Recap: basic data structures

Data Structures and Algorithms for Computational Linguistics III
ISCL-BA-07

Çağrı Çöltekin
ccoltekin@sfs.uni-tuebingen.de

University of Tübingen
Seminar für Sprachwissenschaft

Winter Semester 2020/21

Overview

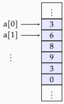
- Some basic data structures
  - Arrays
  - Lists
  - Stacks
  - Queues
- Revisiting searching a sequence

Abstract data types and data structures

- An *abstract data type* (ADT), or abstract data structure, is an object with well-defined operations. For example a *stack* supports `push()` and `pop()` operations
- An abstract data structure can be implemented using different *data structures*. For example a stack can be implemented using a linked list, or an array
- Sometimes names, usage is confusingly similar

Arrays

- An array is simply a contiguous sequence of objects with the same size
- Arrays are very close to how computers store data in their memory
- Arrays can also be multi-dimensional. For example, matrices can be represented with 2-dimensional arrays
- Arrays support fast access to their elements through indexing
- On the downside, resizing and inserting values in arbitrary locations are expensive



`a = [3, 6, 8, 9, 3, 0]`

Arrays
in Python

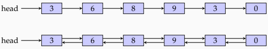
- No built-in array data structure in Python
- Lists are indexable
- For proper/faster arrays, use the `numpy` library

List indexing in Python

```
a = [3, 6, 8, 9, 3, 0]
a[0]   # 3
a[-1]  # 0
a[1:4] # [6, 8, 9]
a2d = [[3, 6, 8], [9, 3, 0]]
a2d[0,1] # 6
```

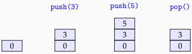

Lists

- Main operations for list ADT are
  - append
  - prepend
  - head (and tail)
- Lists are typically implemented using linked lists (but array-based lists are also common)
- Python lists are array-based



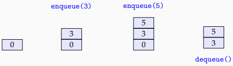
Stacks

- A stack is a last-in-first (LIFO) out data structure
- Two basic operations:
  - push
  - pop
- Stacks can be implemented using linked lists (or arrays)



Queues

- A queue is a first-in-first (FIFO) out data structure
- Two basic operations:
  - enqueue
  - dequeue
- Queues can be implemented using linked lists (or maybe arrays)



Other common ADT

- *Strings* are often implemented based on character arrays
- *Maps* or *dictionaries* are similar to arrays and lists, but allow indexing with (almost) arbitrary data types
  - Maps are generally implemented using hashing (later in this course)
- Sets implement the mathematical (finite) sets: a collection unique elements without order
- Trees are used in many algorithms we discuss later (we will revisit trees as data structures)

Studying algorithms

- In this course we will study a series of important algorithms, including
  - Sorting
  - Pattern matching
  - Graph traversal
- For any algorithm we design/use, there are a number of desirable properties

Correctness an algorithm should do what it is supposed to do
Robustness an algorithms should (correctly) handle all possible inputs it may receive
Efficiency an algorithm should be light on resource usage
Simplicity an algorithm should be as simple as possible
– ...

- We will briefly touch upon a few of these issues with a simple case study

A simple problem: searching a sequence for a value

```
def linear_search(seq, val):
    answer = None
    for i in range(len(seq)):
        if seq[i] == val:
            answer = i
    return answer
```

Is this a good algorithm? Can we improve it?

Linear search: take 2

```
def linear_search(seq, val):
    for i in range(len(seq)):
        if seq[i] == val:
            return i
    return None
```

Can we do even better?

Linear search: take 3

```python
def linear_search(seq, val):
    n = len(seq) - 1
    last = seq[n]
    seq[n] = val
    i = 0
    while seq[i] != val:
        i += 1
    seq[n] = last
    if i < n  or seq[n] == val:
      return i
    else:
      return None
```

- Is this better?
- Any disadvantages?
- Can we do even better?

Binary search

```python
def binary_search(seq, val):
    left, right = 0, len(seq)
    while left <= right:
        mid = (left + right) // 2
        if seq[mid] == val:
            return mid
        if seq[mid] > val:
            right = mid - 1
        else:
            left = mid + 1
    return None
```

- We can do (much) better if the sequence is sorted.

Binary search

recursive version

```python
def binary_search_recursive(seq, val, left=None, right=None):
    if left is None:
        left = 0
    if right is None:
        right = len(seq)
    if left > right:
      return None
    mid = (left + right) // 2
    if seq[mid] == val:
        return mid
    if seq[mid] > val:
      return binary_search_recursive(seq, val, left, mid - 1)
    else:
      return binary_search_recursive(seq, val, mid + 1, right)
```


A note on recursion

- Some problems are much easier to solve recursively.
- Recursion is also a mathematical concept, properties of recursive algorithms are often easier to prove
- Reminder:
  - You have to define one or more *base cases* (e.g., `if left > right` for binary search)
  - Each recursive step should approach the base case (e.g., should run on a smaller portion of the data)
- We will see quite a few recursive algorithms, it is time for getting used to if you are not

Exercise: write a recursive function for linear search.

Summary

- This lecture was a slow review of some basic data structure and algorithms.
- We will assume you know these concept, revise your earlier knowledge if needed

Next:

- A few common patterns of algorithms
- Analysis of algorithms

An interesting, but not-so-relevant anecdote

How hard can binary search could be?

- It was first suggested in lecture in 1946 (by John Mauchly)
- First fix to this version was suggested in 1960 (by Derrick Henry Lehmer)
- Another, fix/improvement over this was published on 1962 (by Hermann Bottenbruch)
- In 2006, a bug in Java's binary search implementation was discovered

Acknowledgments, credits, references

- Some of the slides are based on the previous year's course by Corina Dima.