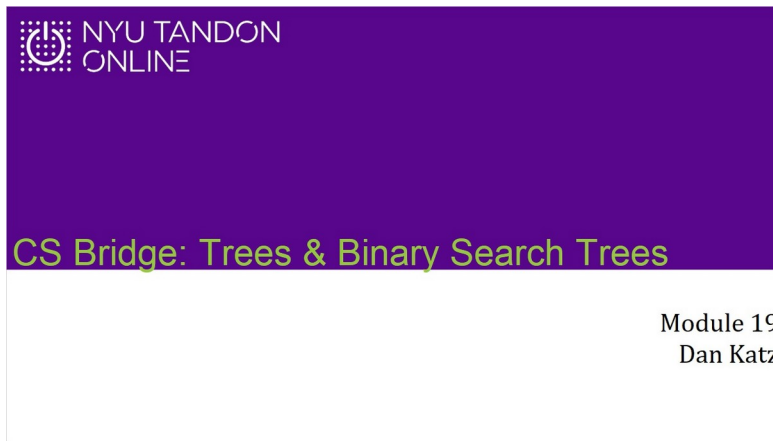


CS Bridge Module 19 Trees & Binary Search Trees

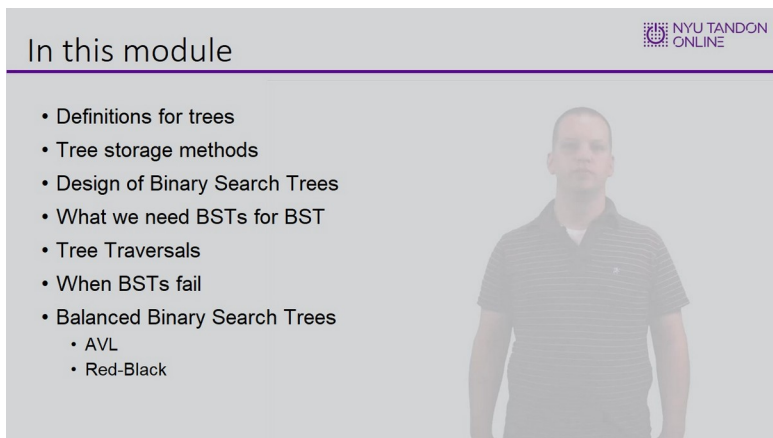
1. Trees and BST

1.1 CS Bridge: Trees & Binary Search Trees



Notes:


1.2 In this module



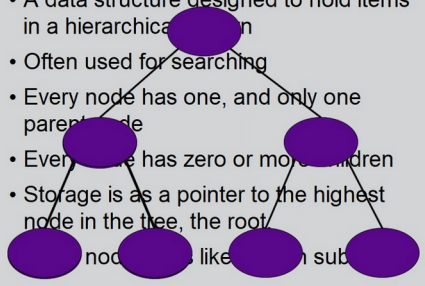
Notes:

1.3 What is a tree

What Is A Tree




- A data structure designed to hold items in a hierarchical fashion
- Often used for searching
- Every node has one, and only one parent node
- Every node has zero or more children
- Storage is as a pointer to the highest node in the tree, the root
- Nodes are like sub-nodes



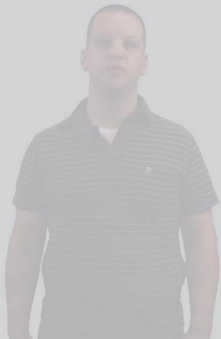
1.4 Some definitions

Some Definitions



Rollover for details

- Binary Tree
- Size
- Height
- Depth
- Leaf
- Full

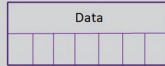


1.5 Tree Storage

Tree Storage



- For trees with unlimited children, child storage pointers is either in the form of an array or a linked list
- Parent –Multi-Child – The parent node has an array of child nodes



- Parent-Child-Sibling

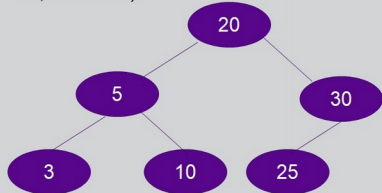


1.6 Binary Search Tree

Binary Search Tree



- In a binary tree, each node contains only a left and right pointer
- The BST is designed in such a way that the left child points to nodes whose data values are all less than the node, the right points to values greater than (or equal to, if allowed)



1.7 What we need BSTs for

What We Need BSTs For



- A BST provides, best case:
 - Search in $O(\log N)$
 - Insertion in $O(\log N)$
 - Deletion in $O(\log N)$
- We can use these for in order storage of any items which can be compared using a less-than operator
- Very efficient for in-order storage!

1.8 BST Node code

BST Node Code

NYU TANDON ONLINE

```
template <class T>
class BSTNode{
    T data;
    BSTNode<T>* parent;
    BSTNode<T>* left;
    BSTNode<T>* right;
public:
    BSTNode(T newdata = t(), BSTNode<T>* newParent = nullptr,
            BSTNode<T>* newLeft = nullptr, BSTNode<T>* newRight = nullptr) :
        data(newdata), parent(newParent), left(newLeft), right(newRight){}
    friend class BST<T>;
    int getSize() const;
};
```

1.9 Recursion in trees

Recursion In Trees

NYU TANDON ONLINE

```
template <class T>
int BSTNode<T>::getSize() const{
    int count = 1;
    if (left != nullptr)
        count += left->getSize();
    if (right != nullptr)
        count += right->getSize();
    return count;
}
```

1.10 Tree Traversals

Tree Traversals

NYU TANDON ONLINE

In Order

Pre Order

Post Order

Level Order

1.11 Knowledge Check

(Multiple Choice, 10 points, 1 attempt permitted)

Knowledge Check

NYU TANDON
ONLINE

What order are the nodes visited during an in-order traversal?

- ☒ Left nodes first, then root, then right nodes
- ☐ Root first, then left nodes, then right nodes
- ☐ Left nodes first, then right nodes, then root
- ☐ Nodes closest to the root first, then move to lower levels

Correct	Choice	Feedback
X	Left nodes first, then root, then right nodes	Correct!
	Root first, then left nodes, then right nodes	This is Pre-Order.
	Left nodes first, then right nodes, then root	This is Post Order.
	Nodes closest to the root first, then move to lower levels	This is Level Order.

Incorrect (Slide Layer)

Knowledge Check

NYU TANDON
ONLINE

What order are the nodes visited during an in-order traversal?

- ☒ Left nodes
- ☐ Root first,
- ☐ Left nodes
- ☐ Nodes clo

Correct

Correct!

Continue

Incorrect (Slide Layer)

Knowledge Check

NYU TANDON
ONLINE

What order are the nodes visited during an in-order traversal?

- ☒ Left nodes
- ☐ Root first,
- ☐ Left nodes
- ☐ Nodes clo

Incorrect

This is Pre-Order.

Continue

Incorrect (Slide Layer)

Knowledge Check

NYU TANDON
ONLINE

What order are the nodes visited during an in-order traversal?

- ☒ Left nodes
- ☐ Root first,
- ☐ Left nodes
- ☐ Nodes clo

Incorrect

This is Post Order.

Continue

Incorrect (Slide Layer)

Knowledge Check

What order are the nodes visited during an in-order traversal?

☒ Left nodes

☐ Root first,

☐ Left nodes

☐ Nodes clo

Incorrect

This is Level Order.

Continue

1.12 Implementation

Implementation

```
template <class T>
void BST<T>::printInOrder(BSTNode<T>* node) {
    if (node != nullptr) {
        printInOrder(node->left);
        cout << node->data << ", ";
        printInOrder(node->right);
    }
}

template <class T>
void BST<T>::printPreOrder(BSTNode<T>* node) {
    if (node != nullptr) {
        cout << node->data << ", ";
        printPreOrder(node->left);
        printPreOrder(node->right);
    }
}

template <class T>
void BST<T>::printPostOrder(BSTNode<T>* node) {
    if (node != nullptr) {
        printPostOrder(node->left);
        printPostOrder(node->right);
        cout << node->data << ", ";
    }
}
```

1.13 Level order traversal

Level Order Traversal

- Requires a queue to contain the node pointers to be processed
- Also known as a breadth first search

```
template <class T>
void BST<T>::printLevelOrder(){
    queue<BSTNode<T>*> q;
    q.push(root);
    while (!q.empty()){
        BSTNode<T>* temp = q.front();
        q.pop();
        cout << temp->data << ", ";
        if (temp->left != nullptr)
            q.push(temp->left);
        if (temp->right != nullptr)
            q.push(temp->right);
    }
}
```

1.14 Traversal results

Traversal Results

- Pre – 20,5,3,10,30,25
- In – 3,5,10,20,25,30
- Post – 3,10,5,25,30,20
- Level – 20,5,30,3,10,25

```
graph TD; 20((20)) --> 5((5)); 20 --> 30((30)); 5 --> 3((3)); 5 --> 10((10)); 30 --> 25((25));
```

NYU TANDON ONLINE

1.15 Insertion into trees

Insertion Into Trees

```
template <class T>
void BST<T>::insert(T item){
    if (root == nullptr){ //tree is empty
        root = new BSTNode<T>(item);
        return; }
    BSTNode<T>* temp = root;
    BSTNode<T>* prev = root;
    while (temp != nullptr){
        prev = temp;
        if (item < temp->data) //go left
            temp = temp->left;
        else
            temp = temp->right; }
    if (item < prev->data)
        prev->left = new BSTNode<T>(item, prev);
    else
        prev->right = new BSTNode<T>(item, prev);}
```

NYU TANDON ONLINE

1.16 Removal from a tree

Removal From A Tree

NYU TANDON ONLINE

1.17 Removal, given the node, no children

Removal, Given the Node, No Children



```
template <class T>
void BST<T>::remove(BSTNode<T>* & temp){
    if (temp->left == nullptr && temp->right == nullptr){ //no children
        //need to determine if this is the root, or the left/right of parent
        if (parent == nullptr) //last node on the tree
            root = nullptr;
        else if (parent->left == temp)
            parent->left = nullptr;
        else
            parent->right = nullptr;
        delete temp;
    }
}
```

1.18 Removal, given the node, one child

Removal, Given the Node, One Child



```
else if (temp->left == nullptr){ //We have a right but no left
    //Promote temp->right
    BSTNode<T>* toDelete = temp->right;
    temp->data = toDelete->data;
    temp->left = toDelete->left;
    if (temp->left != nullptr){
        temp->left->parent = temp;
    }
    temp->right = toDelete->right;
    if (temp->right != nullptr){
        temp->right->parent = temp;
    }
    delete toDelete;
}

else if (temp->right == nullptr){ //We have a left but no right
    //Promote temp->left
    BSTNode<T>* toDelete = temp->left;
    temp->data = toDelete->data;
    temp->left = toDelete->left;
    if (temp->left != nullptr){
        temp->left->parent = temp;
    }
    temp->right = toDelete->right;
    if (temp->right != nullptr){
        temp->right->parent = temp;
    }
    delete toDelete;
}
```

1.19 Removal, Given the node, two children

Removal, Given the Node, Two Children




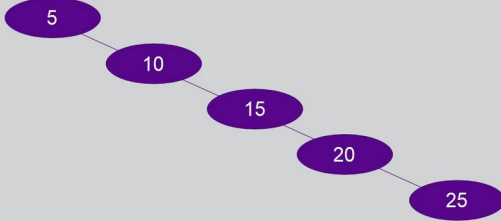
```
else{ //ugh... we have both... sigh, lets
make it easy and choose someone else to
delete!
    BSTNode<T>* minOfRight = temp->right;
    while (minOfRight->left != nullptr)
        minOfRight = minOfRight->left;
    temp->data = minOfRight->data;
    remove(minOfRight); //recursion! (but
not infinite because no left child)
}
```

1.20 When BSTs fail

When BSTs Fail

NYU TANDON
ONLINE

- In a best case scenario, BSTs result in $O(\log N)$ time for everything.
- Unfortunately, if insertions into the tree are already in order...




1.21 Balanced Binary Search Trees

Balanced Binary Search Trees

NYU TANDON
ONLINE

- If we need to guarantee $O(\log N)$ search time, we need to do some additional work during insertion and removal,
- The Balanced binary search tree protects the $O(\log N)$ insertion time
- Two popular types of balanced binary search trees are
 - AVL trees
 - Red-Black Trees




1.22 AVL Trees

AVL Trees

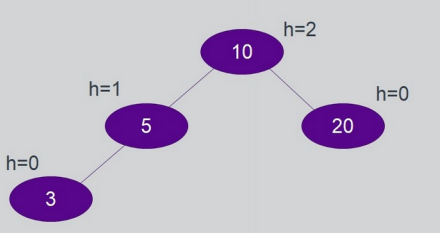
NYU TANDON
ONLINE

- Named for their creators, ..., an AVL tree is an easy balanced binary tree to understand
- Each node records its own height
- The AVL tree requires that the height of the left and right subtrees of every node differ by a height of no more than 1
- If the heights differ by more than one, a rotation is necessary to "rebalance" the subtree.



1.23 Good AVL trees

Good AVL Trees

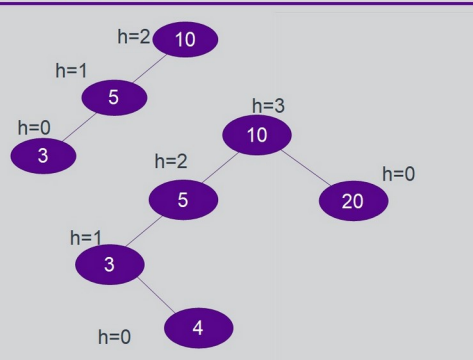


A diagram of a Good AVL Tree. The root node is 10 with a balance factor of h=2. Node 10 has a left child 5 (h=1) and a right child 20 (h=0). Node 5 has a left child 3 (h=0). The tree is balanced.

NYU TANDON ONLINE

1.24 Bad AVL trees

Bad AVL Trees



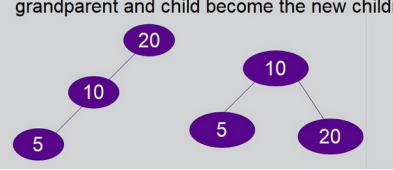
A diagram of a Bad AVL Tree. The root node is 10 with a balance factor of h=2. Node 10 has a left child 5 (h=1) and a right child 20 (h=0). Node 5 has a left child 3 (h=0). Node 3 has a right child 4 (h=0). The tree is unbalanced.

NYU TANDON ONLINE

1.25 Rotation solutions

Rotation Solutions

- Given a grandparent, parent, child situation which is unbalanced
- If the left parent's left subchild (or right parent's right subchild) is greater than the left parent's right subchild (or right parent's left subchild), **Single rotation** – make the parent the new grandparent and the grandparent and child become the new children



A diagram illustrating a Single rotation. On the left, a tree with root 20, left child 10, and left child of 10 is 5. On the right, a tree with root 10, left child 5, and right child 20. This represents a single rotation around node 10.

NYU TANDON ONLINE

1.26 Double Rotation

Double Rotation

NYU TANDON ONLINE

Perform two single rotation to produce a double rotation

```
graph TD; 20((20)) --> 10_1((10)); 10_1 --> 15_1((15)); 15_1 --> 10_2((10)); 15_1 --> 20_2((20))
```

Notes:

1.27 Red-Black Trees

Red-Black Trees

NYU TANDON ONLINE

The Laws:


1. All nodes are colored, red or black
2. The root is always black
3. A red node cannot have a red child
4. All paths from root to all children must pass through the same number of black nodes

```
graph TD; Root(( )) --- L(( )); Root --- R(( )); L --- L1(( )); L --- L2(( )); R --- R1(( ))
```


1.28 In this module, we learned

What We've Learned

- Definitions for trees
- Tree storage methods
- Design of Binary Search Trees
- What we need BSTs for
- BST Design
- Tree Traversals
- When BSTs fail
- Balanced Binary Search Trees
 - AVL
 - Red-Black



1.29 End of Module

 NYU TANDON
ONLINE

End of Module

Exit