CS 61BL Summer 2026

BSTs, Midterm Review

Reciation 02: July 13th, 2026

1 BST Asymptotics

Below we define the **find** method of a BST (Binary Search Tree) as in lecture, which returns the BST rooted at the node with key **sk** in our overall BST. In this setup, assume a **BST** has a **key** (the value of the tree root) and then pointers to two other child BSTs, **left** and **right**.

```
public static BST find(BST tree, Key sk) {
    if (tree == null) {
        return null;
    }
    if (sk.compareTo(tree.key) == 0) {
        return tree;
    } else if (sk.compareTo(tree.key) < 0) {
        return find(tree.left, sk);
    } else {
        return find(tree.right, sk);
    }
}
```

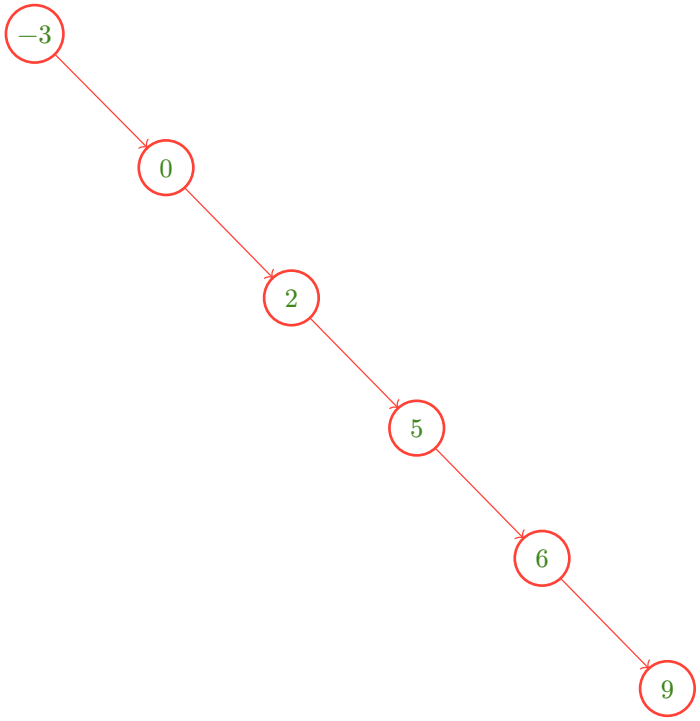
(a) Assume our BST is perfectly bushy. What is the runtime of a single **find** operation in terms of N, the number of nodes in the tree? Can we generalize the runtime of **find** to a theta bound?

Find operations on a perfectly bushy BST take $O(\log(N))$ time, as the height of a perfectly bushy BST is $\log(N)$. In the worst case scenario, the key we're looking at is all the way at a leaf, so we have to traverse a path from root to leaf of length $\log(N)$.

We cannot generalize the runtime of **find** to a theta bound because the lower and upper bounds are different. It is lower-bounded by $\Omega(1)$ (the case where the key we are looking for is at the root of the BST provided) and upper-bounded by $O(\log(N))$ (mentioned above as the path from root to leaf). Therefore, there is no theta bound for **find**.

(b) Say we have an empty BST and want to insert the keys **[6, 2, 5, 9, 0, -3]** (in some order). In what order should we insert the keys into the BST such that the runtime of a single **find** operation after all keys are inserted is $O(N)$? Draw out the resulting BST.

We should insert the keys in ascending sorted order: `[-3, 0, 2, 5, 6, 9]`. This results in a perfectly linear BST, which means that the longest path for `find` (from root to leaf) traverses every single node in the BST. The resulting BST looks like a direct chain of nodes:



Alternatively, we could insert the keys in descending sorted order, which also results in a perfectly linear BST (but the keys would chain left from $9 \to 6 \to 5 \to 2 \to 0 \to -3$).

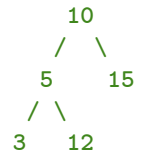
2 Is This a BST?

In this setup, assume a **BST** (Binary Search Tree) has a **key** (the value of the tree root represented as an **int**) and pointers to two other child BSTs, **left** and **right**. Additionally, assume that **key** is between **Integer.MIN_VALUE** and **Integer.MAX_VALUE** non-inclusive.

(a) The following code should check if a given binary tree is a BST. However, for some trees, it returns the wrong answer. Give an example of a binary tree for which **brokenIsBST** fails.

```
public static boolean brokenIsBST(BST tree) {
    if (tree == null) {
        return true;
    } else if (tree.left != null && tree.left.key >= tree.key) {
        return false;
    } else if (tree.right != null && tree.right.key <= tree.key) {
        return false;
    } else {
        return brokenIsBST(tree.left) && brokenIsBST(tree.right);
    }
}
```

Here is an example of a binary tree for which **brokenIsBST** fails:



The method fails for some binary trees that are not BSTs because it only checks that the value at a node is greater than its left child and less than its right child, not that its value is greater than every node in the left subtree and less than every node in the right subtree. Above is an example of a tree for which it fails.

It is important to note that the method does indeed return true for every binary tree that actually is a BST (it correctly identifies proper BSTs).

(b) Now, write **isBST** that fixes the error encountered in part (a).

Hint: You will find **Integer.MIN_VALUE** and **Integer.MAX_VALUE** helpful.

Hint 2: You want to somehow store information about the keys from previous layers, not just the direct parent and children. How do you use the parameters given to do this?

```
public static boolean isBST(BST T) {
    return isBSTHelper(T, Integer.MIN_VALUE, Integer.MAX_VALUE);
}

public static boolean isBSTHelper(BST T, int min, int max) {

    if (T == null) {
        return true;
    } else if (T.key <= min || T.key >= max) {
        return false;
    } else {
        isBSTHelper(T.left, min, T.key) && isBSTHelper(T.right, T.key, max);
    }
}
```

Solution: A BST is a naturally recursive structure, so it makes sense to use a recursive helper to go through the BST and ensure it is valid. Specifically, our recursive helper will traverse the BST while tracking the minimum and maximum valid values for subsequent nodes along our current path. We can get these minimum and maximum values by remembering the key property of BSTs: nodes to the left of our current node are always less than the current value, and nodes to the right of our current node are always greater than our current value. So for example, if we encounter a node with value 5, anything to the left must be < 5.

In our base case, an empty BST is always valid. Otherwise, we can check the current node. If it doesn't fall within our precomputed min/max, we know this is invalid, and return immediately.

Otherwise, we use the properties of BSTs to bound our subsequent min and max values. If we traverse to the left, everything must be less than or equal to the current value, so the value of our current node becomes the new `max` for the tree at `T.left`. Similar logic applies to the right.

3 Transpose

The transpose of a 2D array is a new 2D array whose rows are the columns of the original (and vice versa). In an upper-triangular 2D array, you are guaranteed that every row has exactly **one** less element than the row above it. The last row is guaranteed to have one element. For example, let `A` be the 2D array below on the left. The transpose of `A` is the resulting 2D array below on the right.

Extra: What if we were only guaranteed that the length of each row is less than or equal to the size of the row above it (not always 1 smaller)? What would you change about the algorithm?

$$A = \begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 \\ \cline{1-3} 8 & 9 \\ \cline{1-2} 10 \\ \cline{1-1}\end{array} \implies \begin{array}{|c|c|c|c|}\hline 1 & 5 & 8 & 10 \\ \hline 2 & 6 & 9 \\ \cline{1-3} 3 & 7 \\ \cline{1-2} 4 \\ \cline{1-1}\end{array} = transposeTriangular(A)$$

Given an upper-triangular 2D array `A`, non-destructively transpose `A`.

```
// Non-destructively transposes A. Assume that A is upper-triangular.
public static int[][] transposeTriangular(int[][] A) {

    if (A.length == 0) {
        return new int[0][];
    }

    int[][] transpose = new int[A[0].length][];

    for (int i = 0; i < transpose.length; i++) {
        int rowLength = A.length - i;
        transpose[i] = new int[rowLength];

        for (int j = 0; j < rowLength; j++) {
            transpose[i][j] = A[j][i];
        }
    }

    return transpose;
}
```

Here is a video walkthrough of the solutions.

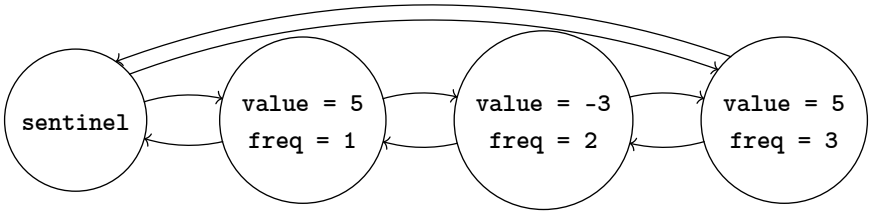
4 Compressed Doubly-Linked List

from Spring 2025 Midterm 1

Creating new nodes for consecutive identical elements in a doubly-linked list is inefficient! To combat this, 61B Inc. has created a new data structure called the Compressed Doubly-Linked List (`CDLL`).

Each node in a `CDLL` has a next and prev pointer, along with parameters value and freq, and represents a sequence of freq consecutive elements equal to value.

For example, a `CDLL` representing the list `[5, -3, -3, 5, 5, 5]` would look like:



Except for the sentinel, a `CDLL` may not have two consecutive nodes with the same value, and may not have nodes with freq less than 1. The class is defined as follows:

```
public class CDLL implements Iterable<Integer> {
  private Node sentinel;
  private class Node {
    Node prev;
    Node next;
    int value;
    int freq;
    public Node (Node prev, Node next, int value, int freq) {
      this.prev = prev;
      this.next = next;
      this.value = value;
      this.freq = freq;
    }

    public CDLL(List<Integer> original) {
    // part a
    }

    @Override
    public Iterator<Integer> iterator() {
      return new CDLLIterator();
    }
    private class CDLLIterator implements Iterator<Integer> {
    // part b
    }
  }
```

(a) Fill out the constructor of the CDLL class, which takes in a List as an argument and converts it into its corresponding CDLL representation. You may not need all lines.

```
public CDLL(List<Integer> original) {

  sentinel = new Node(null, null, -1, -1);
  Node curr = sentinel;

  for (int i:original) {
    if (i == curr.value&&curr != sentinel) {
      curr.freq += 1;
    } else {
      Node n = new Node(curr, sentinel, i, 1);
      curr.next = n;
      curr = curr.next;
    }
  }
  sentinel.prev = curr;
  curr.next = sentinel;
}
```

Note that due to autounboxing, both int and Integer are interchangeable. This solution adds new items to the next, moving the curr pointer. An alternate solution attempts to add new items to the prev of the sentinel, without using the curr pointer. This doesn't quite work, since the sentinel starts with two null pointers, instead of pointers to itself. However, this alternate solution was sufficiently close that it was granted 3/4 credit (with more for attempts to address the null pointer).

Alternate solutions:

- Other ways to determine if curr!=sentinel. The two blanks in the if clause can be swapped. Note that curr!=sentinel is necessary to avoid the edge case where the first item in original is -1.
- The 3 lines in the else condition can be condensed into 2 by immediately assigning curr.next to the new node.
- The new node's next can also be null. This is because each node's next pointer is either reassigned when an additional node is created or after the loop ends.

(b) Complete the CDLLIterator class, whose iteration returns the values of the CDLL in the order they would appear in a regular List. Your iterator may not modify the CDLL. You may not need all lines.

```
public CDLLIterator implements Iterator<Integer> {
  Node curr = sentinel.next;
  int count = 0;

  @Override
  public boolean hasNext() {
    return curr != sentinel;
  }

  @Override
  public Integer next() {
    int i = curr.value;
    count += 1;
    if (count == curr.freq) {
      curr = curr.next;
      count = 0;
    }
    return i;
  }
}
```

5 Flip Flop

Suppose we have the flip function as defined below. Assume the method unknown returns a random integer between 1 and N, exclusive, and runs in constant time. For each definition of the flop method below, give the best and worst case runtime of flip in big Theta notation as a function of N.

```
public static void flip(int N) {
  if (N <= 100) {
    return;
  }
  int stop = unknown(N);
  for (int i = 1; i < N; i++) {
    if (i == stop) {
      flop(i, N);
      return;
    }
  }
}
```

(a)
```
public static void flop(int i, int N) {
  flip(N - i);
}
```

Best case: $\Theta(N)$	Worst case: $\Theta(N)$

Consider some arbitrary value of stop that is always returned by `unknown(N)`. When `stop = x`, we do x work inside of flip (the for loop) and recursively call `flip(N - x)` through flop. This results in a total of $\frac{N}{x}$ calls before reaching our base case, and x work per call, for a total of $\Theta(N)$ work. Note that this holds for any value of x, so our best and worst case are the same.

(b)
```
public static void flop(int i, int N) {
  int minimum = Math.min(i, N - i);
  flip(minimum);
  flip(minimum);
}
```

Best case: $\Theta(1)$	Worst case: $\Theta(N \log N)$

In the best case, `unknown` always returns 1, so `stop = 1`. T
2 calls to flip then stop for $\Theta(1)$ work.

In the worst case, `stop = N / 2`. This results in flip maki
`N / 2`. If one were to draw out the recursive tree, there wou
work. This results in a runtime of $\Theta(N \log N)$.

You might wonder why we analyze best-worst case in terms
as an input to unknown, and we don't care about input siz
the value of stop, but notice that the behavior of the functi
latter is what we care about in asymptotic analysis.

(c)
```
public static void flop(int i, int N) {
  flip(i);
  flip(N - i);
}
```

Best case: $\Theta\ (\ N\ )$	Worst case: $\Theta\ (\ N^2\ )$

In the best case, suppose stop = 1. Then flip(N) makes recursive calls to `flip(1)` and `flip(N - 1)`, the first of which terminates immediately in the base case. `flip(N - 1)` then calls `flip(1)` and `flip(N - 2)`. The pattern is a linear recursion: constant work per call, N calls total for $\Theta(N)$ work.

In the worst case, suppose `stop = N - 1`. Note that this case is symmetrical to the best case in terms of recursive calls; however we do work proportional to N inside of flip each time because of the for loop. The overall work is $(N-1)+(N-2)+(N-3)+...+2+1=\Theta(N^2)$