

COMP1917: Computing 1

16. Binary Search Trees

Reading: Moffat, Section 10.3, 10.5

Binary Search Trees - Motivation

- a Linked List is a one-dimensional recursive structure – each node has one pointer to the next node
 - ▶ Problem: finding, deleting or inserting items takes a long time because of the need to **linearly** search for the item of interest
- a Binary Tree is constructed from nodes, where each node contains:
 - ▶ a “left” pointer (which could be NULL)
 - ▶ a “right” pointer (which could be NULL)
 - ▶ a “data” value
- this leads to a multi-branching recursive structure which can speed up finding, deleting and inserting of items considerably

Binary Tree Structure

We define a self-referential structure similar to a Linked List, but with **two** pointers, to the “left” and “right” branch:

```
typedef struct tnode Tnode;

struct tnode {
    int    data;
    Tnode *left;
    Tnode *right;
};
```

Binary Trees

- a “root” pointer points to the topmost node in the tree
- the left and right pointers recursively point to smaller “subtrees” on either side
- a NULL pointer represents a binary tree with no elements – an empty tree (or subtree)

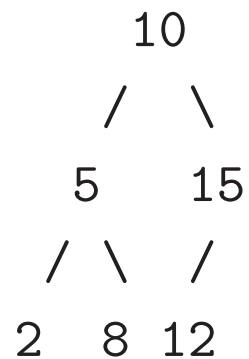
Binary Search Tree

- a **Binary Search Tree** is a binary tree in which the items are **ordered** from left to right across the tree.
- To ensure the items remain ordered, new items must be inserted according to these rules:
 - ▶ if the new item has a data value **less than** the data value at this node, it is recursively inserted into the **left** subtree
 - ▶ if the new item has a data value **greater than** the data value at this node, it is recursively inserted into the **right** subtree

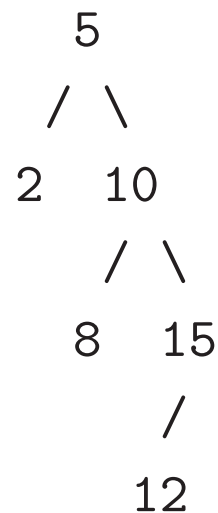
Building a Binary Search Tree

The shape of the tree depends on the **order** in which the nodes are inserted:

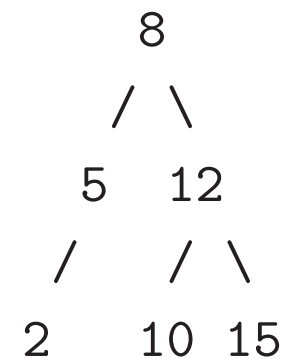
10, 5, 15, 2, 8, 12



5, 2, 10, 8, 15, 12



8, 12, 5, 15, 2, 10



Question: what if the order is 2, 5, 8, 10, 12, 15 ?

Binary Search Tree Operations

```
Tnode * makeTnode( int data ); // create new node
```

```
Tnode * findTnode( int data, Tnode *root );
```

```
Tnode * insertTnode( Tnode *new_node, Tnode *root );
```

```
void printTree( Tnode *root ); // print all items
```

```
void freeTree ( Tnode *root ); // free entire tree
```

```
int treeSize ( Tnode *root ); // number of items
```

```
int treeHeight( Tnode *root ); // max depth of an item
```

Making a New Node

```
/*  
 * Create a new Tnode with the specified data value.  
 */  
Tnode * makeTnode( int data )  
{  
    Tnode *new_node =(Tnode *)malloc( sizeof( Tnode ));  
    if( new_node == NULL ) {  
        fprintf(stderr,"Error: memory allocation failed.\n");  
        exit( 1 );  
    }  
    new_node->data  = data;  
    new_node->left  = NULL;  
    new_node->right = NULL;  
    return( new_node );  
}
```


Finding a Node in the Tree

```
/*  
 * Search tree to find a node with specified data value.  
 */  
Tnode * findTnode( int data, Tnode *root )  
{  
    Tnode *node = root; // start at the root of the tree  
  
    // keep searching until data found, or exit from tree  
    while(( node != NULL )&&( node->data != data )) {  
        if( data < node->data )  
            node = node->left; // branch left or right,  
        else // depending on data value  
            node = node->right;  
    }  
    return( node );  
}
```

Recursive version of findTnode()

```
/* Search tree to find a node with the specified
   data value, and return a pointer to this node.
   */ If no such node exists, return NULL.
Tnode * findTnode( int data, Tnode *root )
{
    if(( root == NULL )||( root->data == data )) {
        return( root );      // found node, or exited tree
    }
    else if( data < root->data ) { // search left subtree
        return( findTnode( data, root->left ));
    }
    else {                    // search right subtree
        return( findTnode( data, root->right ));
    }
}
```

Insert New Node into Binary Search Tree

```
Tnode * insertTnode( Tnode *new_node, Tnode *root )
{
    Tnode *child = root, *parent = NULL;
    while( child != NULL ) { // find parent for new node
        parent = child;
        if( new_node->data < parent->data )
            child = parent->left;
        else
            child = parent->right;
    }
    if( parent == NULL ) // tree was empty
        root = new_node;
    else if( new_node->data < parent->data )
        parent->left = new_node; // insert to the left
    else
        parent->right = new_node; // insert to the right
    return( root );
}
```

}

Recursive version of `insertTnode()`

```
Tnode * insertTnode( Tnode *new_node, Tnode *root )
{
    if( root == NULL ) { // we have reached the insertion point
        root = new_node;
    }
    else if( new_node->data < root->data ) {
        // insert new node into (and update) left subtree
        root->left = insertTnode(new_node, root->left);
    }
    else { // insert new node into (and update) right subtree
        root->right = insertTnode(new_node, root->right);
    }
    return( root );
}
```

Printing a Binary Search Tree

```
/*  
 * Print all items of the tree in order  
 */  
void printTree( Tnode *root )  
{  
    if( root != NULL ) {  
        printTree( root->left ); // recursively print smaller items  
        printf("%c",root->data ); //          print current item  
        printTree( root->right ); // recursively print larger items  
    }  
}
```

Freeing all items from a Tree

```
/*  
 * Recursively free all the items from a Binary Tree  
 */  
void freeTree( Tnode *root )  
{  
    if( root != NULL ) {  
        freeTree( root->left );  
        freeTree( root->right );  
        free( root );  
    }  
}
```

Computing the Size of a Tree

```
/*  
    Compute the size of a Binary Tree  
    (the number of items stored in the tree)  
*/  
int treeSize( Tnode *root )  
{  
    if( root == NULL ) {  
        return( 0 );  
    }  
    else {  
        return( 1 + treeSize( root->left  )  
                + treeSize( root->right ));  
    }  
}
```

Computing the Height of a Tree

```
int treeHeight( Tnode *root )
{
    int leftHeight, rightHeight;
    if( root == NULL ) {
        return( 0 );
    }
    else {
        leftHeight  = treeHeight( root->left );
        rightHeight = treeHeight( root->right );
        if( leftHeight > rightHeight )
            return( 1 + leftHeight );
        else
            return( 1 + rightHeight );
    }
}
```


Questions

- what is the maximum number of items that could be stored in a BST with height H ?
- if the number of items in a BST is N , what is the minimum height such a tree could have?
- how would you **delete** an item from a BST (so that the BST structure is preserved)?