

# ICY Data Structures and Algorithms

&

## Fundamentals: Data Structures

Tomáš Jakl

(based on lectures by Paul Blain Levy)

Spring 2018

# Module organisation

## Assessment

80% (exam) + 20% (2 assignments)  
+ 0% (2 unassessed assignments)

## Classess

*theoretical* lecture + (more) *practical* tutorial classes

## Material

These slides on canvas + John Bullinaria's lecture notes on

<http://www.cs.bham.ac.uk/~jxb/dsa.html>

## Shared document

We have a shared document/forum for Questions and Answers (**experimental**):

<http://cs.bham.ac.uk/~txj300/dsa2018-doc>

# Programs = Algorithms + Data Structures

We use Data Structures to efficiently organise and structure collections of data in a computer.

Algorithms manipulate data structures to achieve a given goal.

Therefore, in order for a program to be fast, it has to use efficient data structures and do so efficiently.

In this module we focus on:

- various data structures
- basic algorithms
- understanding the strengths and weaknesses of those, in terms of their time and space complexities

# Abstract data types (ADT)

An *abstract data type* is

- a type
- with associated operations
- whose representation is hidden to the user

## Example

Integers are an abstract data type with operations `+`, `-`, `*`,  
`mod`, `div`, ...

- A type is a collection of values, e.g. integers, Boolean values (true and false).
- The operations on ADT might come with mathematically specified constraints, for example on the time complexity of the operations.
- Advantages of ADT's as explained by Aho, Hopcroft and Ullman (1983):

*“At first, it may seem tedious writing procedures to govern all accesses to the underlying structures. However, if we discipline ourselves to writing programs in terms of the operations for manipulating abstract data types rather than making use of particular implementations details, then we can modify programs more readily by reimplementing the operations rather than searching all programs for places where we have made accesses to the underlying data structures. This flexibility can be particularly important in large software efforts, and the reader should not judge the concept by the necessarily tiny examples found in this book.”*

## List is an ADT

An example of a list of numbers

$\langle 2, 5, 1, 8, 23, 1 \rangle$  (ordered collection of elements)

List is an ADT; list operations are:

- insert an entry (on a certain position)
- delete an entry
- access data by position
- search
- move around the list
- concatenate two lists
- sort
- ...

(we focus only on some of them)

## Different representations of lists

Depending on what operations are needed for our application, we choose from different data structures (some implement certain operations faster than the others):

- Arrays.
- Linked lists.
- Dynamic arrays.
- Unrolled linked lists.

You've already met the first two in the Software Workshop.

# Arrays

---



## (Simplified) memory model

To demonstrate how memory management in an Operating System (OS) works we are going to treat memory as a gigantic array

Mem[-]

(for simplicity we assume that every entry can contain either an integer or a string).

Two operations provided by the OS:

- `allocate_memory(n)` – the OS finds a continuous segment of `n` unused locations, designates that memory as used, and returns the address of the first location.
- `free_memory(address)` – the OS designates the memory as free again.

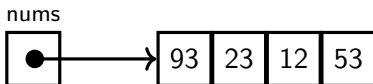
This is a simplification from what is in Java, C and other languages. In reality, strings, for example, are stored as blocks of items in `Mem` (i.e. they occupy more than just one location).

## List as an array of a fixed length

```
1 int [] nums = new int [4];
```

in Java is (roughly) translated as

```
1 nums = allocate_memory(4*1);
```



```
1 x = nums[3]; // 53  
2 nums[3] = 4;
```

becomes

```
1 x = Mem[nums+3];  
2 Mem[nums+3] = 4;
```

i	Mem[i]
⋮	⋮
3321	"Alice"
3322	"Bob"
3323	93
3324	23
3325	12
3326	53
3327	333
⋮	⋮

} array  
} **nums**

- The variable `nums` stores the address where the array starts in `Mem`; in our case it is equal to 3323.
- Every `int` in the array occupies one location in `Mem`. In the next slide we show an example of where every item in the array occupies more than just one location in `Mem`.
- Notice that the value of `x` is 53. This is because we the indexing arrays starts from 0, i.e. the indexes of elements in `nums` are 0, 1, 2, 3.

## More complicated arrays in Mem[-]

```
1 class Student {  
2     String name;  
3     int id;  
4 }  
5 Student[] studs = new Student[3];
```

becomes

```
1 studs = allocate_memory(3*2);
```

---

i	Mem[i]
:	:
4029	"Sarah"
4030	1419231
4031	"Berry"
4032	2113812
4033	"Gale"
4034	1322813
:	:

```
1 studs[1].name = "John";  
2 studs[1].id = 1419231;
```

becomes

```
1 Mem[studs+2*1+0] = "John";  
2 Mem[studs+2*1+1] = 1419231;
```

- To store one `Student` we need to allocate two locations in `Mem`.
- Note that our interpretation of `new Student[3]` is not exactly as in Java. In Java, this code creates an array of three pointers (i.e. addresses) and each of the pointers points to a newly allocated `Student` object.

# Memory management

In Java

- allocations automatic
- freeing memory is automatic (by the garbage collector)
- bounds of arrays are checked

In C or C++

- allocations explicit
- freeing memory explicit
- bounds not checked

Java is slower and safe, C (or C++) is fast and dangerous.

A very common mistake is to forget to subtract 1:

```
int[] a = new int[5];  
a[5] = 1000; // Kaboom!
```

This leads to an `ArrayIndexOutOfBoundsException` in Java whereas in C (or C++) this goes through without a warning and can lead to a corruption of data in memory!



## Inserting by shifting

To insert a student at position `pos`, where  $0 \leq pos \leq size$ :

```
1 Students[] studs = new Students[maxsize];
2 int size = 0;    // number of students stored
3
4 void insert(int pos, String name, int id) {
5     if (size == maxsize)
6         throw ArrayFullException;
7
8     for (int i=size; i > pos; i--) {
9         // Copy entry in position i-1 to the right
10        Mem[studs + 2*i      ] = Mem[studs + 2*(i-1)      ];
11        Mem[studs + 2*i + 1] = Mem[studs + 2*(i-1) + 1];
12    }
13
14    Mem[studs + 2*pos] = name;
15    Mem[studs + 2*pos + 1] = id;
16    size++;
17 }
```

If we want to insert a value to an array (at a certain position) we can do this in two steps:

1. Create a new array, of size bigger by one.
2. Copy elements of the old array to the new one to the corresponding positions.

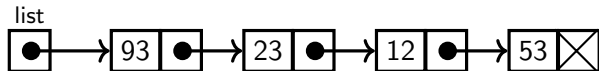
However, this requires to copy the whole array every single time. Instead, we can allocate a big array at the beginning (of size `maxsize` ) and then always “only” shift elements whenever we are inserting/deleting one.

## Linked Lists

---

## Linked Lists in Mem[-]

Linked list representing a list  $\langle 93, 23, 12, 53 \rangle$ :



Inserting at the beginning of a list:

```
1 void insert_beg(int number) {
2     newblock = allocate_memory(2);
3     Mem[newblock] = number;
4     Mem[newblock+1] = list;
5     list = newblock;
6 }
```

Convince yourself that it works, even if `list == END`.


What is the complexity of `insert_beg`?

Does it depend on the size of the list?


`list = 3823`

and

i	Mem[i]
⋮	⋮
0324	23
0325	3111
⋮	⋮
3111	12
3112	7479
⋮	⋮
3823	93
3823	0324
⋮	⋮
7479	53
7480	END
⋮	⋮

Similarly to what we had before, each  is realised as a block of two consecutive locations in `Mem`. The first location of such block stores a number and the second location stores the address of the following block.

The variable `list` stores the address of the first such block.

`END` indicates the end of the list (graphically as ). Its value can be anything that is not a valid address, for example, `-1`.

A linked list is empty whenever `list` is equal to `END`.

An advantage of linked lists over arrays is that the length of linked lists is not fixed. We can insert and delete items as we want. On the other hand, accessing an entry on a specific position requires traversing the list.

To represent the list  $\langle 93, 23, 12, 53 \rangle$  as a linked list do the following:

```
1 // allocating memory for the boxes
2 block1 = allocate_memory(2);
3 Mem[block1] = 93;
4
5 block2 = allocate_memory(2);
6 Mem[block2] = 23;
7
8 block3 = allocate_memory(2);
9 Mem[block3] = 12;
10
11 block4 = allocate_memory(2);
12 Mem[block3] = 53;
13
14 // linking the boxes together
15 list = block1;
16 Mem[block1+1] = block2;
17 Mem[block2+1] = block3;
18 Mem[block3+1] = block4;
19 Mem[block4+1] = END;
```

Alternatively, use the function `insert_beg` :

```
1 list = END;
2
3 insert_beg(53);
4 insert_beg(12);
5 insert_beg(23);
6 insert_beg(93);
```

## More operations on Linked Lists

```
1 void delete_beg() {
2     // is empty?
3     if (list == END) {
4         throw
5             EmptyListException;
6     }
7
8     firstnode = list;
9     list = Mem[firstnode+1];
10    free_memory(firstnode);
11 }
```

```
1 Boolean is_empty() {
2     return list == END;
3 }
```

```
1 int value_at(int index) {
2     x = list;    // pointer
3
4     while (index > 0) {
5         if (Mem[x+1] == END)
6             throw
7                 OutOfBoundsException;
8
9         x = Mem[x+1];
10        index--;
11    }
12
13    return Mem[x];
14 }
```

What is the time complexity of those operations?

How would you implement `insert_end` and `delete_end`?

## Insert at the end

```
1 void insert_end(int number) {
2     newblock = allocate_memory(2);
3     Mem[newblock] = number;
4     Mem[newblock+1] = END;
5
6     if (list == END) {
7         list = newblock;
8     } else {
9         pointer = list;
10
11         while (Mem[pointer+1] != END)
12             pointer = Mem[pointer+1];
13
14         Mem[pointer+1] = newblock;
15     }
16 }
```



## The same by recursion

```
1 void insert_end(int number) {
2     list = insert_help(list, number);
3 }
4
5 // just a helper function:
6 int insert_help(int address, int number) {
7     if (address == END) {
8         newblock = allocate_memory(2);
9         Mem[newblock] = number;
10        Mem[newblock+1] = END;
11        return newblock;
12    } else {
13        Mem[address+1] = insert_help(Mem[address+1], number);
14        return address;
15    }
16 }
```

## Comparison

If we store a list of  $n$  elements as an array or a linked list, how efficient are going to be the basic operations of lists (seen as an ADT)?

	Array	Linked List
access data by position		
search		
insert an entry at the beginning		
insert an entry at the end		
insert an entry (on a certain position)		
delete an entry		
concatenate two lists		

*Search* is a procedure which finds the position (counting from 0) where a value given on input is stored in the array or linked list.

We compare the efficiencies by how many elements of the list do we have to traverse (in the worst case) in order to finish the operation.

In practise, we choose between using an array or linked list depending on which operations we need the most often.

## Comparison (solution)

If we store a list of  $n$  elements as an array or a linked list, how efficient are going to be the basic operations of lists (seen as an ADT)?

	Array	Linked List
access data by position	constant	linear
search	linear	linear
insert an entry at the beginning	linear	constant
insert an entry at the end	linear*	linear*
insert an entry (on a certain position)	linear	linear
delete an entry	the same	the same
concatenate two lists	linear	linear*

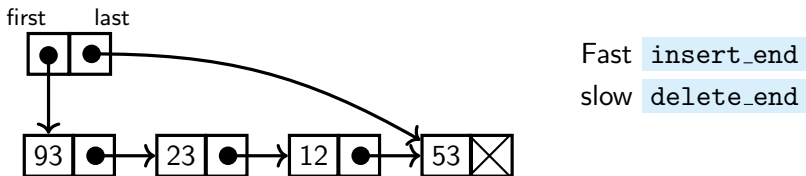
By “how efficient” we mean how many items in the list we have to traverse?

*Constant* means that the number of operations does not depend on the number of elements in the list. *Linear* means that we might, in the worst case, traverse the whole list.

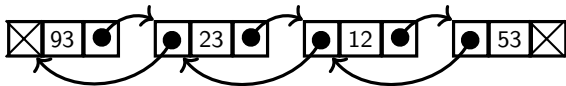
The stars indicate that it could be improved to the constant time if we use a slightly different representation. For example, inserting at the end of an array in constant time is possible if we implement `insert` ing as on slide 10.

## Modifications

Linked list with a pointer to the last node:



Doubly linked list:



Try yourself: Write `insert_beg`, `insert_end`, `delete_beg` and `delete_end` for those two representations.

If we remember where the first and the last block of a doubly linked list is stored, then inserting at the end and deleting from the end could be implemented in constant time.

## Time for a quiz show!

Let `nums` be the address of an array of integers of length `len`.



Which of the following algorithms creates a linked list faster?

```
1 int list = END;
2
3 for (int i=0; i < len; i++)
4   insert_end(list, Mem[nums+i]);
```

```
1 int list = END;
2
3 for (int i=len-1; i >= 0; i--)
4   insert_beg(list, Mem[nums+i]);
```



If we do not keep track of the last node (as was the case on slide 17) then the second algorithm is faster. This is because every `insert_beg` takes constant time to execute whereas when running `insert_end` we need to traverse 1, 2, 3, ..., `len`-many elements, that is, in total we traverse

$$1 + 2 + 3 + \dots + n = \frac{n(n+1)}{2}$$

many elements.