Lab 1 - AVL and Splay Tree

Group 13, Shuting Guo, Ruijia Zhang, Ziyue Tang

Feb. 26^{th}, 2019

Contents

Chapter 1

Introduction

1.1 Introduction of Main Task

This lab project requires a Splay and an AVL tree implement programmed by C/C++. And then, test the performance on several cases of data inputs and analyze the complexity of time and space, and compare with basic non-balanced BST.

1.2 Algorithm specification

AVL and Splay are optimized BST. The main idea of optimizing a BST data structure is, control the height of the tree. So that the height of a balanced BST can be reduced to O(logN). In this case, all operation on a BST can be done in O(logN) time.

In AVL and Splay, the main method keeping balance is based on *Rotate* operation. For AVL tree structure, *Rotate* is performed when the tree is partly unbalanced, therefore, AVL is strictly balanced binary tree. Because each insert operation add O(1) height on a subtree, this needs only O(1) rotation to keep the balance. So, AVL can run fast compared with other BST structures. However, AVL is strictly balanced, and for the same dataset, the shape of AVL is unique.

However, Splay rotates the node with every operation. The reason why Splay can do this while keeping balance is, in Splay's rotate operation, the order of rotation is decided on double nodes. This double-rotation guarantees the height will not be worse during the large amount of rotation. So, when the amount of operation become huge, each operation is implied with rotate operation, means that the shape of Splay is getting more and more balanced. Splay can be rotated arbitrarily, so splay is more flexible. But it needs more time to rotate, so splay can be a little bit slower than other balanced trees.

Both structures can deal data in O(logN) time and O(N) space.

Chapter 2

Program Design

2.1 Class Design

For each structure (AVL, Splay and normal BST), we design a class who has several public members shown as follows.

```
1: class BST
2: {
3:   public:
4:     void Insert(const _tp & x);
5:     bool Find(const _tp & x);
6:     void Delete(const _tp & x);
7:     int Size();
8:     bool Empty();
9:     _tp Find_less(const _tp & x);
10:    _tp Find_greater(const _tp & x);
11:    std::vector<_tp> To_vector();
12: };
```

1. *Insert* is used to insert an element
2. *Find* returns true if an element is in the tree
3. *Delete* is used to delete an element
4. *Size* returns the number of current elements in the tree
5. *Empty* returns true if *Size* is zero
6. *To_vector* returns a `std::vector<_tp>`, contains all element in increasing order
7. *Find_less* returns the first number strictly less than x
8. *Find_greater* returns the first number strictly greater than x

note: If no element is less (greater) than x, returns the Min (Max) element in the tree. If the tree is empty, return an initialized element _tp()

The BST class is designed as a C++ template class which can store any data type whose compare operators are defined already.

For memory allocation, we use a simple, size-fixed memory pool to avoid from allocating small blocks of data frequently from OS directly (which is slow). We first allocate a block of memory size MAXN, and then allocate one-by-one manually. For recycled memory, we use a stack to store these addresses, and use them first.

2.2 AVL Implement

2.2.1 Insert

Insert a node recursively, until a NULL node is met, allocate a new node and replace it. While tracing back, update the height of each node and check if a subtree is still balanced. If not, fix it by rotation.

2.2.2 Delete

First, find the node to delete. If it's a leaf node, delete it directly. Else, replace it by its precursor or subsequence in left or right subtree, and delete its precursor or subsequence recursively. Update the node to keep balance by the same way as insert.

2.2.3 Update node

Let $height = max\{LeftHight, RightHeight\} + 1$.
If $|LeftHeight - RightHeight| > 1$, check the direction of edges that link nodes, and always rotate the higher subtree to lower side.

2.3 Splay Implement 1 (Rotate from leaf up to root)

2.3.1 Insert

1. For normal cases, splay precursor to root, splay subsequence to root->r, then the node should be linked to root->r->l.

2. If the new node X is min(max) element in the tree, splay min element to root and link root to the right(left) side of X, then replace the root with X. That is, after insertion, the old root should be right(left) son or new root X.

3. If the tree is empty, use X as root.

2.3.2 Delete

1. For normal cases, splay precursor to root, splay subsequence to root->r, then the node should be deleted is root->r->l.

2. If the element is min(max) element in the tree, splay it to root, use root->r(root->l) as root and delete the old root.

3. If the element the only element int the tree, delete it directly, and the tree becomes empty.

2.3.3 Rotate

Rotate a node to shallower position.

2.3.4 Splay

Use Zig-Zig, Zig-Zag, Zag-Zig, Zag-Zag, rotate a node up, until its father node is target. (If target is NULL, the node will be rotated to root.)

2.3.5 Find

After node is found or not, splay the last-accessed node to root. (This should be well noticed, because if you forget to splay especially when the node is not found, the time complexity can increase to O(N).)

2.3.6 Note

Because while inserting nodes into a splay, the height of the tree can be very high in some special cases. In splay implement, we use **non-recursively** (loop instead) way to do all operation, in case of stack overflow. And *To_vector* is written by using *Find_greater* N times, so *To_vector* is O(NlogN) rather than O(N).

2.4 Splay Implement 2 (Split from root down to leaves)

2.4.1 Splay

In this version, we build two extra trees *Ltree* and *Rtree*. Suppose we have a pivot x. On a node t, if t->*val*<x, then link it to Ltree, else link it to Rtree. In the end, all elements less than x will in Ltree, and all elements greater than x will in Rtree.

2.4.2 Insert

If we want to insert an element x, first `Splay(x)`, then link Ltree to its left, link Rtree to its right.

2.4.3 Delete

First of all, `Splay(x)` to root. Then check

1. If T is Min(Max) node in the tree, let Root->Left (Root->Right) be the root.

2. Else split left subtree by x, then Root->Left->Right must be empty, for x is greater any element in Root->Left. Now, link Root->Right to Root->Left->Right. Finally delete x.

Note

For more information and details, see ppt.

Chapter 3

Testing

3.1 Correctness

The correctness is checked by comparing with `std::set<int>`. Data is generated by random generator. To ensure the correctness of algorithm, we write 4 codes, each use STL `std::set<int>`, AVL, Splay and normal unbalanced BST.

Input N and N commands. Commands are input as following way.

`i x` , insert x, output the number of elements in the tree
`d x` , delete x, output the number of elements in the tree
`f x` , find x, output True if x is found, otherwise output False
`l x` , find the element less than x, and output it
`g x` , find the element greater than x, and output it.

After all operations, output all elements in the tree in order.

For each random data, for codes run together, then check if four output are same.

Conclusion For $N = \{3, 5, 10, 50, 100, 1000, 1e4, 1e5, 1e6\}$ All codes works well, and output the same answer on random data (frequency of each operation is same).

Note that while testing, all input numbers are non-negative.

3.2 Performance

We have tested these codes in five situations, and measure the time cost on each, result is shown as follows.

3.2.1 Situation 1

The same as correctness testing.

Algorithm	1000	$1e4$	$1e5$	$1e6$
std::set	31ms	33ms	99ms	1141ms
Splay	31ms	47ms	144ms	1419ms
AVL	31ms	47ms	106ms	1132ms
Normal BST	31ms	47ms	126ms	1059ms

Table 3.2.1.1: Situation 1

In this case, input is random, so that in normal BST, the shape of tree is tend to be balanced. Meanwhile, splay is slower because its frequent rotation.

3.2.2 Situation 2

Insert 1-N in order and delete in the same order.

Algorithm	1000	$1e4$	$1e5$	$1e6$
std::set	28ms	38ms	178ms	1647ms
Splay	35ms	45ms	169ms	1612ms
AVL	28ms	46ms	209ms	2138ms
Normal BST	47ms	141ms	9503ms	-

Table 3.2.2.1: Situation 2

In this case, AVL needs very frequent rotation to keep balance. And normal BST is grown up to $O(N^2)$ and the time cost become unacceptable.

3.2.3 Situation 3

Insert 1-N in order and delete in reversed order.

Algorithm	1000	$1e4$	$1e5$	$1e6$
std::set	28ms	40ms	177ms	1952ms
Splay	39ms	47ms	179ms	1776ms
AVL	35ms	38ms	196ms	2269ms
Normal BST	35ms	231ms	9503ms	-

Table 3.2.3.1: Situation 3

The same as Situation 2. But for Splay, in this case, it doesn't needs to do any rotation, just link nodes one by one to the root, and delete one by one from root.

3.2.4 Situation 4

Number in range [0,N), insert and delete in random order.

Algorithm	1000	1e4	1e5	1e6
std::set	32ms	34ms	258ms	3260ms
Splay	44ms	47ms	319ms	5089ms
AVL	31ms	46ms	253ms	3352ms
Normal BST	31ms	47ms	211ms	2556ms

Table 3.2.4.1: Situation 4

The same as Situation 1.

3.2.5 Situation 5

Insert 1-N in order, then find 0 for N times, find less N+1 for N times, find greater 0 for N times.

Algorithm	1000	1e4	1e5	1e6
std::set	32ms	48ms	291ms	2971ms
Splay	46ms	47ms	271ms	2656ms
AVL	32ms	79ms	357ms	3394ms
Normal BST	31ms	163ms	9611ms	-

Table 3.2.5.1: Situation 5

This case is to test whether complexity of query operations are correct.

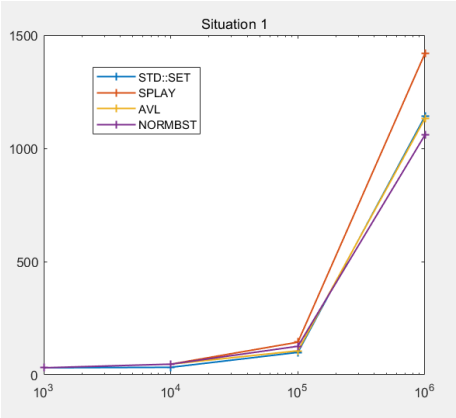

Figure 3.2.5.1: Situation 1

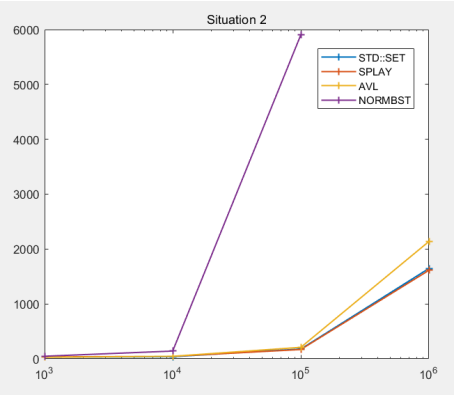

Figure 3.2.5.2: Situation 2

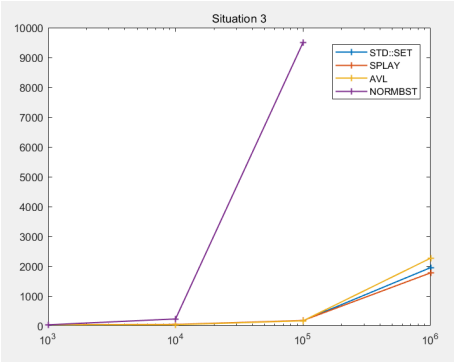

Figure 3.2.5.3: Situation 3

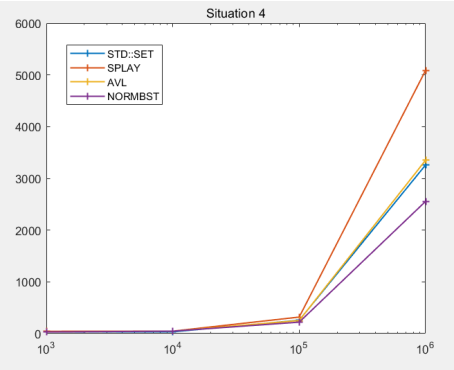

Figure 3.2.5.4: Situation 4

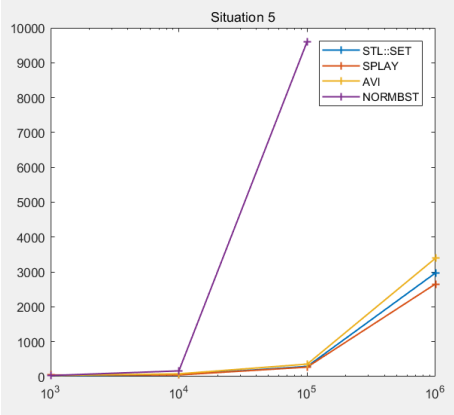

Figure 3.2.5.5: Situation 5

Chapter 4

Conclusion

Though these data, we can see, STL set written by RB-tree is very stable and fast. Splay is quick when the input is unbalanced, when the input is balanced, plenty of rotation slow Splay down. As for AVL, is highly balanced, so that in random data each operation each time cost strict O(logN) time, this is very fast. However, when input is unbalanced, inserting can cause more rotation.

Anyway, these three balanced BST structures guarantee the complexity of insert, delete and find operation are O(logN), this would not be worse. But in normal BST, the shape of tree decides on input, this may cause the degeneration of time complexity, in worst cases, the time complexity is $O(N^2)$.

Chapter 5

Appendix: Code

5.1 AVL.cpp

```
1: #include <iostream>
2: #include <algorithm>
3: #include <string>
4: #include <cmath>
5: #include <cstdio>
6: #include <cstdlib>
7: #include <cstring>
8: #include <ctime>
9: #include <vector>
10:
11: using namespace std;
12:
13: #define Debug
14: #define DEFAULT_MAXN 1000
15:
16: template<typename _tp,int MAXN=DEFAULT_MAXN>
17: class AVL
18: {
19:   private:
20:     class AVL_node
21:     {
22:       friend class AVL;
23:       private:
24:         _tp val;
25:         AVL_node *l,*r;
26:         int h; // Height of tree
27:       public:
28:         AVL_node():val() { l=r=NULL; h=0; }
29:         AVL_node(const _tp & x) { val=x; l=r=NULL; h=0; }
30:     };
31:
```

```
32:     typedef AVL_node node;
33:
34:     node *U;
35:     node **Trash,*ALL;
36:     node *Root;
37:     int Top,size;
38:
39:     // Create a new node with value x
40:     node* ALLOC(const _tp & x)
41:     { size++; return new(Top?Trash[Top--]:ALL++)node(x); }
42:
43:     // Recycle a node in the tree
44:     void RECYCLE(node *& t)
45:     { size--; if(!t) return ; Trash[++Top]=t; return ; }
46:
47:     int AVL_abs(const int x) { return x>=0?x:-x; }
48:
49:     // In case of accessing NULL
50:     int geth(node * t) { return t?t->h:0; }
51:
52:     // Left-Rotate
53:     void RotateL(node *& t)
54:     {
55:       node *temp=t->r;
56:       t->r=temp->l;
57:       temp->l=t; t=temp;
58:       t->l->h=max(geth(t->l->l),geth(t->l->r))+1;
59:       t->h=max(geth(t->l),geth(t->r))+1;
60:       return ;
61:     }
62:
63:     // Right-Rotate
64:     void RotateR(node *& t)
65:     {
66:       node *temp=t->l;
67:       t->l=temp->r;
68:       temp->r=t; t=temp;
69:       t->r->h=max(geth(t->r->l),geth(t->r->r))+1;
70:       t->h=max(geth(t->l),geth(t->r))+1;
71:       return ;
72:     }
73:
74:     // Fix Height after insert or delete
75:     void update_node(node *& t)
76:     {
77:       // Update height
78:       t->h=max(geth(t->l),geth(t->r))+1;
79:       // Unbalanced
80:       if(AVL_abs(geth(t->l) - geth(t->r)) >  1)
81:       {
```

```
82:         if(geth(t->l) > geth(t->r))
83:           if(geth(t->l->l) > geth(t->l->r)) RotateR(t);
84:           else RotateL(t->l), RotateR(t);
85:         else
86:           if(geth(t->r->l) < geth(t->r->r)) RotateL(t);
87:           else RotateR(t->r), RotateL(t);
88:       }
89:     }
90:
91:     void Insert(node *& t,const _tp & x)
92:     {
93:       if(!t) { t=ALLOC(x); t->h=1; return ; }
94:       if(x==t->val) return ;
95:       if(x<t->val) Insert(t->l,x);
96:       else Insert(t->r,x);
97:       update_node(t);
98:       return ;
99:     }
100:
101:    node * Find(node * t,const _tp & x)
102:    {
103:      while(t && x!=t->val)
104:        if(x < t->val) t=t->l;
105:        else if(x > t->val) t=t->r;
106:      return t;
107:    }
108:
109:    node * Find_Max(node * t)
110:    { while(t && t->r) t=t->r; return t; }
111:    node * Find_Min(node * t)
112:    { while(t && t->l) t=t->l; return t; }
113:
114:
115:    void Delete(node *& t,const _tp & x)
116:    {
117:      if(!t) { return ; }
118:      // Found x, delete it
119:      if(x==t->val)
120:        if(!t->l && !t->r) RECYCLE(t),t=NULL;
121:        else if(t->l && !t->r) RECYCLE(t),t=t->l;
122:        else if(t->r && !t->l) RECYCLE(t),t=t->r;
123:        // replace t with min node 'P' in left subtree
124:        //   or max node 'P' in right subtree
125:        // Delete 'P' recursively
126:        else
127:          if(geth(t->l) > geth(t->r))
128:            t->val=Find_Max(t->l)->val,
129:            Delete(t->l,t->val);
130:          else
131:            t->val=Find_Min(t->r)->val,
```

```
132:             Delete(t->r,t->val);
133:       else if(x<t->val) Delete(t->l,x);
134:       else              Delete(t->r,x);
135:
136:       // keep balance
137:       if(t) update_node(t);
138:       return ;
139:     }
140:
141:   protected:
142:     node * Find_greater(node * t, _tp x)
143:     {
144:       node * temp=0;
145:       while(t)
146:         if(x<t->val) temp=t,t=t->l;
147:         else t=t->r;
148:       return temp;
149:     }
150:
151:     node * Find_less(node * t, _tp x)
152:     {
153:       node * temp=0;
154:       while(t)
155:         if(x<=t->val) t=t->l;
156:         else temp=t,t=t->r;
157:       return temp;
158:     }
159:
160:     void To_vector(node * t, std::vector<_tp> & vec)
161:     {
162:       if(!t) return ;
163:       To_vector(t->l,vec);
164:       vec.push_back(t->val);
165:       To_vector(t->r,vec);
166:     }
167:
168:   public:
169: #ifdef Debug
170:     void Show_tree() { P(Root); }
171:     void P(node *t)
172:     { cout << "***************" << endl; _P(t);
173:       cout << "***************" << endl; cout << endl; }
174:     void _P(node * t)
175:     {
176:       if(!t) return ;
177:       if(t->l)
178:       {cout<<t->val<<" —L— >"<<t->l->val<<endl;_P(t->l);}
179:       if(t->r)
180:       {cout<<t->val<<" —R— >"<<t->r->val<<endl;_P(t->r);}
181:     }
```

```
#endif
    AVL()
    {
      U=new node[MAXN];
      Trash=new node*[MAXN];
      memset(U,0,sizeof(_tp)*MAXN);
      ALL=U; Top=0; Root=0; size=0;
    }
    ~AVL()
    {
      delete[] U;
      delete[] Trash;
    }

    void Insert(const _tp & x) { Insert(Root,x); }
    bool Find(const _tp & x) { return Find(Root,x); }
    void Delete(const _tp & x) { Delete(Root,x); }
    int Size() { return size; }
    bool Empty() { return size==0; }

    _tp Find_less(const _tp & x)
    {
      node * temp=Find_less(Root,x);
      if(size) return temp?temp->val:Find_Min(Root)->val;
      return _tp();
    }
    _tp Find_greater(const _tp & x)
    {
      node * temp=Find_greater(Root,x);
      if(size) return temp?temp->val:Find_Max(Root)->val;
      return _tp();
    }
    std::vector<_tp> To_vector()
    {
      std::vector<_tp> vec;
      To_vector(Root,vec);
      return vec;
    }
};

int main()
{
  AVL<int,1000000> T;

  int n,x;
  char op[10];
  scanf("%d",&n);
  while(n--)
  {
    scanf("%s",op);
```

```
232:     scanf("%d",&x);
233:     if(op[0]=='i')
234:     {
235:       T.Insert(x);
236:       printf("%d\n",T.Size());
237:     }
238:     else if(op[0]=='d')
239:     {
240:       T.Delete(x);
241:       printf("%d\n",T.Size());
242:     }
243:     else if(op[0]=='f')
244:       printf(T.Find(x)?"True\n":"False\n");
245:     else if(op[0]=='l')
246:       printf("%d\n",T.Find_less(x));
247:     else if(op[0]=='g')
248:       printf("%d\n",T.Find_greater(x));
249:     //T.Show_tree();
250:   }
251:
252:   vector<int> vec=T.To_vector();
253:   for(auto it=vec.begin();it!=vec.end();++it)
254:     printf("%d ",*it);
255:   printf("\n");
256:
257:   return 0;
258: }
```

5.2 Splay 1

```
1: #include <iostream>
2: #include <algorithm>
3: #include <string>
4: #include <cmath>
5: #include <cstdio>
6: #include <cstdlib>
7: #include <cstring>
8: #include <ctime>
9: #include <vector>
10:
11: using namespace std;
12:
13: #define Debug
14: #define DEFAULT_MAXN 1000
15:
16: template<typename _tp,int MAXN=DEFAULT_MAXN>
17: class Splay
18: {
```

```
19:   private:
20:     class Splay_node
21:     {
22:       friend class Splay;
23:       private:
24:         _tp val;
25:         Splay_node *fa;
26:         Splay_node *s[2];
27:         inline bool getlr() { return fa->s[1]==this; }
28:         Splay_node * link(const int w, Splay_node * t)
29:         { if((s[w]=t)) t->fa=this; return this; }
30:       public:
31:         Splay_node():val() { s[0]=s[1]=fa=NULL; }
32:         Splay_node(const _tp & x)
33:         { val=x; s[0]=s[1]=fa=NULL; }
34:     };
35:
36:     typedef Splay_node node;
37:
38:     node *U,*MinNode,*MaxNode;
39:     node **Trash,*ALL;
40:     node *Root;
41:     int Top,size;
42:
43:     node* ALLOC(const _tp & x)
44:     { size++; return new(Top?Trash[Top--]:ALL++)node(x); }
45:
46:     void RECYCLE(node *& t)
47:     { size--; if(!t) return ; Trash[++Top]=t; return ; }
48:
49:     void Rotate(node * t)
50:     {
51:       node * gfa=t->fa->fa;
52:       t->getlr()?t->link(0,t->fa->link(1,t->s[0])):
53:         t->link(1,t->fa->link(0,t->s[1]));
54:       if(gfa) gfa->link(gfa->s[1]==t->fa,t);
55:       else t->fa=0,Root=t;
56:     }
57:
58:     node * splay(node * t,node * tar=NULL)
59:     {
60:       while(t->fa!=tar && t->fa->fa!=tar)
61:         t->getlr()==t->fa->getlr()?
62:         (Rotate(t->fa),Rotate(t)):
63:         (Rotate(t),    Rotate(t));
64:       if(t->fa!=tar) Rotate(t);
65:       return t;
66:     }
67:
68:     void Insert(node *& t,const _tp & x)
```

```
69:     {
70:       if(Find(x)) return ;
71:       node * templ, * tempr, * tempx;
72:       templ=Find_less(Root,x);
73:       tempr=Find_greater(Root,x);
74:       tempx=ALLOC(x);
75:       if(templ && tempr)
76:         splay(tempr,splay(templ))->link(0,tempx);
77:       else if(templ && !tempr)
78:         Root=tempx->link(0,splay(templ));
79:       else if(!templ && tempr)
80:         Root=tempx->link(1,splay(tempr));
81:       else
82:         Root=tempx;
83:     }
84:
85:     node * Find(node * t,const _tp & x)
86:     {
87:       node * last=0;
88:       while(t && x!=t->val)
89:       {
90:         last=t;
91:         if(x < t->val) t=t->s[0];
92:         else if(x > t->val) t=t->s[1];
93:       }
94:       if(t) splay(t);
95:       else if(last) splay(last);
96:       return t;
97:     }
98:
99:     node * Find_Max(node * t)
100:    { while(t && t->s[1]) t=t->s[1]; return t; }
101:    node * Find_Min(node * t)
102:    { while(t && t->s[0]) t=t->s[0]; return t; }
103:
104:
105:    void Delete(node *& t,const _tp & x)
106:    {
107:      node * tempx, * templ, * tempr;
108:      tempx=Find(Root,x);
109:      if(!tempx) return ;
110:      templ=Find_less(Root,x);
111:      tempr=Find_greater(Root,x);
112:      if(templ && tempr) splay(tempr,splay(templ))->s[0]=0;
113:      else if(templ && !tempr) splay(templ)->s[1]=0;
114:      else if(!templ && tempr) splay(tempr)->s[0]=0;
115:      else                     Root=0;
116:      RECYCLE(tempx);
117:    }
118:
```

```
119:   protected:
120:     node * Find_greater(node * t, _tp x)
121:     {
122:       node * temp=0, * last=0;
123:       while(t)
124:       {
125:         last=t;
126:         if(x<t->val) temp=t,t=t->s[0];
127:         else t=t->s[1];
128:       }
129:       if(temp) splay(temp);
130:       else if(last) splay(last);
131:       return temp;
132:     }
133:
134:     node * Find_less(node * t, _tp x)
135:     {
136:       node * temp=0, * last=0;
137:       while(t)
138:       {
139:         last=t;
140:         if(x<=t->val) t=t->s[0];
141:         else temp=t,t=t->s[1];
142:       }
143:       if(temp) splay(temp);
144:       else if(last) splay(last);
145:       return temp;
146:     }
147:
148:     void To_vector(node * t, std::vector<_tp> & vec)
149:     {
150:       if(!t) return ;
151:       _tp temp=Find_Min(Root)->val;
152:       for(int i=0;i<size;++i)
153:         vec.push_back(temp),temp=Find_greater(temp);
154:     }
155:
156:   public:
157: #ifdef Debug
158:     void Show_tree() { P(Root); }
159:     void P(node *t)
160:     { cout << "***************" << endl; _P(t);
161:       cout << "***************" << endl; cout << endl; }
162:     void _P(node * t)
163:     {
164:       if(!t) return ;
165:       if(t->s[0])
166:       {cout<<t->val<<" —L— >"<<t->s[0]->val<<endl;
167:         _P(t->s[0]);}
168:       if(t->s[1])
```

```
169:       {cout<<t->val<<" —R— >"<<t->s[1]->val<<endl;
170:         _P(t->s[1]);}
171:     }
172: #endif
173:     Splay()
174:     {
175:       U=new node[MAXN];
176:       Trash=new node*[MAXN];
177:       memset(U,0,sizeof(_tp)*MAXN);
178:       ALL=U; Top=0; Root=0; size=0;
179:     }
180:     ~Splay()
181:     {
182:       delete[] U;
183:       delete[] Trash;
184:     }
185:
186:     void Insert(const _tp & x) { Insert(Root,x); }
187:     bool Find(const _tp & x) { return Find(Root,x); }
188:     void Delete(const _tp & x) { Delete(Root,x); }
189:     int Size() { return size; }
190:     bool Empty() { return size==0; }
191:     _tp Find_less(const _tp & x)
192:     {
193:       node * temp=Find_less(Root,x);
194:       if(size) return temp?temp->val:Find_Min(Root)->val;
195:       return _tp();
196:     }
197:     _tp Find_greater(const _tp & x)
198:     {
199:       node * temp=Find_greater(Root,x);
200:       if(size) return temp?temp->val:Find_Max(Root)->val;
201:       return _tp();
202:     }
203:     std::vector<_tp> To_vector()
204:     {
205:       std::vector<_tp> vec;
206:       To_vector(Root,vec);
207:       return vec;
208:     }
209: };
210:
211: int main()
212: {
213:   Splay<int,1000000> T;
214:
215:   int n,x;
216:   char op[10];
217:   scanf("%d",&n);
218:   while(n--)
```

```
219:   {
220:     scanf("%s",op);
221:     scanf("%d",&x);
222:     if(op[0]=='i')
223:     {
224:       T.Insert(x);
225:       printf("%d\n",T.Size());
226:     }
227:     else if(op[0]=='d')
228:     {
229:       T.Delete(x);
230:       printf("%d\n",T.Size());
231:     }
232:     else if(op[0]=='f')
233:       printf(T.Find(x)?"True\n":"False\n");
234:     else if(op[0]=='l')
235:       printf("%d\n",T.Find_less(x));
236:     else if(op[0]=='g')
237:       printf("%d\n",T.Find_greater(x));
238:     //T.Show_tree();
239:   }
240:
241:   vector<int> vec=T.To_vector();
242:   for(auto it=vec.begin();it!=vec.end();++it)
243:     printf("%d ",*it);
244:   printf("\n");
245:
246:   return 0;
247: }
```

5.3 Splay 2

```
1: /*************************************************
2:   This Splay Tree will splay from top to bottom,
3:   and the Splay Tree we learned in class splays
4:   from bottom to top.(which will cost much more
5:   space and should use father node.So we choose
6:   to splay from top to bottom.The link of Tree
7:   Nodes may be different from the other.But it
8:   can be proved that the amortise time is also
9:   O(Log N).Besides it can reduce the time when we
10:  need to find a key.
11: *************************************************/
12: #include<iostream>
13: using namespace std;
14:
15: typedef struct SplayTree* ST;
16: struct SplayTree
```

```
{
  int Key;
  struct SplayTree* Right;
  struct SplayTree* Left;
};

static ST NullNode = NULL;

ST MakeEmpty(ST T);
ST Splay(ST T, int X);
ST FindMin(ST T);
ST FindMax(ST T);
ST Initialize();
ST Insert(ST T, int X);
ST Delete(ST T, int X);

ST RR(ST K2);
ST LL(ST K2);

int main()
{
  ST T = Initialize();
  int N;
  cin>>N;
  while(N--)
  {
    string op;
    int x;
    cin >> op >> x;
    if(op=="i") T=Insert(T,x);
    if(op=="d") T=Delete(T,x);
    if(op=="f")
    {
      T = Splay(T,x);
      cout << (T && T->Key==x?"True":"False") << endl;
    }
  }
}

ST Initialize()
{
  if(NullNode == NULL)
  {
    NullNode = new SplayTree;
    NullNode->Left = NullNode->Right = NullNode;
  }

  return NullNode;
}

```

```
67:
68: ST Splay(ST T,int X)
69: //Because the find process is to confirm the range of X.
70: {
71:   //So the nodes be linked into right tree later must be
72:   static struct SplayTree Header;
73:   //smaller than the former nodes.So we linked the new
74:   ST LeftTreeMax,  RightTreeMin;
75:   //nodes into the min node of Right 'Trees left child
76:   //(which is called right link).And into the max node of
77:   Header.Left = Header.Right = NullNode;
78:   //Left 'Trees Right child(left link).
79:   LeftTreeMax = RightTreeMin = &Header;
80:   NullNode->Key = X;
81:
82:   while(X != T->Key)
83:   {
84:     if(X < T->Key)
85:     {
86:       if(X < T->Left->Key)
87:       //This situation, we find that the Key we want to find
88:         T = RR(T);
89:         //is in Z and its SubTree. So we first single
90:       if(T->Left == NullNode)
91:       //rotation Y and then right link it.
92:         break;
93:
94:       RightTreeMin->Left = T;
95:       //This is used when Y has not left child or Y is
96:       RightTreeMin = T;
97:       //key we are find.We let Y be the root of
98:       T = T->Left;
99:       //the middle Tree and right link X.
100:    }
101:    else
102:    {
103:      if(X > T->Right->Key)
104:        T = LL(T);
105:      if(T->Right == NullNode)
106:        break;
107:
108:      LeftTreeMax->Right = T;
109:      LeftTreeMax = T;
110:      T = T->Right;
111:    }
112:  }
113:
114:  LeftTreeMax->Right = T->Left;
115:  //We we finally find X(X is the root of middle tree).
116:  RightTreeMin->Left = T->Right;
```

```
117:   //We combine the left middle and right tree.
118:   T->Left = Header.Right;
119:   //We right link 'Xs right sub tree and
120:   //  left link 'Xs left sub tree.
121:   T->Right = Header.Left;
122:   //Then we let right tree be 'Xs right sub tree
123:   //  and left tree be 'Xs left sub tree.
124:   //Then we build a new tree with the root X.
125:   return T;
126:   //(And if X is not in the tree,
127:   //  we let the key which is the most near X be the root.)
128: }
129:
130: ST Insert(ST T,int X)
131: {
132:   static ST NewNode = NULL;
133:
134:   if(NewNode == NULL)
135:   //If T is empty tree,let X be the root and return T.
136:   {
137:     NewNode = new SplayTree;
138:   }
139:   NewNode->Key = X;
140:
141:   if(T == NullNode)
142:   {
143:     NewNode->Left = NewNode->Right = NullNode;
144:     T = NewNode;
145:   }
146:   else
147:   {
148:     T = Splay(T, X);
149:     //Otherwise,Splay the Tree by X.If X is not in the tree,
150:     if(X < T->Key)
151:     //then the root must be a number nearly X.
152:     {
153:       //Compare X with it,
154:       //  and link it depend on the situation.
155:       NewNode->Left = T->Left;
156:       NewNode->Right = T;
157:       T->Left = NullNode;
158:       T = NewNode;
159:     }
160:     else if(X > T->Key)
161:     {
162:       NewNode->Right = T->Right;
163:       NewNode->Left = T;
164:       T->Right = NullNode;
165:       T = NewNode;
166:     }
```

```
167:     else return T;
168:   }
169:
170:   NewNode = NULL;
171:   return T;
172: }
173:
174: ST Delete(ST T, int X)
175: {
176:   ST NewTree;
177:   if(T != NullNode)
178:   {
179:     T = Splay(T, X);
180:     //First,Splay the Tree by X.
181:     //And if the root is X,
182:     //  we can know that X is in the Tree.
183:     if(X == T->Key)
184:     {
185:       if(T->Left == NullNode)
186:       //If X 'doesnt have left child,
187:       //  let New T = T->Right,then we delete X.
188:         NewTree = T->Right;
189:       else
190:       {
191:         NewTree = T->Left;
192:         NewTree = Splay(NewTree, X);
193:         //And if X have two child.
194:         //The we splay 'Xs left sub tree by X,
195:         NewTree->Right = T->Right;
196:        //which means that the root of new tree
197:        // ' doesnt have right sub tree.
198:       }
199:       //So we link 'Xs right sub tree to the new tree.
200:       delete T;
201:       T = NewTree;
202:     }
203:   }
204:
205:   return T;
206: }
207: ST RR(ST K2)
208: {
209:   ST K1;
210:   K1 = K2->Left;
211:   K2->Left = K1->Right;
212:   K1->Right = K2;
213:
214:   return K1;
215: }
216:
```

```
217: ST LL(ST K2)
218: {
219:   ST K1;
220:   K1 = K2->Right;
221:   K2->Right = K1->Left;
222:   K1->Left = K2;
223:
224:   return K1;
225: }
226:
227: ST MakeEmpty(ST T)
228: {
229:   if(T != NULL)
230:   {
231:     MakeEmpty(T->Left);
232:     MakeEmpty(T->Right);
233:     delete T;
234:   }
235:   return NULL;
236: }
237: ST FindMin(ST T)
238: {
239:   if(T != NULL)
240:     while(T->Left!=NULL)
241:       T = T->Left;
242:
243:   return T;
244: }
245: ST FindMax(ST T)
246: {
247:   if(T != NULL)
248:     while(T->Right!=NULL)
249:       T = T->Right;
250:
251:   return T;
252: }
```