NLLR

Toteutusdokumentti

Leo Leppänen

24. helmikuuta 2014

Sisältö

1 Algoritmit

1.1 Argmax.single

```
public Result<T> single(Algorithm algorithm, ArrayList<T> args, Object
    [] constants){

  Object[] argList = new Object[constants.length + 1];
  System.arraycopy(constants, 0, argList, 1, constants.length);
  argList[0] = args.get(0);

  double maxVal = algorithm.calculate(argList);
  T maxArg = args.get(0);

  for (int i = 1; i < args.size(); i++) {
    argList[0] = args.get(i);
    double result = algorithm.calculate(argList);

    if (maxVal < result){
      maxVal = result;
      maxArg = args.get(i);
    }
  }
  return new Result(maxArg, maxVal);
}
```

Kuva 1: Argmax.single()

Tilavaativuus

Tilavaativuus on $\mathcal{O}(1)$, sillä algoritmit käyttää vakiomäärää muuttujia ja tulostaa aina yhden Result-olion.

Aikavaativuus

Argmax suorittaa syötteenään saamansa algoritmin kerran per syötteenä saadun argumenttilistan argumentti, joten aikavaativuus on $\mathcal{O}(A \times n)$, missä n on maksimoitavan argumentin kandidaattien määrä ja A on suoritettavan algoritmin aikavaativuus.

1.2 Argmax.multiple

```
public ArrayList<Result<T>> multiple(Algorithm algorithm, int amount,
    ArrayList<T> args, Object[] constants){
  ArrayList<Result<T>> results = new ArrayList<>();

  Object[] argList = new Object[constants.length + 1];
  System.arraycopy(constants, 0, argList, 1, constants.length);
  for (T arg : args){
    argList[0] = arg;
    double result = algorithm.calculate(argList);
    if (results.size() < amount){
      results.add(new Result(arg, result));
      sort(results);
    } else if (results.get(amount-1).getValue() < result){
      results.remove(amount-1);
      results.add(new Result(arg, result));
      sort(results);
    }
  }
  return results;
}
```

Kuva 2: Argmax.multiple()

```java
private void sort(ArrayList<Result<T>> results){
  for (int i = 1; i < results.size(); i++) {
    Result x = results.get(i);
    int j = i;
    while (j > 0 && results.get(j-1).getValue() < x.getValue()){
      results.set(j, results.get(j-1));
      j = j-1;
    }
    results.set(j, x);
  }
}
```

Kuva 3: Argmax.sort()

Tilavaativuus

Tilavaativuus on $\mathcal{O}(n)$, sillä kerrallaan muistissa pidetään korkeintaan $n+1$ Result-oliota sekä vakiomäärää muita muuttujia.

Aikavaativuus

Algoritmi suorittaa syötteenä saadun A aikavaativuuksisen algoritmin n kertaa, jolloin tältä osin aikavaativuus on $\mathcal{O}(A \times n)$. Lisäksi pahimmillaan n kertaa kutsutaan metodia sort(), joka järjestää tuloslistan.

Järjestysalgoritmina toimii InsertionSort. Järjestysalgoritmin valintaan vaikutti uniikki konteksti: jokaisella järjestyskerralla kaikki paitsi yksi alkio ovat valmiina oikeilla paikoillaan. Lisäksi järjestettävät taulukot erittäin pienikokoisia. Näissä tapauksissa InsertionSort on nopein ja tehokkain[1]. Tässä erityistapauksessamme aikavaativuus on lähempänä $\mathcal{O}(n)$ kuin $\mathcal{O}(n^2)$ ja tilavaativuus on $\mathcal{O}(1)$.

[1] http://dl.acm.org/citation.cfm?doid=359024.359026

1.3 TFIDF

```
public static double tfidf(String token, Document query, Corpus
    reference){
  int tf = query.getFrequency(token);
  double idf = idf(reference, token);
  return tf * idf;
}
```

Kuva 4: Tfidf.tfidf()

```
public static double idf(Corpus reference, String token) {
  int totalDocs = reference.getDocuments().size();
  int docsContainingTerm = reference.numOfDocsContainingToken(token);

  return Math.log( (double) totalDocs / docsContainingTerm);
}
```

Kuva 5: Tfidf.idf()

Tilavaativuus

Algoritmin tilavaativuus on $\mathcal{O}(1)$, sillä se käyttää syötteen koosta riippumatta vain vakiomäärää omia muuttujia.

Aikavaativuus

Algoritmin aikavaativuus voidaan laskea kahdessa osassa: *IDF*-algoritmin osuus sekä *TF-IDF*-algoritmin kokonaisaikavaativuus.

IDF-osion aikavaativuus on $\mathcal{O}(1)$. Rivin *reference.getDocuments().size();* suorittaminen vie $\mathcal{O}(1)$, sillä *referece.getDocuments()* palauttaa *ArrayList*-olion ajassa $\mathcal{O}(1)$ ja tältä oliolta voidaan edelleen kysyä ajassa $\mathcal{O}(1)$ sen kokoa. Kutsun *reference.numOfDocsContainingToken(token)* aikavaativuus on myöskin $\mathcal{O}(1)$, sillä kutsussa suoritetaan *HashMap*-olion *containsKey()*- ja *get()*-metodit, jotka molemmat ovat $\mathcal{O}(1)$. Myös lopussa suoritettava *Math.log((double) totalDocs / docsContainingTerm)* vie aikaa $\mathcal{O}(1)$, jolloin *IDF*-osion aikavaativuus on $\mathcal{O}(1+1+1) = \mathcal{O}(1)$

TF-IDF suorittaa yhden *IDF*-kutsun lisäksi yhden *query.getFrequency(token)*-kutsun, joka pahimmassa tapauksessa suorittaa ensin *HashMap.containsKey()*-kutsun ja sen jälkeen *HashMap.get()*-kutsun, joiden molempien aikavaativuus on $\mathcal{O}(1)$. Lisäksi suoritetaan lopuksi vakioaikainen laskutoimitus. Kokonaisuudessaan aikavaativuus on siis $\mathcal{O}(1+1) = \mathcal{O}(1)$

1.4 NLLR

```
public double calculateNllr(Document query, Corpus candidate){
  double nllr = 0;

  Object[] constants = {query, corpus};
  ArrayList<Result<String>> results = new Argmax().multiple(
    new Tfidf(),
    NUMBER_OF_TOKENS_TO_ANALYZE,
    query.getUniqueTokens().toArrayList(),
    constants);

  for(Result<String> result : results){
    String uniqueToken = result.getArgument();

    double tokenProbQuery = calculateTokenProbability(uniqueToken,
   query);
    double tokenProbCandidate = calculateTokenProbability(uniqueToken,
   candidate);
    double tokenProbCorpus = calculateTokenProbability(uniqueToken,
   corpus);

    nllr += tokenProbQuery * Math.log(tokenProbCandidate /
   tokenProbCorpus);
  }

  return nllr;
}
```

Kuva 6: Nllr.nllr()

```
public double calculateTokenProbability(String token, BagOfWords
    bagOfWords) {
  double prob = (double) bagOfWords.getFrequency(token) / bagOfWords.
    getTotalTokens();
  if (prob == 0){
    return NONZERO;
  } else {
    return prob;
  }
}
```

Kuva 7: Nllr.calculateTokenProbability()

Tilavaativuus

$\mathcal{O}(1)$, sillä algoritmi käyttää syötteensä lisäksi vain vakiomäärän tilaa bestTokens-taulukon sekä välitulokset tallentavien muuttujien muodossa.

Aikavaativuus

Algoritmi määrittää aluksi Argmax:lla vakiomäärän parhaan TF-IDF arvon saavia sanoja, joille sen jälkeen kullekin suoritetaan useita $\mathcal{O}(1)$ aikavaativuuksisia calculateTokenPropability-komentoja. Täten aikavaativuus on mallia

$$\mathcal{O}(Argmax.multiple(TFIDF, n) \times a + a \times (3 \times calculateTokenProbability + 1))$$

missä a on vakio, oletusarvoisesti 10 ja n on syötteenä saadun dokumentin sanojen määrä. Koska *TF-IDF* aikavaativuus on $\mathcal{O}(1)$ ja a on vakio, voidaan lauseketta sieventää seuraavaan muotoon: $\mathcal{O}(n + calculateTokenProbability)$.

Metodin *calculateTokenPropability* aikavaativuus puolestaan on $\mathcal{O}(1)$, sillä sen suorittamat metodikutsut ovat vakioaikaisia (*HashMap.containsKey()*, *HashMap.get()*. Näiden metodikutsujen lisäksi suoritetaan vain vakioaikaisia laskentakäskyjä.

Täten *NLLR*:n aikavaativuus on $\mathcal{O}(n + calculateTokenProbability)$, missä n on syötteenä saadun dokumentin sanojen määrä. Näin alhainen aikavaativuus on kuitenkin mahdollista vain, koska suurin osa laskennasta suoritetaan jo dokumentteja korpukseen lisättäessä.

2 Tietorakenteet

2.1 ArrayList

Tilavaativuus

ArrayList käyttää sisäisesti taulukkoa, joka on korkeintaan vakiokertoimen verran sen sisältämien alkioiden lukumäärää n suurempi. Lisäksi käytetään vakiomäärää kirjanpitomuuttujia. Täten pahimman tapauksen tilavaativuus on mallia $\mathcal{O}(x \times n)$, missä x on vakio ja tilavaativuus voidaan siten kirjoittaa muotoon $\mathcal{O}(n)$.

Aikavaativuus

Analysoidaan aikavaativuuden tärkeimpien komentojen osalta erikseen:

get(): Aina $\mathcal{O}(1)$, sillä teemme vakiomäärän vakioaikaisia kutsuja.

```
public T get(int index){
  if (validIndex(index)){
    return (T) array[index];
  }
  throw new IndexOutOfBoundsException();
}
```

Kuva 8: ArrayList.get()

add(): Pahimmassa tapauksessa $\mathcal{O}(n)$, mikäli joudumme kasvattamaan taustalla olevan taulukon kokoa. Koska taulukon koko kuitenkin kasvaa kahden potensseissa, joudumme tekemään tämän muutoksen vain n lisäyksen välein ja amortisoitu aikavaativuus on $\mathcal{O}(1)$

```
public void add(T value) {
  array[size] = value;
  size++;
  modCount++;
  checkCapacity();
}
```

Kuva 9: ArrayList.add()

size(): $\mathcal{O}(1)$, sillä suoritamme vain yhden vakioaikaisen komennon.

```
public int size(){
  return this.size;
}
```

Kuva 10: ArrayList.size()

indexOf(): Käymme pahimmassa tapauksessa koko listan suorittaen jokaiselle alkiolle vakiomäärän $\mathcal{O}(1)$ aikavaativuuksisia vertailuja, sekä lopulta suorittaen $\mathcal{O}(1)$ aikaisen palautuskutsun, jolloin aikavaativuus on $\mathcal{O}(n)$.

```
public int indexOf(T value){
  for (int i = 0; i < size; i++) {
    if ((value == null && array[i] == null) || (value != null && value.
    equals(array[i]))){
      return i;
    }
  }
  return -1;
}
```

Kuva 11: ArrayList.indexOf()

contains(): Kutsumme metodia *indexOf()* jonka aikavaativuus on $\mathcal{O}(n)$, ja suoritamme vakioaikaisen palautuksen. Täten aikavaativuus on yhteensä $\mathcal{O}(n)$.

```java
public boolean contains(T value){
  return indexOf(value) != -1;
}
```

Kuva 12: ArrayList.contains()

isEmpty(): Suoritamme vakioaikaisen tarkastuksen ja palautamme sen tuloksen. Lopputuloksena aikavaativuus $\mathcal{O}(1)$.

```java
public boolean isEmpty(){
  return size == 0;
}
```

Kuva 13: ArrayList.isempty()

clear(): Suoritamme vakiomäärän vakioaikaisia operaatiota, joten aikavaativuus on $\mathcal{O}(1)$.

```java
public void clear(){
  array = new Object[DEFAULT_SIZE];
  limit = DEFAULT_SIZE;
  modCount++;
  size = 0;
}
```

Kuva 14: ArrayList.clear()

remove(int index): Tarkistamme ensin ajassa $\mathcal{O}(1)$ onko indeksi käytössä. Tämän jälkeen siirrämme annettuun indeksiin ja sitä seuraavan indeksin alkion ja siirrymme seuraavaan indeksiin, jossa toistamme saman operaation. Täten pahimmassa tapauksessa (indeksi = 0) käymme läpi koko listan tehden $\mathcal{O}(1)$ aikavaativuuksisia sijoitusoperaatioita. Lopuksi tarkistamme onko listan kokoon tehtävä muutoksia, joka pahimmassa tapauksessa kestää $\mathcal{O}(n)$. Täten pahimma tapauksen aikavaativuus on $\mathcal{O}(n + n) = \mathcal{O}(n)$. Jälkimmäinen listan koon muutos amortisoituu aikavaativuuteen $\mathcal{O}(1)$, mutta tämä ei vaikuta lopullisen aikavaativuuden $\mathcal{O}$-notaatioon joka on jokatapauksessa $\mathcal{O}(n)$.

```
public void remove(int index){
  if (!validIndex(index)){
    throw new IndexOutOfBoundsException();
  }

  while(validIndex(index)){
    array[index] = array[index+1];
    index++;
  }
  size--;
  modCount++;
  checkCapacity();
}
```

Kuva 15: ArrayList.remove(int index)

remove(T value): Pahimmassa tapauksessa suoritamme ensin $\mathcal{O}(n)$ aikaisen *indexOf*-kutsun ja sen jälkeen kutsumme $\mathcal{O}(n)$ aikaista *remove(int index)*-metodia. Täten pahimman tapauksen aikavaativuus on $\mathcal{O}(n+n) = (O)n$.

```
public void remove(T value){
  int index = indexOf(value);
  if (index != -1){
    remove(index);
  }
}
```

Kuva 16: ArrayList.remove(T value)

set(): Aikavaativuus on $\mathcal{O}(1)$, sillä suoritamme vain vakiomäärän vakioaikaisia operaatioita.

```
public void set(int index, T value){
  if (index <= size && index >= 0){
    array[index] = value;
  }
}
```

Kuva 17: ArrayList.set()

Taustalla olevan taulukon koon muutokset vievät, kuten yllä esitettyä $\mathcal{O}(n)$, sillä luomme uuden taulukon ajassa $\mathcal{O}(1)$ ja käymme aiemman taulukon läpi ajassa $\mathcal{O}(n)$, suorittaen jokaisella n alkiolle $\mathcal{O}(n)$ aikavaativuuksisen sijoitusoperaation uuteen taulukkoon. Tämän lisäksi suoritetaan vain vakiomäärä vakioaikaisia operaatioita.

```
private void changeCapacity(int newLimit){
  Object[] newArray = new Object[newLimit];
  int smallerLimit = Math.min(newLimit, limit);
  System.arraycopy(array, 0, newArray, 0, smallerLimit);
  array = newArray;
  limit = newLimit;
}
```

Kuva 18: ArrayList.changeCapacity()

2.2 HashMap

Tilavaativuus

HashMap:n tilavaativuus on pahimmassa tapauksessa, kaikkien hajautusarvojen osuessa yhteen hajautustaulun ylivuotoketjuun, $\mathcal{O}(taulukon_k oko + n)$, missä $taulkon_k oko$ on hajautustaulun taustalla olevan taulukon koko, ja n ym. ylivuotoketjussa olevien alkioiden, ja samalla taulukon kaikkien alkioiden määrä. $taulukon_k oko$ puolestaan on pahimmassa tapauksessa, $x \times n$, missä x on jokin vakio, meidän tapauksessamme noin 2. Täten tilavaativuus on pahimmillaan $\mathcal{O}(2n) = \mathcal{O}(n)$. n puolestaan on tarkemmin sanoen muotoa $|K| + |V|$, missä $|K|$ on hajautustaulun avainten tilavaativuus ja $|V|$ hajautustaulun arvojen tilavaativuus. Täten Lopullinen tilavaativuutemme on $\mathcal{O}(|K| + |V|)$.

Aikavaativuus

Analysoimme aikavaativuuden erikseen jokaiselle metodille.

get(): Pahimmassa tapauksessa kaikki hajautustaulun alkiot ovat samassa ylivuotoketjussa, jolloin löydämme oikean ylivuotoketjun ajassa $\mathcal{O}(1)$, minkä jälkeen käymme koko ketjun läpi ajassa $\mathcal{O}(n)$. Täten pahimman tapauksen aikavaativuus on $\mathcal{O}(n)$. Amortisoidusti hajautustaulun arvot kuitenkin jakautuvat tasaisesti koko taulukon alueelle, jolloin aikavaativuus on $\mathcal{O}(1 + x)$, missä x on keskimääräinen ylivuotoketjun pituus. Tämän ollessa yksi tai hyvin lähellä sitä, on amortisoitu aikavaativuus siten käytännössä $\mathcal{O}(1)$.

```java
public V get(K key){
  // Determine in which bucket the Entry should be in
  int index = getIndex(key, size);
  Entry entry = this.array[index];

  //Go thru that bucket, looking for the key
  while(entry != null) {
    if (entry.getKey().equals(key)){
      return (V) entry.getValue();
    }
    entry = entry.getNext();
  }

  //If bucket is empty or no key found in bucket, return null
  return null;
}
```

Kuva 19: HashMap.get()

HashMap.put(): Löydämme oikean ylivuotolistan ajassa $\mathcal{O}(1)$. Pahimmassa tapauksessa kaikki hajautustaulun arvot ovat samassa ylivuotoketjussa, jolloin joudumme käymään läpi koko ylivuotoketjun ajassa $\mathcal{O}(n)$ennen kuin lisäämme arvon ketjun alkuun ajassa $\mathcal{O}(1)$.

Tämän jälkeen joudumme vielä pahimmassa tapauksessa uudelleenhajauttamaan koko hajautustaulun ajassa $\mathcal{O}(n)$. Täten pahimman tapauksen aikavaativuus on $\mathcal{O}(n+n) = \mathcal{O}(n)$. Keskimäärin joudumme kuitenkin uudelleenhajauttamaan taulun n lisäyksen välein, joten amortisoidusti lisäyksen osuus aikavaativuudesta on $\mathcal{O}(1)$.

Samoin yhdessä ylivuotoketjussa on amortisoidusti 0 (tai 1, kuitenkin käytännössä hyvin lähellä nollaa) alkiota, jolloin lisäyksenkin kesto on amortisoidusti $\mathcal{O}(1)$. Täten koko armotisoitu aikavaativuus on $\mathcal{O}(1)$.

```
public void put(K key, V value){
  //Determine which bucket the key belongs to
  int index = getIndex(key, size);

  //If the bucket is empty, add the Entry as a new bucket
  if (this.array[index] == null){
    Entry e = new Entry<>(key, value);
    this.array[index] = e;
  } else {
    //An existing bucket was found, go thru the bucket looking for the
     key
    Entry entry = this.array[index];
    while (entry != null){
      if (entry.getKey().equals(key)){
        //Found key, overwrite current value and exit
        entry.setValue(value);
        return;
      }
      entry = entry.getNext();
    }

    //Didn't find key in bucket, add a new entry to start of bucket
    Entry newEntry = new Entry<>(key, value);
    newEntry.setNext(array[index]);
    array[index] = newEntry;
  }
  entries++;
  modCount++;
  checkCapacity();
}
```

Kuva 20: HashMap.put()

HashMap.remove(): Löydämme oikean ylivuotolistan ajassa $\mathcal{O}(1)$. Pahimmassa tapauksessa, kaikkien alkioiden ollessa samassa ylivuotolistassa, joudumme jälleen käymään ylivuotolistaa läpi ajassa $\mathcal{O}(n)$. Itse poisto-operaatio on kuitenkin $\mathcal{O}(1)$, sillä suoritamme vakiomäärän vakioaikaisia operaatioita.

Samoin kuin yllä, mahdollinen taustalistan koon muutos on pahimmassa tapauksessa $\mathcal{O}(n)$, mutta amortisoituu aikaa $\mathcal{O}(1)$. Samoin yllä esitetyllä logiikalla ylivuotolistan läpikäynti vie keskimäärin $\mathcal{O}(1)$, jolloin saamme pahimman tapauksen aikavaativuudeksi $\mathcal{O}(n)$, mutta amortisoiduksi aikavaativuudeksi $\mathcal{O}(1)$.

```java
public void remove(K key){
  //Find the correct bucket
  int index = getIndex(key, size);
  Entry e = array[index];

  //If no bucket present, just exit
  if (e == null){
    return;
  }

  //Search the bucket for the key
  Entry previous = null;
  while (e != null){
    Entry next = e.getNext();
    if (e.getKey().equals(key)){
      //Found our entry!
      //If entry is head, update array, else update prev's next().
      if (previous == null){
        array[index] = e.getNext();
      } else {
        previous.setNext(next);
      }

      entries--;
      modCount++;
      checkCapacity();
      return;
    }
    previous = e;
    e = next;
  }
}
```

Kuva 21: HashMap.remove()

HashMap.containsKey(): Oikea ylivuotoketju selviää ajassa $\mathcal{O}(1)$. Pahimmassa tapauksessa joudumme käymään läpi koko ylivuotoketjun ajassa $\mathcal{O}(n)$ tehden kullekin alkiolle vakiomäärän vakioaikaisia operaatioita ajassa $\mathcal{O}(1)$. Pahimman tapauksen aikavaativuus on siis $\mathcal{O}(n)$. Amortisoidusti metodin aikavaativuus kuitenkin on - kuten yllä - $\mathcal{O}(1)$.

```
public boolean containsKey(K key){
  if (key == null){
    return false;
  }

  int index = getIndex(key, size);
  Entry e = array[index];
  while (e != null){
    if (e.getKey().equals(key)){
      return true;
    }
      e = e.getNext();
    }
  return false;
}
```

Kuva 22: HashMap.containsKey()

HashMap.getIndex(): Suoritamme aina vakiomäärän vakioaikaisia kutsuja, joten aikavaativuus on aina $\mathcal{O}(1)$. Metodin toimintalogiikkaa on avattu enemmän JavaDocissa.

```java
private int getIndex(K key, int size){
  int hash = key.hashCode();
  hash ^= (hash >>> 20) ^ (hash >>> 12) ^ (hash >>> 7) ^ (hash >>> 4);

  int index = hash & (size - 1);
  return index;
}
```

Kuva 23: HashMap.getIndex()

Emme esitä tässä taustataulukon koon muutosten koodia sen pituuden vuoksi. Käytännössä kuitenkin tarkistamme ajassa $\mathcal{O}(1)$ onko muutokselle tarvetta. Mikäli tarvetta on, luomme ajassa $\mathcal{O}(1)$ uuden taulukon josta tulee uusi taustataulukkomme. Tämän jälkeen suoritamme uudelleenhajautuksen.

Uudelleenhajautuksessa käymme läpi kaikki taulukon n alkiota ajassa $\mathcal{O}(n)$, suorittaen jokaiselle vakiomäärän komentoja joilla alkiolle selvitettään indeksi uudessa taulukossa ajassa $\mathcal{O}(1)$ ja sijoitetaan se oikean indeksin ylivuotoketjun alkuun ajassa $\mathcal{O}(1)$. Täten uudelleenhajautus vie kokonaisuutenaan aikaa $\mathcal{O}(n)$.

2.3 HashSet

HashSet-toteutuksemme on käytännössä wrapperi *HashMap*-rakenteen ympärille, joten aika- ja tilavaativuutemme ovat erittäin vastaavat.

Tilavaativuus

Tilavaativuus on sama kuin n alkiota sisältävän *HashMap*-tietorakenteen, kuitenkin siten että $|V|$ on $x \times n$, missä x on vakio ja n on setin sisältämien alkoiden määrä. Tämä johtuu siitä kuinka asetamme kaikkii *HashMap*in avain-arvo -pareihin arvoksi viitteen samaan *DUMMY*-olioon.

Lisäksi käytämme vakiomäärää vakiokokoisia apumuuttujia, jotka eivät vaikuta lopulliseen tilavaativuuteemme, joka on siis $\mathcal{O}(|K| + n)$, missä $|K|$ on setin sisältämä arvojoukko ja n on *DUMMY*-olio ja k kappaletta viitteitä siihen.

Aikavaativuus

Tarkastellaan aikavaativuuksia erikseen jokaiselle metodille.

HashSet.add() tarkistaa ensin (keskimäärin) ajassa $\mathcal{O}(1)$ onko arvo jo setissä, ja jos ei ole niin lisää sen edelleen (keskimäärin) ajassa $\mathcal{O}(1)$. Täten keskimääräinen aikavaativuus on $\mathcal{O}(1)$.

Pahin tapaus on kuitenkin $\mathcal{O}(n + n) = \mathcal{O}(n)$, mikäli kaikki arvot ovat hajautustaulun samassa ylivuotoketjussa.

```java
public void add(E e){
  if (e != null){
    if (!map.containsKey(e)) {
      map.put(e, DUMMY);
      size++;
      modCount++;
    }
  }
}
```

Kuva 24: HashSet.add()

HashSet.remove() tarkistaa ensin (keskimäärin) ajassa $\mathcal{O}(1)$ onko arvo rakenteessa ja mikäli on, poistaa sen (keskimäärin) ajassa $\mathcal{O}(1)$.

Samoin kuin yllä, pahimmassa tapauksessa taustalla olevan *HashMap*in kaikki arvot ovat samassa ylivuotolistassa ja molemmat operaatiot vievät $\mathcal{O}(n)$ aikaa. Täten pahimman tapauksen aikavaativuus on $\mathcal{O}(n+n) = \mathcal{O}(n)$, mutta amortisoitu aikavaativuus on $\mathcal{O}(1)$.

```java
public void remove(E e){
  if (e != null){
    if (map.containsKey(e)) {
      map.remove(e);
      size--;
      modCount++;
    }
  }
}
```

Kuva 25: HashSet.remove()

HashSet.contains() on vain wrapperi *HashMap.containsKey()*-metodille ja sen aikavaativuus on sama kuin em. metodilla, eli pahimmassa tapauksessa $\mathcal{O}(n)$, mutta amortisoidusti $\mathcal{O}(1)$.

```java
public boolean contains(E e){
  return map.containsKey(e);
}
```

Kuva 26: HashSet.contains()

HashSet.size() suorittaa vakiomäärän vakioaikaisia operaatioita ja on siten aikavaativuudeltaan $\mathcal{O}(1)$.

```java
public int size(){
  return size;
}
```

Kuva 27: HashSet.size()

HashSet.isEmpty() suorittaa vakiomäärän vakioaikaisia operaatioita ja on siten aikavaativuudeltaan $\mathcal{O}(1)$.

```java
public boolean isEmpty(){
  return size == 0;
}
```

Kuva 28: HashSet.isEmpty()

Metodit *HashSet.removeAll()* ja *HashSet.addAll()* suorittavat molemmat n kertaa oman yksittäismetodinsa ja vastaavat siten n kappaletta kyseisiä kutsuja.