Tree node

- We'll again re-use the same problem as shown earlier (that of storing <name, value> pairs)
- We now need links to the left and right subtree
- Note that the "lesser" and "greater" comments are used to help the programmer
  - Must still ensure the semantics of our operations follow the meaning of the comments!
- Code to create such a node is left as an exercise.

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int      value;
    Nameval *left;  /* lesser */
    Nameval *right; /* greater */
};
```

Construction

- Constructing such a tree means descending into the tree recursively.
  - At insertion time, each node ends up as a leaf node.
  - As other items are inserted later, the node above will become a parent.
- The algorithm must choose the left or right branch until the right place to link is found
- As with the linked-list routines, the insertion algorithm returns the root of the tree as the result.

Inserting node into tree

```
/* Assume newp has been already initialized. */
Nameval *insert(Nameval *treep, Nameval *newp)
{
    int cmp;

    if (treep == NULL) {
        return newp;
    }
    cmp = strcmp(newp->name, treep->name);
    if (cmp == 0) {
        fprintf(stderr, "insert: ignoring duplicate entry %s\n",
            newp->name);
    } else if (cmp < 0) {
        treep->left = insert(treep->left, newp);
    } else {
        treep->right = insert(treep->right, newp);
    }
    return treep;
}
```

Some observations

- The tree routines just shown do not permit duplicate entries.
- The insertion routine does not try to keep the tree balanced
  - It is possible that a sequence of inserts could yield a linear list instead of a tree (i.e., inserting a sequence of items that are already sorted).
  - However, this means our routines are a lot simpler (although it is not an oppressive amount of work to implement an AVL tree; rather, it is just a bit complicated!)
- The code for a lookup is similar to that for insertion
  - Recursively search by choosing the left or right subtrees
  - Return the correct node if matching lookup criteria, NULL otherwise.

lookup

```
Nameval *lookup(Nameval *treep, char *name)
{
    int cmp;

    if (treep == NULL) {
        return NULL;
    }
    cmp = strcmp(name, treep->name);
    if (cmp == 0) {
        return treep;
    } else if (cmp < 0) {
        return lookup(treep->left, name);
    } else {
        return lookup(treep->right, name);
    }
}
```

Must be recursive?

- Both insert and lookup were recursive
  - The routines were defined in terms of themselves.
  - Base case: empty tree
  - Inductive step: left tree, then right tree
- However, not all recursive routines need to be so
  - **Tail recursion**: when the recursive step (i.e., invocation of the recursive function) is the last step of the function
  - We can transform tail-recursive functions into iterative ones
  - All we require is some patching up of arguments (via assignments) and need a way to restart the body of the routine (via some loop)

Non-recursive lookup

```
Nameval *lookup(Nameval *treep,
    char *name)
{
    int cmp;

    if (treep == NULL) {
        return NULL;
    }
    cmp = strcmp(name, treep->name);
    if (cmp == 0) {
        return treep;
    } else if (cmp < 0) {
        return lookup(treep->left,
            name);
    } else {
        return lookup(treep->right,
            name);
    }
}
```

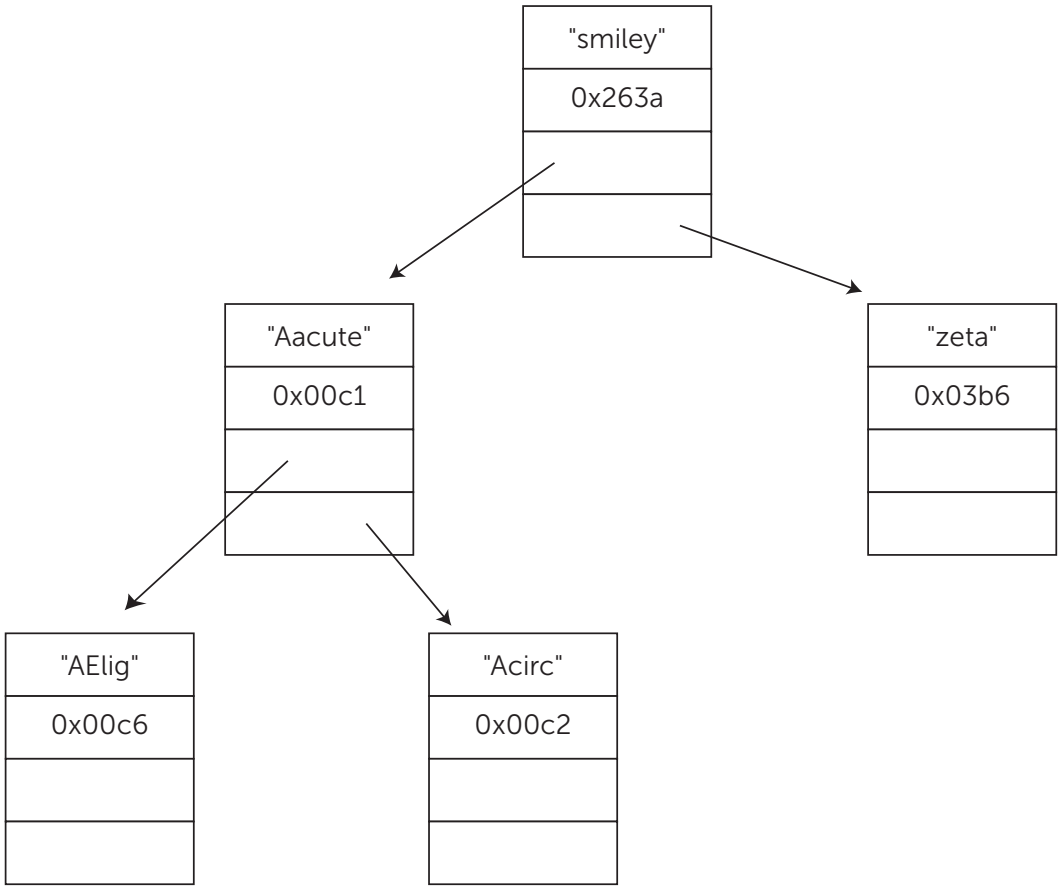
```
Nameval *nrlookup(Nameval *treep,
    char *name)
{
    int cmp;

    while (treep != NULL) {
        cmp = strcmp(name, treep->name);
        if (cmp == 0) {
            return treep;
        } else if (cmp < 0) {
            treep = treep -> left;
        } else {
            treep = treep->right;
        }
    }
    return NULL;
}
```

An observation about trees

- The same observations we made about operations on lists can also be made with respect to operations on trees
  - Traverse through the tree in some order
  - While doing so, compute some value / perform some comparison / etc.
  - After traversing the tree, return some value
- If we want to rewrite **apply** for a binary search tree, we must decide on some order
  - **inorder** traversal?
  - **pre-order** traversal?
  - **post-order** traversal?
- In effect we will have one **apply** function for each ordering, and each of these functions will take arguments similar to what we had for the list version of **apply**.

Binary search tree example




```
inorder:
    AElig
    Aacute
    Acirc
    smiley
    zeta

post-order:
    AElig
    Acirc
    Aacute
    aeta
    smiley

pre-order:
    ???
```

applyinorder

```
void applyinorder(Nameval *treep,
    void (*fn)(Nameval*, void*), void *arg)
{
    if (treep == NULL) {
        return;
    }
    applyinorder(treep->left, fn, arg);
    (*fn)(treep, arg);
    applyinorder(treep->right, fn arg);
}
```

```
/* We can even use some of the functions we passed as arguments
 * to the "list" version of apply!
 */

applyinorder(treep, printnv, "%s: %x\n");
```

```
/* Could you build a sort based on the tree routines +
 * a function (that you would write) given to applyinorder?
 */
```

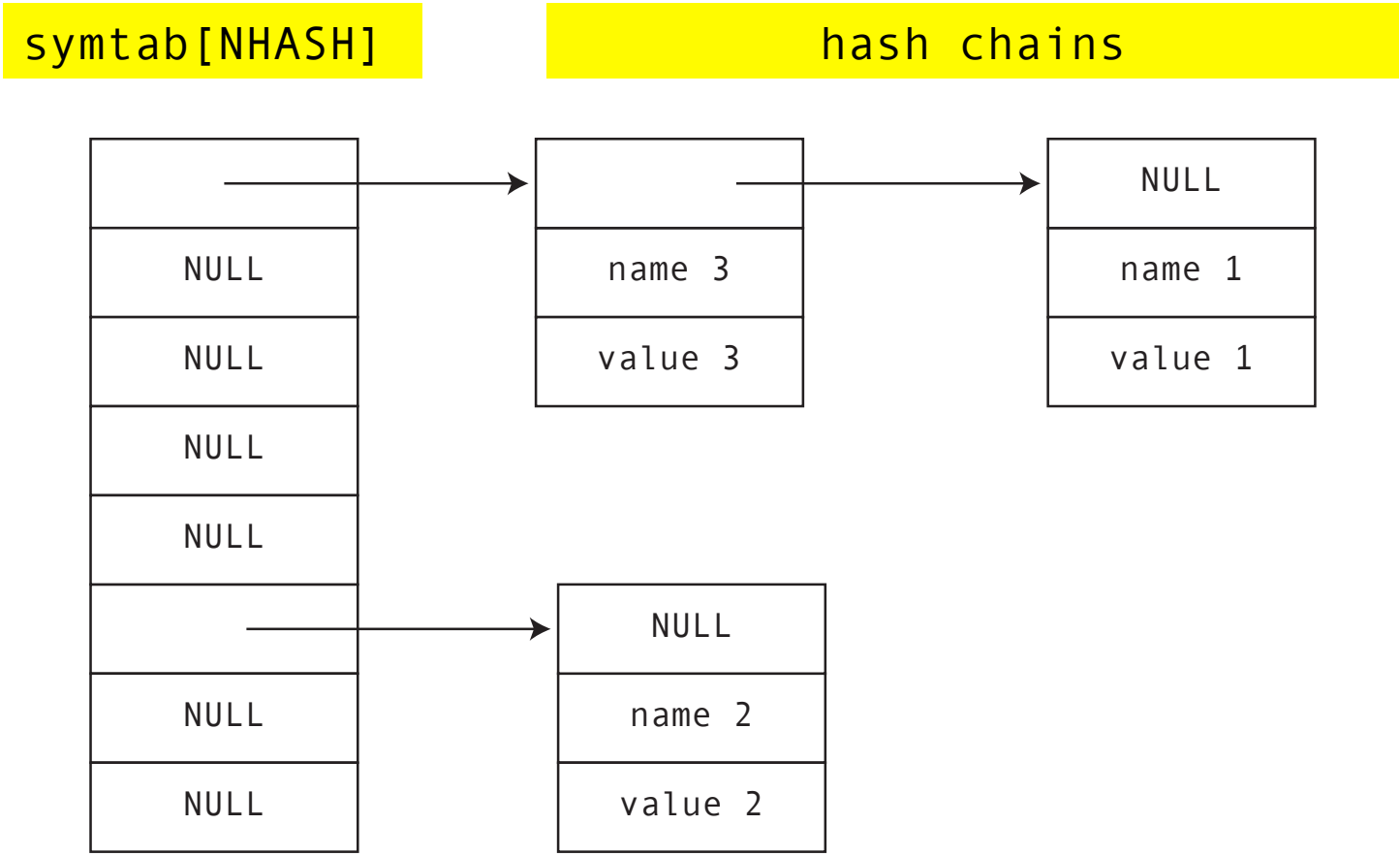
Hash tables

- These combine:
  - arrays
  - lists
  - some mathematics
- Efficient structure for storing and retrieving dynamic data
- Typical application for hash tables: symbol tables
  - Associates some value (the **data**)...
  - with each member of a dynamic set of strings (the **keys**)
- Lots of places where hash tables are used

The idea

- Hash tables work on the following principle
  - Pass a key to the **hash function**
  - The hash function produces a **hash value**
  - These values will be **evenly distributed** through a modest-sized integer range
- The hash value is then used as an array index
  - In C the usual style is to associate each hash value / array index with a list of items that share the hash
  - Each such list (sometimes called a **hash chain**) is known as a **bucket**

Example


symtab[NHASH]
hash chains
NULL
NULL
NULL
NULL
NULL
NULL
NULL
name 3
value 3
NULL
name 1
value 1
NULL
name 2
value 2

The practice

- Hash functions are pre-defined
- Array is sized appropriately (usually at compile time)
- Each element of the array is a list that **chains** together the items that share a hash value (i.e., hash chain)
- Equivalently:
  - A hash table of n items ...
  - ... is an array of lists whose average length is (n/array size)
- Retrieving an item is an O(1) operation provided the following two conditions hold:
  - we pick a good hash function
  - the lists do not grow too long

Element type

- A hash table is an array of lists...
  - ... therefore we can re-use the element type used for lists
- Maintaining individual hash chains is similar to maintaining individual lists
- Once we have a good hash function, the code falls out easily
  - just pick the hash bucket...
  - ... and walk along the list looking for a perfect match

```
typedef struct Nameval Nameval;
struct Nameval {
    char    *name;
    int      value;
    Nameval *next; /* in chain */
};

/* symbol table */
Nameval *symtab[NHASH];
```

Lookup / insertion routine

- If item is found:
  - It is returned
- If item is not found and create flag is set:
  - add item to the table.

```
Nameval *lookup (char *name, int create, int value)
{
    int h
    Nameval *sym;

    h = hash(name);
    for (sym = symtab[h]; sym != NULL; sym = sym->next) {
        if (strcmp(name, sym->name) == 0) { return sym; }
    if (create) {
        sym = (Nameval *) emalloc(sizeof(Nameval));
        sym->name = name; /* assumed allocated elsewhere */
        sym->value = value;
        sym->next = symtab[h];
        symtab[h] = sym;
    }
    return sym;
}
```

Why combine lookup & insertion?

- This is a common combination
- Without it we often duplicate effort
- (Also causes the hash function to be executed twice for the same item.)
- This is a stylistic point (but one which can save a bit of tedium and reduce possibility of buggy code)

```
/*
 * The code the might result if we
 * keep lookup and insertion separate.
 */

if (lookup("name") == NULL) {
    additem(newitem("name", value));
}
```

Two more questions

- How big should the array be?
  - In general: make it large enough that each hash chain will have at most a few elements
  - Example: A compiler might have an array size of a few thousand
- How is the hash function computed?
  - Must be deterministic (i.e., produce same value each time for same key)
  - Must be fast
  - Must distribute data uniformly through the array
  - Lots of research exists that investigates these properties of hash functions

Possible hash function

- Common hashing algorithm for strings:
  - Build a value by adding each byte of the string to a multiple of the has so far
  - Multiplication spreads bits from the new byte through the value so far
- Empirically: the values 31 and 37 prove to be good choices for ASCII strings

```
#define MULTIPLIER 31

/* hash: compute hash value of string */
unsigned int hash (char *str) {
    unsigned int h;
    unsigned char *p;

    h = 0;
    for (p = (unsigned char *) str; *p != '\0'; p++) {
        h = MULTIPLIER * h + *p;
    }
    return h % NHASH;
}
```

Summary

- malloc() is an important tool
  - but it can be tricky at first to use correctly
  - is usually paired with free()
- dynamically-allocated memory needed for implementing many kinds of data structures
- two big takeaways
  - arrays can be resized (and that's handy as arrays are easy to use)
  - lists are used in many data structures (so it is important to know how to write routines that add to, traverse through, and remove from lists)

Colophon

- Some code examples are from "The Practice of Programming" (Addison-Wesley) © 1999 Brian W. Kernighan and Rob Pike