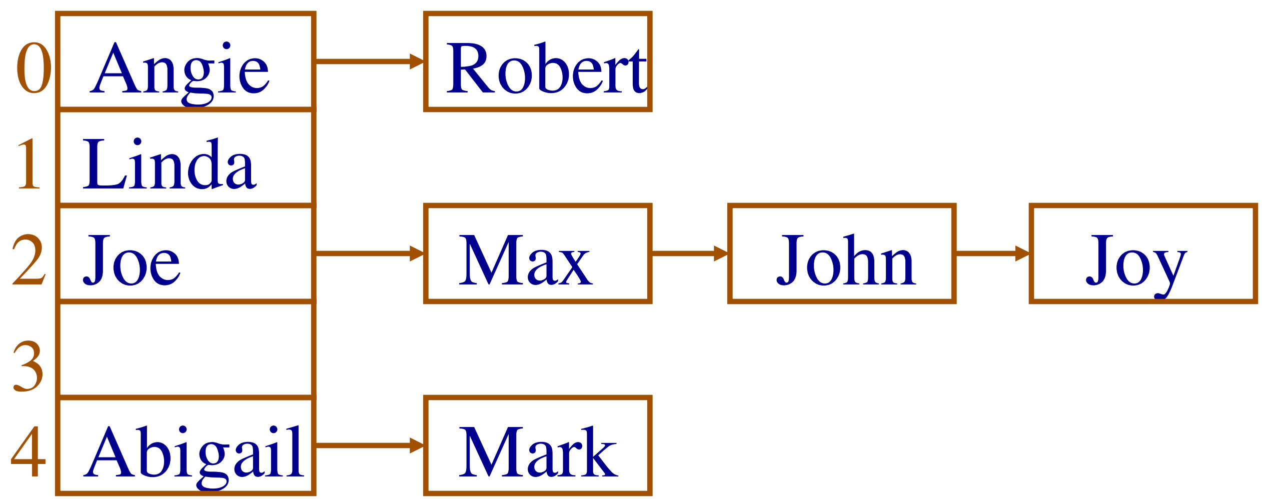
CS 261 – Data Structures

Hash Tables

Buckets/Chaining

Resolving Collisions: Chaining / Buckets

A linked list or other ADT (e.g., AVL tree) at each element of the hash table



Hash Tables: Algorithmic Complexity

- Assumptions:
  - Time to compute hash function is constant
  - Chaining uses a linked list
  - Worst case → All keys hash to the same position
  - Best case → 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 → O(*n*)
  - –Worst case for chaining → O(n)
  - –Best case for open addressing → O(1)
  - –Best case for chaining → O(1)

Hash Table Size

- Load factor:

$$\lambda = n\ /\ m$$

Load factor · # of elements · Size of table

  - For chaining, load factor can be greater than 1

- Want the load factor to **remain small**

- If load factor becomes larger than some threshold → double the table size

Hash Tables: Average Case

- Assume hash function distributes elements uniformly
- Average complexity for remove, contains: $O(\lambda)$
- Want to keep the load factor relatively small
- Resize table
  - –Only improves things *IF* hash function distributes values uniformly

Hash Table: Interface

- **`initHashTable`**
- **`addHashTable`**
- **`containsHashTable`**
- **`removeHashTable`**

Hash Table: Implementation

```
struct HashTable {
  struct Link **table; /* Array of Lists */
  int count; /*number of elements in table*/
  int tablesize; /* the number of lists */
};
```

Hash Table: Implementation

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

struct DataElem {
  TYPE_KEY    key;
  TYPE_VALUE value;
}
```

Initialization

```
void initHashTable(struct HashTable *ht, int size)
{
    int index;
                                    Array of lists
    assert(ht);
                                         Pointer to a list
    ht->table = (struct Link **)
            malloc(sizeof(struct Link *) * size);
    assert(ht->table != 0);


    ...
```

Initialization

```
void initHashTable(struct HashTable *ht, int size)
{
     ...
     ht->tablesize = size;
     ht->count = 0;
     for(index = 0; index < tablesize; index++)
         ht->table[index] = 0; /* initList() */
}
```

NULL pointer

Add

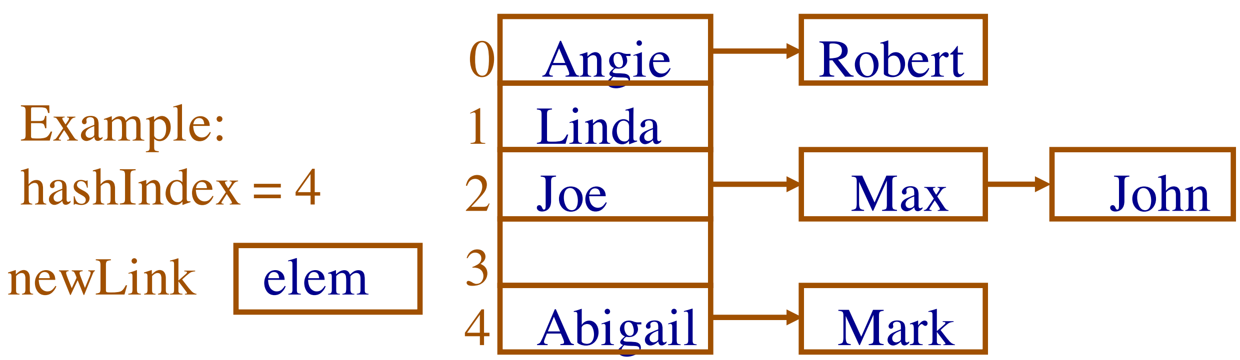
```
void addHashTable (struct HashTable * ht,
                              struct DataElem elem) {
   /* compute hash index to find the bucket */
   int hash = HASH(elem.key);
   int hashIndex =
        (int) (labs(hash) % ht->tablesize);

   ...
```

returns long absolute integer

Example:
hashIndex = 4

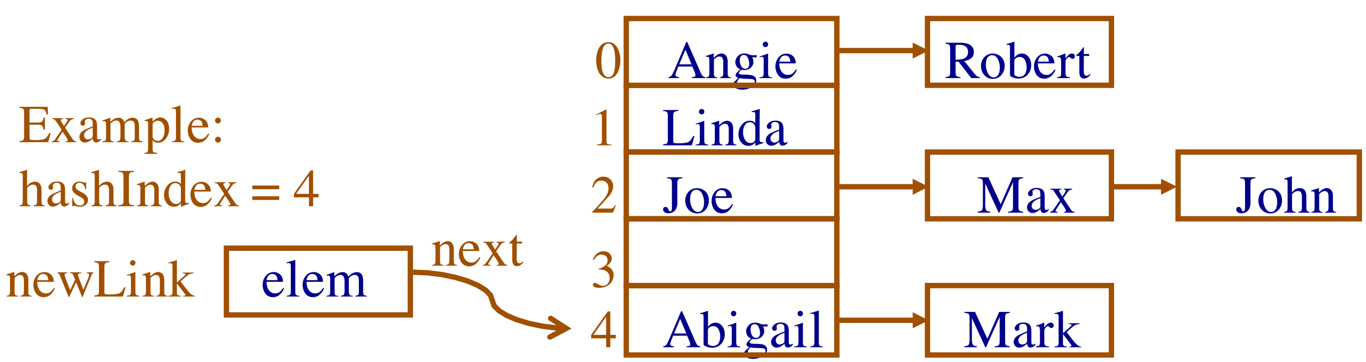
Add

```
void addHashTable (struct HashTable * ht,
                                    struct DataElem elem){
    ...
    struct Link * newLink =
        (struct Link *) malloc(sizeof(struct Link));
    assert(newLink);
    newLink->elem = elem;
    ...
```



Add

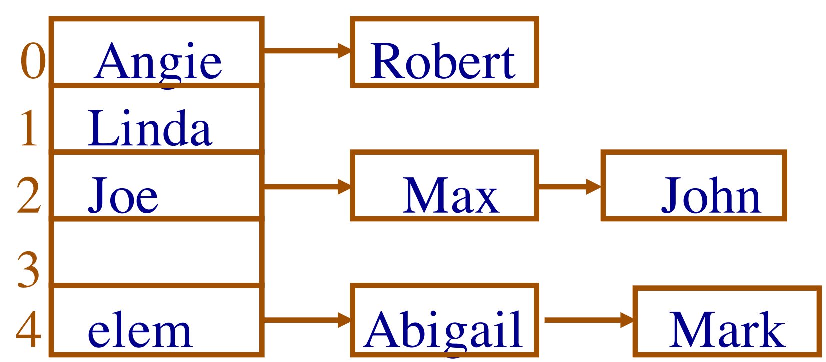
```
void addHashTable (struct HashTable * ht,
                   struct DataElem elem){
   ...
   /* add to bucket */
   newLink->next = ht->table[hashIndex];
   ht->table[hashIndex] = newLink;
   ht->count++;
   ...
}
```



Add

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

Example:
hashIndex = 4



Add

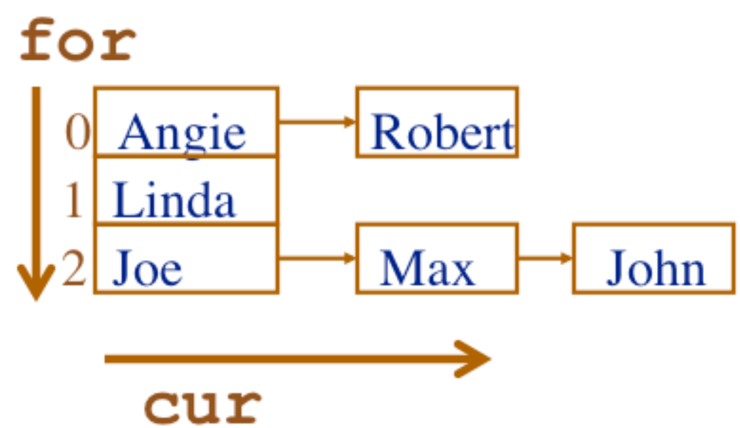
```
void addHashTable (struct HashTable * ht,
                         struct DataElem elem){
   ...
   /* resize if necessary */
   float loadFactor = ht->count/ ht->tableSize;
   if ( loadFactor > MAX_LOAD_FACTOR )
        _resizeTable(ht);
}
```

_resizeTable

```
void _resizeTable(struct HashTable *ht) {
   int oldsize = ht->tablesize;
   struct HashTable *oldht = ht;
   struct Link *cur, *last;
   int i;
   /* New memory location */
   initHashTable(ht, 2*oldsize);
   ...
```

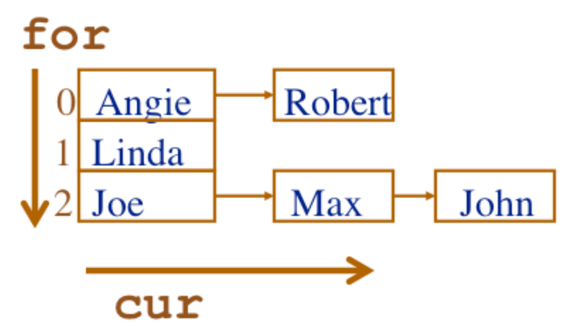
_resizeTable

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



_resizeTable

```
void _resizeTable(struct HashTable *ht) {
   ...
    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 old table */
}
```



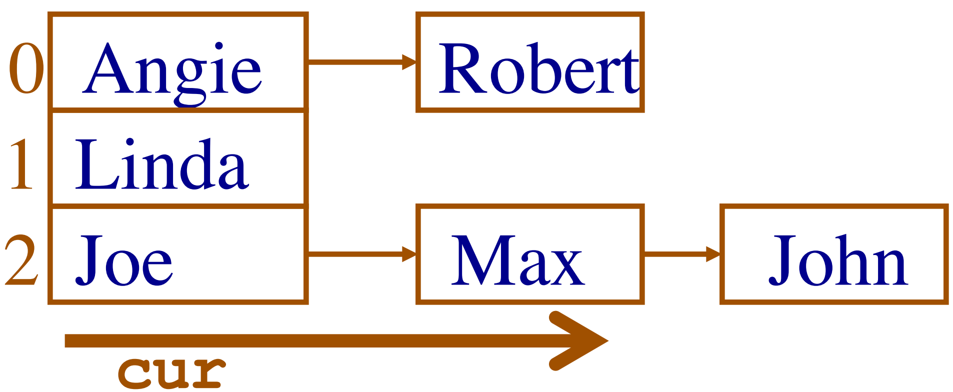
Contains

```
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

```
int containsHashTable(struct HashTable *ht,
                           struct DataElem elem)
{
  ...
  cur= ht->table[hashIndex];
  while(cur != 0){
       if(EQ(cur->elem.value, elem.value)) return 1;
       cur = cur->next;
   }
  return 0;
}
```



Remove

```
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?

Remove

```
void removeHashTable(struct HashTable *ht,
                     struct DataElem elem)
{   ...
    cur = ht->table[hashIndex]; /* for iteration */
    last = ht->table[hashIndex]; /* helps remove */
    while(cur != 0){
        if(EQ(cur->elem.value,elem.value)){
            /* REMOVE */
        }
        else {
            last = cur; /* remembers the previous link */
            cur = cur->next; /* moves to the next link */
        }
    } ...
```

Remove

```
void removeHashTable(struct HashTable *ht,
                              struct DataElem elem)
{  ...
   if(EQ(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, if single remove*/
      ht->count--;
   }
   else { ...
```

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??