

1.1 Shell Sort

1.1.1 셸 정렬 구현 및 테스트

코드 설계

Insertion sort :

배열의 원소를 start부터 h간격으로 하나씩 선택하여 이전의 배열에서 알맞은 위치를 찾아
내어 해당 위치에 원소를 삽입한다.

Shell sort:

입력 size와 K로 m을 구한다.

h를 구해 간격이 h인 Insertion sort를 시작위치를 달리하여 h번 시행하고, 이를 m번 반복한
다.

코드 구현

Insertion sort :

인자 start를 배열의 첫 검사 위치 Pick로 설정한다.

삽입할 위치를 찾을 변수 Current를 생성하여 Pick을 대입한 후, 삽입 위치가 나올 때까지 h
의 간격으로 값을 오른쪽으로 민다. Current는 h씩 감소시킨다.

삽입할 위치에 Pick에 위치한 값을 삽입한다.

Shell sort:

K == 1인 경우, h = 1로 하는 Insertion sort를 한번 실행한다.

K != 1인 경우, size와 K를 인자로 m을 구한다.

m번만큼 h를 구한 후, 구한 h마다 start를 0부터 h-1까지 설정한 Insertion sort를 h번 실행
한다.

m을 구하는 함수는 다음과 같이 동작한다.

정수형 변수 $i = 1$, $newN = size * (K - 1) + 1$ 을 선언한다.

K^i 가 newN보다 작을 동안 i를 1씩 증가시킨다.

i에서 1을 뺀 값을 반환한다.

실행

```
~/cpp/code# make
```

```
~/cpp/code# ./test_shellsort input.txt
```

```
2 1 21 4 16 5 22 8 22 6
Input(K) >>2
Shell Sort :
1 2 4 5 6 8 16 21 22 22
```

Makefile :

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

LIBS    =
CC      =      g++
CFLAGS  =      -lm -std=c++11

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 실행시간 비교

실행

```
~/cpp/code# ./test_shellsort_comp
N=1000, K=1, elapsed_time: 0.000638008 sec
N=1000, K=2, elapsed_time: 0.000127077 sec
N=1000, K=3, elapsed_time: 0.000102997 sec
N=10000, K=1, elapsed_time: 0.0861449 sec
N=10000, K=2, elapsed_time: 0.00216603 sec
N=10000, K=3, elapsed_time: 0.00223708 sec
N=100000, K=1, elapsed_time: 7.37726 sec
N=100000, K=2, elapsed_time: 0.033684 sec
N=100000, K=3, elapsed_time: 0.0272629 sec
N=1000000, K=1, elapsed_time: 726.921 sec
N=1000000, K=2, elapsed_time: 0.457554 sec
N=1000000, K=3, elapsed_time: 0.381688 sec
```

1.1.3 셸 정렬 : Sedgewick

코드 설계

Sedgewick 방식으로 h 를 구하는 방법을 적용한 코드를 새로 작성한다.

코드 구현

shell_sort_sedgewick

size를 인자로 m 을 구한 후, m 번만큼 h 를 구한다.

구한 h 마다 start를 0부터 $h-1$ 까지 설정한 Insertion sort를 h 번 실행한다.

m 을 구하는 함수는 다음과 같이 동작한다.

정수형 변수 $i = 1$ 을 선언한다.

$(4^i) + 3 * (2^{(i-1)}) + 1$ 이 인자 size보다 작을 동안 i 를 1씩 증가시킨다.

i 에서 1을 뺀 값을 반환한다

실행

```
~/cpp/code# ./test_shellsort_sedgewick_comp
N=1000, Pratt, K=3, elapsed_time: 0.000130892 sec
N=1000, Sedgewick, elapsed_time: 9.98974e-05 sec
N=10000, Pratt, K=3, elapsed_time: 0.00161314 sec
N=10000, Sedgewick, elapsed_time: 0.00161099 sec
N=100000, Pratt, K=3, elapsed_time: 0.032228 sec
N=100000, Sedgewick, elapsed_time: 0.0217018 sec
N=1000000, Pratt, K=3, elapsed_time: 0.356745 sec
N=1000000, Sedgewick, elapsed_time: 0.296851 sec
```

1.1.4 셸 정렬 : 계산 복잡도

Pratt의 셸 정렬 방식의 계산 복잡도는 $O(N^{3/2})$ 이고

Sedgewick방법의 계산 복잡도는 $O(N^{4/3})$ 이다.

Sedgewick의 방법이 약간 더 빠름을 알 수 있다.

1.2 정렬 알고리즘 구현 및 비교

1.2.1 병합정렬, 퀵 정렬 구현 및 테스트

코드 설계

병합 정렬

배열을 원소들로 나누어 원소를 두 개씩 합치는 작업을 재귀적으로 반복하여 정렬한다.

퀵 정렬

피벗을 정한 뒤, 피벗을 기준으로 데이터를 이동시키는 작업을 반복하여 정렬한다.

코드 구현

병합정렬

mergesort 함수는 배열을 반으로 나누어 각각의 배열에 대해 mergesort 를 실행한 후,

Merge 함수를 통해 나누어진 배열을 합치며 정렬한다.

Merge 함수는 두 배열을 합쳐서 정렬하는 함수이다.

퀵정렬

quicksort 함수는 Partition 함수로부터 피벗 인덱스를 받아서 피벗 인덱스 이전의 배열과 피벗 인덱스 이후의 배열에 대해 각각 quicksort 를 실행한다.

Partition 함수는 배열의 마지막 요소를 피벗으로 하여 피벗보다 작은 값은 피벗의 앞에, 피벗보다 큰 값은 피벗의 뒤에 위치시키고 피벗 인덱스를 반환한다.

실행

```
Random Ary :
33 16 13 10 25 85 54 74 58 70 52 46 59 92 67
10 36 94 17 71 85 66 79 34 64 62 19 59 66 64
86 99 80 52 10 57 37 16 32 95 39 84 94 50 28
61 60 16 55 30
Mergesort :
10 10 10 13 16 16 16 17 19 25 28 30 32 33 34
36 37 39 46 50 52 52 54 55 57 58 59 59 60 61
62 64 64 66 66 67 70 71 74 79 80 84 85 85 86
92 94 94 95 99
Quicksort :
10 10 10 13 16 16 16 17 19 25 28 30 32 33 34
36 37 39 46 50 52 52 54 55 57 58 59 59 60 61
62 64 64 66 66 67 70 71 74 79 80 84 85 85 86
92 94 94 95 99
```

1.2.2 삽입정렬, 병합정렬, 퀵 정렬 비교

실행 결과

```
~/cpp/code# ./test_sort_comp
N=1000, Shellsort-Pratt, K=3, elapsed_time: 0.000115871 sec
N=1000, Shellsort-Sedgewick, elapsed_time: 0.000102997 sec
N=1000, Mergesort, elapsed_time: 0.000141859 sec
N=1000, Quicksort, elapsed_time: 0.00011301 sec
N=10000, Shellsort-Pratt, K=3, elapsed_time: 0.00191498 sec
N=10000, Shellsort-Sedgewick, elapsed_time: 0.00154877 sec
N=10000, Mergesort, elapsed_time: 0.00171494 sec
N=10000, Quicksort, elapsed_time: 0.00149202 sec
N=100000, Shellsort-Pratt, K=3, elapsed_time: 0.0258732 sec
N=100000, Shellsort-Sedgewick, elapsed_time: 0.0207589 sec
N=100000, Mergesort, elapsed_time: 0.0224509 sec
N=100000, Quicksort, elapsed_time: 0.021045 sec
N=1000000, Shellsort-Pratt, K=3, elapsed_time: 0.377721 sec
N=1000000, Shellsort-Sedgewick, elapsed_time: 0.26179 sec
N=1000000, Mergesort, elapsed_time: 0.264569 sec
N=1000000, Quicksort, elapsed_time: 0.224079 sec
N=10000000, Shellsort-Pratt, K=3, elapsed_time: 6.42276 sec
N=10000000, Shellsort-Sedgewick, elapsed_time: 3.9397 sec
N=10000000, Mergesort, elapsed_time: 3.37565 sec
N=10000000, Quicksort, elapsed_time: 2.59668 sec
```

1.3 기수 정렬

1.3.1 셸 정렬 구현

코드 설계

배열의 데이터를 인덱스로 하여 빈도를 계산한 뒤, 빈도를 통해 정렬된 위치를 계산하여 정렬하는 countingsort 함수를 정의한다.

코드 구현

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

배열 원소의 최대값이 k 이므로 $k + 1$ 크기인 배열 Count 를 생성하고 모든 원소를 0 으로 배열의 원소를 순차적으로 탐색하며 Count 의 각 원소가 해당하는 인덱스의 값을 1 씩 증가시킨다. 또한 임시 배열을 만들어 배열의 원소를 복사한다.

Count 의 각 원소를 이전의 원소와 더해 누계를 구한다.

임시 배열에서 값을 뽑아내어 Count 의 인덱스로 하여 정렬 위치를 구한 후, 해당 위치에 값을 대입한다. Count 의 원소에 1 을 차감한다.

실행

```
Random Array Generated!
32 30 21 20 38 3 39 3 22 33 27 10 46 28 11
24 20 11 11 16 33 45 10 20 47 29 49 18 32 1
18 17 33 40 39 22 45 28 27 19 11 7 31 9 35
42 33 6 6 47
Counting sort :
1 3 3 6 6 7 9 10 10 11 11 11 11 16 17
18 18 19 20 20 20 21 22 22 24 27 27 28 28 29
30 31 32 32 33 33 33 33 35 38 39 39 40 42 45
45 46 47 47 49
```

1.3.2 랜덤 token 생성기 구현

코드 설계

누적분포함수를 계산하는 함수 CDFGenerator,
누적분포함수를 받아 길이를 정하는 함수 RandomLengthGenerator,
무작위 알파벳을 생성하는 함수 RandomCharGenerator,
무작위 토큰을 생성하는 함수 RandomTokenGenerator,
토큰을 txt 로 저장하는 함수 SaveTokenTXT 를 정의한다.

코드 구현

CDFGenerator

확률 배열을 순차적으로 탐색하며 탐색 위치와 이전의 모든 확률을 더한 누적분포함수를 생성한다.

RandomLengthGenerator

누적분포함수를 인자로 받아 100 이하의 무작위 값을 생성하여 누적분포함수와 비교하여 길이를 생성한다.

RandomCharGenerator

'a' 에서 'z'까지의 무작위 알파벳 하나를 생성한다.

RandomTokenGenerator

확률 분포 테이블을 입력받아 CDFGenerator 을 실행해 누적분포함수를 구한다.

구한 누적분포함수를 인자로 RandomLengthGenerator 을 실행해 길이를 생성한다.

길이만큼 RandomCharGenerator 을 실행해 알파벳을 구해 토큰을 완성한다.

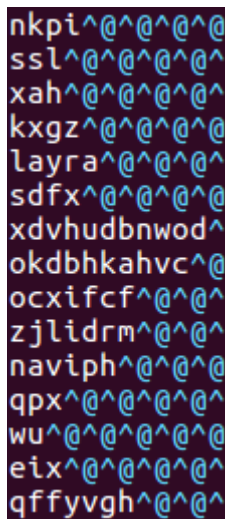
SaveTokenTXT

한 줄에 토큰 하나를 저장하여 txt 파일을 생성한다.

실행

```
# ./test_randomtokengenerator word_length_stat.txt
```

tokens.txt :



```
nkpi^@^@^@^@  
ssl^@^@^@^@  
xah^@^@^@^@  
kxgz^@^@^@^@  
layra^@^@^@^@  
sdfx^@^@^@^@  
xdvhudbnwod^  
okdbhkahvc^@  
ocxifcf^@^@  
zjlidrm^@^@  
naviph^@^@^@  
qpx^@^@^@^@  
wu^@^@^@^@  
eix^@^@^@^@  
qffyvgh^@^@
```

1.3.3 MSD 기수 정렬 구현

코드 설계

배열과 인덱스를 받아 해당 인덱스로 셈 정렬을 수행하는 CountingSort-Token, 셈 정렬을 수행한 뒤 해당 인덱스의 같은 문자 그룹에 대해 재귀함수를 호출하는 RadioSort 함수를 정의한다.

코드 구현

CountingSort-Token

알파벳의 수 + 1 크기의 배열을 생성하고 모든 원소를 0 으로 초기화한다. 1 을 더하는 이유는 NULL 인 경우를 포함하기 때문이다.

배열의 원소를 순차적으로 탐색하여 Count 의 각 원소가 해당하는 인덱스의 값을 1 씩 증가시킨다. 또한 임시 배열을 만들어 배열의 원소를 복사한다.

Count 의 각 원소를 이전의 원소와 더해 누계를 구한다.

임시 배열에서 값을 뽑아내어 Count 의 인덱스로 하여 정렬 위치를 구한 후, 해당 위치에 토큰을 대입한다. Count 의 원소에 1 을 차감한다.

RadioSort

index 를 인자로 받아 index 번째 원소에 대해 CountingSort_Token 을 수행한다.

index 번째 원소를 순차적으로 탐색하여 같은 문자 그룹에 대해 다음 index 번째 원소에 대해 RadioSort 를 수행한다.

실행

```
# ./radio_sort tokens.txt
```

tokens.sorted :

```
zyygdkc  
zyztgq  
zz  
zz  
zzazrh  
zzbx  
zzd  
zzd  
zzdi  
zzh  
zzogne  
zzttwimk  
zzu  
zzujuk  
zzy
```

1.3.4 MSD 기수 교환 정렬 구현

코드 설계

피벗을 구해 피벗을 기준으로 피벗보다 작은 원소는 피벗의 앞에, 큰 원소는 피벗의 뒤에 위치시키는 Partition,

Token 에 대해 퀵 정렬을 수행하는 QuickSort_Token 함수를 정의한다.

코드 구현

```
void QuickSort_Token(char **T, int first, int last, int index)
```

index 번째 자리의 문자에 대해 Partition 함수를 수행하여 피벗과 같은 문자 그룹의 시작 인덱스와 끝 인덱스를 받아온다.

이를 통해 피벗 이전 그룹, 피벗 그룹, 피벗 다음 그룹이 생성된다.

피벗 그룹에 대해서는 토큰의 index + 1 번째 자리의 문자에 대해 QuickSort-Token 을 수행한다.

피벗 이전 그룹에 대해서는 index 번째 자리의 문자에 대해 QuickSort-Token 을 수행한다.

피벗 다음 그룹에 대해서는 index 번째 자리의 문자에 대해 QuickSort-Token 을 수행한다.

QuickSort-Token 을 처음 실행할 때는 first 에 0, last 에 배열의 크기 - 1, index 에 0 을 넣으면 된다.

```
void Partition(char **T, int first, int last, int index, int &firstP, int lastP)
```

배열의 마지막 요소를 피벗으로 설정하고 업 포인터를 first, 다운 포인터를 last 로 설정한다.

업 포인터의 원소가 p 보다 작을 동안 업 포인터를 증가시키고, 다운 포인터의 원소가 p 보다 클 동안 다운 포인터를 감소시킨다.

다운 포인터가 업 포인터보다 같거나 크다면 두 원소를 교환한다.

두 원소가 같은 원소이면서 피벗과 같은 경우, 업 포인터 + 1 에서부터 last 까지의 원소를 한 칸씩 앞으로 당긴 후, 업 포인터에 있던 원소를 last 에 삽입하고 다운 포인터를 감소시킨다.

이 작업을 통해 배열의 last 부근에는 피벗 그룹이 생성된다.

두 원소가 같은 원소이면서 피벗과 다른 경우, low 를 하나 증가시킨다.

위 작업을 업 포인터가 다운 포인터보다 작거나 같을 동안 시행한다.

이 작업이 끝나면 피벗보다 작은 그룹, 피벗보다 큰 그룹, 피벗 그룹 순으로 그룹이 형성되어 있을 것이다.

피벗 그룹을 피벗보다 작은 그룹과 피벗보다 큰 그룹 사이에 위치시킨 후, 함수를 종료한다.

실행

```
# ./radio_exchange_sort tokens.txt
zyygdkc
zyztgq
zz
zz
zzazrh
zzbx
zzd
zzd
zzdi
zzh
zzogne
zzttwink
zzu
zzujuk
zzy
```


1.4 이진 탐색 트리

1.4.1 이진 탐색 트리 : 탐색, 삽입, 삭제, 갱신 구현

코드 설계

Search : 입력 key 와 동일한 key 를 갖고 있는 Tree 의 데이터를 찾아 해당 Tree 를 반환한다.

Insert : 입력 Key

코드 구현

Search : Tree 가 비었으면 NULL 을 반환

Tree 의 현재 데이터의 값이 key 와 동일하면 현재 Tree 반환

Tree 의 현재 데이터의 값이 key 보다 클 경우, Tree 의 Left Child 를 인자로 Search

Tree 의 현재 데이터의 값이 key 보다 작을 경우, Tree 의 Right Child 를 인자로 Search

Insert : Tree 가 비었으면 count = 1 인 새 노드 추가

Tree 의 현재 데이터의 값이 key 보다 클 경우, Tree 의 Left Child 를 인자로 Insert

Tree 의 현재 데이터의 값이 key 보다 작을 경우, Tree 의 Right Child 를 인자로 Insert

Tree 반환

Update : Tree 가 비었으면 count = 1 인 새 노드 추가

Tree 의 현재 데이터의 값이 key 와 동일하면 해당 Tree 의 데이터의 count 증가

Tree 의 현재 데이터의 값이 key 보다 클 경우, Tree 의 Left Child 를 인자로 Update

Tree 의 현재 데이터의 값이 key 보다 작을 경우, Tree 의 Right Child 를 인자로 Update

Delete : Tree 가 비었으면 삭제할 Key 가 없다고 출력

Tree 의 현재 데이터의 값이 key 보다 클 경우, Tree 의 Left Child 를 인자로 Delete

Tree 의 현재 데이터의 값이 key 보다 작을 경우, Tree 의 Right Child 를 인자로 Delete

Tree 의 현재 데이터의 값이 key 와 동일하면 현재 Tree 제거

현재 Tree 를 제거하는 방법은 다음과 같다.

현재 Tree 가 Leaf Node 일 경우, 현재 Tree 를 제거한다.

현재 Tree 가 Right Child 가 존재할 경우, Tree 를 제거하고 그 위치에 Right Child 를 넣는다.

현재 Tree 가 Left Child 가 존재할 경우, Tree 를 제거하고 그 위치에 Left Child 를 넣는다.

현재 Tree 가 Left Child 와 Right Child 가 존재할 경우, T->RChild, T->Data 를 인자로

SuccessorCopy 함수를 실행한다.

void SuccessorCopy(Nptr& T, datatype& DelNodeData)

T 가 삭제할 노드의 중위후속자라면, DelNodeData 의 Key 값을 T 의 Data 의 Key 값으로 설정한다.

그 후, T->RChild 를 T 로 설정하고 원래의 T 를 제거한다.

T 가 중위후속자를 갖고 있지 않다면, T->LChild, DelNodeData 를 인자로 SuccessorCopy 를 실행한다.

실행

삽입

```
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> abc
abc 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> bcd
abc 1
bcd 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> def
abc 1
bcd 1
def 1
```

검색

```
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 3
Key(string) >> efg
No Such Node
abc 1
bcd 1
def 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 3
Key(string) >> def
result :
    Key : def
    Count : 1
abc 1
bcd 1
def 1
```

갱신

```
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> abc
abc 2
bcd 1
def 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> def
abc 2
bcd 1
def 2
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> efg
abc 2
bcd 1
def 2
efg 1
```

제거

```
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 2
Key(string) >> bcd
abc 2
def 2
efg 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 2
Key(string) >> fgh
No Record with Such Key
abc 2
def 2
efg 1
```

1.4.2 이진 탐색 트리를 이용한 Word Count 계산

코드 설계

파일을 입력받아 Tree 로 만드는 함수 MakeTree,
Tree 를 저장하는 함수 SaveWordCount,
Tree 를 중위순회로 저장하는 함수 InOrderSave 를 정의한다.

코드 구현

```
void MakeTree(ifstream& fin, Nptr& T)
```

char item[20]; fin >> item 을 통해 단어를 입력받은 후, Update(Tree, item)을 통해 중복된 단어는 count 를 증가시키고 새로운 단어는 노드로 추가한다.

```
void SaveWordCount(Nptr T, string fileName)
```

InOrderSave(fout, T)함수를 실행하여 Tree 를 정렬하여 저장한다.

```
void InOrderSave(ofstream& fout, Nptr T)
```

중위순회를 통해 T를 저장한다. 이 결과는 정렬된 T와 동일하다.

실행

```
# ./BST_word_count The-Road-Not-Taken.tokens.txt
```

출력 :

```
was 1
way 2
wear 1
where 1
with 1
wood 2
worn 1
yellow 1
yet 1
Saved!
```

```
# ./BST_word_count Dickens_Oliver_1839.tokens.txt
```

출력 :

```
yourself 51
yourselves 3
youth 8
youthful 6
youths 1
z 1
zealous 2
zenith 1
Saved!
```

1.4.3 Word Count 결과로부터 이진탐색

코드 설계

파일을 읽어 배열로 만드는 함수 MakeAry,

이진탐색을 수행하는 함수 BinarySearch 를 정의한다.

코드 구현

MakeAry 함수는 파일의 줄을 읽어 " "를 기준으로 " " 이전의 값은 Key, 이후의 값은 count 에 대입하여 배열에 저장한다.

BinarySearch 함수는 배열의 검색 범위를 반으로 줄여 나가면서 key 를 탐색하여 key 의 index 를 반환한다.

실행

```
# ./BST_word_count_test The-Road-Not-Taken.wordcount
```

```
Loading is complete
input (Exit : "EXIT")> and
9
input (Exit : "EXIT")> a
3
input (Exit : "EXIT")> abcdef
Not found
input (Exit : "EXIT")> taken
1
input (Exit : "EXIT")> not
2
```

```
# ./BST_word_count_test Dickens_Oliver_1839.wordcount
```

```
Loading is complete
input (Exit : "EXIT")> and
5239
input (Exit : "EXIT")> a
3760
input (Exit : "EXIT")> abcdef
Not found
input (Exit : "EXIT")> oliver
864
input (Exit : "EXIT")> not
747
```

1.4.4 Word Count 결과로부터 이진탐색 : 대용량 데이터로 확장방안

현재 작성된 트리는 맨 처음 읽은 값을 루트 노드로 설정한 다음, 그 값을 변경하지 않고 노드를 추가해 나가는 방식이다. 이 방식은 트리의 균형이 깨지기 쉽고, 데이터가 많아지면 필요한 메모리가 그만큼 늘어난다는 문제를 갖고 있다.

B+ Tree 는 노드에 데이터를 저장하지 않고 데이터 레코드를 가리키는 포인터를 저장한다. 노드 하나에서 나오는 최대 링크 수를 설정할 수 있어 이 값을 적절하게 잡으면 트리의 높이를 크게 줄일 수 있다. 노드의 삽입과 삭제 시, 트리의 균형을 유지한다.

B+ Tree 를 사용한다면 모든 데이터를 메모리에 올릴 필요 없이 검색한 데이터가 존재하는 레코드만 메모리에 올릴 수 있기에 메모리의 사용이 크게 줄어든다. 또한 삽입, 삭제 시에도 트리의 균형이 크게 변하지 않는다.

따라서 B+ Tree 를 사용한다면 대용량 데이터에서의 효율이 크게 올라갈 것이다.

1.5 해시

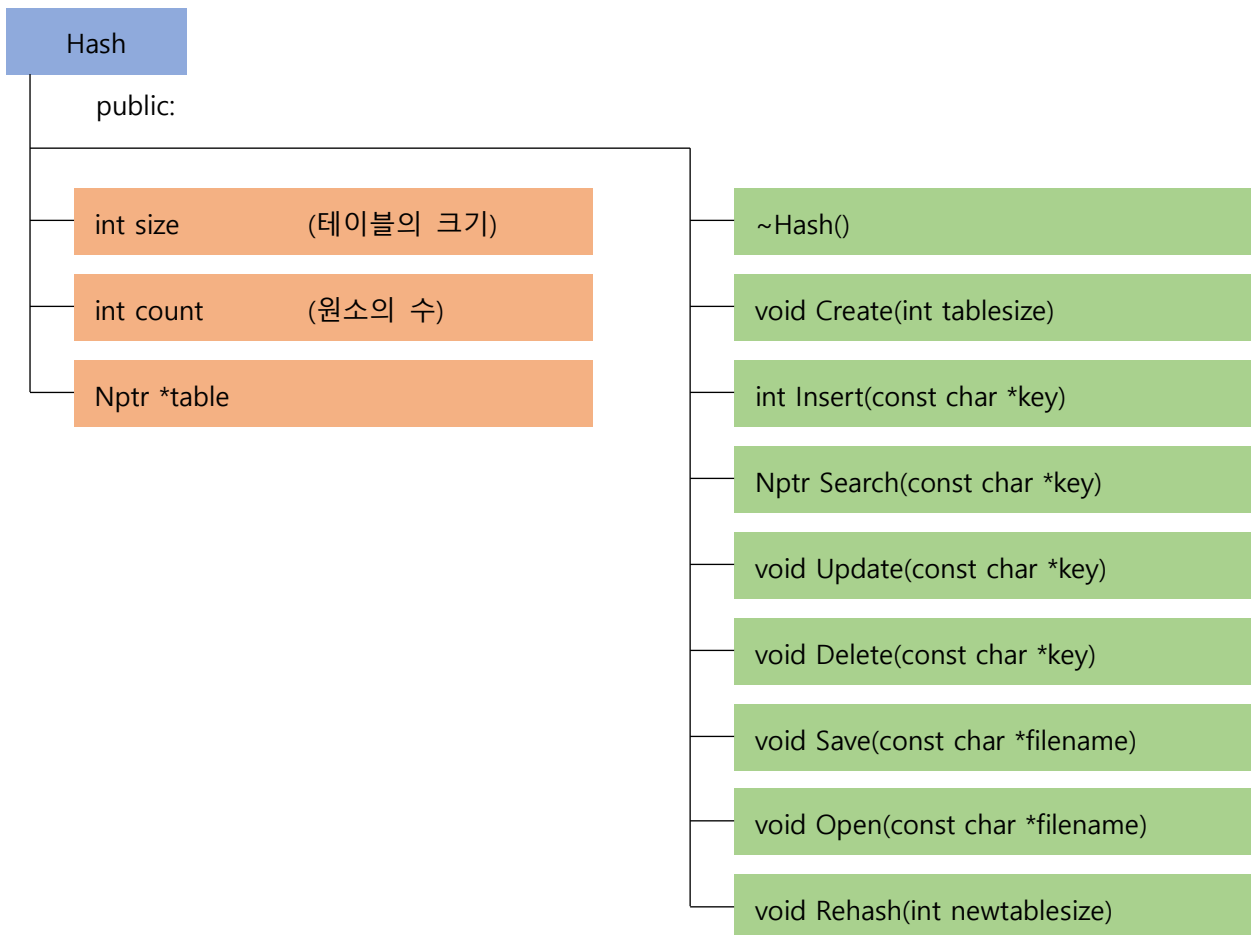
1.5.1 해시 클래스 구현

코드 설계

~Hash() 함수를 통해 프로그램이 종료될 때, Hash 에 할당된 공간을 해제한다.
table 을 새로 생성하는 Create,
key 를 갖는 노드가 없을 경우 노드를 table 에 추가하는 Insert,
table 에서 key 를 갖는 노드를 검색하는 Search,
key 를 갖는 노드가 없을 경우 노드를 추가하고 존재할 경우 해당 노드의 count 를
증가시키는 Update,
해당 key 를 갖는 노드를 제거하는 Delete,
Hash Table 을 저장하는 Save,
Hash Table 을 불러오는 Open,
Table 의 모든 노드에 대해 Rehash 를 수행하는 Rehash 함수를 정의한다.

코드 구현

클래스 계층도



~Hash()

테이블의 원소를 순차적으로 탐색하며 원소가 비어있지 않을 경우, 해당 원소에 연결된 모든 노드를 제거함으로써 hash 를 소멸시킨다.

void Create(int tablesize)

size 를 인자 tablesize 로 설정하고 size 크기의 table 을 생성한 후, table 의 모든 원소를 NULL 로 설정한다.

int Insert(const char *key)

Search 함수를 통해 key 를 갖는 노드가 있는지 탐색한 후, 노드가 존재하지 않을 경우 table[hash]에 노드를 삽입한다.

table 의 반절 이상이 찬 경우, size 를 두 배로 키워 Rehash 를 수행한다.

노드를 삽입했을 경우, count 를 하나 증가시키고 1 을 반환한다.

노드가 이미 존재했을 경우, 0 을 반환한다.

Nptr Search(const char *key)

key 의 hash 를 구한 후, table[hash]에 해당 key 가 있는지 탐색하여 존재할 경우 해당 노드의 포인터를, 존재하지 않을 경우 NULL 을 반환한다.

void Update(const char *key)

key 의 hash 를 구한 후, table[hash]가 비었다면 새 노드를 삽입한다.

table[hash]가 존재할 경우, 노드 연결을 탐색하며 동일한 key 가 존재할 경우 해당 Data 의 count 를 증가시킨다.

동일한 key 가 존재하지 않을 경우, 연결된 노드의 마지막에 새 노드를 삽입한다.

노드를 삽입했을 경우, count 를 하나 증가시킨다.

table 의 반절 이상이 찬 경우, size 를 두 배로 키워 Rehash 를 수행한다.

void Delete(const char *key)

key 의 hash 를 구한 후, table[hash]가 비었다면 오류를 출력한다.

table[hash]가 존재할 경우, key 를 갖는 노드가 첫 노드가 아니고 다음 노드가 존재한다면 이전 노드의 next 를 다음 노드로 설정한다.

다음 노드가 존재하지 않는다면 이전 노드의 next 를 NULL 로 설정한다.

key 를 갖는 노드가 첫 노드이고 다음 노드가 존재한다면 table[hash]를 다음 노드로 설정한다.

다음 노드가 존재하지 않는다면 table[hash]를 NULL 로 설정한다.

노드를 삭제했다면 count 를 하나 감소시킨다.

void Save(const char *filename)

파일의 첫 줄에 table 의 크기를 저장한다.

table 을 순차적으로 탐색하면서 원소가 존재할 경우, 연결된 모든 노드를 탐색하며 각 노드마다 "<key> <count>"형식으로 한 줄을 저장한다.

원소가 존재하지 않을 경우, 빈 줄을 저장한다.

void Open(const char *filename)

파일의 첫 줄을 읽어 hash table 의 크기를 구한다.

파일의 줄이 비어있을 경우 건너뛰다.

파일의 줄이 비어있지 않을 경우, key 와 count 를 구하고 hash 를 구한다.

table[hash]를 탐색하여 해당 원소가 비어있을 경우, key 와 count 를 갖는 노드를 삽입한다.

해당 원소가 비어있지 않을 경우, 연결된 노드의 마지막에 key 와 count 를 갖는 노드를 삽입한다.

void Rehash(int newtablesize)

원래의 테이블은 tempTable 배열을 새로 생성하여 저장해 놓은 뒤, Create 함수를 통해 newtablesize 크기의 테이블을 새로 생성한다.

tempTable 을 순차적으로 탐색하며 원소가 존재할 경우, 연결된 노드를 모두 탐색하며 각 노드의 hash 를 구해 새 테이블에 삽입한다.

탐색한 노드는 공간을 반환한다.

원소가 존재하지 않는 경우는 건너뛰다.

rehash 작업이 끝난 후, tempTable 의 공간을 반환한다.

실행

삽입

```
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> abc
hash : 94      key : abc      count : 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> bcd
hash : 94      key : abc      count : 1
hash : 97      key : bcd      count : 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 1
Key(string) >> def
hash : 3       key : def      count : 1
hash : 94      key : abc      count : 1
hash : 97      key : bcd      count : 1
```

검색


```

Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 3
Key(string) >> efg
No Such Node
hash : 3      key : def      count : 1
hash : 94     key : abc      count : 1
hash : 97     key : bcd      count : 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 3
Key(string) >> def
result :
    Key : def
    Count : 1
hash : 3      key : def      count : 1
hash : 94     key : abc      count : 1
hash : 97     key : bcd      count : 1

```

갱신

```

Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> abc
hash : 3      key : def      count : 1
hash : 94     key : abc      count : 2
hash : 97     key : bcd      count : 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> def
hash : 3      key : def      count : 2
hash : 94     key : abc      count : 2
hash : 97     key : bcd      count : 1
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 4
Key(string) >> efg
hash : 3      key : def      count : 2
hash : 6      key : efg      count : 1
hash : 94     key : abc      count : 2
hash : 97     key : bcd      count : 1

```

삭제

```

Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 2
Key(string) >> bcd
key : bcd      count : 1
hash : 3      key : def      count : 2
hash : 6      key : efg      count : 1
hash : 94     key : abc      count : 2
Insert : 1, Delete : 2, Search : 3, Update : 4, Exit : 0 >> 2
Key(string) >> fgh
No Record with Such Key
hash : 3      key : def      count : 2
hash : 6      key : efg      count : 1
hash : 94     key : abc      count : 2

```

1.5.2 해시를 이용한 Word count 계산 및 사용자 테스트

코드 설계

파일을 입력받아 hash Table 로 저장하는 TokensToHash 함수 정의

코드 구현

void TokensToHash(ifstream& fin, Hash& hash)

파일의 단어를 읽어 Update 실행.

실행

```
# ./Hash_word_count The-Road-Not-Taken.tokens.txt
```

The-Road-Not-Taken.hash :

```
200: size
back 1
if 1
made 1
undergrowth 1
as 5
in 4
oh 1
ages 2
fair 1
by 1
come 1
it 2
no 1
on 1
```

```
# ./Hash_word_count_test The-Road-Not-Taken.hash
```

```
input (Exit : "EXIT")> and
9
input (Exit : "EXIT")> a
3
input (Exit : "EXIT")> abcdef
Not found
input (Exit : "EXIT")> taken
1
input (Exit : "EXIT")> not
2
```

```
# ./Hash_word_count Dickens_Oliver_1839.tokens.txt
```

Dickens_Oliver_1839.hash :

```
12800: size
! 1447
' 4903
, 16151
- 704
. 6870
: 644
; 2642
? 1014
'' 48
a 3760
b 3
d 12
e 1
```

```
# ./Hash_word_count_test Dickens_Oliver_1839.hash
input (Exit : "EXIT")> and
5239
input (Exit : "EXIT")> a
3760
input (Exit : "EXIT")> abcdef
Not found
input (Exit : "EXIT")> oliver
864
input (Exit : "EXIT")> not
747
```

1.5.3 해시와 이진탐색트리 효율 비교

```
# ./test_BST_Hash_comp Dickens_Oliver_1839.wordcount Dickens_Oliver_1839.hash
N : 1000000
hash elapsed_time: 0.225985 sec
binary elapsed_time: 0.526311 sec
```

일반적인 상황에서 해시가 이진탐색보다 더 빠르다는 것을 알 수 있다.

1.6 힙 정렬

1.6.1 하향식 힙 구성

코드 설계

배열을 입력받아 heap 을 구성하는 함수 build_heap 을 정의한다.

코드 구현

```
void build_heap(int *A, int size)
```

size 크기의 새 배열 tempAry 를 만들어서 A 의 원소들을 모두 복사한다.

tempAry 를 순차적으로 탐색하면서 탐색한 원소를 A 에 순서대로 넣는다.

A 에 원소를 넣을 때마다 넣은 원소와 그 부모를 비교하여 원소가 부모보다 클 경우 둘을 교체한다.

교체가 일어나면 또 다시 원소의 부모와 비교를 한다. 이는 루트 노드에 도달하거나 부모 노드가 더 클 때까지 반복된다.

실행

실행 파일은 test_heap_sort.cpp 이며, 결과는 1.6.2 에서 확인한다.

1.6.2 힙 정렬

코드 설계

루트 노드에 있는 원소를 정렬하는 함수 DownHeap,
루트 노드를 제거하고 배열의 마지막 원소를 루트 노드로 설정한 후 DownHeap 을 호출하고
배열의 크기를 반환하는 함수 remove_heap 을 정의한다.

코드 구현

```
void DownHeap(int *A, int size, int current)
```

먼저, 왼쪽 자식이 존재하는지 확인한다. 존재하지 않을 경우 리프 노드라는 의미이므로
함수를 종료한다.

왼쪽 자식이 존재할 경우, 오른쪽 자식이 존재할 시 오른쪽 자식과 비교한다.

현재의 노드가 두 자식 중 큰 노드보다 작을 시, 큰 자식과 교환하고 DownHeap 함수를
재귀호출한다.

```
int remove_heap(int *A, int size)
```

배열을 받아 배열의 마지막 원소를 배열의 첫번째 자리에 위치시키고 배열의 크기를 하나
줄인다.

배열과 루트 노드의 위치인 0 을 인자로 주고 DownHeap 함수를 호출한다.

줄어든 배열의 크기를 반환하고 함수를 종료한다.

실행

```
# ./test_heap_sort
arySize >>50
MaxNum >>50
Random Array Generated!
11 31 24 40 1 10 8 44 9 0 43 5 6 31 13
31 8 43 33 27 33 34 26 42 19 33 5 19 9 5
32 21 36 6 13 37 18 24 33 27 26 28 33 32 10
48 13 18 41 49

build_heap :
49
44 48
43 43 42 32
36 37 33 40 41 31 19 24
31 13 33 33 27 33 32 34 19 33 6 5 8 9 5 13
11 21 6 8 9 18 24 31 0 26 27 28 1 10 26 13 5 18 10
```

```

remove?(Y/N) >>y
remove_heap :
48
44 42
43 43 41 32
36 37 33 40 33 31 19 24
31 13 33 33 27 33 32 34 19 10 6 5 8 9 5 13
11 21 6 8 9 18 24 31 0 26 27 28 1 10 26 13 5 18

remove?(Y/N) >>y
remove_heap :
44
43 42
37 43 41 32
36 33 33 40 33 31 19 24
31 13 18 33 27 33 32 34 19 10 6 5 8 9 5 13
11 21 6 8 9 18 24 31 0 26 27 28 1 10 26 13 5

```

기타 사항

모든 프로그램은 Ubuntu 16.04 LTS 에서의 정상 실행을 확인했으며 우분투에서 실행파일을 생성하기 위해 사용한 명령어는 다음과 같다

```
g++ -o <실행파일명> <소스코드> -std=c++11
```

우분투는 가상머신 VMware WorkStation 15 Player 에서 구동하였으며, CPU 는 i5-6500 이며 processor 은 2 개를 사용하였고, Ram 은 2GB 를 할당하였다.

Dickens_Oliver_1839.tokens.txt 의 첫 줄

d' » zoliver twist 에 유니코드 포맷 문자가 섞여 있었기에 이를 oliver twist 로 수정하여 실행하였다.

#include <sys/timeb.h> 에서 오류가 발생하여 이를 사용한 모든 코드에서 #include <sys/time.h>로 변경하였다.

1.3.3 의 countingsort 함수는 1.3.1 의 함수를 사용하지 않고 새로이 정의하였다.