# STL: Algorithms

#include <algorithm>

copy(b, e, Dest)

find(b, e, val)

remove(b, e, val)

sort(b, e)

unique(b, e)

Count\_if(b, e, val)

accumulate(b, e, initialSum)

# STL: Associative Containers

Pair: data type then parameter

*make\_pair(4, “fred”)*

change means delete and insert.

You must be able to compare the items you are storing in the *set*

The default comparison is *less*

map: 1:1, sorted, m[k] = v

multimap: 1:n, sorted, mm.insert(make\_pair(k,v))

set: unique elements, sorted

multiset: multiple keys allowed, sorted

# STL:

Vector:

Array-indexing syntax:

* + **vector<int> v(10);**
  + **v[0] = 4;**

v.end() returns an iterator to **just past** the last element in the vector

v.at(i) reference, range checked!

v[i] reference, not range checked

v.front() and v.back() give you values

v.begin() and v.end() give you iterators

# STL: List & Deque

## List:

It's like a doubly-linked list with head and tail

* + It does **not** allow array notation

Use list when you will be doing a lot of insertions and deletions from other than the end of the container

for( list<char \*>::iterator i = v.begin(); i != v.end(); i++ )

Bidirectional, linear list. Sequential access only (can't do myList[52])

l.push\_front(31); not available in vector

A lot of the functions (c.begin(), c.end()) are the same! So, write a “generic” display function. Now you can display a “supported” container with one function call!

## Deque:

Double-ended queue

Good for adding/removing at beginning or end

vector should be used by default (i.e. unless you have a good reason not to)

list should be used when there are frequent insertions and deletions from the middle

deque should be used when most insertions and deletions are at the beginning or end

## Queue:

FIFO

## Stack:

LIFO

## Iterator:

Never dereference end! (\*c.end() is bad!)

Prefer ++i because i++ makes a temporary object and returns it, incrementing later.

const\_iterator Like iterator, but changes can’t be made

reverse\_iterator Goes from the end to the beginning with same semantics as iterator

iterator insert(iterator pos, const T& x) Inserts x into the vector before pos

Using insert() on a vector or deque using iterators may (or may not) invalidate iterators and indices already referencing elements of that container

Insertions are performed by moving all the elements between *position* and the end of the vector to their new positions, and then inserting the new elements

# STL overloaded operator

Can put your classes in associative containers

IF the class has a bool operator<(T a, T b) or you must explicitly state the comparator

“equal-to” is simply: if( ! ((e1 < e2) || (e2 < e1)) )

And, if it’s not “less-than” and it’s not “equal-to”, then it must be “greater-than”!!

# Big O

There’s three ways to look at the analysis of algorithm efficiency: worst case, best case, average case

Typically, we use the Worst Case analysis. Four reasons why:

* 1. Many algorithms work to worst case (e.g. not finding item).
  2. Determining what the average case is often difficult.
  3. The best case scenarios of many algorithms are trivial (e.g. finding the first item).
  4. The worst case gives an upper bound.

The most important factors are loops and function calls

mathematical notation T(n)

O(2^n): generating all possible subsets of a set of data

# Extras:

**Tree:**

A tree is a series of elements organized in a hierarchical arrangement

A tree is typically sorted for speed of retrieval, not for adding or deleting data

The root is the top of the tree

Other nodes are connected under the current node. They are called the node’s children

Each node (except the root) has exactly one parent

A node that has no children is called a leaf

A series of links between the root and a leaf is called a branch

A level is a series of nodes equidistant from the root

Binary search tree:

A Binary Search Tree is a Binary Tree in which the nodes are ordered so that data that is less than the current node is always to the left of it

Preorder traversal

1. Process root, then
2. Traverse left, then
3. Traverse right

It is easy for a tree to be very lopsided

This is bad, since it defeats the purpose of making a tree in the first place (speed of access)

It is much preferred to make sure that your tree is as balanced as possible

A balanced tree never has leaves separated by more than one level

A balanced tree doesn’t have to be perfectly balanced

Balanced trees are far more efficient in searching than unbalanced trees

An AVL Tree:

An AVL Tree has a *balance factor* associated with each node

The balance factor of a node is the height of the left subtree minus the height of the right subtree

An AVL Tree must have all nodes’ balance factors equalling 0, 1, or -1

B-Tree: are **not** binary trees

Designed for quick searches **and** quick insertions and deletions

Internal nodes can have any number of child nodes

Number of child nodes is limited by the implementation

a 3-6 B-tree has between 3 and 6 child nodes

A B-tree of 10,000,000 keys with 50 keys per node never needs to retrieve more than 4 nodes to find any key.

Priority queues are like queues, except each element has a priority. When processing, it’s only FIFO for elements of the same priority

**Graphs:**

A graph is a set of nodes together with a set of edges that connect pairs of distinct nodes

Travelling Salesman problem:

Given a number of cities and the costs of travelling from any city to any other city, what is the least-cost round-trip route that visits each city exactly once and then returns to the starting city?

If most of the values in the matrix are identical, much space can be wasted. Instead, store the non-zero values as triplets of {row, column, value}

**Sorting**