Binary Search Trees

author:

Date:2018-03-16

Chapter 1: Introduction

Analyze and compare the performances of a sequence of insertions and deletions between **unbalanced binary search trees**, **AVL trees** and **splay trees**.

Chapter 2: Data Structure / Algorithm Specification

Structure of a binary tree :

```
struct TreeNode{
    Element key;
    TreeNode *left;
    TreeNode *right;
    TreeNode *parent;
    Value Height;
}
```

Algorithms of Function "Find" :

```
Position Find( Element x, Tree T )
{
    if( T == NULL ) return NULL;
    if( T->key == x ) return T;
    if( T->key > x ) return Find( x, T->left );
    if( T->key < x ) return Find( x, T->right );
}
```

Unbalanced trees :

```
TreeNode Insert(Element x, TreeNode T)
{
    if( T == NULL )
    {
        T = new TreeNode;
        T->key = x;
        T->left = T->right = NULL;
    }
    else if( x < T->key)

            T->left = Insert( x, T->left );
```

```
    else if( x > T->key )
                T->right = Insert( x, T->right );
    return T;
}


TreeNode Delete( Element x, TreeNode T )
{
    if( T == NULL )
        return T;
    else if( x < T->key)
            T->left = Delete( x, T->left );
    else if( x > T->key )
            T->right = Delete( x, T->right );
    else
        {
            if( T->left == NULL )
            {
                Position P = T->right;
                free(T);
                return P;
            }
            else
            {
                Position P = FindMax( T->left );
                T->key = P->key;
                T->left = Delete( P->key, T->left );
            }
        }
    return T;
}
```

AVL trees :

To re-balance the subtree, we have four cases of rotate algorithms.

```
Position SingleLeft( Position k2 )
{
    Position k1;
    k1 = k2->left;
    k2->left = k1->right;
    if(k2->left != NULL)
        k2->left->parent = k2;
    k1->right = k2;
    k1->parent = k2->parent;
    k2->parent = k1;

    k2->height = Max( Height( k2->left ), Height( k2->right ) ) + 1;
    k1->height = Max( Height( k1->left ), Height( k2->right ) ) + 1;

    return k1;
}
```

```
Position DoubleLeft( Position k3 )
{
    k3->left = SingleRight( k3->left );
    return SingleLeft( k3 );
}

static Position SingleRight( Position k2 )
{
    Position k1;
    k1 = k2->right;
    if(k1 == NULL){
        return k2;
    }
    k2->right = k1->left;
    if(k2->right != NULL)
        k2->right->parent = k2;
    k1->left = k2;
    k1->parent = k2->parent;
    k2->parent = k1;

    k2->height = Max( Height( k2->right ), Height( k2->left ) ) + 1;
    k1->height = Max( Height( k1->right ), Height( k2->left ) ) + 1;
    return k1;
}

static Position DoubleRight( Position k3 )
{
    k3->right = SingleLeft( k3->right );
    return SingleRight( k3 );
}
```

To insert a value *x* into a splay tree:

- Perform standard BST insert for x.
- Starting from x, travel up and find the first unbalanced node. Let m be the first unbalanced node, n be the child of m that comes on the path from x to m and m be the grandchild of m that comes on the path from x to m.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with m.

```
TreeNode InsertAVL( Element x, TreeNode T )
{
    if( T == NULL )
    {
        T = new TreeNode
        T->key = x;
        T->height = 0;
        T->left = T->right = NULL;
    }
    else
        if( x < T->key)
        {

            T->left = InsertAVL( x, T->left );
```

```
            if( Height( T->left ) - Height( T->right ) == 2 )
            {
                if( x < T->left->key )
                    T = SingleLeft(T);
                else
                    T = DoubleLeft(T);
            }
        }
        else
            if( x > T->key )
            {
                T->right = InsertAVL( x, T->right );
                if( Height( T->right ) - Height( T->left ) == 2 )
                {
                    if( x > T->right->key )
                        T = SingleRight(T);
                    else
                        T = DoubleRight(T);
                }
            }
    T->height = Max( Height( T->left ), Height( T->right ) ) + 1;
    return T;
}
```

To delete a node W :

- Perform standard BST delete for w.
- Starting from w, travel up and find the first unbalanced node. Let z be the first unbalanced node, y be the larger height child of z, and x be the larger height child of y.
- Re-balance the tree by performing appropriate rotations on the subtree rooted with z.

```
TreeNode DeleteAVL( Element x, TreeNode T )
{
    if( T == NULL )
        return T;
    else if( x < T->key)
        {
            T->left = DeleteAVL( x, T->left );
            if( Height( T->right ) - Height( T->left ) == 2 )
            {
                if( T->right->left == NULL || ( T->right->right != NULL && T->right->left-
>height < T->right->right->height ) )
                    T = SingleRight(T);
                else T = DoubleRight(T);
            }
        }
        else if( x > T->key )
            {
                T->right = DeleteAVL( x, T->right );
                if( Height( T->left ) - Height( T->right ) == 2 )
                {

                    if( T->left->right == NULL || ( T->left->left != NULL && T->left->left-
```

```
>height > T->left->right->height ) )
                        T = SingleRight(T);
                    else
                        T = DoubleLeft(T);
                }
            }
        else
            {
                if( T->left == NULL ){
                    Position P = T->right;
                    free(T);
                    return P;
                }
                else if( T->right == NULL ){
                    Position P = T->left;
                    free(T);
                    return P;
                }
                else if( T->left->height > T->right->height )
                {
                    Position P = FindMax( T->left );
                    T->key = P->key;
                    T->left = DeleteAVL( P->key, T->left );
                }
                else
                {
                    Position P = FindMin( T->right );
                    T->key = P->key;
                    T->right = DeleteAVL( P->key, T->right );
                }
            }
    T->height = Max( Height( T->left ), Height( T->right ) ) + 1;
    return T;
}
```

Splay trees :

A splay tree is a binary search tree. It has one interesting difference, however: whenever an element is looked up in the tree, the splay tree reorganizes to move that element to the root of the tree, without breaking the binary search tree invariant. If the next lookup request is for the same element, it can be returned immediately. In general, if a small number of elements are being heavily used, they will tend to be found near the top of the tree and are thus found quickly.

To insert a value *x* into a splay tree:

- Insert *x* as with a normal binary search tree.
- when an item is inserted, a splay is performed.
- As a result, the newly inserted node *x* becomes the root of the tree.

```
TreeNode InsertSPL( Element x, TreeNode T, Position parent )
{
    if( T == NULL )
    {
```

```
        T = new TreeNode;
        T->key = x;
        T->left = T->right = NULL;
        T->parent = parent;
    }
    else
        if( x < T->key)
        {
            T->left = InsertSPL( x, T->left, T );
        }
        else
            if( x > T->key )
            {
                T->right = InsertSPL( x, T->right, T );
            }
    return T;
}
```

To delete a node *x*, use the same method as with a binary search tree: if *x* has two children, swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children. Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree.

```
TreeNode DeleteSPL( Element x, TreeNode T )
{
    Position P = Find( x, T );
    if(P == NULL)
        return T;
    T = Splay( P, NULL );
    if(T->left == NULL){
        if(T->right != NULL)
            T->right->parent = NULL;
        P = T->right;
        free(T);
        return P;
    }
    P = FindMax(T->left);
    P = Splay( P, T );
    P->right = T->right;
    if(P->right != NULL){
        P->right->parent = P;
    }
    P->parent = NULL;
    free(T);
    return P;
}


TreeNode Splay( Position c, Position T )
{
    Position p = c->parent;
```

```
    if(p == T || p == NULL)
        return c;
    Position g = p->parent;
    if(g == T || g == NULL)
    {
        if(c->key < p->key){
            return Splay( SingleLeft(p), T );
        }
        else
        {
            return Splay( SingleRight(p), T );
        }
    }
    else
    {
        if(g->parent != NULL)
        {
            if(g->parent->left == g){
                g->parent->left = c;
            }
            else g->parent->right = c;
        }
        if(c->key < p->key && p->key < g->key){
            SingleLeft(g);
            return Splay( SingleLeft(p), T );
        }
        else if(c->key > p->key && p->key > g->key){
            SingleRight(g);
            return Splay( SingleRight(p), T );
        }
        else if(c->key > p->key && p->key < g->key){
            return Splay( DoubleLeft(g), T );
        }
        else if(c->key < p->key && p->key > g->key){
            return Splay( DoubleRight(g), T );
        }
    }
}
```

Chapter 3 : Testing Results

- ***Insert N integers in increasing order and delete them in the same order***

Size(k)	BST	AVL	SPLAY
1	3.699	0.614	0.482
2	13.886	1.292	0.933
3	31.243	2.049	1.29
4	57.713	2.693	1.702
5	95.465	3.601	2.14
6	134.547	4.301	2.698
7	188.58	4.929	3.012
8	250.152	5.609	3.626
9	319.714	6.97	3.788
10	395.328	6.916	4.418

- ***Insert N integers in increasing order and delete them in the reverse order***

Size(k)	BST	AVL	SPLAY
1	8.095	0.546	0.284
2	33.56	1.192	0.547
3	75.818	1.816	0.773
4	141.807	2.462	0.967
5	231.247	3.165	1.241
6	340.504	4.05	1.524
7	473.064	4.395	1.748
8	632.216	4.958	1.998
9	805.574	5.581	2.277
10	1009.729	6.199	2.606

- ***Insert N integers in random order and delete them in random order***

Size(k)	BST	AVL	SPLAY
1	0.481	0.96	2.042
2	0.951	1.634	4.516
3	1.493	2.975	7.302
4	2.064	3.815	9.829
5	2.59	4.743	12.724
6	3.4	6.081	15.611
7	3.854	6.843	18.839
8	4.515	8.417	22.569
9	5.247	9.56	25.788
10	5.853	11.348	29.029

Chapter 4 : Analysis and Comments

References

- Wikipedia contributors, "AVL Tree", *Wikipedia, The Free Encyclopedia* https://en.wikipedia.org/wiki/AVL_tree (accessed March 16, 2018).
- Wikipedia contributors, "Splay tree", *Wikipedia, The Free Encyclopedia* https://en.wikipedia.org/wiki/Splay_tree (accessed March 16, 2018).

Author List

Declaration

We hereby declare that all the work done in this project titled "Binary Search Trees" is of our independent effort as a group.

Signatures