

```

1 /**
2  * @author Aviruk Basak, CSE214047, Sem 3, Year 2
3  * @topic 1 dimensional array data structure implemented in C
4  * @date 3-8-2022
5  * @cc gcc -Wall -D ARR_TYPE=int -D TYPE_FORMAT="%d" -o dsa-int-array dsa-array.c
6  */
7
8 # include <stdio.h>
9 # include <stdlib.h>
10
11 # if !defined(ARR_TYPE) || !defined(TYPE_FORMAT)
12 #     undef ARR_TYPE
13 #     undef TYPE_FORMAT
14 #     define ARR_TYPE int
15 #     define TYPE_FORMAT "%d"
16 # endif
17
18 # define ERR_NULLPTR (200)
19 # define ERR_OUTOFBOUNDS (201)
20 # define ERR_ARRAYFULL (203)
21 # define ERR_MENUDRIVELIM (204)
22
23 # define MENUDRIVE_LIMIT ((size_t) 10000)
24
25 typedef ARR_TYPE ArrayType;
26 typedef ARR_TYPE* Array;
27
28 void arr_nullPtrCheck(char *fname, Array arr);
29 Array new_array(size_t size);
30 Array arr_resize(Array arr, size_t size, size_t new_sz);
31 size_t arr_length(Array arr, size_t size);
32 void arr_print(Array arr, size_t size);
33 void arr_traverse(Array arr, size_t size, void (*callback)(size_t index, ArrayType val));
34 size_t arr_search(Array arr, size_t size, ArrayType val);
35 size_t *arr_searchAll(Array arr, size_t size, ArrayType val, size_t *matches);
36 Array arr_insert(Array arr, size_t size, size_t index, ArrayType val);
37 Array arr_delIndex(Array arr, size_t size, size_t index);
38 Array arr_delValue(Array arr, size_t size, ArrayType val);
39 Array arr_concat(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz);
40 Array arr_merge(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz);
41 Array arr_intersect(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz);
42 void arr_free(Array* arr_ptr);
43
44 void arr_nullPtrCheck(char *fname, Array arr)
45 {
46     if (!arr) {
47         printf("array: %s: null pointer\n", fname);
48         exit(ERR_NULLPTR);
49     }
50 }
51
52 Array new_array(size_t size)
53 {
54     // calloc initialises all indices with 0, last index 0 indicates atleast 1 empty index
55     Array arr = calloc(sizeof(ArrayType), size);
56     arr_nullPtrCheck("new", arr);
57     return arr;
58 }
59
60 Array arr_resize(Array arr, size_t size, size_t new_sz)
61 {
62     size_t i;
63     arr_nullPtrCheck("resize", arr);
64     // calloc initialises all indices with 0, last index 0 indicates atleast 1 empty index
65     Array arr2 = calloc(sizeof(ArrayType), new_sz);
66     arr_nullPtrCheck("resize", arr2);

```

```

67     for (i = 0; i < new_sz; i++) {
68         if (i >= size)
69             break;
70         arr2[i] = arr[i];
71     }
72     arr_free(&arr);
73     return arr2;
74 }
75
76 size_t arr_length(Array arr, size_t size)
77 {
78     size_t i;
79     for (i = size - 1; i >= 0; i--) {
80         if (arr[i] != 0) {
81             return i + 1;
82         }
83     }
84     return 0;
85 }
86
87 void arr_print(Array arr, size_t size)
88 {
89     size_t i, len;
90     arr_nullPtrCheck("print", arr);
91     len = arr_length(arr, size);
92     printf("{ ");
93     for (i = 0; i < len; i++) {
94         printf(TYPE_FORMAT "%s", arr[i], i == len - 1 ? " " : ", ");
95     }
96     printf("}\n");
97 }
98
99 void arr_traverse(Array arr, size_t size, void (*callback)(size_t index, ArrayType val))
100 {
101     size_t i;
102     arr_nullPtrCheck("traverse", arr);
103     for (i = 0; i < size; i++) {
104         callback(i, arr[i]);
105     }
106 }
107
108 size_t arr_search(Array arr, size_t size, ArrayType val)
109 {
110     size_t i, len;
111     len = arr_length(arr, size);
112     for (i = 0; i < len; i++) {
113         if (arr[i] == val)
114             return i;
115     }
116     return size;
117 }
118
119 size_t *arr_searchAll(Array arr, size_t size, ArrayType val, size_t *matches)
120 {
121     size_t *indices, i, len;
122     arr_nullPtrCheck("search all", arr);
123     *matches = 0;
124     indices = NULL;
125     len = arr_length(arr, size);
126     for (i = 0; i < len; i++) {
127         if (arr[i] == val) {
128             indices = realloc(indices, ++(*matches) * sizeof(size_t));
129             if (!indices) {
130                 printf("array: search all: null pointer\n");
131                 exit(ERR_NULLPTR);
132             }

```

```

133         indices[*matches - 1] = i;
134     }
135 }
136 return indices;
137 }
138
139 Array arr_insert(Array arr, size_t size, size_t index, ArrayType val)
140 {
141     size_t i;
142     arr_nullPtrCheck("insert", arr);
143     if (index >= size) {
144         printf("array: insert: index out of bounds\n");
145         exit(ERR_OUTOFBOUNDS);
146     } else if (arr[size - 1] != 0) {
147         printf("array: insert: array is full\n");
148         exit(ERR_ARRAYFULL);
149     }
150     for (i = size - 1; i > index; i--) {
151         arr[i] = arr[i - 1];
152     }
153     arr[index] = val;
154     return arr;
155 }
156
157 Array arr_delIndex(Array arr, size_t size, size_t index)
158 {
159     size_t i;
160     arr_nullPtrCheck("delete index", arr);
161     if (index >= size) {
162         printf("array: delete index: index out of bounds\n");
163         exit(ERR_OUTOFBOUNDS);
164     }
165     for (i = index; i < size - 1; i++) {
166         arr[i] = arr[i + 1];
167     }
168     arr[i] = 0;
169     return arr;
170 }
171
172 Array arr_delValue(Array arr, size_t size, ArrayType val)
173 {
174     size_t *indices = NULL, matches = 0, i;
175     arr_nullPtrCheck("delete value", arr);
176     indices = arr_searchAll(arr, size, val, &matches);
177     if (!indices) {
178         printf("array: delete value: null pointer\n");
179         exit(ERR_NULLPTR);
180     }
181     for (i = 0; i < matches; i++) {
182         arr_delIndex(arr, size, indices[i]);
183     }
184     free(indices);
185     return arr;
186 }
187
188 Array arr_concat(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz)
189 {
190     size_t i, j, k, len1, len2;
191     arr_nullPtrCheck("concat", arr1);
192     arr_nullPtrCheck("concat", arr2);
193     len1 = arr_length(arr1, sz1);
194     len2 = arr_length(arr2, sz2);
195     *new_sz = len1 + len2;
196     Array arr3 = new_array(*new_sz);
197     for (i = k = 0; i < len1 && k < *new_sz; i++, k++) {
198         arr3[k] = arr1[i];

```

```

199     }
200     for (j = 0; j < len2 && k < *new_sz; j++, k++) {
201         arr3[k] = arr2[j];
202     }
203     return arr3;
204 }
205
206 Array arr_merge(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz)
207 {
208     size_t i, j, k, len1, len2;
209     arr_nullPtrCheck("merge", arr1);
210     arr_nullPtrCheck("merge", arr2);
211     len1 = arr_length(arr1, sz1);
212     len2 = arr_length(arr2, sz2);
213     *new_sz = len1 + len2;
214     Array arr3 = new_array(*new_sz);
215     i = j = k = 0;
216     while (i < len1 && j < len2) {
217         if (arr1[i] < arr2[j]) {
218             arr3[k] = arr1[i];
219             i++;
220             k++;
221         } else if (arr2[j] < arr1[i]) {
222             arr3[k] = arr2[j];
223             j++;
224             k++;
225         } else {
226             arr3[k] = arr1[i] = arr2[j];
227             i++;
228             j++;
229             k++;
230         }
231     }
232     while (i < len1) {
233         arr3[k] = arr1[i];
234         i++;
235         k++;
236     }
237     while (j < len2) {
238         arr3[k] = arr2[j];
239         j++;
240         k++;
241     }
242     return arr3;
243 }
244
245 Array arr_intersect(Array arr1, size_t sz1, Array arr2, size_t sz2, size_t *new_sz)
246 {
247     size_t i, j, k, len1, len2;
248     arr_nullPtrCheck("intersect", arr1);
249     arr_nullPtrCheck("intersect", arr2);
250     len1 = arr_length(arr1, sz1);
251     len2 = arr_length(arr2, sz2);
252     *new_sz = len1 + len2;
253     Array arr3 = new_array(*new_sz);
254     i = j = k = 0;
255     while (i < len1 && j < len2) {
256         if (arr1[i] < arr2[j]) {
257             i++;
258         } else if (arr2[j] < arr1[i]) {
259             j++;
260         } else {
261             arr3[k] = arr1[i] = arr2[j];
262             i++;
263             j++;
264             k++;

```

```

265     }
266 }
267 return arr3;
268 }
269
270 void arr_free(Array* arr_ptr)
271 {
272     if (arr_ptr && *arr_ptr) {
273         free(*arr_ptr);
274         *arr_ptr = NULL;
275     }
276 }
277
278 int main()
279 {
280     int choice;
281     size_t i, size, len, menudrive_iterations = 0;
282     printf("enter array max size: ");
283     scanf("%zu", &size);
284     printf("enter no of elements to store: ");
285     scanf("%zu", &len);
286     Array arr = new_array(size);
287     printf("enter %zu elements = ", len);
288     for (i = 0; i < len; i++) {
289         scanf(TYPE_FORMAT, &arr[i]);
290     }
291     do {
292         printf(
293             "\nchoices:\n"
294             "  0: exit\n"
295             "  1: print array\n"
296             "  2: search for matching value\n"
297             "  3: search for every matching value\n"
298             "  4: insert a value\n"
299             "  5: delete value at index\n"
300             "  6: delete every match of a value\n"
301             "  7: concatenate two arrays\n"
302             "  8: merge two sorted arrays\n"
303             "  9: intersection of two sorted arrays\n"
304             "enter your choice: "
305         );
306         scanf("%d", &choice);
307         printf("\n");
308         switch (choice) {
309             // exit
310             case 0: {
311                 break;
312             }
313             // print
314             case 1: {
315                 printf("arr = ");
316                 arr_print(arr, size);
317                 printf("len = %zu\n", arr_length(arr, size));
318                 printf("size = %zu\n", size);
319                 break;
320             }
321             // search
322             case 2: {
323                 ArrayType val;
324                 size_t index;
325                 printf("enter value to be searched: ");
326                 scanf(TYPE_FORMAT, &val);
327                 index = arr_search(arr, size, val);
328                 if (index == size) {
329                     printf("value not found\n");
330                 } else {

```

```

331         printf("value '" TYPE_FORMAT "' found at index = %zu\n", val, index);
332     }
333     break;
334 }
335 // searchAll
336 case 3: {
337     ArrayType val;
338     size_t i, *indices, matches;
339     printf("enter value to be searched: ");
340     scanf(TYPE_FORMAT, &val);
341     indices = arr_searchAll(arr, size, val, &matches);
342     if (matches == 0) {
343         printf("value not found\n");
344     } else {
345         printf("value '" TYPE_FORMAT "' found at indices: ", val);
346         for (i = 0; i < matches; i++) {
347             printf("%zu%s", indices[i], i == matches - 1 ? "" : ", ");
348         }
349         printf("\n");
350     }
351     free(indices);
352     break;
353 }
354 // insert
355 case 4: {
356     ArrayType val;
357     size_t index;
358     printf("enter index of insertion: ");
359     scanf("%zu", &index);
360     printf("enter value to be inserted: ");
361     scanf(TYPE_FORMAT, &val);
362     arr = arr_insert(arr, size, index, val);
363     printf("arr = ");
364     arr_print(arr, size);
365     break;
366 }
367 // delIndex
368 case 5: {
369     size_t index;
370     printf("enter index of deletion: ");
371     scanf("%zu", &index);
372     arr = arr_delIndex(arr, size, index);
373     printf("arr = ");
374     arr_print(arr, size);
375     break;
376 }
377 // delValue
378 case 6: {
379     ArrayType val;
380     printf("enter value to delete: ");
381     scanf(TYPE_FORMAT, &val);
382     arr = arr_delValue(arr, size, val);
383     printf("arr = ");
384     arr_print(arr, size);
385     break;
386 }
387 // concat
388 case 7: {
389     size_t size2, len2, new_sz;
390     printf("enter second array max size: ");
391     scanf("%zu", &size2);
392     printf("enter no of elements to store: ");
393     scanf("%zu", &len2);
394     Array arr2 = new_array(size2);
395     printf("enter %zu elements = ", len2);
396     for (i = 0; i < len2; i++) {

```

```

397         scanf(TYPE_FORMAT, &arr2[i]);
398     }
399     Array arr3 = arr_concat(arr, size, arr2, size2, &new_sz);
400     printf("new arr = ");
401     arr_print(arr3, new_sz);
402     arr_free(&arr3);
403     break;
404 }
405 // merge
406 case 8: {
407     size_t size2, len2, new_sz;
408     printf("enter second array max size: ");
409     scanf("%zu", &size2);
410     printf("enter no of elements to store: ");
411     scanf("%zu", &len2);
412     Array arr2 = new_array(size2);
413     printf("enter sorted %zu elements = ", len2);
414     for (i = 0; i < len2; i++) {
415         scanf(TYPE_FORMAT, &arr2[i]);
416     }
417     Array arr3 = arr_merge(arr, size, arr2, size2, &new_sz);
418     printf("new arr = ");
419     arr_print(arr3, new_sz);
420     arr_free(&arr3);
421     break;
422 }
423 // intersect
424 case 9: {
425     size_t size2, len2, new_sz;
426     printf("enter second array max size: ");
427     scanf("%zu", &size2);
428     printf("enter no of elements to store: ");
429     scanf("%zu", &len2);
430     Array arr2 = new_array(size2);
431     printf("enter sorted %zu elements = ", len2);
432     for (i = 0; i < len2; i++) {
433         scanf(TYPE_FORMAT, &arr2[i]);
434     }
435     Array arr3 = arr_intersect(arr, size, arr2, size2, &new_sz);
436     printf("new arr = ");
437     arr_print(arr3, new_sz);
438     arr_free(&arr3);
439     break;
440 }
441 default: {
442     printf("choice invalid\n");
443 }
444 }
445 menudrive_iterations++;
446 } while (choice && menudrive_iterations < MENU_DRIVE_LIMIT);
447 // MENU_DRIVE_LIMIT to avoid an accidental infinite loop due to scanf I/O error
448 if (menudrive_iterations >= MENU_DRIVE_LIMIT) {
449     printf("exceeded menu drive limit of '%zu' iterations\n", MENU_DRIVE_LIMIT);
450     exit(ERR_MENU_DRIVE_LIMIT);
451 }
452 arr_free(&arr);
453 return 0;
454 }

```