Lecture 11

Announcement

- No class on Monday 3/14 (the following Monday after the Spring break)

- Various variable-length storage
  - Hash Set
  - Hash table
  - Enumerating keys and values in the hash table

Adding moving operator and constructor in binary_tree class.

Hash Set

- An order-insensitive group of elements.
- Example: a group of integer numbers. You want to know whether the number is in the group or not, quickly.
- With a binary-tree, you can do it with O(logN) time and O(N) storage space.
- Hash Set can do it with O(1) time and O(N) storage space.

A set of unsigned integers

- Imagine you need to store an order-insensitive set of unsigned integers.
- If you use a std::vector, what’s the order of computation to check if a number is included in the set?
- How about a binary-tree?

A set of numbers – what if you have an infinitely long table?

- If you have an infinitely long table of 1/0, what is the order of computation for checking if a number is included?

- Bad news: Looking-up is fast, but it takes infinity to initialize such a table.

Solution

- Hash Set: 2D table of elements organized based on the hash code.

Hash Set: Simple case – Hash Code==Hash Key

- Example: A hash set that can check if an unsigned integer is already included in the set.
- Need a table, or an array of variable-length arrays.

Table size=7

Hash Code%7								
0								
1								
2								
3								
4								
5								
6								

Hash Set: Simple case – Hash Code==Hash Key

- After adding 41, 67, 34, 0, 69, 24, 78, 58, 62, 64, 5, 45, 81, 27, 61, 91, 95, 42, 27, 36, the table looks like:
  (* 27 is added twice, but the number can appear only once in the table.)

Table size=7

Hash Code%7								
0	0	91	42					
1	78	64	36					
2	58							
3	24	45						
4	67	81	95					
5	5	61						
6	41	34	69	62				

Hash Set: Simple case – Hash Code==Hash Key

- To check if 23 is included in the set, you can only check elements included in the row of 23%7=2.
- In this case, 23 is not included in the set.

Table size=7

Hash Code%7								
0	0	91	42					
1	78	64	36					
2	58							
3	24	45						
4	67	81	95					
5	5	61						
6	41	34	69	62				

Hash Set: Simple case – Hash Code==Hash Key

- To check if 69 is included in the set, you can only check elements included in the row of 69%7=6.
- In this case, 69 is found after checking against three integers.

Table size=7

Hash Code%7								
0	0	91	42					
1	78	64	36					
2	58							
3	24	45						
4	67	81	95					
5	5	61						
6	41	34	69	62				

Hash Set

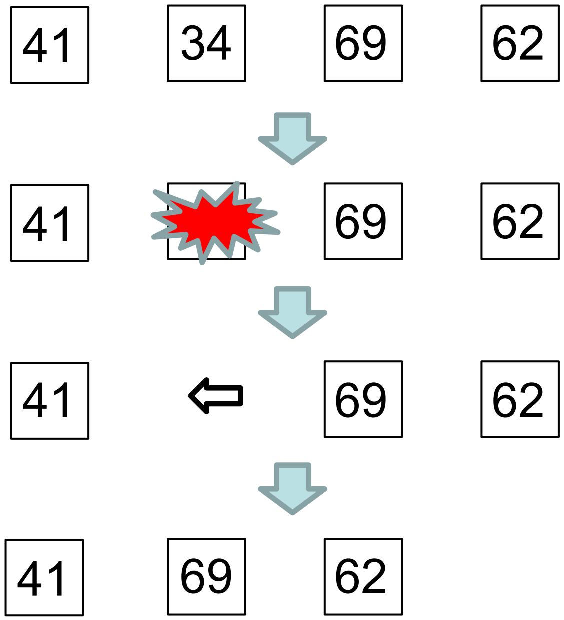
- When you have N elements to look up, make a table with N/C rows (C is a small number compared to N).
- You can find an element in C steps on average.
- Hash Set uses O(N) storage space and reduces the look-up time to O(1).
- By doing this, $O(N^2)$ computation can be reduced to O(N).

```
#ifndef HASHSET_IS_INCLUDED
#define HASHSET_IS_INCLUDED

#include <vector>

template <class KeyType>
class SimpleHashSet
{
private:
    enum
    {
        MINIMUM_TABLE_SIZE=7
    };
    std::vector <std::vector <KeyType> > table;
    long long int nElem;
public:
    SimpleHashSet();
    ~SimpleHashSet();
    void CleanUp(void);
    void Add(KeyType key);
    bool IsIncluded(KeyType key) const;
};

template <class KeyType>
SimpleHashSet<KeyType>::SimpleHashSet()
{
    table.resize(MINIMUM_TABLE_SIZE);
    nElem=0;
}
template <class KeyType>
SimpleHashSet<KeyType>::~SimpleHashSet()
{
}
template <class KeyType>
void SimpleHashSet<KeyType>::CleanUp(void)
{
    table.resize(MINIMUM_HASH_SIZE);
    for(auto &t : table)
    {
        t.clear();
    }
    nElem=0;
}
```

```
template <class KeyType>
void SimpleHashSet<KeyType>::Add(KeyType key)
{
    auto idx=key%table.size();
    for(auto e : table[idx])
    {
        if(e==key)
        {
            return;
        }
    }
    table[idx].push_back(key);
    ++nElem;
}
template <class KeyType>
bool SimpleHashSet<KeyType>::IsIncluded(KeyType key) const
{
    auto idx=key%table.size();
    for(auto e : table[idx])
    {
        if(e==key)
        {
            return true;
        }
    }
    return false;
}

#endif
```

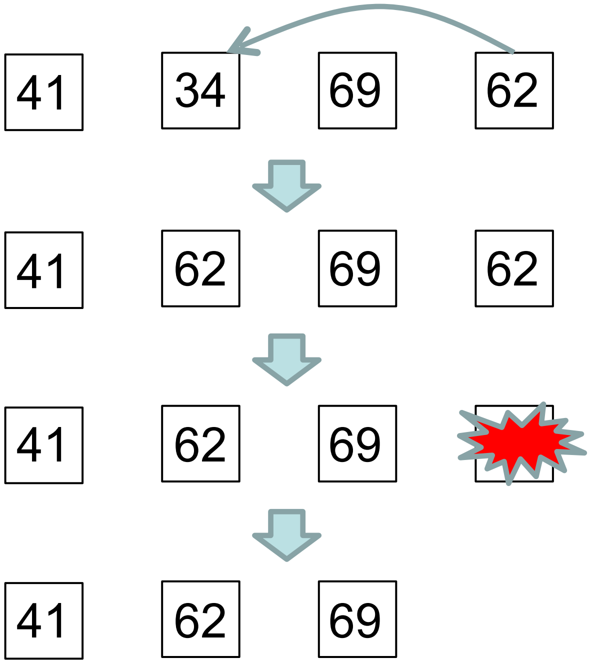
Deleting

- Deleting is just a matter of finding a number and deleting from std::vector.
- The question can be how can I delete an element from std::vector quickly?
- Would you do this?



Deleting (Faster method)

- Take advantage of the order insensitivity.
- Moving the last element of the array to the element which is deleted.
- Shorten the length of the array by one.



In this case, the order in which the keys are stored in the table doesn’t matter. You can delete an element &e in the std::vector by:

1. Overwrite e with the last element in the std::vector.
2. Shrink the size of the array by one.

```
template <class KeyType>
void SimpleHashSet<KeyType>::Delete(KeyType key)
{
    auto idx=key%table.size();
    for(auto &e : table[idx])
    {
        if(e==key)
        {
            e=table[idx].back();
            table[idx].pop_back();
            break;
        }
    }
}
```

Re-sizing

- When the number of elements in the table exceeds certain count, the table may need to grow to stay efficient.
- For example, if you want to keep the average element count per row to four, grow the table size when the total element count exceeds four times the table size.
- An easy option is to move all elements to a separate array, resize the table, and then put them back.

```
template <class KeyType>
void SimpleHashSet<KeyType>::Print(void) const
{
    int idx=0;
    for(auto &t : table)
    {
        printf("[%3d]",idx);
        for(auto e : t)
        {
            printf(" %d",(int)e);
        }
        printf("\n");
        ++idx;
    }
}

template <class KeyType>
void SimpleHashSet<KeyType>::Resize(long long int tableSize)
{
    std::vector <KeyType> buffer;
    for(auto &t : table)
    {
        for(auto e : t)
        {
            buffer.push_back(e);
        }
        t.clear();
    }
    table.resize(tableSize);
    for(auto b : buffer)
    {
        Add(b);
    }
}
```

About the hash table size.

- When you grow your table, you may think it’s a good idea to double it.
- Theoretically, using a prime number as the hash table size gives the best efficiency.

Variation

- Using an array of binary-trees instead of an array of a variable-length arrays.
  - Faster inquiry
  - Slower construction
  - More storage space
- Or, you can keep each row sorted by the hash code. You will get faster look up but slower insertion/deletion.

Hash Set

- General case : Hash Code==A function of Hash Key
- A hash key can be anything, as long as a hash code can be calculated from a hash key, and also hash key is comparable.

- Hash-code collision problem: A hash code from two different hash key may be equal.
- Location of the hash key needs to be found in two steps:
  1. Find a hash code that is same as the code for the key.
  2. Compare if the key is really the same as what is being searched.

```
template <class KeyType>
class HashSet
{
public:
    typedef unsigned long long CodeType;

private:
    class Entry
    {
    public:
        KeyType hashKey;
        CodeType hashCode;
    };

    enum
    {
        MINIMUM_TABLE_SIZE=7
    };
    std::vector <std::vector <Entry> > table;
    long long int nElem;

public:
    unsigned long long int HashCode(const KeyType &key) const;

    HashSet();
    ~HashSet();
    void CleanUp(void);

    void Add(const KeyType &key);
    bool IsIncluded(const KeyType &key) const;
    void Resize(long long int tableSize);
    void Delete(const KeyType &key);

};
```

Hash code is calculated from a key.

```
template <class KeyType>
HashSet<KeyType>::HashSet()
{
    table.resize(MINIMUM_TABLE_SIZE);
    nElem=0;
}
template <class KeyType>
HashSet<KeyType>::~HashSet()
{
}
template <class KeyType>
void HashSet<KeyType>::CleanUp(void)
{
    table.resize(MINIMUM_HASH_SIZE);
    for(auto &t : table)
    {
        t.clear();
    }
    nElem=0;
}
template <class KeyType>
void HashSet<KeyType>::Add(const KeyType &key)
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            return;
        }
    }
    Entry entry;
    entry.hashKey=key;
    entry.hashCode=hashCode;
    table[idx].push_back(entry);
    ++nElem;
}
```

```
template <class KeyType>
bool HashSet<KeyType>::IsIncluded(const KeyType &key) const
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            return true;
        }
    }
    return false;
}
```

First compare hashCode, which probably is faster than comparing a hash key.

Then compare the hash key.

```
template <class KeyType>
void HashSet<KeyType>::Resize(long long int tableSize)
{
    std::vector <KeyType> buffer;
    for(auto &t : table)
    {
        for(auto e : t)
        {
            buffer.push_back(e.hashKey);
        }
        t.clear();
    }
    table.resize(tableSize);
    for(auto b : buffer)
    {
        Add(b);
    }
}
```

```
template <class KeyType>
void HashSet<KeyType>::Delete(const KeyType &key)
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto &e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            e=table[idx].back();
            table[idx].pop_back();
            break;
        }
    }
}
```

Example of usage. Need template specialization.

- Template Specialization: Describing a specific behavior of the template function.

```
#include <string>

#include <stdio.h>
#include <stdlib.h>
#include <time.h>


#include "hashset.h"


template <>
unsigned long long int HashSet<std::string>::HashCode(const std::string &key) const
{
   unsigned long long sum=0;
   for(auto c : key)
   {
      sum+=c;
   }
   return sum;
}
```

```
int main(void)
{
    HashSet <std::string> set;

    set.Add("Phantom");
    set.Add("Tiger");
    set.Add("Crusader");
    set.Add("Tomcat");
    set.Add("Eagle");
    set.Add("Falcon");
    set.Add("Hornet");
    set.Add("Raptor");

    for(;;)
    {
        printf("Enter String>");

        char str[256];
        fgets(str,255,stdin);
        for(int i=0; 0!=str[i]; ++i)
        {
            if(' '>str[i])
            {
                str[i]=0;
                break;
            }
        }

        std::string tst=str;
        if(true==set.IsIncluded(tst))
        {
            printf("Included\n");
        }
        else
        {
            printf("Not Included\n");
        }
    }

    return 0;
}
```

Hash Table

- When you have a hash set, you can make a hash table by adding a value that corresponds to a key.

```
#ifndef HASHSET_IS_INCLUDED
#define HASHSET_IS_INCLUDED

#include <vector>
#include <stdio.h>

template <class KeyType,class ValueType>
class HashTable
{
public:
    typedef unsigned long long CodeType;

private:
    class Entry
    {
    public:
        KeyType hashKey;
        CodeType hashCode;
        ValueType value;
    };

    enum
    {
        MINIMUM_TABLE_SIZE=7
    };
    std::vector <std::vector <Entry> > table;
    long long int nElem;

public:
    unsigned long long int HashCode(const KeyType &key) const;

    HashTable();
    ~HashTable();
    void CleanUp(void);

    void Update(const KeyType &key,const ValueType &value);
    bool IsIncluded(const KeyType &key) const;
    void Resize(long long int tableSize);
    void Delete(const KeyType &key);

    ValueType *operator[](const KeyType key);
    const ValueType *operator[](const KeyType key) const;
};
```

Returns a pointer, not a reference. If no value is set for the key, it can return a nullptr.

```
template <class KeyType,class ValueType>
HashTable<KeyType,ValueType>::HashTable()
{
    table.resize(MINIMUM_TABLE_SIZE);
    nElem=0;
}
template <class KeyType,class ValueType>
HashTable<KeyType,ValueType>::~HashTable()
{
}
template <class KeyType,class ValueType>
void HashTable<KeyType,ValueType>::CleanUp(void)
{
    table.resize(MINIMUM_TABLE_SIZE);
    for(auto &t : table)
    {
        t.clear();
    }
    nElem=0;
}
template <class KeyType,class ValueType>
void HashTable<KeyType,ValueType>::Update(
    const KeyType &key,const ValueType &value)
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto &e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            e.value=value;
            return;
        }
    }
    Entry entry;
    entry.hashKey=key;
    entry.hashCode=hashCode;
    entry.value=value;
    table[idx].push_back(entry);
    ++nElem;
}
```

```
template <class KeyType,class ValueType>
bool HashTable<KeyType,ValueType>::IsIncluded(const KeyType &key) const
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            return true;
        }
    }
    return false;
}
template <class KeyType,class ValueType>
void HashTable<KeyType,ValueType>::Resize(long long int tableSize)
{
    std::vector <Entry> buffer;
    for(auto &t : table)
    {
        for(auto e : t)
        {
            buffer.push_back(e);
        }
        t.clear();
    }
    table.resize(tableSize);
    for(auto b : buffer)
    {
        Update(b.hashKey,b.value);
    }
}

template <class KeyType,class ValueType>
void HashTable<KeyType,ValueType>::Delete(const KeyType &key)
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto &e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            e=table[idx].back();
            table[idx].pop_back();
            break;
        }
    }
}
```

```
template <class KeyType,class ValueType>
ValueType *HashTable<KeyType,ValueType>::operator[](
    const KeyType key)
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto &e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            return &e.value;
        }
    }
    return nullptr;
}

template <class KeyType,class ValueType>
const ValueType *HashTable<KeyType,ValueType>::operator[](
    const KeyType key) const
{
    auto hashCode=HashCode(key);
    auto idx=hashCode%table.size();
    for(auto &e : table[idx])
    {
        if(e.hashCode==hashCode && e.hashKey==key)
        {
            return &e.value;
        }
    }
    return nullptr;
}

#endif
```

Enumerating keys and values in the hash table

- How can I visit keys and values stored in the hash table?
- Same problem for a hash set.
- So far, hash key and hash set are useful for checking if a key is in the set.
- But, not as useful as a storage unless I can enumerate key and values in the hash table and hash set.

- Using element-enumerating handle.

- Is it different from the iterator?
  - Iterator = something that points to an element + pointer to the storage.
  - Iterator itself needs to be able to evaluate what it is pointing.

```
template <class KeyType,class ValueType>
class HashTable
{
public:
    typedef unsigned long long CodeType;
    class EnumHandle
    {
    public:
        long long int hashIdx;
        long long int arrayIdx;
    };
private:
    class Entry
    {
    public:
        KeyType hashKey;
        CodeType hashCode;
        ValueType value;
    };
    enum
    {
        MINIMUM_TABLE_SIZE=7
    };
    std::vector <std::vector <Entry> > table;
    long long int nElem;
public:
    unsigned long long int HashCode(const KeyType &key) const;
    HashTable();
    ~HashTable();
    void CleanUp(void);

    void Update(const KeyType &key,const ValueType &value);
    bool IsIncluded(const KeyType &key) const;
    void Resize(long long int tableSize);
    void Delete(const KeyType &key);

    ValueType *operator[](const KeyType key);
    const ValueType *operator[](const KeyType key) const;

    EnumHandle First(void) const;
    bool IsNotNull(EnumHandle hd) const;
    EnumHandle Next(EnumHandle hd) const;
    ValueType *operator[](EnumHandle hd);
    const ValueType *operator[](EnumHandle hd) const;
    const KeyType &GetKey(EnumHandle hd) const;
};
```

Can be made private, and make
HashTable class as a friend.

```
template <class KeyType,class ValueType>
typename HashTable<KeyType,ValueType>::EnumHandle HashTable<KeyType,ValueType>::First(void) const
{
    EnumHandle hd;
    hd.hashIdx=0;
    hd.arrayIdx=0;
    while(hd.hashIdx<table.size())
    {
        if(0<table[hd.hashIdx].size())
        {
            return hd;
        }
        ++hd.hashIdx;
    }
    hd.hashIdx=-1;
    hd.arrayIdx=-1;
    return hd;
}
```

Increment table index until the first key is found.

```
template <class KeyType,class ValueType>
bool HashTable<KeyType,ValueType>::IsNotNull(EnumHandle hd) const
{
    if(hd.hashIdx<0 || table.size()<=hd.hashIdx)
    {
        return false;
    }
    return true;
}
```

```
template <class KeyType,class ValueType>
typename HashTable<KeyType,ValueType>::EnumHandle HashTable<KeyType,ValueType>::Next(EnumHandle hd) const
{
    if(true!=IsNotNull(hd))
    {
        return First();
    }

    ++hd.arrayIdx;
    if(hd.arrayIdx<table[hd.hashIdx].size())
    {
        return hd;
    }

    hd.arrayIdx=0;
    ++hd.hashIdx;
    while(hd.hashIdx<table.size())
    {
        if(0<table[hd.hashIdx].size())
        {
            return hd;
        }
        ++hd.hashIdx;
    }

    hd.hashIdx=-1;
    hd.arrayIdx=-1;
    return hd;
}
template <class KeyType,class ValueType>
ValueType *HashTable<KeyType,ValueType>::operator[](EnumHandle hd)
{
    return &table[hd.hashIdx][hd.arrayIdx].value;
}
template <class KeyType,class ValueType>
const ValueType *HashTable<KeyType,ValueType>::operator[](EnumHandle hd) const
{
    return &table[hd.hashIdx][hd.arrayIdx].value;
}

template <class KeyType,class ValueType>
const KeyType &HashTable<KeyType,ValueType>::GetKey(EnumHandle hd) const
{
    return table[hd.hashIdx][hd.arrayIdx].hashKey;
}
```