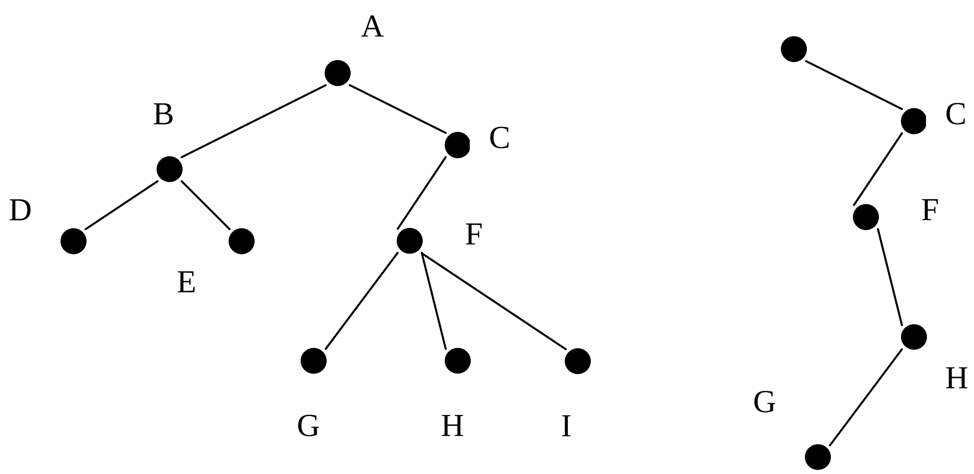
CSCI 2170 **Lecture Notes (15)**
Tree

Tree – hierarchical structure of organizing data
Tree terminologies:

- Parent
- Child
- Sibling
- Root : exactly one
- Leaf : no children
- Ancestor
- Descendant
- Degree of a node
- Subtree : any node in the tree together with all its descendants
- Subtree of a node n : subtree rooted at a child of n
- Level of a node: 1, 2, …
- Height of a tree/subtree: maximum level of any node in the tree

Show examples of each using the following tree:

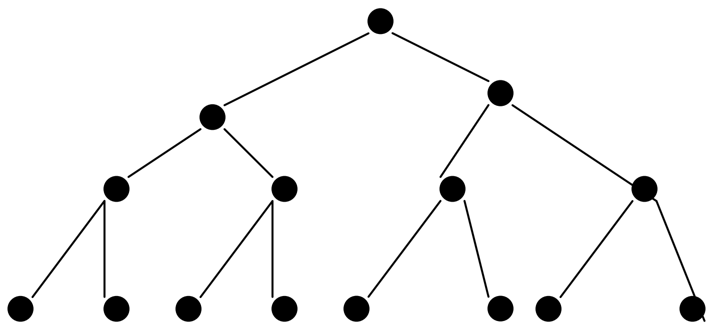


Binary tree – tree where each node has at most 2 child nodes
Full binary tree - a binary tree of height h, where all nodes at levels less than h have 2 children nodes

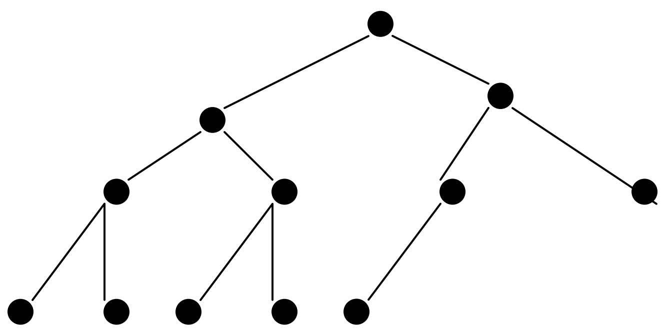
- How many nodes are in a full binary tree of height h?
  $2^0+2^1+2^2+2^3….2^{(h-1)}=2^h-1$
- Given a full binary tree with N nodes, what is the height of the tree?
  $2^h-1 = N \quad \Rightarrow \quad h = \log2(N+1)$

Complete binary tree : binary tree full down the level (h-1), with level h filled in from left to right

- All nodes at level >= h-2 have 2 children
- When a node at level h-1has children, all nodes to its left at the same level have 2 children each
- When a node at level h-1 has one child, it is a left child

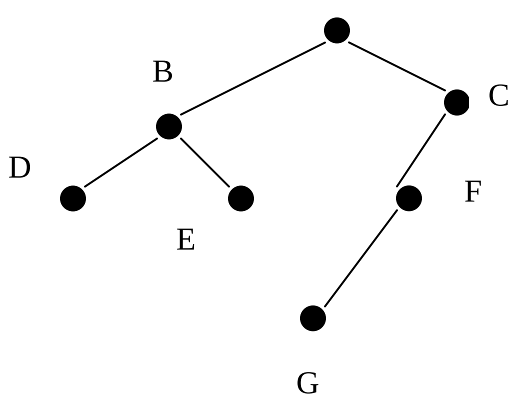


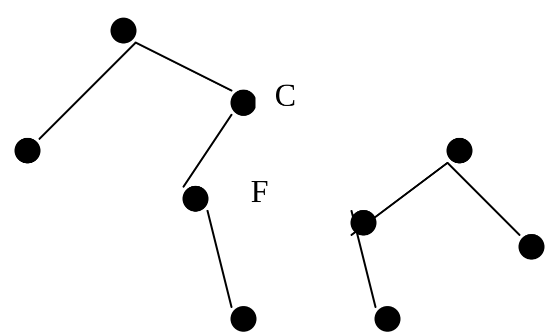
Full binary tree



Complete binary tree

Balanced binary tree: binary tree is balanced, if the height of any node's left subtree differs from the height of any node's right subtree by no more than 1

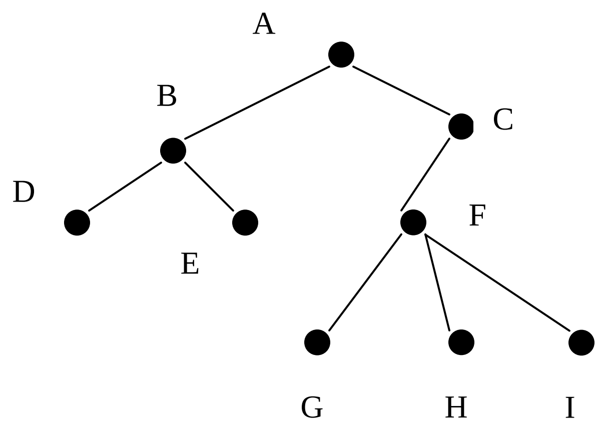




If a binary tree is complete, is it always balanced?   y
If a binary tree is full, is it always balanced? y
If a binary tree is balanced, is it always full? n

If a binary tree is balanced, is it always complete? N
Traversal of a binary tree
A traversal algorithm for a binary tree visits each node in the tree

- Preorder traversal : visit root first, than left child, than right child (ABDECFGHI)
- Inorder traversal : left → root → right (
- Postorder traversal : left → right → root




```
Preorder (T)
{       if (T is not empty)
        {       visit root;
                preorder (T_leftsubtree);
                preorder(T_rightsubtree);
        }
}

Inorder(T)
{       if (T is not empty)
        {       preorder (T_leftsubtree);
                visit root;
                preorder(T_rightsubtree);
        }
}

Postorder(T)
{       if (T is not empty)
        {       preorder (T_leftsubtree);
                preorder(T_rightsubtree);
                visit root;
        }
}
```

binary search tree - a binary tree with the ordering property: the **key data** in any node P is greater than the **key data** in any node in the left subtree of P and less than the **key data** in any node in the right subtree

Build a BST from scratch:

Insert record with key=3
Insert record with key=7
Insert record with key=4
Insert record with key=13
Insert record with key=43
Insert record with key=30
Insert record with key=8
Insert record with key=19
…

Main advantage of BST over linked list in organizing data is efficiency (search efficiently : visit as few nodes as possible before reaching the target record)

- If the tree is skewed, it does not improve efficiency, e.g., it is equivalent to linked list
- If the BST is full/complete BST, it has the smallest height, most efficient, require the minimum number of visits for any target record
- It is quite expensive to build the full/complete BST, but it is quite inexpensive (computational-wise) to build a BST that has a height that is close to minimum height. This type of BST is called the AVL tree.

Binary Search Tree (BST)

```
typedef string keyType;
class videoClass;
typedef videoClass  treeItemType;
typedef void (*functionType)(treeItemType& AnItem);

struct treeNode;  // defined in implementation file
typedef treeNode* ptrType;  // pointer to node

class bstClass
{
public:
   // constructors and destructor:
   bstClass();                      // default constructor
   bstClass(const bstClass& Tree);  // copy constructor
    ~bstClass();             // destructor

   // binary search tree operations:
   // Precondition for all methods: No two items in a binary
   // search tree have the same search key.

    bool SearchTreeIsEmpty() const;
   // Determines whether a binary search tree is empty.
   // Postcondition: Returns true if the tree is empty; otherwise returns false.

    void SearchTreeInsert(const treeItemType& NewItem, bool& Success);
   // Inserts an item into a binary search tree.
   // Precondition: The item to be inserted into the tree is NewItem.
   // Postcondition: If the insertion was successful, NewItem is in its proper order in the tree and
   // Success is true. Otherwise, the tree is unchanged and Success is false.

    void SearchTreeDelete(keyType SearchKey, bool& Success);
   // Deletes an item with a given search key from a binary search tree.
   // Precondition: SearchKey is the search key of the item to be deleted.
   // Postcondition: If the item whose search key equals
   // SearchKey existed in the tree, the item is deleted and Success is true. Otherwise, the tree is
   // unchanged and Success is false.

    void SearchTreeRetrieve(keyType SearchKey, treeItemType& TreeItem,
                              bool& Success) const;
   // Retrieves an item with a given search key from a binary search tree.
   // Precondition: SearchKey is the search key of the item to be retrieved.
   // Postcondition: If the retrieval was successful, TreeItem contains the retrieved item and
   // Success is true. If no such item exists, TreeItem and the tree are unchanged and Success is
false.

    void PreorderTraverse(functionType Visit);
   // Traverses a binary search tree in preorder, calling function Visit once for each item.
   // Precondition: The function represented by Visit exists outside of the class implementation.
```

```
   // Postcondition: Visit's action occurred once for each item in the tree.
   // Note: Visit can alter the tree.

    void InorderTraverse(functionType Visit);
   // Traverses a binary search tree in sorted order, calling function Visit once for each item.

   // Traverses a binary search tree in postorder, calling function Visit once for each item.
    void PostorderTraverse(functionType Visit);

   // overloaded operator:
    bstClass& operator=(const bstClass& Rhs);

private:
    void InsertItem(ptrType& TreePtr, const treeItemType& NewItem, bool& Success);
   // Recursively inserts an item into a binary search tree.
   // Precondition: TreePtr points to a binary search tree, NewItem is the item to be inserted.
   // Postcondition: Same as SearchTreeInsert.

   void DeleteItem(ptrType& TreePtr, keyType SearchKey, bool& Success);
   // Recursively deletes an item from a binary search tree.
   // Precondition: TreePtr points to a binary search tree, SearchKey is the search key of the item
to be deleted.
   // Postcondition: Same as SearchTreeDelete.

   void DeleteNodeItem(ptrType& NodePtr);
   // Deletes the item in the root of a given tree.
   // Precondition: RootPtr points to the root of a binary search tree; RootPtr != NULL.
   // Postcondition: The item in the root of the given tree is deleted.

   void ProcessLeftmost(ptrType& NodePtr, treeItemType& TreeItem);
   // Retrieves and deletes the leftmost descendant of a given node.
   // Precondition: NodePtr points to a node in a binary search tree; NodePtr != NULL.
   // Postcondition: TreeItem contains the item in the leftmost descendant of the node to which
   // NodePtr points. The leftmost descendant of NodePtr is deleted.

   void RetrieveItem(ptrType TreePtr, keyType SearchKey, treeItemType& TreeItem,  bool&
Success) const;
   // Recursively retrieves an item from a binary search tree.
   // Precondition: TreePtr points to a binary search tree,
   // SearchKey is the search key of the item to be retrieved.
   // Postcondition: Same as SearchTreeRetrieve.

   void CopyTree(ptrType TreePtr, ptrType& NewTreePtr) const;
   void DestroyTree(ptrType& TreePtr);

   void Preorder(ptrType TreePtr, functionType Visit);
   void Inorder(ptrType TreePtr, functionType Visit);
   void Postorder(ptrType TreePtr, functionType Visit);

   ptrType RootPtr() const;
   void SetRootPtr(ptrType NewRoot);
```

```
  void GetChildPtrs(ptrType NodePtr, ptrType& LChildPtr,  ptrType& RChildPtr) const;
  void SetChildPtrs(ptrType NodePtr, ptrType LChildPtr,  ptrType RChildPtr);

  ptrType Root;  // pointer to root of tree
};  // end class
// End of header file.

struct treeNode
{  treeItemType Item;
  ptrType      LChildPtr, RChildPtr;

  // constructor:
  treeNode(const treeItemType& NodeItem, ptrType L, ptrType R);
};  // end struct

treeNode::treeNode(const treeItemType& NodeItem, ptrType L, ptrType R): Item(NodeItem),
                         LChildPtr(L), RChildPtr(R)
{
}  // end constructor

bstClass::bstClass() : Root(NULL)
{
}  // end default constructor

bstClass::bstClass(const bstClass& Tree)
{
  CopyTree(Tree.Root, Root);
}  // end copy constructor

bstClass::~bstClass()
{
  DestroyTree(Root);
}  // end destructor

bool bstClass::SearchTreeIsEmpty() const
{
  return bool(Root == NULL);
}  // end SearchTreeIsEmpty

void bstClass::SearchTreeInsert(const treeItemType& NewItem,  bool& Success)
{
  InsertItem(Root, NewItem, Success);
}  // end SearchTreeInsert

void bstClass::SearchTreeDelete(keyType SearchKey,   bool& Success)
{
  DeleteItem(Root, SearchKey, Success);
}  // end SearchTreeDelete

void bstClass::SearchTreeRetrieve(keyType SearchKey,  treeItemType& TreeItem,
```

```
                              bool& Success) const
{
   RetrieveItem(Root, SearchKey, TreeItem, Success);
}  // end SearchTreeRetrieve

void bstClass::PreorderTraverse(functionType Visit)
{
   Preorder(Root, Visit);
}  // end PreorderTraverse

void bstClass::InorderTraverse(functionType Visit)
{
   Inorder(Root, Visit);
}  // end InorderTraverse

void bstClass::PostorderTraverse(functionType Visit)
{
   Postorder(Root, Visit);
}  // end PostorderTraverse

void bstClass::InsertItem(ptrType& TreePtr, const treeItemType& NewItem, bool& Success)
{
   if (TreePtr == NULL)
   {  // position of insertion found; insert after leaf

      // create a new node
      TreePtr = new treeNode(NewItem, NULL, NULL);

      // was allocation successful?
      Success = bool(TreePtr != NULL);
   }

   // else search for the insertion position
   else if (NewItem.Key() < TreePtr->Item.Key())
      // search the left subtree
      InsertItem(TreePtr->LChildPtr, NewItem, Success);

   else  // search the right subtree
      InsertItem(TreePtr->RChildPtr, NewItem, Success);
}  // end InsertItem

void bstClass::DeleteItem(ptrType& TreePtr, keyType SearchKey,  bool& Success)
// Calls: DeleteNodeItem.
{
   if (TreePtr == NULL)
      Success = false;  // empty tree

   else if (SearchKey == TreePtr->Item.Key())
   {  // item is in the root of some subtree
      DeleteNodeItem(TreePtr);  // delete the item
      Success = true;
```

```
   }  // end if in root

   // else search for the item
   else if (SearchKey < TreePtr->Item.Key())
      // search the left subtree
      DeleteItem(TreePtr->LChildPtr, SearchKey, Success);

   else  // search the right subtree
      DeleteItem(TreePtr->RChildPtr, SearchKey, Success);
}  // end DeleteItem

void bstClass::DeleteNodeItem(ptrType& NodePtr)
// Algorithm note: There are four cases to consider:
//   1. The root is a leaf.
//   2. The root has no left child.
//   3. The root has no right child.
//   4. The root has two children.
// Calls: ProcessLeftmost.
{
   ptrType      DelPtr;
   treeItemType ReplacementItem;

   // test for a leaf
   if ( (NodePtr->LChildPtr == NULL) && (NodePtr->RChildPtr == NULL) )
   {  delete NodePtr;
      NodePtr = NULL;
   }  // end if leaf

   // test for no left child
   else if (NodePtr->LChildPtr == NULL)
   {  DelPtr = NodePtr;
      NodePtr = NodePtr->RChildPtr;
      DelPtr->RChildPtr = NULL;
      delete DelPtr;
   }  // end if no left child

   // test for no right child
   else if (NodePtr->RChildPtr == NULL)
   {  DelPtr = NodePtr;
      NodePtr = NodePtr->LChildPtr;
      DelPtr->LChildPtr = NULL;
      delete DelPtr;
   }  // end if no right child

   // there are two children:
   // retrieve and delete the inorder successor
   else
   {  ProcessLeftmost(NodePtr->RChildPtr, ReplacementItem);
      NodePtr->Item = ReplacementItem;
   }  // end if two children
}  // end DeleteNodeItem
```

```
void bstClass::ProcessLeftmost(ptrType& NodePtr,  treeItemType& TreeItem)
{
   if (NodePtr->LChildPtr == NULL)
   {  TreeItem = NodePtr->Item;
      ptrType DelPtr = NodePtr;
      NodePtr = NodePtr->RChildPtr;
      DelPtr->RChildPtr = NULL;  // defense
      delete DelPtr;
   }

   else
      ProcessLeftmost(NodePtr->LChildPtr, TreeItem);
}  // end ProcessLeftmost

void bstClass::RetrieveItem(ptrType TreePtr, keyType SearchKey,  treeItemType& TreeItem,
                            bool& Success) const
{
   if (TreePtr == NULL)
      Success = false;  // empty tree

   else if (SearchKey == TreePtr->Item.Key())
   {  // item is in the root of some subtree
      TreeItem = TreePtr->Item;
      Success = true;
   }

   else if (SearchKey < TreePtr->Item.Key())
   // search the left subtree
      RetrieveItem(TreePtr->LChildPtr, SearchKey, TreeItem, Success);

   else  // search the right subtree
      RetrieveItem(TreePtr->RChildPtr, SearchKey, TreeItem, Success);
}  // end RetrieveItem


bstClass& bstClass::operator=(const bstClass& Rhs)
{
   if (this != &Rhs)
   {  DestroyTree(Root);  // deallocate left-hand side
      CopyTree(Rhs.Root, Root);  // copy right-hand side
   }  // end if
   return *this;
}  // end operator=

void bstClass::CopyTree(ptrType TreePtr, ptrType& NewTreePtr) const
{
   // preorder traversal
   if (TreePtr != NULL)
   {  // copy node
      NewTreePtr = new treeNode(TreePtr->Item, NULL, NULL);
```

```
      assert(NewTreePtr != NULL);

      CopyTree(TreePtr->LChildPtr, NewTreePtr->LChildPtr);
      CopyTree(TreePtr->RChildPtr, NewTreePtr->RChildPtr);
   }  // end if
   else
      NewTreePtr = NULL;  // copy empty tree
}  // end CopyTree

void bstClass::DestroyTree(ptrType& TreePtr)
{
   // postorder traversal
   if (TreePtr != NULL)
   {  DestroyTree(TreePtr->LChildPtr);
      DestroyTree(TreePtr->RChildPtr);
      delete TreePtr;
      TreePtr = NULL;
   }  // end if
}  // end DestroyTree

ptrType bstClass::RootPtr() const
{
   return Root;
}  // end RootPtr

void bstClass::SetRootPtr(ptrType NewRoot)
{
   Root = NewRoot;
}  // end SetRoot

void bstClass::GetChildPtrs(ptrType NodePtr, ptrType& LeftPtr, ptrType& RightPtr) const
{
   LeftPtr = NodePtr->LChildPtr;
   RightPtr = NodePtr->RChildPtr;
}  // end GetChildPtrs

void bstClass::SetChildPtrs(ptrType NodePtr, ptrType LeftPtr,  ptrType RightPtr)
{
   NodePtr->LChildPtr = LeftPtr;
   NodePtr->RChildPtr = RightPtr;
}  // end SetChildPtrs

void bstClass::Preorder(ptrType TreePtr, functionType Visit)
{
   if (TreePtr != NULL)
   {  Visit(TreePtr->Item);
      Preorder(TreePtr->LChildPtr, Visit);
      Preorder(TreePtr->RChildPtr, Visit);
   } // end if
}  // end Preorder
```

```
void bstClass::Inorder(ptrType TreePtr, functionType Visit)
{
   if (TreePtr != NULL)
   {  Inorder(TreePtr->LChildPtr, Visit);
      Visit(TreePtr->Item);
      Inorder(TreePtr->RChildPtr, Visit);
   } // end if
}  // end Inorder

void bstClass::Postorder(ptrType TreePtr, functionType Visit)
{
   if (TreePtr != NULL)
   {  Postorder(TreePtr->LChildPtr, Visit);
      Postorder(TreePtr->RChildPtr, Visit);
      Visit(TreePtr->Item);
   } // end if
}  // end Postorder
// End of implementation file.
```