CSCI 335 Software Design and Analysis III

Sets & Maps in Standard Library

Chapter 4

Ioannis Stamos

STL Containers

- `vector` and `list` are inefficient for search and insert.
- The STL provides the `set` and `map` containers where these methods are guaranteed logarithmic.
- How?

- Recall from the B-Tree the notion of (Key, Value) pairs

STL Container: set

- Properties:
  - Stores objects of type Key in sorted order.
  - The Value is the Key itself, no additional data.
  - No duplicates are allowed.

insert

- iterator, const_iterator as in list/vector
- insert(x) returns iterator
  - Either of newly inserted item, or
  - Already existed item (i.e. insert fails then).
- Two versions of insert:
  - pair<iterator, bool> insert(const Object &x);
  - pair<iterator, bool> insert(iterator hint, const Object &x);
    - If hint is accurate insert is O(1)
  - Example:

```
set<int> s;
  for (int i = 0; i < 10000; i++)
     s.insert(i);
```

pair

- pair<T1,T2> is a heterogeneous pair: it holds one object of type T1 and one of type T2.
- Example:

```
pair<bool, double> result;
result.first = true;
result.second = 0.233;
if (result.first) do_something_more(result.second);
 else report_error();
```

insert

- Example:

```
set<int> s;
for (int i = 0; i < 10000; ++i) {
  const int random_number = rand();
  const auto result = s.insert(random_number);
  // const pair<set<int>::iterator, bool>
  //         result = s.insert(random_number);
  if (!result.second) cerr << “Didn’t insert “ << i << endl;
  // What is result.first ?
 }
```

Insert (with hint)

- Two versions of insert:
  - pair<iterator, bool> insert(const Object &x);
  - pair<iterator, bool> insert(iterator hint, const Object &x);
    - If hint is accurate insert is O(1)
  - Example:

```
set<int> s;
for (int i = 0; i < 10000; ++i) {
const auto result = s.insert(s.end(), i);
// what is result.first ?
}
```

erase

- size_type **erase**(const Object &x)

  erases object x if found. Returns number of objects removed (0 or 1)

- iterator **erase**(iterator itr);

  erases object at itr, returns iterator following itr, invalidates itr.

- void **erase** (iterator start, iterator end);
  - Erase range of values from start to end
  - Note: Efficiency of this can be implementation dependent! We'll see why.

Example

```
set<int> s;
for (int i = 100; i >= 0;  --i)
  const auto result = s.insert(i);
 // contents?
 cout << s.erase(80) << endl;  // contents?

 set<int>::iterator itr1 = s.begin();
 for (int j = 0; j < 20; ++j) ++itr1;
 set<int>::iterator itr2 = itr1;
 ++itr2;
const auto itr3 = s.erase(itr2);  // contents?
s.erase(s.begin(), itr1);  // contents?
```

find

- iterator **find**(const Object &x) const;
  - returns the end iterator (s.end()) in case of failure
- Ordering by default is less<Object>
  - less<T> is a function object. If f is an object of class less<T> and x and y are objects of class T, then f(x,y) returns true if x < y and false otherwise.
  - So, by default operator < is used.

Example find

```
set<int> s;
for (int i = 100; i >= 0;  --i)
   const auto result = s.insert(i);

auto itr = s.find(10);
if (itr != s.end())
   cout << “Did not find  10” << endl;
else
  cout << *itr << endl;  // What is the output here?

itr = s.find(300);  // What is the value of itr here?
```

find

```
20      return findMax( arr, less<Object>( ) );
21  }
22
23  class CaseInsensitiveCompare
24  {
25    public:
26      bool operator( )( const string & lhs, const string & rhs ) const
27        { return stricmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
28  };
29
30  int main( )
31  {
32      vector<string> arr( 3 );
33      arr[ 0 ] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
34      cout << findMax( arr, CaseInsensitiveCompare( ) ) << endl;
35      cout << findMax( arr ) << endl;
36
37      return 0;
38  }
```

```
set<string, CaseInsesitiveCompare> s;
s.insert("Hello"); s.insert("HeLLo");
cout << "The size is: " << s.size() << endl;
```

Maps

- Collection of order entries consisting of **keys and their values**.
  - Keys must be unique.
- iterator's value is a pair
  - *itr is of type pair<KeyType, ValueType>
- .begin(),.end(),.size(),.empty(), .insert(),.find(),.erase()
- insert(const pair<KeyType, ValueType> &x);
- find(const KeyType &x); // Returns a pair-valued iterator.
- ValueType & **operator**[](const KeyType & key);
  - If key is in the Map a reference to the value is returned
  - If key is not in the map, the key is inserted, and a reference to the value is returned (now the value is initialized with the **zero-parameter constructor**)
- Can you use [] **in a constant map?**

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

void foo() {
   map<string, double> salaries;
   salaries.insert(pair<string, double>{“Chris”, 75000.0});
   salaries.insert(pair<string, double>{“Helen”, 85000.0});
   const pair<string, double> a_pair{“Calliope”, 10000.0};
   salaries.insert(a_pair);
   map<string, double>::const_iterator itr = salaries.find(“Helen”);
   // (*itr).first, i.e. itr->first is the key (of type string).
   // (*itr).second, i.e. itr->second is the value (of type double).
   if (itr == salaries.end()) return;  // What is happening here?
   cout << itr->first << endl;
   cout << itr->second << endl;
}
```

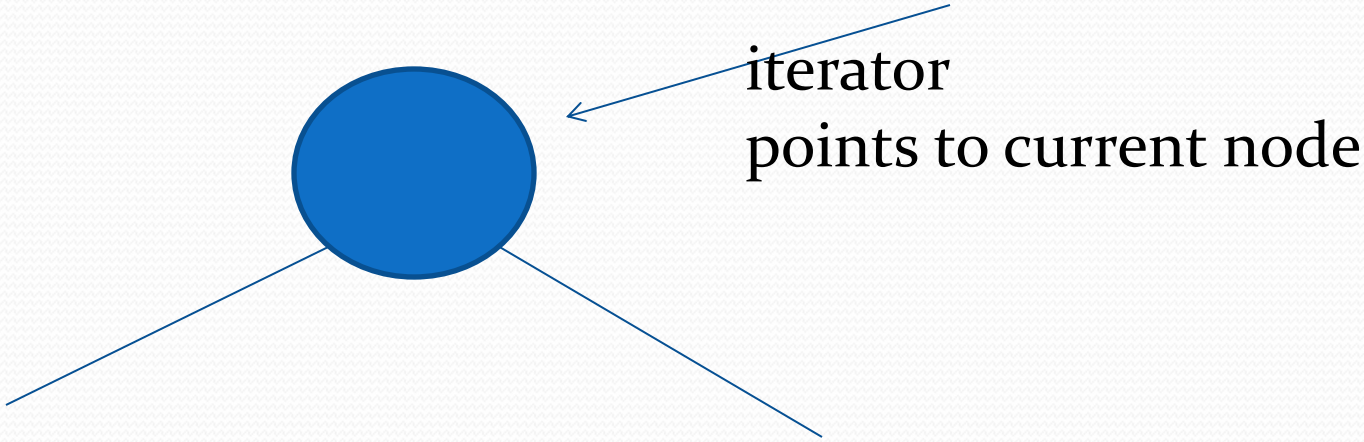
Example

```
1       map<string,double> salaries;
2
3       salaries[ "Pat" ] = 75000.00;
4       cout << salaries[ "Pat" ] << endl;
5       cout << salaries[ "Jan" ] << endl;
6
7       map<string,double>::const_iterator itr;
8       itr = salaries.find( "Chris" );
9       if( itr == salaries.end( ) )
10          cout << "Not an employee of this company!" << endl;
11      else
12          cout << itr->second << endl;
```

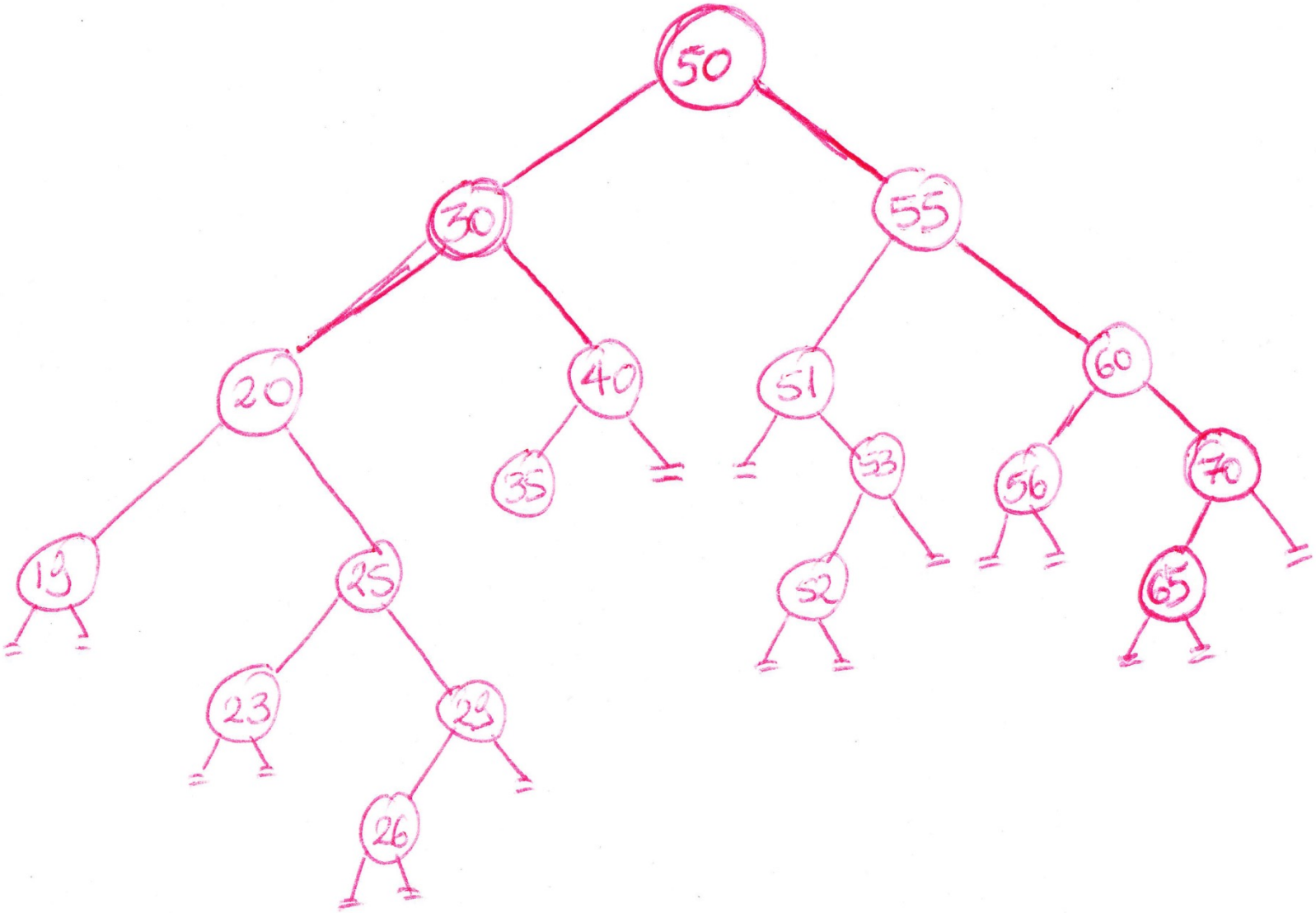
Can we write “itr->second = 20000.00” ?

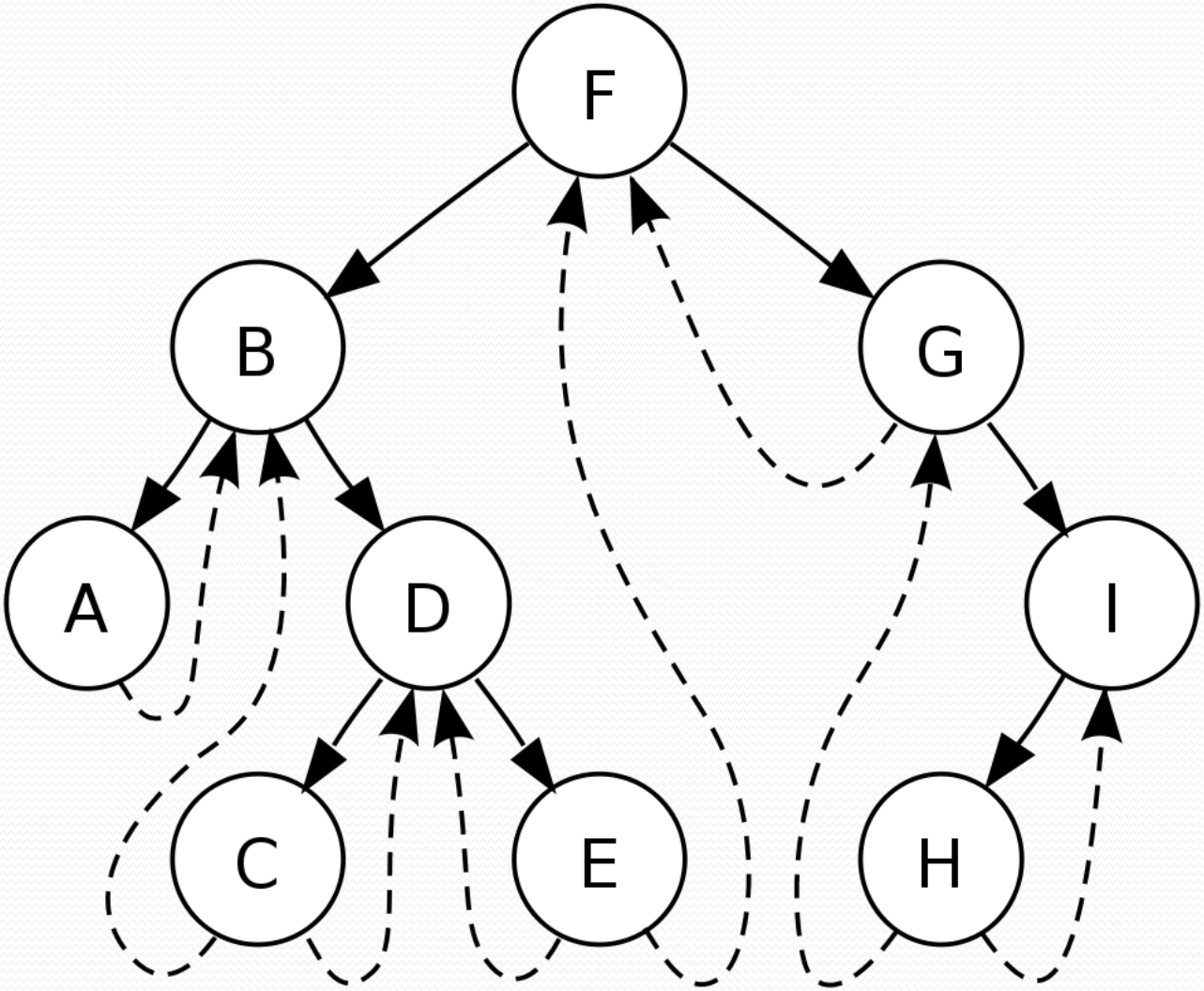
Implementation of set/map in C++

- insert(), erase(), find() in logarithmic time (worst case)=> ?
- iterator internally “points” to current node. How to efficiently advance to the next node?



Smart solution: **threaded** tree


50
30
55
20
40
51
60
35
53
56
70
19
25
52
65
23
29
26


F
B
G
A
D
I
C
E
H

An example

- Input: a dictionary of words (89,000 in this case)
- Problem: find all words that can be changed in at least 15 other words by a single one-character substitution.
- Example: wine -> dine, fine, line, pine, vine

  ->wind, wing, wink, wins…

An example

- Solution?
- map with keys being the words, and values being a vector of words:

  (“wine”, <“dine”,”fine”,”mine”,”nine”,....>)

  string vector<string>

Given a map output words

```
1   void printHighChangeables( const map<string,vector<string> > & adjWords,
2                              int minWords = 15 )
3   {
4       map<string,vector<string> >::const_iterator itr;
5
6       for( itr = adjWords.begin( ); itr != adjWords.end( ); ++itr )
7       {
8           const pair<string,vector<string> > & entry = *itr;
9           const vector<string>                & words = entry.second;
10
11          if( words.size( ) >= minWords )
12          {
13              cout << entry.first << " (" << words.size( ) << "):";
14              for( int i = 0; i < words.size( ); i++ )
15                  cout << " " << words[ i ];
16              cout << endl;
17          }
18
19      }
20  }
```

With C++11 elements

```
// @param adjacent_words: input map from string to vector of strings.
// @param min_words: minimum number of words to consider for printing.
// The functions couts the contents of the map only for the elements
// for which the vector of strings has size greater than or equal to min_words.
void PrintHighChangeables(const map<string, vector<string>> &adjacent_words,
                          int min_words = 15) {
    for (auto &entry : adjacent_words) {
       const vector<string> &words = entry.second;
       if (words.size( ) >= min_words) {
           cout << entry.first << " (" << words.size( ) << "):";
           for (auto &str : words) cout << " " << str;
           cout << endl;
       }
    }
}
```

Check if two words differ by one character

```
1   // Returns true if word1 and word2 are the same length
2   // and differ in only one character.
3   bool oneCharOff( const string & word1, const string & word2 )
4   {
5       if( word1.length( ) != word2.length( ) )
6           return false;
7
8       int diffs = 0;
9
10      for( int i = 0; i < word1.length( ); i++ )
11          if( word1[ i ] != word2[ i ] )
12              if( ++diffs > 1 )
13                  return false;
14
15      return diffs == 1;
16  }
```

Construct the Map

```
// Computes a map in which the keys are words and values are
// vectors of words
// that differ in only one character from the corresponding key.
// Uses a quadratic algorithm.
map<string, vector<string>>
ComputeAdjacentWordsSlow(const vector<string> &words) {
    map<string, vector<string>> adjacent_words;
    for (int i = 0; i < words.size(); ++i)
        for (int j = i + 1; j < words.size(); ++j)
            if (OneCharOff(words[i], words[j])) {
                adjacent_words[words[i]].push_back(words[j]);
                adjacent_words[words[j]].push_back(words[i]);
            }
    return adjacent_words;
}
```

Construct map (better)

- Just compare words of equal size only
- =>Organize words by length. How?
  - 1 -> all words of length 1
  - 2 -> all words of length 2
  - …
- Can you use a map for this?

```
map<string, vector<string>>
ComputeAdjacentWordsMedium(const vector<string> &words) {
    map<string, vector<string>> adjacent_words;
    map<int, vector<string>> words_by_length;
    // Group the words by their length.
    for (auto &this_word : words)
        words_by_length[this_word.length()].push_back(this_word);
    // Work on each group separately.
    for (auto &entry : words_by_length) {
        const vector<string> &word_groups = entry.second;
        for (int i = 0; i < word_groups.size(); ++i)
            for (int j = i + 1; j < word_groups.size(); ++j)
                if (OneCharOff(word_groups[i], word_groups[j])) {
                    adjacent_words[word_groups[i]].push_back(word_groups[j]);
                    adjacent_words[word_groups[j]].push_back(word_groups[i]);
                }
    }
    return adjacent_words;
}
```

Even better….

- Idea:

  Organize words by length as before

  Consider words of length 4 for example

  words “wine”, “dine”, “fine”, … have “ine” as their representative

  construct a map:

  (“ine”, <“wine”, “dine”, “fine”,….>)

  => key is the common 3-letter part of the words

Even better…

For each group g (contain words of length len)

  for each position p (0 through len-1)

    Make empty map<string,vector<string> > representatives

    for each word w in group g

    {

      Obtain w's representative by removing position p

      Update representative

    }

    Use cliques in representatives

Even better…

```
for each group g, containing words of length len {
    for each position p (ranging from 0 to len-1) {
        Make an empty map<string,vector<string>> representatives
        for each word w {
            Obtain w's representative by removing position p
            Update representatives
        }
    for each rep in representatives {
        for each pair of words in rep's clique
            Update adjacent_words
    }
}
```

- with map, with unordered_map.

Even better

```
map<string,vector<string>>
ComputeAdjacentWords(const vector<string> &words) {
    map<string,vector<string>> adjacent_words;
    map<int,vector<string>> words_by_length;
    // Group the words by their length
    for (auto & this_word : words )
          words_by_length[this_word.length()].push_back(this_word);
    //Continues
```

```
    // Work on each group separately.
   for (auto &entry : words_by_length) {
       const vector<string>  &word_groups= entry.second;
       const int num_group = entry.first;
       // Work on each position in each group.
       for (int i = 0; i < num_group; ++i ) {
           // Remove one character in specified position, computing representative.
           // Words with same representatives are adjacent; so first populate a map.
           map<string,vector<string>> representatives;
           for (auto &str : word_groups) {
               string rep = str;
               rep.erase(i, 1);
               representatives[rep].push_back(str);
           }
           // Look for map values with more than one string.
           for (auto &entry2 : representatives) {
               const vector<string> & clique = entry2.second;
               if (clique.size() >= 2) {
                   for (int p = 0; p < clique.size( ); ++p)
                       for (int q = p + 1; q < clique.size( ); ++q) {
                           adjacent_words[clique[p]].push_back(clique[q]);
                           adjacent_words[clique[q]].push_back(clique[p]);
                       }
               }
       }  // End of second for loop.
   } // End of top-level for loop.
  return adjacent_words;
}
```

Summary

- Sets and maps in the STL
- Do you really need a sorted map for all applications?
  - C++11 offers unordered_map