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 was an abstraction

ADTs: Abstract Data Types

- Multiset and Set are ADTs
  - 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 linear structures
  - Map is non-linear, like table implementation of MultiSet
  - Priority queue 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;
```
**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 * 5) - (7 + (4 * 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
  - 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?

From BSTs to n-ary trees

- We’ve seen animal trees, expression trees, binary search trees
  - All nodes have 0, 1, 2 children, organized differently
  - What about n-ary trees, nodes have more than 2 children
- Alternatives for implementation? Vector of children? List?
  - What are advantages and disadvantages of each
  - In practice a linked-list is used, why?
  - Root has how many children?
    - Redraw as “tree”
    - Why similar to binary tree?

Functions for n-ary trees

- Counting leaf nodes (or internal nodes)
  - What’s base case? Identify leaf? Complexity?
  
```c
struct Gtree
{
    string info;
    Gtree * child;
    Gtree * sibling;
};
int leafCount(Gtree * t)
// post: return # leaves
```

Recognizing Recurrences

- Solve once, re-use in new contexts
  - T must be explicitly identified
  - n must be some measure of size of input/parameter
    - T(n) is the time for quicksort to run on an n-element vector

\[
\begin{align*}
T(n) &= T(n-1) + O(n) & \text{sequential search} & \mathcal{O}(n) \\
T(n) &= T(n/2) + O(1) & \text{binary search} & \mathcal{O}(\log n) \\
T(n) &= 2T(n/2) + O(1) & \text{tree traversal} & \mathcal{O}(n) \\
T(n) &= 2T(n/2) + O(n) & \text{quicksort} & \mathcal{O}(n^2)
\end{align*}
\]

- Remember the algorithm, re-derive complexity