Oving4

Del 1

Programmet leser fila med navn, og klarer å legge alle i hashtabellen.

```
{
  FILE *fp;
  fp = fopen("navn.txt", "r"); // Open a file in readmode hence "r"
  if (fp == NULL)
  {
    printf("Not able to open file\n");
    return 1;
  }
```

Her leses fila med `fopen` og lagrer en pointervariabel til fila

```
// insert a key (whole name) -value (row position in the .txt file)
void insertHashTable(HashTable *hash_table, char *key, int value, int *total_collisions)
{
  size_t index = hashFunction(hash_table, key);
  Node *head = hash_table->array[index];

  size_t bucket_length = 0;
  for (Node *p = head; p; p = p->next)
    bucket_length++;
  if (bucket_length)
  {
    printf("Collision on %zu insert %s with:", index, key);
    for (Node *p = head; p; p = p->next)
      printf(" %s", p->key);
    printf("\n");
  }
  *total_collisions += bucket_length;

  Node *new_node = (Node *)malloc(sizeof(Node));
  new_node->key = strdup(key);
  new_node->value = value;
  new_node->next = head;
  hash_table->array[index] = new_node;
  hash_table->entries++;
}
```

Her er metoden som legger et navnet i hashtabellen samt teller antall kollisjoner.

```c
while ((fgets(name_file, sizeof(name_file), fp)))
{
  // Trim for white space
  size_t n = strlen(name_file);
  if (n && name_file[n - 1] == '\n')
    name_file[n - 1] = '\0';
  line_count++;
  insertHashTable(&hash_table, name_file, line_count, &insert_collision);
```

hvor i `main()` den itererer over alle navnene i `navn.txt` og legger den til vha metoden ovenfor.

Kollisjoner håndteres med lenka lister.

```c
// Linked list
typedef struct Node
{
  char *key;
  int value; // row i navn.txt
  struct Node *next;
} Node;
```

```c
  size_t bucket_length = 0;
  for (Node *p = head; p; p = p->next)
    bucket_length++;
  if (bucket_length)
  {
    printf("Collision on %zu insert %s with:", index, key);
    for (Node *p = head; p; p = p->next)
      printf(" %s", p->key);
    printf("\n");
  }
  *total_collisions += bucket_length;

  Node *new_node = (Node *)malloc(sizeof(Node));
  new_node->key = strdup(key);
  new_node->value = value;
  new_node->next = head;
  hash_table->array[index] = new_node;
  hash_table->entries++;
}
```

I insertmetoden håndteres kollisjoner med egendefinert singly linked list. Lenka listen er definert vha `struct` og navngitt `Node` da lenka lister er en kolleksjon med noder der `*next` er pointer som inneholder addressen til neste node.

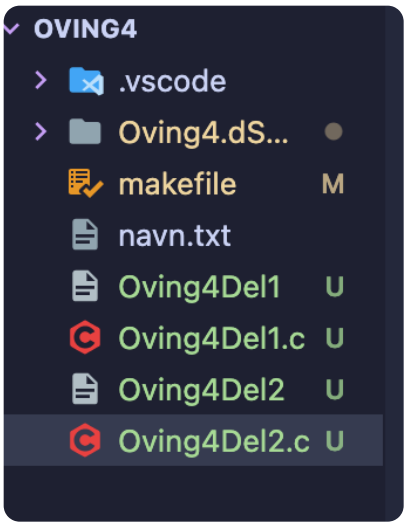
Programmet skriver ut alle kollisjoner, og lastfaktoren til slutt

```
Total collisions at insert: 44
Load factor: 0.751592
```

Programmet klarer å slå opp personer i faget ved hjelp av hashtabellen. (Bruk gjerne oppslag på eget navn.)

```
Search collision on 81, collide with Nicoline Lean,Tønnessen, searching for Tri Tac,Le
Search collision on 81, collide with Erlend,Sundsdal, searching for Tri Tac,Le
Searching for: Tri Tac,Le at row in name.txt: 59, search collision: 2
```

Programmet bruker «navn.txt», ikke «mappe/navn.txt». Retting tar for lang tid, om jeg må håndtere hundrevis av mapper. Så det skjer ikke.



`make Oving4Del1` eller `make run1` for å builde/kjør del 1.

`make Oving4Del2` eller `make run2` for å builde/kjør del 2.

`make run-all` for å kjøre begge

`make clean` for å bjerne binary executables.

Legg ved utskrift fra kjøringen.

```
(base) Tri-macbook-pro:Oving4 trile$ make run
gcc -Wall -g Oving4.c -o Oving4
./Oving4
Collision on 78 insert Siri Volan,Hatling with: Trang Minh,Duong
Collision on 126 insert Martin Olai Amundsen,Henøen with: Stian,Dølerud
Collision on 42 insert Helena An Haaland,Haarr with: Kasper Østerlie,Gladsøy
Collision on 89 insert Alexander Tawan Atjanthuek,Johnsen with: Regine,Hjelmtveit
Collision on 76 insert Markus Hysing,Jøssund with: Helge,Hafting
Collision on 34 insert Nora Lilliedahl,Kirknes with: Aleksandrs,Broks
Collision on 75 insert Casper,Kolsrud with: Amadeus Syvertsen,Berg
Collision on 127 insert Benjamin,Kvello-Hansen with: Mikael Stray,Frøyshov
Collision on 78 insert Hel Eline,Larsen with: Siri Volan,Hatling Trang Minh,Duong
Collision on 90 insert Eilif Hjermann,Lindblad with: Sverre Grønhaug,Halvorsen
Collision on 129 insert Oskar Hoddø,Lindh with: Jørgen Fenstad,Kottum
Collision on 153 insert Dennis,Moe with: Thomas Oliver Wallin,Higgins
Collision on 153 insert Anders Flatland,Myrvang with: Dennis,Moe Thomas Oliver Wallin,Higgins
Collision on 12 insert Vetle,Nilsen with: Galina,Kristensen
Collision on 126 insert Durva,Parmar with: Martin Olai Amundsen,Henøen Stian,Dølerud
Collision on 140 insert Jostein,Rodahl with: Oscar Kehinde Asplin,Martins
Collision on 44 insert Audun Nåvik,Rosvold with: Edvard Emmanuel,Klavenes
Collision on 106 insert Filip Engan,Ræder with: Serghii,Dykyy
Collision on 106 insert Mahshid,Saleh with: Filip Engan,Ræder Serghii,Dykyy
Collision on 69 insert Kristian Ask,Selmer with: Helene,Askeland
Collision on 106 insert Shakti Om,Sharma with: Mahshid,Saleh Filip Engan,Ræder Serghii,Dykyy
Collision on 6 insert Gustav Haverstad,Skyberg with: Stian Mathisen,Myrbekk
Collision on 26 insert Eskild,Smestu with: Axel Aartun,Hansson
Collision on 84 insert Eva,Stamatovska with: Maamoun Adnan,Hmaidoush
Collision on 13 insert Theodor,Strøm-Thrane with: Anine Selle,Standnes
Collision on 81 insert Erlend,Sundsdal with: Tri Tac,Le
Collision on 103 insert Julie Bao Thi,Tran with: Tord Johannesen,Fosse
Collision on 81 insert Nicoline Lean,Tønnessen with: Erlend,Sundsdal Tri Tac,Le
Collision on 18 insert Bjørn Adam,Vangen with: Christian Bellika,Jensen
Collision on 34 insert Harald Theodor Helland,Velde with: Nora Lilliedahl,Kirknes Aleksandrs,Broks
Collision on 47 insert Piranavan,Visvalingam with: Håkon Nikolai Indrestrand,Gray
Collision on 102 insert Cecilie,Vu with: Maria Kristin,Olafsdottir
Collision on 99 insert Petter Tjelde,Vaagen with: Thea Marie Alver,Lønvik
Collision on 144 insert Ole-Kristian,Wigum with: Ingunn Tonetta,Erdal
Collision on 90 insert Christian Midtskaug,Aas with: Eilif Hjermann,Lindblad Sverre Grønhaug,Halvorsen
Total names: 118
Table size: 157
Total collisions at insert: 44
Load factor: 0.751592
Average collision per person: 0.372881
Search collision on 81, collide with Nicoline Lean,Tønnessen, searching for Tri Tac,Le
Search collision on 81, collide with Erlend,Sundsdal, searching for Tri Tac,Le
Searching for: Tri Tac,Le at row in name.txt: 59, search collision: 2
```

Antall kollisjoner pr. person er under 0,4 (men ikke eksakt 0, regn med desimaltall…

`Average collision per person: 0.372881`

Hasjfunksjonen jeg brukte kalles polynomial rolling hash med base 31 med formål om å unngå å få samme verdi hvis navnene inneholder like bokstaver, men ulik rekkefølge.
Kilde: https://www.geeksforgeeks.org/dsa/string-hashing-using-polynomial-rolling-hash-function/

Del 2

1. Valgte å gjøre `m` lik et primtall på litt over 10 millioner vha. funksjonen `findPrime(m)`
2. Lager en unik tabell der hver tall er lik indeksverdien, men tabellene stokkes med den troverdige fisher-yates shuffle, kilde: https://www.geeksforgeeks.org/dsa/shuffle-a-given-array-using-fisher-yates-shuffle-algorithm/
   Dette så til å fungerte best da å kjøre "forrige tall + rand(1,1000)" tok altfor lang tid og programmet frøs. Kan hende at det er hardware faktorer som spiller inn da min pc er ganske gammel.
3.
```c
size_t probeLinear(size_t hash_val, size_t i, size_t table_size)
{
    return ((long long)hash_val + (long long)i) % table_size;
}

size_t probeDouble(size_t h1, size_t h2, size_t i, size_t table_size)
{
    long long t = (long long)h1 + ((long long)i * (long long)h2);
    return (t % table_size);
}
```
   Metodene bruker `size_t` for å unngå overflow da det kan fort hende seg at noen verdier blir `>INT_MAX`.
```c
int insert(int key, size_t table_size, int hash_table[table_size], char chooseMethod, long long *i
{
    size_t h1 = hashFunction1(key, table_size);
    size_t h2 = ((chooseMethod == 'l') ? 0 : hashFunction2(key, table_size));
    for (size_t i = 0; i < table_size; i++)
    {
        size_t pos = ((chooseMethod == 'l') ? probeLinear(h1, i, table_size) : probeDouble(h1, h2, i,
        if (!hash_table[pos])
        {
            hash_table[pos] = key + 1;
            return (int)pos;
        }
        if (hash_table[pos] != key + 1)
            (*insert_collision)++;
    }
    return -1;
}
```
   Her er metoden som setter inn verdiene i `hash_table`. Metoden er delvis kopiert fra boka da jeg bestemte meg for å ikke bruke `struct` til del 2

```
size_t hashFunction1(size_t key, size_t table_size)
{
  // 2^64, tablesize er primtall    "primtall": Unkno
  const size_t A = 11400714819323198485ull;
  return (key * A) % table_size;
  // return key % table_size;
}

size_t hashFunction2(size_t key, size_t table_size)
{
  return 1 + (key % (table_size - 1));
}
```

Ganger nøkkelen med et tall $A = 2^{64}$

```
Insert collision linear: 8006947
Insert collision double: 5030999
```

Som man ser her er dobbelt hasing mer effektiv enn lineær hashing. Her er last faktor satt til 0.7.

```
Load factor: 0.500000
Insert collision linear: 791876
Timing on linear probing: 442977395.00 (ns)
Insert collision double: 823380
Timing on double hashing: 300215172.00 (ns)
Load factor: 0.800000
Insert collision linear: 5515335
Timing on linear probing: 651514123.00 (ns)
Insert collision double: 4433429
Timing on double hashing: 763833122.00 (ns)
Load factor: 0.900000
Insert collision linear: 12196938
Timing on linear probing: 810875575.00 (ns)
Insert collision double: 8931734
Timing on double hashing: 1008997811.00 (ns)
Load factor: 0.990000
Insert collision linear: 45372466
Timing on linear probing: 1420730304.00 (ns)
Insert collision double: 29820973
Timing on double hashing: 2071122855.00 (ns)
Load factor: 1.000000
Insert collision linear: 255570433
Timing on linear probing: 3956886160.00 (ns)
Insert collision double: 127184054
Timing on double hashing: 4742596194.00 (ns)
```

4. Målinger på tid og antall kollisjon med innsetting, der lastfaktoren er påvirkningsfaktoren
5. **Lag en kort rapport, med tabeller (og evt. kurver/illustrasjoner) som viser tidsbruk og hvor mange kollisjoner det blir for ulike fyllingsgrader og type hashtabell.**

   Dataene innsatt i grafene nedenfor er samme basert på samme data som utskriften i oppgave 4

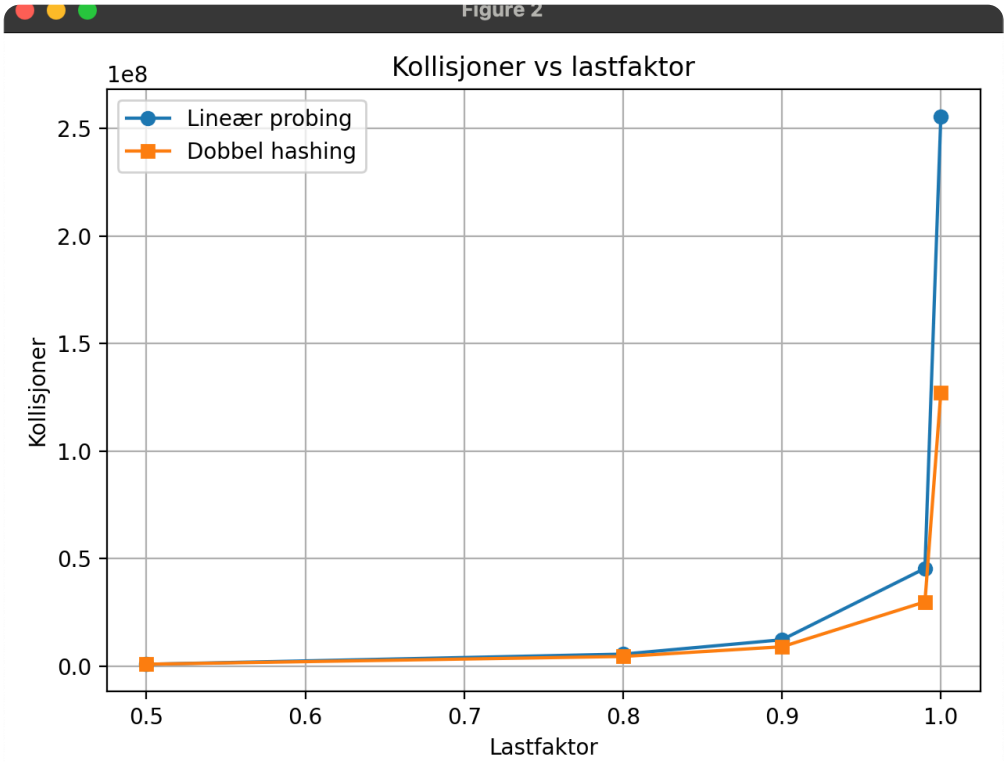


figure 2

Denne grafen viser hvordan høyere lastfaktor fører til flere kollisjoner. Begge metodene får omtrent lik kollisjoner ved lastfaktor under intervallet $[0.7, 0.8]$Derimot håndterer (skarerer) dobbelt hashing kollisjoner bedre etter denne terskelen, altså ved nær full tabell. Vi ser at begge når en metodene får eksponentielt økt antall kollisjoner på lastfaktorverdi rett under 1 .

6. **Se om det går an å trekke noen konklusjoner ang. antall kollisjoner og tidsbruk:**

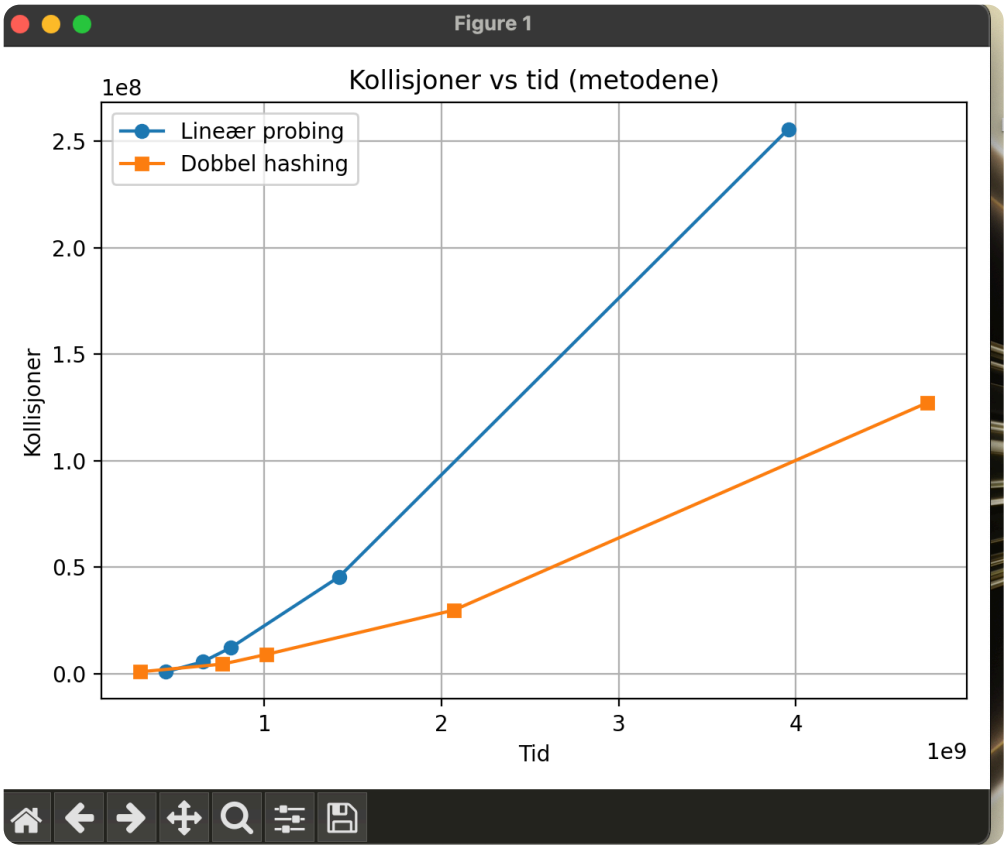


figure 1

Vi ser her at flere kollisjoner tar lengre tid, men lineær probing er faktisk raskere enn dobbelt hashing når det gjelder antall kollisjoner. Men dobbelt hashing unngår å få mange kollisjoner, men probingen er mer kostbart.

Ved sammenligning av figure 1 og figure 2 kan man konkludere med at lineær probing er mer effektiv med lastfaktor < 0.9, men dobbelt hashing er mer effektiv ved høyere lastfaktor, altså om man vil fylle tabellen.

Etter observasjon i henhold til graf nr 1 viser det seg at antall kollisjoner begynner å øke i intervallet $[0.7 - 0.8]$. I følge kilden under er default lastfaktor til HashMap (Java Collections) 0.75f noe som stemmer godt overens med observasjonene.
https://www.geeksforgeeks.org/dsa/load-factor-in-hashmap-in-java-with-examples/

Python scripts for grafene:

```python
import matplotlib.pyplot as plt

load_factor = [0.5, 0.8, 0.9, 0.99, 1.0]

xL_time = [442977395.00, 651514123.00, 810875575.00, 1420730304.00, 3956886160.
xD_time = [300215172.00, 763833122.00, 1008997811.00, 2071122855.00, 4742596194

yL_col  = [791876.00, 5515335.00, 12196938.00, 45372466.00, 255570433.00]
yD_col  = [823380.00, 4433429.00, 8931734.00, 29820973.00, 127184054.00]


plt.figure()
plt.plot(xL_time, yL_col, marker="o", label="Lineær probing")
plt.plot(xD_time, yD_col, marker="s", label="Dobbel hashing")
plt.xlabel("Tid")
plt.ylabel("Kollisjoner")
plt.title("Kollisjoner vs tid (metodene)")
plt.grid(True)
plt.legend()
plt.tight_layout()

# Kollisjoner vs lastfaktor
plt.figure()
plt.plot(load_factor, yL_col, marker="o", label="Lineær probing")
plt.plot(load_factor, yD_col, marker="s", label="Dobbel hashing")
plt.xlabel("Lastfaktor")
plt.ylabel("Kollisjoner")
plt.title("Kollisjoner vs lastfaktor")
plt.grid(True)
```

```
plt.legend()
plt.tight_layout()

plt.show()
```