

## Trees

Johan Montelius

KTH

VT19

Compound data structures in Elixir:

- **tuples:** `{:student, "Sune Mangs", :cinte, 2012, :sunem}`
- **lists:** `[:sunem, :joed, :sueb, :anng]`

1/1

2/1

We could implement lists using tuples:

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

```
[: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.

3/1

4/1

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

```
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:

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

5 / 1

Return the n'th element from a list:

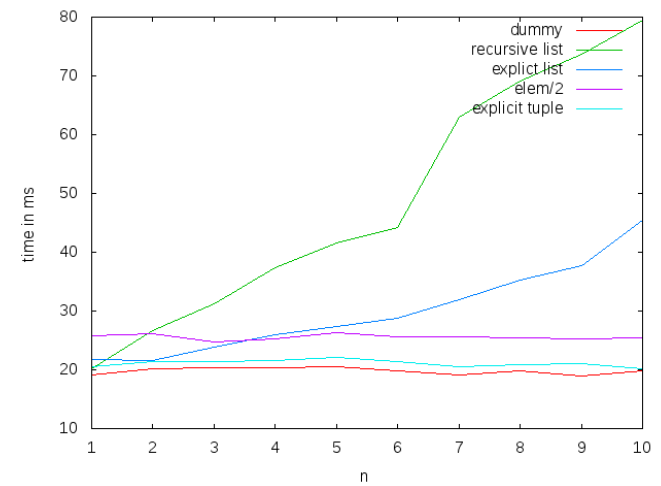
```
def nth(1, [r|_]) do r end
def nth(n, [_|t]) do nth(n-1, t) end
```

6 / 1

- `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)

7 / 1

Benchmark different versions of n'th:



8 / 1

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.*

9 / 1

How do we implement a *queue*?

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

def remove(queue) do
  :
end
```

10 / 1

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

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

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

11 / 1

How do we represent a binary tree?

How do we represent a leaf node?

```
{:leaf, value}
```

How do we represent a branch node?

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

How do we represent an empty tree?

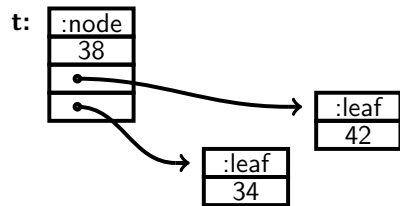
```
:nil
```

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

12 / 1

## what is happening

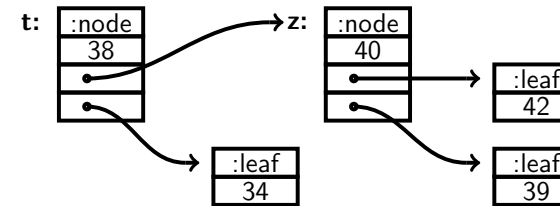
```
t = {:node, 38, {:leaf, 42}, {:leaf, 34}}
```



13 / 1

## what is happening

```
z = {:node, 40, {:leaf, 42}, {:leaf, 39}},  
t = {:node, 38, z, {:leaf, 34}}
```



14 / 1

## search a tree

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

```
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 ... do  
    :yes -> :yes  
    :no -> ....  
  end  
end
```

Navigation icons: back, forward, search, etc.

15 / 1

## ordered tree

How is the situation changed if the tree is ordered?

```
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, v, left, right}) do  
  if n < v do  
    ...  
  else  
    ...  
  end  
end
```

What is the asymptotic time complexity of this function?

16 / 1

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

```
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`.

17/1

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

```
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
    ...
  else
    ...
  end
end
```

18/1

```
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
```

19/1

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

```
def insert(key, value, :nil) do ... end
def insert(key, value, {:node, k, v, left, right}) do
  if key < k do
    ...
    true ->
    ...
  end
end
```

20/1

## delete

How do we implement `delete(item, tree)` (assuming it does exist)?

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

21 / 1

## 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

23 / 1

## deleting a element

Algorithm first - then implement.

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

22 / 1