CSCI 335
Software Design and Analysis III
Lecture 11: Part 1 Sets, Maps, Trees Summary

Professor Anita Raja

10-06-22

1

1

Announcements

- No class (Hunter closed) Monday 10/10
- Survey available after class, due Tuesday 10/11
- HW2 due 10/13
- In-class low stake review exercise 10/13
- Midterm 10/20 classtime (all material covered including 10/17 lecture)

2

2

Agenda

- B-trees
- Sets, Maps
- Hash tables
- Go over HW1 solution

3

3

STL Containers

- `vector` and `list` are inefficient for search and insert (Chapter 3).
- The STL provides the `set` and `map` containers
  - insertion, deletion and searching are guaranteed logarithmic.
- How?
  - Recall from the B-Tree the notion of (Key, Value) pairs

4

4

STL Container: set

- Properties:
  - Ordered container that does not allow duplicates.
  - Stores objects of type Key in sorted order.
  - The Value is the Key itself, no additional data.

5

insert

- iterator, const_iterator as in list/vector
- insert(x) returns iterator
  - Either of newly inserted item, or
  - Already existing 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);
```


6

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();
```


7

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);
}
```


9

erase

- size_type **erase**(const Object &x)
  - erases object x if found. Returns number of objects removed (0 or 1)
- iterator **erase**(iterator itr);
  - Same as in vector and list.
  - erases object at itr, returns iterator following itr, invalidates itr.
- void **erase** (iterator start, iterator end);
  - Same as in vector and list.
  - Erase range of values from start to end but not including item at end.
  - Note: Efficiency of this can be implementation dependent!

10

10

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.

12

12

find

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

```
// Generic findMax, with a function object, C++ style.
// Precondition: a.size( ) > 0.
template <typename Object, typename Comparator>
const Object & findMax( const vector<Object> & arr, Comparator isLessThan )
{
    int maxIndex = 0;

    for( int i = 1; i < arr.size( ); i++ )
        if( isLessThan( arr[ maxIndex ], arr[ i ] ) )
            maxIndex = i;

    return arr[ maxIndex ];
}

// Generic findMax, using default ordering.
#include <functional>
template <typename Object>
const Object & findMax( const vector<Object> & arr )
{
    return findMax( arr, less<Object>( ) );
}

class CaseInsensitiveCompare
{
  public:
    bool operator( )( const string & lhs, const string & rhs ) const
      { return stricmp( lhs.c_str( ), rhs.c_str( ) ) < 0; }
};

int main( )
{
    vector<string> arr( 3 );
    arr[ 0 ] = "ZEBRA"; arr[ 1 ] = "alligator"; arr[ 2 ] = "crocodile";
    cout << findMax( arr, CaseInsensitiveCompare( ) ) << endl;
    cout << findMax( arr ) << endl;

    return 0;
}
```

14

14

Maps

- Used to store a collection of ordered entries consisting of **keys and their values**.
  - Keys must be unique.
  - Several keys can map to the same values (values need not be unique).
  - Keys are maintained in a logically sorted order.
- iterator's value is a pair
  - *itr is of type pair<KeyType, ValueType>
- .begin(),.end(),.size(),.empty(), .insert(),.find(),.erase()

15

15

Maps

- 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? No!**

16

16

Example

```
map<string,double> salaries;

salaries[ "Pat" ] = 75000.00;
cout << salaries[ "Pat" ] << endl;
cout << salaries[ "Jan" ] << endl;

map<string,double>::const_iterator itr;
itr = salaries.find( "Chris" );
if( itr == salaries.end( ) )
    cout << "Not an employee of this company!" << endl;
else
    cout << itr->second << endl;
```

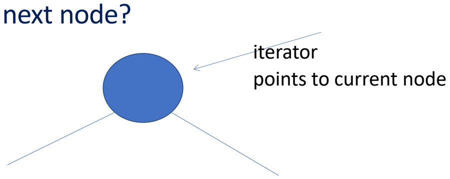
Can we write “itr->second = 20000.00” ?

18

18

Implementation of set/map in C++

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



Smart solution: **threaded** tree

19

19

Why do we need Threaded Binary Tree?

- Binary trees have a lot of wasted space:
  - the leaf nodes each have 2 null pointers. We can use these pointers to help us in inorder traversals.
- Threaded binary tree makes the tree traversal faster since we do not need stack or recursion for traversal

20

20

What is Threaded Binary Tree??

- A binary tree is threaded by
  - making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and
  - all left child pointers that would normally be null point to the inorder predecessor of the node.
- We have the pointers reference the next node in an inorder traversal; called threads
- We need to know if a pointer is an actual link or a thread, so we keep a boolean for each pointer

21

21

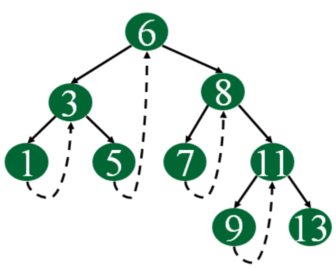
Types of threaded binary trees:

- **Single Threaded**: each node is threaded towards either the in-order predecessor or successor (left **or** right) means all right null pointers will point to inorder successor **OR** all left null pointers will point to inorder predecessor.

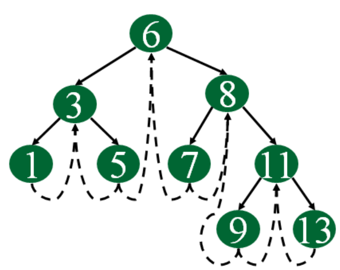
- **Double threaded:** each node is threaded towards both the in-order predecessor and successor (left **and**right) means all right null pointers will point to inorder successor **AND** all left null pointers will point to inorder predecessor.

22

22



Single Threaded Binary Tree



Double Threaded Binary Tree

23

An example

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

26

26

An example

- Solution:
  - 8205 6 letter words, 11989 7 letter, 12K+ eight letter, 13K+ nine letter, 11k+ ten letter and 8k+ eleven letter words.
- Approach:
  - map with keys being the word, and values being a vector of words:

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

string vector<string>

27

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;
        }
    }
}
```

29

Check if two words differ by one character

```
// Returns true if word1 and word2 are the same length
// and differ in only one character.
bool oneCharOff( const string & word1, const string & word2 )
{
    if( word1.length( ) != word2.length( ) )
        return false;

    int diffs = 0;

    for( int i = 0; i < word1.length( ); i++ )
        if( word1[ i ] != word2[ i ] )
            if( ++diffs > 1 )
                return false;

    return diffs == 1;
}
```

30

Construct the Map - Method 1

```
// 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;
}
```

31

Construct map (better) – Method 2

- 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?
  - Yes use a second map – the key is an integer representing a word length and a vector to store each collection

32

32

```
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;}
```

33

33

Even better….Method 3

- 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

34

34

Even better…

Running time is 2 seconds.
Note: use of additional maps makes algorithm faster, syntax is relatively clean ; the code makes no use of the fact that t keys of the map are maintained in sorted order.

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
}

35

35

Ordered vs Unordered maps

- Sets and maps in the STL
- Do you really need a sorted map for all applications?
  - C++11 offers unordered_map – Chapter 5
  - Reduces running time from 2 sec to 1.5sec.

36

36

Summary

- Trees in OS, compiler design, search
- Expression trees:
  - Example of parse tree (not binary) central to compiler design.
- Search trees:
  - Crucial to algorithm design, support useful ops and O(logn) ave cost is small.
  - Non-recursive implementations of search trees are somewhat faster but recursive are sleeker.
  - Important for input to be random for good performance; if not random, running time increases (==expensive linked lists).

37

37

Summary: To avoid performance issues

- AVL trees
  - lhs and rhs tree heights differ at most by 1; not too deep.
  - Operations that don't change the tree use std bst code
  - Operations like insert, delete must restore the tree – show O(logN) approach to restore the tree: single and double rotations
- Splay trees
  - Nodes can get arbitrarily deep but after every access the tree is adjusted in mysterious manner: zig-zag, zig-zig.
  - Net effect: any sequence of M operations takes O(MlogN) which is the same as a balanced tree.
- B-trees
  - Balanced M-way (as opposed two–way or binary trees) which are well suited for disks, a special case of 2-3 tree (M=3) is another way to implement balanced search tress.

38

38

Summary: Balanced-tree schemes

- In practice, running time of all these schemes
  - while slightly faster for searching,
  - is worse (by a constant factor) for insertions, deletions than the simple BST.
  - This is generally acceptable in view of the protection being given against easily obtained worse-case input.
  - See chapter 12 for more details.
- Finally, by inserting elements into a search tree and then performing an inorder tree traversal, we obtain the elements in a sorted order.
  - Gives O(NlogN) algorithm to sort which is worse case bound if any sophisticated search tree is used.
  - Chapter 7 better way but none with a lower time bound.

39

39