

**(컴퓨터구조) 과제 #3**

**201312845 김일식 | 201511183김영서 | 201511237 허승리**

**제출일 : 2016년 12월 11일**

|  |
| --- |
| (컴퓨터 구조) 과제 #2 |
| 3조 |

|  |
| --- |
| 1) 문제 정의 |

LRU, FIFO, LFU 및 random 방식의 4가지 교체 알고리즘을 구현하고 성능을 분석하시오. 사용자로부터 임의의 데이터 세트 (양의 정수)를 입력 받은 후, 각 알고리즘을 수행한 결과, 수행 시간 및 적중률을 비교 분석한다. 입력 데이터의 개수와 세트 당 슬롯의 개수는 사용자가 프로그램을 실행하면서 임의로 선택할 수 있도록 구현한다.

- 사용자가 입력하는 데이터 세트는 모두 양의 정수이다.

- 각 알고리즘은 독립된 함수로 구현하여야 하며, 각 함수는 적중률을 반환한다.

|  |
| --- |
| 2) 알고리즘 별 주요 함수 및 변수 |

**void initialization(vector<int> & c, vector<int> & cnt, vector<int> & d, int & sn)**

모든 알고리즘에 앞서 각 알고리즘에 쓰일 모든 변수를 초기화한다. vector<int> c는 캐쉬에 쓰일 자료구조이다. vector<int> cnt는 LRU, LFU 알고리즘에서 각 변수의 카운터 배열로 쓰일 자료구조이다. vector<int> d는 데이터셋을 담을 배열로 쓰일 자료구조이다. int sn은 슬롯 개수를 저장하는 변수이다. 이를 통해서 각 자료구조를 초기화한다. c.assign(sn, -1)은 -1 값으로 sn 만큼 c의 방을 만들고 초기화한다. 마찬가지로 cnt.assign(sn, 0)은 cnt 배열을 0으로 초기화하고 sn개의 방을 만든다. dn은 데이터 셋의 개수를 담는 변수이다. d.assign(dn, 0)은 vector d를 dn개의 방을 만들고 0으로 초기화한다. 그리고 데이터 셋을 입력받아 vector d에 입력받은 데이터 값들을 저장한다.

**double random(vector<int> & cache, vector<int> & data)**

랜덤 알고리즘을 구현한 함수이다. vector<int> cache는 캐쉬로 쓸 자료구조이다. 슬롯의개수만큼 미리 방이 할당되어 있고 각 방은 -1로 초기화 되어 있다. vector<int> data는 데이터셋을 담고 있는 자료구조이다. 이미 initialization() 함수에서 데이터 셋을 입력 받았으므로 data에 데이터 셋이 담겨 있다. int hit는 적중된 데이터의 개수이다. 이를 데이터의 개수로 나누면 적중률을 계산할 수 있다. int cNum은 슬롯이 얼마나 찼는지를 나타낸다. cNum이 슬롯의 개수와 같다면 슬롯이 다 찬 것이다. bool isHit은 적중되었는지 아닌지를 나타낸다. 적중 되었다면 true가 된다. 알고리즘은 다음과 같다. 먼저 데이터의 개수 만큼 반복하며 슬롯에 현재 찾고자 하는 데이터와 같은 데이터가 있는지 확인한다. 있으면 적중이므로 hit변수를 하나 증가시키고 isHit를 true로 만든다. 데이터가 슬롯에 없다면 miss이므로 데이터를 슬롯에 추가하여야 한다. 슬롯이 비었다면 데이터는 그냥 삽입된다. 슬롯이 꽉 차 있다면 랜덤하게 슬롯을 선택하여 데이터를 넣는다. 모든 데이터 셋에 대하여 알고리즘을 수행하면 적중률을 리턴한다.

**double LFU(deque<int> & cache, vector<int> & count, vector<int> & data)**

LFU 알고리즘을 구현한 함수이다. deque<int> cache는 캐쉬로 쓸 자료구조이다. LFU 알고리즘에서는 두 슬롯의 카운트가 같다면 FIFO형식으로 구현되므로 큐 형태의 자료구조를 사용한다. vector<int> count는 얼마나 참조되었는지를 카운트하는 배열로 쓸 자료구조이다. vector<int> data는 데이터셋을 담은 자료구조이다. int hit는 적중한 횟수를 저장한다. Int cNum은 슬롯이 얼마나 찼는지를 나타낸다. cNum이 슬롯의 개수와 같다면 슬롯이 다 찬 것이다. bool isHit은 적중되었는지 아닌지를 나타낸다. 적중 되었다면 true가 된다. Int minIdx는 참조 카운트가 최소인 방의 번호를 저장하는 용도이다. deque<int>::iterator it는 큐의 방 전체를 둘러보며 적중인지 아닌지를 확인할 때 쓰는 이터레이터이다. 알고리즘은 다음과 같다. 먼저 데이터의 개수 만큼 반복하며 슬롯에 현재 찾고자 하는 데이터와 같은 데이터가 있는지 확인한다. 있으면 적중이므로 hit변수를 하나 증가시키고 isHit를 true로 만든다. 그리고 데이터가 적중되면서 참조 카운트가 하나 증가한다. 데이터가 슬롯에 없다면 miss이므로 데이터를 슬롯에 추가하여야 한다. 슬롯이 비었다면 데이터는 그냥 삽입된다. 슬롯이 꽉 차 있다면 가장 적게 참조된 데이터를 카운트 배열을 참조하여 인덱스를 구한다. 그리고 데이터를 새로운 데이터로 대체한 후에 카운트를 1로 초기화한다. 모든 데이터 셋에 대하여 알고리즘을 수행하면 적중률을 리턴한다.

**double FIFO(deque<int> & cache, vector<int> & data, int slotNum)**

FIFO 알고리즘을 구현한 함수이다. deque<int> cache는 캐쉬로 쓸 자료구조이다. FIFO 알고리즘에서는 자료구조가 FIFO형식으로 구현되므로 큐 형태의 자료구조를 사용한다. vector<int> data는 데이터셋을 담은 자료구조이다. int slotNum은 캐쉬의 슬롯 개수를 저장하는 변수이다. int hit는 적중한 횟수를 저장한다. int cNum은 슬롯이 얼마나 찼는지를 나타낸다. cNum이 슬롯의 개수와 같다면 슬롯이 다 찬 것이다. bool isHit은 적중되었는지 아닌지를 나타낸다. 적중 되었다면 true가 된다. deque<int>::iterator it는 큐의 방 전체를 둘러보며 적중인지 아닌지를 확인할 때 쓰는 이터레이터이다. 알고리즘은 다음과 같다. 먼저 데이터의 개수 만큼 반복하며 슬롯에 현재 찾고자 하는 데이터와 같은 데이터가 있는지 확인한다. 있으면 적중이므로 hit변수를 하나 증가시키고 isHit를 true로 만든다. 데이터가 슬롯에 없다면 miss이므로 데이터를 슬롯에 추가하여야 한다. 슬롯이 비었다면 데이터는 그냥 삽입된다. 슬롯이 꽉 차 있다면 가장 오래된 데이터 순으로 슬롯에서 제거되고 새로운 데이터가 큐로 삽입된다. 이는 cache.pop\_front(), cache.push\_back(data)의 형태로 구현된다. 모든 데이터 셋에 대하여 알고리즘을 수행하면 적중률을 리턴한다.

**double LRU(vector<int> & cache, vector<int> & count, vector<int> & data)**

LRU 알고리즘을 구현한 함수이다. vector<int> cache는 캐쉬로 쓸 자료구조이다. LRU 알고리즘에서는 카운트에 따라서 데이터를 교환하면 되므로 큐 형태의 자료구조를 쓸 필요가 없고 배열 형태의 자료구조를 사용한다. vector<int> count는 얼마나 참조되었는지를 카운트하는 배열로 쓸 자료구조이다. vector<int> data는 데이터셋을 담은 자료구조이다. Int hit는 적중한 횟수를 저장한다. int cNum은 슬롯이 얼마나 찼는지를 나타낸다. cNum이 슬롯의 개수와 같다면 슬롯이 다 찬 것이다. bool isHit은 적중되었는지 아닌지를 나타낸다. 적중되었다면 true가 된다. int maxIdx는 가장 참조되지 않은 데이터의 인덱스를 저장하는 변수이다. 알고리즘은 다음과 같다. 먼저 데이터의 개수만큼 반복하며 슬롯에 현재 찾고자 하는 데이터와 같은 데이터가 있는지 확인한다. 있으면 적중이므로 hit변수를 하나 증가시키고 isHit를 true로 만든다. 그리고 데이터가 적중되면서 데이터의 참조 카운트가 0으로 된다. 가장 최근에 참조되었다는 뜻이다. 데이터가 슬롯에 없다면 miss이므로 데이터를 슬롯에 추가하여야 한다. 슬롯이 비었다면 데이터는 그냥 삽입된다. 슬롯이 꽉 차 있다면 가장 카운트가 큰 슬롯, 즉 가장 오랫동안 참조되지 않은 데이터의 슬롯을 찾는다. 그 방에 데이터를 삽입하고, 카운트를 0으로 초기화한다. 나머지 슬롯의 카운터는 하나씩 증가시킨다.

|  |
| --- |
| 3) 교체 알고리즘 별 성능 결과 분석 |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#slot**  **2** | **10** | | **100** | | **1000** | | **10000** | |
| **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** |
| **LRU** | **0** | **10** | **2** | **19** | **30** | **31.7** | **239** | **19.6** |
| **FIFO** | **0** | **10** | **3** | **24** | **50** | **31.5** | **352** | **19.6** |
| **LFU** | **1** | **10** | **11** | **24** | **131** | **33.1** | **1262** | **19.7** |
| **RAND** | **0** | **20** | **2** | **23** | **7** | **22** | **71** | **19.76** |

**시간 단위 : [ms], 적중률 단위 : [%]**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#slot**  **3** | **10** | | **100** | | **1000** | | **10000** | |
| **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** |
| **LRU** | **0** | **30** | **2** | **34** | **21** | **30.9** | **247** | **30.02** |
| **FIFO** | **0** | **30** | **3** | **35** | **39** | **31.4** | **356** | **30.18** |
| **LFU** | **0** | **30** | **12** | **24** | **176** | **29.4** | **1338** | **29.86** |
| **RAND** | **0** | **10** | **1** | **30** | **8** | **28.5** | **88** | **30.58** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#slot**  **4** | **10** | | **100** | | **1000** | | **10000** | |
| **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** |
| **LRU** | **1** | **20** | **4** | **41** | **37** | **41.8** | **268** | **39.5** |
| **FIFO** | **1** | **20** | **5** | **44** | **47** | **40.3** | **408** | **40.33** |
| **LFU** | **1** | **20** | **14** | **37** | **98** | **40.7** | **1113** | **40.9** |
| **RAND** | **0** | **10** | **1** | **33** | **9** | **37.1** | **108** | **39.89** |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **#slot**  **5** | **10** | | **100** | | **1000** | | **10000** | |
| **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** | **time** | **Hit-ratio** |
| **LRU** | **0** | **40** | **3** | **52** | **24** | **49.9** | **264** | **50.09** |
| **FIFO** | **0** | **40** | **4** | **54** | **43** | **49.2** | **569** | **49.74** |
| **LFU** | **0** | **40** | **8** | **55** | **105** | **50.9** | **1101** | **50.11** |
| **RAND** | **0** | **50** | **1** | **43** | **11** | **51.6** | **93** | **49.91** |

* Date의 개수를 10, 100, 1000, 10000개로 늘려가며 수행
* Slot의 개수가 2, 3, 4, 5개일 때에 대하여 결과 분석

|  |
| --- |
| 4) 토의사항 |

수행 결과, Data의 개수가 적을 때엔 LRU, FIFO, LFU알고리즘의 각 적중률에 큰 차이가 보이지 않았지만 Data 개수를 늘려갈수록 대체적으로 LFU의 적중률이 높게 나타났다. 하지만 가장 적게 참조된 데이터의 인덱스를 구하고, 슬롯이 꽉 찼을 때에 큐의 형태로 슬롯을 교체하는 과정이 있어 수행 시간은 다른 알고리즘보다 오래 걸렸다.

위의 세 알고리즘은 논리적인 방법을 취했지만, Random 알고리즘은 이름 그대로 랜덤하게 슬롯을 선택하므로 데이터 셋이 늘어나면 다른 알고리즘에 비해 적중률이 저하될 것이라고 예상하였다. 하지만 다른 알고리즘과 적중률이 크게 차이가 나지 않았고, 오히려 수행시간은 확연히 짧았기 때문에 그렇게 비효율적이지만은 않은 알고리즘이라 생각되었다.

캐시 교체 알고리즘에 있어서 가장 중요한 부분은 적중률 이므로, 이 부분을 더 높일 수 있는 알고리즘을 생각해 보았다. 캐시에 저장된 이후로 교체되기 전까지 다시 사용되지 않는 데이터의 발생 추이를 관찰하여 주기적으로 이 데이터의 수를 카운트하는 방식으로 LFU의 ‘바로 불러온 블록이 교체될 수 있는 문제점’을 보완할 수 있을 것 같다.

|  |
| --- |
| 5) 기여도 |

김일식 (34%) : LFU, Random

김영서 (33%) : LRU

허승리 (33%) : FIFO