Australian National University

A1

Abstract Data Types: Lists 1

ADTs

The List ADT

A List interface and its implementation: Array List

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, serving as a holder of objects. A **list** is an example of a specific container ADT.

An ADT is described in terms of the semantics of the operations that may be performed over it.

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:

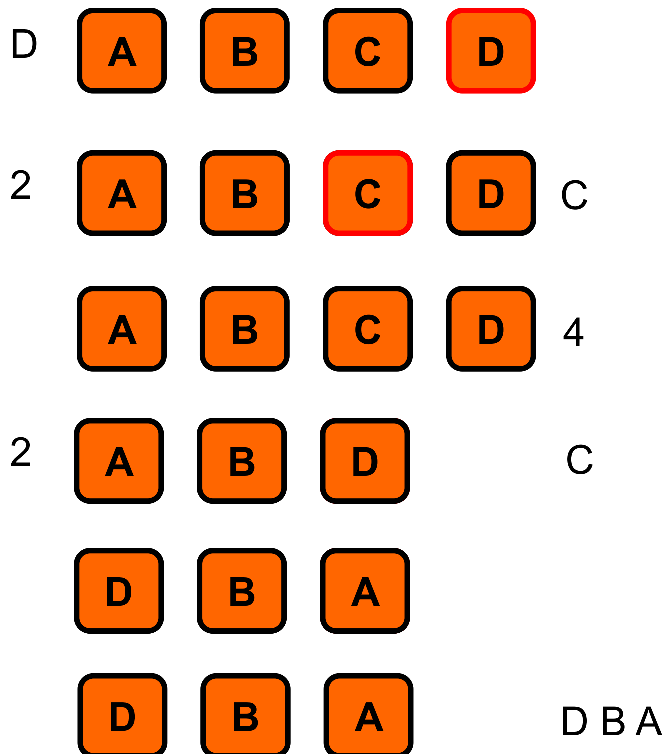
- *create*: construct an empty list
- *add*: add an element to the list
- *is empty*: test whether the list is empty

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();
}
```

```
void add(T value);

T get(int index);

int size();

T remove(int index);

void reverse();

String toString();
```



List Implementation

- 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 find arbitrary element

Australian National University

Abstract Data Types: Lists 2

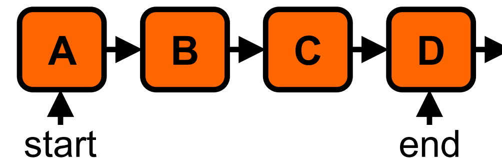
A2

A List interface and its implementation: Linked List

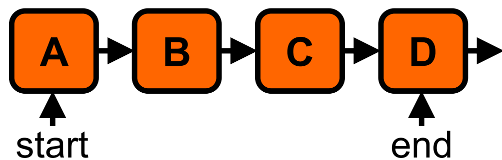
List Implementation: Linked Lists

- 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 find arbitrary element

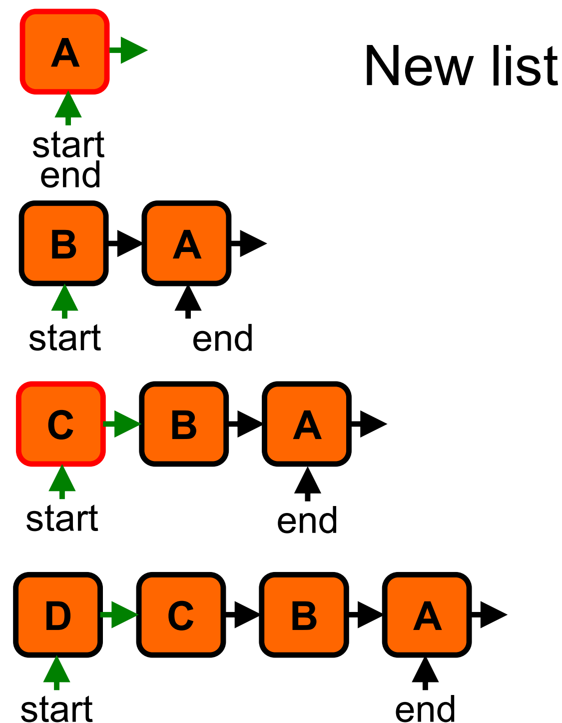
```
public class LinkedList<T> {
  private class LLNode<T> {
    T value;
    LLNode<T> next;
  }
  LLNode<T> start;
  LLNode<T> end;
}
```



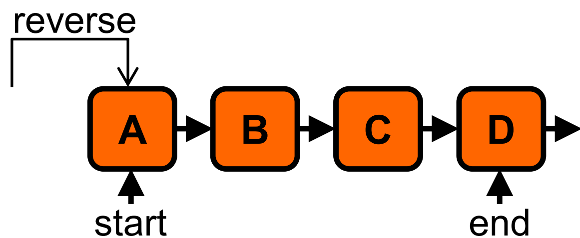
Linked List Reversal



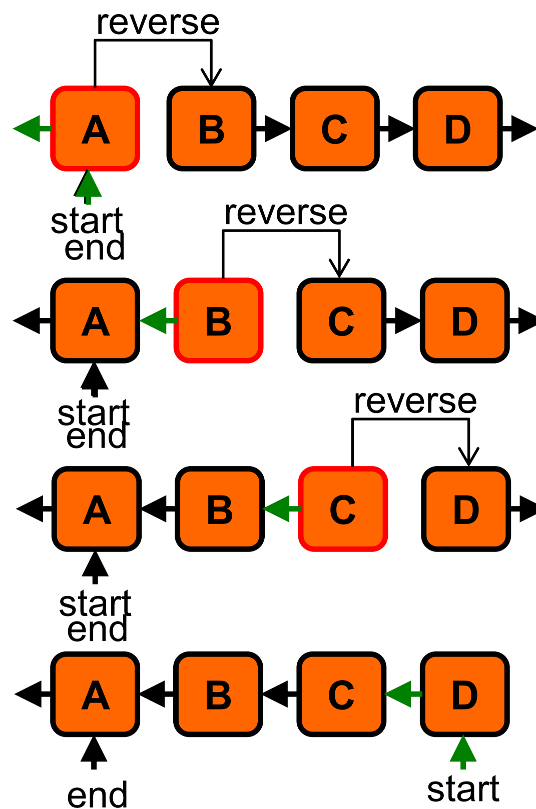
New list



Linked List Reversal



Pointer reversal



Australian National University

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);
}
```

Australian National University

Abstract Data Types: Hash Tables

A4

Hash Table

Implementation of a Set 1

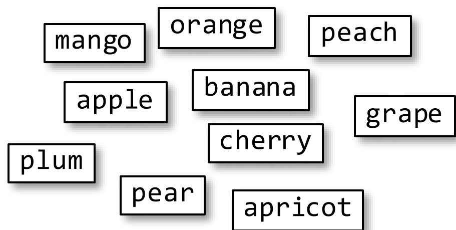
Hash Tables

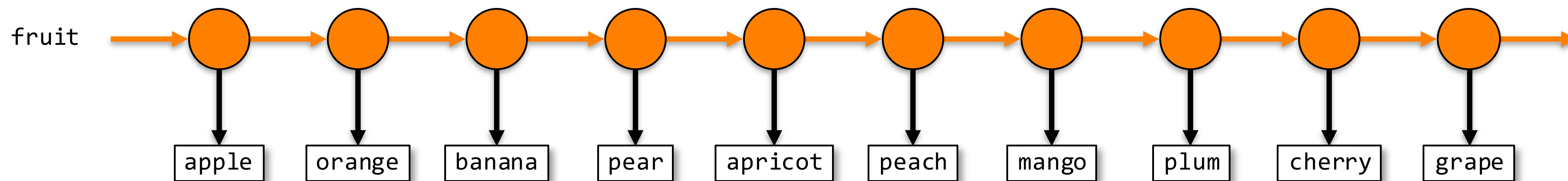
A hash table stores (*key, value*) pairs, using a hash function to map a key into a table. Key challenges are: a) dealing with hash collisions, and b) dealing with load (how big to make the table).

Two broad approaches:

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

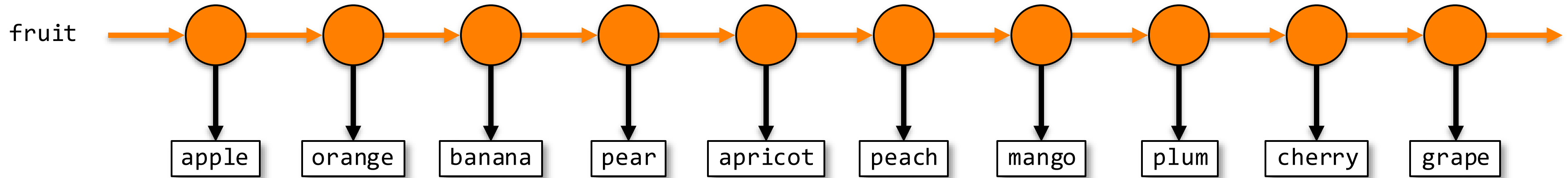
Abstract Data Types: Hash Table





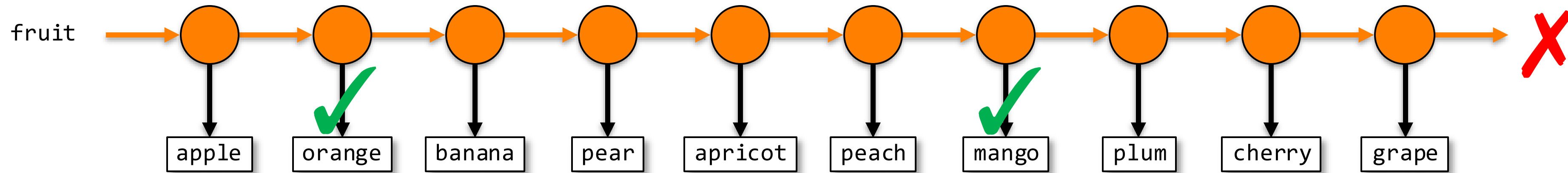
Abstract Data Types: Hash Table A4

```
fruit.add(“[illegible]”)
```

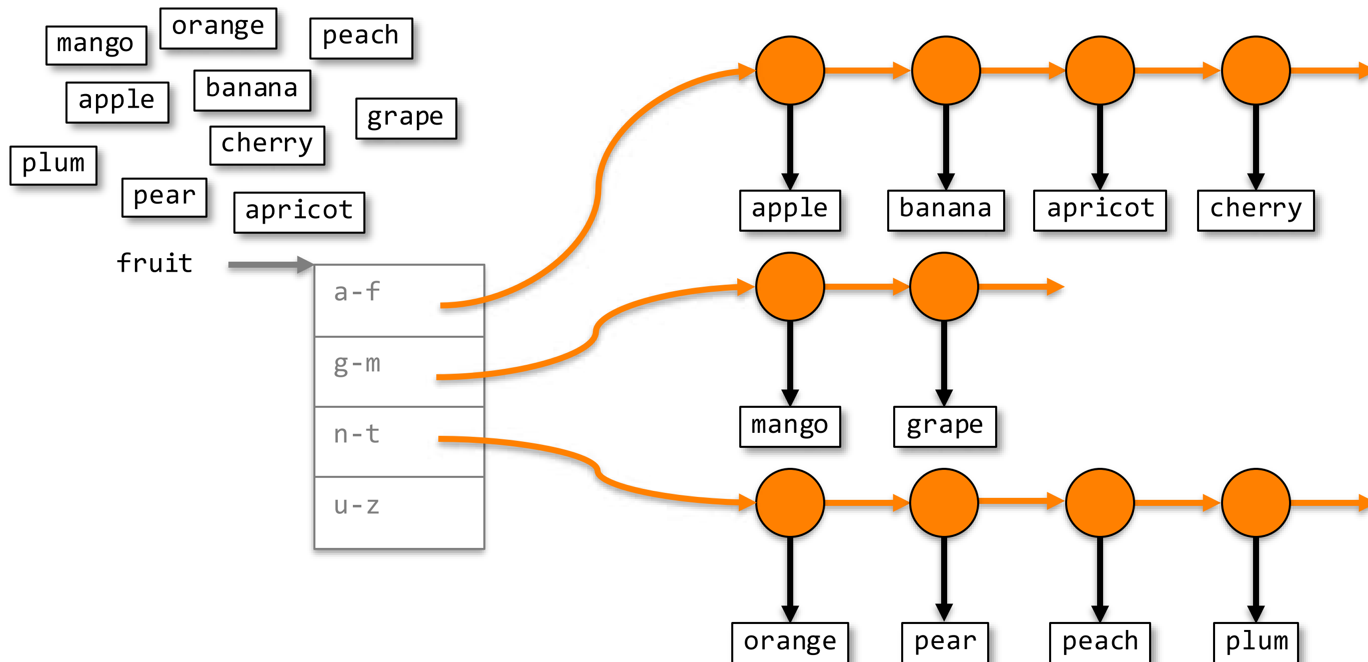


Abstract Data Types: Hash Table

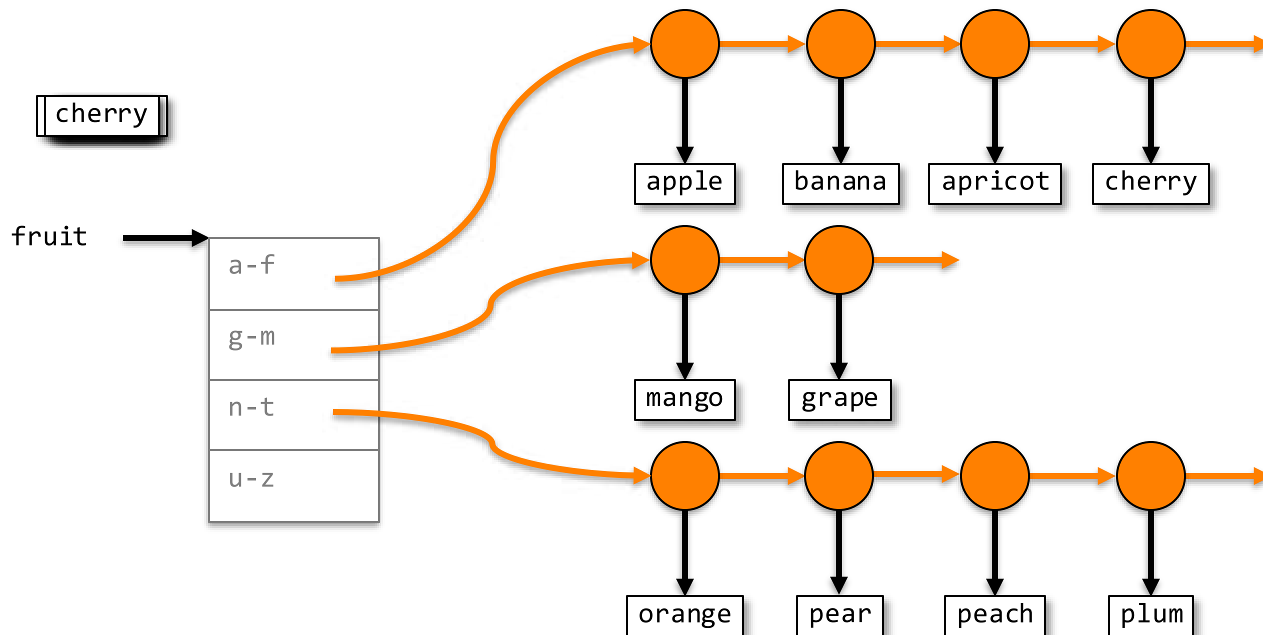

```
fruit.contains(“orange”)
fruit.contains(“mango”)
fruit.contains(“fig”)
```



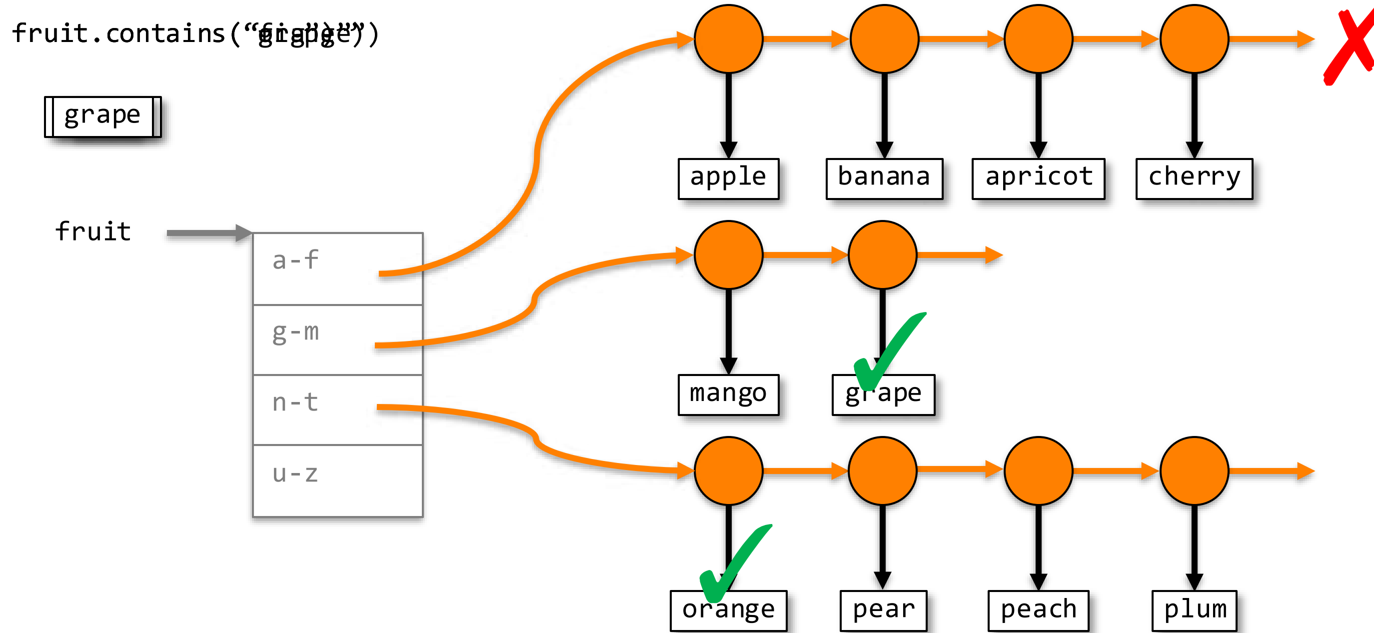
Abstract Data Types: Hash Table



Abstract Data Types: Hash Table A4



Abstract Data Types: Hash Table A4



Abstract Data Types: Trees

A5

The Tree ADT

Implementation of a Set 2

The Tree ADT

The **tree** ADT corresponds to a mathematical *tree*. 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

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

mango orange peach apple banana grape cherry plum pear apricot

apple

orange

banana

apricot

mango

cherry

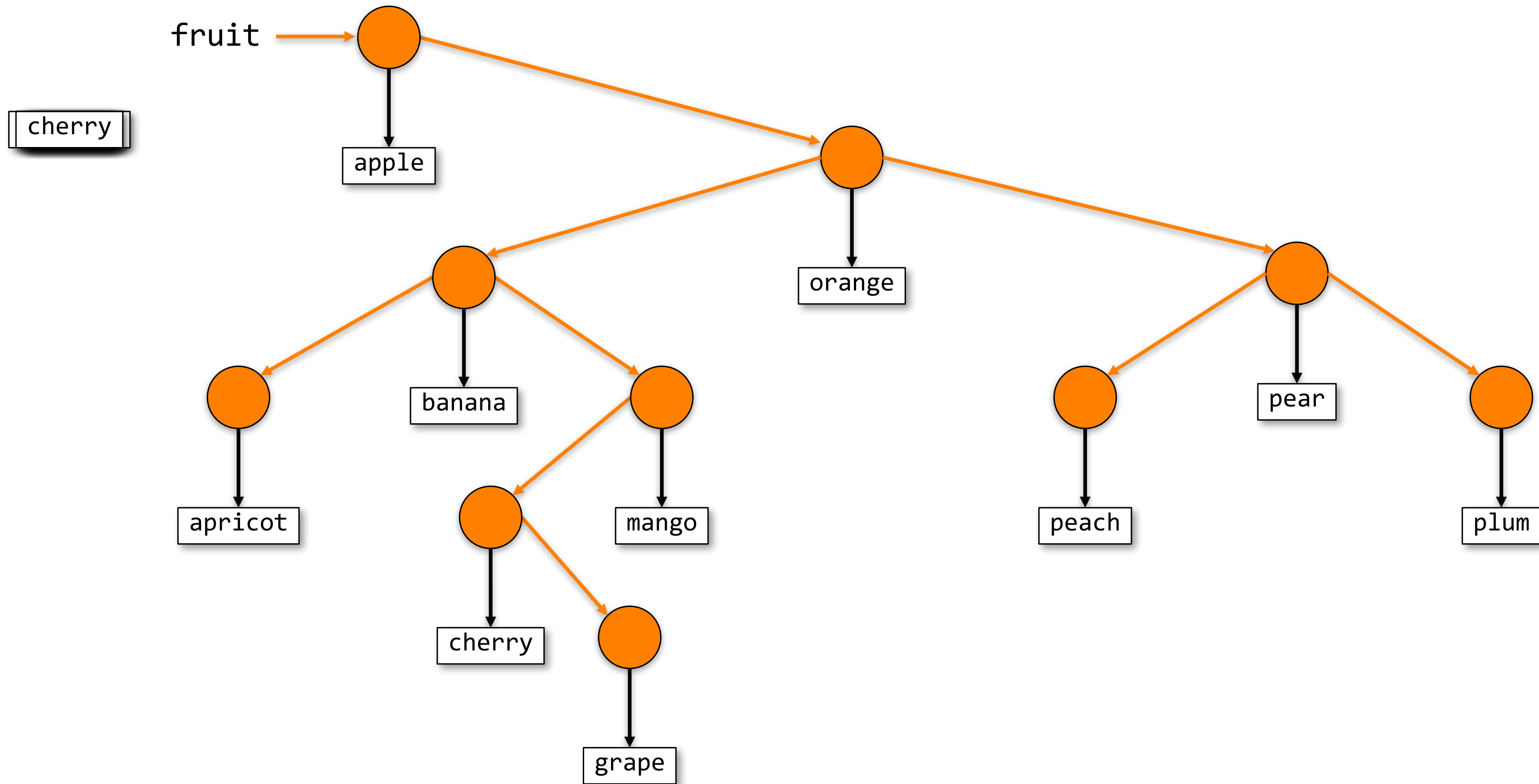
grape

pear

peach

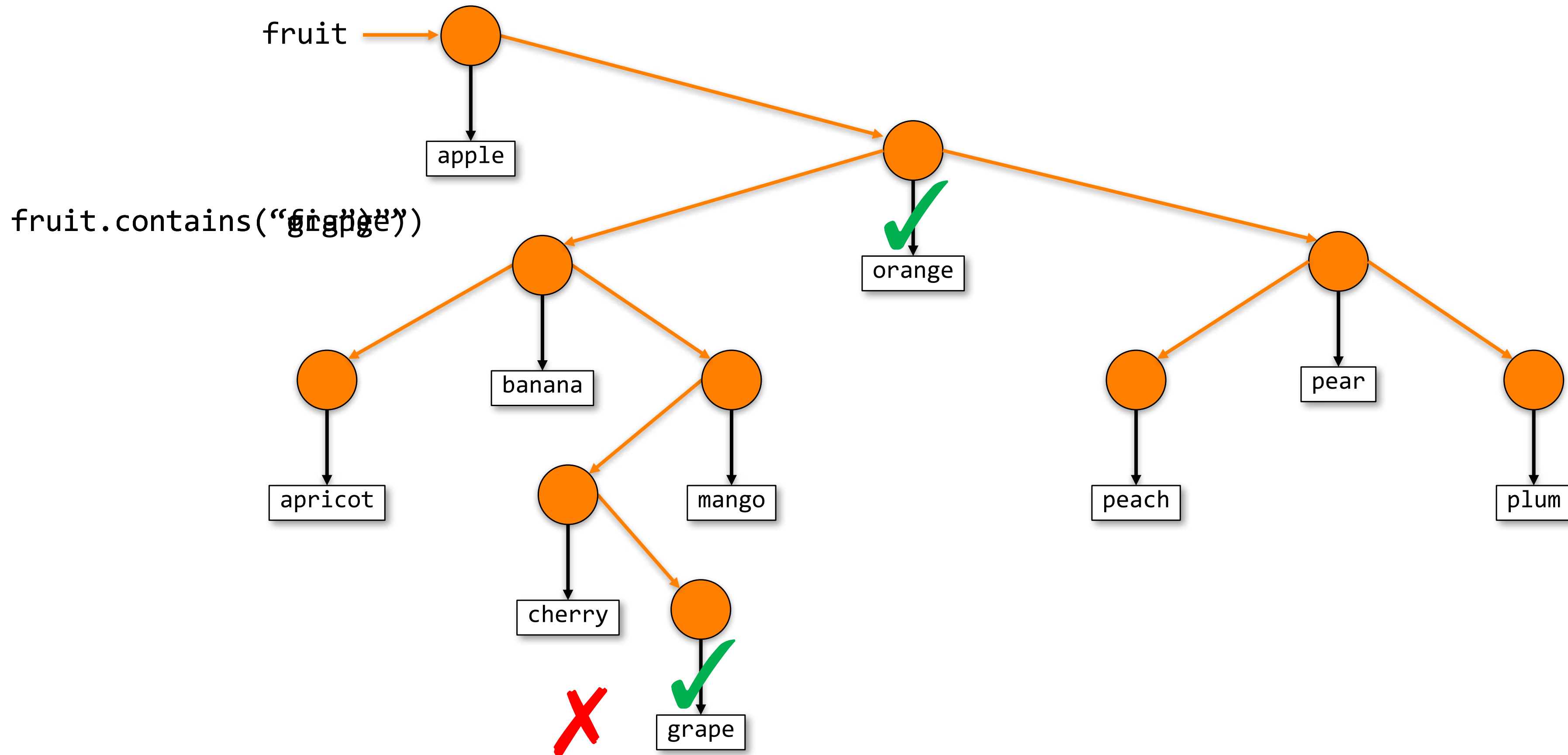
plum

Abstract Data Types: Trees



Australian National University

Abstract Data Types: Trees A5



Australian National University

Abstract Data Types: Maps

A6

The Map ADT

A Map interface and its implementation

ADT Recap

ADT Recap

First-principles implementation of three Java container types:

- List
  - ArrayList, LinkedList implementations *(A1, A2)*
- Set
  - HashSet, BSTSet implementations *(A3, A4, A5)*
- Map
  - HashMap, BSTMap implementations *(A6)*

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

The Map ADT (A.K.A. Associative Array)

A map consists of (key, value) pairs

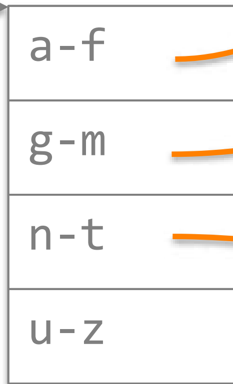
- Each key may occur only once in the map
- Values are retrieved from the map via the key
- Values may be modified
- Key, value pairs may be removed

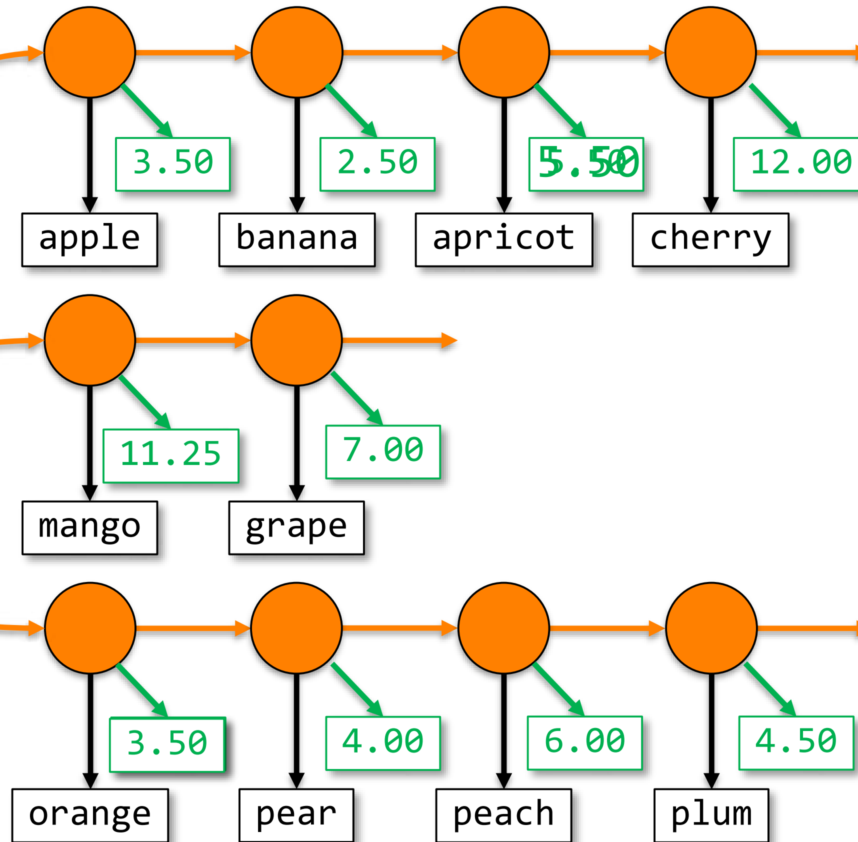
Our Map Interface

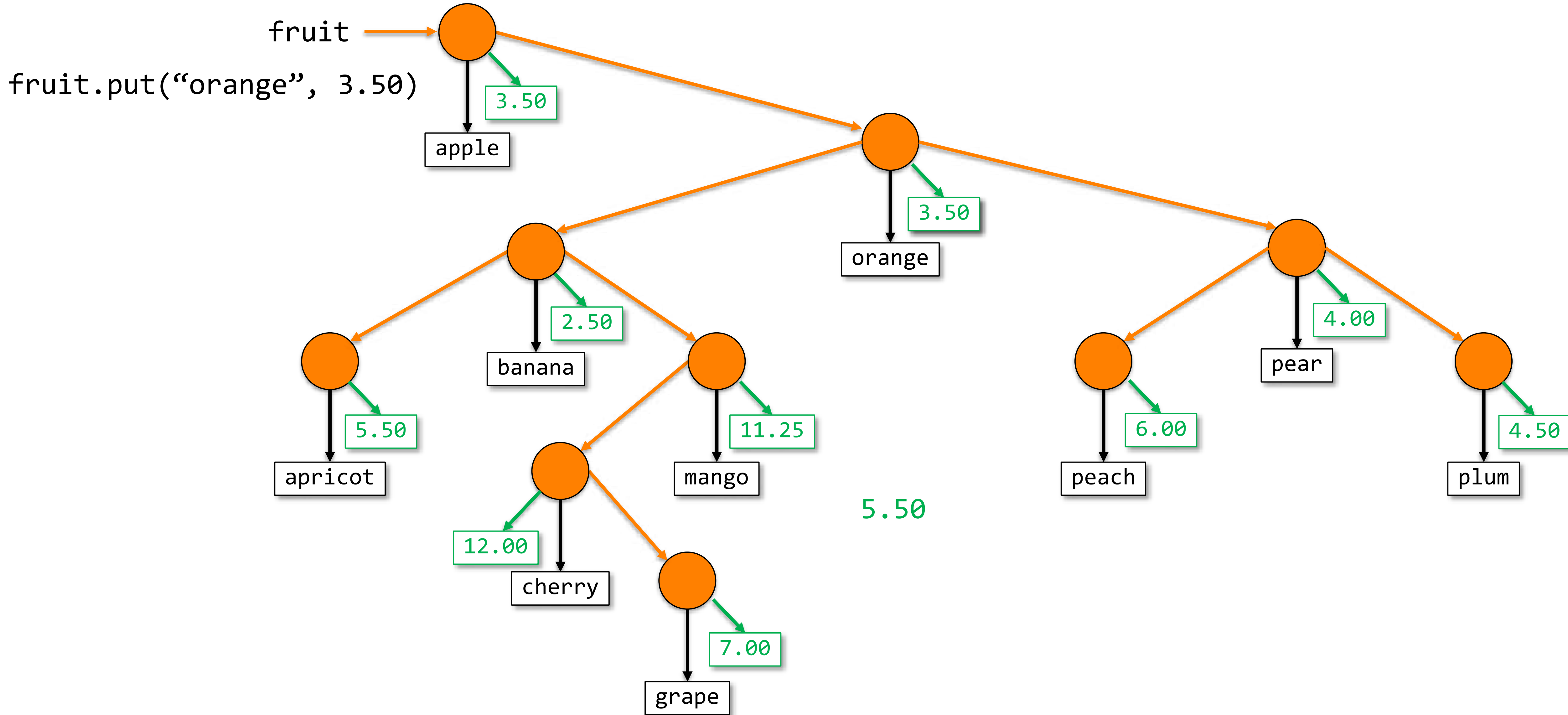
We will explore maps using a simple interface:

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

```
fruit.put("orange", 3.50)
```






fruit
fruit.put("orange", 3.50)
3.50
apple
3.50
orange
2.50
banana
4.00
pear
5.50
apricot
11.25
mango
6.00
peach
4.50
plum
5.50
12.00
cherry
7.00
grape