# **`map`** idea

- Stores a collection of key-value pairs
- Keys are unique
- Can look up an element by its key
- Can remove an element given its key
- When you iterate over a map key-value pairs are visited in order from smallest to largest key.
- Need to have **`<`** defined for key type.
  - does equality as **`!(a < b) && !(b < a)`**
  - does not use **`==`**

# **map** syntax

- Two template parameters: <keytype, valuetype>
  - (Note: STL uses name value_type to refer to both together)
- collection of students and their scores:

```
#include <map>
typedef map<string, int> Students;
Students students;

students[“Joe”] = 23;   // inserts if not there

cout << students[“Joe”];  // looks up joe’s score

students[“Joe”] = 5;   // looks up and updates
```

# **`map`** vs. **`vector`**

- map is sometimes called an *associative array*

```
cout << students["Joe"];
```

- array index syntax, but it's not random access
- Even **`int`** keys can be useful, e.g., for a sparse set of data.
  - Example: storing a polynomial

$x^{12} + 4x^3 + 10$

(12,1) (3,4) (0,10)

poly[expon] = coeff;

# **`map`** internal representation

- uses a balanced search tree (red-black tree)
- What is the complexity of the following operations:
  - **`m.find(key)`** (lookup – returns iterator)
  - **`m[key]`** (lookup / insert)
  - **`m.erase(key)`**
  - visit all elements in order by key (iterator)
- Category of container is called *sorted associative containers*, because the tree maintains the data in sorted order

# Iterating over a **`map`**

- Each element consists of two parts: key and value
- dereferenced iterator is a **`pair`** object.
- **`pair<T1,T2>`** glues two objects together.
- **`pair`** has two public fields: **`first`** and **`second`**
- here **`first`** is the key and **`second`** is its associated value

```
// print out students and scores sorted by name
   Students::const_iterator iter;
   for (iter=students.begin();
        iter != students.end(); iter++) {
      cout << (*iter).first << " ";
      cout << (*iter).second << endl;
   }
```

- Or can use **`iter->first`** **`iter->second`**

# Non-mutable **`map`** keys

- Can't modify the key once it's in a map because it's used to determine element's location in data structure. (Unlike sequence containers.) E.g.:

```
Students studs;  // maps strings to ints
studs["joe"] = 24;
Students::iterator firstOne =studs.begin();
firstOne->second = 55;
          // ok to change the value
firstOne->first = "Sam";
          // not ok to change the key
*firstOne =
       pair<string, int>("Sally", 100); // not ok
```

# `set` class

- Like a `map`, but we only have keys (and no values to go with them).
- Does not use array indexing syntax.
- Also a sorted associative container using a balanced tree representation.
- Keys are unique (i.e., like a mathematical notion of set)
- STL algorithms for set operations, e.g., union, intersection

# **set** example

- E.g., store a set of words.

```
set<string> words;
words.insert(“foo”);
words.insert(“bar”);
words.insert(“blob”);
words.insert(“foo”);  // does NO insert
if (words.find(“joe”)!=words.end()) {
  ...// is present
```

# iterating over a **`set`**

- Each element is a single object (not a pair).
- Elements are visited in increasing order
- cannot assign to **`*iter`**: like with **`map`**, it determines ordering in underlying data structure.

```
set<string>::const_iterator iter;
for (iter = words.begin();
     iter != words.end(); iter++) {
  cout << *iter << " ";
}
```