

# CS225 Assignment 6

Group 18: Xu Ke, Wang Dingkun, Qian shuaicun, Hua Bingshen

March 2021

## 1 Assignment 6 Exercise 1

### 1.1

Please see the code in the attached file.

### 1.2

Please see the code in the attached file.

## 2 Assignment 6 Exercise 2

### 2.1

Please see the code in the attached file.

### 2.2

In this algorithm, the part of searching delete value is same, so we just need to compare the merge and re-insert part. Obviously, in merge part the time complexity is  $O(h)$ , but in re-insert part, the time complexity will be  $O(nh)$  for this method will go through every elements under the deleted node. So the merge method is much better than re-insert.

## 3 Assignment 6 Exercise 3

### 3.1 the iterative implementation

#### 3.1.1 one possible implementation (normal but complex)

For the AVL tree, the iterative implementation of insert and delete operation can be achieved shown as the following code, which is conventional but very complex:

```
1  /* the iterative insertion function */
2  template<class T> void AVL<T>::insert(const T& item)
3  {
4      if(header->parent == NULL)
5      {
6          insertLeaf(item, header, header->parent);
7          header->left = header->parent;
8          header->right = header->parent;
9          return header->parent;
10     } // insert at the root
11     else
12     {
13         Link parent = header, child = header->parent, ancestor = NULL;
14         while(child != NULL)
```

```

15         {
16             parent = child;
17             if (child->balanceFactor != '=')
18                 ancestor = child;
19             if (compare(item, child->item))
20                 child = child->left;
21             else
22                 child = child->right;
23         }
24         if (compare(item, parent->item))
25         {
26             insertLeaf(item, parent, parent->left);
27             fixAfterInsertion(ancestor, parent->left);
28             if (header->left == parent)
29                 header->left = parent->left;
30             return parent->left;
31         } // insert at the left side of the parent
32         else
33         {
34             insertLeaf(item, parent, parent->right);
35             fixAfterInsertion(ancestor, parent->right);
36             if (header->right == parent)
37                 header->right = parent->right;
38             return parent->right;
39         } // insert at the right side of the parent
40     } // the tree is not empty
41 } // insert
42
43 template<class T> void AVL<T>::insertLeaf(const T& item, Link parent, Link& child)
44 {
45     child = new tree_node;
46     child->balanceFactor = '=';
47     child->isHeader = false;
48     child->item = item;
49     child->left = NULL;
50     child->right = NULL;
51     child->parent = parent;
52     node_count++;
53 } // insertLeaf
54
55 template<class T> void AVL<T>::fixAfterInsertion(Link ancestor, Link inserted)
56 {
57     Link root = header->parent;
58     T item = inserted->item;
59     if (ancestor == NULL)
60     {
61         if (compare(item, root->item))
62             root->balanceFactor = 'L';
63         else
64             root->balanceFactor = 'R';
65         adjustPath(root, inserted);
66     } // Case 1: all ancestor have balance factor '='
67     else if ((ancestor->balanceFactor == 'L' && !compare(item, ancestor->item)) ||
68             (ancestor->balanceFactor == 'R' && compare(item, ancestor->item)))
69     {
70         ancestor->balanceFactor = '=';

```

```

71         adjustPath(root, inserted);
72     } // Case 2: insert at the child tree opposite to the balance factor of the
       ancestor
73     else if (ancestor->balanceFactor == 'R' && !compare(item, ancestor->right->
       item))
74     {
75         ancestor->balanceFactor = '=';
76         rotateLeft(ancestor);
77         adjustPath(ancestor->parent, inserted);
78     } // Case 3: insert at the right child tree of the right child tree of the
       ancestor
79     else if (ancestor->balanceFactor == 'L' && compare(item, ancestor->left->item)
       )
80     {
81         ancestor->balanceFactor = '=';
82         rotateRight(ancestor);
83         adjustPath(ancestor->parent, inserted);
84     } // Case 4: insert at the left child tree of the left child tree of the
       ancestor
85     else if (ancestor->balanceFactor == 'L' && !compare(item, ancestor->left->
       item))
86     {
87         rotateLeft(ancestor->left);
88         rotateRight(ancestor);
89         adjustLeftRight(ancestor, inserted);
90     } // Case 5: insert at the right child tree of the left child tree of the
       ancestor
91     else
92     {
93         rotateRight(ancestor->right);
94         rotateLeft(ancestor);
95         adjustRightLeft(ancestor, inserted);
96     } // Case 6: insert at the left child tree of the right child tree of the
       ancestor
97 } // fixAfterInsertion
98
99 /* the iterative delete function */
100 template<class T> void AVL<T>::delete(T item)
101 {
102     if (item.nodePtr->parent->parent == item.nodePtr)
103     // item is located at the root node
104         deleteLink(itr.nodePtr->parent->parent);
105     else if (item.nodePtr->parent->left == item.nodePtr)
106     // item is located at left child
107         deleteLink(itr.nodePtr->parent->left);
108     else // item is located at right child
109         deleteLink(itr.nodePtr->parent->right);
110 } // delete
111
112 template<class T> void AVL<T>:: deleteLink(Link& link)
113 {
114     if (link->left == NULL || link->right == NULL) // the linked node has at
       most one child
115         prune(link);
116     else
117     {

```

```

118         deleteSuccessor(link);
119     }// the linked node has two child
120 }// deleteLink
121
122 template<class T> void AVL<T>:: deleteSuccessor(Link& link1)
123 {
124     T successor;
125     Link link = link1->right;
126     if(link->left == NULL)
127     {
128         successor = link->item;
129         link1->item = successor;
130         prune(link);
131     }//the left tree of link is empty
132     else
133     {
134         Link temp = link;
135         while(temp->left != NULL)
136             temp = temp->left;
137         successor = temp->item;
138         link1->item = successor;
139         prune(temp->parent->left);
140     } //the left tree of link is not empty, move downward to the most left side
        of link->right, assign the value to successor and delete it.
141 }//deleteSuccessor

```

### 3.1.2 another possible implementation (using stack to simplify)

For the AVL tree, the interactive implementation of insert and delete operation can be achieved using stack which greatly simplify the process and the sample code is shown as following:

```

1  template<class T> void AVL<T>:: Insert (AVLNode<Type>* &rt, const Type& x)//rt is
    the root node
2  {
3      AVLNode<Type>* pr = NLL;//father node
4      AVLNode<Type>* t = rt;//child node
5      stack<AVLNode<Type>*> st;
6      while(t != NULL)//find suitable place to insert
7      {
8          if(x == t->data)
9              return;
10         pr=t;
11         st. push(pr) ://record the path
12
13         if(x < t->data)
14             t =t->leftChild;
15         else
16             t = t->rightChild;
17     }
18     t = new AVLNode<Type>(x);
19     assert(t != NULL);
20
21     if(rt = NLL)//if root is empty, insert directly
22     {
23         rt=t;
24         return;

```

```

25 }
26 if(x < pr->data)//if not root
27     pr->leftChild = t;
28 else
29     pr->rightChild = t;
30
31 while(!st.empty())
32 { //trace back all the nodes in the stack, and judge whether they are balanced
    after the insertion
33
34     pr = st.top();
35     st.pop();
36
37     if(pr->leftChild == t)
38         //mark bf=right-left, thus insert at left, child tree becomes higher, bf
        decrease, vice versa.
39         pr->bf--;
40     else
41         pr->bf++;
42
43     //Will not influence the balance of the whole AVL Tree and bf
44     if(pr->bf == 0)
45         break;
46     else if(pr->bf == 1 || pr->bf == -1)
47         //this node is balanced cannot guarantee the balance of other nodes in the
        stack, thus track back the previous one
48
49         t = pr;
50
51     else //not balance
52     {
53         if(r->bf < 0)
54         {
55             if(t->bf < 0)
56             {
57                 RotateR(pr);
58             }
59             else
60             {
61                 RotateLR(pr);
62             }
63         }
64         else
65         {
66             if(t->bf < 0)
67             {
68                 RotateRL(pr);
69             }
70             else
71             {
72                 RotateL(pr);
73             }
74         }
75         break;
76     }
77 } //After rotation, need to set the child tree to the father node of the

```

```

    previous child tree
78  if(st.empty0)
79  //stack is empty means that trace to the root node
80      rt= pr;
81  else
82  {
83      AVLNode<Type> *s = st.top();
84      //reconnect, the element on the top on the stack is the father node of this
      tree, find the corresponding place and connect
85      if(pr->data < s->data)
86          s->leftChild = pr;
87      else
88          s->rightChild = pr;
89  }
90  }

```

### 3.2 the provided recursive implementation

For the AVL tree, the provided recursive implementation of insert and delete operation achieve the purpose shown as the following code:

```

1  /* the recursive insertion function */
2  template<class T> void AVL<T>::insert(T item)
3  {
4      root = _insert(root, item);
5      return;
6  }
7  template<class T> avlnode<T> *AVL<T>::_insert(avlnode<T> *pt, T val)
8  {
9      if (pt == 0) // if the tree is empty, we have to create a root node
10     {
11         avlnode<T> *newnode = new avlnode<T>;
12         (*newnode).setdata(val); // the stored value is the one given as argument
13         (*newnode).setbalance(0); // the balance must be 0
14         // note that left and right pointer are 0 by default
15         /* for the upward propagation of balance changes (and rotations, if
            necessary) we initialise the bad child and bad grandchild */
16         bchild = newnode;
17         bgrandchild = 0;
18         /* mode indicates, if balances need to be adjusted; a value false means
            that we are done */
19         mode = true;
20         return newnode;
21     }
22     if (val == (*pt).getdata())
23     {
24         /* the first case is the do-nothing case, when the given value already
            occurs in the AVL tree */
25         mode = false;
26         return pt;
27     }
28     if (val < (*pt).getdata()) // the case for insertion into the left successor
        tree
29     {
30         avlnode<T> *pt_new;

```

```

31  /* the recursive call returns a pointer to an updated left successor tree;
    it remains to adjust the balance */
32  pt_new = _insert((*pt).getleft(), val);
33  (*pt).setleft(pt_new);
34  /* the first case is the do-nothing case; the insertiion into the left
    successor did not alter the height */
35  if (mode == false)
36      return pt;
37  else
38  {
39      /* first compute the new balance, i.e. decrement it, as the insertion
        was done in the left successor tree */
40      int newbal = (*pt).getbalance() - 1;
41      /* if the new balance is -1, 0 or 1, only the balance needs to be
        updated */
42      if (newbal <= 1 && newbal >= -1)
43      {
44          (*pt).setbalance(newbal);
45          /* if the new balance is 0, no more changes further up the tree are
            necessary */
46          if (newbal == 0)
47              mode = false;
48          else
49          {
50              /* otherwise, the bad child and bad grandchild need to be moved
                one step up the tree */
51              bgrandchild = bchild;
52              bchild = pt;
53          }
54          return pt;
55      }
56      /* this leaves the case, when the old balance was already -1;
        the first case covers the bad grandchild being the left successor of
        the bad child; i.e., a single right rotation is required */
57      if ((*bchild).getleft() == bgrandchild)
58      {
59          avlnode<T> *newnode;
60          /* the new balance values 0, 0 are determined by the previous
            analysis; a single right rotation produces these values */
61          newnode = rotateright(pt, bchild, 0, 0);
62          /* as the root of the new subtree has balance 0, no more balance
            changes are needed */
63          mode = false;
64          return newnode;
65      }
66      /* the second case covers the bad grandchild being the right successor
        of the bad child; i.e., a left rotation with child and grandchild
        followed by a right rotation with parent and the former grandchild
        are required */
67      avlnode<T> *newnode1;
68      avlnode<T> *newnode2;
69      /* again, new balance values are determined by the previous analysis; a
        left-right rotation produces these values */
70      int c = 0, n = 0;
71      if (val < (*bgrandchild).getdata())
72          n = 1;
73

```

```

74         else if (val > (*bgrandchild).getdata())
75             c = -1;
76         newnode1 = rotateleft(bchild, bgrandchild, c, 0);
77         newnode2 = rotateright(pt, newnode1, n, 0);
78         /* again, as the root of the new subtree has balance 0, no more balance
79            changes are needed */
80         mode = false;
81         return newnode2;
82     }
83 else // if (val > (*pt).getdata())
84     // the dual case for insertion into the right successor tree
85     {
86         avlnode<T> *pt_new;
87         /* the recursive call returns a pointer to an updated right successor tree;
88            it remains to adjust the balance */
89         pt_new = _insert((*pt).getright(), val);
90         (*pt).setright(pt_new);
91         /* the first case is the do-nothing case; the insertiion into the right
92            successor did not alter the height */
93         if (mode == false)
94             return pt;
95         else
96         {
97             /* first compute the new balance, i.e. increment it, as the insertion
98                was done in the right successor tree */
99             int newbal = (*pt).getbalance() + 1;
100             if (newbal <= 1 && newbal >= -1)
101                 /* if the new balance is -1, 0 or 1, only the balance needs to be
102                    updated */
103                 {
104                     (*pt).setbalance(newbal);
105                     /* if the new balance is 0, no more changes further up the tree are
106                        necessary */
107                     if (newbal == 0)
108                         mode = false;
109                     else
110                     {
111                         /* otherwise, the bad child and bad grandchild need to be moved
112                            one step up the tree */
113                         bgrandchild = bchild;
114                         bchild = pt;
115                     }
116                     return pt;
117                 }
118             /* this leaves the case, when the old balance was already 1;
119                the first case covers the bad grandchild being the right successor of
120                the bad child; i.e., a single left rotation is required */
121             if ((*bchild).getright() == bgrandchild)
122             {
123                 avlnode<T> *newnode;
124                 /* the new balance values 0, 0 are determined by the previous
125                    analysis; a single left rotation produces these values */
126                 newnode = rotateleft(pt, bchild, 0, 0);
127                 /* as the root of the new subtree has balance 0, no more balance
128                    changes are needed */

```



```

120         mode = false;
121         return newnode;
122     }
123     /* the second case covers the bad grandchild being the left successor
        of the bad child; i.e., a right rotation with child and grandchild
        followed by a left rotation with parent and the former grandchild
        are required */
124     avlnode<T> *newnode1;
125     avlnode<T> *newnode2;
126     /* again, new balance values are determined by the previous analysis; a
        right-left rotation produces these values */
127     int c = 0, n = 0;
128     if (val < (*bgrandchild).getdata())
129         c = 1;
130     else if (val > (*bgrandchild).getdata())
131         n = -1;
132     newnode1 = rotateright(bchild, bgrandchild, c, 0);
133     newnode2 = rotateleft(pt, newnode1, n, 0);
134     /* again, as the root of the new subtree has balance 0, no more balance
        changes are needed */
135     mode = false;
136     return newnode2;
137 }
138 }
139 }
140
141 /* the recursive delete function */
142 template<class T> void AVL<T>::remove(T item)
143 {
144     root = _delete(root, item);
145     return;
146 }
147
148 template<class T> avlnode<T> *AVL<T>::_delete(avlnode<T> *pt, T val)
149 {
150     // nothing needs to be done for an empty tree
151     if (pt == 0)
152         return pt;
153     /* the first case occurs, when the sought value has been found */
154     if (val == (*pt).getdata())
155     {
156         /* in case there is no left successor tree, the result of the delete is the
            right successor tree */
157         if ((*pt).getleft() == 0)
158         {
159             mode = true;
160             return (*pt).getright();
161         }
162         /* in case there is no right successor tree, the result of the delete is
            the left successor tree */
163         if ((*pt).getright() == 0)
164         {
165             mode = true;
166             return (*pt).getleft();
167         }
168         /* if both left and right successor trees are not empty, we use the

```

```

    auxiliary function findswapleft to swap the value to be deleted with the
    maximum in the left successor tree; in addition, the node containing
    this maximum is deleted, and the whole left successor tree is
    reorganised to become again an AVL tree */
169 avlnode<T> *newnode;
170 newnode = findswapleft(pt, (*pt).getleft());
171 (*pt).setleft(newnode);
172 /* still the balance needs to be adjusted; this is done in the same as
    below in the recursive case covering a deletion in the left successor
    tree */
173 if (mode == false) // no change of height, no action required
174     return pt;
175 else
176 {
177     // compute the new balance (increment)
178     int newbal = (*pt).getbalance() + 1;
179     /* if the new balance is -1, 0 or 1 only the new balance needs to be
        stored */
180     if (newbal <= 1 && newbal >= -1)
181     {
182         (*pt).setbalance(newbal);
183         /* in addition, a new balance != 0 indicates that the height has
            not been altered, so no more changes are needed further up the
            tree */
184         if (newbal != 0)
185             mode = false;
186         return pt;
187     }
188     /* if the old balance was already 1, rotations become necessary; for
        these determine the bad child and bad grandchild in the right
        successor tree */
189     bchild = (*pt).getright();
190     /* if the bad child had balance 0, only a single left rotation is
        needed */
191     if ((*bchild).getbalance() == 0)
192     {
193         avlnode<T> *newnode;
194         /* the new balance values are those that must result from such a
            single left rotation */
195         newnode = rotateleft(pt, bchild, 1, -1);
196         /* as the root of the new subtree has balance !=0, the height has
            not been altered, so no changes are need further up the tree
            indicated by mode */
197         mode = false;
198         return newnode;
199     }
200     /* also, if the bad child had balance 1, only a single left rotation is
        needed */
201     if ((*bchild).getbalance() == 1)
202     {
203         avlnode<T> *newnode;
204         /* the new balance values are those that must result from such a
            single left rotation */
205         newnode = rotateleft(pt, bchild, 0, 0);
206         /* in this case the new subtree has balance 0, so the height has
            been altered; further changes are needed further up the tree */

```

```

207         return newnode;
208     }
209     /* the remaining case concerns a bad child with balance -1; in this
210        case a right rotation followed by a left rotation are needed */
211     bgrandchild = (*bchild).getleft();
212     avlnode<T> *newnode1;
213     avlnode<T> *newnode2;
214     /* the new balance values are those that must result from such a left-
215        right rotation */
216     int c = 0, n = 0;
217     if ((*bgrandchild).getbalance() == 1)
218         n = -1;
219     if ((*bgrandchild).getbalance() == -1)
220         c = 1;
221     newnode1 = rotateright(bchild, bgrandchild, c, 0);
222     newnode2 = rotateleft(pt, newnode1, n, 0);
223     /* in this case also the new subtree has balance 0, so the height has
224        been altered; further changes are needed further up the tree */
225     return newnode2;
226 }
227 }
228 /* as long as the sought value has not been found a recursive descent is
229    required; the first case continues the search in the left successor tree */
230 if (val < (*pt).getdata())
231 {
232     avlnode<T> *pt_new;
233     /* the recursive call returns an updated left successor tree, in which the
234        sought value (if in there) has been deleted */
235     pt_new = _delete((*pt).getleft(), val);
236     (*pt).setleft(pt_new);
237     /* mode indicates, if balance adjustments are still needed; the first case
238        is the no-action case */
239     if (mode == false)
240         return pt;
241     else
242     {
243         /* otherwise, first compute the new balance (increment) */
244         int newbal = (*pt).getbalance() + 1;
245         /* if the new balance is -1, 0 or 1 only the new balance needs to be
246            stored */
247         if (newbal <= 1 && newbal >= -1)
248         {
249             (*pt).setbalance(newbal);
250             /* in addition, a new balance != 0 indicates that the height has
251                not been altered, so no more changes are needed further up the
252                tree */
253             if (newbal != 0)
254                 mode = false;
255             return pt;
256         }
257         /* if the old balance was already 1, rotations become necessary; for
258            these determine the bad child and bad grandchild in the right
259            successor tree */
260         bchild = (*pt).getright();
261         /* if the bad child had balance 0, only a single left rotation is
262            needed */

```

```

251     if ((*bchild).getbalance() == 0)
252     {
253         avlnode<T> *newnode;
254         /* the new balance values are those that must result from such a
           single left rotation */
255         newnode = rotateleft(pt, bchild, 1, -1);
256         /* as the root of the new subtree has balance !=0, the height has
           not been altered, so no changes are needed further up the tree
           indicated by mode */
257         mode = false;
258         return newnode;
259     }
260     /* also, if the bad child had balance 1, only a single left rotation is
       needed */
261     if ((*bchild).getbalance() == 1)
262     {
263         avlnode<T> *newnode;
264         /* the new balance values are those that must result from such a
           single left rotation */
265         newnode = rotateleft(pt, bchild, 0, 0);
266         /* in this case the new subtree has balance 0, so the height has
           been altered; further changes are needed further up the tree */
267         return newnode;
268     }
269     /* the remaining case concerns a bad child with balance -1; in this
       case a right rotation followed by a left rotation are needed */
270     bgrandchild = (*bchild).getleft();
271     avlnode<T> *newnode1;
272     avlnode<T> *newnode2;
273     /* the new balance values are those that must result from such a left-
       right rotation */
274     int c = 0, n = 0;
275     if ((*bgrandchild).getbalance() == 1)
276         n = -1;
277     if ((*bgrandchild).getbalance() == -1)
278         c = 1;
279     newnode1 = rotateright(bchild, bgrandchild, c, 0);
280     newnode2 = rotateleft(pt, newnode1, n, 0);
281     /* in this case also the new subtree has balance 0, so the height has
       been altered; further changes are needed further up the tree */
282     return newnode2;
283 }
284 }
285 /* the second case continues the search in the right successor tree */
286 else // if (val > (*pt).getdata())
287 {
288     avlnode<T> *pt_new;
289     /* the recursive call returns an updated right successor tree, in which the
       sought value (if in there) has been deleted */
290     pt_new = _delete((*pt).getright(), val);
291     (*pt).setright(pt_new);
292     /* mode indicates, if balance adjustments are still needed; the first case
       is the no-action case */
293     if (mode == false)
294         return pt;
295     else

```

```

296 /* otherwise, first compute the new balance (decrement) */
297 {
298     int newbal = (*pt).getbalance() - 1;
299     /* if the new balance is -1, 0 or 1 only the new balance needs to be
        stored */
300     if (newbal <= 1 && newbal >= -1)
301     {
302         (*pt).setbalance(newbal);
303         /* in addition, a new balance != 0 indicates that the height has
            not been altered, so no more changes are needed further up the
            tree */
304         if (newbal != 0)
305             mode = false;
306         return pt;
307     }
308     /* if the old balance was already -1, rotations become necessary; for
        these determine the bad child and bad grandchild in the left
        successor tree */
309     bchild = (*pt).getleft();
310     /* if the bad child had balance 0, only a single right rotation is
        needed */
311     if ((*bchild).getbalance() == 0)
312     {
313         avlnode<T> *newnode;
314         /* the new balance values are those that must result from such a
            single right rotation */
315         newnode = rotateright(pt, bchild, -1, 1);
316         /* as the root of the new subtree has balance !=0, the height has
            not been altered, so no changes are needed further up the tree
            indicated by mode */
317         mode = false;
318         return newnode;
319     }
320     /* also, if the bad child had balance -1, only a single right rotation
        is needed */
321     if ((*bchild).getbalance() == -1)
322     {
323         avlnode<T> *newnode;
324         /* the new balance values are those that must result from such a
            single right rotation */
325         newnode = rotateright(pt, bchild, 0, 0);
326         /* in this case the new subtree has balance 0, so the height has
            been altered; further changes are needed further up the tree */
327         return newnode;
328     }
329     /* the remaining case concerns a bad child with balance 1; in this case
        a left rotation followed by a right rotation are needed */
330     bgrandchild = (*bchild).getright();
331     avlnode<T> *newnode1;
332     avlnode<T> *newnode2;
333     /* the new balance values are those that must result from such a right-
        left rotation */
334     int c = 0, n = 0;
335     if ((*bgrandchild).getbalance() == 1)
336         c = -1;
337     if ((*bgrandchild).getbalance() == -1)

```

```

338         n = 1;
339         newnode1 = rotateleft(bchild, bgrandchild, c, 0);
340         newnode2 = rotateright(pt, newnode1, n, 0);
341         /* in this case also the new subtree has balance 0, so the height has
           been altered; further changes are needed further up the tree */
342         return newnode2;
343     }
344 }
345 }

```

### 3.3 difference analysis

For the iterative implementation of AVL tree insert and delete operation, we claim that there are mainly two ways to achieve the purpose. The first way is conventional but very complex. The way to do the insert and delete operation with AVL balanced tree nodes is to first use the method of BST(binary search tree) to insert and delete, and then maintain the inserted or deleted tree to meet the requirement of AVL balanced tree, that is the balance must be -1 or 0 or 1. If using the iterative idea to implement, there are a lot of things to consider while deleting, and the code thus is messy and disgusting. The second way is using stack to trace and record the path while traversing thus simplifying the process of adjusting balance. But the disadvantage of this is very clear, that is it will use much memory location to perform stack functionality.

For the recursive implementation of AVL tree insert and delete operation, it seems to be very easy to implement compared with the iterative implementation, and the code is clearer and readable. However, due to the nested calling while doing the recursion, it maybe time-consuming. And by the way, it also requires much memory location.

## 4 Assignment 6 Exercise 4

### 4.1

for an AVL tree satisfies the median property and it has odd number of nodes, then  $l(v)$  will be the median of the set. And the number of nodes in the left subtree must be equal to the right subtree. And we notice that the two subtrees rooted at  $v$  also satisfies the median property, so the symmetric structure is about the root  $v$ . Then the AVL property of root node  $v$  is satisfied. Induction shows that all AVL trees which satisfies the median property with odd number of nodes are perfectly balanced. For an AVL tree satisfies the median property and it has odd number of nodes, then the  $l(v)$  must be the high median of the set.

### 4.2

For insert( $x$ ) operation, we developed a recursive algorithm.

1) If the root of the AVL tree is NULL, we just create a new node which contains the value  $x$  and set the root pointer to this node.

2) If the root is not NULL, then we perform an AVL insert operation. A new node containing  $x$  is created and is set as child of that leaf node according to the comparison result between  $x$  and the value of that leaf node.

3) Then we need to adjust the tree to satisfy both the AVL property and the median property of the new tree.

– If the tree rooted at  $v$  has odd number of nodes before insertion:

a) If the value of the new node is greater than the root value, we need to find the smallest element that is greater than  $v.value$ . Take the node out of the tree and place it to the root  $v$ . Then perform insert( $v.value$ ) to the subtree rooted at  $root.left$ .

b) Else, nothing to do with the current tree rooted at  $v$ .

– If the tree rooted at  $v$  has even number of nodes before insertion:

a) If the value of the new node is greater than the root value, nothing to do with the current tree rooted at  $v$ .

b) Else, we need to find the largest element that is smaller than  $v.value$ . Take the node out of the tree and place it to the root  $v$ . Then perform insert( $v.value$ ) to the subtree rooted at  $root.right$ .

### 4.3

Perform the AVL remove operation, the rest operations are the same as the insert operation. And the entire implementation is shown in the code.

### 4.4

We consider the worst cases, every node requires  $2\log 2$  height (height is the height of the subtree rooted on the node). Assume  $n$  is the 2 to the power of  $m$ ,  $h(2n) = h(n) + 2n$ . so  $h(2 \text{ to the power of } m + 1) = h(2 \text{ to the power of } m) + 2 \text{ to the power of } m + 1$ . By using the characteristic polynomial equation we can get  $h(n) \in O(n)$  ( $n = 2 \text{ to the power of } m$ ). So  $h(n) \in O(n)$ .