COP 3502: Computer Science I
Spring 2004

– Note Set 17 –
Binary Trees

Instructor : Mark Llewellyn
markl@cs.ucf.edu
CC1 211, 823-2790
http://www.cs.ucf.edu/courses/cop3502/spr04

School of Electrical Engineering and Computer Science
University of Central Florida

Linked List Implementation of a Queue

```
// MJL – 3/16/2004
// A small queue implementation
#include <stdio.h>

// Struct used to form a queue of integers.
struct queue {
  int data;
  struct queue *next;
};

// Prototypes
int enqueue(struct queue **rear, int num);
struct queue* dequeue(struct queue **front);
int empty(struct queue *front);
void init(struct queue **front, struct queue **rear);
```


Linked List Implementation of a Queue (cont.)

```
int main() {

  struct queue *queue1, *temp;
  int tempval;

  init(&queue1);

  if (!enqueue(&queue1, 3))
    printf("Enqueue failed.\n");
  if (!enqueue(&queue1, 5))
    printf("Enqueue failed.\n");

  temp = dequeue(&queue1);
  if (temp !=NULL)
    printf("Dequeue = %d\n", temp->data);
```


Linked List Implementation of a Queue (cont.)

```
if (empty(queue1))
    printf("Empty queue\n");
  else
    printf("Contains elements.\n");

  temp = dequeue(&queue1);
  temp = dequeue(&queue1);
  return 0;
}
void init(struct queue **front, **rear) {
  *front = NULL;
  *rear = NULL;
}
```


Linked List Implementation of a Queue (cont.)

```
// Pre-condition: rear points to the tail of the queue.
// Post-condition: a new node storing num will be added to the queue
//        if memory is available. In this case a 1 is returned. If no memory is found,
//         no enqueue is executed and a 0 is returned.

int enqueue(struct queue **rear, int num) {

  struct queue *temp;
  // Create temp node and link it to the rear of the queue.
  temp = (struct queue *)malloc(sizeof(struct queue));

  if (temp != NULL) {
    temp->data = num;
    temp->next = NULL;
    *rear->next = temp;
    *rear = temp;
    return 1;
  }
  else
    return 0;
}
```

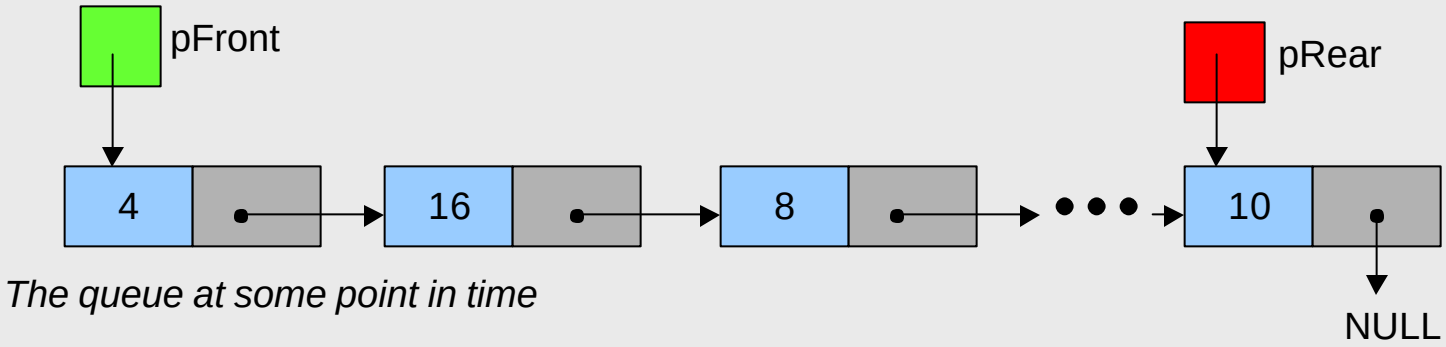

Linked List Implementation of a Queue (cont.)

```
// Pre-condition: front points to the head of the queue
// Post-condition: A pointer to a node storing the head value from the
//                 queue will be returned. If no value exists, the pointer
//                 returned will be pointing to null.

struct queue* dequeue(struct queue **front) {

  struct queue *temp;
  temp = NULL;

  if (*front != NULL) {
    temp = (*front);
    *front = (*front)->next;
    temp -> next = NULL;
  }
  return temp;
}
```

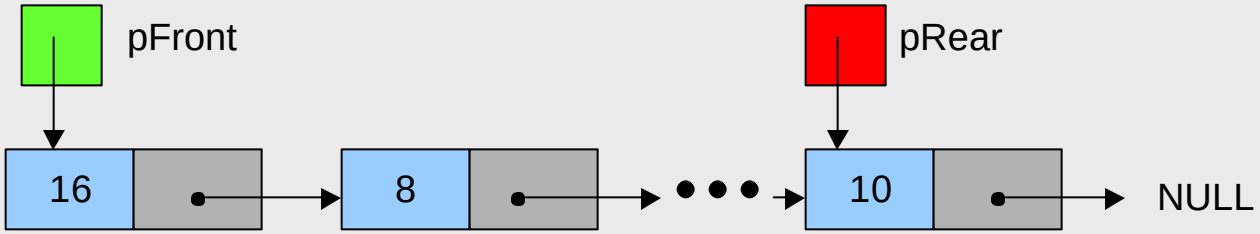
Linked List Implementation of a Queue (cont.)

```
// Pre-condition: front points to the head of the queue
// Post-condition: returns true if the queue is empty, false otherwise.
int empty(struct queue *front) {
  if (front == NULL)
    return 1;
  else
    return 0;
}
```

Visualizing a Queue Implemented with a Linked List [Structure Diagrams]

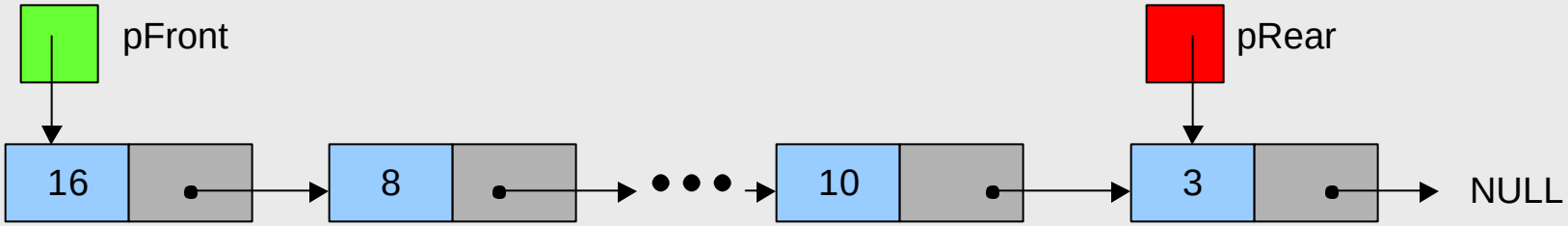


The queue at some point in time



The queue after a dequeue operation

Visualizing a Queue Implemented with a Linked List [Structure Diagrams] (cont.)



The queue after an enqueue(3) operation

Binary Trees

- A binary tree is a data structure that is made up of nodes and pointers, much in the same way that a linked list is structures. The difference between them lies in how they are organized.

- A linked list represents a linear or predecessor/successor relationship between the nodes of the list. A tree represents a hierarchical or ancestral relationship between the nodes.

- In general, a node in a tree can have several successors (called children). In a binary tree this number is limited to a maximum of 2.

Binary Trees (cont.)

- The top node in the tree is called the root.
- Every node in a binary tree has 0, 1, or 2 children.
- There are actually two different approaches to defining a tree structure, one is a recursive definition and the other is a non-recursive definition.
- The non-recursive definition basically considers a tree as a special case of a more general data structure, the *graph*. In this definition the tree is viewed to consist of a set of nodes which are connected in pairs by directed edges such that the resulting graph is connected (every node is connected to a least one other node – no node exists in isolation) and cycle-free. This general definition does not specify that the tree have a root and thus a *rooted-tree* is a further special case of the general tree such every one of the node except the one designated as the root is connected to at least one other node. In certain situations the non-recursive definition of a tree has certain advantages, however, for our purposes we will focus on the recursive definition of a tree which is:

Binary Trees (cont.)

Definition: A *tree t* is a finite, nonempty set of nodes,

$$t = \{r\} \cup T_1 \cup T_2 \cup \ldots \cup T_n$$

with the following properties:

1. A designated node of the set, *r*, is called the *root* of the tree; and
2. The remaining nodes are partitioned into $n \geq 0$ subsets $T_1, T_2, \ldots, T_n$ each of which is a tree (called the *subtrees of t*).

For convenience, the notation $t = \{r, T_1, T_2, \ldots, T_n\}$ is commonly used to denote the tree *t*.

- A complete set of terminology has evolved for dealing with trees and we'll look at some of this terminology so that we can discuss tree structures with some degree of sophistication.
- As you will see the terminology of trees is derived from mathematical, genealogical, and botanical disciplines.

Binary Trees (cont.)

- Rooted Tree: (from the non-recursive definition) A tree in which one node is specified to be the *root*, (call it node *c*). Every node (other than *c*), call it *b* is connected by exactly one edge to exactly one other node, call it *p*. Given this situation, *p* is *b*'s parent. Further, *b* is one of *p*'s children.

- Degree of a node: The number of subtrees associated with a particular node is the *degree* of that node. For example, using our definition of a tree the node designated as the root node *r* has a degree of *n*.

- Leaf Node: A node of degree 0 has no subtrees and is called *leaf* node. All other nodes in the tree have degree of at least one and are called *internal* nodes.

- Child Node: Each root r_i of subtree t_i of tree *t* is called a *child* of *r*. The term *grandchild* is defined in a similar fashion as is the term *great-grandchild*.

Binary Trees (cont.)

- Parent: The root node r of tree t is the *parent* of all the roots r_i of the subtrees t_i, $1 < i \leq n$. The term *grandparent* is defined in a similar manner.

- Siblings: Two roots r_i and r_j of distinct subtrees t_i and t_j of tree t are called *siblings*. (These are nodes which have the same parent.)

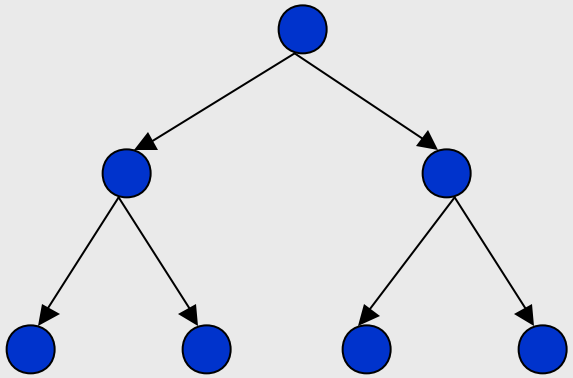
- The definitional restrictions placed on a binary tree when compared to a general tree give rise to certain properties that a binary tree will exhibit that are not exhibited by a general tree. Some of these properties and corresponding terminology are defined below.

  Number of nodes in a binary tree: A binary tree t of height h, $h \geq 0$, contains at least h and at most $2^h - 1$ nodes.

- Height of a binary tree: The height of a binary tree that contains n, $n \geq 0$, nodes is at most n and at least $\lceil \log_2 (n+1) \rceil$.

Binary Trees (cont.)

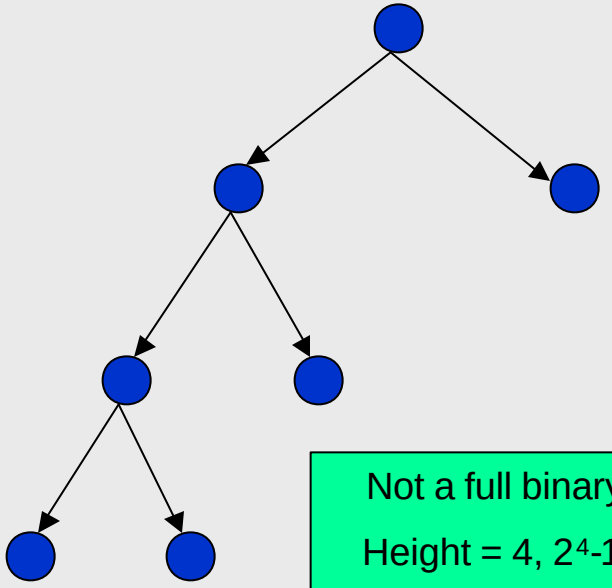
- Full binary tree: A binary tree of height *h* that contains exactly 2^h-1 nodes is called a *full binary tree*. (Each level *i* in the tree contains the maximum number of nodes, i.e., every node in level *i-1* has two children.)



A full binary tree

Height = 3, $2^3-1 = 7$

Number of nodes = 7



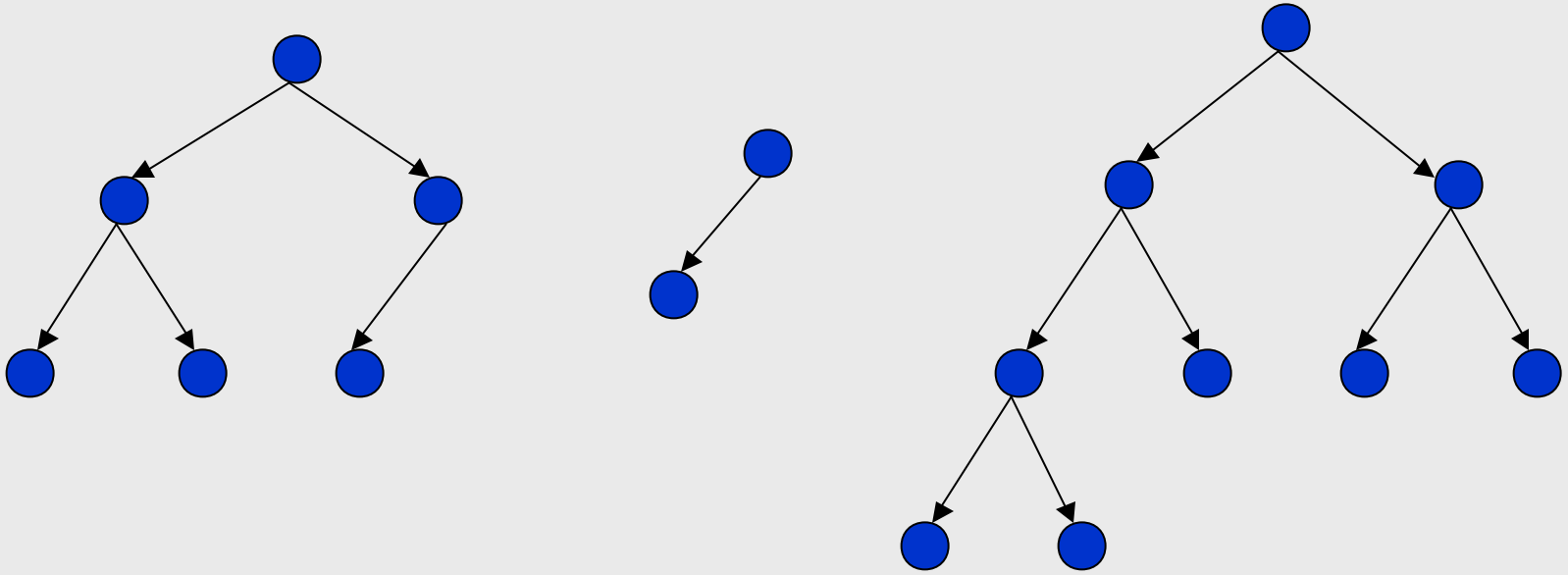
Not a full binary tree

Height = 4, $2^4-1 = 15$

Number of nodes = 7

Binary Trees (cont.)

- Complete binary tree: A binary tree of height *h* in which every level except level 0 has the maximum number of nodes and level 0 nodes are placed from left to right on the level with no missing nodes. Note that a full binary tree is a special case of a complete binary tree in which level 0 contains the maximum number of nodes. Some complete binary trees are shown below.



Binary Tree Implementation

- A binary tree has a natural linked representation. A separate pointer is used to reference the root of the tree.
- Each node has a left and right subtree which is reachable with pointers.

```
struct treeNode {
        int data;  //any data type can be used
        struct treeNode *left;
        struct treeNode *right;
};
```

- We'll look at the specific details for implementing binary trees a bit later, for now we'll assume a dynamic structure with a node structure similar to that shown above.

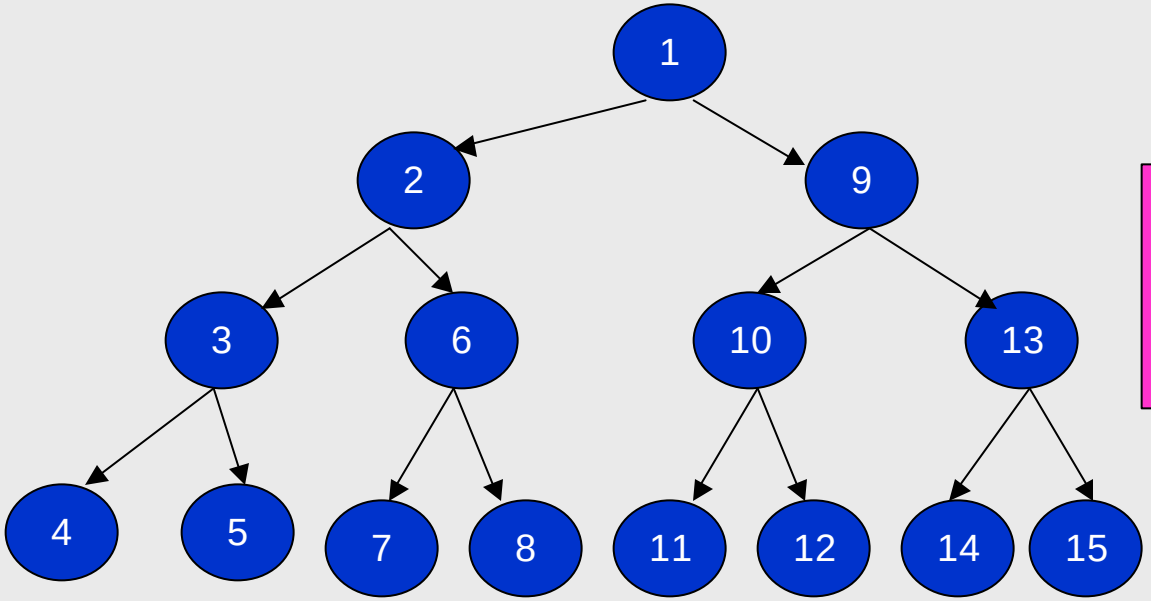
Binary Tree Traversals

- As with any data structure, moving through the structure is a fundamental necessity. We've seen algorithms to traverse a singly-linked list and are familiar with the basic concept of data structure traversal.
- For binary trees, there are basically three different traversals that can be defined. The tree different traversal algorithms arise as a result of the different ways in which the root node of a tree can be "visited" with respect to when its children are "visited".
  - There are actually more than three traversal techniques, but some of the symmetric cases are never used.

Binary Tree Traversals (cont.)

Preorder Traversal

- In a preorder traversal, the root node of the tree is visited before either the left or right child of the root node is visited.
- Labeling the right child as R, the left child as L, and the root node as N, the order of visitation in a preorder traversal is: NLR.

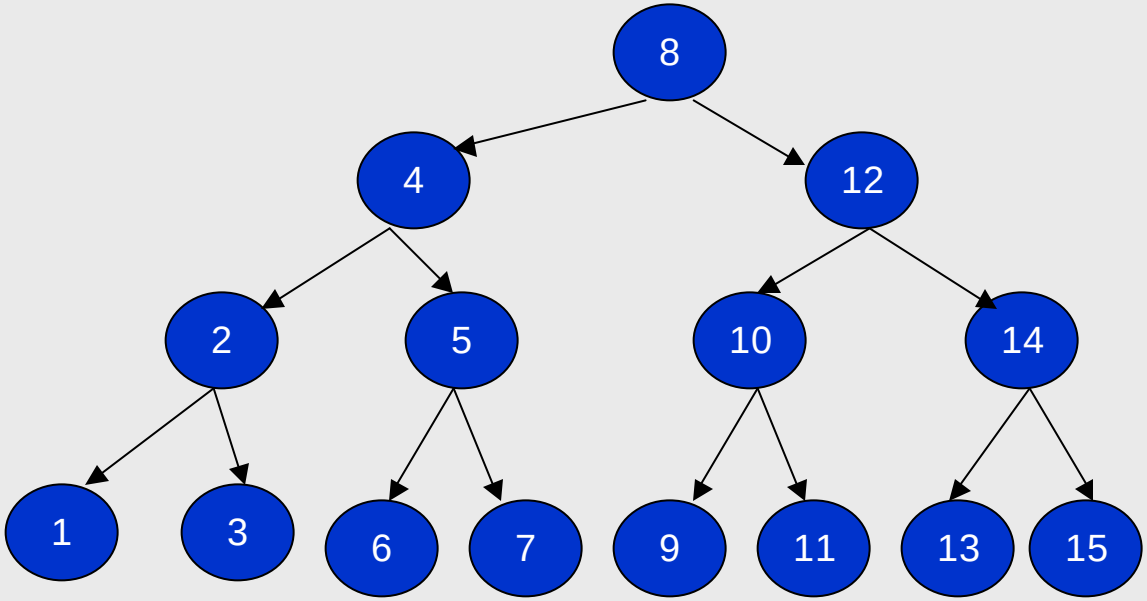


The numbers shown in each node in the tree to the left indicate the order in which the node is visited in a preorder traversal of the tree.

Binary Tree Traversals (cont.)

Inorder Traversal

- In an inorder traversal, the left child is visited before the root node is visited and the right child is visited after the root node is visited.
- Labeling the right child as R, the left child as L, and the root node as N, the order of visitation in an inorder traversal is: LNR.

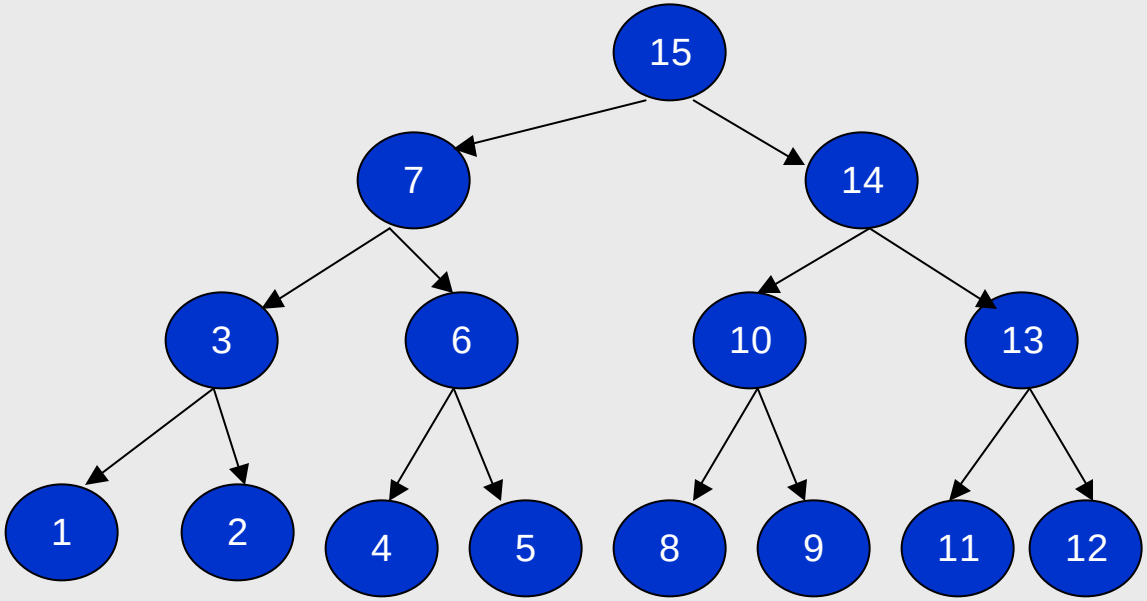


The numbers shown in each node in the tree to the left indicate the order in which the node is visited in an inorder traversal of the tree.

Binary Tree Traversals (cont.)

Postorder Traversal

- In a postorder traversal, both the left child and the right child are visited before the root node is visited.
- Labeling the right child as R, the left child as L, and the root node as N, the order of visitation in a postorder traversal is: LRN.



The numbers shown in each node in the tree to the left indicate the order in which the node is visited in a postorder traversal of the tree.

Binary Tree Traversal Algorithms

Preorder Traversal Algorithm

```
void preorder( struct treeNode *p)
{
   if (p != NULL)
       {       printf("%d\n", p->data);  // this is the "visit"
               preorder(p->left);
               preorder(p->right);
}
}
```


Binary Tree Traversal Algorithms (cont.)

Inorder Traversal Algorithm

```
void inorder( struct treeNode *p)
{
      if (p != NULL)
      {         inorder(p->left);
                printf("%d\n", p->data);  // this is the "visit"
                inorder(p->right);
}
}
```


Binary Tree Traversal Algorithms (cont.)

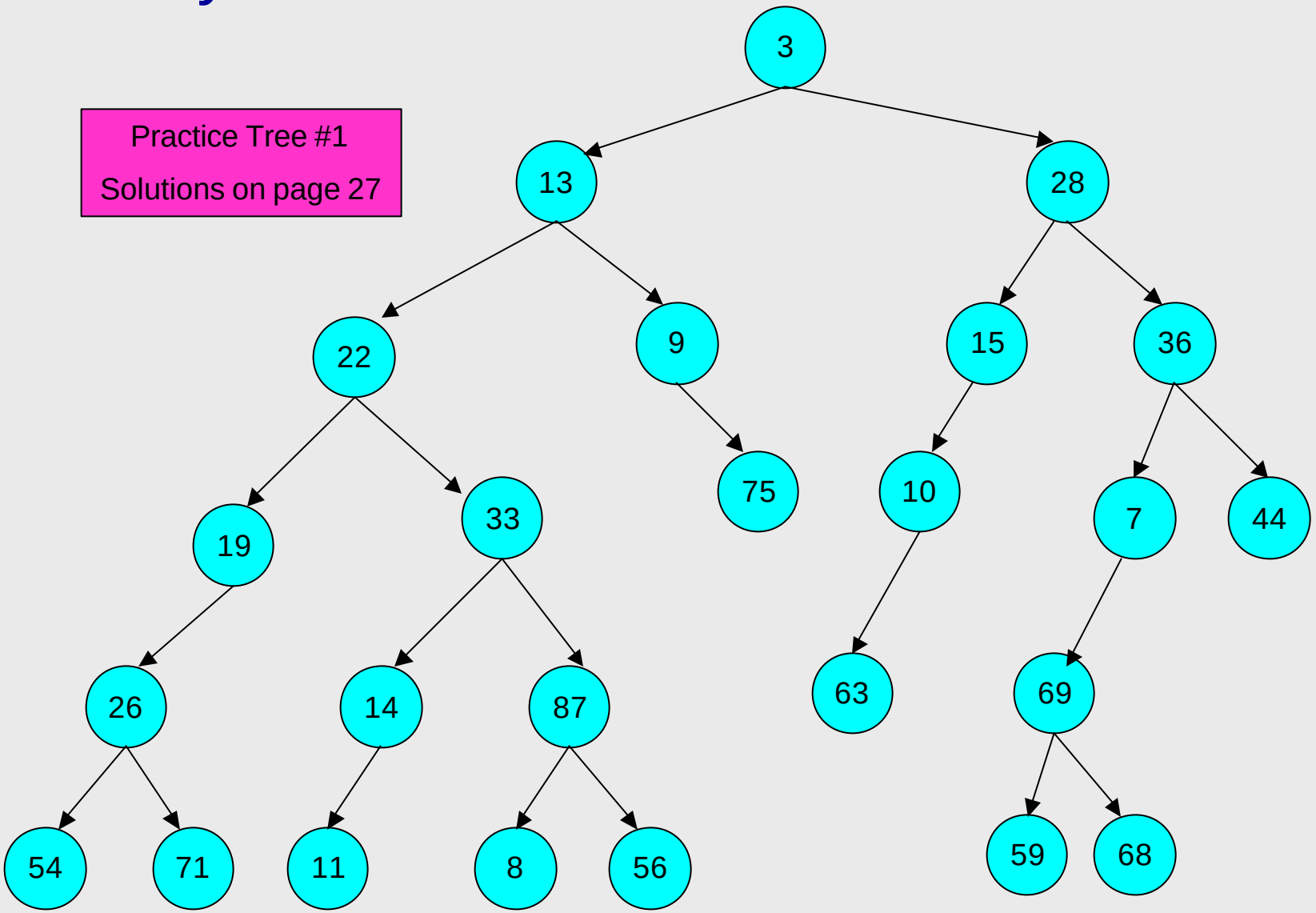
Postorder Traversal Algorithm

```
void postorder( struct treeNode *p)
{
        if (p != NULL)
{               postorder(p->left);
                postorder(p->right);
                printf("%d\n", p->data);  // this is the "visit"
}
}
```

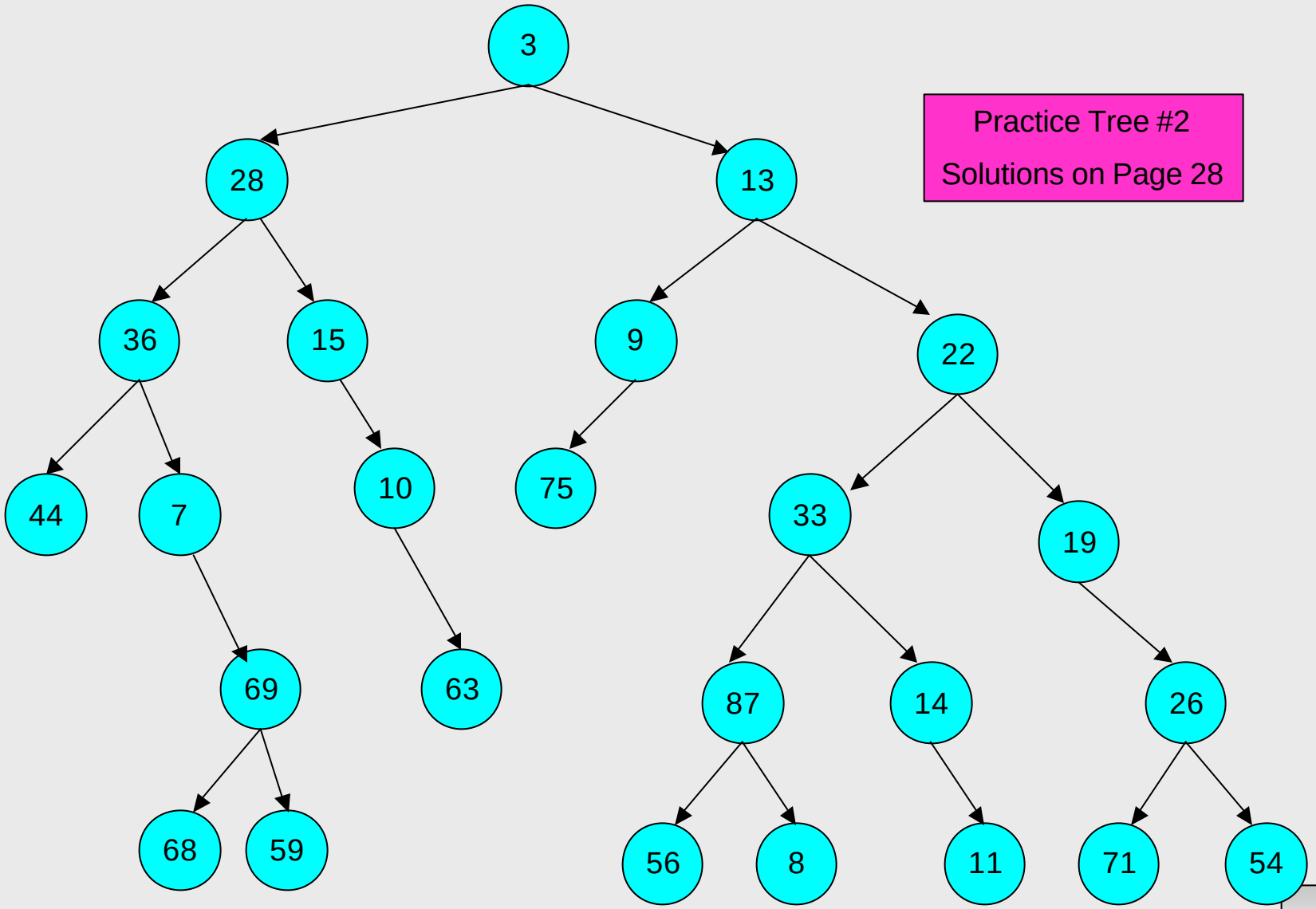

Binary Tree Traversals – Practice Problems

Practice Tree #1

Solutions on page 27



Binary Tree Traversals – Practice Problems



Practice Tree #2

Solutions on Page 28

Practice Problem Solutions – Tree #1

- Preorder Traversal:

  3, 13, 22, 19, 26, 54, 71, 33, 14, 11, 87, 8, 56, 9, 75, 28, 15, 10, 63, 36, 7, 69, 59, 68, 44

- Inorder Traversal:

  54, 26, 71, 19, 22, 11, 14, 33, 8, 87, 56, 13, 9, 75, 3, 63, 10, 15, 28, 59, 69, 68, 7, 36, 44

- Postorder Traversal:

  54, 71, 26, 19, 11, 14, 8, 56, 87, 33, 22, 75, 9, 13, 63, 10, 15, 59, 68, 69, 7, 44, 36, 28, 3

Practice Problem Solutions – Tree #2

- Preorder Traversal:

  3, 28, 36, 44, 7, 69, 68, 59, 15, 10, 63, 13, 9, 75, 22, 33, 87, 56, 8, 14, 11, 19, 26, 71, 54

- Inorder Traversal:

  44, 36, 7, 68, 69, 59, 28, 15, 10, 63, 3, 75, 9, 13, 56, 87, 8, 33, 14, 11, 22, 19, 71, 26, 54

- Postorder Traversal:

  44, 68, 59, 69, 7, 36, 63, 10, 15, 28, 75, 9, 56, 8, 87, 11, 14, 33, 71, 54, 26, 19, 22, 13, 3