平成 29 年 5 月 25 日

知的システム構成特論

Data Structures and Algorithms, Chapter 4 (pp.169-195)

Stacks, Queues, and Recursion

西田研究室 17344229 牛 佳偉

4.3 Queues

A Queue is a receptacle which can inserted and removed objects from the first-in first-out(FIFO) principle. Elements can insert in at any time, but cannot remove when the queue is empty. The elements come in the queue at the rear, as well as remove from the front.

4.3.1 The Queue Abstract Data Type

Initially, the queue abstract data type defines a container which keeps objects in a sequence. Elements are inserted only need the first element in the queue.

The queue abstract data type (ADT) as follows:

enqueue(o): insert element o at the rear of the queue

dequeue (o): remove from the queue the element at the front, an error happens when the queue is empty

size (): return the number of the elements in the queue

isEmpty (): if the queue is empty, it will return a Boolean

first (): only return the front of element in the queue

4.3.2 A simple Array-Based Implementation

The main problem with this is deciding how to keep track of the front and rear of the queue. One solution is that using the stack implementation, initially, let the Q(0) be the front of the queue, then add elements from here, it is an inefficient solution since it lets us move all the elements forward one array cell every time when we perform a dequeue() function. Another solution if we want to achieve constant time for each queue is that we define two variables f and r, when removing the element from the front of the queue, we can increment f to index the next cell, as well as when adding an element, we can increment r to index the next available cell in Q.

Since we cannot tell the differences between a full queue and an empty one, we have to introduced a new exception, named QueueFullException, meaning that no more elements can be inserted in the queue.

The array-based stack implementation has a disadvantage that we automatically set the capacity of the queue to be some number like N, but in real application, we may need more than or less than N, if we can estimate the number which will be in the queue, then the array-based will be efficient.

4.3.3 Memory Allocation in C++

Memory can be allocated by using the new operator, built in C++.

4.4 Linked List

The array-based stack has a drawback that it has to decide the size N at the first time, but in this section, singly linked lists do not meet this situation.

4.4.1 Singly Linked Lists

A linked list is a collection of nodes that form a linear ordering together. Each node has a data number called element as well as a pointer called next.

Link hopping is that moving from one node to another uses the next pointer.

The first and the last node are called head and tail, and tail has a null next pointer, indicating the end of the list.

The advantage of a singly linked list is that it does not have a predetermined fixed size and uses space proportional to the number of its elements.

Each node of a singly linked list contains of two fields which the element value associated with the node as well as a pointer to the next node in the linked list. Additionally, the linked list class itself maintains a pointer to the head of the list.

Using a singly linked list, we can easily insert or delete elements at the head of the list.

4.4.2 Implementing a Stack with a Singly Linked List

Basically, the top of the stack can be either at the head or the tail of the list. Since we can only insert and delete elements at the head, so it is more efficient to have the top of the stack at the head.

Housekeeping Functions

Since the LinkedClass allocates memory, we have to take care of our memory management through defining a copy constructor, assignment operator, a destructor. Initially, we define two function called removeAll() and copyFrom(), which are very useful for copying and deleting.

4.4.3 Implementing a Queue with a Singly Linked List

It is efficient to choose the front of the queue to be at the head of the linked list and the rear of the queue to be at the tail of the linked list. The structure of the linkedQueue is similar to the structure of the linkedStack, except that we need to pursue pointers to both the head and the tail of the linked list. The functions in the singly linked list queues are more complicated so we need to pay attention to some cases, when the queue is empty before inserting an element or the queue becomes empty after deleting an element.

4.5 Doubled-Ended Queues

A Doubled-Ended Queue is able to insertion and deletion at both the front and the rear of the queue.

4.5.1 The Deque Abstract Data Type

insertFirst(o): insert an element o at the beginning
insertLast(o): insert an element at the end
removeFirst(): remove the first object
removeLast(): remove the last object
first(): return the first object
last(): return the last object
size(): return the number of the DQ
isEmpty(): determine whether DQ is empty

4.5.2 Implementing a Duque with a Doubly Linked List

It is more efficient to use the Doubly Linked List than Singly linked list since the Deque can insertion and deletion at both ends of the list. A node in doubly linked list owns two pointers- a next link which points to the next node and a prev link which points to the previous node. In doubly linked list, we will add two sentinel nodes at the beginning and the ending of the list,which called the header and the trailer. These nodes do not store elements and that are not counted in the size() function. The advantage of these sentinels is that they simplify the code for insertion and deletion operation.

4.5.3 The Adapter Design Pattern

Table 4.1: Implementing a stack with a Duque.

Stack Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
top()	first()
push(o)	insertFirst(o)
pop()	removeFirst()

Table 4.2: Results of grasp planning.

Queue Method	Deque Implementation
size()	size()
isEmpty()	isEmpty()
front()	first()
enqueue(o)	insertLast(o)
dequeue()	removeFirst()

4.6 Smaple Case Study Application

we introduce a stack application used to solve the financial analysis problem.