IDATT2101 Øving 4

Hashtabeller

Trang Minh Duong

Deloppgave 1: Hashtabell med tekstnøkler

I løsningen til denne deloppgaven er tabellstørrelse satt til $m = 128$ for ca. $n \approx 118$ navn, som gir forventet lastfaktor $\alpha = n/m \approx 0.922$. Kjøring ga: lastfaktor `0.922`, totalt innsettingskollisjoner `35`, og kollisjoner per person `(K/n)=0.297` < 0.4.

I denne oppgaven bruker jeg Horners metode, der strengen behandles som et polynom $h = h \cdot B + cp$ for hvert Unicode-tegn (cp), med $B = 31$. Hvorfor akkurat 31 er fordi den er både et oddetallsprimtall og er relativt primisk med 2^k. Vi kan tenke på dette at tabellstørrelsen er 2^7, som betyr at vi beholder 7 nederste bit av h (h % `tableSize`). Siden basen er oddetall så skifter det ikke bare til venstre (som i tilfellet med partallbase), det blander også de nederste bitene. Dermed kan alle tegn i strengen påvirke de 7 nederste bitene man beholder, og man får mye bedre spredning. Dette tallet er også brukt i Java's `String.hashCode`-funksjon. Komma og mellomrom ignoreres både ved hashing og ved sammenligning. Verdien holdes i et 64-bits heltall slik at naturlig overflow fungerer som mod 2^{64}, før vi tar h mod m for å finne bøtteindeksen. Vi leser UTF-8 og fjerner `\r` ved CRLF.

```
1 static size_t hash_poly(const string& name, size_t tableSize
      , uint64_t B = 31) {
2     uint64_t h = 0;
3     for (uint32_t cp : utf8_codepoints(name)) {
4         h = h * B + cp; // Horner
5     }
6     return static_cast<size_t>(h % tableSize);
7 }
```

Listing 1: Horners metode

Kollisjoner håndteres med separat kjeding (enkle lenkede lister), hver bøtte peker på hodet av en liste. Ved innsetting lenkes den nye noden inn i hodet, dersom bøtta ikke er tom logges paret "[nytt navn] vs [første i bøtta]". Under søk traverseres listen og hvert steg forbi en node logges som en kollisjon under søk. Forventet tidsbruk er $O(1+\alpha)$, der $\alpha = n/m$ er lastfaktoren.

```
1 // Lenket liste-node struct
2 struct Node {
3     string key;
4     Node* next;
5     explicit Node(string k, Node* n=nullptr) : key(move(k)),
          next(n) {}
6 };
```

Listing 2: Lenket lister

```
1  // Contain-metoden
2  bool contains(const string& name) {
3          size_t idx = indexOf(name);
4          Node* p = buckets[idx];
5          if (!p) return false;
6
7          const string qcan = canonical(name);
8          if (canonical(p->key) == qcan) return true;
9
10         for (Node* q = p->next; q; q = q->next) {
11             cout << "Kollisjon (s k): [" << name << "] vs [
                   " << p->key << "]\n";
12             search_collisions++; // <- viktig
13             if (canonical(q->key) == qcan) return true;
14             p = q;
15         }
16         return false;
17     }
18
19 // print kollisjoner metoden
20 void printCollisions() const {
21         size_t totalCollision = 0;
22         for (size_t i = 0; i < tableSize; i++) {
23             size_t s = chainSize(buckets[i]);
24             if (s > 1) {
25                 cout << "Indeks " << i << " (listest rrelse
                       " << s << "): ";
26                 for (Node* p = buckets[i]; p; p = p->next)
                       cout << "[" << p->key << "] ";
27                 cout << "\n";
28                 totalCollision += (s - 1);
29             }
30         }
31         cout << "\nTotalt elementer: " << count << "\n";
32         cout << "Totalt kollisjoner (beregnet fra kjeder): "
                << totalCollision << "\n";
33         cout << fixed << setprecision(3)
34              << "Kollisjoner per person (beregnet): " << (
                    count ? (double)totalCollision / count :
                    0.0) << "\n";
35     }
```

Listing 3: Bruk av lenket lister i koden

Kompleksitet

La L være lengden på en streng, n antall navn og m tabellstørrelsen.

- **Hashing/indeksering:** $O(L)$ for å lese UTF-8 $\rightarrow$ code points og beregne $h = h \cdot B + cp$, deretter h mod m.

- **Innsetting (separat kjeding):** Forventet $O(1+\alpha)$, hvor lastfaktor $\alpha = \frac{n}{m}$. Vi lenker i hodet (konstant tid) og logger én kollisjon hvis bøtta ikke er tom.
- **Søk:**
  - Suksess: $\approx 1 + \alpha/2$ sammenligninger (finner et tilfeldig element i en kjede av forventet lengde α).
  - Feil: $\approx 1 + \alpha$ (må skumme hele kjeden).
- **Verste fall:** $O(n)$ hvis alle hasher til samme bøtte (degenererer til én lang liste).
- **Bygging fra fil:** $O(\sum L_i)$ for hashing + $O(n)$ for lenking.
- **Minne:** $O(m+n)$; hver node lagrer `string` + én peker.

Forventet antall kollisjoner ved innsetting: Vi teller én kollisjon når en ny nøkkel lander i en ikke-tom bøtte. Med uniform hashing er forventet antall ikke-tomme bøtter $m\big(1-(1-\frac{1}{m})^n\big)$, så får vi forventet belegg (balls into bins problem) fra statistikk :

$$\mathbb{E}[C] \;=\; n \;-\; m\Big(1-\Big(1-\tfrac{1}{m}\Big)^n\Big)\,.$$

Eksempel $n \approx 118$, $m = 128$: $\mathbb{E}[C] \approx 40.73$ og $\mathbb{E}[C]/n \approx 0.345 \;<\; 0.4$ (kravet).

Begrensninger

Det er fortsatt ulike begrensninger i løsningen. For eksempel, ingen resizing/rehash (siden i denne oppgaven er det bare 118 elementer), so hvis α blir høy, vokser kjedene. Vanlig praksis er å rehashe til neste primtall eller dobling når f.eks. $\alpha > 1$. Programmet sjekkes ikke for duplikater, samme navn kan legges inn flere ganger og vil telle som ekstra kollisjoner.

Vi ignorerer kun komma og mellomrom. Rekkefølge beholdes ("Etternavn, Fornavn" $\neq$ "Fornavn, Etternavn"). Ingen case-folding eller Unicode-normalisering (NFC/NFD), så "Å" vs "A + ring" behandles ulikt. Bruk av `wstring_convert`/`codecvt` i programmet fungerer til øvingen, men er deprekert i nyere C++ og robuste alternativer (ICU, egen UTF-8-parser) er utenfor scope. For valg av tabell og base, gir $m = 128$ (toerpotens) enkel %-reduksjon, valg av oddetallsbase (31) unngår artefakter i lave bit'er ved $m = 2^k$. Tilslutt skjer utskrift av kollisjoner i sanntid under innsetting/søk og påvirker kun I/O-tid, ikke de teoretiske kostnadene.

Deloppgave 2: Hashtabell og ytelser

I denne oppgaven har jeg implementert to hashtabeller med åpen adressering, lineær probing og dobbel hashing. I implementasjonen er det brukt Knuths multiplikative metode med gyldne snitt multiplikator som primær hashfunksjon:

$$h_1(x) = (x \cdot A) \,\%\, m$$

hvor $A = 11400714819323198485$ (Fibonacci hashing multiplikator), og m er tabellstørrelsen. Denne metoden er kjent for å gi en jevn fordeling av nøkler og unngå store klynger i starten.

For dobbel hashing er det i tillegg definert en sekundær hashfunksjon:

$$h_2(x) = 1 + (x \cdot B) \,\%\, (m - 1)$$

med $B = 7046029254386353131$, en annen multiplikativ konstant (brukt i RNG-er, i denne oppgaven for at `h2` skal være uforutsigbar og spre kollisjoner utover tabellen i stedet for å stille dem opp i klynger). Denne er valgt slik at steglengdene blir forskjellige for ulike nøkler, og for å garantere at $h_2(x) \neq 0$. Dermed unngår vi evige løkker og sikrer at hele tabellen kan besøkes. Effekten av å bruke to uavhengige multiplikative konstanter er at dobbel hashing gir mye bedre spredning av kollisjoner. I stedet for at mange nøkler havner i samme "klynge" (primary clustering, som i lineær probing), vil hver nøkkel følge sin egen probe-sekvens. Dette gjør at antall kollisjoner holder seg lavt, selv når fyllingsgraden nærmer seg 100%.

```
1  // Polynomial rolling/h1
2  static size_t hash_poly(uint64_t key, size_t tableSize) {
3      return (key * 11400714819323198485ULL) % tableSize;
           // Knuths multiplikativ
4  }
5
6  // h2
7  class HashTableDouble : public HashTable {
8      ...
9      static size_t h2(uint64_t key, size_t tableSize) {
10         return 1 + ((key * 7046029254386353131ULL) % (
               tableSize - 1));
11     }
```

Listing 4: `h1(keys)` og `h2(keys)`

Jeg har satt minste datamengden til 8 millioner slike at programmet skal produsere et resultat med fyllingsgrad på 100%. Programmet funker også med et datasett på minst 10 milioner tall men er ekstrem treg når fyllingsgrad er 100%. I tillegg bruker jeg også to ulike typer tabellstørrelser for lineær og dobbel (i et forsøk å forminske tiden til lineær probing), toerpotens for lineær og primtall for dobbel. Dette er fordi toerpotens lar man bruke bitmasking (`& (m - 1)`) som er mye raskere en `%`, mens primtall sikrer at `h2(x)` genererer trinn som dekker hele tabbelen og unngår sirkler. Dette er også en grunn til hvorfor jeg brukte 8 millioner som resultat siden tabellstørrelser for 8 millioner dataer er mest likt sammenlignet med 9 mil. eller 10 mil. (toerpotens : '8 388 608', primtall: '8 000 009')

```
1  bool contains(uint64_t key) const override {
2          size_t idx = hash_poly(key, tableSize);  // finn "
               home" slot
3
4          size_t start = idx;
5          while (used[idx]) {
6              if (table[idx] == key) return true;
7              // Bitmasking
8              idx = (idx + 1) & (tableSize - 1);     //Flytt
                   til neste slot om false
9
10             if (idx == start) break;
11         }
12         return false;
13     }
```

Listing 5: Bitmasking i lineær probing

```
1  void reset() {
2          fill(used.begin(), used.end(), uint8_t{0});
3          inserted = 0;
4          totalCollisions = 0;
5          maxProbes = 0;
6      }
```

Listing 6: Reset-funksjon

Jeg har også implementert en `reset()`-funksjon som nullstilles tabellen før hver kjøring, rydder brukte posisjoner og tellerne for kollisjoner. Dette sikrer at hvert eksperiment starter med en tom hashtabell, slik at fyllingsgraden og tidsmålingene blir korrekte. Under er tidtaking og kollisjon per insetting for lineær probing og dobbel hashing for fyllingsgrader:

Fra tabellene kan vi observere at for tidsbruket er lineær probing raskest når tabellen er halvtomt (50%), men er ekstremt treg når fyllingsgradene nærmer seg 100%. Dobbel hashing derimot bruker litt mer tid i begynnelsen, men skalerer seg mye bedre når tabellen blir full. Grunnen til at dobbel hashing er tregere i begynnelsen er fordi den beregner både `h1(keys)`/`hash_poly(keys)` og `h2(keys)` i motsetning til lineær probing (bare `h1(keys)`) selvom dette trengs nesten aldri i starten, koster det alikevel CPU-tid.

For kollisjoner gir lineær probing veldig mange kollisjoner når fyllingsgraden øker (kollisjoner per innsetting ved 100%). Dobbel hashing holder kollisjonene mye lavere, med bare kollisjoner per innsetting for 100% fyllingsgrad. Dette er fordi lineær probing lider av "primary clustering" hvor kollisjoner bygger på kollisjoner og lange sammenhengde blokker bygges, slik at dersom et nytt element havner i nærheten av en slik blokk kan måtte sjekke mange plasser på rad før den finner en ledig plass. Dobbel hashing bruker en sekvens med hopp basert på `h2(keys)` slik at kollisjonene spres ut over hele tabellen istedenfor å samle seg i blokker. "Secondary clustering" kan oppstå hvis `h2` er dårlig valgt (da får ulike nøkler like probe-sekvenser), men dette kan man unngå med bruk av gyld-

Fyllingsgrader	Lineær probing	Dobbel hashing
50%	1.279 sek	0.699 sek
80%	2.912 sek	1.885 sek
90%	2.091 sek	1.572 sek
99%	8.305 sek	2.734 sek
100%	112.051 sek	4.567 sek

Tabell 1: Tidsmåling for lineær probing og dobbel hashing

Fyllingsgrader	Lineær probing	Dobbel hashing
50%	7.534	0.385
80%	31.389	1.011
90%	4.468	1.557
99%	196.417	3.645
100%	3431.887	14.381

Tabell 2: Kollisjoner per innsetting

ne snitt multiplikator. Det blir dermed sjeldnere at man må prøve mange plasser etter hverandre, altså færre prober, jevnere oppførsel og totalt sett raskere.

Fra teori vet vi at gjennomsnittlig antall probes (kollisjonstester) vokser med fyllingsgraden $\alpha = n/m$:

$$\mathbb{E}[\text{probes}]_{\text{lineær probing}} \approx \tfrac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right), \qquad \mathbb{E}[\text{probes}]_{\text{dobbel hashing}} \approx \frac{1}{1-\alpha}.$$

Kollisjoner per innsetting estimeres som

$$\mathbb{E}[\text{collisions per insert}] \approx \mathbb{E}[\text{probes}] - 1.$$

Dette forklarer hvorfor lineær probing blir dramatisk treg når $\alpha \to 1$, mens dobbel hashing holder seg mer stabil.

Vi kan da konkludere med at høyere kollisjoner direkte fører til høyere tidsbruk, spesielt for lineær probing. Når fyllingsgraden nærmer seg 100%, blir lineær probing ubrukelig treg. Dobbel hashing fungerer bedre, men har også økt tidsbruk. Dermed bør grensen for hvor full en tabell bør være ligge under 80-90%. Dobbel hashing er mer robust ved høy fyllingsgrad, med langt færre kollisjoner og bedre tidsbruk. Lineær probing er enklere og raskere ved lav fyllingsgrad, men skalerer dårlig. Valg av tabellstørrelse (primtall vs toerpotens) har stor innvirkning på ytelsen.

Begrensninger og kompleksitet

Selv om implementasjonen fungerer korrekt, er det flere praktiske begrensninger:

- 100% fyllingsgrad er upraktisk i virkelige systemer, da både lineær probing og dobbel hashing får svært høy tidsbruk.

- Lineær probing lider av primary clustering, mens dobbel hashing kan få secondary clustering dersom $h_2(x)$ er dårlig valgt.
- Hashfunksjonens kvalitet er avgjørende: en dårlig funksjon gir mange kollisjoner.
- Sletting er ikke implementert; med åpen adressering krever dette en egen strategi (tombstones eller rehash).

Kompleksitetsmessig er forventet tidsbruk per innsetting/oppslag $O(1)$ så lenge fyllingsgraden α er under en viss grense (typisk 70–80%). I verste fall, når tabellen nærmer seg full, kan både lineær probing og dobbel hashing bli $O(n)$. Plassforbruket er $O(m)$, der m er tabellstørrelsen.