CME 2001
Data Structures and Algorithms

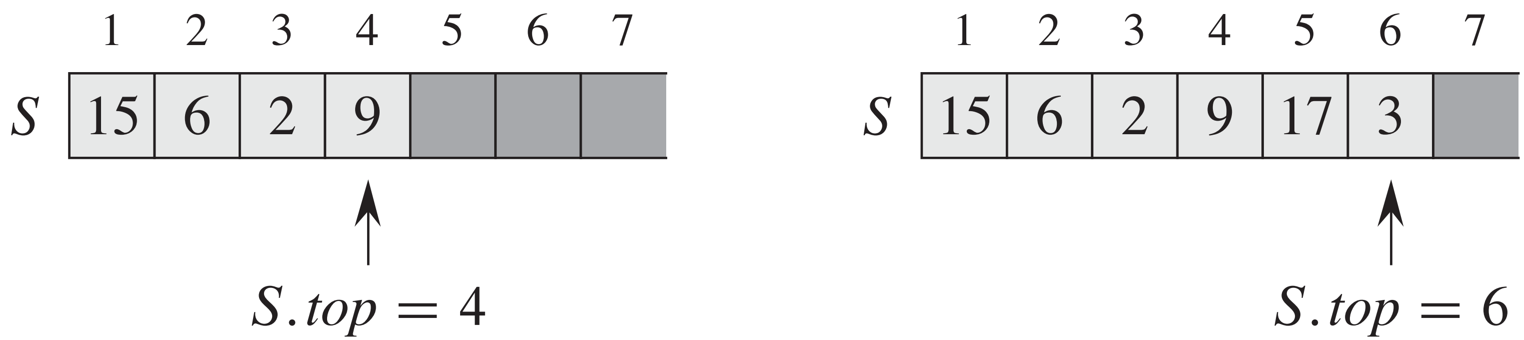
Zerrin Işık
zerrin@cs.deu.edu.tr

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
```

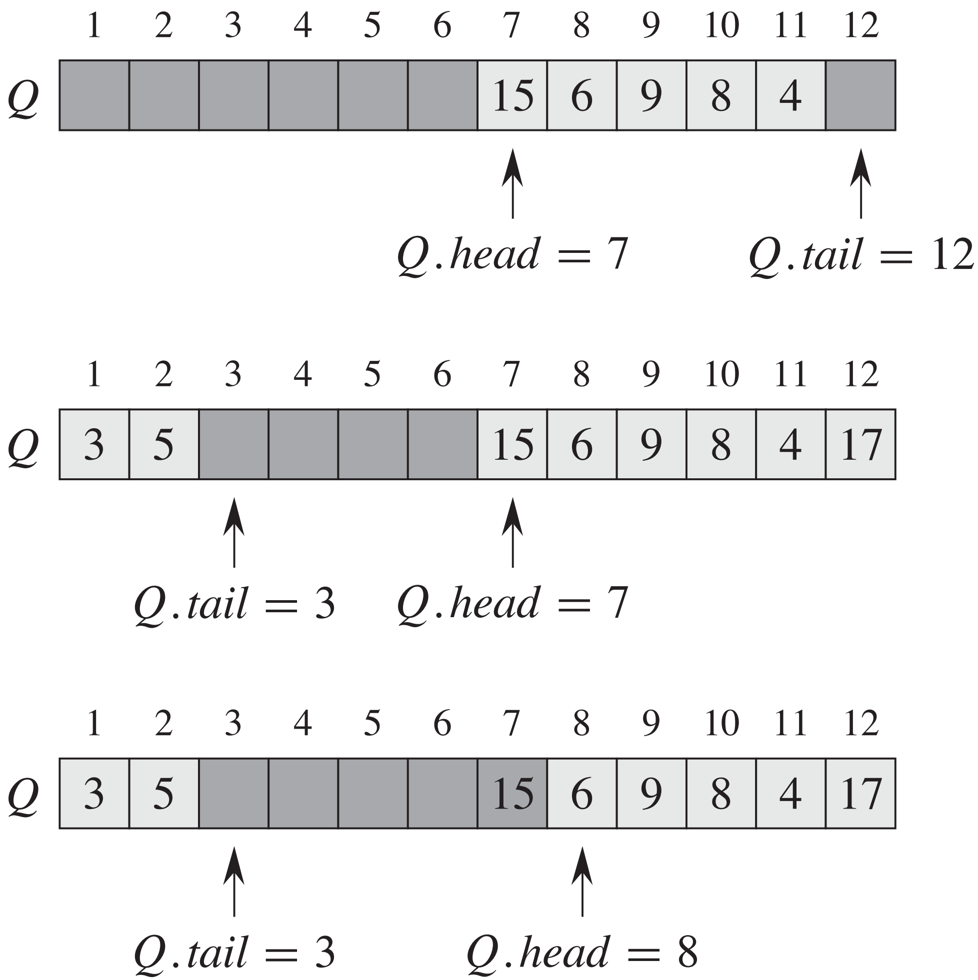
```
POP(S)
if STACK-EMPTY(S)
    error "underflow"
else S.top = S.top - 1
    return S[S.top + 1]
```

- 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

ENQUEUE(Q, x)

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

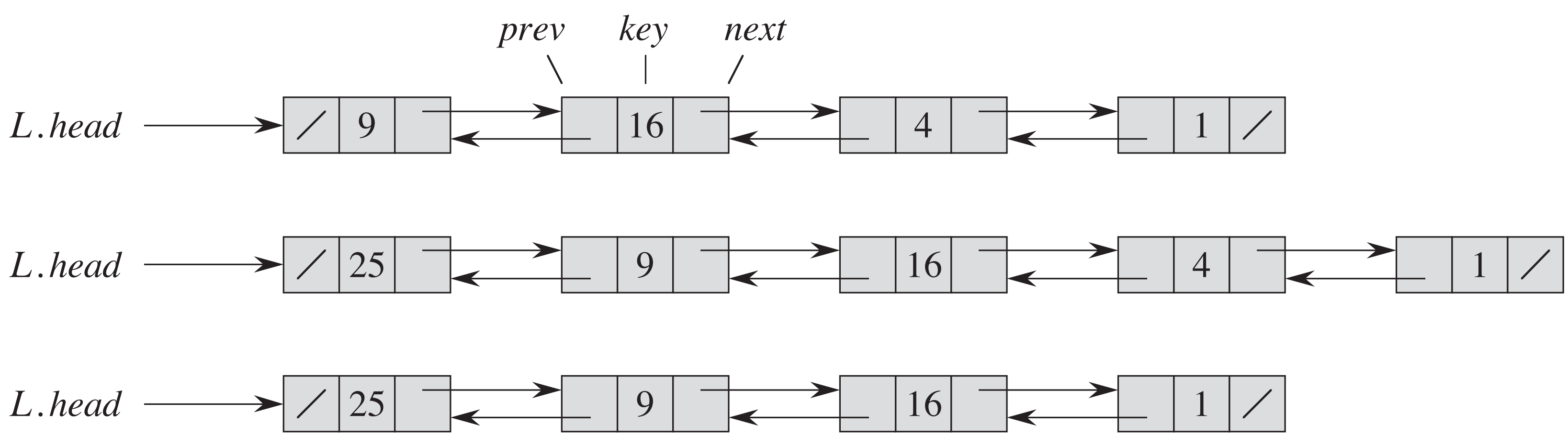
DEQUEUE(Q)

$x = Q[Q.head]$
if $Q.head == Q.length$
    $Q.head = 1$
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 $x \neq$ NIL and $x.key \neq k$
    $x = x.next$
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)

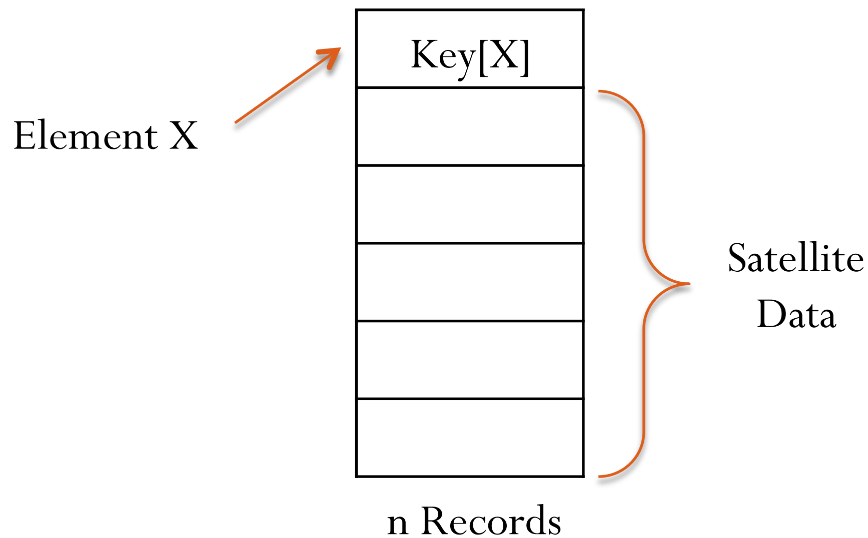
if $x.prev \neq$ NIL
    $x.prev.next = x.next$
else $L.head = x.next$
if $x.next \neq$ NIL
    $x.next.prev = x.prev$

- 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

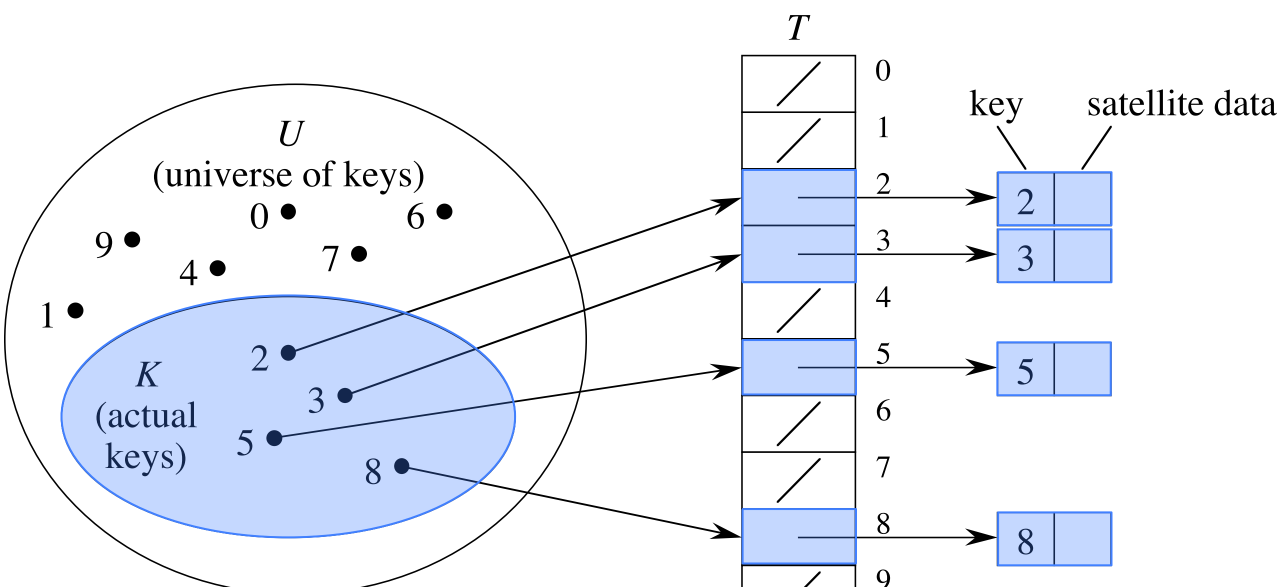
Hash Tables

- 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

DIRECT-ADDRESS-SEARCH(T, k)

return $T[k]$

DIRECT-ADDRESS-INSERT(T, x)

$T[key[x]] = x$

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:

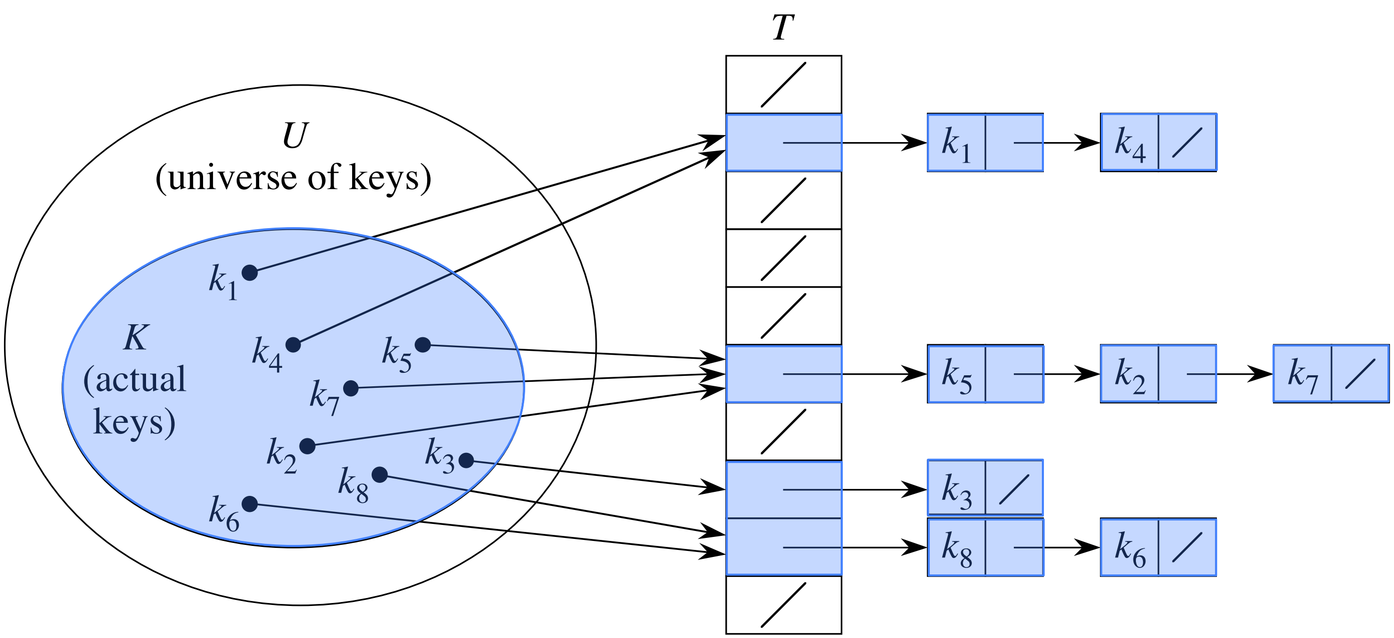
- Instead of storing an element with key ***k*** in slot k, use a function ***h*** and store the element in slot ***h(k)***.
- ***h*** is called as ***hash function***.
- ***h***: U → {0,1,…,m-1}, so that ***h(k)*** is a legal slot number in **T**.
- *k* ***hashes*** to slot ***h(k)***.

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

Deletion:

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.
  - 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 $\Theta(1+\alpha)$.

Analysis of Hashing with Chaining …

If the hash table contains an element with key ***k***, then the search is *successful*. It takes expected time $\Theta(1+\alpha)$.

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***

- 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.