SETS & MAPS.

HASHING

SETS

- Collection that contains no duplicate elements and at most one null element. ; {"apples", "oranges", "pineapples"} → no order!

- Operations:
  - *Testing for membership
  - *Adding elements
  - *Removing elements
  - *Union A∪B
  - *Difference A-B
  - *Subset A⊂B

Union

$\{1,3,5,7\} \cup \{2,3,4,5\} \Rightarrow \{1,2,3,4,5,7\}$

Intersection

$\{1,3,5,7\} \cap \{2,3,4,5\} \Rightarrow \{3,5\}$

Difference.

$\{1,3,5,7\} - \{2,3,4,5\} \Rightarrow \{1,7\}$

$\{2,3,4,5\} - \{1,3,5,7\} \Rightarrow \{2,4\}$

Subset

$\{1,3,5,7\} \subset \{1,2,3,4,5,7\} \Rightarrow$ True

LIST vs. SET

- Sets allow no duplicates
- Sets do not have positions - in opposition to lists
- Sets can give constant operation time for lookup, insert or delete

MAPS

- Map is related to Set; it is a set of ordered pairs
- Ordered pair (key, value)
  - ↳ In a given map, there are no duplicate keys
  - ↳ Values may appear more than once.
- Maps support efficient organization of information in tables
- Mathematically, these maps are:
  - ↳ many-to-one (not necessarily one-to-one)
  - ↳ Onto (every value in the map has a key)

Key ≡ "mapping to" to a particular value

The map functions

- Template parameters
  - *Key_Type: the type of the keys
  - *Value_Type: the type of the values
  - *Compare: the function class that compares the keys
- All functions defined for the set are defined for the map, taking a pair <Key_Type, Value_Type>

- Methods: → v.get(Object Key)
  → v.put(K key, V value)
- When info about an item is stored in table, it should have a unique ID. (is equivalent to a key; may or may not be a number)

HASHTABLES

- **GOAL**: access item given its key (not its position)

  Therefore, we want to locate it directly from the key (we wish to avoid much searching)

- O(1) → in the average case!
- O(n) → in the worst case!

- Searching an array: O(n)
- Searching BBST: O(log n)

Suppose we have a table of size N.

- A hash code is
  - * n° in range 0 to N-1
  - * We compute hash code from the key
  - * You can think of this as a "default position" when inserting, or a "position hint" when looking up.

- A hash function is a way of computing a hash code
- Desire: the set of keys should spread evenly over the N values
- When 2 keys have the same hash code: collision

- Good hash functions
  - Spread values evenly (as if random)
  - Cheap to compute

- Generally, n° of possible values >> table size.

We'll consider 2 ways to organize hash tables
- Open addressing and probing
- Separate Chaining

Open addressing & probing

- Hashed items are in a single array
- Hash code gives position hint
- Handle collisions by checking multiple positions
- Each check is called a probe of the table

→ Linear Probing (simplest approach)

- Probe by incrementing the index
- If "fall off end", wrap around to the beginning
  (careful with infinite loops)

1. Compute index as hash_fcn() % table.size()
2. if table[index] == NULL, item is not in the table
3. if table[index] matches item, found item (done)
4. Increment index circularly and go to 2

Why must we probe repeatedly?
↳ hashCode may produce collisions

→ Search Termination

- Ways to obtain proper termination:
  - ↳ Stop when you come back to your starting point.
  - ↳ Stop after probing N slots, where N is table size.
  - ↳ Stop when you reach the bottom the second time
  - ↳ Ensure table never full
    - → Reallocate when occupancy exceeds threshold

→ Hash Table Considerations

- Cannot traverse a hash table → Order of stored values is arbitrary
  → Can use an iterator to produce in arbitrary order
- When item is deleted, cannot just set its entry to null → Doing so would break probing
  → Must store a "dummy value" instead
  → Deleted items waste space and reduce efficiency.
- Higher occupancy causes makes for collisions

*Note: many lookups will probe a lot!
*Size 11 gives these slots: 3,5,10,5 → 6,7

HASH TABLE EXAMPLE

- Table of strings, initial size 5
  - Add "Tom", hash 84274 → 4 ⇒ Slot 4
  - Add "Dick", hash 2129869 → 4 ⇒ Slot 0 (wraps)
  - Add "Harry", hash 69496448 → 3 ⇒ Slot 3
  - Add "Sam", hash 82879 → 4 ⇒ Slot 1 (wraps)
  - Add "Pete", hash 2484038 → 3 ⇒ Slot 2 (wraps)

0	1	2	3	4
Dick	Sam	Pete	Harry	Tom

→ Reducing Collisions by Growing: REHASHING

- Choose a new larger size, e.g. doubling
- (Re) insert non-deleted items into new array
- Install the new array and drop the old
- Similar to reallocating a vector
  ↳ But, elements can move around in reinsertion
  ↳ Rehashing distributes items at least as well

→ Quadratic Probing

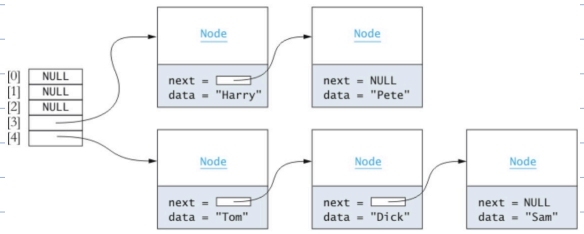
- Linear Probing
  ↳ Tends to form long clusters of keys in the table
  ↳ This causes longer search chains
- Quadratic probing can reduce the effect of clustering
  ↳ Index increments form a quadratic series
  ↳ Direct calculation involves multiply, add, remainder
    - Incremental calculation better
  ↳ Probe sequence may not produce all table slots
- Generating the quadratic sequence
  want: $s, s+1^2, s+2^2, s+3^2, s+4^2$, etc. (all % length)
  "Trick" to calculate incrementally:
  Initially:
  size_t index = ... 1st probe slot ...
  int k = -1;
  At each iteration:
  k += 2;
  index = (index + k) % table.size()

→ Double hashing

- Another way to deal with conflicts is double hashing
- When a collision happens the next index is found by applying a second function on the key and add this value to the original hash code: $hash_1\_fcn(i) + hash_2\_fcn(i)$
- If this also produces a conflict 2 times, the second function is added and if that also produces a conflict, then 3 time, etc.

→ Separate chaining

- Alternative to open addressing and probing
- Each table slot references a linked list
  - ↳ List contains all items that hash to that slot
  - ↳ The linked list is often called a bucket
  - ↳ So sometimes called bucket hashing



Two items hashed to bucket 3
Three items hashed to bucket 4

- Examines only items with same hash code
- Insertion about as complex
- Deletion is simpler
- Linked list can become long → rehash

Performance of Hash Tables

- Load factor = # filled cells / table size (Between 0 and 1 for open addressing)
  - ↳ has greatest effect on performance
- Lower load factor: reduce collisions in sparsely populated tables ⇒ Better performance
- Expected # probes p for open addressing, linear probing, load factor L: $p = \frac{1}{2}(1 + 1/(1-L))$
  - ↳ As L approaches 1, this zoomes up
- For chaining, $p = 1 + (L/2)$
  - ↳ NOTE: here L can be greater than 1!

- Hash Table
  - ↳ Insert: average O(1)
  - ↳ Search: average O(1)
- Sorted array
  - ↳ Insert: average O(n)
  - ↳ Search: average O(log n)
- Binary Search Tree
  - ↳ Insert: average O(log n)
  - ↳ Search: average O(log n)
- A balance tree can guarantee O(log n)