Programming techniques

Week 5: Stack & Queue

03/2024

What is next?

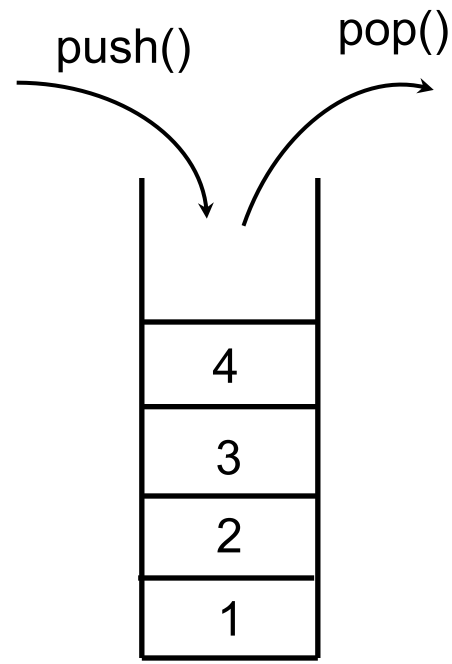
- Stack
- Queue
- Review for the Midterm

Stack

- We have talked about lists of data much of the term
- One common use of a list is to represent a stack abstraction
- Stacks allow us to add data and remove data at only one end (called the top)
- We can push data onto the top and pop data off of the top (Last In First Out - LIFO)

Stack

- Main operations
  - **`push(x)`**: add an element x to the top of the stack
  - **`pop()`**: get and remove the top element from the stack
- Thus, the elements come in and out of the stack based on the order of Last In First Out (LIFO)



Stack using Singly Linked List

- Our stack

```
Node* pStack;
```

- Operations

```
void push(Node* &pStack, int x);
bool pop(Node* &pStack, int& x);
bool isEmpty(Node* pStack);
```

Stack

- Implementing **`push`** and **`pop`** with a singly linked list data structure
  - represent very simple insert and remove algorithms
  - in fact, **`push`** is the same as adding at the beginning of a singly linked list
  - and, **`pop`** is the same as removing at the beginning of a singly linked list
  - why wouldn’t we add/remove at the end (or tail) instead?

Stack

- why wouldn’t we add/remove at the end (or tail) instead?
  - pushing at the tail would be just as easy and efficient as pushing at the head <u>iff</u> we kept a <u>tail pointer</u>.
  - but, popping at the tail would <u>require</u> that we traverse to the (tail-1) node...regardless of whether or not there was a tail pointer.
  - a doubly linked list is <u>not</u> the answer

Stack Implementation with Data Abstraction

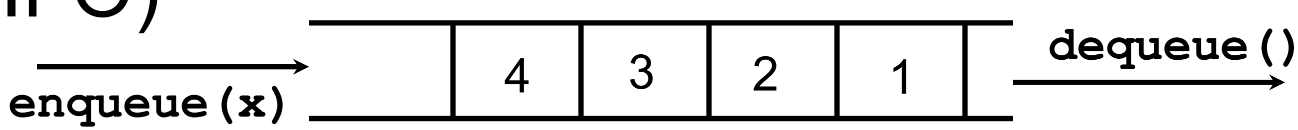
```
struct Stack {
   Node* pStack;
   void push(int x);
   bool pop(int& x);
   bool isEmpty();
};
void Stack::push(int x) {
     //your code here
}
```

Queue

- Another common use of a list is to represent a queue abstraction
- Queues allow us to add data at one end (the rear) and remove data at the other end (at the front)
- We can enqueue data at the rear and dequeue data at the front (First In First Out - FIFO)

Queue

- Main operations
  - **`enqueue(x)`**: add an element x to the end of the queue
  - **`dequeue()`**: get and remove the first element from the queue
- Thus, the elements come in and out of the queue based on the order of First In First Out (LIFO)



Queue

- Our queue

```
Node* qQFront, *qQRear;
```

- Operations

```
void enqueue(Node* &qQFront,
             Node* &qQRear, int x);
bool dequeue(Node* &qQFront,
             Node* &qQRear, int& x);
bool isEmpty();
```

Queue

- Implementing enqueue and dequeue with a linear linked list data structure
  - are also simple
  - but, should the “rear” pointer point to the first or the last node? And, should the “front” pointer point to the first or the last node?
  - draw the pointer diagrams for either way and decide which would traverse less...

Queue

- enqueue should add at the rear - the tail
- dequeue should remove at the front - the head
- Why?
  - enqueuing at the front or rear is equally easy and efficient
  - but, dequeuing at the rear requires that we traverse to the “last-1” node (but a doubly linked list would be <u>overkill</u>). Luckily, dequeuing at the front is simple and efficient

Review for the midterm exam

- Topics
  - Pointer
  - Dynamically allocated array
  - Singly linked list
  - Doubly linked list
  - Circular linked list
  - Ordered linked list
  - Stack
  - Queue
- You must understand the topics deeply and keep practicing a lot at home