자료구조 과제 5

20231609 정희선

Problem 1.

Linked binary tree로 나타낸 max heap의 insertion과 deletion 기능을 구현한 프로그램이다.

- **data structure**

1. struct node

각 node는 int형의 key값, 부모 node를 가리키는 node pointer parent, 왼쪽과 오른쪽의 자식 node를 가리키는 node pointer leftChild, rightChild를 멤버 변수로 가진다.

2. Max heap

각 node의 key 값이 그 자식의 key값 보다 작지 않은 tree이다. 이는 새로운 node를 왼쪽부터 insert 하는 complete binary tree이다. Max heap을 구현하기 위해, insertion 기능에서는 node를 insert할 가장 오른쪽 위치를 찾는다.(findInsertPositon 함수) 노드를 삽입한 이후에는 부모 노드와 키 값을 비교하며 올바른 위치를 찾는다.(heapifyUp 함수) deletion 기능에서는 root node를 삭제한 후, 가장 마지막 위치에 있는 node를 root node의 위치로 가져온다.(findLastNode 함수) 이후 자식 node와 key 값을 비교하며 올바른 위치를 찾는다.(heapifyDown 함수)

3. Queue

Tree를 탐색할 때 사용된다. First in First out 구조이므로, tree를 level 순서로 탐색하는 데 사용된다.

- **Algorithm**

1. Swap()

인자로 전달받은 두 node의 key 값을 바꾼다.

2. findInsertPosition()

root 가 NULL이면(tree가 비어있으면), NULL값을 반환한다. 그렇지 않은 경우, queue를 사용해 자식 node의 자리가 꽉 차있지 않은 가장 첫번째 node의 위치를 찾는다.

3. findLastNode()

root 가 NULL이면(tree가 비어있으면), NULL값을 반환한다. 그렇지 않으면, queue를 사용해 level order tree 탐색에서 가장 마지막에 위치한 node를 찾는다.

4. heapifyUp()

node의 부모가 존재하고, 부모의 key값보다 자식의 key값이 크다면(max heap을 위반한다면), 부모와 자식의 key값을 교환한다.

5. heapifyDown()

node의 key 값, leftChild의 key값, rightChild의 key값 모두를 비교한 후 최대 key값을 설정한다. 최대 값이 현재 node의 key보다 크다면, 그 자식 노드의 key와 노드의 key를 교환한다. 교환한 자식 node를 새로운 node로 설정한 후, node가 존재할 동안 반복한다.

6. keyExists()

queue를 사용해 level order로 tree를 탐색하며, 현재 삽입하려는 key값이 이미 tree에 존재하는지 확인한다.

7. Insert()

Key가 이미 존재한다면, insert 하지 않는다. 그렇지 않은 경우, 새로운 node를 생성한 후 가장 오른 쪽 위치에 새 node를 insert한다. 이후 부모 node와의 key값 비교를 통해 max heap을 완성한다.

8. Pop()

Tree가 빈 경우 메시지를 출력한다. 트리가 root node 하나로 구성된 경우 root node를 삭제한다. 일반적인 경우 tree에서 가장 오른쪽에 위치한 node를 찾고, 그 node와 root node의 key 값을 교환한다. 가장 오른쪽에 위치한 node를 tree에서 연결을 끊은 후 메모리를 해제한다. 이후 root node에서부터 자식 node와 key값을 비교하며 max heap을 구현한다.

Problem 2

주어진 preorder traversal을 가지고 binary search tree를 만들고, 그 tree를 inorder과 postorder로 traverse하는 프로그램이다.

- **data structure**

1. struct node

각 node는 int형의 key값, 왼쪽과 오른쪽의 자식 node를 가리키는 node pointer left, right를 멤버 변수로 가진다.

2. Binary search tree

각 node의 왼쪽 subtree에는 해당 node의 key 값보다 작은 key값을 가진 node들로 이루어져 있어야 하고, 각 node의 오른쪽 subtree에는 해당 node의 key 값보다 큰 key값을 가진 node들로 이루어져 있어야 한다.

- **Algorithm**

1. createNode()

새로운 node를 동적할당한다.

1. insert()

tree가 비어있는 경우, insert할 key값을 갖는 새로운 노드를 하나 생성한다. 그렇지 않은 경우, root node의 key값과 insert할 key값을 비교한다. Insert할 key값이 더 큰 경우 root node의 right로, 그렇지 않은 경우 left로 가서 위 과정을 반복한다.

2. constructBST()

preorder traversal로 출력된 tree node의 key들로 구성된 배열과 그 배열의 크기를 인자로 받는다. 배열을 순회하며 해당 element를 key값으로 가지는 node를 tree에 insert한다.

3. inorderTraversal()

left, key, right 순으로 tree를 출력하는 함수이다.

4. postorderTraversal()

left, right, key 순으로 tree를 출력하는 함수이다.

Problem 3

Binary search tree로 구현된 max priority queue 프로그램이다.

- **data structure**
  1. struct node

각 node는 int형의 key값, 왼쪽과 오른쪽의 자식 node를 가리키는 node pointer left, right를 멤버 변수로 가진다.

2. Binary search tree

각 node의 왼쪽 subtree에는 해당 node의 key 값보다 작은 key값을 가진 node들로 이루어져 있어야 하고, 각 node의 오른쪽 subtree에는 해당 node의 key 값보다 큰 key값을 가진 node들로 이루어져 있어야 한다.

- **Algorithm**

1. createNode()

새로운 node를 동적할당한다.

2. keyExists()

queue를 사용해 level order로 tree를 탐색하지 않고, root node부터 node의 key값과의 비교를 통해 다음으로 비교할 자식 노드를 결정함으로써 현재 삽입하려는 key값이 이미 tree에 존재하는지 확인한다.

3. push()

tree가 비어있는 경우, push할 key값을 갖는 새로운 node를 하나 생성한다. 그렇지 않은 경우, root node의 key값과 push할 key값을 비교한다. push할 key값이 더 큰 경우 root node의 right로, 그렇지 않은 경우 left로 가서 위 과정을 반복한다.

4. top()

tree가 비어있지 않은 경우, tree의 가장 오른쪽 노드의 key 값(가장 큰 값)을 출력한다.

5. pop()

tree가 비어있지 않은 경우 tree의 가장 오른쪽 노드로 이동한 후, 마지막 노드의 parent에서 마지막 노드와의 연결을 끊어내고 메모리를 해제한다.

- **Complexity**

모든 함수가 root node에서부터 key값과의 비교를 통해 다음 차례에 어떤 자식 node와 비교할지 결정하기 때문에 complexity는 tree의 height h에 의존하는 O(h)가 된다.