

University of Southern California

Viterbi School of Engineering

Software Design

C++ STLs – Iterators

ADTs

Reference: Professor Mark Redekopp's Course Materials, Online Resources

No more 'for(i=0; i < max; i++)'

ITERATORS

Arrays Review

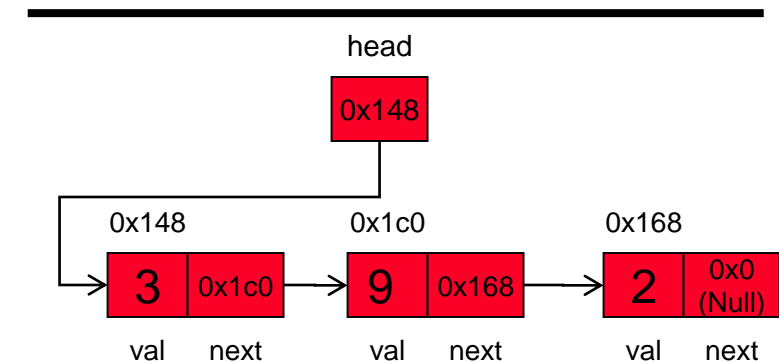
- Arrays are contiguous pieces of memory
- To find a single value, computer only needs
 - The start address
 - Remember the name of the array evaluates to the starting address, e.g., data = 120
 - Which element we want
 - Provided as an index, e.g., [20]
 - This is all thanks to the fact that items are contiguous in memory
- Linked list items are not contiguous
 - Thus, linked lists have an explicit field to indicate where the next item is

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

int main()
{
    int data[25];
    data[20] = 7;
    return 0;
}
```

data = 100

100	104	108	112	116	120	
45	31	21	04	98	73	...
Memory						

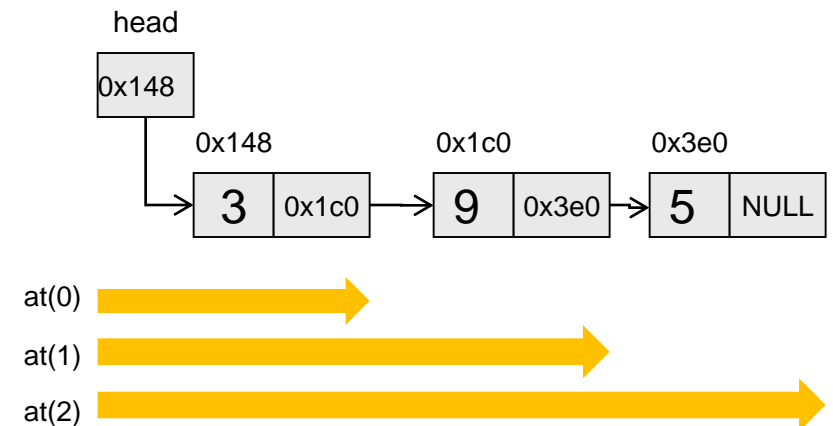


Iteration

- Consider how you iterate over all the elements in a list
 - Use a for loop and `at()` or `operator[]`
- For an array list (vector) this is fine since each call to `at()` is $O(1)$
- For a linked list, calling `at(i)` requires taking i steps through the linked list
 - 0^{th} call = 0 steps
 - 1^{st} call = 1 step
 - 2^{nd} call = 2 steps
 - $0+1+2+\dots+n-2+n-1 = O(n^2)$
- You are re-walking over the linked list a lot of the time

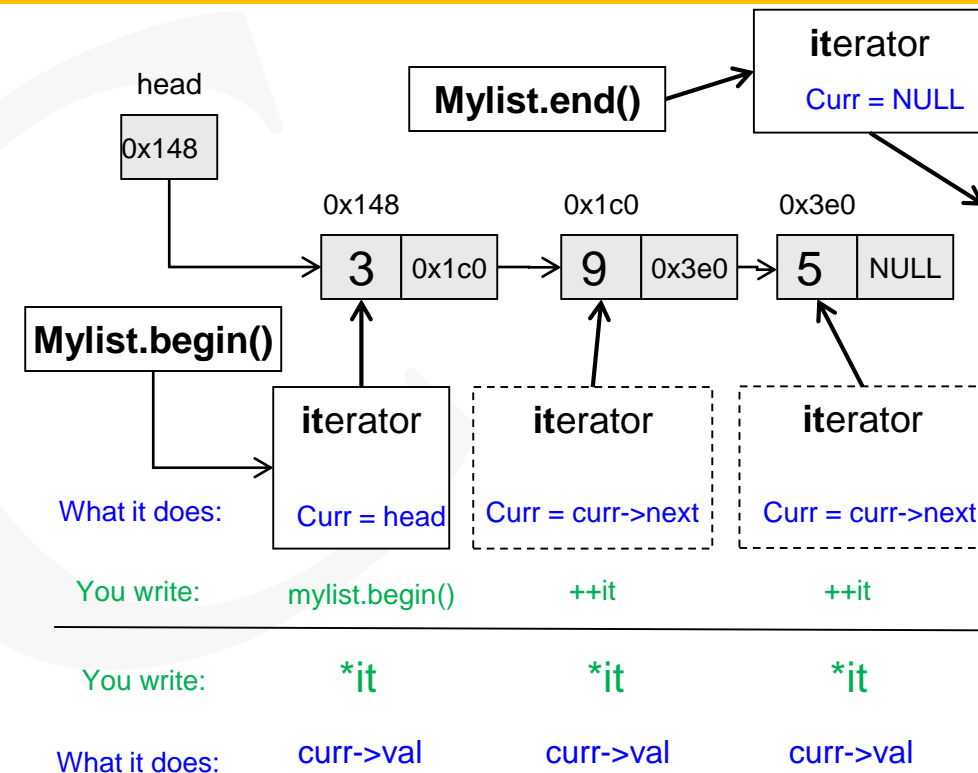
```
vector<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.at(i) << endl;
}
```

```
LinkedList<int> mylist;
...
for(int i=0; i < mylist.size(); ++i)
{
    cout << mylist.at(i) << endl;
}
```



Iteration: A Better Approach

- **Solution: Don't use get()**
- **Use an iterator**
 - an internal state variable (i.e., another pointer) of the class that moves one step in the list at a time as you iterate
- **Iterator tracks the internal location of each successive item**
- **Iterators provide the semantics of a pointer (they look, smell, and act like a pointer to the values in the list)**
- **Assume**
 - **Mylist.begin()** returns an "iterator" to the beginning item
 - **Mylist.end()** returns an iterator "one-beyond" the last item
 - **++it (preferred) or it++** moves iterator on to the next value



```
LinkedList<int> mylist;
...
iterator it = mylist.begin()
for(it = mylist.begin();
    it != mylist.end();
    ++it)
{
    cout << *it << endl;
}
```

Iterators – Type

- A function like 'get' gives the programmer the appearance that items are located contiguously in memory (like an array) though in implementation they may not
- To iterate over the whole set of items we could use a counter variable and the array indexing ('myvec[i]'), but it can be more efficient (based on how STL is actually implemented) to keep an 'internal' pointer to the next item and update it appropriately
- C++ STL containers define 'helper' classes called iterators that help iterate over each item or find an item in the container

Iterators – Type (cont.)

- Iterators are a new class type defined in the scope of each container
 - Type is *container::iterator* (`vector<int>::iterator` is a type)
- Initialize them with *objname.begin()*, check whether they are finished by comparing with *objname.end()*, and move to the next item with `++` operator

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(i+50);
    }
    vector<int>::iterator it;
    for(it = myvec.begin() ; it != my_vec.end(); ++it){
        ...
    }
}
```

```
// vector.h

template<class T>
class vector
{
    class iterator {

    }

};
```

Iterators – Dereferencing

- Iterator variable has same semantics as a pointer to an item in the container
 - Use * to 'dereference' and get the actual item
 - Since you're storing integers in the vector below, the iterator acts and looks like an int*

```
#include <iostream>
#include <vector>

using namespace std;

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(i+50);
    }

    for(vector<int>::iterator it = my_vec.begin() ; it != my_vec.end(); ++it){
        cout << *it;
    }
    cout << endl;
    return 0;
}
```


Iterator

- Think of an iterator variable as a 'pointer'...when you declare it, it points at nothing
- Think of `begin()` as returning the address of the first item and assigning that to the iterator
- Think of `end()` as returning the address AFTER the last item (i.e., off the end of the collection or maybe NULL) so that as long as the iterator is less than or not equal, you are safe

C++ STL Algorithms

- Many useful functions defined in <algorithm> library
 - <http://www.cplusplus.com/reference/algorithm/sort/>
 - <http://www.cplusplus.com/reference/algorithm/count/>
- All of these functions usually accept iterator(s) to elements in a container

```
#include <iostream>
#include <vector>
#include <cstdlib>
#include <algorithm>
using namespace std;

int main()
{
    vector<int> my_vec(5); // 5 = init. size
    for(int i=0; i < 5; i++){
        my_vec.push_back(rand());
    }
    sort(my_vec.begin(), my_vec.end());
    for(vector<int>::iterator it = my_vec.begin() ; it != my_vec.end(); ++it){
        cout << (*it);
    }
    cout << endl;
    return 0;
}
```

Primary data types

ABSTRACT DATA TYPES

Abstract Data Types

- **An ADT is a:**
 - **Specification/model for a group of values/data and the operations on those values**
- **The model allows us to separate...**
 - **The decision of what data structure to use and how it will be used in our higher level application**
 - **And the implementation of the specific data structure**
- **A data structure is:**
 - **An implementation of an ADT in a given programming language**
- **Each ADT we examine in this course has certain:**
 - **Well defined operations and capabilities that are often useful**
 - **Time & space advantages**
 - **Time & space disadvantages**

4 Popular ADTs

- **List**
- **Dictionary/Map**
- **Set**
- **Priority Queue**

USC

Lists

- **Ordered collection of items, which may contain duplicate values, usually accessed based on their position (index)**
 - Ordered = Each item has an index and there is a front and back (start and end)
 - Duplicates allowed (i.e., in a list of integers, the value 0 could appear multiple times)
 - Accessed based on their position (`list[0]`, `list[1]`, etc.)
- **What are some operations you perform on a list?**

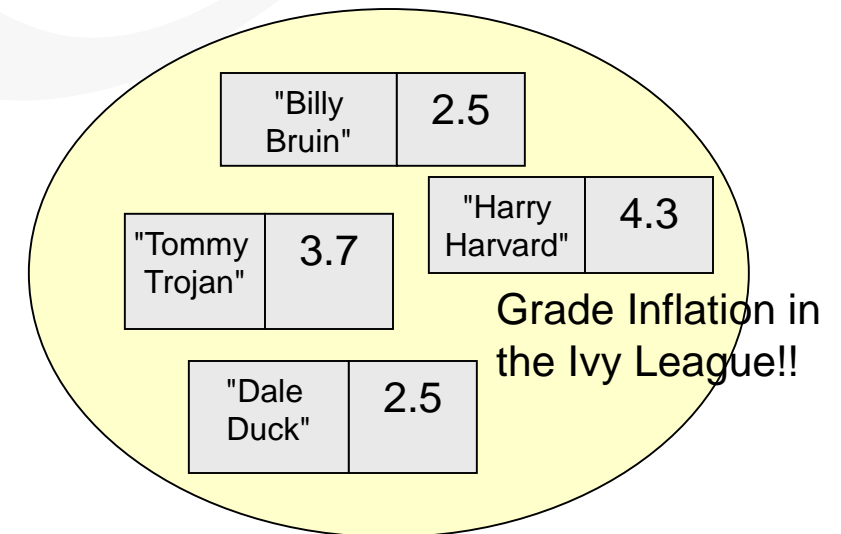


List Operations

Operation	Description	Input(s)	Output(s)
insert	Add a new value at a particular location shifting others back	Index : int Value	
remove	Remove value at the given location	Index : int	Value at location
get / at	Get value at given location	Index : int	Value at location
set	Changes the value at a given location	Index : int Value	
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int
push_back / append	Add a new value to the end of the list	Value	
find	Return the location of a given value	Value	Int : Index

Maps / Dictionaries

- Stores key,value pairs
 - Example: Map student names to their GPA
- Keys must be unique (can only occur once in the structure)
- No constraints on the values
- What operations do you perform on a map/dictionary?
- No inherent ordering between key,value pairs
 - Can't ask for the 0th item...

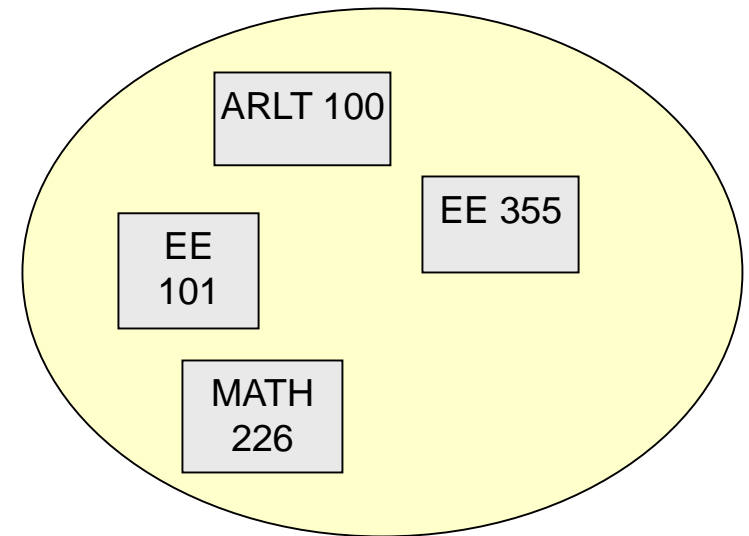


Map / Dictionary Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key,value pair to the dictionary (assuming its not there already)	Key, Value	
Remove	Remove the key,value pair with the given key	Key	
Get / lookup	Lookup the value associated with the given key or indicate the key,value pair doesn't exist	Key	Value associated with the key
In / Find	Check if the given key is present in the map	Key	bool (or ptr to pair/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		int

Set

- A set is a dictionary where we only store keys (no associated values)
 - Example: All the courses taught at USC (ARLT 100, ..., EE 454, MATH 226, ...)
- Items (a.k.a. Keys) must be unique
 - No duplicate keys (only one occurrence)
- Not accessed based on index but on value
 - We wouldn't say, "What is the 0th course at USC?"
- What operations do we perform on a set?



Set Operations

Operation	Description	Input(s)	Output(s)
Insert / add	Add a new key to the set (assuming its not there already)	Key	
Remove	Remove	Key	
In / Find	Check if the given key is present in the map	Key	bool (or ptr to item/NULL)
empty	Returns true if there are no values in the list		bool
size	Returns the number of values in the list		Int
intersection	Returns a new set with the common elements of the two input sets	Set1, Set2	New set with all elements that appear in both set1 and set2
union	Returns a new set with all the items that appear in either set	Set1, Set2	New set with all elements that appear in either set1 and set2
difference	Returns a set with all items that are just in set1 but not set2	Set1, Set2	New set with only the items in set1 that are not in set2

What's Your ADT?

- Scores on a test
 - Students in a class
 - Courses & their enrollment
 - Temperature Reading at a location
 - Usernames and password
 - Index in a textbook
 - Facebook friends
 - Adjacent countries of a map
- List
 - Set (maybe List)
 - Map (Key = course, Value = enrollment)
 - List
 - Map
 - Map
 - Set
 - Set

Some Implementation Details

List

- An array acts as a list
- Index provides ordering
 - First at location 0
 - Last at location $n-1$

0	1	2	3	4	5	6	7	8	9	10	11
30	51	30	53	30	10						

Set

- Can use an array
- Must check for duplicate on insertion
 - $O(n)$ solution
- Can we do better? Yes...

0	1	2	3	4	5	6	7	8	9	10	11
30	51	53	10								

Map

- Can also use an array
- Again check for duplicate key on insertion

```
struct Pair{
    string key;
    double value;;
};
```

0	1	2	3
"Tommy" 3.7	"Billy" 2.5	"Harry" 4.3	

Maps (a.k.a. Dictionaries or Ordered Hashes)

ASSOCIATIVE CONTAINERS

Student Class

```
class Student {  
public:  
    Student();  
    Student(string myname, int myid);  
    ~Student();  
    string get_name() { return name; } // get their name  
    void add_grade(int score); // add a grade to their grade list  
    int get_grade(int index); // get their i-th grade  
private:  
    string name;  
    int id;  
    vector<int> grades;  
};
```

Creating a List of Students

- How should I store multiple students?
 - Array, Vector, Deque?
- It depends on what we want to do with the student objects and *HOW we want to access them*
 - Just iterate over all students all the time (i.e., add a test score for all students, find test average over all students, etc.) then use an array or vector
 - If we want to access random (individual) students a lot, this will require searching and finding them in the array/vector...computationally expensive!
 - $O(n)$ [linear search] or $O(\log n)$ [binary search] to find student or test membership

```
#include <vector>
#include "student.h"
using namespace std;

int main()
{
    vector<student> slist1;
    ...

    unsigned int i;

    // compute average of 0-th score
    double avg = 0;
    for(i=0; i< slist1.size(); i++){
        avg += slist1[i].get_score(0);
    }
    avg = avg / slist1.size();

    // check "Tommy"s score
    int tommy_score= -1;
    for(i=0; i < slist1.size(); i++){
        if(slist1[i].get_name() == "Tommy"){
            tommy_score = slist1[i].get_score(2);
            break;
        }
    }
    cout<< "Tommy's score is: " <<
        tommy_score << endl;
}
```


Index and Data Relationships

- Arrays and vectors are indexed with integers 0...N-1 and have no relation to the data
- Could we somehow index our data with meaningful values
 - `slist1["Tommy"].get_score(2)`
- YES!!! Associative Containers

```
#include <vector>
#include "student.h"
using namespace std;

int main()
{
    vector<student> slist1;
    ...

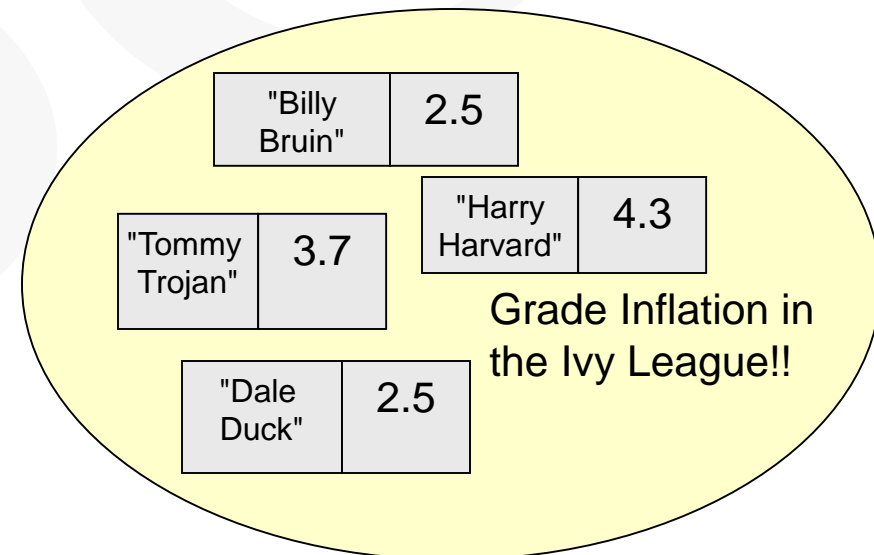
    unsigned int i;

    // compute average of 0-th score
    double avg = 0;
    for(i=0; i< slist1.size(); i++){
        avg += slist1[i].get_score(0);
    }
    avg = avg / slist1.size();

    // check "Tommy"s score
    int tommy_score= -1;
    for(i=0; i < slist1.size(); i++){
        if(slist1[i].get_name() == "Tommy"){
            tommy_score = slist1[i].get_score(2);
            break;
        }
    }
    cout<< "Tommy's score is: " <<
        tommy_score << endl;
}
```

Maps / Dictionaries

- Stores key,value pairs
 - Example: Map student names to their GPA
- Keys must be unique (can only occur once in the structure)
- No constraints on the values
- No inherent ordering between key,value pairs
 - Can't ask for the 0th item...
- Operations:
 - Insert
 - Remove
 - Find/Lookup



C++ Pair Struct/Class

- C++ library defines a struct 'pair' that is templated to hold two values (first and second) of different types
 - Types are provided by the template
- C++ map class internally stores its key/values in these *pair* objects
- Defined in 'utility' header but if you `#include <map>` you don't have to include utility
- Can declare a pair as seen in option 1 or call library function "make_pair" to do it

```
template <class T1, class T2>
struct pair {
    T1 first;
    T2 second;
}
```

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

void func_with_pair_arg(pair<char,double> p)
{    cout << p.first << " " << p.second <<endl; }

int main()
{
    string mystr = "Bill";
    pair<string, int> p1(mystr, 1);
    cout << p1.first << " " << p1.second <<endl;

    // Option 1: Anonymous pair constructed and passed
    func_with_pair_arg( pair<char,double>('c', 2.3) );

    // Option 2: Same thing as above but w/ less typing
    func_with_pair_arg( make_pair('c', 2.3) );
}
```

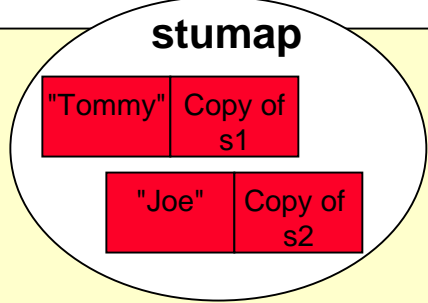
Bill 1

c 2.3

c 2.3

Associative Containers

- C++ STL 'map' class can be used for this purpose
- Maps store (key,value) pairs where:
 - key = index/label to access the associated value
 - Stored value is a copy of actual data
- Other languages refer to these as 'hashes' or 'dictionaries'
- **Keys must be unique**
 - **Just as indexes were unique in an array or list**
- Value type should have a default constructor [i.e. Student()]
- Key type must have less-than (<) operator defined for it
 - For strings, use C++ string rather than char array since it has a < operator defined that will look at the string value (if you use char* it will just compare the actual pointer)
- Efficient at finding specified key/value and testing membership ($O(\log_2 n)$)



```

#include <map>
#include "student.h"
using namespace std;

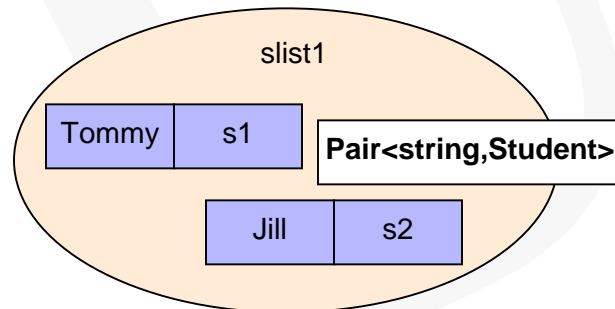
int main()
{
    map<string,Student> stumap;
    Student s1("Tommy",86328);
    Student s2("Joe",54982);
    ...
    // Option 1: this will insert an object
    stumap[ "Tommy" ] = s1;
    // Option 2: using insert()
    stumap.insert( pair<string,Student>("Joe", s2) );
    // or stumap.insert( make_pair("Joe", s2) );
    ...
    int tommy_score=stumap[ "Tommy" ].get_score(1);
    Returns 'Copy of s1' and then you can call Student
    member functions
    stumap.erase( "Joe" );
    cout << "Joe dropped the course..Erased!";
    cout << endl;
}

```

slis1 is a map that associates C++ strings (keys) with Student objects (values)

Maps & Iterators

- Can still iterate over all elements in the map using an iterator object for that map type
- Iterator is a “pointer”/iterator to a pair struct
 - it->first is the key
 - it->second is the value



```
#include <map>
#include "student.h"
using namespace std;

int main()
{
    map<string, student> slist1;
    Student s1("Tommy", 86328);
    Student s2("Jill", 54982);
    ...
    slist1["Tommy"] = s1;
    slist1[s1.get_name()].add_grade(85);

    slist1["Jill"] = s2;
    slist1["Jill"].add_grade(93);
    ...

    map<string, student>::iterator it;
    for(it = slist1.begin(); it != slist1.end(); ++it) {
        cout << "Name/key is " << it->first;
        cout << " and their 0th score is ";
        cout << (it->second).get_score(0);
    }
}
```

Name/key is Tommy and their 0th score is 85
Name/key is Jill and their 0th score is 93

Map Membership [Find()]

- Check/search whether key is in the map object using *find()* function
- Pass a key as an argument
- Find returns an iterator
- If key is IN the map
 - Returns an iterator/pointer to that (key,value) pair
- If key is NOT IN the map
 - Returns an iterator equal to *end()*'s return value

```
#include <map>
#include "student.h"
using namespace std;

int main()
{
    map<string, student> slist1;
    Student s1("Tommy", 86328);
    Student s2("Bill", 14259);

    slist1["Tommy"] = s1;    // Insert an item
    slist1["Tommy"].add_grade(85);  // Access it

    if(slist1.find("Bill") != slist1.end() ){
        cout << "Bill exists!" << endl;
    }
    else {
        cout << "Bill does not exist" << endl;
        slist1["Bill"] = s2;  // So now add him
    }

    map<string, student>::iterator it = slist1.find(name);
    if( it != slist1.end() ){
        cout << it->first << " got score=" <<
            it->second.get_grade(0) << endl;
    }
}
```

Another User of Maps: Sparse Arrays

- **Sparse Array:** One where there is a large range of possible indices but only small fraction will be used (e.g., are non-zero, etc.)
- **Example 1:** Using student IDs to represent students in a course (large 10-digit range, but only 30-40 used)
- **Example 2:** Count occurrences of zip codes in a user database
 - **Option 1:** Declare an array of 100,000 elements (00000-99999)
 - Wasteful!!
 - **Option 2:** Use a map
 - Key = zipcode, Value = occurrences



Set Class

- C++ STL "set" class is like a list but each value can appear just once
- Think of it as a map that stores just keys (no associated value)
- **Keys are unique**
- **insert()** to add a key to the set
- **erase()** to remove a key from the set
- Very efficient at testing membership ($O(\log_2 n)$)
 - Is a specific key in the set or not!
- Key type must have a less-than (<) operator defined for it
 - Use C++ string rather than char array
- Iterators to iterate over all elements in the set
- **find()** to test membership

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

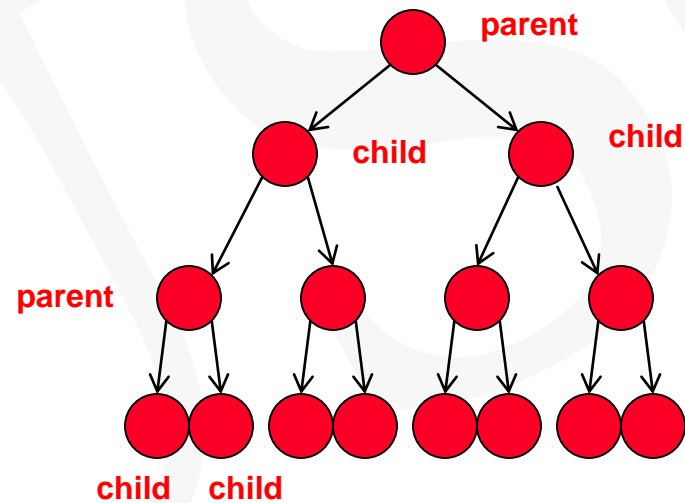
int main()
{
    set<string> people;

    people.insert("Tommy");
    string myname = "Jill";
    people.insert(myname);
    people.insert("Johnny");

    for(set<string>::iterator it=people.begin();
        it != people.end();
        ++it){
        cout << "Person: " << *it << endl;
    }
    myname = "Tommy";
    if(people.find(myname) != people.end()){
        cout<< "Tommy is an EE major!" << endl;
    }
    else {
        cout<< "Tommy is the wrong major!" << endl;
    }
    myname = "Johnny";
    people.erase("Johnny");
}
```


A Deeper Look: Binary Tree

- Data structure where each node has at most 2 children (no loops/cycles in the graph) and at most one parent
- Tree nodes w/o children are called "leaf" nodes
- Depth of binary tree storing N elements? _____



Binary Search Tree

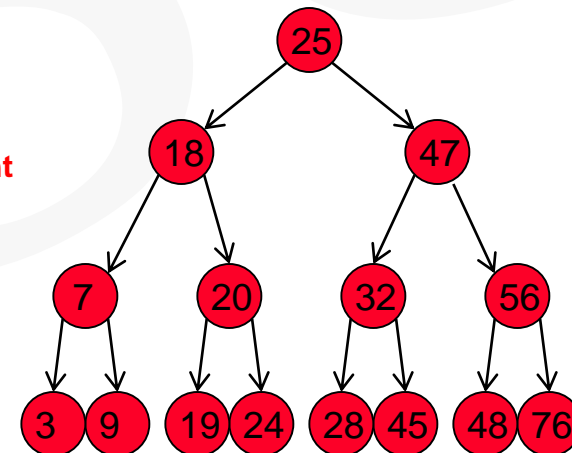
- Tree where all nodes meet the property that:
 - All descendants on the left are less than the parent's value
 - All descendants on the right are greater than the parent's value
- Can find value (or determine it doesn't exist) in $\log_2 n$ time by doing binary search

Q: is 19 in the tree?

19 < 25..left

19 > 18..right

19 < 20..left



Q: is 34 in the tree?

34 > 25..right

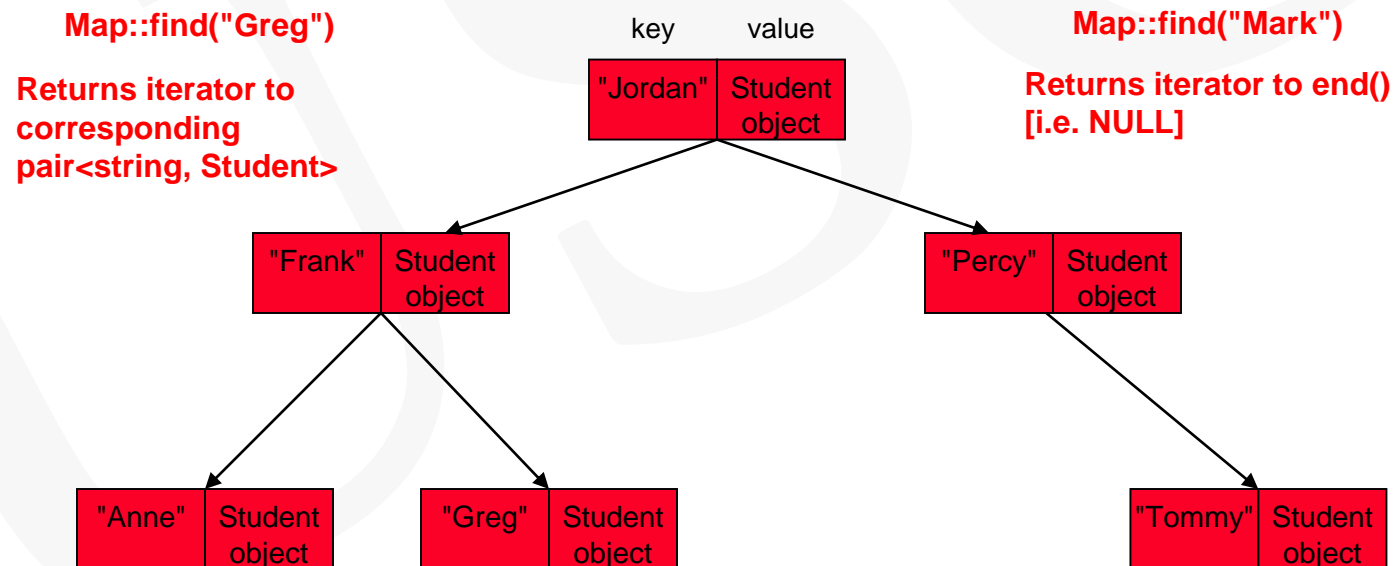
34 < 47..left

34 > 32..right

34 != 45 and we're at
a leaf node...34 does
not exist

Trees & Maps/Sets

- Maps and sets use binary trees internally to store the keys
- This allows logarithmic find/membership test time
- This is why the less-than (<) operator needs to be defined for the data type of the key



Exercise

- Practice using iterators and a map
- `$ tar xvf zipmap.tar`
- Edit `map_zip.cpp` and view the data file `zipcodes.txt`
 - We have written code to read in all the zipcodes in `zipcodes.txt` into a vector `'all_zips'`
 - Iterate through the zipcodes in `'all_zips'` using an iterator
 - Create (insert the first time, increment remaining times) a map that stores zip codes as the keys and the number of occurrences as the value
 - The code at the bottom will iterate through your map and print out your results so you can check your work
- **Bonus: Lookup the `std::algorithm` library**
 - Look at `'count()'` and `'sort()'` and see if you can use them to sort the `all_zips` vector data or count the occurrences of a particular zip code in `all_zips` (rather than using the map)