Dynamic Data Structures with C

Terry Marris September 2010

8 Binary Trees

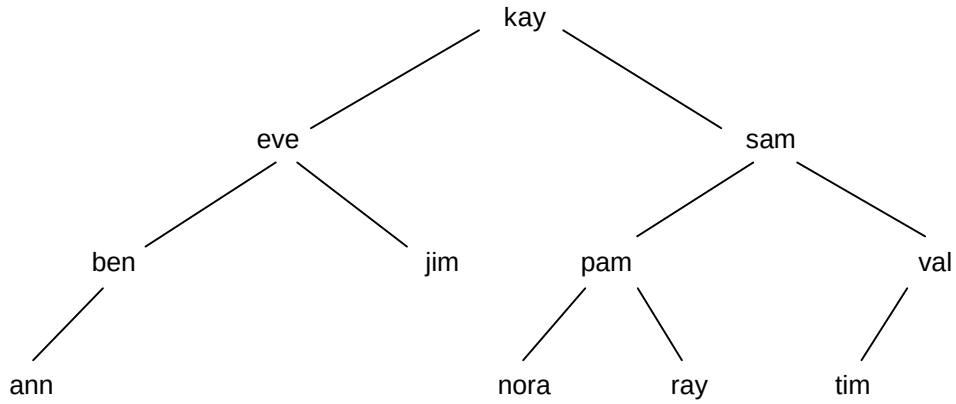
We have looked at pointers to functions and recursion. We see how they feature in binary search trees.

8.1 Binary Search Trees

By way of example we construct a binary search tree from a sequence of names:

kay, eve, sam, ben, val, jim, pam, ann, nora, tim, ray

The first name is *kay*. *kay* becomes the *root* element. *eve* comes before *kay* in alphabetical order, and so *eve* is placed to the left of *kay*. The next element is *sam*. *sam* comes after *kay* in alphabetical order and so is placed to the right of *kay*. Next is *ben*. *ben* comes before both *kay* and *eve*, and so is placed to the left of *eve*. Then we have *val*. *val* is placed to the right of both *kay* and *sam*. Then we come to *jim*. *jim* comes before *kay* and after *eve*. So *jim* is placed to the right of *eve*.



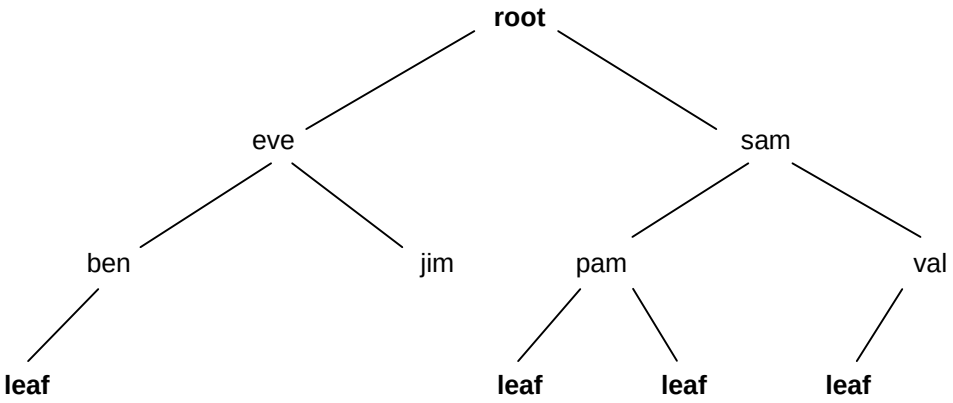
You can check that the remaining names, *pam*, *ann*, *nora*, *tim* and *ray*, are shown in their right places in the binary tree shown above.

We have just one *root node* at the head of the tree. *kay* is the element in this node.

The root node has no predecessors, and zero, one or two successor nodes. Here, the root node has two successor nodes occupied by *eve* and *sam* respectively.

A *leaf* node has just one predecessor and no successors. *ann*, *nora*, *ray* and *tim* represent leaf nodes. *ann*'s predecessor is *ben*.

Nodes that are neither root nor leaf nodes have one predecessor and up to two successors.



The terms *parent*, *child*, *sibling*, *ancestors* and *descendants* describe particular nodes.

The predecessor of a node is that node's parent. *eve* is the parent of *ben*. Every node (except the root) has a unique parent.

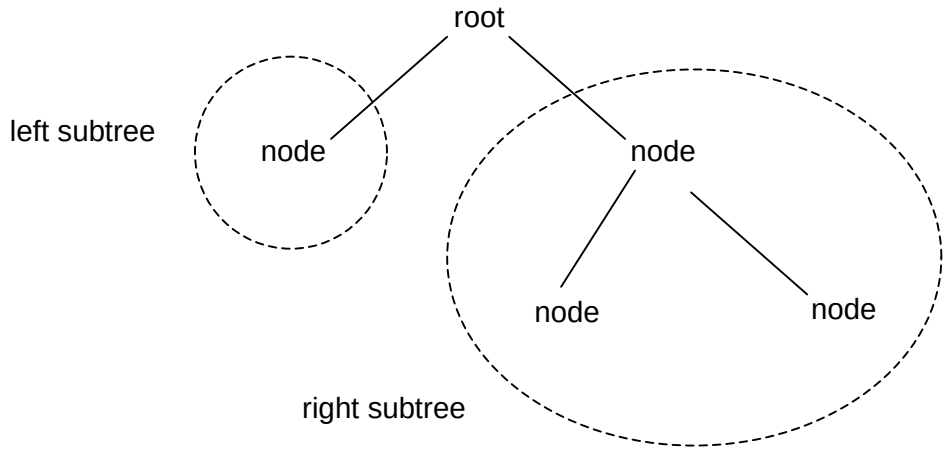
The successor of a node is that node's child. *jim* is the child of *eve*. Leaf nodes have no children.

Two nodes with the same parent are siblings. Here, *jim* and *ben* are siblings.

The root node has no ancestors. But every other node has its parent as an ancestor, and all its parent's ancestors are also ancestors of that node. Ancestors of *nora* include *pam*, *sam* and *kay*.

A leaf node has no descendants. But every other node has its children as descendents, and the descendants of all its children, are also decendants of that node. *ben*, *jim* and *ann* are descendants of *eve*.

A subtree is a node together with its descendants.



Each node is the root of a subtree.

A tree may be empty, or may have one or two subtrees. The subtrees are known as the left and right subtrees of their root. A subtree is itself a tree.

8.2 Binary Search Tree Interface

We present the binary search tree header file.

```
/* bstree.h - binary search tree */

#ifndef BST
#define BST

typedef int (*Comparable)(void *, void *);
typedef char *(*Duplicate)(void *);

typedef struct BSTreeStruct *BSTree;
typedef struct BSTNodeStruct *BSTreeNode;
typedef struct InOrderIterStruct *InOrderIterator;

/* newBSTree: returns a new binary tree */
BSTree newBSTree(Comparable, Duplicate);

/* deleteBSTree: erases the given tree */
int deleteBSTree(BSTree bst);

/* isEmptyBSTree: returns 1 if the given tree is empty */
int isEmptyBSTree(BSTree bst);

/* insertInBSTree: inserts the given object in the tree */
int insertInBSTree(BSTree bst, void *obj);

/* removeFromBSTree: removes the given object from the tree */
int removeFromBSTree(BSTree bst, void* element);

/* bSTreeContains: returns 1 if the given object is in the tree */
int bSTreeContains(BSTree bst, void *obj);

/* newInOrderIterator: returns a new iterator for
   the given binary tree */
InOrderIterator newInOrderIterator(BSTree tree);

/* inOrderIteratorHasNext: returns 1 if the tree has an element not
   yet visited in the current iteration */
int inOrderIteratorHasNext(InOrderIterator iter);

/* inOrderIteratorNext: returns the current element */
void *inOrderIteratorNext(InOrderIterator iter);

#endif
```

We have two user-defined functions:

```
typedef int (*Comparable)(void *, void *);
typedef char *(*Duplicate)(void *);
```

Comparable() compares two objects and returns zero if they are identical, -1 if the first comes before the second in lexical order, and +1 if the first comes after the second in alphabetical order.

$$\text{comparable}(a, b) = \begin{cases} -1 & \text{if } a < b \\ 0 & \text{if } a == b \\ +1 & \text{if } a > b \end{cases}$$

Duplicate() returns a duplicate copy of its argument.

Then we have three type declarations:

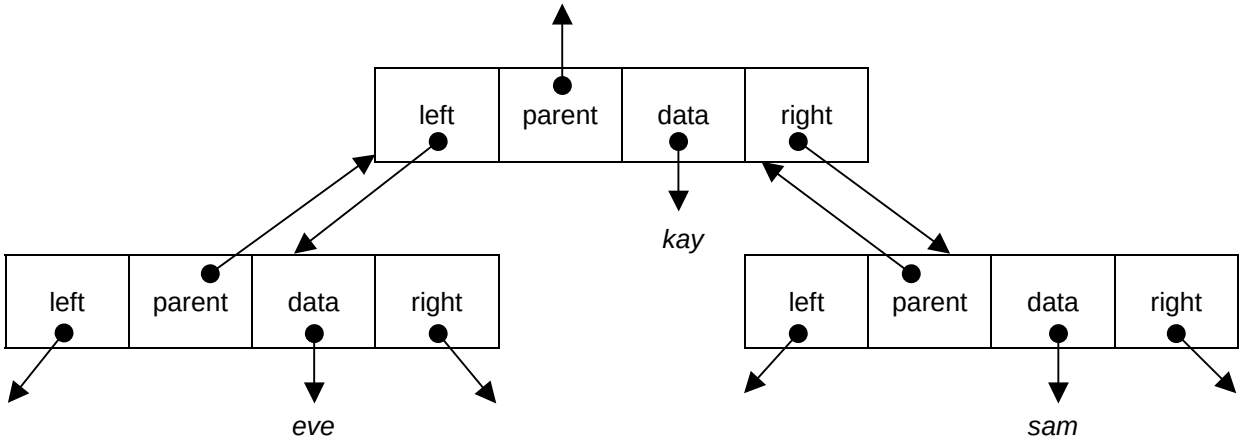
```
typedef struct BSTreeStruct *BSTree;
typedef struct BSTNodeStruct *BSTreeNode;
typedef struct InOrderIterStruct *InOrderIterator;
```

a pointer to a binary search tree structure, a pointer to a binary search tree node structure, and a pointer to an in-order iterator structure.

8.3 Binary Search Tree Node

```
struct BSTNodeStruct {
  struct BSTNodeStruct *left;
  struct BSTNodeStruct *right;
  struct BSTNodeStruct *parent;
  void *data;
};
```

A node has a pointer to its left child, a pointer to its right child, a pointer to its parent, and a pointer to its data.



Of course, the root node has no parent.

8.4 Binary Search Tree Structure

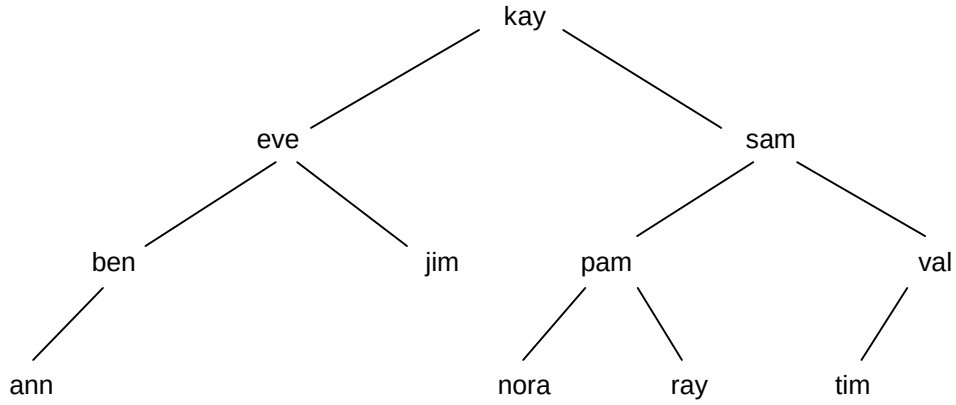
```
struct BSTreeStruct {
  BSTreeNode root;
  Comparable comparable;
  Duplicate duplicate;
};
```

A binary search tree has a pointer to its root node, as well as pointer to the functions *comparable()* and *duplicate()*.

8.5 In Order Iterator

```
struct InOrderIterStruct {
  Stack stack;
  BSTreeNode current;
  BSTree tree;
};
```

An in-order iterator visits the nodes containing *ann*, *ben*, *eve*, *jim*, *kay*, *nora*, *pam*, *ray*, *sam*, *tim* and *val* in that order.



The essence of an in-order traversal is:

if the tree is not empty
traverse the left subtree
visit the root
traverse the right subtree

For example

```
/* _inOrderTraverse: displays binary tree */
int _inOrderTraverse(BSTreeNode node)
{
  if (node != NULL) {
    _inOrderTraverse(node->left);
    printf("%s ", node->data);
    _inOrderTraverse(node->right);
  }
  return 0;
}
```

So:

traverse the left subtree of *kay*
    traverse the left subtree of *eve*
        traverse the left subtree of *ben*
            traverse the left subtree of *ann* - empty
            visit **ann**
            traverse the right subtree of *ann* - empty
        visit **ben**
        traverse the right subtree of *ben* - empty
    visit **eve**
    traverse the right subtree of *eve*
        traverse the left sutbtree of *jim* - empty
        visit **jim**
        traverse the right subtree of *jim* - empty
visit **kay**
traverse the right subtree of *kay*
    traverse the left subtree of *sam*
        traverse the left subtree of *pam*
            traverse the left subtree of *nora* - empty,
            visit **nora**
            traverse the right subtree of *nora* - empty
        visit **pam**
        traverse the right subtree of *pam*
            traverse the left subtree of *ray* - empty
            visit **ray**
            traverse the right subtree of *ray* - empty
    visit **sam**
    traverse the right subtree of *sam*
        traverse the left subtree of *val*
            traverse the left subtree of *tim* - empty
            visit **tim**
            traverse the right subtree of *tim* - empty
        visit **val**
        traverse the right subtree of *val* - empty

The nodes passed when traversing left are held on a stack so that they can be retrieved when you need to backtrack up the tree. You can look at the implementation, §8.11 Binary Search Tree Iterator, for more detail.

8.6 New Tree Node

```
/* newBSTreeNode: returns a new, empty tree node */
BSTreeNode newBSTreeNode()
{
  BSTreeNode n = (BSTreeNode)malloc(sizeof(struct BSTNodeStruct));
  assert(n != NULL);
  n->left = NULL;
  n->right = NULL;
  n->data = NULL;
  n->parent = NULL;
  return n;
}
```

We simply initialise each member to *NULL*. You could perhaps replace the call to *assert()* with a call to a purpose made error function.

8.7 New Binary Search Tree

```
/* newBSTree: returns a new binary tree */
BSTree newBSTree(Comparable comp, Duplicate dup)
{
  BSTree bst = (BSTree)malloc(sizeof(struct BSTreeStruct));
  assert(bst != NULL);
  bst->root = NULL;
  bst->comparable = comp;
  bst->duplicate = dup;
  return bst;
}
```

The creation of a new binary search tree is straightforward.

8.8 Empty Binary Search Tree

```
/* isEmptyBSTree: returns 1 if the given tree is empty */
int isEmptyBSTree(BSTree bst)
{
  assert(bst != NULL);
  if (bst->root == NULL)
    return 1;
  return 0;
}
```

A tree is empty if its root node is *NULL*.

8.9 Membership

We do not allow duplicate elements in the binary search tree. *bsTreeContains()* returns true (i.e. non-zero) if the tree contains the given object.

```
/* bSTreeContains: returns 1 if the given object is in the tree */
int bSTreeContains(BSTree bst, void *obj)
{
  assert(bst != NULL);
  return _bsTreeContains(bst->comparable, bst->root, obj);
}
```

A call to *_bsTreecontains()* is made, passing to it the function to compare two items, the root of the tree and the object to search for.

Incidentally, the public functions, those whose prototype is in the header file, have names without a leading underscore, whereas helper functions, functions that are private to the implementation, have names beginning with an underscore. This is just a convenient convention.

```
/* _bsTreeContains: returns 1 if the given object is in the tree */
int _bsTreeContains(
            Comparable comparable, BSTreeNode parent, void *obj)
{
  if (parent == NULL)
    return 0;  /* false */
  int cmp = comparable(obj, parent->data);
  if (cmp == 0)
    return 1;  /* true */
  else if (cmp < 0)
    return _bsTreeContains(comparable, parent->left, obj);
  else if (cmp > 0)
    return _bsTreeContains(comparable, parent->right, obj);
  return 0;
}
```

You can see that the function is recursive - it repeatedly calls itself until the stopping case, either *node == NULL* (object not found) , or the compare function returns zero (because the object searched for is found). If the element in the current node comes before the object you are looking for, go left. If the element in the current node comes after the object you are looking for, go right. If the element in the current node is identical to the one you are looking for, the given object is found, return and halt the sequence of recursive calls.
Because we do not allow repeated data items, we could use this data structure to implement a set.

8.10 Insert in Binary Search Tree

We are obliged to maintain the sorted order when inserting a new item in a binary search tree. We have two cases: either we are inserting a new object into a non-empty binary tree, or we are inserting a new object into an empty binary tree.

```
/* insertInBSTree: inserts the given object in the tree */
int insertInBSTree(BSTree bst, void *obj)
{
  assert(bst != NULL);
  if (!isEmptyBSTree(bst))
    return _insertInBSTree(
                  bst->comparable, bst->duplicate, bst->root, obj);
  else if (isEmptyBSTree(bst)) {
    bst->root = newBSTreeNode();
    bst->root->data = bst->duplicate(obj);
    return 0;
  }
  return 1;  /* unexpected error */
}
```

If the tree is not empty, we make a call to *_insertInBSTree()*, passing to it the user defined *compare()* and *duplicate()* functions, as well as the root of the binary tree and the object to be inserted. We rely on *_insertInBSTree()* to deal with duplicate objects. If the tree is empty the object to be inserted becomes the root element.

```
/* _insertInBSTree: inserts the given item in the tree */
int _insertInBSTree(Comparable comparable, Duplicate duplicate,
                                          BSTreeNode node, void *obj)
{
  int cmp = comparable(obj, node->data);
  if (cmp == 0)
    return 1;  /* silent error - duplicate entry */
  else if (cmp < 0) {
    if (node->left != NULL)
      return _insertInBSTree(comparable, duplicate, node->left, obj);
    else {
      node->left = newBSTreeNode();
      node->left->data = duplicate(obj);
      node->left->parent = node;
      return 0;
    }
  }
  else if (cmp > 0) {
    if (node->right != NULL)
      return _insertInBSTree(
                        comparable, duplicate, node->right, obj);
    else {
      node->right = newBSTreeNode();
      node->right->data = duplicate(obj);
      node->right->parent = node;
      return 0;
    }
  }
  return 1;
}
```

We compare the object to be inserted with the element in the current node. If they are the same we immediately return because duplicate elements are not allowed.

If the given object comes before the current node's element, we go left. If we cannot go left we create a new node to hang off the *left* pointer of the current node.

If the given object comes after the current node's element, we go right. If we cannot go right we create a new node to hang off the *right* pointer of the current node.

8.11 Binary Search Tree Iterator

The problem here is to visit each node in turn and in order.

Our helper function just goes left as far as possible, pushing nodes onto a stack as it does so.

```
/* _goLeft: traverse a left path */
BSTreeNode _goLeft(InOrderIterator iter, BSTreeNode node)
{
  if (node == NULL)
    return NULL;
  while (node->left != NULL) {
    push(node, iter->stack);
    node = node->left;
  }
  return node;
}
```

We create a new iterator for the given tree and initialise its stack with the sequence of nodes down the left side of the tree.

```
/* newInOrderIterator: returns a new iterator for
   the given binary tree */
InOrderIterator newInOrderIterator(BSTree tree)
{
  InOrderIterator iter = (InOrderIterator)malloc(sizeof(
                                    struct InOrderIterStruct));
  assert(iter != NULL);
  assert(tree != NULL);
  iter->stack = newStack();
  iter->tree = tree;
  iter->current = _goLeft(iter, iter->tree->root);
  return iter;
}
```

inOrderIteratorHasNext() returns true if there is a node in the tree to be visited.

```
/* inOrderIteratorHasNext: returns 1 if the tree has an element not
   yet visited in the current iteration */
int inOrderIteratorHasNext(InOrderIterator iter)
{
  return iter->current != NULL;
}
```

inOrderIteratorNext() returns the current element and advances the *iter->current* pointer onto the next node, if there is a next node. If not, *iter->current* is set to *NULL*, which sets *inOrderIteratorHasNext()* to true.

```
/* inOrderIteratorNext: returns the current element */
void *inOrderIteratorNext(InOrderIterator iter)
{
  assert(inOrderIteratorHasNext(iter));
  void *data = iter->current->data;
  if (iter->current->right != NULL)
    iter->current = _goLeft(iter, iter->current->right);
  else if (!isEmptyStack(iter->stack)) {
    iter->current = peek(iter->stack);
    pop(iter->stack);
  }
  else
    iter->current = NULL;
  return data;
}
```

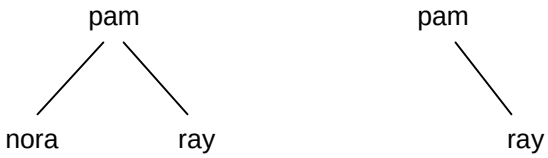
First, the data in the current node is stored. Then we have three cases that update the current node. If there is a node to the right, traverse onto to it then go left as far as possible. If the stack is not empty, then pop a node off it. If neither of these two cases are met, then we have reached the end of the tree and so we set *iter->current* to *NULL*.

8.12 Remove from Binary Search Tree

We can remove an object from a binary tree only if it is there in the first place.

```
/* removeFromBSTree: removes the given object from the tree */
int removeFromBSTree(BSTree bst, void* obj)
{
  assert(bst != NULL);
  if (!bSTreeContains(bst, obj))
    return 1;  /* silent error */
  return _removeFromBSTree(
                 bst->comparable, bst->duplicate, bst->root, obj);
}
```

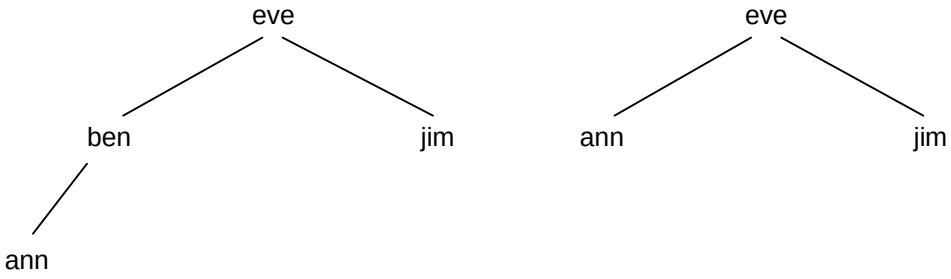
If the node to be deleted is a leaf node, we simply dispose of the node.



before and after deleting nora

```
/* case 1: no children */
if (node->left == NULL && node->right == NULL) {
  if (node->parent->left == node)
    node->parent->left = NULL;
  else
    node->parent->right = NULL;
  free(node);
  node = NULL;
}
```

If the node to be deleted has just one child we append the child to the node's parent.

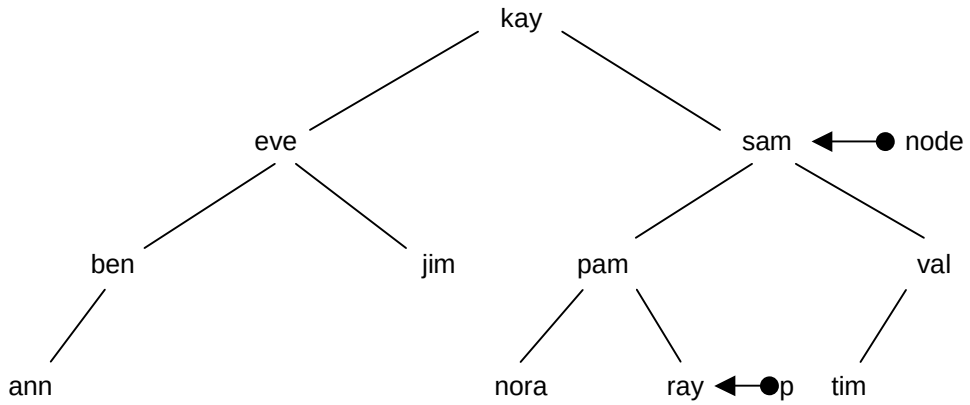


before and after deleting ben

```
/* case 2: no right child */
if (node->right == NULL) {
  BSTreeNode p = node->left;
  node->left = p->left;
  if (node->left != NULL)
    p->left->parent = node;
  node->right = p->right;
  if (node->right != NULL)
    node->right->parent = node;
  node->data = duplicate(p->data);
  free(p);
  p = NULL;
  return 0;
}
```

Case 3: no left child is similar.

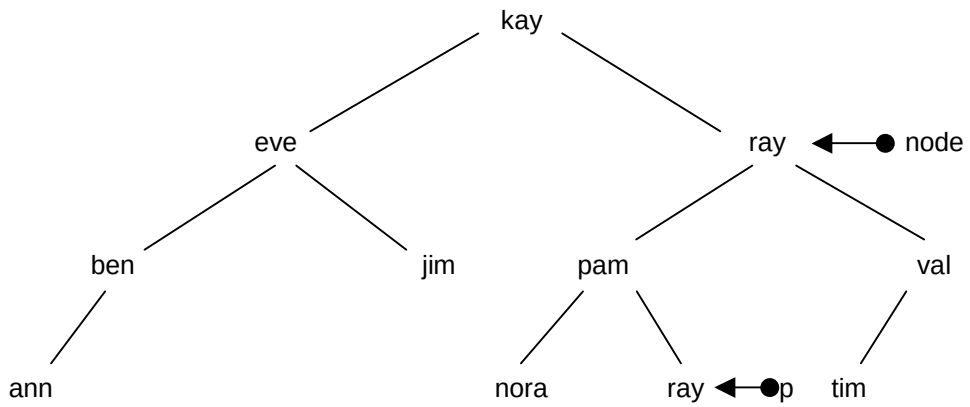
Now we examine case 4: deleting a node with two children, *sam*, for example.



We set up a pointer, *p*, which goes left of the node to be deleted, then right as far as it can.

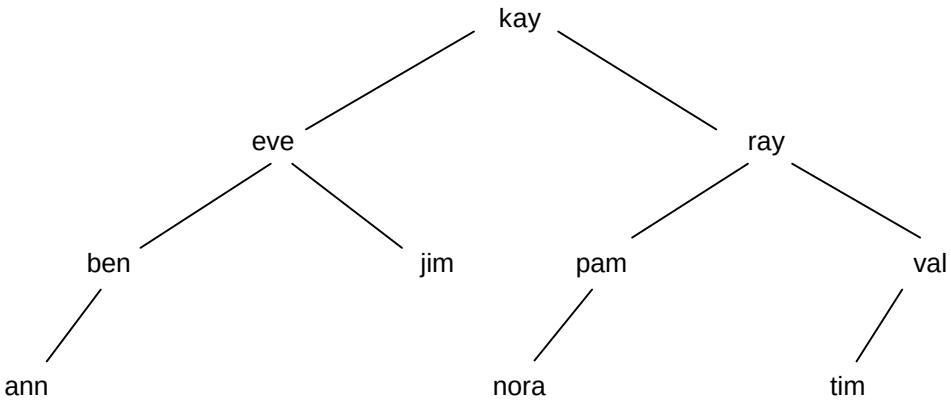
```
if (node->left != NULL && node->right != NULL) {
  BSTreeNode p = node->left;
  while (p->right != NULL)
    p = p->right;
  node->data = duplicate(p->data);
  _deleteRoot(duplicate, p);
}
```

We copy *p->data* into *node->data* and then call *_deleteRoot()* with *p* as an argument.



```
/* _deleteRoot: deletes the root node */
int _deleteRoot(Duplicate duplicate, BSTreeNode node)
{
  /* case 1: no children */
  if (node->left == NULL && node->right == NULL) {
    if (node->parent->left == node)
      node->parent->left = NULL;
    else
      node->parent->right = NULL;
    free(node);
    node = NULL;
    return 0;
  }
      ....
```

p has no children and so its removal is straightforward.



after removing sam

The two helper functions are shown in their entirety below.

```
/* _deleteRoot: deletes the root node */
int _deleteRoot(Duplicate duplicate, BSTreeNode node)
{
  /* case 1: no children */
  if (node->left == NULL && node->right == NULL) {
    if (node->parent->left == node)
      node->parent->left = NULL;
    else
      node->parent->right = NULL;
    free(node);
    node = NULL;
    return 0;
  }
  /* case 2: no right child */
  else if (node->right == NULL) {
    BSTreeNode p = node->left;
    node->left = p->left;
    if (node->left != NULL)
      p->left->parent = node;
    node->right = p->right;
    if (node->right != NULL)
      node->right->parent = node;
    node->data = duplicate(p->data);
    free(p);
    p = NULL;
```

```
    return 0;
  }
  /* case 3: no left child */
  else if (node->left == NULL) {
    BSTreeNode p = node->right;
    node->right = p->right;
    if (node->right != NULL)
      node->right->parent = node;
    node->left = p->left;
    if (node->left != NULL)
      node->left->parent = node;
    node->data = duplicate(p->data);
    free(p);
    p = NULL;
    return 0;
  }
  return 1;
}

/* _removeFromBSTree: removes the given object from the tree */
int _removeFromBSTree(Comparable comparable, Duplicate duplicate,
BSTreeNode node, void *obj)
{
  if (node == NULL)
    return 0;
  int cmp = comparable(obj, node->data);
  if (cmp < 0)
    return _removeFromBSTree(comparable, duplicate, node->left, obj);
  else if (cmp > 0)
    return _removeFromBSTree(
                         comparable, duplicate, node->right, obj);
  else if (cmp == 0) {
    if (node->left == NULL || node->right == NULL)
      return _deleteRoot(duplicate, node);
    else if (node->left != NULL && node->right != NULL) {
      BSTreeNode p = node->left;
      while (p->right != NULL)
        p = p->right;
      node->data = duplicate(p->data);
      return _deleteRoot(duplicate, p);
    }
  }
  return 1;  /* unexpected error */
}
```

8.13 Delete a Binary Search Tree

If the tree is not empty we call *_deleteSubTrees()* with the root as its argument.

```
/* deleteBSTree: erases the given tree */
int deleteBSTree(BSTree bst)
{
  assert(bst != NULL);
  if (!isEmptyBSTree(bst)) {
    if (bst->root->parent == NULL) {
      _deleteSubTrees(bst->root);
      bst->root = NULL;
    }
  }
  return 0;
}
```

Basically, we go left as far as we can, then right as far as we can, then back up the sequence of recursive calls deleting nodes as we go.

```
/* _deleteSubTrees: deletes all nodes - except the node with no
   parent */
int _deleteSubTrees(BSTreeNode node)
{
  if (node->left != NULL)
    return _deleteSubTrees(node->left);
  else if (node->right != NULL)
    return _deleteSubTrees(node->right);
  else if (node->parent != NULL) {
    free(node);
    node = NULL;
    return 0;
  }
  return 1;
}
```

8.14 Binary SearchTree Implementation

The entire implementation is shown below.

```
/* bstree.c - binary search tree */

#include <stdio.h>
#include <stdlib.h>
#include <stddef.h>
#include <assert.h>
#include "stack.h"
#include "bstree.h"

struct BSTNodeStruct {
  struct BSTNodeStruct *left;
  struct BSTNodeStruct *right;
  struct BSTNodeStruct *parent;
  void *data;
};

struct BSTreeStruct {
  BSTreeNode root;
```

```
  Comparable comparable;
  Duplicate duplicate;
};

struct InOrderIterStruct {
  Stack stack;
  BSTreeNode current;
  BSTree tree;
};

/* newBSTreeNode: returns a new, empty tree node */
BSTreeNode newBSTreeNode()
{
  BSTreeNode n = (BSTreeNode)malloc(sizeof(struct BSTNodeStruct));
  assert(n != NULL);
  n->left = NULL;
  n->right = NULL;
  n->data = NULL;
  n->parent = NULL;
  return n;
}

/* newBSTree: returns a new binary tree */
BSTree newBSTree(Comparable comp, Duplicate dup)
{
  BSTree bst = (BSTree)malloc(sizeof(struct BSTreeStruct));
  assert(bst != NULL);
  bst->root = NULL;
  bst->comparable = comp;
  bst->duplicate = dup;
  return bst;
}

/* _deleteSubTrees: deletes all nodes - except the node with no
  parent */
int _deleteSubTrees(BSTreeNode node)
{
  if (node->left != NULL)
    return _deleteSubTrees(node->left);
  else if (node->right != NULL)
    return _deleteSubTrees(node->right);
  else if (node->parent != NULL) {
    free(node);
    node = NULL;
    return 0;
  }
  return 1;
}

/* deleteBSTree: erases the given tree */
int deleteBSTree(BSTree bst)
{
  assert(bst != NULL);
  if (!isEmptyBSTree(bst)) {
    if (bst->root->parent == NULL) {
      _deleteSubTrees(bst->root);
      bst->root = NULL;
    }
  }
  return 0;
}
```

```
/* isEmptyBSTree: returns 1 if the given tree is empty */
int isEmptyBSTree(BSTree bst)
{
  assert(bst != NULL);
  if (bst->root == NULL)
    return 1;
  return 0;
}

/* _insertInBSTree: inserts the given item in the tree */
int _insertInBSTree(Comparable comparable, Duplicate duplicate,
BSTreeNode node, void *obj)
{
  int cmp = comparable(obj, node->data);
  if (cmp == 0)
    return 1;  /* silent error - duplicate entry */
  else if (cmp < 0) {
    if (node->left != NULL)
      return _insertInBSTree(comparable, duplicate, node->left, obj);
    else {
      node->left = newBSTreeNode();
      node->left->data = duplicate(obj);
      node->left->parent = node;
      return 0;
    }
  }
  else if (cmp > 0) {
    if (node->right != NULL)
      return _insertInBSTree(
                        comparable, duplicate, node->right, obj);
    else {
      node->right = newBSTreeNode();
      node->right->data = duplicate(obj);
      node->right->parent = node;
      return 0;
    }
  }
  return 1;
}

/* insertInBSTree: inserts the given object in the tree */
int insertInBSTree(BSTree bst, void *obj)
{
  assert(bst != NULL);
  if (!isEmptyBSTree(bst))
    return _insertInBSTree(
                  bst->comparable, bst->duplicate, bst->root, obj);
  else if (isEmptyBSTree(bst)) {
    bst->root = newBSTreeNode();
    bst->root->data = bst->duplicate(obj);
    return 0;
   }
   return 1;  /* unexpected error */
}

/* _bsTreeContains: returns 1 if the given object is in the tree */
int _bsTreeContains(
            Comparable comparable, BSTreeNode parent, void *obj)
{
  if (parent == NULL)
```

```
    return 0;  /* false */
  int cmp = comparable(obj, parent->data);
  if (cmp == 0)
    return 1;  /* true */
  else if (cmp < 0)
    return _bsTreeContains(comparable, parent->left, obj);
  else if (cmp > 0)
    return _bsTreeContains(comparable, parent->right, obj);
  return 0;
}

/* bSTreeContains: returns 1 if the given object is in the tree */
int bSTreeContains(BSTree bst, void *obj)
{
  assert(bst != NULL);
  return _bsTreeContains(bst->comparable, bst->root, obj);
}

/* _deleteRoot: deletes the root node */
int _deleteRoot(Duplicate duplicate, BSTreeNode node)
{
  /* case 1: no children */
  if (node->left == NULL && node->right == NULL) {
    if (node->parent->left == node)
      node->parent->left = NULL;
    else
      node->parent->right = NULL;
    free(node);
    node = NULL;
    return 0;
  }
  /* case 2: no right child */
  else if (node->right == NULL) {
    BSTreeNode p = node->left;
    node->left = p->left;
    if (node->left != NULL)
      p->left->parent = node;
    node->right = p->right;
    if (node->right != NULL)
      node->right->parent = node;
    node->data = duplicate(p->data);
    free(p);
    p = NULL;
    return 0;
  }
  /* case 3: no left child */
  else if (node->left == NULL) {
    BSTreeNode p = node->right;
    node->right = p->right;
    if (node->right != NULL)
      node->right->parent = node;
    node->left = p->left;
    if (node->left != NULL)
      node->left->parent = node;
    node->data = duplicate(p->data);
    free(p);
    p = NULL;
    return 0;
  }
  return 1;
}
```

```
/* _removeFromBSTree: removes the given object from the tree */
int _removeFromBSTree(Comparable comparable, Duplicate duplicate,
                                          BSTreeNode node, void *obj)
{
  if (node == NULL)
    return 0;
  int cmp = comparable(obj, node->data);
  if (cmp < 0)
    return _removeFromBSTree(comparable, duplicate, node->left, obj);
  else if (cmp > 0)
    return _removeFromBSTree(
                        comparable, duplicate, node->right, obj);
  else if (cmp == 0) {
    if (node->left == NULL || node->right == NULL)
      return _deleteRoot(duplicate, node);
    else if (node->left != NULL && node->right != NULL) {
      BSTreeNode p = node->left;
      while (p->right != NULL)
        p = p->right;
      node->data = duplicate(p->data);
      return _deleteRoot(duplicate, p);
    }
  }
  return 1;  /* unexpected error */
}

/* removeFromBSTree: removes the given object from the tree */
int removeFromBSTree(BSTree bst, void* obj)
{
  assert(bst != NULL);
  if (!bSTreeContains(bst, obj))
    return 1;  /* silent error */
  return _removeFromBSTree
                 (bst->comparable, bst->duplicate, bst->root, obj);
}

/* _inOrderTraverse: displays binary tree */
int _inOrderTraverse(BSTreeNode node)
{
  if (node != NULL) {
    _inOrderTraverse(node->left);
    printf("%s ", node->data);
    _inOrderTraverse(node->right);
  }
  return 0;
}

/* inOrderTraverse: displays the binary tree */
int inOrderTraverse(BSTree bst)
{
  assert(bst != NULL);
  return _inOrderTraverse(bst->root);
}

/* _goLeft: traverse a left path */
BSTreeNode _goLeft(InOrderIterator iter, BSTreeNode node)
{
  if (node == NULL)
    return NULL;
  while (node->left != NULL) {
```

```
    push(node, iter->stack);
    node = node->left;
  }
  return node;
}

/* newInOrderIterator: returns a new iterator for
   the given binary tree */
InOrderIterator newInOrderIterator(BSTree tree)
{
  InOrderIterator iter = (InOrderIterator)malloc(sizeof(
                         struct InOrderIterStruct));
  assert(iter != NULL);
  assert(tree != NULL);
  iter->stack = newStack();
  iter->tree = tree;
  iter->current = _goLeft(iter, iter->tree->root);
  return iter;
}

/* inOrderIteratorHasNext: returns 1 if the tree has an element not
   yet visited in the current iteration */
int inOrderIteratorHasNext(InOrderIterator iter)
{
  return iter->current != NULL;
}

/* inOrderIteratorNext: returns the current element */
void *inOrderIteratorNext(InOrderIterator iter)
{
  assert(inOrderIteratorHasNext(iter));
  void *data = iter->current->data;
  if (iter->current->right != NULL)
    iter->current = _goLeft(iter, iter->current->right);
  else if (!isEmptyStack(iter->stack)) {
    iter->current = peek(iter->stack);
    pop(iter->stack);
  }
  else
    iter->current = NULL;
  return data;
}
```

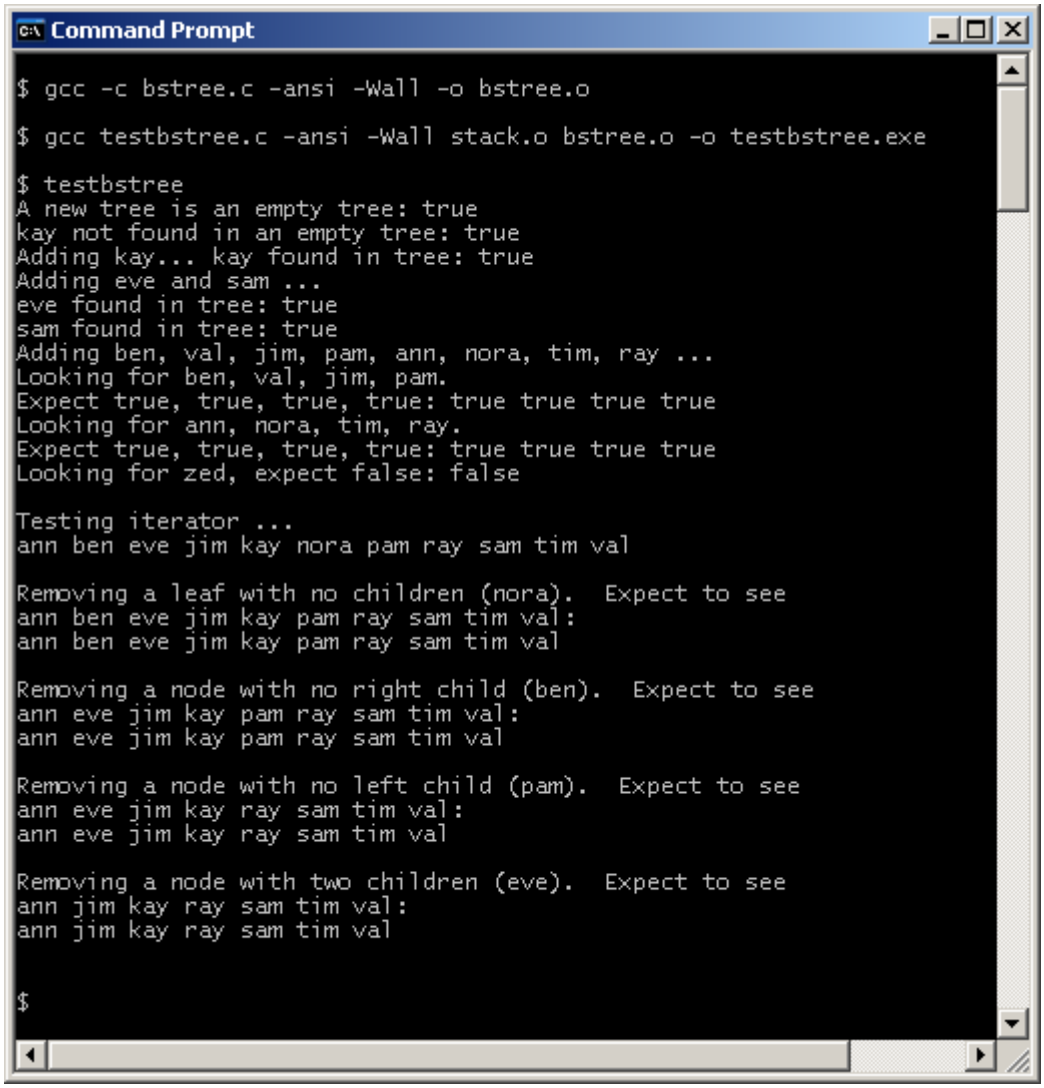
8.15 Binary Search Tree Test

A test program and its run are shown below.

```
/* testbstree.c */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "bstree.h"
```

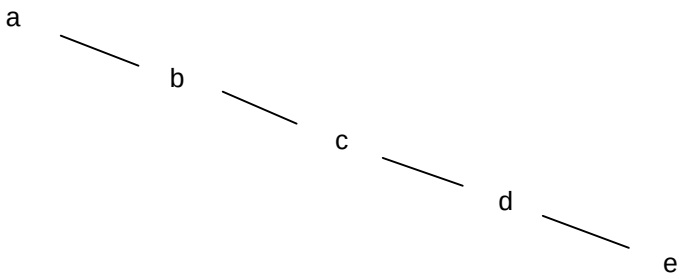
```
/* compareStrings: returns -1 if s1 < s2, 0 if s1 == s2,
   1 if s1 > s2 in dictionary order.  Case is significant */
int compareStrings(const char *s1, const char* s2)
{
  return strcmp(s1, s2);
}

/* copyString: returns a copy of the given string */
char *copyString(char *s)
{
  char *t = (char *)malloc(sizeof(s) + 1);
  strcpy(t, s);
  return t;
}

int main()
{
  BSTree bst = newBSTree((Comparable)compareStrings,
                         (Duplicate)copyString);
  printf("A new tree is an empty tree: ");
  isEmptyBSTree(bst)? printf("true") : printf("false");
  printf("\n");
  printf("kay not found in an empty tree: ");
  bSTreeContains(bst, "kay")? printf("false") : printf("true");
  printf("\n");

  printf("Adding kay... ");
  insertInBSTree(bst, "kay");
  printf("kay found in tree: ");
  bSTreeContains(bst, "kay")? printf("true") : printf("false");
  printf("\n");

  printf("Adding eve and sam ...\n");
  insertInBSTree(bst, "eve");
  insertInBSTree(bst, "sam");
  printf("eve found in tree: ");
  bSTreeContains(bst, "eve")? printf("true") : printf("false");
  printf("\n");
  printf("sam found in tree: ");
  bSTreeContains(bst, "sam")? printf("true") : printf("false");
  printf("\n");
  printf("Adding ben, val, jim, pam, ann, nora, tim, ray ... ");
  insertInBSTree(bst, "ben");
  insertInBSTree(bst, "val");
  insertInBSTree(bst, "jim");
  insertInBSTree(bst, "pam");
  insertInBSTree(bst, "ann");
  insertInBSTree(bst, "nora");
  insertInBSTree(bst, "tim");
  insertInBSTree(bst, "ray");
  printf("\n");
  printf("Looking for ben, val, jim, pam. \n");
  printf("Expect true, true, true, true: ");
  bSTreeContains(bst, "ben")? printf("true ") : printf("false");
  bSTreeContains(bst, "val")? printf("true ") : printf("false");
  bSTreeContains(bst, "jim")? printf("true ") : printf("false");
  bSTreeContains(bst, "pam")? printf("true ") : printf("false");
  printf("\n");
  printf("Looking for ann, nora, tim, ray. \n");
  printf("Expect true, true, true, true: ");
  bSTreeContains(bst, "ann")? printf("true ") : printf("false");
```

```
 bSTreeContains(bst, "nora")? printf("true ") : printf("false");
 bSTreeContains(bst, "tim")? printf("true ") : printf("false");
 bSTreeContains(bst, "ray")? printf("true ") : printf("false");
 printf("\n");

 printf("Looking for zed, expect false: ");
 bSTreeContains(bst, "zed")? printf("true") : printf("false");
 printf("\n\n");

 inOrderTraverse(bst);
 printf("\n\n");

 printf("Testing iterator ...\n");
 InOrderIterator iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));
 printf("\n\n");

 printf("Removing a leaf with no children (nora). ");
 printf("Expect to see \n");
 printf("ann ben eve jim kay pam ray sam tim val:\n");
 removeFromBSTree(bst, "nora");
 iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));
 printf("\n\n");

 printf("Removing a node with no right child (ben). ");
 printf(" Expect to see \n");
 printf("ann eve jim kay pam ray sam tim val:\n");
 removeFromBSTree(bst, "ben");
 iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));
 printf("\n\n");

 printf("Removing a node with no left child (pam).  ");
 printf("Expect to see \n");
 printf("ann eve jim kay ray sam tim val:\n");
 removeFromBSTree(bst, "pam");
 iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));
 printf("\n\n");

 printf("Removing a node with two children (eve).  ");
 printf("Expect to see \n");
 printf("ann jim kay ray sam tim val:\n");
 removeFromBSTree(bst, "eve");
 iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));
 printf("\n\n");

deleteBSTree(bst);
iter = newInOrderIterator(bst);
 while (inOrderIteratorHasNext(iter))
   printf("%s ", inOrderIteratorNext(iter));

 return 0;
}
```



Exercise 8.1

1. In the worst case, a binary search tree may be a linear if, for example, elements were inserted in alphabetical order: *a*, *b*, *c*, *d*, *e*:



Investigate how you could make such a tree balanced. A balanced binary tree has all its leaves on the two bottom levels.

Bibliography

Kernighan B, Ritchie D, *The C Programming Language*, Prentice Hall, 1988
Mark Williams Company, *ANSI C - A lexical Guide*, Prentice Hall 1988
Shiflet A *Data Structures in C++ Including Breadth and Labs* West Publishing Co 1996