Open addressing

- In case of open addressing every element of the hash table is inside the table, we have no pointers, no next links.

- When we want to insert a new element, we will successively generate positions for the element, check (*probe*) the generated positions, and place the element in the first available one.

Open addressing

- In order to generate multiple positions, we will extend the hash function and add to it another parameter, i, which is the *probe number* and starts from 0.

$$h : U \times \{0, 1, ..., m-1\} \rightarrow \{0, 1, ..., m-1\}$$

- For an element k, we will successively examine the positions $< h(k, 0), h(k, 1), h(k, 2), ..., h(k, m-1) >$ - called the *probe sequence*
- The *probe sequence* should be a permutation of the hash table positions $\{0, ..., m-1\}$, so that eventually every slot is considered.
- We would also like to have a hash function which can generate all the $m!$ possible permutations (spoiler alert: we cannot)

Open addressing - Linear probing

- One version of defining the hash function is to use linear probing:

$$h(k, i) = (h'(k) + i) \ mod \ m \ \ \forall i = 0, ..., m - 1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$)

- the *probe sequence* for linear probing is: $< h'(k), h'(k) + 1, h'(k) + 2, ..., m - 1, 0, 1, ..., h'(k) - 1 >$

Open addressing - Linear probing - example

- Consider a hash table of size $m = 16$ that uses open addressing and linear probing for collision resolution
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.
- Let's compute the value of the hash function for every key for i = 0:

Key	76	12	109	43	22	18	55	81	91	27	13	16	39
Hash	12	12	13	11	6	2	7	1	11	11	13	0	7

Open addressing - Linear probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
27	81	18	13	16		22	55	39			43	76	12	109	91

- Disadvantages of linear probing:
    - There are only *m* distinct probe sequences (once you have the starting position everything is fixed)
    - *Primary clustering* - long runs of occupied slots
- Advantages of linear probing:
    - Probe sequence is always a permutation
    - Can benefit from caching

Open addressing - Linear probing - primary clustering

- Why is primary clustering a problem?
- Assume m positions, n elements and $\alpha = 0.5$ (so $n = m/2$)
- Best case arrangement: every second position is empty (for example: even positions are occupied and odd ones are free)
- What is the average number probes (positions verified) that need to be checked to insert a new element?

- Worst case arrangement: all n elements are one after the other (assume in the second half of the array)
- What is the average number of probes (positions verified) that need to be checked to insert a new element?

- In case of quadratic probing the hash function becomes:

$$h(k, i) = (h'(k) + c_1 * i + c_2 * i^2) \ mod \ m \quad \forall i = 0, ..., m-1$$

- where $h'(k)$ is a *simple* hash function (for example: $h'(k) = k \ mod \ m$) and c_1 and c_2 are constants initialized when the hash function is initialized. c_2 should not be 0.

- Considering a simplified version of $h(k, i)$ with $c_1 = 0$ and $c_2 = 1$ the probe sequence would be: $< k, k+1, k+4, k+9, k+16, ... >$

Open addressing - Quadratic probing

- One important issue with quadratic probing is how we can choose the values of m, c_1 and c_2 so that the probe sequence is a permutation.

- If m is a prime number only the first half of the probe sequence is unique, so, once the hash table is half full, there is no guarantee that an empty space will be found.
  - For example, for $m = 17$, $c_1 = 3$, $c_2 = 1$ and $k = 13$, the probe sequence is $< 13, 0, 6, 14, 7, 2, 16, 15, 16, 2, 7, 14, 6, 0, 13, 11, 11 >$
  - For example, for $m = 11$, $c_1 = 1$, $c_2 = 1$ and $k = 27$, the probe sequence is $< 5, 7, 0, 6, 3, 2, 3, 6, 0, 7, 5 >$

Open addressing - Quadratic probing

- If m is a power of 2 and $c_1 = c_2 = 0.5$, the probe sequence will always be a permutation. For example for $m = 8$ and $k = 3$:
  - $h(3,0) = (3 \% 8 + 0.5 * 0 + 0.5 * 0^2) \% 8 = 3$
  - $h(3,1) = (3 \% 8 + 0.5 * 1 + 0.5 * 1^2) \% 8 = 4$
  - $h(3,2) = (3 \% 8 + 0.5 * 2 + 0.5 * 2^2) \% 8 = 6$
  - $h(3,3) = (3 \% 8 + 0.5 * 3 + 0.5 * 3^2) \% 8 = 1$
  - $h(3,4) = (3 \% 8 + 0.5 * 4 + 0.5 * 4^2) \% 8 = 5$
  - $h(3,5) = (3 \% 8 + 0.5 * 5 + 0.5 * 5^2) \% 8 = 2$
  - $h(3,6) = (3 \% 8 + 0.5 * 6 + 0.5 * 6^2) \% 8 = 0$
  - $h(3,7) = (3 \% 8 + 0.5 * 7 + 0.5 * 7^2) \% 8 = 7$

Open addressing - Quadratic probing

- If m is a prime number of the form $4 * k + 3$, $c_1 = 0$ and $c_2 = (-1)^i$ (so the probe sequence is $+0$, -1, $+4$, -9, etc.) the probe sequence is a permutation. For example for $m = 7$ and $k = 3$:
  - $h(3, 0) = (3 \% 7 + 0^2) \% 7 = 3$
  - $h(3, 1) = (3 \% 7 - 1^2) \% 7 = 2$
  - $h(3, 2) = (3 \% 7 + 2^2) \% 7 = 0$
  - $h(3, 3) = (3 \% 7 - 3^2) \% 7 = 1$
  - $h(3, 4) = (3 \% 7 + 4^2) \% 7 = 5$
  - $h(3, 5) = (3 \% 7 - 5^2) \% 7 = 6$
  - $h(3, 6) = (3 \% 7 + 6^2) \% 7 = 4$

Open addressing - Quadratic probing - example

- Consider a hash table of size $m = 16$ that uses open addressing with quadratic probing for collision resolution (h'(k) is a hash function defined with the division method), $c_1 = c_2 = 0.5$.
- Insert into the table the following elements: 76, 12, 109, 43, 22, 18, 55, 81, 91, 27, 13, 16, 39.

Open addressing - Quadratic probing - example

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
13	81	18	16		91	22	55	39		27	43	76	12	109	

- Disadvantages of quadratic probing:
  - The performance is sensitive to the values of m, c_1 and c_2.
  - *Secondary clustering* - if two elements have the same initial probe positions, their whole probe sequence will be identical: $h(k_1, 0) = h(k_2, 0) \Rightarrow h(k_1, i) = h(k_2, i)$.
  - There are only m distinct probe sequences (once you have the starting position the whole sequence is fixed).

Open addressing - Double hashing

- In case of double hashing the hash function becomes:

$$h(k, i) = (h'(k) + i * h''(k)) \ \% \ m \ \forall i = 0, ..., m-1$$

- where $h'(k)$ and $h''(k)$ are *simple* hash functions, where $h''(k)$ should never return the value 0.

- For a key, k, the first position examined will be $h'(k)$ and the other probed positions will be computed based on the second hash function, $h''(k)$.

Open addressing - Double hashing

- Similar to quadratic probing, not every combination of m and $h''(k)$ will return a complete permutation as a probe sequence.

- In order to produce a permutation m and all the values of $h''(k)$ have to be relatively primes. This can be achieved in two ways:
  - Choose m as a power of 2 and design h'' in such a way that it always returns an odd number.
  - Choose m as a prime number and design h'' in such a way that it always returns a value from the $\{0, \text{m-1}\}$ set (actually $\{1, \text{m-1}\}$ set, because $h''(k)$ should never return 0).

Open addressing - Double hashing

- Choose m as a prime number and design h'' in such a way that it always return a value from the $\{0, \text{m-1}\}$ set.
- For example:
  $h'(k) = k\%m$
  $h''(k) = 1 + (k\%(m-1))$.
- For $m = 11$ and $k = 36$ we have:
  $h'(36) = 3$
  $h''(36) = 7$
- The probe sequence is: $< 3, 10, 6, 2, 9, 5, 1, 8, 4, 0, 7 >$

- Consider a hash table of size $m = 17$ that uses open addressing with double hashing for collision resolution, with $h'(k) = k\%m$ and $h''(k) = (1 + (k\%16))$.
- Insert into the table the following elements: 75, 12, 109, 43, 22, 18, 55, 81, 92, 27, 13, 16, 39.
- Values of the two hash functions for each element:

key	75	12	109	43	22	18	55	81	92	27	13	16	39
h' (key)	7	12	7	9	5	1	4	13	7	10	13	16	5
h''(key)	12	13	14	12	7	3	8	2	13	12	14	1	8

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
16	18		55	109	22		75		43	27	39	12	81		13	92

- Main advantage of double hashing is that even if $h(k_1, 0) = h(k_2, 0)$ the probe sequences will be different if $k_1 \neq k_2$.
- For example:
  - 75: $< 7, 2, 14, 9, 4, 16, 11, 6, 1, 13, 8, 3, 15, 10, 5, 0, 12 >$
  - 109: $< 7, 4, 1, 15, 12, 9, 6, 3, 0, 14, 11, 8, 5, 2, 16, 13, 10 >$
- Since for every $(h'(k), h''(k))$ pair we have a separate probe sequence, double hashing generates $\approx m^2$ different permutations.

Open addressing - operations

- In the following we will discuss the implementation of some of the basic dictionary operations for collision resolution with open addressing.

- We will use the notation $h(k, i)$ for a hash function, without mentioning whether we have linear probing, quadratic probing or double hashing (code is the same for each of them, implementation of h is different only).

Open addressing - representation

- What fields do we need to represent a hash table with collision resolution with open addressing?

HashTable:
    T: TKey[]
    m: Integer
    h: TFunction

- For simplicity we will consider that we only have keys.

Open addressing - insert

- What should the *insert* operation do?

```
subalgorithm insert (ht, e) is:
//pre: ht is a HashTable, e is a TKey
//post: e was added in ht
   i ← 0
   pos ← ht.h(e, i)
   while i < ht.m and ht.T[pos] ≠ -1 execute
   //-1 means empty space
      i ← i + 1
      pos ← ht.h(e, i)
   end-while
   if i = ht.m then
      @resize and rehash and compute the position for e again
   else
      ht.T[pos] ← e
   end-if
end-subalgorithm
```

Open addressing - other operations

- What should the *search* operation do?
- How can we *remove* an element from the hash table?
- Removing an element from a hash table with open addressing is not simple:
  - we cannot just mark the position empty - *search* might not find other elements
  - you cannot move elements - *search* might not find other elements
- Remove is usually implemented to mark the deleted position with a special value, *DELETED*.
- How does this special value change the implementation of the *insert* and *search* operation?

Open addressing - Performance

- In a hash table with open addressing with load factor $\alpha = n/m$ ($\alpha < 1$), the *average* number of probes is at most
  - for *insert* and *unsuccessful search*

$$\frac{1}{1-\alpha}$$

  - for *successful search*

$$\frac{1}{\alpha} * \ln \frac{1}{1-\alpha}$$

- If α is constant, the complexity is $\Theta(1)$
- Worst case complexity is $\Theta(n)$