Dictionaries

Dictionary ADT

- Stores associations between keys and items or records (also called associative map)
- Operations :

(a) insertElement(k,e) – insert association between a key and item; replace if necessary. Note there are no duplicate items for a key.

(b) removeElement(k) – remove association between key k and its element if it exists, else return NO_SUCH_KEY error

(c) findElement(k) – find the element associated with key k if it exists, else return EMPTY_ELEMENT

- Keys are from a set which may or may not be totally ordered but keys checked for "equality".

Map Interface

Method	Behavior
`V get(Object key)`	Returns the value associated with the specified `key`. Returns **`null`** if the `key` is not present.
`boolean isEmpty()`	Returns **`true`** if this map contains no `key-value` mappings.
`V put(K key, V value)`	Associates the specified `value` with the specified `key` in this map (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`V remove(Object key)`	Removes the mapping for this `key` from this map if it is present (optional operation). Returns the previous `value` associated with the specified `key`, or **`null`** if there was no mapping for the `key`.
`int size()`	Returns the number of `key-value` mappings in this map.

Dictionary implementation using arrays

- Store key-value pairs in an array with new key-value pairs inserted at the end.

  -insertElement(k,e) – O(1) time worst-case

  - removeElement(k) – O(n) time worst-case

  - findElement(k) – O(n) time worst-case

Dictionary implementation- Hash table

- Hash tables

-- Define a function $f : S \rightarrow \{0,1, \ldots N-1\}$ where S is the set of keys and N is the table size.

--- Use this hash function to identify the index in the table where the key is stored. (similar to direct access in an array)

--- Since it is not $1-1$ function, more than one key may map into the same index in the table causing collision.

-- Ideally f should distribute keys evenly in the table.

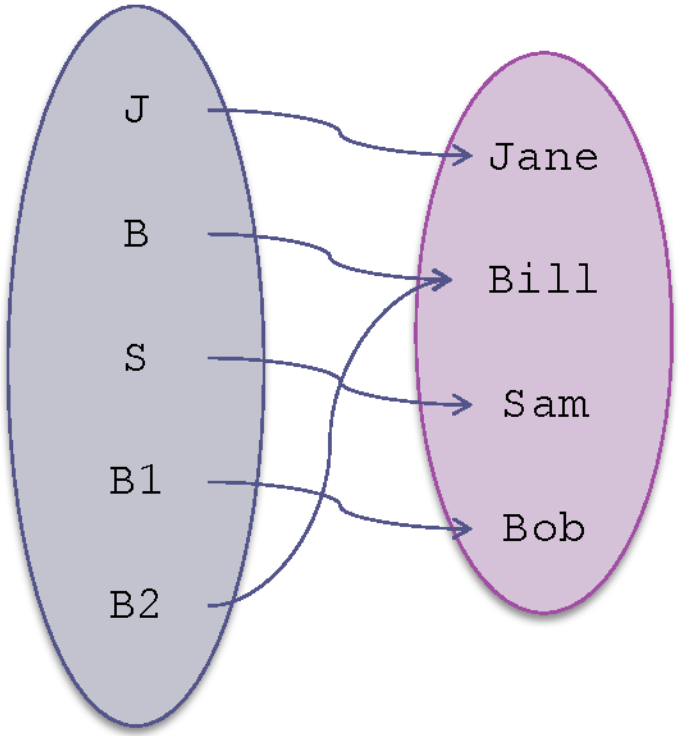
- Java hash map uses this implementation.

Map Interface (cont.)

- The following statements build a Map object:

```
Map<String, String> aMap =
   new HashMap<String,
   String>();
```

```
aMap.put("J", "Jane");
aMap.put("B", "Bill");
aMap.put("S", "Sam");
aMap.put("B1", "Bob");
aMap.put("B2", "Bill");
```

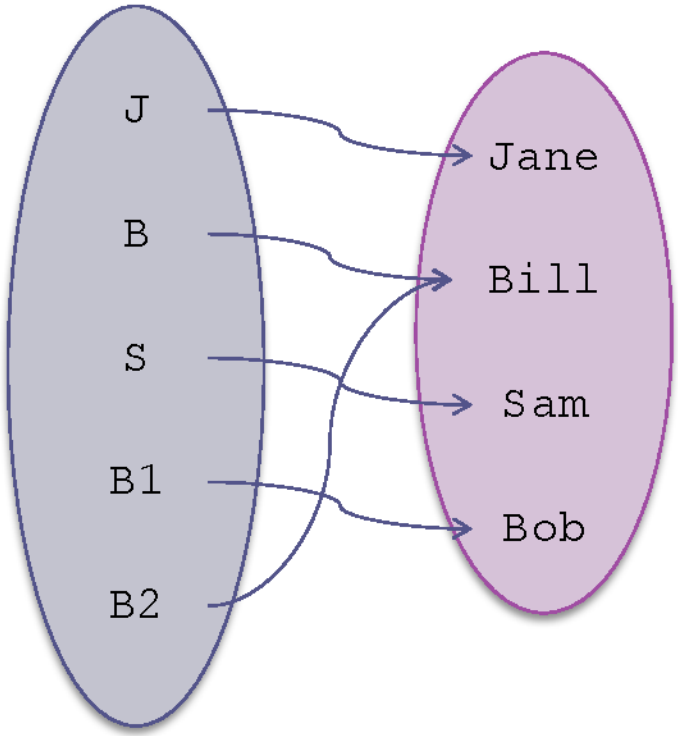


Map **Interface** (cont.)


```
aMap.get("B1")
```

returns:

```
"Bob"
```



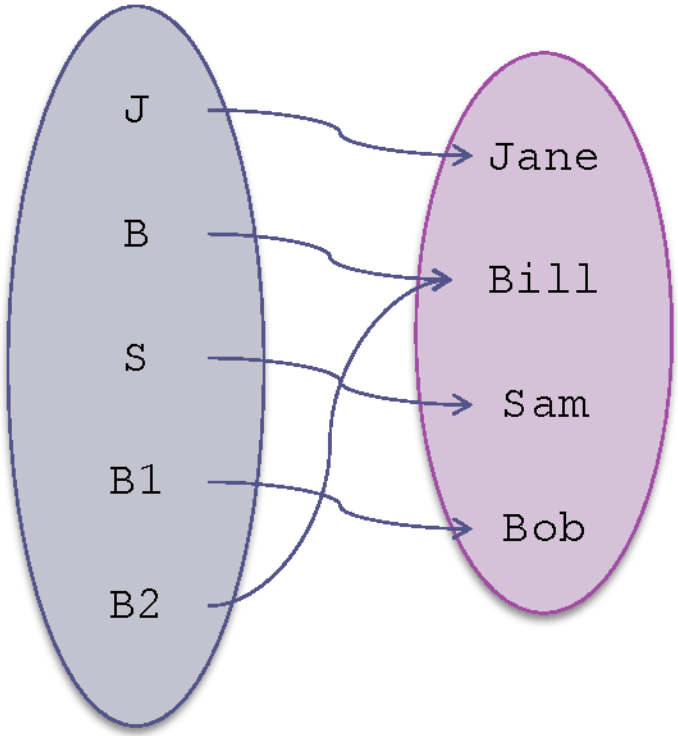
Map **Interface** (cont.)


```
aMap.get("Bill")
```

returns:

```
null
```

("Bill" is a value, not a key)



Hash table time complexity

- When there is collision (i.e. 2 keys mapped to same location in hash table), how do you resolve it ?

  - Use a chain or a list of key-value pairs in each location of hash table; keys in this list have same hash function value

- Worst case time complexity for insert, remove and find – O(n) where n is number of keys
- With a good hash function, average time complexity for these operations is O(n/N) where N is table size. n/N is ”load factor”
- When n/N is $O(1)$, average time complexity is $O(1)$

Source : Dr. Calvin & textbook