Lists, stacks, queues, iterators

Advanced programming

Artur Kulmizev

2021-02-03

(slides used with permission from Sara Stymne)

Today

- Lists
  - Array-based
  - Linked
- Stacks
- Queues
- Iterators

Abstract data types

- Abstract data type (ADT)
- A theoretical concept (mathematical model)
- Defined as:
  - A type
  - A list of operations that can be performed
- The ADT does not say anything about how to implement it, only what you should be able to do with it
- Sometimes complexity statements are also included in the ADT
- Contrasts with a data structure, which also defines the implementation

Abstract data types – list

- List is an ADT
- Corresponds to the mathematical notion of sequence
- A list is a **finite, ordered seqence** of data objects
- An object can occur more than once in the list

Abstract data types – list

- List is an ADT
- Corresponds to the mathematical notion of sequence
- A list is a **finite, ordered seqence** of data objects
- An object can occur more than once in the list
- ordered in the way that each object has a position, not necessarily sorted

List methods

- A list should typically have the following methods (at least):
  - isEmpty()
  - size()
  - addFirst(E element)
  - addLast(E element)
  - get(int index)
  - remove(int index)
  - . . .

List – implementation

- How to implement a list is not defined
- Two standard ways to implement lists
  - Array-based
  - As a linked structure

Array-based lists

- In an array, elements are stored in a contiguous span of memory
- An array has "random access", which mean that we can access an element anywhere in the array in constant time
- An array can typically only contain elements of a single pre-defined type
- Static arrays have a fixed size

Array-based lists in Python

- The standard Python list
- Uses an underlying array
- In Python there is no static array
- Python lists are "dynamic arrays"
  - The size is automatically increased when the underlying array becomes full
  - The implementation keeps track of how many items are in the list
- Python lists can have a mix of types in them
  - Because the list contains "references", which are of equal size

Linked structures

- Linked list
- Each element knows its value, and what the next element is
- The list class only needs to know the first element

Linked list – implementation

```
//Node class:
class Node:
    def __init__(self, data, next):
        self.next = next
        self.data = data

//List class
class LinkedList:
    def __init__(self):
        self.head = None
```

Linked list – implementation with dummy node

```
//Node class:
class Node:
    def __init__(self, data, next):
        self.next = next
        self.data = data

//List class
class LinkedList:
    def __init__(self):
        self.head = Node("**dummy**", None)
```

The classes

- The node class is used to represent each element in the list (each node)
  - next points at the next node, or is empty (None) if the element is the last in the list
- The list class knows only the first list element
  - None or dummy node if the list is empty,
    points to the first node otherwise
- The list may also hold a pointer to the last element

Linked list – operations

- Get a value in a given position
- Insert a new value
- Remove a value
- Print the values in the list

Example – insert a new value first in the list

```
class LinkedList:
    def __init__(self):
        self.head = None

    def insertFirst(self, value):
        self.head = Node(value, self.head)
```

Example – get last element of the list

```
def getLast(self):
    if not self.head:
        return None
    n = self.head
    p = None
    while n:
        p = n
        n = n.next
    return p.data
```

Example – get list size

```
def size(self):
    n = self.head
    count = 0
    while n:
        count = count + 1
        n = n.next

    return count
```

Example – remove an item

```
def remove(self,item):
    n = self.head
    p = None
    found = False
    while n and not found:
        if n.data == item:
            found = True
        else:
            p = n
            n = n.next
    if found:
        if p:
            p.next = n.next
        else:
            self.head = n.next
```

Example – remove an item

```
def remove(self,item):
    n = self.head
    p = None
    found = False
    while n and not found:
        if n.data == item:
            found = True
        else:
            p = n
            n = n.next
    if found:
        if p:
            p.next = n.next
        else:
            self.head = n.next
```

This can be made simpler if empty list has a dummy node

Types of linked lists

- Singly-linked list
  - Each node knows the next node
  - The list class knows the first node
- Doubly-linked list
  - Each node knows both the next and the previous node
  - The class knows the first and the last node

Types of linked lists

- Singly-linked list
  - Each node knows the next node
  - The list class knows the first node
  - The list class might also know the last element
- Doubly-linked list
  - Each node knows both the next and the previous node
  - The class knows the first and the last node

Types of linked lists

- Singly-linked list
  - Each node knows the next node
  - The list class knows the first node
  - The list class might also know the last element
- Doubly-linked list
  - Each node knows both the next and the previous node
  - The class knows the first and the last node
  - Makes certain operations more efficient

Doubly-linked list – time complexity

- Get a value
  - First/last: $O(1)$
  - At position x: $O(n)$

Doubly-linked list – time complexity

- Get a value
  - First/last: $O(1)$
  - At position x: $O(n)$
- Insertion/deletion of a value
  - First/last: $O(1)$

Doubly-linked list – time complexity

- Get a value
  - First/last: $O(1)$
  - At position x: $O(n)$
- Insertion/deletion of a value
  - First/last: $O(1)$
  - At position x:
    - Find the correct position: $O(n)$
    - The insertion/deletion itself: $O(1)$

Linked lists in Python

- collections.deque
  - Deque = double-ended queue
  - Implemented as a doubly-linked list
  - Slightly more complex than standard implementation
  - Based on linked list of “blocks”, access is still O(n), but in practice it does not have to iterate through all elements and is faster than the standard implementation.

Complexity for different list implementations

	Array	Linked list
Iteration	$O(n)$	$O(n)$
Get value at position x	$O(1)$	$O(n)$
Insert value first	$O(n)$	$O(1)$
Insert value at position x	$O(n)$	$O(n)$
Insert value last	$O(1)$	$O(1)$
Remove value first	$O(n)$	$O(1)$
Remove value at positon x	$O(n)$	$O(n)$

Complexity for different list implementations

	Array	Linked list
Iteration	$O(n)$	$O(n)$
Get value at position x	$O(1)$	$O(n)$
Insert value first	$O(n)$	$O(1)$
Insert value at position x	$O(n)$	$O(n)$
Insert value last	$O(1)$	$O(1)$
Remove value first	$O(n)$	$O(1)$
Remove value at positon x	$O(n)$	$O(n)$

- Above is for static arrays
- A dynamic array that needs to be resized costs $O(n)$ for insertion, but still amortized $O(1)$
- This means that insertion last is fast on average, but individual operations can be slow

Other linked structures

- Linked structures can be used for other purposes than linked lists
- For instance binary trees
- Instead of pointing to the next node, each node points to the left and right children

Linked binary tree – implementation

```
//Node class:
class Node:
    def __init__(self, data, right, left):
        self.right = right
        self.left = left
        self.data = data

//List class
class LinkedList:
    def __init__(self):
        self.root = None
```

Binary search trees

- Mapping between keys and values (like a hash table)
- Special case of binary trees, where the keys are arranged in a sorted way
- If the tree is balanced, it has a good time complexity for search/insertion: $O(\log n)$

Binary search trees

- Mapping between keys and values (like a hash table)
- Special case of binary trees, where the keys are arranged in a sorted way
- If the tree is balanced, it has a good time complexity for search/insertion: $O(\log n)$
- Unbalanced trees have the same complexity as a linked list (O(n))

Binary search trees

- Mapping between keys and values (like a hash table)
- Special case of binary trees, where the keys are arranged in a sorted way
- If the tree is balanced, it has a good time complexity for search/insertion: $O(\log n)$
- Unbalanced trees have the same complexity as a linked list (O(n))
- There are many methods for balancing trees
  - AVL-trees
  - Red-black trees
  - ...

Binary search trees

- Mapping between keys and values (like a hash table)
- Special case of binary trees, where the keys are arranged in a sorted way
- If the tree is balanced, it has a good time complexity for search/insertion: $O(\log n)$
- Unbalanced trees have the same complexity as a linked list (O(n))
- There are many methods for balancing trees
  - AVL-trees
  - Red-black trees
  - ...
  - Details are outside the scope of this course

Complexity for different list implementations

	Array	Linked list
Iteration		
Get value at position x		
Insert value first		
Insert value at position x		
Insert value last		
Remove value first		
Remove value at positon x		

Abstract data type – stack

- A list like structure, but where elements can only be inserted and removed in one end
- "LIFO" – last in, first out
- Elements are removed in the reverserd order of insertion
- As a pile of books

Abstract data type – stack

- A list like structure, but where elements can only be inserted and removed in one end
- "LIFO" – last in, first out
- Elements are removed in the reverserd order of insertion
- As a pile of books
- Example use cases:
  - Reverse a word
  - Undo
  - Depth-first search

Stack – basic methods

- push(E element)
- pop()
- peek()

Stack – basic methods

- push(E element)
- pop()
- peek()
- isEmpty()
- size()
- clear()

Stack – implementations

- How a stack is implemented is not specified
- Two common implementations
  - Array-based
  - Linked structure

Complexity for stack implementations

	Array	Linked list
pop	$O(1)$	$O(1)$
push	$O(1)$	$O(1)$
peek	$O(1)$	$O(1)$

Complexity for stack implementations

	Array	Linked list
pop	$O(1)$	$O(1)$
push	$O(1)$	$O(1)$
peek	$O(1)$	$O(1)$

- Since stack is so specialized, it can be implemented efficiently

Abstract datatype – queue

- Queue
- A list like structure, but where elements can only be inserted in one end and removed in the other end
- "FIFO" – first in, first out
- Elements are removed in the same order as they are inserted
- As a queue in a store

Abstract datatype – queue

- Queue
- A list like structure, but where elements can only be inserted in one end and removed in the other end
- "FIFO" – first in, first out
- Elements are removed in the same order as they are inserted
- As a queue in a store
- Example use cases:
  - Print queue
  - Breadth-first search

Queue – basic methods

- enqueue(element)
- dequeue()

Queue – basic methods

- enqueue(element)
- dequeue()
- isEmpty()
- size()
- clear()

Queue – implementation

- How a queue is implemented is not specified
- Two common implementations
  - Array-based
  - Linked structure

Complexity for queue implementations

	Array Naive	Array Complex	Linked list
enqueue	$O(1)$	$O(1)$	$O(1)$
dequeue	$O(n)$	$O(1)$	$O(1)$

Complexity for queue implementations

	Array Naive	Array Complex	Linked list
enqueue	$O(1)$	$O(1)$	$O(1)$
dequeue	$O(n)$	$O(1)$	$O(1)$

- Since queue is so specialized, it can be implemented efficiently
- The array-based efficient implementation is not straight-forward (circular queue)

Iteration

```
for w in words:
    print (w)
```

What happens here?

Iteration

```
for w in words:
    print (w)
```

What happens here? Or here?

```
for i,w in enumerate(words):
    print (i,w)
```

Iteration

```
for w in words:
    print (w)
```

What happens here? Or here?

```
for i,w in enumerate(words):
    print (i,w)
```

Or here?

```
S = [x**2 for x in range(10)]
```

Iterators

- An iterator is an object that is used to traverse a container, mostly a list
- It can be thought of as a pointer to the current item in the list
- Most programming languages uses this concept
- Iterators are used in the examples on the previous slide
- A **generator** is a way to implement iterator
  - It is a (co)routine that yields a sequence of values
  - It does not return all values at once, but one at a time, and keeps its state
  - Can save memory

Iterators in Python

- **Iterable**:
  - anything that can be looped over or
  - anything that can appear on the right-side of a for-loop or
  - anything you can call with iter() that will return an iterator or
  - an object that defines __iter()__ that returns an iterator, or
  - an object that has a __getitem(i)__ method suitable for indexed lookup.
- **Iterator**:
  - An object with state that remembers where it is during iteration,
  - with a __next()__ method that:
    - returns the next value in the iteration
    - updates the state to refer to the next value
    - signals when it is done by raising StopIteration

Python generator functions

- A generator function contains at least one yield statements
- When called, it (automatically) returns an iterator but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Python generator example

```
def fibonacci(n):
    a, b, c = 0, 1, 0
    while c <= n:
        yield a
        (a, b) = (b, a + b)
        c += 1

for x in fibonacci(10):
    print (x)
```

Iteration vs indexing

- What is the complexity of the following examples
  - If s is an array?
  - If s is a linked list?

```
i = 0
for i in range(len(s))
    print (s[i])

for x in s:
    print (x)
```

Lab package 2

- Searching
  - Implement and try different methods for search (in hash tables and lists)
  - Time them, and compare to theoretical complexity
- Linked lists
  - Implement a linked list for a sorted word frequency list
- Sorting
  - Implement two sorting algorithms using a given API
  - Compare the theoretical complexity with the number of operations used in the different algorithms

Coming up

- Next week
  - Lecture on sorting
  - 2 lab sessions (Tuesday and Wednesday)
- Own work
  - Lab assignment
  - Read up on the theory in Python DS book!