Practical 2
Implementation of B+ Tree

Gahan M. Saraiya, 18MCEC10

18mcec10@nirmauni.ac.in

I. Introduction

Aim of this practical is to analyze the time complexity of sorting algorithms for various input size.

Algorithms analyzed are listed below:

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort
- Heap Sort

II. Implementation

I. Utility *utility.h*

```
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_UTILITY_H
#define DSA_LAB_UTILITY_H

#include <string.h>
#include <stdarg.h>

int write_log(const char *format, ...) {
    if(DEBUG) {
        va_list args;
        va_start (args, format);
        vprintf(format, args);
        va_end (args);
    }
}
```

```
19
20  int *get_min_max(int *array, int no_of_elements, int min_max[]){
21      // get minimum and maximum of array
22  //    printf("elements of array: ");
23      for(int i=0; i<no_of_elements; i++){
24  //        printf("%d ", *(array + i));
25          if (*(array + i) < min_max[0])
26              min_max[0] = *(array + i);
27          if (*(array + i) > min_max[1])
28              min_max[1] = *(array + i);
29      }
30      return min_max;
31  }
32
33  int display_array(int *array, int no_of_elements){
34      // display given array of given size(no. of elements require because sizeof()
        ↪  returns max bound value)
35      write_log(": ");
36      for(int i=0; i<no_of_elements; i++){
37          write_log( "%d ", *(array + i));
38      }
39      return 0;
40  }
41
42
43  void swap(int *one, int *two){
44      // swap function to swap elements by location/address
45      int temp = *one;
46      *one = *two;
47      *two = temp;
48  }
49
50  char** split_string(char* str) {
51      // split string by separator space
52      char** splits = NULL;
53      char* token = strtok(str, " ");
54      int spaces = 0;
55
56      while (token) {
57          splits = realloc(splits, sizeof(char*) * ++spaces);
58          if (!splits) {
59              return splits;
60          }
61          splits[spaces - 1] = token;
62          token = strtok(NULL, " ");
63      }
64      return splits;
65  }
```

```

void read_file_input() {
    // under development function to read inputs from file
    int ptr[100], count = 0, i, ar_count;
    char c[100];
    FILE *fp = fopen("file.in", "r");

    char in = fgetc(fp);
    // ar_count = (int) (in - '0');
    printf("\narr\n");
    while (in != EOF){
        if ((int) (in -'0') == -16){
            printf("\nspace\n");
        }
        else{
            printf("%c - %d\n",in,  (int) (in - '0'));
        }
        in = fgetc(fp);
    }
    printf("\n\n");
    fclose (fp);
}

#endif //DSA_LAB_UTILITY_H
```

II. Constants *constant.h*

```
//
// Created by jarvis on 17/8/18.
//

#ifndef DSA_LAB_CONSTANT_H
#define DSA_LAB_CONSTANT_H
#define TEST_NUM 5000
#define DEBUG 0

#endif //DSA_LAB_CONSTANT_H
```

III. Main Program - *bPlusTree.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>
#ifdef _WIN32
#define bool char
```

```
#define false 0
#define true 1
#endif

// Default order is 4.
#define DEFAULT_ORDER 4
#define BUFFER_SIZE 256

typedef struct record {
        int value;
} record;

typedef struct node {
        void ** pointers;
        int * keys;
        struct node * parent;
        bool is_leaf;
        int num_keys;
        struct node * next; // Used for queue.
} node;


int order = DEFAULT_ORDER;

node * queue = NULL;

bool verbose_output = false;


void license_notice(void);
void usage_1(void);
void usage_2(void);
void usage_3(void);
void enqueue(node * new_node);
node * dequeue(void);
int height(node * const root);
int path_to_root(node * const root, node * child);
void print_leaves(node * const root);
void print_tree(node * const root);
void find_and_print(node * const root, int key, bool verbose);
void find_and_print_range(node * const root, int range1, int range2, bool
 ↪  verbose);
int find_range(node * const root, int key_start, int key_end, bool verbose,
                int returned_keys[], void * returned_pointers[]);
node * find_leaf(node * const root, int key, bool verbose);
record * find(node * root, int key, bool verbose, node ** leaf_out);
int cut(int length);

```

```
54  // Insertion.
55
56  record * make_record(int value);
57  node * make_node(void);
58  node * make_leaf(void);
59  int get_left_index(node * parent, node * left);
60  node * insert_into_leaf(node * leaf, int key, record * pointer);
61  node * insert_into_leaf_after_splitting(node * root, node * leaf, int key,
62                                          record * pointer);
63  node * insert_into_node(node * root, node * parent,
64                  int left_index, int key, node * right);
65  node * insert_into_node_after_splitting(node * root, node * parent,
66                                          int left_index,
67                  int key, node * right);
68  node * insert_into_parent(node * root, node * left, int key, node * right);
69  node * insert_into_new_root(node * left, int key, node * right);
70  node * start_new_tree(int key, record * pointer);
71  node * insert(node * root, int key, int value);
72
73  void enqueue(node * new_node) {
74          node * c;
75          if (queue == NULL) {
76                  queue = new_node;
77                  queue->next = NULL;
78          }
79          else {
80                  c = queue;
81                  while(c->next != NULL) {
82                          c = c->next;
83                  }
84                  c->next = new_node;
85                  new_node->next = NULL;
86          }
87  }
88
89
90  node * dequeue(void) {
91          node * n = queue;
92          queue = queue->next;
93          n->next = NULL;
94          return n;
95  }
96
97
98  void print_leaves(node * const root) {
99          if (root == NULL) {
100                 printf("Empty tree.\n");
101                 return;
```

```
102         }
103         int i;
104         node * c = root;
105         while (!c->is_leaf)
106                 c = c->pointers[0];
107         while (true) {
108                 for (i = 0; i < c->num_keys; i++) {
109                         if (verbose_output)
110                                 printf("%p ", c->pointers[i]);
111                         printf("%d ", c->keys[i]);
112                 }
113                 if (verbose_output)
114                         printf("%p ", c->pointers[order - 1]);
115                 if (c->pointers[order - 1] != NULL) {
116                         printf(" | ");
117                         c = c->pointers[order - 1];
118                 }
119                 else
120                         break;
121         }
122         printf("\n");
123 }
124
125 int height(node * const root) {
126         int h = 0;
127         node * c = root;
128         while (!c->is_leaf) {
129                 c = c->pointers[0];
130                 h++;
131         }
132         return h;
133 }
134
135 int path_to_root(node * const root, node * child) {
136         int length = 0;
137         node * c = child;
138         while (c != root) {
139                 c = c->parent;
140                 length++;
141         }
142         return length;
143 }
144
145 void print_tree(node * const root) {
146
147         node * n = NULL;
148         int i = 0;
149         int rank = 0;
```

```
150         int new_rank = 0;
151
152         if (root == NULL) {
153                 printf("Empty tree.\n");
154                 return;
155         }
156         queue = NULL;
157         enqueue(root);
158         while(queue != NULL) {
159                 n = dequeue();
160                 if (n->parent != NULL && n == n->parent->pointers[0]) {
161                         new_rank = path_to_root(root, n);
162                         if (new_rank != rank) {
163                                 rank = new_rank;
164                                 printf("\n");
165                         }
166                 }
167                 if (verbose_output)
168                         printf("(%p)", n);
169                 for (i = 0; i < n->num_keys; i++) {
170                         if (verbose_output)
171                                 printf("%p ", n->pointers[i]);
172                         printf("%d ", n->keys[i]);
173                 }
174                 if (!n->is_leaf)
175                         for (i = 0; i <= n->num_keys; i++)
176                                 enqueue(n->pointers[i]);
177                 if (verbose_output) {
178                         if (n->is_leaf)
179                                 printf("%p ", n->pointers[order - 1]);
180                         else
181                                 printf("%p ", n->pointers[n->num_keys]);
182                 }
183                 printf("| ");
184         }
185         printf("\n");
186 }
187
188 void find_and_print(node * const root, int key, bool verbose) {
189     node * leaf = NULL;
190         record * r = find(root, key, verbose, NULL);
191         if (r == NULL)
192                 printf("Record not found under key %d.\n", key);
193         else
194                 printf("Record at %p -- key %d, value %d.\n",
195                                 r, key, r->value);
196 }
197
```

```
198 void find_and_print_range(node * const root, int key_start, int key_end,
199                 bool verbose) {
200         int i;
201         int array_size = key_end - key_start + 1;
202         int returned_keys[array_size];
203         void * returned_pointers[array_size];
204         int num_found = find_range(root, key_start, key_end, verbose,
205                         returned_keys, returned_pointers);
206         if (!num_found)
207                 printf("None found.\n");
208         else {
209                 for (i = 0; i < num_found; i++)
210                         printf("Key: %d   Location: %p  Value: %d\n",
211                                         returned_keys[i],
212                                         returned_pointers[i],
213                                         ((record *)
214                                          returned_pointers[i])->value);
215         }
216 }
217
218 int find_range(node * const root, int key_start, int key_end, bool verbose,
219                 int returned_keys[], void * returned_pointers[]) {
220         int i, num_found;
221         num_found = 0;
222         node * n = find_leaf(root, key_start, verbose);
223         if (n == NULL) return 0;
224         for (i = 0; i < n->num_keys && n->keys[i] < key_start; i++) ;
225         if (i == n->num_keys) return 0;
226         while (n != NULL) {
227                 for (; i < n->num_keys && n->keys[i] <= key_end; i++) {
228                         returned_keys[num_found] = n->keys[i];
229                         returned_pointers[num_found] = n->pointers[i];
230                         num_found++;
231                 }
232                 n = n->pointers[order - 1];
233                 i = 0;
234         }
235         return num_found;
236 }
237
238 node * find_leaf(node * const root, int key, bool verbose) {
239         if (root == NULL) {
240                 if (verbose)
241                         printf("Empty tree.\n");
242                 return root;
243         }
244         int i = 0;
245         node * c = root;
```

```
246         while (!c->is_leaf) {
247                 if (verbose) {
248                         printf("[");
249                         for (i = 0; i < c->num_keys - 1; i++)
250                                 printf("%d ", c->keys[i]);
251                         printf("%d] ", c->keys[i]);
252                 }
253                 i = 0;
254                 while (i < c->num_keys) {
255                         if (key >= c->keys[i]) i++;
256                         else break;
257                 }
258                 if (verbose)
259                         printf("%d ->\n", i);
260                 c = (node *)c->pointers[i];
261         }
262         if (verbose) {
263                 printf("Leaf [");
264                 for (i = 0; i < c->num_keys - 1; i++)
265                         printf("%d ", c->keys[i]);
266                 printf("%d] ->\n", c->keys[i]);
267         }
268         return c;
269 }
270
271 record * find(node * root, int key, bool verbose, node ** leaf_out) {
272     if (root == NULL) {
273         if (leaf_out != NULL) {
274             *leaf_out = NULL;
275         }
276         return NULL;
277     }
278
279         int i = 0;
280     node * leaf = NULL;
281
282         leaf = find_leaf(root, key, verbose);
283
284     /* If root != NULL, leaf must have a value, even
285      * if it does not contain the desired key.
286      * (The leaf holds the range of keys that would
287      * include the desired key.)
288      */
289
290         for (i = 0; i < leaf->num_keys; i++)
291                 if (leaf->keys[i] == key) break;
292     if (leaf_out != NULL) {
293         *leaf_out = leaf;
```

```
294     }
295         if (i == leaf->num_keys)
296                 return NULL;
297         else
298                 return (record *)leaf->pointers[i];
299 }
300
301 int cut(int length) {
302         if (length % 2 == 0)
303                 return length/2;
304         else
305                 return length/2 + 1;
306 }
307
308 record * make_record(int value) {
309         record * new_record = (record *)malloc(sizeof(record));
310         if (new_record == NULL) {
311                 perror("Record creation.");
312                 exit(EXIT_FAILURE);
313         }
314         else {
315                 new_record->value = value;
316         }
317         return new_record;
318 }
319
320 node * make_node(void) {
321         node * new_node;
322         new_node = malloc(sizeof(node));
323         if (new_node == NULL) {
324                 perror("Node creation.");
325                 exit(EXIT_FAILURE);
326         }
327         new_node->keys = malloc((order - 1) * sizeof(int));
328         if (new_node->keys == NULL) {
329                 perror("New node keys array.");
330                 exit(EXIT_FAILURE);
331         }
332         new_node->pointers = malloc(order * sizeof(void *));
333         if (new_node->pointers == NULL) {
334                 perror("New node pointers array.");
335                 exit(EXIT_FAILURE);
336         }
337         new_node->is_leaf = false;
338         new_node->num_keys = 0;
339         new_node->parent = NULL;
340         new_node->next = NULL;
341         return new_node;
```

```
342 }
343
344 node * make_leaf(void) {
345         node * leaf = make_node();
346         leaf->is_leaf = true;
347         return leaf;
348 }
349
350 int get_left_index(node * parent, node * left) {
351
352         int left_index = 0;
353         while (left_index <= parent->num_keys &&
354                         parent->pointers[left_index] != left)
355                 left_index++;
356         return left_index;
357 }
358
359 node * insert_into_leaf(node * leaf, int key, record * pointer) {
360
361         int i, insertion_point;
362
363         insertion_point = 0;
364         while (insertion_point < leaf->num_keys && leaf->keys[insertion_point] <
        ↪  key)
365                 insertion_point++;
366
367         for (i = leaf->num_keys; i > insertion_point; i--) {
368                 leaf->keys[i] = leaf->keys[i - 1];
369                 leaf->pointers[i] = leaf->pointers[i - 1];
370         }
371         leaf->keys[insertion_point] = key;
372         leaf->pointers[insertion_point] = pointer;
373         leaf->num_keys++;
374         return leaf;
375 }
376
377 node * insert_into_leaf_after_splitting(node * root, node * leaf, int key, record
    ↪  * pointer) {
378
379         node * new_leaf;
380         int * temp_keys;
381         void ** temp_pointers;
382         int insertion_index, split, new_key, i, j;
383
384         new_leaf = make_leaf();
385
386         temp_keys = malloc(order * sizeof(int));
387         if (temp_keys == NULL) {
```

```
388             perror("Temporary keys array.");
389             exit(EXIT_FAILURE);
390         }
391
392         temp_pointers = malloc(order * sizeof(void *));
393         if (temp_pointers == NULL) {
394             perror("Temporary pointers array.");
395             exit(EXIT_FAILURE);
396         }
397
398         insertion_index = 0;
399         while (insertion_index < order - 1 && leaf->keys[insertion_index] < key)
400             insertion_index++;
401
402         for (i = 0, j = 0; i < leaf->num_keys; i++, j++) {
403             if (j == insertion_index) j++;
404             temp_keys[j] = leaf->keys[i];
405             temp_pointers[j] = leaf->pointers[i];
406         }
407
408         temp_keys[insertion_index] = key;
409         temp_pointers[insertion_index] = pointer;
410
411         leaf->num_keys = 0;
412
413         split = cut(order - 1);
414
415         for (i = 0; i < split; i++) {
416             leaf->pointers[i] = temp_pointers[i];
417             leaf->keys[i] = temp_keys[i];
418             leaf->num_keys++;
419         }
420
421         for (i = split, j = 0; i < order; i++, j++) {
422             new_leaf->pointers[j] = temp_pointers[i];
423             new_leaf->keys[j] = temp_keys[i];
424             new_leaf->num_keys++;
425         }
426
427         free(temp_pointers);
428         free(temp_keys);
429
430         new_leaf->pointers[order - 1] = leaf->pointers[order - 1];
431         leaf->pointers[order - 1] = new_leaf;
432
433         for (i = leaf->num_keys; i < order - 1; i++)
434             leaf->pointers[i] = NULL;
435         for (i = new_leaf->num_keys; i < order - 1; i++)
```

```
436                 new_leaf->pointers[i] = NULL;
437
438         new_leaf->parent = leaf->parent;
439         new_key = new_leaf->keys[0];
440
441         return insert_into_parent(root, leaf, new_key, new_leaf);
442 }
443
444 node * insert_into_node(node * root, node * n,
445                 int left_index, int key, node * right) {
446         int i;
447
448         for (i = n->num_keys; i > left_index; i--) {
449                 n->pointers[i + 1] = n->pointers[i];
450                 n->keys[i] = n->keys[i - 1];
451         }
452         n->pointers[left_index + 1] = right;
453         n->keys[left_index] = key;
454         n->num_keys++;
455         return root;
456 }
457
458 node * insert_into_node_after_splitting(node * root, node * old_node, int
    ↪  left_index,
459                 int key, node * right) {
460
461         int i, j, split, k_prime;
462         node * new_node, * child;
463         int * temp_keys;
464         node ** temp_pointers;
465
466         temp_pointers = malloc((order + 1) * sizeof(node *));
467         if (temp_pointers == NULL) {
468                 perror("Temporary pointers array for splitting nodes.");
469                 exit(EXIT_FAILURE);
470         }
471         temp_keys = malloc(order * sizeof(int));
472         if (temp_keys == NULL) {
473                 perror("Temporary keys array for splitting nodes.");
474                 exit(EXIT_FAILURE);
475         }
476
477         for (i = 0, j = 0; i < old_node->num_keys + 1; i++, j++) {
478                 if (j == left_index + 1) j++;
479                 temp_pointers[j] = old_node->pointers[i];
480         }
481
482         for (i = 0, j = 0; i < old_node->num_keys; i++, j++) {
```

```
483                 if (j == left_index) j++;
484                 temp_keys[j] = old_node->keys[i];
485         }
486
487         temp_pointers[left_index + 1] = right;
488         temp_keys[left_index] = key;
489
490         split = cut(order);
491         new_node = make_node();
492         old_node->num_keys = 0;
493         for (i = 0; i < split - 1; i++) {
494                 old_node->pointers[i] = temp_pointers[i];
495                 old_node->keys[i] = temp_keys[i];
496                 old_node->num_keys++;
497         }
498         old_node->pointers[i] = temp_pointers[i];
499         k_prime = temp_keys[split - 1];
500         for (++i, j = 0; i < order; i++, j++) {
501                 new_node->pointers[j] = temp_pointers[i];
502                 new_node->keys[j] = temp_keys[i];
503                 new_node->num_keys++;
504         }
505         new_node->pointers[j] = temp_pointers[i];
506         free(temp_pointers);
507         free(temp_keys);
508         new_node->parent = old_node->parent;
509         for (i = 0; i <= new_node->num_keys; i++) {
510                 child = new_node->pointers[i];
511                 child->parent = new_node;
512         }
513
514
515         return insert_into_parent(root, old_node, k_prime, new_node);
516 }
517
518 node * insert_into_parent(node * root, node * left, int key, node * right) {
519
520         int left_index;
521         node * parent;
522         parent = left->parent;
523         if (parent == NULL)
524                 return insert_into_new_root(left, key, right);
525
526         left_index = get_left_index(parent, left);
527         if (parent->num_keys < order - 1)
528                 return insert_into_node(root, parent, left_index, key, right);
529         return insert_into_node_after_splitting(root, parent, left_index, key,
          ↪  right);
```

```
530 }
531
532 node * insert_into_new_root(node * left, int key, node * right) {
533
534         node * root = make_node();
535         root->keys[0] = key;
536         root->pointers[0] = left;
537         root->pointers[1] = right;
538         root->num_keys++;
539         root->parent = NULL;
540         left->parent = root;
541         right->parent = root;
542         return root;
543 }
544
545 node * start_new_tree(int key, record * pointer) {
546
547         node * root = make_leaf();
548         root->keys[0] = key;
549         root->pointers[0] = pointer;
550         root->pointers[order - 1] = NULL;
551         root->parent = NULL;
552         root->num_keys++;
553         return root;
554 }
555
556 node * insert(node * root, int key, int value) {
557
558         record * record_pointer = NULL;
559         node * leaf = NULL;
560
561         record_pointer = find(root, key, false, NULL);
562     if (record_pointer != NULL) {
563         record_pointer->value = value;
564         return root;
565     }
566
567         record_pointer = make_record(value);
568
569         if (root == NULL)
570                 return start_new_tree(key, record_pointer);
571
572         leaf = find_leaf(root, key, false);
573         if (leaf->num_keys < order - 1) {
574                 leaf = insert_into_leaf(leaf, key, record_pointer);
575                 return root;
576         }
577         return insert_into_leaf_after_splitting(root, leaf, key, record_pointer);
```

```
578 }
579
580 int get_neighbor_index(node * n) {
581
582         int i;
583
584         for (i = 0; i <= n->parent->num_keys; i++)
585                 if (n->parent->pointers[i] == n)
586                         return i - 1;
587
588         // Error state.
589         printf("Search for nonexistent pointer to node in parent.\n");
590         printf("Node:  %#lx\n", (unsigned long)n);
591         exit(EXIT_FAILURE);
592 }
593
594
595 node * adjust_root(node * root) {
596
597         node * new_root;
598
599         if (root->num_keys > 0)
600                 return root;
601
602         if (!root->is_leaf) {
603                 new_root = root->pointers[0];
604                 new_root->parent = NULL;
605         }
606
607         else
608                 new_root = NULL;
609
610         free(root->keys);
611         free(root->pointers);
612         free(root);
613
614         return new_root;
615 }
616
617
618 node * redistribute_nodes(node * root, node * n, node * neighbor, int
    ↪  neighbor_index,
619                 int k_prime_index, int k_prime) {
620
621         int i;
622         node * tmp;
623
624         if (neighbor_index != -1) {
```

```
625                 if (!n->is_leaf)
626                         n->pointers[n->num_keys + 1] = n->pointers[n->num_keys];
627                 for (i = n->num_keys; i > 0; i--) {
628                         n->keys[i] = n->keys[i - 1];
629                         n->pointers[i] = n->pointers[i - 1];
630                 }
631                 if (!n->is_leaf) {
632                         n->pointers[0] = neighbor->pointers[neighbor->num_keys];
633                         tmp = (node *)n->pointers[0];
634                         tmp->parent = n;
635                         neighbor->pointers[neighbor->num_keys] = NULL;
636                         n->keys[0] = k_prime;
637                         n->parent->keys[k_prime_index] =
                             ↪  neighbor->keys[neighbor->num_keys - 1];
638                 }
639                 else {
640                         n->pointers[0] = neighbor->pointers[neighbor->num_keys -
                             ↪  1];
641                         neighbor->pointers[neighbor->num_keys - 1] = NULL;
642                         n->keys[0] = neighbor->keys[neighbor->num_keys - 1];
643                         n->parent->keys[k_prime_index] = n->keys[0];
644                 }
645         }
646
647         else {
648                 if (n->is_leaf) {
649                         n->keys[n->num_keys] = neighbor->keys[0];
650                         n->pointers[n->num_keys] = neighbor->pointers[0];
651                         n->parent->keys[k_prime_index] = neighbor->keys[1];
652                 }
653                 else {
654                         n->keys[n->num_keys] = k_prime;
655                         n->pointers[n->num_keys + 1] = neighbor->pointers[0];
656                         tmp = (node *)n->pointers[n->num_keys + 1];
657                         tmp->parent = n;
658                         n->parent->keys[k_prime_index] = neighbor->keys[0];
659                 }
660                 for (i = 0; i < neighbor->num_keys - 1; i++) {
661                         neighbor->keys[i] = neighbor->keys[i + 1];
662                         neighbor->pointers[i] = neighbor->pointers[i + 1];
663                 }
664                 if (!n->is_leaf)
665                         neighbor->pointers[i] = neighbor->pointers[i + 1];
666         }
667
668         n->num_keys++;
669         neighbor->num_keys--;
670
```

```
671         return root;
672 }
673
674 int main(int argc, char ** argv) {
675
676         char * input_file;
677         FILE * fp;
678         node * root;
679         int input_key, input_key_2;
680         char instruction;
681
682         root = NULL;
683         verbose_output = false;
684
685         if (argc > 1) {
686                 order = atoi(argv[1]);
687                 if (order < 3) {  // less than minimum order 3
688                         fprintf(stderr, "Invalid order: %d .\n\n", order);
689                         exit(EXIT_FAILURE);
690                 }
691         }
692
693         if (argc > 2) {
694                 input_file = argv[2];
695                 fp = fopen(input_file, "r");
696                 if (fp == NULL) {
697                         perror("Failure to open input file.");
698                         exit(EXIT_FAILURE);
699                 }
700                 while (!feof(fp)) {
701                         fscanf(fp, "%d\n", &input_key);
702                         root = insert(root, input_key, input_key);
703                 }
704                 fclose(fp);
705                 print_tree(root);
706         return EXIT_SUCCESS;
707         }
708
709         printf("> ");
710     char buffer[BUFFER_SIZE];
711     int count = 0;
712     bool line_consumed = false;
713         while (scanf("%c", &instruction) != EOF) {
714         line_consumed = false;
715                 switch (instruction) {
716                         case 'i':
717                                 fgets(buffer, BUFFER_SIZE, stdin);
718                                 line_consumed = true;
```

```
719                                 count = sscanf(buffer, "%d %d", &input_key,
                                     ↪  &input_key_2);
720                                 if (count == 1)
721                                         input_key_2 = input_key;
722                                 root = insert(root, input_key, input_key_2);
723                                 print_tree(root);
724                                 break;
725                         case 'f':
726                         case 'p':
727                                 scanf("%d", &input_key);
728                                 find_and_print(root, input_key, instruction ==
                                     ↪  'p');
729                                 break;
730                         case 'r':
731                                 scanf("%d %d", &input_key, &input_key_2);
732                                 if (input_key > input_key_2) {
733                                         int tmp = input_key_2;
734                                         input_key_2 = input_key;
735                                         input_key = tmp;
736                                 }
737                                 find_and_print_range(root, input_key, input_key_2,
                                     ↪  instruction == 'p');
738                                 break;
739                         case 'l':
740                                 print_leaves(root);
741                                 break;
742                         case 'q':
743                                 while (getchar() != (int) '\n');
744                                 return EXIT_SUCCESS;
745                                 break;
746                         case 't':
747                                 print_tree(root);
748                                 break;
749                         case 'v':
750                                 verbose_output = !verbose_output;
751                                 break;
752                 }
753         if (!line_consumed)
754            while (getchar() != (int)'\n');
755                 printf("> ");
756         }
757         printf("\n");
758
759         return EXIT_SUCCESS;
760 }
```

III. Analysis