CS 2213
Advanced Programming
Ch 13 – Trees 3
Generalized BST

Turgay Korkmaz

Office: SB 4.01.13
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

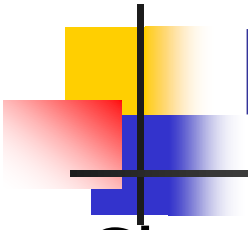
Thanks to Eric S. Roberts, the author of our textbook, for providing some slides/figures/programs.

A general interface for binary search trees

- Suppose we want to build a new application program using a binary search tree.
- How can we use the current implementation as is, or generalize it for various applications?

Self-Study

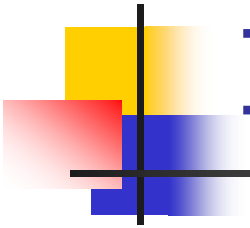
Might be in the next HW



First approach: Non-Interface-based design

- Change the definition of the **nodeT** type to include the data required by our application.
- This means we **edit the source** code, so we violate the basic principles of interface-based design.
- As a client, we should never have to edit the implementation code or know the details of the implementation at all.

Second approach: Interface-based design



- We can make binary search trees (BST) usable as a general tool by defining a **BST interface**/ADT that allows clients to invoke the basic operations without having to understand the underlying detail.
- In particular, the following features would certainly make the **BST** interface more useful: (see next slide)

The interface should allow the client to define the structure of the data in a node. The binary search trees you've seen so far have included no data fields except the key itself. In most cases, clients want to work with nodes that contain additional data fields as well.

The keys should not be limited to char/int/strings. Although the implementations in the preceding sections have used char/int/strings as keys, there is no reason that a general package would need to impose this constraint. To maintain the proper order in a binary tree, **all the interface implementation needs to know is how to compare two keys.** As long as the client provides **a comparison function**, it should be possible to use any type as a key. **--- Pointers to Functions ---**

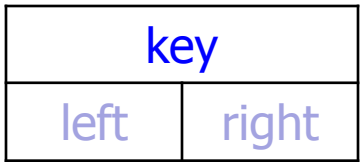
It should be possible to remove nodes as well as to insert them. Some clients need to be able to remove entries from a binary search tree. Removing a node requires some care, but is easy enough to specify in the implementation.

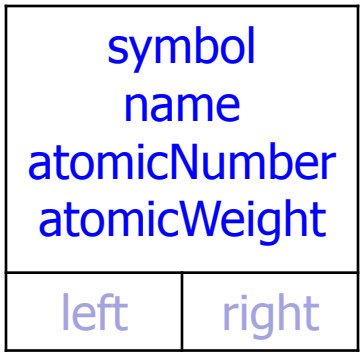
The details of any balancing algorithm should lie entirely on the implementation side of the abstraction boundary. The interface itself should not reveal what strategy, if any, the implementation uses to keep the tree in balance. Making the process of balancing the tree private to the implementation allows you to substitute new algorithms (e.g. red-black tree) that perform more effectively than the AVL strategy without forcing clients to change their code.

```
#ifndef _bst_h
#define _bst_h
#include "genlib.h"
#include "cmpfn.h“

 typedef struct bstCDT *bstADT;

// Pointer to functions: type defines the class of callback functions
typedef void (*nodeFnT)(void *np, void *clientData);
typedef void (*nodeInitFnT)(void *np, void *kp, void *clientData);

bstADT NewBST(int size, cmpFnT cmpFn, nodeInitFnT nodeInitFn);
void FreeBST(bstADT bst, nodeFnT freeNodeFn);
void *FindBSTNode(bstADT bst, void *kp);
void *InsertBSTNode(bstADT bst, void *kp, void *clientData);
void *DeleteBSTNode(bstADT bst, void *kp);

typedef enum { InOrder, PreOrder, PostOrder } traversalOrderT;
void MapBST(nodeFnT fn, bstADT bst, traversalOrderT order,  void *clientData);
void *BSTRoot(bstADT bst);
void *BSTLeftChild(bstADT bst, void *np);
void *BSTRightChild(bstADT bst, void *np);
#endif
```

np: the address of a node in the bst tree
kp: the address of the key in the client
clientData: the address of the data in the client

```
main()  /* bsttest.c application….*/
{
   scannerADT scanner;
   bstADT bst;
   string line, cmd, key, *np;
   bool newFlag;

   scanner = NewScanner();   SetScannerSpaceOption(scanner, IgnoreSpaces);
   bst = NewBST(sizeof(string), StringCmpFn, NodeInitFn);
   while (TRUE) {
      printf(">");       line = GetLine();
      SetScannerString(scanner, line);      cmd = ConvertToLowerCase(ReadToken(scanner));
      if (StringEqual(cmd, "find")) {
         key = ReadToken(scanner);  np = FindBSTNode(bst, &key);
         if (np == NULL) printf("No such node\n");
         else printf("Key %s found in node at %08lx\n", *np, (long) np);
      } else if (StringEqual(cmd, "insert")) {
         key = ReadToken(scanner);  np = InsertBSTNode(bst, &key, NULL);
      } else { /* other operations */ }
   }
}

int StringCmpFn(const void *p1, const void *p2) // this is in cmpfn.c
{
   return (StringCompare(*((string *) p1), *((string *) p2)));
}

static void NodeInitFn(void *np, void *kp, void *clientData)
{
   *((string *) np) = CopyString(*((string *) kp));
}
```





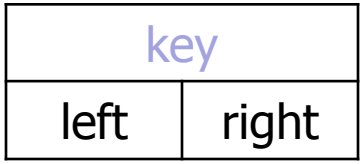
```
/*  File: bst.c  */

#include <stdio.h>
#include "genlib.h"
#include "cmpfn.h"
#include "bst.h"

typedef void *treeT;

struct bstCDT {
   treeT root;
   int userSize, totalSize;
   cmpFnT cmpFn;
   nodeInitFnT nodeInitFn;
};

typedef struct {
   treeT left, right;
} bstDataT;

/* Private function prototypes */  /* Exported entries */
```

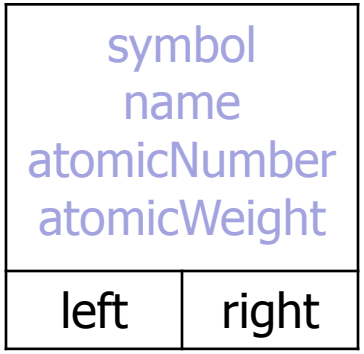
/* Because the implementation does not know the structure of a node, pointers to nodes cannot be defined explicitly and must be represented using void *. For readability, the code declares any void * pointers that are in fact trees to be of type treeT. */

/* This type is the concrete type used to represent the bstADT. */

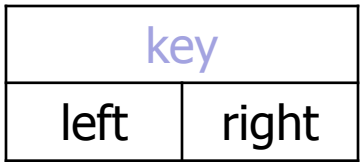
/ * This record is allocated at the end of the client's structure and is used to maintain the structure of the tree. The code calls BSTData on the node pointer to derive this address. */

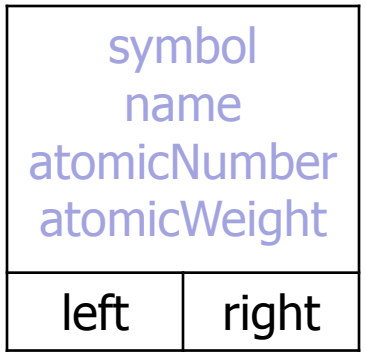
```
bstADT NewBST(int size, cmpFnT cmpFn,   nodeInitFnT nodeInitFn)
{
   bstADT bst;

   bst = New(bstADT);
   bst->root = NULL;
   bst->userSize = size;
   bst->totalSize = bst->userSize + sizeof(bstDataT);
   bst->cmpFn = cmpFn;
   bst->nodeInitFn = nodeInitFn;
   return (bst);
}

void *InsertBSTNode(bstADT bst, void *kp, void *clientData)
{
   return (RecInsertNode(bst, &bst->root, kp, clientData));
}
```





```
static void *RecInsertNode(bstADT bst, treeT *tptr,  void *kp, void *clientData)
{
    bstDataT *dp;
    treeT t;
    int sign;
    t = *tptr;
    if (t == NULL) {
        t = GetBlock(bst->totalSize);
        bst->nodeInitFn(t, kp, clientData);
        dp = BSTData(bst, t);
        dp->left = dp->right = NULL;
        *tptr = t;          return (t);
    }
    sign = bst->cmpFn(kp, t);
    if (sign == 0) return (t);
    dp = BSTData(bst, t);
    if (sign < 0) {
        return (RecInsertNode(bst, &dp->left, kp, clientData));
    } else {
        return (RecInsertNode(bst, &dp->right, kp, clientData));
    }
}
```






```
bstDataT *BSTData(bstADT bst, treeT t)
{
    return ((bstDataT *)
            ((char *) t + bst->userSize));
}
```

```
void *FindBSTNode(bstADT bst, void *kp)
{
    return (RecFindNode(bst, bst->root, kp));
}

static treeT *RecFindNode(bstADT bst, treeT t, void *kp)
{
    bstDataT *dp;
    int sign;

    if (t == NULL) return (NULL);
    sign = bst->cmpFn(kp, t);
    if (sign == 0) return (t);
    dp = BSTData(bst, t);
    if (sign < 0) {
        return (RecFindNode(bst, dp->left, kp));
    } else {
        return (RecFindNode(bst, dp->right, kp));
    }
}
```

```
#include "bst.h "                    /* another application.c using bst.h */
       typedef struct  {
          string symbol;
          string name;
          int atomicNumber;
          double atomicWeight;
       } ApplicationDataT;
main()
{
  bstADT bst;
  ApplicationDataT *np, mydata = {"He", "Helium" , 2, 4.0026};
  MyBST = NewBST(sizeof(ApplicationDataT), NodeCmpFn, NodeInitFn);
  np = (ApplicationDataT *) InsertBSTNode(MyBST, &mydata.symbol , &mydata);
}

static void NodeInitFn(void *np, void *kp, void *clientData)
{
    ((ApplicationDataT *) np)->symbol= ApplicationDataT * CopyString(*((string *) kp));
    // also copy client data into  node…
}
static void NodeCmpFn(void *p1, void *p2)
{
    return (StringCompare(*((string *) p1), *((string *) p2)));
}
```