ADTs
List as an ADT
A List interface

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Abstract Data Types (ADTs)

Abstract data types* describe the behaviour (semantics) of a data type without specifying its implementation. An ADT is thus **abstract**, not concrete.

- A **container** is a very general ADT, a holder of objects.
- A **list** is an example of a more specific container ADT.

* Not to be confused with: *Algebraic* Data Type.

The List ADT

The **list** ADT is a container known mathematically as a finite sequence of elements. A list has these fundamental properties:

- duplicates are allowed
- order is preserved

A list may* support operations such as these:

- *create*: construct an empty list
- *add*: add an element to the list
- *is empty*: test whether the list is empty

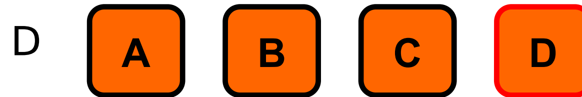
* The operations a given ADT must support will vary depending on the author / library

Our List Interface

We will explore lists using a simple interface:

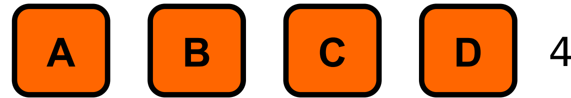
```
public interface List<T> {
    void add(T value);
    T get(int index);
    int size();
    T remove(int index);
    void reverse();
}
```

```java
void add(T value);
```

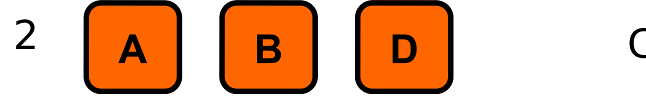



```java
T get(int index);
```

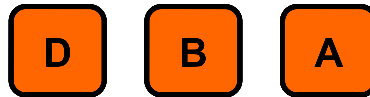
```java
int size();
```




```java
T remove(int index);
```




```java
void reverse();
```




```java
String toString();
```

A02 List Implementations

An array-based implementation
A linked-list-based implementation

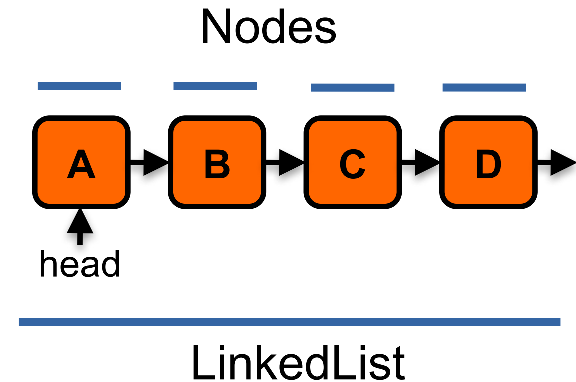
ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

List Implementation Options

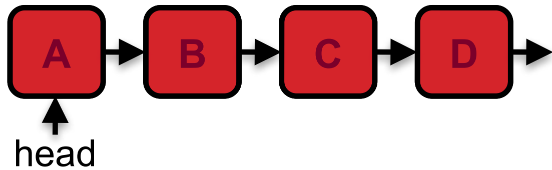
- Arrays
  - Fast lookup of any element
  - A little messy to grow and contract
- Linked list
  - Logical fit to a list, easy to grow, contract
  - Need to traverse list to access elements

Linked Lists: Singly Linked List

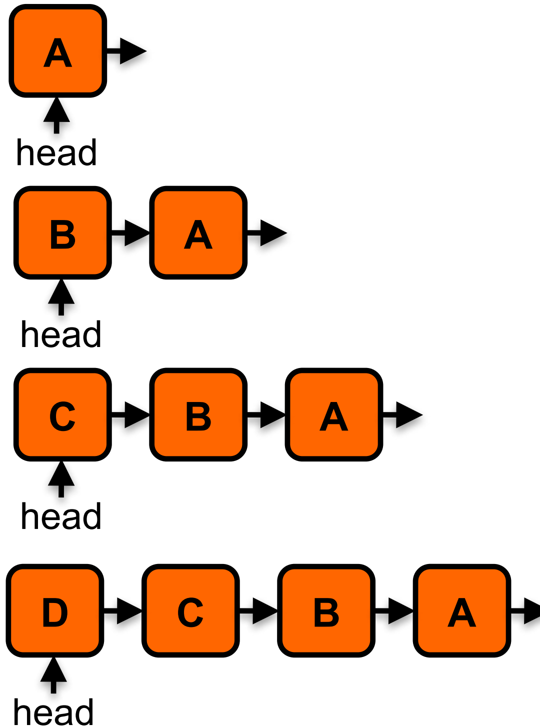
```
public class LinkedList<T> {
  private class Node<T> {
    T value;
    Node<T> next;
  }
  Node<T> head;
}
```



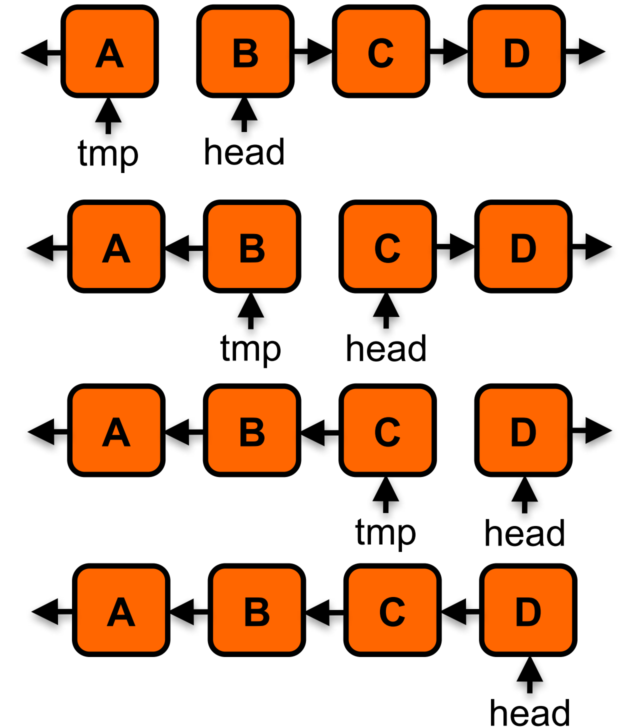
Linked List Reversal: Two Approaches



- Add each item to start:



- Pointer reversal:



Complexity

```
void add(T value);
T get(int index);
int size();
T remove(int index);
void reverse();
```

ArrayList

- `add` – **Time O(1) amortized, O(n) worst**
- `get` – **Time O(1)**
- `size` – **Time O(1)**
- `remove` – **Time O(n)**
- `reverse` – **Time O(n)**

Space O(n)

LinkedList

- `add` – **Time O(1)**
  - if explicitly tracking last node
- `get` – **Time O(n)**
- `size` – **Time O(1)**
  - if explicitly tracked
- `remove` – **Time O(n)**
- `reverse` – **Time O(n)**

Space O(n)

A03 Sets

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);
}
```

Hash tables
A hash-table-based Set implementation

Hash Tables

Stores **keys**, using a hash function to map a key into a table entry. Optionally, **values** can be associated with keys and stored alongside them in the table.

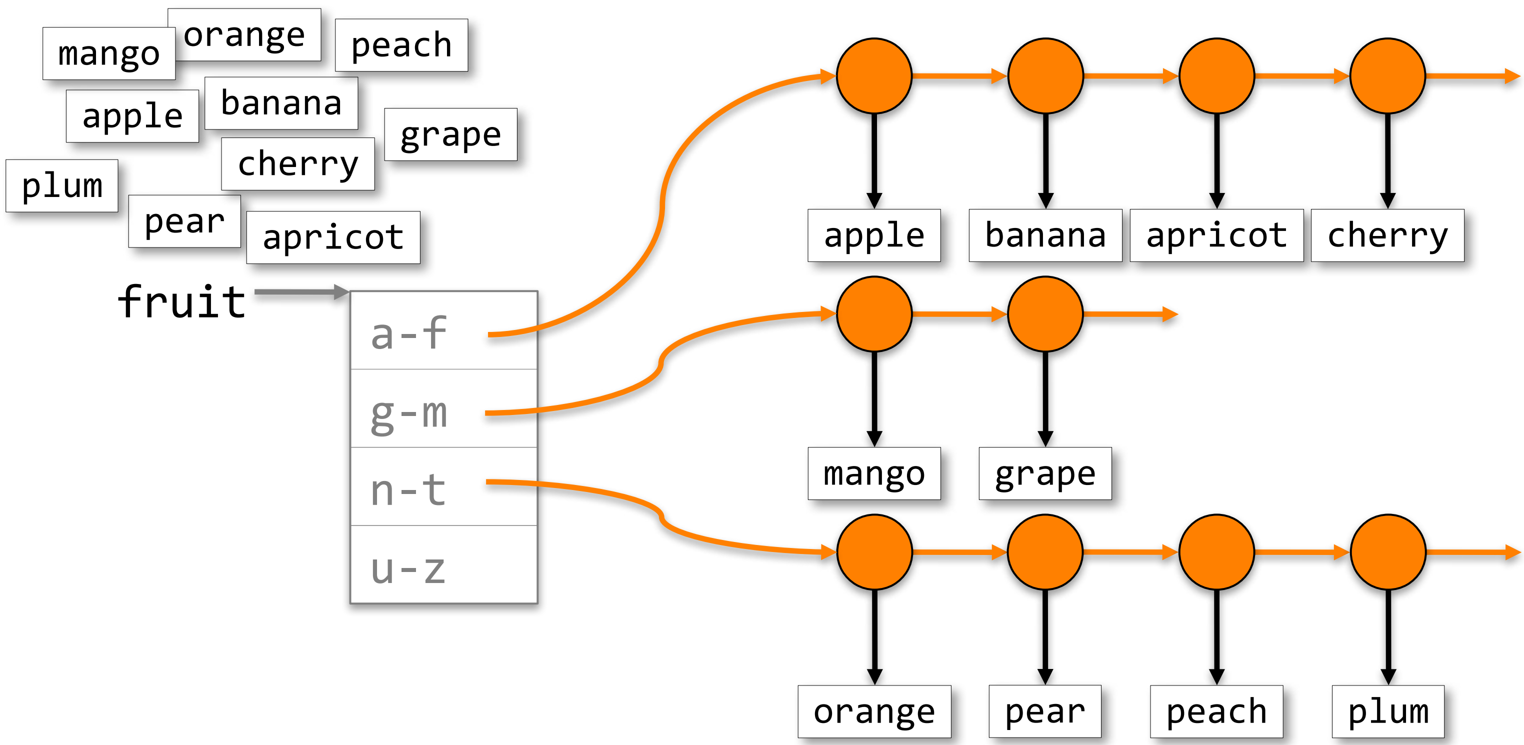
Main challenges are: a) dealing with **hash collisions** and dealing with **load** (how big to make the table).

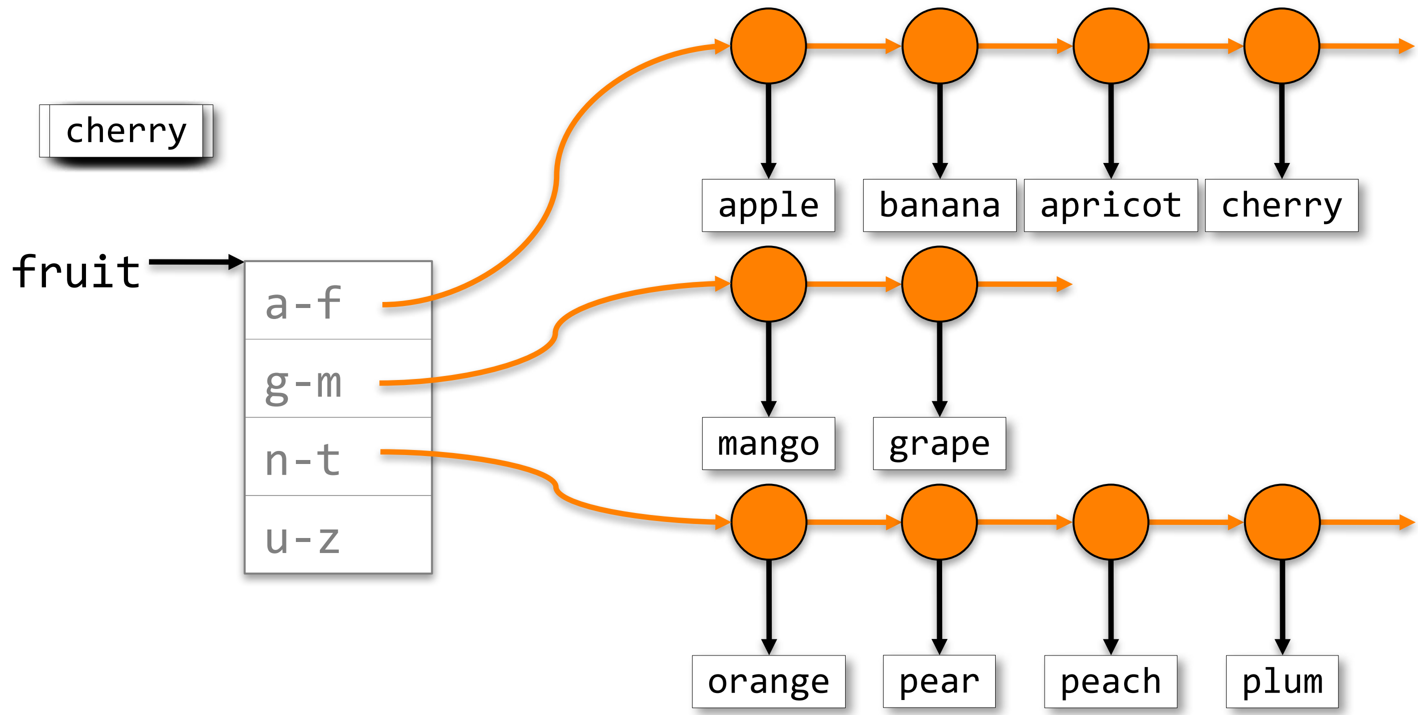
Two broad approaches:

- Separate chaining
  - Hash table entries are lists: (key, value) pairs are in lists.
- Open addressing
  - Hash table entries are (key, value) pairs.
  - Collisions resolved by probing – e.g. find next entry slot

HashSet Implementation of a Set

- Special case of hash table where we only have **key** (it is not associated with any *value*).
- We'll demonstrate **separate chaining** where our lists only needs to store a single item rather than a pair.

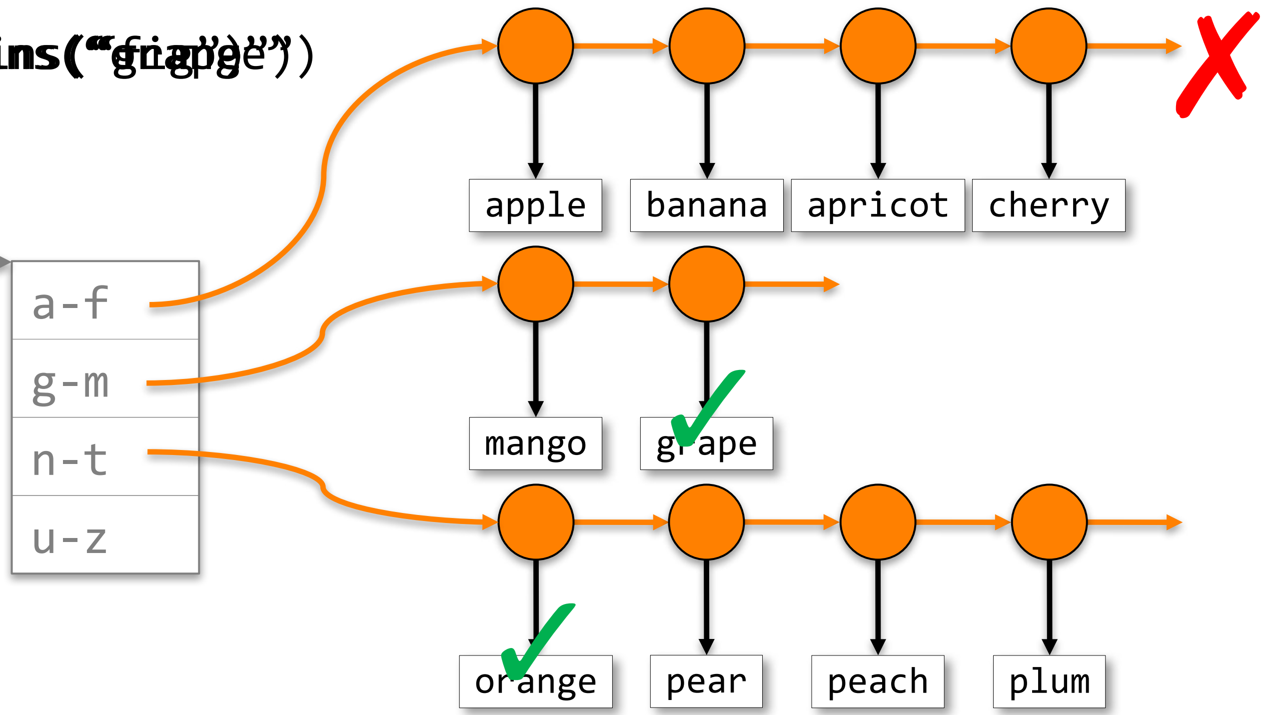

mango
orange
peach
apple
banana
grape
cherry
plum
pear
apricot
fruit
a-f
g-m
n-t
u-z
apple
banana
apricot
cherry
mango
grape
orange
pear
peach
plum


cherry
fruit
a-f
g-m
n-t
u-z
apple
banana
apricot
cherry
mango
grape
orange
pear
peach
plum

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

grape

fruit



Load Factor

The **load factor** is the ratio of number of elements to the number of "buckets" (size of table).

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

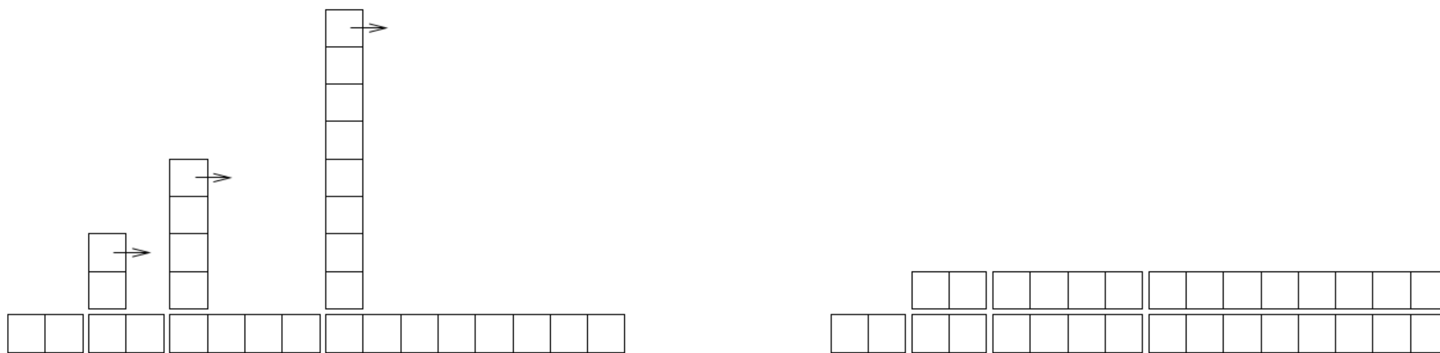


Figure B.1: The cost of a hashtable add.

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

Complexity

```
boolean add(T value);
boolean contains(T value);
int size();
boolean remove(T value);
```

- `add`, `contains`, `remove` – **Time O(1) amortized, O(n) worst**
  - *good* hash function
  - table resized to keep table *load factor* in a range
- `size` – **Time O(1)**
  - explicitly tracked

Space O(n)

A05 Sets: TreeSet

Tree as an ADT
A tree-based Set implementation

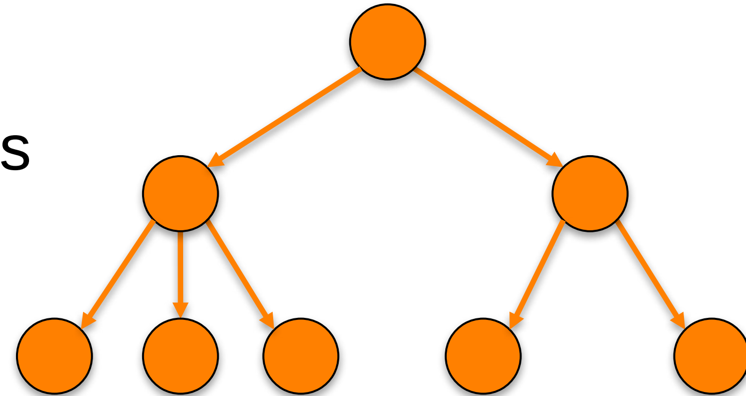
ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

Tree as an ADT

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

A tree is defined recursively in terms of nodes:

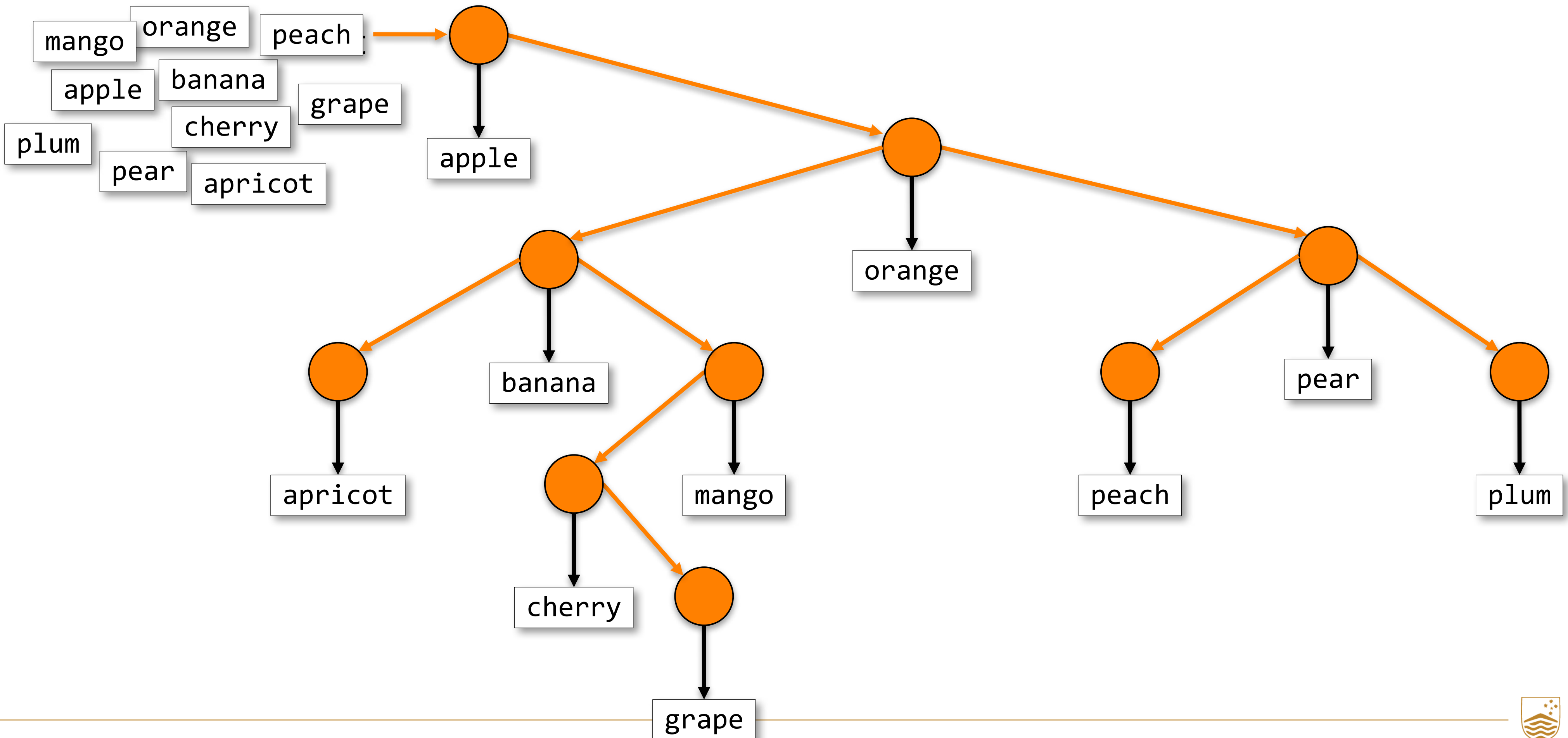
- A tree is a node
- A node contains a value and a list of trees
- No node is duplicated

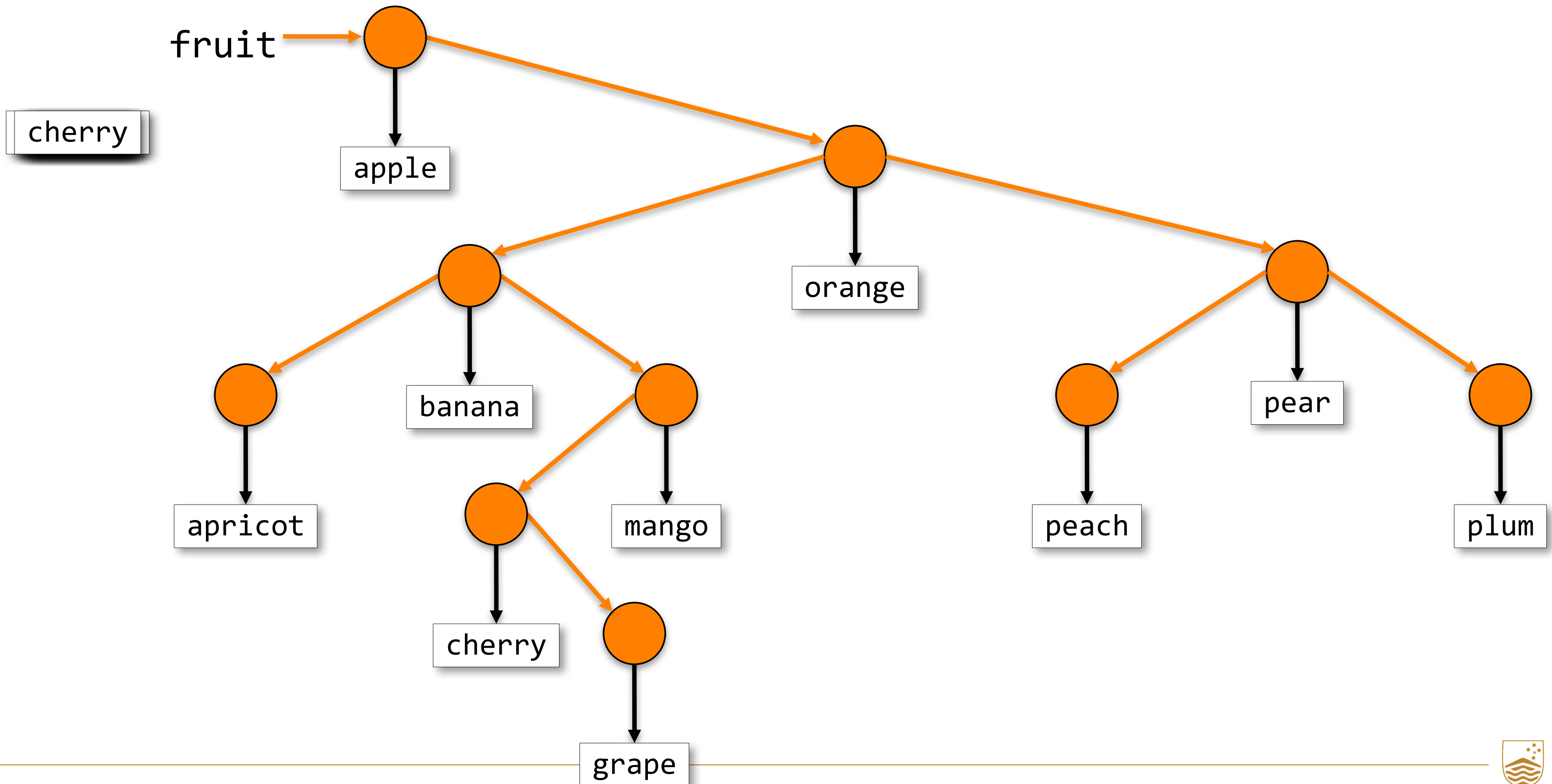


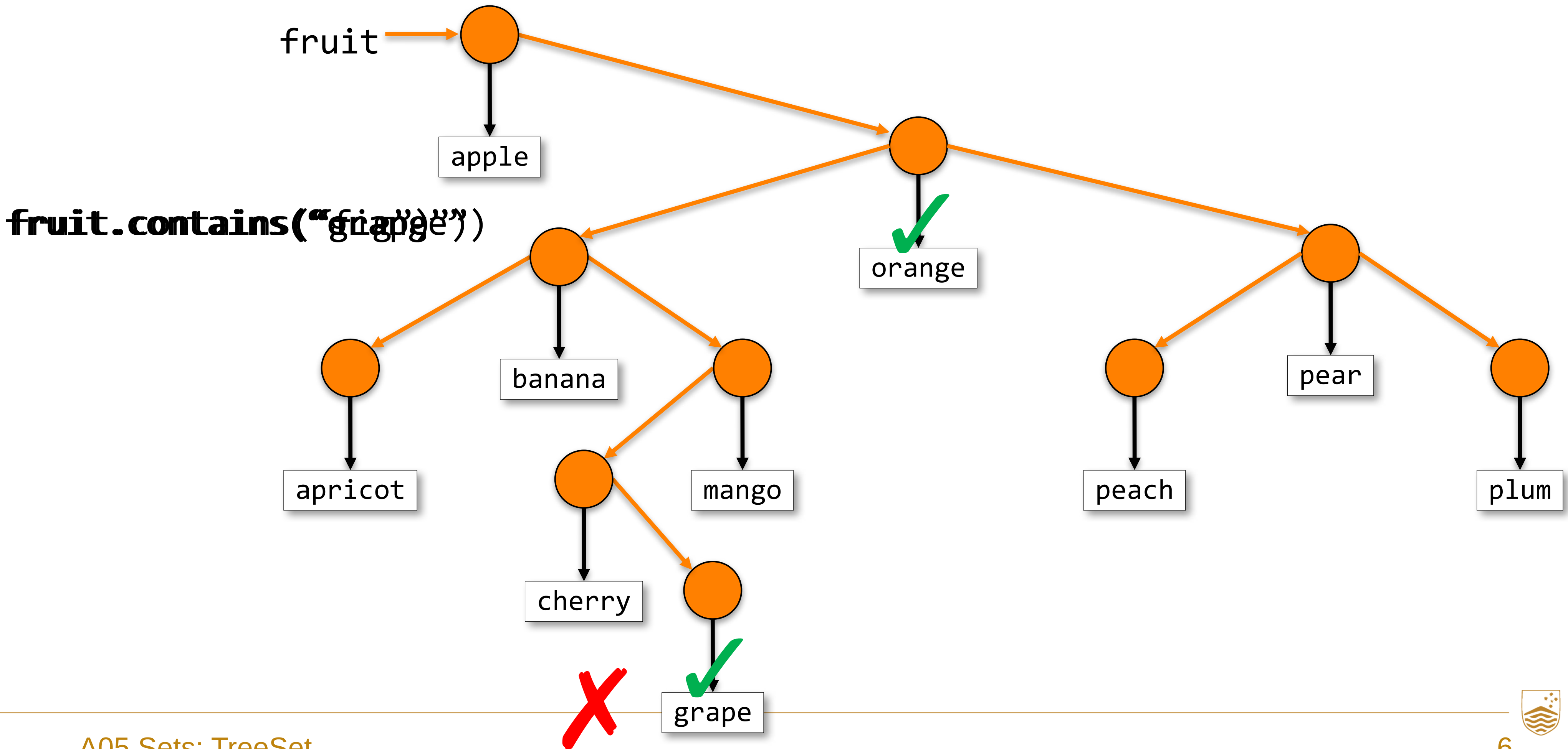
Binary Search Tree to Implement Set

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


mango
orange
peach
apple
banana
grape
plum
cherry
pear
apricot
apple
orange
banana
pear
apricot
mango
peach
plum
cherry
grape


fruit
cherry
apple
orange
banana
apricot
mango
cherry
grape
peach
pear
plum


fruit
apple
orange
banana
pear
apricot
mango
peach
plum
cherry
grape

Ordering in Java (Recall J14)

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)

Our `Set` interface does not bound our contained type parameter to be `Comparable`, what to do?

- Bound T in the `TreeSet` class declaration:
  - `class TreeSet<T extends Comparable<T>> implements Set<T>`
- Throw runtime exception on use of non-comparable types (the approach in `java.utils.TreeSet`).
- Force users to provide `Comparator` (e.g., as lambda expression).

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/lang/Comparable.html

Complexity

```
boolean add(T value);
boolean contains(T value);
int size();
boolean remove(T value);
```

- `add`, `contains`, `remove` – **Time O(log(n)) amortized, O(n) worst**
  - self-balancing trees (e.g., B-Trees) have O(log(n)) worst case
- `size` – **Time O(1)**
  - explicitly tracked

Space O(n)

A06 Maps: HashMap and TreeMap

Map as an ADT
A Map interface
A hash-table-based Map implementation
A tree-based Map implementation

ANU – School of Computing – Structured Programming 1110 / 1140 / 6710

ADT Recap

First-principles implementation of three Java container types:

- List
  - **`ArrayList`**, **`LinkedList`** implementations (A1, A2)
- Set
  - **`HashSet`**, **`TreeSet`** implementations (A3, A4, A5)
- Map
  - **HashMap**, **TreeMap** implementations (A6)

Introduced **hash tables**, **trees** (A4, A5)

The Map ADT (also known as Associative Array)

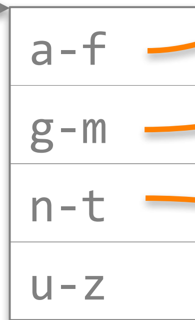
A map consists of **(key, value)** pairs

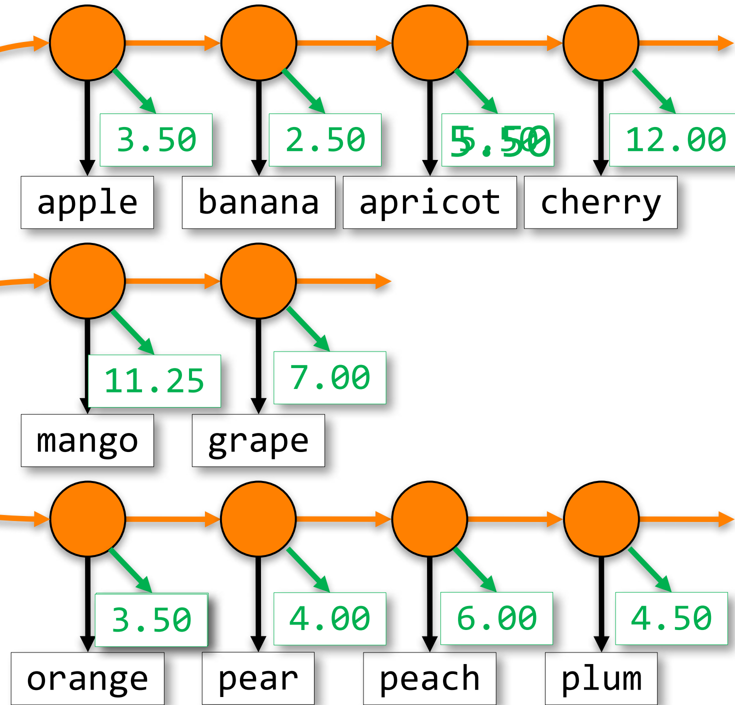
- Each key may occur only once in the map
- Values are retrieved from the map via the key
- Values may be modified
- Key, value pairs may be removed

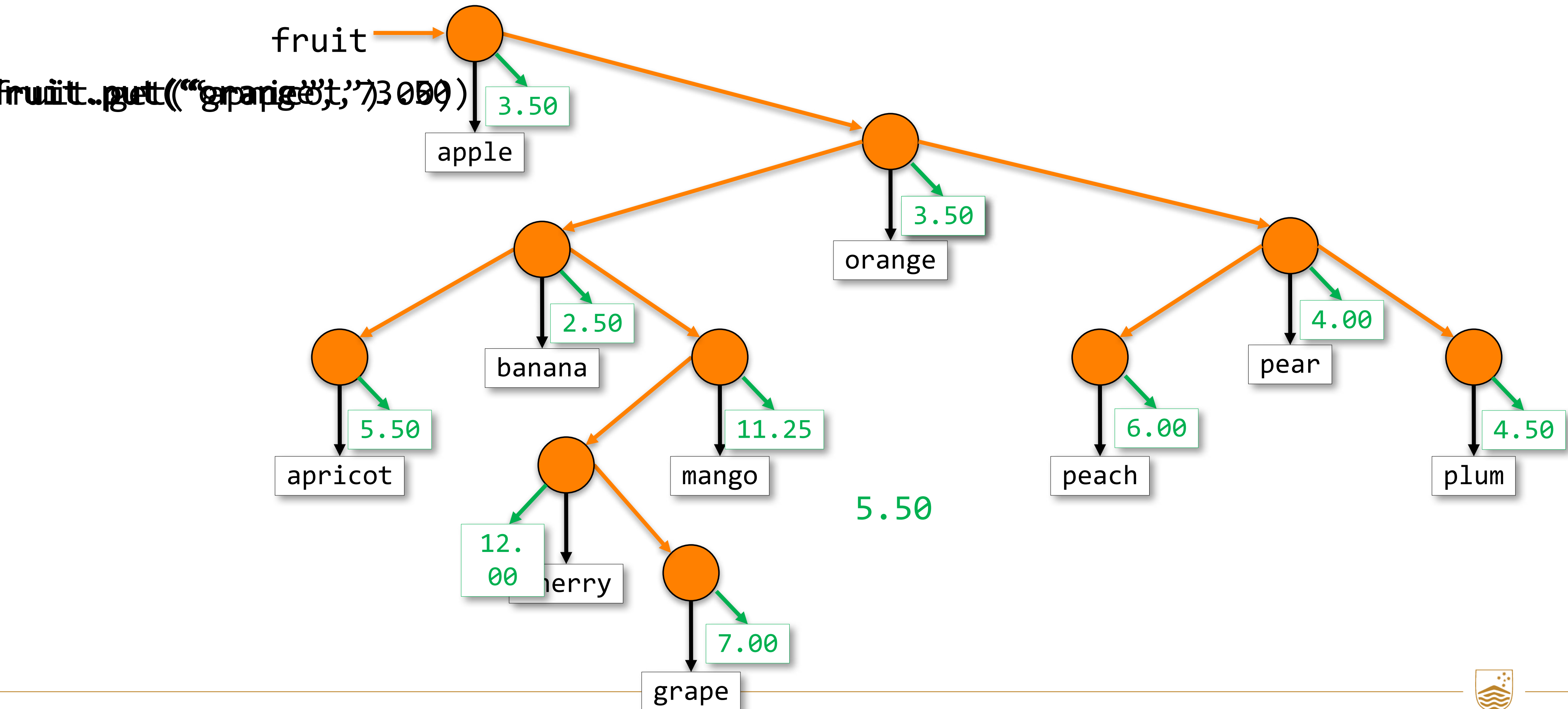
Our Map Interface

We will explore maps using a simple interface:

```
public interface Map<K,V> {
  V put(K key, V value);
  V get(K key);
  V remove (K key);
  int size();
}
```


fruit
a-f
g-m
n-t
u-z


3.50
2.50
12.00
apple
banana
apricot
cherry
11.25
7.00
mango
grape
3.50
4.00
6.00
4.50
orange
pear
peach
plum


fruit
3.50
apple
3.50
orange
2.50
banana
4.00
pear
5.50
apricot
11.25
mango
6.00
peach
4.50
plum
12.
00
herry
7.00
grape
5.50