Listing 1: `a_b_tree.c`

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4
5  #define A 2
6  #define B 4
7  #define BLOCKSIZE 256
8
9  typedef int object_t;
10 typedef int key_t;
11
12 typedef struct tr_n_t { int           degree;
13                         int           height;
14                         key_t         key[B];
15                       struct tr_n_t * next[B];
16             /* possibly other information */ } tree_node_t;
17
18
19 tree_node_t *currentblock = NULL;
20 int     size_left;
21 tree_node_t *free_list = NULL;
22
23 tree_node_t *get_node()
24 { tree_node_t *tmp;
25   if( free_list != NULL )
26   {  tmp = free_list;
27      free_list = free_list->next[0];
28   }
29   else
30   {  if( currentblock == NULL || size_left == 0)
31      {  currentblock =
32                (tree_node_t *) malloc( BLOCKSIZE * sizeof(tree_node_t) );
33         size_left = BLOCKSIZE;
34      }
35      tmp = currentblock++;
36      size_left -= 1;
37   }
38   return( tmp );
39 }
40
41
42 void return_node(tree_node_t *node)
43 {  node->next[0] = free_list;
44    free_list = node;
45 }
46
47 tree_node_t *create_tree()
48 {  tree_node_t *tmp;
49    tmp = get_node();
50    tmp->height = 0;
```

```
51     tmp->degree = 0;
52     return( tmp );
53  }
54
55  void list_node(tree_node_t *tree)
56  { int i;
57    if(tree->height == 0 )
58    {  printf("leaf, degree %d, keys",tree->degree);
59       for(i = 0; i< tree->degree; i++)
60           printf(" %d", tree->key[i]);
61       printf(" end ");
62    }
63    else
64    {  printf("node height %d, degree %d, keys", tree->height, tree->degree);
65       for(i = 1; i< tree->degree; i++)
66           printf(" %d", tree->key[i]);
67       printf(" end ");
68    }
69  }
70
71
72  object_t *find(tree_node_t *tree, key_t query_key)
73  {  tree_node_t *current_node;
74     object_t *object;
75     current_node = tree;
76     while( current_node->height >= 0 )
77     {  int lower, upper; /* binary search among keys */
78        lower = 0;    upper = current_node->degree;
79        while( upper > lower +1 )
80        {  if( query_key < current_node->key[ (upper+lower)/2 ] )
81              upper = (upper+lower)/2;
82           else
83              lower = (upper+lower)/2;
84        }
85        if( current_node->height > 0)
86           current_node = current_node->next[lower];
87        else
88        {  /* block of height 0, contains the object pointers */
89           if( current_node->key[lower] == query_key )
90              object = (object_t *) current_node->next[lower];
91           else
92             object = NULL;
93           return( object );
94        }
95     }
96  }
97
98  int insert(tree_node_t *tree, key_t new_key, object_t *new_object)
99  {  tree_node_t *current_node, *insert_pt;
100    tree_node_t *node_stack[20]; int stack_p =0;
101    key_t  insert_key;
```

```
102     int finished;
103     current_node = tree;
104     if( tree->height == 0 && tree->degree == 0)
105     {  tree->key[0]  = new_key;
106        tree->next[0] = (tree_node_t *) new_object;
107        tree->degree  = 1;
108        return(0); /*insert in empty tree */
109     }
110     while( current_node->height > 0 ) /* not at leaf level */
111     {  int lower, upper; /* binary search among keys */
112        node_stack[stack_p++] = current_node ;
113        lower = 0;    upper = current_node->degree;
114        while( upper > lower +1 )
115        {  if( new_key < current_node->key[ (upper+lower)/2 ] )
116              upper = (upper+lower)/2;
117           else
118              lower = (upper+lower)/2;
119        }
120        current_node = current_node->next[lower];
121     } /* now current_node is leaf node in which we insert */
122     insert_pt = (tree_node_t *) new_object;
123     insert_key = new_key;
124     finished = 0;
125     while( !finished )
126     {  int i, start;
127        if( current_node->height>0)
128           start = 1; /* insertion in non-leaf starts at 1*/
129        else
130       start = 0; /* insertion in non-leaf starts at 0*/
131        if( current_node->degree < B ) /* node still has room */
132        {  /* move everything up to create the insertion gap */
133           i = current_node->degree;
134           while( (i > start)&&( current_node->key[i-1] > insert_key))
135       {  current_node->key[i]  = current_node->key[i-1];
136          current_node->next[i] = current_node->next[i-1];
137              i -= 1;
138           }
139           current_node->key[i]  = insert_key;
140       current_node->next[i] = insert_pt;
141           current_node->degree +=1;
142           finished = 1;
143        } /* end insert in non-full node */
144        else /* node is full, have to split the node*/
145        {  tree_node_t *new_node; int j, insert_done=0;
146           new_node = get_node();
147           i= B-1; j = (B-1)/2;
148           while(j>=0) /* copy upper half to new node */
149       {  if( insert_done || insert_key < current_node->key[i] )
150          {  new_node->next[j]  = current_node->next[i];
151                new_node->key[j--] = current_node->key[i--];
152             }
```

```
153          else
154          {  new_node->next[j]  = insert_pt;
155             new_node->key[j--] = insert_key;
156                 insert_done = 1;
157              }
158           } /* upper half done, insert in lower half, if necessary*/
159           while( !insert_done)
160       {   if( insert_key < current_node->key[i] && i >= start )
161           {  current_node->next[i+1] = current_node->next[i];
162                 current_node->key[i+1]  = current_node->key[i];
163                 i -=1;
164              }
165          else
166          {  current_node->next[i+1] = insert_pt;
167             current_node->key[i+1]  = insert_key;
168                 insert_done = 1;
169              }
170           } /*finished insertion */
171           current_node->degree = B+1 - ((B+1)/2);
172           new_node->degree = (B+1)/2;
173           new_node->height = current_node->height;
174           /* split nodes complete, now insert the new node above */
175           insert_pt = new_node;
176           insert_key = new_node->key[0];
177           if( stack_p >0 ) /* not at root; move one level up */
178       {   current_node = node_stack[ --stack_p ];
179           }
180           else /* splitting root: needs copy to keep root address*/
181       {   new_node =get_node();
182           for( i=0; i< current_node->degree; i++ )
183           {  new_node->next[i]  = current_node->next[i];
184                 new_node->key[i] = current_node->key[i];
185              }
186              new_node->height = current_node->height;
187              new_node->degree = current_node->degree;
188              current_node->height += 1;
189              current_node->degree  = 2;
190              current_node->next[0] = new_node;
191              current_node->next[1] = insert_pt;
192              current_node->key[1]  = insert_key;
193          finished =1;
194           } /* end splitting root */
195        } /* end node splitting */
196     } /* end of rebalancing */
197     return( 0 );
198  }
199
200  object_t *delete(tree_node_t *tree, key_t delete_key)
201  {  tree_node_t *current, *tmp_node;
202     tree_node_t *node_stack[20]; int index_stack[20];
203     int finished, i, j, stack_p =0;
```

```
204     current = tree;
205     while( current->height > 0 ) /* not at leaf level */
206     {  int lower, upper; /* binary search among keys */
207        lower = 0;    upper = current->degree;
208        while( upper > lower +1 )
209        {  if( delete_key < current->key[ (upper+lower)/2 ] )
210              upper = (upper+lower)/2;
211           else
212              lower = (upper+lower)/2;
213        }
214        index_stack[stack_p]  = lower ;
215        node_stack[stack_p++] = current ;
216        current = current->next[lower];
217     } /* now current is leaf node from which we delete */
218     for( i=0; i < current->degree ; i++ )
219        if( current->key[i] == delete_key )
220           break;
221     if( i == current->degree )
222     {  return( NULL ); /* delete failed; key does not exist */
223     }
224     else /* key exists, now delete from leaf node */
225     {  object_t *del_object;
226        del_object = (object_t *) current->next[i];
227        current->degree -=1;
228        while( i < current->degree )
229        {  current->next[i] = current->next[i+1];
230           current->key[i]  = current->key[i+1];
231           i+=1;
232        } /* deleted from node, now rebalance */
233        finished = 0;
234        while( ! finished )
235        {  if(current->degree >= A )
236      {  finished = 1; /* node still full enough, can stop */
237           }
238           else /* node became underful */
239      {  if( stack_p == 0 ) /* current is root */
240           {  if(current->degree >= 2 )
241             finished = 1; /* root still necessary */
242              else if ( current->height == 0 )
243             finished = 1; /* deleting last keys from root */
244                  else /* delete root, copy to keep address */
245              {  tmp_node = current->next[0];
246                 for( i=0; i< tmp_node->degree; i++ )
247             {  current->next[i] = tmp_node->next[i];
248                       current->key[i] = tmp_node->key[i];
249                   }
250                   current->degree = tmp_node->degree;
251                   current->height = tmp_node->height;
252                   return_node( tmp_node ); finished = 1;
253                }
254              } /* done with root */
```

```
255         else /*  delete from non-root node */
256         {  tree_node_t *upper, *neighbor; int curr;
257            upper = node_stack[ --stack_p ];
258                curr  = index_stack[stack_p];
259            if( curr < upper->degree -1 ) /* not last*/
260            {  neighbor = upper->next[curr+1];
261                   if( neighbor->degree >A )
262                   {  /* sharing possible */
263              i = current->degree;
264                      if( current->height > 0 )
265                         current->key[i] =  upper->key[curr+1];
266                      else /* on leaf level, take leaf key */
267                      {  current->key[i]  = neighbor->key[0];
268                     neighbor->key[0] = neighbor->key[1];
269                      }
270                      current->next[i] = neighbor->next[0];
271                      upper->key[curr+1] = neighbor->key[1];
272                      neighbor->next[0] = neighbor->next[1];
273                      for( j = 2; j < neighbor->degree; j++)
274              {  neighbor->next[j-1] = neighbor->next[j];
275                         neighbor->key[j-1]  = neighbor->key[j];
276                      }
277              neighbor->degree -=1; current->degree+=1;
278                      finished  =1;
279                   } /* sharing complete */
280                   else /* must join */
281            {  i = current->degree;
282                      if( current->height > 0 )
283                         current->key[i] =  upper->key[curr+1];
284                      else /* on leaf level, take leaf key */
285                         current->key[i] =  neighbor->key[0];
286                      current->next[i] = neighbor->next[0];
287                      for( j = 1; j < neighbor->degree; j++)
288              {  current->next[++i] = neighbor->next[j];
289                         current->key[i]  = neighbor->key[j];
290                      }
291                      current->degree = i+1;
292                      return_node( neighbor );
293                      upper->degree -=1; i = curr+1;
294                      while( i < upper->degree )
295                      {  upper->next[i] = upper->next[i+1];
296                         upper->key[i]  = upper->key[i+1];
297                         i +=1;
298                      } /* deleted from upper, now propagate up */
299                      current = upper;
300                   } /* end of share/joining  if-else*/
301               }
302            else /* current is last entry in upper */
303            {  neighbor = upper->next[curr-1];
304                   if( neighbor->degree >A )
305                   {  /* sharing possible */
```

```
306                     for( j = current->degree; j > 1; j--)
307               {  current->next[j] = current->next[j-1];
308                        current->key[j]  = current->key[j-1];
309                     }
310                     current->next[1] = current->next[0];
311               i = neighbor->degree;
312                     current->next[0] = neighbor->next[i-1];
313                     if( current->height > 0 )
314                     {  current->key[1] =  upper->key[curr];
315                     }
316                     else /* on leaf level, take leaf key */
317                     {  current->key[1] =  current->key[0];
318                  current->key[0] =  neighbor->key[i-1];
319                     }
320               upper->key[curr] = neighbor->key[i-1];
321               neighbor->degree -=1; current->degree+=1;
322                     finished  =1;
323                  } /* sharing complete */
324                  else /* must join */
325            {  i = neighbor->degree;
326                     if( current->height > 0 )
327                        neighbor->key[i] =  upper->key[curr];
328                     else /* on leaf level, take leaf key */
329                        neighbor->key[i] =  current->key[0];
330                     neighbor->next[i] = current->next[0];
331                     for( j = 1; j < current->degree; j++)
332               {  neighbor->next[++i] = current->next[j];
333                        neighbor->key[i]  = current->key[j];
334                     }
335                     neighbor->degree = i+1;
336                     return_node( current );
337                     upper->degree -=1;
338                     /* deleted from upper, now propagate up */
339                     current = upper;
340                  } /* end of share/joining  if-else */
341               } /* end of current is (not) last in upper if-else*/
342         } /* end of delete root/non-root if-else */
343       } /* end of full/underfull if-else */
344        } /* end of while not finished */
345        return( del_object );
346    } /* end of delete object exists if-else */
347 }
348
349 void check_tree(tree_node_t *tree, int lower, int upper)
350 {  int i; int seq_error = 0;
351    if( tree->height > 0 )
352    {  printf("(%d:", tree->height);
353       for(i = 1; i< tree->degree; i++ )
354         printf(" %d", tree->key[i]);
355       for(i = 1; i< tree->degree; i++ )
356       {  if( !( lower <= tree->key[i] && tree->key[i]<upper) )
```

```
357           {  seq_error = 1;
358           }
359       }
360       if( seq_error == 1)
361         printf(":?");
362       printf(")");
363       check_tree(tree->next[0], lower, tree->key[1]);
364       for(i = 1; i< tree->degree-1; i++ )
365         check_tree(tree->next[i], tree->key[i], tree->key[i+1]);
366       check_tree(tree->next[tree->degree-1],
367               tree->key[tree->degree-1], upper);
368    }
369    else
370    {  printf("[");
371       for(i = 0; i< tree->degree; i++ )
372         printf(" %d", tree->key[i]);
373       for(i = 0; i< tree->degree; i++ )
374       {  if( !( lower <= tree->key[i] && tree->key[i]<upper) )
375          {  seq_error = 1;
376          }
377       }
378       if( seq_error == 1)
379         printf(":?");
380       printf("]");
381    }
382 }
383
384 int main()
385 {  tree_node_t *searchtree;
386    char nextop;
387    searchtree = create_tree();
388    printf("Made Tree: (%d,%d)-Tree\n", A, B);
389    while( (nextop = getchar())!= 'q' )
390    { if( nextop == 'i' )
391      { int inskey, *insobj, success;
392        insobj = (int *) malloc(sizeof(int));
393        scanf(" %d", &inskey);
394        *insobj = 10*inskey+2;
395        success = insert( searchtree, inskey, insobj );
396        if ( success == 0 )
397          printf("  insert successful, key = %d, object value = %d\n",
398               inskey, *insobj);
399        else
400            printf("  insert failed, success = %d\n", success);
401      }
402      if( nextop == 'f' )
403      { int findkey, *findobj;
404        scanf(" %d", &findkey);
405        findobj = find( searchtree, findkey);
406        if( findobj == NULL )
407          printf("  find failed, for key %d\n", findkey);
```

```
408         else
409           printf("  find successful, found object %d\n", *findobj);
410       }
411       if( nextop == 'd' )
412       { int delkey, *delobj;
413         scanf(" %d", &delkey);
414         delobj = delete( searchtree, delkey);
415         if( delobj == NULL )
416           printf("  delete failed for key %d\n", delkey);
417         else
418           printf("  delete successful, deleted object %d\n",
419               *delobj);
420       }
421       if( nextop == '?' )
422       {  printf("  Checking tree\n");
423          check_tree(searchtree,-1000,1000);
424          printf("\n");
425          /*if( searchtree->left != NULL )
426        printf("key in root is %d, height of tree is %d\n",
427        searchtree->key, searchtree->height );*/
428          printf("  Finished Checking tree\n");
429      }
430     }
431     return(0);
432 }
```