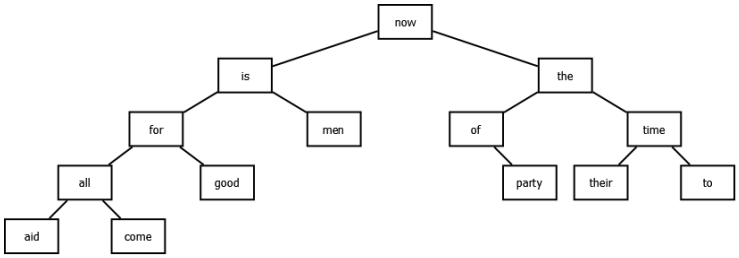
LECTURE 20: SELF-REFERENTIAL STRUCTURES (K&R §§ 6.5, 6.6)

SELF-REFERENTIAL STRUCTURES

Example:

Goal:	We want to count the number of times we see each unique word in some input.
Solution:	Create a binary tree so that we can quickly locate words we've already seen as we read through the input.

Binary tree for the sentence:
"now is the time for all good men to come to the aid of their party"



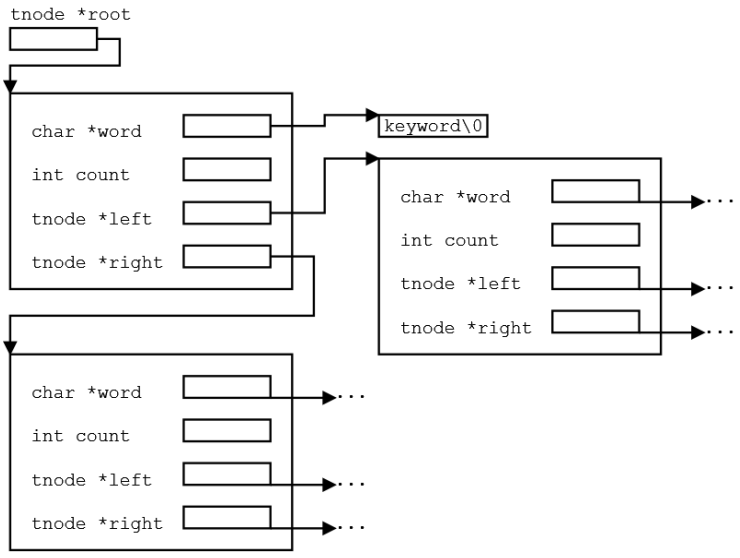
At any node, the left subtree only has words that are lexicographically less than the word at the node. The right subtree has words that are greater.

Implement a "node" structure with the following members:

1. a pointer to the text of the word (char *)
2. a count for the number of times it has been seen (int)
3. a pointer to the left child node (struct node *)
4. a pointer to the right child note (struct node *)

```
struct tnode {     /* the tree node struct */
   char *word;     /* points to the word at this node  */
   int count;      /* has a count of occurances */
   struct tnode *left;     /* a word < one at this node */
   struct tnode *right;    /* a word > one at this node */
};
```

Note:	It's ok to have a pointer to a struct of the same type in its own definition.
	It's not ok to have a struct itself in its own definition (but it's ok to have a different struct as a member).



Each node is a struct tnode with a string value and a count of occurrences.

Each node also contains a pointer to a left child and a right child.

Each child is another struct tnode.

ADDTREE

```
struct tnode *talloc(void); /* see def below */
char *strdup(char *);       /* see def below */

/* addtree: add a node with w, at or below p */
struct treenode *addtree(struct tnode *p, char *w) {
  int cond;
  if (p == NULL) { /* a new word has arrived */
    p = talloc();  /* make a new node */
    p->word = strdup(w);
    p->count = 1;
    p->left = p->right = NULL;
  }
  else if ((cond = strcmp(w, p->word)) == 0)
    p->count++;       /* repeated word */
  else if (cond < 0) /* less than into left subtree */
    p->left = addtree(p->left, w);
  else               /* greater than into right subtree */
    p->right = addtree(p->right, w);
  return p;
}
```

addtree() is recursive: when we add a node, it will either be the root or a child somewhere lower in the tree.

we pass struct tnode *p as an argument
we return the same type

any time a new node is created,
  root,
  p->left, or
  p->right
in its parent node will be set for the first time based on the return value from addtree.

TALLOC / STRDUP

This is the same thing we've been doing with malloc, we just wrap it in a function dedicated specifically to allocating memory for tnodes (hence the name talloc).

```
struct tnode *talloc(void) {
  return (struct tnode *) malloc (sizeof(struct tnode));
}
```

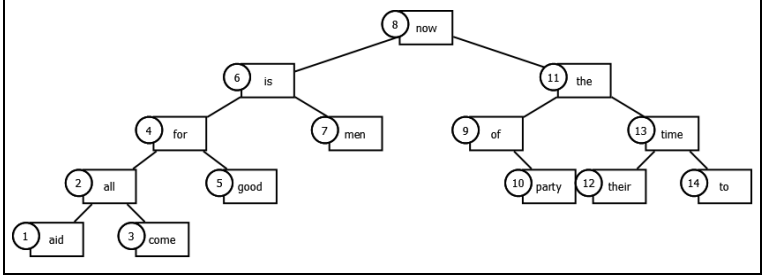
Similarly, strdup is allocating the right amount of memory for the word we read in.

```
char *strdup(char *s) {
  char * p;
  p = (char *) malloc (strlen(s)+1); /* +1 for '\0' */
  if (p != NULL)   /* if malloc didn't fail */
    strcpy(p, s);  /* library function in string.h */
  return p;
}
```

TREEPRINT

```
void treeprint (struct tnode *p) {
  if (p != NULL) {
    treeprint (p->left);
    printf (%4d %s\n", p->count, p->word);
    treeprint (p->right);
  }
}
```

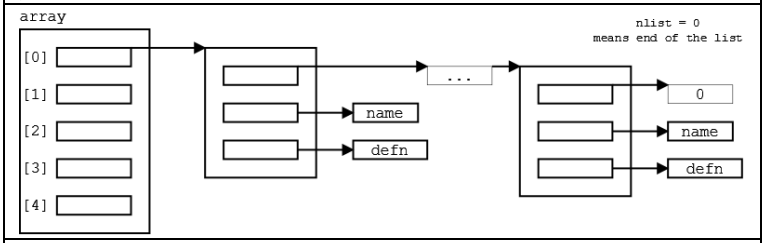
More recursion; make sure you understand how this works. The numbers indicate the order the nodes will be printed.



ANOTHER APPLICATION OF SELF-REFERENTIAL STRUCTS: TABLE LOOKUP (HASH TABLE)

Struct for an element in a linked list of name/definition pairs:

```
struct nlist {
  struct nlist *next;  /* link to next */
  char *name;          /* word name */
  char *defn;          /* word definition */
};
```



Incoming name is converted to a nonnegative integer, which is used to index into an array of pointers.

Each pointer in the array points to a linked list comprised of nlist structures.