# Data Structures

# Today's Agenda

- **Stack and Queue ADTs**
  - What are they
  - Like Ordered Lists, are "position oriented"
- **Use of Data Structures for Stacks and Queues**
  - arrays (statically & dynamically allocated)
  - linear linked lists
  - circular linked lists

# Remember

- Data abstraction is a technique for controlling the interaction between a program and its data structures and the operations performed on this data.
- It builds "walls" around a program's data structures. Such walls make programs easier to design, implement, read, and modify.

# Stacks

- Stacks are considered to be the easiest type of list to use.
- The only operations we have on stacks are to add things to the top of the stack and to remove things from the top of the stack.
- We never need to insert, delete, or access data that is somewhere other than at the top of the stack.

# Stacks: Operations

- When we add things to the top of the stack we say we are pushing data onto the stack.
- When we remove things from the top of the stack we say we are popping data from the stack.
- We also might want to determine if the stack is empty or full.

# Stacks: Operations

- Many computers implement function calls using a mechanism of pushing and popping information concerning the local variables on a stack.
- When you make a function call, it is like pushing the current state on the stack and starting with a new set of information relevant to this new function. When we return to the calling routine, it is like popping.

# Stacks: Operations

- When implementing the code for a stack's operations, we need to keep in mind that we should have functions to push data and to pop data.
- This way we can hide the implementation details from the user as to how these routines actually access our memory…why?
  - because you can actually implement stacks using either arrays or using linked lists with dynamic memory allocation.

# Stacks: Operations

- There are five important stack operations:
  1) Determine whether the stack is empty
  2) Add a new item onto the stack...push
  3) Remove an item from the stack...pop
  4) Initialize a stack (this operation is easy to overlook!)
  5) Retrieve the item at the top of the stack...without modifying the stack...peek

# Stack Operations

- Notice what was interesting about the last few slides
- The operations should be pushing, popping, peeking at the underlying data not allowing the client direct access to the data structure used for storing this data.

# Client Interface

- In addition, there are many different interpretations of how these operations should actually be implemented
  - should “is empty” be combined with “pop”
  - should “is full” be combined with “push”
  - should pop return and remove the item on the top of the stack, or
  - should pop just remove the item at the top of the stack, requiring that a “peek” also be provided

# Client Interface

```
class stack {
  public:
     stack();
     ~stack();
     int push(const data & );
     int pop(data &);
     int peek(data &);
     int isempty();  int isfull();
```

# Client Interface

- With the previous class public section
  - the constructor might be changed to have an integer argument if we were implementing this abstraction with an array
  - the int return types for each member function represent the “success/failure” situation; if more than two states are used (to represent the error-code) ints are a good choice; otherwise, select a bool return type

# Data Structures

- Now let's examine various data structures for a stack and discuss the efficiency tradeoffs, based on:
  - run-time performance
  - memory usage
- We will examine:
  - statically/dynamically allocated arrays
  - linked lists (linear, circular, doubly)

# Data Structures

- Statically Allocated array…

```
private:
        data array[SIZE];
        int number_of_items;
```

- Efficiency Discussion:
  - direct access (insert at “top” position, remove at “top” position)
  - the first element would not be considered the “top” in all cases, why? ... unnecessary shifts!
  - problem: fixed size, all stack objects have the same size array, size is set at compile time

# Data Structures

- Dynamically Allocated array…

```
private:
    data * array;
    int number_of_items;
    int size_of_array;
```

- Efficiency Discussion:
  - same as the previous slide, except for the timing of when the array size is selected
  - each stack object may have a different size array
  - problem: memory limitations (fixed size)

# Data Structures

- What this tells us is that a dynamically allocated list is better than a statically allocated list (one less problem)
  - if the cost of memory allocation for the array is manageable at run-time.
  - may not be reasonable if the array sizes are very large or if there are many stack objects
  - is not required if the size of the data is known up-front at compile time (and is the same for each instance of the class)

# Data Structures

- We also should have noticed from the previous discussion that…
  - stacks are well suited for array implementations, since they are truly direct access abstractions
  - with stacks, data should never need to shift
  - if pop doesn't return the "data" you need peek
  - if pop does return the "data" you may still have peek

# Data Structures

- Linear Linked list…

```
private:
        node * head;
        node * tail;  //???helpful? no!
```

- So, Where is the “top”?
  - if the top is at the “tail”, what would the implication of push and pop be? Push would be fine if a tail pointer were provided...but Pop would be disastrous, requiring traversal!!
  - if the top is at the “head”, push and pop are essentially “direct access”, no traversals needed!

# Data Structures

- So, did we need a tail pointer?
  - No! Head is sufficient
- How can we say it is essentially “direct access”?
- Array: array[top_index] = data
  - which actually is: *(array+top_index) = data
- Linked list: head->member = data
  - which actually is: (*head).member = data
- Of course, = may not be appropriate

# Data Structures

- Circular Linked list…

```
private:
        node * tail;
```

- There is nothing in a stack that will benefit from the last node pointing to the first node
- A circular linked list will require additional code to manage, with no additional benefits

# Data Structures

- Doubly Linked list…

```
private:     (each node has a node * prev)
        node * head;
        node * tail;  //???helpful?
```

- Again, there is nothing in a stack that will benefit from each node having a pointer to the previous node.
- Using a doubly linked list to allow the "top" to be at the "end" or tail position, this is a poor choice…wastes memory and adds additional operations…

# Data Structures

- So, which data structure is best for a stack?
- If we know at compile time the size of the stack, and if that size doesn't vary greatly, and if all objects of the class require the same size
  - then use statically allocated arrays
- If we know at run time (before we use the stack), how large the stack should be, it is manageable and doesn't vary greatly
  - then use a dynamically allocated array

# Data Structures

- If the size varies greatly, or cannot be allocated contiguously, or is unknown
  - use a linear linked list with little additional costs in run-time performance
  - does, however, add the memory cost of one additional address per node which should not be forgotten
  - what happens if we implemented the traversal algorithms using recursion?

# Queues

- We can think of stacks as having only one end; because, all operations are performed at the top of the stack.
  - That is why we call it a Last in -- First out data structure.
- A queue, on the other hand, has two ends: a front and a rear (or tail).

# Queues

- With a queue, data items are only added at the rear of the queue and items are only removed at the front of the queue.
  - This is what we call a First in -- First out structure (FIFO).
  - They are very useful for modeling real-world characteristics (like lines at a bank).

# Queue: Operations

- When we add things to the **rear** of the queue we **enqueue** the data
- When we remove things from the **front** of the queue we **dequeue** the data
- We also might want to determine if the queue is empty or full
- We may also want to **peek** at the front
- A **DEQUE** is a double ended queue where you can enqueue and dequeue at either front or rear

# Client Interface

- As with stacks, there are many different interpretations of how these operations should actually be implemented
  - should “is empty” be combined with “dequeue”
  - should “is full” be combined with “enqueue”
  - should dequeue return and remove the item at the front of the queue
  - should dequeue just remove the item at the front of the queue, requiring that a “peek” also be provided

# Client Interface

```
class queue {
  public:
     queue();
     ~queue();
     int enqueue(const data & );
     int dequeue(data &);
     int peek(data &);
     int isempty();  int isfull();
```

# Client Interface

- With the previous class public section
  - the constructor might be changed to have an integer argument if we were implementing this abstraction with an array
  - the int return types for each member function represent the “success/failure” situation; if more than two states are used (to represent the error-code) ints are a good choice; otherwise, select a bool return type
  - peek might be useful regardless of the interpretation of dequeue...

# Data Structures

- Now let's examine various data structures for a queue and discuss the efficiency tradeoffs, based on:
  - run-time performance
  - memory usage
- We will examine:
  - statically/dynamically allocated arrays
  - "circular" arrays
  - linked lists (linear, circular, doubly)

# Data Structures

- Linear Arrays…
  - direct access (insert at the “rear” position, remove at “front” position)
  - extreme problem: rightward drift
  - as data is inserted and removed it slowly drifts to the right; soon the ‘queue’ may seem empty but yet there is not data!
  - alternating enqueues and dequeues will cause this disastrous result...

# Data Structures

- Circular Array…dynamically allocated

```
private:
        data * array;
        int number_of_items;
        int size_of_array;
```

- Manner of Operation:
  - when the “rear” or “front” indices progress to the end, they wrap around back to the beginning
  - front = front % size_of_array +1;
  - rear = rear % size_of_array +1;

# Data Structures

- Efficiency Discussion of a Circular Array:
  - Progressing the indices is not as simple as just adding 1 with the increment operator (++front) which reduces the gain achieved from direct access
  - Still has fixed size memory limitations that we examined with stacks
    - do we know at compile time the maximum size
    - is there are large variance in the size
  - At least no shifting should be required (unless it is implemented incorrectly!)

# Data Structures

- Linear Linked list…

```
private:
        node * head;
        node * tail;  //???helpful? yes!
```

- So, Where is the “rear” and “front”?
  - if the rear is at the “head”, and the front is at the “tail” what would the implication of enqueue and dequeue be? Enqueue would be fine...but Pop would be disastrous, requiring traversal (even with a tail pointer,...why?)!!

# Data Structures

- So, Where is the “rear” and “front”?
  - if the rear is at the “tail”, and the front is at the “head” what would the implication of enqueue and dequeue be?
  - Enqueue would be fine **if there is a tail pointer**...and pop would be great (inserting at the head
  - No traversals needed!
  - No shifting!

# Data Structures

- So, did we need a tail pointer?
  - Yes! Otherwise traversals would have been required
- How does the linked list compare with the circular array?
- Array:

```
array[rear] = data
rear = rear % size +1;
```

- Linked list:

```
tail->next = new node;
tail = tail->next;
tail->member = data
tail->next = NULL;
```

# Data Structures

- Circular Linked list…

```
private:
        node * rear;
```

- With a queue, a circular linked list is a viable alternative but does require additional dereferencing to get to the “front” which is not required with a linear linked list.

# Data Structures

- Doubly Linked list…

```
private:     (each node has a node * prev)
        node * head;
        node * tail;  //???helpful?
```

- Again, there is nothing in a queue that will benefit from each node having a pointer to the previous node.
- UNLESS, the abstraction is a **deque**

# Next Time...

- Now that we have applied data structures to stacks and queues
- We will move on to examine other types of linked lists:
  - doubly linked lists
  - arrays of linked lists
  - linked lists of linked lists
  - linked lists with “dummy” head nodes
- We will also walk through the homework

# Data Structures