Binary Search Trees

Michael P. Fourman

February 2, 2010

Introduction

Many algorithms make use of datastructures that represent *dynamic sets,* that is, a collection of elements that can grow, shrink, or otherwise change, over time. Stacks, queues, priority queues, and dictionaries may all be viewed as dynamic sets. If algorithms are to make use of dynamic sets without efficiency worries, it is important that the appropriate data structures are carefully chosen. The choice of implementation may be affected by the particular types of element involved, and by the relative frequencies of different operations being performed on the dynamic set. In this note we introduce *search trees,* which support a variety of operations on dynamic sets.

Search Trees

A heap is a vertically-ordered tree; a search tree is horizontally ordered. A *binary search tree* is a binary tree whose nodes are labelled by items in such a way that in-order traversal of the tree gives an ordered list of items. Searching for an item in a search tree is an $O(h)$ operation, where h is the height of the tree. Balanced trees are important because the height of a balanced tree is $O(\mathrm{lgn})$, where n is the number of nodes in the tree. In this section we look at functions to insert and retrieve elements from a binary search tree without worrying about keeping the tree balanced. Techniques for balancing will be covered later.

Recall the type declaration for a binary tree. We declare it in a signature for later use.

```
signature BTreeSig =
sig datatype 'a Tree = Lf
  | Nd of 'a Tree * 'a * 'a Tree
end
```

We will implement sets of items as search trees of type `Item tree`, where the type `Item` is equipped with an ordering, `<`. A binary tree is a search tree if, and only if, for each internal node `Nd(lt, v, rt)`, every label in the left subtree, `lt` is less than `v`, and every label in the right subtree, `rt`, is greater than `v`.

The basic idea is to build into our data-structure the divide-and-conquer strategy used in algorithms like quicksort and mergesort. In the quicksort algorithm, we use a pivot to divide the sorting problem into two independent parts. In a binary search tree, each internal node divides the data-structure into two independent parts. We place smaller items in the left sub-tree, and larger items in the right sub-tree. An in-order traversal of a binary search tree gives an ordered list.

Here is a function, based on our first implementation of quicksort, that builds a binary search tree from a list.

```
local
    fun divide x (h :: t) =
        let val (low, high) = divide x t
        in if x < h then (low, h :: high)
                    else (h :: low, high)
        end
      | divide _ [] = ([],[])
in
    fun mkTree (h :: t) =
        let val (x, y) = divide h t
        in Nd( mkTree x, h, mkTree y )
      | mkTree [] = Lf
end
```

We can picture the action of quicksort by building this tree, and then producing an in-order traversal—except that in quicksort we don't actually bother to *build* the tree. It may pay to build the tree, in order to implement operations, such as `member`, that involve searching. The function to look for a given element may be written as

```
fun member Lf              k = false
  | member (Nd(lt, k', rt) k =
      if k  < k' then member lt k
      else
      if k' < k  then member rt k
      else true (* k = k' *) ;
```

The cost of a call to `member` is bounded by the height of the tree; if the tree is balanced, this is $O(\lg n)$. The work invested in building the tree is $O(n \lg n)$.

If we only expect to make $O(\lg n)$ calls to `member`, we might as well use a list (with an $O(n)$ implementation of `member`) to represent our set. Otherwise, the investment is probably worthwhile.

Insertion To insert a new element, we replace a leaf by a tree with a node containing the new element, and two leaves. We recurse down the tree to find the appropriate position for the new leaf.

```
fun insert (e, Lf) = Nd(Lf, e, Lf)
  | insert (e, Nd(lt, r, rt)) =
      if      e < r then Nd(insert(e, lt), r, rt)
      else if r < e then Nd(lt, r, insert(e, rt))
      else Nd(lt, r, rt) (* e = r *)
```

Note that the function returns a new tree, leaving the original unchanged. When we try to insert an element, we may find it already in the tree, in which case the tree we return is equal to the original tree. However, the recursion still rebuilds the tree. We can use exceptions to optimise this case by *really* returning the original tree.

```
local
exception NoChange
fun ins (e, Lf) = Nd(Lf, e, Lf)
  | ins (e, Nd(lt, r, rt)) =
      if      e < r then Nd(ins(e, lt), r, rt)
      else if r < e then Nd(lt, r, ins(e, rt))
      else raise NoChange
in
fun insert(e, t) = ins(e, t) handle NoChange => t
end
```

Maximum We can use a binary search tree to implement a priority queue. The largest label in the tree must be found at the end of the right-most branch.

```
fun getmax (Nd(lt, v, Lf)) = (lt, v)
  | getmax (Nd(lt, v, rt)) =
    let val (r, m) = getmax rt
     in (Nd(lt, v, r), m) end
```

This could be used as an implementation of `deq`, but this isn't a very good way to implement a priority queue. A search tree used as a priority queue will tend to become unbalanced. The main reason for introducing the `deq` operation is that we will use it in our implementation of the set operation, `delete`.

Deletion The delete operation is more interesting. The entry to be deleted may occur anywhere in the tree, we must be able to re-constitute a binary search tree from the remainder. Fortunately, it suffices to consider only one case. If we can write a function `join` to re-constitute a binary search tree from the two orphan children that remain when we remove the root node of a tree, we can implement `delete` as follows:

```
fun delete(e, Lf) = Lf
  | delete(e, Nd(lt, v, rt)) =
      if      e < v then Nd(delete(e, lt), v, rt)
      else if v < e then Nd(lt, v, delete(e, rt))
      else join lt rt
```

If either of the children is a leaf, joining is trivial; otherwise we have to ensure that we maintain the correct ordering of nodes within the joined tree. The two children are quite special: all the values in `lt` come before all the values in `rt`. Before we can join them, we must remove one value to place at the root of the new tree. We can then place the remainder of the two subtrees to the left and right of this element. The partitioning element may either be the largest value in the left child, or the smallest value in the right child. In our implementation of `join`, we use `deq` to remove the largest member of the left child.

```
fun join Lf x = x
  | join x Lf = x
  | join lt rt =
    let val (l, m) = rmmax lt
     in Nd(l, m, rt) end
```

An implementation of several set operations is provided by the functor `TREESET` given in Figure 1.

A binary search tree can also be used to support dictionary operations, as shown in Figure 2. We implement a dictionary as a search tree of `Key * Item` pairs. `TREESET` provides most of the operations, but we need to access the representation directly to implement `lookup`. (C) Michael Fourman 1994-2006

```
functor TREESET( structure T : BTreeSig
                    type Item
                    val < : Item * Item -> bool ) =
struct
local
open T
exception NoChange

fun ins (e, Lf) = Nd(Lf, e, Lf)
  | ins (e, Nd(lt, v, rt)) =
      if      e < v then Nd(ins(e, lt), v, rt)
      else if v < e then Nd(lt, v, ins(e, rt))
      else raise NoChange

fun getmax (Nd(lt, v, Lf)) = (lt, v)
  | getmax (Nd(lt, v, rt)) =
    let val (rt', m) = getmax rt
     in (Nd(lt, v, rt'), m) end
  | getmax Lf = raise NoChange

fun join Lf x = x
  | join x Lf = x
  | join lt rt =
    let val (l, m) = getmax lt
     in Nd(l, m, rt) end

fun del(e, Lf) = raise NoChange
  | del(e, Nd(lt, v, rt)) =
      if      e < v then Nd(del(e, lt), v, rt)
      else if v < e then Nd(lt, v, del(e, rt))
      else join lt rt
in
type Item = Item and Set = Item Tree
val empty = Lf
fun isEmpty Lf = true | isEmpty _ = false

fun member  Lf              k = false
  | member (Nd(t1, k', t2)) k =
      if k  < k' then member t1 k
      else
      if k' < k  then member t2 k
      else true;

fun insert(e, t) = ins(e, t) handle NoChange => t

fun delete(e, t) = del(e, t) handle NoChange => t
end
end
```

Figure 1: Sets based on a Search Tree

```
functor TREEDICT( structure T : BTreeSig
                  type Key
                  val < : Key * Key -> bool
                  type Item ) : DictSig =
let structure TS =
    TREESET(structure T = T
            type Item = Key * Item
            val op < = fn ((k, _), (k', _)) => k < k' )
in
struct
open T
exception Lookup

type Dict = (Key * Item) Tree
type Key  = Key
type Item = Item

val empty = TS.empty
val enter = TS.insert

fun lookup  Lf                   k = raise Lookup
  | lookup  (Nd(lt, (k',e), rt)) k =
      if k  < k' then lookup lt k
      else
      if k' < k  then lookup rt k
      else e ;

fun remove(k, d) = let val e = lookup d k in
                       TS.delete((k,e), d)
                   end handle Lookup => d
end
end
```

Figure 2: Dictionary based on a Search Tree