Summary of Sorting

• Bubble sort, insertion sort, selection sort
  – Pros: very easy to implement.
  – Cons: $O(N^2)$ cost, slow.

• Merge sort
  – Pros: $O(N \times \log N)$ cost.
  – Cons: requires additional memory space.

• Quick sort
  – Pros: $O(N \times \log N)$ cost, very fast.
  – Cons: picking the pivot element can be tricky.

• The above are all comparison based sorting.
Summary of Sorting

• Radix sort
  – Pros: not based on comparison, $O(d*N)$ cost. This is often faster than comparison based sorting.
  – Cons: relies on strong assumption of the input numbers, not suitable for general sorting.

• Assignment 5: experiment with sorting and compare the performance of different methods.
Lecture 19: Binary Trees
Binary Trees

• Generalized from Linked List.

• Advantages:
  – Fast to search
  – Fast to insert and delete

• Recall the search and insertion costs of
  1. Ordered Array
  2. Linked List

• Binary tree combines the advantages of both.
Trees

• What is a tree?
  – **Nodes**: store data and links
  – **Edges**: links, typically directional

• The tree has a top node: **root node**

• The structure looks like reversed from real trees. Or not?

• Example.
Binary Trees

• Strictly speaking: trees are connected acyclic graphs (i.e. no loops).

• Some other examples of abstract tree structure:
  – Think about the way computer files are organized.

• There are many different kinds of trees.

• Here we will focus on **binary trees**
  – Each node has at most 2 children
  – In other words, at most 2 branches at each node
Terminology

- **Root**: The topmost node in the tree.
- **Path**: A sequence of nodes connected by edges.
- **Parent**: A node that has at least one child.
- **Child**: A node that has a parent.
- **Left Child**: A node that is connected to its parent from the left.
- **Right Child**: A node that is connected to its parent from the right.
- **Subtree**: A tree consisting of a node and all of its descendants.

**Example**:
- **D** is the left child of **B**.
- **E** is the right child of **B**.
- **F** has a subtree with **I** and **J** as its children.
- The dashed line represents a path.
Terminology

• **Root**
  – The node at the top of the tree.
  – There is only one root.

• **Path**
  – The sequence of nodes traversed by traveling from the root to a particular node.
  – Each path is *unique*, why?
Terminology

• **Parent**
  – The node that points to the current node.
  – Any node, except the root, has 1 and only 1 parent.

• **Child**
  – Nodes that are pointed to by the current node.

• **Leaf**
  – A node that has no children is called a leaf.
  – There can be many leaves in a tree.
Terminology

• **Interior node**
  – An interior node has at least one child.

• **Subtree**
  – Any node can be considered the root of a subtree.
  – It consists of all descendants of the current node.

• **Visit**
  – Checking the node value, display node value etc.

• **Traverse**
  – Visit all nodes in some specific order.
  – For example: visit all nodes in ascending key value.
Terminology

• **Levels**
  – The path length from root to the current node.
  – Recall that each path is unique.
  – Root is at level 0.

• **Height**
  – The maximum level in a tree.
  – $O(\log N)$ for a reasonably balanced tree.

• **Keys**
  – Each node stores a key value and associated data.
**Terminology**

- **Left child / Right child**
  - These are specific to binary trees.
  - Some nodes may have only 1 child.
  - Leaf nodes have no child.

- **Binary Search Tree (BST)**
  - For any node A, its **entire left subtree** must have values less than A, and the **entire right subtree** must have values larger than or equal to A.
Binary Search Tree (BST)

• Data Structure. The Node class:

```java
class Node {
    int iData;       // key value
    double dData;   // other data
    Node leftChild; // reference to l.c.
    Node rightChild; // reference to r.c.
    public void displayNode();
}
```
Binary Search Tree (BST)

• Data Structure. The Tree class:

```java
class Tree {
    private Node root; // reference to root
    public Node find(int key);
    public void insert(int key, double dd);
    public void delete(int key);
}
```
Binary Search Tree (BST)

• Example application code:

```java
public static void main(...) {
    Tree tree = new Tree;
    tree.insert(50, 1.5);
    tree.insert(25, 1.7);
    tree.insert(75, 1.9);
    Node ref = tree.find(25);
    tree.delete(75);
}
```
Search (Find)

• Workshop applet.
• Find a node.
  – How?
  – This is very similar to binary search in ordered array.

• Now let’s write code to implement search.
  – Non-recursive version (iterative).
  – Recursive version.
Search (Find)

• What’s the cost of search?
  \(O(\log N)\)

• This is the same with binary search
  – Not surprising given the name: BST

• But what if the tree is unbalanced?
Tree Balance

• **Balance** of the tree.
  – The relative number of nodes on one side vs. the other side.
  – This can vary with respect to different nodes.

• When the tree is **highly unbalanced**, the search cost can degrade to **O(N)**.
Insertion

• Workshop applet

• Observation:
  – It’s **always inserted as a leaf node**.

• Code.
Traversing the Tree

• Traversing – visiting all nodes in a specific order.
  – This is obvious for Array and Linked List.
  – For a tree, there are three different ways.
• In-order traversal
• Pre-order traversal
• Post-order traversal

• All of these use recursion.
In-Order Traversal

• At any node, follows this recursion:
  1. Traverse the left subtree.
  2. Visit (e.g. print out) the current node.
  3. Traverse the right subtree.

• Step 2 is why it’s called in-order traversal.
• Let’s write code together.
In-Order Traversal

• For a BST, in-order traversal will visit all nodes in **ascending order**.

• For other types of trees, in-order traversal still works, but it won’t guarantee ascending order.
Pre-Order Traversal

• At any node, follows this recursion:
  1. Visit (e.g. print out) the current node.
  2. Traverse the left subtree.
  3. Traverse the right subtree.

• Step 1 is why it’s called pre-order traversal.

• Let’s look at an example.

• What’s the result of this for BST?
Post-Order Traversal

• At any node, follows this recursion:
  1. Traverse the **left** subtree.
  2. Traverse the **right** subtree.
  3. Visit (e.g. print out) the **current** node.

• Step 3 is why it’s called post-order traversal.

• Let’s look at an example.

• What’s the result of this for BST?
Post-Order Traversal

• At any node, follows this recursion:
  1. Traverse the **left** subtree.
  2. Traverse the **right** subtree.
  3. Visit (e.g. print out) the **current** node.

• Step 3 is why it’s called post-order traversal.

• Let’s look at an example.

• How to visit nodes in descending order?
Traversing a Tree

• Given a tree, you need to be able to write the results of in-order, pre-order, and post-order traversals. This **WILL** be tested in final exam.

• We know in-order traversal is useful for BST, so what’s the point of pre-order and post-order?