CS 261 – Data Structures

Hash Tables
Buckets/Chaining
Resolving Collisions: Chaining / Buckets

Maintain a linked list (or other collection type data structure, such as an AVL tree) at each table entry

<table>
<thead>
<tr>
<th></th>
<th>Angie</th>
<th>Robert</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Linda</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Joe</td>
<td>Max</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>John</td>
</tr>
<tr>
<td>4</td>
<td>Abigail</td>
<td>Mark</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Joy</td>
</tr>
</tbody>
</table>
Hash Table: Interface

• `initHashTable`

• `addHashTable`

• `containsHashTable`

• `removeHashTable`
Hash Table: Implementation

struct HashTable {

    struct Link **table; /* Hash table → Array of Lists. */

    int count; /* counts the total number of elements in the hash table */

    int tablesizze; /* the number of lists */

};
Hash Table: Implementation

```c
struct Link {
    struct DataElem elem;
    struct Link * next;
};

struct DataElem {
    TYPE key;
    double value;
};
```
Initialization

```c
void initHashTable(struct HashTable *ht, int size){
    int index;

    if(ht == NULL) return;

    ...
}
```
void initHashTable(struct HashTable *ht, int size){
    int index;
    if(ht == NULL) return;
    ht->table = (struct Link **) malloc(sizeof(struct Link *) * size);
    assert(ht->table != 0);
    assert(ht->table != 0);
    ...
}
Initialization

void initHashTable(struct HashTable *ht, int size){

    ... /* (continued) */

    ht->tablesize = size;

    ht->count = 0;

    for(index = 0; index < tablesize; index++)
        ht->table[index] = 0; /* initList() */
```c
void addHashTable (struct HashTable * ht,
                 struct DataElem elem)
{
    /* compute hash index to find the correct bucket */
    int hash = HASH(elem.key);
    int hashIndex = (int) (labs(hash) % ht->tablesize);
    ...

    hashIndex = 4

    returns long absolute integer
```
Add

```c
void addHashTable (struct HashTable * ht,
                  struct DataElem elem)
{
    ... /* (continued) */

    struct Link * newLink =
        (struct Link *) malloc(sizeof(struct Link));
    assert(newLink);
    newLink->elem = elem;
    ...
}
```

```
hashIndex = 4
newLink elem

<table>
<thead>
<tr>
<th></th>
<th>Angie</th>
<th>Robert</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Linda</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Joe</td>
<td>Max</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Abigail</td>
<td>Mark</td>
</tr>
</tbody>
</table>
```
Add

```c
void addHashTable (struct HashTable * ht,
                   struct DataElem elem)
{
    ... /* (continued) */

    newLink->next = ht->table[hashIndex];  /* add to bucket */
    ht->table[hashIndex] = newLink;
    ht->count++;

    ...
}
```

```plaintext
hashIndex = 4
newLink elem
next
```

```plaintext
0  Angie  Robert
1  Linda
2  Joe  Max  John
3  Abigail
4  Mark
```
Add

```c
void addHashTable (struct HashTable * ht,
        struct DataElem elem)
{  ... /* (continued) */
    newLink->next = ht->table[hashIndex];  /*add to bucket*/
    ht->table[hashIndex] = newLink;
    ht->count++;
    ... }```

hashIndex = 4
Add

void addHashTable (struct HashTable * ht,
                    struct DataElem elem)
{
    ... /* (continued) */

    /* resize if necessary */
    float loadFactor = ht->count / ht->tableSize;
    if (loadFactor > MAX_LOAD_FACTOR) /* e.g., 8.0 */
        _resizeTable(ht);
}

void _resizeTable(struct HashTable *ht) {
    int oldsize = ht->tablesize;
    struct HashTable *oldht = ht;
    struct Link *cur, *last;
    int i;

    initHashTable(ht, 2*oldsize);  //New memory location*/

    ...
}

}
void _resizeTable(struct HashTable *ht) {
    /* (continued) */
    for (i = 0; i < oldsize; i++) {
        cur = oldht->table[i];
        while (cur != 0) {
            ...
        }
    }
}
_resizeTable

void _resizeTable(struct HashTable *ht) {
    ...
    /* (continued) */
    for( i = 0; i < oldsize; i++) {
        cur = oldht->table[i];
        while(cur != 0){
            addHashTable(ht, cur->elem);
            last = cur;
            cur = cur->next;
            free(last);
        }
    }
    free(oldht); /* Free up the table */
}
Contains

```c
int containsHashTable(struct HashTable *ht,
                     struct DataElem elem)
{
    int hash = HASH(elem.key);
    int hashIndex = (int) (labs(hash) % ht->tablesize);
    struct Link *cur;

    cur = ht->table[hashIndex]; /*go to the right bucket*/
    ...

    Where to look for the element?
}
```
Contains

```c
int containsHashTable(struct HashTable *ht, 
                      struct DataElem elem)
{
    ...
    cur = ht->table[hashIndex]; hashIndex = 2

    while(cur != 0){
        if(cur->elem.value == elem.value) return 1;
        cur = cur->next;
    }

    return 0;
}
```
Remove

```c
void removeHashTable(struct HashTable *ht,
                     struct DataElem elem)
{
    int hash = HASH(elem.key);
    int hashIndex = (int) (labs(hash) % ht->tablesize);
    struct Link *cur, *last;
    ...
```

Where to look for the element?
void removeHashTable(struct HashTable *ht, 
        struct DataElem elem)
{
    .../* continued */
    cur = ht->table[hashIndex]; /* for iteration */
    last = ht->table[hashIndex]; /* helps remove */
    while(cur != 0){
        if(cur->elem.value == elem.value){
            /* REMOVE */
        }
    }
    else {
        last = cur; /* stores the previous link */
        cur = cur->next; /* moves to the next link */
    }
} ...
Hash Table Implementation: Remove

```c
void removeHashTable(struct HashTable *ht,
                     struct DataElem elem)
{
    /* continued */
    if(cur->elem.value == elem.value){
        /* handle the special case !! */
        if(cur == ht->table[hashIndex])
            ht->table[hashIndex] = cur->next;
        else
            last->next = cur->next;
        free(cur);
        cur = 0; /* jump out of loop */
        ht->count--;
    }
    else { ...
```
Hash Tables: Algorithmic Complexity

• Assumptions:
  – Time to compute hash function is constant
  – Chaining uses a linked list
  – Worst case $\rightarrow$ All keys hash to the same position
  – Best case $\rightarrow$ Hash function uniformly distributes the values (all buckets have the same number of objects in them)
Hash Tables: Algorithmic Complexity

• Contains operation:
  – Worst case for open addressing $\rightarrow O(n)$
  – Worst case for chaining $\rightarrow O(n)$
  – Best case for open addressing $\rightarrow O(1)$
  – Best case for chaining $\rightarrow O(1)$
Hash Table Size

• Load factor:

\[ \lambda = \frac{n}{m} \]

– For chaining, load factor can be greater than 1

• Want the load factor to remain small

• Same as open table hashing: if load factor becomes larger than some fixed limit (say, 8) \( \rightarrow \) double the table size
Hash Tables: Average Case

• Assume hash function distributes elements uniformly

• Average complexity for all operations: $O(\lambda)$

• Want to keep the load factor relatively small

• Resize table
  – Only improves things IF hash function distributes values uniformly
When should you use hash tables?

• Data values must have good hash functions

• Need a guarantee that elements are uniformly distributed

• Otherwise, a Skip List or AVL tree is often faster
Your Turn

• Worksheet 38: Hash Tables using Buckets
  – Use linked list for buckets
  – Keep track of number of elements
  – Resize table if load factor is bigger than 8

• Questions??