Electronic Appendix

for

Untangling the Balancing and Searching of Balanced Binary Search Trees

Matt Austern Bjarne Stroustrup Mikkel Thorup John Wilkinson

AT&T Labs—Research

June 2003

Contents

A Source code **2**
A.1 Tree . . . 2
A.2 Searchers . . . 5
A.2.1 search_tree.hpp . . . 5
A.2.2 selection_search_tree.hpp . . . 8
A.2.3 simple_string_search_tree.hpp . . . 13
A.2.4 optima_string_search_tree.hpp . . . 16
A.3 Balancers . . . 22
A.3.1 red_black.hpp . . . 22
A.3.2 treap.hpp . . . 25
A.3.3 splay.hpp . . . 26
A.3.4 trivial.hpp . . . 28

B Test source code **30**
B.1 utilities . . . 30
B.1.1 comp.hpp . . . 30
B.1.2 gen.hpp . . . 31
B.1.3 del.hpp . . . 33
B.1.4 timer.hpp . . . 33
B.2 Performance tests . . . 34
B.2.1 time_inserts.hpp . . . 34
B.2.2 time_inserts.cpp . . . 35
B.2.3 time_finds.hpp . . . 36
B.2.4 time_finds.cpp . . . 37

A Source code

A.1 Tree

```
#ifndef BINTREES_TREE_HPP_GUARD
#define BINTREES_TREE_HPP_GUARD

// node_base class template and functions that operate
// on general binary tree nodes.

namespace bintrees {
template <class Node, class Balance> struct node_base
  :public Balance {
  Node * child[2];
  Node * parent;
  int    parity;

  node_base(bool is_header=false):
    Balance(is_header) {
      child[0] = static_cast<Node*>(&nil_obj);
      child[1] = static_cast<Node*>(&nil_obj);
      parent   = static_cast<Node*>(&nil_obj);
      parity   = is_header ? 1 : 0;
    }

  static Node* create_header()
  {
    Node * result =
      static_cast<Node*>
        (new node_base<Node, Balance>(/*is_header=*/true));
    result->parity = 1;
    return result;
  }

  static void destroy_header(Node* p)
  {
    delete static_cast<node_base<Node, Balance>*>(p);
  }

  static Node* nil;

  static node_base nil_obj;

  static void pre_rotate(Node*) {}

  static void pre_splice(Node*) {}

  static void pre_splice(Node*, Node*) {}
```

```
};

template <class Node, class Balance>
node_base<Node, Balance> node_base<Node, Balance>::nil_obj;

template <class Node, class Balance>
Node* node_base<Node, Balance>::nil =
  static_cast<Node*>(&node_base<Node, Balance>::nil_obj);

template <class Node>
inline Node* add_leaf(Node* p, Node* q, int c)
{
  p->child[c] = q;
  q->parent   = p;
  q->parity   = c;
  Node::add_fixup(q);
  return q;
}

template <class Node>
Node* splice_out(Node* q)

// q is a node with at least one nil child
{
  Node::pre_splice(q);
  Node* p = q->parent;
  Node* x = q->child[q->child[0] == Node::nil];
  x->parent = p;
  x->parity = q->parity;
  p->child[q->parity] = x;
  return x;
}

template <class Node>
Node* splice_out(Node* q, Node* r)

// q is a node with at least one nil child, r an ancestor node.
{
  Node::pre_splice(q, r);

  // splice out q
  Node* p = q->parent;
  Node* x = q->child[q->child[0] == Node::nil];
  x->parent = p;
  x->parity = q->parity;
  p->child[q->parity] = x;
  // q has been spliced out, now relink it in place of r.
  r->parent->child[r->parity] = q;
  q->parent = r->parent;
  q->parity = r->parity;
  q->child[0] = r->child[0];
  q->child[1] = r->child[1];
  q->child[0]->parent = q;
  q->child[1]->parent = q;
  return x;
}

template <class Node>
inline void rotate(Node * q)
{
  int   c = q->parity;
  Node* p = q->parent;
  Node* B = q->child[!c];

  Node::pre_rotate(q);
  p->child[c] = B;
  B->parent = p;
  B->parity = c;
  q->parent = p->parent;
  q->parity = p->parity;
  q->parent->child[q->parity] = q;
  p->parent = q;
  p->parity = !c;
  q->child[!c] = p;
}

// convenience functions left and right
template <class Node> inline Node* left (Node* p)
   {return p->child[0];}
template <class Node> inline Node* right(Node* p)
   {return p->child[1];}

// min and max

template <class Node> inline Node* extremum(Node* p, int c) {
  while (p->child[c] != Node::nil)
    p = p->child[c];
  return p;
}

template <class Node> inline Node* min(Node* p)
  { return extremum(p, 0); }
template <class Node> inline Node* max(Node* p)
```

```
  { return extremum(p, 1); }

// Successor and predecessor

template <class Node>
Node* step(Node* p, int c) {
  if (p->child[c] != Node::nil)
    return extremum(p->child[c], !c);

  while (p->parity == c)
    p = p->parent;

  return p->parent;
}

template <class Node> inline Node* predecessor(Node* p)
  { return step(p, 0); }
template <class Node> inline Node* successor  (Node* p)
  { return step(p, 1); }

} // Close namespace bintrees

#endif /* BINTREES_TREE_HPP_GUARD */
```

A.2 Searchers

A.2.1 search_tree.hpp

```
#ifndef BINTREES_SEARCH_TREE_HPP_GUARD
#define BINTREES_SEARCH_TREE_HPP_GUARD

#include "../tree/tree.hpp"
#include "key_ordered_type.hpp"

// Binary search tree class template

namespace bintrees {

template <class Val, class Balance>
struct search_node
  : public node_base<search_node<Val, Balance>, Balance>
{
  Val val;
  search_node(Val v): val(v) {}
};


template <class Val, class Balance,
          class Key, class Extract, class Comp>
class search_tree
  : public key_ordered_type<Val, Key, Extract, Comp>
{
public:
  typedef search_node<Val, Balance> Node;
  search_tree(): header(Node::create_header()){}

  ~search_tree() {
    if (header->child[0] != Node::nil)
      erase_tree(header->child[0]);
    Node::destroy_header(header);
  }

  Node* root()             { return header->child[0]; }
  const Node* root() const { return header->child[0]; }

  Node* insert(const Val& v) {
    Node* p = header;
    Node* q = header->child[0];
    int   c = 0;

    while (q!=Node::nil) {
      p = q;
      c = !comp(key(v), key(p->val));
      q = p->child[c];
    }

    Node* r = new Node(v);
    add_leaf(p, r, c);
    return r;
  }

  Node* lower_bound(const Key& k) {
    Node* p = header;
    Node* q = header->child[0];
    int   c = 0;

    while (q != Node::nil) {
       p = q;
       c = comp(key(p->val),k);
       q = p->child[c];
    }
    Node* r = p;
    if (c) {
      while (p->parity) p = p->parent;
      p = p->parent;
    }
    Node::touch((p == header) ? r : p);
    return p;
  }

  Node* find(const Key& k) {
    Node* p = header;
    Node* q = header->child[0];
    while (q != Node::nil) {
      p = q;
      if (comp(k, key(q->val))) // k < key(q->val)
        q = q->child[0];
      else {
        if (!comp(key(q->val), k)) {
          Node::touch(q);  // possibly rebalance
          return q;
        }
        q = q->child[1];
      }
    }
    Node::touch(p);
    return header;
  }

  Node* upper_bound(const Key& k) {
    Node* p = header;
    Node* q = header->child[0];
    int c   = 0;
    while (q != Node::nil) {
       p = q;
       c = !comp(k, key(p->val));
       q = p->child[c];
    }
    Node* r = p;
    if (c) {
      while (p->parity)
        p = p->parent;
      p = p->parent;
    }
    Node::touch(r);
    return p;
  }

  void erase(Node* n) {
    Node::detach(n);
    delete n;
  }
```

```
  Node* begin() { return min(header);}
  Node* end()   { return header; }

  const Node* begin() const { return min(header);}

  const Node* end()   const { return header; }

private:
  void erase_tree(Node* n) {
    while (n != header) {
      while (n->child[0] != Node::nil ||
             n->child[1] != Node::nil) {
        if (n->child[0] != Node::nil)
          n = n->child[0];
        else
          n = n->child[1];
      }
      Node* q = n;
      n = n->parent;
      n->child[q->parity] = Node::nil;
      delete q;
    }
  }

  Node*   header;
  Extract key;
  Comp    comp;
};

} // Close namespace bintrees

#endif /* BINTREES_SEARCH_TREE_HPP_GUARD */
```

A.2.2 selection_search_tree.hpp

```
#ifndef BINTREES_SELECTION_SEARCH_TREE_HPP_GUARD
#define BINTREES_SELECTION_SEARCH_TREE_HPP_GUARD

#include "../tree/tree.hpp"
#include "key_ordered_type.hpp"

namespace bintrees {

// Base class for nodes with left_size.
// Used to implement selection search trees.
```

```
template <class Node, class Balance>
struct selection_node: node_base<Node, Balance> {
  unsigned int left_size;

  selection_node(unsigned int r = 1, bool is_header = false)
    : bintrees::node_base<Node, Balance>
       (is_header), left_size(r)
    {
      child[0] = nil;
      child[1] = nil;
      parent   = nil;
    }

  static Node* create_header()
  {
    return static_cast<Node*>(new selection_node(0, true));
  }

  static void destroy_header(Node* p)
  {
    delete static_cast<selection_node*>(p);
  }

  static Node* nil;

  static selection_node nil_obj;

  static void pre_rotate(Node* q) {
    Node* p = q->parent;
    if (q->parity == 0)
      p->left_size -= q->left_size;
    else
      q->left_size += p->left_size;
  }

  static void pre_splice(Node* y, Node* z = 0)
  {
    Node* p = y;
    while (p->parent != Node::nil) {
      if (p->parity == 0)
        --p->parent->left_size;
      p = p->parent;
    }
    if (z != 0) {
      y->left_size = z->left_size;
    }
  }
```

```
};

template <class Node, class Balance>
selection_node<Node, Balance>
  selection_node<Node, Balance>::nil_obj = 0;

template <class Node, class Balance>
Node* selection_node<Node, Balance>::nil =
  static_cast<Node*>(&selection_node<Node, Balance>::nil_obj);

// Node type for selection search tree.

template <class Val, class Balance>
struct selection_search_node
  :public selection_node<
    selection_search_node<Val, Balance>, Balance>
{
  Val val;
  selection_search_node(Val v): val(v) {}
};


template <class Val, class Balance,
          class Key, class Extract, class Comp>
class selection_search_tree
  : public key_ordered_type<Val, Key, Extract, Comp>
{
public:
  typedef selection_search_node<Val, Balance> Node;
  selection_search_tree(): header(Node::create_header()){}

  ~selection_search_tree() {
    if (header->child[0] != Node::nil)
      erase_tree(header->child[0]);
    Node::destroy_header(header);
  }

  Node* root()             { return left(header); }
  const Node* root() const { return left(header); }

  Node* insert(const Val& v) {
    Node* p = header;
    Node* q = header->child[0];
    int   c = 0;
    ++header->left_size;
    while (q != Node::nil) {
      p = q;
      c = !comp(key(v), key(p->val));
      if (c == 0)
        ++p->left_size;
      q = p->child[c];
    }

    Node* r = new Node(v);
    add_leaf(p, r, c);
    return r;

  }

  Node* lower_bound(const Key& k) {
    Node* p = header;
    Node* q = header->child[0];
    int   c = 0;

    while (q != Node::nil) {
       p = q;
       c = comp(key(p->val),k);
       q = p->child[c];
    }
    Node* r = p;
    if (c) {
      while (p->parity) p = p->parent;
      p = p->parent;
    }
    Node::touch((p == header) ? r : p);
    return p;
  }

  Node* find(const Key& k) {
    Node* p = lower_bound(k);
    return p == header ? header
                       : comp(k, key(p->val)) ? header : p;
  }


  Node* upper_bound(const Key& k) {
    Node* p = header;
    Node* q = header->child[0];
    int c   = 0;
    while (q != Node::nil) {
       p = q;
       c = !comp(k, key(p->val));
       q = p->child[c];
    }
```

```
    Node* r = p;
    if (c) {
      while (p->parity)
        p = p->parent;
      p = p->parent;
    }
    Node::touch(r);
    return p;
  }

  Node* select(unsigned int i) {
    if (header->left_size < i)
      return header;
    ++i;
    Node* p = header->child[0];
    for (;;) {
      if (i < p->left_size)
        p = p->child[0];
      else {
        i -= p->left_size;
        if (i == 0) return p;
        p = p->child[1];
      }
    }
  }

  void erase(Node* n) {
    Node::detach(n);
    delete n;
  }

  Node* begin() { return min(header);}
  Node* end()   { return header; }

  const Node* begin() const { return min(header);}
  const Node* end()   const { return header; }

private:
  void erase_tree(Node* n) {
    while (n != header) {
      while (n->child[0] != Node::nil ||
             n->child[1] != Node::nil) {
        if (n->child[0] != Node::nil)
  n = n->child[0];
else
  n = n->child[1];
      }
      Node* q = n;
      n = n->parent;
      n->child[q->parity] = Node::nil;
      delete q;
    }
  }

  Node* header;
  Extract key;
  Comp comp;
};

} // Close namespace bintrees

#endif /* BINTREES_SELECTION_SEARCH_TREE_HPP_GUARD */
```

A.2.3 simple_string_search_tree.hpp

```
#ifndef BINTREES_SIMPLE_STRING_SEARCH_TREE_HPP_GUARD
#define BINTREES_SIMPLE_STRING_SEARCH_TREE_HPP_GUARD

#include "../tree/tree.hpp"
#include "key_ordered_type.hpp"

// Binary simple_string_search tree class template

namespace bintrees {

template <class Val, class Balance>
struct simple_string_search_node
  : public node_base<
     simple_string_search_node<Val, Balance>, Balance>
{
  Val val;
  simple_string_search_node(Val v): val(v) {}
};


template <class Val, class Balance,
          class Key, class Extract, class Comp>
class simple_string_search_tree
  : public key_ordered_type<Val, Key, Extract, Comp>
{
public:
  typedef simple_string_search_node<Val, Balance> Node;
  simple_string_search_tree(): header(Node::create_header()){}
```

```
~simple_string_search_tree() {
  if (header->child[0] != Node::nil)
    erase_tree(header->child[0]);
  Node::destroy_header(header);
}

Node* root()             { return header->child[0]; }
const Node* root() const { return header->child[0]; }

Node* insert(const Val& v) {
  Node* p = header;
  Node* q = header->child[0];
  int   c = 0;

  while (q!=Node::nil) {
    p = q;
    c = !comp(key(v), key(p->val));
    q = p->child[c];
  }

  Node* r = new Node(v);
  add_leaf(p, r, c);
  return r;
}

Node* lower_bound(const Key& k) {
  Node* p = header;
  Node* q = header->child[0];
  int   c = 0;

  while (q != Node::nil) {
     p = q;
     c = comp(key(p->val),k);
     q = p->child[c];
  }
  Node* r = p;
  if (c) {
    while (p->parity) p = p->parent;
    p = p->parent;
  }
  Node::touch((p == header) ? r : p);
  return p;
}

Node* find(const Key& k) {
  Node* p = header;
  Node* q = header->child[0];
  int   m;
  while (q != Node::nil) {
    p = q;
    m = 0;
    if (comp(k, key(q->val), m)) // k < key(q->val)
      q = q->child[0];
    else {
      if (!comp(key(q->val), k, m)) {
        Node::touch(q);  // possibly rebalance
        return q;
      }
      q = q->child[1];
    }
  }
  Node::touch(p);
  return header;
}

Node* upper_bound(const Key& k) {
  Node* p = header;
  Node* q = header->child[0];
  int c   = 0;
  while (q != Node::nil) {
     p = q;
     c = !comp(k, key(p->val));
     q = p->child[c];
  }
  Node* r = p;
  if (c) {
    while (p->parity)
      p = p->parent;
    p = p->parent;
  }
  Node::touch(r);
  return p;
}

void erase(Node* n) {
  Node::detach(n);
  delete n;
}

Node* begin() { return min(header);}
Node* end()   { return header; }

const Node* begin() const { return min(header);}
```

```
  const Node* end()   const { return header; }

private:
  void erase_tree(Node* n) {
    while (n != header) {
      while (n->child[0] != Node::nil ||
             n->child[1] != Node::nil) {
        if (n->child[0] != Node::nil)
          n = n->child[0];
        else
          n = n->child[1];
      }
      Node* q = n;
      n = n->parent;
      n->child[q->parity] = Node::nil;
      delete q;
    }
  }

  Node*   header;
  Extract key;
  Comp    comp;
};

} // Close namespace bintrees

#endif /* BINTREES_SIMPLE_STRING_SEARCH_TREE_HPP_GUARD */
```

A.2.4 optima_string_search_tree.hpp

```
#ifndef BINTREES_OPTIMAL_STRING_SEARCH_TREE_HPP_GUARD
#define BINTREES_OPTIMAL_STRING_SEARCH_TREE_HPP_GUARD
#include "../tree/tree.hpp"
#include "key_ordered_type.hpp"

namespace bintrees {

template <class Node, class Balance>
struct prefix_node: public node_base<Node, Balance> {
  unsigned int prefix[2];

  prefix_node(bool is_header = false)
    : node_base<Node, Balance> (is_header)
    {
      child[0] = nil;
      child[1] = nil;
      parent   = nil;
      prefix[0] = 0;
      prefix[1] = 0;
    }

  static Node* create_header()
  {
    return static_cast<Node*>(new prefix_node(true));
  }

  static void destroy_header(Node * p)
  {
    delete static_cast<prefix_node*>(p);
  }

  static void pre_rotate(Node* q)
  {
    Node* p = q->parent;
    int c = q->parity;
    p->prefix[!c] = q->prefix[c];
    if (q->prefix[c] > p->prefix[c])
      q->prefix[c] = p->prefix[c];
  }


};

template <class Val, class Balance,
          class Key, class Extract, class Comp>
struct prefix_search_node
  : public prefix_node<prefix_search_node
                         <Val, Balance, Key, Extract, Comp>,
                        Balance>
{
  typedef prefix_search_node<Val, Balance,
                             Key, Extract, Comp> Node;
  Val val;
  prefix_search_node(Val v): val(v) {}

  static void pre_splice(Node* p)
  {
    for (int c = 0; c <= 1; ++c) {
      Node* q = p->child[c];
      while (q != Node::nil) {
        if (q->prefix[c] > p->prefix[c])
          q->prefix[c] = p->prefix[c];
        q = q->child[!c];
      }
    }
  }

  static void pre_splice(Node* p, Node* r)
  {
    int     m;
    Comp    comp;
    Extract key;

    int   c = r->parity;
    Node* q = r->parent;

    pre_splice(p);

    if (q->parent == Node::nil) // q is header
      p->prefix[0] = p->prefix[1] = 0;
    else {
      m = 0;
      comp(key(p->val), key(q->val), m);
      p->prefix[c] = m;
      while (q->parity == c) q = q->parent;
      if (q->parent == Node::nil ||
          q->parent->parent == Node::nil)
        p->prefix[!c] = 0;
      else {
        m = 0;
        comp(key(p->val), key(q->parent->val), m);
        p->prefix[!c] = m;
      }
    }

    for (int c = 0; c <= 1; ++c) {
      m = 0;
      q = r->child[c];

      if (q == p)
        q = q->child[q->child[0] == Node::nil];

      while (q != Node::nil) {
        if (q == p) break;
        comp(key(p->val), key(q->val), m);
        q->prefix[c] = m;
        q = q->child[!c];
        if (q == p)
          q = q->child[q->child[0] == Node::nil];
      }
    }
  }
};

template <class Val, class Balance,
          class Key, class Extract, class Comp>
class optimal_string_search_tree
  : public key_ordered_type<Val, Key, Extract, Comp>
{
  public:
    typedef prefix_search_node
              <Val, Balance, Key, Extract, Comp> Node;

    optimal_string_search_tree()
     : header(Node::create_header()) {}

    ~optimal_string_search_tree()
    {
      if (header->child[0] != Node::nil)
        erase_tree(header->child[0]);
      Node::destroy_header(header);
    }

    Node* root()             {return left(header);}
    const Node* root() const {return left(header);}

    Node* insert(const Val& v) {
      Node* p = header;
      Node* q = header->child[0];
      int   c = 0;
      int            i = 0;
      int            m = 0;

      while (q != Node::nil) {
        p = q;
        if (p->prefix[i] < m)
          c = !i;
        else
          i = c = !comp(key(v), key(p->val), m);
        q = p->child[c];
      }

      Node* r = new Node(v);
      r->prefix[i] = m;
      if (i == c)
        r->prefix[!i] = p->prefix[!i];
      else
        r->prefix[!i] = p->prefix[i];
      add_leaf(p, r, c);
      return r;
    }

    Node* lower_bound(const Key& k) {
      Node* p = header;
      Node* q = header->child[0];
      int   c = 0;
      int   i = 0;
      int   m = 0;

      while (q != Node::nil) {
        p = q;
        if (p->prefix[i] < m)
          c = !i;
        else
          i = c = comp(key(p->val), k, m);
        q = p->child[c];
      }

      Node* r = p;
      if (c) {
        while (p->parity)
          p = p->parent;
        p = p->parent;
      }

      Node::touch(p == header ? r : p);
      return p;
    }

    Node* find(const Key& k) {
      Node* p = header;
      Node* q = header->child[0];
      int   c = 0;
      int   i = 0;
      int   m = 0;

      while (q != Node::nil) {
        p = q;
        if (p->prefix[i] < m)
          c = !i;
        else {
          i = c = comp(key(p->val), k, m);
          if (!c && !comp(k, key(p->val), m)) {
            Node::touch(p);
            return p;
          }
        }
        q = p->child[c];
      }

      Node::touch(p);
      return header;
    }

    Node* upper_bound(const Key& k) {
      Node* p = header;
      Node* q = header->child[0];
      int   c = 0;
      int   i = 0;
      int   m = 0;

      while (q != Node::nil) {
        p = q;
        if (p->prefix[i] < m)
          c = !i;
        else
          i = c = !comp(k, key(p->val), m);
        q = p->child[c];
      }

      Node* r = p;
      if (c) {
        while (p->parity)
          p = p->parent;
        p = p->parent;
      }
      Node::touch(r);

      return p;
    }

    void erase(Node* p)
    {
      Node::detach(p);
      delete p;
    }

    Node* begin() {return min(header);}
    Node* end()   {return header;}

    const Node* begin() const {return min(header);}
    const Node* end()   const {return header;}
```

```
  private:
    void erase_tree(Node* n) {
      while (n != header) {
        while (n->child[0] != Node::nil ||
               n->child[1] != Node::nil) {
          if (n->child[0] != Node::nil)
            n = n->child[0];
          else
            n = n->child[1];
        }

        Node * q = n;
        n = n->parent;
        n->child[q->parity] = Node::nil;
        delete q;
      }
    }

    Node*   header;
    Extract key;
    Comp    comp;
  };
} // end namespace bintrees

#endif /*BINTREES_OPTIMAL_STRING_SEARCH_TREE_HPP_GUARD */
```

A.3 Balancers

A.3.1 red_black.hpp

```
#ifndef BINTREES_RED_BLACK_HPP_GUARD
#define BINTREES_RED_BLACK_HPP_GUARD

//
// red-black tree implementation of balanced binary tree
//

namespace bintrees {

struct red_black_balance {
  enum node_color {black, red, green};
  node_color color;
  red_black_balance(bool is_header)
    : color(is_header ? green : black) {}

  template <class Node>
  static void add_fixup(Node* x)
```

```
{
  x->color = red;
  while (x->parent->color == red) {
    Node* y = x->parent->parent->child[!x->parent->parity];
    if (y->color == red) {
      x->parent->color = black;
      x->parent->parent->color = red;
      y->color = black;
      x = x->parent->parent;
    }
    else {
      if (x->parity != x->parent->parity) {
        rotate(x);
        x = x->child[x->parity];
      }
      x->parent->color = black;
      x->parent->parent->color = red;
      rotate(x->parent);
    }
  }
  if (x->parent->color == green)
    x->color = black;
}

template <class Node>
static void detach(Node* z)
{
  Node *x, *p, *y;
  if (z->child[0] == Node::nil ||
      z->child[1] == Node::nil) {
    y = z;
    x = splice_out(y);
    if (y->color == red)
      return;
  }
  else {
    y = min(z->child[1]);
    node_color old_y_color = y->color;
    x = splice_out(y, z);
    y->color = z->color;
    if (old_y_color == red)
     return;
  }

  // fixing up the coloring. We know that we would like to
  // give x an "extra" black. In particular, if
  // x is already black, accounting implies that the
  // brother of x cannot be nil.

  p = x->parent;
  Node* w = p->child[!x->parity];
  while (x->color           == black &&
         p->color           != green &&
         w->color           == black &&
         w->child[0]->color == black &&
         w->child[1]->color == black) {
    w->color = red;
    x = p;
    p = x->parent;
    w = p->child[!x->parity];
  }

  if (x->color == red || p->color == green) {
    x->color = black;
    return;
  }

  int c = x->parity;
  if (w->color == red) {
    w->color = black;
    p->color = red;
    rotate(w);
    w = p->child[!c];
    if (w->child[0]->color == black &&
        w->child[1]->color == black) {
      w->color = red;
      p->color = black;
      return;
    }
  }

  if (w->child[!c]->color == black) {
    Node* v = w->child[c];
    v->color = black;
    w->color = red;
    rotate(v);
    w = v;
  }

  w->color = p->color;
  p->color = black;
  w->child[!c]->color = black;
  rotate(w);
  return;
```

```
  }

  template <class Node>
  static void touch(const Node*) {}
};

} // Close namespace bintrees

#endif /* BINTREES_RED_BLACK_HPP_GUARD */
```

A.3.2 treap.hpp

```
#ifndef BINTREES_TREAP_HPP_GUARD
#define BINTREES_TREAP_HPP_GUARD

// treap implementation of balanced binary tree.

#include <limits.h>
#include <stdlib.h>

namespace bintrees {

struct treap_balance {
  int priority;
  treap_balance(bool is_header = false):
    priority(is_header? INT_MAX : 0) {}

  template <class Node>
  static void add_fixup(Node* q)
  {
    q->priority = (lrand48() >> 1) + 1;
    while (q->priority > q->parent->priority)
      rotate(q);
  }

  template <class Node>
  static void detach(Node* p)
  {
    Node * q;
    while ((q =
              p->child[p->child[1]->priority >
              p->child[0]->priority]) != Node::nil)
      rotate(q);
    splice_out(p);
  }

  template <class Node>
  static void touch(Node*) {}
};


} // Close namespace bintrees

#endif /* BINTREES_TREAP_HPP_GUARD */
```

A.3.3 splay.hpp

```
#ifndef BINTREES_SPLAY_HPP_GUARD
#define BINTREES_SPLAY_HPP_GUARD

// splay tree implementation of balanced binary tree

namespace bintrees {

struct splay_balance {
  splay_balance(bool) {}

  template <class Node>
  static void splay(Node* q)
  {
    Node* p = q->parent;
    Node* g = p->parent;
    while (g != Node::nil) {
      if (g->parent == Node::nil) {
        rotate(q);
        return;
      }

      if (p->parity == q->parity) {
        rotate(p);
        rotate(q);
      }
      else {
        rotate(q);
        rotate(q);
      }

      p = q->parent;
      g = p->parent;
    }
  }
```

```
template <class Node>
static void splay(Node* q, Node* r)
{
  Node* p = q->parent;
  Node* h = r->parent;
  while (p != h) {
    if (p->parent == h) {
      rotate(q);
      return;
    }

    if (p->parity == q->parity) {
      rotate(p);
      rotate(q);
    }
    else {
      rotate(q);
      rotate(q);
    }
    p = q->parent;
  }
}

template <class Node>
static void add_fixup(Node* q)
{
  splay(q);
}

template <class Node>
static void detach(Node* q)
{
  if (q->child[0] == Node::nil) {
    splice_out(q);
    return;
  }

  Node* r = q->child[0];
  Node* s = max(r);
  splay(s, r);
  splice_out(s, q);
  return;
}

template <class Node>
static void touch(Node* p)
```

```
  {
    if (p->parent != Node::nil)
      splay(p);
  }
};

} // Close namespace bintrees

#endif /* BINTREES_SPLAY_HPP_GUARD */
```

A.3.4 trivial.hpp

```
#ifndef BINTREES_TRIVIAL_HPP_GUARD
#define BINTREES_TRIVIAL_HPP_GUARD

// trivial implementation of binary tree.
// Defines very simple versions
// of attach and detach which do no rebalancing.

namespace bintrees {

struct trivial_balance {
  trivial_balance(bool) {}

  template <class Node>
  static void add_fixup(Node* q)
  {
  }

  template <class Node>
  static void detach(Node* p)
  {
    if (p->child[0] == Node::nil ||
        p->child[1] == Node::nil) {
      splice_out(p);
      return;
    }

    Node* q = min(p->child[1]);
    splice_out(q, p);
    return;
  }

  template <class Node>
  static void touch(Node*) {}
};
} // Close namespace bintrees

#endif /* BINTREES_TRIVIAL_HPP_GUARD */
```

B Test source code

B.1 utilities

We describe briefly the supporting components that supply comparators, generators, and the timer class.

B.1.1 comp.hpp

This header provides three comparators as specializations of a class template `less`. The first provides simple comparison of integers and the second simple lexicographic comparison of strings. The third, used by the simple string search and optimal string search trees, also provides lexicographic comparison of strings, but takes a third integer reference parameter that is used as a starting position for the comparison and is reset to the new maximum match position after the comparison.

```
#ifndef BINTREES_COMP_HPP_GUARD
#define BINTREES_COMP_HPP_GUARD

namespace bintrees {
  template <class Key>
  struct less {};

  template <>
  struct less<int> {
    bool operator()(int x, int y) const {return x < y;}
  };

  template <>
  struct less<char*> {
    bool operator()(char* x, char* y) const {
      char* p = x, *q = y;
      while (*q != 0) {
      if (*p < *q)
          return true;
      if (*p++ > *q++)
          return false;
      }
      return false;
    }

    bool operator()(char* x, char* y, int& m) const {
      char * p = x+m, *q = y+m;
      while (*q != 0) {
        if (*p < *q) {
          m = p-x;
          return true;
        }

        if (*p > *q)
```

```
          break;

        ++p, ++q;
      }

      m = p-x;
      return false;
    }
  };

} // Close namespace bintrees

#endif /* BINTREES_COMP_HPP_GUARD */
```

B.1.2 gen.hpp

This header provides a number of generators that supply successive values of the type specified by the template argument. The specializations that we provide all generate either integer or string values.

`simple_gen` just supplies consecutive integers or consecutive decimal representations of integers.

`binary_gen` supplies consecutive integers or consecutive binary representations of integers.

`constant_gen` always supplies the same value (integer or string). It is included only as an example; it is not used in either of the timing programs.

```
#ifndef BINTREES_GEN_HPP_GUARD
#define BINTREES_GEN_HPP_GUARD
#include <stdio.h>

using std::vector;
namespace bintrees {

template <class Val>
struct simple_gen {};

template <>
struct simple_gen<int> {
  mutable int x;
  simple_gen(): x(0) {}
  int operator()() const {
    return ++x;
  }
};

template <>
struct simple_gen<char*> {
  mutable int x;
  simple_gen(): x(0) {}
  char* operator()() const {
    ++x;
    char * result = new char[10];
    sprintf(result, "%d", x);
    return result;
  }
};

template <class Val>
struct binary_gen {};

template <>
struct binary_gen<int> {
  mutable int x;
  binary_gen(): x(0) {}
  int operator()() const {
    return ++x;
  }
};

template <>
struct binary_gen<char*> {
  mutable vector<char> v;
  binary_gen(): v() {}
  char* operator()() const {
    vector<char>::iterator first = v.begin();
    vector<char>::iterator last  = v.end();
    while (first != last) {
      if (*first == '0') {
        *first = '1';
        break;
      }
      else {
        *first = '0';
        ++first;
      }
    }
    if (first == last)
      v.push_back('1');
    char* result = new char[v.size() + 1];
    reverse_copy(v.begin(), v.end(), result);
    result[v.size()] = 0;
    return result;
  }
};

template <class Val>
struct constant_gen {};
```

```
template <>
struct constant_gen<int> {
  constant_gen() {}
  int operator()() const {
    return 1;
  }
};

template <>
struct constant_gen<char*> {
  constant_gen() {}
  char* operator()() const {
    char * result = new char[4];
    sprintf(result, "%s", "foo");
    return result;
  }
};

} // Close namespace bintrees

#endif /* BINTREES_GEN_HPP_GUARD */
```

B.1.3 del.hpp

This header simply supplies a function object for deleting objects of the type specified by the template parameter. We need this to keep our simple timing programs from grabbing memory and not releasing it.

```
#ifndef BINTREES_DEL_HPP_GUARD
#define BINTREES_DEL_HPP_GUARD

template <class Val>
struct deletor {
  void operator()(Val) const {}
};

template <>
struct deletor<char*> {
  void operator()(char* v) const {delete v;}
};

#endif /* BINTREES_DEL_HPP_GUARD */
```

B.1.4 timer.hpp

This header just provides a simple timer class.

```
#ifndef BINTREES_TIMER_HPP_GUARD
```

```
#define BINTREES_TIMER_HPP_GUARD
#include "time.h"

namespace bintrees {

class timer {
public:
  timer()  {accum_time = 0;}
  ~timer() {}
  void start() {start_time = clock();}
  void stop()  {accum_time += (clock() - start_time);}
  double total_time()
    {return (double) accum_time / CLOCKS_PER_SEC;}

private:
  clock_t start_time;
  clock_t accum_time;
};

} // Close namespace bintrees

#endif /* BINTREES_TIMER_HPP_GUARD */
```

B.2 Performance tests

B.2.1 time_inserts.hpp

```
#include <iostream>
#include <time.h>
#include <functional>
#include <algorithm>
#include <vector>
#include <set>
#include "../tree/tree.hpp"
#include "timer.hpp"
#include "comp.hpp"
#include "gen.hpp"
#include "del.hpp"

template <
  template <class Val, class Balance,
            class Key, class Extract, class Comp>
    class Tree,
  class Val, class Balance, class Key,
  class Extract, class Comp,
  template <class Val> class Generator>
double time_inserts(unsigned int size)
{
  typedef Tree<Val, Balance, Key, Extract, Comp> tree_type;
  tree_type x;
  bintrees::timer tmr;

  std::vector<Val>    v(size);
  Generator<Val> gen;
  deletor<Val>   del;
  for (int i = 0; i < size; ++i)
    v[i] = gen();
  std::random_shuffle(v.begin(), v.end());
  tmr.start();
  for (int i = 0; i < size; ++i)
    x.insert(v[i]);
  tmr.stop();
  for (int i = 0; i < size; ++i)
    del(v[i]);
  return tmr.total_time();
}
```

B.2.2 time_inserts.cpp

```
#include <iostream>
#include "../search/search_tree.hpp"
#include "time_inserts.hpp"
#include "../balance/red_black.hpp"
#include "../balance/treap.hpp"
#include "../balance/splay.hpp"
#include "../balance/trivial.hpp"

using std::identity;

int main()
{
  using namespace bintrees;

  int min_size = 100000, max_size = 1000000;

  double rb_time, treap_time, splay_time, triv_time;

  cout << '\t' << '\t' << "INSERTIONS" << endl << endl;
  cout << "size" << '\t' << "rb" << '\t' << "treap"
               << '\t' << "splay"     << '\t' << "trivial" << endl;
  cout << "---------------------------------------" << endl;

  for (int size = min_size; size <= max_size; size += min_size) {
    rb_time    = time_inserts<search_tree, int, red_black_balance, int,
                             identity<int>, bintrees::less<int>,
                             simple_gen> (size);
    treap_time = time_inserts<search_tree, int, treap_balance, int,
                              identity<int>, bintrees::less<int>,
                              simple_gen> (size);
    splay_time = time_inserts<search_tree, int, splay_balance, int,
                              identity<int>, bintrees::less<int>,
                              simple_gen> (size);
    triv_time  = time_inserts<search_tree, int, trivial_balance, int,
                              identity<int>, bintrees::less<int>,
                              simple_gen> (size);
    cout << size << '\t' << rb_time << '\t' << treap_time
             << '\t' << splay_time << '\t' << triv_time << endl;
  }
}
```

B.2.3 time_finds.hpp

```
#include <iostream>
#include <time.h>
#include <functional>
#include <algorithm>
#include <vector>
#include "../tree/tree.hpp"
#include "timer.hpp"
#include "comp.hpp"
#include "gen.hpp"
#include "del.hpp"
#include "dist.hpp"

using std::vector;
using std::cout;
using std::endl;
using std::random_shuffle;

using namespace bintrees;

template <
  template <class Val, class Balance, class Key, class Extract, class Comp>
    class Tree,
  class Val, class Balance, class Key, class Extract, class Comp,
  template <class Val> class Generator>

double time_finds(unsigned int size)
{
  typedef Tree<Val, Balance, Key, Extract, Comp> tree_type;
  tree_type x;
  timer tmr;

  vector<Val>    v(size);
  deletor<Val>   del;
  Generator<Val> gen;
  for (int i = 0; i < size; ++i)
    v[i] = gen();
  random_shuffle(v.begin(), v.end());
  for (int i = 0; i < size; ++i)
    x.insert(v[i]);
  tmr.start();
  for (int i = 0; i < size; ++i)
    x.find(v[i]);
  tmr.stop();
  for (int i = 0; i < size; ++i)
    del(v[i]);
  return tmr.total_time();
}
```

B.2.4 time_finds.cpp

```
#include <iostream>
#include "../search/search_tree.hpp"
#include "../search/optimal_string_search_tree.hpp"
#include "../search/simple_string_search_tree.hpp"
#include "time_finds.hpp"
#include "../balance/red_black.hpp"
#include "../balance/treap.hpp"
#include "../balance/splay.hpp"
#include "../balance/trivial.hpp"

int main()
{
  using std::identity;
  using bintrees::less;

  int size = 1000000;

  double search_time, simple_search_time, optimal_search_time;


  cout <<
    "balancer  key search    simple search   optimal search" << endl;
```

```
cout <<
   "------------------------------------------------------" << endl;
search_time = time_finds
                <search_tree, char*,
                 red_black_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



simple_search_time = time_finds
                <simple_string_search_tree, char*,
                 red_black_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



optimal_search_time = time_finds
                       <optimal_string_search_tree, char*,
                        red_black_balance, char*,
                        identity<char*>, less<char*>,
                         binary_gen> (size);

cout << "red_black     " << search_time
     << "              " << simple_search_time
     << "              " << optimal_search_time << endl;

search_time = time_finds
                <search_tree, char*,
                 treap_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



simple_search_time = time_finds
                <simple_string_search_tree, char*,
                 treap_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



optimal_search_time = time_finds
                       <optimal_string_search_tree, char*,
                        treap_balance, char*,
                        identity<char*>, less<char*>,
                         binary_gen> (size);

cout << "treap         " << search_time
     << "              " << simple_search_time
     << "              " << optimal_search_time << endl;

search_time = time_finds
                <search_tree, char*,
                 splay_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



simple_search_time = time_finds
                <simple_string_search_tree, char*,
                 splay_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



optimal_search_time = time_finds
                       <optimal_string_search_tree, char*,
                        splay_balance, char*,
                        identity<char*>, less<char*>,
                         binary_gen> (size);

cout << "splay         " << search_time
     << "              " << simple_search_time
     << "              " << optimal_search_time << endl;

search_time = time_finds
                <search_tree, char*,
                 trivial_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);



simple_search_time = time_finds
                <simple_string_search_tree, char*,
                 trivial_balance, char*,
                 identity<char*>, less<char*>,
                 binary_gen> (size);
```

```
  optimal_search_time = time_finds
                        <optimal_string_search_tree, char*,
                         trivial_balance, char*,
                         identity<char*>, less<char*>,
                          binary_gen> (size);

  cout << "trivial       " << search_time
       << "              " << simple_search_time
       << "              " << optimal_search_time << endl;


}
```