**Heap정렬을 하기 전에... Heap이란?**

- Heap은 '더미'라는 뜻을 갖고 있음. (완전 이진 트리 기반의 더미와 모습이 비슷한 자료 구조를 의미)

- Heap은 완전 이진 트리(Complete Binary Tree)다. (이진 탐색 트리와는 다르게 중복을 허용한다)

- 여러 개의 값들 중, **가장 큰 값**이나 **가장 작은 값**을 **빠르게 찾아내도록** 만들어진 자료 구조.

- 부모 노드의 Key >= 자식 노드의 Key : MAX Heap

- 부모 노드의 Key <= 자식 노드의 Key : MIN Heap

**\* Heap의 목적**

- 전체 정렬을 할 필요 없이, 삭제 연산이 수행될 때마다 가장 큰 값(or 가장 작은 값)을 찾아내는 것.

**\* Heap의 구현**

- 배열로 구현한다.

- root 노드를 1번 배열이라고 가정 했을 때, 다음 그림처럼 순서대로 배열에 정렬 시킬 수 있다.

- 현재의 노드를 기준으로

1) 부모의 노드를 알고 싶다면 : (현재 노드의 인덱스) / 2

2) 왼쪽 자식의 노드를 알고 싶다면 : (현재 노드) \* 2

3) 오른쪽 자식의 노드를 알고 싶다면 : (현재 노드) \* 2 + 1

**[삽입 알고리즘]**

1) Heap의 끝에 새로운 노드를 삽입한다.

2) 삽입된 노드와 부모 노드 키 값 비교(MAX Heap 기준)

if ) 삽입 노드의 키 값 > 부모 노드의 키 값 : 삽입 노드와 부모 노드 Swap

else ) 종료

3) 삽입 노드로 계속 2번을 종료될 때까지 반복.

**[Code]**

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25 | #include<stdio.h>  #include<stdlib.h>    typedef struct \_Heap{      int heap[200], heap\_size;  }H;    H heap1;    void Swap(int \*a, int \*b) {      int temp = \*a;      \*a = \*b;      \*b = temp;  }    void insert\_max\_heap(int item) {      int idx = ++heap1.heap\_size;        heap1.heap[idx] = item;        while ((idx > 0) && (heap1.heap[idx / 2] < heap1.heap[idx])) {          Swap(&heap1.heap[idx / 2], &heap1.heap[idx]);          idx /= 2;      }  }  [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

**[삭제 알고리즘]**

1. 루트 노드를 삭제한다. (루트 노드는 MAX Heap에서 최대, MIN Heap에서 최소)

2. 삭제된 루트 노드에는 히프의 마지막 노드를 가져다 놓는다.

3. MAX Heap이라고 가정하고 시작하자.

1) 자신의 왼쪽 노드와 값을 비교한다.

if ) 자신이 더 크다

- 오른쪽 자식과 비교.

if ) 오른쪽 자식보다도 자신이 크다 : 오른쪽 노드와 Swap

else ) 오른쪽 자식보다는 작다 : 왼쪽 노드와 Swap

else ) 자신이 더 작다 : 종료

4. 3번이 끝날 때까지 반복.

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | int delete\_max\_heap() {      int root\_node = heap1.heap[1];      heap1.heap[1] = heap1.heap[heap1.heap\_size--];        int parent = 1, child = 2;        while (child <= heap1.heap\_size) {          // child가 히프의 마지막 노드가 될 수도 있으므로 첫 번째 조건 필수.          // 왼쪽 자식노드와 오른쪽 자식노드 비교, 더 큰 노드를 타겟으로 잡음.          if (child < heap1.heap\_size && heap1.heap[child] < heap1.heap[child + 1]) child++;            // 자식 노드 중 가장 큰 노드와 부모 노드 비교시, 부모노드가 크거나 같으면 종료          if (heap1.heap[parent] >= heap1.heap[child]) break;            // 부모 노드와 자식 노드 교체          Swap(&heap1.heap[parent], &heap1.heap[child]);            // 인덱스 재설정          parent = child;          child \*= 2;      }        return root\_node;  }  [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

**1. Heap Sort**

- 굳이 전체를 전부 정렬할 필요 없이, 가장 큰(or 가장 작은) 원소부터 차례로 추출할 수 있는 Heap의 장점을 이용한 Sort

- 오름 차순으로 정렬시에는 MIN Heap, 내림 차순으로 정렬시에는 MAX Heap을 구성하면 된다.

- 시간 복잡도는 O(nlog2n)

- 주로 Priority Queue(우선 순위 큐)를 만들 때 자주 쓰인다.

**2. Quick Sort**

- 분할 정복(Divide & Conquer) 방법에 근거한 정렬 방법.

- 합병 정렬(Merge Sort)과 비슷하게 전체 리스트를 2개의 부분 리스트로 분할한 후, 각 부분 리스트를 다시 퀵 정렬함.

- 보통 재귀로 설계함.

- 시간 복잡도 마찬가지 O(nlogn)

**[정렬 메커니즘]**

1. pivot을 설정(가장 왼쪽, 가장 오른쪽, 가운데, 랜덤 등 무작위로 선택 가능)

2. 해당 pivot을 제외한 가장 왼쪽을 left, 가장 오른쪽을 right으로 설정.

1) left가 pivot보다 큰 값을 찾을 때까지 left를 증가.

2) right가 pivot보다 작은 값을 찾을 때까지 right를 감소.

3) left와 right가 교차하지 않은 상태인지 확인한 후, left index의 값과 right index의 값을 Swap 해준다.

3. 위의 과정을 반복한다. left와 right가 교차하는 순간, 교차 위치(left)와 pivot의 값을 Swap해준다.

4. 교차 위치를 기준으로 두 개의 구간으로 나누어, 재귀함수를 실행한다.

**[Code]**

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30 | void Quick\_Sort(int \*arr, int left, int right) {      // 인덱스 교차시 재귀에서 벗어남      if (left >= right) return;        // pivot은 배열 오른쪽 끝으로 설정.      int pivot = right;        // l은 left, r은 피봇 바로 왼쪽으로 설정.      int l = left;      int r = right - 1;        while (l < r) {            // left는 피봇보다 큰 놈이 나올 때까지 증가          while (l < r && arr[pivot] > arr[l]) l++;            // right는 피봇보다 작은 놈이 나올 때까지 감소          while (l < r && arr[pivot] < arr[r]) r--;            // 두 인덱스가 교차하지 않았는지 체크한 후, left와 right의 값 Swap          if (l < r) Swap(&arr[l], &arr[r]);      }        // 위에서 정렬이 다 끝나면 pivot과 교차지점 Swap      Swap(&arr[pivot], &arr[l]);        // 재귀로 이분할      Quick\_Sort(arr, left, l - 1);      Quick\_Sort(arr, l + 1, right);  }  [*Colored by Color Scripter*](http://colorscripter.com/info#e) | [cs](http://colorscripter.com/info#e) |

[장점]

- 타 정렬보다 속도가 빠르다.

- 추가 메모리 공간을 필요로 하지 않는다.

[단점]

- 그러나, 이미 정렬이 된 배열을 Sorting할 때에는 O(n^2)의 최악의 복잡도가 발생할 수 있다는 치명적인 단점을 갖고 있다.

**[좀 더 효율적인 Quick Sort]**

- 단순히 pivot을 왼쪽, 가운데, 오른쪽 혹은 랜덤으로 지정하는 방법보다 훨씬 효율적으로 pivot을 설정하는 방법으로 중간값 Quick Sort가 있다.

- 일반적으로 리스트의 왼쪽, 가운데, 오른쪽 3개의 데이터 중, 크기 순으로 중간 값을 설정하는 방법(Median of Three Quick Sort)가 있다.

**[정렬 메커니즘]**

<출처 : [코딩과 디버깅 사이](https://m.blog.naver.com/PostView.nhn?blogId=occidere&logNo=220870294816&proxyReferer=https%3A%2F%2Fwww.google.co.kr%2F)>