Automated Data Structure Generation: Refuting Common Wisdom

Kyle Dewey, Lawton Nichols, Ben Hardekopf

University of California, Santa Barbara

Teaser

From the standpoint of automatically generating intricate, highly constrained data structures:

- Common wisdom: imperative techniques are fast but inexpressive, while declarative techniques slow but easy to work with
- In contrast, we find that *declarative techniques are uniformly lightning fast* (~30x to 9,000,000x)
- However, for *previously unattempted complex data structures*, *declarative techniques lack usability*

Outline

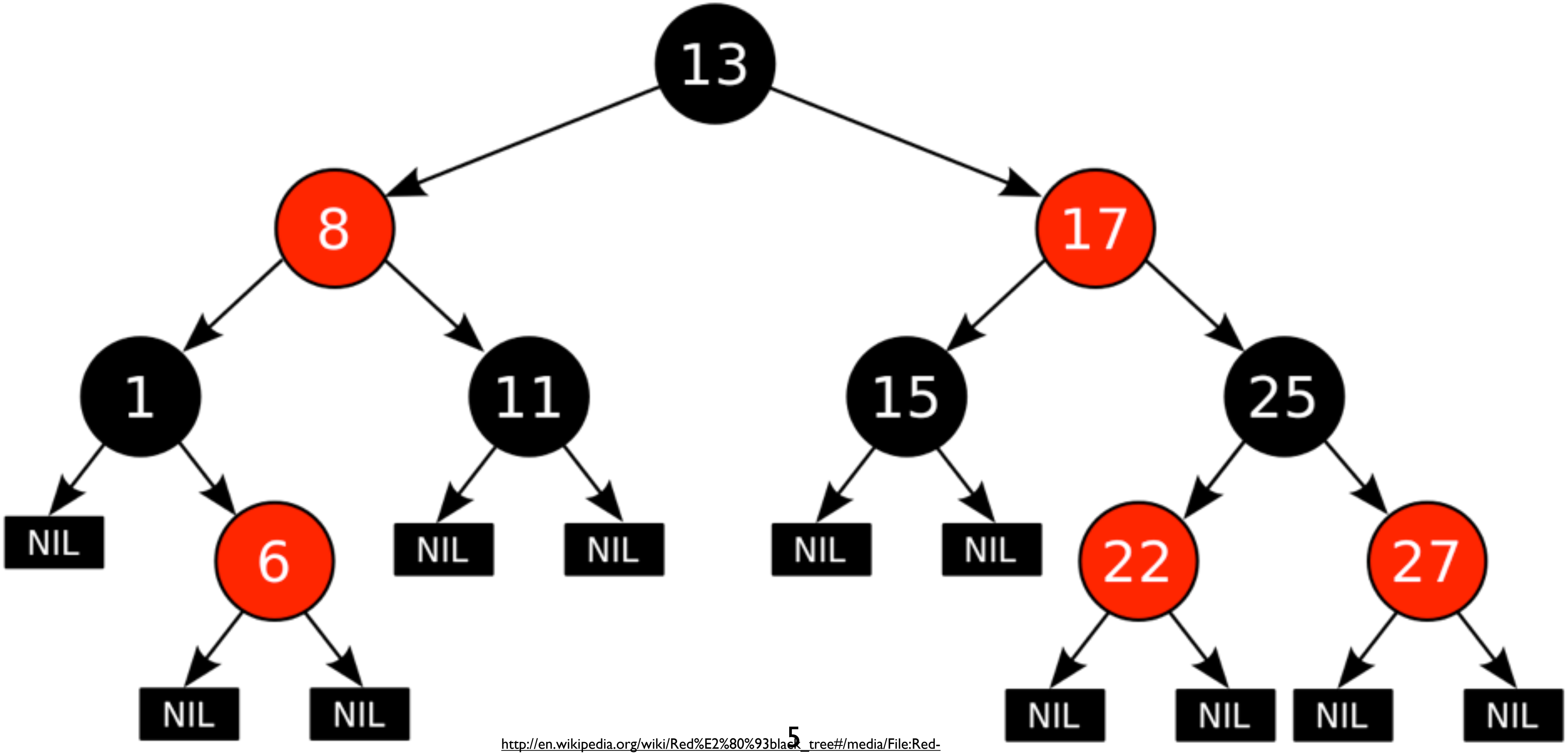
- Background
- Simple example
- Usability problems
- Performance evaluation

Outline

- Background
- Simple example
- Usability problems
- Performance evaluation

Basic Problem

We want to develop black-box *generators* for complex, constrained data structures, in order to enable automated testing of code that operates on these data structures



http://en.wikipedia.org/wiki/Red%E2%80%93black_tree#/media/File:Red-black_tree_example.svg

Specifying Data Structure Generators

Two general approaches: *imperative* and *declarative*

- *Imperative* approaches feature loops and assignment, and are focused on *how* to generate
- *Declarative* approaches lack imperative features, and allow for logical descriptions of high-level features focused on *what* to generate

Common Wisdom

- Imperative techniques are fast, but potentially unwieldy
- Declarative techniques are slow, but easier to use

Our Observation: There are Hidden Caveats to This Common Wisdom

Performance Caveat

Imperative means fast?

- One 13 year old result
- Compares a SAT-based approach to a non-SAT-based approach
  - SAT is not the only way to write declarative code

Usability Caveat

Declarative means expressive?

- Most complex data structure ever generated: valid red/black trees
  - These are not actually all that complicated
  - Nothing considers operations on the data structures

Our Contribution

- Test using a declarative approach that *is not* SAT-based
- Test with more complex data structures, along with *special variants* of them
    - E.g., red/black trees which will rebalance upon the insertion of some value `k`

Declarative Without SAT

- Our observation: related work has been incrementally moving towards implementing a constraint logic programming (CLP) engine
- We will use CLP directly as our declarative stand-in
    - Re-use decades of existing work

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees Covered in related work
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Novel to this work

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Data Structures

- Sorted linked lists
- Red-black trees
- Array heaps
- ANI images (via grammars)
- Skip lists
- Splay trees
- B-trees

Special Variants

- For each of these data structures, we also defined a special variant of them which tends to indicate a more interesting version for testing purposes
- Tried to select variants that stressed data structure specific operations
- More details in the paper

Special Variants with an Operational Nature

- Red-black trees: need insertion and rebalancing
- Array heaps: need dequeueing
- Splay trees: need splay
- B-trees: need insertion and node splitting

We are the first to look at these operations in the context of generation.

Outline

- Background
- Simple example
- Usability problems
- Performance evaluation

Example: Sorted Linked Lists

Sorted Linked Lists

- Each element is between 0 and `K`
- A list contains between 0 and `N` elements
- Each element is ≤ the element after it, if applicable
  - I.e., the list is in ascending order

“Each element is between 0 and `K`”

“Each element is between 0 and `K`”

```
inBound(K, Element)
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
  0 #=< Element
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
  0 #=< Element,
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
   0 #=< Element,
   Element #=< K
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
   0 #=< Element,
   Element #=< K.
```

“Each element is between 0 and `K`”

```
inBound(K, Element) :-
   0 #=< Element,
   Element #=< K.
```

Sorted Linked Lists

- Each element is between 0 and `K`
- A list contains between 0 and `N` elements
- Each element is ≤ the element after it, if applicable
    - I.e., the list is in ascending order

Sorted Linked Lists

- Each element is between 0 and `K`
- A list contains between 0 and `N` elements
- Each element is ≤ the element after it, if applicable
  - I.e., the list is in ascending order

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
  N > 1
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
  N > 1,
  Elm1 #=< Elm2
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
  N > 1,
  Elm1 #=< Elm2,
  inBound(K, Elm1),
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
  N > 1,
  Elm1 #=< Elm2,
  inBound(K, Elm1),
  NewN is N - 1,
```

“A list contains between 0 and `N` elements in ascending order, all between 0 and `K`”

```
% sorted: (N, K, List)
sorted(_, _, []).
sorted(N, K, [Element]) :-
  N > 0,
  inBound(K, Element).
sorted(N, K, [Elm1, Elm2|Rest]) :-
  N > 1,
  Elm1 #=< Elm2,
  inBound(K, Elm1),
  NewN is N - 1,
  sorted(NewN, K, [Elm2|Rest]).
```

Putting it All Together

```
% sorted: (N, K, List)
%
% Query below:
?- sorted(3, 4, List), label(List).
```

Putting it All Together

```
% sorted: (N, K, List)
%
% Query below:
?- sorted(3, 4, List), label(List).
```

Putting it All Together

```
% sorted: (N, K, List)
%
% Query below:
?- sorted(3, 4, List), label(List).
```

```
List = [] ;
List = [0] ;
...
List = [1, 3, 3] ;
...
List = [2, 2, 4] ;
...
```

Outline

- Background
- Simple example
- Usability problems
- Performance evaluation

Fundamental Problem: Not Everything is as Simple as a Sorted List

B-Tree Invariants

- Include:
    - Every node has at most `m` children
    - All leaves appear in the same level
- Decidedly logical in nature
- Easy to express declaratively

An Operational Twist

- The invariants before define what a B-tree is
- What if we are interested in testing *operations* on B-trees, specifically with trees *intentionally designed* to stress corner cases?
  - Under *specific conditions*, tree structure must radically change upon *element insertion*
- Requires us to explain operations to the generator

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while i ≥ 1 && k < $key_i[x]$
            do $key_{i+1}[x]$ <- $key_i[x]$
               i <- i - 1
        $key_{i+1}[x]$ <- k
        n[x] <- n[x] + 1
        DISK-WRITE(x)
    else while i ≥ 1 && k < $key_i[x]$
            do i <- i - 1
        i <- i + 1
        DISK-READ($c_i[x]$)
        if n[$c_i[x]$] = 2t - 1
          then B-TREE-SPLIT-CHILD(...)
...
```

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while $i \geq 1$ && $k < key_i[x]$
          do $key_{i+1}[x]$ <- $key_i[x]$
             i <- i - 1
     $key_{i+1}[x]$ <- k
     n[x] <- n[x] + 1
     DISK-WRITE(x)
  else while $i \geq 1$ && $k < key_i[x]$
          do i <- i - 1
     i <- i + 1
     DISK-READ($c_i[x]$)
     if $n[c_i[x]] = 2t - 1$
        then B-TREE-SPLIT-CHILD(...)
...
```

How to implement?

Imperative Setting: Implement Directly

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while i ≥ 1 && k < $key_i[x]$
             do $key_{i+1}[x]$ <- $key_i[x]$
                i <- i - 1
         $key_{i+1}[x]$ <- k
         n[x] <- n[x] + 1
         DISK-WRITE(x)
   else while i ≥ 1 && k < $key_i[x]$
              do i <- i - 1
         i <- i + 1
         DISK-READ($c_i[x]$)
         if n[$c_i[x]$] = 2t - 1
            then B-TREE-SPLIT-CHILD(...)
...
```

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while i ≥ 1 && k < $key_i[x]$
           do $key_{i+1}[x]$ <- $key_i[x]$
              i <- i - 1
...
```

Imperative Specification

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

Actual Imperative Implementation Code (Korat)

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while i ≥ 1 && k < $key_i[x]$
          do $key_{i+1}[x]$ <- $key_i[x]$
             i <- i - 1
...
```

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while $i \geq 1$ && $k < key_i[x]$
          do $key_{i+1}[x]$ <- $key_i[x]$
             i <- i - 1
...
```

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while $i \geq 1$ && $k < key_i[x]$
            do $key_{i+1}[x]$ <- $key_i[x]$
               i <- i - 1
...
```

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

```
B-TREE-INSERT-NONFULL(x, k)
    i <- n[x]
    if leaf[x]
    then while $i \ge 1$ && k < $key_i[x]$
              do $key_{i+1}[x]$ <- $key_i[x]$
                 i <- i - 1
...
```

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

```
B-TREE-INSERT-NONFULL(x, k)
i <- n[x]
if leaf[x]
then while $i \geq 1$ && $k < key_i[x]$
               do $key_{i+1}[x]$ <- $key_i[x]$
                  i <- i - 1
...
```

```
void insertNonFull(Node x, int k) {
  int i = x.n;
  if (x.leaf) {
    while (i >= 1 && k < x.key[i]) {
      x.key[i + 1] = x.key[i];
      i = i - 1;
    }
....
```

Declarative Setting: Logical Implication

```
B-TREE-INSERT-NONFULL(x, k) => ???
i <- n[x]
if leaf[x]
then while $i \geq 1$ && $k < key_i[x]$
            do $key_{i+1}[x]$ <- $key_i[x]$
               i <- i - 1
     $key_{i+1}[x]$ <- k
     n[x] <- n[x] + 1
     DISK-WRITE(x)
  else while $i \geq 1$ && $k < key_i[x]$
            do i <- i - 1
     i <- i + 1
     DISK-READ($c_i[x]$)
     if $n[c_i[x]] = 2t - 1$
       then B-TREE-SPLIT-CHILD(...)
...
```

```
B-TREE-INSERT-NONFULL(x, k)
    i <- n[x]
    if leaf[x]
    then while $i \geq 1$ && $k < key_i[x]$
             do $key_{i+1}[x]$ <- $key_i[x]$
                i <- i - 1
         $key_{i+1}[x]$ <- k
         n[x] <- n[x] + 1
         DISK-WRITE(x)
      else while $i \geq 1$ && $k < key_i[x]$
               do i <- i - 1
         i <- i + 1
         DISK-READ($c_i[x]$)
         if $n[c_i[x]] = 2t - 1$
           then B-TREE-SPLIT-CHILD(...)
...
```

```
B-TREE-INSERT-NONFULL(x, k)
i $\boxed{\text{<-}}$ n[x]
if leaf[x]
then while $i \ge 1$ && $k < key_i[x]$
        do $key_{i+1}[x]$ $\boxed{\text{<-}}$ $key_i[x]$
           i $\boxed{\text{<-}}$ i - 1
     $key_{i+1}[x]$ $\boxed{\text{<-}}$ k
     n[x] $\boxed{\text{<-}}$ n[x] + 1
     $\boxed{\text{DISK-WRITE(x)}}$
  else while $i \ge 1$ && $k < key_i[x]$
          do i $\boxed{\text{<-}}$ i - 1
     i $\boxed{\text{<-}}$ i + 1
     $\boxed{\text{DISK-READ}(c_i[x])}$
     if $n[c_i[x]] = 2t - 1$
       then B-TREE-SPLIT-CHILD(...)
...
```

Our Observation: Imperative Features are Desirable for Modeling Operations on Data Structures

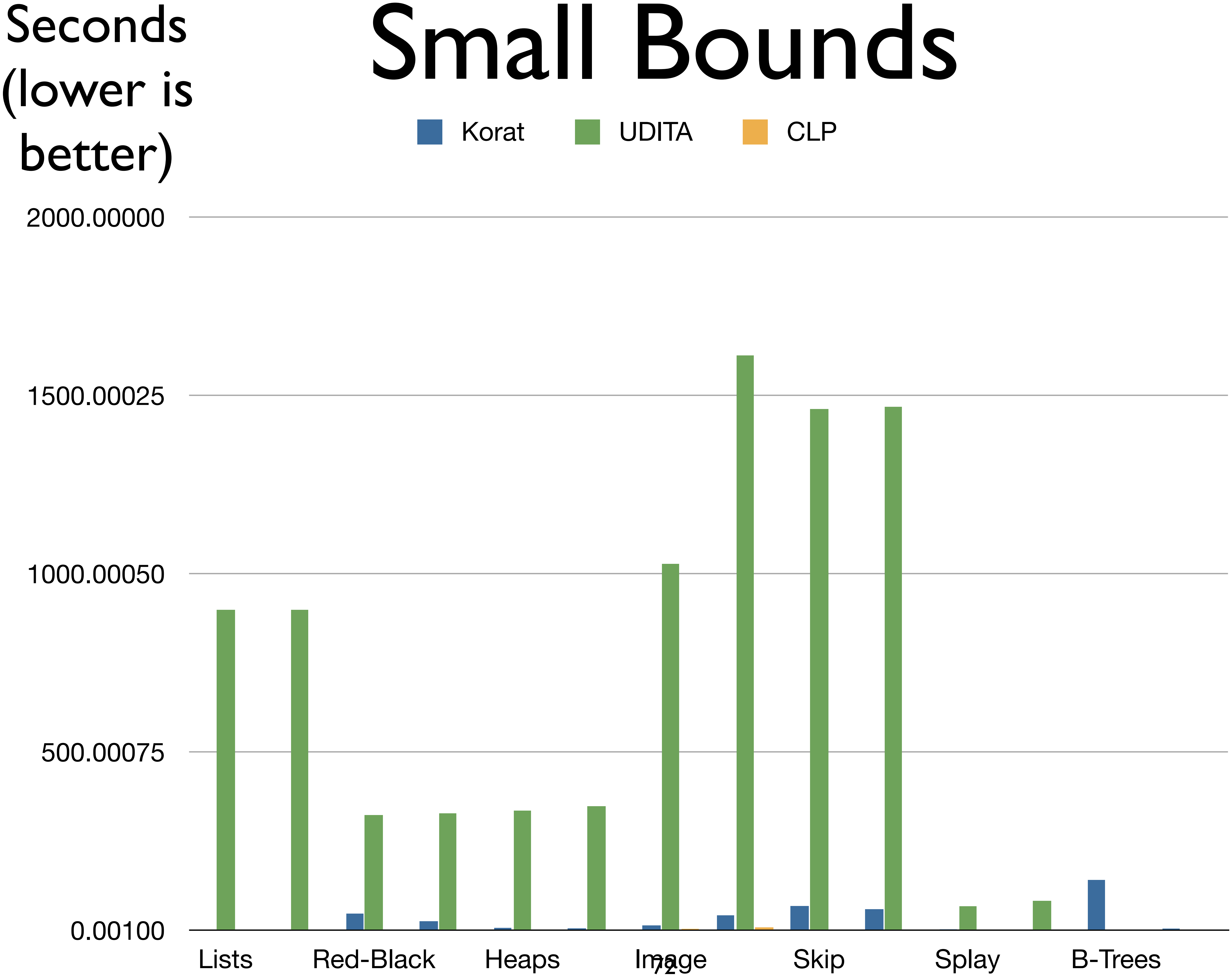
Outline

- Background
- Simple example
- Usability problems
- Performance evaluation

Measuring Performance

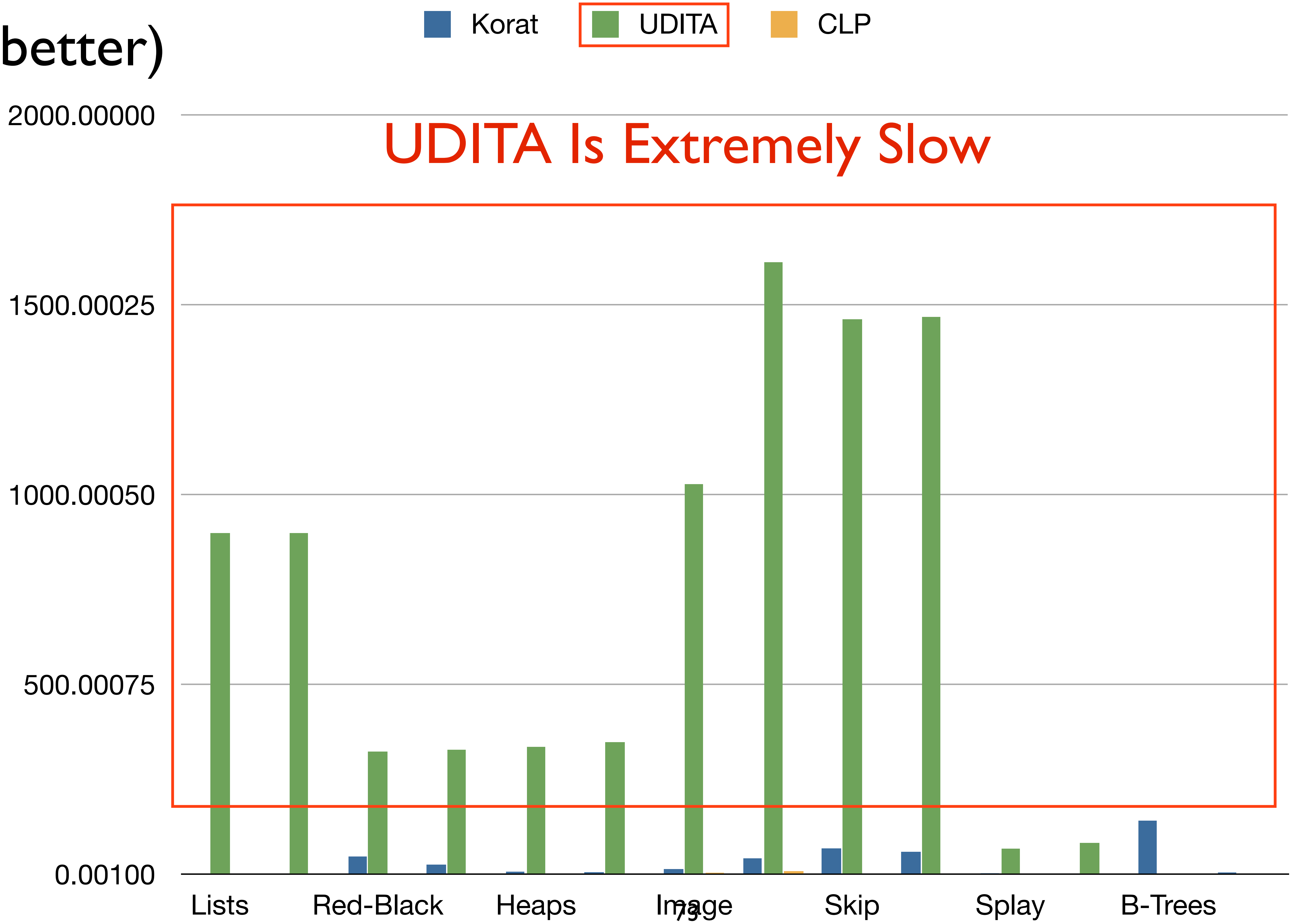
- Tested all aforementioned data structures and their special variants on Korat, UDITA, and CLP (using GNU Prolog)
- Measured how quickly all data structures within certain bounds could be generated, with a 30 minute timeout

Small Bounds



Small Bounds

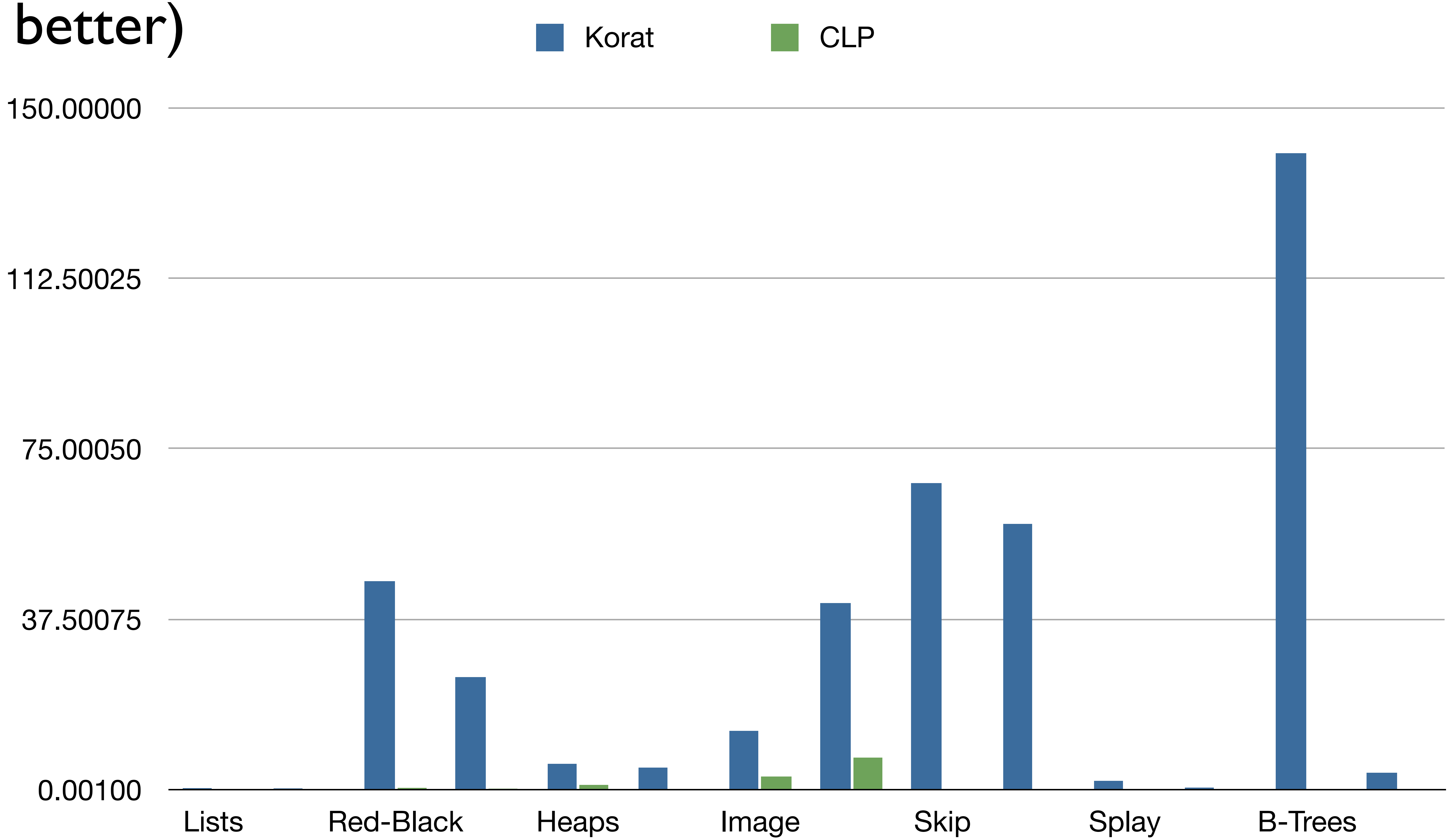
Seconds (lower is better)



UDITA Is Extremely Slow

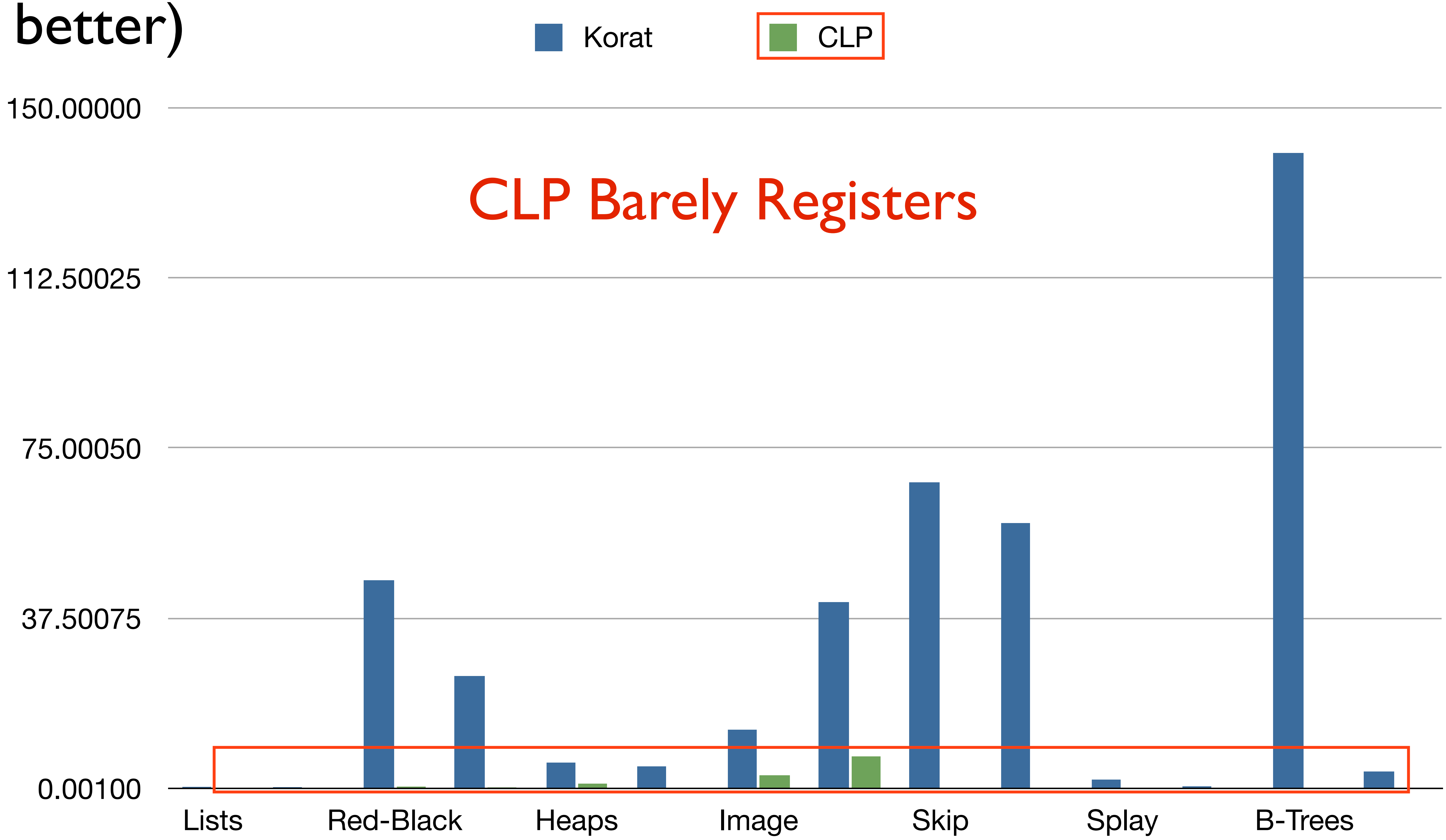
Seconds (lower is better)

Small Bounds



Seconds (lower is better)

Small Bounds



CLP Barely Registers

Medium Bounds

- UDITA times out on everything
- Korat times out on 5 / 14 experiments
- CLP is generally ~30x - 1,000x faster
- For B-trees, Korat and UDITA both timeout, but CLP completes within **a single millisecond**

Large Bounds

- Korat and UDITA timeout on everything
- Depending on the data structure, CLP takes between ~70 seconds and just under 30 minutes

On Usability

- Informal argument
- No data structure took more than 90 minutes to specify in Korat or UDITA
  - Code and algorithm reuse
- CLP variants *always* took *significantly* longer, up to 10 hours for B-trees
  - Existing code all imperative, with little explanation of why it works

Conclusions

- CLP, a declarative technique, *dramatically outperforms* the imperative Korat and UDITA, *defying common wisdom*
- Korat and UDITA allow for much easier modeling than CLP, *entirely because* they are imperative in nature