Heap

Mariano Giménez - 99789 - Franco Roberti - 97463

Corrector: Martin Buchwald

1. Código fuente:

1.1. Heap.c

```c
#include "heap.h"
#include <stdlib.h>
#include <stdbool.h>
#include <string.h>

#define TAM_INICIAL 20
//redim
#define FACTOR_DE_CARGA_MAX 0.7
#define FACTOR_DE_CARGA_MIN 0.15
#define INCREMENTO_CAPACIDAD 2
#define DECREMENTO_CAPACIDAD 2

bool heap_redimensionar(heap_t *heap);
void downheap(void *elementos[],size_t cant, size_t posicion, cmp_func_t cmp);
void upheap(void *elementos[], size_t posicion, cmp_func_t cmp);
void swap(void* elementos[], size_t pos1, size_t pos2);
void _heap_sort(void* elementos[],size_t cant,cmp_func_t cmp);

struct heap{
        void** arreglo;
        size_t cantidad;
        size_t capacidad;
        cmp_func_t cmp;

};

heap_t *heap_crear(cmp_func_t cmp){

        heap_t *heap = malloc(sizeof(heap_t));

        if(!heap)
                return NULL;
        heap -> arreglo = malloc(sizeof(void*)*TAM_INICIAL);

        if(!heap->arreglo){
                free(heap);
                return NULL;
        }

        heap->cantidad = 0;
        heap->capacidad = TAM_INICIAL;
        heap->cmp = cmp;

        return heap;
```

```
45  }
46
47  heap_t *heap_crear_arr(void *arreglo[], size_t n, cmp_func_t cmp){
48
49          heap_t *heap = heap_crear(cmp);
50
51          for(size_t i=0;i<n;i++){
52                  heap_encolar(heap, arreglo[i]);
53          }
54
55          return heap;
56
57  }
58  size_t heap_cantidad(const heap_t *heap){
59          return heap->cantidad;
60  }
61
62  bool heap_esta_vacio(const heap_t *heap){
63          return heap->cantidad == 0;
64  }
65
66  void *heap_ver_max(const heap_t *heap){
67          if(!heap->cantidad || !heap)
68                  return NULL;
69
70          return (void*)heap->arreglo[0];
71  }
72
73  bool heap_redimensionar(heap_t *heap){
74          double fc = (double)(heap->cantidad)/((double)heap->capacidad);
75
76          if((fc>=FACTOR_DE_CARGA_MAX)||((fc<=FACTOR_DE_CARGA_MIN) &&
77      (heap->capacidad > TAM_INICIAL))){
78
79                  size_t tam;
80                  if(fc>=FACTOR_DE_CARGA_MAX) tam = heap->capacidad*INCREMENTO_CAPACIDAD
81
82                  else tam = heap->capacidad/DECREMENTO_CAPACIDAD;
83
84                  void* aux_array = realloc(heap->arreglo,sizeof(void*)*tam);
85                  if(!aux_array) return false;
86                  heap->arreglo = aux_array;
87                  heap->capacidad = tam;
88          }
89
90          return true;
91  }
92
93  bool heap_encolar(heap_t *heap,void *elemento){
94
95          if(!heap || !elemento)
96                  return false;
97
98          heap->arreglo[heap->cantidad] = elemento;
99          upheap(heap->arreglo, heap->cantidad, heap->cmp);
100         heap->cantidad ++;
101
102         if(!heap_redimensionar(heap)) return false;
103
104
105         return true;
```

```
106 }
107
108 void heap_destruir(heap_t *heap, void destruir_elemento(void *e)){
109
110         if(!heap)
111                 return;
112
113         if(heap->cantidad){
114                 if(destruir_elemento){
115                         for(size_t i=0;i<heap->cantidad;i++){
116                                 destruir_elemento(heap->arreglo[i]);
117                         }
118                 }
119         }
120         free(heap->arreglo);
121         free(heap);
122
123 }
124
125 void *heap_desencolar(heap_t* heap){
126
127         void *aux;
128
129         if(!heap || !heap->cantidad)
130                 return NULL;
131
132         aux = heap->arreglo[0];
133         heap->arreglo[0] = heap->arreglo[heap->cantidad-1];
134
135         downheap(heap->arreglo,heap->cantidad,0,heap->cmp);
136
137         heap->cantidad --;
138         if(!heap_redimensionar(heap)) return false;
139         return aux;
140 }
141
142 void heap_sort(void *elementos[], size_t cant, cmp_func_t cmp){
143         for(int i = (int)(cant-1)/2;i>=0;i--){
144                 downheap(elementos,cant,i,cmp);
145         }
146
147         for(int i = (int)cant-1;i>0;i--){
148                 swap(elementos,0,i);
149                 downheap(elementos,i,0,cmp);
150         }
151
152 }
153
154
155
156 void downheap(void *elementos[],size_t cant, size_t posicion, cmp_func_t cmp){
157         size_t pos_h_izq, pos_h_der, pos_mayor;
158
159         if(posicion >= cant){
160                 return;
161         }
162
163         pos_h_izq = posicion*2 + 1;
164         pos_h_der = posicion*2 + 2;
165         pos_mayor = posicion;
166
```

```
167         if(pos_h_izq < cant && cmp(elementos[posicion],elementos[pos_h_izq]) <0)
168                 pos_mayor = pos_h_izq;
169
170         if(pos_h_der < cant && cmp(elementos[pos_mayor],elementos[pos_h_der]) <0)
171                 pos_mayor = pos_h_der;
172
173         if(pos_mayor != posicion){
174                 swap(elementos, posicion, pos_mayor);
175                 downheap(elementos,cant,pos_mayor,cmp);
176         }
177
178 }
179
180
181 void upheap(void *elementos[], size_t posicion, cmp_func_t cmp){
182         size_t pos_padre;
183
184         if(!posicion)
185                 return;
186
187         pos_padre = (posicion-1)/2;
188
189         if(cmp(elementos[posicion],elementos[pos_padre])>0){
190                 swap(elementos,posicion,pos_padre);
191                 upheap(elementos,pos_padre,cmp);
192         }
193 }
194
195 void swap(void* elementos[], size_t pos1, size_t pos2){
196         void* aux = elementos[pos1];
197         elementos[pos1] = elementos[pos2];
198         elementos[pos2] = aux;
199 }
```

1.2. pruebas_alumno.c

```c
#include "heap.h"
#include "testing.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>  // For ssize_t in Linux.


int comparar_cadenas(const void* a, const void*b){
    return strcmp((char*)a,(char*)b);
}

int comparar_enteros(const void* a, const void*b){
        if(*(int*)a > *(int*)b)
                return 1;
        if(*(int*)a < *(int*)b)
                return -1;

        return 0;
}

static void prueba_crear_heap_vacio()
{
  heap_t* heap = heap_crear(comparar_cadenas);

  print_test("Prueba crear heap vacio", heap);
  print_test("Prueba heap la cantidad de elementos es 0", heap_cantidad(heap) == 0);
  print_test("Prueba heap esta vacio", heap_esta_vacio(heap));
  print_test("Prueba heap ver maximo es NULL", !heap_ver_max(heap));
  print_test("Prueba heap desencolar es NULL", !heap_desencolar(heap));

  heap_destruir(heap,NULL);
}

static void prueba_heap_encolar_cadenas(){
  heap_t* heap = heap_crear(comparar_cadenas);

  char *clave4 = "cabra";
  char *clave2 = "gato";
  char* clave7 = "jirafa";
  char *clave5 = "mono";
  char *clave1 = "perro";
  char *clave3 = "vaca";
  char *clave6 = "zorro";

  print_test("Prueba heap encolar cadenas clave1", heap_encolar(heap, clave1));
  print_test("Prueba heap encolar cadenas clave2", heap_encolar(heap, clave2));
  print_test("Prueba heap ver max es clave2", heap_ver_max(heap)==clave1);
  print_test("Prueba heap encolar cadenas clave3", heap_encolar(heap, clave3));
  print_test("Prueba heap encolar cadenas clave4", heap_encolar(heap, clave4));
  print_test("Prueba heap la cantidad de elementos es 4", heap_cantidad(heap) == 4);
  print_test("Prueba heap ver max es clave3", heap_ver_max(heap)==clave3);
  print_test("Prueba heap encolar cadenas clave5", heap_encolar(heap, clave5));
  print_test("Prueba heap encolar cadenas clave6", heap_encolar(heap, clave6));
  print_test("Prueba heap encolar cadenas clave7", heap_encolar(heap, clave7));
  print_test("Prueba heap la cantidad de elementos es 7", heap_cantidad(heap) == 7);
  print_test("Prueba heap ver max es clave6", heap_ver_max(heap)==clave6);

```

```
59    heap_destruir(heap,NULL);
60  }
61
62  static void prueba_heap_desencolar_cadenas(){
63    heap_t* heap = heap_crear(comparar_cadenas);
64
65    char *clave4 = "cabra";
66    char *clave2 = "gato";
67    char* clave7 = "jirafa";
68    char *clave5 = "mono";
69    char *clave1 = "perro";
70    char *clave3 = "vaca";
71    char *clave6 = "zorro";
72
73    print_test("Prueba heap encolar cadenas clave1", heap_encolar(heap, clave1));
74    print_test("Prueba heap encolar cadenas clave2", heap_encolar(heap, clave2));
75    print_test("Prueba heap la cantidad de elementos es 2",
76      heap_cantidad(heap) == 2);
77    print_test("Prueba heap desencolar cadenas es clave1",
78      heap_desencolar(heap)==clave1);
79    print_test("Prueba heap desencolar cadenas es clave2",
80      heap_desencolar(heap)==clave2);
81    print_test("Prueba heap desencolar cadenas es NULL",
82     heap_desencolar(heap)==NULL);
83    print_test("Prueba heap la cantidad de elementos es 0",
84      heap_cantidad(heap) == 0);
85    print_test("Prueba heap encolar cadenas clave3", heap_encolar(heap, clave3));
86    print_test("Prueba heap encolar cadenas clave4", heap_encolar(heap, clave4));
87    print_test("Prueba heap ver max es es clave3", heap_ver_max(heap)==clave3);
88    print_test("Prueba heap encolar cadenas clave5", heap_encolar(heap, clave5));
89    print_test("Prueba heap encolar cadenas clave6", heap_encolar(heap, clave6));
90    print_test("Prueba heap desencolar cadenas es clave6",
91      heap_desencolar(heap)==clave6);
92    print_test("Prueba heap encolar clave7", heap_encolar(heap, clave7));
93    print_test("Prueba heap desencolar es clave6", heap_ver_max(heap)==clave3);
94
95    heap_destruir(heap,NULL);
96  }
97
98  static void prueba_heap_encolar_enteros(){
99    heap_t* heap = heap_crear(comparar_enteros);
100
101   int clave4 = 6;
102   int clave2 = 30;
103   int clave7 = 555;
104   int clave5 = 3244;
105   int clave1 = 3245;
106   int clave3 = 45213;
107   int clave6 = 543254;
108
109   print_test("Prueba heap encolar enteros clave1", heap_encolar(heap, &clave1));
110   print_test("Prueba heap encolar enteros clave2", heap_encolar(heap, &clave2));
111   print_test("Prueba heap ver max es clave2", heap_ver_max(heap)==&clave1);
112   print_test("Prueba heap encolar enteros clave3", heap_encolar(heap, &clave3));
113   print_test("Prueba heap encolar enteros clave4", heap_encolar(heap, &clave4));
114   print_test("Prueba heap ver max es clave3", heap_ver_max(heap)==&clave3);
115   print_test("Prueba heap encolar enteros clave5", heap_encolar(heap, &clave5));
116   print_test("Prueba heap encolar enteros clave6", heap_encolar(heap, &clave6));
117   print_test("Prueba heap encolar enteros clave7", heap_encolar(heap, &clave7));
118   print_test("Prueba heap ver max es clave6", heap_ver_max(heap)==&clave6);
119
```

```
120     heap_destruir(heap,NULL);
121   }
122
123   static void prueba_heap_desencolar_enteros(){
124     heap_t* heap = heap_crear(comparar_enteros);
125
126     int clave4 = 6;
127     int clave2 = 30;
128     int clave7 = 555;
129     int clave5 = 3244;
130     int clave1 = 3245;
131     int clave3 = 45213;
132     int clave6 = 543254;
133
134     print_test("Prueba heap encolar enteros clave1", heap_encolar(heap, &clave1));
135     print_test("Prueba heap encolar enteros clave2", heap_encolar(heap, &clave2));
136     print_test("Prueba heap desencolar enteros es clave1",
137       heap_desencolar(heap)==&clave1);
138     print_test("Prueba heap desencolar enteros es clave2",
139       heap_desencolar(heap)==&clave2);
140     print_test("Prueba heap desencolar enteros es NULL", heap_desencolar(heap)==NULL);
141     print_test("Prueba heap encolar enteros clave3", heap_encolar(heap, &clave3));
142     print_test("Prueba heap encolar enteros clave4", heap_encolar(heap, &clave4));
143     print_test("Prueba heap desencolar enteros es clave3",
144       heap_ver_max(heap)==&clave3);
145     print_test("Prueba heap encolar enteros clave5", heap_encolar(heap, &clave5));
146     print_test("Prueba heap encolar enteros clave6", heap_encolar(heap, &clave6));
147     print_test("Prueba heap desencolar enteros es clave6",
148       heap_desencolar(heap)==&clave6);
149     print_test("Prueba heap encolar enteros clave7", heap_encolar(heap, &clave7));
150     print_test("Prueba heap desencolar enteros es clave6",
151       heap_ver_max(heap)==&clave3);
152
153     heap_destruir(heap,NULL);
154   }
155
156   static void prueba_heap_free_propio(){
157     heap_t* heap = heap_crear(comparar_enteros);
158
159           int *aux,i;
160
161           for(i=0;i<100;i++){
162                   aux = malloc(sizeof(int));
163                   *aux = i;
164                   heap_encolar(heap,aux);
165           }
166
167     print_test("Prueba heap free propio", *(int*)(heap_ver_max(heap))== 99);
168     heap_destruir(heap,free);
169   }
170   //Encola en volumen,
171   static void prueba_heap_volumen(int tam){
172     heap_t* heap = heap_crear(comparar_enteros);
173
174           int i,*aux;
175
176           for(i=0;i<tam;i++){
177                   aux = malloc(sizeof(int));
178                   *aux = i;
179                   heap_encolar(heap,aux);
180           }
```

```
181
182     print_test("Prueba heap de volumen", *(int*)(heap_ver_max(heap))== tam-1);
183
184     heap_destruir(heap,free);
185   }
186
187   //Se pasa un arreglo con claves ordenadas de menor a mayor, y se espera que
188   //las encole a un heap, para luego ir obteniendo de mayor a menor.
189   static void prueba_heapify(size_t tam){
190           void** arreglo = malloc(sizeof(int*)*tam);
191
192           for(int i=0;i<(int)tam;i++){
193                   int* aux = malloc(sizeof(int));
194                   *aux = i;
195                   arreglo[i]=aux;
196           }
197
198           heap_t* heap = heap_crear_arr(arreglo,tam,comparar_enteros);
199           print_test("Prueba heap_crear_arr devuelve heap ", heap);
200           print_test("Prueba heap_crear_arr, el heap no esta vacio",
201                   !heap_esta_vacio(heap));
202           print_test("Prueba heap_crear_arr, ver_max",
203       *(int*)heap_ver_max(heap)==*(int*)arreglo[tam-1]);
204
205           bool ok;
206           for(int i =(int)tam-1;i>=0;i--){
207                   ok = (heap_desencolar(heap) == arreglo[i]);
208
209           }
210           print_test("Prueba orden de desencolado en heapify",ok);
211
212
213
214           for(int i = 0;i<(int)tam;i++){
215                   free(arreglo[i]);
216           }
217           free(arreglo);
218
219           heap_destruir(heap,free);
220
221   }
222
223
224   //Se pasa un arreglo con claves ordenadas de menor a mayor, y se espera que
225   //las encole a un heap, para luego ir obteniendo de mayor a menor.
226   static void prueba_heap_sort(size_t tam){
227           void** arreglo = malloc(sizeof(int*)*tam);
228
229           for(int i=0;i<(int)tam;i++){
230                   int* aux = malloc(sizeof(int));
231                   *aux = i;
232                   arreglo[i]=aux;
233           }
234
235           heap_sort(arreglo,tam,comparar_enteros);
236
237           bool ok=true;
238           for(int i = 0;i<=(int)tam-2;i++){
239                   if(comparar_enteros(arreglo[i],arreglo[i+1])>0){
240                           ok = false;
241                           break;
```

```
242                 }
243         }
244
245         print_test("Prueba orden de arreglo prueba_heap_sort",ok);
246
247
248         for(int i = 0;i<(int)tam;i++){
249                 free(arreglo[i]);
250         }
251         free(arreglo);
252
253
254 }
255
256
257
258 void prueba_heap_sort_2(){
259         size_t tam = 3;
260         void** arreglo = malloc(sizeof(int*)*tam);
261
262         int* a = malloc(sizeof(int));
263         int* b = malloc(sizeof(int));
264         int* c = malloc(sizeof(int));
265
266         *a = 20;
267         *b = 300;
268         *c = 4000;
269
270         arreglo[1]=a;
271         arreglo[2]=b;
272         arreglo[0]=c;
273
274         heap_sort(arreglo,tam,comparar_enteros);
275
276         print_test("Prueba heap_sort, ver pos 0 es a", arreglo[0]==a);
277         print_test("Prueba heap_sort, ver pos 1 es b", arreglo[1]==b);
278         print_test("Prueba heap_sort, ver pos 2 es c", arreglo[2]==c);
279
280         heap_sort(arreglo,tam,comparar_enteros);
281
282         bool ok=true;
283         if(arreglo[0]!=a || arreglo[1] != b || arreglo[2]!=c) ok = false;
284         print_test("Prueba heap_sort en un arreglo ordenado",ok);
285
286
287         for(int i = 0;i<(int)tam;i++){
288                 free(arreglo[i]);
289         }
290         free(arreglo);
291
292
293 }
294 void prueba_heap_encolar_memoria_dinamica(){
295         heap_t* heap = heap_crear(comparar_enteros);
296
297
298         int* a = malloc(sizeof(int));
299         int* b = malloc(sizeof(int));
300         int* c = malloc(sizeof(int));
301
302         *a = 1;
```

```
303         *b = 2;
304         *c = 3;
305 
306         print_test("Prueba heap encolar elemento a",heap_encolar(heap,a));
307         print_test("Prueba heap encolar elemento b",heap_encolar(heap,b));
308         print_test("Prueba heap encolar elemento c",heap_encolar(heap,c));
309         print_test("Ver maximo es c",heap_ver_max(heap)==c);
310 
311         heap_destruir(heap,free);
312 
313 }
314 void pruebas_heap_alumno()
315 {
316     /* Ejecuta todas las pruebas unitarias. */
317     prueba_crear_heap_vacio();
318     prueba_heap_encolar_cadenas();
319     prueba_heap_desencolar_cadenas();
320     prueba_heap_encolar_enteros();
321     prueba_heap_desencolar_enteros();
322     prueba_heap_free_propio();
323     prueba_heap_volumen(5000);
324     prueba_heapify(5000);
325     prueba_heap_sort(100);
326     prueba_heap_sort_2(1000);
327         prueba_heap_encolar_memoria_dinamica();
328 
329     if(!failure_count())
330         printf("Se finalizaron todas las pruebas sin errores\n");
331 }
```