서약서: 나는 본 과제를 수행함에 있어서 허가되지 않은 도움을 주거나 받지 않았음을 서약합니다.			
분반:42	학번:2016312761	이름:여혁수	(서명)여혁수

기말 과제 리포트 [LRU 캐시의 이해]

1. 개요) LRU 분석에서는 LRU 캐시가 무엇인지와 hash table의 구조, 역할에 대해 설명하고 LRU 동작원리와 성능향상의 조건에 대해 얘기합니다. LRU 설계에서는 제가 생각해본 hash table을 사용하지 않는 방법을 설명하고, 제일 괜찮아보이는 방법과, 그 이유를 얘기합니다. 구현 설명 및 비교 분석에서는 알고리즘을 설명하고 hash table을 사용하는 방법과 런타임을 비교합니다.

2. LRU 분석) LRU 캐시는 크게 doubly linked list와 hash table로 구성되어 있다. Doubly linked list는 접근하려는 캐시 엔트리를 찾고 편하게 list의 가장 앞 부분으로 옮길 수 있고, 제거해야하는 경우에도 편하게 할 수 있어 사용된다. Hash table은 찾고자 하는 엔트리의 존재 여부를 한 번에 알기 위해 사용된다. LRU 캐시 구현 시에 hash table에 들어가는 원소의 형태는 {key, value} 순서쌍이다. 어떤 원소를 hash table에서 찾던, 새롭게 넣으려고 하던, 그 원소의 key값을 어떤 hash function으로 변형해서 나온 값이 hash table의 인덱스가 되고, 그 인덱스에 해당하는 칸에 데이터를 저장하거나 있던 데이터를 빼오는 방식으로 사용된다.

LRU 캐시는 캐시에 어떤 엔트리를 추가하고, 캐시에서 엔트리를 얻어오고, 캐시가 꽉 차서 어떤 엔트리를 삭제하고, 이렇게 크게 세 동작이 있다. 엔트리를 추가할 때, 일단 넣고자 하는 엔트리는 가장 최근에 접근한 것이므로 list의 가장 앞 부분을 가리키는 head라는 포인터가 가리키게 해서 list의 가장 앞에 추가한다. 또한 엔트리의 key 값으로 hash table에 접근해서 매핑되는 곳에 마찬가지로 엔트리를 추가해준다. 캐시에서 어떤 엔트리를 얻어오려면 hash table에서 찾고자 하는 엔트리의 key값으로 엔트리를 찾는다. 그 엔트리는 가장 최근에 접근한 엔트리가 되므로 doubly linked list에서 head가 그 엔트리를 가리키도록 해서 list의 가장 앞에 위치하게 한다. 이러한 작업이 반복되면 접근된 시간 순서대로 엔트리들이 list에 위치할 것이다. 캐시가 꽉 차서 엔트리를 삭제하는 경우에 가장 오랫동안 접근하지 않은 엔트리를 제거한다. list의 마지막을 가리키는 포인터에 담겨 있는 엔트리를 제거하면 된다. 그리고 그 엔트리의 key값으로 hash table에도 접근해서 해당 엔트리를 제거한다.

캐시의 성능은 일단 접근하려는 엔트리가 캐시에 있다면 그 엔트리를 빠르게 가져올수록 더 좋다고 말할 수 있고, 또 엔트리가 캐시에 없어서 하위의 캐시나 메모리에서 엔트리를 찾아야하는 그런 경우가 잘 없을수록 성능이 좋다고 말할 수 있다. 캐시의 용량을 줄이면 탐색해야하는 엔트리 수가 줄어들기 때문에 평균적으로 캐시에서 엔트리를 탐색하고 가져오는 시간이 빨라질 것이다. 하지만 엔트리가 캐시에 없는 경우를 줄이려면 근본적으로 캐시 용량을 늘려야 할 것이

다. 그러므로 캐시의 용량을 적절하게 설정하는 것이 성능 향상을 위한 조건이라고 할 수 있다.

3. LRU 설계) 첫번째 방법은 오직 배열만 사용하고 timestamp라는 각 엔트리에 접근된 시간을 비교하기 위한 변수를 이용하는 것이다. 우선 엔트리를 추가할 때는 단순히 배열의 마지막 칸에 값을 추가한다. 추가할 때나 필요한 엔트리를 가져와야 할 때마다 timestamp는 계속 1씩 증가하면서 그 엔트리의 timestamp에 overwrite된다. 엔트리를 제거해야할 때 timestamp 값이 가장 작은 엔트리를 찾아서 제거 후 그 배열 칸에 추가할 엔트리 값들을 넣는다. 엔트리를 가져올 때는 순차적으로 key값을 찾고 그 key와 짝인 value값을 리턴하면 된다. 이 방법의 장점은 공간을 적게 사용한다는 것이다. 캐시에 timestamp만 추가한 것이므로 캐시 사이즈 만큼의 배열 row만 있으면 되고, 각 row에는 key, value, timestamp가 있으니 3*sizeof(int)*n의 공간을 사용할 것이다. 공간을 덜 쓰면 캐쉬를 그만큼 더 크게 만들어서 miss rate를 줄일 수 있다. 단점은 순차적 탐색을 해서 O(n)이 걸리는 동작이 있다는 것이다.

두번째 방법은 3개의 doubly linked list를 사용하는 것이다. 안 배운 자료구조라고 말할 순 없지만 segregated list라고 부른다. 엔트리를 추가할 때는 list의 맨 앞 부분에 추가하여 가장 최근에 접근한 엔트리임을 나타내주는데 key값의 끝자리를 3으로 나눈 나머지가 0이냐 1이냐 2이냐에 따라 다른 list로 추가된다. 그리고 첫번째 방법과 같은 역할인 timestamp 변수를 추가했다. 엔트리를 제거해야할 때 3개의 list의 가장 뒤 엔트리 중에 timestamp가 가장 작은 엔트리를 제거한다. 엔트리를 가져올 때는 list를 순차적으로 살펴봐야 하지만 탐색범위가 총 엔트리의 1/3로 줄었다고 할 수 있다. 장점은 한 종류의 자료구조만 사용해서 구조가 단순하고 공간을 적게 사용한다. 단점은 timestamp가 int범위를 넘어가게 되면 timestamp가 제 역할을 하지 못하게 되므로 별도로 값을 정리하는 알고리즘이 추가되어야한다는 것이다.

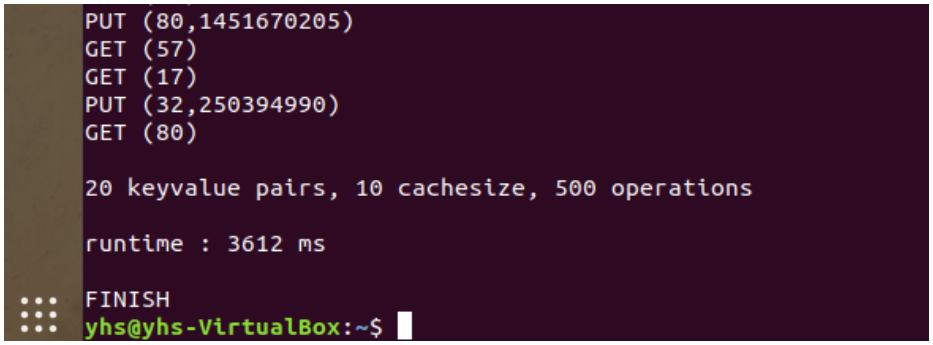
마지막은 heap과 doubly linked list를 쓰는 것인데, 기존 doubly linked list가 제거해야할 엔트리를 알기 위해 존재하고, heap이 추가되었다. heap에는 각 엔트리의 key, value가 별도로 저장되어 있다. Key를 기준으로 heap을 구성하는데 모든 엔트리의 key 중 중간 값이 root가 되고 root의 left child는 root보다 더 큰 key값이 모여있고 min heap으로 만든다. 반대로 right child는 root보다 더 작은 key가 모여 있고 max heap으로 만든다. 엔트리를 추가할 때는 heap의 리프로 추가하고 heapify를 진행한다. 만약 엔트리를 추가하려는데 캐시가 꽉 찼다면 제거할 엔트리의 key값을 heap에서 binary search로 찾아서 제거하고, 새로운 엔트리를 추가하고 그에 맞게 heapify를 해준다. 어떤 엔트리를 캐시에서 가져오는 경우에는 그 엔트리의 key값으로 heap을 탐색한다. Root의 key부터 보면서 찾는 key가 더 크면 left child로, 더 작으면 right child로 간다. 그리고 그 엔트리의 key가 찾는 key가 아니면 다시 left child부터 key값을 비교해서 찾을 때까지 하위 레벨로 간다. 예를 들어 Max heap안에서 탐색하는 경우라면 leftchild보다 찾는 key가 작거나 같으면 leftchild로 내려가고, 아니라면 rightchild로 내려간다. 이 방법의 장점은 뭐든 heap에서 찾기 때문에 시간복잡도가 O(logn)으로 비교적 월등하다는 것이다. 단점은 알고리즘이 복잡하다는 것이다. 엔트리 추

가를 위해 heap상에서 중간 레벨의 어떤 엔트리를 제거해야 한다면 그 엔트리의 key값을 엔트리가 위치했던 자리의 parent 엔트리, 2개의 child 엔트리와 비교해서 그에 맞게 heapify가 수행되고 또 추가를 위해 다시 리프 엔트리로 추가하고 heapify가 수행되어야 할 것이다.

저는 두번째 방법을 선택했습니다. 왠지 모르게 정말 빨라서 선택했습니다. 일단 첫번째 방법은 캐시에서 엔트리를 교체해야할 때 O(n)만큼 탐색을 해야하는데, 두번째 방법은 3개의 엔트리만 비교하면 교체될 엔트리를 찾을 수 있습니다. 세번째 방법이 이론상 모든 경우가 O(log n)이라 효율 면에서 좋지만 구조가 복잡해서 실제로 더 효율적으로 동작하게끔 구현하기가 힘들었습니다. 정말 구현을 잘해서 어느 정도 최적화가 된다면 세번째 방법이 제일 좋을 것 같습니다.

4. 구현 설명 및 성능 비교 분석) 알고리즘을 위에서 말한 것에서 자세하게 설명하자면, 일단 doubly linked list 3개가 있고, 엔트리의 key값을 3으로 나눈 나머지가 0,1,2 중 하나 일텐데, 그 값에 따라 다른 linked list로 들어가고, 물론 맨 앞 칸에 들어갑니다. 3개 list에 있는 엔트리의 수가 cache size와 같게 되면 엔트리를 추가할 때 대신 제거될 엔트리를 선택하기 위해 3개의 linked list 각각의 맨 뒤 엔트리의 timestamp값을 보고 가장 작은 값을 가진 엔트리를 제거합니다. 어려웠던 점은 딱히 없었습니다. 해쉬 테이블을 사용했을 때와 성능 비교를 위해 인풋의 경우를 크게 둘로 나눴습니다. 하나는 캐시 사이즈가 충분하지 않아 엔트리 교체가 계속해서 발생하는 경우고, 하나는 캐시 사이즈가 충분해서 교체가 안 일어나는 경우입니다. 전자의 경우는 존재하는 key,value 쌍을 캐시 사이즈의 2배만큼 생성했고, 명령의 수는 캐시 사이즈의 20~100배로 설정했습니다. 후자의 경우는 존재하는 key,value 쌍과 명령의 수 모두 캐시 사이즈의 1/2로 설정했습니다. 런타임은 time.h로 clock()함수를 받아와서 millisec 단위로 측정해보았습니다. 출력된 값을 보니까 단위가 millisec 아닌 것 같았습니다. 분명히 1초 안에는 끝났는데 4000, 5000이 떠서.. 수치로만 봐주시면 될 것 같습니다.

테스트 결과 분석: 캐시 사이즈가 충분하지 않아서 교체가 많이 일어나면 해쉬테이블을 사용한 경우는 한 번에 제거할 엔트리를 찾으니까 3개의 엔트리를 비교해서 제거할 엔트리를 결정하는 저의 알고리즘에 비해 더 빠릅니다. 명령의 수가 많아질수록 차이가 커지는 것을 볼 수 있습니다. 캐시 사이즈가 충분하면 교체가 일어나지 않고, 그런 경우에는 속도가 비슷한 것을 볼 수 있었습니다. 엔트리 추가 시에 해쉬테이블이 있으면 두 곳에 추가를 해줘야하는데 저의 알고리즘은 그냥 doubly linked list에만 추가하면 끝이어서 이런 결과가 나온 것 같습니다. Hash table쓰지 않는 것에서 점수가 잘 안나왔습니다. 기능적으로는 문제가 없어보이는데 Cache capacity가 3으로 나누어서 분배되었다는 부분에서 오류가 있는 것 같습니다.



<결과> i) 캐시사이즈 충분 X :

(key, value) pairs	Cache size	Number of operations	Using hash table algorithm	My segregated list algorithm
100	50	1000	9300	9900
50	25	500	3700	4500
20	10	200	1200	1300
20	10	500	3600	4400
20	10	1000	7600	9600

ii) 캐시사이즈 충분 :

(key, value) pairs	Cache size	Number of operations	Using hash table	My segregated list
200	400	200	1600	1600
100	200	100	840	830
50	100	50	480	500

5. 비고) --이번학기 고생 많으셨습니다 교수님 조교님들. 행복하세요!