CS61b

This memo is based on CS61b Spring 2025, by UC Berkeley. I have previously taken CS50x and CS61a, also C and C++ programming course in my freshman year.

Instructors: Josh Hug, Justin Yokota

Lec 1: Introduction

Java programs are quite like C or C++ programs.

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello, World!");
    }
}
```

Java emphasizes Object-Oriented Programming. Everything (including the `main` function) must be inside a class.

The filename should be the same as the class name.

To run a Java Program in a terminal:

```
$ javac HelloWorld.java # Complile and generate HelloWorld.class
$ java HelloWorld # Execute the *.class file
Hello, World!
```

Lec 2: Defining and Using Classes

Quite similar to C++, but several points to notice:

- `this` pointer no long needs dereferencing : no `this->length` or `*this`.
- Use `new` to instantiate objects. e.g. `Scanner sc = new Scanner(System.in);`

There are two kinds of methods in a class:

- Class Method, aka Static Method
- Instance Method, aka Non-static Method

For example, `sqrt` is a static method:

```
x = Math.java(100);
```

Otherwise, we will have an embarrassing syntax:

```
Math m = new Math(); // What the hell does this mean lol
x = m.sqrt(100);
```

Static members belong to the **class**, while instance members belong to **objects**.

Lec 3: Primitive Types, Reference Types, and Linked Data Structures

There are **8 primitive types** in Java: `byte`, `short`, `int`, `long`, `float`, `double`, `boolean`, `char`.

Everything else, including arrays, are **reference types** (an arrow in box-and-pointer notation).

When declaring a variable of any reference type (`Walrus`, `Dog`, `Car`) :

- Java allocates exactly a box of size **64 bits**, no matter what type of object.
- These bits can be either set to `null` or the 64-bit **address** returned by `new` keyword.

The Golden Rule of Equals (the GRoEs)

- `y = x` copies all the bits from `x` to `y`.
- Passing parameter obeys the same rule: Simply copy the bits to the new scope.

We can think of the `new` keyword as returning the **address** of the newly created object.

Below is an example of declaration, instantiation, and assignment:

```
int[] a = new int[]{0, 1, 2, 95, 4};
// Declaration: int[] a
// Instantiation: new int[]{0, 1, 2, 95, 4}
// Assignment: =
```

IntList

```
public class IntList {
    public int first;
    public IntList rest;
    public IntList(int f, IntList r) {
        first = f;
        rest = r;
    }
    public int recursive_size() {
        if (rest == null) {
            return 1;
        }
        return 1 + rest.recursize_size();
    }

    public int iterative_size() {
        int size = 0;
        IntList p = this;
        while (p != null) {
            p = p.rest;
            size++;
        }
        return size;
    }

    public int get(int i) {
```

```
        if (i == 0) {
            return first;
        }
        return rest.get(i - 1);
    }

    public static void main(String[] args) {
        // To create 5 -> 15 -> 20:
        // Adding elements from front to back:
        IntList L = new IntList(5, null);
        L.rest = new IntList(15, null);
        L.rest.rest = new IntList(20, null);

        // Adding elements from back to front:
        IntList L= new IntList(20, null);
        L = new IntList(15, L);
        L = new IntList(5, L);
    }
}
```

Lec 4: SLLists, Nested Classes, Sentinel Nodes

IntNodes

Last time, our IntList is considered a naked data structure.

```
public class IntNode {
    public int item;
    public IntNode next;
    public IntNode(int n, IntNode p) {
        item = n;
        next = p;
    }
}
```

```
public class SLList {
    private IntNode first;
    public SLList(int x) {
        first = new IntNode(x, null);
    }
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }
    public int getFirst() {
        return first.item;
    }
    public static void main(String[] args) {
        SLList y = new SLList(10);
        // SLList is easier to use since we don't need to specify null
        // C.f: IntList y = new IntList(10, null);
    }
```

```
}
```

By restricting access using `private` and `public`, we hide implementation details from users.

In fact, `IntNode` is only a feature for our `SLList`. Therefore, we could make a **nested class**.

```
public class SLList {

    private static class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int n, IntNode p) {
            item = n;
            next = p;
        }
    }

    private IntNode first;

    public SLList(int x) {
        first = new IntNode(x, null);
    }
    public void addFirst(int x) {
        first = new IntNode(x, first);
    }
    public int getFirst() {
        return first.item;
    }
    public static void main(String[] args) {
        SLList y = new SLList(10);
    }

}
```

By declaring the class `IntNode` as `static`, the instance variables inside cannot access elements of the outer class, and we ensure that `IntNode` can exist **independently** without relying on an instance of the outer class `SLList`.

```
public class SLList {
    ....
    public void addLast(int x) {
        IntNode p = first;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }
    public int size() {
        return size(first);
    }

    // Returns the size of the list, starting at IntNode p, as a helper function.
    private int size(IntNode p) {
        if (p.next == null) {
```

```
            return 1;
        }
        return 1 + size(p.next);
    }
}
```

To implement a recursive method in a class that is not itself recursive, create a private recursive helper method, and have the public method call the private recursive helper method.

The `size` function we have is slow. We can maintain a special `size` variable that **caches** the size of the list.

```
public class SLList {
    ...
    private int size;
    public SLList(int x) {
        first = new IntNode(x, null);
        size = 1;
    }
    public void addFirst(int x) {
        ...
        size++;
    }
    ...
    public int size() {
        return size;
    }
}
```

How about we create an empty list?

```
public class SLList {
    public SLList() {
        size = 0;
        first = null;
    }
}
```

But it will cause a bug. We can fix it:

```
public void AddLast(int x) {
    size++;
    IntNode p = first;
    if (first == null) {
        first = new IntNode(x, null);
        return;
    }
    ...
}
```

Which is really ugly. Adding this special case to all methods is not good.

We can create a special node that is always there! Let's call it a '"sentinel code".

The first item, if it exists, is at `sentinel.next`.

Invariants

An invariant is a condition that is always true during code execution.

Below are some invariants in the `SLList` class:

- The sentinel reference always points to the sentinel node
- The first code (if it exists), is always at `sentinel.next`
- The `size` variable is always the total number of items that have been added

The full code goes below.

```
public class SLList {
    private static class IntNode {
        public int item;
        public IntNode next;
        public IntNode(int i, IntNode r) {
            item = i;
            next = r;
        }
    }
    public IntNode sentinel;
    public int size;

    public SLList() {
        sentinel = new IntNode(0, null);
        size = 0;
    }

    public SLList(int x) {
        sentinel = new IntNode(0, null);
        sentinel.next = new IntNode(x, null);
        size = 1;
    }

    public void addFirst(int x) {
        IntNode first = new IntNode(x, sentinel.next);
        sentinel.next = first;
        size++;
    }

    public int getFirst() {
        return sentinel.next.item;
    }

    public void addLast(int x) {
        size++;
        IntNode p = sentinel;
        while (p.next != null) {
            p = p.next;
        }
        p.next = new IntNode(x, null);
    }
```

```
}
```

Lec 5: DDLists and Arrays

We can have a doubly linked list (DDList), as opposed to our previous singly linked list (SLList).

We can have `size`, `sentinel_front`, `sentinel_back`, `last` variables.

Or, with only one `sentinel` but make the list a loop. Make it both the beginning and the end of the list.

DDLists will be implemented in Project 1A.

Generic Lists

```
public class SLList<T> {
    private class TNode {
        public T item;
        public TNode next;
        public TNode(T i, TNode r) {
            item = i;
            next = r;
        }
    }
    public TNode sentinel;
    public int size;

    public SLList() {
        sentinel = new TNode(null, null);
        size = 0;
    }

    public SLList(T x) {
        sentinel = new TNode(0, null);
        sentinel.next = new TNode(x, null);
        size = 1;
    }

    public void addFirst(T x) {
        TNode first = new TNode(x, sentinel.next);
        sentinel.next = first;
        size++;
    }

    public T getFirst() {
        return sentinel.next.item;
    }
    ...
    public static void main(String[] args) {
        SLList<Integer> L = new SLList<>();

    }
}
```

- Write out desired type during declaration, but use the empty diamond operator `<>` during instantiation.
- When declaring or instantiating your data structure, use the **reference type** (as opposed to the primitive type):
  - `int` : `Integer`
  - `double` : `Double`
  - `char` : `Character`
  - `boolean` : `Boolean`
  - `long` : `Long`
  - etc.

Arrays

Arrays are a special kind of object which consists of a **numbered** (i.e. the index) sequence of memory boxes. Arrays are fixed-size.

Unlike class instances which have **named** (i.e. the variable names) memory boxes.

Unlike classes, arrays do not have methods.

```
int[] x = new int[3]; // 3 int boxes, 96 bits in total, default value 0
int[] y = new int[]{1, 2, 3, 4, 5};
int[] z = {9, 10, 11, 12, 13};
```

There is an old `System.arraycopy()` function, which takes 5 parameters:

- Source array
- Start position in source
- Target array
- Start position in target
- Number to copy

A 2D array is really a 1D array of references to 1D arrays.

Arrays vs. Classes

Arrays and Classes can both be used to organize a bunch of memory boxes.

- Array boxes are accessed using `[]` notation.
- Class boxes are accessed using dot notation.
- Array boxes must all be of the same type.
- Class boxes may be of different types.
- Both have a fixed number of boxes.

Array indices can be computed at runtime.

Lec 6: Testing

The most natural approach to writing a test is to start with an input and expected result.

In computer programming, unit testing is a software testing method by which individual units of source code are tested to determine whether they are fit to use.

Let's try writing a unit test using `Truth`.

```
import static com.google.common.truth.Truth.assertThat;
import org.junit.jupiter.api.Test;

public class TestSort {
    @Test // annotation
    public void testSort() { // Non-static
        String[] input = {"cat", "luca", "cpp", "bob"};
        String[] expected = {"bob", "cat", "cpp", "luca"};
        Sort.sort(input);
        assertThat(input).isEqualto(expected);
    }
    // No need for the main method
}
```

Let's try selection sort:

- Find the smallest item
- Move it to the front (move by swapping the smallest item with the front item)
- Selection sort the remaining N-1 items without touching front item

Through Test-driven development:

```
public class TestSort {
    @Test // annotation
    public void testFindSmallest() {
        String[] input = {"cat", "bob", "luka", "cpp"};
        int expected = 1;
        int actual = Sort.findSmallest(input, 0);
        assertThat(actual).isEqualTo(expected);
        int expected = 3;
        int actual2 = Sort.findSmallest(input, 2);
        assertThat(actual2).isEqualTo(expected);
    }
    @Test
    public void testSwap() {
        String[] input = {"cat", "bob", "luka", "cpp"};
        String[] expected = {"bob", "cat", "luka", "cpp"}
        Sort.swap(input, 0, 1);
        assertThat(input).isEqualTo(expected);
    }
}
```

```
public class Sort {
    public static void sort(String[] x) {
        sort(x, 0);
    }
```

```
    private static void sort(String[] x, int s) {
        if (s == x.length) {
            return;
        }
        int smallest = findSmallest(x, s);
        swap(x, s, smallest);
        sort(x, s + 1);
    }


    public static int findSmallest(String[] x, int s) {
        int smallestIndex = s;
        for (int i = s; i < x.length; i++) {
            if (x[i].compareTo(x[smallestIndex]) < 0) {
                smallestIndex = i;
            }
        }
        return smallest;
    }

    public static void swap(String[] x, int a, int b) {
        String temp = x[a];
        x[a] = x[b];
        x[b] = temp;
    }
}
```

Tests provide stability and scaffolding. It provides confidence in basic units.

We should avoid autograder-driven development.

Lec 7: ArrayLists, Resizing

Retrieval from any position of an array is very fast.

```
public class AList {
    private int[] items;
    private int size;
    public AList() {
        size = 0;
        items = new int[100];
    }
    public void resize(int capacity) {
        int[] a = new int[capacity];
        System.arraycopy(items, 0, a, 0, size);
        items = a;
    }
    public void addLast(int x) {
        if (size == item.length) {
            resize(size * 2);
        }
        items[size] = x;
        size++;
    }
    public int getLast() {
```

```
        return items[size - 1];
    }
    public int get(int i) {
        if (i >= size) {
            throw new IllegalArgumentException("We don't have that much stuff");
        }
        return items[i];
    }
    public int removeLast() {
        int itemToReturn = getLast();
        size--;
        return itemToReturn;
    }
}
```

AList Invariants:

- The position of the next item to be inserted is always `size`.
- `size` is always the number of items in the AList.
- The last item in the list is always in position `size - 1`.

We speed tested the different `resize()` strategy. It turns out that `resize(size * RFACTOR);` has great performance, and incidentally, this is how Python list is implemented.

Also, if we define the "usage ratio" `R = size / items.length`, we should half array size when `R < 0.25`.

Generic AList

We can also make this AList generic.

```
public class AList<T> {
    private T[] items;
    private int size;
    public AList() {
        items = (T[]) new Object[8]; // weird type conversion, though
        size = 0;
    }
    ...
}
```

Unlike integer based ALists, we should null out deleted items, so that Java can destroy those unwanted items (which happens when the last reference has been lost).

Keeping references to unneeded objects is sometimes called loitering.

Obscurantism in Java

The user of a class does not and should not know how it works.

- User's mental model : `{1, 2, 3, 4, 5, 6} -> {1, 2, 3, 4, 5}`
- Actual truth: the number 6 still exists, but `size = size - 1`.

Lec 8: Interface and Implementation Inheritance

Interface Inheritance

Method Overloading: Multiple methods with the same name, but different parameters.

Hypernyms: e.g. "Dog" is a hypernym of poodle, malamute, yorkie, etc.

Hyponym: e.g. "Poodle" is the hyponym of dog.

To specify these hyponymic relationships in Java,

- Step 1: Define a reference type for our hypernym (`List61B`)
  - Use the keyword `interface`
  - Interface is a specification of what a List is able to do, not how to do it
- Step 2: Specify that `SLList` and `AList` are hyponyms of that type
  - Use the keyword `implements`

```
public interface List61B<Item> {
    // No variable
    // No constructor
    // Cannot be instantiated directly
    public void insert(Item x, int position);
    public void addFirst(Item x);
    public void addLast(Item x);
    public Item getFirst();
    public Item getLast();
    public Item get(int i);
    public int size();
    public Item removeLast();

}
```

```
public class AList<Item> implements List61B<Item> {
    ...
}

public class SLList<Blorp> implements List61B<Blorp> {
    ...
}
```

So we can use `List61B` as our parameter type:

```
public class WordUtils {
    public static String longest(List61B<String> list) {
        ...
    }
}
```

Overriding vs. Overloading:

- **Override**: Same signature (method name and parameters). In subclass and superclass.

```
public interface Animal {
    public void makeNoise();
}

public class Pig implements Animal {
    @Override
    public void makeNoise() {
        System.out.println("Oink");
    }
}
```

- **Overload**: Same name, but different signatures

```
public class Dog implements Animal {
    public void makeNoise(Dog x) {
        ...
    }
}
```

In 61b, we'll always mark every overriding method with the `@Override` annotation. The only effect of the tag is that the code won't compile if it is not actually an overriding method.

And what we've just had done is something called **interface inheritance**:

- Interface: The list of all method signatures
- Using the keyword `implements`
- The subclass "inherits" the interface from a superclass
- Subclasses must override all of these methods
- Such relationships can be multi-generational

Implementation Inheritance

Implementation Inheritance: Subclass inherits signatures and implementation.

Use `default` keyword to specify a method that subclasses should inherit from an interface.

```
public interface List61B<Item> {
    ...
    default public void print() {
        for (int i = 0; i < size(); i++) {
            System.out.print(get(i) + " ");
        }
    }
}
```

But for `SLList`, this `print()` method is not efficient.

```java
public class SLList<Blorp> implements List61B<Blorp> {
    ...
    @Override
    public void print() {
        Node p = sentinel.next;
        while (p != null) {
            System.out.print(p.item + " ");
            p = p.next;
        }
    }
}
```

Abstract Data Types

An abstract data type (ADT) is defined only by its operations, not by its implementation.

```java
List<Integer> L = new ArrayList<>();
```

Dynamic Method Selection

Used to be taught in 61b, but just annoying.

Lec 9: Subtype Polymorphism vs. Function Passing

Unlike Python, where function passing is common, most idiomatic Java code relies more heavily on polymorphism.

Polymorphism: the ability in programming to present the same programming interface for differing underlying forms.

e.g. Operator overloading in Python is a form of polymorphism.

Function passing: e.g. `sort(TO_BE_SORTED, COMPARING_FUNCTION)`

Comparable

Unlike Python, where we'd define a `__gt__` function to overload the `>` operator, in Java, we have to specify that a `Dog` is something that can be compared.

In Java, we will declare to the world that a `Dog` is-a `Comparable`.

In `Comparable.java`, we can see:

```java
public interface Comparable<T> {
    /**
     * Compares this object with the specifies object for order.
     * Returns a negative integer, zero, or a positive integer
     * as this object is less than, equal to, or greater than the
     * specified object.
     * ...
    */
    public int compareTo(T o);
}
```

```
public class Dog implements Comparable<Dog> {
    String name;
    int size;
    public Dog(String n, int s) {
        name = n;
        size = s;
    }
    @Override
    public int compareTo(Dog uddaDog) {
        return size - Dog.size;
    }
}


public class CollectionDogDemo {
    public static void main(String[] args) {
        List<Dog> dogs = new ArrayList<>();
        ...
        Dog maxDog = Collections.max(dogs);
    }
}
```

In this example, a supertype (Comparable) specifies the capabilities (in this case, comparison). A subtype overrides the supertype's abstract method.

Comparators

Natural order: the ordering implied by a `Comparable`'s `compareTo` method.

Java provides a `Comparator` interface for objects that are designed for comparing other objects.

```
public class Dog implements Comparable<Dog> {
    ...
    public static class NameComparator implements Comparator<Dog> {
        @Override
        public int compare(Dog a, Dog b) {
            return a.name.compareTo(b.name);
        }
    }
}
public class Demo {
    public static void main(String[] args) {
        Dog a = new Dog("Frank", 1);
        Dog b = new Dog("Zeke", 1);
        Comparator<Dog> nc = new Dog.NameComparator();
        Dog maxNameDog = Collections.max(dogs, new Dog.NameComparator());
        System.out.println(nc.compare(a, b));
    }
}
```

As you can see, the second argument in `Collections.mex()` is an object of type `Dog.NameComparator()`.

In Java, we package our comparison function inside of a `Comparator` object. We rely on subtype polymorphism.

Creating an object `Dog.NameComparator()` using `new` inside a function is really weird. In most cases people would do:

```
public class Dog implements Comparable<Dog> {
    ...
    private static class NameComparator implements Comparator<Dog> {
        ...
    }
    public static Comparator<Dog> NAME_COMPARATOR = new NameComparator();
}
public class Demo {
    public static void main(String[] args) {
        ...
        Dog maxNameDog = Collections.max(dogs, Dog.NAME_COMPARATOR);
    }
}
```

Or, we could use lambdas.

```
public class Demo {
    public static void main(String[] args) {
        Comparator<Dog> dc = (d1, d2) -> d1.name.compareTo(d2.name);
        Dog maxNameDog = Collections.max(dogs, dc);
    }
}
```

Comparable vs. Comparators

- The `Comparable` interface specifies that a "natural order" exists.
    - Instances of the class can compare themselves to other objects.
    - Only one such order is possible.
- The `Comparator` interface is used to compare extrinsically (by other classes).
    - May have classes like `NameComparator`, `SpeedComparator`, `SizeComparator`.

Writing our own Max function

We can make the static method itself generic

- We do this by sticking `<T>` after the word `static`
- Read as "I am declaring a public static function that works on objects of type `<T>`, and it returns a `T`, and it is called `pickRandom`"

```java
public class RandomPicker {
    public static <T> T pickRandom(T[] x) {
        Random random = new Random();
        int randomIndex = random.nextInt(x.length);
        return x[randomIndex];
    }
}
```

Generic types only works with reference types!

And we should let Java know T is-a `Compatable<T>` using a Type Bound

```java
public class Maximizer {
    public static <T extends Comparable<T>> T max(T[] items) {
        T maxItem = items[0];
        for (int i = 0; i < items.length; i++) {
            int cmp = items[i].compareTo(maxItem);
            if (cmp > 0) {
                maxItem = items[i];
            }
        }
        return maxItem;
    }
}
```

But Java in fact has a messy signature, which goes like this

```java
public static <T extends Comparable<? super T>> T max(T[] items) {
    ...
}
```

Lec 10: Iterators, Object Methods

Today's goal: Build an implementation of a `Set` called `ArraySet`.

```java
public class ArraySet<T> {
    private T[] items;
    private int size;

    public ArraySet() {
        items = (T[]) new Object[100];
        size = 0;
    }

    public boolean contains(T x) {
        for (int i = 0; i < size; i++) {
            if (items[i].equals(x)) {
                return true;
            }
        }
        return false;
    }
```

```
    public void add(T x) {
        if (x == null) {
            throw new IllegalArgumentExpection("can't add null");
        }
        if (contains(x)) {
            return;
        }
        item[size] = x;
        size += 1;
    }
}
```

Iteration

The "enhanced for loop" or "for each loop"

```
for (int i : javaset) {
    System.out.println(i);
}
```

But it doesn't work with our own `ArraySet`.

So we have to tell Java that our `ArraySet` is-an iterable.

The enhanced for loop works by first calling the `.iterator` method of the object

- This returns an object of type `Iterator<Integer>`
- The `Iterator` interface has its own API for fetching values one-by-one:
    - `hasNext`: tells us whether there are more values
    - `next`: gets the next value and advances the pointer forwards

```
Iterator<Integer> seer = javaset.iterator();
while (seer.hasNext()) {
    int x = seer.next();
    System.out.println(x);
}
```

```
public class ArraySet<T> implements Iterable<T> {
    private class ArraySetIterator implements Iterator<T> {
        // the position of the wizard (or the seer)
        private int wizPos;
        ArraySetIterator() {
            wizPos = 0;
        }
        @Override
        public boolean hasNext() {
            return wisPos < size;
        }
        @Override
        public T next() {
            T itemToReturn = items[wizPos];
            wizPos += 1;
            return itemToReturn;
```

```
        }
    }
    public Iterator<T> iterator() {
        return new ArraySetIterator();
    }
}
```

Object Methods

All classes are hyponyms of Object.

The `toString()` method

To achieve:

```
System.out.println(javaset);
```

we should have a `toString()` method.

```
public class ArraySet<T> implements Iterable<T> {
    @Override
    public String toString() {
        StringBuilder stringToReturn = new StringBuilder("{");
        for (T x : this) {
            stringToReturn.append(x);
            stringToReturn.append(", ");
        }
        stringToReturn.append("}");
        return stringToReturn.toString();
    }
}
```

`equals` vs. `==`

`==` compares the bits, which means "referencing the same object"

The default implementation of `equals`

```
public class Object {
    public boolean equals(Object obj) {
        return (this == obj);
    }
}
```

Which is not really good, we can implement it as follows:

```
public class ArraySet<T> implements Iterable<T> {
    @Override
    public boolean equals(Object o) {
        // Check if o is an ArraySet
        if (this == o) {
            return true;
        }
        if (o instanceof ArraySet uddaSet) {
```

```
            if (this.size != o.size) {
                return false;
            }
            for (T x : this) {
                if (!uddaSet.contains(x)) {
                    return false;
                }
            }
            return true;
        }
        return false;
    }
}
```

The `instanceof` keyword is really powerful in Java.

- Checks to see if `o` is pointing at a `Dog`
- If no, returns false.
- If yes, returns true and puts `o` in a new variable of type `Dog` called `uddaDog`
- Works correctly, even is o is `null`

```
@Override
public boolean equals(Object o) {
    if (o instanceof Dog uddaDog) {
        return this.size == uddaDog.size;
    }
    return false;
}
```

Lec 11: Asymptotic Analysis I

In most cases, we only care about asymptotic behavior, i.e. what happens for very large N.

The formal definition of **Big-Theta**:

$$R(N) \in \Theta(f(N))$$

means there exist positive constants k_1 and k_2 such that:

$$k_1 * f(N) \leq R(N) \leq k_2 * f(N)$$

for all values of N greater than some N_0

For example: $4N^2 + N \in \Theta(N^2)$

where $R(N) = 4N^2 + N,\ f(N) = N^2,\ k_1 = 3$ and $k_2 = 5$.

Lec 12: Asymptotic Analysis II (CS70 preview)

Big O and Big Omega

Whereas Big Theta can informally be thought of as something like "equals", Big O can be thought as "less than or equal" and Big Omega can be thought of as "greater than or equal"

e.g. $N^3 + 3N^4 \in \Theta(N^4)$;

$N^3 + 3N^4 \in O(N^4)$

$N^3 + 3N^4 \in O(N^6)$

$N^3 + 3N^4 \in \Omega(N^2)$

...

Common Theta Classes

- Polynomials: 1, N, N^2, N^3
- Exponentials / Hyper-exponentials
- Logarithms
- Combinations of above

Lec 14: Asymptotic Analysis III

If my function f(n) is ...	Doubling N (N -> 2N)	Adding 1 to N (N -> N + 1)
$\Theta(1)$ (constant runtime)	Doesn't affect runtime	Doesn't affect runtime
$\Theta(\log n)$ (base independent)	Adds 1 to runtime	Affects runtime minimally
$\Theta(n)$	Doubles runtime	Adds 1 to runtime
$\Theta(n^2)$	Quadruples runtime	Adds n to runtime
$\Theta(2^n)$ (base independent)	Squares runtime	Doubles runtime

Amortized Runtime

Amortized runtime gives a better estimate of how much time it takes to use something in practice. e.g. We break $\Theta(N)$ and things alike to $\Theta(1)$.

Recursive Runtime

```
public static int f(int n) {
    if (n <= 1) {
        return 1;
    }
    return f(n - 1) + f(n - 1)
}
```

The runtime $R(N)$ should be $\Theta(2^N)$.

e.g. Merge sort have the order of growth of $\Theta(N \log N)$.

Lec 15: Disjoint Sets

The Disjointed Sets data structure has two operations:

- `connect(x, y)`: Connects x and y.
- `isConnected(x, y)`: Returns true if x and y are connected

To keep things simple, we design:

```
the original  -> {0}, {1}, {2}, {3}, {4}, {5}
connect(0, 1) -> {0, 1}, {2}, {3}, {4}, {5}
connect(1, 2) -> {0, 1, 2}, {3}, {4}, {5}
connect(3, 5) -> {0, 1, 2}, {3, 5}, {4}
connect(0, 3) -> {0, 1, 2, 3, 5}, {4}
```

Check `isConnected(x, y)`: to check whether x and y are in the same set

Then we need a data structure to represent these sets.

If we use `List<Set<Integer>>`, which is a very intuitive idea, it will be really slow. It requires iterating through all the sets to find anything

Another idea is we use a list of integers where i`th entry gives set number (a.k.a. "id") of item`i`.

`connect(p, q)`: Change entries that equal `id[p]` to `id[q]`.

```java
public class QuickFindDS implements DisjointSets {
    private int[] id;

    // Very fast: O(1)
    public boolean isConnected(int p, int q) {
        return id[p] == id[q];
    }

    // Relatively slow: O(N)
    public void connect(int p, int q) {
        int pid = id[p];
        int qid = id[q];
        for (int i = 0; i < id.length; i++) {
            if (id[i] == pid) {
                id[i] = qid;
            }
        }
    }

    public QuickFindDS(int N) {
        id = new int[N];
        for (int i = 0; i < N; i++) {
            id[i] = -1;
        }
    }
}
```

The more radical solution is called `QuickUnion`.

We draw a tree-shaped data structure:

`isConnected(x, y)`: climb up the tree to see if x and y have the same root.

If x is the root, `parent[x]` should be -1.

The worst case still is `O(N)` (where you have to go up once and once to find the root), which is not ideal.

We've got a better way: to track the weight (size)

- Track tree size (number of elements)
- New rule: Always link root of smaller tree to larger tree

`QuickFind -> QuickUnion -> WeightedQuickUnion -> WeightedQuickUnion + Path Compression`

```
public class WeightedQuickUnionDS implements DisjointSets {
    private int[] parent;  // parent[i] indicates the parent of element i
    private int[] size;    // size[i] tracks the number of nodes in the tree
rooted at i

    // Constructor: Initialize each element's parent as itself and tree size as 1
    public WeightedQuickUnionDS(int N) {
        parent = new int[N];
        size = new int[N];
        for (int i = 0; i < N; i++) {
            parent[i] = i;   // Each element starts as its own parent
            size[i] = 1;     // Each tree starts with a single node
        }
    }

    // Find the root of element p with path compression
    private int find(int p) {
        while (p != parent[p]) {
            parent[p] = parent[parent[p]];  // Path compression: link p directly
to its grandparent
            p = parent[p];
        }
        return p;
    }

    // Check if p and q are connected (in the same set)
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }

    // Connect the sets containing p and q (merge smaller tree into larger one)
    public void connect(int p, int q) {
        int rootP = find(p);
        int rootQ = find(q);
        if (rootP == rootQ) return;  // Already connected, no action needed

        // Attach smaller tree to larger tree to maintain balance
        if (size[rootP] < size[rootQ]) {
```

```
            parent[rootP] = rootQ;
            size[rootQ] += size[rootP];
        } else {
            parent[rootQ] = rootP;
            size[rootP] += size[rootQ];
        }
    }
}
```

Lec 16: Extends, Sets, Maps and BSTs

Extends

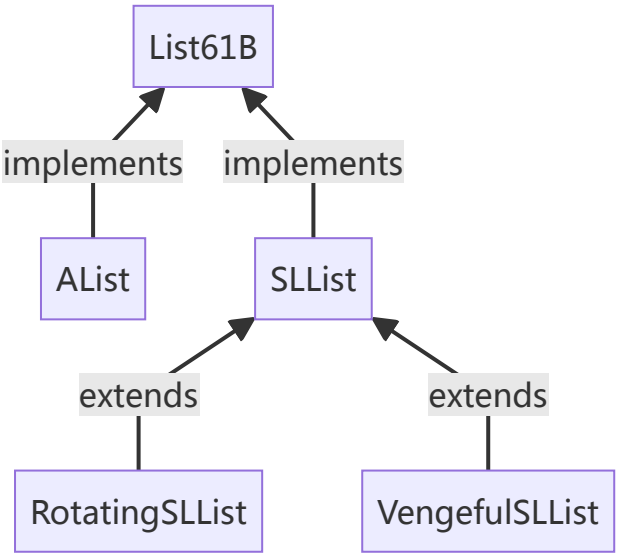
When a class is a hyponym of an interface, we use `implements`.

If you want one class to be a hyponym of another class (instead of another interface), use `extends`.

We'd like to build `RotatingSLList` that can perform any `SLList` operation as well as:

`rotateRight()`: Moves back item the front.

- e.g . [5, 9, 15, 22] -> [22, 5, 9, 15]




```
public class RotatingSLList<Item> extends SLList<Item> {
   /* Rotates list to the right. */
   public void rotateRight() {
      Item oldLast = removeLast();
      addFirst(oldLast);
   }
}
```

Because of `extends`, `RotatingSLList` inherits all members (except private) of `SLList`:

- All instance and static variables.
- All methods.
- All nested classes.

Constructors are not inherited.

```
public class VengefulSLList<Item> extends SLList<Item> {
    private List<Item> deletedItems;
   public VengefulSLList() {
       super(); // Calling the super's constructor. Done implicitly so you don't
have to say
       deletedItems = new ArrayList<Item>(); // make sure it is not null
   }

   @Override
   public Item removeLast() {
       Item x = super.removeLast(); // super refers to the 'SLList'
       deletedItems.addLast(x);
       return x;
   }

   public void printLostItems() {
       deletedItems.print();
   }
}
```

So we use `extends` to say that it is a subclass of another class.

- `implements` is used between classes / interfaces
- `extends` is used between classes / classes and interfaces / interfaces

Superclass constructor happens implicitly even if you don't do this.

Binary Search Trees (BST)

An ADT (abstract data type) is defined only by its operations, not by its implementation

With an array, it is $\Theta(N)$ to check `contains(x)`, which is slow.

Binary search tree helps reduce the worst runtime to $\Theta(\log N)$ if the tree is balanced.

A tree consists of a set of nodes, and a set of edges. Only one path between any two nodes.

In a rooted binary tree, every node has either 0, 1, or 2 children.

A BST is a rooted binary trees, where for every node X in the tree (BST Properties)

- Every key in the left subtree is less than X's key
- Every key in the right subtree is greater than X's key

Ordering must be complete, transitive and antisymmetric.

Bushy BSTs are extremely fast to search for some items.

To insert a new key into a BST:

- Search for the key
    - if found, do nothing
    - if not found:
        - Create a new node

        - Set appropriate link

To delete from a BST

- If deletion key has no children
    - Sever the parent's link
    - The deletion key will be garbage collected (along with its instance variables)
- If deletion key has one child
    - Move the parent's pointer to its child
    - The deletion key will be garbage collected (along with its instance variables)
- If deletion key has two children
    - Hibbard Deletion
        - Delete either "predecessor" or "successor", and stick new copy in the root position
        - The **predecessor** of a node `x` is the **largest node** in its **left subtree**, i.e., the node with the largest value smaller than `x`.
        - The **successor** of a node `x` is the **smallest node** in its **right subtree**, i.e., the node with the smallest value larger than `x`.

Sets vs. Maps

To represent maps, just have each BST node store key/value pairs.

Lec 17: B-Trees (and 2-3 and 2-3-4 Trees)

BSTs Height and Average Depth

Height varies dramatically between "bushy" $\Theta(\log N)$ and "spindly" $\Theta(N)$ (linked-list like) trees.

Big O is often used as shorthand for "worst case", although mathematically it is not the same thing.

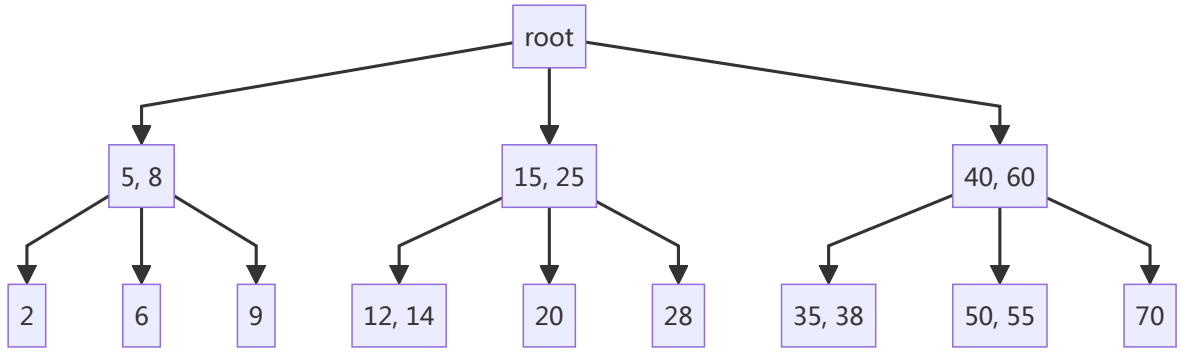
The depth of a node is how far it is from the root.

Height and average depth determine runtimes for BST operations.

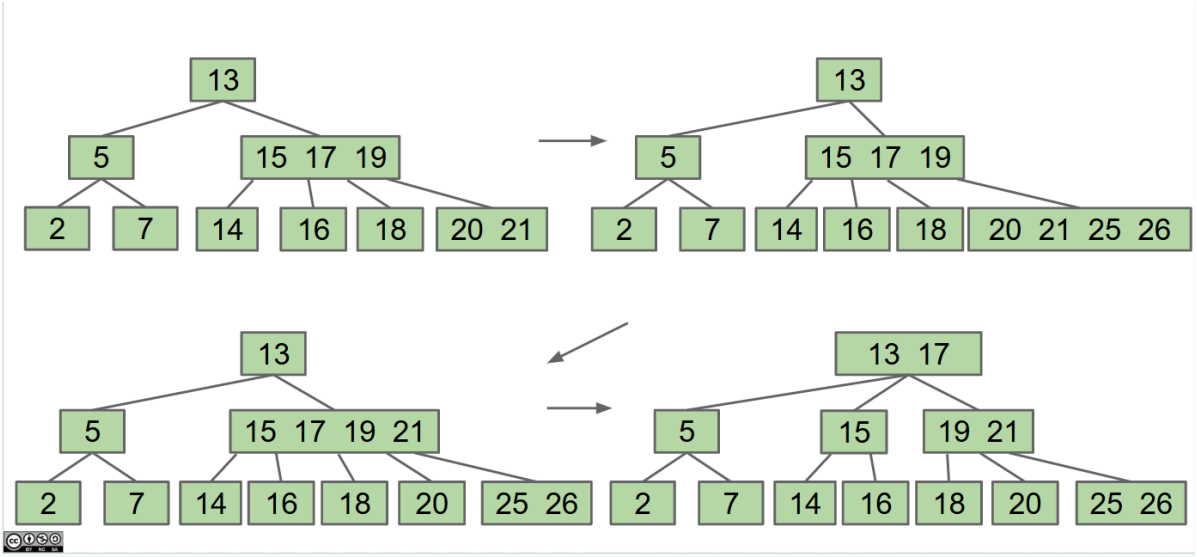
Random trees have $\Theta(\log N)$ average depth and height.

Note: ~ is the same thing as Big Theta, but you don't drop the multiplicative constants

B-Trees



Adding 25, 26 to a existing B-tree (L = 3, i.e. less than or equal to 3 items per node)



B-Tree Terminology:

- Splitting-trees have perfect balance
  - If we split the root, every node gets pushed down by exactly one level
  - If we split a leaf node or internal node, the height doesn't change
  - All operations have guaranteed $O(\log N)$ time.
- B-trees of order L=3 are also called a 2-3-4 tree or a 2-4 tree.
  - Any node could have 2 or 3 or 4 children
- B-trees of order L=2 are also called a 2-3 tree.

Interactive demo: B-Tree Visualization

Invariants:

- All leaves must be the same distance from the root (perfectly balanced)
- A non-leaf node with k items must have exactly k+1 children

Lec 18: Red Black Trees

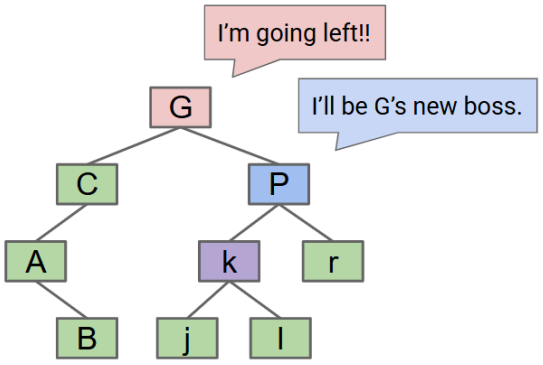
B-Trees are really hard to implement.

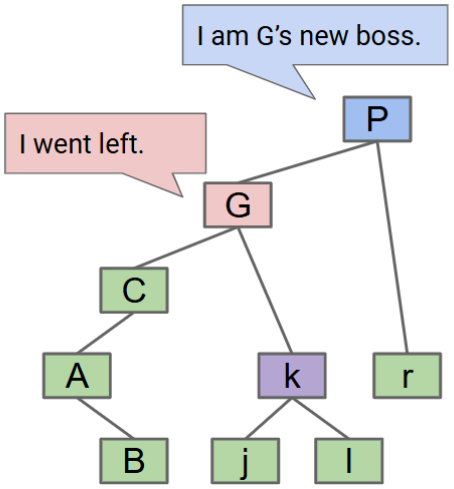
Tree Rotation

You can have your BST move to another configuration in $2n - 6$ rotations.

`rotateLeft(G)` : (Let x be the right child of G.) Make G the new left child of x.

- Preserves search tree property. No change to semantics of tree.



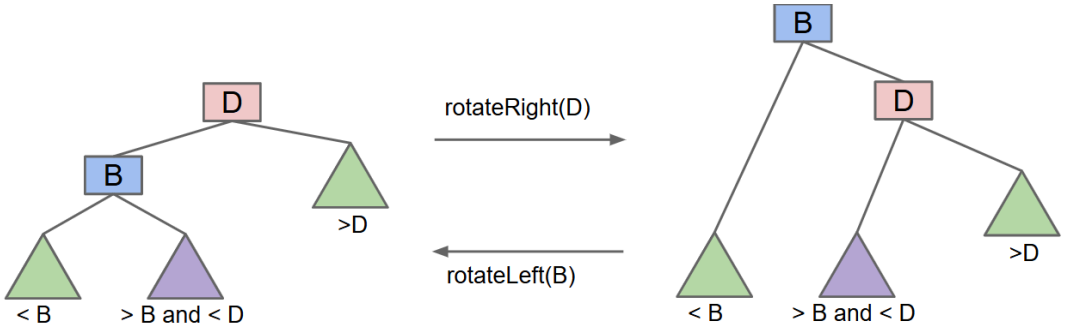


- The node k also moves to G, in order to maintain its "binary" property
- Can think of as temporarily merging G and P, then sending G down and left.

`rotateRight(P)` : (Let x be the left child of P.) Make P the new right child of x.

- Can think of as temporarily merging G and P, then sending P down and right.

Rotation can shorten or lengthen a tree.



Using rotation, we can balance a BST.

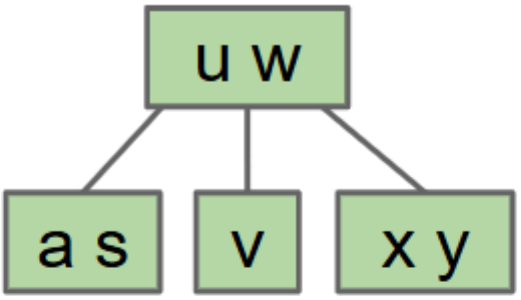
- Rotating allows balancing of a tree in $O(N)$ moves.

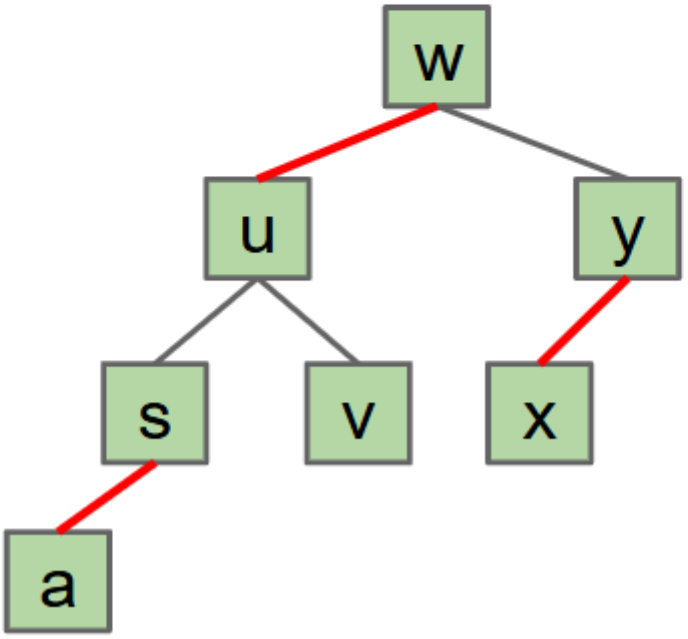
Red Black Trees

A BST with left glue links (usually in red) that represent a 2-3 tree is often called a "Left Leaning Red Black Binary Tree" or LLRB.

- LLRBs are normal BSTs!
- There is a 1-1 correspondence between an LLRB and an equivalent 2-3 tree.

- The red is just a convenient fiction. Red links don't do anything special.





An LLRB has no more than ~2x the height of its 2-3 tree.

There exists a 1-1 mapping between 2-3 Tree and LLRB.

To construct a red-black BST

- When inserting: use a red link
- If there is a right leaning "3-node", we have a Left Leaning Violation
  - Rotate left the appropriate node to fix
- If there are two consecutive left links, we have an Incorrect 4 Node Violation
  - Rotate right the appropriate node to fix
- If there are any nodes with two red children, we have a Temporary 4 Node
  - Color flip the node to emulate the split operation

The runtime analysis for LLRBs is $O(\log N)$.

Lec 19: Hash Tables

1. Motivation

Existing Set/Map implementations (e.g., BST, 2-3 Tree) have limitations:

- Require items to be comparable (not feasible for all types, e.g., 苹 vs 橙).
- Best performance is Θ(log N); we aim for better.

2. Deriving Hash Tables

- `WriteItOnTheWallSet`: Analogy to `ArraySet`.
  - Add: Write items randomly on a wall (Θ(1)).
  - Contains: Search entire wall (Θ(N)).
  - Pros: Works for any type; Cons: Slow contains.
- `BobaCounterSet`: Categorizes items into bins (e.g., last digit of numbers).
  - Add: Place in bin by last digit (Θ(1)).
  - Contains: Search only the relevant bin (faster if items are random).
  - Issues: Wasted space; poor performance for non-random data.
- `DynamicArrayOfListsSet`: Uses linked lists in bins + dynamic resizing.
  - Bins are linked lists (avoids fixed space waste).
  - Uses modulus (%) to map items to bin indices (generalizes "last digit" logic).
  - Resizes bins (doubles M) when average bin size exceeds a threshold, keeping N/M constant (Θ(1) amortized ops).

3. Handling Non-Integer Data (e.g., Strings)

- **Goal**: Convert data to integers for bin mapping.
- For strings: Treat as numbers in a base system (e.g., base 26 for lowercase letters, base 126 for ASCII).
  - Example: "cat" → 3×262+1×26+20=2074.
- **Problem**: Integer overflow (Java `int` has finite range: -2^{31} to $2^{31}-1$).

4. Hash Codes

- **Definition**: A function projecting data to a fixed range (Java `int`).
- Java uses base 31 for string hash codes (avoids overflow issues with larger bases).
- Collisions (different data → same hash code) are inevitable but manageable.
  - Only one possible case makes sense.
- Use small primes (e.g., 31) as bases (avoids overflow, ensures even distribution).
- Avoid simple functions (e.g., returning 0, summing characters) → causes collisions.

5. Hash Tables in Java

- **Implementations**: `HashMap` (Map) and `HashSet` (Set).
- Mechanism
  1. Data → hash code via `hashCode()`.
  2. Hash code → bin index via `Math.floorMod(hashCode, M)` (handles negatives).
- Warnings
  - Don't store mutable objects (hash code changes → items get lost).
  - Override `hashCode()` if overriding `equals()` (else, unexpected behavior).
- Performance
  - With resizing (M grows with N) and even distribution: Θ(1) average time for `add`/`contains`.
    - e.g. first using "a - z", later using "aa - zz" for faster iteration within each bin
  - Without resizing: Θ(N) (bins grow linearly with N).

Lec 20: Hash Tables II

Always override `hashCode()` when overriding `equals()`

Basic rule: If two objects are equal, they'd better have the same hash code so that the hash table can find it.

The default hash code is just based on the memory address.

The `HashSet` and `HashMap` might not work otherwise.

They use `hashCode()` to distribute items into different bins, but use `equals()` to check if the bin already contains the item. If there is no overridden `hashCode()` , chances are that the same item might appear in another bin.

Mutable and Immutable Types

Immutable data type: cannot change in any observable way after instantiation, e.g

- Mutable: `ArrayDeque`
- Immutable: `Integer`, `String`

The `final` keyword will help the compiler ensure immutability.

- Assign a value once, and after it can never change

Bottom line: Never mutate an Object being used as a key

Lec 21: Heaps and Priority Queues

Priority Queues

```
public interface MinPQ<Item> {
    public void add(Item x);
    public Item getSmallest();
    public Item removeSmallest();
    public int size();
}
```

Application: track the M best

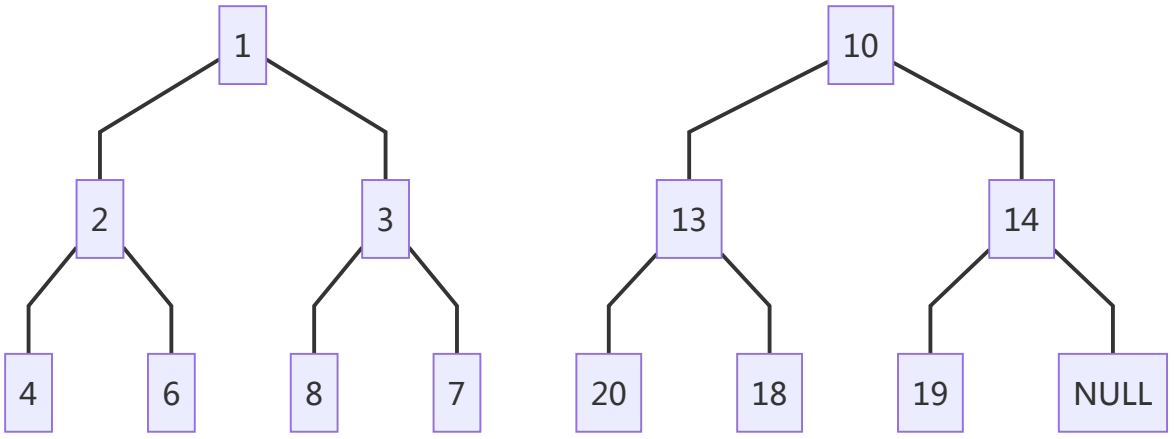
Implementation: using heap

Heap

Binary min-heap: Binary tree that is complete and obeys min-heap property

- Min-heap: Every node is less than or equal to both of its **children**
- Complete: Every level is full, except the bottom level may be partially empty. All nodes in the bottom level are as far left as possible

e.g.



To insert an item:

- Add to the end of the heap temporarily (for completeness)
- Swim up the hierarchy to the rightful place

To get the smallest item:

- Return the item in the root node

To remove the smallest item:

- Swap the last item in the heap into the root
- Then sink your way down the hierarchy, yielding to most qualified folks

How do we represent a tree in Java?

- Approach 1: Create mapping from node to children

```
public class Tree1A<Key> {
    Key k;
    Tree1A left;
    Tree1A middle;
    Tree1A right;
}

public class Tree1B<Key> {
    Key k;
    Tree1B[] children;
    ...
}

public class Tree1C<Key> {
    Key k;
    Tree1C favoredChild;
    Tree1C sibling;
    ...
}
```

- Approach 2: Store keys in an array. Store parent's IDs in an array

```
public class Tree2<Key> {
    Key[] keys;
    int[] parents;
}
```

- Approach 3 (only works when complete): Store keys in an array. Don't store structure anywhere.

```
public class Tree3<Key> {
    Key[] keys;
    ...
}
```

  - This works because for certain kinds of trees, the structure is determined as long as the items are given.
- Approach 3b: Princeton book implementation. Store keys in an array. Offset everything by 1 spot.
  - Same as 3, but leave spot 0 empty
  - Make computation of children/parents "nicer".
    - `leftChild(k) = k * 2`
    - `rightChild(k) = k * 2 + 1`
    - `parent(k) = k / 2`

We use approach 3 here to implement heaps.

Data Structure

It is a particular way of organizing data. We've covered many of the most fundamental ADTs, their common implementations and the tradeoffs thereof.

We will also do tries and graphs later.

Lec 22: Graph and Traversals

Tree Traversals

Iterating over a tree is called a **tree traversal**.

There are several patterns to traverse a tree

- Level order: top to bottom, left to right
- Depth first traversals:
    - Pre-order: ***key*** --> left --> right
    - In-order: left --> ***key*** --> right
    - Post-order left --> right --> ***key***

Graphs

A graph consists of:

- A set of nodes
- A set of zero or more edges, each of which connects two nodes

A ***simple*** graph is a graph with:

- No edges that connect a vertex to itself (no loops)
- No parallel edges

Vertices with an edge between are adjacent.

Vertices or edges may have labels (or weights).

A path is a sequence of vertices connected by edges

- A simple path is a path without repeated vertices
- A cycle is a path whose first and last vertices are the same
    - A graph with a cycle is "cyclic".

Two vertices are connected if there is a path between them.

If all vertices are connected, we say the graph is connected.

s-t path: is there a path between vertices s and t?

Graph problems are among the most mathematically rich areas of CS theory.

Graph Traversals

To check whether two vertices s and t are connected, consider depth first search:

- mark s
- does s == t?
    - return true if so
    - otherwise, if connected(v, t) for any unmarked neighbor v of s, return true
- return false

Similar to tree traversals, there are several patterns of graph traversal:

- DFS Preorder (DFS calls order)
- DFS Postorder (DFS return order)
- BFS order: Act in order of distance from s
    - Breadth first search

Lec 23: BFS, DFS and Implementations

Graph Representations

An API(Application Programming Interface) for graphs, from Princeton Textbook:

```
public class Graph {
    public Graph(int V); // Create an empty graph with v vertices
    public void addEdge(int v, int w); // Add an edge v-w
    Iterable<Integer> adj(int v); // Vertices adjacent to v
    int V();    // number of vertices
    int E();    // number of edges
}
```

- Number of vertices must be specified in advance
- Does not have weights
- Has no method for getting the degree for a vertex

```
public static void print(Graph G) {
    for (int i = 0; i < G.V(); i++) {
        for (int w : G.adj(w))
        System.out.println(i + "-" + w);
    }
}

public class Paths {
    public Paths(Graph G, int s); // Find all paths from G
    boolean hasPathTo(int v); // Is there a path from s to v?
    Iterable<Integer> pathTo(int v); // Path from s to v (if any)
}
```

Graph Representation 1: Adjacency Matrix

Representation 2: Edge sets (collection of all edges)

Representation 3: Adjacency lists. Maintain array of lists indexed by vertex number

- In this implementation, print takes $\Theta(V + E)$
- For DFS, it takes $O(V + E)$

Find the shortest path from s to v:

- Initialize the fringe (a queue with a starting vertex s) and mark that vertex
- Repeat until fringe is empty
    - Remove vertex v from the front of fringe
    - For each unmarked neighbor n of v:
        - Mark n
        - Set `edgeTo[n] = v`, set `distTo[n] = distTo[v] + 1`
        - Add n to the back of fringe

Lec 28: Shortest Paths

All the shortest paths from a certain node to any other nodes in the graph form a tree.

This is called a Shortest Paths Tree (SPT).

Dijkstra's Algorithm

Dijkstra's:

- `PQ.add(source, 0)`
- For other vertices `v`, `PQ.add(v, infinity)`
- While `PQ` is not empty:
    - `p = PQ.removeSmallest()`
    - Relax all edges from `p`

Relaxing an edge `p -> q` with weight `w`:

- If `distTo[p] + w < distTo[q]`:
    - `distTo[q] = distTo[p] + w`
    - `edgeTo[q] = p`
    - `PQ.changePriority(q, distTo[q])`

Key invariants:

- `edgeTo[v]` is the best known predecessor of `v`
- `distTo[v]` is the best known total distance from source to `v`
- `PQ` contains all unvisited vertices in order of `distTo`

Important proerties:

- Always visits vertices in order of total distance from source
- Relaxation always fails on edges to visited (white) vertices

In short: Visit vertices in order of best known distance from source. On visit, relax every edge from the visited vertex.

It will return the correct answer as long as the weights are all non-negative.

Overall runtime: `O(E * log(V))`

A*

A* is an extension to Dijkstra.

In Dijkstra, if you want to get the shortest path between `s` and `t,` you have to get almost all the shortest paths from `s` in the graph.

But in A*, we introduce a heuristic function, which **estimates the cost from a given node to the target node** `t`.

A* evaluates nodes using a combination of two values:

1. $g(n)$: The actual cost from the start node `s` to the current node `n` (this is the same as the distance tracked in Dijkstra's algorithm).
2. $h(n)$: The heuristic estimate of the cost from node `n` to the target `t`.

The total cost for node `n` is $f(n) = g(n) + h(n)$. A* always expands the node with the lowest $f(n)$ value, as it represents the best guess for the shortest path through that node.

Lec 25: Minimum Spanning Tree

Given a undirected graph `G`, a **spanning tree** `T` is a subgraph of `G`, where `T`:

- Is a tree (connected and acyclic)
- Includes all of the vertices

A **minimum spanning tree** (MST) is a spanning tree of minimum total weight.

The Cut Property

A cut is an assignment of a graph's nodes to two non-empty sets.

A crossing edge is an edge which connects a node from one set to the other set.

Cut property: Given any cut, minimum weight crossing edge (the crossing edge with the minimum weight) is in the MST.

Prim's Algorithm

Start from some arbitrary start node

- Add shortest edge (mark black) that has one node inside the MST under construction.
- Repeat until `V - 1` nodes.

Kruskal's Algorithm

Consider edges in order of increasing weight.

Add to MST unless a cycle is created.

Repeat until `V - 1` edges.

Checking if adding the edge with create a cycle uses a disjoint set.

Lec 26: Directed Acyclic Graphs (DAGs)

Topological Sorting

A **topological sort** (or topological ordering) of a DAG is a linear ordering of its vertices such that for every directed edge (`u -> v`) in the graph, vertex `u` comes before vertex `v` in the ordering.

In other words, it respects the direction of all edges, ensuring that dependencies (if any) are satisfied.

Common Algorithms for Topological Sort:

1. **Kahn's Algorithm** (using in-degree counts):
    - Compute the in-degree (number of incoming edges) for all vertices.
    - Initialize a queue with vertices of in-degree 0.
    - While the queue is not empty:
        - Remove a vertex u from the queue and add it to the result.
        - For each neighbor v of u, decrement their in-degree by 1. If any in-degree becomes 0, add v to the queue.
    - If the result contains all vertices, it is a valid topological sort; otherwise, the graph has a cycle.
2. **Depth-First Search (DFS) with Post-order Traversal**:
    - Perform a DFS traversal, and after visiting all descendants of a vertex, add it to a stack.
    - Reverse the stack to get the topological order. This works because a vertex is added to the stack only after all its dependents are processed.

A topological sort exists only if the graph is a DAG.

DAG Short Paths

Dijkstra doesn't work when there is negative numbers.

First, find a topological order.

Second, visit vertices in topological order

- When we visit a vertex: relax all of its going edges

The Longest Paths

DAG LPT solution for a graph G:

- Form a new copy of graph `G'` with sighs of all edges weights flipped
- Run DAG SPT on `G'` yielding result `X`.
- Flip signs of all values in `X.distTo[]` and `X.edgeTo[]` is already correct

Reduction

A **reduction** is a method to transform one problem into another such that:

- If you can solve the second problem efficiently, you can use that solution to solve the first problem efficiently.
- Formally: Problem A reduces to Problem B.

Lec 27: Tries

Conceptual Overview

- **Trie**: A specialized data structure for string keys.
- Key Ideas
  - Each node stores a single character.
  - Nodes are shared among multiple keys.
  - Nodes have a flag (`isKey`) to indicate if they mark the end of a valid key.

Implementation & Performance

- **Basic Structure**: Nodes contain a map of child nodes (char → node) and an `isKey` flag.
- Performance
  - `add(x)` and `contains(x)` run in **O(L)** time, where L is the length of the string (independent of the number of keys N).
  - Compared to other structures:

Structure	**Key Type**	`get(x)`	`add(x)`
Balanced BST	Comparable	Θ(log N)	Θ(log N)
Hash Table	Hashable	Θ(1) (avg)	Θ(1) (avg)
Data-indexed Array	Chars	Θ(1)	Θ(1)
Trie	Strings	Θ(L)	Θ(L)

Alternate Child Tracking Strategies

- 3 Approaches to map characters to child nodes:
    1. `DataIndexedCharMap` (Array): Fast (Θ(1) access) but memory-heavy (stores R links per node, e.g., 128 for ASCII).
    2. **Hash Table**: Nearly as fast as array, uses less memory (stores only used links).
    3. **Balanced BST**: Slightly slower than hash table (O(log R) access), similar memory efficiency to hash table.

Summary

- **Advantages**: Superior performance for string keys; efficient prefix operations.
- **Trade-offs**: Memory usage depends on child tracking (array = high memory, hash table/BST = better efficiency).
- **Applications**: Autocomplete, spell check, prefix matching.

Lec 28: Basic Sorts

A sort is a permutation of a sequence of elements that puts the keys into non-decreasing order relative to a given ordering relation.

An inversion is a pair of elements that are out of order with respect to `<`.

So sorting is:

- Given a sequence of elements with Z inversions.
- Perform a sequence of operations that reduces inversions to 0.

Selection Sort

The selection sort we mentioned at Lecture 6 goes as follows.

- Find the smallest item in the unsorted portion of the array
- Move it to the end of the sorted portion of the array
- Selection sort the remaining unsorted items

$\Theta(N^2)$ time complexity and $\Theta(1)$ space complexity.

Heap Sort

The heap sort goes at follow:

- Convert input into a heap
    - Use max heap and bottom-up heapification
        - Sink nodes in reverse level order: `sink(k)`
        - After sinking, guaranteed that tree rooted at position `k` is a heap
- Repeat N times:

- Delete largest item from the max heap, and move deleted items to vacated array slot (Uses no extra memory!)

$\Theta(N \log N)$ time complexity and $\Theta(1)$ space complexity.

Lec 29: Software Engineering I

The lectures are heavily inspired by "A Philosophy of Software Design" by John Ousterhout.

Over time, program becomes more difficult for programmers to understand all the relevant pieces as they make future modifications.

Dealing with Complexity

- Making code simpler and more obvious
- Encapsulation into modules

Three symptoms of complexity:

- Change amplification: A simple change requires modification in many places
- Cognitive load: How much you need to know in order to make a change.
- Unknown unknowns: You don't know what you need to modify to make a change.

Refactoring means to redo your code.

The most important thing is the long term structure of the system.

Lec 30: Merge sort, Insertion sort, and other sorts

Merge Sort

Merge sort does merges all the way down (no selection sort):

- If the array is of size 1, return
- Merge sort the left half
- Merge sort the right half
- Merge the results

It has runtime of $\Theta(N \log N)$ and space complexity of $\Theta(N)$.

Insertion Sort

General Strategy:

- Repeat for `i = 0` to `i = N - 1`:
  - Designate item `i` as the traveling item
  - Swap item backwards until traveler is in the right place among all previously examined items

Runtime analysis: $\Omega(N)$ and $O(N^2)$.

One swap reduce exactly one inversion.

For small arrays (N < 15 or so), insertion sort is the fastest.

This is known as an **adaptive sorting algorithm** since it takes advantage of the existing order.

Some Bad Sorts: Bubble Sort

- Iterate through the list and look at each adjacent pair
- If the two elements are in a wrong order, swap them
- Repeat iterating through list until the array is sorted

Shuffling Algorithm

Shuffle: Given a list of n elements and a random number generator, return a random permutation (reordering) of the list.

For each number `i` from `0` to `n - 1`:

- Pick a random item by selecting an index from `i` to `n - 1`
- Swap that item with the item in index `i`

Runtime $\Theta(N)$

Some Bad Sorts: BOGO Sort

While list is not sorted:

- Shuffle the list

Average runtime: $\Theta(N * N!)$

Lec 31: Software Engineering II

What do a software engineer do day to day?

- In an open office: collaborations and conversations
- What about remote work and work not in real time?

Today's topic: agile development and git

Agile Development

An iterative approach to delivering a project, which focuses on continuous releases that incorporate customer feedback.

4 principles:

- Individuals and interactions
- Working software
- Customer collaboration
- Responding to change

Scrum is an Agile framework for managing work.

To put these plans into action, **Jira** is a software application for issue tracking and project management.

- Backlog: A list of features that need to be developed
- Sprint: Tasks are divided into several sprints
- Increment: The usable product version created at the end of each sprint

Git

Committing is like a stronger type of saving.

- Saves a snapshot of your files
- You can go back to this snapshot if you would like

`git add` tells Git which file to save (add to the staging area)

`git commit` executes the save

- Adds commit to version history in your local repository

`gti push` sends you updated version history to a remote repo

Commits contain pointers to the commits that came before them.

- Commit history forms a linked list
- `git log` gives you the commit history
- `git checkout` inspects the past version
- `git revert` or `git reset` undoes the changes

Remote repositories can be shared by multiple people via GitHub, GitLab and Bitbucket...

- `git push` updates the remote with your local changes
- `git pull` incorporates changes from the remote into local
- Remember to pull before pushing

`git push` and `git pull` tries to copy the linked list of the commits back and forth between local and remote repos.

If the remote and local commit histories at one point shared a common ancestor commit but have since then gone their own way, we say that the two histories **diverge** from each other.

- Solution: merge the remote and local commit history together
- The merge commit will have two parents

If both history modified the same file in different ways

- Git cannot automatically decide which version to use in the merge commit
- Let the developer decide by resolving a merge conflict

So merge conflicts are messy! How do we avoid them?

- Push and pull less
- Create other lines of development (branches)

Branches are pointers to commits.

- Create new branches with `git branch <branch-name>`
    - All repos start off with one branch, normally `main` branch
- Can set the active branch (`HEAD`) with `git switch <branch-name>`
    - When we create a commit, we move only the active branch forward to point to the new commit
    - Does not affect other branches
- `git merge <branch-name>` merges the specified branch into the active branch and creates a merge commit

Feature branches:

- ALL development occurs in branches apart from `main`
- Name feature branches according to the feature being developed
- Once we finish developing, merge into the `main` branch
- **Important**: Test your changes on your feature branch as much as possible!!!

Pull Requests

The `main` branch is often reserved for deployment.

Pull requests (PR): way to request that your changes be merged into `main`.

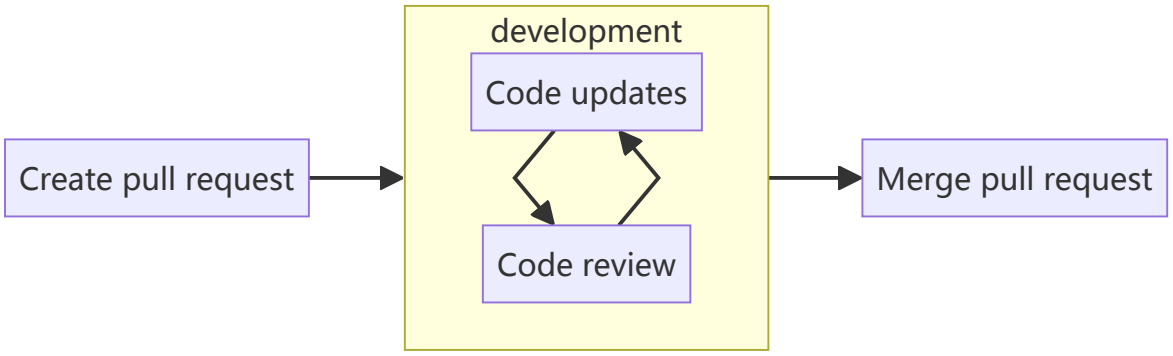
- Developers can review your changes, leave feedback (code review), and approve the request
- Good practice for others to review your code before it's merged

- Most major tech companies mandate that at least one additional person review your code

You can contribute to open-source projects (e.g. Firefox) by making a pull request

- A maintainer will review your changes and provide feedback.

The goal of a PRs is to get feedback on your code.



Collaboration is the key.

Lec 32: Software Engineering III

Using Cursor and other LLM tools to help you code more efficiently.

Always embrace the new thing.

Make the code more narrative.

There are two primary sources of complexity:

- Dependencies: When a piece of code cannot be read, understood, and modified independently.
- Obscurity: When important information is not obvious.

DO NOT create mutable static variables.

Deep Modules: Hiding complexity by using helper functions. A system should be broken into modules, and each module is completely independent.

Lec 33: Quick Sort

Partition

Core idea: Partitioning

A partition of an array, given a pivot x, is a rearrangement of the items so that:

- All entries to the left of x are less than or equal to x
- All entries to the right of x are greater or equal to x

A Good Algorithm: Hoare Partitioning

- Create L and G pointers at left and right ends:
    - L pointer is a friend to small items, and hate large or equal items

    - G pointer is a friend to large items, and hates small or equal items
    - Walk pointers towards each other, stopping on a hated item
        - When both pointers have stopped, swap and move pointers by one
        - When pointers cross, you are done walking
    - Swap pivot with G.

Quick Sort

Quick sorting N items:

- Partition on leftmost item
- Quicksort left half
- Quicksort right half

The more sorted your array is, the slower quick sort will be.

Theoretical analysis:

- Best case: $\Theta(N \log N)$
- Worst case: $\Theta(N^2)$
- Randomly chosen array case: $\Theta(N \log N)$ expected

Avoiding the Worst Case

Approach 1: Add some randomness (Preferred by professors)

- Picking pivots randomly
- Shuffle the array (so its not sorted)

Approach 2: Select the pivot more smartly

- The best possible pivot is the median

Approach 3: Switch to another algorithm when taking long time

- Detect the recursion depth

Approach 4: Predetermine whether the array is good for quick sort

Find the median

We can use the quick select algorithm to find the median of the array, to help pick a better pivot:

A median algorithm called BFPRT (a.k.a. Median of Medians).

Lec 34: Sorting and Algorithmic Bounds

Sorting

Sorting is a fundamental problem.

Today, we are trying to prove we've done well enough to sort.

Comparison sorts: Sorts based on `compareTo` and `swap` operations.

- e.g. Selection sort, Quick sort, Merge sort, Insertion sort, Heap sort

Non-comparison sorts:

- e.g. BOGO sort, Sleep sort, Stalin sort (destroy unsorted items, then everything is sorted), Miracle sort (wish for a miracle to happen), Radix sort, Gravity Sort.

Important Math:

$$\log N! \in \Theta(N \log N)$$

$$N \log N \in \Theta(\log N!)$$

Prove this using $N! > \frac{N}{2}^{\frac{N}{2}}$

The Ultimate Comparison Sort (TUCS)

Let **the ultimate comparison sort (TUCS)** be the asymptotically fastest possible comparison sorting algorithm, possibly yet to be discovered, and let $R(N)$ be its worst-case runtime.

After several analysis, we confirm the $R(N)$ should be within $\Theta(N)$ and $\Theta(N \log N)$.

Determine the Counterfeit Coin

9 identical coins. 1 coin, which is counterfeit, is slightly heavier.

To find the counterfeit coin:

- Step 1: Compare coins 1 2 3 vs 4 5 6
  - If equal, the coin is between 7 8 9
  - If not equal, the coin is between either 1 2 3 or 4 5 6.
- Step 2: Compare 2 coins from the 3 coin set.

What if 10 coins? Impossible:

- 2-layer decision tree has only 9 leaves that leads to the result.

In general, any decision tree with K layers and a branching factor of R (3 in this case, which is heavier, lighter, or equal) has at most R^K universes, where each universe yields a different expected return value.

Determine the order of A, B and C

You have a scale, and three items with different weights: A, B and C.

To determine the ordering of the three items: the minimum decision tree height should be 3.

- Which means, every order could be yielded within 3 steps of scaling

- Why? 3 items $3! = 6$ different ordering --> The decision tree should have at least 6 leaves --> minimum decision tree height of 3

N items, $\Omega(\log N!)$ -> $N \log N$.

However, we can use sorting. It **reduces** to sorting.

The Sorting Lower Bound

Any **comparison based sort** requires at least order $N \log N$ comparisons in its worse case.

Lec 35: Software Engineering IV

The best modules are those whose API is much simpler than their implementation.

API

The API for a Java class has both a formal and an informal part:

- Formal: The list of method signatures
- Informal: Rules for using the API that are not enforced by the compiler
    - Can only be specified in comments

The more informal rules there are in your API, the more complex it will be.

Information Hiding and Information Leakage

Embed knowledge and design decision into the module itself, without exposing them to the outside world.

Hiding information keeps the API simple.

The opposite of information hiding is information leakage.

It occurs when design decision is reflected in multiple modules.

Life Advice

The hedonic treadmill.

Enjoy youth.

Lec 36: Radix Sort

Stability of a Sorting Algorithm

A sort is stable if it preserves the relative order of equal elements in the input array.

Sorting instability can be really annoying!

Insertion sort is stable.

Quick sort is not stable depending on your partitioning strategy.

In Java, `Array.sort(someArray)` uses:

- Merge sort (specifically the Tim-Sort variant) if it is `Object[]`.
- Quick sort if it consists of primitives.

Digit-by-Digit Sorting

We can sort numbers digit by digit.

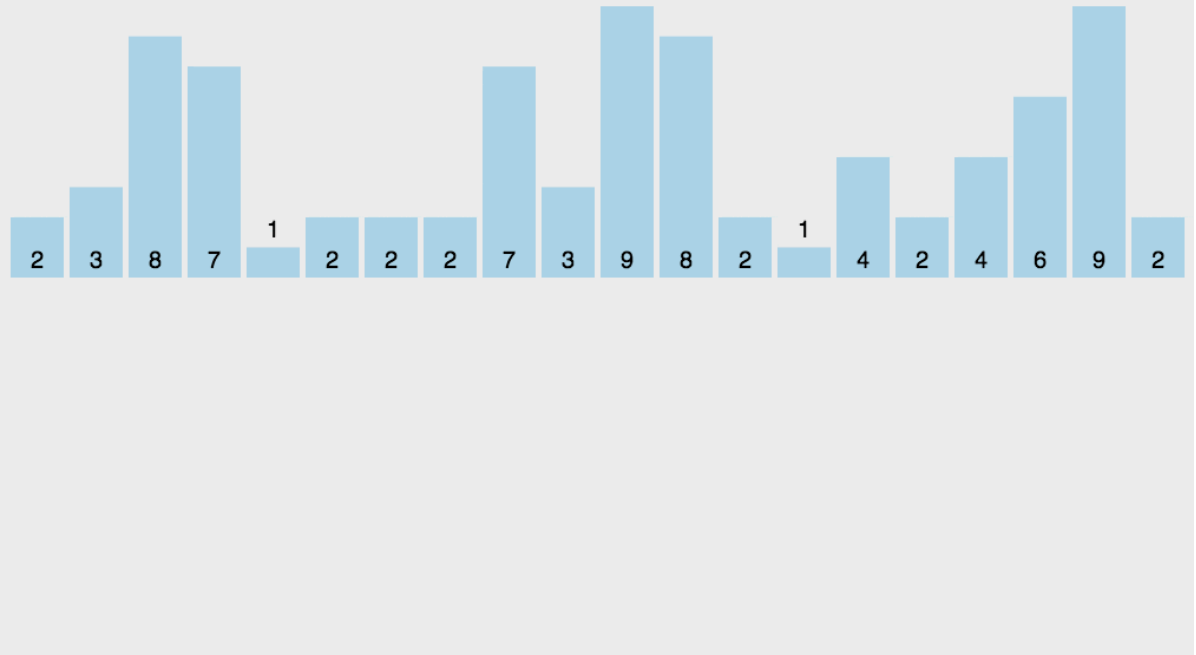
First stably sort according to the rightmost digit, next sort according to the rightmost but one digit...

The sorting algorithm used should be stable to work.

Counting Sort

Counting sort is a non-comparison sort.

1. Counts occurrences of each value in the input.
2. Computes cumulative counts to determine each value's position in the sorted output.
    - Determine the size of the output array by summing the occurrences.
    - Do some clever math.
3. Places elements into their correct positions using these counts.



It doesn't use comparison, because it takes advantage of the already sorted array index.

- e.g. when iterating through an array, `array[8]` comes always after `array[2]`

Total runtime on N keys with alphabet of size R: $\Theta(N + R)$

Memory usage: $\Theta(N + R)$

Counting sort will be way better if R is a small number.

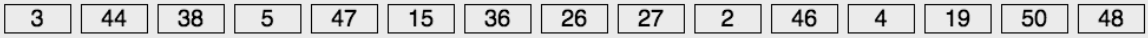
It is stable.

LSD Radix Sort

Use **counting sort** as the stable sort in the **digit-by-digit sort**, we can get LSD Radix Sort.

- LSD: Least Significant Digit

Radix: In a positional numeral system, the radix or base is the number of unique digits, including the digit zero, used to represent numbers.



Given N as number of items, R as size of alphabets, W as width of each item in number digits:

- Runtime: $\Theta(WN + WR)$
- When dealing with numbers, R is 10 (from 0 to 9).

MSD Radix Sort

`9xxxx` is always greater than `3xxxx`, without caring about the following `x`s!

Key idea: Sort each subproblem separately.

The algorithm works by:

1. **Grouping elements** based on their most significant digit (e.g., the first character of a string, the leftmost digit of a number).
2. **Recursively sorting** each group using the next significant digit (e.g., the second character, the next digit to the right).
3. **Combining the sorted groups** to produce the final sorted result.

Runtime: Best-case $\Theta(N + R)$, worst-case $\Theta(WN + WR)$.

Lec 37: Sorting Conclusion, Algorithm Design Practice

Sorting Conclusion

Merge sort takes $\Theta(N \log N)$ comparisons. If the comparison doesn't take constant time, the runtime will be different.

So, for string of length W, the runtime of merge sort is between $\Theta(N \log N)$ and $\Theta(WN \log N)$.

Treating alphabet size as constant, LSD sort has run time $\Theta(WN)$.

Which is better?

- LSD sort
  - Sufficiently large N
  - If strings are very similar to each other
- Merge sort
  - If strings are highly dissimilar from each other

For integers, we don't have a `charAt()` method. So, in LSD radix sort:

- Approach 1: Convert to a string
- Approach 2: Implement `getDthDigit()`

And there is no need to stick with base 10.

Algorithm Design Practice

Taking advantage of the algorithms you already know.

e.g. You have a graph whose edges are values between 0 and 1 (not inclusive), a start vertex, and an end vertex. Find the path whose edges multiply to the largest value.

- Create a new graph, but replace each edge of weight w with an edge of weight $\log \frac{1}{w}$
- Run Dijkstra to get the shortest path:
  - Smallest $\log \frac{1}{w_1} + \log \frac{1}{w_2} + \log \frac{1}{w_3} + \ldots + \log \frac{1}{w_n} = log \frac{1}{\Pi w_i}$ works the same way as biggest $w_1 w_2 w_3 \ldots w_n$.

Reducing is important.

Lec 38: Compression

The original file is unchanged via `zip` and after that `unzip`. But how?

In a lossless algorithm, we require that no information is lost.

Information Theory

Less information = Easier to memorize (generally)

The Shannon Entropy of a dataset is a measure of how predictable a dataset is.

- For strings, informally, it's a measure of how many possible strings exist with that characteristic.

Prefix Free Codes

Morse code is not very good. The same string can have multiple representations of English words.

Alternate Strategy: Avoid ambiguity by making code prefix free.

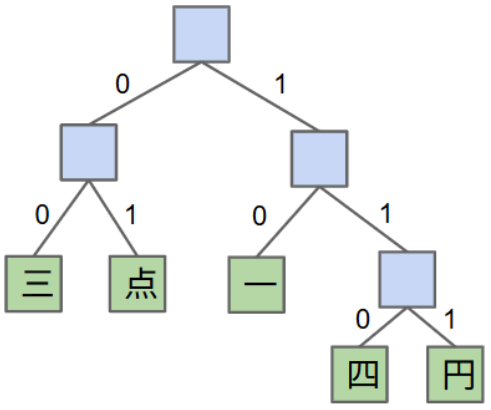
A prefix-free code is one in which no codeword is a prefix of any other.

We can create a algorithm that creates a prefix-free code based on the original data.

Shannon Fano Coding

- Count relative frequencies of all characters in a text.
- Split into "left" and "right halves" of roughly equal frequency.
    - Left half gets a leading zero. Right half gets a leading one.
    - Repeat.

Symbol	Frequency	Code
三	0.35	00
点	0.17	01
一	0.17	10
四	0.16	110
円	0.15	111



It is not optimal.

Huffman Coding

Calculate relative frequencies.

- Assign each symbol to a node with weight = relative frequency.
- Take the two smallest nodes and merge them into a super node with weight equal to sum of weights.
- Repeat until everything is part of a tree.

As it turns out, the Huffman code is the optimal.

Data Structures for Huffman Coding

We use a trie. We need to look up longest matching prefix, an operation that tries excel at.

Given a file `X.txt` that we'd like to compress into `X.huf`:

- Consider each b-bit symbol (e.g. 8-bit chunks, Unicode characters, etc.) of `X.txt`, counting occurrences of each of the 2^b possibilities, where b is the size of each symbol in bits.
- Use Huffman code construction algorithm to create a decoding trie and encoding map. Store this trie at the beginning of `X.huf`.
- Use encoding map to write codeword for each symbol of input into `X.huf`.

To decompress `X.huf`:

- Read in the decoding trie.
- Repeatedly use the decoding trie's `longestPrefixOf` operation until all bits in `X.huf` have been converted back to their uncompressed form.

In general, no compression algorithm can compress below the entropy of a dataset.

The more predictable data is, the less information it carries, and therefore, the better we can compress that data.

Lec 39: Complexity and P vs NP

If we have a solution to any one of these three problems, we can create a solution to the other two. They are called **equivalent under Turing Reduction**.

We will call functions that return T/F **decision problems**.

Deterministic Turing Machine vs Nondeterministic Turing Machine

Any decision problem in theoretical CS is said to run on a Turing Machine.

Anything that can be solved by a computer program can also be solved by a Turing Machine.

A Turing Machine has infinite memory.

A programming language is said to be **Turing-Complete** if any Turing machine can be simulated with a program written in that language.

- Redstone in Minecraft is Turing Complete.

A Deterministic Turing Machine (DTM) is a Turing machine without randomness involved, called DTM for short.

P is the set of all decision problems solvable by a DTM within polynomial time.

A **Nondeterministic Turing Machine** or **NTM** is a Turing Machine with one additional operation that can be performed: guessing.

Informally, we allow an operation `int guess(int min, int max)` that returns a number between min and max.

- If ANY pattern of guesses ends up causing us to return True, then the nondeterministic Turing Machine returns True.
- If ALL guess patterns return False, then the nondeterministic Turing Machine returns False.

The set **NP** is the set of problems that can be solved in **polynomial time** with a NTM.

In general, solving a problem with a NTM boils down to those two steps:

1. Generate (nondeterministically) a random solution to the problem
2. Verify (deterministically) if that solution actually solves the problem

NP is also defined as the set of problems whose solution can be verified in polynomial time by a DTM.

We will call a problem NP-Hard if every problem in NP reduces to that problem.

If a problem is both NP and NP-Hard, it's called NP-Complete.

Any NP-Complete problem is the hardest problem in NP. They are equivalent under Turing reduction.

This is P=NP problem: Find a Turing reduction from an NP problem to P (P=NP), or prove that no reduction exists (P!=NP).

If P = NP is proven (even if we find an algorithm that takes $\Theta(n^{1000000})$ time:

- Every problem in NP collapses into P
- All modern cryptography breaks
- A fundamental assumption made by 89% of CS theorists is broken, and further improvements are likely

If P != NP is proven:

- An entire new branch of theory will likely be developed off those methods
    - There are currently a LOT of unsolved problems in complexity hierarchy
- Despite hundreds of people working on this problem, there's been basically no progress on solving this in the past decade.

Just because a problem hasn't been solved doesn't mean you can't find a solution.

The famous "Haruki Suzumiya" Problem.

Lec 40: Conclusion

Wrapping up CS61B

Josh wrapped things up in less than 12 minutes. Check out the lecture video.

Justin's Messages

This is the last lecture in CS61B for Justin.

Your grade tells you how hard your next semester will be.

The best way to learn CS from now is to make something by yourself, and to play with code.

Josh's Messages

AMA - Ask me anything.

Cramming can work in limited circumstances, not good for us.

Petition for Justin!

Special Thanks to the Staff!

Aug 29, 2025