Data structure: Assignment 2

Seung-Hoon Na

November 23, 2018

1 Assignment 2

1.1 셸 정렬 (shell sort)

본 절에서는 셸 정렬을 구현하는 것을 목표로 한다. **셸 정렬** (shell sort)는 **삽입 정렬**(insertion sort)을 개선한 것으로, 한칸씩 원소 교환을 하는 것이 아니라, 여러 칸 간격으로 떨어진 원소들간의 교환을 수행한ㄷ. Sequence가 h-**sorted** (h-정렬) 되었다라고 함은, 전체적으로 정렬된 것은 아니나 h**칸 단위로 정렬된 상태**를 의미한다. 다음은 3-정렬된 sequence예를 보여준다[1].

```
2 1 0 4 3 5 8 6 7 9 10

2 --- 4 --- 8 --- 9
  1 --- 3 --- 6 --- 10
    0 --- 5 --- 7
```

셸 정렬에서는 40-정렬, 13-정렬, 4-정렬, 1-정렬 순으로 갈 수도 있고, 15-정렬, 7-정렬, 4-정렬, 1-정렬 순으로 갈수도 있으나, 마지막은 항상 1-정렬이어야 한다.

본 절에서는 특정한 간격 순을 따르는 셸 정렬을 구현하고자 한다. 보다 구체적으로 셸 정렬이, 주어진 **간격비** $K > 1$에 대해, $h^{(m)}$-정렬, $h^{(m-1)}$-정렬, $\cdots$, $h^{(0)}$-정렬 순으로 간다고 하자. 이때, $h^{(0)}, \cdots, h^{(m)}$을 **Gap sequences** (간격 열)라고 한다.

본 절의 셸 정렬에서는 $h^{(i)}$는 다음을 따른다.

$$h^{(i)} = K^{i-1} + K^{i-2} + \cdots + K + 1 = \frac{K^i - 1}{K - 1} \tag{1}$$

단, $h^{(0)} = 1$이고, $K = 2$ 또는 $K = 3$으로 한정된다 [2].

또한, $h^{(m)}$이 주어진 배열의 데이터 갯수 n을 초과하지 않아야 한도록, m이 정해져야 한다. 다시 말해, 셸 정렬에서 h-정렬을 수행하는 총 단계수 m은 다음과 같다.

$$m = max\left\{i | h^{(i)} \leq n\right\} \tag{2}$$

[1] 다시 말해, h-sorted (h-정렬된) sequence란 h *interleaved sorted subsequences*라고 한다.

[2] $K = 2$인 경우의 참고문헌은 다음을 참조하시오.

Hibbard, Thomas N. (1963). "An Empirical Study of Minimal Storage Sorting". Communications of the ACM. 6 (5): 206–213. doi:10.1145/366552.366557.

$K = 3$인 경우의 참고문헌은 다음을 참조하시오.

Pratt, Vaughan Ronald (1979). Shellsort and Sorting Networks (Outstanding Dissertations in the Computer Sciences). Garland. ISBN 0-8240-4406-1.

(실제 구현은 더욱 단순하다).

다음은 배열의 길이 n과 간격비 K의 서로 다른 셋팅에 대한 셸 정렬의 진행 순서에 대한 예이다.

1. $n = 150$, $K = 3$인 경우, Eq. (2)에 따라 $m = 5$이다. 즉, 셸 정렬은 총 5단계를 거치며 ($m = 5$), 121-정렬, 40-정렬, 12-정렬, 4-정렬, 1-정렬 순으로 진행된다.

2. $n = 70$, $K = 2$인 경우, Eq. (2)에 따라 $m = 6$이다. 즉, 셸 정렬은 총 6단계를 거치며 ($m = 6$), 63-정렬, 31-정렬, 15-정렬, 7-정렬, 3-정렬, 1-정렬 순으로 진행된다.

1.1.1 셸 정렬 (shell sort) 구현 (c++) 및 테스트

위의 셸 정렬을 위해 삽입 정렬 알고리즘을 일반화하여 주어진 배열 `A`을 시작위치 `start`로부터 `h`칸 단위 정렬된 상태로 만드는 `insertionsort`를 정의하고 이를 `shellsort` 내부에서 `h`를 조정해가며 반복적으로 호출하도록 아래 두 함수를 구현하시오 (`shellsort.h, shellsort.cpp` 작성).

```
void insertionsort(int *A, int size, int start, int h);
void shellsort(int *A, int size, int K);
```

단, `shellsort`의 세 번째 인자 K가 1로 주어지는 경우에는 일반 삽입정렬 (insertion sort)을 수행하면 된다 (즉, $m = 1$로 1-정렬만 수행하고 리턴).

구현된 shell sort를 **랜덤 데이터** 및 **입력파일**을 받아 테스트를 수행하는 테스트 코드도 함께 제출하시오 (`test_shellsort.cpp` 작성). 간단히, `argc`로 인자가 있으면 입력 파일로부터 테스트, 그렇지 않으면 랜덤 데이터를 생성하여 테스트를 수행하도록 할 것.

입력파일의 예는 다음과 같다 (`input.txt`).

```
2 1 21 4
16 5 22 8 22 6
```

리눅스 ubuntu 16.04 이상 환경에서 컴파일되도록 하고, 코드가 여러개인 경우 `Makefile`도 함께 만들어 첨부하라.

다음은 `Makefile`의 예이다.

```
SRCS   =      shellsort.cpp
OBJS   =      $(SRCS:.cpp=.o)
TOBJS  =      $(TSRCS:.cpp=.o)

LIBS   =
CC     =      g++
CFLAGS =

test_shellsort: $(OBJS) test_shellsort.o
	${CC} $(CFLAGS) $(OBJS) test_shellsort.o $(LIBS) -o test_shellsort

.cpp.o:
	$(CC) $(CFLAGS) -c $<
```

1.1.2 셸 정렬 (shell sort) 실행시간 비교

랜덤 배열의 길이 N을 1000, 10000, 100000, 1000000 으로 증가될때, K가 1, 2, 3인 경우 각각에 대해 shell sort를 수행하는데 걸리는 실제 소요시간을 비교하는 테스트 코드 `test_shellsort_comp.cpp`를 작성하시오.

다음은 `test_shellsort_comp.cpp`의 컴파일된 프로그램의 출력예이다.

```
N=1000, K=1, elapsed_time: 0.00229502 sec
N=1000, K=2, elapsed_time: 0.000361919 sec
N=1000, K=3, elapsed_time: 0.000317097 sec
N=10000, K=1, elapsed_time: 0.152209 sec
N=10000, K=2, elapsed_time: 0.00256705 sec
N=10000, K=3, elapsed_time: 0.00183415 sec
N=100000, K=1, elapsed_time: 8.40829 sec
N=100000, K=2, elapsed_time: 0.030952 sec
N=100000, K=3, elapsed_time: 0.0260191 sec
N=1000000, K=1, elapsed_time: 842.6 sec
N=1000000, K=2, elapsed_time: 0.446693 sec
N=1000000, K=3, elapsed_time: 0.368041 sec
```

($N = 1000000$, $K = 1$일때 시간이 지나치게 오래 걸리는 경우는 $K = 2$, $K = 3$의 경우 소요시간을 출력하시오.)

여기서, 소요시간을 측정을 위해 `gettimeofday`를 사용하시오. 사용예는 아래 예제 코드`test_dtime.cpp`에 제시되어 있다.

```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <sys/time.h>
#include <unistd.h>


using namespace std;

double getdtime()
{
       struct timeval tv = {0};
       double dtime;

       gettimeofday(&tv, NULL);
       dtime = tv.tv_sec + (tv.tv_usec / 1000000.0);
       return dtime;
}



int main(int argc, char **argv){
   double oldtime = getdtime();

   for(int i=0;i<1000;i++){
      usleep(1000);
   }

   double elapsed_time = getdtime() - oldtime;
```

```
  cerr << "elapsed_time: " << elapsed_time << " sec" << endl;
  return 1;
}
```

1.1.3 셸 정렬 (shell sort): Sedgewick Upper Bound

Sedgewick은 셸 정렬의 Gap sequences $h^{(1)}, \cdots, h^{(m)}$로 다음을 제안하였다.

$$h^{(i)} = \begin{cases} 4^i + 3 \cdot 2^{i-1} + 1 & \text{if } i \geq 1 \\ 1 & \text{otherwise} \end{cases} \tag{3}$$

위의 Sedgewick의 Gap sequences를 이용한 셸 정렬 `shellsort_sedgewick`을 구현하시오 (`shellsort_sedgewick.cpp` 또는 기존 파일에 포함).

```
void shellsort_sedgewick(int *A, int size);
```

마찬가지로, 랜덤 배열의 길이 N을 1000, 10000, 100000, 1000000 으로 증가될때, K가 3인 경우의 `shellsort` (Pratt방법)와 위의 `shellsort_sedgewick` (Sedgewick방법)을 수행하는데 걸리는 실제 소요시간을 비교하는 코드 `test_shellsort_sedgewick_comp.cpp`를 작성하시오.

다음은 `test_shellsort_sedgewick_comp.cpp`의 컴파일된 프로그램의 출력예이다.

```
N=1000, Pratt, K=3, elapsed_time: 0.000324011 sec
N=1000, Sedgewick, elapsed_time: 6.19888e-05 sec
N=10000, Pratt, K=3, elapsed_time: 0.00490284 sec
N=10000, Sedgewick, elapsed_time: 0.000871897 sec
N=100000, Pratt, K=3, elapsed_time: 0.041738 sec
N=100000, Sedgewick, elapsed_time: 0.00625396 sec
N=1000000, Pratt, K=3, elapsed_time: 0.378449 sec
N=1000000, Sedgewick, elapsed_time: 0.050101 sec
```

1.1.4 셸 정렬 (shell sort): 계산복잡도

다음 문헌을 참조하여, 셸 정렬을 위한 Gap sequences로 **Pratt방법** (Eq. (1)에서 $K = 3$)과 **Sedgewick방법** (Eq. (3)을 사용하여 진행할때의 계산 복잡도를 각각 **빅 오**로 표시하시오.

`http://nlp.jbnu.ac.kr/DS2018/shellsort.pdf`

1.2 정렬 알고리즘 구현 및 비교: 삽입 정열, 셸 정렬, 병합정렬, 퀵 정렬 비교 (c++)

1.2.1 병합정렬, 퀵 정렬 구현 및 테스트

앞 절의 삽입정렬 및 셸 정렬에 추가로, 아래 병합정렬, 퀵 정렬을 수행하는 함수 `mergesort`, `quicksort`를 각각 구현하고, 랜덤 입력에 대해서 각 정렬 알고리즘을 테스트하시오 (`mergesort.cpp`, `quicksort.cpp`작성).

```
void mergesort(int *A, int first, int last);
void quicksort(int *A, int first, int last);
```

1.2.2 삽입정렬, 병합정렬, 퀵 정렬 비교 (c++)

랜덤 배열의 길이 N을 1,000, 10,000, 100,000, 1,000,000, 10,000,000 으로 증가될때, 셸 정렬의 **Pratt방법** (Eq. (1)에서 $K = 3$)과 **Sedgewick방법** (Eq. (3), 병합정렬 및 퀵 정렬을 수행하는데 걸리는 실제 소요시간을 비교하는 테스트 코드 `test_sort_comp.cpp`를 작성하시오. (N이 10,000,000일때의 경우도 적용해볼 것).

다음은 `test_sort_comp.cpp`의 컴파일된 프로그램의 실행예이다.

```
N=1000, Shellsort-Pratt, K=3, elapsed_time: 0.000314951 sec
N=1000, Shellsort-Sedgewick, elapsed_time: 0.000309944 sec
N=1000, Mergesort, elapsed_time: 0.000330925 sec
N=1000, Quicksort, elapsed_time: 0.000251055 sec
N=10000, Shellsort-Pratt, K=3, elapsed_time: 0.00479412 sec
N=10000, Shellsort-Sedgewick, elapsed_time: 0.00402093 sec
N=10000, Mergesort, elapsed_time: 0.00374484 sec
N=10000, Quicksort, elapsed_time: 0.00298095 sec
N=100000, Shellsort-Pratt, K=3, elapsed_time: 0.0406458 sec
N=100000, Shellsort-Sedgewick, elapsed_time: 0.0294759 sec
N=100000, Mergesort, elapsed_time: 0.018712 sec
N=100000, Quicksort, elapsed_time: 0.0148151 sec
N=1000000, Shellsort-Pratt, K=3, elapsed_time: 0.378368 sec
N=1000000, Shellsort-Sedgewick, elapsed_time: 0.276325 sec
N=1000000, Mergesort, elapsed_time: 0.223466 sec
N=1000000, Quicksort, elapsed_time: 0.175819 sec
N=10000000, Shellsort-Pratt, K=3, elapsed_time: 5.93688 sec
N=10000000, Shellsort-Sedgewick, elapsed_time: 3.49171 sec
N=10000000, Mergesort, elapsed_time: 2.56164 sec
N=10000000, Quicksort, elapsed_time: 2.01255 sec
```

1.3 기수 정렬 (radix sort)

1.3.1 셈 정렬 (count sorting) 구현 (c++)

셈 정렬 (counting sort)는 입력 원소 각각이 0부터 k사이에 있는 정수라고 가정한다. 다음 셈 정렬을 위한 함수 `countingsort`의 코드를 구현하시오 (`countingsort.cpp`파일 구현). (단 완성된 `countingsort`는 안정된 정렬결과를 보장해야 한다)

```
void countingsort(int *A, int size, int k)
```

랜덤 입력에 대해 구현된 `countingsort`의 동작을 테스트하는 코드 `test_countingsort`도 함께 작성하시오.

1.3.2 랜덤 token 생성기 구현

단어 길이 (word length)에 대한 확률 분포 테이블 파일을 입력받아, 최대 W개 소문자알파벳으로 구성된 단어를 랜덤하게 N개를 생성하여 출력하는 **랜덤 token 생성기**를 구현하시오. W와 N의 설정은 코드에서는 일반화시키도록 하고, 본 절에는 $W = 20$, $N = 100000$으로 고정시킨다. (이때, 생성되는 N개의 단어는 중복을 허용하는 토큰 (token)단위라고 가정한다.)

입력 단어 길이 확률 분포 테이블 파일은 다음과 같다 (`word_length_stat.txt`).

- 참고: http://norvig.com/mayzner.html

```
1  2.998
2  17.651
3  20.511
4  14.787
5  10.700
6  8.388
7  7.939
8  5.943
9  4.437
10 3.076
11 1.761
12 0.958
13 0.518
14 0.222
15 0.076
16 0.020
17 0.010
18 0.004
19 0.001
20 0.001
21 0.000
22 0.000
23 0.000
```

랜덤 token 생성기로부터 출력된 결과를 별도 파일로 저장한다 (tokens.txt). 출력된 결과의 예는 다음과 같다.

```
...
cssj
jxw
u
fzu
kj
aen
op
nn
of
tq
qvfhi
fj
v
cf
n
rfqxz
...
```

1.3.3 MSD 기수 정렬 구현 (c++)

교과서 9장 11절의 내용에 따라 **MSD 기수 정렬**을 수행하는 프로그램 ratio_sort을 작성하시오 (radio_sort.cpp 작성).

상세 요구사항은 다음과 같다.

1. **MSD 기수 정렬 구현**: 정렬되지 않은 token리스트를 파일로 입력받아, "MSD 기수 정렬"을 수행하여 정렬된 token리스트를 최종적으로 출력하는 프로그램 코드 `radio_sort.cpp`를 작성한다. 위에서 구현한 `countingsort`를 이용한다.

2. **랜덤 token생성기 결과 파일에 적용**: 앞서 랜덤 token 생성기로부터 얻은 `tokens.txt`을 파일을 입력받아, MSD 기수 정렬을 수행한 결과 파일 `tokens.sorted`를 출력하시오.

3. **추가 테스트**: N을 10000, 1000000등 다른 셋팅하에서, 랜덤 token 생성기를 수행하여 얻은 다른 입력파일에 대해서도 작성된 MSD 기수 정렬 프로그램을 테스트하시오.

1.3.4 MSD 기수 교환 정렬 구현 (c++)

교과서 9장 11절의 내용에 따라 **MSD 기수 교환 정렬** (Radix exchange sort)를 수행하는 프로그램 `ratio_exchange_sort`을 작성하시오 (`radio_exchange_sort.cpp` 작성).

상세 요구사항은 다음과 같다.

1. **MSD 기수 교환 정렬 구현**

2. **랜덤 token생성기 결과 파일에 적용**: 마찬가지로, 앞서 랜덤 token 생성기로부터 얻은 `tokens.txt`을 입력 파일로 하여, MSD 기수 교환 정렬을 수행한 결과된 파일 `tokens.exhange_sorted`를 출력하라.

3. **MSD 기수 정렬 및 MSD 기수 교환 정렬 비교**: 추가로, 동일 입력 파일에 대해 MSD 기수 정렬과 MSD 기수 교환 정렬 알고리즘을 수행하였을때 소요시간을 비교하시오.

1.4 이진 탐색 트리 (binary search tree)

본 절에서는 이진탐색트리를 사용하여 주어진 텍스트 파일에 나타나는 각 **단어(word)의 빈도수**를 계산하여 저장하는 프로그램을 작성하고자 한다.

각 단어의 빈도수 계산을 위해 `dataRecord` 및 `treeRecord`는 다음과 같이 정의한다.

```
typedef struct dataRecord
{
  string Key;
  int count;
} dataType;

typedef struct treeRecord
{
  dataType Data;
  struct treeRecord* LChild;
  struct treeRecord* RChild;
} node;

typedef node* Nptr;
```

1.4.1 이진 탐색 트리: 탐색, 삽입,삭제,갱신 구현 (c++)

교과서 10장 7절의 내용을 참조하여, 주어진 key에 대해 이진 탐색 트리 상에서 탐색, 삽입, 삭제를 수행하는 Search, Insert, Delete, Update함수를 각각 구현하고 테스트 하시오 (BST.ccp, test_BST.cpp 작성)

```
Nptr Search(Nptr T, const char *key);
Nptr Insert(Nptr T, const char *key);
void Delete(Nptr &T, const char *key);
void Update(Nptr &T, const char *key);
```

여기서, Update함수는 이진 탐색 트리 상에서 주어진 key를 갖는 노드가 있으면 해당 노드의 count를 증가시키고, 그렇지 않으면 count를 1로 하는 새로운 노드를 트리상에서 삽입시키는 함수이다 (앞의 BST.cpp에 추가). (Search와 Insert를 별개로 수행하는 형태로 구현하지 말고, 하나의 함수로 통합되어야 한다)

1.4.2 이진 탐색 트리를 이용한 Word Count 계산

주어진 텍스트 파일로부터 이진탐색트리를 이용하여 모든 단어의 word count를 계산하고, **전위 순회 (preorder)를 통해 key로 정렬된 word count결과를 출력**하는 프로그램 BST_word_count.cpp를 작성하시오.

구동 테스트를 위해 다음 두 입력 파일 각각에 대해 모든 단어 token이 이진탐색트리상에서 Update가 되었다면, 전위 순회를 수행하여 해당 결과를 각각 The-Road-Not-Taken.wordcount, Dickens_Oliver_1839.wordcount에 저장하시오.

http://nlp.jbnu.ac.kr/DS2018/data/The-Road-Not-Taken.tokens.txt
http://nlp.jbnu.ac.kr/DS2018/data/Dickens_Oliver_1839.tokens.txt

예를 들어, The-Road-Not-Taken.tokens.txt파일내에 있는 모든 단어 token를 이진탐색트리상에서 Update시킨후에 전위 순회를 통해 최종 Word count출력하면 다음과 같다.

```
! 1
, 10
. 3
: 2
; 2
a 3
about 1
ages 2
all 1
and 9
another 1
as 5
back 1
be 2
because 1
bent 1
better 1
black 1
both 2
by 1
claim 1
...
```

1.4.3 Word Count결과로부터 이진탐색 (binary search)

위의 **Word count출력결과를 정렬된 순서대로 로딩한 후**, 콘솔로부터 사용자 입력을 받아 해당 입력단어를 key로 하는 **이진 탐색 (binary search)**을 수행하여 **해당 count를 출력**하는 프로그램의 코드 BST_word_count_test.cpp를 작성하시오. (이진탐색트리를 다시 재구성하지 않고, 배열상 이진 탐색을 수행함을 유의하라)

마찬가지로, 구동 테스트를 위해, 앞 문항에서 얻은 두 종류의 Word count출력결과 파일 The-Road-Not-Taken.wordcount와 Dickens_Oliver_1839.tokens.wordcount를 입력 데이터로하여, 이진 탐색의 정상 작동 여부를 확인하여야 한다.

실행 예는 다음과 같다.

```
> BST\_word\_count\_test The-Road-Not-Taken.wordcount
Loading is complete    <= program's message
input> and
9
input> a
3
input> abcdef
Not found
...
```

1.4.4 Word Count결과로부터 이진탐색 (binary search): 대용량 데이터로 확장방안

앞 문항에서 Word count출력 결과 (n-gram에 대한 count등)가 대용량인 경우에는 주어진 데이터를 모두 메모리상에 로딩할 수 없다.

이러한 대용량 데이터상에서 사용자 입력에 대한 count를 리턴하기 위해, 로딩 및 탐색을 효율적으로 개선시킬 수 있는 방법은 무엇인지 제안하시오.

(제안 방법이 대용량 데이터상에서 효율을 크게 높일 수 있음을 밝히시오.)

1.5 해시 (hash)

본 절에서는 해시를 사용하여 주어진 텍스트 파일에 나타나는 각 **단어 (word)의 빈도수**를 계산하여 저장하는 프로그램을 작성하고자 한다.

별도 체인 (separate chaining)을 이용한 해시를 사용하고, 이를 위해 다음 자료구조를 정의한다.

```
typedef struct dataRecord
{
  string Key;
  int count;
} dataType;

typedef struct nodeRecord
{
  dataType Data;
  struct nodeRecord* next;
} node;

typedef node* Nptr;
```

1.5.1 해시 클래스 구현 (c++)

위의 dataRecord에 특화된 별도 체인 (separate chaining)기반 해시 자료구조로 다음과 Hash class를 선언한다. 주어진 Hash class의 멤버함수를 모두 구현하시오 (Hash.h, Hash.cpp).

구동 테스트를 위한 코드도 함께 작성하시오.

```
class Hash
{
public:
  int size;
  Nptr *table;

public:
  void Create(int tablesize);
  int  Insert(const char *key);
  Nptr Search(const char *key);
  void Update(const char *key);
  void Delete(const char *key);
  void Save(const char *filename);
  void Open(const char *filename);
  void Rehash(int newtablesize);
}
```

각 함수에 대한 설명은 다음과 같다.

- Create: tablesize로 하는 해시 테이블을 생성한다.
- Insert: 주어진 key를 갖는 노드가 없으면 해당 key를 취하고 count를 1로 하는 새로운 노드를 해시에 삽입시킨다.
- Search: 주어진 key를 갖는 노드가 없으면 NULL, 있다면 노드의 포인터를 리턴한다.
- Update: 주어진 key를 갖는 노드가 있으면 해당 노드의 count를 증가시키고, 그렇지 않으면 주어진 key로 count를 1로 하는 새로운 노드를 해시에 삽입시킨다. 이진 탐색트리의 Update와 마찬가지로, Search한후 해당 key노드가 존재하지 않으면 Insert를 호출하는 방식이 아닌, 필요한 내용이 Update 내에서 외부 함수 호출없이 내부 완결성을 갖추는 방식으로 구현할 것.
- Save: 해시테이블을 binary또는 text파일로 저장한다. 차후 빠르게 loading 할수 있도록 파일 포맷을 잘 설계해야 한다.
- Open: 저장한 해시테이블 파일을 읽어들여 메모리상으로 loading한다.
- Rehash: newtablesize로 재 해시를 수행한다.

문자열에 대한 해시함수로 본인만의 문자열 폴딩 방식을 이용하거나, 그렇지 않으면 다음의 코드를 사용하라.

```
int strhashfunc(const char *key)
{
    char *str = (char*)key;
    unsigned int hashval = 0;
```

```
    for (hashval = 0; *str; str++)
        hashval = *str + (hashval << 5) - hashval;
    return hashval;
}
```

1.5.2 해시를 이용한 Word count계산 및 사용자 테스트

해쉬를 이용하여 주어진 텍스트 파일에 있는 각 단어의 word count를 계산하고, 계산된 결과를 해쉬 파일로 저장하는 Hash_word_count.cpp를 작성하시오.

마찬가지로, 또한 구동 테스트를 위해 다음 두 가지 입력 파일에 대해 테스트를 수행하고, 각 파일내 모든 단어 token이 Update되어 각 단어의 word count가 해시내에서 계산된 경우, 해시테이블의 내용을 각각 The-Road-Not-Taken.hash, Dickens_Oliver_1839.hash에 저장하시오..

추가로, 저장된 해시 파일 The-Road-Not-Taken.hash, Dickens_Oliver_1839.hash를 로딩하여 해시를 메모리에 올린후, 사용자로부터 단어를 입력단어 Search를 수행하여 해당 단어가 있으면 count를 그러지 않으면 Not found를 출력하는 코드 Hash_word_count_test.cpp를 작성하시오.

1.5.3 해시와 이진탐색트리 효율 비교

Dickens_Oliver_1839.tokens.txt상에서 N개의 랜덤 key에 대ㅎ Search를 호출할 때 해시와 이진탐색트리상에서의 소요시간을 비교하시오 ($N = 100000$이상).

1.6 힙 정렬 (Heap sort)

주어진 데이터에 대한 힙 정렬을 수행하고자 한다.

1.6.1 하향식 힙 구성

교과서 11장 4절의 내용을 참조하여 힙 정렬을 위해 주어진 배열에 대해 **하향식 힙 구성**을 수행하는 다음 build_heap함수를 구현하시오.

또한, 랜덤 입력 배열에 대해 build_heap동작을 테스트하시오.

```
void build\_heap(int *A, int size);
```

1.6.2 힙 정렬

힙 정렬은 구성된 힙으로부터 가장 우선순위가 큰 루트노드를 계속 삭제하면서 이루어진다. 힙 삭제과정은 제일 마지막 요소를 루트노드에 덮어씌운후, 루트로부터 시작해 힙 모습을 되찾는 **Down heap**을 통해 이루어진다.

다음 힙 삭제함수 remove_heap를 구현하고, 이를 이용하여 힙 정렬을 수행하는 코드 heap_sort.cpp를 작성하시오.

```
int remove\_heap(int *A, int size);
```

추가로, 랜덤 입력 데이터에 대해 구현된 힙 정렬 프로그램을 테스트하시오.

1.7 제출 내용 및 평가 방식

코드는 **c++** (특별한 요구사항이 있을 경우 **java**)로 작성하도록 하고, **c++**프로그램의 경우 구동os환경은 **ubuntu 16.04 LTS** 이상을 원칙으로 한다. 본 과제 결과물로 필수적으로 제출해야 내용들은 다음과 같다.

- **코드 전체**
- **테스트 결과**: 각 내용별 테스트 코드 및 해당 로그 파일
- **결과보고서**: 코드 설계(클래스 계층도 등),구현 방법 및 실행 결과를 요약한 보고서

본 과제의 평가항목 및 배점은 다음과 같다.

- **각 세부내용의 구현 정확성 및 완결성 (80점)**
- **코드의 Readability 및 체계성 (10점)**
- **결과 보고서의 구체성 및 완결성 (10점)**