CMPT 225

Lecture 18 – Binary Search Tree Implementation
Last Lecture

- We saw how to ...
  - 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
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

- Various ways of implementing a Binary Search Tree
Problem Statement

- We often need to keep data in such a way that it can be frequently searched and retrieved
  - For example:
  - Possible data collection so far:
    - A value-oriented List
      - Array-based implementation
        - Search in $O(\log n)$ using binary search
        - Insertion and deletion in $O(n)$
      - Link-based implementation
        - Search, insertion and deletion in $O(n)$
Binary Search Tree

- Considering that …
  - a BST is a sorted data collection, by definition, and
  - we can perform the operations required by the problem statement, namely insert, search and retrieve using a BST
- Then a BST may be an appropriate data collection to use in order to solve the problem

OR

- It may be an appropriate underlying data structure for a Dictionary ADT class
BST Implementation

- Array-based Implementation #1

```cpp
class BSTNode {

private:
    ElementType element; // element stored in tree node
    int leftChild;      // index to left child/subtree
    int rightChild;     // index to right child/subtree

};
```
class BinarySearchTree {

private:
    BSTNode myTree[CAPACITY]; // array of tree elements
    int root;                 // index of root
    int elementCount;        // number of elements
BST Array-Based Implementation #1
- How it works!

- BSTNode
- elementCount
- root

```java
class BSTNode {
    int element; // object
    int index; // index of left child
    int index; // index of right child

    // Constructor
    BSTNode(int element, int leftIndex, int rightIndex) {
        this.element = element;
        this.index = leftIndex;
        this.rightIndex = rightIndex;
    }
}
```
insert( newElement )
BST Implementation

- Array-based Implementation #2

```cpp
class BinarySearchTree {

private:
    ElementType myTree[CAPACITY];  // array of elements
    int elementCount;              // number of elements

};
```
BST Array-Based Implementation #2

```
myTree = new BinarySearchTree()
```

object of Binary Search Tree class type

```
elementCount
```

12
How to connect each cell of the array to its left/right child/subtree?

- Considering cell at index $i$
  - its left child/subtree is located at index
  - its right child/subtree is located at index
  - its parent is located at index
BST Array-Based Implementation #2
- How it works!

[elementCount]
BST array-based implementation #2
- One disadvantage

- It creates sparse array:
BST Implementation

- Link-based Implementation

```cpp
class BSTNode {

private:
    ElementType element;   // element stored in tree node
    BSTNode* leftChild;   // link to left child/subtree
    BSTNode* rightChild;  // link to right child/subtree
}
```
BinarySearchTree ADT Class

class BinarySearchTree {

private:
    BSTNode* root;       // root of tree
    int elementCount;    // number of elements

Comparing Implementations

- Array-based implementation of Binary or Binary Search tree is not very efficient unless the tree is complete!
  - Why?

- Link-based implementation more common
Time Efficiency of BST Operations
- Activity - 1
Time Efficiency of BST Operations
- Activity - 2
Activity - Analysis

1. How many elements are contained in each resulting BST?
2. Determine the height of each resulting BST.
3. Which element would you be searching for if you were to perform the “best case” scenario of the search operation in each of these BST’s?
4. What would be the time efficiency of this scenario?
5. Which element would you be searching for if you were to perform the “worst case” scenario of the search operation in each of these BST’s?
6. What would be the time efficiency of this scenario?
Time Efficiency of BST operations using the Big O notation

<table>
<thead>
<tr>
<th>Operations</th>
<th>$H &lt;&lt; n$</th>
<th>$H = n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>getElementCount</td>
<td></td>
<td></td>
</tr>
<tr>
<td>insert</td>
<td></td>
<td></td>
</tr>
<tr>
<td>delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>retrieve</td>
<td></td>
<td></td>
</tr>
<tr>
<td>traverse</td>
<td></td>
<td></td>
</tr>
<tr>
<td>successor/predecessor</td>
<td></td>
<td></td>
</tr>
<tr>
<td>min/max</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
We can now ...

- implementation a binary search tree in a variety of ways
  - Array-based
  - Link-based
- Expressed time efficiency of its operations for various cases
Next Lectures

- Self balancing binary search trees