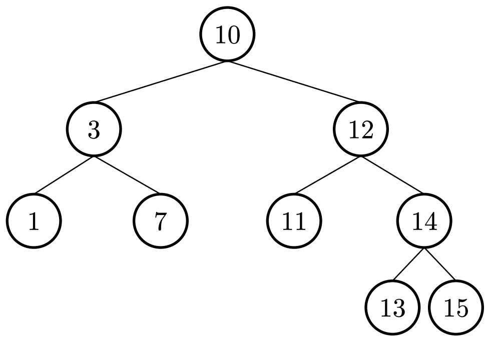
CS 61B
Spring 2025

Graphs, Heaps
Discussion 08: March 17, 2025

1 Trees, Graphs, and Traversals, Oh My!

(a) Write the following traversals of the BST below.

Pre-order:
In-order:
Post-order:
Level-order (BFS):



Solution:

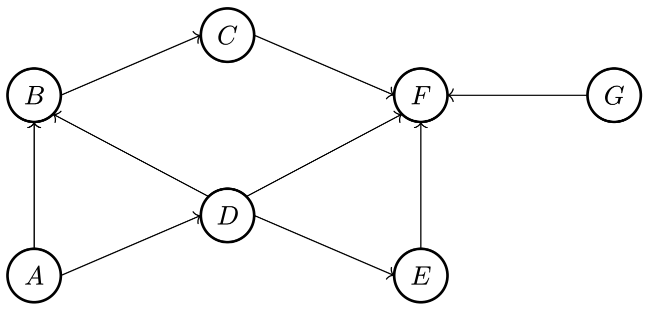
Pre-order: 10 3 1 7 12 11 14 13 15

In-order: 1 3 7 10 11 12 13 14 15

Post-order: 1 7 3 11 13 15 14 12 10

Level-order (BFS): 10 3 12 1 7 11 14 13 15

(b) Write the graph below as an adjacency matrix, then as an adjacency list. What would be different if the graph were undirected instead?



Solution:
Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 0 0 1 0 0 0 0
C 0 0 0 0 0 1 0
D 0 1 0 0 1 1 0
E 0 0 0 0 0 1 0
F 0 0 0 0 0 0 0
G 0 0 0 0 0 1 0
^ start node
```

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

For the undirected version of the graph, the representations look a bit more symmetric. For your reference, the representations are included below:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 1 0 1 1 0 0 0
C 0 1 0 0 0 1 0
D 1 1 0 0 1 1 0
E 0 0 0 1 0 1 0
F 0 0 1 1 1 0 1
G 0 0 0 0 0 1 0
^ start node
```

List:

```
A: {B, D}
B: {A, C, D}
C: {B, F}
D: {A, B, E, F}
E: {D, F}
F: {C, D, E, G}
G: {F}
```

(c) Write the order in which (1) DFS pre-order, (2) DFS post-order, and (3) BFS would visit nodes in the same directed graph above, starting from vertex A. Break ties alphabetically.

Pre-order:

ABCFDE (G)

Post-order:

FCBEDA (G)

BFS:

ABDCEF (G)

To compute DFS, we maintain a stack of nodes, and a visited set. As soon as we add something to our stack, we note it down for preorder. The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited. After we add a node to the stack, we visit its lexicographically next unmarked child. If there is none, we pop the topmost node from the stack and note it down for postorder. *Note that there are two ways DFS could run: with restart or without; DFS with restart is the version where if we have exhausted our stack, and still have unmarked nodes left, we restart on the next unmarked node.*

Stack (bottom-top)	VisitedSet	Preorder	Postorder
A	{A}	A	-
AB	{AB}	AB	-
ABC	{ABC}	ABC	-
ABCF	{ABCF}	ABCF	-
ABC	{ABCF}	ABCF	F
AB	{ABCF}	ABCF	FC
A	{ABCF}	ABCF	FCB
AD	{ABCFD}	ABCFD	FCB
ADE	{ABCFDE}	ABCFDE	FCB
AD	{ABCFDE}	ABCFDE	FCBE
A	{ABCFDE}	ABCFDE	FCBED
-	{ABCFDE}	ABCFDE	FCBEDA

If DFS restarts on unmarked nodes, the following happens in the last line. Otherwise, we do not proceed further.

Stack (bottom-top)	VisitedSet	Preorder	Postorder
G	{ABCFDEG}	ABCFDEG	FCBEDAG

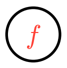
For BFS, we use a queue instead of a stack. BFS does not have the notion of in-order and post-order, so we only visit it when we remove it from the queue.

2 Absolutely Valuable Heaps

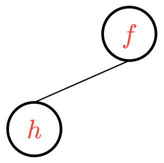
(a) Assume that we have a binary min-heap (smallest value on top) data structure called **`MinHeap`** that has properly implemented the **`insert`** and **`removeMin`** methods. Draw the heap and its corresponding array representation after each of the operations below:

```
MinHeap<Character> h = new MinHeap<>();
h.insert('f');
h.insert('h');
h.insert('d');
h.insert('b');
h.insert('c');
h.removeMin();
h.removeMin();
```

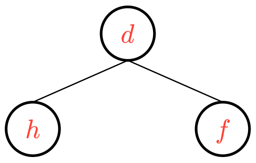
after inserting ‘f’: `[-, 'f']`



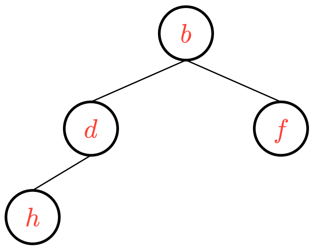
after inserting ‘h’: `[-, 'f', 'h']`



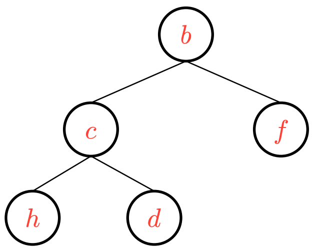
after inserting ‘d’: `[-, 'd', 'h', 'f']`



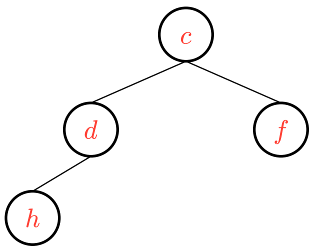
after inserting ‘b’: `[-, 'b', 'd', 'f', 'h']`



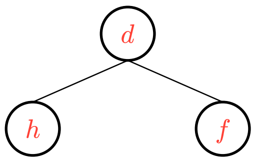
after inserting ‘c’: `[-, 'b', 'c', 'f', 'h', 'd']`



after removing min: `[-, 'c', 'd', 'f', 'h']`



after removing min: `[-, 'd', 'h', 'f']`



(b) Your friendly TA Mihir challenges you to create an integer max-heap without writing a whole new data structure. Can you use your min-heap to mimic the behavior of a max-heap? Specifically, we want to be able to get the largest item in the heap in constant time, and add things to the heap in $\Theta(\log n)$ time, as a normal max heap should.

Hint: You should treat the **`MinHeap`** as a black box and think about how you should modify the arguments/ return values of the heap functions.

Yes. For every insert operation, negate the number and add it to the min-heap.

For a **`removeMax`** operation call **`removeMin`** on the min-heap and negate the number returned. Any number negated twice is itself, and since we store the negation of numbers, the order is now reversed (what used to be the max is now the min).

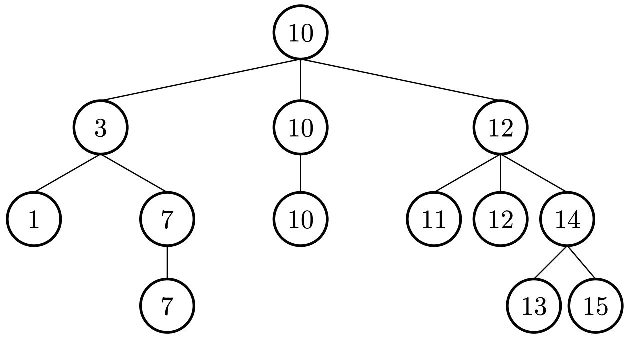
Small note: There's actually one exception in Java to what we said about negation above: -2^{-31}, the most negative number that we can represent in Java, will not be itself when negated twice. This is mostly due to number representation constraints in code, but you don't need to worry about that for this question.

3 Trinary Search Tree

We'd like a data structure that acts like a BST (Binary Search Tree) in terms of operation runtimes but allows duplicate values. Therefore, we decide to create a new data structure called a TST (Trinary Search Tree), which can have up to three children, which we'll refer to as **`left`**, **`middle`**, and **`right`**. In this setup, we have the following invariants, which are very similar to the BST invariants:

1. Each node in a TST is a root of a smaller TST
2. Every node to the **`left`** of a root has a value "lesser than" that of the root
3. Every node to the **`right`** of a root has a value "greater than" that of the root
4. **Every node to the `middle` of a root has a value equal to that of the root**

Below is an example TST to help with visualization.



Describe an algorithm that will print the elements in a TST in **descending** order. *(Hint: recall that an in-order traversal for a BST gives elements in increasing order.)*

Solution:
Inorder traversal on a BST yields the sorted elements in the BST in ascending order. Therefore, the core of the algorithm we'd like here is going to be quite similar to inorder traversal, but reversed (visit the right child before the left child) and with the added caveat that we also must traverse through the middle children.

In essence, given the root of some TST, we reverse onto the right child subtree, then print the root's value, then reverse onto the middle child subtree, then finally reverse onto the left subtree. The print root value and reverse onto the middle child steps can be swapped, because overall the order of the printed values should be the same.

Pseudocode:

```
reverse(tst):
    if tst is null:
        return
    reverse(tst.right)
    print(tst.value)
    reverse(tst.middle)
    reverse(tst.left)
```