CMPT 225
Binary Search Tree – BST
Last Lecture

- Tree terminology
Learning Outcomes

At the end of the lectures related to trees, students will be able to:

- Abstract Data Types (ADTs):
  - define the following data structures:
    - Binary search tree
    - Balanced binary search tree (AVL)
    - Binary heap
  - and 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

- At the end of this lecture, we shall be able to describe binary search tree and its property and given a binary search tree, perform operations such as:
  - Insert an element (a node containing an element)
  - Retrieve an element
  - Delete an element
  - Traverse a tree
  - etc …
N-ary Tree where $N = 2$

- We shall now focus on 2-ary tree i.e., binary tree
- Binary tree:
  - Position-oriented or value-oriented?
Position-oriented or value-oriented?

- To answer this question, let’s create a tree of each type by inserting the following elements (in this order) D, E, B, G, A, C into each of them.
Binary Search Tree (BST)

- **Definition:**

- What about duplication, i.e., elements with search key value = X?

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

• If a binary tree $T$ has height $H$ then $T$ has between $H$ and $2^H - 1$ nodes

• Note that this property holds for binary search tree as well
BST Operations

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

- Find successor of an element
- Find predecessor of an element
- Find minimum element value of BST
- Find maximum element value of BST
Insert

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

- \textit{element} \rightarrow \textit{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 -> search key value of element we are looking for
Let’s try!
Delete

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

if *element* is in a leaf - > delete it!
otherwise if *element*’s left subtree is empty
    replace *element* with *element* in its right child/root of subtree
otherwise if *element*’s right subtree is empty
    replace *element* with *element* in its left child/root of subtree
otherwise replace *element* with its predecessor or successor
    copy *element* of predecessor over to node to be deleted
delete predecessor (following this delete algorithm)
<table>
<thead>
<tr>
<th>Predecessor</th>
<th>Successor</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Definition:</strong></td>
<td><strong>Definition:</strong></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let’s try!
Traverse

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

visit: perform some manipulation such as “display”, update
// PreOrder traversal algorithm
void preOrder(Node *n) {
    if (n != NULL) {
        visit(n);
        preOrder(n->leftChild);
        preOrder(n->rightChild);
    }
}
Let’s try!
Traverse

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

visit: perform some manipulation such as “display”, update
// InOrder traversal algorithm
void inOrder(Node *n) {
    if (n != NULL) {
        inOrder(n->leftChild);
        visit(n);
        inOrder(n->rightChild);
    }
}
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:
We described binary search tree BST
Given a binary search tree, we performed the following operations:
  ◦ Insert an element (a node containing an element)
  ◦ Retrieve an element
  ◦ Delete an element
  ◦ Traverse a tree
  ◦ etc …
Next Lecture

• Binary Search Tree ADT