Compound data structures in Elixir:
- **tuples**: `{:student, "Sune Manges", :cinte, 2012, :sunem}`
- **lists**: `[:sunem, :joed, :sueb, :anng]`

We could implement lists using tuples:

```elixir
<list> ::= :nil |
  '{' :cons ',' <expression> ',' <list> '}'
```

```elixir
[:foo, :bar, :zot]
[:foo | [:bar | [:zot | []]]]
{:cons, :foo, {:cons, :bar, {:cons, :zot, :nil}}}
```

Lists gives us a convenient syntax ... once you get use to it.

Lists are handled more efficiently by the compiler and run-time system.

Important to understand when to use lists and when to use tuples.
n'th element

Return the n'th element from a list of three:

```elixir
def nth_l(1, [r|_]) do r end
def nth_l(2, [_,r|_]) do r end
def nth_l(3, [_,_,r]) do r end
```

Return the n'th element from a tuple of three:

```elixir
def nth_t(1, {r,_,_}) do r end
def nth_t(2, {_,r,_}) do r end
def nth_t(3, {_,_,r}) do r end
```

n'th benchmark

Benchmark different versions of n'th:

- `elem(tuple, n)`: return the n'th element in the tuple (zero indexed)
- `Enum.at(list, n)`: return the n'th element of the list (zero indexed)
why use lists

Represent the following information using either a list or tuple:

- a playing card: ace of club, king of hearts etc
- a stack of playing cards
- 1001 movies to see before you die
- a million things you should do, in order

*Lists are sometimes called “stacks” since the push and pop are cheap.*

what about queues

How do we implement a queue?

```elixir
def add(queue, elem) do
  end
end
```

```
def remove(queue) do
  end
end
```

an ok queue

```elixir
def add(queue, elem) do
  end
end
```

```
def remove(queue) do
  end
end
```

```elixir
def add({:queue, front, back}, elem) do
  {:queue, front, [elem|back]}
end
```

```
def remove({:queue, [elem|rest], back}) do
  {:ok, elem, {:queue, rest, []}}
end
```

```
def remove({:queue, [], back}) do
  case reverse(back) do
    [] -> :fail
    [elem|rest] -> {:ok, elem, {:queue, rest, []}}
  end
end
```

trees

How do we represent a binary tree?

```elixir
How do we represent a leaf node?

```elixir
{:leaf, value}

```elixir
How do we represent a branch node?

```elixir
{:node, value, left, right}

```elixir
How do we represent an empty tree?

```elixir
:⚠️

```elixir
```

tree = {:node, :b, {:leaf, :a}, {:leaf, :c}}
```
what is happening

\[ t = \{\text{node, 38, \{leaf, 42\}, \{leaf, 34\}}\} \]

```
  node
  38
  ↘
  leaf
  42
  ↗
  leaf
  34
```

\[ z = \{\text{node, 40, \{leaf, 42\}, \{leaf, 39\}}\}, \]

\[ t = \{\text{node, 38, z, \{leaf, 34\}}\} \]

```
  node
  40
  ↘
  leaf
  42
  ↗
  leaf
  39
  ↗
  node
  38
  ↗
  leaf
  34
```

search a tree

Given a tree, implement a function that searches for a given number, returning \texttt{yes} or \texttt{no} depending on if the number is in the tree or not.

```ruby
def member(_, :nil) do :no end
def member(n, {:leaf, ...}) do :yes end
def member(_, {:leaf, ...}) do :no end
def member(n, {:node, ..., ..., ...}) do :yes end
def member(n, {:node, _, left, right}) do
  case ...
  yes → :yes
  no → ....
end
end
```

What is the asymptotic time complexity of this function?

ordered tree

How is the situation changed if the tree is ordered?

```ruby
def member(_, :nil) do :no end
def member(n, {:leaf, ... left, right}) do
  if n < v do ...
  else ...
end
```

What is the asymptotic time complexity of this function?
key-value look-up

Assume that we have an ordered tree of key-value pairs:

```elixir
t = {:node, :k, 38,
   {:node, :b, 34, :nil, :nil},
   {:node, :o, 40, {:node, :l, 42, :nil, :nil},
   {:node, :q, 39, :nil, :nil}}}
```

No special leaf nodes, empty branch is represented by :nil.

lookup in order tree

How would we implement a function that searched for a given key and returned {:value, value} if found and :no otherwise?

```elixir
def lookup(key, :nil) do ...
end
def lookup(key, {:node, key, ..., ..., ...}) do ...
end
def lookup(key, {:node, k, _, left, right}) do
  if key < k do ...
  true -> ...
  end
end
```

modify an element

```elixir
def modify(_, _, :nil) do :nil end
def modify(key, val, {:node, key, _, left, right}) do
  {:node, key, val, left, right}
end
def modify(key, val, {:node, k, v, left, right}) do
  if key < k do
    {:node, k, v, ..., right};
  else
    {:node, k, v, left, ...}
  end
end
```

insert

How do we implement insert(key, value, tree) (assuming it does not exists)?

```elixir
def insert(key, value, :nil) do ...
end
def insert(key, value, {:node, k, v, left, right}) do
  if key < k do ...
  true -> ...
  end
end
```
How do we implement delete(item, tree) (assuming it does exist)?

```elixir
define delete(key, {:node, key, _, :nil, :nil}) do ... end
define delete(key, {:node, key, _, :nil, right}) do ... end
define delete(key, {:node, key, _, left, :nil}) do ... end
define delete(key, {:node, key, _, left, right}) do
  ?
end
define delete(key, value, {:node, k, v, left, right}) do
  if key < k do
    ...
  else
    ...
  end
end
```

Algorithm first - then implement.

```elixir
define delete(key, {:node, key, _, left, right}) do
  {k, v} = ...  
  deleted = ...
  {:node, ..., ..., ..., ...}
end
```

**Summary**

- Lists: a stack structure, easy to push and pop, simple to work with
- Tuples: constant time random access, expensive to change when large
- Queues: implemented using lists, amortized time complexity O(1)
- Trees: O(lg(n)) operations