hw1

B811056 노이진

April 2021

1 Binary Search

우선 binary search 함수를 만들었다. 리스트의 가운데 순서값인 m보다 작으면 마지막 순서값을 m으로 옮기고, m보다 크면 첫번째 순서값을 m으로 옮겨 비교할 값들을 반씩 줄여나갈 수 있게 했다. u(첫번째 순서값 = 0)가 l(마지막 순서값)보다 같거나 작아질 때까지 반복해서 빠르게 key를 찾아낼 수 있었다 만약 찾아낼 경우에는 break 한 후 몇 번째에 있는지 print하게 했는데 리스트는 0부터 시작이므로 +1을 했다. 그리고 만약 while문의 조건을 만족하지 못해 탈출 했을 경우에는 None을 출력하도록 했다. key값을 찾아서 탈출했을 때는 None을 출력하지 않도록 u < l라는 조건도 추가해줬다. 그리고 리스트를 만들어 그곳에 정렬된 input값을 int형으로 입력받은 후 찾고 싶은 값을 key라는 변수로 설정해 binary search를 실행했다.

2 QuickSort

quicksort를 하기 위해 먼저 partition을 위한 함수를 작성했다. 맨 처음 값을 pivot으로 설정할 것이기 때문에 비교는 그 다음부터 하기 위해 i = l+1로 설정했다. arr[i]번째 값을 계속 pivot값과 비교해 i를 옮겨줬다. 이때 i가 맨 끝까지 갈 수도 있으니까 그 경우 멈추도록 설정했다.

그 다음에는 j를 i와 같은 방식으로 뒤에서부터 비교해줬다. j도 맨 앞까지 이동할 수도 있기 때문에 j >= l이라는 항목을 넣어 멈추도록 설정했다. i값과 j값이 모두 정해지면 i와 j값을 swap해줬다. 그리고 i값이 j보다 커지게 될 경우에는 이미 피봇보다 작은 값들은 피봇 앞으로, 큰 값은 뒤로 이동이 끝났으므로 break 처리 해줬다. 그 후 pivot과 j의 위치를 바꿔준다. 그러면 중앙값인 pivot이 저절로 중앙으로 가게 된다. 그리고 이것은 나눠진 리스트의 크기가 0이나 1이 될때까지 반복된다.

파티션 함수를 만들고 나서는 quicksort함수도 만들었다. quicksort함수는 먼저 피봇을 나누고 그 앞쪽과 뒤쪽 값을 또 계속 quicksort하는 재귀함수이다.

마지막으로 input값을 int형 list로 받아서 저장한 다음에 그냥 그 값을 quicksort해 print했다.

3 MergeSort

먼저 나눠진 값들을 합치기 위한 merge함수를 작성했다. merge 함수는 왼쪽, 오른쪽을 비교해 병합할 것이기 때문에 l과 r 배열을 변수로 받아서 그 각각의 길이를 l_ l, r_l에 저장했다.

i와 j값이 그 배열의 길이를 넘기지 않도록 while문을 설정한 후 왼쪽의 i번째, 오른쪽의 j번째를 비교해 버퍼에 순서대로 집어넣었다. 만약 i값이 더 작아서 i를 넣었을 경우에는 i를 +1했고 j값이 더 작아서 j값을 buffer에 넣었을 경우에는 j를 +1해서 배열에서 한칸 이동했다. 계속 비교해서 buffer에 넣다보면 크기 순으로 buffer에 쌓이게 되고, 그럼 정렬이 된다. 혹시 while문에서 벗어나게 된다면 이때 i나 j가 끝까지 도달했다는 뜻이 되고, 그럴 경우 남은 쪽의 값은 그냥 buffer에 뒤쪽에 순서대로 들어가면 되므로 새로운 while문을 작성해 만약 left가 남았을 경우에는 남은 left값을 순서대로 buffer에 넣도록 하고, right가 남았을 때는 right를 순서대로 buffer에 넣도록 했다. 이때 i값과 j값은 위의 while문에서 쓰던 그대로이므로 이 값을 활용해 buffer에 넣지 않은 값부터 이어서 넣을 수 있다. 그리고 최종적으로 buffer를 return하면 배열된 리스트를 얻을 수 있다.

그 다음에는 merge_sort함수를 작성했다. m값을 arr의 길이의 절반값의 정수형태로 설정하고 그 m을 중심으로 arr를 왼쪽, 오른쪽 배열으로 나눠줬다. 그리고 그런식으로 계속 반으로 나눠서 완전히 쪼개질 수 있도록 left_s = merge_sort(left)이런식으로 재귀함수를 작성했다. 그리고 이렇게 만든 left_s, right_s 값은 merge함수에 넣으면 정렬된 buffer를 return받을 수 있게 된다. 이때 계속 쪼개지다가 arr의 길이가 1보다 작은 값이 될 수도 있기 때문에 만약 arr의 길이가 1보다 작게 된다면 그대로 arr를 return하도록 조건을 만들어줬다.

그리고는 그냥 m_list 리스트를 만들어 int형으로 input을 받고 그 리스트를 merge_sort함수에 넣어 정렬된값을 프린트 할 수 있도록 했다.

4 Binary Tree

트리를 위한 class를 먼저 만들었다. 이때 필요한 것은 이 노드의 자체값과 왼쪽과 오른쪽 노드로 연결하기 위한 값이다.

그리고는 preorder, inorder, postorder를 위한 재귀함수를 작성했다. preorder는 루트− >왼쪽− >오른쪽, inorder는 왼쪽− >루트− >오른쪽, postorder는 왼쪽− >오른쪽− >루트 순서로 방문하기 때문에 루트를 print할 위치를 조정해 그 순서를 나타냈다. 트리는 root라는 클래스 객체를 만든 후 그 객체의 왼쪽, 오른쪽, 그 왼쪽의 왼쪽, 오른쪽의 값을 직접 설정하는 방법을 사용해 만들었다.

그리고는 만들어놓은 preorder, inorder, postorder 함수를 사용해 프린트했다.

5 Class Time

먼저 강의의 개수를 int형으로 input받은 후 3차원 배열을 만들어 강의 번호, 강의 시간, 종료 시간을 강의의 개수만큼 반복해 int형으로 input받았다. 최대한 많은 강의를 배치시키기 위해서는 끝나는 시간이 중요하다고 생각했다. 따라서 배열의 0번째에 끝나는 시간을 받도록 했다. 그리고 먼저 끝나는 시간대로 배열하기 위해 끝나는 시간을 기준으로 sort했다. 그러면 가장 먼저 끝나는 수업부터 배열에 순서대로 배치되기 때문에 내가 강의를 선택하면 끝나는 시간이 갱신될 수 있도록 time이라는 변수를 0으로 초기화해 만들었다. 의 개수를 int형으로 input받은 후 3차원 배열을 만들어 강의 번호, 강의 시간, 종료 시간을 강의의 개수만큼 반복해 int형으로 input받았다. 최대한 많은 강의를 배치시키기 위해서는 시작하는 시간보다 끝나는 시간이 중요하다고 생각했다. 따라서 배열의 0번째에 끝나는 시간을 받도록 했다. 그리고 먼저 끝나는 시간대로 배열하기 위해 끝나는 시간을 기준으로 sort했다. 그러면 가장 먼저 끝나는 수업부터 배열에 순서대로 배치되기 때문에 내가 강의를 선택하면 끝나는 시간이 갱신될 수 있도록 time이라는 변수를 0으로 초기화해 만들었다. 그리고 어떤 강의가 배치될지 which라는 배열을 만들었다.

이제 가장 먼저 끝나는 수업부터 which배열에 넣고 time을 가장 먼저 끝나는 수업의 끝나는 시간으로 갱신한다. 이후에는 시작하는 시간과 time을 비교해 만약 시작하는 시간이 time보다 뒤라면 그 강의를 which배열에 넣고 다시 time을 갱신했다. 만약 시작 시간이 time보다 작다면 continue를 써서 다음 강의 시간과 비교할 수 있게 하였다. 그리고 총 몇개인지 알 수 있게 하는 count함수를 넣어줘서 강의 개수를 셌다. 이렇게 끝나는 시간을 기준으로 해야 가장 많은 수업을 배치할 수 있다.

마지막에는 count값이랑 which 배열을 print했다.

6 Pattern

정해진 패턴을 정규식으로 만드는 것은 구글링을 통해 해결했다.

하지만 문제점은 match나 findall, search를 활용해도 내가 원하는 결과를 얻을 수 없다는 점이었다. match를 사용할 경우 앞쪽에서 패턴이 맞을 경우 무조건 패턴과 일치한다는 결과가 나오고, search도 마찬가지다. 그리고 findall은 전체에 패턴과 맞는 부분이 있으면 무조건 DANGER이 되기 때문에 고민했다.

그래서 결국 fullmatch라는 함수를 구글링을 통해 찾아서 해결했다.

fullmatch는 전체가 패턴에 해당되지 않을 경우 None을 리턴한다. 따라서 result라는 이름의 배열을 만들어 만약 none이 나올 경우 0, 아닐 경우에 1을 저장하도록 해서 0이면 PASS, 1이면 DANGER을 프린트하도록 하였다.