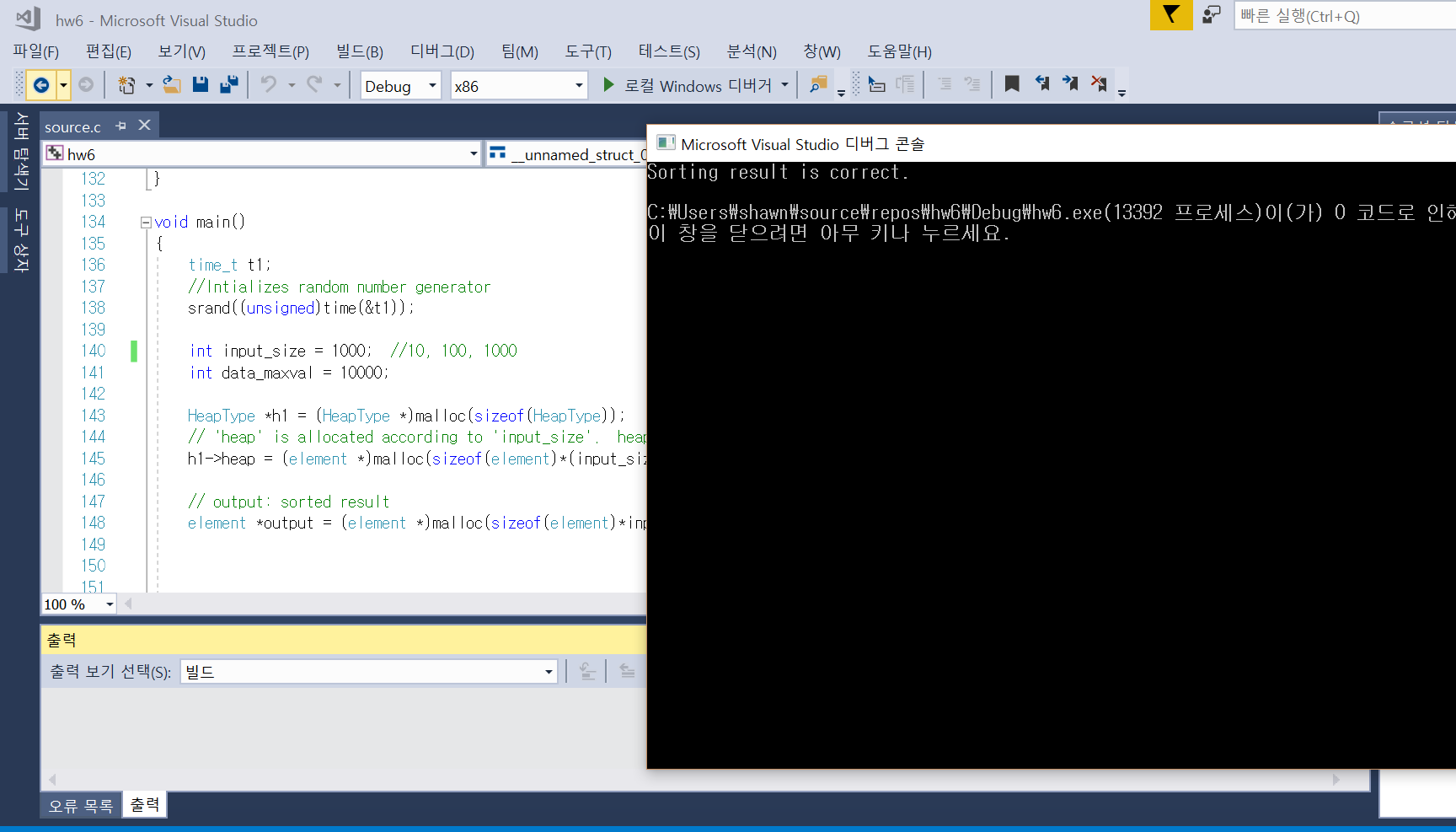
1. 코드 분석 및 주석
2. // heapsort.cpp : Defines the entry point for the console application.
3. //
4. #include "stdlib.h"
5. #include "stdio.h"
6. #include "string.h"
7. #include "time.h"
8. #define MAX\_ELEMENT 2000
9. typedef struct element {
10. int key;
11. } element;
12. typedef struct {
13. element \*heap;
14. int heap\_size;
15. } HeapType;
16. // Integer random number generation function between 0 and n-1
17. int random(int n)
18. {
19. return rand() % n;
20. }
21. // Initialization
22. void init(HeapType \*h) {
23. h->heap\_size = 0;
24. }
25. // Insert the item at heap h, (# of elements: heap\_size)
26. void insert\_max\_heap(HeapType \*h, element item)
27. {
28. int i;
29. i = ++(h->heap\_size);
30. // The process of comparing with the parent node as it traverses the tree
31. while ((i != 1) && (item.key > h->heap[i / 2].key)) {
32. h->heap[i] = h->heap[i / 2];
33. i /= 2;
34. }
35. h->heap[i] = item; // Insert new node
36. }
37. // Delete the root at heap h, (# of elements: heap\_size)
38. element delete\_max\_heap(HeapType \*h)
39. {
40. int parent, child;
41. element item, temp;
42. item = h->heap[1];
43. temp = h->heap[(h->heap\_size)--];
44. parent = 1;
45. child = 2;
46. while (child <= h->heap\_size) {
47. // Find a smaller child node
48. if ((child < h->heap\_size) &&
49. (h->heap[child].key) < h->heap[child + 1].key)
50. child++;
51. if (temp.key >= h->heap[child].key) break;
52. // Move down one level
53. h->heap[parent] = h->heap[child];
54. parent = child;
55. child \*= 2;
56. }
57. h->heap[parent] = temp;
58. return item;
59. }
60. void build\_max\_heap(HeapType \*h)
61. {
62. int length = h->heap\_size;
63. int i = length / 2;
64. int parent;
65. int child;
66. int temp;
67. while (i!=0) {
68. parent = i;
70. if (parent \* 2 == h->heap\_size)
71. child = parent \* 2;
72. else {
73. if (h->heap[parent \* 2].key > h->heap[parent \* 2 + 1].key)
74. child = parent \* 2;
75. else child = parent \* 2 + 1;
76. }
77. while (child<=h->heap\_size&&(h->heap[parent].key<h->heap[child].key)) {
79. temp = h->heap[child].key;
80. h->heap[child].key = h->heap[parent].key;
81. h->heap[parent].key = temp;
82. parent = child;
83. child = child \* 2;
84. if (child > h->heap\_size)
85. break;
86. else{
87. if (h->heap[child + 1].key > h->heap[child].key)
88. child = child + 1;
89. }
90. }
91. i--;
92. }
93. }
94. //input: heap 'h'
95. //output: sorted element array 'a'
96. void heap\_sort(HeapType \*h, element \*a, int n)
97. {
98. int i;
99. build\_max\_heap(h);
100. for (i = (n - 1); i >= 0; i--) {
101. a[i] = delete\_max\_heap(h);
102. }
103. }
104. int check\_sort\_results(element \*output, int n)
105. {
106. int index = 1;
107. for (int i = 0; i < n - 1; i++)
108. if (output[i].key > output[i + 1].key)
109. {
110. index = 0;
111. break;
112. }
113. return index;
114. }
115. void main()
116. {
117. time\_t t1;
118. //Intializes random number generator
119. srand((unsigned)time(&t1));
120. int input\_size = 1000; //10, 100, 1000
121. int data\_maxval = 10000;
122. HeapType \*h1 = (HeapType \*)malloc(sizeof(HeapType));
123. // 'heap' is allocated according to 'input\_size'. heap starts with 1, so 'input\_size+1' is used.
124. h1->heap = (element \*)malloc(sizeof(element)\*(input\_size + 1));
125. // output: sorted result
126. element \*output = (element \*)malloc(sizeof(element)\*input\_size);
127. // Generate an input data randomly
128. for (int i = 0; i < input\_size; i++)
129. h1->heap[i + 1].key = random(data\_maxval); // note) heap starts with 1.
130. h1->heap\_size = input\_size;
131. if (input\_size < 20) {
132. printf("Input data\n");
133. for (int i = 0; i < input\_size; i++) printf("%d\n", h1->heap[i + 1].key);
134. printf("\n");
135. }
136. // Perform the heap sort
137. heap\_sort(h1, output, input\_size);
138. if (input\_size < 20) {
139. printf("Sorted data\n");
140. for (int i = 0; i < input\_size; i++) printf("%d\n", output[i].key);
141. printf("\n");
142. }
143. // Your code should pass the following function (returning 1)
144. if (check\_sort\_results(output, input\_size))
145. printf("Sorting result is correct.\n");
146. else
147. printf("Sorting result is wrong.\n");
148. }

2. 결과 및 콘솔 창

1) input\_size= 1000일 경우



2) input\_size=10인 경우



100, 5000인 경우 모두 동일한 결과.

3. run time.

Building max heap의 time complexity는 tight bound O(n)이다. 따라서 running time도 데이터의 수와 정비례 해서 올라갈 것이다. 만약 n=10의 run time이 1초라면 n=100의 경우는 10초, n=5000의 경우는 500초이다.

4. building max heap 분석

1)

void build\_max\_heap(HeapType \*h)

{

int length = h->heap\_size;

int i = length / 2;

int parent;

int child;

int temp;

length 는 heap size를 직접 받아서 사용했고 i에는 parent node의 index를 저장하기 위해 2로 나눴다. Parent와 child는 while 문을 수행하면서 바뀔 수 있기 때문에 따로 또 변수를 선언하였고 temp는 swap을 수행하기 위해 변수를 선언하였다.

2)

while (i!=0) {

parent = i;

비교를 진행하는 큰 while문의 조건은 i!=0으로 했다. Root node의 index가 1 일 것이기 때문이다.

3)

if (parent \* 2 == h->heap\_size)

child = parent \* 2;

else {

if (h->heap[parent \* 2].key > h->heap[parent \* 2 + 1].key)

child = parent \* 2;

else child = parent \* 2 + 1;

}

만약 parent\*2가 heap size랑 똑같다면 맨 자식노드가 딱 하나뿐인 맨 마지막 parent node를 뜻한다. 이런 경우에는 주저없이 child=parent\*2로 선언하고 만약 아닌 경우라면 child node들의 key 갑을 비교해서 더 큰 것을 child로 해준다.

4)

while (child<=h->heap\_size&&(h->heap[parent].key<h->heap[child].key)) {

temp = h->heap[child].key;

h->heap[child].key = h->heap[parent].key;

h->heap[parent].key = temp;

parent = child;

child = child \* 2;

if (child > h->heap\_size)

break;

else{

if (h->heap[child + 1].key > h->heap[child].key)

child = child + 1;

}

}

만약 부모 노드가 자식노드보다 작은 경우 계속 바꿔줘야 한다. 이 때, heapsize를 초과하지 않도록 제동장치를 걸어두고 계속 parent에 child을 넣어주면서 비교 연산이 완전히 끝날 때 까지 swap를 진행해 준다.

5)

i--;

}

하나의 부모 연산이 끝나면 다음 부모 연산으로 넘어가기 위한 것이다.