Binary Search Tree (III): Search

CSCD 300 – Data Structures

Eastern Washington University

EASTERN WASHINGTON UNIVERSITY
start something big

Goal: we will learn and implement the mechanism of the following various search operations on an existing BST.

```
class BST{
   BST_Node root;   //the root of the BST

   BST(){ root = null; } /* the constructor */

   BST_Node search(int k){...}  /* If the given key k exists in the BST, return
      the reference to the node that contains k; otherwise, return null; */

   BST_Node Min(BST_Node subtree_root){...}  /* return the reference to the node
      that contains the smallest key in the subtree rooted on "subtree_root". */

   BST_Node Max(BST_Node subtree_root){...} /* return the reference to the node
      that contains the largest key in the subtree rooted on "subtree_root". */

   BST_Node Successor(BST_Node node){...} /* Return the reference to the node
      whose key is the successor of the key in "node", if such successor exists;
      otherwise, return null. */

   BST_Node Predecessor(BST_Node node){...} /* Return the reference to the node
      whose key is the predecessor of the key in "node", if such successor exists;
      otherwise, return null. */

   /* Other methods will follow here  */
}
```

Outline

1. Search for a given key
2. Find the minimum in a given subtree
3. Find the maximum in a given subtree
4. Find the successor of a given node
5. Find the predecessor of a given node

EASTERN WASHINGTON UNIVERSITY
start something big

Search for a given key

```
BST_Node BST::search(int k){
   BST_Node temp = root;

   while(temp != null and k != temp.key)
      if(k < temp.key)
         temp = temp.left;
      else
         temp = temp.right;
   return temp;
}
```

Idea

By using the definition of BST, walk along the path from the global root toward the node (which may not be existing) that contains the given key by keeping comparing the given key with the current node on the path.

Theorem

The time complexity of the `search` operation is $O(h)$, where h is the height of the tree.

- Because the search procedure is essentially a walk from the global root to the node which contains the given key or to a leaf node. Each walk step takes constant time.
- We have had a detailed discussion on the height of the BST in the previous lecture. Refer to the lecture slides if needed.

Find the minimum in a given subtree

Idea

Grab and return the reference of the left-bottom node in the subtree rooted at the given node `subtree_root`.

```
BST_Node BST::Min(BST_Node subtree_root){
    BST_Node temp = subtree_root;
    while(temp.left != null)
        temp = temp.left;
    return temp;
}
```

Theorem

The time complexity of finding the minimum is $O(h)$, where h is the height of the tree. (The reason is obvious ...)

Find the maximum in a given subtree

Idea

Grab and return the reference of the right-bottom node in the subtree rooted at the given node `subtree_root`.

```
BST_Node BST::Max(BST_Node subtree_root){
    BST_Node temp = subtree_root;
    while(temp.right != null)
        temp = temp.right;
    return temp;
}
```

Theorem

The time complexity of finding the minimum is $O(h)$, where h is the height of the tree. (The reason is obvious ...)

Find the successor of a given node

```
BST_Node BST::Successor(BST_Node x){
 if(x.right != null)
      return Min(x.right);

/* Find x's lowest ancestor, such
   that x is in its left subtree */

   y = x.parent
   while(y != null and x == y.right)
      x = y;
      y = y.parent;

   /* If y turns out to be null,
      it means x does not have a
      successor.
   */
   return y;
}
```

Questions

Why must the successor be in x's right subtree, if x's right subtree exists ?

Theorem

The procedure takes $O(h)$ time, where h is the height of the BST. (The reason is obvious ...)

Predecessor

The `Predecessor()` procedure is similar and symmetry to the `Successor()` procedure.

Find the predecessor of a given node

```
BST_Node BST::Predecessor(BST_Node x){
 if(x.left != null)
      return Max(x.left);

/* Find x's lowest ancestor, such
   that x is in its right subtree */

   y = x.parent
   while(y != null and x == y.left)
      x = y;
      y = y.parent;

   /* If y turns out to be null,
      it means x does not have a
      predecessor.
   */
  return y;
}
```

Questions

Why must the predecessor be in x's left subtree, if x's left subtree exists ?

Theorem

The procedure takes $O(h)$ time, where h is the height of the BST. (The reason is obvious ...)