Unit 4.1

review

application of sequences

- MCSS
  - scons
  - divide & conquer

  } $O(n)$ wk, $O(\log n)$ span — optimal

- BFS on graphs

  } $O(m)$ wk, $O(\text{diameter})$ span

inject : $O(\log m)$ span

Binary Search Trees (sets & dictionaries)

a BST : each node is either internal or leaf

internal nodes have keys (whatever, vals)

each node has exactly 2 or 0 children

```
datatype α tree = Node(α tree, α, α tree)
                | Leaf
```

find : $\alpha \times \alpha$ tree $\to$ bool

insert : $\alpha \times \alpha$ tree $\to \alpha$ tree

delete : $\alpha \times \alpha$ tree $\to \alpha$ tree

"in bulk"

intersect : α tree $\times \alpha$ tree $\to \alpha$ tree

union : α tree $\times \alpha$ tree $\to \alpha$ tree

difference : α tree $\times \alpha$ tree $\to \alpha$ tree

Sequential BSTs : Balancing

- red-black trees
- AVL trees
- weight-balanced trees
- treaps (probabilistically balanced trees)

split : α tree $\times \alpha \to$ bool $\times \alpha$ tree $\times \alpha$ tree

join : α tree $\times \alpha$ tree $\to \alpha$ tree

---

split, join minimalist implementations

wk = $O(\log n)$ `find k t = let (found, l, r) = split t k in found`

wk = $O(\log n)$ `insert k t = let (found, l, r) = split t k in` if found then t else `join(l, join(singleton(k), r))`

$l < k < r$

$l <$ singleton(k) $< r$

delete k t = let (_, l, r) = split t k in join (l, r) wk = O(lg n)

intersection t u = case (t, u) of
  (leaf, u) ⇒ leaf
  (t, leaf) ⇒ leaf
  (Node (l, k, r), u) ⇒ let (fly, l2, r2) = split k, u
    (l, r) = intersect (l, l2) || intersect (r, r2)
  in if flag then join M (l, k, r) else join (l, r)

union t u = case (t, u) of
  (leaf, u) ⇒ u
  (t, leaf) ⇒ t
  (Node (l, k, r), u) ⇒ let (_, l2, r2) = split k, u
    (l, r) = union (l, l2) || union (r, r2)
  in join M (l, k, r)

R we outsource all our balancing to split & join

wk where m is size of tree on left, n is size of tree on right

we assume $m \le n$

$$wk(m,n) = 2wk\left(\frac{m}{2}, \frac{n}{2}\right) + O(\lg(n+m))$$ ⟨solve the recurrence⟩

$$wk(m,n) = O\left(m \lg\left(\frac{n}{m}\right)\right)$$ ⟨if $n \approx m$ (approximately the same)⟩

$$wk(n,n) = O(n)$$

UNIT 4,3 continued

union, intersect,

if n,m = size of each tree

$$Wk(n,m) = O\left(n \log \frac{n}{m}\right)$$

$$Spn(n,m) = O(\log^2(n+m))$$

Summary

- sequential
  - find
  - insert
  - delete
- parallel
  - intersection
  - union
  - difference

all in terms of: split, join, joinM → lgn Wk, lgn Spn

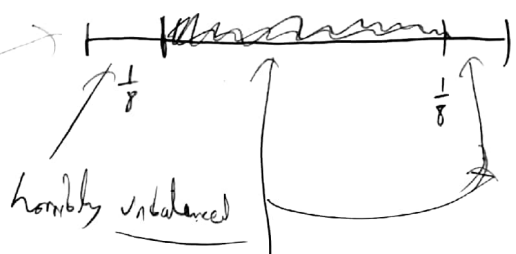
"tenary ops"

Balancing

① randomized technique

- consider the permutations of the list

let that line be that space



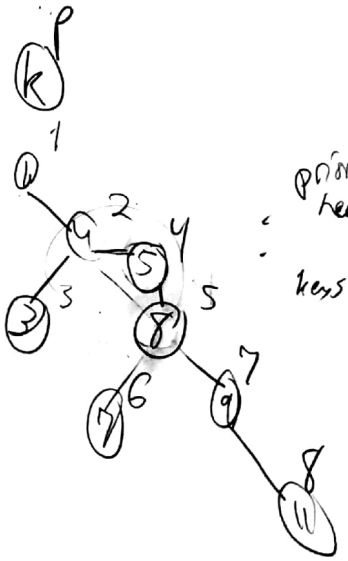
you have "3/4" probability of getting a balanced tree (each attempt)

Treap

priority	1	2	3	4	5	6	7	8
keys	3	1	4	5	7	8	9	11

- take an ordering of priorities, arbitrarily

priority	3	1	2	4	6	5	7	8
keys	1	4	3	5	8	7	9	11



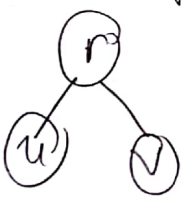
priorities are heap-ordered

keys are tree-ordered

VMST

4.4 `datatype α treap = Node of α treap × (α × int) × α treap | Leaf`

p(v) := "priority of key v"



$p(r) \leq p(u)$

$p(r) < p(v)$

`fun singleton k =` let p be random() in `Node(Leaf, (k,p), Leaf)`

`fun split t k = case t of`

`Leaf ⇒ (false, Leaf, Leaf)` = $O(\lg n)$ w high probability (when n is large)

`Node(l, (kk,p), r)` ⇒ if `k<kk`

then let `(found, ll, rr)` be `split(l, k)`

in `(found, ll, Node(rr, (kk,p), r))` (ll < k < rr)

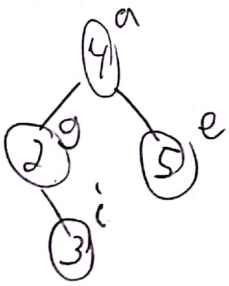
else if `k=kk` then `(true, l, r)`

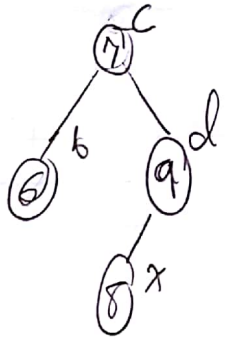
else if `k>kk`

then let `(found, ll, rr)` be `split(r, k)`

in `(found, Node(l, (kk,p), ll), rr)`

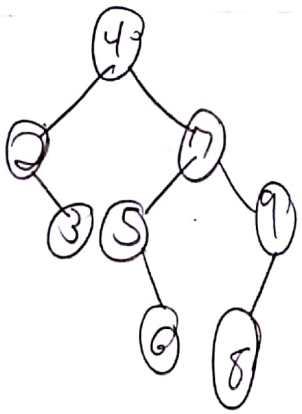
E





4, 7, 5, 2, 6, 3, 9, 8

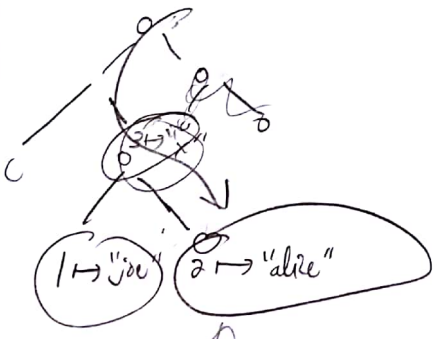
a c e g h i l x



Unit 4, 5

```
fun join t u =
    case (t, u) of (leaf, y) ⇒ u
                   (t, leaf) ⇒ t
                   (Node(lt, (kt, pt), rt), Node(lu, (ku, pu), ru))
                     ⇒ if pt < pu
                       then Node(lt, (kt, pt), join rt u)
                       else Node(join t lu, (ku, pu), ru)
```

Augmentation



"fifth person" 3 length 3

saying "the fifth person" should return a path less than 5

select query — k^{th} element in the sorted order of key in $\log n$ work.

rank — given a key k find the # of nodes

y 3.1

Breadth-first Search

Goals

1. sequential semantics (purely functional)
2. cost model & provable bound
3. work efficiency
4. low depth/span
5. As simple as sequential
6. Sequential as a side effect

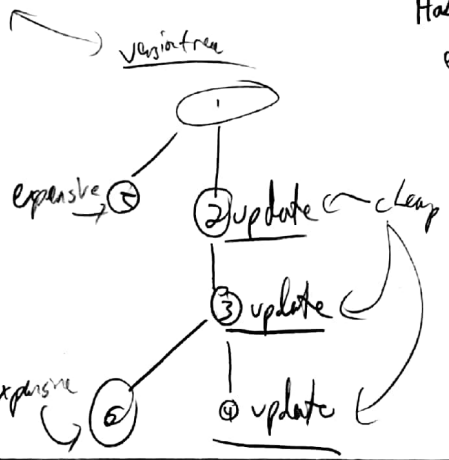
problem - pure array

changes means copying the whole thing (immutable) ≠ in laziness,

1. persistence "requires" copying
2. keep a history of changes

Haskell's cost semantics are impure even though Haskell is pure

	leaf		interior	
	wk	Spn	wk	Spn
subscript	O(1)	—	O(lgn)	O(lgn)
update	O(1)		O(n)	O(lgn)
inject A n	O(m/l)	O(1)	O(m/l+n)	O(lgn)



$\delta : l \to \{+, -\}$ where l is labels, + means leaf, − means interior

a store δ is some memory region that can be passed around,

pure array

$$\delta ; e \to (\delta', v, w, d) \qquad \frac{A = (a, l) \quad \delta[l \to +] \text{ generate } l'}{\delta, \text{update}(A, v) \longrightarrow \delta[l \to -, l' \to +], (\text{update}(a, v), l')}$$

database w history

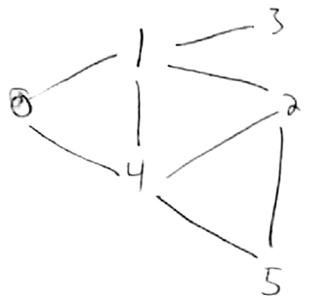
timestamps ↕ value w timestamps, do binary search on timestamps

like a log

← guarantee that total # of versions is linear

Guy 3.2

BFS



first search for 1 lvl away from start,
then 2,
then 3,
etc.
for unweighted shortest path,

d = diameter/longest path
m = |Edges|
and n = |vertices|

goal $wk(BFS) = O(m)$
$Spn(BFS) = O(d \lg n)$

$d=0$: $F = \{0\}$ ⟼ $d=1$: $F\{1,4\}$ ⟼ $d=2$: $\{3,2,5\} = F$

frontier

① Repeat over levels F, P

② every member of F points back to its parent, forming a tree P

③ update V   ④ remove duplicates

all neighbors of 0, all neighbors of 1

G = ⟨⟨1,4⟩, ⟨0,2,3,4⟩, ⟨ ...⟩

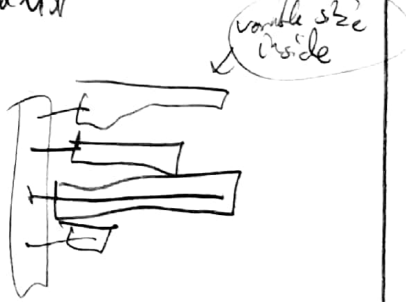
d=1: P = 0 ← 1, 0 ← 4 , d=2: 0 ← 1 ← 3, 0 ← 4 ← 2, 4 ← 5 = P, etc.

---

incidence Mtrx array², where but you have to fill in zeros

adjacency Mtrx array ∘ cons — parallel can't do linked list

array², which optimizes over sparseness,



d=2

F = ⟨1,4⟩ p = ⟨0, 0, -1, -1, 0, -1⟩ (indices 0 1 2 3 4 5) ← haven't visited; parentage relation

d=3

F = ⟨2,3,5⟩ p = ⟨0, 0, 1, 1, 0, 4⟩

2 needs a tie breaker which has to do w implementation.

$N = \text{flatten}(\text{Map}(\lambda v.\, \text{tag}(G[v], v))\ F)$

⟼ ⟨(0,1), (2,1), (3,1), (4,1), (0,4), (1,4), ...⟩ ← tagged w 1 or 4.

helper $\text{tag}(S, v) = \text{Map}(\lambda x.\,(x, v))\ S$

$N' = \text{filter}(\lambda(u,v).\, P[v] = -1)\ N$
$P' = \text{inject}\ P\ N'$

GV 5.1

Dynamic programming

functional & parallel
recursion w sharing

recipe: ① recursive solution to problem (sharing)
② Calculate the span
③ Calculate wk → identify unique calls
↘ determine costs of each

E fib(n) =
if $n \leq 1$ then 1
else $fib(n-2) + fib(n-1)$
} $Span = O(n)$
because sharing
$wk = (\text{\# unique args}) \times (O(1)) = O(n)$

Minimum edit distance (MED) String Char

Given two strings s,t how many inserts into s, deletes from t needed, best case

```
fun MED s t = let
    fun MED' i j be case i j of (0,_) ⇒ j
                                (_,0) ⇒ i
                                (  ) ⇒ if S[i-1] == T[j-1] then MED'(i-1, j-1)
                                       else 1 + Min (MED(i-1,j), MED(i, j-1))
    in MED(|S|, |T|) end
```

R this is exponential time w/o sharing

size of S times size of T

$wk(MED) =$ # of unique arguments $O(|S||T|)$ ⊗ cost of work per call $O(|S||T|) \to O((|S||T|)^2)$

$Span(MED) = O(|S| + |T|)$

E subset sum(SS) (like unweighted knapsack)

Given a set of nats S and a $k \in \mathbb{N}$ is there a subset $X \subseteq S$ s.t. $\left(\sum_{a \in X} a\right) = k$

R for arbitrary k this is NP hard, but if you bind k to a polynomial

```
fun SS (s,k) = let
    fun SS'(i,k) be case (i,k) of (_,0) ⇒ true
                                  (0,_) ⇒ false
                                  (_,_) ⇒ if S[i-1] > k then SS'(i-1, k)
                                          else SS'(i-1, k-S[i-1]) or SS'(i-1, k)
    in SS'(|S|, k) end
```

$Span(SS) = O(|S|)$

$wk(SS) = \begin{cases} \text{\# of unique args} = O(|S| \cdot k) \\ \text{cost of work per call} = O(1) \end{cases}$ ⊗ $\to O(|S| \cdot k)$

parallelism $= O(k)$

Fun 5.2

Longest increasing subsequence (LIS) Subseq of A = subset of A's elements in same order

$\langle 10, 22, 9, 33, 21, 50, 41, 60, 80 \rangle \mapsto_{LIS} 6$

will consume list from end to start

```
fun LIS S = let
    fun LIS'(i,n)
        case (i,n) of
            (0,_) => 0
          | _ => if (S[i-1] > n) then LIS'(i-1, n)
                 else max(1 + LIS'(i-1, S[i-1]), LIS'(i-1, n))
    in LIS'(|S|, ∞)
```

$Spn(LIS) = O(|S|)$

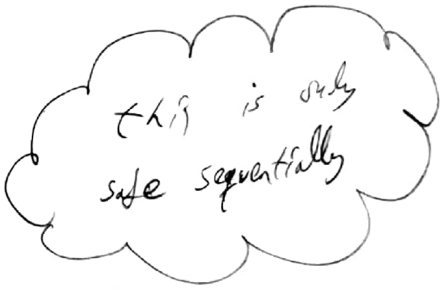
$Wk(LIS) = \begin{pmatrix}\#\text{ of}\\ \text{recursive}\\ \text{calls}\end{pmatrix}\begin{pmatrix}\text{cost}\\ \text{per}\\ \text{call}\end{pmatrix} = \left(O(|S|^2)\right)\left(O(1)\right) = O(|S|^2)$

How to memoize table equipped w find & insert of $O(1)$ (expectation via probability)

"table.find, table.insert, table.new()"

where a = arg type
b = result type

memoize : $((a \to b) \to (a \to b)) \to (a \to b)$



Example

```
val fib = memoize(
    fn fib' n => if (n<=1) then 1 else fib'(n-1) + fib'(n-2))
```

```
fun memoize f = let
    val cache = ref(table.new())
    fun mem arg = case table.find(!cache) arg
                    Some r => r
                  | None => let val r be f mem arg
                            val _ be cache := Table.insert(!cache, (arg, r))
                            in r end.
in mem end
```

wrap mutable cell around cache

just wrap any fun from today in this.

goal : make table safe for concurrency

response to a find : empty | busy | full

recurrence $W(n) = 4\left(W\left(\frac{n}{2}\right)\right) + O(n)$

solves to $= O(n^2)$

$$Spn(n) = 3\ Spn\left(\frac{n}{2}\right) + O(1)$$

$$= 3^{\lg n} = n^{\lg(3)} \approx n^{1.4}$$

$$\text{parallelism} = \frac{n^2}{n^{\lg(3)}} \approx n^{\frac{2}{3}}$$

R

program synthesis : takes a PLT level abstraction and converts it to Divide & Conquer.