Deletion: The Curse of the Red-Black Tree

Functional Pearl

Kimball Germane Matt Might
University of Utah

Abstract

Red-black trees were born in the imperative world and only adopted into the functional. Okasaki's exposition of the simplicity of a functional implementation actually felt more like a rebirth than mere adoption. With this regime shift came a shift in focus: the energy that would otherwise have been spent on pointer manipulation is now applied to establishing correctness with formal methods–a step forward, to be sure–but one wonders if there is a simple, obviously-correct, functional delete. Indeed there is.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

General Terms red-black tree, delete, data structure

Keywords red-black tree, delete, data structure

1. Introduction

When looking for a data structure to back a functional implementation of sets, red-black trees–a type of balanced binary search tree–are a natural choice. Common set operations, such as membership testing and persistent addition, map naturally to their native operations of search and insertion. And, speaking of maps, minor modifications can turn a set membership test into a map lookup operation and set addition into map extension.

The usefulness of red-black trees stems from their balanced nature. A red-black tree is a binary tree in which each node is colored red or black, and which satisfies the local property that

1. every red node has two black children,

and the global property that

2. every path from the root to a leaf[1] node contains the same number of black nodes.

These conditions guarantee that the longest path from root to leaf can be no more than twice the shortest (the only difference being individual red nodes interspersed along the way), and so the worst-case penalty of the search for an element in a red-black tree search is a reasonable constant factor over that of a a perfectly balanced tree.

[1] For our purposes, leaf nodes do not contain data and are colored black.

there hasn't been a satisfactory method to delete items; enforced with types leads to Byzantine code, proven correct is good if the language is a target for extraction, but above the capabilities of typical programmers

we've encountered this situation before: Okasaki gave a functional rendition of tree insertion and a smashing intuition at the same time. the intuition, from which the code can be derived, is what we're after: we want a fairly simple, obviously correct way to perform tree deletion.

THREE THINGS TO REASON ABOUT: ORDERING, LOCAL, GLOBAL

INSERTION

to review and as an example to introduce our reasoning methods, we go over insertion.

2. Insertion

Okasaki [**?**] made functional red-black trees accessible by presenting a clear method for element insertion by means of the recursive `insert` function and the `balance` helper function.

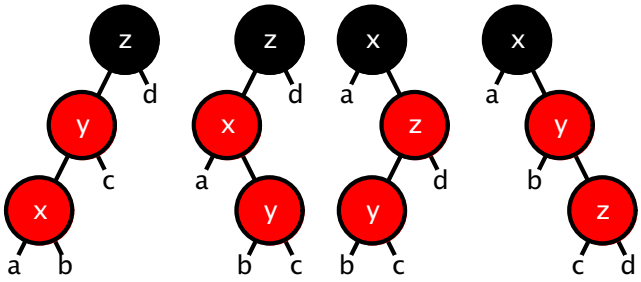
Using a bit of syntactic sugar on top of Racket, we can express the `insert`[2] function with

```
(define (insert t k)
  (match t
    [(L) (R (L) k (L))]
    [(N c a x b)
     (switch-compare
       (k x)
       [< (balance (N c (insert a k) x b))]
       [= (N c a x b)]
       [> (balance (N c a x (insert b k)))])]))
```

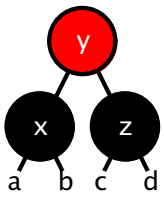
where `balance` yet remains undefined.

This definition of `insert` unilaterally colors each newly-added node red, but this may create a problem: if the parent of this new node is red, this action violates the local property. Okasaki persists in the face of this possibility, reasoning that it is easier to resolve a violation of the local property than the global one. This is achieved by the `balance` function which takes trees of one of the following four forms

[2] We use the variable `k` to denote the value to insert to avoid later confusion.



and transforms them into



When performing local tree transformations, we must justify the correctness of the transformation, which entails the consideration of the local property–that red nodes must have two black children– and the global property–that every path from the root to a leaf must have the same number of black nodes.

We can verify that this transformation indeed resolves red-red violations by observing that the subtrees a, b, c, ande d are compatible with their newly-assigned parents, no matter the colorings of their roots. However, consider the parent of each of the four cases. If the parent is red, then this transformation *introduces* exactly the same kind of red-red violation it resolved!, We can reason about the effect this transformation has on the local property by considering the possible colorings of the root of each subtree a, b, c, and d. If the parent of this node is red, this node must be black and there are no restrictions on its placement in the reconstruction. If the parent is black, this node is possibly red, and the assignment of a red parent possibly introduces a red-red violation. The `balance` function is designed to resolve precisely this situation. We can verify this transformation preserves the global property by verifying that any possible introduction of this violation is subjected to `balance`.

Furthermore, we can reason locally about the effect this transformation has on the *global* property by considering the number of black nodes this portion of the tree contributes to each path that travels through it. We can see that this transformation preserves the global property by verifying that the paths that reach the subtrees $a - d$ accumulate the same number of black nodes both before and after it occurs.

The purpose of this transformation is to resolve a violation of the local property, and while it does resolve one, it potentially introduces another further up the tree. We can treat this violation the same way and even with the same function; hence, the `balance` function is applied preventively at each level of the recursion which has the effect of working up the tree as the semantic stack unwinds. This transformation cannot possibly introduce a red-red violation at the root of the tree, being the child of no node, so the correctness of the entire process follows from a simple inductive argument.

The final stop of Okasaki's insertion algorithm is to blacken the root of the tree, which is benign in all cases. This requires a small modification of `insert` to

```
(define (insert t k)
  (define (ins t k)
    (match t
      [(L) (R (L) k (L))]
      [(N c a x b)
       (switch-compare
         (k x)
         [< (balance (N c (insert a k) x b))]
         [= (N c a x b)]
         [> (balance (N c a x (insert b k)))])]))
  (blacken (ins t k)))
```

with `blacken` given by

```
(define/match blacken
  [(N _ a x b) (N 'B a x b)])
```

Our formulation of the red-black invariants allows trees to have red roots in some cases, so our root-coloring policy is more conservative, only blackening if the red-black construction demands it. This leads only to a change to `blacken`, which is now

```
(define/match blacken
  [(R (R? a) x b) (B a x b)]
  [(R a x (R? b)) (B a x b)]
  [t t])
```

3. Deletion

Insertion in binary trees has the advantage that new nodes are added only to the fringe, whereas deletion might also target interior nodes. With deletion, we only have to be slightly clever to reduce the latter situation to the former: when deleting a value that resides in an interior node, replace that node's value with the minimum value of its right subtree, and delete that value from that subtree.[3] This strategy contains a reference to deletion itself, so, like insertion, this approach can be defined recursively. Roughly, we express it by

```
(define (delete t k)
  (match t
    [(N c a x b)
     (switch-compare
       (k x)
       [< (N c (delete a k) x b)]
       [= (let ([v (min-element b)])
            (N c a v (delete b k)))]
       [> (N c a x (delete b k))])]))
```

Essentially, this algorithm first locates the given value with a simple binary search and then applies our strategy, invoking itself. By enhancing this approach to account for red-black properties, we obtain a sound, persistent method of deletion from red-black trees!

We start by considering the genuine base cases of the delete algorithm: the configurations that don't entail a node with a right subtree from which we can extract the minimum element.

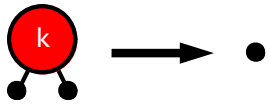
- If the value to delete is not present in the tree, the search will terminate on an empty tree. This presents no difficulty: the empty tree remains unchanged after the removal of any element.

As a case for the `delete` function, this can be written

```
[(L) (L)]
```

[3] An alternative is to distinguish left subtrees and use the maximum element. By considering right subtrees, we get a `min-element` function for free, which is critical for priority queues.

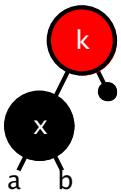
- Because red nodes do not contribute to the height of the tree, we can soundly remove them from the bottom. Therefore, a single red node becomes the empty tree.



As a case for the `delete` function, this is

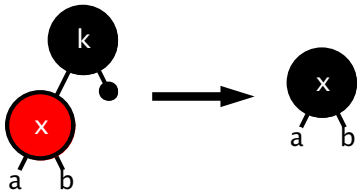
```
[(R (L) (== k) (L)) (L)]
```

- A red node with a black-rooted left subtree



violates the global property and cannot occur.

- A black node with a red-rooted left subtree becomes the subtree itself, only black-rooted.



Conceptually, we are only removing a single red node from the tree. Absent subtree merging, this is completely straightforward. As a case for the `delete` function, this is

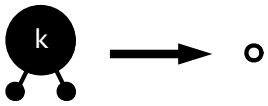
```
[(B (R a x b) (== k) (L)) (B a x b)]
```

- Finally, a black node with no left or right subtree

presents us with a challenge. The paths that end at one of its leaves accumulate two black nodes from this portion of the tree–one for the node itself and one for the leaf. Thus, the careless excision of it would violate the global property by altering the height of the tree. Repurposing some wisdom from Okasaki, perhaps we should attempt to preserve the global property at the expense of something more local. We do this by introducing the double-black color (`BB`) of which both branches

and leaves

can be classified. For accounting purposes, a double-black node contributes two black nodes to any path that travels through it. With this intuition, it is obvious what a lone black node becomes after deletion.



Of course, once this substitution is made, we no longer have a red-black tree, and must reconcile our newly-created tree with the red-black properties. Having adopted Okasaki's initial approach, it seems only natural to apply the rest of it, if possible.

Recall that `insert` adds a red node to the tree, possibly introducing a red-red violation, and that `balance` resolves red-red violations locally, possibly introducing one higher in the tree. Because `insert` is recursive, it can apply `balance` at each level, pulling red-red violations to the root where they can be resolved unequivocally.

We now find wisdom in formulating `delete` recursively, as we can apply the same strategy. Instead of introducing red-red violations to be resolved by `balance`, we introduce double-black nodes to be discharged by `rotate`.

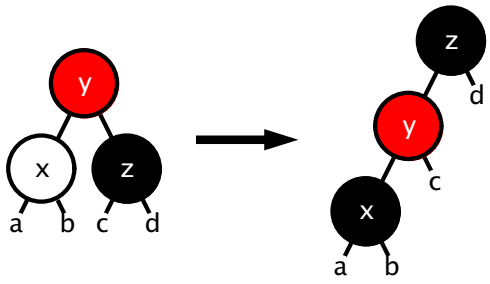
Our `delete` is now defined as

```
(define (delete t k)
  (match t
    [(L) (L)]
    [(R (L) (== v) (L)) (L)]
    [(B (L) (== v) (L)) (BB)]
    [(B (R a x b) (== v) (L)) (B a x b)]
    [(N c a x b)
     (switch-compare
       (k x)
       [< (rotate (N c (delete a k) x b))]
       [= (let ([v (min-element b)])
            (rotate (N c a v (delete b k))))]
       [> (rotate (N c a x (delete b k)))])]))
```

with `rotate` still undefined.

The `rotate` function rearranges trees whose root node has a double-black child and either discharges the double-black node immediately or moves it to the root of the tree. Surprisingly, it need only be applied to three cases and their reflections.

The first case is a red-rooted tree with a double-black child. This condition is sufficient to conclude that the other child is black–not red–and is a node–not a leaf. In this case, the double black node can be discharged immediately with the rotation



We can verify that the number of black nodes this tree contributes to each path through it is unchanged by this rotation, so it doesn't

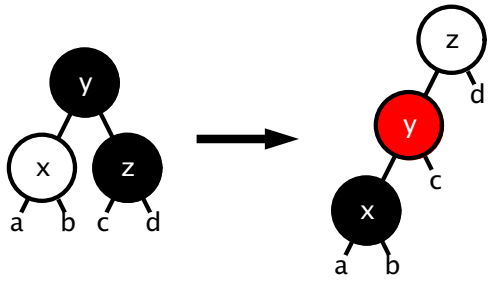
disrupt the global property. However, it possibly introduces a red-red violation. Fortunately, this is no matter: we can `balance` it away! As a case for the `rotate` function, we can express the rotation of this case and its reflection by

```
[(R (BB? a) x (B b y c))
 (balance (B (R (-B a) x b) y c))]
[(R (B a x b) y (BB? c))
 (balance (B a x (R b y (-B c))))]
```

where `BB?` matches a double-black node or leaf without deconstructing it and `-B` demotes a double-black node or leaf to its black counterpart and is undefined for red nodes.

The second case is a black-rooted tree with a double-black child and, necessarily, a black child also. The situation is identical to the previous case but for the additional black node contributed by the root to each path.

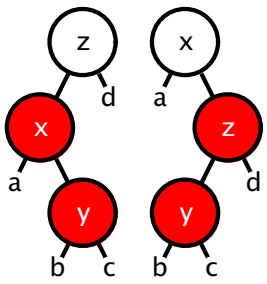
This additional black node prevents us from discharging the double-black node immediately so we defer its resolution by arranging it at the root.



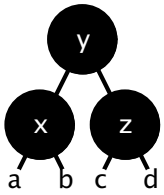
As cases for the `rotate` function, this is

```
[(B (BB? a) x (B b y c))
 (balance (BB (R (-B a) x b) y c))]
[(B (B a x b) y (BB? c))
 (balance (BB a x (R b y (-B c))))]
```

This rotation presents a minor complication: because of its double-black root, the red-red violation can no longer be handled by `balance`. We cope with this by extending `balance` over just these situations. This is as simple as adding cases for trees of the form

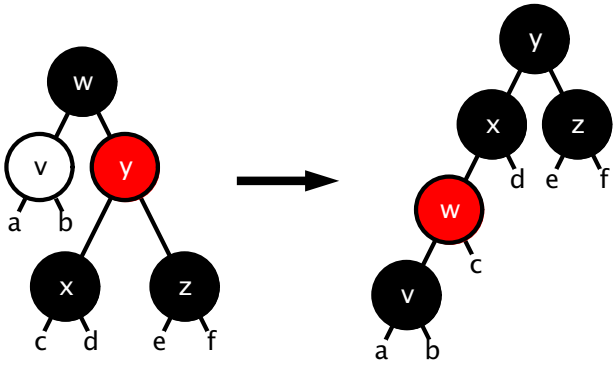


and transforming them each to



Unlike the original cases of `balance`, there is no way that this transformation can introduce a red-red violation; it doesn't introduce any red nodes! In fact, the need to balance is almost preferable here since the operation itself would discharge the double-black node!

The final case is the most difficult, because it requires us to look at more of the tree. Do not be discouraged, however, the tree is

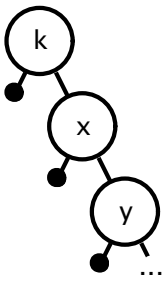



```
(define/match balance
  [(or (B (R (R a x b) y c) z d)
       (B (R a x (R b y c)) z d)
       (B a x (R (R b y c) z d))
       (B a x (R b y (R c z d))))
   (R (B a x b) y (B c z d))]
  [(or (BB (R (R a x b) y c) z d)
       (BB (R a x (R b y c)) z d)
       (BB a x (R (R b y c) z d))
       (BB a x (R b y (R c z d))))
   (B (B a x b) y (B c z d))]
  [t t])

(define min
  (match-lambda
    [(L) (error 'min "empty tree")]
    [(N _ (L) x _) x]
    [(N _ a _ _) (min a)]))

(define -B
  (match-lambda
    [(L2) (L)]
    [(BB a x b) (B a x b)]
    [a (error '-B "unsupported node ~a" a)]))

(define rotate
  (match-lambda
    [(R (BB? a) x (B b y c))
     (balance (B (R (-B a) x b) y c))]
    [(R (B a x b) y (BB? c))
     (balance (B a x (R b y (-B c))))]

    [(B (BB? a) x (B b y c))
     (balance (BB (R (-B a) x b) y c))]
    [(B (B a x b) y (BB? c))
     (balance (BB a x (R b y (-B c))))]

    [(B (BB? a) x (R (B b y c) z (B? d)))
     (B (balance (B (R (-B a) x b) y c)) z d)]
    [(B (R (B? a) x (B b y c)) z (BB? d))
     (B a x (balance (B b y (R c z (-B d)))))]
```

```
    [t t]))

(define blacken
  (match-lambda
    [(R (R? a) x b)
     (B a x b)]
    [(R a x (R? b))
     (B a x b)]
    [t t]))

(define redden
  (match-lambda
    [(B (B? a) x (B? b))
     (R a x b)]
    [t t]))

(define (delete t v cmp)
  (define (del t v cmp)
    (match t
      [(L) (L)]
      [(R (L) (== v) (L))
       (L)]
      [(B (L) (== v) (L))
       (BB)]
      [(B (R a x b) (== v) (L))
       (B a x b)]
      [(N c a x b)
       (switch-compare
        (cmp v x)
        [< (rotate (N c (del a v cmp) x b))]
        [= (let ([v (min b)])
             (rotate (N c a v (del b v cmp))))]
        [> (rotate (N c a x (del b v cmp)))])]))
  (del (redden t) v cmp))
```

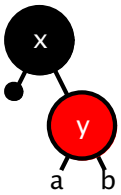
WHEN TALKING ABOUT WHY WE DON'T SPECIAL CASE THE RIGHT TREE STUFF

We have, seemingly for the sake of simplicity, omitted the black-rooted tree with a red-rooted right subtree as a base case of the `delete` function, instead letting this case be handled by the recursive step. When dealing with unrestricted binary trees, this omission alters the time complexity of the algorithm. For, consider the number of invocations of `delete` on

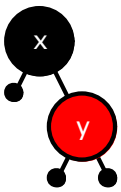


with and without this base case.

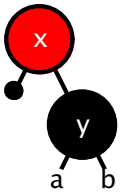
After the initial, each invocation of `delete` targets the minimum value of the given tree. This entails traversing to the leftmost node of the tree and recurring on its right subtree, only bottoming out when it lacks one. In an unrestricted binary search tree, this approach recurs on the entire tree in the worst case! Fortunately, the restrictions of red-black trees impose a tight bound on this its maximum recursion depth when applied to them. Specifically, the global property ensures that the right subtree of



is at worst



and that



cannot occur. This limits the the opportunities for recursion to appear at most twice. The first appears when the candidate value resides in an interior node, and the deletion recurs on the right subtree. The second appears when the minimum element in that subtree resides in a node with a right subtree, and the deletion recurs on that subtree. The global property ensures that this subtree is a singleton, so the recursion is cut off at this point.

following the wisdom of Okasaki, we will take the approach of maintaining the global property at the expense of the local. we do this by temporarily introducing a double black node into the tree for accounting purposes. considering a local portion of the tree, we perform tree rotations in an attempt to discharge the black node. there are only three cases, with their reflections.

in the first case, with a red node as parent, we are able to discharge the double-black node. the inner subtree of the black node could be red, so this may introduce a local red-red violation. we have just the function to handle this in `balance`! because the parent node of this tree began as red, a balance operation will not introduce a red-red violation higher in the tree as it can with insertion. for the same reason, no violation occurs if the balance is unnecessary.

the second case is when both the parent and sibling nodes are black. in this case, we cannot discharge the black node locally and must propagate it up the tree. like the first case, we may introduce a red-red violation here, but, because of the placement of the double-black node, cannot resolve it with the `balance` function. our solution is to extend the balance function to handle a double-black root. just as with a regular black root, we have four cases to consider

cases

each of which can be resolved by transforming it into this tree.

resolved-case.

In addition to maintaining the global property and immediately resolving a local violation, there is no way that we can introduce a local violation. Furthermore, we are able to discharge the double black node by balancing!

the last case is the most complex: a black parent and a red sibling. in this case, we rely on the fact that the red sibling must have nonempty left and right subtrees, and that they must be black rooted. by also considering these nephews of the double-black node, we can discharge the double black node immediately. there is once again the possibility that we introduce a red-red violation, but it is in a subtree of the result. we cope with this by balancing this subtree during construction of the rotation.

note that if the parent or sibling node is red, the double-black node can be discharged immediately.

like the insertion algorithm, the recursive nature of delete allows us to work our way up the tree as we unwind, rotating if necessary. we might wonder if a double-black node is ever propagated to the root. if it were, the remedy would be simple: color the root node black if necessary as is done in the insertion algorithm. this is completely satisfactory, but requires us to expose the `blacken` function to the transient double-black leaf and node. in order to contain it as much as possible, we perform the dual operation when deleting as when inserting; that is, instead of blackening the root node if necessary to complete insertion, we redden the root node if possible to prepare deletion. if it is not possible, then the root node must have at least one red child, and so, in any case, there will be red slack in the tree with which to discharge the node.

CONCLUSION

Okasaki gave intuition to insertion in a functional paradigm, which made red-black trees accessible, albeit limited. A simple, intuitive, obviously correct deletion method might do the same thing and remove the limitation.

Used in this way, red-black trees are efficient, persistent, and still leave us wanting.

Deletion from red-black trees is notoriously complex. In the imperative world, programmers must contend with a bevy of cases and must concern themselves with pointer manipulation. In the functional world, that same attention is devoted to correctness whether it be enforced by types [cite Kahrs] or proved by formal methods [cite Appel]. A functional implementation of deletion from red-black trees should enjoy an intuition.

Much of the simplicity of Okasaki's explanation comes from the simplicity of going from the diagrams to

```
(define/match balance
  [(or (B (R (R a x b) y c) z d)
       (B (R a x (R b y c)) z d)
       (B a x (R (R b y c) z d))
       (B a x (R b y (R c z d))))
   (R (B a x b) y (B c z d))]
  [t t])
```

4. Deletion

Maybe move.

In addition to introducing a functional treatment of tree insertion, Okasaki succeeded in adding intuition to something generally considered to be intricate and error-prone. The net effect of his contribution is not code so short that it is easily memorizable but a concept so elegant and natural that, from it, the code is easily derivable–very befitting of its pearl namesake. As a result, the attitude of intricacy and proneness to error has shifted to tree deletion. While we grant that deletion seems to be more complex than insertion, we hope to show that it is only slightly so, and that, with the right intuition, it can enjoy the same status as insertion.

Much like insertion, we have to account for the possibility of a deletion operation introducing a double-black node from the start. Where the insertion operation balances the tree, attempting to resolve red-red violations, the deletion operation rotates the tree, attempting to discharge double-black nodes. Just as the insertion operation possibly passes unbalance up the tree, the deletion operation possibly passes a double-black node up the tree. Finally, both operations "bottom out" at the top of the tree, guaranteeing resolution.

The final step of Okasaki's algorithm unilaterally changes the root node to black.

When inserting an element, our final step is to blacken the root, but only if necessary. Dual to this, when deleting an element, our *first* step is to redden the root, but only if possible. This gives us slack if the ascending rotations happen to make it to the root. (If we are unable to redden the tree without violating the local condition, then at least on of the children must itself be red which puts a red node sufficiently close to the action.)

The delete function first locates the given value in recursive fashion. If we reach a leaf node, the value was not present, and the tree is left unchanged. If it is found at the bottom and colored red, it is soundly removed. If it is colored black, we replace it with a double-black leaf. If we imagine that such a leaf counts for two black nodes, thin this action preserves the global property. Of course, the properties are simply to guarantee specific time complexities, and traversing this node costs no more than traversing any other. This node is a temporary marker to indicate the need for rotation, and will be discharged accordingly.

If the value is found enough away from the fringe, we replace it with the minimum value of its right child and remove the node that contained that. This strategy is convenient for two reasons: First, it allows us to provide or leverage a *min* function, which is useful when implementing priority queues. Second, it allows us to only worry about removing nodes from the bottom of the tree.

We must consider the final case when the value resides in node not on the bottom of the tree, but with no right child. In this case, the global condition constrains the node to be black and its sole child to be red. To satisfy this case, we remove the black node and replace it with its child, colored black. In doing this, we face no danger in violating either constraint.

In the same way that we consider leaf nodes to be black, we consider double-black leaf nodes to be double black.

The presence of a double-black child node indicates the need for a rotation operation. In some cases, the rotation can discharge the double-black node, and no further rotations will be made. In others, the local structure of the tree prevents resolution in its scope, and the double-black node is pushed higher to be treated by another rotation occurring higher in the tree.

We might ask whether a double black node will reach the top before resolution. Such an occurrence would not be fatal since we could soundly demote it to a single-black node once at the root. This would require a minor intrusion of concerns into the *blacken* function, but is fortunately unnecessary. [Recall that] the first step of the deletion algorithm is to redden the root if possible, and that, if it's not possible, the root must have a red child. This is sufficient to guarantee that a double-black node, if it reaches the top, can be discharged there by the natural flow of the algorithm; no exceptions need be made.

In doing so, he overcame one deficiency and exposed another: the delete operation. Multiple strategies have been developed to offer the delete operation, but each has its own weakness.

Suppose we have a basic implementation of a red-black tree in a language that supports variants and pattern matching. Following Okasaki [**?**], such an implementation might look like this:

5. Conclusion

A. Appendix Title

This is the text of the appendix, if you need one.

Acknowledgments

Acknowledgments, if needed.

References

[] P. Q. Smith, and X. Y. Jones. ...reference text...