Assignment 2 - Comparison of BST and Red-Black Tree Set Implementations

Rory Stephenson 300160212 stepherory

14 April 2014

Test Data

In order to test the relative performance of `Set` and `BSet` three sources of data were used:

1. A list containing the `Int`s 1 to 10000.
2. A list containing the `Int`s 1 to 4999.
3. A list containing the `Int`s 5000 to 10000.
4. A randomly generated list containing 10000 `Int`s in the range 1 to 10000.

Tests

The first test was to add the first data set to the two set implementations in order:

- `Set` performance: 14 seconds
- `BSet` performance: 0.41 seconds

The cost for this operation in `Set` has a prohibitively large cost due to the fact that it is traversing every element in the set every time it adds an element because they are all on one path. `BSet`'s performance is a vast improvement because it rebalances the set as it adds items to it such that it is only traversing a small fraction of the set's elements to add a new one.

The second test repeats the first one using the list's random equivalent, data source 4:

- `Set` performance: 0.55 seconds
- `BSet` performance: 0.4 seconds

Here the difference in performance is negligble because the source list is not sorted. This means that the first set implementation is not unbalanced to anywhere near the same extent.

The next two pairs of results come from removing every element from a test with their relevant data sets. The sets being removed were reversed, otherwise the first set implementation only has to remove the root for each removal. It will perform very well in this case, but if this niche case was required for an application, a linked list should be used instead. The reversal was preprocessed to prevent it from influencing the results.

Remove all elements from set of 10000 random `Int`s:

- `Set`: 0.36 seconds
- `BSet`: 0.35 seconds

Remove all elements from set of the `Int`s 1 to 10000:

- `Set`: 25.6 secs
- `BSet`: 0.34 secs

In the first test the differences are negligible because both trees will be of a similar height so searching for the item to remove takes a similar amount of time. The second test shows a large difference in performance. This is again due to the linked list effect which occurs when items are added to the first set implementation in order. This is because every removal has to traverse every element.

Valid

The implementations of valid in `BSet` adds multiple extra cases which must be checked. These checks are O(n) so the increase in computation time is very small. In both instances every element is checked so the linked list does not cause any significant difference in performance. The results for calling valid on sets created from an ordered list of 10000 `Int`s are:

- `Set`: 0.02 secs
- `BSet`: 0.02 secs

Calling valid on sets created from a randomly generated list of 10000 `Int`s ranging from 0 to 10000:

- `Set`: 0.11 secs
- `BSet`: 0.16 secs

The performance for large sets like these is very good. The difference between a random and an ordered list is negligible.

Functions that use folds

The remaining (non-trivial) functions rely on folds. This means that their performance can be assessed as a group. There is an unfortunate drawback in the way that fold is implemented for both types of sets, it occurs in order. This means that if the fold is used to create a set from an existing set the resulting one will be created in order, leading to the linked list problem in the case of the unbalanced `Set`. To demonstrate this problem union is called in each implementation on two sets which come from the second and third datasets respectively:

- `Set`: 2.1 secs
- `BSet`: 0.5 secs

Here it can be seen that the first set implementation's performance suffers because the second list will be added as one long linked list starting from the node which holds 4999.

The second test calls `==` on the result of union:

- `Set`: 2.8 secs
- `BSet`: 0.7 secs

Since `==` works by checking every element in each set is the member of the other set (two way fold), it is affected by the fact that the resulting union in the first set implementation contains a large linked list. Unsurprisingly the balanced implementation does not suffer from this issue.

Conclusion

It is obvious from the results that `BSet` is a far better set implementation that `Set` in almost every use case. The only exception is when you will only ever want to remove the first element of the set (by order) because in `Set` this is always the root of the set is created in order. However a plain linked list will have less overhead and should be used over `Set` if this is the case. `Set` performs similarly to `BSet` when elements are added in random order, otherwise it descends into linked list performance. Even if the data is known to be randomly ordered, `BSet` is still slightly faster and should be used instead of `Set`.