Hash Tables

Real-World Associations

Last time - Associative Containers

- Associative Container
  - Create an ‘Association’ between the data and container position
  - Search times improve as we know the ‘general area’
  - Association can be made directly (1-to-1)
    - These are usually smaller cases and it’s not always possible
- Ideally:
  - Create an association between larger and more dynamic data sets
  - Computationally efficient processing of this association
  - Association gives an accurate area with limited extra searching

Last time - Hash Functions

- Hash Functions
  - Calculates a position a starting point for your search
  - KEY is used to generate an index that occurs in the range [0 , TableSize-1]
  - Reduce your Data to a smaller numerical range [0, TableSize-1]

- Good Hashing
  - Computationally Efficient
  - Uses minimal extra requirements (memory or resources)
  - Effectively distributes elements throughout the container

Last Time - Hashing

- Perfect Hash (1-to-1) is the goal
  - 1 key maps to 1 index.
  - There is an index for each potential key
- Realistic Hash (Many keys to 1 index)
  - Try to minimize these “collisions” but they will occur
  - Test our hash function by testing possible keys
    - Generate new hash or combinations of hash to reduce collisions
  - Provide a “collision resolution strategy”
    - What do we do when a collision happens?

Last Time - Hash Functions

- Categories of Hash Functions
  - Truncation (Reducing information by ignoring it)
  - Folding (Reducing information by breaking into smaller pieces)
  - Division (Reducing information by modulus operation %)
- Collision Resolution Strategies
  - Separate Chaining (Storing all values using an ADT (List))
  - OpenAddress (Finding another position in the table)
    - Linear Probing (+1)
    - Random Probing (+N)
    - Quadratic probing (polynomial(i))
    - Double or rehashing (f(n) + g(n))

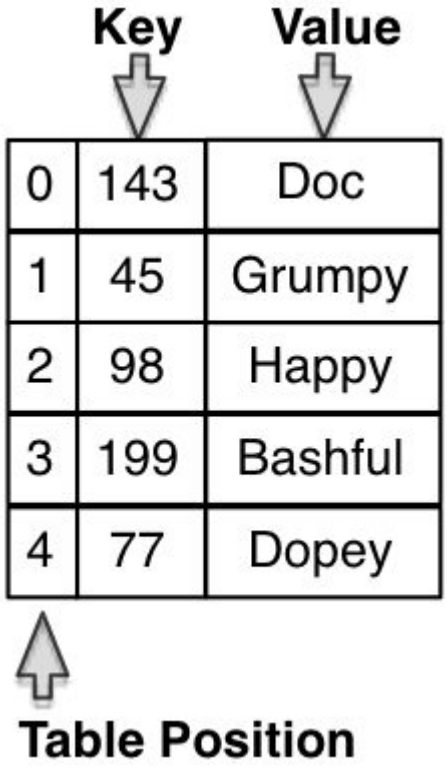
Associative Array

0	Sleepy
1	Doc
2	Grumpy
3	Happy
4	EMPTY
5	Bashful
6	Dopey
7	EMPTY

- **Index = KEY**
- Key is how we access positions
- Random Access -> O(1)

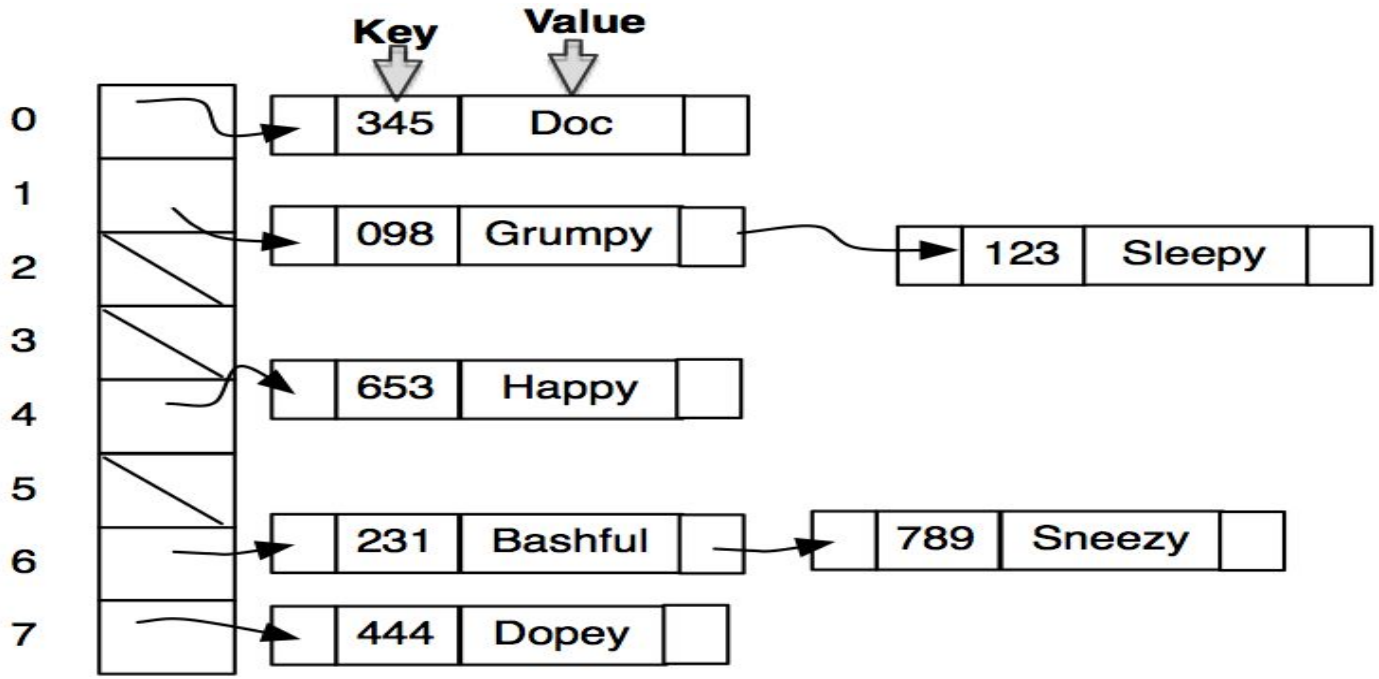
- What do we do about array[4] ?
- Wasteful
- Difficult to fill all positions with ‘real’ data

Hash Table



- **Index = function(KEY)**
- Key is how we calculate index
- Random Access -> O(1)
- Store key in our Node
- Key is used to identify data in the search
- Doesn’t just help find it but identify

Hash Table - Separate Chaining



Visualization - VisuAlgo - Separate Chaining

Hash Table

- Hash Table usually implemented as an Array
  - At least in this course but other implementations exist
- Add the interface to do common operations
- Interface mostly abstracts away the details
  - What the hash function does
  - How the collisions are resolved
  - How the container actually works

Hash Table API Operations

- create()
- destroy()
- insert (HTable, key, value)
- remove(HTable, key)
- lookup(HTable, key)
- update(HTable, key, newValue)

Identical API to our Associative Array

Key is unique -> index likely isn't

Let's practice....

I gss y cn rd ths txt wtht vwls.

Can you read the text?

It was written by applying an algorithm to each word in the sentence and using the result instead of the original word.

What is the algorithm?

Does this algorithm result in completely unique words?

Let's practice....

I gss y cn rd ths txt wtht vwls.

Can you write a function that takes a single word as a parameter and spewed out the de-voweled word?

char * vowelEater (char * word);

Take a minute to jot down the algorithm.

Vowel Eater - Code

```
Char * vowelEater( char * word ) {
    Int length = sizeof(word)/ sizeof(char);
    Int currentId = 0;
    Char * newStr = malloc(sizeof(char)*length);
    for( int i = 0; i < length;i++)
        if( isVowel( word[i] ))
                continue;
        newStr[count] = word[i]
        Count++
newStr[0]%tableSize
```

Back to Node

```
typedef struct Node
{
	int key; ///< integer that represents a piece of data in the table (eg 35->"hello")
	void *data; ///< pointer to generic data that is to be stored in the hash table
	struct Node *next; ///< pointer to the next Node if a collision is detected
} Node;

typedef struct Node
{
	char* key; ///< char* that represents a piece of data in the table (eg “34”->"hello")
	void *data; ///< pointer to generic data that is to be stored in the hash table
	List* collisions; ///< pointer to keys that collide at this index
} Node;
```

Define the Table

```
typedef struct HTable
{
	size_t size; ///< number that represents the size of the hash table
	Node **table; ///< array that contains all of the table nodes
	void (*destroyData)(void *data); ///< delete a single item from the hash table
	int (*hashFunction)(size_t tableSize, int key); ///< hash the data
	void (*printNode)(void *toBePrinted); ///< prints out a data element of the table
}HTable;
```

Where is our comparison function pointer?

Define the Table

```
typedef struct HTable
{
	size_t size; ///< number that represents the size of the hash table
	Node **table; ///< array that contains all of the table nodes
	void (*destroyData)(void *data); ///< delete a single item from the hash table
	int (*hashFunction)(size_t tableSize, int key); ///< hash the data
	void (*printNode)(void *toBePrinted); ///< prints out a data element of the table
}HTable;
```

Size is different than count… you can add count if you want

The dreaded double pointer!

- Usually not allowed in class but likely need them Hash Table
- If you hate the syntax use typedefs
- It’s useful in many situations just usually poorly handled
  - Dynamic Arrays
  - Command line arguments

    char ** argv or char * argv[]

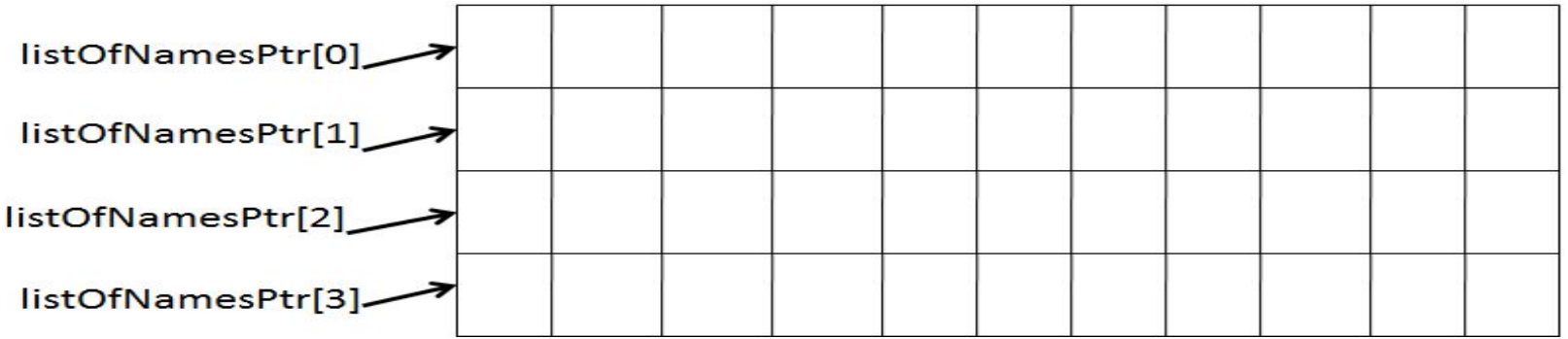
- Could be considered just a Grid
  - Where the rows don’t have to have the same length (Jagged)

DataType ** variableName

Array holding list of names names

char ** listOfNamesPtr = malloc(sizeof(char*) * 4)

each row: malloc (sizeof(char)*11)



Create our Table Let's build this....

```
HashTable * createTable(int tablesize, void (*destroyData)(void *data))
{
  HashTable * tmpPtr = malloc(sizeof(HashTable));
   if( tmpPtr != NULL ) {

        /* set the initial value of the struct members */
         tmpPtr->size = tablesize;
        tmpPtr->tableSpace= malloc(sizeof(Node*)*tmpPtr->size));

       tmpPtr->destroyData = destroyData;
       Return tempPtr;
}
```

insertData(*hashTable, key, data)

1. Index = hashFunction(tableSize,key)
2. If Tablex[index] = Empty
    a. Table[index] = (Key,data)
3. Else
    a. newIndex = hasFunction2(Key)
    b. Table[index] = (Key,Data)

What are some of the steps?

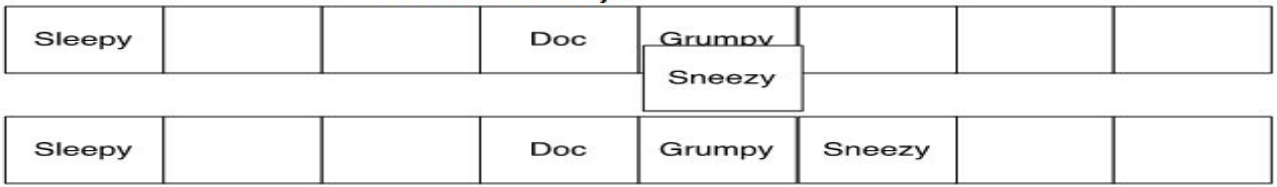
Collision Strategy - Linear Probing

H(Sleepy)=0
H(Doc) = 3
H(Grumpy) = 3
H(Sneezy) = 4
H(Happy) = 5
H(Bashful) = 7
H(Dopey) = 7

Linear Probing

Put Grumpy in the next available space

Add Sneezy



Add Happy

Add Bashful

Add Dopey

Sleepy			Doc	Grumpy	Sneezy	Happy	
Sleepy			Doc	Grumpy	Sneezy	Happy	Bashful
Sleepy	Dopey		Doc	Grumpy	Sneezy	Happy	Bashful

How many probes to find Dopey

H(Sleepy)=0
H(Doc) = 3
H(Grumpy) = 3
H(Sneezy) = 4
H(Happy) = 5
H(Bashful) = 7
H(Dopey) = 7

Linear Probing

How many probes to find Dopey?

A. 2
B. 5
C. 3
D. 1
E. 7

Remove Data

- Separate Chaining it was easy to delete from the list
- Open Addressing was harder
  - if a collision occurred during insert, searched for new position
  - Inserted at new position based on a 2nd hash function
  - What happens to item in the 2nd position if item in 1st is removed

Removing data

H1(Sleepy)=0
H1(Doc) = 3
H1(Grumpy) = 3
H1(Sneezy) = 4
H1(Happy) = 5
H1(Bashful) = 7
H1(Dopey) = 7

Think about Delete

Lets delete Doc

Now lets find Grumpy

Grumpy is in a different table space because of a collision, but it looks like Grumpy is not in the table

Lazy Delete

- When you use open addressing, you must implement a lazy delete strategy.
- When something is deleted, the space is not marked ‘empty’ it is marked as ‘deleted’ so that searches know to keep going.
- Treated as an empty space when doing insertions

Sleepy	Grumpy	Dopey	#	Sneezy	Happy		Bashful

Now lets find Grumpy

LookupData

- Key is used to generate a lookup positions
- Keep searching until we find it or an empty space.
- Should follow the same path as insert
- We have to compare the keys
- To make sure we found our data
- This is why keys are unique and stored

Reminder - Hashing Truncation

- Truncating or remove a portion of the key
- Examples
  - hash(“Dopey”) -> “D”
  - hash(100023) -> 100
- Reducing the information provided to a smaller range

Adv: Quick to compute

Dis: Not that good at reducing collisions, unless portion of key is significant

ex) hash(“University of Guelph”) -> “Guelph”)

Reminder Hashing Folding

- Folding breaks down the key into multiple pieces usually equal sizes
- Each piece still combines to the resulting index.
- Examples
  - hash(“519-811-9117”) -> 51 + 98 + 11 + 91 + 17 continue until value is less than table size
  -
- Reducing the information but not ignoring that it exists

Adv: Fast and more evenly distributed.

Dis: May still result in uneven distribution

Reminder - Hashing Division

- Integer Division divides the space into containers
- N % tablesize
- Examples
  - hash(5,8) and hash(4,3) -> both return 3
  -
- Helps when table size is a large odd prime large

Adv: Quick and evenly distributes data among the container.

Dis: Keys are not always numeric

Double Hashing

Can avoid primary and secondary clustering by using a second hash function to calculate the constant.

constant = h2(key)

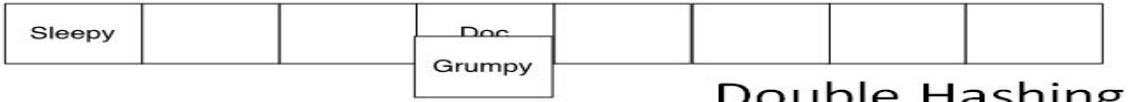
newLocation = (mostRecentLocation + constant) mod tableSize

Two colliding keys will have a different constant, thus will hash to a different second location.

Double Hash

H1(Sleepy)=0
H1(Doc) = 3
H1(Grumpy) = 3
H1(Sneezy) = 4
H1(Happy) = 5
H1(Bashful) = 7
H1(Dopey) = 7

H2(Grumpy) = 6
H2(Sneezy) = 0
H2(Happy) = 3
H2(Bashful) = 6
H2(Dopey) = 4

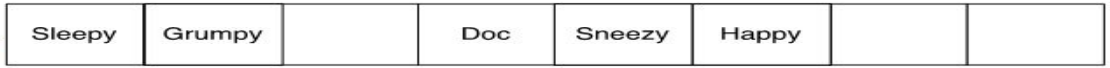


Double Hashing

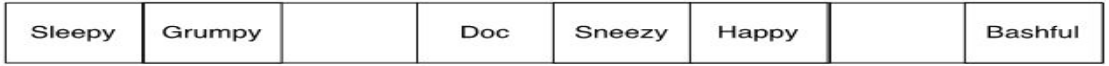
Grumpy: (3 + 6)mod 8 = 1

Add Sneezy (using H1)

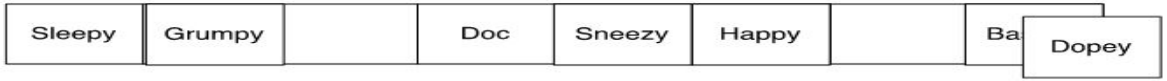
Add Happy



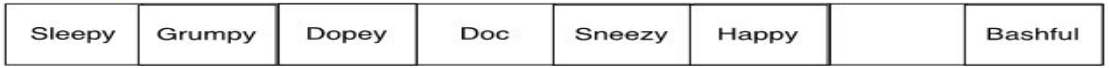
Add Bashful



Add Dopey



Dopey: (7+4)mod 8 = 3



Kahoot