A04 Sets: HashSet

Hash tables
A hash-table-based Set implementation

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

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.

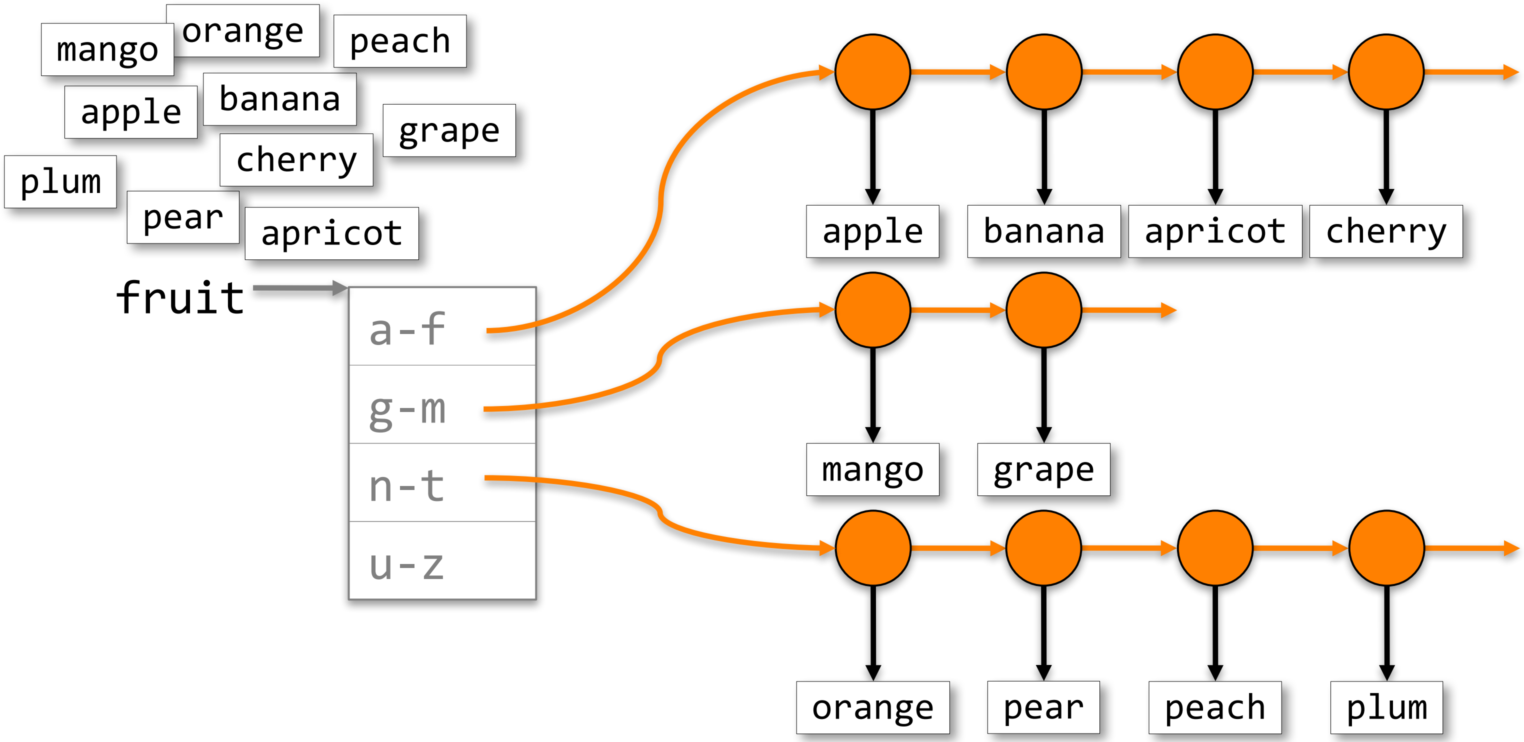
Main challenges are: a) dealing with **hash collisions** and dealing with **load** (how big to make the table).

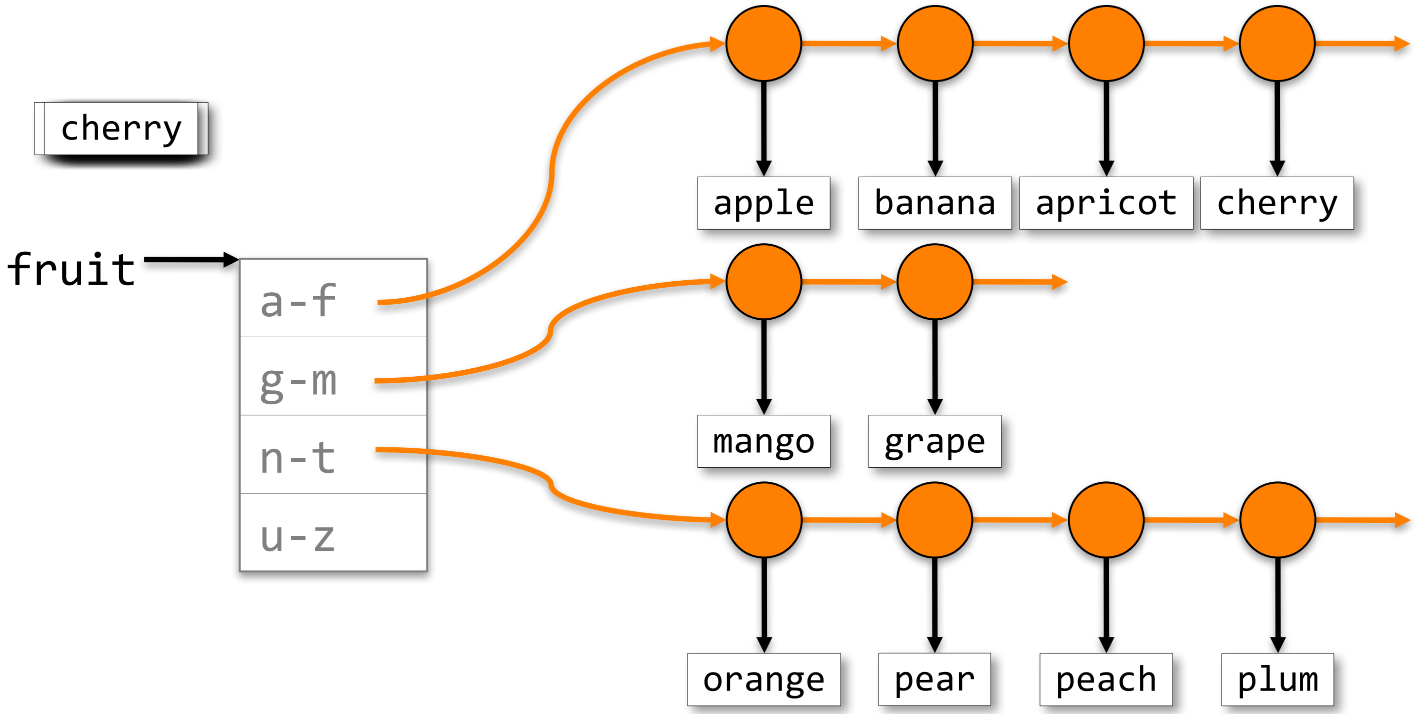
Two broad approaches:

- 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

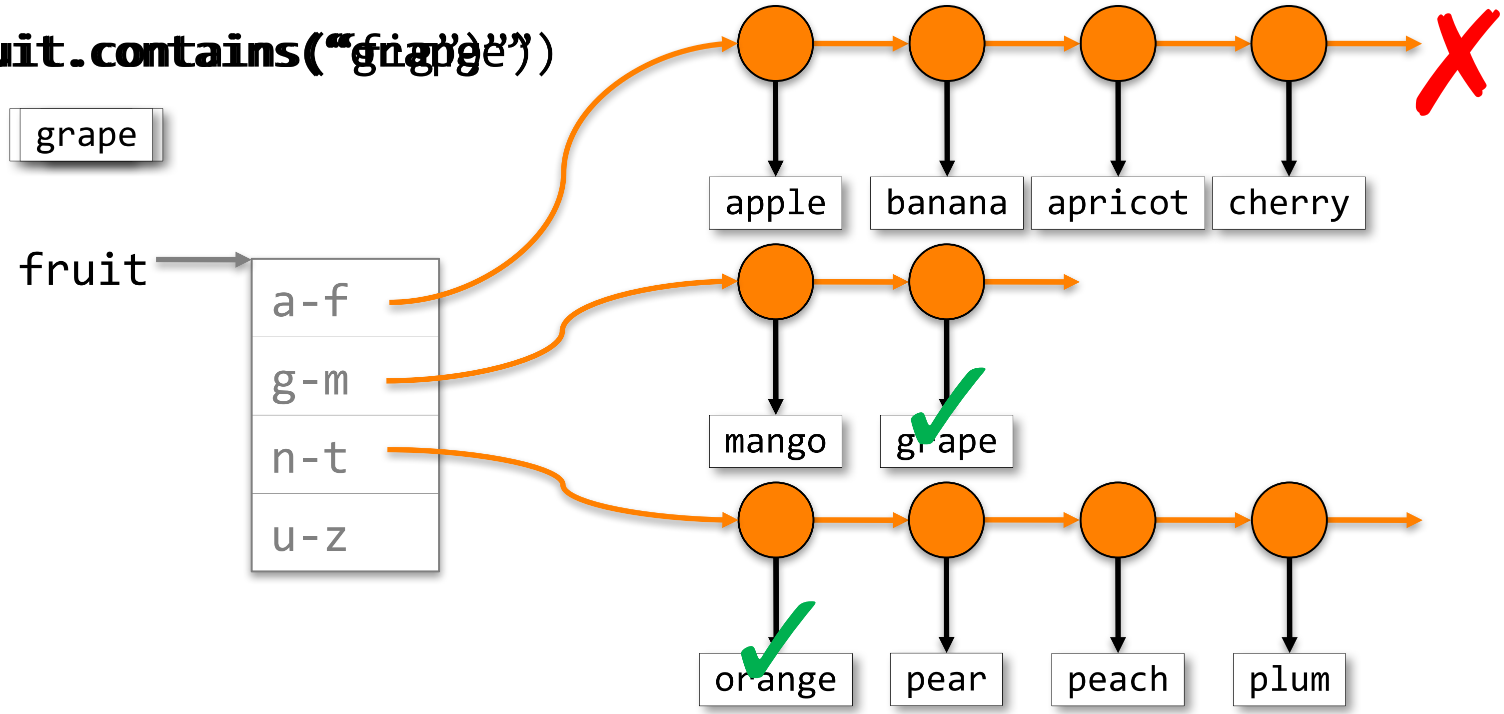
HashSet Implementation of a Set

- 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")



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).

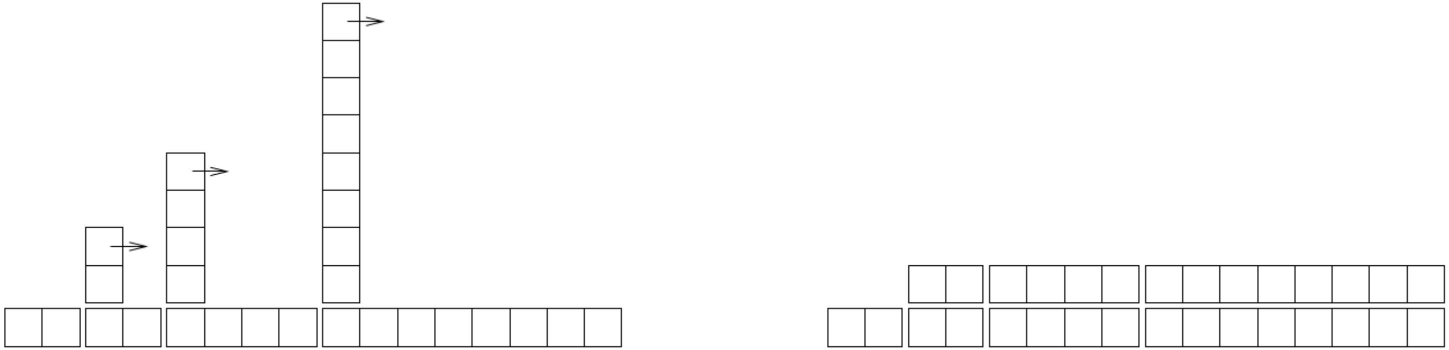


Figure B.1: The cost of a hashtable add.

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

Complexity

```java
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)