Justin Ciocoi

Nov. 1, 2023

CSCI 373 Textbook Notes

Chapter 5: Stacks, Queues, and Dequeues

5.1: Stacks

- A *stack* is a container of objects that are inserted and removed according to the *last-in, first-out (LIFO)* principle
- The name, "stack," was derived from the metaphor of a stack of plates in a spring-loaded cafeteria plate dispenser
- **5.1.1: The Stack Abstract Data Type**
  - Stacks are fairly simple in the world of data structures, as they are one of the simplest, however they are also among the most important data structures due to their prevalence in application development and software engineering
  - Formally, a stack is an abstract data type that supports the following operations
    - `push(e)` : Insert element `e` at the top of the stack
    - `pop()` : Remove the top element form the stack
    - `top()` : Return a reference to the top element on the stack, without removing it
  - Additionally, we can define the following operations
    - `size()` : Return the number of elements in the stack
    - `empty()` : Return true if the stack is empty and false otherwise
- **5.1.2: The STL Stack**
  - The *Standard Template Library* provides programmers with an implementation of a stack
  - This implementation is based on the STL `vector` class

    - In order to declare an object of type `stack` it is necessary to first include the definition file, which is called "stack"
    - As with `string` or `vector` types, you must use `std::stack` or provide a `using` statement at the beginning of your code
    - For example, the following declares a stack of integers

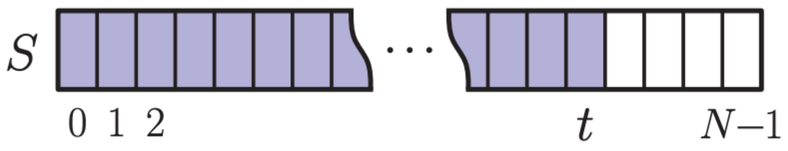
        -
          ```
          #include <stack>

          using std::stack;

          stack<int> myStack;
          ```
    - The type of the elements within a stack is referred to as the stack's *base type*
    - An STL stack dynamically resizes itself as new elements are pushed on
    - The functions for an abstract stack type that we defined earlier also apply to the STL implementation, with one small difference
        - The functions `pop()` and `top()` do not throw an exception when applied to an empty stack and it is thus up to the programmer to be sure that such illegal memory access does not occur
- 5.1.3: A C++ Stack Interface
    - Let us display an example of a C++ stack interface for a templated data type, `E`

        -
          ```
          template <typename E>
          class Stack
          {
              public:
                  int size() const;
                  bool empty() const;
                  const E& top() const throw(StackEmpty);
                  void push(const E& e);
                  void pop() throw(StackEmpty);
          };
          ```
    - Here, we can note that the `size()`, `empty()`, and `top()` functions are all declared to be constant, which informs the compiler that they do not alter the contents of the stack

- We should also note that the `pop()` operation does not return the popped value, and if the user wants to know this value, it is necessary to perform a `top()` operation first
- We can also see the error conditions for `pop()` and `top()`, and we can define the class `StackEmpty` in the code fragment below

  -
    ```
    //Exception for stack
    class StackEmpty : public RuntimeException
    {
        public:
            StackEmpty(const string& err) : RuntimeException(err)
            {}
    };
    ```

- **5.1.4: A Simple Array-Based stack Implementation**



- We can implement a stack easily by storing its elements in an array
- Specifically, this implementation would contain an N- element array, S, as well as an integer variable, t, that gives the index of the top element in array S
- To illustrate, let us first show the interface for the ArrayStack class

  -
    ```
    template <typename E>
    class ArrayStack
    {
            enum{DEF_CAPACITY = 100};
        public:
            ArrayStack(int cap = DEF_CAPACITY);
            int size() const;
            bool empty() const;
            const E& top() const throw(StackEmpty);
            void push(const E& e) throw(StackFull);
            void pop() throw(StackEmpty);

        private:
            E* S;
            int capacity;
            int t;
    }
    ```

- Now, let us implement the member functions of the above class in C++

    -
      ```
      template <typename E> ArrayStack<E>::ArrayStack(int cap)
      : S(new E[cap]), capacity(cap), t(-1) {}

      template <typename E> int ArrayStack<E>::size() const
      {
          return (t+1);
      }

      template <typename E> bool ArrayStack<E>::empty() const
      {
          return (t<0);
      }

      template <typename E> const E& ArrayStack<E>
      ::top() const throw(StackEmpty)
      {
          if(empty()) throw StackEmpty("Top of Empty Stack");
          return S[t];
      }

      template <typename E> void ArrayStack<E>
      ::push(const E& e) throw(StackFull)
      {
          if(size()==capacity) throw StackFull("Push to Full Stack")
          S[++t] = e;
      }

      template <typename E> void ArrayStack<E>
      ::pop() throw(StackEmpty)
      {
          if(empty()) throw StackEmpty("Pop from Empty Stack");
          --t;
      }
      ```

- Before moving on, we should also consider the time complexity of the various stack class member functions)

Operation	*Time*
size	$O(1)$
empty	$O(1)$
top	$O(1)$
push	$O(1)$
pop	$O(1)$

- **5.1.5: Implementing a Stack with a Generic Linked List**
    - In this section, we will explore how we can implement the stack abstract data type in C++ using the generically linked list sLinkedList, which was defined earlier in the text
    - Instead of using a generic templated function, we will use a string type sLinkedList for ease of understanding and avoiding syntactic messiness
    - Here is the interface for this class

        -
          ```
          typedef string Elem;
          class LinkedStack
          {
              public:
                  LinkedStack();
                  int size() const;
                  bool empty() const;
                  const Elem& top() const throw(StackEmpty);
                  void push(const Elem& e);
                  void pop() throw(StackEmpty);
              private:
                  sLinkedList<Elem> S;
                  int n;
          };
          ```
    - Here, the principal data member is the generic linked list of type `<Elem>`, called `S`
        - Since this class does not provide a member function `size()`, we will store the current size in the variable `int n`
    - Now, let us define the implementation of the above interface

-
  ```cpp
  LinkedStack::LinkedStack()
  : S(), n(0) {}

  int LinkedStack::size() const
  {
      return n;
  }

  bool LinkedStack::empty() const
  {
      return n==0;
  }

  const Elem& LinkedStack::top() const throw(StackEmpty)
  {
      if(empty()) throw StackEmpty("Top of Empty Stack")
      return S.front();
  }

  void LinkedStack::push(const Elem& e)
  {
      ++n;
      S.addFront(e);
  }

  void LinkedStack::pop() throw(StackEmpty)
  {
      if(empty()) throw StackEmpty("Pop from Empty Stack")
      --n;
      S.removeFront();
  }
  ```

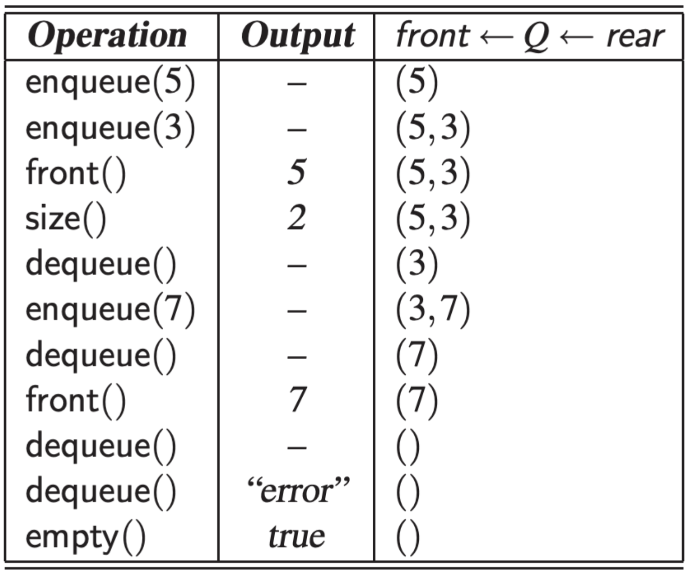
- **5.1.6: Reversing a Vector Using a Stack**
  - We can use the stack data type to reverse the elements in a vector, thereby producing a non-recursive algorithm for the array reversal problem
  - Here, the basic idea is to push all the elements of the vector in order into a stack and fill the vector back up by popping the elements off of the stack
  - Below is the implementation in C++ of this algorithm

-
  ```
  template <typename E>
  void reverse(vector<E>& V)
  {
      //init stack
      ArrayStack<E> S(V.size());

      //push elements onto stack
      for(int i=0;i<V.size();i++)
          S.push(V[i]);

      //pop elements off of stack and back onto vector
      for(int i=0;i<V.size();i++
      {
          V[i] = S.top();
          S.pop();
      }
  }
  ```

- **5.1.7: Matching Parentheses and HTML Tags**
  - Arithmetic expressions and other code in C++ will often contain various pairs of grouping symbols, including
    - (and)
    - [and]
    - { and }
  - When using these symbols, each opening symbol must correspond with an appropriate closing symbol
  - Here we can use a stack, and each time we encounter an opening symbol, we can push it onto the stack
  - Then, when we encounter a closing symbol, we can pop the top of the stack and compare it to ensure they are the same grouping symbols
  - Assuming that the `push()` and `pop()` operations are implemented to run in constant time, $O(1)$, this algorithm will run in linear time $O(n)$

5.2: Queues

- Another fundamental type of data structure is the *queue*, which is a close relative of the stack class
- A queue is a container of elements that are inserted and removed according to the *first-in first-out (FIFO)* principle
- Usually, we say that elements enter the queue at the *rear* and are removed from the *front*
- **5.2.1: The Queue Abstract Data Type**
  - Formally, the queue abstract data type defines a container that keeps elements in a sequence as defined above
  - The *queue* abstract data type supports the following operations
    - `enqueue(e)` : Insert element `e` at the end of the queue
    - `dequeue()` : Remove element at the front of the queue
      - Error occurs if the queue is empty
    - `front()` : Return, but do not remove, a reference to the front element in the queue
      - Error occurs if the queue is empty
    - `size()` : Return the number of elements in the queue
    - `empty()` : Return true if the queue is empty and false otherwise
  - We can observe a table showing a series of queue operations and their expected outputs



Operation	***Output***	*front* $\leftarrow Q \leftarrow$ *rear*
enqueue(5)	–	(5)
enqueue(3)	–	(5,3)
front()	*5*	(5,3)
size()	*2*	(5,3)
dequeue()	–	(3)
enqueue(7)	–	(3,7)
dequeue()	–	(7)
front()	*7*	(7)
dequeue()	–	()
dequeue()	*“error”*	()
empty()	*true*	()

- **5.2.2: The STL Queue**
  - The Standard Template Library (STL), provides an implementation of a queue
  - The underlying implementation here, as with the STL stack class, is based on the underlying vector class in the standard template library
  - Similar to the stack data type, the below code fragment declared a queue of float type variables

    -
      ```cpp
      #include <queue>
      using std::queue;

      queue<float> myQueue;
      ```
  - As with vectors and stacks, the STL queue dynamically resizes itself as new elements are added
  - The STL queue class supports roughly the same operators as thee abstract interface we defined above, with slight semantic and syntactic differences
    - `size()` : Return the number of elements in a queue
    - `empty()` : Return true f the queue is empty and false otherwise
    - `push(e)` : Enqueue e at the rear of the queue
    - `pop()` : Dequeue the element at the front of the queue
    - `front()` : Return a reference to the element at the queue's front
    - `back()` : Return a reference to the element at the queue's back
  - Unlike the abstract interface, the STL queue class provides access to both the front and the back of the queue
  - Like the STL stack class, no exceptions are thrown for full or empty lists and it is up to the programmer to ensure that such illegal memory access does not occur
- **5.2.3: A C++ Interface**
  - Here we can define a templated interface for the abstract queue data type

-
  ```
  template <typename E>
  class Queue
  {
      public:
          int size() const;
          bool empty() const;
          const E& front() const throw(QueueEmpty);
          void enqueue(cinst E& e);
          void dequeue() thro(QueueEmpty);
  };
  ```
- We can define the error conditions by defining the QueueEmpty class, which is a subclass of the RuntimeException class

  -
    ```
    class QueueEmpty : public RuntimeException
    {
        public:
            QueueEmpty(const string& err) : RuntimeException(err)
            {}
    };
    ```

- **5.2.4: A Simple Array-Based Implementation**
  - Let us consider an implementation of an array, Q with N elements, such that we can represent Q in a queue
  - The main issue with such an implementation is deciding how to keep track of the front and rear of the queue
  - One possibility to solve this issue is to adapt our implementation from the stack class such that $Q[0]$ is the front of the queue, and the queue will grow from there
    - This, however, is not efficient since it requires we move every element in the array when we perform a `dequeue()` operation
  - *Using an Array in a Circular Way*
    - Let us define three variables, f, r, and n
      - f is the index of the cell of Q storing the front of the queue
        - If the queue is not empty, then this is the index of the element that will be removed by `dequeue()`
      - r is the index of Q following the rear of the queue
        - If the queue is not full, this is where new elements will be inserted by `enqueue()`
      - n is the current number of elements in the queue
    - Initially, we set $n = 0$ and $f = r = 0$, which indicates an empty queue

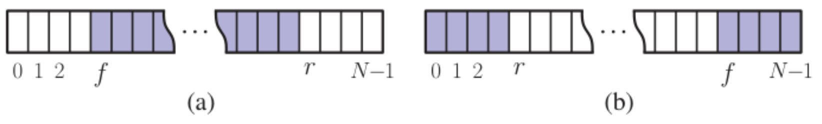
    - When we dequeue an element, we decrement n and increment f to the next cell in Q
    - When we enqueue an element, we increment r and increment n
    - This approach will allow us to implement the dequeue and enqueue functions in constant time
    - However, enqueuing and dequeuing a single element N, leads us to $f = r = N$, which gives us an out of bounds error for enqueue operations despite the availability of space in the queue
    - Thus, we must implement the array in a circular fashion, as shown below
    - 

      Figure 5.4: Using array Q in a circular fashion: (a) the "normal" configuration with $f \leq r$; (b) the "wrapped around" configuration with $r < f$. The cells storing queue elements are shaded.
  - *Using the Modulo Operator to Implement a Circular Array*

    -
      ```
      int size()
      {
          return n;
      }
      ```

    -
      ```
      bool empty()
      {
          return (n==0);
      }
      ```

    -
      ```
      Elem front()
      {
          if empty()
              throw QueueEmpty;
          return Q[f];
      }
      ```

-
  ```
  void dequeue()
  {
      if empty()
          throw QueueEmpty;
      f=(f+1)%N;
      n--;
  }
  ```

-
  ```
  void enqueue()
  {
      if(size()==N)
          throw QueueFull;
      Q[r] = e;
      r = (r+1)%N;
      n++;
  }
  ```

- 5.2.5: Implementing a Queue with a Circularly Linked List

-
  ```
  //constructor
  LinkedQueue::LinkedQueue()
  : C(), n(0) {}

  //number of items
  int LinkedQueue::size() const
  {
      return n;
  }

  //is the queue empty?
  bool LinkedQueue::empty() const
  {
      return n==0;
  }

  //return the front element
  const Elem& LinkedQueue::front() const throw(QueueEmpty)
  {
      if(empty())
          throw QueueEmpty("front of empty queue");
      return C.front();
  }

  //enqueue element at rear
  void LinkedQueue::enqueue(const Elem& e)
  {
      C.add(e);
      C.advance();
      n++;
  }

  //dequeue element at front
  void LinkedQueue::dequeue() throw(QueueEmpty)
  {
      if(empty())
          throw QueueEmpty("front of empty queue");
      C.remove();
      n--;
  }
  ```