LAB 5: Tabella di Hash con liste di collisione

```
struct dict::cell {
    Elem elem;
    cell* next;
};
```

// H1

```
// Funzione di hash che considera unicamente il valore ascii del primo carattere della chiave (se esiste) e restituisce il resto della divisione di tale valore per tableDim
int h1(Key s) {
   removeBlanksAndLower(s);        // Funzione ausiliaria che rimuove gli spazi e maiuscole (questo garantisce che la chiave sia in un formato standard e coerente prima di procedere con l'hashing)
   if (s.empty()) {
      return 0;    // Se la chiave risulta vuota dopo la pulizia, la funzione restituisce 0
   }
   char first = s[0];           // Il primo carattere della chiave s viene memorizzato nella variabile first
   int num = first - 'a';       // Converte char in int (0 = 'a' - 25 = 'z')
   return num % tableDim;       // Usa il modulo % per calcolare l'hash
}
```

// H2

```
// Funzione di hash che somma il codice ascii di ogni carattere nella chiave e restituisce il resto della divisione di tale somma per tableDim
int h2(Key s) {
   removeBlanksAndLower(s);     // Funzione ausiliaria che rimuove gli spazi e maiuscole
   int sum = 0;                 // Variabile sum che verrà utilizzata per accumulare la somma dei valori ascii dei caratteri nella chiave.
   for (int i = 0; i < s.size(); ++i) {
      sum = sum + s[i];         // Il valore ascii del carattere corrente viene aggiunto a sum
   }
   return sum % tableDim;       // Usa il modulo % per calcolare l'hash
}
```

// H

```
int h(Key s) {
   return h1(s);           // Modificare questa chiamata per sperimentare l'utilizzo delle funzioni di hash h1, h2, ecc. , definite prima
}
```

// deleteElem

```
Error dict::deleteElem(const Key k, Dictionary &s) {
   if (search(k, s) != emptyValue) {       // se la chiave k è presente nel dizionario s
      cell *cur = s[h(k)];                 // cur -> puntatore alla cella corrente della lista
                                              concatenata che esploreremo per trovare ed
                                              eventualmente eliminare l'elemento.
      cell *prev = nullptr;
      while (cur != nullptr) {       // ciclo che scorre la lista concatenata associata all'indice
                                        h(k) finché ci sono nodi
         if (cur -> elem.key == k) {       // se cur è l'elemento (quindi trovo l'elemento)
            if (prev == nullptr) {                 // se l'elemento è in testa
               s[h(k)] = cur -> next;     // aggiorno la testa della lista all'elemento successivo
            } else {                               // altri casi (non in testa)
               prev -> next = cur -> next;// salta il nodo corrente (cur) aggiornando il
                                             collegamento del nodo precedente (prev) per
                                             puntare al nodo successivo di cur
            }
            delete cur;       // elimino cur
            return OK;        // restituisco OK per indicare che l'eliminazione è avvenuta con successo
         }
         prev = cur;           // passo all'elemento successivo
         cur = cur -> next;
      }
   }
   return FAIL;    // ritorno fail se la chiave non è presente nel dizionario s
}
```

// search

```
Value dict::search(const Key k, const Dictionary &s) {
   cell *cur = s[h(k)];              // h(k) trova l'indice della chiave, cur punta al primo nodo
                                        della lista concatenata
   while (cur != nullptr) {          // ciclo che scorre la lista concatenata associata all'indice
                                        h(k) finché ci sono nodi
      if (cur -> elem.key == k) {    // se cur è l'elemento (quindi trovo l'elemento
         return cur -> elem.value;   // se la chiave k è trovata, ritorno il valore
      }
      cur = cur -> next;             // se il nodo corrente non contiene la chiave cercata, aggiorna
                                        cur per passare al nodo successivo nella lista concatenata.
   }
   return emptyValue;                // se il ciclo termina senza trovare la chiave, restituisce null
}
```

// insertElem

```
Error dict::insertElem(const Key k, const Value v,  Dictionary &s) {
   if (search(k, s) != emptyValue) {// se la chiave è già presente
      return FAIL;                  // restituisce FAIL e termina la funzione, per evitare duplicati
   }
   cell *aux = new cell;        // nuova cella aux, ossia il nodo da inserire nella lista concatenata.
   aux -> elem.key = k;
   aux -> elem.value = v;
   aux -> next = s[h(k)];       // collega la nuova cella aux alla testa della lista concatenata nel
                                   bucket corrispondente. Il nuovo nodo punta al vecchio nodo in
                                   testa, diventando la nuova testa della lista concatenata.
   s[h(k)] = aux;               // Aggiorna la testa della lista concatenata nel bucket h(k) per
                                   puntare al nuovo nodo aux
   return OK;       // restituisce OK per indicare che l'inserimento è avvenuto con successo
}
```