CSC 115 Spring 2023 - Pre-Lecture Notes

CSC 115: Fundamentals of Programming: II

Based on the pre-lecture videos by Dr. Anthony Estey.

Course:

- Course Outline
- Course Content

Abstraction

During our **Interfaces** and **Abstract Data Type** (ADT) unit we discussed some of the benefits of hiding unnecessary details from clients/users. This allows us to manage the complexity of the project.

Clients/users should know what the system does and how to use it, but do not need to know the details about how it was implemented.

We can expand this notion when working on multiple components within the same software system as well.

Encapsulation allows a programmer to bundle all of the features of a component into a single unit, which we call a class in Java. And allows the programmer to restrict access to fields that do not need to be accessed or manipulated externally, while still providing access to certain methods.

This can also apply to hiding unnecessary details from other programmers, even within the same team!

Classes

Classes often provide an abstraction that hide internal implementation details (which are often unnecessary). When working with another class, all a programmer needs to know is which methods are available to call, and what input parameters need to be given in order to trigger their intended behaviour or result. Programmers don't need to know how each method is implemented.

This makes large systems that have many different programmers working on many different components much easier to use. It would be impossible for each program to learn all of the details of every other component, the design of the system would never get completed!

Key Takeaways

- Abstraction us used to reduce the complexity of a system.
- This is achieved by hiding unnecessary details about how an operation works.

ADT - List

Assume you wanted to create something that allowed someone to:

- Keep track of what groceries they needed to buy.
- Maintain information about all of their contacts.
- Record all of the courses they have completed as they progress through their undergraduate degree program.

Can these be generalized into a common set of required features?

The Notion of A List

A list allows us one to manage a collection or items. Elements can be inserted and removed in any order. Any element can be accessed at any given time by their position in the list.

ADT List Operations:

- Create an empty list.
- Determine whether a list is empty.
- Determine the number of items in a list.
- Add an item at a given position in a list.
- Remove the item at a given position in a list.
- Get the item at a given position in a list.
- Remove all items from a list.

Item are referenced by their position in a list.

Specifications of the Operations (Interface):

- Define the contract for the ADT list.
- Include the operations a list must be able to perform.
- Do not specify how to store the list or how to perform the operations.

Remember: the operations can be used in an application without knowledge of how the operations are implemented. Programmer implements a list using a data structure.

Arrays

Arrays are used to store multiple values in a single variable. They allow us to work with a collection of items.

If we know what values we want to put in our array, we list them when we declare the array.

```
type[] name = {<comma separated values>};
```

Otherwise, we can just declare a new array and specify the length of the array.

```
type[] name = new type[<size>];
```

Working with Arrays

Square brackets [] are used to specify which index within the collection of items is to be accessed.

Note: The first index in an array is index 0.

We can use the length property to determine how many elements an array has: `System.out.println(array.length);`.

When used in combination with a for loop we can easily visit each element in an array.

```
for (int i = 0; i < array.length; i++) {
    System.out.println(array[i]);
}
```

Big-Oh

Algorithm Growth Rates

We define the growth rate as the change in runtime as n changes.

For example: Let's say we have computed the number of statements in method relative to n to be: $2n^2 + 12n + 5$.

What can we say about the runtime for very large values of n. We will see that we can ignore some of the lower terms, as the growth rate will be dominated by the n^2 term. Thus, we say this algorithm run "on the order of" n^2, and write it as $O(n^2)$ (i.e., "Big-Oh of n-squared").

Asymptotic Analysis

In general, the algorithms we will see in this course will likely fall under one of the following categories:

- Constant -- $O(1)$
- Logarithmic -- $O(\log n)$
- Linear -- $O(n)$
- Quadratic -- $O(n^2)$
- Cubic -- $O(n^3)$

Array VS Linked List

When comparing the asymptotic runtime of the `addFront()` method we can see that...

- Linked List Implementation: $O(1)$
- Array Implementation: $O(n)$

Thus, the linked list implementation does outperform the array implementation for all operations.

Binary Search Trees

A **binary tree** is a tree with at *most* two children per node. The children are typically referred to as left and right.

Binary Search Tree

A **Binary Search Tree** (BST) is a binary tree with a special property:

- For every node n in the tree:
  - All node's in n's left subtree have keys less than n.
  - All node's in n's right subtree have keys greater than n.

BST Search

To find an element in a BST search beginning at node n:

- If the target key is less than n's key, then search n's left subtree.
- If the target key is greater than n's key, then search n's right subtree.
- If n's key is equal to the target value, return true (or a pointer to the data).

How many comparisons? One for each node on the path. Worst case the height of the tree + 1.

BST Insertion

The BST property must hold after each insertion: Observation from search is there is only one valid place to insert a new node.

Thus,

- The position is determined by performing a search.
  - If the search ends at the null left tree child of a node n, insert the new node so that it is n's left child.
  - If the search ends at null right child of a node n, insert the new node so that is is n's right child.

The runtime is also bound by the tree's height: $O(\text{height})$.

BST Insertion

```
TreeNode n = new TreeNode(val);

TreeNode cur = root;
if (val < cur.data) {
        if (cur.left != null) {
                cur = cur.left;
        } else {
                cur.left = n;
                return;
        }
} else if (val > cur.data) {
        if (cur.right != null) {
                cur = cur.right;
        }
}
```

BST Minimum Maximum Key

To find the minimum: From the root, follow left reference arrows until no more left children exist.

To find the maximum: From the root, follow right reference arrows until no more right children exist.

BST Removal

The BST property must hold after each removal.

Note: Removal is not as straightforward as search or insertion. With insertion the strategy is to insert a new leaf node. This avoids changing the internal structure of the tree. Unless a leaf node is removed, this isn't possible with the removal operation (and we don't know where the element we want to remove is located within the tree).

BST Removal Cases

There are a number of different cases we need to consider:

1. The node to be removed has no children.
    1. Remove the node.
    2. Assign its parent to point to null instead of the node.
2. The node to be removed has one child.
    1. Replace the node with the subtree rooted by the child.
3. The node to be removed has two children.

Leaf Node (No Children)

1. Locate the target.
2. Parent points to null instead of node.

Node with One Child

1. Locate the target.
2. Replace node with subtree rooted by child.

Looking At The Next Node

One of the issues with implementing a BST is the necessity to look ahead at both children in order to update the reference arrows. This is similar to singly-linked lists when we needed to look ahead for insertion and removal.

Looking Ahead

1. Locate the node to remove and its parent.
2. To make the correct link, we need to know id the node to be removed is a left or right child.

```
if (n == NULL) {
        return;
}
if (target < n.data) {
        parent = n;
        n = n.left;
        isLeftChild = true;
} else {
        parent = n;
        n = n.right;
        isLeftChild = false;
}
```

Node with Two Children

The most difficult case is when the node to be removed has 2 children. We can't just replace the node with its only child.

Which child should we replace the node with? Can we just arbitrarily pick one? What happens if the replacement nodes have children?

1. Locate the target.
2. Replace node with subtree rooted by child.

Replacement Nodes

When a node has two children, instead of replacing it with one of its children, it will need to be replaced with its **predecessor** or **successor**.

A predecessor is the rightmost node in a node's left subtree. The predecessor is the node in the left subtree with the highest key.

A successor is the leftmost node in a node's right subtree. The successor is the node in the right subtree with the lowest key.

The predecessor and successor are the only two nodes that can replace the node and maintain the binary search tree property. They are also good choices because neither of them have 2 children.

Predecessor Node

The predecessor of a node is the rightmost node in the node's left subtree. The predecessor cannot have a right child (or it wouldn't be predecessor).

In a BST, the predecessor is the largest value in the tree less than target node. Similarly, the successor is the smallest value in the tree greater than the target.

Predecessor and Successor

Both are good candidates to replace the node to remove with. They are easy to locate, they don't have two children, they maintain the BST property.

Pick one to use for a removal of a node with two children (pick either one, but be consistent)!

Node with Two Children

1. Locate the target.
2. Find the successor (or predecessor) and detach it from the tree.
3. Attach removed node's children to successor.
4. Make the successor the child of target's parent.

Classes and Objects

We sometimes want the information we work with in our programs to not be represented by a single number or text.

For example, we could represent a song with simply a title, but we may also want to know about the artist, duration, genre, etc..

Classes

Classes allow us to define our own data types by defining:

- their attributes (that help us differentiate one from another)
- and associated behaviour (what operations we cna do with it)

We can think of a class as the blueprint for an object. When we create an object we say that it is an instance of a class.

Each student object has access to the methods of the class. For example, if we did `student1.setProgram("History");` then the student1 object would call the setProgram method and update its program attribute.

Student Example

Student:

- Attributes
  - sID (String)
  - name (String)
  - gpa (double)
  - program (String)
- Behaviours
  - updateGPA (method)
  - setProgram (method)

Student 1:

- sID: "V00123456"
- name: "Ali Sa"
- gpa: 4.3
- program: "Mathematics"

The Comparable Interface

- Java's built in Comparable Interface:
  - Oracle Docs: Comparable

> This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's `compareTo` method is referred to as its natural comparison method.

Why implement the Comparable Interface? Often we want to be able to order the items in our collection. If the objects in our collection implement the Comparable class, we know the class has a `compareTo` method which allows us to order any two instances of the class.

Ordering Elements

Ordering elements is easy with integers... We know that the value 4 is less than the value 10. So, when determining order for a sorted list or heap we can use $<$ and $>$.

```
int first = 4;
int second = 10;
System.out.println(first < second); // outputs true
```

But, we can't use the $<$ to compare other types, like a String:

```
String first = "computer";
String second = "science";
System.out.println(first < second);

// Output: Error: Bad operand types for binary operator '<'
```

compareTo

`compareTo` allows us to compare two objects of the same type. Any class that implements the **Comparable** interface must have a `compareTo`. The String class implements **Comparable**!

- Java's built in Comparable Interface:
  - Oracle Docs: compareTo

```
int compareTo(T o)
```

Compares this object with the specified 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.

Usage

```java
String first = "computer";
String second = "science";

// We want to determine if first should be ordered
// before second.
if (first < second) { ... } // but this does not work :(

// We can do this with compareTo
if (first.compareTo(second) < 0) { ... }

// Similarly, if we wanted to determine
if (first > second) { ... }
// as
if (first.compareTo(second) > 0) { ... }
```

Implementation Comparable

We will create some objects that implement the Comparable Interface.

This means we will need to include a `compareTo` method in the class. And we can define what fields to use form the object to determine how to order this object with another object of the same type.

Context-Preserving Accumulator

We have used accumulators to accumulate a result as we iterate through all of the elements in a collection.

Example: Sum up all the values in a list. We initialize sum to 0 and as we visit each element in the list, add its data value to sum and then return sum at the end.

We can also use accumulators to hold information for us that we might need later, but may not have direct access to currently.

Example: Comparing all elements to the first element, or to the element that comes before them.

```
Node cur = head;
while (cur != null) {
        System.out.print(cur.data));
        cur = cur.next;
}
```

What if we want to access data from an earlier node in the list?

Example

Non-Recursive

```
public boolean twoInARow() {
        if (this.size() <= 1) {
                return false;
        }

        Node prev = head;
        Node cur = head.next;

        while (cur != null) {
                if (prev.value == cur.value) {
                        return true;
                }
                prev = cur;
                cur = cur.next;
        }

        return false;
}
```

Recursive

```
public boolean twoInARow() {
        if (this.size() <= 1) {
                return false;
        }

        return twoInARowRecursive(Node prev, Node cur) {
                if (cur != null) {
                        return false;
                }

                if (prev.value == cur.value) {
                        return true;
                }

                return twoInARowRecursive(cur, cur.next);
        }
}
```

Data Abstraction

We have now seen that we can create interfaces in Java. Interfaces provide required methods, but omit the details of fields and how the methods are implemented. The separation of what something does (specification) and how it does it (implementation) is a fundamental concept in engineering!

Data Structures

A storage structure for data. There are different types of data structures.

It is a way of storing, accessing, organizing, and manipulating data using well-defined operations.

> A collection of data values, the relationships among them, and the functions or operations that can be applied to the data. - Wikipedia

The only data structure we have used so far is an array. There are other data structures we can use to perform the same operations on our collection of data, which are implemented in different ways (and have different speeds and memory requirements).

Abstract Data Types (ADTs)

An ADT is composed of:

- A description of what data is stored (but not how the data is stored).
- A set of operations on that data (but not how the operations are implemented).

Specifications of an ADT indicate what the ADT operations do (i.e., interface).

Implementations of an ADT include choosing a particular data structure (i.e., how is the data stored, accessed, organized, etc. (data structure)).

ADT Example

ADT Dictionary:

- Stores a pair of string, representing the word and definition (data).
- Operations: insert(word, definition), delete(word), find(word).

We use a data structure to implement an ADT.

We also know the effect operations have on the data. If we **delete** a word from the dictionary, a subsequent **find** operation should fail.

Client VS Programmer

- Clients know how to use something.
    - What operations are available and what they do.
- Programmers must decide how to implement the operations.
    - Their choices may be influence by a number of things:
        - execution speed
        - memory requirements
        - maintenance (debugging, scalability, etc.)

Doubly Linked List

A doubly linked list is a linked list where each node keeps a reference to both the preceding and following nodes in the chain.

```
public class Node {
        private int data;
        private Node prev;
        private Node next;
        ...
}
```

We can now traverse the list in either direction.

Insertion

- First: Determine where to insert the node.
- Second: Update pointers so that the order is correct.

```
Node n = new Node(20, null);

n.next = cur.next
n.prev = cur;

cur.next.prev = n;
cur.next = n;
```

Deletion

- First: Locate the element to remove.
- Second: Update pointers so that the deleted node is skipped.

```
cur.next.prev = cur.prev;
cur.prev.next = cur.next;
cur = null;
```

Encapsulation

When writing software to solve problems, we need to represent many different kinds of information.

Examples:

- The `x` and `y` position of a point in a graph.
- The `title`, `artist`, and `duration` of a song.
- The `name`, `ID`, `gpa`, and `program` for a student.

In Java, we have created classes to solve certain problems. For example, all of the data we need to represent student information can be written as fields within a Student class. The (non-static) methods in the class allow us to operate on that data. These methods make up the behaviours of the class.

One part of **encapsulation** is this bundling of data and code that operates on that data into a single unit (i.e., a class).

The other part is the fact that we can restrict access to that data.

Access Modifiers

In Java, we use **access modifiers** to control whether other classes can use a particular field or call a particular method.

The access modifiers we have seen so far:

- **private**: can only be accessed from within the same class.
- **protected**: can be accessed from within the same class, package, or folder.
- **public**: can be accessed from everywhere.

Encapsulation is used to hide certain values within a class. Values are accessed or modified through publicly accessible methods.

Key Takeaways

- Classes in Java allow us to bundle data fields and operations into a single unit.
- Encapsulation is used to hide the values of the data fields inside a class.
- Some data fields may be accessed or updated through public setters and getters.

Equality

Often we want to check for equality in our programs. For example, searching through a database for a particular item or searching through a list to count the number of occurrences of some particular item.

We use the `==` to determine if two primitive type variables are equal to one another.

However, this only works for variables of type int, double, boolean, etc.. It is not quite as simple for Strings, arrays, or objects.

It all comes down to how the data is stored in memory.

Primitive Types

```
int num1 = 8;
int num2 = 5;
int num3 = 8;
```

Thus, if we check `num1 == num2` we get `false` and if we check `num1 == num3` we get `true`.

Strings, Arrays, and Objects

The same thing happens using `==` still compares two values in memory, but we need to consider what the values of our variable are.

Remember: that for these types the variable is referencing a location in memory where the data we associate with the particular object is store.

So although it is possible to use `==` to compare two objects, the operation likely isn't doing what we intended it to do.

Example

Instead of using `==` we will add a method to the Student class that allows us to determine if two students are equivalent.

```
public String getSID() {
    return sID;
}

public boolean equals(Student student) {
    return this.getSID().equals(student.getSID());
}
```

Note: the String class also has an equals method.

Summary

We want to call the `equals` method when comparing two objects, not `==`.

```
Student s1 = new Student("V00123456", "Ali Sa");
Student s2 = new Student("V00123456", "Ali Sa");

s1 == s2; // false
s1.equals(s2); // true
```

Errors and Debugging

There are three kinds of errors when writing code in Java:

- Syntax Errors
- Runtime Errors
- Logic Errors

Syntax Errors

When we compile our code, syntax errors are reported by the compiler.

If a syntax error occurs, the compiler does not create a new executable file, so we will not be able to run our programs until we have fixed all syntax errors in our code.

Runtime Errors

Errors that occur when the program is running are called runtime errors.

There is nothing wrong with the syntax, but the way we are trying to use a variable or object is incorrect.

Logic Errors

Logic errors occur when a program successfully compiles and executes, but the result is incorrect (it does the wrong thing).

Typically we write tests to detect logic errors. After running the tests, logic errors can be identified and fixed based on the failing tests.

Exceptions

An exception is an error message that halts the program during execution and outputs the contents of the issue.

Example

```
Exception in thread "main" java.lang.NullPointerException
        at ExceptionExample.main(ExceptionExample.java:12)
```

Motivation

At the moment, our exceptions result in the program "crashing". However, we would like to be more graceful with the way we handle errors.

Types of Exceptions

There are two types of exceptions that can occur in a Java program: **unchecked** exceptions and **checked** exceptions.

The main difference between these two types of exceptions are that the Java compiler checks for one type during compilation, but not the other.

Unchecked Exceptions

Unchecked exceptions occur when the program was able to compile without problem, but during execution a problem occurs.

Examples: ArrayOutOfBoundsException, NullPointerExceptions, StackOverflowException

ArrayOutOfBoundsException

The compiler doesn't ensure that all array indexes stay within the bounds of the array.

Why not? The index being accessed might be specified by user input or read in from a webpage or input file. It's impossible to know ahead of time when this might occur.

Checked Exceptions

Checked exceptions are exceptions that the Java compiler forces us to handle in our program in order for it to compile without issues.

Example:

- When reading from a file, we need to specify what to do if the file cannot be found (FileNotFoundException) or the program will not compile.

```
Scanner fileReader = new Scanner(inputFile);
```

Error

```
Error: unreported exception FileNotFoundException; must be caught or
declared to be thrown.
```

Thus, we must either state that we choose not to handle the exception (which would be accepting that the program will crash if the exception occurs). OR declare that we will catch the exception, for which we write code that will be executed if the exception occurs (and hopefully handles the exception gracefully). Either way, we need to specify what to do if the exception occurs.

Option - Throw Clause

The **throws** clause allows us to specify that we are aware an exception could occur within a block of code, and that we choose to ignore it.

Sometimes this is compared to a waiver of liability form: "I hereby agree that this method might throw an exception, and I accept the consequences (that the program will **crash**) if this happens."

Option - Exception Handling

Exception Handling allows us to change the default behaviour of a program when something does wrong, by specifying what should be done when a specific error occurs.

Exception handling typically has two parts:

1. **A try block**: where we try and do something that may cause an error.
2. An **exception handler**: where we indicate what should happen if the error occurs.

General Form

An exception is an error that occurs while the programming is running, causing the program to abruptly halt. The try/catch statements are used to gracefully handle exceptions.

The code in the try block is executed, and if no errors occurs, the code in the catch block is skipped.

If a statement in the try block raises an exception, then the code in the catch block is immediately executed, and any remaining code in the try block is skipped.

```
try {
        statment
} catch (theException exception) {
        statement
}
```

Fields and Constructors

Fields

Fields are another name for a class attributes. Fields can be accessed and updated like variables:

```
// Output
System.out.println(fieldname);

// Modify
fieldname = newvalue;
```

Initializing Objects

We can use the new keyword to allocate memory for a new instance of a class.

```
Student s1 = new Student();
s1.sID = "V00123456";
s1.name = "Ali Sa";
```

Note: there is a way to initialize the values of an object's field when it is first declared.

```
Student s1 = new Student("V00123456", "Ali Sa");
```

The lines of code above both call the Student class' Constructor.

Constructor

The constructor initializes the field values when an object is created.

The constructor is automatically called when the new keyword is used.

```
public Name(parameters) {
    statements;
}
```

Note: a constructor looks similar to a method, with some key differences:

- the name of the constructor always matches the class name
- no return type is specified
    - constructors implicitly return the new object being created

If a class has no constructor, Java gives it a default constructor with no parameters that sets all fields to 0 or null.

```java
public class Student {
    // Fields
    String sID;
    String name;
    double gpa;
    String program;

    public Student(String sID, String name) {
        this.sID = sID;
        this.name = name;
        gpa = 0.0;
        program = "Undeclared";
    }
}
```

Generics

Motivation

Let's look at some of the **Node** classes we have used so far:

```java
public class Node {
        private int data;
        protected Node next;

        public Node() {...}

        public int getData() {
                return data;
        }
}
```

and

```java
public class Node {
        private String data;
        protected Node next;

        public Node() {...}

        public String getData() {
                return data;
        }
}
```

And we see that the only difference is `int` and `string` for each implementation of data.

What about our **List** classes?

```java
public interface IntegerList {
        void addFront (int val);
        void addBack (int val);
        int size ();
        int get (int position);
}
```

and

```
public interface StringList {
        void addFront (String val);
        void addBack (String val);
        int size ();
        String get (int position);
}
```

The operations for each method would be the same, only the type of data associated with each list element is different.

What about if we want to make a list of **Student** objects, or **Songs**? It doesn't make sense to duplicate almost all of the code in the Node and List classes every time the type of data changes. There must be a way we can program our lists so that they are more generic.

Generics in Java

Generics allow the development of classes and interfaces without needing to decide **types** until the class is actually being used.

- Definition of the class or interface is followed by `<T>`. `T` represents the data type that the client code will specify.

General Form

```
public class Name<T> {
        public T variable;

        public Name(T var) {
                this.variable = var;
        }
}
```

And to create an instance of the class:

```
Name<T> x = new Name<T>(value);
```

Specific Form

```
public class Node<T> {
        public Node<T> next;
        public T data;

        public Node(T data, Node<T> next) {
                this.data = data;
                this.next = next;
        }
}

Node<int> a = new Node<int>(5, null);
Node<int> b = new Node<int>(7, a);

Node<String> c = new Node<String>("A", null);
```

Hash Collisions

Load Factor

$$\alpha = (\text{number of elements})/(\text{size of array})$$

Separate Chaining

Each entry in the hash table is a pointer to a linked list (or other dictionary-compatible data structure). If a collisions occurs the new item is added to the end of the list at the appropriate location.

Performance degrades less rapidly using separate chaining with uniform random distribution, separate chaining maintains good performance even at high load factors (i.e., $\alpha > 1$).

Open Addressing

When an insertion results in a collision, find an empty spot in the array. Start at the index to which the hash function mapped the inserted item. Look for a free space in the array following a particular search pattern, known as probing.

There are three major open addressing schemes:

- Linear Probing
- Quadratic Probing
- Double Hashing

Linear Probing

The hash table is searched sequentially. Starting with the original hash location. For each time the table is probed (for a free location) add one to the index. If the sequence of probes reaches the last element of the array, wrap around to $arr[0]$.

Problems

Linear probing leads to primary clustering.

The table contains groups of consecutively occupied locations. These clusters tend to get larger as time goes on, thus, reducing the efficiency of the has table.

Algorithm

```
Algorithm Find(k):
        Input: the target key to search the hash table for
        Output: the entry with the given key; null if key not found

        p <-- h(k) % m
        for i <-- 1 to m do
                e <-- data[p]
                if e is null then
                        return null
                end if
                if e's key is equal to k then
                        return e
                end if
                p <-- (p + 1) % m
        end for
        return null
end
```

Quadratic Probing

Quadratic probing is a refinement of linear probing that prevents primary clustering. For each probe p, add p^2 to the original integer index after each collision.

Example

- After first collision, look into table at index $h(k) + 1^2$.
- After second collision, look into table at index $h(k) + 2^2$.
- After third collision, look into table at index $h(k) + 3^2$.
- ...

The main idea is that unlike linear probing that checks indexes $h(x)$, $h(x) + 1$, $h(x) + 2$, etc. quadratic probing checks indexes $h(x)$, $h(x) + 1$, $h(x) + 4$, $h(x) + 9$, etc..

Problems

Quadratic probing can result in something called secondary clustering where keys that hash to the same index all go up in the same sequence. This delays the collision resolution for those keys, and sometimes quadratic probing will never find a valid index even when the table is not full.

Thus, quadratic probing solves the problem of primary clustering, but introduces a new problem of secondary clustering.

Secondary clustering is not a problem if:

- the data (keys) are not significantly skewed
- the hash table is large enough (i.e., has enough free space)
- the has function scatters the keys evenly across the table

Quadratic probing may start to fail at $\alpha > 0.5$.

Double Hashing

In both linear and quadratic probing, the probe sequence is independent of the key.

- for linear probing, the probe sequence increases by 1 index each time
- for quadratic probing, the probe sequence jumps 1, 4, 9, etc. from the original index

Double hashing produces key dependent probe sequence (i.e., a second hash function, h_2 , determine the probe sequence).

The second hash function, h_2, should follow these guidelines:

- $h_2(k) \neq 0$
- $h_2 \neq h_1$
- A typical h_2 function is $h_2(k) = p - (k \ mod \ p)$ where p is a prime number $< m$.

Recap

- for double hashing, the number of positions the sequence jumps each time from the original index is specified by a secondary hash function

Tips

It is best for both the table size, m, and the secondary hash function to be prime numbers. We don't want the secondary hash function to produce a number that is a multiple of the table size.

Hash Table Removal

Removals add complexity to hash tables. It's easy to find and remove an element, given the hash function. After an element is removed from the table, the space becomes unoccupied. The now empty location may make a probe sequence terminate prematurely.

Problem

When we remove an entry, it may make a probe sequence terminate prematurely. This means that the find algorithm may determine an entry isn't in the hash table when it actually is!

Solutions

- Search through the whole has table.
- Rehash items after each removal.

Both solutions are not ideal because they run in O(n) time and not O(1).

Tombstone

Mark table locations where an item has been removed. The find algorithm does not identify tombstones as null.

If the table becomes clogged with tombstones, then we can re-hash the entries (but it's inefficient). It's okay if the runtime inefficient re-hashing operation is only performed infrequently.

Hash Tables

Generally, we want to be able to efficiently insert, find, and remove elements from our collection.

	Insert	**Remove**	**Find**
Unordered Array	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(n)	O($\log n$)
Unordered List	O(1)	O(n)	O(n)
Ordered List	O(n)	O(n)	O(n)
BST	O(height)	O(height)	O(height)

BST Operation Run Time

BST operations are bound by the height of the tree.

- Worst Case: Tree is not balanced - O(n)
- Best Case: Tree is balanced - O($\log n$)

Array Operatons

Arrays have efficient O(1) access. we can also assign a value to an array at a specific index in O(1) time. However, to maintain order, we need to shuffle all other elements.

Hash Tables

Hash tables consist of an array to store data. We can complete insert, find, and remove operations in O(1) time.

Data often consists of complex types, or pointers to such objects (i.e., one attribute of the object is designated as the table's key).

A hash function maps the key to an array index. The key is converted to an integer. The integer mapped to an array index using some function (often the modulo function).

Hash Functions

Using knowledge of the kind and number of keys to be stored, we should choose/design a hash function so that it is:

- fast to compute
- causes few collisions

A collision occurs when there is already an element in the array at the index we wish to insert into. There are a number of ways to handle collisions efficiently that we will explore.

Example

For data with numeric keys, we might use `hash(x) = x mod m`, where m is the size of the table (e.g., assume m is larger than the number of keys we expect to store).

Assume $m = 7$:

```
// hash(x) = x mod 7

insert(4)
insert(17)
find(12)
insert(9)
delete(17)
```

		9	17	4	12	
0	1	2	3	4	5	6

Collision Handling

A collision occurs when two keys are mapped to the same index. Collisions may occur even when the hash function is good. Inevitably if enough elements are added to the has table.

Collisions start happening more frequently with higher load factors (i.e., load factor, α, is defined as $\alpha =$ (number of elements)/(size of array)).

There are two main ways of dealing with collisions: open addressing and separate chaining.

Heaps

The Notion of a Priority Queue

A collection of items characterized by the way it is organized to allow for fast access to and removal of the element with the smallest (or largest) key. Element with the highest priority is the first element to be removed (although it is named a priority queue, it is not FIFO).

The Priority Queue ADT

The priority queue ADT specifies the following operations:

- `insert(o, k))`: insert object `o` with key `k` into the collection
- `removeMin()` or `removeMax()`: removes the element with the minimum (or maximum) key from the collections
- `isEmpty()`: Determines whether the collection is currently empty

In general , a single priority queue supports only `removeMin` or only `removeMax`, but not both.

Note: We can implement a priority queue with a array or a linked list, but either the `insert` of the `removeMin` operation would not be efficient.

The Heap Data Structure

A heap is a realization of a priority queue that is efficient for both insertion and removal of minimum or maximum values.

A heap satisfies the following properties:

- **Heap-Shape Property**: a heap is a complete binary tree.
- **Heap-Order Property**: for every node v other than the root, the priority of the key stored at v is less than or equal to the key stored at v's parent

Note: All levels are filled down to the bottom level, and it is filled from left to right.

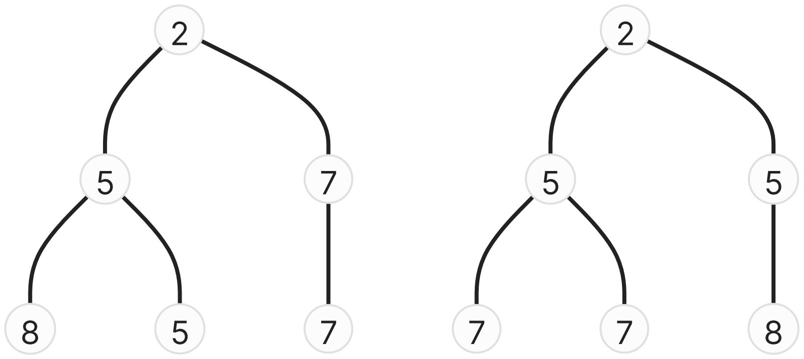
Heaps are partially ordered, they maintain a relationship between parent and child, but not across children.

Duplicate Keys, Different Ordering

We will need to consider the Heap-Order Property when we implement the insertion and removal methods.

Observation: two binary heaps can contain the same data, but the elements may appear at different positions within the structure.

Remember: Heap-Order Property looks at parent-child relationships.



Height of a Heap

The Heap-Shape Property is important for runtime analysis.

Theorem: A heap sorting n keys has height $O(\log n)$.

- Let h be the height of a heap storing n keys.
- Since there are 2^i keys at level i for $i = 0, \cdots, h-2$ and at least one key at depth $h-1$, we have $n \geq 1 + 2 + 4 + \cdots + 2^{h-2} + 1$.
- Thus, $n \geq 2^{h-1}$, i.e., $h \leq \log n + 1$.

Heap Implementation

When we implement an array using a heap, we can store the values beginning and index 0 or index 1.

- Starting at index 1 wastes part of the array (index 0), but makes it easier to access the indexes of a node's children or parent.
- Starting at index 0 is better for memory utilization, but is a little more complicated to access the indexes of the children or the parent of a node.

Heap Using 1-Based Array

- Index of a Child:
  - Left: $2(i)$
  - Right: $2(i) + 1$

- Index of a Parent:
  - Parent: $\lfloor \frac{i}{2} \rfloor$

, where i is the current index of the node.

Heap Using 0-Based Array

- Index of a Child:
  - Left: $2(i) + 1$
  - Right: $2(i) + 2$
- Index of a Parent:
  - Parent: $\lfloor \frac{i-1}{2} \rfloor$

, where i is the current index of the node.

Heap Insertion

Heap properties need to be maintained.

- Heap-Shape Property:
  - Insert the element at the first available spot (this equates to the left-most index at the bottom level of the tree).
  - Therefore, the heap remains a complete binary tree.
- Heap-Order Property:
  - "Bubble" the element up the tree until the heap-order property is restored.
  - Repeatedly compare with parent and swap until the ordering is restored.

Heap Removal

Heap properties need to be maintained.

- Heap-Shape Property:
  - Swap the root with last element in the array (the right-most element on the bottom level of the tree).
  - The last element is removed, and the heap remain a complete binary tree.
- Heap-Order Property:
  - "Bubble" the element down the tree until the order property is restored.
  - Repeatedly compare with children and swap until the ordering is restored.

Heaps - Recap

We have seen that the height of a balanced binary tree is $O(\log n)$.

So the number of times the **Bubble Up** and **Bubble Down** operations will occur during an insertion or removal are $O(\log n)$. Which means that heap insertion and removal are $O(\log n)$ operations.

If and Switch Statements

If Statements

Specific a block of code to be executed if a condition is true.

- else if: specify a new condition to test, if the first condition is false.
- else specify a block of code to be executed if all prior conditions are false.

```
if (condition) {
    ...
}
```

Note: It always starts with an if statement, followed by as many additional else if statements, followed by a single else statement.

Each condition is checked, one at a time until a condition evaluates to true. Only one code block is ever executed (the one for which the condition was true). If no conditions evaluate to true, then the else code block is executed.

Operators for Conditions

Operator	Meaning	Example	Result
>	greater than	x > y	True if x is greater than y, False otherwise.
<	less than	x < y	True if x is less than y, False otherwise.
>=	greater than or equal to	x >= y	True if x is greater than or equal to y, False otherwise.
<=	less than or equal to	x <= y	True if x is less than or equal to y, False otherwise.
==	equal to	x == y	True if x is equal to y, False otherwise.
!=	not equal to	x != y	True if x is not equal to y, False otherwise.
&&	and	x< y && x < z	True if x is less than y AND x is less than z.
||	or	x < y || x == z	True if x is less than y OR if x is equal to z.

Switch Statements

Switch statements provide the same functionality as nested else if statements, but with different syntax.

```
switch(expression) {
    case 0:
        ...
        break;
    case 1:
        ...
        break;
    default:
        ...
}
```

Note: the `break` statement exits the entire switch statement.

Inheritance

One of the reasons we write methods, or functions, is to reduce redundancy and code repetition. Sometimes when we create classes there is a lot of repetition too.

For example, within a school system there may be instructors, students, and staff members each with their own class. But all three of these classes share some things in common. This is one of the problems inheritance solves.

Java Inheritance Terminology

Inheritance: refers to the ability for one class to inherit from another.

When a class inherits from another, we say it **extends** the other class. A **subclass** extends (inherits from) a **superclass**. A subclass is often sometimes called a *specialization* of a superclass.

Subclasses

A subclass can:

- Add new fields in addition to those it inherits. A subclass inherits private fields, but cannot access them directly.
- Can call/invoke methods found in the superclass.
- Override an inherited method of its superclass. This occurs when a method in the subclass has the same method name and signature as a method in it superclass. Sometimes subclasses refine a method from the superclass.

Inheritance

Inheritance allows us to make the information in our projects much more organized and manageable.

- Subclasses inherit properties and behaviours from their parent class, allowing us to reuse existing code instead of writing new code for each class.

This can make it easier to implement new classes. The inherited features should already be testing and working correctly. Focus is shifted to updating behaviours and adding new features applicable to the subclass.

By default a subclass inherits all of the methods from its parent class. Subclasses can also override methods of its superclass to refine or change the behaviour for all instances of the subclass.

Terminology

There is some terminology commonly associated inheritance.

- The **is-a** relationship: When a class extends another class, there is an "is-a" relationship (e.g., a dog is-a mammal, a mammal is-a animal).
- The **has-a** relationship: This is related to composition instead of inheritance, as sometimes our objects might be composed of other objects. The fields in our object might be other objects we have created.

Key Takeaway

Inheritance allows us to reuse large amounts of code when a subclass inherits properties from its parent class.

Interface Implementation

General Structure

```
public interface name {
        /**
         * Purpose:
         * Parameters:
         * Returns:
         */
        public returnType method1();
        public returnType method2();
}
```

- Our classes begin with: `public class name`.
- Interfaces instead have: `public interface name`.

The methods are only signatures (they don't have a body). An interface doesn't do anything; it specifies desired behaviours. An interface typically does provide documentation for each method specified (comments).

General Implementation Structure

```
public class name implements interfaceName {
        public returnType method1() {
                statements;
        }
}
```

Classes that implement an interface specify which interface they implement)which contract they fulfill). The methods do have bodies, as they provide an implementation of the required operations.

Purpose of An Interface

What are the reasons we might want to use an Interface?

- Being able to guarantee a program (or suite of programs) all include certain features that work in a certain way is very useful.

Also think about it from a software development perspective:

- Interfaces only contain desired behaviours, no code.
    - This is the perfect medium for which clients and developers can communicate.
    - Clients can request behaviours and/or operations about their desired product; developers can discuss these requests, until an agreement is made.
    - Clients are not programmers, they are not interested in implementation details. They only need to see the interface (contract).
    - Developers then provide an implementation based on the contract (the clients will use the end product, but don't ever have to see the underlying code).

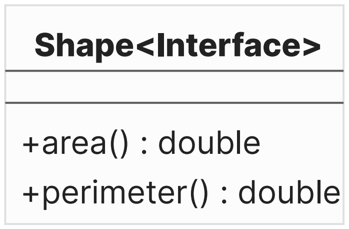
Interfaces

A Java interface specifies methods and constants but supplies no implementation details.

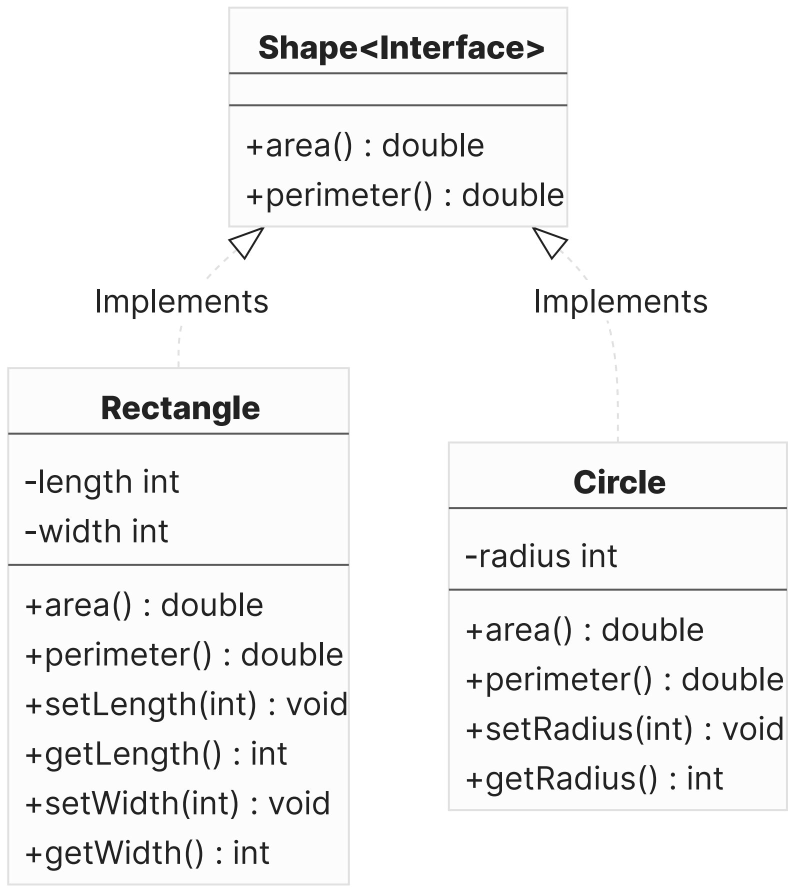
It can be used to specify some common behaviours that may be useful over many different types of objects.

- We can think of an interface like a contract. The person writing the interface (presenting the contract) says, "I want these features".
- The developer implementing the interface (fulfills the contract), replies, "Okay, I will build those features into the solution".

Interface Shape Example



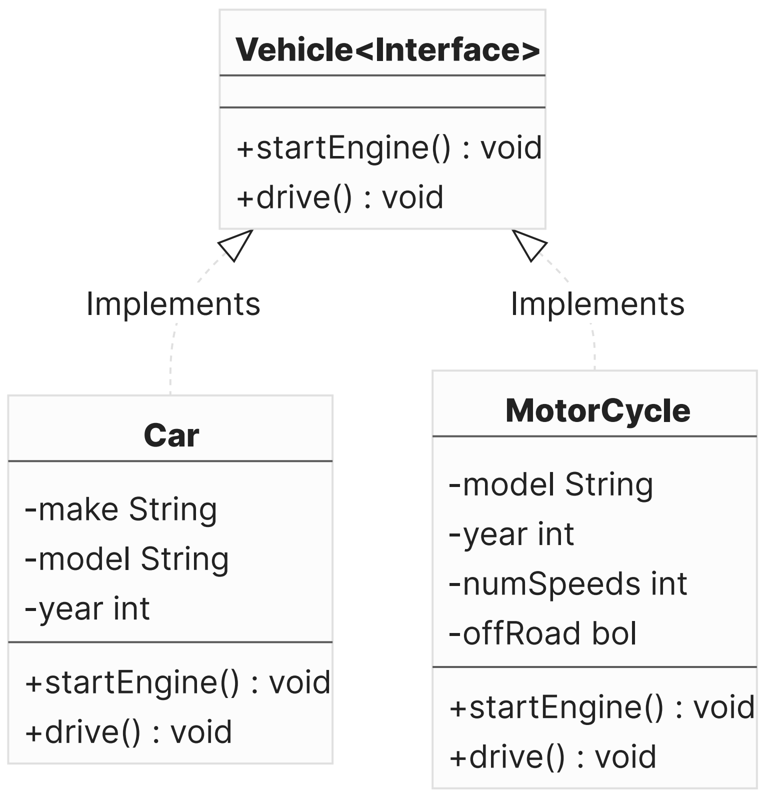
We must be able to calculate the area and perimeter of all shapes.



Both implementations fulfill the contract (we can get the shape's perimeter and area).

Interface Vehicle Example

We must be able to start a vehicle's engine, and it must be able to drive.


Vehicle<Interface>
+startEngine() : void
+drive() : void
Implements
Implements
Car
-make String
-model String
-year int
+startEngine() : void
+drive() : void
MotorCycle
-model String
-year int
-numSpeeds int
-offRoad bol
+startEngine() : void
+drive() : void

Iterators

Java has built in list classes. So, why would we use these?

- They have all of the list operations implemented for us already (tested and correct).
- They are generic, so we can use them with any types of data.

Why does this matter? Well we don't need to implement a list for every type of object we create for our programs.

Data Management and Organization

We have explored a number of different ways we can store and operate on a collection of data: linked list, trees, arrays, hash tables.

We have also discussed the pros and cons of each implementation (i.e., Big-Oh).

	Insert	**Remove**	**Find**
Unordered Array	O(1)	O(n)	O(n)
Ordered Array	O(n)	O(n)	O($\log n$)
Unordered List	O(1)	O(n)	O(n)
Ordered List	O(n)	O(n)	O(n)
BST	O(n)	O(n)	O(n)
BST (Balanced)	O($\log n$)	O($\log n$)	O($\log n$)

There also different ways to output the contents of our collections.

Iterators

Iterators allows us to loop through all of the elements in a collection.

- `forEachRemaining()`
- `hasNext()`
- `next()`
- `remove()`

Examples

```
LinkedList<Integer> entries = new LinkedList<Integer>();
```

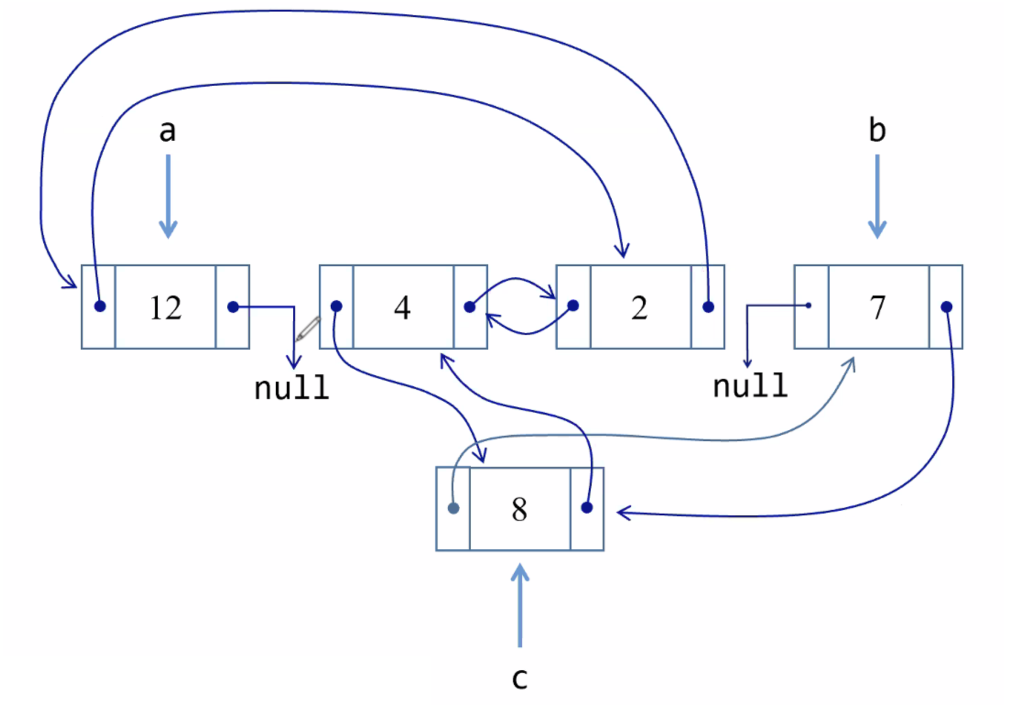
```java
entries.add(7);
entries.add(12);
entries.add(2);
entries.add(5);

Iterator<Integer> entriesIterator = entries.iterator();

while(entriesIterator.hasNext()) {
        Integer currentEntry = entriesIterator.next();
        System.out.println(currentEntry);
}
```

Linked List Example

An example of a doubly linked list.



```
a.next.prev = c;
c.prev = b;
a.prev = c.prev.prev;
a.next.next.next = a;
c.next = a.next;
a.next = b.next;
b.prev = null;
b.next = c;
```

Can we loop through each node one beginning at a node? What about starting a node and going to all the `.prev` ?

Linked List: Insertion and Removal

Insertion

Linked List:

15	16	19	22	28

New Node:

```
Node n = new Node(20, null);
```

We want to insert Node `n` into our linked list in ascending order.

- First: Determine where to insert the node.
- Second: Update the next pointers appropriately.

```
n.next = cur.next;
cur.next = n;
```

Note: The order of operations is important! The following would not insert Node `n` correctly into the linked list. Instead it will have Node `n` point to itself.

```
// Wrong Implementation
Node n = new Node(20, null);

cur.next = n;
n.next = cur.next;
```

Removal

Linked List:

15	16	19	20	22	28

We want to remove Node `n` from our linked list.

- First: Locate the element preceding the one to remove.
- Second: Update the next pointers so that the deleted node is skipped.

```
cur.next = cur.next.next;
```

Note: Java's garbage collection will delete any object that doesn't have a pointer.

Linked List Nodes

Motivation

Imagine we have chosen to implement the List ADT using an array... Assume our list currently has the following items in it:

3	5	6	7	9	13	14	16	20	
0	1	2	3	4	5	6	7	8	9

What would happen if we call `addFront(2)`? We would need to shuffle every element to the right by 1 index and then insert 2 in the front.

So, a linked list allows for fast insertion/removal from the front or back of a list without the need to shuffle all other items up or down an index.

Array List

- Array:
  - Allocate Space: `int[] data = new int[SIZE];`
  - Expand and Shrink Array: Allocate Space and then Copy Elements
  - Note: Java arrays have a fixed length.

Linked List

A linked list is a data structure composed of nodes linked together. A node is a data structure that contains:

- Data (what we store in the list).
- A pointer to the location of the next element in the list.
- Optional: A pointer to the location of the previous element in the list.

```java
public class Node {
        private int data;
        private Node next; // pointer to next element
        ...
}

Node a = new Node(7, null);
```

To create a new node and add it to the linked list...

```java
Node b = new Node(3, null);
a.next = b;
```

Iteration Implementation

We need to keep a reference to the head of the list. And then we can reach all subsequent elements.

```java
// For Loop Method
for (Node cur = head; cur != null; curr = cur.next) {
        System.out.println(cur.data);
}

// While Loop Method
Node cur = head;
while (cur != null) {
        System.out.println(cur.data);
        cur = cur.next;
}
```

List Implementation

In this unit, we will explore two ways of implementing a list:

- Using an array.
- Using a linked list.

Array Implementation

```
int[] data = new int[SIZE];
```

Positives:

- Easy to access and update array values.
- Fast operations.

Negatives:

- Memory usage:
    - Memory allocation.
    - Array copying.
    - Array expansion.
- Mainting element order:
    - Shuffle elements.

Imagine an array with a billion elements. All of them need to be copied to the new location. Insert to the front when there are a billion elements means a lot of shuffling.

Maps

Motivation

It would be nice if we could easily and effectively order elements within our data structures.

Introduction

A **Map** is also called an **Associate Array**, **Associative List**, or a **Dictionary**.

The main idea is that it assigns a unique key to every element in the collection. It constructs the keys such that they can be easily and efficiently be compared and ordered. Thus, we can map each key to a data value.

So, we get a collection of **key-value** pairs, where keys are unique and each key maps to (or is associated with) a value (e.g., student information or dictionary).

Map ADT

- The Map ADT specifies the following operations:
    - Insert an entry (key-value pair) into the map.
    - Determine whether a key exists in the map.
    - Get the value associated with an existing key in the map.
    - Remove an entry from the map (specified by its key).
    - Update the value associated with a key.
    - Output all entries in the map.
    - Remove all entries from the map.

Note: We are always using the key for all the operations.

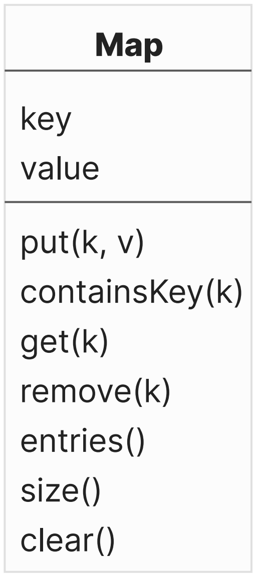
Summary

- Keys must be distinct.
- Each value is "mapped" to a unique key.
- Each element must have both a key and a value.
- We need the key to insert, find, update, and remove elements.

Implementation

We will use Java Generics for flexibility since this allows us to use any type for the key and value.

General Structure: `Map<KeyType, ValueType> myMap;`



Note: Keys are distinct, but values do not have to be unique.

Measuring Height and Depth

- The **height** of node v is the length of the longest path from v to a leaf.
  - The height of a tree is the height of the root.
- The **depth** of node v is the length of the path from the root to v.
  - This is also referred to as the level of the node.

This leads to slightly different definition of the height of a tree: the height of a tree is the number of different levels found in the tree.

Height

The height of a binary tree T is defined as follows:

- If T is a leaf, the height of T is 0.
- Otherwise, the height of T is 1 + the maximum height of T's left and right subtrees.

Multi-Dimensional Arrays

We can store different types of data inside each index in an array. We can also store an array inside each index of an array.

```
int c[][] = new int[3][4]
```

This creates a 3×4 array.

```
+---+---+---+---+
| 0 | 0 | 0 | 0 |
+---+---+---+---+
| 0 | 0 | 0 | 0 |
+---+---+---+---+
| 0 | 0 | 0 | 0 |
+---+---+---+---+
```

Example

```
int c[][] = {{1, 2}, {3, 4, 5}, {6, 7}, {8, 9, 10}};
```

This creates the following array.

```
+----+---+---+----+
| c: | 0 | 1 | 2  |
+----+---+---+----+
| 0  | 1 | 2 |    |
+----+---+---+----+
| 1  | 3 | 4 | 5  |
+----+---+---+----+
| 2  | 6 | 7 |    |
+----+---+---+----+
| 3  | 8 | 9 | 10 |
+----+---+---+----+
```

```
System.out.println(c[1][2]); // Outputs 5
```

Object-Oriented Programming

We have now created our own objects, and more recently, used inheritance to extend our objects into subclasses.

The four principles of object-oriented programming:

1. **Encapsulation**: objects combine data and operations into a single unit.
2. **Abstraction**: unnecessary details are hidden.
3. **Inheritance**: classes can inherit properties from other classes.
4. **Polymorphism**: objects determine appropriate operations at runtime.

Polymorphism

Polymorphism, which literally translates to *many forms*, at its core allows us to perform a single action in different ways.

Something can have many forms in the real world too: One person can be a mother, a partner, an employee, and a teammate. One person might have different behaviours in different scenarios.

We can translate this idea into programming if we imagine that an object might have different behaviours or implementation in different situations.

Note: we can also do this when we implement interfaces and extend classes.

Implementation

Imagine we have implemented a list with an array or linked list. Assume both implementations implement a List interface. We can create a reference variable of type List. `List myList;`.

And use either implementation:

- `myList = new ArrayList();`
- `myList = new LinkedList();`

Both implementations implement the same List interface, so the same operations can be applied to `myList` (add, remove, get, etc). But they are implemented in very different ways.

We can also use polymorphism to produce different behaviours.

Animal `speak()` method:

```
public void speak() {
        System.out.println(
                "I am a " + species + " and I say " + sound
        );
}
```

Dog `speak()` method (which overrides the Animal speak for dogs):

```
public void speak() {
        System.out.println("My name is " + name);
        super.speak();
}
```

Key Takeaways

Polymorphism allows us to use an instance of a class as if it were different types. The method that is invoked is not determined at runtime.

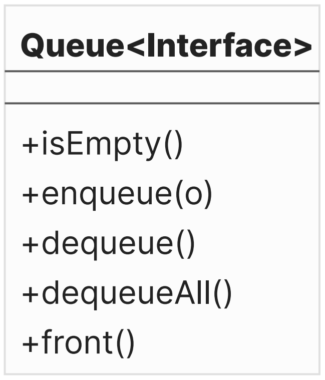
The Notion of a Queue

Collection of items: items are returned in the same order they were added (i.e., FIFO (First In, First Out)). For example, waiting in line.

Queue ADT

The Queue ADT specifies the following operations:

- Create an empty queue.
- Determine whether a queue is empty.
- Add an object to the back of the queue.
- Remove the object from the front of the queue.
- Remove all objects from the queue.
- Access the objects at the front of the queue.



Queue Implementation

A queue can be implemented in multiple ways.

Queue Implementation (Linked List)

In a linked list implementation, a queue can easily be implemented using singly-linked nodes with head and tail references.

Key Observation: `addBack()` and `removeFront()` methods are used in this implementation.

Queue Implementation (Array)

In an array linked list implementation, we typically have the following...

- **data**: an n-element array
- **f**: an integer representing the index of the front element in the queue
- **b**: an integer to keep of the index to insert the next element

Circular Arrays

Problem: ArrayIndexOutOfBoundsException

Solution: Have the **b** variable go back around to the first index.

Key Observation: `addBack()` and `removeFront()` methods are used in this implementation.

Recursion

A divide and conquer approach to solving problems, where the solution depends on solutions to smaller instances of the same problem.

A recursive solution to a problem by using methods that call themselves within their own code. Recursion is one of the central ideas of computer science; we will se a lot of recursion solutions as we explore different algorithms and data structures throughout this course.

Recursion Example

A recursive call is when a method calls itself.

```
// n * (n - 1) * (n - 2) * ... * 3 * 2 * 1
// 4! = 4 * 3 * 2 * 1 = 24
// 0! = 1
public static int factorial(int n) {
        if (n <= 1) {
                return 1;
        } else {
                return n * factorial(n-1)
        }
}
```

Rules of Recursion

Three important properties of a recursive algorithm:

1. The recursive algorithm must call itself (called a recursive call).
2. The recursive algorithm must have a base case.
3. The recursive calls must converge to the base case.

Recursion and Lists

Printing a List - Loop

```
Node cur = head;
while (cur != null) {
        System.out.print(cur.data));
        cur = cur.next;
}
```

Printing a List - Recursion

```
public void printList(Node cur) {
        if (cur == null) {
                return;
        }
        System.out.print(cur.data));
        printList(cur.next);
}
```

Repetition Structures

Repetition structures, or loops, allow us to repeat a block of code.

There are three types of loops in Java:

- while:
  - continue to execute a block of code as long as specified condition is met.
  - useful because sometimes we don't know how long a condition will be true.
- do-while:
  - always execute the block of code once.
  - and continues to execute the block of code as long as a specified condition is met.
- for:
  - repeat the execution of a block of code a specified number of times.

While Loops

```
while(condition) {
    statement;
}
```

Do While Loops

```
do {
    statement;
} while (condition);
```

For Loops

```
for (initialization; condition; increment) {
    statement;
}
```

Initialize: this happens once (usually is simply initializing a variable).

Condition: determines whether or not to execute the block of code.

Increment: this code runs every time after the code block is executed.

Runtime Analysis

Analysis

Question: What is the "right" way to analyze the speed of an algorithm?

- Idea: We could just execute it and see how long it takes to complete.
  - Problem: It is difficult to setup a controlled environment to accurately compare two different algorithms (the hardware environment, the software environment).

Answer: We can simply count the number of operations that need to be executed when running the algorithm.

- Idea: Count the number of operations that occur!
  - Problem: One algorithm might work better than another under different circumstances!

It is best to perform several experiments on many different test inputs and with test inputs of various sizes, and in particular think about how one algorithm scales as our input size increases.

Proposed Methodology: Associate each algorithm a function $f(n)$ that characterizes the running time of the algorithm based on the number of Java statements that would need to be executed in terms of the input size n.

Assumptions

Any single Java statement takes the same amount of time to run. A method call's runtime is measured by the total statements executed inside the method's body. A loop's runtime is N times the runtime of the statements inside the method's body (where N is the number of times the loop repeats).

```java
// Runtime of 3
statement1;
statement2;
statement3;

// Runtime of N
for (int i = 1; i <= N; i++) {
        statement4;
}
```

```
// Runtime of 3N
for (int i = 1; i <= N; i++) {
        statement5;
        statement6;
        statement6;
}

// Runtime of N^{2}
for (int i = 1; i <= N; i++) {
        for (int j = 1; j <= N; j++) {
                statement1;
        }
}
```

Runtime Analysis Chart

Method	**Array**	**Singly Linked**	**Singly Linked**	**Doubly Linked**	**Doubly Linked**
	`numElements`	No Tail	Tail	No Tail	Tail
Object get(int position)	0(1)	0(n)	0(n)	0(n)	0(n)
size()	0(1)	0(1)	0(1)	0(1)	0(1)
int find(Object o)	0(n)	0(n)	0(n)	0(n)	0(n)
toString()	0(n)	0(n)	0(n)	0(n)	0(n)
addAt(Object o, int position)	0(n)	0(n)	0(n)	0(n)	0(n)
addFront(Object o)	0(n)	O(1)	0(1)	0(1)	0(1)
addBack(Object o)	0(n)	0(n)	0(1)	0(n)	0(1)
removeAt(int position)	0(n)	0(n)	0(n)	0(n)	0(n)
removeFront()	0(n)	0(1)	0(1)	0(1)	0(1)
removeBack()	0(1)	0(n)	0(n)	0(n)	0(1)
swapElements (int position1, int position2)	0(1)	0(n)	0(n)	0(n)	0(n)

Runtime Analysis - Stacks and Queues

Method	**Array**	**Array**	**Singly Linked**	**Singly Linked**	**Doubly Linked**	**Doubly Linked**
	Average	Worst	No Tail	Tail	No Tail	Tail
Object get(int position)	0(1)	0(1)	0(n)	0(n)	0(n)	0(n)
size()	0(1)	0(1)	0(1)	0(1)	0(1)	0(1)
int find(Object o)	0(n)	0(n)	0(n)	0(n)	0(n)	0(n)
toString()	0(n)	0(n)	0(n)	0(n)	0(n)	0(n)
addAt(Object o, int position)	0(n)	0(n)	0(n)	0(n)	0(n)	0(n)
addFront(Object o)	0(n)	0(n)	O(1)	0(1)	0(1)	0(1)
addBack(Object o)	0(1)	0(n)	0(n)	0(1)	0(n)	0(1)
removeAt(int position)	0(n)	0(n)	0(n)	0(n)	0(n)	0(n)
removeFront()	0(n)	0(n)	0(1)	0(1)	0(1)	0(1)
removeBack()	0(1)	0(1)	0(n)	0(n)	0(n)	0(1)
swapElements(int position1, int position2)	0(1)	0(1)	0(n)	0(n)	0(n)	0(n)

Stacks - Array

- push: `addBack`
- pop: `removeBack`

Thus, we have $O(1)$ runtime!

Stacks - Linked List

- push: `addFront`
- pop: `removeFront`

Thus, we have $O(1)$ runtime!

Queues - Array

- push: `addBack`
- pop: `removeFront`

Thus, we have $O(n)$ runtime!

But, we can use a circular array!

Thus, we have $O(1)$ runtime!

Queue - Linked List

- push: `addFront`
- pop: `removeFront`

Thus, we have $O(1)$ runtime!

Setters and Getters

Access modifiers and accessors/mutators.

Access Modifiers

The public keyword is an access modifier.

Access Modifiers:

- public: allows accessibility by any other class
- private: only accessible within the declared class
- protected: only accessible by classes within the same package, or by subclasses

Note: we can set the access modifiers for all of our fields to private. However, we will need another way to work with these fields!

```java
public class Student {
    // Fields
    private String sID;
    private String name;
    private double gpa;
    private String program;
}
```

Accessors and Mutators Methods

They are sometimes called "getters" and "setters". Typically, many fields within a class are not publicly accessible.

Instead, we will define methods that allow us to access (get) or mutate (set) the values of fields indirectly.

```java
public String getProgram() {
    return program;
}

public void setProgram(String newProgram) {
    program = newProgram;
}
```

Note: These methods are public so can be accessed by other classes.

These methods are defined within the Student class, so they can access private fields.

Errors

If we try to access a private field directly, we will get the following error...

```
System.out.println(s1.sID);

error: sID has private access in Student
```

However, we can access the field indirectly by calling the get method for that field.

```
System.out.println(s1.getsID());
```

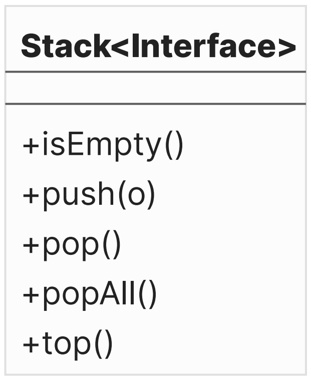
The Notion of Stack

Collection of items: items are returned in the reverse order they were added (i.e., LIFO (Last In, First Out)). For example, "undo" function and back button, nested method calls and recursion.

Stack ADT

The Stack ADT specifies the following operations:

- Create an empty stack.
- Determine whether a stack is empty.
- Insert an object onto the stack.
- Remove the most recently added item from the stack.
- Remove all items from the stack.
- Access the most recently added item from the stack.



Stack Implementation

A stack can be implemented in multiple ways.

Stack Implementation (Array)

In an array implementation, we typically have the following...

- **data**: an n-element array
- **top**: an integer to keep track of the index to insert the next element

Key Observation: `addBack()` and `removeBack()` methods are used in this implementation.

Stack Implementation (Linked List)

In a linked list implementation, a stack can easily be implemented using singly-linked nodes and a head reference (tail isn't necessary!)

Key Observation: `addFront()` and `removeFront()` methods are used in this implementation.

Stacks and Queues

Why?

- Runtime efficiency: All operations run in $O(1)$ (scale well as input sizes grow).
- Simplicity: Simple implementation (no loops (i.e., if you have a loop you aren't correctly implementing stacks and queues)). Easy to use (small number of straightforward operations).

Stack Implementation

Array

Recall from our array implementation of a list:

- Data: Array that holds all of the elements.
- Top: Keep track of the number of elements and also the first empty location.

How does this relate to the implementation of a stack? We can use the fields the same way -- data to hold the elements, top to keep track of the "top" of the stack.

- push: `addBack` (assuming room to insert)

```
date[top] = element;
top++;
```

- pop: `removeBack` (assuming there is an element to remove)

```
top--;
int removed = data[top];
return removed;
```

Linked List

Implement with a head reference (no tail), singly linked list.

- push: `addFront`
- pop: `removeFront`

Queue Implementation

Linked List

Will have head and tail references (still only singly linked).

- enqueue: `addBack` (tail reference allows us to avoid loops)
- dequeue: `removeFront`

Array

Remember: No loops! All operations should run in $O(1)$. In order to add one side and remove from the other, we are stuck with an $O(n)$ operation because of shuffling.

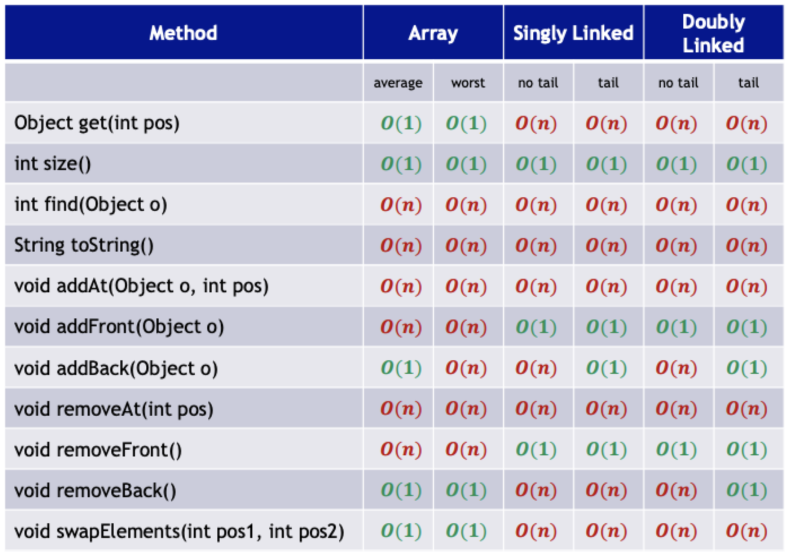
- enqueue: `addBack` (same as before -- fast)
- dequeue: `removeFront` (no need to shuffle to index 0)

Idea: We used numElements to keep track of the first open location.

Two Variables:

- back: keep track of first open location
- front: keep track of first used location

Stacks and Queues Runtime Analysis



Method	Array		Singly Linked		Doubly Linked	
	average	worst	no tail	tail	no tail	tail
Object get(int pos)	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
int size()	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
int find(Object o)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
String toString()	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
void addAt(Object o, int pos)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
void addFront(Object o)	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
void addBack(Object o)	$O(1)$	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(1)$
void removeAt(int pos)	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
void removeFront()	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
void removeBack()	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(1)$
void swapElements(int pos1, int pos2)	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$

Static VS Non-Static

A **static** method means that it can be accessed without creating an object of the class.

Typically for generic methods that operate on data passed in as a parameter. For example, the methods in our ArrayOperations.java file.

Note: They cannot access instance variables within a class.

An **instance** method is defined within a class.

Instance methods are called on instance of class. Operate on the object's instance variables (fields).

Static Methods

A static method operates on the parameters it is passed.

- Utility Methods: Operations that work on the array passed in as a parameter. Operations you might find on a calculator.

These aren't associated with instances of an object (or that object's fields).

Instance Methods

An instance method operates on an object and the object's fields.

- Student Class: Update information about the student.

Static Variables

A static variable is not associated with each specific instance of a class.

Instance Variables

Each object has its own values for each of it's instance variables (fields).

Example

```java
public class BankAccount {
    String id;
    double savings;
    static double interestRate = 0.05;
}
```

Each BankAccount object will have it's own id and amount of money in their savings account.

There is only one interestRate that is shared by all BankAccounts.

Linked List: Tail Reference

Adding an Item to the Front of a List

- First: Determine where to insert the new node.
- Second: Update the next pointers appropriately.

```
Node n = new Node(12, null);

n.next = head;
head = n;

// A constructor
Node n = new Node(12, head);

head = n;
```

Adding an Item to the Back of a List

- First: Determine where to insert the new node.
- Second: Update the next pointers appropriately.

```
Node n = new Node(30, null);

cur = head;
// Inefficient
while(cur.next != null) {
        cur = cur.next;
}

cur.next = n;
```

Tail Reference

Idea: We have a reference to the front (head) of our list, so why don't we do the same with the back (tail).

```
Node n = new Node(30, null);

tail.next = n;
tail = n;
```

Linked List Variations

Adding a tail reference requires very little overhead, and makes inserting/removing from the back of the list much more efficient.

```java
public class IntegerLinkedList implements IntegerList {
        private int numElements;
        private Node head;
        private Node tail;

        public IntegerLinkedList() {
                head = null;
                tail = null;
                numElements = 0;
        }
}
```

The `toString` Method

The `toString` method returns a String representation of an object.

Note: it is up to the programmer to decide what information is present in String representation of the object.

This method is automatically called when we print out an object.

Example

The following code snippet will output `Ali Sa - V00123456` when calling `System.out.println(s1);`.

```java
public class Student {
    // Fields
    private String sID;
    private String name;
    private double gpa;
    private String program;

    public String toString() {
        String s = name + " - " + sID;
        return s;
    }
}
```

The `this` Keyword

We use `this` to refer to something that is part of the class.

We often use the `this` keyword in our constructors.

```
public Student(
    String sID,
    String name,
    double gpa,
    String program) {
        this.sID = sID;
        this.name = name;
        this.gpa = gpa;
        this.program = program;
}
```

Note: we use `this` to refer to the instance variables (fields). In the constructor, we want to initialize these with values passed in as parameters.

Example

The following could be rewritten to use `this` keyword.

```
public void setProgram(String newProgram) {
    program = newProgram;
}
```

Rewritten:

```
public void setProgram(String program) {
    this.program = program;
}
```

Note: the `this.program` refers to the class instance variable (field) and the `program` refers to the parameter.

Tree Classifications

There are a few tree classifications: Perfect, Complete, and Balanced.

Binary Tree Classifications (Perfect)

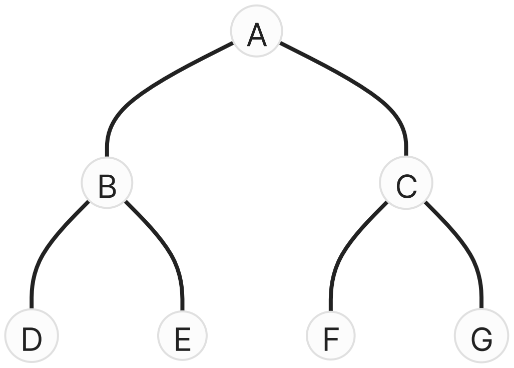
A binary tree is perfect if:

- No node has only one child.
- All leaf nodes have the same depth.

A perfect binary tree of height h has:

- $2^{h+1} - 1$ total nodes, of which 2^h are leaves.
- Solving for h, we get $h \leq \log_2(n+1) - 1$.
- Height is bound by $\log_2(n)$.

Example

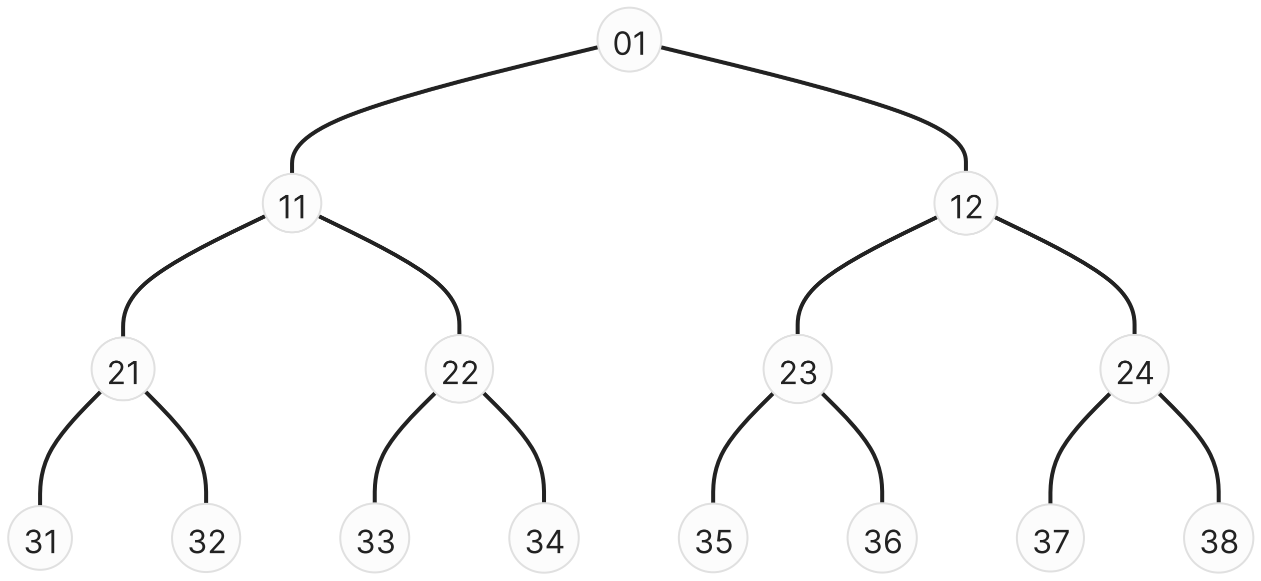


In a perfect binary tree, each level doubles the number of nodes:

- Level 1 has 2 nodes (2^1).
- Level 2 has 4 nodes (2^2), or 2 times the number of nodes from Level 1.
- Level 3 has 8 nodes (2^3), or 2 times the number of nodes from Level 2.

Therefore a tree with h levels has $2^{h+1} - 1$ nodes: with 2^h nodes on Level h.

Example



Level	Nodes	Total Nodes
1	2	3 $(2^2 - 1)$
2	4	7 $(2^3 - 1)$
3	8	15 $(2^4 - 1)$

Binary Tree Classifications (Complete)

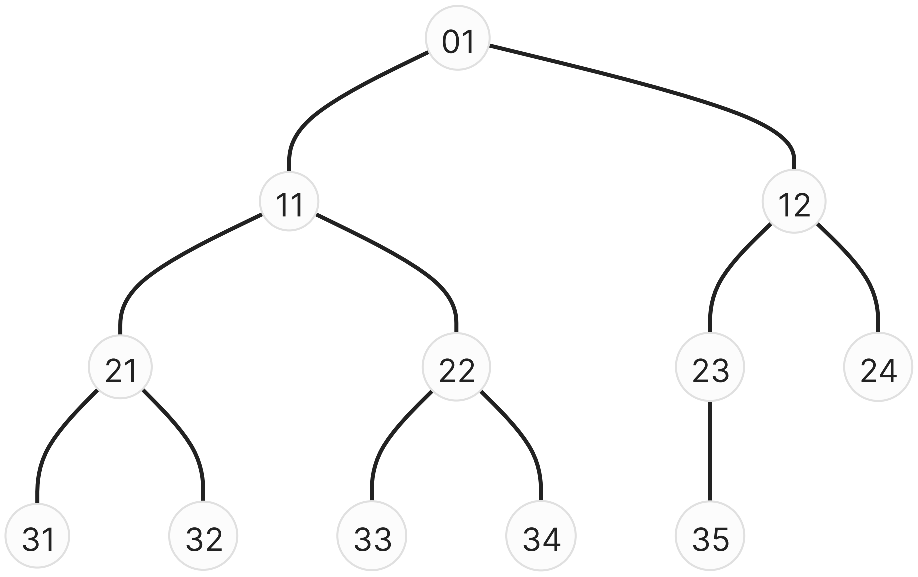
A binary tree is complete if:

- The leaves are found on at most 2 levels.
- The second to bottom level is completely filled.
- The leaves on the bottom level are as far left as possible.

Missing nodes (from a perfect tree):

- are on the bottom level.
- on the right-most side.

Example

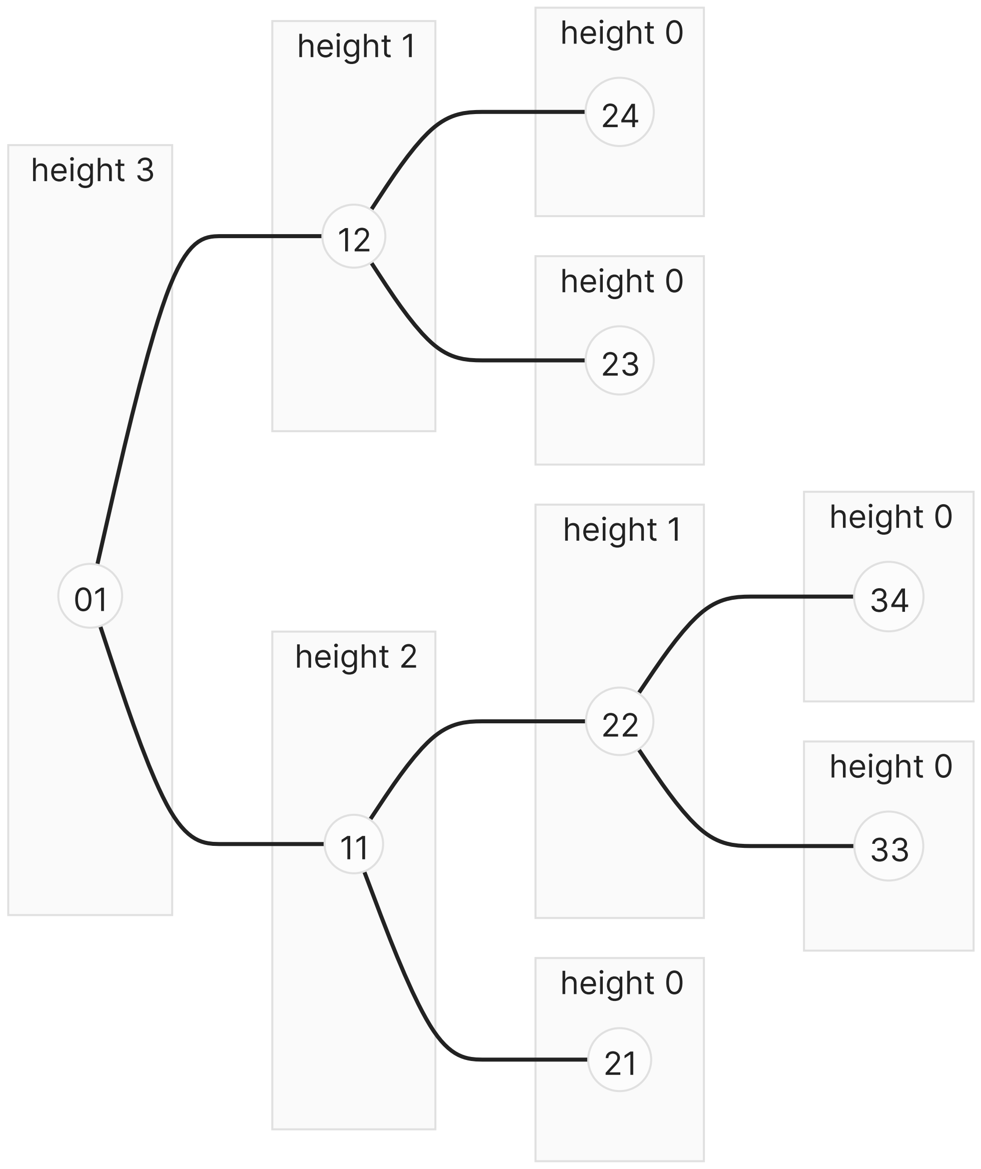


Binary Tree Classifications (Balanced)

A binary tree is balanced if:

- For any node v in the tree, the height of v's right subtree is at most one different from the height of its left subtree.

Example



Importance of Tree Classifications

The worst-case access time for any node is bound by the tree's height. The height of a balanced tree is bound by $O(\log(n))$.

To reach any node from the root, the longest the path will be is bound by the height. So, a tree of height h containing 2^{h+1} nodes means that we can reach a lot of nodes very quickly.

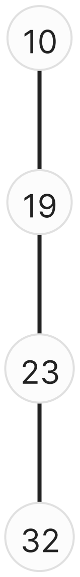
Example: Let us have a binary tree with 15 elements, in the worst case we had to visit 4 elements. (Assume we have a balanced Binary Tree containing a larger number of elements).

Number of Elements	Number of Elements Searched
1000	at most 10

Number of Elements	Number of Elements Searched
1000000	at most 20
1000000000	at most 30

Thus, we can search through certain Binary Trees very quickly!

Example Let us have a binary tree with 4 elements, in the worst case we had to visit 4 elements.



Note: We have to traverse the entire path in the worst case and we do not cut our search elements.

Asymptotic Analysis

In general, the algorithms we will see in this course will likely fall under one of the following categories.

Example: Linear is $O(n)$ and Logarithmic is $O(\log(n))$.

Summary

- Perfect Tree: All nodes on all levels are filled (both complete and balanced)
- Complete Tree: All nodes except those on last level are filled (any missing are on the right)
- Balanced Tree: No nodes left and right subtree heights differ by more than 1.

Tree Structure

Trees consist of nodes chained together: trees start at the top (called the root) and each node links to other nodes lower in the tree.

Tree Visualization

Often trees are represented visually as vertices and edges:

- Each vertex is drawn as a circle, and represents the data component of a node.
- The edges represent the references to other nodes within the data structure.

Tree Application

We can use trees for decision making. They will also help us perform operations on a list of elements efficiently.

Tree Relationships

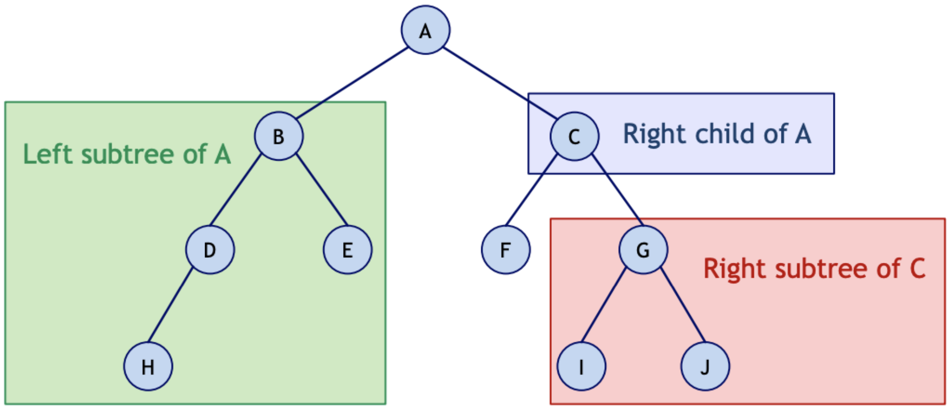
General Terminology of the Relationship between Nodes in a Tree

- Node v is said to be a **child** of u, and u the **parent** of v if:
  - there is an edge connecting u and v.
  - u is above v in the tree.
- Generalization: - Descendants of u. - Ancestors of v. - Siblings of v.

Tree Terminology Reference

- **root**: the single node with no parent
- **leaf**: a node with no children
- **child**: a node pointed to by me
- **parent**: the node that points to me
- **sibling**: another child of my parent
- **ancestor**: my parent or my parent's parent
- **descendant**: my child or my child's descendant
- **subtree**: a node and its descendants
- **path**: is a sequence of nodes $v_1, v_2, \cdots, v_n$ where v_i is a parent of v_{i+1}.
- **binary tree**: is a tree with at most two children per node: left and right.

Tree Example


A
B
C
Right child of A
Left subtree of A
D
E
F
G
Right subtree of C
H
I
J

Using A Stack to Solve A Problem

Assume we now have a working implementation of a Stack or Queue. What are the benefits of using an ADT like a stack to solve a problem? The operations are clearly defined and easy to understand. It can be reused to solve many different problems. The implementation can be changed without changing the behaviour of the program that uses the ADT.

One error you may have encountered programming in Java is when your open and close curly braces ({ and }) don't line up. Sometimes, we forget a closing curly brace. Or, for some reason, there is an extra closing brace.

How could a compiler determine when there is an error with opening and closing braces in a program?

There are a few things that must be true:

- There must be the same total number of {'s and }'s.
- There should never be a point where we have seen more }'s than {'s.

We can determine if either of the two violations above occur using a stack!

Brace Matching

Read through the contents of the `.java` file. Every time we see an open brace ({), push it to the stack. Every time we see a closing brace (}), pop from the stack.

At the end of the code, the stack should be empty.

But how do we determine if at some point there were more close braces than open braces? Yes, if we try and pop from an empty stack!