Abstractions

Johan Montelius

KTH

HT22

interface vs implementation

interface vs implementation

Let's say you want a data structure where you should be able to:

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key
- given a key, find a value associated with the key

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key
- given a key, find a value associated with the key
- given a key, remove the value associated with the key

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key
- given a key, find a value associated with the key
- given a key, remove the value associated with the key

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key
- given a key, find a value associated with the key
- given a key, remove the value associated with the key

Adding duplicate values for a key is undefined.

interface vs implementation

Let's say you want a data structure where you should be able to:

- given a key and value, associate the value with the key
- given a key, find a value associated with the key
- given a key, remove the value associated with the key

Adding duplicate values for a key is undefined.

Let's go.

an array where the index is the key

```
public class KeyValue<Value> {

  Value[] store;
  int size = 100;

  public KeyValue() {
    store = (Value[]) new Object[this.size];
  }

  :
}
```

what if...

what if...

An index in an array 0..max does not work as a key?

comparable keys

comparable keys

We could always define an order for keys - but we might not have one.

Equality might not be the same as identity.

Identity is cheap, equality might be ... undecidable.

in Java

```
public class Person implements Comparable {

  String first;
  String last;

  @Overide
  public int compareTo(Person b) {
    int cmp = this.last.compareTo(b.last);
    if (cmp == 0)
       cmp = this.first.compareTo(b.first);
    return cmp;
  }
  :
}
```

a sorted/unsorted array of Key/Values

```
public class KeyValue<Key extends Comparable<Key>, Value>

  KeyVal[] store;
  int size = 100;

  public class KeyVal {
    Key key;
    Value val;
  }

  public KeyValue() {
    store = new KeyValue.KeyVal[this.size];
  }
  :
}
```

a linked list

```
public class KeyValue<Key, Value> {

  KeyVal store;

  private class KeyVal {
    Key key;
    Value val;
    KeyVal next;
    :
  }

  public KeyValue() { store = null; }

  :
}
```

a tree

```
public class KeyValue<Key extends Comparable<Key>, Value>

  KeyVal store;

  private class KeyVal {
    Key key;
    Value val;
    KeyVal left;
    KeyVal right;
      :
  }

  public KeyValue() { store = null; }
    :
}
```

time complexity

operation	array*	unsorted	sorted	list	tree**

*** using indicies as keys*

** given that the tree is fairly balanced

time complexity

operation	array*	unsorted	sorted	list	tree**
lookup	$O(1)$	$O(n)$	$O(lg(n))$	$O(n)$	$O(lg(n))$

*** using indicies as keys*

** given that the tree is fairly balanced

time complexity

operation	array*	unsorted	sorted	list	tree**
lookup	$O(1)$	$O(n)$	$O(lg(n))$	$O(n)$	$O(lg(n))$
add	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(lg(n))$

*** using indicies as keys*

** given that the tree is fairly balanced

time complexity

operation	array*	unsorted	sorted	list	tree**
lookup	$O(1)$	$O(n)$	$O(lg(n))$	$O(n)$	$O(lg(n))$
add	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(lg(n))$
remove	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(lg(n))$

*** using indicies as keys*

** given that the tree is fairly balanced

Evaluation

Which implementation to choose:

- The keys: are indeces 0..max fine

Evaluation

Which implementation to choose:

- The keys: are indeces 0..max fine
- The keys: is there an order?

Evaluation

Which implementation to choose:

- The keys: are indeces 0..max fine
- The keys: is there an order?
- Which operations are most frequent/critical?

Evaluation

Which implementation to choose:

- The keys: are indeces 0..max fine
- The keys: is there an order?
- Which operations are most frequent/critical?
- How about the memory footprint?

Evaluation

Which implementation to choose:

- The keys: are indeces 0..max fine
- The keys: is there an order?
- Which operations are most frequent/critical?
- How about the memory footprint?

other operations

other operations

- Split a data structure in two.

other operations

- Split a data structure in two.
- Merge two structures.

other operations

- Split a data structure in two.
- Merge two structures.
- Selecting a range of keys.

other operations

- Split a data structure in two.
- Merge two structures.
- Selecting a range of keys.
- Selecting keys that are "close to each other" but not necessarily in order.

an alternative implementation of a tree

an alternative implementation of a tree

Let's store the nodes of the tree in an array.

an alternative implementation of a tree

Let's store the nodes of the tree in an array.

- The root of the tree is at index 0.

an alternative implementation of a tree

Let's store the nodes of the tree in an array.

- The root of the tree is at index 0.
- The left branch of a node at i is at index $i \times 2 + 1$.

an alternative implementation of a tree

Let's store the nodes of the tree in an array.

- The root of the tree is at index 0.
- The left branch of a node at i is at index $i \times 2 + 1$.
- The right branch of a node at i is at index $i \times 2 + 2$.

an alternative implementation of a tree

Let's store the nodes of the tree in an array.

- The root of the tree is at index 0.
- The left branch of a node at i is at index $i \times 2 + 1$.
- The right branch of a node at i is at index $i \times 2 + 2$.

a generic key/value tree

```
public class Tree<Key extends Comparable<Key>, Value> {

  KeyVal[] store;
  int size;

  public class KeyVal {
    Key key;
    Value val;
  }

  public KeyVal(int max) {
    this.size = max;
    this.store = new Tree.KeyVal[max];    // warning
  }
  :
```

add a key/value pair

```
public void add(Key k, Value v) {
  int indx = 0;
  while (true) {
    if (store[indx] == null) {
      store[indx] = new KeyVal(k,v);
      break;
    }
    if (store[indx].key == k) {
      store[indx].val = v;
      break;
    }
    :
```

add a key/value pair

```
      :
      if (store[indx].key.compareTo(k) > 0) {
        indx = 2*indx + 1;
      } else {
        indx = 2*indx + 2;
      }
    }
  }
```

lookup a value given key

```
public Value lookup(Key k) {
  int indx = 0;
  while (true) {
    if (store[indx] == null) { break; }
    if (store[indx].key == k) { return store[indx].val;}
    if (store[indx].key.compareTo(k) > 0) {
      indx = 2*indx + 1;
    } else {
      indx = 2*indx + 2;
    }
    if (indx >= this.size) break;
  }
  return null;
}
```

what's the catch

what’s the catch

When might an array implementation of a tree not be a suitable solution?

Abstractions

Abstractions

- There is a difference between the interface provided, and the implementation.

Abstractions

- There is a difference between the interface provided, and the implementation.
- The less requirements specified by the interface, the more freedome do we have in the implementation.

Abstractions

- There is a difference between the interface provided, and the implementation.
- The less requirements specified by the interface, the more freedome do we have in the implementation.
- Linked data structures and arrays are questions about the implementation.

Abstractions

- There is a difference between the interface provided, and the implementation.
- The less requirements specified by the interface, the more freedome do we have in the implementation.
- Linked data structures and arrays are questions about the implementation.
- The interface describes the functionality and ... runtime complexity.

Examples of abstractions

- A key/value store: add, lookup, remove, ...

Examples of abstractions

- A key/value store: add, lookup, remove, ...
- A stack : push, pop, constant time operations

Examples of abstractions

- A key/value store: add, lookup, remove, ...
- A stack : push, pop, constant time operations
- A queue : enqueue, dequeue, constant time operations

Examples of abstractions

- A key/value store: add, lookup, remove, ...
- A stack : push, pop, constant time operations
- A queue : enqueue, dequeue, constant time operations
- ... there will be more.

One man's ceiling ..

One man's ceiling ..

... is another man's floor.