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?

```c
#include <iostream>
#include <vector>

template<typename T>
class tstack
{
private:
    std::vector<T> data;
public:
    void push(T val)
    {
        data.push_back(val);
    }
    T pop()
    {
        if (data.empty())
            return T();
        return data.back();
    }
    T top()
    {
        if (data.empty())
            return T();
        return data.back();
    }
    size_t size()
    {
        return data.size();
    }
};

int main()
{
    tstack<int> s;
    s.push(2); s.push(3); s.push(1);
    std::cout << s.size() << std::endl;
    std::cout << s.top() << std::endl;
    s.pop();
    std::cout << s.top() << std::endl;
    int val;
    s.pop(val);
    std::cout << val << std::endl;
    return 0;
}```
 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 put back character on stream, only last one read
- What about prefix and infix notations, advantages?

Prefix notation in action

- Scheme/LISP and other functional languages tend to use a prefix notation
  ➞ (function arg1 arg2 ...) returns a value

```lisp
(define (square x) (* x x))
```

```lisp
(define (expt b n)
  (if (= n 0)
    1
    (* b (expt b (- n 1)))))
```

Postfix notation in action

- Practical example of use of stack abstraction
- PostScript is a stack language mostly used for printing
  ➞ drawing an X with two equivalent sets of code

```postscript
%!
200 200 moveto
100 100 rlineto
200 300 moveto
100 -100 rlineto
stroke showpage
```

- Other ways of drawing an X?
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 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?