Trees

Johan Montelius

KTH

VT23

compound structures

Compound data structures in Elixir:

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

lists

We could implement lists using tuples:

```
  :foo,  :foo, :bar,  :foo, :bar, :zot,
```

```
[:foo | [:bar | [:zot | [] ] ] ]
```

```
[:foo, :bar, :zot]
```


lists

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:

```
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
```


nth element

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
```


nth element

- `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)

n'th benchmark

Benchmark different versions of n'th:

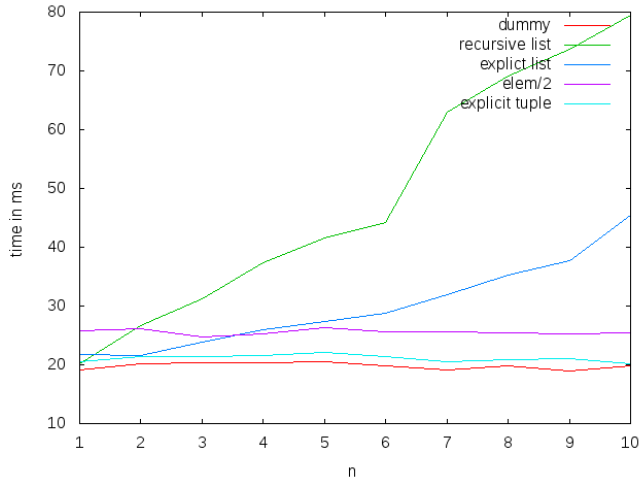


Figure: Execution time in ms of 1000 000 calls

update an item in a list

```
update(lst, pos, itm)
```

```
def update([_|rest], 0, itm) do [itm|rest] end
def update([first|rest], n, itm) do
   [first|update(rest, n-1, itm)]
end
```

this is an expensive operation

update an item in a tuple

```
put_elem(tuple, pos, itm)
```

```
> put_elem({:a, :b, :c}, 1, :x)
{:a, :x, :c}
```

this is an expensive operation

what about queues

How do we implement a *queue*?

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

end

def remove(queue) do
   :

end
```


an ok queue

```
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
```

trees

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}`

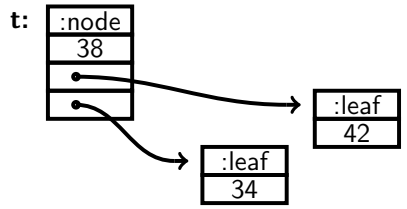
How do we represent an empty tree?

`:nil`

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

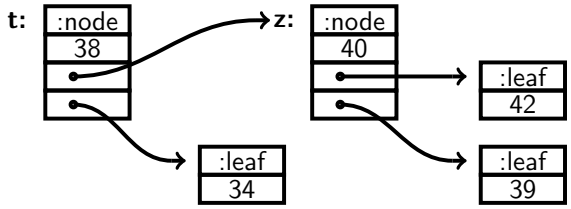

what is happening

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



what is happening

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



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
```

What is the asymptotic time complexity of this function?

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?

key-value look-up

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`.

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?

```
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
```


modify an element

```
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)?

```
def insert(key, value, :nil) do ...    end

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


delete

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

```
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
```


deleting an element

Algorithm first - then implement.

```
def 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