WORKSHEET WEEK 05: PRIORITY QUEUES

Priority queues are data structures that do not require full sorting, but provide the benefit of sorted lists – at every moment we can find the minimum (or maximum) of the items.

5.1 Concepts and Facts

Definition: A priority queue is an abstract data type supporting the following operations:

PRIORITYQUEUE() – create empty priority queue.
Q.INSERT($item$) – insert an item (with any key).
Q.EXTRACTMIN() – remove and return the element with the minimum key.

Priority queues can be implemented in various ways:

- As an ordered list (easy to find the minimum, but inserting new items may be expensive).
- As an unordered list (easy to add new items, but expensive to find the minimum).
- Create a *heap* (described below; insertion and finding the minimum are both fast).

Definition: A binary tree is called a *complete tree* if it has all layers filled in, except, perhaps, the last layer, which is filled from left to right (and may contain some empty slots on the right side of the last layer).

Complete trees were introduced just because they can be easily stored in arrays (all nodes are listed in the array – layer by layer).

Definition: A binary tree storing items with ordered keys is called a *minimum heap*, if it satisfies the following representation invariant:

- The binary tree is a complete binary tree.
- Each parent node in the tree has key that cannot be larger than either of its children.

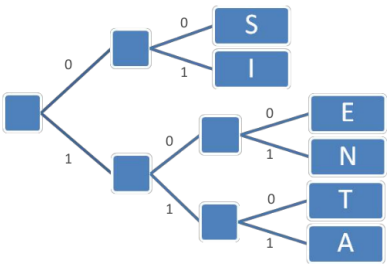
Similarly we define a *maximum heap* (no parent can be smaller than either of its children). Stacks, queues, deques and their implementations can preserve any *extrinsic order* of the items – typically the order they were inserted. In contrast, heap is a data structure that respects intrinsic ordering of item keys.

Definition: A *prefix encoding* for the given alphabet of messages $\mathcal{A} = \{m_1, m_2, \ldots, m_k\}$ is a function $e : \mathcal{A} \to \{0,1\}^*$ that maps each message to a codeword (a sequence of 0s and 1s) so that there are no two messages m_i, m_j such that $e(m_i)$ is a prefix of $e(m_j)$.

Any prefix encoding can be represented as a rooted tree with edges labeled with 0s and 1s – and every message is represented by a leaf in that tree.

Statement: Given any sequence of 0s and 1s and any prefix code for the alphabet of messages $\mathcal{A}$, the sequence of bits can be decoded into messages in no more than one way (i.e. there is no ambiguity).

Example: Consider the following Prefix Tree to encode letters in alphabet $\mathcal{A} = \{S, I, E, N, T, A\}$.



Every letter is encoded as a sequence of 0s and 1s (the path from the root to the respective letter).

- `11100110100` decodes as `111.00.110.100` or `A.S.T.E`.
- `0001100101111` decodes as `00.01.100.101.111` or `S.I.E.N.A`.

It may happen that some codeword has not been received completely (e.g. the codeword is expected to have three bits, but we only got two bits). These are the only scenarios when decoding cannot be completed.

Prefix trees that are optimal for encoding some alphabet of messages (with known message probabilities) uses priority queues. It is named Huffman algorithm.

Huffman Algorithm: Let C be the collection of letters to be encoded; each letter has its frequency $c.freq$ (frequencies are numbers describing the probability of each letter).

HUFFMAN(C):
  $n = |C|$
  $Q =$ PRIORITYQUEUE(C) (*Minimum heap by c.freq*)
  for $i = 1$ to $n - 1$ (*Repeat n-1 times*)
    $z =$ NODE()
    $z.left = x =$ EXTRACTMIN(Q)
    $z.right = y =$ EXTRACTMIN(Q)
    $z.freq = x.freq + y.freq$
    INSERT(Q, z)
  return EXTRACTMIN(Q) (*Return the root of the tree*)

Intuitively, if some message occurs with probability at least $1/2$, an optimal prefix tree (as obtained by Huffman algorithm) will encode it with one bit. If it occurs with probability at least $1/4$, an optimal prefix tree will spend up to two bits. If some message occurs with probability at least $1/8$, an optimal prefix tree will spend up to three bits. This can be formalized as follows:

Definition: For the message source (with messages $c \in \mathcal{A}$ define *Shannon entropy* with this formula:

$$H(\mathcal{A}) = \sum_{c \in \mathcal{A}} (-\log_2 P(c)) \cdot P(c),$$

where $P(c)$ denotes the probability of the character c in the alphabet.

Statement: No prefix tree can encode the alphabet $\mathcal{A}$ more efficiently than the Shannons entropy. Formally speaking, the expected number (the probabilistic weighted mean) of bits sent per one message $c \in \mathcal{A}$ will be at least $H(\mathcal{A})$.

5.2 Problems

Problem 1:

(A) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMIN(H) operation that removes the minimum element (and returns the heap into consistent state).

Find, if the heap property holds in the following array:

$$H[0] = 6, 17, 25, 20, 15, 26, 30, 22, 33, 31, 20.$$

If it is not satisfied, find, which two keys you could swap in this array so that the heap property is satisfied again. Write the correct sequence of array H.

Note: A *consistent state* in a minimum heap means that the key in parent does not exceed keys in left and right child.

(B) Assume that heap is implemented as a 0-based array (the root element is `H[0]`), and the heap supports DELETEMAX(H) operation that removes the maximum element.

If the heap does not satisfy invariant (in a consistent max-heap, every parent should always be at least as big as both children), then show how to swap two nodes to make it correct.

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 2 (Insert into a min-heap): Show what is the final state of a heap after you insert number 6 into the following minimum-heap (represented as a zero-based array):

$$9, 18, 28, 23, 20, 29, 33, 25, 36, 34, 23.$$

Problem 3 (Delete maximum from a Max-Heap): Show what is the final state of a heap after you remove the maximum from the following heap (represented as a zero-based array):

$$96, 67, 94, 10, 67, 68, 69, 9, 10, 11, 50, 67.$$

Problem 4 (Removing from Maximum Heap): Here is an array for a Max-Heap:

16	14	10	8	7	9	3	2	4	1

The image shows array used to store Maximum Heap (a data structure allowing inserts and removal of the maximum element). The array starts with the 0-th element (and any parent node in such tree should always be at least as big as any of its children).

(A) Draw the initial heap based on this array. Heap should be drawn as a complete binary tree.

(B) Run the command DELETEMAX(H) on this initial heap. Draw the resulting binary tree (after the heap invariant is restored – any parent node is at least as big as its children). Draw the binary tree image you get.

(C) On the tree that you got in the previous step (B) run the command INSERT(H, x), where $x = a + b + c$ is the sum of the last three digits of your student ID. Draw the binary tree image you get.

(D) Show the array for the binary tree you got in the previous step (C) (i.e. right after the DELETEMAX(H) and INSERT(H, x) commands have been executed).

Problem 6: Let us remind the *postfix notation* for arithmetic expressions. The following postfix expression:

```
2  17  1  -  *  3  4  *  +
```

represents the same expression as the infix expression $2 * (17 - 1) + 3 * 4$. Consider the following algorithm to evaluate postfix expressions:

```
POSTORDEREVALUATE(E : array[0..n − 1]): Int
    stack = emptyStack()
    for i from 1 to n:
        if ISNUMBER(E[i]):
            stack.PUSH(E[i])
        else:
            x1 = stack.POP()
            x2 = stack.POP()
            res = APPLYOP(E[i], x1, x2)
            stack.PUSH(res)
```

Assume that `stack` in this pseudocode is implemented as an array-based stack. Write the current state of `stack` right after the number 4 is inserted from the input tokens `2 17 1 - * 3 4 * +`.

Problem 7: Consider the task to identify correctly matched vs. incorrectly matched sequences of parentheses. There are three kinds of parentheses (round parentheses `()`, square brackets `[]` and curly braces `{}`); they are correctly matched iff opening and closing parentheses of the same kind (round, square or curly) can be put in pairs so that any two pairs either do not intersect at all or one pair is entirely inside another pair. Here are some examples:

```
correct: ( ) ( ( ) ) { ( [ ( ) ] ) }
correct: ( ( ( ) ( ( ) ) { ( [ ( ) ] ) }
incorrect: ) ( ( ) ) { ( [ ( ) ] ) }
incorrect: ( { [ ] ) }
incorrect: (
```

Input: An array X of n tokens, each of which is either a grouping symbol, a variable, an arithmetic operator, or a number.

Output: True if and only if all the grouping symbols in X match

Answer:

```
PARENMATCH(X[0..n − 1]):
    S = EMPTYSTACK()
    for i in range(n):
        if X[i] is an opening parenthesis:
            S.PUSH(X[i])
        else if X[i] is a closing parenthesis:
            if S.EMPTY():
                return FALSE (nothing to match with)
            if S.pop() does not match the type of X[i]:
                return FALSE (wrong type of parenthesis)
    if S.EMPTY():
        return TRUE (every symbol matched)
    else
```

return FALSE *(some symbols were never matched)*

Problem 8: Let the alphabet have 6 characters $\mathcal{A} = \{A, B, C, D, E, F\}$ and their probabilities are shown in the table:

c	A	B	C	D	E	F
$P(c)$	45%	13%	12%	16%	9%	5%

(A) Use the Huffman algorithm to create a Prefix Tree to encode these characters.

(B) Compute the Shannon entropy of this information source (sending the messages with the probabilities shown).

(C) Also compute the expected number of bits needed to encode one random letter by the Huffman code you created in (A). (Assume that letters arrive with the probabilities shown in the table.)