Standard Template Library (STL)

To provide advanced, generic data structures to C++ programmers.

- (doubly) linked list (called list)
- dynamic array (called vector)
- stack and queue
- priority_queue
- tree (called set and multiset)
- hash table

This talk does not provide a detailed account of STL; it tries to cover the concept of STL designs, helping you read documents on your own.

Dynamic Arrays: Vectors

- To declare an integer dynamic array: vector<int> v;
- When vector v is created, it contains 0 elements.
- Use the push_back() method to add new elements to the vector.
  - To initialize v to contain N 0s,
    for (int i=0; i<N; i++) v.push_back(0);
  - each push_back() (or array growth) takes O(1) times
- Decrease the size of the vector by the pop_back() method.
- The current number of elements in v can be obtained by v.size().
- Use v[i] to access the i-th element of v, as long as i < v.size().
To print all the elements in vector \( v \):

```cpp
type<int>::iterator p;
for (p=v.begin(); p!=v.end(); p++) cout << *p;
```

- **Iterators** are generalized pointers.
- All STL containers (vectors, lists, sets, maps, etc.) provide iterators so that you use:
  - `begin()` to obtain a “pointer” to the first item in the container
  - `end()` to obtain the “NULL” value of the container
  - `++` and `--` operators to move to the next/previous item
  - `*` as the “dereference” operator to access the current item

---

**Implementing Adjacency Matrix by STL**

```cpp
class Graph {
protected:
    vector< vector<bool> > edges;
    vector<string> labels;
public:
    Graph () : edges(), labels() {}
    void add_vertex (const string& label);
    void add_edge (int u, int v) {edges[u][v] = true;}
    void remove_edge (int u, int v) {edges[u][v] = false;}
    bool is_connected (int u, int v) {return edges[u][v];}
    string& operator[] (int i) {return labels[i];}
    int size() {return labels.size();}
};
```

Note that we have removed the capacity limit of our previous adjacency matrix implementation.
**Deque**

- A deque is a dynamic array that can grow and shrink on both ends.
- Declaration: `deque<char> d;`
- The $i$-th element in a deque $d$ is still accessed by $d[i]$.
- Furthermore, `push_front()` and `pop_front()` methods are provided.
- How is a deque implemented?

**Linked List**

- To declare a (doubly) linked list: `list<int> l;`
- A doubly linked list can be manipulated on both ends.
  - `push_back()`, `pop_back()`, `push_front()`, and `pop_front()` methods are provided.
- Various `insert()` and `erase()` methods are provided to add/delete nodes anywhere in the list.
- You don’t have access to the next/prev pointers; you use iterators instead:
  ```
  list<int>::iterator p;
  for (p=v.begin(); p!=v.end(); p++) cout << *p;
  ```
- Of course, you can’t use the [] operator on lists.
```cpp
class Graph {
protected:
    vector<list<int>> neighbors;
    vector<string> labels;
public:
    Graph() : neighbors(), labels() {}
    void add_vertex(const string& label);
    void add_edge(int u, int v);
    void remove_edge(int u, int v);
    bool is_connected(int u, int v);
    string& operator[](int i) {return labels[i];}
};

void Graph::add_vertex(const string& label) {
    neighbors.push_back(list<int>());
    labels.push_back(label);
}

bool Graph::add_edge(int u, int v) {
    if (is_connected(u, v)) return false;
    neighbors.push_front(v);
    return true;
}

list<int>::iterator Graph::find_neighbor(int u, int v) {
    list<int>::iterator p;
    for (p = neighbors[u].begin(); p != neighbors[u].end(); p++)
        if (*p == v) return p;
    return neighbors[u].end();
}
```
bool Graph::is_connected (int u, int v) {
   return find_neighbor(u,v) != neighbors[u].end();
}

bool Graph::remove_edge (int u, int v) {
   list<int>::iterator p = find_neighbor (u,v);
   if (p == neighbors[u].end()) return false;
   neighbors[u].erase (p);
   return false;
}

---

**Stacks and Queues**

- To declare an integer stack: `stack<int> s1;`
  - `s1` will be implemented by a deq
- You can also specify the implementation.
  - `stack<int, list<int>> s2;`
  - `stack<double, vector<double>> s3;`
- You can do the usual things with STL stacks, such as `push()`, `pop()`, and `top()`.
- Queues:
  - `queue<StudentRec> stuQ;` (implemented by a deq)
  - `queue<int, list<int>> q2;`
  - `queue<double, vector<double>> q3;`
- Queue methods: `push()`, `pop()`, `front()`, and `back();`
Priority Queues

- `priority_queue< int, less<int> >` (default), a priority queue of integers with the largest element at the top.
- `priority_queue< char, greater<char> >`, a priority queue of char with the smallest element at its top.

Important methods:
- `push(T&)`: add a new entry to the queue
- `pop()`: remove and return the top entry of the queue
- `top()`: return the top entry
- `size()`: the number of entries in the queue

Examples

```cpp
priority_queue <int> q1;
priority_queue <int, greater<int> > q2

q1.push(1); q2.push(1);
q1.push(2); q2.push(2);
q1.push(3); q2.push(3);
q1.push(4); q2.push(4);
q1.push(5); q2.push(5);

q1.top() ======> ?
q2.top() ======> ?
```
Sets and Multisets

- To declare an integer set: `set<int> s1;`
- To declare a string multiset: `multiset<string> s2;`  
  - A multicast allows multiple instances of a value
- The `insert(T& new_value)` method returns a `pair<iterator,bool>`.
  - The `iterator` part, named `first`, points to the place in the tree that contains `new_value`
  - The `bool` part, named `second`, indicates whether the insertion is successful or not
  ```cpp
  pair<set<int>::iterator,bool> result = s1.insert (99);
  if (result.second==true) // insertion successful
  else // insertion failed; 99 already in s1
  ```

- The `find(T& target)` method returns an `iterator` that points to the `target` value in the set.
  - If `target` does not exist in the set, then the method returns the value of `end()`
  ```cpp
  set<int>::iterator p = s1.find (99);
  if (p==s.end()) // 99 not in set s1
  else // 99 exists in set s1
  ```

- Before declaring a `set<T>` object, you must make sure that class `T` provides comparison operators, such as `>`, `>=`, `<`, and `<=`.
- This requirement is necessary because STL sets/multisets are implemented by a balanced tree data structures, called red-black trees.
- All of the `insert()`, `erase()`, and `find()` methods require only $O(\log N)$ time.
Since trees are not linear, sets don’t support push_back(), pop_back(), push_front(), and pop_front.

You still use iterators to enumerate all the entries in a set.

```cpp
set<int>::iterator p;
for (p=s1.begin(); p!=s1.end(); p++)
    cout << *p;
```

The order in which the entries are visited confirms with the inorder traversal.

Set union, intersection and difference operations are provided.

---

Maps

STL maps are like AssocArrays; they store the mapping from keys to values.

Example: `map<string, int> wc;`

The similarity goes even deeper — the [] operator is provided too, including the default value semantic.

You can search a key without automatically adding it to the map by

```cpp
iterator find (KeyType& key);
```

Removal is achieved by `erase(KeyType& key).`
Project 2 Solution

#include <map.h>
#include <string>
#include <fstream>

void main(int argc, char* argv[]) {
    ifstream infile(argv[1]);
    map<string, int> wc;
    string current = "";
    char c;

    while((c=infile.get()) != EOF) {
        if(c >= 'A' && c <= 'Z') c = c + 'a' - 'A';

        if(c >= 'a' && c <= 'z')
            current += c;
        else if (current != "") {
            wc[current]++;
            current = "";
        }
    }
    // enter interactive mode
}
Red-Black Trees

- A B-tree with MINIMUM=1 is called a 2-3 tree.
- There are two representations of a 2-3 tree.
  1. Use the general-case B-tree representation with MINIMUM set to 1.
  2. Use a variation of binary tree representation.
template <class T>
class RedBlackTreeNode {
    T data;
    RedBlackTreeNode* left;
    RedBlackTreeNode* right;
    bool color; // color of the incoming pointer
};

Let $p$ be a pointer to the root node of the B-tree.

data[0] of the root is accessed by $p->data$.
data[1] of the root is accessed by $p->right->data$.
subtree[0] of the root is accessed by $p->left$.
subtree[1] of the root is accessed by $p->right->left$.
subtree[2] of the root is accessed by $p->right->right$. 

Another Drawing of Red-Black Tree

![Red-Black Tree Diagram]
Observations

- A red-black tree, despite of its origin as a B-tree, satisfies all the rules of binary search trees.
  
  You can use the search() method of binary search trees to search a red-black tree.

- The depth of a red-black tree of $N$ nodes is of $O(\log N)$. 