알고리즘과 자료구조 워크북

교과목명 : 알고리즘과 자료구조
차시명: 14차시 트리2

◈ 담당교수: 신일훈 **(서울과학기술대학교)**

◉ 세부목차
- 이진탐색트리 구현
- 이진탐색트리 단점 및 보완 방법

학습에 앞서

■ 학습개요

이진탐색트리는 이진트리의 한 종류로서 모든 노드가 왼쪽 서브트리의 노드들보다는 큰 키 값을 갖고, 오른쪽 서브트리의 노드들보다는 작은 키 값을 갖는 조건을 만족하는 트리이다. 본 강에서는 삽입, 탐색, 최소값찾기, 최대값찾기, 출력, 삭제 등의 연산을 제공하는 이진탐색트리를 구현한다.

■ 학습목표

1	이진탐색트리의 주요 연산을 구현할 수 있다.
2	이진탐색트리의 단점 및 이를 해결하는 방법을 이해한다.

■ 주요용어

용어	해설
이진탐색트리	이진트리의 한 종류이다. 모든 노드가 왼쪽 서브트리의 노드들보다는 큰 키 값을 갖고, 오른쪽 서브트리의 노드들보다는 작은 키 값을 갖는 조건을 만족하는 트리이다.
AVL트리	이진탐색트리의 한 종류이다. 모든 노드에 대해 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이의 차이가 1이하인 조건을 만족한다. AVL 트리는 회전 연산을 통해 위 조건을 만족한다.

학습하기

1. 이진탐색트리 구현
   가. 클래스 BSTNode 정의
- 트리를 구성하는 하나의 노드에 해당
- 멤버 변수
  - left: 왼쪽 자식 노드를 가리키는 링크

  - right: 오른쪽 자식 노드를 가리키는 링크
  - key
  - value

- 메서드
  - 생성자

나. 클래스 BSTNode 코드

```
class BSTNode:
        def __init__(self, key, value, left=None, right=None):
                self.key = key
                self.value = value
                self.left = left
                self.right = right
```

- 생성자는 네 개의 인자, key, value, left, right를 전달받아 이들 값으로 멤버 변수들을 초기화함.
- left, right는 생략 가능하며, 생략된 경우에는 None으로 초기화된다.

다. 클래스 BST 정의

- 이진탐색트리를 나타냄
- 멤버 변수
  - root: 최상위 노드를 가리킴
- 메서드
  - 생성자
  - insert()
  - delete()
  - search()
  - get_min()
  - get_max()
  - print()
  - ...

라. 클래스 BST 생성자 코드

```
class BST:
        def __init__(self):
                self.root = None
```

- 트리가 생성될 때 비어 있는 상태이므로 root를 None으로 초기화함.

마. 클래스 BST의 insert() 메서드 코드

- 트리에 새로운 아이템을 삽입하는 메서드이다.

```
1.          def insert(self, key, value):
2.                  node = BSTNode(key, value)
3.                  if (self.root == None):
4.                          self.root = node
5.                          return True
6.
7.                  target = self.root
8.                  while (True):
9.                          if (target.key == key):
10.                                 # 중복.. value를 수정하도록 구현할 수도 있음
11.                                 del(node)
12.                                 return False
13.                         elif (target.key > key): # 왼쪽으로 진행
14.                                 if (target.left == None):
15.                                         target.left = node
16.                                         return True
17.                                 target = target.left
18.                         elif (target.key < key): # 오른쪽으로 진행
19.                                 if (target.right == None):
20.                                         target.right = node
21.                                         return True
22.                                 target = target.right
```

- 추가할 key, value를 인자로 전달받고, BSTNode를 호출하여 추가할 노드를 생성한다. 이 때 생성된 노드의 left, right는 None으로 초기화된다.
- 3번 라인에서, 기존에 트리가 비어 있는 경우에는 self.root가 None이었을 것이므로 self.root가 생성된 노드를 가리키고, 추가 작업을 종료한다.
- 7번 라인에서 타겟 노드를 최상위 노드로 설정하고, 8번 이하의 while 문을 반복한다.
- 9번 라인에서 타겟의 키와 추가하는 키가 동일하면, 추가하려고 하는 아이템이 존재한다는 의미이다. 경우에 따라 value만 수정하는 것으로 구현할 수도 있지만, 여기서는 에러로 처리한다. 추가할 노드의 메모리를 반납하고 (del()), False를 반환한다.
- 13번 라인에서 타겟의 키가 추가하려는 키보다 크다면, left 링크를 따라간다. 만약 왼쪽 자식이 존재하지 않으면 이 노드의 왼쪽 자식 노드로 생성한 노드를 연결한다. 왼쪽 자식이 존재하는 경우에는 (17번 라인), 왼쪽 자식 노드를 새로운 타겟 노드로 설정하고 while 문을 다시 반복한다.
- 18번 라인에서 타겟의 키가 추가하려는 키보다 작다면, right 링크를 따라간다. 만약 오른쪽 자식이 존재하지 않으면 이 노드의 오른쪽 자식 노드로 생성한 노드를 연결한다. 오른쪽 자식이 존재하는 경우에는 (22번 라인), 오른쪽 자식 노드를 새로운 타겟 노드로 설정하고 while 문을 다시 반복한다.

.

바. 클래스 BST의 print() 메서드 코드

- 모든 노드들의 키들을 출력하는 메서드이다.
- 중위순회(LNR) 방식으로 구현한다. (재귀 함수 활용)
  - print(left subtree)
  - node의 key 출력
  - print(right subtree)
- 결과적으로 키들이 오름차순으로 정렬되어 출력된다.

```
1.          def print(self):
2.                  self.doLNRPrint(self.root)
3.                  print()
4.
5.          def doLNRPrint(self, node):
6.                  if (node != None) :
7.                          self.doLNRPrint(node.left)
8.                          print(node.key, end='  ')
9.                          self.doLNRPrint(node.right)
```

- 1번 라인에 중위순회를 수행하는 print() 메서드가 정의된다.
- 2번 라인에서 최상위 노드를 시작 노드로 하여 중위순회를 수행하는 doLNRPrint() 메서드를 호출한다.
- 중위순회가 종료되면 print()를 호출하여 한 행을 띈다.
- 5번 라인의 doLNRPrint() 메서드는 인자로 주어진 node를 시작 위치로 하여 node이하의 트리를 중위순회한다.
- 중위순회의 알고리즘 그대로, 7번 라인에서 왼쪽 서브트리를 중위순회한다.
- 8번 라인에서 node의 키를 출력한다.
- 9번 라인에서 오른쪽 서브트리를 중위순회한다.

사. 클래스 BST의 get_min() 메서드 코드

- 트리에서 최소값을 가진 노드를 반환하는 메서드이다.
- 최상위 노드에서 시작하여 왼쪽 자식이 없을 때까지 반복해서, 왼쪽 자식이 없는 노드를 만나면, 해당 노드가 최소값을 가진 노드이다.

```
1.          def get_min(self):
2.                  target = self.root
3.                  while (target):
4.                          if (target.left == None):
5.                                  break
6.                          target = target.left
7.                  return target
```

- 2번 라인에서, 최상위 노드를 타겟 노드로 설정하고 탐색을 시작한다.
- 3번 라인에서, 타겟 노드가 존재하는 동안 반복한다. 따라서 트리가 비어 있는 경우에는 self.root가 None이므로 while 문이 수행되지 못하고 None이 반환된다.
- 4번 라인에서, 현재 타겟 노드가 왼쪽 자식을 가지고 있는지 검사한다. 만약 왼쪽 자식이 없으면 현재의 타겟 노드가 최소값을 가진 노드이므로 break를 통해 while 문을 빠져나오고 현재 타겟 노드를 반환한다.
- 6번 라인에서, 현재 타겟 노드가 왼쪽 자식을 가지고 있으면, 왼쪽 자식 노드를 새로운 타겟 노드로 설정하고 3번 라인의 while문을 다시 반복한다.

아. 클래스 BST의 get_max() 메서드 코드

- 트리에서 최대값을 가진 노드를 반환하는 메서드이다.
- 최상위 노드에서 시작하여 오른쪽 자식이 없을 때까지 반복해서, 오른쪽 자식이 없는 노드를 만나면, 해당 노드가 최대값을 가진 노드이다.

```
1.      def get_max(self):
2.              target = self.root
3.              while (target):
4.                      if (target.right == None):
5.                              break
6.                      target = target.right
7.              return target
```

- 2번 라인에서, 최상위 노드를 타겟 노드로 설정하고 탐색을 시작한다.
- 3번 라인에서, 타겟 노드가 존재하는 동안 반복한다. 따라서 트리가 비어 있는 경우에는 self.root가 None이므로 while 문이 수행되지 못하고 None이 반환된다.
- 4번 라인에서, 현재 타겟 노드가 오른쪽 자식을 가지고 있는지 검사한다. 만약 오른쪽 자식이 없으면 현재의 타겟 노드가 최대값을 가진 노드이므로 break를 통해 while 문을 빠져나오고 현재 타겟 노드를 반환한다.
- 6번 라인에서, 현재 타겟 노드가 오른쪽 자식을 가지고 있으면, 오른쪽 자식 노드를 새로운 타겟 노드로 설정하고 3번 라인의 while문을 다시 반복한다.

자. 클래스 BST의 search() 메서드 코드

- 트리에서 특정 키를 가진 노드를 반환하는 메서드이다.

```
1.      def search(self, key):
2.              target = self.root
3.              while (target):
4.                      if (target.key == key):
5.                              return target
6.                      elif (target.key > key):
7.                              target = target.left
8.                      else:
9.                              target = target.right
10.              return None
```

- 2번 라인에서, 최상위 노드를 타겟 노드로 설정하고 탐색을 시작한다.
- 3번 라인에서, 타겟 노드가 존재하는 동안 반복한다. 따라서 트리가 비어 있는 경우에는 self.root가 None이므로 while 문이 수행되지 못하고 None이 반환된다.
- 4번 라인에서, 타겟의 키가 탐색하는 키와 일치하면 발견한 경우이므로 현재의 타겟 노드를 반환하고 종료한다.
- 6번 라인에서 타겟의 키가 탐색하는 키보다 크면 타겟을 왼쪽 자식으로 수정하여 3번 라인의 while 문을 반복한다. (왼쪽 자식이 None인 경우에는 while 문이 만족되지 않으므로 None이 반환된다.)
- 8번 라인에서 타겟의 키가 탐색하는 키보다 작으면 타겟을 오른쪽 자식으로 수정하여 3번 라인의 while 문을 반복한다. (오른쪽 자식이 None인 경우에는 while 문이 만족되지 않으므로 None이 반환된다.)

차. 클래스 BST의 delete() 메서드 코드

- 트리에서 특정 키를 가진 노드를 삭제하는 메서드이다.
- 크게 4가지 경우가 존재한다.
- 첫째는 삭제하려는 노드가 존재하지 않는 경우이다. 이 때는 아무 것도 반환하지 않는다. (None을 반환하도록 구현하는 것도 좋겠다.)
- 둘째는 삭제하려는 노드가 리프 노드인 경우이다. 이 때는 부모 노드와 삭제할 노드를 연결하는 링크를 제거한다.

- 셋째는 삭제하려는 노드가 하나의 자식을 가진 경우이다. 이 때는 부모 노드와 삭제할 노드의 자식 노드를 직접 연결한다. (손자가 자식이 되는 격임.)
- 넷째는 가장 복잡한 경우로, 삭제하려는 노드가 두 개의 자식을 가진 경우이다. 이 때는 삭제할 노드의 왼쪽 서브 트리의 노드들 중 가장 큰 키를 가진 노드를 대체자 노드로 정한다. 대체자 노드는 리프 노드이거나 한 개의 자식 노드를 갖는다. 다음으로 대체자 노드의 키와 값을 삭제할 노드로 복사한다. 이후 대체자 노드를 삭제한다. 대체자 노드는 리프 노드이거나 하나의 자식 노드만 가지므로 둘째 또는 셋째 경우에 해당하며 동일한 방식으로 삭제하면 된다.

```
1.          def delete(self, key):
2.                  target = self.root
3.                  parent = None
4.
5.                  while (target):
6.                          if (target.key != key):
7.                                  parent = target
8.                                  if (target.key > key):
9.                                          target = target.left
10.                                 else:
11.                                         target = target.right
12.                         else:
13.                                 if (target.left == None and target.right == None) :
14.                                        # leaf node
15.                                         self.do_delete(parent, target, None)
16.                                         return target
17.                                 elif (target.left != None and target.right == None) :
18.                                         self.do_delete(parent, target, target.left)
19.                                         return target
20.                                 elif (target.left == None and target.right != None) :
21.                                         self.do_delete(parent, target, target.right)
22.                                         return target
23.                                 else :
24.                                         substitute = target.left
25.                                         sub_parent = target
26.                                         while (substitute and substitute.right != None) :
27.                                                 sub_parent = substitute
28.                                                 substitute = substitute.right
29.
30.                                         self.swap_node(target, substitute)
31.                                         self.do_delete(sub_parent,              substitute,
    substitute.left)
32.
33.                                         return substitute
```

- 2번 라인에서, 최상위 노드를 타겟 노드로 설정하고 탐색을 시작한다.
- 3번 라인에서, 최상위 노드는 parent가 존재하지 않으므로 None 값을 갖는다.
- 5번 라인에서, 타겟 노드가 존재하는 동안 반복하며, 삭제할 노드를 탐색한다. 트리가 비어 있는 경우에는 self.root가 None이므로 while 문이 수행되지 못하고 삭제가 종료된다. (None을 반환하도록 구현하려면 5번 라인의 while문 바깥, 즉 34번 라인에 return None을 추가한다.)

- 6번 라인에서, 타겟 노드와 삭제할 키를 비교하여 같지 않은 경우에는 비교 결과에 따라 왼쪽 자식 또는 오른쪽 자식을 새로운 타겟 노드로 변경하고 다시 탐색을 시작한다. 이때 이전 타겟 노드가 parent 노드로 설정된다.
- 12-33번 라인은 삭제할 노드가 발견된 경우에 해당한다.
- 13번 라인은 삭제할 타겟 노드가 리프노드인 경우이다. 이 때는 부모와 삭제할 타겟 노드를 연결하는 링크를 제거하면 된다. 즉 타겟이 왼쪽 자식이면 left 링크를, 오른쪽 자식이면 right 링크를 None으로 수저한다. 해당 작업은 do_delete() 메서드에서 수행된다. 세 번째 인자가 삭제할 타겟의 자식을 의미하는데 None으로 전달된다. 따라서 부모의 left 또는 right가 세 번째 인자인 None으로 설정되게 됨. 삭제할 노드를 반환한다.
- 17번 라인은 삭제할 타겟 노드가 왼쪽 자식만 가지고 있는 경우에 해당한다. do_delete() 메서드를 호출하여 parent와 왼쪽 자식을 연결한다. 그리고 삭제할 노드를 반환한다.
- 20번 라인은 삭제할 타겟 노드가 오른쪽 자식만 가지고 있는 경우에 해당한다. do_delete() 메서드를 호출하여 parent와 오른쪽 자식을 연결한다. 그리고 삭제할 노드를 반환한다.
- 23번 라인은 삭제할 타겟 노드가 두 자식을 모두 가지고 있는 경우이다.
- 24-28번 라인을 통해, 삭제할 노드의 왼쪽 서브트리에서 가장 큰 키를 가진 노드를 대체자 노드로 설정한다. sub_parent()는 대체자 노드의 부모를 의미한다.
- 30번 라인에서 swap_node() 메서드를 호출하여 대체자 노드의 키와 값을 삭제할 타겟 노드와 맞바꾼다.
- 31번 라인에서 do_delete() 메서드를 호출하여 대체자 노드를 삭제한다. 이 때 대체자 노드는 리프 노드이거나 왼쪽 자식만 가지고 있는 노드일 것이므로 셋째 인자로 대체자 노드의 left를 전달한다. 그리고 삭제할 대체자 노드를 반환한다. (대체자 노드는 삭제할 타겟의 키와 값을 가지고 있음.)

- 다음의 do_delete()와 swap_node()의 정의이다.

```
1.          def do_delete(self, parent, target, child) :
2.                  if (parent == None) :
3.                          self.root = child
4.                  elif (parent.left == target) :
5.                          parent.left = child
6.                  elif (parent.right == target) :
7.                          parent.right = child
8.
9.          def swap_node(self, node1, node2) :
10.                 node1.key, node2.key = node2.key, node1.key
11.                 node1.value, node2.value = node2.value, node1.value
```

- do_delete() 메서드에서, parent가 None인 경우는 삭제할 노드가 최상위 노드인 경우이다. 이 경우는 셋째 인자로 전달된 child가 새로운 최상위 노드가 되므로 self.root를 child로 설정한다.
- 4번 라인에서 타겟이 parent의 왼쪽 자식인 경우에는 parent의 left가 셋째 인자인 child를 가리키도록 한다. 따라서 삭제할 타겟이 하나의 자식만 갖는 경우에는 해당 자식이 parent의 left로 연결되며, 타겟이 자식이 없는 리프 노드인 경우에는 child가 None일 것이므로 left 링크 값이 None으로 설정된다.
- 6번 라인에서 타겟이 parent의 오른쪽 자식인 경우에는 parent의 right가 셋째 인자인 child를 가리키도록 한다. 따라서 삭제할 타겟이 하나의 자식만 갖는 경우에는 해당 자식이 parent의 right로 연결되며, 타겟이 자식이 없는 리프 노드인 경우에는 child가 None일 것이므로 right 링크 값이 None으로 설정된다.
- 9번 라인에서 swap_node() 메서드는 node1과 node2의 key와 value를 서로 맞바꾼다. 즉 대체자 노드의 키와 값이 삭제할 타겟 노드로 복사된다. 반대로, 대체자 노드는 타겟 노드의 키와 값을 갖게 된다.

카. 테스트 코드 및 실행 결과

- insert(), print를 테스트하는 코드이다.

```
1. if __name__ == '__main__':
2.         myBST = BST()
3.         myBST.insert(5, "a")
4.         myBST.insert(7, "b")
5.         myBST.insert(3, "c")
6.         myBST.insert(1, "d")
7.         myBST.insert(9, "e")
8.         myBST.insert(15, "f")
9.         myBST.print()
10.        myBST.insert(8, "g")
11.        myBST.print()
```

- 비어 있는 트리를 생성한 후, 5, 7, 3, 1, 9, 15 키들을 차례로 삽입한다.
- 트리를 출력하면 삽입된 키들이 오름차순으로 출력되어야 한다. (9번 라인)
- 10번 라인에서 8을 추가로 삽입한다.
- 트리를 출력하면 모든 키들이 오름차순으로 출력되어야 한다. (11번 라인)
- 다음은 이에 대한 실행 결과이다.

```
1  3  5  7  9  15
1  3  5  7  8  9  15
```

타. 테스트 코드 및 실행 결과

- get_min(), get_max()를 테스트하는 코드이다.

```
1. if __name__ == '__main__':
2.     myBST = BST()
3.     myBST.insert(5, "a")
4.     myBST.insert(7, "b")
5.     myBST.insert(3, "c")
6.     myBST.insert(1, "d")
7.     myBST.insert(9, "e")
8.     myBST.insert(15, "f")
9.     myBST.insert(8, "g")
10.    myBST.print()
11.    print(myBST.get_min().key)
12.    print(myBST.get_max().key)
```

- 비어 있는 트리를 생성한 후, 5, 7, 3, 1, 9, 15, 8 키들을 차례로 삽입한다.
- 트리를 출력하면 삽입된 키들이 오름차순으로 출력되어야 한다. (10번 라인)
- 11번 라인에서 get_min()을 호출하면 1을 가진 노드가 반환된다. 따라서 1이 출력된다.
- 12번 라인에서 get_max()를 호출하면 15를 가진 노드가 반환된다. 따라서 15가 출력된다.
- 다음은 이에 대한 실행 결과이다.

```
1  3  5  7  8  9  15
1
15
```

파. 테스트 코드 및 실행 결과

- search()를 테스트하는 코드이다.

```
1. if __name__ == '__main__':
2.     myBST = BST()
3.     myBST.insert(5, "a")
4.     myBST.insert(7, "b")
5.     myBST.insert(3, "c")
6.     myBST.insert(1, "d")
7.     myBST.insert(9, "e")
8.     myBST.insert(15, "f")
9.     myBST.insert(8, "g")
10.    myBST.print()
11.
12.    node = myBST.search(7)
13.    if (node != None) :
14.            print(node.value)
15.    else:
16.            print('fail')
17.
18.    node = myBST.search(6)
19.    if (node != None) :
20.            print(node.value)
21.    else:
22.            print('fail')
```

- 비어 있는 트리를 생성한 후, 5, 7, 3, 1, 9, 15, 8 키들을 차례로 삽입한다.
- 트리를 출력하면 삽입된 키들이 오름차순으로 출력되어야 한다. (10번 라인)
- 12번 라인에서 search()를 호출하여 7을 검색한다. 발견되면 해당 노드의 값을 출력하고 발견되지 않으면 fail을 출력한다. 7은 존재하기 때문에 b가 출력되어야 한다.
- 18번 라인에서 search()를 호출하여 6을 검색한다. 발견되면 해당 노드의 값을 출력하고 발견되지 않으면 fail을 출력한다. 6은 존재하지 않기 때문에 fail이 출력되어야 한다.
- 다음은 이에 대한 실행 결과이다.

```
1  3  5  7  8  9  15
b
fail
```

하. 테스트 코드 및 실행 결과

- delete()를 테스트하는 코드이다.

```
1. if __name__ == '__main__':
2.     myBST = BST()
3.     myBST.insert(20, "a")
4.     myBST.insert(24, "b")
5.     myBST.insert(28, "c")
6.     myBST.insert(26, "d")
7.     myBST.print()
8.     myBST.delete(28)
9.     myBST.print()
10.    myBST.delete(20)
11.    myBST.print()
12.    myBST.delete(26)
13.    myBST.print()
14.    myBST.delete(24)
15.    myBST.print()
16.    myBST.insert(20, "a")
17.    myBST.print()
```

- 비어 있는 트리를 생성한 후, 20, 24, 28, 26 키들을 차례로 삽입한다.
- 트리를 출력하면 삽입된 키들이 오름차순으로 출력되어야 한다. (7번 라인)
- 8번 라인에서 delete()를 호출하여 28을 삭제한다. 이후 트리를 출력하면 20, 24, 26이 출력되어야 한다.
- 10번 라인에서 delete()를 호출하여 20을 삭제한다. 이후 트리를 출력하면 24, 26이 출력되어야 한다.
- 12번 라인에서 delete()를 호출하여 26을 삭제한다. 이후 트리를 출력하면 24가 출력되어야 한다.
- 14번 라인에서 delete()를 호출하여 24를 삭제한다. 이후 트리를 출력하면 아무 것도 출력되지 않는다.
- 16번 라인에서 insert()를 호출하여 20을 추가한다. 이후 트리를 출력하면 20이 출력되어야 한다.
- 다음은 이에 대한 실행 결과이다.

```
20  24  26  28
20  24  26
24  26
24

20
```

2. 기타 트리

가. AVL 트리

- 이진탐색트리는 균형이 무너진 skewed tree가 될 수 있음. (대다수 노드가 하나의 자식만 갖는 트리)
- skewed tree의 경우에는 선형 자료구조와 유사한 탐색 시간을 갖게 됨.
- 이 문제를 해결하기 위해 나온 것이 AVL 트리이다.
- AVL 트리는 트리의 균형이 깨지면, 즉 특정 노드의 왼쪽 서브 트리와 오른쪽 서브 트리의 높이 차이가 2 이상이면 회전 연산을 통해, 높이 차이를 1이하로 수정하는 균형 작업을 수행한다.
- 결과적으로 AVL 트리는 모든 노드들에 대해 다음의 조건을 만족하는 이진탐색트리가 된다.
    - 노드의 왼쪽 서브트리의 높이와 오른쪽 서브트리의 높이 차이가 1 이하임

나. 기타 트리

- RB트리

- 이진탐색트리의 한 종류로서 AVL 트리와 마찬가지로 균형을 유지함.
    - 모든 노드가 레드 노드이거나 블랙 노드임.
- B트리
    - 하나의 노드가 여러 자식 노드를 가질 수 있음. 즉 이진트리가 아님.
    - 노드가 여러 자식을 가질 수 있으므로 트리의 높이가 낮아져서 검색 시간이 빨라짐.
- B*트리
    - B트리 계열의 트리임.
- B+트리
    - B트리 계열의 트리로서 리프 노드들이 선형으로 연결되어 있는 특성이 있음.

연습문제

1. 이진탐색트리에서 최소값을 찾기 위한 방법을 설명하시오.

정답 : 최상위 노드에서 시작하여 반복적으로 왼쪽 자식 노드의 링크를 따라간다. 만약 왼쪽 자식 노드가 존재하지 않는 노드를 만나면 해당 노드가 최소값을 가진 노드가 된다.

해설 : 정답 참조

2. 이진탐색트리의 한계에 대해 설명하시오.

정답 : 이진탐색트리는 트리의 균형을 유지하기 위한 회전 연산을 수행하지 않으므로 최악의 경우에 일반적인 연결 리스트의 형태를 가진 skewed tree가 될 수 있다. 이 때의 탐색 연산의 최악 시간 복잡도는 O(N)이 된다.

해설 : 정답 참조

3. 이진탐색트리를 생성한 후, 5, 3, 7, 2, 9의 키 값을 추가하였다. 이 트리를 중위 순회 방식으로 순회하며 키 값을 출력할 때 출력되는 값들을 차례로 적으시오.

정답 : 2 3 5 7 9

해설 : 중위 순회 방식은 최상위 노드부터 순회를 시작한다. 노드 n에 도착하면 n의 탐색을 보류하고 먼저 n의 왼쪽 서브 트리들을 전부 순회하고, 이후, 노드 n을 탐색하고, 마지막으로 노드 n의 오른쪽 서브 트리들을 전부 순회한다. 따라서 이진탐색트리의 경우에는 킷값들이 정렬되어 출력된다.

정리하기

1. 이진탐색트리의 구현
2. 이진탐색트리의 한계와 이를 보완하기 위한 방법

참고자료

- 파이썬과 함께하는 자료구조의 이해, 양성봉, 2021, 생능출판

다음 차시 예고

- 그래프에 대해 학습한다.