Structuri de Date și Algoritmi
Tabele de dispersie

Mihai Nan

Departamentul de Calculatoare
Facultatea de Automatică și Calculatoare
Universitatea POLITEHNICA din București

Anul Universitar 2022–2023

Conținutul cursului

1. **Introducere și definiție**

2. **Mecanisme de tratare pentru coliziuni**
   - Rezolvarea coliziunilor prin înlănțuire
   - Rezolvarea coliziunilor prin căutare liniară
   - Rezolvarea coliziunilor prin dispersie dublă

Motivație

- Există multe aplicații care necesită folosirea unei structuri de date care să asigure implementări eficiente pentru operațiile **Inserare**, **Căutare**, **Ștergere**.

Ce structuri de date, din cele studiate până acum, putem folosi într-un astfel de caz?

Motivație

- Există multe aplicații care necesită folosirea unei structuri de date care să asigure implementări eficiente pentru operațiile **Inserare**, **Căutare**, **Ștergere**.

Ce structuri de date, din cele studiate până acum, putem folosi într-un astfel de caz?

Putem utiliza arbori de căutare.

Motivație

- Există multe aplicații care necesită folosirea unei structuri de date care să asigure implementări eficiente pentru operațiile **Inserare**, **Căutare**, **Ștergere**.

Ce structuri de date, din cele studiate până acum, putem folosi într-un astfel de caz?

Putem utiliza arbori de căutare.

Aceste operații au o complexitate care depinde de structura arborelui.

Motivație

Există vreo structură de date care să permită *Căutarea* unui element aflat la o anumită poziție în $O(1)$?

Motivație

Există vreo structură de date care să permită *Căutarea* unui element aflat la o anumită poziție în $O(1)$?

Acest lucru se întâmplă în cazul vectorilor.

- Pornind de la această informație, ne dorim să obținem o variantă generalizată pentru o structură de date care să implementeze conceptul de dicționar (avem o mulțime de perechi de forma **cheie** și **valoare**).
- În particular, vectorul poate fi perceput drept un dicționar în care **cheia** este indexul elementului, iar **valoarea** este chiar elementul aflat la indexul respectiv.

Tabele de dispersie

O tabelă de dispersie este o structură de date eficientă pentru implementarea conceptului de dicționar.

- Deși căutarea unui element într-o tabelă de dispersie poate necesita la fel de mult timp ca operația de căutarea a unui element într-o listă înlănțuită: o complexitate de $\Theta(n)$ în cazul cel mai defavorabil; în practică, ne folosim de conceptul de **dispersie** pentru a remedia acest inconvenient.

Funcție de dispersie

O **funcție de dispersie** (cunoscută și sub numele de funcție hash) este o funcție matematică utilizată pentru a transforma o valoare de intrare (numită cheie) într-o valoare de ieșire, denumită adesea cod de dispersie. Scopul acestei transformări este de a atribui cheilor valori unice și predictibile, ceea ce facilitează căutarea rapidă a cheilor într-o structură de date numită tabel de dispersie sau hash table.

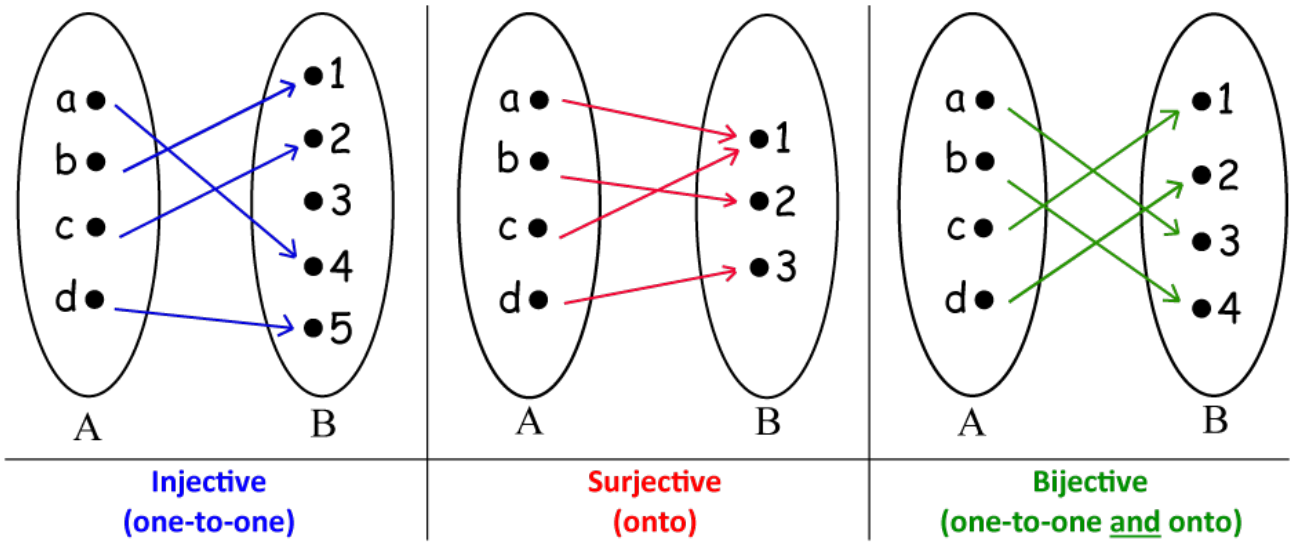
Funcție de dispersie

- Funcția de dispersie aplică o operație matematică asupra cheii și produce un indice, care este utilizat pentru a accesa o anumită poziție din tabelul de dispersie. În ideal, funcția de dispersie ar trebui să producă coduri de dispersie diferite pentru fiecare cheie, pentru a minimiza coliziunile (când două chei produc același cod de dispersie) și pentru a maximiza performanța.
- Funcțiile de dispersie sunt utilizate într-o gamă largă de aplicații, inclusiv în bazele de date, algoritmi de criptare, algoritmi de căutare și sortare și multe altele.

În practică, funcția de dispersie **NU** va putea să fie definită ca o funcție **bijectivă**.

Funcție de dispersie

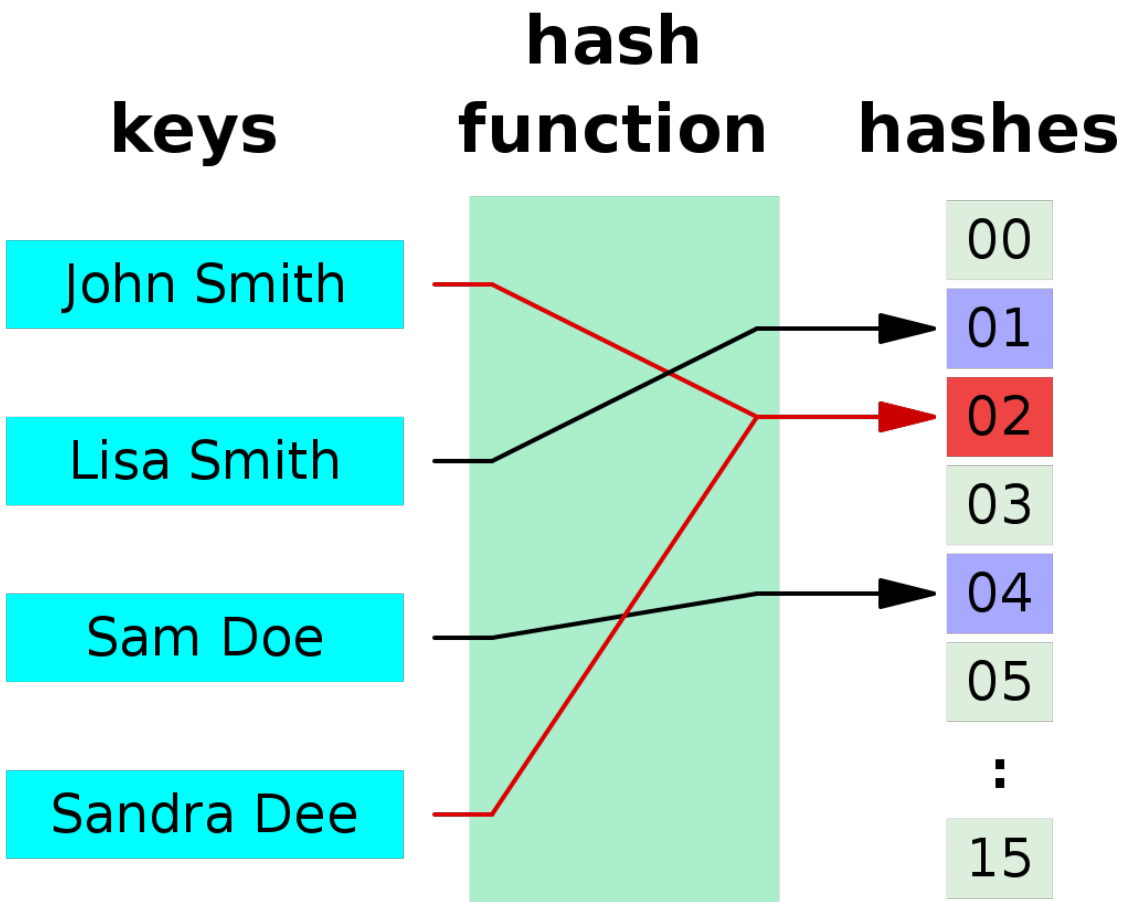
În mod ideal



Calcworkshop.com

Funcție de dispersie

În practică



Tabele de dispersie

Scopul principal

Ne dorim să implementăm o structură de date care să permită o operație similară cu indexarea pentru un vector în care indexul să poată avea alt tip.

structure[key] = value, unde **key** nu este un număr natural

Cum am putea obține o astfel de structură de date?

Tabele de dispersie

Scopul principal

Ne dorim să implementăm o structură de date care să permită o operație similară cu indexarea pentru un vector în care indexul să poată avea alt tip.

structure[key] = value, unde **key** nu este un număr natural

Cum am putea obține o astfel de structură de date?

Putem folosi un vector și o funcție de dispersie pentru a converti **key** într-un număr natural!

structure[hash(key)] = value, **hash(key)** este un număr natural

Tabele de dispersie – Exemplu

- Vrem să implementăm un **Catalog** în care va trebui să reținem pentru fiecare **Student** ce **Notă** a obținut la materia SDA.

Cum putem modela o astfel de problemă utilizând o tabelă de dispersie?

Tabele de dispersie – Exemplu

- Vrem să implementăm un **Catalog** în care va trebui să reținem pentru fiecare **Student** ce **Notă** a obținut la materia SDA.

Cum putem modela o astfel de problemă utilizând o tabelă de dispersie?

Considerăm un vector în care **vect[hash(key)] = value**

- **key – Student**
- **value – Notă**
- **hash(key)** – ID-ul unic al fiecărui student

Tabele de dispersie – Exemplu

- Considerăm că **ID**-ul unic al fiecărui student este un număr natural compus din 5 cifre (ex. 12345, 22334, 22694).

Câte elemente va trebui să aibă vectorul nostru?

Tabele de dispersie – Exemplu

- Considerăm că **ID**-ul unic al fiecărui student este un număr natural compus din 5 cifre (ex. 12345, 22334, 22694).

Câte elemente va trebui să aibă vectorul nostru?

10^5

Tabele de dispersie – Exemplu

- Considerăm că **ID**-ul unic al fiecărui student este un număr natural compus din 5 cifre (ex. 12345, 22334, 22694).

Câte elemente va trebui să aibă vectorul nostru?

10^5

O astfel de abordare este eficientă din punct de vedere al timpului, $O(1)$, dar ineficientă din perspectiva spațiului!

Tabele de dispersie – Exemplu

Cum am putea eficientiza din punct de vedere al memoriei?

Tabele de dispersie – Exemplu

Cum am putea eficientiza din punct de vedere al memoriei?

Presupunem că avem înscriși 150 de studenți și folosim un vector ce are doar 150 elemente.

$$\textbf{hash(stud)} = \textbf{ID(stud)} \bmod \textbf{150}$$

Tabele de dispersie – Exemplu

Cum am putea eficientiza din punct de vedere al memoriei?

Presupunem că avem înscriși 150 de studenți și folosim un vector ce are doar 150 elemente.

$$\textbf{hash(stud)} = \textbf{ID(stud) mod 150}$$

Ce limitări are o astfel de abordare?

$\textbf{12345} = \textbf{82 * 150 + 45}$ și $\textbf{12495} = \textbf{83 * 150 + 45}$

Tabele de dispersie

Definiție

Atunci când definim o structură de date de tip **Tabelă de dispersie**, trebuie să furnizăm următoarele:

1. o structură în care să reținem elementele (pentru care specificăm dimensiunea);
2. o funcție de dispersie: **hash(key)**;
3. un mecanism de tratare al coliziunilor.

În contextul funcțiilor de dispersie, coliziunea se referă la situația în care două sau mai multe chei diferite produc același cod de dispersie sau indice în tabelul de dispersie. Aceasta poate apărea din cauza limitărilor funcției de dispersie sau a numărului mare de chei care trebuie stocate într-un tabel de dispersie.

Mecanisme de tratare pentru coliziuni

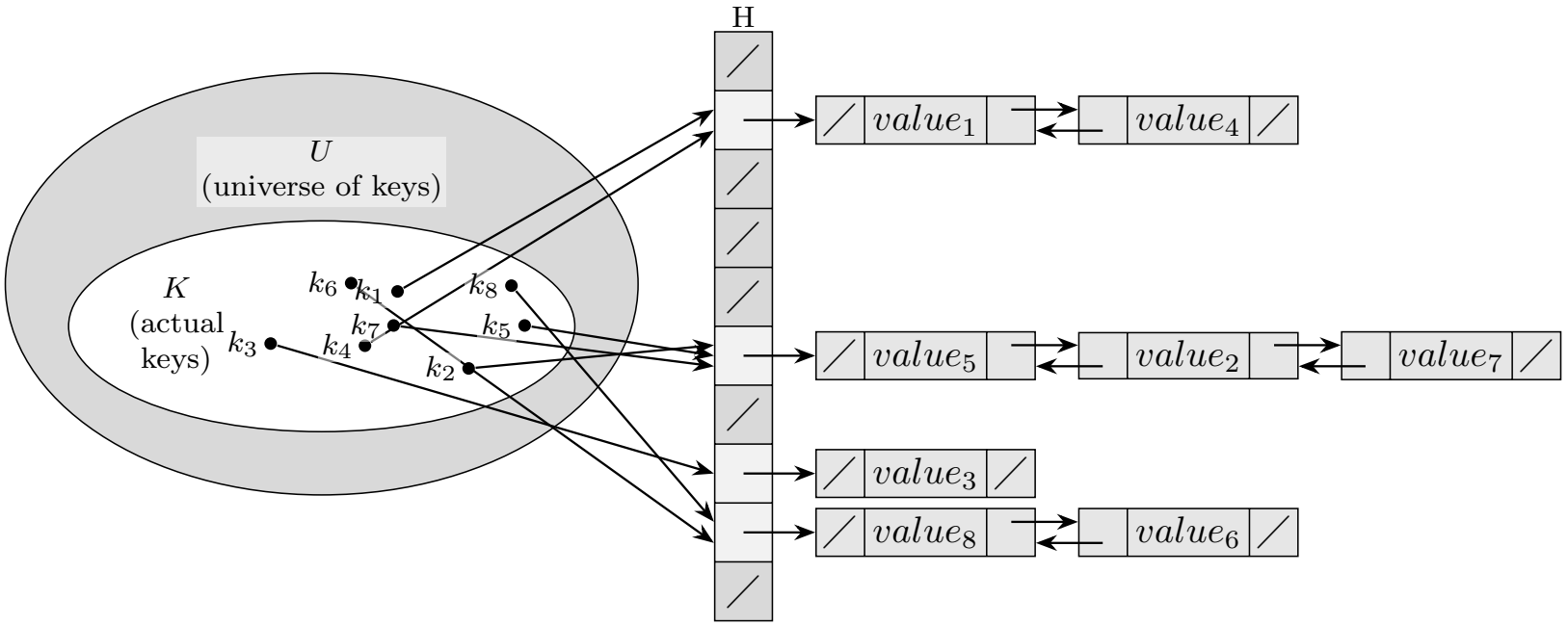
1. **Separate Chaining** (lanțuri separate)
   Această metodă implică crearea unei liste înlănțuite pentru fiecare indice din tabelul de dispersie. În cazul în care două chei diferite au același indice, valorile corespunzătoare sunt stocate în aceeași listă. Atunci când se caută o valoare pentru o anumită cheie, se parcurge lista corespunzătoare și se returnează prima valoare care se potrivește cu cheia. Această metodă este ușor de implementat și este eficientă pentru stocarea unui număr relativ mic de elemente în tabelul de dispersie.
2. **Linear Probing** (căutare liniară)
   Această metodă implică căutarea secvențială a următorului indice disponibil atunci când un indice este deja ocupat de o altă cheie. De exemplu, dacă o cheie este atribuită la indicele 5 și acest indice este deja ocupat, se va căuta următorul indice disponibil (6) și se va stoca valoarea acolo.

Mecanisme de tratare pentru coliziuni

3. **Quadratic Probing** (căutare pătratică)
   Această metodă implică căutarea următorului indice disponibil utilizând o serie de formule pătratice. De exemplu, dacă o cheie este atribuită la indicele 5 și acest indice este deja ocupat, se va căuta următorul indice disponibil utilizând formula: $5 + 1^2$, apoi $5 + 2^2$, apoi $5 + 3^2$ și așa mai departe.
4. **Double Hashing** (dublu hashing)
   Această metodă implică utilizarea a două funcții de dispersie pentru a determina următorul indice disponibil. În primul rând, se utilizează prima funcție de dispersie pentru a determina indicele inițial. Dacă acest indice este deja ocupat, se utilizează a doua funcție de dispersie pentru a determina un pas suplimentar de deplasare către următorul indice disponibil.

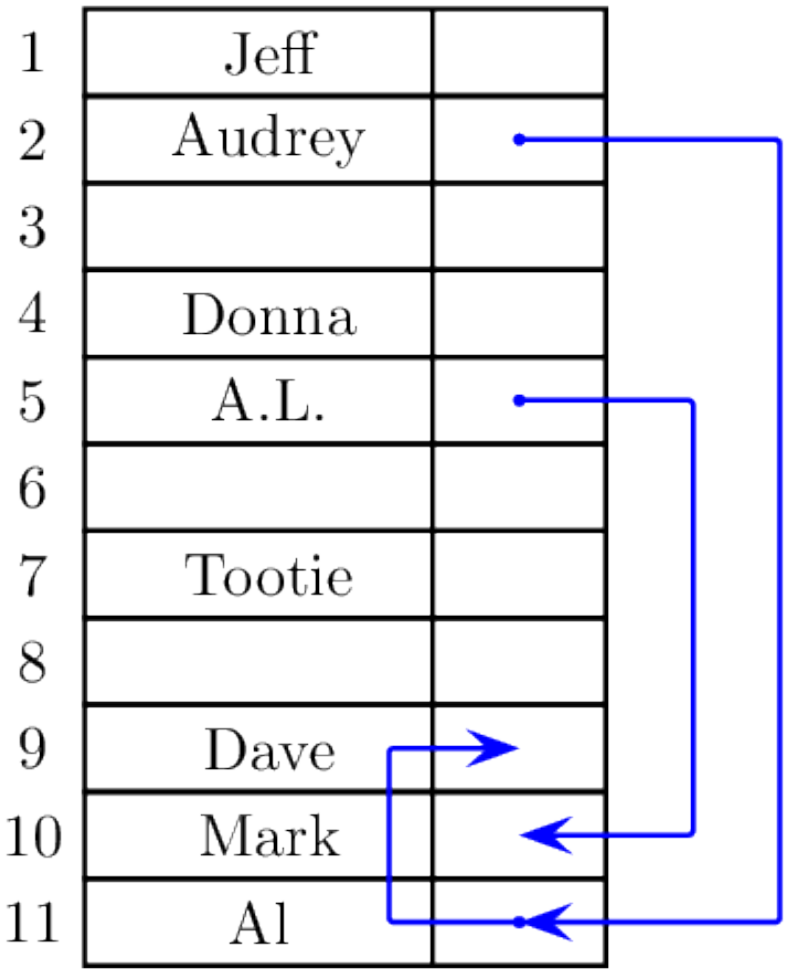
Mecanisme de tratare pentru coliziuni

Lanțuri separate



Mecanisme de tratare pentru coliziuni

Căutarea poziției



Rezolvarea coliziunilor prin înlănțuire

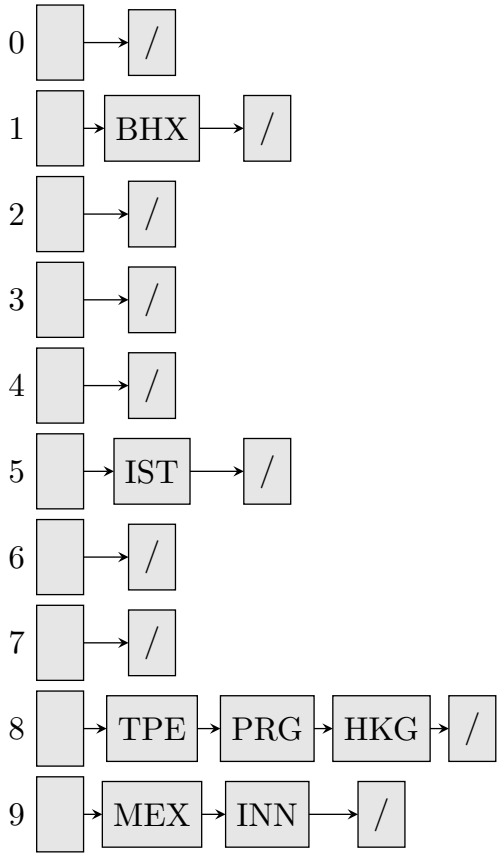
- Pentru această metodă, punem toate elementele ce se dispersează în aceeași locație, într-o listă înlănțuită.
- Pentru fiecare locație din vector reținem un pointer către capul listei tuturor elementelor care se dispersează în locație.
- Dacă nu există elemente pentru o locație, aceasta va conține `NULL`.

Exemplu

- Pornim de la o variantă simplificată în care cheia coincide cu valoarea.
- Vrem să folosim chei care sunt șiruri de caractere ce conțin literele mari din alfabetul englez.
- Aceste șiruri le tratăm ca pe niște numere în baza 26 ($A = 0, B = 1, \ldots Z = 25$)
- Funcția **hash** calculează indexul pe baza numărului obținut în baza 26.

$$hash(BHX) = (1 * 26^2 + 7 * 26 + 23) \mod 10 = 1$$

Rezolvarea coliziunilor prin înlănțuire



Tabele de dispersie ce folosesc înlănțuire

Cum am putea implementa o tabelă de dispersie care utilizează pentru coliziuni **înlănțuirea**?

Tabele de dispersie ce folosesc înlănțuire

Cum am putea implementa o tabelă de dispersie care utilizează pentru coliziuni **înlănțuirea**?

Reprezentarea seamănă destul de mult cu ceea ce am folosit pentru o reprezentare a grafului sub forma listelor de adiacență. Vom avea un vector de liste. În fiecare nod din listă putem reține cheia și valoarea.

Tabele de dispersie ce folosesc înlănțuire

Cum am putea implementa o tabelă de dispersie care utilizează pentru coliziuni **înlănțuirea**?

Reprezentarea seamănă destul de mult cu ceea ce am folosit pentru o reprezentare a grafului sub forma listelor de adiacență. Vom avea un vector de liste. În fiecare nod din listă putem reține cheia și valoarea.

De ce este nevoie să reținem și cheia?

Tabele de dispersie ce folosesc înlănțuire

```
1  typedef struct node {
2      char *key, *value;
3      struct node *next;
4  } Node;
5  typedef struct list {
6      Node *first, *last;
7  } *List;
8  typedef struct hashTable {
9      List* buckets;
10     int size;
11 } *HashTable;
```

Tabele de dispersie ce folosesc înlănțuire

Exemplu de funcție de dispersie

```
12  unsigned int hash(const char* key, int size) {
13      unsigned int hash_value = 0;
14      for (int i = 0; key[i] != '\0'; i++)
15          hash_value = hash_value * 31 + key[i];
16      return hash_value % size;
17  }
```

Tabele de dispersie ce folosesc înlănțuire

Funcții auxiliare pentru listă

```
18  Node *createNode(char *key, char *value) {
19      Node *node = malloc(sizeof(struct node));
20      node->key = strdup(key);
21      node->value = strdup(value);
22      node->next = NULL;
23      return node;
24  }
25  Node *freeNode(Node *node) {
26      if (!node)
27          return NULL;
28      free(node->key);
29      free(node->value);
30      free(node);
31      return NULL;
32  }
```

Tabele de dispersie ce folosesc înlănțuire

```
33  List initList() {
34      List list = malloc(sizeof(struct list));
35      list->first = NULL;
36      list->last = NULL;
37      return list;
38  }
39  List freeList(List list) {
40      if (!list) return NULL;
41      Node *iter = list->first, *temp;
42      while (iter != NULL) {
43          temp = iter;
44          iter = iter->next;
45          temp = freeNode(temp);
46      }
47      free(list);
48      return NULL;
49  }
```

Tabele de dispersie ce folosesc înlănțuire

```
50  List addLast(List list, char *key, char *value) {
51      Node *node = createNode(key, value);
52      if (list == NULL)
53          list = initList();
54      if (list->first == NULL) {
55          list->first = list->last = node;
56          return list;
57      }
58      list->last->next = node;
59      return list;
60  }
```

Tabele de dispersie ce folosesc înlănțuire

Implementarea funcțiilor de bază pentru tabela de dispersie

```
61  HashTable initHashTable(int size) {
62      int i;
63      HashTable hashTable = malloc(sizeof(struct
    ↪  hashTable));
64      hashTable->size = size;
65      hashTable->buckets = malloc(size * sizeof(List));
66      for (i = 0; i < size; i++)
67          hashTable->buckets[i] = initList();
68      return hashTable;
69  }
70  // Varianta în care nu verific dacă mai există cheia
71  void put(HashTable table, char *key, char *value) {
72      unsigned int bucket = hash(key, table->size);
73      table->buckets[bucket] =
    ↪  addLast(table->buckets[bucket], key, value);
74  }
```

Tabele de dispersie ce folosesc înlănțuire

Implementarea funcțiilor de bază pentru tabela de dispersie

```
1  // Varianta în care verific dacă mai există cheia
2  void put(HashTable table, char *key, char *value) {
3      unsigned int bucket = hash(key, table->size);
4      Node *node = table->buckets[bucket]->first;
5      while (node != NULL) {
6          if (!strcmp(node->key, key)) {
7              free(node->value);
8              node->value = strdup(value);
9              return;
10         }
11         node = node->next;
12     }
13     // Nu am găsit cheia
14     table->buckets[bucket] =
    ↪   addLast(table->buckets[bucket], key, value);
15 }
```

Tabele de dispersie ce folosesc înlănțuire

Implementarea funcțiilor de bază pentru tabela de dispersie

```
75  const char* get(HashTable table, const char* key) {
76      unsigned int bucket = hash(key, table->size);
77      Node *current = table->buckets[bucket]->first;
78      while (current != NULL) {
79          if (strcmp(current->key, key) == 0)
80              return current->value;
81          current = current->next;
82      }
83      return NULL;
84  }
```

Tabele de dispersie ce folosesc înlănțuire

Implementarea funcțiilor de bază pentru tabela de dispersie

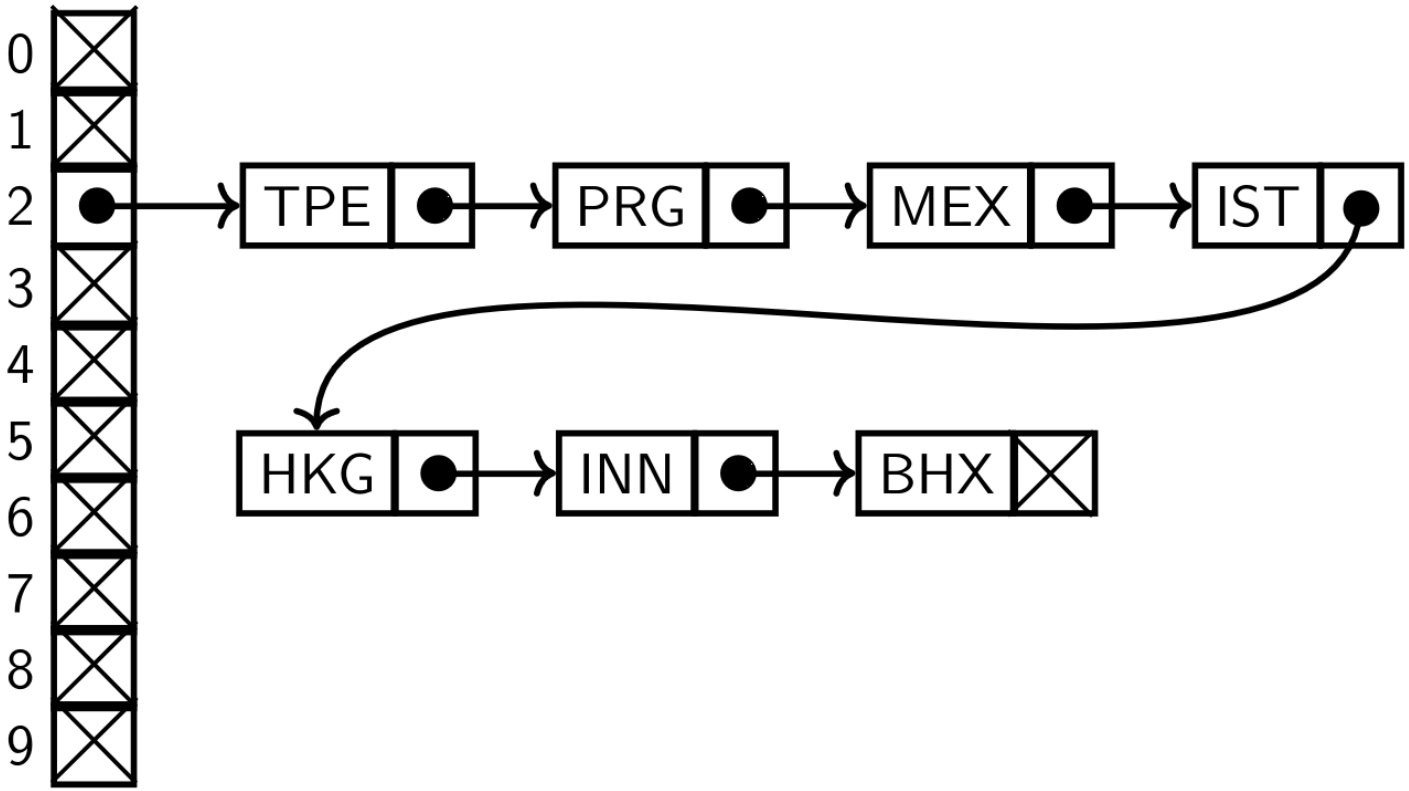
```
85  List freeList(List list) {
86      if (!list)
87          return NULL;
88      Node *iter = list->first, *temp;
89      while (iter != NULL) {
90          temp = iter;
91          iter = iter->next;
92          temp = freeNode(temp);
93      }
94      free(list);
95      return NULL;
96  }
```

Tabele de dispersie ce folosesc înlănțuire

Ce se întâmplă dacă nu alegem corect funcția de dispersie?

Tabele de dispersie ce folosesc înlănțuire

Ce se întâmplă dacă nu alegem corect funcția de dispersie?



Tabele de dispersie ce folosesc înlănțuire

Ce se întâmplă dacă vrem să redimensionăm tabela de dispersie? Spre exemplu, vrem să dublăm nunărul de elemente.

Tabele de dispersie ce folosesc înlănțuire

Ce se întâmplă dacă vrem să redimensionăm tabela de dispersie? Spre exemplu, vrem să dublăm nunărul de elemente.

Va trebui să reconstruim toată tabela de dispersie, deoarece funcția de dispersie este calculată în funcție de dimensiune!

Tabele de dispersie ce folosesc înlănțuire

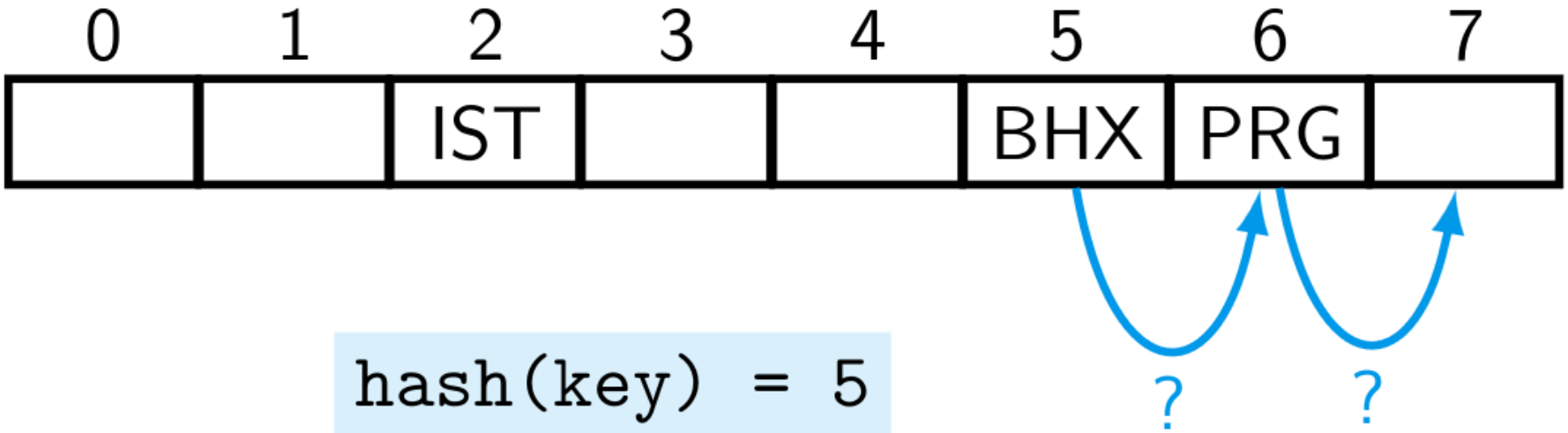
Ce se întâmplă dacă vrem să redimensionăm tabela de dispersie? Spre exemplu, vrem să dublăm nunărul de elemente.

Va trebui să reconstruim toată tabela de dispersie, deoarece funcția de dispersie este calculată în funcție de dimensiune!

Cu alte cuvinte, va trebui să reinserăm toate perechile de forma **key** și **value**.

Rezolvarea coliziunilor prin căutare liniară

- Verificăm dacă este sau nu inserată o valoare la poziția **hash(key)**.
- Dacă avem deja inserată o valoare, atunci verificăm dacă la poziția **[hash(key) + 1] mod Size** avem o valoare.
- Dacă și de această dată găsim o valoare, vom încerca să facem inserarea la poziția **[hash(key) + 2] mod Size**.
- Și tot așa până găsim o poziție liberă.



Rezolvarea coliziunilor prin căutare liniară

De ce nu a mai fost nevoie să reținem și **key** în acest vector?

Rezolvarea coliziunilor prin căutare liniară

De ce nu a mai fost nevoie să reținem și **key** în acest vector?

Ar putea să apară probleme atunci când căutăm valoarea pentru o cheie dacă inserarea nu a fost realizată fix la indexul furnizat de **hash(cheie)**.

Rezolvarea coliziunilor prin căutare liniară

De ce nu a mai fost nevoie să reținem și **key** în acest vector?

Ar putea să apară probleme atunci când căutăm valoarea pentru o cheie dacă inserarea nu a fost realizată fix la indexul furnizat de **hash(cheie)**.

Va trebui să reținem și de această dată cheia pe lângă valoare.

Rezolvarea coliziunilor prin căutare liniară

Cum putem realiza operația de ștergere a unei perechi de forma (**cheie**, **valoare**)?

Rezolvarea coliziunilor prin căutare liniară

Cum putem realiza operația de ștergere a unei perechi de forma (**cheie**, **valoare**)?

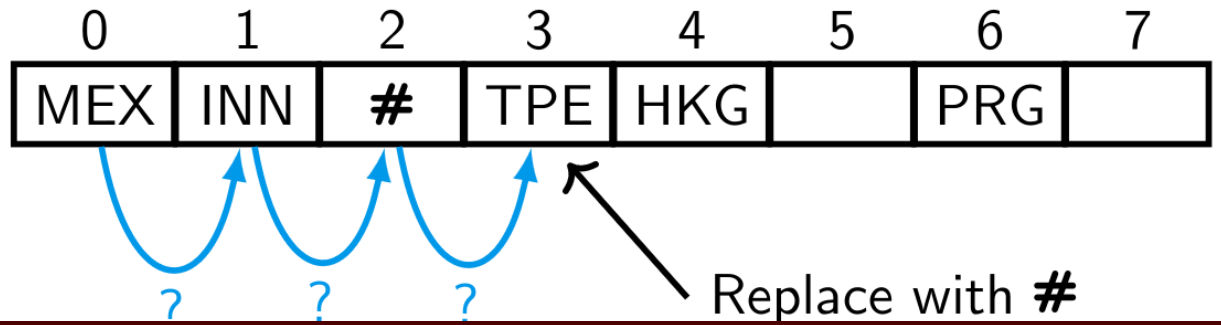
Putem să suprascriem valoarea de la indexul respectiv cu o valoare aleasă astfel încât să fie diferită de valorile pe care le inserăm.

Rezolvarea coliziunilor prin căutare liniară

Cum putem realiza operația de ștergere a unei perechi de forma (**cheie**, **valoare**)?

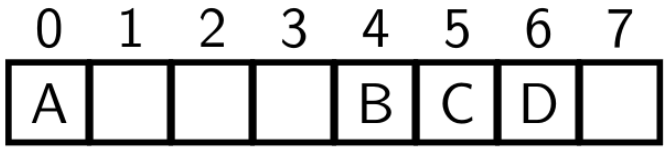
Putem să suprascriem valoarea de la indexul respectiv cu o valoare aleasă astfel încât să fie diferită de valorile pe care le inserăm.

Deleting `key = TPE` such that `hash(key) = 0`:

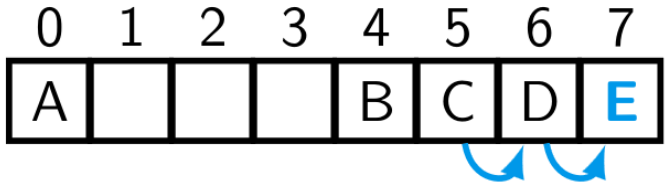


Rezolvarea coliziunilor prin căutare liniară

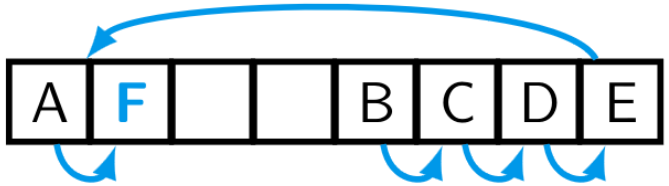
key	A	B	C	D	E	F
hash	0	4	5	6	5	4



1. `insert(E)`

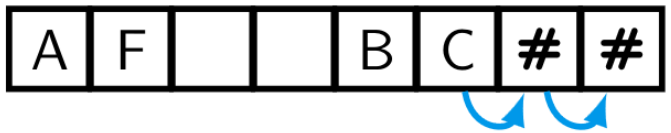


2. `insert(F)`

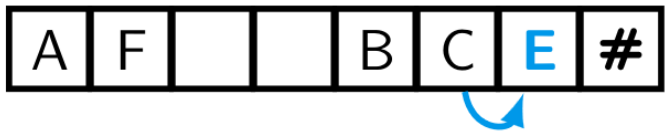


3. `delete(D)`

4. `delete(E)`



5. `insert(E)`



Rezolvarea coliziunilor prin căutare liniară

Ce probleme ar putea să apară de data aceasta dacă nu alegem corespunzător funcția de dispersie?

Rezolvarea coliziunilor prin căutare liniară

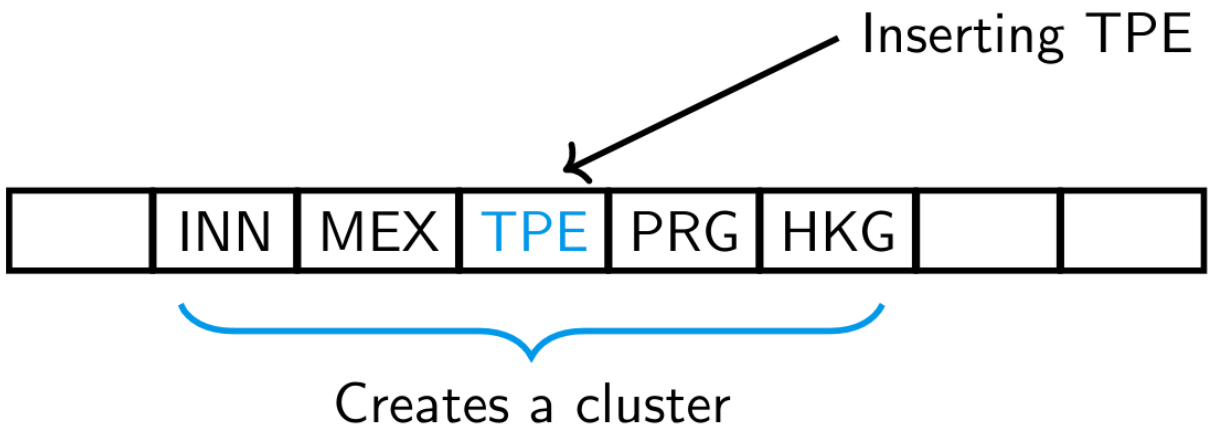
Ce probleme ar putea să apară de data aceasta dacă nu alegem corespunzător funcția de dispersie?

Ar putea să apară doar anumite clustere ocupate în vector.

Rezolvarea coliziunilor prin căutare liniară

Ce probleme ar putea să apară de data aceasta dacă nu alegem corespunzător funcția de dispersie?

Ar putea să apară doar anumite clustere ocupate în vector.



Rezolvarea coliziunilor prin căutare liniară

Ce facem în această situație dacă rămânem fără spațiu?

Rezolvarea coliziunilor prin căutare liniară

Ce facem în această situație dacă rămânem fără spațiu?

O să fie nevoie să realocăm memorie pentru vector și să reinserăm toate perechile care existau înainte!

Rezolvarea coliziunilor prin căutare liniară

Ce facem în această situație dacă rămânem fără spațiu?

O să fie nevoie să realocăm memorie pentru vector și să reinserăm toate perechile care existau înainte!

Această operație este foarte costisitoare și trebuie să avem grijă cum facem redimensionare pentru a nu fie nevoie de foarte multe ori de acest lucru!

Rezolvarea coliziunilor prin dispersie dublă

- Dispersia dublă este una dintre cele mai bune metode disponibile pentru tratarea coliziunilor.
- Acest avantaj apare, deoarece permutările produse au multe dintre caracteristicile permutărilor alese aleator.
- Dispersia dublă folosește o funcție de dispersie de forma:

$$hash(key, i) = [hash_1(key) + i \cdot hash_2(key)] \mod Size$$

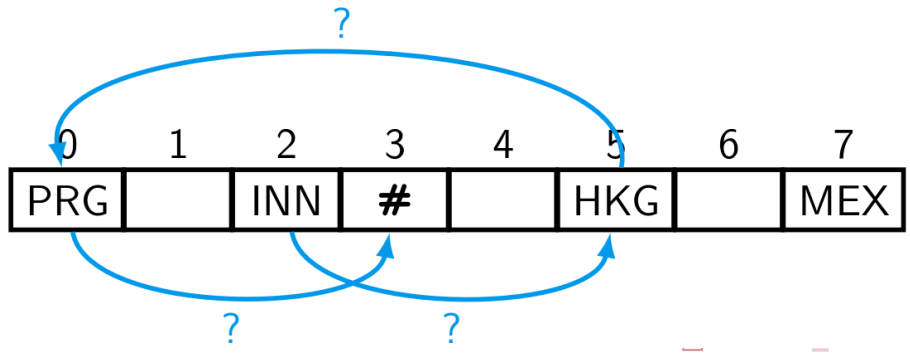
1. `hash1(key) + 1*hash2(key) mod T`
2. `hash1(key) + 2*hash2(key) mod T`
3. `hash1(key) + 3*hash2(key) mod T`
4. ...

`T` is the table size

(until we find an available space)

Example

If `key = TPE`, `hash1(key) = 2`, `hash2(key) = 3`:



Vă mulțumesc pentru atenție!