협성대학교 소프트웨어공학과

자료구조 기말고사 정리

10 ~ 12 주차 강의 내용 정리

이시헌

2023-12-20

10 주차 수업 정리.

10 주차 수업은 느린 정렬 3 종에 대한 내용이었다. 원형 연결 리스트도 약간 언급되었고, 정렬에 대한 기본적인 것 들 또한 언급되었다.

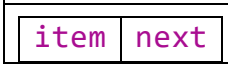
연결된 구조: 흩어진 데이터를 링크로 연결해서 관리하는 구조.

1. 특징
   A. 용량이 고정되지 않음.
   B. 중간에 자료를 삽입, 삭제하는데 유리함.
   C. 특정 위치의 원소에 접근하는데 불리함. (마지막 원소에 접근시 O(n)속도, 첫번째 위치 접근시 O(1)속도.)

연결 리스트

1. 리스트를 노드간 참조를 통하여 구현한 것이 연결 리스트이다.
2. 노드의 구조

```
class ListNode:
    def __init__(self, newItem, nextNode:'ListNode'):
        self.item = newItem
        self.next = nextNode
```



//next 는 다음 노드를 참조. item 은 노드의 값.

3. 연결 리스트를 이용하여 스택 등등의 다양한 자료구조를 구현할 수 있다.
4. 10주차 강의는 스택을 연결 리스트로 구현해보는데 시간을 상당히 썼다.

연결리스트를 이용한 스택의 구현

1. 스택의 탑을 연결리스트의 헤드로 하는 것이 기본 아이디어다. 원소의 삽입과 삭제가 가장 빠른 곳이 연결리스트의 가장 앞부분이기 때문이다.
2. 연결리스트를 이미 클래스로 만들었기에, 교재에서는 그 클래스를 사용하여 스택을 간편하게 구현한다. 예를 들자면, insert(0, x)함수로 연결리스트의 0번 인덱스에 데이터를 삽입하고, pop(0)함수로 0번 인덱스의 데이터를 삭제하는 식이다.
3. 더미 헤드 노드가 존재한다는 것 정도만 잘 기억하고 있으면 된다.

연결리스트의 x번 노드의 값 얻어내기

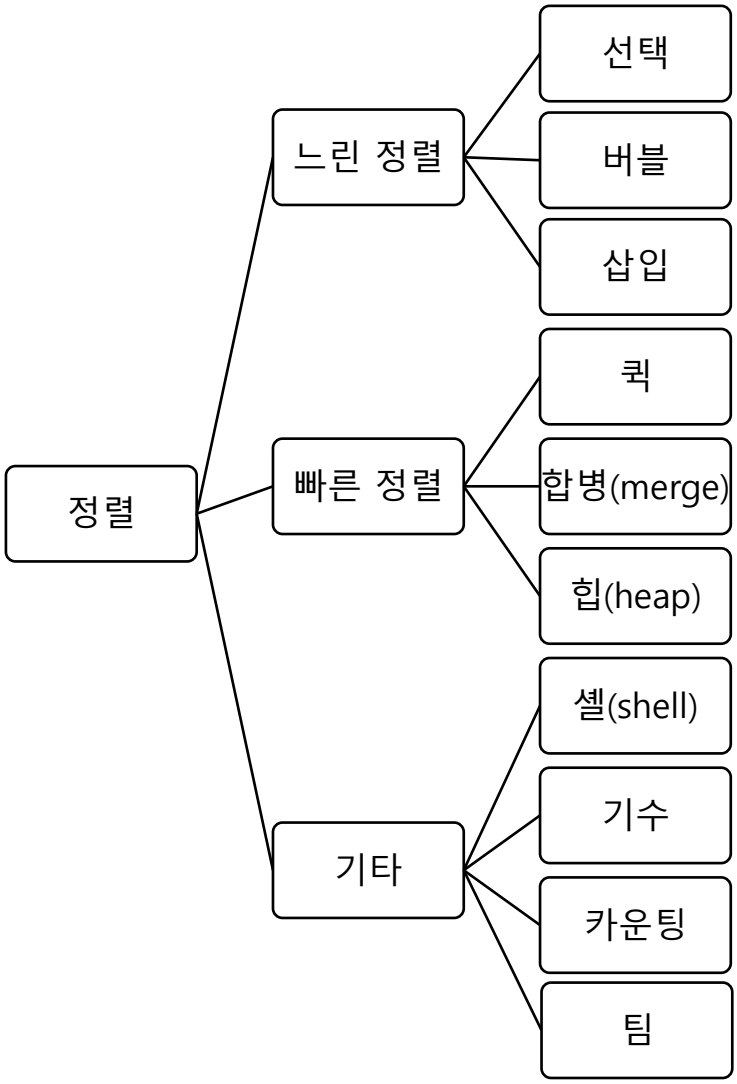
1. 연결리스트는 메모리 주소를 통하여 바로 원하는 인덱스에 접근할 수 없다. -> 첫 번째 노드의 next부터 읽어서, 원하는 값이 나올 때 까지 징검다리를 건너듯 노드의 next로 계속 이동해야 한다.
2. 다음은 교재에 있는 코드의 일부이다. 이 메소드는 연결리스트 클래스 내부에 정의된 메소드이며, 특정 인덱스를 받아서 해당 인덱스에 해당하는 노드의 참조를 반환한다. (p.126)

```
def __getNode(self, i:int):
    curr = self.__head #더미헤드를 curr에 대입. 현재 index는 -1이다.
    for index in range(i+1):  #-1에서 시작했으니 i+1까지 반복해야 함.
        curr = curr.next  #계속하여 다음 노드로 curr가 가리키는 객체를 바꾼다.
    return curr  #curr는 i번 인덱스에 대응되는 객체의 참조다.
```

원형연결리스트

1. 연결리스트의 tail의 다음 노드가 head인 경우를 원형연결리스트라 한다.
2. 이때 노드들에 들어있는 값의 시작 지점을 front라 하고, 끝 지점을 rear라 한다.
3. 일반적인 연결리스트로도 큐를 구현할 수 있지만, 원형연결리스트로 구현하는 것이 더 효율적이다.
4. tail==None인 경우: 큐가 비어있음.
5. peak( ):원소 확인 / enqueue( ):원소 삽입 / dequeue( ):원소 삭제
6. 실제 코드로 어떻게 구현하는지는 생략하겠다. 핵심은 tail위치에 원소를 삽입하고, tail의 다음 위치에서 원소를 삭제하는 것이다.
7. 중간고사 범위였던 원형 큐/덱 부분을 참고하는 것이 좋겠다.

정렬의 종류 (총 10가지)



위의 10가지 정렬 중, 선택 정렬부터 셸 정렬 까지가 시험범위다. 다만 카운팅 정렬까지는 잘 알고 있는 것이 좋다.

팀 정렬은 단순한 하이브리드 정렬의 일종일 뿐이다. (삽입정렬 + 병합정렬)

정렬에 대한 기본 용어들

1. 정렬의 정의: 레코드들을 키의 순서로 재배열하는 것. 혹은, 역이 없는 상태로 만드는 것. (데이터베이스의 primary key 를 떠올려라.)

2. 오름/내림차순
   A. 오름차순: 계속 증가.
   B. 내림차순: 계속 감소.
   C. 비 내림차순: 동일 수 등장을 허용하는 오름차순.

3. 레코드: 정렬해야 할 대상이다. 다른 말로 하면 filed(데이터)가 모인 것이다.

4. 내부/외부 정렬
   A. 내부 정렬(internal): 모든 데이터가 메인 메모리에 속하는 레코드를 정렬하는 경우를 말함.
   B. 외부 정렬(external): 외부 기억 장치에 대부분의 레코드가 존재하는 경우의 정렬을 말함. <- 먼 옛날, 시퀀셜 저장장치에 별도로 레코드가 분산 저장된 경우에 사용하던 용어이다. (이런 용어가 존재한다는 것 정도만 기억하라. 현대의 거의 모든 정렬은 내부 정렬이다. == 현대에는 거의 항상 데이터를 메모리에 모두 읽어온 상태로 정렬을 수행할 수 있다.)

5. 정렬 알고리즘의 안정성
   A. 안정적: 동일한 두 값에 대한 위치 변경이 없으면 안정적인 것.
   B. 불안정적: 동일한 두 값의 위치 변경 있음.

선택 정렬

1. 알고리즘: 정렬되지 않은 부분에서 가장 작은 것을 찾아서, 정렬된 부분의 직후 원소와 교환하기를 반복한다.
2. 이중 for 문이라는 것을 기억하라. 즉, 동작 횟수가 정해져 있다.
3. 코드는 다음과 같다.

```
def selection_sort(A):
    n = len(A)
    for i in range(n-1):  #i 는 0 부터 n-2 까지 순회.
        least = i
        for j in range(i+1, n):  #j 는 i 다음부터 n-1 까지 순회.
            if A[j]<A[least]:  #이 조건문은 정렬되지 않은 부분에서 최솟값을 찾아내는 것.
                least = j
        A[i], A[least] = A[least], A[i]  #j 로 순회하는 for 문이 종료되면, least 변수에 최솟값이 있으므로, 교환을 수행하면 됨.
```

4. 시간 복잡도는, 내부의 for 문이 반복하는 범위가, 외부의 for 문의 1 회 수행마다 1 씩 감소하는 것을 통하여 계산할 수 있다. 아래 수식을 보라.
   A. $(n-1)+(n-2)+\cdots+2+1=\frac{n(n-1)}{2}=O(n^2)$
   B. 정렬을 끝내기 위해서는 반복문이 항상 모두 수행되어야 하므로, 어떠한 경우에도 시간 복잡도는 ($O(n^2)$으로) 항상 동일하다. (최고, 최악, 평균의 시간 복잡도가 동일함.)
5. 로직이 간단하다는 장점이 있다.
6. n 번째로 작은 수를 찾는 작업을 수행할 수 있다는 점에서 쓸모가 있다. -> 하지만 퀵 정렬이 더 우수하다.

버블 정렬

1. 알고리즘
   A. 인접한 두 개의 레코드를 비교하여 더 큰 것이 오른쪽으로 가도록 교환.
   B. 모든 레코드에 대하여 비교와 교환을 수행하면, 그것을 스캔이라 함.
   C. 한 번의 스캔이 완료될 때 마다 하나의 레코드가 정렬이 완료됨. (1 회 스캔마다 맨 오른쪽에 하나씩 정렬이 완료된 데이터가 쌓임)
   D. 남은 영역에 대하여 A~C 를 계속 반복.
2. 이 또한 이중 for 문으로 되어있다. 즉, 수행 횟수가 정해져 있다.
3. 스캔에서 교환이 한 번도 일어나지 않은 경우, 정렬이 완료된 것이다. 이를 이용하여 코드를 작성하면, 반복문을 모두 수행하지 않고서도 정렬이 완료되는 경우가 생길 수 있다. 다만 이런 방법을 사용한다고 해도, 정말 느린 알고리즘인 것은 변하지 않는다.
4. 버블 정렬은 안정적이다. (상대적 위치 변동이 없다.)
5. 버블 정렬의 코드는 다음과 같다.

```
def bubble_sort(A) :
    n = len(A)
    for i in range(n-1):  #i 는 0 에서 n-2 까지 순회한다.
        bChanged = False
        for j in range (n-1-i):  #j 는 0 에서 n-2-i 까지 순회한다. 바깥 for 문의 1 회 동작마다 이 for 문의 동작 횟수가 1 회 감소하는 것 이다.
            if (A[j]>A[j+1]):  #j 가 0 부터 시작하고, i 는 정렬 되지 않은 부분을 한정하므로, j 와 j+1 을 비교한다.
                A[j], A[j+1] = A[j+1], A[j]   #교환 부분.
                bChanged = True   #스캔 중에 교환이 일어났는지를 체크한다. (정렬이 for 문 완수 이전에 끝났는지를 확인한다.)
        if not bChanged: break;
```

6. 정렬을 완수하기까지의 교환 횟수(:이동 횟수. 비교 횟수가 아니다.)가 너무 많아서 매우 비효율적이다. 그러니 선택정렬이든 버블정렬이든 사용하지 않는 것이 좋다.
7. 비교 횟수는 최상, 평균, 최악의 경우 모두 일정하다.
8. 이동 횟수는 다음과 같다.
   A. 최악(역순으로 정렬된 경우): 비교 횟수*3 / 최선: 0 / 평균: $O(n^2)$

삽입 정렬

1. 느린 정렬 3 종 중에서 가장 중요한 정렬이다.
   A. 삽입 정렬은 분명 느리지만, 레코드의 크기가 작은 경우에는 일반적으로 상당히 빠르다. 이는, 어느 정도 정렬 되어있는 데이터의 경우 삽입 정렬이 빠르게 완수되기 때문이다.
   B. 레코드의 크기가 작은 경우, 어떤 수가 내부에 존재하더라도 어느 정도의 뭉침-경향성-이 존재하기 때문에, 이미 정렬이 어느 정도 완수되었다고 간주할 수 있기 때문이다.
2. for 문 내부에 while 문이 들어있는 구조이다. 즉, 언제 정렬이 완수될지를 미리 예측할 수 없다. (고정 횟수로 동작하지 않는 알고리즘이다.)
3. 안정된 정렬 방식이다.
4. 알고리즘:
   A. 정렬할 대상을 정렬 완료된 부분에 삽입한다. (적절한 위치인지를 비교로 판단한다.)
   B. 다음 정렬할 대상을 완료된 부분에 삽입하기를 반복한다.
5. 코드는 다음과 같다.

```
def insertion_sort(A) :
    n = len(A)
    for i in range(1, n):    #i 는 n-1 까지 순회한다. (모든 원소를 순회한다.)
        key = A[i]           #삽입할 대상을 변수 key 에 저장해둔다.
        j = i-1              #j 는 현재의 i 의 직전부터 순회할 예정이다.
        while j>=0 and A[j] > key:
        #j 는 정렬이 완료된 영역을 순회하며, 0 보다 작아지지 않는다.
            A[j + 1] = A[j]   #while 문에서 비교를 수행하므로, 비교 구문이 참인 경우는
항상 리스트의 원소를 뒤로 한칸 밀어낸다.
            j -= 1       #j 는 한 칸 왼쪽으로 이동한다. (1 작아진다.)
        A[j + 1] = key
        #while 문이 비교 조건에 의해 종료되면, 미리 저장해둔, 삽입할 대상을 삽입한다.
        #j 가 1 감소한 이후 A[j]>key 를 비교하므로, A[j+1]의 위치에 key 를 대입한다.
```

6. 속도
   A. 최선의 경우: $O(n)$ (n-1 회만 동작함. 즉, while 문의 조건식에 의해 교환 코드가 동작하지 않음.)
   B. 최악의 경우: $O(n^2)$ (역순으로 정렬된 경우)

10 주차 수업 정리 끝.

11 주차 수업 정리 시작.
11 주차에는 느린 정렬에 대한 긴 복습으로 강의가 시작되었다.
강의의 중복된 내용은 생략하고, 바로 셸 정렬과 탐색에 대한 내용을 정리하겠다.

셸 정렬

1. 정의: 삽입 정렬을 개선한 것. 임의의 숫자를 기준으로 수를 건너뛰며 삽입 정렬을 수행한다. 이때, 임의의 숫자는 1 이 될 때 까지 작아진다. (간격을 기준으로 레코드를 여러 레코드로 나누어 삽입 정렬을 수행한다는 뜻.)
2. 일반적으로 $O(n^{1.5})$정도의 속도로 동작한다.
3. 간격은 소수로 정하는 것이 유리하며, 적절한 간격에 대한 다양한 연구 결과가 있다.
4. 수업시간에는 코드를 직접 보여주시거나 하지는 않았다. 간단히 이해만 하고 넘기길 바란다.

탐색(Search)

➔ 정의: 테이블에서 원하는 탐색 키를 가진 레코드를 찾는 작업을 말한다.

맵(딕셔너리)

1. 탐색을 위한 자료구조이다.
2. Map ADT
   A. search(key): 해당 키를 가진 레코드 반환.
   B. insert(entry): 해당 엔트리를 맵에 삽입.
   C. delete(key): key 를 가진 레코드를 찾아 삭제.
3. 하여튼 흔히 사용하는 연관 배열을 파이썬에서 딕셔너리라고 하며 자료구조로는 맵이라고 하는 것이다.

순차탐색: 처음부터 끝까지 원하는 대상을 찾을 때 까지 순차적으로 탐색하는 것.
최선: $O(1)$ / 최악: $O(n)$

이진탐색

1. 알고리즘
   A. 리스트의 중간 값을 탐색 대상과 비교하여, 대상이 없는 절반을 버린다.
   B. 남아있는 절반에 대하여 A 를 다시 수행한다.
2. 위와 같은 방식을 '분할정복법'이라 한다.
3. 이진탐색은 정렬이 끝난 배열에서 탐색할 때 효과적이다.
4. 코드는 다음과 같다. 재귀 버전과 반복 버전 두가지의 차이를 중심으로 보라.

```
def binary_search(A, key, low, high) :
    if (low <= high) : #탈출조건.
        middle = (low + high) // 2
        if key == A[middle] :   #탐색 성공.
            return middle
        elif (key<A[middle]) : #탐색 대상보다 A[middle]이 큰 경우
            return binary_search(A, key, low, middle - 1)   #탐색 대상은 왼쪽 절반에
있음.
        else :
            return binary_search(A, key, middle + 1, high)  #탐색 대상은 오른쪽 절반에
있음.
    return None
```

<재귀 버전>

```
def binary_search_iter(A, key, low, high) :
    while (low <= high) :
        middle = (low + high) // 2
        if key == A[middle]:
            return middle
        elif (key > A[middle]):
            low = middle + 1
        else:
            high = middle - 1
    return None
```

<반복 버전>

5. $O(\log n)$의 속도로 동작한다.
   A. a 를 임의의 자연수라고 하면,
   B. $2^a = n$ 인 경우, $\log n = a$ 이므로,
   C. n 개의 원소에서 탐색을 수행한다면, $\log n$만큼 비교를 수행하는 것이다. (n 개의 수를 절반씩 나누어, 나누어진 레코드가 하나의 원소를 가질 때가 최악의 경우인 것이다.)

보간 탐색

1. 정의: 탐색 키가 존재할 위치를 예측하여 탐색을 수행하는 것. 기본적으로 이진탐색이나, 반으로 나누기 위한 키를 특별히 만든 함수로 얻어내는 것이다.
2. 탐색 위치를 정하는 함수는 일반적으로 직선의 방정식과 유사한 형태이다. 다만 레코드에 저장된 데이터의 특성에 따라서 얼마든지 다른 함수를 사용할 수 있다.
3. PPT 에는 다음과 같은 수식을 제시한다.
4. 탐색위치 $= \text{low} + \frac{(\text{high}-\text{low})(\text{k}-\text{A[low]})}{\text{A[high]}-A[low]}$
   A. 괜히 어려워 보이지만, 잘 보면 직선의 방정식이다. 정렬된 레코드의 가장 큰 값과 작은 값을 x 축으로, 레코드의 인덱스 범위를 y 축으로 한 1 차함수인 것이다.

해싱(hashing): 계산에 의한 탐색의 일종이다. (비교 기반이 아니다.)

1. 해싱은 키 값을 사용하여, 데이터의 주소를 계산하는 행위를 지칭한다.
2. 해시함수: 탐색 키를 입력받고, 데이터를 저장할 해시 주소를 반환하는 함수를 말한다.
3. 해시 테이블: 키 값과 연결된 데이터를 저장하는 테이블이다.
4. 충돌: 서로 다른 키에 대하여 계산된 해시 주소가 같은 경우를 말한다.
5. 오버플로우: 해시 테이블이 가득 찬 것을 말한다. 충돌을 처리할 수 없다는 뜻이다.
6. 충돌 처리법 – 실제 충돌 처리 과정은 11 주차 과제물을 볼 것.
   A. 복수의 슬롯 사용
   B. 체이닝: 충돌시 충돌이 일어난 해시 값을 연결 리스트로 연결함.
   C. 선형 조사법: 충돌이 발생하면, 다음 칸에 저장한다. 저장이 가능할 때까지 계속하여 다음 칸이 비었는지 확인한다.
   D. 이차 조사법: 충돌시 해시 값에 대하여 임의의 상수를 더하여 새로운 해시 값 산출. 빈 칸의 해시 주소가 나올 때 까지 상수를 더 크게 바꾼다.
   E. 이중 해시법: 별도의 해시 함수의 산출 주소를, 주 해시 함수의 산출 주소에 더한다. 빈 칸이 나올 때 까지 반복해서 더한다.

7. 군집화: 충돌 처리 과정에서 데이터가 계속 밀리며 특정 위치에 뭉치는 것. 선형 조사법에서 가장 빈번하게 발생한다.
8. 좋은 해시 함수의 조건
   A. 충돌이 적어야 함.
   B. 함수 값이 주소 영역 내에서 고르게 분포해야 함. (군집화가 잘 일어나지 않아야 함.)
   C. 계산이 빨라야 함.
9. 일반적인 해시 함수의 구조
   A. $h(k) = k \bmod M$ //k 는 데이터의 값. M 은 해시 테이블의 크기.
   B. 해시 테이블은 일반적으로 소수 크기로 정한다.
10. 해싱 적재 밀도 = $\frac{\textit{저장된 항목 개수}}{\textit{버킷 개수}}$
   A. 해시 테이블이 넉넉한 공간을 가질수록 해싱이 빨라진다. (충돌 처리 횟수가 감소할수록 빨라지는 것. + 공간이 넉넉하면 충돌을 처리하는 속도 자체도 빨라진다.)

11 주차 수업 정리 끝.

12 주차 수업 정리 시작.
12 주차 강의 내용은, 트리와 힙에 대한 내용이다.

트리: 계층적인 자료를 표현하는데 적합한 구조.

1. 주요 용어 설명
   A. 차수: 자식의 개수.
   B. 나머지 용어는 굳이 적지 않겠다.

이진 트리

1. 정의: 모든 노드가 최대 2 개의 서브 트리를 갖는 트리.
   A. 서브 트리는 공집합일 수 있다.
   B. 이진 트리의 모든 서브 트리는 이진 트리여야 한다. (순환적으로 정의된다.)
2. 분류
   A. 포화(full) 이진 트리: 트리의 각 레벨에 노드가 꽉 차 있는 이진 트리.
   B. 완전(complete) 이진 트리: 순서대로 채워진 이진 트리. (건너뜀이 없이 채워진 이진트리.)
3. 성질
   A. 노드가 n 개이면, 간선은 n-1 개이다.
   B. 높이가 h 이면, h ~ $2^n - 1$ 개의 노드를 갖는다.
      i. $2^n - 1$개 노드: 포화 이진 트리의 경우이다.
      ii. h 개 노드: 일렬로 h 개 층이 이어진 모양. 트리 구조의 의미를 상실한 경우이다. 이때는 배열과 같다.

4. 구현 – 1 차원 배열의 경우.
   A. 이진 트리는 2 차원 공간에서 정의되는 자료구조이지만, 꼭 2 차원 배열로 저장해야 하는 것은 아니다.
   B. 각 레벨에 존재 가능한 원소의 수가 한정되어 있기에, 1 차원 배열로도 표현이 가능하다. (일반적으로 1 차원 배열을 사용한다.)
   C. 0 번 인덱스는 사용하지 않는다.
   D. 레벨 순회와 동일한 방식으로 1 번 인덱스부터 노드를 배정한다.
   E. 이 경우 인덱스와 노드 사이에는 다음과 같은 관계가 성립한다.
      i. 노드 i 의 부모 노드 인덱스: $\frac{i}{2}$
      ii. 노드 i 의 왼쪽 자식 인덱스: $2i$
      iii. 노드 i 의 오른쪽 자식 인덱스: $2i+1$
5. 구현 – 링크 표현 방식의 경우.
   A. 3 개의 필드를 갖는 클래스 객체를 노드로 삼아도 된다.
   B. 이 경우 각 노드는, 왼쪽 자식의 참조와 / 노드 자신의 값과 / 오른쪽 자식의 참조를 저장할 3 개의 변수를 갖는다.
6. 순회
   A. 순회 방법은 크게 두가지가 있다.
      i. 전위 / 중위 / 후위 순회
         ① 전위(preorder): 나 먼저: VLR
         ② 중위(inorder): 나를 중간: LVR
         ③ 후위(postorder): 나를 마지막: LRV
      ii. 레벨 순회 <- 위의 세가지는 별로 중요치 않다. 이것이 가장 중요!
7. 레벨 순회
   A. 큐로 구현한다.
   B. 알고리즘
      i. 루트를 큐에 저장.
      ii. 큐에서 방문할 장소를 하나 꺼내고(삭제하고) 방문한다.
      iii. 방문한 위치의 자식들을 큐에 저장한다.
      iv. ii 와 iii 를 큐가 완전히 빌 때까지 반복한다.

C. 선입선출이므로, 자연스럽게 각 레벨의 모든 원소를 왼쪽부터 오른쪽으로 순회하고, 다음 레벨로 넘어가게 된다.

D. 코드는 다음과 같다.

```
def levelorder(root):
    queue = CircularQueue()
    queue.enqueue(root)  #루트 노드를 큐에 삽입.
    while not queue.isEmpty():
        n = queue.dequeue()   #큐에서 현재 방문한 노드를 꺼냄(삭제함)
        if n is not None:   #큐가 비어있지 않다면 자식을 큐에 삽입. (꺼내지 못했다면 큐가 비어있는 것이다.)
            print(n.data, end=' ')
            queue.enqueue(n.left)   #왼쪽 자식을 큐에 삽입.
            queue.enqueue(n.right)   #오른쪽 자식을 큐에 삽입.
```

8. 노드의 개수와 단말 노드의 수와 트리의 높이는 순환호출로 셀 수 있다.

A. 각각의 코드는 다음과 같다.

```
#노드의 개수
def count_node(n):
    if n == None:   #공백 트리인 경우 0 리턴. (순환호출에 사용된 n.left 와 n.right 가 없는 경우를 말함.)
        return 0
    else:   #두 자식을 순환호출 함. 동시에 1 을 더하여 현재 노드를 센다.
        return 1 + count_node(n.left) + count_node(n.right)


#단말 노드의 수
def count_leaf(n):
    if n is None:  #없는 노드를 참조한 경우, 0 을 리턴.
        return 0
    elif n.left == None and n.right == None:  #자식이 없는 경우 == 단말 노드인 경우.
        return 1  #1 을 리턴.
    else:
        return count_leaf(n.left) + count_leaf(n.right)

#트리의 높이
def calc_height(n):
    if n == None:   #참조 불가능한 노드의 경우 0 리턴.
        return 0
    hLeft = calc_height(n.left)  #함수는 매 호출마다 두 자식으로 순환호출 수행.
    hRight = calc_height(n.right)
    if hLeft > hRight:   #더 깊은 자식을 호출한 첫 함수 소멸 지점에 1 이 됨.
        return hLeft + 1  #순환호출의 결과는 이 지점에서 계속 쌓임.
    else:
        return hRight + 1
```

9. 모스 부호 디코딩에 이진 트리가 유용하다. 기억만 해 둘 것.

힙(heap)

1. 정의: 완전이진트리 기반의 자료구조.
2. 특징
   A. 가장 크거나 작은 값을 빠르게 찾는데 유용하다.
   B. 우선순위 큐와 유사하다. 사실, 우선순위 큐를 구현하는데 힙을 사용한다.
   C. 힙은 대표적인 우선순위 큐이다.
   D. 일반적으로 1 차원 배열을 통하여 구현한다.
3. 두 종류가 있다.
   A. Max Heap: 큰 수부터 나오는 힙. 〈- 부모 노드의 키 값은 항상 자식 노드의 키 값보다 크거나 같은 완전이진트리이다.
   B. Min Heap: 작은 수부터 나오는 힙. 〈- [부모 〈= 자식]
4. 구현
   A. 데이터 추가(UpHeap): 말단에 추가 후, 올바른 자리에 도달할 때 까지 〈부모 노드와 비교〉하여 교환 수행.
   B. 데이터 삭제(DownHeap): 루트 노드를 POP 수행 후, 말단을 없어진 루트 자리에 놓고, 옮겨진 말단이 제 자리를 찾아갈 때 까지 〈자식노드와 비교 + 자식노드간 비교〉를 수행하여 교환하기를 반복.
5. 힙 정렬: 힙에 데이터를 모두 넣고, 모두 꺼내면 정렬이 완료되어 있다.
   A. 삽입에 $O(\log n)$ 소요 + 삭제(꺼내기)에 $O(\log n)$ 소요.
   B. 즉, 힙 정렬에는 $2\log n$ 의 시간이 소요된다.
   C. 하나의 레벨이 올라가거나 내려갈 때 마다, 전체 노드의 절반을 버리는 것과 동일하기에 $O(\log n)$ 이 소요되는 것이다. (각 레벨의 노드 개수는 두배 차이가 난다는 것을 기억하라.)

6. 코드는 삽입과 삭제만 보라.

```
#삽입 연산(UpHeap)
def insert(self, n) :
    self.heap.append(n)   #맨 마지막 노드에 n 삽입.
    i = self.size()    #방금 삽입한 노드의 인덱스 값을 i 에 저장.
    while (i != 1 and n > self.Parent(i)):   #부모보다 큰지 비교하며 반복 수행.
        self.heap[i] = self.Parent(i)    #부모를 자식 위치로 집어넣음.
        i = i // 2   #인덱스를 부모 인덱스로 변경.
    self.heap[i] = n   #반복문이 종료되었다면, 현재 i 는 적절한 위치이다. 그러므로
i 위치에 n 을 삽입.

#삭제 연산(DownHeap)
#피곤해서 로직이 눈에 잘 안들어오네...
#while 의 조건식하고 그 내부의 if 문 두개만 잘 읽어보길 바란다. 근데 상당히 귀찮은 코드라서
시험에 안나올 것 같기는 하다...
def delete(self) :
    parent = 1
    child = 2
    if not self.isEmpty() :
        hroot = self.heap[1]
        last = self.heap[self.size()]
        while (child <= self.size()):
            if child<self.size() and self.Left(parent)<self.Right(parent):
                child += 1
            if last >= self.heap[child] :
                break;
            self.heap[parent] = self.heap[child]
            parent = child
            child *= 2;
        self.heap[parent] = last
        self.heap.pop(-1)
        return hroot
```

7. 허프만 코드

A. 비손실 압축 방식에 사용되는 가변길이코드의 일종이다.

B. 자주 사용되는 문자를 더 작은 크기의 이진 코드로 대응시킨 코드이다.

C. 최소 힙을 구성하면서 각 문자별 이진 코드를 생성할 수 있다.

D. 생성된 힙을 읽으며 디코딩을 하고, 생성된 힙으로부터 읽은 값으로 인코딩을 수행한다. 즉, 자연스럽게 고빈도 문자에게 짧은 길이의 이진수 코드를 배정할 수 있는 것이다.

E. 개념만 대충 이해하고 넘어가라.

12 주차 수업 정리 끝.

이후 13, 14, 15 주차 강의 내용의 정리는 생략하겠다. 필기 내용을 볼 것.
(*)