Branchless Search Programs: Electronic Appendix

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 1, 2100 Copenhagen East, Denmark

Abstract. This report is an electronic appendix to the paper "Branchless Search Programs" which has been sent for publication. This report together with an accompanying `tar` ball gives the source code used in the experiments discussed in that paper.

Keywords. Data structures, balanced search trees, implicit search trees, branch mispredictions, cache misses

CPH STL Report 2012-1, August 2012. Revised January 2013.

Copyright notice

Release date

2013-01-31

Included files

Models

§ 1 *two_way_model.h++*

```
#define ALPHA 2

namespace two_way_model {

  template <typename natural>
  bool search(natural n, natural s) {
    double alpha = 1.0 / double(ALPHA);
    while (n > 1) {
      natural left_border = natural(alpha * double(n));
      if (s < left_border) {
        n = left_border;
      }
      else {
        n = n - 1 - left_border;
        s = s - left_border;
      }
    }
    return n <= 1;
  }
}
```

§ 2 *three_way_model.h++*

```
#define ALPHA 2

namespace three_way_model {

  template <typename natural>
  bool search(natural n, natural s) {
    double alpha = 1.0 / double(ALPHA);
    while (n > 1) {
      natural left_border = natural(alpha * double(n));
      if (s < left_border) {
        n = left_border;
      }
      else {
        if (s > left_border) {
          n = n - 1 - left_border;
          s = s - 1 - left_border;
        }
        else {
          return true;
        }
      }
    }
    return true;
  }
}
```

§ 3 *branchless_model.h++*

```
#define ALPHA 2

namespace branchless_model {

  template <typename natural>
  bool search(natural n, natural s) {
    double alpha = 1.0 / double(ALPHA);
    while (n > 1) {
```

```
9       natural left_border = natural(alpha * double(n));
10      bool smaller = (s < left_border);
11      s = s - left_border + smaller * left_border;
12      n = smaller * left_border + (1 - smaller) * (n - 1 - left_border);
13    }
14    return n <= 1;
15  }
16 }
```

Search-tree nodes

§ 4 *binary-search-tree-node.h++*

```
1 /*
2   A node used by a binary search tree
3
4   Author: Jyrki Katajainen © 2012
5 */
6
7 namespace cphstl {
8
9   template <typename V>
10  class binary_search_tree_node {
11  public:
12
13    // types
14
15    typedef V value_type;
16
17  protected:
18
19    // variables
20
21    binary_search_tree_node* left_child;
22    binary_search_tree_node* right_child;
23    binary_search_tree_node* parent_;
24    V element_;
25
26  public:
27
28    // structors
29
30    binary_search_tree_node()
31      : left_child(nullptr), right_child(nullptr), parent_(nullptr) {
32    }
33
34    binary_search_tree_node(V const & v)
35      : left_child(nullptr), right_child(nullptr), parent_(nullptr), element_(v)
            {
36    }
37
38    ~binary_search_tree_node() {
39    }
40
41    // accesors
42
43    binary_search_tree_node* left() const {
44      return left_child;
45    }
46
47    binary_search_tree_node* right() const {
48      return right_child;
49    }
50
```

```
51      binary_search_tree_node* parent() const {
52        return parent_;
53      }
54
55      V const & element() const {
56        return element_;
57      }
58
59      bool is_root() const {
60        return parent_ == nullptr;
61      }
62
63      // modifiers
64
65      void left(binary_search_tree_node* p) {
66        left_child = p;
67      }
68
69      void right(binary_search_tree_node* p) {
70        right_child = p;
71      }
72
73      void parent(binary_search_tree_node* p) {
74        parent_ = p;
75      }
76
77      V & element() {
78        return element_;
79      }
80    };
81  }
```

Skewed search trees

§ 5 *skewed_search_tree.h++*

```
1  /*
2    An implementation of a skewed binary search tree
3    - memory layout is random
4
5    Author: Jyrki Katajainen © 2012
6  */
7
8  #define ALPHA 2
9
10 #include <algorithm>  // std::max
11 #include "binary-search-tree-node.h++"
12 #include <cstddef> // std::size_t
13 #include <functional> // std::less
14 #include <memory> // std::allocator
15 #include <vector>
16
17 namespace cphstl {
18
19   template <
20     typename V,
21     typename C = std::less<V>,
22     typename N = cphstl::binary_search_tree_node<V>,
23     typename A = std::allocator<N>
24   >
25   class skewed_search_tree {
26   public:
27
28     // types
```

```
29
30      typedef V value_type;
31      typedef C comparator_type;
32      typedef N node_type;
33      typedef A allocator_type;
34      typedef std::size_t size_type;
35
36    protected:
37
38      // variables
39
40      C less;
41      A allocator_;
42      N* root;
43      size_type n;
44      N* pool;
45      size_type h;
46
47    public:
48
49      // structors
50
51      template <typename I>
52      explicit skewed_search_tree(I from, I to, C const& c, A const& a = A())
53        : less(c), allocator_(a), root(nullptr), n(0) {
54        n = to - from;
55        pool = allocator_.allocate(n);
56        N* p = pool;
57        std::vector<N*, A> free;
58        for (size_type i = 0; i != n; ++i) {
59          free.push_back(p);
60          ++p;
61        }
62        root = construct(from, to, free);
63        (*root).parent(nullptr);
64        h = height(root);
65      }
66
67      ~skewed_search_tree() {
68        allocator_.deallocate(pool, n);
69        root = nullptr;
70        n = 0;
71      }
72
73      // accessors
74
75      C comparator() const {
76        return less;
77      }
78
79      A allocator() const {
80        return allocator_;
81      }
82
83      size_type size() const {
84        return n;
85      }
86
87  #if defined(BRANCHLESS) || defined(UNROLLED)
88
89    private:
90
91      N* choose(bool condition, N* x, N* y) {
92        return (N*)((char*) y + condition * ((char*) x - (char*) y));
93      }
```

```
94
95 #endif
96
97   public:
98
99 #if defined(UNROLLED)
100
101 /*
102   Warning: Works only for ALPHA = 2!
103 */
104
105     bool is_member(V const & e) {
106       N* y = nullptr;
107       N* x = root;
108       bool smaller;
109       switch (h) {
110       case 31:
111         smaller = less(e, (*x).element());
112         y = choose(smaller, y, x);
113         x = choose(smaller, (*x).left(), (*x).right());
114       case 30:
115         smaller = less(e, (*x).element());
116         y = choose(smaller, y, x);
117         x = choose(smaller, (*x).left(), (*x).right());
118       case 29:
119         smaller = less(e, (*x).element());
120         y = choose(smaller, y, x);
121         x = choose(smaller, (*x).left(), (*x).right());
122       case 28:
123         smaller = less(e, (*x).element());
124         y = choose(smaller, y, x);
125         x = choose(smaller, (*x).left(), (*x).right());
126       case 27:
127         smaller = less(e, (*x).element());
128         y = choose(smaller, y, x);
129         x = choose(smaller, (*x).left(), (*x).right());
130       case 26:
131         smaller = less(e, (*x).element());
132         y = choose(smaller, y, x);
133         x = choose(smaller, (*x).left(), (*x).right());
134       case 25:
135         smaller = less(e, (*x).element());
136         y = choose(smaller, y, x);
137         x = choose(smaller, (*x).left(), (*x).right());
138       case 24:
139         smaller = less(e, (*x).element());
140         y = choose(smaller, y, x);
141         x = choose(smaller, (*x).left(), (*x).right());
142       case 23:
143         smaller = less(e, (*x).element());
144         y = choose(smaller, y, x);
145         x = choose(smaller, (*x).left(), (*x).right());
146       case 22:
147         smaller = less(e, (*x).element());
148         y = choose(smaller, y, x);
149         x = choose(smaller, (*x).left(), (*x).right());
150       case 21:
151         smaller = less(e, (*x).element());
152         y = choose(smaller, y, x);
153         x = choose(smaller, (*x).left(), (*x).right());
154       case 20:
155         smaller = less(e, (*x).element());
156         y = choose(smaller, y, x);
157         x = choose(smaller, (*x).left(), (*x).right());
158       case 19:
```

```
159         smaller = less(e, (*x).element());
160         y = choose(smaller, y, x);
161         x = choose(smaller, (*x).left(), (*x).right());
162       case 18:
163         smaller = less(e, (*x).element());
164         y = choose(smaller, y, x);
165         x = choose(smaller, (*x).left(), (*x).right());
166       case 17:
167         smaller = less(e, (*x).element());
168         y = choose(smaller, y, x);
169         x = choose(smaller, (*x).left(), (*x).right());
170       case 16:
171         smaller = less(e, (*x).element());
172         y = choose(smaller, y, x);
173         x = choose(smaller, (*x).left(), (*x).right());
174       case 15:
175         smaller = less(e, (*x).element());
176         y = choose(smaller, y, x);
177         x = choose(smaller, (*x).left(), (*x).right());
178       case 14:
179         smaller = less(e, (*x).element());
180         y = choose(smaller, y, x);
181         x = choose(smaller, (*x).left(), (*x).right());
182       case 13:
183         smaller = less(e, (*x).element());
184         y = choose(smaller, y, x);
185         x = choose(smaller, (*x).left(), (*x).right());
186       case 12:
187         smaller = less(e, (*x).element());
188         y = choose(smaller, y, x);
189         x = choose(smaller, (*x).left(), (*x).right());
190       case 11:
191         smaller = less(e, (*x).element());
192         y = choose(smaller, y, x);
193         x = choose(smaller, (*x).left(), (*x).right());
194       case 10:
195         smaller = less(e, (*x).element());
196         y = choose(smaller, y, x);
197         x = choose(smaller, (*x).left(), (*x).right());
198       case 9:
199         smaller = less(e, (*x).element());
200         y = choose(smaller, y, x);
201         x = choose(smaller, (*x).left(), (*x).right());
202       case 8:
203         smaller = less(e, (*x).element());
204         y = choose(smaller, y, x);
205         x = choose(smaller, (*x).left(), (*x).right());
206       case 7:
207         smaller = less(e, (*x).element());
208         y = choose(smaller, y, x);
209         x = choose(smaller, (*x).left(), (*x).right());
210       case 6:
211         smaller = less(e, (*x).element());
212         y = choose(smaller, y, x);
213         x = choose(smaller, (*x).left(), (*x).right());
214       case 5:
215         smaller = less(e, (*x).element());
216         y = choose(smaller, y, x);
217         x = choose(smaller, (*x).left(), (*x).right());
218       case 4:
219         smaller = less(e, (*x).element());
220         y = choose(smaller, y, x);
221         x = choose(smaller, (*x).left(), (*x).right());
222       case 3:
223         smaller = less(e, (*x).element());
```

```
224         y = choose(smaller, y, x);
225         x = choose(smaller, (*x).left(), (*x).right());
226       case 2:
227         smaller = less(e, (*x).element());
228         y = choose(smaller, y, x);
229         x = choose(smaller, (*x).left(), (*x).right());
230       case 1:
231         smaller = less(e, (*x).element());
232         y = choose(smaller, y, x);
233         x = choose(smaller, (*x).left(), (*x).right());
234       default:
235         smaller = (x == nullptr) || less(e, (*x).element());
236         y = choose(smaller, y, x);
237       }
238       if ((y == nullptr) || less((*y).element(), e)) {
239         return false;
240       }
241       return true;
242     }
243
244 #elif defined(BRANCHLESS)
245
246     bool is_member(V const& e) {
247       N* y = nullptr; // candidate node
248       N* x = root; // current node
249       while (x != nullptr) {
250         bool smaller = less(e, (*x).element());
251         y = choose(smaller, y, x);
252         x = choose(smaller, (*x).left(), (*x).right());
253       }
254       if (y == nullptr || less((*y).element(), e)) {
255         return false;
256       }
257       return true;
258     }
259
260 #else
261
262     bool is_member(V const& e) {
263       N* y = nullptr; // candidate node
264       N* x = root; // current node
265       while (x != nullptr) {
266         if (less(e, (*x).element())) {
267           x = (*x).left();
268         }
269         else {
270           y = x;
271           x = (*x).right();
272         }
273       }
274       if (y == nullptr || less((*y).element(), e)) {
275         return false;
276       }
277       return true;
278     }
279
280 #endif
281
282   private:
283     int height(N* t) const {
284       if (t == nullptr) {
285         return -1;
286       }
287       return 1 + std::max(height((*t).left()), height((*t).right()));
288     }
```

```
289
290     template <typename I, typename F>
291     N* construct(I p, I r, F& free) {
292       size_type n = r - p;
293       if (n == 0) {
294         return nullptr;
295       }
296       double alpha = 1.0 / double(ALPHA);
297       size_type border = size_type(alpha * double(n));
298       I cut_point = p + border;
299       size_type i = std::rand() % free.size();
300       N* place = free[i];
301       N* last = free.back();
302       free[i] = last;
303       free.pop_back();
304       (*place).element() = *cut_point;
305       N* child = construct(p, cut_point, free);
306       if (child != nullptr) {
307         (*child).parent(place);
308       }
309       (*place).left(child);
310       child = construct(cut_point + 1, r, free);
311       if (child != nullptr) {
312         (*child).parent(place);
313       }
314       (*place).right(child);
315       return place;
316     }
317   };
318 }
```

§ 6 *branchless_skewed_search_tree.h++*

```
1 /*
2   An implementation of a skewed binary search tree;
3   − memory layout is random
4   − search procedure is branchless
5
6   Author: Jyrki Katajainen © 2012, 2013
7 */
8
9 #define skewed_search_tree branchless_skewed_search_tree
10 #define BRANCHLESS
11
12 #include "skewed_search_tree.h++"
```

§ 7 *unrolled_perfectly_balanced_search_tree.h++*

```
1 /*
2   An implementation of a perfectly balanced search tree;
3   − memory layout is random
4   − search procedure is branchless and unrolled
5
6   Author: Jyrki Katajainen © 2012, 2013
7 */
8
9 #define skewed_search_tree unrolled_perfectly_balanced_search_tree
10 #define UNROLLED
11 #define ALPHA 2
12
13 #include "skewed_search_tree.h++"
```

Local search trees

§ 8 *local_search_tree.h++*

```
/*
  An implementation of a local search tree

  Warning: The formulas overflow if n is close to 2^{32}. For example,
  for n = 2^{25}, the largest f that could be used was 2^{6}.

  Author: Jyrki Katajainen © 2012
*/

#define ARITY 16

#include <algorithm>  // std::max
#include "binary_search_tree_node.h++"
#include <cstddef> // std::size_t
#include <functional> // std::less
#include <memory> // std::allocator

namespace cphstl {

  template <
    typename V,
    typename C = std::less<V>,
    typename N = cphstl::binary_search_tree_node<V>,
    typename A = std::allocator<N>
  >
  class local_search_tree {
  public:

    // types

    typedef V value_type;
    typedef C comparator_type;
    typedef N node_type;
    typedef A allocator_type;
    typedef std::size_t size_type;

  protected:

    // variables

    C less;
    A allocator_;
    N* root;
    size_type n;
    N* pool;
    size_type f;
    size_type h;

  private:

    // helpers

    size_type parent(size_type i) const {
      size_type j = i / f;
      if (i != j * f) {
        return (i + j * f - 1) / 2;
      }
      else {
        return (j + (f - 1) * ((j - 1) / (f + 1)) + f - 2) / 2;
      }
```

```
61      }
62
63      size_type left_child(size_type i) const {
64        size_type j = i / f;
65        if (i < f / 2 + j * f) {
66          return 2 * i - j * f + 1;
67        }
68        else {
69          return f * (2 * i + (1 - f) * j - f + 2);
70        }
71      }
72
73      size_type right_child(size_type i) const {
74        size_type j = i / f;
75        if (i < f / 2 + j * f) {
76          return 2 * i - j * f + 2;
77        }
78        else {
79          return f * (2 * i + (1 - f) * j - f + 3);
80        }
81      }
82
83  #if defined(BRANCHLESS) || defined(UNROLLED)
84
85      N* choose(bool condition, N* x, N* y) {
86        return (N*)((char*) y + condition * ((char*) x - (char*) y));
87      }
88
89  #endif
90
91      int height(N* t) const {
92        if (t == nullptr) {
93          return -1;
94        }
95        return 1 + std::max(height((*t).left()), height((*t).right()));
96      }
97
98      template <typename I>
99      size_type populate(I from, N* v) {
100       if (v == nullptr) {
101         return size_type(0);
102       }
103       size_type left_weight = populate(from, (*v).left());
104       (*v).element() = *(from + left_weight);
105       size_type right_weight = populate(from + left_weight + 1, (*v).right());
106       return left_weight + right_weight + 1;
107     }
108
109   public:
110
111     // structors
112
113     template <typename I>
114     explicit local_search_tree(I from, I to, C const & c, A const & a = A())
115       : less(c), allocator_(a), root(nullptr), n(0) {
116       n = to - from;
117       pool = allocator_.allocate(n);
118       if (n != 0) {
119         root = pool;
120       }
121       f = ARITY - 1;
122
123       // structure
124
125       for (size_type i = 0; i != n; ++i) {
```

```
126         N* current = pool + i;
127         if (i == 0) {
128           (*current).parent(nullptr);
129         }
130         else {
131           (*current).parent(pool + parent(i));
132         }
133         if (left_child(i) >= n) {
134           (*current).left(nullptr);
135         }
136         else {
137           (*current).left(pool + left_child(i));
138         }
139         if (right_child(i) >= n) {
140           (*current).right(nullptr);
141         }
142         else {
143           (*current).right(pool + right_child(i));
144         }
145       }
146
147       // data
148
149       size_type size = populate(from, root);
150       h = height(root);
151     }
152
153     ~local_search_tree() {
154       allocator_.deallocate(pool, n);
155       root = nullptr;
156       n = 0;
157     }
158
159     // accessors
160
161     C comparator() const {
162       return less;
163     }
164
165     A allocator() const {
166       return allocator_;
167     }
168
169     size_type size() const {
170       return n;
171     }
172
173   public:
174
175 #ifdef BRANCHLESS
176
177     bool is_member(V const& e) {
178       N* y = nullptr; // candidate node
179       N* x = root; // current node
180       while (x != nullptr) {
181         bool smaller = less(e, (*x).element());
182         y = choose(smaller, y, x);
183         x = choose(smaller, (*x).left(), (*x).right());
184       }
185       if (y == nullptr || less((*y).element(), e)) {
186         return false;
187       }
188       return true;
189     }
190
```

```
191 #elif defined(UNROLLED)
192
193     bool is_member(V const& e) {
194       N* y = nullptr;
195       N* x = root;
196       bool smaller;
197       switch (h) {
198       case 31:
199         smaller = less(e, (*x).element());
200         y = choose(smaller, y, x);
201         x = choose(smaller, (*x).left(), (*x).right());
202       case 30:
203         smaller = less(e, (*x).element());
204         y = choose(smaller, y, x);
205         x = choose(smaller, (*x).left(), (*x).right());
206       case 29:
207         smaller = less(e, (*x).element());
208         y = choose(smaller, y, x);
209         x = choose(smaller, (*x).left(), (*x).right());
210       case 28:
211         smaller = less(e, (*x).element());
212         y = choose(smaller, y, x);
213         x = choose(smaller, (*x).left(), (*x).right());
214       case 27:
215         smaller = less(e, (*x).element());
216         y = choose(smaller, y, x);
217         x = choose(smaller, (*x).left(), (*x).right());
218       case 26:
219         smaller = less(e, (*x).element());
220         y = choose(smaller, y, x);
221         x = choose(smaller, (*x).left(), (*x).right());
222       case 25:
223         smaller = less(e, (*x).element());
224         y = choose(smaller, y, x);
225         x = choose(smaller, (*x).left(), (*x).right());
226       case 24:
227         smaller = less(e, (*x).element());
228         y = choose(smaller, y, x);
229         x = choose(smaller, (*x).left(), (*x).right());
230       case 23:
231         smaller = less(e, (*x).element());
232         y = choose(smaller, y, x);
233         x = choose(smaller, (*x).left(), (*x).right());
234       case 22:
235         smaller = less(e, (*x).element());
236         y = choose(smaller, y, x);
237         x = choose(smaller, (*x).left(), (*x).right());
238       case 21:
239         smaller = less(e, (*x).element());
240         y = choose(smaller, y, x);
241         x = choose(smaller, (*x).left(), (*x).right());
242       case 20:
243         smaller = less(e, (*x).element());
244         y = choose(smaller, y, x);
245         x = choose(smaller, (*x).left(), (*x).right());
246       case 19:
247         smaller = less(e, (*x).element());
248         y = choose(smaller, y, x);
249         x = choose(smaller, (*x).left(), (*x).right());
250       case 18:
251         smaller = less(e, (*x).element());
252         y = choose(smaller, y, x);
253         x = choose(smaller, (*x).left(), (*x).right());
254       case 17:
255         smaller = less(e, (*x).element());
```

```
256         y = choose(smaller, y, x);
257         x = choose(smaller, (*x).left(), (*x).right());
258       case 16:
259         smaller = less(e, (*x).element());
260         y = choose(smaller, y, x);
261         x = choose(smaller, (*x).left(), (*x).right());
262       case 15:
263         smaller = less(e, (*x).element());
264         y = choose(smaller, y, x);
265         x = choose(smaller, (*x).left(), (*x).right());
266       case 14:
267         smaller = less(e, (*x).element());
268         y = choose(smaller, y, x);
269         x = choose(smaller, (*x).left(), (*x).right());
270       case 13:
271         smaller = less(e, (*x).element());
272         y = choose(smaller, y, x);
273         x = choose(smaller, (*x).left(), (*x).right());
274       case 12:
275         smaller = less(e, (*x).element());
276         y = choose(smaller, y, x);
277         x = choose(smaller, (*x).left(), (*x).right());
278       case 11:
279         smaller = less(e, (*x).element());
280         y = choose(smaller, y, x);
281         x = choose(smaller, (*x).left(), (*x).right());
282       case 10:
283         smaller = less(e, (*x).element());
284         y = choose(smaller, y, x);
285         x = choose(smaller, (*x).left(), (*x).right());
286       case 9:
287         smaller = less(e, (*x).element());
288         y = choose(smaller, y, x);
289         x = choose(smaller, (*x).left(), (*x).right());
290       case 8:
291         smaller = less(e, (*x).element());
292         y = choose(smaller, y, x);
293         x = choose(smaller, (*x).left(), (*x).right());
294       case 7:
295         smaller = less(e, (*x).element());
296         y = choose(smaller, y, x);
297         x = choose(smaller, (*x).left(), (*x).right());
298       case 6:
299         smaller = less(e, (*x).element());
300         y = choose(smaller, y, x);
301         x = choose(smaller, (*x).left(), (*x).right());
302       case 5:
303         smaller = less(e, (*x).element());
304         y = choose(smaller, y, x);
305         x = choose(smaller, (*x).left(), (*x).right());
306       default:
307         do {
308           bool smaller = less(e, (*x).element());
309           y = choose(smaller, y, x);
310           x = choose(smaller, (*x).left(), (*x).right());
311         } while (x != nullptr);
312       }
313       if ((y == nullptr) || less((*y).element(), e)) {
314         return false;
315       }
316       return true;
317     }
318
319 #else
320
```

```
321     bool is_member(V const & e) {
322       N* y = nullptr; // candidate node
323       N* x = root; // current node
324       while (x != nullptr) {
325         if (less(e, (*x).element())) {
326           x = (*x).left();
327         }
328         else {
329           y = x;
330           x = (*x).right();
331         }
332       }
333       if (y == nullptr || less((*y).element(), e)) {
334         return false;
335       }
336       return true;
337     }
338
339 #endif
340
341   };
342 }
```

§ 9 *branchless_local_search_tree.h++*

```
1  /*
2    An implementation of a local search tree;
3    - memory layout is local
4    - search procedure is branchless
5
6    Author: Jyrki Katajainen © 2012, 2013
7  */
8
9  #define local_search_tree branchless_local_search_tree
10 #define BRANCHLESS
11
12 #include "local_search_tree.h++"
```

Implicit local search trees

§ 10 *implicit_local_search_tree.h++*

```
1  /*
2    An implementation of an implicit local search tree
3
4    Author: Jyrki Katajainen © 2012
5  */
6
7  #define ARITY 16
8
9  #include <functional> // std::less
10 #include <memory> // std::allocator
11
12 namespace cphstl {
13
14   template <
15     typename V,
16     typename C = std::less<V>,
17     typename A = std::allocator<V>
18   >
19   class implicit_local_search_tree {
20   public:
21
```

```
22    // types
23
24    typedef V value_type;
25    typedef C comparator_type;
26    typedef A allocator_type;
27    typedef unsigned int size_type;
28    typedef int difference_type;
29
30  protected:
31
32    // variables
33
34    C less;
35    A allocator_;
36    size_type n;
37    V* pool;
38    size_type f;
39
40  private:
41
42    // helpers
43
44    size_type parent(size_type i) const {
45      size_type j = i / f;
46      if (i != j * f) {
47        return (i + j * f - 1) / 2;
48      }
49      else {
50        return (j + (f - 1) * ((j - 1) / (f + 1)) + f - 2) / 2;
51      }
52    }
53
54    size_type left_child(size_type i) const {
55      size_type j = i / f;
56      if (i < f / 2 + j * f) {
57        return 2 * i - j * f + 1;
58      }
59      else {
60        return f * (2 * i + (1 - f) * j - f + 2);
61      }
62    }
63
64    size_type right_child(size_type i) const {
65      size_type j = i / f;
66      if (i < f / 2 + j * f) {
67        return 2 * i - j * f + 2;
68      }
69      else {
70        return f * (2 * i + (1 - f) * j - f + 3);
71      }
72    }
73
74    V const & element(size_type i) const {
75      return *(pool + i);
76    }
77
78    template <typename I>
79    size_type populate(I from, V* v) {
80      if (v == nullptr) {
81        return size_type(0);
82      }
83      size_type i = v - pool;
84      V* u = nullptr;
85      if (left_child(i) < n) {
86        u = pool + left_child(i);
```

```
87        }
88        size_type left_weight = populate(from, u);
89        *v = *(from + left_weight);
90        V* w = nullptr;
91        if (right_child(i) < n) {
92          w = pool + right_child(i);
93        }
94        size_type right_weight = populate(from + left_weight + 1, w);
95        return left_weight + right_weight + 1;
96      }
97
98    public:
99
100     // structors
101
102     template <typename I>
103     explicit implicit_local_search_tree(I from, I to, C const& c, A const& a =
            A())
104       : less(c), allocator_(a), n(0) {
105       n = to - from;
106       pool = allocator_.allocate(n);
107       f = ARITY - 1;
108       size_type size = populate(from, pool);
109     }
110
111     ~implicit_local_search_tree() {
112       allocator_.deallocate(pool, n);
113       n = 0;
114     }
115
116     // accessors
117
118     C comparator() const {
119       return less;
120     }
121
122     A allocator() const {
123       return allocator_;
124     }
125
126     size_type size() const {
127       return n;
128     }
129
130 #ifdef BRANCHLESS
131
132   private:
133
134     int choose(bool condition, int x, int y) {
135       return y + condition * (x - y);
136     }
137
138   public:
139
140     bool is_member(V const& e) {
141       int y = -1;
142       int x = 0;
143       while (size_type(x) < n) {
144         bool smaller = less(e, *(pool + x));
145         y = choose(smaller, y, x);
146         x = choose(smaller, left_child(x), right_child(x));
147       }
148       if (y == -1 || less(*(pool + y), e)) {
149         return false;
150       }
```

```
151       return true;
152     }
153
154 #else
155
156     bool is_member(V const & e) {
157       int y = -1;
158       int x = 0;
159       while (size_type(x) < n) {
160         if (less(e, *(pool + x))) {
161           x = left_child(x);
162         }
163         else {
164           y = x;
165           x = right_child(x);
166         }
167       }
168       if (y == -1 || less(*(pool + y), e)) {
169         return false;
170       }
171       return true;
172     }
173
174 #endif
175
176   };
177 }
```

§ 11 *branchless_implicit_local_search_tree.h++*

```
1 /*
2   An implementation of an implicit local search tree;
3   - memory layout is local
4   - search procedure is branchless
5
6   Author: Jyrki Katajainen © 2012, 2013
7 */
8
9 #define implicit_local_search_tree branchless_implicit_local_search_tree
10 #define BRANCHLESS
11
12 #include "implicit_local_search_tree.h++"
```

Other search trees

§ 12 *red_black_tree.h++*

```
1 /*
2   A wrapper around the library red–black tree
3
4   Author: Jyrki Katajainen © 2012
5 */
6
7 #include <algorithm> // std::random_shuffle
8 #include <cstddef> // std::size_t
9 #include <memory> // std::allocator
10 #include <set>
11 #include <utility> // std::pair
12
13 namespace cphstl {
14
15   template <typename V, typename C, typename A = std::allocator<V> >
16   class red_black_tree {
```

```
17    public:
18
19      // types
20
21      typedef V value_type;
22      typedef C comparator_type;
23      typedef A allocator_type;
24      typedef std::size_t size_type;
25
26    protected:
27
28      // variables
29
30      C less;
31      A allocator_;
32      std::set<V, C, A> s;
33
34    public:
35
36      // structors
37
38      template <typename I>
39      explicit red_black_tree(I from, I to, C const & c, A const & a = A())
40        : less(c), allocator_(a), s(c, a) {
41        std::random_shuffle(from, to);
42        typedef typename std::set<V, C, A>::iterator iterator;
43        std::pair<iterator, bool> r;
44        for (I i = from; i < to; ++i) {
45          r = s.insert(*i);
46          if (r.second != true) {
47            std::cerr << "Duplicate:" << *i << std::endl;
48          }
49        }
50      }
51
52      ~red_black_tree() {
53        s.clear();
54      }
55
56      // accessors
57
58      C comparator() const {
59        return less;
60      }
61
62      A allocator() const {
63        return allocator_;
64      }
65
66      size_type size() const {
67        return s.size();
68      }
69
70      bool is_member(V const & e) {
71        typedef typename std::set<V, C, A>::iterator iterator;
72        iterator a = s.find(e);
73        return a != s.end();
74      }
75    };
76  }
```

§ 13 *sorted_array.h++*

```
1  /*
```

```
2   An implementation of an implicit binary search tree (a sorted array)
3
4   Author: Jyrki Katajainen © 2012, 2013
5 */
6
7 #include <algorithm> // std::sort std::binary_search
8
9 #include <cstddef> // std::size_t
10 #include <iterator> // std::iterator_traits
11 #include <memory> // std::allocator
12 #include <vector>
13
14 namespace cphstl {
15
16   template <typename V, typename C, typename A = std::allocator<V> >
17   class sorted_array {
18   public:
19
20     // types
21
22     typedef V value_type;
23     typedef C comparator_type;
24     typedef A allocator_type;
25     typedef std::size_t size_type;
26
27   protected:
28
29     // variables
30
31     C less;
32     A allocator_;
33     std::vector<V, A> v;
34
35   public:
36
37     // structors
38
39     template <typename I>
40     explicit sorted_array(I from, I to, C const & c, A const & a = A())
41       : less(c), allocator_(a), v(a) {
42       for (I i = from; i < to; ++i) {
43         v.push_back(*i);
44       }
45       std::sort(v.begin(), v.end(), c);
46     }
47
48     ~sorted_array() {
49       v.clear();
50     }
51
52     // accessors
53
54     C comparator() const {
55       return less;
56     }
57
58     A allocator() const {
59       return allocator_;
60     }
61
62     size_type size() const {
63       return v.size();
64     }
65
66 #if defined(BRANCHLESS)
```

```
67
68      V* choose(bool condition, V* x, V* y) {
69        return (V*)((char*) y + condition * ((char*) x - (char*) y));
70      }
71
72      int choose(bool condition, int x, int y) {
73        return y + condition * (x - y);
74      }
75
76      template <typename T, typename Comparator>
77      T* lower_bound(T* first, T* last, const T& e, Comparator less) {
78        int len = std::distance(first, last);
79        while (len > 0) {
80          int half = len >> 1;
81          T* middle = first;
82          std::advance(middle, half);
83          bool condition = less(*middle, e);
84          first = choose(condition, middle + 1, first);
85          len = choose(condition, len - half - 1, half);
86        }
87        return first;
88      }
89
90      bool is_member(V const& e) {
91        V* i = lower_bound( &v[0], &v[v.size()], e, less);
92        return i != &v[v.size()] && ! less(e, *i);
93      }
94
95  #elif defined(THREE_WAY)
96
97      bool is_member(V const& e) {
98        return cphstl::binary_search(v.begin(), v.end(), e, less);
99      }
100
101 #else
102
103     bool is_member(V const& e) {
104       return std::binary_search(v.begin(), v.end(), e, less);
105     }
106
107 #endif
108
109   };
110 }
```

§ 14 *branchless_sorted_array.h++*

```
1  /*
2    An implementation of an implicit search tree;
3    - sorted array
4    - search procedure is branchless
5
6    Author: Jyrki Katajainen © 2012, 2013
7  */
8
9  #define sorted_array branchless_sorted_array
10 #define BRANCHLESS
11
12 #include "sorted_array.h++"
```

Drivers

§ 15 *model-driver.c++*

```
#define REPETITIONS 10000000

#include <climits>
#define MAXSIZE UINT_MAX

#include <cmath> // ilogb
#include <random>
#include "timer.h++"

#include "model.h++"

template <typename position, typename integer>
void generate(position p, position r, integer n) {

  double lower_bound = 0.0;
  double upper_bound = 1.0;
  std::uniform_real_distribution<double> unif(lower_bound, upper_bound);
  std::default_random_engine re;
  re.seed(123456789);
  for (position q = p; q < r; ++q) {
    *q = integer(unif(re) * n);
  }
}

void usage(int argc, char **argv) {
  std::cerr << "Usage: " << argv[0] << " <lg n>" << std::endl;
  exit(1);
}

int main(int argc, char** argv) {
  typedef unsigned int element;

  unsigned int N = 15;
  if (argc == 2) {
    N = atoi(argv[1]);
  }
  else if (argc > 2) {
    usage(argc, argv);
  }
  if (N < 1 || N > MAXSIZE) {
    std::cerr << "N out of bounds [1.." << MAXSIZE << "]" << std::endl;
    usage(argc, argv);
  }
  if (N == 999) {
    N = MAXSIZE;
  }

  element* r = new element[REPETITIONS];
  generate(r, r + REPETITIONS, N);
  timer clock;
  unsigned int check_sum = 0;
  clock.start();
  for (volatile unsigned int t = 0; t < REPETITIONS; ++t) {
    check_sum += NAME::search(N, r[t]);
  }
  clock.stop();
  delete[] r;
  return 0;
}
```

§ 16 *tree-driver.c++*

```
#define MAXSIZE 64 * 1024 * 1024

#define REPETITIONS 1000000

#include <algorithm> // std::min std::swap
#include <cmath> // ilogb
#include <cstdlib> // std::rand, std::srand
#include <functional> // std::less
#include <iostream> // std::cerr and std::cout
#include <iterator> // std::iterator_traits
#include <random>
#include "timer.h++"
#include <vector>

#include "data-structure.h++"

template <typename position>
void generate(position p, position r) {
  double lower_bound = 0.0;
  double upper_bound = 1.0;
  std::uniform_real_distribution<double> unif(lower_bound, upper_bound);
  std::default_random_engine re;
  re.seed(123456789);
  typedef typename std::iterator_traits<position>::value_type element;
  for (position q = p; q < r; ++q) {
    *q = element(unif(re));
  }
}

template <typename position>
void create_permutation(position p, position r) {
  typedef typename std::iterator_traits<position>::difference_type index;
  typedef typename std::iterator_traits<position>::value_type element;
  index n = r - p;
  position q = p;
  for (index i = 0; i < n; ++i) {
    *q = element(i);
    ++q;
  }
}

template <typename R, typename integer>
void shuffle(R p, R r, integer how_many) {
  typedef typename std::iterator_traits<R>::difference_type index;
  std::srand(123456789);
  index n = r - p;
  index repetitions = std::min(n, index(how_many));
  index j = n - 1;
  index t = repetitions;
  while (t > 0) {
    index i = std::rand() % (j + 1);
    std::swap(p[i], p[j]);
    --j;
    --t;
  }
  j = n - 1;
  t = 0;
  while (t < repetitions) {
    std::swap(p[t], p[j]);
    ++t;
    --j;
  }
}
```

```
64
65 void usage(int argc, char **argv) {
66   std::cerr << "Usage: " << argv[0] << " <lg n>" << std::endl;
67   exit(1);
68 }
69
70 int main(int argc, char** argv) {
71   typedef int V;
72   typedef std::less<V> C;
73   unsigned int N = 15;
74   if (argc == 2) {
75     N = atoi(argv[1]);
76   }
77   else if (argc > 2) {
78     usage(argc, argv);
79   }
80   if (N < 1 || N > MAXSIZE) {
81     std::cerr << "N out of bounds [1.." << MAXSIZE << "]" << std::endl;
82     usage(argc, argv);
83   }
84
85   std::vector<V> v(N);
86   create_permutation(v.begin(), v.end());
87   cphstl::NAME<V, C> tree(v.begin(), v.end(), C());
88   ::shuffle(v.begin(), v.end(), REPETITIONS);
89   timer clock;
90   unsigned int check_sum = 0;
91   clock.start();
92   for (volatile unsigned int t = 0; t < REPETITIONS; ++t) {
93     unsigned int i = t % N;
94     check_sum += tree.is_member(v[i]);
95   }
96   clock.stop();
97   double nano_seconds = clock.getElapsedTimeNanoSec() / double(REPETITIONS);
98   double scaled_time = nano_seconds / double(ilogb(N));
99   std::cout << N << '\t' << scaled_time << std::endl;
100  return 0;
101 }
```

Timers

§ 17 *timer.h++*

```
1 /*
2   This high–resolution timer is able to measure the elapsed time with
3   one microsecond accuracy
4
5   Author: Song Ho Ahn (song.ahn@gmail.com) © 2003, 2006
6 */
7
8 #include <sys/time.h>
9
10 class timer {
11 public:
12
13   timer();
14   ~timer();
15
16   void start();
17   void stop();
18   double getElapsedTime();         // get elapsed time in seconds
19   double getElapsedTimeSec();      // same as getElapsedTime
20   double getElapsedTimeMilliSec();// get elapsed time in milliseconds
21   double getElapsedTimeMicroSec();// get elapsed time in microseconds
```

```
22   double getElapsedTimeNanoSec(); // get elapsed time in nanoseconds
23
24 private:
25
26   double startTimeMicroSec;       // starting time in microseconds
27   double endTimeMicroSec;         // ending time in microseconds
28   int    stopped;                 // stop flag
29   timeval startCount;
30   timeval endCount;
31 };
32
33 #include "timer.i++"
```

§ 18 *timer.i++*

```
1  /*
2    This high-resolution timer is able to measure the elapsed time with
3    one micro-second accuracy
4
5    Author: Song Ho Ahn (song.ahn@gmail.com) © 2003, 2006
6  */
7
8  #include <stdlib.h>
9
10 /*
11   constructor
12 */
13 timer::timer() {
14   startCount.tv_sec = startCount.tv_usec = 0;
15   endCount.tv_sec = endCount.tv_usec = 0;
16   stopped = 0;
17   startTimeMicroSec = 0;
18   endTimeMicroSec = 0;
19 }
20
21 /*
22   destructor
23 */
24 timer::~timer() {
25 }
26
27 /*
28   start timer; startCount will be set at this point
29 */
30 void timer::start() {
31   stopped = 0; // reset stop flag
32   gettimeofday( &startCount, NULL);
33 }
34
35 /*
36   stop the timer; endCount will be set at this point
37 */
38 void timer::stop() {
39   stopped = 1; // set timer stopped flag
40   gettimeofday( &endCount, NULL);
41 }
42
43 /*
44   multiply elapsedTimeMicroSec by 1000
45 */
46 double timer::getElapsedTimeNanoSec() {
47   return getElapsedTimeMicroSec() * 1000.0;
48 }
49
```

```
/*
  compute elapsed time in micro-second resolution;
  other getElapsedTime will call this, then convert to correspond resolution
*/
double timer::getElapsedTimeMicroSec() {
  if (! stopped) {
    gettimeofday(&endCount, NULL);
  }
  startTimeMicroSec = (startCount.tv_sec * 1000000.0) + startCount.tv_usec;
  endTimeMicroSec = (endCount.tv_sec * 1000000.0) + endCount.tv_usec;
  return endTimeMicroSec - startTimeMicroSec;
}

/*
  divide elapsedTimeMicroSec by 1000
*/
double timer::getElapsedTimeMilliSec() {
  return getElapsedTimeMicroSec() * 0.001;
}

/*
  divide elapsedTimeMicroSec by 1000000
*/
double timer::getElapsedTimeSec() {
  return getElapsedTimeMicroSec() * 0.000001;
}

/*
  same as getElapsedTimeSec()
*/
double timer::getElapsedTime() {
  return getElapsedTimeSec();
}
```

Makefile

§ 19 *makefile*

```
#CXX=icpc -I/usr/include/i386-linux-gnu/
#CXXFLAGS= -fast -Wall -x c++
CXX=g++
CXXFLAGS= -O3 -Wall -std=c++0x
TESTFLAGS= -O3 -Wall -pedantic -std=c++0x -x c++ -DDEBUG -g

model-files:= $(wildcard *model.h++)
model-bases:= $(basename $(model-files))
model-times:= $(addsuffix .time, $(model-bases))
model-comparisons:= $(addsuffix .comp, $(model-bases))
model-mispredictions:= $(addsuffix .branch, $(model-bases))
model-alphas:= $(addsuffix .alpha, $(model-bases))
model-statistics:= $(addsuffix .statistics, $(model-bases))

tree-files:= $(wildcard *tree.h++) $(wildcard *array.h++)
tree-bases:= $(basename $(tree-files))
tree-times:= $(addsuffix .time, $(tree-bases))
tree-comparisons:= $(addsuffix .comp, $(tree-bases))
tree-mispredictions:= $(addsuffix .branch, $(tree-bases))
tree-misses:= $(addsuffix .cache, $(tree-bases))
tree-alphas:= $(addsuffix .alpha, $(tree-bases))
tree-arities:= $(addsuffix .arity, $(tree-bases))

N = 65536 16777216 999
M = 1024 32768 1048576 33554432
alpha = 2 3 4 5
```

```
arity = 4 8 16 32 64 128 256 512 1024 2048 4096 8192 16384 32768 65536 131072
    262144
S = 25000

$(model-times): %.time : %.h++
        @cp $*.h++ model.h++
        $(CXX) $(CXXFLAGS) -DNAME=$* model-driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f ./a.out

$(model-comparisons): %.comp : %.h++
        @cp $*.h++ model.h++
        $(CXX) $(CXXFLAGS) -DMEASURE_COMPARISONS -DNAME=$* model-driver.c++
        @for n in $(N) ; do \
          ./a.out $$n ; \
        done; \
        rm -f ./a.out

$(model-mispredictions): %.branch : %.h++
        @cp $*.h++ model.h++
        @for n in $(N) ; do \
          python branch_mispredictions.py model-driver.c++ $* $$n ; \
          rm -f ./a.out ; \
          rm -f ./cachegrind.out.* ; \
        done

$(model-alphas): %.alpha : %.h++
        @cp $*.h++ model.h++
        @for n in $(N) ; do \
          for a in $(alpha) ; do \
            echo "alpha: " $$a ; \
            $(CXX) $(CXXFLAGS) -DALPHA=$$a -DNAME=$* model-driver.c++ ;\
            ./a.out $$n ; \
          done \
        done
        rm -f ./a.out

$(model-statistics): %.statistics : %.h++
        @cp $*.h++ model.h++
        @for n in $(S) ; do \
          for a in $(alpha) ; do \
            echo "alpha: " $$a ; \
            $(CXX) $(CXXFLAGS) -DGATHER_STATISTICS -DALPHA=$$a -DNAME=$* model-
                driver.c++ ;\
            ./a.out $$n ; \
          done \
        done
        rm -f ./a.out

$(tree-times): %.time : %.h++
        @cp $*.h++ data-structure.h++
        $(CXX) $(CXXFLAGS) -DNAME=$* tree-driver.c++
        @for n in $(M) ; do \
          ./a.out $$n ; \
        done; \
        rm -f ./a.out

$(tree-comparisons): %.comp : %.h++
        @cp $*.h++ data-structure.h++
        $(CXX) $(CXXFLAGS) -DMEASURE_COMPARISONS -DNAME=$* tree-driver.c++
        @for n in $(M) ; do \
          ./a.out $$n ; \
        done; \
```

```
#       rm -f ./a.out

$(tree-mispredictions): %.branch : %.h++
        @cp $*.h++ data-structure.h++
        @for n in $(M) ; do \
          python branch_mispredictions.py tree-driver.c++ $* $$n ; \
          rm -f ./a.out ; \
          rm -f ./cachegrind.out.* ; \
        done

$(tree-misses): %.cache : %.h++
        @cp $*.h++ data-structure.h++
        @for n in $(M) ; do \
          python cache_misses.py tree-driver.c++ $* $$n ; \
          rm -f ./cachegrind.out.* ; \
        done

#         rm -f ./a.out ; \

$(tree-alphas): %.alpha : %.h++
        @cp $*.h++ data-structure.h++
        @for n in $(M) ; do \
          for a in $(alpha) ; do \
            echo "alpha: " $$a ; \
            $(CXX) $(CXXFLAGS) -DALPHA=$$a -DNAME=$* tree-driver.c++ ;\
            ./a.out $$n ; \
          done \
        done
        rm -f ./a.out

$(tree-arities): %.arity : %.h++
        @cp $*.h++ data-structure.h++
        @for n in $(M) ; do \
        echo "n: " $$n ; \
          for a in $(arity) ; do \
            echo "  " $$a ; \
            $(CXX) $(CXXFLAGS) -DARITY=$$a -DNAME=$* tree-driver.c++ ;\
            ./a.out $$n ; \
          done \
        done
        rm -f ./a.out

beta:
        gcc -O3 -Wall beta.c
        @./a.out 31

gamma:
        gcc -O3 -Wall gamma.c
        @./a.out 31

tau:
        gcc -O3 -Wall tau.c
        @./a.out 29 64

skewed:
        @for a in 2 3 4 ; do \
          echo "alpha: " $$a ; \
          $(CXX) $(TESTFLAGS) -DALPHA=$$a -DUNITTEST_SKEWED_SEARCH_TREE
               skewed_search_tree.h++ ;\
          ./a.out ; \
        done

unrolled:
        $(CXX) $(TESTFLAGS) -DUNITTEST_LOCAL_SEARCH_TREE
             unrolled_local_search_tree.h++
```

```
153         ./a.out
154
155 random:
156         $(CXX) $(TESTFLAGS) -DUNITTEST_SKEWED_SEARCH_TREE skewed_search_tree.h++
                ;\
157         ./a.out
158
159 local:
160         @for a in 2 4 8 ; do \
161           echo "arity: " $$a ; \
162           $(CXX) $(TESTFLAGS) -DARITY=$$a -DUNITTEST_LOCAL_SEARCH_TREE
                local_search_tree.h++ ; \
163           ./a.out ; \
164         done
165
166 implicit:
167         @for a in 2 4 8 ; do \
168           echo "arity: " $$a ; \
169           $(CXX) $(TESTFLAGS) -DUNITTEST_IMPLICIT_LOCAL_SEARCH_TREE
                implicit_local_search_tree.h++ ;\
170           ./a.out ; \
171         done
172
173 # Other tools
174
175 clean:
176         - rm  -f a.out temp cachegrind.out.* *.o 2>/dev/null
177
178 veryclean: clean
179         - rm -f *~ */*~ 2>/dev/null
180
181 find:
182         find . -type f -print -exec grep $(word) {} \; | less # or -name '*.cc'
```