CS 61B Spring 2026 — Heaps, Tree Traversals, and Graphs

Discussion 09: March 16, 2026

1 Heaps and Graphs Summary

Heap. A min-heap is a complete binary tree where every node is smaller than its children. When used to implement a Priority Queue, then operations are as follows:

- **Add:** Add to the end of the array (i.e., bottom of the tree), then *swim up* (repeatedly swap with parent while smaller). Runtime: $O(\log N)$.
- **Remove Smallest:** Swap root with last element, remove last, then *sink down* the root (repeatedly swap with the smaller child while larger). Runtime: $O(\log N)$.

Tree Traversals. Suppose we take some action on every node of a tree, e.g. printing. Suppose current node is v, left subtree L, and right subtree R:

Pre-order:	action(v), L, R
In-order:	L, action(v), R
Post-order:	L, R, action(v)
Level-order:	Top to bottom, left to right

Graphs. A graph $G = (V, E)$ consists of vertices V and edges E. Representations:

- **Adjacency list:** For each vertex, store a list of its neighbors. Space: $\Theta(V + E)$.
- **Adjacency matrix:** $V \times V$ grid; entry $(i, j) = 1$ if edge exists. Space: $\Theta(V^2)$.

Graph Traversals.

- **DFS (Depth-First Search):** Recursively explore vertices by going as deep as possible before backtracking. Use a `marked` array to avoid infinite recursion. Runtime: $\Theta(V + E)$.

```
dfs(Graph G, int v) {
  marked[v] = true;                 // LINE 1
  for (int w : G.adj(v)) {
      if (!marked[w]) {             // LINE 2
          edgeTo[w] = v;
          dfs(G, w);
      }
  }
```

  - DFS Pre-order: The order in which DFS calls are made.
  - DFS Post-order: The order in which DFS calls complete.
- **BFS (Breadth-First Search):** Visit vertices in order of number of edges from source. Uses a **`Queue`**. Runtime: $\Theta(V + E)$. Covered on next worksheet.
- **Dijkstra's (Best-First Search):** Visit vertices in order of total distance from source (based on edge weights). Uses a **`Priority Queue`**. Runtime: $\Theta((V + E) \log V)$. Covered on next worksheet.

For some interactive demos of this week's data structures, see https://joshh.ug/61b/sp26-w9/.

2 Heap Array Drawing

Consider the following array representing a min-heap (index 0 is left blank):

```
[-, 3, 5, 7, 9, 6, 8, 12]
```

(a) Draw the min-heap that this array represents.

(b) Assume we **`insert(4)`** into our min heap. Show the resulting array after the insertion is complete.

(c) Starting from the result of part (b), we now call **`removeMin()`**. Show the resulting array after the removal is complete.

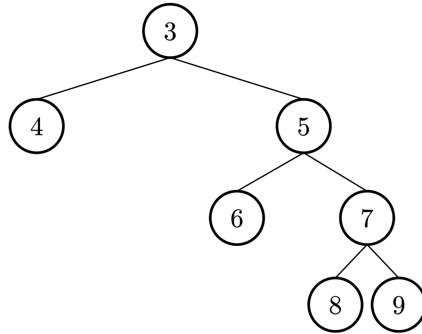
3 Heap Insert Code

Fill in the blanks to complete the **insert** method for a min-heap. The **parent** and **swap** methods are provided for you. Assume **items** is 1-indexed (index 0 is unused) and **size** tracks the number of elements. Assume the constructor initializes the instance variables.

```
public class IntHeapMinPQ {
    private int[] items;
    private int size;

    /** Returns the index of the parent of node at index k. */
    private int parent(int k) {
        return ____________________;
    }

    /** Swaps the items at indices i and j. */
    private void swap(int i, int j) { ... }

    /** Inserts item into the min-heap. */
    public void add(int item) {
        if (size == items.length) { resize(items.length * 2); }
        size = ____________________
        items[____________] = ____________
        ____________________
    }

    /** Bubbles the item at index k up to restore heap order. */
    private void swim(int k) {
        if (________________________ ||
______________________________________________________________________) {
            ________________________
        }
        swap(____________________, ____________________);
        swim(______________________________);
    }
}
```

4 Graph and Tree Traversals

Consider the following tree:



(a) Write the following tree traversals of the tree above.

Pre-order:

In-order:

Post-order:

Level-order (BFS):

(b) Now treat the tree above as an undirected graph (i.e. each edge can be traversed in both directions). Write the order in which (1) DFS pre-order, (2) DFS post-order, and (3) BFS would visit nodes, starting from vertex **5**. Break ties by choosing the smaller number.

DFS Pre-order:

DFS Post-order:

BFS:

(c) Give the adjacency list representation of the graph above.

(d) Fill in the first three rows of the adjacency matrix representation of the graph above.

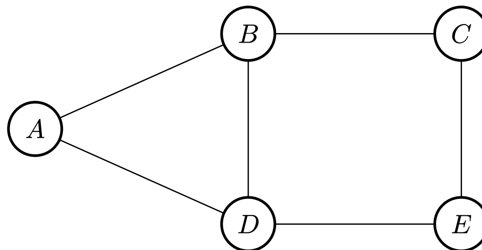
	3	**4**	**5**	**6**	**7**	**8**	**9**
3							
4							
5							

5 DFS Runtime

Consider the following implementation of depth-first search. Assume the graph is **connected** and **undirected**.

```
public class DepthFirstPaths {
    private boolean[] marked;
    private int[] edgeTo;
    private int s;

    public DepthFirstPaths(Graph G, int s) {
        ...
        dfs(G, s);
    }

    private void dfs(Graph G, int v) {
        marked[v] = true;                   // LINE 1
        for (int w : G.adj(v)) {
            if (!marked[w]) {               // LINE 2
                edgeTo[w] = v;
                dfs(G, w);
            }
        }
    }
}
```

Consider running **new DepthFirstPaths(G, A)** on the following undirected graph, where ties are broken alphabetically:



(a) How many times is LINE 1 executed? How many times is LINE 2 executed?

LINE 1: ________ LINE 2: ________

(b) Now consider a general connected undirected graph with V vertices and E edges. How many times is LINE 1 executed? How many times is LINE 2 executed? Give exact answers in terms of V and/or E.

LINE 1: __________ LINE 2: __________

Recall from our asymptotics worksheet that a *cost model* is the operation that we count as a proxy for the overall runtime of a piece of code. In our earlier asymptotic analyses, we would always pick a single operation as our cost model, e.g. number of `i += 1` calls. For example, the runtime of `contains` in a list of length `N` is $\Theta(N)$ because there are exactly N `i += 1` calls.

Things are different for graphs, and that's because there are **two different ways our input can grow**: The number of vertices and the number of edges.

For DFS, a typical cost model is the sum of LINE 1 and LINE 2 calls together. That is, if we define $C(E, V)$ as the sum of the number of LINE 1 and LINE 2 calls, then $C(E, V) = V + 2E$.

Let $R(E, V)$ be the runtime of DFS as a function of the number of edges and vertices. In this problem we'll explore why $R(E, V) \in \Theta(E + V)$.

(c) i. Give an example of a family of graphs for which $C(E, V) \in \Theta(V)$, as the graph grows and grows, the vertices dominate. One way to solve this is to describe a type of graph where LINE 1 gets called more times than LINE 2.

ii. Give an example of a family of graphs for which $C(E, V) \in \Theta(E)$, as the graph grows and grows, the edges dominate. One way to solve this is to describe a type of graph where LINE 2 gets called more times than LINE 1.

iii. Explain why the total can *always* be correctly written as $\Theta(V + E)$, regardless of the structure of the graph.

6 Extra: Graph Conceptuals

(a) Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n - 1$ edges, and the graph is connected, it **must** be a tree.

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

3. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

(b) Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.