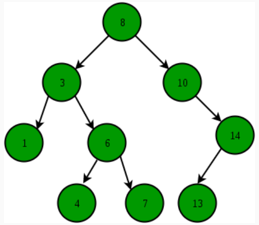
CSCI 104

Week 9, Lab 14: Hashtables

Remember: Maps

- A data structure for fast look-ups
- Holds <key, value> pairs, where keys are unique
- Look-up speed and key ordering depends on how we implement the map!

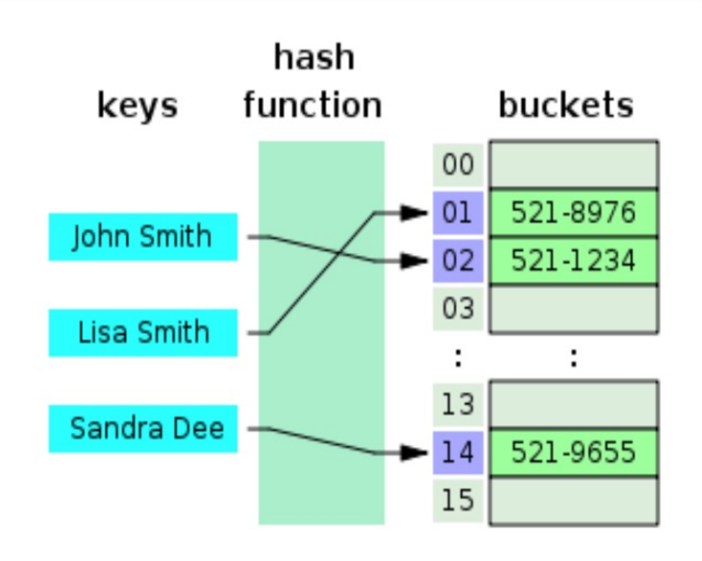


- ❑ Balanced BSTs
- ❑ O(logn) operations
- ❑ Ordered keys!

- ❑ Today's Focus
- ❑ Hashtables

Hashtables

- Another way of implementing maps
- Like an array!
- Every *key* is converted into an index location using a 'hash function'
- *Value* is stored at that index
- What's the advantage?
    - If the hash function is good, this makes operations O(1) on average!!



Hash Functions

- What does it mean to have good hash function?
  1. Fast and easy to compute
  2. Uniformly distributes keys across the array

```
int hash(int data) {
    return 42 % size;
}
```

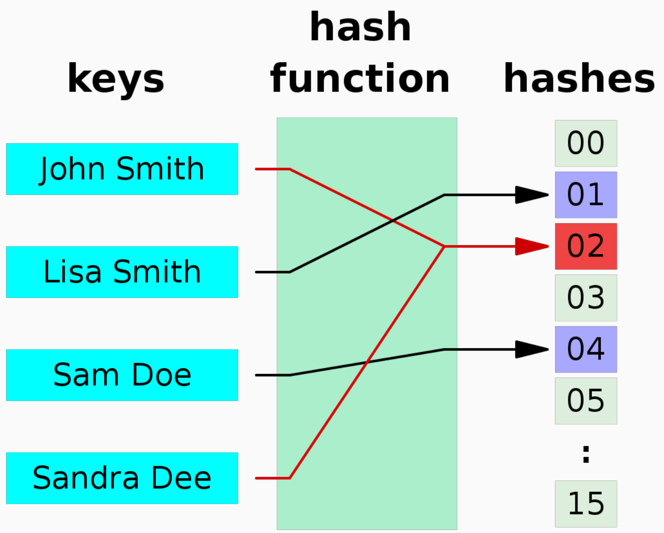
- ❑ Fast to compute
- ❑ Non-uniform distribution

```
int hash(int data){
    return 31 * 54059 ^ (data * 76963) % size;
}
```

- ❑ Fast to compute
- ❑ Also distributes keys more uniformly!

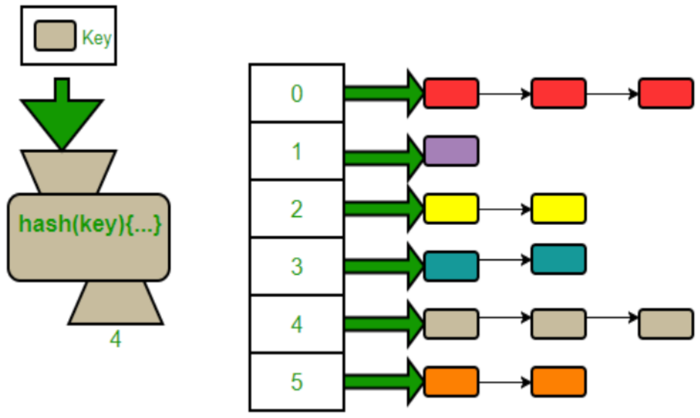
Collisions

- Collisions happen when different keys map to the same index in the hash table!
- How can we handle these?
- Two ways:
  - Close addressing (hash value fully determines the location)
  - Open addressing (location may change if collisions happen)



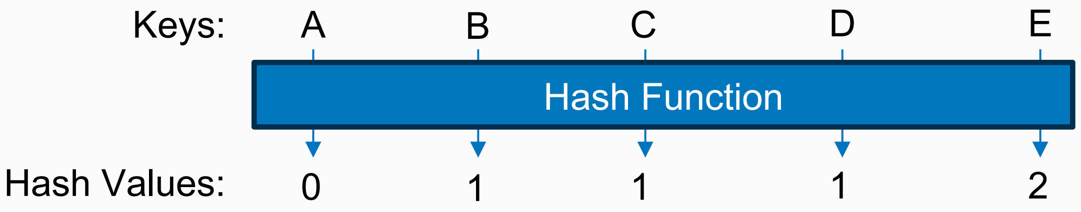
Chaining

- Close addressing (like using buckets)
- Multiple items at every index
  - As computed by the hash function
- New items for that location gets chained to that location
  - Using linked lists
  - Or other structures (like a Balanced BST)
- How can this go wrong?



Open Addressing

- Location of the item changes if there's a collision
    - Linear probing, quadratic probing, double hashing etc.
- Only one item per hashtable index!
- Linear Probing:
    - Compute hash value for key
    - Look at the next index until you find the key you are looking for
- Example:



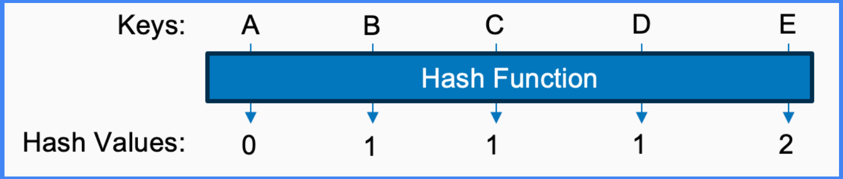
Linear Probing

Inserting A Key

1. Hash the element, which gives you a position in the table.
2. If the position is already taken, try the next position (if you reach the end of the table, wrap back to the start).
3. Repeat step 2 until you have found an empty spot.
4. Insert the element at that empty spot.

Removing A Key

1. First find the position of the key with the method illustrated above.
2. Delete the key there.
3. Move to the next position. If it is not empty, delete the key there and reinsert it.
4. Repeat step 3 until you have reached an empty spot or you have looped back to the position you found in step 1.



	Insert B	Insert C	Insert E	Insert A	Insert D	Remove B	Remove E
0				A	A	A	A
1	B	B	B	B	B	C	C
2		C	C	C	C	E	D
3			E	E	E	D	
4					D		

Re-inserted

The Lab

- Follow the bytes page on the lab
- Implement an unordered set with linear probing for string type keys
  - Specifically, the remove function
- Hash function is already implemented
- Run “make” to compile your program and then run the executable
- Need to pass all tests to get checked off OR be working throughout the whole lab section
- Also, check out the bonus test!