GAME2001 Data Structures and Algorithms

Fall 2020

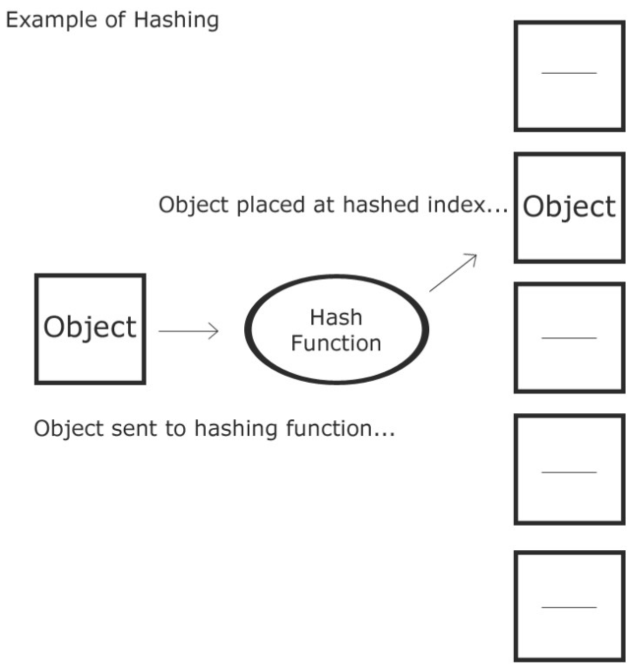
Week 10
Hash Tables

Hash Tables

- fast data structure used to hold a database of information in memory
- very fast insertions and searching for objects that are inserted into the container
  - not dependent on the number of objects already inside a container 0(1)
- very efficient and useful in all types of applications
  - MMORPG - storing a database of players and their stats, items, weapons and equipment, character information, and so forth

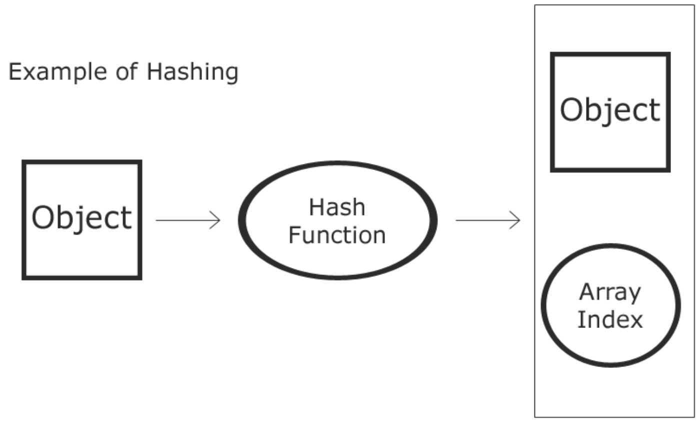
Hash Tables

- based on the array data structure
- expensive and hard to expand
  - if size changes the table has to be rehashed
- copying hash tables can be expensive
  - requires that both tables be of the same size to avoid having to rehash all existing elements
- objects cannot be ordered easily
- insertions and lookups are fast



Hash Tables

- work by taking an object and hashing it to an integer
  - used as the array position for that object in the hash table



Hash Tables

- not perfect data structures
  - poor locality of reference
    - objects get accessed in random locations in memory (cache misses)
  - writing efficient hash tables and algorithms is error prone and can be difficult with advanced techniques
  - a good hash function is needed

Hash Tables

- efficiency of the hashing function is very important
  - a poor hash function can lead to collisions in the data structure
    - when more than one object hashes to the same index

- four things can be done
  - not insert the object
  - replace the old object with the new one
  - find a new position for the object using what is known as open addressing
  - use separate chaining to allow more than one object to exist at an index

Hash Tables

- hash function
  - an algorithm that takes a value (known as the key) and compresses it into the range of the hash table's array
    - can be anything from a string, to a number, and so fourth

- the hash value of the key depends on the size of the hash table's array.
  - a change in the array size invalidates all hashed keys and require them to be rehashed

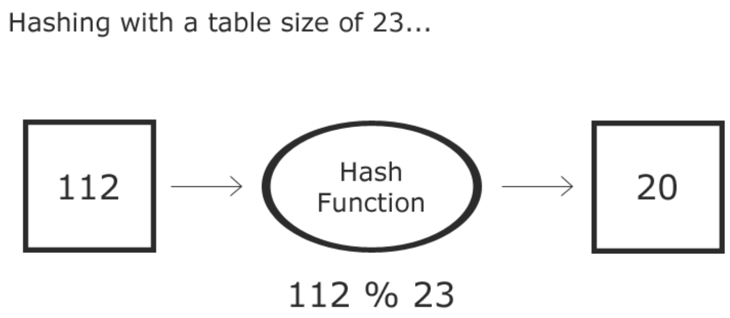
Hash Tables

- when a hash table is created
  - its size can be a prime number
    - necessary for some algorithms such as double hashing
  - roughly twice the number of elements

Hash Tables

- Hashing Values
  - based on the mod operator (%)
  - will be used in conjunction with the array size to return a value that is within the array bounds

key % m_arraySize;



Hash Tables

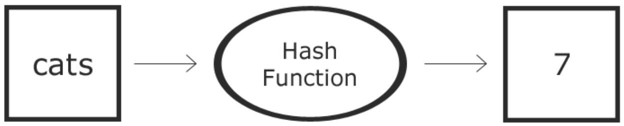
- Hashing Strings
  - simple algorithm that loops through each letter of the key and use the modulus operator to build the hash value

```
int HashFunction(const string& HashString)
{
    int hash = 0;

    for(int i = 0; i < (int)HashString.length(); i++)
          hash += HashString[i];

    return hash % m_size;
}
```

Hashing with a table size of 23...



Hash Tables

- Resolving collisions
  - If we are inserting into the hash table and if an item is already at the position
  - If we are searching for an item and its not at the initial hashed index

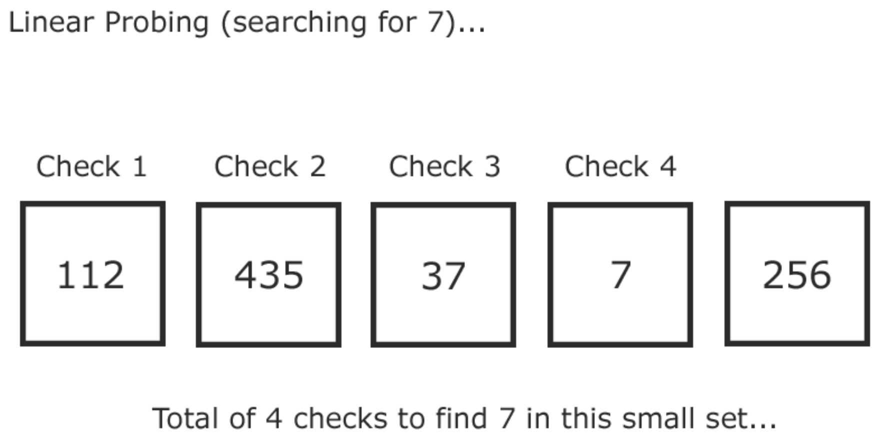
- Open Addressing
  - linear probing, quadric probing, and double hashing
- Separate Chaining

Hash Tables

- Open Addressing: Linear Probing
  - Insertion
    - If a collision occurs sequentially step through the hash table looking for the next empty position
  - Searching
    - If the key at the index doesn’t match linearly step through the table until:
      - You find the value
      - Find an empty slot
      - Have processed all elements

Hash Tables

- Open Addressing: Linear Probing
  - suffers from clustering (primary clustering)
    - when a cluster of objects exist next to one another in the table



Hash Tables

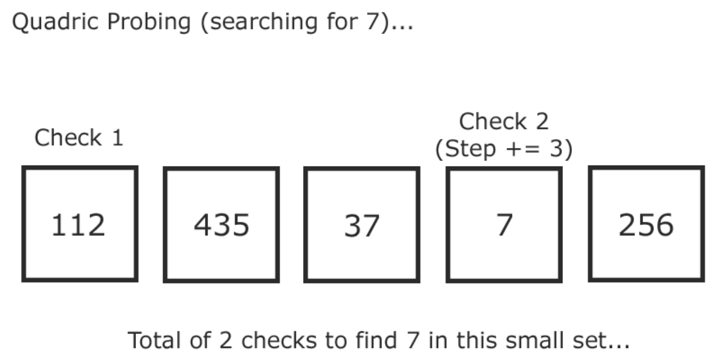
- Open Addressing: Quadric Probing
  - Uses the same principle as linear probing
  - Instead of moving one element at a time it moves in multiple steps

```
// I = number of times the rehash function has been called
hash_value += I*I;
```

  - 1^2 2^2 3^2 4^2 …
  - 1 4 9 16

Hash Tables

- Open Addressing: Quadric Probing
  - Eliminates primary clustering
  - Introduces secondary clustering
    - Because the change in the step size is constant



Hash Tables

- Open Addressing: Double Hashing
  - Uses a diffenent hash function to hash the key to a new position
  - Since the new position is based on a key it will be different for each item
    - Gets rid of clustering (no constant step)

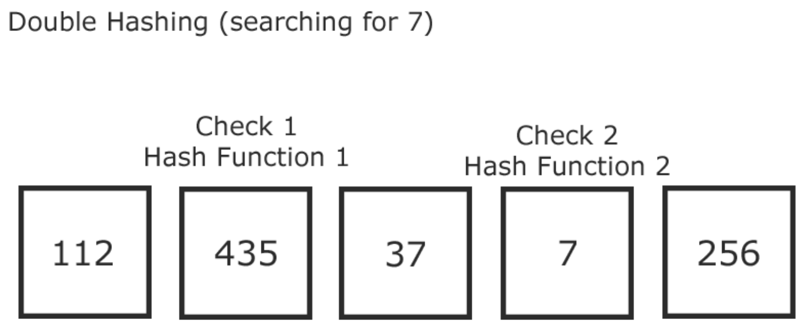
- Key requirements for a second hash
  - Can’t be the same as the first
  - Can’t return 0

Hash Tables

- Open Addressing: Double Hashing

```
double hash = constant - (key % constant);
```

  - Important that the array size is a prime number



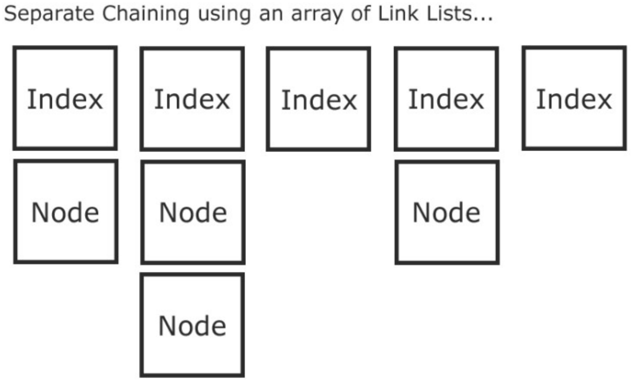
Hash Tables

- Open Addressing
  - does not need to allocate additional memory upon collisions
  - linear and quadric probing suffers from various clustering side effects
  - has a better locality of reference with small tables, which can lead to better performance
  - generally better for smaller tables
  - when using double hashing, it is important to have a prime table size

The ratio of the total number of items to the hash table's size is known as the load factor.

Hash Tables

- Separate Chaining
  - Uses a link list for every element in the hash table’s array instead of looking for a new position
  - If there is a collision the item is inserted into that index’s link list
  - When searching for an item, we can perform a normal search on the link list at that index
  - Don’t need prime table sizes



Hash Tables

- Separate Chaining
  - simple to implement with link lists
  - can use less memory than open addressing
    - since the links are only created when necessary
  - less space is wasted with large tables than using open addressing
  - better for large tables

Hash Tables

- Table Resizing
  - Once the table is resized all keys will need to be rehashed and items reinserted
  - Best option is not to increase the size by one
    - Increase the size by a load factor

Hash Tables

- Implementing Hash Tables
  - STL does not have hash tables
  - we will use prime table sizes
    - can decrease clustering

Hash Item

```
#include <string>

using namespace std;

template<typename T>
class HashItem
{
   public:
      HashItem() : m_key("") {}
      ~HashItem() {}

      string GetKey()        { return m_key; }
      void SetKey(string k) { m_key = k; }

      T GetObject()       { return m_obj; }
      void SetObj(T obj) { m_obj = obj; }

      bool operator==(HashItem &item)
      {
         if(m_key == item.GetKey())
            return true;

         return false;
      }

      void operator=(HashItem item)
      {
         m_key = item.GetKey();
         m_obj = item.GetObject();
      }

   private:
      string m_key;
      T m_obj;
};
```

Hash Table With Linear Probing

```
template<typename T>
class HashTable
{
   public:
      HashTable(int size) : m_size(0), m_totalItems(0)
      {
         if(size > 0)
         {
            m_size = GetNextPrimeNum(size);
            m_table = new HashItem<T>[m_size];
         }
      }


      ~HashTable()
      {
         if(m_table != NULL)
         {
            delete[] m_table;
            m_table = NULL;
         }
      }
```

```
private:
   bool isNumPrime(int val)
   {
      for(int i = 2; (i * i) <= val; i++)
      {
         if((val % i) == 0)
            return false;
      }

      return true;
   }

   int GetNextPrimeNum(int val)
   {
      int i;

      for(i = val + 1; ; i++)
      {
         if(isNumPrime(i))
            break;
      }

      return i;
   }
```

Hash Table With Linear Probing

```
public:
   bool Insert(T &obj)
   {
      if(m_totalItems == m_size)
         return false;

      int hash = HashFunction(obj);

      while(m_table[hash].GetKey() != "")
      {
         hash++;
         hash %= m_size;
      }

      m_table[hash].SetKey(obj.GetHashString());
      m_table[hash].SetObj(obj);

      m_totalItems++;

      return true;
   }
```

```
void Delete(T &obj)
{
   int hash = HashFunction(obj);
   int originalHash = hash;

   while(m_table[hash].GetKey() != "")
   {
      if(m_table[hash].GetKey() == obj.GetHashString())
      {
         m_table[hash].SetKey("");
         m_totalItems--;

         return;
      }

      hash++;
      hash %= m_size;

      if(originalHash == hash)
         return;
   }
}
```

Hash Table With Linear Probing

```
bool Find(string hashString, T *obj)
{
   int hash = HashFunction(hashString);
   int originalHash = hash;

   while(m_table[hash].GetKey() != "")
   {
      if(m_table[hash].GetKey() == hashString)
      {
         if(obj != NULL)
            *obj = m_table[hash].GetObject();

         return true;
      }

      hash++;
      hash %= m_size;

      if(originalHash == hash)
         return false;
   }

   return false;
}
```

Hash Table With Linear Probing

```
int HashFunction(T &obj)
{
    return HashFunction(obj.GetHashString());
}

int HashFunction(const string& HashString)
{
    int hash = 0;

    for(int i = 0; i < (int)HashString.length(); i++)
        hash += HashString[i];

    return hash % m_size;
}
```

```
    int GetSize()
    {
        return m_size;
    }

    int GetTotalItems()
    {
        return m_totalItems;
    }

  private:
    HashItem<T> *m_table;
    int m_size, m_totalItems;
};
```

```cpp
#include <iostream>
#include "LinearProbing.h"

using namespace std;

class PlayerInfo
{
public:
    PlayerInfo() : Name(""), Score(0) {}

    PlayerInfo(string name, int score)
        : Name(name)
        , Score(score)
    {}

    const string& GetHashString() { return Name; }
    const string& GetName() { return Name; }
    int GetScore() { return Score; }

private:
    string Name;
    int Score;
};
int main(int args, char **argc)
{
   cout << "Linear Probing Example" << endl;
   cout << endl;

   // Create table and fill it in.
   HashTable<PlayerInfo> hashTable(20);
   PlayerInfo p1("Joe", 11);
   PlayerInfo p2("Pete", 12);
   PlayerInfo p3("Neta", 2);
   PlayerInfo p4("Nate", 30);
   PlayerInfo p5("Jeff", 5);

   hashTable.Insert(p1);
   hashTable.Insert(p2);
   hashTable.Insert(p3);
   hashTable.Insert(p4);
   hashTable.Insert(p5);

   hashTable.Delete(p5);

   PlayerInfo p6;
   // Search for inserted items.
   if(hashTable.Find("Nate", &p6))
      cout << "Item: Nate has a score of "
           << p6.GetScore() << "." << endl;
   else
      cout << "Item: Nate not found." << endl;

   PlayerInfo p7;
   // Search for inserted items.
   if(hashTable.Find("Jeff", &p7))
       cout << "Item: Jeff has a score of "
            << p7.GetScore() << "." << endl;
   else
       cout << "Item: Jeff not found." << endl;

   return 1;
}
```

```
Linear Probing Example

Item: Nate has a score of 30.
Item: Jeff not found.
```

Hash Table With Double Hashing

```
bool Insert(T &obj)
{
    if(m_totalItems == m_size)
        return false;

    int hash = HashFunction(obj);
    int step = HashFunction2(obj);

    while(m_table[hash].GetKey() != "")
    {
        hash += step;
        hash %= m_size;
    }

    m_table[hash].SetKey(obj.GetHashString());
    m_table[hash].SetObj(obj);

    m_totalItems++;

    return true;
}
```

```
void Delete(T &obj)
{
    int hash = HashFunction(obj);
    int step = HashFunction2(obj);
    int originalHash = hash;

    while(m_table[hash].GetKey() != "")
    {
        if(m_table[hash].GetKey() == obj.GetHashString())
        {
            m_table[hash].SetKey("");
            m_totalItems--;

            return;
        }

        hash += step;
        hash %= m_size;

        if(originalHash == hash)
            return;
    }
}
```

Hash Table With Double Hashing

```
bool Find(string hashString, T *obj)
{
   int hash = HashFunction(hashString);
   int step = HashFunction2(hashString);
   int originalHash = hash;

   while(m_table[hash].GetKey() != "")
   {
      if(m_table[hash].GetKey() == hashString)
      {
         if(obj != NULL)
            *obj = m_table[hash].GetObject();

         return true;
      }

      hash += step;
      hash %= m_size;

      if(originalHash == hash)
         return false;
   }

   return false;
}
```

```
int HashFunction2(T &obj)
{
    return HashFunction2(obj.GetHashString());
}

int HashFunction2(const string& HashString)
{
    int hash = 0;

    for(int i = 0; i < (int)HashString.length(); i++)
        hash = (hash * 256 + HashString[i]) % m_size;

    return hash;
}
```

Hash Table with Separate Chaining

```
template<typename T>
class HashTable
{
   public:
      HashTable(int size) : m_size(0)
      {
         if(size > 0)
         {
            m_size = GetNextPrimeNum(size);
            m_table = new list<HashItem<T> >[m_size];
         }
      }

      ~HashTable()
      {
         if(m_table != NULL)
         {
            delete[] m_table;
            m_table = NULL;
         }
      }
```

```
public:
   void Insert(T &obj)
   {
      HashItem<T> item;
      item.SetKey(obj.GetHashString());
      item.SetObj(obj);

      int hash = HashFunction(obj);
      m_table[hash].push_back(item);
   }

   void Delete(T &obj)
   {
      int hash = HashFunction(obj);

      list<HashItem<T> > *ptr = &m_table[hash];
      typename list<HashItem<T> >::iterator it;

      for(it = ptr->begin(); it != ptr->end(); it++)
      {
         if((*it).GetKey() == obj.GetHashString())
         {
            ptr->erase(it);
            break;
         }
      }
   }
```

Hash Table with Separate Chaining

```
        bool Find(string hashString, T *obj)
        {
           int hash = HashFunction(hashString);

           list<HashItem<T> > *ptr = &m_table[hash];
           typename list<HashItem<T> >::iterator it;

           for(it = ptr->begin(); it != ptr->end(); it++)
           {
              if((*it).GetKey() == hashString)
              {
                 if(obj != NULL)
                    *obj = (*it).GetObject();

                 return true;
              }
           }

           return false;
        }

     private:
        list<HashItem<T> > *m_table;
        int m_size;
};
```