Binary Search Tree

Juan Zhai

juan.zhai@rutgers.edu

Watch the videos first and then come back

- BST structure and search: https://www.youtube.com/watch?v=J3YY-Ef2xlE&ab_channel=SeshVenugopal
- BST insert: https://www.youtube.com/watch?v=BVeEmH26PQ4&ab_channel=SeshVenugopal
- BST delete: https://www.youtube.com/watch?v=3TOl3Fv4394&ab_channel=SeshVenugopal

```
/** This interface imposes a total ordering on the objects
 *  of each class that implements it. */
public interface Comparable<T>{
     /** Compares this object with the specified object for
       * order. Returns a negative integer, zero, or a
       * positive integer as this object is less than,
       * equal to, or greater than the specified object. */
   public int compareTo(T o);
}
```


- An interface is a group of related methods with empty bodies.
- Interfaces form a contract: Implementing an interface requires a class to have behaviors specified by the interface.

```
public class String implements Comparable<String>{
     public int compareTo(String anotherString){
          …
     }
}
```

- To implement this interface, use the implements keyword in the class declaration.

Generic Binary Search Tree

```
public class Node<T extends Comparable<T>> {
    private T key;
    private Node<T> left;
    private Node<T> right;
    public Node(T key) {
        this.key = key;
        this.right = null;
        this.left = null;
    }
}

public class BST<T extends Comparable<T>>{
    Node<T> root;
}
```

This restricts the type parameter T to be a type that implements the interface `Comparable<T>`, guaranteeing that the call to `compareTo()` is valid, meaning support comparison with other instances of its own type

Refer to Sakai Code

Generic Search

Q: Why search returns T?

A: The search is based on key, while what is returned is the entire object

```
public T search(T targetKey) {
        Node current = root;
        while(current != null){
            int c = targetKey.compareTo(current.key);
            if(c == 0)
              return current.key;
            if(c < 0)
              current = current.left;
            else
              current = current.right;
        }
        return null;
}
```


conditional operator: ? :

`(condition) ? a : b`

is an expression which returns a when condition is true and returns b when condition is false

```
current = (c<0) ? current.left : current.right;
```

```java
//Student MUST implement the Comparable interface
class Student implements Comparable<Student> {
    String id;
    String name;
    String address;

    public Student(String id, String name, String addr) {
        this.id = id;
        this.name = name;
        this.address = addr;
    }

    public int compareTo(Student other) {
        return id.compareTo(other.id);
    }
}
```

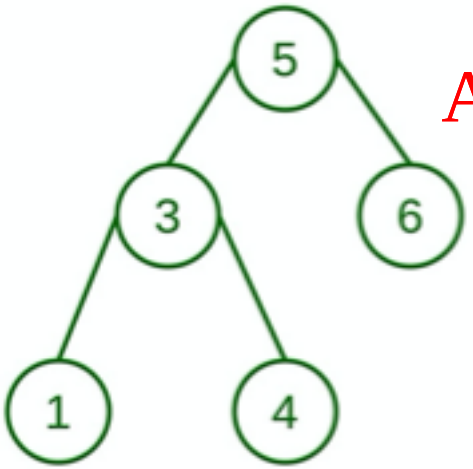
```java
        BST<Student> studentBST = new BST<Student>();
        studentBST.insert(new Student("B052","David Jones","110 Frelinghuysen Rd"));
        studentBST.insert(new Student("D032","Lucy Smith", "305 Univeristy St"));
        studentBST.insert(new Student("M016","Emily Taylor","77 Massachusetts Ave"));

        // get address for a student based on the id
        Student student = studentBST.search(new Student("D032",null,null));
        System.out.println(student.name + ": " + student.address);
```

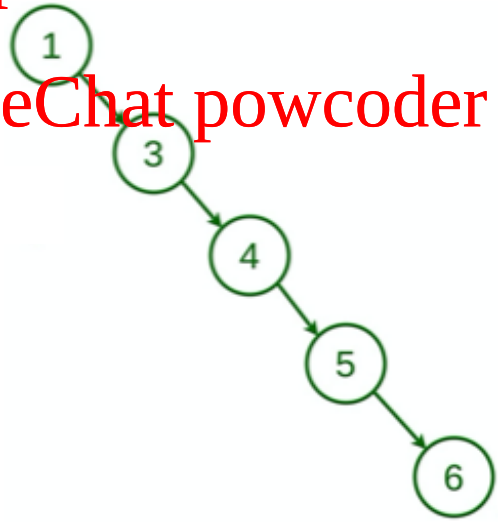

- Specific class decides how to compare two objects
  - Implements *compareTo()* in *Comparable*
- student-to-be-search is a student with id, and student-to-be-returned has more information

Different shapes

- For the same set of keys, we can have binary search tree in different shapes.
- Depends on what sequence of the items are inserted into the tree



{5, 3, 1, 4, 6}

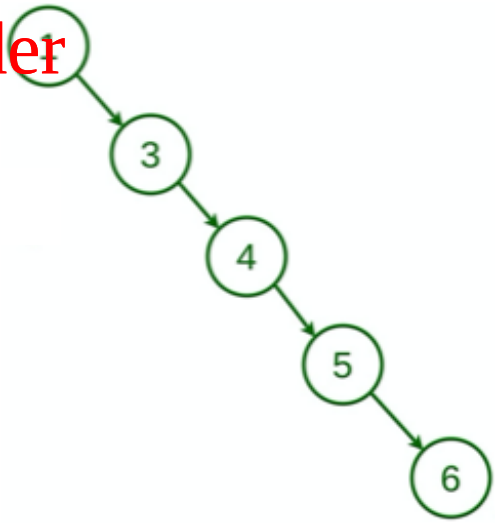


{1, 3, 4, 5, 6}

skewed tree: each node contains either only left or only right sub tree

Skewed Tree

- Worst case running time
  - search: O(n), even worse than binary search in a sorted array which has worst case O(log *n*)
  - insertion/deletion: O(n)
    - O(n) for search the position-to-insert-into/element-to-be-deleted
    - O(1) for insertion and deletion
    - O(n) + O(1) → O(n)
  - Reshape



	Unordered List		Ordered Array		Binary Search Tree	
	Best	Worst	Best	Worst	Best	Worst
Search	O(1)	O(n)	O(1)	O(log *n*)	O(1)	O(n) [5]
Insert	O(1) [0]	O(1) [0]	O(log *n*) [2]	O(n) [3]	O(log *n*) [4]	O(n) [5]
Delete	O(1)	O(n) [1]	O(log *n*) [2]	O(n) [3]	O(log *n*) [6]	O(n) [5]

Best case and worse case for the same algorithm with different situations.

[0] Insertion algorithm always inserts at the front.

[1] Search to end, then delete

[2] Insert/delete at end, O(1), but search needs O(log *n*)

[3] Insert/delete the middle key, O(1), but move n/2 keys

[4] Insertion is always done at the leaf level, so the insertion process has to traverse O(log *n*) distance

[5] Skewed tree

[6] Delete a leaf node, O(1), but search needs O(log *n*). Deleting root node needs to find the minimum/maximum which takes O(log *n*)

	Unordered List		Ordered Array		Binary Search Tree	
	Best	Worst	Best	Worst	Best	Worst
Search	O(1)	O(n)	O(1)	O(log *n*)	O(1)	O(n) [5]
Insert	O(1) [0]	O(1) [0]	O(log *n*) [2]	O(n) [3]	O(log *n*) [4]	O(n) [5]
Delete	O(1)	O(n) [1]	O(log *n*) [2]	O(n) [3]	O(log *n*) [6]	O(n) [5]

It seems that BST performs even worse than sorted array.

Goal: Insertions and deletions should be faster than $O(n)$, and search time should not be slower than $O(\log n)$.

Solution: Keep the structure of BST balanced, meaning height never exceeds $O(\log n)$. Then the search/insert/delete times would never exceeds $O(\log n)$.

→ Reshape a BST

Get help from CAVE

- The CAVE is available to students – no appointment necessary, help with concepts and assignments
  - https://resources.cs.rutgers.edu/docs/rooms-equipment/cave/
  - Monday through Thursday, 1-11PM
  - Friday 1-6PM
  - Sunday 3-11PM

https://rutgers.webex.com/meet/cs-the-cave

Learning Center

- Get help from https://rlc.rutgers.edu/node/95