G++ 2.91.57，cygnus\cygwin-b20\include\g++**stl_tree.h** 完整列表

```
/*
 *
 * Copyright (c) 1996,1997
 * Silicon Graphics Computer Systems, Inc.
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Silicon Graphics makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 * Copyright (c) 1994
 * Hewlett-Packard Company
 *
 * Permission to use, copy, modify, distribute and sell this software
 * and its documentation for any purpose is hereby granted without fee,
 * provided that the above copyright notice appear in all copies and
 * that both that copyright notice and this permission notice appear
 * in supporting documentation.  Hewlett-Packard Company makes no
 * representations about the suitability of this software for any
 * purpose.  It is provided "as is" without express or implied warranty.
 *
 *
 */

/* NOTE: This is an internal header file, included by other STL headers.
 *   You should not attempt to use it directly.
 */

#ifndef __SGI_STL_INTERNAL_TREE_H
#define __SGI_STL_INTERNAL_TREE_H

/*
本檔實作Red-black tree（紅-黑樹）class，用以實作 STL 關聯式容器（如set,
multiset, map, multimap）。所用之insertion 和deletion 演算法係以
Cormen, Leiserson 和 Rivest 所著之 Introduction to Algorithms
(MIT Press, 1990) 一書為基礎，唯以下兩點不同：

(1) header 不僅指向 root，也指向紅黑樹的最左節點，以便實作出常數時間之
begin()；並且也指向紅黑樹的最右節點，以便set 相關泛型演算法（如set_union
等等）有線性時間之表現。

(2) 當一個即將被刪除之節點擁有兩個子節點時，它的successor node is
relinked into its place, rather than copied, 如此一來唯一失效
（invalidated）的迭代器就只是那些referring to the deleted node.
```

```
*/

#include <stl_algobase.h>
#include <stl_alloc.h>
#include <stl_construct.h>
#include <stl_function.h>

__STL_BEGIN_NAMESPACE

typedef bool __rb_tree_color_type;
const __rb_tree_color_type __rb_tree_red = false;  // 紅色為 0
const __rb_tree_color_type __rb_tree_black = true; // 黑色為 1

struct __rb_tree_node_base
{
  typedef __rb_tree_color_type color_type;
  typedef __rb_tree_node_base* base_ptr;

  color_type color;    // 節點顏色，非紅即黑。
  base_ptr parent;     // RB 樹的許多操作，必須知道父節點。
  base_ptr left;       // 指向左節點。
  base_ptr right;      // 指向右節點。

  static base_ptr minimum(base_ptr x)
  {
    while (x->left != 0) x = x->left;     // 一直向左走，就會找到最小值，
    return x;                             // 這是二元搜尋樹的特性。
  }

  static base_ptr maximum(base_ptr x)
  {
    while (x->right != 0) x = x->right;   // 一直向右走，就會找到最大值，
    return x;                             // 這是二元搜尋樹的特性。
  }
};

template <class Value>
struct __rb_tree_node : public __rb_tree_node_base
{
  typedef __rb_tree_node<Value>* link_type;
  Value value_field;   // 節點實值
};

struct __rb_tree_base_iterator
{
  typedef __rb_tree_node_base::base_ptr base_ptr;
  typedef bidirectional_iterator_tag iterator_category;
  typedef ptrdiff_t difference_type;
```

```
base_ptr node;  // 它用來與容器之間產生一個連結關係（make a reference）

// 以下其實可實作於 operator++ 內，因為再無他處會呼叫此函式了。
void increment()
{
  if (node->right != 0) {      // 如果有右子節點。狀況(1)
    node = node->right;        // 就向右走
    while (node->left != 0)    // 然後一直往左子樹走到底
      node = node->left;       // 即是解答
  }
  else {                       // 沒有右子節點。狀況(2)
    base_ptr y = node->parent; // 找出父節點
    while (node == y->right) { // 如果現行節點本身是個右子節點，
      node = y;                // 就一直上溯，直到「不為右子節點」止。
      y = y->parent;
    }
    if (node->right != y)      // 「若此時的右子節點不等於此時的父節點」。
      node = y;                // 狀況(3) 此時的父節點即為解答。
                               // 否則此時的node 為解答。狀況(4)
  }
  // 注意，以上判斷「若此時的右子節點不等於此時的父節點」，是為了應付一種
  // 特殊情況：我們欲尋找根節點的下一節點，而恰巧根節點無右子節點。
  // 當然，以上特殊作法必須配合 RB-tree 根節點與特殊之header 之間的
  // 特殊關係。
}

// 以下其實可實作於 operator-- 內，因為再無他處會呼叫此函式了。
void decrement()
{
  if (node->color == __rb_tree_red &&    // 如果是紅節點，且
      node->parent->parent == node)      // 父節點的父節點等於自己，
    node = node->right;                  // 狀況(1) 右子節點即為解答。
  // 以上情況發生於node為header時（亦即 node 為 end() 時）。
  // 注意，header 之右子節點即 mostright，指向整棵樹的 max 節點。
  else if (node->left != 0) {            // 如果有左子節點。狀況(2)
    base_ptr y = node->left;             // 令y指向左子節點
    while (y->right != 0)                // 當y有右子節點時
      y = y->right;                      // 一直往右子節點走到底
    node = y;                            // 最後即為答案
  }
  else {                                 // 既非根節點，亦無左子節點。
    base_ptr y = node->parent;           // 狀況(3) 找出父節點
    while (node == y->left) {            // 當現行節點身為左子節點
      node = y;                          // 一直交替往上走，直到現行節點
      y = y->parent;                     // 不為左子節點
    }
    node = y;                            // 此時之父節點即為答案
  }
}
```

```
};

template <class Value, class Ref, class Ptr>
struct __rb_tree_iterator : public __rb_tree_base_iterator
{
  typedef Value value_type;
  typedef Ref reference;
  typedef Ptr pointer;
  typedef __rb_tree_iterator<Value, Value&, Value*>     iterator;
  typedef __rb_tree_iterator<Value, const Value&, const Value*> const_iterator;
  typedef __rb_tree_iterator<Value, Ref, Ptr>   self;
  typedef __rb_tree_node<Value>* link_type;

  __rb_tree_iterator() {}
  __rb_tree_iterator(link_type x) { node = x; }
  __rb_tree_iterator(const iterator& it) { node = it.node; }

  reference operator*() const { return link_type(node)->value_field; }
#ifndef __SGI_STL_NO_ARROW_OPERATOR
  pointer operator->() const { return &(operator*()); }
#endif /* __SGI_STL_NO_ARROW_OPERATOR */

  self& operator++() { increment(); return *this; }
  self operator++(int) {
    self tmp = *this;
    increment();
    return tmp;
  }

  self& operator--() { decrement(); return *this; }
  self operator--(int) {
    self tmp = *this;
    decrement();
    return tmp;
  }
};

inline bool operator==(const __rb_tree_base_iterator& x,
                       const __rb_tree_base_iterator& y) {
  return x.node == y.node;
  // 兩個迭代器相等，意指其所指的節點相等。
}

inline bool operator!=(const __rb_tree_base_iterator& x,
                       const __rb_tree_base_iterator& y) {
  return x.node != y.node;
  // 兩個迭代器不等，意指其所指的節點不等。
}
```

```
#ifndef __STL_CLASS_PARTIAL_SPECIALIZATION

inline bidirectional_iterator_tag
iterator_category(const __rb_tree_base_iterator&) {
  return bidirectional_iterator_tag();
}

inline __rb_tree_base_iterator::difference_type*
distance_type(const __rb_tree_base_iterator&) {
  return (__rb_tree_base_iterator::difference_type*) 0;
}

template <class Value, class Ref, class Ptr>
inline Value* value_type(const __rb_tree_iterator<Value, Ref, Ptr>&) {
  return (Value*) 0;
}

#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */

// 以下都是全域函式：__rb_tree_rotate_left(), __rb_tree_rotate_right(),
// __rb_tree_rebalance(), __rb_tree_rebalance_for_erase()

// 新節點必為紅節點。如果安插處之父節點亦為紅節點，就違反紅黑樹規則，此時必須
// 做樹形旋轉（及顏色改變，在程式它處）。
inline void
__rb_tree_rotate_left(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
  // x 為旋轉點
  __rb_tree_node_base* y = x->right; // 令y 為旋轉點的右子節點
  x->right = y->left;
  if (y->left !=0)
    y->left->parent = x;        // 別忘了回馬槍設定父節點
  y->parent = x->parent;

  // 令 y 完全頂替 x 的地位（必須將 x 對其父節點的關係完全接收過來）
  if (x == root)                    // x 為根節點
    root = y;
  else if (x == x->parent->left)    // x 為其父節點的左子節點
    x->parent->left = y;
  else                              // x 為其父節點的右子節點
    x->parent->right = y;
  y->left = x;
  x->parent = y;
}

// 新節點必為紅節點。如果安插處之父節點亦為紅節點，就違反紅黑樹規則，此時必須
// 做樹形旋轉（及顏色改變，在程式它處）。
inline void
__rb_tree_rotate_right(__rb_tree_node_base* x, __rb_tree_node_base*& root)
```

```
{
  // x 為旋轉點
  __rb_tree_node_base* y = x->left; // y 為旋轉點的左子節點
  x->left = y->right;
  if (y->right != 0)
    y->right->parent = x;   // 別忘了回馬槍設定父節點
  y->parent = x->parent;

  // 令 y 完全頂替 x 的地位（必須將 x 對其父節點的關係完全接收過來）
  if (x == root)                    // x 為根節點
    root = y;
  else if (x == x->parent->right)   // x 為其父節點的右子節點
    x->parent->right = y;
  else                              // x 為其父節點的左子節點
    x->parent->left = y;
  y->right = x;
  x->parent = y;
}

// 重新令樹形平衡（改變顏色及旋轉樹形）
// 參數一為新增節點，參數二為 root
inline void
__rb_tree_rebalance(__rb_tree_node_base* x, __rb_tree_node_base*& root)
{
  x->color = __rb_tree_red;     // 新節點必為紅
  while (x != root && x->parent->color == __rb_tree_red) { // 父節點為紅
    if (x->parent == x->parent->parent->left) { // 父節點為祖父節點之左子節點
      __rb_tree_node_base* y = x->parent->parent->right;// 令y 為伯父節點
      if (y && y->color == __rb_tree_red) {         // 伯父節點存在，且為紅
        x->parent->color = __rb_tree_black;         // 更改父節點為黑
        y->color = __rb_tree_black;                 // 更改伯父節點為黑
        x->parent->parent->color = __rb_tree_red;   // 更改祖父節點為紅
        x = x->parent->parent;
      }
      else {  // 無伯父節點，或伯父節點為黑
        if (x == x->parent->right) { // 如果新節點為父節點之右子節點
          x = x->parent;
          __rb_tree_rotate_left(x, root); // 第一參數為左旋點
        }
        x->parent->color = __rb_tree_black;     // 改變顏色
        x->parent->parent->color = __rb_tree_red;
        __rb_tree_rotate_right(x->parent->parent, root); // 第一參數為右旋點
      }
    }
    else {    // 父節點為祖父節點之右子節點
      __rb_tree_node_base* y = x->parent->parent->left; // 令y 為伯父節點
      if (y && y->color == __rb_tree_red) {         // 有伯父節點，且為紅
        x->parent->color = __rb_tree_black;         // 更改父節點為黑
        y->color = __rb_tree_black;                 // 更改伯父節點為黑
```

```
        x->parent->parent->color = __rb_tree_red;   // 更改祖父節點為紅
        x = x->parent->parent;   // 準備繼續往上層檢查...
      }
      else {  // 無伯父節點，或伯父節點為黑
        if (x == x->parent->left) {  // 如果新節點為父節點之左子節點
          x = x->parent;
          __rb_tree_rotate_right(x, root);      // 第一參數為右旋點
        }
        x->parent->color = __rb_tree_black;    // 改變顏色
        x->parent->parent->color = __rb_tree_red;
        __rb_tree_rotate_left(x->parent->parent, root); // 第一參數為左旋點
      }
    }
  }// while 結束
  root->color = __rb_tree_black;     // 根節點永遠為黑
}

inline __rb_tree_node_base*
__rb_tree_rebalance_for_erase(__rb_tree_node_base* z,
                              __rb_tree_node_base*& root,
                              __rb_tree_node_base*& leftmost,
                              __rb_tree_node_base*& rightmost)
{
  __rb_tree_node_base* y = z;
  __rb_tree_node_base* x = 0;
  __rb_tree_node_base* x_parent = 0;
  if (y->left == 0)             // z has at most one non-null child. y == z.
    x = y->right;               // x might be null.
  else
    if (y->right == 0)          // z has exactly one non-null child.  y == z.
      x = y->left;              // x is not null.
    else {                      // z has two non-null children.  Set y to
      y = y->right;             //   z's successor.  x might be null.
      while (y->left != 0)
        y = y->left;
      x = y->right;
    }
  if (y != z) {                 // relink y in place of z.  y is z's successor
    z->left->parent = y;
    y->left = z->left;
    if (y != z->right) {
      x_parent = y->parent;
      if (x) x->parent = y->parent;
      y->parent->left = x;      // y must be a left child
      y->right = z->right;
      z->right->parent = y;
    }
    else
      x_parent = y;
```

```
    if (root == z)
      root = y;
    else if (z->parent->left == z)
      z->parent->left = y;
    else
      z->parent->right = y;
    y->parent = z->parent;
    __STD::swap(y->color, z->color);
    y = z;
    // y now points to node to be actually deleted
  }
  else {                        // y == z
    x_parent = y->parent;
    if (x) x->parent = y->parent;
    if (root == z)
      root = x;
    else
      if (z->parent->left == z)
        z->parent->left = x;
      else
        z->parent->right = x;
    if (leftmost == z)
      if (z->right == 0)        // z->left must be null also
        leftmost = z->parent;
    // makes leftmost == header if z == root
      else
        leftmost = __rb_tree_node_base::minimum(x);
    if (rightmost == z)
      if (z->left == 0)         // z->right must be null also
        rightmost = z->parent;
    // makes rightmost == header if z == root
      else                      // x == z->left
        rightmost = __rb_tree_node_base::maximum(x);
  }
  if (y->color != __rb_tree_red) {
    while (x != root && (x == 0 || x->color == __rb_tree_black))
      if (x == x_parent->left) {
        __rb_tree_node_base* w = x_parent->right;
        if (w->color == __rb_tree_red) {
          w->color = __rb_tree_black;
          x_parent->color = __rb_tree_red;
          __rb_tree_rotate_left(x_parent, root);
          w = x_parent->right;
        }
        if ((w->left == 0 || w->left->color == __rb_tree_black) &&
            (w->right == 0 || w->right->color == __rb_tree_black)) {
          w->color = __rb_tree_red;
          x = x_parent;
          x_parent = x_parent->parent;
```

```
        } else {
          if (w->right == 0 || w->right->color == __rb_tree_black) {
            if (w->left) w->left->color = __rb_tree_black;
            w->color = __rb_tree_red;
            __rb_tree_rotate_right(w, root);
            w = x_parent->right;
          }
          w->color = x_parent->color;
          x_parent->color = __rb_tree_black;
          if (w->right) w->right->color = __rb_tree_black;
          __rb_tree_rotate_left(x_parent, root);
          break;
        }
      } else {                  // same as above, with right <-> left.
        __rb_tree_node_base* w = x_parent->left;
        if (w->color == __rb_tree_red) {
          w->color = __rb_tree_black;
          x_parent->color = __rb_tree_red;
          __rb_tree_rotate_right(x_parent, root);
          w = x_parent->left;
        }
        if ((w->right == 0 || w->right->color == __rb_tree_black) &&
            (w->left == 0 || w->left->color == __rb_tree_black)) {
          w->color = __rb_tree_red;
          x = x_parent;
          x_parent = x_parent->parent;
        } else {
          if (w->left == 0 || w->left->color == __rb_tree_black) {
            if (w->right) w->right->color = __rb_tree_black;
            w->color = __rb_tree_red;
            __rb_tree_rotate_left(w, root);
            w = x_parent->left;
          }
          w->color = x_parent->color;
          x_parent->color = __rb_tree_black;
          if (w->left) w->left->color = __rb_tree_black;
          __rb_tree_rotate_right(x_parent, root);
          break;
        }
      }
    if (x) x->color = __rb_tree_black;
  }
  return y;
}

template <class Key, class Value, class KeyOfValue, class Compare,
          class Alloc = alloc>
class rb_tree {
protected:
```

```
  typedef void* void_pointer;
  typedef __rb_tree_node_base* base_ptr;
  typedef __rb_tree_node<Value> rb_tree_node;
  typedef simple_alloc<rb_tree_node, Alloc> rb_tree_node_allocator;
  typedef __rb_tree_color_type color_type;
public:
  // 注意，沒有定義 iterator（喔，不，定義在後面）
  typedef Key key_type;
  typedef Value value_type;
  typedef value_type* pointer;
  typedef const value_type* const_pointer;
  typedef value_type& reference;
  typedef const value_type& const_reference;
  typedef rb_tree_node* link_type;
  typedef size_t size_type;
  typedef ptrdiff_t difference_type;
protected:
  link_type get_node() { return rb_tree_node_allocator::allocate(); }
  void put_node(link_type p) { rb_tree_node_allocator::deallocate(p); }

  link_type create_node(const value_type& x) {
    link_type tmp = get_node();           // 配置空間
    __STL_TRY {
      construct(&tmp->value_field, x);    // 建構內容
    }
    __STL_UNWIND(put_node(tmp));
    return tmp;
  }

  link_type clone_node(link_type x) {     // 複製一個節點（的值和色）
    link_type tmp = create_node(x->value_field);
    tmp->color = x->color;
    tmp->left = 0;
    tmp->right = 0;
    return tmp;
  }

  void destroy_node(link_type p) {
    destroy(&p->value_field);         // 解構內容
    put_node(p);                      // 釋還記憶體
  }

protected:
  // RB-tree 只以三筆資料表現。
  size_type node_count; // 追蹤記錄樹的大小（節點數量）
  link_type header;
  Compare key_compare; // 節點間的鍵值大小比較準則。應該會是個 function object。

  // 以下三個函式用來方便取得 header 的成員
```

```
  link_type& root() const { return (link_type&) header->parent; }
  link_type& leftmost() const { return (link_type&) header->left; }
  link_type& rightmost() const { return (link_type&) header->right; }

  // 以下六個函式用來方便取得節點 x 的成員
  static link_type& left(link_type x) { return (link_type&)(x->left); }
  static link_type& right(link_type x) { return (link_type&)(x->right); }
  static link_type& parent(link_type x) { return (link_type&)(x->parent); }
  static reference value(link_type x) { return x->value_field; }
  static const Key& key(link_type x) { return KeyOfValue()(value(x)); }
  static color_type& color(link_type x) { return (color_type&)(x->color); }

  // 以下六個函式用來方便取得節點 x 的成員
  static link_type& left(base_ptr x) { return (link_type&)(x->left); }
  static link_type& right(base_ptr x) { return (link_type&)(x->right); }
  static link_type& parent(base_ptr x) { return (link_type&)(x->parent); }
  static reference value(base_ptr x) { return ((link_type)x)->value_field; }
  static const Key& key(base_ptr x) { return KeyOfValue()(value(link_type(x)));}
  static color_type& color(base_ptr x) { return (color_type&)(link_type(x)->color); }

  // 求取極大值和極小值。node class 有實作此功能，交給它們完成即可。
  static link_type minimum(link_type x) {
    return (link_type)  __rb_tree_node_base::minimum(x);
  }
  static link_type maximum(link_type x) {
    return (link_type) __rb_tree_node_base::maximum(x);
  }

public:
  typedef __rb_tree_iterator<value_type, reference, pointer> iterator;
  typedef __rb_tree_iterator<value_type, const_reference, const_pointer>
          const_iterator;

#ifdef __STL_CLASS_PARTIAL_SPECIALIZATION
  typedef reverse_iterator<const_iterator> const_reverse_iterator;
  typedef reverse_iterator<iterator> reverse_iterator;
#else /* __STL_CLASS_PARTIAL_SPECIALIZATION */
  typedef reverse_bidirectional_iterator<iterator, value_type, reference,
                                         difference_type>
          reverse_iterator;
  typedef reverse_bidirectional_iterator<const_iterator, value_type,
                                         const_reference, difference_type>
          const_reverse_iterator;
#endif /* __STL_CLASS_PARTIAL_SPECIALIZATION */
private:
  iterator __insert(base_ptr x, base_ptr y, const value_type& v);
  link_type __copy(link_type x, link_type p);
  void __erase(link_type x);
  void init() {
```

```
    header = get_node();    // 產生一個節點空間，令 header 指向它
    color(header) = __rb_tree_red; // 令 header 為紅色，用來區分 header
                                   // 和 root（在 iterator.operator++ 中）
    root() = 0;
    leftmost() = header;    // 令 header 的左子節點為自己。
    rightmost() = header;   // 令 header 的右子節點為自己。
  }
public:
                            // allocation/deallocation
  rb_tree(const Compare& comp = Compare())
    : node_count(0), key_compare(comp) { init(); }

  // 以另一個 rb_tree 物件 x 為初值
  rb_tree(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x)
    : node_count(0), key_compare(x.key_compare)
  {
    header = get_node();    // 產生一個節點空間，令 header 指向它
    color(header) = __rb_tree_red;    // 令 header 為紅色
    if (x.root() == 0) {    //  如果 x 是個空白樹
      root() = 0;
      leftmost() = header;  // 令 header 的左子節點為自己。
      rightmost() = header; // 令 header 的右子節點為自己。
    }
    else {    //  x 不是一個空白樹
      __STL_TRY {
        root() = __copy(x.root(), header);      // ???
      }
      __STL_UNWIND(put_node(header));
      leftmost() = minimum(root());  // 令 header 的左子節點為最小節點
      rightmost() = maximum(root()); // 令 header 的右子節點為最大節點
    }
    node_count = x.node_count;
  }
  ~rb_tree() {
    clear();
    put_node(header);
  }
  rb_tree<Key, Value, KeyOfValue, Compare, Alloc>&
  operator=(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x);

public:
                            // accessors:
  Compare key_comp() const { return key_compare; }
  iterator begin() { return leftmost(); }      // RB 樹的起頭為最左（最小）節點處
  const_iterator begin() const { return leftmost(); }
  iterator end() { return header; }  // RB 樹的終點為 header所指處
  const_iterator end() const { return header; }
  reverse_iterator rbegin() { return reverse_iterator(end()); }
  const_reverse_iterator rbegin() const {
```

```
    return const_reverse_iterator(end());
  }
  reverse_iterator rend() { return reverse_iterator(begin()); }
  const_reverse_iterator rend() const {
    return const_reverse_iterator(begin());
  }
  bool empty() const { return node_count == 0; }
  size_type size() const { return node_count; }
  size_type max_size() const { return size_type(-1); }

  void swap(rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& t) {
    // RB-tree 只以三個資料成員表現。所以互換兩個 RB-trees時，
    // 只需將這三個成員互換即可。
    __STD::swap(header, t.header);
    __STD::swap(node_count, t.node_count);
    __STD::swap(key_compare, t.key_compare);
  }

public:
                                // insert/erase
  // 將 x 安插到 RB-tree 中（保持節點值獨一無二）。
  pair<iterator,bool> insert_unique(const value_type& x);
  // 將 x 安插到 RB-tree 中（允許節點值重複）。
  iterator insert_equal(const value_type& x);

  iterator insert_unique(iterator position, const value_type& x);
  iterator insert_equal(iterator position, const value_type& x);

#ifdef __STL_MEMBER_TEMPLATES
  template <class InputIterator>
  void insert_unique(InputIterator first, InputIterator last);
  template <class InputIterator>
  void insert_equal(InputIterator first, InputIterator last);
#else /* __STL_MEMBER_TEMPLATES */
  void insert_unique(const_iterator first, const_iterator last);
  void insert_unique(const value_type* first, const value_type* last);
  void insert_equal(const_iterator first, const_iterator last);
  void insert_equal(const value_type* first, const value_type* last);
#endif /* __STL_MEMBER_TEMPLATES */

  void erase(iterator position);
  size_type erase(const key_type& x);
  void erase(iterator first, iterator last);
  void erase(const key_type* first, const key_type* last);
  void clear() {
    if (node_count != 0) {
      __erase(root());
      leftmost() = header;
      root() = 0;
```

```
      rightmost() = header;
      node_count = 0;
    }
  }

public:
                              // 集合（set）的各種操作行為：
  iterator find(const key_type& x);
  const_iterator find(const key_type& x) const;
  size_type count(const key_type& x) const;
  iterator lower_bound(const key_type& x);
  const_iterator lower_bound(const key_type& x) const;
  iterator upper_bound(const key_type& x);
  const_iterator upper_bound(const key_type& x) const;
  pair<iterator,iterator> equal_range(const key_type& x);
  pair<const_iterator, const_iterator> equal_range(const key_type& x) const;

public:
                              // Debugging.
  bool __rb_verify() const;
};

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline bool operator==(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x,
                       const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& y) {
  return x.size() == y.size() && equal(x.begin(), x.end(), y.begin());
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline bool operator<(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x,
                      const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& y) {
  return lexicographical_compare(x.begin(), x.end(), y.begin(), y.end());
}

#ifdef __STL_FUNCTION_TMPL_PARTIAL_ORDER

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline void swap(rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x,
                 rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& y) {
  x.swap(y);
}

#endif /* __STL_FUNCTION_TMPL_PARTIAL_ORDER */


template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>&
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::
operator=(const rb_tree<Key, Value, KeyOfValue, Compare, Alloc>& x) {
```

```
  if (this != &x) {
                                // Note that Key may be a constant type.
    clear();
    node_count = 0;
    key_compare = x.key_compare;
    if (x.root() == 0) {
      root() = 0;
      leftmost() = header;
      rightmost() = header;
    }
    else {
      root() = __copy(x.root(), header);
      leftmost() = minimum(root());
      rightmost() = maximum(root());
      node_count = x.node_count;
    }
  }
  return *this;
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::
__insert(base_ptr x_, base_ptr y_, const Value& v) {
// 參數x_ 為新值安插點，參數y_ 為安插點之父節點，參數v 為新值。
  link_type x = (link_type) x_;
  link_type y = (link_type) y_;
  link_type z;

  // key_compare 是鍵值大小比較準則。應該會是個 function object。
  if (y == header || x != 0 || key_compare(KeyOfValue()(v), key(y))) {
    z = create_node(v);  // 產生一個新節點
    left(y) = z;         // 這使得當 y 即為 header時，leftmost() = z
    if (y == header) {
      root() = z;
      rightmost() = z;
    }
    else if (y == leftmost())    // 如果y為最左節點
      leftmost() = z;            // 維護leftmost()，使它永遠指向最左節點
  }
  else {
    z = create_node(v);          // 產生一個新節點
    right(y) = z;                // 令新節點成為安插點之父節點 y 的右子節點
    if (y == rightmost())
      rightmost() = z;           // 維護rightmost()，使它永遠指向最右節點
  }
  parent(z) = y;       // 設定新節點的父節點
  left(z) = 0;         // 設定新節點的左子節點
  right(z) = 0;        // 設定新節點的右子節點
```

```
                          // 新節點的顏色將在 __rb_tree_rebalance() 設定（並調整）
  __rb_tree_rebalance(z, header->parent);// 參數一為新增節點，參數二為 root
  ++node_count;          // 節點數累加
  return iterator(z);  // 傳回一個迭代器，指向新增節點
}

// 安插新值；節點鍵值允許重複。
// 注意，傳回值是一個 RB-tree 迭代器，指向新增節點
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::insert_equal(const Value& v)
{
  link_type y = header;
  link_type x = root();     // 從根節點開始
  while (x != 0) {          // 從根節點開始，往下尋找適當的安插點
    y = x;
    x = key_compare(KeyOfValue()(v), key(x)) ? left(x) : right(x);
    // 以上，遇「大」則往左，遇「小於或等於」則往右
  }
  return __insert(x, y, v);
}

// 安插新值；節點鍵值不允許重複，若重複則安插無效。
// 注意，傳回值是個pair，第一元素是個 RB-tree 迭代器，指向新增節點，
// 第二元素表示安插成功與否。
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
pair<typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator, bool>
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::insert_unique(const Value& v)
{
  link_type y = header;
  link_type x = root();     // 從根節點開始
  bool comp = true;
  while (x != 0) {          // 從根節點開始，往下尋找適當的安插點
    y = x;
    comp = key_compare(KeyOfValue()(v), key(x)); // v 鍵值小於目前節點之鍵值？
    x = comp ? left(x) : right(x);    // 遇「大」則往左，遇「小於或等於」則往右
  }
  // 離開 while 迴圈之後，y 所指即安插點之父節點（此時的它必為葉節點）

  iterator j = iterator(y);   // 令迭代器j指向安插點之父節點 y
  if (comp)  // 如果離開 while 迴圈時 comp 為真（表示遇「大」，將安插於左側）
    if (j == begin())   // 如果安插點之父節點為最左節點
      return pair<iterator,bool>(__insert(x, y, v), true);
      // 以上，x 為安插點，y 為安插點之父節點，v 為新值。
    else // 否則（安插點之父節點不為最左節點）
      --j;    // 調整 j，回頭準備測試...
  if (key_compare(key(j.node), KeyOfValue()(v)))
    // 小於新值（表示遇「小」，將安插於右側）
    return pair<iterator,bool>(__insert(x, y, v), true);
```

```
  // 進行至此，表示新值一定與樹中鍵值重複，那麼就不該插入新值。
  return pair<iterator,bool>(j, false);
}


template <class Key, class Val, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Val, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Val, KeyOfValue, Compare, Alloc>::insert_unique(iterator position,
                                                             const Val& v) {
  if (position.node == header->left) // begin()
    if (size() > 0 && key_compare(KeyOfValue()(v), key(position.node)))
      return __insert(position.node, position.node, v);
  // first argument just needs to be non-null
    else
      return insert_unique(v).first;
  else if (position.node == header) // end()
    if (key_compare(key(rightmost()), KeyOfValue()(v)))
      return __insert(0, rightmost(), v);
    else
      return insert_unique(v).first;
  else {
    iterator before = position;
    --before;
    if (key_compare(key(before.node), KeyOfValue()(v))
        && key_compare(KeyOfValue()(v), key(position.node)))
      if (right(before.node) == 0)
        return __insert(0, before.node, v);
      else
        return __insert(position.node, position.node, v);
    // first argument just needs to be non-null
    else
      return insert_unique(v).first;
  }
}

template <class Key, class Val, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Val, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Val, KeyOfValue, Compare, Alloc>::insert_equal(iterator position,
                                                            const Val& v) {
  if (position.node == header->left) // begin()
    if (size() > 0 && key_compare(KeyOfValue()(v), key(position.node)))
      return __insert(position.node, position.node, v);
  // first argument just needs to be non-null
    else
      return insert_equal(v);
  else if (position.node == header) // end()
    if (!key_compare(KeyOfValue()(v), key(rightmost())))
      return __insert(0, rightmost(), v);
```

```
    else
      return insert_equal(v);
  else {
    iterator before = position;
    --before;
    if (!key_compare(KeyOfValue()(v), key(before.node))
        && !key_compare(key(position.node), KeyOfValue()(v)))
      if (right(before.node) == 0)
        return __insert(0, before.node, v);
      else
        return __insert(position.node, position.node, v);
    // first argument just needs to be non-null
    else
      return insert_equal(v);
  }
}

#ifdef __STL_MEMBER_TEMPLATES

template <class K, class V, class KoV, class Cmp, class Al> template<class II>
void rb_tree<K, V, KoV, Cmp, Al>::insert_equal(II first, II last) {
  for ( ; first != last; ++first)
    insert_equal(*first);
}

template <class K, class V, class KoV, class Cmp, class Al> template<class II>
void rb_tree<K, V, KoV, Cmp, Al>::insert_unique(II first, II last) {
  for ( ; first != last; ++first)
    insert_unique(*first);
}

#else /* __STL_MEMBER_TEMPLATES */

template <class K, class V, class KoV, class Cmp, class Al>
void
rb_tree<K, V, KoV, Cmp, Al>::insert_equal(const V* first, const V* last) {
  for ( ; first != last; ++first)
    insert_equal(*first);
}

template <class K, class V, class KoV, class Cmp, class Al>
void
rb_tree<K, V, KoV, Cmp, Al>::insert_equal(const_iterator first,
                                          const_iterator last) {
  for ( ; first != last; ++first)
    insert_equal(*first);
}

template <class K, class V, class KoV, class Cmp, class A>
```

```
void
rb_tree<K, V, KoV, Cmp, A>::insert_unique(const V* first, const V* last) {
  for ( ; first != last; ++first)
    insert_unique(*first);
}

template <class K, class V, class KoV, class Cmp, class A>
void
rb_tree<K, V, KoV, Cmp, A>::insert_unique(const_iterator first,
                                          const_iterator last) {
  for ( ; first != last; ++first)
    insert_unique(*first);
}

#endif /* __STL_MEMBER_TEMPLATES */

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline void
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::erase(iterator position) {
  link_type y = (link_type) __rb_tree_rebalance_for_erase(position.node,
                                                          header->parent,
                                                          header->left,
                                                          header->right);
  destroy_node(y);
  --node_count;
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::size_type
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::erase(const Key& x) {
  pair<iterator,iterator> p = equal_range(x);
  size_type n = 0;
  distance(p.first, p.second, n);
  erase(p.first, p.second);
  return n;
}

template <class K, class V, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<K, V, KeyOfValue, Compare, Alloc>::link_type
rb_tree<K, V, KeyOfValue, Compare, Alloc>::__copy(link_type x, link_type p) {
                                // structural copy.  x and p must be non-null.
  link_type top = clone_node(x);
  top->parent = p;

  __STL_TRY {
    if (x->right)
      top->right = __copy(right(x), top);
    p = top;
    x = left(x);
```

```
    while (x != 0) {
      link_type y = clone_node(x);
      p->left = y;
      y->parent = p;
      if (x->right)
        y->right = __copy(right(x), y);
      p = y;
      x = left(x);
    }
  }
  __STL_UNWIND(__erase(top));

  return top;
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
void rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::__erase(link_type x) {
                                // erase without rebalancing
  while (x != 0) {
    __erase(right(x));
    link_type y = left(x);
    destroy_node(x);
    x = y;
  }
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
void rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::erase(iterator first,
                                                            iterator last) {
  if (first == begin() && last == end())
    clear();
  else
    while (first != last) erase(first++);
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
void rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::erase(const Key* first,
                                                            const Key* last) {
  while (first != last) erase(*first++);
}

// 尋找 RB 樹中是否有鍵值為 k 的節點
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::find(const Key& k) {
  link_type y = header;        // Last node which is not less than k.
  link_type x = root();        // Current node.
```

```
  while (x != 0)
    // 以下，key_compare 是節點鍵值大小比較準則。應該會是個 function object。
    if (!key_compare(key(x), k))
      // 進行到這裡，表示 x 鍵值大於 k。遇到大值就向左走。
      y = x, x = left(x);   // 注意語法!
    else
      // 進行到這裡，表示 x 鍵值小於 k。遇到小值就向右走。
      x = right(x);

  iterator j = iterator(y);
  return (j == end() || key_compare(k, key(j.node))) ? end() : j;
}

// 尋找 RB 樹中是否有鍵值為 k 的節點
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::const_iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::find(const Key& k) const {
  link_type y = header; /* Last node which is not less than k. */
  link_type x = root(); /* Current node. */

  while (x != 0) {
    // 以下，key_compare 是節點鍵值大小比較準則。應該會是個 function object。
    if (!key_compare(key(x), k))
      // 進行到這裡，表示 x 鍵值大於 k。遇到大值就向左走。
      y = x, x = left(x);   // 注意語法!
    else
      // 進行到這裡，表示 x 鍵值小於 k。遇到小值就向右走。
      x = right(x);
  }
  const_iterator j = const_iterator(y);
  return (j == end() || key_compare(k, key(j.node))) ? end() : j;
}

// 計算 RB 樹中鍵值為 k 的節點個數
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::size_type
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::count(const Key& k) const {
  pair<const_iterator, const_iterator> p = equal_range(k);
  size_type n = 0;
  distance(p.first, p.second, n);
  return n;
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::lower_bound(const Key& k) {
  link_type y = header; /* Last node which is not less than k. */
  link_type x = root(); /* Current node. */
```

```
  while (x != 0)
    if (!key_compare(key(x), k))
      y = x, x = left(x);
    else
      x = right(x);

  return iterator(y);
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::const_iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::lower_bound(const Key& k) const {
  link_type y = header; /* Last node which is not less than k. */
  link_type x = root(); /* Current node. */

  while (x != 0)
    if (!key_compare(key(x), k))
      y = x, x = left(x);
    else
      x = right(x);

  return const_iterator(y);
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::upper_bound(const Key& k) {
  link_type y = header; /* Last node which is greater than k. */
  link_type x = root(); /* Current node. */

   while (x != 0)
     if (key_compare(k, key(x)))
       y = x, x = left(x);
     else
       x = right(x);

   return iterator(y);
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::const_iterator
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::upper_bound(const Key& k) const {
  link_type y = header; /* Last node which is greater than k. */
  link_type x = root(); /* Current node. */

   while (x != 0)
     if (key_compare(k, key(x)))
       y = x, x = left(x);
     else
```

```
      x = right(x);

    return const_iterator(y);
}

template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
inline pair<typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator,
            typename rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::iterator>
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::equal_range(const Key& k) {
  return pair<iterator, iterator>(lower_bound(k), upper_bound(k));
}

template <class Key, class Value, class KoV, class Compare, class Alloc>
inline pair<typename rb_tree<Key, Value, KoV, Compare, Alloc>::const_iterator,
            typename rb_tree<Key, Value, KoV, Compare, Alloc>::const_iterator>
rb_tree<Key, Value, KoV, Compare, Alloc>::equal_range(const Key& k) const {
  return pair<const_iterator,const_iterator>(lower_bound(k), upper_bound(k));
}

// 計算從 node 至 root 路徑中的黑節點數量。
inline int __black_count(__rb_tree_node_base* node, __rb_tree_node_base* root)
{
  if (node == 0)
    return 0;
  else {
    int bc = node->color == __rb_tree_black ? 1 : 0;
    if (node == root)
      return bc;
    else
      return bc + __black_count(node->parent, root); // 累加
  }
}

// 驗證己身這棵樹是否符合 RB 樹的條件
template <class Key, class Value, class KeyOfValue, class Compare, class Alloc>
bool
rb_tree<Key, Value, KeyOfValue, Compare, Alloc>::__rb_verify() const
{
  // 空樹，符合RB樹標準
  if (node_count == 0 || begin() == end())
    return node_count == 0 && begin() == end() &&
      header->left == header && header->right == header;

  // 最左（葉）節點至 root 路徑內的黑節點數
  int len = __black_count(leftmost(), root());
  // 以下走訪整個RB樹，針對每個節點（從最小到最大）...
  for (const_iterator it = begin(); it != end(); ++it) {
    link_type x = (link_type) it.node; // __rb_tree_base_iterator::node
    link_type L = left(x);       // 這是左子節點
```

```
    link_type R = right(x);      // 這是右子節點

    if (x->color == __rb_tree_red)
      if ((L && L->color == __rb_tree_red) ||
          (R && R->color == __rb_tree_red))
        return false;   // 父子節點同為紅色，不符合 RB 樹的要求。

    if (L && key_compare(key(x), key(L))) // 目前節點的鍵值小於左子節點鍵值
      return false;          // 不符合二元搜尋樹的要求。
    if (R && key_compare(key(R), key(x))) // 目前節點的鍵值大於右子節點鍵值
      return false;          // 不符合二元搜尋樹的要求。

    // 「葉節點至 root」路徑內的黑節點數，與「最左節點至 root」路徑內的黑節點數不同。
    // 這不符合 RB 樹的要求。
    if (!L && !R && __black_count(x, root()) != len)
      return false;
  }

  if (leftmost() != __rb_tree_node_base::minimum(root()))
    return false; // 最左節點不為最小節點，不符合二元搜尋樹的要求。
  if (rightmost() != __rb_tree_node_base::maximum(root()))
    return false; // 最右節點不為最大節點，不符合二元搜尋樹的要求。

  return true;
}

__STL_END_NAMESPACE

#endif /* __SGI_STL_INTERNAL_TREE_H */

// Local Variables:
// mode:C++
// End:
```