CMPUT204: Introduction to Algorithms

Agenda:

- Algorithms concepts (Ch. 1.1)
- Pseudo-code (p. 20-22 — but somewhat different from our conventions)
- RAM model & Arrays
- Recursions
- Linked Lists (Ch. 10.2)

Theory Courses @ UofA

- Welcome to Your First Theory Course!

- 204 **Algorithm I**
  - Introduction to algorithms
  - Data-Structures
  - Basic algorithm design and analysis principles
- 304 **Algorithms II**
  - More advanced algorithms, and their design and analysis
  - Notion of reduction between problems, NP and NP-completeness
- 474 **Formal Languages, Automata and Computability**
  - More formal approach to models, complexity, and computability
- Grad courses, seminars, research papers...

So… What's an Algorithm?

- Definition #1: Problem: Given an input X satisfying… output Y satisfying…
- Definition #2: Instance: A specific input for a problem is an *instance*
- Definition #3: Algorithm: A well-defined step-by-step procedure that is guaranteed to solve the problem: to take any X and output the correct Y
- Examples (that you already know!):
  - Given $X = \langle n_1, n_2 \rangle$, output $Y = n_1 \times n_2$. (Example of an instance: 7×28)
  - Given a natural number X, output $Y = X!$ (Example of an instance $444!$)
  - Given flour, coco, sugar and…, output a chocolate cake
    (Feel free to bring an instance to class!)
- The difference: such algorithms were just given to you, and you only needed to implement them.
  In this course: you'll be the ones designing the algorithms.

Why do we need algorithms?

- Definition: An Algorithm is a well-defined step-by-step procedure that is guaranteed to solve the problem: to take any X and output the correct Y

- Why is it required that I will be able to devise a step-by-step procedure for certain problems?
- Can't I just go and code?

- By now you've learnt that the worse thing you can do with a problem is to run and immediately code.
    - Taking one parse of the problem isn't enough — you'll end up solving a different problem
    - You don't figure out the bugs in your approach — the inputs which cause you to err
    - You end up coding some parts that are redundant
    - You hurt yourself in terms of modularity
    - You don't realize your code is inefficient
- You have to THINK before you ACT — and think rigorously!

- What do we need to do when providing an algorithm
    1. Provide an accurate description
    2. Correctness
    3. Amount of resources
        - For *any* instance? For a *good* instance? For an *average* instance?
    4. Can we do any better?
- This unit focuses (almost entirely) on #1

How to Describe an Algorithm

- Option I: Fix a language (e.g., C++) and write the code in it.
- E.g.:

```
int multiply(int a, int b) {
    int sum = 0;
    while (b > 0) {
        sum += a;
        b--;
    }
    return sum;
}
```

  - What if we wish to use a different language?
  - What if we are using a computational devise unsuited for this language? (e.g., using a function call which is machine-dependent or coding for a small sensor unit)
  - Readability
- Option II: free-form English
- "To multiply two positive integers a and b, just create a new variable sum, initialize it as 0 and as long as $b > 0$: first add a to sum and then decrement b. Finally return sum."
  - Unstructured

How to Describe an Algorithm

- Describing algorithms: `pseudocode`
  A combination of the two approaches: structured, yet abstracting away from a specific language/machine.
- I.e., we are NOT writing code! We are giving the reader a sequence of instructions that should be the outline of a code.
- Pseudocode example:

  <u>`procedure Multiply(`a`,`b`)`</u>
  $**$ Takes two integers and returns their multiplications
  $**$ `precondition:` b `is a non-negative integer`
  $**$ `postcondition:` b `is decremented to` 0
  $sum \leftarrow 0$
  `while` $(b > 0)$ `do`
      $sum \leftarrow sum + a$
      $b \leftarrow b - 1$
  `return` sum

- Pseudocode template:

  <u>`procedure PROCEDURE_NAME(` INPUT `)`</u>
  $**$ Procedure's purpose, in a nutshell
  $**$ `precondition:` what *must* be true about the input
  $**$ `postcondition:`what will happen as a by-product of the code
  `CODE: statements, if-then-else, while/for-loops, function-calls`
  `return` OUTPUT

How to Describe an Algorithm

- Pseudocode template:

```
procedure PROCEDURE_NAME( INPUT )
** Short description
** precondition:  what must be true about the input
** postcondition: what will happen as a by-product of the code
CODE: statements, if-then-else, loops, function-calls
return OUTPUT
```

- Name and a clear indication of the input is a must
- Description, pre/post-conditions are optional
- Pseudocode conventions
  - Basic Statements: assignment, basic arithmetic operations
  - Note: assignment $var \leftarrow val$ vs. boolean equality $val_1 = val_2$
  - Block structure — indicated by indentation
  - Conditionals: `if` (boolean condition) `then` BLOCK `else` BLOCK
  - While-loop: `while` (boolean condition)-do BLOCK
  - For-loop: `for (variable` var `from` val_first `to` val_last`) do BLOCK`
    - for-loops are equivalent to:
      $var \leftarrow val\_first$
      `while (` $var \leq val\_last$ `) do {` `BLOCK; increment(`var`)}`
  - Method: $name(par1, par2, \ldots)$
  - ** or ▷ comment
- Warning: the book has slightly different conventions (p. 20-22)
- You are free to adopt your own conventions — as long as they are CONSISTENT AND CLEAR

How to Describe an Algorithm

- Writing pseudocode (like writing code) is an art
  - The more you write, the better you get at it
- However, the goal of a pseudocode is to convey the sequence of instructions to the reader, so clarity is your #1 priority (in terms of "style").
  - I.e., reading the code, it should be clear WHAT it does
  - But not necessarily WHY, and not always HOW (due to encapsulation).
- One of the key concepts in writing a good pseudocode is *encapsulation*.
-
```
procedure Power(b,n)
** Returns b^n
** precondition: b and n are both non-negative integers
res ← 1
while (n > 0) do
    res ← Multiply(res, b)
    n ← n − 1
return res
```
- Similarly, better to write a single line "exchange(a, b)" rather than 3 lines:

$temp \leftarrow a$
$a \leftarrow b$
$b \leftarrow temp$

How NOT to Describe an Algorithm

- When asked to "give / describe an algorithm" always means "write drown a series of instructions that solve the problem on any instance" — and best in pseudocode.
- Writing pseudocode (like writing code) is an art — and you'll improve with time
- But still, there are a few bad habits you should avoid from the get-go.

- Bad idea: a non-informative pseudocode

  procedure Multiply(a,b)

  return $a \times b$
- Bad idea: statements that are overly descriptive

  procedure Multiply(a,b)

  set $result$ to zero first, then add a, and keep doing that until you count to b
- Bad idea: not providing a general recipe, but rather giving an example (often comes with the non-exact description from before)

  procedure Multiply(a,b)

  Set $result$ to zero

  Now add a to $result$ as long you count from 1 to b

  So if $a = 7$ and $b = 4$ we get $result = 7 + 7 + 7 + 7$.

Model of computation: RAM

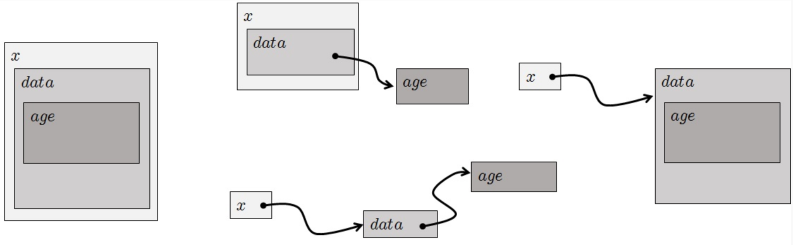
- In general, we try to abstract away from a particular machine and assume a generic computation model.
- Alas, we have to make some assumptions about our computational model.
- Benign assumptions:
  1. Code is run sequentially (not concurrently)
  2. Variables are local (unless stated otherwise)
  3. Each basic instruction takes some constant number of clock-tics
- One important assumption: integers are represented by their binary representation
  - E.g., 13 is given by 1101
  - Q: How many bits does it take to represent a number n?
  - A: With i bits we can write numbers from $\underbrace{\mathtt{000..0}}_{i} = 0$ to $\underbrace{\mathtt{111...1}}_{i} = 2^i - 1$.

    If n requires i bits, then $n \leq 2^i - 1$.
    But since n requires i bits and not $i-1$ bits, then $n > 2^{i-1} - 1$.
    So i is the smallest integer satifying $2^i \geq n+1$, i.e. $\lceil \log_2(n+1) \rceil$ bits.
- Most important assumption — RAM model: random access machine
- We assume that each memory cell has an address, and by using this address we get *direct* access to the cell.
- ...A.K.A pointers.

  In other words, using pointers takes a constant number of clock-tics — and it is regardless of how the memory is organized.
- But note that we do not make any assumptions about memory organization or memory hierarchy (disk, CPU memory, cache)

Model of computation: RAM

- We assume that each memory cell has an address, and by using this address we get *direct* access to the cell, via *pointers*.
- In fact, we will think of fields/attributes as pointers as well — namely, ignore memory implementation issues.
- So when an element x has field which is $data$ and we change some field of $data$ (e.g., $x.data.age \leftarrow 65$) the objects in the machine's memory may be arranged in any of the following forms, and they are all equivalent to us:

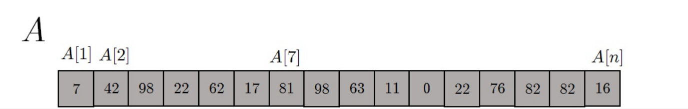


- Lastly, there's the pointer that is invalid/uninitialized: `nil`
  So better code:

  if ($x \neq$**nil** **and** $x.data \neq$**nil**) **then**
  $\quad x.data.age \leftarrow 65$

First Data-Structure: Array

- A *capacitated* data-structure, for a collection of elements of *the same type*, where each element is associated with a unique *index / key*.

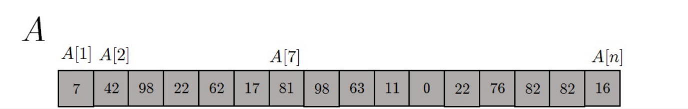


  - An array is created to hold n elements, and if you want to hold more than n elements – you must create a new one.
  - You cannot have multiple types in the same array (all elements are of the same type)
  - Most importantly: Accessing a particular element by index / key takes only a constant number of clock-tics.
    I.e., to access $A[707]$ we do not need to read all 707 first indices of A
  - In general, a cell in an array can be empty (or `nil`), but (unless told otherwise) assume all cells are filled (or that at least the first indices $A[1, ..., k]$ are filled)
  - But we do not assume the elements in A are sorted (or even comparable), unique, or satisfy any other "nice" property.

- We think of the array as a consecutive chuck of memory, designed to contain the n elements. Thus, the index i ranges between 1 and n (the array's capacity)

First Data-Structure: Array

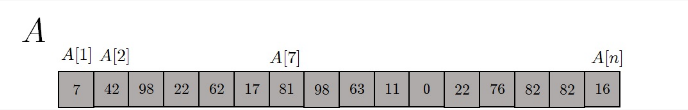
- A *capacitated* data-structure, for a collection of elements of *the same type*, where each element is associated with a unique *index / key*.
- We think of the array as a consecutive chuck of memory, designed to contain the n elements. Thus, the index i ranges between 1 and n (the array's capacity)



- Our notation:
  - First element indexed by 1, last by array's capacity (Not 0, as opposed to `C` / `C++` / `Java`)
  - A subarray is a contiguous chunk of indices $A[p...q]$ contains $\underline{q-p+1}$ elements: $A[p], A[p+1], ..., A[q-1], A[q]$.
- Question: how many non-empty subarrays are there?
  - Case 1: $p = q$ (subarrays of size 1) — there are n subarrays of size 1
  - Case 2: $p < q$ (subarryas of size > 1) — each subarray is uniquely identified by p and q; there are $\binom{n}{2}$ way to pick p and q.
  - Conclusion: $\binom{n}{2} + n = \frac{n(n-1)}{2} + n = \frac{n(n+1)}{2}$.
  - Note how $\frac{n(n+1)}{2} = (n + (n-1) + (n-2) + ... + 2 + 1) = \sum_{p=1}^{n}(n+1-p)$ which gives you an alternative way of counting all (non empty) subarrays...
- Q: how many prefix-subarrays are there? (of the form $A[1...p]$ for some p?)

First Data-Structure: Array

- A *capacitated* data-structure, for a collection of elements of *the same type*, where each element is associated with a unique *index* / *key*.
- We think of the array as a consecutive chuck of memory, designed to contain the n elements. Thus, the index i ranges between 1 and n (the array's capacity)



- Our notation:
  - First element indexed by 1, last by array's capacity
    (Not 0, as opposed to `C` / `C++` / `Java`)
  - A subarray is a contiguous chunk of indices
    $A[p...q]$ contains $\underline{q-p+1}$ elements: $A[p], A[p+1], ..., A[q-1], A[q]$.
  - When dealing with arrays to which we insert / remove elements it would be convenient to assume the array also has fields $capacity$ (how many elements it may contain) and $size$ (how many elements it currently contains)

    <u>`procedure Insert(`A, x`)`</u>
    `if` $(A.size < A.capacity)$ `then`
    $\quad A.size \leftarrow A.size + 1$
    $\quad A[A.size] \leftarrow x$

First Data-Structure: Array

- Of course, there can be many potential ways to do `insert`.
- `procedure Insert2(`A, x`)`

  `if (`$A.size < A.capacity$`) then`
      `for (`i `from` $A.size$ `downto 1) do`
          $A[i+1] \leftarrow A[i]$
      $A[1] \leftarrow x$
      $A.size \leftarrow A.size + 1$

  What's the difference from the previous version of `Insert`?
- Similarly, to delete a certain $A[i]$

  `procedure Delete(`A, i`)`

  `if (`$i \leq A.size$`) then`
      $A[i] \leftarrow A[A.size]$
      $A[A.size] \leftarrow$`nil` $**$ optional
      $A.size \leftarrow A.size - 1$

First Data-Structure: Array

- And there are other things one might wish to do on arrays:
  - Find a particular element in an array
  - Find largest / smallest element in an array
  - Copy one (sub)array into another
  - Merge two arrays
  - Sort an array
  - ...
- We will provide pseudo-code for those — using a recursion

- Think of recursion as “cheating, but with a lame friend”
  - I am asking you to solve a problem.
  - You don’t know how to solve the problem, so naturally you think of cheating and ask your friend to do it for you.
  - Your friend replies: ”I don’t know how to solve it on this instance...”
  - ”...but if only your instance was one element smaller / just one instance preceding this one — then I would have helped you.”
  - So your job now becomes:
    Find a way to <u>leverage on a solution for a smaller instance</u> to get a solution to the original instance.
  - And of course, making sure you know what to do with the first / smallest / simplest of all instances.

Recursion

- The factorial operation: $n! = \prod_{i=1}^{n} i$ (with $0! = 1$)
- Recursive implementation:

```
procedure factorial(n)
if (n = 0) then
   return 1                              ** Base case
else
   return n × factorial(n − 1)           ** Recursive call
```

- Recursion: solving the problem by assuming we know how to solve smaller / preceding instances.
- Must-haves in recursion:
  - A *clearly defined* base case - as first step
  - Recursing on *preceding* and *well-defined* instances
  - A correct transition

Problem Solving using Recursions

- Recursion: solving the problem by assuming we know how to solve smaller / preceding instances.
- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #1: Finding an element in an array —
  Input: an array A of n elements, x an element of the same type.
  Output: i such that $A[i] = x$ or `nil` if no such i exists.
  - Q1: What will be my base case?
  - A1: The empty array, return `nil`.
  - Q2: Can I solve this problem if I assume I know how to solve it on smaller instances?
  - A2: Yes. After all, the sought-after index can either be n or an index strictly smaller than n.
- procedure Find(A, n, x)

```
if (n = 0) then
   return nil
else if (A[n] = x) then
   return n
else
   return Find(A, n − 1, x)
```

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #2: Finding the largest element in an array — `FindMax(`A`)` returns $\max\{A[1], A[2], ..., A[n]\}$
  - Q1: What will be my base case?
  - A1: The array of size 1, return $A[1]$.
  - Q2: Can I solve this problem if I assume I know how to solve it on smaller instances?
  - A2: Yes. The max-element is either $A[n]$ or the max of $A[1, .., n-1]$.
- **procedure** `FindMax(`A, n`)`

```
if (n = 1) then
   return A[1]
else
   return max(A[n], FindMax(A, n − 1))
```

- How will you write, using a recursion, `Sum(`A`)` which returns $A[1] + A[2] + ... + A[n]$?

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #3: A person is about to climb a staircase of n stairs. Each step she makes can be either a regular-size step that climbs 1 stair, or a big-size step that climbs 2 stairs. In how many different ways can she climb the whole set of n steps?
  - E.g., for climbing 3 stairs she has 3 possible ways: (reg,reg,reg), (reg,big), (big,reg).
  - E.g., for climbing 4 stairs she has 5 possible ways: (reg,reg,reg,reg), (big, big), (big, reg,reg), (reg,reg,big), (reg, big, reg).
- The recursive solution: Let's look at her last step. It is either reg or big. If it is reg, then she first has to get to stair $n-1$; if it is big, she first needs to get to stair $n-2$. Denote $W(n) =$#ways to climb n stairs.
  Above argument: $W(n) = W(n-1) + W(n-2)$.
  Base cases: climbing 0 stairs – a single way: ();
  climbing 1 stair – a single way: (reg).
- So $W(0) = W(1) = 1$ and for any $n \geq 2$: $W(n) = W(n-1) + W(n-2)$.
- Looks familiar?

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #4: The towers of Hanoi. You are given 3 pegs: A, B, C. On A there are n disks, from the largest (on the bottom) to the smallest (on the top).
  In one valid move $X \to Y$, you can take the top disk on peg X and place it at the top of peg Y — only if this disk is smaller than all disks already placed on Y. (If Y is empty, the move is always valid.)
  What is the minimal number of moves required to move all disks from A to C?
  - E.g., for moving 1 disk we only need one move: $A \to C$.
  - E.g., for moving 2 disks we need 3 moves: $A \to B$, $A \to C$, $B \to C$.
- The recursive solution:
  1. Move the top $n-1$ disks from A to B.
  2. Move the nth disk from A to C.
  3. Move the $n-1$ disks from B to C.
- But steps 1 and 3 are not strictly similar to the original problem (moving from A to C...)

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #4: The towers of Hanoi. You are given 3 pegs: A, B, C. On A there are n disks, from the largest (on the bottom) to the smallest (on the top).
  In one valid move $X \to Y$, you can take the top disk on peg X and place it at the top of peg Y — only if this disk is smaller than all disks already placed on Y. (If Y is empty, the move is always valid.)
  What is the minimal number of moves required to move all disks from A to C?
- <u>`procedure Hanoi(`$n, from, to, aux$`)`</u>
  $**$ n— number of disks, positive
  $**$ $from$ — starting peg, to — goal peg, aux — intermediate peg
  `if` $(n > 0)$ `then`
      `Hanoi(`$n-1, from, aux, to$`)`
      `MoveTopDisk(`$from, to$`)` $**$ moves a single disk
      `Hanoi(`$n-1, aux, to, from$`)`

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #5: Given an array A of n items such that every two are comparable, sort A from the smallest to the largest.
- Base case: A has a single element — already sorted, so do nothing.
- How to use recursion:
  - Sort first $n-1$ elements.
  - Now put $A[n]$ in its right place.
    - Find i such that $A[i-1] \leq A[n] < A[i]$.
    - Move all elements $A[i, i+1, ..., n-1]$ to $A[i+1, i+2, ..., n]$
    - Place $A[n]$ in $A[i]$.
- procedure InsertionSort(A, n)

```
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

Problem Solving using Recursions

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- Example #5: Given an array A of n items such that every two are comparable, sort A from the smallest to the largest.
- Base case: A has a single element — already sorted, so do nothing.
- How to use recursion:
  - Sort first $n-1$ elements.
  - Putting $A[n]$ in its right place can also be done recursively.
  - Basically, because $A[1,...,n-1]$ is sorted, we can easily find the largest element and put it in the last place ($A[n]$) —
    Just compare x and $A[n-1]$.
    - If $x > A[n-1]$, it means x should be in the nth cell — and now the array is sorted.
    - If $x \le A[n-1]$ then $A[n-1]$ should be placed in the nth cell, and we now need to put x in its right place in the sorted subarray $A[1,...,n-2]$

- procedure InsertionSort(A, n)

```
if (n > 1) then
    InsertionSort(A, n − 1)
    x ← A[n]
    PutInPlace(A, n − 1, x)
```

procedure PutInPlace(A, j, key)

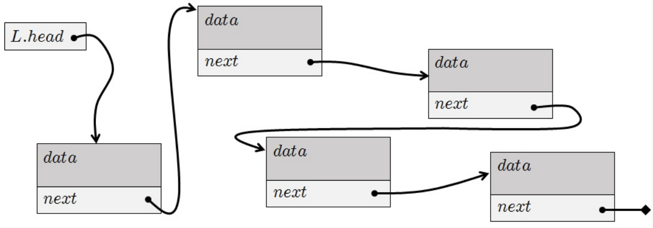
```
if (j = 0) then
    A[1] ← key
else if (key > A[j]) then
    A[j + 1] ← key
else             ** i.e., key ≤ A[j]
    A[j + 1] ← A[j]
    PutInPlace(A, j − 1, key)
```

Recursion — a Way of Thinking

- Recursion isn't a way of writing code — it's a paradigm for solving problems!
- When solving problems using a recursion, your solution must include:
  - What is your base case? (normally, size 0 or size 1 instances)
  - How will you solve the whole problem based on solution to the subproblems? Often: what will be your first step? your last step?
- Musts and Optionals in a recursion:
  - You needn't necessarily make a single recursive call
    (In the Towers of Hanoi problem we used two recursive calls)
  - You don't have to recurse on instances that are just 1 size smaller
    (That just happened to be the case in these examples. In future classes: sorting by recursing first on $A[1, ..., \frac{n}{2}]$, then on $A[\frac{n}{2} + 1, ..., n]$ and combining the two sorted halves.)
  - You can have recursions where procedure $P1$ invokes procedure $P2$, and $P2$ invokes $P1$ (on a small instance).
  - But you *must* invoke a recursion on a subproblem which is *identical* to the same original problem.
    Max, Sum, Prod are such problems. Median — not.[1]
- Recursions should be one of the first ways in which you try to tackle a new problem. It isn't always the answer / the best answer, but it should be on your go-to list!

[1] It doesn't mean that the median problem cannot be solved using a recursion, but rather that the recursion is more delicate than an obvious one.

Recursion — a Way of Thinking

- Recursion isn't a way of writing code, and isn't just a paradigm for solving problems.
- It's truly a way of thinking.
- In fact, we can use recursions to define mathematical objects.
- Definition: A *linked-list* is either
  - An empty data-structure (denoted `nil`)
  - A data-structure composed of a special node, called *head*, with two fields: $data$ and $next$.
    (i) Under the field $data$, the head holds a single element.
    (ii) Under the field $next$, the head holds a pointer to a linked-list.
- 

Linked Lists

- Lists are uncapacitated, and as we are dealing with pointers only, it is possible that each $data$ is of a different type.
  - It isn't common to mix types in lists, but it has been known to happen. A likely example: error log (each error has a different type, a different error msg, a different debugging info attached to it).
- But lists are also not-indexed
  E.g., to access node #409 we must iterate over the first 408 nodes.
- Finding a particular value — done, not surprisingly, using recursion:
- <u>`procedure Find(`L, x`)`</u>

  `if (`$L =$`nil) then`
  `    return nil`
  `else if (`$L.head.data = x$`) then`
  `    return `L $\quad **$ we return the list whose $head$ is x
  `else`
  `    return Find(`$L.head.next, x$`)`

Linked Lists

- Lists are uncapacitated, and as we are dealing with pointers only, it is possible that each $data$ is of a different type.
- But lists are also not-indexed
  E.g., to access node #409 we must iterate over the first 408 nodes.
- Finding a particular value — done, not surprisingly, using recursion
- Insertion and deletion — require solely we deal with the $head$ of the list
- `procedure InsertHead(`L, x`)`
  ** inserts a new node whose $data = x$ as L's new head
  $temp \leftarrow L$
  $L.head \leftarrow$ `new node`
  $L.head.data \leftarrow x$
  $L.head.next \leftarrow temp$
- `procedure DeleteHead(`L`)`
  `if (`$L.head \neq$`nil) then`
  $temp \leftarrow L.head$
  $L.head \leftarrow L.head.next$
  `delete` $temp$
- Note: if you wish to insert / delete the element following node #2988 — your new node is the head of the list that node-2988's $next$ points to...

Linked Lists

- There are of course versions of linked-lists:
  - Doubly linked-list:
    Also has a special node $tail$ where using the list's $tail$ and the pointer $prev$ for each node we get a linked list in the reverse order.
  - Circular linked-list:
    Last element points to the head of the list.
  - Linked-list with sentinel:
    `nil` is represented by a dummy node.
  - Any combination of the previous 3
- Doubly-linked list (or rather, just having a pointer $L.tail$ to the last node in the list) makes it faster to merge / concatenate lists.
- HW: Assuming lists have a pointer $tail$, write a pseudocode for `Merge`(L_1, L_2) which puts L_2 at the end of L_1. (Pay attention to `nil`)

Summary

- An algorithm: a set of instructions that is guaranteed to always take us from the input to the correct output.
- Pseudocode: the way we describe algorithms
  - We abstract away from a particular machine / language / code
  - But stick to the notion of structured set of instructions
  - And put readability as the key notion — "does someone who reads this knows what the code does?"
  - RAM model: access data using pointers, takes only constant number of clock-tics
- Recursion: a powerful paradigm for solving problems
  - How to solve a very simple (base) case
  - How to solve the general case *based on the ability* to solve smaller / preceding instances.
  - And a paradigm for defining (data-structures in our case)
- Arrays: capacitated, unitype, indexed
  - `Insert`(A, x) put x in the last cell.
  - `Find`(A, x) traverses all cells, returns i s.t. $A[i] = x$.
  - `Delete`(A, i) replaces last cell with cell i and deletes last cell.
- Linked-List: uncapacitated, free-form, non-indexed
  - `Insert`(L, x) put x in the top of the list.
  - `Find`(L, x) traverses the list, returns $\bar{L}$ s.t. $\bar{L}.head = x$.
  - `DeleteHead`(L) deletes the list's head.