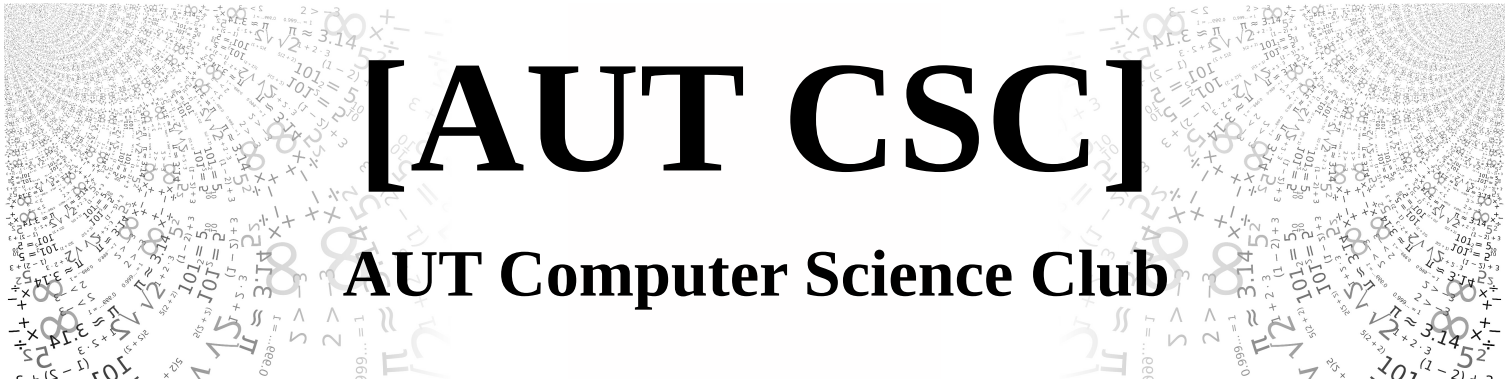
CONS cells

Or: start simple!

Glen Osborne

3rd August, 2017



Outline

All those data structures...

We have an *embarassment* of different data structures available to us:

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)
- Trees

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)
- Trees
- Graphs

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)
- Trees
- Graphs
- And so many more!

All those data structures...

We have an *embarassment* of different data structures available to us:

- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)
- Trees
- Graphs
- And so many more!

These can be *very* complicated to understand and implement!

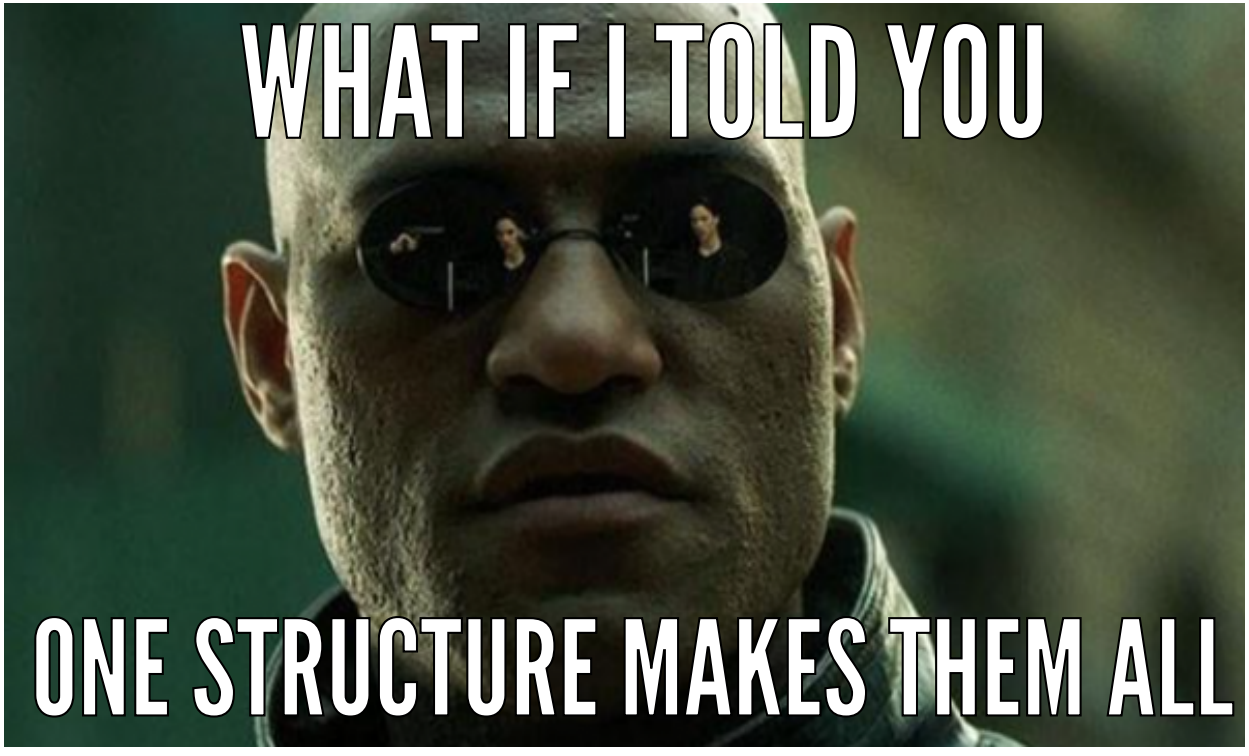
All those data structures...

We have an *embarassment* of different data structures available to us:

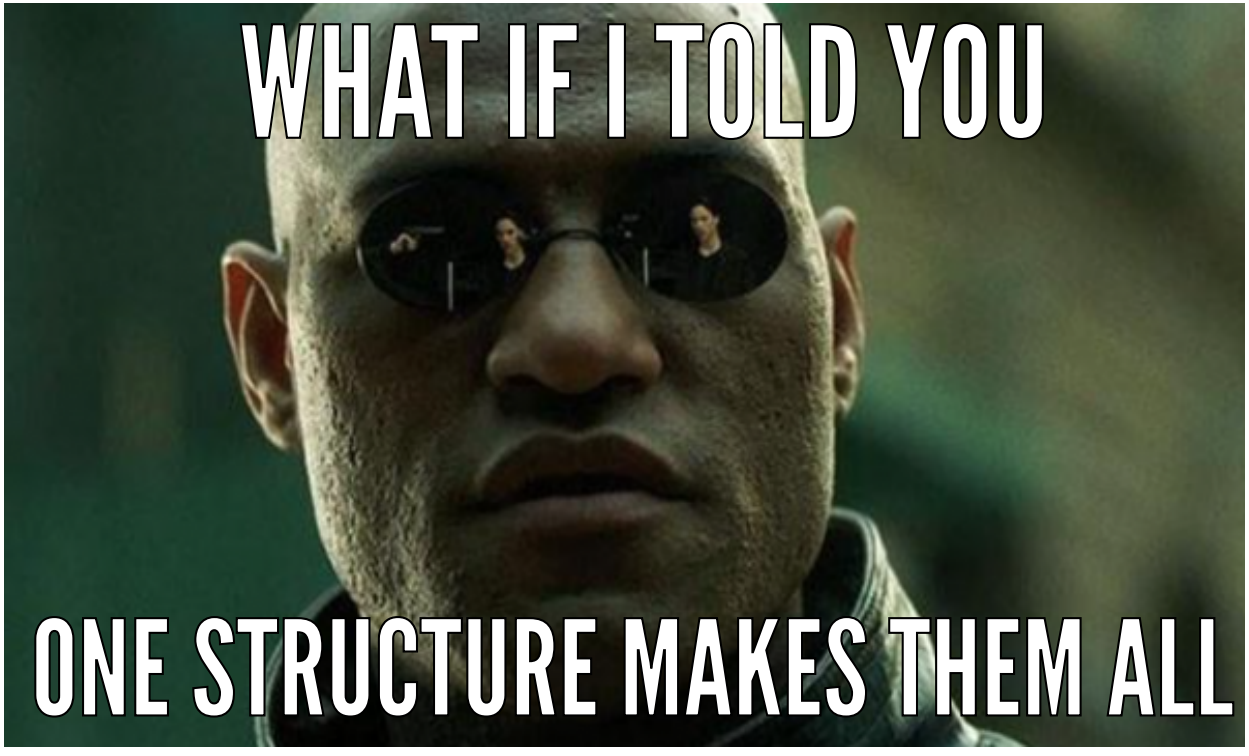
- Lists
- Dictionaries
- Matrices (with arbitrary dimensions)
- Trees
- Graphs
- And so many more!

These can be *very* complicated to understand and implement! They also often 'lock you in' to a fixed set of operations — and whether these are the ones you need can be hard to determine when initially solving a problem.

A ‘novel’ proposition

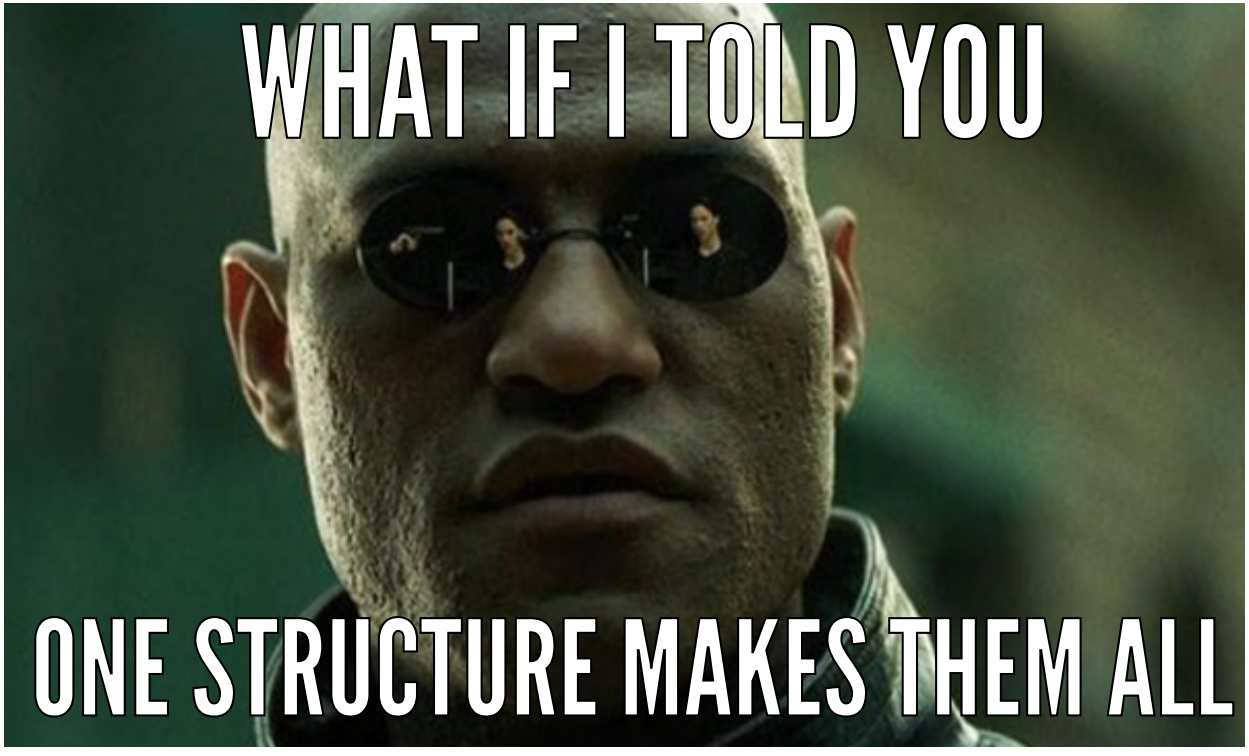


A ‘novel’ proposition



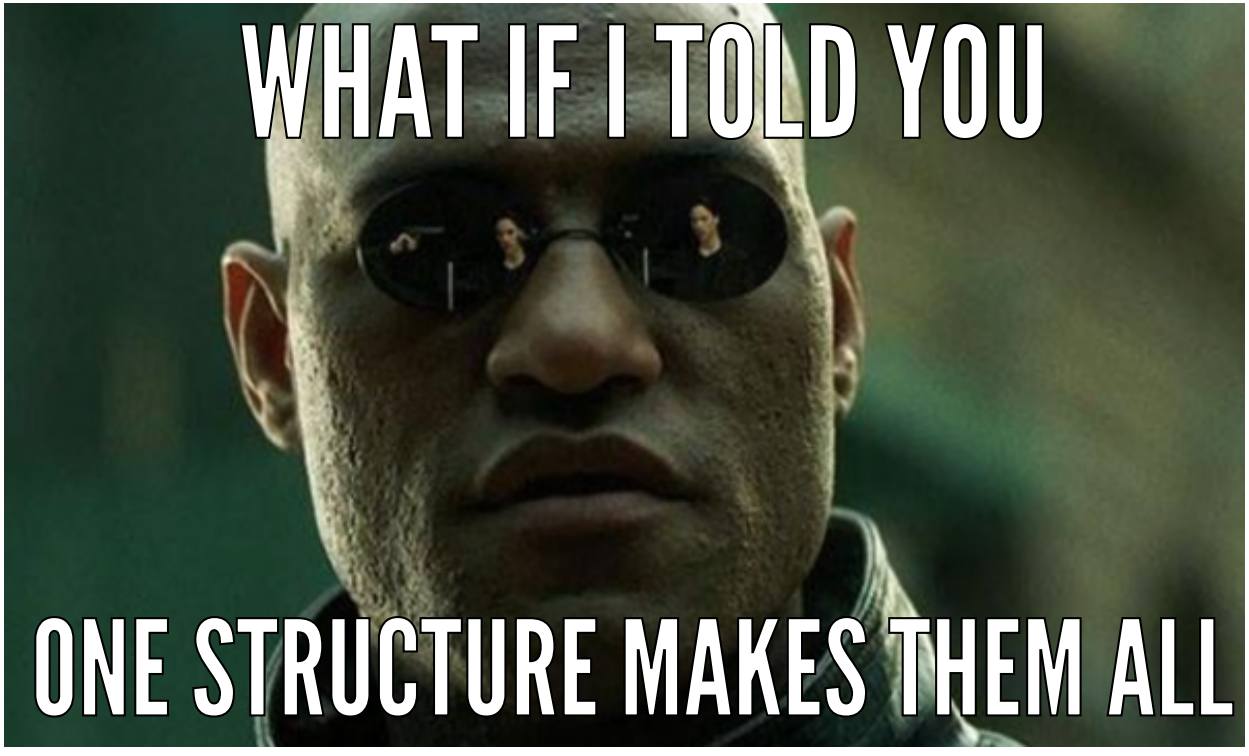
Yes, really!

A ‘novel’ proposition



Yes, really! Not a new idea by any means — first put forward in the 19*50*s.

A ‘novel’ proposition



Yes, really! Not a new idea by any means — first put forward in the 19*50*s. Let’s see how this is possible...

Some definitions

Some definitions

Definition

A *machine word* is a fixed-size chunk of memory.

Some definitions

Definition

A *machine word* is a fixed-size chunk of memory.

What this ‘fixed size’ is doesn’t matter — most modern computers have a 32 or 64-bit machine word.

Some definitions

Definition

A *machine word* is a fixed-size chunk of memory.

What this 'fixed size' is doesn't matter — most modern computers have a 32 or 64-bit machine word.

Definition

A piece of data (*datum*) is *primitive* if it fits into one machine word. Otherwise, it is *complex*.

Some definitions

Definition

A *machine word* is a fixed-size chunk of memory.

What this 'fixed size' is doesn't matter — most modern computers have a 32 or 64-bit machine word.

Definition

A piece of data (*datum*) is *primitive* if it fits into one machine word. Otherwise, it is *complex*.

Definition

A *reference* is a primitive datum, consisting of a memory address.

Some definitions

Definition
A *machine word* is a fixed-size chunk of memory.

What this 'fixed size' is doesn't matter — most modern computers have a 32 or 64-bit machine word.

Definition
A piece of data (*datum*) is *primitive* if it fits into one machine word. Otherwise, it is *complex*.

Definition
A *reference* is a primitive datum, consisting of a memory address.

We can imagine a reference as 'pointing' to another piece of data in memory.

Some definitions

Definition
A *machine word* is a fixed-size chunk of memory.

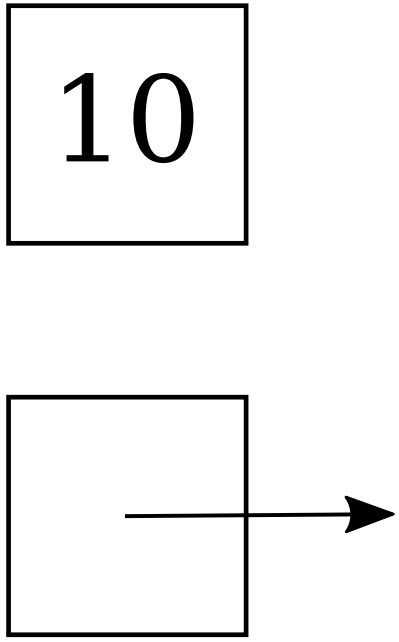
What this 'fixed size' is doesn't matter — most modern computers have a 32 or 64-bit machine word.

Definition
A piece of data (*datum*) is *primitive* if it fits into one machine word. Otherwise, it is *complex*.

Definition
A *reference* is a primitive datum, consisting of a memory address.

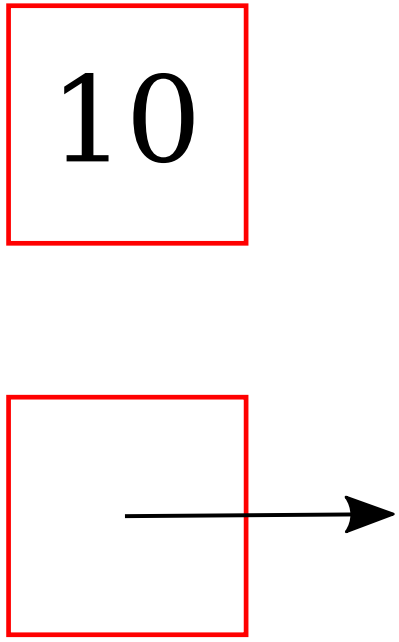
We can imagine a reference as 'pointing' to another piece of data in memory. Besides references, we also have integers and floats as primitive data (and possibly others as well).

Gettin' visual with it



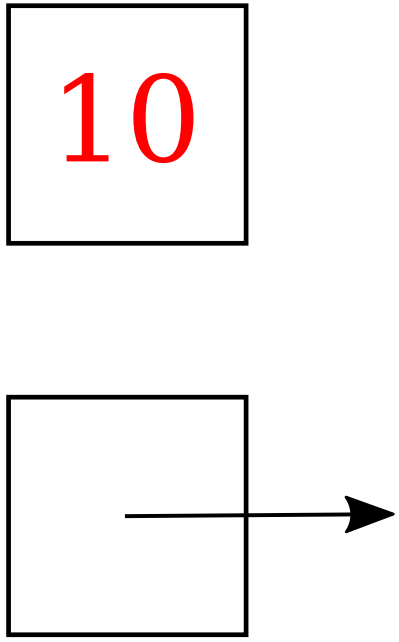
We will use drawings like this one to represent data in memory.

Gettin' visual with it



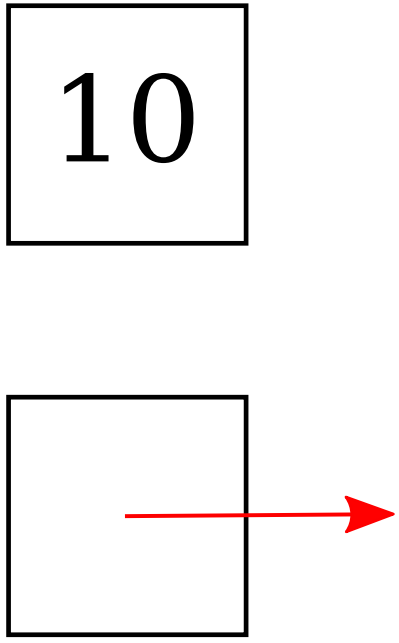
We represent machine words using boxes. Adjacent boxes are adjacent machine words in memory (these two are not).

Gettin' visual with it



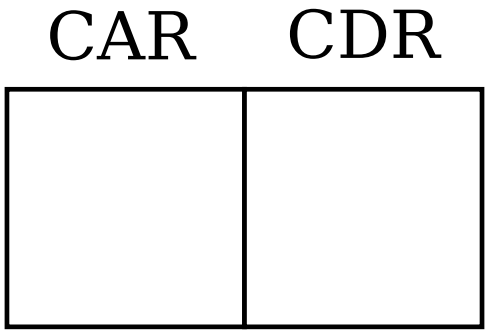
Non-reference primitive data will be drawn inside the box representing its machine word.

Gettin' visual with it

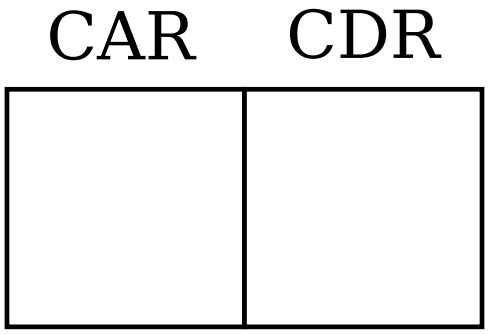


References will be drawn as arrows to the data they 'point' to (this one isn't pointing anywhere useful).

The CONS cell

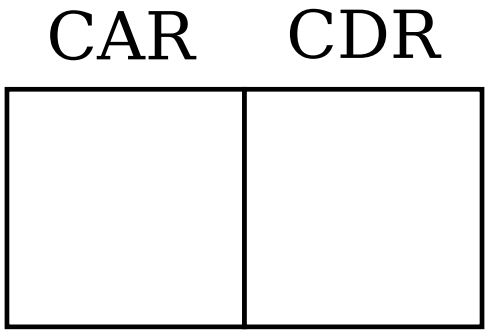


The CONS cell



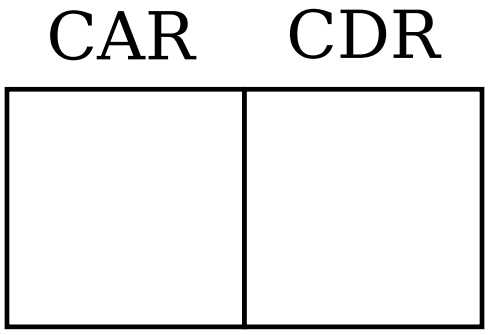
- Consists of two adjacent machine words

The CONS cell



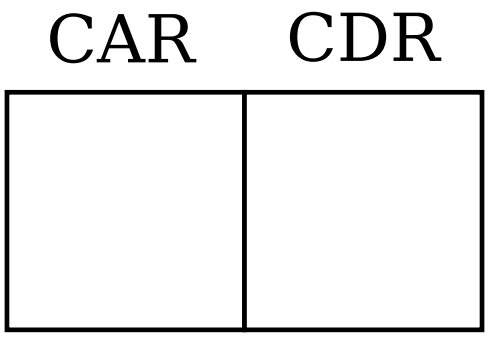
- Consists of two adjacent machine words
- A CONS cell's first word is called its *CAR* (pronounced like 'motor vehicle')

The CONS cell



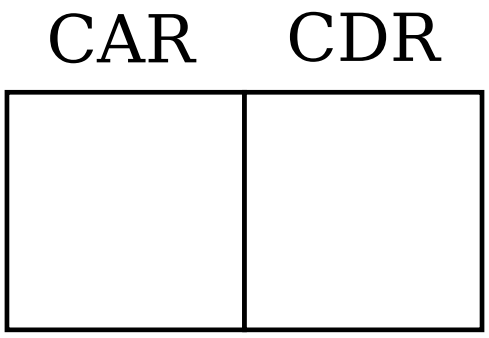
- Consists of two adjacent machine words
- A CONS cell's first word is called its *CAR* (pronounced like 'motor vehicle')
- A CONS cell's second word is called its *CDR* (pronounced like 'COULD-ruh')

The CONS cell



- Consists of two adjacent machine words
- A CONS cell's first word is called its *CAR* (pronounced like 'motor vehicle')
- A CONS cell's second word is called its *CDR* (pronounced like 'COULD-ruh')
- Either the CAR or the CDR can store reference, or primitive non-reference, data as needed

The CONS cell



- Consists of two adjacent machine words
- A CONS cell's first word is called its *CAR* (pronounced like 'motor vehicle')
- A CONS cell's second word is called its *CDR* (pronounced like 'COULD-ruh')
- Either the CAR or the CDR can store reference, or primitive non-reference, data as needed
- Can implement *every single one* of the data structures mentioned at the start of this talk!

What you're probably thinking right now

What you're probably thinking right now

Lists

Definition

NIL refers to a unique machine word representing ‘no useful data’.

Lists

Definition

NIL refers to a unique machine word representing ‘no useful data’.

Initially, we’ll assume that lists only store primitive data. We’ll later show how to overcome this.

Lists

Definition

NIL refers to a unique machine word representing ‘no useful data’.

Initially, we’ll assume that lists only store primitive data. We’ll later show how to overcome this.

The intuition here is based on the fact that a CONS cell looks a lot like a singly-linked list node.

Lists

Definition

NIL refers to a unique machine word representing 'no useful data'.

Initially, we'll assume that lists only store primitive data. We'll later show how to overcome this.

The intuition here is based on the fact that a CONS cell looks a lot like a singly-linked list node. More precisely, we can store each node's *data* in the CAR, and *'next' reference* in the CDR.

Lists

Definition

NIL refers to a unique machine word representing 'no useful data'.

Initially, we'll assume that lists only store primitive data. We'll later show how to overcome this.

The intuition here is based on the fact that a CONS cell looks a lot like a singly-linked list node. More precisely, we can store each node's *data* in the CAR, and *'next' reference* in the CDR. For the last node, we can have its 'next' reference 'point to' NIL to mark the end of the list.

Lists

Definition

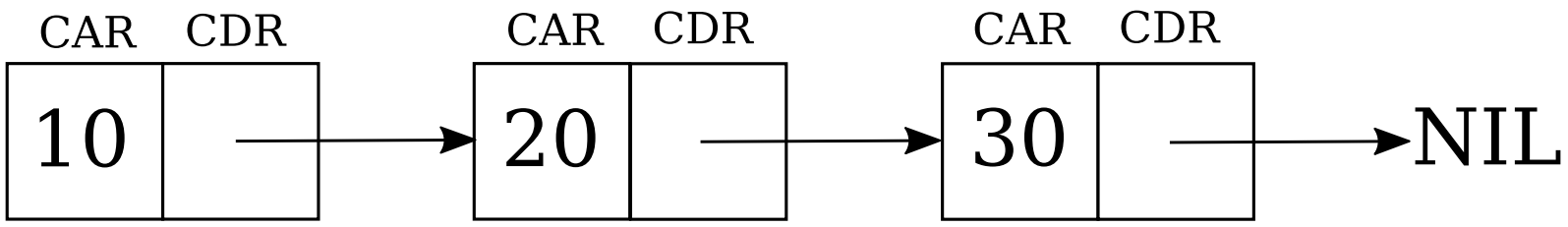
NIL refers to a unique machine word representing 'no useful data'.

Initially, we'll assume that lists only store primitive data. We'll later show how to overcome this.

The intuition here is based on the fact that a CONS cell looks a lot like a singly-linked list node. More precisely, we can store each node's *data* in the CAR, and *'next' reference* in the CDR. For the last node, we can have its 'next' reference 'point to' NIL to mark the end of the list.

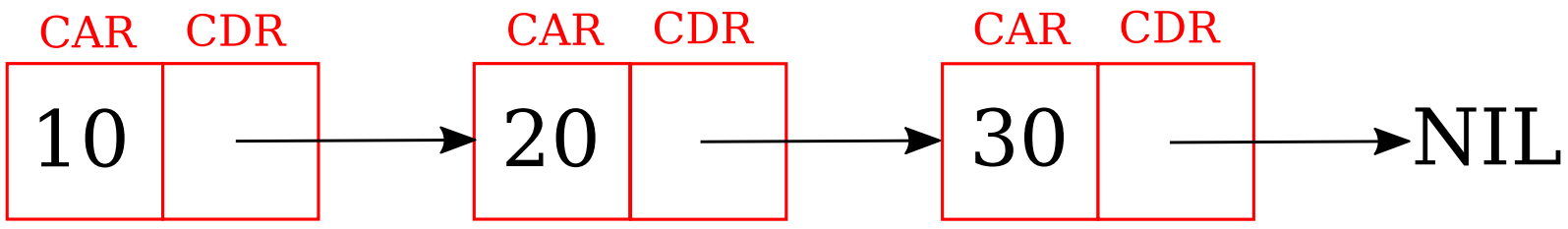
Using this approach, we can define all the usual list operations in a straightforward way.

Visualizing CONS cell lists



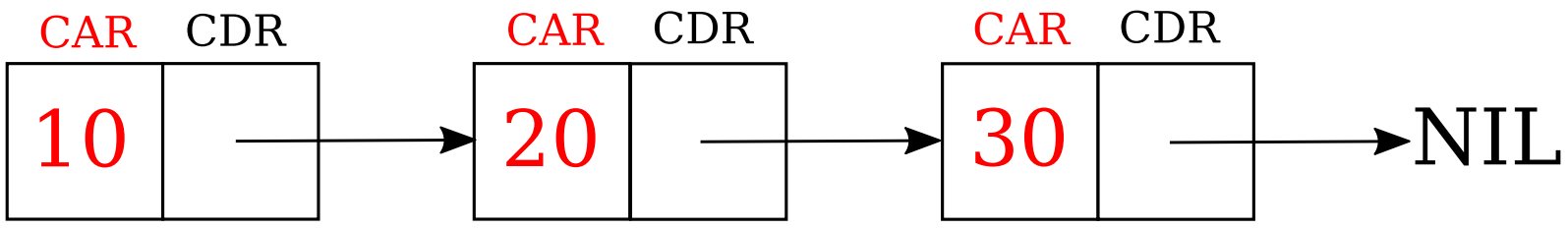
This is the in-memory representation of the list `[10, 20, 30]` using CONS cells.

Visualizing CONS cell lists



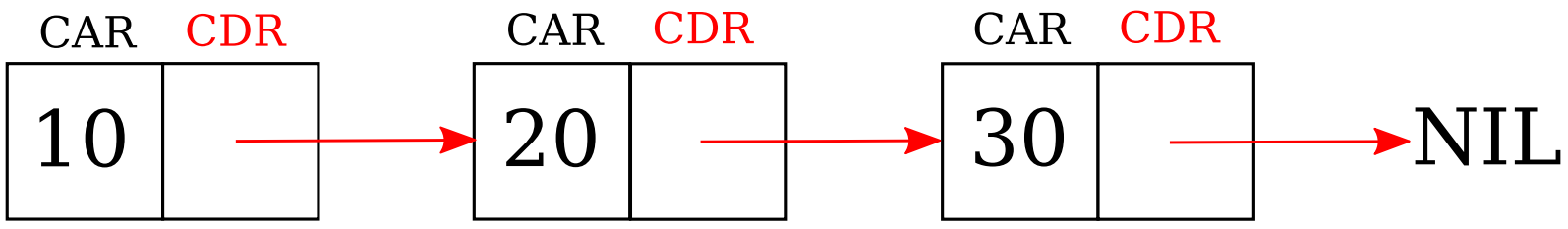
This is the in-memory representation of the list `[10, 20, 30]` using CONS cells. Each CONS cell is one list node.

Visualizing CONS cell lists



This is the in-memory representation of the list `[10, 20, 30]` using CONS cells. Each CONS cell is one list node. CARs contain the data.

Visualizing CONS cell lists



This is the in-memory representation of the list `[10, 20, 30]` using CONS cells. Each CONS cell is one list node. CARs contain the data. CDRs contain references to the next list node, or NIL for the end.

Lists with complex data

- Most interesting data won't fit into a single machine word.

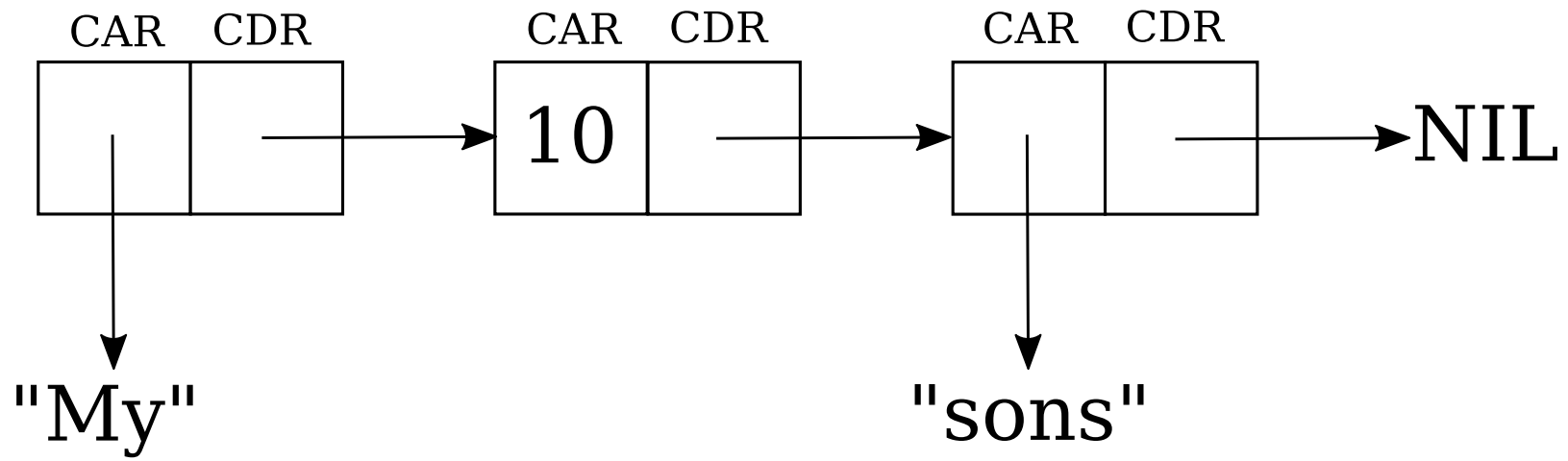
Lists with complex data

- Most interesting data won't fit into a single machine word.
- To have lists storing complex data, we have the CARS of our list node CONS cells be *references* to the data elsewhere in memory.

Lists with complex data

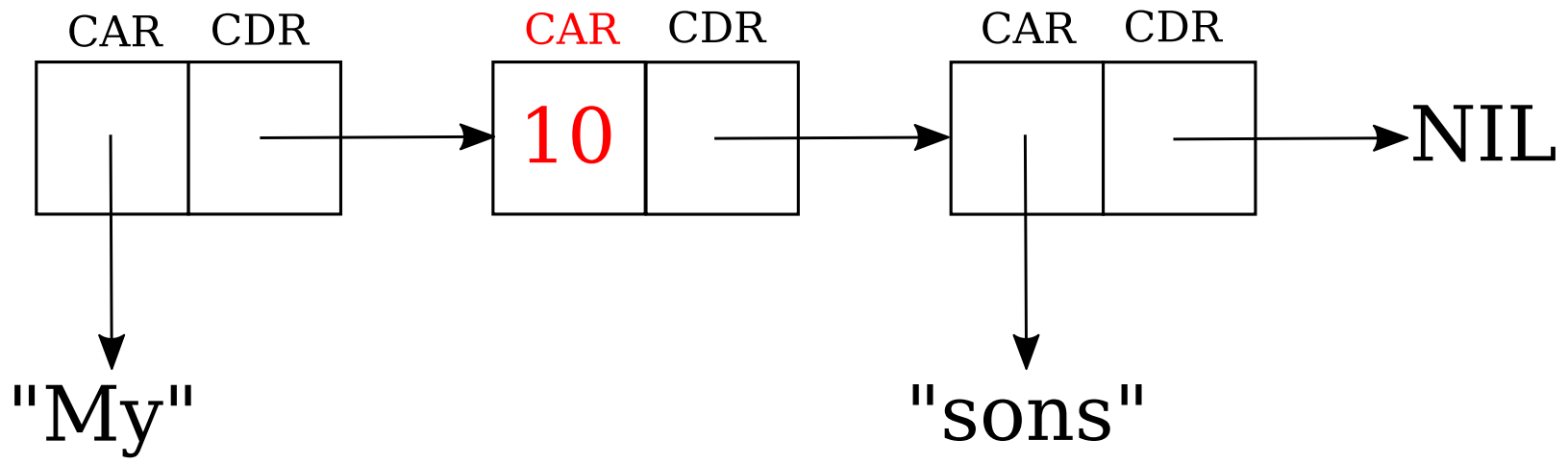
- Most interesting data won't fit into a single machine word.
- To have lists storing complex data, we have the CARS of our list node CONS cells be *references* to the data elsewhere in memory.
- We can even mix-and-match the two!

Visualizing lists with complex data



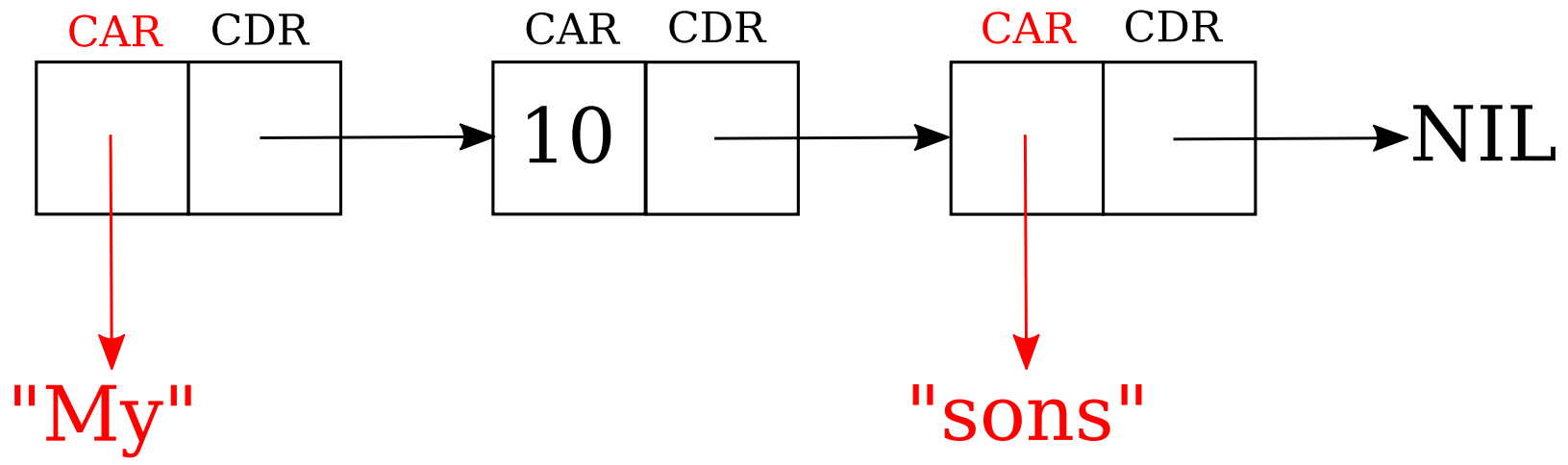
This is the in-memory representation of the list `["My", 10, "sons"]` using CONS cells.

Visualizing lists with complex data



This is the in-memory representation of the list `["My", 10, "sons"]` using CONS cells. Primitive data is stored in the CAR of the relevant cell as before.

Visualizing lists with complex data



This is the in-memory representation of the list `["My", 10, "sons"]` using CONS cells. Primitive data is stored in the CAR of the relevant cell as before. Complex data is stored elsewhere, and the CDRs of the relevant cells store references to that data instead.

Dictionaries

Dictionaries

Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

Dictionaries

Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

We can represent a key-value pair by using a CONS cell where both the CAR and CDR store data (or a reference to data).

Dictionaries

Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

We can represent a key-value pair by using a CONS cell where both the CAR and CDR store data (or a reference to data). We can then create a list of these as complex data.

Dictionaries

Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

We can represent a key-value pair by using a CONS cell where both the CAR and CDR store data (or a reference to data). We can then create a list of these as complex data. To ensure that we don't end up with key duplication, we do two things:

Dictionaries

Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

We can represent a key-value pair by using a CONS cell where both the CAR and CDR store data (or a reference to data). We can then create a list of these as complex data. To ensure that we don't end up with key duplication, we do two things:

- Ensure that queries always start from the first CONS cell in the list

Dictionaries

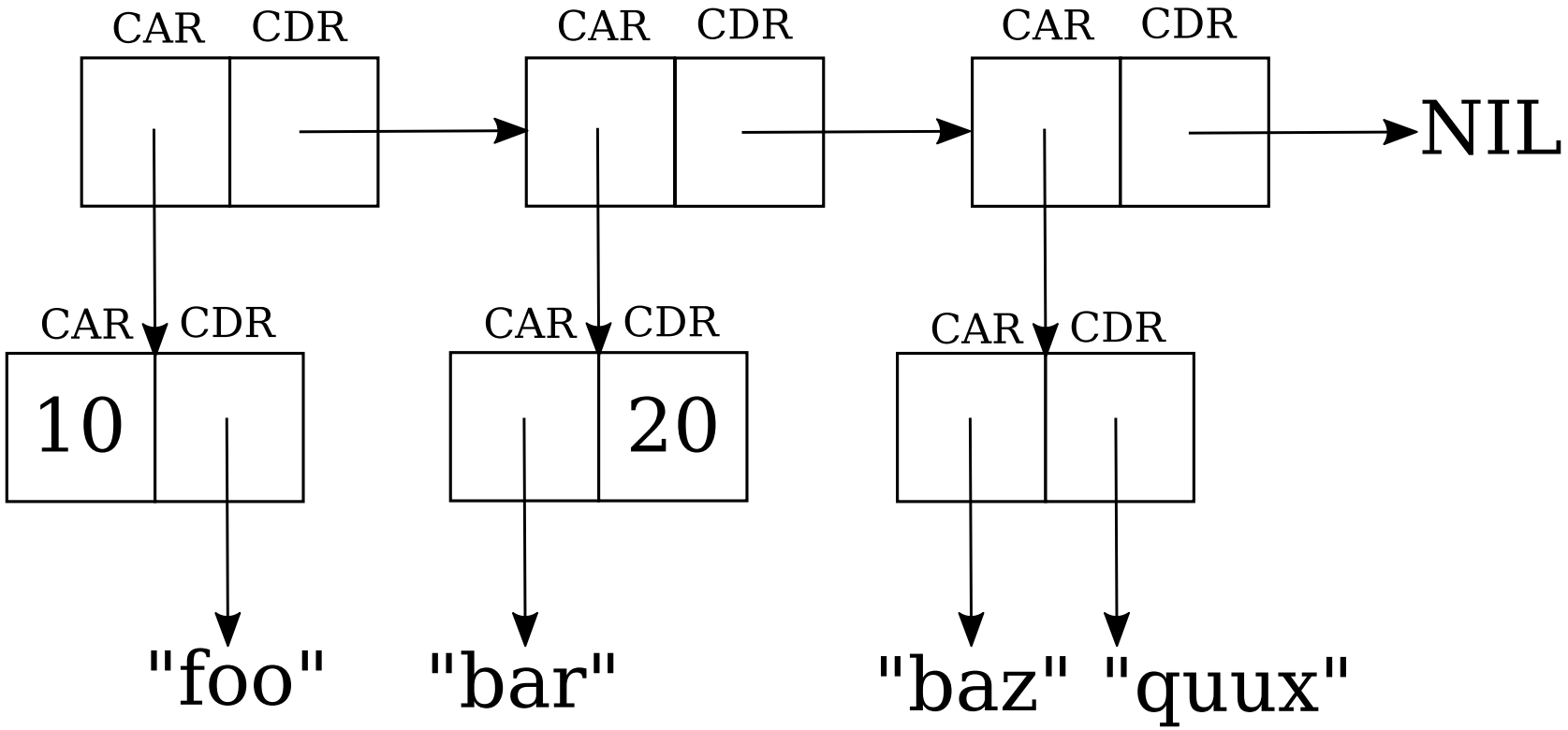
Definition

A *dictionary* stores a set of key-value pairs, such that keys in the dictionary are unique.

We can represent a key-value pair by using a CONS cell where both the CAR and CDR store data (or a reference to data). We can then create a list of these as complex data. To ensure that we don't end up with key duplication, we do two things:

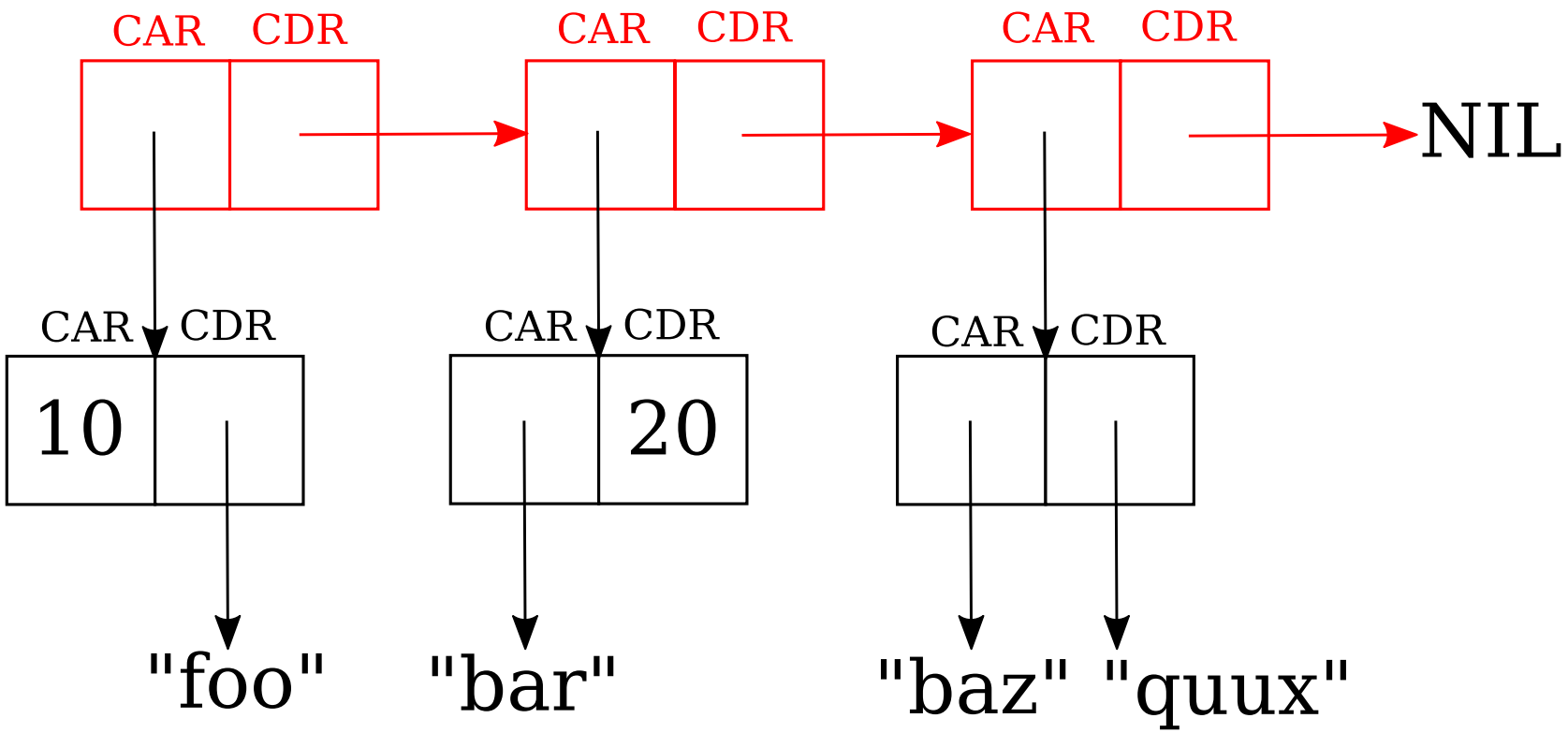
- Ensure that queries always start from the first CONS cell in the list
- Put inserts at the front of the list (so they'll be found before any previous entries with the same key)

Example of CONS cell-based dictionary



This is the in-memory representation of the dictionary $\{(10, \text{"foo"}), (\text{"bar"}, 20), (\text{"baz"}, \text{"quux"})\}$.

Example of CONS cell-based dictionary



This is the in-memory representation of the dictionary $\{(10, \text{"foo"}), (\text{"bar"}, 20), (\text{"baz"}, \text{"quux"})\}$. The entries form a list of complex data (the *spine* of the dictionary).

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk).

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

- Matrices

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

- Matrices
- Trees

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

- Matrices
- Trees
- Graphs

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

- Matrices
- Trees
- Graphs

To save time, we will only draw diagrams where useful.

A slight intermission

Given that we've now described how to define dictionaries using CONS cells, we have the ability to define any other data structure (as explained in the previous talk). However, we can implement some structures a bit better than this using CONS cells, including:

- Matrices
- Trees
- Graphs

To save time, we will only draw diagrams where useful. If we refer to lists or dictionaries anywhere, assume we mean ones based on CONS cells as described previously.

Matrices

Definition

The *rank* of a matrix is the number of dimensions it has.

Matrices

Definition

The *rank* of a matrix is the number of dimensions it has.

A rank 1 'matrix' is just a list.

Matrices

Definition

The *rank* of a matrix is the number of dimensions it has.

A rank 1 'matrix' is just a list. A rank 2 matrix is a list of lists, a rank 3 matrix is a list of rank 2 matrices, and so on, and so forth.

Matrices

Definition

The *rank* of a matrix is the number of dimensions it has.

A rank 1 'matrix' is just a list. A rank 2 matrix is a list of lists, a rank 3 matrix is a list of rank 2 matrices, and so on, and so forth.

We can define a rank n matrix (for $n > 1$) as a spine of rank $n - 1$ matrices.

Matrix example

Consider the following rank 2 matrix:

Matrix example

Consider the following rank 2 matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

We can represent it in two ways.

Matrix example

Consider the following rank 2 matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

We can represent it in two ways. We can have the spine go along the columns:

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Matrix example

Consider the following rank 2 matrix:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

We can represent it in two ways. We can have the spine go along the columns:

```
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

or along the rows:

```
[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
```

Trees

We will begin with a very simple kind of tree: a *binary leaf tree*.

Trees

We will begin with a very simple kind of tree: a *binary leaf tree*. More precisely:

Definition

A *binary leaf tree* has data only in its leaf nodes, and its internal nodes have a maximum of two children.

Trees

We will begin with a very simple kind of tree: a *binary leaf tree*. More precisely:

Definition

A *binary leaf tree* has data only in its leaf nodes, and its internal nodes have a maximum of two children.

We can represent a binary leaf tree using CONS cells as follows:

Trees

We will begin with a very simple kind of tree: a *binary leaf tree*. More precisely:

Definition

A *binary leaf tree* has data only in its leaf nodes, and its internal nodes have a maximum of two children.

We can represent a binary leaf tree using CONS cells as follows:

- An internal node is represented by a CONS cell whose CAR stores a reference to its left child (or NIL if none) and whose CDR stores a reference to its right child (or NIL if none)

Trees

We will begin with a very simple kind of tree: a *binary leaf tree*. More precisely:

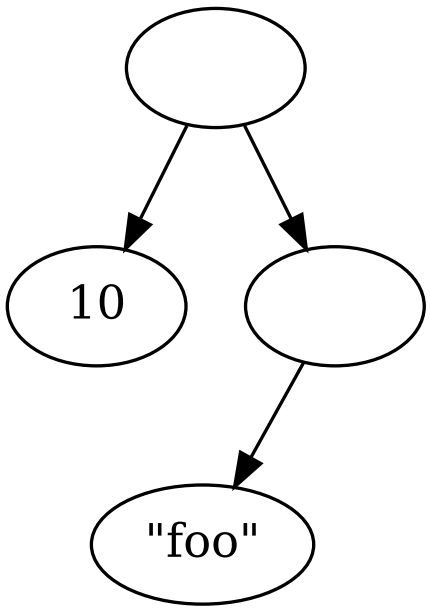
Definition

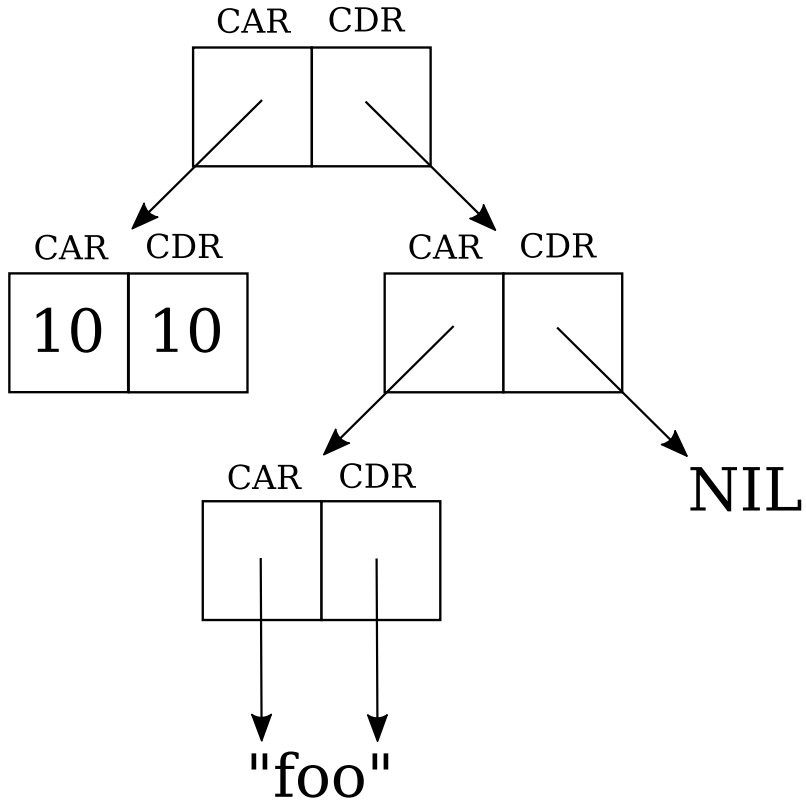
A *binary leaf tree* has data only in its leaf nodes, and its internal nodes have a maximum of two children.

We can represent a binary leaf tree using CONS cells as follows:

- An internal node is represented by a CONS cell whose CAR stores a reference to its left child (or NIL if none) and whose CDR stores a reference to its right child (or NIL if none)
- A leaf node is represented by a CONS cell storing the same data (or reference) in both its CAR and CDR

Example binary leaf tree and its CONS cell representation





Rose trees

If we need trees of arbitrary branching factor (number of children), or where data has to be stored in the leaves as well, we can instead use a *rose tree* representation:

Rose trees

If we need trees of arbitrary branching factor (number of children), or where data has to be stored in the leaves as well, we can instead use a *rose tree* representation:

- A leaf node is represented the same way as for binary leaf trees

Rose trees

If we need trees of arbitrary branching factor (number of children), or where data has to be stored in the leaves as well, we can instead use a *rose tree* representation:

- A leaf node is represented the same way as for binary leaf trees
- An internal node is represented by a CONS cell whose CAR stores the data for that node (or a reference to the data), and whose CDR stores a list of references to its children in left-to-right order

Rose trees

If we need trees of arbitrary branching factor (number of children), or where data has to be stored in the leaves as well, we can instead use a *rose tree* representation:

- A leaf node is represented the same way as for binary leaf trees
- An internal node is represented by a CONS cell whose CAR stores the data for that node (or a reference to the data), and whose CDR stores a list of references to its children in left-to-right order

We can also include a parent reference in the list of children (typically as the first element) if we want.

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*.

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before.

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

- An adjacency list is represented by an *adjacency spine*

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

- An adjacency list is represented by an *adjacency spine*
- An adjacency spine is made up of *vertex cells*

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

- An adjacency list is represented by an *adjacency spine*
- An adjacency spine is made up of *vertex cells*
- A vertex cell is a CONS cell whose CAR stores a reference to the next vertex cell in the adjacency spine, and whose CDR stores a *vertex list*

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

- An adjacency list is represented by an *adjacency spine*
- An adjacency spine is made up of *vertex cells*
- A vertex cell is a CONS cell whose CAR stores a reference to the next vertex cell in the adjacency spine, and whose CDR stores a *vertex list*
- A vertex list is a CONS cell whose CAR stores the vertex's label (or a reference to it), and whose CDR stores a reference to a *child list*

Graphs

Typically, graphs are stored as an *adjacency matrix* or an *adjacency list*. The adjacency matrix is easy to represent using a rank 2 matrix as we defined before. The adjacency list is a bit trickier:

- An adjacency list is represented by an *adjacency spine*
- An adjacency spine is made up of *vertex cells*
- A vertex cell is a CONS cell whose CAR stores a reference to the next vertex cell in the adjacency spine, and whose CDR stores a *vertex list*
- A vertex list is a CONS cell whose CAR stores the vertex's label (or a reference to it), and whose CDR stores a reference to a *child list*
- A child list's data are references to elements of the adjacency spine, corresponding to the neighbours of that vertex

The bad news

All of these are really neat, but unfortunately, these implementations can be slow, especially when we have a lot of data, because:

The bad news

All of these are really neat, but unfortunately, these implementations can be slow, especially when we have a lot of data, because:

- Many operations involve scanning 'chains' of CONS cells, which give us linear (or worse!) time complexity

The bad news

All of these are really neat, but unfortunately, these implementations can be slow, especially when we have a lot of data, because:

- Many operations involve scanning 'chains' of CONS cells, which give us linear (or worse!) time complexity
- CONS cells are not cache-friendly; typically, we'll only cache the cell itself, and *not* any data its CAR or CDR might hold references to

The bad news

All of these are really neat, but unfortunately, these implementations can be slow, especially when we have a lot of data, because:

- Many operations involve scanning 'chains' of CONS cells, which give us linear (or worse!) time complexity
- CONS cells are not cache-friendly; typically, we'll only cache the cell itself, and *not* any data its CAR or CDR might hold references to
- This means that we could potentially be getting cache misses *every time* we follow a stored reference!

The bad news

All of these are really neat, but unfortunately, these implementations can be slow, especially when we have a lot of data, because:

- Many operations involve scanning 'chains' of CONS cells, which give us linear (or worse!) time complexity
- CONS cells are not cache-friendly; typically, we'll only cache the cell itself, and *not* any data its CAR or CDR might hold references to
- This means that we could potentially be getting cache misses *every time* we follow a stored reference!
- There are ways to make CONS cell-based structures more cache-friendly, but they're usually more trouble than they're worth

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast
- CONS cell-based structures can be built and *extended* quickly

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast
- CONS cell-based structures can be built and *extended* quickly
- This makes them *ideal* for prototyping

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast
- CONS cell-based structures can be built and *extended* quickly
- This makes them *ideal* for prototyping
- There is no *fast* — only fast *enough*; you might find CONS cells are good enough for your purposes!

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast
- CONS cell-based structures can be built and *extended* quickly
- This makes them *ideal* for prototyping
- There is no *fast* — only fast *enough*; you might find CONS cells are good enough for your purposes!
- Once you know exactly what you need, you can always replace some (or all) of the structure with something more efficient

So why use them?

- When we initially set out to solve a problem, we don't often know what operations we need, how much data we'll have, or which operations need to be fast
- CONS cell-based structures can be built and *extended* quickly
- This makes them *ideal* for prototyping
- There is no *fast* — only fast *enough*; you might find CONS cells are good enough for your purposes!
- Once you know exactly what you need, you can always replace some (or all) of the structure with something more efficient

In short, CONS cell-based structures can help you *learn your tradeoffs* quickly and easily.

Questions?