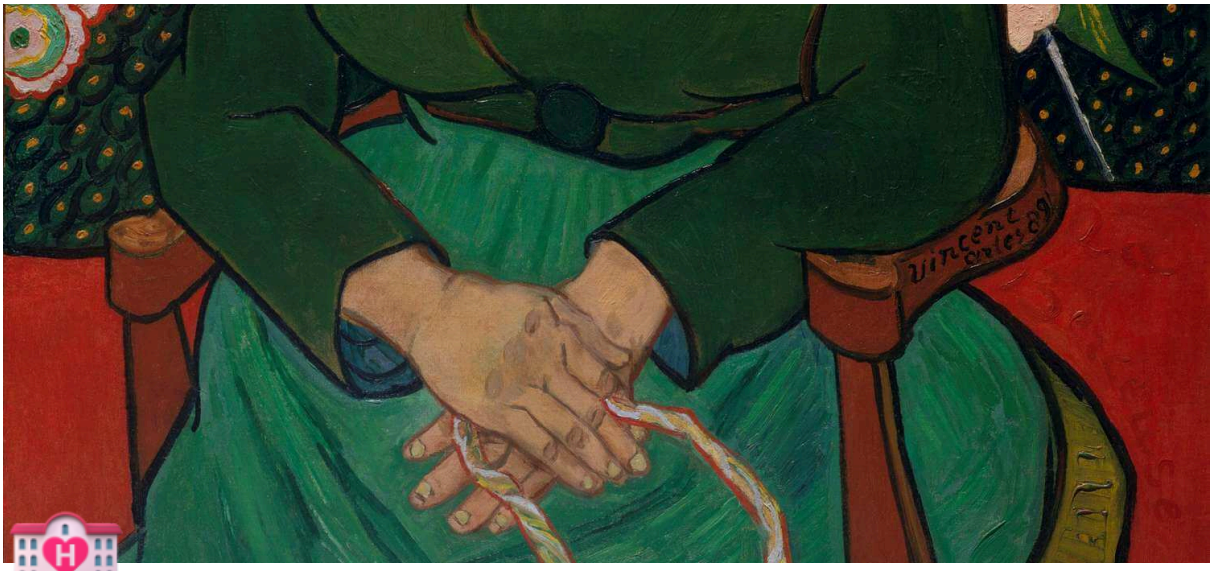


Computer memory

Computer memory:

Computer Memory to a Program

Data is stored, managed and manipulated in computer memory.

- The computer architecture (the hardware) and the operating system work together to allow each running program to see an illusion that they have all the memory on the computer to themselves.
- This memory is organised as a long list of memory cells, each 1 byte = 8 bits large. A bit can hold either a zero or a one. Every one of these 1 byte cells has an address, a number in the range 0 to the highest possible address for this system combination
- On 64 bit systems, this highest address is 0xFFFFFFFFFFFFFFFF (i.e., 16 'F's), or $9,223,372,036,854,775,807$ (i.e., $2^{64}-1$).
- For technical reasons, the hardware usually manipulates memory a whole word at a time, where is word is either 4 bytes (32 bits) or 8 bytes (64 bits)

Memory Management

Of course, computers can not hold such gigantic amounts of memory, and if a program tried to access every one of those bytes of memory, it would reach addresses in memory where the illusion breaks down, and an error would be triggered.

To support the illusion of all this memory for a program, programs have to explicitly request the operating system to add large sections of memory to their memory space via *operating system calls*

However, programs themselves need to manage their own memory with much finer and more efficient control than these expensive operating system calls

So the *Runtime System*, a software library that is integral to each programming language, typically provides one of two options

1. Explicit Memory Management, with *allocate* and *free* or
2. Implicit Memory Management with Garbage Collection

2

- ***Memory Management:***
  - *Programs request large sections of memory from the operating system (OS) via system calls to manage memory efficiently.*
- ***Purpose:***
  - *Avoids errors and maintains the illusion of continuous memory availability.*
- ***Memory Requests:***
  - *Programs explicitly request large chunks of memory from the OS instead of tiny bits, similar to ordering inventory in bulk.*
- ***Repeated Requests:***
  - *When the initial memory chunk is used up, the program makes another request for more memory from the OS.*
- ***Efficient Distribution:***
  - *By requesting chunks, memory is distributed evenly among programs, preventing any single program from hogging all resources.*

Explicit Memory Management:

Explicit Memory Management

The programming languages C and C++ are examples of programming languages with explicit memory management.

C's runtime system maintains it's own data structure to organise the memory it has obtained from the operating system. This data structure is usually accessed by two functions:

1. `malloc`: allows the program to request a contiguous amount of memory. If available, this is recorded in the memory managment data structure, the address of the start of the block is returned to the program and the data structure is updated accordingly.

   If not available, a system call requests more memory from the operating system. If successful, it integrates the memory into its data structure and continues as before. If unsuccessful, it reports an error back to the program.

2. `free`: Given the address of a block that was previously allocated, marks it as available for future allocation

3

Explicit Memory Management

Explicit Memory Management is a simple model and very efficient. However, it suffers from a number of disadvantages:

- The program needs to keep track of every block requested with `malloc` so that it can be freed later. Otherwise, the program will have memory leaks — these are blocks of allocated memory that can no longer be used or freed. This causes the program to use more memory than it needs (which causes performance problems), and may cause the program to crash if it eventually runs out of memory.
- If an error is made in giving an address to `free` that had not previously been returned by `malloc`, or if that address had already been freed, then the memory management data structure can be corrupted, causing unpredictable errors or crashes

4

Implicit Memory Management:

Implicit Memory Management

Implicit Memory Management is much more sophisticated. The programming language itself, through its runtime system, provides mechanisms to allocate data structures (without exposing memory addresses to programmers), and identifies allocated structures that can no longer be accessed by the running program and adds them back into its data structure of available free memory without requiring the program to specifically ask for the memory to be freed.

This mechanism is used by Java, Python, Functional programming languages like Haskell and OCaml, and many more.

It relieves the heavy burden of keeping track of allocated memory from the programmer, and avoids most of the bugs and problems that address manipulation in languages like C are famous for.

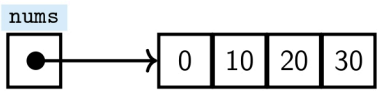
5

Represenatation of arrays in memory

This java code allocates 4 * 4 bytes of memory, and assign values.

```
1  int[] nums = new int[4]
2  for (int i=0, i<nums.length; i++)
3      nums[i] = i * 10;
```

This can be described in a diagram such as the one below, and results in the memory layout shown to the right



Address	Memory content
⋮	⋮
3344	23
3340	30271
3336	30
3332	20
3328	10
3324	0
3320	6738
⋮	⋮
3100	3324 ← nums

array

Here we assume that the word size is 32 bits or 4 bytes, and integers (and memory pointers) are also 32 bits.

6

- `nums` is a variable whose cell in memory is at address 3100. The contents of this cell is a memory address, 3324, which is where the array starts
- Every `int` in the array occupies one word or 4 bytes in memory
- Because we declared our array to be an array of integers, the compiler knows that every entry in the array takes 4 bytes, and if the array starts at location 3324, then it knows that:
  - `nums[0]` is at address 3324,
  - `nums[1]` is at address $3324 + (1 \times 4)$,
  - `nums[2]` is at address $3324 + (2 \times 4)$, etc.

More complicated arrays in memory

In its simplist form, a Java class collects variables together into a single structure

```
1  class Point {
2      float x;
3      float y;
4  }
5  Point[] locations = new Point[3];
6  locations[1].x = 25.2;
7  locations[1].y = 38.6;
```

Addr	Memory
⋮	⋮
4052	0.0
4048	0.0
4044	38.6
4040	25.2
4036	0.0
4032	0.0
⋮	⋮
3100	4032← `locations`

Given that a float is 4 bytes, the following is what happens in memory:

- cell at address `locations` $+ (1 * 2 * 4) + (0 * 4)$ is set to 25.2, and
- cell at address `locations` $+ (1 * 2 * 4) + (1 * 4)$ is set to 38.6.

In the $1 * 2 * 4$: the 1 is the index into locations, the 2 is the number of words in a Point object, and the 4 is the size of the word.

7

A very common mistake is to try to access the last cell in an array incorrectly:

```
int[] a = new int[5];
a[5] = 1000; // Error: the cells are a[0] to a[4]
```

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!

Abstract Data Types (ADT):

Abstract Data Types (ADT)

A type is

- a set of possible **values**
- with a set of allowed **operations** on those values

An abstract data type (ADT) is a type whose internal representation is hidden to the user.

Thus users of an abstract data type may have no information about how the ADT is implemented, but depends only on the published information about how it behaves. This means that the implementation of an abstract type can be changed without having to change the code that uses it.

Example. Integer is an abstract data type consisting of integer values with operations `+`, `-`, `*`, `mod`, `div`, ...

8

- A type is a collection of values, e.g. integers, Boolean values (true and false) with their operations.
- The operations on an 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."*

Example of an ADT:

Lists:

List as an ADT

A list is an ordered collection of elements (values). An example of a list of numbers:

$$\langle 2, 5, 17, 1, 8, 7, 23, 1 \rangle$$

List is an ADT; list operations may include:

- insert an entry (on a certain position)
- delete an entry
- access data by position, e.g., `list[i]`
- search an element in the list
- concatenate two lists
- sort
- ...

9

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
- ...

We will study some of these in detail later.

10

Lists in Java

List is an ADT. A specific instance of a list, e.g. a List of integers, would be specified in Java by `List<int>`, a List of Strings as `List<String>` etc.

There are different implementations of the List ADT in the Java library, for example an Array based List (`ArrayList<int>`, `ArrayList<String>`, ...) and a Linked List (`LinkedList<int>`, `LinkedList<String>`, ...)

In Java we can declare and allocate a List, specifying which implemention we want, with the code:

```
1     List<int> myArrayList = new ArrayList<int>();
2     List<int> myLinkedList = new LinkedList<int>();
```

From this point on, you can use any of the predefined List methods on the `myArrayList` or `myLinkedList` variables

11

List Abstract Data Type

Let us fix some operations for a List abstract data type:

- Constructors:
  - `EmptyList`: returns an empty List
  - `MakeList(element, list)`: adds an element at the front of a list.
- Accessors
  - `first(list)`: returns the first element of the list[1]
  - `rest(list)`: returns the list excluding the first element[1]
  - `isEmpty(list)`: reports whether the list is empty

From these, all other operations (e.g., find the *n*th element of the list, append one list onto another) can be implemented without requiring any other access to the List implementation details.

[1] Triggers error if the list is empty

12

List Operations: last element

The following would get the *last* element of a list lst:

```
type last(List<int> lst) {
  if (lst.isEmpty()) {
    throw new IllegalStateException;
  } else if (rest(lst).isEmpty()) {
    return first(lst);
  } else {
    return last(rest(lst));
  }
}
```

13

List Operations: getElementByIndex

The following code gets the element at position *i*:

```
type getElementByIndex(int i, List<int> lst) {
  if (i < 0 or lst.isEmpty()) {
    throw new IndexOutOfBoundsException;        // i >= lst.length
  } else if (i == 0) {
    return first(lst);
  } else {
    return getElementByIndex(i-1, rest(lst));
  }
}
```

14

Basically removes each element upto that element then spits out that element

List Operations: append

The following code appends `lst2` at the end of `lst1`:

```
List<int> append(List<int> lst1, List<int> lst2) {
  if (lst1.isEmpty())
    return lst2;        // <> + <ys..> = <ys..>
  else
    return MakeList(first(lst1), append(rest(lst1), lst2))
                        // <x, xs..> + <ys..> = <x> + <xs.., ys..>
}
```

15

Arrays as lists:

Arrays as abstract lists

The basic array data type has a few operations:

```
int[] nums = new int[4];        // array creation
value = nums[i];                // get value by index i
nums[i] = 23;                   // replace value at index i
len = nums.length;              // get length of the array
```

More sophisticated List ADTs often use basic arrays in their implementation, but add more complex operations such as increasing the size of a List, Sorting a list, concatenating Lists etc.

16

Inserting into an Array by Shifting Up

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

```
1  MAXSIZE = 128           // maximal size of the array
2  Point[] locations = new Point[MAXSIZE];
3  int size = 0;           // number of points currently stored
4
5  void insert(int pos, Point pt) {
6    if (size == MAXSIZE) {
7        throw new ArrayFullException;
8    }
9    for (int i = size-1; i >= pos; i--) {
10     // shift the entry in position i by 1 position towards the end
11     locations[i+1] = locations[i];
12   }
13   locations[pos] = pt;
14   size++;
15 }
```

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.

Exercise: write the corresponding pseudocode to remove an item from an array by shifting down

Dynamic array: First attempt

Naive approach:

1. initially allocate an array of 128 entries
2. whenever the array becomes full, increase its size by 128

To insert n entries, starting from empty, how long does it take?

For simplicity assume that $n = 128 + 128k$ for some k.

$$
\begin{aligned}
&128 \text{ insertions} + \\
128 \text{ copies} \; &+ 128 \text{ insertions} + \\
256 \text{ copies} \; &+ 128 \text{ insertions} + \\
384 \text{ copies} \; &+ 128 \text{ insertions} + \\
512 \text{ copies} \; &+ 128 \text{ insertions} + \\
&\dots
\end{aligned}
$$

- insertions: $128 + 128k = n$
- copies:

$$128 \times (1 + 2 + \cdots + k) = 64k(k+1)$$

- **total:** $128(k+1) + 64k(k+1) = O(n^2)$

18

$$128, 128 + 128, (128 * 2) + 128, (128 * 3) + 128$$

$$\therefore \text{sum to k} =$$

$$\sum_{i=1}^{k+1} 128 + \sum_{i=1}^{k} 128k$$

$$128(k+1) + 128 \sum_{i=1}^{k} k$$

$$= 128(k+1) + \tfrac{128}{2}(k+1) = 128(k+1) + 64k(k+1) = O(n^2)$$

At the beginning we have

```
MAXSIZE = 128;
arr = new int[MAXSIZE];
stored = 0;
```

We add elements to it by storing them at the end and increasing `stored` by one. Then, anytime `stored == MAXSIZE` (i.e. `arr` becomes full), we have to allocate a new array of size `MAXSIZE = MAXSIZE + 128` and copy all elements from `arr` into it.

Recall that $1 + 2 + \ldots + n = \frac{n(n+1)}{2}$ therefore

$$1 + 2 + \ldots + (k-1) = \frac{(k-1)k}{2}$$

The following analysis, however, does not depend on the exact choice of the parameters. If we started with an array of length 10 and increased its size by 5 every time it becomes full, for example, the resulting amortized complexity would still be the same.

Dynamic array: Amortised complexity of the first attemp

Recall $n = 128 + 128k$.

- Insertions: $128(k+1)$
- Copies : $64k(k+1)$
- **Total:** $64k(k+1) + 128(k+1)$

Amortized cost of one insertion:

$$\frac{64k(k+1) + 128(k+1)}{128(k+1)} = \frac{1}{2}k + 1 = O(n)$$

$\implies$ the amortized complexity of insertion is $O(n)$.

19

Wanna rewrite the amortized cost of one insertion calculation just ehh because:

$$\frac{64k(k+1)+128(k+1)}{128(k+1)} = \frac{1}{2}k + 1 = O(n)$$

The amortized cost is computed as the average number of operations needed for one insertion. In our case:

$$\frac{\text{number of copying and inserting}}{\text{number of inserting}}$$

We see that copying is the problem. In the following smarter approach we try to suggest a different strategy which makes sure that copying happens less often.

Dynamic array: Better approach (= Java's ArrayList)

Smart approach:

1. initially allocate an array of 128 entries
2. whenever the array becomes full, double its size

To insert n entries, starting from empty, how long does it take? For simplicity assume that $n = 128 \times 2^k$ (for some k).

$$\begin{aligned} & 128 \text{ insertions} + \\ 128 \text{ copies} \; & + 128 \text{ insertions} + \\ 256 \text{ copies} \; & + 256 \text{ insertions} + \\ 512 \text{ copies} \; & + 512 \text{ insertions} + \\ 1024 \text{ copies} \; & + 1024 \text{ insertions} + \\ & \dots \end{aligned}$$

- insertions:

$$128 + 128 \times (1 + 2 + 4 + \cdots + 2^{k-1}) = 128 \times 2^k = n$$

- copies:

$$128 \times (1 + 2 + 4 + \cdots + 2^{k-1}) = n - 128$$

- **total:** $n + n - 128 \leq 2n$

20

Insertion Cost Analysis

We analyze how long it takes to insert **n** elements when the array resizes dynamically.

$\text{Assume } n = 128 * 2^k \text{ for some k}$

Each time the array expands, we:

- Copy all existing elements into the new array.
- Perform the new insertions.

Step-by-step breakdown:

- Start with **128 insertions**.
- Resize to **256** → Copy **128** old elements, insert **128** new ones.
- Resize to **512** → Copy **256**, insert **256**.
- Resize to **1024** → Copy **512**, insert **512**.
- Continue until reaching n.

  nn

Total Insertions

$128 + 128 \times (1 + 2 + 4 + \cdots + 2^{k-1})$

Since the sum of a geometric series is:

$$S_n = \frac{a(1 - r^n)}{(1 - r)}$$

$1 + 2 + 4 + \cdots + 2^{k-1} = 2^k - 1$

because:

$a = 1, r = 2, n = k - 1$

$\frac{1(1-2^k)}{1-2} = -(1 - 2^k) = -1 + 2^k = 2^k - 1$

We get:

$128 + 128 \times (2^k - 1) = 128 \times 2^k = n.$

Total Copies

Each resize step involves copying all existing elements:

$$128 \times (1 + 2 + 4 + \cdots + 2^{k-1})$$

Using the same geometric series formula:

$$128 \times (2^k - 1) = n - 128.$$

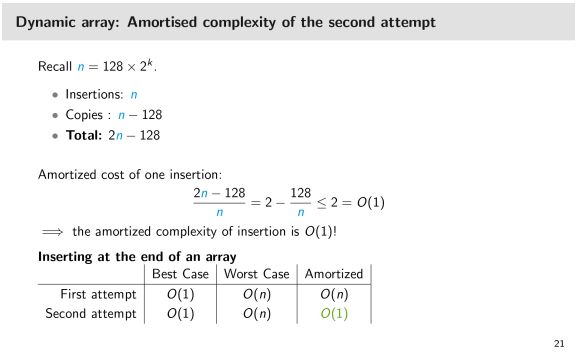
Final Total Cost

$$\text{Total operations} = n + (n - 128) = 2n - 128.$$

Since $2n - 128 \leq 2n$, the overall complexity is **O(n)**.

Conclusion

Even though doubling the array causes extra copying, the total number of operations is at most **2n**, making dynamic array insertions **amortized O(1) per operation**.



Dynamic array: Amortised complexity of the second attempt

Recall $n = 128 \times 2^k$.

- Insertions: n
- Copies : $n - 128$
- **Total:** $2n - 128$

Amortized cost of one insertion:

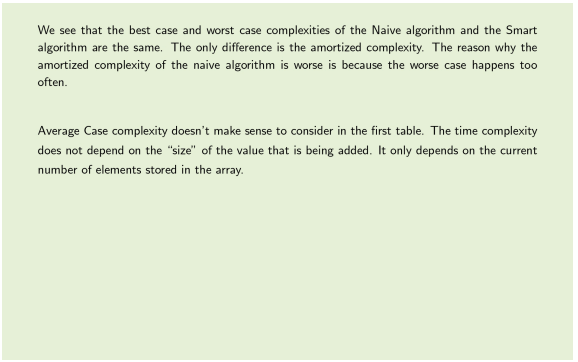
$$\frac{2n - 128}{n} = 2 - \frac{128}{n} \leq 2 = O(1)$$

$\implies$ the amortized complexity of insertion is $O(1)$!

Inserting at the end of an array

	Best Case	Worst Case	Amortized
First attempt	$O(1)$	$O(n)$	$O(n)$
Second attempt	$O(1)$	$O(n)$	$O(1)$

21



We see that the best case and worst case complexities of the Naive algorithm and the Smart algorithm are the same. The only difference is the amortized complexity. The reason why the amortized complexity of the naive algorithm is worse is because the worse case happens too often.

Average Case complexity doesn't make sense to consider in the first table. The time complexity does not depend on the "size" of the value that is being added. It only depends on the current number of elements stored in the array.

Amortized complexity

Linked Lists:

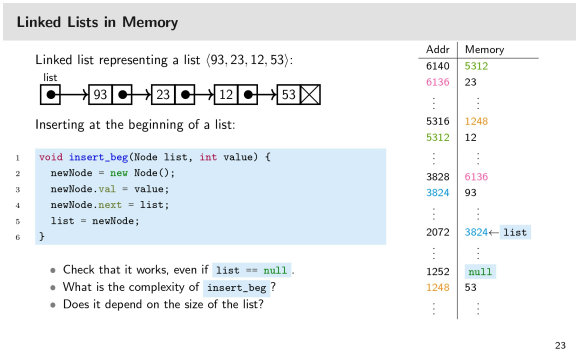
Linked Lists in Memory

Linked lists hold values of a particular type. They are constructed from structures (Class objects in Java), called *nodes* that have a `value` variable to hold the value and one or more `node` variables to identify the next node in the list.

The linked list is then a collection of Node structures each connected to others in a chain.

```
1  class Node {
2    int val;
3    Node next;
4  }
5  Node list = null;
```

Note that no Node has yet been allocated and therefore the list is empty. `null` is a special value that represents an impossible memory address. Any attempt to access `Node.val` or `Node.next` when `list` has the value `null`, would immediately cause an error.



Linked Lists in Memory

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

list → 93 → 23 → 12 → 53 → ⊠

Inserting at the beginning of a list:

```
1  void insert_beg(Node list, int value) {
2    newNode = new Node();
3    newNode.val = value;
4    newNode.next = list;
5    list = newNode;
6  }
```

- Check that it works, even if `list == null`.
- What is the complexity of `insert_beg`?
- Does it depend on the size of the list?

Addr	Memory
6140	5312
6136	23
⋮	⋮
5316	1248
5312	12
⋮	⋮
3828	6136
3824	93
⋮	⋮
2072	3824← list
⋮	⋮
1252	null
1248	53
⋮	⋮

23

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

The `list` variable contains the address pointing to the first node, called the *head pointer*.

`null` indicates the end of the list (graphically as ⊠). Its value can be anything that is not a valid address, for example, `-1`. Most languages use `0` for this purpose, which, although theoretically is a valid address, is never so in practice. Java uses a special value called `null`.

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

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.

A linked list is not a list of lists. It's a list with the elements not necessarily one after each other. You have a number and then a pointer to the location of the next element in the list

Deleting at the beginning

```
bool is_empty(Node list) {
  return (list == null);
}
```

```
void delete_begin(Node list) {
  if is_empty(list) {
    throw new EmptyListException;
  }
  list = list.next;
}
```

24

Lookup

```
int value_at(Node list, int index) {
  int i = 0;
  Node nextnode = list;
  while (true) {
    if (nextnode == null) {
      throw new OutOfBoundsException();
    }
    if (i == index) {   // check to break the loop
      break;
    }
    nextnode = nextnode.next;
    i++;
  }
  return nextnode.val;
}
```

25

Search is a procedure which finds the position (counting from 0) where a value, given as a parameter, is stored in the array or linked list.

By "cost as a function of the number of elements" we mean: how does the number of items we have to inspect or modify grow as the number of elements in the structure grows? *Constant* means that the number of operations does not depend on the number of elements. *Linear* means that if we increase the number of elements in the structure by a factor of *n*, then the cost of the operation in the worst case will be multiplied by *n* too.

We compare costs by comparing the number of elements of the list we have to inspect (in the worst case) in order to finish the operation.

In practise, we choose between using an array or linked list depending on both relative frequency of use of the operations and their relative costs.

What is the time complexity of these operations?

(How does this compare to arrays?)

How would you implement insert_end and delete_end?

Insert at the end

```
1  void insert_end(Node list, int value) {
2    newblock = new Node();
3    newblock.val = value;
4    newblock.next = null;
5    if (list == null) {
6      list = newblock;
7    }
8    else
9    {
10     cursor = list;
11     while (cursor.next != null){
12       cursor = cursor.next;
13     }
14     cursor.next = newblock;
15   }
16 }
```

26

Comparison (solution)

If we store a list of `n` elements as an array (without spare space) or a linked list, what costs will the basic operations of lists have as a function of the number of elements in the list (when the list is seen as an ADT)?

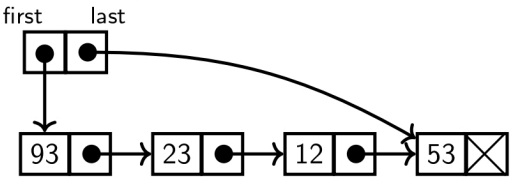
	Array	Linked List
access data by position	$O(1)$	$O(n)$
search for an element	$O(n)$	$O(n)$
insert an entry at the beginning	$O(n)$	$O(1)$
insert an entry at the end	$O(n)$	$O(n)$*
insert an entry (on a certain position)	$O(n)$	$O(n)$
delete first entry	$O(n)$	$O(1)$
delete `i` th entry	$O(n)$	$O(n)$
concatenate two lists	$O(n)$	$O(n)$*

* The stars indicate that it could be improved to constant time if we modified the representation slightly. As we'll see very soon...

28

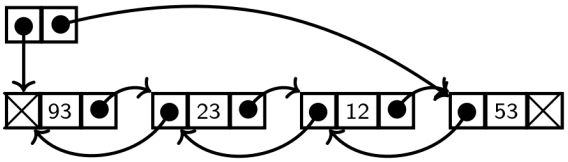
Modifications

Linked list with a pointer to the last node:



Fast `insert_end`
Slow `delete_end`

Doubly linked list:



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

29

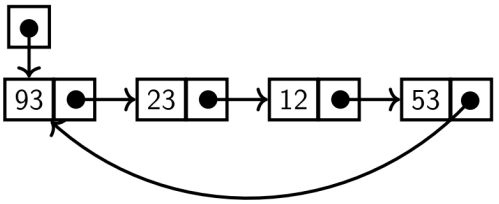
If we remember where the first and the last block of a doubly linked list is stored, then inserting at the end and deleting from the end could be implemented in constant time.

- For a singly linked list node we use:
  - `a.val` for the value contained node `a`
  - `a.next` for the (pointer to the) next node in the list
- For a doubly linked list node we use:
  - `a.prev` for the (pointer to the) previous node in the list
  - `a.val` for the value contained node `a`
  - `a.next` for the (pointer to the) next node in the list

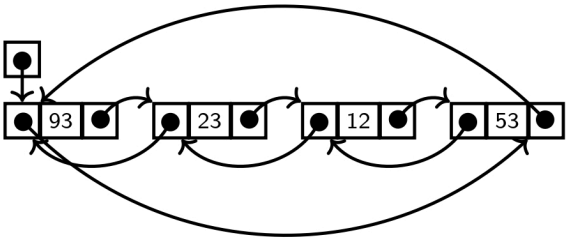
Now we know what node we're at what node is in front of us and what node is behind us

More Modifications

Circular Singly Linked List:



Circular Doubly Linked List:

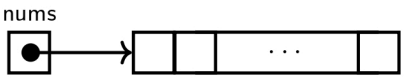


Try yourself: Write `insert_beg`, `insert_end`, `delete_beg` and `delete_end` for those two representations.

30

Time for a quiz!

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



Which of the following algorithms creates a **Singly Linked List** faster?

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

```
1  list = null;
2  for (int i=len-1; i >= 0; i--) {
3    insert_beg(list, nums[i]);
4  }
```

31

If we do not keep track of the last node then the second algorithm is faster. This is because every `insert_beg` takes constant time to execute whereas when running `insert_end` we need, every time we insert, to traverse the list to the end before actually adding the new element, and each time the list grows by one element so in total we have to traverse first a list of 0 element, then a list of 1 elements,..., until finally we traverse a list of `len` -1 elements. That is, in total we do $0+1+2+3+\ldots($ `len` $-1)$, traversal steps, that is:

$$0+1+2+3+\cdots+(\texttt{len}-1)=\frac{\texttt{len}\,(\texttt{len}-1)}{2}$$