CSE3013 (컴퓨터공학 설계 및 실험 I)
PRJ-2 테트리스 프로젝트 2주차 예비 보고서

서강대학교 컴퓨터공학과 박수현 (20181634)

서강대학교 컴퓨터공학과

1 목적

테트리스 게임에 랭킹을 추가한다.

2 문제

다음과 같은 쿼리를 효율적으로 처리할 수 있는 자료구조를 생각한다.

- $(key, value)$의 2-tuple 원소 삽입
- 원소 삭제
- 정렬된 상태의 원소들에서 특정 인덱스 범위의 원소 조회
- 특정 *key*를 갖는 모든 *value*를 정렬된 상태로 조회

2.1 링크드 리스트 (Linked List)

교재에 소개된 대로 링크드 리스트를 이용해 랭킹을 관리할 수 있다. 링크드 리스트는 항상 정렬되어 있는 상태이다.

삽입 이 상황에서는 링크드 리스트가 정렬된 상태여야 하므로 원소를 특정한 위치에 삽입해야 한다. 링크드 리스트에서 원소를 특정 정렬된 위치에 삽입하는 것은 최악의 경우 마지막 원소까지 순회해야 하므로 $\mathcal{O}(n)$이다.

LINKEDLIST-INSERT(x)

```
1  Allocate new node newNode
2  newNode.value = x
3  curr = head
4  if x > curr.value
5      newNode.link = head
6      head = newNode
7  else
8      while curr.link ≠ NULL and x ≤ curr.next.value
9          curr = curr.link
10     newNode.link = curr.link
11     curr.link = newNode.link
```

삭제 링크드 리스트에서 x번째 원소를 삭제하는 것은 x번째 원소까지 순회하므로 $\mathcal{O}(x)$이다.

LINKEDLIST-REMOVE(x)

```
1  curr = head
2  if x == 1
3      head = head.link
4      Remove curr
5  else
6      for i = 1 to x − 1
7          if curr == NULL
8              error Array index out of range
9          else
10             curr = curr.link
11     next = curr.link
12     curr.link = next.link
13     Remove next
```

특정 범위의 원소 쿼리 l번째 원소부터 r번째 원소를 쿼리하는 것은 r번째 원소까지 순회하므로 $\mathcal{O}(r)$이다.

LINKEDLIST-QUERY(l, r)

```
1  curr = head
2  if (1 ≤ l ≤ r ≤ |list|) ≠ TRUE
3      error Array index out of range
4  i = 1
5  curr = head
6  repeat
7      if l ≤ i ≤ r
8          Add curr to query result
9      i = i + 1
10     curr = curr.link
11 until curr == NULL or i > r
```

2.2 이진 탐색 트리 (Binary Search Tree)

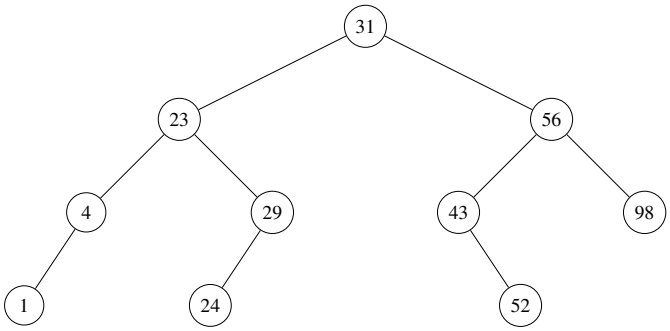
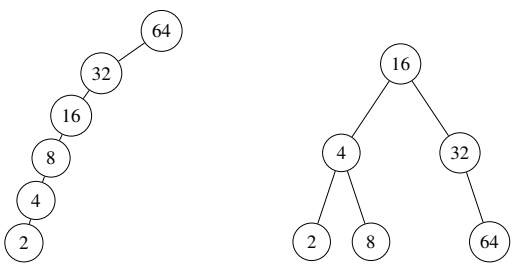


Fig. 1: 이진 탐색 트리의 예시

이진 탐색 트리는 이진 트리의 일종으로서, 각 노드의 왼쪽 서브트리는 해당 노드의 값보다 작은 노드들만, 오른쪽 서브트리는 해당 노드의 값보다 큰 노드들만을 포함하고 있는 트리를 말한다.

이진 탐색 트리에서 원소 삽입과 삭제에 걸리는 시간은 일반적으로 $\mathcal{O}(\log n)$이다. 또한 트리를 중위 순회($\mathcal{O}(n)$)할 경우 항상 정렬된 상태로 순회하게 되는 특징이 있다. 따라서 랭킹을 정렬된 상태로 조회해야 하는 쿼리들을 수행하는 데 효율적이다.



(a) 최악의 경우의 이진 탐색 트리

(b) 최적의 경우의 이진 탐색 트리

Fig. 2: 최악과 최적의 경우의 이진 탐색 트리

하지만 Figure 2a의 경우처럼 이진 탐색 트리의 균형이 맞지 않을 경우 최대 n개의 노드를 순회해야 하므로 삽입/삭제 연산의 시간 복잡도는 링크드 리스트와 같아지게 된다. 여기에서 자가 균형 이진 탐색 트리[self-balancing binary search tree]를 도입한다면 트리의 높이는 항상 $\lceil \log_2 n \rceil$가 되고, 최대 $\lceil \log_2 n \rceil$개의 노드만 순회하게 되므로 평균 $\mathcal{O}(\log n)$의 시간으로 원소를 삽입 및 삭제할 수 있게 된다.

자가 균형 이진 탐색 트리에는 AVL 트리, red-black 트리 등이 존재한다. C++ STL의 정렬 리스트 자료구조인 `std::set`는 일반적으로 red-black 트리로 구현되어 있으므로 여기서도 red-black 트리(이하 RB 트리)를 이용한다.

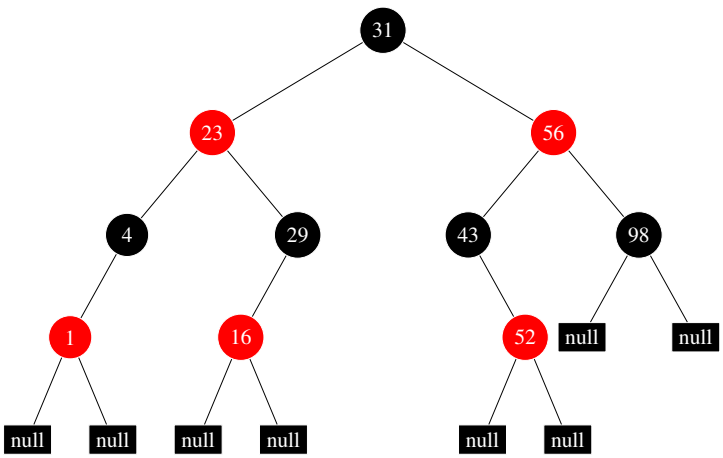


Fig. 3: Red-black 트리의 예시

RB 트리는 이진 탐색 트리에 더해 다음과 같은 성질들이 추가로 적용되는 트리이다.

- 노드는 null일 수 있다.
- 각 노드의 색은 빨강, 검정 중 하나이다. 루트 노드나 null 노드는 항상 검정색이다.
- 어떤 노드가 빨강색일 경우 자식 노드는 모두 검정색이다.
- 임의의 노드에서 리프 노드까지의 모든 경로에는 같은 수의 검정 노드가 있다.

루트에서 리프 노드까지의 최단 경로가 모두 검정 노드로만 구성되어 있다고 했을 때, 최장 경로는 검정 노드와 빨강 노드가 번갈아 나오는 경로가 된다. 임의의 노드에서 리프 노드까지의 모든 경로에는 같은 수의 검정 노드가 있다고 했고, 빨강 노드의 자식은 무조건 검정 노드이므로 모든 경로에 대해 최장 경로의 거리는 최단 경로의 거리의 두 배 이상이 될 수 없게 된다.

각 노드마다 서브트리의 크기를 전처리해 추가로 저장해 주도록 한다.

삽입, 삭제 연산을 다루기 전에 우선 RB 트리의 성질들을 만족하게 하기 위해 연결 구조를 바꾸는 연산들을 정의한다. 모두 $\mathcal{O}(1)$ 연산들이다.

BST-LEFT-ROTATE: 노드 x의 위치로 x의 오른쪽 자식을 옮긴다.

BST-LEFT-ROTATE(x)

```
1  if x has left child
2      return
3  y = x.right
4  x.size = x.left.size + y.left.size + 1
5  y.size = x.size + y.right.size + 1
6  x.right = y.left
7  if y.left ≠ NULL
8      y.left.parent = x
9  y.parent = y.parent
10 if x.parent == NULL
11     root = y
12 elseif x == x.parent.left
13     x.parent.left = y
14 else
15     x.parent.right = y
16 y.left = x
17 x.parent = y
```

BST-RIGHT-ROTATE: 노드 x의 위치로 x의 왼쪽 자식을 옮긴다.

BST-RIGHT-ROTATE(x)

```
1  if x has right child
2      return
3  y = x.left
4  x.size = x.right.size + y.right.size + 1
5  y.size = x.size + y.left.size + 1
6  x.left = y.right
7  if y.right ≠ NULL
8      y.right.parent = x
9  y.parent = y.parent
10 if x.parent == NULL
11     root = y
12 elseif x == x.parent.right
13     x.parent.right = y
14 else
15     x.parent.left = y
16 y.right = x
17 x.parent = y
```

BST-TRANSPLANT: u의 서브트리를 v의 서브트리로 대체한다.

```
BST-TRANSPLANT(u, v)
1  if u.parent == NULL
2      root = v
3  elseif u == u.parent.left
4      u.parent.left = v
5  else
6      u.parent.right = v
7  v.parent = u.parent
```

i번째 순위의 노드를 가져오는 연산을 추가로 정의한다. DFS와 비슷한 식으로 탐색하며, 최대 $\lceil \log_2 n \rceil$ 개의 노드만 순회하므로 시간 복잡도는 $\mathcal{O}(\log n)$이다.

BST-GET: i번째 순위의 노드를 가져온다.

```
BST-GET(u, i)
1  if i < u.left.size
2      return BST-GET(u.left, i)
3  elseif i > u.left.size
4      return BST-GET(u.right, i − u.left.size − 1)
5  else
6      return u
```

삽입 새 데이터를 갖는 빨강색 노드를 맨 밑에 삽입하고 트리의 성질에 맞도록 노드들을 재배치한다. BST-INSERT에서 탐색 과정에 최악의 경우 트리의 높이만큼 탐색하고, 또 BST-INSERT-REVALIDATE에서 최악의 경우 루트로 두 칸씩 되돌아면서 $\frac{\text{트리의 높이}}{2}$만큼 탐색하므로 시간 복잡도는 $\mathcal{O}(\log n)$이다.

```
BST-INSERT(u)
 1  y = NULL
 2  x = root
 3  u.size = 1
 4  while x ≠ NULL
 5      x.size = x.size + 1
 6      y = x
 7      if u.key < x.key
 8          x = x.left
 9      else
10          x = x.right
11  u.parent = y
12  if y == NULL
13      root = u
14  elseif u.key < y.key
15      y.left = u
16  else
17      y.right = u
18  u.left = NULL, u.right = NULL, u.color = RED
19  BST-INSERT-REVALIDATE(u)
20  size = size + 1
```

BST-INSERT-REVALIDATE(u)

```
1  while u.color == RED
2      if u.parent == u.parent.parent.right
3          v = u.parent.parent.right
4          if v.color == RED
5              u.parent.color = BLACK
6              v.color = BLACK
7              u.parent.parent.color = RED
8              u = u.parent.parent
9          else
10             if u == u.parent.right
11                 u = u.parent
12                 BST-LEFT-ROTATE(u)
13             u.parent.color = BLACK
14             u.parent.parent.color = RED
15             BST-RIGHT-ROTATE(u.parent.parent)
16     else
17         v = u.parent.parent.left
18         if v.color == RED
19             u.parent.color = BLACK
20             v.color = BLACK
21             u.parent.parent.color = RED
22             u = u.parent.parent
23         else
24             if u == u.parent.left
25                 u = u.parent
26                 BST-RIGHT-ROTATE(u)
27             u.parent.color = BLACK
28             u.parent.parent.color = RED
29             BST-LEFT-ROTATE(u.parent.parent)
30 root.color = BLACK
```

삭제 BST-GET를 이용해 노드를 가져오고 일반적인 트리에서 원소를 삭제하듯이 삭제한다. BST-DELETE 탐색 과정은 BST-GET과 같은 시간이 걸리고, BST-DELETE-REVALIDATE에서 최악의 경우 마찬가지로 루트로 두 칸씩 되돌아면서 $\frac{\text{트리의 높이}}{2}$만큼 탐색하므로 시간 복잡도는 $\mathcal{O}(\log n)$이다.

```
BST-DELETE(i)
 1  u = BST-GET(root, i)
 2  y = u
 3  originalColor = y.color
 4  if u.left == NULL
 5      x = u.right
 6      BST-TRANSPLANT(u, u.right)
 7  elseif u.right == NULL
 8      x = u.left
 9      BST-TRANSPLANT(u, u.left)
10  else
11      y = (node with minimum index in u.right)
12      x = y.right
13      if y.parent == u
14          x.parent = y
15      else
16          BST-TRANSPLANT(y, y.right)
17          y.right = u.right
18          y.right.parent = y
19      BST-TRANSPLANT(u, y)
20      y.left = u.left, y.parent = y, y.color = u.color
21      if originalColor == BLACK
22          BST-DELETE-REVALIDATE(x)
```

BST-DELETE-REVALIDATE(x)

```
1  while x ≠ root and x.color == BLACK
2      if x == x.parent.left
3          w = x.parent.right
4          if w.color == RED
5              w.color = BLACK, w.parent.color = RED
6              BST-LEFT-ROTATE(x.parent)
7              w = x.parent.right
8          if w.left.color == BLACK and w.right.color == BLACK
9              w.color = RED
10             x = x.parent
11         else
12             if w.right.color == BLACK
13                 w.left.color = BLACK, w.color = RED
14                 BST-RIGHT-ROTATE(w)
15                 w = x.parent.right
16             w.color = w.parent.color
17             x.parent.color = BLACK, w.right.color = BLACK
18             BST-LEFT-ROTATE(x.parent)
19             x = root
20     else
21         w = x.parent.left
22         if w.color == RED
23             w.color = BLACK, w.parent.color = RED
24             BST-RIGHT-ROTATE(x.parent)
25             w = x.parent.left
26         if w.right.color == BLACK and w.left.color == BLACK
27             w.color = RED
28             x = x.parent
29         else
30             if w.left.color == BLACK
31                 w.right.color = BLACK, w.color = RED
32                 BST-LEFT-ROTATE(w)
33                 w = x.parent.left
34             w.color = w.parent.color
35             x.parent.color = BLACK, w.left.color = BLACK
36             BST-RIGHT-ROTATE(x.parent)
37             x = root
38 x.color = BLACK
```

특정 범위의 원소 쿼리 l번째부터 r번째까지의 원소를 순회하도록 한다. 중위 순회한 결과값의 l부터 r번째 원소를 반환하면 된다. 시간 복잡도는 $\mathcal{O}(r)$아다. 다만 모든 l부터 r번째 인덱스마다 $\mathcal{O}(\log n)$의 BST-GET을 호출하는 경우 시간 복잡도는 $\mathcal{O}((r-l)\log n)$이므로 r보다 $(r-l+1)\log n$이 현저히 작을 경우 이와 같은 방법으로 최적화할 수 있어 보인다. 아래 의사 코드는 최적화가 이뤄져 있지는 않다.

BST-QUERY(l, r)

```
1  if (1 ≤ l ≤ r ≤ size) ≠ TRUE
2      error Array index out of range
3  Allocate new stack s
4  curr = root
5  while curr ≠ NULL
6      Push curr to s
7      curr = curr.left
8  for i = 1 to r
9      if l ≤ i
10         curr = pop element from s
11         Add curr to query result
12     else
13         Pop element from s
14     curr = curr.right
15     while curr ≠ NULL
16         Push curr to s
17         curr = curr.left
```

BST-GET의 공간 복잡도는 스택의 크기, 즉 $\mathcal{O}(r)$아다.

보고서에서 언급한 다른 모든 함수들의 공간 복잡도는 $\mathcal{O}(1)$이다.

참고문헌

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*. Cambridge, MA: The MIT Press, 2009, pp. 287-338.