28. Stacks and queues

Herman Kamper

In the previous lecture we met the linked list: a chain of nodes, each holding some data and a pointer to the next node, with a pointer to the first node so that we can find the whole thing. We saw that we can create a node with `malloc` and slot it in anywhere we like, and that we can unlink a node and `free` it to hand its memory back.

That freedom is useful, but it is also more than most programs need. In practice, a plain linked list is not often used to store data; if you were writing the software for a bank, you would build something tailored to what the bank actually does. What *is* used constantly are two simplified versions of the linked list, where we deliberately restrict where things can be added and removed. These are the stack and the queue. In this lecture we look at both. They are not much harder than the linked list: they are the same nodes and the same pointer operations, just with fewer choices.

A quick recap of the linked list

A node holds some data and a pointer to the next node. A list is accessed through a pointer to its first node. The figure below summarises the previous lecture.

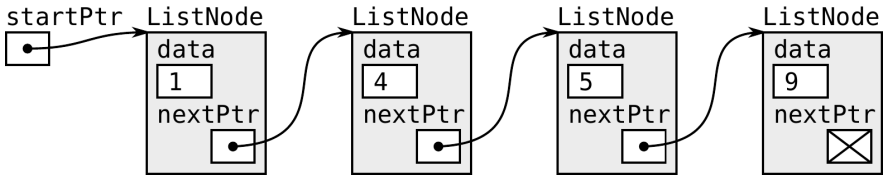

Figure 1: A linked list: a start pointer, nodes holding data, and a `NULL` in the last node.

One thing is worth stressing, which is easy to miss. The pointer to the first node is your only way into the list. If you lose it, you lose all of your data: you no longer know where the nodes sit, you cannot read them, and you cannot even `free` them. This is true of linked lists, stacks and queues alike.

Stacks

A *stack* is a linked list with a restriction: elements can only be added at the start of the list (called the *top* of the stack), and elements can only be removed from the top as well. Think of a pile of books. You can only put a new book on top of the pile, and you can only take a book from the top; you cannot pull one out from the middle.

Adding an element to a stack is called *pushing*, and removing one is called *popping*. Since the last thing that went in is the first thing that comes out, a stack is called a *last in, first out* (LIFO) structure. It is a very undemocratic system: whoever arrived last gets served first.

The node of a stack looks exactly like a linked-list node, and the stack is accessed through a pointer to its top node, which we call `topPtr`:

```
struct stackNode {
    int data;
    struct stackNode *nextPtr;
};
typedef struct stackNode StackNode;
```

The figure below shows a stack of four nodes with `topPtr` pointing at the top one.

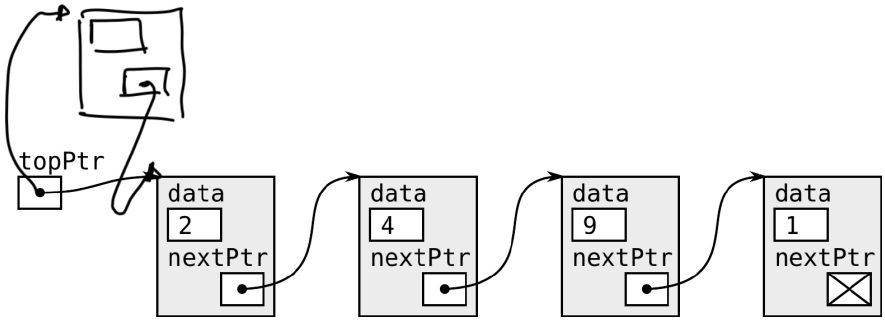

Figure 2: A stack of four nodes. The pointer `topPtr` points at the top node, which is the only one we can add to or remove from. The hand-drawn node above the stack is a new one that is about to be pushed.

When the stack is empty, `topPtr` is `NULL`. The first element pushed makes `topPtr` point at a single node, the next push puts a new node in front of it, and so on.

Stacks appear everywhere in computing. When you press undo in a text editor, the last thing that you did is the first thing that gets undone. Each action goes onto a stack as you make it, and undoing pops from the top. (Redo works by moving things back the other way.) A program's function calls are kept on a stack: when a function is called, its local variables and the place to return to are pushed, and when it returns they are popped. Compilers use stacks when evaluating expressions.

Pushing onto a stack

Suppose the stack already has the nodes holding 4, 9 and 1, with 4 on top, and we want to push the value 2. In the code, `value` holds the 2 and `topPtr` is the stack's top pointer. We use a temporary pointer `newPtr`:

```
StackNode *newPtr = NULL;

newPtr = malloc(sizeof(StackNode));
newPtr->data = value;
newPtr->nextPtr = topPtr;
topPtr = newPtr;
```

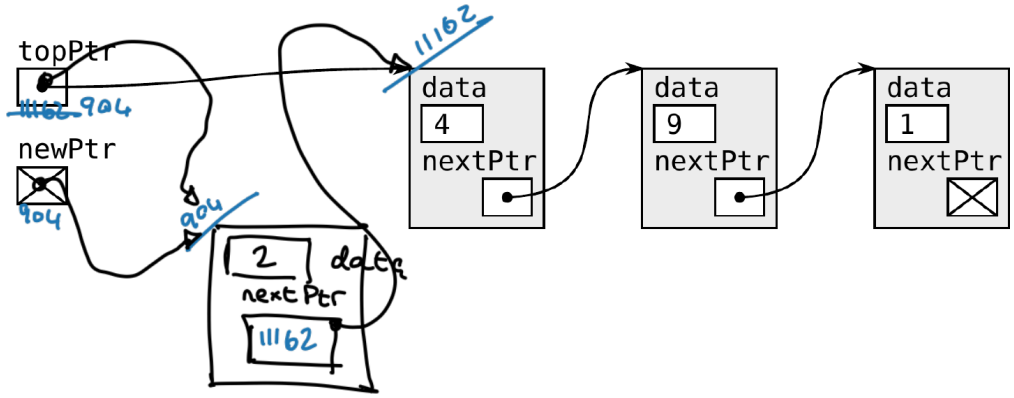

Figure 3: The four steps of a push, annotated with the addresses. The new node is at address 904; it ends up with 2 in its data member and the old top (address 11162) in its `nextPtr`, and `topPtr` is changed to hold 904.

Let us take the lines one at a time. The first line creates a pointer called `newPtr`, which is a little box that holds an address, and sets it to `NULL` so that it points at nothing yet.

The second line, the `malloc`, reserves enough bytes somewhere in memory for one `StackNode`. Suppose they start at address 904. `malloc` returns that address, and it is stored in `newPtr`. (How many bytes is that? An `int` takes four bytes and the pointer eight, so twelve, which the compiler will probably pad to sixteen.) The block now exists, but it has rubbish in it.

The third line goes to the box that `newPtr` points at, opens up its `data` member, and puts the value there; the arrow `->` is the shorthand for "go to the box that this pointer points at, and open this member". The C compiler knows the layout of a `StackNode`, so it knows which of the bytes it must write the value in.

The fourth line sets the new node's `nextPtr` to whatever is in `topPtr`. Say `topPtr` holds 11162, the address of the old top node. Then 11162 is copied into the new node's `nextPtr`, which means that the new node now points at the old top.

The last line is `topPtr = newPtr;`. It wipes out the 11162 in `topPtr` and puts 904 there instead. Now the stack points at the new node, and the new node points at the old top. This last line is really important: without it, the stack would still think the old node is the top, and the new node would be unreachable.

The result is shown below.

Notice that these steps always do the same thing, whether the stack has three nodes in it or none at all. If the stack is empty, `topPtr` is `NULL`, so the new node's `nextPtr` becomes `NULL`,

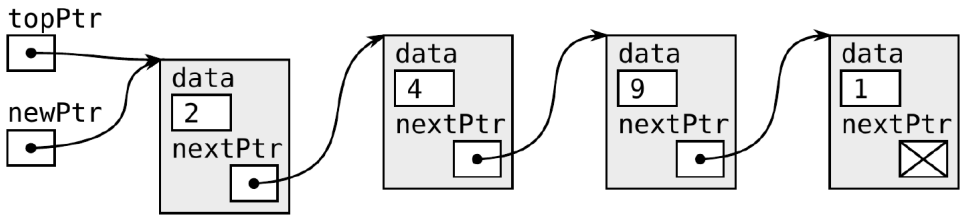

Figure 4: The stack after the push. The new node holding 2 is at the top, and `topPtr` points at it.

which is exactly right for the last node, and `topPtr` points at the new node. That is how a stack is built up in the first place.

If you find yourself lost in code like this, the main thing is to follow the boxes. Every variable is just a box. The only thing that is special about the boxes in this code is that they hold addresses; and when you see `->`, you go to the box that lives at that address.

Popping from a stack

Popping does two things: it gives you the value that was on top of the stack, and it removes that node from the stack. Stacks are nearly always used like this. For example, if the stack holds the actions that you have taken, you want to know what the last action was, and then you want to remove it.

So how do we write the code for this? Try it yourself on paper before you read on, starting from the picture of the stack above, with `topPtr` pointing at the node holding 2.

First, getting the value. The top node is where `topPtr` points, so we dereference it and read the `data` member. We can write that as `(*topPtr).data`, or in the shorter arrow notation as `topPtr->data`. The two are exactly the same.

```
int popValue = topPtr->data;
```

Second, removing the node. It looks like all we need to do is to move `topPtr` to the next node:

```
topPtr = topPtr->nextPtr;
```

This changes `topPtr` from the address of the old top (say 32) to the address held in its `nextPtr` (say 4). Now the stack starts at the second node, as we wanted. But there is a problem: what is holding the address 32 now? Nothing. The old top node still exists in memory, but we have no way to get to it, and we cannot `free` it. This is called a *memory leak*: the program is using bytes in memory that it will never use again. In a program that pushes and pops all day, it adds up.

The solution is to hold on to the old top in a temporary pointer before we move `topPtr`. Putting it all together:

```
int popValue;
StackNode *tempPtr;

tempPtr = topPtr;
popValue = topPtr->data;
topPtr = topPtr->nextPtr;
free(tempPtr);
```

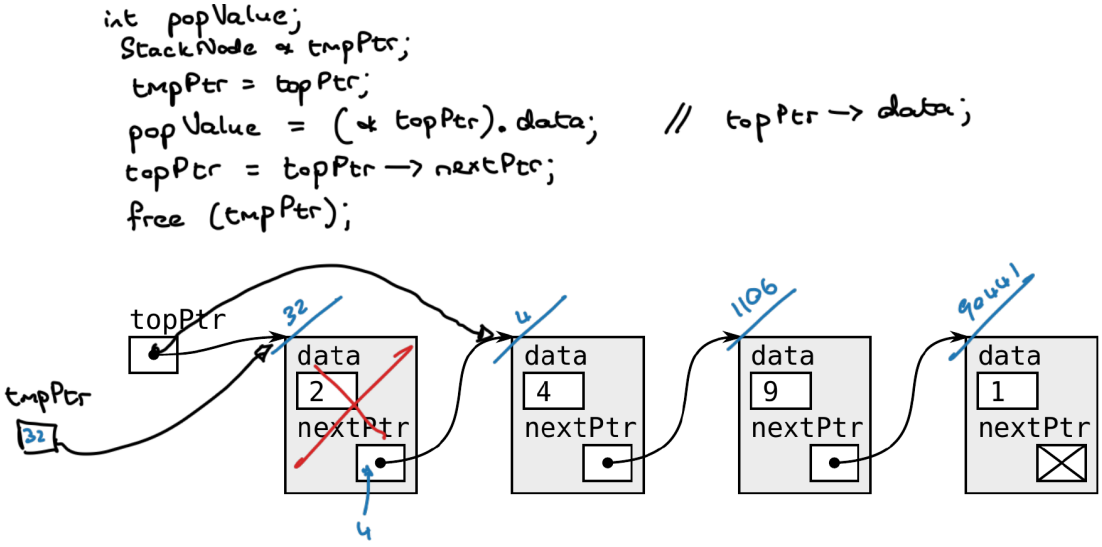

Figure 5: The steps of a pop. The temporary pointer `tempPtr` holds the address 32 of the old top before `topPtr` is moved on to the next node at address 4. The old node is then freed.

The picture shows what happens. We copy the address 32 into `tempPtr`. We read the 2 out of the top node into `popValue`. We change `topPtr` to hold 4. And finally `free(tempPtr)` hands the memory at address 32 back to the operating system.

You may wonder why changing `topPtr` does not also change `tempPtr`. They are two separate boxes. When we write `tempPtr = topPtr`, we copy the number in one box into the other, and from then on the two boxes just happen to hold the same number. It is like having two integers that both equal 32: changing one does not change the other. Dereferencing either one takes you to the same node, but they are not linked in any other way.

Before popping, a program should check that the stack is not empty, that is, that `topPtr` is not `NULL`. Dereferencing a null pointer would crash the program. We will return to this in the exercises.

Queues

A queue is another restricted version of a linked list, and one you know well. When you arrive at a coffee shop, where do you join the line? At the back. Who gets served first? The person who has been waiting the longest, at the front. The same thing applies to a queue in a program: new elements are added at the back and removed from the front. Since the first one in is the first one out, it is called a *first in, first out* (FIFO) structure.

The terminology for the two ends is the *head* (the front, where elements are removed) and the *tail* (the back, where elements are added). Adding to a queue is called *enqueueing* and removing is called *dequeueing*.

The node is the same as before:

```
struct queueNode {
    int data;
    struct queueNode *nextPtr;
};
typedef struct queueNode QueueNode;
```

The one new thing is that a queue needs *two* pointers: `headPtr` points at the first node, where we remove, and `tailPtr` points at the last node, where we add. The figure below shows a queue with three nodes, with the person who will get coffee first at the head.

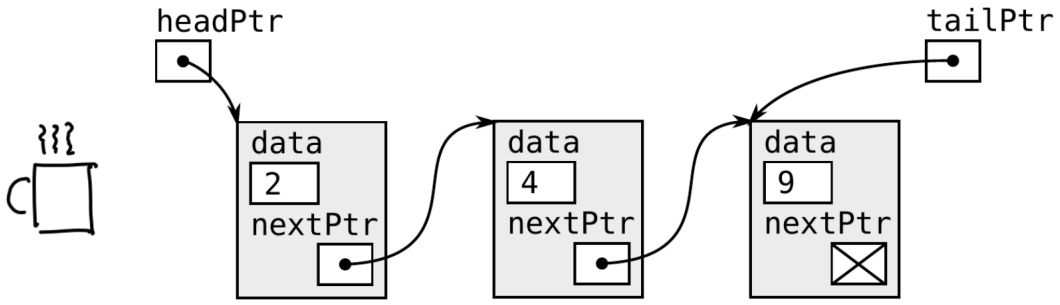

Figure 6: A queue. The pointer `headPtr` points at the front, where nodes are removed (the coffee is on this side), and `tailPtr` points at the back, where nodes are added.

Why do we need the tail pointer? Without it, we would have to start at the head and walk along the whole queue to find the end every time we want to add something. With `tailPtr`, we know where the end is immediately. This is the one extra thing a queue needs compared to a stack, which only ever touches one end.

Queues are used for things like print spooling (jobs are printed in the order they arrived), routing data packets in computer systems, and buffers. A call centre, where the call that has waited longest is answered first, is a queue.

Enqueueing

Suppose we have a queue with the values 2, 4 and 9, with 2 at the head, and we add the value 1 at the back:

```
QueueNode *newPtr = NULL;

newPtr = malloc(sizeof(QueueNode));
newPtr->data = value;
newPtr->nextPtr = NULL;
tailPtr->nextPtr = newPtr;
tailPtr = newPtr;
```

```
1 newPtr = malloc( sizeof( QueueNode ) );
2 newPtr->data = value;
3 newPtr->nextPtr = NULL;
4 tailPtr->nextPtr = newPtr;
5 tailPtr = newPtr;
```

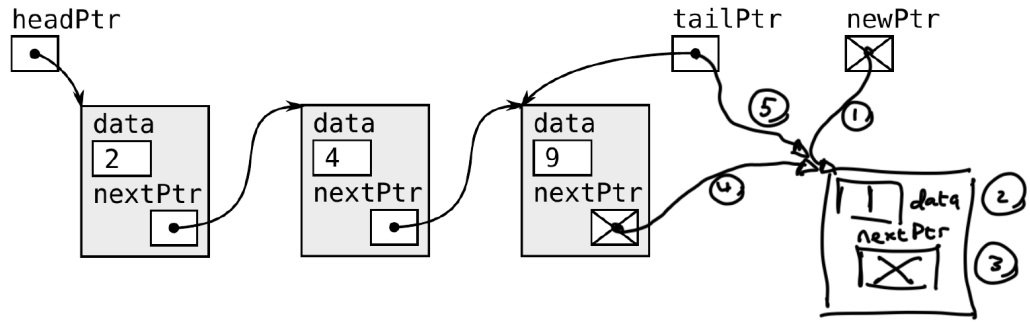

Figure 7: The five steps of an enqueue, numbered on the figure to match the lines of code. The new node is created (1), given its value (2) and a `NULL` link (3), linked from the old tail (4) and becomes the new tail (5).

Again, we go through the lines in the order of the numbers on the figure. The `malloc` reserves memory for a node and puts its address in `newPtr` (step 1). We put the value 1 in the node's `data` member (step 2). Then we set its `nextPtr` to `NULL` (step 3). It is important that we do this explicitly, because this node is about to become the new tail, and the tail's `nextPtr` must be `NULL` to mark the end of the queue.

Step 4 is `tailPtr->nextPtr = newPtr;`: go to the node the tail points at (the one holding 9), and replace the `NULL` in its `nextPtr` with the address of the new node. This links the old last node to the new one. Finally, step 5, `tailPtr = newPtr;`, makes the tail pointer point at the new node, so that the next enqueue will add after it. After all of this, the result is:

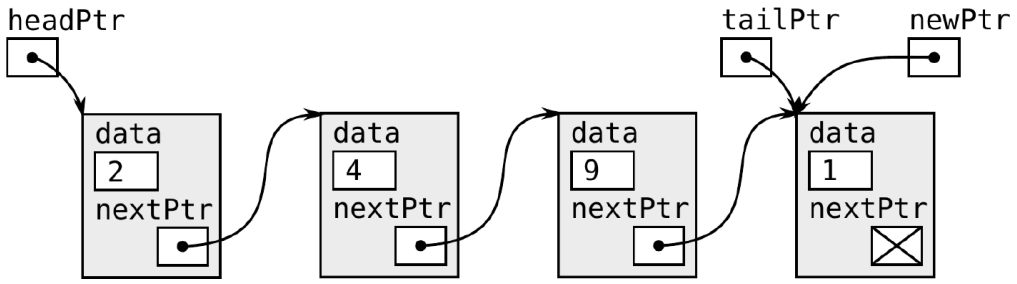

Figure 8: The queue after enqueueing the value 1. The head is unchanged, and the tail now points at the new node holding 1.

There is one special case. If the queue is empty, then both `headPtr` and `tailPtr` are `NULL`, and `tailPtr->nextPtr` would dereference a null pointer. In that case the new node becomes both the head and the tail:

```
if (headPtr == NULL) {
    headPtr = newPtr;
}
else {
    tailPtr->nextPtr = newPtr;
}
tailPtr = newPtr;
```

Dequeueing

Dequeueing is almost the same as popping. We want the value at the head (the customer's name, or whatever the data is), and then we remove the head node:

```
value = headPtr->data;
tempPtr = headPtr;
headPtr = headPtr->nextPtr;
free(tempPtr);
```

The figure below shows the state of the queue before these lines.

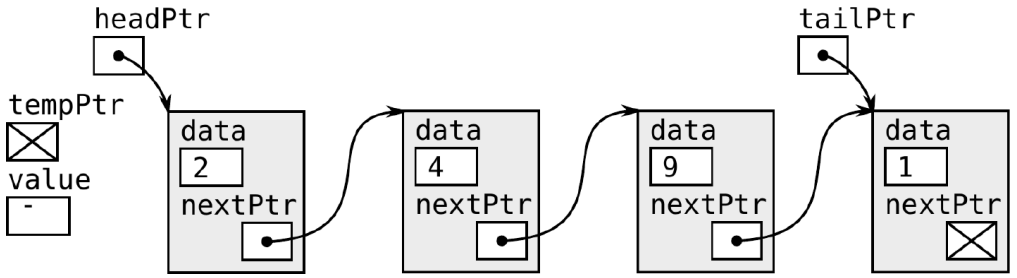

Figure 9: The queue before dequeueing. The variables `tempPtr` and `value` have no meaningful content yet.

The first line copies the 2 from the head node into `value`. The second saves the address of the head node in `tempPtr`, so that we will still be able to free it. The third moves `headPtr` along to

the second node. The fourth frees the old head node. The result is below.

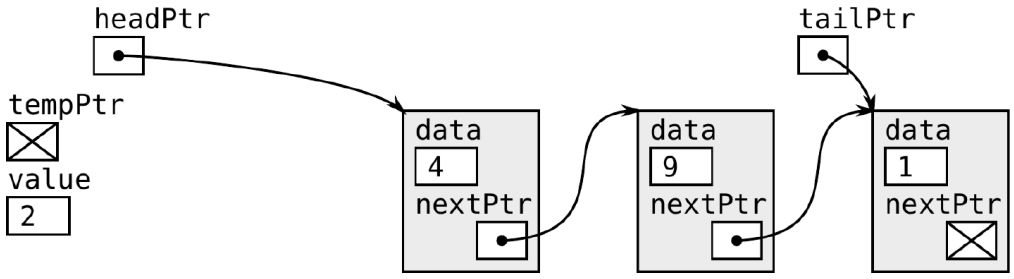

Figure 10: The queue after dequeueing. The value 2 is in `value`, the head points at the node holding 4, and the old head node has been freed.

Again there is a special case. If the node that was removed was the only one in the queue, then after the third line `headPtr` is `NULL`, and the queue is empty. But `tailPtr` still points at the node that has just been freed. So we must also set `tailPtr` to `NULL`:

```
if (headPtr == NULL) {
    tailPtr = NULL;
}
```

Forgetting this is a classic bug: the next enqueue would then see a dangling tail pointer, and write into memory that has been freed.

Why bother with stacks and queues?

You might ask what the point is. A stack is just a linked list where we only touch the top; a queue is just a linked list where we only touch the two ends. The code for a stack or queue is the same code as for a linked list, only with fewer cases.

That is exactly the point. The restriction *is* the feature. When you know that the only operations are push and pop, or enqueue and dequeue, they can be written very simply and are very efficient: each one takes the same few steps, however many elements are in the structure, with no searching. And just as importantly, a stack or a queue states clearly what the program is doing. When you are given a problem, you should ask which of the structures fits it, and then choose the most restricted one that does the job. If you need to undo the most recent action, use a stack. If you need to serve things in the order they arrived, use a queue. If you need to insert anywhere, then you need a linked list.

Changing the start pointer inside a function

The code so far has been written as bare lines inside `main`. In a real program you would put each operation in a function: `push`, `pop`, `enqueue` and `dequeue`. This leads to one place where many students get confused, so it is worth looking at carefully.

Suppose we write a function to push a value onto a stack. The function must change `topPtr`, because after a push the stack starts at a different node. But `topPtr` is a variable in `main`, and C

passes arguments by value: if we pass `topPtr` to a function, the function receives a copy of the address, and changing the copy does nothing to the original. This is the same problem that we had with `swap` earlier in the course, where we fixed it by passing the *address* of the variable that we wanted to change.

Here the variable that we want to change is itself a pointer, `StackNode *topPtr`. So we must pass its address, and the type of an address of a `StackNode *` is `StackNode **`. This is a pointer to a pointer to a `StackNode`. That is the origin of the double star below:

```
void push(StackNode **topPtr, int value)
{
    StackNode *newPtr = malloc(sizeof(StackNode));

    if (newPtr != NULL) {
        newPtr->data = value;
        newPtr->nextPtr = *topPtr;
        *topPtr = newPtr;
    }
}
```

Inside the function, `topPtr` is the address of the caller's variable. Writing `*topPtr` goes to that variable: it reads the current top, and assigning to it changes the top in `main`. The calling code in `main` passes the address:

```
StackNode *topPtr = NULL;

push(&topPtr, 1);
push(&topPtr, 2);
push(&topPtr, 3);
```

The two lines inside `push` that have `*topPtr` are exactly the third and fourth lines from our bare version, with `*topPtr` in the place of `topPtr`. In the same way, `pop`, `enqueue` and `dequeue` take pointers to the head and tail pointers.

A function that only reads the stack, for instance to print it, does not change `topPtr`, so a plain `StackNode *` is enough.

Summary

A stack and a queue are linked lists with restrictions on where nodes can be added and removed.

A stack is last in, first out. Nodes are added (pushed) and removed (popped) at the top, and one pointer, `topPtr`, is needed. To push, create a node, make it point at the old top and make `topPtr` point at it. To pop, save the old top in a temporary pointer, read its value, move `topPtr` to the next node and free the old top.

A queue is first in, first out. Nodes are added (enqueued) at the tail and removed (dequeued) from the head, so two pointers, `headPtr` and `tailPtr`, are needed. To enqueue, create a node

with a `NULL` link, make the old tail point at it and make `tailPtr` point at it. To dequeue, do the same as a pop at the head, and if the queue becomes empty, set `tailPtr` to `NULL` too.

In all of these, the pointer or pointers that give access to the structure must be kept up to date, or the data is lost. Check for an empty stack or queue before removing, free every node that you remove, and when a function needs to change one of these pointers, pass its address.

The code in this lecture shows the individual operations. To see a complete program that brings them together, work through the stack and queue examples in Deitel and Deitel (Sections 12.5 and 12.6) and write your own versions from scratch.

Exercises

1. Values 1, 2, 3 and 4 are pushed onto an empty stack in that order, and then two values are popped. Which values come out, and what is left on the stack from top to bottom?
2. The same values 1, 2, 3 and 4 are enqueued into an empty queue, and then two values are dequeued. Which values come out, and what is left?
3. What happens if `pop` is called on an empty stack with the code in this lecture? Rewrite it so that it checks first, and decide how the function should tell the caller that the stack was empty.
4. In `pop`, what goes wrong if the lines `topPtr = topPtr->nextPtr;` and `free(tempPtr);` are written in the opposite order?
5. Why does a queue need a tail pointer when a stack does not? How would you add to the back of a queue if only `headPtr` were available, and why would this be slower?
6. What would be the right data structure for each of the following: undoing edits in a text editor, a print server handling jobs from many users, checking that the brackets in an expression are balanced?
7. Write complete `push`, `pop`, `enqueue` and `dequeue` functions, including the empty cases, and use them in a small program that reads integers from the user and prints them in reverse order.