This module verifies the *SeekLowerBound* algorithm in the go-immutable-radix Go library (https://*github.com*/hashicorp/go-immutable-radix).

MODULE *RadixSeekLowerBound*

EXTENDS *FiniteSets*, *Integers*, *Sequences*, *TLC*

Set of characters to use for the alphabet of generated strings.

CONSTANT *Alphabet*

CmpOp is the comparison operator for ordered iteration. This should be TRUE if the first value is less than the second value. This is called on a single element of a sequence.

CONSTANT $CmpOp(\_, \_)$

Length of input strings generated

CONSTANT *MinLength*, *MaxLength*

ASSUME

$\land \{MinLength, MaxLength\} \subseteq Nat$

$\land MinLength \leq MaxLength$

Number of unique elements to construct the radix tree with. This is a set of numbers so you can test with inputs of multiple sizes.

CONSTANT *ElementCounts*

ASSUME $ElementCounts \subseteq Nat$

INSTANCE *RadixTrees*

INSTANCE *RadixIterator*

Inputs is the set of input strings valid for the tree.

$Inputs \triangleq \text{UNION } \{[1 \,..\, n \to Alphabet] : n \in MinLength \,..\, MaxLength\}$

InputSets is the full set of possible inputs we can send to the radix tree.

$InputSets \triangleq \{T \in \text{SUBSET } Inputs : Cardinality(T) \in ElementCounts\}$

TRUE iff the sequence s contains no duplicates. Copied from *CommunityModules*.

$isInjective(s) \triangleq \forall\, i, j \in \text{DOMAIN } s : (s[i] = s[j]) \Rightarrow (i = j)$

Converts a set to a sequence that contains all the elements of S exactly once. Copied from *CommunityModules*.

$setToSeq(S) \triangleq \text{CHOOSE } f \in [1 \,..\, Cardinality(S) \to S] : isInjective(f)$

bytes.Compare in Go

RECURSIVE $GoBytesCompare(\_, \_)$

$GoBytesCompare(X,\ Y) \triangleq$

CASE $X = Y \quad \to 0$

$\square \quad Len(X) = 0 \quad \to -1$

$\square \quad Len(Y) = 0 \quad \to 1$

```
  □ OTHER →
    IF X[1] = Y[1]
      THEN GoBytesCompare(Tail(X), Tail(Y))
      ELSE IF CmpOp(X[1], Y[1]) THEN −1 ELSE 1

 CmpSeq compares two full inputs whereas CmpOp compares only a single element
 of the alphabet.
CmpSeq(X, Y) ≜ GoBytesCompare(X, Y) ≤ 0

 CmpGte is checks if X ≥ Y
CmpGte(X, Y) ≜ X = Y ∨ ¬CmpOp(X, Y)

 Sorted edges based on CmpOp
SortedEdgeLabels(Node) ≜ SortSeq(setToSeq(DOMAIN Node.Edges), CmpOp)

 Returns the index of the first edge
GetLowerBoundEdgeIndex(Node, Label) ≜
  IF ¬∃ e ∈ DOMAIN Node.Edges : e = Label ∨ ¬CmpOp(e, Label) THEN 0
     if there is no lower bound edge, return 0
   ELSE LET
     e ≜ SortedEdgeLabels(Node)
       sorted edges
  IN CHOOSE idx ∈ 1 .. Len(e) :  find the index
    ∧ CmpGte(e[idx], Label)      ≥ to our search label
    ∧ ∨ idx = 1                  and its the first element that is gte
      ∨ CmpOp(e[idx − 1], Label)
```

```
 The expected value is the sorted set of all inputs where the element
 is greater than or equal to the given key.

 EXPLANATION:
  1. We convert the input set to a sequence
  2. Sort the input sequence, this is all inputs sorted now.
  3. Select the subset of the input sequence where it satisfies our comparison.
     The sequence now only has elements greater than or equal to our key
Expected(input, key) ≜
  SelectSeq(SortSeq(setToSeq(input), CmpSeq), LAMBDA elem : CmpSeq(key, elem))

  --algorithm seek_lower_bound
variables
  iterStack = ⟨⟩,
  input ∈ InputSets,
  key ∈ Inputs,
  root = RadixTree(input),
  node = {},
```

```
  search = {},
  result  = {},
  prefixCmp = "UNSET" ;

procedure findMin()begin
FindMin:
  while Len(node.Value) = 0 do
    with
      labels = SortedEdgeLabels(node),
      edges = [n ∈ 1 .. Len(labels) ↦ node.Edges[labels[n]]]
     do
      if Len(edges) > 1 then
        iterStack := iterStack ∘ SubSeq(edges, 2, Len(edges)) ;
      end if ;

      if Len(edges) > 0 then
          recurse again
        node := edges[1] ;
       else
          shouldn't be possible
        return ;
      end if    ;
    end with ;
  end while ;

  iterStack := iterStack ∘ ⟨node⟩ ;
  return ;
end procedure ;


 This entire algorith is almost 1:1 translated where possible from the
 actual implementation in iter.go. That's the point: we're trying to verify
 our algorithm is correct for all inputs.

 Source: https://github.com/hashicorp/go-immutable-radix/blob/f63f49c0b598a5ead21c5015fb4d08fe7e3c21ea/iter.go ≠ L77
begin
    I could've just set these variables in the initializer above but
    to better closely match the algorithm, I reset them here.
Begin:
  iterStack := ⟨⟩ ;
  node := root ;
  search := key ;

Seek:
  while TRUE do
    if Len(node.Prefix) < Len(search) then
      prefixCmp := GoBytesCompare(node.Prefix, SubSeq(search, 1, Len(node.Prefix))) ;
```

```
    else
      prefixCmp := GoBytesCompare(node.Prefix, search);
    end if ;

    if prefixCmp < 0 then
      goto Result;
    elsif prefixCmp > 0 then
      call findMin();
      goto Result;
    end if ;

  Search:
    if Len(node.Value) > 0 ∧ node.Value = key then
      iterStack := iterStack ∘ ⟨node⟩;
      goto Result;
    end if ;

  Consume:
    search := SubSeq(search, Len(node.Prefix) + 1, Len(search));

    if Len(search) = 0 then
      call findMin();
      goto Result;
    end if ;

  NextEdge:
    with
      idx = GetLowerBoundEdgeIndex(node, search[1]),
      labels = SortedEdgeLabels(node),
      edges = [n ∈ 1 .. Len(labels) ↦ node.Edges[labels[n]]]
    do
      if idx = 0 then
        goto Result;
      else
        if idx + 1 ≤ Len(edges) then
          iterStack := iterStack ∘ SubSeq(edges, idx + 1, Len(edges));
        end if ;

        node := edges[idx];
      end if   ;
    end with ;

  end while ;

Result:
  result := Iterate(iterStack);

CheckResult:
```

assert $result = Expected(input,\ key)$ **;**
end algorithm ;

!!! NOTE !!! The rest of the file is auto-generated based on the *PlusCal* above. For those who are reading this to learn TLA+/*PlusCal*, you can stop reading here.

BEGIN TRANSLATION ($chksum(pcal) = \text{“}7f5569db\text{”} \land chksum(tla) = \text{“}b81f869c\text{”}$)

VARIABLES $iterStack,\ input,\ key,\ root,\ node,\ search,\ result,\ prefixCmp,\ pc,\ stack$

$vars \triangleq \langle iterStack,\ input,\ key,\ root,\ node,\ search,\ result,\ prefixCmp,\ pc,\ stack\rangle$

$Init \triangleq$ Global variables
$\land\ iterStack = \langle\rangle$
$\land\ input \in InputSets$
$\land\ key \in Inputs$
$\land\ root = RadixTree(input)$
$\land\ node = \{\}$
$\land\ search = \{\}$
$\land\ result = \{\}$
$\land\ prefixCmp = \text{“UNSET”}$
$\land\ stack = \langle\rangle$
$\land\ pc = \text{“Begin”}$

$FindMin \triangleq \land\ pc = \text{“FindMin”}$
$\land$ IF $Len(node.Value) = 0$
THEN $\land$ LET $labels \triangleq SortedEdgeLabels(node)$ IN
LET $edges \triangleq [n \in 1\ ..\ Len(labels) \mapsto node.Edges[labels[n]]]$ IN
$\land$ IF $Len(edges) > 1$
THEN $\land\ iterStack' = iterStack \circ SubSeq(edges,\ 2,\ Len(edges))$
ELSE $\land$ TRUE
$\land$ UNCHANGED $iterStack$
$\land$ IF $Len(edges) > 0$
THEN $\land\ node' = edges[1]$
$\land\ pc' = \text{“FindMin”}$
$\land\ stack' = stack$
ELSE $\land\ pc' = Head(stack).pc$
$\land\ stack' = Tail(stack)$
$\land\ node' = node$
ELSE $\land\ iterStack' = iterStack \circ \langle node\rangle$
$\land\ pc' = Head(stack).pc$
$\land\ stack' = Tail(stack)$
$\land\ node' = node$

$\land$ UNCHANGED $\langle input, key, root, search, result, prefixCmp\rangle$

$findMin \triangleq FindMin$

$Begin \triangleq$ $\land\ pc = \text{“Begin”}$
$\land\ iterStack' = \langle\rangle$
$\land\ node' = root$
$\land\ search' = key$
$\land\ pc' = \text{“Seek”}$
$\land$ UNCHANGED $\langle input, key, root, result, prefixCmp, stack\rangle$

$Seek \triangleq$ $\land\ pc = \text{“Seek”}$
$\land$ IF $Len(node.Prefix) < Len(search)$
THEN $\land\ prefixCmp' = GoBytesCompare(node.Prefix, SubSeq(search, 1, Len(node.Prefix)))$
ELSE $\land\ prefixCmp' = GoBytesCompare(node.Prefix, search)$
$\land$ IF $prefixCmp' < 0$
THEN $\land\ pc' = \text{“Result”}$
$\land\ stack' = stack$
ELSE $\land$ IF $prefixCmp' > 0$
THEN $\land\ stack' = \langle[procedure \mapsto \text{“findMin”},$
$pc \mapsto \text{“Result”}]\rangle$
$\circ\ stack$
$\land\ pc' = \text{“FindMin”}$
ELSE $\land\ pc' = \text{“Search”}$
$\land\ stack' = stack$
$\land$ UNCHANGED $\langle iterStack, input, key, root, node, search, result\rangle$

$Search \triangleq$ $\land\ pc = \text{“Search”}$
$\land$ IF $Len(node.Value) > 0 \land node.Value = key$
THEN $\land\ iterStack' = iterStack \circ \langle node\rangle$
$\land\ pc' = \text{“Result”}$
ELSE $\land\ pc' = \text{“Consume”}$
$\land$ UNCHANGED $iterStack$
$\land$ UNCHANGED $\langle input, key, root, node, search, result, prefixCmp,$
$stack\rangle$

$Consume \triangleq$ $\land\ pc = \text{“Consume”}$
$\land\ search' = SubSeq(search, Len(node.Prefix) + 1, Len(search))$
$\land$ IF $Len(search') = 0$
THEN $\land\ stack' = \langle[procedure \mapsto \text{“findMin”},$
$pc \mapsto \text{“Result”}]\rangle$
$\circ\ stack$
$\land\ pc' = \text{“FindMin”}$
ELSE $\land\ pc' = \text{“NextEdge”}$
$\land\ stack' = stack$
$\land$ UNCHANGED $\langle iterStack, input, key, root, node, result,$

$prefixCmp\rangle$

$$
\begin{array}{l}
NextEdge \triangleq \ \land\ pc = \text{“NextEdge”} \\
\qquad \land\ \text{LET } idx \triangleq GetLowerBoundEdgeIndex(node,\ search[1])\text{IN} \\
\qquad\quad \text{LET } labels \triangleq SortedEdgeLabels(node)\text{IN} \\
\qquad\qquad \text{LET } edges \triangleq [n \in 1\ldots Len(labels) \mapsto node.Edges[labels[n]]]\text{IN} \\
\qquad\qquad\quad \text{IF } idx = 0 \\
\qquad\qquad\qquad \text{THEN } \land\ pc' = \text{“Result”} \\
\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } \langle iterStack,\ node\rangle \\
\qquad\qquad\qquad \text{ELSE } \land\ \text{IF } idx + 1 \leq Len(edges) \\
\qquad\qquad\qquad\qquad\qquad \text{THEN } \land\ iterStack' = iterStack \circ SubSeq(edges,\ idx + 1,\ Len(edges)) \\
\qquad\qquad\qquad\qquad\qquad \text{ELSE } \land\ \text{TRUE} \\
\qquad\qquad\qquad\qquad\qquad\qquad\ \land\ \text{UNCHANGED } iterStack \\
\qquad\qquad\qquad\qquad\ \land\ node' = edges[idx] \\
\qquad\qquad\qquad\qquad\ \land\ pc' = \text{“Seek”} \\
\qquad \land\ \text{UNCHANGED } \langle input,\ key,\ root,\ search,\ result,\ prefixCmp,\ stack\rangle
\end{array}
$$

$$
\begin{array}{l}
Result \triangleq \ \land\ pc = \text{“Result”} \\
\qquad\ \land\ result' = Iterate(iterStack) \\
\qquad\ \land\ pc' = \text{“CheckResult”} \\
\qquad\ \land\ \text{UNCHANGED } \langle iterStack,\ input,\ key,\ root,\ node,\ search,\ prefixCmp, \\
\qquad\qquad\qquad stack\rangle
\end{array}
$$

$$
\begin{array}{l}
CheckResult \triangleq \ \land\ pc = \text{“CheckResult”} \\
\qquad\qquad\ \land\ Assert(result = Expected(input,\ key), \\
\qquad\qquad\qquad\qquad \text{“Failure of assertion at line 192, column 3.”}) \\
\qquad\qquad\ \land\ pc' = \text{“Done”} \\
\qquad\qquad\ \land\ \text{UNCHANGED } \langle iterStack,\ input,\ key,\ root,\ node,\ search, \\
\qquad\qquad\qquad\qquad result,\ prefixCmp,\ stack\rangle
\end{array}
$$

Allow infinite stuttering to prevent deadlock on termination.

$$Terminating \triangleq pc = \text{“Done”} \land \text{UNCHANGED } vars$$

$$
\begin{array}{l}
Next \triangleq findMin \lor Begin \lor Seek \lor Search \lor Consume \lor NextEdge \lor Result \\
\qquad\quad \lor CheckResult \\
\qquad\quad \lor Terminating
\end{array}
$$

$$Spec \triangleq Init \land \Box[Next]_{vars}$$

$$Termination \triangleq \Diamond(pc = \text{“Done”})$$

END TRANSLATION

* Modification History
* Last modified *Thu Jul* 01 22:44:22 *PDT* 2021 by *mitchellh*
* Created *Thu Jul* 01 10:43:00 *PDT* 2021 by *mitchellh*