

HW #3: Concurrent Data Structures 개발 & 디버깅

2015004857 이영수

1. BST 에 구현한 fine-grained lock 에 대한 설명

1.1. Insert Operation

기존 코드의 insert-operation 은 *recursive* 하게 개발되어 있었다. 하지만 *recursive* 한 방법을 사용하면 fine-grained lock 을 구현하기 매우 복잡하기에 우선 *iterative* 하게 메시지를 수정하였다. 이후 lock 을 구현하였는데, 이의 flow 는 아래와 같다.

- ① 트리에 lock 을 걸음
- ② *root* 가 null 이라면, 새 *root* 노드를 추가하고, 트리의 lock 을 해제한 후 종료
- ③ 그렇지 않으면 *root* 를 *cur* 노드로 설정하고, *cur* 에 lock 을 건 후 트리의 lock 을 해제
- ④ 삽입하려는 *data* 가 *cur.data* 보다 작으면 *next* 를 *cur.left* 로, 크면 *cur.right* 로 설정
- ⑤ *next* 가 null 이라면 그 위치에 새로운 노드를 삽입하고 종료
- ⑥ 그렇지 않으면 *next* 에 lock 을 걸고, *cur* 의 lock 을 해제한 뒤 *cur* 에 *next* 노드를 대입하고 ④부터 반복

1.2. Delete Operation

Insert Operation 과 비슷하게 *iterative* 하게 변경한 후 lock 을 구현하였고, flow 는 아래와 같다.

- ① 트리에 lock 을 걸음
- ② *root* 가 null 이라면, 트리의 lock 을 해제하고 종료 (실패 반환)
- ③ 그렇지 않으면 *root* 에 우선 lock 을 걸음
- ④ 만약 *root.data* 가 toDelete 라면 *root* 의 *boundaryNode*^{1.2.1}를 찾아서 *root* 에 대입하고, 기존 *root* 의 lock 과 트리의 lock 을 해제하고 종료
- ⑤ 그렇지 않으면 우선 *root* 를 *parent* 로 설정하고, toDelete 가 *root.data* 보다 작으면 *cur* 를 *root.left* 로, 크면 *root.right* 로 설정하고 *cur* 에 lock 을 걸음
- ⑥ 이후 *cur.data* 가 toDelete 와 일치할 때 까지 탐색 반복
- ⑦ *cur.data* 가 toDelete 와 일치한다면 *cur* 의 *boundaryNode*^{1.2.1}를 찾아서 *tmp* 로 둔 뒤, *cur* 위치에 *cur* 대신 *tmp* 를 삽입하고 (*cur* 노드 삭제), *cur* 와 *parent* 의 lock 을 해제한 뒤 종료 (성공 반환)
- ⑧ 그렇지 않으면 *parent* 의 lock 을 해제하고 (2 개 이전 노드), *cur* 을 *parent* 에

대입한 뒤, toDelete 가 *parent.data* 보다 작으면 *cur* 를 *parent.left* 로, 크면 *parent.right* 로 설정

- ⑨ 이 때 *cur* 이 null 이라면 노드를 찾지 못한 것임으로 종료(실패 반환)
- ⑩ 그렇지 않으면 *cur* 에 lock 을 걸고, ⑥ 으로 돌아가 작업 반복

1.2.1. boundaryNode 함수

기존 RetrieveData 함수를 변형 한 것으로, 삭제 할 노드를 대체할 수 있는 노드를 반환 하는 함수이다. (대체해도 정렬 상태가 깨지지 않는) 이 프로그램에서는 origin 노드의 왼쪽 자식의 최 우측 자식 노드를 반환하도록 되어 있으며, 이 함수도 마찬가지로 fine-grained lock 이 구현되어 있다.

1.3. Search Operation

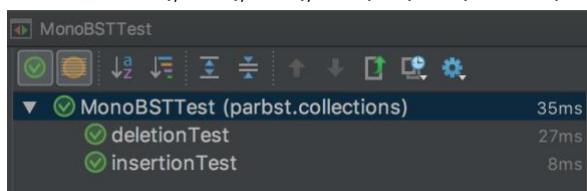
마찬가지로 iterative 하게 변경한 후 lock 을 구현하였다. Flow 는 아래와 같다.

- ① 트리에 lock 을 걸음
- ② *root* 가 null 이라면, 트리의 lock 을 해제한 후 종료(실패 반환)
- ③ 그렇지 않으면 *root* 를 *cur* 노드로 설정하고, *cur* 에 lock 을 건 후 트리의 lock 을 해제
- ④ 삽입하려는 data 가 *cur.data* 보다 작으면 *next* 를 *cur.left* 로, 크면 *cur.right* 로 설정
- ⑤ *next* 가 null 이라면 *cur* 의 lock 을 풀고 종료 (실패 반환)
- ⑥ 그렇지 않으면 *next* 에 lock 을 걸고, *cur* 의 lock 을 해제한 뒤 *cur* 에 *next* 노드를 대입하고 ④부터 반복

2. 테스트 방법

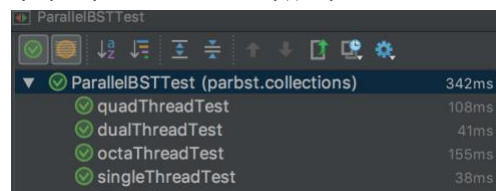
JUnit 으로 테스트 코드를 작성하고, search 함수와 함께 insert 와 delete 를 테스트하여 올바르게 동작함을 확인하였다.

Parallel 을 지원하는 BST 와 지원하지 않는 BST 를 각각 테스트하였으며, Parallel 의 경우 thread 를 1 개, 2 개, 4 개, 8 개 사용하는 경우에 각각 테스트를 진행하였다.



MonoBSTTest	
▼ MonoBSTTest (parbst.collections)	35ms
✓ deletionTest	27ms
✓ insertionTest	8ms

그림 1: Non-parallel BST 테스트



ParallelBSTTest	
▼ ParallelBSTTest (parbst.collections)	342ms
✓ quadThreadTest	108ms
✓ dualThreadTest	41ms
✓ octaThreadTest	155ms
✓ singleThreadTest	38ms

그림 2: Parallel BST 테스트

3. 성능 분석

3.1. 컴퓨터의 스펙 및 분석

- Macbook Pro (13-inch, 2016)
- Intel Core i5 (2.9 GHz)
- 프로세서 1 개, 총 코어 2 개
- 메모리: 8GB
- L2 캐시: 256KB
- L3 캐시: 4MB

코어 수가 적으므로 병렬 처리의 효율이 극대화 되지는 않을 것이다. 성능이 좋은 컴퓨터가 아닌, 모바일 CPU 등을 탑재한 랩탑을 이용했으므로 lock overhead로 인한 손해가 multithreading으로 인한 성능 향상보다 클 수 있다.

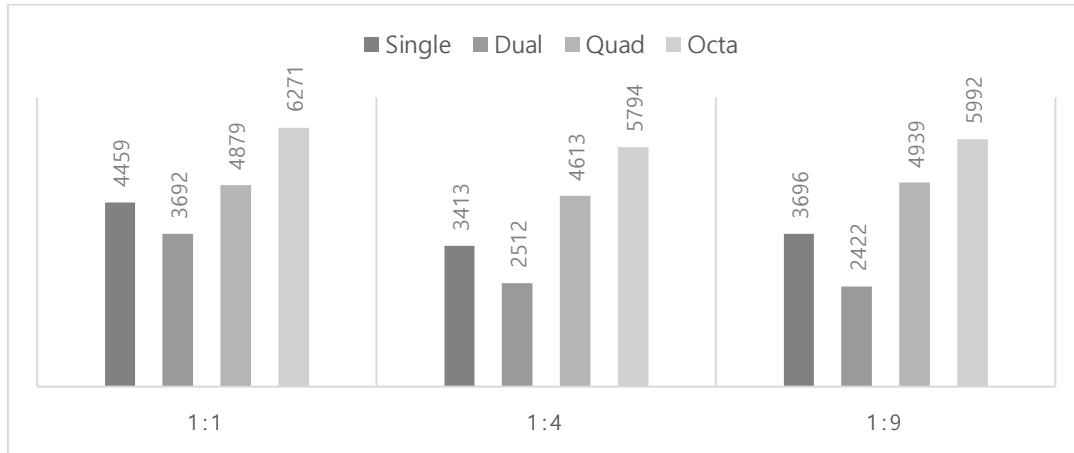
3.2. (a) 100 만개의 랜덤한 숫자 insertion

성능테스트는 BSTPerformanceTest 라는 클래스를 만들고 JUnit 을 통해 진행하였다. 각 테스트는 같은 초기에 생성된 랜덤한 데이터 셋을 공유하여 같은 환경에서 진행되었다.



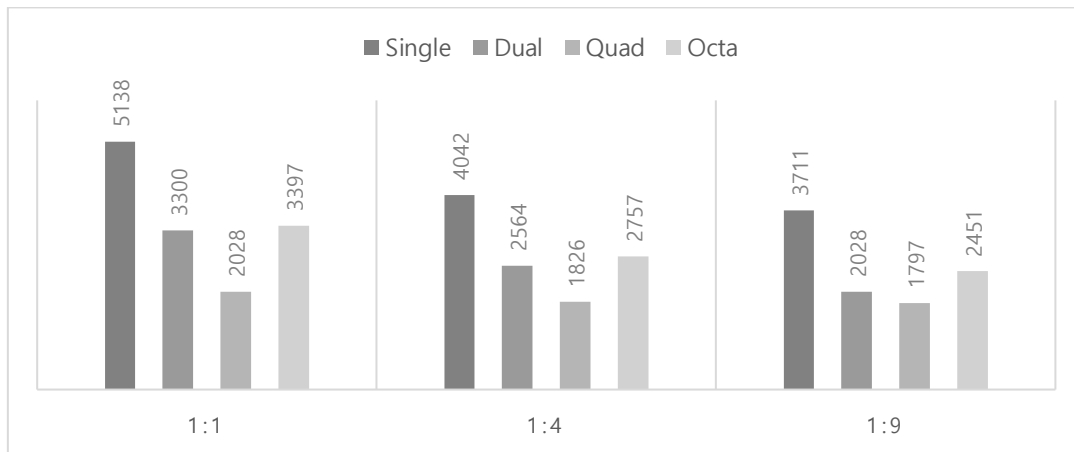
Single Thread 보다 Dual 이 빠르긴 하지만 Quad 나 Octa 는 오히려 실행시간이 느리다. PC 의 성능상 4 개 이상의 thread 를 사용하게 될 경우 lock overhead 가 더 커짐을 추측할 수 있다.

3.3. (b) 100 만개의 랜덤한 숫자 insert 후 추가 100 만개 insert/search



전체적으로 (a)와 비슷한 성능을 보이며, search 연산이 많을수록 2 개의 thread 를 사용했을 때 더 성능 향상이 증가됨을 볼 수 있었다. 4 개나 8 의 thread 를 사용한 경우에는 search 의 비율을 늘여도 속도에 큰 차이가 없는데, 이를 통해 insert 나 search 자체의 연산 속도보다 lock 으로 인한 overhead 가 영향을 더 끼침을 간접적으로 확인할 수 있다.

3.4. (c) ReadWriteLock 사용 시의 성능



ReadWriteLock 으로 변경 이후 (b)와 성능이 크게 차이가 남을 확인할 수 있다. 특히 4 개 thread 와 8 개 thread 사용 시의 성능이 크게 향상되었다. 또한 (b)에서는 insert 보다 search 의 횟수를 늘여도 속도의 큰 차이가 없었던 반면, *ReadWriteLock* 사용시에는 search 가 많아질 수록 전체적으로 성능 향상이 이루어졌다

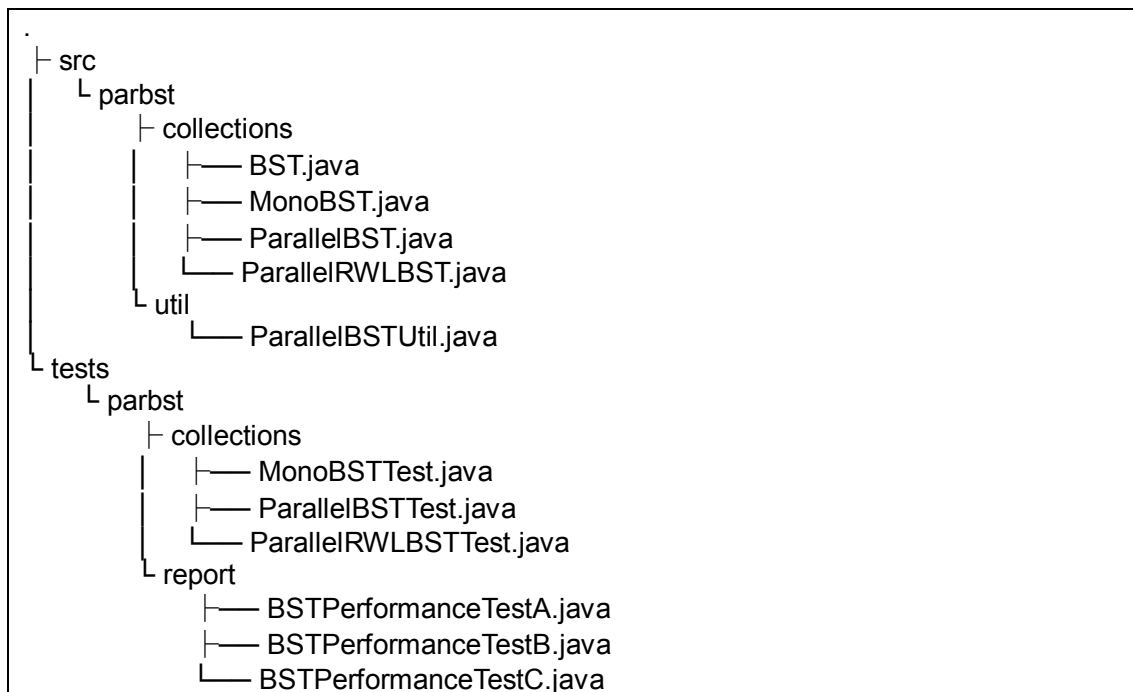
다만 Single thread 의 경우에는 오히려 전체적인 성능이 저하되었는데, 이는 *ReadWriteLock* 을 사용하며 생긴 오버헤드가 기존 오버헤드보다 크기 때문에 발생 한 것으로 문제로 보인다.

4. 기타

4.1. 프로그램 설계

개발은 주어진 BST 코드를 활용하고 이 자료구조를 병렬처리가 가능한 형태로 개선하는 식으로 진행하였는데, 이에 따라 일반적인 BST 에 대한 인터페이스를 만들고, 병렬을 지원하는 ParallelBST와 ReadWriteLock 을 사용한 ParallelRWLBST, 그리고 병렬을 지원하지 않는 MonoBST 클래스를 나누도록 설계하였다.

각각의 클래스는 동작에 대한 보증을 위해 테스트 코드를 작성하였고, 성능 측정을 위해서 별도의 패키지에 테스트를 작성한 상태이다.



4.2. Part 2

Part2 는 시간상의 문제로 진행하지 못하였다.