Data Structures and Algorithms

Dr. L. Rajya Lakshmi

Open Addressing

- We have to maintain a large number of pointers
- Also space is wasted
- There is another collision resolution technique that uses the space in the hash table only

Open addressing

- All items occupy the hash table itself (no separate lists maintained)
- Saves memory space; can use that space for defining a large hash table; reduces collisions
- For each item a sequence of buckets are searched
- Do not probe buckets in sequence 0, 1, . . ., N-1
- The buckets probed depend upon the key of the item
- Modified hash function h, which takes the probe number as an argument in addition to hash key
- The probe sequence for key k (assumption: k is an integer) is: h(k, 0), h(k, 1), . . ., h(k, N-1)
- It is a permutation of 0, 1, . . ., N-1

Open addressing

- Assume that items are keys themselves and key are integers
- Each bucket contains either an item/a key or NIL

```
Algorithm Hash_Insert(T, k)
    i ← 0
    while i < N do
        j ← h(k, i)
        if T[j] is NIL
            T[j] ← k
            return j
        else i ← i+1
    raise an error “hash table overflow”
```

Open addressing

- The search algorithm probes the same sequence
- It encounters an empty bucket (unsuccessful search) or bucket having the search key

```
Algoritm Hash_Search(T, k)
        i ← 0
        j ← h(k, i)
        while i < N and T[j] is not NIL
                if T[j] == k
                        return j
                i ← i+1
                j ← h(k, i)
        return NIL
```

Open addressing

- Cannot simply delete the keys?
- If the buckets i1, i2, i3, . . . were probed while inserting key k into the hash table and finally inserted k into the bucket j
- If the key stored in bucket i3 is deleted, the search algorithm would return a wrong output
- Can be handled by marking these buckets differently, say storing “DELETED” instead of NIL

Open addressing

```
Algoritm Hash_Delete(T, k)
	i ← 0
	j ← h(k, i)
	while i < N and T[j] is not NIL
		if T[j] == k
			T[j] ← DELETED
			return j
		i ← i+1
		j ← h(k, i)
	return NIL
```

Open addressing

- Three techniques are commonly used to compute probe sequences
  - Linear probing
  - Quadratic probing
  - Double hashing
- All three guarantee that the probe sequence is a permutation of 0, 1, . . ., N-1

Linear probing

- Assume that h’ is the hash function used for compression mapping (auxiliary hash function)
- Hash function used by linear probing is:

  $h(k, i) \leftarrow (h'(k) + i) \bmod N$, for $i=0, 1, \ldots, N-1$
- We first probe the bucket given by the auxiliary hash function, followed by other buckets

Linear probing

- h'(k) ← k mod N; N = 10; h(k) = (h'(k) + i) mod N
- {89, 18, 49, 58, 69}

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Linear probing

- Suffers from a problem called primary clustering
- Blocks of occupied buckets start forming (primary clustering)
- Issue?
- Any key that hashes into the cluster, will join that cluster after several attempts to resolve the collision

Quadratic probing

- Assume that h’ is the hash function used for compression mapping
- Hash function used by quadratic probing is:

  $h(k, i) \leftarrow (h'(k) + c_1 i + c_2 i^2) \bmod N$, where c_1 and c_2 are auxiliary constants, $i=0, 1, \ldots, N-1$
- We first probe the bucket given by the auxiliary hash function, followed by other buckets provided by the hash function

Quadratic probing

- $h'(k) \leftarrow k \bmod N$; $N = 10$; $h(k, i) = (h'(k) + i^2) \bmod N$
- {89, 18, 49, 58, 69}

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Quadratic probing

- If two keys hash to the same initial bucket, then their probe sequences are the same; h(k1, 0) = h(k2, 0) implies h(k1, i) = h(k2,i)
- Same alternative buckets are probed
- Suffers from a problem called secondary clustering

Double hashing

- It uses hash function of the form

  $h(k, i) = (h'(k) + i\, h''(k)) \bmod N$,

  h' and h'' are auxiliary hash functions and $i = 0, 1, \ldots, N-1$
- The initial probe goes to position $T[h'(k)]$
- The successive attempts to resolve the collision will probe the positions that are offset from the previous positions by the amount $h''(k) \bmod N$
- The probe sequence depends upon the key in two ways

Double hashing

- The value of $h''(k)$ must be relatively prime to the table size in order to probe the entire table
- Can select N as a power of 2 and select h'' so that it always produces an odd number
- Can select N as a prime, and design h'' so that it always returns a positive integer less than N
- Another common choice is: N is prime, N' is a prime smaller than N, and $h''(k) = N' - (k \bmod N')$
- Ex: can select N as 13, then

  $h'(k) = k \bmod 13$

  $h''(k) = 1 + k \bmod N'$,

  where N' is chosen slightly lower than N, say 11

Double hashing

- $N = 13$, $N' = 11$, $h'(k) \leftarrow k \bmod N$; $h''(k) = 1 + k \bmod N'$
- $h(k) = (h'(k) + i\, h''(k)) \bmod N$
- Insert 14