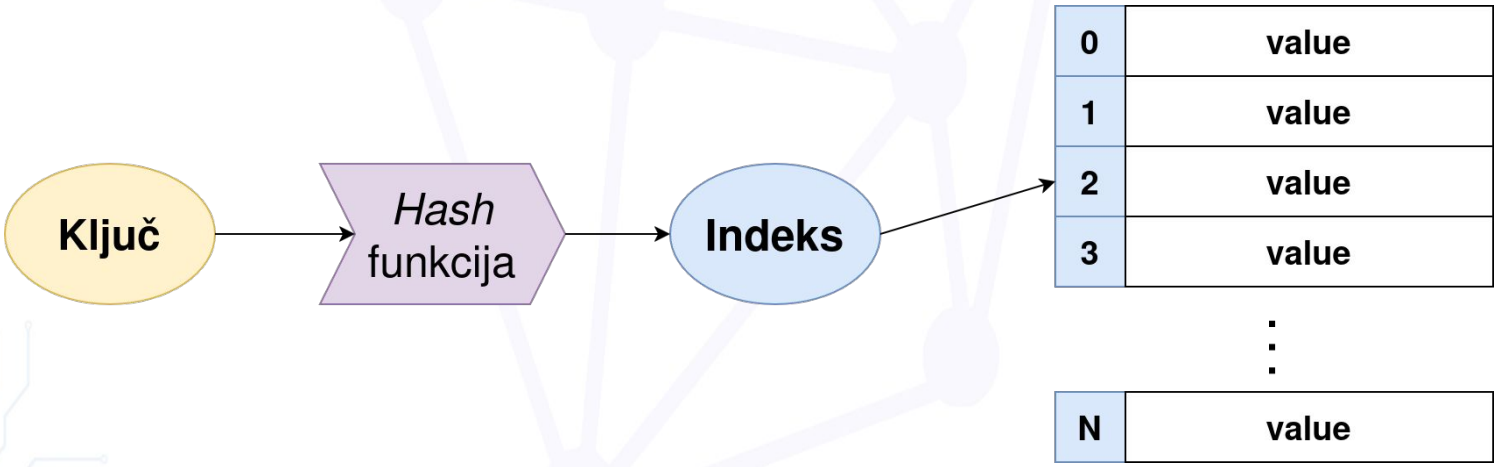
Teorija Algoritama

Hash tabele

Hash tabele (mape)

- Ideja iza *Hash* tabela je da nudi operacije dodavanja, brisanja i pretrage u konstantnim vremenima.
- *Hash* tabela je struktura podataka koja koristi *Hash* funkciju za mapiranje ključeva na odgovarajuće indekse u nizu.



Hash funkcija

- *Od Hash* funkcija zavise performanse *Hash* tabela.
- Mora da ispunjava 3 uslova:
  - Determinizam - Za isti ulaz uvek mora dati isti izlaz;
  - Efikasnost - Izračunavanje mora biti veoma brzo (***O(1)***);
  - Uniformna distribucija - Treba da raspoređuje ključeve što ravnomernije kako bi se izbeglo gomilanje na istim mestima.
- Ukoliko za dva (ili više) ključa *Hash* funkcija daje isti indeks, dolazi do kolizije.
- Kolizija se može rešiti:
  - Ulančavanjem - Svaki element u tabeli je pokazivač na jednostruko spregnutu listu;
  - Otvoreno adresiranje - Ako je mesto zauzeto, traži se naredno;
    - Linearno ispitivanje (probing) - Traži se redom naredno mesto (*i + 1, i + 2, ...*);
    - Kvadratno ispitivanje - Koristi se kvadratna funkcija za razmak;
    - Dvostruko heširanje - Koristi se druga *Hash* funckija kako bi se odradio korak pretrage.
- Faktor opterećenja - računa se kao količnik broja unetih elemenata i veličine tabele.
  - Kada pređe granicu od oko 0.75, performanse naglo opadaju;
  - Rešenje: Realocira se veća mapa, a svi elementi se ponovno heširaju u novi prostor.

Hash mape (lančanje) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)

    idx = hash(key)
    curr = map.buckets[idx]

    while(curr != NULL)
        if(curr->key = key)
            # Logika za duplikate
        curr = curr.next;

    newNode = makeNode(key, value)
    if(!newNode)
        # Logika za neuspešnu alokaciju

    ...
```

Hash mape (lančanje) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)

    ...

    newNode.next = map.buckets[idx]
    map.buckets[idx] = newNode
    map.count++

    if(map.count / map.size > LIMIT)
        rehash(map)
```

Hash mape (lančanje) - reheširanje (*pseudo-kod*)

```
function rehash(map)

    oldSize = map.size
    newSize = oldSize * MULTIPLICATOR
    oldBuckets = map.buckets

    map.buckets = calloc(newSize, SIZE)
    if(!map.buckets)
        map.buckets = oldBuckets
        return
    map.size = newSize
    map.count = 0

    ...
```

Hash mape (lančanje) - reheširanje (*pseudo-kod*)

```
function rehash(map)

    ...

    for i = 0 to oldSize
        curr = oldBuckets[i]
        while(curr != NULL)
            temp = curr
            curr = curr.next

            idx = hash(temp.key)
            temp.next = map.buckets[idx]
            map.buckets[idx] = temp
            map.count++

    free(oldBuckets)
```

Hash mape (lančanje) - brisanje (*pseudo-kod*)

```
function delete(map, key)

    idx = hash(key)
    curr = map.buckets[idx]
    prev = NULL

    while(curr != NULL)
        if(curr->key == key)
            if(!prev)
                map.buckets[idx] = curr.next
            else
                prev.next = curr.next
            free(curr)
            map.count--
        prev = curr
        curr = curr.next
```

Hash mape (lančanje) - pretraga (*pseudo-kod*)

```
function find(map, key)

    idx = hash(key)
    curr = map.buckets[idx]

    while(curr != NULL)
        if(curr.key == key)
            return curr
        curr = curr.next

    return NULL
```

Hash mape (probing) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)

    if(map.count / map.size > LIMIT)
        rehash(map)

    idx = hash(key)
    firstDeletedIdx = -1

    for i = 0 to map.size
        currIdx = (idx + i) % map.size
        curr = map.table[currIdx]

    ...
```

Hash mape (probing) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)

    ...

    for i = 0 to map.size

        ...

        if(curr.state == EMPTY)
            targetIdx = firstDeletedIdx != -1 ?
                                        firstDeletedIdx : currIdx

            map.table[targetIdx].key = key
            map.table[targetIdx].value = value
            map.table[targetIdx].state  = OCCUPIED
            map.count++
            return
        ...
```

Hash mape (probing) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)
    ...
    for i = 0 to map.size
        ...
        if(curr.state == OCCUPIED && curr.key = key)
            # Logic for key duplicates
        if(curr.state == DELETED && firstDeltedIdx == -1)
            firstDeletedIdx = currIdx
        ...
```

Hash mape (probing) - dodavanje (*pseudo-kod*)

```
function insert(map, key, value)

    ...

    if(firstDeletedIdx != -1)
        map.table[firstDeletedIdx].key = key
        map.table[firstDeletedIdx].value = value
        map.table[firstDeletedIdx].state  = OCCUPIED
        map.count++
```

Hash mape (probing) - pretraga (*pseudo-kod*)

```
function find(map, key)

    idx = hash(key)

    for i = 0 to map.size
        currIdx = idx + i
        curr = map.table[currIdx]

        if(curr.state == EMPTY)
            return NULL

        if(curr.state == OCCUPIED && curr.key == key)
            return curr

    return NULL
```

Hash mape (probing) - brisanje (*pseudo-kod*)

```
function delete(map, key)

    toDelete =  find(map, key)

    if(toDelete)
        toDelete.state = DELETED
        map.count--
```

Zadatak 1

- Implementirati *Hash* mapu, u kojoj se problemi kolizije rešavaju lančanjem.
- Ključ je tekstualnog tipa, a vrednost razlomljeni broj.
- *Hash* funkcija je *sdbm* funkcija:

```
sdbm2hash(map, key)
    hash = 0 # Očekivati veliku vrednost
    i = 0
    while(c = key[i++])
        hash = c + (hash << 6) + (hash << 16) - hash
    return hash % map.size
```

- Inicijalni kapacitet mape je 4, a svakim *rehashing*-om kapacitet se duplira.
  - *Rehash* se poziva kada faktor opterećenja pređe 0.75.

Zadatak 2

- Implementirati *Hash* mapu, u kojoj se problemi kolizije rešavaju kvadratnim probinom (*i + 1, i + 4, i + 9, ...*).
- Ključ je JMBG, a vrednost su podaci o osobi (ime, prezime, zanimanje).
- *Hash* funkcija je *djb2* funkcija:

```
djb2hash(map, key)
    hash = 5381
    i = 0
    while(c = key[i++])
        hash = hash * 33 + c
    return hash % map.size
```

- Inicijalni kapacitet mape je 7, a pozivom *rehash*-a kapacitet se postavlja na sledeći prost broj, koji je bar 2 puta veći od prethodnog kapaciteta.
  - *Rehash* se poziva kada faktor opterećenja pređe 0.5.