CSE 143

More Collection Types: Sets, Bags, and Tables

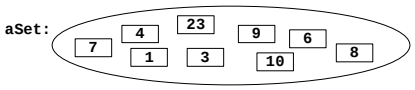
[Chapter 11]

Set ADT

- Attributes of *set* type
  - Nonlinear collection of elements
  - Varying number of elements
  - No duplicates
- Operations on sets
  - Element operations: **`add`**, **`remove`**, **`isMember`**
  - Aggregate operations: **`union`**, **`intersection`**, **`difference`**
  - Possibly associated iterator class

Set Terminology

- *Union*: Create a new set whose elements are those found in either set
- *Intersection*: Create a new set whose elements are those found in both sets
- *Difference*: Remove elements found in one set from other set



Set Interface

```
class IntSet {
  IntSet();
  // standard size(), isFull(), isEmpty() methods

  bool isMember(int item);
  void add(int item);
  void remove(int item);
  friend IntSet union(IntSet a, IntSet B);
  friend IntSet intersection(IntSet a, IntSet B);
  friend IntSet difference(IntSet a, IntSet B);
private:
  ...
};
```


Set Implementation

- We've seen several implementations of sets
  - Unordered array
  - Sorted array
  - Single linked list
    - `Unordered`
    - `Sorted`
  - Double linked list
    - `Unordered`
    - `Sorted`
- Choice of implemented depends on how it will be used -- which operations need to be fast

Bag ADT

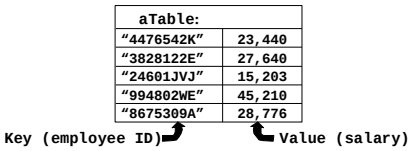
- Attributes of *bag* type -- same as *set* but duplicates allowed
  - Nonlinear collection of elements
  - Varying number of elements
  - May contain duplicate entries
- Operations on sets
  - Element operations: **`add`**, **`remove`**, **`isMember`**
  - Aggregate operations: **`union`**, **`intersection`**, **`difference`**
  - Possibly associated iterator class
- Implementation choices: same as sets

Table ADT

- Attributes of *table* type
  - Set of key/value pairs
  - No duplicate keys
- Operations on tables
  - **`lookup`** value from key
  - **`insert`** value at key
  - **`remove`** key and associated value from table
- Uses:
  - Phone book, class roster, book index, databases
- Also called a dictionary

Table Terminology

- *Key*: Portion of pair used to look up data, like an index (aka *domain* value)
- *Value*: Portion of pair that contains data (aka *range* value)



Example of Operations

```
// phone book example:

Table pb;

pb.insert("Julie", 5552345);
pb.insert("Bob", 3450011);
pb.insert("Bart", 6661212);
int bartsNumber = pb.lookup("Bart");
pb.delete("Bob");
```


Table Interface

```
class Table {
 public:
   Table();        // Create new empty table

   int size();     // = size of table
   bool isEmpty(); // = "table is empty"
   bool isFull();  // = "table is full"
   // return data with given key
   RangeType lookup(DomainType key);
   // store data in table with given key
   void insert(DomainType key, RangeType value);
   // remove data from table with given key
   RangeType remove(DomainType key);

  private:
      . . .
};
```


Table Implementation

- Many different possibilities
  - Array of structs
  - List of structs
  - Dynamic memory structure
- Implementations can also use other techniques
  - Searching, sorting
  - Hash tables
  - Trees
- Implementation details to come...

Container Type Summary

- Stack
  - List with LIFO structure
  - Access via push(item), pop(), and top()
- Queue
  - List with FIFO structure
  - Access via enqueue(item), dequeue(), and front()
- Set
  - Unordered collection, without duplicates
- Bag
  - Unordered collection, without duplicates
- Table
  - Unordered association of key-value pairs
  - Access via insert(key, value), delete(key), lookup(key)