CS 106B, Lecture 27
Hashing

小结：

hashing的idea和概念

collision（不同value的hash value相同）及其解决方案：separate chaining

rehashing：当前储存容器过密（load factor过高）：增长hash table，打散separate chain

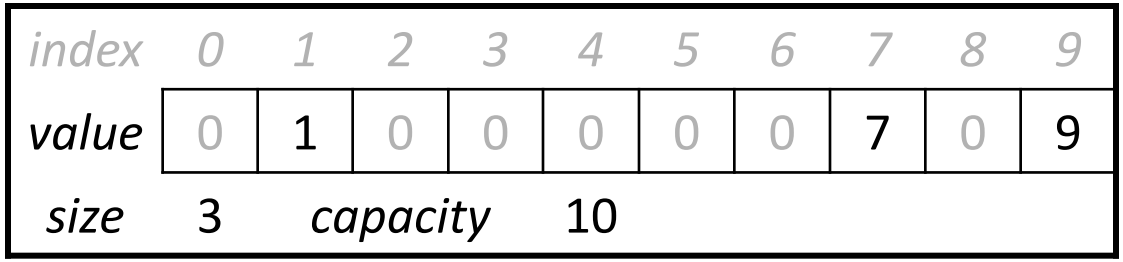
好的hash function特征

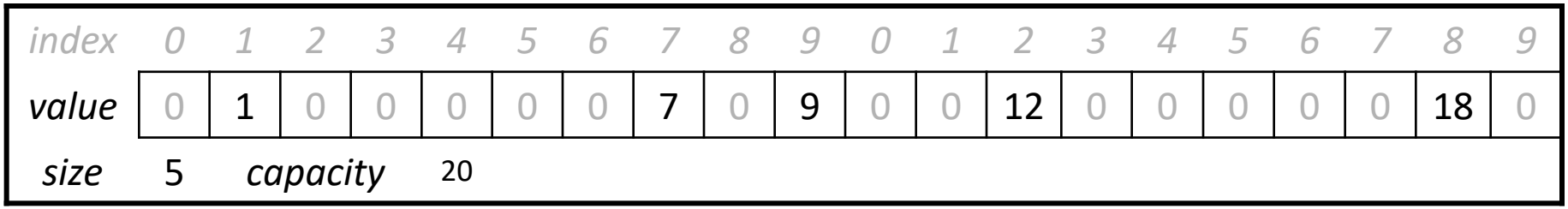
A strange idea

- *Silly idea:* When client adds value `i`, store it at index `i` in the array.
  - Would this work?
  - Problems / drawbacks of this approach? How to work around them?

```
set.add(7);
set.add(1);
set.add(9);
...

set.add(18);
set.add(12);
```



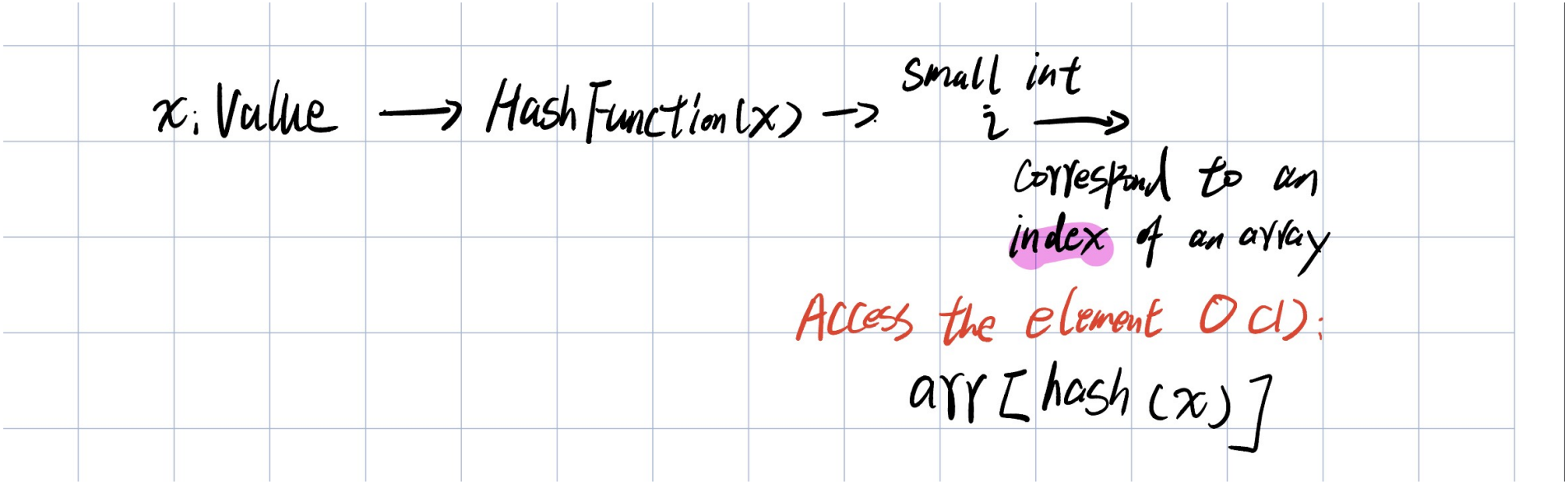


Hashing (15.3)

- **hash**: To map a large domain of values to a smaller fixed domain.
  - *Idea:* Store any given element value in a particular predictable index.
  - **hash table**: An array that stores elements via hashing.
  - **hash function**: An algorithm that maps values to indexes.
  - **hash code**: The output of a hash function for a given value.

  - In previous slide, our "hash function" was: **hashCode(i) → i**.
    Drawbacks?

idea of hashing

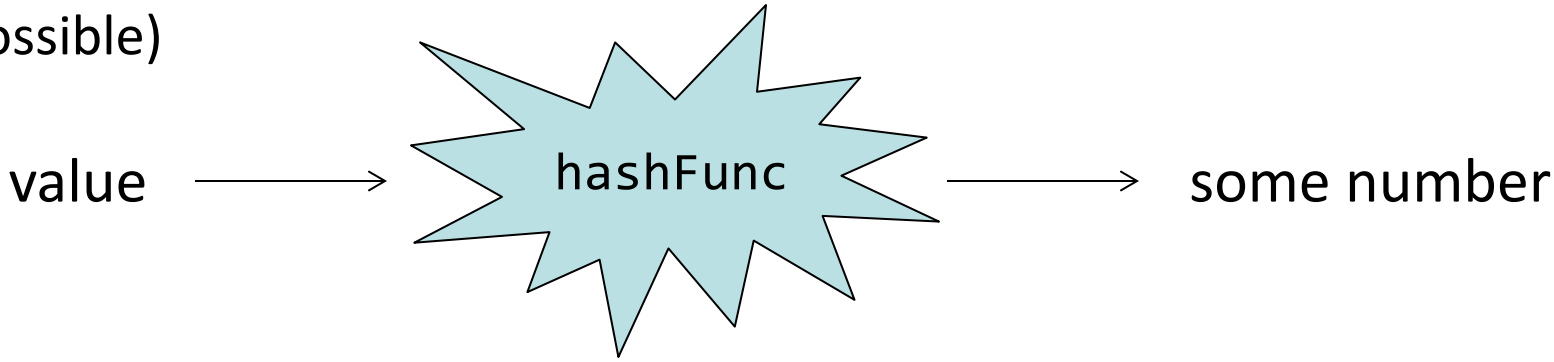


Hash Functions

- **hash function**: function of the form

```
int hashFunc(Type arg);
```

  - must be **deterministic** (same input produces the same output)
  - should be **well-distributed** (the numbers produced are as spread out as possible)



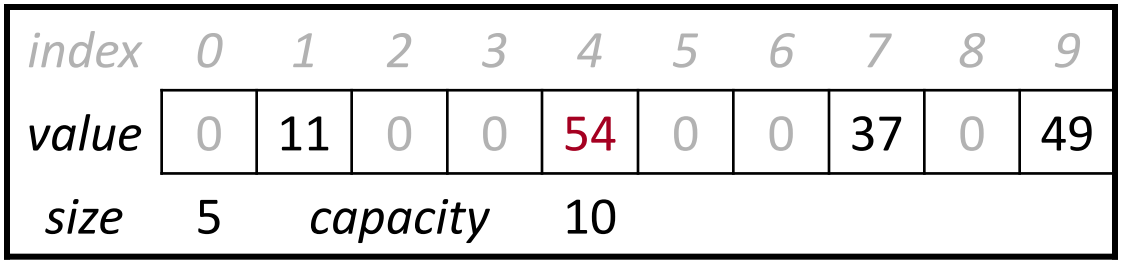
- *Idea:* Store any given element value in the index given by the hash function (why hash functions must be **consistent**)
  - In previous slide, our (bad) "hash function" was: **hashCode(i) → i**.
  - Drawbacks?
    - Potentially requires a large array (array capacity > i).
    - Array could be very sparse, mostly empty (memory waste).

Collisions

- **collision**: When a hash function maps 2 values to same index.

```
// hashCode = abs(i)
```

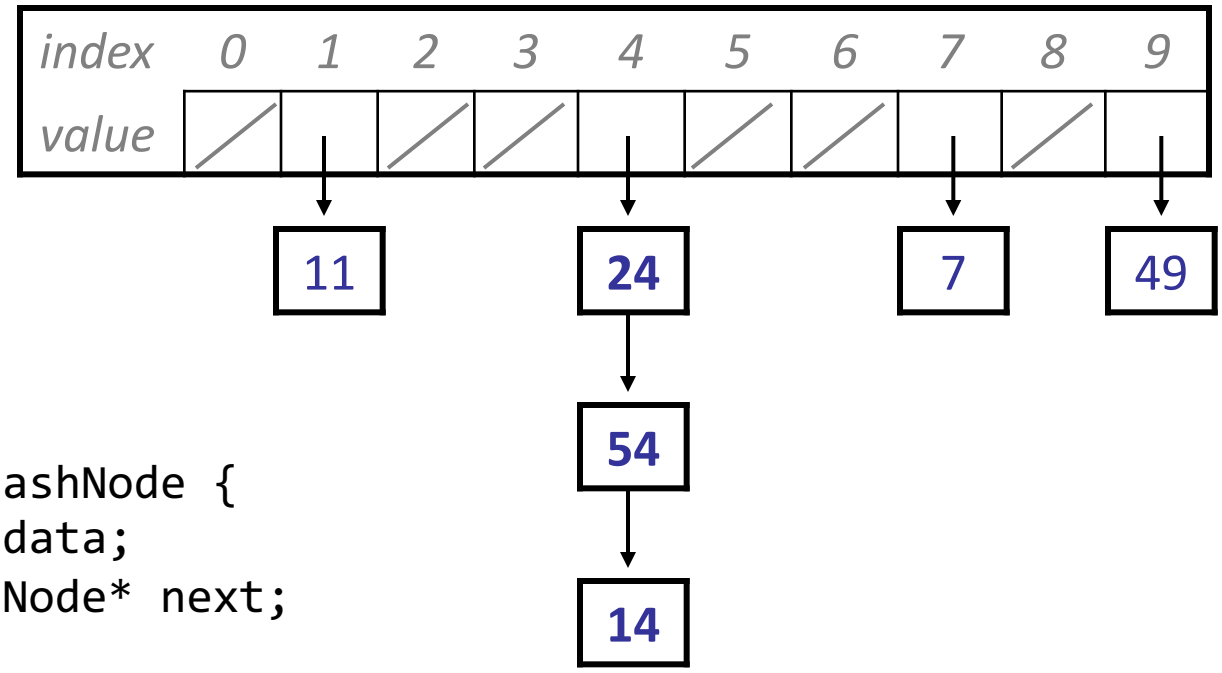
```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);   // collides with 24  :-(
```



- **collision resolution**: An algorithm for fixing collisions.
- A hash function should be **well-distributed** to minimize collisions.

Separate chaining

- **separate chaining**: Solving collisions by storing a list at each index.
  - add/search/remove must traverse lists, but the lists are short
  - impossible to "run out" of indexes, unlike with probing




```
struct HashNode {
    int data;
    HashNode* next;
};
```

Load Factor

- **Load factor:** the average number of values stored in a single index.

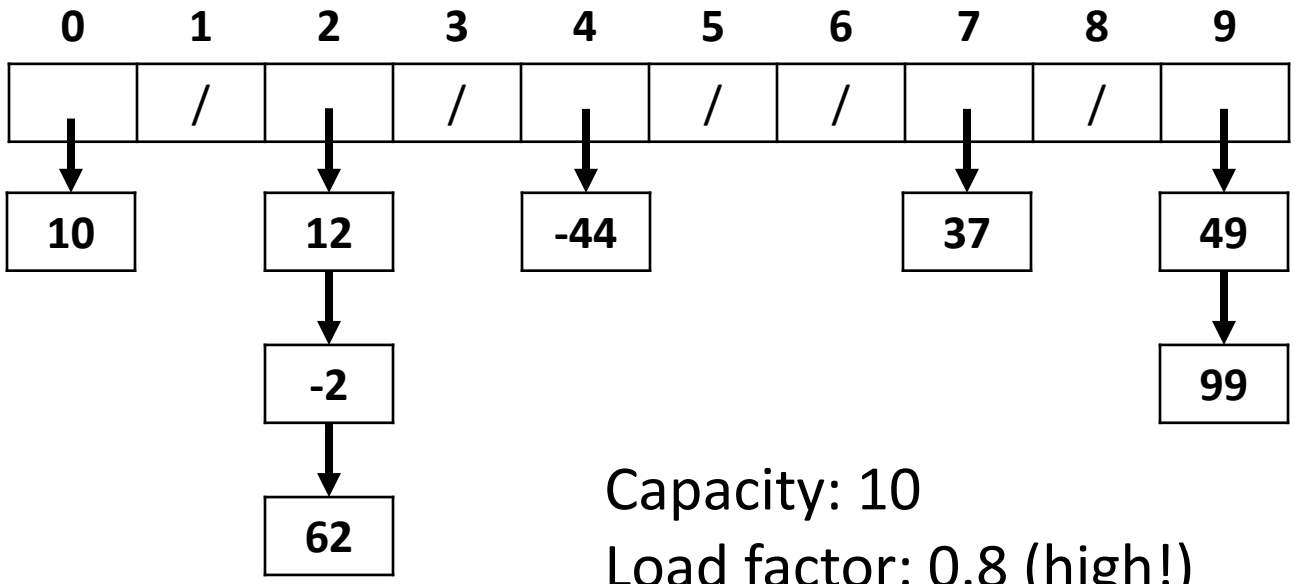
$$load\ factor = \frac{total\ \#\ entries}{total\ \#\ indices}$$

- A lower load factor means better runtime.

- Need to **rehash** after exceeding a certain load factor.
  - Generally after load factor >= 0.75.

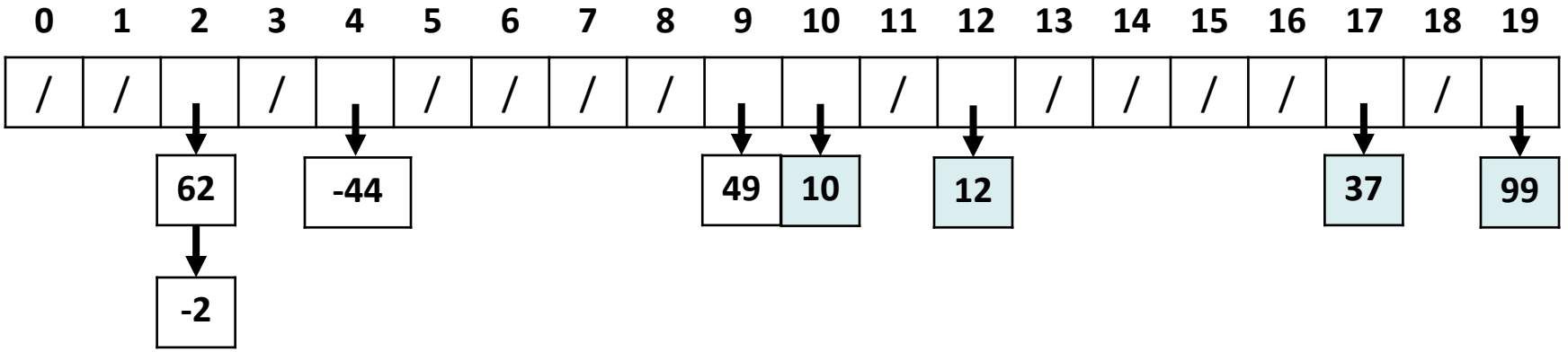
Rehashing

- **Rehashing:** growing the hash table when the load factor gets too high.
  - Can't just copy the old array to the first few indices of a larger one (why not?)



Rehashing

- **Rehashing:** growing the hash table when the load factor gets too high.
  - Loop through lists and re-add elements into new hash table
  - Blue elements are ones that moved indices



Capacity: 20
Load factor: 0.4 (better!)

Hash function properties

- REQUIRED: a hash function must be **consistent.**
  - Consistent with itself:
    - **hashCode(A) == hashCode(A)** as long as A doesn't change
  - Consistent with equality:
    - **If A == B,** then **hashCode(A) == hashCode(B)**
    - Note that A != B doesn't necessarily mean that hashCode(A) != hashCode(B)

- DESIRABLE: a hash function should be **well-distributed.**
  - A good hash function minimizes collisions by returning mostly unique hash codes for different values.