Linear Probing vs. Double Hashing

Isabella Felix
Nicholas Gannon
Jasiel Garcia
Eden O’Leary

Collision Handling

Linear Probing

- Collision resolved by putting item in the next empty place in the array following the occupied place
- Clusters items in array which may slow down search

Double Hashing

- Method of open addressing in which collision is resolved by searching for an empty place at intervals using a different hash function
- Clustering is minimized because colliding values are more spread out
- h’(k) = q - (h(k) modulo q)

Implementation

- Dictionary and Hash Comparator interfaces, Item, String Comparator, and LPHash provided
- Linear Probing and Double Hash only differ in equation used for inserting elements
- Abstraction is possible!
- Create abstract class for hashing and extend for separate collision handlers

Abstract Hashing Class

```
public abstract class Hash <K, E> implements IDictionary <K, E>{

        protected Item<K, E> AVAILABLE = new Item<K, E>(null, null);
// deleted cell item
        protected int n; // number of elements in the hash table
        protected int N; //size of hash table
        protected ArrayList<Item<K, E>> A;
        protected HashComparator<K> h;
        protected int Collisions = 0;

        protected abstract int find(K k);

        public Hash(int s, HashComparator<K> hc) {
                n = 0; h = hc;
                N = s;
                int i = -1;
                A = new ArrayList<Item<K, E>>(s);
                while (i < s - 1) {
                        i = i + 1;
                        A.add(i, null);
                }
        }
```

```
// Purpose: to determine if a spot is available
        public boolean available(int i) {
                return(A.get(i) == AVAILABLE);
        }
        // Purpose: to determine if a spot is empty
        public boolean empty(int i) {
                return(A.get(i) == null);
        }
        // Purpose: to define a key
        public K key(int i) {
                return(A.get(i).getKey());
        }
        // Purpose: to define an element
        public E elem(int i) {
                return(A.get(i).getElem());
        }
        // Dictionary methods
        public Integer size() {
                return(n);
        }
        public Boolean isEmpty() {
                return(n == 0); }
```

Abstract Hashing Class (cont.)

```
// Purpose: to go through the elements in a dictionary
        public Iterator<E> elements() {
                Iterator<Item<K, E>> htlooper = A.iterator();
                ArrayList<E> elems = new ArrayList<E>();
                Item<K, E> k;
                while (htlooper.hasNext()) {
                        k = htlooper.next();
                        if ((k != null) && (k != AVAILABLE)) {
                                elems.add(k.getElem()); }}
                return(elems.iterator());}

// Purpose: to go through the keys in a dictionary
        public Iterator<K> keys() {
                Iterator<Item<K, E>> htlooper = A.iterator();
                ArrayList<K> keys = new ArrayList<K>();
                Item<K, E> k;
                while (htlooper.hasNext()) {
                        k = htlooper.next();
                        if ((k != null) && (k != AVAILABLE)) {
                                keys.add(k.getKey()); }}
                return(keys.iterator());}
```

```
// Purpose: to find an element
        public E findElement(K k) {
                int i = find(k);
                if (i < 0) {
                        return (null);
                }
                else {
                        return (elem(i));
                }
        }
        //Purpose: to delete an element and key
        public void delete(K k) {
                int i = find(k);
                if (i > -1) {
                        A.set(i, AVAILABLE);
                        n = n - 1;
                }
        }
```

Linear Probing Class

```
public class LPHash <K, E> extends Hash<K, E>{
    public LPHash(int s, HashComparator<K> hc) {
        super(s, hc);
    }
    //Purpose: to insert an element and key
    public void insert(K k, E e) {
        int i = h.hashIndex(k) % N;
        int j = i;
        boolean done = false;
        while (!done) {
            if (empty(j) || available(j)) {
                A.set(j, new Item<K, E>(k, e));
                n = n + 1;
                done = true;}
            else {
                this.Collisions++;
                j = (j+1)%N;
            }}}
```

```
//Purpose: to find a key in a Table
public int find(K k) {
    int i = (this.h.hashIndex(k) % N);
    int j = i; int res = -1;
    boolean done = false;
    while (!done) {
        if (this.empty(j)) {
            done = true; }
        else if (this.available(j)) {
            j = (j + 1) % N;
            if (j == i) {
                done = true; } }
        else if (h.keyEqual(key(j), k)) {
            res = j; done = true;}
        else { j = (j+1)%N;
            if (j == i) { done = true; }}
            }
            return(res); }}
```

Double Hashing Class 

```
public class DbHash<K, E> extends Hash<K, E>{
        private int q = 17;
        public DbHash(int s, HashComparator<K> hc) {
                super(s, hc);}
        //Purpose: to insert an element and key
        public void insert(K k, E e) {
                int i = h.hashIndex(k) % N;
                int j = i; int hp = q-(j%q);
                boolean done = false;
        while (!done) {
                if (empty(j) || available(j)) {
                        A.set(j, new Item<K, E>(k, e));
                        n = n + 1; done = true;}
                else {
                        this.Collisions++;
                        j = (j + hp) % N;
                                }}}
```

```
//Purpose: to find a key in a Table
        public int find(K k) {
        int i = (this.h.hashIndex(k) % N);
        int j = i; int res = -1; int hp = q-(j%q);
        boolean done = false;
                while (!done) {
                        if (this.empty(j)) { done = true; }
                        else if (this.available(j)) {
                                j = (j + hp) % N;
                                if (j == i) { done = true; }
                                        }
                        else if (h.keyEqual(key(j), k)) {
                                res = j; done = true;}
                        else {
                                j = (j + hp) % N;
                        if (j == i) { done = true; }}}
                                return(res);
                        }
```

Testing

```
class LPHashTests {

HashComparator<String> scomp = new StringComparator();
LPHash<String, String> ht = new LPHash<String, String>(101, scomp);

	@Test
	public void testHT() {

		assertEquals(ht.size(), 0);
		assertEquals(ht.isEmpty(), true);
		assertEquals(ht.findElement("Isabella"), null);

		ht.insert("Isabella", "Felix");
		ht.insert("Nicholas", "Gannon");
		ht.insert("Eden", "O'Leary");
		ht.insert("Jasiel", "Garcia");

		assertEquals(ht.size(), 4);
		assertEquals(ht.isEmpty(), false);
		assertEquals(ht.findElement("Eden"), "O'Leary");
		assertEquals(ht.findElement("Craig"), null);

		ht.delete("Isabella");
		assertEquals(ht.findElement("Isabella"), null);

		Iterator<String> klooper = ht.keys();
		Iterator<String> elooper = ht.elements();
		String res = "";
		while (klooper.hasNext()) {
			res = res + " " + klooper.next() + " " + elooper.next();
			}
	assertEquals(res, " Eden O'Leary Jasiel Garcia Nicholas Gannon");

		}
	}
```

Testing

```
class LPHashTests {

HashComparator<String> scomp = new StringComparator();
DbHash<String, String> ht = new DbHash<String, String>(101, scomp);

	@Test
	public void testHT() {

		assertEquals(ht.size(), 0);
		assertEquals(ht.isEmpty(), true);
		assertEquals(ht.findElement("Isabella"), null);

		ht.insert("Isabella", "Felix");
		ht.insert("Nicholas", "Gannon");
		ht.insert("Eden", "O'Leary");
		ht.insert("Jasiel", "Garcia");

		assertEquals(ht.size(), 4);
		assertEquals(ht.isEmpty(), false);
		assertEquals(ht.findElement("Eden"), "O'Leary");
		assertEquals(ht.findElement("Craig"), null);

		ht.delete("Isabella");
		assertEquals(ht.findElement("Isabella"), null);

		Iterator<String> klooper = ht.keys();
		Iterator<String> elooper = ht.elements();
		String res = "";
		while (klooper.hasNext()) {
			res = res + " " + klooper.next() + " " + elooper.next();
			}
	assertEquals(res, " Eden O'Leary Jasiel Garcia Nicholas Gannon");

		}
	}
```

Which one is better? 

```
public class CompareHash {
	public static String genRandomString() {
		String s = "";
		int i = 0;
		Random r = new Random();
		while (i < 1 + r.nextInt(20)) {
			s += (char) r.nextInt(256);
			i++;}
		return s;}
	public static ArrayList<String> makeList(int len){
		ArrayList<String> arr = new ArrayList<String>();
		String str; int i = 0;
		while(i < len) {
			str = genRandomString();
			if(!arr.contains(str)) {
				arr.add(str);
				i++;
			}
		}
		return arr;
	}
```

```
public static void main(String[] args) {
	int size = 1000;
	while(size<=10000) {
	HashComparator<String> hc = new StringComparator();
	LPHash<String, String> LPhasher = new LPHash<String, String>(10007, hc);
	ArrayList<String> keys = makeList(size);
	ArrayList<String> elems = makeList(size);
		for(int i = 0; i<keys.size(); i++)
			{
				Dbhasher.insert(keys.get(i), elems.get(i));
			}
			int dbc = Dbhasher.Collisions;
			System.out.println(size);
			System.out.println(dbc);
			size +=1000;
		}
	}
}
```

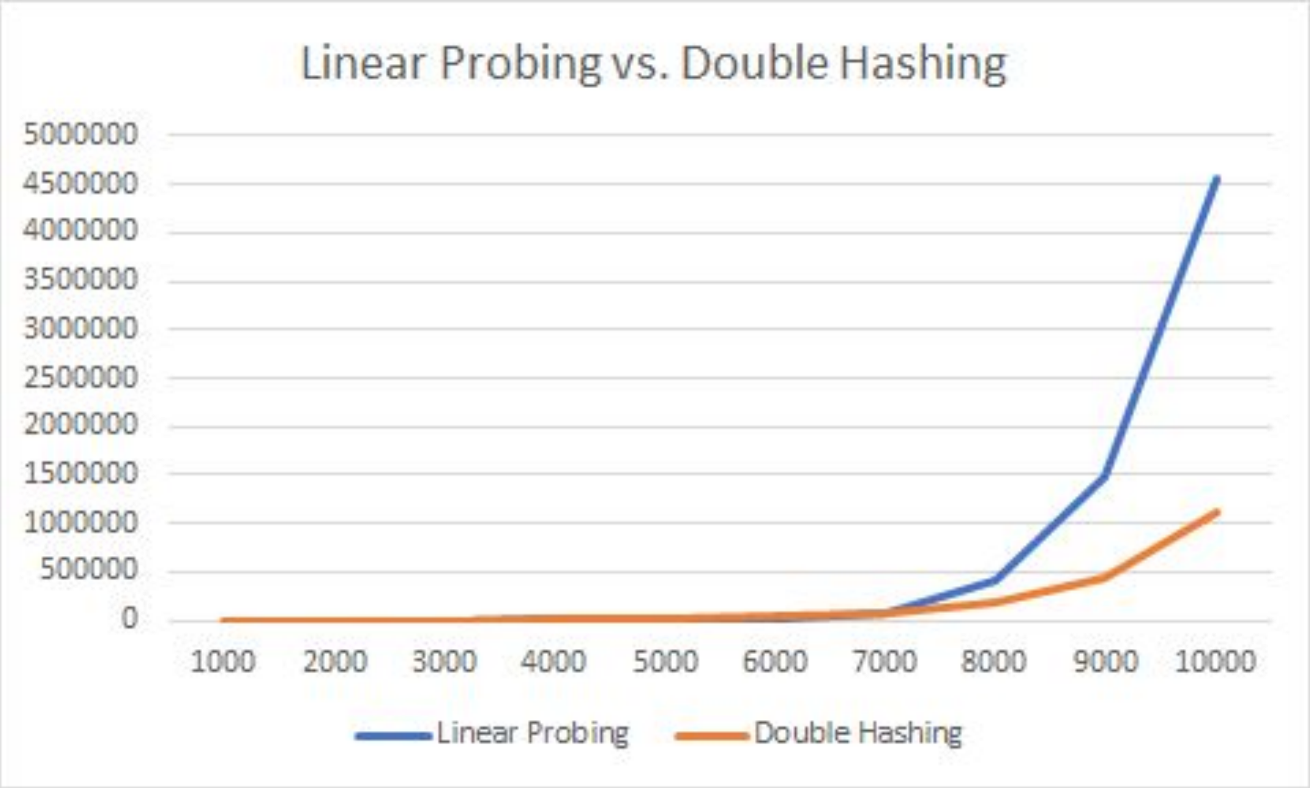
Quantitative Data

Linear Probing										
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	58	282	665	2035	7210	26015	72386	421094	1442830	4215677
2	74	251	722	3307	13210	28942	44084	48087	1590098	4806541
3	60	275	1077	4022	8417	31552	88744	397762	1555867	4840770
4	50	269	736	1962	11091	17780	98204	511513	1621626	4780163
5	65	333	779	1840	9872	25616	81078	329672	1532115	4528989
6	69	278	939	2288	12449	22451	69233	562048	1421215	4649980
7	64	295	914	3360	7731	34028	93428	350237	1450692	4680226
8	60	266	748	3265	7607	20086	77163	641121	1471974	4177295
9	58	295	900	2486	11266	21072	77022	503645	1498351	4870697
10	69	310	784	3490	8406	29598	57841	418072	1391725	3940158

Quantitative Data

Double Hashing										
	1000	2000	3000	4000	5000	6000	7000	8000	9000	10000
1	51	284	870	2239	10352	47838	78073	192083	462182	1072445
2	58	292	949	3321	18317	31306	84295	184448	396196	1040028
3	65	269	844	3300	11428	31158	69789	203419	477556	1120198
4	60	297	808	1989	8970	24281	78492	165379	441144	1106580
5	50	251	1059	2299	13720	32118	67519	166596	409646	1187518
6	65	311	1068	2204	9692	36845	68381	188425	461190	1088529
7	64	323	815	2772	6676	41129	80062	161644	451769	987346
8	63	296	934	3460	6090	22341	65388	143551	393872	1243163
9	66	344	1112	3284	10063	41514	80084	187004	431022	1100814
10	51	335	862	2256	8579	31672	78849	145391	451772	1166834

Quantitative Data



Thank You

Hash Table