CMSC 123: Data Structures

1st Semester AY 2019-2020

Prepared by: Clinton Poserio

Lab 04: Binary Search Trees

In this laboratory topic, a new ADT, which utilizes a non-linear structure, is introduced. We also explore how this new structure improve several operations used to access and modify stored data items.

(Generic) Tree ADT

Before we begin with Binary Search Trees, we review Trees.

Definition and Properties

In your graph theory class, a `tree` is defined as a graph with no cycles, where any two vertices (nodes) are connected by exactly one edge (path). As an ADT, we define a `tree` as an ADT that simulates a hierarchical structure; unlike lists, stacks, and queues, trees have a *non-linear structure*. A tree ADT has also a set of operations such as, but not limited to, insertion, deletion, searching, and enumeration. Commonly, a tree is represented as a set of *linked nodes*, where one node is set as root.

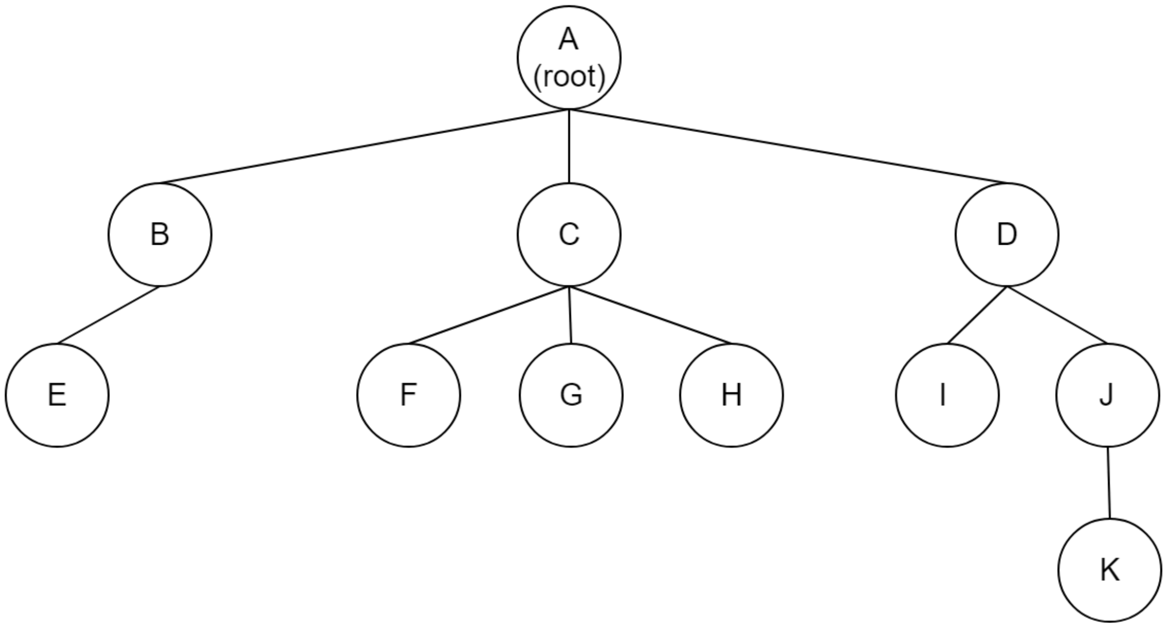

Figure 1: Tree Example

Unlike *natural trees*, the root in `tree` ADT is located at the top. A sample tree rooted at node *A* is illustrated in Figure 1. Each node can have a *subtree* or child nodes. Similarly, child nodes have *parent nodes*.

Tree Terminology

Here are key words commonly used when discussing trees.

- Hierarchical Terms
  - **Node** - represents a vertex; a structure containing a value
  - **Edge** - the link or connection between two nodes
  - **Root** - the top node in a tree; a node without a *parent*
  - **Parent** - the immediate node when moving closer to the root
  - **Child** - the converse of the parent node
  - **Siblings** - nodes with same parent
  - **Neighbor** - node connected to a node; a parent or a child
  - **Ancestor** - node reachable when repeatedly moving from child to parent (*e.g.* parent, grandparent, grand-grandparent *etc.*)
  - **Descendant** - node reachable when repeatedly moving from parent to child (*e.g.* child, grandchild, *etc.*)
  - **Leaf** - (*a.k.a.* external node) a node with no children
  - **Internal node** - a non-leaf node
- Measures
  - **Path** - a sequence of nodes (or edges) from a node to its descendant
  - **Distance** - the number of edges along the shortest path between two nodes
  - **Depth** - the distance between a node and the root
  - **Level** - depth of a node plus one
  - **Height** - the number of edges on the longest path from a node to a descendant leaf
  - **Height of the tree** - height of the root node
  - **Size of the tree** - the number of nodes in the tree

Implementation

A *generic* `tree` ADT, that is, a tree with no specified *branching factor* (the maximum number of child nodes per node) can be hard to implement, because no bounds or parameters were preset. However, a common approach simplifies this by simply adding two pointers in a node - the pointers to the *first child* and to the *next sibling* of the node.

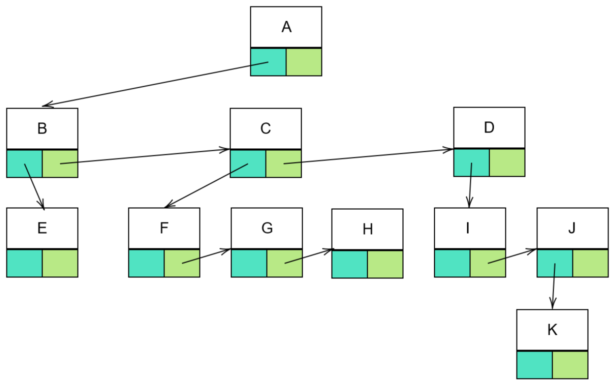

Figure 2: Tree ADT for Figure 1

Using this implementation for the tree in Figure 1, we get the tree in Figure 2. Each node contains a value, a first-child pointer, and a next-sibling pointer indicated by an uppercase letter, a blue box, and a green box, respectively. This can be easily implemented in `C`; so, we suggest you try!

Binary Trees

A **binary tree** is a tree in which the branching factor is two; that is, every node can have at most two children, which are typically called the *left* child and *right* child.

The generic tree implementation shown at Figure 2 can be considered a binary tree - treat the first child node as the left child node and treat the next sibling node as the right child node. To visualize this, an illustration is shown in Figure 3.

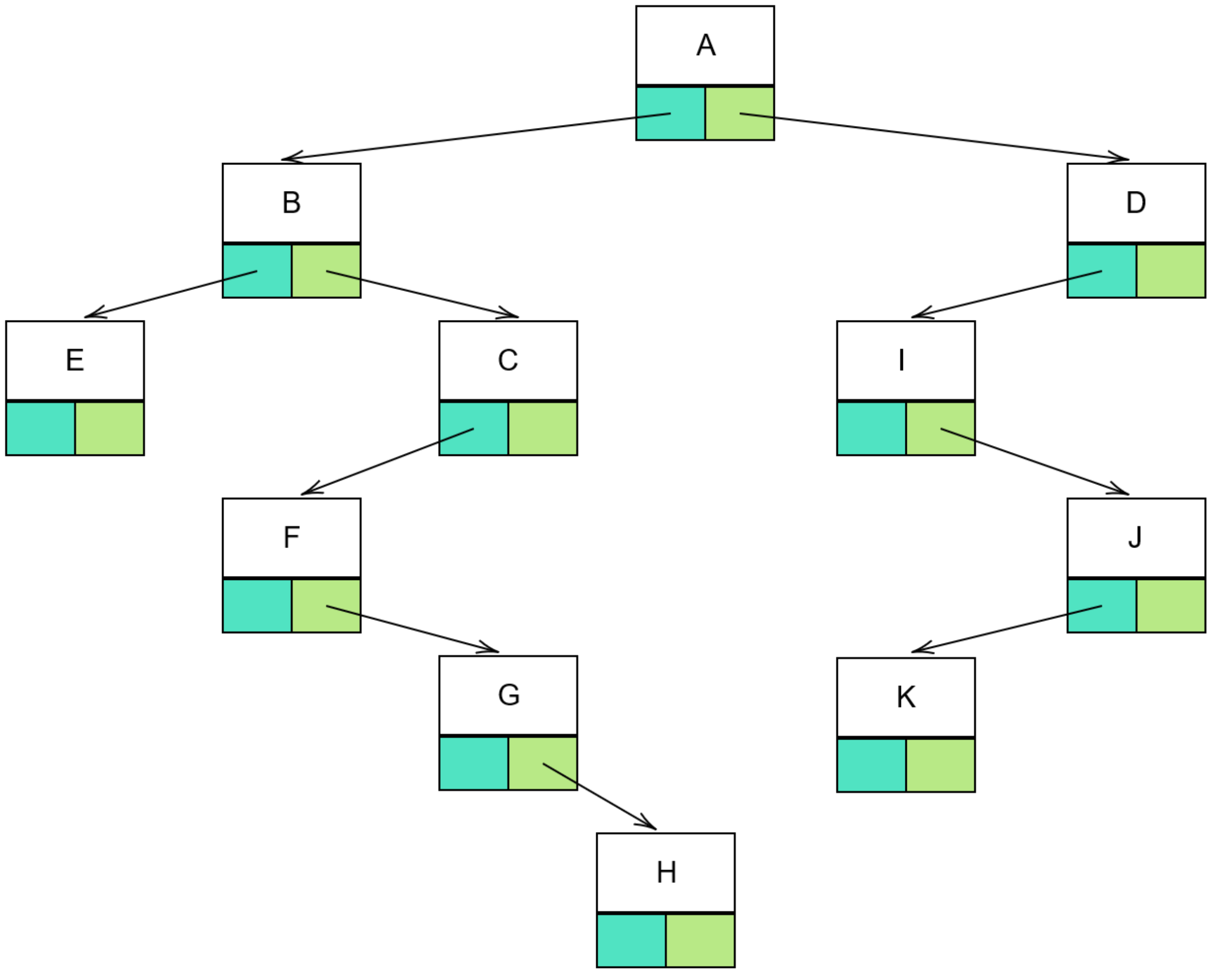

Figure 3: Figure 1 as a Binary Tree

Figure 3 is no different from Figure 2; the nodes were reorganized to quickly see the *binary property*. Since a generic tree structure can be implemented as described above; therefore, we can conclude that any generic tree can be converted or restructured as a binary tree. Moreover, binary trees are very useful because it provides more efficient insertion and deletion. Other operations are also improved by specific types of binary trees.

Binary Search Trees

Definition and Properties

A **binary search tree** (simply, BST) is a binary tree which can be represented also as a linked data structure, where each node contains a *key*, a data item, a left child pointer, and a right child pointer (additionally, we add a parent pointer to make links bidirectional). The keys in a BST are stored in such a way that satisfies the *binary-search-tree property*:

> Let `x` be a node in a BST. If `y` is a node in the left subtree of `x`, then `y->key < x->key`. If `y` is a node in the right subtree of `x`, then `y->key > x->key`.

In other words, the key of a node is greater than the keys of those in the node's left subtree; moreoever, it is less than the keys of those in the node's right subtree. Although other literatures use a non-strict condition, that is, a node's key could be less than *or equal* to the keys of those in the right subtree and greater than *or equal* to those keys in the left subtree, we use the strict condition to avoid duplicate keys in the BST.

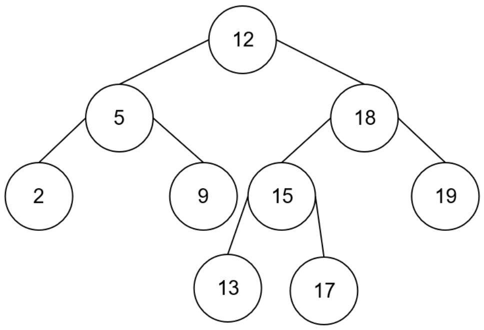

Figure 4: Sample BST with integer keys

Illustrated in Figure 4 is a BST rooted at the node with key 12. It can be observed that each node in this BST follows the binary-search-tree-property described earlier.

BST ADT

In BST, data are organized using **linked nodes** and the root node is stored. The nodes are arranged according to their keys, following the BST property. In `C`, we defined its structure as:

```c
typedef struct bst{
    BST_NODE* root;
    int maxSize;
    int size;
}BST;
```

The current size of the BST is also stored and a maximum size is set to simulate overflow. Furthermore, each node of a BST contains a `key`, `left`, `right`, and `parent` pointers. Data item is not included to focus on the behavior of the keys. Also, the height of the node is also stored. We defined it in a `C structure` as follows (see `BST.h`):

```c
typedef struct bst_node{
    struct bst_node* left;
    struct bst_node* right;
    struct bst_node* parent;
    int key;
    int height;
} BST_NODE;
```

BST Operations

Here are the operations associated with `BST` ADT and basic descriptions (see `BST.h` for detailed description):

1. `BST_NODE* createBSTNode(int, BST_NODE*, BST_NODE*, BST_NODE*)` - creates a new BST node that sets its fields using given parameters

2. `BST* createBST(int)` - creates an empty BST and sets its fields; maximum size is set using the given parameter

3. `int isEmpty(BST*)` - returns `1` if the BST is empty; otherwise, returns `0`

4. `int isFull(BST*)` - returns `1` if the BST is full; otherwise, returns `0`

5. `void insert(BST*, BST_NODE*)` - inserts the given node to the BST

6. `BST_NODE* search(BST*, int)` - returns a pointer to the node containing the given search key, if found; otherwise, returns `NULL`

`showTree()` is already provided to help you in debugging. Take note that this functions displays the BST in tree mode; that is, the tree is rotated +90° (*i.e.* root is at the leftmost, right subtree is at the top, and left subtree is at the bottom).

Questions?

If you have any questions, approach your lab instructor.

References

Cormen, Thomas H., Charles E Leiserson, Ronald L Rivest, and Clifford Stein. 2009. *Introduction to Algorithms.* MIT press.

Weiss, Mark Allen. *Data Structures and Algorithms.* Benjamin/Cummings.