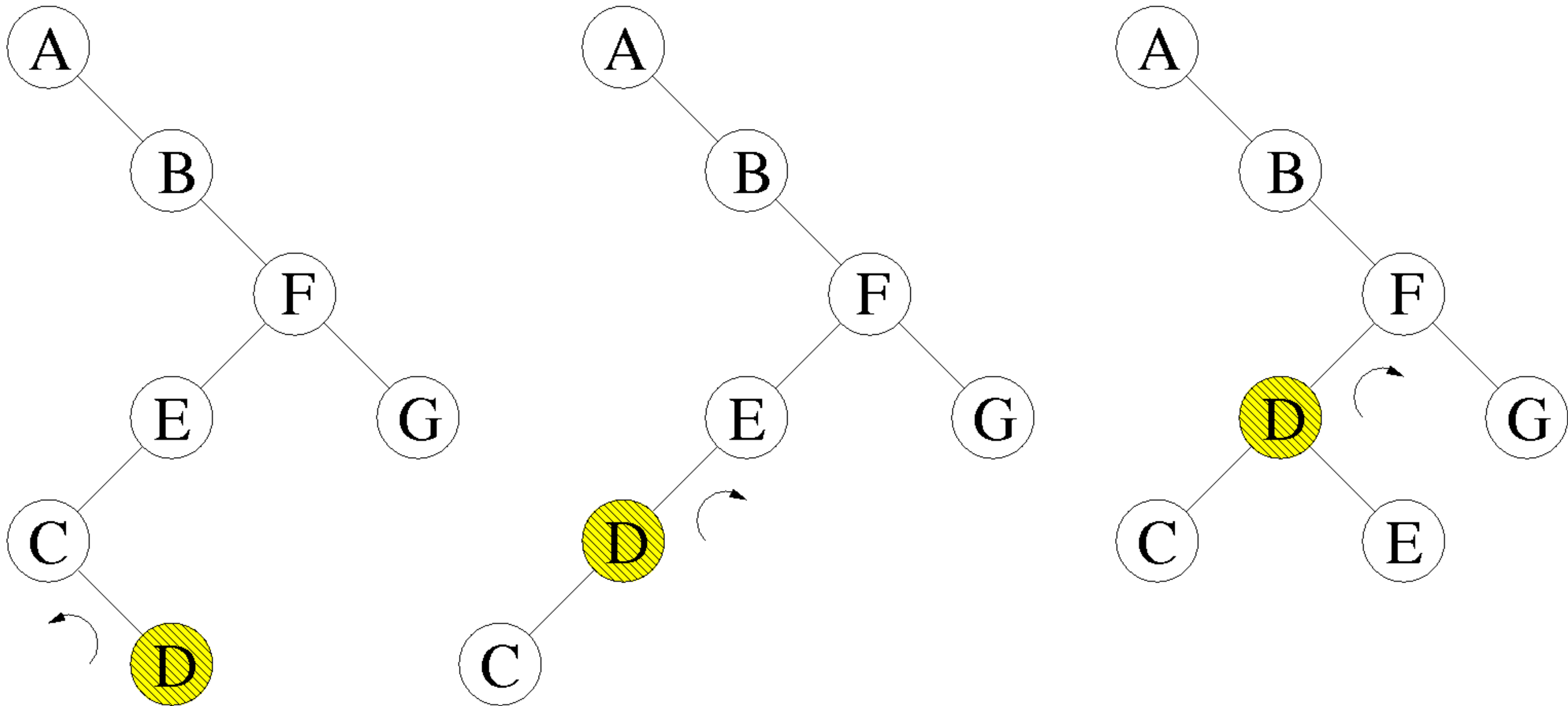
AVL Trees

Splay Trees

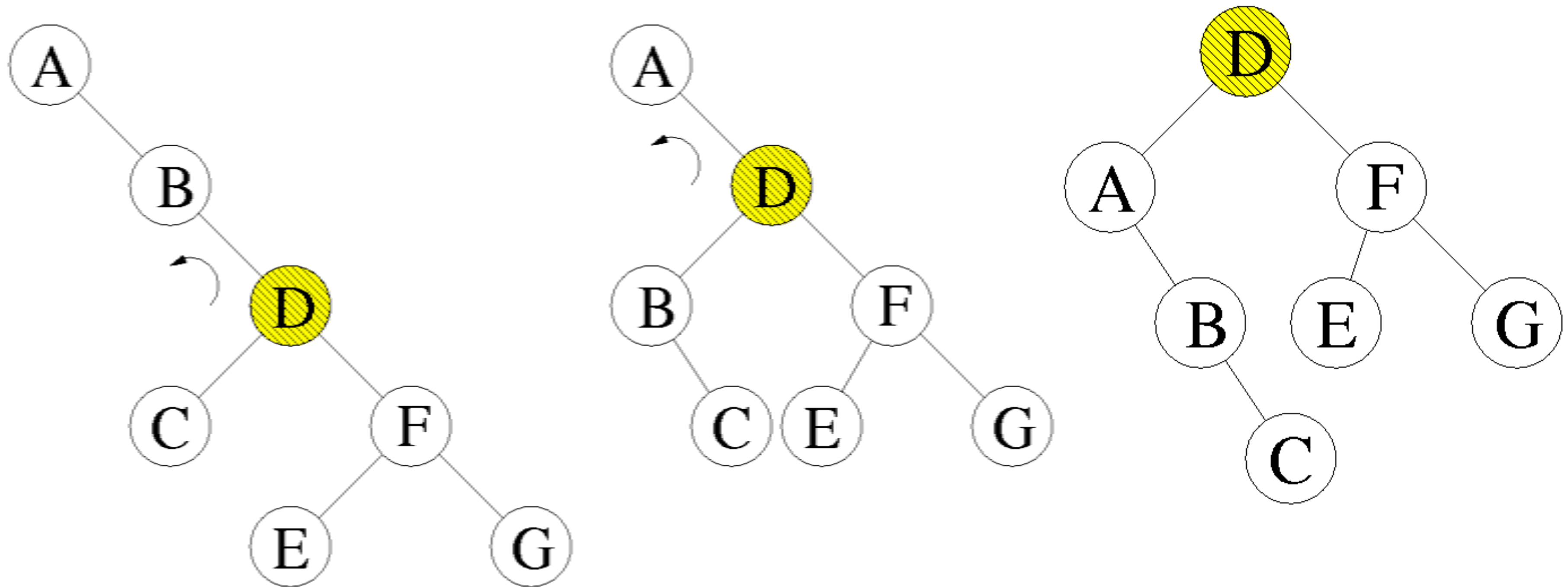
No adjustment is done in a splay tree when nodes are inserted or removed. All rotations occur within the *Search* function - the element being searched for is rotated to the root of the tree.

Initial searches in the tree may take $O(n)$ time, but they will rapidly reduce to $O(\log n)$

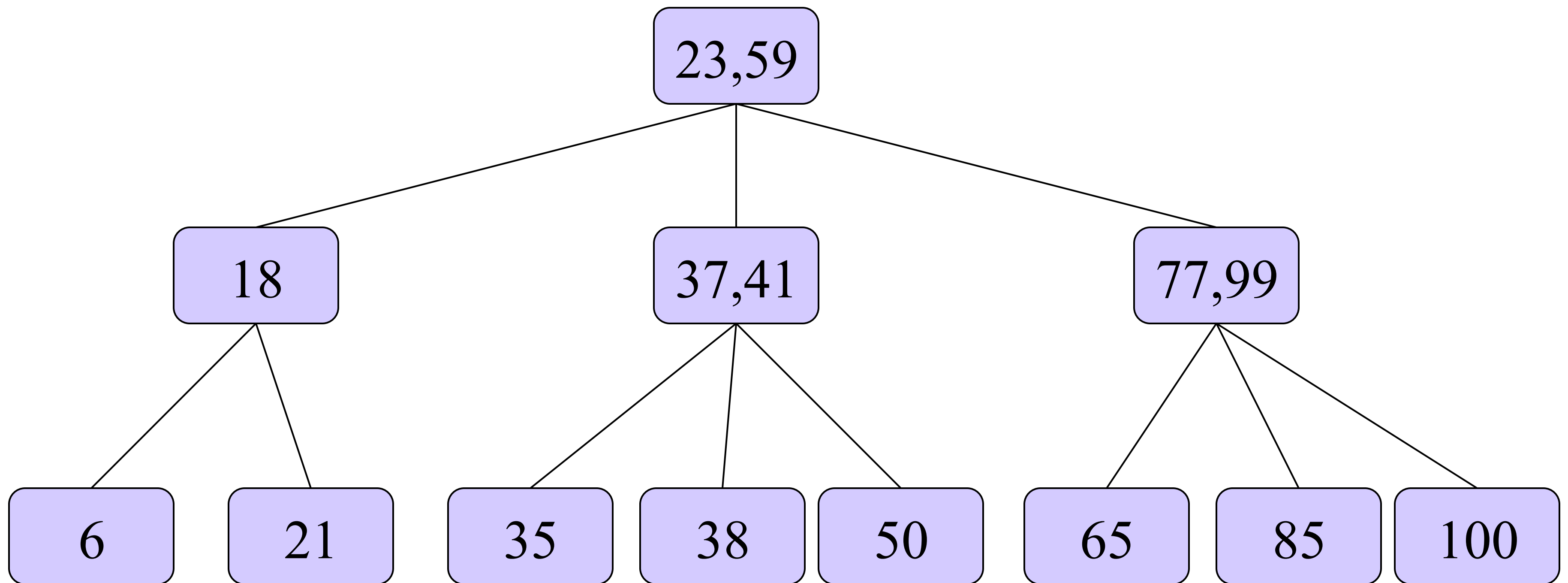
Splay Tree Example



Splay Tree Example



2-3 Trees

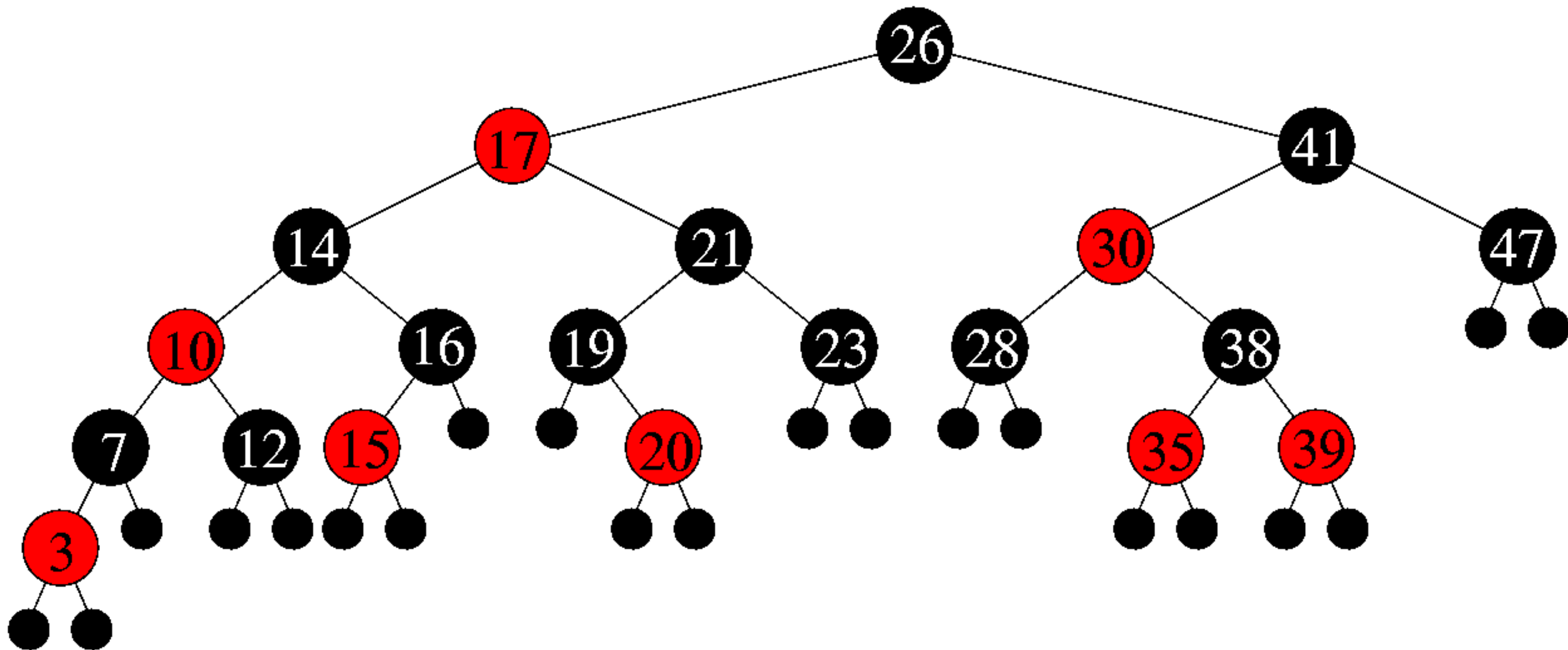


Red-Black Trees

- All nodes in the tree are either **red** or **black**.
- Every null-child is included and colored black.
- All red nodes must have two black children.
- Every path from the root to a leaf must have the same number of black nodes.

How balanced of a tree will this produce? How hard will it be to maintain?

Example Red-Black Tree



AVL-Tree as Red-Black Tree