Discussion 2C Notes (Week 3, January 21)

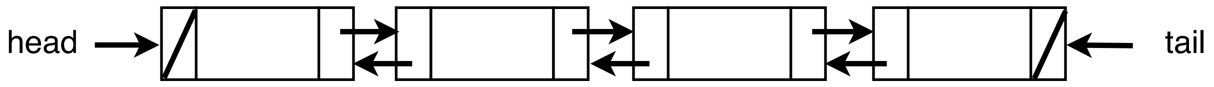
TA: Brian Choi (schoi@cs.ucla.edu)
Section Webpage: http://www.cs.ucla.edu/~schoi/cs32

Abstraction

In Homework 1, you were asked to build a class called Bag. Let us look at it from the user's point of view. All the user needs to know is that he/she can insert items into the Bag, remove them, count them, and iterate through the items in the Bag, without having to know how exactly the items are stored within the Bag. There is more than one way of implementing the Bag class with the same interface and behavior, your solution to Homework 1 being one of them. Project 2 will let you implement the Bag class in a different way, without altering the interface; that is, a program that uses the previous version of the Bag class should work with the new version without any modification. Such a division of **interface** and **implementation** is called **abstraction**. The user is said to **be abstracted from** the implementation. We'll do this for many of the data structures we'll see in this class.

Doubly Linked Lists

We studied linked lists last week. Each node in a linked list keeps a pointer to the next node. This time, let us create a linked list whose nodes have two links each -- one to the next node (`next`) and another to the previous one (`prev`). It's called a **doubly linked list**, and it looks like this:

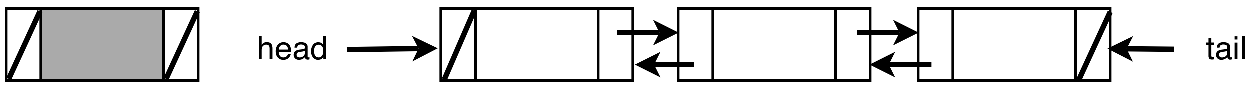


doubly-linked list with 4 nodes

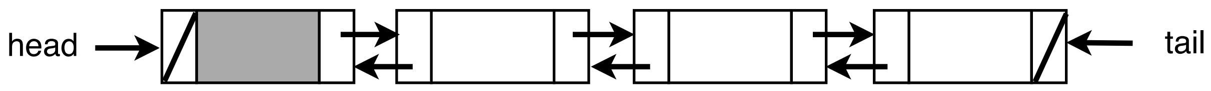
Insertion

Suppose I want to add a new node. Where shall it be? You are free to add it anywhere you want in the list, but since we have the head pointer that points to the first node, it might be easier to put it in front. Last week, for a (singly) linked list, we simply created a new node, made the new node point to the first one, and then updated the head pointer. However, a node in the doubly linked list has an extra pointer `prev`, which complicates our problem.

Before:



After:



Let us first figure out how to set `next` and `prev` for the new node. Because it is going to be the first one in the list, there will be no "previous" node. Thus we set `prev` to `NULL`. `next` should point to the node that is in front before the insertion, which is currently pointed by `head` (and `head` may point to `NULL`, no problem.) So for the new node,

1) Set new node's `prev` to `NULL`.
2) Set new node's `next` to `head`.

There are two more things we need to update at this stage. `head` is pointing to a wrong one, and the original head node's `prev` should be updated to point to the newly added node. However, there may be no node in the list, in which case there is no such `prev` to set. Therefore, we set `prev` only if the list is not empty before the insertion.

3) If the list was empty before the insertion, just make `head` point to the new node.
4) If not, set `head`'s `prev` to the new node, and then make `head` point to the new node.

If the `tail` pointer is present and is pointing to `NULL`, the list must have been empty. We should set `tail` to point to the new node in this case. Otherwise, since we added a new node to the front, `tail` shouldn't change. This completes the insertion process.

Question: Think about how the algorithm should change if the new node should go to the end of the list.

Search

You can apply the same searching algorithm you used for (singly) linked lists.

Removal

Removing a node involves searching the node to remove, fixing the links such that the resulting list is a valid doubly linked list, and actually deleting the node from the memory. There can be four cases, though, and for each case we have to do different things. The node's location determines our action. Suppose we performed a search and have a pointer `p` that points to the searched node. (Is this enough? We needed two pointers for a singular linked list.) Let us do the following:

- We can check if the node is the head by checking `p == head`. Let this boolean be **A**.
- Also, check if the node is the tail by checking `p == tail`. Let this boolean be **B**.

Case 1 (A, but not B): The node is the head of the list, and there is more than one node in the list.

Case 2 (B, but not A): The node is the tail of the list, and there is more than one node in the list.

Case 3 (A and B): The node is both the head and the tail of the list (i.e. it is the only node).

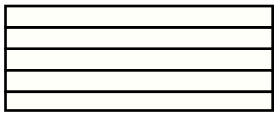
Case 4 (not A and not B): The node is in the middle of the list.

I will leave what to do for each case as an exercise. I suggest you draw a before-and-after diagram like the one on the previous page, and figure out which links should point to which. Again, remember to delete the node as the last step, since you might delete a pointer that you must know in order to fix the broken link. Some of the above cases can be combined when you actually implement it.

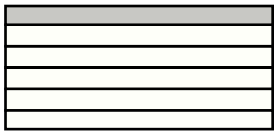
NOTE: The textbook presents a **circular doubly linked list**, which uses a dummy node to get rid of special cases. The doubly linked list here is different from the textbook's version. The linear version is presented in this handout for the sake of diversity (so you get a chance to see and deal with a different version). For Project 2, you are allowed to use either one. In fact, most people will choose to implement a circular linked list, because it is slightly easier, with less special cases to worry about.

Stacks

A **stack** is just another data structure we'll see. It doesn't mean anything more than what you think it means. It is a "bunch" of items stacked on top of another. Suppose I have a stack of 5 items.



What should happen if I add an item to this stack? Naturally, it should go onto the top of the stack.



Now the stack has 6 items. Now maybe I want to pick one item up from the stack (imagine picking up a newspaper from a pile of them.) Which one should I pick up? It is probably easiest to take the one on the top. As a result, the state of the stack will return to what we had before -- one with 5 items.

In summary, a stack is a "Last-In-First-Out" data structure.

Interface

A stack interface should support the two operations described above:

1) `push()` : add a new item onto the stack
2) `pop()` : remove the top item from the stack

Remember how we talked about every data structure needs to support three operations, namely "insert," "remove," and "search"? `push()` and `pop()` correspond to "insert" and "remove" operations, respectively, and they define what a stack is. A stack features a limited "search" operation -- you can only look at the top item. See the next page for an example `Stack` interface.

Implementation

Like other ADTs, there are many ways of implementing a stack. Let us use an array as an example. Just to make the programming task easier, let us keep a counter for the number of items currently in the stack.

```
private:
     ItemType m_items[MAX_ITEMS];  // MAX_ITEMS is our capacity
     int      m_numItems;          // assume it's initialized to 0
```

For now, suppose that `ItemType` is `typedef`-ed to be `int`. Now suppose this is what I have in main:

```
Stack s;
s.push(18);
```

```
class Stack
{
    public:
        bool push(const ItemType& item); // true if successful
        ItemType pop();                  // pop
        bool empty() const;              // true if empty
        int  count() const;              // number of items
    private:
         // Some data structure that keeps the items.
};
```

Stack Interface

The natural place to insert the new value is the front (or the first available slot) of the array. (Notice this position is conveniently marked by `m_numItems`.) So we will add 18 to the position 0, and then increment `m_numItems`. After this, `m_items` will look like this:

18														

Question: Write the C++ implementation of `push()` function.

Now let us add more items to the array. Suppose my array is in the following state at some point:

18	30	22	43	7	5									

Question: Just looking at the array, what can you say about this stack? Which one was added most recently? Which one was added least recently?

Consider the above line of code. It calls `pop()` on `s` and store the value into `x`, if `s` is not empty.

```
ItemType x;
if (!s.empty())
    x = s.pop();
```

Question: What will the array look like after the execution of the above code? How about the values of `x` and `m_numItems`?


```
x =                    m_numItems =
```

Question: Provide the C++ implementation of `pop()` function.

Page 298 of the textbook has an array-based implementation of a stack similar to one we have looked here. It is also possible to implement it using a linked list. Notice that whichever form of implementation we use, we need two information: 1) how to check for emptiness, and 2) where the top is. In the array version, `m_numItems` does the job for us. For a linked list, we can use the head pointer to indicate the top position and `NULL` to check for emptiness.

Queues

A **queue** is very similar to a stack, but differs in that it is a FIFO (First-In-First-Out) structure.

```
class Queue
{
    public:
        bool enqueue(const ItemType& item); // push
        ItemType dequeue();                 // pop
        bool empty() const;                 // true if empty
        int  count() const;                 // number of items
    private:
         // some data structure that keeps the items
};
```

Queue Interface

Interface

A stack interface should support the two operations described above:

1) `enqueue()` : add a new item to the back of the queue
2) `dequeue()` : remove the item in front of the queue

Here is an example run, assuming `Item` is typedef-ed to be `int`.

```
Queue q;
q.enqueue(1);
q.enqueue(2);
cout << q.dequeue() << endl;  // prints 1
q.enqueue(3);
cout << q.dequeue() << endl;  // prints 2
```

Implementation

Again, let us use an array to implement it. Assume the array has a fixed capacity of `MAX_ITEMS`. We will insert items in the same order as we did for a stack, and suppose we have the following array at some point.

18	30	22	43	7										

18 is the first item that was added, or enqueued, and thus when we call `dequeue()`, 18 must be returned and removed. Then our resulting queue must begin with 30 and end at 7. The starting position has changed. When we add an item, say 5, then it should be added to the end of the array, after 7, thus shifts the "ending position." After `dequeue()` and `enqueue(5)`, we'll have:

	30	22	43	7	5									

This is different from the stack's case, where the starting position was always 0. For a stack, it was enough to keep the number of items, yet for a queue, we need an extra variable that keeps track of where the queue begins.

```
private:
     ItemType m_items[MAX_ITEMS];  // MAX_ITEMS is our capacity
     int      m_numItems;          // assume it's initialized to 0
     int      m_startPos;          // starting position
```

`m_startPos` marks the starting position, which we will shift by 1 (i.e. `m_startPos++`) at each `dequeue()` call. Then when we enqueue a new item, we can put the new item to the position `(m_startPos + m_numItems)`. Wait, is this correct? Consider the following situation: suppose we kept enqueueing and dequeueing, and at some point we have the following array:

											4	5	12	7

There are only 4 items in the queue, thus we should be able to add more items to the queue. Yet `(m_startPos + m_numItems)` points to a location that's out of bounds. Looking at the figure above, it seems we can start adding more items to the front of the array. So we must "wrap back" when we hit the end of the array. A CS32 student should be able to figure out how to do it with a simple trick.

Question: Give a C++ implementation of the function `enqueue()`.

Question: Give a C++ implementation of the function `dequeue()`.

Challenge(?) Question: Can you implement a queue if you're given two stacks of the same size, instead of an array?

Question: Do you think you can implement `Stack` and `Queue` with a linked list/doubly-linked list?