ALGORITHMS AND DATA STRUCTURES № 8

Overview of different basic Abstract Data Structures

Why abstract? Because we define them by the operations they support, and not by their implementations. The same operations performed with use of different data structures may have different complexity.

1 Priority queues (and continuation of heaps)

- Maintains a dynamic set S of elements.
- Each set element has a *key* - an associated value.
- Max-priority queue supports dynamic-set operations:
  - INSERT(S, x): inserts element x into set S.
  - MAXIMUM(S): returns element of S with largest key
  - EXTRACT-MAX(S): removes and returns element of S with largest key.
  - INCREASE-KEY(S, x, k): increases value of element x's key to k. Assume k x's current key value.

Heaps efficiently implement priority queues. A heap gives a good compromise between fast insertion but slow extraction and vice versa. Both operations take $O(lgn)$ time.
Finding the maximum element: Getting the maximum element is easy: it's the root. Time: $\Theta(1)$.

```
HEAP-MAXIMUM(A)
return A[1]
```

Extracting max element: Given the array A:

- Make sure heap is not empty.
- Make a copy of the maximum element (the root).
- Make the last node in the tree the new root.
- Re-heapify the heap, with one fewer node.
- Return the copy of the maximum element.

Time: O(lg n).

```
HEAP-EXTRACT-MAX(A, n)
if n < 1
    then error ''heap underflow``
max := A[1]
A[1] := A[n]
MAX-HEAPIFY(A, 1, n - 1) // remakes heap
return max
```

Increasing key value: Given set S, element x, and new key value k:

- Make sure $k \geq x$'s current key.
- Update x's key value to k.
- Traverse the tree upward comparing x to its parent and swapping keys if necessary, until x's key is smaller than its parent's key.

```
HEAP-INCREASE-KEY(A,i, key)
if key < A[i]
    then error ''new key is smaller than current key``
A[i] := key
while i > 1 and A[PARENT(i)] < A[i]
    do
        exchange A[i] <-> A[PARENT(i)]
        i := PARENT(i)
```

Analysis: Upward path from node i has length $O(lgn)$ in an n-element heap. Time: $O(lgn)$
Inserting into the heap: Given a key k to insert into the heap:

- Insert a new node in the very last position in the tree with key $-\infty$.
- Increase the $-\infty$ key to k using the HEAP-INCREASE-KEY procedure defined above.

```
MAX-HEAP-INSERT(A, key, n)
A[n + 1] := - infty
HEAP-INCREASE-KEY(A, n + 1, key)
```

Analysis: constant time assignments + time for HEAP-INCREASE-KEY. Time: O(lg n).

Problem 1

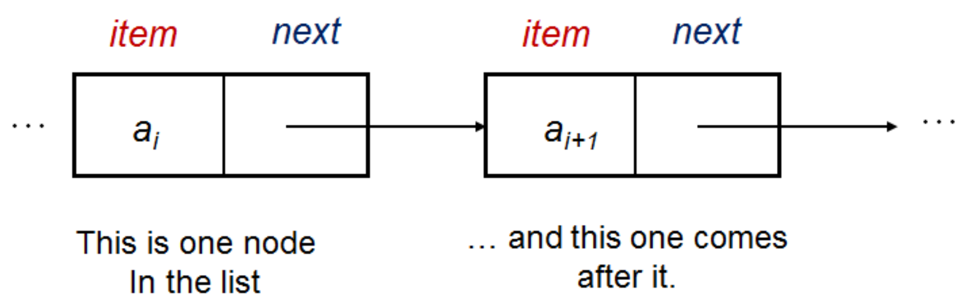
Illustrate the operation of HEAP-EXTRACT-MAX on the heap A =[15; 13; 9; 5; 12;8;7;4;0;6;2;1].

Problem 2

Illustrate the operation of MAX-HEAP-INSERT(A;10) on the heap A = [15;13;9; 5;12;8;7;4;0;6;2;1].

2 Lists

Linked list is a simple linear data structure. List is a sequence of items/data where positional order matter $(a_0, a_1, \ldots, a_{N-2}, a_{N-1})$.


item
next
item
next
a_i
a_{i+1}
This is one node In the list
... and this one comes after it.

Common List operations are:

- GET(i) – maybe a trivial operation, return a_i (0-based indexing),
- SEARCH(v) – decide if item/data v exists (and report its position/index) or not exist (and usually report a non existing index -1) in the list,
- INSERT(i, v) – insert item/data v specifically at position/index i in the list, potentially shifting the items from previous positions: [i..N-1] by one position to their right to make a space,
- REMOVE(i) – remove item that is specifically at position/index i in the list, potentially shifting the items from previous positions: [i+1..N-1] by one position to their left to close the gap.

Problem 3

What is the time complexity of the following list operations:

- GET(i)
- SEARCH(v)
- INSERT(i,v)
- REMOVE(i)

Explain shortly.

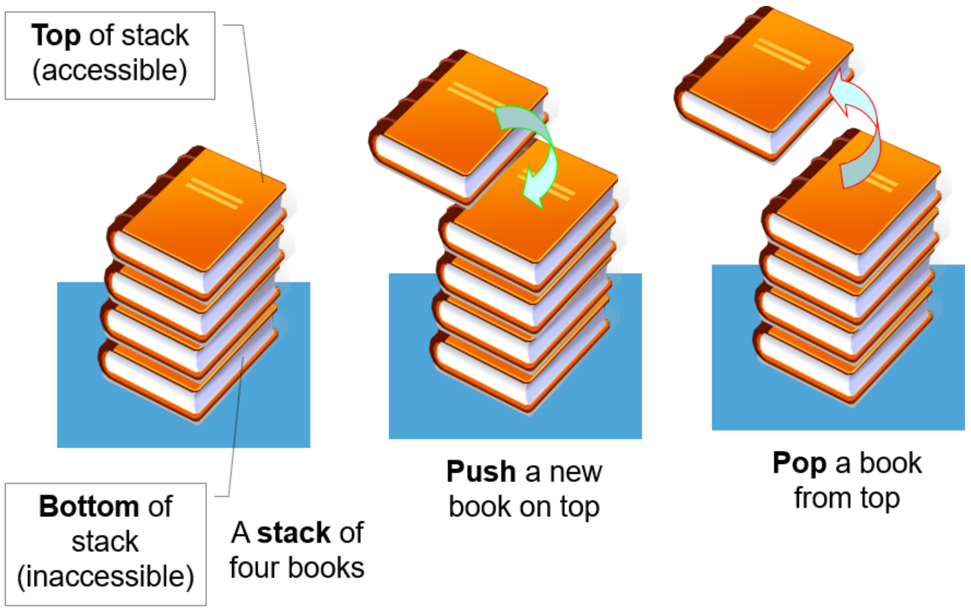
Problem 4

Solve the challenges on Hackerrank or write the solution in pseudocode:

- https://www.hackerrank.com/challenges/insert-a-node-at-the-head-of-a-linked-list/problem
- https://www.hackerrank.com/challenges/insert-a-node-at-the-tail-of-a-linked-list/problem
- https://www.hackerrank.com/challenges/delete-a-node-from-a-linked-list/problem
- https://www.hackerrank.com/challenges/reverse-a-linked-list/problem

3 Stacks

Stack is a particular kind of Abstract Data Type in which the main operations on the collection are the addition of an item to the collection, known as push, only to the top of the stack and removal of an item, known as pop, only from the top of the stack. It is known as Last-In-First-Out (LIFO) data structure, e.g. the stack of book below.

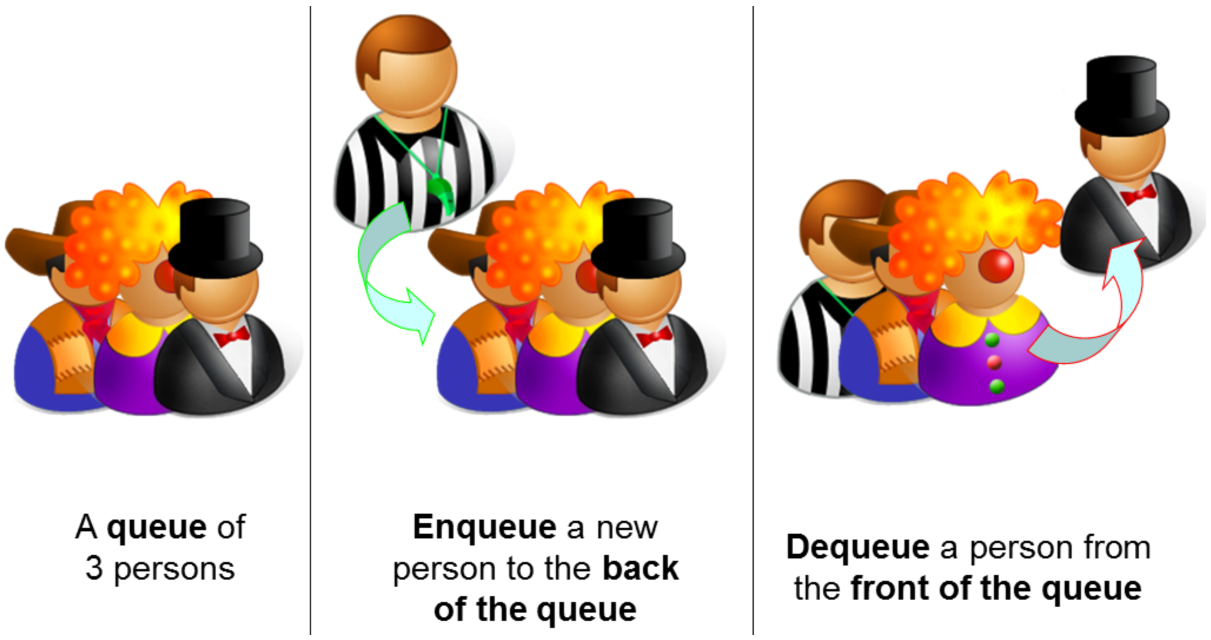

Top of stack (accessible)
Bottom of stack (inaccessible)
A **stack** of four books
Push a new book on top
Pop a book from top

Problem 5

Solve the challenge: https://www.hackerrank.com/challenges/balanced-brackets/problem on Hack-errank or in a pseudocode.

4 Queues

Queue is another Abstract Data Type in which the items in the collection are kept in order and the main operations on the collection are the addition of items to the back position (enqueue) and removal of items from the front position (dequeue). It is known as First-In-First-Out (FIFO) data structure as the first item to be enqueued will eventually be the first item to be dequeued, as in real life queues (see below).


A **queue** of 3 persons
Enqueue a new person to the **back of the queue**
Dequeue a person from the **front of the queue**

Problem 6

Solve the challenge: https://www.hackerrank.com/challenges/queue-using-two-stacks/problem on Hackerrank or in a pseudocode.

Problem 7 (1 point)

Show how to implement a first-in, first-out queue with a priority queue. Show how to implement a stack with a priority queue.