EECE372 HW#2

20190445 허수범

March 26, 2024

1 Source Code에 대한 설명

1.1 Flow

전체적인 시스템의 flow는 Figure 1의 flowchart로 시각화할 수 있습니다.

먼저 두 번의 Exception handling을 거치며 커맨드라인 input들을 모두 int 변수에 입력합니다. 이후, n개의 k의 배수를 랜덤으로 저장할 배열 'arrL'과 정렬 결과를 저장할 배열 'arr_result_arrL'과 'LL_result_arrL'을 동적으로 할당해줍니다. 'prod_MultofK' 함수를 이용해 'arrL'에 n개의 k의 배수를 무작위 순서로 생성해 저장하도록 하였습니다. 'prod_MultofK' 함수의 코드는 Listing 1에서 확인할 수 있는데, n개의 k의 배수를 순서대로 생성한 후, Knuth shuffle algorithm을 이용해 무작위 순서로 섞도록 하였습니다. 지난 HW#1에서는 Knuth shuffle algorithm을 이용하지 않고 무작위 숫자를 생성하도록 하였는데, n의 값이 커짐에 따라 숫자 생성에 소요되는 시간이 과도하기 늘어나 이번 HW#2에서는 더욱 효율적인 알고리즘인 Knuth shuffle을 채택하였습니다. 여기에 쓰인 'swap'함수는 HW#1에서 사용하였던 함수와 동일합니다.

```
// produce random n multiples of k using srand in array
void prod_MultofK(int* arr, int k, int n){
    srand(time(NULL));

    int i;
    for (i = 0; i < n; i++) {
        arr[i] = (i + 1) * k;
    }

    // Knuth shuffle
    for (i = 0; i < n - 1; i++) {
        int j = i + rand() % (n - i);
        swap(&arr[i], &arr[j]);
    }
}
```

Listing 1: prod_MultofK Function.

이후 'create_pqarray()'과 'create_pqlinkedList()'함수를 사용해 'Queue_array' 구조체와 'Queue_linkedList' 구조체를 선언합니다. 각 구조체와 구조체 생성 함수, 그리고 각 구조체의 Node의 정의는 Listing 2에서 확인할 수 있습니다. Linked-list의 Node와 Array의 Node는 Memory Overhead를 줄이기 위해 다르게 정의해 주었습니다. 구조체들을 생성해 준 이후에는 정확한 Execution 시간을 측정하기 위해 'struct timespec'

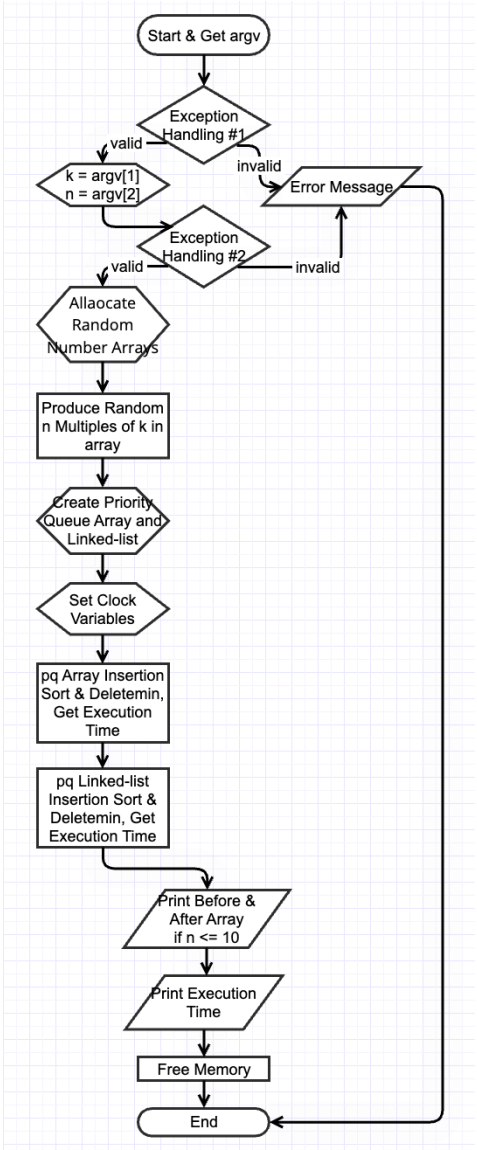


Figure 1: Flowchart.

```c
// node structure of pq linked list
typedef struct Node {
    int key;
    int data;
    struct Node* next;
} Node;

// node structure of pq array
typedef struct {
    int key;
    int data;
} Node_arr;

// pq array structure
typedef struct {
    Node_arr* arr;
    int head, tail;
} Queue_array;

// pq linked list structure
typedef struct {
    Node *front, *rear;
} Queue_linkedList;

// create qriority queue array
Queue_array* create_pqarray(int size){
    Queue_array* q = (Queue_array*)malloc(sizeof(Queue_array)); // dynamically allocate memory for array
    q->arr = (Node_arr*)malloc(sizeof(Node_arr) * size);
    q->head = 0;
    q->tail = -1; // to indicate no element
    return q;
}

// create new node for Linked-list
Node* create_node(int value) {
    Node* new_node = (Node*)malloc(sizeof(Node)); // dynamically allocate memory for new node
    new_node->key = value;
    new_node->data = value;
    new_node->next = NULL;
    return new_node;
}

// create qriority queue linked list
Queue_linkedList* create_pqlinkedList(){
    Queue_linkedList* q = (Queue_linkedList*)malloc(sizeof(Queue_linkedList)); // dynamically allocate memory for Linked-list queue
    q->front = NULL; // set front and rear node to NULL
    q->rear = NULL;
    return q;
}
```

Listing 2: 구조체 선언과 create함수들.

구조체들을 선언해 주고, for문을 활용해 'insert_arr()'함수를 n번, 'deleteMin_arr()'함수를 n번 실행하고 그 실행시간을 'clock_gettime()' 함수를 이용해 측정해 주었습니다. 같은 과정을 Linked-list에 대해서도 진행해 주었습니다. 두 경우 모두 실행 결과를 확인하기 위해 'deleteMin' 함수에서 front(혹은 head)의 key를 return하도록 해주어 return된 정수를 위해서 동적으로 할당해 주었던 배열들에 입력하도록 하였습니다.

이후 만약 n이 10 이하의 정수라면, 'printer()' 함수를 호출하여 sorting 이전과 이후의 배열을 출력해 주도록 하였습니다. 여기서 'printer()' 함수는 HW#1때와 같이 단순한 for loop으로 구현하였습니다. 그 다음에는 n의 값에 상관 없이 실행 시간을 소수점 6자리까지 출력하도록 해주었습니다.

마지막으로, main함수는 이때까지 동적으로 할당된 모든 메모리를 해제하는 것으로 끝이 납니다. 'pq_array'의 경우, 해당 구조체 내의 Node Array를 함께 할당 해제해주었지만, 'pq_linkedList'의 경우에는, 'deleteMin_linkedList()' 함수 내에서 각 node의 메모리를 할당 해제해주기 때문에, 'pq_linkedList' 구조체만 할당 해제하도록 구현하였습니다.

1.2 Algorithm 동작 원리

HW#2의 요구사항은 무작위 숫자가 저장되어 있는 list에서 원소를 하나씩 빼 priority queue(이하 pq)에 insertion sort 방식으로 입력하고, 모든 원소가 입력된 후에는 다시 하나씩 deleteMin을 진행하는 것이었습니다. Array based pq와 Linked-list based pq의 insert와 delete 함수를 각각 따로 구현하였습니다.

1.2.1 Array

Array Based Priority Queue의 insertion과 deletion은 Listing 3와 같이 구현하였습니다. 'insert_arr()' 함

```
// insert element in pq array
void insert_arr(Queue_array* q, int value){
    q->tail = q->tail + 1; // increment tail pointing integer by 1

    int i = q->tail;
    while(i > q->head && value < (q->arr[i - 1]).data) {
        q->arr[i] = q->arr[i - 1]; // move all element bigger than new key to the
    right
        i--;
    }
    q->arr[i].key = value; // insert value in the right place
    q->arr[i].data = value;
}

// delete min element of pq array
int deleteMin_arr(Queue_array* q){
    int temp = q->arr[q->head].data;

    int i;
    for(i = q->head; i < q->tail; i++){
        q->arr[i] = q->arr[i + 1]; // move every element to the left
    }

    q->arr[q->tail].data = -1; // delete data in the place of current tail
    q->arr[q->tail].key = -1; // delete key in the place of current tail
    q->tail = q->tail - 1; // decrement tail pointing integer by 1
    return temp; // return min key
}
```

Listing 3: Insertion and deleteMin function for Array Based Priority Queue.

수는 먼저 tail을 가리키는 정수의 숫자을 1 더하고, 해당 숫자에서부터 입력된 value보다 작은 값을 발견할

때까지 while문을 이용하여 배열의 모든 원소를 한 칸씩 오른쪽(인덱스가 증가하는 방향)으로 이동합니다. value보다 작은 값을 발견하면, while문은 종료되고, 찾아낸 인덱스의 구조체의 key와 data 값을 입력 받은 value로 설정합니다.

'deleteMin_arr()' 함수는 먼저 최솟값일 head 구조체의 data를 정수 temp변수에 저장하고, for문을 활용하여 배열의 모든 변수를 왼쪽으로 이동합니다. 이후 원래 tail이 있던 인덱스에 있는 구조체의 data와 key를 '-1'로 설정하고(이는 배열의 원소가 비어 있다는 것을 indicate합니다), tail을 가리키는 정수의 값을 1만큼 decrement합니다. 마지막으로, temp에 저장되어 있던 정수 값을 return합니다.

1.2.2 Linked-list

Linked-list Based Priority Queue의 insertion과 deletion은 Listing 4와 같이 구현하였습니다. 'insert_linkedList()'

```
// insert element in pq linked list
void insert_linkedList(Queue_linkedList* q, int value){
    Node* new_node = create_node(value); // create new node

    // If the linked list is empty or the new node has higher priority than the front node
    if (q->front == NULL || value < q->front->data) {
        new_node->next = q->front; // Set new node as the new front
        q->front = new_node;
        if (q->rear == NULL) { // If the linked list was empty
            q->rear = new_node; // Set new node as rear node
        }
        return;
    }

    Node* curr = q->front;
    while(curr->next != NULL && value > curr->next->data){ // find location for new node by traversing through Linked-list
        curr = curr->next;
    }
    new_node->next = curr->next;
    curr->next = new_node;
    if(new_node->next == NULL){
        q->rear = new_node;
    }
}

// delete min element of pq linked list
int deleteMin_linkedList(Queue_linkedList* q){ // delete(free memory) front node and return the value of it
    Node* curr = q->front;
    int temp = q->front->data;
    q->front = q->front->next;
    free(curr);
    return temp;
}
```

Listing 4: Insertion and deleteMin function for Linked-list Based Priority Queue.

함수는 먼저 'create_node()'함수를 사용해 새로운 node를 만들고, Linked-list가 비어 있거나 입력 받은 value가 front 노드의 데이터보다 크기가 작을 시에는 새로운 노드를 front로 연결해주고, 그렇지 않은 경우에는 value의 값보다 큰 node가 등장할 때까지 Linked-list의 Node들을 traversing하도록 구현하였습니다. new Node가 들어갈 위치를 찾았다면, 해당 위치에 Node를 삽입하고, 포인터를 연결해주며, 해당 위치가 rear인 경우. pq의 rear를 가리키는 포인터가 new Node를 가리키도록 하였습니다.

'deleteMin_linkedList()' 함수는 먼저 최솟값을 가진 Node일 front Node를 임시로 선언한 Node 포인터 변수인 Node* curr에 저장하고 해당 노드의 data를 정수 temp 변수에 입력한 후, pq의 front pointer

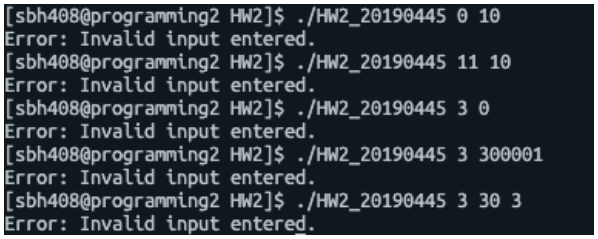


Figure 2: 예외 처리 메시지 출력 결과 예시.

가 가리키는 Node를 front의 next Node로 바꾸도록 하였습니다. 이후 curr를 free하여 원래 front Node에 할당된 메모리를 해제해 주고, temp를 return하도록 하였습니다.

1.3 HW2 code에서의 개선점

HW#1과 비교해 HW#2에서 개선된 점은 크게 두가지로 볼 수 있습니다.

1.3.1 난수 생성 과정의 변화

위에서 설명했듯이, 난수 생성 과정에는 확실한 변화가 있었습니다. HW#1에서는 매번 난수를 생성할 때마다 이제까지 생성된 수와 겹치는 것이 있는 확인하며 수를 생성해야 했기 때문에, sorting을 진행하기 이전의 준비 과정에 많은 시간이 소요되었습니다. 하지만, HW#2에서는 Knuth algorithm을 활용하여 더욱 빠른 시간 내에 난수를 생성할 수 있었습니다.

1.3.2 데이터 구조의 변화

HW#1에서는 모든 과정을 array based로 구현하였습니다. 하지만, 이런 array based approach는 메모리의 낭비를 일으키는 원인이 됩니다. 이번 HW#2에서는 Linked-list based queue를 구현함으로써 숫자의 갯수에 따른 데이터 구조의 크기를 자유롭게 조정할 수 있었습니다. 또한, 밑에서 설명할 deletion 과정의 속도를 살펴보면, Linked-list based 구현이 array based 구현에 비해 압도적으로 빠른 것을 확인할 수 있습니다. 이는 front(head)의 node에 접근하고, 이를 제거하는 과정에서 배열의 원소를 한칸씩 미는 연산이 따로 필요하지 않기 때문입니다. 비록 insertion 과정에서는 Linked-list 구현이 더 빠르다고 할 수 없는 결과가 출력되었지만, n이 작은 수일 때와, deletion 과정에서는 새로운 데이터 구조의 사용에 의한 효과를 확인할 수 있었습니다.

1.4 Error와 예외 처리

예외 처리 출력 결과의 예시는 Figure 2에서 확인할 수 있습니다. HW#2의 요구 사항 그대로, n 혹은 k의 값이 범위는 넘어가거나, 커맨드라인 인자의 갯수가 예상과 다르면 예외 처리 메시지가 출력되는 것을 확인할 수 있습니다. 이 출력 결과들은 main함수에서 두 차례의 예외 처리를 통해 이루어졌습니다.

첫 번째 예외처리는 입력된 커맨드라인 인자의 갯수를 확인하는 것으로 정했습니다. 인자의 갯수가 정확히 두 개가 아닌 경우(즉 argc의 값이 3이 아닌 경우) 예외 처리 메시지를 표시하고, main 함수에서 1

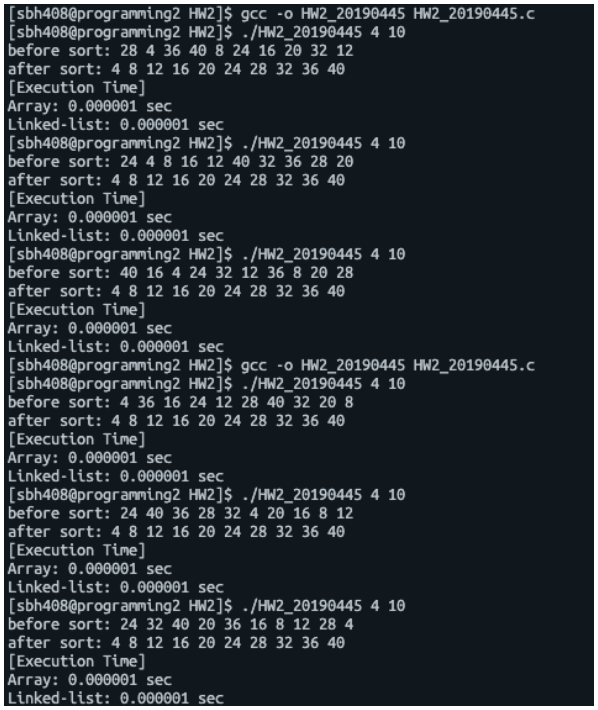


Figure 3: k = 4, n = 10일 때 출력 결과 예시.

을 반환하도록 하였습니다. 두 번째 예외처리는 HW#2 pdf에 의거해 k 혹은 n의 값이 정해진 범위 바깥의 정수이면 에러 메시지를 출력하도록 하였습니다.

2 실행 과정과 결과

본 과제의 출력 결과는 크게 두가지로 볼 수 있습니다. 먼저 n이 10 이하인 경우 sorting이 제대로 진행되었는지 확인해야 하고, sorting이 수행되는 시간이 적절이 출력되었는지 확인하여야 합니다. pdf의 필수 측정 요소에서는 특히 k = 4, n = 10인 경우 두 방식(Array와 Linked-list)의 결과와 k=4. n=300, 3000, 30000, 300000인 경우 각 함수의 평균적인 수행 시간을 비교하라고 되어 있었습니다. 저는 여기에 각 방식의 insertion 시간과 deletion 시간을 추가적으로 측정하여 더욱 정확한 분석을 하고자 하였습니다.

2.1 k = 4, n = 10일 때 Queue와 Sorting Algorithm의 정상 동작 확인

위의 Figure 3를 통해 n이 10 이하일 때 sorting과 queue의 동작이 정상적으로 작동하는 것을 확인할 수 있습니다. 여러 차례 결과를 출력해 보았을 때 난수가 적절히 생성되어 겹치는 수열이 나타나지 않는 것을 확인할 수 있고, sorting 결과 역시 정상적으로 출력되는 것을 볼 수 있습니다. 또한 Figure 3에서 중간에 한번 다시 compile되는 것을 볼 수 있는데, 이는 after sorting결과를 array-based pq의 list L을 출력하도록 한 것에서 Linked-list-based pq의 list L을 출력하도록 한 코드로 교체하여 다시 실행한 것입니다. 그 결과, 두 구현 방식 모두에서 정상적인 결과가 출력되는 것을 확인할 수 있습니다.

```
[sbh408@programming2 HW2]$ bash script_total
for 4 300
Average execution time for Array_insert: .000063 seconds
Average execution time for Array_delete: .000107 seconds
Average execution time for Array: .000170 seconds

Average execution time for Linked-list_insert: .000084 seconds
Average execution time for Linked-list_delete: .000004 seconds
Average execution time for Linked-list: .000089 seconds

for 4 3000
Average execution time for Array_insert: .005768 seconds
Average execution time for Array_delete: .036850 seconds
Average execution time for Array: .042619 seconds

Average execution time for Linked-list_insert: .045917 seconds
Average execution time for Linked-list_delete: .000047 seconds
Average execution time for Linked-list: .045965 seconds

for 4 30000
Average execution time for Array_insert: 2.433147 seconds
Average execution time for Array_delete: 4.123695 seconds
Average execution time for Array: 6.556842 seconds

Average execution time for Linked-list_insert: 6.621439 seconds
Average execution time for Linked-list_delete: .000615 seconds
Average execution time for Linked-list: 6.622055 seconds
```

(a) n = 30000까지의 알고리즘 수행시간 출력 결과.

```
[sbh408@programming2 HW2]$ bash script_300000
for 4 300000
Average execution time for Array_insert: 1019.906061 seconds
Average execution time for Array_delete: 1512.586796 seconds
Average execution time for Array: 2532.492856 seconds

Average execution time for Linked-list_insert: 4589.940240 seconds
Average execution time for Linked-list_delete: 0.007996 seconds
Average execution time for Linked-list: 4589.948236 seconds
```

(b) 내림차순

Figure 4: n = 300000의 알고리즘 수행시간 출력 결과.

	array	insertion		Linked-list	insertion
300	0.000063		300	0.000084	
900	0.000535		900	0.000672	
3000	0.005768		3000	0.045917	
9000	0.316833		9000	0.706572	
30000	2.433147		30000	6.621439	
300000	1019.90606		300000	4589.940244	
	array	deletion		Linked-list	deletion
300	0.000107		300	0.000004	
900	0.000916		900	0.000012	
3000	0.03685		3000	0.000047	
9000	0.60329		9000	0.000143	
30000	4.123695		30000	0.000615	
300000	1512.586796		300000	0.007996	
	array	total		Linked-list	total
300	0.00017		300	0.000089	
900	0.001451		900	0.000684	
3000	0.042619		3000	0.045965	
9000	0.920124		9000	0.706716	
30000	6.556842		30000	6.622055	
300000	2532.492856		300000	4589.948236	
	array	insertion		Linked-list	insertion
300	0.000168		300	0.000088	
3000	0.015902		3000	0.009717	
30000	1.576504		30000	1.525227	
300000	156.851713		300000	562.788824	

Figure 5: 알고리즘 수행시간 출력 결과표.

2.2 k = 4, n = 300, 3000, 30000, 300000 인 경우 평균적인 수행 시간 측정 결과

저는 HW#1때와 동일하게, 프로그램의 평균 수행시간을 측정하기 위해 Listing 5와 같은 shell script를 활용하였습니다. 위의 스크립트를 각 n에 대해 실행한 결과는 Figure 4과 같습니다. 각 경우에 대한 서로 다른 구현 방식의 insertion, deletion, total 수행 시간이 모두 출력되어 있는 것을 확인할 수 있습니다. 그 결과는 Figure 5과 같은 표로 정리하였습니다.

3 알고리즘 수행시간 측정, 비교 및 분석

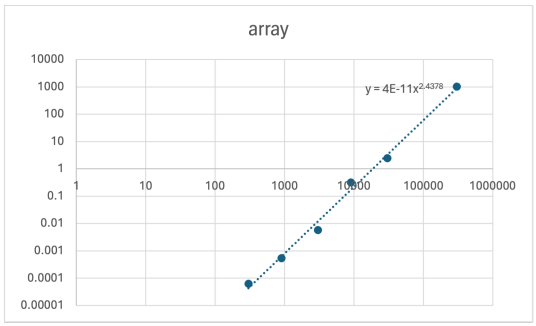
3.1 알고리즘 수행시간 측정 및 비교

알고리즘 수행 시간의 측정은 위의 섹션에서 소개한 바와 같이 Shell script를 통해 진행하였고, 그 결과를 그래프로 plot 한 결과는 Figure 6와 같습니다. x축과 y축 모두 범위가 크기 때문에 모든 그래프의 양 축을 로그 스케일로 하여 plot하였습니다. 또한, 정확한 경향성 파악을 위해 n = 900인 때와 n = 9000일 때를 추가하여 결과를 출력해 보았습니다.

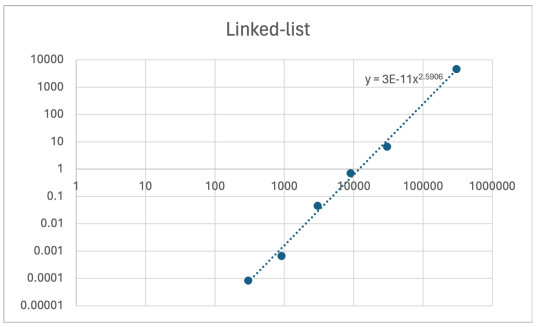
최종 결과를 내기 이전, key와 data를 하나의 변수로 취급하고 코드를 구현하여 결과를 출력해 본 결

```bash
#!/bin/bash

total_array=0
total_linked_list=0
total_array_insert=0
total_linked_list_insert=0
total_array_delete=0
total_linked_list_delete=0

# Iterate 50 times
for ((i=0; i<20; i++)); do
    # Execute the program and capture the output
    output=$(./HW2_20190445 4 300)

    # Extract execution times for array and linked list operations
    array_time=$(echo "$output" | awk '/Array:/ {print $2}')
    linked_list_time=$(echo "$output" | awk '/Linked-list:/ {print $2}')

    array_time_insert=$(echo "$output" | awk '/Array_insert:/ {print $2}')
    linked_list_time_insert=$(echo "$output" | awk '/Linked-list_insert:/ {print $2}')

    array_time_delete=$(echo "$output" | awk '/Array_delete:/ {print $2}')
    linked_list_time_delete=$(echo "$output" | awk '/Linked-list_delete:/ {print $2}')

    # Add execution times to total
    total_array=$(echo "scale=6; $total_array + $array_time" | bc)
    total_linked_list=$(echo "scale=6; $total_linked_list + $linked_list_time" | bc)

    total_array_insert=$(echo "scale=6; $total_array_insert + $array_time_insert" | bc)
    total_linked_list_insert=$(echo "scale=6; $total_linked_list_insert + $linked_list_time_insert" | bc)

    total_array_delete=$(echo "scale=6; $total_array_delete + $array_time_delete" | bc)
    total_linked_list_delete=$(echo "scale=6; $total_linked_list_delete + $linked_list_time_delete" | bc)
done

# Calculate averages
average_array_insert=$(echo "scale=6; $total_array_insert / 20" | bc)
average_linked_list_insert=$(echo "scale=6; $total_linked_list_insert / 20" | bc)

average_array_delete=$(echo "scale=6; $total_array_delete / 20" | bc)
average_linked_list_delete=$(echo "scale=6; $total_linked_list_delete / 20" | bc)

average_array=$(echo "scale=6; $total_array / 20" | bc)
average_linked_list=$(echo "scale=6; $total_linked_list / 20" | bc)

# Print the results
echo "for 4 300"
echo "Average execution time for Array_insert: $average_array_insert seconds"
echo "Average execution time for Array_delete: $average_array_delete seconds"
echo -e "Average execution time for Array: $average_array seconds\n"

echo "Average execution time for Linked-list_insert: $average_linked_list_insert seconds"
echo "Average execution time for Linked-list_delete: $average_linked_list_delete seconds"
echo -e "Average execution time for Linked-list: $average_linked_list seconds\n"
```

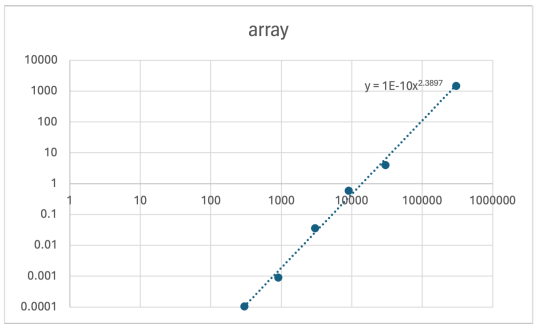
Listing 5: Linux Shell Script for Calculating Average Execution Time.



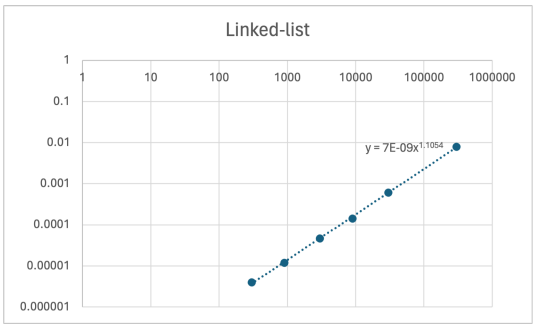
(a) Array Insertion



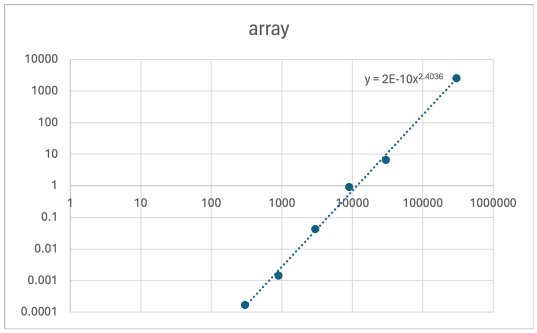
(b) Linked List Insertion



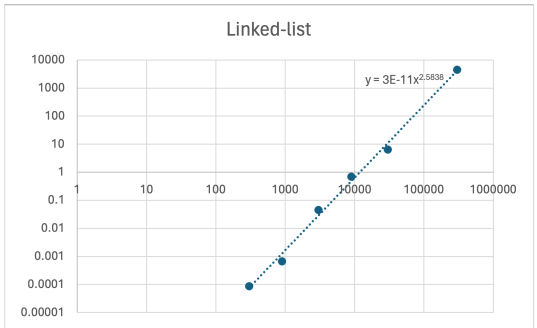
(c) Array Deletion



(d) Linked List Deletion



(e) Array Total

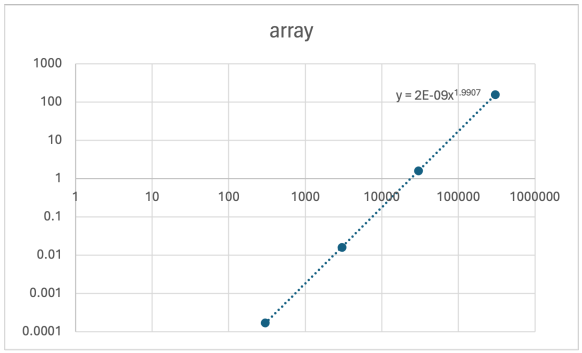


(f) Linked List Total

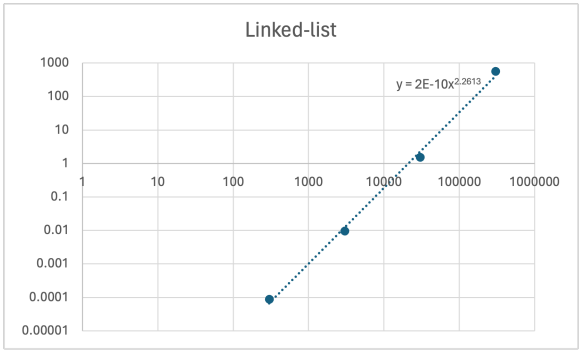
Figure 6: 알고리즘 수행시간 plot 결과.

```
[sbh408@programming2 HW2]$ bash script
Average execution time for Array for 300: .000168 seconds
Average execution time for Linked-list for 300: .000088 seconds\n
Average execution time for Array for 3000: .015902 seconds
Average execution time for Linked-list for 3000: .009717 seconds\n
Average execution time for Array for 30000: 1.576504 seconds
Average execution time for Linked-list for 30000: 1.525227 seconds\n
Average execution time for Array for 300000: 157.851713 seconds
Average execution time for Linked-list for 300000: 562.788824 seconds\n
```

(a) 초안의 출력 결과.



(b) 초안의 array time plot.



(c) 초안의 Linked-list time plot.

Figure 7: 코드 초안의 결과.

과들은 Figrues 7와 같습니다. 이 결과를 추가로 첨부하는 이유는 서버 사용자의 증가에 의해서인지 data 변수의 추가로 인한 연산 복잡도의 증가로 인해서인지 급격히 실행 시간이 늘었기 때문입니다. 하지만 그러한 차이점에도 불구하고 밑의 n과 실행 시간 간의 관계와 array와 Linked-list based pq의 차이에 대한 분석은 유효합니다.

3.2 결과 분석

위의 결과에서 확인하고, 분석할 수 있는 것은 크게 두가지가 있습니다.

3.2.1 각각 Array based implementation과 Linked-list Implementation의 Insertion, Deletion, Total Execution Time과 n의 관계

Figure 6를 확인해 보면, 모든 그래프가 로그 스케일에서 직선의 형태를 띠는 것을 확인할 수 있습니다. 해당 그래프들의 추세선을 그리고, 그 식을 구한 결과(모두 Figure 6을 확대하면 확인 가능합니다), Execution time은 n의 거듭제곱으로 표현되는 것을 확인할 수 있습니다. 주목할 점은 Linked-list의 deletion을 제외한 모든 그래프의 실행 시간이 n의 2제곱을 조금 넘는 값에 비례한다는 점입니다. Linked-list의 deletion 만이 유일하게 n에 linear한 것에 가까운 실행 시간을 가지는 것을 볼 수 있습니다. 이는 각 구현 방식의 insertion과 deletion 함수가 구현 되어 있는 방식에서 원인을 찾아볼 수 있습니다.

Array의 경우에는 insertion과 deletion은 n번씩 반복되며, average case에서는 매 insertion과 deletion은 O(n)의 time complexity를 가집니다. 그 이유는 매 deletion과 insertion마다 array 내의 모든 원소들을 한 칸씩 밀어 주어야 하기 때문입니다. 그렇기 때문에 각각 insertion, deletion, 그리고 total execution time이 n의 2.x 거듭제곱으로 표현되는 것은 자연스럽습니다. 정확히 n의 제곱에 비례하지 않는 것은 흔히 algorithm time complexity를 고려할 때 무시하는 연산들인 비교 연산과 요소에 접근하는 연산이 영향을

끼친 것으로 보입니다.

하지만 Linked-list의 경우에는 흥미로운 결과를 확인할 수 있습니다. 이론적으로 Linked-list의 insertion과 deletion은 모두 O(1)의 time complexity를 가집니다(priority를 고려할 경우 O(n)). 배열의 원소들을 밀어주는 연산이 필요 없이, 위치에 그대로 삽입이 가능하기 때문입니다. Deletion의 경우에는 실행 시간이 n의 1.x 거듭제곱으로 표현되기에, 실제 실행 시간과 이론적인 time complexity의 차이라고 생각할 수 있지만, insertion의 경우에는 Array의 경우와 마찬가지로 n의 2.x 거듭제곱으로 표현되고, 이는 실험과 이론의 차이라기엔 너무 큰 차이라고 보여집니다. 이를 분석하기 위해 다시 Listing 4로 돌아가 보면, 매 insertion마다 한번씩 반복되는 동작과 평균적으로 n에 비례하는 횟수만큼 반복되는 동작을 몇가지 찾아볼 수 있습니다. 먼저 'create_node' 함수는 매 iteration마다 한번씩 반복되며, 연산 시간에 영향을 미칠 것으로 보입니다. 하지만, 이 함수는 전체 과정이 n의 제곱에 비례하는 실행 시간을 가지는 것은 설명하지 못합니다. 그렇다면 남은 것은 두가지 동작인데, node를 pointer를 통해 traversing하는 동작과, node마다 data에 접근하여 이를 value와 비교하는 동작입니다. 두 동작 모두 평균적으로 매 iteration마다 약 n/2번 반복되는 동작이며, 수업시간에 다룬 데로 Linked-list가 caching의 효율성을 해쳐 메모리에의 접근을 어렵게 만든다면, Linked-list based pq의 insertion이 약 $O(n^{2.x})$의 time complexity를 가지는 것을 설명할 수 있습니다.

3.2.2 두 pq 구현 방식 간의 실행 시간 차이

deletion 시간을 비교해 보면, 예상대로 Linked-list가 좋은 성능을 보이는 것을 확인할 수 있습니다. 하지만 Insertion 시간에서는 다소 예상하지 못한 결과가 나왔는데, 이는 위에서 분석하였듯이, caching 성능의 차이로 인해 Node의 data에 접근하고, 수를 비교하는 연산과 Node를 create하는 함수의 time clomplexity가 요소로 작용하여 매우 큰 n에서는 Linked-list가 좋지 않은 성능을 보인다고 해석할 수 있습니다.