Lectures 18-19-20-21- Trees

- BST runs in $O(\log N)$ for search/insertion/deletion, and $O(N)$ for traversal

Binary Search Tree (BST)

```java
class Node {
    public int key; // key-id value
    public Node left; // left child
    public Node right; // right child

    // constructor
    public Node(int key) {
        this.key = key;
        left = null;
        right = null;
    }
}
```

```java
class Tree {
    private Node root; // Ref. root

    public Tree() { // constructor
        root = null;
    }
    // methods...Here
}
```

```java
public Node find(int i) {
    Node current = root; // start at root
    while (current != null && current.key != i) {
        if (i < current.key) { // go left
            current = current.left;
        } else { // go right
            current = current.right;
        }
    }
    return current; // return Node or null
}
```

---

10

5

1

7

14

16

parent

left child

right child

level 0

level 1

level 2
Transform a list into a binary search tree:

10, 4, 7, 15, 3, 18, 16
Lectures 18-19-20-21 - Trees

- Main methods (recursive or iterative): find, insert, delete, maximum/minimum, traversal (in-order, post-order, pre-order)

```java
public Node recfind(Node current, int i) {
    if (current == null) return null;
    elseif (i < current.key) // search left
        recfind(current.left, i);
    elseif (i > current.key) // search right
        recfind(current.right, i);
    else // find the target
        return current;
}

public void recinsert(Node current, int i) {
    if (i < current.key) {
        // search left
        if (current.left == null) // insert node?
            current.left = new Node(i);
        else
            recinsert(current.left, i); // keep searching
    }
    else if (i >= current.key) {
        // search right
        if (current.right == null) // insert node?
            current.right = new Node(i);
        else
            recinsert(current.right, i); // keep searching
    }
}

public Node minimum() {
    Node current, last;
    current = root;
    while (current != null) {
        last = current; // remember node
        current = current.left;
    }
    return last;
}
```

Lecture 20: Deleting a node
### Trees

#### in-order
1. Visit the left subtree
2. Visit the node (ex: display it)
3. Visit the right subtree

#### pre-order
1. Visit the node (ex: display it)
2. Visit the left subtree
3. Visit the right subtree

#### post-order
1. Visit the left subtree
2. Visit the right subtree
3. Visit the node (ex: display it)

```java
public void inOrder(Node current) {
    if (current!=null){
        inOrder(current.left);  //1
        System.out.print(current.key+" ");  //2
        inOrder(current.right);  //3
    }
}
```

**Result is (In-Order) :** 1 5 7 10 14 16
**Result is (Pre-Order) :** 10 5 1 7 14 16
**Result is (Post-Order):** 1 7 5 16 14 10
**Lectures 18-19-20-21- Trees**

- **Red-Black Tree rules (to ensure that the tree is balanced)**
  1. Every node is either **red** or **black**
  2. The root is always **black**
  3. If a node is **red**, its children must be **black**
  4. Every path from the root to a leaf, or to a null child, must contain the same number of **black** nodes (black height is the same)

**Insertion**

- Color of inserted node is always **red** by default
- The insertion is first similar to the one for BST, find the position where the node should be inserted
- Three main stages to fix violations of the rules
  - a- **Color flips on the way down**
    Every time the insertion routine encounters a **black** node that has **two red** children, it must change the children to black and the parent to red (unless the later is the root)
  - b- **Rotations after the node is inserted**
  - c- **Rotations on the way down**

**List:** 1, 2, 3, 4, 5
Lectures 22-23: Heaps

- Heap offers a fast implementation for priorityQ \(\rightarrow O(\log N)\) for enqueue/dequeue
- Priority item can be the largest, smallest, etc.
- Priority queue ← Heap == (CBT+ heap cond.) ← array implementation
- CBT: Complete Binary Tree
- **heap condition:** *every node's key is larger than (or equal to) the keys of its children.*
- A heap is then weakly ordered compared to a BST
- Difference with BST: the priority item always at the root level (no need to search/traverse to locate it)
- Example of HeapArray and its CBT representation
Lectures 22-23: Heaps

- Insertion (uses trickle-up)

- Removal (uses trickle-down)

O(logN) Algorithms
Lectures 22-23: Heaps

- Heapify an array (using CBT representation): Example:
  - 1<sup>st</sup> approach: using successive insertion
  - 2<sup>nd</sup> approach: using trickledown in place

O(NlogN) Algorithms - Work 'in-place'
Lectures 22-23: Heaps

- **Heapsort - Motivations**
  - It is not difficult to Heapify a random array → cost is $O(N \log N)$
  - Removing one priority item is $O(\log N)$

  **Remark:** once root item is replaced by last node, only one trickledown is enough to heapify the new data

  - Removing N times (successive removals) → $O(N \log N)$ sorting algorithm (in practice a bit slower than quicksort, but no worst case and it easy enough to implement)

  - Work in place (no additional memory)

- Heapify + Heapsort:

  ```c
  for (int i=N/2-1;i>=0;i--) trickleDown(i);
  ```

  ```c
  for (int j=N-1;j>=0;j--)
  {
    Node priorityNode=remove();
    heapArray[j] = priorityNode;
  }
  ```
Lectures 22-23: Heaps
Lectures 24-25- Hash Tables

- **Motivation**: optimal insertion and search O(1) (cannot traverse)
- Array-based implementation
- **Hashing**: It is a mapping that converts a 'number key' (integer, String, etc.) that belongs to a large range into a much smaller index array number.
  - Hash function: int `hash(key)`
    - Simple approach: `smallNumber=largeNumber%smallRange`
    - It is not possible to avoid collisions
- Two main approaches to deal with collisions
  - Open addressing
    - Use x2 size array
    - If a cell is already occupied, one must find another location
    - 3 options: Linear probing, Quadratic Probing, Double hashing
  - Separate Chaining (each index of the hash table associated with linked list)
- Application: Hashing Strings (Horner method and modulo operator)
Lecture 26-29: Graphs

- Graphs (non-directed/directed, weighted/unweighted) and adjacency matrix

- Graph operations:
  - 1- Depth-First Search (DFS)- using a stack
  - 2- Breadth-First Search (BFS)- using a queue
  - 3- Minimum spanning Tree (MST) (weighted)
  - 4- Shortest Path (Dijkstra) (weighted)
Lecture 26-29: Graphs- DFS

Visit: ABCDEF
Lecture 26-29: Graphs- BFS

Visit: ABCEFD
A in Tree
AB6,AD4
Dequeue → AD4

D in Tree
DE12,DC8,DB7,AB6
Dequeue → AB6

B in Tree
DE12,BC10,DC8,DB7,BE7
Dequeue → BE7

E in Tree
DE12,BC10,DC8,DB7,EF7, EC5
Dequeue → EC5

C in Tree
DE12,BC10,DC8,DB7,EF7, CF6
Dequeue → CF6

F in Tree
Done Tree {A,D,B,E,C,F}
Remark: distance to D through B is 140. However, 80<140, 80A is then retained

PATHS: B (AB), D (AD), C (ADC), E (ADCE)
Lecture 30: Epilogue

**DATA STRUCTURES**

- Array (1d, 2d, etc.)
- Linked List (simple, doubly)
- Unsorted list
- Sorted list
- Stacks
- Queues
- Trees (BST, CBT, RBT)
- Hash-Tables
- Heap (Priority Q)
- Graphs

**ALGORITHMS**

- Insert
- Remove
- Search
- Sort
- Traverse
- Pop, Push, dequeue, enqueue, etc.
<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
<th>Additional Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bubble Sort</td>
<td>$O(N^2)$</td>
<td>Slow, slow, slow</td>
</tr>
<tr>
<td></td>
<td>$O(N^2)$ comp.+ $O(N^2)$ swaps</td>
<td></td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$O(N^2)$</td>
<td>Intuitive but still slow</td>
</tr>
<tr>
<td></td>
<td>$O(N^2)$ comp.+ $O(N)$ swaps</td>
<td></td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$O(N^2)$</td>
<td>Half #comp. than Bubble</td>
</tr>
<tr>
<td></td>
<td>$O(N^2)$ comp.+ $O(N^2)$ copies</td>
<td></td>
</tr>
<tr>
<td>Enhanced Insertion Sort</td>
<td>$O(N^2)$</td>
<td>Use binary search rather than linear search</td>
</tr>
<tr>
<td></td>
<td>$O(N\log N)$ comp.+ $O(N^2)$ copies</td>
<td></td>
</tr>
<tr>
<td>List Insertion Sort</td>
<td>$O(N^2)$</td>
<td>Only 2N copies</td>
</tr>
<tr>
<td></td>
<td>$O(N^2)$ comp.+ $O(N)$ copies</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$O(N)$ copies by $O(\log N)$ levels</td>
<td></td>
</tr>
<tr>
<td>MergeSort</td>
<td>$O(N\log N)$</td>
<td>Divide &amp; Conquer + Recursive</td>
</tr>
<tr>
<td></td>
<td>$O(N)$ copies by $O(\log N)$ levels</td>
<td></td>
</tr>
<tr>
<td>ShellSort</td>
<td>$O(N(\log N)^2)$</td>
<td>'Insertion sort' using increment</td>
</tr>
<tr>
<td></td>
<td>In average</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Worst case not far from average</td>
<td></td>
</tr>
<tr>
<td>QuickSort</td>
<td>$O(N\log N)$</td>
<td>Divide and Conquer</td>
</tr>
<tr>
<td></td>
<td>Comp.&gt;swaps</td>
<td>Uses partitioning recursively</td>
</tr>
<tr>
<td>HeapSort</td>
<td>$O(N\log N)$</td>
<td>Require a heap data-structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>No worst case</td>
</tr>
</tbody>
</table>
Final Words

- Exam: Tuesday December 19- [7:50am-10am]- Marcus 131 auditorium
- Practice all algorithms, come well-prepared, do your best, no pressure
- Data Structure and Algorithms – Complement

- Hope you enjoyed the class and the projects
- This was the last ECE242 class (ever)……curriculum change
- C'est la fin, merci.