

```

/* bstree */
/* lang=C++11 */

/* binary search tree */

#ifdef CF_DEBUGS
#define CF_DEBUGS 0 /* compile-time debugging */
#endif

/* revision history:

    = 2013-03-03, David A>D> Morano
    Originally written for Rightcore Network Services.

*/

/* Copyright ' 2013 David A>D> Morano. All rights reserved. */

/*****

    Binary-Search-Tree (without rebalancing)

    Notes:

    Q. How do we handle iterative traversal?

    A. There are at least three popular ways to handle iterative traversal:

    1. maintain a stack of pointers to previous node at each level of the
       tree; space would be O(depth)

    2. use a so-called "threaded" tree structure where each dangling
       right-side node (with no additional right-child) actually
       has a pointer to its in-order successor

    3. use the "parent" method where each node has a pointer to its own
       parent; we use this method below

    The "parent" scheme:

    If we find that there is no right-child (from our current node), we
    traverse back up to our parent, checking (at each step) if we (the
    current node) is equal to the right-child node of our parent. If we
    arrive at our parent from its left-child, then our parent becomes the
    next current node. If we arrive at our parent from its right-child node,
    then we continue to go up in the tree until we either arrive at our
    parent from a left-child or we reach the top of the tree (the parent
    itself has no parent). At any node, we try to go down a right-child (if
    it exists) and then its left-child, if it exists.

    This, so-called, "parent" scheme uses up an extra pointer-amount of
    space in each node, O(n) more space, but even though the "stack" scheme
    only uses up O(depth), we prefer the "parent" scheme since we do not
    have to go through the trouble of instantiating a stack in the first
    place.

    For reference, here is a stack solution (pseudo code):

    class iterator {
        stack<node> stack = new stack<node> ;
        node n ;
        iterator(node an) : n(an) { } ;
        iterator &findnext() {
            iterator it(nullptr) ;

```

```

        while (n != NULL) {
            stack.push(n) ;
            n = n->left ;
        }
        n = stack.pop() ;
        it = n ;
        n = n->right ;
        return it ;
    } ;
} ;

```

*****/

```

#ifndef BSTREE_INCLUDE
#define BSTREE_INCLUDE 1

```

```

#include <envstandards.h>          /* MUST be first to configure */
#include <sys/types.h>
#include <limits.h>
#include <cinttypes>
#include <new>
#include <initializer_list>
#include <algorithm>
#include <functional>
#include <stack>
#include <vector>
#include <vsystem.h>
#include <localmisc.h>

```

/* external subroutines */

```

#if CF_DEBUGS
extern "C" int  debugprintf(cchar *,...) ;
extern "C" int  strlinelen(cchar *,cchar *,int) ;
#endif

```

/* local structures */

```

template <typename T,typename Comp = std::less<T>>
class bstree ;

```

```

template <typename T,typename Comp = std::less<T>>
class bstree_iter ;

```

```

template <typename T,typename Comp = std::less<T>>
class bstree_node {
    bstree_node<T,Comp>      *parent = NULL ;
    bstree_node<T,Comp>      *left = NULL ;
    bstree_node<T,Comp>      *right = NULL ;
    T                          val ;
    void SetVal(const T v) {
        val = v ;
    } ;
public:
    bstree_node(T av) : val(av) {
    } ;
    bstree_node(const bstree_node<T> &other) = delete ;
    bstree_node &operator = (const bstree_node<T> &other) = delete ;
    ~bstree_node() {
    } ;
    friend bstree<T,Comp> ;
    friend bstree_iter<T,Comp> ;

```

```

} ; /* end class (bstree_node) */

template <typename T,typename Comp>
class bstree_iter {
    typedef bstree_node<T,Comp>      nodetype ;
    bstree_node<T,Comp>      *n = NULL ;
    mutable T                defval ;
    bstree_iter<T,Comp>      &findnext(int) ;
    typedef bstree_iter      bit ;

public:
    bstree_iter() { } ;
    bstree_iter(bstree_node<T,Comp>* an) : n(an) { } ;
    bstree_iter(const bstree_iter<T,Comp> &it) {
        if (this != &it) {
            n = it.n ;
        }
    } ;
    bstree_iter(const bstree_iter<T,Comp> &&it) {
        if (this != &it) {
            n = it.n ;
        }
    } ;
    bstree_iter<T,Comp> &operator = (const bstree_iter<T,Comp> &it) {
        if (this != &it) {
            n = it.n ;
        }
        return (*this) ;
    } ;
    bstree_iter<T,Comp> &operator = (bstree_iter<T,Comp> &&it) {
        if (this != &it) {
            n = it.n ;
        }
        return (*this) ;
    } ;
    bstree_iter<T,Comp> &operator = (const bstree_iter<T,Comp> *ip) {
        if (this != ip) {
            n = ip->n ;
        }
        return (*this) ;
    } ;
    bstree_iter<T,Comp> &operator = (const bstree_node<T,Comp> *nn) {
        n = nn ;
        return (*this) ;
    } ;
    ~bstree_iter() {
        n = NULL ;
    } ;
    void setnode(bstree_node<T,Comp> *nn) {
        n = nn ;
    } ;
    T &operator * () const {
        T &rv = defval ;
        if (n != NULL) {
            rv = n->val ;
        }
        return rv ;
    } ;
    bstree_iter<T,Comp> &operator ++ () { /* pre-increment */
        return findnext(1) ;
    } ;
    bstree_iter<T,Comp> &operator ++ (int) { /* post-increment */
        return findnext(1) ;
    } ;
    bstree_iter<T,Comp> &operator += (int inc) {
        return findnext(inc) ;
    } ;

```



```

    }
    } else if (keyequal(n->val,v)) { /* equal */
        it.setnode(n) ;
    } else {
        if (n->right != NULL) {
            it = FindNodeByVal(n->right,v) ;
        }
    }
} /* end if non-null root) */
return it ;
} ;

void ReplaceUsInParent(nodetype *np,nodetype *c) {
    nodetype *p = np->parent ;
    if (p->left == np) {
        p->left = c ;
    } else {
        p->right = c ;
    }
    if (c != NULL) c->parent = p ;
} ;

nodetype *GetChild(nodetype *np) const {
    return (np->left != NULL) ? np->left : np->right ;
} ;

nodetype *FindMinNode(nodetype *np) const {
    while (np->left != NULL) {
        np = np->left ;
    }
    return np ;
}

int delnodes(bstree_node<T,Comp> *n) {
    int i = 0 ;
    if (n != NULL) {
        i += 1 ;
        c += delnodes(n->left) ;
        c += delnodes(n->right) ;
        delete n ;
    }
    return i ;
} ;

int insert(bstree_node<T,Comp> *n,bstree_node<T,Comp> *nn) {
    int d = 0 ;
    if (keycmp(nn->val,n->val)) {
        if (n->left != NULL) {
            d = insert(n->left,nn) ;
        } else {
            nn->parent = n ;
            n->left = nn ;
        }
    } else {
        if (n->right != NULL) {
            d = insert(n->right,nn) ;
        } else {
            nn->parent = n ;
            n->right = nn ;
        }
    }
    return d ;
} ;

int walk(std::vector<T> &vl,bstree_node<T,Comp> *n) const {
    int i = 0 ;
    if (n != NULL) {
        if (n->left) {
            i += walk(vl,n->left) ;
        }
        vl.push_back(n->val) ;
        i += 1 ;
    }
}

```

```

        if (n->right) {
            i += walk(v1,n->right) ;
        }
    }
    return i ;
} ;
bool keyequal(const T &v1,const T &v2) const { /* equal */
    bool f = TRUE ;
    f = f && (! keycmp(v1,v2)) ;
    f = f && (! keycmp(v2,v1)) ;
    return f ;
} ;

public:
typedef          bstree_iter<T,Comp> iterator ;
typedef          T value_type ;
bstree() {
} ;
bstree(const bstree<T,Comp> &a1) {
    if (this != &a1) {
        bstree_node<T,Comp>      *an = a1.root ;
        if (root != NULL) clear() ;
        while (an != NULL) {
            add(an->val) ;
            an = an->next ;
        }
    }
} ;
bstree(const bstree<T,Comp> &&a1) {
    if (this != &a1) {
        if (root != NULL) clear() ;
        root = a1.root ;
        c = a1.c ;
        a1.root = NULL ;
        a1.c = 0 ;
    }
} ;
bstree &operator = (const bstree<T,Comp> &a1) {
    if (this != &a1) {
        bstree_node<T,Comp>      *an = a1.root ;
        if (root != NULL) clear() ;
        while (an != NULL) {
            add(an->val) ;
            an = an->next ;
        }
    }
    return (*this) ;
} ;
bstree &operator = (const bstree<T,Comp> &&a1) {
    if (this != &a1) {
        if (root != NULL) clear() ;
        root = a1.root ;
        c = a1.c ;
        a1.root = NULL ;
        a1.c = 0 ;
    }
    return (*this) ;
} ;
bstree(const std::initializer_list<T> &list) {
    if (root != NULL) clear() ;
    for (const T &v : list) {
        add(v) ;
    }
} ;
bstree &operator = (const std::initializer_list<T> &list) {
    if (root != NULL) clear() ;
    for (const T &v : list) {

```

```

        add(v) ;
    }
    return (*this) ;
} ;
bstree &operator += (const std::initializer_list<T> &list) {
    for (const T &v : list) {
        add(v) ;
    }
    return (*this) ;
} ;
bstree &operator += (const T v) {
    add(v) ;
    return (*this) ;
} ;
~bstree() {
    if (root != NULL) {
        delnodes(root) ;
        root = NULL ;
    }
    c = 0 ;
} ;
int clear() {
    int rc = c ;
    if (root != NULL) {
        delnodes(root) ;
        root = NULL ;
    }
    c = 0 ;
    return rc ;
} ;
int add(const T v) {
    bstree_node<T,Comp> *nn = new bstree_node<T,Comp>(v) ;
    int rc = -1 ;
    if (nn != NULL) {
        if (root != NULL) {
            insert(root,nn) ;
        } else {
            root = nn ;
        }
        rc = ++c ;
    }
    return rc ;
} ;
int add(const std::initializer_list<T> il) {
    for (const T &v : il) {
        add(v) ;
    }
    return c ;
} ;
bstree &operator = (const std::initializer_list<T> il) {
    for (const T &v : il) {
        add(v) ;
    }
    return (*this) ;
} ;
int add(const bstree<T,Comp> &other) {
    for(const T &v : other) {
        add(v) ;
    }
    return c ;
} ;
int del(iterator it) {
    int rc = -1 ;
    if (it) {
        nodetype *n = it.n ; /* friend */
        nodetype *l, *r ;
    }
} ;

```

```

    l = n->left ;
    r = n->right ;
    if ((l != NULL) || (r != NULL)) { /* one or two children */
        if ((l != NULL) && (r != NULL)) { /* two children */
            nodetype *np = FindMinNode(n->right) ;
            n->SetVal(np->val) ;
            ReplaceUsInParent(np,np->right) ;
            delete np ;
        } else { /* one child */
            nodetype *child = GetChild(n) ;
            if (n->parent != NULL) {
                ReplaceUsInParent(n,child) ;
            } else {
                root = child ;
                if (child != NULL) child->parent = NULL ;
            }
            it.setnode(child) ;
            delete n ;
        }
    } else { /* leaf node */
        if (n->parent != NULL) {
            ReplaceUsInParent(n,NULL) ;
            it.setnode(n->parent) ;
        } else {
            root = NULL ;
            it.setnode(nullptr) ;
        }
        delete n ;
    }
    rc = --c ;
} /* end if (iterator not at end) */
return rc ;
} ;

int delval(const T &v) {
    int rc = -1 ;
    if (root != NULL) {
        iterator it ;
        if (it = FindNodeByVal(root,v)) {
            rc = del(it) ;
        }
    }
    return rc ;
} ;

int topval(const T **rpp) const {
    if (root != NULL) {
        bstree_node<T,Comp> *n = root ;
        *rpp = &n->val ;
    } else {
        *rpp = NULL ;
    }
    return c ;
} ;

int minval(const T **rpp) const {
    if (root != NULL) {
        bstree_node<T,Comp> *n = root ;
        while (n->left != NULL) {
            n = n->left ;
        }
        *rpp = &n->val ;
    } else {
        *rpp = NULL ;
    }
    return c ;
} ;

int maxval(const T **rpp) const {
    if (root != NULL) {

```



```

        bstree_node<T,Comp>      *n = root ;
        while (n->right != NULL) {
            n = n->right ;
        }
        *rpp = &n->val ;
    } else {
        *rpp = NULL ;
    }
    return c ;
} ;
int count() const {
    return c ;
} ;
int empty() const {
    return (c == 0) ;
} ;
operator int() const {
    return (c != 0) ;
} ;
operator bool() const {
    return (c != 0) ;
} ;
int storevec(std::vector<T> &vl) {
    int c = 0 ;
    if (root != NULL) {
        c = walk(vl,root) ;
    }
    return c ;
} ;
iterator begin() const {
    iterator it ;
    if (root != NULL) {
        bstree_node<T,Comp>      *n = root ;
        while (n->left != NULL) {
            n = n->left ;
        }
        it = iterator(n) ;
    }
    return it ;
} ;
iterator end() const {
    iterator it ;
    return it ;
} ;
iterator find(const T& v) const {
    iterator it ;
    if (root != NULL) {
        it = FindNodeByVal(root,v) ;
    }
    return it ;
} ;
int depth(bstree_depth *resp) {
    int d = 0 ;
    if (resp != NULL) resp->clear() ;
    d = depthrecurse(resp,0,root) ;
    return d ;
} ; /* end method (depth) */
int depthrecurse(bstree_depth *resp,int i,bstree_node<T,Comp> *rp) {
    int d = 0 ;
#if CF_DEBUGS
    debugprintf("bstree::depthrecurse: ent i=%u\n",i) ;
#endif
    if (rp != NULL) {
        int d_left = depthrecurse(resp,(i+1),rp->left) ;
        int d_right = depthrecurse(resp,(i+1),rp->right) ;
        d = 1 ;
    }
}

```

```

        d += std::max(d_left,d_right) ;
    } else {
        if (resp != NULL) {
            resp->min = std::min(resp->min,i) ;
            resp->max = std::max(resp->max,i) ;
        }
    }
}
#ifdef CF_DEBUGS
    debugprintf("bstree::depthrecurse: ret d=%u\n",d) ;
#endif
    return d ;
} ; /* end method (depthrecurse) */
} ; /* end class (bstree) */

#endif /* BSTREE_INCLUDE */

```