Pointer Structures

Troels Henriksen (athas@sigkill.dk)

DIKU
University of Copenhagen

Arrays and Records

Linked Lists

Trees

Representing arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how is this stored in memory?

Representing arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how is this stored in memory?

0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	`i32`				`i8`	...

Problem?

Representing arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how is this stored in memory?

0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	`i32`				`i8`	...

Problem? Unaligned accesses.

Representing arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how is this stored in memory?

0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	`i32`				`i8`	…

Problem? Unaligned accesses.

0	1	2	3	4	5	6	7	8	9	10
`i32`				`i8`	*unused*			`i32`		…

Problem?

Representing arrays of tuples

Consider arrays of type `[](i32, i8)`. Since an `i32` is four bytes and a `i8` is one byte, how is this stored in memory?

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th></tr>
<tr><td colspan="4"><code>i32</code></td><td><code>i8</code></td><td colspan="4"><code>i32</code></td><td><code>i8</code></td><td>...</td></tr>
</table>

Problem? Unaligned accesses.

<table>
<tr><th>0</th><th>1</th><th>2</th><th>3</th><th>4</th><th>5</th><th>6</th><th>7</th><th>8</th><th>9</th><th>10</th></tr>
<tr><td colspan="4"><code>i32</code></td><td><code>i8</code></td><td colspan="3">unused</td><td colspan="3"><code>i32</code> ...</td></tr>
</table>

Problem? Waste of memory.

Tuples of arrays

Representation

An array `[](t1, t2, t3...)` is represented in memory as `([]t1, []t2, []t3...)`, i.e. as *multiple arrays*, each containing only primitive values.

0	1	2	3	4	5	6	7	8	9	10
`i32`				`i32`				`i32`		...
`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	`i8`	...

- Common (and crucial) optimisation.
- Called "struct of arrays" in legacy languages.
- Automatically done by the Futhark compiler.

”Unzipped” SOACs

Instead of

```
let tmp = map (\(x,y) -> (x-1, y+1))
              (zip xs ys)
let (xs, ys) = unzip xs_ys’
```

could we write

```
let (xs, ys) = map (\x y -> (x-1, y+1)) xs ys
```

?

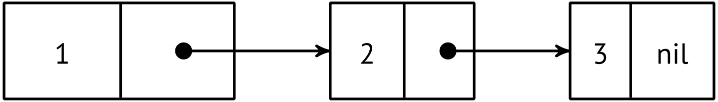
- Annoying to give type rules, **but this is actually what the compiler does internally.**
- **Isomorphic to source language**, but this form is easier to manipulate in a compiler.
- This will be relevant in a later lecture.

Arrays and Records

Linked Lists

Trees

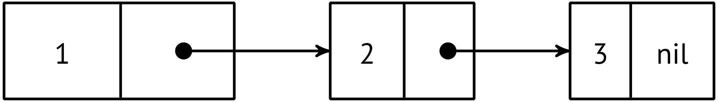
Standard representation: Cells with Pointers




```haskell
data List a = Nil
            | Cons a (List a)

-- values look like
Cons 1 (Cons 2 (Cons 3 Nil))
```

Standard representation: Cells with Pointers




```haskell
data List a = Nil
            | Cons a (List a)

-- values look like
Cons 1 (Cons 2 (Cons 3 Nil))
```

Challenges for Data Parallelism

- Many languages do not support recursive data structures.
- Traversing a list is **sequential**.

Let us try to address these problems.

An array encoding of the link structure for list with n nodes

The *Successor Array* S of length n

$S[i]$ denotes index of successor for node i, with $S[i] = n$ indicating last element.

The *Value Array* V of length n

$V[i]$ denotes the value of node i.

Example

$$S = [4, 0, 5, 2, 3]$$

encodes a list with nodes stored in order

$$[1, 0, 4, 3, 2]$$

meaning the first node is at index 1, second node at index 0, etc.

Questions

$$S = [4, 0, 5, 2, 3]$$

- Given S, how do we find the head node?

Questions

$$S = [4, 0, 5, 2, 3]$$

- Given S, how do we find the head node?
  - Must find the node *not* marked as successor by any other node.
  - Can be done with `scatter` followed by `reduce`.

Questions

$$S = [4, 0, 5, 2, 3]$$

- Given S, how do we find the head node?
  - Must find the node *not* marked as successor by any other node.
  - Can be done with `scatter` followed by `reduce`.
- How do we find the last node?

Questions

$$S = [4, 0, 5, 2, 3]$$

- Given S, how do we find the head node?
  - Must find the node *not* marked as successor by any other node.
  - Can be done with `scatter` followed by `reduce`.
- How do we find the last node?
  - Find the node i where $S[i] = n$.
  - Just a `reduce`.

Questions

$$S = [4, 0, 5, 2, 3]$$

- Given S, how do we find the head node?
  - Must find the node *not* marked as successor by any other node.
  - Can be done with `scatter` followed by `reduce`.
- How do we find the last node?
  - Find the node i where $S[i] = n$.
  - Just a `reduce`.

What about all the other nodes?

List Ranking

The List Ranking Problem

Determine the rank of each element in list, such that first element has rank 1, second has rank 2, etc.

List Ranking

The List Ranking Problem

Determine the rank of each element in list, such that first element has rank 1, second has rank 2, etc.

- We will actually study a variant, where the *last* element has rank 1.
- Can see it as "distance from end".
- Are we cheating? **Does that make the problem harder or easier?**

Wyllie's List Ranking

Each list element i has a *successor pointer* $S[v]$, and at each time step we update

$$S[v] \leftarrow \begin{cases} S[S[v]] & \text{when } S[v] \neq n \\ S[v] \end{cases}$$

- Distance covered by pointer doubles for each time step.
- After $\lceil \log(n) \rceil$ steps each $S[i]$ is to n (end of list).
- Total work is $O(n \log(n))$ and span $O(\log(n))$.

Wylie's List Ranking

Each list element i has a *successor pointer* $S[v]$, and at each time step we update

$$S[v] \leftarrow \begin{cases} S[S[v]] & \text{when } S[v] \neq n \\ S[v] \end{cases}$$

- Distance covered by pointer doubles for each time step.
- After $\lceil \log(n) \rceil$ steps each $S[i]$ is to n (end of list).
- Total work is $O(n \log(n))$ and span $O(\log(n))$.

Can compute rank by initially setting $R[i] = 1$ and then

$$R[v] \leftarrow \begin{cases} R[i] = R[i] + R[S[i]] & \text{when } S[v] \neq n \\ R[i] \end{cases}$$

in each step.

In Futhark

```
def step [n] (R: [n]i32) (S: [n]i64) =
  let f i = if S[i] == n
            then (R[i], S[i])
            else (R[i] + R[S[i]], S[S[i]])
  in unzip (tabulate n f)

def wyllie [n] (S: [n]i64) : [n]i32 =
  let R = replicate n 1
  let (R,_) = loop (R, S) for _i < 64 - i64.clz n do
                 step R S
  in R
```

Is this work efficient?

In Futhark

```
def step [n] (R: [n]i32) (S: [n]i64) =
  let f i = if S[i] == n
            then (R[i], S[i])
            else (R[i] + R[S[i]], S[S[i]])
  in unzip (tabulate n f)

def wyllie [n] (S: [n]i64) : [n]i32 =
  let R = replicate n 1
  let (R,_) = loop (R, S) for _i < 64 - i64.clz n do
                step R S
  in R
```

Is this work efficient?

- **No**, a sequential implementation has work $O(n)$.
- Reason is that we keep inspecting nodes that have already finished.
- Work efficient algorithms exist, but are more complicated.

Converting list to array

```
def list_to_array [n] 'a (V: [n]a) (S: [n]i64) =
  scatter (copy V) (map (\i -> n - i64.i32 i) (wyllie S)) V
```

Converting list to array

```
def list_to_array [n] 'a (V: [n]a) (S: [n]i64) =
  scatter (copy V) (map (\i -> n - i64.i32 i) (wyllie S)) V
```

- To `scan` or `reduce` a list we could convert to array, then use array operations.
- **Is there another option?**

List ranking, but now a scan

```
def wyllie_scan_step [n] 'a (op: a -> a -> a)
                            (V: [n]a) (S: [n]i64) =
  let f i = if S[i] == n
            then (V[i], S[i])
            else (V[i] `op` V[S[i]], S[S[i]])
  in unzip (tabulate n f)

def wyllie_scan [n] 'a (op: a -> a -> a)
                       (V: [n]a) (S: [n]i64) =
  let (V,_) = loop (V, S) for _i < 64 - i64.clz n do
                wyllie_scan_step op V S
  in V
```

Packing it all up

What might it look like to actually construct a library of list operations?

```
type list [n] 'a = { S: [n]i64
                   , V: [n]a
                   , head: i64
                   , last: i64
                   }

def head [n] 'a (l: list [n] a) : a = l.V[l.head]

def last [n] 'a (l: list [n] a) : a = l.V[l.last]
```

From array to list

From array to list

```
def from_array 'a [n] (V: [n]a) : list [n] a =
  { S = map (+1) (iota n)
  , V
  , head = 0
  , last = n-1
  }
```

Reversing a list

Reversing a list

```
def rev [n] 'a (l: list [n] a) =
  let f i = (l.S[i], i)
  let (is, vs) = unzip (tabulate n f)
  in l with S = scatter (replicate n n) is vs
       with head = l.last
       with last = l.head
```

Scans

```
def scan [n] 'a (op: a -> a -> a) (l: list [n] a) =
  let l' = rev l
  in l with V = (wyllie_scan op l'.V l'.S)
```

What about reductions?

```
def reduce [n] 'a (op: a -> a -> a)
                  (_ne: a) (l: list [n] a) =
  last (scan op l)
```

Note: *list* `last`, not the builtin array `last`.

What about reductions?

```
def reduce [n] 'a (op: a -> a -> a)
                  (_ne: a) (l: list [n] a) =
  last (scan op l)
```

Note: *list* `last`, not the builtin array `last`.
We can do better in the *commutative* case.

```
def reduce_comm [n] 'a (op: a -> a -> a)
                       (ne: a) (l: list [n] a) =
  reduce_comm op ne l.V -- Array reduction.
```

Concatenation

Concatenation

```
def (++) [n] [m] 'a (x: list [n] a) (y: list [m] a) =
  { S = if n == 0 || m == 0 then x.S ++ y.S
        else (copy x.S with [x.last] = n + y.head)
             ++
             map (\i -> i+n) y.S
  , V = x.V ++ y.V
  , head = x.head
  , last = y.last + n
  }
```

Take?

```
val take [n] 'a (i: i64) (l: list [n] a) : list [i] a
```

Exercise for the reader.

Arrays and Records

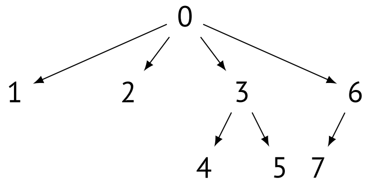
Linked Lists

Trees

Trees are also pointer structures

- Optimal representation depends on what we want to do with them.
- The "parent pointer representation" will be our main object of study, but we will look at a few others, too.
- As with lists, the main problem is to find a **linearized representation.**

Semi-linear representations



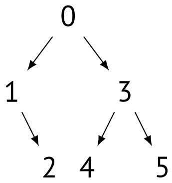
- Could represent this as a linked list (or irregular array) of *levels*.

$$
\begin{array}{c}
0 \\
\downarrow \\
[2, 3, 0, 0] \\
\downarrow \\
[4, 0, 5]
\end{array}
$$

- How do we know the parent-child relationship?
  - Can add ancillary structure.
  - But this is still *recursive*, and we can't have that.

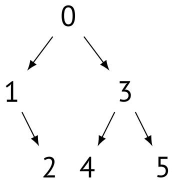
A representation for binary trees

- *Child vectors* L, R where $L[i]$ is the left child of i and $R[i]$ is the right child.
- $L[i] = -1$ or $R[i] = -1$ denotes no such child.



A representation for binary trees

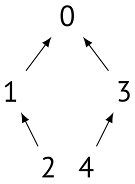
- *Child vectors* L, R where $L[i]$ is the left child of i and $R[i]$ is the right child.
- $L[i] = -1$ or $R[i] = -1$ denotes no such child.



$$L = [\quad 1, \quad -1, \quad -1, \quad 4, \quad -1, \quad -1, \quad -1 \quad]$$
$$R = [\quad 3, \quad 2, \quad -1, \quad 5, \quad -1, \quad -1, \quad -1 \quad]$$

This is an OK representation for many purposes, but it cannot represent *n*-ary trees.

What if we flip the pointers?

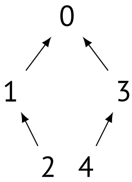


Parent Vector P

- $P[i]$ is the parent of i.
- Root is its own parent: $P[i] = i$.

Either siblings are considered unordered, or they are ordered by their index.

What if we flip the pointers?



Parent Vector P

- $P[i]$ is the parent of i.
- Root is its own parent: $P[i] = i$.

Either siblings are considered unordered, or they are ordered by their index.

Converting from child vectors to parent vector

$$L[i] = j \vee R[i] = j \Rightarrow P[j] = i$$

A more complicated tree

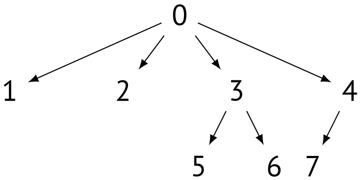
Can you draw this tree?

$$P = [0, 0, 0, 0, 0, 3, 3, 4]$$

A more complicated tree

Can you draw this tree?

$$P = [0, 0, 0, 0, 0, 3, 3, 4]$$



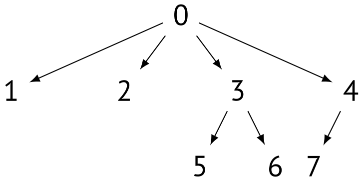
What about this one?

$$P = [1, 7, 7, 2, 2, 7, 7, 7]$$

A more complicated tree

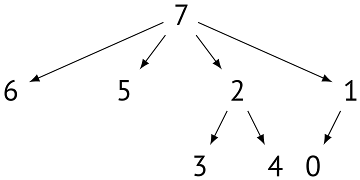
Can you draw this tree?

$$P = [0, 0, 0, 0, 0, 3, 3, 4]$$



What about this one?

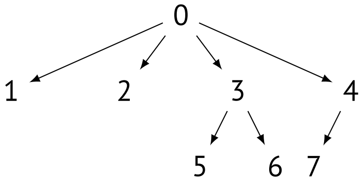
$$P = [1, 7, 7, 2, 2, 7, 7, 7]$$



A more complicated tree

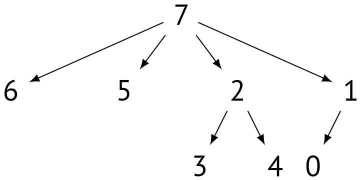
Can you draw this tree?

$$P = [0, 0, 0, 0, 0, 3, 3, 4]$$



What about this one?

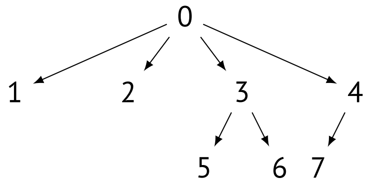
$$P = [1, 7, 7, 2, 2, 7, 7, 7]$$



- The same tree can have different parent vectors.
- **Element order (almost) does not matter!**

Depth Vectors

For a tree $P = [0, 0, 0, 0, 0, 3, 3, 4]$



we can state the distance of each node from the root as

$$D = [0, 1, 1, 1, 1, 2, 2, 2]$$

- Mononotically increasing here, but that depends on node ordering.

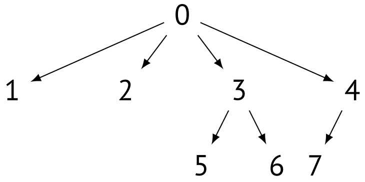
Depth Vector D

$D[i]$ is the distance of node i from the root.

Using depth vector as representation

By assuming a specific node ordering (e.g. preorder traversal) the depth vector is an unambiguous representation of the tree.

For a tree



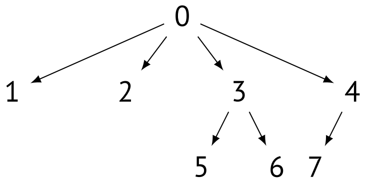
the preorder traversal is

$$[0, 1, 2, 3, 5, 6, 4, 7]$$

and so

$$D = [0, 1, 1, 1, 2, 2, 1, 2]$$

Constructing depth vector from parent vector

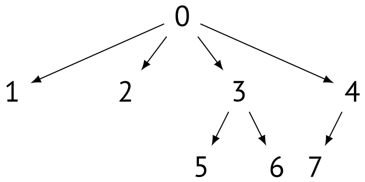


$$\begin{array}{cc} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ P = & [0, 0, 0, 0, 0, 3, 3, 4] \end{array}$$

Idea:

- The parent vector encodes *multiple linked lists* from leaves to the root.
  - $[1, 0]$
  - $[2, 0]$
  - $[5, 3, 0]$
  - $[6, 3, 0]$
  - $[7, 4, 0]$

Constructing depth vector from parent vector

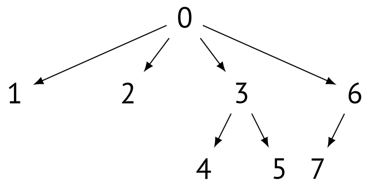


$$P = \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ [0, 0, 0, 0, 0, 3, 3, 4] \end{array}$$

Idea:

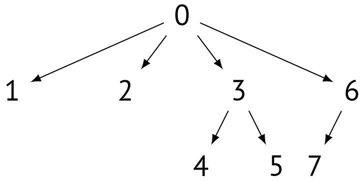
- The parent vector encodes *multiple linked lists* from leaves to the root.
  - $[1, 0]$
  - $[2, 0]$
  - $[5, 3, 0]$
  - $[6, 3, 0]$
  - $[7, 4, 0]$
- Do essentially list ranking on each, simultaneously!
- Work $O(n \log n)$, span $O(\log n)$.

Constructing parent vector from depth vector, assuming preorder



$$\begin{array}{cc} & \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ D = & [0, 1, 1, 1, 2, 2, 1, 2] \end{array}$$

Constructing parent vector from depth vector, assuming preorder



$$D = \begin{array}{l} \quad\;\; \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ [\begin{array}{cccccccc} 0, & 1, & 1, & 1, & 2, & 2, & 1, & 2 \end{array}] \end{array}$$

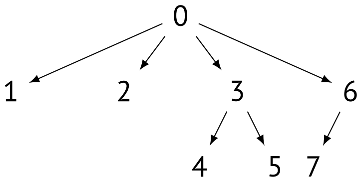
Idea

- For each node, linear search *left* until you find a node with lower depth.
  - That is your parent.
- Potentially costly *sequential* search.
  - Span $O(n)$ in worst case.
- **Optimisation idea**.
  - Pair depth vector with original index.
  - Sort in an appropriate way.
  - Use binary search.

From traversal vector to depth vector

A traversal vector describes a preorder traversal of tree

- Contains elements 1 and -1.
  - 1 descends into a new node.
  - -1 returns to parent.

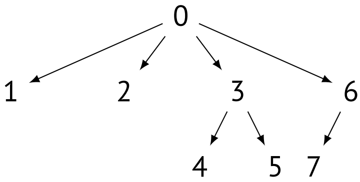


$$[1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1]$$

From traversal vector to depth vector

A traversal vector describes a preorder traversal of tree

- Contains elements 1 and -1.
  - 1 descends into a new node.
  - -1 returns to parent.



$[1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1]$

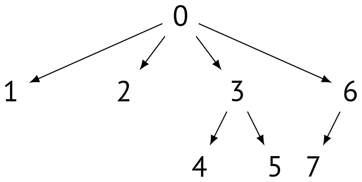
What happens if we take the prefix sum?

$[1, 0, 1, 0, 1, 2, 1, 2, 1, 0, 1, 2]$

From traversal vector to depth vector

A traversal vector describes a preorder traversal of tree

- Contains elements 1 and -1.
  - 1 descends into a new node.
  - -1 returns to parent.



$[1, -1, 1, -1, 1, 1, -1, 1, -1, -1, 1, 1]$

What happens if we take the prefix sum?

$[1, 0, 1, 0, 1, 2, 1, 2, 1, 0, 1, 2]$

It is (almost) the depth vector!

- Indexed from 1 instead of 0.
- Must remove the elements corresponding to -1 in traversal.

Summary of trees: the parent pointer representation

n-node tree is represented by the these n-element vectors.

P: parent vector.

- $P[i]$ is the parent of node i.
- For the root node, $P[i] = i$.

D: depth vector.

- $D[i]$ is the distance from root to node i.
- For the root node, $D[i] = 0$.
- For other nodes, $D[i] = D[P[i]] + 1$.

V: value vector.

- $V[i]$ is the value of node i.

- P can be computed from D.
- D can be computed from P.
- ...but usually convenient to have both.

Summary

- Recursive pointer structures are not natural in data parallel languages, but can work well if we are careful.
- `https://github.com/diku-dk/containers/blob/main/lib/github.com/diku-dk/containers/list.fut`
- An interesting DPP project might be to implement various pointer structures (lists, trees, graphs) in a data parallel language and see what their performance is like.