HASHING

MSCI 240: Algorithms & Data Structures

lecture/tutorial swap (Nov 26 → Nov 19; Nov 28 → Dec 3)

- today: two lectures—normal lecture + lecture in tutorial
- this week otherwise normal
- next Mon (Nov 26): tutorial in lecture time
- next Wed (Nov 28): no class or tutorial (would be Monday's tutorial)
- two Mon (Dec 3): two lectures—normal lecture + lecture in tutorial

lecture summary

- hashing motivation
- hashing
- collisions/probing/chaining
- rehashing
- complexity

Topic	Building Java Programs	Algorithms (Sedgewick)
classes, ADTs	chapter 8	1.2
arrays	chapter 7	
ArrayList<T>	chapter 10	1.3
Stack/Queue	chapter 14, (11)	1.3
LinkedList	chapter 16	1.3
Complexity		1.4
Searching	chapter 13	pp. 46-47
Sorting		chapter 2.1-2.3
Recursion	chapter 12	1.1 (p. 25)
Binary Trees	chapter 17	chapter 3.1-3.2
Dictionaries	chapter 18.1	chapter 3.4
Graphs	N/A (Wikipedia good)	chapter 4.1
Heaps/Priority Queues	chapter 18.2	chapter 2.4

motivation (similar to BSTs)

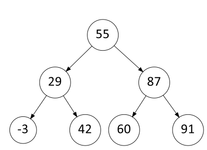
data structure	insert	remove	search/ contains
array / ArrayList	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
sorted array / List	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
BST *average	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
ideal	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

sounds great! what can we sacrifice to improve time complexity?

`SearchTree` as a set

we implemented a class `SearchTree` to store a BST of `int`s:

our BST is essentially a set of integers
(if we don't allow duplicates)



operations:

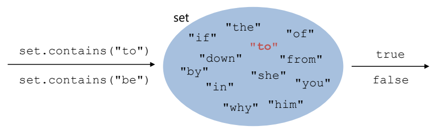
```
add
contains
remove
   ...
```

but there are other ways to implement a set...

recall: a set is a collection of unique values (no duplicates) that can perform the following operations efficiently:

add, remove, search (contains)

we don't think of a set as having indexes; we just add things to the set in general and don't worry about order



problem: imagine we are creating an `IntSet` class, what data structure should we store the values in?

consider storing a set in an unfilled array (e.g., an `ArrayList`)

it doesn't really matter what order the elements appear in a set, so long as they can be added and searched quickly

if we store them in the next available index, as in a list, ...

```
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	9	23	8	-3	49	12	0	0	0	0

size: 6

How efficient is `add`? `contains`? `remove`?

$O(1)$, $O(n)$, $O(n)$

(`contains` must loop over the array; `remove` must shift elements) 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

How efficient is `add`? `contains`? `remove`?

$O(n)$, $O(\log n)$, $O(n)$

(you can do a binary search to find elements in `contains`, and to find the proper index in `add`/`remove`; but `add`/`remove` still need to shift elements right/left to make room)

silly idea: when client adds value `i`, store it at index `i`

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

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]
value	0	1	0	0	0	0	0	7	0	9	0	0	12	0	0	0	0	0	18	0

size: 5

hash: to map a large domain of values to a smaller fixed domain

typically, mapping a set of elements to integer indexes in an array

idea: store a given element value in a particular predictable index

that way, adding / removing / looking for it are constant-time, $O(1)$

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 example, our “hash function” was:

`hash(i) → i`

- potentially requires a large array (`a.length > i`)
- doesn’t work for negative numbers
- array could be very sparse, mostly empty (memory waste)

improved hash function:

- to deal with negative numbers: `hash(i) → abs(i)`
- to deal with large numbers: `hash(i) → abs(i) % length`

```
set.add(37);    // abs(37) % 10 == 7
set.add(-2);    // abs(-2) % 10 == 2
set.add(49);    // abs(49) % 10 == 9
```

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
value	0	0	-2	0	0	0	0	37	0	49

size: 3

```
// inside HashIntSet class
private int hash(int i) {
    return Math.abs(i) % elements.length;
}
```


sketch of implementation

```
public class HashIntSet {
    private int[] elements;
    // ...
    public void add(int value) {
        elements[hash(value)] = value;
    }

    public boolean contains(int value) {
        return elements[hash(value)] == value;
    }

    public void remove(int value) {
        elements[hash(value)] = 0;
    }
}
```

problems with this approach?

$O(1)$

$O(1)$

$O(1)$

collisions

collision: when hash function maps two values to same index

```
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: 4

collision resolution: an algorithm for fixing 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!
```

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
value	0	11	0	0	24	54	0	37	0	49

size: 5

variation: quadratic probing moves increasingly far away:

`+1, +4, +9, ...`

add operation (assume `0` is an illegal value):

- use the hash function to find the proper bucket index
- if we see a `0`, put it there
- if not, move forward until we find an empty (`0`) index to store it
- if we see that the value is already in the table, don't re-add it

search/contains operation:

- use the hash function to find the proper bucket index
- loop forward until we either find the value, or an empty index (`0`)
- if find the value, return `true`, if we find `0`, return `false`

remove operation:

- we cannot remove by simply zeroing out an element (why not?)
- instead, we replace it by a special "removed" placeholder value
- (can be re-used on `add`, but keep searching on `contains`)

problem: full array

clustering: clumps of elements at neighboring indexes

- slows down the hash table lookup; you must loop through them
- where does each value go in the array?

```
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: 7

- how many indexes must be examined to answer `contains(94)`?
- what will happen if the array completely fills?

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 capacity)

- many collections rehash when load factor ≈.75

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
value	95	11	0	0	24	54	14	37	66	48

size: 8

index	[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]	[10]	[11]	[12]	[13]	[14]	[15]	[16]	[17]	[18]	[19]
value	0	0	0	0	24	0	66	0	48	0	0	11	0	0	54	95	14	37	0	0

size: 8

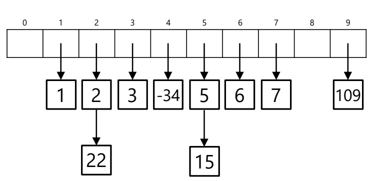
separate chaining

separate chaining: solving collisions by storing a list at each index

add/contains/remove must traverse lists, but the lists are short

impossible to "run out" of indexes, unlike with probing

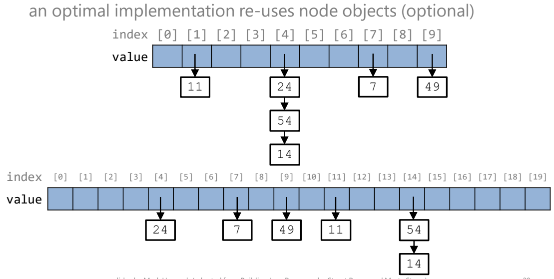

```
HashSet<Integer> set = new HashSet<>(10);
set.add(5);
set.add(7);
set.add(3);
set.add(2);
set.add(6);
set.add(1);
set.add(109);
set.add(-34);
set.add(22);
set.add(15);
set.add(1); // no change
```



rehashing with chaining

separate chaining handles rehashing similarly to probing

- loop over the list in each hash bucket; re-add each element
- an optimal implementation re-uses node objects (optional)



dictionary/map

store data in key/value pairs

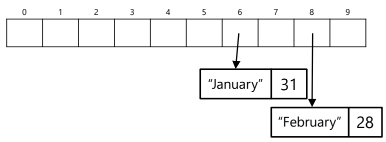
- e.g., key = String, value = Integer

```
HashMap<String, Integer> daysInMonth = new HashMap<>();
daysInMonth.put("January", 31);      } insert: Θ(1)
daysInMonth.put("February", 28);
// ...
int janDays = daysInMonth.get("January");     } search: Θ(1)
int febDays = daysInMonth.get("February");
```

use hash code from key

"January" hash code: -162006966

"February" hash code: -199248958



`HashMap` vs. `HashSet`

the hashing is always done on the keys, not the values

the `contains` method is now `containsKey`; there and in `remove`, you search for a node whose key matches

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("February", 28);
map.put("February", 29);   // replace 28 with 29
```


hashing complexity

what is a worst-case example for `add`/`contains`/`remove` using separate chaining?

how long will each chain be in the average case?

hashing complexity

assume hash function picks "slots" with uniform probability

what is input size?

number of elements $= n$

number of slots $= m$

$$T(m,n) = \underbrace{T_{hash}(m,n)}_{O(1)} + \underbrace{T_{search}(m,n)}_{O(?)}$$

examples (average length of chain?):

$n = 1000, m = 100 \quad \Longrightarrow 10$

$n = 100, m = 1000 \quad \Longrightarrow \frac{1}{10}$

general case:

$$\alpha = \frac{n}{m} \text{ } \}\text{load factor}$$

average case complexity

case 1: unsuccessful search

cost $= O(\alpha) = O\left(\frac{n}{m}\right)$

case 2: successful search

cost $= O\left(\frac{\alpha}{2}\right) = O(\alpha) = O\left(\frac{n}{m}\right)$

slides by Mark Hancock 37

can choose m!

typically grow (rehash) when $\alpha = 0.75$

i.e., ensure $m \geq \frac{4}{3}n$

$\therefore O(\alpha) = O\left(\frac{n}{m}\right) = O\left(\frac{3}{4}\right) = O(1)$

slides by Mark Hancock 38

hashing summary

hashing allows constant time `search`/`add`/`remove`

linear probing and separate chaining can be used to deal with collisions

rehashing necessary when the load factor (α) gets too high

average case complexity is actually $O(\alpha)$, but we can choose number of slots (i.e., rehash when α too high)

slides by Mark Hancock 39

clicker question

which statement is true

A. a hash table has $O(\log n)$ average time to add and search for elements
B. the higher a hash table's load factor, the more quickly elements can be found
C. once a hash table's load factor reaches 0.75, no more elements can be added
D. a hash function maps element values to integer indexes in the hash table
E. a good hash function returns the same value as much as possible

next:
graphs