Binary Search Trees

Jonathan Turner

April 11, 2013

Binary search trees are used in a wide variety of applications. In particular, they can be used to implement Edmonds algorithm for max weight matching in general graphs and the dynamic trees data structure used to speed up Dinic's algorithm. Here, we will use binary search trees to represent a collection of sets, where the set elements are drawn from some base set, with each item appearing in exactly one set. Each item has a unique *key* and the data structure implements the following operations.

- *setkey*(i, k) initializes the key of item i to k; i must be a singleton.
- *access*(k, s) return the item in set s having key k, or **null** if there is no such set.
- *insert*(i, s) inserts item i into s; i must be a singleton.
- *delete*(i, s) removex item i from s; i becomes a singleton.
- *join*(s_1, i, s_2) returns the set formed by combining s_1, i and s_2; all items in s_1 must have keys less than *key*(i) and all items in s_2 must have keys greater than *key*(i); this operation replaces s_1 and s_2 with the new set.
- *split*(i, s) divides the set s containing i into three sets: s_1 containing items with keys less than *key*(i), $\{i\}$ and s_2 containing items with keys greater than *key*(i); it returns the pair $[s_1, s_2]$.

Each set in the data structure is represented by a binary search tree, with each element in the underlying base set corresponding to a node in some tree (possibly a single node tree). Each item x has a pointer to a left child *left*(x), a right child *right*(x) and a parent $p(x)$. Any of these may be **null**. A common implementation trick uses an explicit **null** node with appropriatedly defined fields, to reduce the number of boundary cases that

must be handled explicitly. The key property of a binary search tree is that for all nodes x

$$left(x) \neq \textbf{null} \Rightarrow key(left(x)) < key(x)$$
$$right(x) \neq \textbf{null} \Rightarrow key(right(x)) > key(x)$$

This allows us to quickly locate an item in the tree using its key as illustrated by the *access* operation shown below.

```
item function access(key k, set s)
    do s ≠ null and k < key(s)  ⇒  s := left(s)
    |  s ≠ null and k > key(s)  ⇒  s := right(s)
    od;
    return s
end;
```

A **null** return value indicates that the specified key is not present in the tree.

The *insert* operation works similarly, first searching for the location where the key of the item to be inserted belongs, then adding a node for the item at that location. Figure 1 shows an example of a binary search tree and the effect of an insert operation.

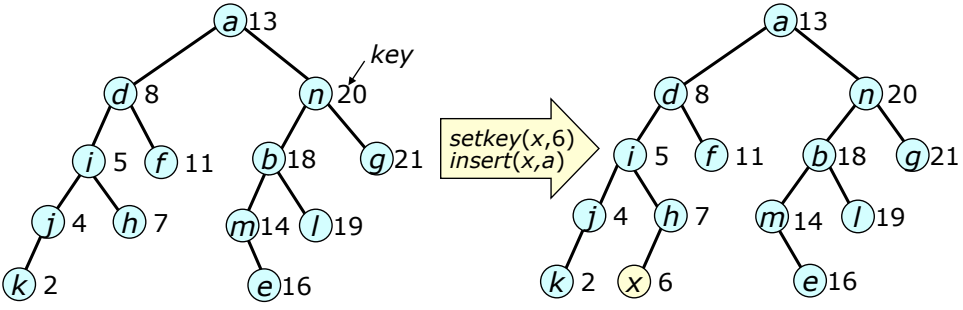

Figure 1: Binary search tree example

An implementation of the *insert* operation appears below.

```
sorted set function insert(item i, sorted set s)
    item x; x := s;
    if s = null  ⇒  return i fi;
    do key(i) < key(x) and left(x) ≠ null  ⇒  x := left(x)
    |  key(i) > key(x) and right(x) ≠ null  ⇒  x := right(x)
```

```
    od;
    if key(i) = key(x) ⇒ return null
    | key(i) < key(x) ⇒ left(x) := i;
    | key(i) > key(x) ⇒ right(x) := i;
    fi;
    p(i) := x left(i), right(i) := null;
    return s;
end;
```

In this implementation, the *swapplaces* method exchanges the positions of i and j in the tree.

There are several distinct cases for the *delete* operation. If the node for the item i to be deleted, has no children, we can simply remove the node from the tree. If it has a single child, we can move that child up into the position currently occupied by i. If i has two children things get more complicated. First, we identify the item j with the largest key that is $\leq key(i)$. The node for this item will be in the left subtree of i. More specifically, it will be the rightmost node in the left subtree, as illustrated in Figure 2. Once we have

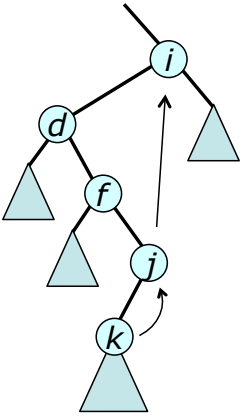

Figure 2: Example of deletion

found the node for item j, we move it up to the position originally occupied by i and move its original child (if any) into the position that it originally occupied. An implementation of this procedure appears below.

```
procedure delete(item i, set s)
    item j;
    if left(i) ≠ null and right(i) ≠ null ⇒
        j := left(i);
```

```
        do right(j) ≠ null ⇒ j := right(j) od;
        swapplaces(i, j);
    fi;
    if left(i) = null ⇒ left(i) ↔ right(i) fi;
    p(left(i)) := p(i);
    if i = left(p(i)) ⇒ left(p(i)) := left(i)
     | i = right(p(i)) ⇒ right(p(i)) := left(i)
    fi;
    left(i), right(i), p(i) := null;
end;
```

The *join* and *split* operations are useful in a variety of applications, including the dynamic trees data structure used by some network flow algorithms, and Edmonds algorithm for maximum weight matchings in general graphs. The *join* operation has a simple implementation, as illustrated in Figure 3.

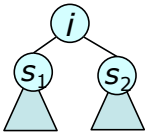

Figure 3: Join operation

An implementation appears below.

```
sorted set function join(sorted set s1, item i, sorted set s2);
    left(i) := s1; right(i) := s2;
    p(s1), p(s2) := i;
    return i;
end;
```

The *split* operation moves up the tree from the item i on which the split is taking place. As it goes, it builds the two sets s_1 and s_2 that eventually get returned. The initial values of s_1 and s_2 are the left and right subtrees of i. As the operation moves up the tree, it combines the "next node" x with either s_1 or s_2, depending on whether i was in its left or right subtree. This process is illustrated in Figure 4 for the case where i was in the left subtree of x.

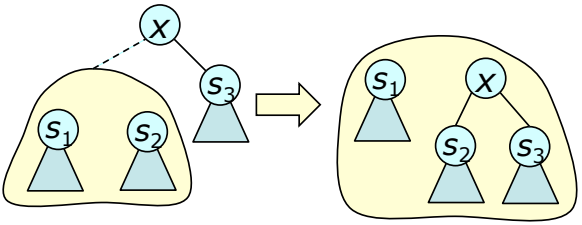

Figure 4: Split operation

After the operation, x is the root of the new s_2. An implementation of the *split* operation appears below.

```
[sorted set, sorted set] function split(item i, sorted set s);
    sorted set x, s1, s2;
    x := p(i); s1, s2 := left(i), right(i);
    bit leftchild := (i = left(x));
    do x ≠ null ⇒
        if leftchild ⇒ s2 := join(s2, x, right(x))
         | not leftchild ⇒ s1 := join(left(x), x, s1)
        fi;
        leftchild := (x = left(p(x)));  x := p(x)
    od;
    left(i), right(i), p(i) := null;
    p(s1), p(s2) := null;
    return [s1, s2];
end;
```

In addition to the operations described above, there are others that can be useful in some contexts. For example, we might define *first* and *last* operators that identify the items with the smallest or largest keys in a set. These can be easily implemented by following left pointers in one case, or right pointers in the other. The operation $next(x)$ returns the item with the next larger key. If x has a right subtree, the next item is found at the leftmost node in x's right subtree. If x has no right subtree, the next item is found by following parent pointers, and stopping at the first node that contains x in its left subtree.