Binary Search Trees

EECS2011 N & Z:
Fundamentals of Data Structures
Winter 2022

CHEN-WEI WANG

Learning Outcomes of this Lecture

This module is designed to help you understand:

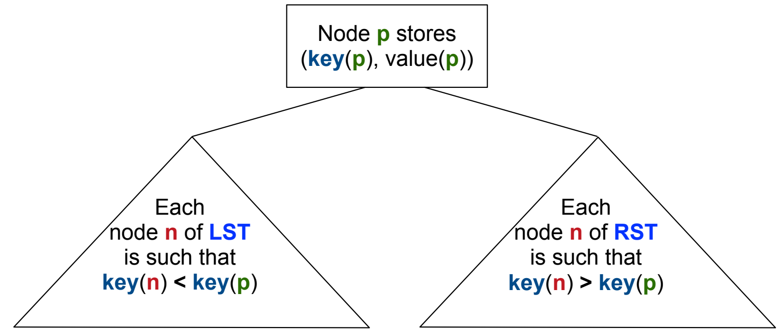
- ***Binary Search Trees* (*BSTs*)** = ***BTs*** + ***Search Property***
- Implementing a ***Generic*** BST in Java
- BST Operations:
  - ***Searching***: Implementation, Visualization, RT
  - ***Insertion***: (Sketch of) Implementation, Visualization, RT
  - ***Deletion***: (Sketch of) Implementation, Visualization, RT

Binary Search Tree: Recursive Definition

A *Binary Search Tree* (*BST*) is a ***BT*** satisfying the ***search property***:

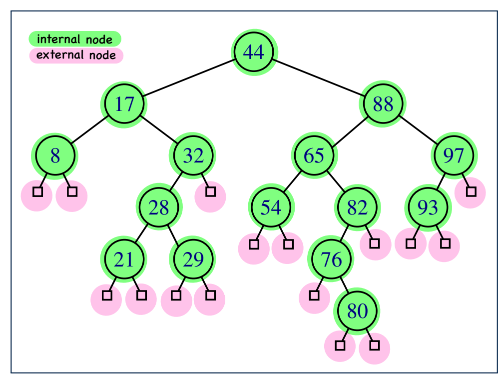
Each ***internal node*** p stores an ***entry***, a key-value pair (k, v), such that:

- For each node ***n*** in the ***LST*** of ***p***: $key(n) < key(p)$
- For each node ***n*** in the ***RST*** of ***p***: $key(n) > key(p)$



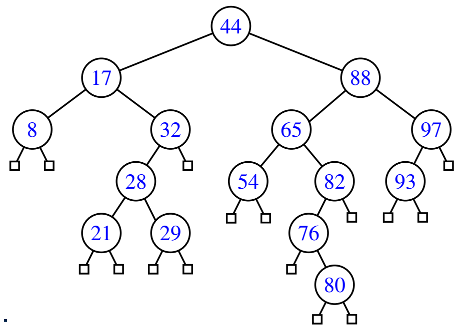
BST: Internal Nodes vs. External Nodes

- We store key-value pairs only in ***internal nodes***.
- Recall how we treat ***header*** and ***trailer*** in a DLL.
- We treat ***external nodes*** as ***sentinels***, in order to **simplify** the ***coding logic*** of BST algorithms.



BST: Sorting Property

- An ***in-order traversal*** of a *BST* will result in a sequence of nodes whose ***keys*** are arranged in an ***ascending order***.
- Unless necessary, we may only show ***keys*** in BST nodes:



Justification:
- ***In-Order Traversal***: Visit ***LST***, then ***root***, then ***RST***.
- ***Search Property of BST***: keys in ***LST***/***RST*** < / > ***root***'s key

Implementation: Generic BST Nodes

```java
public class BSTNode<E> {
  private int key; /* key */
  private E value; /* value */
  private BSTNode<E> parent; /* unique parent node */
  private BSTNode<E> left; /* left child node */
  private BSTNode<E> right; /* right child node */

  public BSTNode() { ... }
  public BSTNode(int key, E value) { ... }

  public boolean isExternal() {
    return this.getLeft() == null && this.getRight() == null;
  }
  public boolean isInternal() {
    return !this.isExternal();
  }
  public int getKey() { ... }
  public void setKey(int key) { ... }
  public E getValue() { ... }
  public void setValue(E value) { ... }
  public BSTNode<E> getParent() { ... }
  public void setParent(BSTNode<E> parent) { ... }
  public BSTNode<E> getLeft() { ... }
  public void setLeft(BSTNode<E> left) { ... }
  public BSTNode<E> getRight() { ... }
  public void setRight(BSTNode<E> right) { ... }
}
```


Implementation: BST Utilities – Traversal

```java
import java.util.ArrayList;
public class BSTUtilities<E> {
  public ArrayList<BSTNode<E>> inOrderTraversal(BSTNode<E> root) {
    ArrayList<BSTNode<E>> result = null;
    if(root.isInternal()) {
      result = new ArrayList<>();

      if(root.getLeft().isInternal) {
        result.addAll(inOrderTraversal(root.getLeft()));
      }

      result.add(root);

      if(root.getRight().isInternal) {
        result.addAll(inOrderTraversal(root.getRight()));
      }
    }
    return result;
  }
}
```


Testing: Connected BST Nodes

Constructing a *BST* is similar to constructing a *General Tree*:

```java
@Test
public void test_binary_search_trees_construction() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<String> extN3 = new BSTNode<>();
  BSTNode<String> extN4 = new BSTNode<>();
  n28.setLeft(n21); n21.setParent(n28);
  n28.setRight(n35); n35.setParent(n28);
  n21.setLeft(extN1); extN1.setParent(n21);
  n21.setRight(extN2); extN2.setParent(n21);
  n35.setLeft(extN3); extN3.setParent(n35);
  n35.setRight(extN4); extN4.setParent(n35);
  BSTUtilities<String> u = new BSTUtilities<>();
  ArrayList<BSTNode<String>> inOrderList = u.inOrderTraversal(n28);
  assertTrue(inOrderList.size() == 3);
  assertEquals(21, inOrderList.get(0).getKey());
  assertEquals("mark", inOrderList.get(0).getValue());
  assertEquals(28, inOrderList.get(1).getKey());
  assertEquals("alan", inOrderList.get(1).getValue());
  assertEquals(35, inOrderList.get(2).getKey());
  assertEquals("tom", inOrderList.get(2).getValue());
}
```

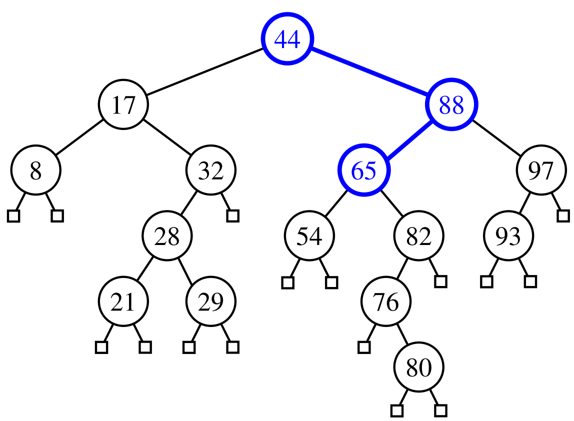
Implementing BST Operation: Searching

Given a *BST* rooted at node ***p***, to locate a particular ***node*** whose ***key*** matches ***k***, we may view it as a ***decision tree***.

```
public BSTNode<E> search(BSTNode<E> p, int k) {
  BSTNode<E> result = null;
  if(p.isExternal()) {
    result = p; /* unsuccessful search */
  }
  else if(p.getKey() == k) {
    result = p; /* successful search */
  }
  else if(k < p.getKey()) {
    result = search(p.getLeft(), k); /* recur on LST */
  }
  else if(k > p.getKey()) {
    result = search(p.getRight(), k); /* recur on RST */
  }
  return result;
}
```


Visualizing BST Operation: Searching (1)

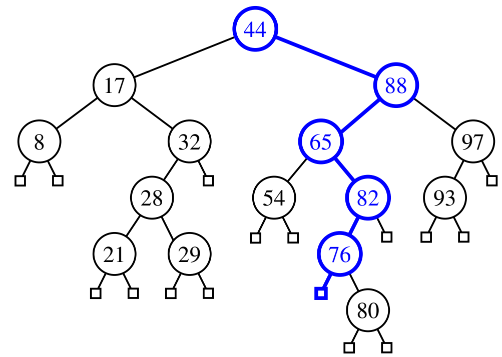
A ***successful*** search for ***key 65***:



The ***internal node*** storing key 65 is **returned**.

Visualizing BST Operation: Searching (2)

- An ***unsuccessful*** search for ***key 68***:



The ***external, left child node*** of the ***internal node*** storing ***key 76*** is **returned**.

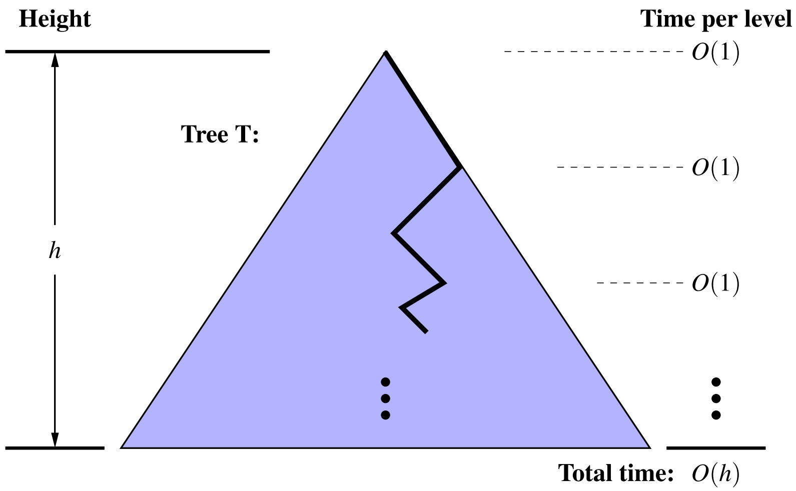
- *Exercise* : Provide ***keys*** for different ***external nodes*** to be **returned**.

Testing BST Operation: Searching

```
@Test
public void test_binary_search_trees_search() {
  BSTNode<String> n28 = new BSTNode<>(28, "alan");
  BSTNode<String> n21 = new BSTNode<>(21, "mark");
  BSTNode<String> n35 = new BSTNode<>(35, "tom");
  BSTNode<String> extN1 = new BSTNode<>();
  BSTNode<String> extN2 = new BSTNode<>();
  BSTNode<String> extN3 = new BSTNode<>();
  BSTNode<String> extN4 = new BSTNode<>();
  n28.setLeft(n21); n21.setParent(n28);
  n28.setRight(n35); n35.setParent(n28);
  n21.setLeft(extN1); extN1.setParent(n21);
  n21.setRight(extN2); extN2.setParent(n21);
  n35.setLeft(extN3); extN3.setParent(n35);
  n35.setRight(extN4); extN4.setParent(n35);

  BSTUtilities<String> u = new BSTUtilities<>();
  /* search existing keys */
  assertTrue(n28 == u.search(n28, 28));
  assertTrue(n21 == u.search(n28, 21));
  assertTrue(n35 == u.search(n28, 35));
  /* search non-existing keys */
  assertTrue(extN1 == u.search(n28, 17)); /* *17* < 21 */
  assertTrue(extN2 == u.search(n28, 23)); /* 21 < *23* < 28 */
  assertTrue(extN3 == u.search(n28, 33)); /* 28 < *33* < 35 */
  assertTrue(extN4 == u.search(n28, 38)); /* 35 < *38* */
}
```

RT of BST Operation: Searching (1)



RT of BST Operation: Searching (2)

- Recursive calls of `search` are made on a ***path*** which
  - Starts from the **root**
  - Goes down one ***level*** at a time
    RT of deciding from each node to go to LST or RST? [***O(1)***]
  - Stops when the key is found or when a ***leaf*** is reached
    Maximum number of nodes visited by the search? [$h + 1$]

∴ RT of ***search on a BST*** is $O(h)$

- Recall: Given a BT with ***n*** nodes, the ***height h*** is bounded as:

$$log(n+1) - 1 \leq h \leq n - 1$$

  - ***Best*** RT of a ***binary search*** is ***O(log(n))*** [***balanced*** BST]
  - ***Worst*** RT of a ***binary search*** is ***O(n)*** [***ill-balanced*** BST]
- ***Binary search*** on non-linear vs. linear structures:

	Search on a **BST**	Binary Search on a **Sorted Array**
START	Root of BST	Middle of Array
PROGRESS	LST or RST	Left Half or Right Half of Array
BEST RT	***O(log(n))***	***O(log(n))***
WORST RT	***O(n)***	

Sketch of BST Operation: Insertion

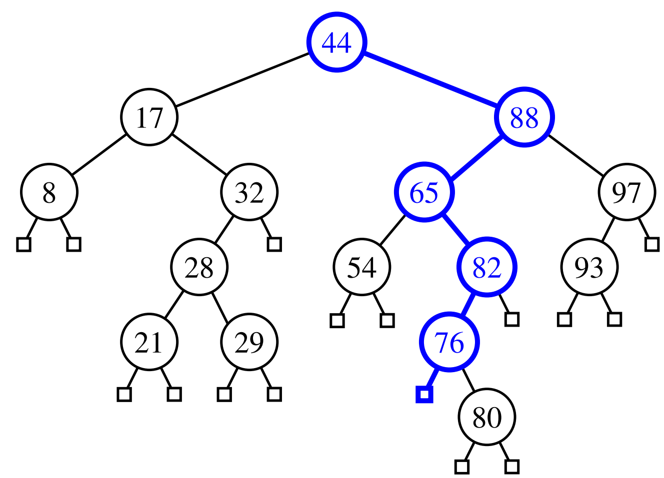
To *insert* an ***entry*** (with **key** *k* & **value** *v*) into a BST rooted at ***node*** *n*:

- Let node ***p*** be the return value from `search(n, k)`.
- If ***p*** is an ***internal node***
  ⇒ Key *k* exists in the BST.
  ⇒ Set *p*'s value to *v*.
- If ***p*** is an ***external node***
  ⇒ Key *k* deos **not** exist in the BST.
  ⇒ Set *p*'s key and value to *k* and *v*.

Running time? [$O(h)$]

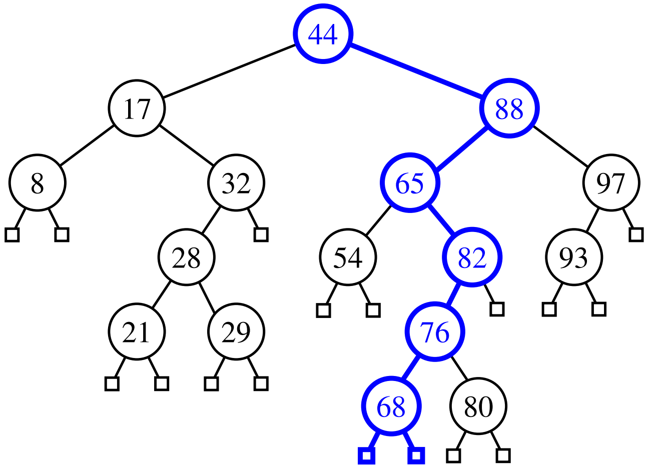
Visualizing BST Operation: Insertion (1)

Before *inserting* an entry with ***key 68*** into the following BST:



Visualizing BST Operation: Insertion (2)

After *inserting* an entry with ***key 68*** into the following BST:



Exercise on BST Operation: Insertion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void insert(BSTNode<E> p, int k, E v)` method.

Sketch of BST Operation: Deletion

To *delete* an ***entry*** (with **key** k) from a BST rooted at ***node*** n:

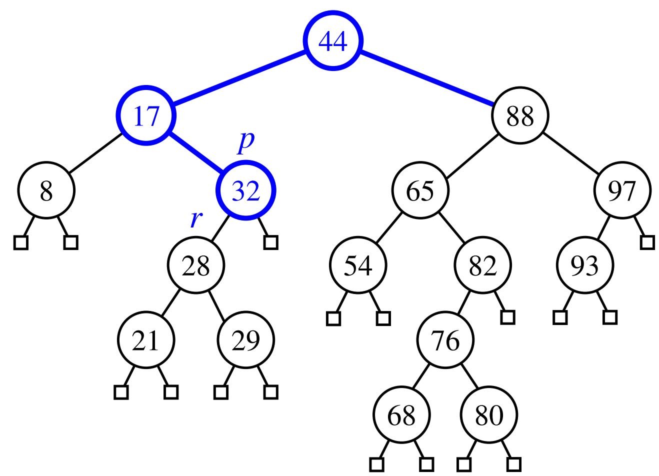
Let node ***p*** be the return value from `search(n, k)`.

- **Case 1**: Node ***p*** is ***external***.
  k is not an existing key ⇒ Nothing to remove
- **Case 2**: Both of node ***p***'s child nodes are ***external***.
  No "orphan" subtrees to be handled ⇒ Remove ***p*** [Still BST?]
- **Case 3**: One of the node ***p***'s children, say r, is ***internal***.
  - r's sibling is ***external*** ⇒ Replace node ***p*** by node r [Still BST?]
- **Case 4**: Both of node ***p***'s children are ***internal***.
  - Let r be the ***right-most internal node p's LST***.
    ⇒ r contains the ***largest key s.t. key(r) < key(p)***.
    Exercise: Can r contain the ***smallest key s.t. key(r) > key(p)***?
  - Overwrite node ***p***'s entry by node r's entry. [Still BST?]
  - r being the ***right-most internal node*** may have:
    - Two ***external child nodes*** ⇒ Remove r as in **Case 2**.
    - An ***external, RC*** & an ***internal LC*** ⇒ Remove r as in **Case 3**.

Running time? [$O(h)$]

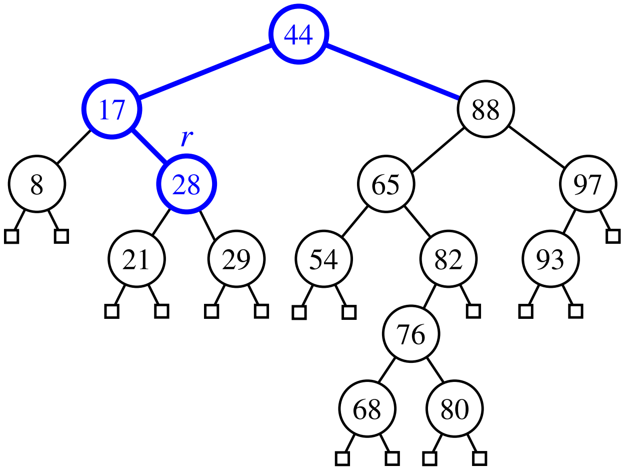
Visualizing BST Operation: Deletion (1.1)

(**Case 3**) Before *deleting* the node storing ***key 32***:



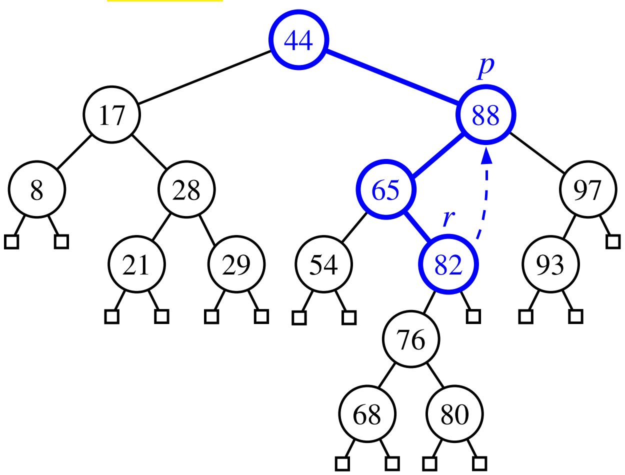
Visualizing BST Operation: Deletion (1.2)

(**Case 3**) After *deleting* the node storing ***key 32***:



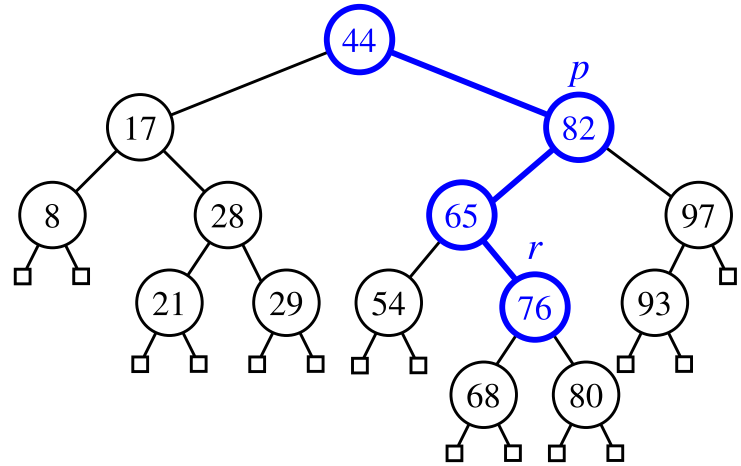
Visualizing BST Operation: Deletion (2.1)

(**Case 4**) Before *deleting* the node storing ***key 88***:



Visualizing BST Operation: Deletion (2.2)

(**Case 4**) After *deleting* the node storing ***key 88***:



Exercise on BST Operation: Deletion

Exercise: In `BSTUtilities` class, ***implement*** and ***test*** the `void delete(BSTNode<E> p, int k)` method.

Index (1)

Index (2)

Index (3)