CSCI 241

Lecture 13
Interface vs Implementation
Set, Priority Queue
Intro to Heaps
Announcements

• Quiz 3 is graded, available on Gradescope. Grades are also in Canvas.

• Dr. Hearne is putting together teams for the ACM programming contest coming up on November 11th
  • He is hoping to form multiple teams, including some consisting of lower division students (you!)
  • Email or talk to Dr. Hearne if you are interested and/or want to learn more.
Goals

• Understand how public and private members in Java reflect the separation between interface and implementation.

• Understand the purpose and interface of the Set ADT, and how AVL trees implement it.

• Understand the purpose and interface of the Priority Queue ADT.

• Know the definition and properties of a heap.
A2: Updating Heights

1. **Ignore** height field until implementing rebalance.

2. Heights change when the tree structure changes: insertions and rotations.


4. Strategy - rotations: only x and y can change height. Update them after rotation is complete.
AVL Insertion

\[ \text{insert} \left( \text{Node } n, \text{ int } v \right): \]

// ...(other case, irrelevant here)
else:  // v > n.value
    if n has right:
        \text{insert} \left( n.\text{right}, v \right)
    else:
        // attach new node w/ value
        // v to n.right
        \text{rebalance} \left( n \right);

\text{insert}(a, 16)
=> \text{insert}(c, 16)
  => \text{insert}(f, 16)
    => \text{attach new node}
    rebalance(f)
    rebalance(c)
    rebalance(a)
AVL Insertion

```
insert(Node n, int v):
    //...(other case, irrelevant here)
else: // v > n.value
    if n has right:
        insert(n.right, v)
    else:
        // attach new node w/ value
        //   v to n.right
        rebalance(n);
```

```
insert(a, 16)
=>insert(c, 16)
=>insert(f, 16)
=>attach new node
    rebalance(f)
    rebalance(c)
    rebalance(a)
```
AVL Insertion

```
insert(Node n, int v):
// ...(other case, irrelevant here)
else: // v > n.value
    if n has right:
        insert(n.right, v)
    else:
        // attach new node w/ value
        // v to n.right
        rebalance(n);
```

- `insert(a, 16) => insert(c, 16) => insert(f, 16)`
- `attach new node w/ value v to n.right`
- `rebalance(f)`
- `rebalance(c)`
- `rebalance(a)`
- `update f's height`
AVL Insertion

```
insert(Node n, int v):
  //...(other case, irrelevant here)
else: // v > n.value
  if n has right:
    insert(n.right, v)
  else:
    // attach new node w/ value
    //  v to n.right
    rebalance(n);
```

insert(a, 16)
=>insert(c, 16)
  =>insert(f, 16)
    =>attach new node
    rebalance(f)
  rebalance(c)
rebalance(a)
AVL Insertion

```
insert(Node n, int v):
    // ...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            //   v to n.right
            rebalance(n);
```

insert(a, 16)
=> insert(c, 16)
=> insert(f, 16)
=> attach new node
    rebalance(f)
rebalance(c)
(reotation happens)
rebalance(a)
AVL Insertion

```python
insert(Node n, int v):
    //...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            // v to n.right
            rebalance(n);
```

- `insert(a, 16)`
  - `=>insert(c, 16)`
    - `=>insert(f, 16)`
      - `=>attach new node`
        - `rebalance(f)`
      - `rebalance(c)`
    - `rebalance(a)`

- Update heights:
  - Update f’s height
  - Update c’s height
  - Update a’s height
AVL Insertion

```
insert(Node n, int v):
    //...(other case, irrelevant here)
    else: // v > n.value
        if n has right:
            insert(n.right, v)
        else:
            // attach new node w/ value
            // v to n.right
            rebalance(n);
```

Why shouldn’t we use the recursive height() method from lab3 to update heights?

```
insert(a, 16)
=>insert(c, 16)
    =>insert(f, 16)
    =>attach new node
    rebalance(f)
    rebalance(c)
    rebalance(a)
```

update f’s height
update c’s height
update a’s height
A2: Updating Heights

1. **Ignore** height field until implementing rebalance.

2. Heights change when the tree structure changes: insertions and rotations.


4. Strategy - rotations: only x and y can change height. Update them after rotation is complete.
Recap: **Interface vs Implementation**

- Abstract data structures are **interfaces**
  - they specify only the **interface** to a class (method names and specifications)
  - not its **implementation** (method bodies, fields, …)

- Abstract data structures can have multiple possible **implementations**.
Interface vs Implementation: in Java

- A **class** is a “blueprint” for an **object**.
- It contains **members** including
  - **fields** (variables)
  - **methods** (functions)
- Fields and methods can be **public** or **private**
  - **Private** members **can’t** be seen outside the class.
  - **Public** members **can** be seen outside the class.
- Public members provide the class’s **interface**.
- Private members are used to **implement** the interface.
Interface vs Implementation: in Java

- Public members provide the class’s interface.
- Private members are used to implement the interface.

Important consequence:

- If a public method has a specification, it has to implement that specification precisely and completely.
- Just because you never encounter an edge case doesn’t mean someone else using your class won’t.
Interface vs Implementation: in Java

- Public members provide the class’s interface.
- Private members are used to implement the interface.

Important consequence:

- If a public method has a specification, it must implement that specification precisely and completely.
- Just because you never encounter an edge case doesn’t mean someone else using your class won’t.

(e.g., me grading your code using unit tests)
**Interface vs Implementation:**
A pertinent example.

```java
/** partition A around the pivot A[pivIndex].
 * return the pivot's new index.
 * precondition: start <= pivIndex < end
 * where i is the return value */
public int partition(int[] A, int start, int end, int pivIndex) {

Even if your quickSort always calls this with

    pivIndex = end-1

partition is public and must implement the spec exactly!
It needs to work if pivIndex is any value where
    start <= pivIndex < end (the precondition says so!)```
Interface vs Implementation: Javadoc Comments

There’s a **big** difference between

```java
/** this comment */
public void myMethod() {

and

```java
/* this comment */
private void myMethod() {
```

Appears in documentation that tells people how to use your class!

Does not appear elsewhere - “merely” helpful to someone reading your code.

Scanner.nextInt() documentation  Scanner.nextInt() source code
Interface vs Implementation: Tips for Assignments (and life)

- Public method specifications, names, return values, and parameters should never be changed.
- All public methods must implement its specification completely, even if your use case doesn’t require it.
- You can do basically whatever you want with private methods, as long as you stick to good coding style.
  - It’s still a good idea to write a specification and test private methods.
  - Use them to make your code easier to read (and write).
- In my A1 solution, I wrote a helper to find the pivot:
  ```java
  /* put the median of A[start], A[middle], A[end-1] at the start */
  private void medOfThree(int[] A, int start, int end) {
  ```
Interface vs Implementation

An abstract data type specifies only interface, not implementation

What the operations do

How they are accomplished
Interface vs Implementation

An abstract data type specifies only **interface**, not **implementation**

**What** the operations do

**How** they are accomplished

Abstract data types can have multiple possible **implementations**.
Abstract Data Types

- **interface** List defines an “abstract data type”
- It has public methods: add, get, remove, ...
- Various classes **implement** List:

<table>
<thead>
<tr>
<th>Class:</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Backing storage:</strong></td>
<td>array</td>
<td>chained nodes</td>
</tr>
<tr>
<td>add(i, val)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>add(0, val)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(n, val)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get(i)</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>get(0)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>get(n)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>
Stacks and Queues are **restricted** Lists

<table>
<thead>
<tr>
<th>Stack Method</th>
<th>ArrayList Time Complexity</th>
<th>LinkedList Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(i, val)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>add(0, val)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>add(n, val)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get(i)</td>
<td>$O(1)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>get(0)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>get(n)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(0)</td>
<td>$O(n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>remove(i)</td>
<td>$O(n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>remove(n)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

**Stack**

**Class:**
- ArrayList
- LinkedList

**Backing storage:**
- array
- chained Node objects
Stacks and Queues are **restricted** Lists

<table>
<thead>
<tr>
<th>Queue</th>
<th>Backing storage:</th>
<th>Class:</th>
<th>ArrayList</th>
<th>LinkedList</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>array</td>
<td>chained Node objects</td>
</tr>
<tr>
<td><strong>add(i, val)</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>add(0, val)</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>enQ(val)</strong></td>
<td><strong>add(n, val)</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><strong>peek()</strong></td>
<td><strong>get(0)</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>get(n)</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td><strong>deQ()</strong></td>
<td><strong>remove(0)</strong></td>
<td>O(n)</td>
<td>O(1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>remove(i)</strong></td>
<td>O(n)</td>
<td>O(n)</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>remove(n)</strong></td>
<td>O(1)</td>
<td>O(1)</td>
<td></td>
</tr>
</tbody>
</table>
The **Set** ADT

- A **Set** maintains a collection of **unique** things.

- Java has this ADT built in as an interface:
  
  ```java
  java.util.Set
  ```

- Some methods from `java.util.Set`:
  - `boolean add(Object ob)`
  - `boolean contains(Object ob)`
  - `boolean remove(Object ob)`
The **Set** ADT

Methods from `java.util.Set`:
- `boolean add(Object ob)`
- `boolean contains(Object ob)`
- `boolean remove(Object ob)`

Possible implementations:
- array
- linked list
- BST
- AVL Tree
The **Set** ADT

Methods from `java.util.Set`:

- `boolean add(Object ob)`
- `boolean contains(Object ob)`
- `boolean remove(Object ob)`

Possible implementations:

- **array**
- linked list
- BST
- AVL Tree

Array operations:

- indexing/assignment (`A[i] = v`)
- length (`A.length`)

How can we implement Set’s `add` method?
The **Set** ADT

A **Set** maintains a collection of **unique** things.

Methods from **java.util.Set**:
- boolean add(Object ob)
- boolean contains(Object ob)
- boolean remove(Object ob)

In small groups:
1. Write **English** descriptions of how to implement `contains` and `remove` using an **array**.
2. Do the same, but using the operations provided by an **AVL Tree**.

**Array** operations:
- indexing/assignment (A[i] = v)
- length (A.length)

**AVL** operations:
- search(Object ob)
- insert(Object ob)
- remove(Object ob)
Our next two topics:

- Priority Queues
- Hashing, **HashSets**, HashMaps
Queue vs Priority Queue

add (enqueue): inserts an item into the queue
remove (dequeue): removes the first item to be inserted (FIFO)

add (enqueue): inserts an item into the queue
remove (poll): remove the highest-priority item from the queue
Uses for Priority Queues

• Surface simplification [Garland and Heck bert 1997]
• Graph searching: Dijkstra's algorithm, Prim's algorithm
• Statistics: maintain largest M values in a sequence
• Graphics and simulation: "next time of contact" for colliding bodies
• AI Path Planning: A* search (e.g., Map directions)
• Operating systems: load balancing, interrupt handling
• Discrete optimization: bin packing, scheduling