Assignment 7 DESIGN.pdf

Ning Jiang

1. **Description of Program:** For this assignment, we will be creating a program that attempts to identify the most likely authors for an anonymous sample of text given a large database of texts with known authors.

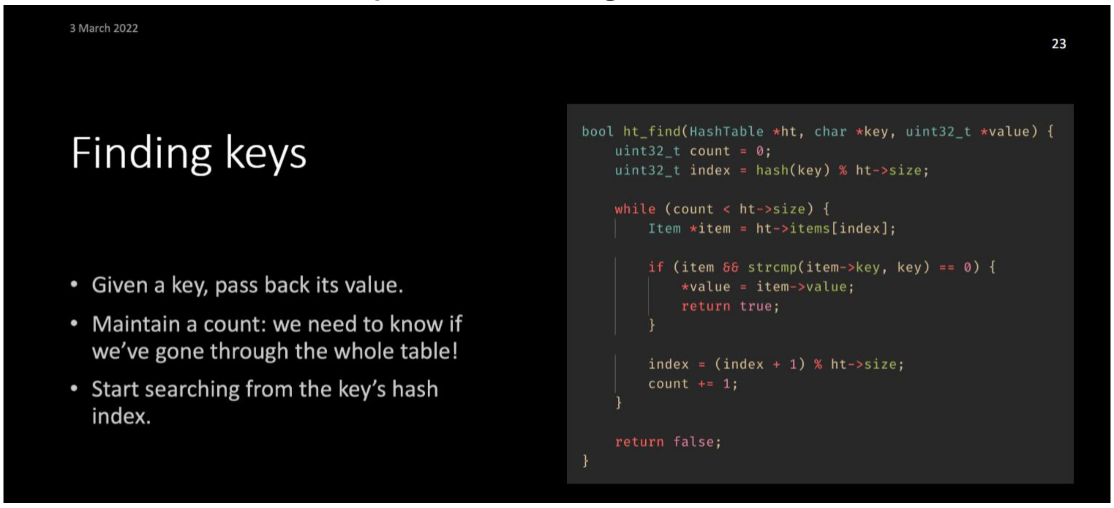
Files to be included in directory “asgn7”:

1. bf.h: Defines the interface for the Bloom filter ADT. Do not modify this.
2. bf.c: Contains the implementation of the Bloom filter ADT.
3. bv.h: Defines the interface for the bit vector ADT. Do not modify this.
4. bv.c: Contains the implementation of the bit vector ADT.
5. ht.h: Defines the interface for the hash table ADT and the hash table iterator ADT. Do not modify this.
6. ht.c: Contains the implementation of the hash table ADT and the hash table iterator ADT.
7. identify.c: Contains main() and the implementation of the author identification program.
8. metric.h: Defines the enumeration for the distance metrics and their respective names stored in an array of strings. Do not modify this.
9. node.h: Defines the interface for the node ADT. Do not modify this.
10. node.c: Contains the implementation of the node ADT.
11. parser.h: Defines the interface for the regex parsing module. Do not modify this.
12. parser.c: Contains the implementation of the regex parsing module.
13. pq.h: Defines the interface for the priority queue ADT. Do not modify this.
14. pq.c: Contains the implementation for the priority queue ADT.
15. salts.h: Defines the primary, secondary, and tertiary salts to be used in your Bloom filter implementation. Also defines the salt used by the hash table in your

hash table implementation.
16. speck.h: Defines the interface for the hash function using the SPECK cipher. Do not modify this.
17. speck.c: Contains the implementation of the hash function using the SPECK cipher. Do not mod-
ify this.
18. text.h: Defines the interface for the text ADT. Do not modify this.
19. text.c: Contains the implementation for the text ADT

Structure & explanation / Pseudocode:
Explanations for some of the equations are cited from asgn7.pdf.
ht.c
It is used to create a hash table to quickly store and retrieve unique words in a sample of text. We well use linear probing to resolving linear collection.
The hash table will contain slats, the size of the text, the node which supposed to be put into the hash table.

Iterating Over Hash Tables
Cite hash table from professor Long's class lecture slides.



3 March 2022

23

Finding keys

- Given a key, pass back its value.
- Maintain a count: we need to know if we've gone through the whole table!
- Start searching from the key's hash index.

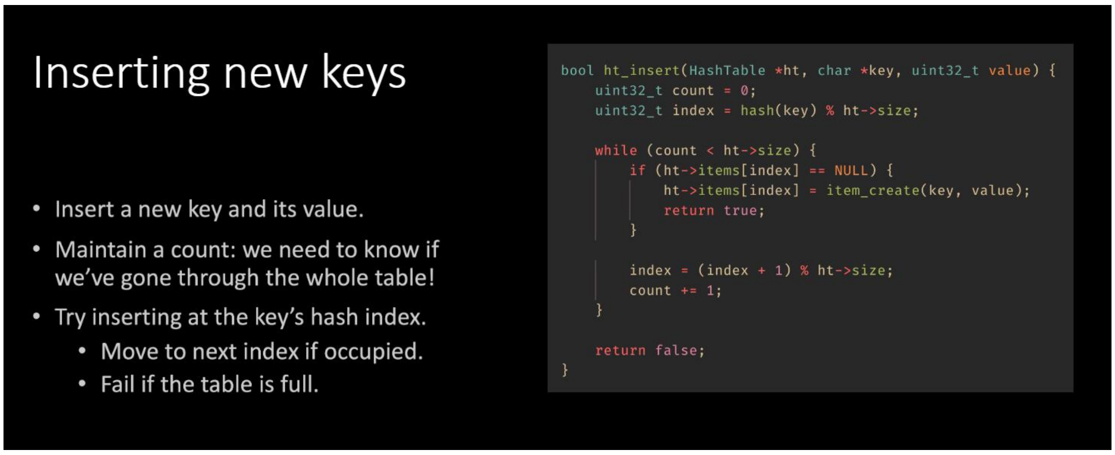
```c
bool ht_find(HashTable *ht, char *key, uint32_t *value) {
    uint32_t count = 0;
    uint32_t index = hash(key) % ht->size;

    while (count < ht->size) {
        Item *item = ht->items[index];

        if (item && strcmp(item->key, key) == 0) {
            *value = item->value;
            return true;
        }

        index = (index + 1) % ht->size;
        count += 1;
    }

    return false;
}
```



Inserting new keys

- Insert a new key and its value.
- Maintain a count: we need to know if we've gone through the whole table!
- Try inserting at the key's hash index.
  - Move to next index if occupied.
  - Fail if the table is full.

```c
bool ht_insert(HashTable *ht, char *key, uint32_t value) {
    uint32_t count = 0;
    uint32_t index = hash(key) % ht->size;

    while (count < ht->size) {
        if (ht->items[index] == NULL) {
            ht->items[index] = item_create(key, value);
            return true;
        }

        index = (index + 1) % ht->size;
        count += 1;
    }

    return false;
}
```

This is ADT for iterate over all the entries in a hash table.

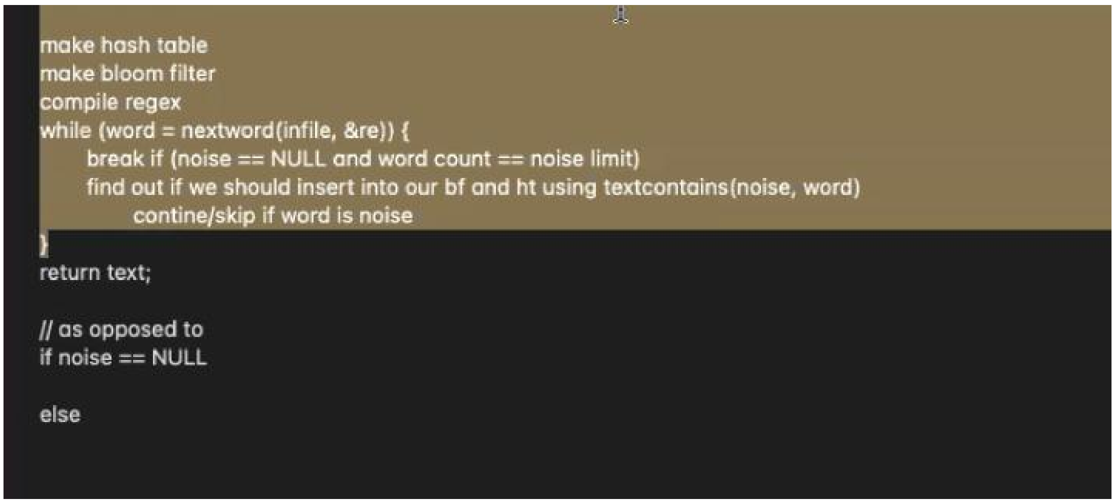
*Node *ht_iter(HashTableIterator *hti)*
Returns the pointer to the next valid entry in the hash table.

Node.c
This is the ADT which will create the node for hash table.
Wach node will contain a word and a it's count.
Remember to make a copy of word! Using strdup().

Bf.c
A Bloom filter can be represented as an array of m bits, or a bit vector. A Bloom filter should utilize k different hash functions. Using these hash functions, a set element added to the Bloom filter is mapped to at most k of the m bit indices, generating a uniform pseudo-random distribution.

Bv.c
A bit vector is an ADT that represents a one dimensional array of bits, the bits in which are used to denote if something is true or false (1 or 0).This is an efficient ADT since, in order to represent the truth or falsity of a bit vector of n items, we can use n/8 uint8_ts instead of n, and being able to access 8 indices with asingle integer access is extremely cost efficient. Since we cannot directly access a bit, we must use bitwise operations to get, set, and clear a bit within a byte.

Text.c
Create text cite from Audrey



```
make hash table
make bloom filter
compile regex
while (word = nextword(infile, &re)) {
    break if (noise == NULL and word count == noise limit)
    find out if we should insert into our bf and ht using textcontains(noise, word)
        contine/skip if word is noise
}
return text;

// as opposed to
if noise == NULL

else
```

We will need a new ADT to encapsulate the parsing of a text and serve as the in-memory representation of the distribution of words in the file. This is the Text ADT. It will contain a hash table, Bloom filter, and the count of how many words are in the text. This is not the number of unique words in the text, but the total number of words in the text.

*double text_dist(Text *text1, Text *text2, Metric metric)*
In this function, we will use three different method to calculate the distance:

2.1 Manhattan Distance

The *Manhattan distance* is the simplest of the three to compute. To compute it, you simply take the magnitude of the difference between each component of the vector. You can think of it as how far you would have to go if you had to follow the gridded streets of Manhattan. This is computed as:

$$\mathrm{MD} = \sum_{i \in n} |u_i - v_i|$$

With the example texts we get the absolute value of the component distances before summing them:

- "hello" : $|u_1 - v_1| = |\frac{1}{2} - \frac{0}{3}| = \frac{1}{2}$
- "goodbye" : $|u_2 - v_2| = |\frac{0}{2} - \frac{2}{3}| = \frac{2}{3}$
- "world" : $|u_3 - v_3| = |\frac{1}{2} - \frac{1}{3}| = \frac{1}{6}$

This gives us a Manhattan distance of $\frac{4}{3}$.

2.2 Euclidean Distance

The *Euclidean distance* is calculated very similarly to the Manhattan distance, except you are able to go as the bird flies. Thus, we just need to remember how Euclid taught us to find the length of the hypotenuse of a triangle, so we add the squares of all the magnitudes together before getting the square root of the sum. This is computed as:

$$\mathrm{ED} = \sqrt{\sum_{i \in n} (u_i - v_i)^2}$$

With the example texts we first square the component distances before summing them:

- "hello" : $(u_1 - v_1|)^2 = (\frac{1}{2} - \frac{0}{3})^2 = \frac{1}{4}$
- "goodbye" : $(u_2 - v_2|)^2 = (\frac{0}{2} - \frac{2}{3})^2 = \frac{4}{9}$
- "world" : $(u_3 - v_3|)^2 = (\frac{1}{2} - \frac{1}{3})^2 = \frac{1}{36}$

We then add all the values together to get a sum of $\frac{26}{36}$, giving us a Euclidean distance of $\sqrt{\frac{26}{36}} \approx 0.85$.

2.3 Cosine Distance

The *cosine distance* is different from the first two in that we derive it from the *cosine similarity* of two vectors, or the cosine of the angle between two vectors.

$$\text{Cos}(\Theta) = \frac{\vec{u} \cdot \vec{v}}{|\vec{u}| \times |\vec{v}|}$$

If we are to normalize the vectors, the bottom of the fraction is just 1, and so we only need to get the dot product of the two vectors:

$$\vec{u} \cdot \vec{v} = \sum_{i \in n} u_i \times v_i$$

Note that the angle approaches 0 as the cosine of the angle approaches 1. To stay consistent with the other metrics, we instead get the cosine *distance* by subtracting the similarity from 1. This ensures that the texts that are more similar will have a smaller computed distance. For our example texts we multiply the vector components together before summing all the results:

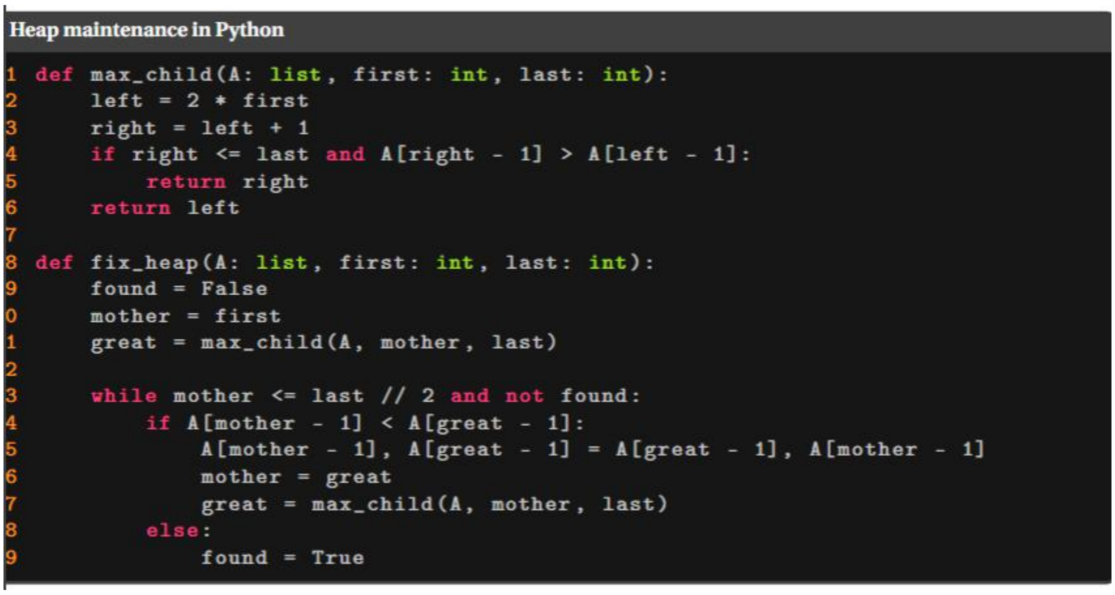
- "hello" : $u_1 \times v_1 = \frac{1}{2} \times \frac{0}{3} = 0$
- "goodbye" : $u_2 \times v_2 = \frac{0}{2} \times \frac{2}{3} = 0$
- "world" : $u_3 \times v_3 = \frac{1}{2} \times \frac{1}{3} = \frac{1}{6}$

The computed cosine similarity is $\frac{1}{6}$. We subtract this from 1 for the cosine distance, giving us a cosine distance of $1 - \frac{1}{6} = \frac{5}{6}$.

Pq.c

Cite pq from asgn3.

Use heap sort as same as asgn3, instead, we use min_child
First, we make a ADT to contain author and dist, the same as Node.c.

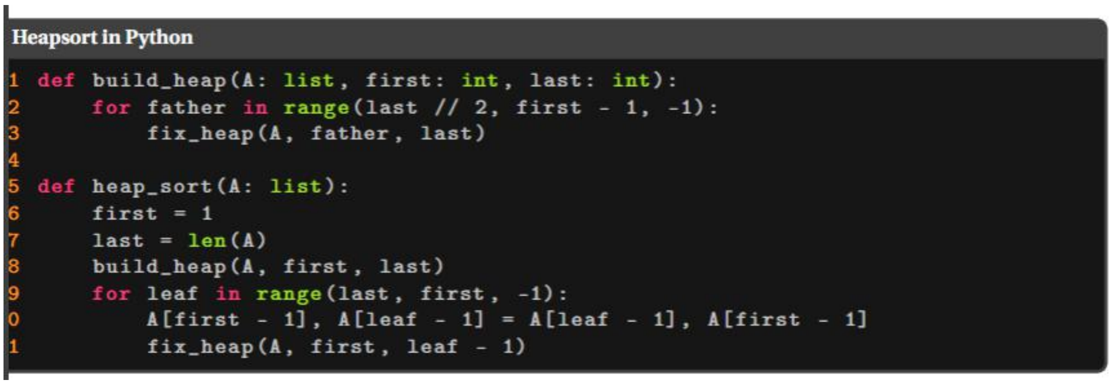


Heap maintenance in Python

```python
def max_child(A: list, first: int, last: int):
    left = 2 * first
    right = left + 1
    if right <= last and A[right - 1] > A[left - 1]:
        return right
    return left

def fix_heap(A: list, first: int, last: int):
    found = False
    mother = first
    great = max_child(A, mother, last)

    while mother <= last // 2 and not found:
        if A[mother - 1] < A[great - 1]:
            A[mother - 1], A[great - 1] = A[great - 1], A[mother - 1]
            mother = great
            great = max_child(A, mother, last)
        else:
            found = True
```


Heapsort in Python

```python
def build_heap(A: list, first: int, last: int):
    for father in range(last // 2, first - 1, -1):
        fix_heap(A, father, last)

def heap_sort(A: list):
    first = 1
    last = len(A)
    build_heap(A, first, last)
    for leaf in range(last, first, -1):
        A[first - 1], A[leaf - 1] = A[leaf - 1], A[first - 1]
        fix_heap(A, first, leaf - 1)
```

Identify.c
Comment line options:
• -d : Specify path to database of authors and texts. The default is lib.db.
• -n : Specify path to file of noise words to filter out. The default is noise.txt.
• -k : Specify the number of matches. The default is 5.
• -l : Specify the number of noise words to filter out. The default is 100.
• -e : Set the distance metric to use as the Euclidean distance. This should be the default metric if
no other distance metric is specified.
• -m : Set the distance metric to use as the Manhattan distance.
• -c : Set the distance metric to use as the cosine distance.
• -h : Display program help and usage.

progress
• Create a new Text that contains words in the noise word file. This will be the noise word Text that
contains words to filter out. Make sure that each word in the noise word Text is lowercased.
• Create a new anonymous Text from text passed in to stdin. This is the text that you want to
identify. Remember to filter out all the noise words in the noise word Text that you just created.
• Open up the database of authors and text files. This file, by default, should be lib.db. The first
line of this file indicates the number of author/text pairs. We'll refer to this number as n.
• Create a priority queue that can hold up to n elements. This will be used to efficiently find the top
k likely authors who authored the anonymous text.
• Process the rest of the lines in the database file in pairs. You will need to do the following:

– Use fgets() to read in the name of an author, then use fgets() again to read in the path to the file of text that author authored. You will need to remove the newline character that
fgets() preserves.
– Open up the author’s file of text and use it to create a new Text instance. If the file could not
be opened, continue to the next line of the database file. Remember to turn each parsed word
to lowercase before checking if the word should be added to the Text.
– Compute the distance between the author’s Text and the anonymous Text, filtering out words in the noise word Text.
– Enqueue the author’s name and the computed distance into the priority queue.
• Get the top k likely matches by dequeueing the priority queue. Note that there may be less k entries in the priority queue, depending on whether or not a file could be processed.