Scientific Programming: Part B

Data structures 1

Luca Bianco - Academic Year 2019-20
luca.bianco@fmach.it

Introduction

Data

In programming languages, data are pieces of information that can be assigned to variables (i.e. **values** that can be assigned to **variables**)

Abstract Data Type (ADT)

A **mathematical model**, defined by a **collection of values** and a **set of operations** that can be performed on them.

Primitive Abstract Data Types

Primitive abstract data types that are **provided directly** by the language

Specification vs. Implementation

Specification

The specification of a type of data is its “manual”. It is a **description of the data** that **does not provide details**

Implementation

The **actual code** (with all the specific details) that **realizes** (i.e. implements) the abstract data type

Example: **Real numbers vs IEEE-754**

- “a **real number** is a value of a continuous quantity that can represent a distance along a line”
- IEEE-754 is a standard that defines the format for the representation of floating point numbers

Sometime they differ!

```
>>> 0.1+0.2
0.30000000000000004
```

Data structures

Data structures

Data structures are collections of data, characterized more by the organization of the data rather than the type of contained data.

How to describe data structures

- a systematic approach to organize the collection of data
- a set of operators that enable the manipulation of the structure

Data structures can be

- **Linear**: if the position of an element relative to the ones inserted before/after does not change
- **Static / Dynamic**: depending on if the content or size can change

Data structures

Type	Java	C++	Python
Sequences	List, Queue, Deque LinkedList, ArrayList, Stack, ArrayDeque	list, forward_list vector stack queue, deque	list tuple deque
Sets	Set TreeSet, HashSet, LinkedHashSet	set unordered_set	set, frozenset
Dictionaries	Map HashTree, HashMap, LinkedHashMap	map unordered_map	dict
Trees	-	-	-
Graphs	-	-	-

Sequence: description

Sequence

A dynamic data structure representing an "ordered" group of elements

- The ordering is not defined by the content, but by the relative position inside the sequence (first element, second element, etc.)
- Values could appear more than once
- Example: [0.1, "alberto", 0.05, 0.1] is a sequence

How the data is organized

Operators

- It is possible to add / remove elements, by specifying their position
  - $s = s_1, s_2, \ldots, s_n$
  - the element s_i is in position pos_i
- It is possible to access *directly* some of the elements of the sequence
  - the beginning and/or the end of the list
  - having a reference to the position
- It is possible to sequentially access all the other elements

What we can do with the data

Sequence: specification (prototype)

SEQUENCE

% Return **True** if the sequence is empty
boolean isEmpty()

% Returns the position of the first element
POS head()

% Returns the position of the last element
POS tail()

% Returns the position of the successor of p
POS next(POS p)

% Returns the position of the predecessor of p
POS prev(POS p)

Sequence: specification (prototype)

SEQUENCE (continue)
% Inserts element v of type OBJECT in position p. % Returns the position of the new element POS insert(POS p, OBJECT v)
% Removes the element contained in position p. % Returns the position of the successor of p, which % becomes successor of the predecessor of p POS remove(POS p)
% Reads the element contained in position p OBJECT read(POS p)
% Writes the element v of type OBJECT in position p write(POS p, OBJECT v)

To build our “Sequence” data structure

SEQUENCE (continue)

% Inserts element v of type OBJECT in position p.
% Returns the position of the new element
POS insert(POS p, OBJECT v)

% Removes the element contained in position p.
% Returns the position of the successor of p, which % becomes successor of the predecessor of p
POS remove(POS p)

% Reads the element contained in position p
OBJECT read(POS p)

% Writes the element v of type OBJECT in position p
write(POS p, OBJECT v)

```
def checkMaxMaf(snpEntry,infoEl,val):
    info = snpEntry[7].split(";")
    found = -1
    for i in range(0,len(info)):
        if(info[i].find(infoEl+"=")>-1):
            found = i

    if(found == -1 and i == len(info)-1):
        print "ERROR: cannot find field " + infoEl
        exit(1)
    else:
        v=float(info[found].split('=')[1])
        maf = min(v,1-v)
        if(maf<= val):
            return True
        else:
            #print infoEl + " " + str(maf)
            return False

def checkMaxMissingGen(snpEntry,infoEl,val):
    info = snpEntry[7].split(";")
```

“specifications”
method prototype
ADT

“implementation”

Python code

Sequence: implementation (sketch)

```python
class mySequence:

    def __init__(self):
        #the sequence is implemented as a list
        self.__data = []

    #isEmpty returns True if sequence is empty, false otherwise
    def isEmpty(self):
        return len(self.__data) == 0

    #head returns the position of the first element
    def head(self):
        if not self.isEmpty():
            return 0
        else:
            return None
    #tail returns the position of the last element
    def tail(self):
        if not self.isEmpty():
            return len(self.__data) -1
        else:
            return None
    #next returns the position of the successor of element
    #in position pos
    def next(self, pos):
        if pos <len(self.__data)-1:
            return pos +1
        else:
            return None

    #prev returns the position of the predecessor of element
    #in position pos
    def prev(self, pos):
        if pos > 0 and pos < len(self.__data):
            return pos  - 1
        else:
            return None
```

```python
#insert inserts the element obj in position pos
#or at the end
def insert(self, pos, obj):
    if pos <len(self.__data):
        self.__data.insert(pos, obj)
        return pos
    else:
        #Not necessary! Already done by list's insert!!!
        self.__data.append(obj)
        return len(self.__data) -1
#remove removes the element in position pos
#(if it exists in the sequence) and returns the index
#of the element that now follows the predecessor of pos
def remove(self, pos):
    #TODO
    pass

#read returns the element in position pos (if
#it exists) or None
def read(self, pos):
    #TODO
    pass

#write changes the object in position pos to new_obj
#if pos is a valid position
def write(self,pos,new_obj):
    #TODO
    pass

#converts the data structure into a string
def __str__(self):
    return str(self.__data)
```

Set: description

Set

A dynamic, non-linear data structure that stores an unordered collection of values without repetitions.

- We can consider a total order between elements as the order defined over their abstract data type, if present.

Operators

- Basic operators:
  - insert
  - delete
  - contains
- Sorting operators
  - Maximum
  - Minimum
- Set operators
  - union
  - intersection
  - difference
- Iterators:
  - `for x in S:`

Set: abstract data type

SET
% Returns the size of the set $\texttt{int}$ len()
% Returns **True** if x belongs to the set; Python: `x in S` **boolean** contains(OBJECT x)
% Inserts x in the set, if not already present add(OBJECT x)
% Removes x from the set, if present discard(OBJECT x)
% Returns a new set which is the union of A and B SET union(SET A, SET B)
% Returns a new set which is the intersection of A and B SET intersection(SET A, SET B)
% Returns a new set which is the difference of A and B SET difference(SET A, SET B)

Set: implementation (exercise)

```python
class MySet:
    def __init__(self, elements):
        #HOW are we gonna implement the set?
        #Shall we use a list, a dictionary?
        pass

    #let's specify the special operator for len
    def __len__(self):
        pass

    #this is the special operator for in
    def __contains__(self, element):
        pass

    #we do not redefine __add__ because that is for S1 + S2
    #where S1 and S2 are sets
    def add(self,element):
        pass

    def discard(self,element):
        pass

    def iterator(self):
        pass

    def __str__(self):
        pass

    def union(self, other):
        pass
    def intersection(self, other):
        pass

    def difference(self, other):
        pass
```

SET
% Returns the size of the set int len()
% Returns **True** if x belongs to the set; Python: `x in S` **boolean** contains(OBJECT x)
% Inserts x in the set, if not already present add(OBJECT x)
% Removes x from the set, if present discard(OBJECT x)
% Returns a new set which is the union of A and B SET union(SET A, SET B)
% Returns a new set which is the intersection of A and B SET intersection(SET A, SET B)
% Returns a new set which is the difference of A and B SET difference(SET A, SET B)

Dictionary

Dictionary

Abstract data structure that represents the mathematical concept of partial function $R : D \to C$, or key-value association

- Set D is the domain (elements called keys)
- Set C is the codomain (elements called values)

Operators

- Lookup the value associated to a particular key, if present, `None` otherwise
- Insert a new key-value association, deleting potential association that are already present for the same key
- Remove an existing key-value association

Dictionary: ADT

DICTIONARY

% Returns the value associated to key k, if present; returns **none** otherwise

OBJECT lookup(OBJECT k)

% Associates value v to key k

insert(OBJECT k, OBJECT v)

% Removes the association of key k

remove(OBJECT k)

Linked lists

List (Linked List)

A sequence of memory objects, containing arbitrary data and 1-2 pointers to the next element and/or the previous one

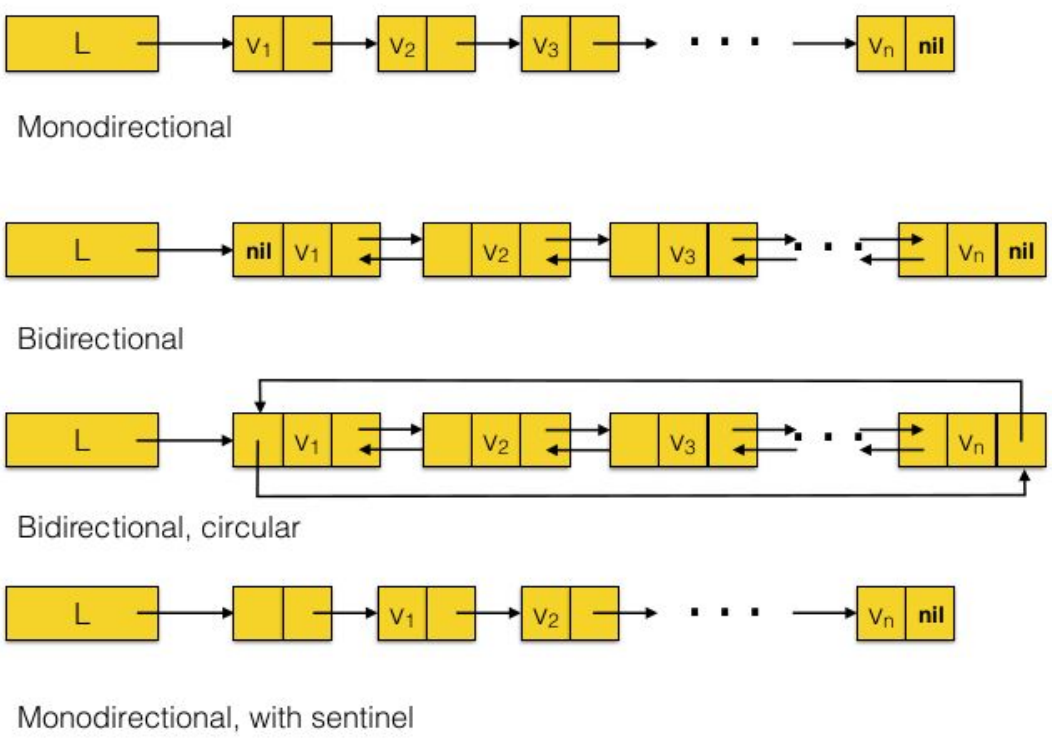
Note

- Contiguity in the list $\nRightarrow$ contiguity in memory
- All the operations require $O(1)$, but in some cases you need a lot of single operations to complete an action

Possible implementations

- Bidirectional / Monodirectional
- With sentinel / Without sentinel
- Circular / Non-circular

Linked lists (types)



Linked lists are dynamic collections of **objects and pointers** (either 1 or 2) that **point to the next** element in the list **or to both the next and previous** element in the list.

Example: monodirectional list in python

Monodirectional list

%adds a node **n** to the Monodirectional list placing it as the **head**

`add(node n)`

%searches a node n and returns True if it is found, false otherwise

`boolean search(node n)`

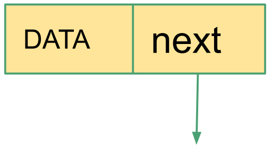
%removes a node n if it is found, does nothing otherwise

`remove(node n)`

%produces the string representation of the Monodirectional list as: el1 -> el2 -> … -> eln

`__str__()`

Node

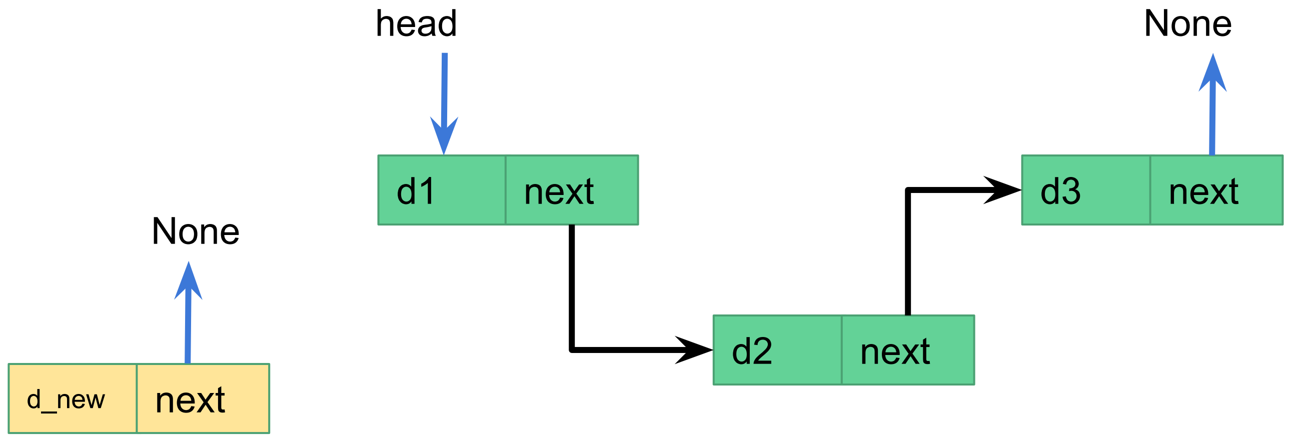


A list is a sequence of nodes, the first of which is the **head.**

Elements are added **at the beginning** and become the new head

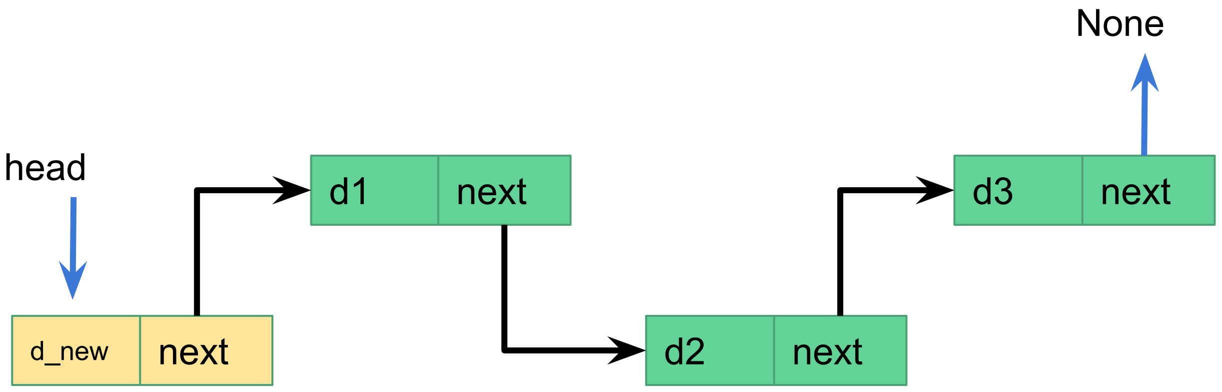
Example: monodirectional list in python

Add one element (d_new)



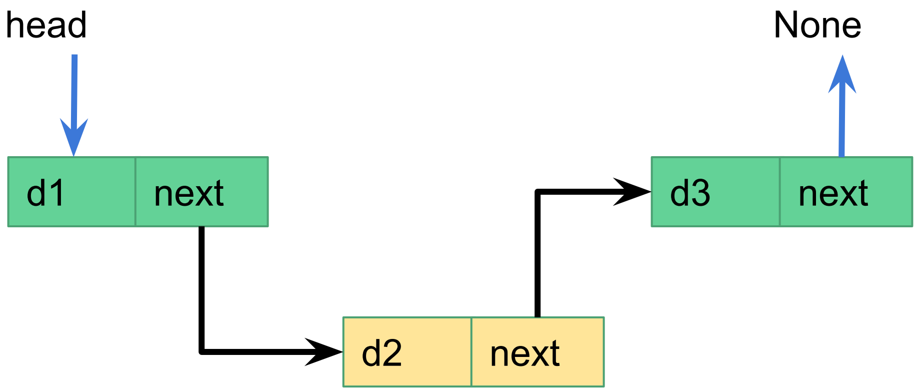
Monodirectional list in python: add

Add one element (d_new)



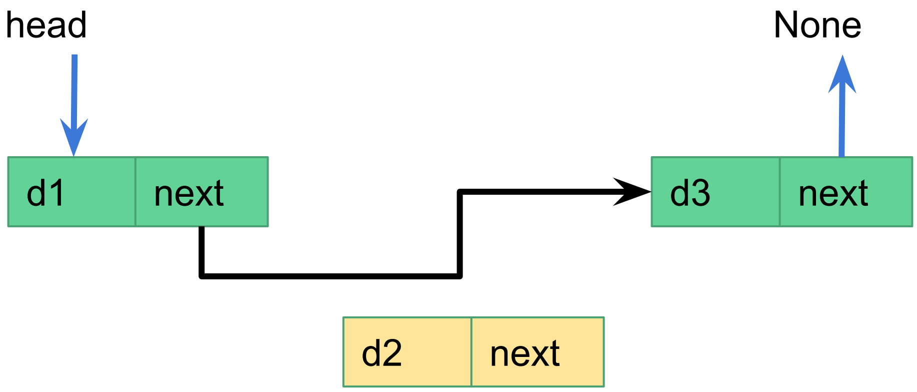
Monodirectional list in python: remove

Remove one element (d2)



Monodirectional list in python: remove

Remove one element (d2)



The code

```python
""" Can place this in Node.py"""
class Node:
    def __init__(self, data):
        self.__data = data
        self.__next = None

    def get_data(self):
        return self.__data

    def set_data(self, newdata):
        self.__data = newdata

    def get_next(self):
        return self.__next

    def set_next(self, node):
        self.__next = node


    def __str__(self):
        return str(self.__data)

    #for sorting
    def __lt__(self, other):
        return self.__data < other.__data
```

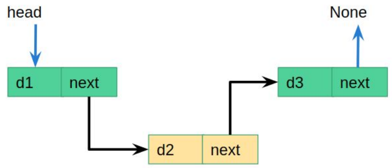
```python
class MonodirList:
    def __init__(self):
        self.__head = None #None is the sentinel!

    def add(self,node):
        if type(node) != Node:
            raise TypeError("node is not of type Node")
        else:
            node.set_next(self.__head)
            self.__head = node

    def search(self, item):
        current = self.__head
        found = False
        while current != None and not found:
            if current.get_data() == item:
                    found = True
            else:
                    current = current.get_next()
        return found

    def remove(self,item):
        current = self.__head
        prev = None
        found = False
        while not found and current != None:
            if current.get_data() == item:
                found = True
            else:
                prev = current
                current = current.get_next()
        if found:
            if prev == None:
                self.__head = current.get_next()
            else:
                prev.set_next(current.get_next() )

    def __str__(self):
        if self.__head != None:
            dta = str(self.__head.get_data())
            cur_el = self.__head.get_next()
            while cur_el != None:
                dta += " -> " + str(cur_el.get_data())
                cur_el = cur_el.get_next()
            return str(dta)
        else:
            return ""
```

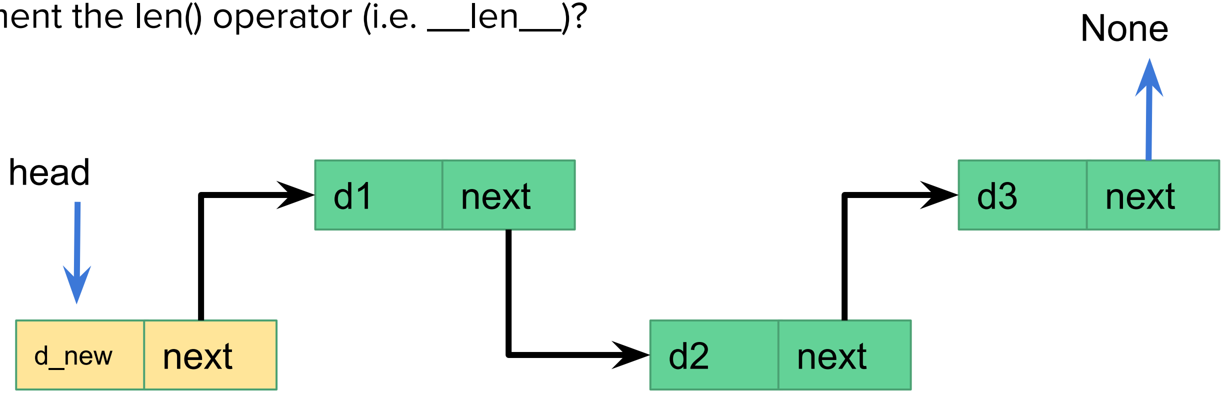



```python
if __name__ == "__main__":
    ML = MonodirList()
    for i in range(1,50,10):
        n = Node(i)
        ML.add(n)
    print(ML)
    print("Adding 111")
    new_n = Node(111)
    ML.add(new_n)
    print("Adding 27")
    new_n2 = Node(27)
    ML.add(new_n2)
    print(ML)
    print("Removing 1")
    ML.remove(1)
    print(ML)
    print("Removing 1")
    ML.remove(1)
    print("Removing 111")
    print("Removing 31")
    ML.remove(111)
    ML.remove(31)
    print(ML)
```

```
41 -> 31 -> 21 -> 11 -> 1
Adding 111
Adding 27
27 -> 111 -> 41 -> 31 -> 21 -> 11 -> 1
Removing 1
27 -> 111 -> 41 -> 31 -> 21 -> 11
Removing 1
Removing 111
Removing 31
27 -> 41 -> 21 -> 11
```

Monodirectional list in python: len?

How could we implement the len() operator (i.e. __len__)?



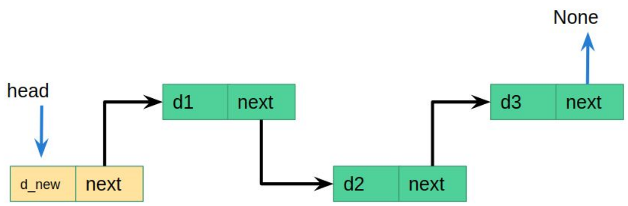
Go from first to last element and sum

Monodirectional list in python: __len__()?

How could we implement the len() operator (i.e. __len__)?

The code:

```python
def __len__(self):
    current = self.__head
    length = 0
    while current != None:
        length += 1
        current = current.get_next()
    return length
```



Complexity is **O(n)**.
Is it possible to improve this?

Monodirectional list in python: __len__()?

Faster __len__().
Idea: store and update the number of elements present

The code:

```python
class MonodirList:
    def __init__(self):
        self.__head = None #None is the sentinel!
        self.__len = 0

    def add(self,node):
        if type(node) != Node:
            raise TypeError("node is not of type Node")
        else:
            node.set_next(self.__head)
            self.__head = node
            self.__len += 1
```

...

```python
def __len__(self):
    return self.__len
```

```python
def remove(self,item):
    current = self.__head
    prev = None
    found = False
    while not found and current != None:
        if current.get_data() == item:
            found = True
        else:
            prev = current
            current = current.get_next()
    if found:
        if prev == None:
            self.__head = current.get_next()
        else:
            prev.set_next(current.get_next() )
        self.__len -= 1
```

Complexity is O(1).

Exercise: How about O(1) min/max values? Hint: change again __init__, add, and remove.

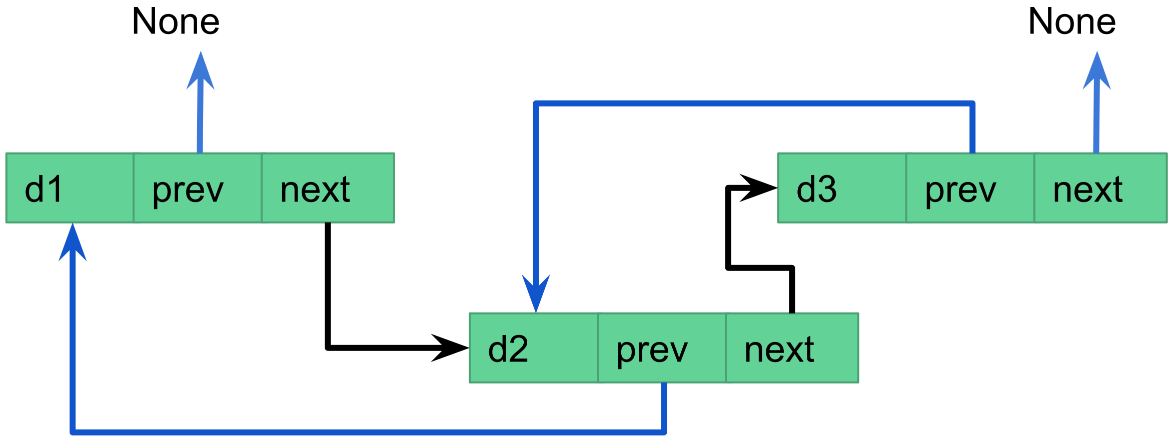
Bidirectional linked list

Each node now has:

- the data
- a prev pointer
- a next pointer

prev pointer of the **first** element in the list is **None**

next pointer of the **last** element is **None**



Bidirectional linked list

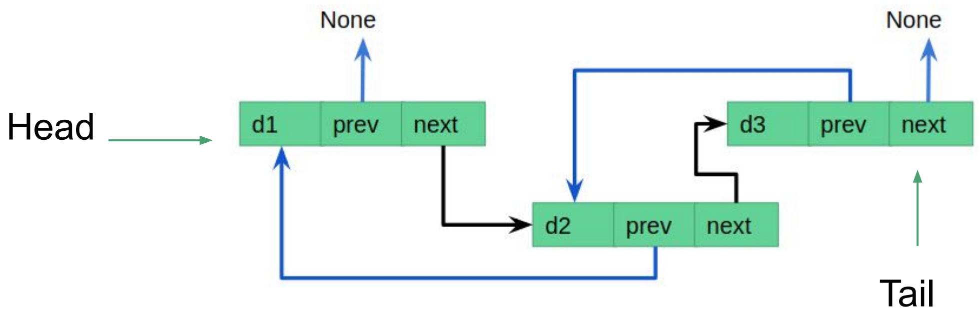
Each node now has:

- the data
- a prev pointer
- a next pointer

prev pointer of the **first** element in the list is **None**

next pointer of the **last** element is **None**

The list can have a **head** and **tail** pointer

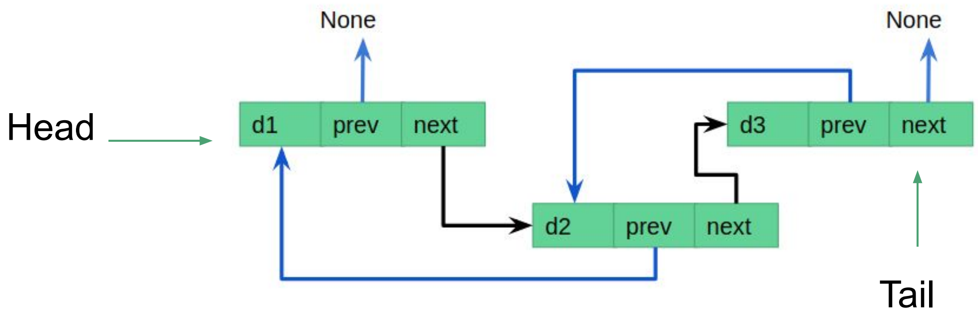


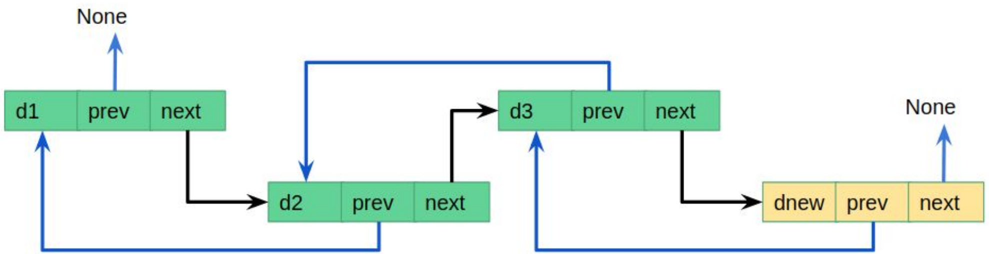
Bidirectional linked list: append

Each node now has:

- the data
- a prev pointer
- a next pointer

Append: add a node as next of the current tail



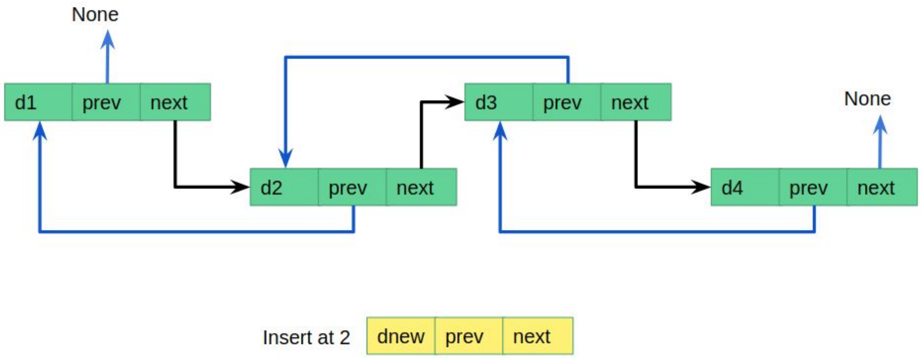


Bidirectional linked list: insert at/remove

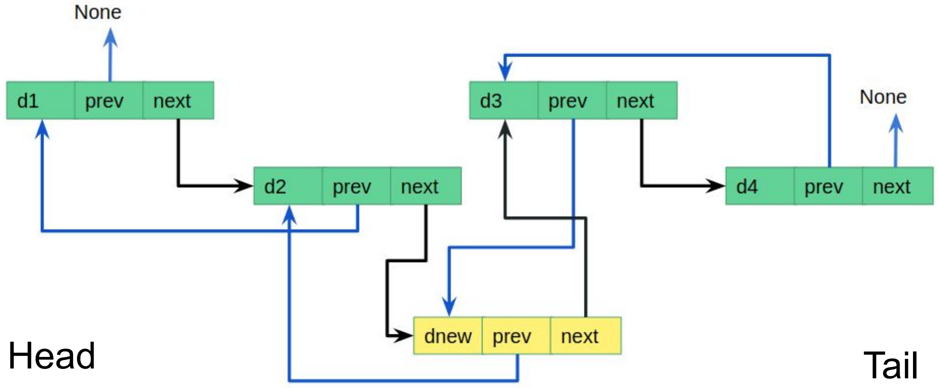
Each node now has:

- the data
- a prev pointer
- a next pointer

Insert at/remove :
reach the correct position and update the next/prev pointers of the **three** involved nodes



Insert at 2
First loop until you reach 2 (cur = cur.get_next())



Dynamic Vectors

Lists in Python implemented through dynamic vectors

- A vector of a given size (initial capacity) is allocated
- When inserting an element before the end, all elements have to be moved - cost $O(n)$
- When inserting an element at the end (append), the cost is $O(1)$ (just writing the element at first available slot)

Problem:

- It is not known how many elements have to be stored
- The initial capacity could be insufficient

Solution:

- A new (larger) vector is allocated, the content is copied in the new vector, the old vector is released

Dynamic Vectors

Lists in Python implemented through dynamic vectors

- A vector of a given size (initial capacity) is allocated
- When inserting an element before the end, all elements have to be moved - cost $O(n)$
- When inserting an element at the end (append), the cost is $O(1)$ (just writing the element at first available slot)

Problem:

- It is not known how many elements have to be stored
- The initial capacity could be insufficient

Solution:

- A new (larger) vector is allocated, the content is copied in the new vector, the old vector is released

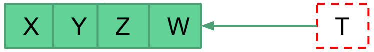
Dynamic Vectors

Lists in Python implemented through dynamic vectors

- A vector of a given size (initial capacity) is allocated
- When inserting an element before the end, all elements have to be moved - cost $O(n)$
- When inserting an element at the end (append), the cost is $O(1)$ (just writing the element at first available slot)

Problem:

- It is not known how many elements have to be stored
- The initial capacity could be insufficient



Solution:

- A new (larger) vector is allocated, the content is copied in the new vector, the old vector is released

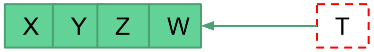
Dynamic Vectors

Lists in Python implemented through dynamic vectors

- A vector of a given size (initial capacity) is allocated
- When inserting an element before the end, all elements have to be moved - cost $O(n)$
- When inserting an element at the end (append), the cost is $O(1)$ (just writing the element at first available slot)

Problem:

- It is not known how many elements have to be stored
- The initial capacity could be insufficient



Solution:

- A new (larger) vector is allocated, the content is copied in the new vector, the old vector is released

Dynamic Vectors

Lists in Python implemented through dynamic vectors

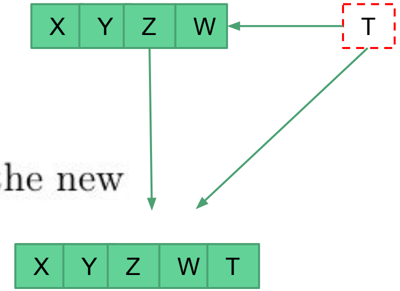
- A vector of a given size (initial capacity) is allocated
- When inserting an element before the end, all elements have to be moved - cost $O(n)$
- When inserting an element at the end (append), the cost is $O(1)$ (just writing the element at first available slot)

Problem:

- It is not known how many elements have to be stored
- The initial capacity could be insufficient

Solution:

- A new (larger) vector is allocated, the content is copied in the new vector, the old vector is released



Dynamic Vectors

Question

Which is the best approach?

Approach 1

If the old vector has size n, allocate a new vector of size dn. For example, $d = 2$

doubling

Approach 2

If the old vector has size n, allocate a new vector of size $n + d$, where d is a constant. For example, $d = 16$

increment

Dynamic Vectors: Amortized cost (doubling)

Actual cost of an append() operation:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{otherwise} \end{cases}$$

Assumptions:

- Initial capacity: 1
- Writing cost: $\Theta(1)$

ex. 3 elements in. Append now: 1 operation

n	cost
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Doubling

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (doubling)

Actual cost of an append() operation:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{otherwise} \end{cases}$$

Assumptions:

- Initial capacity: 1
- Writing cost: $\Theta(1)$

ex. 4 elements in.

n	cost
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Doubling

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (doubling)

Actual cost of an append() operation:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{otherwise} \end{cases}$$

Assumptions:

- Initial capacity: 1
- Writing cost: $\Theta(1)$

ex. 4 elements in. Append now: cost 1 + 4 allocations

n	cost
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Doubling

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (doubling)

Actual cost of an append() operation:

$$c_i = \begin{cases} i & \exists k \in \mathbb{Z}_0^+ : i = 2^k + 1 \\ 1 & \text{otherwise} \end{cases}$$

Assumptions:

- Initial capacity: 1
- Writing cost: $\Theta(1)$

ex. 4 elements in. Append now: cost 1 + 4 allocations

n	cost
1	1
2	$1 + 2^0 = 2$
3	$1 + 2^1 = 3$
4	1
5	$1 + 2^2 = 5$
6	1
7	1
8	1
9	$1 + 2^3 = 9$
10	1
11	1
12	1
13	1
14	1
15	1
16	1
17	$1 + 2^4 = 17$

Doubling

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (doubling)

Actual cost of n operations append():

$$T(n) = \sum_{i=1}^{n} c_i$$
$$= n + \sum_{j=0}^{\lfloor \log n \rfloor} 2^j$$
$$= n + 2^{\lfloor \log n \rfloor + 1} - 1$$
$$\leq n + 2^{\log n + 1} - 1$$
$$= n + 2n - 1 = O(n)$$

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

Amortized cost of a single append():

$$T(n)/n = \frac{O(n)}{n} = O(1)$$

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (increment)

Actual cost of an append() operation:

$$c_i = \begin{cases} i & (i \bmod d) = 1 \\ 1 & \text{altrimenti} \end{cases}$$

Assumptions

- Increment: d
- Initial size: d
- Writing cost: $\Theta(1)$

Example

- $d = 4$

n	cost
1	1
2	1
3	1
4	1
5	$1 + d = 5$
6	1
7	1
8	1
9	$1 + 2d = 9$
10	1
11	1
12	1
13	$1 + 3d = 13$
14	1
15	1
16	1
17	$1 + 4d = 17$

Increment

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic Vectors: Amortized cost (increment)

Actual cost of n operations append():

$$
\begin{aligned}
T(n) &= \sum_{i=1}^{n} c_i \\
&= n + \sum_{j=1}^{\lfloor n/d \rfloor} d \cdot j \\
&= n + d \sum_{j=1}^{\lfloor n/d \rfloor} j \\
&= n + d\frac{(\lfloor n/d \rfloor + 1)\lfloor n/d \rfloor}{2} \\
&\leq n + \frac{(n/d + 1)n}{2} = O(n^2)
\end{aligned}
$$

$$\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$$

Amortized cost of a single append():

$$T(n)/n = \frac{O(n^2)}{n} = O(n)$$

Amortized analysis tells how the average of the performance of a set of operations on a large data set scales.

We consider a block of operations.

Dynamic vectors: growth factor

Language	Data structure	Expansion factor
GNU C++	`std::vector`	2.0
Microsoft VC++ 2003	`vector`	1.5
Python	`list`	1.125
Oracle Java	`ArrayList`	2.0
OpenSDK Java	`ArrayList`	1.5

Performance of Python’s data structures

The choice of the data structure has implications on the performances

It is important to know the properties of built-in structures to use them properly!

Performance of Python's lists

lists are dynamic vectors!

Operator		Worst case	Worst case amortized
`L.copy()`	Copy	$O(n)$	$O(n)$
`L.append(x)`	Append	$O(n)$	$O(1)$
`L.insert(i,x)`	Insert	$O(n)$	$O(n)$
`L.remove(x)`	Remove	$O(n)$	$O(n)$
`L[i]`	Index	$O(1)$	$O(1)$
`for x in L`	Iterator	$O(n)$	$O(n)$
`L[i:i+k]`	Slicing	$O(k)$	$O(k)$
`L.extend(s)`	Extend	$O(k)$	$O(n+k)$
`x in L`	Contains	$O(n)$	$O(n)$
`min(L), max(L)`	Min, Max	$O(n)$	$O(n)$
`len(L)`	Get length	$O(1)$	$O(1)$

Reality check

Operator		Worst case	Worst case amortized
`L.copy()`	Copy	$O(n)$	$O(n)$
`L.append(x)`	Append	$O(n)$	$O(1)$
`L.insert(i,x)`	Insert	$O(n)$	$O(n)$
`L.remove(x)`	Remove	$O(n)$	$O(n)$
`L[i]`	Index	$O(1)$	$O(1)$
`for x in L`	Iterator	$O(n)$	$O(n)$
`L[i:i+k]`	Slicing	$O(k)$	$O(k)$
`L.extend(s)`	Extend	$O(k)$	$O(n+k)$
`x in L`	Contains	$O(n)$	$O(n)$
`min(L), max(L)`	Min, Max	$O(n)$	$O(n)$
`len(L)`	Get length	$O(1)$	$O(1)$

```python
import time

from collections import deque

N = 750
L = []
start = time.time()
for i in range(N):
    for j in range(N):
        L.insert(0, i)        # O(n)

end = time.time()
print("[list: insert] {:.2f}s elapsed".format(end-start))
L=[]
start = time.time()
for i in range(N):
    for j in range(N):
        L.append(i)           # O(1)

end = time.time()
print("[list: append] {:.2f}s elapsed".format(end-start))

start = time.time()
for i in range(len(L)):       # O(n)
    L.pop(0)
end = time.time()
print("[list: remove] {:.2f}s elapsed".format(end-start))
```

```
[list: insert] 88.90s elapsed
[list: append] 0.04s elapsed
[list: remove] 30.33s elapsed
```

```python
D = deque()
start = time.time()
for i in range(N):
    for j in range(N):
        D.insert(0, i)        # O(1)

end = time.time()
print("[deque: insert] {:.2f}s elapsed".format(end-start))
D = deque()
start = time.time()
for i in range(N):
    for j in range(N):
        D.append(i)           # O(1)
end = time.time()

print("[deque: append] {:.2f}s elapsed".format(end-start))

start = time.time()
for i in range(len(D)):
    D.popleft()               # O(1)
end = time.time()
print("[deque: remove] {:.2f}s elapsed".format(end-start))
```

```
[deque: insert] 0.06s elapsed
[deque: append] 0.04s elapsed
[deque: remove] 0.04s elapsed
```

collections.deque
https://docs.python.org/3.7/library/collections.html#collections.deque