Advanced Data Structures and Algorithms

CS 361 – Fall 2013

Lec. #08: Associative Containers

Tamer Nadeem
Dept. of Computer Science
Class Objective/Overview

• Familiarize with *Associative Containers*

• Understand *Set, MultiSet, Map, and MultiMap Containers*

• Familiarize with *miniSet and miniMap Implementation*

• Understand *Hashing and Hash Tables*

• Familiarize with *Balanced Search Trees - AVL*

• Familiarize with *Balanced Search Trees – B-Tree*

• Understand *Red-Black Trees*
Associative Containers
std Containers

• We call an ADT a “container” if its main purpose is to provide a collection of items of some other, simpler types.

• Types of Containers

  • **Sequence Containers**: data are stored and accessed by position in linear order.

  • **Associative Containers**: elements are stored and accessed by a key such as name or number that has no relation to the location of the element (i.e., random).

  • **Adapter Containers**: contains another container as underlying structure with a restricted set of operations.

<table>
<thead>
<tr>
<th>Sequence Containers</th>
<th>Adapter Containers</th>
<th>Associative Containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector</td>
<td>Stack</td>
<td>Set, Multiset</td>
</tr>
<tr>
<td>Deque</td>
<td>Queue</td>
<td>Map, Multimap</td>
</tr>
<tr>
<td>List</td>
<td>Priority Queue</td>
<td></td>
</tr>
</tbody>
</table>
Associative Containers

• Most of the data structures we have looked at so far are sometime called *sequential containers* in which elements are maintained in a known *sequence*.

• Access elements by indicating a *position* in the sequence
  • *Numerically*: e.g., [23]
  • *Symbolically*: e.g., myVector.front()

• We switch our attention to *associative containers*, elements maintained in a sequence that allows *rapid access* to elements based upon their *value*.

• The major associative classes in the standard library are
  • set, map, multiset and multimap.

• Implementation of sequential containers typically is more *difficult* than sequential container.
Sets and Maps
Overview of Sets and Maps

• **Sets** are containers to which we can add elements (called “**keys**”)
  • Use to check to see if certain key values are **present** in the set.

• Maps, also known in other contexts as “**lookup tables**” or “**dictionaries**”, allow us to store **pairs** consisting of a **key** value and associated **data** value.
  • Use to look up the data value, if any, associated with a given key.
  • In some contexts, especially more mathematical ones, the set of keys is called the **domain** of the map and the set of associated data values is called the **range** of the map.
Overview of Sets and Maps

• In a set or map, a given key value may appear only once
  • Adding a key $K$ to a set replaces any existing key equal to $K$.
  • Adding a key-data $(K, D_1)$ to a map that has $(K, D_2)$ replaces $(K, D_2)$ with $(K, D_1)$

• In a multiset or multimap, the same key can occur any number of times.
  • For a multiset we can now ask “how many $K$’s are in this set?
  • For a multimap, adding a key-data pair $(K, D_1)$ to a multimap that already has $(K, D_2)$ results in multimap that has both $(K, D_1)$ and $(K, D_2)$.

intSet: Set of ints

degreeMajor: Map of string-int pairs

<table>
<thead>
<tr>
<th>Degree Major</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>English</td>
<td>117</td>
</tr>
<tr>
<td>Economics</td>
<td>75</td>
</tr>
<tr>
<td>Computer Science</td>
<td>240</td>
</tr>
<tr>
<td>Business</td>
<td>290</td>
</tr>
<tr>
<td>Music</td>
<td>40</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>intSet</th>
<th>18</th>
<th>54</th>
<th>12</th>
<th>67</th>
<th>45</th>
<th>89</th>
</tr>
</thead>
</table>
The “set” ADT

• Constructors:

  set();
  Create an empty set. This is the Default Constructor.

  set(T *first, T *last);
  Initialize the set by using the address range [first, last).

• Operations:

  bool set() const;
  Is the set empty?

  int size() const;
  Return the number of elements in the set.

  int count(const T& key) const;
  Search for key in the set and return 1 if it is in the set and 0 otherwise.
The “set” ADT

• **Iterators:**
  
  iterator `find`(const T& key);
  
  Search for key in the set and return an iterator pointing at it, or
  `end()` if it is not found.

  `Const_iterator find`(const T& key) const;
  
  Constant version.

  iterator `begin`();
  
  Return an iterator pointing at the first member in the set.

  `const_iterator begin`(const);
  
  Constant version of `begin()`.
The “set” ADT

• **Iterators:**

  iterator end();
  
  Return an iterator pointing just past the last member in the set.

  const_iterator end() const;
  
  Constant version of end().

• **Update Operations:**

  pair<iterator, bool> insert(const T& key);
  
  If key is not in the set, insert it and then return a pair whose first element is an iterator pointing to the new element and whose second element is true. Otherwise, return a pair whose first element is an iterator pointing at the existing element and whose second element is false.

  Postcondition: The set size increases by 1 if key is not in the set.
The “set” ADT

• Update Operations:

  int erase(const T& key);
  If key is in the set, erase it and return 1; otherwise, return 0.
  Postcondition: The set size decreases by 1 if key is in the set.

  void erase(iterator pos);
  Erase the item pointed to by pos.
  Preconditions: The set is not empty, and pos points to a valid
  set element.
  Postcondition: The set size decreases by 1.

  void erase(iterator first, iterator last);
  Erase the elements in the range [first, last).
  Precondition: The set is not empty.
  Postcondition: The set size decreases by the number of elements
  in the range.
The “set” ADT

• The interface to multiset is identical to that of the set (aside from replacing the name “set” by “multiset”)
  • It’s only the behavior of a few of the operations that differ.
  • So, as we look at the set interface, keep in mind that the same things apply to multiset s as well.
Implementation of STL “set”

- STL uses a red-black search tree.
- A red-black tree is a binary search tree that maintains balance between left and right subtrees of a node.
- The corresponding running time for red-black tree is always \( O(\log_2 n) \)
- STL implements ordered associated containers.
void spellChecker(string& filename)
{
    // sets storing the dictionary and the misspelled words
    set<string> dictionary, misspelledWords;

    // dictionary and document streams
    ifstream dict, doc;
    string word;
    char response;

    // open "dict.dat"
    dict.open("dict.dat");
    if (!dict)
    {
        cerr << "Cannot open \"dict.dat\"" << endl;
        exit(1);
    }

    // open the document file
    doc.open(filename.c_str());
    if (!doc)
    {
        cerr << "Cannot open " << filename << endl;
        exit(1);
    }
while(true) { // insert each word from the file "dict.dat" into the dictionary set
dict >> word;
if (!dict) break;
dictionary.insert(word); // insert into the dictionary
}

while(true) { // read the document word by word and check spelling
doc >> word;
if (!doc) break;

// lookup word up in the dictionary. if not present
// assume word is misspelled. prompt user to add or ignore word
if (dictionary.find(word) == dictionary.end()) {
    cout << word << endl;
    cout << " 'a' (add) 'i' (ignore) 'm' (misspelled) " ;
cin >> response;
    // if response is 'a' add to dictionary; otherwise add to the set of misspelled words
    if (response == 'a') dictionary.insert(word);
    else if (response == 'm') misspelledWords.insert(word);
}
}
cout << endl << "Set of misspelled words" << endl; // display the set of misspelled words
writeContainer(misspelledWords.begin(), misspelledWords.end());
cout << endl;
Set Example 2 - Sieve of Eratosthenes

- The method is used to find all prime numbers less than or equal to an integer value \( n \).
- The method passes over the set of \( n \) numbers \( m \) iterations \((m = \sqrt{n})\).
- Each each \( m \) iteration, you remove the multiples of \( m \) from the set.
- Time complexity of the algorithm is \( O(n\sqrt{n}) \).

Pass \( m = 2 \):
remove all multiples of 2

Pass \( m = 3 \):
remove all multiples of 3 still in the set

Pass \( m = 5 \):
remove all multiples of 5 still in the set

7, 11, 13, 17, 19, and 23 contain no multiples in the range 2 to 25
Primes \( \{2, 3, 5, 7, 11, 13, 17, 19, 23\} \)
Set Operations

• Assuming two sets: \( A = \{1, 3, 8, 9, 10\} \) and \( B = \{2, 3, 6, 9\} \)

• **Set-Union Operator**+ (\( A + B \)):
The set of all elements \( x \) such that \( x \) is an element in set \( A \) **OR** \( x \) is an element in set \( B \).
Example: \( A + B = \{1, 2, 3, 6, 8, 9, 10\} \)

• **Set-Intersection Operator*** (\( A * B \)):
The set of all elements \( x \) such that \( x \) is an element in set \( A \) **AND** \( x \) is an element in set \( B \).
Example: \( A * B = \{3, 9\} \)

• **Set-Difference Operator**- (\( A - B \)):
The set of all elements \( x \) such that \( x \) is an element in set \( A \) **BUT** \( x \) is **NOT** an element in set \( B \).
Example: \( A - B = \{1, 8, 10\} \)
Implementing Set Intersection

- The implementation uses iterators to scan each of the ordered sets.
- Make a pairwise comparison of the elements, looking for a match that identifies elements of the intersection.
Maps and Multimaps

A map can be thought of as:

• a lookup table, or as

• a generalization of an array/vector in which the index need not be numeric.

• A map associates a key type `Key` with an associated data type `T`. In the example above, the key type is `string`, and the associated data is `int`.

• The map allows you to store `(key, data)` pairs (e.g., the two assignment statements in the example above), and to retrieve the data previously stored with some key (e.g., the `cout` output statement).

• Like sets, a map may only contain a single copy of any given Key value

  • But a `multimap` can contain multiple copies of the same key.

```
typedef Zipcodes int;
map<string, Zipcodes> zips;
zips [ "ODU" ] = 23529;
zips [ "Sentara" ] = 23452;
cout << zips [ "ODU" ] ;
```
The “map” ADT

• Operations:

  int count(const T& item) const;
  Return the number of duplicate occurrences of item in the multiset.

  pair<iterator, iterator> equal_range(const T& item);
  Return a pair of iterators such that all occurrences of item are in the iterator range [first member of pair, second member of pair).

  iterator insert(const T& item);
  Insert item into the multiset and return an iterator pointing at the new element.
  Postcondition: The element item is added to the multiset.
The “map” ADT

• Operations:

    int erase(const T& item);

        Erase all occurrences of item from the multiset and return the number of items erased.

        Postcondition: The size of the multiset is reduced by the number of occurrences of item in the multiset.
int main() {
    // a map<string, time24> object whose entries are student names and total hours worked during a week
    map<string, time24> studentWorker;
    map<string, time24>::iterator iter; // map iterator

    ifstream fin; // object used to input the data from file "studwk.dat"
    string studName;
    time24 workTime;

    fin.open("studwk.dat"); // open the file "studwk.dat"

    // input successive lines in the file consisting of the student name and the scheduled work time
    while (true) {
        fin >> studName;
        if (!fin) break;
        fin >> workTime;

        // add a new student with workTime as time worked or update the
        // accumulated work time if the student is already in the map
        studentWorker[studName] += workTime;
    }

    writeMap(studentWorker, 
    // output the map, one key-value pair per line
    return 0;
}
miniSet and minMap Implementations
miniSet

```cpp
#ifndef MINISET_CLASS
#define MINISET_CLASS

#include <utility>
#include "d_stree.h" // stree class

using namespace std;

// implements a set which does not contain duplicate data values
template <typename T>
class miniSet
{
    public:

    typedef typename stree<T>::iterator iterator;
typedef typename stree<T>::const_iterator const_iterator;
    // miniSet iterators are simply stree iterators

    miniSet(); // default constructor

    miniSet(T *first, T *last);
        // build a set whose data are determined by pointer range
        // [first, last)
```
miniset

bool empty() const;
  // is the set empty?

int size() const;
  // return the number of elements in the set

iterator find (const T& item);
  // search for item in the set and return an iterator
  // pointing at it, or end() if it is not found

const_iterator find (const T& item) const;
  // constant version

pair<iterator,bool> insert(const T& item);
  // if item is not in the set, insert it and return a pair
  // whose first element is an iterator pointing to the
  // new element and whose second element is true.
  // otherwise, return a pair whose first element is an
  // iterator pointing at the existing element and whose
  // second element is false
  // Postcondition: the set size increases by 1 if item is
  // not in the set
miniSet

int erase(const T& item);
   // if item is in the set, erase it and return 1;
   // otherwise, return 0
   // Postcondition: the set size decreases by 1 if item is
   // in the set

void erase(iterator pos);
   // erase the item pointed to by pos.
   // Preconditions: the set is not empty and pos points
   // to an item in the set. if the set is empty, the
   // function throws the underflowError exception. if the
   // iterator is invalid, the function throws the referenceError
   // exception.
   // Postcondition: the set size decreases by 1

void erase(iterator first, iterator last);
   // erase the elements in the range [first, last)
   // Precondition: the set is not empty. if the set is empty,
   // the function throws the underflowError exception.
   // Postcondition: the set size decreases by the number
   // elements in the range

iterator begin();
   // return an iterator pointing at the first member
   // in the set
miniSet

const_iterator begin() const;
    // constant version of begin()

iterator end();
    // return an iterator pointing just past the last
    // member in the set

const_iterator end() const;
    // constant version of end()

private:
    // set implemented using a binary search tree
    stree<T> t;

};

// CONSTRUCTORS

template<typename T>
miniSet<T>::miniSet()
{
}

template<typename T>
miniSet<T>::miniSet(T *first, T *last): t(first, last)
{
}
miniSet

```cpp
template <typename T>
bool miniSet<T>::empty() const
{
    return t.empty();
}

template <typename T>
int miniSet<T>::size() const
{
    return t.size();
}

template <typename T>
miniSet<T>::iterator miniSet<T>::find (const T& item)
{
    // return stree iterator which is a miniSet iterator
    return t.find(item);
}

template <typename T>
miniSet<T>::const_iterator miniSet<T>::find (const T& item) const
{
    // return stree iterator which is a miniSet iterator
    return t.find(item);
}
```
miniSet

template <typename T>
pair<miniSet<T>::iterator, bool> miniSet<T>::insert(const T& item)
{
    // insert item into the binary search tree and return
    // the iterator-bool pair
    return t.insert(item);
}

template <typename T>
int miniSet<T>::erase(const T& item)
{
    // erase item from the tree
    return t.erase(item);
}

template <typename T>
void miniSet<T>::erase(iterator pos)
{
    if (t.size() == 0)
        throw underflowError("miniSet erase(): set is empty");
    if (pos == end())
        throw referenceError("miniSet erase(): invalid iterator");
    // erase the item in the tree pointed to by pos
    t.erase(pos);
}
miniSet

template <typename T>
void miniSet<T>::erase(iterator first, iterator last)
{
    if (t.size() == 0)
        throw underflowError("miniSet erase(): set is empty");
    // erase range [first, last) in the tree
    t.erase(first, last);
}
template <typename T>
miniSet<T>::iterator miniSet<T>::begin()
{
    // a miniSet iterator is an stree iterator
    return t.begin();
}
template <typename T>
miniSet<T>::const_iterator miniSet<T>::begin() const
{
    return t.begin();
}
template <typename T>
miniSet<T>::iterator miniSet<T>::end()
{
    return t.end();
}
miniSet

template <typename T>
miniSet<T>::const_iterator miniSet<T>::end() const
{
    return t.end();
}

// determine if sets lhs and rhs have the same size and
// are equal element by element
template <typename T>
bool operator==(const miniSet<T>& lhs, const miniSet<T>& rhs);

// return a miniSet object containing all elements that
// are in lhs or rhs
template <typename T>
miniSet<T> operator+ (const miniSet<T>& lhs, const miniSet<T>& rhs);

// return a miniSet object containing all elements that
// are in goth lhs and rhs
template <typename T>
miniSet<T> operator* (const miniSet<T>& lhs, const miniSet<T>& rhs);

// return a miniSet object containing all elements that
// are in lhs but not in rhs
template <typename T>
miniSet<T> operator- (const miniSet<T>& lhs, const miniSet<T>& rhs);
// SET FUNCTION IMPLEMENTATIONS

template <typename T>
bool operator==(const miniSet<T>& lhs, const miniSet<T>& rhs)
{
    miniSet<T>::const_iterator myself = lhs.begin(), other = rhs.begin();

    // return false if the sets do not have the same size
    if (lhs.size() == rhs.size())
    {
        // compare until encounter end of the sets or
        // find two elements that are not equal
        while (myself != lhs.end() && *myself++ == *other++);

        // if we left the loop before reaching the end
        // of the sets, they are not equal
        if (myself != lhs.end())
            return false;
        else
            return true;
    }
    else
        return false;
}
template <typename T>
miniSet<T> operator+ (const miniSet<T>& lhs, const miniSet<T>& rhs) {
    miniSet<T> setUnion; // construct union
    // iterators that traverse the sets
    miniSet<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();
    // move forward as long as we have not reached the end of either set
    while (lhsIter != lhs.end() && rhsIter != rhs.end())
        if (*lhsIter < *rhsIter)
            // *lhsIter belongs to the union. insert and move iterator forward
            setUnion.insert(*lhsIter++);
        else if (*rhsIter < *lhsIter)
            // *rhsIter belongs to the union. insert and move iterator forward
            setUnion.insert(*rhsIter++);
        else{
            // the two values are equal. insert just one and move both iterators forward
            setUnion.insert(*lhsIter++);
            rhsIter++;
        }
    if (lhsIter != lhs.end()) // flush any remaining items
        while (lhsIter != lhs.end())
            setUnion.insert(*lhsIter++);
    else if (rhsIter != rhs.end())
        while (rhsIter != rhs.end())
            setUnion.insert(*rhsIter++);
    return setUnion;
}
miniSet

template <typename T>
miniSet<T> operator* (const miniSet<T>& lhs, const miniSet<T>& rhs)
{
    // construct intersection
    miniSet<T> setIntersection;
    // iterators that traverse the sets
    miniSet<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();

    // move forward as long as we have not reached the end of either set
    while (lhsIter != lhs.end() && rhsIter != rhs.end())
    {
        if (*lhsIter < *rhsIter)
            // *lhsIter is in lhs and not in rhs. move iterator forward
            lhsIter++;
        else if (*rhsIter < *lhsIter)
            // *rhsIter is in rhs and not in lhs. move iterator forward
            rhsIter++;
        else{
            // the same value is in both sets. insert one value
            // and move the iterators forward
            setIntersection.insert(*lhsIter);
            lhsIter++;
            rhsIter++;
        }
    }
    return setIntersection;
}
template <typename T>
miniSet<T> operator- (const miniSet<T>& lhs, const miniSet<T>& rhs) {
    miniSet<T> setDifference;  // construct difference
    // iterators that traverse the sets
    miniSet<T>::const_iterator lhsIter = lhs.begin(), rhsIter = rhs.begin();
    // move forward as long as we have not reached the end of either set
    while (lhsIter != lhs.end() && rhsIter != rhs.end())
        if (*lhsIter < *rhsIter)  // *lhsIter belongs to lhs but not to rhs. put it in the difference
            setDifference.insert(*lhsIter++);
        else if (*rhsIter < *lhsIter)  // *rhsIter is in the rhs but not in the lhs. Pass over it
            rhsIter++;
        else{  // the same value is in both sets. move the iterators forward
            lhsIter++;
            rhsIter++;
        }
    // flush any remaining items from lhs
    if (lhsIter != lhs.end())
        while (lhsIter != lhs.end())
            setDifference.insert(*lhsIter++);
    return setDifference;
}
#ifndef MINIMAP_CLASS
#define MINIMAP_CLASS

#include "d_pair.h"  // miniPair class
#include "d_stree.h"  // stree class

// implements a map containing key/value pairs.
// a map does not contain multiple copies of the same item.
// types T and Key must have a default constructor
template <typename Key, typename T>
class miniMap
{
    public:

        typedef stree<miniPair<const Key, T>>::iterator iterator;
        typedef stree<miniPair<const Key, T>>::const_iterator const_iterator;
            // miniMap iterators are simply stree iterators. an iterator cannot
            // change the key in a tree node, since the key attribute
            // of a miniPair object in the tree is const

        typedef miniPair<const Key, T> value_type;
            // for programmer convenience

        miniMap();
            // default constructor. create an empty map
miniMap

    miniMap(value_type *first, value_type *last);
    // build a map whose key/value pairs are determined by pointer
    // values [first, last)

    bool empty() const;
    // is the map empty?

    int size() const;
    // return the number of elements in the map

    iterator find (const Key& key);
    // search for item in the map with the given key
    // and return an iterator pointing at it, or end()
    // if it is not found

    const_iterator find (const Key& key) const;
    // constant version of find()

    T& operator[](const Key& key);
    // if no value is associated with key, create a new
    // map entry with the default value T() and return a
    // reference to the default value; otherwise,
    // return a reference to the value already associated
    // with the key
miniMap

int count(const Key& key) const;
   // returns 1 if an element with the key is in the map
   // and 0 otherwise

miniPair<iterator,bool> insert(const value_type& kvpair);
   // if the map does not contain a key/value pair whose
   // key matches that of kvpair, insert a coy of kvpair
   // and return a miniPair object whose first element is an
   // iterator positioned at the new key/value pair and whose second
   // element is true. if the map already contains a key/value
   // pair whose key matches that of kvpair, return a miniPair
   // object whose first element is an iterator positioned at the
   // existing key/value pair and whose second element is false

int erase(const Key& key);
   // erase the key/value pair with the specified key
   // from the map and return the number
   // of items erased (1 or 0)

void erase(iterator pos);
   // erase the map key/value pair pointed by to pos

void erase(iterator first, iterator last);
   // erase the key/value pairs in the range [first, last)
miniMap

iterator begin();
    // return an iterator pointing at the first member
    // in the map
const_iterator begin() const;
    // constant version of begin()

iterator end();
    // return an iterator pointing just past the last
    // member in the map
const_iterator end() const;
    // constant version of end()

private:
    // miniMap implemented using an stree of key-value pairs
    stree<miniPair<const Key, T>> t;
};

template <typename Key, typename T>
miniMap<Key,T>::miniMap()
{}

template <typename Key, typename T>
miniMap<Key,T>::miniMap(value_type *first, value_type *last):
    t(first, last)
{}
miniMap

```cpp
template <typename Key, typename T>
bool miniMap<Key,T>::empty() const
{
    return t.empty();
}
template <typename Key, typename T>
int miniMap<Key,T>::size() const
{
    return t.size();
}
template <typename Key, typename T>
miniMap<Key,T>::iterator miniMap<Key,T>::find (const Key& key)
{
    // pass a miniPair to stree find() that contains key as its
    // first member and T() as its second
    return t.find(value_type(key, T()));
}
template <typename Key, typename T>
miniMap<Key,T>::const_iterator miniMap<Key,T>::find (const Key& key) const
{
    // pass a miniPair to stree find() that contains key as its
    // first member and T() as its second
    return t.find(value_type(key, T()));
}
```
```cpp
template<typename Key, typename T>
T& miniMap<Key,T>::operator[](const Key& key)
{
    // build a miniPair object consisting of key and the default value T()
    value_type tmp(key, T());
    // will point to a pair in the map
    iterator iter;

    // try to insert tmp into the map. the iterator
    // component of the pair returned by t.insert()
    // points at either the newly created key/value
    // pair or a pair already in the map. return a
    // reference to the value in the pair
    iter = t.insert(tmp).first;

    return (*iter).second;
}

template<typename Key, typename T>
int miniMap<Key,T>::count(const Key& key) const
{
    // pass a miniPair to stree count() that contains key as its
    // first member and T() as its second
    return t.count(value_type(key, T()));
}
```
miniMap

template <typename Key, typename T>
miniPair<miniMap<Key,T>::iterator,bool>
miniMap<Key,T>::insert(const miniMap<Key,T>::value_type& kvp)
{
    // t.insert() returns a pair<iterator,bool> object, not a
    // miniPair<iterator,bool> object
    pair<iterator, bool> p = t.insert(kvp);

    // build and return a miniPair<iterator,bool> object
    return miniPair<iterator, bool>(p.first, p.second);
}

template <typename Key, typename T>
int miniMap<Key,T>::erase(const Key& key)
{
    // pass a miniPair to stree erase() that contains key as its
    // first member and T() as its second
    return t.erase(value_type(key, T()));
}

template <typename Key, typename T>
void miniMap<Key,T>::erase(iterator pos)
{
    t.erase(pos);
}
miniMap

```cpp
template <typename Key, typename T>
void miniMap<Key,T>::erase(iterator first, iterator last)
{
    t.erase(first, last);
}
template <typename Key, typename T>
miniMap<Key,T>::iterator miniMap<Key,T>::begin()
{
    return t.begin();
}
template <typename Key, typename T>
miniMap<Key,T>::iterator miniMap<Key,T>::end()
{
    return t.end();
}
template <typename Key, typename T>
miniMap<Key,T>::const_iterator miniMap<Key,T>::begin() const
{
    return t.begin();
}
template <typename Key, typename T>
miniMap<Key,T>::const_iterator miniMap<Key,T>::end() const
{
    return t.end();
}
#endif // MINIMAP_CLASS
```
Hashing and Hash Tables
Hashing

- **Hashing** is an important approach to set/map construction.

- We’ve seen sets and maps with $O(N)$ and $O(\log N)$ search and insert operations.

- **Hash tables** trade off space for speed, sometimes achieving an average case of $O(1)$ search and insert times.

- Hash tables use a *hashing function* to compute an element’s position within the array that holds the table.
Hashing

• If we had a really good hashing function, we could implement set insertion this way:

```cpp
template <class T>
class set {
    ...  
    private:
        const unsigned hSize = <[ . . ] >;
        T table[hSize] ;
};
template <class T>
void set <T>::insert ( const T& key )
{
    unsigned h = hash( key ) ;
    table [h] = key ;
}
```  

• and searching through the table would not be much harder:

```cpp
template <class T>
size_type set <T>::find ( const T& key ) const
{
    int h = hash( key ) ;
    if ( table [h] == key )
        return 1;
    else
        return 0;
}
The Ideal: Perfect Hash Functions

• For overly-simple form of hashing to work, the hash function must
  1. return values in the range 0 . . . hSize-1
  2. be fast and easy to compute
  3. return a unique value for each key.

• A function that satisfies these requirements is called a perfect hash function.

• Perfect hash functions are usually only possible if we know all the keys in advance.
  • Ex: programming languages have a large fixed number of reserved words such as “if” or “while”, a compiler for that language may use a perfect hash function over the language’s keywords to recognize when a word read from the source code file is really a reserved word.
The Reality: Collisions

• For the most part, though, we can’t really expect to have perfect hash functions.
  • Some keys will hash to the same table location.

• Two keys collide if they have the same hash function value.

• Since collisions are, in most cases, unavoidable, we say that a good hash function will
  1. return values in the range 0 . . . hSize-1
  2. be fast and easy to compute
  3. minimize the number of collisions.
The Reality: Collisions

• The first requirement is usually enforced inside the hash table code by the simple technique of taking $\text{hash()} \mod h\text{Size}$

• Unless we have special knowledge about the keys, the best we can say about “minimizing the number of collisions” is that we hope that our hashing function will distribute the keys uniformly
  • If we are drawing keys at random, the probability of the next key’s going into any particular position in the hash table should be the same as for any other position.

• So the characteristics that we’ll look for in a good hash function are
  • Fast and easy to compute
  • Distributes the keys uniformly across the table.

• The possibility of collisions also forces us to revise those simple algorithms to include collision handling.
Hashing Functions – Integers

• This is the easiest possible case.
• If we have a set of integer keys that are already in the range 0 ... hSize-1, we don’t need to do anything:

• If the keys are in a wider range, we can employ the modulus trick:

• The distribution of the original key values is important
  • Assume integer numbers of multiple of 5: 0005, 0010, …, 0100, 0105, …
  • If use hSize = 100 with hashing fn:
    • All numbers map to 20 entries (out of 100) of the hash table
  • use hSize = 101 instead with hashing fn:
    • If hSize is a prime number, it tends to increase the uniformity of the key distribution.

```c
int hash( int i ) { return i; }
```

```c
int hash( int i ) { return i % hSize; }
```

```c
int hash( int i ) { return i % 100; }
```

```c
int hash( int i ) { return i % 101; }
```

<table>
<thead>
<tr>
<th>keys</th>
<th>hash to</th>
</tr>
</thead>
<tbody>
<tr>
<td>00005, 00010, …, 00100</td>
<td>5, 10, …, 100</td>
</tr>
<tr>
<td>00105, 00110, …, 00200</td>
<td>4, 9, …, 99</td>
</tr>
<tr>
<td>00205, 00210, …, 00300</td>
<td>3, 8, …, 98</td>
</tr>
<tr>
<td>00305, 00310, …, 00400</td>
<td>2, 7, …, 97</td>
</tr>
</tbody>
</table>
Hashing Functions – Character Strings

• Hash functions for strings generally work by *adding up some expression* applied to each character in the string (remember that a char is just another integer type in C++).

• Although a char could be any of 255 different values, most strings actually contain only the 96 “printable” characters starting at 32 (blank).

• This code doesn’t work very well. Words that differ only by *transposition of characters* would have the same hash value.

• Use multipliers to make *every character position “count” differently* in the sum
  - C is an integer multiplier, M is a modulus
  - Both choses as *prime numbers*
  - C could be small, M should be large

```cpp
unsigned hash ( const string& s )
{
    unsigned h = 0;
    for ( int i = 0; i < s.length(); i ++ )
        h += s[ i ];
    return h;
}
```

```cpp
unsigned hash ( const string& s )
{
    unsigned h = 0;
    for ( int i = 0; i < s.length(); i ++ )
        h = (C*h + s[ i ]) % M;
    return h;
}
```
Hashing Functions – Compound Structures

• Figure out **which components of the compound type are critical to identifying** the object

• Compute **hash functions on those components** and combine those hash values into an overall hash function.

• **Ex: Book class:**
  
  • take advantage of the fact that each book has a unique ISBN number
    
    • maps to simple problem of hashing a single string.

  • use a combination of the other fields that, combined, would uniquely identify the book.

```cpp
class Book {
public:
    Book ( … ) ;
    …
    int hash ( ) const ;
private:
    Author theAuthor ;
    string theTitle ;
    string theIsbn ;
    Publisher thePublisher ;
    int theEdition ;
    int theYear ;
} ;

unsigned hash ( const string& s )
{
    unsigned h = 0 ;
    for ( int i = 0 ; i < s.length ( ) ; i ++)
        h = (7*h + s [ i ] ) % 32761 ;
    return h ;
}

int Book : : hash ( ) const
{
    return hash ( theIsbn ) ;
}

int Book : : hash ( ) const
{
    return theAuthor.hash ( ) + 3*hash ( theTitle )
    + 5* thePublisher.hash ( ) + 7*theEdition ;
}
```
Hashing and Equality

• A key requirement if we are going to use hashing is that comparing hash codes is treated as a way to see if two values *might* be equal to one another

• For a good hash function,
  • If \( x = y \), then \( \text{hash}(x) = \text{hash}(y) \).
  • If \( \text{hash}(x) = \text{hash}(y) \), then there is a good chance that \( x = y \).
  • If \( \text{hash}(x) \neq \text{hash}(y) \), then \( x \neq y \).
C++ Function Objects

• The hash container should contain a hash function in its declaration
  • How to associate the hash with the container?

• C++ has syntax that allows to treat a function as an object defined by a class called a function object type.
  • We define the hash functions as function object types
  • Extend the template syntax to include the function object types as template argument

• The definition of a function object begins with a template class that includes an overloaded version of the function call operator().
  • Argument list is the list of arguments required by the function

```cpp
template<typename T>
class greaterThan {
  public:
    bool operator() (const T& x, const T& y) const;
    {
      return x > y;
    }
```

Any type

Any no of arguments
Example – Insertion Sort

// objects of type lessThan<T> evaluate x < y
template<typename T>
class lessThan {
public:
    bool operator() (const T& x, const T& y) const {
        return x < y;
    }
};

// use the insertion sort to order v using
// function object comp

template<typename T, typename Compare>
void insertionSort(vector<T>& v, Compare comp)
{
    int i, j, n = v.size();
    T temp;
    // place v[i] into the sublist v[0] ... v[i-1],
    // 1 <= i <= n-1, so it is in the correct position
    for (i = 1; i < n; i++) {
        // index j scans down list from v[i] looking for
        // correct position to locate target. assigns it to v[j]
        j = i;
        temp = v[i];
        // locate insertion point by scanning downward as
        // long as comp(temp, v[j-1]) is true and we have
        // not encountered the beginning of the list
        while (j > 0 && &comp(temp, v[j-1])) {
            // shift elements up list to make room for insertion
            v[j] = v[j-1];
            j--;
        }
        // the location is found; insert temp
        v[j] = temp;
    }
}

int main()
{
    int arr[] = {2, 1, 7, 8, 12, 15, 3, 5};
    int arrSize = sizeof(arr)/sizeof(int);
    vector<int> v(arr, arr+arrSize);
    // put the vector in ascending order
    insertionSort(v, lessThan<int__()});
    // output it
    writeVector(v);
    return 0;
}

Run:
1 2 3 5 7 8 12 15
C++ Function Objects

• C++ STL implements the function object types `greater` and `less` in the header file `<functional>`. These types perform the same actions `greaterThan` and `lessThan` types.

```cpp
template <typename T>
class greaterThan {
public:
    bool operator() (const T& x, const T& y) const;
    {
        return x > y;
    }
};

template <typename T>
class lessThan {
public:
    bool operator() (const T& x, const T& y) const;
    {
        return x < y;
    }
};
```
Resolving Collisions

• Given hash functions are not perfect, we can expect collisions. *How do we resolve these collisions in the hash container?*

• There are two general approaches:
  • *Structural Indexing:* The hash table performs as an “index” to a set of structures that hold multiple items.
  • *Open Addressing:* Search for an open slot within the table.
Structural Indexing

• Use fixed size *buckets*, e.g., an array that can hold up to \( k \) (some small constant) elements.
  - The problem with this is that if we get *more than \( k \) collisions* at the same location, we still need to fall back to some other scheme.

• Instead, use *separate chaining*, in which the hash table is implemented as an *array of variable sized containers* that can hold however many elements.
  - Typical choices for this container would be a *linked list* (which is where the term “chaining” actually comes from), or a *tree-based set*.
  - Although these containers are variable size, some people *still call them* “buckets”.

\[
\begin{array}{ll}
\text{< Bucket}_0> & \rightarrow 77(1) \\
\text{< Bucket}_1> & \rightarrow 89(1) \quad 45(2) \\
\text{< Bucket}_2> & \rightarrow 35(1) \\
\text{< Bucket}_3> & \rightarrow 14(1) \\
\text{< Bucket}_4> & \rightarrow 94(1) \\
\text{< Bucket}_5> & \rightarrow \quad \quad \\
\text{< Bucket}_6> & \rightarrow \quad \quad \\
\text{< Bucket}_7> & \rightarrow \quad \quad \\
\text{< Bucket}_8> & \rightarrow \quad \quad \\
\text{< Bucket}_9> & \rightarrow \quad \quad \\
\text{< Bucket}_{10}> & \rightarrow 54(1) \quad 76(2)
\end{array}
\]
Implementation of Structural Indexing

• Implementing separate chaining as a **vector of linked lists** of elements.
  • HashFun is a function object class used to provide the hash function *hash*.
  • CompareEQ is another function object class, used to provide an *equality comparison*

• Finding the Bucket
  • Locates the list that will contain a given element, if that element really is somewhere in the table.

```c++
template <class T, int hSize ,
        class HashFun,
        class CompareEQ=equal_to<T> >
class hash_set
{
   typedef list<T> Container ;
   public :
      hash_set( ) : buckets(hSize), theSize(0) { }
      ...
   private :
      vector <Container> buckets ;
      HashFun hash ;
      CompareEQ compare ;
      int theSize ;
};

Container& bucket ( const T& element )
{
   return buckets [hash( element ) % Size ] ;
}
```
Implementation of Structural Indexing

• Searching the Set
  • First we use bucket() to find the list where this element would be.
  • Then we search the list for an item equal to element.

• Inserting into the Set
  • We use bucket() to find the list
  • Then, search the list for the element we want to insert.
  • If it’s already in there, we replace it. If not, we add it to the list

• Removing from the Set
  • Follows pretty much the same pattern.
Complexity of Structural Indexing

• Suppose we have inserted $N$ items into a hash table of size $hSize$.
  • In the *worst case*, all $N$ items will hash to the same list, and we will be reduced to doing a linear search of that list: $O(N)$.
    • If we use sets instead of lists for the buckets, this would reduce this cost to $O(\log N)$.
  • In the *average case*, we assume that the $N$ items are distributed evenly among the lists. Since we have $N$ items distributed among $hSize$ lists, we are looking at $O(N / hSize)$.
    • If $hSize$ is much larger than $N$, and if our hash function uniformly distributes our keys, then most lists will have 0 or 1 item, and the average case would be approximately $O(1)$.
    • If $N$ is much larger than $hSize$, we are looking at an $O(N)$ linear search sped up by a constant factor ($hSize$), but still $O(N)$.

• Thus, hash tables let us trade space for speed.
Open Addressing

- In open addressing, the hash array contains *individual elements* rather than a collection of elements.

- When a key we want to insert *collides* with a key already in the table, we resolve the collision by *searching for another open slot* within the table where we can place the new key.

![Diagram showing open addressing with keys inserted and collisions resolved.](image)
Implementation of Open Addressing

- Each slot in the hash table contains one data element and a status field indicating whether that slot is **occupied**, **empty**, or **deleted**.

```cpp
enum HashStatus { Occupied, Empty, Deleted };

template <class T>
struct HashEntry {
    T data;
    HashStatus info;

    HashEntry() : info(Empty) {}
    HashEntry(const T& v, HashStatus status) : data(v), info(status) {}
};
```

- The hash table itself consists of a **vector/array** of these HashEntry elements.

```cpp
template <class T, int hSize, class HashFun, class CompareEQ=equal_to<T> >
class hash_set {
    public:
        hash_set() : table(hSize), theSize(0) {}
    ...
    private:
        int find(const T& element, int h0) const
        ...
        vector<HashEntry<T>> table;
        HashFun hash;
        CompareEQ compare;
        int theSize;
};
```
Implementation of Open Addressing

• Collisions are resolved by trying a series of locations, $h_0$, $h_1$, $h_{\text{size}-1}$, until we find what we are looking for.

• These locations are given by
  $$h_i(\text{key}) = (\text{hash(\text{key})} + f(i)) \mod \text{hSize}$$

• $f$ is some integer function, with $f(0) = 0$. We’ll look at what makes a good $f$. With these locations, the basic idea is
  • **Searching**: try cells $h_i(\text{key})$, $i = 0, 1, \ldots$ until we find the key we want or an empty slot.
  • **Inserting**: try cells $h_i(\text{key})$, $i = 0, 1, \ldots$ until we find the same key, an empty slot, or a deleted slot. Put the new key there, and mark the slot “occupied”.
  • **Erasing**: try cells $h_i(\text{key})$, $i \in 0, 1, \ldots$ until we find the key we want or an empty slot. If we find the key, mark that slot as “deleted”.


Implementation of Open Addressing

• The most common schemes for $f(i)$ are

• **Linear Probing**
  • $f(i) = i$
  • If a collision occurs at location $h$, we next check location $(h+1) \% h\text{Size}$, then $(h+2) \% h\text{Size}$, and so on.

• **Quadratic probing**
  • $f(i) = i^2$
  • If a collision occurs at location $h$, we next check location $(h+1) \% h\text{Size}$, then $(h+4) \% h\text{Size}$, then $(h+9) \% h\text{Size}$, and so on.
  • This function tends to reduce clumping (and therefore results in shorter searches).
  • Not guaranteed to find an available empty slot if the table is more than half full or if $h\text{Size}$ is not a prime number.
Implementation of Open Addressing

• **Double Hashing**
  
  • \( f(i) = i \times h_2(\text{key}) \)
    
    • \( h_2 \) is a second hash function.

    • If a collision occurs at location \( h \), and \( h_2 = h_2(\text{key}) \), we next check location \((h+h_2) \mod \text{hSize}\), then \((h+2 \times h_2) \mod \text{hSize}\), then \((h+3 \times h_2) \mod \text{hSize}\), and so on.

    • This also tends to reduce clumping, but, as with quadratic hashing, it is possible to get unlucky and miss open slots when trying to find a place to insert a new key
Analysis of Open Addressing

• Define $\lambda$, the \textit{load factor} of a hash table, as the number of items contained in the table divided by the table size.
  • The load factor measures what fraction of the table is full.
  • By definition, $0 \leq \lambda \leq 1$.

• Given an ideal collision strategy, the probability of an \textit{arbitrary cell being full} is $\lambda$.

• Consequently, the probability of an \textit{arbitrary cell being empty} is $1 - \lambda$.

• The average number of cells we would expect to examine before finding an open cell is therefore $1/(1 - \lambda)$.

• For an ideal collision strategy, finds and inserts are, on average, $O(min(1/(1 - \lambda), hSize))$.
Analysis of Open Addressing

• Here you can see the behavior of the function \(1/(1-\lambda)\) as the load factor, \(\lambda\) increases.

• If the table is less than half full \((\lambda < 0.5)\), then we are talking about looking at, on average, no more than 2 slots during a search or insert.

• As \(\lambda\) gets larger, the average number of slots examined grows toward \(h\text{Size}\) (and, if the table is getting full, then \(N\) is approaching \(h\text{Size}\), so we are once again degenerating toward \(O(N)\) behavior).

• So, the rule of thumb for hash tables is to keep them no more than half full.
  • Treat searches and inserts as \(O(1)\) operations.
  • If we let the load factor get much higher, we start seeing \(O(N)\) performance.
Hash-Based Sets and Maps

• The new C++11 standard includes hashing-based versions of set and map containers to serve in such circumstances.
  • will offer an average of nearly $O(1)$ time for insertion and searching.
  • we pay for this increase in speed with an increase in memory required.
  • Will rehash when the tables get full enough to degrade the performance.

• The *tree-based* set and map containers have the property that they keep their keys in order.
  • one of the things that we give up when using hash-based storage is that ordering.

• Because of this, the new hash-based containers have been dubbed *unordered associative containers*.
Balanced Search Trees
AVL Trees
Balanced Trees

• BST operation performance is bounded by the height of the tree, which can range from an ideal of $O(\log N)$ (balanced trees) to $O(N)$ (degenerate trees).

• Various algorithms have been developed for building search trees that remain balanced. We’ll look at 2:
  • AVL trees
  • B trees
AVL Trees

• An AVL tree (Adelson-Velskii and Landis) is a binary search tree for which each node’s children **differ in height by at most 1**.
  • Guarantees that the height of the tree is $O(\log N)$.
  • Need to maintain **height info** in each node.

• AVL insertion starts out identical to normal binary search tree insertion.
  • After the new node has been created and put in place, each of its ancestors must check to see if still balanced.
  • In a balanced tree, this difference must be **-1, 0, or 1**. 0 means that both subtrees have the same height. -1 means that the left tree is higher (by 1), and 1 means that the right tree is higher.
  • If any are unbalanced, the balance is restored by a process called **rotation**.

```cpp
template <class T>
class avlNode
{
    public:
        ...
        T value;
        avlNode<T> * parent;
        avlNode<T> * left;
        avlNode<T> * right;
        short balanceFactor;
};
```
AVL Trees Example

• Insert operation may cause balance factor to become 2 or –2 for some node
  • only nodes on the path from insertion point to root node have possibly changed in height
  • So after the Insert, go back up to the root node by node, updating heights
  • If a new balance factor (the difference $h_{\text{left}}-h_{\text{right}}$) is 2 or –2, adjust tree by rotation around the node
AVL Trees – Single Rotation

- Assume the shown BST tree.
- Let’s say that U has height 18.
- Assuming that U is unbalanced,
  - The height of its children must differ by 2
- If H is the higher child.
  - H must have height 17, and x must have height 15.
- There are two possibilities for the heights of H’s children.
  - Both be height 16, or
  - One could be 16 and the other 15.
Single Rotation

- We can solve both of these problems by **shifting the “y” subtree over to become a child of U**.
  - The resulting tree is balanced, and is shorter than it had been.

- The BST ordering rules suggest that the elements of the rotated tree, in ascending order, would be: x U y H z.

- So the implied ordering is the same as before, and this is still a BST after the rotation.

- This transformation is called a **single left rotation**.
Single Rotation

- The basic steps of **single left rotation** are:
  - Let U be the unbalanced node and H the higher of U’s two children.
  - Let I be the “interior” child of H, the child reached by stepping in the opposite direction used in going from U to H. For example, if H is a left child of U, then I would be the right child of H.
  - In U, replace the pointer to H by I.
  - In H, replace the pointer to I by U.
  - Treat H as the root of the resulting.
- Same steps work for **single right rotations** as well.
- We have to **re-compute** the heights of the affected nodes.
Insertions in AVL Trees

• Let the node that needs rebalancing be $\alpha$.

• There are 4 cases:
  
  **Outside Cases** (require single rotation) :
  1. Insertion into left subtree of left child of $\alpha$.
  2. Insertion into right subtree of right child of $\alpha$.

  **Inside Cases** (require double rotation) :
  3. Insertion into right subtree of left child of $\alpha$.
  4. Insertion into left subtree of right child of $\alpha$.

• The rebalancing is performed through four separate rotation algorithms.
Consider a valid AVL subtree

Inserting into X destroys the AVL property at node j

Do a "right rotation"

"Single Right rotation" done!

AVL property has been restored!
Example - Single Rotation
Implementing AVL Single Rotation

template <class T>
avlNode<T>* avlNode<T>::singleRotateLeft ()
// perform single rotation rooted at current node
{
    avlNode<T>* U = this;
    avlNode<T>* H = U->right;
    avlNode<T>* I = H->left;

    U->right = I;
    H->left = U;
    if ( I != 0 )
        I->parent = U;
    H->parent = U->parent;
    U->parent = H;
    // now update the balance factor
    int Ubf = U->balanceFactor;
    int Hbf = H->balanceFactor;
    if (Hbf <= 0) {
        if (Ubf >= 1)
            H->balanceFactor = Hbf - 1;
        else
            H->balanceFactor = Ubf + Hbf - 2;
        U->balanceFactor = Ubf - 1;
    }
    else {
        if (Ubf <= Hbf)
            H->balanceFactor = Ubf - 2;
        else
            H->balanceFactor = Hbf - 1;
        U->balanceFactor = (Ubf - Hbf) - 1;
    }
    return H;
}
AVL Trees – Double Rotation

• A single rotation doesn’t always do the job.

• In single rotation, we assumed that subtree $z$ was higher than $y$. What would happen if $y$ were the higher of the two?
  
  • Single rotation does not produce a balanced tree

• The solution is:
  
  1. Do a single right rotation of $H$ to shift height to the right, making “$z$” higher
  2. Do a single left rotation of $U$.

• This combination is called a double left rotation. (There is, of course, a mirror image “double right rotation” as well.)
AVL Insertion: Inside Case

Consider a valid AVL subtree

Inserting into Y destroys the AVL property at node j

Does “right rotation” restore balance?

“Right rotation” does not restore balance… now k is out of balance
AVL Insertion: Inside Case

Consider a valid AVL subtree

\[ \begin{array}{c}
H \\
\downarrow \\
X \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Z \\
\end{array} \]

Inserting into Y destroys the AVL property at node j

\[ \begin{array}{c}
H \\
\downarrow \\
X \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Z \\
\end{array} \]

Consider the structure of subtree Y

\[ \begin{array}{c}
H \\
\downarrow \\
X \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Z \\
\end{array} \]

We will do a left-right; "doublerotaton"

left rotation complete

Now do a right rotation

Y = node i and subtrees V and W

\[ \begin{array}{c}
H \\
\downarrow \\
X \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Z \\
\end{array} \]

\[ \begin{array}{c}
H \\
\downarrow \\
X \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Y \\
\end{array} \quad \begin{array}{c}
H \\
\downarrow \\
Z \\
\end{array} \]
 AVL Insertion: Inside Case

right rotation complete

Balance has been restored

right rotation complete
Example - Double Rotation

Insertion of 34
template <class T>
avlNode<T>* avlNode<T>::insert ( const T& val )
// insert a new element into balanced AVL tree
{
    if ( val < value ) { // insert into left subtree
        if ( left != 0 ) {
            int oldbf = left ->balanceFactor ;
            left = left ->insert ( val ) ;
            // check to see if tree grew
            if ( ( left ->balanceFactor != oldbf ) &&
                 left ->balanceFactor )
                balanceFactor--; // check if we are now out of balance, if so balance
        } else {
            left = new avlNode ( val , this ) ;
            balanceFactor--; // check if we are now out of balance, if so balance
        }
    } else { // insert into right subtree
        if ( right != 0 ) {
            int oldbf = right ->balanceFactor ;
            right = right ->insert ( val ) ;
            // check to see if tree grew
            if ( ( right ->balanceFactor != oldbf ) &&
                 right ->balanceFactor )
                balanceFactor++; // check if we are now out of balance, if so balance
        } else {
            right = new avlNode ( val , this ) ;
            balanceFactor++; // check if we are now out of balance, if so balance
        }
    }
    // check if we are now out of balance, if so balance
    if ( ( balanceFactor < -1 ) || ( balanceFactor > 1 ) ) ,
        return balance ( ) ;
    else
        return this ;
}
AVL Balance

The process of rebalancing a node consists mainly of determining whether we need a single or double rotation, then applying the appropriate rotation routines.

template <class T>
avlNode<T>* avlNode<T> :: balance ()
{
    // balance tree rooted at node
    // using single or double rotations as appropriate
    if (balanceFactor < 0) {
        if (left->balanceFactor <= 0)
            // perform single rotation
            return singleRotateRight ( ) ;
        else {
            // perform double rotation
            left = left->singleRotateLeft ( );
            return singleRotateRight ( ) ;
        }
    }
    else {
        if (right->balanceFactor >= 0)
            return singleRotateLeft ( ) ;
        else {
            // perform double rotation
            right = right->singleRotateRight ( ) ;
            return singleRotateLeft ( ) ;
        }
    }
}
AVL Trees Deletion

• Similar but *more complex* than insertion
  • Rotations and double rotations needed to rebalance
  • Imbalance may propagate upward so that *many rotations* may be needed.
AVL Trees Complexity

• An AVL tree is balanced, so its height is $O(\log N)$ where $N$ is the number of nodes.

• The rotation routines are all themselves $O(1)$

• Insertion into an AVL tree has a worst case $O(\log N)$.

• Searching an AVL tree is completely unchanged from BST’s, and takes time proportional to the height of the tree, making $O(\log N)$.

• Removing nodes from a binary tree also requires rotations, but remains $O(\log N)$ as well.
Balanced Search Trees
B-Trees
B-Trees

- B-trees are a form of balanced search tree based upon general trees.
- A B-tree node can contain several data elements, rather than just one as in binary search trees.
- They are especially useful for search structures stored on disk. Disks have different retrieval characteristics than internal memory (RAM).
  - Obviously, disk access is much, much slower.
  - Furthermore, data is arranged in concentric circles (called tracks) on each side of a disk “platter”.
- B-trees are a good match for on-disk storage and searching because we can choose the node size to match the cylinder (multiple parallel tracks) size.
- In doing so, we will store many data members in each node, making the tree flatter, so fewer node-to-node transitions will be needed.
B-Trees

For a B-tree of order m:

- All **data is in leaves**. **Keys** (only) can be **replicated** in interior nodes.
- The root is either
  - a leaf, or
  - an interior node with 2 ... m children
- All **interior nodes** other than the root have \( \lfloor m/2 \rfloor \) ... m children
- All **leaves** are at the **same depth**.

**Example of a B-tree of order 4.**
B-Tree Search

The process of rebalancing a node consists mainly of determining whether we need a single or double rotation, then applying the appropriate rotation routines.

```cpp
BTreeNode<T>* find ( const T& x , BTreeNode<T> * t )
{
    if ( t is a leaf )
        return t ;
    else
    {
        i = 1;
        while ( ( i < m ) && ( x >= t->key [ i ] ) )
            ++ i;
        return find ( x , t->child [ i ] ) ;
    }
}
```
B-Tree Insertion

• Inserting into a B-tree starts out by "find"ing the leaf in which to insert.
  • If there is room in the leaf for another data item, then we’re done.
  • If the leaf already has m items, then there’s no room.
    • Split the overfull node in half and pass the middle value up to the parent for insertion there.
    • If the value passed up to the parent causes the parent to be over-full, then it too splits and passes the middle value up to its parent.

• Deletion is usually lazy or semi-lazy (delete from leaf but do not remove keys within the interior nodes).
B-Tree Complexity

- The maximum depth of an order m BTree is $\lceil \log_{m/2}(n) \rceil$
- At each node, we do $O(\log m)$ work to choose branch
- An insert or delete may need $O(m)$ work to fix up info in a node

Worst cases are:
- Find: $O(\log(m) * \log_m(n))$
  
  But, since $\log_m(n) = \frac{\log(n)}{\log(m)}$, this simplifies to $O(\log(n))$.
- Insert/delete: $O(m \log_m(n)) = O\left(\frac{m}{\log(m)} \log(n)\right)$
Balanced Search Trees
Red-Black Trees
Red-Black Trees

- B-trees are generally used with a fairly high width (order).
- A closely related data structure arises when we take a B-tree of order 4 and relax just a few rules, including not storing all the data in the leaves but allowing some data to reside in the internal tree nodes.
- The result is called a 2-3-4 tree because each non-leaf node will, depending upon how full it is, have either 2, 3, or 4 children.
- There is a fairly simple way to map 2-3-4 trees onto binary trees to which a "color" has been added

```cpp
class RedBlackNode
{
    public:
        <[ : ]>
        T value;
        RedBlackNode<T> * parent;
        RedBlackNode<T> * left;
        RedBlackNode<T> * right;
        bool color; // true=red, false=black
};
```
Red-Black Trees

• A 2-3-4 node with 2 children (1 data value) is represented by a *black binary tree node* whose *children are either leaves or black nodes*.

• A 2-3-4 node with 3 children (2 data values) is represented by a *black binary tree node with one red child*, the *other child being a leaf or a black node*. (Either child could be the red one, so the mirror image of the binary tree in this diagram is also legal.)

• A 2-3-4 node with 4 children (3 data values) is represented by a *black binary tree node with two red children*. 
Red-Black Trees

- Here is an example of the red-black equivalent to a 2-3-4 search tree.
Red-Black Trees

Some things to note:

• The root of a red-black tree is always black.
• No red node will ever have a red child.
• The red-black tree is a binary search tree and can be searched using the conventional binary search tree "find" algorithm.
• The height of a red-black tree is no more than twice the height of the equivalent 2-3-4 tree.
  • And we have already noted that the height of B-trees, including 2-3-4 trees, is $O(\log N)$ where $N$ is the number of data items in the tree.
  • We therefore know that the height of a red-black tree is also $O(\log N)$.
  • And that searches on a red-black tree have a $O(\log N)$ worst case.
Red-Black Trees

• The algorithms to insert nodes into a red-black tree add no more than a constant time for each node in the path from the root to the newly added leaf.

• Insertions into a red-black tree are worst case $O(\log N)$.
  • code for red-black trees re based on rotations very similar to those of AVL trees.

• Red-black trees are used in most implementations of set, mset, map, and mmap in the C++ std library.
Questions?
Assignment #6

• Due Sun Nov 10th, 11:59pm

• Written Assignment
  • Ford & Topp,
    Chapter #11 & #12:

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.14</td>
<td>P.639</td>
</tr>
<tr>
<td>Q.21</td>
<td>P.640</td>
</tr>
<tr>
<td>Q.22</td>
<td>P.640</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Question #</th>
<th>Page #</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q.13</td>
<td>P.717</td>
</tr>
<tr>
<td>Q.15</td>
<td>P.718</td>
</tr>
<tr>
<td>Q.18</td>
<td>P.718</td>
</tr>
<tr>
<td>Q.21</td>
<td>P.720</td>
</tr>
<tr>
<td>Q.24</td>
<td>P.721</td>
</tr>
</tbody>
</table>
Assignment #6

• Due Sun Nov 10th, 11:59pm

• Submission Format:
  • Written Assignment
    • Create single PDF file with name:  
      cs361_assignment_6_<firstName>_ <lastName>
    • Have a cover page with your name and your email
    • Submit through Blackboard.

• Final Submission Materials (1 Files):
  • One PDF for written assignment.