Trees

What are trees?

- Nonlinear two-dimensional data structures
- The data is organized so that items of information are related by the branches
- Based on nodes, where each node contains two or more links
- The links are known as *children* of the node
- A tree is a finite set of one or more nodes such that

  1. There is a specially designed node called the *root*
  2. The remaining nodes are partitioned into $n \geq 0$ disjoint sets $T_1, T_2, \ldots, T_n$, where each of these sets is a tree. $T_1, \ldots, T_n$ are known as the subtrees of the root.

- A *node* stands for the item of information and the branches to other nodes
  - *Degree* of a node is the number of subtrees of the node
  - *Degree* of a tree is the maximum degree of the nodes in the tree
  - A node with degree zero is called the *leaf* or *terminal node*
  - Parent and child nodes
    - A node that has subtrees is called the *parent* of the root of the subtrees
    - The roots of the subtrees are the *children* of the node
    - Children of the same parent are called *siblings*
    - All the nodes along the path from the root to a node are the *ancestors* of that node
    - All the nodes in the subtree are the *descendants* of the root of the subtree (the node)

Binary tree

- A structure with a unique starting node (known as the *root of the tree*), in which each node is capable of having two *child nodes*, and in which a unique *path* exists from each node to every other node
- Each node in the binary tree may have 0, 1, or 2 children, or the degree of a node in a binary tree never exceeds 2
- The link to the left of a node is called its *left child* while the link to the right is called the *right child*, the node itself is called the *parent* of the children
- Each of the root node's children is itself the root of a smaller binary tree, or a *subtree*
  - The root node's left child is the root of its *left subtree*
  - The root node's right child is the root of its *right subtree*
- Recursive definition of a binary tree: *A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree*
- The children of a node, and the generations after that, are known as the *descendants* of the node, while the node itself is known as an *ancestor* of those descendants
- *Level* of a node
  - Defined as the distance of the node from the root
  - The level of root is 0 (zero)
  - The level of the children of the root is 1
  - The maximum level in a tree is called the *height* of the tree
  - The minimum number of levels for a given set of nodes in a binary tree is determined by giving each node two children until we run out of nodes

  - The minimum number of levels for n nodes is given by $\lfloor \lg n \rfloor$
  - The height of a binary tree is the critical factor in how fast the tree can be searched
    * With minimum height tree, any node can be searched in time $O(\lg n)$
    * Maximum height of the tree makes the search degenerate to $O(n)$

- Maximum number of nodes
  - The maximum number of nodes on level i of a binary tree is $2^i,\ i \geq 0$
  - The maximum number of nodes in a binary tree of height k is $2^{k+1} - 1,\ k \geq 0$

- Relation between number of leaf nodes and nodes of degree 2
  - For any nonempty binary tree T, if n_0 is the number of leaf nodes and n_2 is the number of nodes of degree 2, then, $n_0 = n_2 + 1$

- A *full binary tree* of height k is a binary tree of height k with $2^{k+1} - 1$ nodes, $k \geq 0$
- *Binary search tree*
  - A tree where the nodes with the value smaller than the value in the root are placed in the left subtree while the nodes with a value larger than that in the root are placed in the right subtree
  - The same property holds for each subtree, and the ones below
  - We could also define it as a binary tree in which the key value in any node is greater than the key value in its left child and any of its children (the nodes in the left subtree) and less than the key value in its right child and any of its children (the nodes in the right subtree)

Binary search tree ADT

- We'll assume that the tree will be accessed through a pointer to the root of the tree
- The structure of the tree
  - The placement of each element in the binary tree must satisfy the binary search tree property
  - The value of the key of an element is larger than the value of the key of any element in the left subtree, and smaller than the value of the key of any element in the right subtree

- Operations

  Creating the tree. Initializes the tree to an empty state
  Destroying the tree. Destroys all tree elements, leaving the tree empty
  Retrieving an element. Searches the binary tree for the element whose key is given and returns a copy of the element
  Inserting an element. Adds an element to the binary search tree
  Modifying an element. Replaces an existing tree element with the same key
  Deleting an element. Deletes an element specified by the key from the binary search tree
  Printing the tree. Prints all the elements in the order indicated by the *traversal order*

- The operations are similar to the list ADT

Implementation of a binary tree

- Tree data design
  - The nodes in a tree consist of several parts
  - The structure of each node is:

```
typedef int key_t;
typedef struct
{
    key_t   key;
    ...                     /* Other fields as needed */
} tree_element_t;

typedef struct node
{
    tree_element_t    info;    /* Information fields, including key */
    struct node     * left_child,
                    * right_child;
} node_t;

typedef tree_t *   node_t;

tree_t * bst;   /* Declare a pointer to the root of binary search tree */
```

  - This structure of a node makes it difficult to determine the parent of a given node but is adequate for most purposes

- Creating a binary tree
  - Easily done by assigning `NULL` to root

```
tree_t create_tree ( tree_t * tree )
{
    /* Initialize the tree to an empty state     */

    tree = NULL;
    return ( tree );
}
```

- Searching a binary search tree
  - Internal (private) operation in the tree ADT that is to be used by other tree operations
  - Specified by the `find_node` function that returns a pointer to the desired node, if it exists
  - We'll also return a pointer to the parent to facilitate deletion
  - If the key value is not found in the tree, it returns a `NULL`, with the parent pointer pointing to the logical parent of the node if it existed
  - We'll write a recursive function

```
tree_t find_node ( tree_t tree, key_t key, tree_t * parent )
{
    if ( ! tree )            /* Already reached a leaf        */
        return ( NULL );

    if ( tree->info->key == key )       /* Found the node     */
        return ( tree );

    *parent = tree;         /* Current node becomes the parent */
    if ( tree->info->key > key )
        return ( find_node ( tree->left_child, key, parent ) );
    else
        return ( find_node ( tree->right_child, key, parent ) );
}
```

  - Notice that `parent` is always passed by reference and so, acts more like a global variable to the function, the last change in the variable is transmitted back to the top

  - If the node is not found, the parent will contain the logical parent of the would-be node
  - If the node is same as root, `parent` is left undefined (should take care of this in the calling functions)

- Retrieving an element
  - Looks for a node that has a given key value and returns a copy of its information field
  - Designed to return a copy of the node's information field rather than letting the user access the node directly

```
tree_element_t * retrieve_element ( tree_t tree, key_t key )
{
    /* Searches the binary search tree for the element specified by the */
    /* key and returns a pointer to it.  Returns NULL if the element is */
    /* not found.                                                       */

    tree_t  node, parent;

    if ( node = find_node ( tree, key, &parent ) )
    {
        tree_element_t *tmp;
        tmp = (tree_element_t *) malloc ( sizeof ( tree_element_t ) );
        *tmp = node -> info;
        return ( tmp );
    }
    else
        return ( NULL );
}
```

- Modifying an element
  - Replaces the value in the information field of the node pointed to by the key
  - Does not modify the key field

```
#define FALSE 0
#define TRUE  1

tree_t modify_element ( tree_t tree, tree_element_t mod_element, \
   int *error )
{
    tree_t node, parent;

    *error = FALSE;

    if ( node = find_node ( tree, mod_element.key, &parent ) )
        node->info = mod_element;
    else
        *error = TRUE;

    return ( tree );
}
```

- Inserting an element
  - A new node is always inserted in the appropriate position as a *leaf*
  - It can be assumed that the key value is not found in the tree

```
#define FALSE 0
#define TRUE  1
```

```
tree_t insert_element ( tree_t tree, tree_element_t element, int \
          *error )
{
    tree_t node, parent;

    /* Initialize parent to NULL (to ensure its integrity if node is    */
    /* found in the root of the tree                                    */

    parent = NULL;

    /* Check if the element already exists in the tree             */

    if ( node = find_node ( tree, element.key, &parent ) )
    {
        *error = TRUE;
        return ( tree );
    }

    /* Element does not exist, logical parent of the node to be inserted */
    /* is in parent, insert the new node there                          */

    /* node is empty, make room for new element in there                */

    node = ( tree_element_t * ) malloc ( sizeof ( tree_element_t ) );
    node -> left_child = node -> right_child = NULL;
    node -> info = element;

    /* Check if this is the first node to be inserted in the tree       */

    if ( ! tree )
        return ( node );
    else
    {
        /* Not the first node; parent should already have a value       */

        if ( (parent->info).key > element.key )
            parent->left_child = node;
        else
            parent->right_child = node;
        return ( tree );
    }
}
```

  - Recursive version of `insert_element` – Do as an exercise using the `find_element` as a model (you do not need to use `find_element` in the recursive function)
  - Order of insertion of elements and its effect on the height of the tree

- Deleting an element
  - The operation finds the element specified by its key and deletes it from the tree
  - The find operation is straightforward (using the function `find_element`) but the deletion is more complicated and is handled by the following subcases
    1. Deleting a leaf (no children)
       * Simple case

    - Easily achieved by setting the appropriate link of the parent to `NULL` and then, freeing the memory allocated to the deleted node
  2. Deleting a node with only one child
    - We cannot simply delete the node as in the previous case as we do not want to lose the descendants of the deleted node
    - Make a pointer from the parent of the node to be deleted to skip over the deleted node and point to the child of the deleted node, and then, free the memory allocated to the deleted node
  3. Deleting a node with two children
    - Most complicated case
    - We cannot make the parent of the deleted node to point to both the children of the deleted node
    - We will replace the `info` part of the node with the `info` part of its logical predecessor – the node whose key is closest in value to, but less than, the key of the node to be deleted
    - The replacement node will have 0 or 1 child
    - We then delete the replacement node by changing one of its parent's pointers

```
#define FALSE 0
#define TRUE  1

tree_t delete_element ( tree_t tree, key_t key, int *error )
{
    /* Delete the element containing key from the binary search tree     */

    tree_t node, parent;

    *error = FALSE;
    parent = NULL;

    /* Check if the node exists, and get its position                    */

    if ( ! ( node = find_node ( tree, key, &parent ) ) )
    {
        *error = TRUE;       /* Node does not exist; return error        */
        return ( tree );
    }

    /* Node exists and the its position is known at this point           */
    /* Delete the node using a different function and return the         */
    /* resulting tree                                                    */

    return ( delete_node ( tree, node, parent ) );
}
```

– We need to develop the function `delete_node`

- The first two cases to delete a node (0 or 1 child) are simple
- In the third case, when the node has two children, we need to determine its logical predecessor; we do not actually delete the node but replace its contents with that of the logical predecessor which is to be deleted
- The logical predecessor is the largest key value in the tree that is less than the node to be deleted
- This value will be the largest key in the left subtree of the node

```
tree_t delete_max ( tree_t tree, tree_t * l_child )
{
    /* Find and remove the maximum element in the tree                */

    if ( ! tree->right_child )    /* There is no right child; stop recursion */
```

```
    {
        l_child = tree->left_child;  /* Left child needs to be saved         */
        return ( tree );             /* Return the pointer to max node       */
    }
    else
        delete_max ( tree->right, &l_child );
}
```

– With `delete_max`, we can write the function `delete_node` that we used in the function `delete_element`
  * We'll have to copy the information in the max node that is being deleted by `delete_max`

```
tree_t delete_node ( tree_t tree, tree_t node, tree_t parent )
{
    /* Deletes node pointed to by the node from the binary search tree     */

    tree_t tmp;

    /* Case when we have to delete a leaf                                  */

    if ( ( ! node->left_child ) && ( ! node->right_child ) )
    {
        if ( ! parent )                 /* node is the last node in tree   */
            return ( NULL );

        /* Delete the leaf node                                            */

        if ( parent->right_child == node )
            parent->right = NULL;
        else
            parent->left = NULL;

        free ( node );
        return ( tree );
    }

    /* Case when the node to be deleted has two children                  */

    if ( node->left_child && node->right_child ) /* node has two children */
    {
        /* Find and remove the replacement value from node's left subtree */
        /* tmp will contain the current left child                        */

        tmp = delete_max ( node->left_child, node->left_child );

        /* Replace the delete element                                     */

        node->info = tmp->info;

        /* Set node to point to the node to be deleted to free memory     */

        node = tmp;
    }
    else                    /* Node has one child                         */
    {
        /* Reset one of the pointer fields of the parent node based on    */
```

```
        /* whether the node being deleted has a left child or right child */

        if ( node->right_child )        /* Node has a right child         */
            if ( ! parent )
                tree = node->right_child;   /* Delete root                */
            else                            /* Delete non-root node       */
                if ( parent->right_child == node )
                    parent->right_child = node->right_child;
                else
                    parent->left_child = node->right_child;
        else                            /* Node has a left child          */
            if ( ! parent )
                tree = node->left_child;   /* Delete root                */
            else                            /* Delete non-root node       */
                if ( parent->right_child == node )
                    parent->right_child = node->left_child;
                else
                    parent->left_child = node->left_child;
    }

    /* Free the memory occupied by the deleted node                       */

    free ( node );

    return ( tree );
}
```

Tree traversals

- Traversing a tree means visiting all its nodes, for example to print the information field at each node of the tree
- Different ways of traversal include pre-order traversal, post-order traversal, in-order traversal, and level-order traversal
- All the traversals start at the root of the tree, and are most easily implemented as recursive functions
- In-order traversal
  - The algorithm can be summarized as

    ```
    visit the left subtree
    visit the root of the tree
    visit the right subtree
    ```

  - Obviously, it is in-order as the root is sandwiched between the traversal of left and right subtrees
  - The implementation is

    ```
    void print_in_order ( tree_t tree )
    {
        if ( ! tree )                 /* Recursion stops when a leaf is reached */
        {
            print_in_order ( tree->left_child );
            print_info ( tree->info );
            print_in_order ( tree->right_child );
        }
    }
    ```

  - In-order traversal printing results in the entire tree being printed in sorted order (smallest to largest)

- Pre-order traversal
  - The algorithm can be summarized as

```
visit the root of the tree
visit the left subtree
visit the right subtree
```

  - The implementation is

```
void print_pre_order ( tree_t tree )
{
    if ( ! tree )                  /* Recursion stops when a leaf is reached */
    {
        print_info ( tree->info );
        print_pre_order ( tree->left_child );
        print_pre_order ( tree->right_child );
    }
}
```

- Post-order traversal
  - The algorithm can be summarized as

```
visit the left subtree
visit the right subtree
visit the root of the tree
```

  - The implementation is

```
void print_post_order ( tree_t tree )
{
    if ( ! tree )                  /* Recursion stops when a leaf is reached */
    {
        print_post_order ( tree->left_child );
        print_post_order ( tree->right_child );
        print_info ( tree->info );
    }
}
```