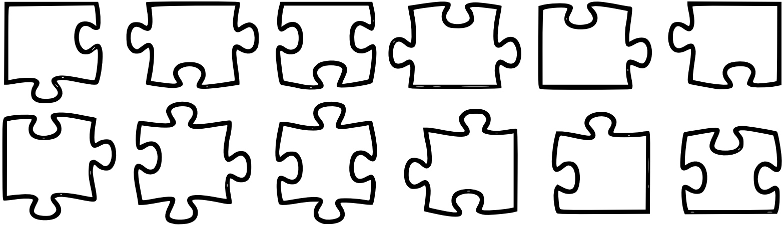
COMP20230: Data Structures & Algorithms
Lecture 7: Data Structures

Dr Andrew Hines

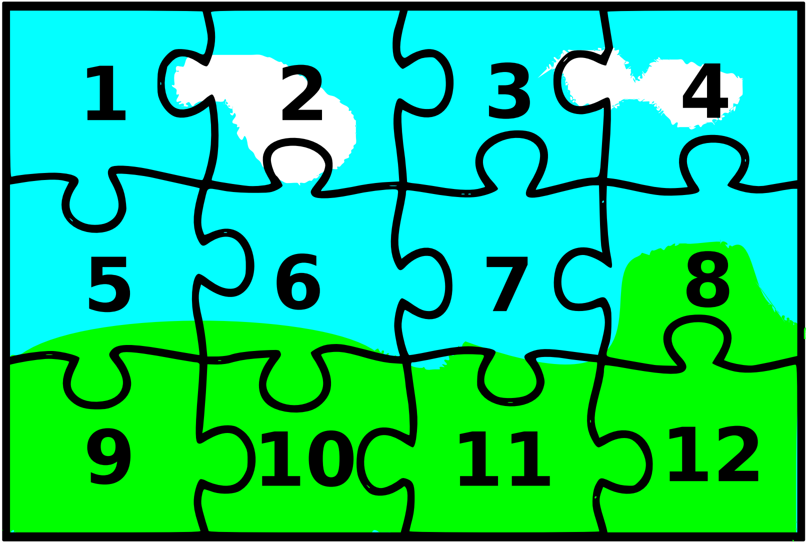
Office: E3.13 Science East
School of Computer Science
University College Dublin

andrew.hines@ucd.ie

Feeling Puzzled?



Algorithms and Data Structures



Outline

Last Week: Algorithms

Recursion-based algorithms – iterative and recursive complexity.

This week: Data Structures

- **Abstract Data Types** (ADT) – e.g., Sequence or Queue
- **Concrete Data Structures**: Array-based vs. Linked-list-based – e.g. data structure implementations of ADTs.

Take home message

ADT: describes the operations on a data type.
CDS: implements them.
Trade-offs: The Array-based sequence ADT has good complexity for accessing/modifying values but not for inserting/deleting

Algorithms and Data Structures are

two sides of the same coin – changing one will change the complexity of the other.

Image: Ephesos coin from 620-600 BC
Source: https://commons.wikimedia.org/wiki/File:Ephesos_620-600_BC.jpg

Abstract Data Types (ADTs)

- To abstract ourselves from the particular implementations of algorithms (in programming languages) we use a pseudo-code
- Likewise we use Abstract Data Types as a higher level for data structures

For a given programming language, different data types and data structures can be used for the same job

Similar concepts, different implementations

- no (Python) tuple in Java
- (Python) list $\sim$ (Java) ArrayList but no slicing

We can use abstractions that describe the general ideas (semantics/behaviours) about a class of data types.

Example: Sequence ADT

- The sequence ADT is one of the most fundamental data types
- Many others ADTs are variations on sequence (e.g., stack and queue)

Sequence

- Consists of a homogeneous ordered collection of objects of any type

Example: Sequence ADT

- The sequence ADT is one of the most fundamental data types
- Many others ADTs are variations on sequence (e.g., stack and queue)

Sequence

- Consists of a homogeneous ordered collection of objects of any type

What sequences are there in Python?

Python Examples

List, Tuple, str

Sequence ADT: Main Operations

- `get_elem_at_rank(r)`: Return the element of S with rank r; an error occurs if r<0 or r> n - 1
  Input: Integer; **Output**: Object
- `set_elem_at_rank(r,e)`: Replace the element at rank r with e; an error occurs if r < 0 or r > n - 1.
  Input: Integer r, Object e; **Output**: none
- `insert_elem_at_rank(r,e)`: Insert a new element into S which will have rank r; an error occurs if r < 0 or r > n - 1.
  Input: Integer r, Object e; **Output**: none
- `remove_elem_at_rank(r)`: Remove from S the element at rank r; an error occurs if r<0 or r>n-1.
  Input: Integer; **Output**: none

Sequence ADT: Secondary Operations

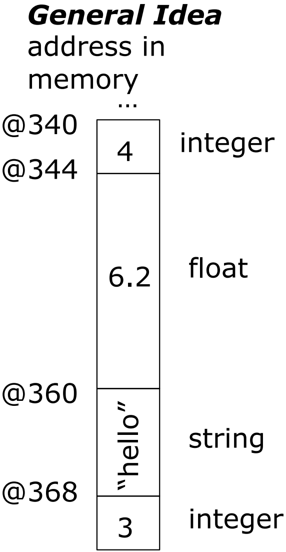
- `insert_first(e), insert_last(e)`:: Insert an element at one of the ends of the sequence
  Input: element; **Output**: none
- `insert_after(p, e), insert_before(p, e)`: Insert an element after or before a position
  Input: position and element; **Output**: none

Sequence ADT: Support Operations

- `size()`:: Returns the number of objects in the sequence
  Input: none; **Output**: integer
- `is_empty()`: Return a boolean indicating if S is empty.
  Input: none; **Output**: boolean

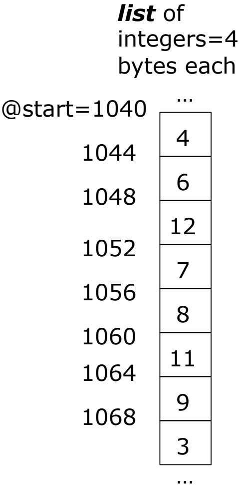
Machine Representation of Data

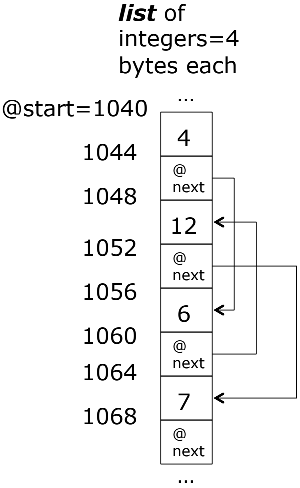
Data: Objects, Elements



Machine Representation of Data

Example: a list in Python



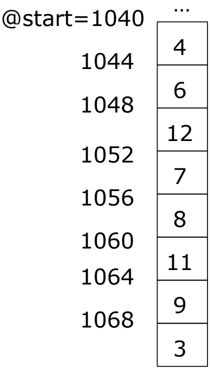


Machine Representation of Data

Data Structure Implementations: Arrays and Linked Lists

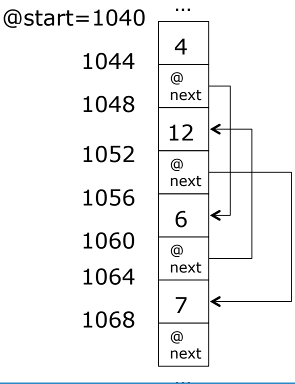
Array

list of
integers=4
bytes each



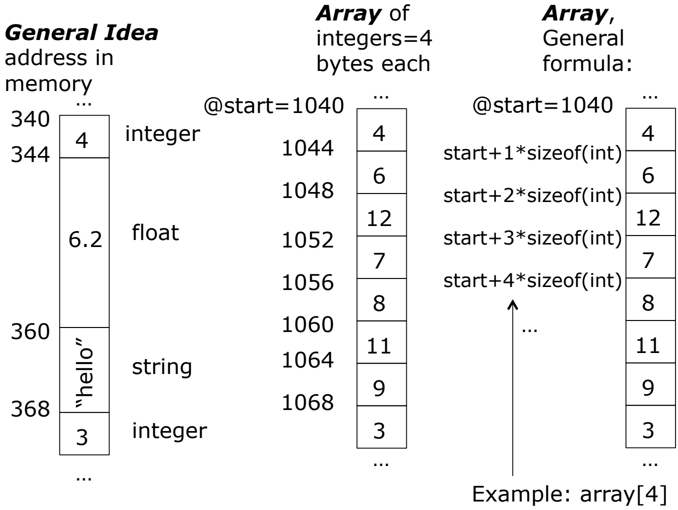
Linked List

list of
integers=4
bytes each



Arrays

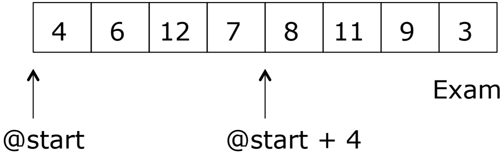
Addressing arrays in memory



Array-based data structures

Characteristics of an array:

+ Elements can be accessed using an **index**
+ Index can be computed very efficiently: access $= \mathcal{O}(\mathbf{1})$
+ Modification of an element is also very efficient $= \mathcal{O}(\mathbf{1})$
– Modification of an array is complex (sometimes impossible)
– Editing/ deleting $= \mathcal{O}(\mathbf{n})$(resizing/ removing elements)



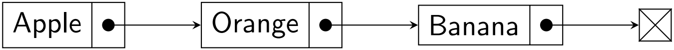
Example: array[4]

Linked List Characteristics

Linked lists are a data structure

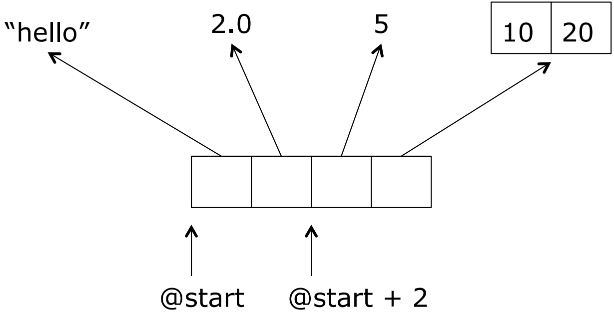
Main characteristic of an linked-list is that each element is linked to its successor using a pointer

- Access $= \mathcal{O}(\mathbf{n})$ (worst case)
- Modification: $\mathcal{O}(\mathbf{n})$ (worst case)
- BUT the modification of the array is simpler: $\mathcal{O}(\mathbf{1})$: once at the right position $\mathcal{O}(\mathbf{n}) + \mathcal{O}(\mathbf{1})$



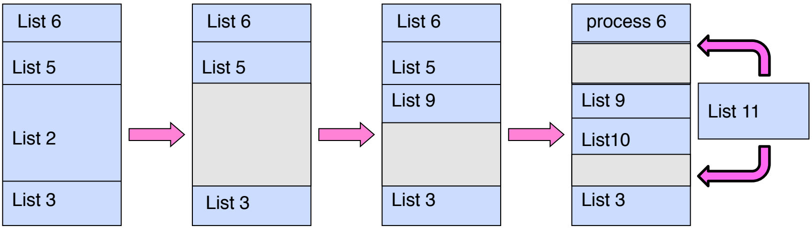
(Note: This is for reference, to help put the puzzle pieces together but we will come back to linked lists later in the module)

Python List



Array Based Data Structure

Problem with memory allocation?



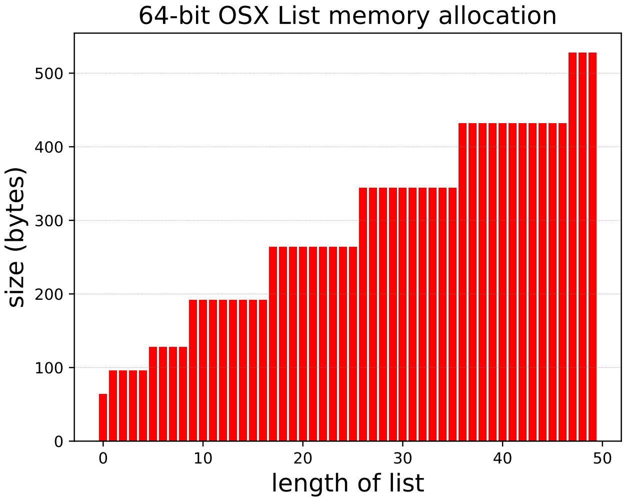
Here I've got a block of memory for storing lists. I delete list #2 and add list #9 in the memory I freed up. Then I added list #10 and deleted list #5. I have enough space for list #11 but not contiguously.

Python Lists

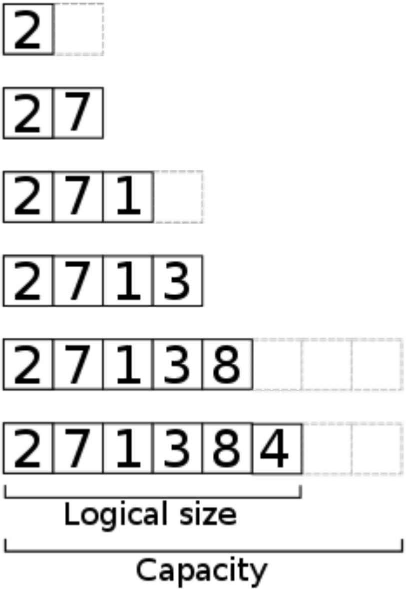
List is Dynamic: Append method

```
import sys
import matplotlib.pyplot as plt
my_list = [] #This is my list
my_list_size =[] #This list will store the size

for i in range(50):
    a = len(my_list)+1
    b = sys.getsizeof(my_list)
    print("Length: ", a, "; Size in bytes: ", b)
    my_list.append(i)
    my_list_size.append(b)

fig, ax = plt.subplots()
plt.bar(my_list,my_list_size, color='r')

ax.grid(color='gray', linestyle=':', linewidth=.2, axis='y' )
ax.set_title('64-bit OSX list memory allocation',fontsize=16)
ax.set_xlabel('length of list',fontsize=16)
ax.set_ylabel('size (bytes)',fontsize=16)
plt.savefig('listSizeFig.pdf')
```

List Memory Growth



Dynamic/Unbounded Array

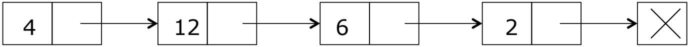


ADT Sequence

A sequence can be represented using an **array**:

4	6	12	7	8	11	9	3

Or a **linked list**: a linear collection of elements pointing to each other (using pointers)



Array-based Sequence

get and set patterns

ADTs often follow common patterns. Get and set methods are usually quite simple – error checking and data access or edit.

Algorithm **get_elem_at_rank**

Input: A an array representing a sequence, $r \geq 0$ an integer representing the rank of the queried object
Output: e the element at rank r

```
if r > size of A then
    return ERROR
end if
return A[r]
```

Algorithm **set_elem_at_rank**

Input: A an array representing a sequence, $r \geq 0$ an integer representing the rank of the queried object
Output: e has been put at rank r

```
if r > size of A then
    return ERROR
end if
A[r] ← e
```

Array-based Sequence

insert patterns

insert can be more complex in terms of the error checking, validation, and memory management.

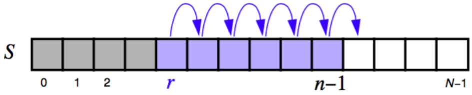
Algorithm **insert_element_at_rank**

Input: A an array representing a sequence, $r \geq 0$ an integer representing the rank where the object will be inserted, e the element to insert

Output: A grows by 1 and e is inserted at rank r

increase the size of A (by 1) if needed
for $i = n-1, n-2, \ldots$ **to** r **do**
  $A[i+1] \leftarrow A[i]$
end for
$A[r] \leftarrow e$

A graphical representation:



Array-based Sequence

delete patterns

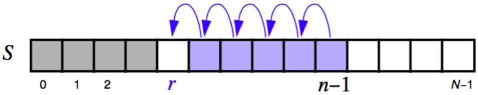
similar to insert, delete or remove can be more complex in terms of the error checking, validation, and memory management/deallocation.

Algorithm **remove_elem_at_rank**

Input: A an array representing a sequence, $r \geq 0$ an integer representing the rank of the object to be deleted
Output: A decreases its size by 1 and the element at rank r is removed

  if $r + 1 \geq sizeof A$ **then**
    return ERROR
  end if
  for $i = r, r + 1, \ldots$ **to** $n - 2$ **do**
    $A[i] \leftarrow A[i + 1]$
  end for
  shrink the size of A by one

A graphical representation:



Array-based Sequence

Other patterns

utility methods are helpful to allow you to perform logical checks on your data structure in a standard way.

Algorithm **is_empty**

Input: A an array representing a sequence

Output: $true$ if the sequence is empty

return $size\ of\ A < 0$

Complexity Analysis: Sequence

Operations using an array. We will compare to Linked list complexity next time.

Operations	Array
`size, is_empty`	$\mathcal{O}(1)$
`get_elem_at_rank`	$\mathcal{O}(1)$
`set_elem_at_rank`	$\mathcal{O}(1)$
`insert_element_at_rank`	$\mathcal{O}(n)$
`remove_element_at_rank`	$\mathcal{O}(n)$
`insert_first, insert_last`	$\mathcal{O}(1)$
`insert_after, insert_before`	$\mathcal{O}(n)$

Conclusion

- We've seen that the concept of ADT (Abstract Data Type) encapsulates the behaviours of real Data Structures
    - Arrays and Linked-lists implement (concrete) data structures from ADTs
- The Array-based sequence is good for accessing and modifying but not so efficient for inserting/deleting

Take home message

ADT: describes the operations on a data type.
CDS: implements them.
Trade-offs: Always about trade-offs. Data structure storage complexity vs access complexity, e.g. the Array-based sequence ADT has good complexity for accessing/modifying values but not for inserting/deleting