시스템 프로그램 프로젝트 보고서

2022315160 이수형

1. Target program 개요

교안 SP05-3에서 공부했던 “Bigram Analyzer”를 따라갔으며, 텍스트는 “Complete-Works-of-William-Shakespeare.txt”를 인터넷에서 가져왔다. 즉, 이 프로그램은 텍스트 파일을 읽고, 각 단어가 몇 번 나오는지와 유일한 bigram의 표를 만들고, bigram들을 발생 순서가 높은 것부터 차례대로 정렬하도록 나오게 하는 프로그램이다. 최적화하기 전, 첫 번째 프로그램은 initial.c이다. initial.c의 경우, 교재에 나온 점은 최대한 따라가려고 노력하였다. 하지만 교재에서 코드를 준 것은 아니기에, 주관적으로 코드를 작성해야하는 부분 때문인지 교재에 나온 것처럼 프로파일링이 되진 않았다. 이 점 또한 흥미로운 점이다. 과제에서 권장했던 것을 따라 15개 이상의 function들을 사용하였으며, 300 lines 이상인 프로그램이다.

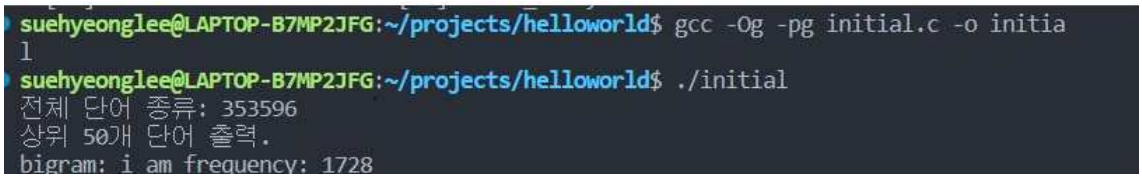
2. Target program의 컴파일 및 실행방법

Ubuntu 18.04 환경에서 진행하였다.

“gcc -Og -pg initial.c -o initial”로 컴파일하였으며, “./initial”로 실행하면 된다. 그러고서 분석을 할 때는 gprof initial을 실행하였다.

여기서 보여준 파일 이름은 initial.c이지만 낸 파일의 경우, 형식에 맞게 파일 이름이 2023-2-SP-A03-Initial-2022315160-이수형.c이다. 이러한 식으로 이름만 바뀌었다. 실행방법은 같다.

출력 값은 다음과 같다. 전체 unique한 bigram의 종류의 수를 출력한다. 그리고 bigram이 빈도가 높은 순으로 잘 정렬되어있는지 확인하기 위해 출력은 상위 50개의 빈도수를 가지고 있는 각 bigram과 frequncy(빈도수)를 뽑도록 보여주게 하였다. 아래의 사진에 다 안 보이지만 실제로 50개의 단어가 출력되었다.



전체 단어 종류: 353596
상위 50개 단어 출력.
bigram: i am frequency: 1728
bigram: of the frequency: 1681
bigram: in the frequency: 1560
bigram: i have frequency: 1504
bigram: i will frequency: 1455
bigram: to the frequency: 1350

bigram: it is frequency: 954
bigram: my lord frequency: 932
bigram: to be frequency: 886
bigram: that i frequency: 878
bigram: i do frequency: 757
bigram: and the frequency: 752
bigram: of my frequency: 677
bigram: and i frequency: 674
bigram: is the frequency: 646
bigram: you are frequency: 633
bigram: i would frequency: 627
bigram: with the frequency: 591
bigram: you have frequency: 591
bigram: he is frequency: 578
bigram: of his frequency: 540
bigram: i know frequency: 536
bigram: all the frequency: 530
bigram: is a frequency: 527
bigram: with a frequency: 527
bigram: if you frequency: 522
bigram: of a frequency: 516
bigram: as i frequency: 506
bigram: in my frequency: 495
bigram: let me frequency: 495
bigram: for the frequency: 495
bigram: shall be frequency: 495
bigram: is not frequency: 491
bigram: by the frequency: 490
bigram: the king frequency: 489
bigram: like a frequency: 476
bigram: thou art frequency: 472
bigram: to my frequency: 469
bigram: will not frequency: 467
bigram: in his frequency: 463
bigram: of this frequency: 459
bigram: do not frequency: 458
bigram: this is frequency: 457
bigram: of your frequency: 451
bigram: in a frequency: 446
bigram: if i frequency: 441
bigram: for your frequency: 439

bigram: out of frequency: 432
bigram: on the frequency: 429
bigram: from the frequency: 422

3. Target program 구조

Target program의 구조는 다음과 같이 이루어져있다. 구조체와 함수를 소개하겠다. 구조체는 다음과 같다.

1. Bigram

Bigram의 경우, char* bigram과 int frequency를 가지고 있다. 이는 두 개의 단어로 이루어진 문자열과 빈도수를 저장하기 위해 만든 것이다. 이런 Bigram은 typedef Bigram LData;를 통해 LData로 지정하였다.

2. Node

Node의 경우, LData data와 struct _node* next를 가지고 있다. 이는 Bigram을 저장하고, next를 통해 다음 Bigram을 가지고 있는 노드로 이동할 수 있다. struct _node*은 Node*과 같은 뜻이다.

3. LinkedList

LinkedList의 경우, Node* head, Node* cur, Node* before, int numOfData를 가지고 있다. 이는 노드들을 가지고 있는 연결 리스트들을 구현한 것이다. head는 연결리스트의 맨 앞 노드, cur은 연결리스트의 현재 노드(LFirst나 LNext를 통해 이동한 현재), before은 cur의 바로 전 노드이다.

numOfData의 경우, 연결리스트의 모든 노드의 개수이다.

4. Table

List tbl[MAX_TBL]과 HashFunc* hf로 이루어져있다. tbl[MAX_TBL]의 경우는 해시 태그 별로 리스트를 모아놓은 배열이다. hf는 해시 태그를 결정하는 함수이다.

구조체를 정리해보면 다음과 같다. Bigram을 갖고 있는 노드들을 모아 놓은 연결 리스트들을 해시 태그 별로 모아 놓은 테이블을 구현한 것이다. 다음은 함수를 설명하겠다.

1. main

말 그대로 main함수이다. 실제로 text를 가져와서, bigram으로 쪼갠 후 리스트 함수를 통해 hash에 맞게 mytbl의 table에 bigram과 frequency를 넣는다. 그 후, mergedList라는 리스트를 새로 만들어 여기다가 모든 bigram과 frequency를 붙여 넣는다. 그 후 정렬을 쉽게 하기 위해 arrayList라는 배열을 만든 후, mergedList의 모든 bigram과 frequency를 넣는다. 그 후 삽입 정렬을 하고 프로그램은 상위 50개를 출력한다.

2. LInsert

교재에서 첫 번째 재귀적 버전이 리스트의 끝에 넣는다고 되어있어 LInsert를 구현해 연결리스트의 경우, 뒤에 넣는 것으로 구현했다.

3. LFirst

리스트의 첫 번째 node의 LData의 주소값을 준다.

4. LNext

리스트의 cur이 가르키고 있는 다음의 node의 LData의 주소값을 준다.

5. list

list의 경우, table에서 TBLSearch를 통해 그 bigram이 있으면 그냥 그 bigram의

frequency를 1더해주고, 만약 없다면 새로운 노드로 넣는 것으로 구성했다.

6. MergedAllLists

MergedAllLists의 경우, table에 있는 모든 리스트들을 변수로 주어진 리스트에 연결해준다.

7. MergeLists

MergeLists의 경우, MergedAllLists에서 사용하는 것으로 원래 리스트에 있는 모든 것을 변수로 주어진 리스트에 넣어준다.

8. TBLSearch

TBLSearch의 경우, 해쉬함수를 통해 TBL에서 어디에 속하는지 찾은 후, LFirst를 통해 리스트에서 첫 번째 요소와 bigram을 비교한다. 만약 여기서 bigram가 비교했을 때 같지 않다면 TBLSearchRecursive를 호출한다.

9. TBLSearchRecursive

TBLSearchRecursive의 경우, LNext를 통해 다음 요소를 bigram과 비교한다. 만약 여기서 비교했을 때 같지 않다면 다시 재귀로 TBLSearchRecursive를 호출한다. 교재에서 리스트 검색에서 재귀적 함수를 사용했다고 되어있어 이 부분을 반영한 것이다.

10. InsertionSort

배열의 원소들을 삽입 정렬했다. 이 부분은 교재에서 삽입 정렬을 했다고 나와 있어 그 부분을 반영했다.

11. Hash

Hash의 경우, 문자들에 대해서 단순히 ASCII 코드를 합해서 1021의 나머지를 구하였다.

12. lower1

lower1의 경우, 문자열을 넣으면 그 문자열의 알파벳 대문자 모두를 아스키코드의 소문자로 바꿔준다.

13. strlen2

strlen2의 경우, 교재에서 스트링 길이를 계산하는 함수를 프로파일링에 보이게 만든 것을 보니 직접 구현했던 것 같다. 따라서 나도 strlen의 직접 구현 버전인 strlen2를 만들었다. 이는 문자열의 길이를 계산해준다.

14. TBLInit

TBLInit의 경우, 테이블을 정의해주는 함수이다. 테이블 각각의 리스트들에 초기화해주는 함수인 ListInit을 실행한다. 해시 함수 또한 정의해준다.

15. ListInit

ListInit의 경우, 리스트 함수를 초기화해준다. 이는 테이블 안의 여러 개의 리스트들에 사용하기 위해 만들어진 것이다. head 값을 더미로 쓰기 위해 head 노드를 만들어주고 numOfData값 또한 0으로 설정해준다.

16. LCount

LCount의 경우, 리스트에 몇 개의 노드들이 있는지 알려준다. 이를 위해 numOfData 값을 리턴해준다.

5. 용도 및 테스트 결과

이 프로그램의 용도는 텍스트 파일을 읽고, 각 발생하는 빈도와 함께 유일한 두 단어 표를 만들고, 배열을 새로 만들어 표에 있는 것을 다 저장한 후, 두 단어들이 발생순서가 높은 순

으로 정렬한다.

이에 대해 앞의 50개를 뽑으면 잘 정렬되었음을 알 수 있었다. 테스트 결과는 다음과 같다.

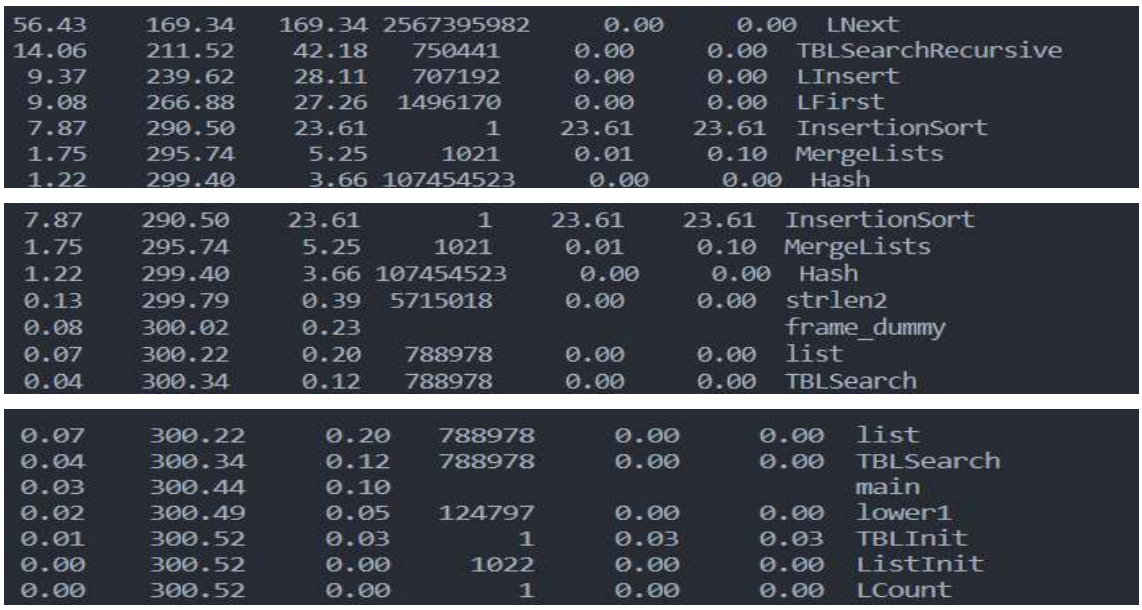
전체 단어 종류: 353596
상위 50개 단어 출력.
bigram: i am frequency: 1728
bigram: of the frequency: 1681
bigram: in the frequency: 1560
bigram: i have frequency: 1504
bigram: i will frequency: 1455
bigram: to the frequency: 1350
bigram: it is frequency: 954
bigram: my lord frequency: 932
bigram: to be frequency: 886
bigram: that i frequency: 878
bigram: i do frequency: 757
bigram: and the frequency: 752
bigram: of my frequency: 677
bigram: and i frequency: 674
bigram: is the frequency: 646
bigram: you are frequency: 633
bigram: i would frequency: 627
bigram: with the frequency: 591
bigram: you have frequency: 591
bigram: he is frequency: 578
bigram: of his frequency: 540
bigram: i know frequency: 536
bigram: all the frequency: 530
bigram: is a frequency: 527
bigram: with a frequency: 527
bigram: if you frequency: 522
bigram: of a frequency: 516
bigram: as i frequency: 506
bigram: in my frequency: 495
bigram: let me frequency: 495
bigram: for the frequency: 495
bigram: shall be frequency: 495
bigram: is not frequency: 491
bigram: by the frequency: 490
bigram: the king frequency: 489

bigram: like a frequency: 476
bigram: thou art frequency: 472
bigram: to my frequency: 469
bigram: will not frequency: 467
bigram: in his frequency: 463
bigram: of this frequency: 459
bigram: do not frequency: 458
bigram: this is frequency: 457
bigram: of your frequency: 451
bigram: in a frequency: 446
bigram: if i frequency: 441
bigram: for your frequency: 439
bigram: out of frequency: 432
bigram: on the frequency: 429
bigram: from the frequency: 422

6. 최적화

6-1. initial.c

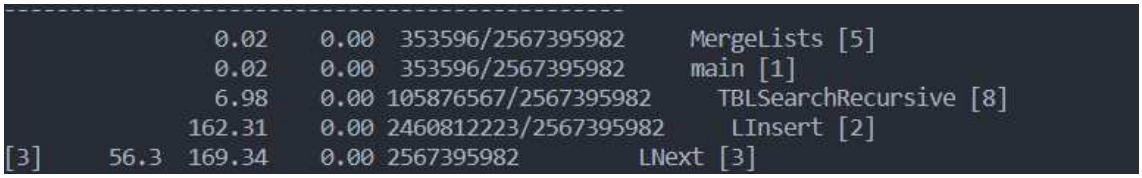
첫 번째 버전의 프로그램의 경우, 사진 3장을 보면 다음과 같이 flat profile이 나타났다. 전체시간은 300.52초, 총 5분이다. 이 flat profile을 보면 LNext가 절반 이상을 차지하고 있음을 알 수 있다. 그런데 LNext의 경우, 실제 self는 0.00초에 가까울 정도로 작은 값임을 알 수 있다. 그러면 약 25억 정도로 많이 호출되어서 문제가 된 것이 아닐까? 이 값을 많이 호출하게 하는 원인을 찾기 위해 Call Graph를 찾아 가보아야겠다.



56.43	169.34	169.34	2567395982	0.00	0.00	LNext
14.06	211.52	42.18	750441	0.00	0.00	TBLSearchRecursive
9.37	239.62	28.11	707192	0.00	0.00	LInsert
9.08	266.88	27.26	1496170	0.00	0.00	LFirst
7.87	290.50	23.61	1	23.61	23.61	InsertionSort
1.75	295.74	5.25	1021	0.01	0.10	MergeLists
1.22	299.40	3.66	107454523	0.00	0.00	Hash
7.87	290.50	23.61	1	23.61	23.61	InsertionSort
1.75	295.74	5.25	1021	0.01	0.10	MergeLists
1.22	299.40	3.66	107454523	0.00	0.00	Hash
0.13	299.79	0.39	5715018	0.00	0.00	strlen2
0.08	300.02	0.23				frame_dummy
0.07	300.22	0.20	788978	0.00	0.00	list
0.04	300.34	0.12	788978	0.00	0.00	TBLSearch
0.07	300.22	0.20	788978	0.00	0.00	list
0.04	300.34	0.12	788978	0.00	0.00	TBLSearch
0.03	300.44	0.10				main
0.02	300.49	0.05	124797	0.00	0.00	lower1
0.01	300.52	0.03	1	0.03	0.03	TBLInit
0.00	300.52	0.00	1022	0.00	0.00	ListInit
0.00	300.52	0.00	1	0.00	0.00	LCount

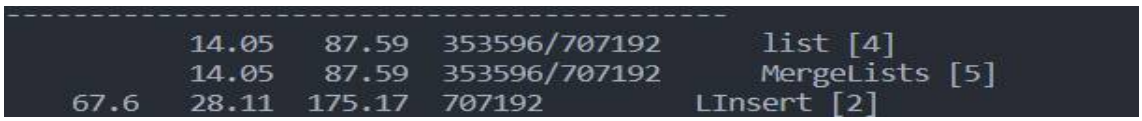
<Flat Profile>

LNext의 Call Graph는 다음과 같다.



<LNext의 Call Graph>

LNext를 25억 중 24억 정도나 호출하는 LInsert의 문제를 찾게 되었다. 한번 LInsert의 Call Graph도 살펴보아야겠다.



<LInsert의 Call Graph>

353596번 list와 MergeLists를 호출하는 이유는 전체 bigram 종류가 353596개이기 때문이다. 따라서 bigram의 종류를 줄이는 수는 없고 list와 MergedLists의 함수는 필요하다고 생각되어진다. 따라서 우리는 24억개나 LNext를 호출하는 LInsert의 근본적인 문제점을 생각해보아야한다.

LNext는 연결리스트의 다음 요소들을 계속해서 찾아서준다. 이 LNext가 LInsert에서 많이 호출된 것은 LInsert 자체가 모든 연결리스트의 끝에서 연결을 해주어야하기 때문이다. 따라서 LNext의 호출을 줄이려면 끝이 아닌 맨 앞에다가 넣어버리면 되는 것이다. 그러면 LInsert에서 LNext를 호출할 일은 사라질 것이다.

그리고 LInsert도 실제로 flat profile 상 3위를 기록하는 28초를 지니고 있다. 353596번 모든 요소들을 LNext로 리스트 끝까지 넣는데 걸리는 시간동안 계속해서 실행하고 있어야 돼서가 아닐까 생각을 해본다. 따라서 LInsert를 끝이 아닌 맨 앞에다가 넣어버리면 LInsert도 빨리 종료될 것이기에 이것 또한 줄어들 가능성이 있어보인다.

6-2. Finsert.c

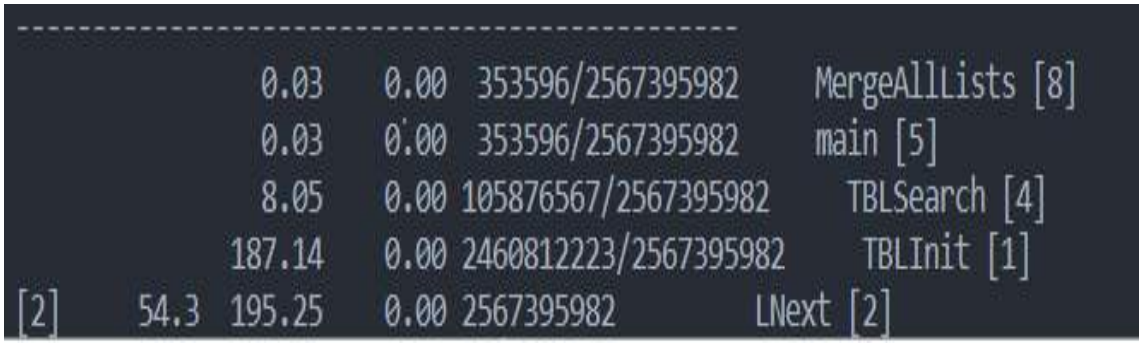
FInsert를 만들어 LInsert가 있는 list 함수(해쉬에 해당하는 연결리스트를 찾아서 그 리스트에 붙여주는 함수), MergedLists(태그로 연결되어있는 연결리스트들을 하나의 연결리스트로 연결)에서 LInsert를 FInsert로 대체시켰다.

```c
void LInsert(List* plist, LData data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    LData * pdata;
    if (plist->numOfData == 0)
    {
        newNode->next = plist->head->next;
        plist->head->next = newNode;
        (plist->numOfData)++;
    }
    else
    {
        if (LFirst(plist, &pdata))
        {
            while (LNext(plist, &pdata));

        }
        plist->cur->next = newNode;
        newNode->next = NULL;
        (plist->numOfData)++;
    }
```

```c
void FInsert(List* plist, LData data)
{
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;

    newNode->next = plist->head->next;
    plist->head->next = newNode;
    (plist->numOfData)++;
    /* 연결 리스트 앞에다가 넣기. */
}
```

<LInsert와 FInsert 비교>

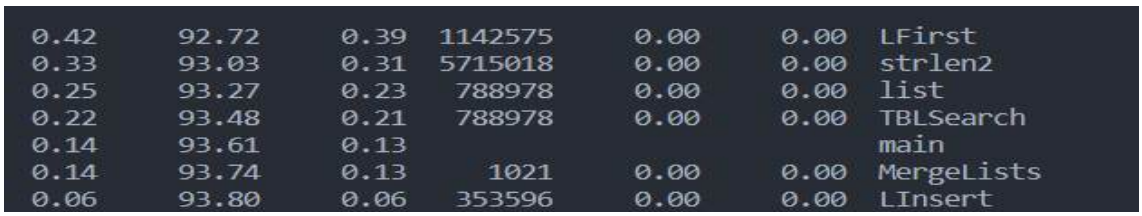
LInsert를 FInsert로 대체했더니 에러가 나는 것 아닌가? 에러가 나타나도 gprof가 나오길래 그것을 분석했다. FInsert는 앞에 넣는 것 빼고는 LInsert와 다를 바가 없는데 에러가 어째서 나오는가 궁금했다. TBLSearch가 10억을 기록한다. 다른 말로 frequency가 높은 애들이 대부분 뒤에 가 있는 현상인 것이다. frequency가 높은 애들이 뒤에 있기 때문에 TBLSearch가 계속해서 찾지 못한다는 의미이다. 우리 교재인 컴퓨터 시스템 책에도 저자도 iter first로 앞에다가 놓았는데 위와 같이 frequency가 높은 요소들이 뒤에 가는 현상 때문에 오히려 성능이 안 좋아졌음을 언급했다. 그러면 해시와 관련이 있는 list 함수에만 이것이 해당될 것이다. MergedLists 함수에서 LInsert만을 FInsert로 바꾸고, list 함수에 LInsert는 그대로 두면 FInsert.c처럼 오류도 나지 않고 크게 성능도 개선될 수 있지 않을까 생각한다.

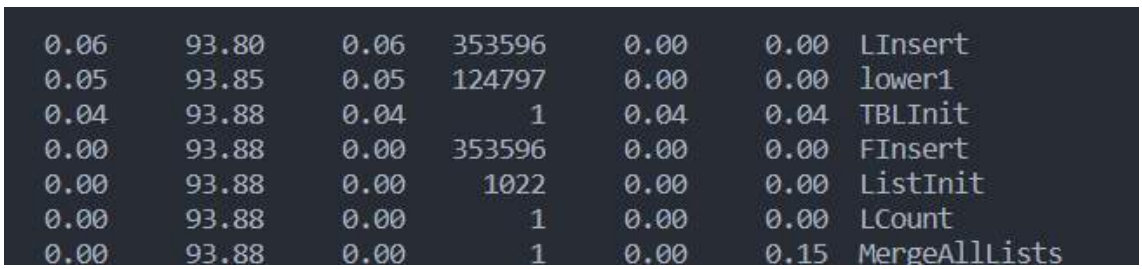


<FInsert.c의 LNext Call Graph>

6-3. FInsert2.c

이렇게 바꾸니 300.52초가 실제로 93.88초까지 내려갔다. 총 3배 정도의 성능 향상이 이루어졌다. LNext가 169.34초에서 9.5초로 16배 이상 줄어들었다.

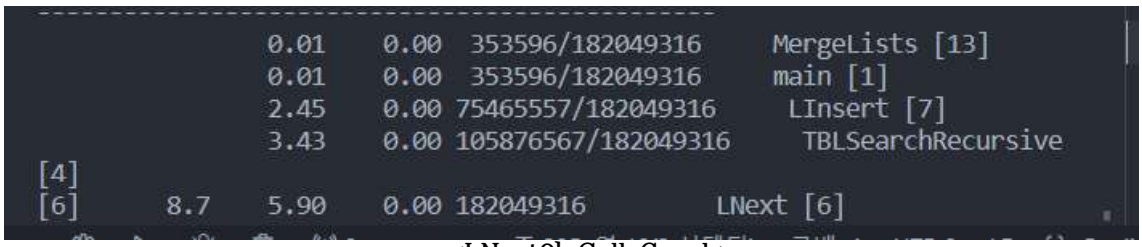




<Flat Profile>

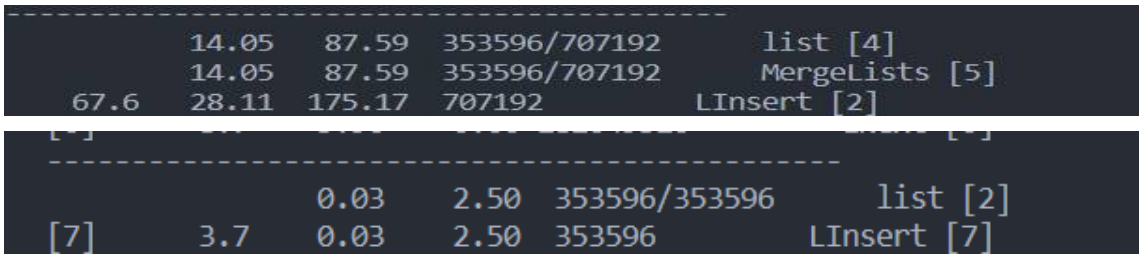
그 이유로 다음과 같다. 아래 사진을 LNext의 Call Graph를 보면 LInsert가 24억번 정도 LNext를 부르는 것을 7천만 정도로 줄였다. 사실상 LInsert의 LNext 호출의 대부분을 MergedLists가 쓰고 있었다는 말이다. 이는 당연한 이유가 1021개의 Hash 태그로 연결리스트들을 나누었으므로 353596/1021 = 346.3으로 list의 LInsert는 평균적으로 LNext를 346번 쓸 것이다. 당연히 list의 LInsert는 MergedLists가 35만개를 이어붙이는 것을 생각하면 list

함수의 LInsert가 MergedLists의 LInsert보다 LNext를 훨씬 덜 쓸 것이다.



<LNext의 Call Graph>

그러므로 아래사진과 같이 LInsert가 70만개에서 35만개로 줄은 것인데 그 LNext의 숫자는 사실상 25배 이상 줄어든 것이나 다름없는 것이다.



<첫 번째 버전과 세 번째 버전 차이>

지금까지 300.52초가 실제로 93.88초로 내려가는 과정을 보았고 그 이유까지 알아내었다. TBLSearchRecursive가 1위로 %가 높은 것을 알고 있으나, 일단 먼저 위의 Flat Profile에서 InsertionSort를 보아야한다. 이는 실제로 35.06초로 프로그램의 37.40%정도의 시간을 차지하고 있다. 나는 삽입 정렬을 더 좋은 성능을 가지고 있는 평균 성능시간 O(n*logn)을 지닌 퀵 소트로 대체할 것이다.

6-4. QuickSort.c

나는 InsertionSort를 아래의 QuickSort 구현 코드 사진과 같이 QuickSort로 바꾸었다. QuickSort에 활용되는 Partition, Swap 함수까지 같이 구현했다. 이를 InsertionSort와 대체하였다. 성능이 O(nlogn) 정도니 분명 좋은 성과가 있을 것이라 예상하였다.

```
}
void QuickSort(LData * arr, int left, int right)
{
    if(left <= right)
    {
        int pivot = Partition(arr, left, right);
        QuickSort(arr,left, pivot-1);
        QuickSort(arr, pivot+1, right);
    }
}
```

```c
void Swap(LData * arr, int idx1, int idx2)
{
    LData temp = arr[idx1];
    arr[idx1] = arr[idx2];
    arr[idx2] = temp;
}
int Partition(LData * arr, int left, int right)
{
    LData pivot = arr[left];
    int low = left + 1;
    int high = right;

    while(low<=high)
    {
        while(pivot.frequency <= arr[low].frequency && low<=right)
          low++;

        while(pivot.frequency >= arr[high].frequency && high >= (left+1))
          high--;

        if(low <= high)
          Swap(arr,low,high);


    }
    Swap(arr, left, high);
    return high;
```

<Quick Sort 구현 코드>

구현했더니 다음과 같은 flat profile 결과가 나타났다. InsertionSort가 35.06초인 것을 감안했을 때 QuickSort 안의 세 함수(QuickSort, Partition, Swap)으로 인해 걸린 시간은 Partition의 16.13초만 걸렸다. 2배 정도를 단축시켰음을 알 수 있다.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
58.79	44.57	44.57	750441	0.00	0.00	TBLSearchRecursive
21.28	60.70	16.13	353597	0.00	0.00	Partition
12.59	70.25	9.54	182049316	0.00	0.00	LNext
5.02	74.05	3.81	107454523	0.00	0.00	Hash
0.62	74.52	0.47				frame_dummy
5.02	74.05	3.81	107454523	0.00	0.00	Hash
0.62	74.52	0.47				frame_dummy
0.50	74.90	0.38	5715018	0.00	0.00	strlen2
0.38	75.19	0.29	1142575	0.00	0.00	LFirst
0.25	75.38	0.19				main
0.21	75.54	0.16	788978	0.00	0.00	TBLSearch
0.20	75.69	0.15	1021	0.00	0.00	MergeLists
0.14	75.79	0.11	788978	0.00	0.00	list
0.08	75.85	0.06	353596	0.00	0.00	LInsert
0.04	75.88	0.03	124797	0.00	0.00	lower1
0.03	75.91	0.03	1	0.03	0.03	TBLInit
0.01	75.91	0.01	1022	0.00	0.00	ListInit
0.00	75.91	0.00	420443	0.00	0.00	Swap
0.00	75.91	0.00	353596	0.00	0.00	FInsert

<flat profile>

우리는 93.88초에서 75.91초까지 단축시켰다. 총 시간은 1.2배 정도 향상되었다. InsertionSort를 QuickSort로 바꾸는 작업은 TBLSearchRecursive 작업이 끝난 이후이기 때문에 TBLSearchRecursive와 아무 연관성이 없었다. 따라서 전 버전과 같이

TBLSearchRecursive가 1위로 비중을 차지함을 알 수 있었다. InsertionSort를 QuickSort로 바꾸면서 줄였기 때문에 차지하는 비율 또한 46%에서 58%로 증가하였다. 이제 TBLSearchRecursive를 해결해야할 차례이다.

그러면 TBLSearchRecursive를 누가 실행하는지 보아야하기 때문에 Call Graph를 살펴보아야한다.

```
                                105126126           TBLSearchRecursive [4]
                43.80    9.03  750441/750441      TBLSearch [3]
[4]     56.3    43.80    9.03  750441+105126126 TBLSearchRecursive [4]
```

<TBLSearchRecursive Call Graph>

TBLSearch와 TBLSearchRecursive 자신과 관계있음을 알 수 있다. 그런데 다음 TBLSearch Call Graph를 살펴보자.

```
-----------------------------------------------
                0.21   53.12  788978/788978      list [2]
 [3]     56.8   0.21   53.12  788978          TBLSearch [3]
```

<TBLSearch Call Graph>

list, 즉 찾는 함수와 관계되어있음으로 데이터양을 모두 찾아야하기에 호출되는 횟수가 많음을 알 수 있다. 그러면 호출되는 횟수와 상관없이 함수의 효율을 높여 시간을 줄여볼 수 있지 않을까?

6-5. revision_hash.c

일단 TBLSearchRecursive 함수는 아래의 사진을 보면 다음과 같다. 그런데 이 함수의 두 번째 줄인 pt->hf(v);를 보면 hv를 주소로서 가져옴을 알 수 있다. pt->tbl[*hv]와 같이 주소로써 사용되어져 *hv를 씀을 알 수 있다. 메모리 참조를 할 필요 없는 상황에서 메모리 참조를 만드는 것은 매우 비효율적이다. 심지어 TBLSearchRecursive를 보면 엄청 많은 횟수로 돌려짐을 알 수 있는데 엄청 많은 횟수로 주소가 참조되는 것이다. list 또한 hv를 사용한다. list도 보면 알다시피 80만번 정도 호출된다. 여기서도 주소로써 참조되는 것은 매우 좋지 않다.

```c
LData* TBLSearchRecursive(Table* pt, char* text)
{
    int * hv = malloc(sizeof(int));
    pt->hf(text, hv);
    LData* bi = NULL;

    if (LNext(&(pt->tbl[*hv]), &bi))
    {
        if (strcmp(text, bi->bigram)==0)
        {

            return bi;
        }
        else
        {

            return TBLSearchRecursive(pt, text);
        }
    }

    return NULL;
}
```

<TBLSearchRecursive 함수>

따라서 나는 HashFunc의 typedef void HashFunc(char* bigram, int *hv);과 같은 함수 선언을 typedef int HashFunc(char* bigram);로 바꿀 것이다. list, TBLSearchrecursive, TBLSearch까지의 모든 함수 내에서 메모리 참조가 되는 부분들을 없앨 것이다.

수정하고 revision_hash.c를 flat profile 표출한 결과 다음과 같다.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
45.51	17.17	17.17	750441	0.00	0.00	TBLSearchRecursive
41.23	32.72	15.55	353597	0.00	0.00	Partition
5.50	34.79	2.07	182049316	0.00	0.00	LNext
4.10	36.34	1.55	107454523	0.00	0.00	Hash
0.80	36.64	0.30	1142575	0.00	0.00	LFirst
4.10	36.34	1.55	107454523	0.00	0.00	Hash
0.80	36.64	0.30	1142575	0.00	0.00	LFirst
0.68	36.89	0.26	5715018	0.00	0.00	strlen2
0.46	37.07	0.18	788978	0.00	0.00	TBLSearch
0.42	37.23	0.16	1	0.16	0.17	TBLInit
0.29	37.34	0.11				main
0.27	37.44	0.10	353596	0.00	0.00	LInsert
0.24	37.53	0.09	788978	0.00	0.00	list
0.24	37.62	0.09				frame_dummy
0.21	37.70	0.08	1021	0.00	0.00	MergeLists
0.15	37.76	0.06	124797	0.00	0.00	lower1
0.03	37.77	0.01	353596	0.00	0.00	FInsert
0.01	37.77	0.01	1022	0.00	0.00	ListInit
0.00	37.77	0.00	420443	0.00	0.00	Swap

<flat profile>

우리는 75.91초에서 37.77초까지 줄일 수 있었다. 총 시간을 약 2배 정도 줄일 수 있었던 것이다. hash함수의 메모리 참조를 없애면서 예상했던 데로 TBLSearchRecursive를 크게 줄

일 수 있었다. TBLSearchRecursive의 경우, 44.57초에서 17.17초로 줄일 수 있었다. 이는 2.6배 정도 성능 향상을 이룬 것이다. 하지만 여전히 TBLSearchRecursive는 17.17초로 해결해야할 1순위이다. 어떻게 하면 더 줄일 수 있을까?

6-6. iteration.c

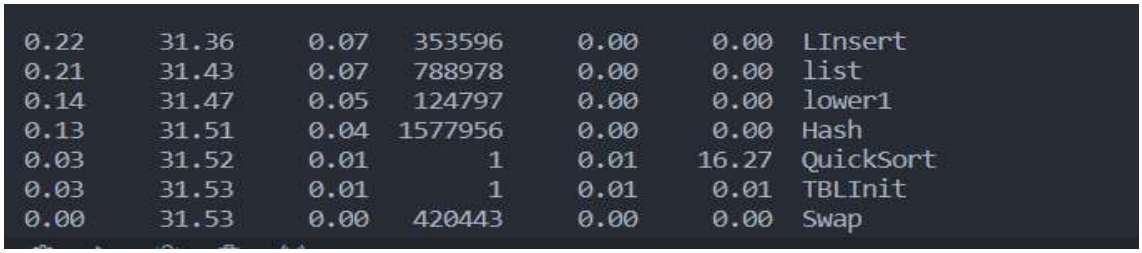
TBLSearchRecursive는 재귀를 계속 호출하는 형태로 되어있다. 즉 계속 재귀 호출하면서 데이터를 찾는 것이다. 일반적으로 재귀적 버전보다 반복하는 버전이 더 시간적으로나 공간적으로나 좋다는 것은 기정사실이다. 따라서 이러한 재귀적 형태를 반복적 형태로 바꿔볼만한 것 같다. 반복적으로 구현할 것이면 TBLSearchRecursive와 TBLSearch를 이렇게 두 가지로 나눌 필요 없이 TBLSearch로 합쳐도 된다.

```
LData *bi= NULL;
if (LFirst(&(pt->tbl[hv]), &bi))
{
    if (strcmp(text, bi->bigram)==0)
    {

        return bi;
    }
    else
    {

        while(LNext(&(pt->tbl[hv]), &bi))
        {
            if(strcmp(text, bi->bigram)==0)
            {
                return bi;
            }
```

<TBLSearch iteration 버전>

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
51.64	16.26	16.26	353597	0.00	0.00	Partition
38.51	28.39	12.13	788978	0.00	0.00	TBLSearch
6.90	30.56	2.17	182049316	0.00	0.00	LNext
0.95	30.86	0.30	1142575	0.00	0.00	LFirst
0.73	31.09	0.23	5715018	0.00	0.00	strlen2

6.90	30.56	2.17	182049316	0.00	0.00	LNext
0.95	30.86	0.30	1142575	0.00	0.00	LFirst
0.73	31.09	0.23	5715018	0.00	0.00	strlen2
0.35	31.20	0.11				main
0.29	31.29	0.09	1021	0.00	0.00	MergeLists
0.22	31.36	0.07	353596	0.00	0.00	LInsert
0.21	31.43	0.07	788978	0.00	0.00	list



0.22	31.36	0.07	353596	0.00	0.00	LInsert
0.21	31.43	0.07	788978	0.00	0.00	list
0.14	31.47	0.05	124797	0.00	0.00	lower1
0.13	31.51	0.04	1577956	0.00	0.00	Hash
0.03	31.52	0.01	1	0.01	16.27	QuickSort
0.03	31.53	0.01	1	0.01	0.01	TBLInit
0.00	31.53	0.00	420443	0.00	0.00	Swap

<flat profile>

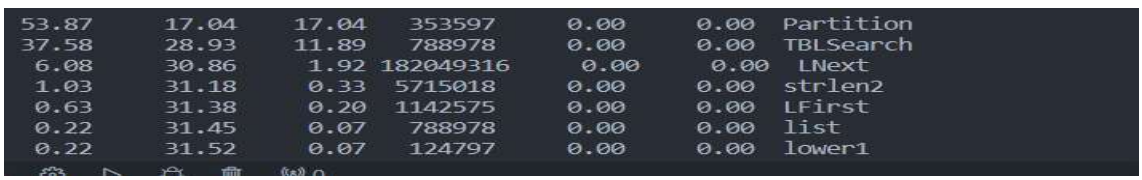
37.77초에서 31.53초로 줄일 수 있었다. 대략 성능 향상이 1.2배 정도 된 것이다. TBLSearch와 TBLRecursive 함수는 17.17초에서 12.13초로 총 5초나 감소하였다. 대략 30%정도 감소한 것이다. 이제 다시 주된 관심사는 Partition인 Sort 함수와 관련된 함수가 되었다. InsertionSort에서 감소시킨 것이었지만 우리는 더욱 더 최적화를 시켜보도록 노력해 보아야한다. Call graph를 살펴보아도 QuickSort만이 Partition을 호출하기에 이를 살펴보는 것은 의미없다. 근본적인 Partition함수를 수정해야한다.

6-7. QuickSort2.c

```
int pivotIndex = left + rand() % (right - left + 1);
Swap(arr, left, pivotIndex);
LData pivot = arr[left];
int low = left + 1;
```

<Pivot random 지정>

이처럼 Pivot index를 랜덤으로 지정해보면 어떨까했다. 오히려 Partition 함수가 증가했음을 알 수 있었다. 우리는 전 버전인 31.53초 버전을 사용할 것이다.



53.87	17.04	17.04	353597	0.00	0.00	Partition
37.58	28.93	11.89	788978	0.00	0.00	TBLSearch
6.08	30.86	1.92	182049316	0.00	0.00	LNext
1.03	31.18	0.33	5715018	0.00	0.00	strlen2
0.63	31.38	0.20	1142575	0.00	0.00	LFirst
0.22	31.45	0.07	788978	0.00	0.00	list
0.22	31.52	0.07	124797	0.00	0.00	lower1

<flat profile>

partition의 경우, QuickSort보다 더 나은 알고리즘이 생각나지 않고, 피봇을 랜덤으로 하는 것 이외에는 방법이 생각나지 않는다. 일단 2위인 TBLSearch를 최적화하는 방안을 찾아봐야겠다.

6-8. new_hash.c

TBLSearch가 오래 걸리는 이유는 해시 함수 때문이라고 생각했다. 지금까지 #define MAX_TBL 1021로 지정하면서 해시 태그의 버킷을 총 1021개만 가지고 있었다. 하지만 353,596/1021 = 346.3으로 실제로 평균적으로 골고루 분포한다고 쳐도 연결리스트에 노드가 346개나 연결되어있는 것이다. 이는 TBLSearch가 어째서 연결리스트의 노드들을 찾는데 오래 걸리는지 알려준다. 따라서 우리는 hash 태그 개수를 #define MAX_TBL 100000로 지정하며 10만개로 늘릴 것이다.

그리고 원래 해시 함수의 경우, 모든 문자들에 대해 단순히 아스키코드를 합한 다음에 1021로 나눈 나머지 값을 해쉬 버킷으로 썼다. 하지만 이 값은 10만개의 해시 태그를 쓸 경우, 이상하다. 한 문자의 최대 아스키코드 값은 122이며, n 문자 스트링은 최대 122n의 합을

만들 것이다. “honorifica bilitudinitatibus thou”는 겨우 3371의 합을 가진다. 따라서 해시 테이블의 대부분의 버킷들을 사용되지 않은 채로 남을 것이다. 추가로, 덧셈 교환성이 있는 해시 함수는 한 개의 스트링의 가능한 여러 가지 문자순서를 구분하지 못할 것이다. “rat”와 “tar”과 같은 것 말이다.

따라서 우리는 Hash함수를 아래 사진과 같이 제곱의 형태로 계속해서 더한 후, 마지막에 shift를 왼쪽에 5번할지, 오른쪽에 9번할지 Or로 지정하였다. 이정도면 10만 개의 해시 버킷들을 모두 골고루 사용할 수 있다고 생각했다.

```c
/* 교재에 있는 Hash 함수 지정.*/
int Hash(char* bigram)
{
    int hv = 0;

    while (*bigram != 0)
    {
        hv += *bigram * *bigram;
        bigram++;
    }
    hv = (hv << 5) | (hv >> 9);
    hv %= 100000;
    return hv;
}
```

<새로운 해시 함수 구성>

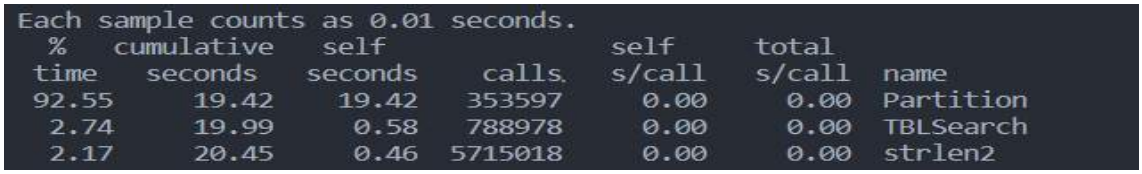
이렇게 구성했더니 다음과 같은 Flat Profile이 나타났다.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
92.30	10.61	10.61	353597	0.00	0.00	Partition
2.87	10.95	0.33	788978	0.00	0.00	TBLSearch
1.83	11.16	0.21	5715018	0.00	0.00	strlen2
1.65	11.35	0.19	1203313	0.00	0.00	LFirst
0.44	11.40	0.05	100000	0.00	0.00	MergeLists
0.44	11.40	0.05	100000	0.00	0.00	MergeLists
0.44	11.45	0.05				main
0.26	11.48	0.03	788978	0.00	0.00	list
0.17	11.50	0.02	1577956	0.00	0.00	Hash
0.09	11.51	0.01	7476857	0.00	0.00	LNext
0.09	11.52	0.01	124797	0.00	0.00	lower1
0.00	11.52	0.00	407073	0.00	0.00	Swap
0.00	11.52	0.00	353596	0.00	0.00	FInsert
0.00	11.52	0.00	353596	0.00	0.00	LInsert
0.00	11.52	0.00	100001	0.00	0.00	ListInit

<Flat Profile>

TBLSearch가 0.33으로 전 전 버전인 iteration.c 기준으로 12.13에서 0.33만큼 떨어졌다. 36배 이상 성능이 향상되었음을 알 수 있다. Partition 함수 또한 자연스럽게 16.26에서 10.61로 최적화되었는데, 이는 해시 함수가 바뀌면서 자연스럽게 Partition 함수가 더 최적화

되어지는 분포로 연결리스트의 구조가 바뀐 것임을 알 수 있었다. Partition 함수 또한 1.6배 성능이 향상되었다. 하지만 Partition 함수의 경우, 아래 사진을 보면 new_hash를 다시 실행해보면 19.42초까지 올라가는 경우도 있었다. 이는 해시 함수가 (hv << 5) | (hv >> 9)의 구조를 따르기에 Partition 함수에 최적화되어지는 구조로 연결 리스트들이 구성되는지는 랜덤하게 되어지는 것으로 보인다.



Each sample counts as 0.01 seconds.

% time	cumulative seconds	self seconds	calls	self s/call	total s/call	name
92.55	19.42	19.42	353597	0.00	0.00	Partition
2.74	19.99	0.58	788978	0.00	0.00	TBLSearch
2.17	20.45	0.46	5715018	0.00	0.00	strlen2

<new_hash 다시 실행 Flat Profile>

하지만, 그래도 TBLSearch가 12초 정도에서 1초 미만으로 떨어졌다는 사실은 변함이 없다. Partition 함수만 11~20초 정도의 범위를 가질 뿐이다. Partition 함수의 최악의 경우를 고려하더라도 전체 시간은 31.53초에서 21초 정도로 내려온다. 이는 최적화되어진 것이다.

7. 최종 결과

QuickSort 함수의 경우, 내가 공부했던 자료구조 서적인 윤상우의 열혈 자료구조 서적을 참고해서 만들었으나, 혹시 인터넷의 자료들로 QuickSort를 바꾼다면 더 효율이 높아지지 않을까 시도해보았다. 하지만 당연히 QuickSort 함수는 서적이든 인터넷이든 너무 유명하고, 당연한 함수이기 때문에 같은 구조를 지니고 있었고, 성능 향상이 이루어지지 못했다.

이 프로그램에서 QuickSort보다 더 나은 정렬 알고리즘은 생각이 나지 않으며, 따라서 여기서 프로그램 최적화를 완료하고자 한다. 각 단계를 정리해보자면 다음과 같다.

1. initial.c로 최초의 버전이다. 300.52초의 시간을 지니고 있다.

2. list 함수와 MergedLists 함수의 연결리스트에 요소를 뒤에 넣는 방식인 LInsert를 앞에 넣는 방식인 FInsert로 대체한 버전이다. 이 버전은 list에서 에러가 나타났다. 그 이유는 본 교재와 수업에서 언급해있다시피 Frequency가 높은 것들이 FInsert로 놓을 경우, 뒤에 들어가기 때문이다. TBLSearchRecursive는 계속 호출될 수밖에 없으며 따라서 에러가 발생한 것이다.

3. MergedLists 함수의 LInsert만 FInsert로 대체하였다. 300.52초에서 93.88초로 내려왔다 총 3배 정도의 성능 향상이 이루어진 것이다. LNext가 169.34초에서 9.5초로 16배 이상 줄어들었다.

4. InsertionSort를 QuickSort로 대체하였다. InsertionSort가 35.06초 정도 걸리는 데 QuickSort로 16.13초 정도로 단축시켰다. 2배 정도 성능을 향상시킨 것이다. 총시간은 93.88초에서 75.91초까지 단축시켰다. 총 시간은 1.2배 정도 향상되었다.

5. list, TBLSearch, TBLSearchRecursive의 쓸 데 없는 해시 함수의 메모리 참조를 없애려고 노력하였다. 특히 TBLSearch는 엄청 많이 실행되면서 비중 또한 58.79%나 되었기에 유용한 최적화가 될 것이라고 생각하였다. 우리는 총 시간을 75.91초에서 37.77초까지 줄일 수 있었다. 총 시간을 약 2배 정도 줄일 수 있었던 것이다. TBLSearchRecursive의 경우, 44.57초에서 17.17초로 줄일 수 있었다. 이는 2.6배 정도 함수의 성능 향상을 이룬 것이다.

6. TBLSearchRecursive는 재귀함수를 사용하는데 이것을 반복문 형식으로 바꾸기로 결정하였다. 37.77초에서 31.53초로 줄일 수 있었다. 대략 성능 향상이 1.2배 정도 된 것이다. TBLRecursive 함수와 TBLSearch 함수가 합쳐진 TBLSearch 함수는 17.17초에서 12.13초

로 총 5초나 감소하였다. 대략 30%정도 감소한 것이다.

7. Partition 함수의 성능을 향상해보기 위해 pivot을 랜덤으로 설정해보았다. 하지만 성능이 크게 향상되어지지 않았다.

8. TBLSearch의 성능 병목이 해시 함수가 1021개의 버킷만을 가지고 있고, 이 해시 함수를 설정하는 방법이 잘못되어서 이루어져있다고 생각했다. TBLSearch가 0.33으로 전 전 버전인 iteration.c 기준으로 12.13에서 0.33만큼 떨어졌다. 36배 이상 성능이 향상되었음을 알 수 있다. partition 함수의 경우, 성능이 향상될 때도 있었고 안 될 때도 있었다. 이는 연결 리스트 분포가 랜덤으로 이루어지기 때문이다. 그래도 TBLSearch가 36배 이상 성능이 향상되었음은 변하지 않는다. 31.53초에서 11~21초 정도로 내려온다. 이는 1.5배~2.8배 성능 향상이 이루어진 것이다.

최종적으로 프로그램이 300.52초에서 11~21초 정도로 최적화가 완료되었다. 이는 15배에서 30배 정도의 성능 향상을 이룬 것이다. 나머지 총 시간의 92% 정도를 차지하는 partition의 경우, 따로 성능 병목 현상이 보이지 않았다. 따라서 이상으로 성능 향상을 마치도록 하겠다.