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

CPH STL Report 2013-2, June 2013.

Copyright notice

Release date

2013-06-03

Included files

B^+-tree

§ 1 *improved_btree.hpp*

```
#pragma once
#include <cassert>
#include <cstdint>
#include <memory>
#include <utility>
#include <iostream>
#include ”internal_rbtree.hpp”

// this class is never instantiated. it has one use: as the template parameter
// to an allocator. this way we can use one allocator for both node types
template<int size>
struct alignas(size) allocated_node
        {
        uint8_t padding[size];
        };

// template parameters:
// K: key type
// M: mapped type
// node_size: size of the nodes - should be tuned to page size
// address_type: type used for addresses - typically uint16_t or uint32_t
// compare_function: compare function, defaults to std::less
// allocator_type: a template taking a type, used as allocator
template<
        typename K,
        typename M,
        size_t node_size = 512,
        typename address_type = uint16_t,
        class compare_function = std::less<K>,
        class allocator_type = std::allocator<allocated_node<node_size>>>
class improved_btree
        {
        // TYPEDEFS
public:

        typedef K key_type;
        typedef M mapped_type;
        typedef std::pair<const K, M> value_type;

        // NODE DEFINITIONS
private:

        struct alignas(node_size) abstract_node {};

        struct alignas(node_size) inner_node : public abstract_node
                {
                typedef internal_rbtree<key_type, abstract_node*, address_type,
                        node_size - sizeof(inner_node*), compare_function>
                            tree_type;

                tree_type internal_tree;
                inner_node* second;

                inner_node(): second(nullptr) {}
                };

        struct alignas(node_size) leaf_node : public abstract_node
                {
                typedef internal_rbtree<key_type, mapped_type, address_type,
                        node_size - sizeof(leaf_node*), compare_function>
```

```
                             tree_type;

                tree_type internal_tree;
                leaf_node* second;

                leaf_node(): second(nullptr) {}
                };

        // ITERATOR TYPE
public:

        class iterator
                {
        private:

                friend improved_btree;

                iterator(const leaf_node* node_, address_type address_)
                :       node(node_), address(address_)
                        {}

        public:
                iterator()
                :       node(nullptr), address()
                        {}

                const value_type& operator*()
                        {
                        assert(node != nullptr);
                        return node->internal_tree.pool[address].value;
                        }

                const value_type* operator->()
                        {
                        assert(node != nullptr);
                        return &node->internal_tree.pool[address].value;
                        }

                // TODO: increment and decrement

        private:
                const leaf_node* node;
                address_type address;
                };

        // DATA MEMBERS
private:

        abstract_node* root;
        unsigned height;

        allocator_type allocator;

        // ALLOCATOR WRAPPERS
private:

        inner_node* allocate_inner()
                { return (inner_node*)(allocator.allocate(1)); }

        leaf_node* allocate_leaf()
                { return (leaf_node*)(allocator.allocate(1)); }

        void deallocate_inner(inner_node* node)
                { allocator.deallocate((allocated_node<node_size>*)node, 1); }

```

```
        void deallocate_leaf(leaf_node∗ node)
                { allocator.deallocate((allocated_node<node_size>∗)node, 1); }

        // CONSTRUCTORS ET CETERA
public:

        improved_btree()
        :       height(0)
                {
                root = allocate_leaf();
                new (root) leaf_node();
                }

        ∼improved_btree()
                {
                destruct(root, height);
                }

        // TODO: copying
        improved_btree(improved_btree&) = delete;
        void operator=(improved_btree&) = delete;

private:

        void destruct(abstract_node∗ node, const unsigned level)
                {
                if (level != 0)
                        { destruct(static_cast<inner_node∗>(node), level); }
                else
                        { destruct(static_cast<leaf_node∗>(node)); }
                }

        void destruct(inner_node∗ node, const unsigned level)
                {
                if (node→second != nullptr)
                        {
                        auto x = node→second→internal_tree.min_address();
                        while (x != node→second→internal_tree.max_address())
                                {
                                destruct(node→second→internal_tree.pool[x].
                                     value.second,
                                        level−1);
                                x = node→second→internal_tree.successor(x);
                                }
                        destruct(node→second→internal_tree.pool[x].value.second
                             ,
                                level−1);

                        deallocate_inner(node→second);
                        }

                auto x = node→internal_tree.min_address();
                while (x != node→internal_tree.max_address())
                        {
                        destruct(node→internal_tree.pool[x].value.second,
                                level−1);
                        x = node→internal_tree.successor(x);
                        }
                destruct(node→internal_tree.pool[x].value.second,
                        level−1);

                deallocate_inner(node);
                }

        void destruct(leaf_node∗ node)
```

```
        {
        if (node->second != nullptr)
            { deallocate_leaf(node->second); }

        deallocate_leaf(node);
        }

    // FIND
public:

    // returns the largest value that is less than or equal to the key given,
    // or, if all values are larger than the key, returns the minimum value
    // todo: implement iterators
    iterator find(const key_type& k) const
        {
        unsigned level = height;
        const abstract_node* current_node = root;

        // search inner nodes
        while (level != 0)
            {
            const inner_node* first = static_cast<const inner_node*>(
                current_node);
            const inner_node* second = first->second;

            if (second != nullptr and
                not compare_function()(k, min_key(second)))
                {
                current_node =
                    second->internal_tree.pool[
                        second->internal_tree.lookup(k)].
                            value.second;
                }
            else
                {
                current_node =
                    first->internal_tree.pool[
                        first->internal_tree.lookup(k)].
                            value.second;
                }

            --level;
            }

        // get the result from the leaf
        const leaf_node* first = static_cast<const leaf_node*>(
            current_node);
        const leaf_node* second = first->second;

        if (second != nullptr and not compare_function()(k, min_key(
            second)))
            {
            return iterator(second, second->internal_tree.lookup(k));
            }
        else
            {
            return iterator(first, first->internal_tree.lookup(k));
            }
        }

    // INSERT
public:

    void insert(const value_type& v)
        {
```

```
        // if the root is full, split it and make a new one
        if (is_full(root, height))
            { split_root(); }

        unsigned level = height;
        abstract_node∗ current_node = root;

        // search inner nodes
        while (level != 0)
            {
            inner_node∗ first = static_cast<inner_node∗>(current_node);
            inner_node∗ second = first→second;

            current_node = insert_inner(first, second, level, v);
            −−level;
            }

        // insert to the leaf
        leaf_node∗ first = static_cast<leaf_node∗>(current_node);
        leaf_node∗ second = first→second;

        return insert_leaf(first, second, v);
        }

private:

    abstract_node∗ insert_inner(inner_node∗ first, inner_node∗ second,
        const unsigned level, const value_type& v)
        {
        // preconditions:
        assert(not is_full(first));

        // second page exists and the value belongs in it
        if (second != nullptr and
            not compare_function()(v.first, min_key(second)))
            {
            // find the child to insert to
            auto insert_to_address = second→internal_tree.lookup(v.first);

            abstract_node∗ insert_to = second→internal_tree.pool[
                    insert_to_address].value.second;

            // is it full?
            if (is_full(insert_to, level−1))
                {
                // split it
                abstract_node∗ splitted = split(insert_to, level−1);
                auto min_key_splitted = min_key(splitted, level−1);

                // create an element representing the new node
                second→internal_tree.insert(
                    {min_key_splitted, splitted});

                // decide which of the resulting nodes to insert to
                insert_to = not compare_function()(v.first, min_key_splitted)
                    ? splitted : insert_to;
                }

            return insert_to;
```

```
        }
// second page doesn't exist or the value doesn't belong in it
else
        {
        // find the child to insert to
        auto insert_to_address = first→internal_tree.lookup(v.first);

        abstract_node* insert_to = first→internal_tree.pool[
                insert_to_address].value.second;

        // is it full?
        if (is_full(insert_to, level−1))
                {
                // split it
                abstract_node* splitted = split(insert_to, level−1);
                auto min_key_splitted = min_key(splitted, level−1);

                // find space for an element representing the new node
                // is this page full?
                if (first→internal_tree.full())
                        {
                        second = create_second_if_needed(first);

                        // if the child is the maximum, the result of the split
                        // will go into the second node
                        if (insert_to_address == first→internal_tree.max_address())
                                {
                                second→internal_tree.insert_min({
                                        min_key_splitted, splitted});
                                }
                        // otherwise, the maximum will go into the second node
                        else
                                {
        move_from_first_to_second(first);

                                first→internal_tree.insert(
                                        {min_key_splitted, splitted});
                                }
                        }
                // this node isn't full
                else
                        {
                        first→internal_tree.insert(
                                {min_key_splitted, splitted});
                        }

                // decide which of the resulting nodes to insert to
                insert_to = not compare_function()(v.first, min_key_splitted)
                        ? splitted : insert_to;
                }

        // because any keys less than the leftmost key are inserted into
```

```
            // the leftmost node - the node corresponding to the leftmost key
            // (we do not have an extra node like in a regular btree) - we
            // might need to update the key of the node we are inserting to.
            // we can do it in this way because we know that the ordering does
            // not change
            if (compare_function()(v.first, first->internal_tree.pool[
                insert_to_address].value.first))
                {
                first->internal_tree.change_key_of_element(
                    insert_to_address, v.first);
                }

            return insert_to;
            }
        }

    void insert_leaf(leaf_node* first, leaf_node* second,
        const value_type& v)
        {
        // preconditions:
        assert(not is_full(first));

        // second node exists and the value belongs in it
        if (second != nullptr and
            not compare_function()(v.first, min_key(second)))
            {
            // simple case: just insert
            second->internal_tree.insert(v);
            }
        // second node doesn't exist or the value doesn't belong in it
        else
            {
            // is this node full?
            if (first->internal_tree.full())
                {
                // slightly more complicated case:

                // one of two values have to go into the second node: either the
                // new value or the current maximum, depending on which will be
                // the new maximum
                auto max_key_first = max_key(first);
                if (compare_function()(max_key_first, v.first))
                    {
                    // if the second node doesn't exist, we need to allocate it
                    second = create_second_if_needed(first);

                    // the new value is larger and is placed in the second node
                    second->internal_tree.insert(v);
                    return;
                    }
                else if (compare_function()(v.first, max_key_first))
                    {
                    move_from_first_to_second(first);
                    // and the new value is placed in this node
```

```
                                        }
                                // edge case: the new value has the same key as
                                     the current
                                // maximum. in this case we just replace, which
                                     the
                                // internal_tree insert function handles
                                }
                        // this node isn't full
                        // simple case: just insert
                        first→internal_tree.insert(v);
                        }
                }

        // ERASE
public:

        void erase(const key_type& k)
                {
                // check for collapse: root is inner node and has only one
                     element
                // (that is, minimum is the same as maximum)
                if (height > 0 and
                        static_cast<inner_node*>(root)→internal_tree.min_address
                             () ==
                        static_cast<inner_node*>(root)→internal_tree.max_address
                             ())
                        {
                        inner_node* root_inner = static_cast<inner_node*>(root);
                        root = root_inner→internal_tree.pool[
                                root_inner→internal_tree.min_address()].value.
                                     second;
                        −−height;
                        }

                unsigned level = height;
                abstract_node* current_node = root;

                // search inner nodes
                while (level != 0)
                        {
                        inner_node* first = static_cast<inner_node*>(current_node
                             );
                        inner_node* second = first→second;

                        current_node = erase_inner(first, second, level, k);
                        level −= 1;
                        }

                // erase from the leaf
                leaf_node* first = static_cast<leaf_node*>(current_node);
                leaf_node* second = first→second;

                return erase_leaf(first, second, k);
                }

private:

        abstract_node* erase_inner(inner_node* first, inner_node* second,
                const unsigned level, const key_type& k)
                {
                // preconditions:
                assert(first == root or not is_small(first));

                address_type erase_from_address;
                abstract_node* erase_from;
```

```

// addresses of adjacent nodes
bool has_left = false;
address_type adjacent_left_address;
abstract_node* adjacent_left;

bool has_right = false;
address_type adjacent_right_address;
abstract_node* adjacent_right;

// find the node to erase from
if (second != nullptr and not compare_function()(k, min_key(second)))
	{
	erase_from_address = second→internal_tree.lookup(k);
	erase_from = second→internal_tree.pool[
		erase_from_address].value.second;

	if (is_small(erase_from, level-1))
		{
		if (erase_from_address == second→internal_tree.min_address())
			{
			adjacent_left_address = first→internal_tree.max_address();
			adjacent_left = first→internal_tree.pool[
				adjacent_left_address].value.second;
			}
		else
			{
			adjacent_left_address =
				second→internal_tree.predecessor(erase_from_address);
			adjacent_left = second→internal_tree.pool[
				adjacent_left_address].value.second;
			}

		if (erase_from_address != second→internal_tree.max_address())
			{
			has_right = true;
			adjacent_right_address =
				second→internal_tree.successor(erase_from_address);
			adjacent_right = second→internal_tree.pool[
				adjacent_right_address].value.second;
			}

		// can we steal from the node to the left?
		if (not is_small(adjacent_left, level-1))
			{
			steal_from_left_to_right(
				adjacent_left, erase_from, level-1);

			// update the key of the node we stole to. we can do it like
			// this because we know that the ordering has not changed
```

```
                    second->internal_tree.change_key_of_element(
                        erase_from_address, min_key(erase_from, level-1));
                    }
                // can we steal from the node to the right?
                else if (has_right and not is_small(
                        adjacent_right, level-1))
                    {
                    steal_from_right_to_left(
                        erase_from, adjacent_right, level-1);

                    // update the key of the node we stole from. we can do it
                    // like this because we know that the ordering has not
                    // changed
                    second->internal_tree.change_key_of_element(
                        adjacent_right_address, min_key(
                            adjacent_right, level-1));
                    }
                // merge with the node to the left
                else
                    {
                    merge(adjacent_left, erase_from, level-1);

                    second->internal_tree.erase_address(erase_from_address);
                    delete_second_if_empty(first);

                    erase_from = adjacent_left;
                    }
                }
            return erase_from;
            }
        else
            {
            erase_from_address = first->internal_tree.lookup(k);
            erase_from = first->internal_tree.pool[
                erase_from_address].value.second;

            if (is_small(erase_from, level-1))
                {
                bool adjacent_right_from_second;

                if (erase_from_address != first->internal_tree.min_address())
                    {
                    has_left = true;
                    adjacent_left_address =
                        first->internal_tree.predecessor(erase_from_address);
                    adjacent_left = first->internal_tree.pool[
                        adjacent_left_address].value.second;
                    }

                if (second != nullptr and
                    erase_from_address == first->internal_tree.max_address())
```

```
        {
        has_right = true;
        adjacent_right_address =
                second→internal_tree.min_address();
        adjacent_right = second→internal_tree.pool[
                adjacent_right_address].value.second;
        adjacent_right_from_second = true;
        }
else if (erase_from_address !=
        first→internal_tree.max_address())
        {
        has_right = true;
        adjacent_right_address =
                first→internal_tree.successor(erase_from_address);
        adjacent_right = first→internal_tree.pool[
                adjacent_right_address].value.second;
        adjacent_right_from_second = false;
        }

// can we steal from the node to the left?
if (has_left and not is_small(adjacent_left, level−1))
        {
        steal_from_left_to_right(
                adjacent_left, erase_from, level−1);

        // update the key of the node we stole to. we can do it like
        // this because we know that the ordering has not changed
        first→internal_tree.change_key_of_element(
                erase_from_address, min_key(erase_from, level−1));
        }
// can we steal from the node to the right?
else if (has_right and not is_small(
                adjacent_right, level−1))
        {
        steal_from_right_to_left(
                erase_from, adjacent_right, level−1);

        // update the key of the node we stole from. we can do it
        // like this because we know that the ordering has not
        // changed
        if (adjacent_right_from_second)
                {
                second→internal_tree.change_key_of_element(
                        adjacent_right_address,
                        min_key(adjacent_right, level−1));
                }
        else
                {
```

```
                        first→internal_tree.change_key_of_element(adjacent_right_address, min_key(adjacent_right, level−1));
                        }
                    }
                // can we merge with the node to the left?
                else if (has_left)
                    {
                    merge(adjacent_left, erase_from, level−1);

                    // erase the merged node
                    first→internal_tree.erase_address(erase_from_address);
                    move_from_second_to_first(first);

                    erase_from = adjacent_left;
                    }
                // last resort, merge with the node to the right
                else
                    {
                    merge(erase_from, adjacent_right, level−1);

                    // erase the merged node
                    if (adjacent_right_from_second)
                        {
                        second→internal_tree.erase_address(adjacent_right_address);
                        delete_second_if_empty(first);
                        }
                    else
                        {
                        first→internal_tree.erase_address(adjacent_right_address);
                        move_from_second_to_first(first);
                        }
                    }
                }
            return erase_from;
            }
        }

    void erase_leaf(leaf_node* first, leaf_node* second,
        const key_type& k)
        {
        // preconditions:
        assert(first == root or not is_small(first));

        if (second != nullptr and not compare_function()(k, min_key(second)))
            {
            second→internal_tree.erase(k);
            delete_second_if_empty(first);
            }
        else
            {
            first→internal_tree.erase(k);
            if (not first→internal_tree.full())
                { move_from_second_to_first(first); }
            }
```

```
                }

        // SPLIT ROOT
        // this handles splitting a root node into two and creating a new root
        // with these as children. it returns a pointer to the new root. this is
        // the only way that the tree grows in height
private:

        void split_root()
                {
                abstract_node* splitted = split(root, height);

                inner_node* new_root = allocate_inner();
                new (new_root) inner_node();

                new_root→internal_tree.insert(
                        std::make_pair(min_key(root, height), root));
                new_root→internal_tree.insert(
                        std::make_pair(min_key(splitted, height), splitted));

                root = new_root;
                height += 1;
                }

        // SPLIT
private:

        abstract_node* split(abstract_node* node, const unsigned level)
                {
                if (level != 0)
                        { return split(static_cast<inner_node*>(node)); }
                else
                        { return split(static_cast<leaf_node*>(node)); }
                }

        abstract_node* split(inner_node* node)
                {
                // preconditions:
                assert(is_full(node));

                auto splitted = node→second;
                node→second = nullptr;
                return splitted;
                }

        abstract_node* split(leaf_node* node)
                {
                // preconditions:
                assert(is_full(node));

                auto splitted = node→second;
                node→second = nullptr;
                return splitted;
                }

        // STEAL FROM LEFT TO RIGHT
private:

        void steal_from_left_to_right(abstract_node* left, abstract_node* right,
                const unsigned level)
                {
                if (level != 0)
                        {
                        steal_from_left_to_right(
                                static_cast<inner_node*>(left),
```

```
                static_cast<inner_node∗>(right));
            }
        else
            {
            steal_from_left_to_right(
                static_cast<leaf_node∗>(left),
                static_cast<leaf_node∗>(right));
            }
        }

    void steal_from_left_to_right(inner_node∗ left, inner_node∗ right)
        {
        // preconditions:
        assert(is_small(right));
        assert(not is_small(left));

    move_from_first_to_second(right);

        auto to_steal_address = left→second→internal_tree.max_address()
              ;
        auto& to_steal = left→second→internal_tree.pool[
            to_steal_address].value;

        right→internal_tree.insert(to_steal);
        left→second→internal_tree.erase_address(to_steal_address);
        delete_second_if_empty(left);
        }

    void steal_from_left_to_right(leaf_node∗ left, leaf_node∗ right)
        {
        // preconditions:
        assert(is_small(right));
        assert(not is_small(left));

        move_from_first_to_second(right);

        auto to_steal_address = left→second→internal_tree.max_address()
              ;
        auto& to_steal = left→second→internal_tree.pool[
            to_steal_address].value;

        right→internal_tree.insert(to_steal);
        left→second→internal_tree.erase_address(to_steal_address);
        delete_second_if_empty(left);
        }

    // STEAL FROM RIGHT TO LEFT
private:

    void steal_from_right_to_left(abstract_node∗ left, abstract_node∗ right,
        const unsigned level)
        {
        if (level != 0)
            {
            steal_from_right_to_left(
                static_cast<inner_node∗>(left),
                static_cast<inner_node∗>(right));
            }
        else
            {
            steal_from_right_to_left(
                static_cast<leaf_node∗>(left),
                static_cast<leaf_node∗>(right));
            }
        }
```

```

        void steal_from_right_to_left(inner_node∗ left, inner_node∗ right)
                {
                // preconditions:
                assert(is_small(left));
                assert(not is_small(right));

                create_second_if_needed(left);

                auto to_steal_address = right→internal_tree.min_address();
                auto& to_steal = right→internal_tree.pool[
                        to_steal_address].value;

                left→second→internal_tree.insert(to_steal);
                right→internal_tree.erase_address(to_steal_address);
                move_from_second_to_first(right);
                }

        void steal_from_right_to_left(leaf_node∗ left, leaf_node∗ right)
                {
                // preconditions:
                assert(is_small(left));
                assert(not is_small(right));

                create_second_if_needed(left);

                auto to_steal_address = right→internal_tree.min_address();
                auto& to_steal = right→internal_tree.pool[
                        to_steal_address].value;

                left→second→internal_tree.insert(to_steal);
                right→internal_tree.erase_address(to_steal_address);
                move_from_second_to_first(right);
                }

        // MERGE
private:

        void merge(abstract_node∗ left, abstract_node∗ right, const unsigned level)
                {
                if (level != 0)
                        {
                        merge(
                                static_cast<inner_node∗>(left),
                                static_cast<inner_node∗>(right));
                        }
                else
                        {
                        merge(
                                static_cast<leaf_node∗>(left),
                                static_cast<leaf_node∗>(right));
                        }
                }

        void merge(inner_node∗ left, inner_node∗ right)
                {
                // preconditions:
                assert(is_small(left));
                assert(is_small(right));

                left→second = right;
                }

        void merge(leaf_node∗ left, leaf_node∗ right)
```

```
		{
		// preconditions:
		assert(is_small(left));
		assert(is_small(right));

		left→second = right;
		}

	// CREATE SECOND IF NEEDED
private:

	inner_node* create_second_if_needed(inner_node* node)
		{
		if (node→second == nullptr)
			{
			node→second = allocate_inner();
			new (node→second) inner_node();
			}
		return node→second;
		}

	leaf_node* create_second_if_needed(leaf_node* node)
		{
		if (node→second == nullptr)
			{
			node→second = allocate_leaf();
			new (node→second) leaf_node();
			}
		return node→second;
		}

	// DELETE SECOND IF EMPTY
private:

	void delete_second_if_empty(inner_node* node)
		{
		if (node→second != nullptr and node→second→internal_tree.empty())
			{
			deallocate_inner(node→second);
			node→second = nullptr;
			}
		}

	void delete_second_if_empty(leaf_node* node)
		{
		if (node→second != nullptr and node→second→internal_tree.empty())
			{
			deallocate_leaf(node→second);
			node→second = nullptr;
			}
		}

	// MOVE FROM FIRST TO SECOND
private:

	void move_from_first_to_second(inner_node* node)
		{
		create_second_if_needed(node);

		auto max_address = node→internal_tree.max_address();
		auto& max = node→internal_tree.pool[max_address].value;
		node→second→internal_tree.insert_min(max);
		node→internal_tree.erase_address(max_address);
```

```
		}

	void move_from_first_to_second(leaf_node* node)
		{
		create_second_if_needed(node);

		auto max_address = node->internal_tree.max_address();
		auto& max = node->internal_tree.pool[max_address].value;
		node->second->internal_tree.insert_min(max);
		node->internal_tree.erase_address(max_address);
		}

	// MOVE FROM SECOND TO FIRST
private:

	void move_from_second_to_first(inner_node* node)
		{
		if (node->second != nullptr)
			{
			auto min_address = node->second->internal_tree.min_address();
			auto& min = node->second->internal_tree.pool[min_address].value;
			node->internal_tree.insert_max(min);
			node->second->internal_tree.erase_address(min_address);

			delete_second_if_empty(node);
			}
		}

	void move_from_second_to_first(leaf_node* node)
		{
		if (node->second != nullptr)
			{
			auto min_address = node->second->internal_tree.min_address();
			auto& min = node->second->internal_tree.pool[min_address].value;
			node->internal_tree.insert_max(min);
			node->second->internal_tree.erase_address(min_address);

			delete_second_if_empty(node);
			}
		}

	// IS FULL?
private:

	bool is_full(const abstract_node* node, const unsigned level) const
		{
		if (level != 0)
			{ return is_full(static_cast<const inner_node*>(node)); }
		else
			{ return is_full(static_cast<const leaf_node*>(node)); }
		}

	bool is_full(const inner_node* node) const
		{
		return node->second != nullptr and node->second->internal_tree.full();
		}

	bool is_full(const leaf_node* node) const
		{
		return node->second != nullptr and node->second->internal_tree.
```

```
                        full();
                }

        // IS SMALL?
private:

        bool is_small(const abstract_node∗ node, const unsigned level) const
                {
                if (level != 0)
                        { return is_small(static_cast<const inner_node∗>(node));
                             }
                else
                        { return is_small(static_cast<const leaf_node∗>(node)); }
                }

        bool is_small(const inner_node∗ node) const
                {
                return node→second == nullptr;
                }

        bool is_small(const leaf_node∗ node) const
                {
                return node→second == nullptr;
                }

        // MIN KEY
private:

        key_type min_key(const abstract_node∗ node, const unsigned level) const
                {
                if (level != 0)
                        { return min_key(static_cast<const inner_node∗>(node)); }
                else
                        { return min_key(static_cast<const leaf_node∗>(node)); }
                }

        key_type min_key(const inner_node∗ node) const
                {
                return node→internal_tree.pool[
                        node→internal_tree.min_address()].value.first;
                }

        key_type min_key(const leaf_node∗ node) const
                {
                return node→internal_tree.pool[
                        node→internal_tree.min_address()].value.first;
                }

        // MAX KEY
private:

        key_type max_key(const abstract_node∗ node, const unsigned level) const
                {
                if (level != 0)
                        { return max_key(static_cast<inner_node∗>(node)); }
                else
                        { return max_key(static_cast<leaf_node∗>(node)); }
                }

        key_type max_key(const inner_node∗ node) const
                {
                return node→internal_tree.pool[
                        node→internal_tree.max_address()].value.first;
                }

```

```
        key_type max_key(const leaf_node∗ node) const
                {
                return node→internal_tree.pool[
                        node→internal_tree.max_address()].value.first;
                }
        };
```

Red-black tree

§ 2 *internal_rbtree.hpp*

```
#pragma once
#include <cassert>
#include <utility>
#include ”pool.hpp”

#include <iostream>

template<typename K, typename M, typename address_type, size_t size,
        class compare_function = std::less<K>>
class internal_rbtree
        {
public:
        // TYPEDEFS
        typedef K key_type;
        typedef M mapped_type;
        typedef std::pair<const key_type, mapped_type> value_type;

        typedef address_type size_type;

private:
        // NODE STRUCTURE

        // a type representing the color of a node
        enum class color_type : uint8_t { red, black };

        // the node structure
        struct node
                {
                node(
                        value_type value_,
                        address_type parent_,
                        address_type left_,
                        address_type right_,
                        color_type color_)
                :       value(value_),
                        parent(parent_),
                        left(left_),
                        right(right_),
                        color(color_)
                        {}

                value_type value;
                address_type parent;
                address_type left;
                address_type right;
                color_type color;
                };

        // DATA MEMBERS

public:
        pool<node, address_type, size − 4∗sizeof(address_type)> pool;

```

```
private:
        address_type root;      // address of the root of the tree
        address_type nil;       // address of the nil node
        address_type min;       // address of the minimum element
        address_type max;       // address of the maximum element

        // CONST HELPERS

        address_type tree_minimum(address_type x) const
                {
                while (pool[x].left != nil)
                        { x = pool[x].left; }
                return x;
                }

        address_type tree_maximum(address_type x) const
                {
                while (pool[x].right != nil)
                        { x = pool[x].right; }
                return x;
                }

public:
        address_type successor(address_type x) const
                {
                if (pool[x].right != nil)
                        { return tree_minimum(pool[x].right); }

                address_type y = pool[x].parent;

                while (y != nil and x == pool[y].right)
                        {
                        x = y;
                        y = pool[y].parent;
                        }
                return y;
                }

        address_type predecessor(address_type x) const
                {
                if (pool[x].left != nil)
                        { return tree_maximum(pool[x].left); }

                address_type y = pool[x].parent;

                while (y != nil and x == pool[y].left)
                        {
                        x = y;
                        y = pool[y].parent;
                        }
                return y;
                }

private:
        // MUTATING HELPERS

        void left_rotate(address_type x)
                {
                address_type y = pool[x].right;

                pool[x].right = pool[y].left;
                if (pool[y].left != nil)
                        { pool[pool[y].left].parent = x; }

                pool[y].parent = pool[x].parent;
```

```
        if (pool[x].parent == nil)
            { root = y; }
        else if (x == pool[pool[x].parent].left)
            { pool[pool[x].parent].left = y; }
        else
            { pool[pool[x].parent].right = y; }

        pool[y].left = x;
        pool[x].parent = y;
        }

    void right_rotate(address_type y)
        {
        address_type x = pool[y].left;

        pool[y].left = pool[x].right;
        if (pool[x].right != nil)
            { pool[pool[x].right].parent = y; }

        pool[x].parent = pool[y].parent;
        if (pool[y].parent == nil)
            { root = x; }
        else if (y == pool[pool[y].parent].left)
            { pool[pool[y].parent].left = x; }
        else
            { pool[pool[y].parent].right = x; }

        pool[x].right = y;
        pool[y].parent = x;
        }

    void transplant(address_type u, address_type v)
        {
        if (pool[u].parent == nil)
            { root = v; }
        else if (u == pool[pool[u].parent].left)
            { pool[pool[u].parent].left = v; }
        else
            { pool[pool[u].parent].right = v; }

        pool[v].parent = pool[u].parent;
        }

    // INSERT FIXUP

    void insert_fixup(address_type z)
        {
        while (pool[pool[z].parent].color == color_type::red)
            {
            if (pool[z].parent ==
                pool[pool[pool[z].parent].parent].left)
                {
                address_type y = pool[pool[pool[z].parent].parent].right;

                if (y != nil and pool[y].color == color_type::red)
                    {
                    pool[pool[z].parent].color = color_type::black;
                    pool[y].color = color_type::black;
                    pool[pool[pool[z].parent].parent].color = color_type::red;
                    z = pool[pool[z].parent].parent;
                    }
```

```
                else
                    {
                    if (z == pool[pool[z].parent].right)
                        {
                        z = pool[z].parent;
                        left_rotate(z);
                        }
                    pool[pool[z].parent].color = color_type::black;
                    pool[pool[pool[z].parent].parent].color = color_type::red;
                    right_rotate(pool[pool[z].parent].parent);
                    }
                }
            else
                {
                address_type y = pool[pool[pool[z].parent].parent].left;

                if (y != nil and pool[y].color == color_type::red)
                    {
                    pool[pool[z].parent].color = color_type::black;
                    pool[y].color = color_type::black;
                    pool[pool[pool[z].parent].parent].color = color_type::red;
                    z = pool[pool[z].parent].parent;
                    }
                else
                    {
                    if (z == pool[pool[z].parent].left)
                        {
                        z = pool[z].parent;
                        right_rotate(z);
                        }
                    pool[pool[z].parent].color = color_type::black;
                    pool[pool[pool[z].parent].parent].color = color_type::red;
                    left_rotate(pool[pool[z].parent].parent);
                    }
                }
            }

        pool[root].color = color_type::black;
        }

    // ERASE ADDRESS

public:
    void erase_address(address_type z)
        {
        // maintain minimum and maximum addresses
        if (min == z)
            { min = successor(z); }
        if (max == z)
            { max = predecessor(z); }

        // erase the value from the tree
        address_type x = nil;
        address_type y = z;
        color_type y_original_color = pool[y].color;

```

```
		if (pool[z].left == nil)
			{
			x = pool[z].right;
			transplant(z, pool[z].right);
			}
		else if (pool[z].right == nil)
			{
			x = pool[z].left;
			transplant(z, pool[z].left);
			}
		else
			{
			y = tree_minimum(pool[z].right);
			y_original_color = pool[y].color;
			x = pool[y].right;

			if (pool[y].parent == z)
				{
				pool[x].parent = y;
				}
			else
				{
				transplant(y, pool[y].right);
				pool[y].right = pool[z].right;
				pool[pool[y].right].parent = y;
				}

			transplant(z, y);
			pool[y].left = pool[z].left;
			pool[pool[y].left].parent = y;
			pool[y].color = pool[z].color;
			}

		// deallocate
		pool.deallocate(z);

		// fixup if necessary
		if (y_original_color == color_type::black)
			{ erase_fixup(x); }
		}

	// ERASE FIXUP
private:
	void erase_fixup(address_type x)
		{
		while (x != root and pool[x].color == color_type::black)
			{
			if (x == pool[pool[x].parent].left)
				{
				address_type w = pool[pool[x].parent].right;

				if (pool[w].color == color_type::red)
					{
					pool[w].color = color_type::black;
					pool[pool[x].parent].color = color_type::red;
					left_rotate(pool[x].parent);
					w = pool[pool[x].parent].right;
					}

				if (pool[pool[w].left].color == color_type::black and
					pool[pool[w].right].color == color_type::black)
					{
```

```
                pool[w].color = color_type::red;
                x = pool[x].parent;
                }
            else
                {
                if (pool[pool[w].right].color == color_type::black)
                    {
                    pool[pool[w].left].color = color_type::black;
                    pool[w].color = color_type::red;
                    right_rotate(w);
                    w = pool[pool[x].parent].right;
                    }

                pool[w].color = pool[pool[x].parent].color;
                pool[pool[x].parent].color = color_type::black;
                pool[pool[w].right].color = color_type::black;
                left_rotate(pool[x].parent);
                x = root;
                }
            }
        else
            {
            address_type w = pool[pool[x].parent].left;

            if (pool[w].color == color_type::red)
                {
                pool[w].color = color_type::black;
                pool[pool[x].parent].color = color_type::red;
                right_rotate(pool[x].parent);
                w = pool[pool[x].parent].left;
                }

            if (pool[pool[w].left].color == color_type::black and
                pool[pool[w].right].color == color_type::black)
                {
                pool[w].color = color_type::red;
                x = pool[x].parent;
                }
            else
                {
                if (pool[pool[w].left].color == color_type::black)
                    {
                    pool[pool[w].right].color = color_type::black;
                    pool[w].color = color_type::red;
                    left_rotate(w);
                    w = pool[pool[x].parent].left;
                    }

                pool[w].color = pool[pool[x].parent].color;
                pool[pool[x].parent].color = color_type::black;
                pool[pool[w].left].color = color_type::black;
                right_rotate(pool[x].parent);
```

```
                    x = root;
                    }
                }
            }

        pool[x].color = color_type::black;
        }

    // CONSTRUCTORS ET CETERA

    // copy protection
    internal_rbtree(internal_rbtree&);
    void operator=(internal_rbtree&);

public:

    internal_rbtree()
        {
        // create the sentinel node, nil
        // (parent, left and right are arbitrarily selected to point to itself)
        nil = pool.allocate();
        new( &pool[nil]) node(
            std::make_pair(key_type(), mapped_type()),
            nil, nil, nil, color_type::black);

        // root, min and max starts out as nil
        root = nil;
        min = nil;
        max = nil;

        assert(not full());
        }

    ~internal_rbtree()
        {}

    // ATTRIBUTES

    // is the tree empty?
    bool empty() const
        { return root == nil; }

    // is the tree full?
    bool full() const
        { return pool.full(); }

    // LOOKUP

    // returns the address to the node with the greatest key less than or equal
    // to the one given, or, if all keys are larger than the one given, returns
    // the node with the minimum key.
    // this behaviour is of specific use for our btree - it defines which chil-
    // dren are selected for which keys, in the inner nodes.
    address_type lookup(key_type key) const
        {
        address_type x = root;
        address_type y = root;
        address_type z = nil;

        while (x != nil)
            {
```

```
            y = x;
            if (pool[x].value.first > key)
                {
                x = pool[x].left;
                }
            else
                {
                z = x;
                x = pool[x].right;
                }
            }

        return (z != nil) ? z : y;
        }

    // INSERT

    // inserts a value into the tree
    void insert(value_type value)
        {
        // find parent, and check if we just need to replace
        address_type y = nil;
        address_type x = root;
        while (x != nil)
            {
            y = x;
            x = (compare_function()(value.first, pool[x].value.first))
                ? pool[x].left : pool[x].right;

            if (value.first == pool[x].value.first)
                {
                // key is already here, replace value and return
                pool[x].value.second = value.second;
                return;
                }
            }

        // if we couldn't just update we need a new node, and must not be full.
        // this needs to be checked by the caller
        assert(not full());

        // allocate a new node and fill in what we know
        address_type z = pool.allocate();
        new( &pool[z]) node(value, y, nil, nil, color_type::red);

        // hook it up to the tree
        if (y == nil)
            {
            root = z;
            // the tree is empty so the new value is both min and max
            min = z;
            max = z;
            }
        else if (compare_function()(pool[z].value.first, pool[y].value.first))
            {
            pool[y].left = z;
            // maintain minimum address
            if (min == y) { min = z; }
            }
        else
            {
            pool[y].right = z;
```

```
            // maintain maximum address
            if (max == y) { max = z; }
            }

        // fixup
        insert_fixup(z);
        }

    // use only if it is known that the inserted value is larger than any of the
    // existing
    void insert_max(value_type value)
        {
        // if we couldn't just update we need a new node, and must not be full.
        // this needs to be checked by the caller
        assert(not full());

        // allocate a new node and fill in what we know
        address_type z = pool.allocate();
        new( &pool[z]) node(value, max, nil, nil, color_type::red);

        // hook it up to the tree
        if (max == nil)
            {
            root = z;
            // the tree is empty so the new value is both min and max
            min = z;
            }
        else
            {
            pool[max].right = z;
            }

        max = z;

        insert_fixup(z);
        }

    // use only if it is known that the inserted value is smaller than any of
    // the existing
    void insert_min(value_type value)
        {
        // if we couldn't just update we need a new node, and must not be full.
        // this needs to be checked by the caller
        assert(not full());

        // allocate a new node and fill in what we know
        address_type z = pool.allocate();
        new( &pool[z]) node(value, min, nil, nil, color_type::red);

        // hook it up to the tree
        if (min == nil)
            {
            root = z;
            // the tree is empty so the new value is both min and max
            max = z;
            }
        else
            {
            pool[min].left = z;
            }

        min = z;
```

```

        insert_fixup(z);
        }

    // ERASE

    void erase(key_type key)
        {
        // find the value to be erased, if it exists
        address_type to_erase = root;
        while (to_erase != nil and pool[to_erase].value.first != key)
            {
            to_erase = (compare_function()(key, pool[to_erase].value.first))
                ? pool[to_erase].left
                : pool[to_erase].right;
            }

        if (to_erase != nil)
            { erase_address(to_erase); }
        }

    // KEY CHANGING

    // changes the key of an element, without checking!
    // this should be done with extreme care from the callers side, and *only*
    // when the caller knows that it does not change the ordering!
    void change_key_of_element(address_type address, key_type new_key)
        {
        *const_cast<key_type*>(&pool[address].value.first) = new_key;
        }

    // MIN/MAX

    address_type min_address() const
        { return min; }

    address_type max_address() const
        { return max; }
    };
```

Pool manager

§ 3 *pool.hpp*

```
#pragma once
#include <cassert>
#include <cstdint>

template<typename value_type, typename address_type, size_t size>
class pool
    {
private:
    // CONSTANTS

    static constexpr size_t max_size(size_t a, size_t b)
        { return a > b ? a : b; }

    // the size of an element. each element holds either a value or an
    // address, so we use the size of the largest of those types
    static constexpr size_t element_size =
        max_size(sizeof(value_type), sizeof(address_type));

```

```
        // the total size of the allocated memory. two addresses are subtracted
        // for bookkeeping, these are 'head' and 'back' - see below
        static constexpr size_t memory_size = size - 2 * sizeof(address_type);

        // DATA MEMBERS

        int8_t memory[memory_size];
        address_type head; // the next element to be allocated
        address_type back; // the first element that has never been allocated

        // HELPERS

        // these access an element as either a value_type or an address_type
        inline value_type& value_at(address_type address)
                { return *reinterpret_cast<value_type*>(memory + address); }

        inline const value_type& value_at(address_type address) const
                { return *reinterpret_cast<const value_type*>(memory + address); }

        inline address_type& address_at(address_type address)
                { return *reinterpret_cast<address_type*>(memory + address); }

        inline const address_type& address_at(address_type address) const
                { return *reinterpret_cast<const address_type*>(memory + address); }

        // CONSTRUCTORS ET CETERA

        // copy protection
        pool(pool&);
        void operator=(pool&);

public:
        pool() : head(0), back(0) {}

        ~pool() {}

        // FULL

        bool full() const
                { return (memory_size - head) < element_size; }

        // ALLOCATE

        // get memory for one value from the pool
        address_type allocate()
                {
                // precondition: pool is not full. must be checked by caller!
                assert(not full());

                // get the address to return
                address_type address = head;

                head = (address < back)
                // update 'head'. if the address we got is less than 'back' it has been
                // previously allocated and we can follow the address stored at it to
                // get the next head
                        ? address_at(address)
                // otherwise we are at 'back', which has not been previously allocated.
                // thus, all the higher addresses are free, and we can just point to
```

```
                // the next element
                        : back += element_size;

                return address;
                }

        // DEALLOCATE

        // return memory for one value to the pool
        void deallocate(address_type address)
                {
                // store the current head at the position of this element and make
                // this the new head
                address_at(address) = head;
                head = address;
                }

        // ACCESS

        // access memory at address
        value_type& operator[](address_type address)
                {
                assert(address <= memory_size - element_size);
                return value_at(address);
                }

        const value_type& operator[](address_type address) const
                {
                assert(address <= memory_size - element_size);
                return value_at(address);
                }
        };
```