# III. 설계과제 최종보고서

설계과제명 : 간단한 검색 엔진(Simple Search Engine) 구현

| | |
|---|---|
| 교과목명 | 자료구조와 실습 |
| 담당교수 | 김준태 교수님 |
| 팀 원 | 2018112005 신승윤 |
| | 2018112007 이승현 |
| | |
| | |

# 설계과제 요약서

## 과 제 요 약 서

| | |
|---|---|
| **설계과제명** | 간단한 검색 엔진(Simple Search Engine) 구현 |
| **주요기술용어 (5~7개 단어)** | 동적 배열(vector), hash, chaining |

### 1. 과제 목표

제한 요소를 고려하여 색인 및 검색 기능을 수행하기 위한 효율적인 자료구조를 설계하고 이를 활용하여 간단한 검색엔진을 구현합니다.

### 2. 수행 내용 및 방법

hash를 사용해 어떤 데이터를 넣으면 이 데이터를 **[**0,k**]** (k는 어떤 자연수) 로 mapping 합니다. 즉, 단어를 문서번호와 문서에서의 위치 벡터로 mapping → 데이터를 key, 그 단어에 해당하는 문서번호와 그 문서에서 단어의 시작위치를 value로 하는 dictionary로 구성해, key가 주어졌을 때 value로 접근할 수 있도록 합니다.

만약 hash collision이 발생할 경우, chaining을 이용해 collision을 해결합니다.

### 3. 수행 결과

찾고자 하는 단어를 입력하면 컴퓨터에 저장되어 있는 문서를 불러와 그 문서에 찾고자 하는 단어가 존재하는지 검색합니다. 만약 문서가 다수 존재한다면 각 문서별로 단어를 검색하며, 문서에 존재하는 총 단어 수, 문서 별 존재하는 검색 단어 개수를 출력합니다. 검색 단어 개수에 따라 문서별 검색 결과를 내림차순으로 정렬 후 출력하고, 검색한 단어 전후 3단어씩 검색 단어와 함께 출력합니다.

### 4. 결과 분석

예제에 주어진 모든 단어에 대해 1~3번의 비교를 통해 정확한 위치집합을 반환하였습니다. chaining을 이용하였기 때문에 평균적인 검색 속도는 1 + a/2(a는 적재밀도) 인 것을 감안하면 적당한 수치가 나온 것으로 생각됩니다.

또한 이 보고서에서는 linked list (동적배열) 을 이용한 chaining이외에 더 빠르게 비교 검색할 수 있는 방법인 레드블랙트리를 제시하고 있습니다.

# 제 1 장 서론

## 제 1 절 설계과제 목적

⇒ 검색 단어, 검색 대상 문서 파일, 색인 및 검색 결과 출력, 검색 방식에 대한 제한 요소를 고려하여 색인 및 검색 기능을 수행하기 위한 효율적인 자료구조를 설계하고 이를 활용하여 간단한 검색엔진을 구현합니다.

### <제한 요소>

- 사용자가 입력하는 단어는 하나이며, 검색 단어에서 대문자와 소문자는 같은 것으로 취급하고, 알파벳 이외의 문자는 검색 단어로 사용하지 않습니다. 검색 단어를 포함하는 단어는 검색 대상이 아니다. 즉, "inform"으로 검색할 경우 "information"이 포함된 문서는 검색하지 않는 것으로 합니다. **(검색 단어)**

- 검색 대상이 될 문서 파일의 내용은 ASCII코드의 영문자 (a~z, A~Z), 개행문자 ('\n', '\t', ' ', …) 및 문장부호 (“, ‘, ,, ., ?, …)들로 이루어져 있고, 단어와 단어 사이는 ‘ ’ (공백문자)로 구분됩니다. 공백과 개행문자들로 구분되는 문자열 중에서 영문자로만 이루어진 부분이 단어에 해당됩니다.**(검색 대상 문서 파일)**

- 색인 과정이 종료된 후에는 총 문서 수, 색인된 총 단어 수, 색인이 완료될 때까지 수행한 스트링 비교연산 횟수를 출력합니다. 이후 검색이 수행될 때마다 검색 결과와 함께 검색이 완료될 때까지 수행한 스트링 비교연산 회수를 출력합니다. 이때 검색 단어가 포함된 파일에서 가장 많이 포함된 파일의 이름이 먼저 나오도록 정렬하며, 또한 비교연산 횟수의 측정은 설계자에 의해 코드 상에서 포함되어져야 하고, 주어진 검색어가 자료구조 내의 자료들과 몇 번이나 비교가 이루어지는 지를 모두 측정해야 합니다. 이 비교연산 횟수가 대체적으로 검색 엔진의 성능(효율성)을 나타내게 됩니다. **(색인 및 검색 결과 출력)**

- 색인 및 검색을 위하여 어떤 자료구조와 알고리즘을 사용할 것인지는 설계자가 자유롭게 선택할 수 있지만, 설계 과제의 심사는 결과물로서 제출될 소프트웨어의 기능상의 ‘완성도’와 함께 ‘효율성’을 체크할 것이므로 가급적 비교연산 횟수가 작아지도록 자료구조 및 알고리즘을 설계하여야 합니다. **(검색 방식)**

## 제 2 절 설계과제 내용

구현할 검색 엔진은 네이버나 Google과 같은 일반적인 키워드 기반 검색 엔 진을 단순화한 것으로서, 영문으로 이루어진 문서들을 대상으로 간단한 검색 기능 을 수행합니다.

색인과정 : 검색 엔진은 우선 문서들을 읽어 각 문서별로 단어들이 몇 번 반복되는지에 대한 정보를 적당한 자료구조를 이용하여 저장합니다.

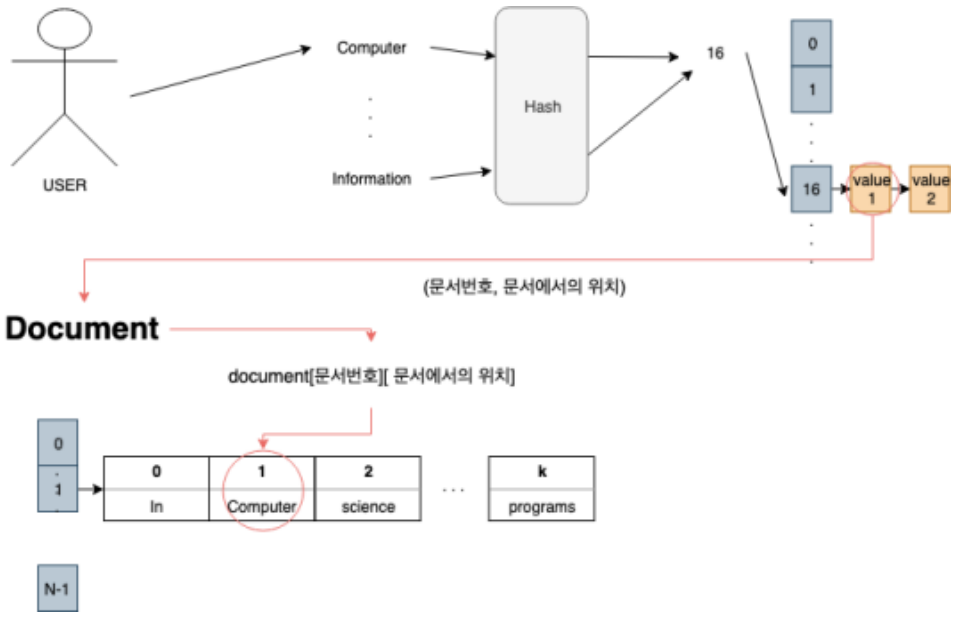
검색과정 : 이후 사용자가 검색하고 자 하는 단어를 입력하면, 미리 저장해놓은 자료구조를 활용해, 그 단어가 존재하는 문서들을 찾아 문서명, 단어 출현 빈도, 그 단어가 존재하는 문서 부분(전후 3 단어씩)을 출력합니다.

## 제 3 절 진행 일정 및 개인별 담당분야

| 성명 | 담당 분야 | 참여도(%) |
|---|---|---|
| 신승윤 | 함수 코어 부분 설계 및 보고서 작성 | 50 |
| 이승현 | 전반적인 프로그램 설계 및 보고서 작성 | 50 |
| | | |

# 제 2 장 프로그램의 구성

## 제 1 절 전체 구성도




프로그램은 우선 USER가 search_keyword를 입력합니다. 예를 들어 search_keyword가 Computer라면 computer를 받아서 hash 값을 구합니다. 이 경우에는 hash값이 16입니다. 그러면 hash table에서 16번째 주소로 갑니다. 여기에는 chaining된 다른 값이 있을 수도 있다 그렇기 때문에 선형탐색이 필요할 수도 있습니다. 그렇지만 설계지침서의 예제로 실험해보았을 때 collision은 매우 적게 발생하기 때문에 사실상 이 과정도 O(1)로 고려할 수 있습니다.

이렇게 해서 접근한 Value1은 Computer에 대한 위치집합입니다.

우리 프로그램에서는 vector <loc> 입니다. value1의 원소는 두 개의 값을 가진 튜플인데 (chaining에서 식별을 위해 구현과정에서는 사실상 1개의 필드를 더 추가함) 이를 이용해 O(1)로 문서에서 search_keyword가 있는 단어에 접근할 수 있습니다.

## 제 2 절 프로그램 세부 구성

| 파일 | 설명 |
|---|---|
| main.cpp | dictionary.h에서 만들어진 table을 이용해 사용자의 입력에 대한 searching을 수행한다. |
| dictionary.h | key-value로 정의되는 dictionary 를 만든다. |
| file.h | 파일에서 단어를 읽어와 동적배열에 할당한다. |

▶ 자료구조

우선 우리의 목표를 다시 생각해보면 예를 들어 computer 라는 단어가 주어지면 이에 대한 위치들을 반환해주어야 합니다. 우리는 그러한 위치를 location이라는 구조체로 아래와 같이 설계하였습니다.

```
typedef struct loctaion{
  string word;
  int doc;
  int index;
}loc;
```

그리고 이런 Location을 저장할 효율적인 구조도 필요합니다.

```
vector <vector<loc> > table[MOD];
```

우리는 chaining 을 이용한 table을 설계하기위해 vector<loc> 라는 특정 단어에 대한 워드 집합을 저장하는 동적배열 vector 기반 chain 을 정의하였습니다.

▶ insert

```
long long hash_code = hashCode(str);
int hash = (int)(hash_code%mod);
loc temp; temp.word =str; temp.doc = i; temp.index = j;
//cout << hash <<" "<< str <<endl;
if(table[hash].size() == 0){
  vector<loc> first; first.push_back(temp);
  table[hash].push_back(first);
}
else{
  bool find = false;
  for(int h=0; h<table[hash].size(); h++){
    if((table[hash][h][0].word).compare(str) == 0){
      table[hash][h].push_back(temp);
      find = true;
      break;
    }
  }
  if(!find){
    vector<loc> next; next.push_back(temp);
    table[hash].push_back(next);
  }
}
```

insert의 경우에 우선 hash 값을 구해야합니다.

그 뒤에

```
if(table[hash].size() == 0){
  vector<loc> first; first.push_back(temp);
  table[hash].push_back(first);
}
```

아무런 데이터도 들어가 있지 않다면 그냥 넣으면 됩니다. 문제는 항상 상황이 ideal 하지는 않기 때문에 collision이 발생하게 됩니다.

```
else{
  bool find = false;
  for(int h=0; h<table[hash].size(); h++){
    if((table[hash][h][0].word).compare(str) == 0){
      //collison
      table[hash][h].push_back(temp);
      find = true;
      break;
    }
  }
  if(!find){
    //chaining
    vector<loc> next; next.push_back(temp);
    table[hash].push_back(next);
  }
}
```

collision이 발생한 경우에는 chain 들을 순차탐색하면서 같은 단어가 있는지 확인해봐야 합니다. 이 부분이 for문입니다. 만약 key 값이 중복된다면 어떻게 해야 될까요? 즉, for문 안에 if문에 key값이 중복되는 상황이다. 이때는 일반적인 Map 구조처럼 새롭게 value를 업데이트하면 안됩니다. insert하는 모든 key-value는 파일을 순차적으로 탐색하면서 모두 index를 만들게 되고 key 값이 같더라도

location 구조체인 value 값은 모두 다를 수밖에 없습니다. 이후 전체 chain을 모두 탐색했는데 같은 key 값을 찾을 수 없다면 (!find) 새롭게 위치집합을 만들고 이를 추가해주면 됩니다.

▶ 해시 함수

실제 해시 함수를 어떻게 설정하는지는 생각보다 runtime에 큰 영향을 미쳤다. 단순히 생각해볼 수 있는 것이 f(x) = x%10 인 제산함수인데 많은 collision이 발생할 것입니다.

우선 우리의 key는 문자열이므로 문자열을 어떤 방식을 적용하여 수치로 표현하여야 한다. 이를 처리하는 함수가 string_preprocess이다.

저희의 해시 함수는

```
r += r*a + s[i];
```

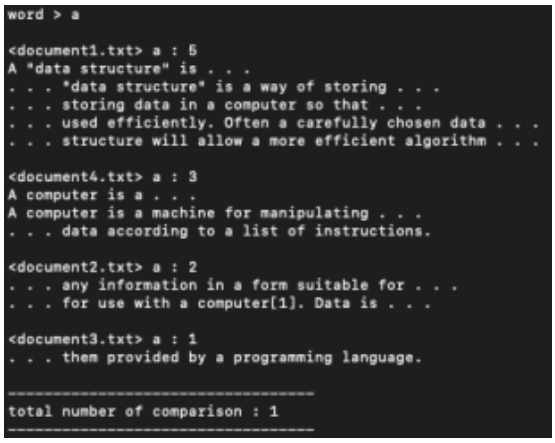
다음과 같이 정의되는데, 예를 들어

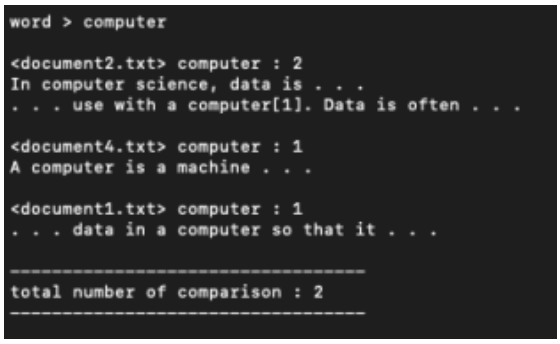
```
hash += key[i]
```

과 같은 해시 함수는 key의 글자마다 특징이 다를 때 hash 함수에 있어 큰 차이를 만들기 힘듭니다. 반면 우리가 정의한 함수는 r에 소수를 곱한 값을 더해줌으로서 조금이라도 다르면 우선 다른 값이 저장되고 그 뒤에 또 다른 값이 나오면 더 큰 차이를 벌려서 저장되므로 어느 정도 clustering을 막을 수 있고 이는 곧 좀 더 uniform distribution에 가까운 해시 함수를 만들 수 있습니다.

# 제 3 장 결과 및 토의

## 제 1 절 프로그램 테스트 결과



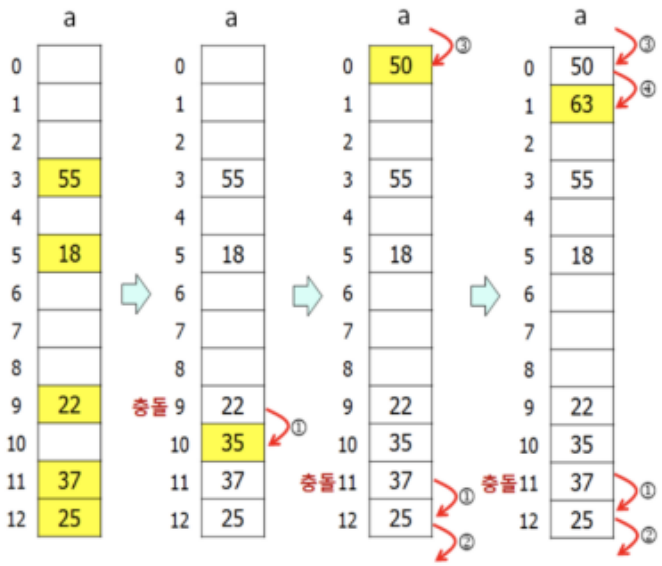




⇒ 위 프로그램 예제에서는 4개의 문서를 불러와 단어 검색을 수행하였습니다. 총 88개의 단어가 있었으며 검색하고자 하는 단어를 입력했을 때 문서 별로 존재하는 단어 개수가 출력, 내림차순으로 정렬되었습니다. 이때, 검색한 단어 전후로 단어 3개씩 출력되었습니다.

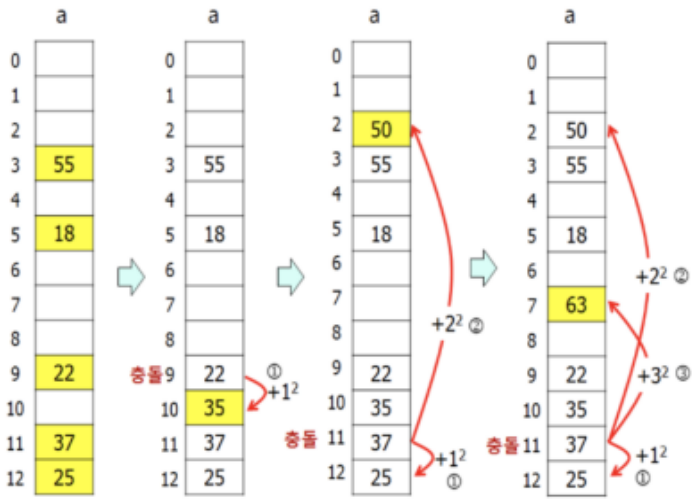
## 제 2 절 수행 결과에 대한 토의

단어를 모두 탐색할 필요 없이 검색한 단어의 hash값을 구해 hash table의 (hash값)번째의 데이터와 비교하고, 단어가 중복되는 경우 chaining을 이용해 충돌을 해결할 수 있습니다. Time complex는 O(1)를 만족하게 되고, 적은 비교 횟수를 가지게 됩니다. 또한 검색 속도도 linear probing 이나 quadratic probing보다 빠른데 linear probing 같은 경우는 hash table에 존재하는 단어를 검색할 경우 소요되는 시간이 1 + { 1 / (1 – a) } 인 방면에 chaining 같은 경우는 1 + a / 2 (여기서 a는 적재밀도) 이므로 chaining이 더 빠른 모

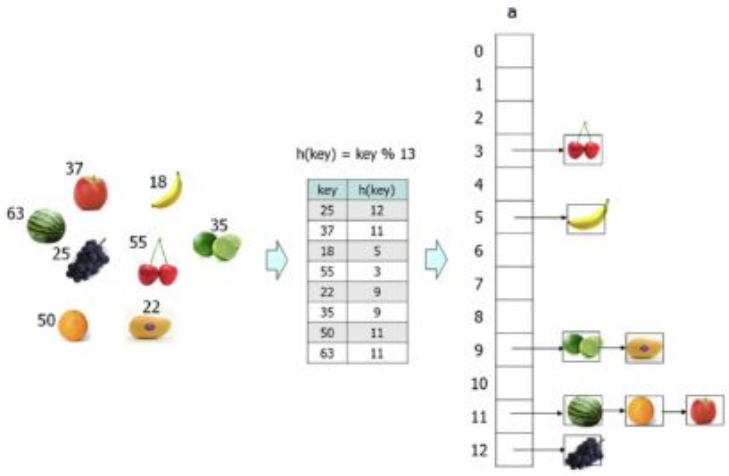
습을 보여줍니다. 그리고 linear probing이나 quadratic probing 은 hash table보다 많은 양의 데이터를 저장할 수 없는 데 비해 chaining은 hash table의 크기보다 많은 양의 데이터를 저장할 수 있으므로 chaining이 더 많은 데이터를 저장하고 검색할 수 있습니다.




<linear probing>




<quadratic probing>




<chaining>

그런데 우리가 풀어야하는 문제는 query word에 대한 문서를 순서대로 보여 주어야 합니다. 순서를 정하는 기준은 실제 검색엔진에서는 복잡한 알고리즘을 통한 가중치일 것이고 우리는 단순히 다큐먼트의 순서를 우선적으로 고려하고 다큐먼트의 순서가 같다면 다음으로 다큐먼트의 위치에서의 순위를 고려합니다.

hash 특성상 데이터의 비교와 특정 값을 search 하는 데에는 O(N) 시간이 걸리게 됩니다. 즉, query word에 대한 위치집합은 이론적으로 O(1) 상수 시간 안에 쿼리가 가능하지만 특정 값을 search 하는 문제는 비교적 느립니다.

그래서 위치집합에 대한 트리구조를 구성하여 search에 대한 효율도 높이고자 생각했지만 개발의 편의성을 고려해 단순 sorting을 이용하기로 했습니다. 우리가 이렇게 결정한 이유에는 적재밀도가 1에 가까울 때도

```
using, 1
so, 1 according, 1
any, 1
is, 4
it, 1
science, 1
programming, 1


programs, 1
them, 1
of, 2
instructions, 1
structures, 1


efficiently, 1

machine, 1
on, 1

to, 2 from, 1
structure, 2 often, 2 language, 1


used, 2
more, 1 efficient, 1 types, 1
way, 1
algorithm, 1


chosen, 1


for, 2
that, 1 can, 1
the, 1
```

[명세서에 주어진 예제에 대한 hash table의 일부]

최대 chain 개수는 3개였습니다. 레드블랙트리로 작성한다면(레드블랙트리는 complete tree이므로) 물론 단순 linked list 식 chaining 보다 빠르겠지만 hash table size와 word를 같게 하고 uniform distribution을 가정하면 uniformly하게 들어가지 않더라도 평균적으로 3개정도 들어갈 것이라는 이야기이고 사실상 비교횟수는 1~2 개 이상 나올 수 없습니다. 거기에 메모리 측면에서 보면 chaining이라면 레드블랙트리와 linked list 식 chaining 개수는 똑같습니다. 그렇지만 명확한 사실은 만약 collision이 매우 많이 발생한다면 레드블랙트리로 chaining을 구현하는 것이 훨씬 효율적입니다.

그렇지만 실험결과를 토대로 collision이 특정 index에서 모아서 발생할 확률은 gaussian distribution에 따를 것이고 사실상 대부분의 경우에 보편적으로 해시 함수를 잘 만든다면 최대한 uniformly 하게 분포될 것이라고 가정하였습니다.

### 제 3 절 기타

## 제 4 장 부록

► 참고문헌


- Data structure a pseudocode approach with c(Richar F. Gillberg)
- http://www.cs.fsu.edu/~asriniva/courses/DS12/lectures/Lec24.ppt (FSU hashing ppt)

+ dictionary_make(hash insert)

```c
void dictionary_make(vector <string> text[]){
  long long mod = MOD;
  int word_count = 0;
  for(int i=0; i<MAX_DOC_NUM; i++){
    comparison++;
    for(int j=0; j<text[i].size(); j++){
      comparison++;
      string str = string_preprocess(text[i][j]);
      word_count++;
      long long hash_code = hashCode(str);
      int hash = (int)(hash_code%mod);
      loc temp; temp.word =str; temp.doc = i; temp.index = j;
      //cout << hash <<" "<< str <<endl;
      if(table[hash].size() == 0){
        vector<loc> first; first.push_back(temp);
        table[hash].push_back(first);
      }
      else{
        bool find = false;
        for(int h=0; h<table[hash].size(); h++){
          if((table[hash][h][0].word).compare(str) == 0){
            //collison
            table[hash][h].push_back(temp);
            find = true;
            break;
          }
        }
        if(!find){
          //chaining
          vector<loc> next; next.push_back(temp);
          table[hash].push_back(next);
        }
      }
    }
  }
  dict_print();
  printf("dictionary maked successful with %d words\n",word_count);
}
```

+hash

```
long long hashCode(string s){
    long long r = 0;
    long long a = 19; //prime number
    for(int i = 0; i < s.length(); i++)r +=  r*a + s[i];
    return r;
}
```