HASH TABLES

OVERVIEW

OVERVIEW

- **The problem of storing and retrieving information has been studied for many years**
  - Given a list of names and phone numbers, how can we quickly find a person's phone number given their name?

- **The classic solution is to use array based binary search:**
  - Store names and phone numbers in an array N long
  - Sort these records by name in $O(N\log_2 N)$ time
  - Use binary search to find data the array in $O(\log_2 N)$ time

OVERVIEW

The preprocessing phase:

Index	Name	Phone
0	Brown, Jim	521-9876
1	Jones, Tom	521-1234
2	Taylor, Brian	521-2121
3	Johnson, John	521-5500
4	Smith, John	521-3456
5	Williams, Anne	521-1020
6	Davis, Tony	521-7384
7	Smith, Lisa	521-2468
8	White, Betty	521-6543

Unsorted array

Sort array

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Johnson, John	521-5500
3	Jones, Tom	521-1234
4	Smith, John	521-3456
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020

Sorted array

OVERVIEW

The search phase:

Binary search for Jones, Tom starts at (0+8)/2=4

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Johnson, John	521-5500
3	Jones, Tom	521-1234
4	Smith, John	521-3456
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020

Sorted array

OVERVIEW

The search phase:

Binary search for Jones, Tom goes to index (0+4)/2=2

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Johnson, John	521-5500
3	Jones, Tom	521-1234
4	Smith, John	521-3456
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020

Sorted array

OVERVIEW

The search phase:

Jones, Tom phone record is found at index (2+4)/2=3

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Johnson, John	521-5500
3	Jones, Tom	521-1234
4	Smith, John	521-3456
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020

Sorted array

OVERVIEW

What happens if we insert data?

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Johnson, John	521-5500
3	Jones, Tom	521-1234
4	Smith, John	521-3456
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020
9	Jackson, Janet	521-1111

Array with new phone number

Resort array

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Jackson, Janet	521-1111
3	Johnson, John	521-5500
4	Jones, Tom	521-1234
5	Smith, John	521-3456
6	Smith, Lisa	521-2468
7	Taylor, Brian	521-2121
8	White, Betty	521-6543
9	Williams, Anne	521-1020

Sorted array

OVERVIEW

What happens if we delete data?

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Jackson, Janet	521-1111
3	Johnson, John	521-5500
4	Jones, Tom	521-1234
~~5~~	~~Smith, John~~	~~521-3456~~
6	Smith, Lisa	521-2468
7	Taylor, Brian	521-2121
8	White, Betty	521-6543
9	Williams, Anne	521-1020

Delete phone number from array

Resort array

Index	Name	Phone
0	Brown, Jim	521-9876
1	Davis, Tony	521-7384
2	Jackson, Janet	521-1111
3	Johnson, John	521-5500
4	Jones, Tom	521-1234
5	Smith, Lisa	521-2468
6	Taylor, Brian	521-2121
7	White, Betty	521-6543
8	Williams, Anne	521-1020

Sorted array

OVERVIEW

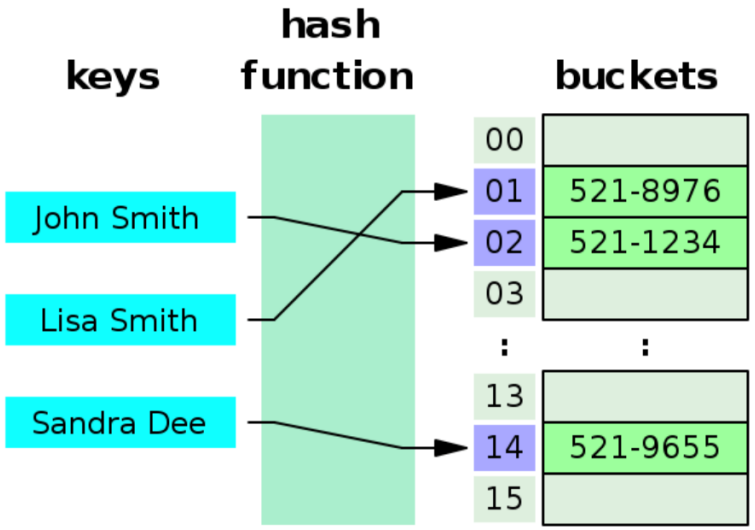
- **How much work does insertion require?**
  - Shift data records down to make room for new data O(N)
  - Put new record into the hash table O(1)

- **How much work does deletion require?**
  - Remove record from the hash table O(1)
  - Shift data records up to fill in empty space O(N)

- **When there are a large number of additions or deletions the classic binary search approach is simply too slow**

OVERVIEW

- **The idea of a hash table is to quickly store and search for data in an array without keeping it in sorted order**
  - To do this, we define a hash function based on the search key to decide where the data should be stored in the array and we jump directly to that location



- For example, we know John Smith belongs in location 02 so that is where we store his phone number, and where we look for his phone number (image from Wikipedia)

OVERVIEW

- **How do hash tables work?**
  - First, we must define a “hash function” on the data to decide where the data should be stored in the array
  - When we insert data, we go to the hash location, and if it is currently empty, we save the record
  - When we search for data, we go to the hash location, and if it is not empty, we retrieve the record

- **Because the hash function uses the content of the record to calculate the hash location, hash tables are often called “content accessible memory” or “associative memory”**

OVERVIEW

Data insertion phase

Assume the value of hash("White, Betty") = 1

Since this is empty we store the phone number

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6		
7		
8		

OVERVIEW

Data insertion phase

Assume the value of hash("Jones, Tom") = 6

Since this is empty we store the phone number

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

OVERVIEW

Data search phase

Assume the value of hash("White, Betty") = 1

This location is not empty we so can retrieve the phone number

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

OVERVIEW

Data search phase

Assume the value of hash("Claus, Santa") = 4

This location is currently empty so no phone number is returned

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

OVERVIEW

- **In the next section we describe how hash functions can be created for different data types**
  - Integers, floats, strings
- **In the following sections, we describe four techniques for implementing hash tables**
  - Linear probing
  - Secondary hashing
  - Hash buckets
  - Separate chaining

HASH TABLES

HASH FUNCTIONS

HASH FUNCTIONS

- **The "magic" of hash tables is in the hash function**
  - Goal is to use the contents of the data to calculate an array index in range [0..N-1]

- **For integer data, the solution is to use modulo operator**
  - Hash(num) = num % N

- **We can spread the data out using more complex formulas**
  - Hash(num) = (a*num + b) % N
  - Hash(num) = (a*num*num + b*num + c) % N

HASH FUNCTIONS

- **For float data, convert to integer and use modulo operator**
  - Hash(num) = int(num * 1000) % N
- **We can also use more creative functions**
  - Hash(num) = int(N * sqrt(num)) % N
- **We need to be careful to avoid negative values**
  - Hash(num) = abs(num * 1000) % N
  - Hash(num) = int(N * sqrt(fabs(num))) % N
  - Hash(num) = int(N * (sin(num) + 1)) % N

HASH FUNCTIONS

- **For string data it is possible to create a wide range of hash functions based on the ASCII codes of letters**

- **One solution is to calculate the sum of the ASCII codes for letters in the word**
  - Hash("cat") = (99 + 97 + 116) % N

Letter order is ignored so Hash("cat") = Hash("act")

- **This can be easily implemented**

```
int hash = 0;
for (int i = 0; i < word.length(); i++)
      hash = hash + word[i];
hash = hash % N
```

HASH FUNCTIONS

- **Or we can calculate the product of the ASCII codes for all letters in the word**
  - Hash(“dog”) = (100 * 111 * 103) % N

Letter order is ignored so Hash(“dog”) = Hash(“god”)

- **We can use any formula we like**
  - Hash(word) = (word[2] * 17 + word[3] * 42) % N
  - Hash(word) = (word[1] * word[4] + word[0]) % N

- **We need to be careful to avoid array index errors when processing short strings with hard coded formulas above**

HASH FUNCTIONS

- **One popular option is to treat letters as digits base 26 and convert string to integer**
  - Hash("cat") = 26^2 * ('c'-'a') + 26 * ('a'-'a') + ('t'-'a')) % N

```
int hash = 0;
for (int i = 0; i < word.length(); i++)
      hash = 26 * hash + tolower(word[i]) – 'a';
hash = hash % N;
```

Gives us a value between 0..25

- **This approach is very effective for using the whole [0..N-1] range so strings are spread out in the hash table**

HASH FUNCTIONS

- **One issue with hash functions is that different data values may produce the same hash value**
  - This causes a “collision” when we try to insert data in a location that is already occupied

- **Assume N = 100, and Hash(num) = num % N**
  - Hash(1024) = 1024 % 100 = 24
  - Hash(8024) = 8024 % 100 = 24
  - Hash(24) = 24 % 100 = 24

- **Hash tables must be implemented to take care of possible collisions**

This is a many-to-one function because different inputs can produce the same output

HASH TABLES

LINEAR PROBING

LINEAR PROBING

- **Linear probing is a popular technique for dealing with collisions when inserting data into a hash table**
  - We calculate the hash value for the input data
  - We look at that location in the hash table
  - If the location is empty, we insert the data
  - If the location is already occupied, we “probe” the next locations until an empty location is found to insert data
  - We use modulo operation on the table index to “wrap around” when we reach bottom of table

- **We will illustrate the insertion process using phone data**

LINEAR PROBING

Data insertion phase

Assume that we have already inserted two phone records into the hash table

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data insertion phase

If hash("Smith, John") = 2 this location is empty, so we can insert this phone record

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data insertion phase

If hash("Brown, Jim") = 1 this location is full, so we look at next location in table

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data insertion phase

This location is also full, so we look at next location in table

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data insertion phase

This location is empty, so we store the phone record here

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

- **How does search work with linear probing?**
  - We calculate the hash value for the data we are looking for
  - We look at that location in the hash table
  - If the location is empty, the data was not found
  - If the location is not empty, we need to look at the record to see if the name field matches
  - If the name matches, we found the desired data
  - If name does not match, we look at next locations until a match is found or an empty location is found

- **We will illustrate the search process using phone data**

LINEAR PROBING

Data search phase

Since hash("Brown, Jim") = 1 we look at this location, but the name does not match so we look at the next location

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

This name at this location does not match so we probe again

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

The name at this location does match so the record was found

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

If hash("Claus, Santa") = 2 we look at this location, but the name does not match so we look at the next location

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

This name at this location does not match so we probe again

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

We come to an empty location so there is no phone record for "Claus, Santa" (too bad)

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

- **How do we delete data when using linear probing?**
  - First we search the hash table to find location of record
  - Next we overwrite the current data at that location with a special “deleted value” like “Zzzz, Zzzz” 999-9999

- **How does this effect insert and search?**
  - We need to adapt the insert algorithm to stop when an empty location or a “deleted value” is found
  - We need to adapt the search algorithm to continue searching if a “deleted value” is found

LINEAR PROBING

Delete phase

If hash("Smith, John") = 2 we look at this location, and we replace with Zzzz data

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

If hash("Claus, Santa") = 2 we look at this location, and insert his phone number here

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Zzzz, Zzzz	999-9999
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

Data search phase

If hash("Claus, Santa") = 2 we look at this location, and insert his phone number here

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Claus, Santa	123-4567
3	Brown, Jim	521-9876
4		
5		
6	Jones, Tom	521-1234
7		
8		

LINEAR PROBING

```
class HashTable
{
 public:
   HashTable(int size);
   HashTable(const HashTable & ht);
   ~HashTable();

   bool Insert(string key, int value);
   bool Search(string key, int &value);
   bool Delete(string key);
   void Print();
```

Constructors and destructor

Hash table API methods for insertion, search and deletion

LINEAR PROBING

```
...
private:
    int Hash(string key);
    int Hash2(int index);


    int Size;
    int *Value;
    string *Key;
};
```

Private methods to calculate the hash value for the key

Dynamic arrays to store the search key and data values

LINEAR PROBING

```
HashTable::HashTable(int size)
{
    Size = size;
    Value = new int[Size];
    Key = new string[Size];

    for (int index = 0; index < Size; index++)
    {
        Value[index] = NONE;
        Key[index] = EMPTY;
    }
}
```

Allocate dynamic arrays for the search key and data values

Initial both arrays to special “empty” values

LINEAR PROBING

```
bool HashTable::Insert(string key, int value)
{
    // Find desired key
    int index = Hash(key);
    while ((Key[index] != key) && (Key[index] != EMPTY))
        index = Hash2(index);

    // Insert value into hash table
    Value[index] = value;
    Key[index] = key;
    return true;
}
```

Loop over hash table to find desired key or an empty location

Store key and value at this hash table location

LINEAR PROBING

```
bool HashTable::Search(string key, int &value)
{
    // Find desired key
    int index = Hash(key);
    while ((Key[index] != key) && (Key[index] != EMPTY))
        index = Hash2(index);

    // Return value from hash table
    if (Key[index] == key)
        value = Value[index];
    return (Key[index] == key);
}
```

Loop over hash table to find desired key or an empty location

If the key is found, save corresponding value in reference parameter

LINEAR PROBING

```
bool HashTable::Delete(string key)
{
    // Find desired key
    int index = Hash(key);
    while ((Key[index] != key) && (Key[index] != EMPTY))
        index = Hash2(index);
    // Delete value from hash table
    if (Key[index] == key)
    {
        Key[index] = DELETED;
        return true;
    }
    return false;
}
```

Loop over hash table to find desired key or an empty location

We store a special "deleted" value in key so search function works

LINEAR PROBING

```
int HashTable::Hash(string key)
{
    int num = 42;
    for (int i = 0; i < int(key.length()); i++)
        num = (num * 17 + key[i]) % Size;
    return num;
}

int HashTable::Hash2(int index)
{
    return (index + 1) % Size;
}
```

We calculate the hash value as weighted sum of letters in string

We go to next hash table location (with wrap around)

LINEAR PROBING

```
HashTable::~HashTable()
{
    if (Value != NULL)
        delete[]Value;
    if (Key != NULL)
        delete[]Key;


    Value = NULL;
    Key = NULL;
    Size = 0;
}
```

First we release the memory for the dynamic arrays

Then we set pointers to NULL (not really needed but good practice)

LINEAR PROBING

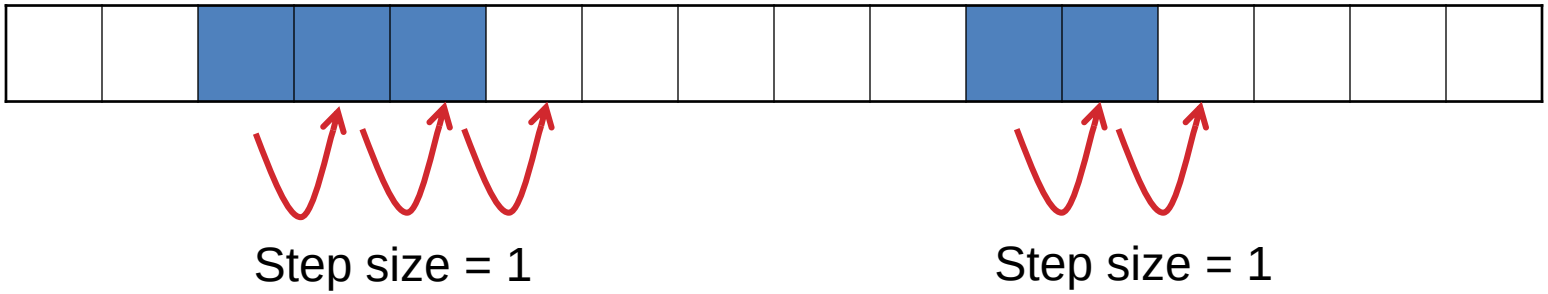
- **The implementation of linear probing is fast and easy**
  - It can be easily extended to different key data types
  - The data field can also be expanded as needed
- **Linear probing works well**
  - When hash table has lots of empty locations (20% full)
  - When the hash function is uniformly distributed
- **Linear probing works poorly**
  - When the hash table becomes nearly full (90% full)
  - When the hash function produces large clusters of values

HASH TABLES

DOUBLE HASHING

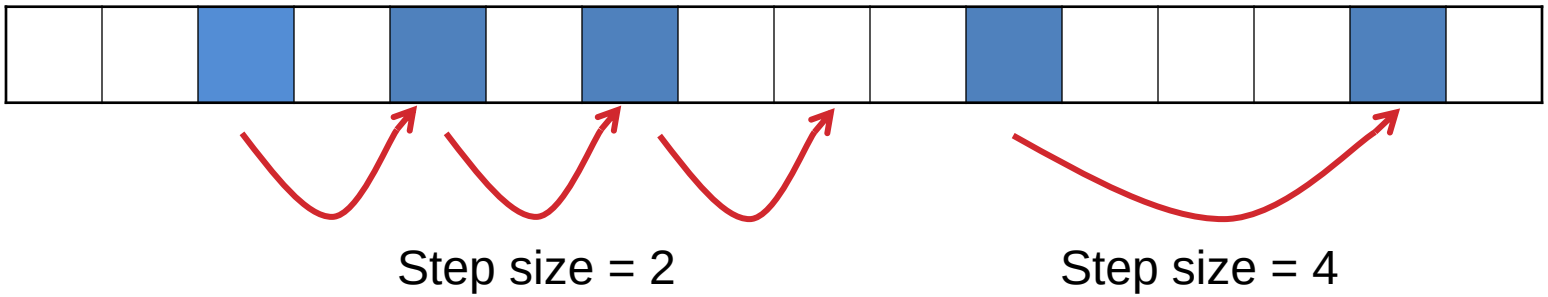
DOUBLE HASHING

- **One problem with linear probing**
  - When collisions occur during insertion the linear probing approach puts new data in the next empty array location
  - This may create clusters of values in the hash table
  - Insertions and searches within this cluster may require multiple probes to find desired record (or empty spot)
  - This extra probing can really slow down the hash table



DOUBLE HASHING

- **The solution is to modify the probing algorithm**
  - Instead of probing one location at a time we jump forward multiple locations in the array
  - To decide what step size to use, we use a second hash function based on the key or index
  - This reduces chance of clustering in the hash table



DOUBLE HASHING

- **The double hashing algorithm for inserting data into a hash table is very similar to linear probing**
  - We calculate the hash value for the input data
  - We look at that location in the hash table
  - If the location is empty, we insert the data
  - If the location is already occupied, we use a second hash function to calculate the step size for probing the array
  - We jump forward step locations in the hash table and continue until an empty location is found to insert data
  - We use modulo operation to “wrap around” index

- **We will illustrate the insertion process using phone data**

DOUBLE HASHING

Data insertion phase

- Assume that we have already inserted two phone records into the hash table

Index	Name	Phone
0		
1	White, Betty	521-6543
2		
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

DOUBLE HASHING

Data insertion phase

- Want to insert phone record with hash("Smith, John") = 2
- Location 2 is empty, so we insert new data there

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4		
5		
6	Jones, Tom	521-1234
7		
8		

DOUBLE HASHING

Data insertion phase

- Want to insert phone record with hash(“Brown, Jim”) = 1
- Location 1 is full so we calculate hash2(1) = 3
- Location 1+3=4 is empty so we insert the record there

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4	Brown, Jim	521-9876
5		
6	Jones, Tom	521-1234
7		
8		

DOUBLE HASHING

Data insertion phase

- Want to insert phone record with hash(“Adams, Doug”) = 1
- Location 1 is full so we calculate hash2(1) = 3
- Location 1+3=4 is also full so we jump forward again
- Location 4+3=7 is empty so we insert the record there

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4	Brown, Jim	521-9876
5		
6	Jones, Tom	521-1234
7	Adams, Doug	521-4242
8		

DOUBLE HASHING

- **When we search the hash table we use hash2 to control the step size for probing the array**
  - We calculate hash(key) to get initial index into hash table
  - If the data there matches the key, the search was success
  - If the location is empty, the search was unsuccessful
  - If the data at first location does not match key, we use hash2 to get step size, add to index, check location.
  - This process repeats until we either find the data or an empty location (when key not found)

DOUBLE HASHING

Data search phase

- Search for phone record with hash("Jones, Tom") = 6
- Location 6 does match so phone record was found

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4	Brown, Jim	521-9876
5		
6	Jones, Tom	521-1234
7	Adams, Doug	521-4242
8		

DOUBLE HASHING

Data search phase

- Search for phone record with hash(“Brown, Jim”) = 1
- Location 1 does not match so calculate hash2(1) = 3
- Location 1+3=4 does match so phone record was found

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4	Brown, Jim	521-9876
5		
6	Jones, Tom	521-1234
7	Adams, Doug	521-4242
8		

DOUBLE HASHING

Data search phase

- Search for phone record with hash(“Claus, Santa”) = 6
- Location 6 does not match so calculate hash2(6) = 2
- Location 6+2=8 is empty so phone record was not found

Index	Name	Phone
0		
1	White, Betty	521-6543
2	Smith, John	521-3456
3		
4	Brown, Jim	521-9876
5		
6	Jones, Tom	521-1234
7	Adams, Doug	521-4242
8		

DOUBLE HASHING

- **With double hashing we need to wrap around the search index whenever we get to the bottom of the hash table**
  - This is done with the modulo operator
  - If size of hash table is N we step forward using

    index = (index + step) % N

- **We can get into big trouble if step is a multiple of N**
  - If N=10 and step=2 and hash(key)=3 we will probe locations 3, 5, 7, 9, 1, 3, 5, 7, 9, 1, etc.
  - We will never probe even locations in the array
  - Solution is to make hash table size N a prime number

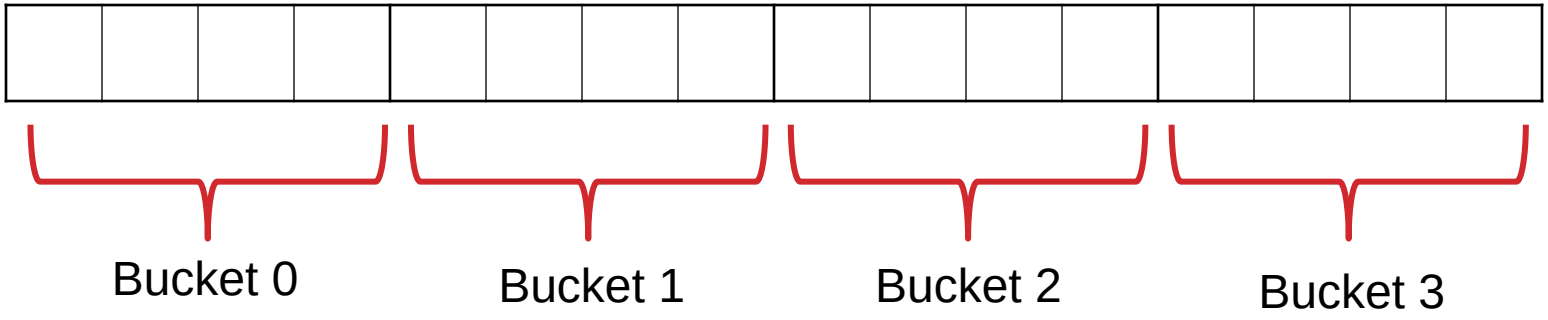
HASH TABLES

HASH BUCKETS

HASH BUCKETS

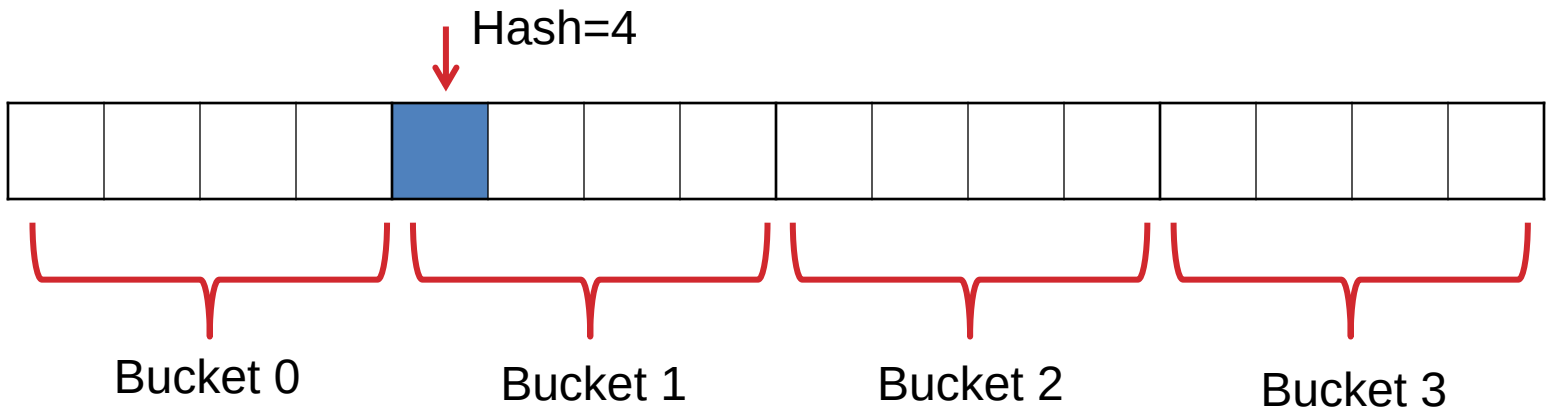
- **With linear probing and double hashing, it is possible for the hash function to go to any location in the array**

- **The idea behind hash buckets is to define a hash function that only returns index values that are multiples of K**
  - This divides the array into N/K buckets of length K



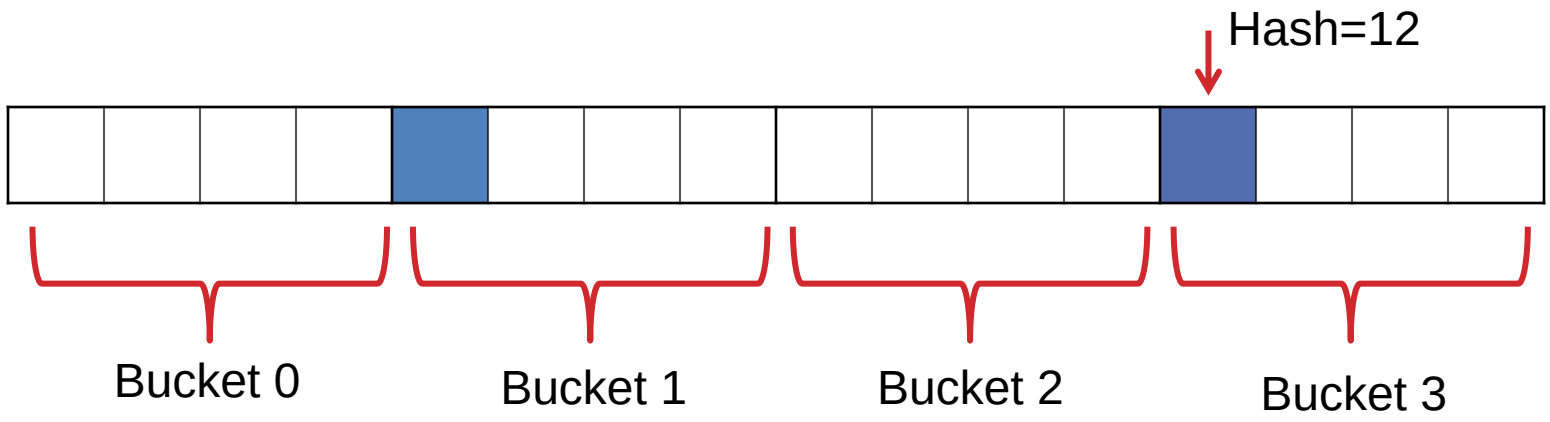
HASH BUCKETS

- **When we insert data into the hash table, we fill each bucket from left to right**
  - The hash function will take us to first location in bucket
  - If this location is empty, we store data there



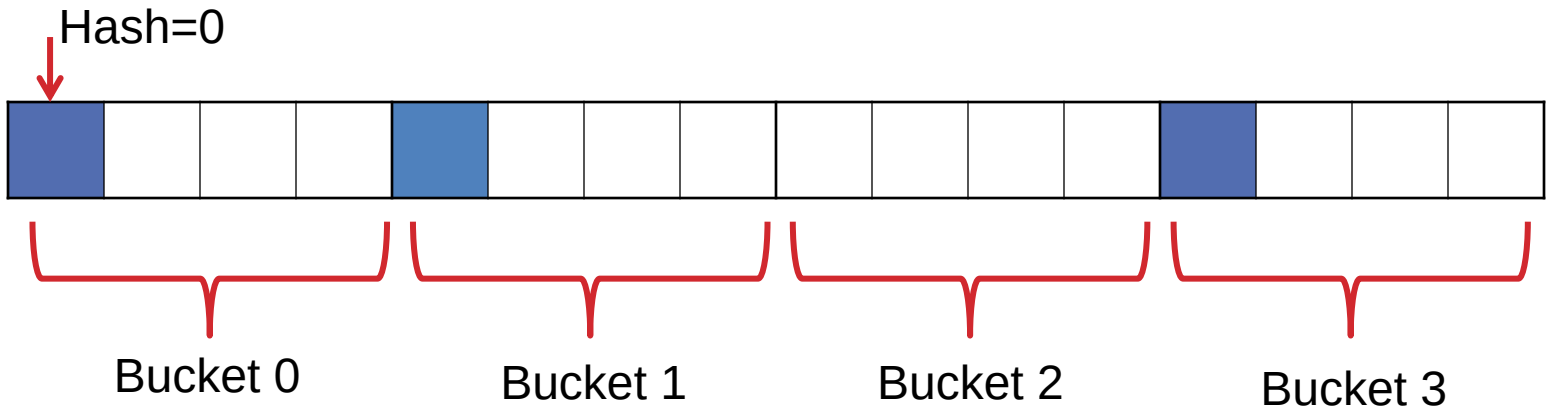
HASH BUCKETS

- **When we insert data into the hash table, we fill each bucket from left to right**
  - The hash function will take us to first location in bucket
  - If this location is empty, we store data there



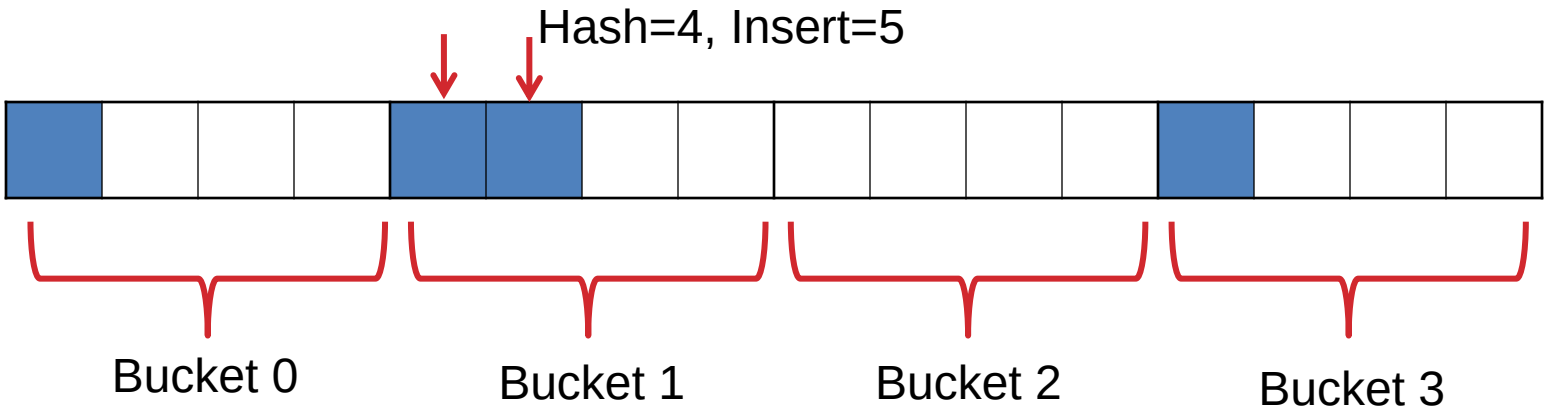
HASH BUCKETS

- **When we insert data into the hash table, we fill each bucket from left to right**
  - The hash function will take us to first location in bucket
  - If this location is empty, we store data there



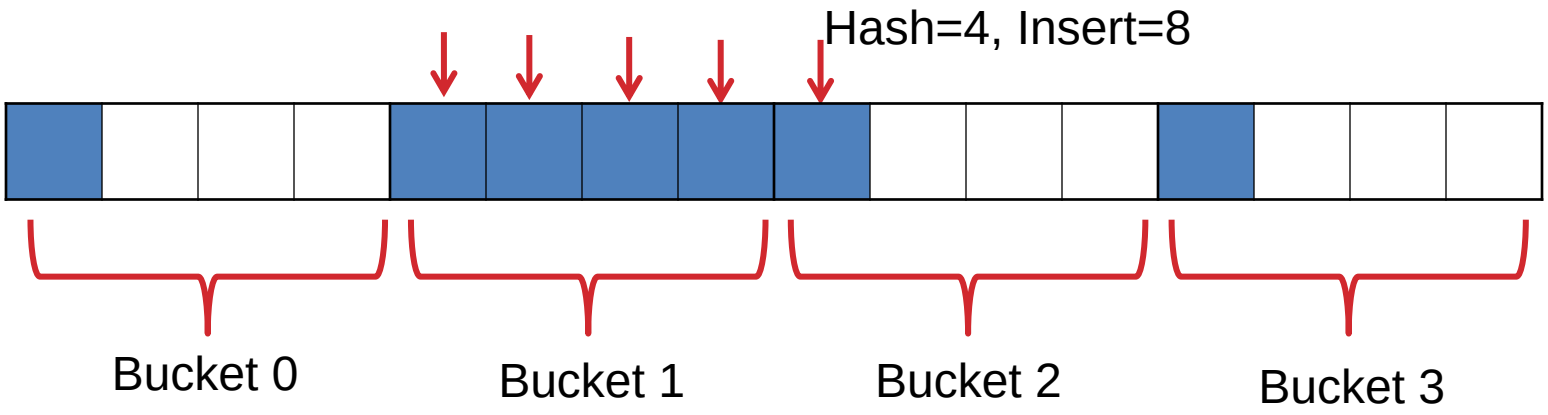
HASH BUCKETS

- **When we insert data into the hash table, we fill each bucket from left to right**
  - The hash function will take us to first location in bucket
  - If this location is empty, we store data there
  - Else we do linear probing to find an empty spot



HASH BUCKETS

- **When we insert data into the hash table, we fill each bucket from left to right**
  - The hash function will take us to first location in bucket
  - If this location is empty, we store data there
  - Linear probing to find empty spot may go beyond bucket



HASH BUCKETS

- **The only difference between hash buckets and linear probing is in the hash function**
  - To create a hash function that returns a multiple of K we multiply the old hash value by K and do mod N again

    NewHash(key) = (OldHash(key) * K) % N

- **Clusters of values can occur with hash buckets but they typically effect only one hash index**
  - Trick is to select value of K to minimize the total number of collisions that occur during insertion and search

HASH TABLES

SEPARATE CHAINING

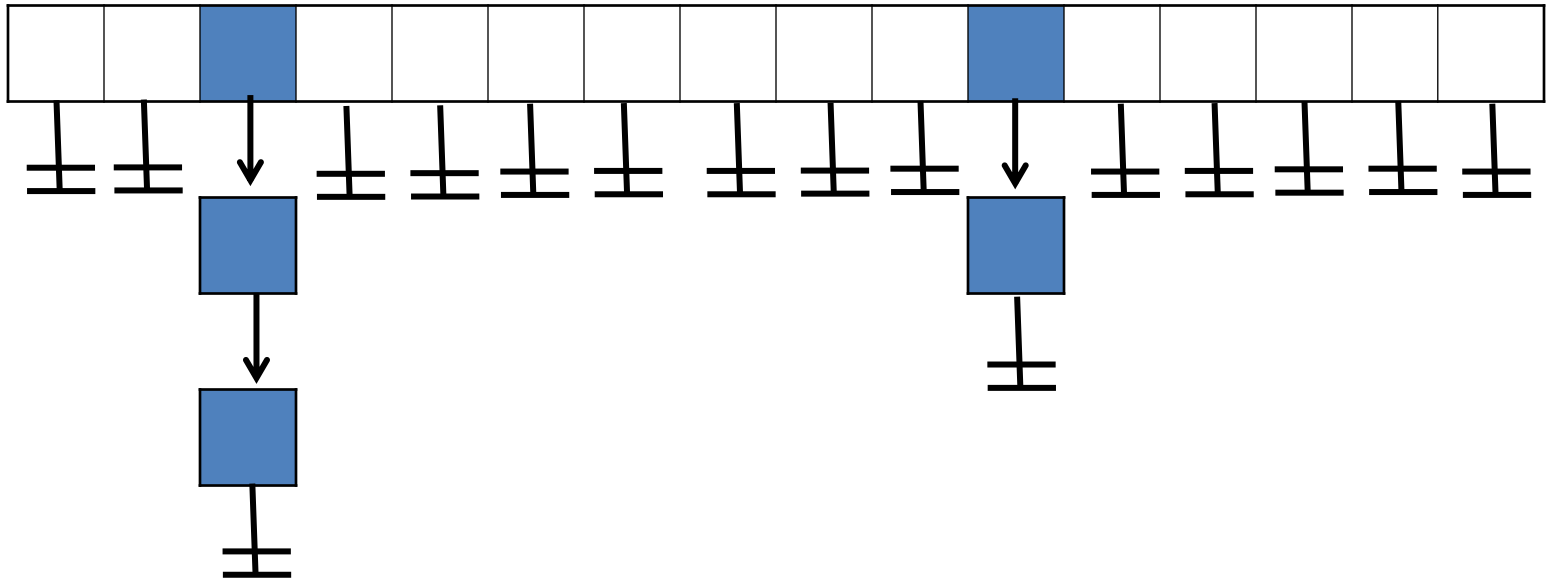
SEPARATE CHAINING

- **All of the hash table techniques above are examples of "open addressing" algorithms**
  - Hash table data is stored in an array of size N
  - We use hash(key) and hash2(index) to probe this array for empty locations when inserting data
  - We use hash(key) and hash2(index) to search this array for the desired data
  - Because the hash table has a fixed size, it is possible for the hash table to become full, which is never good

- **What is the alternative to a fixed size array?**
  - Use a dynamic data structure to store records

SEPARATE CHAINING

- **The separate chaining approach uses an array of linked lists to store the hash table data**
  - When multiple pieces of data hash to the same location we simply insert them into a linked list at that location



SEPARATE CHAINING

- **How do we insert data with separate chaining?**
  - Adapt linked list code so each node can store one piece of hash table data (e.g. name and phone number)
  - Use hash(key) to find the linked list to insert data
  - Call linked list insert method to store data in this linked list

- **How fast is this?**
  - If we use “insert at head” approach the insertion is trivial
  - If we use “sorted insert” approach the insertion has to walk halfway down the linked list on average
  - Most people use “insert at head”

SEPARATE CHAINING

- **How do we search for data with separate chaining?**
  - Adapt linked list class so each node can store one piece of hash table data (e.g. name and phone number)
  - Use hash(key) to find the linked list to search
  - Call linked list search method to walk this linked list until you find a match (or the end of the linked list)

- **How fast is this?**
  - If the linked list is short, this search is fast
  - If the linked list is long, this search is slow
  - Hence we want a hash function that minimizes collisions

SEPARATE CHAINING

```
class HashTable
{
 public:
   HashTable(int size);
   HashTable(const HashTable & ht);
   ~HashTable();

   bool Insert(string key, int value);
   bool Search(string key, int &value);
   bool Delete(string key);
   void Print();
```

Constructors and destructor

Hash table API methods for insertion, search and deletion

SEPARATE CHAINING

```
...
private:
    // Private methods
    int Hash(string key);

    // Private data
    int Size;
    List *Table;
};
```

Private method to calculate the hash value for the key

Dynamic array of linked lists to store hash table data

SEPARATE CHAINING

```
HashTable::HashTable(int size)
{
    Size = size;
    Table = new List[Size];
}
```

Create dynamic array of linked lists

```
HashTable::~HashTable()
{. // delete lists first
    Size = 0;
    delete [] Table;
}
```

Release memory for dynamic array of lists

SEPARATE CHAINING

```
bool HashTable::Insert(string key, int value)
{
    // Find hash index
    int index = Hash(key);
    return Table[index].Insert(key, value);
}
```

Use hash function to calculate array index

Call insert method to add record to this linked list

SEPARATE CHAINING

```
bool HashTable::Search(string key, int &value)
{
    // Find hash index
    int index = Hash(key);
    return Table[index].Search(key, value);
}
```

Use hash function to calculate array index

Call search method to find record in this linked list

SEPARATE CHAINING

```
bool HashTable::Delete(string key)
{
    // Find hash index
    int index = Hash(key);
    return Table[index].Delete(key);
}
```

Use hash function to calculate array index

Call delete method to remove record from this linked list

SEPARATE CHAINING

- **Advantages:**
  - Separate chaining is trivial to implement once you have a linked list for storing hash table data
  - Hash tables using separate chaining can never become full because the linked lists can never become full

- **Disadvantages:**
  - There is some operating system overhead allocating and freeing memory for linked list nodes
  - Search can be slow if linked lists become long

HASH TABLES

SPEED ANALYSIS

SPEED ANALYSIS

- **Speed of hashing depends on the expected number of probes needed during insertion or search**
  - This is a function of how full the hash table is
  - Let α be the fraction of the table that is currently full
  - For each probe find the odds location is occupied or empty

Probe	Occupied	Empty
1	α	$(1-\alpha)$
2	α^2	$(1-\alpha)\alpha$
3	α^3	$(1-\alpha)\alpha^2$
4	α^4	$(1-\alpha)\alpha^3$
...		
n	α^n	$(1-\alpha)\alpha^{n-1}$

Odds to find empty location after 1 probe is $(1-\alpha)$

SPEED ANALYSIS

- **Speed of hashing depends on the expected number of probes needed during insertion or search**
  - This is a function of how full the hash table is
  - Let α be the fraction of the table that is currently full
  - For each probe find the odds location is occupied or empty

Probe	Occupied	Empty
1	α	$(1-\alpha)$
2	α^2	$(1-\alpha)\alpha$
3	α^3	$(1-\alpha)\alpha^2$
4	α^4	$(1-\alpha)\alpha^3$
...		
n	α^n	$(1-\alpha)\alpha^{n-1}$

Odds to find empty location after 2 probes is $(1-\alpha)$ times the odds occupied after 1 probe

SPEED ANALYSIS

- **Speed of hashing depends on the expected number of probes needed during insertion or search**
  - This is a function of how full the hash table is
  - Let α be the fraction of the table that is currently full
  - For each probe find the odds location is occupied or empty

Probe	Occupied	Empty
1	α	$(1-\alpha)$
2	α^2	$(1-\alpha)\alpha$
3	α^3	$(1-\alpha)\alpha^2$
4	α^4	$(1-\alpha)\alpha^3$
...		
n	α^n	$(1-\alpha)\alpha^{n-1}$

Odds to find empty location after 3 probes is $(1-\alpha)$ times the odds occupied after 2 probes

SPEED ANALYSIS

- **Speed of hashing depends on the expected number of probes needed during insertion or search**
  - This is a function of how full the hash table is
  - Let α be the fraction of the table that is currently full
  - For each probe find the odds location is occupied or empty

Probe	Occupied	Empty
1	α	$(1-\alpha)$
2	α^2	$(1-\alpha)\alpha$
3	α^3	$(1-\alpha)\alpha^2$
4	α^4	$(1-\alpha)\alpha^3$
...		
n	α^n	$(1-\alpha)\alpha^{n-1}$

Odds to find empty location after 4 probes is $(1-\alpha)$ times the odds occupied after 3 probes

SPEED ANALYSIS

- **Speed of hashing depends on the expected number of probes needed during insertion or search**
  - This is a function of how full the hash table is
  - Let α be the fraction of the table that is currently full
  - For each probe find the odds location is occupied or empty

Probe	Occupied	Empty
1	α	$(1-\alpha)$
2	α^2	$(1-\alpha)\alpha$
3	α^3	$(1-\alpha)\alpha^2$
4	α^4	$(1-\alpha)\alpha^3$
...		
n	α^n	$(1-\alpha)\alpha^{n-1}$

← Finally odds to find empty location after n probes

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$$

  - Calculate sum of num_probes * odds_empty

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$$
$$\alpha*S(n) = 1*(1-\alpha)\alpha + 2*(1-\alpha)\alpha^2 + 3*(1-\alpha)\alpha^3 + \ldots n*(1-\alpha)\alpha^{n-1}$$

  - Multiply both sides by α

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$

$\alpha*S(n) = 1*(1-\alpha)\alpha + 2*(1-\alpha)\alpha^2 + 3*(1-\alpha)\alpha^3 + \ldots n*(1-\alpha)\alpha^{n-1}$

$(1-\alpha)*S(n) = (1-\alpha) + (1-\alpha)\alpha + (1-\alpha)\alpha^2 + (1-\alpha)\alpha^3 + \ldots (1-\alpha)\alpha^{n-1}$

  - Subtract line 2 from line 1 to cancel integer coefficients

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$$
$$\alpha*S(n) = 1*(1-\alpha)\alpha + 2*(1-\alpha)\alpha^2 + 3*(1-\alpha)\alpha^3 + \ldots n*(1-\alpha)\alpha^{n-1}$$
$$(1-\alpha)*S(n) = (1-\alpha) + (1-\alpha)\alpha + (1-\alpha)\alpha^2 + (1-\alpha)\alpha^3 + \ldots (1-\alpha)\alpha^{n-1}$$
$$S(n) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \alpha^{n-1}$$

  - Divide both sides by $(1-\alpha)$ to simplify formula

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$

$\alpha*S(n) = 1*(1-\alpha)\alpha + 2*(1-\alpha)\alpha^2 + 3*(1-\alpha)\alpha^3 + \ldots n*(1-\alpha)\alpha^{n-1}$

$(1-\alpha)*S(n) = (1-\alpha) + (1-\alpha)\alpha + (1-\alpha)\alpha^2 + (1-\alpha)\alpha^3 + \ldots (1-\alpha)\alpha^{n-1}$

$S(n) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \alpha^{n-1}$

$S(n) - \alpha*S(n) = 1$

  - Subtract $\alpha*S(n)$ from both sides to cancel terms

SPEED ANALYSIS

- **How can we calculate the expected number of probes to find an empty location in the hash table?**

$S(n) = 1*(1-\alpha) + 2*(1-\alpha)\alpha + 3*(1-\alpha)\alpha^2 + \ldots n*(1-\alpha)\alpha^{n-1}$

$\alpha*S(n) = 1*(1-\alpha)\alpha + 2*(1-\alpha)\alpha^2 + 3*(1-\alpha)\alpha^3 + \ldots n*(1-\alpha)\alpha^{n-1}$

$(1-\alpha)*S(n) = (1-\alpha) + (1-\alpha)\alpha + (1-\alpha)\alpha^2 + (1-\alpha)\alpha^3 + \ldots (1-\alpha)\alpha^{n-1}$

$S(n) = 1 + \alpha + \alpha^2 + \alpha^3 + \ldots \alpha^{n-1}$

$S(n) - \alpha*S(n) = 1$

$S(n) = 1 / (1-\alpha)$

- Divide both sides by $(1-\alpha)$ to get closed form solution

SPEED ANALYSIS

- **What does S(n) = 1 / (1-α) mean?**
  - The number of probes does **not** depend on amount of data being stored, it only depends on how **full** the hash table is

- **How should we choose the hash table size?**
  - When α = 0.9, we will have 10 probes on average
  - When α = 0.5, we will have 2 probes on average
  - When α = 0.33 we will have 1.5 probes on average
  - When α = 0.25 we will have 1.33 probes on average
  - When α = 0.1 we will have 1.1 probes on average

- **Common choice to to make hash table 2-4 times larger than the amount of data being stored**

HASH TABLES

SUMMARY

SUMMARY

- **In this section we described how hash functions can be used to convert data fields into hash table locations**

- **We described four hash table implementations**
  - Linear probing jumps to hash location, and searches hash array one location at a time
  - Double hashing jumps to the hash location and searches with step size determined by second hash function
  - Hash buckets are a variation on linear probing where the initial hash locations are K locations apart in the array
  - Separate chaining uses an array of linked lists to store and search for hash table data