Listing 1: `a_b_tree.c`

```c
#include <stdio.h>
#include <stdlib.h>


#define A 2
#define B 4
#define BLOCKSIZE 256

typedef int object_t;
typedef int key_t;

typedef struct tr_n_t { int           degree;
                        int           height;
                        key_t         key[B];
                     struct tr_n_t * next[B];
             /* possibly other information */ } tree_node_t;


tree_node_t *currentblock = NULL;
int    size_left;
tree_node_t *free_list = NULL;

tree_node_t *get_node()
{ tree_node_t *tmp;
  if( free_list != NULL )
  {  tmp = free_list;
     free_list = free_list->next[0];
  }
  else
  {  if( currentblock == NULL || size_left == 0)
     {  currentblock =
                (tree_node_t *) malloc( BLOCKSIZE * sizeof(tree_node_t) );
        size_left = BLOCKSIZE;
     }
     tmp = currentblock++;
     size_left -= 1;
  }
  return( tmp );
}


void return_node(tree_node_t *node)
{  node->next[0] = free_list;
   free_list = node;
}

tree_node_t *create_tree()
{  tree_node_t *tmp;
   tmp = get_node();
   tmp->height = 0;
   tmp->degree = 0;
   return( tmp );
}

void list_node(tree_node_t *tree)
{ int i;
  if(tree->height == 0 )
  {   printf("leaf, degree %d, keys",tree->degree);
      for(i = 0; i< tree->degree; i++)
          printf(" %d", tree->key[i]);
      printf(" end ");
  }
  else
  {   printf("node height %d, degree %d, keys", tree->height, tree->degree);
      for(i = 1; i< tree->degree; i++)
          printf(" %d", tree->key[i]);
      printf(" end ");
  }
}


object_t *find(tree_node_t *tree, key_t query_key)
{  tree_node_t *current_node;
   object_t *object;
   current_node = tree;
   while( current_node->height >= 0 )
   {  int lower, upper; /* binary search among keys */
      lower = 0;   upper = current_node->degree;
      while( upper > lower +1 )
      {  if( query_key < current_node->key[ (upper+lower)/2 ] )
            upper = (upper+lower)/2;
         else
            lower = (upper+lower)/2;
      }
      if( current_node->height > 0)
         current_node = current_node->next[lower];
      else
      {  /* block of height 0, contains the object pointers */
         if( current_node->key[lower] == query_key )
            object = (object_t *) current_node->next[lower];
         else
        object = NULL;
         return( object );
      }
   }
}

int insert(tree_node_t *tree, key_t new_key, object_t *new_object)
{  tree_node_t *current_node, *insert_pt;
   tree_node_t *node_stack[20]; int stack_p =0;
   key_t  insert_key;
   int finished;
   current_node = tree;
   if( tree->height == 0 && tree->degree == 0)
   {  tree->key[0]  = new_key;
      tree->next[0] = (tree_node_t *) new_object;
      tree->degree  = 1;
      return(0); /*insert in empty tree */
   }
   while( current_node->height > 0 ) /* not at leaf level */
   {  int lower, upper; /* binary search among keys */
      node_stack[stack_p++] = current_node ;
      lower = 0;   upper = current_node->degree;
      while( upper > lower +1 )
      {  if( new_key < current_node->key[ (upper+lower)/2 ] )
            upper = (upper+lower)/2;
         else
            lower = (upper+lower)/2;
      }
      current_node = current_node->next[lower];
   } /* now current_node is leaf node in which we insert */
   insert_pt = (tree_node_t *) new_object;
   insert_key = new_key;
   finished = 0;
   while( !finished )
   {  int i, start;
      if( current_node->height>0)
         start = 1; /* insertion in non-leaf starts at 1*/
      else
     start = 0; /* insertion in non-leaf starts at 0*/
      if( current_node->degree < B ) /* node still has room */
      {  /* move everything up to create the insertion gap */
         i = current_node->degree;
         while( (i > start)&&( current_node->key[i-1] > insert_key))
     {  current_node->key[i]  = current_node->key[i-1];
        current_node->next[i] = current_node->next[i-1];
            i -= 1;
         }
         current_node->key[i]  = insert_key;
     current_node->next[i] = insert_pt;
         current_node->degree +=1;
         finished = 1;
      } /* end insert in non-full node */
      else /* node is full, have to split the node*/
      {  tree_node_t *new_node; int j, insert_done=0;
         new_node = get_node();
         i= B-1; j = (B-1)/2;
         while(j>=0) /* copy upper half to new node */
     {  if( insert_done || insert_key < current_node->key[i] )
        {  new_node->next[j]  = current_node->next[i];
               new_node->key[j--] = current_node->key[i--];
            }
        else
        {  new_node->next[j]   = insert_pt;
           new_node->key[j--] = insert_key;
               insert_done = 1;
            }
         } /* upper half done, insert in lower half, if necessary*/
         while( !insert_done)
     {  if( insert_key < current_node->key[i] && i >= start )
        {  current_node->next[i+1] = current_node->next[i];
               current_node->key[i+1]   = current_node->key[i];
               i -=1;
            }
        else
        {  current_node->next[i+1] = insert_pt;
           current_node->key[i+1]   = insert_key;
               insert_done = 1;
            }
         } /*finished insertion */
         current_node->degree = B+1 - ((B+1)/2);
         new_node->degree = (B+1)/2;
         new_node->height = current_node->height;
         /* split nodes complete, now insert the new node above */
         insert_pt = new_node;
         insert_key = new_node->key[0];
         if( stack_p >0 ) /* not at root; move one level up */
     {  current_node = node_stack[ --stack_p ];
         }
         else /* splitting root: needs copy to keep root address*/
     {  new_node =get_node();
        for( i=0; i< current_node->degree; i++ )
        {  new_node->next[i]   = current_node->next[i];
               new_node->key[i] = current_node->key[i];
            }
            new_node->height = current_node->height;
            new_node->degree = current_node->degree;
            current_node->height += 1;
            current_node->degree  = 2;
            current_node->next[0] = new_node;
            current_node->next[1] = insert_pt;
            current_node->key[1]  = insert_key;
        finished =1;
         } /* end splitting root */
      } /* end node splitting */
   } /* end of rebalancing */
   return( 0 );
}

object_t *delete(tree_node_t *tree, key_t delete_key)
{  tree_node_t *current, *tmp_node;
   tree_node_t *node_stack[20]; int index_stack[20];
   int finished, i, j, stack_p =0;
   current = tree;
   while( current->height > 0 ) /* not at leaf level */
   {  int lower, upper; /* binary search among keys */
      lower = 0;    upper = current->degree;
      while( upper > lower +1 )
      {  if( delete_key < current->key[ (upper+lower)/2 ] )
            upper = (upper+lower)/2;
         else
            lower = (upper+lower)/2;
      }
      index_stack[stack_p]  = lower ;
      node_stack[stack_p++] = current ;
      current = current->next[lower];
   } /* now current is leaf node from which we delete */
   for( i=0; i < current->degree ; i++ )
      if( current->key[i] == delete_key )
         break;
   if( i == current->degree )
   {  return( NULL ); /* delete failed; key does not exist */
   }
   else /* key exists, now delete from leaf node */
   {  object_t *del_object;
      del_object = (object_t *) current->next[i];
      current->degree -=1;
      while( i < current->degree )
      {  current->next[i] = current->next[i+1];
         current->key[i]  = current->key[i+1];
         i+=1;
      } /* deleted from node, now rebalance */
      finished = 0;
      while( ! finished )
      {  if(current->degree >= A )
     {  finished = 1; /* node still full enough, can stop */
         }
         else /* node became underful */
     {  if( stack_p == 0 ) /* current is root */
        {  if(current->degree >= 2 )
          finished = 1; /* root still necessary */
           else if ( current->height == 0 )
          finished = 1; /* deleting last keys from root */
               else /* delete root, copy to keep address */
           {  tmp_node = current->next[0];
              for( i=0; i< tmp_node->degree; i++ )
          {  current->next[i] = tmp_node->next[i];
                     current->key[i] = tmp_node->key[i];
                  }
                  current->degree = tmp_node->degree;
                  current->height = tmp_node->height;
                  return_node( tmp_node ); finished = 1;
               }
            } /* done with root */
        else /*  delete from non-root node */
        {  tree_node_t *upper, *neighbor; int curr;
           upper = node_stack[ --stack_p ];
               curr  = index_stack[stack_p];
           if( curr < upper->degree -1 ) /* not last*/
           {  neighbor = upper->next[curr+1];
                  if( neighbor->degree >A )
                  {  /* sharing possible */
             i = current->degree;
                     if( current->height > 0 )
                        current->key[i] =  upper->key[curr+1];
                     else /* on leaf level, take leaf key */
                     {  current->key[i]  = neighbor->key[0];
                neighbor->key[0] = neighbor->key[1];
                     }
                     current->next[i] = neighbor->next[0];
                     upper->key[curr+1] = neighbor->key[1];
                     neighbor->next[0] = neighbor->next[1];
                     for( j = 2; j < neighbor->degree; j++)
             {  neighbor->next[j-1] = neighbor->next[j];
                        neighbor->key[j-1]  = neighbor->key[j];
                     }
             neighbor->degree -=1; current->degree+=1;
                     finished  =1;
                  } /* sharing complete */
                  else /* must join */
          {  i = current->degree;
                     if( current->height > 0 )
                        current->key[i] =  upper->key[curr+1];
                     else /* on leaf level, take leaf key */
                        current->key[i] =  neighbor->key[0];
                     current->next[i] = neighbor->next[0];
                     for( j = 1; j < neighbor->degree; j++)
             {  current->next[++i] = neighbor->next[j];
                        current->key[i]  = neighbor->key[j];
                     }
                     current->degree = i+1;
                     return_node( neighbor );
                     upper->degree -=1; i = curr+1;
                     while( i < upper->degree )
                     {  upper->next[i] = upper->next[i+1];
                        upper->key[i]  = upper->key[i+1];
                        i +=1;
                     } /* deleted from upper, now propagate up */
                     current = upper;
                  } /* end of share/joining  if-else*/
               }
           else /* current is last entry in upper */
           {  neighbor = upper->next[curr-1];
                  if( neighbor->degree >A )
                  {  /* sharing possible */
```

```
                     for( j = current->degree; j > 1; j--)
             {  current->next[j] = current->next[j-1];
                        current->key[j]  = current->key[j-1];
                     }
                     current->next[1] = current->next[0];
             i = neighbor->degree;
                     current->next[0] = neighbor->next[i-1];
                     if( current->height > 0 )
                     {  current->key[1] =  upper->key[curr];
                     }
                     else /* on leaf level, take leaf key */
                     {  current->key[1] =  current->key[0];
                current->key[0] =  neighbor->key[i-1];
                     }
             upper->key[curr] = neighbor->key[i-1];
             neighbor->degree -=1; current->degree+=1;
                     finished  =1;
                  } /* sharing complete */
                  else /* must join */
          {  i = neighbor->degree;
                     if( current->height > 0 )
                        neighbor->key[i] =  upper->key[curr];
                     else /* on leaf level, take leaf key */
                        neighbor->key[i] =  current->key[0];
                     neighbor->next[i] = current->next[0];
                     for( j = 1; j < current->degree; j++)
             {  neighbor->next[++i] = current->next[j];
                        neighbor->key[i]  = current->key[j];
                     }
                     neighbor->degree = i+1;
                     return_node( current );
                     upper->degree -=1;
                     /* deleted from upper, now propagate up */
                     current = upper;
                  } /* end of share/joining  if-else */
               } /* end of current is (not) last in upper if-else*/
        } /* end of delete root/non-root if-else */
     } /* end of full/underfull if-else */
      } /* end of while not finished */
      return( del_object );
   } /* end of delete object exists if-else */
}

void check_tree(tree_node_t *tree, int lower, int upper)
{  int i; int seq_error = 0;
   if( tree->height > 0 )
   {  printf("(%d:", tree->height);
      for(i = 1; i< tree->degree; i++ )
        printf(" %d", tree->key[i]);
      for(i = 1; i< tree->degree; i++ )
      {  if( !( lower <= tree->key[i] && tree->key[i]<upper) )
```

```
         {   seq_error = 1;
         }
      }
      if( seq_error == 1)
        printf(":?");
      printf(")");
      check_tree(tree->next[0], lower, tree->key[1]);
      for(i = 1; i< tree->degree-1; i++ )
        check_tree(tree->next[i], tree->key[i], tree->key[i+1]);
      check_tree(tree->next[tree->degree-1],
              tree->key[tree->degree-1], upper);
   }
   else
   {  printf("[");
      for(i = 0; i< tree->degree; i++ )
        printf(" %d", tree->key[i]);
      for(i = 0; i< tree->degree; i++ )
      {   if( !( lower <= tree->key[i] && tree->key[i]<upper) )
         {   seq_error = 1;
         }
      }
      if( seq_error == 1)
        printf(":?");
      printf("]");
   }
}

int main()
{  tree_node_t *searchtree;
   char nextop;
   searchtree = create_tree();
   printf("Made Tree: (%d,%d)-Tree\n", A, B);
   while( (nextop = getchar())!= 'q' )
   { if( nextop == 'i' )
     { int inskey,  *insobj, success;
       insobj = (int *) malloc(sizeof(int));
       scanf(" %d", &inskey);
       *insobj = 10*inskey+2;
       success = insert( searchtree, inskey, insobj );
       if ( success == 0 )
         printf("  insert successful, key = %d, object value = %d\n",
              inskey, *insobj);
       else
           printf("  insert failed, success = %d\n", success);
     }
     if( nextop == 'f' )
     { int findkey, *findobj;
       scanf(" %d", &findkey);
       findobj = find( searchtree, findkey);
       if( findobj == NULL )
         printf("  find failed, for key %d\n", findkey);
       else
         printf("  find successful, found object %d\n", *findobj);
     }
     if( nextop == 'd' )
     { int delkey, *delobj;
       scanf(" %d", &delkey);
       delobj = delete( searchtree, delkey);
       if( delobj == NULL )
         printf("  delete failed for key %d\n", delkey);
       else
         printf("  delete successful, deleted object %d\n",
             *delobj);
     }
     if( nextop == '?' )
     {  printf("  Checking tree\n");
        check_tree(searchtree,-1000,1000);
        printf("\n");
        /*if( searchtree->left != NULL )
      printf("key in root is %d, height of tree is %d\n",
      searchtree->key, searchtree->height );*/
        printf("  Finished Checking tree\n");
    }
   }
   return(0);
}
```