Elementary Data Structures

Data Structures and Algorithms (094224)

Yuval Emek

Winter 2022/23

Data structure concepts

- Data structure = set of data elements (a.k.a. objects)
- The set is typically dynamic
- Often, each object is identified by a unique *key* ⟨מפתח⟩
  - An attribute (a.k.a. field) of the object
  - Sometimes the keys belong to a totally ordered set (e.g., $\mathbb{Z}$ or $\mathbb{Q}$)
- The data structure is defined by the set of operations it supports
  - *queries* ⟨שאילתות⟩: return information without changing the set
    - $\texttt{Search}(S, k)$, $\texttt{Minimum}(S)$, $\texttt{Successor}(\texttt{S}, \texttt{x})$
  - *modifying operations*: the set is changed
    - $\texttt{Insert}(S, x)$, $\texttt{Delete}(S, x)$
- Objects typically contain satellite data
  - Designated attribute(s)
  - Not accessed by the data structure's operations
- Efficiency considerations:
  - Run-time of each operation
  - Total space
  - Expressed as functions of n = current #objects

Array

Array ⟨מערך⟩

- Size declared at initialization
  - Cannot be modified afterwards
- Access any entry in $O(1)$ time

Linked list

Linked list ⟨רשימה מקושרת⟩

- Objects are arranged in a linear order
- Data structure's attribute:
  - *head* = pointer to first object in the list
- Object's attributes:
  - *key*
  - *next* = pointer to next object in the list
  - *prev* = pointer to previous object in the list (doubly linked list)
- Focus on doubly linked lists
  - Supported operations have the same asymptotic run-time

Doubly linked list operations

Search in list L for an object whose key is k (run-time $\Theta(n)$)

`List_Search`(L, k)

1: $x = L.head$
2: **while** $x \neq NIL$ and $x.key \neq k$ **do**
3: $\quad x = x.next$
4: return x

Insert object x to the front of list L (run-time $\Theta(1)$)

`List_Insert`(L, x)

1: $x.next = L.head$
2: **if** $L.head \neq NIL$ **then**
3: $\quad L.head.prev = x$
4: $L.head = x$
5: $x.prev = NIL$

Doubly linked list operations — cont.

Insert object x after object y in list L (run-time $\Theta(1)$)

`List_Insert_After(L, x, y)`

1: **if** $y.next \neq NIL$ **then**
2: $\quad y.next.prev = x$
3: $x.next = y.next$
4: $x.prev = y$
5: $y.next = x$

Remove object x from list L (run-time $\Theta(1)$)

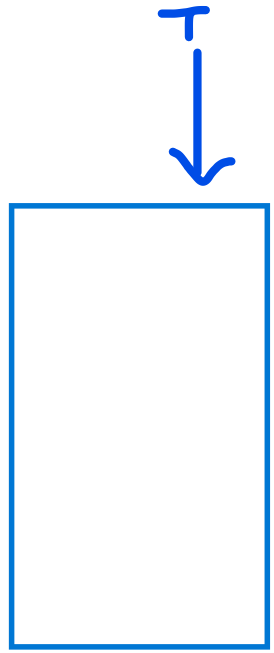
`List_Delete(L, x)`

1: **if** $x.prev \neq NIL$ **then**
2: $\quad x.prev.next = x.next$
3: **else** $L.head = x.next$
4: **if** $x.next \neq NIL$ **then**
5: $\quad x.next.prev = x.prev$

Stack

Stack ⟨מחסנית⟩

- Last in first out (LIFO) policy
- Data structure's attribute:
  - L = a (doubly) linked list, for implementation purposes only
- Can be implemented based on an array too
  - Bounded size



Stack operations

Insert object x at the top of stack S (run-time $\Theta(1)$)

`Push`(S, x)

1: $X.data = x$ ▷ new list object
2: `List_Insert`(`S.L`, `X`)

Remove the object at the top of stack S and return it (run-time $\Theta(1)$)

`Pop`(S)

1: **if** $S.L.head == NIL$ **then**
2: return ERROR
3: $x = S.L.head.data$
4: `List_Delete`$(S.L, S.L.head)$
5: return x

Queue

Queue ⟨תור⟩

- First in first out (FIFO) policy
- Data structure's attributes:
  - L = a (doubly) linked list, for implementation purposes only
  - *tail* = pointer to last object of L, for implementation purposes only
- Can be implemented based on an array too
  - Bounded size

Queue operations

Insert object x at the tail of queue Q (run-time $\Theta(1)$)

`Enqueue`(Q, x)

1: $X.data = x$ ▷ new list object
2: **if** $Q.tail \neq NIL$ **then**
3: `List_Insert_After`$(Q.L, X, Q.tail)$
4: **else** `List_Insert`$(Q.L, X)$
5: $Q.tail = X$

Queue operations — cont.

Remove the object at the head of queue Q and return it (run-time $\Theta(1)$)

`Dequeue`(Q)

1: **if** $Q.L.head == NIL$ **then**
2: return ERROR
3: $x = Q.L.head.data$
4: `List_Delete`($Q.L, Q.L.head$)
5: **if** $Q.L.head == NIL$ **then**
6: $Q.tail = NIL$
7: return x