CSE 143

More Collection Types: Stacks and Queues

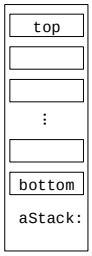
[Chapters 6, 7]

Collection ADTs

- We'll study the following "classic" ADTs
  - Stack: ordered collection accessible in LIFO manner
  - Queue: ordered collection accessible in FIFO manner
  - Set: unordered collection without duplicates
  - Table: unordered mapping of keys to values (e.g. names -> StudentRecords)
- We'll spend most of the rest of the quarter discussing various implementation techniques
  - Those techniques can then be used with every more complicated data structures.

Stack ADT

- *Top*: Uppermost element of stack, first to be removed
- *Bottom*: Lowest element of stack, last to be removed
- *Height* or Depth: Number of elements
- *Elements are always inserted and removed from the top (LIFO)*
- Homogeneous collection



Stack Operations

- *push(item)*: Adds an element to top of stack, increasing stack height by one
- *pop()*: Removes topmost element from stack and return it, decreasing stack height by one
- *top()*: Return a copy of topmost element of stack, leaving stack unchanged
- No iterator or cursor

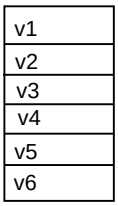
Stack Example

- Show the changes to the stack in the following example:

```
Stack s;
s.push(5);
s.push(3);
s.push(9);
int v1 = s.pop();
int v2 = s.top();
s.push(6);
s.push(4);
```


What is the result of:

```
Stack s;
s.push(1);
s.push(2);
int v1 = s.pop();
s.push(3);
s.push(4);
int v2 = s.pop();
s.push(5);
int v3 = s.pop();
int v4 = s.pop();
int v5 = s.pop();
int v6 = s.pop();
```



Common Uses of Stacks

- Implementing function calls
- Postfix notation
  - **`4 7 + 5 *`** instead of **`(4 + 7) * 5`**
  - **push** operands until operator found, **pop** 2 values, apply operator, **push** result
- Backtracking, e.g., to explore paths through a maze

Stack Interface

```
class IntStack {
public:
   IntStack( );           // construct empty stack
   bool isEmpty( );       // = "stack is empty"
   bool isFull( );        // = "stack is full"
   void push(int item);   // add item to top
   int pop( );            // remove & return top item
   int top( );            // return the top item
private:
   . . .
};
```


A Stack Client

```
// Read numbers and print in reverse order

void ReverseNumbers() {
 IntStack s;
 int k;
 while ( cin >> k ) {
   if ( !s.isFull( ) )
     s.push(k);
 }
 while ( !s.IsEmpty( ) )
   cout << s.pop() << endl;
}
```


Stack Implementations

- Many possibilities
  - Array-based
  - Linked-List
  - Reuse another collection type (list)
- Show one example here; rest left as exercises

Stack Implementation

- One possibility: store the data in an array

```
class IntStack {
public:
   ...
private:
   const int maxItems = 100;  // max # items in stack
   int nItems;          // # items currently in stack
   int data[maxElts];   // Items in stack are stored
                        // in data[0..nItems-1]. Data[0]
                        // is the bottom of the stack;
                        // data[nItems-1] is the top
                        // item on the stack.
};
```


Stack Implementation (2)

```
// construct empty IntStack
IntStack::IntStack() { nItems = 0; }

// = "this stack is full"
bool IntStack::isFull() {
   return nItems == maxItems;
}

// = "this stack is empty"
bool IntStack::isEmpty() {
   return nItems == 0;
}
```

Stack Implementation (3)

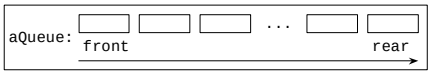
```
// Push item onto top of this stack
// pre: this stack is not full
void IntStack::push(int item) {
   assert (!isFull( ));
   data[nItems] = item;
   nItems++;
}

// return a copy of the top item on the stack
// pre: this stack is not empty
int IntStack::top() {
   assert (!isEmpty( ));
   return data[nItems-1];
}
```


Stack Implementation (4)

```
// Return the top item on this stack, and delete
// that item from the top of the stack.
// pre: this stack is not empty
int IntStack::pop() {
   assert (!isEmpty( ));
   int topVal = data[nItems-1];
   nItems--;
   return topVal;
}
```


Queue ADT

- Homogeneous, ordered collection, accessed only at the front (remove) and rear (insert)
- *Front*: First element in queue
- *Rear*: Last element of queue
- *FIFO*: First In, First Out



Queue Operations

- *enqueue(item)* : Adds an element to rear of queue
- *dequeue( )* : Removes and returns element at the front of queue
- *front( )* : Returns a copy of the front element of queue
- No iterator or cursor

Queue Example

- Show the changes to the queue in the following example:

```
Queue q;
q.enqueue(4);
q.enqueue(7);
q.enqueue(5);

int v1 = q.dequeue();
int v2 = q.front();

q.enqueue(9);
q.enqueue(2);
```


What is the result of:

```
Queue q;
q.enqueue(1);
q.enqueue(2);
q.enqueue(3);
int v1 = q.dequeue();
int v2 = q.front();
q.enqueue(4);
int v3 = q.dequeue();
int v4 = q.front();
q.enqueue(5);
int v5 = q.dequeue();
int v6 = q.front();
```

V1
V2
V3
V4
V5

Common Uses of Queues

- Managing a first-come, first-serve line
  - Processes in an operating system
  - Print jobs at the printer
- Buffering input/output
  - When printing to screen with **cout**, characters don't appear right away: they are buffered and sent out in groups.
  - Mouse events
- Simulations
  - People waiting in line for service

Queue Interface

```
class IntQueue {
public:
   IntQueue( );               // construct empty queue
   bool isFull( );            // = "queue is full"
   bool isEmpty( );           // = "queue is empty"
   void enqueue(int item);    // place item at end of q
   int dequeue( );            // return first item in
                              //   queue and remove it
   int front( );              // return copy of 1st item
private:
   ...
};
```


Queue Implementation

- Many possibilities. This example:linked list with pointers to ends of the queue

```
class IntQueue
public:
   ...
private:
   struct Node {
     int val;          // value in this queue node
     Node * next;      // ptr to next item in queue
   };
   Node * first;       // ptr to first node in queue
   Node * last;        // ptr to last node in queue
};                     // If queue is empty,
                       //   set first=last=NULL
```


Queue Implementation (2)

```
// construct empty queue
IntQueue::IntQueue( ) {
   first = last = NULL;
}

// = "this queue is full" (always false unless no
// more storage available to allocate new nodes.)
bool IntQueue::isFull( ) {
  return false;
}

// = "this queue is empty"
bool IntQueue::isEmpty( ) {
  return (first == NULL);
}
```


Queue Implementation (3)

```
// insert item at rear of this queue
void IntQueue::enqueue(int item) {
   Node * p = new Node;
   p->val = item;
   p->next = NULL;
   if (first == NULL) {
      // insert first node in empty queue
      first = p; last = p;
   } else {
      // insert new node in non-empty queue
      last->next = p;
      last = p;
}
```


Queue Implementation (4)

```
// return item at front of this queue and remove it
// pre: queue is not empty.
int IntQueue::dequeue( ) {
   assert ( !isEmpty( ) );
   Node * p = first;
   int val = first->val;
   first = first->next;
   delete p;
   if (first == NULL) last = NULL;
   return val;
}
// return copy of first item in this queue
// pre: queue is not empty
int IntQueue::front( ) {
   assert ( !isEmpty( ); )
   return first->val;
}
```