2 Stacks and Queues, Hashing, Trees

1. We use the following notation for operations:
   - *send* the first input character is removed from the input and sent directly to the output.
   - *push* the first input character is removed from the input and pushed on the stack.
   - *pop* the stack is popped and the obtained character is sent to the output.
   - *enqueue* the first input character is removed from the input and placed in the queue.
   - *dequeue* a character is removed from the queue and sent to the output.

(a) The permutation X U W V Y can be produced by the following operations:

   send, push, send, push, push, pop, pop, pop

It cannot be produced by using queue. After sending X directly to the output we have to read Y. To obtain the required sequence it cannot be sent to the output. We can only enqueue it. Then U has to be sent to the output, since it has to precede Y in the required sentence. The next character is V. We cannot send it to the output, but if we enqueue it we still cannot obtain the required sequence since on the output it will be preceded by already enqueued Y.

(b) The permutation Y U W X V cannot be obtained by using stack. The longest possible sequence of operations not violating the requirement is push, send, send. The next input character is V. We cannot send it to output. If we push it on the stack, then in the resulting sequence it will precede X which is presently on the stack.

This permutation can be obtained by using queue:

enqueue, send, send, enqueue, send, dequeue, dequeue

2. (a) Denote the stacks S1 and S2. The items pushed on S1 are stored in the cells T[0], T[1], ..., T[k1] where k1 is the index of the top when S1 contains k1 + 1 items. Similarly, the items pushed on S2 are stored in T[n], T[n-1], ..., T[n-k2] where n - k2 is the index of the top when S2 contains k2 + 1 items. Overflow error is reported by an attempted push operation if k1 + k2 = n + 1.

(b)  

a**
---

\text{Front} = 0, \text{Length} = 1
abc
---

\text{Front} = 0, \text{Length} = 2
abc
---

\text{Front} = 0, \text{Length} = 3
abc
---

\text{Front} = 1, \text{Length} = 2
abc
---

\text{Front} = 2, \text{Length} = 1
dbc
---

\text{Front} = 2, \text{Length} = 2.
3. In the following we assume that input reads and returns one character from the input.

\[
\text{function palindrome}(n); \quad \text{boolean}
\]

\[
\text{makeEmptyStack}(S);\]

\[
\text{for } i = 1 \text{ to } \lfloor n/2 \rfloor \text{ do}
\]

\[
\text{Push}(S, \text{input})
\]

\[
\text{if } 2 \times \lfloor n/2 \rfloor \neq n \text{ then } \text{input}
\]

\[
\text{for } i = 1 \text{ to } \lfloor n/2 \rfloor \text{ do}
\]

\[
\text{if } \text{Pop}(S) \neq \text{input} \text{ then return } \text{false}
\]

\[
\text{return } \text{true}
\]

The worst case running time is \(O(n)\) (justification omitted).

4. (a) \quad \bullet \text{ MakeEmptyQueue: create two empty stacks: } E \text{ and } D.

\bullet \text{ To enqueue data item } d \text{ push it on } E.

\bullet \text{ To dequeue:}

\hspace{1em} \text{If } D \text{ is empty and } E \text{ is empty report error.}

\hspace{1em} \text{Otherwise:}

\hspace{2em} \text{i. If } D \text{ is empty, pop one-by-one all elements of } E \text{ and push them on } D. \text{ After this all enqueued data items are placed on } D \text{ from top to bottom in order of enqueuing: the most recently enqueued one is in the bottom and the Front element is placed on the top of } D. \text{ We pop it to complete the dequeue operation.}

\hspace{2em} \text{ii. Otherwise pop from } D.

\bullet \text{ Front operation is as dequeue but the data item is not popped from } D.

\bullet \text{ IsEmptyQueue is implemented as two checks: IsEmptyStack}(E) \text{ and IsEmptyStack}(D); \text{ it returns true iff both checks return true.}

(b) Pseudocode not included

(c) We assume that each stack operation runs in \(\Theta(1)\).

\hspace{1em} \text{i. Enqueue operation is always a push on } E, \text{ so it runs in constant time } \Theta(1).

\hspace{1em} \bullet \text{ A single enqueue operation is implemented as push on } E, \text{ thus works always in } \Theta(1); \text{ there is no specific worst case.}

\hspace{1em} \bullet \text{ The worst case of dequeue is if it is requested after pushing } n \text{ data items on } E. \text{ In that case we need:}

\hspace{2em} (1) emptiness check of both stacks done in } \Theta(1)

\hspace{2em} (2) \text{ } n \text{ pop operations and } n \text{ emptiness checks on } E \text{ done in } \Theta(n)

\hspace{2em} (3) \text{ } n \text{ push operations on } D \text{ done in } \Theta(n)

\hspace{2em} (4) \text{ one pop operation on } D \text{ done in } \Theta(1)

\text{Thus the worst case dequeue takes } \Theta(n) \text{ time.}
(d) Each data item of the sequence is only once pushed on \( E \), popped from \( E \), pushed on \( D \) and popped on \( D \). Thus the amortized cost to enqueue and to dequeue all \( n \) data items is \( \Theta(n) \), and the amortized cost per item is \( \Theta(1) \).

5. The solution is a simple modification of the ring buffer (see p.76 of the textbook). Thus the elements placed in deque \( D \) are stored in a table \( T[0..N-1] \). Two variables: \( F \) and \( L \) are used to store, respectively, the index of the front and the length of the actually stored sequence. Empty deque is created by setting \( F \) and \( L \) to 0 and allocating \( T \). If \( D \) is nonempty its elements \( d_0, \ldots, d_{L-1} \), are stored in the positions \( T(F), T(F+1) \mod N, \ldots, T(F+L) \mod N \). The operations are executed as follows:

- \( addFront(E,D) \): If \( L = N \) report error. Otherwise increase \( L \) by one and place \( E \) in \( T(F+L-1)\mod N \).
- \( addFront(E,D) \): If \( L = N \) report error. Otherwise increase \( L \) by one and place \( E \) in \( T(N-1) \) if \( F = 0 \) and in \( T(F-1) \) if \( F \neq 0 \). Update \( F \) respectively.
- \( deleteFront(D) \): If \( L = 0 \) report error. Otherwise return \( T(F) \), and decrease \( L \) by one.
- \( deleteFront(D) \): If \( L = 0 \) report error. Otherwise return \( T(F-L+1) \mod N \) and decrease \( F \) by one and increase \( L \) modulo \( N \).

6. (a) Coalesced chaining; the pointers are represented as table indices.

<table>
<thead>
<tr>
<th></th>
<th>0 x</th>
<th>8a 0 x</th>
<th>8a 0 2</th>
<th>8a 0 2</th>
<th>8a 0 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15c</td>
<td>0 x</td>
<td>15c 0 0</td>
<td>15c 0 0</td>
<td>15c 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 x</td>
<td>0 x</td>
<td>14b 0 x</td>
<td>14b 0 x</td>
<td>14b 0 x</td>
</tr>
<tr>
<td>3</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
</tr>
<tr>
<td>4</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
</tr>
<tr>
<td>5</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
</tr>
<tr>
<td>6</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
<td>0 x</td>
</tr>
</tbody>
</table>

(b) Solution not included.

(c) Double hashing.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
<th>14b</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
<td>0</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
<td>15c</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>8a</td>
<td>0</td>
<td>8a</td>
<td>0</td>
<td>8a</td>
<td>0</td>
<td>8a</td>
<td>0</td>
<td>8a</td>
<td>0</td>
<td>8a</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>32d</td>
<td>0</td>
<td>32d</td>
<td>0</td>
<td>32d</td>
<td>0</td>
<td>32d</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4e</td>
<td>0</td>
<td>4e</td>
<td>0</td>
<td>4e</td>
<td>0</td>
<td>4e</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>7f</td>
<td>0</td>
<td>7f</td>
<td>0</td>
<td>7f</td>
<td>0</td>
<td>7f</td>
<td>0</td>
</tr>
</tbody>
</table>

7. With the \textit{deleted bit marking} technique the space of the deleted elements is not
re-used for insertion. The probing sequences become very long even if they include mostly the deleted elements, this increases the time of the search.

8. Two letters give less conflicts over the space of all car identifications and are thus better than the remaining variants, but require larger hash table.

9. The level order.

10. (a) preorder
    (b) procedure TraverseTree(n: treenode);
        var Q: ADT Queue
        MakeEmptyQueue(Q)
        Enqueue(n)
        while not IsEmptyQueue(Q) do
            n ← Dequeue(Q)
            Print label of n
            foreach child c of n in reverse order do
                Enqueue(c)

        This is "reversed" level order traversal.

11. (a) full trees: a, c, f, g
    (b) complete trees: a, b, c, g
    (c) perfect trees: a, c

    The trees d and h do not belong to any of these categories.

12. The indexing of the array starts with 1, according to the variant in the course book p.296 and p.338, or with 0, according to the variant discussed at the lecture.

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

| a | b | i | c | f | j | k | d | e | g | h |