Maggie Johnson
CS109

# Trees I

*Key topics:*

* Implementation of Trees: Binary Trees
* Leftmost-Child-Right-Sibling Representation of General Trees
* Trees and Recursion
* Heaps

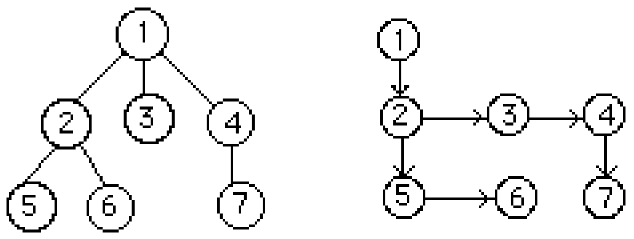
---

## Implementation of Trees: Binary Trees and Leftmost-Child-Right-Sibling Representation of General Trees

You learned in 106B or 106X how to implement a binary tree with left and right child pointers. A possible representation of a node for a tree given in C:

```
char *generic_ptr;
struct tnode {
        generic_ptr data;
        struct tnode *left;
        struct tnode *right;
};
```

There are seven basic operations in a tree ADT: FindElement, RetrieveElement, InsertElement, DeleteElement, TraverseTree, CreateTree & EmptyTree. The above node structures work fine for binary trees, but what about general trees, i.e., a tree which may have any number of children. There is a representation for general trees called **leftmost-child-right-sibling** where we put into each node a pointer to its leftmost child only. To find the second and subsequent children of a node n, we create a linked list of those children, with each child c pointing to the child n immediately to the right of c. That node is called the right sibling of c. For example, in the tree below:



3 is the right sibling of 2, 4 is the right sibling of 3 and 4 has no right sibling. We find the children of 1 by following its leftmost pointer to 2 then a right sibling pointer to 3, etc. At 4, we find a nil in the right sibling pointer indicating 1 has no more children; but 4 does have a leftmost child. The illustration to the right of the tree above shows the leftmost-child-rightmost-sibling representation of the tree. The down arrows are the leftmost child links, the sideways arrows are the right-sibling links.

A record structure for this representation is given below:

```
char *generic_ptr;
struct tnode {
        generic_ptr data;
        struct tnode *lmost;
        struct tnode *rsibling;
};
```

As an example of the use of this representation, consider the evaluation of an expression tree. For this application, we will use the following record structure. (Note: we could use a left-right child pointer representation because expression trees are always binary trees.)

```
typedef struct tnode {
        char op;
        int value;
        struct tnode *lmost;
        struct tnode *rsibling;
} tnode;

typedef struct tnode *tree;
```

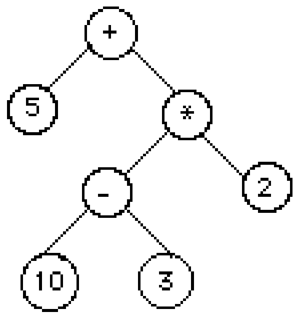
The field op will hold a char for the operator at an internal node or "i" to stand for integer, meaning the node is a leaf. This notation allows operators with any number of arguments although the code below assumes binary operators.

```
        int eval(tree pT)
        {
                struct tnode *secondchild;

                if (pT == NULL)
                        return 0;
                else if (pT->op == 'i')
                        return (pT->value);
                else {
                                pT->value = eval(pT->lmost);
                                secondchild = pT->lmost->rsibling;

                                switch (pT->op) {
                                        case '+': pT->value = pT->value + eval(secondchild);
                                                break;
                                        case '-': pT->value = pT->value - eval(secondchild);
                                                break;
                                        case '*': pT->value = pT->value * eval(secondchild);
                                                break;
                                        case '/': pT->value = pT->value / eval(secondchild);
                                                break;
                                }
                                return  pT->value;
                }
}
```

If node pT is a leaf, the first if condition is true and the value at the node is returned. If the node is not a leaf, we evaluate the expression associated with its leftmost child. Then, we find the root of the tree representing the right operand, which is the second child of pT. The rest of the code applies the operator at node pT to the left and right operands.  For example, the following tree and its representation would result in the sequence of calls as illustrated.



```
eval(+)
+       eval(5)
5               return 5
+       eval(*)
*               eval(-)
-                       eval(10)
10                              return 10
-                       eval(3)
3                               return 3
-                       return 7
*               eval(2)
2                       return 2
*               return 14
+       return 19
```

## Trees & Recursion

Because of the recursive nature of trees, it is frequently much easier to express tree algorithms using recursion rather than iteration.  This is a major advantage of trees because we can express some very complex operations cleanly and efficiently with recursion.  Consider an algorithm that traverses a tree printing the data element of the node.  According to the definition, a tree is either empty or has the structure:

```
          root
T(left)          T(right)
```

If the tree is empty, the traversal algorithm takes no action, i.e., the base case.  If the tree is not empty, then the traversal algorithm must perform three tasks:  it must print the data in the root node, and then traverse the two subtrees, each of  which is a binary search tree smaller than the original tree.

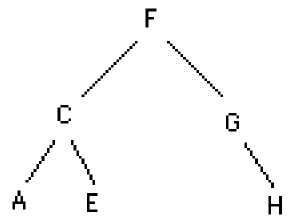
```
Traverse(T);
        if (T is not empty) {
                Traverse(T(Left))
                Traverse(T(Right)) }
```

The only  question is where do we print the root node information?  We can do this in three different places: before the traversals, in between them, or after them.  There are basically three standard ways of traversing a binary tree:

Preorder: 1) process the root R
2) traverse the left subtree of R in preorder
3) traverse the right subtree of R in preorder

Inorder: 1) traverse the left subtree of R in inorder
2) process the root
3) traverse the right subtree of R in inorder

PostOrder: 1) traverse the left subtree of R in postorder
2) traverse the right subtree of R in postorder
3) process the root

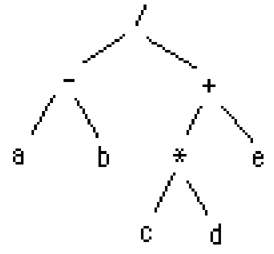


To do a traversal, we just make recursive calls to the algorithm. For example, the preorder traversal of the above tree processes F first (call #1), then proceeds down to the left subtree of F. Next, we process this subtree in preorder (call #2), so we process the root first which is C. Then, we proceed down to its left subtree and process that in preorder (call #3), so we process A. There is no left subtree or right subtree of A so call #3 is completed. We return to call #2 where we have completed the left subtree traversal. The right subtree traversal in preorder processes E (new call #3) . Since this has no left or right subtree, new call #3 is completed. Call #2 is also completed, so we return up to call #1 where we have finished the left subtree traversal. Now, we have to traverse the right subtree in preorder (new call #2). We process the root of the right subtree, G in preorder. G has no left subtree, but it does have a right subtree, so we process that subtree in preorder (new call #3). We process H and have completed new call #3 because H has no subtrees. We have also completed new call #2, and call #1. The preorder traversal: FCAEGH. In exactly the same manner, the inorder traversal can be evaluated: ACEFGH (notice this is in sorted order); and the postorder: AECHGF.

*If I give you the postorder traversal of a binary tree, can you draw it?* ____________*Will the tree be unique?*____________ *If not, what more is needed?* ___________________
*Can traversals be done on general trees, or only on binary trees?* ____________________________

One of the interesting features of traversals is how they work on algebraic expressions. This is one of the reasons why they are so useful in compilers. If you take any tree representation of an algebraic expression where operators are the internal nodes and the operands are the leaves:

E = (a - b) / ((c * d) + e)



A preorder traversal will give you the prefix representation: /-ab+*cde;   an inorder traversal gives infix: a-b/c*d+e; and the postorder traversal gives postfix:  ab-cd*e+/.

*Can unary operators be represented in an expression tree?* ________________________________

## Other Recursive Procedures on Trees

**1)** A search algorithm  to find a particular value in a binary search tree is based on the following recursive definition of a binary search tree:

For each node n, a binary search tree satisfies the following three properties:
- n's search key is greater than all search keys in its left subtree
- n's search key is less than all search keys in its right  subtree
- both of n's left and right subtrees are binary search trees

We assume the following definition for a binary tree; to keep it simple we will assume that we are just storing strings in the tree:

```
typedef struct tnode {
        char *word;
        struct tnode *left;
        struct tnode *right;
}tnode;
typedef struct tnode *tree;
```

----------------------------------------------------------------------------------------------------------

```
void BSTFind(tree pT, char *X)
{
        if (pT == NULL)
                printf("%s is not here", X);            /* empty tree */
        else if (strcmp(pT->word,X)  == 0)
                printf("%s is here", X);                /* name is found */
        else if ((strcmp(X, pT->word)) < 0)
                BSTFind(pT->left, X);                   /* search L subtree */
        else
                BSTFind(pT->right, X);                  /* search R subtree */
}
```

*What is the complexity of the above function?* ________________________

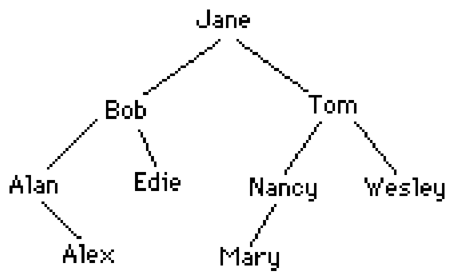
**2)** An iterative algorithm for inserting into a binary search tree calls a FindElement function to find the appropriate spot for inserting the element, and then just modifies the pointers.  A recursive insert function is given below.

```
struct tnode *Insert(tree pT, char *w)
{
        if (pT == NULL) {                       /* empty tree */
                pT = talloc();                  /* allocates for a new node */
                pT->word = w;
                pT->left = pT->right = NULL;
        } else if ((strcmp(w, pT->word)) < 0)   /* less than so into left subtree */
                pT->left = Insert(pT->left, w);
        else                                    /* more than so into right subtree */
                pT->right = Insert(pT->right, w);
        return p;
}
```

**3)** A simple recursive function that returns the number of nodes in a tree:

```
int NodeCount(tree pT)
{
        if (pT == NULL)
                return 0;
        else
                return (1 + NodeCount(pT->left) + NodeCount(pT->right));
}
```

Trace this through on the following tree:



The first call sends in the root of the tree:

| *Level Number* | *Description* | *Params & Local Vars on entering function* |
|---|---|---|
| 1 | NodeCount(Jane) | NodeCount = 1 + NodeCount(Bob) + NodeCount(Tom) |
| 2 | NodeCount(Bob) | NodeCount = 1 + NodeCount(Alan)+NodeCount(Edie) |
| 3 | NodeCount(Alan) | NodeCount = 1 + NodeCount(null)+NodeCount(Alex) |
| 4 | NodeCount(null) | return 0 |
| 3 | Return | NodeCount(null) = 0 |
| | NodeCount(Alex) | NodeCount = 1+ NodeCount(null) |
| 4 | NodeCount(null) | return 0 |
| 3 | Return | NodeCount(Alex) = 1 |
| 2 | Return | NodeCount(Alan) = 2 |
| | NodeCount(Edie) | NodeCount=1+NodeCount(null)+NodeCount(null) |
| 3 | NodeCount(null) | NodeCount = 0 { twice } |
| 2 | Return | NodeCount(Edie) = 1<br>NodeCount(Bob) = 2 + 1 = 3 |

etc.

**4)** A not so-simple recursive function that returns if a binary tree is actually a binary search tree.

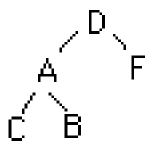
```
Boolean IsBST(tree pT)
{
        Boolean LeftIsBST, RightIsBST;

        if  (pT == NULL)
                return TRUE;
        else {
                if (pT->left == NULL)
                        LeftIsBST = TRUE;
                else
                        if ((strcmp(pT->left->word, pT->word)) > 0)
                                LeftIsBST = FALSE;
                        else
                                LeftIsBST = IsBST(pT->left);

                if (pT->right == NULL)
                        RightIsBST = TRUE;
                else
                        if ((strcmp(pT->right->word, pT->word)) < 0)
                                RightIsBST = FALSE;
                        else
                                RightIsBST = IsBST(pT->right);

                return ((LeftIsBST) && (RightIsBST));
        }
}
```

Trace this function on the following tree:



| *Level Number* | *Description* | *Params & Local Vars on entering function* |
|---|---|---|
| 1 | IsBST(D) | LeftIsBST = IsBST(A) |
| 2 | IsBST(A) | LeftIsBST = false<br>RightIsBST = IsBST(B) |
| 3 | IsBST(B) | pT->left = NULL & pT->right = NULL  so<br>LeftIsBST = true and RightIsBST = true<br>return true and true = true |
| 2 | return | LeftIsBST = false;  RightIsBST  = true<br>return true and false = false |
| 1 | return | LeftIsBst = false<br>RightIsBST = IsBST(F) |
| 2 | IsBST(F) | pT->left = NULL & pT->right = NULL  so<br>LeftIsBST = true and RightIsBST = true |

| | | |
|---|---|---|
| | | return true and true = true |
| 1 | return | LeftIsBST = false;  RightIsBST  = true<br>return true and false = false |
| | | return false |

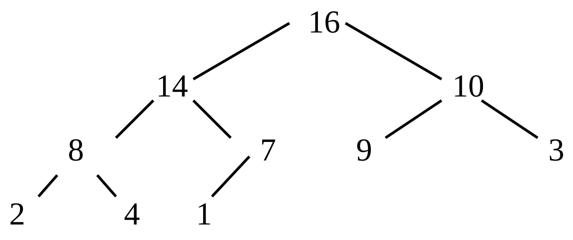
*Does the above function work on all possible binary trees?* ________________________________

## Heaps

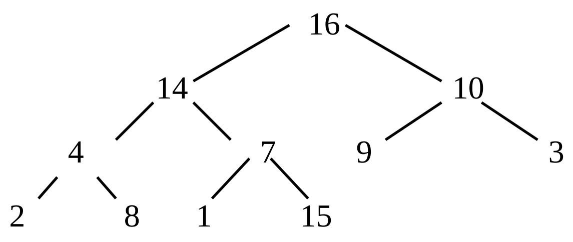
A **heap** is an array-based implementation of a special type of **complete binary tree**, i.e., a tree which is completely filled on all levels except possibly the lowest, which is filled from the left up to a point.  The most important characteristic of a heap is every node is greater than or equal to the values stored in the data field of its children.  Thus, the largest element of a heap is stored at the root, and the subtrees rooted at a node contain smaller values than does the node itself.

An important consideration in using a heap is maintaining the "heap property" while keeping the tree complete.  The only way to do this is to control carefully the inserts.  To insert an element into a heap, we do a "bubble-up" procedure.  We insert the element in the next available position at the lowest level of the heap and let it bubble-up to its appropriate place.

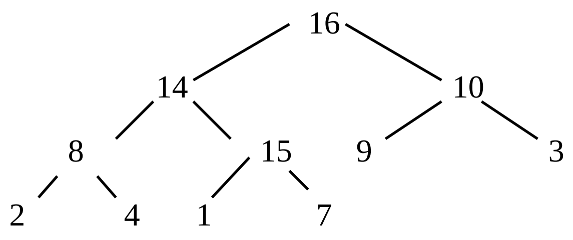
*Example*



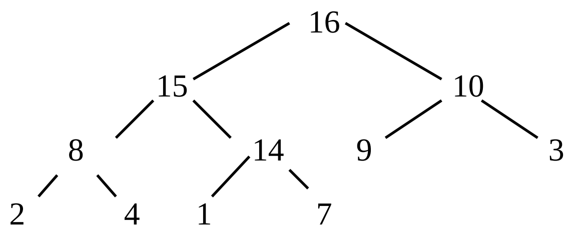
We will now insert a 15 into this tree using the bubble-up procedure:



15 is greater than its parent (7) so switch them:



15 is greater than its parent (14) so switch them:



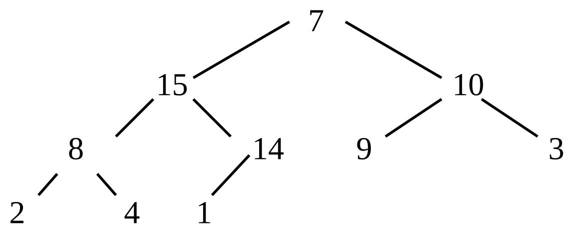
And now we have a heap.

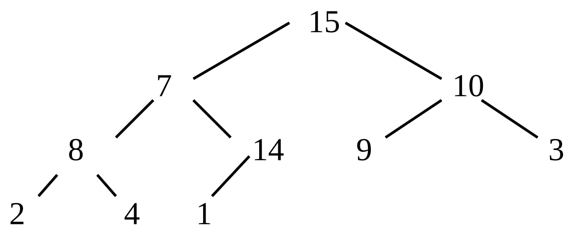
Because of the heap property, we can get a sorted list of the elements in the heap array, by deleting the root, and then re-ordering the heap using a bubble-down procedure. We have to be careful how we do this or we will not end up with a complete tree.

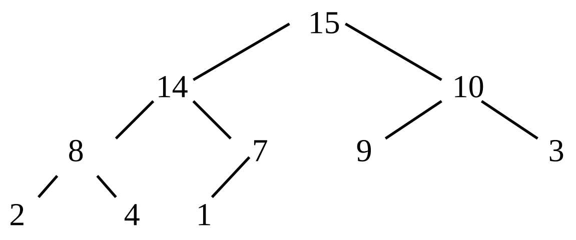
What element should replace the deleted root in the tree to assure that we end up with a complete tree?

Once the chosen element is in the root’s position, we bubble it down to its appropriate place by switching it with the greater of its two children. We continue to do this, until the element placed at the root is not greater than either of its children.

*Example*







The bubble-up and bubble-down procedures can be implemented recursively since exactly the same process is being performed at each subtree. The running time of each of these procedures is O($\log_2 n$) since each recursive call moves us up or down one node in the tree, and a complete binary tree has approximate height $\log_2 n$ (stay tuned - we will see why in a future lecture). The actual work done in the procedures is O(1) since there are no loops; just a couple of comparisons.

Bubble-up and bubble-down can be used to implement a very clever little sorting algorithm called heapsort. Here is how it works:

```
for (j = 1; j <= n; j++)
        insert_element_into_heap_and_bubble_up;
for (j = 1; j <= n; j++)
        delete_root_and_bubble_down;
```

So we insert all the elements into an empty heap, and then we delete the root n times, to get a sorted list. This algorithm is comparable in running time to the best sort algorithms we have (quicksort and mergesort).

```
What is the running time of heapsort?
```

## Bibliography

The evaluate expression algorithm using the leftmost-child-right-sibling representation comes from Aho & Ullman. This is also an excellent source for basic information on trees, the implementation of binary trees, and recursions on trees. More advanced sources for tree implementations and recursions are Aho, Hopcroft, Ullman; Knuth; and Corman. A good source book for data structure implementations in C is Esakov, Weiss.

A. Aho, J. Hopcroft, J.D. Ullman, *Data Structures and Algorithms*, Reading, MA: Addison-Wesley, 1983.

A. Aho, J.D. Ullman, *Foundations of Computer Science*, New York: W.H. Freeman, 1992.

T. Cormen, C. Leiserson, R. Rivest, *Introduction to Algorithms*, New York: McGraw-Hill, 1991.

J. Esakov, T. Weiss, *Data Structures: An Advanced Approach Using C,* Englewood Cliffs, NJ: Prentice-Hall, 1989.

D. Knuth, *Fundamental Algorithms, volume 1 of The Art of Programming, 2nd ed.*, Menlo Park, CA: Addison-Wesley, 1973.

J.W. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM* 7:6, 1964, p. 347.

## Historical Notes

According to Knuth, "trees have been in existence since the third day of creation, and perhaps earlier". Trees have had many common usages, most notably family trees. The concept of a tree as a formal mathematical entity seems to have appeared first in the work of G. Kirchoff in 1847, who used trees to find a set of fundamental cycles in an electrical network. The name "tree" and many results dealing with them began to appear ten years later in the work of the English mathematician Arthur Cayley, who used trees to represent his enumeration of various chemical compounds, and in his study of the structure of algebraic formulas.

Tree structures in computer science date back to the A-1 compiler language, developed by Grace Hopper in 1951. She used them to represent algebraic formulas using a three-address code that corresponds to a data-rchild-lchild usage. Since then, tree structures have been studied independently by many people, but the basic techniques for tree manipulations seldom appeared in print except as a part of a more detailed description of an algorithm. The first general survey (of all data structures) was by K. Iverson and L. Johnson of IBM in 1962.

The whole idea of a heap and the heapsort algorithm was introduced by J.W.Williams in 1964.