

Work-Efficient B^+ -Trees: Electronic Appendix

Henrik Thorup Andersen

*Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark*

Abstract. This report is an electronic appendix to the paper “Work-Efficient B^+ -Trees” 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. External-memory data structures, database indexes, B -trees, B^+ -trees

Copyright notice

Copyright © 2000–2013 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH STL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH STL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

2013-06-03

Included files

File	Page
§ 1 improved_btree.hpp	3
§ 2 internal_rbtrees.hpp	21
§ 3 pool.hpp	30

B^+ -tree§ 1 *improved_btree.hpp*

```

1  #pragma once
2  #include <cassert>
3  #include <cstdint>
4  #include <memory>
5  #include <utility>
6  #include <iostream>
7  #include "internal_rbtrees.hpp"
8
9  // this class is never instantiated. it has one use: as the template parameter
10 // to an allocator. this way we can use one allocator for both node types
11 template<int size>
12 struct alignas(size) allocated_node
13     {
14         uint8_t padding[size];
15     };
16
17 // template parameters:
18 // K: key type
19 // M: mapped type
20 // node_size: size of the nodes – should be tuned to page size
21 // address_type: type used for addresses – typically uint16_t or uint32_t
22 // compare_function: compare function, defaults to std::less
23 // allocator_type: a template taking a type, used as allocator
24 template<
25     typename K,
26     typename M,
27     size_t node_size = 512,
28     typename address_type = uint16_t,
29     class compare_function = std::less<K>,
30     class allocator_type = std::allocator<allocated_node<node_size>>>
31 class improved_btree
32     {
33     // TYPEDEFS
34 public:
35
36     typedef K key_type;
37     typedef M mapped_type;
38     typedef std::pair<const K, M> value_type;
39
40     // NODE DEFINITIONS
41 private:
42
43     struct alignas(node_size) abstract_node {};
44
45     struct alignas(node_size) inner_node : public abstract_node
46     {
47         typedef internal_rbtrees<key_type, abstract_node*, address_type,
48             node_size – sizeof(inner_node*), compare_function>
49             tree_type;
50
51         tree_type internal_tree;
52         inner_node* second;
53
54         inner_node(): second(nullptr) {}
55     };
56
57     struct alignas(node_size) leaf_node : public abstract_node
58     {
59         typedef internal_rbtrees<key_type, mapped_type, address_type,
60             node_size – sizeof(leaf_node*), compare_function>

```

```

        tree_type;
60
61     tree_type internal_tree;
62     leaf_node* second;
63
64     leaf_node(): second(nullptr) {}
65     };
66
67     // ITERATOR TYPE
68 public:
69
70     class iterator
71     {
72     private:
73
74         friend improved_btree;
75
76         iterator(const leaf_node* node_, address_type address_)
77         :     node(node_), address(address_)
78         {}
79
80     public:
81         iterator()
82         :     node(nullptr), address()
83         {}
84
85         const value_type & operator*()
86         {
87             assert(node != nullptr);
88             return node->internal_tree.pool[address].value;
89         }
90
91         const value_type* operator->()
92         {
93             assert(node != nullptr);
94             return &node->internal_tree.pool[address].value;
95         }
96
97         // TODO: increment and decrement
98
99     private:
100         const leaf_node* node;
101         address_type address;
102     };
103
104     // DATA MEMBERS
105 private:
106
107     abstract_node* root;
108     unsigned height;
109
110     allocator_type allocator;
111
112     // ALLOCATOR WRAPPERS
113 private:
114
115     inner_node* allocate_inner()
116     { return (inner_node*)(allocator.allocate(1)); }
117
118     leaf_node* allocate_leaf()
119     { return (leaf_node*)(allocator.allocate(1)); }
120
121     void deallocate_inner(inner_node* node)
122     { allocator.deallocate((allocated_node<node_size>*)node, 1); }
123

```

```

124     void deallocate_leaf(leaf_node* node)
125         { allocator.deallocate((allocated_node<node_size>*)node, 1); }
126
127     // CONSTRUCTORS ET CETERA
128 public:
129
130     improved_btree()
131     :     height(0)
132         {
133             root = allocate_leaf();
134             new (root) leaf_node();
135         }
136
137     ~improved_btree()
138     {
139         destruct(root, height);
140     }
141
142     // TODO: copying
143     improved_btree(improved_btree &) = delete;
144     void operator=(improved_btree &) = delete;
145
146 private:
147
148     void destruct(abstract_node* node, const unsigned level)
149     {
150         if (level != 0)
151             { destruct(static_cast<inner_node*>(node), level); }
152         else
153             { destruct(static_cast<leaf_node*>(node)); }
154     }
155
156     void destruct(inner_node* node, const unsigned level)
157     {
158         if (node->second != nullptr)
159             {
160                 auto x = node->second->internal_tree.min_address();
161                 while (x != node->second->internal_tree.max_address())
162                     {
163                         destruct(node->second->internal_tree.pool[x].
164                             value.second,
165                             level-1);
166                         x = node->second->internal_tree.successor(x);
167                     }
168                 destruct(node->second->internal_tree.pool[x].value.second
169                     ,
170                     level-1);
171                 deallocate_inner(node->second);
172             }
173
174         auto x = node->internal_tree.min_address();
175         while (x != node->internal_tree.max_address())
176             {
177                 destruct(node->internal_tree.pool[x].value.second,
178                     level-1);
179                 x = node->internal_tree.successor(x);
180             }
181         destruct(node->internal_tree.pool[x].value.second,
182             level-1);
183
184         deallocate_inner(node);
185     }
186
187     void destruct(leaf_node* node)

```

```

187         {
188         if (node->second != nullptr)
189             { deallocate_leaf(node->second); }
190
191         deallocate_leaf(node);
192     }
193
194     // FIND
195 public:
196
197     // returns the largest value that is less than or equal to the key given,
198     // or, if all values are larger than the key, returns the minimum value
199     // todo: implement iterators
200     iterator find(const key_type & k) const
201     {
202         unsigned level = height;
203         const abstract_node* current_node = root;
204
205         // search inner nodes
206         while (level != 0)
207             {
208                 const inner_node* first = static_cast<const inner_node*>(
209                     current_node);
210                 const inner_node* second = first->second;
211
212                 if (second != nullptr and
213                     not compare_function()(k, min_key(second)))
214                     {
215                         current_node =
216                             second->internal_tree.pool[
217                                 second->internal_tree.lookup(k)].
218                                 value.second;
219
220                     }
221                 else
222                     {
223                         current_node =
224                             first->internal_tree.pool[
225                                 first->internal_tree.lookup(k)].
226                                 value.second;
227
228                     }
229
230                 --level;
231             }
232
233         // get the result from the leaf
234         const leaf_node* first = static_cast<const leaf_node*>(
235             current_node);
236         const leaf_node* second = first->second;
237
238         if (second != nullptr and not compare_function()(k, min_key(
239             second)))
240             {
241                 return iterator(second, second->internal_tree.lookup(k));
242             }
243         else
244             {
245                 return iterator(first, first->internal_tree.lookup(k));
246             }
247     }
248
249     // INSERT
250 public:
251
252     void insert(const value_type & v)
253     {

```

```

247 // if the root is full, split it and make a new one
248 if (is_full(root, height))
249     { split_root(); }
250
251 unsigned level = height;
252 abstract_node* current_node = root;
253
254 // search inner nodes
255 while (level != 0)
256     {
257         inner_node* first = static_cast<inner_node*>(current_node
258             );
259         inner_node* second = first->second;
260
261         current_node = insert_inner(first, second, level, v);
262         --level;
263     }
264
265 // insert to the leaf
266 leaf_node* first = static_cast<leaf_node*>(current_node);
267 leaf_node* second = first->second;
268
269 return insert_leaf(first, second, v);
270 }
271 private:
272
273 abstract_node* insert_inner(inner_node* first, inner_node* second,
274     const unsigned level, const value_type& v)
275     {
276         // preconditions:
277         assert(not is_full(first));
278
279         // second page exists and the value belongs in it
280         if (second != nullptr and
281             not compare_function()(v.first, min_key(second)))
282             {
283                 // find the child to insert to
284                 auto insert_to_address = second->internal_tree.lookup(v.
285                     first);
286
287                 abstract_node* insert_to = second->internal_tree.pool[
288                     insert_to_address].value.second;
289
290                 // is it full?
291                 if (is_full(insert_to, level-1))
292                     {
293                         // split it
294                         abstract_node* splitted = split(insert_to, level-
295                             1);
296                         auto min_key_splitted = min_key(splitted, level-
297                             1);
298
299                         // create an element representing the new node
300                         second->internal_tree.insert(
301                             {min_key_splitted, splitted});
302
303                         // decide which of the resulting nodes to insert
304                         to
305                         insert_to = not compare_function()(v.first,
306                             min_key_splitted)
307                             ? splitted : insert_to;
308                     }
309
310                 return insert_to;

```

```

306     }
307     // second page doesn't exist or the value doesn't belong in it
308     else
309     {
310         // find the child to insert to
311         auto insert_to_address = first->internal_tree.lookup(v.
            first);
312
313         abstract_node* insert_to = first->internal_tree.pool[
314             insert_to_address].value.second;
315
316         // is it full?
317         if (is_full(insert_to, level-1))
318         {
319             // split it
320             abstract_node* splitted = split(insert_to, level-
                1);
321             auto min_key_splitted = min_key(splitted, level-
                1);
322
323             // find space for an element representing the new
                node
324             // is this page full?
325             if (first->internal_tree.full())
326             {
327                 second = create_second_if_needed(first);
328
329                 // if the child is the maximum, the
                    result of the split
330                 // will go into the second node
331                 if (insert_to_address == first->
                    internal_tree.max_address())
332                 {
333                     second->internal_tree.insert_min
                        (
334                         {
335                             min_key_splitted,
336                             splitted});
337                 }
338                 // otherwise, the maximum will go into
                    the second node
339                 else
340                 {
341                     move_from_first_to_second(first);
342
343                     first->internal_tree.insert(
344                         {min_key_splitted,
345                             splitted});
346                 }
347             }
348             // this node isn't full
349             else
350             {
351                 first->internal_tree.insert(
352                     {min_key_splitted, splitted});
353             }
354
355             // decide which of the resulting nodes to insert
                to
356             insert_to = not compare_function()(v.first,
                min_key_splitted)
                ? splitted : insert_to;
357         }
358
359         // because any keys less than the leftmost key are
            inserted into

```



```

358         // the leftmost node – the node corresponding to the
359         // (we do not have an extra node like in a regular btree)
360         // – we
361         // might need to update the key of the node we are
362         // inserting to.
363         // we can do it in this way because we know that the
364         // ordering does
365         // not change
366         if (compare_function()(v.first, first->internal_tree.pool
367         [
368             insert_to_address].value.first))
369         {
370             first->internal_tree.change_key_of_element(
371                 insert_to_address, v.first);
372         }
373         return insert_to;
374     }
375 }
376
377 void insert_leaf(leaf_node* first, leaf_node* second,
378                 const value_type& v)
379 {
380     // preconditions:
381     assert(not is_full(first));
382
383     // second node exists and the value belongs in it
384     if (second != nullptr and
385         not compare_function()(v.first, min_key(second)))
386     {
387         // simple case: just insert
388         second->internal_tree.insert(v);
389     }
390     // second node doesn't exist or the value doesn't belong in it
391     else
392     {
393         // is this node full?
394         if (first->internal_tree.full())
395         {
396             // slightly more complicated case:
397             // one of two values have to go into the second
398             // node: either the
399             // new value or the current maximum, depending on
400             // which will be
401             // the new maximum
402             auto max_key_first = max_key(first);
403             if (compare_function()(max_key_first, v.first))
404             {
405                 // if the second node doesn't exist, we
406                 // need to allocate it
407                 second = create_second_if_needed(first);
408                 // the new value is larger and is placed
409                 // in the second node
410                 second->internal_tree.insert(v);
411                 return;
412             }
413             else if (compare_function()(v.first,
414                                     max_key_first))
415             {
416                 move_from_first_to_second(first);
417                 // and the new value is placed in this
418                 // node

```

```

412     }
413     // edge case: the new value has the same key as
414     // the current
415     // maximum. in this case we just replace, which
416     // the
417     // internal_tree insert function handles
418     }
419     // this node isn't full
420     // simple case: just insert
421     first→internal_tree.insert(v);
422     }
423 }
424 // ERASE
425 public:
426 void erase(const key_type & k)
427 {
428     // check for collapse: root is inner node and has only one
429     // element
430     // (that is, minimum is the same as maximum)
431     if (height > 0 and
432         static_cast<inner_node*>(root)→internal_tree.min_address
433         () ==
434         static_cast<inner_node*>(root)→internal_tree.max_address
435         ())
436     {
437         inner_node* root_inner = static_cast<inner_node*>(root);
438         root = root_inner→internal_tree.pool[
439             root_inner→internal_tree.min_address()].value.
440         second;
441         --height;
442     }
443     unsigned level = height;
444     abstract_node* current_node = root;
445     // search inner nodes
446     while (level != 0)
447     {
448         inner_node* first = static_cast<inner_node*>(current_node
449         );
450         inner_node* second = first→second;
451         current_node = erase_inner(first, second, level, k);
452         level --;
453     }
454     // erase from the leaf
455     leaf_node* first = static_cast<leaf_node*>(current_node);
456     leaf_node* second = first→second;
457     return erase_leaf(first, second, k);
458 }
459 private:
460 abstract_node* erase_inner(inner_node* first, inner_node* second,
461 const unsigned level, const key_type & k)
462 {
463     // preconditions:
464     assert(first == root or not is_small(first));
465     address_type erase_from_address;
466     abstract_node* erase_from;

```

```

470
471 // addresses of adjacent nodes
472 bool has_left = false;
473 address_type adjacent_left_address;
474 abstract_node* adjacent_left;
475
476 bool has_right = false;
477 address_type adjacent_right_address;
478 abstract_node* adjacent_right;
479
480 // find the node to erase from
481 if (second != nullptr and not compare_function()(k, min_key(
482     second)))
483     {
484     erase_from_address = second->internal_tree.lookup(k);
485     erase_from = second->internal_tree.pool[
486         erase_from_address].value.second;
487
488     if (is_small(erase_from, level-1))
489         {
490         if (erase_from_address == second->internal_tree.
491             min_address())
492             {
493             adjacent_left_address = first->
494                 internal_tree.max_address();
495             adjacent_left = first->internal_tree.pool
496                 [
497                 adjacent_left_address].value.
498                 second;
499             }
500         else
501             {
502             adjacent_left_address =
503                 second->internal_tree.predecessor
504                     (erase_from_address);
505             adjacent_left = second->internal_tree.
506                 pool[
507                 adjacent_left_address].value.
508                 second;
509             }
510         }
511
512     if (erase_from_address != second->internal_tree.
513         max_address())
514         {
515         has_right = true;
516         adjacent_right_address =
517             second->internal_tree.successor(
518                 erase_from_address);
519         adjacent_right = second->internal_tree.
520             pool[
521             adjacent_right_address].value.
522             second;
523         }
524
525     // can we steal from the node to the left?
526     if (not is_small(adjacent_left, level-1))
527         {
528         steal_from_left_to_right(
529             adjacent_left, erase_from, level-
530             1);
531
532         // update the key of the node we stole to
533         // we can do it like
534         // this because we know that the ordering
535         // has not changed

```

```

520         second→internal_tree.
521             change_key_of_element(
522                 erase_from_address, min_key(
523                     erase_from, level-1));
524     }
525     // can we steal from the node to the right?
526     else if (has_right and not is_small(
527         adjacent_right, level-1))
528     {
529         steal_from_right_to_left(
530             erase_from, adjacent_right, level
531             -1);
532
533         // update the key of the node we stole
534         // from. we can do it
535         // like this because we know that the
536         // ordering has not
537         // changed
538         second→internal_tree.
539             change_key_of_element(
540                 adjacent_right_address, min_key(
541                     adjacent_right, level-1))
542             ;
543     }
544     // merge with the node to the left
545     else
546     {
547         merge(adjacent_left, erase_from, level-1)
548         ;
549
550         second→internal_tree.erase_address(
551             erase_from_address);
552         delete_second_if_empty(first);
553
554         erase_from = adjacent_left;
555     }
556     }
557     return erase_from;
558 }
559 else
560 {
561     erase_from_address = first→internal_tree.lookup(k);
562     erase_from = first→internal_tree.pool[
563         erase_from_address].value.second;
564
565     if (is_small(erase_from, level-1))
566     {
567         bool adjacent_right_from_second;
568
569         if (erase_from_address != first→internal_tree.
570             min_address())
571         {
572             has_left = true;
573             adjacent_left_address =
574                 first→internal_tree.predecessor(
575                     erase_from_address);
576             adjacent_left = first→internal_tree.pool
577                 [
578                     adjacent_left_address].value.
579                     second;
580         }
581
582         if (second != nullptr and
583             erase_from_address == first→
584                 internal_tree.max_address())

```

```

571         {
572         has_right = true;
573         adjacent_right_address =
574             second->internal_tree.min_address
575             ();
576         adjacent_right = second->internal_tree.
577             pool[
578                 adjacent_right_address].value.
579                 second;
580         adjacent_right_from_second = true;
581     }
582     else if (erase_from_address !=
583             first->internal_tree.max_address())
584     {
585         has_right = true;
586         adjacent_right_address =
587             first->internal_tree.successor(
588                 erase_from_address);
589         adjacent_right = first->internal_tree.
590             pool[
591                 adjacent_right_address].value.
592                 second;
593         adjacent_right_from_second = false;
594     }
595     // can we steal from the node to the left?
596     if (has_left and not is_small(adjacent_left,
597         level-1))
598     {
599         steal_from_left_to_right(
600             adjacent_left, erase_from, level-
601             1);
602         // update the key of the node we stole to
603         // we can do it like
604         // this because we know that the ordering
605         // has not changed
606         first->internal_tree.
607             change_key_of_element(
608                 erase_from_address, min_key(
609                     erase_from, level-1));
610     }
611     // can we steal from the node to the right?
612     else if (has_right and not is_small(
613         adjacent_right, level-1))
614     {
615         steal_from_right_to_left(
616             erase_from, adjacent_right, level
617             -1);
618         // update the key of the node we stole
619         // from. we can do it
620         // like this because we know that the
621         // ordering has not
622         // changed
623         if (adjacent_right_from_second)
624         {
625             second->internal_tree.
626                 change_key_of_element(
627                     adjacent_right_address,
628                     min_key(adjacent_right,
629                         level-1));
630         }
631     }
632     else
633     {

```

```

619         first→internal_tree.
620             change_key_of_element(
621                 adjacent_right_address,
622                 min_key(adjacent_right,
623                     level-1));
624     }
625     // can we merge with the node to the left?
626     else if (has_left)
627     {
628         merge(adjacent_left, erase_from, level-1)
629         ;
630
631         // erase the merged node
632         first→internal_tree.erase_address(
633             erase_from_address);
634         move_from_second_to_first(first);
635
636         erase_from = adjacent_left;
637     }
638     // last resort, merge with the node to the right
639     else
640     {
641         merge(erase_from, adjacent_right, level-
642             1);
643
644         // erase the merged node
645         if (adjacent_right_from_second)
646         {
647             second→internal_tree.
648                 erase_address(
649                     adjacent_right_address);
650             delete_second_if_empty(first);
651         }
652         else
653         {
654             first→internal_tree.
655                 erase_address(
656                     adjacent_right_address);
657             move_from_second_to_first(first);
658         }
659     }
660     }
661     return erase_from;
662 }
663 }
664
665 void erase_leaf(leaf_node* first, leaf_node* second,
666 const key_type & k)
667 {
668     // preconditions:
669     assert(first == root or not is_small(first));
670
671     if (second != nullptr and not compare_function()(k, min_key(
672         second)))
673     {
674         second→internal_tree.erase(k);
675         delete_second_if_empty(first);
676     }
677     else
678     {
679         first→internal_tree.erase(k);
680         if (not first→internal_tree.full())
681             { move_from_second_to_first(first); }
682     }
683 }

```

```

676         }
677
678         // SPLIT ROOT
679         // this handles splitting a root node into two and creating a new root
680         // with these as children. it returns a pointer to the new root. this is
681         // the only way that the tree grows in height
682 private:
683
684     void split_root()
685     {
686         abstract_node* splitted = split(root, height);
687
688         inner_node* new_root = allocate_inner();
689         new (new_root) inner_node();
690
691         new_root->internal_tree.insert(
692             std::make_pair(min_key(root, height), root));
693         new_root->internal_tree.insert(
694             std::make_pair(min_key(splitted, height), splitted));
695
696         root = new_root;
697         height += 1;
698     }
699
700     // SPLIT
701 private:
702
703     abstract_node* split(abstract_node* node, const unsigned level)
704     {
705         if (level != 0)
706             { return split(static_cast<inner_node*>(node)); }
707         else
708             { return split(static_cast<leaf_node*>(node)); }
709     }
710
711     abstract_node* split(inner_node* node)
712     {
713         // preconditions:
714         assert(is_full(node));
715
716         auto splitted = node->second;
717         node->second = nullptr;
718         return splitted;
719     }
720
721     abstract_node* split(leaf_node* node)
722     {
723         // preconditions:
724         assert(is_full(node));
725
726         auto splitted = node->second;
727         node->second = nullptr;
728         return splitted;
729     }
730
731     // STEAL FROM LEFT TO RIGHT
732 private:
733
734     void steal_from_left_to_right(abstract_node* left, abstract_node* right,
735         const unsigned level)
736     {
737         if (level != 0)
738             {
739                 steal_from_left_to_right(
740                     static_cast<inner_node*>(left),

```

```

741         static_cast<inner_node*>(right));
742     }
743     else
744     {
745         steal_from_left_to_right(
746             static_cast<leaf_node*>(left),
747             static_cast<leaf_node*>(right));
748     }
749 }
750
751 void steal_from_left_to_right(inner_node* left, inner_node* right)
752 {
753     // preconditions:
754     assert(is_small(right));
755     assert(not is_small(left));
756
757     move_from_first_to_second(right);
758
759     auto to_steal_address = left->second->internal_tree.max_address()
760     ;
761     auto& to_steal = left->second->internal_tree.pool[
762         to_steal_address].value;
763
764     right->internal_tree.insert(to_steal);
765     left->second->internal_tree.erase_address(to_steal_address);
766     delete_second_if_empty(left);
767 }
768
769 void steal_from_left_to_right(leaf_node* left, leaf_node* right)
770 {
771     // preconditions:
772     assert(is_small(right));
773     assert(not is_small(left));
774
775     move_from_first_to_second(right);
776
777     auto to_steal_address = left->second->internal_tree.max_address()
778     ;
779     auto& to_steal = left->second->internal_tree.pool[
780         to_steal_address].value;
781
782     right->internal_tree.insert(to_steal);
783     left->second->internal_tree.erase_address(to_steal_address);
784     delete_second_if_empty(left);
785 }
786
787 // STEAL FROM RIGHT TO LEFT
788 private:
789
790 void steal_from_right_to_left(abstract_node* left, abstract_node* right,
791     const unsigned level)
792 {
793     if (level != 0)
794     {
795         steal_from_right_to_left(
796             static_cast<inner_node*>(left),
797             static_cast<inner_node*>(right));
798     }
799     else
800     {
801         steal_from_right_to_left(
802             static_cast<leaf_node*>(left),
803             static_cast<leaf_node*>(right));
804     }
805 }

```



```

804
805 void steal_from_right_to_left(inner_node* left, inner_node* right)
806 {
807     // preconditions:
808     assert(is_small(left));
809     assert(not is_small(right));
810
811     create_second_if_needed(left);
812
813     auto to_steal_address = right->internal_tree.min_address();
814     auto& to_steal = right->internal_tree.pool[
815         to_steal_address].value;
816
817     left->second->internal_tree.insert(to_steal);
818     right->internal_tree.erase_address(to_steal_address);
819     move_from_second_to_first(right);
820 }
821
822 void steal_from_right_to_left(leaf_node* left, leaf_node* right)
823 {
824     // preconditions:
825     assert(is_small(left));
826     assert(not is_small(right));
827
828     create_second_if_needed(left);
829
830     auto to_steal_address = right->internal_tree.min_address();
831     auto& to_steal = right->internal_tree.pool[
832         to_steal_address].value;
833
834     left->second->internal_tree.insert(to_steal);
835     right->internal_tree.erase_address(to_steal_address);
836     move_from_second_to_first(right);
837 }
838
839 // MERGE
840 private:
841
842 void merge(abstract_node* left, abstract_node* right, const unsigned
level)
843 {
844     if (level != 0)
845     {
846         merge(
847             static_cast<inner_node*>(left),
848             static_cast<inner_node*>(right));
849     }
850     else
851     {
852         merge(
853             static_cast<leaf_node*>(left),
854             static_cast<leaf_node*>(right));
855     }
856 }
857
858 void merge(inner_node* left, inner_node* right)
859 {
860     // preconditions:
861     assert(is_small(left));
862     assert(is_small(right));
863
864     left->second = right;
865 }
866
867 void merge(leaf_node* left, leaf_node* right)

```

```

868         {
869             // preconditions:
870             assert(is_small(left));
871             assert(is_small(right));
872
873             left->second = right;
874         }
875
876         // CREATE SECOND IF NEEDED
877 private:
878
879         inner_node* create_second_if_needed(inner_node* node)
880         {
881             if (node->second == nullptr)
882             {
883                 node->second = allocate_inner();
884                 new (node->second) inner_node();
885             }
886             return node->second;
887         }
888
889         leaf_node* create_second_if_needed(leaf_node* node)
890         {
891             if (node->second == nullptr)
892             {
893                 node->second = allocate_leaf();
894                 new (node->second) leaf_node();
895             }
896             return node->second;
897         }
898
899         // DELETE SECOND IF EMPTY
900 private:
901
902         void delete_second_if_empty(inner_node* node)
903         {
904             if (node->second != nullptr and node->second->internal_tree.empty
905                 ())
906             {
907                 deallocate_inner(node->second);
908                 node->second = nullptr;
909             }
910
911         void delete_second_if_empty(leaf_node* node)
912         {
913             if (node->second != nullptr and node->second->internal_tree.empty
914                 ())
915             {
916                 deallocate_leaf(node->second);
917                 node->second = nullptr;
918             }
919
920         // MOVE FROM FIRST TO SECOND
921 private:
922
923         void move_from_first_to_second(inner_node* node)
924         {
925             create_second_if_needed(node);
926
927             auto max_address = node->internal_tree.max_address();
928             auto & max = node->internal_tree.pool[max_address].value;
929             node->second->internal_tree.insert_min(max);
930             node->internal_tree.erase_address(max_address);

```

```

931     }
932
933 void move_from_first_to_second(leaf_node* node)
934 {
935     create_second_if_needed(node);
936
937     auto max_address = node->internal_tree.max_address();
938     auto & max = node->internal_tree.pool[max_address].value;
939     node->second->internal_tree.insert_min(max);
940     node->internal_tree.erase_address(max_address);
941 }
942
943 // MOVE FROM SECOND TO FIRST
944 private:
945
946 void move_from_second_to_first(inner_node* node)
947 {
948     if (node->second != nullptr)
949     {
950         auto min_address = node->second->internal_tree.
951             min_address();
952         auto & min = node->second->internal_tree.pool[min_address
953             ].value;
954         node->internal_tree.insert_max(min);
955         node->second->internal_tree.erase_address(min_address);
956
957         delete_second_if_empty(node);
958     }
959
960 void move_from_second_to_first(leaf_node* node)
961 {
962     if (node->second != nullptr)
963     {
964         auto min_address = node->second->internal_tree.
965             min_address();
966         auto & min = node->second->internal_tree.pool[min_address
967             ].value;
968         node->internal_tree.insert_max(min);
969         node->second->internal_tree.erase_address(min_address);
970
971         delete_second_if_empty(node);
972     }
973 }
974
975 // IS FULL?
976 private:
977
978 bool is_full(const abstract_node* node, const unsigned level) const
979 {
980     if (level != 0)
981         { return is_full(static_cast<const inner_node*>(node)); }
982     else
983         { return is_full(static_cast<const leaf_node*>(node)); }
984 }
985
986 bool is_full(const inner_node* node) const
987 {
988     return node->second != nullptr and node->second->internal_tree.
989         full();
990 }
991
992 bool is_full(const leaf_node* node) const
993 {
994     return node->second != nullptr and node->second->internal_tree.

```

```

    full();
991     }
992
993     // IS SMALL?
994 private:
995
996     bool is_small(const abstract_node* node, const unsigned level) const
997     {
998         if (level != 0)
999             { return is_small(static_cast<const inner_node*>(node)); }
1000
1001         else
1002             { return is_small(static_cast<const leaf_node*>(node)); }
1003
1004     bool is_small(const inner_node* node) const
1005     {
1006         return node->second == nullptr;
1007     }
1008
1009     bool is_small(const leaf_node* node) const
1010     {
1011         return node->second == nullptr;
1012     }
1013
1014     // MIN KEY
1015 private:
1016
1017     key_type min_key(const abstract_node* node, const unsigned level) const
1018     {
1019         if (level != 0)
1020             { return min_key(static_cast<const inner_node*>(node)); }
1021
1022         else
1023             { return min_key(static_cast<const leaf_node*>(node)); }
1024
1025     key_type min_key(const inner_node* node) const
1026     {
1027         return node->internal_tree.pool[
1028             node->internal_tree.min_address()].value.first;
1029     }
1030
1031     key_type min_key(const leaf_node* node) const
1032     {
1033         return node->internal_tree.pool[
1034             node->internal_tree.min_address()].value.first;
1035     }
1036
1037     // MAX KEY
1038 private:
1039
1040     key_type max_key(const abstract_node* node, const unsigned level) const
1041     {
1042         if (level != 0)
1043             { return max_key(static_cast<inner_node*>(node)); }
1044
1045         else
1046             { return max_key(static_cast<leaf_node*>(node)); }
1047
1048     key_type max_key(const inner_node* node) const
1049     {
1050         return node->internal_tree.pool[
1051             node->internal_tree.max_address()].value.first;
1052     }
1053

```

```

1054     key_type max_key(const leaf_node* node) const
1055     {
1056         return node->internal_tree.pool[
1057             node->internal_tree.max_address()].value.first;
1058     }
1059 };

```

Red-black tree

§ 2 *internal_rbtrees.hpp*

```

1  #pragma once
2  #include <cassert>
3  #include <utility>
4  #include "pool.hpp"
5
6  #include <iostream>
7
8  template<typename K, typename M, typename address_type, size_t size,
9         class compare_function = std::less<K>>
10 class internal_rbtrees
11 {
12 public:
13     // TYPEDEFS
14     typedef K key_type;
15     typedef M mapped_type;
16     typedef std::pair<const key_type, mapped_type> value_type;
17
18     typedef address_type size_type;
19
20 private:
21     // NODE STRUCTURE
22
23     // a type representing the color of a node
24     enum class color_type : uint8_t { red, black };
25
26     // the node structure
27     struct node
28     {
29         node(
30             value_type value_,
31             address_type parent_,
32             address_type left_,
33             address_type right_,
34             color_type color_)
35             : value(value_),
36               parent(parent_),
37               left(left_),
38               right(right_),
39               color(color_)
40             {}
41
42         value_type value;
43         address_type parent;
44         address_type left;
45         address_type right;
46         color_type color;
47     };
48
49     // DATA MEMBERS
50
51 public:
52     pool<node, address_type, size - 4*sizeof(address_type)> pool;
53

```

```

54 private:
55     address_type root;        // address of the root of the tree
56     address_type nil;        // address of the nil node
57     address_type min;        // address of the minimum element
58     address_type max;        // address of the maximum element
59
60     // CONST HELPERS
61
62     address_type tree_minimum(address_type x) const
63     {
64         while (pool[x].left != nil)
65             { x = pool[x].left; }
66         return x;
67     }
68
69     address_type tree_maximum(address_type x) const
70     {
71         while (pool[x].right != nil)
72             { x = pool[x].right; }
73         return x;
74     }
75
76 public:
77     address_type successor(address_type x) const
78     {
79         if (pool[x].right != nil)
80             { return tree_minimum(pool[x].right); }
81
82         address_type y = pool[x].parent;
83
84         while (y != nil and x == pool[y].right)
85             {
86                 x = y;
87                 y = pool[y].parent;
88             }
89         return y;
90     }
91
92     address_type predecessor(address_type x) const
93     {
94         if (pool[x].left != nil)
95             { return tree_maximum(pool[x].left); }
96
97         address_type y = pool[x].parent;
98
99         while (y != nil and x == pool[y].left)
100             {
101                 x = y;
102                 y = pool[y].parent;
103             }
104         return y;
105     }
106
107 private:
108     // MUTATING HELPERS
109
110     void left_rotate(address_type x)
111     {
112         address_type y = pool[x].right;
113
114         pool[x].right = pool[y].left;
115         if (pool[y].left != nil)
116             { pool[pool[y].left].parent = x; }
117
118         pool[y].parent = pool[x].parent;

```

```

119         if (pool[x].parent == nil)
120             { root = y; }
121         else if (x == pool[pool[x].parent].left)
122             { pool[pool[x].parent].left = y; }
123         else
124             { pool[pool[x].parent].right = y; }
125
126         pool[y].left = x;
127         pool[x].parent = y;
128     }
129
130 void right_rotate(address_type y)
131 {
132     address_type x = pool[y].left;
133
134     pool[y].left = pool[x].right;
135     if (pool[x].right != nil)
136         { pool[pool[x].right].parent = y; }
137
138     pool[x].parent = pool[y].parent;
139     if (pool[y].parent == nil)
140         { root = x; }
141     else if (y == pool[pool[y].parent].left)
142         { pool[pool[y].parent].left = x; }
143     else
144         { pool[pool[y].parent].right = x; }
145
146     pool[x].right = y;
147     pool[y].parent = x;
148 }
149
150 void transplant(address_type u, address_type v)
151 {
152     if (pool[u].parent == nil)
153         { root = v; }
154     else if (u == pool[pool[u].parent].left)
155         { pool[pool[u].parent].left = v; }
156     else
157         { pool[pool[u].parent].right = v; }
158
159     pool[v].parent = pool[u].parent;
160 }
161
162 // INSERT FIXUP
163
164 void insert_fixup(address_type z)
165 {
166     while (pool[pool[z].parent].color == color_type::red)
167     {
168         if (pool[z].parent ==
169             pool[pool[pool[z].parent].parent].left)
170             {
171                 address_type y = pool[pool[pool[z].parent].parent
172                     ].right;
173
174                 if (y != nil and pool[y].color == color_type::red
175                     )
176                     {
177                         pool[pool[z].parent].color = color_type::
178                             black;
179                         pool[y].color = color_type::black;
180                         pool[pool[pool[z].parent].parent].color =
181                             color_type::red;
182                         z = pool[pool[z].parent].parent;
183                     }
184             }
185     }

```

```

180         else
181             {
182                 if (z == pool[pool[z].parent].right)
183                     {
184                         z = pool[z].parent;
185                         left_rotate(z);
186                     }
187                 pool[pool[z].parent].color = color_type::
188                     black;
189                 pool[pool[pool[z].parent].parent].color =
190                     color_type::red;
191                 right_rotate(pool[pool[z].parent].parent)
192                 ;
193             }
194         else
195             {
196                 address_type y = pool[pool[pool[z].parent].parent
197                     ].left;
198                 if (y != nil and pool[y].color == color_type::red
199                     )
200                     {
201                         pool[pool[z].parent].color = color_type::
202                             black;
203                         pool[y].color = color_type::black;
204                         pool[pool[pool[z].parent].parent].color =
205                             color_type::red;
206                         z = pool[pool[z].parent].parent;
207                     }
208                 else
209                     {
210                         if (z == pool[pool[z].parent].left)
211                             {
212                                 z = pool[z].parent;
213                                 right_rotate(z);
214                             }
215                         pool[pool[z].parent].color = color_type::
216                             black;
217                         pool[pool[pool[z].parent].parent].color =
218                             color_type::red;
219                         left_rotate(pool[pool[z].parent].parent);
220                     }
221             }
222         pool[root].color = color_type::black;
223     }
224 // ERASE ADDRESS
225 public:
226 void erase_address(address_type z)
227     {
228         // maintain minimum and maximum addresses
229         if (min == z)
230             { min = successor(z); }
231         if (max == z)
232             { max = predecessor(z); }
233         // erase the value from the tree
234         address_type x = nil;
235         address_type y = z;
236         color_type y_original_color = pool[y].color;

```



```

236     if (pool[z].left == nil)
237     {
238         x = pool[z].right;
239         transplant(z, pool[z].right);
240     }
241     else if (pool[z].right == nil)
242     {
243         x = pool[z].left;
244         transplant(z, pool[z].left);
245     }
246     else
247     {
248         y = tree_minimum(pool[z].right);
249         y_original_color = pool[y].color;
250         x = pool[y].right;
251
252         if (pool[y].parent == z)
253         {
254             pool[x].parent = y;
255         }
256         else
257         {
258             transplant(y, pool[y].right);
259             pool[y].right = pool[z].right;
260             pool[pool[y].right].parent = y;
261         }
262
263         transplant(z, y);
264         pool[y].left = pool[z].left;
265         pool[pool[y].left].parent = y;
266         pool[y].color = pool[z].color;
267     }
268
269     // deallocate
270     pool.deallocate(z);
271
272     // fixup if necessary
273     if (y_original_color == color_type::black)
274     { erase_fixup(x); }
275 }
276
277 // ERASE FIXUP
278 private:
279 void erase_fixup(address_type x)
280 {
281     while (x != root and pool[x].color == color_type::black)
282     {
283         if (x == pool[pool[x].parent].left)
284         {
285             address_type w = pool[pool[x].parent].right;
286
287             if (pool[w].color == color_type::red)
288             {
289                 pool[w].color = color_type::black;
290                 pool[pool[x].parent].color = color_type::
291                     red;
292                 left_rotate(pool[x].parent);
293                 w = pool[pool[x].parent].right;
294             }
295
296             if (pool[pool[w].left].color == color_type::black
297                 and
298                 pool[pool[w].right].color == color_type::
299                     black)
300             {

```

```

298         pool[w].color = color_type::red;
299         x = pool[x].parent;
300     }
301     else
302     {
303         if (pool[pool[w].right].color ==
304             color_type::black)
305         {
306             pool[pool[w].left].color =
307                 color_type::black;
308             pool[w].color = color_type::red;
309             right_rotate(w);
310             w = pool[pool[x].parent].right;
311         }
312         pool[w].color = pool[pool[x].parent].
313             color;
314         pool[pool[x].parent].color = color_type::
315             black;
316         pool[pool[w].right].color = color_type::
317             black;
318         left_rotate(pool[x].parent);
319         x = root;
320     }
321     else
322     {
323         address_type w = pool[pool[x].parent].left;
324         if (pool[w].color == color_type::red)
325         {
326             pool[w].color = color_type::black;
327             pool[pool[x].parent].color = color_type::
328                 red;
329             right_rotate(pool[x].parent);
330             w = pool[pool[x].parent].left;
331         }
332         if (pool[pool[w].left].color == color_type::black
333             and
334             pool[pool[w].right].color == color_type::
335                 black)
336         {
337             pool[w].color = color_type::red;
338             x = pool[x].parent;
339         }
340         else
341         {
342             if (pool[pool[w].left].color ==
343                 color_type::black)
344             {
345                 pool[pool[w].right].color =
346                     color_type::black;
347                 pool[w].color = color_type::red;
348                 left_rotate(w);
349                 w = pool[pool[x].parent].left;
350             }
351             pool[w].color = pool[pool[x].parent].
352                 color;
353             pool[pool[x].parent].color = color_type::
354                 black;
355             pool[pool[w].left].color = color_type::
356                 black;
357             right_rotate(pool[x].parent);

```

```

350                                     x = root;
351                                     }
352                                 }
353                             }
354
355                             pool[x].color = color_type::black;
356                         }
357
358                     // CONSTRUCTORS ET CETERA
359
360                     // copy protection
361                     internal_rbtree(internal_rbtree &);
362                     void operator=(internal_rbtree &);
363
364 public:
365
366     internal_rbtree()
367     {
368         // create the sentinel node, nil
369         // (parent, left and right are arbitrarily selected to point to
370         // itself)
371         nil = pool.allocate();
372         new(&pool[nil]) node(
373             std::make_pair(key_type(), mapped_type()),
374             nil, nil, nil, color_type::black);
375
376         // root, min and max starts out as nil
377         root = nil;
378         min = nil;
379         max = nil;
380
381         assert(not full());
382     }
383
384     ~internal_rbtree()
385     {}
386
387     // ATTRIBUTES
388
389     // is the tree empty?
390     bool empty() const
391     { return root == nil; }
392
393     // is the tree full?
394     bool full() const
395     { return pool.full(); }
396
397     // LOOKUP
398
399     // returns the address to the node with the greatest key less than or
400     // equal
401     // to the one given, or, if all keys are larger than the one given,
402     // returns
403     // the node with the minimum key.
404     // this behaviour is of specific use for our btree - it defines which
405     // children are selected for which keys, in the inner nodes.
406     address_type lookup(key_type key) const
407     {
408         address_type x = root;
409         address_type y = root;
410         address_type z = nil;
411
412         while (x != nil)
413         {

```

```

411         y = x;
412         if (pool[x].value.first > key)
413             {
414                 x = pool[x].left;
415             }
416         else
417             {
418                 z = x;
419                 x = pool[x].right;
420             }
421     }
422
423     return (z != nil) ? z : y;
424 }
425
426 // INSERT
427
428 // inserts a value into the tree
429 void insert(value_type value)
430 {
431     // find parent, and check if we just need to replace
432     address_type y = nil;
433     address_type x = root;
434     while (x != nil)
435     {
436         y = x;
437         x = (compare_function()(value.first, pool[x].value.first)
438             ? pool[x].left : pool[x].right;
439
440         if (value.first == pool[x].value.first)
441             {
442                 // key is already here, replace value and return
443                 pool[x].value.second = value.second;
444                 return;
445             }
446     }
447
448     // if we couldn't just update we need a new node, and must not be
449     // full.
450     // this needs to be checked by the caller
451     assert(not full());
452
453     // allocate a new node and fill in what we know
454     address_type z = pool.allocate();
455     new(&pool[z]) node(value, y, nil, nil, color_type::red);
456
457     // hook it up to the tree
458     if (y == nil)
459     {
460         root = z;
461         // the tree is empty so the new value is both min and max
462         min = z;
463         max = z;
464     }
465     else if (compare_function()(pool[z].value.first, pool[y].value.
466         first))
467     {
468         pool[y].left = z;
469         // maintain minimum address
470         if (min == y) { min = z; }
471     }
472     else
473     {
474         pool[y].right = z;

```

```

473             // maintain maximum address
474             if (max == y) { max = z; }
475             }
476
477             // fixup
478             insert_fixup(z);
479             }
480
481 // use only if it is known that the inserted value is larger than any of
482 // the
483 // existing
484 void insert_max(value_type value)
485 {
486     // if we couldn't just update we need a new node, and must not be
487     // full.
488     // this needs to be checked by the caller
489     assert(not full());
490
491     // allocate a new node and fill in what we know
492     address_type z = pool.allocate();
493     new(&pool[z]) node(value, max, nil, nil, color_type::red);
494
495     // hook it up to the tree
496     if (max == nil)
497     {
498         root = z;
499         // the tree is empty so the new value is both min and max
500         min = z;
501     }
502     else
503     {
504         pool[max].right = z;
505     }
506
507     max = z;
508
509     insert_fixup(z);
510 }
511
512 // use only if it is known that the inserted value is smaller than any of
513 // the existing
514 void insert_min(value_type value)
515 {
516     // if we couldn't just update we need a new node, and must not be
517     // full.
518     // this needs to be checked by the caller
519     assert(not full());
520
521     // allocate a new node and fill in what we know
522     address_type z = pool.allocate();
523     new(&pool[z]) node(value, min, nil, nil, color_type::red);
524
525     // hook it up to the tree
526     if (min == nil)
527     {
528         root = z;
529         // the tree is empty so the new value is both min and max
530         max = z;
531     }
532     else
533     {
534         pool[min].left = z;
535     }
536
537     min = z;

```

```

535
536         insert_fixup(z);
537     }
538
539     // ERASE
540
541     void erase(key_type key)
542     {
543         // find the value to be erased, if it exists
544         address_type to_erase = root;
545         while (to_erase != nil and pool[to_erase].value.first != key)
546             {
547                 to_erase = (compare_function()(key, pool[to_erase].value.
548                             first))
549                             ? pool[to_erase].left
550                             : pool[to_erase].right;
551             }
552
553         if (to_erase != nil)
554             { erase_address(to_erase); }
555
556     // KEY CHANGING
557
558     // changes the key of an element, without checking!
559     // this should be done with extreme care from the callers side, and *only
560     // when the caller knows that it does not change the ordering!
561     void change_key_of_element(address_type address, key_type new_key)
562     {
563         *const_cast<key_type*>(&pool[address].value.first) = new_key;
564     }
565
566     // MIN/MAX
567
568     address_type min_address() const
569     { return min; }
570
571     address_type max_address() const
572     { return max; }
573 };

```

Pool manager

§ 3 *pool.hpp*

```

1  #pragma once
2  #include <cassert>
3  #include <cstdlib>
4
5  template<typename value_type, typename address_type, size_t size>
6  class pool
7  {
8  private:
9      // CONSTANTS
10
11     static constexpr size_t max_size(size_t a, size_t b)
12     { return a > b ? a : b; }
13
14     // the size of an element. each element holds either a value or an
15     // address, so we use the size of the largest of those types
16     static constexpr size_t element_size =
17         max_size(sizeof(value_type), sizeof(address_type));
18

```

```

19 // the total size of the allocated memory. two addresses are subtracted
20 // for bookkeeping, these are 'head' and 'back' - see below
21 static constexpr size_t memory_size = size - 2 * sizeof(address_type);
22
23 // DATA MEMBERS
24
25 int8_t memory[memory_size];
26 address_type head; // the next element to be allocated
27 address_type back; // the first element that has never been allocated
28
29 // HELPERS
30
31 // these access an element as either a value_type or an address_type
32 inline value_type & value_at(address_type address)
33     { return *reinterpret_cast<value_type*>(memory + address); }
34
35 inline const value_type & value_at(address_type address) const
36     { return *reinterpret_cast<const value_type*>(memory + address);
37     }
38
39 inline address_type & address_at(address_type address)
40     { return *reinterpret_cast<address_type*>(memory + address); }
41
42 inline const address_type & address_at(address_type address) const
43     { return *reinterpret_cast<const address_type*>(memory + address)
44     ; }
45
46 // CONSTRUCTORS ET CETERA
47
48 // copy protection
49 pool(pool &);
50 void operator=(pool &);
51
52 public:
53 pool() : head(0), back(0) {}
54
55 ~pool() {}
56
57 // FULL
58
59 bool full() const
60     { return (memory_size - head) < element_size; }
61
62 // ALLOCATE
63
64 // get memory for one value from the pool
65 address_type allocate()
66     {
67     // precondition: pool is not full. must be checked by caller!
68     assert(not full());
69
70     // get the address to return
71     address_type address = head;
72
73     head = (address < back)
74     // update 'head'. if the address we got is less than 'back' it
75     // has been
76     // previously allocated and we can follow the address stored at
77     // it to
78     // get the next head
79     ? address_at(address)
80     // otherwise we are at 'back', which has not been previously
81     // allocated.
82     // thus, all the higher addresses are free, and we can just point
83     // to

```

```
78         // the next element
79         : back += element_size;
80
81         return address;
82     }
83
84     // DEALLOCATE
85
86     // return memory for one value to the pool
87     void deallocate(address_type address)
88     {
89         // store the current head at the position of this element and
90         // make
91         // this the new head
92         address_at(address) = head;
93         head = address;
94     }
95
96     // ACCESS
97
98     // access memory at address
99     value_type & operator[](address_type address)
100    {
101        assert(address <= memory_size - element_size);
102        return value_at(address);
103    }
104
105    const value_type & operator[](address_type address) const
106    {
107        assert(address <= memory_size - element_size);
108        return value_at(address);
109    }
110    };
```