C++ Plus Data Structures

Nell Dale

Chapter 9

Trees Plus

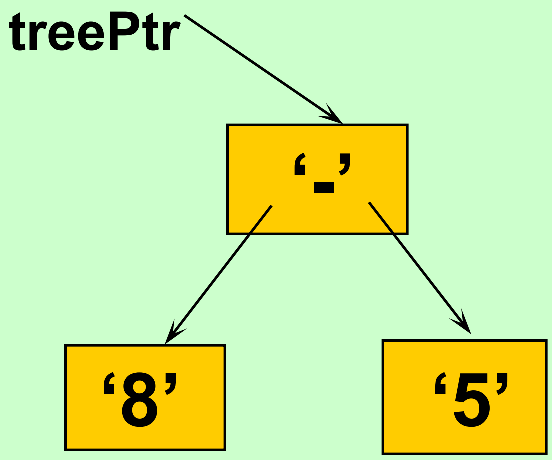
Modified from the slides by Sylvia Sorkin, Community College of Baltimore County - Essex Campus

A Binary Expression Tree is . . .

A special kind of binary tree in which:

1. Each leaf node contains a single operand,
2. Each nonleaf node contains a single binary operator, and
3. The left and right subtrees of an operator node represent subexpressions that must be evaluated before applying the operator at the root of the subtree.

A Two-Level Binary Expression



INORDER TRAVERSAL: `8 - 5 has value 3`

PREORDER TRAVERSAL: `- 8 5`

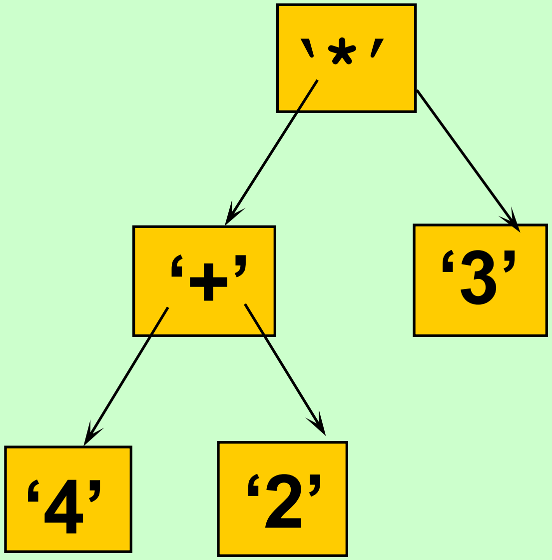
POSTORDER TRAVERSAL: `8 5 -`

Levels Indicate Precedence

When a binary expression tree is used to represent an expression, the levels of the nodes in the tree indicate their relative precedence of evaluation.

Operations at higher levels of the tree are evaluated later than those below them. The operation at the root is always the last operation performed.

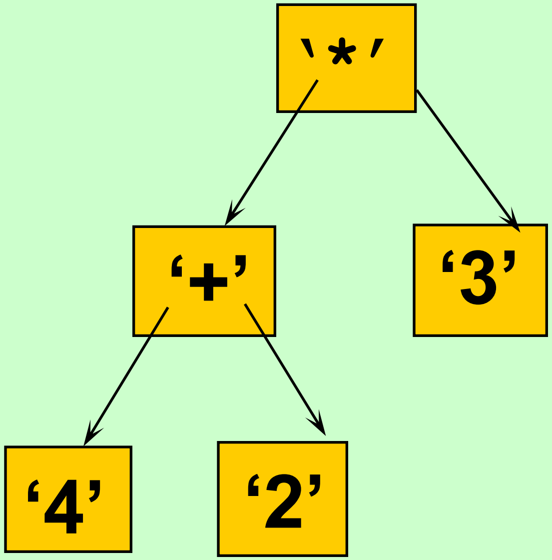
A Binary Expression Tree



What value does it have?

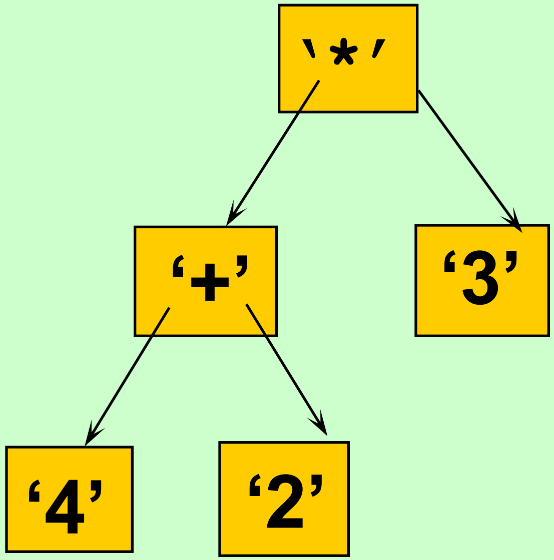
(4 + 2) * 3 = 18

A Binary Expression Tree



What infix, prefix, postfix expressions does it represent?

A Binary Expression Tree

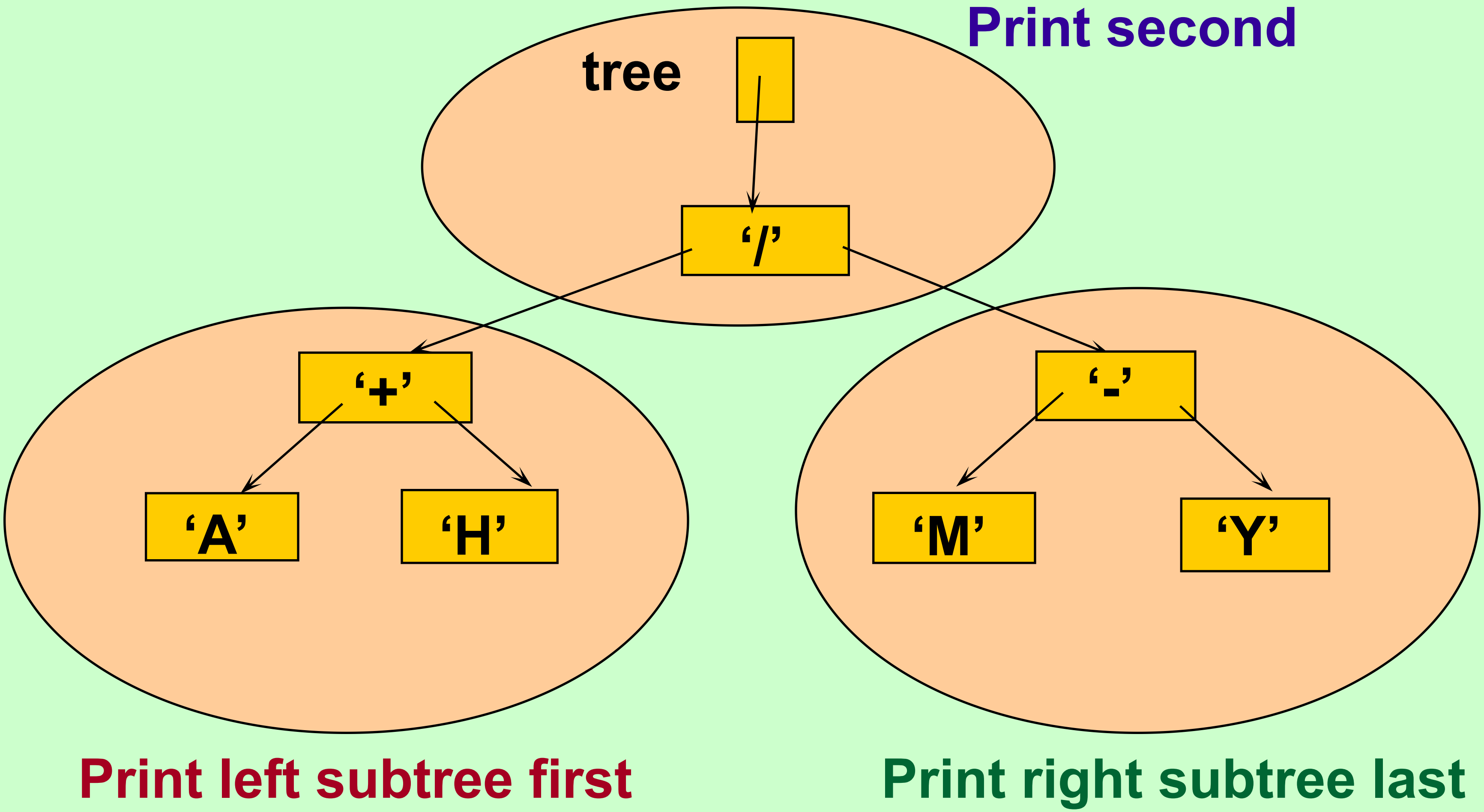


Infix: ((4 + 2) * 3)

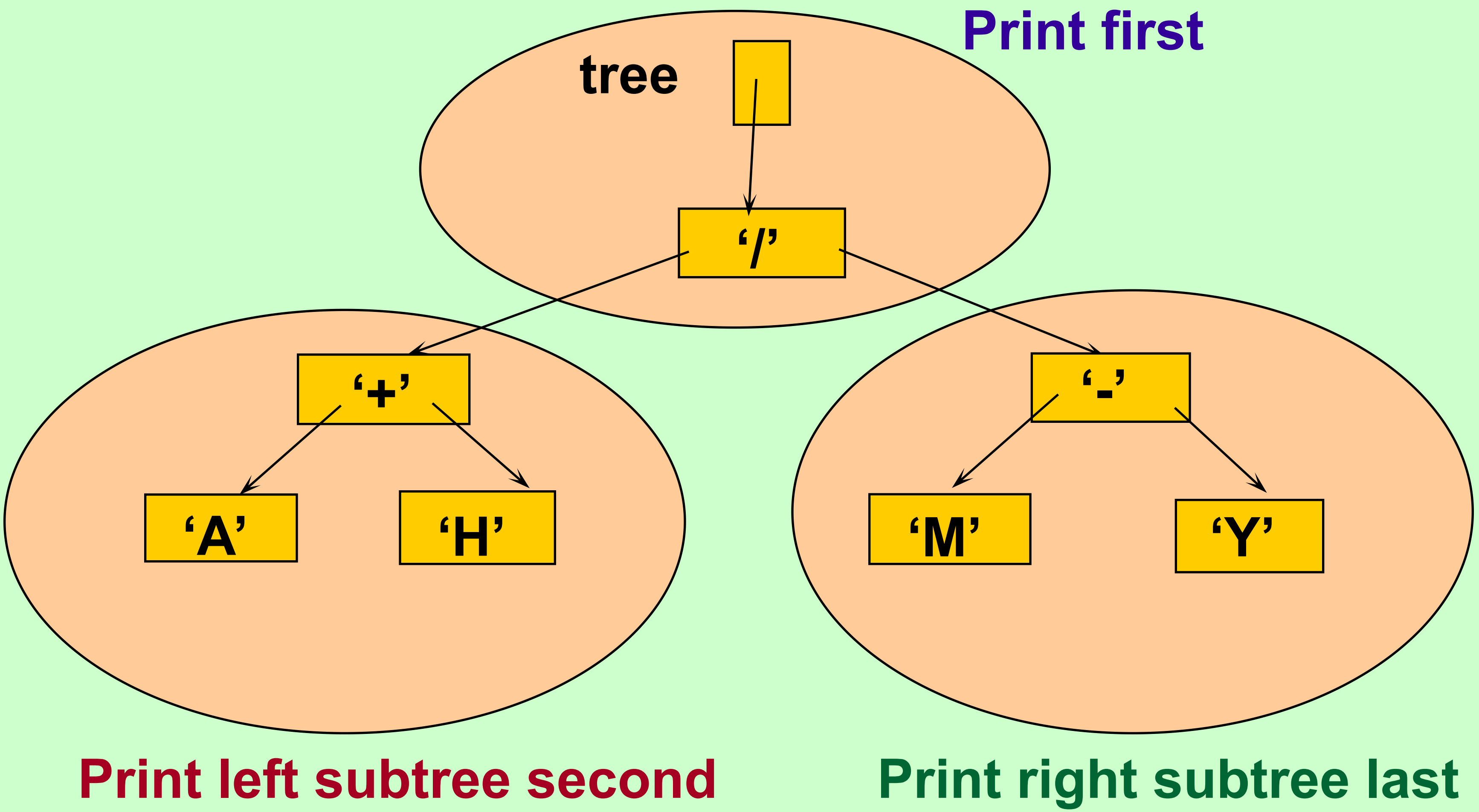
Prefix: * + 4 2 3

Postfix: 4 2 + 3 * ***has operators in order used***

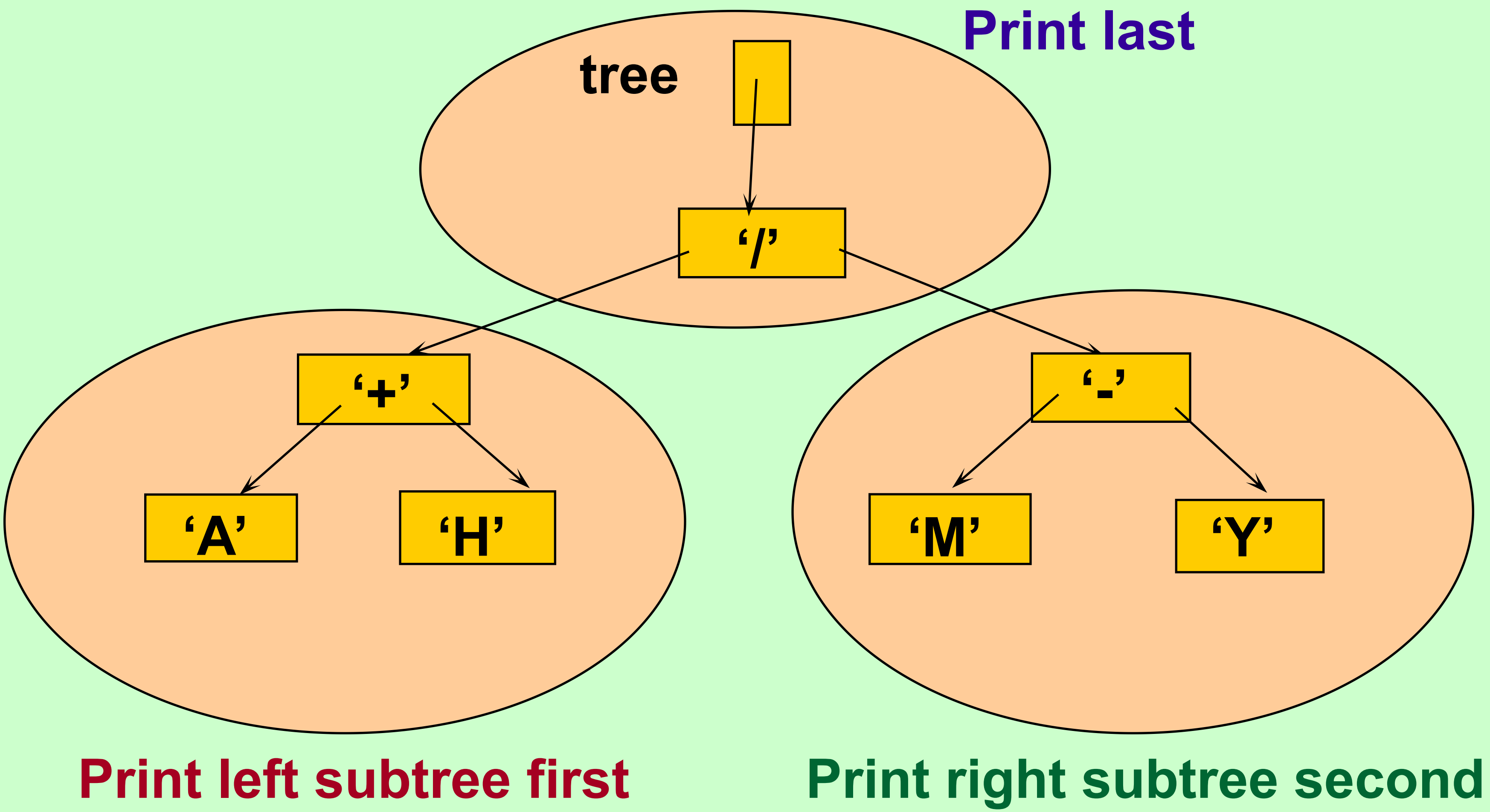
Inorder Traversal: (A + H) / (M - Y)



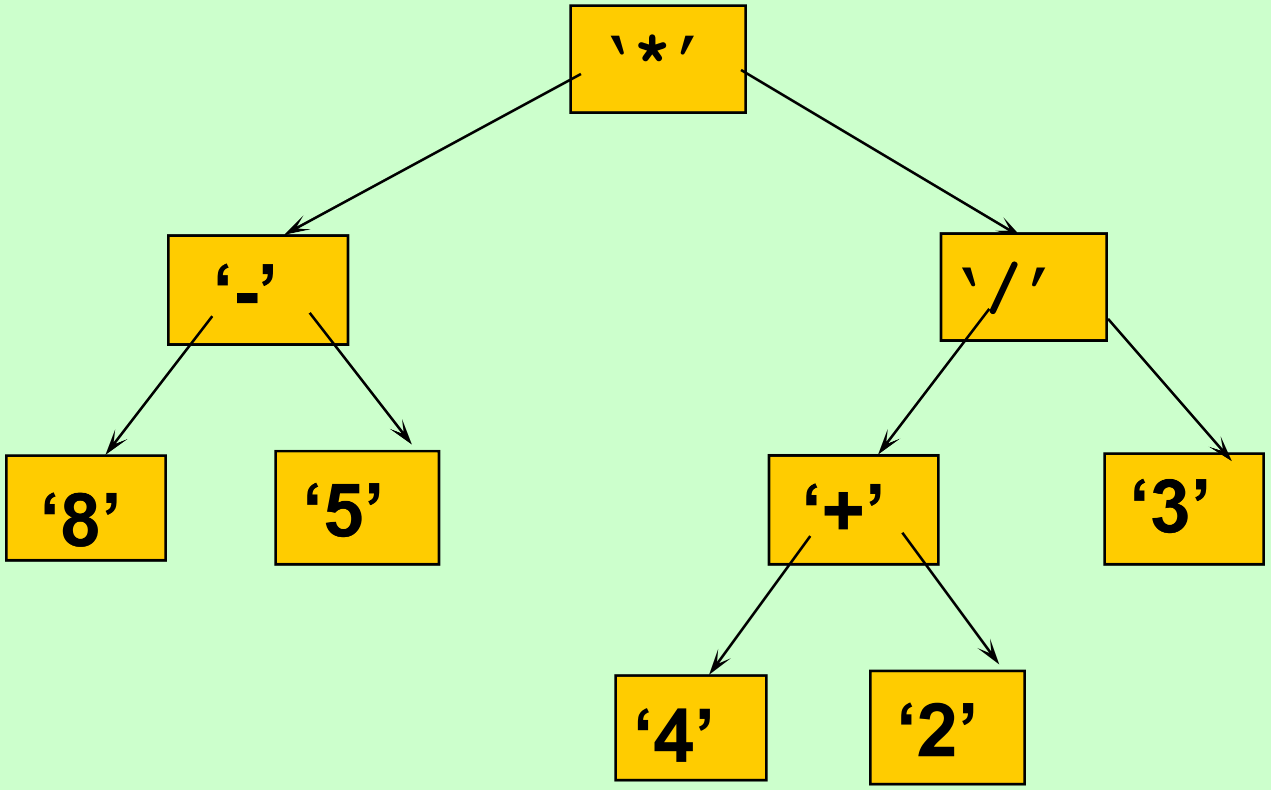
Preorder Traversal: / + A H - M Y



Postorder Traversal: A H + M Y - /

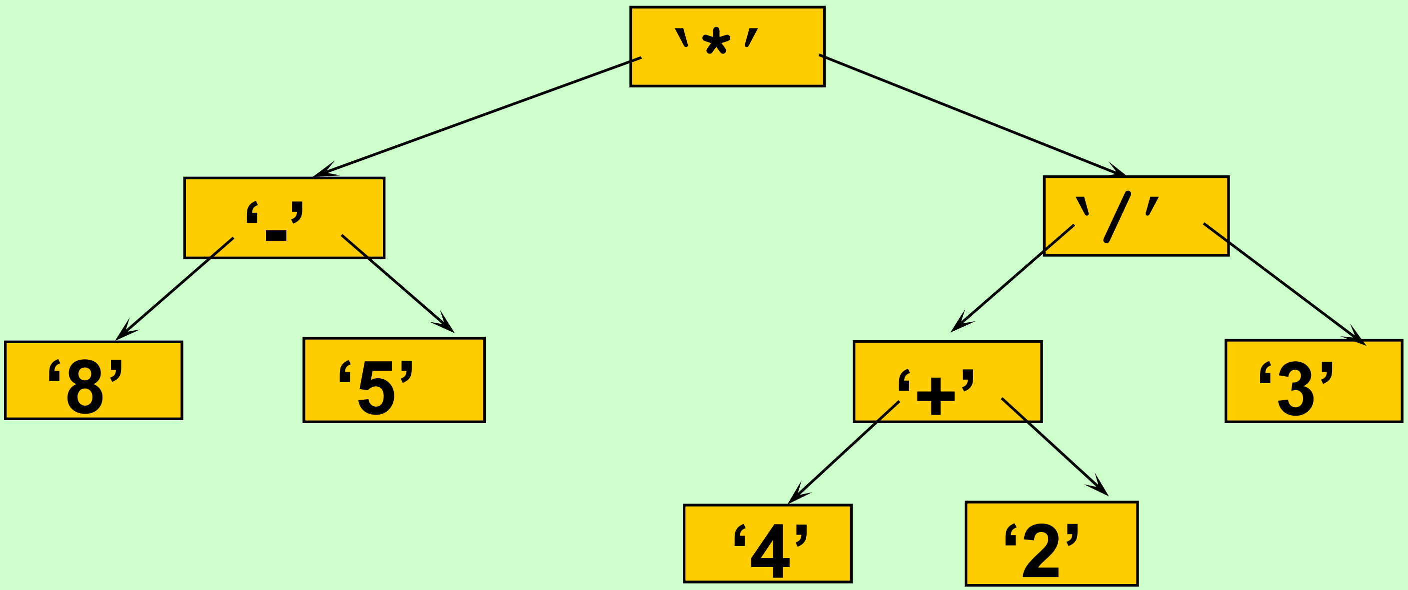


Evaluate this binary expression tree



What infix, prefix, postfix expressions does it represent?

A binary expression tree



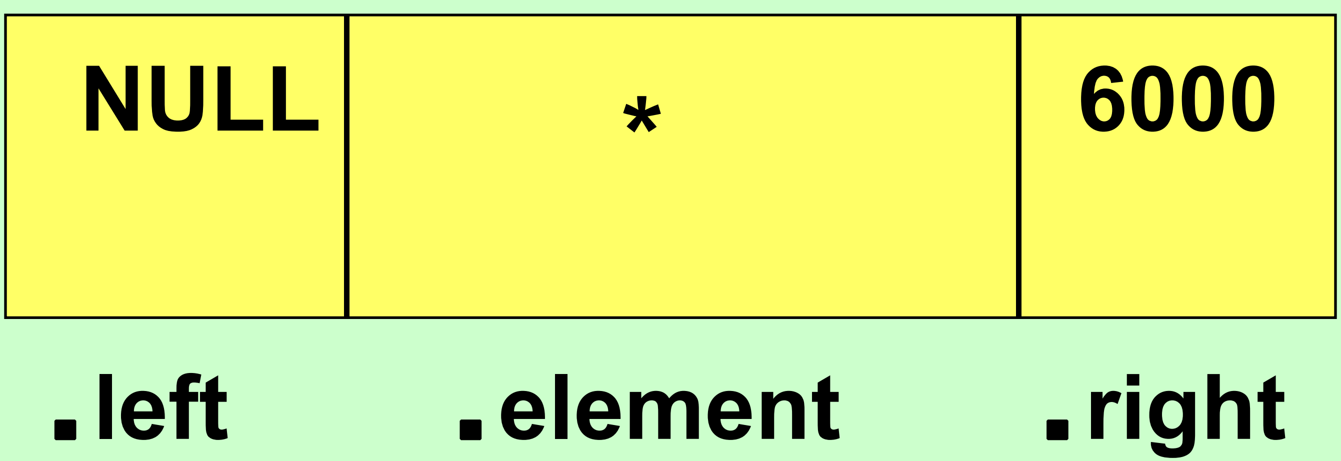
Infix: ((8 - 5) * ((4 + 2) / 3))

Prefix: * - 8 5 / + 4 2 3

Postfix: 8 5 - 4 2 + 3 / * ***has operators in order used***

ExprTreeNode (Lab 11)

```
class   ExprTreeNode  {
   private:
      ExprTreeNode (char elem,
                 ExprTreeNode *leftPtr, ExprTreeNode *rightPtr); // Constructor

      char                    element;   // Expression tree element
      ExprTreeNode  *left,            // Pointer to the left child
                             *right;       // Pointer to the right child
   friend class Exprtree;
};
```



InfoNode has 2 forms

```
enum  OpType  { OPERATOR,  OPERAND } ;

struct   InfoNode  {

   OpType       whichType;
   union                                    // ANONYMOUS union
   {
         char    operation ;
         int      operand ;
   }

};
```

OPERATOR	'+'

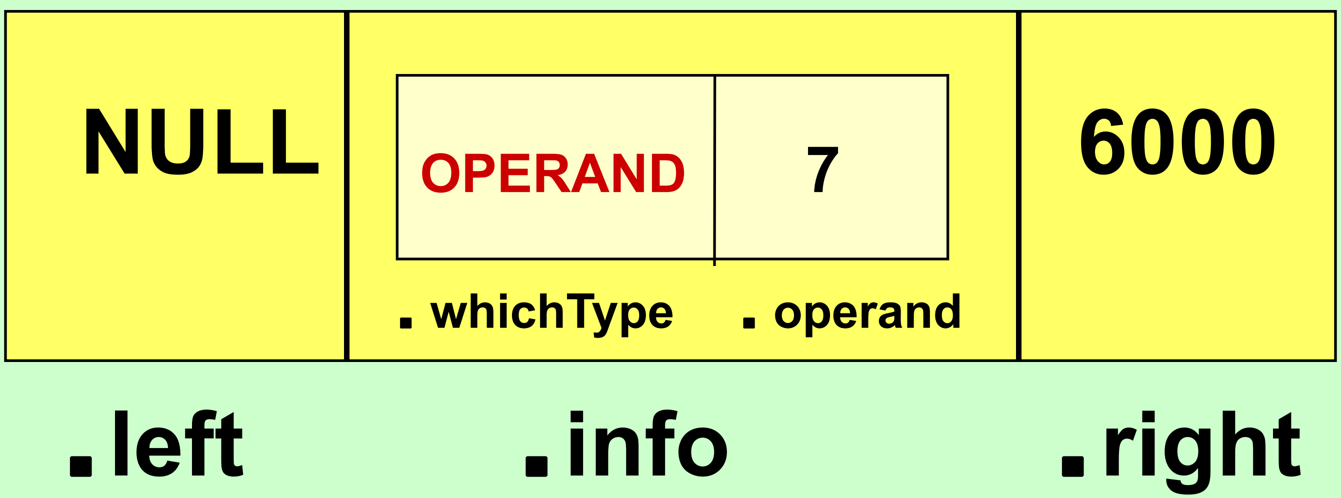
. whichType . operation

OPERAND	7

. whichType . operand

Each node contains two pointers

```
struct   TreeNode  {
    InfoNode      info ;          // Data member
    TreeNode*   left ;            // Pointer to left child
    TreeNode*   right ;           // Pointer to right child
};
```



Function Eval()

- **Definition**: Evaluates the expression represented by the binary tree.

- **Size**: The number of nodes in the tree.

- **Base Case**: If the content of the node is an operand,
  Func_value = the value of the operand.

- **General Case**: If the content of the node is an operator BinOperator,
  Func_value = Eval(left subtree)
  BinOperator
  Eval(right subtree)

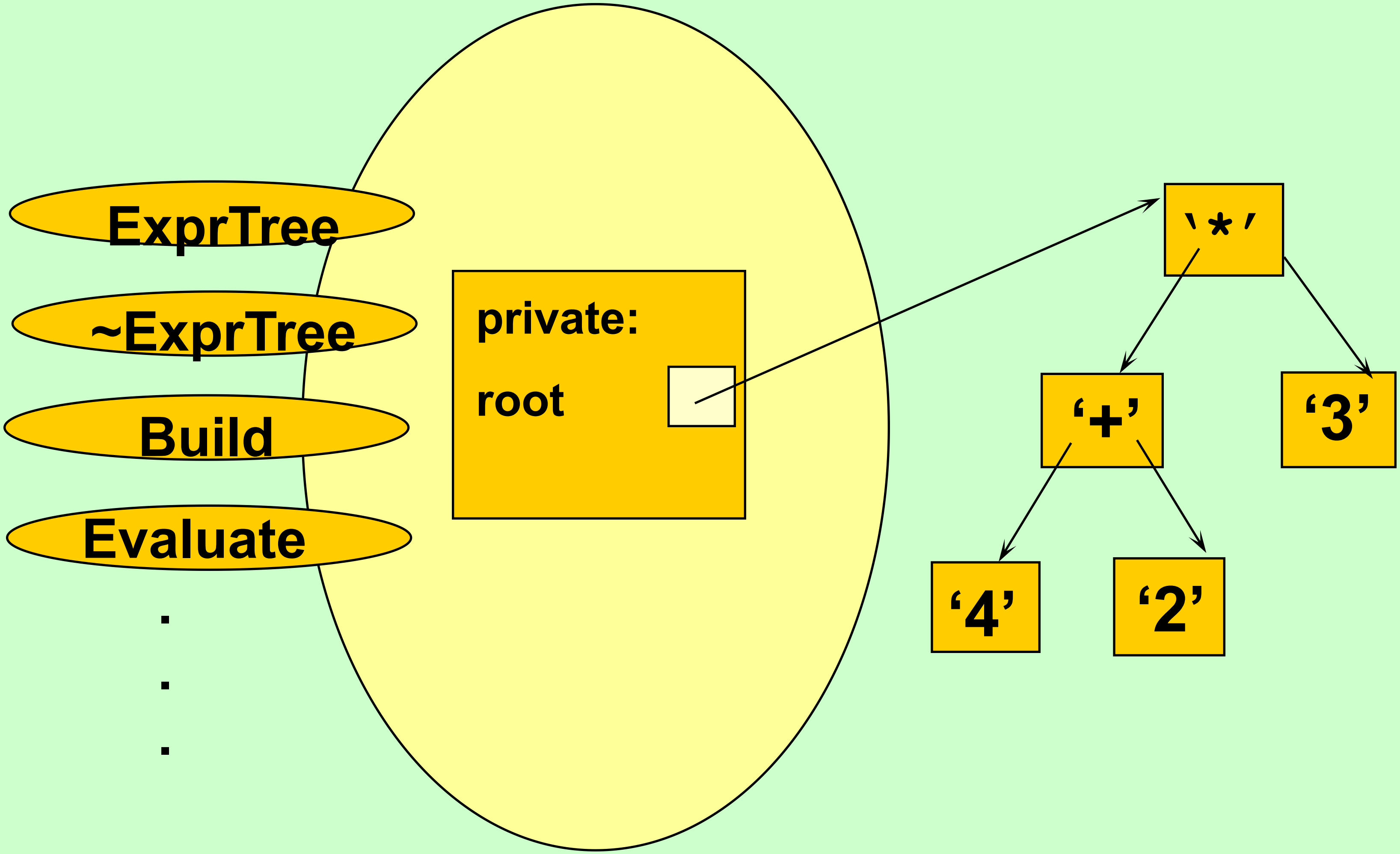
Eval(TreeNode * tree)

Algorithm:

```
IF Info(tree) is an operand
        Return Info(tree)
ELSE
  SWITCH(Info(tree))
    case + :Return Eval(Left(tree)) + Eval(Right(tree))
    case - : Return Eval(Left(tree)) - Eval(Right(tree))
    case * : Return Eval(Left(tree)) * Eval(Right(tree))
    case / : Return Eval(Left(tree)) / Eval(Right(tree))
```

```
int   Eval ( TreeNode*  ptr )

// Pre:     ptr is a pointer to a binary expression tree.
// Post:   Function value = the value of the expression represented
//            by the binary tree pointed to by ptr.

{     switch  ( ptr->info.whichType )
      {
          case OPERAND :  return  ptr->info.operand ;
          case OPERATOR :
             switch ( tree->info.operation )
             {
                case  '+'  :  return  ( Eval ( ptr->left ) + Eval ( ptr->right ) ) ;

                case  '-'  :  return  ( Eval ( ptr->left )  -  Eval ( ptr->right ) ) ;

                case  '*'  :  return  ( Eval ( ptr->left )  *  Eval ( ptr->right ) ) ;

                case  '/'  :  return  ( Eval ( ptr->left )  /  Eval ( ptr->right ) ) ;
             }
      }
}
```

class ExprTree

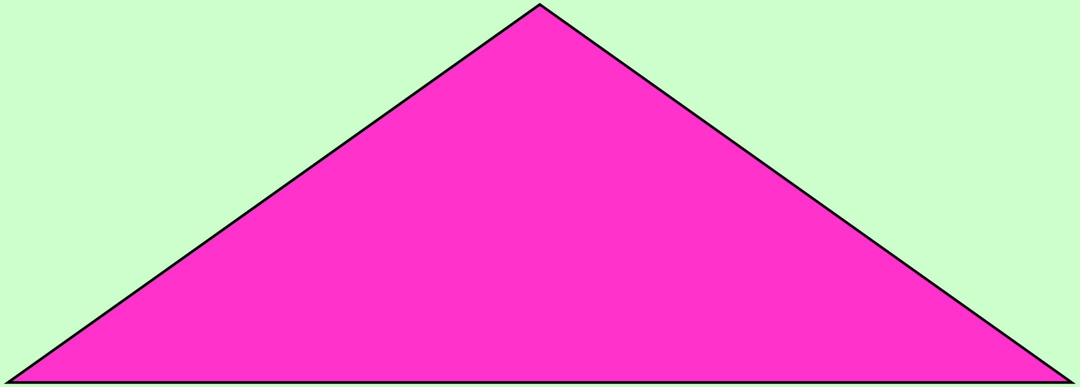


A Nonlinked Representation of Binary Trees

Store a binary tree in an array in such a way that the parent-child relationships are not lost

A full binary tree

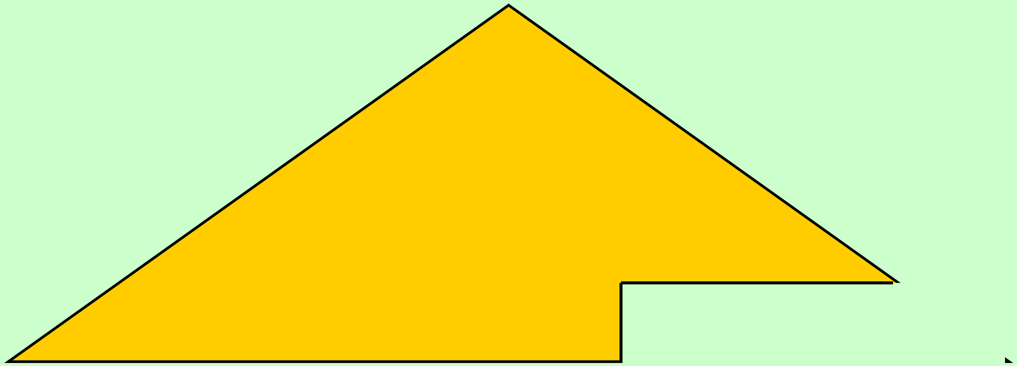
A **full binary tree** is a binary tree in which all the leaves are on the same level and every non leaf node has two children.



SHAPE OF A FULL BINARY TREE

A complete binary tree

A **complete binary tree** is a binary tree that is either full or full through the next-to-last level, with the leaves on the last level as far to the left as possible.



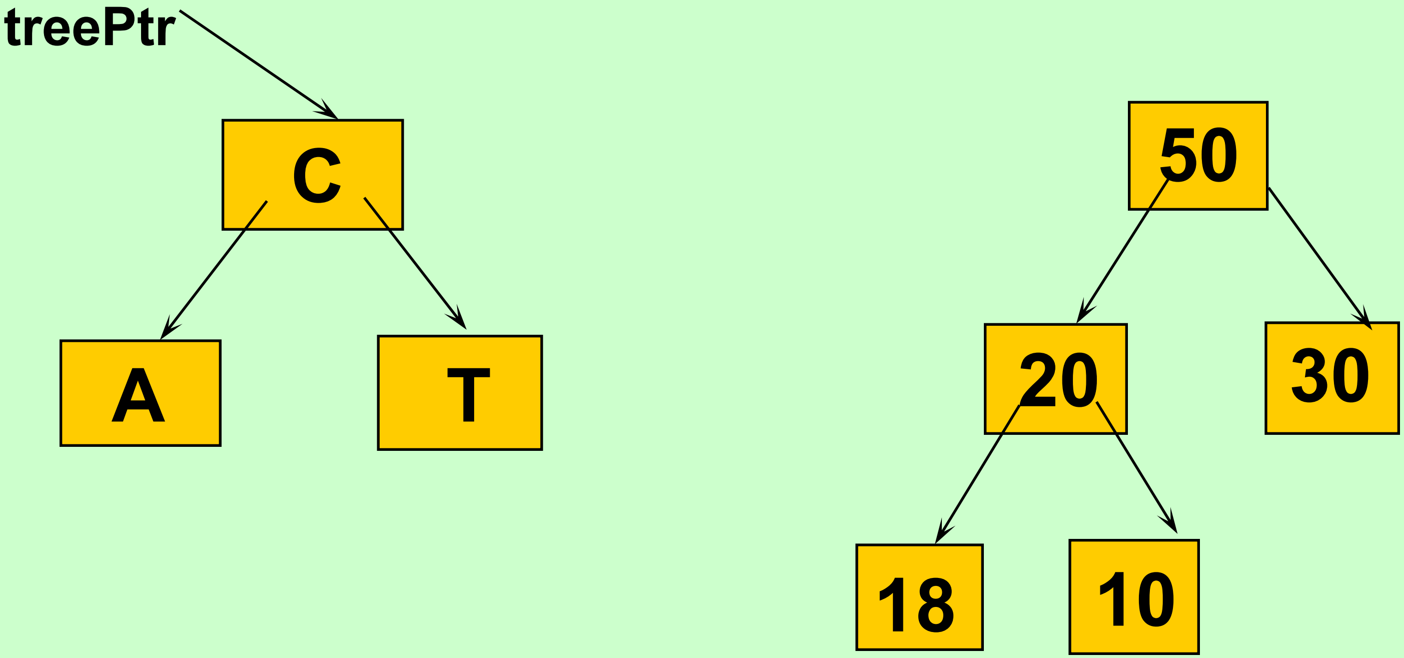
SHAPE OF A COMPLETE BINARY TREE

What is a Heap?

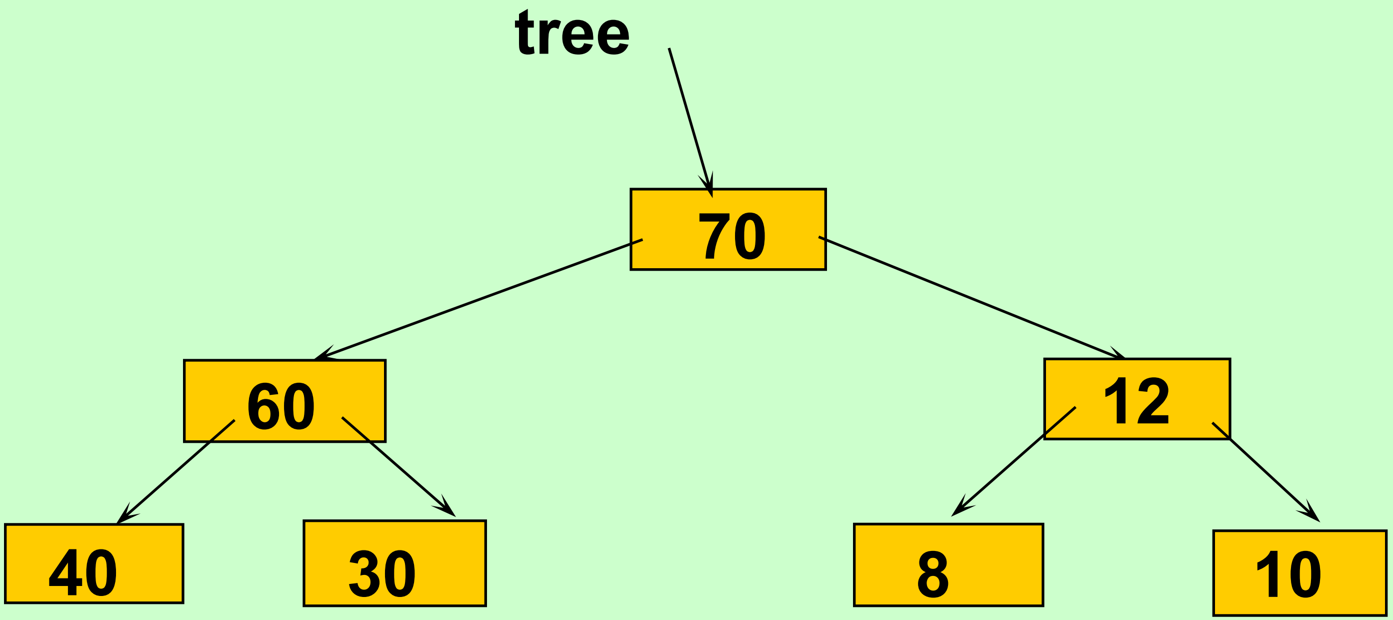
A heap is a binary tree that satisfies these special SHAPE and ORDER properties:

- **Its shape must be a complete binary tree.**
- **For each node in the heap, the value stored in that node is greater than or equal to the value in each of its children.**

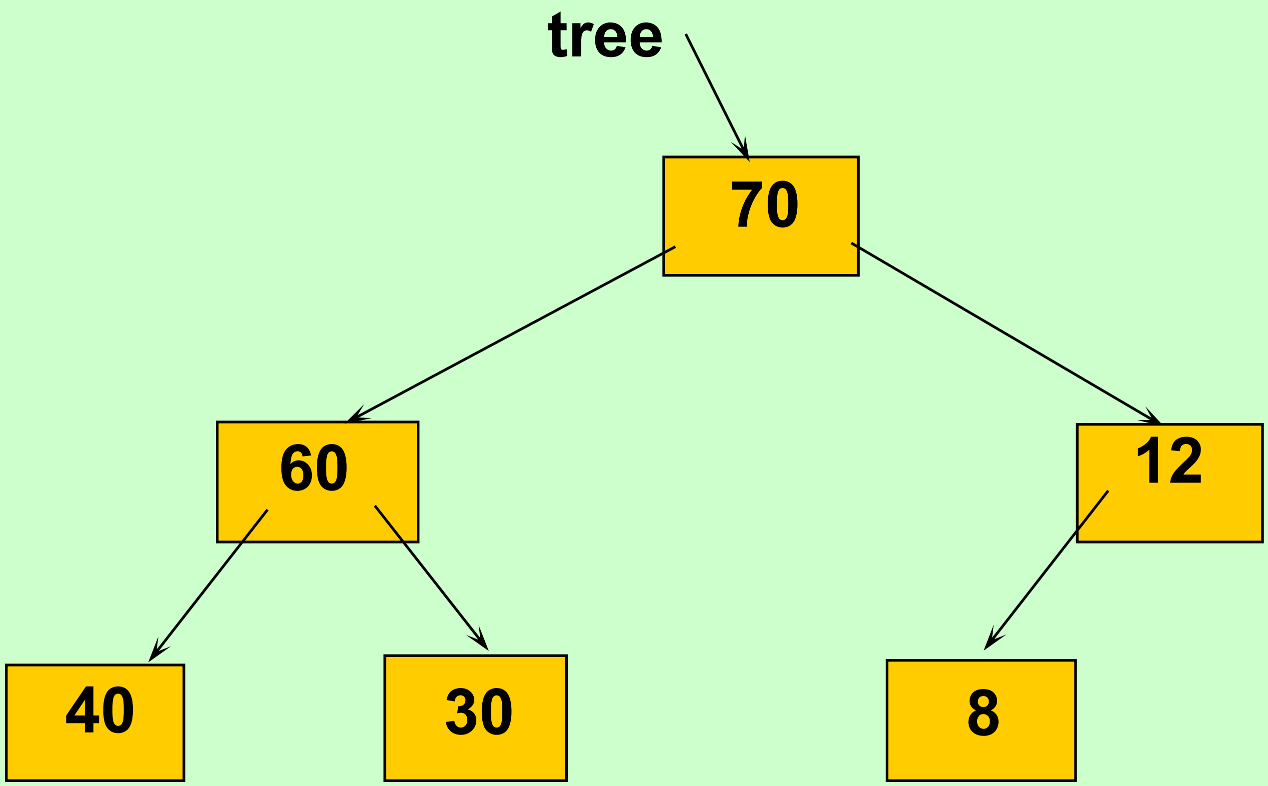
Are these both heaps?



Is this a heap?

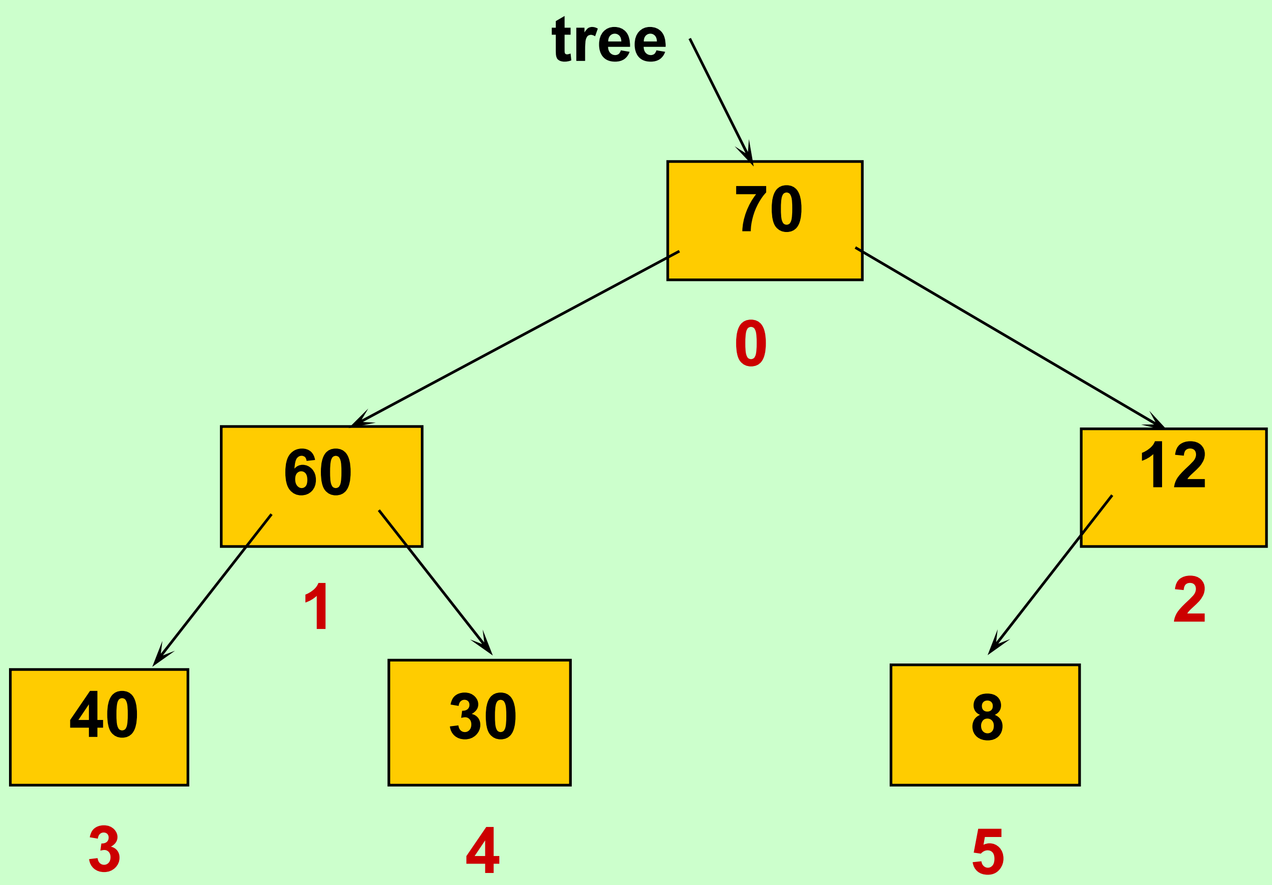


Where is the largest element in a heap always found?

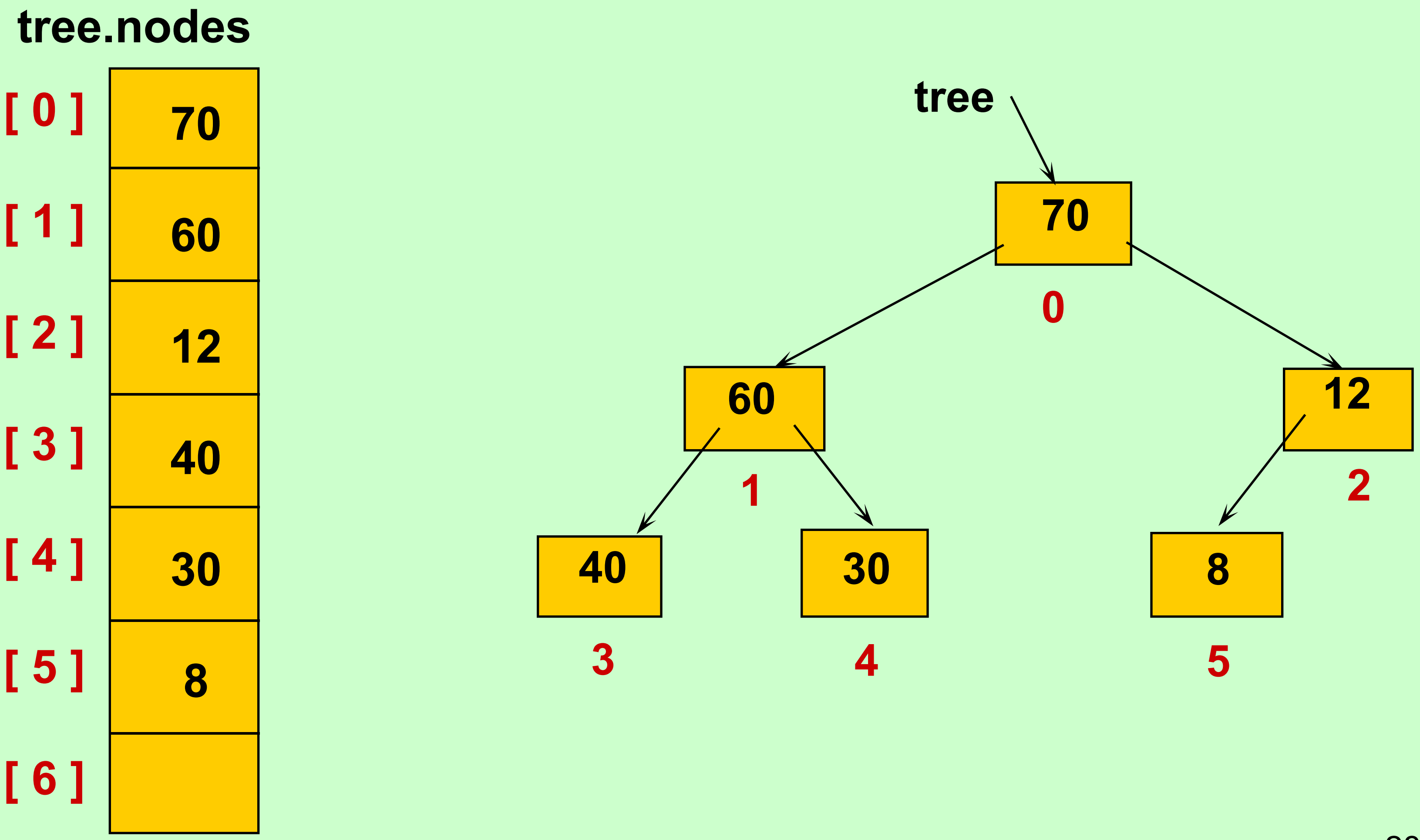


“maximum heap”

We can number the nodes left to right by level this way



And use the numbers as array indexes to store the tree



Parent-Child Relationship?

tree.nodes[index]:

left child: tree.nodes[index*2 + 1]

right child: tree.nodes[index*2 + 2]

parent: tree.nodes[(index-1) / 2]

Leaf nodes:

tree.nodes[numElements / 2]

…

tree.nodes[numElements - 1]

An application …

Fast access to the largest (or highest-priority) element in the structure:

- remove the element with the largest value from a heap …

```
//  HEAP SPECIFICATION

//  Assumes  ItemType  is either a built-in simple data type
//  or a class with overloaded realtional operators.

template< class  ItemType >
struct   HeapType
{
   void   ReheapDown ( int  root ,  int  bottom ) ;
   void   ReheapUp ( int  root,  int  bottom ) ;

   ItemType*  elements ;     // ARRAY to be allocated dynamically
   int  numElements ;

};
```

ReheapDown(root, bottom)

IF elements[root] is not a leaf

    Set maxChild to index of child with larger value

    IF elements[root] < elements[maxChild])

        Swap(elements[root], elements[maxChild])
        ReheapDown(maxChild, bottom)

ReheapDown()

```
// IMPLEMENTATION OF RECURSIVE HEAP MEMBER FUNCTIONS

template< class ItemType >
void   HeapType<ItemType>::ReheapDown ( int  root,  int  bottom )

// Pre:  root is the index of the node that may violate the heap
//         order property
// Post:  Heap order property is restored between root and bottom

{
     int  maxChild ;
     int  rightChild ;
     int  leftChild ;

     leftChild  =  root * 2 + 1 ;
     rightChild  =  root * 2 + 2 ;
```

```
    if  ( leftChild  <=  bottom )              // ReheapDown continued
    {
         if  ( leftChild  == bottom )
            maxChild  =  leftChld ;
         else
         {
              if  (elements [ leftChild ] <= elements [ rightChild ] )
                   maxChild  =  rightChild ;
              else
                   maxChild  =  leftChild ;
         }
         if  ( elements [ root ] < elements [ maxChild ] )
         {
             Swap ( elements [ root ] , elements [ maxChild ] ) ;
             ReheapDown ( maxChild, bottom ) ;
         }
     }
}
```

```
// IMPLEMENTATION              continued

template< class  ItemType >
void   HeapType<ItemType>::ReheapUp ( int  root,  int  bottom )

// Pre:  bottom is the index of the node that may violate the heap
//         order property.  The order property is satisfied from root to
//          next-to-last node.
// Post:  Heap order property is restored between root and bottom

{
     int  parent ;

     if  ( bottom  > root )
     {
          parent = ( bottom - 1 ) / 2;
          if ( elements [ parent ]  <  elements [ bottom ] )
          {
               Swap ( elements [ parent ], elements [ bottom ] ) ;
               ReheapUp ( root, parent ) ;
          }
     }
}
```

Priority Queue

A priority queue is an ADT with the property that only the highest-priority element can be accessed at any time.

Priority Queue ADT Specification

Structure:

The Priority Queue is arranged to support access to the highest priority item

Operations:

- MakeEmpty
- Boolean IsEmpty
- Boolean IsFull
- Enqueue(ItemType newItem)
- Dequeue(ItemType& item)

ADT Priority Queue Operations

Transformers

- MakeEmpty
- Enqueue
- Dequeue

change state

Observers

- IsEmpty
- IsFull

observe state

Dequeue(ItemType& item)

Function:

Removes element with highest priority and returns it in item.

Precondition:

Queue is not empty.

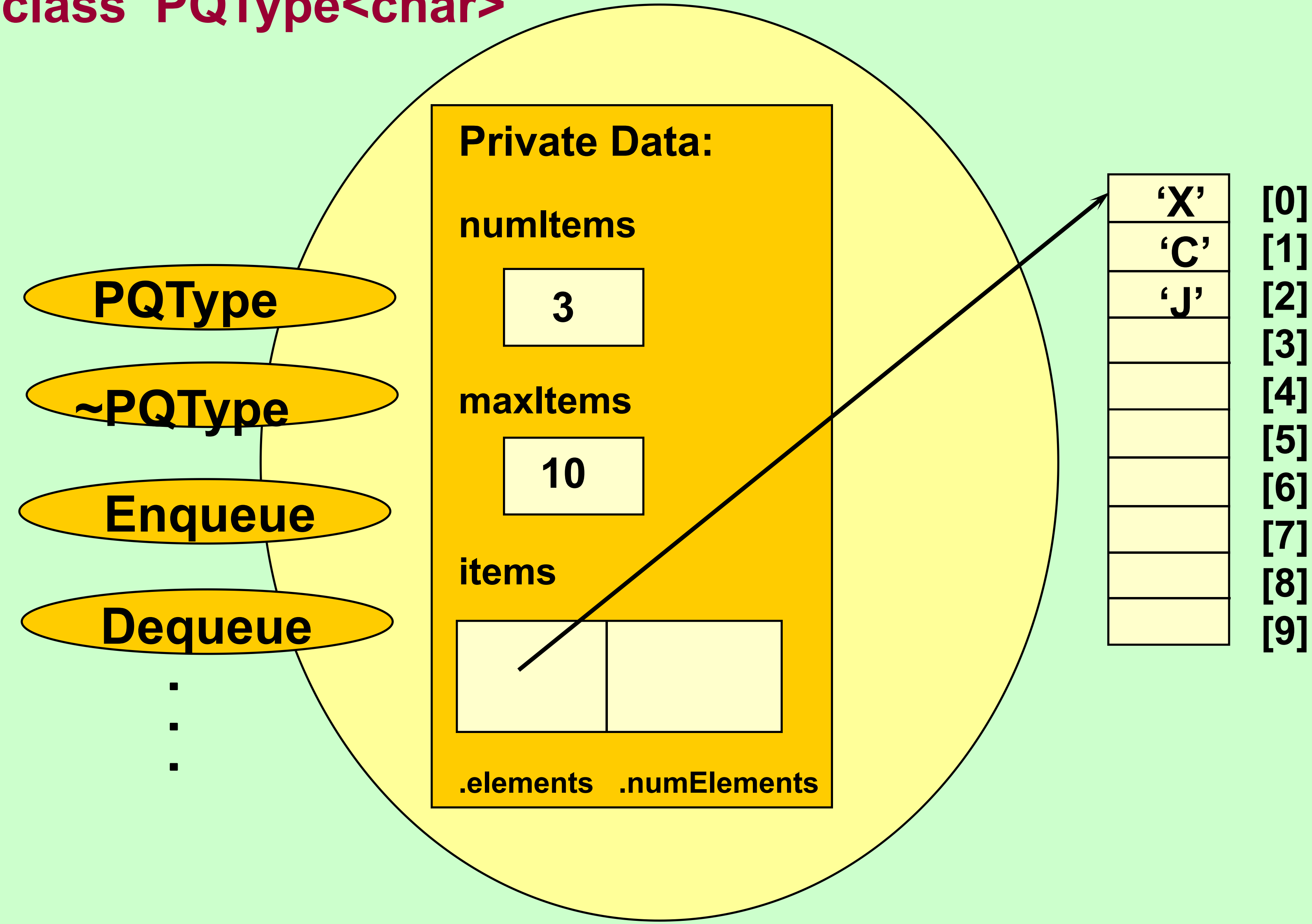
Postcondition:

Highest priority element has been removed from queue.

Item is a copy of removed element.

```
// CLASS PQTYPE DEFINITION AND MEMBER FUNCTIONS
//--------------------------------------------------------
#include "bool.h"
#include "ItemType.h"      // for ItemType

template<class ItemType>
class PQType  {
public:
   PQType( int );
   ~PQType ( );
   void MakeEmpty( );
   bool IsEmpty( ) const;
   bool IsFull( ) const;
   void Enqueue( ItemType item );
   void Dequeue( ItemType&  item );

private:
   int        numItems;
   HeapType<ItemType>  items;
   int        maxItems;
};
```

class PQType<char>



Implementation Level

Algorithm:

Dequeue(): $O(\log_2 N)$

- Set item to root element from queue
- Move last leaf element into root position
- Decrement numItems
- items.ReheapDown(0, numItems-1)

Enqueue(): $O(\log_2 N)$

- Increment numItems
- Put newItem in next available position
- items.ReheapUp(0, numItems-1)

Comparison of Priority Queue Implementations

	Enqueue	Dequeue
Heap	$O(\log_2 N)$	$O(\log_2 N)$
Linked List	$O(N)$	$O(1)$
Binary Search Tree		
Balanced	$O(\log_2 N)$	$O(\log_2 N)$
Skewed	$O(N)$	$O(N)$

Trade-offs: read Text page 548

End