Quicksort with Equal Keys

Sebastian Wild
wild@cs.uni-kl.de



based on joint work with
Martin Aumüller, Martin Dietzfelbinger,
Conrado Martínez, and Markus Nebel

Data Structures
and Advanced Models of
Computation on Big Data

Dagstuhl Seminar 16 101

Motivation

Practical Dimension:

- solve togetherness problem
  e. g. SQL's `GROUP BY` clause
- preprocessing to remove dups

Theoretical Dimension:

- many equals $\rightsquigarrow$ lower entropy

$\rightsquigarrow$ We should be able to sort faster!

Why Equal Keys?

Academic Dimension:

Open Problem of Sedgewick & Bentley:

Is Quicksort with median-of-k ***entropy-optimal*** *for* $k \to \infty$*?*

Personal Dimension:

- This is my tenth talk on Quicksort . . .
- Equal Keys on my wishlist since 2012

Motivation

Practical Dimension:

- solve togetherness problem
  e.g. SQL's `GROUP BY` clause
- preprocessing to remove dups

Theoretical Dimension:

- many equals $\rightsquigarrow$ lower entropy

$\rightsquigarrow$ We should be able to sort faster!

Why Equal Keys?

Academic Dimension:

Open Problem of Sedgewick & Bentley:

Is Quicksort with median-of-k ***entropy-optimal*** *for* $k \to \infty$*?*

Personal Dimension:

- This is my tenth talk on Quicksort ...
- Equal Keys on my wishlist since 2012

Motivation

Practical Dimension:

- solve togetherness problem
  e.g. SQL's `GROUP BY` clause
- preprocessing to remove dups

Theoretical Dimension:

- many equals $\rightsquigarrow$ lower entropy

$\rightsquigarrow$ We should be able to sort faster!

Why Equal Keys?

Academic Dimension:

Open Problem of Sedgewick & Bentley:

Is Quicksort with median-of-k ***entropy-optimal*** *for* $k \to \infty$?

Personal Dimension:

- This is my tenth talk on Quicksort ...
- Equal Keys on my wishlist since 2012

Motivation

Practical Dimension:

- solve togetherness problem
  e.g. SQL's `GROUP BY` clause
- preprocessing to remove dups

Theoretical Dimension:

- many equals $\rightsquigarrow$ lower entropy

$\rightsquigarrow$ We should be able to sort faster!

Why Equal Keys?

Academic Dimension:

Open Problem of Sedgewick & Bentley:

Is Quicksort with median-of-k ***entropy-optimal*** *for* $k \to \infty$*?*

Personal Dimension:

- This is my tenth talk on Quicksort ...
- Equal Keys on my wishlist since 2012

Motivation

Practical Dimension:

- solve togetherness problem
  e.g. SQL's `GROUP BY` clause
- preprocessing to remove dups

Theoretical Dimension:

- many equals $\rightsquigarrow$ lower entropy

$\rightsquigarrow$ We should be able to sort faster!

Why Equal Keys?

Academic Dimension:

Open Problem of Sedgewick & Bentley:

Is Quicksort with median-of-k ***entropy-optimal*** *for* $k \to \infty$*?*

Personal Dimension:

- This is my tenth talk on Quicksort ...
- Equal Keys on my wishlist since 2012

Previous Work

Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*

Previous Work

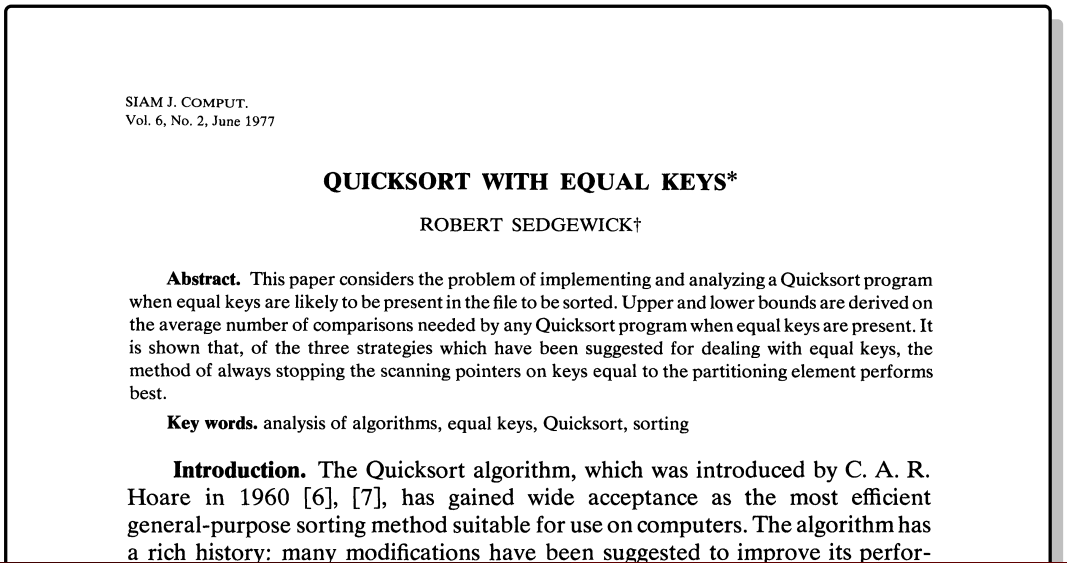
Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



SIAM J. COMPUT.
Vol. 6, No. 2, June 1977

QUICKSORT WITH EQUAL KEYS*

ROBERT SEDGEWICK†

Abstract. This paper considers the problem of implementing and analyzing a Quicksort program when equal keys are likely to be present in the file to be sorted. Upper and lower bounds are derived on the average number of comparisons needed by any Quicksort program when equal keys are present. It is shown that, of the three strategies which have been suggested for dealing with equal keys, the method of always stopping the scanning pointers on keys equal to the partitioning element performs best.

Key words. analysis of algorithms, equal keys, Quicksort, sorting

Introduction. The Quicksort algorithm, which was introduced by C. A. R. Hoare in 1960 [6], [7], has gained wide acceptance as the most efficient general-purpose sorting method suitable for use on computers. The algorithm has a rich history: many modifications have been suggested to improve its perfor-

Previous Work

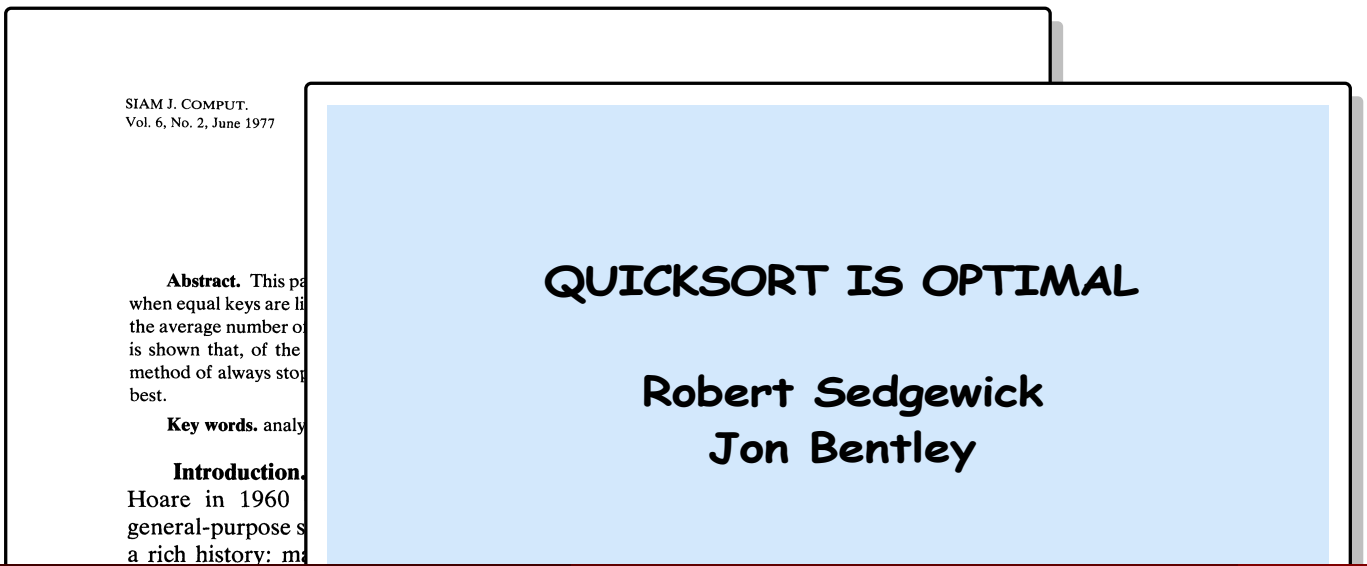
Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Previous Work

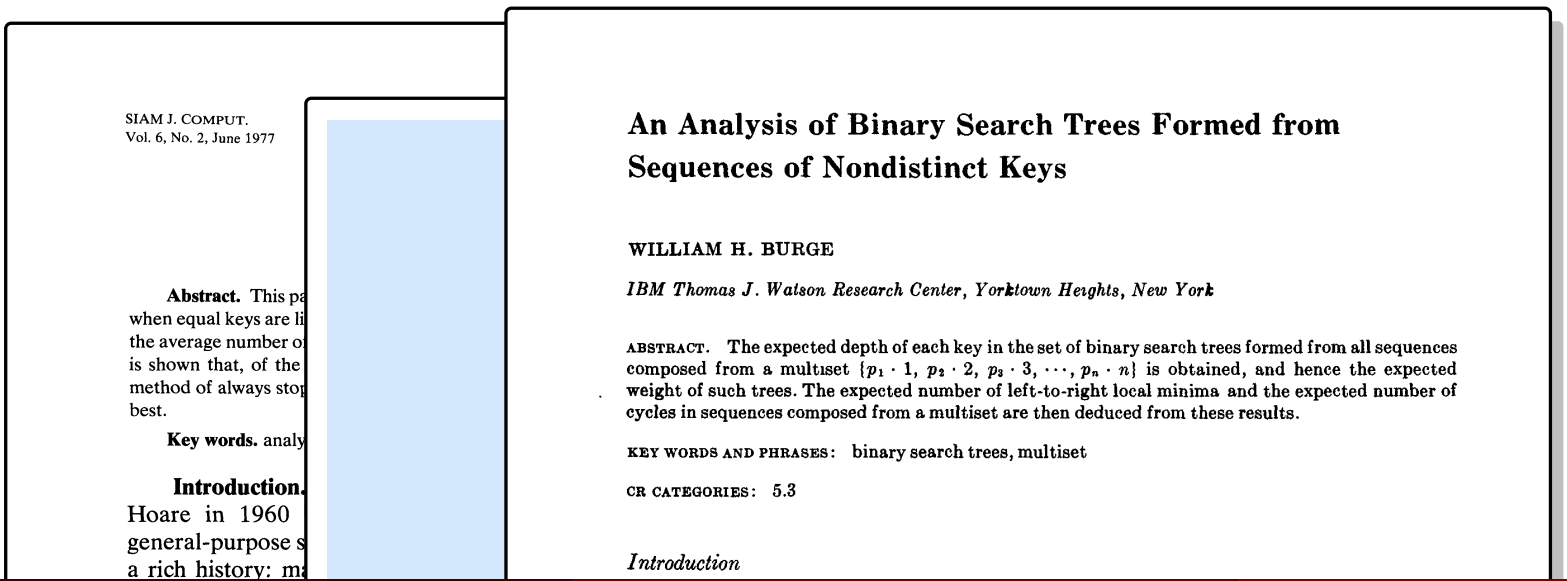
Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Previous Work

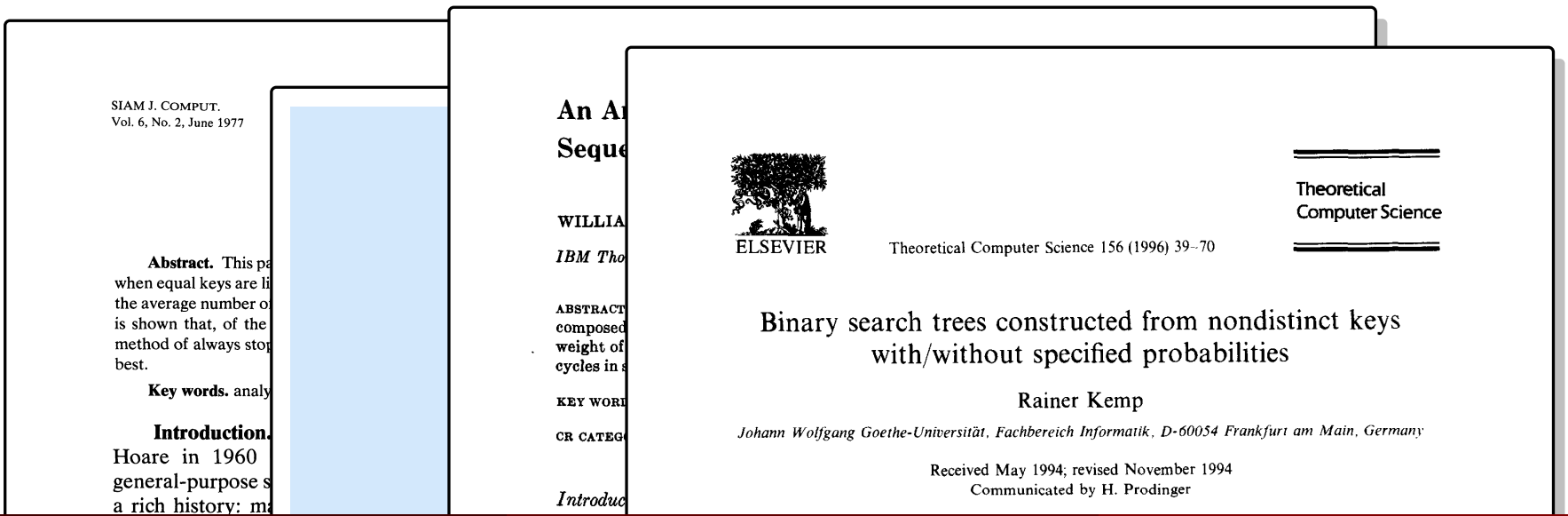
Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Previous Work

Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Previous Work

Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksort is Optimal* (Talk at Knuthfest)

A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Formed from Sequences of Nondistinct Keys*
- **Kemp** 1996: *BSTs constructed from nondistinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Previous Work

Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewic**

A bit more on

- **Burge** 19 *eys*
- **Kemp** 19 *probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

All these works only consider **basic** Quicksort:

- **No sampling** to choose pivots.
- **No multi-way** partitioning.

*This is basically **all** literature on analysis of Quicksort with equal keys!*



Input Models

Random-Permutation Model

no duplicates

1. $1, \dots, n$, randomly permuted

Parameters: n

Input Models

Random-Permutation Model

no duplicates

1. $1, \dots, n$, randomly permuted

Parameters: n

Random u-ary Word Model

1. random choice from $\{1, \dots, u\}^n$

Parameters: n, u

- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models

Random-Permutation Model

no duplicates

1. $1, \dots, n$, randomly permuted
2. n elem. i.i.d. uniform in $(0, 1)$

Parameters: n

Random u-ary Word Model

1. random choice from $\{1, \dots, u\}^n$
2. n elem. i.i.d. uniform in $\{1, \dots, u\}$

Parameters: n, u

- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models

Random-Permutation Model

no duplicates

1. $1, \dots, n$, randomly permuted
2. n elem. i.i.d. uniform in $(0, 1)$

Parameters: n

cont. $\rightsquigarrow$ disc.

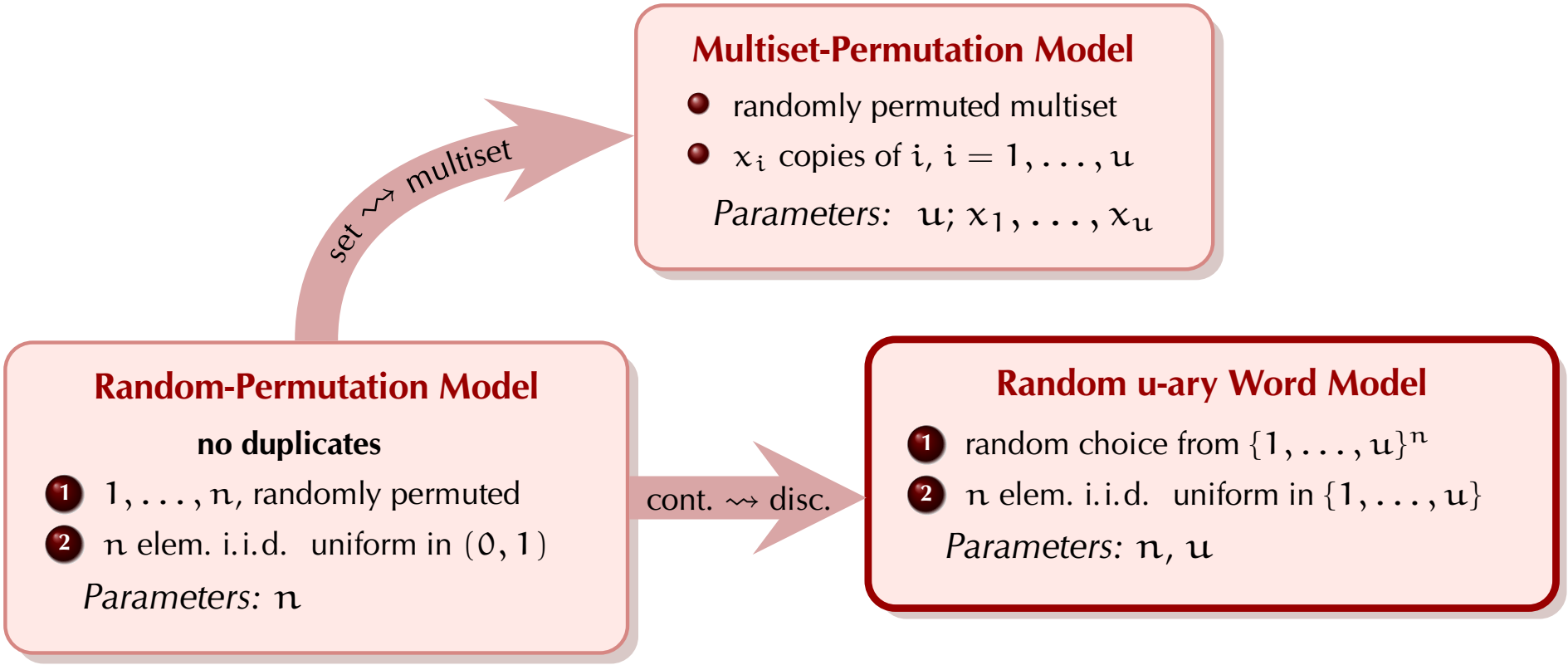
Random u-ary Word Model

1. random choice from $\{1, \dots, u\}^n$
2. n elem. i.i.d. uniform in $\{1, \dots, u\}$

Parameters: n, u

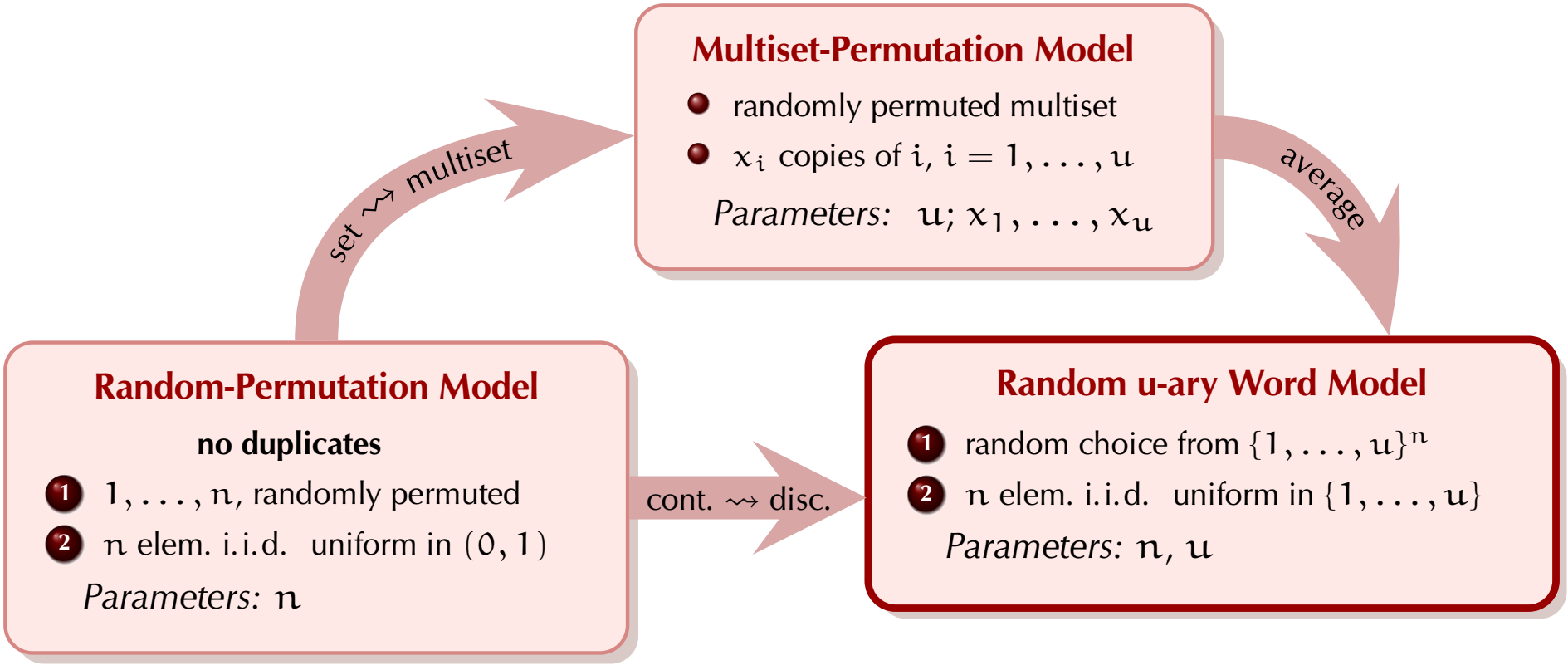
- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models



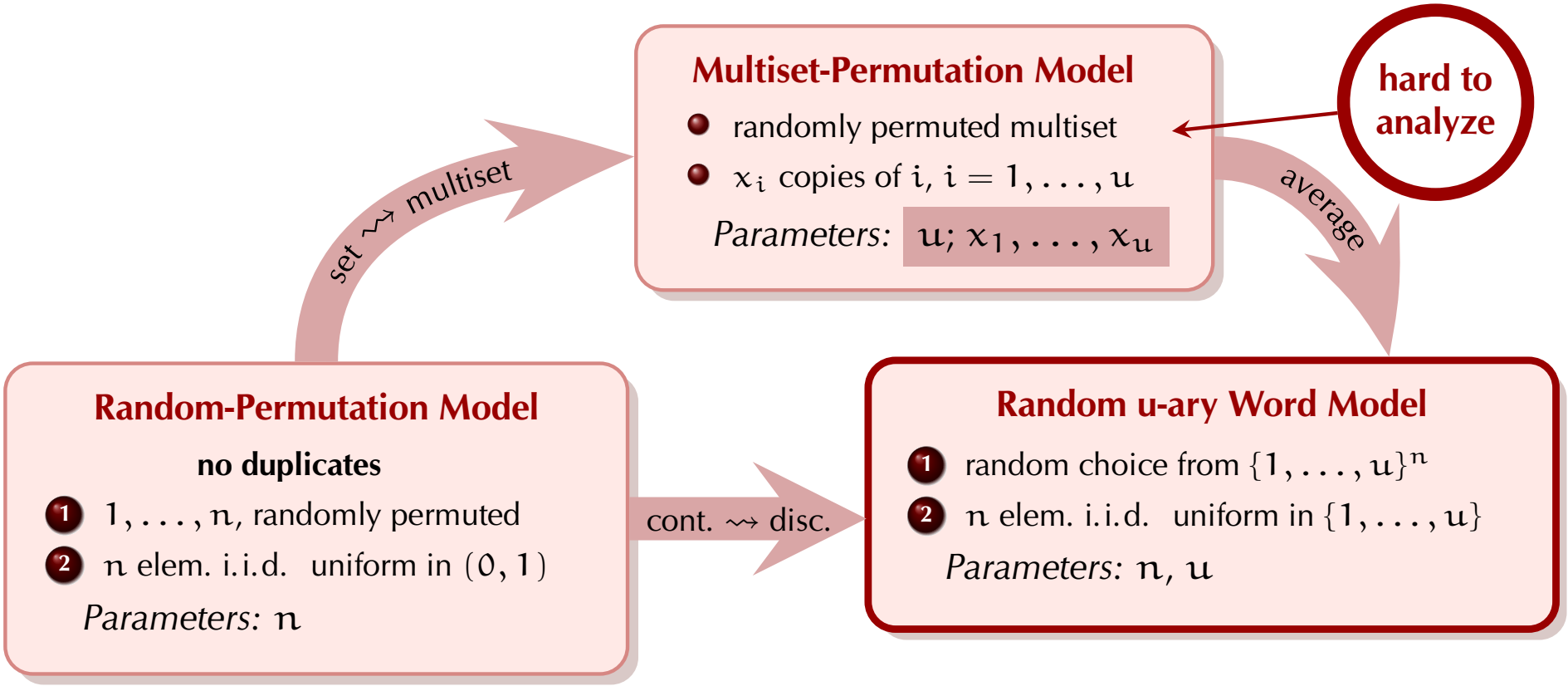
- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models



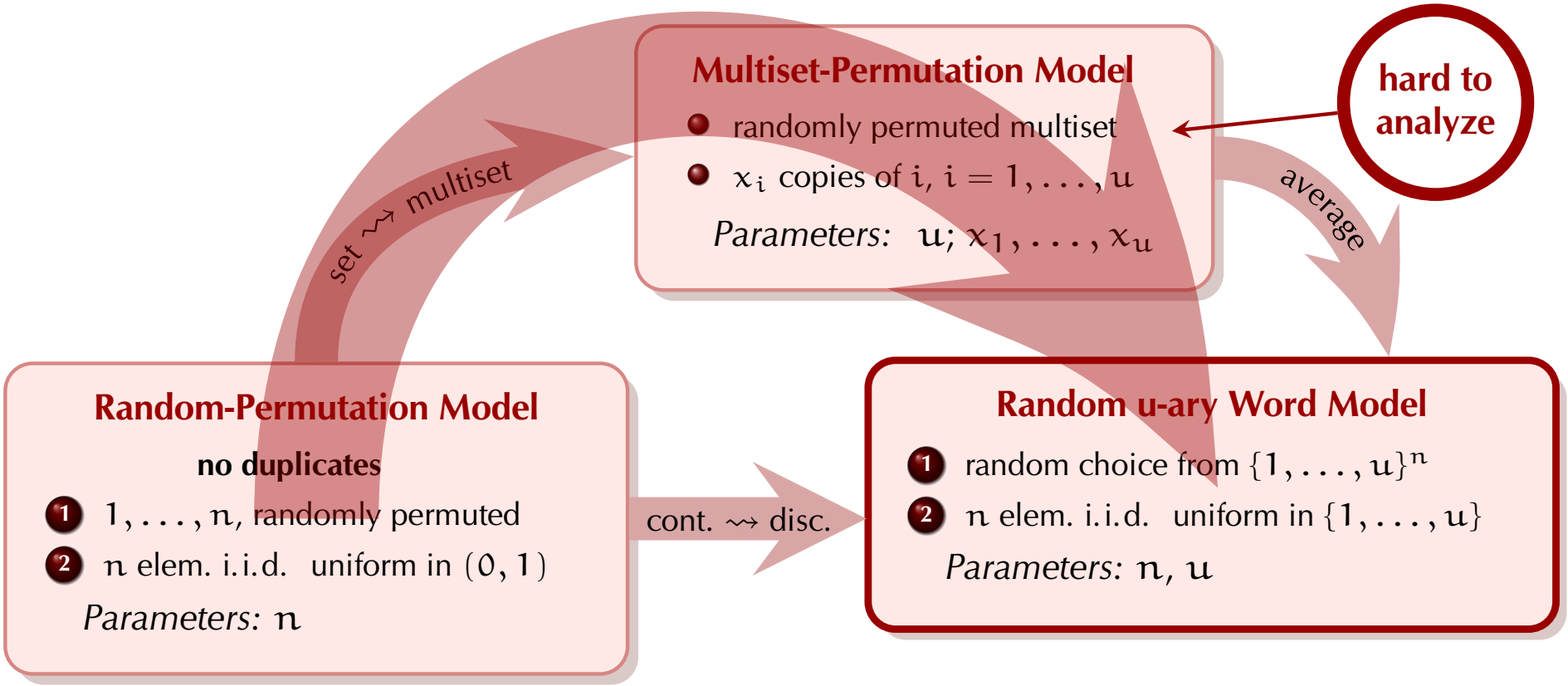
- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models



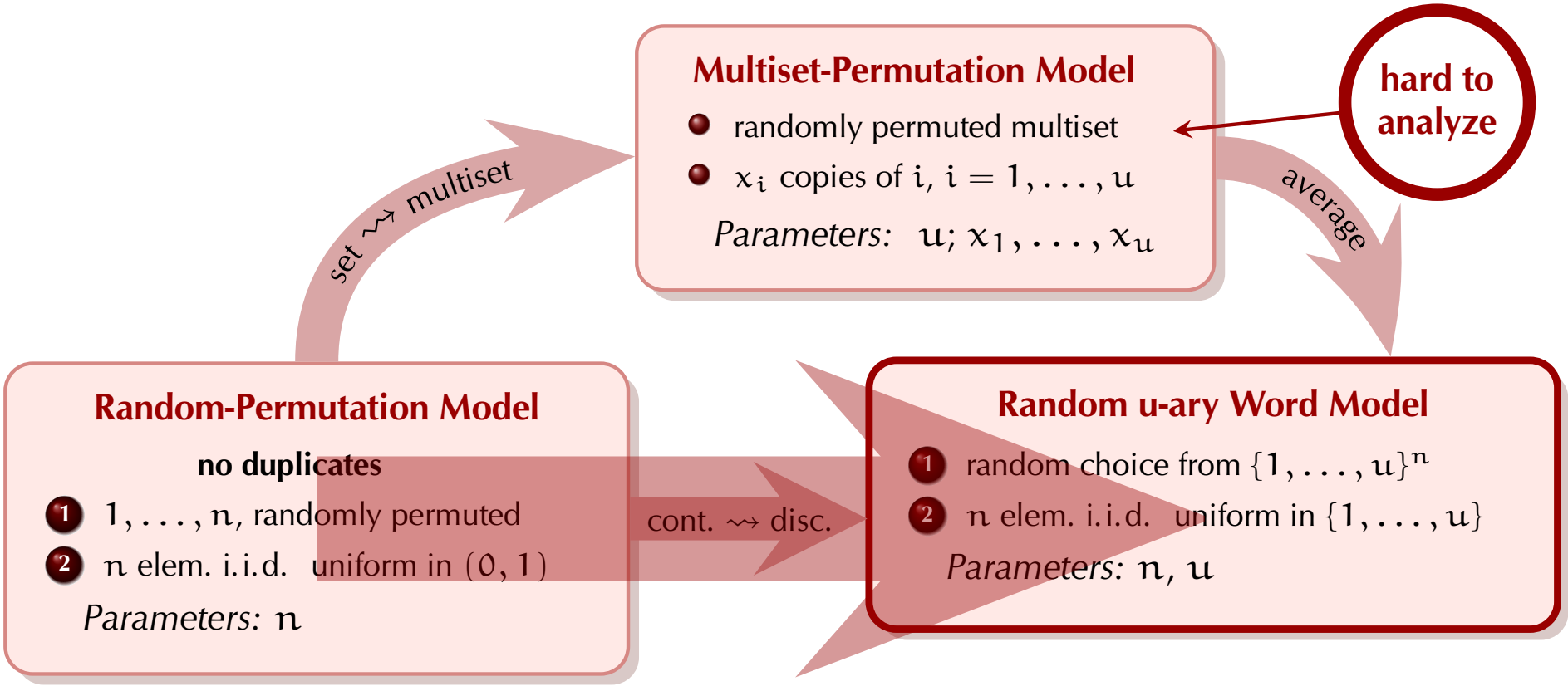
- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models



- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Input Models



- **My claim:** **u-ary words** most natural model ...
  in particular in i.i.d. interpretation!
- All previous works use **combinatorial detour**
- **Stochastic** point of view provides shortcut!

Setup

Assumptions:

1. **Input:** $U_1, \ldots, U_n$, i.i.d. uniform in $\{1, \ldots, u\}$.
2. **fat-pivot** partitioning
   - all duplicates of pivots removed
   - $\rightsquigarrow$ subproblems again random u-ary words, (with smaller n and u)
3. **Cost:** # ternary comparisons
   (extensions possible)

Generalized Quicksort:

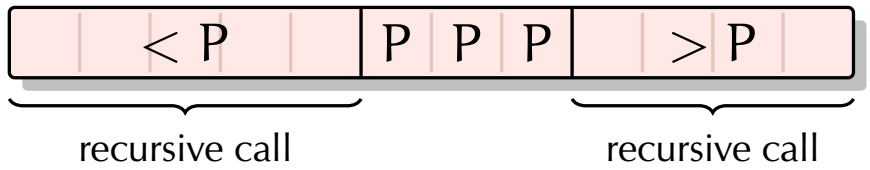
- k-way partitioning: divide into k segments
- pivot sampling: parameter $t = (t_1, \ldots, t_{k-1})$

Setup

Assumptions:

1. **Input:** $U_1, \dots, U_n$, i.i.d. uniform in $\{1, \dots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   $\rightsquigarrow$ subproblems **again** random u-ary words, (with smaller u and n)



3. **Cost:** # ternary comparisons
   (extensions possible)

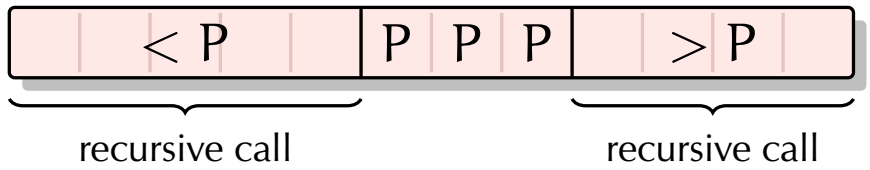
Generalized Quicksort:

- k-way partitioning: divide into k segments
- uses sampling: parameter $\vec{t} = (t_1, \dots, t_{k-1})$

Setup

Assumptions:

1. **Input:** $U_1, \dots, U_n$, i.i.d. uniform in $\{1, \dots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed



   ⇝ subproblems **again** random u-ary words, (with smaller u and n)

3. **Cost:** # ternary comparisons
   (extensions possible)

Generalized Quicksort:

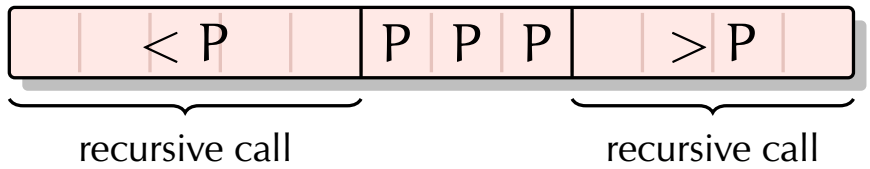
- 3-way partitioning: divide into 3 segments
- pivot sampling parameter $\vec{t} = (t_1, \dots, t_{u-1})$

Setup

Assumptions:

1. **Input:** $U_1, \ldots, U_n$, i.i.d. uniform in $\{1, \ldots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   $\rightsquigarrow$ subproblems **again** random u-ary words, (with smaller u and n)



3. **Cost:** # ternary comparisons
   (extensions possible)

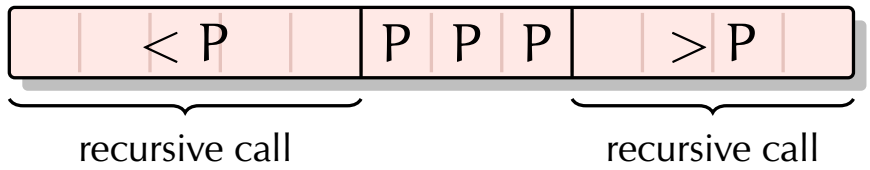
Generalized Quicksort:

- **s-way partitioning:** divide into s segments
- **pivot sampling:** parameter $\vec{t} = (t_1, \ldots, t_{s-1})$

Setup

Assumptions:

1. **Input:** $U_1, \ldots, U_n$, i.i.d. uniform in $\{1, \ldots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   $\rightsquigarrow$ subproblems **again** random u-ary words, (with smaller u and n)
3. **Cost:** # ternary comparisons
   (extensions possible)

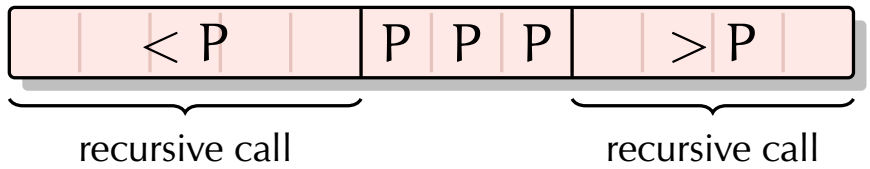
Generalized Quicksort:

- **s-way partitioning:** divide into s segments
- **pivot sampling:** parameter $\vec{t} = (t_1, \ldots, t_s)$

Setup

Assumptions:

1. **Input:** $U_1, \ldots, U_n$, i.i.d. uniform in $\{1, \ldots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   $\rightsquigarrow$ subproblems **again** random u-ary words, (with smaller u and n)
3. **Cost:** # ternary comparisons
   (extensions possible)

Generalized Quicksort:

- **s-way partitioning:** divide into s segments
- **pivot sampling:** parameter $\vec{t} = (t_1, \ldots, t_s)$
  - median-of-$(2t+1)$ $\rightsquigarrow$ $\vec{t} = (t, t)$
  - asymmetric sampling allowed

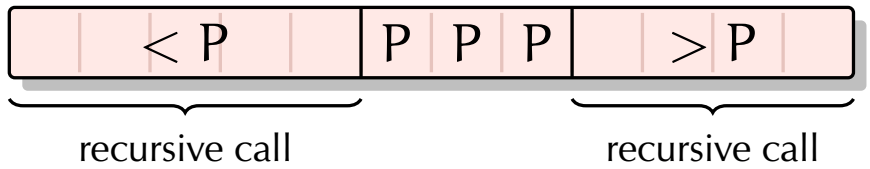
Examples:
$s = 3$ and $\vec{t} = (1, 2, 3)$

$\rightsquigarrow$ 2nd and 5th from sample of 8.

Setup

Assumptions:

1. **Input:** $U_1, \dots, U_n$, i.i.d. uniform in $\{1, \dots, u\}$.
2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

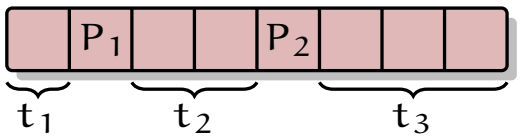
   

   $\rightsquigarrow$ subproblems **again** random u-ary words, (with smaller u and n)
3. **Cost:** # ternary comparisons
   (extensions possible)

Generalized Quicksort:

- **s-way partitioning:** divide into s segments
- **pivot sampling:** parameter $\vec{t} = (t_1, \dots, t_s)$
  - median-of-$(2t+1)$ $\rightsquigarrow$ $\vec{t} = (t, t)$
  - asymmetric sampling allowed

Example:
$s = 3$ and $\vec{t} = (1, 2, 3)$



$\rightsquigarrow$ 2nd and 5th from sample of 8.

Quicksort & Search Trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort & Search Trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort & Search Trees

Classic Fact:

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

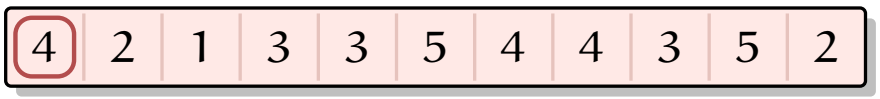
4	2	1	3	3	5	4	4	3	5	2

Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)

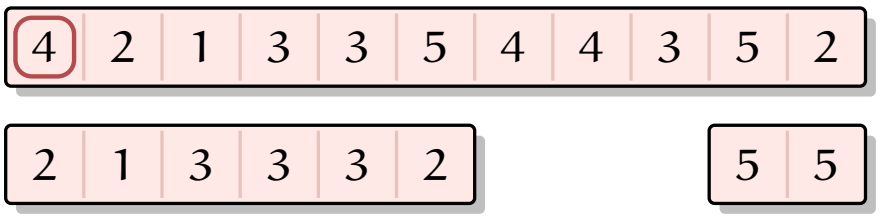


Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



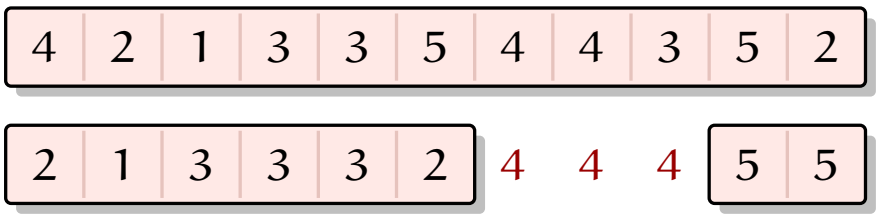
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



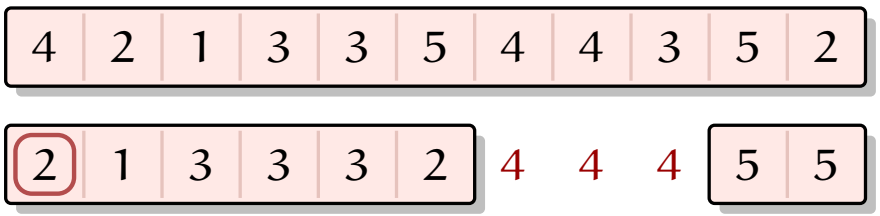
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



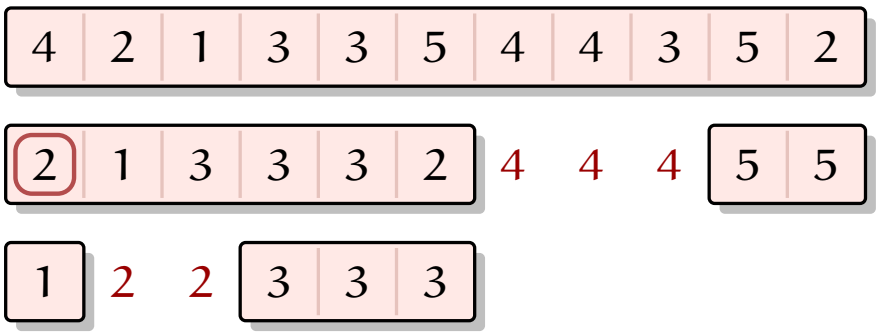
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



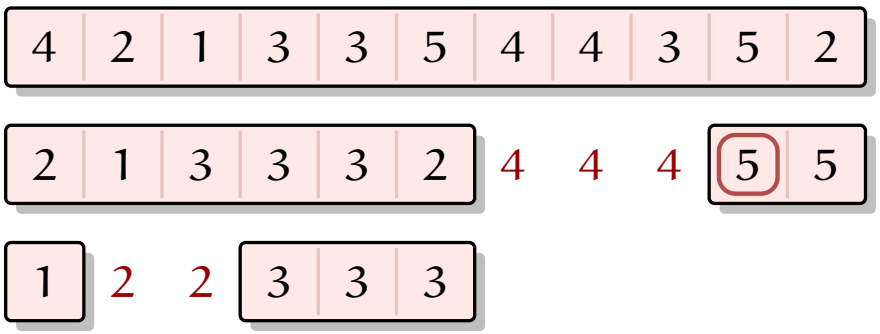
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



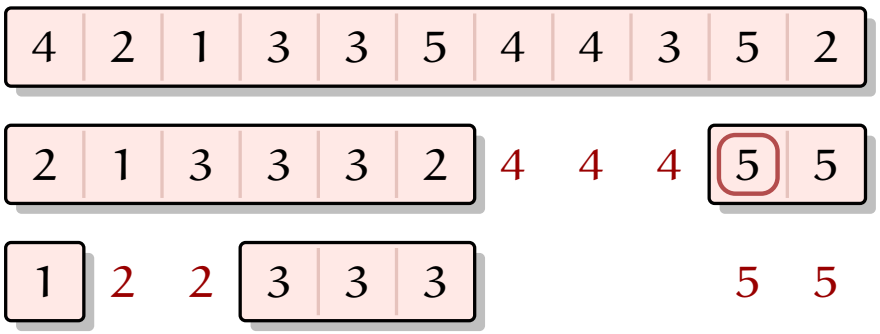
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



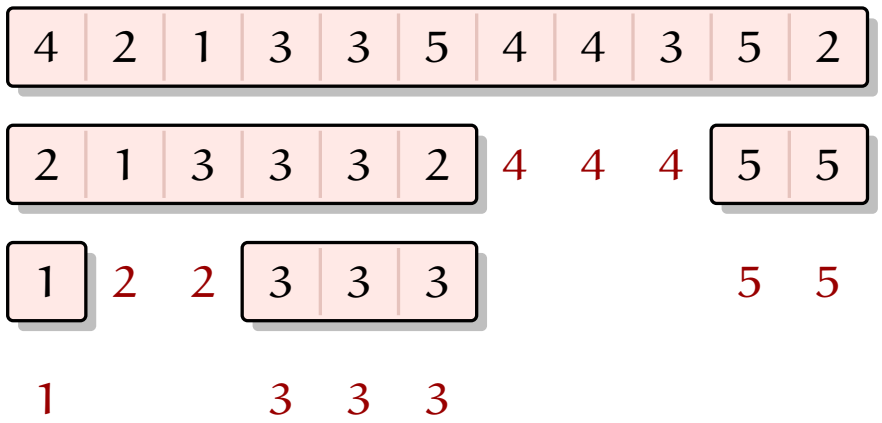
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



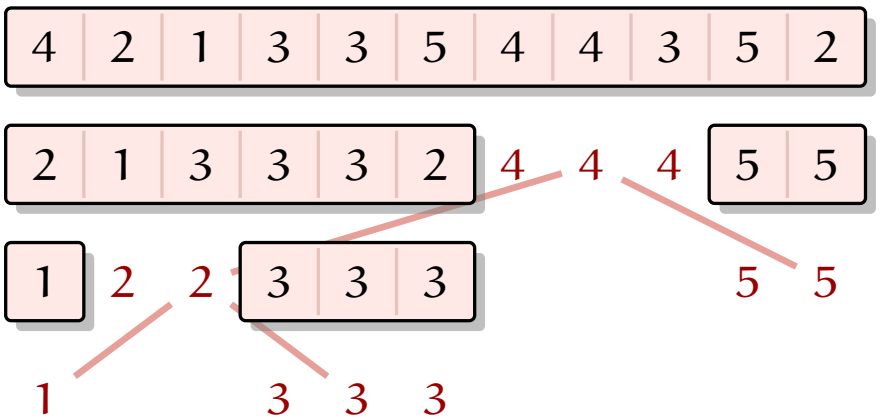
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)

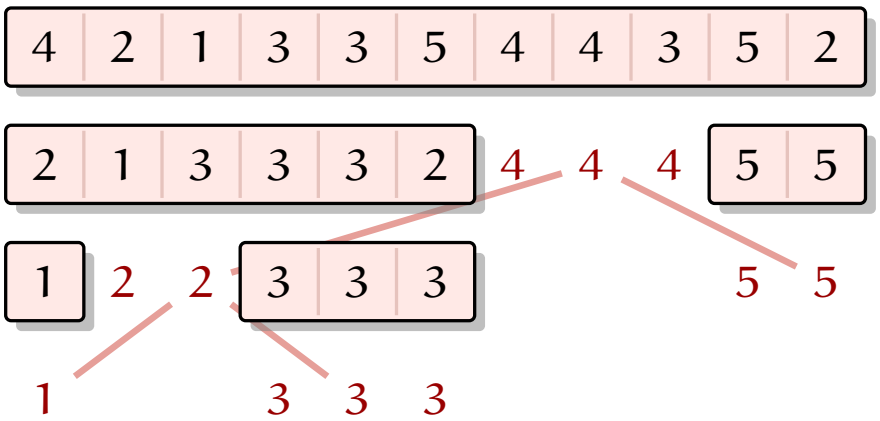


Quicksort & Search Trees

Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

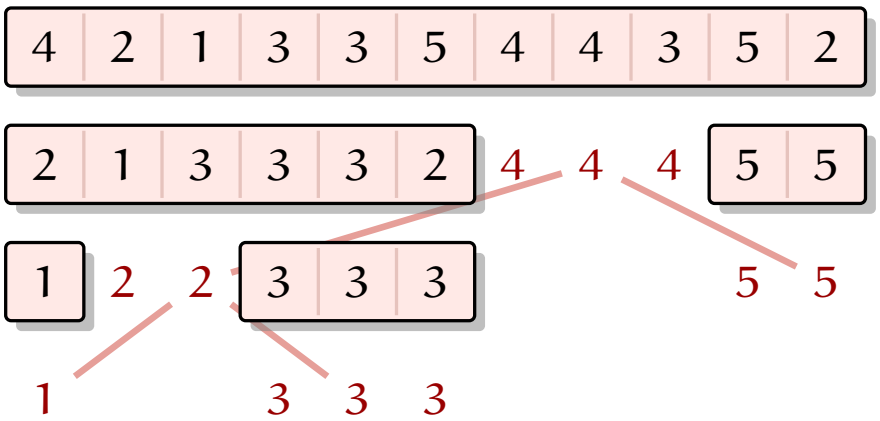
4 2 1 3 3 5 4 4 3 5 2

Quicksort & Search Trees

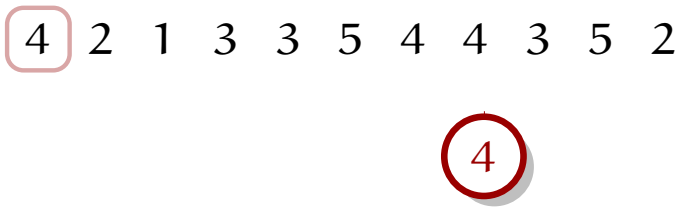
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

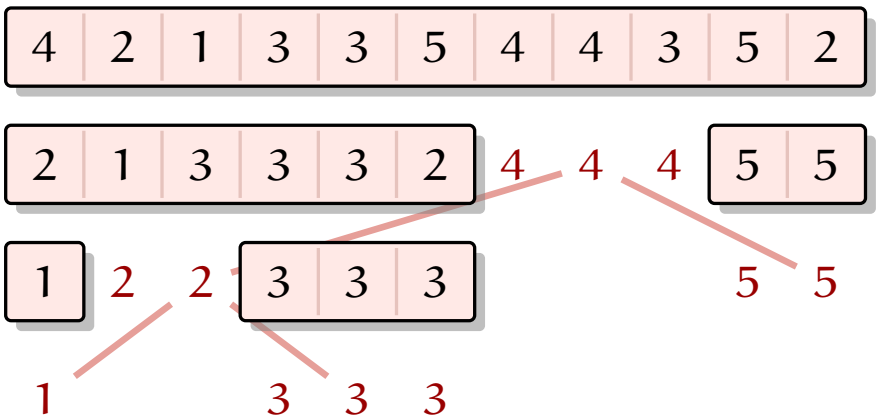


Quicksort & Search Trees

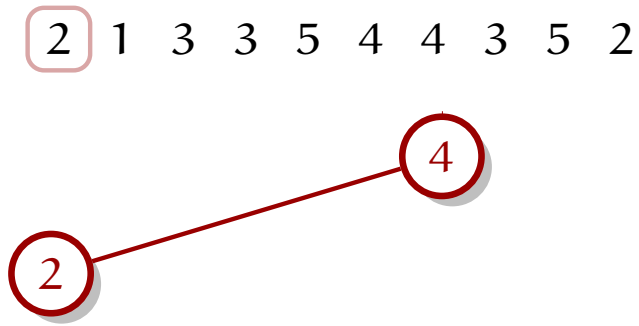
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

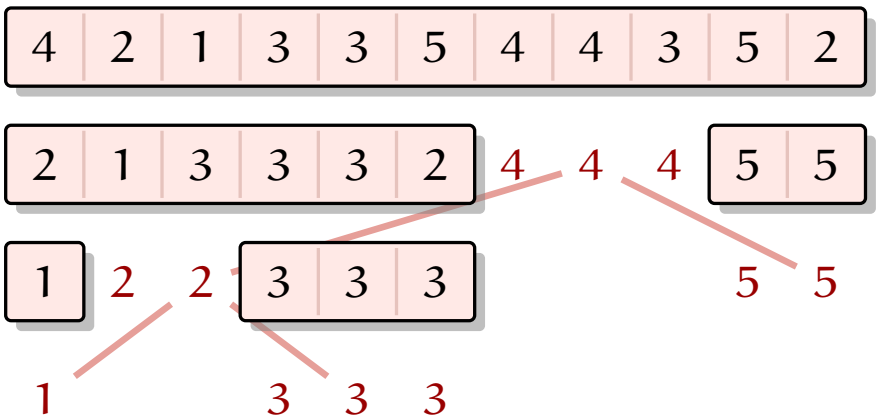


Quicksort & Search Trees

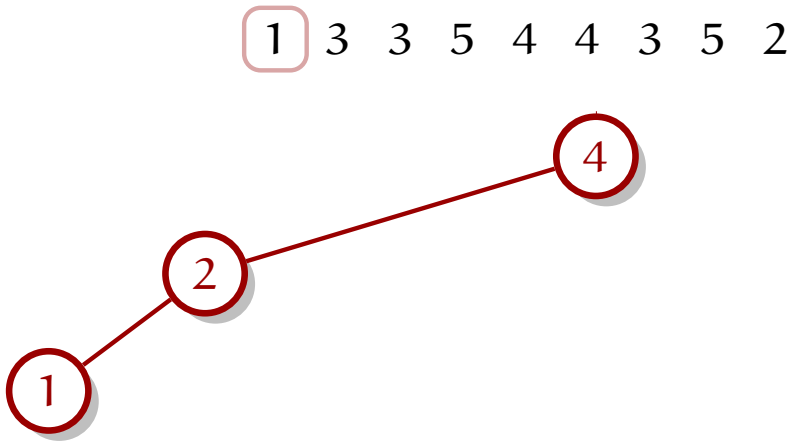
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

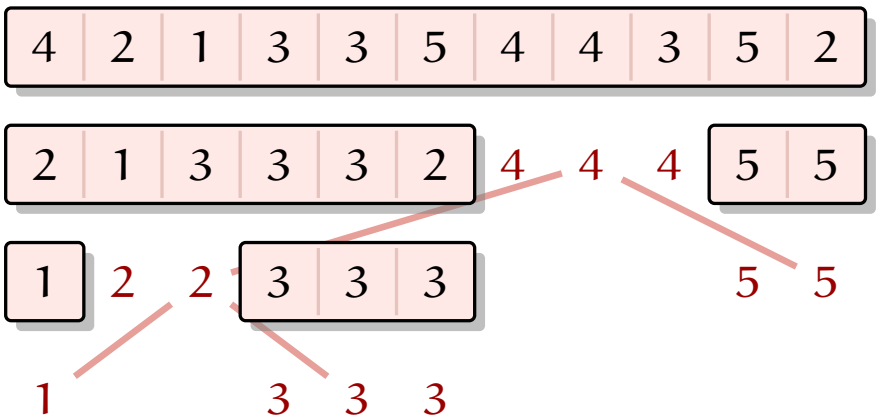


Quicksort & Search Trees

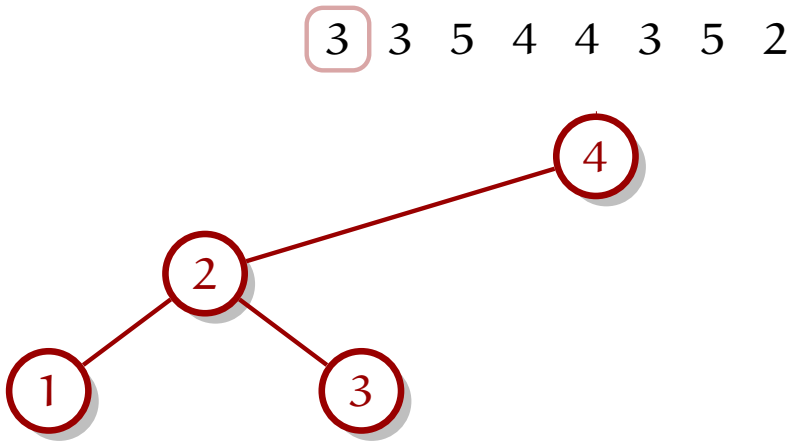
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



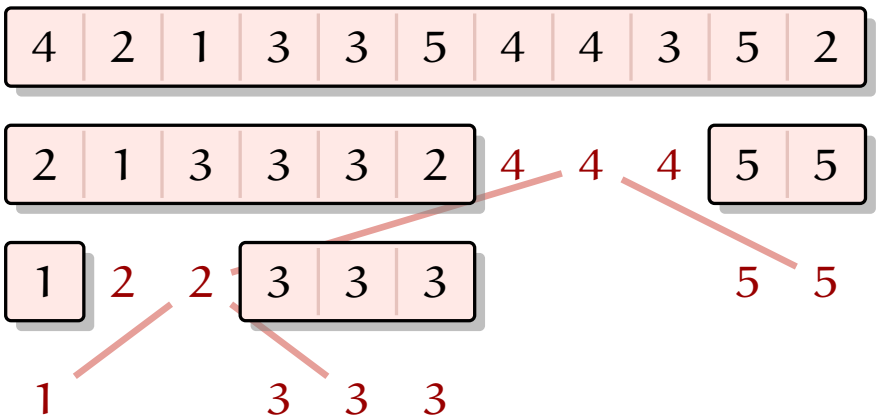
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

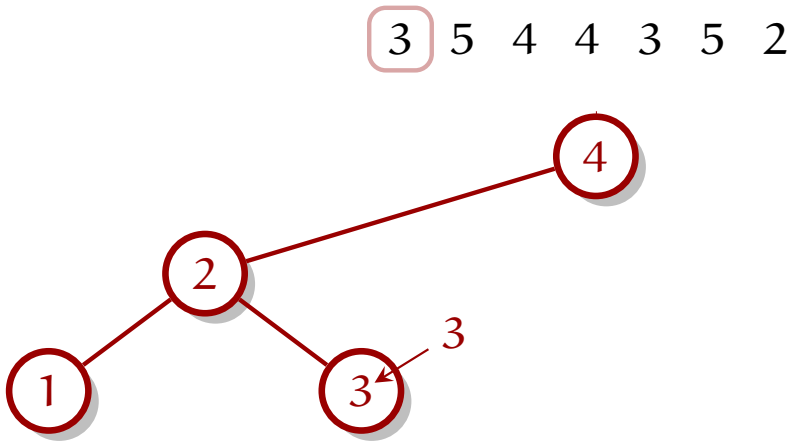
- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

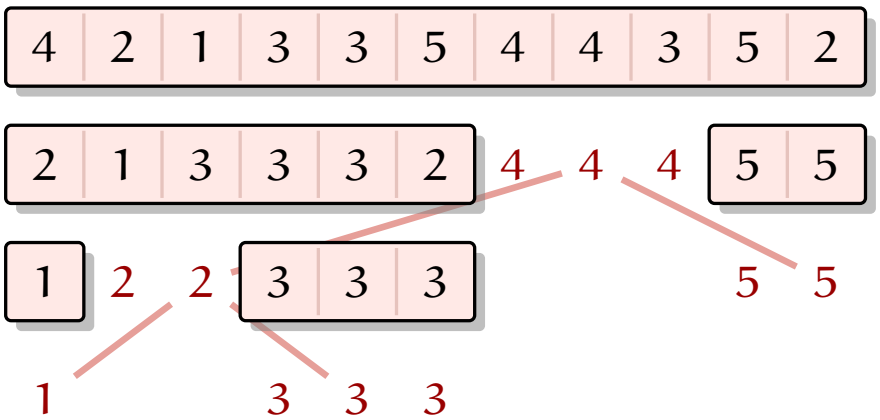


Quicksort & Search Trees

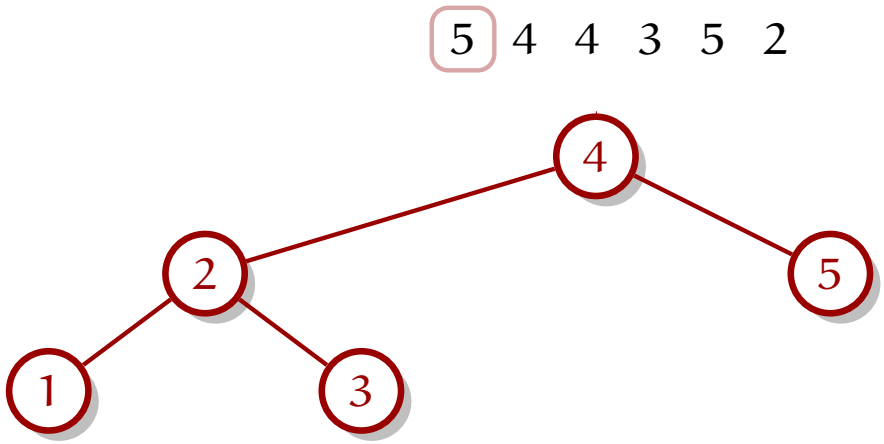
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

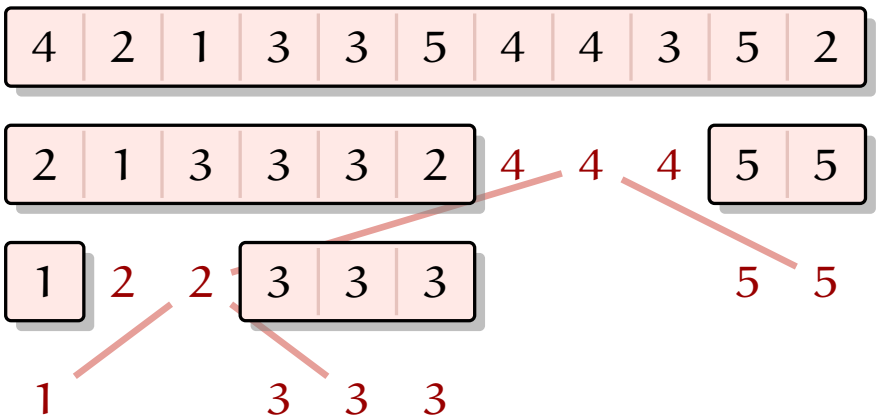


Quicksort & Search Trees

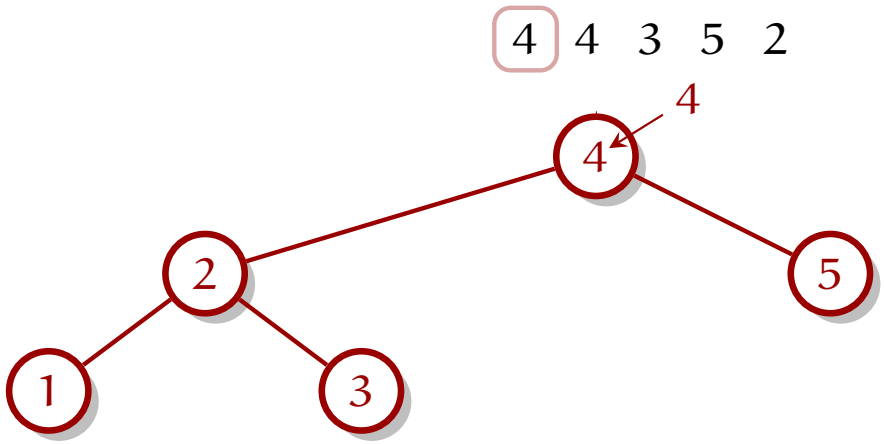
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

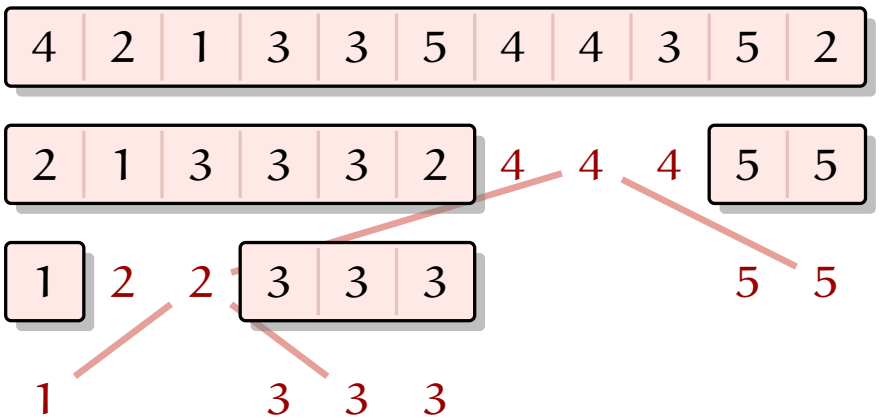


Quicksort & Search Trees

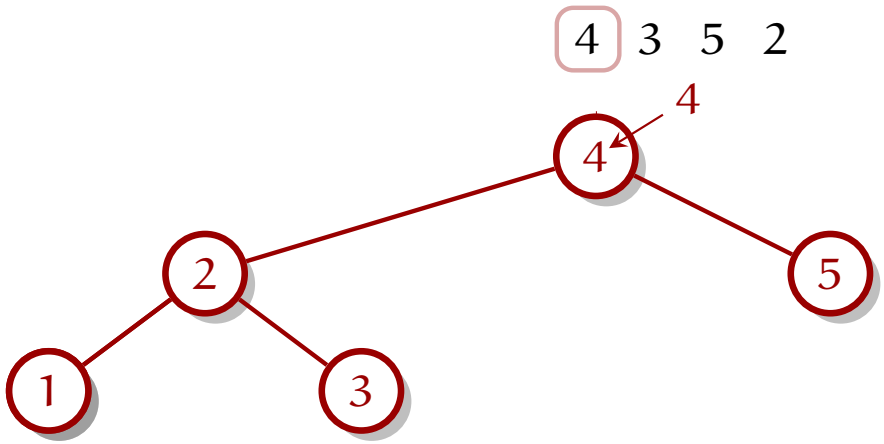
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

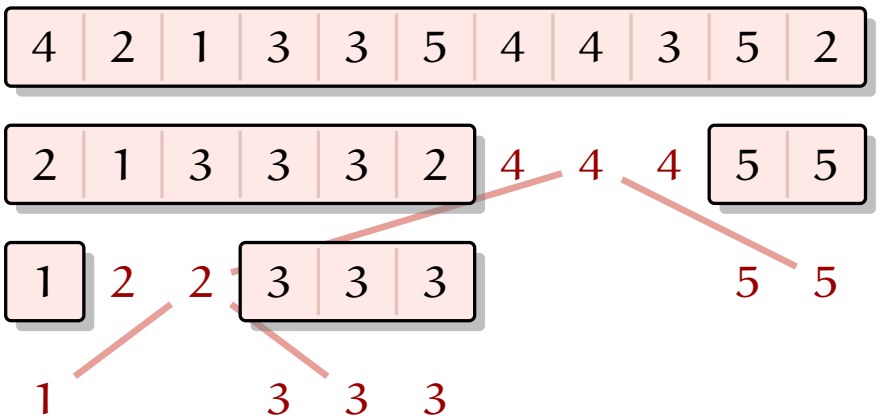


Quicksort & Search Trees

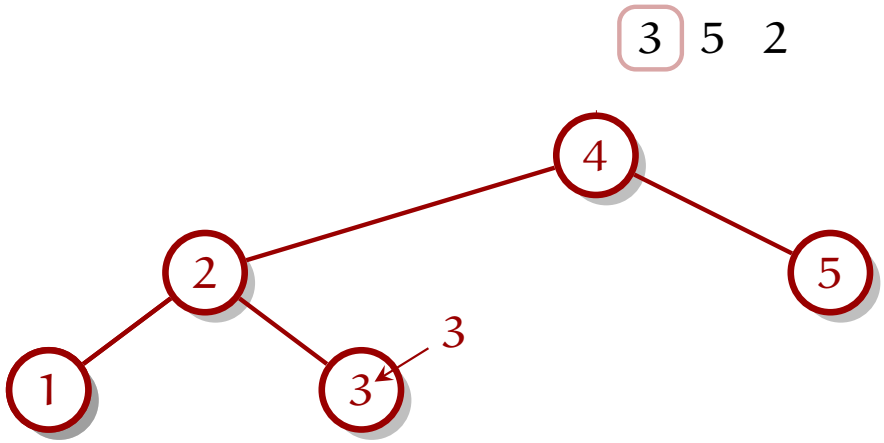
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



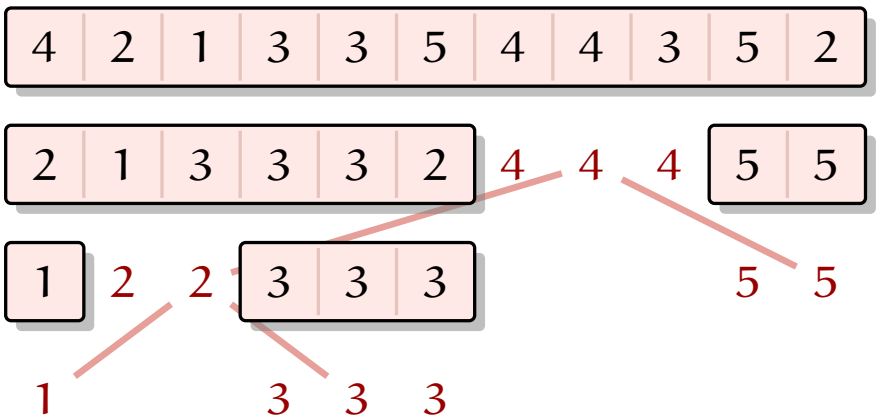
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

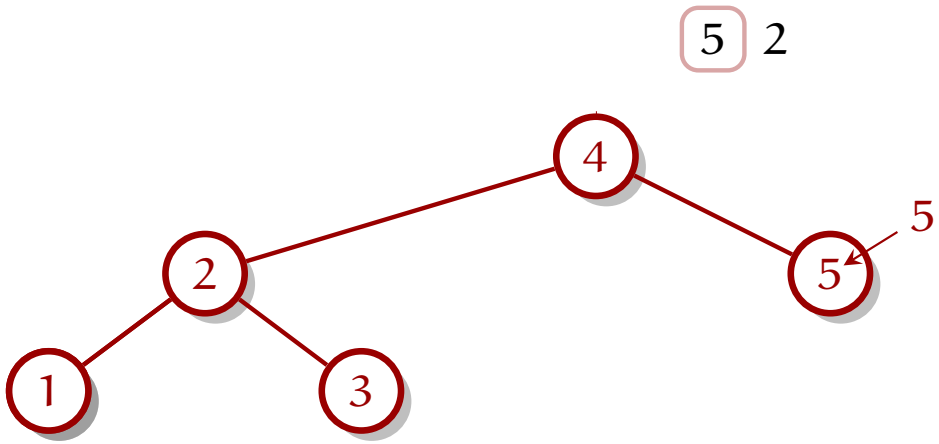
- Recursion Tree of Quicksort = Naturally grown BST from input
  - $\rightsquigarrow$ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



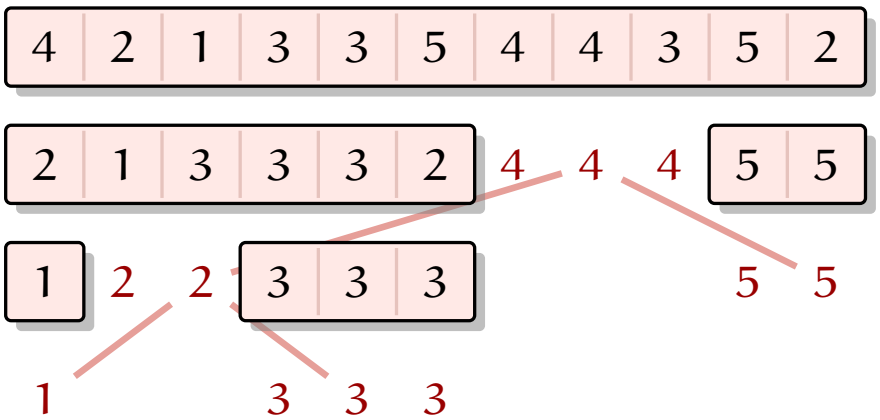
Quicksort & Search Trees

Classic Fact: (**without** duplicates)

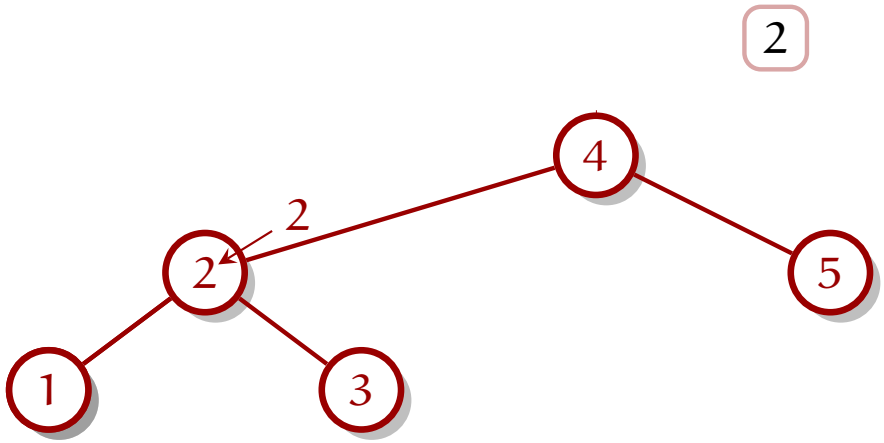
- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



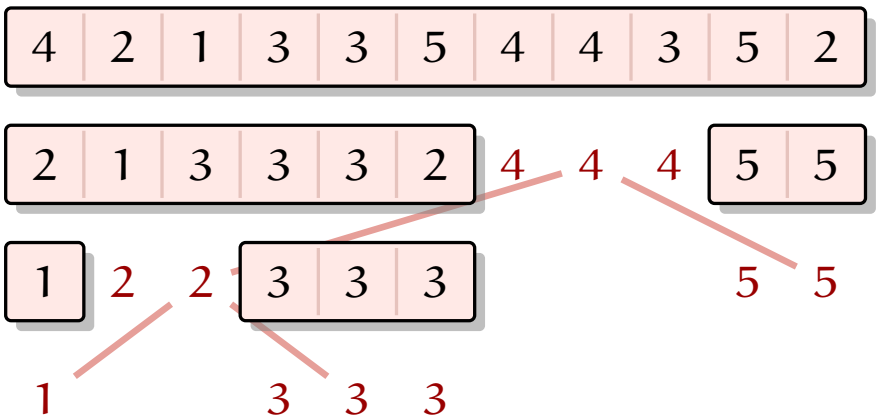
⇝ Equivalence holds also with duplicates.

Quicksort & Search Trees

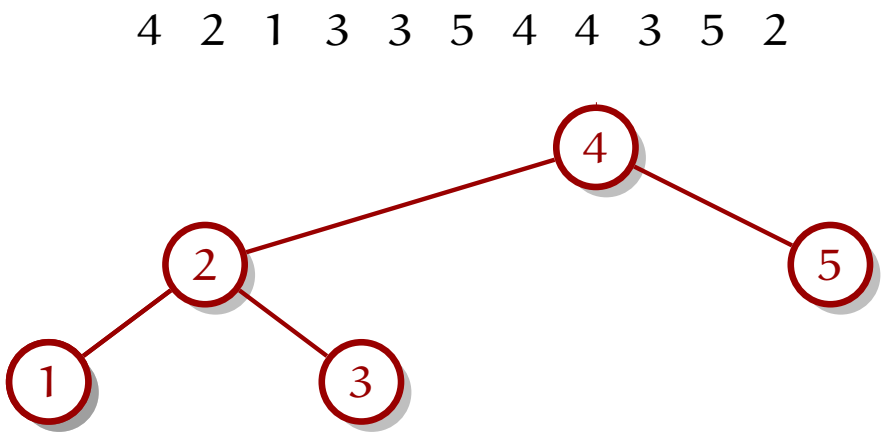
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



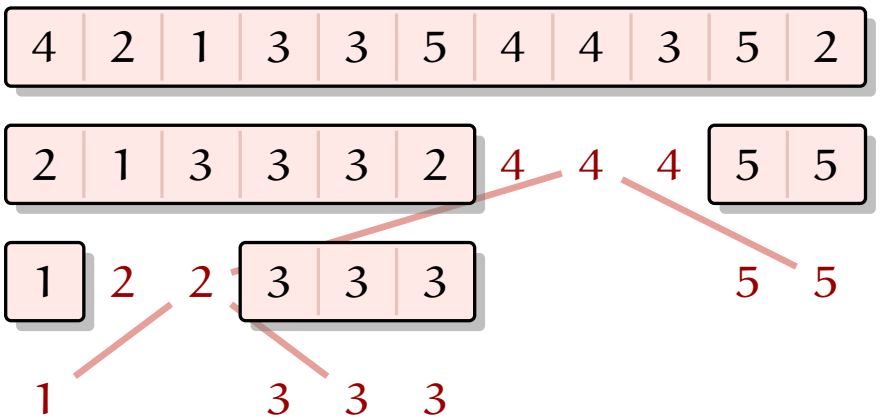
⇝ Equivalence holds also with duplicates.

Quicksort & Search Trees

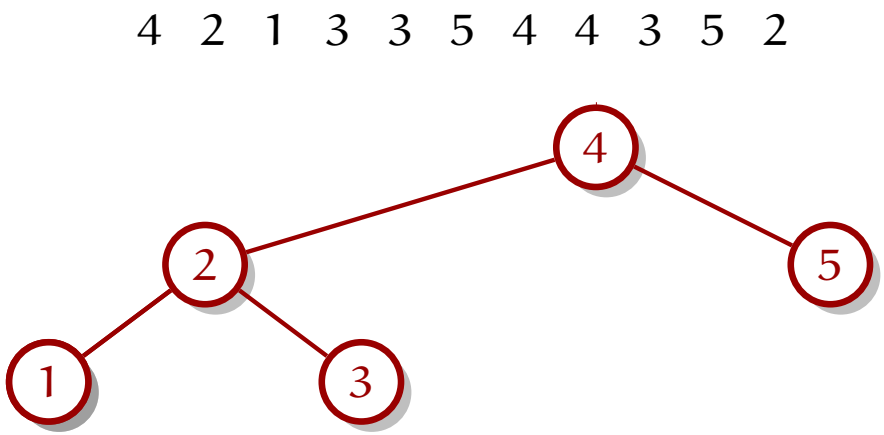
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



⇝ Equivalence holds also with duplicates.

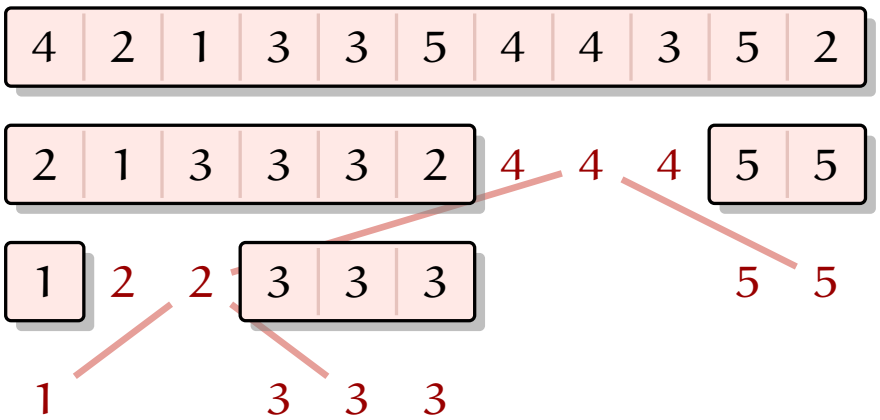
*This was only basic Quicksort . . . how about **pivot sampling**?*

Quicksort & Search Trees

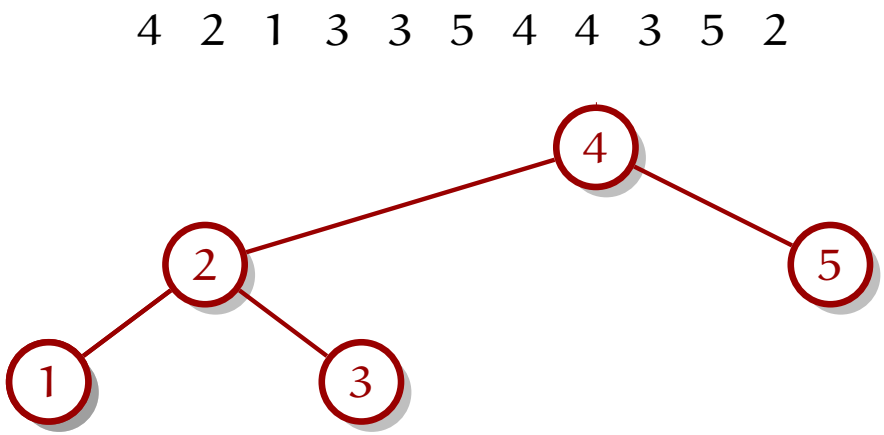
Classic Fact: (**without** duplicates)

- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST
- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



⇝ Equivalence holds also with duplicates.

*This was only basic Quicksort . . . how about **pivot sampling**?*

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

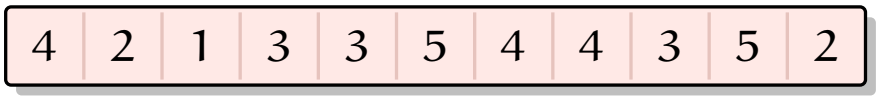
- Leaves have **buffer** for elements, to emulate pivot sampling

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

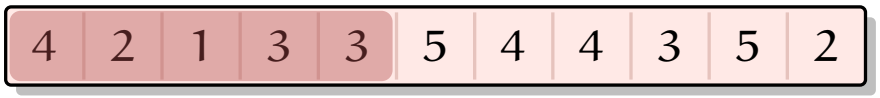


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

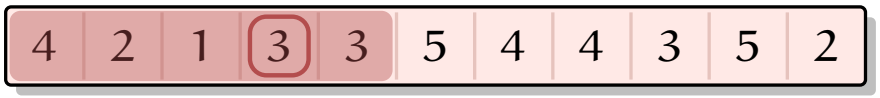


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

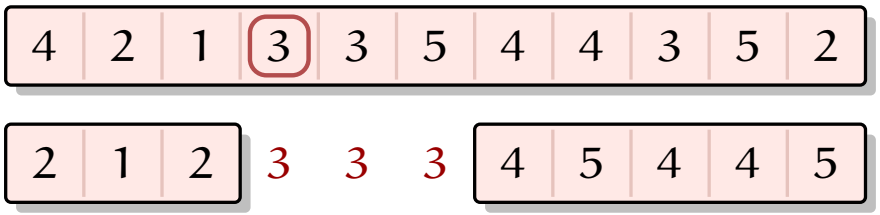


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

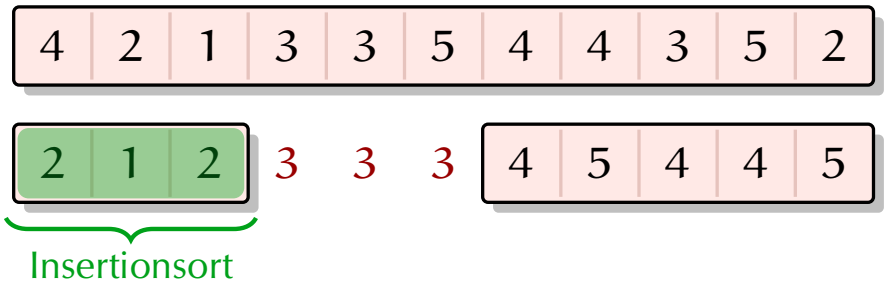


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

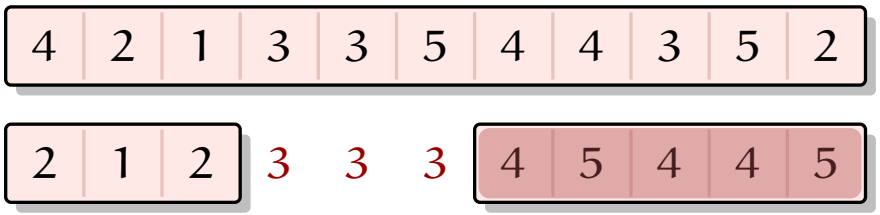


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

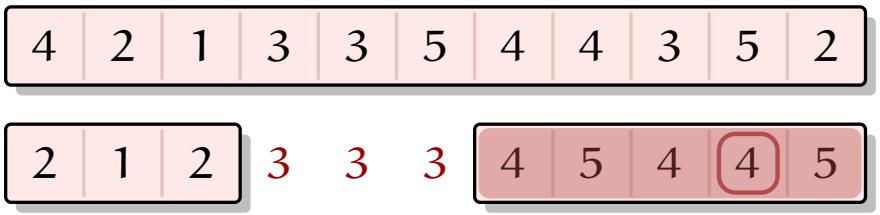


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

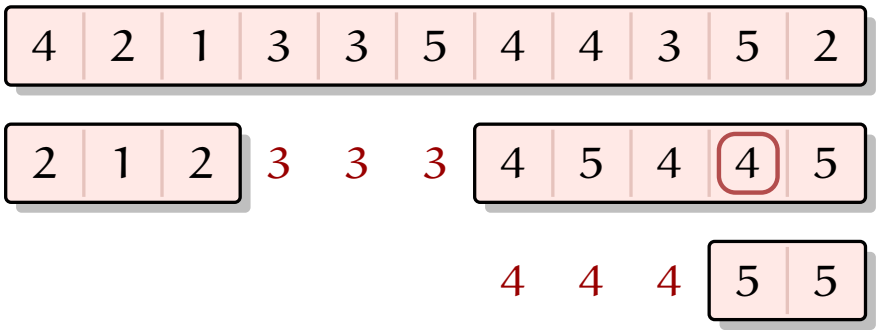


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$

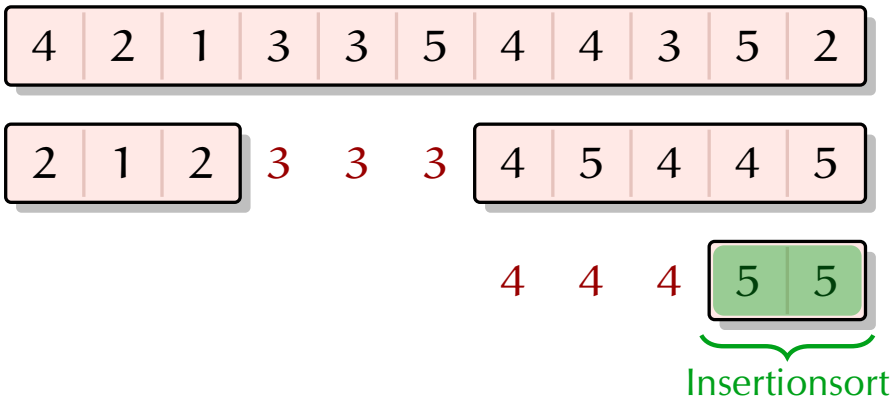


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

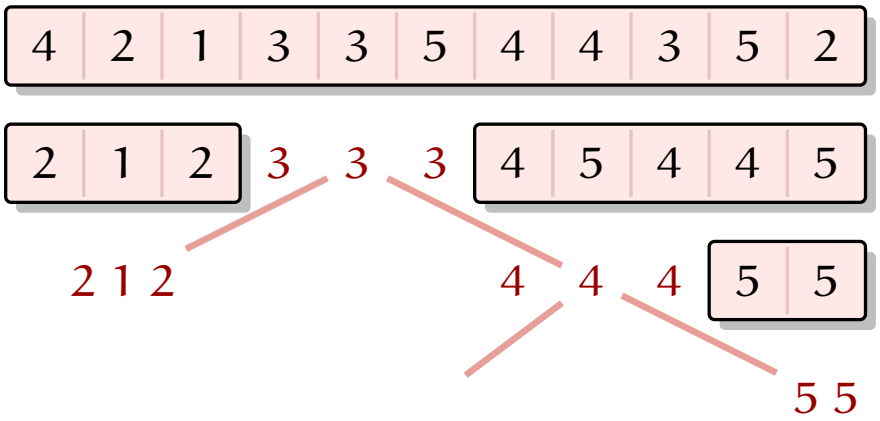


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

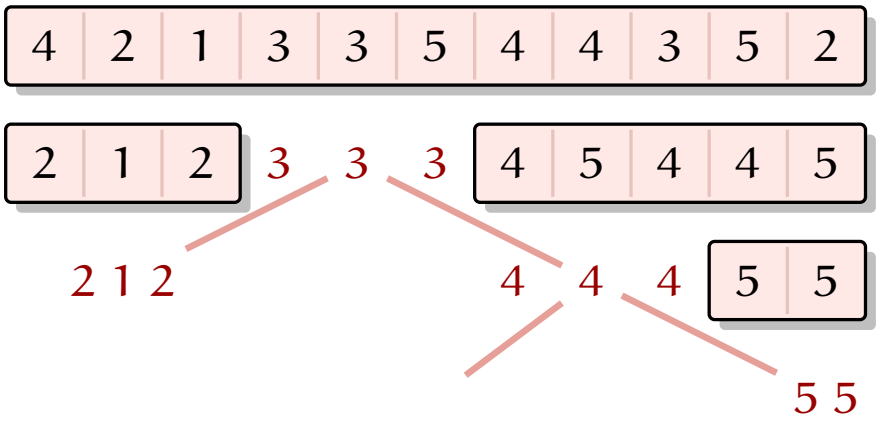


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

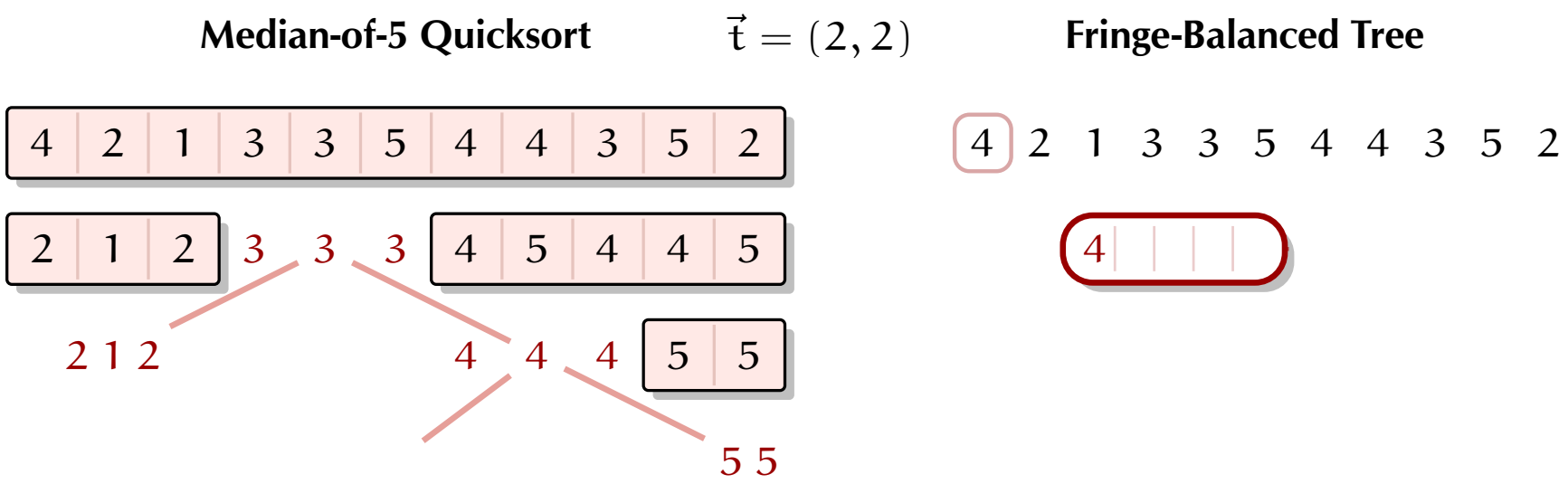




Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

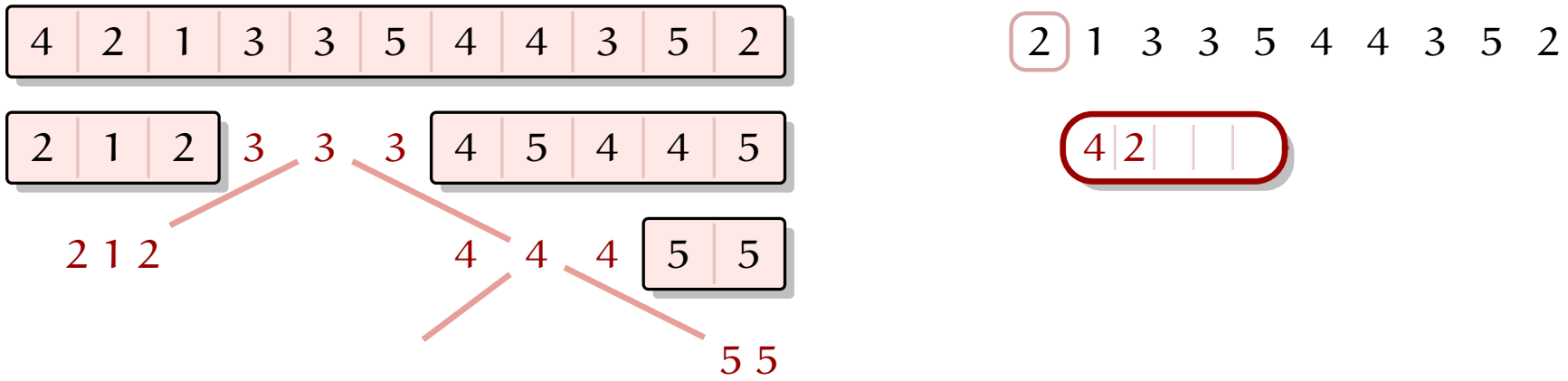


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

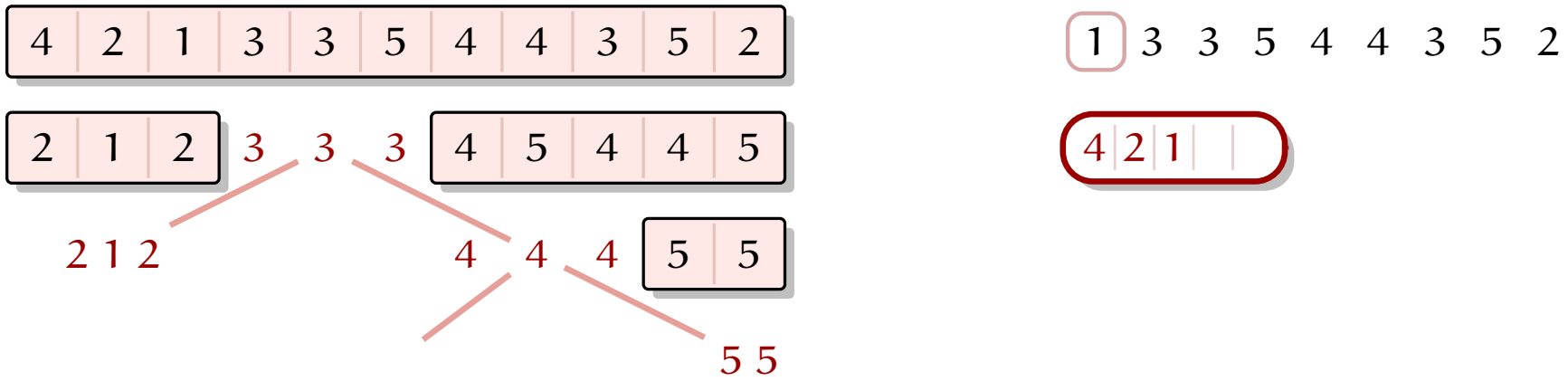


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

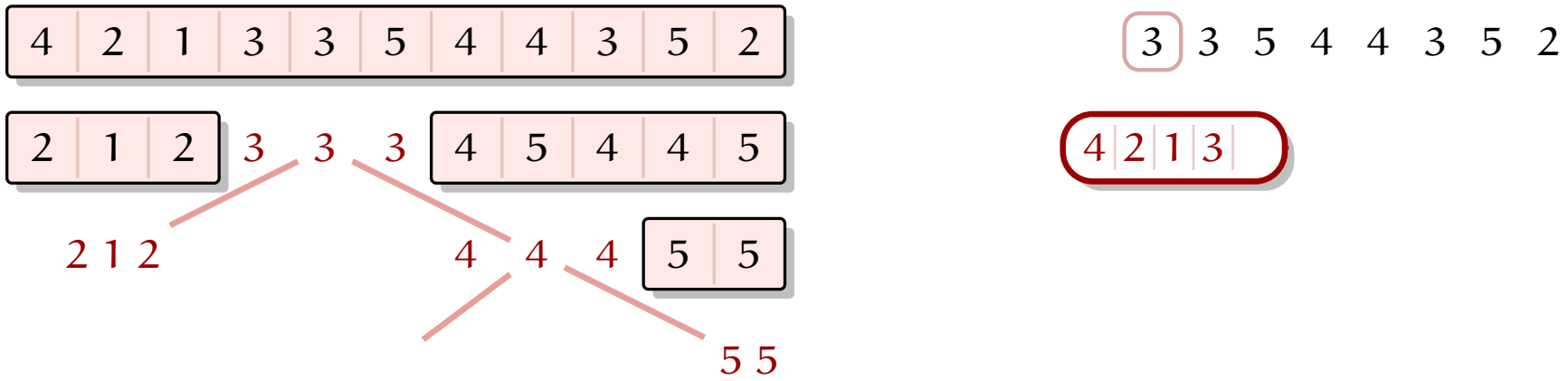


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

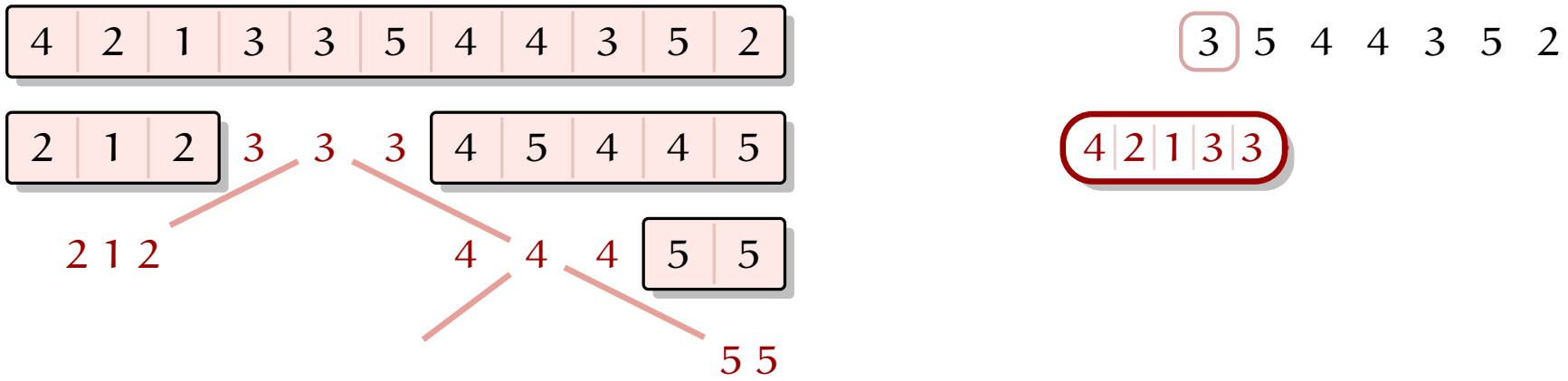


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

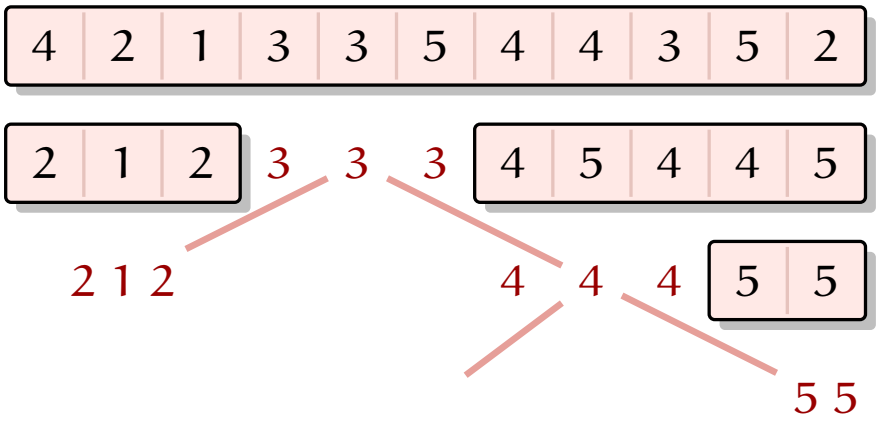


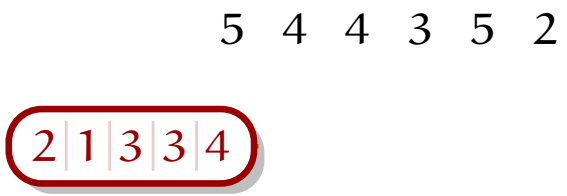
Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**



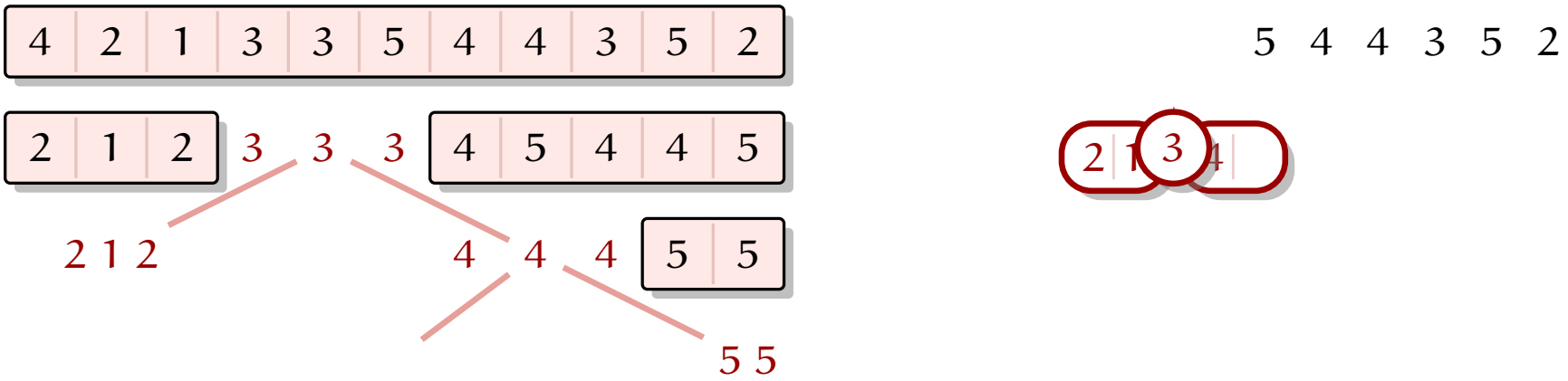


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**

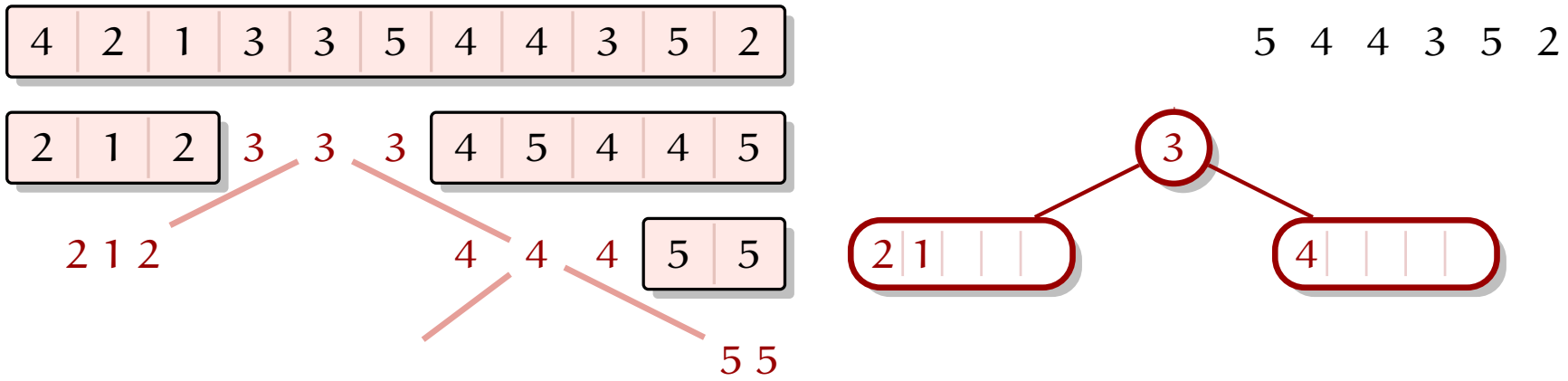


Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling

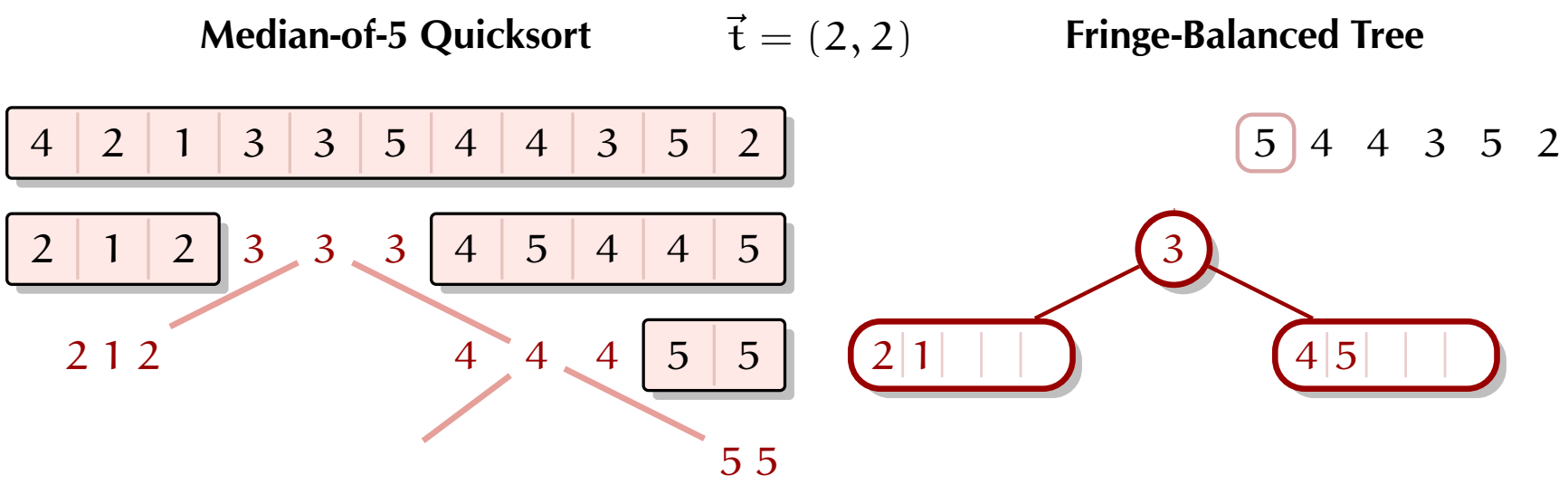
Median-of-5 Quicksort $\quad \vec{t} = (2, 2)$ **Fringe-Balanced Tree**



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

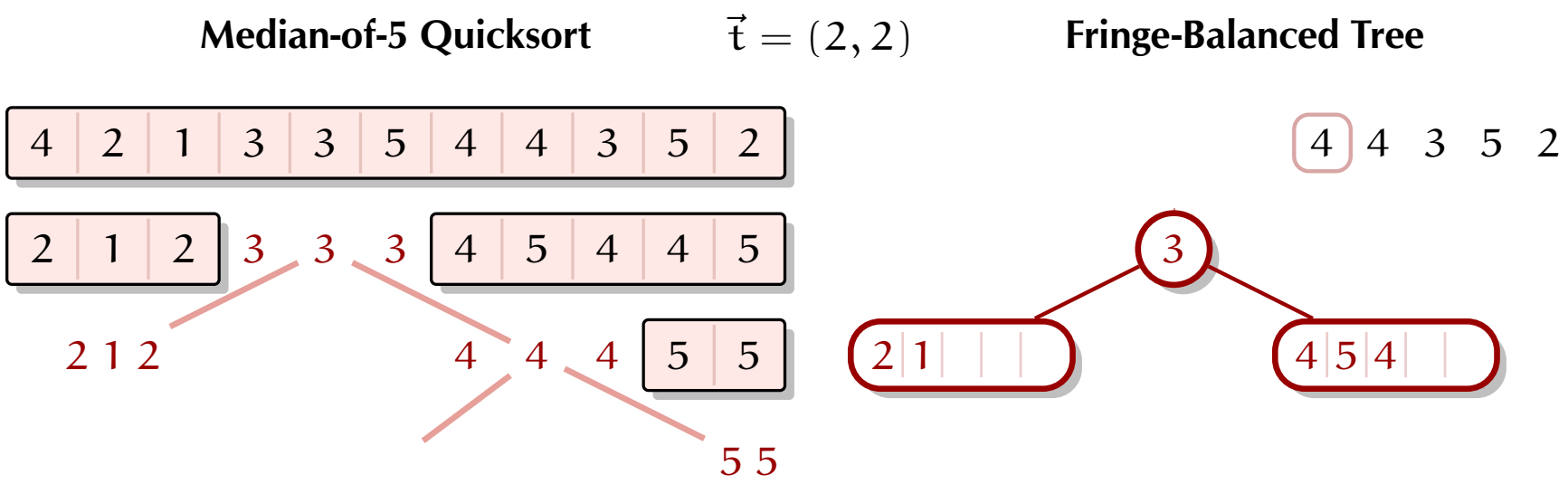
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

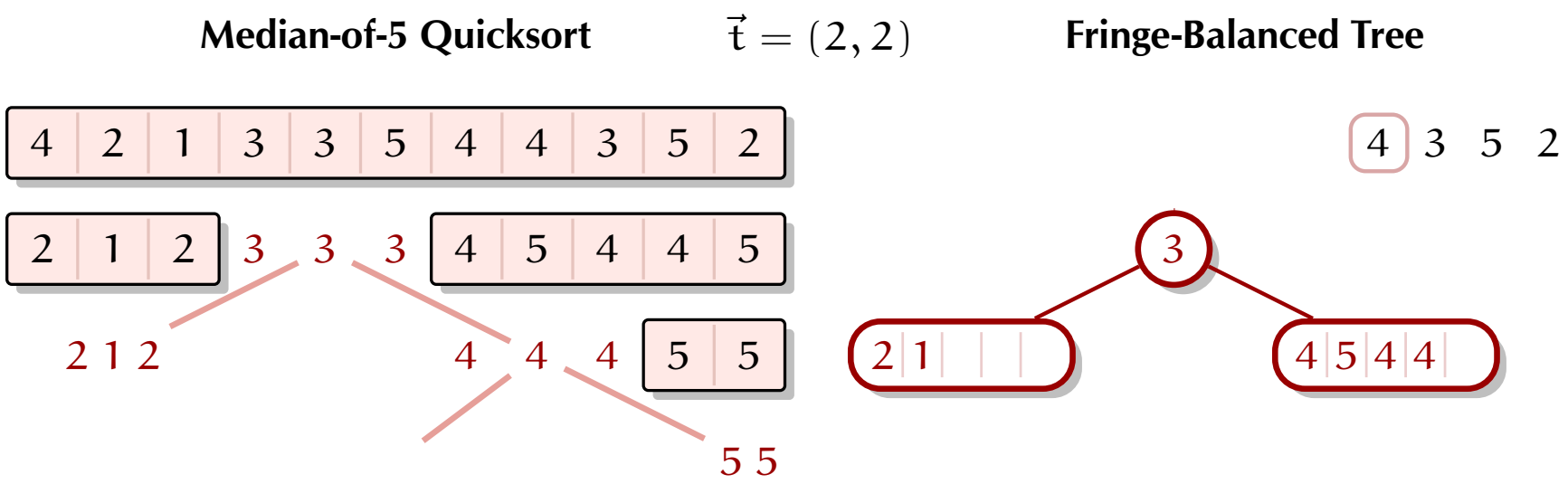
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

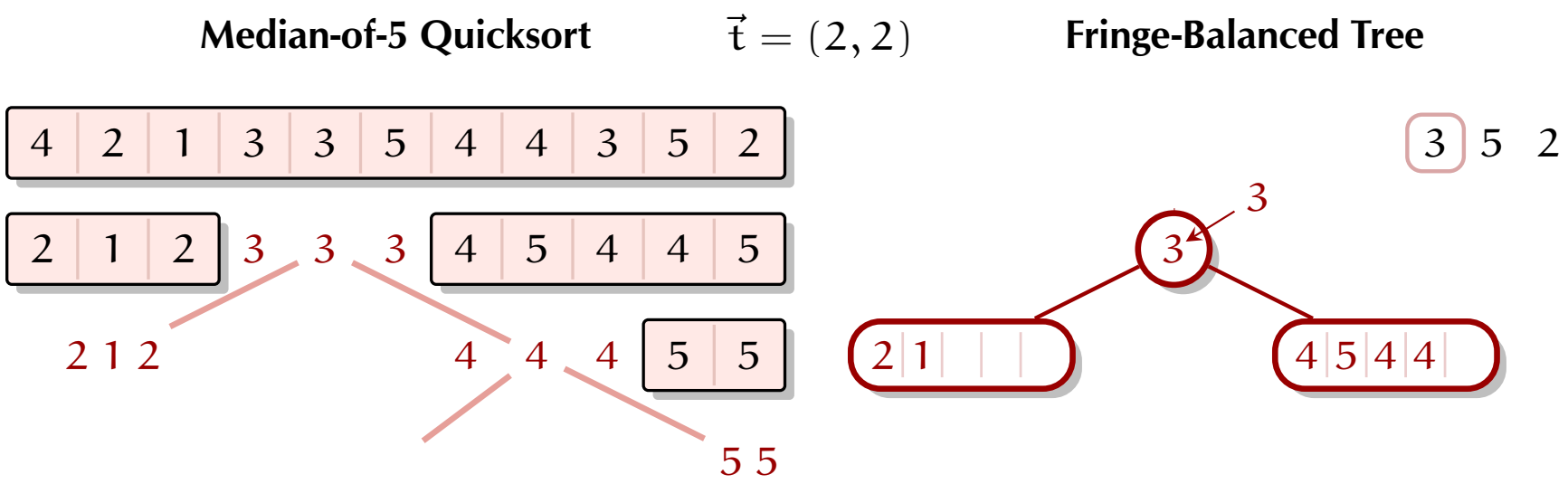
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

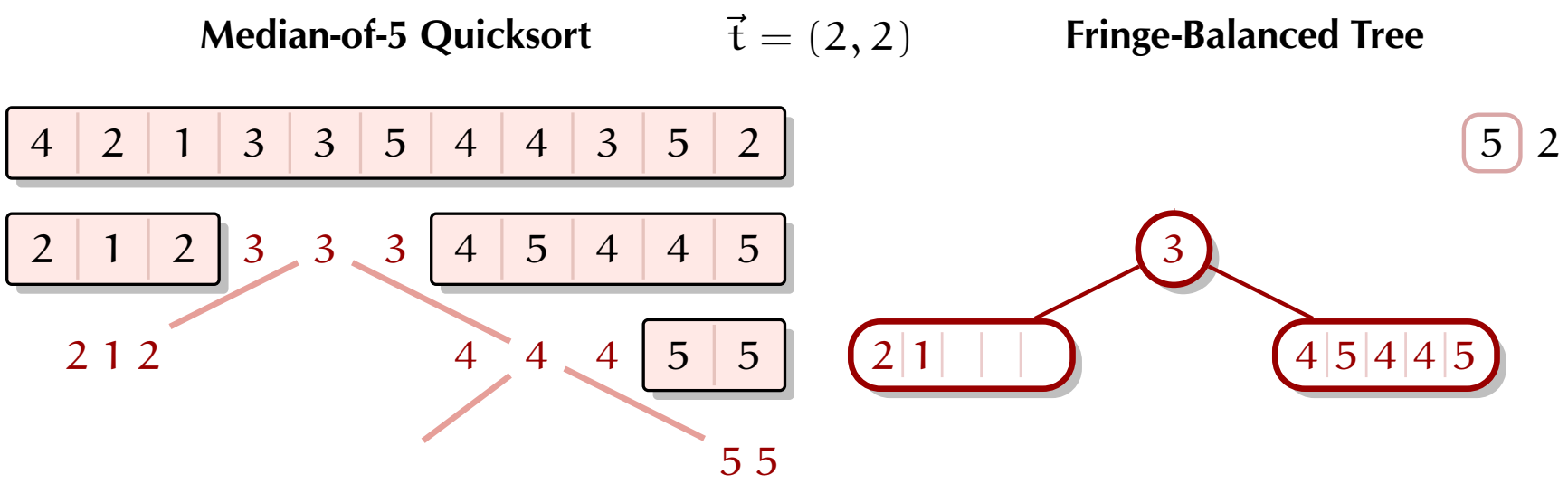
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

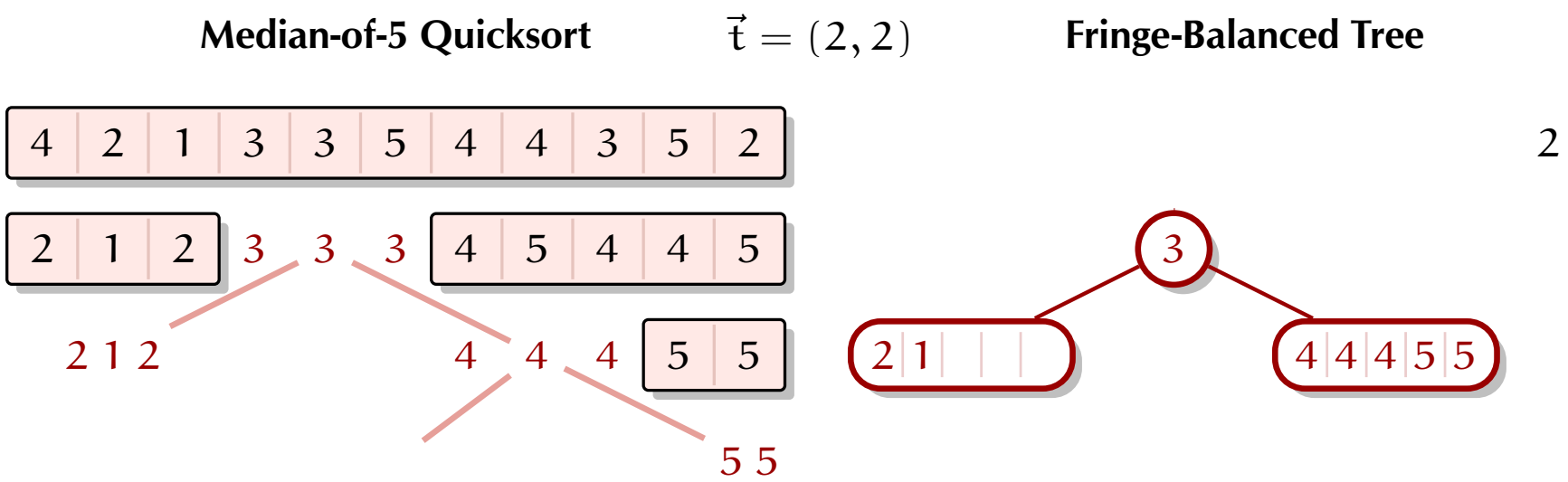
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

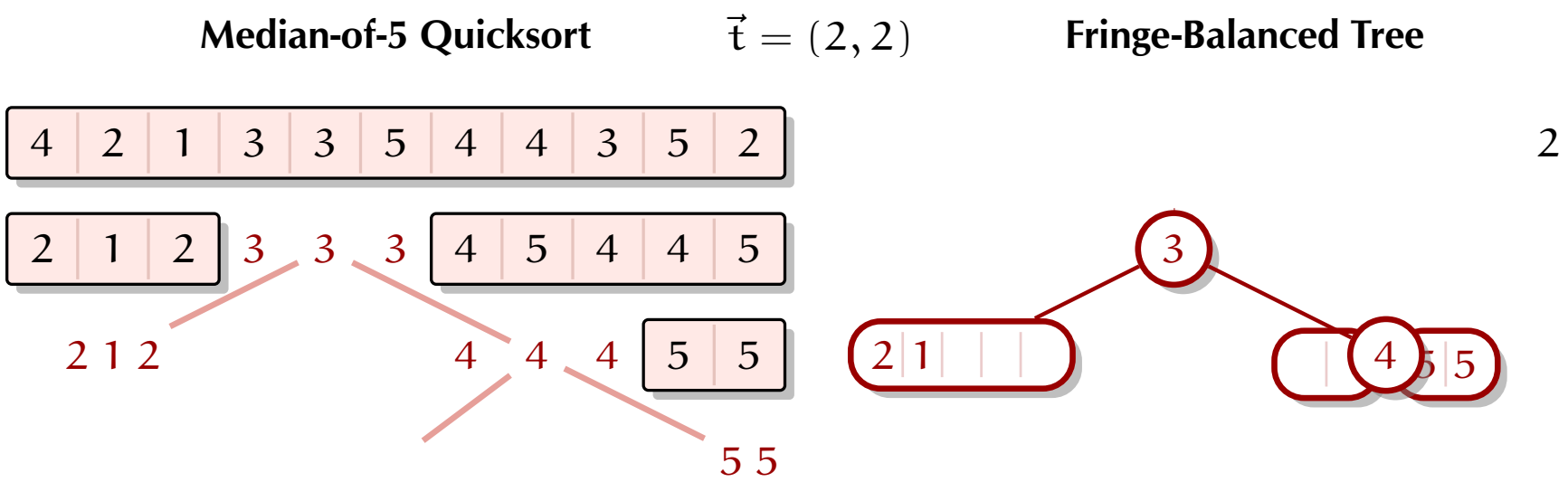
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

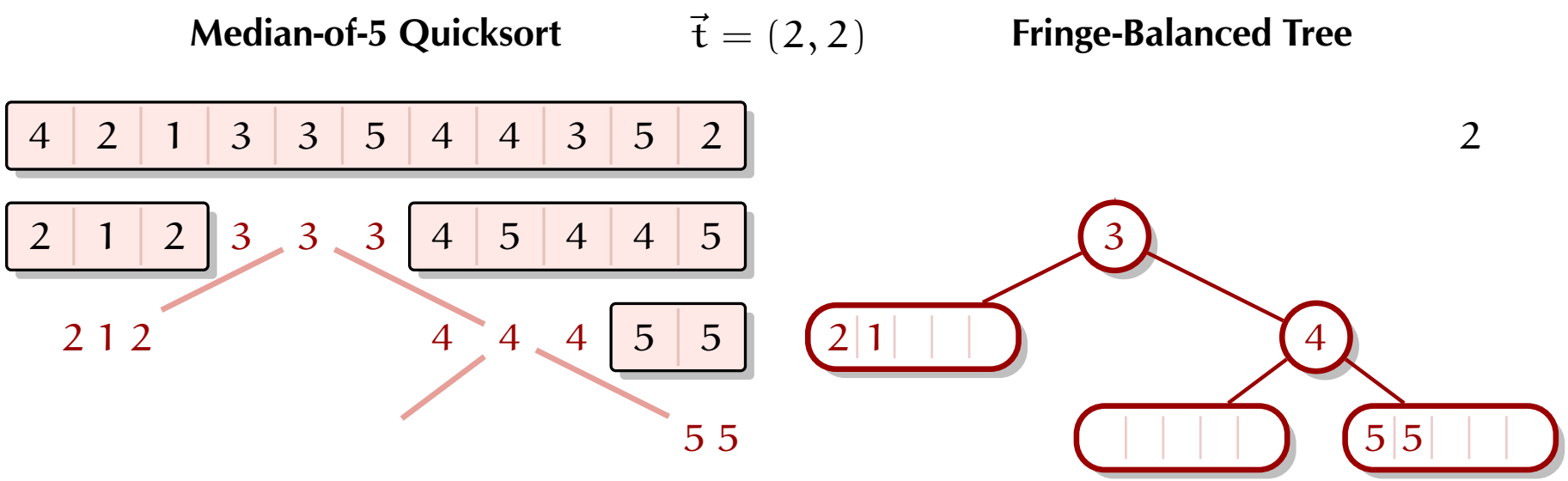
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

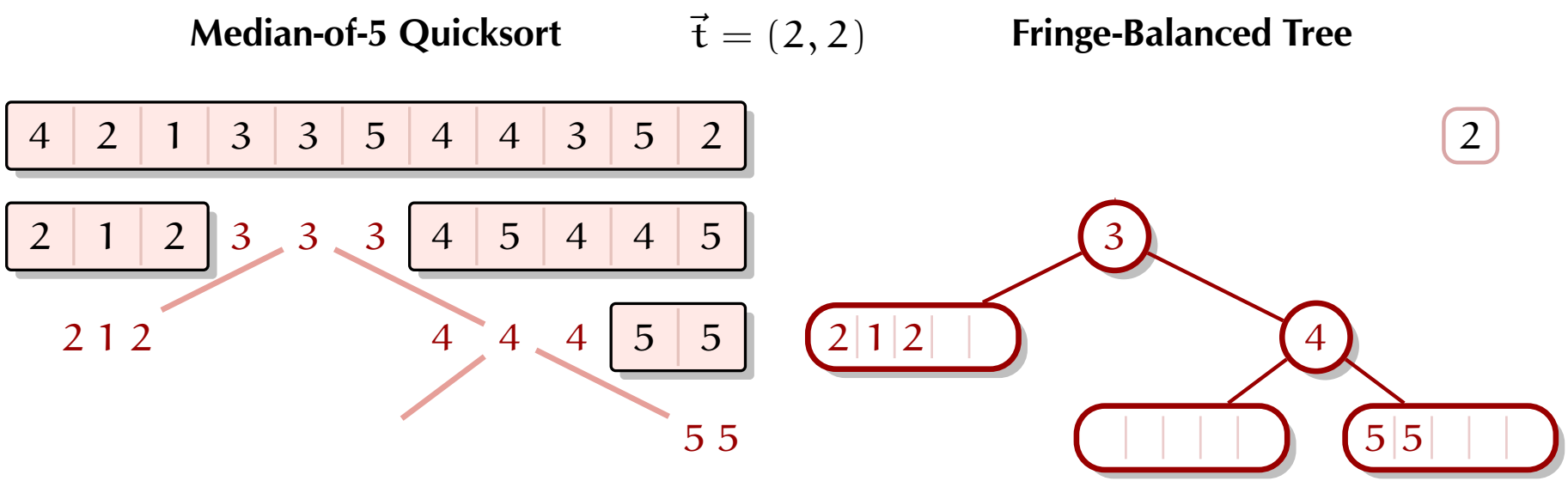
- Leaves have **buffer** for elements, to emulate pivot sampling



Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling



$\rightsquigarrow$ **Correspondence extends to**

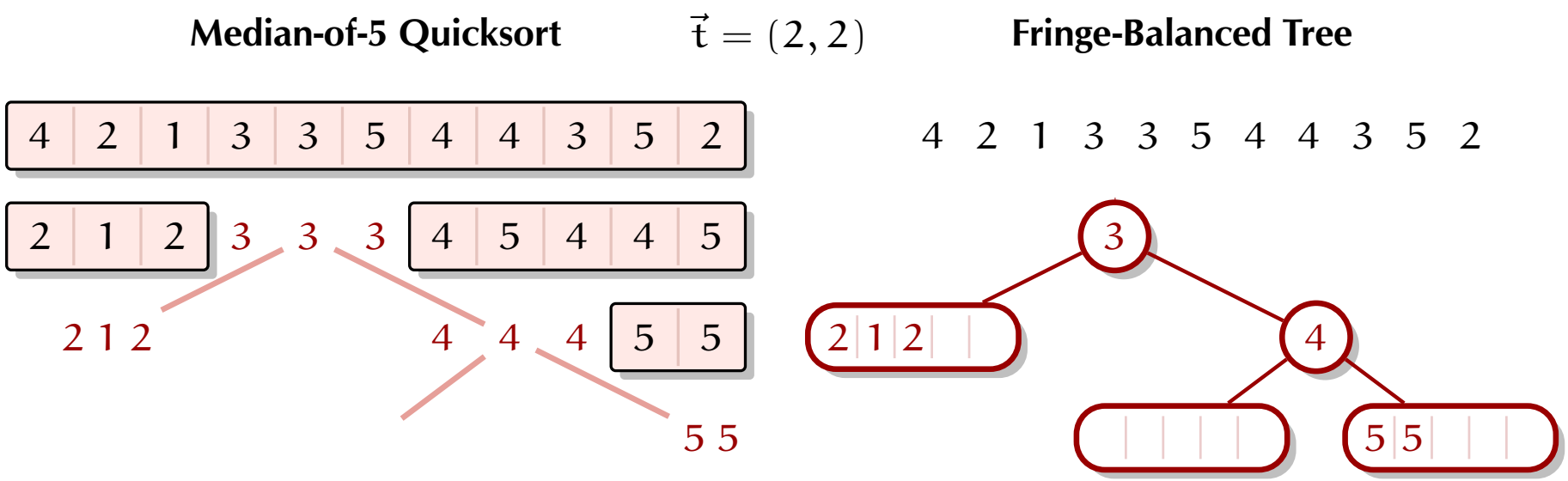
- Pivot Sampling (any scheme, not only median)
- Swap Partitioning vs. Fringe Search Trees

$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling



⇝ **Correspondence extends to**

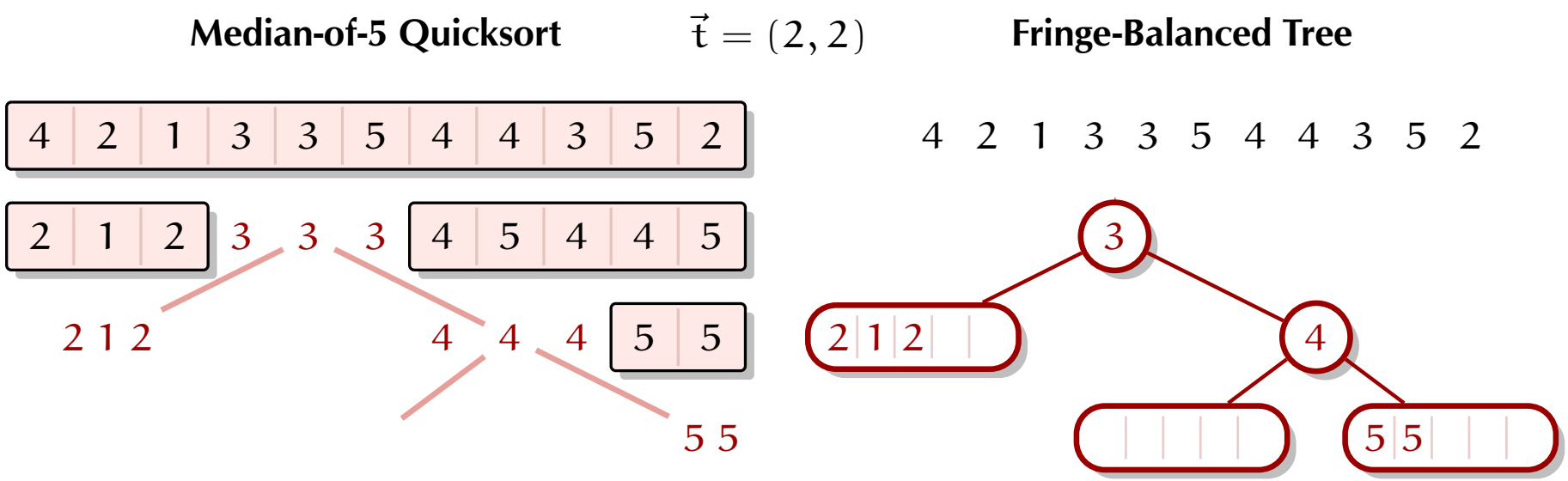
- Pivot Sampling (any scheme, not only median)
- **s-way Partitioning** ⇝ s-ary search trees

⇝ Analyze search trees instead of Quicksort.

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling



⇝ **Correspondence extends to**

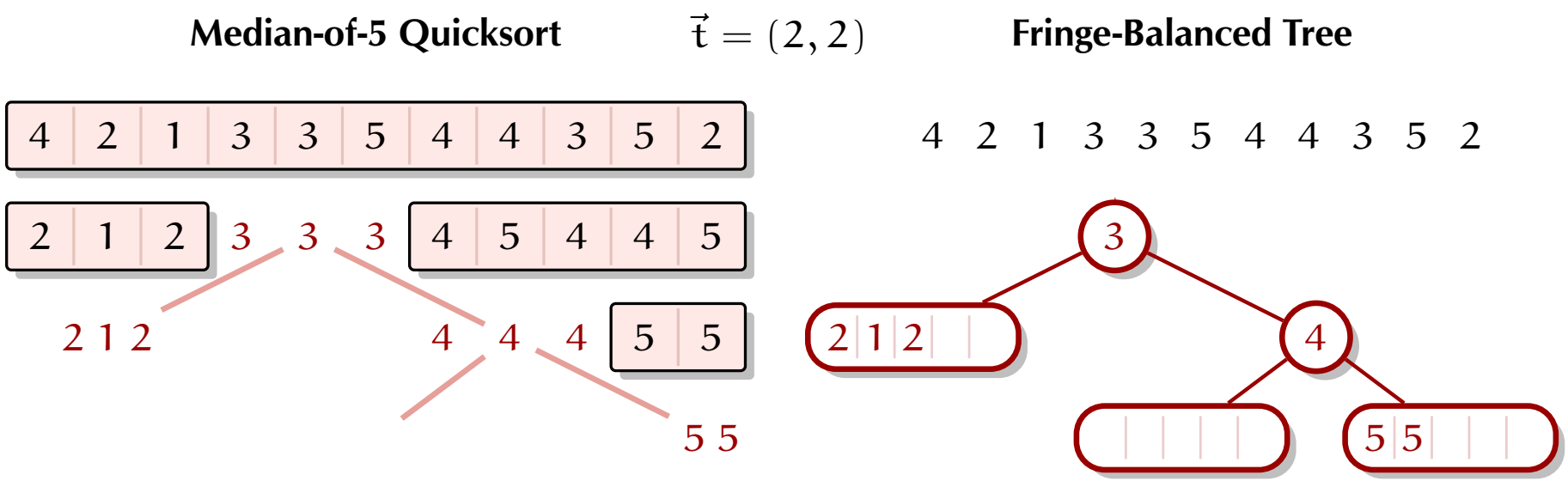
- Pivot Sampling (any scheme, not only median)
- **s-way Partitioning** ⇝ s-ary search trees

⇝ Analyze search trees instead of Quicksort.

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling



⇝ **Correspondence extends to**

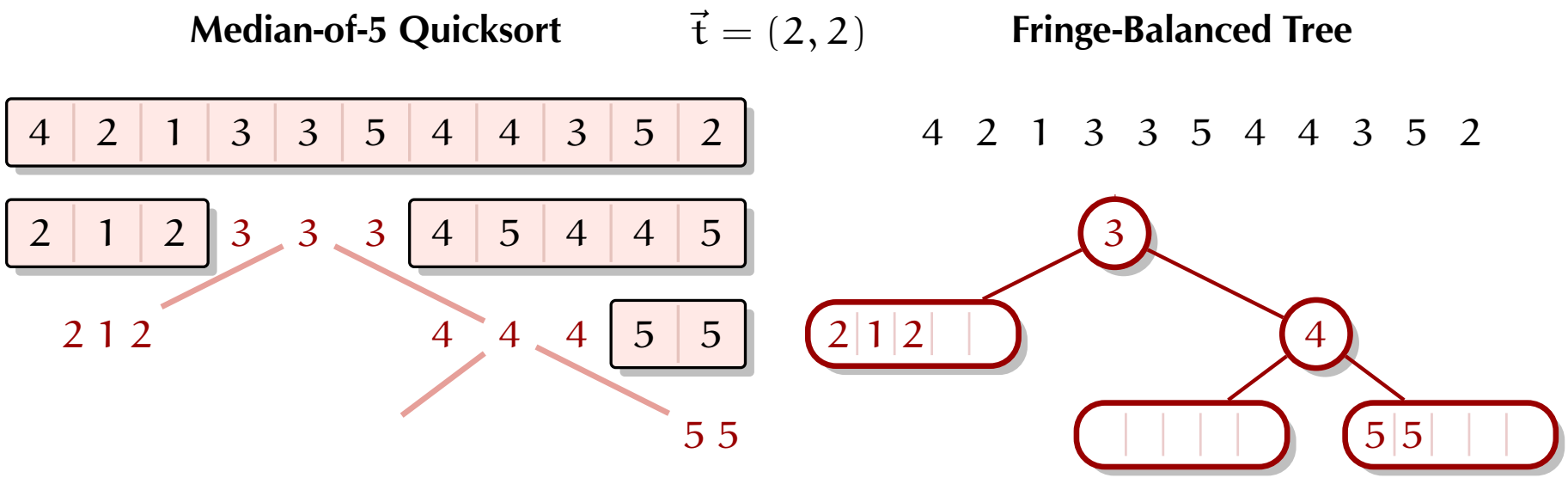
- Pivot Sampling (any scheme, not only median)
- **s-way Partitioning** ⇝ s-ary search trees

⇝ Analyze search trees instead of Quicksort.

Fringe-Balanced Trees

Fringe-Balanced Search Trees:

- Leaves have **buffer** for elements, to emulate pivot sampling



⇝ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- **s-way Partitioning** ⇝ s-ary search trees

⇝ Analyze search trees instead of Quicksort.

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4 5 5 5 2 3 5 4 2 2 2 5 5 5 4 2 5 1 5 4 2 3 4 2 2 1 2 4 2 2 5 5 2 3 4 2 3 4 1 3 2 2 1 4 4 4 1 2 3 3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4 5 5 5 2 3 5 4 2 2 2 5 5 5 4 2 5 1 5 4 2 3 4 2 2 1 2 4 2 2 5 5 2 3 4 2 3 4 1 3 2 2 1 4 4 4 1 2 3 3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4 5 5 5 2 3 5 4 2 2 2 5 5 5 4 2 5 1 5 4 2 3 4 2 2 1 2 4 2 2 5 5 2 3 4 2 3 4 1 3 2 2 1 4 4 4 1 2 3 3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4 5 5 5 2 3 5 4 2 2 2 5 5 5 4 2 5 1 5 4 2 3 4 2 2 1 2 4 2 2 5 5 2 3 4 2 3 4 1 3 2 2 1 4 4 4 1 2 3 3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- $\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3

Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with high probability

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- $\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with high probability

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

with leaf buffer size w

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with high probability

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with high probability

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with high probability

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

with leaf buffer size w

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

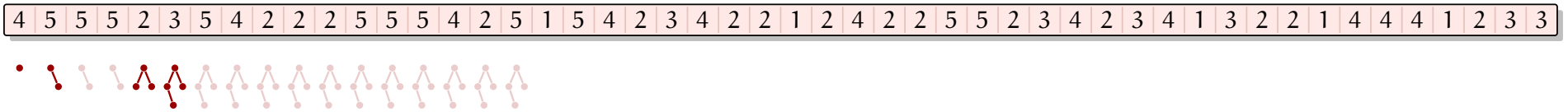
⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \ldots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \ldots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

⇝ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!

with leaf-ranks $1, \dots, n$

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - ⇝ direct analysis no simpler than for Quicksort

4	5	5	5	2	3	5	4	2	2	2	5	5	5	4	2	5	1	5	4	2	3	4	2	2	1	2	4	2	2	5	5	2	3	4	2	3	4	1	3	2	2	1	4	4	4	1	2	3	3



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with leaf-buffer size w

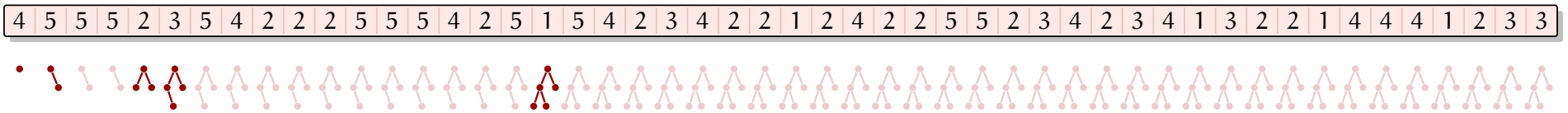
Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- $\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
  - $\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes stationary after 1 insert of each value!
with **leaf-buffer size** w

Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

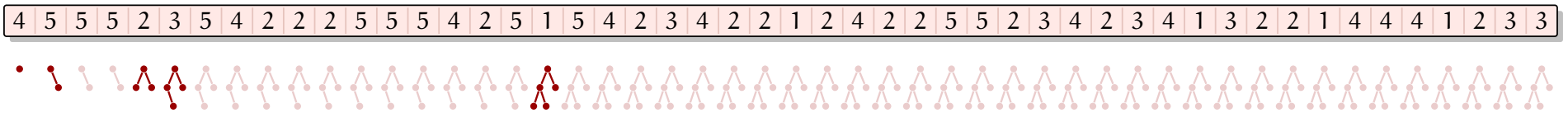
Tree-Growing and Searching

$\rightsquigarrow$ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort

($w = 1$)



Observation: $\mathcal{T}$ becomes stationary after w inserts of each value!
with **leaf-buffer size** w

Split input into **tree-growing** part and **searching** part:

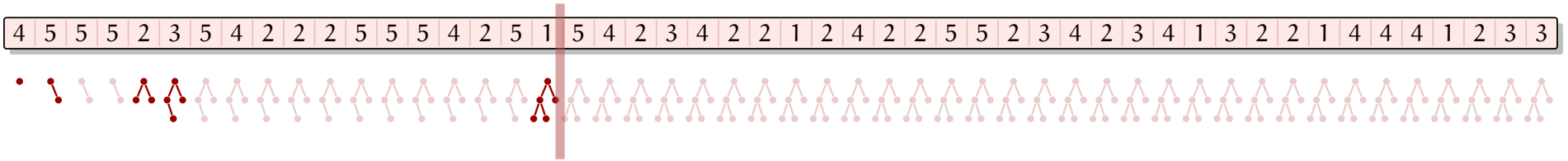
1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- ⇝ Quicksort costs = costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed ⇝ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
- ⇝ direct analysis no simpler than for Quicksort

($w = 1$)



Observation: $\mathcal{T}$ becomes stationary after w inserts of each value!
with **leaf-buffer size** w



Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

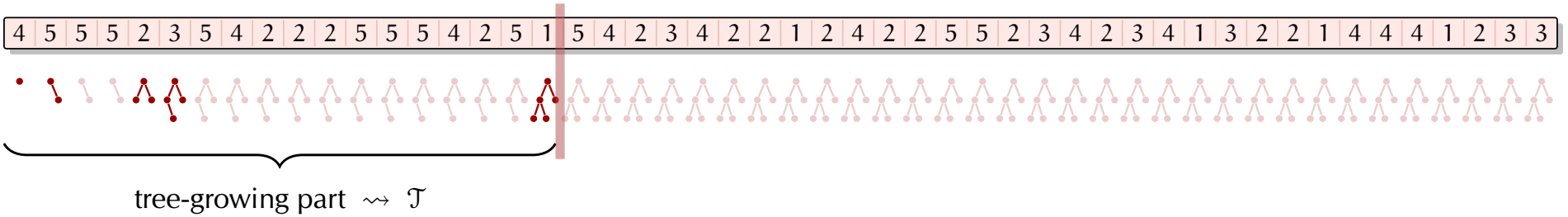
Tree-Growing and Searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort

($w = 1$)



Observation: $\mathcal{T}$ becomes stationary after w inserts of each value!
with **leaf-buffer size** w



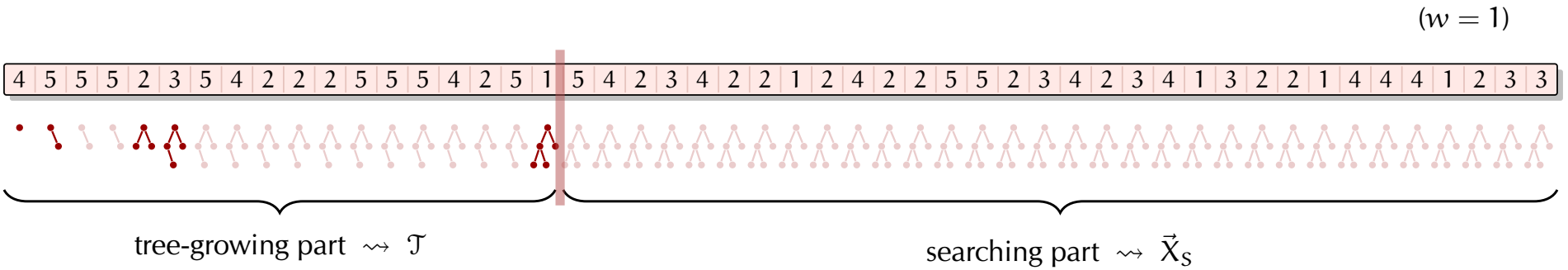
Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

- $\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.
  - $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
  - **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)
- $\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes stationary after w inserts of each value!
with **leaf-buffer size** w



Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

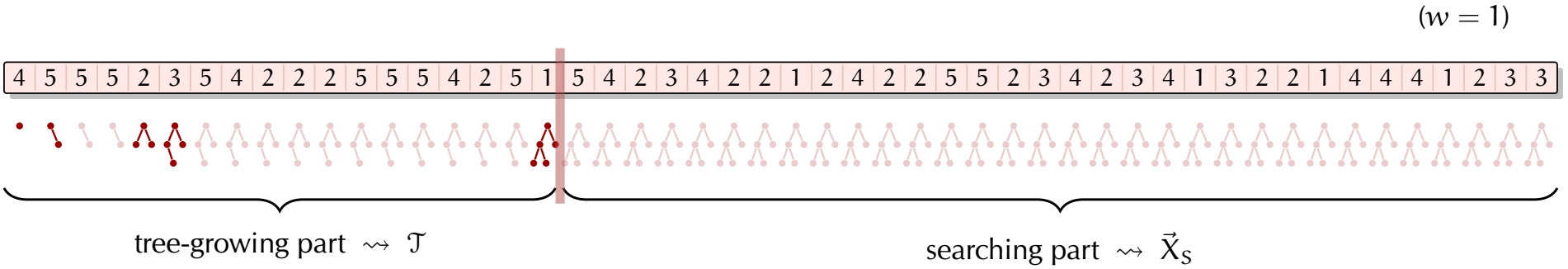
$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Tree-Growing and Searching

$\rightsquigarrow$ Quicksort costs $=$ costs to **search** input $\vec{U} = (U_1, \dots, U_n)$ in **final** tree $\mathcal{T}$.

- $\mathcal{T}$ fixed $\rightsquigarrow$ search cost depends only on **profile** $\vec{X} = (X_1, \dots, X_u)$
- **but:** $\mathcal{T}$ also depends on $\vec{U}$ (Recall: $\mathcal{T}$ is *built* from $\vec{U}$!)

$\rightsquigarrow$ direct analysis no simpler than for Quicksort



Observation: $\mathcal{T}$ becomes stationary after w inserts of each value!
with **leaf-buffer size** w



Split input into **tree-growing** part and **searching** part:

1. We built $\mathcal{T}$ until it is stationary, **ignoring costs**.
2. Determine costs of **searching** remaining elements.

$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.

- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\mathcal{T}$ not complete after n_T inserts, …
  - Have to make this happen rarely.
  - By Chernoff bounds, input is non-degenerate w.h.p. if $\boxed{u = O(n^{1/3})}$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

- α_u = expected search cost in random $\mathcal{T}$
- error term covers tree-growing, sorting samples, degenerate inputs

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.

- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.

$\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts, $\rightsquigarrow$ Have to make this happen rarely.

- By Chernoff bound, input is non-degenerate w.h.p. if $\boxed{u = \Theta(n^{1/3})}$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

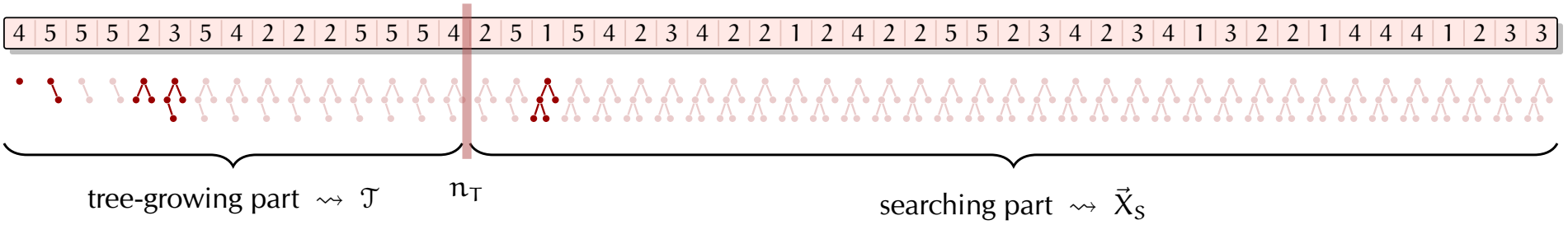
$\rightsquigarrow$ α_u = expected search cost in random $\mathcal{T}$

$\rightsquigarrow$ Error term covers the growing $\mathcal{T}$ and $\mathcal{T}$ sampling, degenerate inputs

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.

$\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts ...

$\rightsquigarrow$ Have to make this happen rarely.

- By Chernoff bounds, input is non-degenerate w.h.p. if $\boxed{u = O(n^{1-\varepsilon})}$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

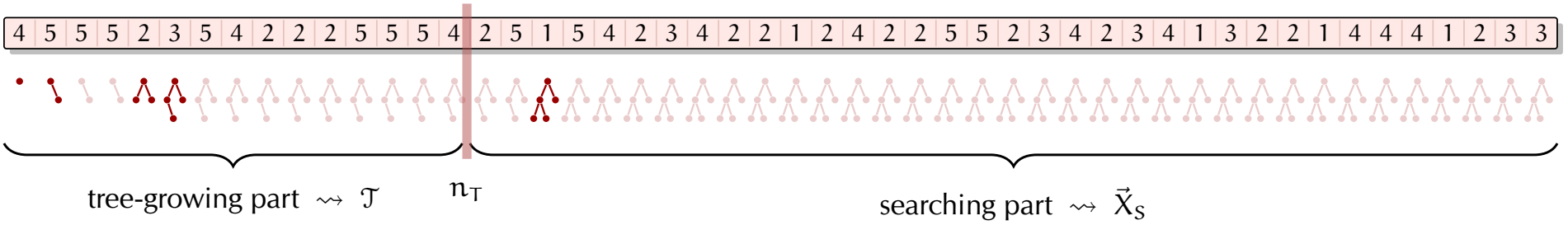
- α_u = expected search cost in random $\mathcal{T}$

- Error term covers the $\mathcal{T}$-growing, $\mathcal{T}$-sampling, degenerate inputs.

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts …
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

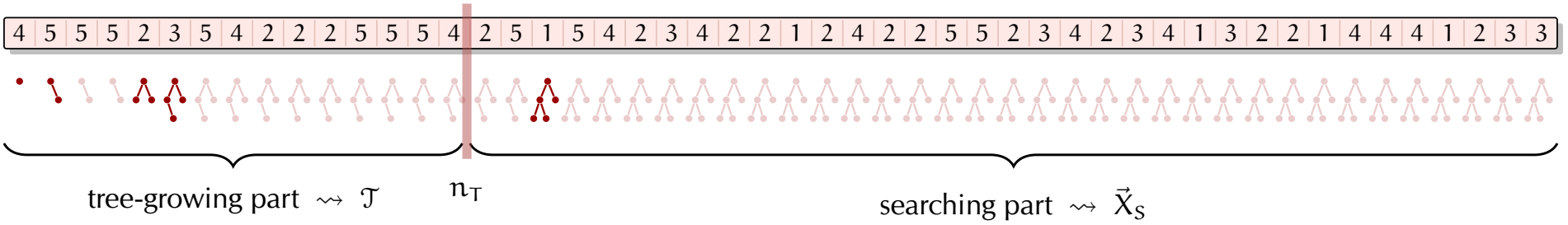
$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts …
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

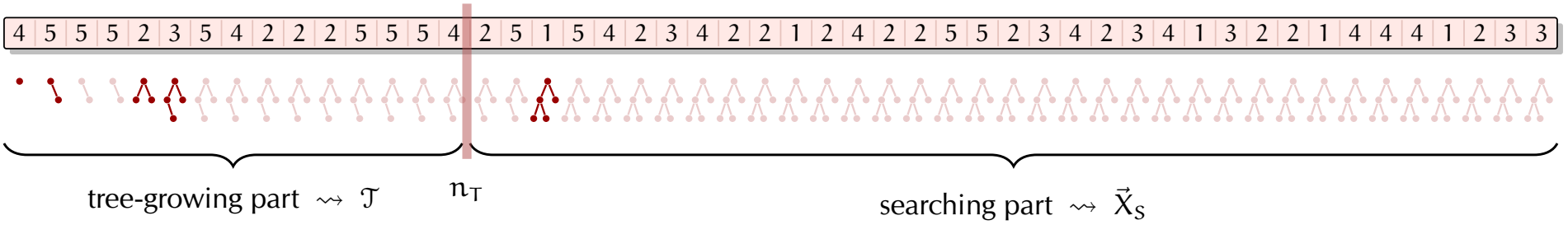
$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts …
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

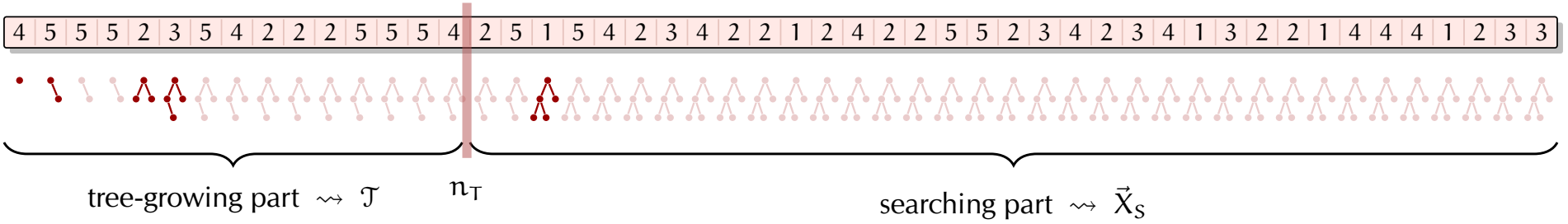
$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts …
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

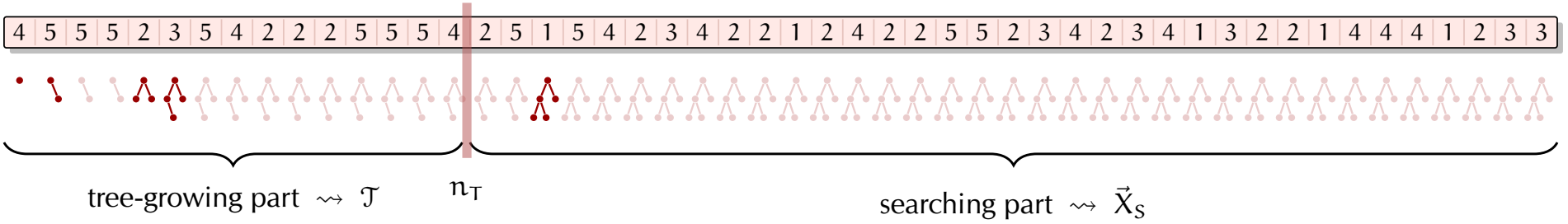
$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts ...
  - $\rightsquigarrow$ Have to make this happen rarely.
  - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $\boxed{u = O(n^{1/4})}$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

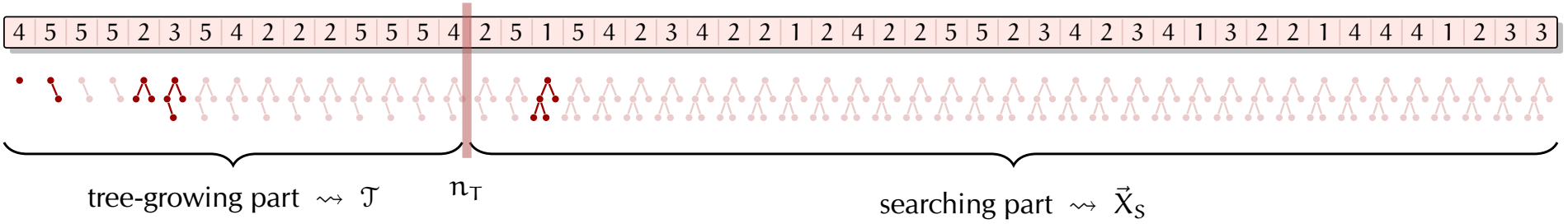
$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

- $\alpha_u =$ expected search cost in random $\mathcal{T}$
- [illegible]

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts … 
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

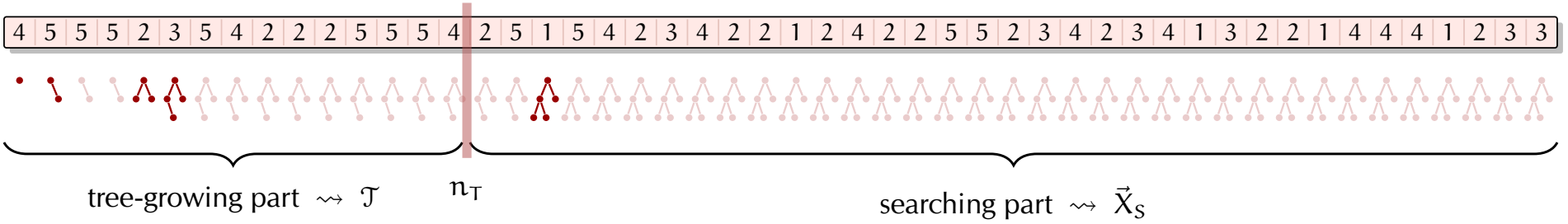
$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

- $\alpha_u =$ expected search cost in random $\mathcal{T}$
- **error term:** covers tree-growing, sorting samples, degenerate inputs

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts … 
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

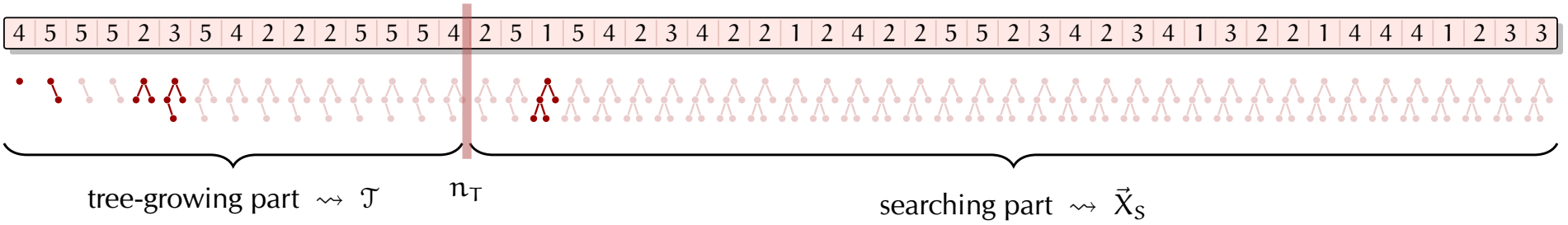
$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

- $\alpha_u =$ expected search cost in random $\mathcal{T}$
- **error term:** covers tree-growing, sorting samples, degenerate inputs

Bounding Tree-Growing Parts

*We ignore **tree-growing** in analysis* $\rightsquigarrow$ *have to make that part **short**.*

$\rightsquigarrow$ Allow first $n_T = \lceil n^{2/3} \rceil$ elements to build $\mathcal{T}$.



- Input is **degenerate** if a value occurs $< w$ times in first n_T elements.
  - $\rightsquigarrow$ $\mathcal{T}$ not complete after n_T inserts … 
  - $\rightsquigarrow$ Have to make this happen rarely.
    - By *Chernoff bounds*, input is **non-degenerate w.h.p.** if $u = O(n^{1/4})$

$\rightsquigarrow$ building $\mathcal{T}$ costs $\leqslant n_T \cdot u = O(n^{1-\varepsilon})$

$\rightsquigarrow$ Expected Quicksort cost $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

- $\alpha_u =$ expected search cost in random $\mathcal{T}$
- **error term:** covers tree-growing, sorting samples, degenerate inputs

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\quad$ ($u \cdot \alpha_u =$ path length of $\mathcal{T}$)

1. **Easy case:** $\vec{t} = (0, 0)$ $\quad \rightsquigarrow$ ordinary BST, $w = 1$

   $\rightsquigarrow$ no duplicates in tree

   $\rightsquigarrow$ $\mathcal{T}$ is BST under random permutation model

   Allen & Munro 1978
   Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:

   - duplicates inside leaves (during construction)
   - for $s \geq 3$: duplicates in same internal node possible

   $\rightsquigarrow$ $\mathcal{T}$ is not a random permutation

   - **But:** can set up a (messy) recurrence for α_u

   Sedgewick 2017
   Lecture Notes Lecture Notes for Linear Sorting Algorithms

   Asymptotic solution by Continuous Master Theorem

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \ldots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad$ ($u \cdot \alpha_u =$ path length of $\mathcal{T}$)

1. **Easy case:** $\vec{t} = (0, 0)$ $\qquad \rightsquigarrow$ ordinary BST, $w = 1$

   $\rightsquigarrow$ no duplicates in tree

   $\rightsquigarrow$ $\mathcal{T}$ is BST under random perm. Cutting model $\qquad$ Allen & Munro 1978

   Self-Adjusting Binary Search Trees

2. **General** $\vec{t} = (t_1, \ldots, t_s)$:

   - duplicates inside leaves (during construction)
   - for $s \geq 3$: duplicates in same internal node possible

   $\rightsquigarrow$ $\mathcal{T}$ is not a random permutation

   - **But:** can set up a (messy) recurrence for α_u $\qquad$ Sedgewick 2017

   Asymptotic solution by Continuous Master Theorem

   Lindsay and Mahmoud 2016

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\qquad \rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model

   Allen & Munro 1978:
   Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - $\rightsquigarrow$ duplicates inside leaves (during construction)
   - $\rightsquigarrow$ for $s \geq 2$: duplicates in same internal node possible
   - $\rightsquigarrow \not\sim$ random permutation
   - $\rightsquigarrow$ Knuth can set up a (messy) recurrence for α_u
   - $\rightsquigarrow$ asymptotic solution by Quellmalz, Mahmoud, Takacs

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\qquad \rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model

   Allen & Munro 1978:
   Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - $\rightsquigarrow$ duplicates inside leaves (during construction)
   - $\rightsquigarrow$ for $s \geq 2$: duplicates in same internal node possible
   - $\rightsquigarrow \not\equiv$ random permutation
   - $\rightsquigarrow$ Knuth can set up a (messy) recurrence for α_u
   - $\rightsquigarrow$ asymptotic solution by Quittner via Mellin Transform

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \ldots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad$ ($u \cdot \alpha_u =$ path length of $\mathcal{T}$)

1. **Easy case:** $\vec{t} = (0, 0)$ $\quad \rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \ldots, t_s)$:
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model

Allen & Munro 1978:
Self-Organizing Binary Search Trees

Sebastian Wild | Quicksort with Equal Keys | 2016-03-07 | 10 / 12

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad$ ($u \cdot \alpha_u =$ path length of $\mathcal{T}$)

1. **Easy case:** $\vec{t} = (0, 0)$ $\qquad \rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2(1 + \frac{1}{u})H_u - 3 = 2\ln(u) \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geq 3$: duplicates in same internal node possible
   - $\rightsquigarrow$ $\not\sim$ random permutation
   - But: can set up a (messy) recurrence for α_u
   - Asymptotic solution by Continuous Master Theorem

Sebastian Wild | Quicksort with Equal Keys | 2016-03-07 | 10 / 12

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2\big(1 + \frac{1}{u}\big)H_u - 3 = \mathbf{2\ln(u)} \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible

   $\rightsquigarrow \neq$ random permutation

   - **But:** can set up a (messy) **recurrence** for α_u

     asymptotic solution by ***Continuous Master Theorem***

3 | 3

Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Search Costs

Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2\big(1 + \frac{1}{u}\big)H_u - 3 = \mathbf{2\ln(u)} \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible

   $\rightsquigarrow \neq$ random permutation
   - **But:** can set up a (messy) **recurrence** for α_u

     asymptotic solution by ***Continuous Master Theorem***

3 | 3

Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Search Costs

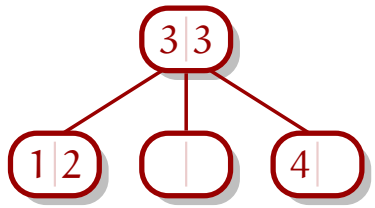
Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\qquad \rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2\big(1 + \frac{1}{u}\big)H_u - 3 = \mathbf{2\ln(u)} \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible
   - $\rightsquigarrow \neq$ random permutation
   - **But:** can set up a (messy) **recurrence** for α_u



asymptotic solution by ***Continuous Master Theorem***

Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Search Costs

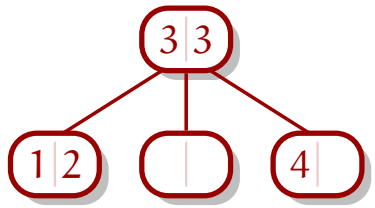
Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1,\dots,u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u}\sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad$ $(u\cdot\alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0,0)$ $\qquad\rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - ♻ $\alpha_u = 2(1+\frac{1}{u})H_u - 3 = \mathbf{2\ln(u)} \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1,\dots,t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible
   - $\rightsquigarrow \neq$ random permutation
   - **But:** can set up a (messy) **recurrence** for α_u



asymptotic solution by ***Continuous Master Theorem***

Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Search Costs

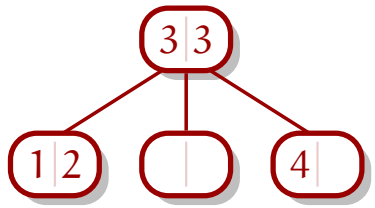
Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2\big(1 + \frac{1}{u}\big)H_u - 3 = 2\ln(u) \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible
   - $\rightsquigarrow$ $\neq$ random permutation
   - **But:** can set up a (messy) **recurrence** for α_u



asymptotic solution by ***Continuous Master Theorem***

Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Search Costs

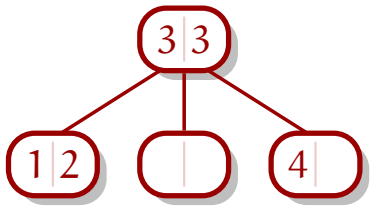
Recall: $\mathcal{T}$ built from inserting i.i.d. uniform $\{1, \dots, u\}$, until **saturation**.

- $\alpha_u = \frac{1}{u} \sum_{v=1}^{u}$ cost of searching v in $\mathcal{T}$ $\qquad (u \cdot \alpha_u =$ path length of $\mathcal{T})$

1. **Easy case:** $\vec{t} = (0, 0)$ $\rightsquigarrow$ ordinary BST, $w = 1$
   - $\rightsquigarrow$ **no duplicates** in tree
   - $\rightsquigarrow$ $\mathcal{T}$ is BST under **random permutation** model
   - $\alpha_u = 2\big(1 + \frac{1}{u}\big)H_u - 3 = \mathbf{2\ln(u)} \pm O(1)$

Allen & Munro 1978:
Self-Organizing Binary Search Trees

2. **General** $\vec{t} = (t_1, \dots, t_s)$:
   - duplicates inside leaves (during construction)
   - for $s \geqslant 3$: duplicates in same internal node possible
   - $\rightsquigarrow$ $\neq$ random permutation
   - **But:** can set up a (messy) **recurrence** for α_u
   - asymptotic solution by ***Continuous Master Theorem***



Roura 2001:
Improved Master Theorems for Divide-and-Conquer Recurrences

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $u \to \infty$
  - $\mathcal{H} = \sum_{i=1}^{u} \frac{b_i + 1}{k+1}(H_{b_i+1} - H_{a_i-1})$
  - $k =$ sample size
  - $\bar{a} =$ avg. search costs inside node
    ($\bar{a} = 1$ for binary trees)

- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ 

$u \to \infty$

- $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$

  k = sample size
- $\bar{a}$ = avg. search costs inside node

  ($\bar{a} = 1$ for binary trees)

- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$

$u = O(n^{1/4})$

Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$  $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a} = 1$ for binary trees)

- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$  $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a} = 1$ for binary trees)

- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\boxed{u \to \infty}$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $\boxed{u = O(n^{1/4})}$

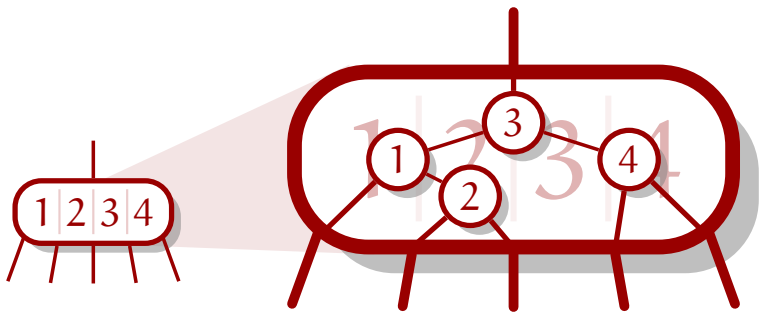
Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$  $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

Time to put the pieces together!

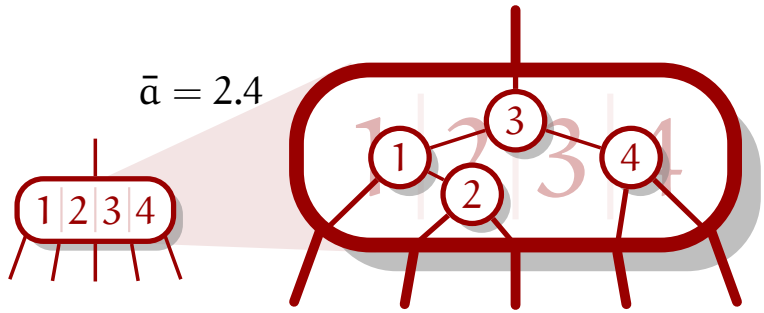
Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $u \to \infty$

  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

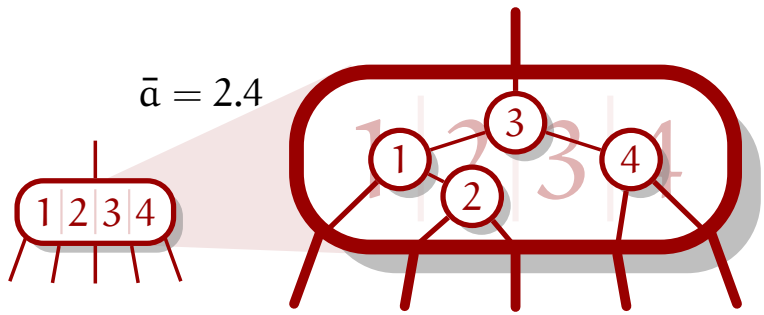
Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs:** $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

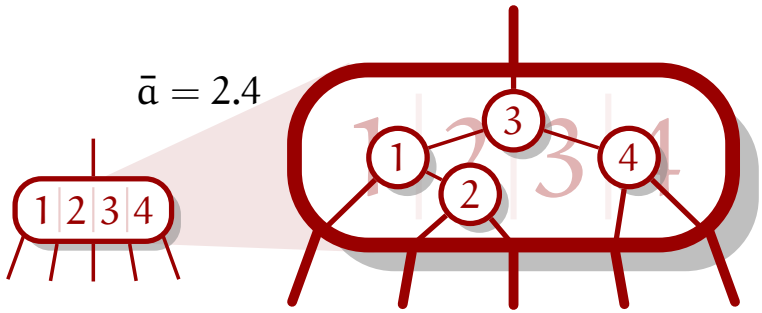
Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$  $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs**: $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$

Time to put the pieces together!

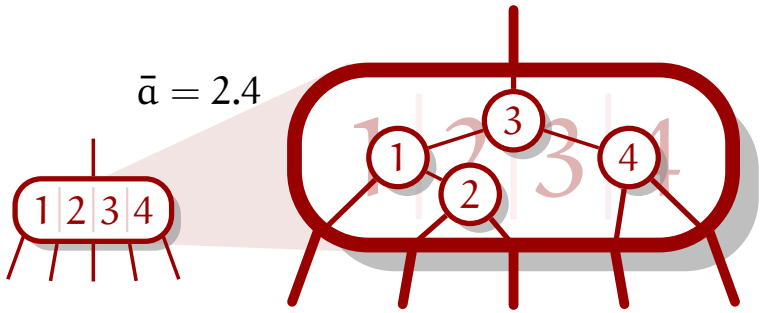
Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Permutations: $\mathbb{E}[C_n] = \frac{2}{\mathcal{H}} n \ln(n) \pm O(n)$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$  $u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs**: $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $u = O(n^{1/4})$



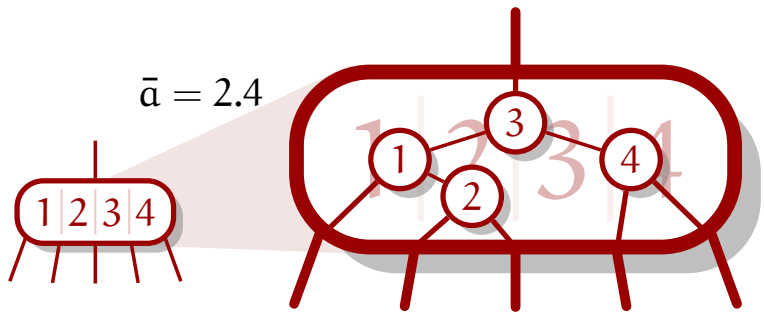
Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $u = \omega(1), u = O(n^{1/4})$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\quad u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs**: $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $\quad u = O(n^{1/4})$

Time to put the pieces together!

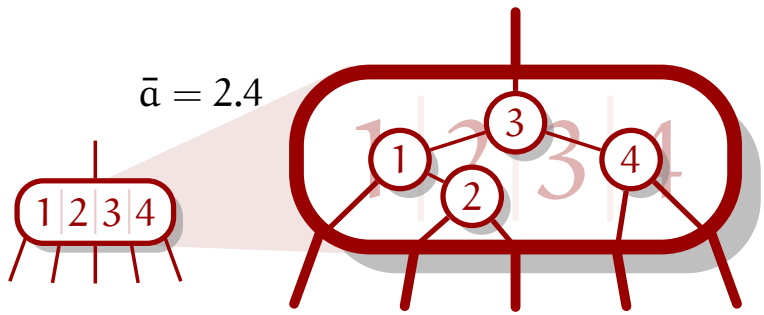
Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $\quad u = \omega(1), u = O(n^{1/4})$

Permutations: $\mathbb{E}[C_n] = \frac{\bar{a}}{\mathcal{H}} n \ln(n) \pm O(n)$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\quad u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs**: $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $\quad u = O(n^{1/4})$

Time to put the pieces together!

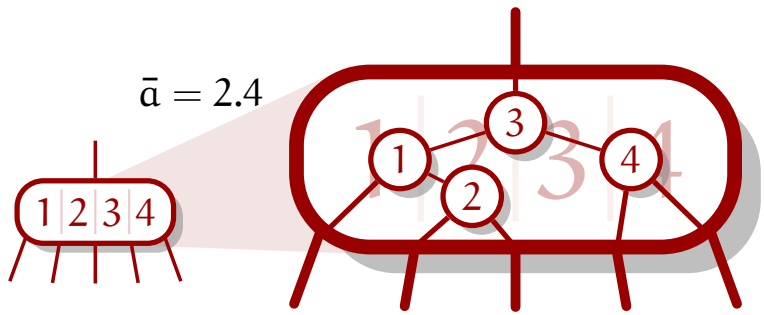
Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $\quad u = \omega(1), u = O(n^{1/4})$

Permutations: $\mathbb{E}[C_n] = \frac{\bar{a}}{\mathcal{H}} n \ln(n) \pm O(n)$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\quad u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node
    ($\bar{a}$ = 1 for binary trees)



- **Quicksort costs**: $\mathbb{E}[C_{n,u}] = \alpha_u \cdot n \pm O(n^{1-\varepsilon})$ $\quad u = O(n^{1/4})$

Time to put the pieces together!

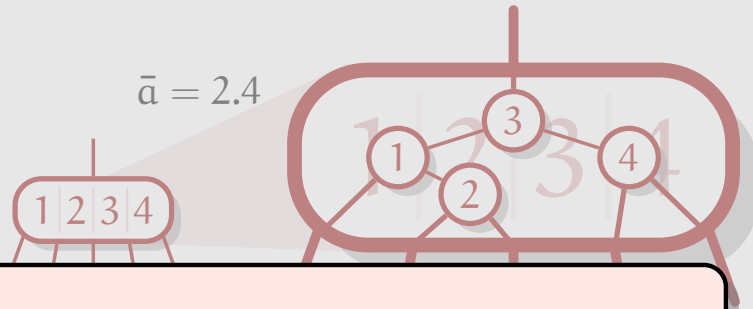
Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $\quad u = \omega(1), u = O(n^{1/4})$

Permutations: $\mathbb{E}[C_n] = \frac{\bar{a}}{\mathcal{H}} n \ln(n) \pm O(n)$

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\quad u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node



Results deceptively similar! (derivation isn't)

- Same speedup from **sampling** and **multi-way partitioning**.

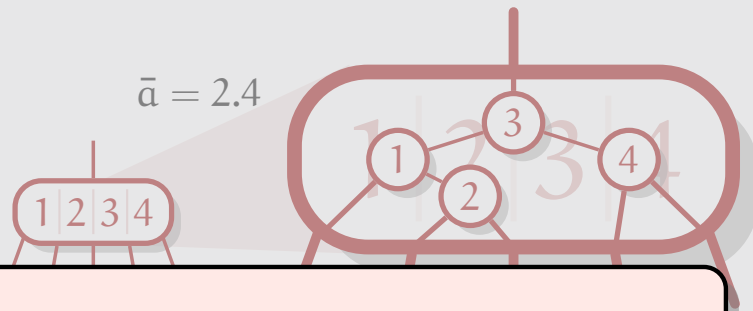
Time to put the pieces together!

Quicksort Costs

u-ary words:	$\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$	$u = \omega(1), u = O(n^{1/4})$
Permutations:	$\mathbb{E}[C_n] = \frac{\bar{a}}{\mathcal{H}} n \ln(n) \pm O(n)$	

Result

- **Search costs** in $\mathcal{T}$: $\alpha_u = \frac{\bar{a}}{\mathcal{H}} \ln(u) \pm O(1)$ $\quad u \to \infty$
  - $\mathcal{H} = \sum_{r=1}^{s} \frac{t_r + 1}{k+1} (H_{k+1} - H_{t_r+1})$
    k = sample size
  - $\bar{a}$ = avg. search costs inside node



- C

Results deceptively similar! (derivation isn't)

Same speedup from **sampling** and **multi-way partitioning**.

Time to put the pieces together!

Quicksort Costs

u-ary words: $\mathbb{E}[C_{n,u}] = \frac{\bar{a}}{\mathcal{H}} n \ln(u) \pm O(n)$ $\quad u = \omega(1), u = O(n^{1/4})$

Permutations: $\mathbb{E}[C_n] = \frac{\bar{a}}{\mathcal{H}} n \ln(n) \pm O(n)$

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.

Limitations:

- restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
- currently only stateless cost measures

- Partial Answer to conjecture of Sedgewick & Bentley:
  Median-of-k Quicksort approaches lower bound for $k \to \infty$ for various distributions

Finally cleared the item from my to-do list.

Future Work

- other discrete distributions $\rightsquigarrow$ expected profile model
- better asymptotic approximation for $u = n$

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:
  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*

Finally cleared the item from my to-do list

Future Work

- *Other discrete distributions* $\rightsquigarrow$ *expected-profile model*
- *Better asymptotic approximation for* $\mathbb{E}[\ldots]$

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*

Finally cleared the item from my to-do list

Future Work

- *other discrete distributions* ⇝ *expected-profile model*
- *Better asymptotic approximation for ε ...*

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:
  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*

Finally cleared the item from my to-do list

Future Work

- other discrete distributions ⇝ expected profile method
- better asymptotic approximation for small n

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty \ldots$ for **uniform** distribution.*

Finally cleared the item from my to-do list

Future Work

- *Other discrete distributions* $\rightsquigarrow$ *Expected-profile model*
- *Better asymptotic approximation for* $\mathbb{E}[\ldots]$

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*



Finally cleared the item from my to-do list

Future Work

- other discrete distributions $\rightsquigarrow$ *expected-profile model*
- better asymptotic approximation for n

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*



Finally cleared the item from my to-do list ... or almost so

Future Work

- other discrete distributions $\rightsquigarrow$ *expected-profile model*
- better asymptotic approximation for α_u

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*



Finally cleared the item from my to-do list ... or almost so

Future Work

- other discrete distributions $\rightsquigarrow$ *expected-profile model*
- better asymptotic approximation for α_u

Conclusion

Findings

- First analysis of generalized Quicksort (sampling & multi-way) on equal keys.
  Limitations:
  - restricted range for universe size: $u = \omega(1)$ and $u = O(n^{1/3-\varepsilon})$
  - currently only *stateless* cost measures
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  *Median-of-k Quicksort approaches lower bound for $k \to \infty$ ... for **uniform** distribution.*



Finally cleared the item from my to-do list ... or almost so

Future Work

- other discrete distributions $\rightsquigarrow$ *expected-profile model*
- better asymptotic approximation for α_u



Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Java 7
- ne
- ou
- m

Q

Track Record of Dual-Pivot Quicksort

- **Observation in practice:**
  Yaroslavskiy's Quicksort (YQS) **faster** than *classic Quicksort (CQS)* ... **why?**

⇝ I did a mathematical analysis of YQS.

- **Traditional** cost measures do **not** explain observation!

	CQS	YQS	Relative
Running Time	(from various experiments)		$-10\pm2\%$
Comparisons	2	1.9	-5%
Swaps	$0.\overline{3}$	0.6	$+80\%$
Bytecode Instructions	18	21.7	$+20.6\%$
MMIX oops υ	11	13.1	$+19.1\%$
MMIX mems μ	$2.\overline{6}$	2.8	$+5\%$
Scanned Elements	2	1.6	-20%
Branch Mispredictions	0.57	0.58	$+2\%$

$\cdot n \ln n + O(n)$, *average case results*

Only plausible explanation for running time: 20% less memory transfers in YQS.

Cache Behavior Analysis – Results

Let M be the size of the cache and B be the size of each block of cache.

1-Pivot Quicksort: $2\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right) + O(\frac{n}{B})$

2-Pivot Quicksort: $\frac{8}{5}\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right) + O(\frac{n}{B})$

Leading constants of 2 and 1.6 for cache faults versus 2 and 1.9 for comparisons.

Lopez-Ortíz 2014: *Multi-Pivot Quicksort: Theory and Experiments*

Scanned Elements

Cache Behavior Analysis – Results

Let M be the size of the cache and B be the size of each block of cache.

1-Pivot Quicksort: $2\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right)+O(\frac{n}{B})$

2-Pivot Quicksort: $\frac{8}{5}\left(\frac{n+1}{B}\right)\ln\left(\frac{n+1}{M+2}\right)+O(\frac{n}{B})$

Leading constants of 2 and 1.6 for cache faults versus 2 and 1.9 for comparisons.

Lopez-Ortíz 2014: *Multi-Pivot Quicksort: Theory and Experiments*

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- Wegner 1980: *Quicksort for Equal Keys*
- Bentley & McIlroy 1993: *Engineering a Sort Function*
- Dual-Pivot Quicksort: *Scanned elements for $\mathcal{U}_{\mathcal{D}}$ → $\mathcal{S}_{\mathcal{D}}$*

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$
?	
≤ ? ≥	

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- Wegner 1980: Quicksort for Equal Keys
- Bentley & McIlroy 1993: Engineering a Sort Function
- Dual-Pivot Quicksort: Sorting Method for Oracle Java 7

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$
$\leq$? $\geq$	$3.0\, n \ln(u) \pm O(n)$
$\leq$? $\geq$	$3.0\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for # **memory transfers** and # **cache misses**

Fat-Pivot Partitioning

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

Wegner 1980: *Quicksort for Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

⇝ **same # scanned elements** for $u \to \infty$

Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

⇝ **same # scanned elements** for $u \to \infty$

Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
$< \mid = \mid ? \mid >$	$3.0\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

$\rightsquigarrow$ **same # scanned elements** for $u \to \infty$

Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$
= < ? >	$2.0\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

$\rightsquigarrow$ **same # scanned elements** for $u \to \infty$

Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$
= < ? > =	$2.0\, n \ln(u) \pm O(n)$
=P <P P<∘<Q ? >Q =Q	$1.6\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

$\rightsquigarrow$ **same # scanned elements** for $u \to \infty$

Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	**Scanned elements** w/o sampling
< = ? >	$3.0\, n \ln(u) \pm O(n)$
= < ? > =	$2.0\, n \ln(u) \pm O(n)$
=P <P P<∘<Q ? >Q =Q	$1.6\, n \ln(u) \pm O(n)$

Scanned Elements

Java 7's Dual-Pivot Quicksort



- new default sorting method since Java 7
- outperforms classic Quicksort on permutations by 10% in practice
- most probably because of **memory hierarchy** effects

Scanned Elements

- *"total distance traveled by pointers"*
- simple to analyze
- approximation for **# memory transfers** and **# cache misses**

Fat-Pivot Partitioning

- collect equals at border, swap in at end

$\rightsquigarrow$ **same # scanned elements** for $u \to \infty$

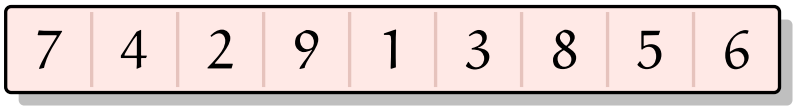
Wegner 1980: *Quicksort For Equal Keys*
Bentley & McIlroy 1993: *Engineering a Sort Function*

Partitioning Scheme	Scanned elements w/o sampling
$< \mid = \mid ? \mid >$	$3.0\, n \ln(u) \pm O(n)$
$= \mid < \mid ? \mid > \mid =$	$2.0\, n \ln(u) \pm O(n)$
$=P \mid <P \mid P<\circ<Q \mid ? \mid >Q \mid =Q$	$1.6\, n \ln(u) \pm O(n)$

Icons made by *Freepik* from `www.flaticon.com`.

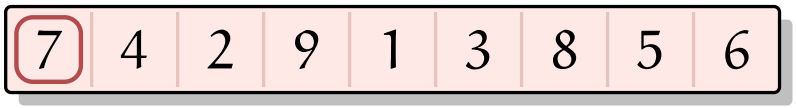
Quicksort & Search Trees (Distinct Keys)

Quicksort



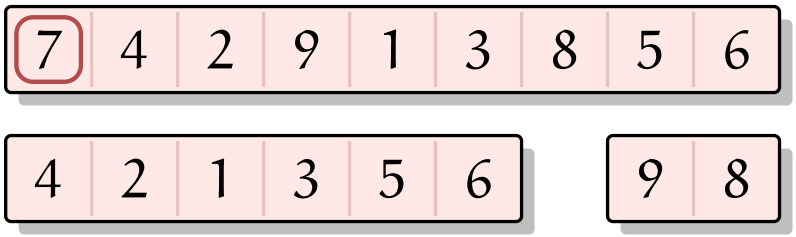
Quicksort & Search Trees (Distinct Keys)

Quicksort



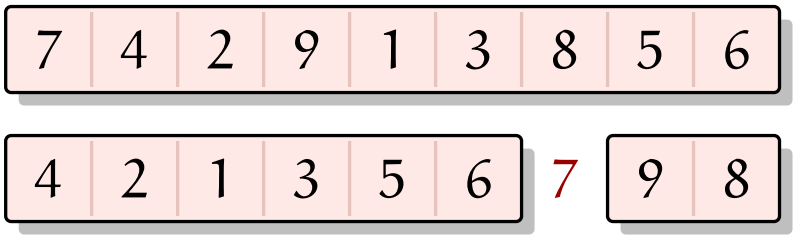
Quicksort & Search Trees (Distinct Keys)

Quicksort



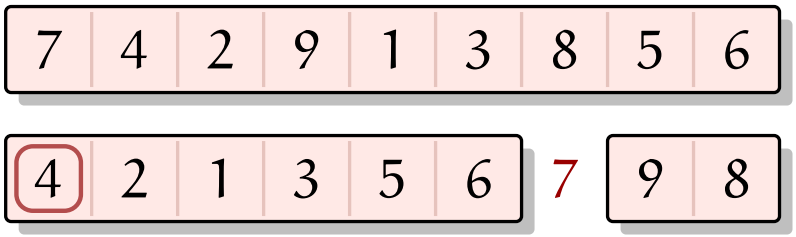
Quicksort & Search Trees (Distinct Keys)

Quicksort



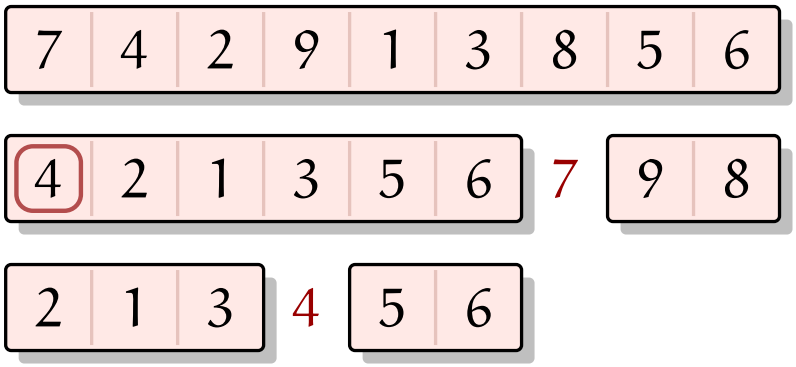
Quicksort & Search Trees (Distinct Keys)

Quicksort



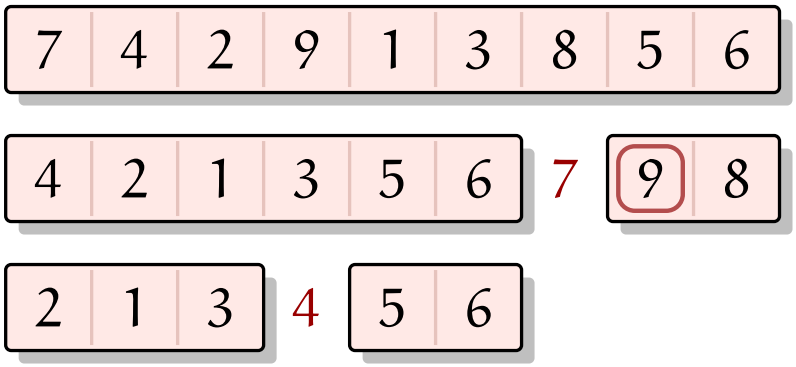
Quicksort & Search Trees (Distinct Keys)

Quicksort



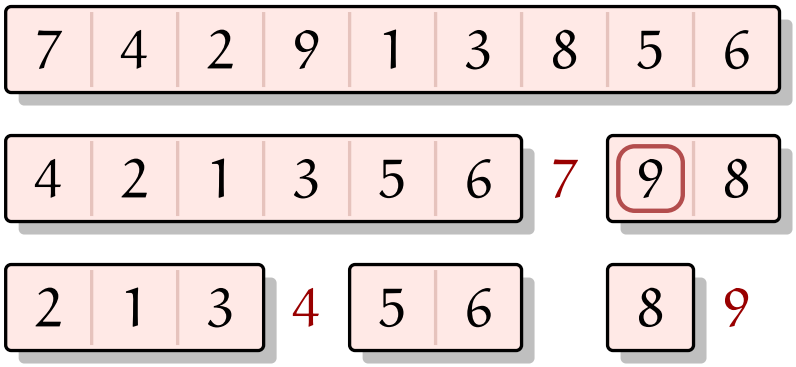
Quicksort & Search Trees (Distinct Keys)

Quicksort



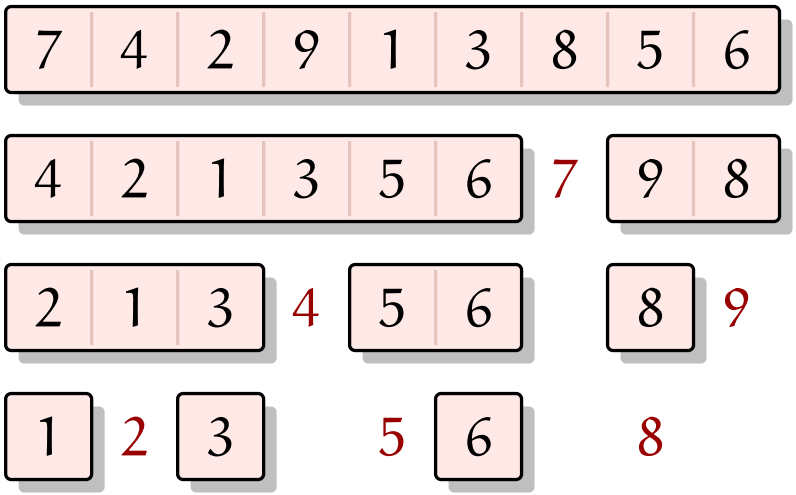
Quicksort & Search Trees (Distinct Keys)

Quicksort



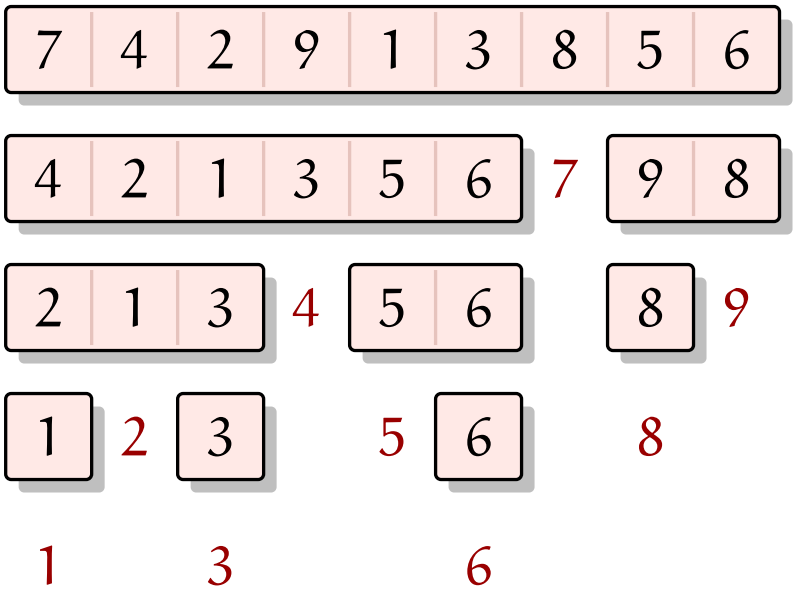
Quicksort & Search Trees (Distinct Keys)

Quicksort

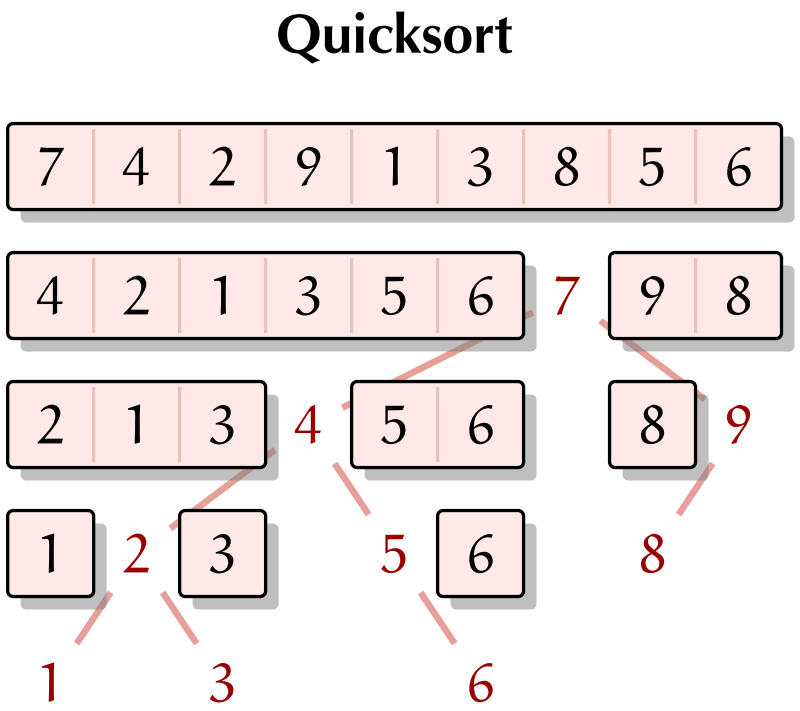


Quicksort & Search Trees (Distinct Keys)

Quicksort

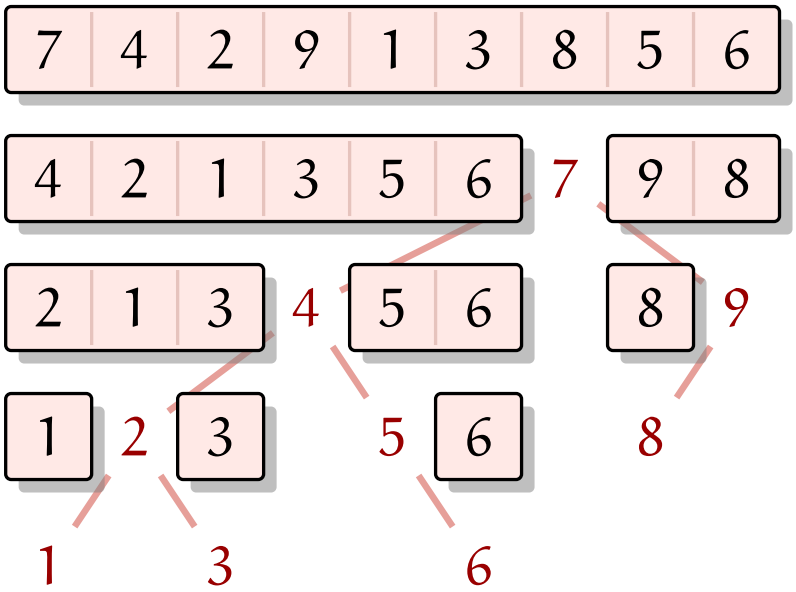


Quicksort & Search Trees (Distinct Keys)



Quicksort & Search Trees (Distinct Keys)

Quicksort

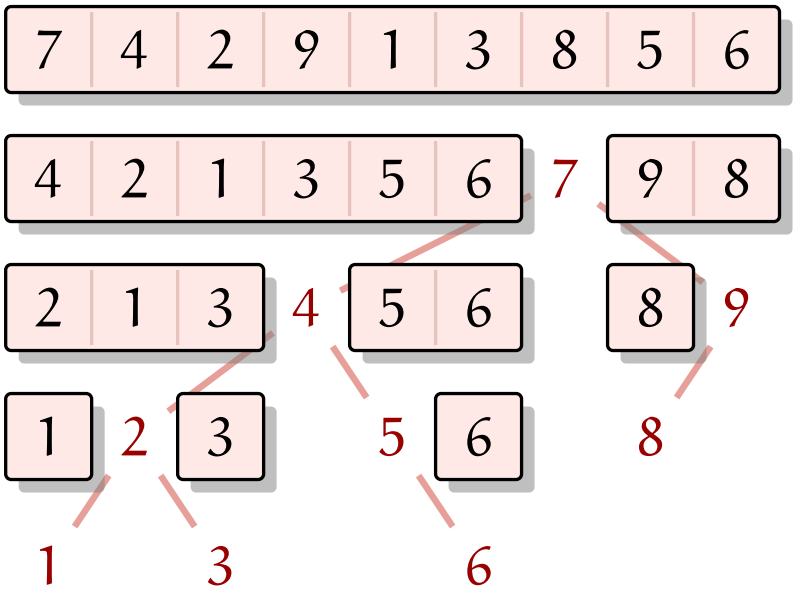


Binary Search Tree

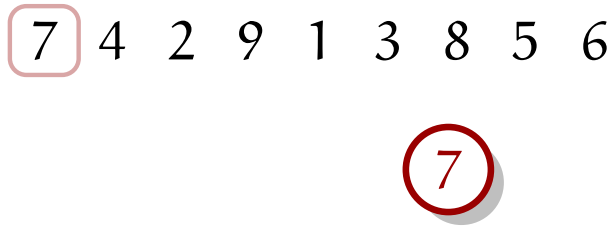
7 4 2 9 1 3 8 5 6

Quicksort & Search Trees (Distinct Keys)

Quicksort

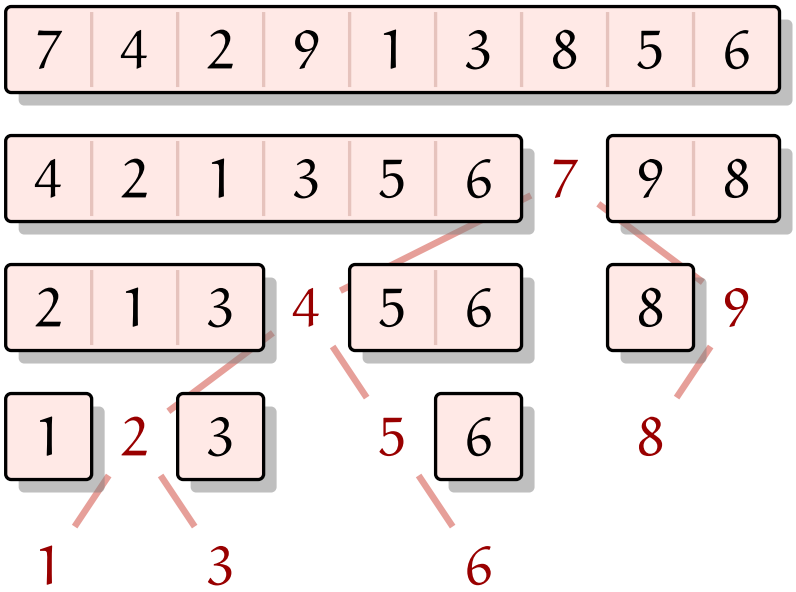


Binary Search Tree

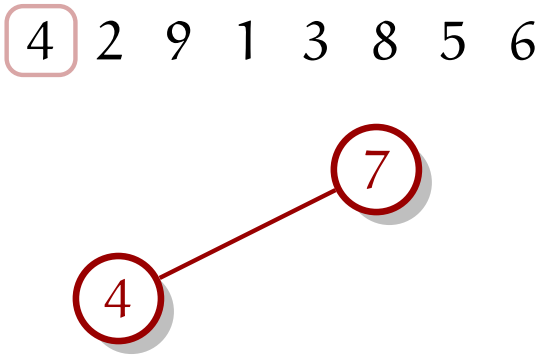


Quicksort & Search Trees (Distinct Keys)

Quicksort

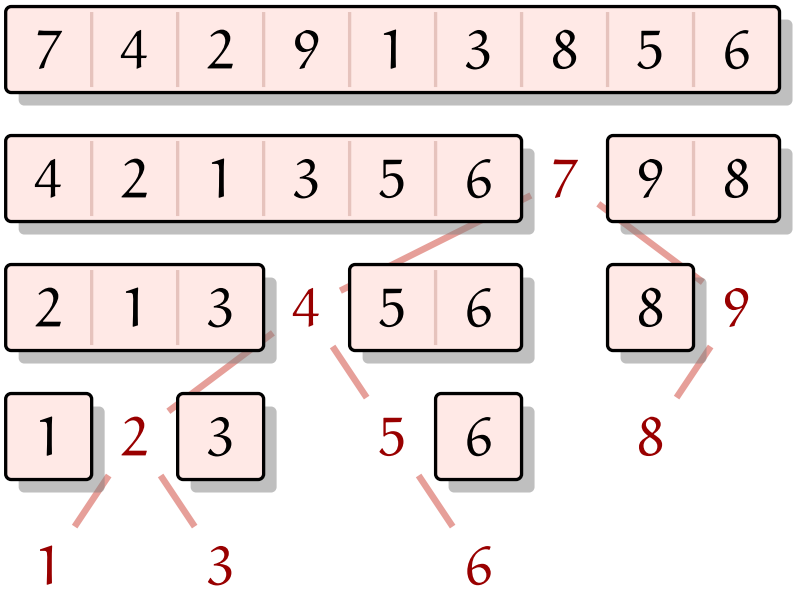


Binary Search Tree

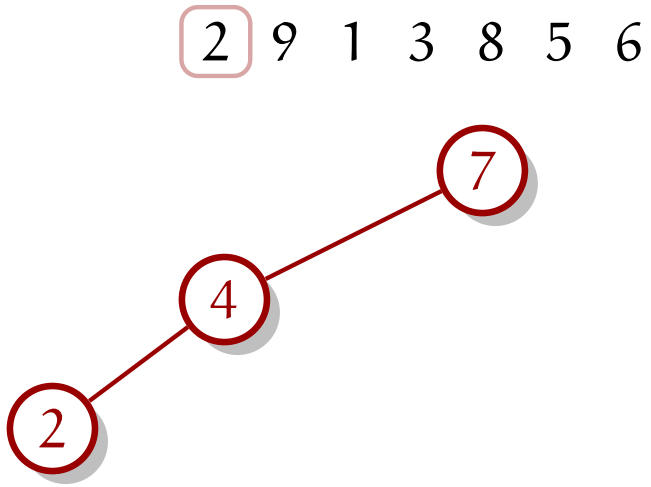


Quicksort & Search Trees (Distinct Keys)

Quicksort

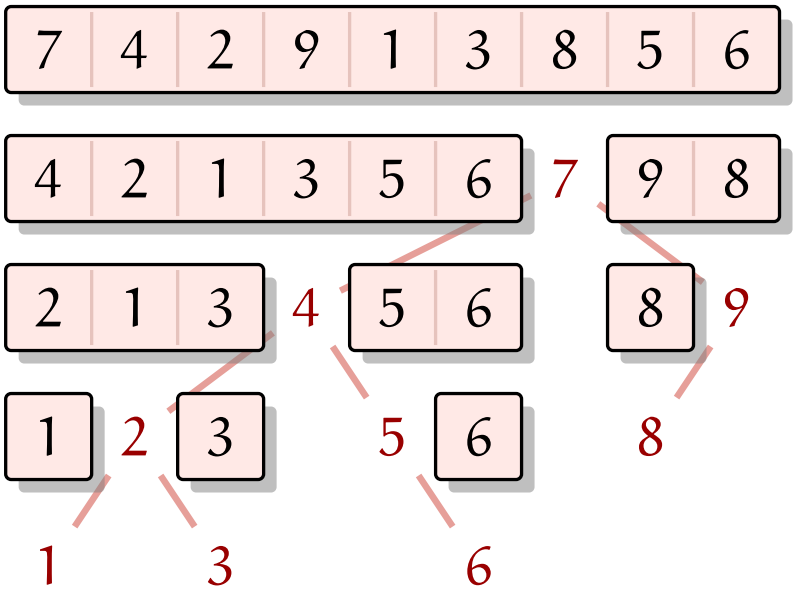


Binary Search Tree

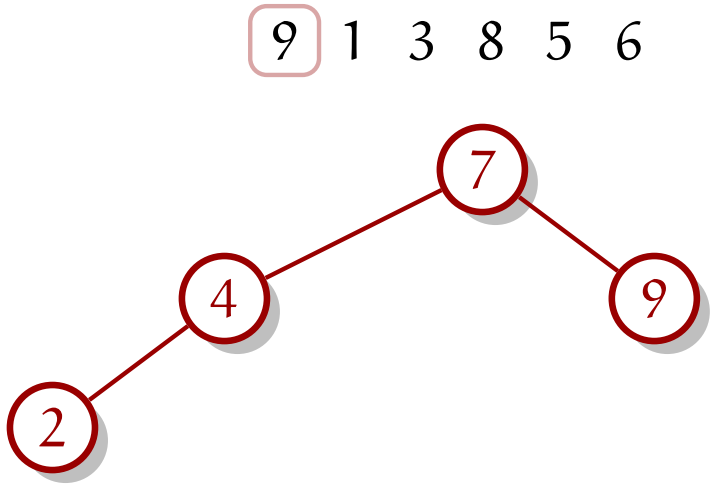


Quicksort & Search Trees (Distinct Keys)

Quicksort

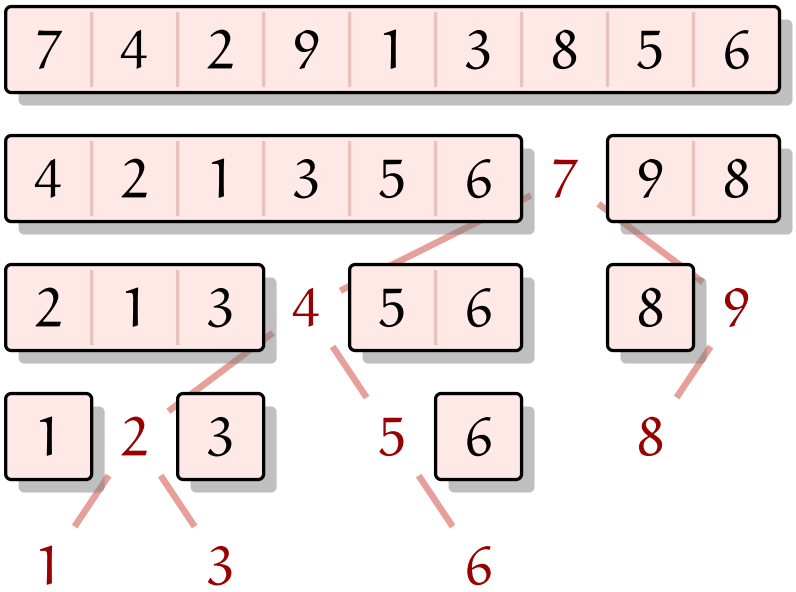


Binary Search Tree

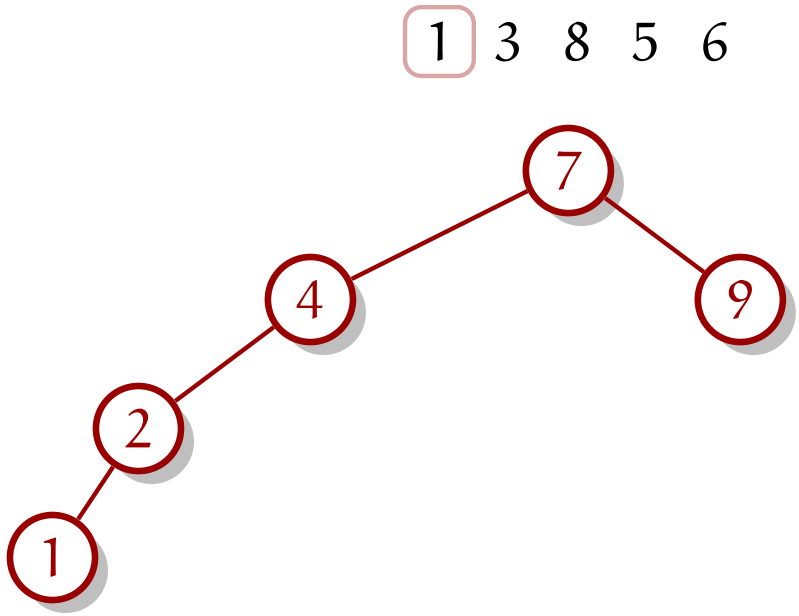


Quicksort & Search Trees (Distinct Keys)

Quicksort

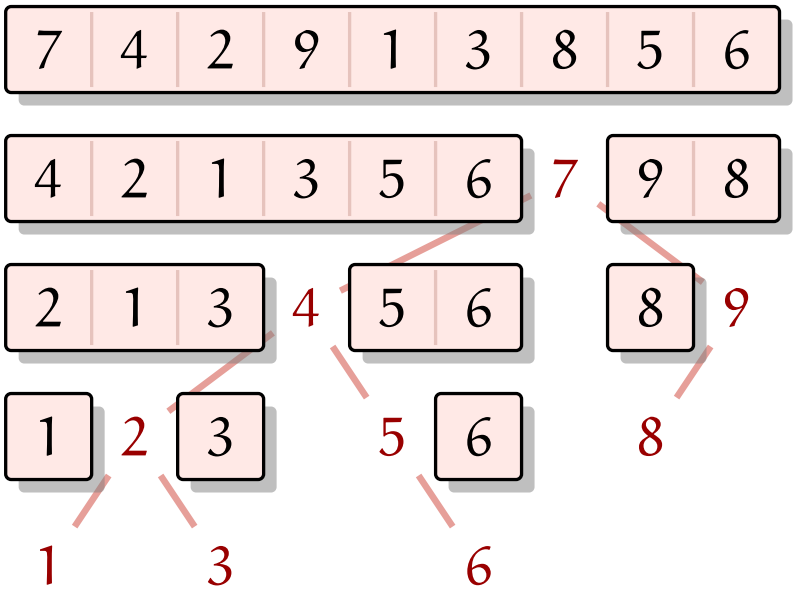


Binary Search Tree

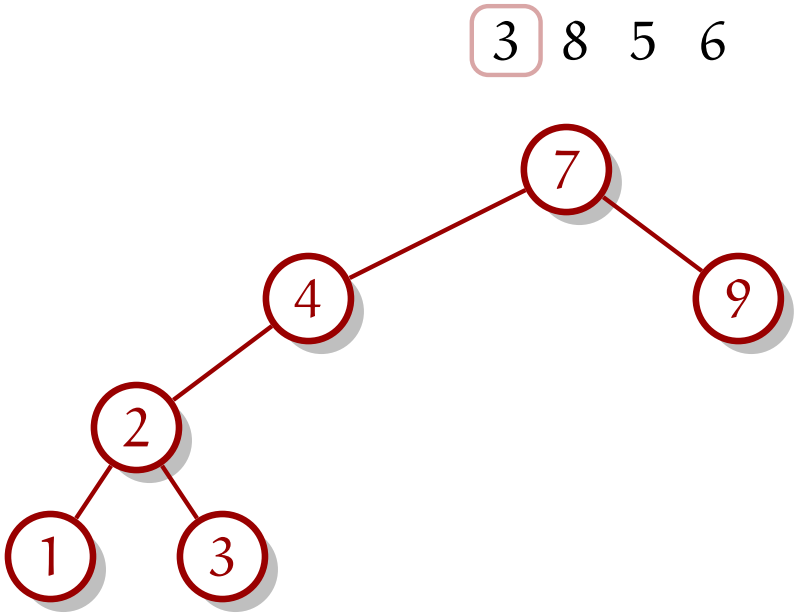


Quicksort & Search Trees (Distinct Keys)

Quicksort

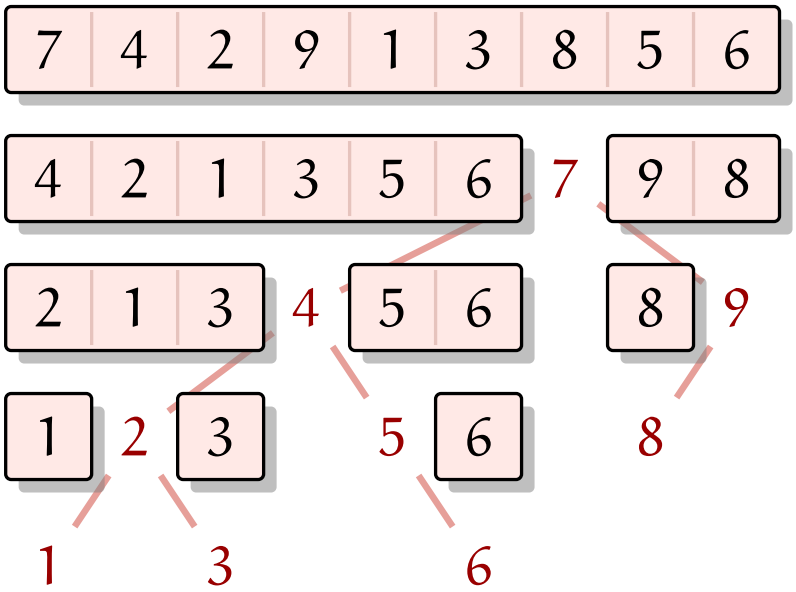


Binary Search Tree

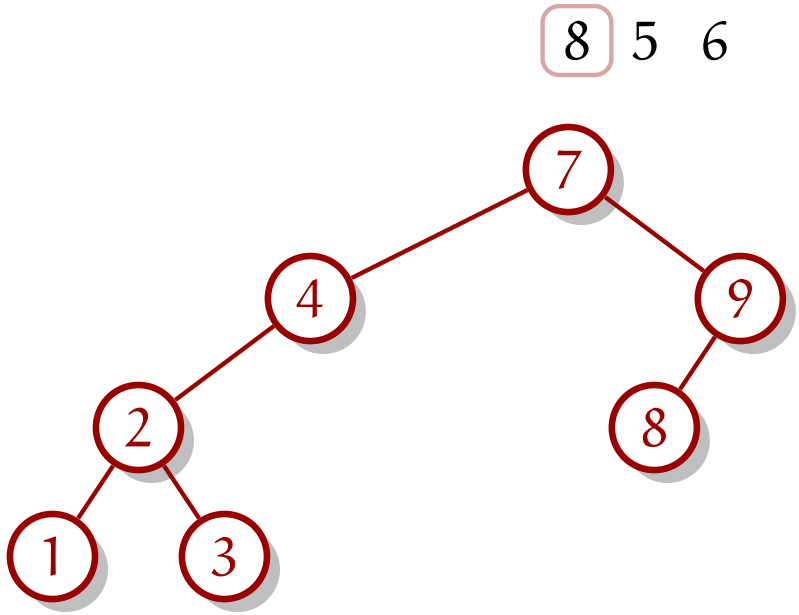


Quicksort & Search Trees (Distinct Keys)

Quicksort

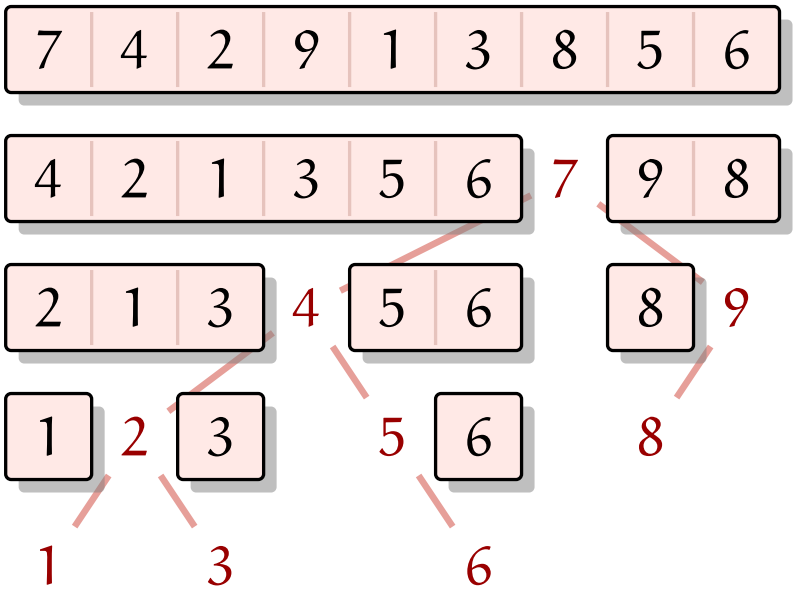


Binary Search Tree

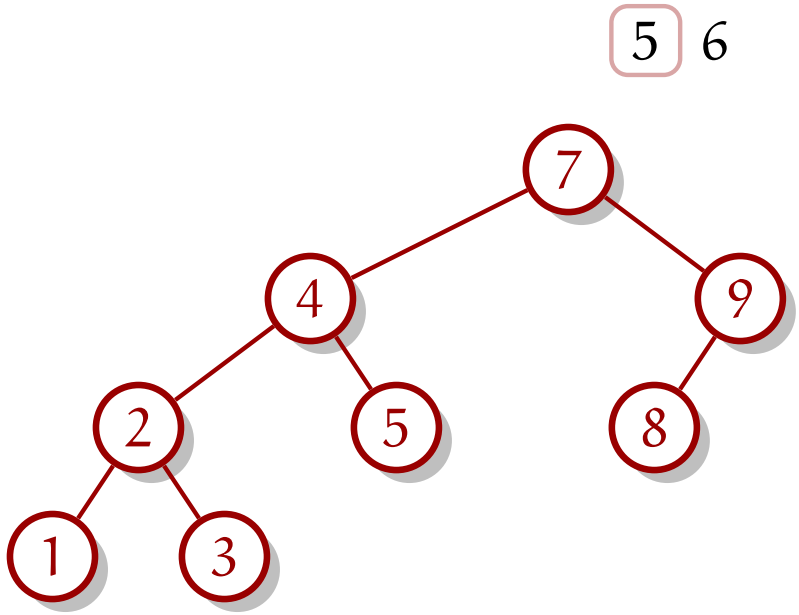


Quicksort & Search Trees (Distinct Keys)

Quicksort

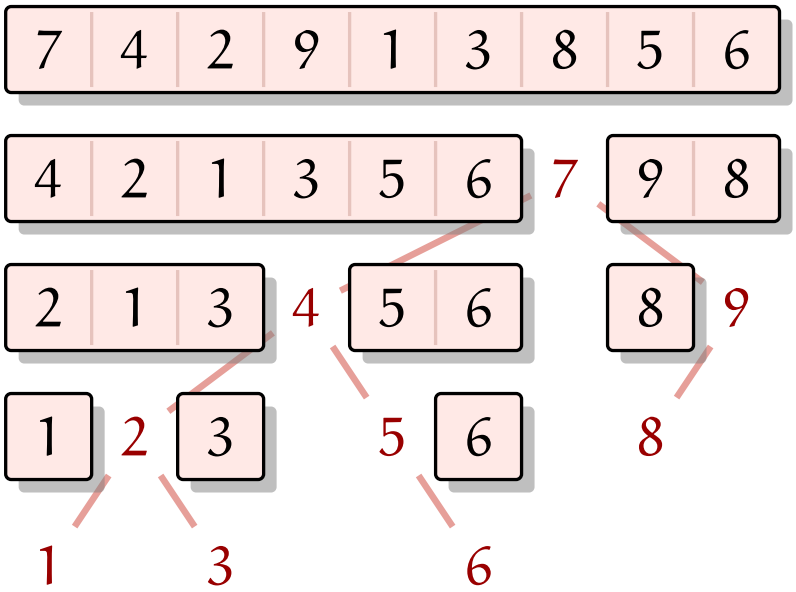


Binary Search Tree

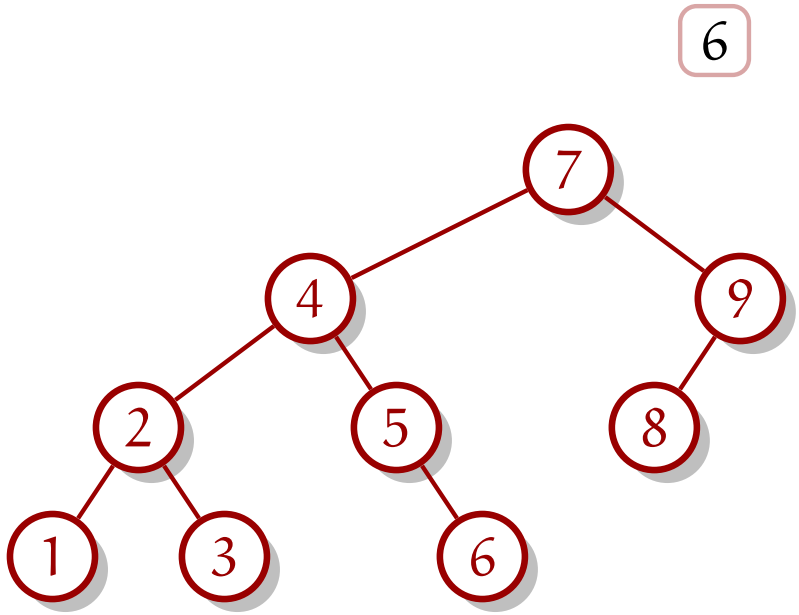


Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree

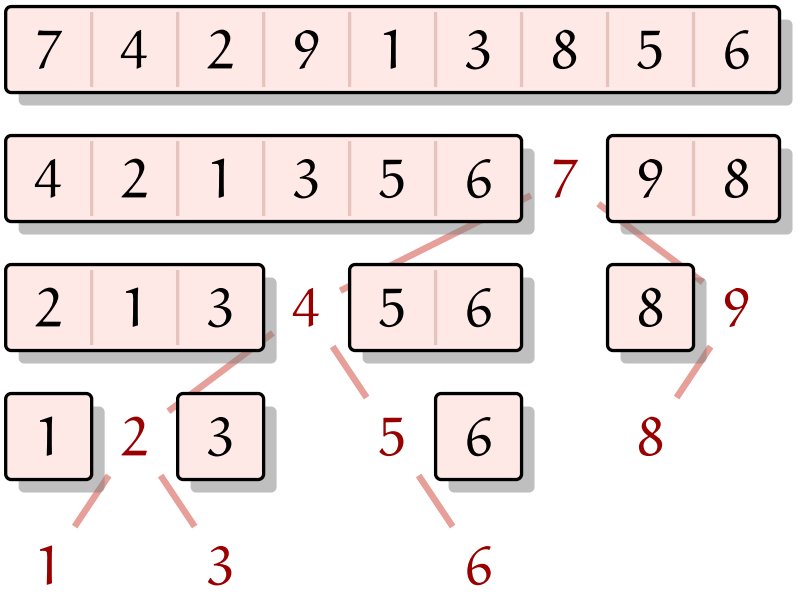


⇝ **Well-Known Facts:**

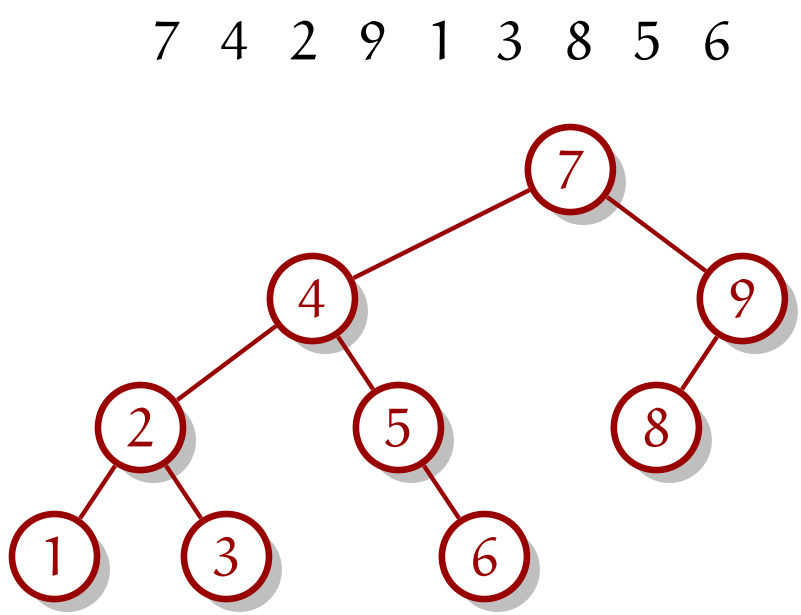
- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree

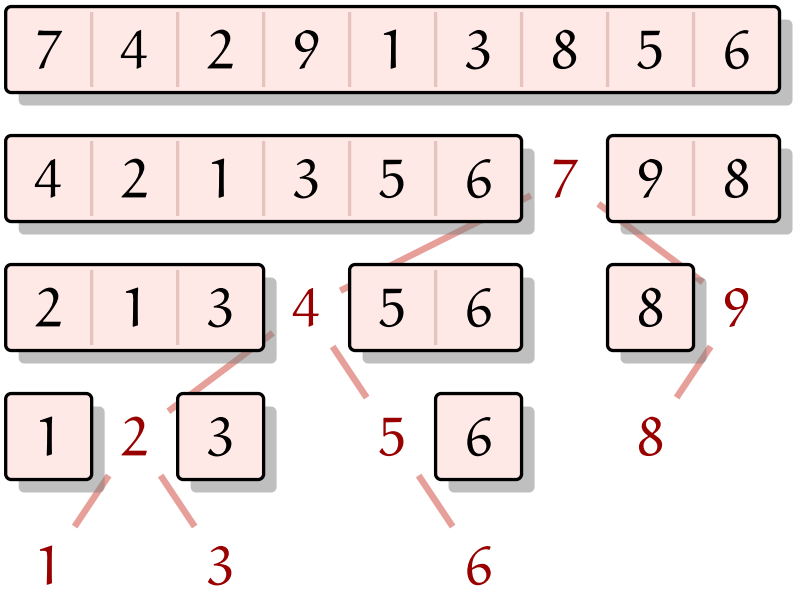


⇝ Well-Known Facts:

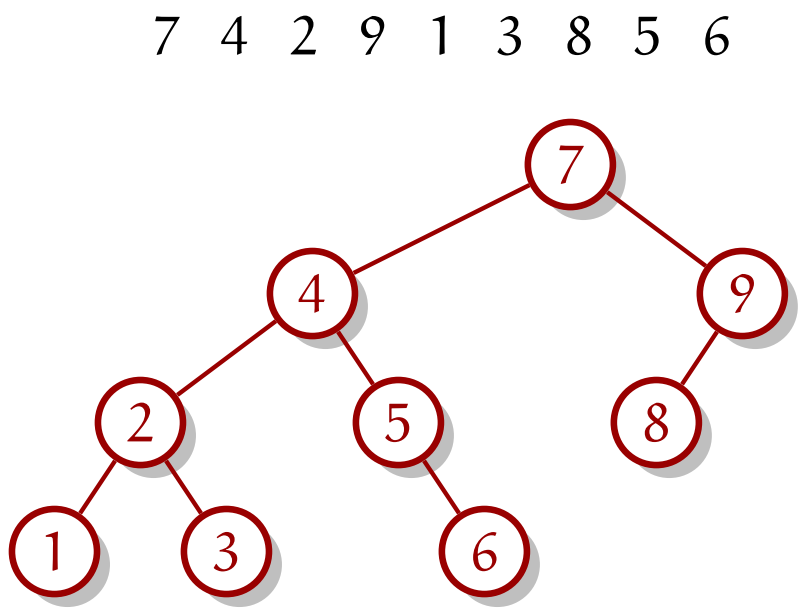
- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree

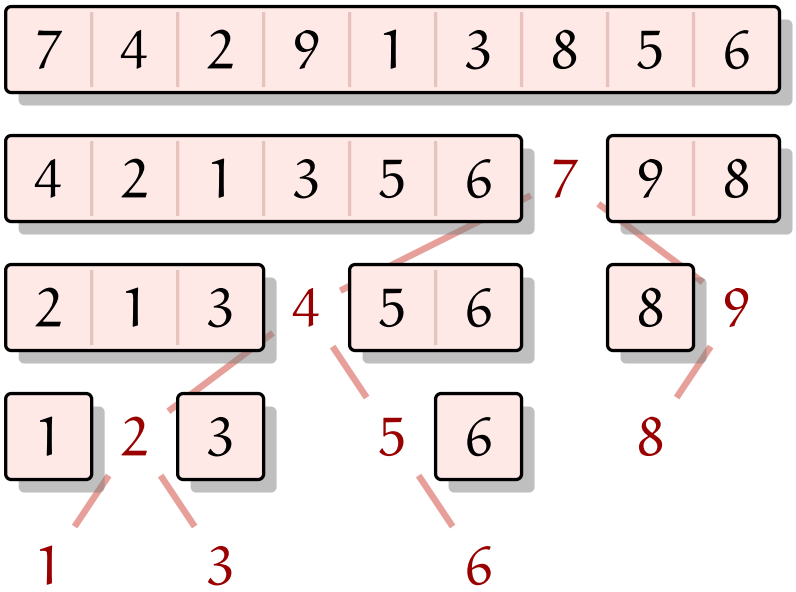


↝ **Well-Known Facts:**

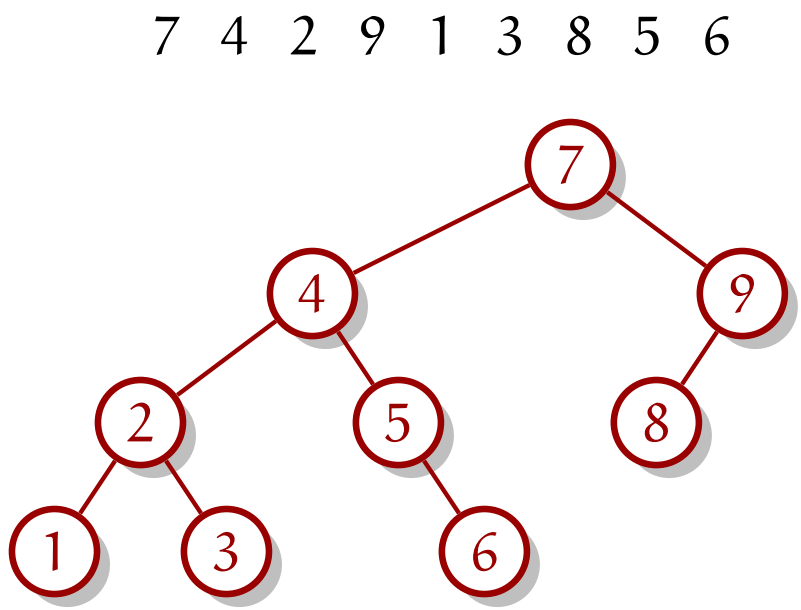
- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree

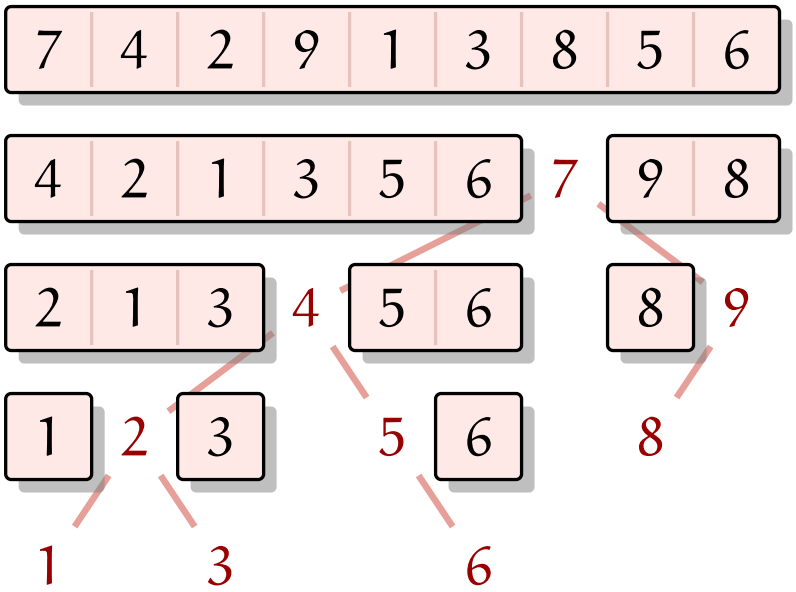


↝ **Well-Known Facts:**

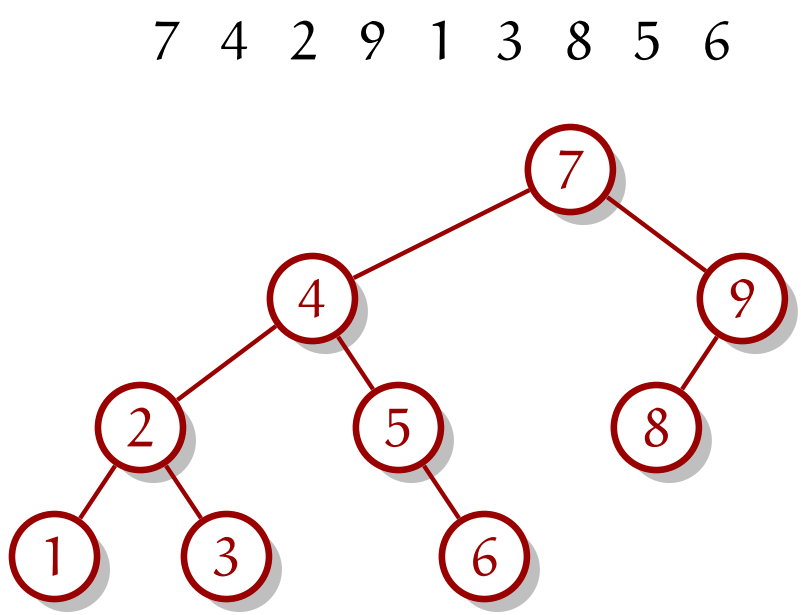
- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree

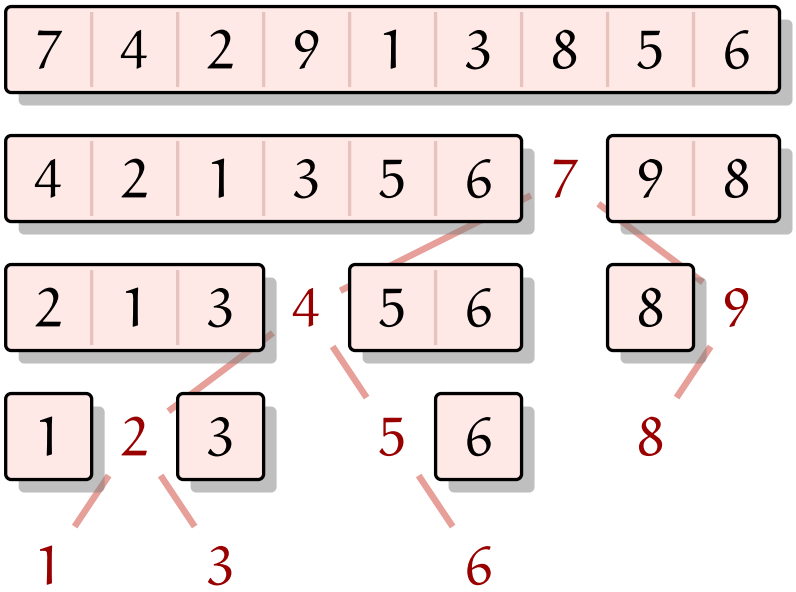


↝ **Well-Known Facts:**

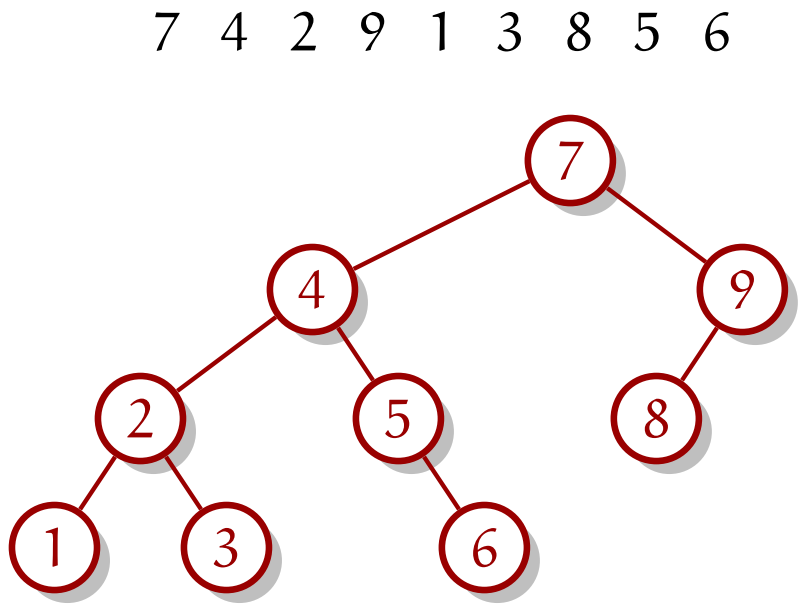
- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

Quicksort & Search Trees (Distinct Keys)

Quicksort



Binary Search Tree



↝ **Well-Known Facts:**

- Recursion Tree of Quicksort = Naturally Grown BST
- Comparisons in Quicksort = Comparison to built BST
- Comparisons in Quicksort = Comparisons to search each element in BST

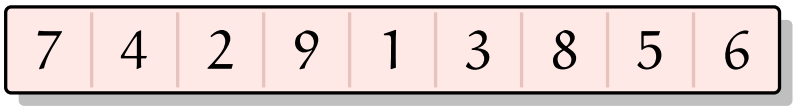
How about median-of-three Quicksort etc.?

Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

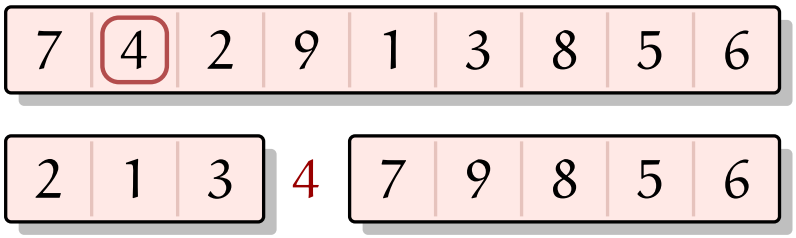


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

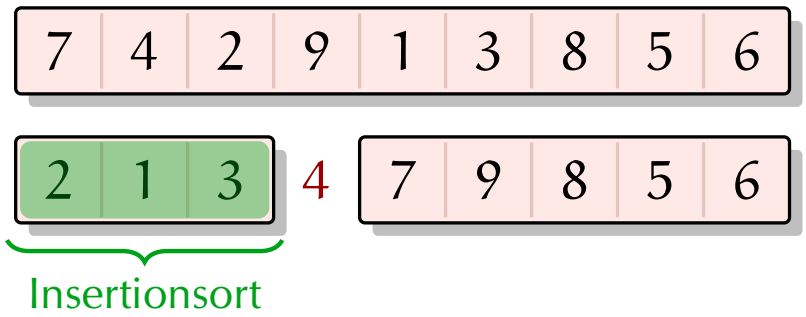


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

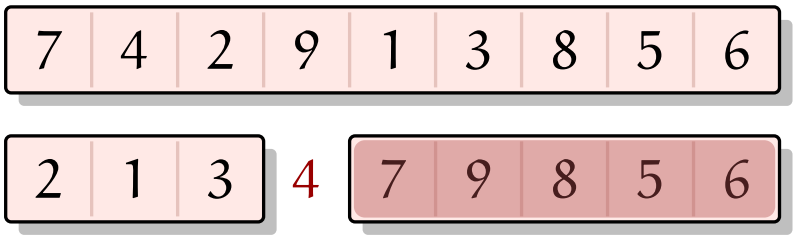


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

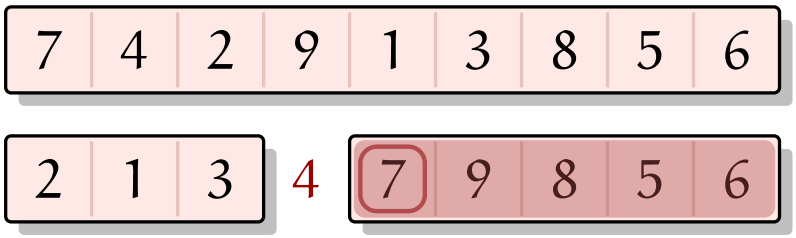


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

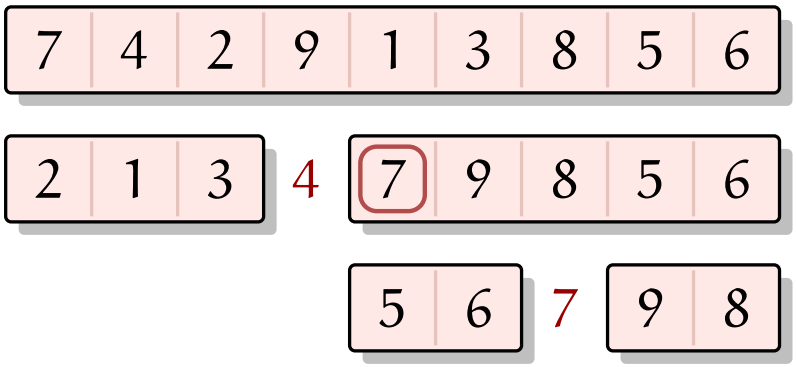


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

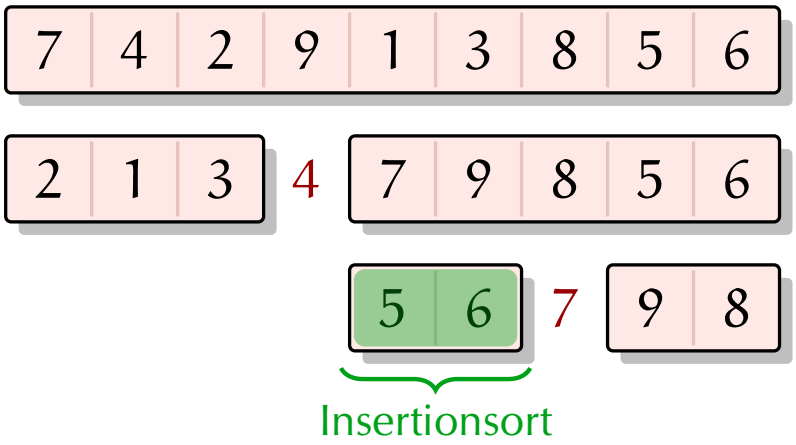


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

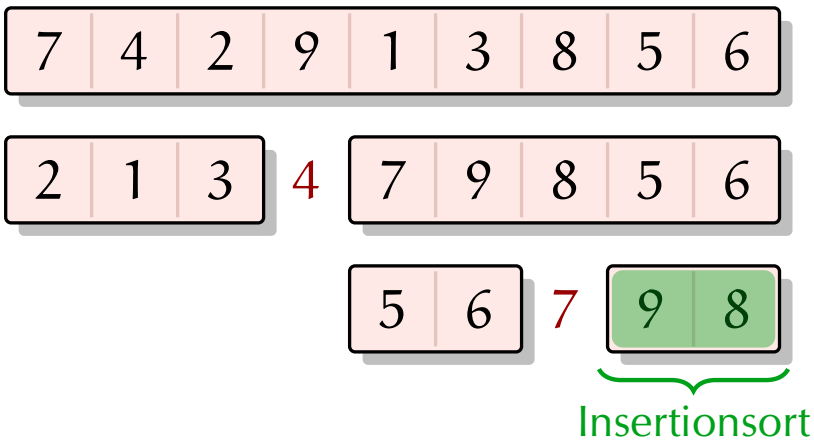


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

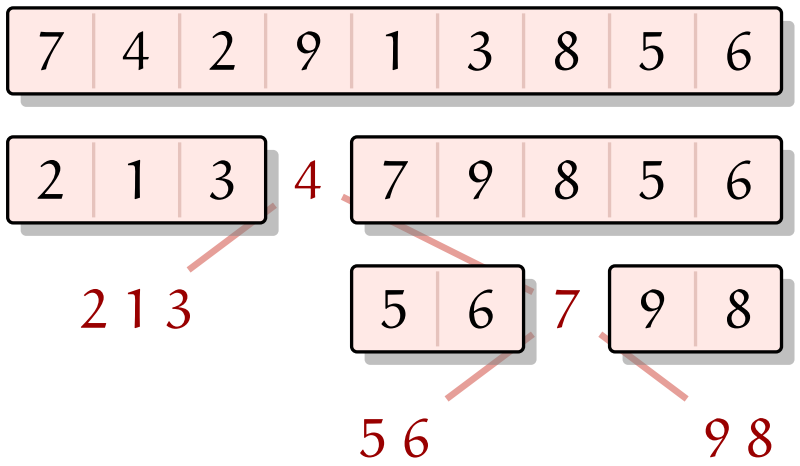


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort

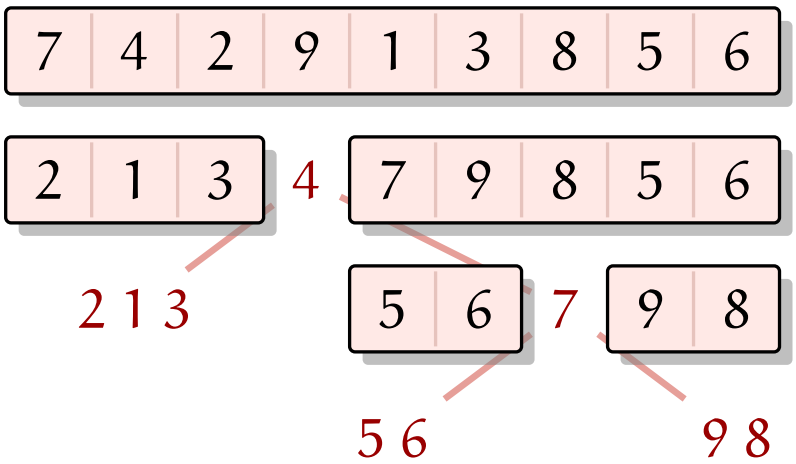


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

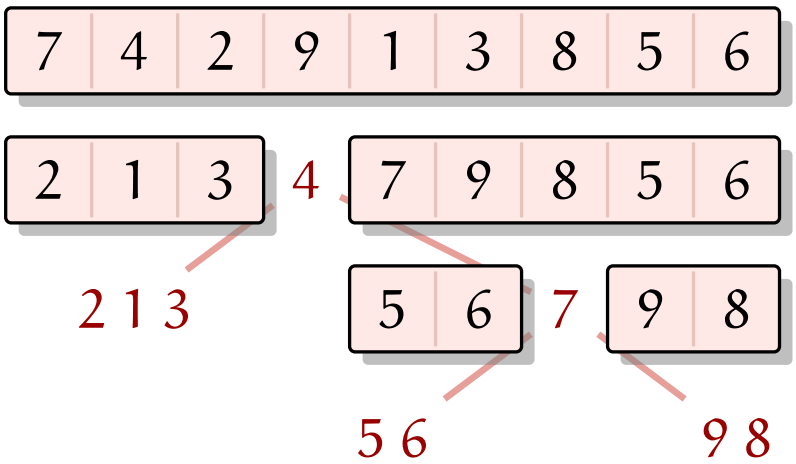
7 4 2 9 1 3 8 5 6

Fringe-Balanced Trees (Distinct Keys)

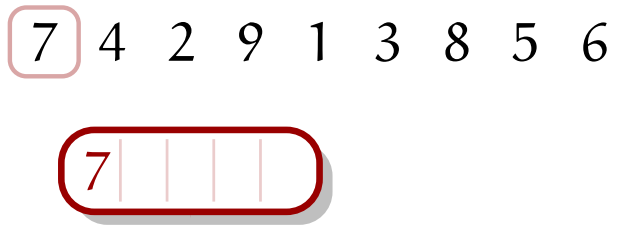
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

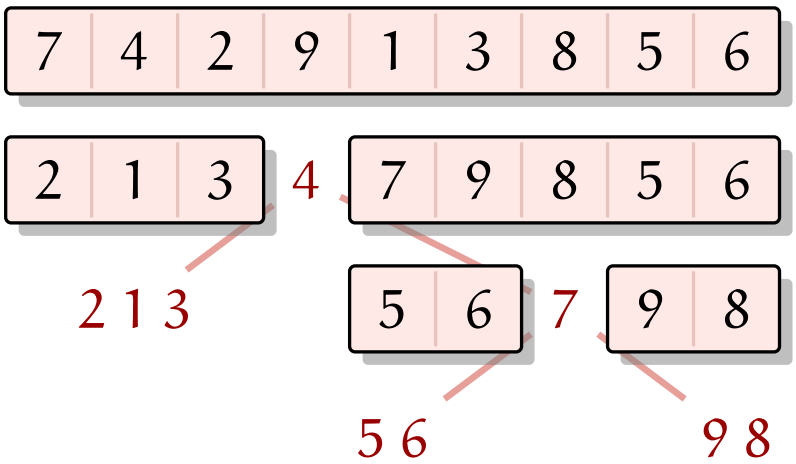


Fringe-Balanced Trees (Distinct Keys)

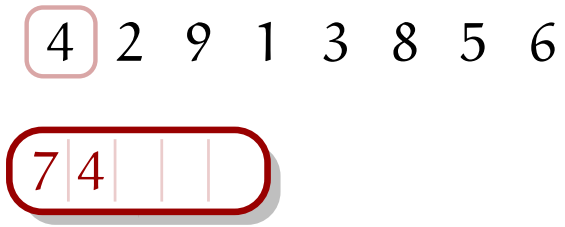
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

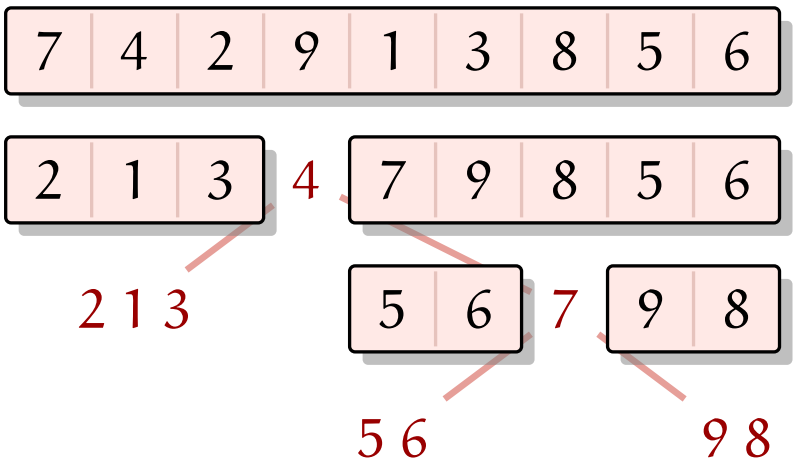


Fringe-Balanced Trees (Distinct Keys)

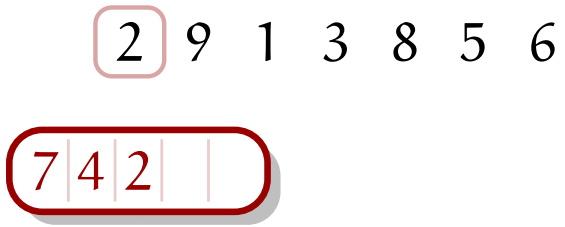
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

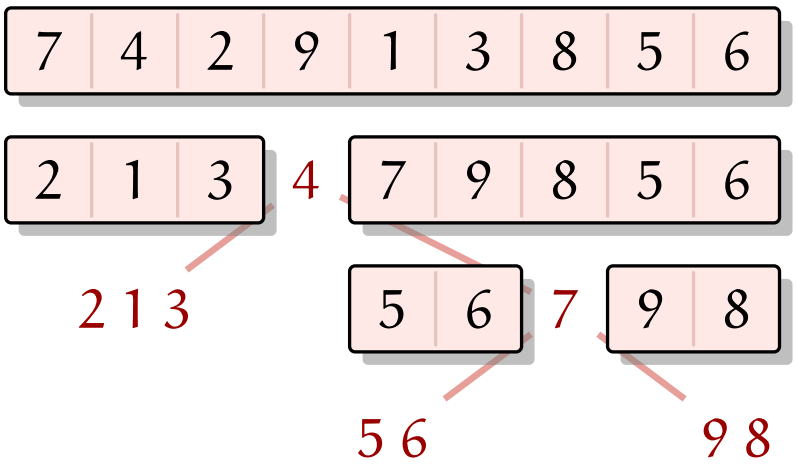


Fringe-Balanced Trees (Distinct Keys)

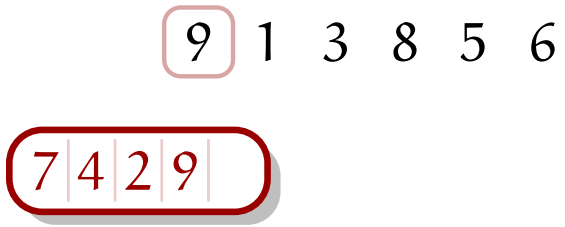
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

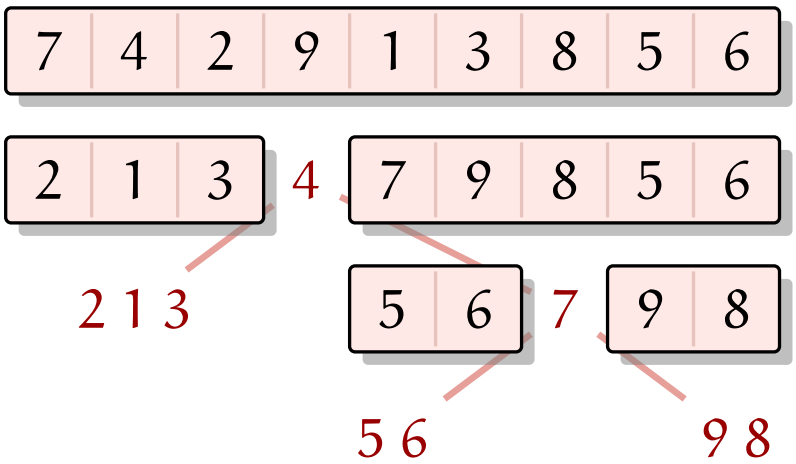


Fringe-Balanced Trees (Distinct Keys)

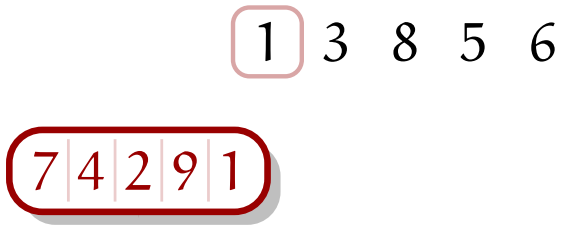
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

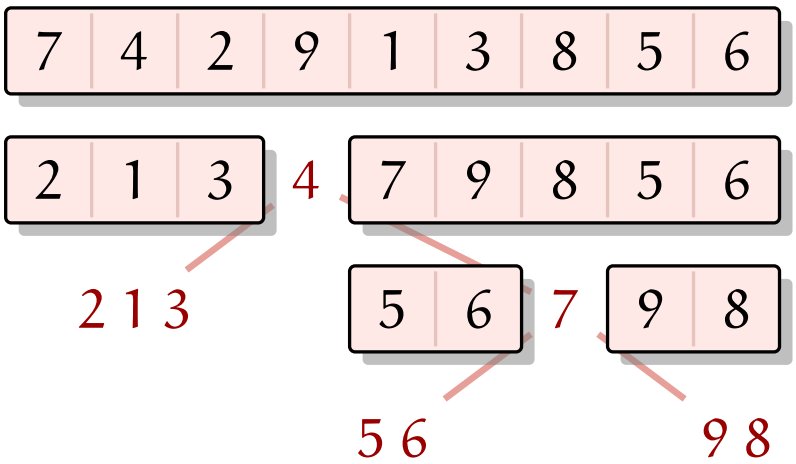


Fringe-Balanced Trees (Distinct Keys)

Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

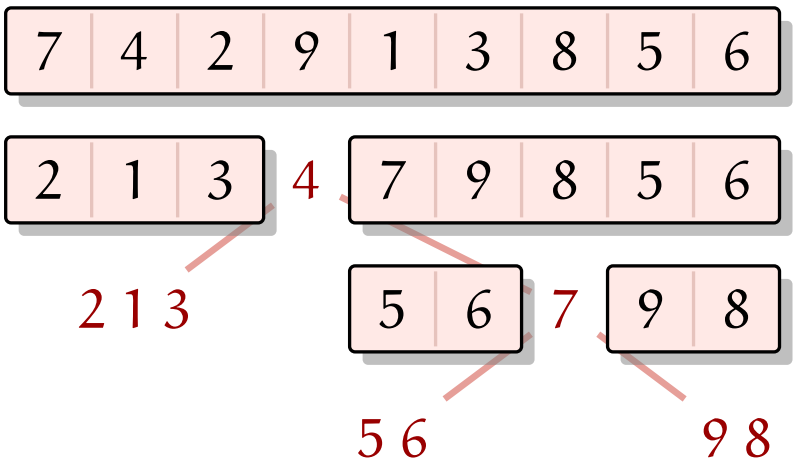


Fringe-Balanced Trees (Distinct Keys)

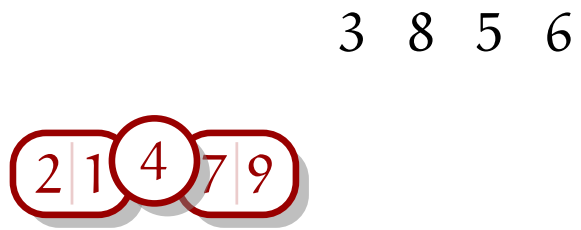
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

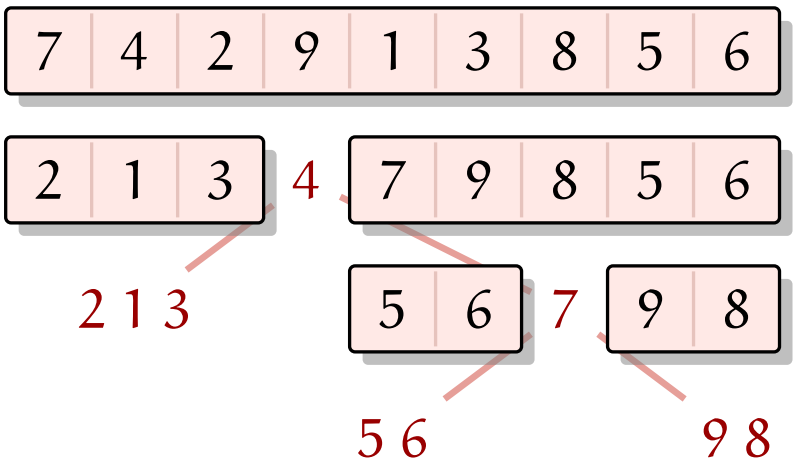


Fringe-Balanced Trees (Distinct Keys)

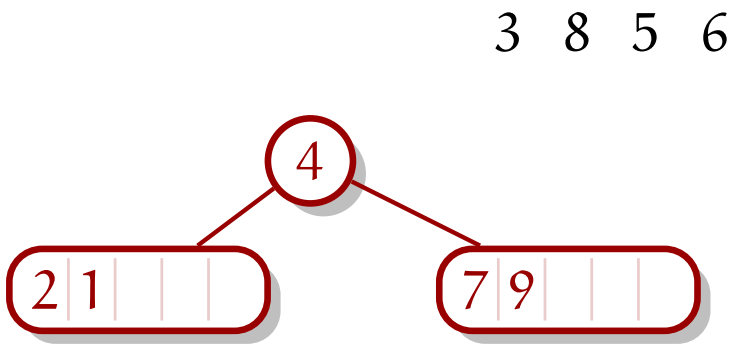
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

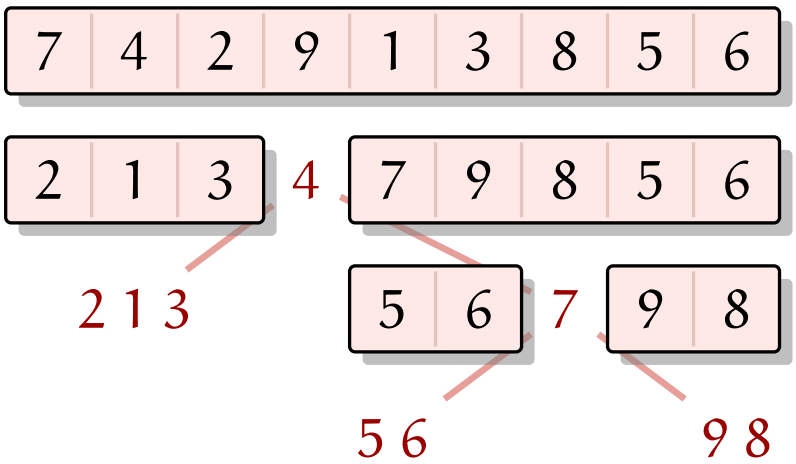


Fringe-Balanced Trees (Distinct Keys)

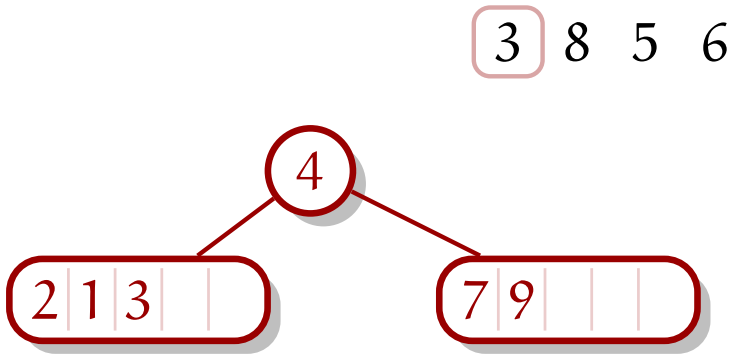
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

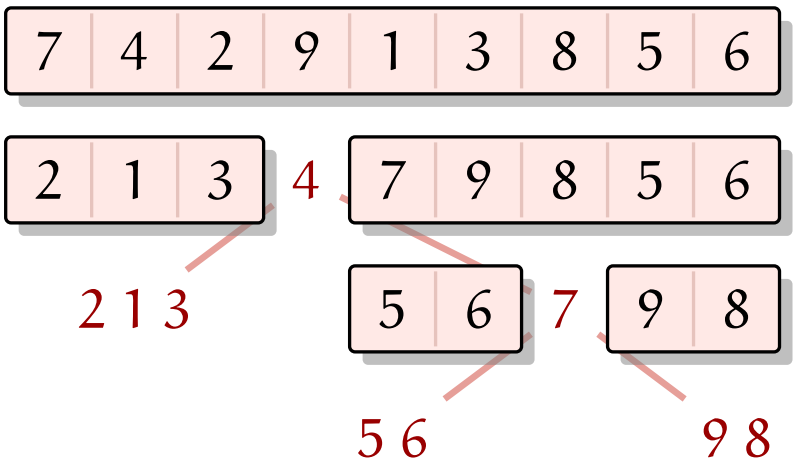


Fringe-Balanced Trees (Distinct Keys)

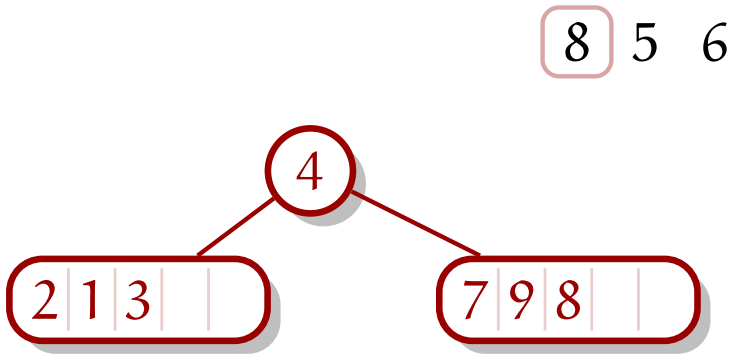
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

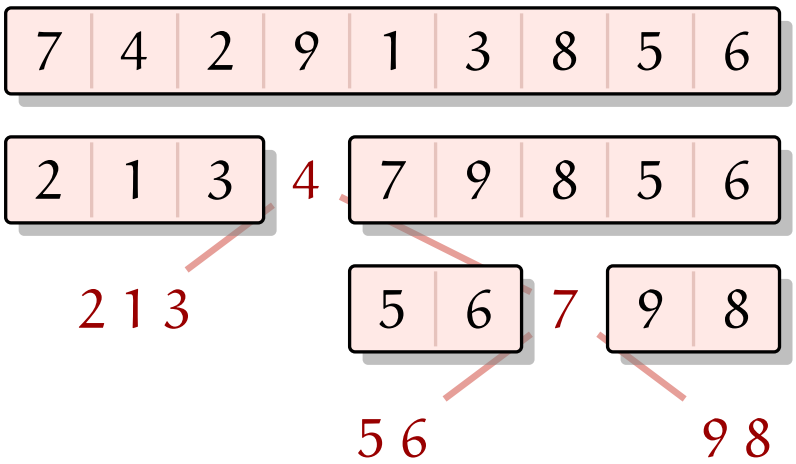


Fringe-Balanced Trees (Distinct Keys)

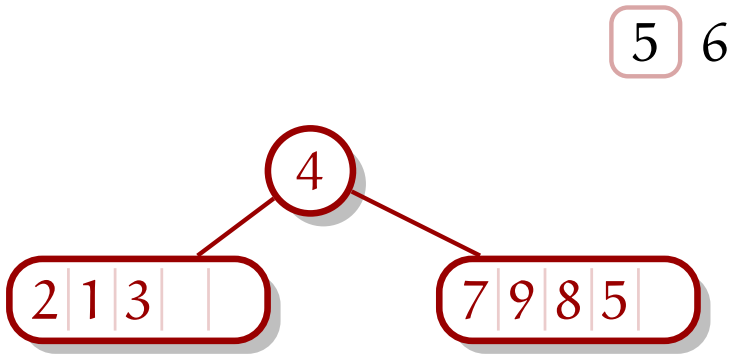
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

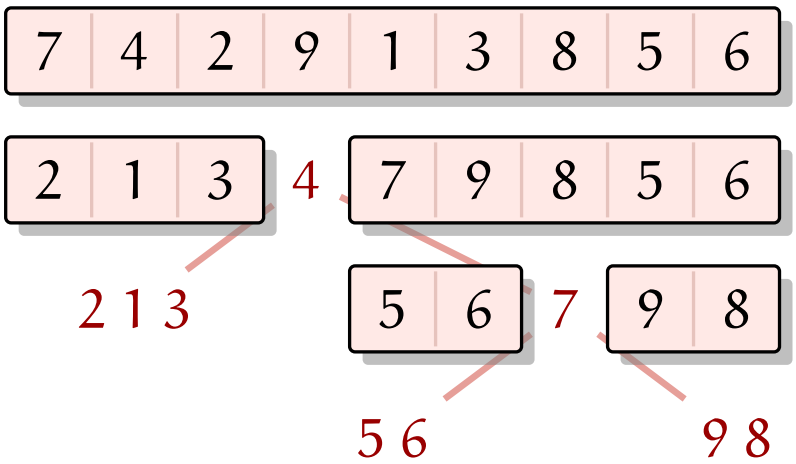


Fringe-Balanced Trees (Distinct Keys)

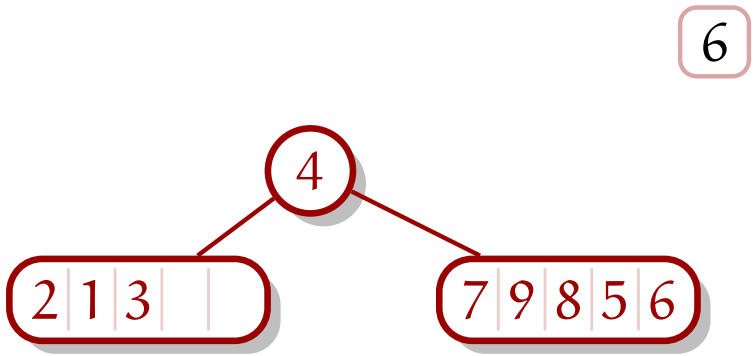
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

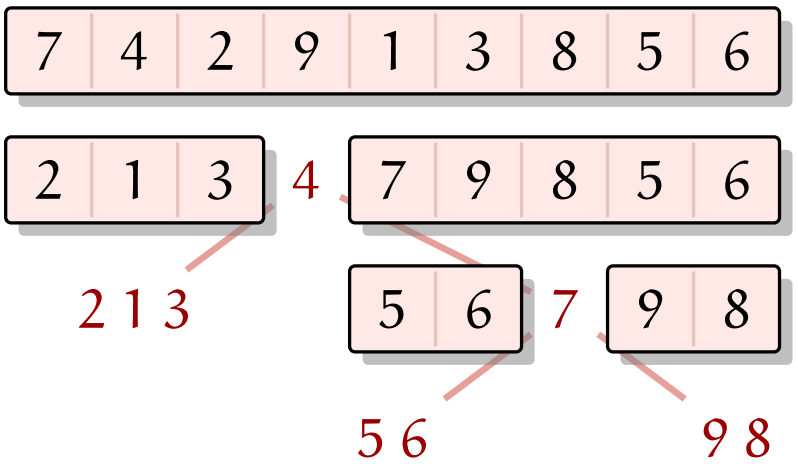


Fringe-Balanced Trees (Distinct Keys)

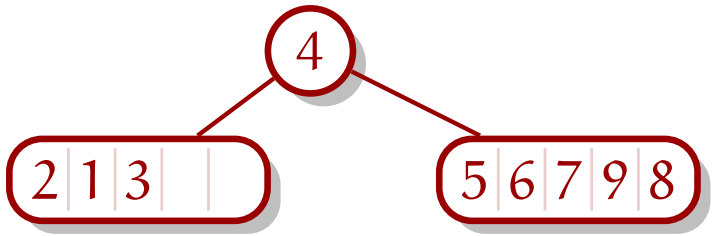
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

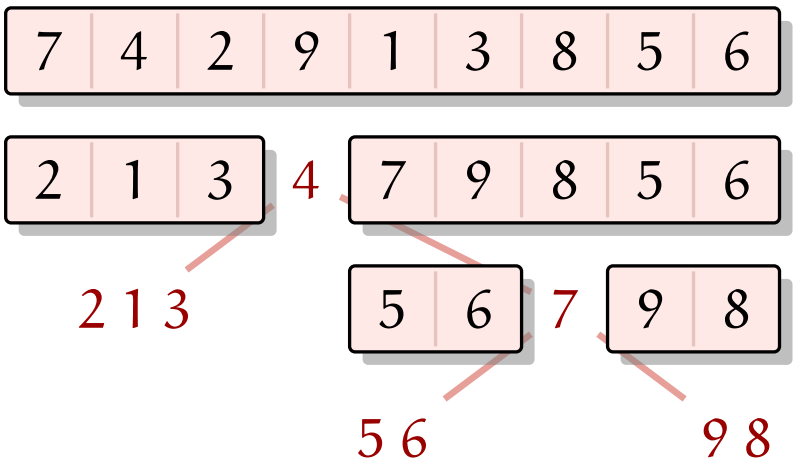


Fringe-Balanced Trees (Distinct Keys)

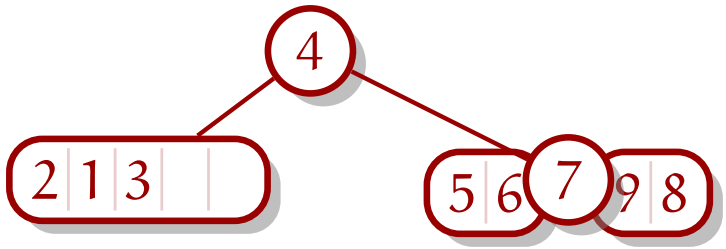
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree

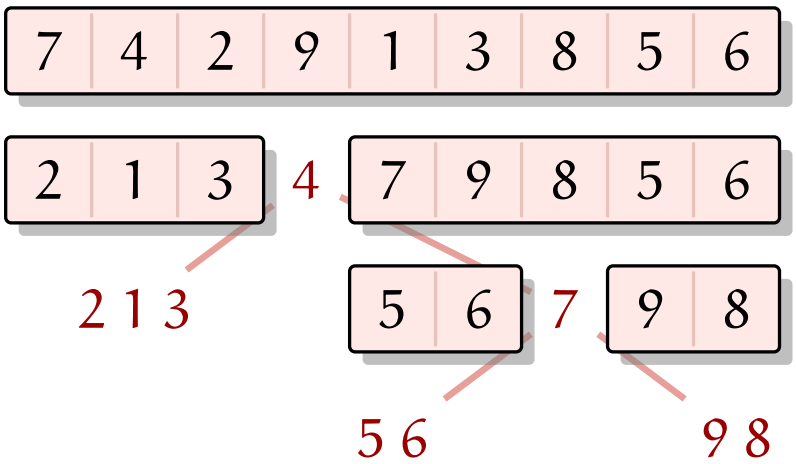


Fringe-Balanced Trees (Distinct Keys)

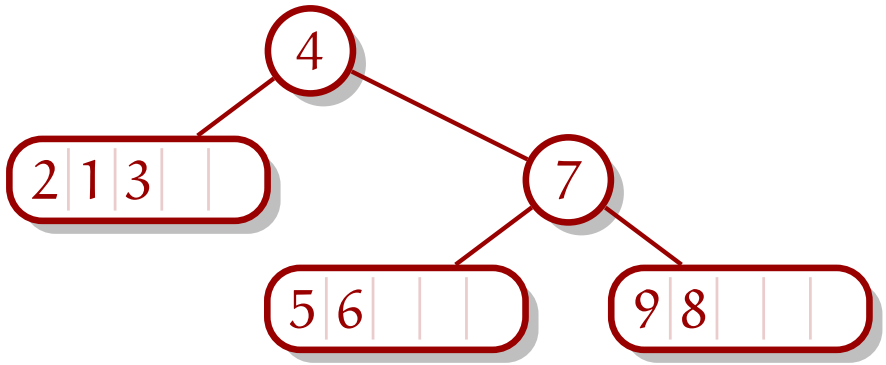
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree



⇝ Correspondence extends to

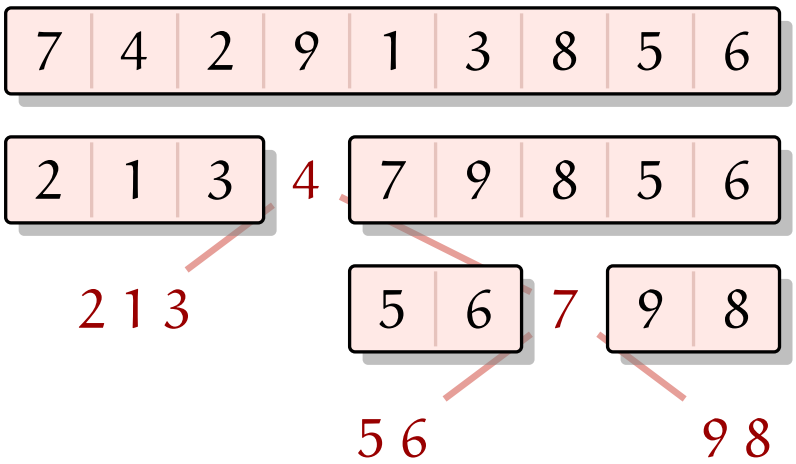
- Pivot Sampling (any scheme, not only median)
- Larger Insertion Sort Cutoff
- Linear Insertion Sort Cutoff

Fringe-Balanced Trees (Distinct Keys)

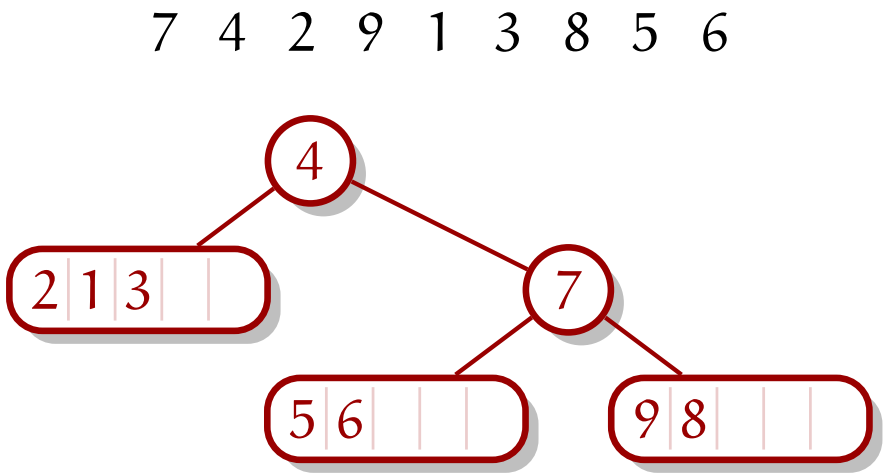
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree



⇝ Correspondence extends to

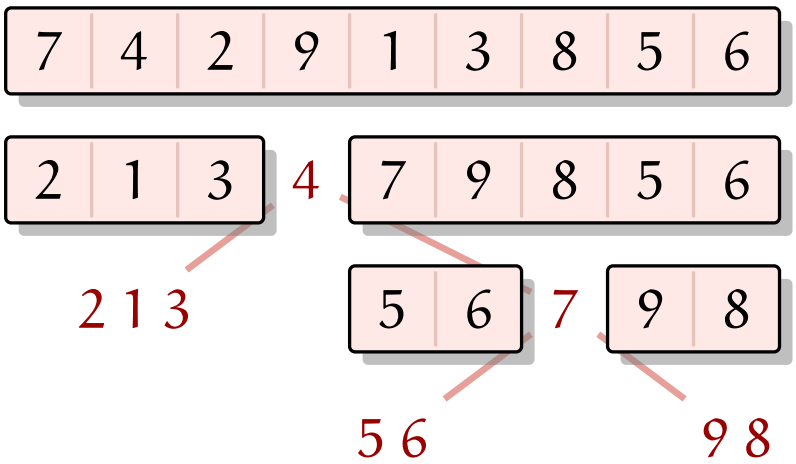
- Pivot Sampling (any scheme, not only median)
- Larger Insertionsort Cutoff
- **s-way Partitioning** ⇝ s-ary search trees

Fringe-Balanced Trees (Distinct Keys)

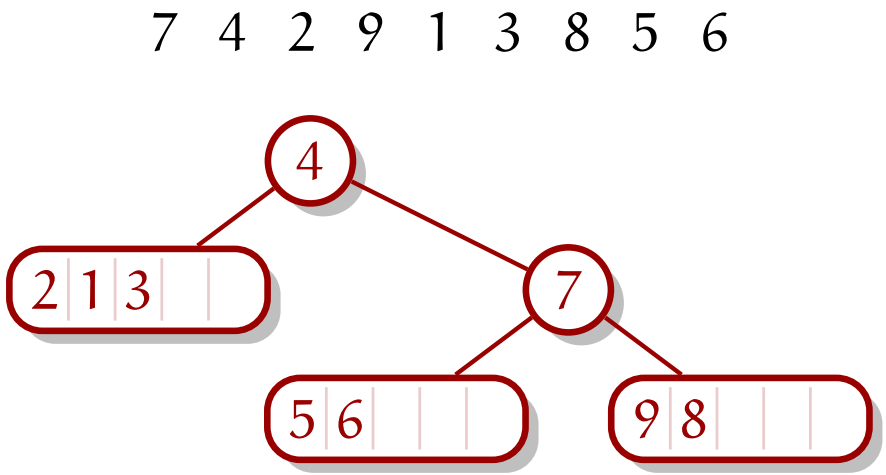
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree



⇝ Correspondence extends to

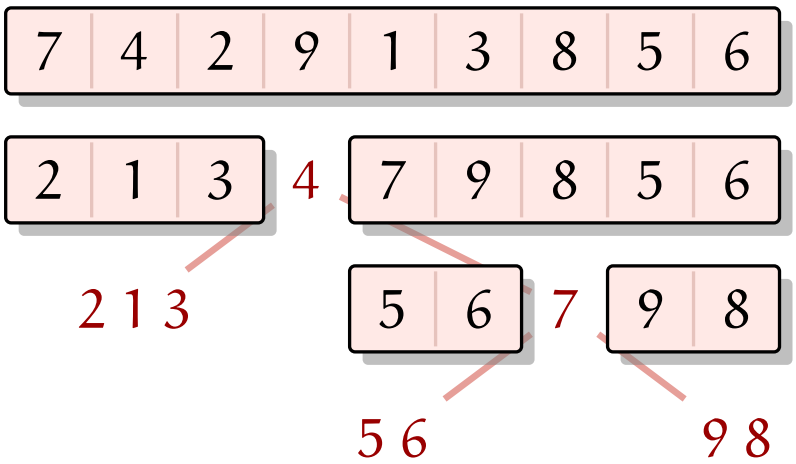
- Pivot Sampling (any scheme, not only median)
- Larger Insertionsort Cutoff
- **s-way Partitioning** ⇝ s-ary search trees

Fringe-Balanced Trees (Distinct Keys)

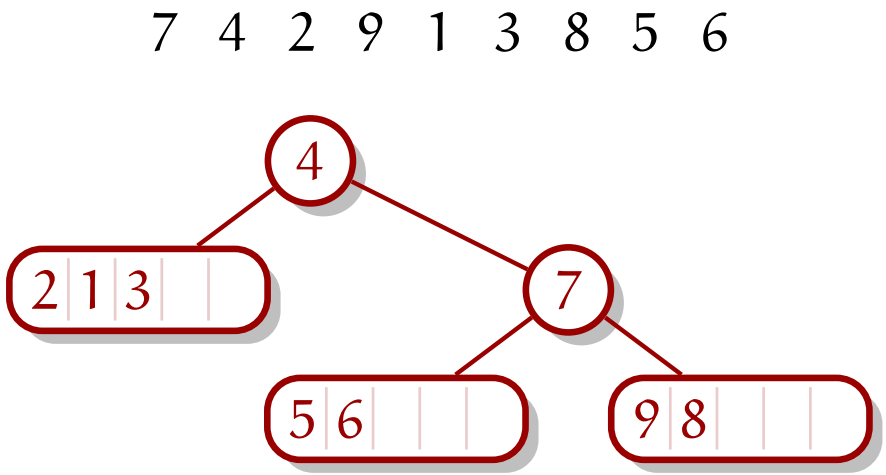
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree



⇝ Correspondence extends to

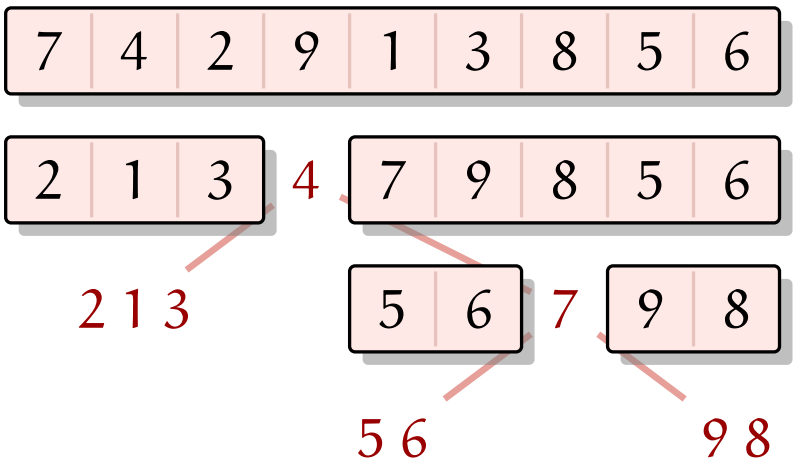
- Pivot Sampling (any scheme, not only median)
- Larger Insertionsort Cutoff
- **s-way Partitioning** ⇝ s-ary search trees

Fringe-Balanced Trees (Distinct Keys)

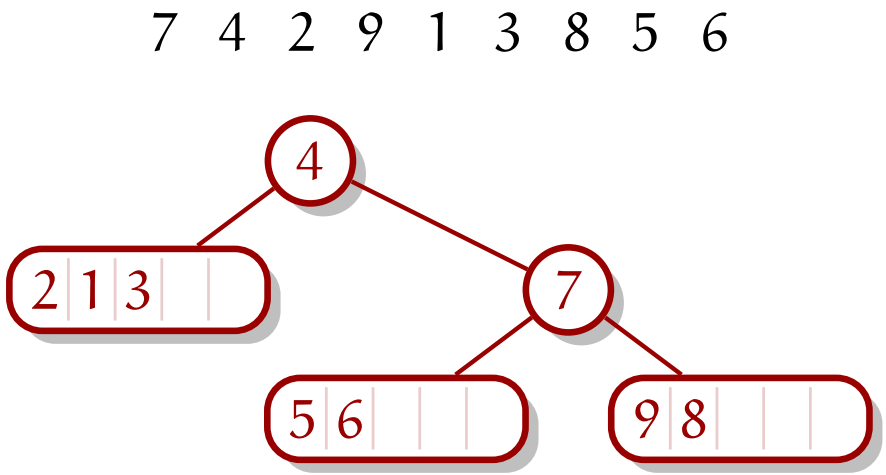
Rules for Fringe-Balanced Search Trees

- Leaves have **buffer** for elements
- If buffer is full, leaf is split
- A new internal node with the median is created.

Median-of-5 Quicksort



Fringe-Balanced Tree



⇝ Correspondence extends to

- Pivot Sampling (any scheme, not only median)
- Larger Insertionsort Cutoff
- **s-way Partitioning** ⇝ s-ary search trees