CMPT 225

Lecture 17 – Binary Search Tree Operations
Last Lecture

- We saw how to ...
  - Describe binary tree and its properties
  - Describe binary search tree
Learning Outcomes

At the end of the next few lectures, a student will be able to:

- Define the following data structures:
  - Binary search tree
  - Balanced binary search tree (AVL)
  - Binary heap
  as well as demonstrate and trace their operations
- Implement the operations of binary search tree and binary heap
- Implement and analyze sorting algorithms: tree sort and heap sort
- Write recursive solutions to non-trivial problems, such as binary search tree traversals
Today’s menu

- Given a binary search tree, perform some operations such as:
  - Insert an element (a node containing an element)
  - Retrieve an element
  - Delete an element
  - Traverse a tree
Binary Search Tree (BST)

- **Definition**: A BST is a binary tree in which element stored in a node has a search key value and satisfies the following constraint:

- **Answer**: Commonly stored in right subtree, but it is up to the designer of the BST ADT class -> Design decision
BST Operations?

- Insert an element
- Retrieve an element (Search for a target element)
- Delete an element (Search for a target element)
- Traverse a tree
Insert

if tree empty
  insert new element in root
otherwise
  if new \texttt{element} < \texttt{element} stored in root
    insert new element into left subtree
  else
    insert new element into right subtree

\texttt{element} -> \texttt{search key value of element} we are inserting
Let’s try!
Retrieve (Search)

if tree empty
    target element not there!
if target element == element stored in root
    return element stored in root
otherwise
    if target element < element stored in root
        search left subtree
    else
        search right subtree

$element \rightarrow$ search key value of $element$ we are looking for
Let’s try!
Delete

search for element to be deleted
if not found -> done!

Case 1: if element is in a leaf -> delete it!

Case 2: otherwise if element’s left subtree is empty
   replace element with element in its right child/root of subtree

Case 3: if element’s right subtree is empty
   replace element with element in its left child/root of subtree

Case 4: otherwise replace element with its predecessor or successor
   copy element of predecessor over to node to be deleted
   delete predecessor (following this delete algorithm)
Let’s try!
Let’s try!
Predecessor

Definition:
Rightmost node of current node's left subtree

Successor

Definition:
Leftmost node of current node's right subtree
Let’s try!
Let’s demonstrate!
Traverse

- **post-order algorithm**
  
  if tree empty -> return
  
  otherwise traverse its left subtree then
  
  traverse its right subtree
  
  **visit** current element then

**visit** : perform some manipulation such as “display”, update
// PostOrder traversal algorithm
void postOrder(Node *n) {
    if (n != NULL) {
        postOrder(n->leftChild);
        postOrder(n->rightChild);
        visit(n);
    }
}
Min of BST?

Algorithm:

Max of BST?

Algorithm:
Learning Check

We can now ...
- Insert an element (a node containing an element)
- Retrieve an element
- Delete an element
- Traverse a tree
- Find successor of an element
- Find predecessor of an element
- Find minimum element value of BST
- Find maximum element value of BST
Next Lectures

- Binary Search Tree implementation