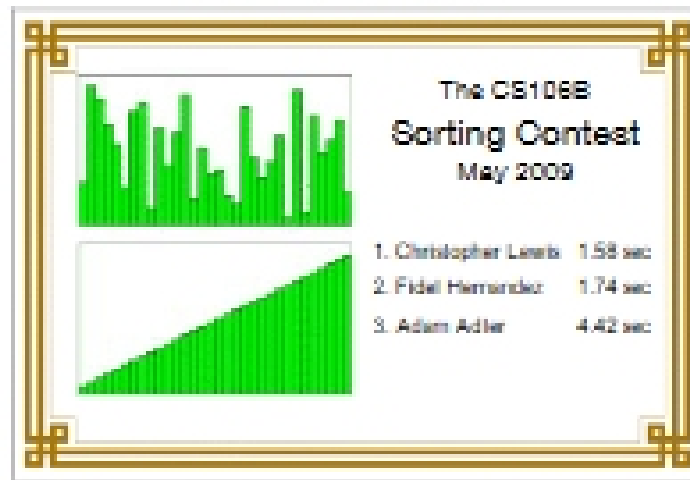


Binary Search Trees

Contest Results



Map Lookup Revisited

- The hash table strategy we introduced a week ago is by far the most common implementation of maps in practice.
- Despite its many advantages—most notably its extraordinary efficiency—the hash table implementation is not necessarily the best strategy for all applications.
- The hash table strategy has the following limitations:
 - Hash tables depend on being able to compute a hash function on some key. Expanding the hash-function idea so that it applies to types other than strings can be subtle.
 - The iterator for a map based on hash tables does not generate its values in any kind of order. If the key type has a natural order (which is called *lexicographic order* in the case of strings), the iterator cannot take advantage of that fact.

Looking Up Keys in an Array

- Instead of using a map, suppose that you entered all the state abbreviations (along with their names) in an array:



- How many operations (as a function of the size N) are needed to find a state abbreviation, such as "GA"?
- The lookup operation in a sorted array runs in $O(\log N)$ time.

Adding a Key to an Array

- But what about adding a new key? Suppose, for example, that Puerto Rico (PR) becomes a state:



- All the cells at the end of the array need to move to make room.
- The add operation in a sorted array runs in $O(N)$ time.

Fixing the Problem

- When we worked with the editor buffer two weeks ago, we solved the insertion problem by using a linked list instead of an array.
- Unfortunately, turning a sorted array into a linked list makes it impossible to apply binary search because there is no way to find the middle element.
- But what if you could point to the middle element in a linked list? That idea seems unlikely, but it is the key to finding a data structure that offers $O(\log N)$ performance for both the lookup and add operations.

Binary Search Trees

- The structure that ends up solving this problem is called a **binary search tree**. Each node in such a tree has exactly two subtrees: a left subtree that contains all the nodes that come before this one and a right subtree that contains all the nodes that come after it.
- The classical example is the following binary search tree of the seven dwarves:



The bst.h Interface

```

/*
 * file: bst.h
 * =====
 * this file provides an interface for a general binary search
 * tree class template.
 */

#ifndef _bst_h
#define _bst_h

#include "mydef.h"

/*
 * class: bst
 * =====
 * this interface defines a class template for a binary search tree.
 * for maximum generality, the tree is supplied as a class template.
 * the data type is set by the client. the client specializes the
 * tree to hold a specific type, e.g. numbers or characters.
 * the one requirement on the type is that the client must supply a
 * comparison function that compares two elements (as in calling
 * on the the default comparison function that takes as a and b).
 */

template <typename element>
class bst {
public:

```

The bst.h Interface

```

/*
 * constructor: bst
 * usage: bst<type> tree;
 * =====
 * bst<type> tree (comparator) {
 *
 * the constructor initializes a new empty binary search tree.
 * the one argument is a comparison function, which is called
 * to compare data values. this argument is optional; if not
 * given, comparator from mydef.h is used, which applies the
 * built-in operator < as its argument. of the behavior of a
 * as your type is defined and modified, you do not need to
 * supply your own comparison function.
 */

    bst<type> (*comp) (element a, element b) = comparator;

/*
 * constructor: tree
 * usage: (usually implicit)
 * =====
 * this function deallocates the storage for a tree.
 */

    ~tree();

```

The bst.h Interface

```

/*
 * method: find
 * usage: int (tree.find(key) != null)...
 * =====
 * this method applies the binary search algorithm to
 * find a particular key in this tree. the argument is the key
 * to use for comparison. if a node matching key appears in the
 * tree, find returns a pointer to the data in that node; otherwise,
 * find returns null.
 */

    element *find(element key);

/*
 * method: add
 * usage: tree.add(val);
 * =====
 * this method adds a new node to this tree. the elem
 * argument is compared with the data in existing nodes to find
 * the proper position. if a node with the same value already
 * exists, the element is overwritten with the new value and
 * data is returned. if no matching node is found, a new node
 * is allocated and added to the tree, null is returned.
 */

    void add(element elem);

```

The bst.h Interface

```

/*
 * method: remove
 * usage: tree.remove(key);
 * =====
 * this method removes a node in this tree that matches
 * the specified key. if a node matching key is found, the node
 * is removed from the tree and null is returned. if no match
 * is found, no changes are made and false is returned.
 */

    void remove(element key);

/*
 * method: clear
 * usage: tree.clear();
 * =====
 * this method removes all elements from this tree. the
 * tree is made empty and will have no nodes after being cleared.
 */

    void clear();

```

The bst.h Interface

```

/*
 * method: inorder
 * usage: tree.inorder(preorder, postorder);
 * =====
 * this method traverses through the binary search tree
 * and calls the function fn once for each element, passing the
 * element and the element's data. this data can be of obscure
 * type is needed for the client's utility. the order of calls
 * is determined by an inorder walk of the tree.
 */

template <typename element>
void inorder(void (*fn) (element elem, element data),
            element data);

private:
#include "mydef.h"
};

#include "testing1.cpp"

#endif

```

The bstpriv.h Data Structure

```

/*
 * file: bstpriv.h
 * =====
 * this file contains the private portion of the bst interface.
 * this file contains the private portion of the bst template.
 * class, including this information in a separate file name
 * when clients don't need to look at these details.
 */

/* type definition for a node */
struct node {
    element data;
    node *left, *right;
};

/* recursive variables */
node *root;
int (*comp) (element, element);

```

The cmpfn.h Interface

```

/*
 * cmpfn.h
 * =====
 * This interface exports a comparison function template.
 */

#ifndef _cmpfn_h
#define _cmpfn_h

/*
 * template comparison operators
 * usage: int sign = comparecmp(val1, val2);
 * =====
 * This function template is a generic function to compare two values
 * using the built-in == and != operators. It is supplied as a convenience
 * for those situations where a comparison function is required, and the
 * type has a built-in ordering that you would like to use.
 */

template <typename type>
int comparecmp(type val1, type val2) {
    if (val1 == val2) return 0;
    if (val1 < val2) return -1;
    return 1;
}

#endif

```

The bstimpl.cpp Implementation

```

/*
 * bstimpl.cpp
 * =====
 * This file implements the bst.h interface, which provides a
 * general implementation of binary search trees.
 */

#include <bst.h>

#include <stdlib.h>

template <typename dataType>
bst<dataType>::bst(int (*comp)(const dataType &, const dataType &)) {
    root = NULL;
    this->comp = comp;
}

template <typename dataType>
bst<dataType>::~bst() {
    clear();
}

```

The bstimpl.cpp Implementation

```

template <typename dataType>
bst<dataType>::~find(bst<dataType> *b) {
    node *found = searchNode(root, b);
    if (found != NULL) return *found->data;
    return NULL;
}

template <typename dataType>
template <typename nodeType>
nodeType bst<dataType>::searchNode(nodeType *root,
                                  bst<dataType> *b) {
    if (root == NULL) return NULL;
    int sign = comp(b->data, *root->data);
    if (sign == 0) return *root;
    if (sign < 0) {
        return searchNode(root->left, b);
    } else {
        return searchNode(root->right, b);
    }
}

```

The bstimpl.cpp Implementation

```

template <typename dataType>
bst<dataType>::add(bst<dataType> *b) {
    return searchNode(root, b);
}

template <typename dataType>
bst<dataType>::searchNode(nodeType *root,
                          bst<dataType> *b) {
    if (root == NULL) {
        root = new nodeType;
        root->data = b->data;
        root->left = root->right = NULL;
        return root;
    }
    int sign = comp(b->data, *root->data);
    if (sign < 0) {
        root->left = searchNode(root->left, b);
    } else if (sign > 0) {
        root->right = searchNode(root->right, b);
    } else {
        return searchNode(root->right, b);
    }
}

```

The bstimpl.cpp Implementation

```

/*
 * The remove node has been eliminated from these slides to save time.
 * You do it by just going through in the case. The clear method, however,
 * which is also called by the destructor, is particularly valuable
 * because it illustrates an important tree pattern. The clear
 * method operates recursively by deleting each of the subtrees and
 * then deleting the current node. In the language of trees,
 * moreover, this ordering is called a postorder traversal.
 */

template <typename dataType>
bst<dataType>::~clear() {
    clearNode(root);
    root = NULL;
}

template <typename dataType>
bst<dataType>::clearNode(nodeType *n) {
    if (n != NULL) {
        clearNode(n->left);
        clearNode(n->right);
        delete n;
    }
}

```

The bstimpl.cpp Implementation

```

template <typename dataType>
bst<dataType>::~clearNode(nodeType *n) {
    if (n != NULL) {
        clearNode(n->left);
        clearNode(n->right);
        delete n;
    }
}

```