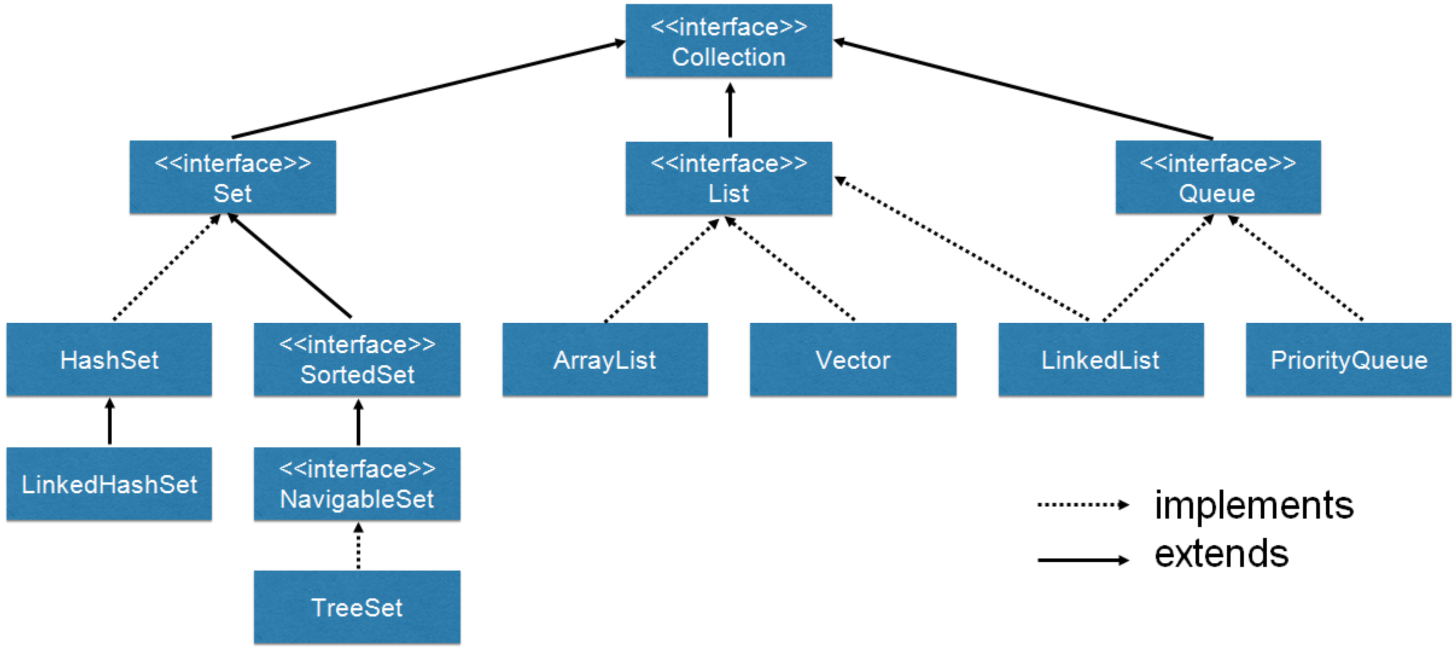
COSC 222 Data Structure

Hashing

Sets

- **Set**: A collection of unique values that can perform the following operations efficiently:
  - unordered collection of objects
  - no duplicates allowed
  - add, remove, search (contains)



Int Set ADT interface

- Own implementation of a set.
  - To simplify the problem, we only store **`int`** in our set for now.
  - We will define sets as an ADT by creating a Set interface.
  - Core operations are: add, contains, remove.

```
public interface IntSet {
    void add(int value);
    boolean contains(int value);
    void clear();
    boolean isEmpty();
    void remove(int value);
    int size();
}
```

Unfilled array set

- Consider storing a set in an unfilled array.

- 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.)

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									

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

    `O(N), O(log N), O(N)`

    (You can do an `O(log N)` binary search to find elements in `contains`, otherwise `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);
...
```

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

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

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																			

Hashing

- **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 any 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 slide, 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 implements IntSet {
    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;
    }
}
```

- Runtime of **add**, **contains**, and **remove**: **O(1)** **!!**
  Are there any problems with this approach?

Collisions

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

```
set.add(11);        //abs(11) % 10 == 1
set.add(49);        //abs(49) % 10 == 9
set.add(24);        //abs(24) % 10 == 4
set.add(37);        //abs(37) % 10 == 7
set.add(54);        //abs(54) % 10 == 4 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									

- **collision resolution**: An algorithm for fixing collisions.

Open Addressing

- When a data item can't be placed at the index calculated by the hash function, another location in the array is sought.
- Resolving a collision by moving to another index.
  - **linear probing**:

    Moves to the next available index (wraps if needed).

    Search the array in order: i, i+1, i+2, i+3 . .
  - **quadratic probing:**

    Moves increasingly far away

    Search the array in this sequence: i, $i+1^2$, $i+2^2$, $i+3^2$ . . .
  - **double hashing:**

    Hash the hash code/key a second time using a different hash function

Probing

- **linear probing**: search the array in order: i, i+1, i+2, i+3 . .

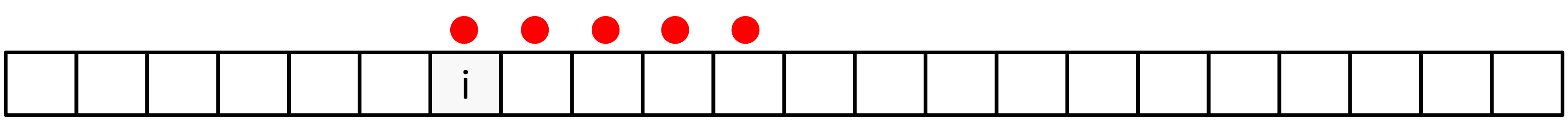
```
set.add(11);        //abs(11) % 10 == 1
set.add(49);        //abs(49) % 10 == 9
set.add(24);        //abs(24) % 10 == 4
set.add(37);        //abs(37) % 10 == 7
set.add(54);        //abs(54) % 10 == 4 collides with 24!
```

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

probe #1 add(54): i full!

probe #2 i+1 has space!

Primary Clustering



Probing

- **Quadratic probing**: Search the array in this sequence with a formula. An example: $i, i+1^2, i+2^2, i+3^2 \ldots$

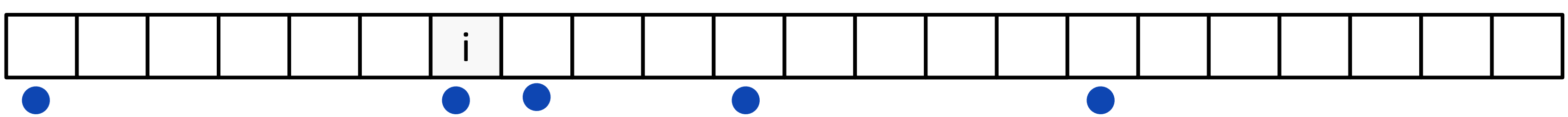
```
set.add(11);
set.add(49);
set.add(24);
set.add(37);
set.add(54);  // collides with 24; must probe
```

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

probe #1 add(54): I full!

probe #2 $i+1^2$ has space!

Secondary Clustering



Probing

- **Double hashing**: Double hashing uses a secondary hash function
- If a key x hashes to a position p that is already occupied, then the next position p' = p + secondaryhashcode(x)
- If this new position is also occupied, then we look to position p" = p + 2 * secondaryhashcode(x)
- Common choice of compression function for the secondary hash function: `h'(k) = q - k mod q` where `q < N` and `q` is a prime

- N= 15
- Keys: 15, 30, 45, 60, 75, 90, 105
- Probe sequence: 0, 5, 10, 0, 5, 10, and so on

Probing

- Insert keys 18, 41, 22, 44, 59, 32, 31, 73, in this order

- N = 13
- h(k) = k mod 13
- d(k) = 7 - k mod 7
- If p is occupied:

  p’ = p + secondaryhashcode(x)

  p” = p + 2 * secondaryhashcode(x)

k	$h(k)$	$d(k)$	Probes		
18	5	3	5		
41	2	1	2		
22	9	6	9		
44	5	5	5	10	
59	7	4	7		
32	6	3	6		
31	5	4	5	9	0
73	8	4	8		

31		41			18	32	59	73	22	44		
0	1	2	3	4	5	6	7	8	9	10	11	12

Implementing HashIntSet

- Let's implement an `int` set using a hash table with linear probing.
  - For simplicity, assume that the set cannot store 0s for now.

```
public class HashIntSet implements IntSet {
    private int[] elements;
    private int size;

    // constructs new empty set
    public HashIntSet() {
        elements = new int[10];
        size = 0;
    }

    // hash function maps values to indexes
    private int hash(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

The add operation

- How do we add an element to the hash table?
  - 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.

  - `set.add(54);`
  - `set.add(14);`

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

The add operation

- How do we add an element to the hash table?

```
public void add(int value) {
    int h = hash(value);
    while (elements[h] != 0 &&
           elements[h] != value) {     // linear probing
        h = (h + 1) % elements.length; //for empty slot
    }
    if (elements[h] != value) {      // avoid duplicates
        elements[h] = value;
        size++;
    }
}
```

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

The contains operation

- How do we search for an element in the hash table?
  - 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, it is contained (`true`). If we find 0, it is not (`false`).

  - `set.contains(24)   // true`
  - `set.contains(14)   // true`
  - `set.contains(35)   // false`

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

Implementing contains

```
public boolean contains(int value) {
    int h = hash(value);
    while (elements[h] != 0) {
        if (elements[h] == value) {   // linear probing
            return true;              // to search
        }
        h = (h + 1) % elements.length;
    }
    return false;                     // not found
}
```

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

The remove operation

- We cannot remove by simply zeroing out an element:

```
set.remove(54);       // set index 5 to 0
set.contains(14)      // false???  oops
```

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	**0**	14	34	0	49
size	5									

- Instead, we replace it by a special "removed" placeholder value
  - (can be re-used on `add`, but keep searching on `contains`)

index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
value	0	11	0	0	24	XX	14	34	0	49
size	5									

Implementing remove

```
public void remove(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value) {
         h = (h + 1) % elements.length;
    }
    if (elements[h] == value) {
        elements[h] = -999;   // "removed“; flag value
        size--;
    }
}
```

index	0	1	2	3	4	5	6	7	8	9
value	0	11	0	0	24	54	14	34	0	49
size	5									

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

Patching add, contains

```
private static final int REMOVED = -999;

public void add(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value &&
           elements[h] != REMOVED) {
        h = (h + 1) % elements.length;
    }
    if (elements[h] != value) {
        elements[h] = value;
        size++;
    }
}

// contains does not need patching;
public boolean contains(int value) {
    int h = hash(value);
    while (elements[h] != 0 && elements[h] != value) {
        h = (h + 1) % elements.length;
    }
    return elements[h] == value;
}
```

Problem: full array - apply dynamic resizing

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

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*	*0*	*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																			

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

Implementing rehash

```
// Grows hash table to twice its original size.
   private void rehash() {
       int[] old = elements;
       elements = new int[2 * old.length];
       size = 0;
       for (int value : old) {
           if (value != 0 && value != REMOVED) {
               add(value);
           }
       }
   }

   public void add(int value) {
       if ((double) size / elements.length >= 0.75) {
           rehash();
       }
       ...
   }
```

Hash table sizes

- Can use prime numbers as hash table sizes to reduce collisions.
- Also improves spread / reduces clustering on rehash.

```
set.add(11);  //  11 % 10 ==  1
set.add(39);  //  39 % 10 ==  9
set.add(21);  //  21 % 10 ==  1
set.add(29);  //  29 % 10 ==  9
set.add(71);  //  81 % 10 ==  1
set.add(41);  //  41 % 10 ==  1
set.add(101);  // 101 % 10 == 1
```

```
set.add(11);  //  11 % 13 == 11
set.add(39);  //  39 % 13 ==  0
set.add(21);  //  21 % 13 ==  8
set.add(29);  //  29 % 13 ==  3
set.add(71);  //  81 % 13 ==  6
set.add(41);  //  41 % 13 ==  2
set.add(101); // 101 % 13 == 10
```

index	0	1	2	3	4	5	6	7	8	9	10	11	12
value	39	0	41	29	0	0	71	0	21	0	101	11	0
size	7												

Other details

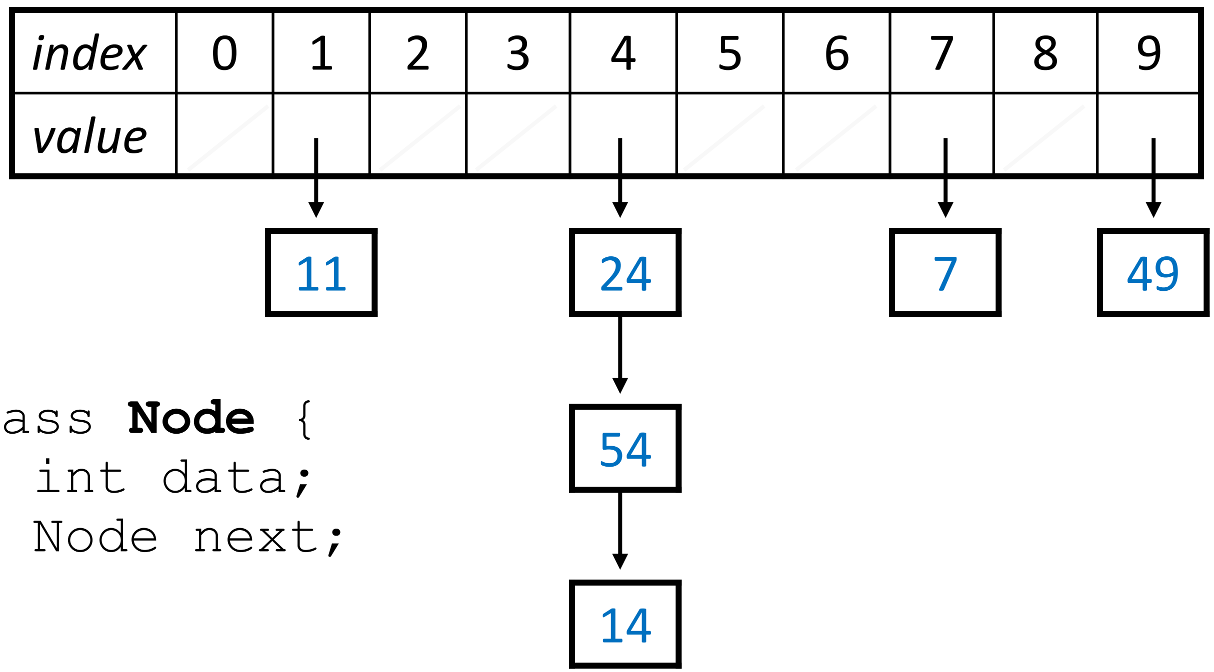
- How would we implement `toString` on our `HashIntSet`?

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

```
System.out.println(set);
// [11, 24, 54, 37, 49]
```

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




```
private class Node {
    public int data;
    public Node next;
    ...
}
```

Implementing HashIntSet

- Let's implement a hash set using separate chaining.

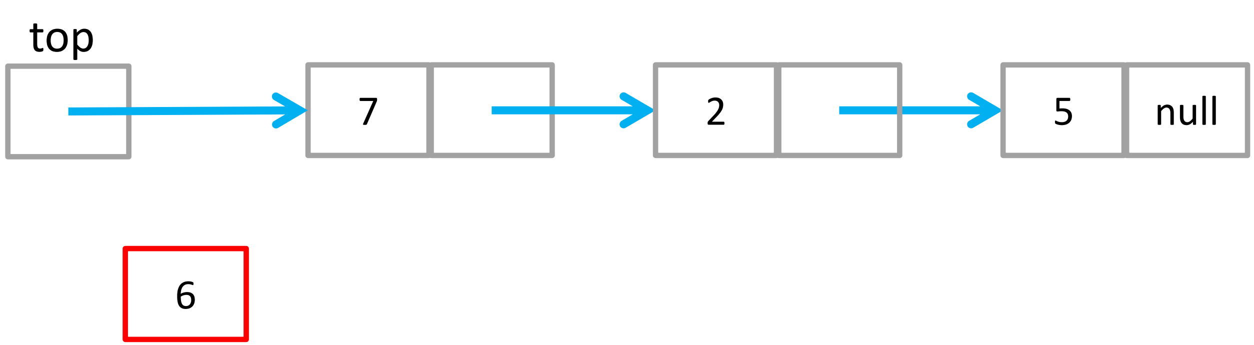
```
public class HashIntSet implements IntSet {
    // array of linked lists;
    // elements[i] = front of list #i (null if empty)
    private Node[] elements;
    private int size;

    // constructs new empty set
    public HashIntSet() {
        elements = new Node[10];
        size = 0;
    }

    // hash function maps values to indexes
    private int hash(int value) {
        return Math.abs(value) % elements.length;
    }
    ...
```

Linked List: An add method

- Add a new element to the beginning of a LinkedList. (This is in the LinkedList class.)

```
public void add(Object newItem) {
  Node newNode = new Node(newItem,top);
  top = newNode;
}
```

- Diagram (simpler version):

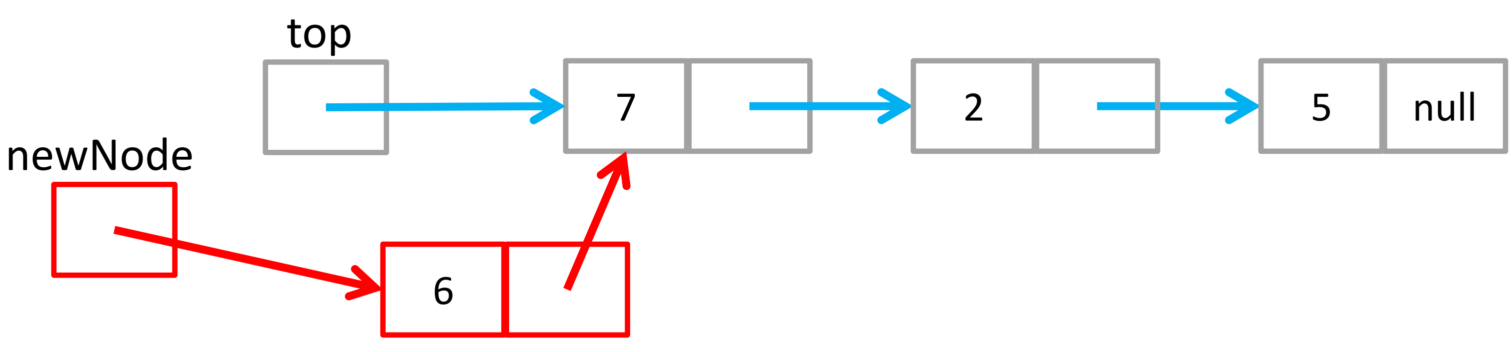


Linked List: An add method

- Add a new element to the beginning of a LinkedList. (This is in the LinkedList class.)

```
public void add(Object newItem) {
  Node newNode = new Node(newItem,top);
  top = newNode;
}
```

- Diagram (simpler version):

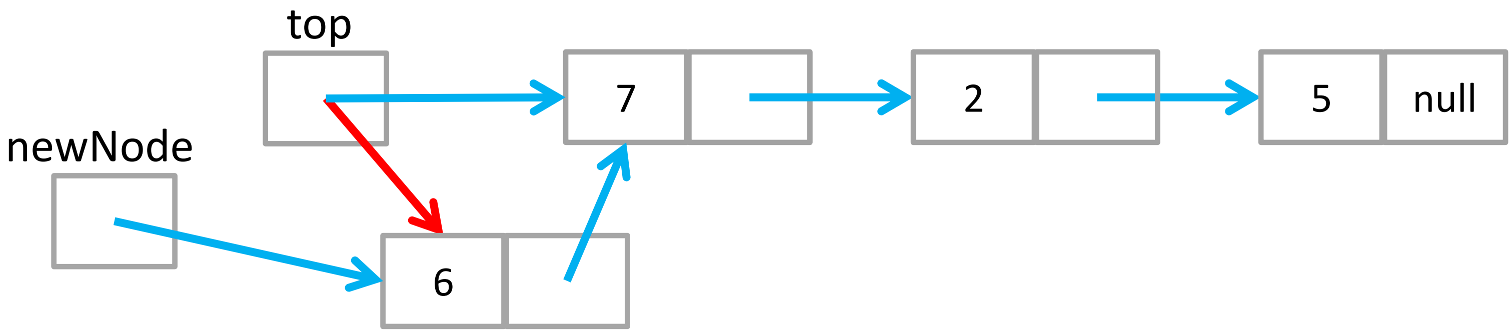


Linked List: An add method

- Add a new element to the beginning of a LinkedList. (This is in the LinkedList class.)

```
public void add(Object newItem) {
  Node newNode = new Node(newItem,top);
  top = newNode;
}
```

- Diagram (simpler version):

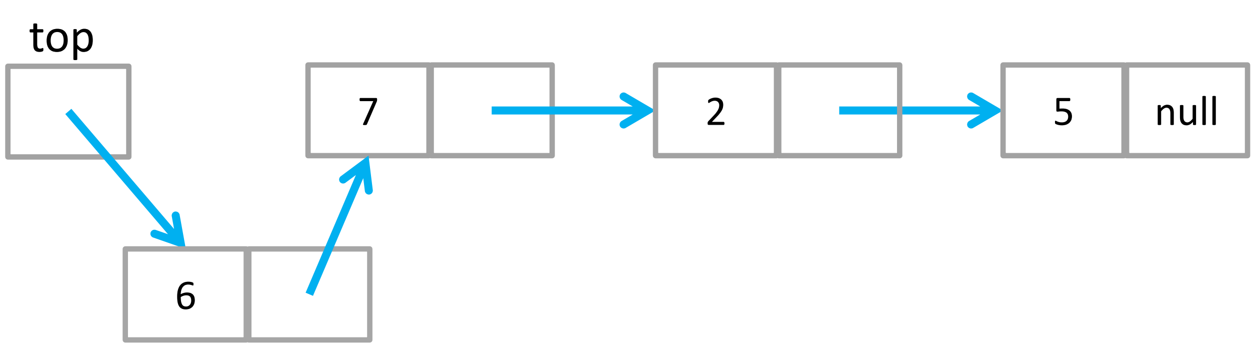


Linked List: An add method

- Add a new element to the beginning of a LinkedList. (This is in the LinkedList class.)

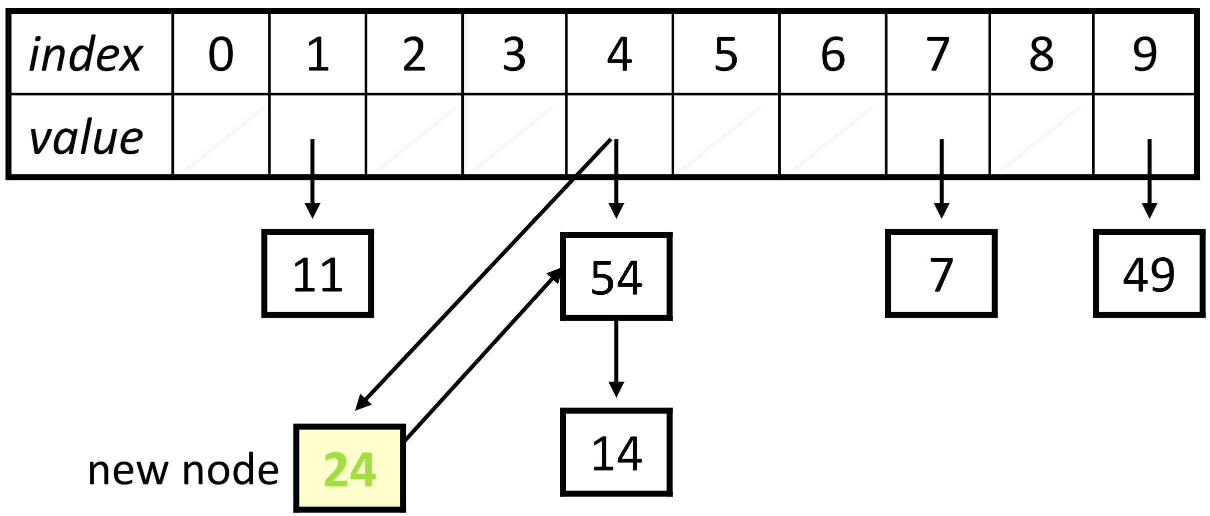
```
public void add(Object newItem) {
  Node newNode = new Node(newItem,top);
  top = newNode;
}//add
```

- Diagram (simpler version):



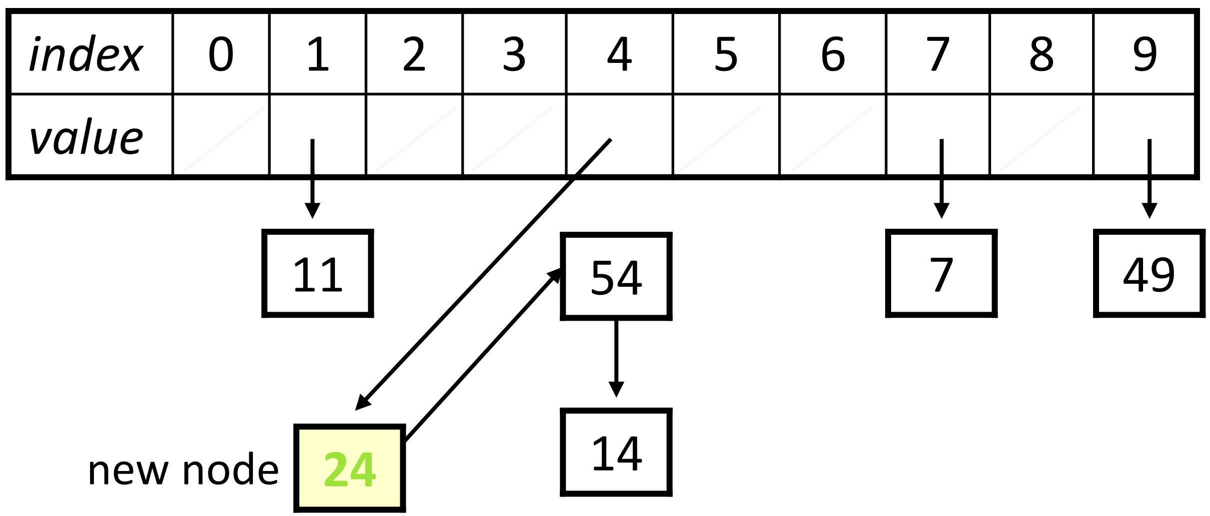
The add operation

- How do we add an element to the hash table?
  - When you want 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);`



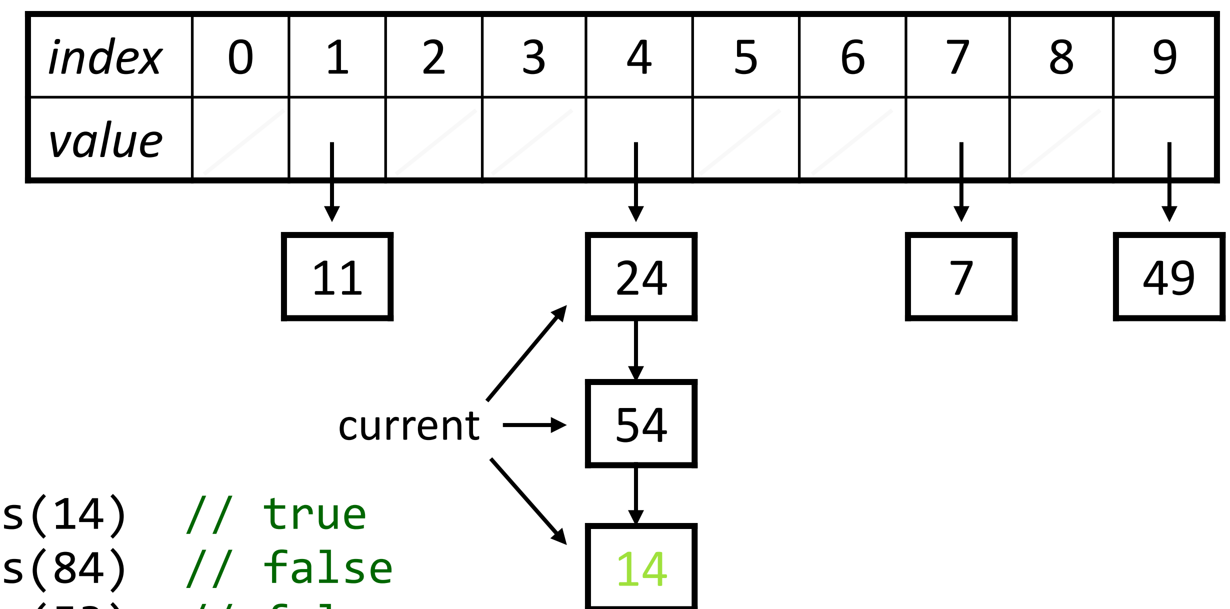
Implementing add

```
public void add(int value) {
    if (!contains(value)) {
        int h = hash(value);              // add to front
        Node newNode = new Node(value);   // of list #h
        newNode.next = elements[h];
        elements[h] = newNode;
        size++;
    }
}
```



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.
  - Looping through a linked list requires a "current" node reference.



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

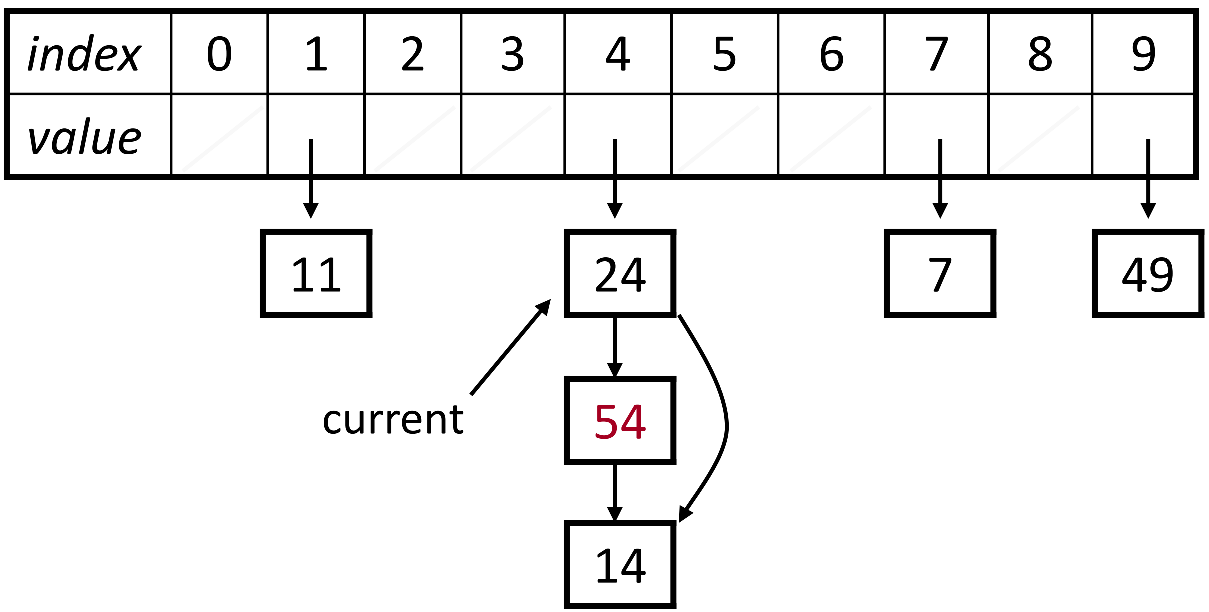
Implementing contains

```
public boolean contains(int value) {
    Node current = elements[hash(value)];
    while (current != null) {
        if (current.data == value) {
            return true;
        }
        current = current.next;
    }
    return false;
}
```

The remove operation

- How do we remove an element from the hash table?
  - To remove a node from a linked list, you must either change the list's front reference, or the `next` field of the `previous` node in the list.

  - `set.remove(54);`



Implementing remove

```
public void remove(int value) {
    int h = hash(value);
    if (elements[h] != null && elements[h].data == value) {
        elements[h] = elements[h].next;  // front case
        size--;
    } else {
        Node current = elements[h];      // non-front case
        while (current != null && current.next != null) {
            if (current.next.data == value) {
                current.next = current.next.next;
                size--;
                return;
            }
            current = current.next;
        }
    }
}
```

Thank you