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.

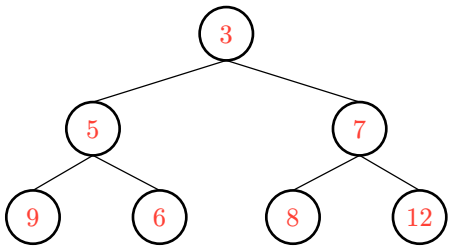
For some interactive demos of this week's data structures, see https://joshh.ug/61b/sp26-w9/.

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.



Recall that for a node at index k, its left child is at index $2k$ and its right child is at index $2k+1$. So the root (index 1) is 3, its children (indices 2 and 3) are 5 and 7, and so on.

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

First, place 4 at the next open position (index 8):

`[-, 3, 5, 7, 9, 6, 8, 12, 4]`

4 is at index 8. Its parent is at index 4 (value 9). Since $4 < 9$, swim up (swap):

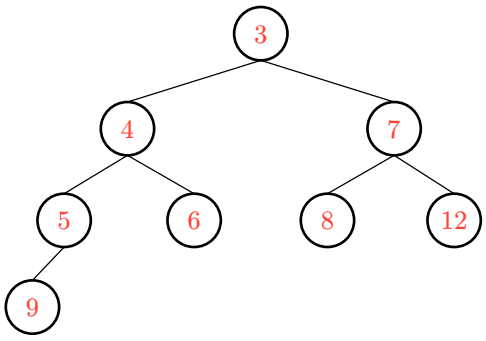
`[-, 3, 5, 7, 4, 6, 8, 12, 9]`

Now 4 is at index 4. Its parent is at index 2 (value 5). Since $4 < 5$, swim up (swap):

`[-, 3, 4, 7, 5, 6, 8, 12, 9]`

Now 4 is at index 2. Its parent is at index 1 (value 3). Since $4 > 3$, we stop.

Final array: `[-, 3, 4, 7, 5, 6, 8, 12, 9]`



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

Remove the root (3) and replace it with the last element (9):

`[-, 9, 4, 7, 5, 6, 8, 12]`

Sink 9 down. Its children are 4 (index 2) and 7 (index 3). The smaller child is 4. Since $9 > 4$, swap:

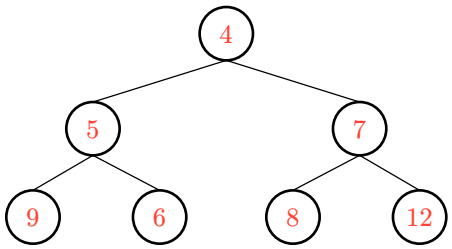
`[-, 4, 9, 7, 5, 6, 8, 12]`

Now 9 is at index 2. Its children are 5 (index 4) and 6 (index 5). The smaller child is 5. Since $9 > 5$, swap:

`[-, 4, 5, 7, 9, 6, 8, 12]`

Now 9 is at index 4. Its children would be at indices 8 and 9, which are out of bounds. We stop.

Final array: `[-, 4, 5, 7, 9, 6, 8, 12]`



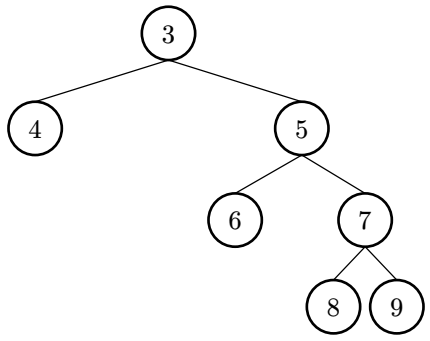
3 Heap Insert Code

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 k / 2;

    }

    /** 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 = size + 1;

        items[size] = item;

        swim(size);

    }

    /** Bubbles the item at index k up to restore heap order. */
    private void swim(int k) {

        if (k <= 1 || items[k] >= items[parent(k)]) {

            return;

        }

        swap( k , parent(k));

        swim(parent(k));

    }
}
```

4 Graph and Tree Traversals

Consider the following tree:



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

Pre-order:

3 4 5 6 7 8 9

In-order:

4 3 6 5 8 7 9

Post-order:

4 6 8 9 7 5 3

Level-order (BFS):

3 4 5 6 7 8 9

(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:

5 3 4 6 7 8 9

DFS Post-order:

4 3 6 8 9 7 5

BFS:

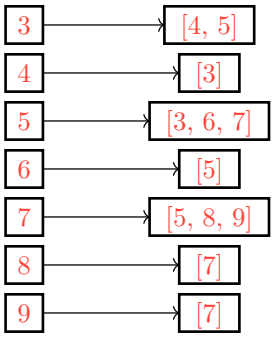
5 3 6 7 4 8 9

Unlike tree traversals, graph traversals require a visited set to avoid revisiting nodes. Starting from 5, the neighbors are {3, 6, 7}. We pick the smallest unvisited neighbor at each step.

For DFS, we go deep before backtracking: 5 → 3 → 4 (backtrack to 5) → 6 (backtrack) → 7 → 8 (backtrack) → 9. Pre-order records when we first visit a node; post-order records when we finish (backtrack from) a node.

For BFS, we process nodes level-by-level from 5: first 5, then its neighbors {3, 6, 7}, then their unvisited neighbors {4, 8, 9}.

(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							

Solution:

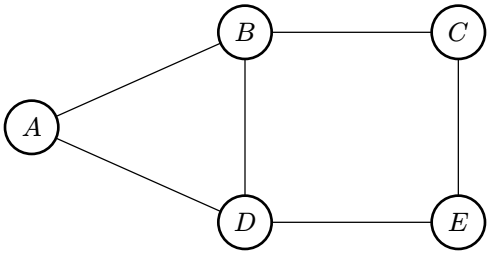
	3	4	5	6	7	8	9
3	0	1	1	0	0	0	0
4	1	0	0	0	0	0	0
5	1	0	0	1	1	0	0

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: _____

Solution:

LINE 1 is executed **5 times** — once per call to `dfs`, which is called exactly once per vertex.

LINE 2 is executed **12 times**. Each call to `dfs(G, v)` iterates over all neighbors of `v`:

- `dfs(G, A)`: check B, D → 2 calls
- `dfs(G, B)`: check A, C, D → 3 calls
- `dfs(G, C)`: check B, E → 2 calls
- `dfs(G, E)`: check C, D → 2 calls
- `dfs(G, D)`: check A, B, E → 3 calls

Total: $2 + 3 + 2 + 2 + 3 = 12$

(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: _____

Solution:

LINE 1 is executed exactly V times — `dfs` is called exactly once per vertex (a vertex is only recursed into when `!marked[w]` is true, and once marked it is never unmarked).

LINE 2 is executed exactly $2E$ times. Each call `dfs(G, v)` iterates over all neighbors of `v`, executing the `if` check once per neighbor. Across all vertices, the total number of neighbor checks equals the sum of all vertex degrees. By the handshake lemma, the sum of degrees in an undirected graph is $2E$.

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.

Solution:

i. There are many solutions. Consider a graph which is just a straight line of vertices, i.e. A - B - C - D - … - Z, which has $E = V - 1$. More generally consider any graph which is a **tree**, which has the property that $E = V - 1$. For such graphs $C(E, V) = V + 2(V - 1) = 3V - 2 = \Theta(V)$.

ii. There are many solutions. One is a **complete graph**, which has the maximum number of edges: $E = V(V - 1)/2$. The total is $V + 2 \cdot V(V - 1)/2 = V^2 = \Theta(V^2) = \Theta(E)$. The $2E$ term dominates since E is much larger than V.

iii. The total is always exactly $V + 2E = \Theta(V + E)$. This expression correctly captures the runtime for **all** graphs because it separately accounts for the vertex processing (V) and edge checking ($2E$) without assuming one dominates the other:
- For sparse graphs ($E = O(V)$): $\Theta(V + E) = \Theta(V)$
- For dense graphs ($E = \Omega(V)$): $\Theta(V + E) = \Theta(E)$

$\Theta(V + E)$ handles both extremes and everything in between.

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.

   Solution: True. First we remind ourselves that a tree is just a graph without any cycles. Then we consider what a graph with n vertices and n - 1 edges looks like. As an example, suppose we have 5 nodes that are initially disconnected. Suppose we add 4 edges, connecting a new node each time. At this point the graph is fully connected. However there are no cycles (since we can only get a cycle by adding edge between two vertices that are already connected).

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

   Solution: True. Say an edge connects u and v. Both u and v will look at the other one through this edge when it's their turn.

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.

   Solution: True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

(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.

Solution: We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle.

However, since the graph is undirected, if an edge connects vertices `u` and `v`, then `u` is a neighbor of `v`, and `v` is a neighbor of `u`. As such, if we visit `v` after `u`, our algorithm will claim that there is a cycle since `u` is a visited neighbor of `v`. To address this case, when we visit the neighbors of `v`, we should ignore `u`. To implement this in code, we could add the parent as another parameter in the method call. In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.

Pseudocode is provided below (for a disconnected graph, we should call `find_cycle` on each component).

```
find_cycle(v, parent=-1):
    visited[v] = true
    for (v, w) in G:
        if !visited[w]:
            if find_cycle(w, v):
                return True
        else if w != parent:
            return True
    return False
```