Binary Search Trees

What is a Binary Tree (recap)

- **Trees**: Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures.

- A general tree can be defined in several ways. One natural way to define a tree is recursively. A tree is a collection of nodes. The collection can be empty; otherwise, a tree consists of a distinguished node, r, called the root, and zero or more nonempty (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by an edge from $r$.

- The root of each subtree is said to be a child of $r$, and $r$ is the parent of each subtree root.

- When the number of subtrees of any node in a tree is $\leq 2$, the tree is called a binary tree.

- Terminology: The topmost node is called **root** of the tree. The elements that are directly under an element are called its children. The element directly above something is called its **parent**. For example, ‘a’ is a **child** of ‘f’, and ‘f’ is the parent of ‘a’. Finally, elements with no children are called **leaves**.
**Binary Search in a Sorted Array**

- Consider an array $A[1...n]$ of integers, sorted in ascending order; we want to search if a given integer (search key), say $k$, is contained in the array. Two possibilities: (1) search is successful; output position $i$ of the search key; (2) search is unsuccessful; output the position after which the key may be placed keeping the array sorted [if $k$ is less than the minimum element in $A[]$, output 0; if $k > A[n]$ output $n$].

```c
binary_search(A, target){
    lo = 1, hi = size(A)
    while (lo <= hi) {
        mid = lo + (hi-lo)/2
        if A[mid] == target:
            return mid
        else if A[mid] < target:
            lo = mid+1
        else:
            hi = mid-1
    }
    return hi;
}
```

1. The program may be slightly incompatible with the requirement specification; if so, fix it; otherwise, justify it meets the specifications.
2. Arrays are not very good for general purpose lists or sets where we need an order or frequent dynamic insert/delete.
3. Binary Trees are much better for some problems.
4. Binary Search Tree (BST in short) is a binary tree with some special properties [i.e., any binary tree is not necessarily a BST]

**Binary Search Tree (BST)**

- A binary search tree is organized as a binary tree with some special property. We represent such a tree by a linked data structure in which each node is a struct. In addition to a key (for simplicity we assume it’s an integer; it can be anything just like in any kind of linked lists), each node contains two pointers left and right; usually no information is explicitly stored about the parent of a node; if left (right) child does not exist, the corresponding link contains a NULL.

```c
typedef struct node{
    struct node *left;
    int key;
    struct node *right;
} node;
```

- The keys in a binary search tree are always stored in such a way as to satisfy the binary-search-tree property: “Let $x$ be a node in a binary search tree. If $y$ is a node in the left subtree of $x$, then $y.key \leq x.key$. If $y$ is a node in the right subtree of $x$, then $y.key \geq x.key$.”

- We can have many different BSTs with the same set of keys [computing how many is outside our scope in this class]. Any given BST can be specified by the address (pointer) of the root; an empty tree is represented by the NULL pointer (Create an empty BST using node *Root =NULL;
Binary Search Tree (BST)

Observe:
- Each node has a unique path to the root of the tree.
- The maximum of the length of all such paths is called the height of the tree.
- \# nodes = n = \# edges + 1 (it is true for any tree, binary or not); why?
  [hint: It is easy to prove the statement. An edge has two distinct endpoints (nodes) and the tree by definition consists of disjoint subtrees...]
- In the worst case, height of the tree can be n – 1; example??

Operations on BSTs

- **Set / Dictionary**: maintain dynamic collection of elements
  - Insert (k) : insert key k
  - Remove (k) : remove (delete) key k
  - Find (k) : is key k present?
  - Traversals (or walks) [will soon define]
- **Map**: maintain collection of (key, value) pairs
  - Insert (k, v) : insert key k with associated value v
  - Remove (k) : remove record with key k
  - Find (k) : return value associated with key k (or report that k isn’t in the structure)
- **How to store a BST?**
  - We can maintain a BST by using a sorted/unsorted array, or linked (singly/doubly) list, – all depends on what we want to do in the applications. We will use linked list implementations in this course.
Store BST using linked lists

Usually, we use a struct like this: typedef Struct node{struct node *left; int key; struct node *right;} node; Remember, we can have many other info in place of key alone; say in stead of value [see above] we can have a pointer pointing to a record of a file. Advantage: we can change the file structure without disturbing the BST.

To create an empty tree, we use node *root = NULL [Just like a linked list, the pointer variable root is a gateway to the entire tree.]

Sometimes, we can add another component to the structure like struct node *parent to keep the information of the parent of a node in a BST.

If we need to store a sparse matrix, we need to have quadruple linked lists (4 pointers: left, right, up, down); we will not use them in this course; just so you know they are possible and fascinating.

Search, Insert

Search a BST T (Tree root pointer) for a given key (Find the tree node containing key k) [will return the pointer to the desired node]: Execution time is O(n) in the worst case

- If the tree is empty or the key is not found, the function returns NULL.
- If the tree is not empty, then compare the search key k with the key of the node; if equal, search is successful; otherwise go down either the left link or right link depending on if k is less or ≥. One can easily write a recursive routine [or iterative routine in C or C++]

Insert a node with key k in T: Search for k; if found exit with an error; otherwise when search returns a NULL, create (malloc) a new node, adjust the fields. One can easily write a recursive routine [or iterative routine in C or C++]
Size of a Binary Tree

**Problem**: Given a binary tree, we need a non-recursive algorithm to find the size of the tree. **Note**: Size of the tree is number of nodes in the tree.

The idea is to traverse the nodes level by level, the root being at the first level and then the children of the root and so on.

Create an empty queue q

temp_node = root /*start from root*/

Loop while temp_node is not NULL

a) Enqueue temp_node’s children (first left then right children) to q
b) Increase count with every en-queuing.
c) De-queue a node from q and assign it’s value to temp_node

**Time Complexity**: O(n)

**Auxiliary Space**: O(level_max) where level max is maximum number of node in any level of the given tree.

**Note**: One can also similarly compute the size of any subtree rooted at any other node.

Height (rank) of a node in a BST

The **height** of the binary tree is the longest path from root **node to any leaf** node in the tree.

Calculating minimum and maximum height from number of nodes – If there are n nodes in binary tree, maximum height of the binary tree is n-1 and minimum height is floor(log₂n).

Calculating minimum and maximum number of nodes from height – If binary tree has height h, minimum number of nodes is h+1 (in case of left skewed and right skewed binary tree).

Calculating minimum and maximum number of nodes from height – If binary search tree has height h, minimum number of nodes is h+1 (in case of left skewed and right skewed binary search tree). If binary search tree has height h, maximum number of nodes will be when all levels are completely full. Total number of nodes will be \(2^0 + 2^1 + \ldots \cdot 2^h = 2^{(h+1)} - 1.\)
**How to compute the height of any node in a BST**

- Recursively calculate height of left and right subtrees of a node and assign height to the node as max of the heights of two children plus 1.

\[
\text{maxDepth}(\text{tree})
\]

1. If tree is empty then return 0
2. Else
   a. Get the max depth of left subtree recursively i.e., call \(\text{maxDepth}(\text{tree->left-subtree})\)
   b. Get the max depth of right subtree recursively i.e., call \(\text{maxDepth}(\text{tree->right-subtree})\)
   c. Get the max of max depths of left and right subtrees and add 1 to it for the current node.
      \[
      \text{max_depth} = \max(\text{max dept of left subtree, max depth of right subtree}) + 1
      \]
   d. Return \(\text{max_depth}\)

**Traversals**

- Tree Traversals: Enumeration [visiting (printing) once] of the contents [keys] of the nodes of a BST in some order. (1) **Inorder**: left, root, right; (2) **Preorder**: root, left, right; (3) **Postorder**: left, right, root;
- Pseudocode (recursive) for Inorder traversal: An inorder traversal prints the elements in sorted order in \(O(n)\) steps. Let us see examples. Try to write iterative programs.

**Inorder (T):**
- if \(T == \text{NULL}\), then return
- \(\text{Inorder}(T->\text{left})\)
- print \(T->\text{key}\)
- \(\text{Inorder}(T->\text{right})\)

**Preorder (T):**
- if \(T == \text{NULL}\), then return
- Print \(T->\text{key}\)
- Preorder \((T->\text{left})\)
- Preorder \((T->\text{right})\)

**Postorder (T):**
- if \(T == \text{NULL}\), then return
- Postorder \((T->\text{left})\)
- Postorder \((T->\text{right})\)
- Print \(T->\text{key}\)

(A) 2 5 5 6 7 8  (B) 2 5 5 6 7 8
Another Example

InOrder(root) visits nodes in the following order:
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

A Pre-order traversal visits nodes in the following order:
25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90

A Post-order traversal visits nodes in the following order:
4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

Search, Insert, Max, Min, Traversals (in C)

```c
node *search (node **lp, int key){
    node *current = *lp;
    while (current != NULL && current->info != key)
        if (key < current->info) current = current->left;
        else current = current->right;
    return (current); } // One can check if the key exists.

void insert (node **lp, int key){
    node *current, *prev, *new;
    current = *lp;
    prev = NULL;
    while (current != NULL) {
        prev = current;
        if (key < current->info) current = current->left;
        else current = current->right;
    }
    new = (node *)malloc(sizeof(node));
    new->info = key;
    new->left = new->right = NULL;
    if (prev == NULL) *lp = new; else {
        if (key < prev->info) prev->left = new;
        else prev->right = new; }
}

int min (node **lp){
    node *x = *lp;
    while (x->left != NULL)
        x = x->left;
    return (x->info); } //Assuming the node is at least one node.

Note: int max is obtained by replacing left with right.

void inorder_recursive (node **lp){
    node *x = *lp;
    if (x != NULL)
        {inorder_recursive (&x->left);
         printf("%d  ", x->info);
         inorder_recursive (&x->right);
        }
}

Note: preorder and post order traversals are similar.

Q: How to write non recursive traversals? [Use stacks to remember the path!]

Note: One can maintain a stack of pointers to keep track of the nodes while searching. Will be useful in some applications.
```
Predecessor & Successor

Predecessor (Successor) of any node in a given binary search tree is usually defined in terms of the unique inorder traversal of the tree; the node may be specified as a key or the address of the node (depending on application requirement) – accordingly, the function (method) has to be adjusted – the principle remains the same.

We’ll study some scenarios first to understand the logic.

- inorder traversal: 1 2 4 6 8 11 12 15 20 21 23 25
- Say, key is 4 ⇒ its right link is not null ⇒ 4’s successor must be the minimum of the node 8, which is 6.
- Say key is 11 ⇒ right link is null ⇒ its successor must be somewhere up in the tree, go to the predecessor, say node p ⇒ two cases:
  - Case 1: key is NOT on its predecessor’s right link ⇒ the key of p is key’s successor (if p is not null) [i.e., 12 is successor of 11]
  - Case 2: if p is not NULL and key is on its predecessor’s right link, go up the chain and repeat.
- Be careful to handle the maximum key in the BST, that does not have a successor

Writing a function (method) for successor is relatively straightforward – remember to maintain a stack of pointers when searching for the tree [necessary to move up]; actually, stack is not needed; one can do it without a stack.

Note: Predecessor is similar to successor – left links will replace the right links.

Successor of a node

```c
int successor (node **lp, int key){

    node *root = *lp, *succ = NULL, *keynode = search (&root, key);

    // Start from root and search down the tree
    while (root != NULL)
    {
        if (keynode->info < root->info)
            [ succ = root; root = root->left; ]
        else
            if (keynode->info > root->info) root = root->right;
            else break;
    }

    if (succ == NULL) return -100;
    else return succ->info;
}
```

Note: if one needs to have a function that returns the address of the successor of the info of the successor node, one needs to redefine the function as

```c
node *successor_node ((node **lp, int key);
```

Observe the same code will work up until the last if statement; the last if statement needs be replaced by return suc !!

One can adjust the code in a relatively straightforward way to get a function for the inorder predecessor; try that.
Removing a node from a BST is a bit more tricky, since we do want to make sure that the BST remains a BST after the deletion. If the node has one child then the child is spliced to the parent of the node. If the node has two children then its successor has no left child; copy the successor into the node and delete the successor instead. TREE-DELETE (T, z) removes the node pointed to by z from the tree T. There are 3 possibilities:

- **The node to be deleted is a leaf node:**
  Easiest; Just delete it; that’s all [Be careful when deleting the root]

- **Node to be deleted has only one child (left or right):**
  Connect the only subtree of the node to the parent of the node to be deleted and delete.

- **Node to be deleted has two children:**
  **Tricky:** Find inorder successor of the node. Copy contents of the inorder successor to the node (to be deleted) and delete the inorder successor.

Note that in the third case inorder successor of the node, say z, (to be deleted) is the minimum of the node z->right.

### C Code for delete

```c
void delete (node **lp, int key){
    node *root = *lp;
    //succ = NULL, *keynode = search (&root, key);
    // base case: the tree is empty.
    if (root == NULL) return;
    // if key is less than root, key is on root's left subtree
    if (key < root->info) delete (&root->left, key);
    // if key is greater than root, key is on root's right subtree
    else if (key > root->info) delete (&root->right, key);
    // if key is equal to root, root is to be deleted.
    else {
        // node with one or no child
        if (root->left == NULL) {
            *lp = root->right; free (root); return; }
        else if (root->right == NULL) {
            *lp = root->left; free (root); return; }

        // node with two children; get inorder successor
        node *temp = minnode (&root->right);
        // copy the inorder successor's content to this node
        root->info = temp->info; *lp = root;
        delete(&root->right, temp->info);
    }
}
```

We have used a helper function (a variation of the min function we have seen before.

```c
node *minnode (node **lp){
    node *x =*lp;
    while (x->left != NULL)
        x = x->left;
    return x;
}
```

Note: It is not very difficult; try to understand the logic as explained by the example and the comments embedded in the code. And, yes it is a recursive routine; the iterative version would be a bit more complicated. If you understand the logic you should be able to do the deletions by hand on small examples.

Practice with several arbitrary BSTs.
Trim a BST

Problem Statement: Given the root of a binary search tree and 2 numbers min and max, trim the tree such that all the numbers in the new tree are between min and max (inclusive). The resulting tree should still be a valid binary search tree. Given the BST on the left and min = 5, max = 13, the resulting BST is shown on the right.

The Approach: We should remove all the nodes whose value is not between min and max. We can do this by performing a post-order traversal of the tree. We first process the left children, then right children, and finally the node itself. So we form the new tree bottom up, starting from the leaves towards the root. As a result, while processing the node itself, both its left and right subtrees are correctly trimmed binary search trees (may be NULL as well).

Note: The complexity of this algorithm is O(N), where N is the number of nodes in the tree, because we basically perform a post-order traversal of the tree, visiting each and every node once. Write the code and experiment – it will enhance your understanding of simple BSTs.

Threaded BSTs

Threaded Binary trees: We note that binary tree leaf nodes have lots of nulls – we want to utilize those spaces to expedite traversals, e.g., doing traversals without using recursion or stacks. We have the pointers reference the next node in an inorder traversal; called threads; we need to know if a pointer is an actual link or a thread, so we keep a Boolean flag for each pointer.

Single Threaded: each node is threaded towards either the in-order predecessor or successor (left or right) means all right null pointers will point to inorder successor OR all left null pointers will point to inorder predecessor.

Double Threaded: each node is threaded towards both the in-order predecessor and successor (left and right) means all right null pointers will point to inorder successor AND all left null pointers will point to inorder predecessor.

Write codes for traversals without using recursion or explicit stack(s), in a threaded tree. Also, given a tree, write code to make it single or double threaded.
### Practice Problems

1. Convert binary tree to its Sum tree: Sum tree of a binary tree, is a binary tree where each node in the converted tree will contains the sum of the left and right sub trees in the original tree.

2. Find the lowest common ancestor of two give nodes.

3. Convert a given binary tree into a threaded binary tree.

4. Print the postorder traversal of a tree given its preorder and inorder traversals without constructing the tree.

5. Find the diameter of a tree.

6. Construct Binary Search Tree from a given Preorder Traversal using Recursion and then get an iterative solution.

### Traversal by Link Reversal (Non Recursive and Stack less)

In a link-reversal traversal, as we descend into a binary tree, we replace normally downward pointing pointers in the link fields of nodes with upward pointers, which point to immediate ancestors. On the way up again, we restore pointer fields to their original condition containing downward pointers. Thus, as the algorithm traces out a path down into the lower parts of the tree, it leaves a "reverse" path to use to climb back out again.

Assume that each node x contains an additional flag field such that initially flag(x) = 0.

```c
typedef struct node (struct node *left, int info, bool flag, struct node *right);
```

The flag field is set to 1 when the right field is set to point upwards to its parent. It is reset to 0 when the right field is reset to point downward to its original right child node.

The flag field is set to 1 when the right link field is set to point upwards to its parent. It is reset to 0 when the right link field is reset to point downward to its original right child node.

We will write a composite traversal algorithm pseudocode (Non Recursive and Stack less) using this slightly modified node structure.
**Composite Traversal Pseudocode**

1. **[Initialize.]** Set PRES to point to the root of the tree, and set PREV = Null
2. **[Preorder visit.]** Visit node PRES if traversing in preorder.
3. **[Descend left.]** Set NEXT = left(PRES). If NEXT ≠ Null, then set left(PRES) = PREY, PREV = PRES, PRES = NEXT, and go to step 2.
4. **[Symmetric order visit.]** Visit node PRES if traversing in symmetric order.
5. **[Descend right.]** Set NEXT = right(PRES). If NEXT ≠ Null, then set TAG(PRES) = 1, right(PRES) = PREV, PREV = PRES, PRES = NEXT, and go to step 2.
6. **[Postorder visit.]** Visit node PRES if traversing in postorder.
7. **[Go up.]** If PREY = A, then the algorithm terminates. Otherwise, if TAG(PREV) = 0, then set NEXT = left(PREV), left(PREV) = PRES, PRES = PREV, PREV = NEXT, and go to step 4. Otherwise, set NEXT = right(PREV), TAG(PREV) = 0, right(PREV) = PRES, PRES = PREV, PREY = NEXT, and go to step 6.

---

**Non Recursive Inorder minus the extra memory**

```c
void inorder(node **lp) {
    node *parent = *lp, *tmp;
    while (parent != NULL) // The outer loop
        if (parent->left == NULL)
            printf("%d  ", parent->info);
        else
            {tmp = parent->left;
             while(tmp->right != NULL && tmp->right != parent)
                 tmp = tmp->right;
             /* Make parent as right child of its inorder predecessor */
             if (tmp->right == NULL){tmp->right=parent;
                                 parent = parent->left;}
            /* Revert the changes made in if part to restore the original
             tree i.e., fix the right child of predecessor */
            else
               {printf("%d  ", parent->info); tmp->right=NULL;
                 parent=parent->right; }
        }
}
```
An Example Execution

The Euler tour traversal of a binary tree $T$ [n nodes and n-1 edges] can be informally defined as a “walk” around $T$, where we start by going from the root toward its left child, viewing the edges of $T$ as being “walls” that we always keep to our left. Each node $v$ of $T$ is encountered three times by the Euler tour:

- “On the left” (before the Euler tour of $v$’s left subtree)
- “From below” (between the Euler tours of $v$’s two subtrees)
- “On the right” (after the Euler tour of $v$’s right subtree).
Euler Tour

Euler tour is defined as a way of traversing a tree such that each vertex is added to the tour when we visit it (either moving down from a parent vertex or returning from a child vertex). We start from the root and return back to the root after visiting all vertices.

It requires exactly \(2n - 1\) vertices to store Euler tour. [How to argue?]

Writing recursive routines is easy, will see. You try to write an iterative routine.

Balanced BSTs

Real-time systems are computational platforms that have real-time constraints, where computations must complete in a given amount of time, e.g., anti-lock braking systems on cars, video and audio processing systems, operating systems kernels, and web applications.

Most frequently used operations on stored data are random insertions, deletions, look-up, neighborhood search based on keys of the searched data records – BSTs perform those operations more efficiently than linear arrays (sorted on keys).

The problem is that without some way of limiting the height of \(T\), in the worst case \(h = \Theta(n)\); the worst case running time for performing searches and updates in \(T\) can be \(\Theta(n)\) in the number of items it stores. Indeed, this worst-case behavior occurs if we insert and delete keys in \(T\) in a somewhat sorted order, which is likely for a database. This results in poor performance for searches and updates.

One needs a way of restructuring a binary search tree while it is being used so that it can guarantee logarithmic-time performance for searches and updates. These restructuring methods [to be incorporated in update operations, e.g., insert and delete] result in a class of data structures known as balanced binary search trees.

There are different ways a BST can be balanced (to maintain a logarithmic height) – we may study some of those techniques, later if time permits.
**Ranks and Rotations**

The primary way to achieve logarithmic running times for search and update operations in a binary search tree, \( T \), is to perform restructuring actions on \( T \) based on specific rules that maintain some notion of “balance” between sibling subtrees in \( T \). Intuitively, the reason balance is so important is that when a binary search tree \( T \) is balanced, the size of the left and right subtrees of any node is “roughly” equal – that’s the goal.

We need a metric for “balance” – there are many (same in principle, but different in implementation) – we will assume the metric is (rank) height \( (h) \) of a tree (or a subtree) – we will augment the node structure as follows:

```c
typedef struct node {
    struct node *left;
    int key;
    int h;
    struct node *right;
} node;
```

We will say a tree (or a subtree) rooted at a node is balanced iff the difference in height of its left and right subtrees is \( \leq 1 \) (will show later that it will make the height of the tree always \( O(\log n) \). [There are many other ways to define rank of a node; all of them are related to height in some ways]. Such a tree is called an AVL tree [other examples are red-black trees, weak AVL trees, B-trees, B+ trees, etc.]

---

**More on Ranks**

Balance in such a tree, \( T \), is enforced by maintaining certain rules on the relative ranks of children and sibling nodes in \( T \). One restructuring operation, which is used in all balanced binary search trees called a rotation; there are 4 types of rotations.

We will use a unified restructuring operation, called trinode restructuring, which combines the four types of rotations into one action.

Note of caution about “rank” of a node in a BST: There are many different definitions of the word “rank” of a node in a BST; be careful.
How to know a BST is an AVL tree

A null BST is also an AVL tree; a non null BST is an AVL tree iff the difference between the heights of its left and right subtrees \( \leq 1 \). Easy to write a recursive routine: \( O(n) \) execution time (one needs to visit each node exactly twice starting from the root), much like any of the traversals; also easy to write non recursive version as well.

```c
bool isBalanced (node **lp) {node *root = *lp;
if (root == NULL) return 1;
    lh = ht(root->left); rh = ht(root->right);
    if (abs(lh – rh) <= 1 && isBalanced(root->left) &&
        isBalanced(root->right)) return 1;
return 0;
NEED a helper function to compute the height (ht) of a tree; easy. An empty tree has a height = 0; otherwise, height = 1 + max of left height and right height. Recursive program is straightforward. To write an iterative program, use level order traversal of the tree.