Review of Data Structures

- We’ve studied concrete data structures
  - Vectors
    - Homogeneous aggregates supporting random access
  - Linked lists
    - Collections supporting constant-time insertion
  - Trees
    - Combine efficiency of search/insert from vector/linked list

- These are concrete because we haven’t viewed them abstractly
  - Abstractly, what are operations performed on vector?
    - Vector implemented using “raw” C++/C arrays
  - Compare to Multiset which is more of an abstraction
    - Different implementations had important trade-offs
ADTs: Abstract Data Types

- Multiset is an ADT
  - Operations together with domain of elements
  - Implementations change, client programs use abstract interface

- Is MSApplicant an abstract data type? (from MultiSet class)

- We’ll look at several other ADTs
  - Stack and queue are related to vector/linked list: linear
  - Map is non-linear (as is tree)
  - Priority Queue is non-linear
  - Graph is non-linear
Stack: What problems does it solve?

- Stacks are used to avoid recursion, a stack can replace the implicit/actual stack of functions called recursively

- Stacks are used to evaluate arithmetic expressions, to implement compilers, to implement interpreters
  - The Java Virtual Machine (JVM) is a stack-based machine
  - Postscript is a stack-based language
  - Stacks are used to evaluate arithmetic expressions in many languages

- Limited range of operations, supports LIFO addition/deletion, last in is first out
  - Operations: push, pop, top, create, clear, size
  - More in postscript, e.g., swap, dup, rotate, …
Simple stack example

- **tstack** is a templated class, stores any type of value that can be assigned (like `tvector`)
  - Implemented simply using a vector, what does pop do?

```cpp
tstack<int> s;
s.push(2);
s.push(3);
s.push(1);
cout << s.size() << endl;
cout << s.top() << endl;
s.pop();
cout << s.top() << endl;
int val;
s.pop(val);
cout << val << endl;
```
Templated class, .h ok, .cpp ugly

- See tstack.h for example

```cpp
template <class Type>
class tstack
{
    public:
        tstack( );                   // construct empty stack
        const Type & top( ) const;   // return top element
        bool  isEmpty( )    const;   // return true iff empty
        int   size( )       const;   // # elements

        void push( const Type & item ); // push item

- But look at part of stack.cpp, class is templated (ugly?)

```cpp
template <class Type>
bool tstack<Type>::isEmpty() const
{
    return myElements.size() == 0;
}
Postfix, prefix, and infix notation

- **Postfix notation used in some HP calculators**
  - No parentheses needed, precedence rules still respected
    - 3 5 + 4 2 * 7 + 3 – 9 7 + *
  - Read expression
    - For number/operand: push
    - For operator: pop, pop, operate, push

- **See postfix.cpp for example code, key ideas:**
  - Read character by character, check state of expression
  - Can putback character on stream, only last one read

- **What about prefix and infix notations, advantages?**
Expression trees and *fix notations

- What is preorder of expression tree?
- Inorder and postorder?
- How can tree be constructed, e.g., if given postfix notation
  - Use postfix.cpp, but make tree
  - What goes on stack?
- What about subexpressions?

\[ 3 + (4 \times 5) - (7 + (4 \times 5)) \]
Queue: another linear ADT

- **FIFO**: first in, first out, used in many applications
  - Scheduling jobs/processes on a computer
  - Tenting policy?
  - Computer simulations

- **Common operations (as used in tqueue.h/tqueue.cpp)**
  - Add to back, remove from front
    - Called `enqueue`, `dequeue`, like `s.push()` and `s.pop()`
    - Analog of `top()` is `front()`

- **We’ll use example of printing a tree in level order (treelevel.cpp)**
  - Compare to preorder without recursion, uses stack
Queue implementations

- Different implementations of queue (and stack) aren’t interesting from an algorithmic standpoint
  - Complexity is the same, performance may change (why?)
  - Use vector or linked list, any sequential structure

- Linked list is easy for stack, where to add/remove nodes?

- Linked list is easy for queue, where to add/remove nodes?

- Vector for queue is tricky, need \textit{ring buffer} implementation, add but wrap-around if possible before growing
  - Tricky to get right, difference between full and empty
Using linear data structures

- **We’ve studied vectors, stacks, queues, which to use?**
  - It depends on the application
  - Vector is multipurpose, why not always use it?
    - Make it clear to programmer what’s being done
    - Other reasons?

- **Other linear ADTs exist**
  - List: add-to-front, add-to-back, insert anywhere, iterate
    - Alternative: create, head, tail (see `CList<>` in tapestry)
    - Linked-list nodes are concrete implementation
  - Deque: add-to-front, add-to-back, random access
    - Why is this “better” than a vector?
    - How to implement?
Why use inheritance?

● **We want to program to an interface (an abstraction, a concept)**
  ➤ The interface may be concretely implemented in different ways, consider stream hierarchy

```cpp
void readStuff(istream& input) {...}

// call function
ifstream input("data.txt");
readStuff(input);
readStuff(cin);
```

➤ What about new kinds of streams, ok to use?

● **Open/closed principle of code development**
  ➤ Code should be open to extension, closed to modification
  ➤ Why is this (usually) a good idea?
Two examples

- Consider the expression example (expression.h/.cpp)
  - What do we need to do to add a Multiplication class?
    - What code must be modified vs. extended?

- Consider the tags assignment
  - How do we print/access the text of a web page, e.g., suppose we want to implement a search engine?
    - Should we ignore tags? All tags?
  - How could we access just the links on a web page?
  - The WebParser class has hook methods, also called the template pattern
    - Nomenclature confusing? Not a C++ template.
Tags and the class WebParser

- The base class has a `process()` method to access a web page
  - Client subclasses implement some of
    - `processTag(...)` called for each tag on a web page
    - `processText(...)` called for each non-tag/non-comment word
    - `processChar(...)` called for every character
  - These are the hook methods (aka template pattern)
  - How can we process a file rather than a web page?

- We inherit some behavior from parent class, don’t modify it, but extend the class by implementing new behavior
  - Good example of open/closed principle
From Google to Maps

• If we wanted to write a search engine we’d need to access lots of pages and keep lots of data
  ➤ Given a word, on what pages does it appear?
  ➤ This is a *map* of words->web pages

• How should map be stored? What are important issues?
  ➤ Speed (possibilities?) Memory?
  ➤ In a general map, what’s the *key* and what’s the *value*?

• Consider a simple map of string->definition pointer
  ➤ Definition can be anything since all pointers are alike
  ➤ Map just stores pointer to something, doesn’t know what
What is an SDmap? A Definition?

- Maps keys (strings) to values (definition pointers)
  - See the code in `mapcount.cpp`

```c
while (input >> w) {  // read string
    Definition * d = map->get(w);  // look it up
    if (d == 0) {
        map->insert(w, new Definition());  // not found, store
    } else {
        d->incr();  // found, bump count
    }
}
```
Accessing values in a map

- We can apply a function object to every element in a map
  - Similar to MultiSet
  - Simple to implement, relatively easy to use
  - Limited: must visit every map element (can’t stop early)

- Alternative: use an iterator (see Directory reading example)

  ```cpp
  Iterator it = map.getIterator();
  for(it.Init(); it.HasMore(); it.Next()) {
    cout << it.Current() << endl;
  }
  ```

- What could this print? String? Definition Pointer? Both?
  - Good idea, a little awkward in C++, but doable