Median-of-k Quicksort Is Optimal For Many Equal Keys

Sebastian Wild

wild@cs.uni-kl.de



originates from joint work with

Martin Aumüller, Martin Dietzfelbinger,
Conrado Martínez, and Markus Nebel

AofA 2017

28th International Meeting on Probabilistic,
Combinatorial and Asymptotic Methods
for the Analysis of Algorithms

Outline

0 Intro

1 Quicksort and Search Trees

2 Saturated Fringe-Balanced Trees

3 Back to Multiset Permutations

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries...)
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don’t we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs			
variance			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won’t go away!

Don’t we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs			
variance			
tail inequalities			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won’t go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs			
variance			
tail inequalities			
limit distributions			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs			
variance			
tail inequalities			
limit distributions			
(semi-)local limit laws			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries...
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences		
variance			
tail inequalities			
limit distributions			
(semi-)local limit laws			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries...
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs variance tail inequalities limit distributions (semi-)local limit laws	telescoping recurrences singularity analysis		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries . . .)
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs variance tail inequalities limit distributions (semi-)local limit laws	telescoping recurrences singularity analysis Euler differential eq.		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences		
variance	singularity analysis		
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws			

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences		
variance	singularity analysis		
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws	contraction method		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences		
variance	singularity analysis		
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws	contraction method		
	branching processes		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences		
variance	singularity analysis		
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws	contraction method		
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	
variance	singularity analysis		
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws	contraction method		
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries . . .)
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	
variance	singularity analysis	Insertionsort cutoff	
tail inequalities	Euler differential eq.		
limit distributions	Martingales		
(semi-)local limit laws	contraction method		
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
  not done in libraries . . .
- partly justified: we can (should!) randomize Quicksort,
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	
variance	singularity analysis	Insertionsort cutoff	
tail inequalities	Euler differential eq.	multiway partitioning	
limit distributions	Martingales		
(semi-)local limit laws	contraction method		
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries...)
  $\rightsquigarrow$ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	
variance	singularity analysis	Insertionsort cutoff	
tail inequalities	Euler differential eq.	multiway partitioning	
limit distributions	Martingales	Quickselect	
(semi-)local limit laws	contraction method		
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	
variance	singularity analysis	Insertionsort cutoff	
tail inequalities	Euler differential eq.	multiway partitioning	
limit distributions	Martingales	Quickselect	
(semi-)local limit laws	contraction method	constant space	
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries...
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	
tail inequalities	Euler differential eq.	multiway partitioning	
limit distributions	Martingales	Quickselect	
(semi-)local limit laws	contraction method	constant space	
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	
limit distributions	Martingales	Quickselect	
(semi-)local limit laws	contraction method	constant space	
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	
(semi-)local limit laws	contraction method	constant space	
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries...
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	branch misses
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	branch misses
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries . . .
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	branch misses
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, not done in libraries…
  ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	branch misses
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries…)
  - ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Don't we know everything about Quicksort by now?

- Extensive literature and results on Quicksort

Type of result	Analysis Techniques	Algorithm variants	Cost Measures
expected costs	telescoping recurrences	pivot sampling	key comparisons
variance	singularity analysis	Insertionsort cutoff	symbol comparisons
tail inequalities	Euler differential eq.	multiway partitioning	swaps
limit distributions	Martingales	Quickselect	scanned elements
(semi-)local limit laws	contraction method	constant space	branch misses
	branching processes		
	continuous master theorem		

- **but:** most results consider **random permutations** as input!
- partly justified: we can (should!) randomize Quicksort, (not done in libraries…)
  - ⇝ **every** input appears randomly ordered
- **Catch:** Elements with **equal keys** won't go away!

Setup

Assumptions:

1. **Input:** (A)
   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$

   (B)
   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - all **duplicates** of pivots removed
   - $\rightsquigarrow$ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible) not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$

   (B)

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - all **duplicates** of pivots removed
   - $\rightsquigarrow$ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B)

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - all **duplicates** of pivots removed
   - ⇝ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$ — not today
- (extension to asymmetric sampling possible)

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - all **duplicates** of pivots removed
   - ⇝ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - all **duplicates** of pivots removed
   - ⇝ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:
- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

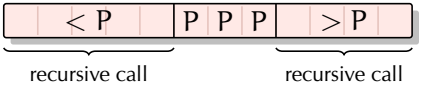
Multiset Model:
Random permutation $U_1, \ldots, U_n$ of **fixed multiset**
$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:
- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

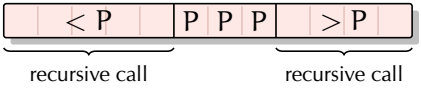
1. **Input:** **(A)** 

Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed



⇝ subproblems of **same** type, (restricted to a sub-universe)

3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$ — not today
- (extension to asymmetric sampling possible)

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

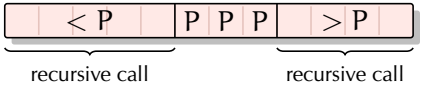
Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \overset{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

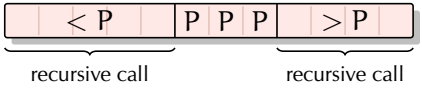
   Multiset Model:
   Random permutation $U_1, \dots, U_n$ of **fixed multiset**
   $x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
   ↖ profile $\vec{x}$ of input

   (B) 

   Discrete i.i.d. Model:
   $U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \overset{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
   $\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

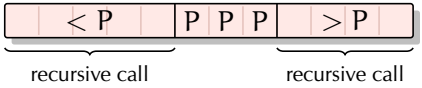
Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$

Setup

Assumptions:

1. **Input:** **(A)** 

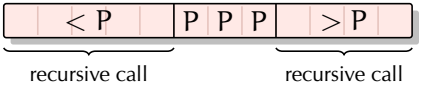
Multiset Model:
Random permutation $U_1, \dots, U_n$ of **fixed multiset**
$x_1, \dots, x_u$ number of occurrences of values $1, \dots, u$
profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \dots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \dots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)



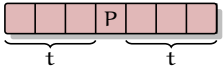
3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible)

not today

Example:
$t = 3$



Setup

Assumptions:

1. **Input:** **(A)** 

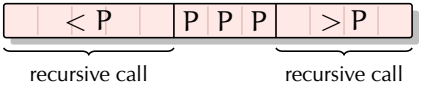
Multiset Model:
Random permutation $U_1, \ldots, U_n$ of **fixed multiset**
$x_1, \ldots, x_u$ number of occurrences of values $1, \ldots, u$
↖ profile $\vec{x}$ of input

(B) 

Discrete i.i.d. Model:
$U_1, \ldots, U_n$ i.i.d. with $\Pr[U_1 = v] = q_v$ ⇝ random profile $\vec{X} \stackrel{\mathcal{D}}{=} \mathrm{Mult}(n, \vec{q})$
$\vec{q} = (q_1, \ldots, q_u)$ a **fixed** universe **distribution**

2. **fat-pivot** partitioning
   - **all duplicates** of pivots removed

   ⇝ subproblems of **same** type, (restricted to a sub-universe)

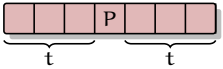


3. **Cost:** # ternary comparisons

Median-of-(2t+1) Quicksort:

- median-of-$(2t+1)$
- (extension to asymmetric sampling possible) ← not today

Example:
$t = 3$



Previous work on equal keys

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 on equal keys

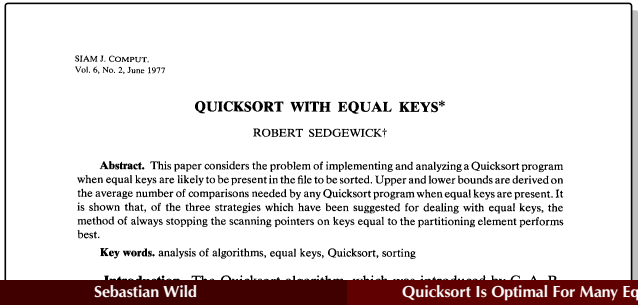
Rather little is known!

- **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

Previous work on equal keys

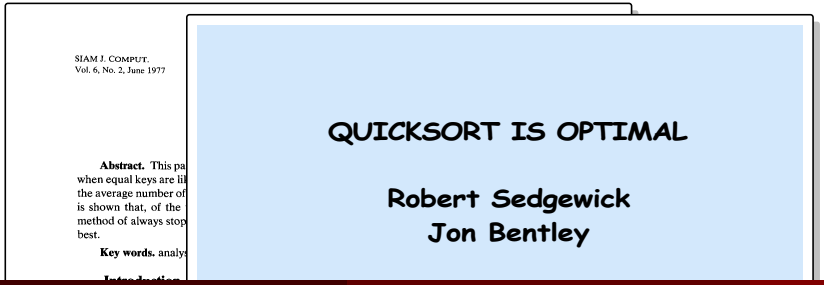
Rather little is known!

- **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 on equal keys

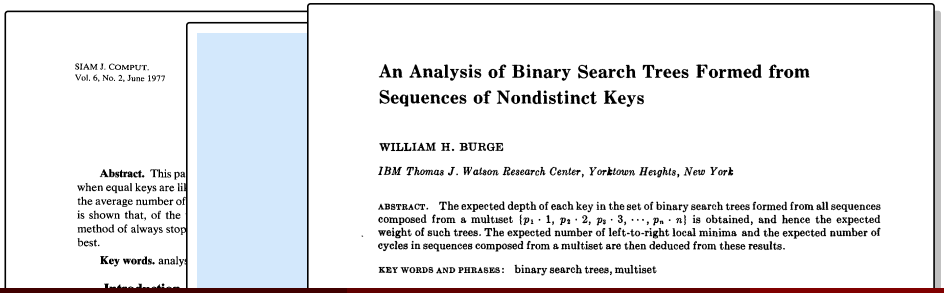
Rather little is known!

- **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

Abstract. This pa
when equal keys are li
the average number of
is shown that, of the
method of always stop
best.

Key words. analy

An Analysis of Binary Search Trees Formed from Sequences of Nondistinct Keys

WILLIAM H. BURGE

IBM Thomas J. Watson Research Center, Yorktown Heights, New York

ABSTRACT. The expected depth of each key in the set of binary search trees formed from all sequences composed from a multiset $\{p_1 \cdot 1,\ p_2 \cdot 2,\ p_3 \cdot 3, \cdots, p_n \cdot n\}$ is obtained, and hence the expected weight of such trees. The expected number of left-to-right local minima and the expected number of cycles in sequences composed from a multiset are then deduced from these results.

KEY WORDS AND PHRASES: binary search trees, multiset

Previous work on equal keys

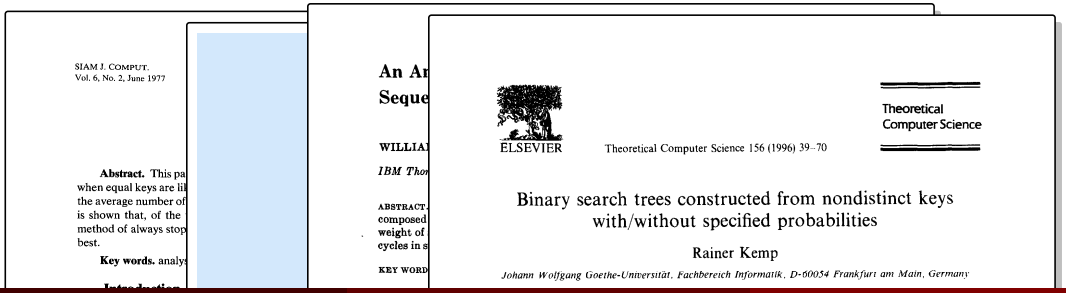
Rather little is known!

- **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

Abstract. This pa
when equal keys are li
the average number of
is shown that, of the
method of always stop
best.

Key words. analy

ELSEVIER

Theoretical Computer Science 156 (1996) 39–70

Theoretical Computer Science

Binary search trees constructed from nondistinct keys with/without specified probabilities

Rainer Kemp

Johann Wolfgang Goethe-Universität, Fachbereich Informatik, D-60054 Frankfurt am Main, Germany

Previous work on equal keys

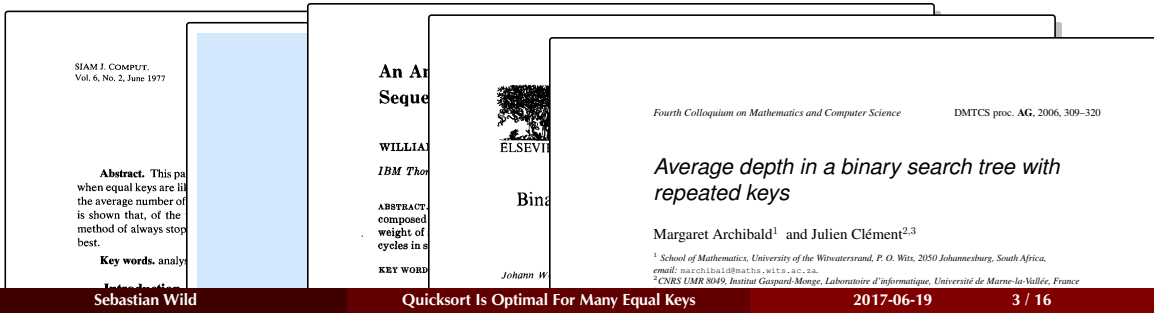
Rather little is known!

- **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 on equal keys

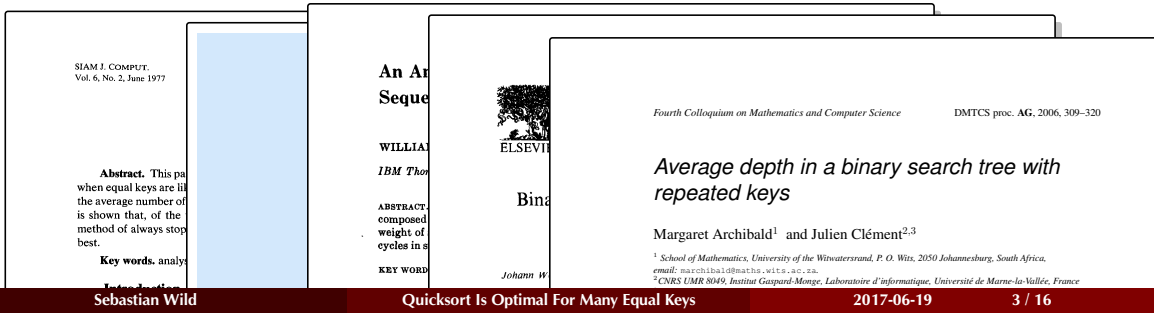
Rather little is known!

- **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

Abstract. This pa
when equal keys are li
the average number of
is shown that, of the
method of always stop
best.

Key words. analy

An Ar
Seque

WILLIA
IBM Tho

ABSTRACT.
composed
weight of
cycles in s

KEY WORD

Johann W

ELSEVI

Bina

Fourth Colloquium on Mathematics and Computer Science DMTCS proc. **AG**, 2006, 309–320

Average depth in a binary search tree with repeated keys

Margaret Archibald[1] and Julien Clément[2,3]

[1] *School of Mathematics, University of the Witwatersrand, P. O. Wits, 2050 Johannesburg, South Africa, email:* marchibald@maths.wits.ac.za
[2] *CNRS UMR 8049, Institut Gaspard-Monge, Laboratoire d'informatique, Université de Marne-la-Vallée, France*

Previous work on equal keys

Rather little is known!

- **Sedgewick** 1977: *Quicksort on Equal Keys*
- **Sedgewick & Bentley** 2002: *Quicksc*

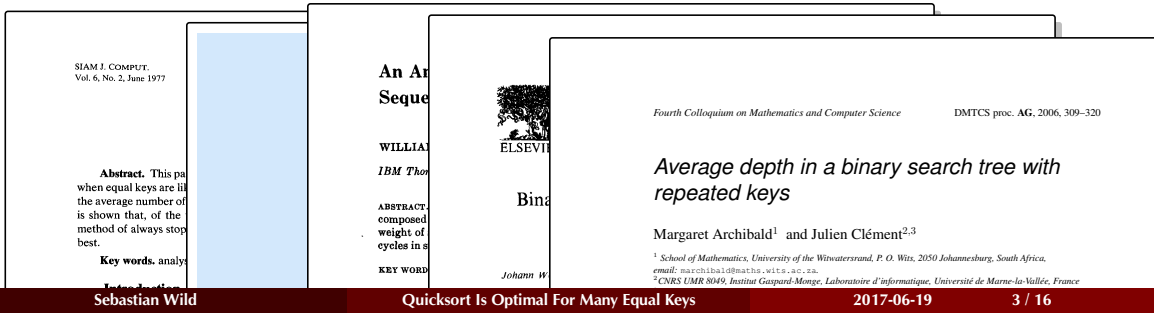
A bit more on BSTs:

- **Burge** 1976: *An Analysis of BSTs Forme*
- **Kemp** 1996: *BSTs constructed from non-distinct keys with/without specified probabilities*
- **Archibald & Clément** 2006: *Average depth in a BST with repeated keys*

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

- **No sampling** to choose pivots.
- (No multiway partitioning.)

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



Sedgewick's analysis for classic Quicksort

Classic Quicksort:

Sedgewick's analysis for classic Quicksort

Classic Quicksort:

Analysis of Quicksort with equal keys

1. Define $C(x_1,\ldots,x_n) = C(1,n)$ to be the mean # compares to sort the file

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j(C(1,j-1) + C(j+1,n))$$

2. Multiply both sides by $N = x_1 + \ldots + x_n$

$$NC(1,n) = N(N-1) + \sum_{1\le j\le n} x_j C(1,j-1) + \sum_{1\le j\le n} x_j C(j+1,n)$$

3. Subtract same equation for $x_2,\ldots,x_n$ and let $D(1,n) = C(1,n) - C(2,n)$

$$(x_1 + \ldots + x_n)D(1,n) = x_1^2 - x_1 + 2x_1(x_2 + \ldots + x_n) + \sum_{2\le j\le n} x_j D(1,j-1)$$

4. Subtract same equation for $x_1,\ldots,x_{n-1}$

$$(x_1 + \ldots + x_n)D(1,n) - (x_1 + \ldots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,n-1)$$

Sedgewick's analysis for classic Quicksort

Classic Quicksort:

Analysis of Quicksort with equal keys

1. Define $C(x_1,\dots,x_n) = C(1,n)$ to be the mean # compares to sor

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j(C(1,j-1) + C(j+1,n))$$

2. Multiply both sides by $N = x_1 + \dots + x_n$

$$NC(1,n) = N(N-1) + \sum_{1\le j\le n} x_j C(1,j-1) + \sum_{1\le j\le n} x_j C(j+1,n)$$

3. Subtract same equation for $x_2,\dots,x_n$ and let $D(1,n) = C(1,n) -$

$$(x_1 + \dots + x_n)D(1,n) = x_1^2 - x_1 + 2x_1(x_2 + \dots + x_n) + \sum_{2\le j\le n} x_j D(1,j$$

4. Subtract same equation for $x_1,\dots,x_{n-1}$

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,$$

Analysis of Quicksort with equal keys (cont.)

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,n-1)$$

5. Simplify, divide both sides by $N = x_1 + \dots + x_n$

$$D(1,n) = D(1,n-1) + \frac{2x_1x_n}{x_1 + \dots + x_n}$$

6. Telescope (twice)

$$C(1,n) = N - n + \sum_{1\le k<j\le n} \frac{2x_kx_j}{x_k + \dots + x_j}$$

THEOREM. Quicksort (with 3-way partitioning, randomized) uses

$N - n + 2QN$ compares (where $Q = \sum_{1\le k<j\le n} \frac{p_kp_j}{p_k + \dots + p_j}$, with $p_i = x_i/N$)

to sort an $(x_1,\dots,x_n)$–file, on the average .

Sedgewick's analysis for classic Quicksort

Classic Quicksort: Expected comparisons expressible **exactly**.

Analysis of Quicksort with equal keys

1. Define $C(x_1,\dots,x_n) \equiv C(1,n)$ to be the mean # compares to sor

$$C(1,n) = N - 1 + \frac{1}{N}\sum_{1\le j\le n} x_j(C(1,j-1) + C(j+1,n))$$

2. Multiply both sides by $N = x_1 + \dots + x_n$

$$NC(1,n) = N(N-1) + \sum_{1\le j\le n} x_j C(1,j-1) + \sum_{1\le j\le n} x_j C(j+1,n)$$

3. Su

4. Su

Analysis of Quicksort with equal keys (cont.)

$$(x_1 + \dots + x_n)D(1,n) - (x_1 + \dots + x_{n-1})D(1,n-1) = 2x_1x_n + x_nD(1,n-1)$$

5. Simplify, divide both sides by $N = x_1 + \dots + x_n$

$$D(1,n) = D(1,n-1) + \frac{2x_1x_n}{x_1 + \dots + x_n}$$

6. Telescope (twice)

$1\le k<j\le n$ $x_k + \dots + x_j$

THEOREM. Quicksort (with 3-way partitioning, randomized) uses

$$N - n + 2QN \text{ compares (where } Q = \sum_{1\le k<j\le n} \frac{p_k p_j}{p_k + \dots + p_j}, \text{ with } p_i = x_i/N)$$

to sort an $(x_1,\dots,x_n)$-file, on the average .

The conjecture of Sedgewick and Bentley

Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2 NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.

The conjecture of Sedgewick and Bentley

Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2 NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.

The conjecture of Sedgewick and Bentley

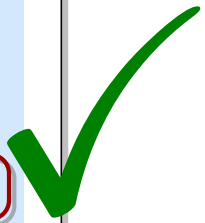
Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2 NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.



* subject to some assumptions

The conjecture of Sedgewick and Bentley

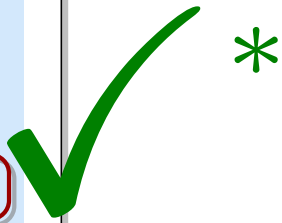
Quicksort is optimal

The average number of compares per element C/N is always within a constant factor of the entropy H

lower bound: $C > NH - N$ (information theory)
upper bound: $C < 2\ln 2 NH + N$ (Burge analysis, Melhorn bound)

No comparison-based algorithm can do better.

Conjecture: With sampling, $C / N \to H$ as sample size increases.



* subject to some assumptions

Outline

0 Intro

1 Quicksort and Search Trees

2 Saturated Fringe-Balanced Trees

3 Back to Multiset Permutations

Quicksort & search trees

Classic Fact:

- 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 & 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)



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)

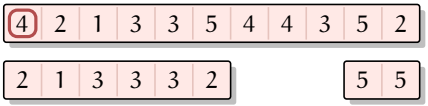


Quicksort & search trees

Classic Fact: (**without** duplicates)

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

Quicksort (Fat-Pivot)

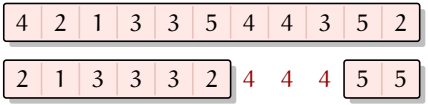


Quicksort & search trees

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)

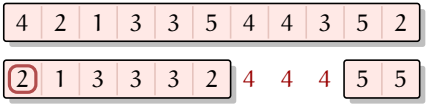


Quicksort & search trees

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)

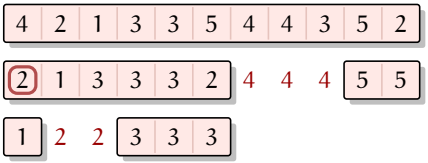


Quicksort & search trees

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)



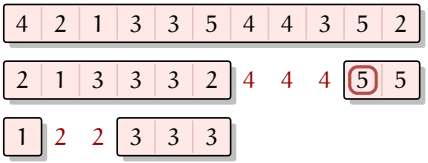
Quicksort & search trees

Classic Fact: (**without** duplicates)

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

- How about inputs with duplicates?

Quicksort (Fat-Pivot)

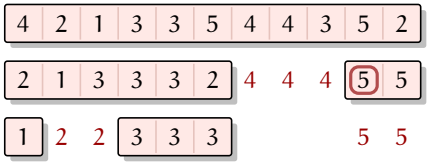


Quicksort & search trees

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)



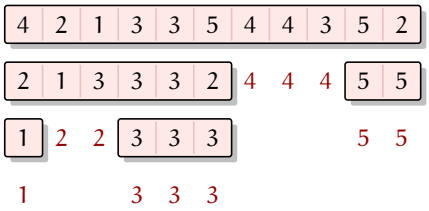
Quicksort & search trees

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)



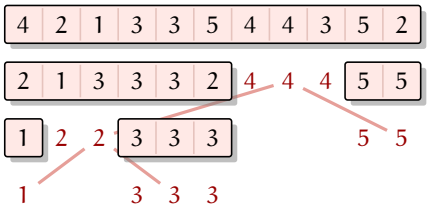
Quicksort & search trees

Classic Fact: (**without** duplicates)

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

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



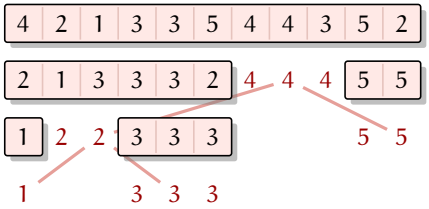
Quicksort & search trees

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

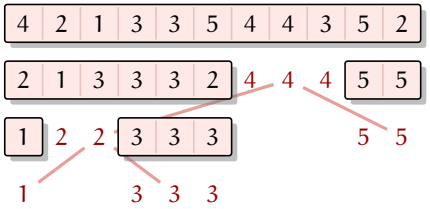
Quicksort & search trees

Classic Fact: (**without** duplicates)

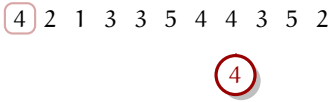
- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree

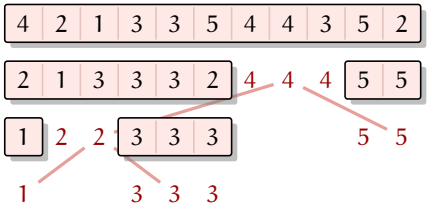


Quicksort & search trees

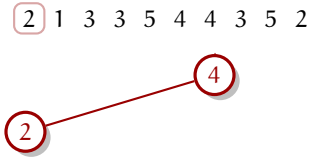
Classic Fact: (**without** duplicates)

- 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

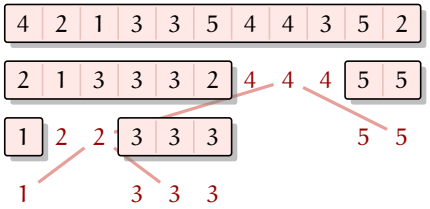


Quicksort & search trees

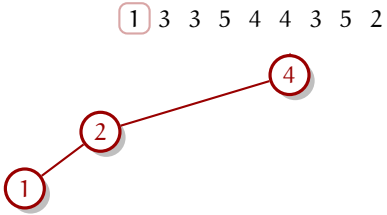
Classic Fact: (**without** duplicates)

- 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

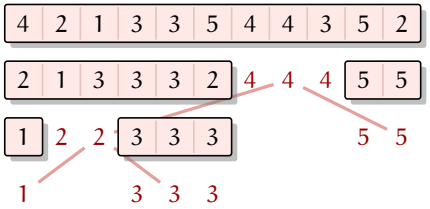


Quicksort & search trees

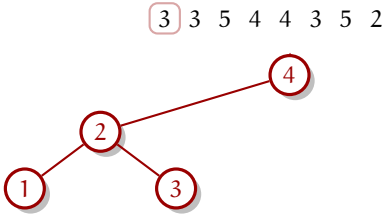
Classic Fact: (**without** duplicates)

- 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

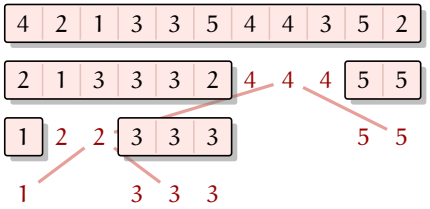


Quicksort & search trees

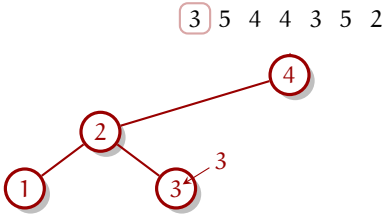
Classic Fact: (**without** duplicates)

- 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

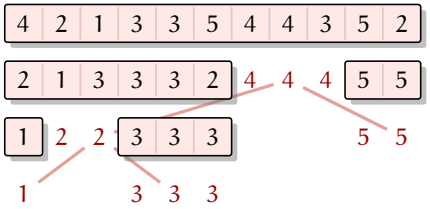


Quicksort & search trees

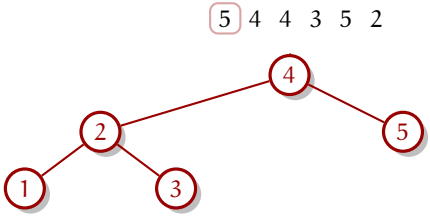
Classic Fact: (**without** duplicates)

- 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

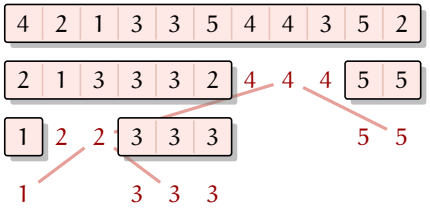


Quicksort & search trees

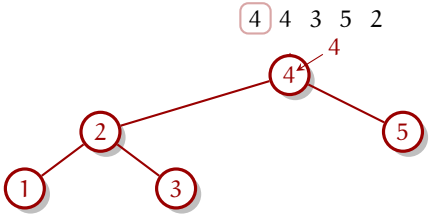
Classic Fact: (**without** duplicates)

- 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

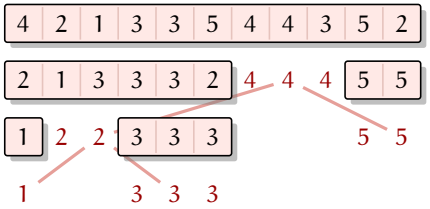


Quicksort & search trees

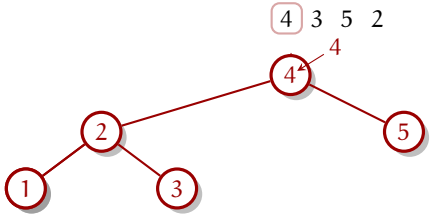
Classic Fact: (**without** duplicates)

- 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

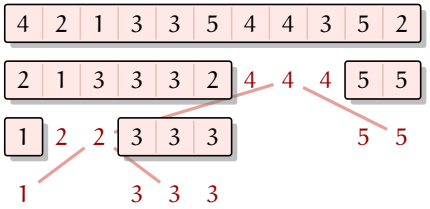


Quicksort & search trees

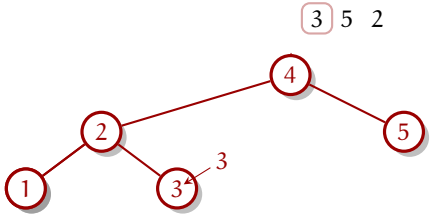
Classic Fact: (**without** duplicates)

- 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

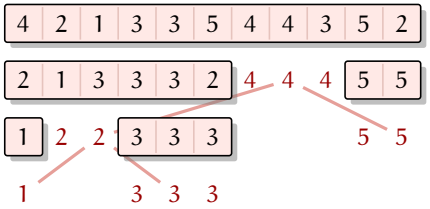


Quicksort & search trees

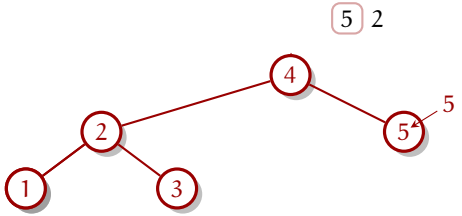
Classic Fact: (**without** duplicates)

- 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



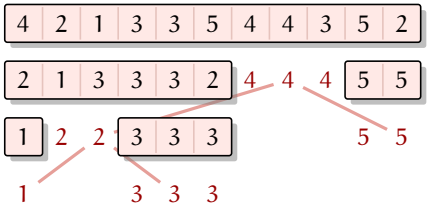
Quicksort & search trees

Classic Fact: (**without** duplicates)

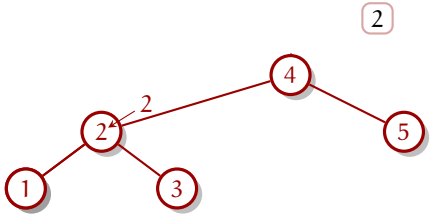
- Recursion Tree of Quicksort = Naturally grown BST from input
  - ⇝ Comparisons in Quicksort = Comparisons to **built** BST
    = Comparisons to **search** input in **final** BST

- How about inputs with duplicates?

Quicksort (Fat-Pivot)



Binary Search Tree



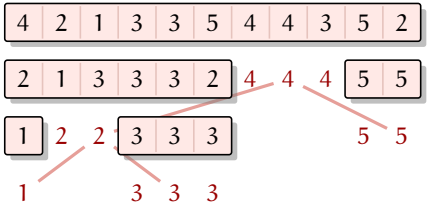
⇝ Equivalence holds also with duplicates.

Quicksort & search trees

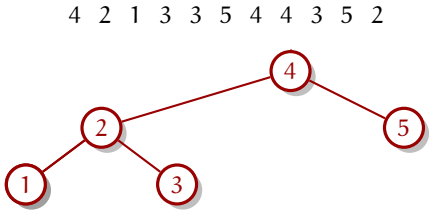
Classic Fact: (**without** duplicates)

- 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



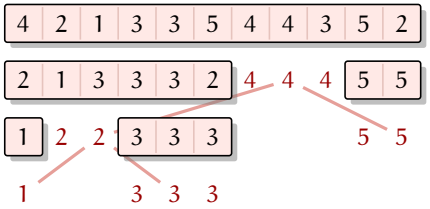
⇝ Equivalence holds also with duplicates.

Quicksort & search trees

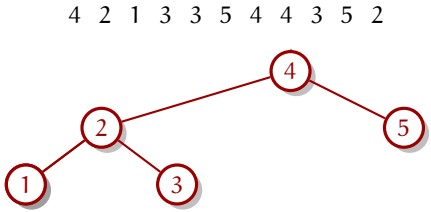
Classic Fact: (**without** duplicates)

- 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.

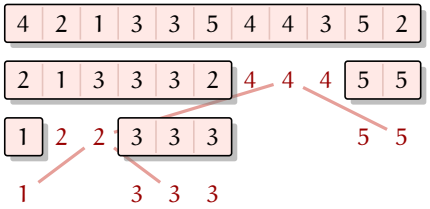
*This was only basic Quicksort . . . how about **pivot sampling**?*

Quicksort & search trees

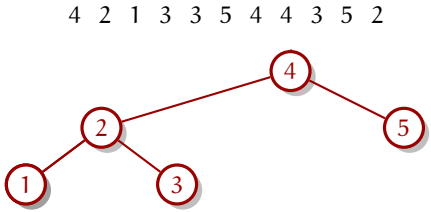
Classic Fact: (**without** duplicates)

- 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

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$

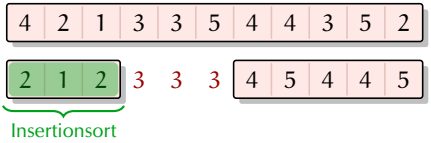


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort $(t = 2)$



Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)

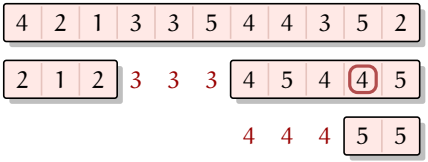


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)

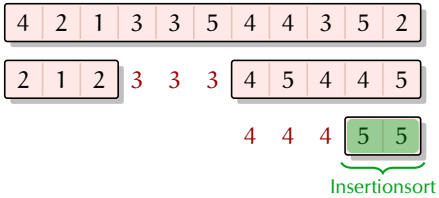


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**

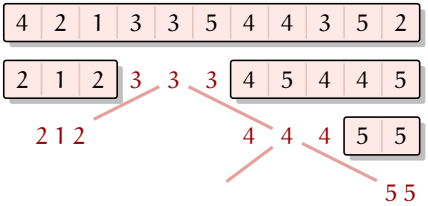


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**

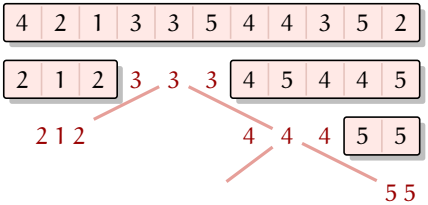


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

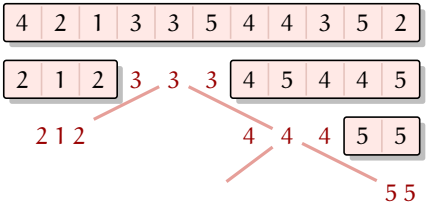


Fringe-balanced trees

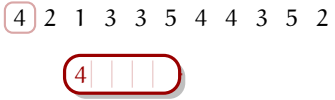
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

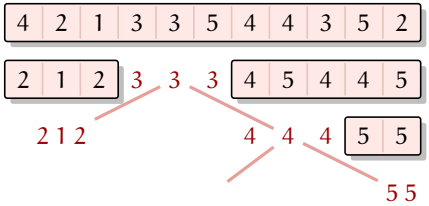


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

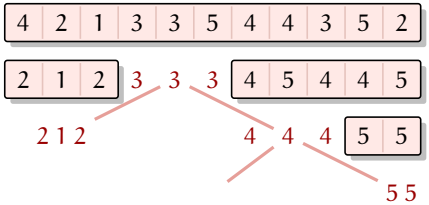


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

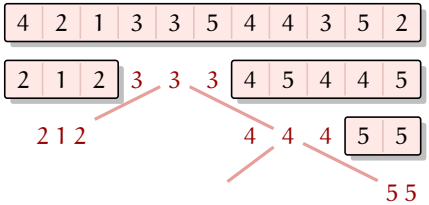


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

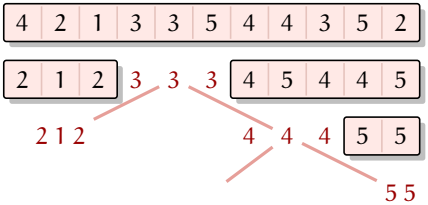


Fringe-balanced trees

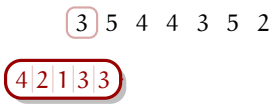
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

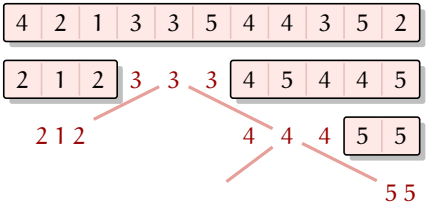


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

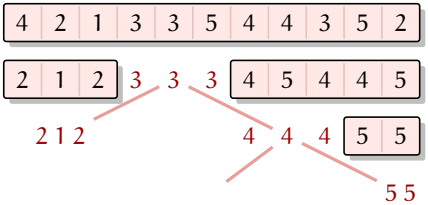


Fringe-balanced trees

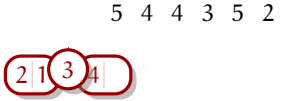
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

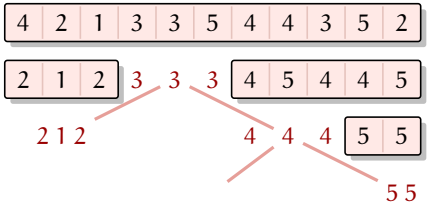


Fringe-balanced trees

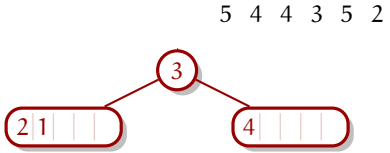
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

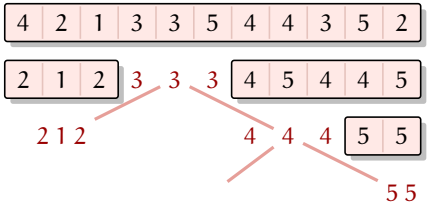


Fringe-balanced trees

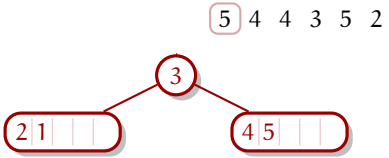
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

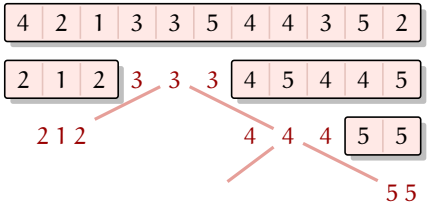


Fringe-balanced trees

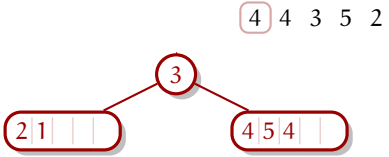
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

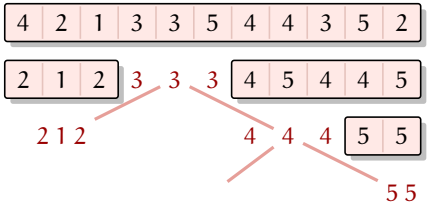


Fringe-balanced trees

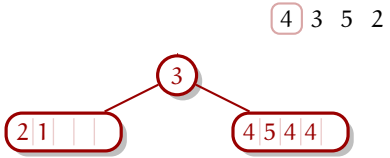
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

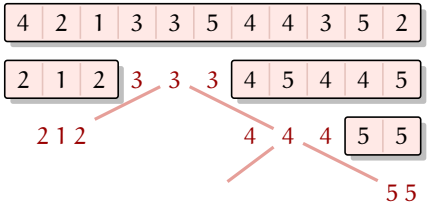


Fringe-balanced trees

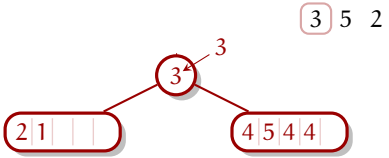
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

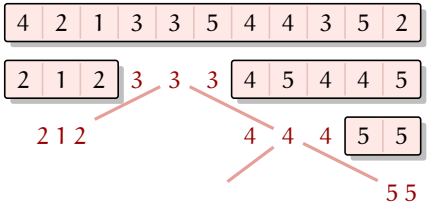


Fringe-balanced trees

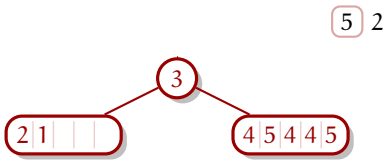
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

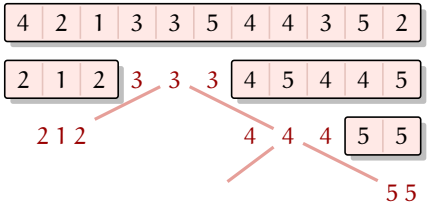


Fringe-balanced trees

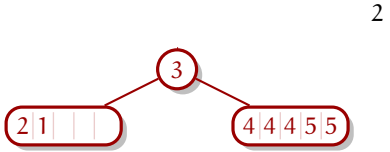
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

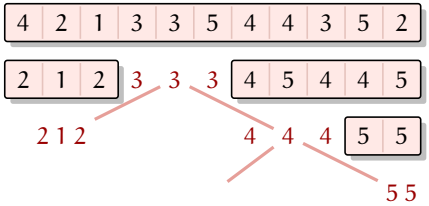


Fringe-balanced trees

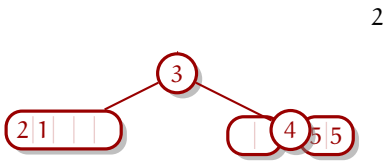
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

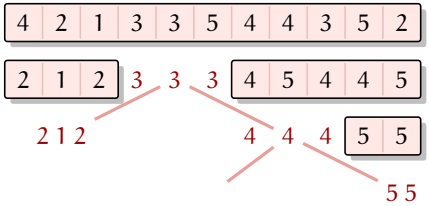


Fringe-balanced trees

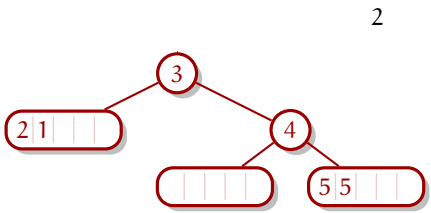
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree

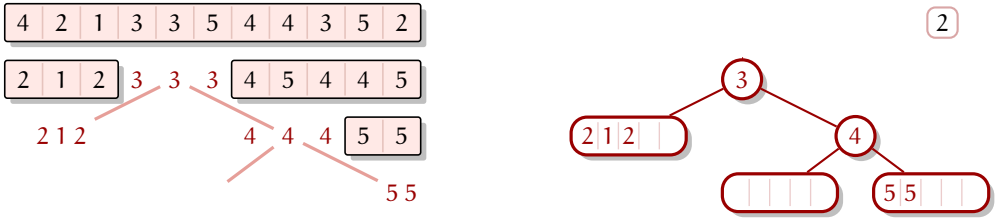


Fringe-balanced trees

k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$) **5-Fringe-Balanced Tree**



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (s-way Partitioning $\rightsquigarrow$ s-ary search trees) ← not today

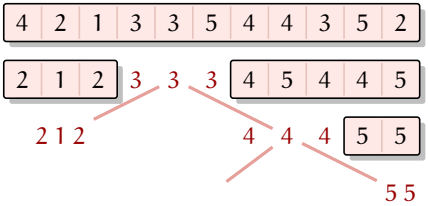
$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Fringe-balanced trees

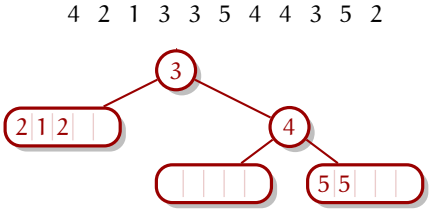
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

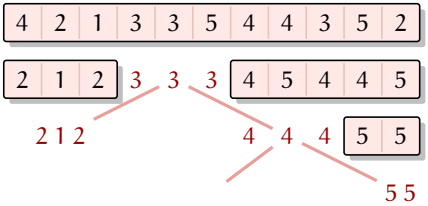
$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Fringe-balanced trees

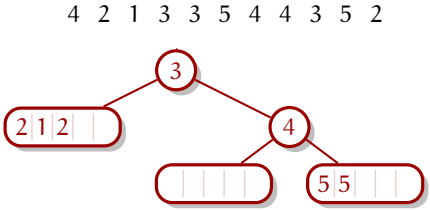
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

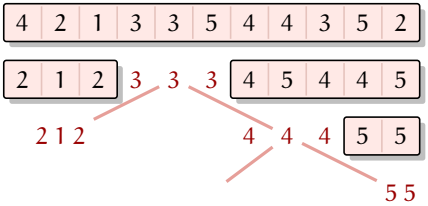
$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Fringe-balanced trees

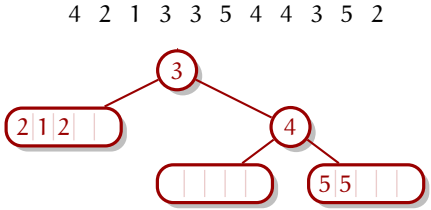
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

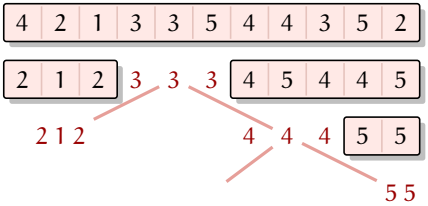
$\rightsquigarrow$ Analyze search trees instead of Quicksort.

Fringe-balanced trees

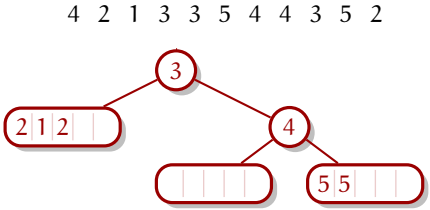
k-Fringe-Balanced Search Trees:

- Leaves **buffer** $k = 2t + 1$ elements.
- If buffer is full, leaf is **split** $\rightsquigarrow$ new internal node with **chosen pivot**.

Median-of-5 Quicksort ($t = 2$)



5-Fringe-Balanced Tree



$\rightsquigarrow$ **Correspondence extends to**

- Pivot Sampling (any scheme, not only median)
- (**s-way Partitioning** $\rightsquigarrow$ s-ary search trees) ← not today

$\rightsquigarrow$ Analyze search trees instead of Quicksort.

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

$(k = 1)$

Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

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

$(k = 1)$

Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

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

$(k = 1)$

Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

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

$(k = 1)$

Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

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

$(k = 1)$

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

$(k = 1)$

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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

(k = 1)

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 each value was inserted!

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 $\mathcal{T}$

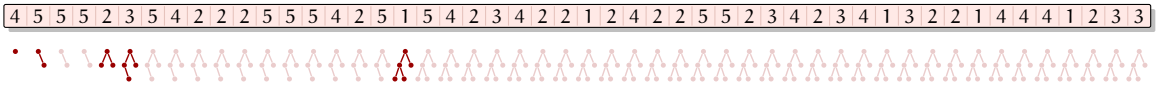
Tree-growing and searching

⇝ 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

($k = 1$)



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
⇝ stationary after each value inserted $k = 2t + 1$ **times**
(up to k duplicates in buffer)

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.

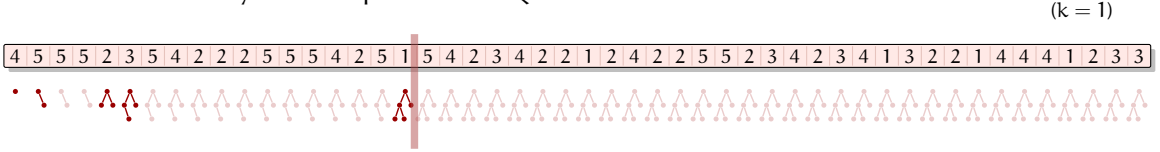
⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

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 each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)

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.

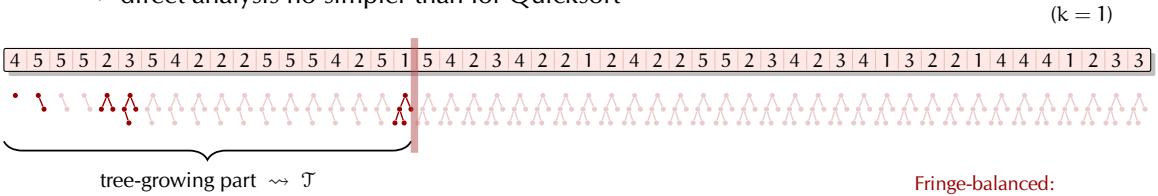
$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

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



Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times**
(up to k duplicates in buffer)

Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!



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.

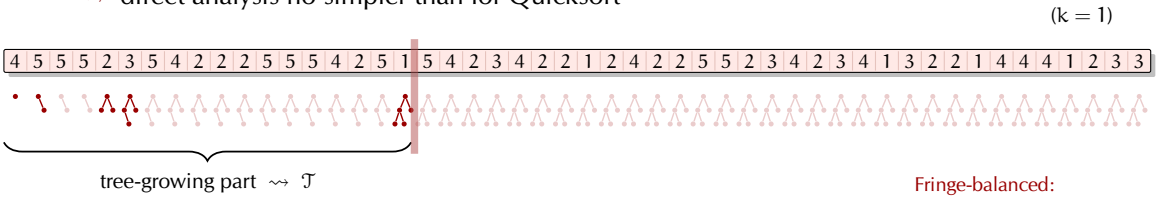
$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

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 each value was inserted!

Fringe-balanced:
$\rightsquigarrow$ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times**
(up to k duplicates in buffer)



Split input into **tree-growing** part and **searching** part: (hopefully a **short** prefix!)

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

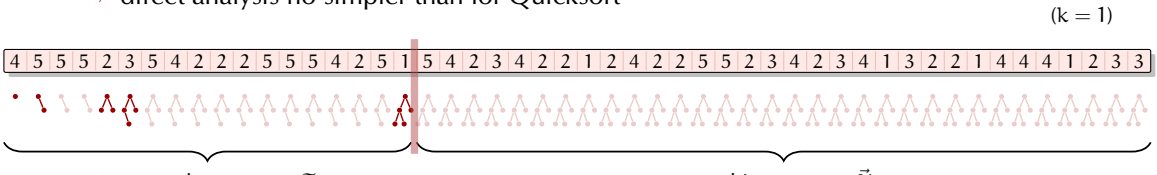
$\rightsquigarrow$ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

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



Observation: $\mathcal{T}$ becomes **stationary** after each value was inserted!

Fringe-balanced:
⇝ stationary after each value inserted $\mathbf{k = 2t + 1}$ **times** (up to k duplicates in buffer)

hopefully a **short** prefix!

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.

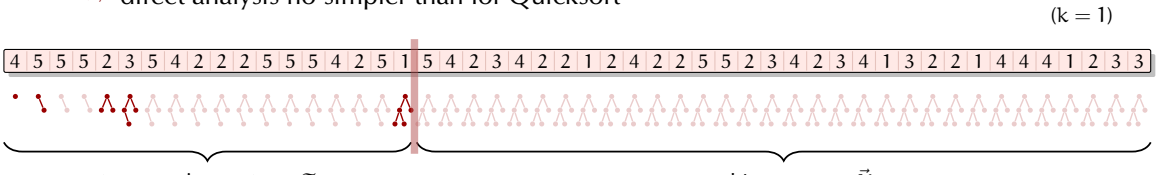
⇝ profile $\vec{X}_S$ of search part **independent** of $\mathcal{T}$

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 each value was inserted!

Fringe-balanced: $\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times** (up to k duplicates in buffer)

hopefully a **short** prefix!

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 $\mathcal{T}$

Two parts of input always dependent! (profiles must **sum to** $\vec{x}$)

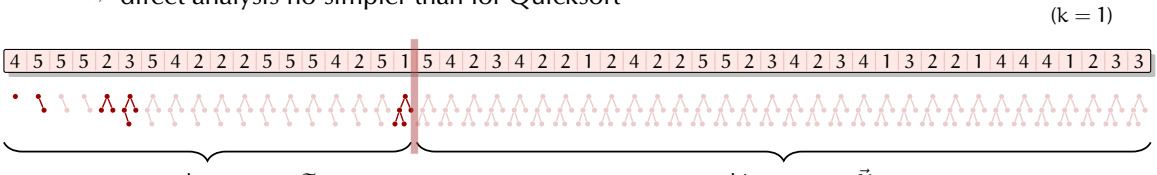
Two parts are **independent** (i.i.d.!)

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 each value was inserted!

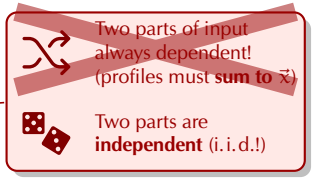
Fringe-balanced: $\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times** (up to k duplicates in buffer)



Split input into **tree-growing** part and **searching** part: (hopefully a **short** prefix!)

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 $\mathcal{T}$

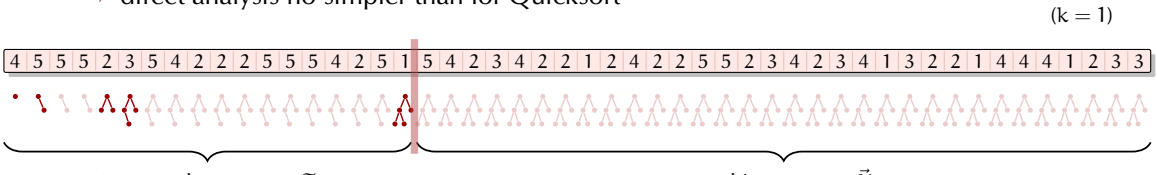


Tree-growing and searching

$\rightsquigarrow$ 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 each value was inserted!

Fringe-balanced: $\rightsquigarrow$ stationary after each value inserted $k = 2t + 1$ **times** (up to k duplicates in buffer)

hopefully a **short** prefix!

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 $\mathcal{T}$

~~Two parts of input always dependent! (profiles must **sum to** $\vec{x}$)~~

Two parts are **independent** (i.i.d.!)

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

$\rightsquigarrow$ Choose n_T large enough to make those **degenerate** inputs rare.

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

$n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

  Binomial tail bound

- **Costs to grow $\mathcal{T}$:**

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** ($\Pr[\text{degenerate}] = o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** ($\Pr[\text{degenerate}] = o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$

  can't use large n_T

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** ($\Pr[\text{degenerate}] = o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

  can't use large n_T

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** ($\Pr[\text{degenerate}] = o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

    can't use large n_T

  - **folklore result:** random BSTs have **logarithmic height** w. h. p.

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

    can't use large n_T

  - **folklore result:** random BSTs have **logarithmic height** w. h. p.
  - can extend this to fringe-balanced trees ⇝ $n_T \cdot O(\log n) = O(n^{1-\tilde{\varepsilon}} \log n)$ to build $\mathcal{T}$

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

$\rightsquigarrow$ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

$\rightsquigarrow$ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ $\rightsquigarrow$ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

    can't use large n_T

  - **folklore result:** random BSTs have **logarithmic height** w. h. p.
  - can extend this to fringe-balanced trees $\rightsquigarrow$ $n_T \cdot O(\log n) = O(n^{1-\tilde{\varepsilon}} \log n)$ to build $\mathcal{T}$

$\rightsquigarrow$ **Expected Quicksort costs:** $\mathbb{E}[C_{n,\vec{q}}] = \alpha(\vec{q}) \cdot n \pm O(n^{1-\delta})$ (for any $\delta \in (0, \varepsilon)$)

- $\alpha(\vec{q}) =$ **expected search cost** in random $\mathcal{T}$
- error term covers: tree-growing, sorting samples, degenerate inputs, inputs with high $\mathcal{T}$

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

    can't use large n_T

  - **folklore result:** random BSTs have **logarithmic height** w. h. p.
  - can extend this to fringe-balanced trees ⇝ $n_T \cdot O(\log n) = O(n^{1-\tilde{\varepsilon}} \log n)$ to build $\mathcal{T}$

⇝ **Expected Quicksort costs:** $\mathbb{E}[C_{n,\vec{q}}] = \alpha(\vec{q}) \cdot n \pm O(n^{1-\delta})$ (for any $\delta \in (0, \varepsilon)$)

- $\alpha(\vec{q}) =$ **expected search cost** in random $\mathcal{T}$
- error term covers: tree-growing, sorting samples, degenerate inputs, inputs with high $\mathcal{T}$

Bounding the tree-growing part

Goal: *ignore tree-growing for analysis.*



to be chosen

⇝ Allow only first n_T elements for tree growing.

Problem: if a value occurs $< k$ times in first n_T elements, $\mathcal{T}$ not complete

⇝ Choose n_T large enough to make those **degenerate** inputs rare.

- Require **"many duplicates"**: $\mathbb{E}[X_v] = \Omega(n^\varepsilon)$ for $\varepsilon > 0$ Note: implies $u = O(n^{1-\varepsilon})$

  Binomial tail bound

  $n_T = \lceil n^{1-\tilde{\varepsilon}} \rceil$ with $\tilde{\varepsilon} < \varepsilon$ ⇝ **non-degenerate w. h. p.** (Pr[degenerate] $= o(n^{-c})$ for any c)

- **Costs to grow $\mathcal{T}$:**
  - never more than $\leqslant n_T \cdot u$ … but that is too coarse! ($n_T \cdot u$ can be close to n^2)

    can't use large n_T

  - **folklore result:** random BSTs have **logarithmic height** w. h. p.
  - can extend this to fringe-balanced trees ⇝ $n_T \cdot O(\log n) = O(n^{1-\tilde{\varepsilon}} \log n)$ to build $\mathcal{T}$

⇝ **Expected Quicksort costs:** $\mathbb{E}[C_{n,\vec{q}}] = \alpha(\vec{q}) \cdot n \pm O(n^{1-\delta})$ (for any $\delta \in (0, \varepsilon)$)

- $\alpha(\vec{q}) =$ **expected search cost** in random $\mathcal{T}$
- error term covers: tree-growing, sorting samples, degenerate inputs, inputs with high $\mathcal{T}$

Outline

0 Intro

1 Quicksort and Search Trees

2 Saturated Fringe-Balanced Trees

3 Back to Multiset Permutations

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$ $\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result: **Allen & Munro** 1978:
Self-Organizing Binary Search Trees

$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1$ with

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \cdots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \ldots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$ $\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result:

Allen & Munro 1978:
Self-Organizing Binary Search Trees

$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1$ with

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \cdots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \ldots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result:

Allen & Munro 1978:
Self-Organizing Binary Search Trees

$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1$ with

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \le i < j \le u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$ $\quad$ $\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

distribution with prob. weights $q_1, \dots, q_u$

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

Warmup: **Ordinary BSTs** ($t = 0$)

Old result: **Allen & Munro** 1978:
Self-Organizing Binary Search Trees

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

***Warmup:* Ordinary BSTs** ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Fringe-balanced trees ($t > 1$)

- probability of given value in root:

$\rightsquigarrow$ prob. that i inserted first among $i, \dots j$??

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

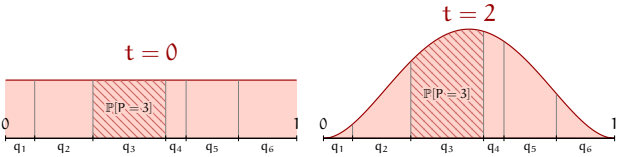
$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



⇝ prob. that i inserted first among $i, \dots j$??

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

distribution with prob. weights $q_1, \dots, q_u$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

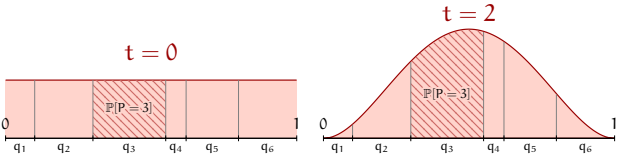
$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



$\rightsquigarrow$ prob. that i inserted first among $i, \dots j$??

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

distribution with prob. weights $q_1, \dots, q_u$

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

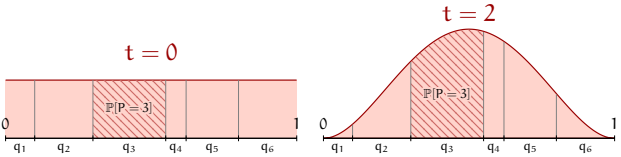
$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \cdots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



⇝ prob. that i inserted first among $i, \dots j$??

old approach does not work

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

distribution with prob. weights $q_1, \dots, q_u$

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

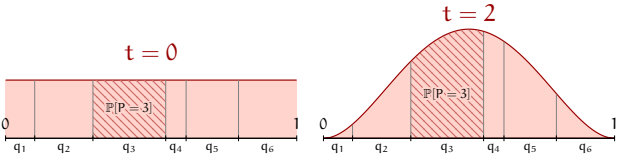
$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$
- In the same paper: $\mathcal{H}_Q(\vec{q}) < \mathcal{H}_{ln}(\vec{q})$

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



⇝ prob. that i inserted first among $i, \dots j$??

old approach does not work

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

distribution with prob. weights $q_1, \dots, q_u$

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

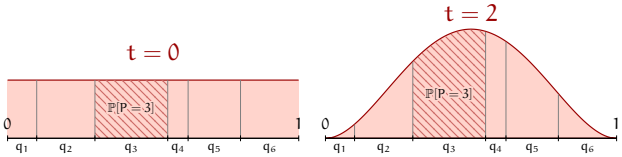
$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$
- In the same paper: $\mathcal{H}_Q(\vec{q}) < \mathcal{H}_{\ln}(\vec{q})$

base e Shannon entropy

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



⇝ prob. that i inserted first among $i, \dots j$??

old approach does not work

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

(distribution with prob. weights $q_1, \dots, q_u$)

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

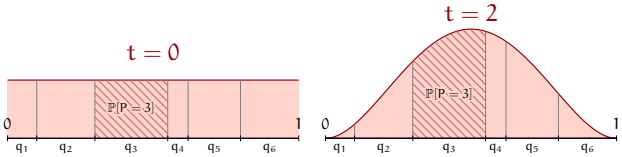
$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$
- In the same paper: $\mathcal{H}_Q(\vec{q}) < \mathcal{H}_{\ln}(\vec{q})$ (base e Shannon entropy)

$\rightsquigarrow$ $\alpha(\vec{q}) < 2 \ln 2 \cdot \mathcal{H}_{ld}(\vec{q}) + 1$, (base 2 entropy)

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



$\rightsquigarrow$ prob. that i inserted first among $i, \dots j$??

old approach does not work

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

(distribution with prob. weights $q_1, \dots, q_u$)

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

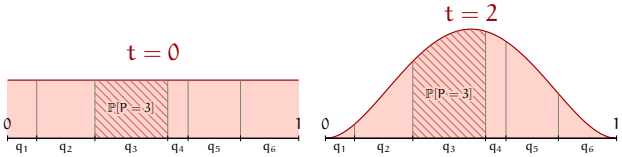
$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$
- In the same paper: $\mathcal{H}_Q(\vec{q}) < \mathcal{H}_{\ln}(\vec{q})$ (base e Shannon entropy)

$\rightsquigarrow$ $\alpha(\vec{q}) < 2 \ln 2 \cdot \mathcal{H}_{ld}(\vec{q}) + 1$, (base 2 entropy)
only factor $2 \ln 2 \approx 1.386$ from **optimal**!

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



$\rightsquigarrow$ prob. that i inserted first among $i, \dots j$??

old approach does not work

Search Costs in Saturated Trees

Recall: $\alpha(\vec{q}) = \sum_{v=1}^{u} q_v \cdot depth_{\mathcal{T}}(v)$

$\mathcal{T}$ from inserting i.i.d. $\mathcal{D}(\vec{q})$ elements **until saturation**

distribution with prob. weights $q_1, \dots, q_u$

Warmup: Ordinary BSTs ($t = 0$)

Old result: **Allen & Munro** 1978: *Self-Organizing Binary Search Trees*

$$\alpha(\vec{q}) = 2\mathcal{H}_Q(\vec{q}) + 1 \text{ with}$$

$$\mathcal{H}_Q(\vec{q}) = \sum_{1 \leqslant i < j \leqslant u} \frac{q_i q_j}{q_i + \dots + q_j}$$

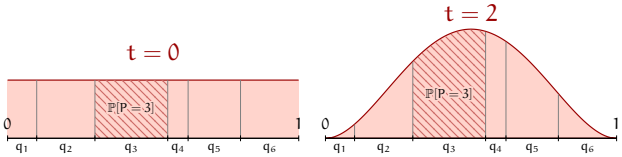
- **Proof sketch:**
  Sum prob. that i is ancestor of j over all i, j
  ancestor $\iff$ i **first** inserted key among $i, \dots, j$
- In the same paper: $\mathcal{H}_Q(\vec{q}) < \mathcal{H}_{\ln}(\vec{q})$ (base e Shannon entropy)

$\rightsquigarrow$ $\alpha(\vec{q}) < 2 \ln 2 \cdot \mathcal{H}_{ld}(\vec{q}) + 1$, (base 2 entropy)

only factor $2 \ln 2 \approx 1.386$ from **optimal**!

Fringe-balanced trees ($t > 1$)

- probability of given value in root:



$\rightsquigarrow$ prob. that i inserted first among $i, \dots j$??

old approach does not work

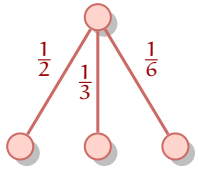
$\rightsquigarrow$ Try to generalize this!

Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***

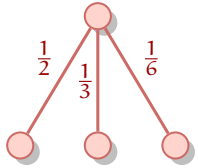
Aggregation of Entropy

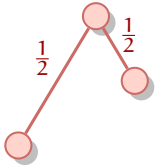
- One of the defining properties of Shannon entropy: ***aggregation***



Aggregation of Entropy

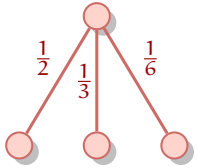
- One of the defining properties of Shannon entropy: ***aggregation***

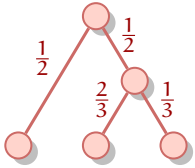




Aggregation of Entropy

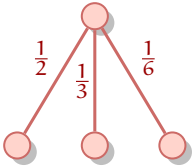
- One of the defining properties of Shannon entropy: ***aggregation***



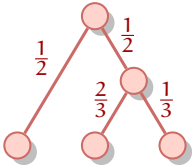


Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***

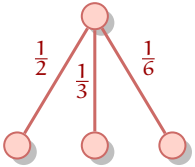


$\mathcal{H}(\frac{1}{2}, \frac{1}{3}, \frac{1}{6})$



Aggregation of Entropy

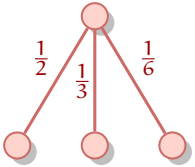
- One of the defining properties of Shannon entropy: ***aggregation***



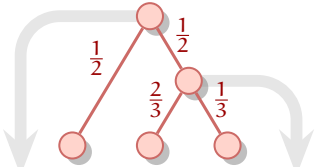
$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot \quad + \tfrac{1}{2}\cdot$$

Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***

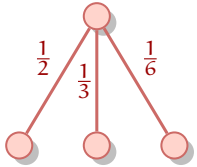


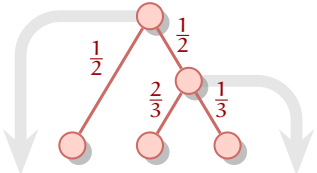
$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



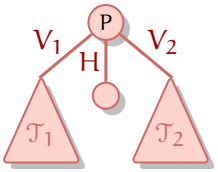
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$

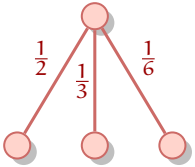


$\rightsquigarrow$ First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

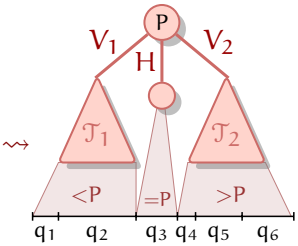
$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{matrix} Z_1 = (\frac{q_1}{V_1}, \ldots, \frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2}, \ldots, \frac{q_u}{V_2}) \end{matrix}$$

Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***



$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$

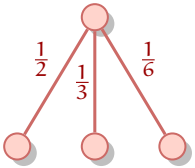


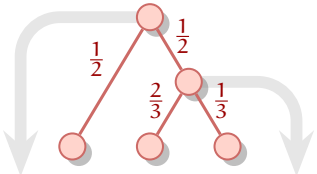
First partitioning step / Root of BST: Split into $\boxed{<P}$, $\boxed{=P}$, $\boxed{>P}$

$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{matrix} Z_1 = (\frac{q_1}{V_1},\ldots,\frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2},\ldots,\frac{q_u}{V_2}) \end{matrix}$$

Aggregation of Entropy

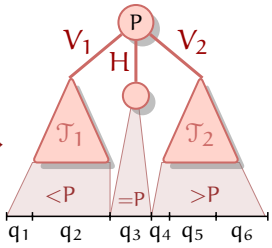
- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \;=\; \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) \;+\; \tfrac{1}{2}\cdot 0 \;+\; \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$

$\rightsquigarrow$

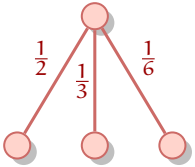


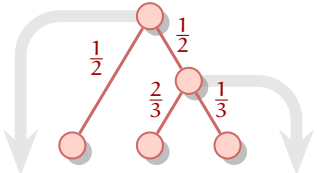
First partitioning step / Root of BST: Split into $\boxed{<P}$, $\boxed{=P}$, $\boxed{>P}$

$$\rightsquigarrow\quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad Z_1 = \left(\tfrac{q_1}{V_1},\ldots,\tfrac{q_{P-1}}{V_1}\right) \quad Z_2 = \left(\tfrac{q_{P+1}}{V_2},\ldots,\tfrac{q_u}{V_2}\right)$$

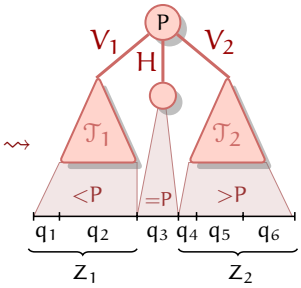
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \;=\; \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) \;+\; \tfrac{1}{2}\cdot 0 \;+\; \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$

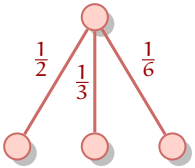


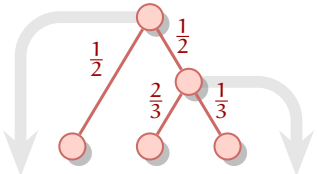
First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

$$\rightsquigarrow\quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{array}{l} Z_1 = (\frac{q_1}{V_1},\ldots,\frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2},\ldots,\frac{q_u}{V_2}) \end{array}$$

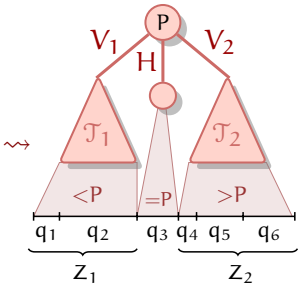
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



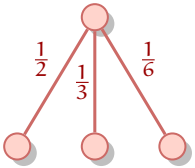
First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

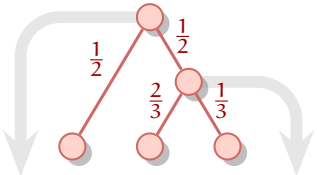
$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{array}{l} Z_1 = (\frac{q_1}{V_1},\ldots,\frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2},\ldots,\frac{q_u}{V_2}) \end{array}$$

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j)$

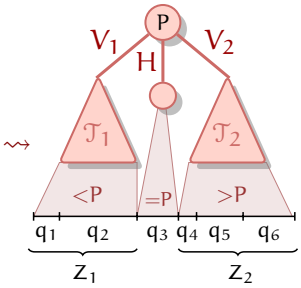
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \;=\; \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

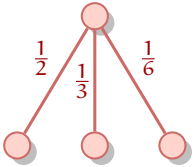
$$\rightsquigarrow\quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad Z_1 = (\tfrac{q_1}{V_1},\dots,\tfrac{q_{P-1}}{V_1}),\quad Z_2 = (\tfrac{q_{P+1}}{V_2},\dots,\tfrac{q_u}{V_2})$$

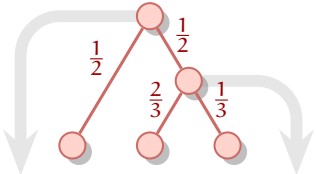
same shape!

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j)$

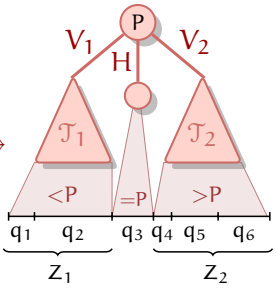
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



$\rightsquigarrow$ First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j)$$

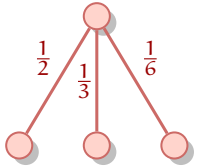
same shape!

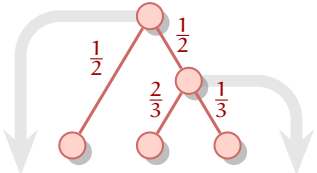
$$Z_1 = \left(\tfrac{q_1}{V_1},\ldots,\tfrac{q_{P-1}}{V_1}\right)$$
$$Z_2 = \left(\tfrac{q_{P+1}}{V_2},\ldots,\tfrac{q_u}{V_2}\right)$$

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j)$ $\rightsquigarrow$ $\mathcal{H}(V_1, H, V_2)$ vs. 1?

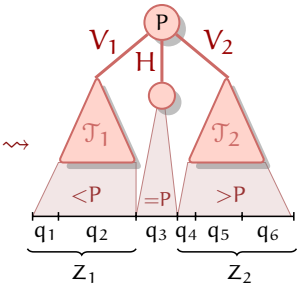
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

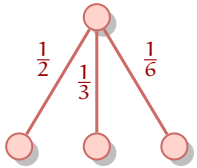
$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{matrix} Z_1 = (\frac{q_1}{V_1}, \ldots, \frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2}, \ldots, \frac{q_u}{V_2}) \end{matrix}$$

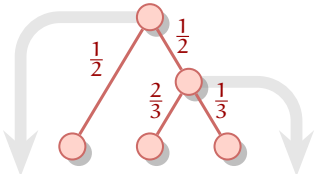
same shape!

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j) \quad \rightsquigarrow \quad \mathcal{H}(V_1, H, V_2)$ vs. 1?
- **Technical Issues**
  1. Pivot P is random $\rightsquigarrow$ take expectations over P (and thus $V_{1,2}$, $Z_{1,2}$).
  2. $\mathbb{E}\big[\mathcal{H}_{\ln}(V_1, H, V_2)\big] \;\approx\; \mathbb{E}\big[\mathcal{H}_{\ln}(D, 1-D)\big] \;=\; H_{k+1} - H_{t+1}$ where $D \overset{\mathcal{D}}{=} \mathrm{Beta}(t+1, t+1)$ but not an inequality in either direction

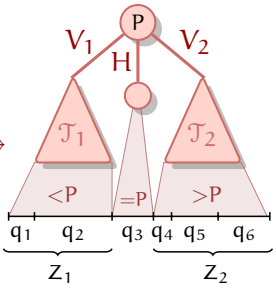
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

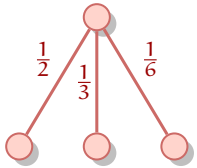
$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) = \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad Z_1 = (\tfrac{q_1}{V_1}, \ldots, \tfrac{q_{P-1}}{V_1}),\ Z_2 = (\tfrac{q_{P+1}}{V_2}, \ldots, \tfrac{q_u}{V_2})$$

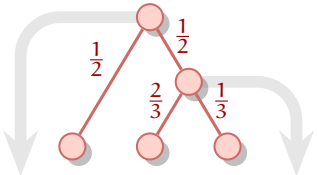
same shape!

- Recurrence for **search costs**: $\alpha(\vec{q}) = 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j) \quad \rightsquigarrow$ $\mathcal{H}(V_1, H, V_2)$ vs. 1?
- **Technical Issues**
  1. Pivot P is random $\rightsquigarrow$ take expectations over P (and thus $V_{1,2}$, $Z_{1,2}$).
  2. $\mathbb{E}[\mathcal{H}_{\ln}(V_1, H, V_2)] \approx \mathbb{E}[\mathcal{H}_{\ln}(D, 1-D)] = H_{k+1} - H_{t+1}$ where $D \overset{\mathcal{D}}{=} \mathrm{Beta}(t+1, t+1)$
     but not an inequality in either direction

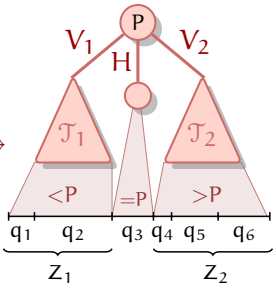
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \quad = \quad \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) + \tfrac{1}{2}\cdot 0 + \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

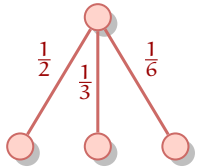
$$\rightsquigarrow \quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1, H, V_2) + \sum_{j=1}^{2} V_j \cdot \mathcal{H}(Z_j) \qquad \begin{matrix} Z_1 = (\frac{q_1}{V_1}, \ldots, \frac{q_{P-1}}{V_1}) \\ Z_2 = (\frac{q_{P+1}}{V_2}, \ldots, \frac{q_u}{V_2}) \end{matrix}$$

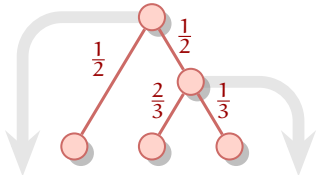
same shape!

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j \cdot \alpha(Z_j) \quad \rightsquigarrow \quad \mathcal{H}(V_1, H, V_2)$ vs. 1?
- **Technical Issues**
  1. Pivot P is random $\rightsquigarrow$ take expectations over P (and thus $V_{1,2}$, $Z_{1,2}$).
  2. $\mathbb{E}\big[\mathcal{H}_{\ln}(V_1, H, V_2)\big] \;\approx\; \mathbb{E}\big[\mathcal{H}_{\ln}(D, 1-D)\big] \;=\; H_{k+1} - H_{t+1}$ where $D \overset{\mathcal{D}}{=} \text{Beta}(t+1, t+1)$
     but **not an inequality** in either direction

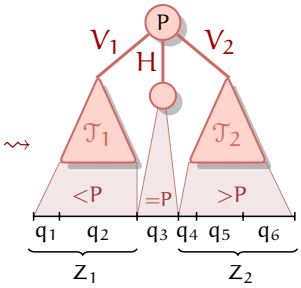
Aggregation of Entropy

- One of the defining properties of Shannon entropy: ***aggregation***





$$\mathcal{H}(\tfrac{1}{2},\tfrac{1}{3},\tfrac{1}{6}) \;=\; \mathcal{H}(\tfrac{1}{2},\tfrac{1}{2}) \;+\; \tfrac{1}{2}\cdot 0 \;+\; \tfrac{1}{2}\cdot\mathcal{H}(\tfrac{2}{3},\tfrac{1}{3})$$



First partitioning step / Root of BST: Split into $<P$, $=P$, $>P$

$$\rightsquigarrow\quad \mathcal{H}(\vec{q}) \;=\; \mathcal{H}(V_1,H,V_2) + \sum_{j=1}^{2} V_j\cdot\mathcal{H}(Z_j) \qquad Z_1 = (\tfrac{q_1}{V_1},\ldots,\tfrac{q_{P-1}}{V_1}),\quad Z_2 = (\tfrac{q_{P+1}}{V_2},\ldots,\tfrac{q_u}{V_2})$$

same shape!

- Recurrence for **search costs**: $\alpha(\vec{q}) \;=\; 1 + \sum_{j=1}^{2} V_j\cdot\alpha(Z_j) \quad\rightsquigarrow\quad \mathcal{H}(V_1,H,V_2)$ vs. 1?
- **Technical Issues**
  1. Pivot P is random $\rightsquigarrow$ take expectations over P (and thus $V_{1,2}$, $Z_{1,2}$).
  2. $\mathbb{E}\big[\mathcal{H}_{\ln}(V_1,H,V_2)\big] \;\approx\; \mathbb{E}\big[\mathcal{H}_{\ln}(D,1-D)\big] \;=\; H_{k+1} - H_{t+1}$ where $D \overset{\mathcal{D}}{=} \mathrm{Beta}(t+1,t+1)$
     but **not an inequality** in either direction

Entropy Bounds for Search Costs

$\rightsquigarrow$ $\alpha(\vec{q}) = c \cdot \mathcal{H}(\vec{q})$ does **not** seem to hold for any constant c

- But we can show

$$\begin{aligned} \alpha(\vec{q}) &\leqslant c \cdot \mathcal{H}(\vec{q}) + d \\ \alpha_{\vec{q}} &\geqslant c' \cdot \mathcal{H}(\vec{q}) - d' \end{aligned}$$

  for family of constants (c, d) and (c', d').

- Always have $c' < \alpha_k < c$ where

$$\alpha_k = \frac{\ln 2}{H_{k+1} - H_{t+1}}$$

Entropy Bounds for Search Costs

$\rightsquigarrow$ $\alpha(\vec{q}) = c \cdot \mathcal{H}(\vec{q})$ does **not** seem to hold for any constant c

- But we can show 

$$\begin{aligned} \alpha(\vec{q}) &\leqslant c \cdot \mathcal{H}(\vec{q}) + d \\ \alpha_{\vec{q}} &\geqslant c' \cdot \mathcal{H}(\vec{q}) - d' \end{aligned}$$

for family of constants (c, d) and (c', d').

- Always have $c' < \alpha_k < c$ where

$$\alpha_k = \frac{\ln 2}{H_{k+1} - H_{t+1}}$$

Entropy Bounds for Search Costs

$\rightsquigarrow$ $\alpha(\vec{q}) = c \cdot \mathcal{H}(\vec{q})$ does **not** seem to hold for any constant c

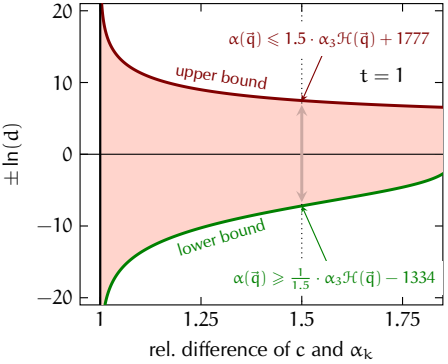
- But we can show 

$$\begin{aligned} \alpha(\vec{q}) &\leqslant c \cdot \mathcal{H}(\vec{q}) + d \\ \alpha_{\vec{q}} &\geqslant c' \cdot \mathcal{H}(\vec{q}) - d' \end{aligned}$$

for family of constants (c, d) and (c', d').

- Always have $c' < \alpha_k < c$ where

$$\alpha_k = \frac{\ln 2}{H_{k+1} - H_{t+1}}$$



Entropy Bounds for Search Costs

$\rightsquigarrow$ $\alpha(\vec{q}) = c \cdot \mathcal{H}(\vec{q})$ does **not** seem to hold for any constant c

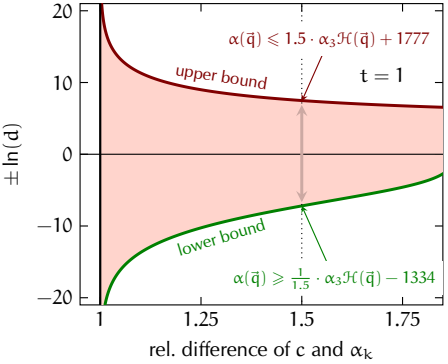
- But we can show 

$$\begin{aligned} \alpha(\vec{q}) &\leqslant c \cdot \mathcal{H}(\vec{q}) + d \\ \alpha_{\vec{q}} &\geqslant c' \cdot \mathcal{H}(\vec{q}) - d' \end{aligned}$$

for family of constants (c, d) and (c', d').

- Always have $c' < \alpha_k < c$ where

$$\alpha_k = \frac{\ln 2}{H_{k+1} - H_{t+1}}$$



Entropy Bounds for Search Costs

$\rightsquigarrow$ $\alpha(\vec{q}) = c \cdot \mathcal{H}(\vec{q})$ does **not** seem to hold for any constant c

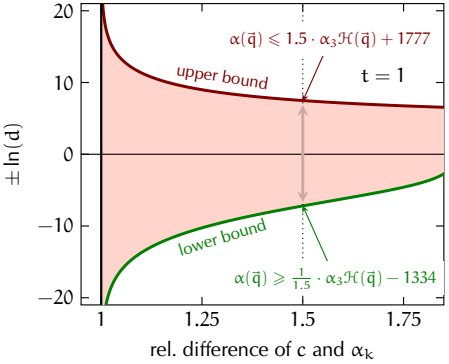
- But we can show 

$$\begin{aligned} \alpha(\vec{q}) &\leqslant c \cdot \mathcal{H}(\vec{q}) + d \\ \alpha_{\vec{q}} &\geqslant c' \cdot \mathcal{H}(\vec{q}) - d' \end{aligned}$$

for family of constants (c, d) and (c', d').

- Always have $c' < \alpha_k < c$ where

$$\alpha_k = \frac{\ln 2}{H_{k+1} - H_{t+1}}$$



$\rightsquigarrow$ **Asymptotically matching** values for c and c'

$\rightsquigarrow$ $$\alpha(\vec{q}) = \alpha_k \cdot \mathcal{H}_{ld}(\vec{q}) \pm O\Big(\mathcal{H}(\vec{q})^{\frac{t+2}{t+3}} \log\big(\mathcal{H}(\vec{q})\big)\Big)$$

Results in the i.i.d. Model



Time to put the pieces together!

Separation Theorem:
Quicksort costs

- in the **i.i.d. model**
- with **"many duplicates"**
  ($\Omega(n^\varepsilon)$ duplicates of each value in expectation)

are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

Average search costs
in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

- $\mathcal{H} = \mathcal{H}_{\mathrm{Id}}(\vec{q})$
- $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

$\rightsquigarrow$

Quicksort Costs (i.i.d. model)

Under the assumptions above, we have for any $\delta \in \big(0, \frac{1}{t+3}\big)$
$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha_k \mathcal{H}_{\mathrm{Id}}(\vec{q}) \cdot n \;\pm\; O\big((\mathcal{H}(\vec{q})^{1-\delta} + 1)n\big)$.

Results in the i.i.d. Model



Time to put the pieces together!

Separation Theorem:
Quicksort costs

- in the **i.i.d. model** 
- with **"many duplicates"**
  ($\Omega(n^{\varepsilon})$ duplicates of each value in expectation)

are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

Average search costs
in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

- $\mathcal{H} = \mathcal{H}_{\mathrm{ld}}(\vec{q})$
- $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

Quicksort Costs (i.i.d. model)

$\rightsquigarrow$ Under the assumptions above, we have for any $\delta \in \big(0, \frac{1}{t+3}\big)$
$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha_k \mathcal{H}_{\mathrm{ld}}(\vec{q}) \cdot n \;\pm\; O\big((\mathcal{H}(\vec{q})^{1-\delta} + 1)n\big)$.

Results in the i.i.d. Model



Time to put the pieces together!

Separation Theorem:
Quicksort costs

- in the **i.i.d. model**
- with **"many duplicates"**
  ($\Omega(n^\varepsilon)$ duplicates of each value in expectation)

are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

Average search costs
in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

- $\mathcal{H} = \mathcal{H}_{\mathsf{ld}}(\vec{q})$
- $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

Quicksort Costs (i.i.d. model)

$\rightsquigarrow$ Under the assumptions above, we have for any $\delta \in \big(0, \frac{1}{t+3}\big)$
$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha_k \mathcal{H}_{\mathsf{ld}}(\vec{q}) \cdot n \;\pm\; O\big((\mathcal{H}(\vec{q})^{1-\delta} + 1)n\big)$.

Results in the i.i.d. Model



Time to put the pieces together!

Separation Theorem:
Quicksort costs

- in the **i.i.d. model**
- with **"many duplicates"**
  ($\Omega(n^{\varepsilon})$ duplicates of each value in expectation)

are given by (as $n \to \infty$, for any $\delta \in (0, \varepsilon)$)

$$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha(\vec{q}) \cdot n \;\pm\; O(n^{1-\delta})$$

Average search costs
in saturated k-fringe-balanced trees:

(as $\mathcal{H} \to \infty$)

$$\alpha(\vec{q}) \;=\; \alpha_k \cdot \mathcal{H} \;\pm\; O\big(\mathcal{H}^{\frac{t+2}{t+3}} \log \mathcal{H}\big)$$

- $\mathcal{H} = \mathcal{H}_{\mathsf{ld}}(\vec{q})$
- $\alpha_k = \dfrac{\ln 2}{H_{k+1} - H_{t+1}}$

$\rightsquigarrow$

Quicksort Costs (i.i.d. model)

Under the assumptions above, we have for any $\delta \in \big(0, \frac{1}{t+3}\big)$
$\mathbb{E}[C_{n,\vec{q}}] \;=\; \alpha_k \mathcal{H}_{\mathsf{ld}}(\vec{q}) \cdot n \;\pm\; O\big((\mathcal{H}(\vec{q})^{1-\delta} + 1)n\big)$.

Outline

0 Intro

1 Quicksort and Search Trees

2 Saturated Fringe-Balanced Trees

3 Back to Multiset Permutations

Back to the Multiset Model

How about the multiset model?

Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

Back to the Multiset Model

How about the multiset model?

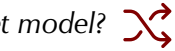


Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

Back to the Multiset Model

How about the multiset model? 

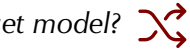


Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

Back to the Multiset Model

How about the multiset model? 

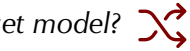


Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

Back to the Multiset Model

How about the multiset model? 



Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

Back to the Multiset Model

How about the multiset model? 



Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

The Annals of Statistics
1983, Vol. 11, No. 3, 896–904

THE EQUIVALENCE OF WEAK, STRONG AND COMPLETE CONVERGENCE IN L_1 FOR KERNEL DENSITY ESTIMATES[1]

BY LUC DEVROYE

McGill University

Let f be a density on R^d, and let f_n be the kernel estimate of f,

$$f_n(x) = (nh^d)^{-1} \sum_{i=1}^{n} K((x - X_i)/h)$$

where $h = h_n$ is a sequence of positive numbers, and K is an absolutely integrable function with $\int K(x)\,dx = 1$. Let $J_n = \int |f_n(x) - f(x)|\,dx$. We show

Back to the Multiset Model

How about the multiset model? 



Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

The Annals of Statistics

LEMMA 3. (*A multinomial distribution inequality*). *Let* $(X_1, \cdots, X_k)$ *be a multinomial* $(n, p_1, \cdots, p_k)$ *random vector. For all* $\varepsilon \in (0, 1)$ *and all* k *satisfying* $k/n \leq \varepsilon^2/20$, *we have*

$$P(\textstyle\sum_{i=1}^{k} |X_i - E(X_i)| > n\varepsilon) \leq 3 \exp(-n\varepsilon^2/25).$$

McGill University

Let f be a density on R^d, and let f_n be the kernel estimate of f,

$$f_n(x) = (nh^d)^{-1} \textstyle\sum_{i=1}^{n} K((x - X_i)/h)$$

where $h = h_n$ is a sequence of positive numbers, and K is an absolutely integrable function with $\int K(x)\,dx = 1$. Let $J_n = \int |f_n(x) - f(x)|\,dx$. We show

Back to the Multiset Model

How about the multiset model? 



Many duplicates $\rightsquigarrow$ profile $\vec{X}$ **concentrated** around $\mathbb{E}[\vec{X}] = \vec{q}n$

1. Replace multiset model with profile $\vec{x}$ by i.i.d. model with $\vec{q} = \vec{x}/n$
2. Use Chernoff bounds to **bound difference** between costs.

$\rightsquigarrow$ Need Chernoff bound for **multinomial variables**.

The Annals of Statistics

LEMMA 3. *(A multinomial distribution inequality). Let $(X_1, \cdots, X_k)$ be a multinomial $(n, p_1, \cdots, p_k)$ random vector. For all $\varepsilon \in (0, 1)$ and all k satisfying $k/n \leq \varepsilon^2/20$, we have*

$$P(\textstyle\sum_{i=1}^{k} |X_i - E(X_i)| > n\varepsilon) \leq 3 \exp(-n\varepsilon^2/25).$$

McGill University

$\rightsquigarrow$ **Same result** holds for **multiset model** with **"many duplicates"**, i.e., $x_\nu = \Omega(n^\varepsilon)$.

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** ... for **"many duplicates"**.
  - $\rightsquigarrow$ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning

Open Problems

- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - insertsorting costs are still negligible
  - difference between U.D. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** ... for **"many duplicates"**.
  - $\rightsquigarrow$ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \ldots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning

Open Problems

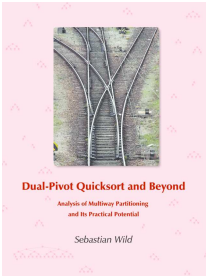
- Get rid of "many duplicates" restriction: $n^{1-\varepsilon}$ seems (to me) best possible so that
  - Inputs are non-degenerate, $n \gg u$
  - Insertion sort and pivot sorting costs are still negligible
  - Difference between 1 and n is negligible
  - Uniform distribution is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** ... for **"many duplicates"**.
  - $\rightsquigarrow$ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

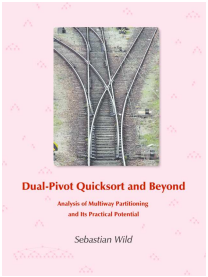
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - kwaySolidify pivots are still reachable
  - difference between UAR model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** ... for **"many duplicates"**.
  - $\rightsquigarrow$ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

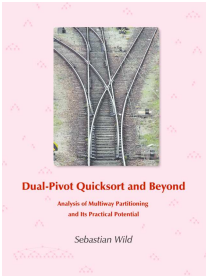
- Get rid of "many duplicates" restriction: $n^{1-\varepsilon}$ seems (to me) best possible so that
  - Inputs are non-degenerate, $n \geq 2$
  - Inputs are non-degenerate, $n \geq 2$
  - Inputs are non-degenerate, $n \geq 2$

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** ... for **"many duplicates"**.
  - $\rightsquigarrow$ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \ldots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

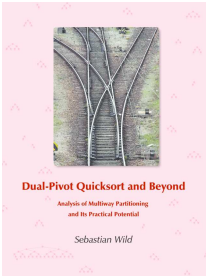
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - insertsorting rests are still negligible
  - difference between UAR model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

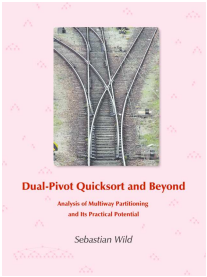
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - tree-building costs are still negligible
  - difference between i.i.d. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

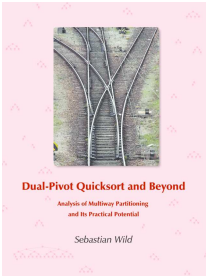
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - tree-building costs are still negligible
  - difference between i.i.d. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

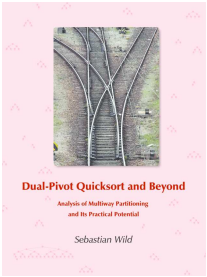
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - tree-building costs are still negligible
  - difference between i.i.d. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-**k **Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k *Quicksort approaches lower bound for* $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

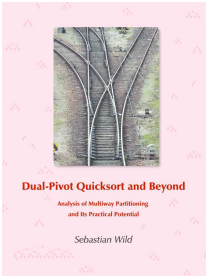
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w. h. p.
  - tree-building costs are still negligible
  - difference between i. i. d. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

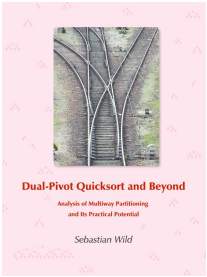
- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - tree-building costs are still negligible
  - difference between i.i.d. model and multiset model is negligible
  - the entropy is a lower bound

Conclusion

Findings

- First analysis of **median-of-k Quicksort on equal keys** … for **"many duplicates"**.
  - ⇝ Same relative speedup as for random permutations.
- Partial Answer to **conjecture of Sedgewick & Bentley**:

  Median-of-k Quicksort approaches lower bound for $k \to \infty$.
- *Not in this talk:* For uniform $\vec{q} = (\frac{1}{u}, \dots, \frac{1}{u})$ with $u = O(n^{1-\varepsilon})$
  - better error bounds
  - extension for multiway partitioning



Open Problems

- Get rid of "many duplicates" restriction; $n^{1-\varepsilon}$ seems (to me) best possible so that
  - inputs are non-degenerate w.h.p.
  - tree-building costs are still negligible
  - difference between i.i.d. model and multiset model is negligible
  - the entropy is a lower bound


RS Royal Sovereign
QUICK-SORT ™
COIN SORTER Model No CO-1000
QUARTERS
$10
40EA
NICKELS
$2
PENNIES
$.50
50EA
DIMES
$5
50EA

Icons made by *Freepik* and *Gregor Cresnar* from `www.flaticon.com`.