Multiway Search Trees

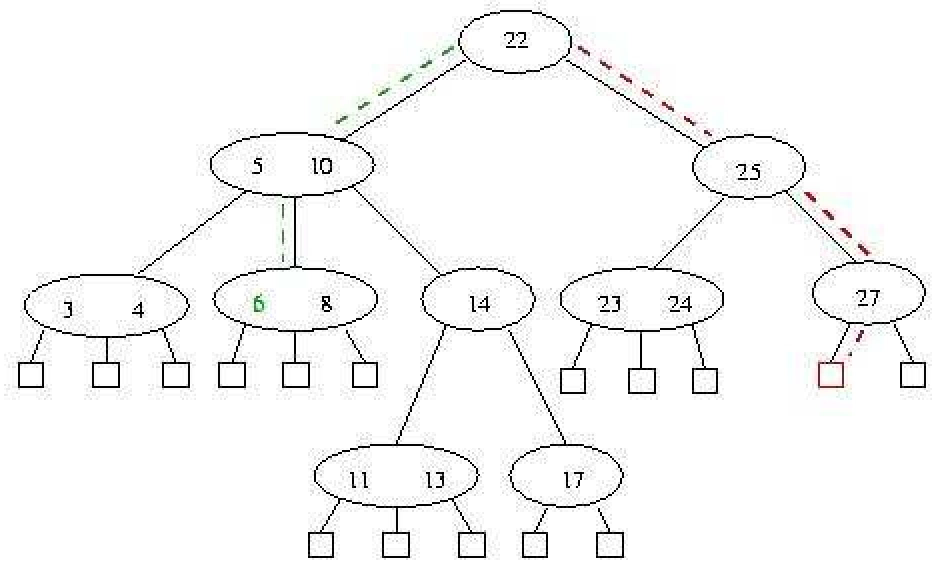
Intuitive Definition

A multiway search tree is one with nodes that have two *or more* children. Within each node is stored a given key, which is associated to an item we wish to access through the structure.
Given this definition, a binary search tree is a multiway search tree.

More Formal Definition

Let *T* be a multiway search tree, then *T* has the following properties:

- *T* is ordered, meaning that the all the elements in subtrees to the left of an item are less than the item itself, and all the elements in subtrees to the right of an item are greater.
- Each internal node of *T* has at least 2 children.
- Each *d*-node (node with *d* children) *v* of *T*, with children $v_1,...,v_d$ stores *d*-1 items $(k_1, x_1),...,(k_{d-1}, x_{d-1})$. Where the k_i's are keys and x_i is the element associated with key number *i*.
- External nodes are empty



```
A multiway search tree with a successful search path for the
number 6 (in green), and an unsuccessful search path for the
                     number 26 (in red)
```

Definition of a (2,4)-tree

A (2,4)-tree is simply a multiway search tree (as defined above) that also satisfies the following properties:

I. `SIZE:` every node can have *no more* than 4 children.
II. `DEPTH:` all external nodes have the same depth.

Assuming that we are able to maintain these properties (which still remains to be seen!), then we can deduce a couple of useful properties of this structure:

1. if follows from the the `SIZE` property that the number of items at each node is less than or equal to 4. Hence d_{max} is constant and our search time is already down to O(*h*)!
2. luckily, the `DEPTH` property ensures that the tree is balanced, but also that the height is restricted to THETA(log*n*) (where *n* is the number of nodes in the tree). Click here if you want to see the proof

Observations 1 and 2 imply that the worst case search time in a (2,4)-tree is O(log*n*) **under the assumption that we can efficiently maintain properties I and II**.

So we've got to see if we can insert and delete items from this tree efficiently without disturbing these properties. This will be the topic of the next two sections.

[Top] [Bottom] [Home]

6.7 Insertion

In this section we will show that:

- The `SIZE` and `DEPTH` depth properties of (2,4)-trees can be maintained upon insertion of a new item.
- The maintenance cost is bounded above by the height of the tree

6.7.1 The insertion algorithm

Let's begin with a basic algorithm for insertion and work from there. We would like to `INSERT` a key *k* into a (2,4)-tree *T*. Here are the steps we follow:

1. perform a `SEARCH` for *k* in *T*.
   if it succeeds then we don't need to `INSERT` the item and we're done,
   otherwise (if it fails) then the search terminates at an external node *z*.
2. let *v* be the parent node of *z*, insert *k* into the appropriate place in *v* and add a new child *w* to *v* on the left of *z* (see `figure 6.2(a)`).

And that's IT (well...essentially)! Since the procedure did not change the depth of *T*, the tree is balanced and we have in no way violated the `DEPTH` property, but what about the `SIZE` property?

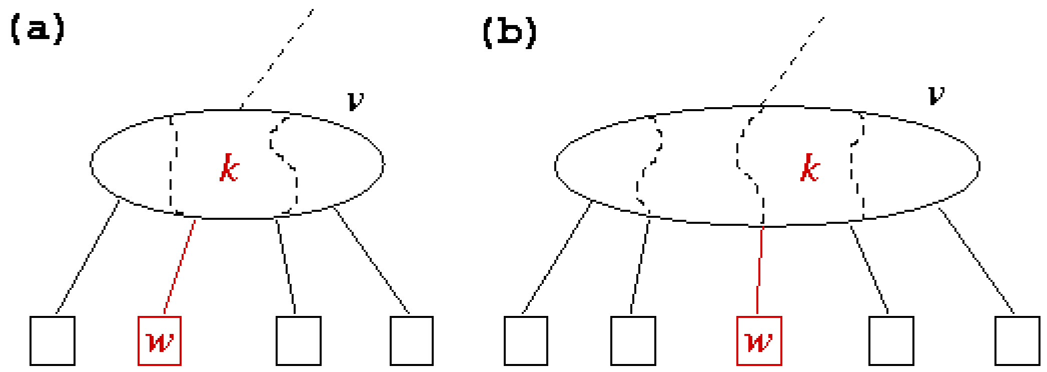


Figure 6.2: insertion of new key *k* resulting in new node *w* (in red) (a) with no overflow, (b) with an overflow at node *v*

Consider what would happen if *v* already had 4 children before we inserted *k* into it (see figure 6.2(b)). After insertion *v* would now be a 5-node thereby violationg the size property of *T*! This is called an **overflow** at node *v*, and it must be resolved in order for our algorithm to be valid for (2,4)-trees.

We fix this little glitch by `SPLITTING` *v* into two smaller nodes as follows (figure 6.3): Let $v_1,...,v_5$ be the children of *v* (which stores keys $k_1,...,k_4$),

1. Split *v* by replacing it with *v'* (a 3-node that stores keys k_1 and k_2) and *v''* (a 2-node that stores k_4).
2. Store k_3 in what *was* the parent of *v* (if *v* was the root we create a new node and store it in there). Call that node *u*.
3. make *v'* and *v''* the children of *u*. (if *v* was the *i*th child of *u*, then *v'* and *v''* become the *i*th and *(i+1)*st children of *u*).

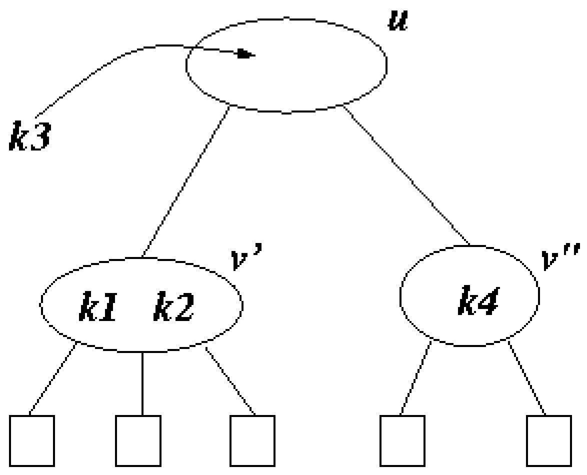


Figure 6.3: local state of *T* after the node *v* has been split into *v'* and *v''*.

First observe that this procedure has perfectly taken care of the overflow situation at node *v*, but has potentially *created* an overflow at node *u* (since *u* now has had one child added to it)! In this case we would simply repeat the `SPLITTING` procedure at node *u*.

Notice also that this procedure will eventually terminate at a node that doesn't overflow or at the root (where we create a brand new node that obviously doesn't overflow).

Finally, you may have spotted the fact that the we change the depth of the tree when we create a new root. However this doesn't violate our `DEPTH` property since every node in the tree's depth increases by 1 (hence the tree stays balanced).

6.7.2 Analysis of insertion

- We began with a search procedure which take O(log*n*) time.
- Next we insert the key into *v* in O(1) time.
- Finally, we had a maximum of O(*h*) split operations to maintain the `SIZE` property. As we showed in section 6.6, *h* is THETA(log*n*). Hence we have at worst case O(log*n*) splits.
- Each split affects a constant number of items in a constant number of nodes of *T* and hence takes O(1) time.

Therefore, insertion can be performed in (2,4)-trees in O(log*n*) time (where *n* is the number of nodes in the tree).

[Top] [Bottom] [Home]

6.8 Deletion

In the previous section, we saw that the `SIZE` and `DEPTH` properties of (2,4)-trees can be maintained efficiently as new items are inserted into the tree. We now show that the same result holds as items are removed.

6.8.1 The deletion algorithm

Once again we'll begin with a basic algorithm that we'll adjust. We want to `DELETE` a key *k* from *T*. For now, we shall assume that *k* is stored in a node *v* whose children are *all external nodes*. Here are the steps we follow:

1. perform a `SEARCH` for *k* in *T*,
   if it fails then we don't need to `DELETE` the item so we exit,
   otherwise (if it succeds) then we find *k* in a node *v* with only external children (by our assumption).
2. **now we simply remove *k* from *v* and delete the external node child to the left of *k* (see `figure 6.4(a)`).**

Simple enough. Since we only deleted external nodes, we didn't change the depth of *T*, so that property is safe. However, we may violated the `SIZE` property once again.

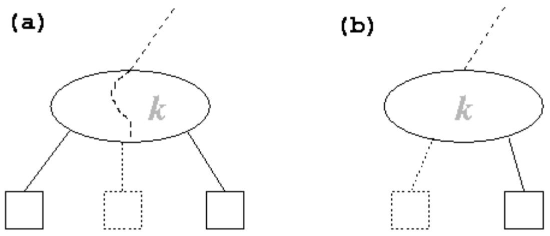


Figure 6.4: deletion of key *k* resulting in (a)two children remaining in *v* so no underflow, (b) underflow at node *v* because it is left with only one child

This is because (as shown in `figure 6.4(b)`) *v* may have only had two children before we removed *k*. In such a case, it will be left as a 1-node, which violates the definition of a multiway search tree (see section 6.2.2). This situation is called an underflow at node *v*. Again, we're going to have to resolve this in order to validate our deletion algorithm.

Let *u* be the parent of *v*. To solve this problem, we consider two seperate cases:

I. ***v* has a sibling *w* that is a 3-node or a 4-node.**
   In this case we perform a `TRANSFER` operation as follows (`figure 6.5`):
   a. **Move a child of *w* to *v*.**
   b. **Move a key from *w* to *u*.**
   c. **Move a key from *u* to *v*.**

   This may seem a little cryptic, but one look at the figure should make this quite clear. As you can see, this operation has the following effects:

- **It adds a child to *v* and removes one from *w* (thereby making *v* a 2-node and *w* a 2 or 3-node, resolving the underflow at *v*).**
- **But now we must transfer the corresponding keys without destroying the ordered nature of the tree. We do this by pushing the extra key from *w* through the appropriate key in the parent *u*, and finally into *v*.**
- **The net effect is that the number of keys in *w* has been reduced by 1, and the number of keys in *v* has been increased by 1 as desired!**

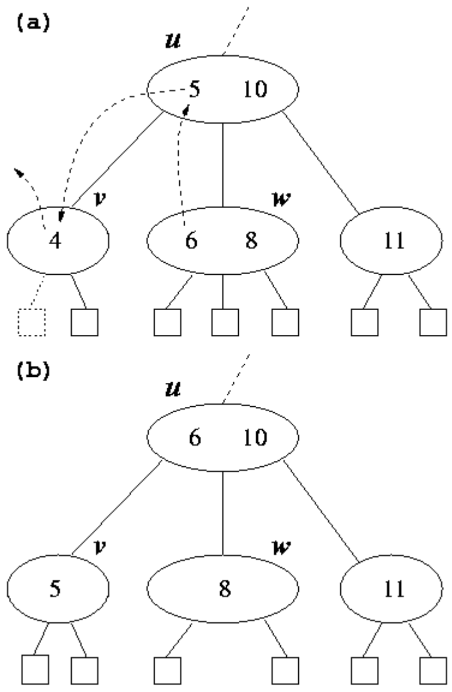


Figure 6.5: deletion of key 4 resulting in an underflow at *v*. (a) transfer operation, (b) the resulting tree after the transfer.

II. ***v* has no such siblings (i.e., they are all 2-nodes).**
This case requires a `FUSION` of two nodes as follows ([figure 6.6]()):
. **Merge *v* with a 2-node sibling *w* creating a new node *v'*.**
a. **Move a key from *u* (*v*'s parent) to *v'*.**

After step (a), *v'* has 3 children, but stores only one key, and *u* has lost a child (since two of it's children merged into one), hence we move a key from *u* to *v'* to preserve the tree.

Note that the `FUSION` operation reduces the number of *u*'s children by 1, potentially causing an underflow at *u*. In such a case we would remedy this with another `FUSION` or `TRANSFER`.

Observe also that this process also terminates at the root (an underflow at the root simply causes its deletion), and is hence bounded above by O(log*n*) as was the `SPLIT` operation in the `INSERTION` algorithm.

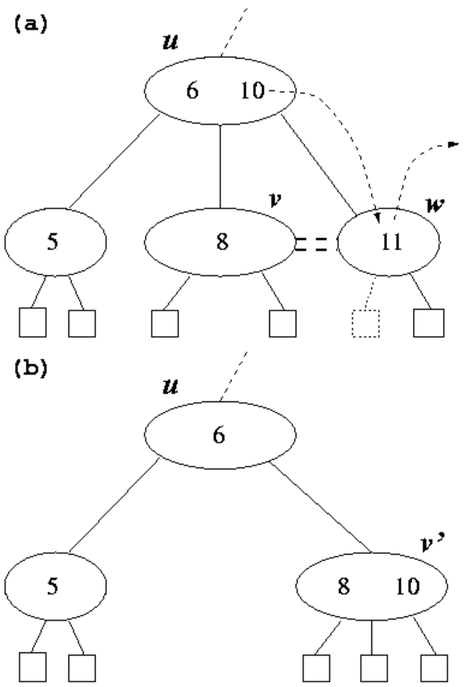


Figure 6.6: deletion of key 11 resulting in an underflow at *v*. (a) fusion of *v* and *w*, (b) the resulting tree after the fusion.

Almost done! However, remember that everything above only applies if the key we're trying to remove is stored at a node with only external nodes for children (if you don't remember making this assumption click here)! What do we do if this is not the case? Well, there just so happens to be a very easy way to swap any key in a (2,4)-tree with one that is stored in a node with the property we desire without destroying the ordering in the tree. This is done as follows:

Swap the internal key k_i with the largest element in the subtree immediately to its left. By the definition of a search tree, this key is the next smallest key in the tree next to k_i. So once we delete k_i, the tree will be correctly ordered.

A natural question to ask would be "how do we find this element?" This is accomplished by performing the following steps:

1. **Let *v* be the internal node in which the element we wish to delete (k_i) is stored.**
2. **Let *w* be the right-most internal node in the subtree rooted at the *i*th child of *v*.**
3. **Swap k_i with the last item of *w*.**

Once this is done, the item we wish to delete will be at a node that has only external nodes for children, and we will be able to use the above procedure to delete it.

6.8.2 Analysis of deletion

- **We began with a search procedure which takes O(log*n*) time.**
- **This may be followed with a swap which also takes O(log*n*).**
- **Finally, we had a maximum total of O(log*n*) `TRANSFER` or `FUSION` operations (each of which takes constant time).**

Therefore, deletion can also be accomplished in (2,4)-trees in O(log*n*) time.