Workshop #1 Competitive Programming: Data Structures and Libraries

Presentation: Imad Dodin
Slides: Muhammad Huzaifa Elahi

>Compete McGill_

What is a Data Structure?

- A means of **storing** and **organizing** data
- Different data structures come with different **pros** and **cons**
- **Horses for courses:** Pick the data structure that suits your need!
- Theory vs Implementation?
    - Understand the theory behind the data structure
    - Utilize existing libraries to quickly implement the data structure

Linear Data Structures

- Static Array
  - What?
  - When?
  - Size?.
  - Dimensions?
  - Typical array operations:
    - Access
    - Sort
    - Scan / Search
  - Problems?

Linear Data Structures

- Dynamically Resizable Arrays
  - Dynamic!
  - Size??? → Dynamic Array
  - In Java: `ArrayList / Vector - java.util`
  - `add(Object o)`
  - `addAll(Collection C)`
  - `add(int index, Object o)`
  - `addAll(int index, Collection C)`
  - `remove(Object o)`
  - `remove(int index)`
  - `removeAll(Collection c)`
  - `get(int index)`
  - `contains(Object o)`
  - `size()`
  - `isEmpty()`
  - `indexOf(Object O)`
  - `lastIndexOf(Object O)`

- `ArrayList()`
- `ArrayList(Collection c)`
- `ArrayList(int capacity)`

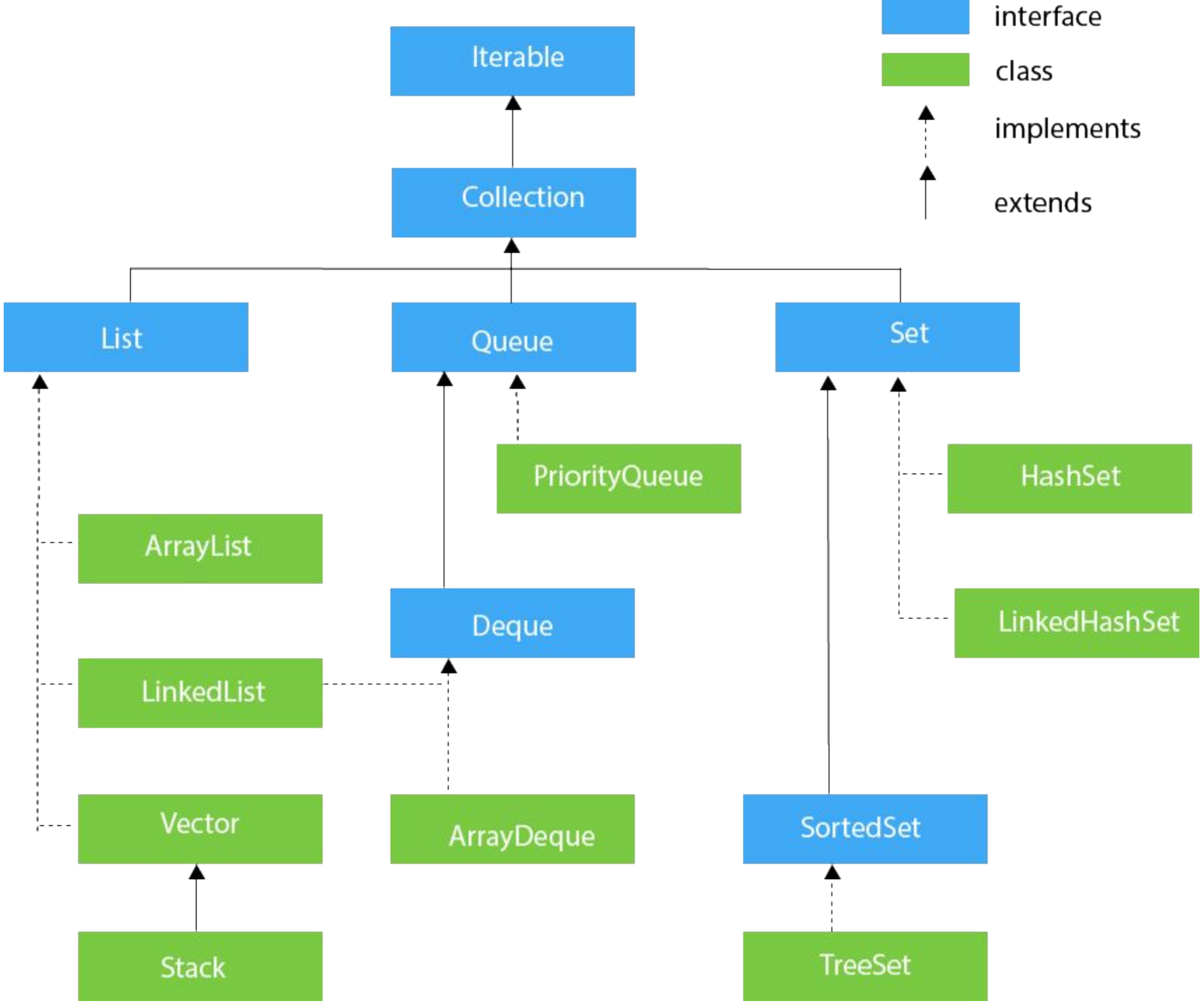
Linear Data Structures

```
// Java program to demonstrate working of
ArrayList in Java
import java.io.*;
import java.util.*;

class arrayli
{
    public static void main(String[] args)
                        throws IOException
    {
        // size of ArrayList
        int n = 5;

        //declaring ArrayList with initial size n
        ArrayList<Integer> arrli = new
ArrayList<Integer>(n);

        // Appending the new element at the end
of the list
        for (int i=1; i<=n; i++)
            arrli.add(i);

        // Printing elements
        System.out.println(arrli);

        // Remove element at index 3
        arrli.remove(3);

        // Displaying ArrayList after deletion
        System.out.println(arrli);

        // Printing elements one by one
        for (int i=0; i<arrli.size(); i++)
            System.out.print(arrli.get(i)+" ");
    }
}
```

ArrayLists

Runtimes:

add(Object o)	O(1)
remove(Object o)	O(n)
remove(int index)	O(1)
contains(Object o)	O(n)



>Compete McGill_

Array Operations: Sorting & Searching (1)

- **Sorting Algorithms:**
  - **O(n^2) comparison-based sorting algorithms:**
    - Bubble/Selection/Insertion Sort, etc
    - These algorithms are **slow** (avoid) but understanding them is useful for certain problems
  - **O(n log n) comparison-based sorting algorithms:**
    - Merge/Heap/Quick Sort, etc
    - These algorithms are fast: O(n log n) is **optimal** for comparison-based sorting
  - **O(n) Special purpose sorting algorithms: Counting/Radix/Bucket Sort, etc.**
    - Rarely used but good to know as it can be more efficient if data has certain characteristics.
    - E.g: Counting Sort can be applied to integer data that lies in a small range
    - Covered *in later workshops.*

Array Operations: **Sorting** & Searching (1)

- Java Collections.sort(List<T> list)
- `java.util`
- So what is the runtime?
- Collections.sort(List<T> list, Comparator<T> c);
- Collections.reverse(List<T> list)

Array Operations: Sorting & **Searching (2)**

- **Search Methods:**
  - **O(n) Linear Search:**
    - Consider every element from index 0 to index n – 1 (**avoid this whenever possible**)
  - **O(log n) Binary Search:**
    - Input **MUST** be sorted: if unsorted: use a O(n log n) sorting algorithm to sort before search
  - **O(1) with Hashing:**
    - Useful technique when fast access to known values (indices) are required
    - If hash function is good, the probability of a collision is made negligibly small

Linked Lists & Stack

- Linked List: Java LinkedList
    - Linked List is usually avoided in contest:
        - Inefficient for accessing elements: A linear scan needed from the head or the tail of a list
        - The usage of pointers makes it prone to runtime error
    - ArrayList is more flexible for competitions (but LinkedList is still used for Queues)
- Stack: Java Stack
    - A stack only allows for $O(1)$ insertion (push) and $O(1)$ deletion (pop) from the top.
    - This behavior is called Last In First Out (LIFO) just like an actual stack
    - Operations:
        - push()/pop() (insert/remove from top of stack)
        - top() (obtain content from the top of stack)
        - empty()

Queues

- Queue: Java LinkedList

```
Queue<String> q = new LinkedList<String>();
q.add(“Imad”);
q.add(“Huzaifa”);
q.add(“Andre”);
String first = q.remove();
String second = q.poll();
```

Queues

- Queue: Java LinkedList

```
Queue<String> q = new LinkedList<String>();
q.add(“Imad”);
q.add(“Huzaifa”);
q.add(“Andre”);
String first = q.element();
String second = q.peek();

q.isEmpty();
q.size();
```

Queues

Runtimes:

add(Object o)	O(1)
element(Object o) / peek(Object o)	O(1)
remove(Object o) / poll(Object o)	O(1)
size()	O(1)
isEmpty()	O(1)

Non-Linear Data Structures

- Sometimes: linear storage is not optimal for organizing data.
- For example:
  - If you need a dynamic collection of pairs (e.g. key → value pairs)
  - Using HashMap<E, E> gives O(1) insertion and access.

Balanced Binary Search Tree

- Balanced Binary Search Tree (BST):
  - The BST is one way to organize data in a tree structure.
  - In each subtree rooted at x, the following BST property holds **(properties for being balanced)**:
    - Items on the left subtree of x are smaller than x
    - items on the right subtree of x are greater than (or equal to) x.
  - **O(log n)** search(key), insert(key), findMin()/findMax(), successor(key)/predecessor(key), delete(key)

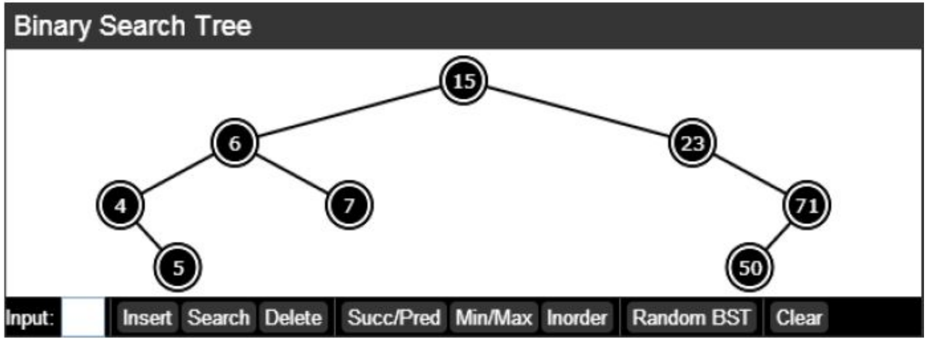


Figure 2.2: Examples of BST

Heap

- Heap: Java PriorityQueue
    - The (Binary) Heap is also a binary tree like the BST, except that it must be a **complete tree**.
    - Complete binary trees can be stored in an array of size n + 1. Can navigate from a certain index i to its:
        - parent : $\lfloor i/2 \rfloor$
        - left child : $2 \times i$
        - right child : $(2 \times i) + 1$
    - The (Max) Heap enforces the Heap property:
        - In each subtree rooted at x, items on the left and right subtrees of x are smaller than (or equal to) x
        - The property guarantees that root of heap is always the max element
    - The Heap has fast deletion of max element: poll() and insertion of new items: add(Element e) - both of which are **O(log n)** and perform swapping operations to maintain the (Max) Heap property afterwards

PriorityQueue

- Java PriorityQueue
- `java.util`

```
PriorityQueue();
PriorityQueue(Collection c);
PriorityQueue(int initialCap);
PriorityQueue(int initialCap, Comparator c);
PriorityQueue(PriorityQueue c);
```

PriorityQueue

```
PriorityQueue<String> p = new PriorityQueue<String>();
p.add("Boustan");
p.add("Chef");
p.add("Mom's spaghetti");
p.poll();
p.remove("Chef");
p.peek();
p.contains("Mom's spaghetti");
p.isEmpty();
p.size();
```

PriorityQueue

Runtimes:

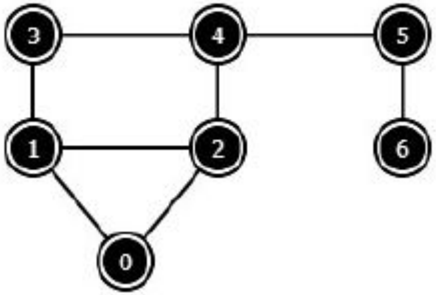
add(Object o)	O(log n)
peek()	O(1)
poll()	O(log n)
remove(Object o)	O(n)
size()	O(1)
isEmpty()	O(1)

Data Structures w/o Built-In Support

- Many important data structures do not have built-in support.
- Thus, important to prepare bug-free implementations of these data structures
- These include:
  - 1. Graphs
  - 2. Disjoint Sets

Graphs

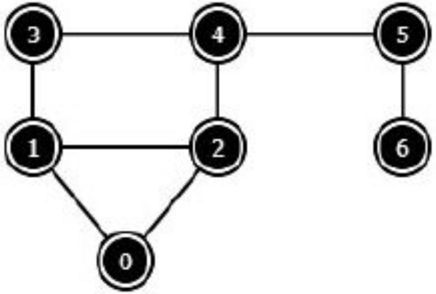
- A graph (G = (V,E)) is a set of vertices (V) and edges (E)
- Weighted Graph: Edges in E contain information (weight) b/w vertices
- Unweighted Graph: Edges in E contain no information b/w vertices except which 2 vertices are connected
- Three basic ways to represent a graph G with V vertices and E edges:
    a. Adjacency Matrix
    b. Adjacency List
    c. Edge List



Graph Data Structure

Graphs: Adjacency Matrix

- In graph problems, when the number of vertices V is usually known
  - Thus we can build a ‘connectivity table’ using a static 2D array: int AdjMat[V][V]
  - This has an $O(V^2)$ space complexity
  - For unweighted graph: set AdjMat[i][j] = 1 if there is an edge between vertex i-j or 0 otherwise
  - For weighted graph, set AdjMat[i][j] = weight(i,j) if there is edge b/w vertex i-j or 0 otherwise
  - For a graph without self-loops: AdjMat[i][i] = 0, $\forall i \in [0..V-1]$
  - <u>**When to use?**</u> if the **connectivity** between two vertices in a **small dense graph** is **frequently required.**
  - <u>**When to avoid?**</u> If **graph is large, sparse** as it would require **too much space** ($O(V^2)$)
  - Another drawback: takes O(V) time to enumerate the list of neighbors of a vertex v

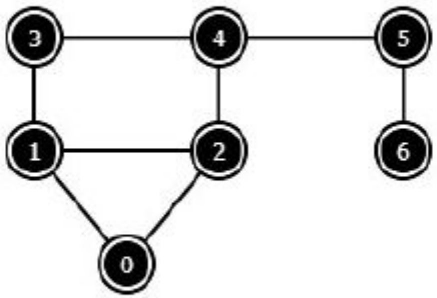


Adjacency Matrix

	0	1	2	3	4	5	6
0	0	2	5	0	0	0	0
1	2	0	7	1	0	0	0
2	5	7	0	0	4	0	0
3	0	1	0	0	3	0	0
4	0	0	4	3	0	9	0
5	0	0	0	0	9	0	8
6	0	0	0	0	0	8	0

Graphs: Adjacency List

- The Adjacency List: a vector of vector of pairs
- Using Java HashMap<Node, LinkedList<Node>>
  - Stores the list of neighbors of each vertex u as ‘edge information’ pairs
  - Each pair contains two pieces of information:
    1. The index of the neighbouring vertex
    2. The weight of the edge.
  - If graph is unweighted, store the weight as 0, 1, or drop the weight attribute
  - The space complexity of Adjacency List is O(V + E) : **more space-efficient** than Adjacency Matrices
  - Can also enumerate the list of neighbors of a vertex v efficiently: **O(k)** where v has k neighbors

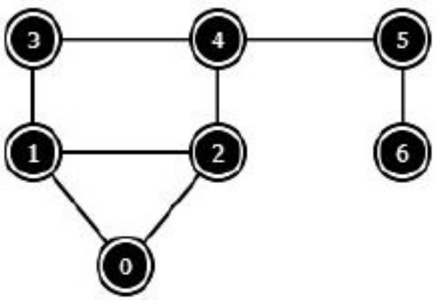


Adjacency List

0: 1 2
1: 0 2 3
2: 0 1 4
3: 1 4
4: 2 3 5
5: 4 6
6: 5

Graphs: Edge List

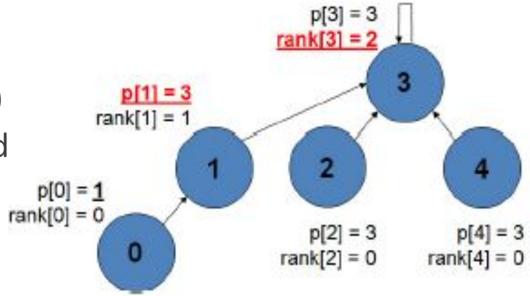
- The Edge List: a vector of triples
  - Normally using Java HashMap<Node, LinkedList<Edge>>
  - We store a list of all E edges, usually in sorted order.
  - For directed graphs, we can store a bidirectional edge twice, one for each direction.
  - The space complexity is O(E)
  - Edge List not best for graph algorithms that require the enumeration of edges incident to a vertex.



Edge List

```
0:  0 1
1:  0 2
2:  1 0
3:  1 2
4:  1 3
5:  2 0
6:  2 1
7:  2 4
8:  3 1
9:  3 4
10: 4 2
11: 4 3
```

Disjoint Sets



- A data structure to model a collection of nonoverlapping sets (collection of items)
- When is it helpful? Questions requiring structures to be merged if not yet merged
- **O(1)** to
  - find which set an item belongs to
  - test whether two items belong to the same set
  - unite two disjoint sets into one larger set.
- Each set has a representative 'parent' item that all other members of the set point to
- We use disjoint sets as a tree structure where the sets form a forest of trees, each tree being a disjoint set.
- The root of the tree is the representative item for a set.
- To find the representative set identifier: follow the chain of parents to the root of the tree