
# Abstract Data Types: Sets

The Set ADT

A Set Interface

# The Set ADT

The **set** ADT corresponds to a mathematical *set*. A set has these fundamental properties:

- duplicates *are not* allowed
- order is *not* preserved

A set may support operations such as these:

- *create*: construct an empty set
- *add*: add an element to the set
- *contains*: does the set contain a given element
- *remove*: remove an element from the set

# Our Set Interface

We will explore sets using a simple interface :

```
public interface Set<T> {
    boolean add(T value);
    boolean contains(T value);
    int size();
    boolean remove(T value);
}
```


# Abstract Data Types: Hash Tables

# A4

Hash Table
Implementation of a Set 1

# Equality and Hashing in Java

- `a == b` : true iff a and b reference the same object.
- `a.equals(b)` : class-specific (semantic) object equality.
- Default inherited from java.lang.Object is just `==`.
- `a.hashCode()` : returns a hash (32-bit signed integer) of object.
  - **Requirement:** If `a.equals(b)` then must have `a.hashCode() == b.hashCode()`
  - Default implementation inherited from java.lang.Object, but likely must be overridden whenever you override `equals` to meet this requirement.

# Hash Tables

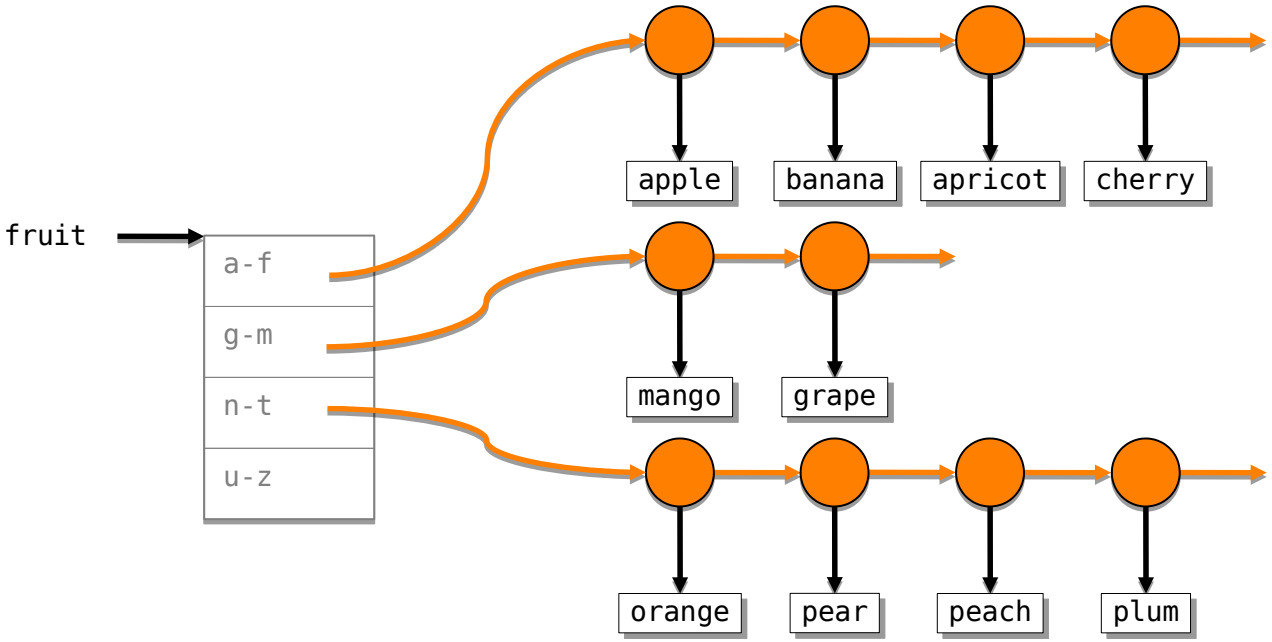
A hash table stores *keys*, using a hash function to map each key into a table entry. Keys can be associated with *values*.

Challenges are: a) dealing with hash collisions, and b) dealing with load (how big to make the table).

Two broad approaches:

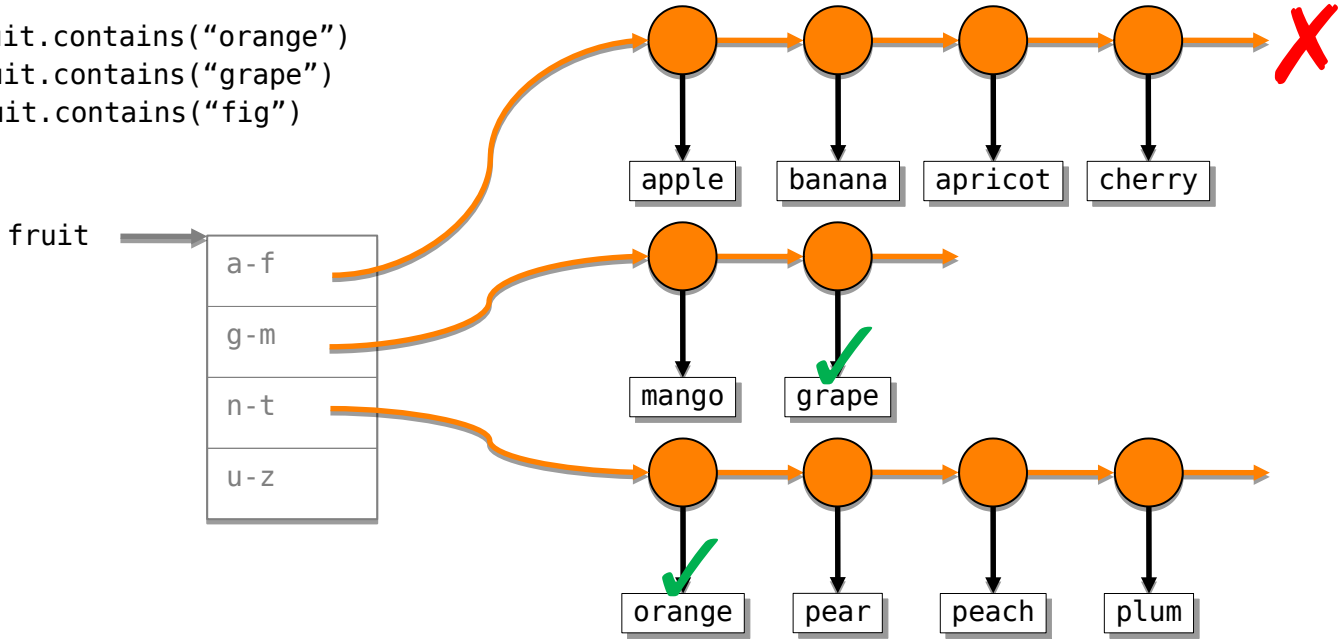
- Separate chaining
  - Hash table entries are lists of keys or (key, value) pairs.
- Open addressing
  - Hash table entries are only keys/(key, value) pairs.
  - Collisions resolved by *probing* – e.g., find next empty table slot.

# Abstract Data Types: Hash Table A4



# Abstract Data Types: Hash Table A4

```
fruit.contains("orange")
fruit.contains("grape")
fruit.contains("fig")
```



- By resizing (doubling) table capacity when lists grow too long, `add` and `contains` can run in amortised constant time (assuming a good hash function).

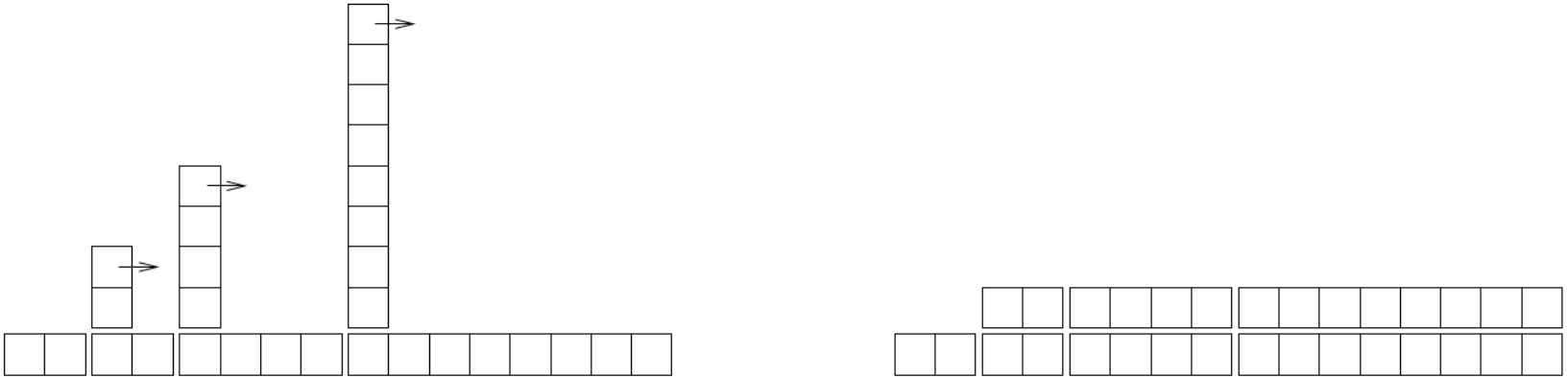
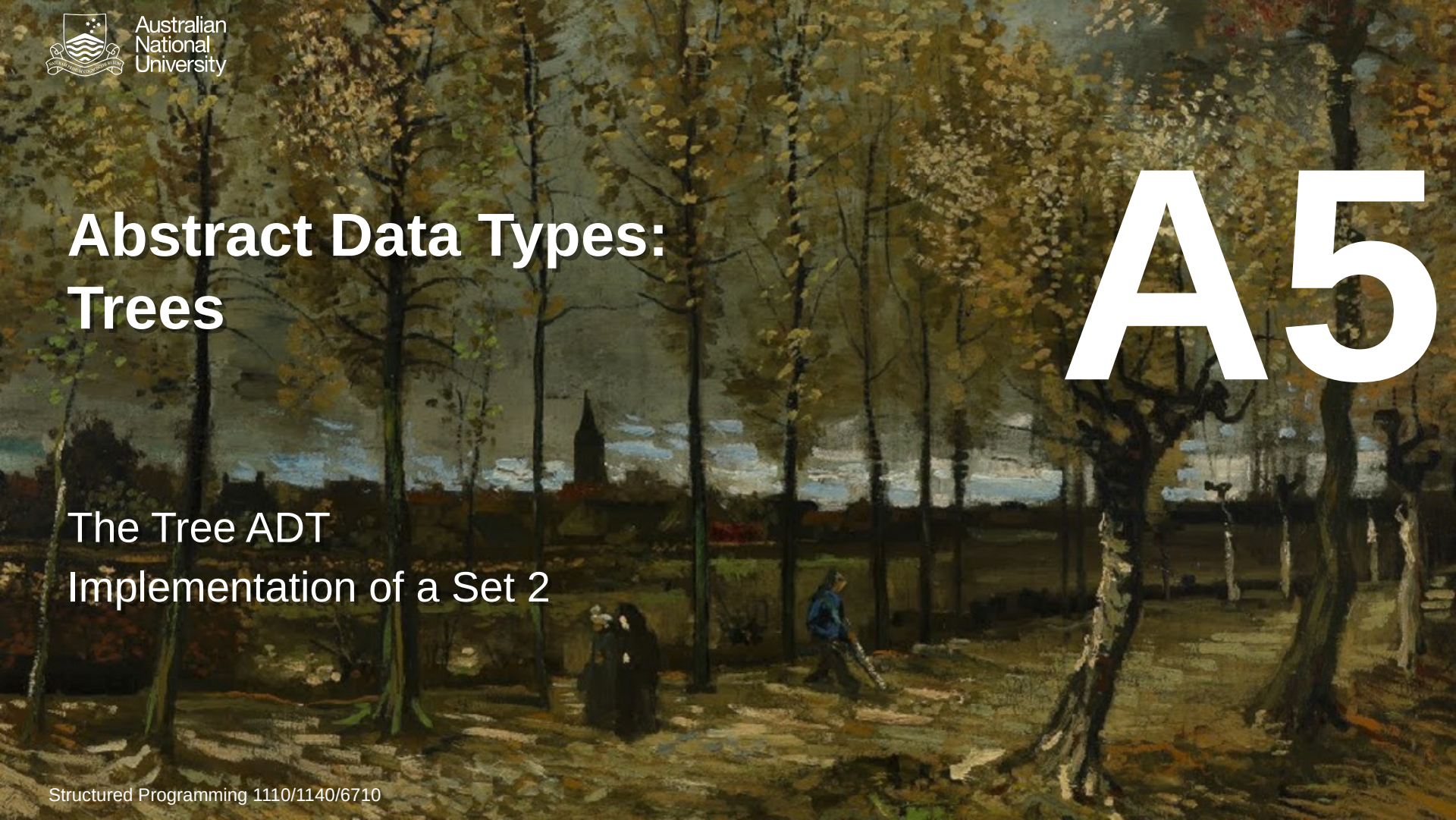


Figure B.1: The cost of a hashtable add.

(Illustration from “Think Python: How to think like a computer scientists” (2nd ed) by Allen B. Downey.)



Australian National University

# Abstract Data Types: Trees

# A5

The Tree ADT

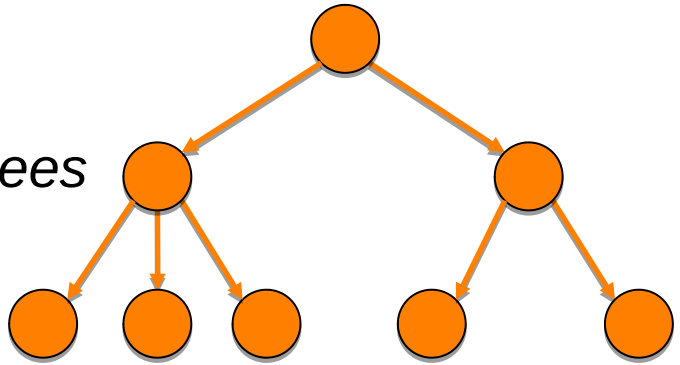
Implementation of a Set 2

# The Tree ADT

The **tree** ADT corresponds to an *ordered tree* in mathematics.

A tree is defined recursively in terms of nodes:

- A tree is a node
- A node contains a *value (key)* and a list of *trees*
- No node is duplicated

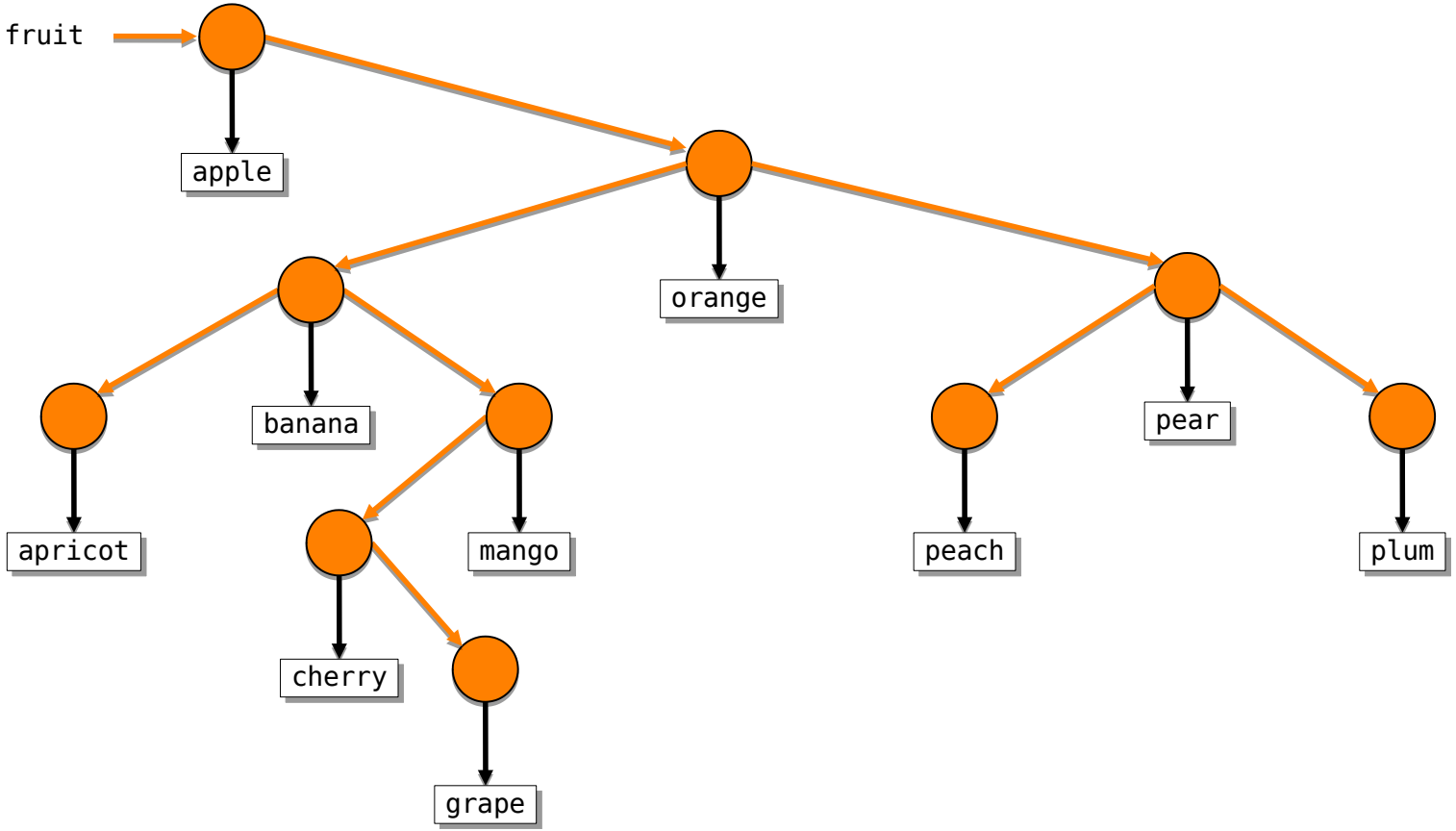


# Binary Search Tree

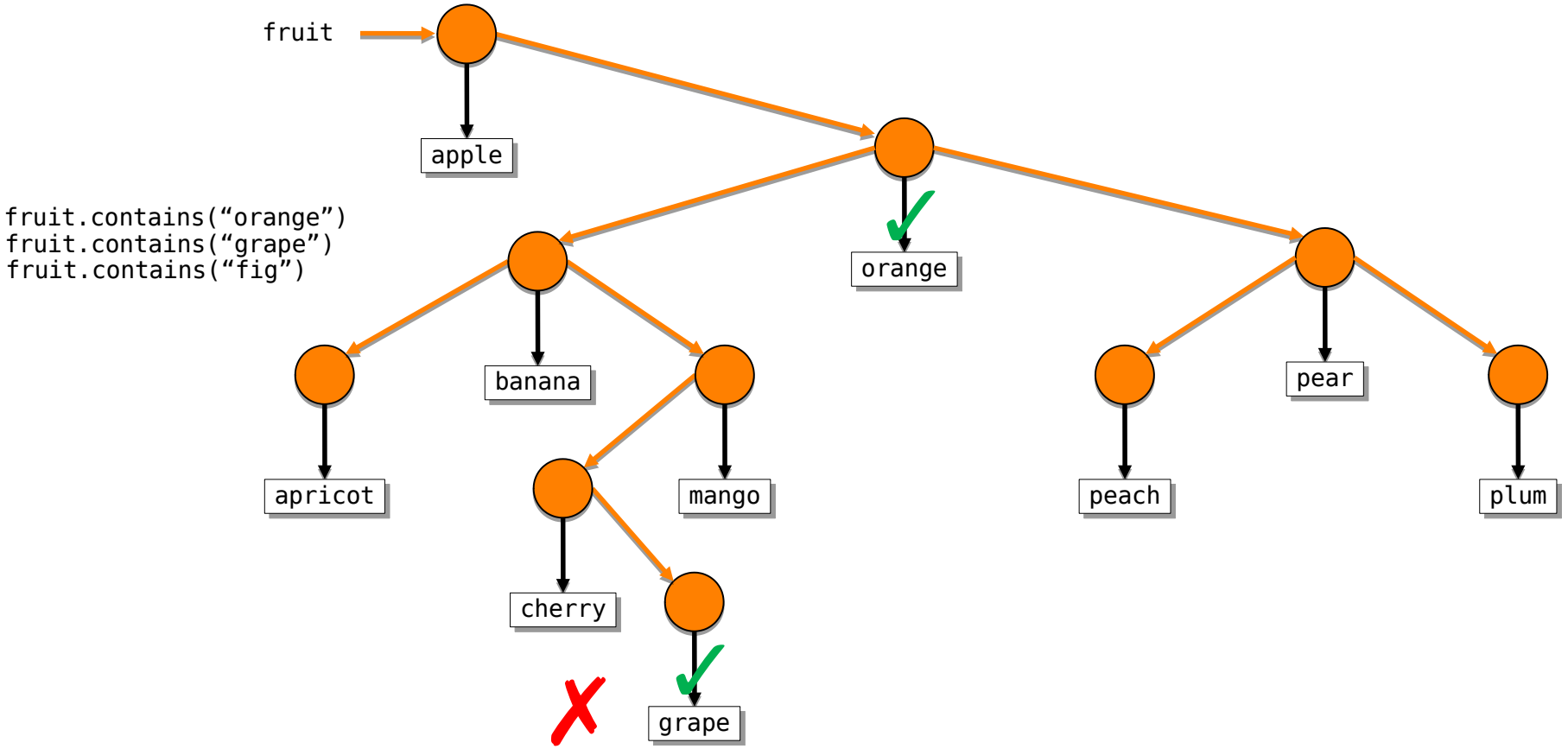
A **binary** search tree is a tree with the following additional properties:

- Each node has *at most* **two** sub-trees
- Nodes may contain *(key, value*) pairs, or just keys
- Keys are ordered within the tree:
  - The left sub-tree only contains keys less than the node's key
  - The right sub-tree only contains keys greater than the node's key

# Abstract Data Types: Trees A5



# Abstract Data Types: Trees A5



# Ordering in Java

Objects of any class that implements the `Comparable` interface can be ordered:

```
a.compareTo(b)
```

- `< 0` iff a is ordered before b
- `> 0` iff a is ordered after b
- `== 0` if `a.equals(b)` (but also if a and b are not ordered)