

CSC 427: Data Structures and Algorithm Analysis

Fall 2004

Associative containers

- STL data structures and algorithms
- sets: insert, find, erase, empty, size, begin/end
- maps: operator[], find, erase, empty, size, begin/end
- hash table versions (`hash_set`, `hash_map`)
 - hash function, probing, chaining

1

Standard Template Library (STL)

the C++ Standard Template Library is a standardized collection of data structure and algorithm implementations

- the classes are templated for generality
- we have seen `<vector>`, `<stack>`, `<queue>`, `<list>`, `<algorithm>`
- `<algorithm>` contains many useful routines, e.g., heap operations

```
vector<int> nums;
for (int i = 1; i <= 10; i++) {
    nums.push_back(i);
    nums.push_back(2*i);
}

make_heap(nums.begin(), nums.end()); // orders entries into heap
Display(nums);

pop_heap(nums.begin(), nums.end()); // removes max (root) value, reheapifies

nums.push_back(33);
push_heap(nums.begin(), nums.end()); // adds new item at end (next leaf)
Display(nums);

sort_heap(nums.begin(), nums.end()); // converts the heap into an ordered list
Display(nums);
```

2

Other STL classes

other useful data structures are defined in STL classes, e.g.,

- `set`: represents a set of objects (unordered collection, no duplicates)
- `hash_set`: an efficient (but more complex) set that utilizes hash tables
- `map`: represents a mapping of pairs (e.g., account numbers & balances)
- `hash_map`: an efficient (but more complex) map that utilizes hash tables
- `multimap`: represents a mapping of pairs, can have multiple entries per key
- `hash_multimap`: an efficient (but complex) multimap that utilizes hash tables
- `slist`: singly-linked list
- `deque`: double-ended queue
- `priority_queue`: priority queue
- `bit_vector`: sequence of bits (with constant time access)

all of these data structure include definitions of iterators, so can use algorithms from the `<algorithm>` library

e.g., `<algorithm>` includes `set_union`, `set_intersection`, `set_difference`

3

Set class

the `set` class stores a collection of items with no duplicates

- repeated insertions of the same item still yield only one entry

construct with no arguments, e.g.,

```
set<int> nums;  
set<string> words;
```

member functions include:

```
void insert(const TYPE & item);           // adds item to set (no duplicates)  
void erase(const TYPE & item);           // removes item from the set  
set<TYPE>::iterator find(const TYPE & item); // returns iterator to item in set  
                                           // (or to end() if not found)  
bool empty();                             // returns true if set is empty  
int size();                                 // returns size of set  
set<TYPE> & operator=(const set<TYPE> & rhs); // assignment operator
```

- to traverse the contents of a set, use an iterator and dereference

```
for (set<string>::iterator iter = nums.begin(); iter != nums.end(); iter++) {  
    cout << *iter << endl;  
}
```

4

Set example: counting unique words

```
#include <iostream>
#include <string>
#include <set>
using namespace std;

int main()
{
    set<string> uniqueWords;           // SET OF UNIQUE WORDS

    string str;
    while (cin >> str) {              // REPEATEDLY READ NEXT str
        uniqueWords.insert(str);      // ADD WORD TO SET OF ALL WORDS
    }

    // TRAVERSE AND DISPLAY THE SET ENTRIES
    cout << "There were " << uniqueWords.size() << " unique words: " << endl;
    for (set<string>::iterator iter = uniqueWords.begin(); iter != uniqueWords.end(); iter++) {
        cout << *iter << endl;
    }

    return 0;
}
```

capitalization?

punctuation?

5

Dictionary utilizing set

```
Dictionary::Dictionary()
{
}

void Dictionary::read(istream & istr)
{
    string word;
    while (istr >> word) {
        words.insert(word);
    }
}

void Dictionary::write(ostream & ostr)
{
    for (set<string>::iterator iter = words.begin();
        iter != words.end(); iter++) {
        ostr << *iter << endl;
    }
}

void Dictionary::addWord(const string & str)
{
    string normal = Normalize(str);
    if (normal != "") {
        words.insert(normal);
    }
}

bool Dictionary::isStored(const string & word) const
{
    return words.find(word) != words.end();
}
```

```
class Dictionary
{
public:
    Dictionary();
    void read(istream & istr = cin);
    void write(ostream & ostr = cout);
    void addWord(const string & str);
    bool isStored(const string & str) const;
private:
    set<string> words;
    string Normalize(const string & str) const;
};
```

would this implementation
work well in our speller
program?

better than vector?

better than list?

better than BST?

6

Map class

the `map` class stores a collection of mappings (`<key, value>` pairs)

- can only have one entry per key (subsequent adds replace old values)

construct with no arguments, e.g.,

```
map<string,int> wc;
```

member functions include:

```
ENTRY_TYPE & operator[] (const KEY_TYPE & key); // accesses value at key
void erase(const KEY_TYPE & key); // removes entry from the map
map<KEY_TYPE, ENTRY_TYPE>::iterator find(const KEY_TYPE & key); // returns iterator to entry in map
// (or to end() if not found)
bool empty(); // returns true if map is empty
int size(); // returns size of map
map<KEY_TYPE, ENTRY_TYPE> & operator=(const set<TYPE> & rhs); // assignment operator
```

- to traverse the contents of a map, use an iterator and dereference and access fields

```
for (map<string, int>::iterator iter = wc.begin(); iter != wc.end(); iter++) {
    cout << iter->first << ": " << iter->second << endl;
}
```

7

Map example: keeping word counts

```
#include <iostream>
#include <string>
#include <map>
using namespace std;

int main()
{
    map<string, int> wordCount; // MAP OF WORDS & THEIR COUNTS

    string str;
    while (cin >> str) { // REPEATEDLY READ NEXT str
        if (wordCount.find(str) == wordCount.end()) { // IF ENTRY NOT FOUND FOR str,
            wordCount[str] = 0; // ADD NEW ENTRY WITH VALUE 0
        }
        wordCount[str]++; // INCREMENT ENTRY
    }

    // TRAVERSE AND DISPLAY THE MAP ENTRIES
    for (map<string,int>::iterator iter = wordCount.begin(); iter != wordCount.end(); iter++) {
        cout << iter->first << ": " << iter->second << endl;
    }

    return 0;
}
```

8

Another example: greedy gift givers

consider the following simplification of a programming contest problem:

For a group of gift-giving friends, determine how much more each person gives than they receive (or vice versa).

- assume each person spends a certain amount on a sublist of friends
- each friend gets the same amount (if 2 friends and \$3 dollars, each gets \$1)

- input format:

```
3
Liz Steve Dave
Liz 30 1 Steve
Steve 55 2 Liz Dave
Dave 0 2 Steve Liz
```

of people
names of people
for each person:
his/her name, initial funds,
number of friends, their names

- output format:

```
Liz -3
Steve -24
Dave 27
```

for each person:
amount received – amount given

9

Greedy Gift Givers

use a map to store the amounts for each person

- key is person's name, value is (amount received – amount given) so far
- as each gift is given, add to recipient's entry & subtract from giver's entry

```
map<string, int> peopleMap;

int numPeople;
cin >> numPeople;
for (int i = 0; i < numPeople; i++) {
    string nextPerson;
    cin >> nextPerson;
    peopleMap[nextPerson] = 0;
}

for (int i = 0; i < numPeople; i++) {
    string giver, givee;
    int amount, numGifts;
    cin >> giver >> amount >> numGifts;
    if (numGifts > 0) {
        int eachGift = amount/numGifts;
        for (int k = 0; k < numGifts; k++) {
            cin >> givee;
            peopleMap[givee] += eachGift;
        }
        peopleMap[giver] -= eachGift*numGifts;
    }
}

for (map<string,int>::iterator iter = peopleMap.begin(); iter != peopleMap.end(); iter++) {
    cout << iter->first << ": " << iter->second << endl;
}
```

10

Set and map implementations

how are set and map implemented?

- the documentation for set and map do not specify the data structure
- they do specify that insert, erase, and find are $O(\log N)$, where N is the number of entries
- also, they specify an ordering on the values traversed by an iterator
 - for set, values accessed in increasing order (based on `operator<` for `TYPE`)
 - for map, values accessed in increasing order (based on `operator<` for `KEY_TYPE`)

what data structure(s) provide this behavior?

there do exist more efficient implementations of these data structures

- `hash_set` and `hash_map` use a hash table representation
- as long as certain conditions hold, can provide $O(1)$ insert, erase, find!!!!!!!

11

Hash tables

a hash table is a data structure that supports constant time insertion, deletion, and search on average

- degenerative performance is possible, but unlikely
- it may waste some storage

idea: data items are stored in a table, based on a key

- the key is mapped to an index in the table, where the data is stored/accessed

example: letter frequency

- want to count the number of occurrences of each letter in a file
- have a vector of 26 counters, map each letter to an index
- to count a letter, map to its index and increment

"A" → 0	1
"B" → 1	0
"C" → 2	3
...	...
"Z" → 25	0

12

Mapping examples

extension: word frequency

- must map entire words to indices, e.g.,

"A" → 0	"AA" → 26	"BA" → 52	...
"B" → 1	"AB" → 27	"BB" → 53	...
...
"Z" → 25	"AZ" → 51	"BZ" → 53	...

- **PROBLEM?**

mapping each potential item to a unique index is generally not practical

of 1 letter words = 26
of 2 letter words = $26^2 = 676$
of 3 letter words = $26^3 = 17,576$
...

- even if you limit words to at most 8 characters, need a table of size 217,180,147,158
- for any given file, the table will be mostly empty!

13

Table size < data range

since the actual number of items stored is generally MUCH smaller than the number of potential values/keys:

- can have a smaller, more manageable table

e.g., vector size = 26
possible mapping: map word based on first letter

"A*" → 0	"B*" → 1	...	"Z*" → 25
----------	----------	-----	-----------

e.g., vector size = 1000
possible mapping: add ASCII values of letters, mod by 1000

"AB" → $65 + 66 = 131$

"BANANA" → $66 + 65 + 78 + 65 + 78 + 65 = 417$

"BANANABANANABANANA" → $417 + 417 + 417 = 1251 \% 1000 = 251$

- **POTENTIAL PROBLEMS?**

14

Collisions

the mapping from a key to an index is called a *hash function*

since $|\text{range}(\text{hash function})| < |\text{domain}(\text{hash function})|$,
can have multiple items map to the same index (i.e., a *collision*)

"ACT" $\rightarrow 67 + 65 + 84 = 216$

"CAT" $\rightarrow 67 + 65 + 84 = 216$

techniques exist for handling collisions, but they are costly (LATER)
it's best to avoid collisions as much as possible – HOW?

- want to be sure that the hash function distributes the key evenly
- e.g., "sum of ASCII codes" hash function
 - OK if array size is 1000
 - BAD if array size is 10,000
 - most words are ≤ 8 letters, so max sum of ASCII codes = 1,016
 - so most entries are mapped to first 1/10th of table

15

Better hash function

a good hash function should

- produce an even spread, regardless of table size
- take order of letters into account (to handle anagrams)

```
unsigned int Hash(const string & key, int tableSize)
// acceptable hash function for strings
// Hash(key) = (key[0]*1280 + key[1]*1281 + ... + key[n]*128n) % tableSize
{
    unsigned int hashIndex = 0;

    for (int i = 0; i < key.length(); i++) {
        hashIndex = (hashIndex*128 + key[i]) % tableSize;
    }
    return hashIndex;
}
```

```
unsigned int Hash(const string & key, int tableSize)
// better hash function for strings, often used in practice
{
    unsigned int hashIndex = 0;

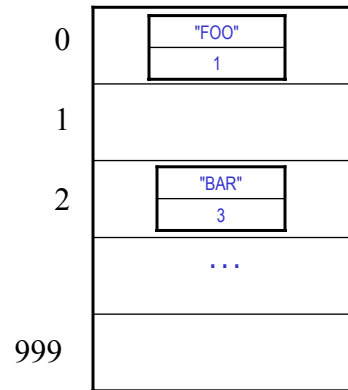
    for (int i = 0; i < key.length(); i++) {
        hashIndex = (hashIndex << 5) ^ key[i] ^ hashIndex;
    }
    return hashIndex % tableSize;
}
```

16

Word frequency example

returning to the word frequency problem

- pick a hash function
- pick a table size
- store word & associated count in the table
- as you read in words, map to an index using the hash function if an entry already exists, increment otherwise, create entry with count = 1



WHAT ABOUT COLLISIONS?

17

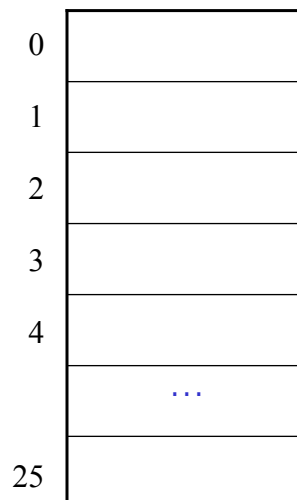
Linear probing

linear probing is a simple strategy for handling collisions

- if a collision occurs, try next index & keep looking until an empty one is found (wrap around to the beginning if necessary)

assume naïve "first letter" hash function

- insert "BOO"
- insert "C++"
- insert "BOO"
- insert "BAZ"
- insert "ZOO"
- insert "ZEBRA"



18

Linear probing (cont.)

with linear probing, will eventually find the item if stored, or an empty space to add it (if the table is not full)

what about deletions?

- delete "BIZ"

can the location be marked as empty?

can't delete an item since it holds a place for the linear probing

- search "C++"

0	"AND" 3
1	"BOO" 1
2	"BIZ" 2
3	"C++" 1
	...

19

Lazy deletion

when removing an entry

- mark the entry as being deleted (i.e., mark location)
- subsequent searches must continue past deleted entries (probe until desired item or an empty location is found)
- subsequent insertions can use deleted locations

ADD "BOO"

ADD "AND"

ADD "BIZ"

ADD "C++"

DELETE "BIZ"

SEARCH "C++"

ADD "COW"

SEARCH "C++"

0	
1	
2	
3	
4	
5	
6	
7	

20

Primary clustering

in practice, probes are not independent

- suppose table is half full

maps to 4-7 require 1 check
map to 3 requires 2 checks
map to 2 requires 3 checks
map to 1 requires 4 checks
map to 0 requires 5 checks

average = $18/8 = 2.25$ checks

0	"AND"
1	"BOO"
2	"BIZ"
3	"C++"
4	
5	
6	
7	

using linear probing, clusters of occupied locations develop

- known as *primary clusters*

insertions into the clusters are expensive & increase the size of the cluster

21

Analysis of linear probing

the *load factor* λ is the fraction of the table that is full

empty table $\lambda = 0$ half full table $\lambda = 0.5$ full table $\lambda = 1$

THEOREM: assuming a reasonably large table, the average number of locations examined per insertion (taking clustering into account) is roughly $(1 + 1/(1-\lambda)^2)/2$

empty table	$(1 + 1/(1 - 0)^2)/2 = 1$
half full	$(1 + 1/(1 - .5)^2)/2 = 2.5$
3/4 full	$(1 + 1/(1 - .75)^2)/2 = 8.5$
9/10 full	$(1 + 1/(1 - .9)^2)/2 = 50.5$

as long as the hash function is fair and the table is less than half full, then inserting, deleting, and searching are all $O(1)$ operations

22

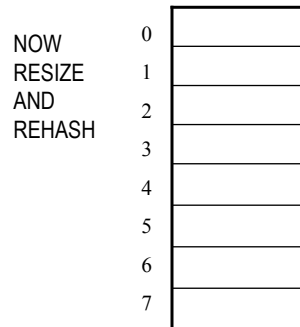
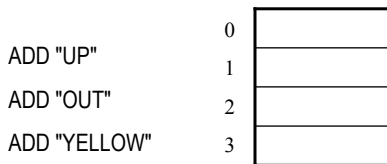
Rehashing

it is imperative to keep the load factor below 0.5

if the table becomes half full, then must resize

- find prime number twice as big
- just copy over table entries to same locations???
- NO! when you resize, you have to rehash existing entries
new table size \rightarrow new hash function (+ different wraparound)

LET $\text{Hash}(\text{word}) = \text{word.length()} \% \text{tableSize}$



23

Chaining

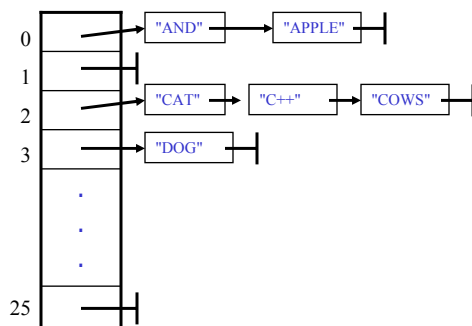
there are variations on linear probing that eliminate primary clustering

- e.g., quadratic probing increases index on each probe by square offset

$\text{Hash}(\text{key}) \rightarrow \text{Hash}(\text{key}) + 1 \rightarrow \text{Hash}(\text{key}) + 4 \rightarrow \text{Hash}(\text{key}) + 9 \rightarrow \text{Hash}(\text{key}) + 16 \rightarrow \dots$

however, the most commonly used strategy for handling collisions is *chaining*

- each entry in the hash table is a bucket (list)
- when you add an entry, hash to correct index then add to bucket
- when you search for an entry, hash to correct index then search sequentially



24

Analysis of chaining

in practice, chaining is generally faster than probing

- cost of insertion is $O(1)$ – simply map to index and add to list
- cost of search is proportional to number of items already mapped to same index
e.g., using naïve "first letter" hash function, searching for "APPLE" might require traversing a list of all words beginning with 'A'

if hash function is fair, then will have roughly $\lambda/\text{tableSize}$ items in each bucket
→ average cost of a successful search is roughly $\lambda/2 * \text{tableSize}$

chaining is sensitive to the load factor, but not as much as probing – WHY?

`hash_map` and `hash_set` in the STL use chaining

- a default hash function is defined for standard classes (e.g., string)
- you can specify your own hash function as part of a `hash_map`/`hash_set` declaration