자료구조 실습 보고서

[제 13 주]

제출일 : 2018.6.17.일요일

학번 : 201604137

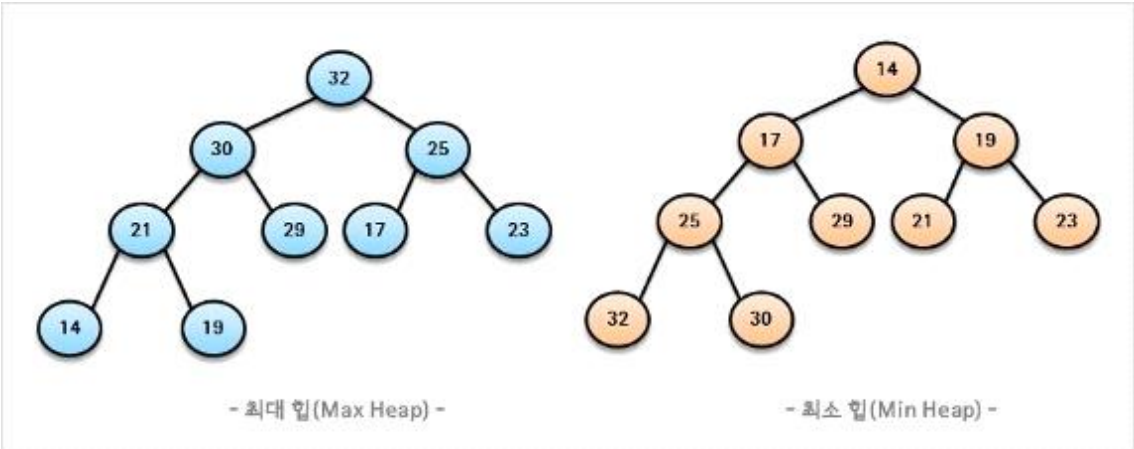
이름 : 김예지

[프로그램 설명서] 힙정렬

–완전 이진 트리에 있는 노드 중에서 키 값이 가장 큰 노드나 키 값이 가장 작은 노드를 찾기 위해서 만든 자료구조

–최대 힙(Max Heap)
부모 노드의 키 값이 자식 노드의 키 값보다 항상 크거나 같은 크기의 관계.
(부모 노드의 키 값 ≥ 자식 노드의 키 값) 의 관계를 가지는 노드들의 완전 이진 트리

–최소 힙(Min Heap)
부모 노드의 키 값이 자식 노드의 키 값보다 항상 작거나 같은 크기의 관계.
(부모 노드의 키 값 ≤ 자식 노드의 키 값) 의 관계를 가지는 노드들의 완전 이진 트리



힙(Heap)에서 항상 가장 큰 원소가 루트 노드가 되고 삭제 연산을 수행하면 항상 루트 노드의 원소를 삭제하여 반환
–최대 힙에서 대해서 원소의 개수만큼 삭제 연산을 수행하여 내림차순으로 정렬 수행
–최소 힙에 대해서 원소의 개수만큼 삭제 연산을 수행하여 오름차순으로 정렬 수행

힙 정렬(Heap Sort) 수행 방법
1. 정렬한 원소들을 입력하여 최대 힙 구성
2. 힙에 대하여 삭제 연산을 수행하여 얻은 원소를 마지막 자리에 배치
3. 나머지 원소에 대해서 다시 최대 힙로 재구성
   원소의 개수만큼 2~3을 반복 수행

[실행 결과 분석]

```
public class HeapSort {
    int[] newData;

    public void sort(int[] a, int size) {
        this.newData=new int[size+1];
        this.newData[0]=0;
        for(int i=0;i<size;i++) {
            this.newData[i+1]=a[i];
        }
        int i;
        for(int j=size;j>0;j--) {
            for(i=size/2;i>0;i--) {
                adjust(this.newData,i,j); //j:정렬시켜준것들을 빼고 제일 마지막을 lastLeaf노드로
            }
            swap(1,j); //첫번째에 있는 노드를 뒤로 정렬시켜준다
        }
        for(int j=0;j<size;j++) {
            a[j]=this.newData[j+1]; //0번 인덱스를 제외하고 1부터 넣기 위해서
        }
        System.out.println();
    }
```

sort 함수에서는 정렬된 노드들을 배열의 뒤로 빼줘서 차례대로 정렬되게 해준다. 정렬된 노드를 빼고 다시 정렬하기 위해 j의 값을 줄여준다

```
    public void adjust(int[] newData, int root, int lastLeaf) {
        int child, rootkey;
        rootkey = newData[root];
        child = 2*root;

        while(child <= lastLeaf) {
            //왼쪽과 오른쪽을 비교한다.
            int idx = child; //child의 인덱스 값을 저장
            int bigger = newData[child]; //일단 왼쪽 자식노드를 넣어서 초기화
            if(child+1 <= lastLeaf) { //child+1이 lastLeaf보다 크면 오른쪽 자식노드가 없다는 것
                if(newData[child] > newData[child+1]) { //왼쪽 오른쪽중에 더 큰 값을 bigger에 넣는다
                    bigger = newData[child];
                    idx = child;
                }else{
                    bigger = newData[child+1];
                    idx = child+1;
                }
            }
            //root와 child를 비교한다.
            if(rootkey < bigger) {
                swap(root,idx);
            }
            child = idx*2;
            root = child/2;
            rootkey = newData[root];
        }
    }
```

adjust함수는 끝 노드부터 시작해서 끝노드와 그 노드의 부모노드를 비교해서 자식노드가 더 크면 부모노드와 자리를 바꿔주는 함수이다.

```
private void swap(int root,int child) {
    int tmp;
    tmp = newData[root];
    newData[root] = newData[child];
    newData[child] = tmp;
}
```

swap함수는 루트와 자식노드를 바꿔주는 역할을 한다.

[정렬 알고리즘의 시간 복잡도]

- 정렬 : 정렬 참고자료

[참고] 정렬 알고리즘 시간복잡도 정리

알고리즘	최선	평균	최악
삽입 정렬	O(n)	O(n²)	O(n²)
선택 정렬	O(n²)	O(n²)	O(n²)
버블 정렬	O(n²)	O(n²)	O(n²)
셸 정렬	O(n)	O(n^1.5)	O(n²)
퀵 정렬	O(nlog₂ n)	O(nlog₂ n)	O(n²)
힙프 정렬	O(nlog₂ n)	O(nlog₂ n)	O(nlog₂ n)
합병 정렬	O(nlog₂ n)	O(nlog₂ n)	O(nlog₂ n)
기수 정렬	O(dn)	O(dn)	O(dn)

[느낀점]

선택정렬: 정렬을 위한 비교 횟수는 많으나 교환 횟수는 상당히 적다는 것이 장점이다. 따라서 교환이 많이 이루어져야하는 자료 상태에서 가장 효율적으로 적용될 수 있는 정렬 방식이다. 선택 정렬이 가장 적합한 자료 상태는 역순 정렬이다. 즉, 내림 차순으로 정렬되어 있는 자료를 오름 차순으로 재정렬할 때 최적이다. 반대로 이미 정렬된 상태에서 소수의 자료가 추가됨으로 재정렬하게 되는 때에는 최악의 처리 속도를 보여준다는 단점이 있다.

삽입정렬: 버블정렬이 조금 비효율적인 면이 있기에 비교횟수를 효율적으로 줄이기 위해서 고안된 삽입정렬이다. 버블정렬은 비교대상의 메모리 값이 정렬되어 있음에도 불구하고 비교연산을 하는 부분이 있는데 반해 삽입정렬은 기본적으로 버블정렬의 비교횟수를 줄이고 ㅋ기가 적은 데이터 집합을 정렬하는 루팅을 작성할 시 버블정렬보다 탁월한 성능 효율을 보인다.

버블정렬: 첫 번째 원소부터 인접한 원소끼리 계속 자리를 교환한다. n개의 원소에 대하여 n개의 메모리를 사용한다. 데이터를 하나씩 비교할 수 있어서 정밀하게 비교가 가능하나 비교횟수가 많아지므로 성능면에서는 좋은 방법이 아니다. 첫 번째 원소부터 마지막 원소까지 반복하여 한 단계가 끝나면 가장 큰 원소를 마지막 자리로 정렬한다

퀵정렬: 연속적인 분할에 의한 정렬이다. 처음 하나의 축을 정해서 이 축의 값보다 작은 값은 왼쪽에 큰 값은 오른쪽으로 위치시킨다. 왼쪽과 오른쪽의 수 들은 다시 각각의 축으로 나누어져 축값이 1이 될 때까지 정렬한다. 안정성이 없으며 가장 많이 사용되는 정렬법이다

합병정렬: 작은 단위부터 정렬해서 정렬된 단위들을 계속 병합해가면서 정렬하는 방식이다. 알고리즘 중에 가장 간단하고 쉽게 떠올릴 수 있는 방법이다. 안정성이 있으며 상당히 좋은 성능을 나타낸다. 큰 결점은 데이터 전체 크기만한 메모리가 더 필요하다는 점이다.

값들이 어떻게 정렬 되어있느냐에 따라서 각 정렬 알고리즘이 어떻게 작용하고 얼마나 시간이 걸리는지가 다 다르다는 것과 얼마나 많은 값들이 있는지에 따라서도 알맞은 정렬 알고리즘이 있는지 알게 되었다.