Features of data structures

- Linked list
  - Simple to implement
  - Lots of flexibility in deciding the order of items
    - The burden is on you to maintain that order
  - Search time not too critical
- Binary search tree
  - Data has some order to it
  - Want efficient searches
  - Ok with one of
    - Complex code to keep the tree balanced
    - Approximations to a balanced tree are ok
    - You're expecting the order of added data will keep the tree semi-balanced
- Heap
  - The data is ordered and you only ever want the biggest/smallest item
- Graph
  - You are representing items _and_ connections or relations between items

Spreadsheet problem

- A spreadsheet is a 2-dimensional grid of cells. Each cell is either blank, contains a number, contains a string, or contains a formula. A formula can include references to other cells of the spreadsheet. When a cell A contains a reference to another cell B, we must ensure that cell B's value is calculated before trying to calculate the value of cell A.

  Describe an efficient data structure to store the cells of a spreadsheet. The data structure will primarily be used to recalculate the value of each affected cell (and all other cells that rely on its value) when one cell value changes.

Data structure performance comparison

Common Data Structure Operations

Data Structure	Time Complexity								Space Complexity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Array	Θ(1)	Θ(n)	Θ(n)	Θ(n)	O(1)	O(n)	O(n)	O(n)	O(n)
Stack	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Queue	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Singly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Doubly-Linked List	Θ(n)	Θ(n)	Θ(1)	Θ(1)	O(n)	O(n)	O(1)	O(1)	O(n)
Skip List	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n log(n))
Hash Table	N/A	Θ(1)	Θ(1)	Θ(1)	N/A	O(n)	O(n)	O(n)	O(n)
Binary Search Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)
Cartesian Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(n)	O(n)	O(n)	O(n)
B-Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Red-Black Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
Splay Tree	N/A	Θ(log(n))	Θ(log(n))	Θ(log(n))	N/A	O(log(n))	O(log(n))	O(log(n))	O(n)
AVL Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(log(n))	O(log(n))	O(log(n))	O(log(n))	O(n)
KD Tree	Θ(log(n))	Θ(log(n))	Θ(log(n))	Θ(log(n))	O(n)	O(n)	O(n)	O(n)	O(n)

Balanced Binary Tree

Image from https://medium.com/omarelgabrys-blog/data-structures-a-quick-comparison-6689d725b3b0

How do you choose a data structure?

- Look at the operations that are needed
- Look at the set of constraints or assumptions allowed
- Look at the context of use
  - Are there any characteristics to the data to be stored?
  - Which operations are more important?
  - How much data will you store?
  - How do you define “best”?
- Look at the short-term and long-term effort needed to create it and to maintain it
- Look at what else is available in the libraries, program, or experience set

Definitions of “best”

- Incomplete list:
  - Fastest / most efficient
  - Least memory
  - Ease of understanding
  - Ease of maintaining
  - Ease of implementing
  - Scalability
  - Parallelizability
  - Portability
  - Serializability
  - Flexibility
  - …

Java classes

- Concrete class
- Abstract class
- Interface

- When do you use each one?

Concrete class

- Describes the complete implementation of a class
  - Essentially describes a data structure or organization of data
- Can be instantiated
- Can implement multiple interface classes

Concrete class syntax

```
public class point {
        private float x, y;

        static void add (Point p) {
                x += p.x;
                y += p.y;
        }

        static void print ( ) {
                System.out.println( “Point at ” + x + “, “ + y);
        }
}
```

The issue with concrete classes

- They connect the “how” of the implementation with the “what” of the class
  - Don’t allow you to code other parts of the program with the idea of an abstract data type

Interface

- Describes the method interfaces and constants of the class
  - Essentially describes an abstract data type
    - Could still be type-specific
  - Cannot include variables
- Cannot be instantiated
- Can only extend other interfaces

Interface syntax

```
public interface int_queue {
        public void add( int value );
        public int remove ( );
}
```

Cannot say : int_queue varName = new int_queue();

Can say: int_queue varName;
varName = new int_queue_implementation_class()

Abstract Class

- Describes methods, method interfaces, constants, and variables of the class
  - More concrete than an abstract data type
  - Still leaves some implementation decisions to be decided
- Cannot be instantiated
- Can extend concrete classes and abstract classes, and can implement interface classes

Abstract class syntax

```
public abstract class tournament {
        private String tournament_name;
        public abstract void add_team ( String team_name);
        public abstract int number_of_teams ( );
        public void set_tournament_name ( String name ) {
                tournament_name = name;
        }
        public void print_signup_sheet ( ) {
                int i;
                System.out.println( "Tournament sign-up: " + tournament_name);
                for (i = 0; i < number_of_teams( ); i++) {
                        System.out.println( i + ". _____");
                }
        }
}
```

Example

- Stack implemented using an array
  - Concrete class
- Stack definition of push(), pop(), and size() methods
  - Interface
- Stack definition of push(), pop(), size(), and print() methods with an implementation of print() that just uses push(), pop(), and size()
  - Abstract class