Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #07: Trees & Binary Search Trees

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Familiarize with *Trees and Related Terminologies*

• Understand *different Traversing Schemes*

• Familiarize with *Binary Trees and Related Facts*

• Familiarize with *Binary Trees Operations*

• Familiarize with *Binary Search Trees and its ADT*

• Understand *BST operations find, insert and delete*

• Understand *How fast is the BST operations*

• Familiarize with *Applications of BST*
Trees
Trees

- Most of the data structures we have looked at so far have been devoted to keeping a *collection* of elements in some *linear order*.

- Trees are the **most common** non-linear data structure in computer science that are useful in representing things that naturally occur in **hierarchies**.

- Properly implemented, a tree can lead to an implementation that can be both searched and inserted into in \( O(\log N) \) time.
  
  - Compare this to the data structures we’ve seen so far, which may allow us to search in \( O(\log N) \) time but insert in \( O(N) \), or insert in \( O(1) \) but search in \( O(N) \).
Tree Terminology

- A tree is a collection of nodes that includes a designated node $r$, the root, and zero or more (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by an edge to $r$.

- The collection of nodes shown in the figure here is a tree. We can designate $A$ as the root, and we note that the collections of nodes $\{B\}, \{C, F\}, \{D\}$, and $\{E, G, H, I\}$, together with their edges, are all trees whose own roots are connected to $A$.

- Focusing on a tree as a collection of nodes leads to some other terminology:
  - Each node except the root has a parent.
  - Parent nodes have children. Nodes without children are leaves.
  - Nodes with the same parent are siblings.
Tree Terminology

• A tree in which every parent has at most 2 children is a *binary tree*.

• Trees in which parents may have more than 2 children are *general trees*. A tree where each node could have up to m children is called *m-ary tree*.

• A *path* from $n_1$ to $n_k$ is a sequence $n_1, n_2, \ldots, n_k$ such that
  \[ \forall i, 1 \leq i \leq k, n_i \text{ is the parent of } n_{i+1}. \]
  – The *length* of a path is the *number of edges* in it.
  – $n_1$ is an *ancestor* of $n_k$.
  – $n_k$ is a *descendant* of $n_1$.

• The *depth/level* of a node is the length of the path from the root to that node.
• The *height* of a node is the length of the longest path from it to any leaf.
  – The height of an empty tree is -1.
Tree Terminology

• The **degree** of a node is the number of subtrees of the node
  - The degree of node A is 4, while the degree of node E is 3.
  - The node with degree 0 is a leaf or terminal node such as nodes B and H

• The **ancestors** of a node are all the nodes along the path from the root to the node.

• The **size** of a tree is the number of nodes in it

• The **depth** of a tree is the depth of its deepest node

• The **height** of a tree is the maximum level that any of its node attains.
  - height(tree) = 1 + max (height(tree.left), height(tree.right))  *(binary tree)*
Tree Traversal

- **Traversing** a tree means to visit each node in the tree and process the data in that node.

- Kinds of Traversals
  - A **pre-order** traversal is one in which the data of each node is processed before visiting any of its children.
  - A **post-order** traversal is one in which the data of each node is processed after visiting all of its children.
  - An **in-order** traversal is one in which the data of each node is processed after visiting its left child but before visiting its right child.
    - This traversal is specific to binary trees.
  - A **level-order** traversal is one in which all nodes of the same height are visited before any lower nodes.
Tree Traversal

- Example: Compilers, interpreters, spreadsheets, and other programs that read and evaluate arithmetic expressions often represent those expressions as trees.
  - Constants and variables go in the leaves.
  - Each non-leaf node represents the application of some operator to the subtrees representing its operands.
- The tree here, for example, shows the product of a sum and of a subtraction.
  - Pre-order: \(* + 13 a - x 1\)
  - In-order: \(13 + a * x - 1 \rightarrow ((13+a)*(x-1))\)
  - Post-order: \(13 a + x 1 - * \rightarrow \text{postfix expression}\)
  - Level-order: \(* + - 13 a x 1\)
Tree Traversal

• We define a binary tree with nodes declared as shown here:
  • A node structure contains a field for data and pointers for up to two children.
  • If the node is a leaf, both the left and right pointers will be null.
  • If the node has only one child, either left or right will be null.

• This is the basic structure pre-, in-, and post-order traversal algorithms using recursion.
  • Function is invoked recursively on the left and right subtrees visiting every node in the tree.

```cpp
// represents a node in a binary tree
template<typename T>
class tnode {
public:
    // tnode is a class implementation structure. Making the data public simplifies building class functions
    T nodeValue;
    tnode<T> *left, *right;
    // default constructor. data not initialized
    tnode() {} 
    // initialize the data members
    tnode(const T& item, tnode<T> * lptr = nullptr, tnode<T> * rptr = nullptr) :
        nodeValue(item), left(lptr), right(rptr) {} }

// basicTraverse is a recursive function that traverses the tree
template<typename T>
void basicTraverse(tnode<T>* t) {
    if (t != 0) {
        basicTraverse(t->left);
        basicTraverse(t->right);
    }
} 
```
Tree Traversal

• Pre-Order Traversals
  • Process the node \textit{before} visiting its children

\begin{verbatim}
template <typename T>
void preorder(tnode<T> *t)
{
    // the recursive scan terminates on a empty subtree
    if (t != nullptr)
    {
        doSomethingWith (t->nodeValue);
        preorder(t->left); // descend left
        preorder(t->right); // descend right
    }
}
\end{verbatim}

• Post-Order Traversals
  • Process the node \textit{after} visiting its children

\begin{verbatim}
template <typename T>
void postorder(tnode<T> *t)
{
    // the recursive scan terminates on a empty subtree
    if (t != nullptr)
    {
        postorder(t->left); // descend left
        postorder(t->right); // descend right
        doSomethingWith (t->nodeValue);
    }
}
\end{verbatim}
Tree Traversal

- In-Order Traversals
  - Process the node *after* visiting its *left* descendants and *before* visiting its *right* descendants
  - In-order traversal only makes sense when applied to binary trees.

```cpp
template <typename T>
void inorder(tnode<T> *t)
{
    // the recursive scan terminates on a empty subtree
    if (t != nullptr)
    {
        preorder(t->left); // descend left
        doSomethingWith (t->nodeValue);
        preorder(t->right); // descend right
    }
}
```
Tree Traversal

- **Level-Order Traversals**
  - Visit the root, then all elements 1 level below the root, then two levels below the root, and so on.
  - Elements visited successively may not be related in the parent-child sense except for having the root as a common.
  - Use a queue to keep track of nodes at the next lower level that need to be visited.

```cpp
template <typename T>
void levelorder(tnode<T> *t)
{
    // store siblings of each node in a queue so that they are
    // visited in order at the next level of the tree
    queue<tnode<T> *> q;
    tnode<T> *p;
    // initialize the queue by inserting the root in the queue
    q.push(t);
    // continue the iterative process until the queue is empty
    while(!q.empty())
    {
        // delete front node from queue and output the node value
        p = q.front();
        q.pop();
        doSomethingWith(t->nodeValue);
        // if a left child exists, insert it in the queue
        if (p->left != nullptr)
            q.push(p->left);
        // if a right child exists, insert next to its sibling
        if(p->right != nullptr)
            q.push(p->right);
    }
}
```
Binary Trees
Binary Trees

• A binary tree is composed of zero or more nodes

• Each node contains:
  • A value (some sort of data item)
  • A reference or pointer to a left child (may be null), and
  • A reference or pointer to a right child (may be null)

• A binary tree may be empty (contain no nodes)

• If not empty, a binary tree has a root node
  • Every node in the binary tree is reachable from the root node by a unique path

• A node with no left child and no right child is called a leaf
  • In some binary trees, only the leaves contain a value
Left ≠ Right

• The following two binary trees are **different**:

![Binary Tree Diagram](image)

• In the first binary tree, node A has a left child but no right child; in the second, node A has a right child but no left child

• Put another way: Left and right are **not** relative terms
Complete versus Full Binary Tree

• A **full binary tree** is a tree in which every node other than the leaves has two children.

• A **perfect binary tree** is a full binary tree in which all leaves are at the same level.

• A **complete binary tree** is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.
Binary Tree Facts

• A **full binary tree** with *i* internal nodes has $2^i + 1$ total nodes.

• The number of leaves in a **full binary tree** with *i* internal nodes is $= (2^i + 1) - i = i + 1$.

• The number of nodes at level *L*, except the lowest level, in a **complete binary tree** is $2^L$.

• The total number of nodes *n* in a **perfect binary tree** is $\sum_{L=0}^{h} (2^L) = 1 + 2^1 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1$.

• The **height (or depth)** *h* of binary trees is $O(\log n)$
  • **Proof:** In the case of perfect trees,
    • The number of nodes *n* is: $n = 1 + 2 + 2^2 + 2^3 + \ldots + 2^h = 2^{h+1} - 1$
    • Therefore, $2^{h+1} = n + 1$, and thus, $h = \log_2(n+1) - 1$
    • Hence, $h = O(\log n)$
  
  • So, for full and complete binary trees, the height is proportional to $\log_2 n$
Binary Tree Facts

- If a complete binary tree with \( n \) nodes is represented sequentially, then for any node with index \( i \), \( 0 \leq i \leq n-1 \), we have:
  
  - parent\((i)\) is at \((i-1)/2\) if \( i \neq 0 \). If \( i = 0 \), \( i \) is at the root and has no parent.
  
  - leftChild\((i)\) is at \(2i+1\) if \( 2i+1 \leq n-1 \). If \( 2i+1 \geq n \), then \( i \) has no left child.
  
  - rightChild\((i)\) is at \(2i+2\) if \( 2i+2 \leq n-1 \). If \( 2i+2 \geq n \), then \( i \) has no right child.

- **Shortest binary tree** with \( n \) nodes packed into it (the bushiest tree) is of height \( h \), where \( h = \log_2(n+1) - 1 \)

- **Tallest binary tree** with \( n \) nodes would be the one of height \( h = n-1 \).

- The height of any other binary tree would be bounded between these two limits: \( \log_2(n+1) - 1 \leq \text{height(binary_tree)} \leq n-1 \)
Computing Leaf Count & Tree Depth

• Computing the leaf count
  • A node in a tree is a *leaf node* if it has no children.

• Computing the tree depth
  • Depth of tree is the deepest level in the tree
  • Depth(tree) = 1 + max(depth(left subtree), depth (right subtree))

```cpp
// return the depth of a binary tree
template <typename T>
int depth (tnode<T> *t)
{
  int depthLeft, depthRight, depthVal;
  if (t == NULL)  //empty tree (stopping condition)
    depthVal = -1;
  else {
    // find the depth of the left tree
    depthLeft = depth(t->left);
    // find the depth of the right tree
    depthRight = depth(t->right);
    // depth of the tree with root t is 1 + maximum
    // of the depths of two subtrees
    depthVal = 1 + (depthLeft > depthRigh ?
                   depthLeft : depthRihgt);
  }
  return depthVal;
}
```

```cpp
// accumulate the number of leaf nodes in count.
// assume that count initialized to 0
template <typename T>
void countLeaf (tnode<T> *t, int& count)
{
  if (t != NULL) {
    // check if t is a leaf node? If so,
    // increment count and return (stopping condition)
    if (t->left == NULL && t->right == NULL)
      count++;
    else{
      countLeaf(t->left, count);
      countLeaf(t->right, count);
    }
  }
}
```
Copying Binary Tree

- The function `copyTree()` takes an initial tree and creates a duplicate (clone) tree.
- It uses recursive postorder traverse scheme

```cpp
template <typename T>
Tnode<T> *copyTree(tnode<T> *t)
{
    // newNode points to new node the algorithm creates. newLeft and newRight point to newNode subtrees
    tnode<T> *newNode, *newRight, *newLeft;

    //stop the recursive scan when we arrive at empty tree
    if (t == NULL)   //empty tree (stopping condition)
        return NULL;
    // build the new tree from the bottom up by building the two subtrees and then building the parent.
    // At node t, make a copy of the left subtree and assign its root node pointer to newLeft.
    // Make a copy of the right subtree and assign its root node pointer to newRight.
    newLeft = copyTree(t->left);
    newRight = copyTree(t->right);

    // Create a new node and copy data of node t to it and the new created subtrees
    newNode = new tnode<T> (t->nodeValue, newLeft, newRight);
    return newNode;
}
```

• The function `copyTree()` takes an initial tree and creates a duplicate (clone) tree.
• It uses recursive postorder traverse scheme
Deleting Tree Nodes

- The function `deleteTree()` deletes all nodes of the tree.
- It uses recursive postorder traverse scheme.
- The function `clearTree()` calls `deleteTree()` to remove all the nodes, and then reinitialize root point to NULL.

```cpp
// traverse the nodes in the binary tree
// and delete each node
template <typename T>
void deleteTree(tnode<T> *t)
{
    // postorder scan. delete left and right
    // subtrees of t and then node t
    if (t != NULL)
    {
        deleteTree(t->left);
        deleteTree(t->right);
        delete t;
    }
}

// removes all the nodes in the tree by calling
// deleteTree(), and then assign the root point
// to NULL. This function enables the root to
// be reused for a new tree.
template <typename T>
void clearTree(tnode<T> * & t)
{
    deleteTree(t);
    t = NULL;
}
```
Common Binary Tree Operations

• Determine the height
• Determine the number of nodes
• Make a copy
• Determine if two binary trees are identical
• Display the binary tree
• Delete a tree
• If it is an expression tree, evaluate the expression
• If it is an expression tree, obtain the parenthesized form of the expression
Binary Search Tree
Binary Search Tree (BST)

• A binary tree $T$ is a binary search tree if for each node $n$ with children $T_L$ and $T_R$:
  • The value in $n$ is greater than the values in every node in $T_L$.
  • The value in $n$ is less than the values in every node in $T_R$.
  • Both $T_L$ and $T_R$ are binary search trees.
The Binary Search Tree ADT

- The following code defines the ADT for Binary Search Tree class
  - The `stnode` template implements individual tree nodes.
  - The `stree` template represents the entire tree with functions for searching, insertion, iteration, etc.

Our primary focus in this lecture will be on the `find`, `insert` and `erase` functions.
The Binary Search Tree ADT

template<typename T>
class stree
{
   Public:
      typedef stree_iterator<T> iterator;
      typedef stree_cons t_iterator<T> cons t_iterator;

      stree(); // constructor. initialize root to NULL and size to 0

      stree(T *first, T *last); // constructor. insert the elements from the pointer
                                 // range [first, last) into the tree

      stree(const stree<T>& tree); // copy constructor

      ~stree(); // destructor

      stree<T>& operator= (const stree<T>& rhs); // assignment operator

      // search for item. if found, return an iterator pointing
      // at it in the tree; otherwise, return end()
      iterator find(const T& item);
       // constant version
      const_iterator find(const T& item) const

      int empty() const; // indicate whether the tree is empty

      int size() const; // return the number of data items in the tree

      // if item is not in the tree, insert it and return a pair whose iterator component points
      // at item and whose bool component is true. if item is in the tree, return a pair whose iterator
      // component points at the existing item and whose bool component is false
      // Postcondition: the tree size increases by 1 if item is not in the tree
      pair<iterator, bool> insert(const T& item);
The Binary Search Tree ADT

```c++
int erase(const T& item);
// if item is in the tree, erase it and return 1; otherwise, return 0
// Postcondition: the tree size decreases by 1 if item is in the tree

void erase(iterator pos);
// erase the item pointed to by pos.
// Preconditions: the tree is not empty and pos points to an item in the tree. if the tree is empty, the
// function throws the underflowError exception. if the iterator is invalid, the function throws the
// referenceError exception.
// Postcondition: the tree size decreases by 1

void erase(iterator first, iterator last);
// erase all items in the range [first, last).
// Precondition: the tree is not empty. if the tree is empty, the function throws the underflowError
// exception.
// Postcondition: the size of the tree decreases by the number of elements in the range [first, last)

// return an iterator pointing to the first item inorder
iterator begin();
// constant version
const_iterator begin() const;

// return an iterator pointing just past the end of the tree data
iterator end();
// constant version
const_iterator end() const;
```
The Binary Search Tree ADT

private:
    stnode<T> *root; // pointer to tree root
    int treeSize; // number of elements in the tree
    stnode<T> *getSTNode(const T& item,
                         stnode<T> *lptr, stnode<T> *rptr, stnode<T> *pptr);
    // allocate a new tree node and return a pointer to it.
    // if memory allocation fails, the function throws the memoryAllocationError exception

    stnode<T> *copyTree(stnode<T> *t);
    // recursive function used by copy constructor and assignment
    // operator to assign the current tree as a copy of another tree

    void deleteTree(stnode<T> *t);
    // recursive function used by destructor and assignment operator to delete all the nodes in the tree

    stnode<T> *findNode(const T& item) const;
    // search for item in the tree. if it is in the tree, return a pointer to its node; otherwise, return NULL.
    // used by find() and erase()
Binary Search Tree Iterator

- Here's the basic declaration for a BST iterator
- You will note that the code shown here is pretty much a standard iterator implemented by a standard pointer.

```cpp
class stree_iterator
{
    friend class stree<T>;
    friend class const_stree_iterator;

public:
    stree_iterator (){} // constructor
    bool operator==(const stree_iterator& rhs) const; // comparison operators. just compare node pointers
    bool operator!=(const stree_iterator& rhs) const;
    T& operator* () const; // dereference operator. return a reference to the value pointed to by nodePtr
    stree_iterator& operator++ (); // preincrement. move forward to next larger value
    stree_iterator operator++ (int); // postincrement
    stree_iterator& operator-- (); // predecrement. move backward to largest value < current value
    iterator operator-- (int); // postdecrement

private:
    // nodePtr is the current location in the tree. we can move freely about the tree using left, right, and parent.
    // tree is the address of the stree object associated with this iterator. it is used only to access the
    // root pointer, which is needed for ++ and -- when the iterator value is end()
    stnode<T> *nodePtr;
    stree<T> *tree;
    // used to construct an iterator return value from an stnode pointer
    iterator (stnode<T> *p, stree<T> *t) : nodePtr(p), tree(t) {}
};
```
// return an iterator pointing to the first item inorder
template <typename T>
type stree<T>::iterator stree<T>::begin()
{
    stnode<T> *curr = root;
    // if the tree is not empty, the first node
    // inorder is the farthest node left from root
    if (curr != NULL)
        while (curr->left != NULL)
            curr = curr->left;
    // build return value using private constructor
    return iterator(curr, this);
}

// return an iterator pointing just past the end of
// the tree data
template <typename T>
type stree<T>::iterator stree<T>::end()
{
    // end indicated by an iterator with NULL
    stnode pointer
    return iterator(NULL, this);
}

Binary Search Tree Iterator

• **begin()** position: by starting from the root and working our way down, always taking left children, until we come to a node with no left child.

• **end()** position: returning a null pointer.

• **operator++()**
  - If the current node has a non-null right child,
    - take a step down to the right
    - then run down to the left as far as possible
  - If the current node has a null right child,
    - move up the tree until we have moved over a *left child link*
Implementing BST Node

- As we will see next when discuss Iterators, you can't implement some iterators functions (e.g., operator++) simply with just a pointer to the node and all the nodes pointing only to their children.
- In a binary tree, to do \texttt{operator++}, we need to know not only where we are, but also \textit{how we got here}.
- One way is to do that is to implement the iterator as a stack of pointers containing the path to the current node \rightarrow too much
- A simpler approach is to add pointers from each node to its \textit{parent}. 
Searching Binary Search Trees

- We search a tree by comparing the value we’re searching for to the current node, \( t \). If the value we want is \textit{smaller}, we \textit{look in the left subtree}. If the value we want is \textit{larger}, we \textit{look in the right subtree}.

<table>
<thead>
<tr>
<th>Current Node</th>
<th>Action</th>
<th>(LOCATING DATA 37 IN A TREE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root = 50</td>
<td>Compare item = 37 and 50</td>
<td>37 &lt; 50, move to the left subtree</td>
</tr>
<tr>
<td>Node = 30</td>
<td>Compare item = 37 and 30</td>
<td>37 &gt; 30, move to the right subtree</td>
</tr>
<tr>
<td>Node = 35</td>
<td>Compare item = 37 and 35</td>
<td>37 &gt; 35, move to the right subtree</td>
</tr>
<tr>
<td>Node = 37</td>
<td>Compare item = 37 and 37</td>
<td>Item found.</td>
</tr>
</tbody>
</table>

![Binary Search Tree Diagram]

- We search a tree by comparing the value we’re searching for to the current node, \( t \). If the value we want is \textit{smaller}, we \textit{look in the left subtree}. If the value we want is \textit{larger}, we \textit{look in the right subtree}. 

![Binary Search Tree Diagram]

- We search a tree by comparing the value we’re searching for to the current node, \( t \). If the value we want is \textit{smaller}, we \textit{look in the left subtree}. If the value we want is \textit{larger}, we \textit{look in the right subtree}. 

![Binary Search Tree Diagram]
Searching Binary Search Trees

• The tree’s `find` operation uses a private utility function, `findNode`, to find a pointer to the node containing the desired data and then uses that pointer to construct an iterator representing the position of that node.

```cpp
// search for item. if found, return an iterator pointing // at it in the tree; otherwise, return end()
template <typename T>
type stree<T>::iterator stree<T>::find(const T& item) {
    stnode<T> *curr;
    // search tree for item
    curr = findNode(item);
    // if item found, return const_iterator with value current; // otherwise, return end()
    if (curr != NULL)
        return iterator(curr, this);
    else
        return end();
}
```

```cpp
// search for data item in the tree. if found, return its node // address; otherwise, return NULL
template <typename T>
stnode<T> *stree<T>::findNode(const T& item) const {
    // cycle t through the tree starting with root
    stnode<T> *t = root;

    // terminate on on empty subtree
    while(t != NULL && !(item == t->nodeValue))
        if (item < t->nodeValue)
            t = t->left;
        else
            t = t->right;

    // return pointer to node; NULL if not found
    return t;
}
```
Inserting into Binary Search Trees

• Always insert new node as leaf node, Start at root node as current node
  • If new node’s key < current’s key
    • If current node has a left child, recursively search left subtree to insert new node
    • Else add new node as current’s left child
  • If new node’s key > current’s key
    • If current node has a right child, recursively search right subtree to insert new node
    • Else add new node as current’s right child

Example: insert 60 in the tree:
1. start at the root, 60 is greater than 25, search in right subtree
2. 60 is greater than 50, search in 50’s right subtree
3. 60 is less than 70, search in 70’s left subtree
4. 60 is less than 66, add 60 as 66’s left child
The first part of the insertion function is closely related to the recursive form of the search. In fact, we are searching for the place where the new data would reside. We know we have not found it when we reach a null pointer (as either the left or right child of some parent node). New data is inserted at that null pointer position.

```cpp
template <typename T>
std::pair<typename stree<T>::iterator, bool> stree<T>::insert(const T& item) {
    // t is current node in traversal, parent the previous node
    stnode<T> *t = root, *parent = NULL, *newNode;
    while(t != NULL) {
        // terminate on on empty subtree
        parent = t;   // update the parent pointer. then go left or right
        // if a match occurs, return a pair whose iterator component points at item
        // in the tree and whose bool component is false
        if (item == t->nodeValue)
            return pair<iterator, bool> (iterator(t, this), false);
        else if (item < t->nodeValue)
            t = t->left;
        else
            t = t->right;
    }
    // create the new leaf node
    newNode = getSTNode(item,NULL,NULL,parent);
    if (parent == NULL)  // if parent is NULL, insert as root node
        root = newNode;
    else if (item < parent->nodeValue)
        parent->left = newNode; // insert as left child
    else
        parent->right = newNode; // insert as right child
    treeSize++;  // increment size
    return pair<iterator, bool> (iterator(newNode, this), true);
}
```
Deleting from Binary Search Trees

• Deletion is perhaps the most complex operation on a BST, because *the algorithm must result in a BST.*
  • The question is: *what value should replace the one deleted?*

• We have three main cases:
  1. Removing a *leaf node*: is trivial, just set the relevant child pointer in the parent node to NULL.
  2. Removing an *internal node with only one subtree*: is also trivial, just set the relevant child pointer in the parent node to target the root of the subtree.
Deleting from Binary Search Trees

3. Removing an internal node which has two subtrees: is more complex.

- To preserve the BST property, we must take the smallest value from the right subtree, which would be the closest successor of the value being deleted.
- Fortunately, the smallest value will always lie in the left-most node of the subtree.
Deleting from Binary Search Trees

3. Removing an *internal node which has two subtrees*: (cont’d)

- So, we first find the left-most node of the right subtree, and then replace the data of targeted node with the data of this node.

- Now we must delete the new node found from the right subtree.

- That looks straightforward here since the node in question is a leaf. However…
  - the node will NOT be a leaf in all cases
template <typename T>
void stree<T>::erase(iterator pos)
{
    // dNodePtr = pointer to node D that is deleted
    // pNodePtr = pointer to parent P of node D
    // rNodePtr = pointer to node R that replaces D
    stnode<T> *dNodePtr = pos.nodePtr, *pNodePtr, *rNodePtr;

    if (treeSize == 0)
        throw underflowError("stree erase(): tree is empty");

    if (dNodePtr == NULL)
        throw referenceError("stree erase(): invalid iterator");

    // assign pNodePtr the address of P
    pNodePtr = dNodePtr->parent;

    // If D has a NULL pointer, the replacement node is the other child
    if (dNodePtr->left == NULL || dNodePtr->right == NULL) {
        if (dNodePtr->right == NULL)
            rNodePtr = dNodePtr->left;
        else
            rNodePtr = dNodePtr->right;

        if (rNodePtr != NULL)
            // the parent of R is now the parent of D
            rNodePtr->parent = pNodePtr;
    }
    else { // both pointers of dNodePtr are non-NULL.
        // find and unlink replacement node for D.
        // starting at the right child of node D,
        // find the node whose value is the smallest of all
        // nodes whose values are greater than the value in D.
        // unlink the node from the tree.

        // pOfRNodePtr = pointer to parent of replacement node
        stnode<T> *pOfRNodePtr = dNodePtr;

        // first possible replacement is right child of D
        rNodePtr = dNodePtr->right;

        // descend down left subtree of the right child of D,
        // keeping a record of current node and its parent.
        // when we stop, we have found the replacement
        while(rNodePtr->left != NULL) {
            pOfRNodePtr = rNodePtr;
            rNodePtr = rNodePtr->left;
        }

        if (pOfRNodePtr == dNodePtr) {
            // right child of deleted node is the replacement.
            // assign left subtree of D to left subtree of R
            rNodePtr->left = dNodePtr->left;
            // assign the parent of D as the parent of R
            rNodePtr->parent = pNodePtr;
            // assign the left child of D to have parent R
            dNodePtr->left->parent = rNodePtr;
        }
        else {
            // both pointers of dNodePtr are non-NULL.
Deleting from Binary Search Trees

else {
    // we moved at least one node down a left branch of the
    // right child of D. unlink R from tree by assigning its
    // right subtree as the left child of the parent of R
    pOfRNodePtr->left = rNodePtr->right;

    // the parent of the right child of R is the parent of R
    if (rNodePtr->right != NULL)
        rNodePtr->right->parent = pOfRNodePtr;

    // put replacement node in place of dNodePtr
    // assign children of R to be those of D
    rNodePtr->left = dNodePtr->left;
    rNodePtr->right = dNodePtr->right;
    // assign the parent of R to be the parent of D
    rNodePtr->parent = pNodePtr;
    // assign the parent pointer in the children of R to point at R
    rNodePtr->left->parent = rNodePtr;
    rNodePtr->right->parent = rNodePtr;
}

// complete the link to the parent node.

// deleting the root node. assign new root
if (pNodePtr == NULL)
    root = rNodePtr;
    // attach R to the correct branch of P
else if (dNodePtr->nodeValue < pNodePtr->nodeValue)
    pNodePtr->left = rNodePtr;
else
    pNodePtr->right = rNodePtr;
    // delete the node from memory and decrement tree size
    delete dNodePtr;
    treeSize--;
}
Deleting from Binary Search Trees

Example of Case 1: Leaf Node

→ **Delete Node 10**

Before

![Before tree image](image1)

After

![After tree image](image2)

Delete leaf node 10.
pNodePtr->left is dNode

No replacement is necessary.
pNodePtr->left is NULL
Example of Case 2: Internal Node with Single Child
→ **Delete Node 35**

Delete node 35 with only a left child:
Node R is the left child.

Attach node R to the parent.
Deleting from Binary Search Trees

Example of Case 2: Internal Node with Single Child

→ **Delete Node 26**

**Delete node 26 with only a right child:**
Node R is the right child.

**Attach node R to the parent.**
Deleting from Binary Search Trees

Example of Case 3: Internal Node with Both Children
→ **Delete Node 25**

![Diagram of deleting node 25 from a binary search tree.](image-url)
How Fast Are Binary Search Tree?

• The number of recursive calls/loop iterations in all these algorithms is therefore no greater than the height of the tree.
• But how high can a BST be?
• **Balanced BST** is a binary tree is balanced if for every interior node, the height of its two children differ by at most 1.
• The **worst case** is when the *data being inserted is already in order* (or in reverse order). In that case, the tree *degenerates* into a sorted linked list, as shown above.
• The **best case** is when the *tree is balanced*, for each node, the heights of the node’s children are nearly the same.

• Complexity of *findNode()* operation
  • For the **best case** is O(log n)
  • For **worst case** is O(n)
How Fast Are Binary Search Tree?

- So the question is, does the "average" binary tree look more like the balanced or the degenerate case?
- An intuitive argument is:
  - No tree with $n$ nodes has height $< \log(n)$
  - No tree with $n$ nodes has height $> n$
  - Average depth of all nodes is therefore bounded between $n/2$ and $(\log n)/2$.
- The more unbalanced a tree is, the less likely that a random insertion would increase the tree height.
- Insertions that don’t increase the tree height make the tree more balanced.
- The more unbalanced a tree is, the more likely that a random insertion will actually tend to increase the balance of the tree.
  - This suggests (but does not prove) that randomly constructed binary search trees tend to be reasonably balanced.
Applications of Binary Search Tree

- Removing duplicate values from a vector
The Binary Search Tree ADT

using namespace std;

template <typename T>
class stree
{
Public:
    typedef stree_iterator<T> iterator;
    typedef stree_cons_t_iterator<T> cons_t_iterator;

    stree(); // constructor. initialize root to NULL and size to 0
    stree(T *first, T *last); // constructor. insert the elements from the pointer
        // range [first, last) into the tree
    stree(const stree<T>& tree); // copy constructor
    ~stree(); // destructor
    stree<T>& operator= (const stree<T>& rhs); // assignment operator

    // search for item. if found, return an iterator pointing
    // at it in the tree; otherwise, return end()
    iterator find(const T& item)
    {
        stnode<T> *curr;
        // search tree for item
        curr = findNode(item);
        // if item found, return const_iterator with value current;
        // otherwise, return end()
        if (curr != NULL)
            return iterator(curr, this);
        else
            return end();
    }
};
The Binary Search Tree ADT

// constant version
const_iterator find(const T& item) const
{
    stnode<T>* curr;
    // search tree for item
    curr = findNode(item);
    // if item found, return const_iterator with value current;
    // otherwise, return end()
    if (curr != NULL)
        return const_iterator(curr, this);
    else
        return end();
}

int empty() const;  // indicate whether the tree is empty
int size() const;   // return the number of data items in the tree

// if item is not in the tree, insert it and return a pair whose iterator component points
// at item and whose bool component is true. if item is in the tree, return a pair whose iterator
// component points at the existing item and whose bool component is false
// Postcondition: the tree size increases by 1 if item is not in the tree
pair<iterator, bool> insert(const T& item)
{
    // t is current node in traversal, parent the previous node
    stnode<T>* t = root, *parent = NULL, *newNode;
    // terminate on on empty subtree
    while(t != NULL)
    {

The Binary Search Tree ADT

// update the parent pointer. then go left or right
parent = t;
// if a match occurs, return a pair whose iterator
// component points at item in the tree and whose
// bool component is false
if (item == t->nodeValue)
    return pair<iterator, bool>(iterator(t, this), false);
else if (item < t->nodeValue)
    t = t->left;
else
    t = t->right;
}

// create the new leaf node
newNode = getSTNode(item, NULL, NULL, parent);

// if parent is NULL, insert as root node
if (parent == NULL)
    root = newNode;
else if (item < parent->nodeValue)
    // insert as left child
    parent->left = newNode;
else
    // insert as right child
    parent->right = newNode;

// increment size
treeSize++;

The Binary Search Tree ADT

```cpp
// return an pair whose iterator component points at
// the new node and whose bool component is true
return pair<iterator, bool> (iterator(newNode, this), true);
}

int erase(const T& item);
    // if item is in the tree, erase it and return 1;
    // otherwise, return 0
    // Postcondition: the tree size decreases by 1 if
    // item is in the tree

void erase(iterator pos);
    // erase the item pointed to by pos.
    // Preconditions: the tree is not empty and pos points
    // to an item in the tree. if the tree is empty, the
    // function throws the underflowError exception. if the
    // iterator is invalid, the function throws the referenceError
    // exception.
    // Postcondition: the tree size decreases by 1

void erase(iterator first, iterator last);
    // erase all items in the range [first, last).
    // Precondition: the tree is not empty. if the tree
    // is empty, the function throws the underflowError
    // exception.
    // Postcondition: the size of the tree decreases by
    // the number of elements in the range [first, last)
```
The Binary Search Tree ADT

// return an iterator pointing to the first item inorder
iterator begin()
{
    stnode<T> *curr = root;

    // if the tree is not empty, the first node
    // inorder is the farthest node left from root
    if (curr != NULL)
        while (curr->left != NULL)
            curr = curr->left;

    // build return value using private constructor
    return iterator(curr, this);
}

// constant version
const_iterator begin() const
{
    const stnode<T> *curr = root;

    // if the tree is not empty, the first node
    // inorder is the farthest node left from root
    if (curr != NULL)
        while (curr->left != NULL)
            curr = curr->left;

    // build return value using private constructor
    return const_iterator(curr, this);
}
The Binary Search Tree ADT

// return an iterator pointing just past the end of the tree data
iterator end()
{
    // end indicated by an iterator with NULL stnode pointer
    return iterator(NULL, this);
}

// constant version
const_iterator end() const
{
    // end indicated by an iterator with NULL stnode pointer
    return const_iterator(NULL, this);
}

void displayTree(int maxCharacters);
    // tree display function. maxCharacters is the largest
    // number of characters required to draw the value of a node

private:

    stnode<T> *root;
        // pointer to tree root

    int treeSize;
        // number of elements in the tree

    stnode<T> *getSTNode(const T& item,
        stnode<T> *lptr, stnode<T> *rptr),

    stnode<T> *lptr, stnode<T> *rptr, stnode<T> *pptr);
        // allocate a new tree node and return a pointer to it. If memory
        // allocation fails, the function throws the memoryAllocationError exception
The Binary Search Tree ADT

```
stnode<T> *copyTree(stnode<T> *t);
    // recursive function used by copy constructor and assignment
    // operator to assign the current tree as a copy of another tree

void deleteTree(stnode<T> *t);
    // recursive function used by destructor and assignment
    // operator to delete all the nodes in the tree

stnode<T> *findNode(const T& item) const;
    // search for item in the tree. if it is in the tree,
    // return a pointer to its node; otherwise, return NULL.
    // used by find() and erase()

tnodeShadow *buildShadowTree(stnode<T> *t, int level, int& column);
    // recursive function that builds a subtree of the shadow tree
    // corresponding to node t of the tree we are drawing. level is the
    // level-coordinate for the root of the subtree, and column is the
    // changing column-coordinate of the tree nodes

void deleteShadowTree(tnodeShadow *t);
    // remove the shadow tree from memory after displayTree()
    // displays the binary search tree
```

```
template <typename T>
stnode<T> *stree<T>::getSTNode(const T& item, stnode<T> *lptr, stnode<T> *rptr, stnode<T> *pptr)
{
    stnode<T> *newNode;

    newNode = new stnode<T> (item, lptr, rptr, pptr); // initialize the data and all pointers
    if (newNode == NULL)
        throw memoryAllocationError("stree: memory allocation failure");
    return newNode;
}

template <typename T>
stnode<T> *stree<T>::copyTree(stnode<T> *t)
{
    stnode<T> *newlptr, *newrptr, *newNode;

    if (t == NULL) // if tree branch NULL, return NULL
        return NULL;
    newlptr = copyTree(t->left); // copy the left branch of root t and assign its root to newlptr
    newrptr = copyTree(t->right); // copy the right branch of tree t and assign its root to newrptr
    // allocate storage for the current root node, assign its value and pointers to its left and right subtrees.
    // the parent pointer of newNode is assigned when newNode's parent is created. if newNode is root,
    // NULL is the correct value for its parent pointer
    newNode = getSTNode(t->nodeValue, newlptr, newrptr, NULL);
    if (newlptr != NULL) // the current node is the parent of any subtree that is not empty
        newlptr->parent = newNode;
    if (newrptr != NULL)
        newrptr->parent = newNode;
    return newNode;
}
The Binary Search Tree ADT

// delete the tree stored by the current object
template <typename T>
void stree<T>::deleteTree(stnode<T> *t)
{
    // if current root node is not NULL, delete its left subtree,
    // its right subtree and then the node itself
    if (t != NULL)
    {
        deleteTree(t->left);
        deleteTree(t->right);
        delete t;
    }
}

// search for data item in the tree. If found, return its node
// address; otherwise, return NULL
template <typename T>
stnode<T> *stree<T>::findNode(const T& item) const
{
    // cycle t through the tree starting with root
    stnode<T> *t = root;
    // terminate on on empty subtree
    while(t != NULL && !(item == t->nodeValue))
    {
        if (item < t->nodeValue)
            t = t->left;
        else
            t = t->right;
    }
    // return pointer to node; NULL if not found
    return t;
}
The Binary Search Tree ADT

template <typename T>
stree<T>::stree(): root(NULL),treeSize(0) {}

template <typename T>
stree<T>::stree(T *first, T *last): root(NULL),treeSize(0)
{
    T *p = first;
    while (p != last)  // insert each item in [first, last) into the tree
    {
        insert(*p);
        p++;
    }
}

template <typename T>
stree<T>::stree(const stree<T>& tree): treeSize(tree.treeSize)
{
    root = copyTree(tree.root);  // copy tree to the current object
}

template <typename T>
stree<T>::~stree()
{
    deleteTree(root);  // erase the tree nodes from memory

    root = NULL;  // tree is empty
treeSize = 0;
}
The Binary Search Tree ADT

template <typename T>
stree<T>& stree<T>::operator= (const stree<T>& rhs)
{
    // can't copy a tree to itself
    if (this == &rhs)
        return *this;
    // erase the existing tree nodes from memory
    deleteTree(root);
    // copy tree rhs into current object
    root = copyTree(rhs.root);
    // set the tree size
    treeSize = rhs.treeSize;

    // return reference to current object
    return *this;
}

template <typename T>
int stree<T>::empty() const
{
    return root == NULL;
}

template <typename T>
int stree<T>::size() const
{
    return treeSize;
}
The Binary Search Tree ADT

template <typename T>
void stree<T>::erase(iterator pos)
{
    // dNodePtr = pointer to node D that is deleted
    // pNodePtr = pointer to parent P of node D
    // rNodePtr = pointer to node R that replaces D
    stnode<T> *dNodePtr = pos.nodePtr, *pNodePtr, *rNodePtr;

    if (treeSize == 0)
        throw underflowError("stree erase(): tree is empty");

    if (dNodePtr == NULL)
        throw referenceError("stree erase(): invalid iterator");

    // assign pNodePtr the address of P
    pNodePtr = dNodePtr->parent;

    // If D has a NULL pointer, the
    // replacement node is the other child
    if (dNodePtr->left == NULL || dNodePtr->right == NULL)
    {
        if (dNodePtr->right == NULL)
            rNodePtr = dNodePtr->left;
        else
            rNodePtr = dNodePtr->right;

        if (rNodePtr != NULL)
            // the parent of R is now the parent of D
            rNodePtr->parent = pNodePtr;
    }
The Binary Search Tree ADT

```c
// both pointers of dNodePtr are non-NULL.
else
{
    // find and unlink replacement node for D.
    // starting at the right child of node D, find the node whose value is the smallest of all
    // nodes whose values are greater than the value in D. unlink the node from the tree.

    // pOfRNodePtr = pointer to parent of replacement node
    stnode<T> *pOfRNodePtr = dNodePtr;

    // first possible replacement is right child of D
    rNodePtr = dNodePtr->right;

    // descend down left subtree of the right child of D, keeping a record of current node and its parent.
    // when we stop, we have found the replacement
    while(rNodePtr->left != NULL)
    {
        pOfRNodePtr = rNodePtr;
        rNodePtr = rNodePtr->left;
    }
    if (pOfRNodePtr == dNodePtr)
    {
        // right child of deleted node is the replacement.
        // assign left subtree of D to left subtree of R
        rNodePtr->left = dNodePtr->left;
        // assign the parent of D as the parent of R
        rNodePtr->parent = pNodePtr;
        // assign the left child of D to have parent R
        dNodePtr->left->parent = rNodePtr;
    }
}
else
{
    // we moved at least one node down a left branch of the right child of D. unlink R from tree by
    // assigning its right subtree as the left child of the parent of R
    pOfRNodePtr->left = rNodePtr->right;
    // the parent of the right child of R is the parent of R
    if (rNodePtr->right != NULL)
        rNodePtr->right->parent = pOfRNodePtr;
    // put replacement node in place of dNodePtr assign children of R to be those of D
    rNodePtr->left = dNodePtr->left;
    rNodePtr->right = dNodePtr->right;
    // assign the parent of R to be the parent of D
    rNodePtr->parent = pNodePtr;
    // assign the parent pointer in the children of R to point at R
    rNodePtr->left->parent = rNodePtr;
    rNodePtr->right->parent = rNodePtr;
}
// complete the link to the parent node.
if (pNodePtr == NULL)  // deleting the root node. assign new root
    root = rNodePtr;
else if (dNodePtr->nodeValue < pNodePtr->nodeValue)  // attach R to the correct branch of P
    pNodePtr->left = rNodePtr;
else
    pNodePtr->right = rNodePtr;
delete dNodePtr;  // delete the node from memory and decrement tree size
treeSize--;
}
The Binary Search Tree ADT

template <typename T>
int stree<T>::erase(const T& item)
{
    int numberErased = 1;
    stnode<T> *p = findNode(item); // search tree for item
    if (p != NULL) // if item found, delete the node
        erase(iterator(p,this));
    else
        numberErased = 0;
    return numberErased;
}

template <typename T>
void stree<T>::erase(iterator first, iterator last)
{
    if (treeSize == 0)
        throw underflowError("stree erase(): tree is empty");

    iterator p = first;
    if (first == begin() && last == end())
    {
        deleteTree(root); // we are asked to erase the entire tree. erase the tree nodes from memory
        root = NULL; // tree is empty
        treeSize = 0;
    }
    else
        while (p != last) // erase each item in a subrange of the tree
            erase(p++);
}
// recursive inorder scan used to build the shadow tree
template <typename T>
tnodeShadow *stree<T>::buildShadowTree(stnode<T> *t, int level, int& column)
{
    tnodeShadow *newNode = NULL;  // pointer to new shadow tree node
    ostringstream ostr;          // text and ostr used to perform format conversion

    if (t != NULL)
    {
        newNode = new tnodeShadow; // create the new shadow tree node

        // allocate node for left child at next level in tree; attach node
        tnodeShadow *newLeft = buildShadowTree(t->left, level+1, column);
        newNode->left = newLeft;

        // initialize data members of the new node
        ostr << t->nodeValue << ends;          // format conversion
        newNode->nodeValueStr = ostr.str();
        newNode->level = level;
        newNode->column = column;

        column++;    // update column to next cell in the table

        // allocate node for right child at next level in tree; attach node
        tnodeShadow *newRight = buildShadowTree(t->right, level+1, column);
        newNode->right = newRight;
    }

    return newNode;
}
template <typename T>
void stree<T>::displayTree(int maxCharacters)
{
    string label;
    int level = 0, column = 0;
    int colWidth = maxCharacters + 1;
    int currLevel = 0, currCol = 0;

    if (treeSize == 0)
        return;

    tnodeShadow *shadowRoot = buildShadowTree(root, level, column); // build the shadow tree
    tnodeShadow *currNode; // use during the level order scan of the shadow tree
    // store siblings of each tnodeShadow object in a queue so that they are visited in order at the next level of the tree
    queue<tnodeShadow *> q;
    q.push(shadowRoot); // insert the root in the queue and set current level to 0

    // continue the iterative process until the queue is empty
    while(!q.empty())
    {
        currNode = q.front(); // delete front node from queue and make it the current node
        q.pop();

        if (currNode->level > currLevel) // if level changes, output a newline
            {
                currLevel = currNode->level;
                currCol = 0;
                cout << endl;
            }
        }
The Binary Search Tree ADT

```cpp
if(currNode->left != NULL) // if a left child exists, insert the child in the queue
    q.push(currNode->left);

if(currNode->right != NULL) // if a right child exists, insert the child in the queue
    q.push(currNode->right);

if (currNode->column > currCol) // output formatted node label
{
    cout << setw((currNode->column-currCol)*colWidth) << " ";
    currCol = currNode->column;
}
    cout << setw(colWidth) << currNode->nodeValueStr;
    currCol++;
}
    cout << endl;

deleteShadowTree(shadowRoot); // delete the shadow tree
}

template <typename T>
void stree<T>::deleteShadowTree(tnodeShadow *t)
{
    // if current root node is not NULL, delete its left subtree, its right subtree and then the node itself
    if (t != NULL)
    {
        deleteShadowTree(t->left);
        deleteShadowTree(t->right);
        delete t;
    }
}
```
Questions?
Assignment #5

• Due Sun Nov 3rd, 11:59pm

• Written Assignment
  • Ford & Topp, Chapter #10:
    | Question # | Page # |
    |------------|--------|
    | Q.15       | P.576  |
    | Q.17       | P.576  |
    | Q.19       | P.576  |

• Programming Assignment
  1. Extend the stree class defined in “stree.h” with implementing the following function

```cpp
template <typename T>
void stree<T>::convertBSTtoDLL(stnode<T>* curNode, stnode<T>*& prevNode, stnode<T>*& listHead)
```

that takes a root of a BST and converts BST to a double linked list. The constraints are as below:
- You have to do it without allocating space (in-place). This means the same BST should be changed to double linked list.
- The double linked list should be in ascending order. This means if we traverse the list from head and print it's data, data should be printed in ascending order.

Write a program that creates a tree containing the values from the integer array `arr`:

```cpp
int arr[] = {5, 3, 4, 1, 2, 7, 6, 10, 8, 15, 9, 0, 20};
int arrSize = sizeof(arr) / sizeof(int);
```

Convert the BST to DDL. Then display the DLL elements by traversing the DLL using starting from the node returned in `listHead`.
Assignment #5

• Due Sun Nov 3rd, 11:59pm
• Submission Format:
  • Written Assignment
    • Create single PDF file with name: `cs361_assignment_5_<firstName>_<lastName>`
    • Have a cover page with your name and your email
    • Submit through Blackboard.
  • Programming Assignment
    • Make sure your two programs compile and execute using `g++` on Dept’s Linux machines.
    • Create a “Readme.txt” file for each program that list how to compile and execute the program. Include your name and your email.
    • Your main file should be named “main.cpp”.
    • You should have a Makefile.
    • Zip all files (.h, .c, Makefile, etc.) for each program and name them `assign5_program` respectively.
    • Submit through Blackboard.
• Final Submission Materials (2 Files):
  • One PDF for written assignment.
  • One ZIP file for the program