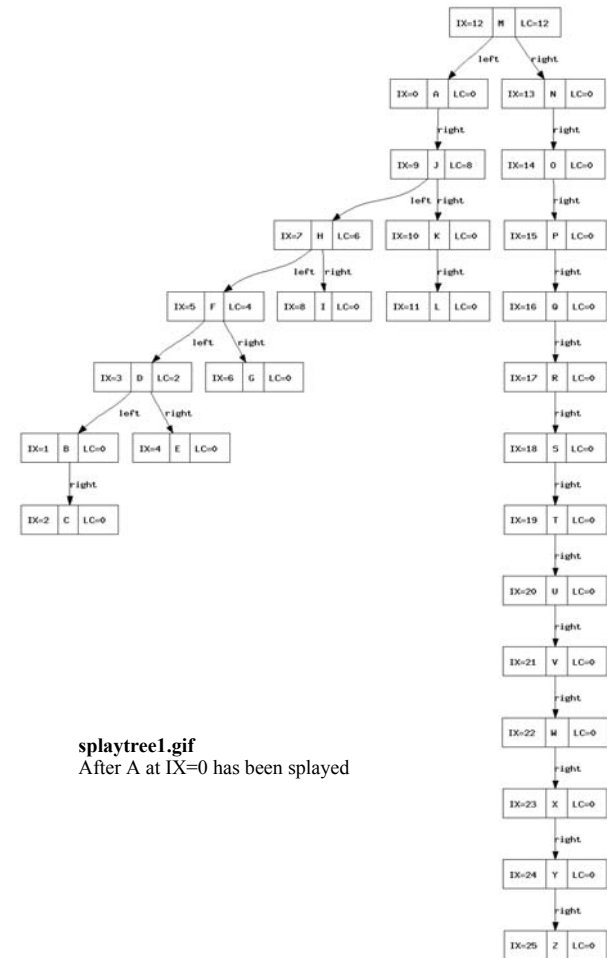
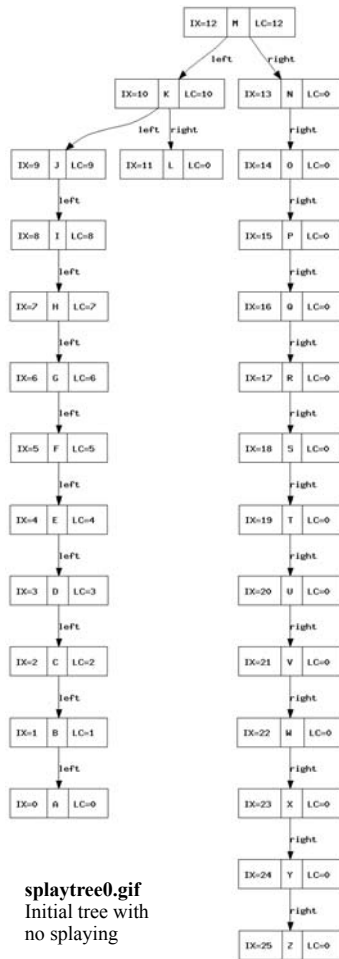
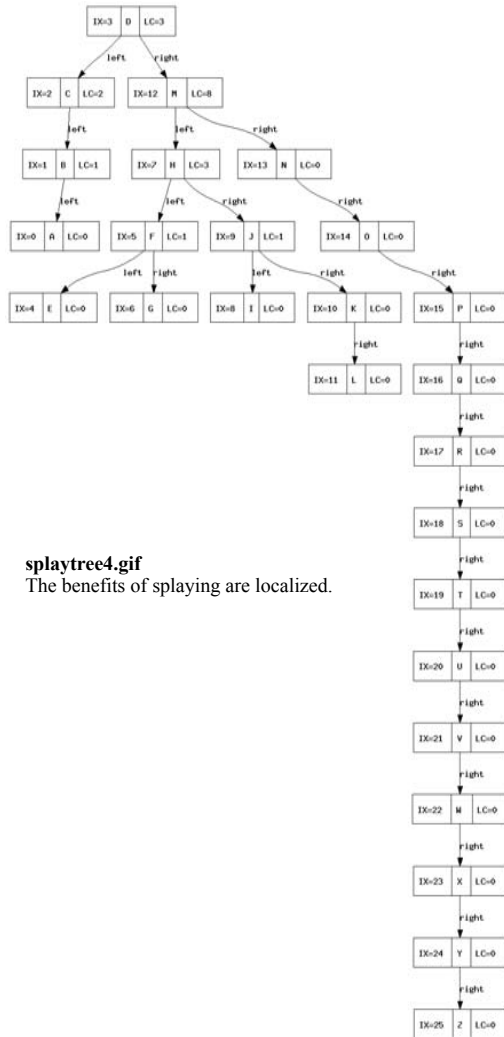


CSC 402 - Data Structures Fall, 2009

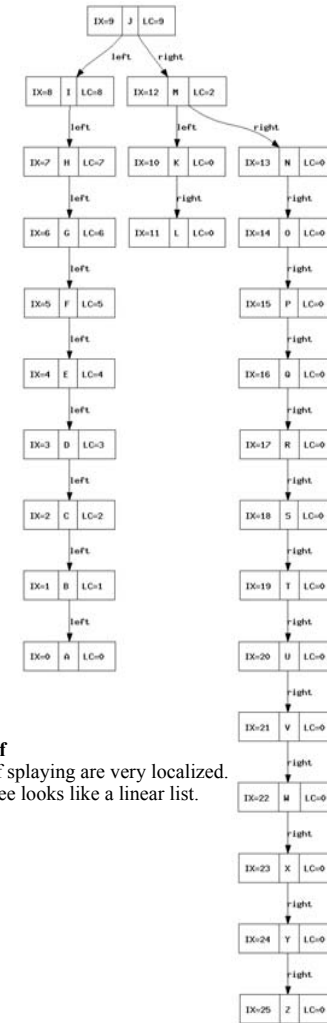
Graphs and listings from ~parson/AdvDataStructures/sequence_adt_tree_splay

Dr. Dale E. Parson, parson@kutztown.edu, http://faculty.kutztown.edu/parson





splaytree4.gif
The benefits of splaying are localized.



splaytree10.gif
The benefits of splaying are very localized.
Globally the tree looks like a linear list.

AdvDataStructures/sequence_adt_tree_splay/sequence_test.cxx

```
1  /* sequence_test.cxx -- test driver for sequence ADT.
2
3  CSC237, Fall, 2008, Dr. Dale Parson.
4  Update Fall, 2009 -- This test driver inserts string objects
5  rather than pointers to string objects into the test sequence.
6  */
7
8  static const char *usage =
9  "usage: sequence_test array | linkedlist | dblinkedlist";
10
11 // THE FOLLOWING EXIT CODES CONSTITUTE THE RETURN VALUES FROM
MAIN:
12 static const int SUCCESS = 0; /* successful program execution */
13 static const int USAGERR = 1; /* invalid command line usage */
14
15 #include <iostream>
16 #include <fstream>
17 #include <stdlib.h>
18 using namespace std;
19
20 // CLASS: NOTICE THAT WHEN USING COMPILING TEMPLATE CLASSES WITH
G++,
21 // INCLUDE THE .CXX (OR .CPP) SOURCE FILES RATHER THAN THE .H FILES
22 // FOR THE TEMPLATE CLASSES!
23 #include "sequence_interface.cxx"
24 #include "sequence_abstract_baseclass.cxx"
25 #include "sequence_arrayimpl.cxx"
26 #include "sequence_linkedlistimpl.cxx"
27 #include "sequence_dblylinkedlistimpl.cxx"
28 #include "sequence_binarytreeimpl.cxx"
29 // I took STL's vector out of here because passing a
30 // "template <typename ElemType>" from our parameterized type to STL's
31 // vector appears to confuse the compiler.
32
33 /*
34 Function: main
35
36 main is a test driver that demonstrates using implementation
37 classes of sequence ADT sequence_interface.
38 */
39
40 static const char *teststrings[] = {
41 "zero", "one", "two", "three", "four", "five", "six", "seven"
```

```
42 };
43
44 static void printseq(sequence_interface<string> *seq);
45
46 int
47 main(const int argc, char *argv[]) {
48     if (argc != 2) {
49         cerr << usage << endl;
50         exit(USAGERR);
51     }
52     char *dotfile = 0;
53     sequence_interface<string> *testseq;
54     bool istree = false;
55     string argtype(argv[1]);
56     if (argtype == "array") {
57         testseq = new sequence_arrayimpl<string>();
58         dotfile = "array.dot";
59     } else if (argtype == "linkedlist") {
60         testseq = new sequence_linkedlistimpl<string>();
61         dotfile = "linkedlist.dot";
62     } else if (argtype == "dbllinkedlist") {
63         testseq = new sequence_dblylinkedlistimpl<string>();
64         dotfile = "dblylinkedlist.dot";
65     } else if (argtype == "binarytree") {
66         testseq = new sequence_binarytreeimpl<string>();
67         dotfile = "binarytreeimpl.dot";
68         istree = true;
69     } else {
70         cerr << usage << endl;
71         exit(USAGERR);
72     }
73     int countstrings = sizeof(teststrings) / sizeof(teststrings[0]);
74     for (int i = 0; i < countstrings; i++) {
75         string newstr(teststrings[i]);
76         testseq->insert(newstr, 0);
77     }
78     string end("end");
79     string startplus1("start+1");
80     testseq->insert(end, -1);
81     testseq->insert(startplus1, 1);
82     printseq(testseq);
83     testseq->remove(5);
84     printseq(testseq);
85     testseq->dump(dotfile);
86     delete testseq; // test to make sure destructor doesn't explode
87     testseq = 0;
```

```

88 // Tests added 8/2009 for binary tree implementation only.
89 // Flesh out the tree. Following pairs have insertion point and value ;
90 if (istree) {
91     testseq = new sequence_binarytreeimpl<string>();
92     static struct testpair {
93         const char *content ;
94         int offset ;
95     } testpairs[] = { // The letters show their final indices.
96         {"M", 0},
97         {"K", 0},
98         {"J", 0},
99         {"I", 0},
100        {"H", 0},
101        {"G", 0},
102        {"F", 0},
103        {"E", 0},
104        {"D", 0},
105        {"C", 0},
106        {"B", 0},
107        {"A", 0},
108        {"L", 11},
109        {"N", -1},
110        {"O", -1},
111        {"P", -1},
112        {"Q", -1},
113        {"R", -1},
114        {"S", -1},
115        {"T", -1},
116        {"U", -1},
117        {"V", -1},
118        {"W", -1},
119        {"X", -1},
120        {"Y", -1},
121        {"Z", -1}
122    };
123    int countpairs = sizeof(testpairs) / sizeof(testpairs[0]) ;
124    for (int i = 0 ; i < countpairs ; i++) {
125        string newstr(testpairs[i].content);
126        testseq->insert(newstr, testpairs[i].offset);
127    }
128    cout << endl << "EXTENDED BINARY TREE BEFORE DELETIONS" << endl ;
129    printseq(testseq);
130    ofstream dotstreamtree1("extendedBinaryTreeBefore.dot",
131        ios_base::out | ios_base::trunc);
132    testseq->dump(dotstreamtree1);
133    dotstreamtree1.close();

```

```

134 // testseq->remove(6); // F@-1
135 // testseq->remove(1); // B@0
136 cout << endl << "EXTENDED BINARY TREE AFTER DELETIONS" << endl ;
137 printseq(testseq);
138 ofstream dotstreamtree2("extendedBinaryTreeAfter.dot",
139     ios_base::out | ios_base::trunc);
140 testseq->dump(dotstreamtree2);
141 dotstreamtree2.close();
142 // Splay for every node in a binary tree and generate a graph.
143 sequence_binarytreeimpl<string> *bintree =
144     dynamic_cast<sequence_binarytreeimpl<string>*>(testseq) ;
145 // bintree is non-NULL only if it is a sequence_binarytreeimpl *.
146 if (bintree != 0) {
147     testseq->dump("splaytree0.dot");
148     for (int i = 0 ; i < bintree->size() ; i++) {
149         bintree->splay(i);
150         string fname("splaytree");
151         char version[16];
152         sprintf(version, "%d", i+1);
153         fname = fname + version + ".dot" ;
154         testseq->dump(fname.c_str());
155     }
156 }
157 delete testseq ;
158 testseq = 0 ;
159 }
160 exit(SUCCESS);
161 }
162
163 static void printseq(sequence_interface<string> *seq) {
164     int limit = seq->size();
165     bool isValid ;
166     cout << "\n" << limit << " elements in sequence\n" ;
167     for (int i = 0 ; i < limit ; i++) {
168         cout << "\t" << seq->get(i, isValid).c_str() << endl ;
169     }
170 }

```

AdvDataStructures/sequence_adt_tree_splay/sequence_binarytreeimpl.h

```

1 /* sequence_binarytreeimpl.h -- Concrete class that implements
35
36 // Add a non-sequence-interface method to splay the tree using the node
37 // at offset. Return true if offset is valid, else false.
38 // A valid offset will not splay if it is too close to the root.
39 virtual bool splay(int offset);

```

```

77
78 int helpsplay(int offsetinsubtree, int subtreecount,
79 treenode * & subtreeeroot);
80
81 // Disallow copy construction and assignment operator.
82 sequence_binarytreeimpl(const sequence_binarytreeimpl &);
83 sequence_binarytreeimpl & operator=(const sequence_binarytreeimpl &);
84 };
85
86 #endif

```

AdvDataStructures/sequence_adt_tree_splay/sequence_binarytreeimpl.cxx

```

1 /* sequence_binarytreeimpl.cxx -- Method implementations for
2 sequence_binarytreeimpl.h.
3
4 CSC402, Fall, 2009, Dr. Dale Parson.
5 */
6
7 #ifndef SEQUENCE_BINARYTREEIMPL_CXX
8 #define SEQUENCE_BINARYTREEIMPL_CXX
9
10 #include "sequence_binarytreeimpl.h"
224
225 template <typename ElemType>
226 bool
227 sequence_binarytreeimpl<ElemType>::splay(int offset) {
228     bool result;
229     int realoff = this->getRealOffset(offset, false);
230     if (realoff >= 0) {
231         /*
232         cerr << "DEBUG START SPLAY AT " << realoff << endl;
233         */
234         helpsplay(realoff, this->count, root);
235         result = true;
236     } else {
237         result = false;
238     }
239     return result;
240 }
241
242 // A return of -1 from helpsplay means that the node to splay was not found.
243 // This condition will not occur because the offset is guaranteed to be valid.
244 // this return would be necessary in a key-based lookup.
245 // A non-negative int returned out of helpsplay is a 2-bit encoding.
246 // The bottom bit is 1 out of the node-to-be-splayed, 0 out of its parent.

```

```

247 // The second bit is 0 for the parent's left side, 1 for its right side.
248 template <typename ElemType>
249 int
250 sequence_binarytreeimpl<ElemType>::helpsplay(int offsetinsubtree,
251 int subtreecount, treenode * & subtreeeroot) {
252     static const int leftside = 0;
253     static const int rightside = 2;
254     if (offsetinsubtree == subtreeeroot->leftcount) {
255         // This is the node to be splayed. Tell its grandparent.
256         return 1;
257     }
258     int childside, myside;
259     if (offsetinsubtree < subtreeeroot->leftcount) {
260         // It resides in front of this node, recur down left subtree.
261         myside = leftside;
262         childside = helpsplay(offsetinsubtree, subtreeeroot->leftcount,
263 subtreeeroot->left);
264     } else {
265         // It resides after this node, recur down right subtree.
266         myside = rightside; // right side
267         childside = helpsplay(offsetinsubtree - subtreeeroot->leftcount - 1,
268 subtreecount - subtreeeroot->leftcount - 1, subtreeeroot->right);
269     }
270     if (childside == 1) { // I am the poor parent, return to grandparent;
271         return myside;
272     }
273     treenode *g = subtreeeroot, *p, *x;
274     // (g) grandparent, (p) parent, and (x) node to be splayed
275     // g, p and x will require leftcount adjustments. Get all the count data
276     // as soon as possible -- each case uses some subset of this data.
277     int pre_g_lc = g->leftcount, pre_g_rc = subtreecount - g->leftcount - 1;
278     if (myside == childside) { // This is one of the zig-zig cases
279         if (myside == leftside) { // zig-zig case p. 152 Fig. 4.48
280             p = g->left;
281             x = g->left->left;
282             /*
283             cerr << "DEBUG ZIGZIG LEFT " << g->content << " -> "
284             << p->content << " -> " << x->content << endl;
285             */
286             treenode *gleft = p->right; // assign into grandparent left, etc.
287             treenode *pleft = x->right;
288             treenode *pright = g;
289             treenode *xright = p;
290             // Retrieve necessary counts.
291             int pre_p_rc = g->leftcount - p->leftcount - 1;
292             int pre_x_rc = p->leftcount - x->leftcount - 1;

```

```

293 // Restructure the links:
294 g->left = gleft ;
295 g->leftcount = pre_p_rc ;
296 p->left = pleft ;
297 p->leftcount = pre_x_rc ;
298 p->right = pright ;
299 x->right = xright ;
300 } else { // the reflected zig-zig case
301 p = g->right ;
302 x = g->right->right ;
303 /*
304 cerr << "DEBUG ZIGZIG RIGHT " << g->content << " -> "
305 << p->content << " -> " << x->content << endl;
306 */
307 treenode *gright = p->left ; // assign into grandparent right, etc.
308 treenode *pright = x->left ;
309 treenode *pleft = g ;
310 treenode *xleft = p ;
311 // Update necessary counts.
312 // g keeps its old left
313 // p's new left is the sum of g's old left, g, and p's old left
314 // x's new left is the sum of p's new left + p + x's old leftcount
315 p->leftcount += (g->leftcount + 1) ;
316 x->leftcount += (p->leftcount + 1) ;
317 // Restructure the links:
318 g->right = gright ;
319 p->right = pright ;
320 p->left = pleft ;
321 x->left = xleft ;
322 }
323 } else if (myside == leftside) { // zig-zag case p. 152 Fig. 4.47
324 p = g->left ;
325 x = g->left->right ;
326 /*
327 cerr << "DEBUG ZIGZAG LEFT-RIGHT " << g->content << " -> "
328 << p->content << " -> " << x->content << endl;
329 */
330 treenode *gleft = x->right ;
331 treenode *pright = x->left ;
332 treenode *xleft = p ;
333 treenode *xright = g ;
334 // Update necessary counts.
335 // p keeps its old leftcount
336 // g's new leftcount is x's old rightcount
337 // x's new leftcount is p's leftcount + 1 + x's old leftcount
338 int pre_x_rc = g->leftcount - p->leftcount - x->leftcount - 2 ;

```

```

339 g->leftcount = pre_x_rc ;
340 x->leftcount += (p->leftcount + 1);
341 // Restructure the links:
342 g->left = gleft ;
343 p->right = pright ;
344 x->left = xleft ;
345 x->right = xright ;
346 cerr.flush();
347 } else { // the reflected zig-zag case
348 p = g->right ;
349 x = g->right->left ;
350 /*
351 cerr << "DEBUG ZIGZAG RIGHT-LEFT " << g->content << " -> "
352 << p->content << " -> " << x->content << endl;
353 */
354 treenode *gright = x->left ;
355 treenode *pleft = x->right ;
356 treenode *xright = p ;
357 treenode *xleft = g ;
358 // Update necessary counts.
359 // g keeps its old leftcount
360 // p's new leftcount is x's old rightcount
361 // x's new leftcount is g's leftcount + g + x's old leftcount
362 int pre_x_rc = p->leftcount - x->leftcount - 1 ;
363 p->leftcount = pre_x_rc ;
364 x->leftcount += (g->leftcount + 1);
365 // Restructure the links:
366 g->right = gright ;
367 p->left = pleft ;
368 x->right = xright ;
369 x->left = xleft ;
370 }
371 // x, the splayed node, is now the root, link to it & pass it up again
372 subtroot = x ;
373 return 1 ; // This may be resplayed again.
374 }
375
376
377 #endif

```