Tree Search & Sorting

15-150 M21

Lecture 0625
25 June 2021

0 Tree Search

0625.0 (polytreeDefn.sml)

```
2 datatype 'a tree =
3   Empty | Node of 'a tree * 'a * 'a tree
```

0625.1 (search.sml)

```
2 (* INVARIANT: For all values p : t pred, p is
    total *)
3 type 'a pred = 'a -> bool
4 (*  isEven : int pred *)
5 fun isEven x = x mod 2 = 0
```

`search : 'a pred -> 'a tree -> 'a option`

REQUIRES: true

ENSURES:

$$\texttt{search p T} \cong \begin{cases} \texttt{SOME(z)} & \text{where } \texttt{z} \text{ is the top-leftmost element of } \texttt{T} \\ & \text{such that } \texttt{p(z)} \cong \texttt{true} \\ \texttt{NONE} & \text{if there is no such } \texttt{z} \end{cases}$$

Tradeoff:

- Optimal work (best-case & worst-case), bad span
- Bad work (best-case), optimal work (worst-case), optimal span

Can't get optimal work and optimal span unless we know more about where in the tree to look!

0625.2 (search.sml)

```
9  fun search p Empty = NONE
10   | search p (Node(L,x,R)) =
11       if p(x) then SOME x
12       else
13         (case search p L of
14               (SOME z) => SOME z
15             | _ => search p R)
```

ENSURES

$$\forall \texttt{p}, \texttt{T}, \quad \texttt{search p T} \cong \begin{cases} \texttt{SOME(z)} & \text{where z is the top-leftmost element of T} \\ & \text{such that p(z)} \cong \texttt{true} \\ \texttt{NONE} & \text{if there is no such z} \end{cases}$$

Check Your Understanding

Verify that this function has $O(n)$ work (in the worst case) when called on a balanced tree of size n.

0625.3 (search.sml)

```
19 fun search' p Empty = NONE
20   | search' p (Node(L,x,R)) =
21       if p(x) then SOME x
22       else
23         (case (search' p L, search' p R) of
24               (SOME z,_) => SOME z
25             |(_,SOME z) => SOME z
26             | _ => NONE)
```

Check Your Understanding Verify that this function has $O(d)$ span when called on a tree of depth d. If we assume the tree is balanced, $d \approx \log n$.

1 Searching Sorted Trees

General idea:

We'll store a bunch of **entries** in a tree. The entries will be indexed by **keys** and each entry will contain a value. We'll use the keys to look up the entries. The entries tree will be **sorted** by key, allowing for fast lookup.

Recall: `ord` type-and-invariant

Invariant Any value `cmp : t ord` is a comparison function.

0625.4 (bst.sml)

```
2 type 'a ord = 'a * 'a -> order
```

Making some concrete choices

0625.5 (bst.sml)

```
9  type key = string
10 type 'a entry = key * 'a
11 val keyCmp : key ord = String.compare
```

0625.6 (bst.sml)

```
74 type 'a bst = 'a entry tree
```

Invariant If `T : t bst`, then `T` is sorted by key (according to `keyCmp`)

`maxKey : 'a entry tree -> key option`
REQUIRES: true
ENSURES: `maxKey Empty` $\cong$ `NONE` and otherwise
`maxKey T` $\cong$ `SOME(k)`, where `k` is the `keyCmp`-largest key of any entry in `T`.

0625.7 (bst.sml)

```
21 fun maxKey (Empty :'a entry tree) = NONE
22   | maxKey (Node(L,(k,_),R)) : key option =
23     let fun max (NONE,Y) = Y
24           | max (X,NONE) = X
25           | max (SOME k,SOME k') =
26                 case keyCmp(k,k') of
27                       GREATER => SOME k
28                     | _ => SOME k'
29     in
30       max(SOME k,max(maxKey L,maxKey R))
31     end
```

0625.8 (bst.sml)

```
41 fun minKey (Empty :'a entry tree) = NONE
42   | minKey (Node(L,(k,_),R)) : key option =
43     let fun min (NONE,Y) = Y
44           | min (X,NONE) = X
45           | min (SOME k,SOME k') =
46                 case keyCmp(k,k') of
47                       GREATER => SOME k'
48                     | _ => SOME k
49     in
50       min(SOME k,min(minKey L,minKey R))
51     end
```

Sortedness

Defn. A value `T : t bst` is **sorted by key** (according to `keyCmp`) if either `T = Empty` or `T=Node(L,(k,v),R)` such that

- `L` and `R` are sorted by key
- `k` is greater than or equal to `maxKey L`, according to `keyCmp`
- `k` is less than or equal to `minKey R`, according to `keyCmp`

Invariant If `T : t bst`, then `T` is sorted by key (according to `keyCmp`)

0625.6 (bst.sml)

```
74 type 'a bst = 'a entry tree
```

Claim If `Node(L,x,R) : t bst`, then `L : t bst` and `R : t bst`

0625.9 (bst.sml)

```
55 val isEmpty = fn Empty => true | _ => false
56 val valOf : 'a option -> 'a = Option.valOf
57 fun leq(k,k') = keyCmp(k,k') <> GREATER
58 fun geq(k,k') = keyCmp(k,k') <> LESS
59 fun isSorted Empty = true
60   | isSorted (Node(L,(k,_),R)) =
61     (isSorted L) andalso (isSorted R)
62     andalso ( (isEmpty L)
63               orelse leq(valOf(maxKey L),k)
64             )
65     andalso ( (isEmpty R)
66               orelse geq(valOf(minKey R),k)
             )
```

Invariant If `T : t bst`, then `T` is sorted by key (according to `keyCmp`)

0625.10 (bst.sml)

```
78 fun asBST (T : 'a entry tree) : 'a bst =
79     let
80       val _ = (isSorted T)
81                 orelse raise Fail "Not sorted"
82     in
83       T
84     end
```

`sortedSearch : key -> 'a bst -> 'a option`

REQUIRES: true

ENSURES:

$$\texttt{sortedSearch k' T} \cong \begin{cases} \texttt{SOME(v)} & \text{for some entry (k,v) in T s.t.} \\ & \texttt{keyCmp(k',k)} \cong \texttt{EQUAL} \\ \texttt{NONE} & \text{if there is no such (k,v)} \end{cases}$$

0625.11 (bst.sml)

```
100 fun sortedSearch _ (Empty) : 'a option = NONE
101   | sortedSearch k' (Node(L,(k,v),R) : 'a bst)=
102       case keyCmp(k',k) of
103           LESS => sortedSearch k' (L : 'a bst)
104         | EQUAL => SOME(v)
105         | GREATER => sortedSearch k' R
```

0 Notion of size: number n of nodes in `T`.

0.5 Assumption: `keyCmp` is $O(1)$, and `T` is balanced.

1

$$W(0) = k$$
$$W(n) = k + W(n/2)$$

4

$$W(n) = \sum_{i=0}^{\log n} k \approx k \log n$$

5 $W(n)$ is $O(\log n)$

No opportunities for parallelism, so recurrence for span is the same, hence $S(n)$ also $O(\log n)$.

5-minute break

2 Sorting Trees

Idea: `msort`

The merge sort algorithm for sorting a list:

1. Split the list in half
2. Sort the two halves separately
3. Merge the sorted halves into a sorted whole

For trees: pretty much similar, except the tree is (almost) split into two halves already

`insert : 'a entry -> 'a bst -> 'a bst`
REQUIRES: true
ENSURES: `insert e T` evaluates to a BST `T'` containing all the entries of `T`, plus `e`

0625.12 (bst.sml)

```
109 fun insert (x : 'a entry) (Empty : 'a bst) =
110         Node(Empty,x,Empty)
111   | insert (k',v') (Node(L,(k,v),R)) =
112         case keyCmp(k',k) of
113             GREATER =>
114                 Node(L,(k,v),insert (k',v') R)
115           | _ =>
                Node(insert (k',v') L,(k,v),R)
```

`splitAt : key -> 'a bst -> 'a bst * 'a bst`
REQUIRES: true
ENSURES: `splitAt k' T` evaluates to `(A,B)` where `A` and `B` are BSTs such that

- Each entry of `T` is in either `A` or `B`
- `maxKey(A)` is less than or equal to `k'`
- `minKey(B)` is greater that or equal to `k'`

0625.13 (bst.sml)

```
120 fun  splitAt k' (Empty : 'a bst) = (Empty,Empty)
121   | splitAt k' (Node(L,(k,v),R))   =
122       case keyCmp(k',k) of
123         LESS => let
124                   val (t1,t2) = splitAt k' L
125                 in
126                   (t1,Node(t2,(k,v),R))
127                 end
128       | _ =>    let
129                   val (t1,t2) = splitAt k' R
130                 in
131                   (Node(L,(k,v),t1),t2)
                    end
```

0625.14 (bst.sml)

```
136 fun merge (Empty : 'a bst,T2 : 'a bst) = T2
137   | merge (Node(L1,(k,v),R1),T2) : 'a bst =
138       let
139           val (L2,R2) = splitAt k T2
140       in
141           Node(
142               merge (L1,L2),
143               (k,v),
144               merge (R1,R2))
145       end
```

0625.15 (bst.sml)

```
149 fun msort (Empty : 'a entry tree) = Empty
150   | msort (Node(L,x,R)) : 'a bst =
151       let
152           val (L',R') = (msort L, msort R)
153       in
154           insert x (merge (L',R'))
155       end
```

Check Your Understanding Verify that `msort` always produces a BST (a tree sorted by keys)

Check Your Understanding

Assuming `T` is balanced BST with n nodes, and `keyCmp` is $O(1)$, verify that

- `splitAt` has $O(\log n)$ work and span
- `insert` has $O(\log n)$ work and span

0 Notion of size: sum of sizes of `T1` and `T2`

0.5 Assumption: `keyCmp` is $O(1)$, and `T1` , `T2` are balanced BSTs of approximately the same size

1

$$S(0) = k$$
$$S(n) = k + S(n/2) + k' \log n$$

4

$$S(n) = \sum_{i=0}^{\log n} (k + k' \log n) \approx k \log n + k' (\log n)^2 \approx k' (\log n)^2$$

5 $S(n)$ is $O((\log n)^2)$

0 Notion of size: number n of nodes in `T`.

0.5 Assumption: `keyCmp` is $O(1)$, and `T` is balanced.

1

$$
\begin{aligned}
S(0) &= k \\
S(n) &= k + S(n/2) + k'(\log n)^2 + k''(\log n) \\
&\approx S(n/2) + k'(\log n)^2
\end{aligned}
$$

4

$$
S(n) = \sum_{i=0}^{\log n} k'(\log n)^2 \approx k'(\log n)^3
$$

5 $S(n)$ is $O((\log n)^3)$

Where's the lie?

Problem of Balancing

The lie in the previous analysis was this: `msort` doesn't always produce a *balanced* BST, violating the assumption we made when analyzing `merge`.

Solutions?

- Have a single "rebalance" function that turns a BST into a balanced BST, which we run on the output of `msort`
- Rewrite `insert` and `merge` to maintain balance as an invariant

We'll take the latter approach, but not for a couple weeks.

- Can optimize tree search for either (best-case) work or span, but need to know "where to look" in tree to optimize both/further
- Sorting the tree (perhaps by keys) allows us to get better work and span
- Can implement merge sort for trees, with much better span
- But we need the trees to self-balance in order to actually achieve the stated bounds

Next Time

- Continuation accumulators
- Control flow continuations
- Continuation Passing Style

Lecture ended here on 25 June 2021. You're not expected to know anything past here.

3 Lazy Combinator Tree Search

0625.3 (search.sml)

```
19 fun search' p Empty = NONE
20   | search' p (Node(L,x,R)) =
21       if p(x) then SOME x
22       else
23         (case (search' p L, search' p R) of
24               (SOME z,_) => SOME z
25             |(_,SOME z) => SOME z
26             | _ => NONE)
```

0625.16 (lazysearch.sml)

```
7  fun optOrelse (SOME x,_) = SOME x
8    | optOrelse (NONE,Y) = Y
9  infixr optOrelse
10
11 fun search (p:'a pred) Empty = NONE
12   | search p (Node(L,x,R) : 'a tree) =
13       if p(x) then SOME x else
14       (search p L) optOrelse (search p R)
```

This is the span-optimized version because both arguments to `optOrelse` will get evaluated, in parallel (assuming adequate processors).

What about the work-optimized version?

0625.2 (search.sml)

```
9  fun search p Empty = NONE
10   | search p (Node(L,x,R)) =
11       if p(x) then SOME x
12       else
13         (case search p L of
14               (SOME z) => SOME z
15             | _ => search p R)
```

Recall SML is a **eager** language, and so will fully evaluate the arguments to a function before stepping into the function body.

So we can't define a "short-circuiting" `optOrelse` which only evaluates its second arg when its first argument is `NONE`.

we tell it not to be!

Recall the built-in type `unit`, which had only one value, `()`

```
datatype unit = ()
```

A value of type `unit -> t` is of the form

```
fn () => e
```

which we think of "e, suspended", that is, `e` but tagged to not evaluate yet.

0625.17 (lazysearch.sml)

```
18 type 'a lazy = unit -> 'a
19 fun Eval (f:'a lazy):'a = f()
20 fun Suspend (x:'a):'a lazy = fn () => x
```

Claim `Suspend` is total

Claim If `e:t` is valuable, `Eval(fn () => e)` is valuableIn particular, for all values `v:t`, `Eval(Suspend v)` is valuable.

```
val rec loop : string lazy =
  fn () => loop ()
```

```
elseTry : 'a option lazy * 'a option lazy -> 'a
option lazy
```

REQUIRES: true

ENSURES:

$$\texttt{Eval(elseTry(f,g))} \cong \begin{cases} \texttt{Eval f} & \text{if } \texttt{Eval f} \text{ is not } \texttt{NONE} \\ \texttt{Eval g} & \text{if } \texttt{Eval(f)} \Longrightarrow \texttt{NONE} \end{cases}$$

0625.18 (lazysearch.sml)

```
24 fun elseTry (f : 'a option lazy,
25                g : 'a option lazy)
26           :'a option lazy =
27     fn () =>
28     case f() of
29       NONE => g()
30     | X => X
31 infixr elseTry
```

```
Search : 'a pred -> 'a tree -> 'a option lazy
```

REQUIRES: true

ENSURES:

$$\texttt{Search p T ()} \cong \begin{cases} \texttt{SOME(z)} & \text{where } \texttt{z} \text{ is the top-leftmost element of } \texttt{T} \\ & \text{such that } \texttt{p(z)} \cong \texttt{true} \\ \texttt{NONE} & \text{if there is no such } \texttt{z} \end{cases}$$

0625.19 (lazysearch.sml)

```
35 fun Return (x:'a):'a option lazy =
36   Suspend(SOME x)
```

0625.20 (lazysearch.sml)

```
40 fun Search p Empty = Suspend NONE
41   | Search p (Node(L,x,R)) =
42       if p(x) then Return x else
43       Search p L elseTry Search p R
```

Thank you!