CME 2001
Data Structures and Algorithms

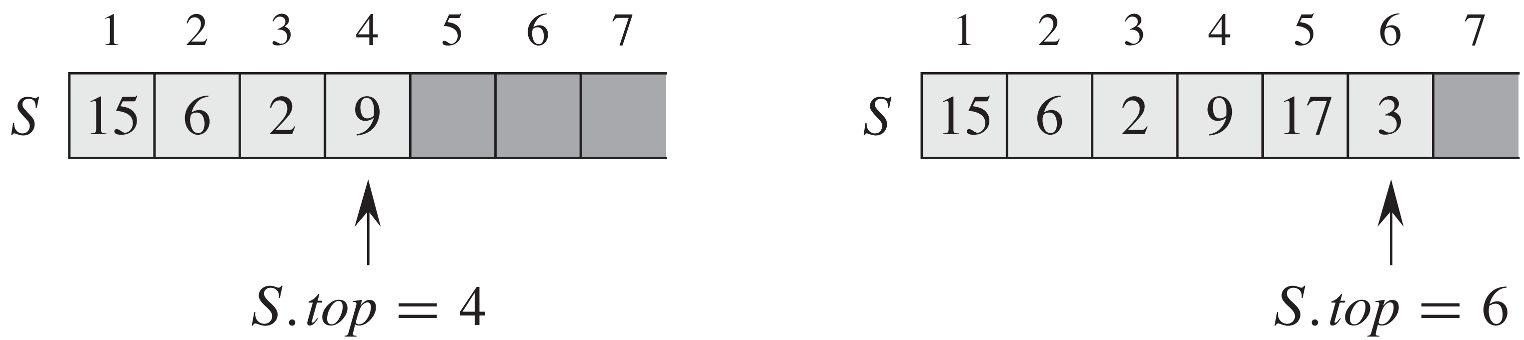
Zerrin Işık
zerrin@cs.deu.edu.tr

Elementary Data Structures

Stacks

- Implements last-in, first-out (LIFO) policy.
- Insertion of a new element is performed by PUSH operation => placed to the top
- The top element is removed by POP operation => last one is removed
- A stack of at most ***n*** elements can be implemented with an array **S** [1…n].
- Array attribute ***S.top*** indexes the most recently inserted element.

Stacks



- S [1] => bottom element
- S [S.top] => top (last) element

Stack Operations

- Implement stack operations with limited running times.

STACK-EMPTY(S)
```
if S.top == 0
    return TRUE
else return FALSE
```

PUSH(S, x)
```
S.top = S.top + 1
S[S.top] = x
```

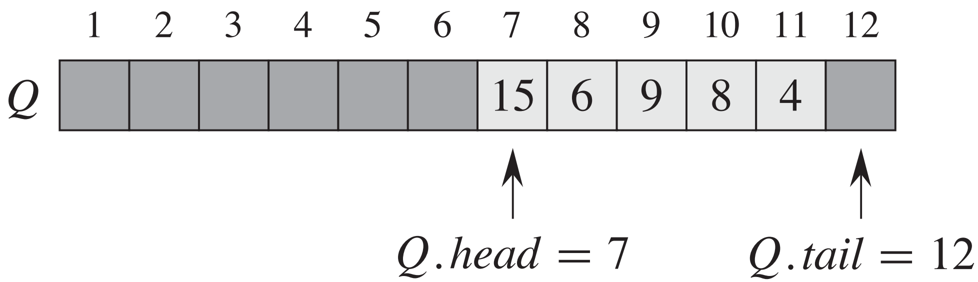
POP(S)
```
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

- Each stack operation takes only O(1) time.

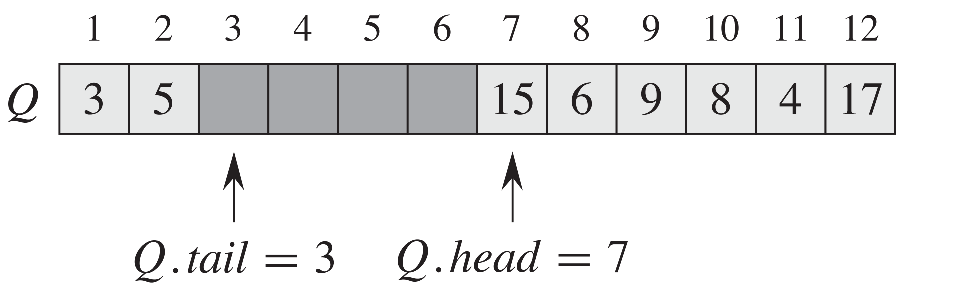
Queues

- Implements first-in, first-out (FIFO) policy.
- It has a head and tail.
- Insertion of a new element is performed by ENQUEUE operation => placed to the tail
- The first element is removed by DEQUEUE => removed from the head
- A queue of at most ***n-1*** elements can be implemented with an array **Q** [1…n]. Because initially $Q.head = Q.tail = 1$.
- When $Q.head = Q.tail$, the queue is empty.
- When $Q.head = Q.tail+1$, the queue is full.

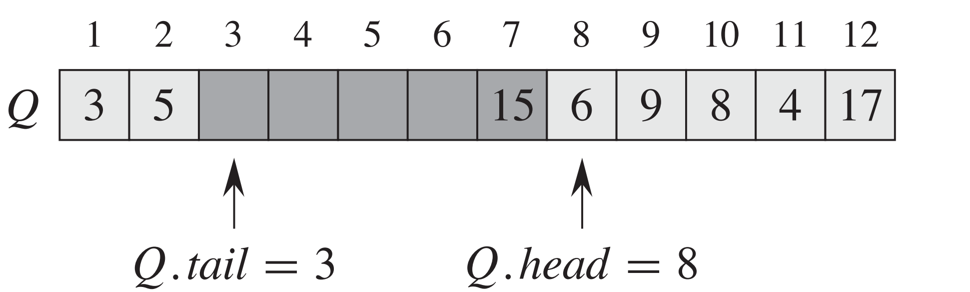
Queues



- Q has 5 elements.



- After adding of 17,3,5 elements to Q.



- After removing 15. The new head has key 6.

Queue Operations

$\text{ENQUEUE}(Q, x)$

$Q[Q.tail] = x$
if $Q.tail == Q.length$
$\quad\quad Q.tail = 1$
else $Q.tail = Q.tail + 1$

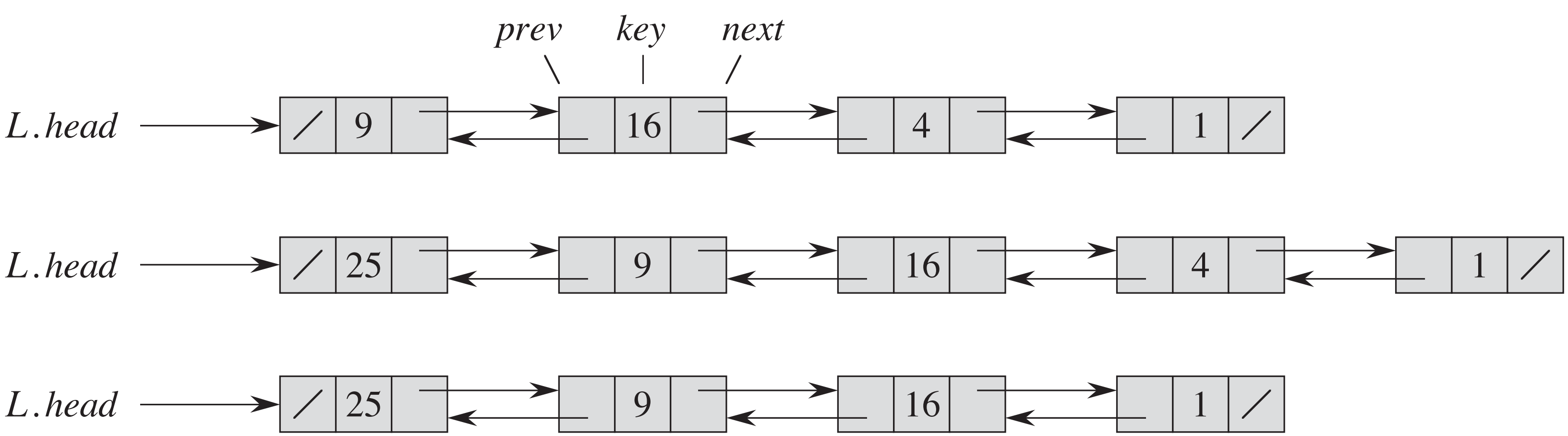
$\text{DEQUEUE}(Q)$

$x = Q[Q.head]$
if $Q.head == Q.length$
$\quad\quad Q.head = 1$
else $Q.head = Q.head + 1$
return x

- Each queue operation takes only O(1) time.

Linked Lists

- Objects are arranged in a linear order
- Order is determined by a pointer in each object
- Simple and flexible representation for dynamic sets (i.e., not known size).



Linked List – Search Operation

```
LIST-SEARCH(L, k)
x = L.head
while x ≠ NIL and x.key ≠ k
    x = x.next
return x
```

- Finds the first element with key ***k*** in list ***L*** by a linear search.
- It takes Θ(n) in the worst-case, since it might look the entire list.

Linked List – Insert Operation

```
LIST-INSERT(L, x)
x.next = L.head
if L.head ≠ NIL
    L.head.prev = x
L.head = x
x.prev = NIL
```

- Given element ***x*** is spliced (i.e., connect) to the head of the list.
- It takes O(1) time.

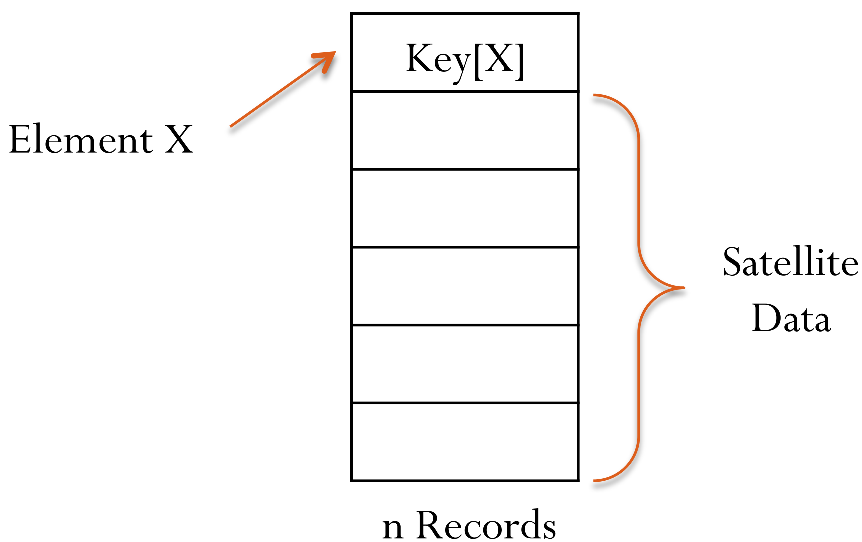
Linked List – Delete Operation

```
LIST-DELETE(L, x)
if x.prev ≠ NIL
        x.prev.next = x.next
else L.head = x.next
if x.next ≠ NIL
        x.next.prev = x.prev
```

- For a given element ***x***, delete operation takes O(1) time.
- BUT if we will delete an element with a given “key”, delete operation takes Θ(n) time in the worst case.
- Why?

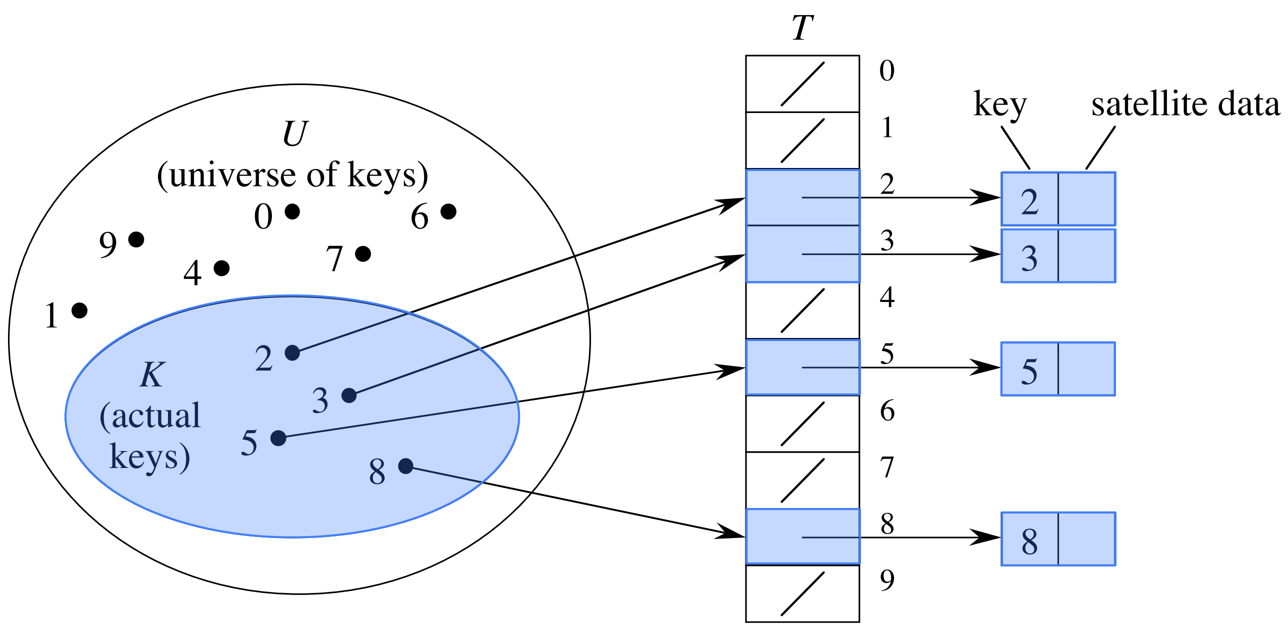
Hash Tables

- Many applications require a dynamic set that supports only the ***dictionary operations*** INSERT, SEARCH, DELETE.
- E.g., A symbol table in a compiler.
- A hash table is effective structure to implement a dictionary.
- The expected search time is O(1), however, it could be Θ(n) in the worst-case.



Direct-Address Tables

- Each element has a key drawn from a set **U** $=\{0,1,\ldots,m\}$ where **m** isn't too large.
- No two elements have the same key.
- Represent by a ***direct-address table***, or array, **T**[0…m-1].
- Each ***slot*** corresponds to a key in **U**.
- If there's an element **x** with key **k**:
  - then **T**[k] contains a pointer to **x**.
  - otherwise, **T**[k] is empty, represented by NIL.



Direct-Address Operations

$\text{DIRECT-ADDRESS-SEARCH}(T, k)$

  return $T[k]$

$\text{DIRECT-ADDRESS-INSERT}(T, x)$

  $T[key[x]] = x$

$\text{DIRECT-ADDRESS-DELETE}(T, x)$

  $T[key[x]] = \text{NIL}$

- Each operation takes O(1) time.

Hash Tables

- The problem with direct addressing is that if the universe **U** is large, storing a table of size (|U|) might be impractical.
- Usually, the set of stored keys is small (compared to U), so that most of the space allocated for T is wasted.

Idea:

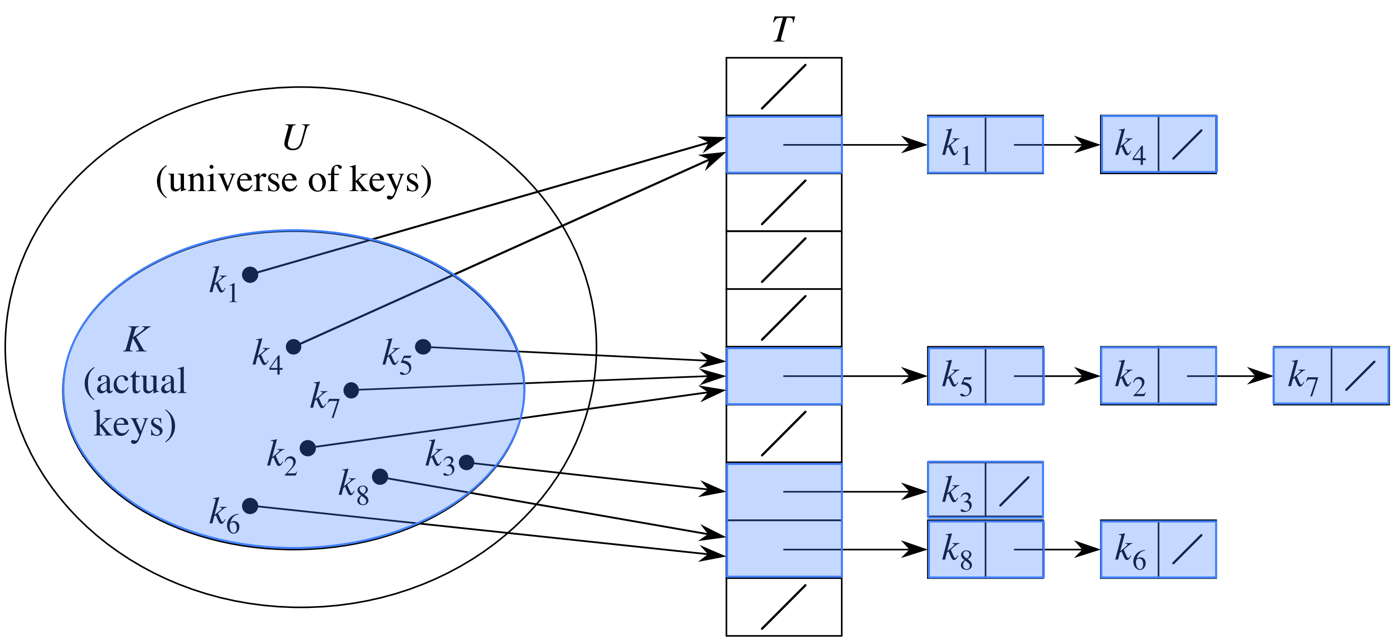
- Instead of storing an element with key ***k*** in slot k, use a function ***h*** and store the element in slot ***h(k)***.
- ***h*** is called as ***hash function***.
- ***h***: U → {0,1,…,m-1}, so that ***h(k)*** is a legal slot number in **T**.
- *k* ***hashes*** to slot ***h(k)***.

Collisions

- When two or more keys hash to the same slot, a collision occurs.
- Can happen when there are more possible keys than available slots ($|U| > m$).
- Therefore, must be prepared to handle collisions in all cases.
- Two methods are used:
  - Chaining
  - Open addressing

Collision Resolution by Chaining

- Place all elements that hash to the same slot into a linked list.
- Slot **j** contains a pointer to the head of the list of all stored elements that hash to **j**.
- If there are no such elements, slot **j** contains NIL.



Dictionary Operations with Chaining

Insertion:

CHAINED-HASH-INSERT (T, x)

insert ***x*** at the head of list ***T*** *[h(x.key)]*

- Worst-case running time is O(1).
- Assumes that the element being inserted isn't already in the list.
- It might need an extra search to check if it was already inserted.

Search:

CHAINED-HASH-SEARCH (T, k)

search for an element with key ***k*** in list ***T*** *[h(k)]*

- Running time is proportional to the length of the list of elements in slot h(k).

Dictionary Operations with Chaining

<u>Deletion:</u>

CHAINED-HASH-DELETE (T,x)

delete ***x*** from the list ***T*** *[h(x.key)]*

- Given pointer ***x*** to the element to delete, no search is needed to find this element.
- Worst-case running time is O(1) time if the lists are doubly linked.
- If the lists are singly linked, deletion takes as long as searching, because we must find x's predecessor in its list to correctly update *next* pointers.

Analysis of Hashing with Chaining …

- Given a key, how long does it take to find an element with that key, or to determine that there is no element with that key?
- Analysis is in terms of the ***load factor*** $\alpha = n/m$:
  - ***n*** : # of elements in the table.
  - ***m*** : # of slots in the table (or # of possibly empty linked lists).
  - Load factor is average number of elements per linked list.
  - Can have $\alpha < 1$, $\alpha = 1$, or $\alpha > 1$.
- Worst-case: when all n keys hash to the same slot => get a single list of length n => worst-case time to search : $\Theta(n)$ + time to compute hash function.
- Average-case depends on how well the hash function distributes the keys among the slots.

Analysis of Hashing with Chaining …

Lets focus on average-case performance of hashing with chaining.

- ***Simple uniform hashing***: any given element is equally likely to hash into any of the **m** slots.
- For j =0,1,…,m-1, denote the length of list T[j] by n_j.
  Then $n = n_0 + n_1 + \ldots + n_{m-1}$.
- Average value of n_j is $E[n_j] = \alpha = n/m$
- Assume that we can compute the hash function in O(1) time, so the time required to search for the element with key ***k*** depends on the length $n_{h(k)}$ of the list T[h(k)].

Analysis of Hashing with Chaining …

If the hash table contains no element with key k, then the search is *unsuccessful*. It takes expected time $\Theta(1+\alpha)$.

Proof: Simple uniform hashing => any key not already in the table is equally likely to hash to any of the ***m*** slots.

- To search unsuccessfully for any key ***k***, need to search to the end of the list T[h(k)], its expected length = α. So, the expected number of elements examined in an unsuccessful search is α.
- When the time to compute the hash function is added (O(1)), the total time required is $\Theta(1+\alpha)$.

Analysis of Hashing with Chaining …

If the hash table contains an element with key ***k***, then the search is *successful*.

- The circumstances are slightly different from an unsuccessful search.
- The probability that each list is searched is proportional to the number of elements it contains.
- A successful search takes expected time $\Theta(1+\alpha)$. It has the same asymptotic bound with unsuccessful search (proof is given on page 260).

Hash Functions

- What makes a good hash function?
  - Uniformly distribute the keys into slots

- In practice: not possible to satisfy this rule because:
  - unknown probability distribution that keys are drawn from
  - keys might not drawn independently

- Use heuristics, based on the domain of the keys, to create a hash function that performs well.

Division method

h(k) = k mod m

e.g. m=20 and k=91 => h(k) = 11

Pros: Fast, requires only one division operation.

Cons: Should avoid certain values of ***m***

- Powers of 2 are bad choices. If m $=2^p$ for integer *p*, then *h(k)* is just the least significant *p* bits of *k*.
- If *k* is a character string interpreted in radix 2^p, then m = 2^p -1 is bad choice: permuting characters in a string does not change its hash value.
- Good choice of *m* : *A prime number*, but not too close to an exact power of 2 or 10.

Multiplication method

1. Choose constant $\boldsymbol{A}$ in the range $0 < \boldsymbol{A} < 1$.
2. Multiply key $\boldsymbol{k}$ by $\boldsymbol{A}$.
3. Extract the fractional part of kA.
4. Multiply the fractional part by $\boldsymbol{m}$.
5. Take the floor of the result.

$h(k) = \lfloor m\,(k\,A \bmod 1) \rfloor$, where $k\,A \bmod 1 = kA - \lfloor kA \rfloor$

Pros: Value of $\boldsymbol{m}$ is not critical.

Cons: Slower than division method.

How to choose $\boldsymbol{A}$: Knuth's suggestion => $\boldsymbol{A} \approx (\sqrt{5} - 1)/2$.

Collision Resolution by Open Addressing

Alternative method for handling collisions.

Idea:

- Store all keys in the hash table itself (needs a larger table)
- If a collision occurs, successfully examine (*probe*) hash table until an empty cell is found.

Hash Function => $h : U \times \underbrace{\{0, 1, \ldots, m-1\}}_{\text{probe number}} \to \underbrace{\{0, 1, \ldots, m-1\}}_{\text{slot number}}.$

Probe Sequence => $\langle h(k,0), h(k,1), \ldots, h(k, m-1) \rangle$

Open Addressing Operations

HASH-SEARCH(T, k)

$i = 0$
repeat
  $j = h(k, i)$
  if $T[j] == k$
    return j
  $i = i + 1$
until $T[j] ==$ NIL or $i = m$
return NIL

HASH-INSERT(T, k)

$i = 0$
repeat
  $j = h(k, i)$
  if $T[j] ==$ NIL
    $T[j] = k$
    return j
  else $i = i + 1$
until $i == m$
error "hash table overflow"

DELETION:

- Marked removed key with "DELETED" flag instead of NIL
- "Search" should treat DELETED as though the slot holds a key that does not match the one being searched for.
- "Insertion" should treat DELETED as though the slot were empty, so that it can be reused.

Probing Strategies

- Linear Probing
- Quadratic Probing
- Double Hashing

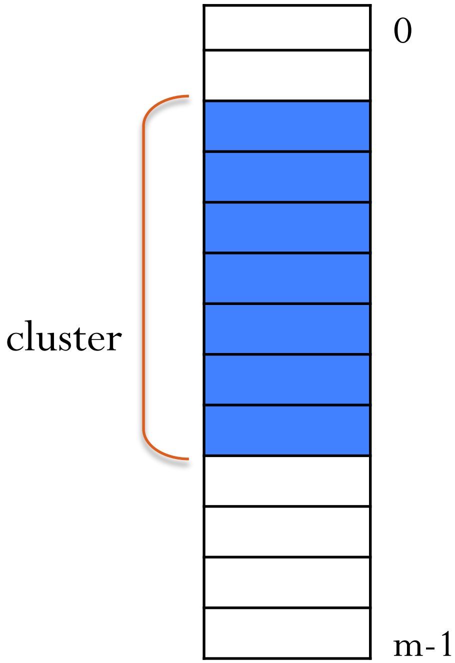
Linear Probing

$h(k,i) = (h`(k) + i) \bmod m$ (where $h`(k)$ is ordinary hash function)

i.e. the probe sequence starts at slot h`(k) and continues sequentially through the table, wrapping after slot m-1 to slot 0.

- Suffers from ***primary clustering***:

  => long runs of occupied sequences build up.
- Avg. search and insertion times increase.



Quadratic Probing

$h(k,i) = (h`(k) + c_1 i + c_2 i^2)$ mod m

where $c_1, c_2 \neq 0$ are constants, $i = 0, 1, \ldots, m-1$.

- Suffers from ***secondary clustering***:

  => if two distinct keys have the same h` value, then they have the same probe sequence.

Double Hashing

$h(k,i) = (h_1(k) + i.h_2(k)) \bmod m$

where $h_1(k)$ and $h_2(k)$ are ordinary hash functions.

E.g.:

$m=13$

$h_1(k) = k \bmod 13$

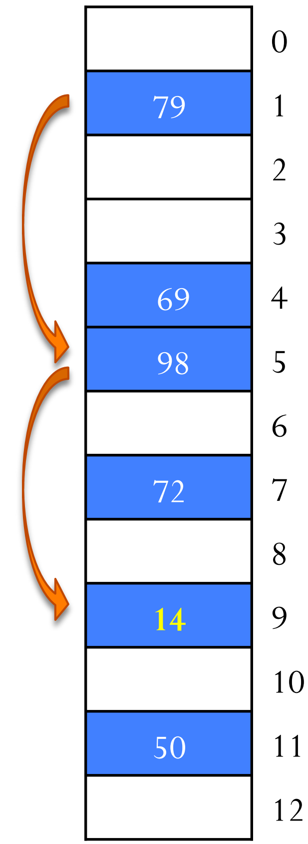
$h_2(k) = 1+(k \bmod 11)$

Insert key “14”

$h(k,0) = (h_1(14) + 0.h_2(14)) \bmod 13 = 1$ ✗

$h(k,1) = (h_1(14) + 1.h_2(14)) \bmod 13 = 5$ ✗

$h(k,2) = (h_1(14) + 2.h_2(14)) \bmod 13 = 9$ ✓



Rehashing

- The hash table will be inefficient, when it becomes full i.e., load factor gets larger, close to 1.
- What to do?
  - Replace hash table with a larger table, re-insert all items to new table with new hash function (i.e., rehashing).
  - New table size should be a *prime number*
- When rehashing should be applied?
  - If load factor > 0.5
  - Get an insertion fail

Rehashing example

- Insert 8, 25, 0, 13
- Linear probing: $h(x) = x \mod 5$
- $\alpha = 4/5 = 0.8$ => *Rehash*

25	0
0	1
	2
8	3
13	4

- New table size = 11
- $h(x) = x \mod 11$
- Insert 8, 25, 0, 13

0	0
	1
13	2
25	3
	4
	5
	6
	7
8	8
	9
	10

Rehashing cost

- Replace hash table with a larger table : O(1)
- Scan current table to fetch each item : O (1).n
- Re-insert all items to new table : O(n)

➦ Total running time: O(n) + O(n) = O(n)

- It is acceptable cost, since rehashing does no occur frequently

Analysis of Open Address Hashing

<u>Assumptions:</u>

- Analysis is in terms of load factor $\alpha = n/m$.
- Assume that the table never completely fills, so we always have $0 \leq \alpha \leq 1$.
- Assume uniform hashing.
- No deletion.
- In a successful search, each key is equally likely to be searched for.

<u>Theorem:</u>

The expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$.

Proof

- Define random variable **X**: # of probes made in an unsuccessful search.
- Define events A_i for $i = 1, 2, \ldots,$ to be the event that there is an i^{th} probe and that it is to an occupied slot.
- **X** $\geq$ i iff probes 1,2,…,i-1 are made and are to occupied slots =>

$$\Pr\{X \geq i\} = \Pr\{A_1 \cap A_2 \cap \cdots \cap A_{i-1}\}.$$

$$= \Pr\{A_1\} \cdot \Pr\{A_2 \mid A_1\} \cdot \Pr\{A_3 \mid A_1 \cap A_2\} \cdots \Pr\{A_{i-1} \mid A_1 \cap A_2 \cap \cdots \cap A_{i-2}\} .$$

$\Pr\{A_1\} = n/m$ ⇨ there are n stored keys and m slots, so the probability that the first probe is to an occupied slot is n/m.

Proof

$$\Pr\{X \geq i\} = \underbrace{\frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdots \frac{n-i+2}{m-i+2}}_{i-1 \text{ factors}} .$$

$n < m \Rightarrow (n-j)/(m-j) \leq n/m$ for $j \geq 0$, which implies

$$\begin{aligned}\Pr\{X \geq i\} &\leq \left(\frac{n}{m}\right)^{i-1} \\ &= \alpha^{i-1} .\end{aligned}$$

$$\begin{aligned}\mathrm{E}[X] &= \sum_{i=1}^{\infty} \Pr\{X \geq i\} \\ &\leq \sum_{i=1}^{\infty} \alpha^{i-1} \\ &= \sum_{i=0}^{\infty} \alpha^{i} \\ &= \frac{1}{1-\alpha}\end{aligned}$$

Interpretation :

If α is constant, an unsuccessful search takes O(1) time.

If α=0.5 => an unsuccessful search takes an avg. 2 probes.

If α=0.9 => it takes 10 probes.

Hash Table - Summary

- Used to implement the insert and find operations in *constant* average time.
  - it depends on the *load factor*
- It is important to have a prime table size, a correct choice of load factor and hash function.
- For separate chaining, the load factor should be close to 1.
- For open addressing, load factor should not exceed 0.5 unless this is completely unavoidable.
  - Rehashing can be implemented to grow (or shrink) the table.

Next Week Topics

- Heap Sort (Chapter 6)