Algorithms and Data Structures

B1. Arrays and Linked Lists

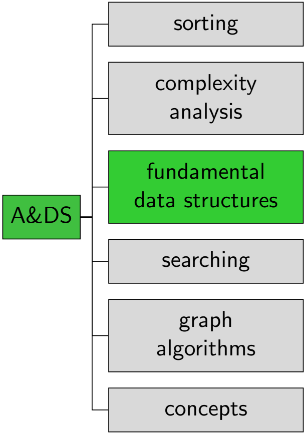
Gabriele Röger

University of Basel

March 27/April 3, 2024

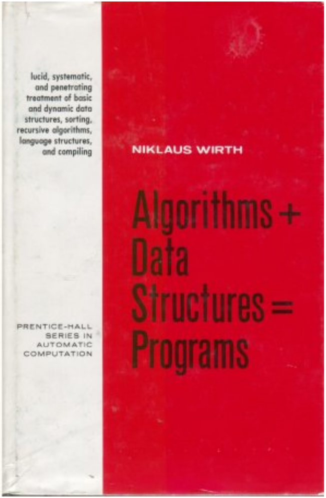
Data Structures

Content of the Course

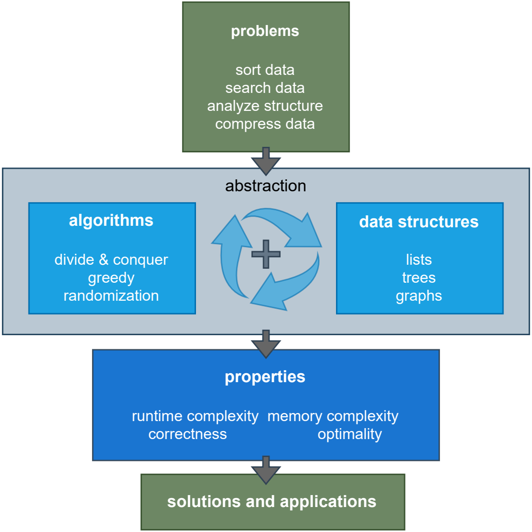


Data Structures

- Programming goes beyond writing algorithms.
  - Organisation of data is central.
- Elegant data structures lead to elegant code.
- Programmers. . .
  - need a catalogue of data structures, and
  - need to know their characteristics.



Overview


problems
sort data
search data
analyze structure
compress data
abstraction
algorithms
divide & conquer
greedy
randomization
data structures
lists
trees
graphs
properties
runtime complexity
memory complexity
correctness
optimality
solutions and applications

Data Structures

> Bad programmers worry about the code. Good programmers worry about data structures and their relationships.
>
> Linus Torwalds

Data Structures

> Show me your algorithm and conceal your data structures, and I shall continue to be mystified.
>
> Show me your data structures, and I won't usually need your algorithm; it will be obvious.
>
> Fred Brooks (paraphrased)

Arrays

Data Structure: Array

- Arrays are one of the fundamental data structures, that can be found in (almost) every programming language.
- An array stores a sequence of elements (of the same memory size) as a contiguous sequence of bytes in memory.
- The number of elements is fixed.
- We can access elements by their index.

In Java:

```
byte[] myByteArray = new byte[100];
char[] myCharArray = new char[50];
```

Example: char Array

- One char occupies 1 byte.
- The first element is at memory address 2000 (7D0 in hexadecimal).
- The first element has index 0.
- Then the element with index i is at address $2000 + \mathrm{i}$.

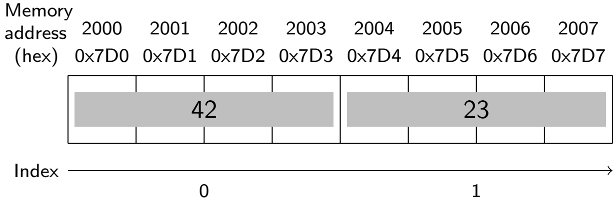
Memory address	2000	2001	2002	2003	2004	2005	2006	2007	2008	2009	2010
(hex)	0x7D0	0x7D1	0x7D2	0x7D3	0x7D4	0x7D5	0x7D6	0x7D7	0x7D8	0x7D9	0x7DA
	h	e	l	l	o	␣	w	o	r	l	d
Index →	0	1	2	3	4	5	6	7	8	9	10

Array: Position of i-th Element Easy to Compute

In general:

- First position typically indexed with 0 or 1.
  In the following, s for the index of the first element.
- Suppose the array starts at memory address a and each array element occupies b bytes.
- Then the element with index i occupies bytes $a + b(i - s)$ to $a + b(i - s + 1) - 1$.

With 32-bit integers (4 byte)



Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size *n* of the array)?
  - `get(i)` – return element at position `i`
  - `set(i, x)` – write object `x` to position `i`
  - `length()` – return length of the array
  - `find(x)` – return index of element `x` or None if not included.

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i`
  - `length()` – return length of the array
  - `find(x)` – return index of element `x` or None if not included.

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i` $\rightsquigarrow \Theta(1)$
  - `length()` – return length of the array
  - `find(x)` – return index of element `x` or None if not included.

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i` $\rightsquigarrow \Theta(1)$
  - `length()` – return length of the array $\rightsquigarrow \Theta(1)$
  - `find(x)` – return index of element `x` or None if not included.

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i` $\rightsquigarrow \Theta(1)$
  - `length()` – return length of the array $\rightsquigarrow \Theta(1)$
  - `find(x)` – return index of element `x` or None if not included.
    $\rightsquigarrow$ iterates over the array and stops if element found.
    $\rightsquigarrow$ Best case $\Theta(1)$, Avg. and worst case $\Theta(n)$

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i` $\rightsquigarrow \Theta(1)$
  - `length()` – return length of the array $\rightsquigarrow \Theta(1)$
  - `find(x)` – return index of element `x` or None if not included.
    $\rightsquigarrow$ iterates over the array and stops if element found.
    $\rightsquigarrow$ Best case $\Theta(1)$, Avg. and worst case $\Theta(n)$
- What is the memory complexity?

Operations and their Running Time?

- Size of entry is constant for a specific array type (such as an int array).
- After allocating the memory, the array stores
  - the size of the array (number of elements) and
  - the address of the start of the allocated memory.
- What is the running time of the following operations (relative to the size n of the array)?
  - `get(i)` – return element at position `i` $\rightsquigarrow \Theta(1)$
  - `set(i, x)` – write object `x` to position `i` $\rightsquigarrow \Theta(1)$
  - `length()` – return length of the array $\rightsquigarrow \Theta(1)$
  - `find(x)` – return index of element `x` or None if not included.
    $\rightsquigarrow$ iterates over the array and stops if element found.
    $\rightsquigarrow$ Best case $\Theta(1)$, Avg. and worst case $\Theta(n)$
- What is the memory complexity?

Observation

Complexity is direct consequence of data representation.

Lists in Python

- Python lists can contain arbitrarily mixed objects.
  e.g. `["word", 42, ([39, "hi"])]`

Lists in Python

- Python lists can contain arbitrarily mixed objects.
  e.g. `["word", 42, ([39, "hi"])]`
  - Elements "live" somewhere else in memory.
  - The memory range of the array only stores their address.

Lists in Python

- Python lists can contain arbitrarily mixed objects.
  e.g. `["word", 42, ([39, "hi"])]`
  - Elements "live" somewhere else in memory.
  - The memory range of the array only stores their address.
- Python lists do not have a fixed size.
  e.g. `["word", 42, ([39, "hi"])].append(3)`

Lists in Python

- Python lists can contain arbitrarily mixed objects.
  e.g. `["word", 42, ([39, "hi"])]`
  - Elements “live” somewhere else in memory.
  - The memory range of the array only stores their address.
- Python lists do not have a fixed size.
  e.g. `["word", 42, ([39, "hi"])].append(3)`
  → dynamic array

Dynamic Arrays

(Static) arrays have fixed capacity that must be specified at allocation.

- Need arrays that can grow dynamically.
- Runtime complexity of previous operations should be preserved.

Dynamic Arrays

(Static) arrays have fixed capacity that must be specified at allocation.

- Need arrays that can grow dynamically.
- Runtime complexity of previous operations should be preserved.

Additional operations:

- `append(x)` (or `push`) – append element `x` at the end.
- `insert(i, x)` – insert element `x` at position `i`.
- `pop()` - remove the last element.
- `remove(i)` - remove the element at position `i`.

Changing the Array Size: Naive Method

- `append` (and `insert`) increase the size of the array.
- `pop` decreases the size.
- Naive method:
  - Allocate new memory range that is one element larger/smaller.
  - Move all (but the potentially popped) element over.

Changing the Array Size: Naive Method

- `append` (and `insert`) increase the size of the array.
- `pop` decreases the size.
- Naive method:
  - Allocate new memory range that is one element larger/smaller.
  - Move all (but the potentially popped) element over.

With this approach, these operations would take linear time in the current size of the array!

Better Approach: Overallocate Memory

- Allocate more memory than needed for the current array size.
- Distinguish
  - capacity = number of elements that fit in the allocated space.
  - size = number of currently contained elements.

Better Approach: Append/Insert

Append

- If capacity $>$ size:
  - Write the new element to position size and increment size.
- Otherwise (capacity $=$ size):
  - Allocate new memory that is larger than necessary (e.g. twice the previous capacity).
  - Copy all elements to the new memory (release the old one).
  - Update the capacity and continue as in case capacity $>$ size.

Insert at pos i: Analogously but move all elements at positions i to size-1 one position to the right before writing the new element to i.

Better Approach: Pop/Remove

- If capacity much too large (e.g. capacity $> 4 \cdot$ size), move all elements into new smaller memory range (e.g. with half the previous capacity)
- Pop: remove element at position size - 1 and decrement size.
- Remove: remove element at position i and move all elements right of i one position to the left, decrepement size.

Amortized Analysis

- Worst-case analysis often pessimistic: append takes linear time if new memory allocated but in a sequence of append operations, this will happen rarely.
- Amortized analysis determines the average cost of an operation over an entire sequence of operations.
- Don't confuse this with an average-case analysis.
- Different methods
  - Aggregate analysis
  - Accounting method $\leftarrow$ now
  - Potential method

Accounting Method

- Assign charges to operations.
- Some operations charged more or less than they actually cost.
- If charged more: save difference as credit
- If charged less: use up some credit to pay for the difference.
- Credit must be non-negative all the time.
- Then the total amortized cost is always an upper bound on the actual total costs so far.

Accounting Method: Append I

- Append without resize: constant cost (e.g. 1).
  Just insert the element at the right position.
- Append with resize: linear cost (1 for every element).
  - If the append element gets position 2^i ($i \in \mathbb{N}_{>0}$),
  - we first allocate overall space for 2^{i+1} elements, and
  - move all $2^i - 1$ existing elements to the new space.

Accounting Method: Append I

- Append without resize: constant cost (e.g. 1).
  Just insert the element at the right position.
- Append with resize: linear cost (1 for every element).
  - If the append element gets position 2^i ($i \in \mathbb{N}_{>0}$),
  - we first allocate overall space for 2^{i+1} elements, and
  - move all $2^i - 1$ existing elements to the new space.
- Starting from an empty array executing a sequence of append operations, we observe cost sequence
  1, 1, 3, 1, 5, 1, 1, 1, 9, 1, 1, 1, 1, 1, 1, 1, 17, 1 ...

Accounting Method: Append II

Charge cost 3 for every append operation.

size (after append)	capacity	charge	cost	credit
1	2	3	1	2
2	2	3	1	4
3	4	3	3	4
4	4	3	1	6
5	8	3	5	4
6	8	3	1	6
7	8	3	1	8
8	8	3	1	10
9	16	3	9	4
10	16	3	1	6

Charging 3 per operation covers all “running time costs”.
$\rightarrow$ Append has constant amortized running time.

Worst-Case Running Time Array

Operation	Array
Access element by position	$O(1)$
Prepend/remove first element	$O(n)$
Append	$O(1)$ (amortized)
Remove last element	$O(1)$ (amortized)
Insert, remove from the middle	$O(n)$
Traverse all elements	$O(n)$

Linked Lists

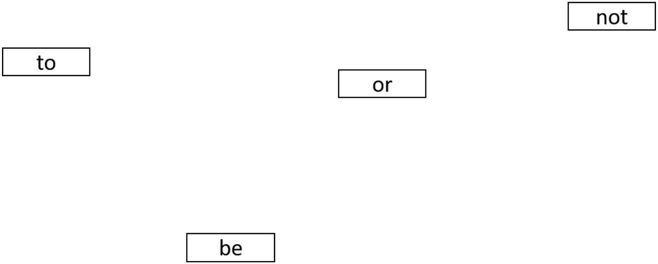
Motivation

- Arrays need a large continuous block of memory.
- Inserting elements at arbitrary positions is expensive.

Alternative that allows us to distribute the elements in memory?

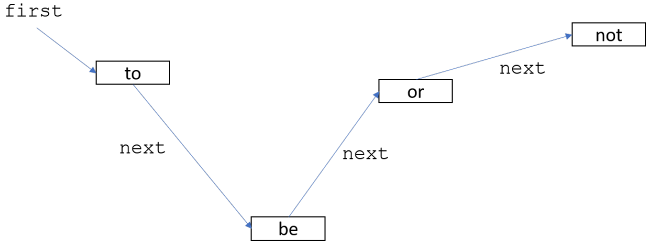
Question?

- How can we order elements that are distributed in memory?



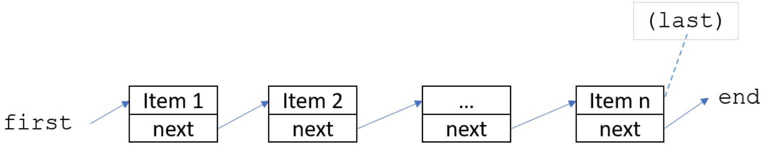
Question?

- How can we order elements that are distributed in memory?



Linked Lists

- Every node stores its entry as well as a reference (pointer) to its successor.
- Need special value for the next pointer of the last element.
- . . . or a reference to the last element.



Jupyter Notebook

Jupyter notebook: `linked_lists.ipynb`

Implementation: Node

```python
class Node:
    def __init__(self, item, next=None):
        self.item = item
        self.next = next
```

Implementation: List (without last reference)

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

    # prepend item at the front of the list
    def prepend(self, item):
        new_node = Node(item, self.first)
        self.first = new_node

    ... # other methods added to notebook after lecture
```

Worst-Case Running Time Array / Linked List

Operation	Array	Linked List
Prepend/remove first element	$O(n)$	$O(1)$
Append	$O(1)$ (amortized)	$O(n)$
Remove last element	$O(1)$ (amortized)	$O(n)$
Insert, remove from the middle	$O(n)$	$O(n)$
Traverse all elements	$O(n)$	$O(n)$
Find an element	$O(n)$	$O(n)$
Access element by position	$O(1)$	–

What running times could we improve if we also maintained a pointer to the last element of the linked list?

Worst-Case Running Time Array / Linked List

Operation	Array	Linked List
Prepend/remove first element	$O(n)$	$O(1)$
Append	$O(1)$ (amortized)	$O(n)$
Remove last element	$O(1)$ (amortized)	$O(n)$
Insert, remove from the middle	$O(n)$	$O(n)$
Traverse all elements	$O(n)$	$O(n)$
Find an element	$O(n)$	$O(n)$
Access element by position	$O(1)$	–

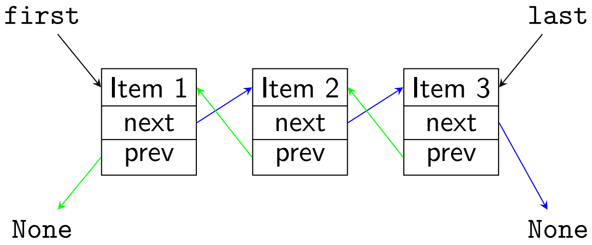
What running times could we improve if we also maintained a pointer to the last element of the linked list?

Take-home Message

- Different data structures have different trade-offs.

Doubly Linked Lists

- Idea: Do not only store a reference to the successor but also to the predecessor.
- Renders appending at/removal from end constant time.



Jupyter Notebook

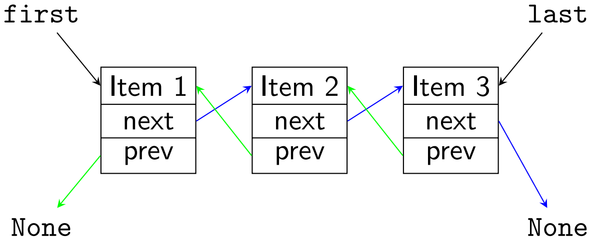
Jupyter notebook: `doubly_linked_lists.ipynb`

Doubly Linked Lists: Implementation

```python
class Node:
    def __init__(self, item, next=None):
        self.item = item
        self.next = next

class DoublyLinkedList:
    def __init__(self):
        self.first = None
        self.last = None

    def is_empty(self):
        return self.first is None

    # other methods on next slides
```

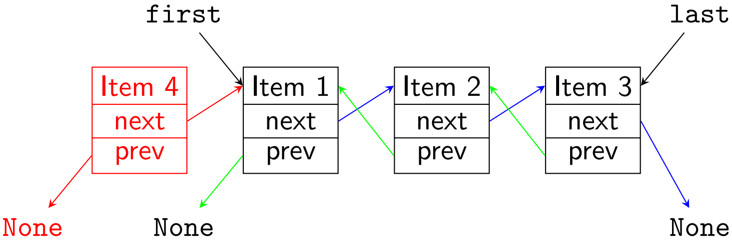
Doubly Linked Lists: prepend

```python
    def prepend(self, item):
        if self.is_empty():
            self.first = Node(item)
            self.last = self.first
        else:
            node = Node(item, self.first, None)
            self.first.prev = node
            self.first = node
```



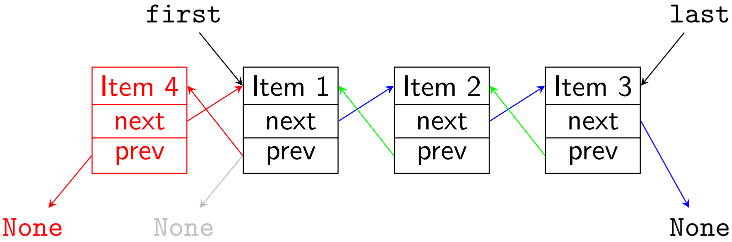
Doubly Linked Lists: prepend

```python
    def prepend(self, item):
        if self.is_empty():
            self.first = Node(item)
            self.last = self.first
        else:
            node = Node(item, self.first, None)
            self.first.prev = node
            self.first = node
```



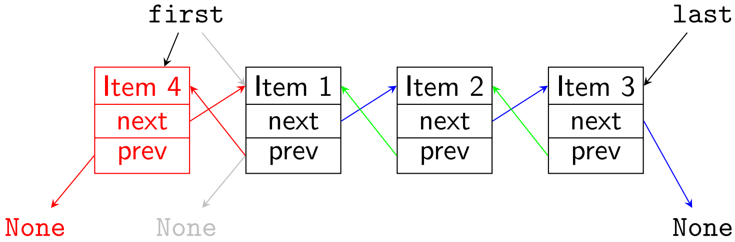
Doubly Linked Lists: prepend

```
    def prepend(self, item):
        if self.is_empty():
            self.first = Node(item)
            self.last = self.first
        else:
            node = Node(item, self.first, None)
            self.first.prev = node
            self.first = node
```



Doubly Linked Lists: prepend

```python
    def prepend(self, item):
        if self.is_empty():
            self.first = Node(item)
            self.last = self.first
        else:
            node = Node(item, self.first, None)
            self.first.prev = node
            self.first = node
```



Doubly Linked Lists: append

```
    def append(self, item):
        if self.is_empty():
            self.first = Node(item)
            self.last = self.first
        else:
            node = Node(item, None, self.last)
            self.last.next = node
            self.last = node
```

Doubly Linked Lists: remove_first

```
    def remove_first(self):
        if self.is_empty():
            raise Exception("removing from empty list")
        item = self.first.item
        self.first = self.first.next
        if self.first is not None:
            self.first.prev = None
        else:
            self.last = None
        return item
```

Doubly Linked Lists: remove_last

With doubly linked lists, removing the last element is analogous to removing the first element:

```
    def remove_last(self):
        if self.is_empty():
            raise Exception("removing from empty list")
        item = self.last.item
        self.last = self.last.prev
        if self.last is not None:
            self.last.next = None
        else:
            self.first = None
        return item
```

Worst-Case Running Time Array / Doubly Linked List

Operation	Array	Doubly Linked List
Prepend/remove first element	$O(n)$	$O(1)$
Append	$O(1)$ (amort.)	$O(1)$
Remove last element	$O(1)$ (amort.)	$O(1)$
Insert, remove in the middle	$O(n)$	$O(n)/O(1)^*$
Traverse all elements	$O(n)$	$O(n)$
Find an element	$O(n)$	$O(n)$
Access element by position	$O(1)$	–

* constant, if node at the position is parameter

Worst-Case Running Time Array / Doubly Linked List

Operation	Array	Doubly Linked List
Prepend/remove first element	$O(n)$	$O(1)$
Append	$O(1)$ (amort.)	$O(1)$
Remove last element	$O(1)$ (amort.)	$O(1)$
Insert, remove in the middle	$O(n)$	$O(n)/O(1)^*$
Traverse all elements	$O(n)$	$O(n)$
Find an element	$O(n)$	$O(n)$
Access element by position	$O(1)$	–

* constant, if node at the position is parameter

Take-home Message

Compared to singly linked lists, doubly linked lists need a linear amount of additional memory (for the `prev` references), but provide better running times for operations at the end of the list.

Summary

Summary

- An amortized analysis determines the average cost of an operation over an entire sequence of operations.
- Arrays and linked lists store sequences of items.
  - Arrays store items in a continuous space and can efficiently access an item by index.
  - Linked lists store items in nodes with a reference to the next node (doubly linked lists: also to previous node).