WORKSHEET WEEK 03: ADTS AND IMPLEMENTATIONS

3.1 Introduction

Some stuff from Python:

```
L = ["Cacophony", "Imbibe", "Writhe", "Ubiquitous", "Paradox"]
S = {"Cacophony", "Imbibe", "Writhe", "Ubiquitous", "Paradox"}
for item in sorted(S):
    print(i)
```

Questions

1. Is a list in Python ordered? (I.e. can we visit its elements in a predictable order?) Is a set in Python ordered?
2. Assume that you want to store a large alphabetically sorted dictionary in Python to enable binary search (it has $O(\log n)$ runtime). Is it a set of words or a list of words?
3. Occasionally we need to insert new words in that Python data structure. Assume that some item is inserted in a wrong order. What will be the consequences for subsequent operations?

Introducing concepts:

- *Extrinsic order:* (imposed by they way data structure is built) vs. *intrinsic order* (wholly depends on the properties of the items themselves). List by definition is ordered (as a sequence of items $a_0, a_1, \ldots, a_{n-1}$); it can also be sorted.
- Mathematical sets do not have any order by themselves (but numbers in a set may be ordered using *order relations* $a \leq b$ or $a < b$). In many programming languages sets may be ordered or unordered (e.g. Java interface java.util.SortedSet), but there are no *extrinsic* enumerations to access its elements.
- *Representation invariant:* Physical representation of a data structure may use some assumptions to function normally. All operations can assume that the assumptions are true (and should not break these assumptions themselves).
- *Mutable data structure:* Immutable objects are easier to pass as arguments to functions (passing a reference is always fine). They make functions more *mathematical* and easier to debug.
- *Sorting in place* vs. *returning a sorted copy*.

```
elevation = dict()
elevation[(0,0)] = 3
elevation[(0,1)] = 4

other_dict = { [0,0]: 3, [0,1]: 4}
# What is "unhashable type"?
```

(continues on next page)

(continued from previous page)

```
a = [0, 1]
a.insert(1,a)
a[1][1][0]
a[1][1][1]
```

In Java you can easily create a set S of sets $s_0, s_1, \ldots, s_{k-1}$ (and then modify one of the s_i). The larger set S behaves in a funny way – you can still check that the size of S equals to k, but you cannot find the modified (or the original) set s_i anymore. If you check, if $s_i \in S$, Java returns false.

3.2 Lists – Redundant, but Convenient ADTs

 # (Constructor: Create an empty list)
LIST $< T > L()$
 # (Initializer list: Create a list with some elements in it)
LIST $< T > L(\{item1 : T, item2 : T, \ldots\})$
 # (Insertion: Inserts an element at a specified position in the list)
L.INSERT(index : int, item : T)
 # (Deletion: Deletes the element at a specified position in the list.)
L.DELETE(index : int)
 # (Access the item at a specified position)
L.GET($index : int$) : T
 # (Traversal: Accesses each element in the list List<T> in its order)
for item **in** L { do something with item }
 # (Search: Finds the position of the first entry of item (>= initialPosition))
L.FIND(item : T) : int
L.FIND(item : $T, startFrom : int$) : int
 # (Concatenation: Combines two lists into a single list)
L.CONCATENATE($L1$: LIST $< T >, L2$: LIST $< T >$) : LIST $< T >$
 # (Sorting: Rearranges the list in sorted order)
L.SORT() : $void$
 # (Getting a sorted copy: Return a sorted copy of the list, but do not change the list)
L.SORTEDCOPY() : LIST $< T >$
 # (Slicing: Extract a fragment of the list from startPos (inclusive) to endPos (exclusive))
L.SLICE($startPos : int, endPos : int$)
 # (Size: Returns the number of elements in the list.)
L.SIZE() : int

3.2.1 Mutable vs. Immutable Lists

Mutable lists support insertion, deletion, appending elements, sorting (in place). Both mutable and immutable lists support all the other operations. From functional programming there are two more operations:

Filtering: Creating a new list that contains only the elements that satisfy a certain condition

Mapping: Creating a new list that contains the results of applying a function to each element of the original list.

3.2.2 Sequence ADT

Container operations: (initializing sequence)

given an iterable X, build sequence from items in iterator X
$S = \text{BUILD}(X)$
return the number of stored items
$S.\text{SIZE}()$

Static operations: (operations not affecting sequence size)

return the stored items one-by-one in sequence order
$S.\text{ITERSEQ}()$
return the ith item
$S.\text{GET}(i)$
replace the ith item with x
$S.\text{SET}(i, x)$

Dynamic operations: (operations affecting sequence size)

add x as the ith item
$S.\text{INSERTAT}(i, x)$
remove and return the ith item
$S.\text{DELETEAT}(i)$
add x as the first item
$S.\text{INSERTFIRST}(x)$
remove and return the first item
$S.\text{DELETEFIRST}()$
add x as the last item
$S.\text{INSERTLAST}(x)$
remove and return the last item
$S.\text{DELETELAST}()$

3.2.3 Stack ADT

$S = \text{EMPTYSTACK}()$ – create an empty stack
$S.\text{PUSH}(item)$ – add one element to the top of the stack
$S.\text{POP}()$ – remove and return one element from the top of the stack
$S.\text{TOP}()$ – return, but do not remove the top element.
$S.\text{ISEMPTY}()$ – return true, iff the stack is empty.

Optionally stacks can support some other functions:

$S.\text{CLEAR}()$ – Remove everything from the stack
$S.\text{SIZE}()$ – Return the number of elements in the stack

3.2.4 Iterator ADT

In an iterator with n items, the cursor typically has $n + 1$ valid states (it can be right before any of the elements, or it can be at the very end). Iterators are convenient to write iterative **for** loops or otherwise process items one by one (as in bulk insert operations, reading items from an input buffer, etc.).

Traversal: Accesses the next element in the iterator and moves 1 step ahead
$it.\text{NEXT}()$
Checking for more elements: Checks if there are more elements without moving 1 step ahead
$it.\text{HASNEXT}()$
Removing the current element: Removes the current element from the data structure, point to the next one
$it.\text{REMOVE}()$
Peeking the current element: Returns the current element without moving the iterator 1 step ahead
$it.\text{PEEK}()$
Rewinding: Resets the iterator to the beginning of the data structure (right before the 1st element)
$it.\text{REWIND}()$
Skipping: Skips a specified number of elements in the data structure.
$it.\text{SKIP}(n : int)$
Filtering: Filters elements in the data structure based on a given predicate.
$it.\text{FILTER}(predicate : item : T => isValid : Boolean)$
Mapping: (Lazy) apply of a given function to each element, returns another iterator.
$it.\text{MAP}(\text{MAPFUNCTION} : item : T => value : U)$

3.2.5 Case Study: Matching Parentheses

```
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

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)*

3.2.6 Case Study: Evaluation of Postfix Notation

```
# infix notation
2 * (17 - 1) + 3 * 4
# postfix notation
2  17  1  -  *  3  4  *  +
```

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)

Question: Given the pseudocode for *text{sc PostorderEvaluate}(E)*, write the current state of the stack right after the

$E[6]$, i.e. the number 4 is insered.

3.2.7 Case Study: Backtracker Object as ADT

Here is an object-oriented way to solve Sudoku, N-Queens problem, and many more combinatorial tasks.

In this example a backtracker object is a sort of iterator designed to visited all nodes of a rooted tree in the DFS order and display the first valid solution (or all valid solutions). Backtracker is usually inefficient (exponential time algorithm), but it can become more efficient, if it can establish early on that in the given subtree there are no more solutions (for example, one of the rules has been violated).

(Initialize Backtracker with its initial state)
$B =$ BACKTRACKER(s : *State*)
(Get moves available from the current state at the given tree level)
B.MOVES(*level* : int) : ITERATOR $<$ Move $>$
(Is the move valid for the given backtracker state)
B.VALID(*level* : int, *move* : $Move$) : BOOLEAN
(Record the move to the backtracker – move down in the search tree)
B.RECORD(*level* : int, *move* : $Move$)
(Opposite to record – undo the move to move up in the search tree)
B.UNDO(*level* : int, *move* : $Move$)
(Is the search successfully completed?)
B.DONE(*level* : int) : BOOLEAN
(Output the successful state of the Backtracker object)
B.OUTPUT()

Write the pseudocode for a function ATTEMPT(*level* : int) so as to find the first solution (or all solutions) starting with the backtracker object on level *level*.

3.3 Problems

Problem 1: In some Python's implementations, the dynamic array is grown by $n/8$ whenever the list overflows. Assume that r is the ratio between inserts and reads for this dynamic array. Find the value r for which this growth factor is the optimal one.

Problem 2: Reverse the order of elements in stack S in the following ways:

(A) Use two additional stacks, but no auxiliary variables.

(B) Use one additional stack and some additional non-array variables (i.e. cannot use lists, sequences, stacks or queues).

Problem 3: Read elements from an iterator and place them on a stack S in ascending order using one additional stack and some additional non-array variables.

Problem 4: Given a data structure implementing the Sequence ADT, show how to use it to implement the Set interface. (Your implementation does not need to be efficient.)

Problem 5: What are the costs for each ADT operation, if a queue is implemented as dynamic array?

Problem 6: Which operations become asymptotically faster, if list ADT is implemented as a doubly linked list (instead of a singly linked list)?

Problem 7: Write a pseudocode to implement the following two operations on the Backtracker object:

(A) FINDFIRSTSOLUTION(b : BACKTRACKER) – a function that returns the first solution (in fact – any 1 solution) for the given backtracker object.

(B) FINDALLSOLUTIONS(b : BACKTRACKER)

Problem 8: Koch snowflake consists of three sides. Each side connects two vertices of an equilateral triangle ABC. Consider, for example, two points A and B and the edge connecting them e.

If the length of e is 1 unit or shorter, then AB is connected by a straight line segment. Otherwise, the segment AB is subdivided into three equal parts: e_1, e_2, e_3. The middle part is complemented with two more line segments f_1 and g_1 to make another equilateral triangle (with side length three times smaller than the ABC). Finally, the Koch snoflake's edge algorithm is called on each of the segments e_1, f_2, g_2, e_3 recursively.

The initial call for $e = AB$ is SNOWFLAKEEDGE$(e, 0)$, where $d = 0$ is the initial depth in the recursion tree. Here is the pseudocode for the algorithm:

```
SNOWFLAKEEDGE(e, d):
    if |e| ≤ 1:
        draw a straight edge e
    else:
        Split e into three equal parts e_1, e_2, e_3
        Construct a regular triangle out of edges e_2, f_2, g_2 to the "outside"
        SNOWFLAKEEDGE(e_1, d + 1)
        SNOWFLAKEEDGE(f_2, d + 1)
        SNOWFLAKEEDGE(g_2, d + 1)
        SNOWFLAKEEDGE(e_3, d + 1)
```

In this algorithm we assume that the Koch snowflake is drawn as vector graphics on a device with infinite resolution.

(A) How many levels does the depth parameter d reach, if the initial size of the edge is $|e| = n$.

(B) Estimate the number of recursive calls of SNOWFLAKEEDGE(e, d), if the initial size of the edge is $|e| = n$.

(C) Write a recursive time complexity of this algorithm $T(n)$ and estimate it with Master's theorem.

3.4 Bibliography

1. Robert E. Noonan. An Object-Oriented View on Backtracking