lib/search/binary-search.ath

```
# Binary search function for searching in a binary search tree, and
# correctness theorems.  Generalized from natural number version in
# binary-search1-nat.ath.

load "binary-search-tree"

#-----------------------------------------------------------------------

extend-module SWO {

declare binary-search: (S) [S (BinTree S)] -> (BinTree S)

module binary-search {
define [x L y R L1 y1 R1 T] := [?x:'S ?L:(BinTree 'S) ?y:'S ?R:(BinTree 'S)
                                 ?L1:(BinTree 'S) ?y1:'S ?R1:(BinTree 'S)
                                 ?T:(BinTree 'S)]

define (axioms as [go-left go-right at-root empty]) :=
  (fun
   [(binary-search x (node L y R)) =
       [(binary-search x L)    when (x < y)
        (binary-search x R)    when (y < x)
        (node L y R)           when (~ x < y & ~ y < x)]
    (binary-search x null) = null])

(add-axioms theory axioms)

# Theorems:

define in := BST.in

define found :=
  (forall T . BST T ==>
              forall x L y R .
                (binary-search x T) = (node L y R) ==> x E y & x in T)

define not-found :=
  (forall T . BST T ==>
              forall x . (binary-search x T) = null ==> ~ x in T)

define in-iff-result-not-null :=
  (forall T .
    BST T ==>
    forall x . x in T <==> (binary-search x T) =/= null)

define theorems := [found not-found in-iff-result-not-null]

define tree-axioms := (datatype-axioms "BinTree")

define (found-property T) :=
  (forall x L1 y1 R1 .
     (binary-search x T) = (node L1 y1 R1) ==> x E y1 & x in T)

define (not-found-prop T) :=
  (forall x . (binary-search x T) = null ==> ~ x in T)

define proofs :=
  method (theorem adapt)
    let {[get prove chain chain-> chain<-] := (proof-tools adapt theory);
         [< <E E BST binary-search] :=
           (adapt [< <E E BST binary-search])}
    match theorem {
      (val-of found) =>
      by-induction (adapt theorem) {
        null =>
          conclude (BST null ==> found-property null)
            assume (BST null)
```

```
              pick-any x L y R
                assume A := ((binary-search x null) = (node L y R))
                  let {is-null :=
                        (!chain
                         [null
                        = (binary-search x null)    [empty]
                        = (node L y R)              [A]]);
                       is-not := (!chain->
                                  [true ==> (null =/= (node L y R))
                                                    [tree-axioms]])}
                  (!from-complements (x E y & x in null) is-null is-not)
      | (T as (node L:(BinTree 'S) y:'S R:(BinTree 'S))) =>
        let {[ind-hyp1 ind-hyp2] := [(BST L ==> found-property L)
                                     (BST R ==> found-property R)]}
        assume hyp := (BST T)
          conclude (found-property T)
            let {p0 := (BST L & (forall x . x in L ==> x <E y) &
                        BST R & (forall z . z in R ==> y <E z));
                 _ := (!chain-> [hyp ==> p0         [BST.nonempty]]);
                 fpl := (!chain->
                         [p0 ==> (BST L)            [left-and]
                             ==> (found-property L) [ind-hyp1]]);
                 fpr := (!chain->
                         [p0 ==> (BST R)            [prop-taut]
                             ==> (found-property R) [ind-hyp2]])}
            pick-any x:'S L1:(BinTree 'S) y1:'S R1:(BinTree 'S)
              let {subtree := (node L1 y1 R1)}
              assume hyp' := ((binary-search x T) = subtree)
               conclude (x E y1 & x in T)
                 (!two-cases
                  assume (x < y)
                    let {in-left := (!prove BST.in.left)}
                    (!chain->
                     [(binary-search x L)
                      = (binary-search x T)         [go-left]
                      = subtree                     [hyp']
                      ==> (x E y1 & x in L)         [fpl]
                      ==> (x E y1 & x in T)         [in-left]])
                  assume (~ x < y)
                    (!two-cases
                     assume (y < x)
                       let {in-right := (!prove BST.in.right)}
                       (!chain->
                        [(binary-search x R)
                         = (binary-search x T)      [go-right]
                         = subtree                  [hyp']
                         ==> (x E y1 & x in R)      [fpr]
                         ==> (x E y1 & x in T)      [in-right]])
                     assume (~ y < x)
                       let {_ := (!chain->
                                  [(~ x < y & ~ y < x)
                                   ==> (x E y)      [E-definition]]);
                            i := conclude (y = y1)
                                   (!chain->
                                    [T = (binary-search x T)
                                                    [at-root]
                                       = subtree    [hyp']
                                       ==> (y = y1) [tree-axioms]]);
                            ii := conclude (x E y1)
                                  (!chain->
                                   [(x E y)
                                ==> (x E y1)        [i]]);
                            in-root := (!prove BST.in.root)}
                       (!chain-> [(x E y)
                              ==> (x in T)          [in-root]
                              ==> (ii & x in T)     [augment]])))
      }
    | (val-of not-found) =>
      by-induction (adapt theorem) {
        null =>
```

```
     assume (BST null)
       conclude (not-found-prop null)
         pick-any x
           assume ((binary-search x null) = null)
             (!chain-> [true ==> (~ x in null) [BST.in.empty]])
| (T as (node L y R)) =>
    let {p1 := (not-found-prop L);
         p2 := (not-found-prop R);
         [ind-hyp1 ind-hyp2] := [(BST L ==> p1) (BST R ==> p2)]}
    assume hyp := (BST T)
        conclude (not-found-prop T)
          let {smaller-in-left := (forall x . x in L ==> x <E y);
               larger-in-right := (forall z . z in R ==> y <E z);
               p0 := (BST L & smaller-in-left &
                      BST R & larger-in-right);
               _ := (!chain-> [hyp ==> p0             [BST.nonempty]]);
               _ := (!chain-> [p0 ==> smaller-in-left [prop-taut]]);
               _ := (!chain-> [p0 ==> larger-in-right [prop-taut]]);
               _ := (!chain-> [p0
                           ==> (BST L)                [prop-taut]
                           ==> (not-found-prop L)     [ind-hyp1]]);
               _ := (!chain-> [p0
                           ==> (BST R)                [prop-taut]
                           ==> (not-found-prop R)     [ind-hyp2]])}
           pick-any x
             assume hyp' := ((binary-search x T) = null)
               (!by-contradiction (~ x in (node L y R))
                assume (x in T)
                  let {C := (!chain->
                              [(x in T)
                           ==> (x E y | x in L |  x in R)
                                                      [BST.in.nonempty]])}
                  (!two-cases
                   assume (x < y)
                     let {_ := (!chain->
                                [(binary-search x L)
                               = (binary-search x T)  [go-left]
                               = null                 [hyp']
                               ==> (~ x in L)         [p1]])}
                     (!cases C
                      assume (x E y)
                        (!absurd
                         (x < y)
                         (!chain->
                          [(x E y)
                           ==> (~ x < y & ~ y < x)    [E-definition]
                           ==> (~ x < y)              [left-and]]))
                      assume (x in L)
                        (!absurd (x in L) (~ x in L))
                      assume (x in R)
                        (!absurd (x < y)
                                 (!chain->
                                  [(x in R)
                                   ==> (y <E x)   [larger-in-right]
                                   ==> (~ x < y)  [<E-definition]])))
                   assume (~ x < y)
                     (!two-cases
                      assume (y < x)
                        let {_ := (!chain->
                                   [(binary-search x R)
                                  = (binary-search x T) [go-right]
                                  = null                [hyp']
                                  ==> (~ x in R)        [p2]])}
                        (!cases C
                         assume (x E y)
                           (!absurd
                            (y < x)
                            (!chain->
                             [(x E y)
                          ==> (~ x < y & ~ y < x)   [E-definition]
```

```
                              ==> (~ y < x)             [right-and]]))
                             assume (x in L)
                                (!absurd
                                 (y < x)
                                 (!chain->
                                  [(x in L)
                                   ==> (x <E y)         [smaller-in-left]
                                   ==> (~ y < x)        [<E-definition]]]))
                             assume (x in R)
                                (!absurd (x in R) (~ x in R)))
                          assume (~ y < x)
                             (!absurd
                              (!chain->
                               [null = (binary-search x T) [hyp']
                                    = T                 [at-root]])
                              (!chain->
                               [true
                                ==> (null =/= T)        [tree-axioms]]))))))
      }
    | (val-of in-iff-result-not-null) =>
      pick-any T
        assume (BST T)
          let {NF := (!prove not-found);
               F := (!prove found)}
          pick-any x
            let {right :=
                   assume (x in T)
                     (!by-contradiction ((binary-search x T) /= null)
                      assume A1 := ((binary-search x T) = null)
                        (!absurd (x in T)
                          (!chain-> [A1 ==> (~ x in T)  [NF]])));
                 left :=
                   assume A2 := ((binary-search x T) =/= null)
                     let {p := (exists ?L ?y ?R .
                                 (binary-search x T) = (node ?L ?y ?R));
                         _ := (!chain->
                                [true
                                 ==> ((binary-search x T) = null | p)
                                                  [tree-axioms]
                                 ==> p            [(dsyl with A2)]])}
                        pick-witnesses L y R for p p'
                          (!chain->
                           [p' ==> (x E y & x in T) [F]
                              ==> (x in T)        [right-and]])}
            (!equiv right left)
    } # match theorem

(add-theorems theory |{theorems := proofs}|)
} # binary-search
} # SWO
```