Lecture Summary

In this lecture, we learned about linked lists: why they are useful, how they work, and how to implement them using pointers.

1 Introduction

Arrays are nice, simple ways to store blocks of data, but we don’t always know the necessary array size right off the bat. How many spaces should we reserve/allocate? Allocating up to an arbitrary size (say, 1000) is not a good idea because it may be too much or too little memory for particular cases.

Dynamically sized arrays, which we saw in the previous lectures, give us a partial solution to the problem: at least, we don’t need to know the array size at compile time. But we do still need to know how large to make the array at run time when we declare it. How large do you think that Facebook should have made its user array when it started? If you have more and more customers arriving over time, it will be very hard to guess the “right” size.

In this class, we will be looking at many data structures that don’t need to know the required number of elements beforehand; rather, their size can dynamically change over time. Perhaps the easiest such data structure is called the linked list; it nicely illustrates many of the techniques we’ll see reappear many times later.

Consider how an array (dynamically or statically allocated) stores data. Whenever new users arrive, a really convenient way of dealing with it would be to just ask the operating system to expand our block of memory to the required size. However, there is no way to do this, and with good reason: the memory right after our array may be in use for some other data, so expanding our array may overwrite existing memory just after our array’s memory block. A way to “expand” our array may be instead to just get a second block of memory somewhere else, and put a pointer at the end of our first block to indicate where the other half is. This loses many of the advantages of arrays, such as being able to do simple arithmetic to look up elements. But at least, it would solve the problem in principle. Once we think about it that way, we could go to the extreme and make every element of our “array” point to the next element, removing all semblance of an array. This is what a linked list is: a series of nodes where each one points to the next one in memory, and each node contains a piece of data. We often depict linked lists as follows:

![Figure 1: Basic illustration of a linked list](image)

Linked lists can be made as long as we want without declaring an initial size first. All we have to do is attach a node after the last one, and ensure that the previously last element of the list now points to the new one.
In class, some students asked for a comparison of vectors and linked lists. For starters, we want to understand here how something like vectors works under the surface. Someone programmed vectors as an abstract data type, by using the types of primitives we are learning about in this class.

What a vector really does is allocate a dynamic array and keep track of its size. When the size is not sufficient any more, an array with larger size will be allocated. (Typically, implementations will double the size, but this can often be changed by passing parameters into the vector at creation.) Next, all data will be copied from the old array to the new one. This copying could slow down the use of the data structures.

More importantly, vectors provide a functionality that we may or may not need: namely, accessing elements by an index. Linked lists don’t really allow that. On the other hand, linked lists are tailored towards appending elements cheaply, and traversing all elements. Like with practically any question about “Which data structure should I use?”, the answer is “It depends”. Namely, it depends on what types of operations you will frequently need to use.

Finally, the main “real” reason to study linked lists is that understanding them thoroughly practices dynamic memory and recursion, and prepares you for some of the more complex data structures we will see later in the class.

Some analogies to keep in mind for linked lists are the following:

- A treasure hunt for children. The parents provide a clue for the first treasure. When the children figure out the clue, they go there, and find a treasure, along with a note that has the next clue. Following that clue, they get to the second treasure, where they find another note with a clue to the next treasure, and so on. The clues play the role of pointers.

- The game of “Assassin,” in which each participant gets the name of another participant to “kill” in a an agreed-upon way (shooting with a nerf gun, touching with a plastic spoon, ec.). When one participant is “killed,” his killer inherits his next assignment, and the winner is the last survivor. Again, the assignments can be thought of to form a linked list (except the tail points back at the head).

- There are several other analogies of sequential events, like dominoes, train cars, and others. What’s nice about the two examples example is the explicit nature of the “pointers”.

## 2 Implementing linked lists

Each node/element in the linked lists contains data (such as an `int`, `string`, etc.) as well as a pointer to the next node/element of the same type. Here, we will build a linked list of integers — it will be pretty obvious how to alter this for other types. In order to keep track of these two elements, we create a struct which we call `IntListElement`. Just to practice the use of the `class` keyword, we will make it a class with `public` data fields. This is actually not good coding practice, and we will return to this issue in two lectures, when we learn more explicitly about classes.

```cpp
class IntListElement {
public:
    int data;
    IntListElement *next;
};
```

Every `IntListElement` has an `int` — a piece of data in the linked list — and a pointer `next` to another node. The first node will have a pointer to the second node, and so on. For the last node, we need a way to make sure to remember that there’s no node after it. The most common way is to have its `next` pointer go to `NULL`, but some people also have it link back to itself instead.

Note that, in order to access the first element of the list at all, we need a `head` pointer to the first `IntListElement`. (If we lose track of this, the rest of the list can no longer be accessed.) Since elements are usually appended at the end, it is also a good idea to maintain a `tail` pointer to the last element of the list. That way, we don’t have to traverse the entire list every time we want to add something.
2.1 Linked list operations

At a minimum, we want to be able to add elements to our list, remove them, and traverse the entire list. Here is how to implement those.

**Traversal:** Unlike arrays, linked lists do not supply functionality to directly access the \( i \)th element. Instead, we start from the first node and then visit the next node repeatedly. To do so, we declare a variable `IntListElement *p` that will keep track of the current node we are looking at. `p` starts out as the pointer to the `head` element. Then in each iteration of a `for` loop, we update it to its next pointer. So the code looks like this:

```c
for (IntListElement *p = head; p != NULL; p = p->next)
    { // Do something with p, such as print or read its data }
```

**Addition:** We take our input data item and create a new `IntListElement` from it. This element will typically be appended to the end of the list, so its `next` pointer will be set to `NULL`. (We could also add it at the beginning of the list, which would change the implementation below.) In order to append it at the end of the list, we first need to find the last element `tail`, then set its `next` pointer to our new element. That’s why we keep track of the `tail` of the list.

We need a special case for a previously empty list, as then, we also have to set the `head` pointer which was previously `NULL`. In summary, the code for adding a new element looks as follows:

```c
void add (int n)
{
    IntListElement *newElement = new IntListElement;
    newElement.data = n;
    newElement.next = NULL;
    if (head == NULL) head = tail = newElement;
    else { tail->next = newElement; tail = newElement; }
}
```

**Removal:** If we are given a pointer `IntListElement *toRemove` to an element of the list we’d like to remove, we’ll eventually have the command `delete toRemove;` But before that, we also need to make sure that the link structure of the list stays intact. To do so, we need a pointer `prev` to the element right before `toRemove` in the list, so that we may set `prev->next = toRemove->next;` One way to get this pointer (if it exists — otherwise, `toRemove` itself must be the `head` of the list) is to start from the beginning of the list and scan through until we find the node `p` with `p->next = toRemove`. But that would take a long time and be cumbersome.

The better solution is to store in each `IntListElement` not only a pointer to the next element in the list, but also to the previous element. The result is called a *doubly linked list*, and unless you have a strong reason to prefer a singly linked list (such as a job interview specifying it), you should normally make your linked list doubly linked. In the definition of `IntListElement`, we add the line `IntListElement *prev;` In the function for adding an element, we set `newElement->prev = NULL` in the first case, and `newElement->prev = tail` in the second case.

For removing an element, we can now write something like:

```c
void remove (IntListElement *toRemove)
{
    toRemove->prev->next = toRemove->next;
    toRemove->next->prev = toRemove->prev;
    delete toRemove;
}
```
This sets the next pointer of the preceding element to the next pointer of the element to be deleted, effectively unlinking it. Similarly for the second line with the prev pointers. While this looks good at first sight, we have to be more careful when the element we want to remove is the head or tail of the list (or both, for a list of a single element). Then, toRemove->prev or toRemove->next could be NULL, which means we can’t change their pointers. However, in those cases, we don’t need to update the corresponding pointers, so the actual implementation looks as follows:

```cpp
void remove (IntListElement *toRemove)
{
    if (toRemove != head)
        toRemove->prev->next = toRemove->next;
    else head = toRemove->next;
    if (toRemove != tail)
        toRemove->next->prev = toRemove->prev;
    else tail = toRemove->prev;
    delete toRemove;
}
```

As we saw, the real reason for having doubly linked lists is that they make deletions much easier and cleaner. (For your own amusement, you should perhaps also implement deletion in a singly-linked list, and measure how much slower it is.) A side benefit is that a doubly linked list can be easily traversed back to front. Sometimes, that’s listed as a reason for having a double linked list, but I think that that’s a red herring: first, traversing a linked list back to front is not hard even for a singly-linked list, if you use recursion (see Homework 2). And second, it’s not a functionality that is often needed.

Once we really wrap our heads around the idea of having a pointer (or two) to other elements in our IntListElement, we may ask ourselves why not have more pointers to different elements. In fact, that is exactly what we will be doing when we get to more complex data structures such as trees and heaps later on.