CS 241
Data Organization
Binary Trees

Brooke Chenoweth

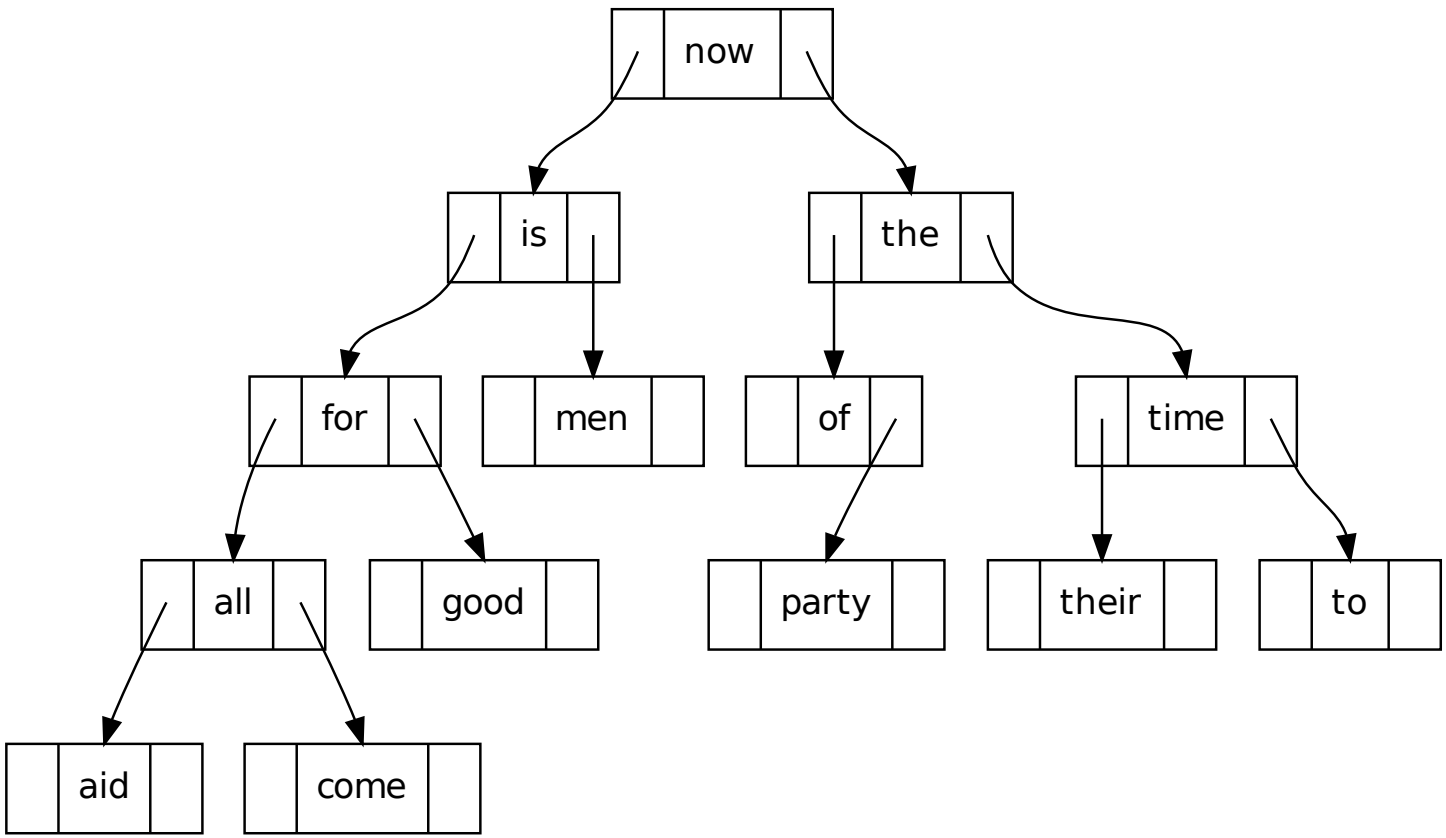
University of New Mexico

Fall 2014

Binary Tree: Kernighan and Ritchie 6.5

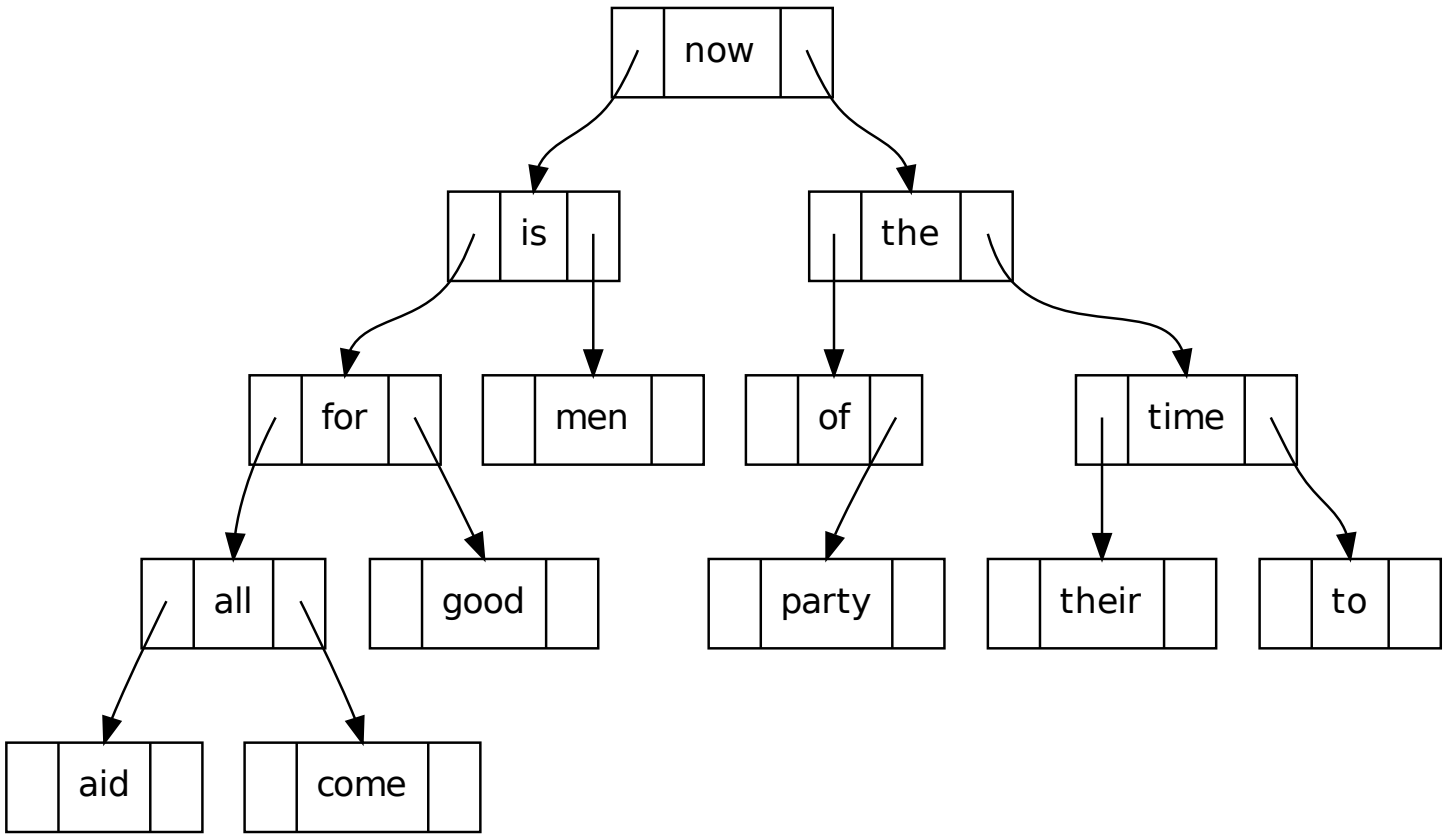
Read a file and count the occurrences of each word.

```
now is the time for all good men to come to the
aid of their party
```



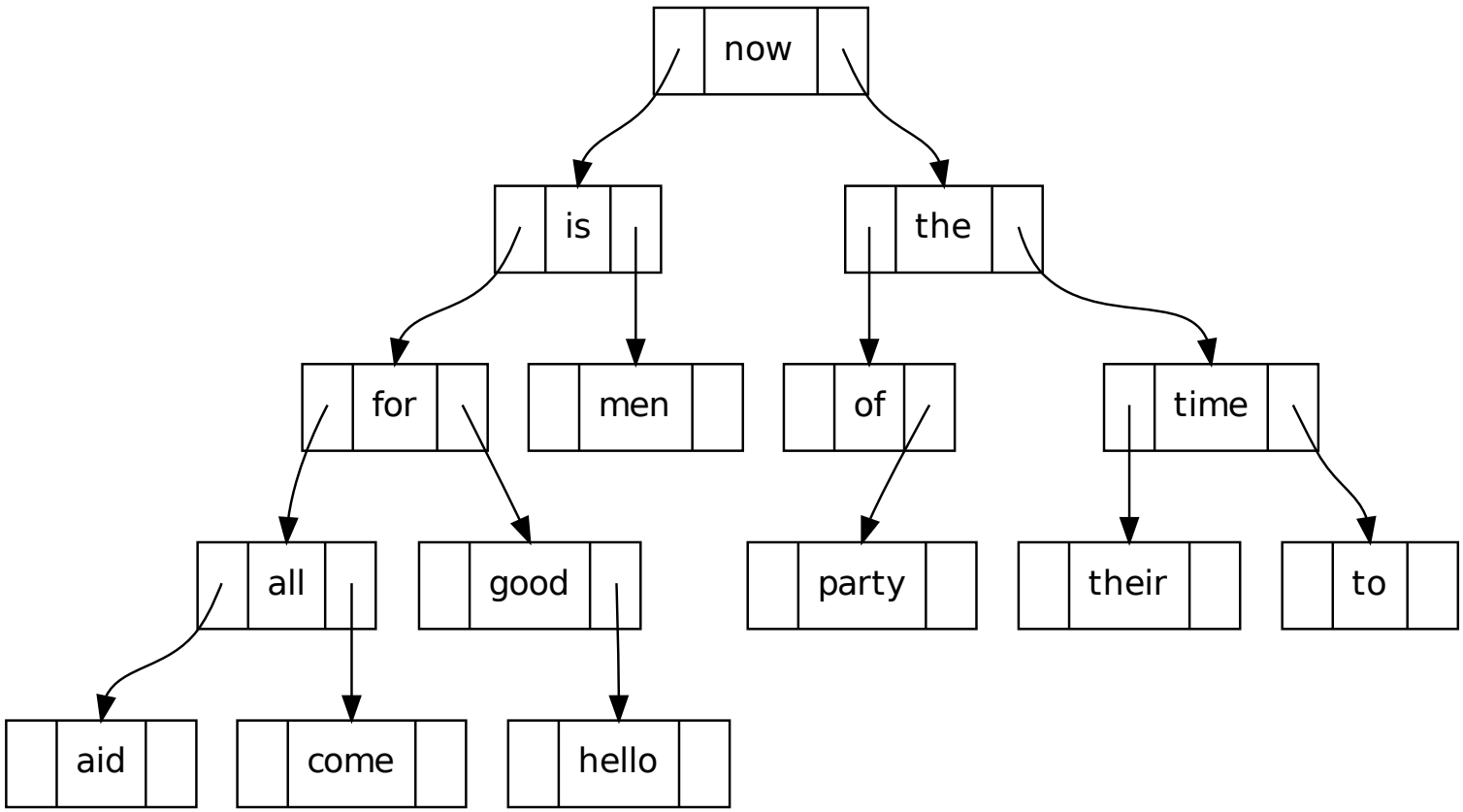
Adding Node to Ordered Tree

If a node were added with the word "hello", where would it be placed?



Adding Node to Ordered Tree

If a node were added with the word "hello", where would it be placed?



Binary Tree: tnode

The structure, tnode, is used for each node of the binary tree.

```
struct tnode
{
  char *word;
  int count;
  struct tnode *left;
  struct tnode *right;
};
```

This is called a self-referential structure since it contains pointers to other tnodes.
An instance of this struct allocates space for a pointer to a char array, two pointers to other tnodes and an int. On a 64-bit address machine, this is a total of 16 bytes.

Binary Tree: `talloc` (NOT a Library Function)

1. Allocate memory for a tree node.
2. In this binary tree, nodes are added to leaves. Thus, initialize the node's children to NULL.
3. Call strCopyMalloc to allocate space for the word and to copy it from the input buffer into the allocated space.

```
struct tnode *talloc(char *newWord)
{
  struct tnode *node =
              malloc(sizeof(struct tnode));
  node->word = strCopyMalloc(newWord);
  node->left = NULL;
  node->right = NULL;
  node->count = 1;
  return node;
}
```

Binary Tree: `strCopyMalloc`

1. Allocated memory for a copy of `newWord`.
2. Copy each character from `newWord` into the allocated block.
3. Return a pointer to the start of the allocated block.

```
char *strCopyMalloc(char *source)
{
  char *sink;
  sink = malloc(strlen(source)+1);
  if (sink != NULL) strcpy(sink, source);
  return sink;
}
```

Binary Tree: Memory Leaks

In Kernighan and Ritchie's binary tree, memory is allocated and *never freed!*
MEMORY LEAK WARNING: DO NOT free a node until:

- Its children have been freed, or pointers to its children have been saved somewhere else.
- Its word has been freed.

```
struct tnode *root;
root = talloc("Memory");
root->right = talloc("Leak");
root->left = talloc("Bad");
free(root); /* Leaves "unreachable" memory. */
```

Binary Tree: `freeSubTree`

Recursive function that frees all allocated memory in a subtree.

```
1  void freeSubtree(struct tnode *node)
2  {
3    if (node == NULL) return;
4    freeSubtree(node->left);
5    freeSubtree(node->right);
6    free(node->word);
7    free(node);
8  }
```

Any references to `node` (such as would be in node's parent) MUST NOT BE USED AFTER calling this. Best practice is to set such references to NULL.

- Is this done here?
- If not, could it be done here?
- If so, between which lines and with what code?

Binary Tree: Simple Test Case

This `main()` demonstrates usage and offers a simple test of creating, setting, printing, and freeing `tnode`.

```
void main(void)
{
  /* Can you tell I just stole Joel's code here? */
  struct tnode *root;
  root = talloc("joel");
  root->left = talloc("cool");
  root->right = talloc("inspirational");
  printf("node: %s (L)=%s, (R)=%s\n", root->word,
         root->left->word, root->right->word);
  freeSubtree(root);
  root = NULL; /* "Best practice"           */
               /* (no effect on valgrind) */
}
```

Binary Tree: No Leaks Are Possible

Using valgrind results in something like the following:

```
node: joel (L)=cool, (R)=inspirational
==24066== HEAP SUMMARY:
==24066== in use at exit: 0 bytes in 0 blocks
==24066== total heap usage:
6 allocs, 6 frees, 120 bytes allocated
==24066==
==24066== All heap blocks were freed -- no leaks are possible
```

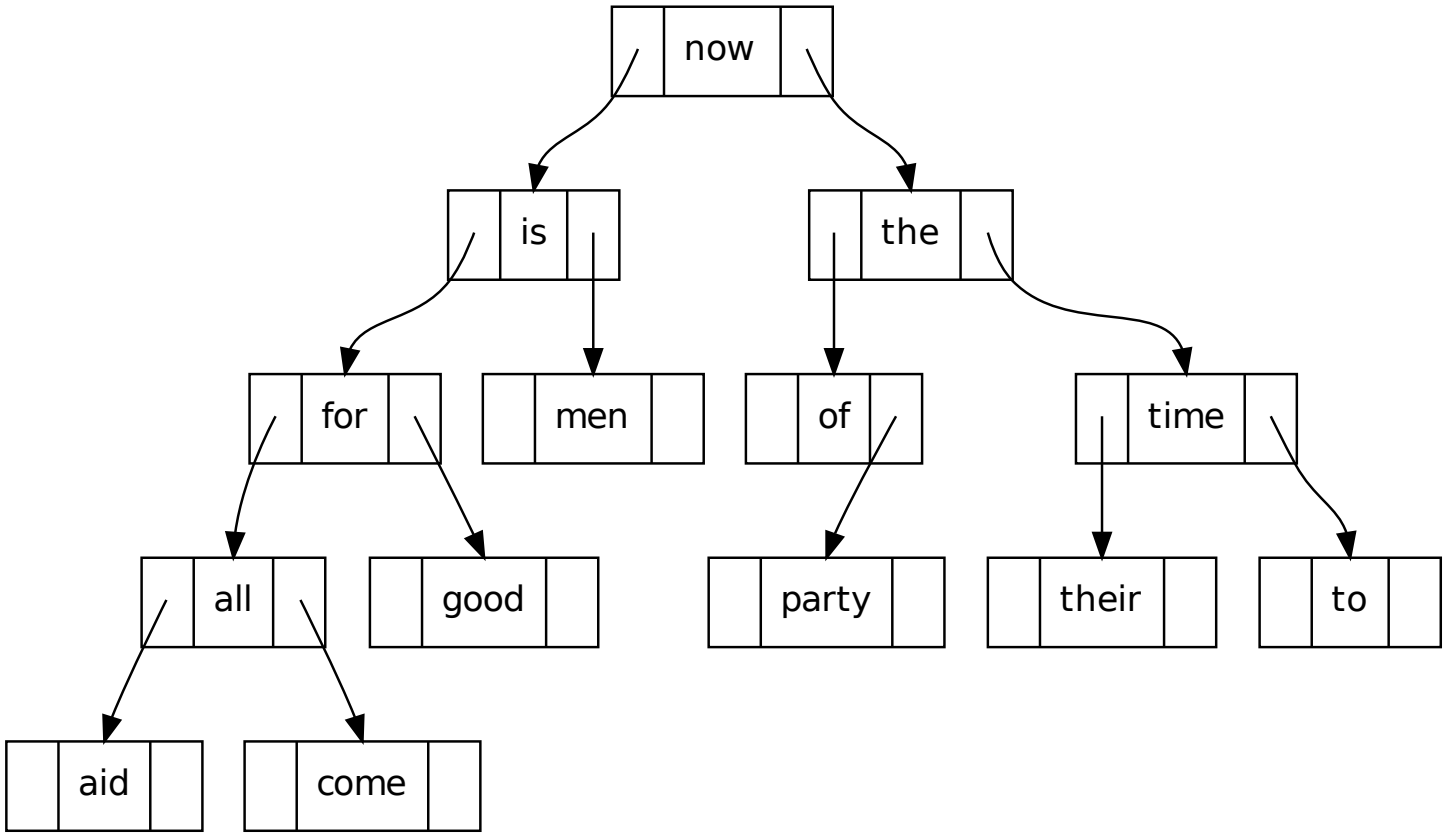
Tree Traversal

Depth First Explore as far as possible along each branch before backtracking

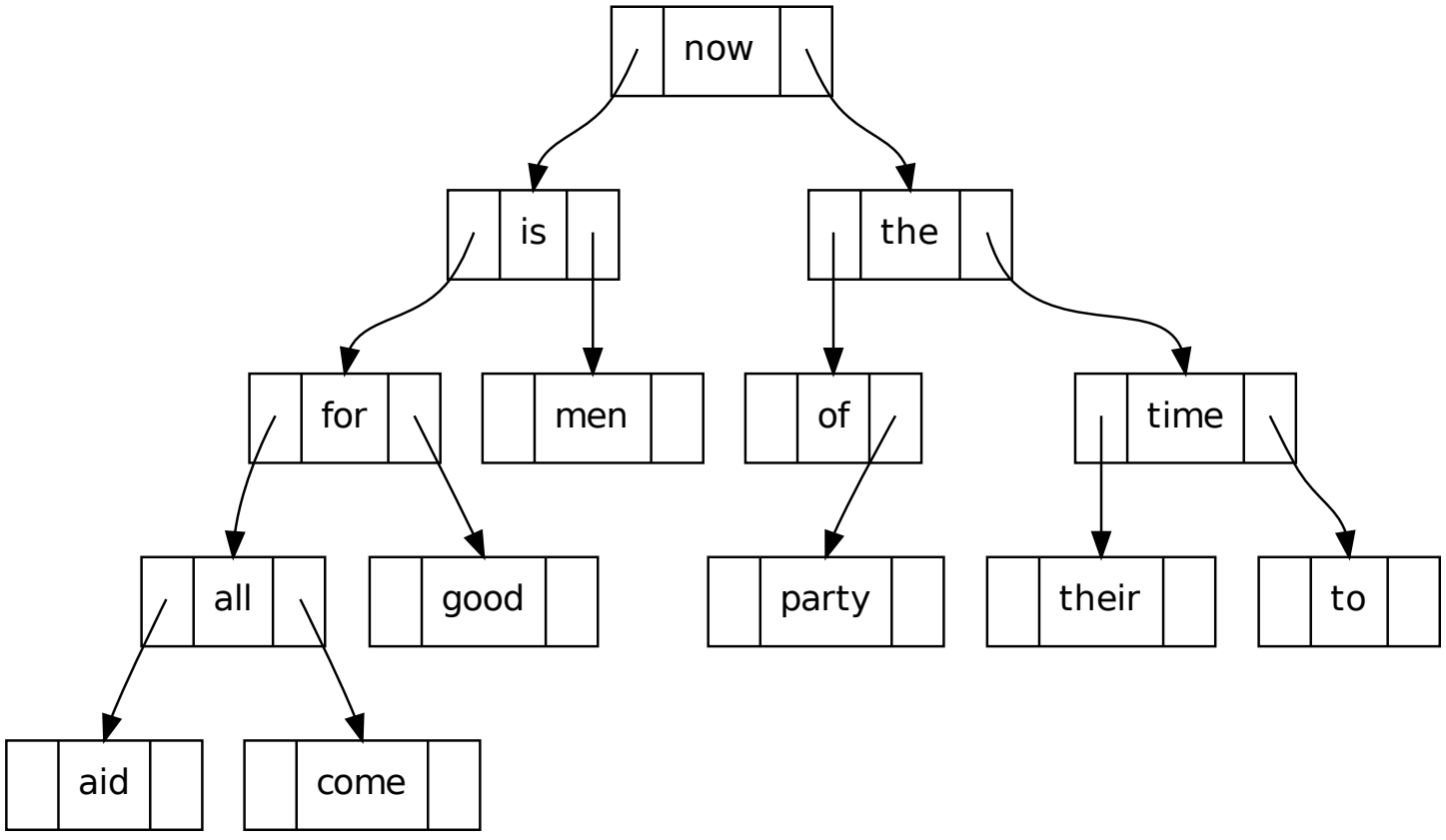
- Pre-order – Root, left subtree, right subtree
- In-order – Left subtree, root, right subtree
- Post-order – Left subtree, right subtree, root

Breadth First Visit every node on a level before going to lower level. (Also known as *level-order*)

Tree Traversal – Example



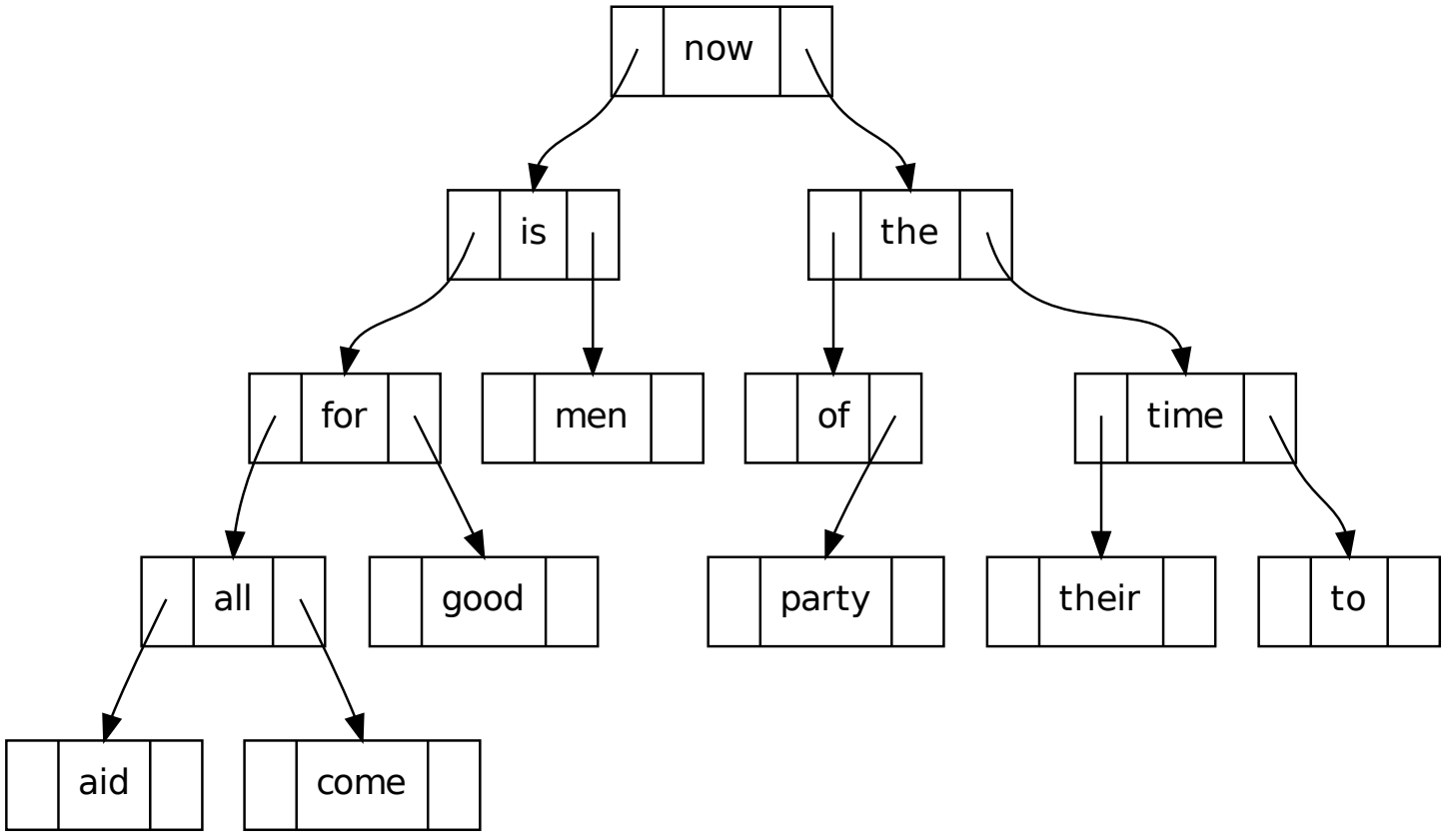
Tree Traversal – Pre-order



Root, then children:

now is for all aid come good men the of party time their to

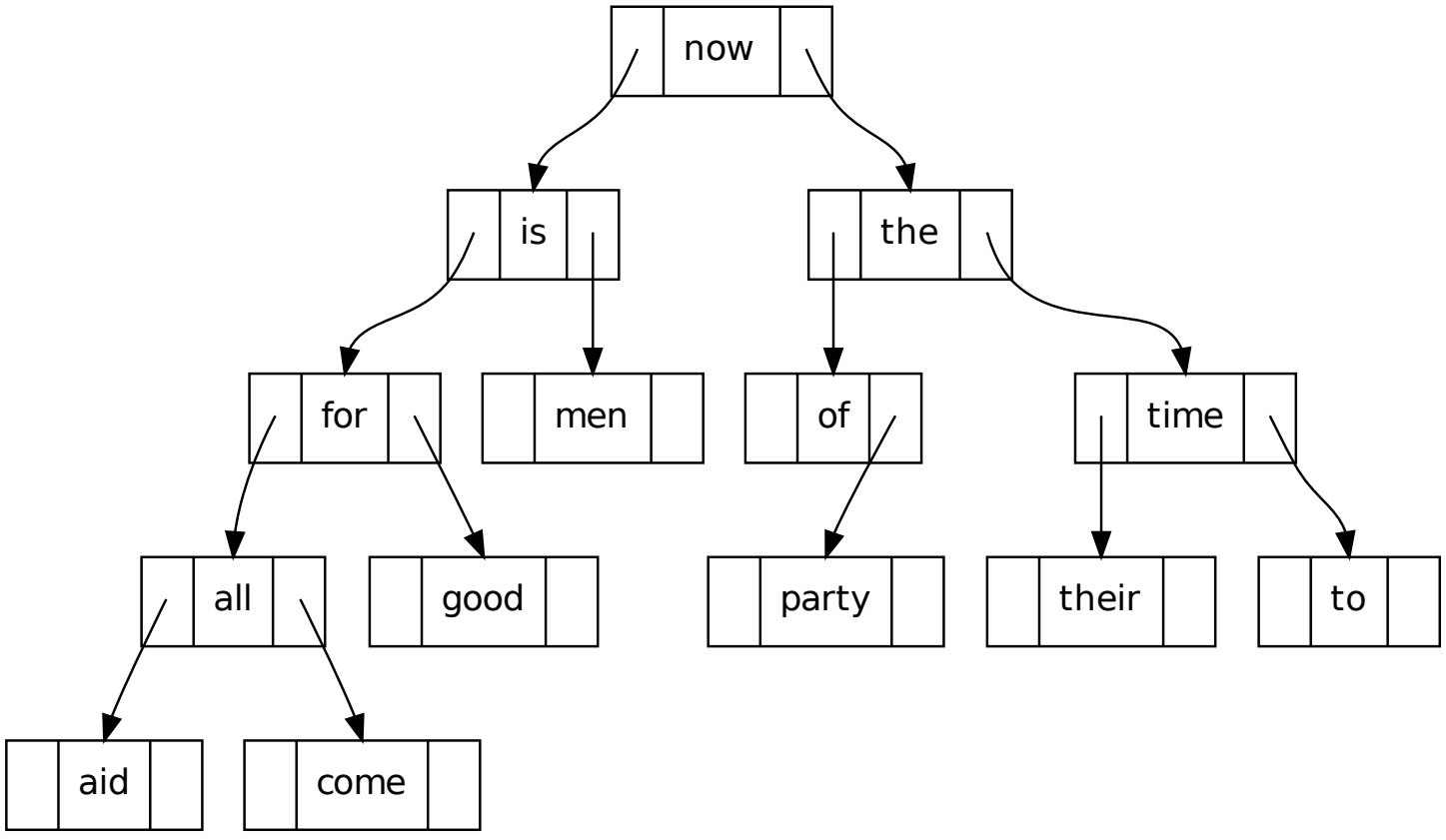
Tree Traversal – In-order



Left, root, right:

aid all come for good is men now of party the their time to

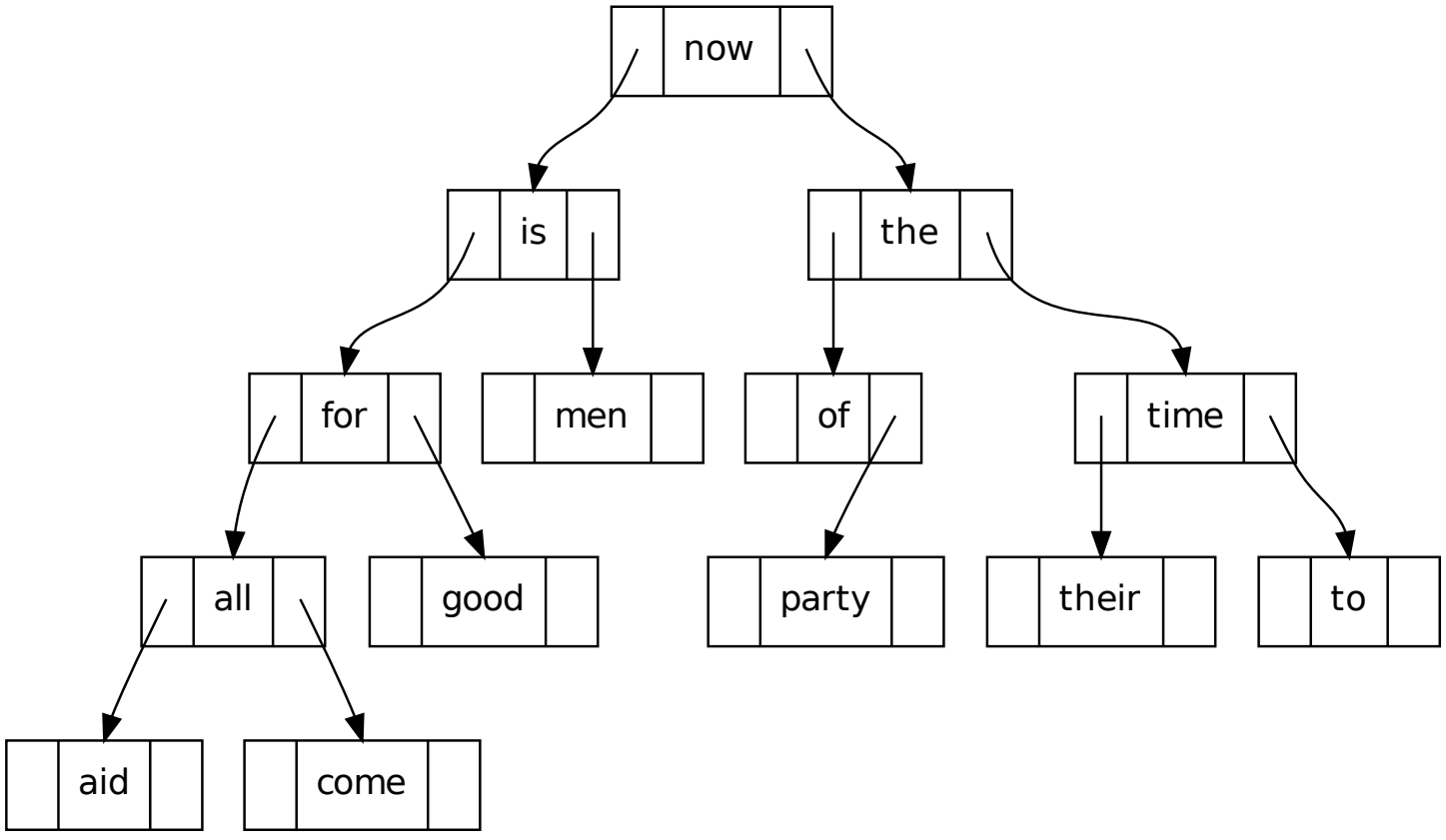
Tree Traversal – Post-order



Children, then root:

aid come all good for men is party of their to time the now

Tree Traversal – Breadth-first



One level at a time:

now is the for men of time all good party their to aid come

Tree Traversal – Code

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

What sort of traversal happens here?

How could we traverse the tree in a different order?

Breadth-first Traversal Algorithm

1. Create a queue to hold tree nodes
2. Add root node to the queue
3. While queue is not empty:
    - Remove node from queue and visit it.
    - Add node's children to queue.

A queue is a FIFO structure. How might we implement it?
A stack is a LIFO structure. What would happen if we replaced the queue with a stack?