Heap

출처
CLRS 6.1 - 6.5
https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/lecture-videos/MIT6_006F11_lec04.pdf

◎(binary) heap : 이진 힙 자료구조란 거의 완전 이진 트리로 볼 수 있는 배열 객체로 추상 자료형(ADT)이다. An every visualized an a nearly(almost) complete binary tree.

완전 이진 트리(complete binary tree): 가장 낮은 레벨을 제외하고는 완전히 차 있고, 가장 낮은 레벨은 왼쪽부터 채워진 이진 트리. 이를 almost complete binary tree라고도 부름.
포화 이진 트리(perfect binary tree): 모든 내부 노드가 2개의 자식노드를 가지고, 모든 단말 노드가 동일한 깊이나 레벨을 가짐

◎heap의 사용: 정렬 알고리즘(heap sort라고 함. insert sort, merge sort와 달리 좋은 성질을 가짐), 우선순위 큐(priority queue)
우선순위 큐(priority queue) : Implements a set S of elements s, each of elements associated with a key.

◎heap 에서 연산
- insert(S, x): insert element x into set S
- max(S): return elements of S with the largest key
- extract_max(S): return elements of S with the largest key and remove it from S
- increase_key(S, x, k): increase the value of x's key to the new value k

◎heap on a tree (힙의 정의)
- root of tree: first element (i=1)
- parent(i) = i/2
- left(i) = 2i
- right(i) = 2i+1

◎ max heap property, min heap property
max heap property(A[parent(i)] ≥ A[i])
the key of a node is ≥ the keys of its children
(⇒ max heap에서 가장 큰 값은 루트에 저장되고, 서브 트리는 자신의 루트보다 크기 않은 값을 가진다.)

min heap property(A[parent(i)] ≤ A[i])
the key of a node is ≤ the keys of its children
(⇒ min heap에서 가장 작은 값은 루트에 저장되고, 서브 트리는 자신의 루트보다 작지 않은 값을 가진다.)

(max heap property, min heap property 둘 다 재귀적으로 정의되어있다. 이 성질을 만족하는지 보려면 트리의 모든 노드에 대해 성립해야한다. 단, 단말노드는 자식노드가 없기 때문에 체크하지 않아도 된다)

◎max heap에서 연산 생각해보기
max -> 쉽다. 루트 노드의 값만 알면된다. 이 때 변경할 것이 없기 때문에 여전히 max heap 이다. (일반적으로 자료구조를 말할 때 이전의 자료구조 형태를 유지해야한다.)
extract_max -> 상대적으로 어렵다. 루트 노드의 값만 알면 되는게 아니라 루트 노드를 제외시키고 다시 max heap property를 만족하게 만들어야하기 때문이다.
extract_max만 잘 설계하면 extrack_max를 지속적으로 실행하면서 정렬 알고리즘을 얻을 수 있다.

◎ heap operations
build_max_heap: produced a max heap from an unordered array (O(n))
max_heapify: correct a single violation of the heap property in a subtree at its root (O(log n))
insert
extract_max
heapsort

◎ Max_heapify

- Max_heapify(A, i)
- precondition: Assume that the trees rooted at left(i) and right(i) are max-heaps.
- 위에 가정을 만족하면서 A[i]가 max heap property를 어기는 경우, A[i]를 내려보내서 인덱스 i를 루트로 하는 서브 트리가 최대 힙이 되도록 한다.

(단말노드들은 정의에 의해 max heap이기 때문에 max_heapify를 하기 위한 가정을 만족시킨다. max_heapify를 아래에서부터 시행하면 max heap을 만들 수 있다.)

- **Max_Heapify Pseudocode**

```
l = left(i)
r = right(i)
if (l <= heap-size(A) and A[l] > A[i])
        then largest = l else largest = i
if (r <= heap-size(A) and A[r] > A[largest])
        then largest = r
if largest ≠ i
        then exchange A[i] and A[largest]
                Max_Heapify(A, largest)
```

- time complexity: O(log n)
  이걸 계산할 때 중요한 사항: 1. 완전 이진트리이다. (트리의 높이가 log n) 2. 전제조건으로 1개의 값만 위반한 상황이다

◎ build_max_heap

- build_max_heap(A)
- **Max_Heapify Pseudocode**
  for i=n/2 downto 1

do Max_Heapify(A, i)

- Why start at n/2?
  (Because elements A[n/2 + 1 … n] are all leaves of the tree 2i > n, for i > n/2 + 1)
  max_heapify를 여러번 호출하는데 그때마다 전제조건을 만족해야한다.
  모든 경우에 n/2 +1 부터 n까지는 단말노드이므로 이미 max heap property를 만족하고 있다. 그러므로 노드 n/2 부터 노드 1까지 살펴보면 된다.
  - 복잡도는?
    O(n log n): n/2개에 대해서 각 과정이 log n인 것처럼 보임
    더 정확한 시간을 구해보자.

    O(n): 높이 올라갈수록 작업량은 증가하고, 노드의 개수는 적어진다.

    Observe however that Max_Heapify takes O(1) for time for nodes that are one level above the leaves, and in general, O(l) for the nodes that are l levels above the leaves. We have n/4 nodes with level 1, n/8 with level 2, and so on till we have one root node that is lg n levels above the leaves.

    (단말노드: n/2개, -> 작업량 O(1)
    단말 노드의 1 level: n/4개, -> 작업량 O(1)
    단말 노드의 2 level: n/8개, -> 작업량 O(2)
    … ,
    log n level: 1개-> 작업량 O(log n))

    Total amount of work in the for loop can be summed as: 노드n/4개* (1 c) + n/8 (2 c) + n/16 (3 c) + … + 1 (lg n c)
    Setting n/4 = 2^k and simplifying we get: c*2^k (1/2^0 + 2/2^1 + 3/2^2 + … (k+1)/2^k)
    The term is brackets is bounded by a constant(3)!
    This means that Build_Max_Heap is O(n)

◎ Heap-Sort

1. Build Max Heap from unordered array;
2. Find maximum element A[1];
3. Swap elements A[n] and A[1]: now max element is at the end of the array!
4. Discard node n from heap (by decrementing heap-size variable)
5. New root may violate max heap property, but its children are max heaps. Run max_heapify to fix this.
6. Go to Step 2 unless heap is empty.

- **Heap-Sort Pseudocode**

```
Build_max_heap(A)   O(n)
for i = A.length downto 2
        exchange A[1] with A[i]  O(1)
        A.heap-size = A.heap-size -1  O(1)
        Max_heapify(A, i)  O(log n)
```

- Heap-Sort time complexity : O(n log n)

◎ Priority queues

- heap을 기반으로 우선순위 큐를 구현해보자. (다른 방식으로도 우선순위 큐를 구현할 수 있음)
- A priority queue is a data structure for maintaining a set S of elements, each with an associated value called a key

- max-priority queue operations
  insert(S, x): **O(log n),** inserts the element x into the set S, which is equivalent to the operation S = S ∪ {x}
  maximum(S): **O(1),** returns the element of S with the largest key.
  extract_max(S): **O(log n),** removes and returns the element of S with the largest key.
  increase_key(S, x, k):**O(log n),** increases the value of element x's key to the new value k, which is assumed to be at least as large as x's current key value

- min-priority queue operations
  insert(S, x)
  minimum(S)
  extract_min(S)
  decrease_key(S, x, k)

- max-priority queue 사용: 공유 컴퓨터에서 작업 순서를 계획하는 것.
- min-priority queue 사용

- **heap-maximum(A) O(1)**
  return A[1] O(1)

- **heap_extract_max(A) O(log n)**
  if A.heap-size < 1 O(1)
      then error 'heap underflow'
  max = A[1]   O(1)
  A[1] = A[A.heap-size]   O(1)
  A.heap-size = A.heap-size - 1   O(1)
  max_heapify(A, 1)   O(log n)
  return max

- **heap_increase_key(A, i, key) O(log n)**
  if key < A[i]
      error '새로운 키가 현재 키보다 작음'
  A[i] = key
  while i > 1 and A[parent(i)] < A[i]   O(log n)
      A[i] ↔ A[parent(i)]
      i = parent(i)

- **max_heap_insert(A, key)** **O(log n)**
  A.heap-size = A.heap-size + 1
  A[A.heap-size] = $-\infty$
  heap_increase_key(A, A.heap-size, key)