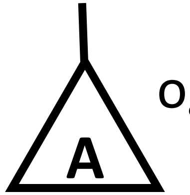
Tvíundartré og tvíleitartré

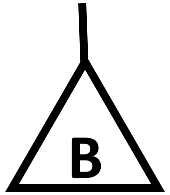
Grunnskilgreiningar og setningar

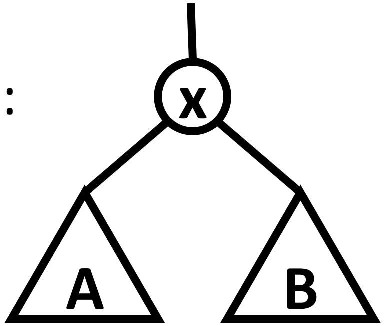
Hvað er tvíundartré (Binary Search Tree, BST)?

- Í Dafny skilgreinum við tvíundartré svona
  ```
  datatype BST = BSTEmpty | BSTNode(BST,int,BST)
  ```

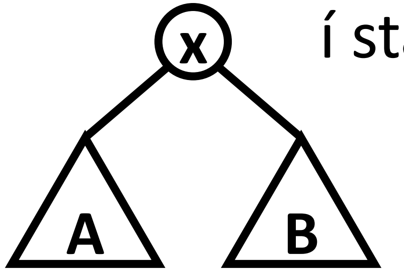
- Tómt tré er því tvíundartré, táknað með BSTEmpty
  - Teiknum tómt tré svona: $\perp$

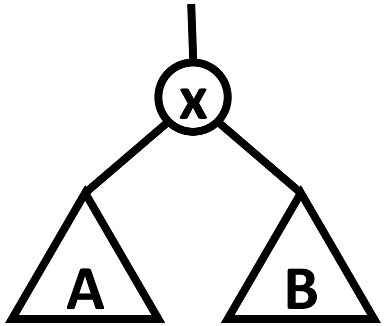
- Ef x er gildi og A og B eru tvíundartré þá er BSTNode(A,x,B) einnig tvíundartré
  - Ef A og B eru teiknuð svona 

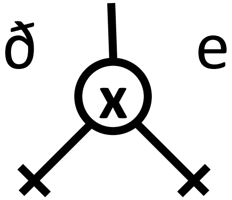
    og svona 

þá teiknum við BSTNode(A,x,B) svona: 

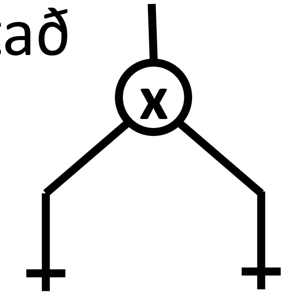
Hvað er tvíundartré (Binary Search Tree, BST)?

- Til hægðarauka teiknum við oft 

í stað 

- Einnig notum við 

eða jafnvel 

í stað 

- Við gerum ráð fyrir að allir séu sammála um merkingu grunnhugtaka varðandi tré, til dæmis „hnútur“, „rót“, „undirtré“, „milliröð“ (inorder röð)

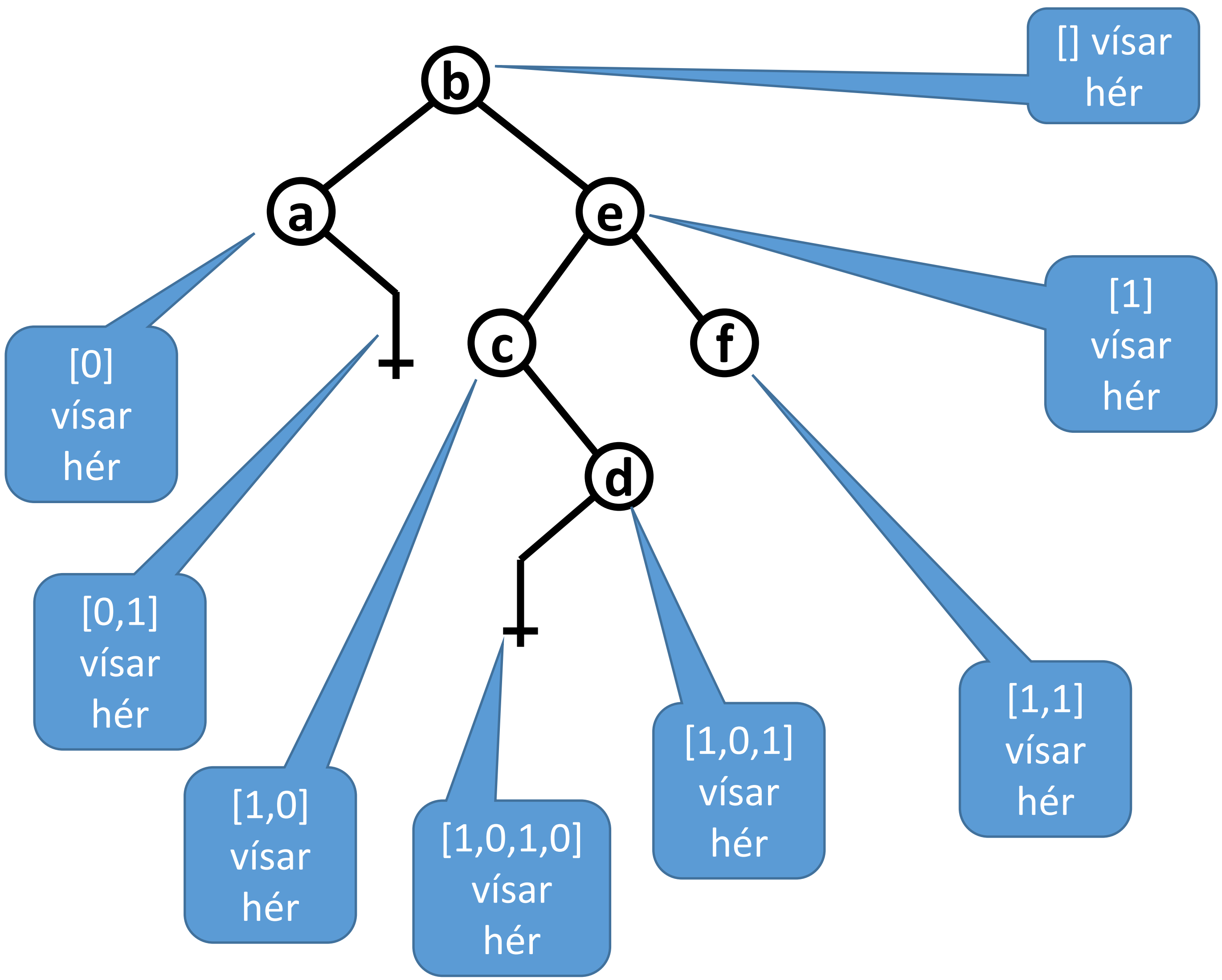
Gildarunur trjáa

- Sérhvert tré t skilgreinir endanlega runu gilda
  TreeSeq(t)
- Tóma tréð, t=BSTEmpty, gefur tómu rununa,
  TreeSeq(t)=[]
- Tré með rót, t=BSTNode(A,x,B), gefur rununa
  TreeSeq(t)=TreeSeq(A)+[x]+TreeSeq(B)
- Gildaruna trés er runan af gildum í hnútum þess í milliröð, þ.e. „inorder“ röð

Trjáslóðir innan trjáa

- Ef t er tré og p er endanleg runa af 0 og 1 þá má vera að p sé trjáslóð (tree path) innan t sem vísar þá á eitthvert undirtré t
- Tóma runan [] er trjáslóð innan allra trjáa og vísar í tréð t, sem er undirtré sjálfs sín
- Ef tré hefur rót, t=BSTNode(A,x,B) þá er runan [0]+p trjáslóð innan t þá og því aðeins að p sé trjáslóð innan A og undirtréð sem [0]+p vísar á innan t er þá undirtréð innan A sem p vísar á
- Ef tré hefur rót, t=BSTNode(A,x,B) þá er runan [1]+p trjáslóð innan t þá og því aðeins að p sé trjáslóð innan B og undirtréð sem [1]+p vísar á innan t er þá undirtréð innan B sem p vísar á
- Ef p er trjáslóð innan trés t þá notum við Subtree(t,p) til að tákna það undirtré innan t sem p vísar á

Dæmi um trjáslóðir



Fjölmargar trjáslóðir, meira en helmingur, vísa á tóm undirtré. Öll lauf hafa til dæmis tvö tóm undirtré.

Hér eru slóðirnar [0,1] og [1,0,1,0] teknar sem dæmi um slóðir sem vísa á tóm undirtré, en aðrar slíkar slóðir innan þessa trés eru til dæmis [0,0], [1,1,0] og [1,0,1,1].

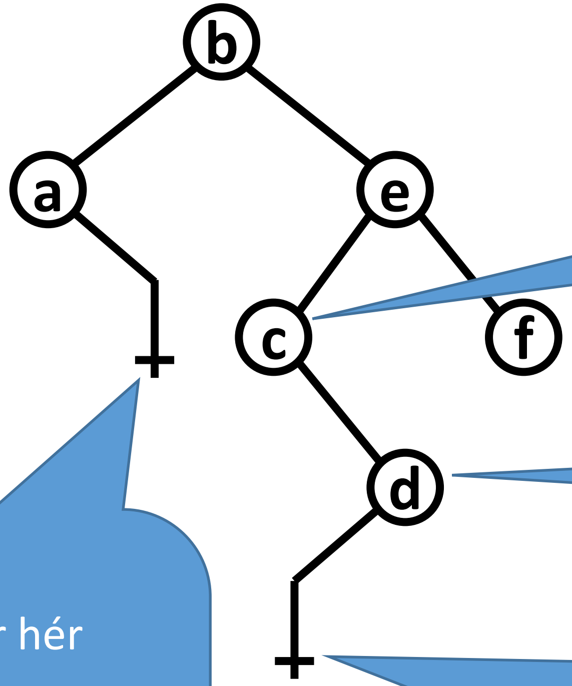
Gildaruna þessa trés er [a,b,c,d,e,f].

Þrískipting gildarunu trjáa

- Fyrir sérhvert tré t skilgreinir sérhver trjáslóð p innan t þrískiptingu á gildarunu trésins
- Skilgreinum PreSeq(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru vinstra megin við það undirtré sem p vísar á
- Skilgreinum PostSeq(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru hægra megin við það undirtré sem p vísar á
- Skilgreinum MidSeq(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru í því undirtré sem p vísar á
- Sanna má (í Dafny) að fyrir sérhverja trjáslóð p innan trés t gildir

TreeSeq(t) = PreSeq(t,p)+MidSeq(t,p)+PostSeq(t,p)

Dæmi um þrískiptingar fyrir tré t



[1,0] vísar hér

PreSeq(t,[1,0])=[a,b]
MidSeq(t,[1,0])=[c,d]
PostSeq(t,[1,0])=[e,f]

[1,0,1] vísar hér

PreSeq(t,[1,0,1])=[a,b,c]
MidSeq(t,[1,0,1])=[d]
PostSeq(t,[1,0,1])=[e,f]

[0,1] vísar hér

PreSeq(t,[0,1])=[a]
MidSeq(t,[0,1])=[]
PostSeq(t,[0,1])=
[b,c,d,e,f]

[1,0,1,0] vísar hér

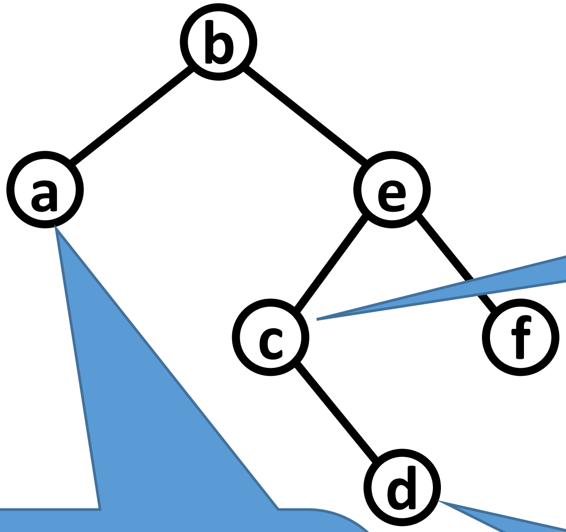
PreSeq(t,[1,0,1,0])=[a,b,c]
MidSeq(t,[1,0,1,0])=[]
PostSeq(t,[1,0,1,0])=[d,e,f]

Tvískipting gildarunu trjáa

- Fyrir sérhvert tré t skilgreinir sérhver trjáslóð p innan t, sem vísar á ekki-tómt undirtré, tvískiptingu á gildarunu trésins
- Skilgreinum PreSeqExcluding(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru vinstra megin við þann hnút sem p vísar á, að hnútnum sjálfum ekki meðtöldum
  - Aftasta gildi rununnar (ef runan er ekki tóm) er gildi þess hnútar, ef einhver er, sem er á undan viðkomandi hnút í milliröð
- Skilgreinum PostSeqIncluding(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru hægra megin við þann hnút sem p vísar á, að hnútnum sjálfum meðtöldum
  - Fremsta gildi rununnar er þá gildi þess hnútar sem p vísar á
- Sanna má (í Dafny) að fyrir sérhverja slíka trjáslóð p innan trés t gildir

  TreeSeq(t) = PreSeqExcluding(t,p)+PostSeqIncluding(t,p)

Dæmi um tvískiptingar fyrir tré t



[1,0] vísar hér

PreSeqExcluding(t,[1,0])=[a,b]
PostSeqIncluding(t,[1,0])=[c,d,e,f]

[0] vísar hér

PreSeqExcluding(t,[0])=[]
PostSeqIncluding(t,[0])=
[a,b,c,d,e,f]

[1,0,1] vísar hér

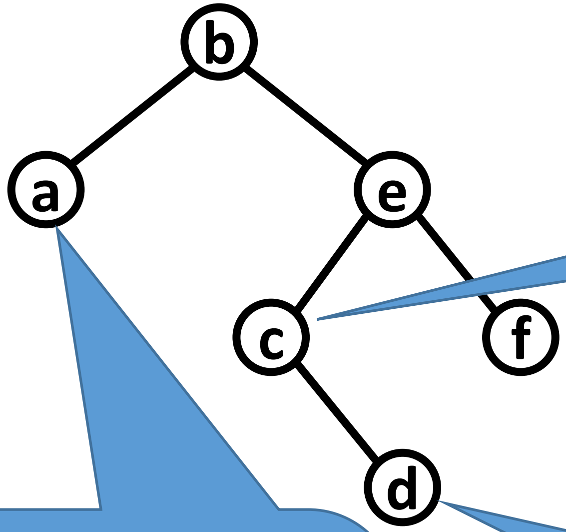
PreSeqExcluding(t,[1,0,1])=[a,b,c]
PostSeqIncluding(t,[1,0,1])=[d,e,f]

Önnur tvískipting gildarunu trjáa

- Fyrir sérhvert tré t skilgreinir sérhver trjáslóð p innan t, sem vísar á ekki-tómt undirtré, aðra tvískiptingu á gildarunu trésins
- Skilgreinum PreSeqIncluding(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru vinstra megin við þann hnút sem p vísar á, að hnútnum sjálfum meðtöldum
  - Aftasta gildi rununnar er gildi þess hnútar sem p vísar á
- Skilgreinum PostSeqExcluding(t,p) sem runu gilda þeirra hnúta, í milliröð, sem eru hægra megin við þann hnút sem p vísar á, að hnútnum sjálfum ekki meðtöldum
  - Fremsta gildi rununnar er þá gildi þess hnútar sem er fyrir aftan hnútinn sem p vísar á, ef einhver er
- Sanna má (í Dafny) að fyrir sérhverja slíka trjáslóð p innan trés t gildir

  TreeSeq(t) = PreSeqIncluding(t,p)+PostSeqExcluding(t,p)

Önnur dæmi um tvískiptingar fyrir tré t



[1,0] vísar hér

PreSeqIncluding(t,[1,0])=[a,b,c]
PostSeqExcluding(t,[1,0])=[d,e,f]

[0] vísar hér

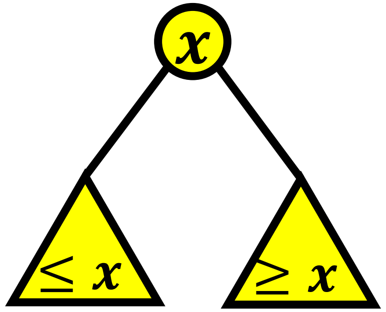
PreSeqIncluding(t,[0])=[a]
PostSeqExcluding(t,[0])=
[b,c,d,e,f]

[1,0,1] vísar hér

PreSeqIncluding(t,[1,0,1])=[a,b,c,d]
PostSeqExcluding(t,[1,0,1])=[e,f]

Tvíleitartré

- Tvíleitartré eru tvíundartré þannig að gildin í trénu eru í vaxandi röð frá vinstri til hægri
- Með öðrum orðum, gildin í trénu í milliröð („inorder“ röð) eru í vaxandi röð
- Með öðrum orðum, tré t er tvíleitartré ef TreeSeq(t) er í vaxandi röð
- Með öðrum orðum, tré er tvíleitartré ef um alla hnúta gildir að öll gildi í vinstra undirtré eru $\leq$ gildi hnútarins og öll gildi í hægra undirtré eru $\geq$ gildi hnútarins

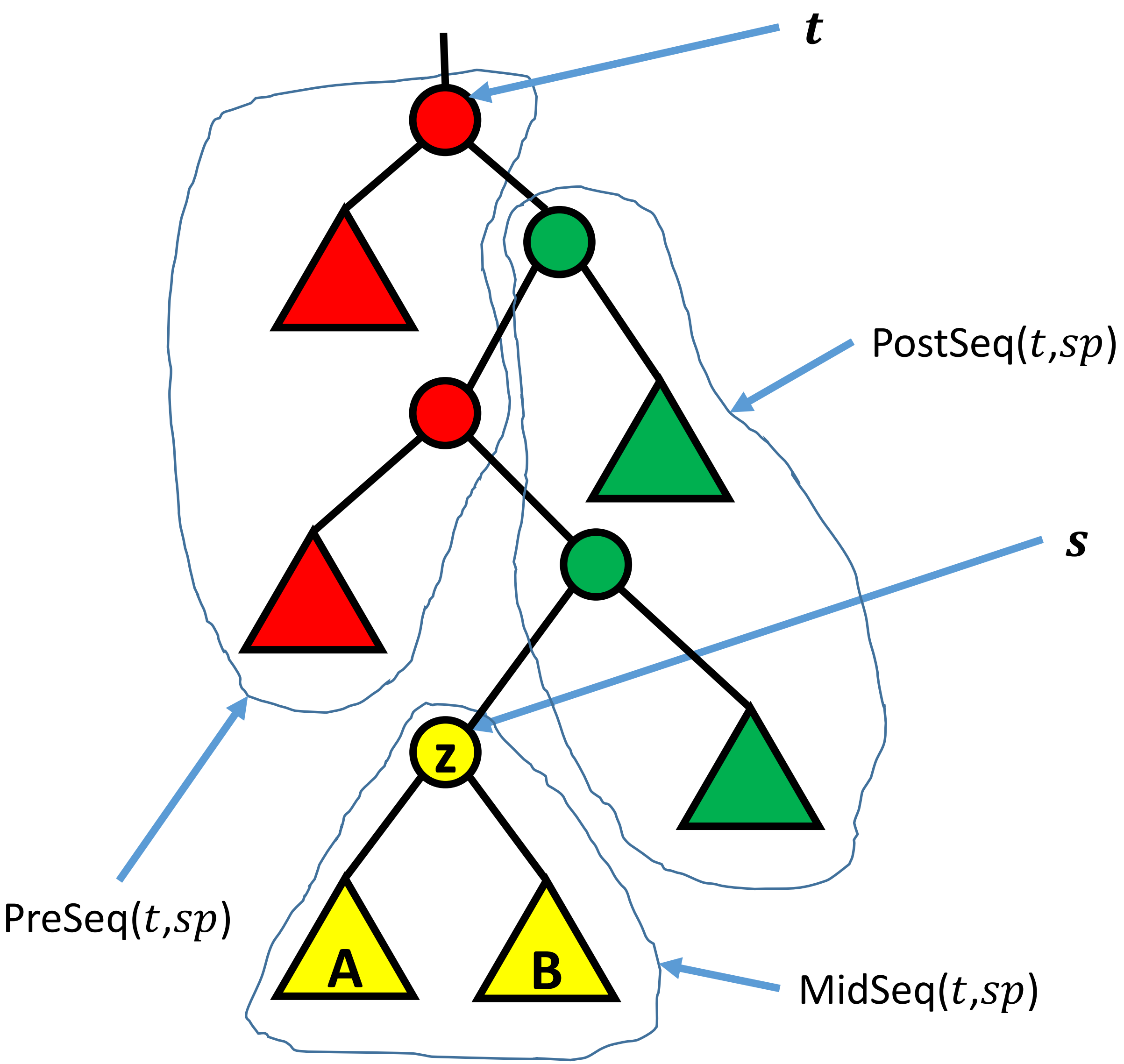


Fastayrðing lykkju fyrir leit að einhverju $= x$

- Leitum að einhverjum hnút í tré t með gildi x
- Notum lykkju og höfum breytur s og p (p má vera draugabreyta) þannig að viðhaldið sé þeirri fastayrðingu að
  - s = Subtree(t,p)
  - Öll gildi í PreSeq(t,p) eru $< x$
  - Öll gildi í PostSeq(t,p) eru $> x$
- Hætt er í lykkjunni ef s er tómt (x er þá ekki til í trénu) eða ef rót undirtrésins s hefur gildi x

PreSeq(t,p)	**MidSeq(t,p) =Treeseq(s)**	**PostSeq(t,p)**
$< x$	**?**	$> x$

Myndrænt ástandsdæmi



Næsta skref er að athuga hvort rótin á trénu s á að vera rauð ($\mathrm{z} < x$) eða græn ($\mathrm{z} > x$), eða hvort gildið sé fundið ($\mathrm{z} = x$)

Ef $\mathrm{z} > x$ þá verða z og B græn:
$s := A$

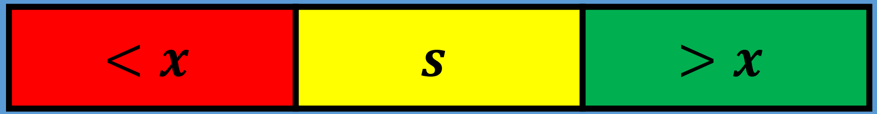
Ef $\mathrm{z} < x$ þá verða z og A rauð:
$s := B$

sp er trjáslóðin innan t sem vísar á undirtréð s.

Í Dafny (án vélrænnar sönnunar)

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
  var s := t;
  while s != BSTEmpty {
    if RootValue(s) > x {
      s := Left(s);
    } else if RootValue(s) < x {
      s := Right(s);
    } else {
      return s;
    }
  }
  return BSTEmpty;
}
```

Fastayrðing lykkju: Miðsvæðið (gula) stendur fyrir gildarunu undirtrésins s.



Athugið að fremsta svæðið er PreSeq(t,sp) og aftasta svæðið er PostSeq(t,sp).

Halaendurkvæm útgáfa

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
  if t == BSTEmpty {
    r := BSTEmpty;
  } else if x < RootValue(t) {
    r := SearchAnyEQ(Left(t),x);
  } else if x > RootValue(t) {
    r := SearchAnyEQ(Right(t),x);
  } else {
    r := t;
  }
}
```

Jafngilt

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
  match t
  case BSTEmpty =>
    { r := BSTEmpty;                    }
  case BSTNode(left,val,right) =>
    {
      if x < val
        { r := SearchAnyEQ(left,x);   }
      else if x > val
        { r := SearchAnyEQ(right,x);  }
      else
        { r := t;                     }
    }
}
```

For- og eftirskilyrði

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
  requires TreeIsSorted(t)
  ensures x in TreeSeq(t) ==>
    r != BSTEmpty &&
    IsSubTree(t,r) &&
    x == RootValue(r)
  ensures x !in TreeSeq(t) ==>
    r == BSTEmpty
```

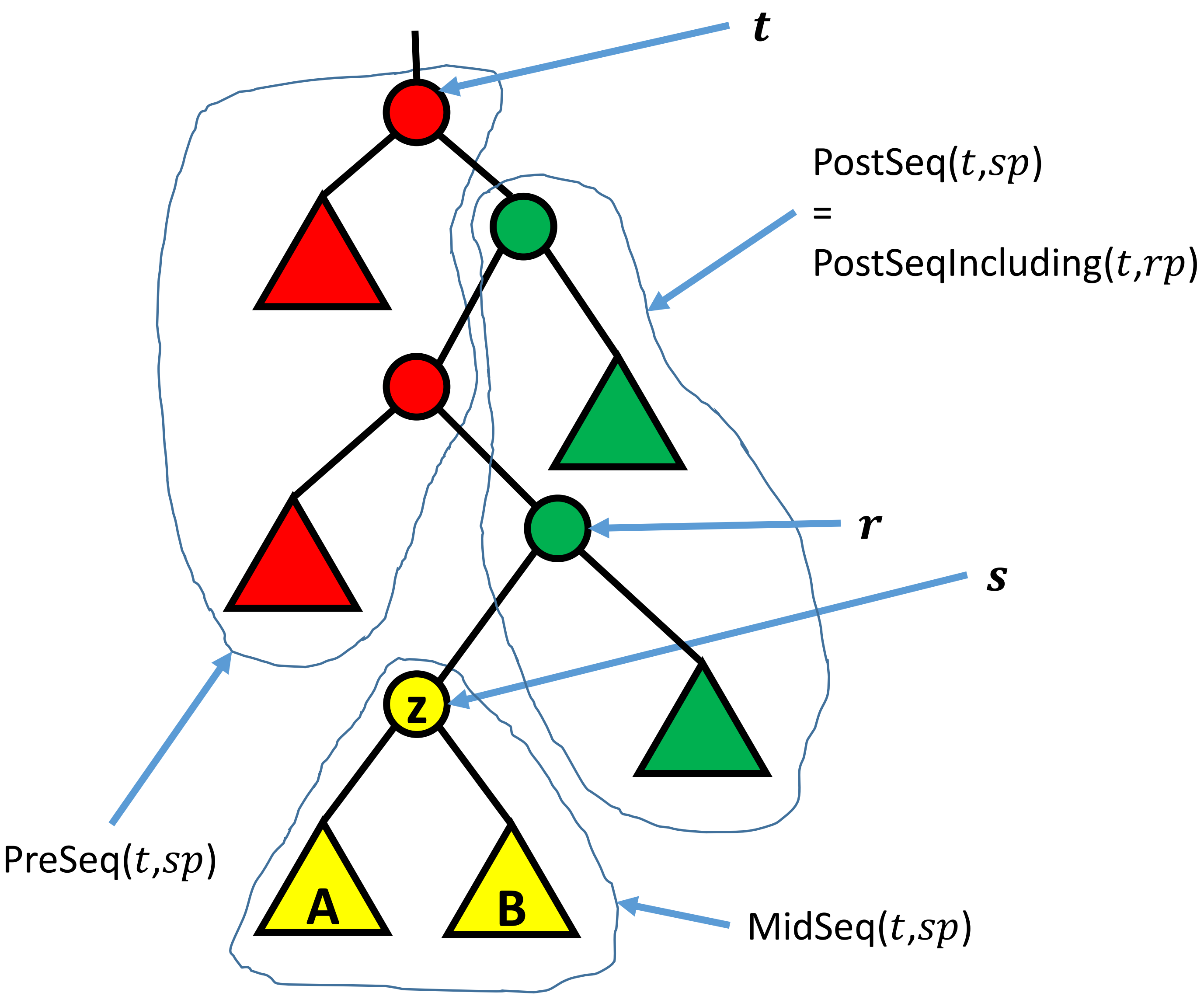
Fastayrðing lykkju fyrir leit að fremsta $\geq x$

- Leitum að fremsta hnút í tré t með gildi $\geq x$
- Notum lykkju og höfum breytur s, r, sp og rp (sp og rp mega vera draugabreytur) þannig að viðhaldið sé þeirri fastayrðingu að
  - s = Subtree(t,sp)
  - Öll gildi í PreSeq(t,sp) eru $< x$
  - Öll gildi í PostSeq(t,sp) eru $\geq x$
  - Ef PostSeq(t,sp) er ekki tóm þá vísar r á ekki-tómt undirtré í t þannig að r=SubTree(t,rp) og PostSeqIncluding(t,rp)=PostSeq(t,sp)
    - með öðrum orðum, r vísar á fremsta hnút fyrir aftan undirtréð s
  - Ef PostSeq(t,sp) er tóm þá er r tóma tréð
- Hætt er í lykkjunni þegar s er tómt og r er þá skilað

PreSeq(t,sp)	MidSeq(t,sp)=Treeseq(s)	PostSeq(t,sp)
$< x$	?	$\geq x$

$\uparrow$
r

Myndrænt ástandsdæmi



Næsta skref er að athuga hvort rótin á trénu s á að vera rauð ($\mathrm{z} < x$) eða græn ($\mathrm{z} \geq x$).

Ef $\mathrm{z} \geq x$ þá uppfærist r ásamt því að z og B færast í græna svæðið:

$$r := s$$
$$s := A$$

Ef $\mathrm{z} < x$ þá færast z og A í rauða svæðið en r breytist ekki:

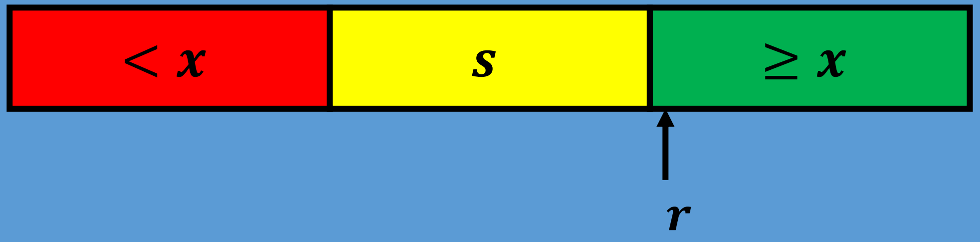
$$s := B$$

sp er trjáslóðin innan t sem vísar á undirtréð s, rp er trjáslóðin innan t sem vísar á undirtréð r (ef eitthvert er).

Í Dafny (án vélrænnar sönnunar)

```
method SearchFirstGELoop( t: BST, x: int )
     returns ( r: BST )
{
  var s := t;
  r := BSTEmpty;
  while s != BSTEmpty {
    if RootValue(s) >= x {
      r := s;
      s := Left(s);
    } else {
      s := Right(s);
    }
  }
}
```

Fastayrðing lykkju: Miðsvæðið (gula) stendur fyrir gildarunu undirtrésins s. r vísar á fremsta hnút í aftasta svæði (græna), ef einhver er, annars er r=BSTEmpty



Athugið að fremsta svæðið er PreSeq(t,sp) og aftasta svæðið er bæði PostSeq(t,sp) og PostSeqIncluding(t,rp) þar sem sp er trjáslóðin til s og rp er trjáslóðin til r (ef til er).

Í Dafny (halaendurkvæmt, án vélrænnar sönnunar)

```
method SearchFirstGERecursiveHelp( t: BST, x: int, c: BST )
  returns ( r: BST )
{
  if t == BSTEmpty {
    r := c;
  } else if RootValue(t) < x {
    r := SearchFirstGERecursiveHelp(Right(t),x,c);
  } else {
    r := SearchFirstGERecursiveHelp(Left(t),x,t);
  }
}
```

Fallið skilar tré r sem hefur rót sem er fremsti hnútur innan t sem hefur gildi $\geq x$, eða skilar trénu c ef enginn slíkur hnútur er til innan t.
Tréð c er því þrautalendingin ef ekkert $\geq x$ finnst í t.

Kallið
`SearchFirstGERecursiveHelp(t,x,BSTEmpty)`
skilar þá réttu tré miðað við hið skilgreinda vandamál

Jafngilt

```
method SearchFirstGERecursiveHelp( t: BST, x: int, c: BST )
           returns ( r: BST )
{
  match t
  case BSTEmpty =>
    { r := c;                                          }
  case BSTNode(left,val,right) =>
    {
      if val < x
        { r := SearchFirstGERecursiveHelp(Right(t),x,c); }
      else
        { r := SearchFirstGERecursiveHelp(Left(t),x,t);  }
    }
}
```

For- og eftirskilyrði

```
method SearchFirstGE( t: BST, x: int ) returns ( r: BST )

  requires TreeIsSorted(t)

  ensures (forall z | z in TreeSeq(t) :: z < x) ==>
    r == BSTEmpty

  ensures (exists z | z in TreeSeq(t) :: z >= x) ==>
    r != BSTEmpty &&
    (exists rp | IsTreePath(t,rp) ::
       r == Subtree(t,rp) &&
       (forall z | z in PreSeqExcluding(t,rp) :: z < x) &&
       (forall z | z in PostSeqIncluding(t,rp) :: z >= x)
    )
```

Fastayrðing lykkju fyrir leit að aftasta $< x$

- Leitum að aftasta hnút í tré t með gildi $< x$
- Notum lykkju og höfum breytur s, r, sp og rp (sp og rp mega vera draugabreytur) þannig að viðhaldið sé þeirri fastayrðingu að
    - s = Subtree(t,sp)
    - Öll gildi í PreSeq(t,sp) eru $< x$
    - Öll gildi í PostSeq(t,sp) eru $\geq x$
    - Ef PreSeq(t,sp) er ekki tóm þá vísar r á ekki-tómt undirtré t þannig að r=SubTree(t,rp) og PreSeqIncluding(t,rp)=PreSeq(t,sp)
        - með öðrum orðum, r vísar á aftasta hnút fyrir framan undirtréð s
    - Ef PreSeq(t,sp) er tóm þá er r tóma tréð
- Hætt er í lykkjunni þegar s er tómt og r er þá skilað

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$< x$	**?**	$\geq x$

r

Fastayrðing lykkju fyrir leit að fremsta $> x$

- Leitum að fremsta hnút í tré t með gildi $> x$
- Notum lykkju og höfum breytur s, r, sp og rp (sp og rp mega vera draugabreytur) þannig að viðhaldið sé þeirri fastayrðingu að
  - s = Subtree(t,sp)
  - Öll gildi í PreSeq(t,sp) eru $\leq x$
  - Öll gildi í PostSeq(t,sp) eru $> x$
  - Ef PostSeq(t,sp) er ekki tóm þá vísar r á ekki-tómt undirtré t þannig að r=SubTree(t,rp) og PostSeqIncluding(t,rp)=PostSeq(t,sp)
    - með öðrum orðum, r vísar á fremsta hnút fyrir aftan undirtréð s
  - Ef PostSeq(t,sp) er tóm þá er r tóma tréð
- Hætt er í lykkjunni þegar s er tómt og r er þá skilað

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$\leq x$	**?**	$> x$

r

Fastayrðing lykkju fyrir leit að aftasta $\leq x$

- Leitum að aftasta hnút í tré t með gildi $\leq x$
- Notum lykkju og höfum breytur s, r, sp og rp (sp og rp mega vera draugabreytur) þannig að viðhaldið sé þeirri fastayrðingu að
  - s = Subtree(t,sp)
  - Öll gildi í PreSeq(t,sp) eru $\leq x$
  - Öll gildi í PostSeq(t,sp) eru $> x$
  - Ef PreSeq(t,sp) er ekki tóm þá vísar r á ekki-tómt undirtré t þannig að r=SubTree(t,rp) og PreSeqIncluding(t,rp)=PreSeq(t,sp)
    - með öðrum orðum, r vísar á aftasta hnút fyrir framan undirtréð s
  - Ef PreSeq(t,sp) er tóm þá er r tóma tréð
- Hætt er í lykkjunni þegar s er tómt og r er þá skilað

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$\leq x$	**?**	$> x$

r

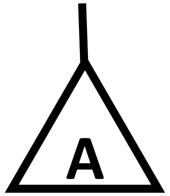
Binary trees and binary search trees

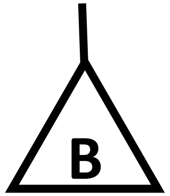
Fundamental definitions and theorems

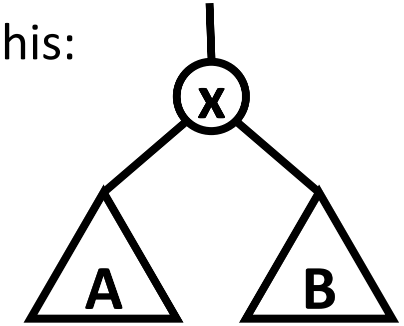
What is a binary tree and a binary search tree?

- In Dafny we define a binary tree like this
  datatype BST = BSTEmpty | BSTNode(BST,int,BST)

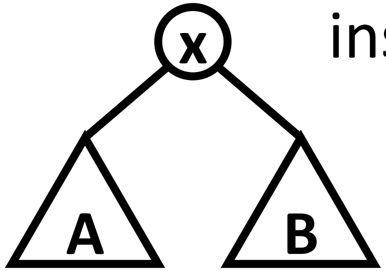
- An empty tree is thus a binary tree, denoted by BSTEmpty
  - We draw an empty tree like this: 

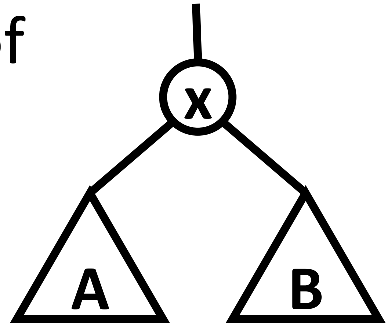
- If x is a value and A and B are binary trees then BSTNode(A,x,B) is also a binary tree
  - If A and B are drawn like this 

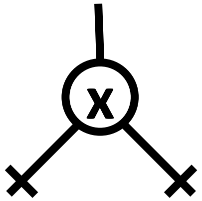
and like this 

Then we draw BSTNode(A,x,B) like this: 

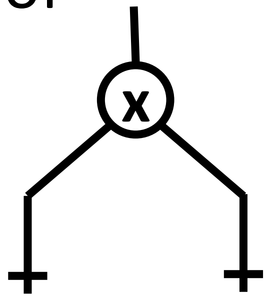
What is a binary tree and a binary search tree?

- For convenience we often draw 

instead of 

- We also use 

or even 

instead of 

- We assume everyone agrees about the meaning of fundamental ideas regarding trees, for example “node”, “root”, “subtree”, “inorder”

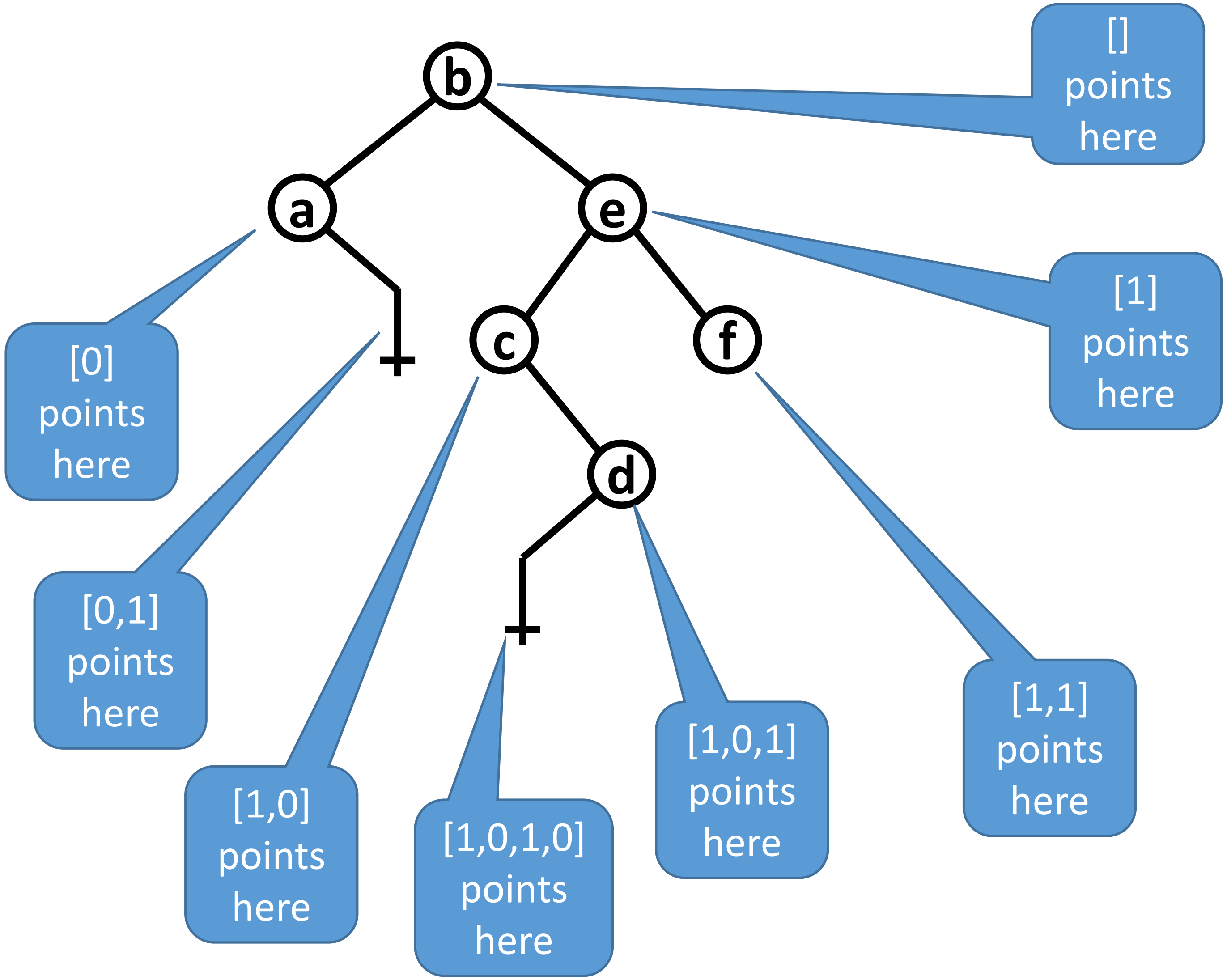
Value sequences of trees

- Each tree t defines a finite sequence of values
  TreeSeq(t)
- The empty tree, t=BSTEmpty, gives the empty sequence,
  TreeSeq(t)=[]
- A tree with a root, t=BSTNode(A,x,B), gives the sequence
  TreeSeq(t)=TreeSeq(A)+[x]+TreeSeq(B)
- The value sequence of a tree is the sequence of the values in the tree in “inorder“ order

Tree paths within trees

- If t is a tree and p is a finite sequence of 0 and 1 then p may be a tree path within t which then points to some subtree of t
- The empty sequence [] is a treepath within every tree t and points to the tree t, which is a subtree of itself
- If a tree has a root, t=BSTNode(A,x,B) then the sequence [0]+p is a tree path within t if and only if p is a tree path within A and the subtree that [0]+p points to within t is then the subtree within A that p points to
- If a tree has a root, t=BSTNode(A,x,B) then the sequence [1]+p is a tree path within t if and only if p is a tree path within B and the subtree that [1]+p points to within t is then the subtree within B that p points to
- If p is a tree path within the tree t we use Subtree(t,p) to denote the subtree within t that p points to

Examples of tree paths



Many tree paths, more than half, point to empty subtrees. Every leaf, for example, has two empty subtrees.

Here the paths [0,1] and [1,0,1,0] are taken as examples of paths that point to empty subtrees, but other paths within this tree point to empty subtrees, for example [0,0], [1,1,0] and [1,0,1,1].

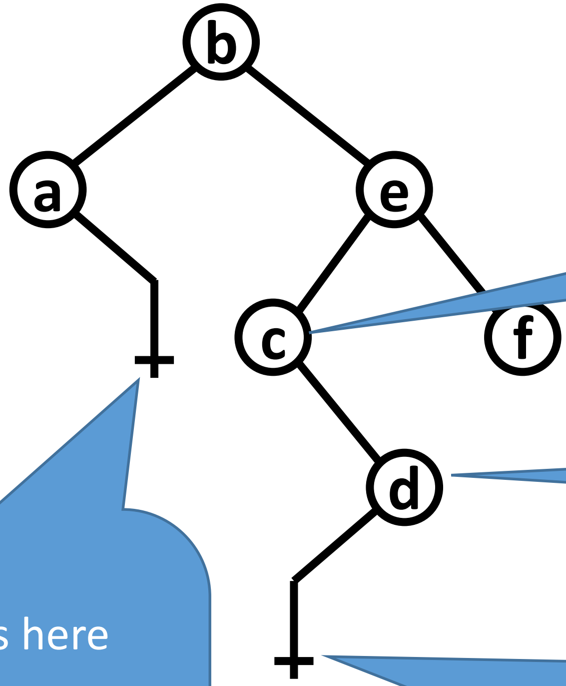
The value sequence of this tree is [a,b,c,d,e,f].

Tripartitioning the value sequences of trees

- For each tree t every tree path p within t defines a tripartition of the value sequence of the tree
- Define PreSeq(t,p) as the sequence of values of those nodes, inorder, that are to the left of the subtree that p points to
- Define PostSeq(t,p) as the sequence of values of those nodes, inorder, that are to the right of the subtree that p points to
- Define MidSeq(t,p) as the sequence of the values, inorder, that are in the subtree that p points to
- It can be proven (in Dafny) that for every tree path p within a tree t we have

  TreeSeq(t) = PreSeq(t,p)+MidSeq(t,p)+PostSeq(t,p)

Examples of tripartitioning a tree t



[1,0] points here

PreSeq(t,[1,0])=[a,b]
MidSeq(t,[1,0])=[c,d]
PostSeq(t,[1,0])=[e,f]

[1,0,1] points here

PreSeq(t,[1,0,1])=[a,b,c]
MidSeq(t,[1,0,1])=[d]
PostSeq(t,[1,0,1])=[e,f]

[0,1] points here

PreSeq(t,[0,1])=[a]
MidSeq(t,[0,1])=[]
PostSeq(t,[0,1])=
[b,c,d,e,f]

[1,0,1,0] points here

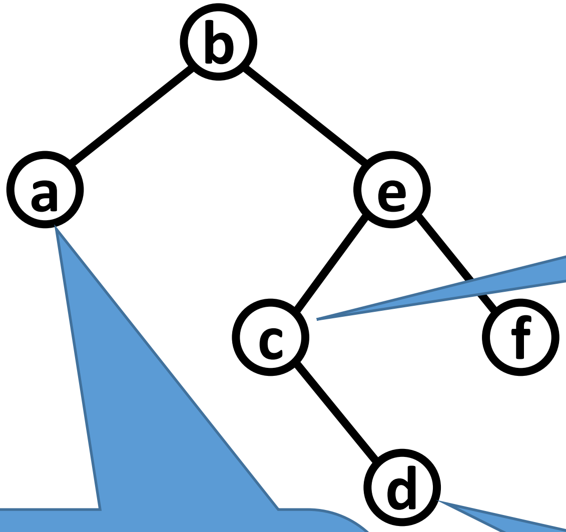
PreSeq(t,[1,0,1,0])=[a,b,c]
MidSeq(t,[1,0,1,0])=[]
PostSeq(t,[1,0,1,0])=[d,e,f]

Bipartitioning the value sequences of trees

- For each tree t every tree path p within t, which points to a non-empty subtree, defines a bipartitioning of the value sequence of the tree
- Define PreSeqExcluding(t,p) as the sequence of the values of the nodes, inorder, that are to the left of the node that p points to, excluding that node
  - The last value of the sequence (if the sequence is non-empty) is the value of the node, if any, that is immediately before the node inorder
- Define PostSeqIncluding(t,p) as the sequence of the values of the nodes, inorder, that are to the left of the node that p points to, including that node
  - The first value in the sequence is then the value in the node that p points to
- It can be proven (in Dafny) that for each such tree path p within a tree t we have

  TreeSeq(t) = PreSeqExcluding(t,p)+PostSeqIncluding(t,p)

Examples of bipartitioning a tree t



[1,0] points here

PreSeqExcluding(t,[1,0])=[a,b]
PostSeqIncluding(t,[1,0])=[c,d,e,f]

[0] points here

PreSeqExcluding(t,[0])=[]
PostSeqIncluding(t,[0])=
[a,b,c,d,e,f]

[1,0,1] points here

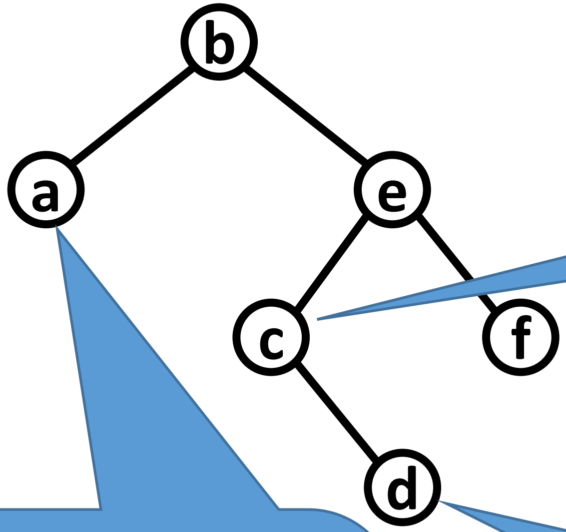
PreSeqExcluding(t,[1,0,1])=[a,b,c]
PostSeqIncluding(t,[1,0,1])=[d,e,f]

Another bipartitioning of the value sequences

- For each tree t every tree path p within t, which points to a non-empty subtree, defines another bipartitioning of the value sequence of the tree
- Define PreSeqIncluding(t,p) as the sequence of the values of the nodes, inorder, that are to the left of the node that p points to, including that node
  - The sequence is non-empty and the last value of the sequence is the value of the node that p points to
- Define PostSeqExcluding(t,p) as the sequence of the values of the nodes, inorder, that are to the left of the node that p points to, excluding that node
- It can be proven (in Dafny) that for each such tree path p within a tree t we have

  TreeSeq(t) = PreSeqIncluding(t,p)+PostSeqExcluding(t,p)

Another example of bipartitioning a tree t



[1,0] points here

PreSeqIncluding(t,[1,0])=[a,b,c]
PostSeqExcluding(t,[1,0])=[d,e,f]

[0] points here

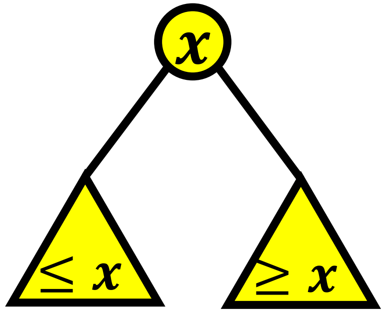
PreSeqIncluding(t,[0])=[a]
PostSeqExcluding(t,[0])=
[b,c,d,e,f]

[1,0,1] points here

PreSeqIncluding(t,[1,0,1])=[a,b,c,d]
PostSeqExcluding(t,[1,0,1])=[e,f]

Binary search trees (BST)

- Binary search trees are binary trees such that the values in the tree are ascending from left to right
- In other words, the values in the tree inorder are in ascending order
- In other words, a tree t is a binary search tree if TreeSeq(t) is in ascending order
- In other words, a tree is a binary search tree if for all nodes we have that all values in the left subtree are $\leq$ the value in the node and all values in the right subtree are $\geq$ the value of the node

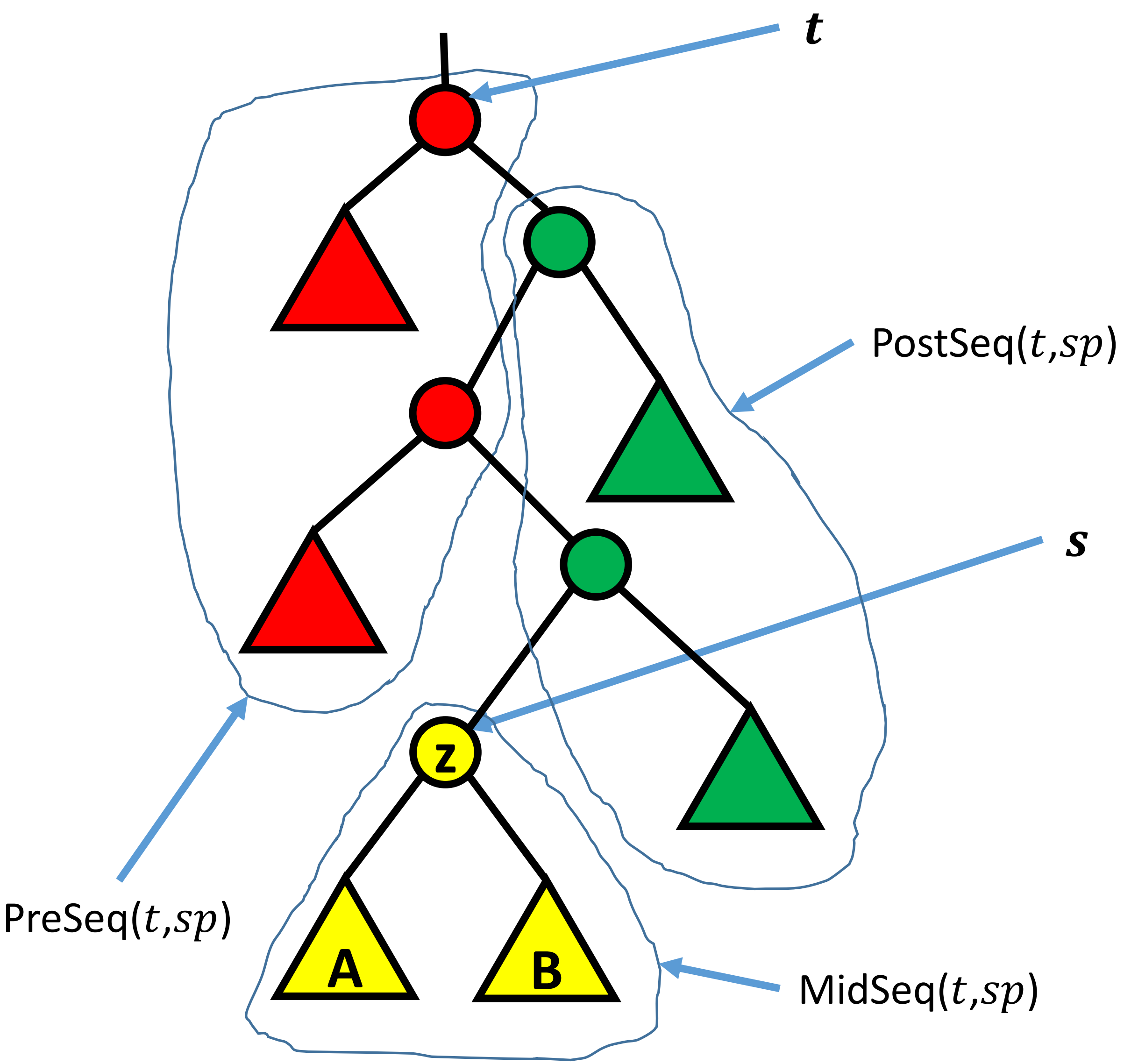


Loop invariant for searching for some $= x$

- Looking for some node within binary search tree t with value x
- Use a loop and have variables s and p (p may be a ghost variable) such that we maintain the invariant that
  - s = Subtree(t,p)
  - All values in PreSeq(t,p) eru $< x$
  - All values in PostSeq(t,p) eru $> x$
- The loop is terminated if s is empty (x then does not exist in the tree) or if the root of the subtree s has the value x

PreSeq(t,p)	**MidSeq(t,p) =Treeseq(s)**	**PostSeq(t,p)**
$< x$	**?**	$> x$

Figurative example state



The next step is to check whether the root of the tree s should be red ($\mathrm{z} < x$) or green ($\mathrm{z} > x$), or whether the value is found ($\mathrm{z} = x$)

if $\mathrm{z} > x$ then z and B become green:
$s := A$

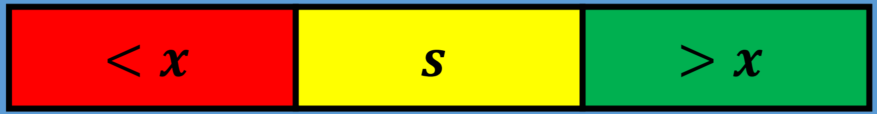
If $\mathrm{z} < x$ then z and A become red:
$s := B$

sp is the tree path within t that points to the subtree s.

In Dafny (without verification)

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
   var s := t;
   while s != BSTEmpty {
      if RootValue(s) > x {
         s := Left(s);
      } else if RootValue(s) < x {
         s := Right(s);
      } else {
         return s;
      }
   }
   return BSTEmpty;
}
```

Loop invariant: The middle area (yellow) stands for the value sequence of the subtree s.



Note that the first area (red) is PreSeq(t,sp) and the last area (green) is PostSeq(t,sp).

Tail-recursive version

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
  if t == BSTEmpty {
    r := BSTEmpty;
  } else if x < RootValue(t) {
    r := SearchAnyEQ(Left(t),x);
  } else if x > RootValue(t) {
    r := SearchAnyEQ(Right(t),x);
  } else {
    r := t;
  }
}
```

Equivalent

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )
{
  match t
  case BSTEmpty =>
    { r := BSTEmpty;                        }
  case BSTNode(left,val,right) =>
    {
      if x < val
        { r := SearchAnyEQ(left,x);  }
      else if x > val
        { r := SearchAnyEQ(right,x); }
      else
        { r := t;                    }
    }
}
```

Pre- and postconditions

```
method SearchAnyEQ( t: BST, x: int ) returns ( r: BST )

  requires TreeIsSorted(t)

  ensures x in TreeSeq(t) ==>
    r != BSTEmpty &&
    IsSubTree(t,r) &&
    x == RootValue(r)

  ensures x !in TreeSeq(t) ==>
    r == BSTEmpty
```

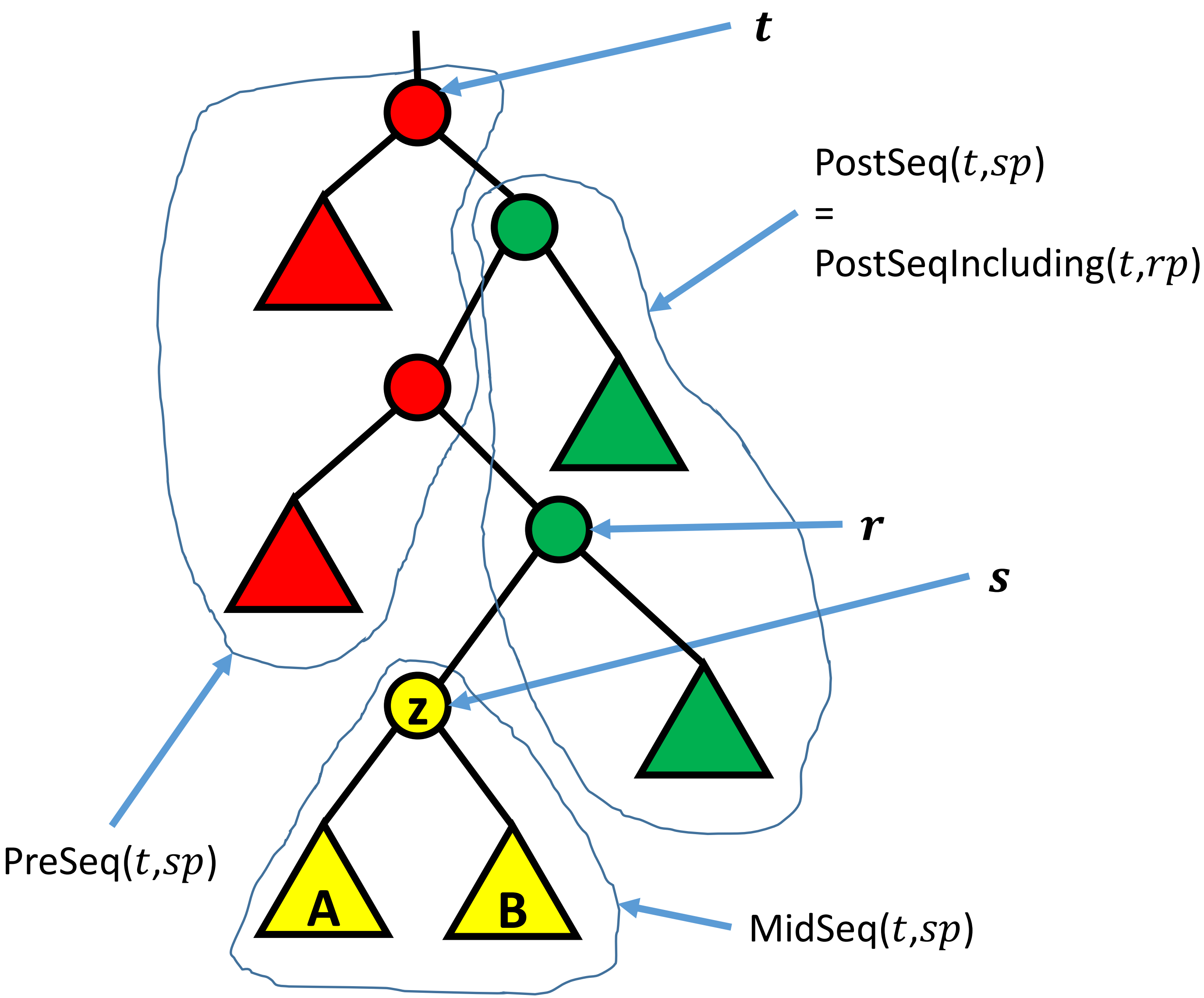
Loop invariant for searching for leftmost $\geq x$

- Looking for leftmost node in a tree t with value $\geq x$
- Use a loop and have variables s, r, sp and rp (sp and rp may be ghost variables) such that we maintain the invariant that
  - s = Subtree(t,sp)
  - All values in PreSeq(t,sp) eru $< x$
  - All values in PostSeq(t,sp) eru $\geq x$
  - If PostSeq(t,sp) is not empty then r points to a non-empty subtree of t such that r=SubTree(t,rp) and PostSeqIncluding(t,rp)=PostSeq(t,sp)
    - In other words, r points to the leftmost node to the right of the subtree s
  - If PostSeq(t,sp) is empty then r is an empty tree
- The loop is terminated when s is empty and then r is returned

PreSeq(t,sp)	**MidSeq(t,sp)=Treeseq(s)**	**PostSeq(t,sp)**
$< x$	**?**	$\geq x$

r

Figurative example state



The next step is to check whether the root of the tree s should become red ($z < x$) or green ($z \geq x$).

If $z \geq x$ then r is updated and z and B become green:

$$r := s$$
$$s := A$$

If $z < x$ then z and A become red but r is unchanged:

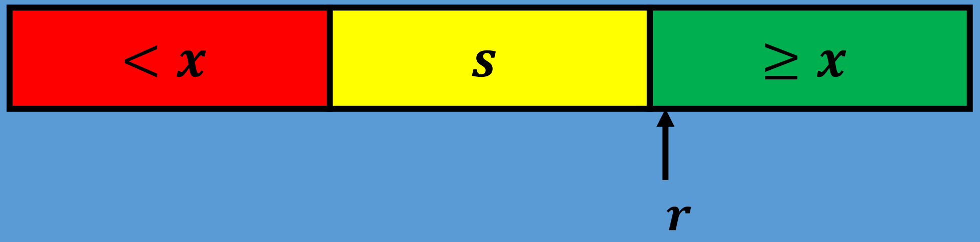
$$s := B$$

sp is the tree path within t that points to the subtree s, rp is the tree path within t that points to the subtree r (if it exists).

In Dafny (without verification)

```
method SearchFirstGELoop( t: BST, x: int )
     returns ( r: BST )
{
  var s := t;
  r := BSTEmpty;
  while s != BSTEmpty {
    if RootValue(s) >= x {
      r := s;
      s := Left(s);
    } else {
      s := Right(s);
    }
  }
}
```

Loop invariant: The middle area (yellow) stands for the value sequence of the subtree s. r points to the first node in the last area (green) if any, otherwise we have r=BSTEmpty



Note that the first area (red) is PreSeq(t,sp) and the last area (green) is both PostSeq(t,sp) and PostSeqIncluding(t,rp) where sp is the tree path pointing to s and rp is the tree path pointing to r (if it exists).

In Dafny (tail-recursive without verification)

```
method SearchFirstGERecursiveHelp( t: BST, x: int, c: BST )
  returns ( r: BST )
{
  if t == BSTEmpty {
    r := c;
  } else if RootValue(t) < x {
    r := SearchFirstGERecursiveHelp(Right(t),x,c);
  } else {
    r := SearchFirstGERecursiveHelp(Left(t),x,t);
  }
}
```

The function returns a tree r that has a root that is the leftmost node within t that has a value $\geq x$, or returns the tree c if no such node exists within t.
The tree c is therefore the default return value if no node $\geq x$ is dound in t.

The call
`SearchFirstGERecursiveHelp(t,x,BSTEmpty)`
will then return the correct tree given the problem definition

Equivalent

```
method SearchFirstGERecursiveHelp( t: BST, x: int, c: BST )
           returns ( r: BST )
{
  match t
  case BSTEmpty =>
    { r := c;                                           }
  case BSTNode(left,val,right) =>
    {
      if val < x
        { r := SearchFirstGERecursiveHelp(Right(t),x,c); }
      else
        { r := SearchFirstGERecursiveHelp(Left(t),x,t);  }
    }
}
```

Pre- and postconditions

```
method SearchFirstGE( t: BST, x: int ) returns ( r: BST )

  requires TreeIsSorted(t)

  ensures (forall z | z in TreeSeq(t) :: z < x) ==>
    r == BSTEmpty

  ensures (exists z | z in TreeSeq(t) :: z >= x) ==>
    r != BSTEmpty &&
    (exists rp | IsTreePath(t,rp) ::
       r == Subtree(t,rp) &&
       (forall z | z in PreSeqExcluding(t,rp) :: z < x) &&
       (forall z | z in PostSeqIncluding(t,rp) :: z >= x)
    )
```

Loop invariant for searching for rightmost $< x$

- Looking for rightmost node in tree t with value $< x$
- Use a loop and have variables s, r, sp and rp (sp and rp may be ghost variables) such that we maintain the invariant that
  - s = Subtree(t,sp)
  - All values in PreSeq(t,sp) are $< x$
  - All values in PostSeq(t,sp) are $\geq x$
  - If PreSeq(t,sp) is not empty then r points to a non-empty subtree of t such that r=SubTree(t,rp) and PreSeqIncluding(t,rp)=PreSeq(t,sp)
    - in other words, r points to the rightmost node to the left of the subtree s
  - If PreSeq(t,sp) is empty then r is the empty tree
- The loop is terminated when s is empty and r is then returned

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$< x$	**?**	$\geq x$

r

Loop invariant searching for leftmost $> x$

- Looking for leftmost node inn tree t wiith value $> x$
- Use a loop and have variables s, r, sp and rp (sp and rp may be gost variables) such that we maintain the invariant that
  - s = Subtree(t,sp)
  - All values in PreSeq(t,sp) are $\leq x$
  - All values in PostSeq(t,sp) are $> x$
  - If PostSeq(t,sp) is not empty then r is a non-empty subtree of t such that r=SubTree(t,rp) and PostSeqIncluding(t,rp)=PostSeq(t,sp)
    - in other words, r refers to the leftmost node to the right of the subtree s
  - If PostSeq(t,sp) is empty then r is the empty tree
- The loop is terminated when s is empty and then r is returned

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$\leq x$	**?**	$> x$

r

Loop invariant searching for rightmost $\leq x$

- Looking for rightmost node in tree t with value $\leq x$
- Use a loop and have variables s, r, sp and rp (sp and rp may be ghost variables) such that we maintain the invariant that
  - s = Subtree(t, sp)
  - All values in PreSeq(t,sp) are $\leq x$
  - All values in PostSeq(t,sp) are $> x$
  - If PreSeq(t,sp) is not empty then r points to a non-empty subtree of t such that r=SubTree(t,rp) and PreSeqIncluding(t,rp)=PreSeq(t,sp)
    - in other words, r points to the rightmost node to the left of the subtree s
  - If PreSeq(t,sp) is empty then r is the empty tree
- The loop is terminated when s is empty and then r is returned

PreSeq(t,sp)	**MidSeq(t,sp) =Treeseq(s)**	**PostSeq(t,sp)**
$\leq x$	**?**	$> x$

r