CS 106B, Lecture 27
Hashing

Plan for Today

- Implementing the last data structure of CS106B: a HashMap/HashSet
  - What is hashing?
  - How can we achieve the O(1) add, remove, contains of a HashSet?

Implementing a set

- Consider implementing a set as an unfilled array.
  - What would make a good ordering for the elements?

- If we store them in the **next available index**, as in a vector, ...

```
set.add(9);
set.add(23);
set.add(8);
...
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	9	23	8	-3	49	12	0	0	0	0
size	6		*capacity*		10					

  - How efficient is add? `contains`? `remove`?
    - O(1), O(*N*), O(*N*)

Sorted array set

- Suppose we store the elements in an unfilled array, but in **sorted** order rather than order of insertion.

```
set.add(9);
set.add(23);
set.add(8);
set.add(-3);
set.add(49);
set.add(12);
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	-3	8	9	12	23	49	0	0	0	0
size	6	*capacity*			10					

– How efficient is add? `contains`? `remove`?
  - $O(N)$, $O(\log N)$, $O(N)$

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);
```

index	0	1	2	3	4	5	6	7	8	9
value	0	1	0	0	0	0	0	7	0	9

size 3 capacity 10

index	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
value	0	1	0	0	0	0	0	7	0	9	0	0	12	0	0	0	0	0	18	0

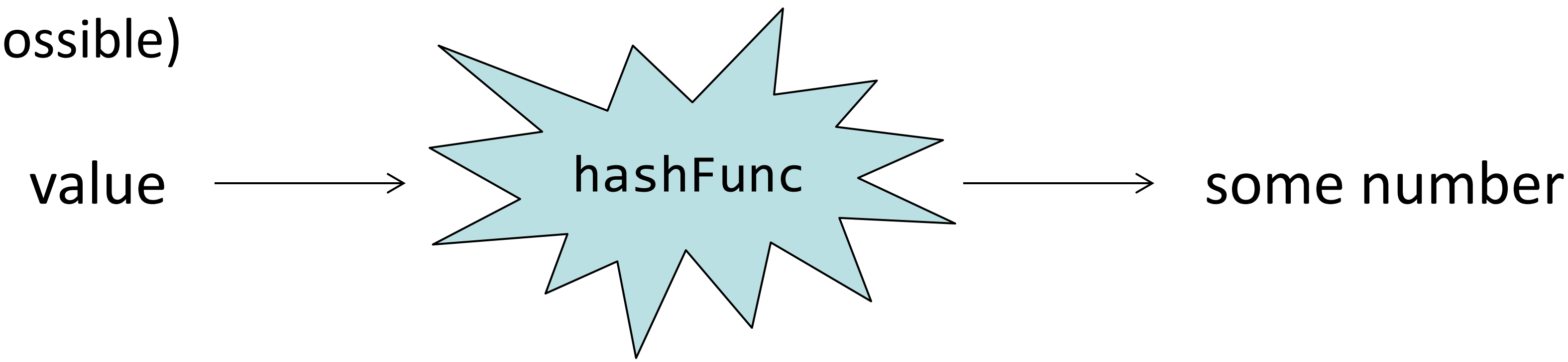
size 5 capacity 20

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).

Improving Space Efficiency

- If any number is equally possible, we'll need a huge array, even if we only have a couple of buckets
- Idea: use a hash function, but modify the result to be within a much smaller range (the size of the array)
- We can then think of the array as a sequence of **buckets** storing elements

```
int getIndex(Type value) {
    return hashCode(value) % capacity;
}
```

Efficiency of hashing

```
int getIndex(int i) {
    return hashCode(i) % capacity;
}
```

- add: `elements[getIndex(i)] = i;`
- contains: `if (elements[getIndex(i)] == i) { ... }`
- remove: `elements[getIndex(i)] = 0;`

- **Q:** What is the runtime of `add`, `contains`, and `remove`?
  A. O(1) **B.** O(log N) **C.** O(N) **D.** O(N log N) **E.** O(N^2)

- Are there any problems with this approach?

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  :-(
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	54	0	0	37	0	49
size	5	*capacity*			10					

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

Probing

- **probing**: Resolving a collision by moving to another index.
  - **linear probing**: Moves to the next available index (wraps if needed).

```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);   // collides with 24; must probe
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	0	37	0	49
size	5	*capacity*			10					

  - **quadratic probing**: a variation that moves increasingly far away:
    - index +1, +4, +9, ...
  - Drawbacks of probing? How does this change `add`, `contains`, etc.?

Clustering

- **clustering**: Clumps of elements at neighboring indexes.
  – slows down the hash table lookup; you must loop through them.

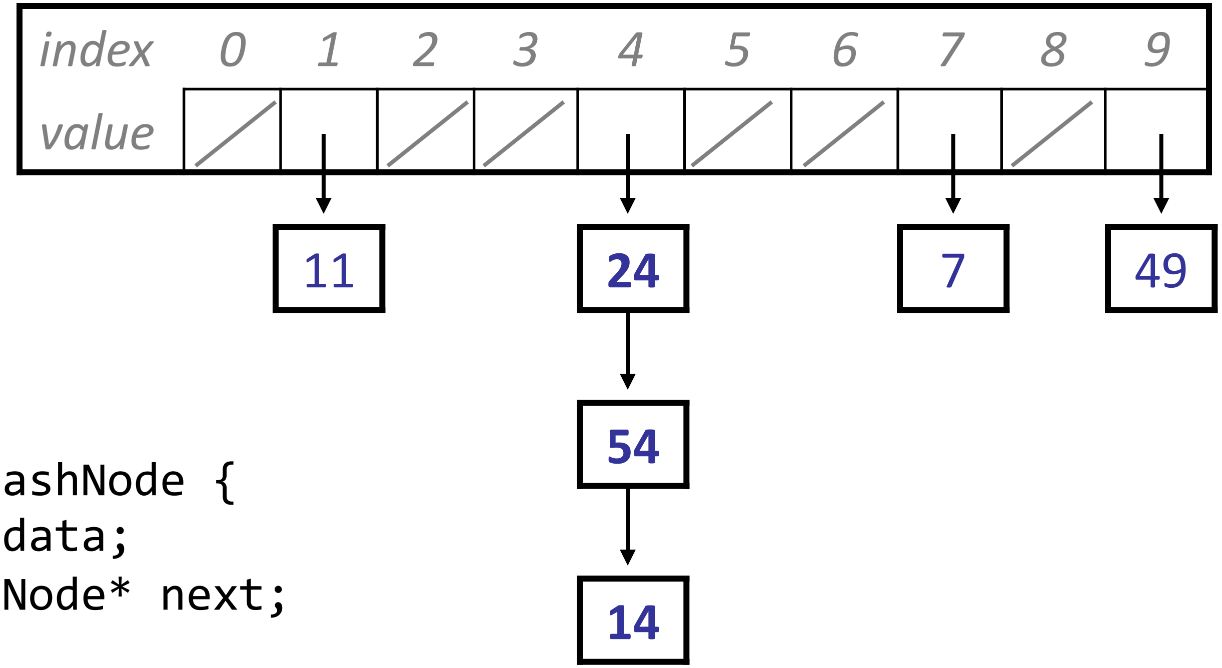
```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);   // collides with 24
set.add(14);   // collides with 24, then 54
set.add(86);   // collides with 14, then 37
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	54	14	37	86	49
size	5	*capacity*			10					

  – A lookup for 94 must look at 7 out of 10 total indexes.
  – Must have a special value for **removed** elements (tombstones).

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

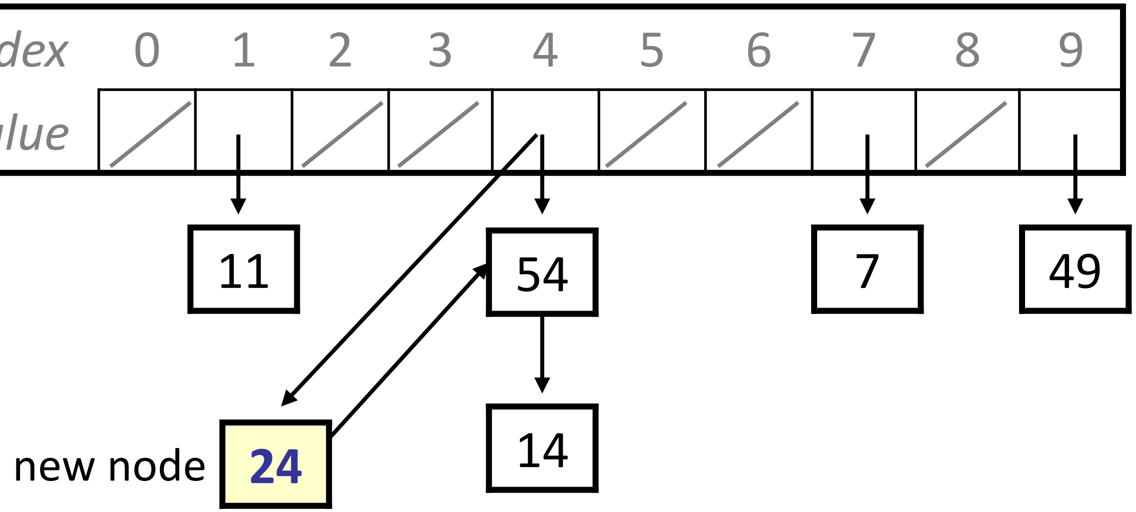



```
struct HashNode {
    int data;
    HashNode* next;
};
```

The add operation

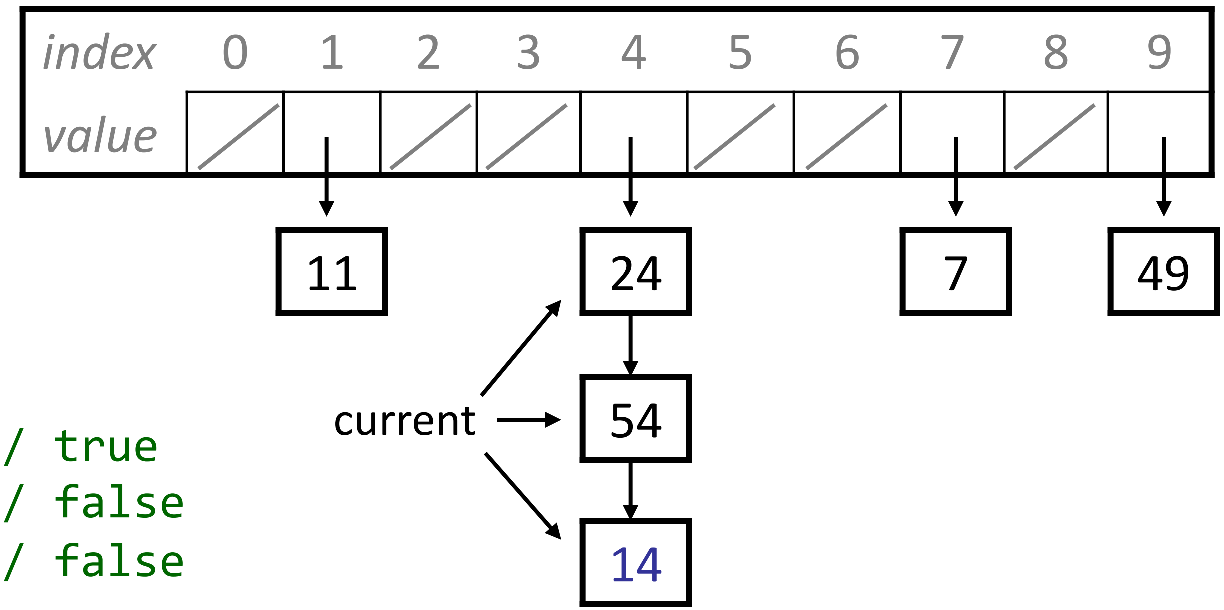
- How do we add an element to the hash table?
  - *Recall:* To modify a linked list, you must either change the list's front reference, or the `next` field of a node in the list.
  - Where in the list should we add the new element?
  - Must make sure to avoid duplicates.

```
set.add(24);
```



The contains operation

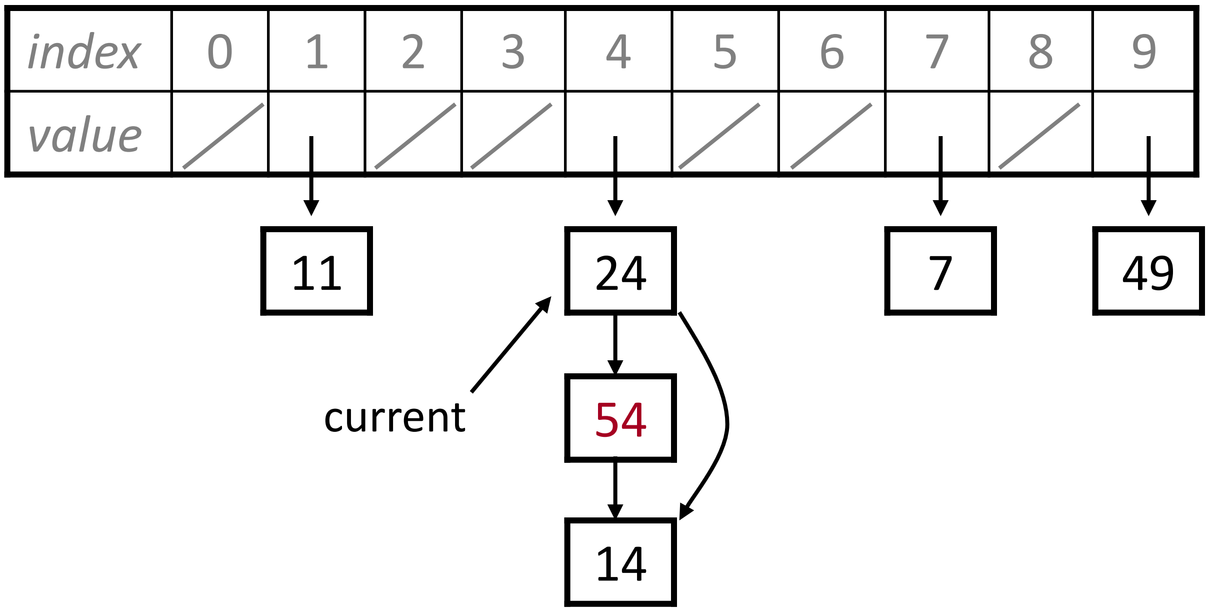
- How do we search for an element in the hash table?
  - Must loop through the linked list for the appropriate hash index, looking for the desired value.
  - *Recall:* Traverse a linked list with a "current" node pointer.




```
set.contains(14)   // true
set.contains(84)   // false
set.contains(53)   // false
```

The remove operation

- How do we remove an element from the hash table?
  - Cases to consider: front (24), non-front (14), not found (94), null (32)
  - To remove a node from a linked list, you must either change the list's front, or the `next` field of the *previous* node in the list.




```
set.remove(54);
```

Announcements

- Assn. 7 is due Thursday
- Please fill out the survey for this class! We really appreciate it.
- Thursday class is optional.
- Final is **Friday**, at 12:15PM, in **Hewlett**
  - More information: https://web.stanford.edu/class/cs106b/exams/final.html

Exercise: HashSet

- Implement a `HashSet` class that represents a set of integers using a hash table.
  - Include the following public members:

```
HashSet()
add(int value)
clear()
contains(int value)
remove(int value)
```

HashSet.h

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

class HashSet {
public:
    HashSet();
    ~HashSet();
    void add(int value);
    void clear();
    bool isEmpty() const;
    bool contains(int value) const;
    void remove(int value);
    int size() const;

private:
    HashNode** elements;
    int mysize;
    int capacity;
    int getIndex(int value) const;
};
```

HashSet.cpp

```
#include "HashSet.h"

HashSet::HashSet() {
    capacity = 10;
    mysize = 0;
    elements = new HashNode*[capacity]();   // all are null
}

void HashSet::add(int value) {
    if (!contains(value)) {
        int h = hashCode(value);    // insert at front of chain
        elements[h] = new HashNode(value, elements[h]);
        mysize++;
    }
}

bool HashSet::contains(int value) const {
    HashNode* curr = elements[hashCode(value)];
    while (curr != nullptr) {
        if (curr->data == value) { return true; }
        curr = curr->next;
    }
    return false;
}
```

HashSet.cpp 2

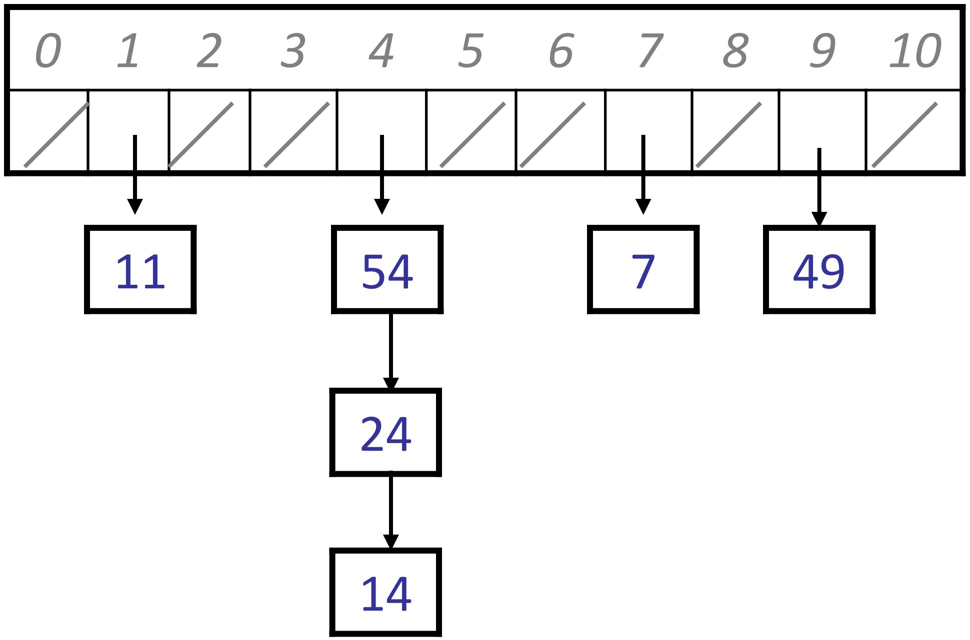
```
HashSet::~HashSet() {
    clear();
    delete[] elements;
}

void HashSet::clear() {
    for (int i = 0; i < capacity; i++) {
        while (elements[i] != nullptr) {   // free all chains
            HashNode* trash = elements[i];
            elements[i] = elements[i]->next;
            delete trash;
        }
    }
    mysize = 0;
}

int HashSet::getIndex(int value) const {
    return hash(value) % capacity;
}
```

HashSet.cpp 3

```
void HashSet::remove(int value) {
    int h = hashCode(value);
    if (elements[h] != nullptr) {
        if (elements[h]->data == value) {    // remove from front
            HashNode* trash = elements[h];
            elements[h] = elements[h]->next;
            mysize--;
            delete trash;
        } else {
            HashNode* curr = elements[h];
            while (curr->next != nullptr) {  // from middle/end
                if (curr->next->data == value) {
                    HashNode* trash = curr->next;    // found it
                    curr->next = curr->next->next;
                    mysize--;
                    delete trash;
                    break;
                }
                curr = curr->next;
            }
        }
    }
}
```

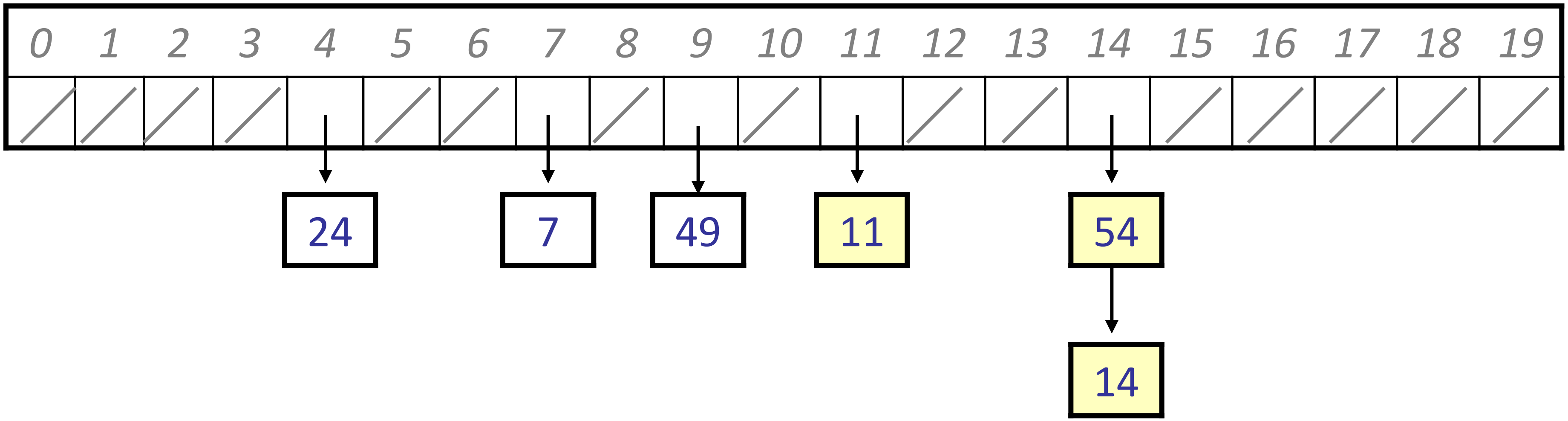
Rehashing

- **rehash**: Growing to a larger array when the table is too full.



Rehashing

- **rehash**: Growing to a larger array when the table is too full.
  - Cannot simply copy the old array to a new one. (Why not?)

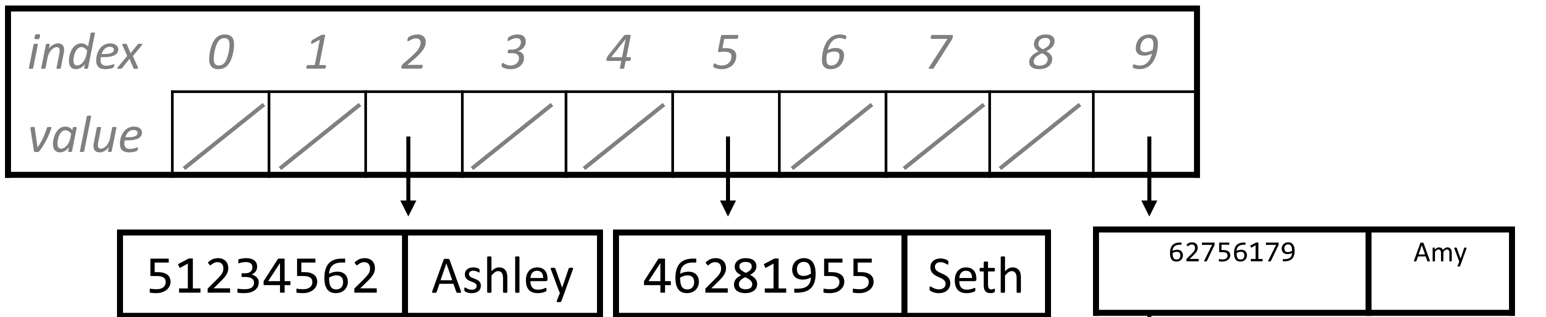


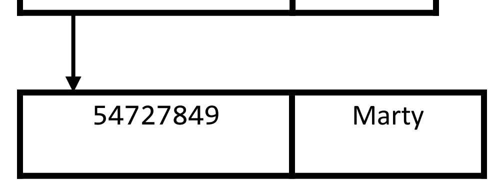
- **load factor**: ratio of (*# of elements*) / (*hash table length*)
  - many implementations rehash when load factor ≅ .75

Overflow

Hash map

- A hash map is like a set where the nodes store key/value pairs:






```
//          key (ID)  value (name)
map.put(51234562, "Ashley");
map.put(62756179, "Amy");
map.put(54727849, "Marty");
map.put(46281955, "Seth");
```

  – Must modify the HashNode class to store a key *and* a value

Hash map vs. hash set

- The hashing is always done on the keys, *not* the values.
- The `contains` function is now **`containsKey`**; there and in **`remove`**, you search for a node whose key matches a given key.
- The `add` method is now **`put`**; if the given key is already there, you must replace its old value with the new one.

```
map.put(54727849, "Chris");    // replace Marty with Chris
```