Project 2

Name: Xiaowei Tan

Student ID: 69203272

Theoretical Analysis

1. Standard Binary Search Tree

The pseudo code is shown as below:

```
search value:
    x = root;
    while x != null and x.v != value:
        if x.v < value:
            x = x.r;
        else
            x = x.l;
    return x;

insert value:
    x = root;
    y = null;
    z = Node(value);
    while x != null and x.v != value:
        y = x;
        if x.v < value:
            x = x.r;
        else
            x = x.l;
    z.p = y;
    if y == null:
        root = z;
    else if x == null and y.v < value:
        y.r = z;
    else if x == null and y.v > value:
        y.l = z;

delete value:
    x = search(value);
    if x != null:
        if x.l == null and x.r == null:
            delete x;
        else if x.r != null:
```

```
            replace x with x.r;
        else if x.l != null:
            replace x with x.l;
        else
            y = x's inorder succeedor;
            replace y with y.r;
            replace x with y;
```

The time complexity:

- search: $O(lgn)$ time for expected time, and $O(n)$ time for worst time.
- insert: $O(lgn)$ time for expected time, and $O(n)$ time for worst time.
- delete: $O(lgn)$ time for expected time, and $O(n)$ time for worst time.

2. Red Black Tree

The pseudo code is shown below:

```
search value:
    x = root;
    while x != null and x.v != value:
        if x.v < value:
            x = x.r;
        else
            x = x.l;
    return x;

insert value:
    x = root;
    y = nil;
    z = Node(value);
    while x != nil and x.v != value:
        y = x;
        if x.v < value:
            x = x.r;
        else
            x = x.l;
    z.p = y;
    if y == nil:
        root = z;
    else if x == nil and y.v < value:
        y.r = z;
    else if x == nil and y.v < value:
        y.l = z;
    z.l = nil;
    z.r = nil;
    z.color = RED;
    insertFixUp(z);
```

```
insertFixUp z:
    while z.p.color == RED: //z and its p are both red
        if z's uncle's color is red: //case 1
            then change z's grandparent color to black
            z's parent's and its uncle's color to red
            go and check z's grandparent
        else:
            if z and its p are not on the same direction of z's grandparent:
                rotate z'p to make them on the same direction // case 2
            z.p.color = BLACK;  //case 3
            z.p.p.color = RED;
            rotate z's grandparent to make z's parent move up
    root.color = BLACK;

delete value:
    z = search(value);
    if z != nil:
        origColor = z.color;
        if z.l == nil:
            x = z.r;
            replace z with z.r;
        else if z.r == nil:
            x = z.l;
            replace z with z.l;
        else
            y = the inorder succeedor of z
            x = y.r;
            origColor = y.color;
            if y.p == z:
                x.p = y;
            else:
                replace y with x;
                make z.r become y.r;
            replace z with y;
            make z.l become y.l;
            y.color = z.color;
    if origColor == BLACK: // if original color is black, the tree may be
affected by                                          deleting
        deleteFixUp(x);

deleteFixUp x:
    while x != root and x.color is BLACK: //x.color is black then the black on
x's one side                                                  must decrease 1
after replace y with x
        if x is left child of x.p:
            w = x.p.r;
            if w.color is RED: // case 1
                x.p.color = RED;
```

```
                w.color = BLACK;
                left rotate x.p then turn into case2/3/4
            else:
                if w's children are both black: //case 2
                    w.color = RED;
                    x = x.p;
                else:
                    if w.r.color is BLACK: // case 3
                        w.l.color = BLACK;
                        w.color = RED;
                        right rotate w and turn into case 4;
                    w.p.color = BLACK;
                    w.color = RED;
                    left rotate w.p and x = root;
        else:
            exchange left and right and same as the first case
    x.color = BLACK;
```

The time complexity:

- search: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.
- insert: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.
- delete: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.

3. B-Tree

The pseudo code is shown below:

```
search value:
    x = root;
    while(true)
        if x is leaf and x not contains value:
            break;
        if x.keys[index] == value:
            return x, index;
        find index which meets x.keys[index] < value and x.keys[index+1] >
value
            x = x.children[index];
    return null;

insert value:
    if root is full:
        x = Node();
        add root into x.children;
        split(x, 0);
        insertNonNull(x, value);
    else
        insertNonNull(root, value);
```

```
split node, index//split the indexTh child of Node
    x = node.children[index];
    y = Node();
    add x.keys[t] into node.keys
    add x.keys[t+1] to x.keys[2t-1] to y.keys
    add x.children[t+1] to x.keys[2t] to y.children
    add y to node.children[index+1]

insertNonNull node, value
    if node is leaf:
        find the right index in node.keys, and put value into
node.keys[index];
    else:
        find the index that makes node.keys[index] < value and
node.keys[index+1] > value;
        if node.children[index] is full:
            split(node, index);
            if node.keys[index] < value:
                index++;
        insertNonNull node.children[index], value

delete value:
    doDelete root, value;

doDelete node, value:
    if node is root and has no keys:
        root = node.children[0];
        node = root;

    if value in node.keys:
        if node is leaf:
            remove value from node.keys;
        else:
            if node's sibling has more than t's keys:
                copy value's predcessor or successor to node and remove value;
                doDelete node.children[index], predcessor
                or doDelete node.children[index+1], successor;
    else:
        find the index that makes node.keys[index] > value and
node.keys[index+1] < value;
        if node.children[index] has more than t elements:
            doDelete node.children[index], value;
        else if node.children[index]'s sibling has more than t keys:
            copy first in node.children[index+1] or last in
node.children[index-1] into node.keys and move node.keys[index] into
node.children[index].keys;
            doDelete node.children[index], value;
        else
```

```
            merge node.children[index] and one of its sibling;
            doDelete node.children[index], value;
```

The time complexity:

- search: $O(tlg_tn)$ time for expected time, and $O(tlg_tn)$ time for worst time.
- insert: $O(tlg_tn)$ time for expected time, and $O(tlg_tn)$ time for worst time.
- delete: $O(tlg_tn)$ time for expected time, and $O(tlg_tn)$ time for worst time.

4. Treap

The pseudo code is shown below:

```
search value:
    x = root;
    while x != null and x.value != value:
        if x.value > value:
            x = x.l;
        else
            x = x.r;
    return x;

insert value:
    v = insert value using standard binary tree insertion
    v.priority = random()
    while v is not root and v.priority > v.p.priority:
        if v is left child:
            right_rotate(v.p);
        else
            left_rotate(v.p);

delete value:
    v = search(value);
    if v is not null:
        while true:
            if v is leaf:
                doDelete(v);
                break;
            if v.l is null:
                rotate_left(v);
            else if(v.r is not null and v.l.priority < v.r.priority)
                rotate_left(v);
            else
                rotate_right(v);
```

The time complexity:

- search: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.

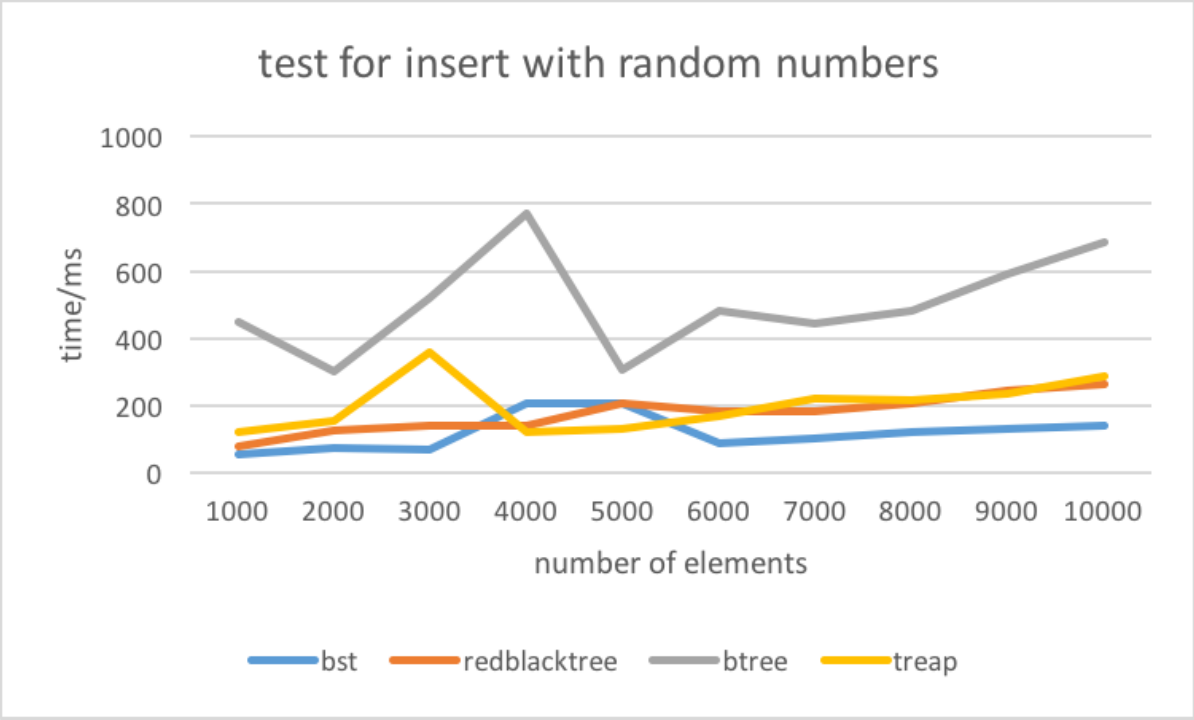
- insert: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.
- delete: $O(lgn)$ time for expected time, and $O(lgn)$ time for worst time.

Experimental Analysis

1. Insert Operation

I tested insert operation in the four kinds of trees with a dataset of random numbers and another of serial numbers. The number of elements in the test ranges from 1000 to 10000. For every different number of elements and different tree, the time is calculated as the sum of 200 runs.

- Test with random numbers

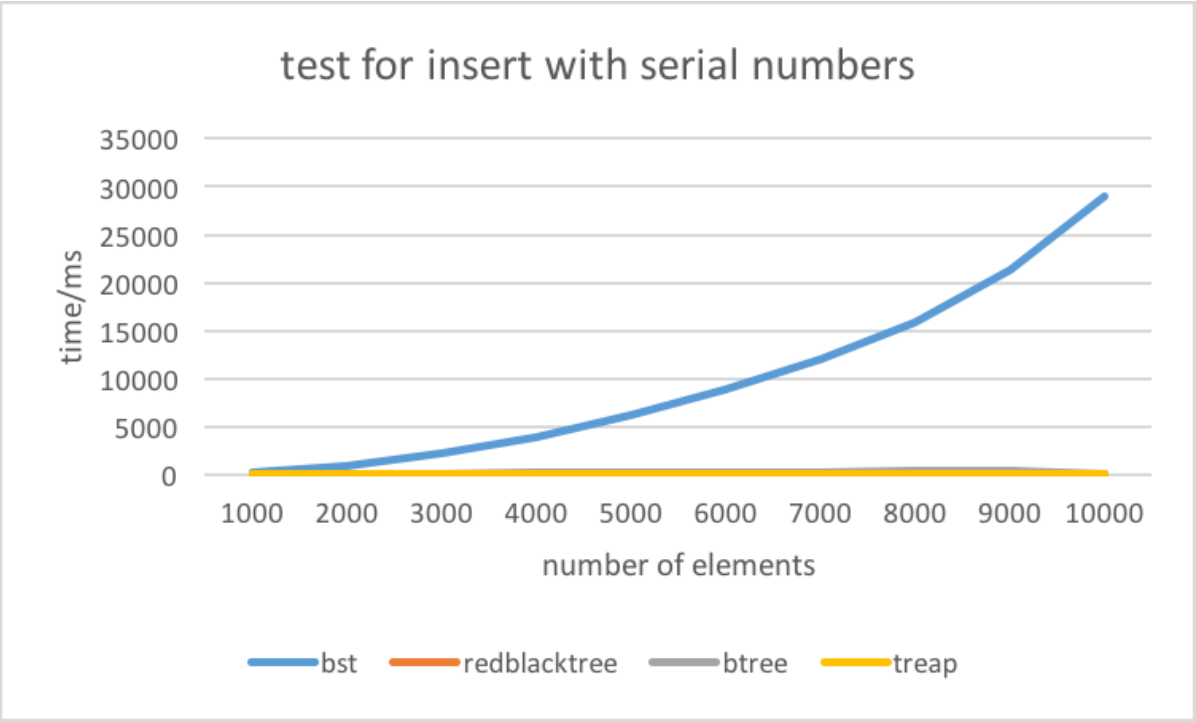


As we can see from the chart, on the random numbers, all the curves tend to increase in $O(lgn)$ speed, which is consistent with the theoretical value.

And among the four kinds of trees, the standard binary search tree can always have the best performance, due to its simpliest implementation and no extra operations, such as changing color and rotation, other than insertion.

What's more, the btree has the largest constant term and always takes more time than others, and it is subject to the factor t. And since there is no reading and writing operations on the disk, its advantage can't be revealed here.

- Test with serial numbers



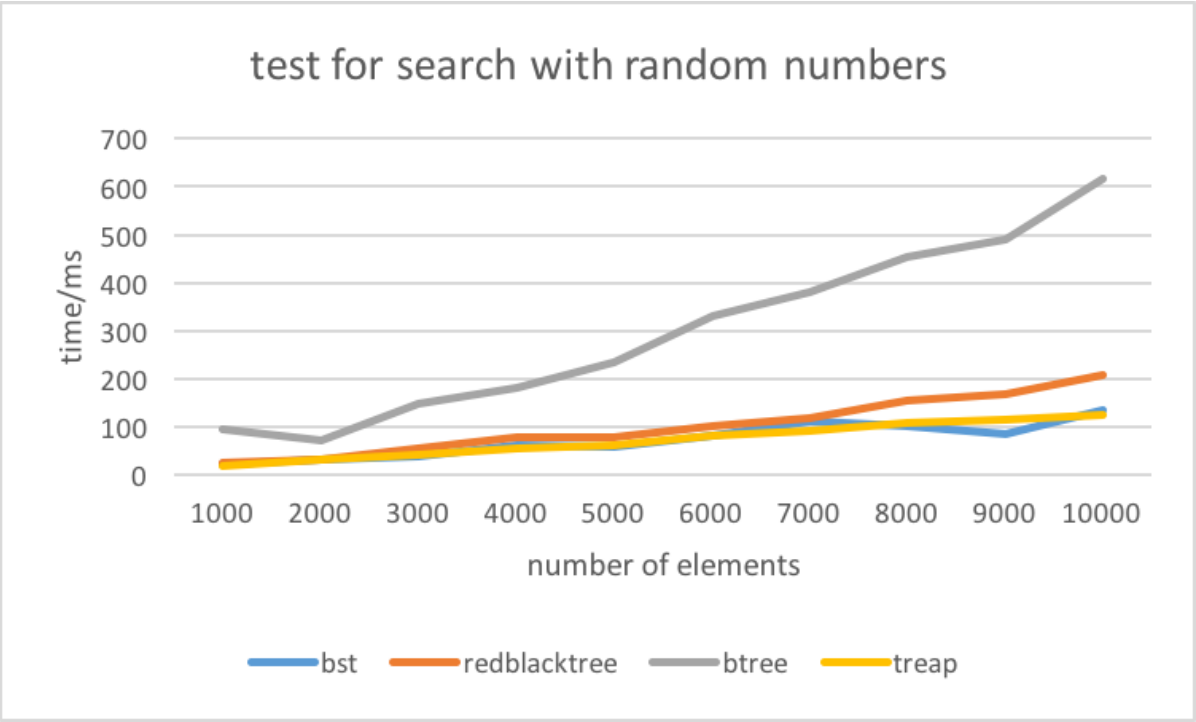
As we can see from this chart, the red black tree, b tree and treap, these three kinds of balanced bst, they perform just as good as on the random numbers, which means the time increases still in $O(lgn)$ with time goes on. While the standard bst is much worse on serial numbers. The time increases in $O(n^2)$.

So in this case, we can conclude that the balanced bst can always perform well for insert operation on both random numbers and serial numbers, while standard bst deteriorate pretty much on serial numbers.

2. Search Operation

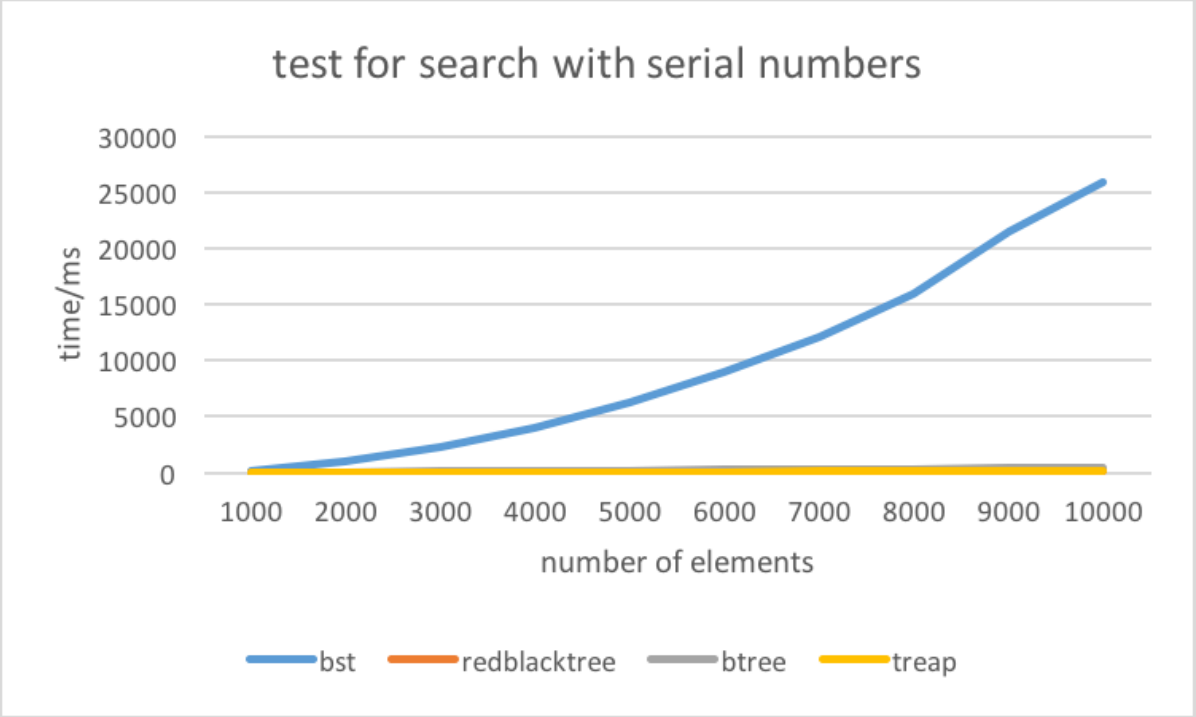
I tested search operation in the four kinds of trees with a dataset of random numbers and another of serial numbers. The number of elements in the test ranges from 1000 to 10000. For every different number of elements and different tree, the time is calculated as the sum of 200 runs.

- Test with random numbers



As seen from the chart, all the four trees' performance align with $O(lgn)$ and b tree has the largest constant term so it increases most fastly.

- Test with serial numbers

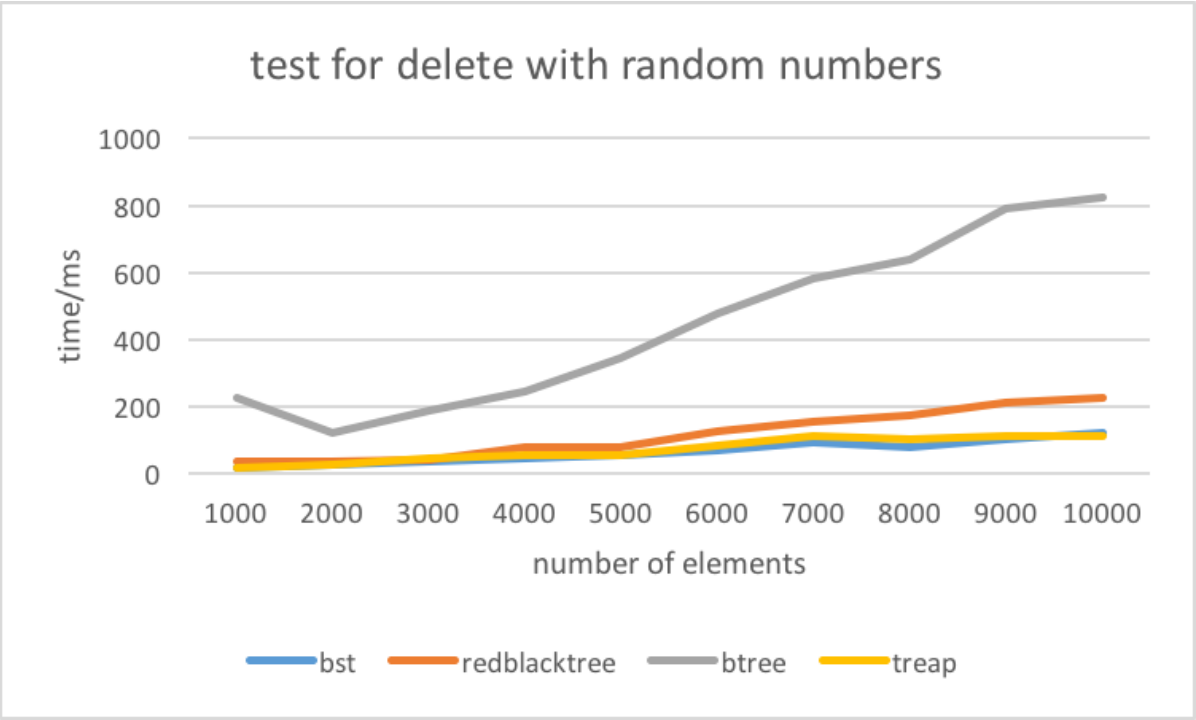


As seen from the chart, the three kinds of balanced bst work well on serial numbers, while standard bst can deteriorate to a linked list which performs in time complexity $O(n^2)$.

3. Delete Operation

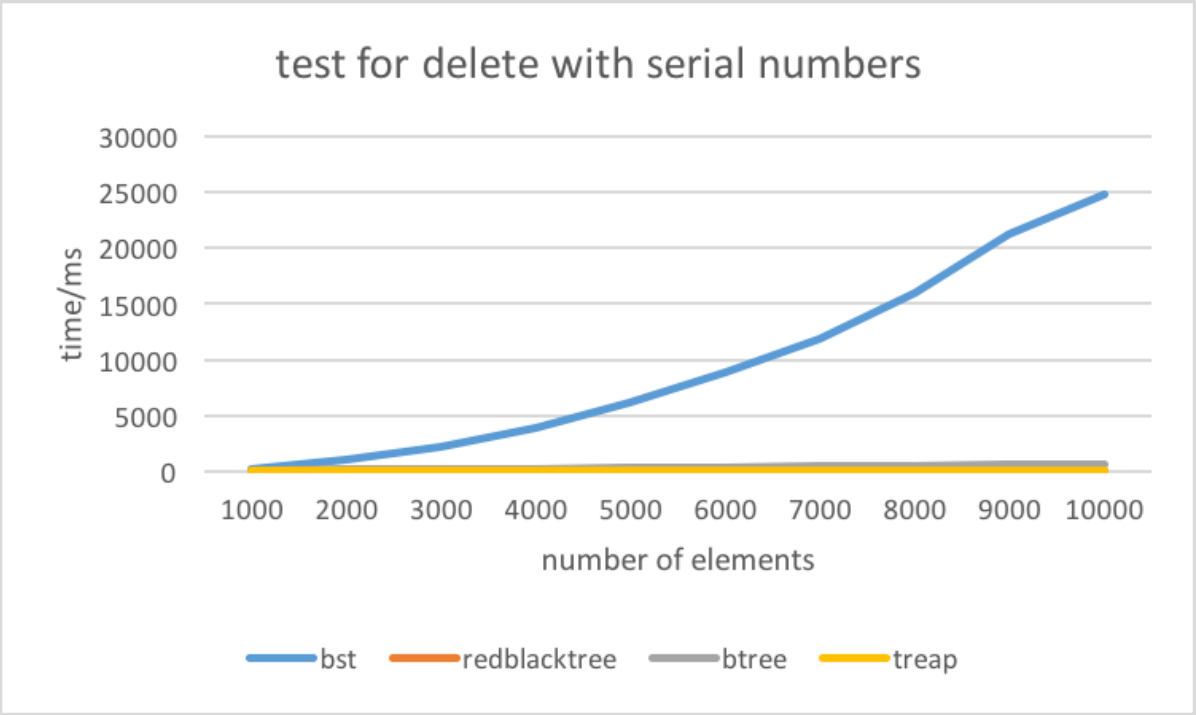
I tested delete operation in the four kinds of trees with a dataset of random numbers and another of serial numbers. The number of elements in the test ranges from 1000 to 10000. For every different number of elements and different tree, the time is calculated as the sum of 200 runs.

- Test with random numbers



As we can see from the chart, all the four trees' performance align with $O(lgn)$ and b tree has the largest constant term so it increases most fastly.

- Test with serial numbers



As seen from the chart, the three kinds of balanced bst work well on serial numbers, while standard bst can deteriorate to a linked list which performs in time complexity $O(n^2)$.